Borislav Đorđević
Marko Carić
Dragan Pleskonjić
Nemanja Maček

UNIX ARHITEKTURA

Autori:	dr Borislav Đorđević, Marko Carić
	mr Dragan Pleskonjić, Nemanja Maček
Recenzenti:	mr Verica Vasiljević, dr Slobodan Obradović
Izdavač:	Viša elektrotehnička škola u Beogradu
Za izdavača:	dr Dragoljub Martinović
Tehnička obrada:	Borislav Đorđević, Marko Carić,
	Dragan Pleskonjić, Nemanja Maček
Dizajn korica:	Katarina Carić
Štampa:	MST Gajić, Beograd
	štampano u 200 primeraka

Predgovor

Od svoje pojave 1969. godine, UNIX je postao krajnje popularan operativni sistem. On funkcioniše na velikom broju računara sa različitom procesorskom snagom, od mikroprocesora do mainframe mašina.

UNIX je visoko performansni operativni sistem, sa izraženom stabilnošću, pogodan za izvršavanje velikog broja različitih aplikacija. Ono što takođe karakteriše UNIX je permanentno trajanje u vremenu od preko 30 godina i rasprostranjenost na brojnim računarskim arhitekturama. Većina velikih svetskih proizvođača računara razvija sopstvenu varijantu UNIX operativnog sistema, ali većina tih UNIX sistema, poput SCO, HP-UX, IBM AIX i Sun Solaris, je komercijalna, što znači da korisnik mora da plati licencu za korišćenje, a izvorni kod operativnog nije raspoloživ. Alternativa kvalitetnim, ali relativno skupim UNIX operativnim sistemima je Linux. Linux postaje sve popularniji operativni sistem, koji zadržava većinu dobrih osobina UNIX sistema, a dodatno se odlikuje raspoloživim izvornim kodom i praktično besplatnim korišćenjem. Zbog toga danas većina proizvođača računara, osim soptvene komercijalne verzije UNIX sistema, nudi i svoju varijantu Linux sistema. Linux se najčešće koristi u manjim ili srednjim klasama servera, a jedna od oblasti primene, u kojoj veliki broj korisnika podržava i promoviše Linux kao bazični server, su Internet servisi, tipa web servera, mail servera itd.

Ova knjiga se prvenstveno odnosi na UNIX Sistem V operativni sistem i namenjena osnovnom kursu iz predmeta Operativni Sistemi 2 na višoj elektrotehničkoj školi u Beogradu. Knjiga može poslužiti kao koristan izvor informacija, svakom čitaocu koji se ozbiljnije bavi UNIX operativnim sistemom.

Ukratko o svakom poglavlju

Knjiga sadrži dve celine: organizacija sistema datoteka i organizacija procesa na UNIX operativnom sistemu. Knjiga je podeljena u 11 glava, čiji ćemo kratak sadržaj prezentovati u nastavku:

U prvoj glavi ”Uvodna razmatranja o UNIX operativnom sistemu” dat je istorijat i generalni istorijski pregled UNIX sistema i kratak opis najznačajnijih UNIX verzija. Takođe su opisane opšte karakteristike UNIX operativnog sistema.

U drugoj glavi ”Uvod u kernel i kernelski keš” opisane su fundamentalne karakteristike UNIX kernela, kao i osobenosti Linux kernela. U trećem delu glave, demonstriran je baferski keš mehanizam, sa opisom algoritama i demonstracijom na praktičnim primerima.

U trećoj glavi ”Interna reprezentacija datoteka” najpre je opisana interna reprezentacija datoteka sa opisom disk i memorijske inode strukture. Potom su dati fundamentalni algoritmi niskog nivoa, kao što su iget, iput, bmap i namei. U trećem delu se opisuju algoritmi za alokaciju inode struktura ialloc i ifree, kao i algoritmi za alokaciju blokova na disku, alloc i free.

U četvrtoj glavi "Osnovni sistemski pozivi u radu sa datotekama" prikazani su osnovni sistemski pozivi za datoteke, kao što su open, read, write, lseek. Potom se demonstriraju sistemski pozivi za kreiranje specijalnih datoteka i sistemski pozivi za rad sa direktorijuma. Većina sistemskih poziva demonstrirana je u C primerima.

U petoj glavi "Napredni sistemski pozivi u radu sa datotekama" prikazani su najpre specijalni sistemski pozivi pipe i dup. Potom su demonstrirani sistemski pozivi za aktiviranje i deaktiviranje sistema datoteka, mount i umount. Na kraju se demostriraju sistemski pozivi link i unlink i diskutuje se konzistencija sistema datoteka. Većina sistemskih poziva je demonstrirana u C primerima.

U šestoj glavi "Struktura UNIX Procesa" prikazana je struktura UNIX procesa, puni dijagram stanja procesa i tranzicije stanja, pojam regiona i osnovni memorijski algoritmi za rad sa memorijskim regionima, alloceg, attachreg, detachreg, loadreg, dupreg i freereg.

U sedmoj glavi "Kontrola procesa" prikazani su fundamentalni algoritmi za procese i to prvo pet fundamentalnih algoritma fork, exit, exec, xalloc i wait. Zatim se opisuju UNIX signali. Na kraju se demostriraju algortimi za uspavljivanje i buđenje sleep i wakeup. Većina sistemskih poziva demonstrirana je u C primerima.

U osmoj glavi "Raspoređivanje procesa i vremenske funkcije" najpre je detaljno opisan algoritam za raspoređivanje procesa na UNIX operativnom sistemu, a kao poseban slučaj je prikazan FFS algoritam. Zatim su prikazani sistemski pozivi za vremenske funkcije, kao što su stime, time, times i alarm, kao i opis algoritma za časovnik (clock). Većina sistemskih poziva demonstrirana je u C primerima.

U devetoj glavi "Upravljanje Memorijom" prikazani su dve fundamentalne memorijske tehnike, swaping i straničenje po zahtevu (Demand Paging). U okviru swaping tehnike demostriran je algoritam malloc i čuveni proces swapper. U okviru straničenja, prvo se opisuju strukture podataka za straničenje, zatim dve rutine za obradu greške u straničenju VFH i PHF i čuveni PS proces (Page stealer).

U desetoj glavi "Ulazno/Izlazni Sistem" prikazani su najpre, generalna struktura drajvera na UNIX operativnom sistemu i dve prekidačke tabele, blok i karakter. Zatim se opisuju osobenosti disk drajvera. Na kraju se prikazuje struktura terminal drajvera i streams mehanizam koji služi za realizaciju savremenih drajvera.

U jedanestoj glavi "Interprocesna komunikacija" prikazan je najpre mehanizam za praćenje programa (ptrace). Zatim se opisuju UNIX IPC mehanizmi, kao što su IPC sistem poruka, IPC preko deljive memorije, IPC preko semafora i UNIX soket mehanizam. Većina sistemskih poziva demonstrirana je u C primerima.

Zahvalnost

Zahvaljujemo se svima koji su učestvovali ili na bilo koji način pomogli u realizaciji ove knjige.

Autori

Sadržaj

Uvodna razmatranja o UNIX operativnom sistemu

1.1. Generalni istorijski pregled Unix sistema

Prva verzija operativnog sistema UNIX, nastala je 1969. godine. Razvio ju je naučnik iz istraživačke grupe Bellovih laboratorija, Ken Thompson i ta verzija se koristila na računaru PDP-7 (računar koji nije imao neku naročitu primenu u to vreme). Ubrzo, njemu se pridružio Dennis Ritchie. Thompson, Ritchie i drugi članovi istraživačke grupe napravili su prvu verziju UNIX operativnog sistema.

Ritchie je predhodno radio na MULTICS[1] projektu, koji je imao veliki uticaj na novonastali operativni sistem. Čak je i ime UNIX igrom reči nastao od MULTICS. Osnovna organizacija sistema datoteka (file system), ideja da komandni interpretator bude korisnički proces, korišćenje zasebnog procesa za svaku komandu, originalni način editovanja karaktera (# da se izbriše poslednji karakter a @ za brisanje celog reda) i brojne druge posebnosti, dolaze dirketno od MULTICS operativnog sistema. Takođe su korišćene ideje iz nekih drugih operativnih sistema, kao npr. MIT CTSS i XDS-940.

Ritchie i Thompson su godinama, bez publiciteta, radili na UNIX operativnom sistemu. Njihov rad na UNIX operativnom sistemu na prvoj verziji omogućio je prelazak na računar PDP-11/20 u drugoj verziji. U trećoj verziji, ponovo su pisali većinu koda, ali ovog puta na programskom jeziku C, a ne kao pre u asembleru. Programski jezik C je razvijen u Bell-ovim laboratorijama u cilju razvoja UNIX operativnog sistema. UNIX je prebačen na veće računare, modele PDP-11, kao što su 11/45 i 11/70. Multiprogramiranje i ostale prednosti su dodate. Kada je sistem ponovo napisan u C jeziku i prebačen na sisteme (kao npr. 11/45) koji su imali hardversku podršku za multiprogramiranje, dodate su i ostale prednosti uključujući sâmo multiprogramiranje.

Kako se UNIX razvijao, postao je široko upotrebljivan u Bell-ovim laboratorijama i postepeno se preneo na nekoliko američkih univerziteta. 1976. godine, izbačena je verzija 6, koja je bila prva verzija koja je puštena van Bell-ovih laboratorija. (Broj verzije za prve UNIX sisteme odgovara broju UNIX-ovog programerskog uputstva – UNIX Programmer's Manual, koji je bio aktuelan u trenutku distribucije izdanja; kod UNIX-a su i uputstva razvijana nezavisno).

1 Tri kompanije su 1965. godine razvijale operativni sistem Multics: Bell, General Electric Company i Project MAC. Cilj je bio da to bude moćan operativni sistem za veliki broj korisnika. Iz tog projekta, Mulitics je nastao 1969. godine i to najpre za procesor PDP-7. Potom, 1971. godine, realizovan je i za PDP-11, procesor sa skromnim hardverskim mogućnostima (16K za operativni sistem, 8K za korisničke programe, disk veličine 512K, file limit 64K). Ken Thompson je napisao najpre Fortran prevodilac (compiler), ali umesto da bude realizovan na interpreterskom programskom jeziku koji se zvao B, Dennis Ritchie je napisao novi programski jezik koga je nazvao C. C je bio sposoban da realizuje mašinski kôd, deklaraciju tipova podataka i definiciju struktura podataka. 1973. godine, UNIX je prepisan na C. Kasnije se UNIX razvijao u sledećim oblicima: prvo UNIX System III a potom UNIX System V sa kojim je paralelna generacija je BSD UNIX.

1978. godine, izbačena je verzija 7, koja je radila na PDP-11/70 i na Interdati 8/32, i to je predak većine modernih UNIX sistema. Tačnije, verzija 7 je uskoro puštena na VAX računarima i ostalim PDP-11 modelima. Verzija za VAX nosila je oznaku 32V. Posle toga, nastavljen je rad na UNIX projektu.

Posle distribucije verzije 7, u 1978. godini, UNIX Support Group (USG) je preuzela kontrolu i odgovornost za UNIX. UNIX sad postaje proizvod, a ne samo alat za istraživanje. Ipak, istraživačka grupa nastavlja da pravi svoje verzije UNIX-a za svoje unutrašnje potrebe. Nastaje verzija 8, koja ima mehanizam koji su nazvali stream I/O system, koji omogućava fleksibilnu konfiguraciju kernelovih IPC modula. Ova verzija sadrži i RFS (remote file system) koji je sličan sistemu SUN NFS. Posle toga dolaze verzije 9 i 10, pri čemu je verzija 10 poslednja verzija koja pripada isključivo Bell-ovim laboratorijama.

UNIX System III

USG je uglavnom pružao podršku za UNIX unutar Bell laboratorija, tj. za UNIX poznat pod nazivom AT&T UNIX. Prva spoljna distribucija USG grupe bio je UNIX Sistem III, koji se pojavljuje 1982 godine. Sistemu III su pridružene mogućnosti iz verzija 7 i 32V, i još nekih verzija UNIX-a koje su se nezavisno razvijale. U Sistem III su uključene mogućnosti UNIX/RT, real-time UNIX sistema, kao i velikog dela PWB softverskih paketa (Programmer's Work Bench).

UNIX System V

1983 godine, USG izbacuje UNIX System V, koji je uglavnom nastao iz UNIX Sistem III. Izlazak raznih Bellovih kompanija iz AT&T-a, naterali su AT&T da agresivno marketinški promoviše Sistem V. Od USG-a nastaje USDL, UNIX System Development Laboratory, koji izbacuje verziju Sistema V, drugo izdanje (V.2), u 1984. godini. A verzija Sistema V, V.2.4 dodaje novu implementaciju virtualne memorije sa copy-on-write paging tehnikom i deljivom memorijom. UDSL postaje ATTIS (AT&T Information Systems), koji 1987. godine izbacuje V.3 verziju Sistema V. Verzija V.3 prilagođava mehanizam „stream I/O system iz verzije 8", koji je sad dostupan kao STREAMS struktura. UNIX System V takođe sadrži udaljeni sistem datoteka RFS (remote file system).

BSD UNIX

Mala veličina, modularnost i čist dizajn ranih UNIX sistema, dovodi do prihvatanja UNIX operativnog sistema na mnogim drugim naučnim i računarskim ustanovama, kao što su Rand, BBN, Univerzitet Illinois, Harvard, Purdue i DEC. Najveći uticaj na UNIX, među njima, imao je Kalifornijski Univerzitet u Berkliju.

Prvi Berklijski VAX UNIX nastaje 1978. godine kao skup sledećih mogućnosti: virtualne memorije, straničenja po zahtevu, i zamene stranica kao kod 32V; tvorci su bili Bill Joy i Ozalp Babaoglu, i tako nastaje 3BSD UNIX. Ovo je bila prva verzija UNIX operativnog sistema koji je imao ove mogućnosti. Veliki prostor virtualne memorije

omogućuje razvoj veoma velikih programa, kao što je Berklijski Franz LISP. Kvalitetno upravljanje memorijom, na BSD UNUX operativnom sistemu, ubedilo je DARPA agenciju (Defense Advanced Research Projecr Agency), da finansira Berkli, da razvija UNIX sistem za vladine institucije; verzija 4BSD UNIX je rezultat toga.

U razvoju 4BSD sistema uticali su brojni poznavaoci UNIX operativnog sistema i računarskih mreža. Jedan od ciljeva ovog rada bio je stvaranje podrške za DARPA Internet protokol (TCP/IP). Ova podrška je urađena sveobuhvatno. Omogućeno je da 4.2BSD komunicira između različitih tipova mreža, uključujući LAN mreže (kao Ethernet i token ring) i WAN mreže (kao NSFNET). Ta implementacija je glavni razlog za današnju popularnost ovih protokola. TCP/IP je korišćen kao osnova za neke druge implementacije drugih izdanja UNIX operativnog sistema, pa čak i za druge operativne sisteme. Omogućio je da Internet poraste sa 60 povezanih mreža 1984. na više od 8000 mreža, a po proceni 10 miliona korisnika u 1993. godini.

Pored toga, Berkli koristi mnoge mogućnosti savremenih operativnih sistema da poboljša dizajn i implementaciju BSD UNIX operativnog sistema . Mnoge funkcije za line-editovanje preuzete su od TENEX (TOPS-20) operativnog sistema, ubacivanjem novog drajvera. Novi korisnički interfejs (C Shell), novi text editor (ex-/vi), kompajleri za Pascal i LISP i mnogi novi sistemski programi napisani su na Berkliju. U verziji 4.2BSD, neke funkcije su preuzete od VMS operativnog sistema.

UNIX softver kreiran na Berkliju dobija ime Berkeley Software Distributions. Obično se kaže da Berklijev VAX UNIX prate verzije 3BSD i 4BSD, iako je zapravo postojalo nekoliko specifičnih izdanja, među kojima su bila glavna 4.1BSD i 4.2BSD. Oznake 2.xBSD sistema koriste se za sisteme PDP-11, a oznake 4.xBSD za VAX distribucije Berklijevog UNIX-a. 4.2BSD, izbačen 1983. godine, bio je kulminacija originalnog Berklijevog DARPA UNIX projekta. 2.9BSD za PDP-11 sisteme, ekvivalentan je verziji 4.2BSD za VAX.

U 1986. godini izašao je 4.3BSD. Zahvaljujući velikoj sličnosti sa verzijom 4.2BSD, njegovo uputstvo je jasnije objašnjavalo 4.2BSD, od samog 4.2BSD uputstva. Na 4.3BSD nisu radili na nekim većim unutrašnjim promenama, već na ispravljanju grešaka i na unapređenju performansi. Dodate su neke nove mogućnosti, uključujući podršku za Xerox Network System protokole.

Sledeća verzija nosila je naziv 4.3BSD Tahoe, a puštena je 1988. godine. Obuhvatala je brojne nove mogućnosti, kao npr. poboljšanu kontrolu mrežnog zagušenja, i povećanje TCP/IP performansi. Takođe, konfiguracije diskova su odvojene od drajvera uređaja, pa su se čitale direktno sa diskova. Dodata je i podrška za vremenske zone. 4.3BSD Tahoe je ustvari razvijen za CCI Tahoe System (Computer Console, Inc., Power 5 computer) a ne za uobičajenu VAX bazu. Verzija 2.10.1BSD za računar PDP-11 ekvivalentna je verziji 4.3BSD Tahoe, koju distribuira USENIX Asocijacija, koja je i izdala uputstva za 4.3BSD.

U verziji 4.32BSD Renoe, dodata je implementacija ISO/OSI mrežnog modela.

Poslednje Berklijevo izdanje 4.4BSD završeno je juna 1993. Ono uključuje podršku za novi X.25 mrežni protokol i za POSIX standard. Takođe, ima radikalno promenjenu organizaciju sistema datoteka. Dodat je novi interfejs prema virtuelnoj memoriji i podrška

za stek-bazirani sistem datoteka, koji omogućuje da sistem datoteka bude realizovan u slojevima, što opet omogućava lako ubacivanje novih mogućnosti. Dodata je i implementacija NFS, kao i novi sistem datoteka sa dnevnikom transakcija. Još nekoliko promena je dodato, kao poboljšana sigurnost i poboljšana struktura kernela. Sa ovim izdanjem Berkli obustavlja rad na ovom projektu.

4BSD operativni sistem je bio izbor za VAX računare od njegovog prvog izdanja (1979) do pojave Ultrix operativnog sistema, DEC-ove BSD implementacije. 4BSD je i dalje najbolji izbor za mnoge istraživačke i mrežne organizacije. Mnoge organizacije su kupile 32V licencu i naručile 4BSD od Berklija.

Druge varijate UNIX operativnog sistema

Trenutni skup verzija UNIX operativnih sistema nije ograničen samo na one iz Bell-ovih laboratorija (čiji je vlasnik Lucent Technology) i na one sa univerziteta iz Berklija. Sun Microsystems je pomogao u popularizaciji BSD verzije UNIX operativnog sistema, tako što je isporučivao svoje radne stanice s njim. UNIX je rastao u populaciji, i bio je instaliran na mnogo različitih računara i računarskih sistema. Kreirana je široka lepeza UNIX bazirnih operativih sistema. Na radnim stanicama, DEC ima svoj UNIX (Ultrix), ali takođe poseduje i njegovog nastavljača OSF/1, koji je takođe nastao od UNIX operativnog sistema. Microsoft je preradio UNIX za Intel 8080 familiju i nazvao ga XENIX, a i Windows NT operativni sistem je nastao pod jakim uticajem UNIX operativnog sistema. IBM koristi UNIX (AIX) na svojim PC stanicama i na serverima. Praktično, UNIX je dostupan na gotovo svim vrstama računara za opštu upotrebu, ima ga na personalnim računarima, radnim stanicama, miniračunarima, serverima i super-računarima, od Apple-Mekintoša do Cray sistema. Zbog široke raspoloživosti, koristi se na raznim mestima, počev od akademskih i vojnih organizacija, do fabrika. Većina ovih sistema bazirana je na Verziji 7, Sistemu III, 4.2BSD-u ili Sistemu V.

UNIX standardizacija

Široka popularnost UNIX operativnog sistema dovela je do toga da UNIX bude najrašireniji operativni sistem i da korisnici očekuju da UNIX okruženje bude nezavisno u odnosu na specifičnost hardvera. Ali veliki broj njegovih implementacija doveo je do toga da postoje razne varijacije u programiranju i korisničkom interfejsu, koji je distribuiran od izdavača. Za pravu nezavisnost, neophodno je da oni koji razvijaju programe obezbede konzistentan interfejs. Takav interfejs bi omogućio da sve “UNIX” aplikacije rade na svim UNIX sistemima, što svakako nije trenutna situacija. Ovaj zahtev postaje veoma važan budući da je UNIX postao omiljena platforma za razvoj programa, počevši od baza podataka, preko grafičkih programa, pa sve do mrežnih aplikacija. Zato je tržište zahtevalo da se postavi standard za UNIX.

Postoje nekoliko projekata standardizacije koji su u toku, počevši od "/usr/group 1984 Standard" koga je finansirala grupa pod nazivom UniForum industry users’ group. Od tada, mnoga tela za standardizaciju bave se tim problemom, uključujući IEEE i ISO

(POSIX standard). Grupa pod imenom X/Open Group je internacionalni konzorcijum. Ona je napravila XPG3, koji je tipično okruženje za aplikacije (Common Application Enviroment). Nažalost, XPG3 je baziran na skici ANSI C standarda, a ne na konačnoj specifikaciji i zato je morao da se prerađuje. XPG4 je završen 1993 godine. U 1989. godini, telo za ANSI standardizaciju je napravilo ANSI C standard, kome se proizvođači lako prilagođavaju. Dok se rad na ovim projektima nastavlja, razna izdanja UNIX operativnog sistema nastaju uz postojanje samo jednog programerskog interfejsa i zato UNIX postaje sve popularniji. Praktično postoje dva različita skupa moćnih UNIX proizvođača koji rade na ovom problemu. UNIX International (UI) i Open Software Foundation (OSF), jedinstveni su u odluci da prate POSIX standard. Nedavno, mnogi proizvođači iz ove dve grupe saglasni su oko dalje standardizacije (COSE sporazum). Ona bi trebala da obuhvati Motif okruženje prozora, i ONC+ (koji uključuje Sun RPC i NFS), kao i DCE mrežne funkcije (koji uključuju AFS i RPC paket).

1989. godine, AT&T promoviše ATTIS grupu u USO (UNIX Software Organization), koja isporučuje prvi spojeni UNIX, System V, Release 4. Ovaj sistem kombinuje mogućnosti Sistema V, 4.3BSD-a i Sun-ovog SunOS operativnog sistema, uključuje duga imena za datoteke, Berkli sistem datoteka, upravljanje za virtualnu memoriju, simboličke linkove, višekorisničke grupe, kontrolu poslova, i sigurne signale. Takođe, rađen je po POSIX standardu, POSIX.1. Nakon toga USO pravi SVR4 i postaje nezavisna AT&T filijala, pod imenom Unix System Laboratories (USL), a 1993. godine, nju kupuje Novell, Inc.

Akademski UNIX

UNIX sistem je porastao od ličnog projekta dvojice saradnika Bell laboratorija, do operativnog sistema koji je definisan međunarodnim standardizacionim telima. Ipak, ovaj sistem je i dalje interesantan akademskim institucijama. Verujemo da je UNIX postao i da će ostati važan deo teorije i prakse o operativnim sistemima. UNIX je odlično vozilo za akademske studije. Na primer, Tunis operativni sistem, Xinu operativni sistem, i Minix operativni sistem, bazirani su na konceptima UNIX operativnog sistema, ali su razvijani eksplicitno za učionice. Postoji obilje aktuelnih istraživačkih radova na UNIX operativnom sistemu, uključujući Mach, Chorus, Comandos i Roisin. Originalni tvorci, Riči i Tompson, nagrađeni su 1983. od Asociation for Computing Machinery, nagradom Turing, za njihov rad na razvoju UNIX operativnog sistema.

FreeBSD

Free BSD je Intelova BSD verzija UNIX operativnog sistema. Sistem je korišćen jer implementira razne interesantne koncepte operativnih sistema, kao što je zahtev za straničenjem sa klaster tehnologijom i umrežavanje. FreeBSD projekat počeo je 1993. godine. 386BSD je nastao od 4.3BSD-Lite (Net/2) i originalno je izbačen u junu 1992. godine od strane Williama Jolitza. FreeBSD (David Greenman) 1.0 je izbačen decembra 1993. godine. FreeBSD 1.1 pušten je maja 1994. godine i obe verzije bile su bazirane na 4.3BSD-Lite. Zbog nekih ugovora između UCB-a i Novell-a, bilo je potrebno je da se kôd iz 4.3BSD više ne koristi, tako da je konačan 4.3BSD-Lite izbačen jula 1994. (FreeBSD

1.1.5.1).

FreeBSD je ponovo napravljen, baziran na kôdu verzije 4.4BSD-Lite, koja je nepotpuna, i izdat je novembra 1994 godine, pod oznakom FreeBSD 2.0. Kasnija izdanja su: 2.0.2 u junu 1995, 2.1.5 i avgustu 1996, 2.1.7 u februaru 1997, 2.2.1 u aprilu 1997, 2.2.8 u novembru 1998, 3.0 u oktobru 1998, 3.1 u februaru 1999, 3.2 u maju 1999, 3.3 u septembru 1999, 3.4 u decembru 1999, 3.5 u junu 2000, 4.0 u martu 2000, 4.1 u julu 2000 i 4.2 u novembru 2000.

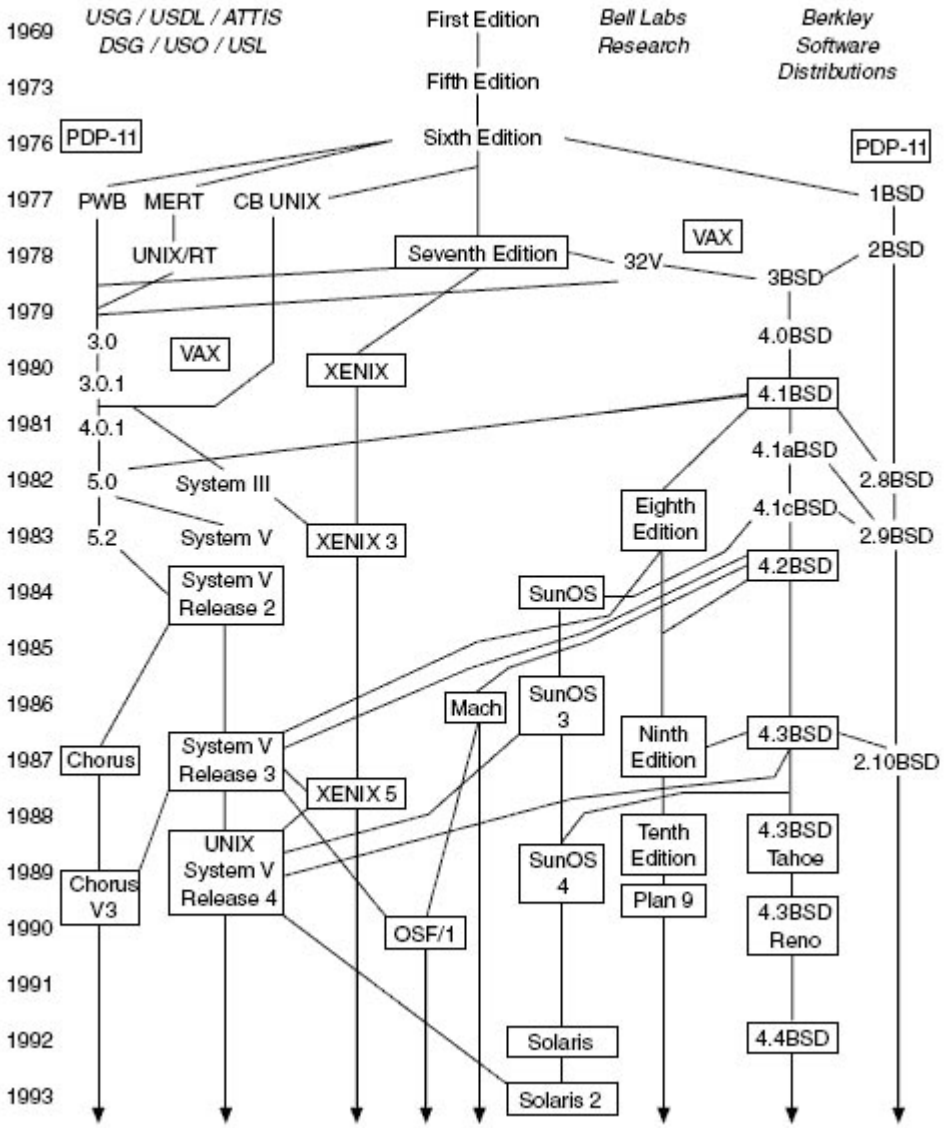
Cilj celog FreeBSD projekta je stvaranje softverskog alata koji bi mogao da se koristi za svaku svrhu bez bilo kakvih obaveza. Ideja je da se kôd u potpunosti iskoristi i da pruži najveću moguću dobit. Osnova je ista kao ona opisana u McKusick et al. [1984], sa dodatkom povezane virtualne memorije, fajl-sistemskog baferskog keša, kernelskih upita i softverskih ažuriranja za sistem datoteka. Trenutno, FreeBSD radi prvenstveno na Intelovim platformama, iako su Alpha platforme podržane. U toku je rad na tome da se podrže i druge procesorske platforme.

Na slici 1.1 dat je rezime veza i odnosa između različitih verzija UNIX operativnih sistema.

1.2. Opšte karakteristike UNIX operativnog sistema

Više razloga je uticalo na popularnost UNIX operativnog sistema:

- Sistem je napisan na visokom programskom jeziku PL (high level PL), što mu omogućava da bude lakši za čitanje, razumevanje, modifikaciju i prenos na druge računarske konfiguracije. Ritchie je procenio da je na prvom UNIX operativnom sistemu, programski jezik C napravio povećanje kôda i usporenje od 20 do 40% u odnosu na asemblersku realizaciju, ali su prednosti višeg programskog jezika kasnije došle do izražaja.
- UNIX poseduje jednostavan korisnički interfejs koji korisnicima omogućava sve što žele
- UNIX obezbeđuje primitive koje omogućavaju kompleksnijim programima da se realizuju iz jednostavnijih programa
- UNIX koristi hijerarhijski sistem datoteka (FS, file system, sistem datoteka) koji omogućava lako održavanje i efikasno korišćenje i realizaciju
- UNIX koristi konzistentan format za datoteke i nizove bajtova, što omogućava lakše pisanje programa
- UNIX obezbeđuje jednostavan konzistetan intefejs za periferijske uređaje



Slika 1.1. *Istorijski pregled operativnog sistema UNIX*

- UNIX je višekorisnički (multi-user), višeprocesni (multi-task) operativni sistem: svaki korisnik može izvršavati više programa istovremeno

- UNIX krije-apstrakuje mašinsku arhitekturu od korisnika, što omogućava lakšu realizaciju programa na različitim hardverskim arhitekturama

Mada su operativni sistem i većina komandi realizovana na C programskom jeziku, UNIX podržava masu drugih jezika kao što su Fortran, Basic, Pascal, Ada, Cobol, Lisp, Prolog itd.

Struktura sistema

UNIX sistem se sastoji od sledećih komponenti:

- Hardver
- Kernel i interfejs sistemskih poziva (SC interfejs)
- Sistemski programi
- cc (C compiler) koji se sastoji od: C pretprocesora, dvoprolaznog prevodioca, asemblera i punioca (link-editor)
- Drugi aplikacioni programi

Korisnička perspektiva

Ova sekcija ukratko opisuje osobine UNIX operativnog sistema na visokom nivou kao što su FS (sistem datoteka), procesi i realizacija blok primitiva kao što je na primer pipe.

Sistem datoteka

UNIX sistem datoteka određen je sledećim osobinama:

- hijerarhijska struktura
- konzistento tretiranje podataka
- mogućnost kreiranja i brisanja datoteka
- mogućnost dinamičkog rasta datoteka
- zaštita datoteka preko prava pristupa
- tretiranje periferijskih uređaja kao datoteka

Sistem datoteka je organizovan kao stablo sa jednim korenskim čvorom-direktorijumom, koji se zove root direktorijum i obeležava sa /. Sve ostalo u stablu čine direktorijumi, regularne i obične datoteke. Ime datoteke uključuje i putanju koja joj određuje mesto u stablu.

Direktorjumi su specijalni nizovi bajtova podataka. Na UNIX-u su to specijalne datoteke, koji sadrže opis datoteka koje se u njemu nalaze.

Uređaji se tretiraju kao datoteke:

- koriste iste komande kao i komande za rad sa datotekama (cp, mv, ls, cat)
- imaju zaštitu i kontrolu pirstupa (rwx) na isti način kao datoteke

Procesi

Proces je jedna instanca programa u izvršavanju. Pod UNIX operativnim sistemom, proces je celina koja je kreirana sistemskim pozivom fork.

Četiri sistemska poziva (SC, system call) karakteristična su za kreiranje procesa pod UNIX-om

- **fork**: kopira adresni prostor od procesa-roditelja za proces dete i svakome dodeljuje pid (process indentifier) i to nulu za dete i nenulu za roditelja.
- **exec**: je sistemski poziv pomoću koga dete proces puni (overlay) odgovarajući program u svoj adresni prostor. Kada se dogodi execl, roditelj i dete nisu više isto. Dete se više ne vraća na početni kôd jer ga je prepisalo sistemskim pozivom execl
- **exit**: proces-dete mora imati sistemski poziv exit, koji znači da je obavilo svoje aktivnosti i da se ukida tj. terminira, oslobađajući resurse
- **wait**: sistemski poziv koji omogućava procesu roditelju da se blokira i čeka da dete obavi svoje aktivnosti i uradi sistemski poziv exit

Navodimo primer upotrebe sistemskih poziva za procese:

```
main (argc, argv)
  int argc;
  char *argv[];
{
  /* predpostavimo dva argumenta: izvorna i ciljna datoteka*/
  if ( fork() == 0) execl("cp", "cp", argv[1],argv[2], 0);
  wait ((int *) 0 );
  printf ("copy done\n");
}
```

UNIX shell i komande

UNIX shell je komandni interpreter, koji dozvoljava tri tipa komandi:

- izvršne binarne datoteke koje sadrže objektni kôd nastao prevođenjem (kompilacijom) izvornog koda
- shell script datoteke koje nastaju kao programske konstrukcije sastavljene od shell komandnih linija, preciznije od UNIX komandi
- interne komande shell-a koje omogućavaju shell programske konstrukcije tipa if, while ... for, ali i neke komande tipa cd...

Shell radi na principu fork-exec-wait za procese u prvom planu čekajući ih da se završe i to je sinhroni rad. Shell može raditi i asinhrono, tako što stvorene procese postavlja u pozadinu, ne čekajući da se završe. To se postiže sledećom sintaksom:

```
command &
```

Shell nije deo kernela, može se lako modifikovati i promeniti.

Realizacija blok primitiva

UNIX obezbeđuje blok primitive koje omogućavaju korisnicima da pišu male, modularne programe koji se koriste za realizaciju kompleksnijih programa. Jedna tipična blok **primitiva** je **redirekcija ulaza i izlaza**. Svaki proces ima konvencionalno tri datoteke ili tri deskriptora datoteke:

- standardni ulaz (standard input, ulaz za proces, obično tastatura terminala)
- standardni izlaz (standard output, izlaz za proces, obično monitor terminala)
- izlaz za greške (error output, izlaz za greške za proces, obično monitor terminala)

Standarni ulazi i izlazi se mogu promeniti-redirektovati i to obično u datoteku ili iz datoteke.

Tipični primeri su:

- Redirekcija izlaza u datoteku output:

  ```
  ls >output
  ```

  Komanda ls prikazuje na ekranu sadržaj direktorijuma; u ovom slučaju rezultat ide u datoteku output.

- Redirekcija ulaza iz datoteke letter:

  ```
  mail mjb <letter
  ```

  Komanda mail šalje elektronsku poštu korisniku mjb, a u ovom slučaju sadržaj poruke se uzima iz datoteke, umesto da se uzima sa tastature

- Redirekcija ulaza iz datoteke doc1.in, redirekcija izlaza u datoteku doc1.out, redirekcija greške u datoteku errors

  ```
  nroff –mm <doc1.in >doc1.out 2>errors
  ```

  Komada nroff formatira ulazni niz. U ovom slučaju ulaz se uzima iz datoteke doc1.in, obrada umesto na ekran ide u datoteku doc1.out, a svaka potencijalna greška upisuje se u datoteku errors.

Druga blok primitiva je pipe mehanizam, koji obezbeđuje da se niz podataka prosledi između procesa čitaoca i procesa pisca. Proces pisac preusmerava svoj standardni izlaz na pipe datoteku, dok proces čitaoc preusmerava svoj standardni ulaz na pipe datoteku.

Tipičan primer je:

```
grep main a.c b.c c.c | wc -l
```

Komanda grep pretražuje ključnu reč main u tri datoteke, a.c, b.c i c.c, a onda svoje rezultate upisuje u pipe datoteku, umesto na ekran. Komanda wc –l, prebrojava broj linija u ulaznom nizu. Zbog pipe mehanizma, wc –l ne uzima ulaz sa tastarure, već iz pipe datoteke. Ukupan efekat ove dve komande u pipe mehanizmu predstavlja broj linija u kojima se pojavljuje ključna reč main u tri pomenute datoteke.

Servisi operativnog sistema

Koncentrisaćemo se na servise kernela:

- upravljanje procesima koje omogućava njihovo stvaranje, završetak, privremeno zaustavljanje i komunikaciju
- raspoređivanje procesa (CPU scheduling)
- alokacije memorije procesima i razmenu stranica (swapping)
- alokaciju sekundarne memorije (servisi sistema datoteka - FS services)
- dozvoljavanje procesima da kontrolišu periferijske uređaje (periferale, I/O system)

Pretpostavke oko hardvera

Izvršavanje UNIX procesa deli se na dva nivoa ili moda: kernelski (kernel mode) i korisnički (user mode).

Kada proces izvrši sistemski poziv, izvršni mod procesa se menja sa korisničkog moda u kernelski mod, a operativni sistem pokušava da zadovolji korisnički zahtev i vraća status uspešnog događaja ili greške.

Bez obzira da li će korisnički proces obaviti sistemski poziv ili ne, operativni sistem obavlja masu aktivnosti kao što su obrada prekidnih rutina (interrupt handling), CPU raspoređivanje itd. Neke procesorske arhitekture dozvoljavaju više nivoa, ali UNIX se zadovoljava sa gore pomenuta dva: kernelski i korisnički.

Procesi u korisničkom modu mogu pristupati svojim instrukcijama i podacima ali ne i kernelskim. Procesi u kernelskom modu mogu pristupati i korisničkim i kernelskim adresama. Mnoge instrukcije su privilegovane i mogu se izvršavati samo u kernelskom modu.

Prekidi i uzuzeci

Hadverski prekid je mehanizam kojim se označava neki događaj (I/O completion) i omogućava I/O uređajima da asihrono prekinu CPU. Po prijemu prekida, izvrši se poslednja instrukcija koja se obavljala u trenutku prekida, a kernel će sačuvati stanje prekinutog procesa, odrediti uzrok prekida i pozvati prekidnu rutinu. Nakon toga kernel obavlja po pravilu novo CPU raspoređivanje (scheduling). Za vreme obrade prekida, mogući su novi prekidi, ali se to obavlja prioritetno, niži prekidi obično ne prekidaju više.

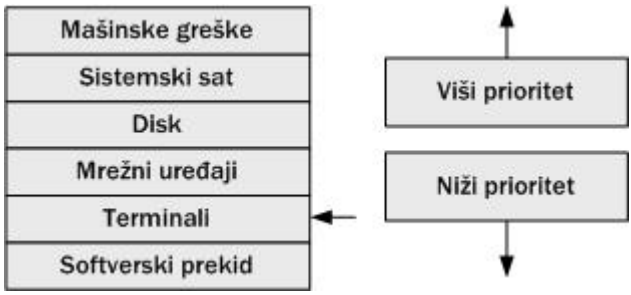
Izuzeci (exception) su neočekivani događaji koje je izazvao sam proces, kao što su:

- greška u straničenju (page fault)
- nelegalan zahtev za memorijom (illegal memory)
- izvršavanje privilegovane instrukcije,
- deljenje sa nulom

Hardverske prekide izazivaju eksterni događaji van procesa, dok izuzetke izaziva sam proces. Hardverski prekidi i izuzeci obrađuju se slično, jer se prekida instrukcija, kernel određuje vrstu prekida i prekidnu rutinu i ako ima smisla obavlja se ponovno pokretanje (restrart) instrukcije koja je napravila izuzetak.

Procesorski nivoi izvršenja (CPU execution level)

Kernel mora da se zaštiti od prekida za vreme nekih kritičnih operacija. Na primer, za vreme ažuriranja povezane liste, disk prekidi moraju biti zabranjeni. Računari koriste skup privilegovanih instrukcija, koje postavljaju CPU nivo izvršenja ili kako se drugačije zove prekidni nivo (CPU execution level). Kada se postavi neki nivo izvršenja, svi prekidi koji su ispod tog nivoa neće biti dozvoljeni kao što se vidi na slici 1.2:



Slika 1.2. *CPU nivoi izvršenja (CPU execution level)*

Na primer, ako je CPU nivo izvršenja na nivou disk prekida, dok se obrađuje disk prekid dozvoljavaju se tajmerski prekid ili prekidi zbog opasnih grešaka u sistemu, ali se ne dozvoljavaju mrežni, softverski ili prekidi sa terminala.

Uvod u upravljanje memorijom

Kernel se rezidentno nalazi u memoriji ili barem jednim delom.

Većina operativnih sistema pa i UNIX koriste princip virtuelne memorije, čija je fundamentalna osobina mapiranje logičkih u fizičke adrese.

Prepostavlja se da su studenti upoznati sa principima za upravljanje memorijom, kao što su:

- straničenje (paging)
- virtuelna memorija
- straničenje po zahtevu (demand paging)

U daljem tekstu, straničenje po zahtevu obeležavaćemo sa DP.

2

Uvod u kernel i kernelski keš

2.1. Uvod u kernel

Na slici 2.1 prikazana je arhitektura operativnog sistema UNIX.

Dva entiteta, datoteke i procesi čine dva centralna koncepta u UNIX sistem modelu. Postoje tri nivoa:

- korisnički nivo
- kernelski nivo
- hardverski nivo

Interfejs sistemskih poziva i bibliotečki interfejs predstavljaju granicu između korisničkog programa i kernela. Sistemski pozivi liče na obične pozive funkcija u C programu, a biblioteke mapiraju ove funkcijske pozive u primitive neophodne da se uđe u operativni sistem. Asemblerski programi, sistemske pozive mogu pozvati direktno, bez poziva sistemskih biblioteka. Programi često koriste pozive za sistemske biblioteke, koji se povezuju sa programom u toku prevođenja (in compile time).

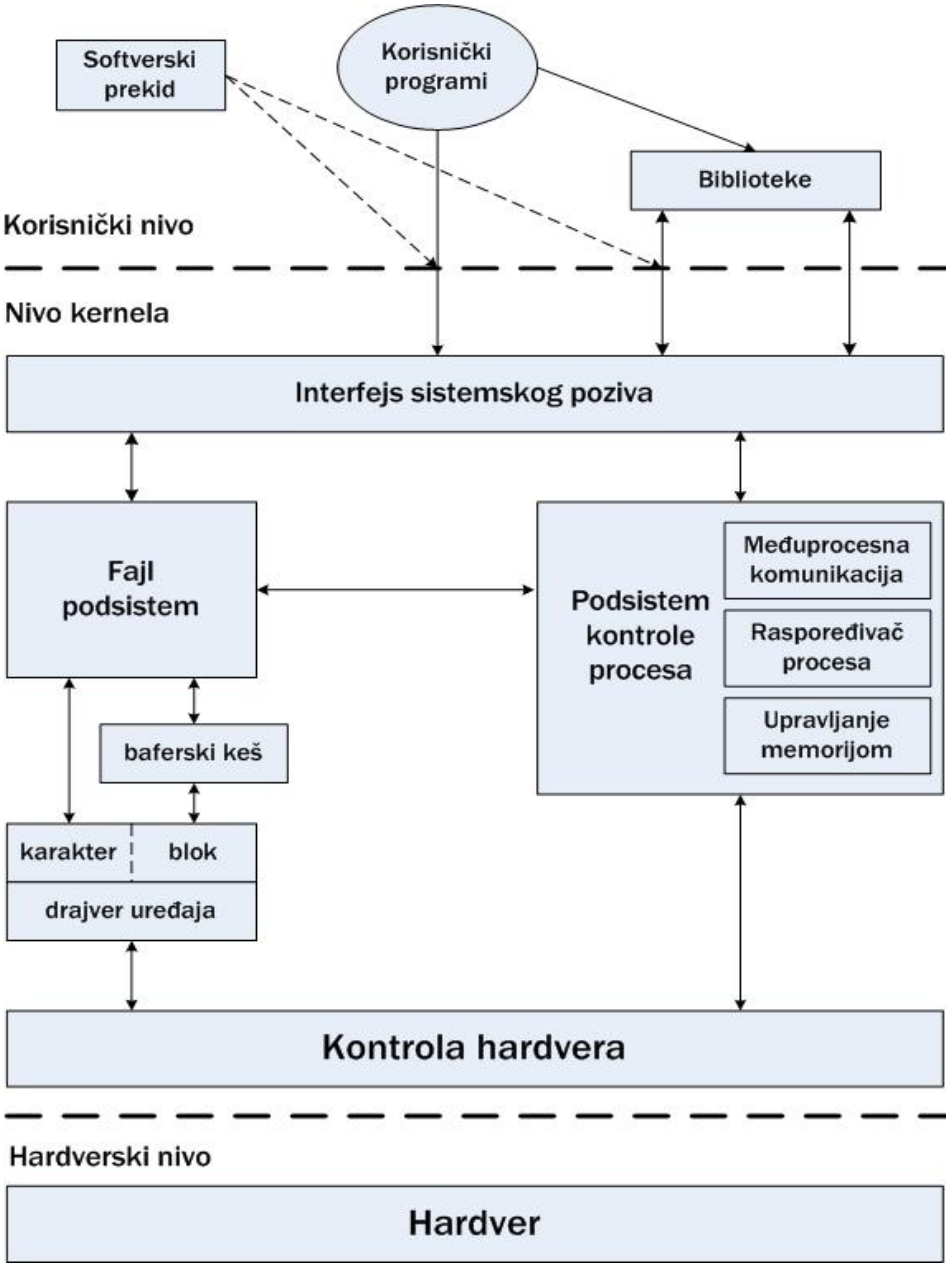
Sistemski pozivi

Sistemske pozive (System Calls, SC) ćemo podeliti u dve grupe: sistemske pozive za sistem datoteka i sistemske pozive za procese.

Sistemski pozivi za sistem datoteka

Sistem datoteka (File System) upravlja datotekama, alocira prostor za datoteke, administrira slobodan prostor, kontroliše pristup datotekama i obezbeđuje korisnicima podatke iz datoteka. Interakcija procesa sa sistemom datoteka odvija se preko skupa sledećih sistemskih poziva:

- **open** (otvara datoteku za čitanje i pisanje)
- **close** (zatvara otvorenu datoteku)
- **read** (čita iz otvorene datoteke)
- **write** (upisuje u otvorenu datoteku)
- **lseek** (pozicionira sledeći pristup u otvorenoj datoteci)
- **stat** (postavlja upit za atribute datoteke)
- **chmod** (podešava prava pristupa za datoteku)
- **chown** (podešava vlasništvo za datoteku)



Slika 2.1. *Arhitektura operativnog sistema UNIX*

Sistem datoteka pristupa podacima na dva načina:

- **Baferisano**. Keš bafer ima specifičnu interakciju sa I/O blok uređajima kroz keš, regulišući protok podataka između kernela i I/O uređaja. Drajveri uređaja (Device drivers) su kernelski moduli koji upravljaju radom I/O uređaja.

- **Neobrađen pristup**. Sistem datoteka može da pristupa blok I/O uređajima direktno, bez baferskog keša i ovaj pristup se naziva neobrađeni pristup (raw access). Na isti način upravlja se svim uređajima koji nisu blok orijentisani.

Sistemski pozivi za procese

Sistem za kontrolu procesa, PCS (Proces Control Subsystem) je odgovoran za sinhronizaciju procesa, interprocesnu komunikaciju (IPC), za upravljanje memorijom i za CPU raspoređivanje. PCS i sistem datoteka komuniciraju kada se program, u cilju izvršavanja, puni iz sistema datoteka u memoriju tj. kada PCS čita izvršnu datoteku u memoriju pre nego što je izvrši.

Sistemski pozivi iz PCS su:

- **fork** (kreiranje novog procesa)
- **exec** (prepisivanje slike programa u izvršni proces)
- **exit** (završetak izvršavanog procesa)
- **wait** (sinhronizacija izvršavanja procesa sa završetkom prethodno kreiranog procesa naredbom fork)
- **brk** (kontrola veličine memorije koja je dodeljena procesu)
- **signal** (kontrola odziva procesa na vanredno izazvane događaje)

PCS sistem se sastoji od tri komponente

- **MMM (Memory Management Module)**. MMM kontroliše alokaciju memorije. Tu svakako spadaju i dodatne funkcije vezane za virtuelnu memoriju, kao što su swaping tehnika i straničenje po zahtevu DP (demand paging).

- **CPU raspoređivač**. CPU raspoređivač dodeljuje procesor CPU procesima. Alokacija nastupa posle blokiranja procesa ili posle isteka vremenskog kvantuma.

- **IPC** (Inter Process Communication, međuprocesna komunikacija). Postoji više formi IPC, počevši od asinhronog signaliziranja događaja, do sinhronih prenosa poruka između njih.

Kontrola hardvera je odgovorna za upravljanje prekidima i za komunikaciju sa mašinom. Prekidni programi se ne servisiraju kao specijalni procesi već kao specijalne funkcije u kernelu, a u kontekstu procesa koji se izvršavaju.

Sistemski koncepti – datoteke i diskovi

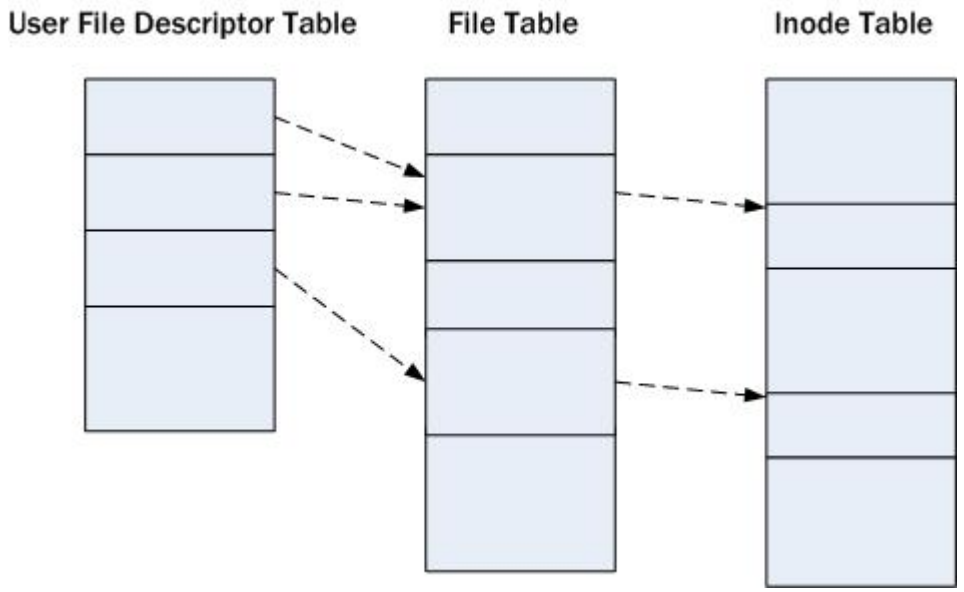
Interna reprezentacija datoteke definisana je preko inode strukture, koja opisuje raspored (layout) datoteke na disku i druge informacije kao što su vlasništvo, prava pristupa i vremena pristupa. Svaka datoteka ima jedinstvenu inode strukturu, ali može imati više imena (hard links). Kada proces traži neku datoteku po imenu, kernel analizira svaku komponentu u putanji (path-name), proverava da li proces ima prava da pretražuje u toj grani. Ako dođe do poslednje grane, proces otvara traženu inode strukturu. Kada proces kreira novu datoteku, kernel mora da dodeli novu, slobodnu inode strukturu.

Inode strukture se čuvaju u inode tabeli na disku, ali u cilju ubrzanja rada, kernel otvorene inode strukture čuva u memorijskoj (in-core) inode tabeli. Kernel sadrži još dve dodatne strukture podataka: tabelu datoeka (FT, file table) i UFDT tabelu (user file descriptor table) tj. tabelu korisničkih deskriptora datoteka. Tabela datoteka FT je globalna kernelska struktura, a UFDT je tabela koja se dodeljuje svakom procesu. Kada proces otvara ili kreira novu datoteku, kernel alocira po jedan ulaz iz svake od ove dve tabele.

Stanje otvorene (live) datoteke i korisničkog pristupa toj datoteci dato je preko tri tabele:

- **inode tabela**
- **Tabela FT**. Čuva zapis o pomeraju (offset) u datoteci gde će se sledeći upis ili čitanje startovati i prava pristupa koje ima proces za tu datoteku.
- **UFDT tabela**. Identifikuje sve otvorene datoteke za taj proces

Kada se datoteka kreira i otvori, kernel vraća deskriptor koji je indeks u UFTD tabeli. Kada se potom obavlja čitanje ili upis, na bazi deskriptora se ulazi u UFDT tabelu, iz koje se čita ulaz u tabelu datoteka FT, a onda se preko inode tabele realizuje odgovarajuće čitanje ili upis. Na slici 2.2 prikazan je odnos između ove tri tabele.



Slika 2.2. *UFDT, FT i in-core inode tabela*

Disk struktura

Disk može biti podeljen na više sistema datoteka od kojih svaki ima svoj logički broj. Logička adresa bloka ima u sebi dve komponente: broj sistema datoteka i broj bloka (FS number i logical block number), a fizička adresa bloka u sebi ima tri komponente: cilindar, glavu i sektor (cyl, head, sector). Konverzija između logičke i fizičke adrese spada u zadatak disk drajvera.

Diskovi se sastoje of fizičkih blokova veličine 512 bajtova. Sistem datoteka se sastoji od logičkih sistemskih blokova veličine 1K, 2K, 4K, 8K. Sistem datoteka FS se sastoji od više komponenti, kao što je prikazano na slici 2.3:

Slika 2.3. *Komponente sistema datoteka na operativnom sistemu UNIX*

- **Blok za podizanje sistema** (Boot block). Boot blok okupira početak sistema datoteka, obično je to prvi sektor na disku, a sadrži boot kôd za inicijalno

podizanje operativnog sistema.

- **Super blok**. Super blok opisuje stanje sistema datoteka, njegovu veličinu, koliko maksimalno datoteka može sadržati, gde se nalazi informacija o slobodnom prostoru u sistemu datoteka i razne druge informacije.
- **Inode lista**. Inode lista je tabela koja sledi iza super bloka. Administratori specificiraju veličinu ove tabele kada konfigurišu sistem datoteka. Kernel pronalazi inode strukture indeksiranjem u inode tabeli. Specijalna inode struktura za taj sistem datoteka je root inode struktura, a to je onaj inode preko koga je direktorijumska struktura tog sistema datoteka raspoloživa nakon uspešne mount komande.
- **Prostor podataka** (data area). Prostor podataka počinje odmah iza inode tabele i sadrži datoteke i direktorijume.

Sistemski koncepti – procesi

Proces je program u izvršavanju i sastoji se od skupa bajtova podeljenih u tri funkcionalne celine:

- tekst
- podaci
- stek

U princupu, ne postoji preklapanje ovih delova sa drugim procesima, a sva komunikacija između procesa odvija se preko IPC mehanizma . Postoje izuzeci od ovog pravila, kao što su deljivi kôd segmenti (shared code segment) i deljiva memorija (shared memory).

Praktično, proces na UNIX sistemu je celina kreirana preko sistemskog poziva fork. Svaki proces izuzev procesa nula, kreira se kada neki drugi proces izvrši fork sistemski poziv, pri čemu se taj drugi proces naziva roditelj, a kreirani proces dete. Svako dete ima samo jednog roditelja, a jedan roditelj može imati puno dece. Kernel svakom procesu dodeljuje jedinstveni broj PID. Proces koji ima PID nula je specijalan proces koji se kreira prilikom podizanja UNIX operativnog sistema. Potom taj proces obavlja fork proceduru za svoje prvo dete koje postaje proces init sa PID-om jednakim jedan, dok proces sa PID-om nula, postaje proces pod nazivom swapper. Proces init je predak svih procesa na UNIX operativnom sistemu i svi ostali procesi imaju specijalnu vezu sa njim.

Objasnimo malo strukturu programa koji je izvršan i koji se sastoji od više funkcionalnih delova:

- skup zaglavlja „header“ koji opisuje atribute datoteke
- takst (text) tj. programski kôd

- sekciju podataka inicijalizovanu (promenljive koje imaju početnu vrednost kada se program pokreće)
- sekciju podataka koja se ne inicijalizuje - bss
- razne sekcije kao što je na primer simbolička tabela

Kada se program puni u memoriju preko sistemskog poziva exec, minimalno mu se moraju dodeliti tri memorijska regiona: tekst, podaci i stek. Tekst region i region podataka odgovaraju tekst i data-bss sekciji samog programa, dok se stek automatski kreira i njegovu veličinu kernel automatski prilagođava u toku izvršenja (run-time). Stek se sastoji od logičkih stek okvira (frame), koji se guraju na stek (push), kada se funkcija pozove i skidaju sa steka (pop), kada se obavlja povratak iz funkcije.

Stek okvir (stack frame) sadrži adresu samog okvira, povratnu adresu funkcije, parametre funkcije, njene lokalne promenljive, podatke potrebne za regeneraciju prethodnog stek okvira, uključujući vrednosti PC registra (Programm Counter) i SP registra (Stack Pointer) u vreme funkcijskog poziva.

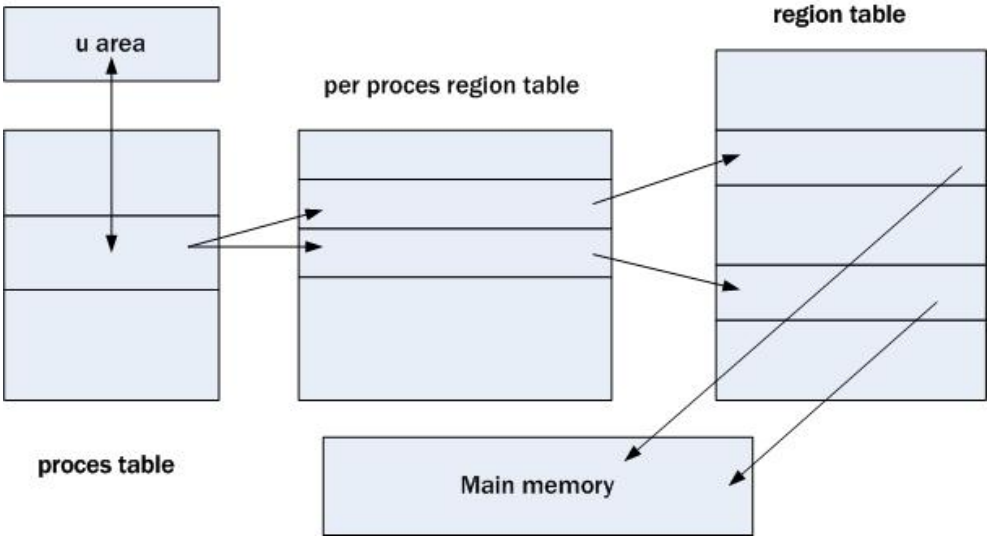
Kako proces na UNIX sistemu može da se izvršava u dva moda, kernelski i korisnički, za svaki mod se mora koristiti poseban stek.

Kernelski stek sastoji se od stek okvira za funkcije koje se izvršavaju u kernelskom modu i po strukturi je isti kao i korisnički stek. Ukoliko nema sistemskih poziva, izuzetaka i prekida, kernelski stek za proces neće postojati.

Svaki proces ima dva ulaza:

- ulaz u kernelskoj tabeli procesa KPT (Kernel Process Table)
- ulaz u u-područje (u-area), pri čemu u je skraćenica od engleske reči user. U-područje je područje kojim manipuliše isključivo kernel.

KPT (kernel process table) ulazi sadrže pokazivače, odnosno pokazuju na PPRT (per proces region table), čiji ulazi ukazuju na region tabelu. Region je kontunalni adresni prostor za jedan proces, kao što su tekst region, region podataka ili stek region. Region tabela (RT), u svojim ulazima opisuje atribute regiona (text-data, private-sharable) i lokaciju regiona u memoriji. Upotreba RT i PPRT omogućava efikasno deljenje regiona. Veza između KPT, PPRT, RT tabela je prikazana na slici 2.4.



Slika 2.4. *Veza između KPT, PPRT, RT tabela*

Objasnimo vezu između ovih tabela. Tabele procesa PT pokazuje na PPRT, PPRT ima pokazivače na glavnu tabelu regiona RT, koja opisuje regione dodeljene procesu. Ulaz u tabeli procesa PT i ulaz u u-području sadrže kontrolne i statusne informacije o svakom procesu.

Polja u ulazu tabele procesa PT su:

- polje stanja (state fileld)
- identifikatori koji opisuju koji korisnik je vlasnik procesa (user ID odnosno UID)
- skup opisivača događaja kada se proces suspenduje (in the sleep state)

Ulaz u u-području sadrži informacije o procesu koje su potrebne jedino kada se proces izvršava:

Polja u ulazu u-područja su:

- pokazivač na PT ulaz za proces koji se trenutno izvršava
- parametri tekućeg sistemskog poziva, povratne vrednosti i kôdove grešaka.
- deskriptore datoteka za sve otvorene datoteke
- tekući direktorijum i tekući root
- ograničenja za veličinu procesa i veličinu datoteka

Kernel uvek direktno pristupa poljima u u-području samo za proces koji se upravo izvršava.

Po pitanju sistemskih poziva i regiona važe sledeća pravila:

- **sistemski poziv exec**: kernel alocira regione za tekst, podatke i stek, a prethodno oslobođa sve regione koje je proces imao pre sistemskog poziva exec
- **sistemski poziv fork**: kernel duplira adresni prostor od procesa roditelja, dozvoljavajući da oba procesa dele regione uvek kad je moguće ili pravi kopije kada se to mora
- **sistemski poziv exit**: kernel oslobođa sve regione koje je proces posedovao

Kontekst procesa

Kontekst procesa je njegovo stanje koje se definiše preko sledećih komponenti:

- tekst ili kôd procesa
- vrednost globalnih promenljivih i struktura podataka
- vrednost CPU registara koje proces koristi
- vrednosti ulaza i u PT i ulaza iz u-područja
- sadržina korisničkog i kernelskog steka

Kada se kaže da operativni sistem izvršava proces, preciznije se kaže da operativni sistem izvršava kontekst procesa. Kada kernel odluči da izvršava neki drugi proces, mora se obaviti prebacivanje konteksta (kontext switch). Kernel dozvoljava prebacivanje konteksta samo pod određenim uslovima. Kada obavlja prebacivanje konteksta, kernel mora da sačuva dovoljno informacija o procesu koji se suspenduje kako bi kasnije mogao da ga nastavi. Takođe i prilikom prebacivanja iz korisničkog moda u kernelski mod, kernel mora sačuvati dovoljno informacija kako bi proces nastavio izvršavanje tamo gde je stao, nakon povratka u korisnički mod. Naravno, u ovom slučaju se ne menja kontekst procesa već samo mod.

Kernel opslužuje prekide u kontekstu jednog istog procesa koji se prekida, pa se nastavlja. Opsluživanje prekida se ne realizuje preko novih procesa, ali se radi uvek u kernelskom modu. Prekinuti proces može biti i u korisničkom modu i u kernelskom modu, a prilikom prekida mora se sačuvati dovoljno informacija da prekinuti proces može da se nastavi.

Stanja procesa

Proces se može naći u nekoliko stanja:

[1] **Izvršavanje u korisničkom modu** (running in user mode): Proces se trenutno izvršava u korisničkom modu

[2] **Izvršavanje u kernelskom modu** (running in kernel mode): Proces se trenutno izvršava u kernelskom modu

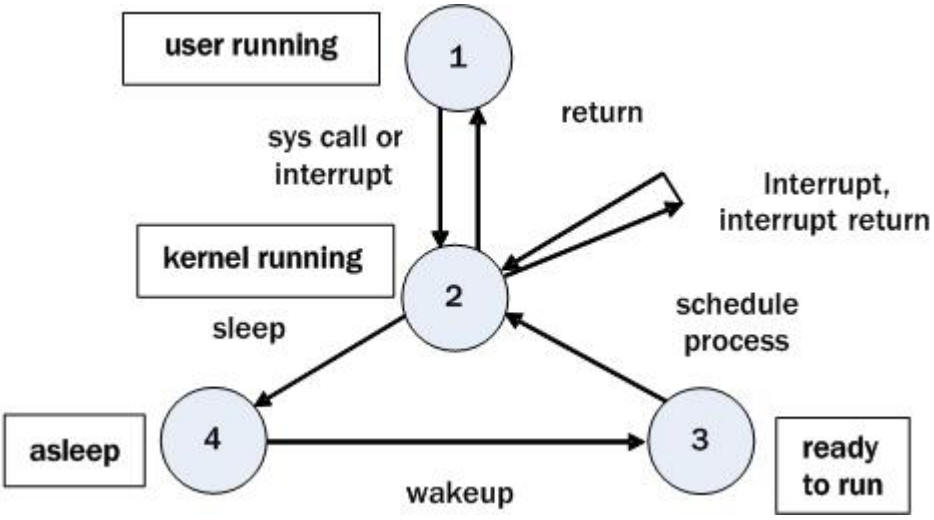
[3] **Spreman** (ready): Proces je spreman: ne izvršava se, nego čeka da ga raspoređivač (scheduler) pozove

[4] **Uspavan u memoriji** (asleep in memory): Proces je uspavan, blokira svoje izvršavanje zato što ne može dalje da nastavi jer čeka na nešto, na primer na završetak I/O operacije.

Pošto procesor može da izvršava samo jedan proces u vremenu, samo jedan proces može biti u stanju 1 ili 2. Svi ostali procesi, osim CPU aktivnog, su u stanju 3 ili 4.

Tranzicije procesa

Procesi često menjaju svoja stanja po dobro poznatim pravilima, koja su prikazana na slici 2.5, gde krug predstavlja stanje, a ivica sa strelicom predstavlja događaj koji izaziva da se proces pomeri iz jednog stanja u drugo.



Slika 2.5. *Stanja i tranzicije procesa*

Više procesa mogu biti u memoriji, a takođe više njih mogu raditi u kernelskom modu. Da ne bi došlo do narušavanja kernelskih struktura podataka, kernel ne dozvoljava bilo kakvo prebacivanje konteksta a takođe kontroliše i prekide.

Kernel dozvoljava prebacivanje konteksta samo kada se proces prebacuje iz stanja 2 (kernel running) u stanje 4 (asleep in memory). Proces koji radi u kernelskom modu ne može izgubiti procesor od nekog drugog procesa (preemted), tako da se za UNIX kernel kaže da je non-preemptive i na taj način se rešava problem međusobnog isključenja ME (mutual exclusion) u kernelskom modu. To znači da proces koji se izvršava u kernelskom

modu ne može izgubiti procesor. Čak i prekidi mogu biti suspendovani ako mogu da dovedu do nekonzistentnih podataka u kernelu. Takav deo kernelskog kôda naziva se kritična sekcija (CS, critical section). Dok se proces nalazi u CS (u kernel modu), kernel podiže CPU nivo tako da je većina prekida blokirana.

UNIX kernel sebe štiti tako što dozvoljava prebacivanje konteksta (context switch) samo na jednom mestu i u svojim kritičnim sekcijama blokira prekide koji su opasni (na primer, dok ažurira keš-baferske pokazivače, blokiraće disk prekide).

Uspavljivanje i buđenje (sleep i wakeup)

Proces koji radi u kernelskom modu ima veliku autonomiju da odluči o načinu reagovanja na sistemske događaje. Ako već mora da čeka na nešto, poželjno je da se uspava-blokira, mada je to odluka samog procesa. Na drugoj strani, rutina za obradu prekida (interrupt handler) ne sme da se uspavljuje, jer ako to učini, prekinuti proces bi podrazumevano bio uspavan.

Procesi se uspavljuju zato što čekaju na neki događaj, kao na primer: završetak I/O operacije, čekanje na drugi proces da se završi, čekanje da resurs postane raspoloživ. Procesi se blokiraju na događaj, a kada se on desi, prelaze u stanje 3 (ready to run). Mnogi procesi mogu biti uspavani na isti događaj, a kada se on desi, svi se bude i prelaze u red spremnih procesa (ready queue). Dakle, ne postoji neposredno izvršavanje.

Na primer, proces koji se izvršava u kernelskom modu može zaključati (lock) neke strukture podataka pre nego što ode na spavanje. Svi drugi moraju da čekaju na odključavanje (unlock), a do tada i oni idu na spavanje. Kernel implementira zaključavanje na sledeći način:

```
lock
while(condition is true)
sleep (event: the condition becomes false)
set condition true
```

Proces obavlja unlock i budi sve procese koji to čekaju:

```
unlock
set condition false;
wakeup (event: condition is false)
```

Strukture podataka kernela

Većina struktura podataka kernela zauzima tabele fiksnih veličina radije nego dinamički alocirani prostor. To ima prednosti jer uprošćava kernelski kôd, ali ograničava broj ulaza u kernelskim strukturama. Ako se prekorači broj ulaza, korisnik mora da čeka i šalje mu se poruka o grešci. Tada za njegov proces nema ulaza u kernelskoj strukturi. Postoje kerneli koji se mogu prilagoditii, ali to se kosi sa njihovom efikasnošću i jednostavnošću.

Sistemska administracija

Postoje razni adminstativni procesi koji obavljaju razne funkcije, kao što su formatiranje diska, stvaranje sistema datoteka, podešavanje parametara kernela i slično. Za takve procese, kernel zahteva root privileguju, odnosno zahteva da takve procese izvodi specijalni korisnik pod imenom root, sa atributima korisnika sa najvećim ovlašćenjima (superuser).

2.2. Uvod u Linux kernel

Linux predstavlja jednu od poslednjih varijanti UNIX operativnih sistema, čiji je razvoj započeo Linus Torvalds 1991. godine na Unverzitetu u Helsinkiju. Torvalds je svoj operativni sistem koji objedinjuje oba standarda, SRV4 i BSD, objavio na Internetu i podsticao druge programere širom sveta da se priključe njegovom daljem razvoju. Ubrzo, Linux je postao veoma popularan među računarskim entuzijastima, koji su tražili alternativno rešenje za postojeće operativne sisteme za PC računare (DOS, Windows). Linux je svojom koncepcijom, stabilnog a jeftinog operativnog sistema doživeo veliku ekspanziju i popularnost. Simbol Linux sistema je mali pingvin (Tux), prikazan na slici 2.6.

Slika 2.6. *Simbol Linuxa*

Linux je prvobitno namenjen 32-bitnim Intel x86 mikroprocesorima (počevši od 80386), na kojima može funkcionisati kao radna stanica (workstation) ili kao server. Linux kernel je kasnije modifikovan i prilagođen procesorima koji ne pripadaju Intel x86 klasi, među kojima treba istaći sledeće: Intel IA-64, DEC Alpha, SUN SPARC/UltraSPARC, Motorola 68000, MIPS, PowerPC i IBM mainframe S/390. Može se konstatovati da današnji Linux u odnosu na bilo koji operativni sistem podržava najširi spektar procesora i računarskih arhitektura.

Veliki deo komponenti Linux operativnom sistemu dodali su nezavisni programeri i programeri GNU projekta (www.gnu.org), koji pripada slobodnoj softverskoj fondaciji (FSF

- Free Software Foundation). Svi GNU/Linux operativni sistemi koriste Linux kernel kao fundamentalni deo koji kontroliše interakciju između hardvera i aplikacija, i GNU aplikacije kao dodatne komponente operativnog sistema.

Linux je raspoloživ kao besplatan operativni sistem pod GNU GPL licencom (GNU General Public License), što važi i za neke druge vrste UNIX sistema, kao što su FreeBSD i NetBSD. Linux je softver sa otvorenim izvornim kôdom (Open Source), što znači da je mu je izvorni kôd javno raspoloživ i može biti modifikovan tako da odgovara specifičnim potrebama. Linux se može slobodno distribuirati među korisnicima. Ovakav koncept je potpuno suprotan konceptu komercijalnog softvera, gde izvorni kôd nije dostupan i svaki korisnik mora da plati licencu za korišćenje. Komercijalni softver je baziran na autorskim pravima (copyright laws), koja preciziraju limite koje korisnici softvera imaju u odnosu na izvorni kôd, korišćenje i dalje distribuiranje softvera. Linux se besplatno može preuzeti sa različitih web lokacija.

Brojne profitno orijentisane i mnoge neprofitne organizacije čine Linux raspoloživim u formi distribucija, odnosno različitih kombinacija kernela, sistemskog softvera i korisničkih aplikacija. Većina distribicuja sadrži kolekciju CD/DVD medijuma na kojima se nalazi operativni sistem, izvorni kôd, detaljna dokumentacija, kao i štampana uputstva za instalaciju i upotrebu sistema. Cene ovakvih distribucija su u većini slučajeva simbolične, osim ako se u distribuciji nalazi komercijalan softver ili je distribucija specifične namene.

Osnovna komponenta svake Linux distribucije je kernel operativnog sistema. Osim kernela i sistemskog softvera, u distribuciji se nalaze i instalacioni alati, softver za podizanje operatvnog sistema (boot loader), razne korisničke aplikacije (kacelarijski paketi - office suite, softver za manipulaciju bit-mapiranih slika) i serverski paketi. Većina distribucija je poput Windows sistema, grafički orijentisana prema korisniku, dok su neke distribucije namenjene za sistemske administratore i programere familijarne sa tradicionalnim UNIX okruženjem.

Opšti pregled Linux sistema

Linux je višekorisnički, višeprocesni operativni sistem sa potpunim skupom UNIX kompatibilnih alata, projektovan tako da poštuje relevantne POSIX standarde. Linux sistemi podržavaju tradicionalnu UNIX semantiku i potpuno implementiraju standardni UNIX mrežni model.

Linux operativni sistem sastoji se od kernela, sistemskog softvera, korisničkih aplikacija, dokumentacije, programskih prevodioca i njihovih odgovarajućih biblioteka (GCC - GNU C Compiler i C biblioteka za Linux). Sadržaj konkretne Linux distribucije definisan je sadržajem instalacionih medijuma, koji u slučaju nekih Linux sistema uključuju razne FTP sajtove širom sveta.

Kernel je srce operativnog sistema - on omogućava konkurentno izvršavanje procesa, dodeljuje memoriju i druge resurse i obezbeđuje mehanizam za ostvarivanje servisa operativnog sistema. Kernel štiti korisničke procese od direktnog pristupa hardveru - procesi pristupaju hardveru korišćenjem sistemskih poziva kernela, čime se obezbeđuje

jedna vrsta zaštite između samih korisnika. Sistemski programi koriste kernel u cilju implemetacije različitih servisa operativnog sistema.

Svi programi, uključujući i sistemske, funkcionišu na nivou iznad kernela, što se naziva korisnički režim rada, dok se sistemske aktivnosti poput pristupa hardveru obavljaju na nivou kernela, odnosno u kernelskom režimu rada. Razlika između sistemskih i aplikativnih programa je u njihovoj nameni: aplikacije su namenjene za razne korisne aktivnosti (kao što su obrada teksta i slike), dok su sistemski programi namenjeni za rad sa sistemom i administraciju. Na primer tekst procesor je korisnička aplikacija, dok je komanda mount sistemski program. Razlike između korisničkih i sistemskih programa su ponekad veoma male i značajne samo za stroge kategorizacije softvera.

Linux kernel

Tri osnovne verzije Linux kernela su početna verzija, verzija 1.x i verzija 2.x. Početna verzija 0.01, koju je 1991. godine kreirao Linus Torvalds, podržavala je samo Intel 80386 kompatibilne procesore, mali broj hardverskih uređaja i Minix sistem datoteka. Mrežni servisi nisu imali kernelsku podršku. Verzija 1.0, nastala u martu 1994. godine, uključivala je podršku za standardne TCP/IP mrežne protokole, BSD-kompatibilni socket interfejs za mrežno programiranje i drajversku podršku za mrežne kartice. Ova verzija je dodatno podržavala ext i ext2 sisteme datoteka, široku klasu SCSI disk kontrolera, kao i brojne hardverske uređaje. Verzija 1.2 (mart 1995) je poslednja verzija Linux kernela namenjena isključivo PC arhitekturi. U verziji 2.0 (jun 1996) uvedena je podrška za više arhitektura (Motorola i Intel procesori, Sun Sparc i PowerMac sistemi), kao i podrška za višeprocesorsku arhitekturu (SMP). Dodatno, poboljšano je upravljanje memorijom i uvećane su performanse TCP/IP protokol steka, a ugrađena je i podrška za unutrašnje kernelske niti (internal kernel threads). Kernel je modularizovan, odnosno uvedena je mikro-kernel struktura sa izmenljivim drajverskim modulima (loadable kernel modules), a standardizovan je i konfiguracioni interfejs.

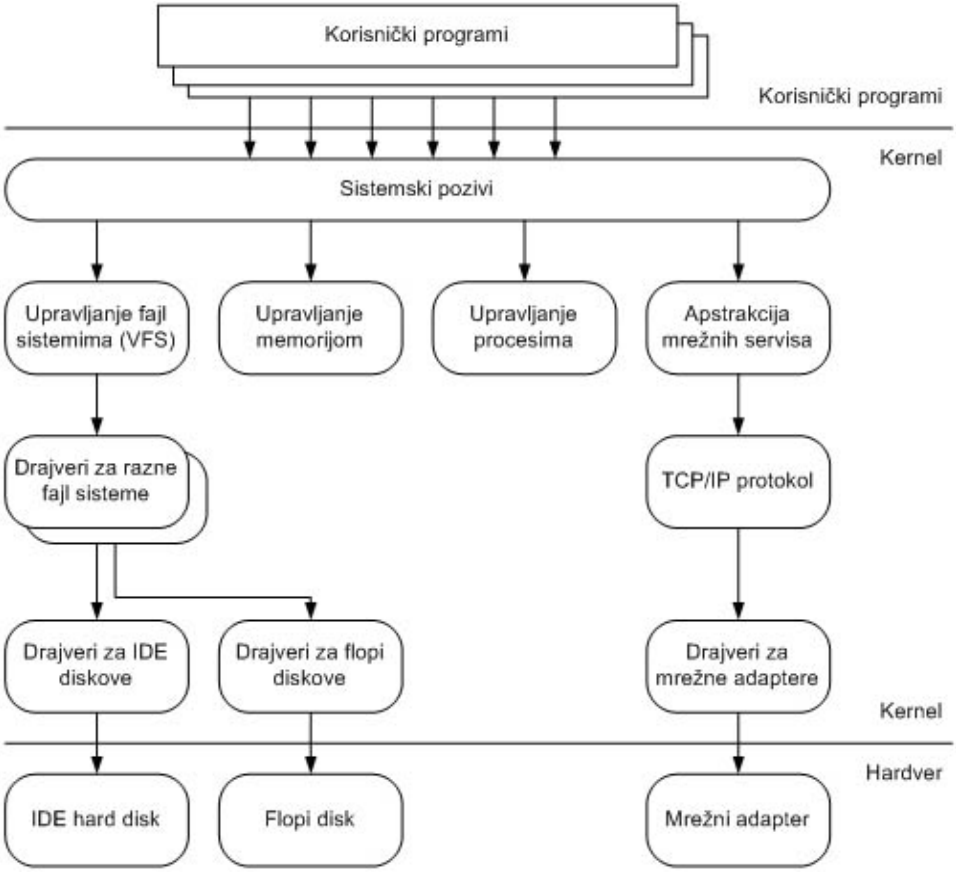
Struktura kernela Linux sistema

Osnovu Linux sistema čine kernel, sistemske biblioteke i sistemski programi. Kernel je odgovoran za najznačajnije funkcije operativnog sistema.

Dve osnovne karakteristike kernela su:

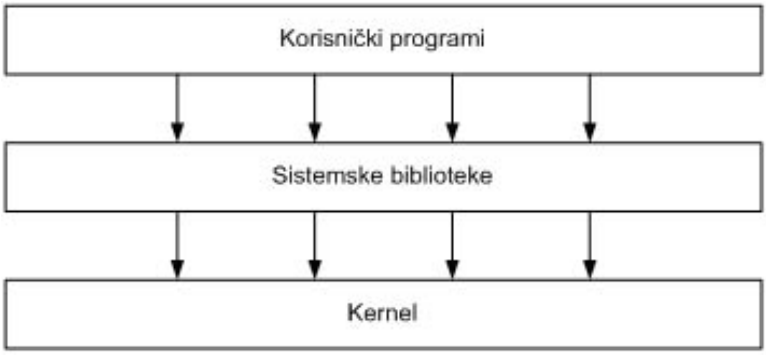
- kernelski kôd se izvršava u kernelskom modu u kome je jedino moguće pristupati svim komponentama hardvera
- kompletan kernel kôd i sve kernel strukture podataka čuvaju se u istom adresnom prostoru (monolithic)

Kod većine UNIX sistema, aplikacije se preko sistemskog poziva direktno obraćaju kernelu, kao što je prikazano na slici 2.7.



Slika 2.7. *Struktura UNIX kernela*

Kod Linux sistema, sistemski pozivi se upućuju kernelu preko sistemskih biblioteka, koje definišu standardni skup funkcija preko kojih aplikacije komuniciraju sa kernelom. Ovaj metod komunikacije sa kernelom prikazan je na slici 2.8.



Slika 2.8. *Komunikacija sa Linux kernelom*

Sistemski programi izvršavaju specifične upravljačke poslove, kao što je konfigurusanje mrežnih uređaja, i protokola, punjenje kernelskih modula itd.

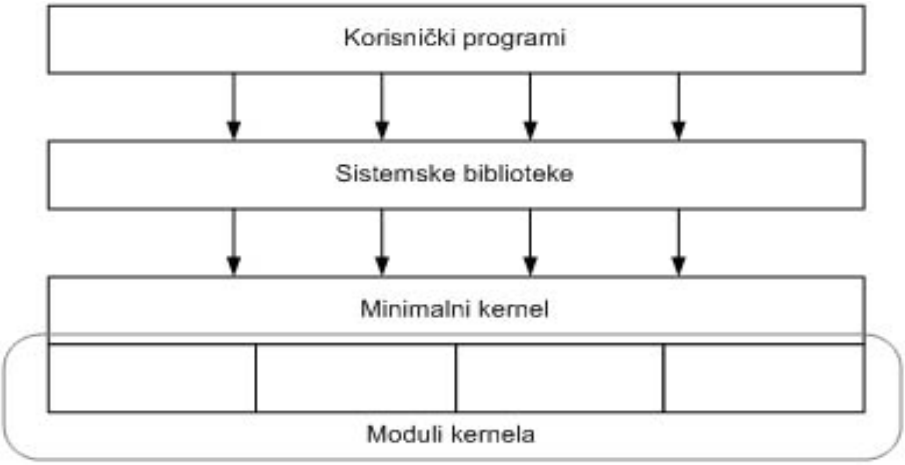
Modularni kernel

Moduli kernela su delovi kernelskog koda koji može da se prevede, napuni u memoriju ili izbaci iz memorije nezavisno od ostatatka kernela. Kernelski moduli implemetiraju, drajvere za hardverske uređaje, novi sistem datoteka, mrežne protokole, itd. Moduli omogućavaju raznim programerima da napišu i distribuiraju drajvere koji ne moraju da prođu GPL licencu.

Moduli kernela omogućavaju micro-kernel arhitekturu, odnosno realizaciju minimalne stabilne konfiguracije kernela bez dodatnih drajvera. Potrebni drajveri pune se u memoriju kao moduli kernela. Module Linux kernela čine tri komponente:

- upravljanje modulom, koja omogućava punjenje modula u kernelsku memoriju i komunikaciju modula sa ostatkom kernela, proveru prisustva modula u memoriji, proveru korišćenja modula i izbacivanje modula iz memorije (pod uslovom da se modul ne koristi).
- registracija drajvera, koja omogućava modulu da objavi ostatku kernela da je novi drajver u memoriji i da je raspoloživ za korišćenje. Kernel održava dinamičku tabelu drajvera, koji se pomoću posebnog skupa programa mogu napuniti ili izbaciti memorije u svakom trenutku.
- rezolucija konflikata, odnosno mehanizam koji služi da spreči hardverske konflikte, tako što omogućava drajveru da rezerviše hardverske resurse (IRQ, DMA, ports) i time spreči druge drajvere ili funkciju koja automatski detektuje hardver (autoprobe) da ih koriste.

Na slici 2.9 prikazana je struktura modularnog Linux kernela:



Slika 2.9. *Modularna struktura Linux kernela*

Značajni delovi kernela

Linux kernel čini nekoliko značajnih komponenti:

- upravljanje procesima
- upravljanje memorijom
- upravljanje sistemima datoteka (VFS)
- apstrakcija mrežnih servisa
- podrška za hardverske uređaje
- podrška za različite sisteme datoteka
- podrška za TCP/IP

Kritične komponente Linux kernela čini upravljanje procesima i upravljanje memorijom. Komponenta za upravljanje memorijom kontroliše bafersko keširanje i dodelu memorije i swap prostora, kako procesima tako i kernelskim komponetama. Komponenta za upravljanje procesima stvara procese i omogućava višeprocesni rad (multitasking), dodeljujući procesor procesima po odgovarajućem algoritmu.

Na najnižem nivou, kernel sadrži podršku u vidu drajvera za razne hardverske uređaje. Veliki broj vrsta hardverskih uređaja povlači veliki broj drajvera. Za hardverske uređaje iste vrste (npr. diskove), koji vrše sličnu funkciju, ali se razlikuju u načinu softverske kontrole, formirane su opšte klase drajvera na sledeći način: svaki član klase pruža isti interfejs prema ostatku kernela, čime obezbeđuje podršku za operacije koje su karakteristične za tu vrstu uređaja, dok opslužuje hardver na odgovarajući način. Na primer, svi disk drajveri pružaju isti interfejs prema ostatku kernela i svi imaju operacije

tipa inicijalizacije uređaja, čitanja podataka iz određenog sektora i upisa podataka u određeni sektor.

Takođe, neki softverski servisi koje kernel podržava imaju slične osobine, čime se omogućava njihova apstrakcija u klase. Na primer, različiti mrežni protokoli se apstrakuju u jedan programski interfejs koji se naziva "BSD socket library". Drugi primer je virtuelni sistem datoteka (VFS - virtual filesystem), koji apstrakuje operacije u sistemu datoteka, pri čemu svaki tip sistema datoteka obezbeđuje specifične implementacije raznih operacija. Zahtev za korišćenjem sistema datoteka koji šalje korisnik prolazi kroz VFS sloj, koji isti prosleđuje na odgovarajući drajver za konkretan sistem datoteka.

Upravljanje procesima

Linux koristi standardni UNIX proces mehanizam (fork), koji kreiranje procesa i njegovo izvršenje razdvaja u dve različite operacije:

- sistemski poziv fork, koji kreira novi proces
- sistemski poziv exec, koji izvršava program u resursima novostvorenog procesa

Pod UNIX sistemom, sve informacije koje operativni sistem mora čuvati da bi kontrolisao jedan proces čine kontekst tog procesa. Pod Linux operativnim sistemom, svaki proces je u potpunosti opisan identitetom, okolinom i kontekstom.

Identitet procesa obuhvata sledeće informacije:

- Identifikator procesa (Process ID - PID), pomoću kog Linux kontroliše proces.
- Akreditivi (Credentials). Svaki proces pripada jednom korisniku koji ima svoj ID i jedan ili više grupnih ID-ova, koji određuju prava pristupa procesu u radu sa datotekama.
- Ličnost (Personality). Ova informacija se ne koristi kod drugih UNIX sistema, a Linux svakom procesu dodeljuje lični identifikator koji može imati uticaja za neke sistemske pozive.

Okolina procesa se nasleđuje od procesa roditelja. U okolinu procesa spadaju vektor argumenata koje proces roditelj prosleđuje programu i vektor okoline, odnosno lista promenljivih koje definišu okolinu procesa (environment).

Kontekst procesa je stanje procesa u datom trenutku vremena. Kontekst procesa čine sledeće komponente:

- Kontekst za raspoređivanje (scheduling context), koji služi za efikasnu suspenziju ili ponovno pokretanje procesa. Obuhvata sve CPU registre, prioritet procesa i kernelski stek procesa.
- Statistički kontekst, koji sadrži informacije o resursima koje proces koristi u jednom trenutku, kao i kompletnu upotrebu resursa za vreme trajanja jednog procesa (accounting information)

- Tabela datoteka (file table), tj. polje pokazivača na kernelske strukture datoteka
- Kontekst sistema datoteka (file-system context)
- Tabela za upravljanje signalima (signal-handler table), koja definiše pokazivače na programe koji se pozivaju nakon određenog signala
- Kontekst virtuelne memorije (virtual-memory context), koji potpuno opisuje korišćenje memorije od strane procesa

Procesi i niti

Linux koristi istu internu reprezentaciju za procese i niti (threads). Nit je jednostavno novi proces koji deli adresni prostor roditelja. Za razliku od novog procesa koji pomoću sistemskog poziva fork formira novi kontekst sa unikatnim adresnim prostorom, nit nastaje pomoću sistemskog poziva clone, koji kreira novi kontekst, ali dozvoljava novom procesu da deli adresni prostor roditelja.

Dodeljivanje procesora procesima

Linux koristi dva algoritma za dodelu procesora procesima (process-scheduling algorithms):

- Algoritam za rad u deljenom vremenu (time-sharing) za korektno raspoređivanje između procesa (fair preemptive scheduling). Dodela se vrši na osnovu prioriteta procesa koji definiše korisnik i kredita (efektivni prioritet) koji raste s porastom vremena čekanja na procesor po sledećoj rekurzivnoj formuli: kredit = kredit/2 + prioritet.
- Algoritam za rad u realnom vremenu (real-time) za procese gde su apsolutni prioriteti mnogo značajniji od ravnomerne raspodele. Linux je ipak soft-real time operativni sistem.

Izbor algoritma koji će biti primenjen zavisi od klase u kojoj se proces nalazi (FIFO ili round-robin). Trenutna pozicija procesa u svakoj od klasa određuje se na osnovu prioriteta, što znači da će se izvršavati onaj proces koji ima najviši prioritet, a u slučaju da su prioriteti isti, izvšava se proces koji je najduže čekao. U FIFO klasi, procesi nastavljaju da rade do prirodnog završetka ili dok ne uđu u blokirano stanje, dok u round-robin klasi, svaki proces radi dok mu ne istekne vremenski kvantum (time-slice), posle čega prekida rad i i odlazi na kraj liste za čekanje.

Počevši od kernala 2.0, Linux podržava SMP, što znači da se različiti procesi ili niti mogu paralelno izvršavati na posebnim procesorima. Da bi se obezbedila procesorska sinhronizacija kernela u SMP okruženju, samo jedan CPU može izvršavati kôd u kernelskom modu.

Interprocesna komunikacija

Interprocesna komunikacija obuhvata obaveštavanje procesa o događaju i prenos podataka s jednog procesa na drugi. Kao i UNIX, u korisničkom režimu, Linux putem signala informiše procese o nastalom događaju. Procesi u kernelskom modu umesto signala, za interprocesnu komunikaciju koriste specijalnu vrstu deljive memorije (wait.queue struktura).

Za prosleđivanje podataka između procesa koriste se pipe mehanizam i deljiva memorija. Pipe omogućava jednosmernu razmenu podataka putem komunikacionog kanala koji proces nasleđuje od roditelja, dok je deljiva memorija brza i fleksibilna, ali zahteva sinhronizaciju.

Upravljanje memorijom

Upravljanje memorijom obuhvata upravljanje operativnom (RAM) i virtuelnom memorijom.

Upravljanje fizičkom memorijom bavi se alokacijom i oslobađanjem stranice (pages, normal extent), grupe stranica (large extent) i malih memorijskih blokova (small extent). Upravljanje fizičkom memorijom obavlja se po sistemu drugova (Buddy heap). Cela fizička memorija deli se na udružene blokove čije su veličine stepeni broja dva. Blokovi se prema potrebi alokacije dalje razbijaju na manje blokove ili se parovi udružuju u veće celine.

Sistem virtuelne memorije povećava ukupan adresni prostor koji je dostupan procesima - sistem kreira stranicu virtuelne memorije na zahtev, upravlja punjenjem te stanice sa diska u fizičku memoriju i povratkom stranice na disk u swap prostor. Kada stranica mora da napusti memoriju i ode na disk, izvršava se takozvani page-out alogoritam, koji je na Linux sistemu realizovan LFU konceptom (Least Frequently Used). Novi virtuleni adresni prostor se formira nakon kreiranja novog procesa sistemskim pozivom fork i nakon izvršavanja novog programa sistemskim pozivom exec.

Regioni virtuelne memorije obuhvataju fizičke stanice (frame) i swap prostor na disku.

Izvršavanje korisničkih programa

Linux podržava brojne formate za punjenje i izvršavanje programa. Među njima svakako treba istaći stari UNIX format a.out i novi elf format, koji je maksimalno prilagođen konceptu virtuelne memorije. ELF format se sastoji od zaglavlja koje opisuje sekcije programa. Sekcije programa su po veličini prilagođenje veličini stanice virtuelne memorije.

Program u čijem su slučaju funkcije iz sistemske biblioteke direktno ugrađene u kôd, predstavlja program sa statičkim povezivanjem (linkovanjem). Glavni nedostatak ovakvog povezivanja ogleda se u povećanju veličine koda. Naime, svaki poziv funkcije iz biblioteke kopira u kôd celu funkciju. Čim je veličina kôda veća, veća je i količina memorije potrebna za njegovo izvršavanje. Na drugoj strani, dinamičko povezivanje je

efikasnije i modernije, sama funkcija se ne kopira u kôd, a i za izvršenje je potrebna manja količina memorije.

Ulazno - izlazni sistem

Linux deli uređaje u tri klase:

- Blok uređaje (poput diskova i CD-ROM uređaja),
- Karakter uređaje (poput štampača)
- Mrežne uređaje.

Svaki uređaj je predstavljen specijalnom datotekom (device node, device file) koja se nalazi u /dev direktorijumu root sistema datoteka. Kada korisnik čita ili upisuje podatke u datoteku koja predstavlja neki uređaj, vrši se neka ulazno-izlazna operacija, odnosno sistem šalje ili prima podatke sa uređaja koji je predstavljen tom datotekom. Time se ukida potreba za postojanjem posebnih programa (a samim tim i posebne metodologije programiranja ulazno-izlaznih operacija) neophodnih za rad sa uređajima. Na primer, korisnik može da odštampa tekst na štampaču jednostavnom redirekcijom standardnog izlaza na datoteku /dev/lp1 koji predstavlja štampač:

```
# cat izvestaj.txt > /dev/lp1
```

Ova komanda će korektno odštampati datoteku izvestaj.txt ukoliko je ona u obliku koji štampač razume (npr. tekstualna datoteka). Međutim, nije preporučljivo istovremeno slanje datoteka na štampač od strane više korisnika pomoću redirekcije izlaza, jer se zaobilazi red za čekanje za štampač (print spooler). Poseban program, lpr (line printer) obezbeđuje da datoteke poslate na štampu čekaju u redu, i prosleđuje ih štampaču tek kad je štampanje prethodne datoteke završeno. Slični programi postoje za većinu uređaja, tako da korisnici uglavnom ne koriste specijalne datoteke.

Direktorijum /dev nastaje prilikom instalacije Linux sistema i u njemu se nalaze sve specijalne datoteke. Bez obzira da li je uređaj instaliran na sistem ili ne - postojanje datoteke /dev/sda ne znači da je na sistem instaliran SCSI disk. Postojanje svih datoteka olakšava proces instalacije novog hardvera, tj. oslobađa administratora sistema potrebe za kreiranjem specijalnih datoteka sa korektnim parametrima.

Sistemi datoteka i aktivno UNIX stablo

Linux sistemi datoteka koriste hijerarhijsku strukturu stabla i semantiku UNIX sistema datoteka. Interno, kernel sakriva detalje i upravlja različitim sistemima datoteka preko jednog nivoa apstrakcije, koji se naziva virtuelni sistem datoteka VFS.

Aktivno Linux stablo datoteka čini jedan ili više sistema datoteka koji su montirani na odgovarajuće direktorijume preko kojih im se pristupa. Osnovu aktivnog stabla datoteka čini korenski sistem datoteka (root filesystem), čiji root direktorijum ujedno predstavlja i root direktorijum aktivnog stabla datoteka. Zavisno od hardverske konfiguracije i odluke

administratora sistema, struktura aktivnog Linux stabla može biti jednostavna (aktivno stablo realizovano jednim sistemom datoteka), ili složena (aktivno stablo realizovano većim brojem sistema datoteka - root, /boot, /var, /usr, /home ...)

Mrežne strukture

Umrežavanje je ključno područje funkcionalnosti Linux sistema. Linux koristi standardni TCP/IP protokol stek kao osnovni komunikacioni protokol, a dodatno podržava i brojne druge protokole koji nisu uobičajeni za komunikaciju dva UNIX sistema (AppleTalk, IPX, Samba).

Interno, umrežavanje pod Linux sistemom obuhvata tri softverska nivoa: soket interfejs, protokol drajvere i drajvere za mrežne kartice.

2.3. Baferski keš (Buffer cache)

Preko baferskog keša, kernel pokušava da smanji broj disk pristupa. Baferski keš je skup internih bafera podataka, koji sadrži podatke od nedavno korišćenih disk blokova. Kada se proces obraća datoteci, kernel pokušava da je locira u kešu; ako je datoteka u kešu, ne čita je sa diska, a ako nije, prvo je dovede u keš, čuvajući u kešu one podatke za koje algoritam smatra da su najkorisniji.

Po pitanju upisa (write), keš je vrlo beneficijalan, može da kompenzuje višestruka upisivanja ili da potpuno eliminiše upis na disk, ako usledi proces brisanja datoteke. Takođe, više odloženih upisa se mogu kombinovati u optimalnom redosledu.

Keš algoritam izdaje instrukcije za čitanje unapred (read-ahead, pre-cache) i odloženi upis (delayed-write) u cilju maksimizovanja keš efekata.

Baferska zaglavlja (Buffer headers)

Za vreme sistemske inicijalizacije, kernel alocira prostor za određeni broj bafera, a taj prostor zavisi od memorijske veličine i sistemskih performansi. Bafer se sastoji od dve komponente:

- bafer: memorijsko polje koje sadrži podatke, obično na nivou logičkog bloka sistema datoteka
- bafersko zaglavlje (header) koje identifikuje bafer

Kernel ispituje zaglavlja kako bi analizirao sadržaj keš bafera. Bafersko zaglavlje (prikazano na slici 2.10) sadrži sledeća polja:

[1] broj uređaja (device number)

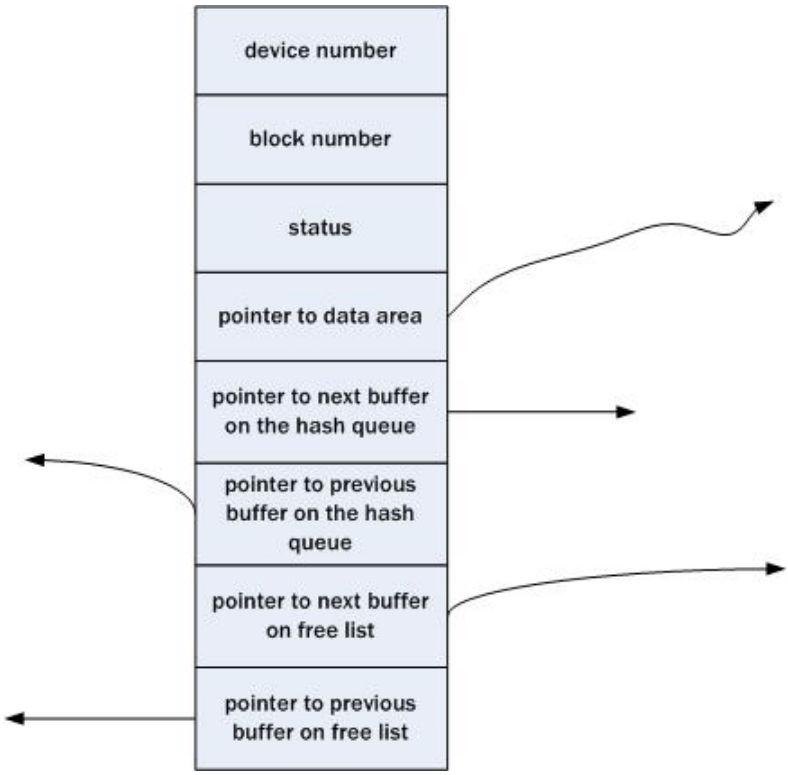
[2] broj bloka (block number)

[3] pokazivač na podatke (data pointer): bafersko zaglavlje sadrži pokazivač na područje podataka za sam taj bafer

[4] status

- zaključan (locked) – bafer je trenutno zaključan što znači da je zauzet (puni se)
- važeći (valid) – bafer sadži validne podatke
- odloženi upis (delayed write) – kernel mora upisati sadržaj bafera na disk pre nego što obavi ponovnu dodelu bafera
- u stanju čitanja/pisanja (in reading/writing) – kernel trenutno čita ili piše po baferu
- proces čeka da bafer postane slobodan (wait for be free)

[5] skup pokazivača za alokaciju bafera koje koristi keš algoritam



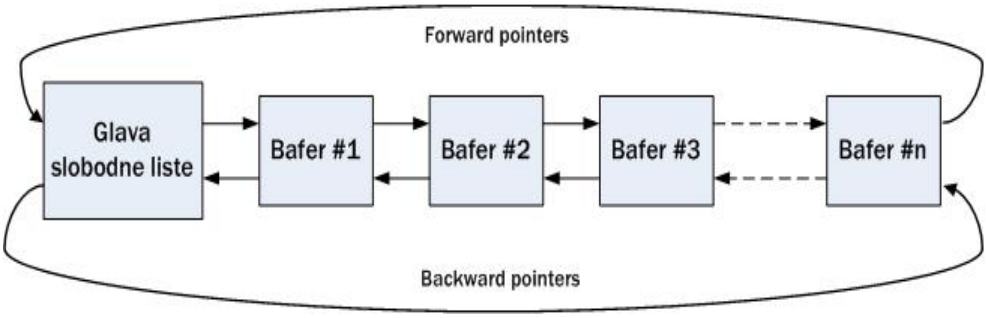
Slika 2.10. *Bafersko zaglavlje*

Gomila bafera (Buffer pool)

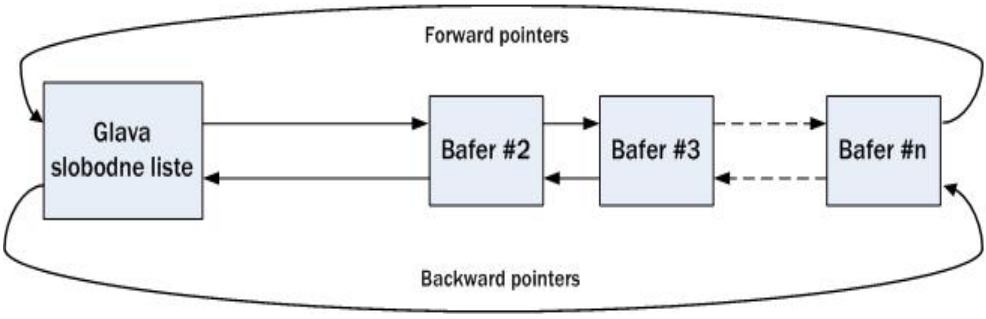
Baferi se opslužuju po LRU (least recently used) algoritmu: kada se bafer dodeli disk bloku, taj blok će ostati u kešu sve dok ne postane najstariji blok u baferu po pitanju pristupa. Kernel održava listu slobodnih bafera u LRU redosledu. Slobodna lista je dvostruko povezana kružna lista sa zaglavljem na početku.

U početku su svi baferi slobodni. Obično kernel uzima bafer koji je na početku i izbacuje ga iz slobodne liste, a po povratku ga stavlja na kraj.

Slika 2.11a prikazuje slobodnu lista pre dodele bafera #1. Slika 2.11b prikazuje slobodna listu posle dodele bafera #1.



Slika 2.11a. *Slobodna lista pre dodele bafera #1*



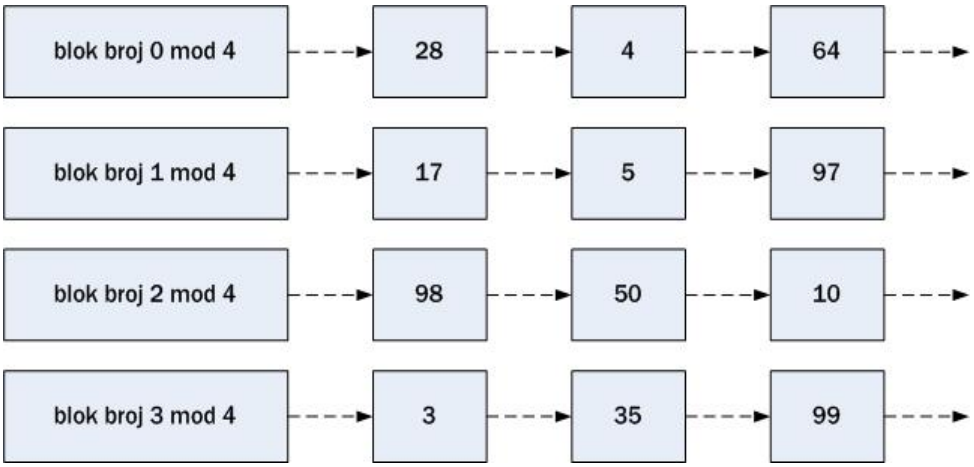
Slika 2.11b. *lobodna lista posle dodele bafera #1*

Kada kernel pristupa disk bloku, on pretražuje sva zaglavlja za odgovarajuću kombinaciju od dva broja (device number, block number). Da ne bi pretraživao sva zaglavlja, kernel organizuje bafer u odvojene redove na bazi heš funkcije od ova dva broja. Kernel povezuje bafere na dvostruko povezane heš redove čekanja, u sličnu

strukturu kao za listu slobodnih blokova. Broj bafera za jedan heš red čekanja (hash queue) je promenljiv i zavisi od obraćanja disku.

Za raspodelu bafera između heš redova, kernel koristi heš funkciju koja mora biti dovoljna brza. Broj heš redova se određuje prilikom podizanja sistema i to može biti podrazumevano (default) ili taj broj određuje sistem administrator.

Na slici 2.12 je dat sistem sa 4 heš reda čekanja (moduo 4).



Slika 2.12. *Baferski keš sa 4 heš reda čekanja*

Na levoj strani nalaze se zaglavlja, a blokovi se raspoređuju u heš redove po funkciji: broj bloka po modulu 4. Isprekidane linije ukazuju na heš pokazivače u oba smera. Svaki bafer se nalazi u nekom od heš redova čekanja, ali može se dogoditi da bude i u slobodnoj listi.

Tehnike za dobijanje podataka iz bafera

Keš algoritam upravlja baferskim kešom. Za svako čitanje sa diska, kernel prvo mora odrediti prisustvo bloka u kešu i ako se tamo ne nalazi, mora da mu se dodeliti slobodan bafer. To se isto dešava i prilikom upisa. Oba algoritma i za čitanje i za upis koriste čuveni UNIX getblk algoritam koji prilikom dodele bafera u kešu ima pet mogućih scenarija:

- scenario #1. Kernel pronalazi bafer u njegovom heš redu čekanja i bafer je slobodan
- scenario #2. Kernel ne nalazi bafer u njegovom heš redu čekanja, zato mora da alocira bafer iz slobodne liste

- scenario #3. Kernel ne nalazi bafer u njegovom heš redu čekanja, zato mora da alocira bafer iz slobodne liste, ali taj bafer ima atribut odloženi upis (delayed write), što znači da mora prvo da se upiše na disk
- scenario #4. Kernel ne nalazi bafer u njegovom heš redu čekanja, ali i slobodna lista je prazna
- scenario #5. Kernel nalazi bafer u njegovom heš redu čekanja, ali je taj bafer zauzet (busy), tj. drugi proces ga je već uzeo za sebe

Obratite pažnju: blok je slobodan, ako je u slobodnoj listi, koja znači da ga niko ne koristi pri čemu može ili ne mora biti u heš listi.

Algoritam Getblk

Navodimo pseudo kôd za algoritam getblk.

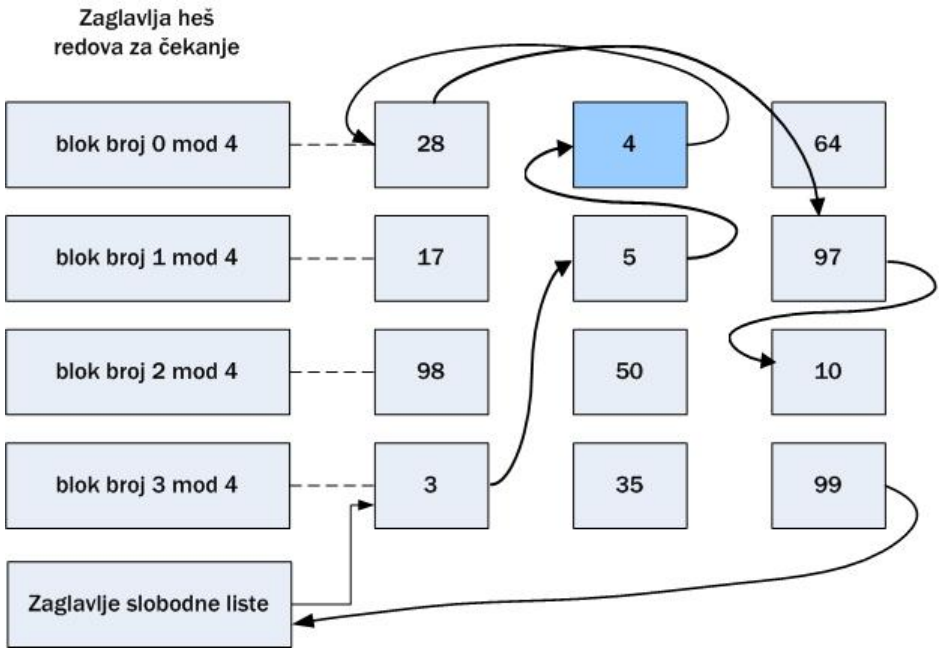
```
input:  FS number, block number
output: locked buffer that can now be used for block

{
  while (buffer not found)
  {
    if (buffer in hash queue)
    {
      if (buffer busy)                          /*scenario 5*/
      {
        sleep (event buffer becomes free);
        continue; /* back to while lop*/
      }
      mark buffer busy;                         /*scenario 1*/
      remove buffer from free list; /*lock*/
      return buffer;
    }
    else /*block not on hash queue*/
    {
      if (there are no buffers on free list)    /*scenario 4*/
      {
        sleep (event any buffer becomes free);
        continue; /* back to while lop*/
      }
      remove buffer from free list; /*lock*/
      if (buffer marked for delayed write)      /*scenario 3*/
      {
        asynchronous write buffer to disk;
        continue; /* back to while loop*/
      }
```

```
        /* scenario 2 - found a free buffer*/
        remove buffer from old hash queue;
        put buffer onto new hash queue;
        return buffer;
      } /*else*/
    }   /*while*/
}       /*main*/
```

Objasnimo getblk na primerima.

Scenario 1

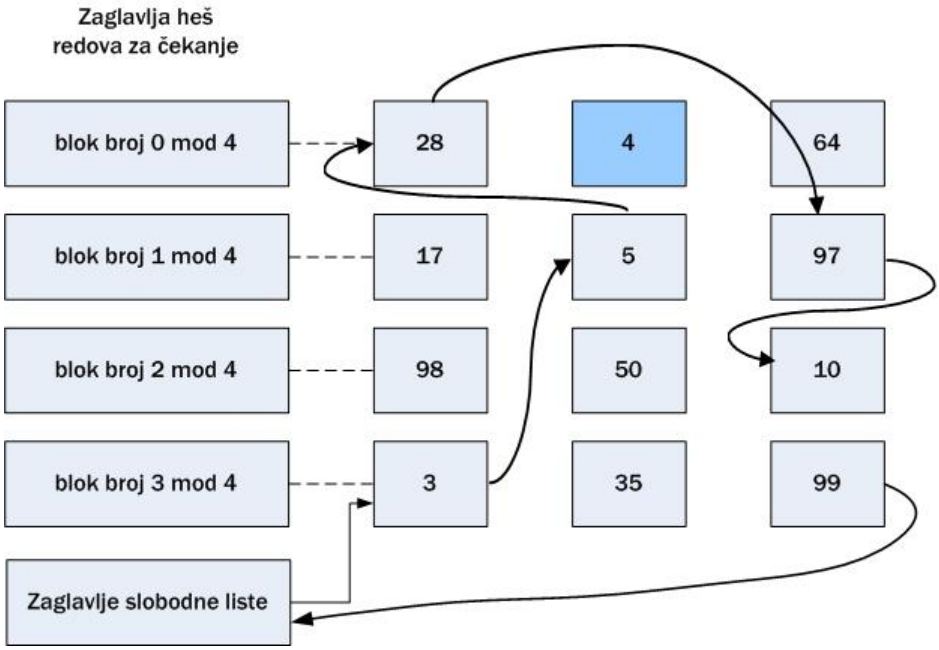
Demonstrirajmo scenario 1[2] na slici 2.13a, kada tražimo blok 4, blok 4 se nalazi u heš redu čekanja 0 i slobodan je.



Slika 2.13a. *Baferski keš u trenutku kada se traži blok 4*

Na slici 2.13b prikazan je blok 4 koji je pronađen i koji se izbacuje iz slobodne liste (ukida se blok i pokazivač se pomera).

2 Kernel pronalazi bafer u njegovom heš redu čekanja i bafer je slobodan.



Slika 2.13b. *Baferski keš nakon dodele bloka 4*

Algoritam Brelse

Pre nego što objasnimo ostale scenarije, demonstrirajmo šta se dešava u slučaju dodele bafera.

Svaki proces posle getblk algoritma dobija bafer koga kernel zaključava samo za njega. Nakon toga, proces može da se čita da se upisuje taj blok, a sve dok mu kernel drži zaključavanje (lock), nijedan proces ne može pristupiti tom baferu. Kada proces obavi svoje, bafer se mora osloboditi preko algoritma brelse. Prilikom svakog ažuriranja slobodne liste, prekidi moraju biti blokirani.

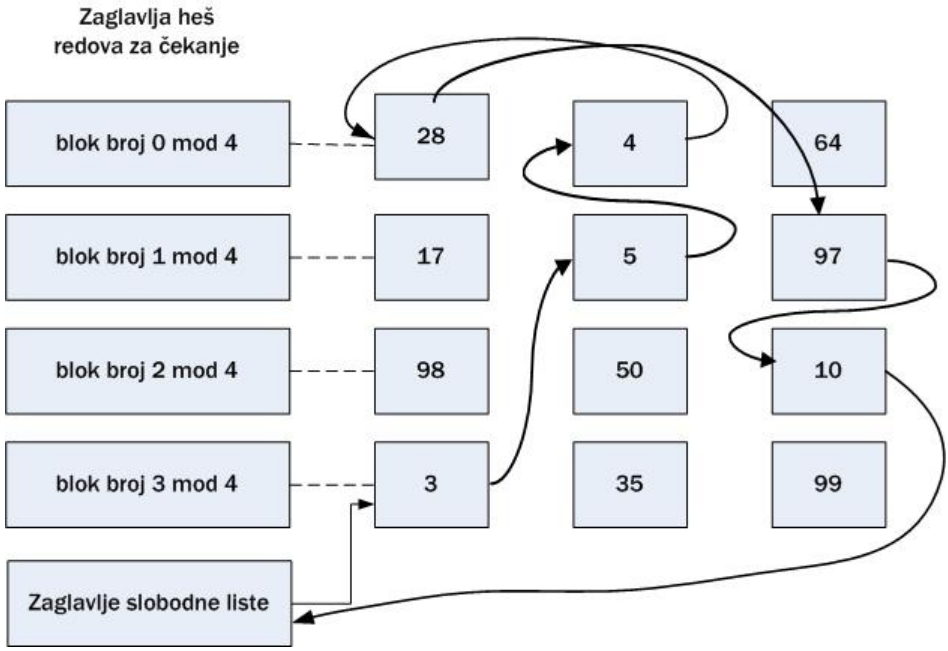
Pseudo kod za algoritam Brelse je sledeći:

```
input: locked buffer
output: none

{
 wakeup all procs: event, waiting for any buffer to become free;
 wakeup all procs: event, waiting for this buffer to become free;
 raise CPU execution level to block interrupts;
 if (buffer contents valid and buffer not old)
  enqueue buffer at the end of free list
```

```
 else
  enqueue buffer at the beginning of free list
 lower CPU execution level to allow interrupts;
 unlock (buffer);
} /*main*/
```

Scenario 2

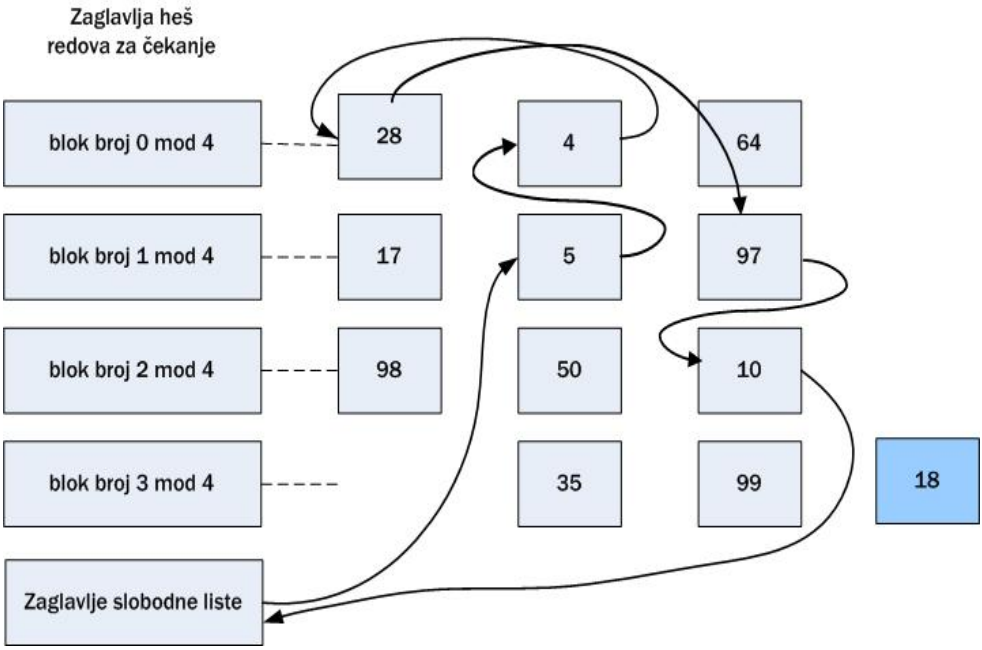
Demonstrirajmo scenario 2[3] na slici 2.14a, kada tražimo blok 18, koji nije u kešu.



Slika 2.14a. *Baferski keš u trenutku kada se traži blok 18*

Demonstrirajmo scenario 2 na slici 2.14b, kada se uklanja prvi blok iz slobodne liste (3) i dodeljuje novom bloku 18.

3 Kernel ne nalazi bafer u njegovom heš redu čekanja, zato mora da alocira bafer iz slobodne liste.



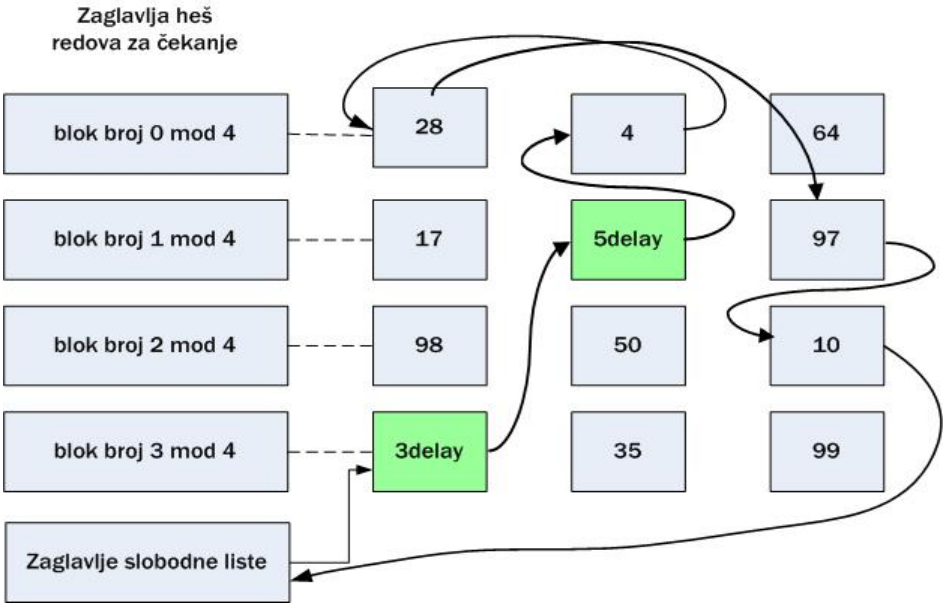
Slika 2.14b. *Baferski keš nakon dodele bloka 18*

Scenario 3

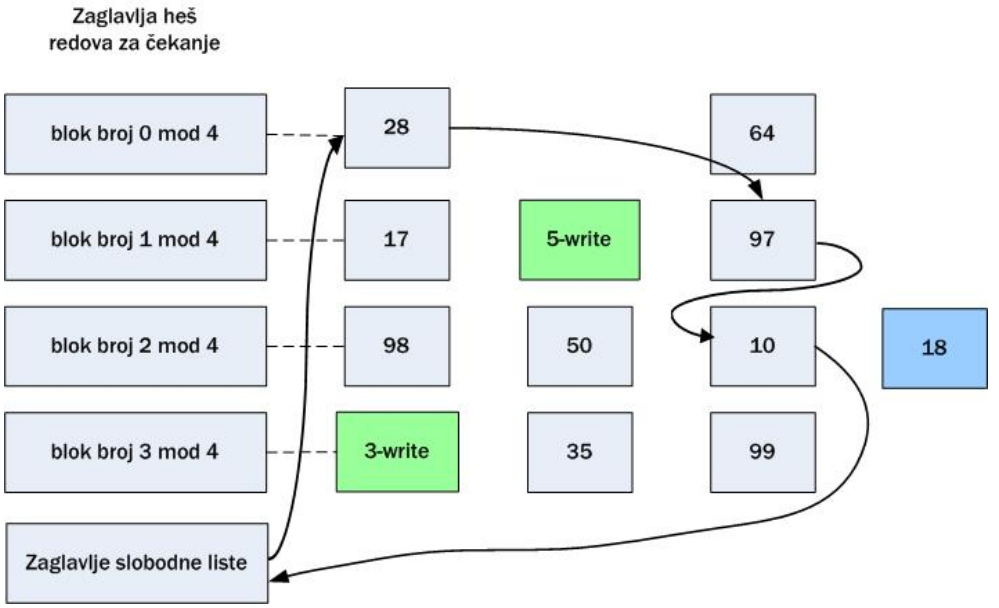
Demonstrirajmo scenario 3[4] na slici 2.15a tražimo blok 18 koji nije u kešu, ali prvo 3 i 5 moraju da se upišu na disk.

Blokovi 3 i 5 se dakle upisuju na disk, a prvi slobodni blok iz liste dodeljuje se za traženi blok 18, a to je u ovom slučaju blok 4 (slika 2.15b).

4 Kernel ne nalazi bafer u njegovom heš redu čekanja, zato mora da alocira bafer iz slobodne liste, ali taj bafer ima atribut odloženi upis (delayed write), što znači da mora prvo da se upiše na disk.



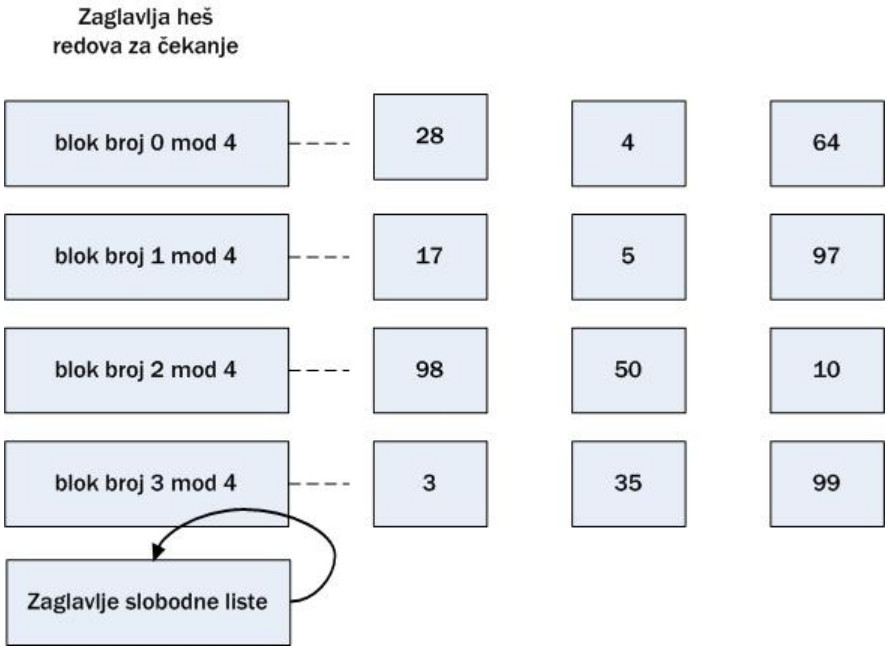
Slika 2.15b. *Baferski keš u trenutku kada se traži blok 18*



Slika 2.15b. *Baferski keš nakon dodele bloka 18*

Scenario 4

Demonstrirajmo scenario 4 na slici 2.16, kada tražimo blok 18, koji nije u kešu, pri čemu nema ni jednog slobodnog bloka zato što je slobodna lista prazna.



Slika 2.16. *Baferski keš kada se traži blok 18, ali slobodna lista je prazna*

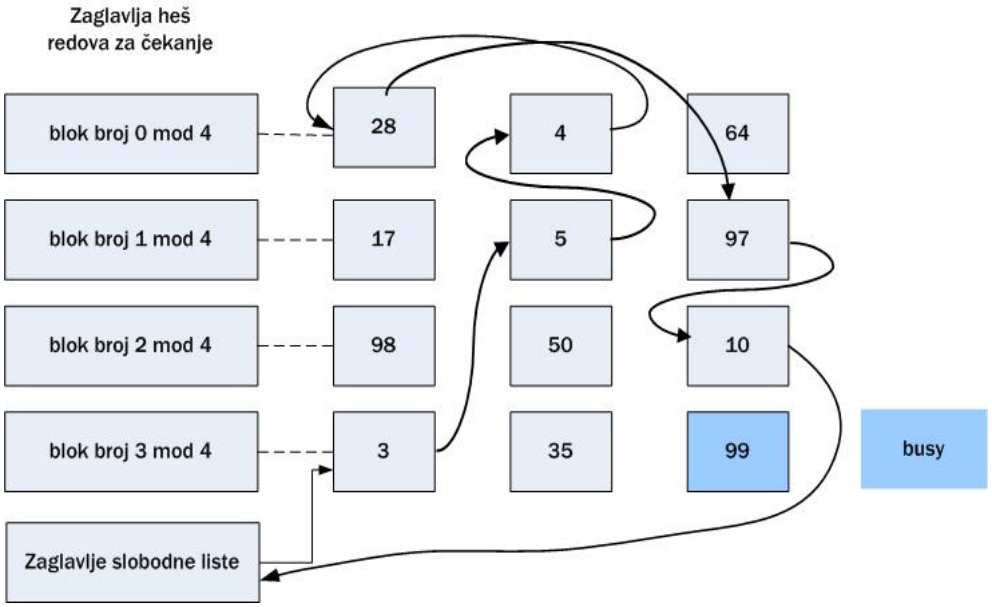
Proces mora otići na spavanje sve dok neko ne obavi algoritam brelse koji će osloboditi bar jedan bafer. Čak i u slučaju pogotka u kešu, proces mora ići na spavanje.

Scenario 5

Demonstrirajmo scenario 5[5] na slici 2.17, kada tražimo blok 99, koji se nalazi u kešu, ali je trenutno zauzet (locked).

Proces mora da čeka oslobađanje bloka.

5 Kernel nalazi bafer u njegovom heš redu čekanja, ali je taj bafer je zauzet (busy), tj. drugi proces ga je već uzeo za sebe



Slika 2.17. *Baferski keš u trenutku kada se traži blok 18, koji je zauzet*

Čitanje i upis kroz bafer

Navešćemo i ukratko opisati nekoliko algoritama za čitanje (bread i breada, koji koristi tehniku čitanja unapred) i upis (bwrite) kroz bafer keš.

Algoritam Bread

Algoritam bread je algoritam za čitanje i opisan je sa dve glavne karakteristike: keš pogodak i keš promašaj.

```
algorthm bread()  /* block */

input: file system block number
output: buffer containing data

{
  get buffer for block (algorithm getblk);
  if (buffer data valid) return (buffer); /*hit*/
  else initiate disk read; /*miss*/
  sleep (event disk read complete);
  return (buffer);
}
```

Algoritam Breada

Za povećanje performansi koristi se tehnika čitanja unapred (read-ahead)

```
algoritam breada

input: (1) file system block number for immediate read
       (2) file system block number for asynchronous read
output: buffer containing data for immediate read

{
  if (first block not in the cache)
  {
    get buffer for first block (algorithm getblk);
    if (buffer data not valid)  initiate disk read;
  }
  if (second block not in the cache)
  {
    get buffer for second block (algorithm getblk);
    if (buffer data valid) release buffer (algorithm brelse)
    else initiate disk read;
  }
  if (first block was originally in the cache)
  {
    read first block (algorithm bread);
    return buffer;
   }
  sleep (event first buffer contains valid data)
  return buffer
}
```

Ako se prvi blok ne nalazi u kešu odvija se sinhrono čitanje sa diska, koje je obično praćeno sa uspavljivanjem procesa (sleep). U tom slučaju obavlja se i asihrono čitanje unapred, uz proveru da li se taj blok već nalazi u kešu. Ako se nalazi, tada se ne obavlja čitanje u napred, ali ako se ne nalazi, poziva se getblk algoritam i ako je dodeljeni bafer prazan, inicira se asinhrono čitanje sa diska. Ukoliko dodeljeni bafer već ima validne podatke, čitanje unapred (read-ahead) se ne izvršava, već se bafer otpušta.

Algoritam Bwrite

Algoritam bwrite je algoritam za upis i opisan je sa dve glavne karakterstike: keš pogodak i keš promašaj.

```
algorthm bwrite()  /* block write*/

input: buffer
output: none
```

```
{
  initiate disk write;
  /* ovaj I/O možda se obavlja ili sihrono ili delayed */
  if(I/O synchronous)
  {
    sleep (event I/O complete)
    release buffer (algorithm brelse);
  }
  else if (buffer marked for delayed write)
    mark buffer to put at head of free list;
  }
}
```

Pretpostavimo da kernel daje nalog za upis na disk. Taj nalog može biti sinhroni upis ili odloženi upis (Delayed Write). Ako je sinhroni upis, kernel upisuje blok na disk i čeka da se I/O završi. Ako je odloženi upis, poseban status se markira za taj bafer i ne vrši se I/O ciklus, a bafer se oslobođa. Upis se inicira u getblk algoritmu po scenariju 3 kada je blok sa odloženim upisom u slobodnoj listi, a dodeljuje se novi blok iz slobodne liste. Svi ti baferi sa odloženim upisom iniciraju se za ciklus upisa, odnosno obavlja se njihovo pražnjenje. Blokovi sa odloženim upisom se zato potiskuju na početak slobodne liste kako bi se praznili na svaki promašaj u kešu.

Prednosti i nedostaci baferskog keša

Prednosti:

- Smanjuje disk saobraćaj
- Odloženi upis je izrazita prednost
- Keš je kernelska memorija koja je zaštićena, dobro sinhronizovana i nije podložna programerskim ili korisničkim greškama

Nedostaci:

- Međutransfer, prvo sa diska u keš, pa iz keša u korisnički bafer (user buffer)
- Odloženi upis može dovesti do gubitka podataka, što se ublažava sa journaling tehnikom

3

Interna reprezentacija datoteka

3.1. Uvod u internu reprezentaciju datoteka

Svaka datoteka na UNIX sistemu ima jedinstvenu inode strukturu, koja sadrži sve informacije potrebne za određivanje prava pristupa datoteci kao što su: vlasništvo, prava pristupa, veličina datoteke, lokacija datoteke na disku. Procesi pristupaju datoteci preko jasno definisanog skupa sistemskih poziva.

Inode struktura

Inode struktura je struktura koja potpuno opisuje datoteku na UNIX operativnom sistemu.

Postoje dve vrste inode struktura:

- disk inode struktura
- memorijska (in-core) inode struktura

Inode strukture postoje u statičkoj formi na disku i kernel ih čita u u memorijsku strukturu koju ćemo nazvati in-core inode struktura.

Disk Inode struktura

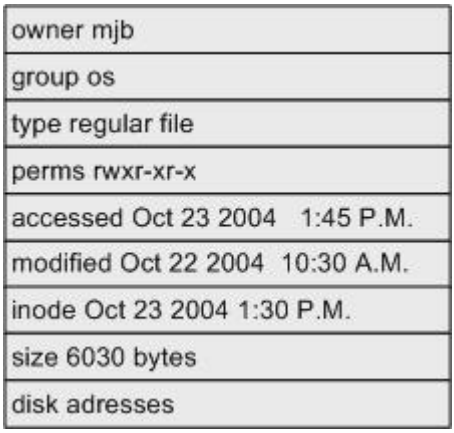
Disk inode strukture se sastoji od sledećih polja:

- Vlasnik datoteke, tj. identifikator korisnika (UID, User ID) kome ta datoteka pripada. To može biti korisnik koji je datoteku kreirao ili korisnik kome je sistem administrator dodelio vlasništvo
- Grupa kojoj pripada datoteka, tj. identifikator grupe (GID, Group ID) kojoj je ta datoteka formalno dodeljena.
- Tip datoteke. Datoteka može biti regularna, direktorijum, karakter ili blok specijalna datoteka, FIFO (pipe) datoteka.
- Prava pristupa za datoteku. Definišu se preko tri vlasničke kategorije (owner, group, other) i tri prava pristupa za njih (read, write, execute) koja imaju precizno značenje, a nezavisno se deklarišu.
- Vremena pristupa datoteke. Postoje tri karakteristična vremena – vreme poslednje modifikacije, vreme poslednjeg pristupa i vreme kad je inode struktura poslednji put modifikovana.
- Broj linkova na datoteku. Najjednostavnije rečeno – ovo polje predstavlja broj različitih imena za isti prostor na disku.

- Tabela koja opisuje alokaciju datoteke na disku, odnosno prostor na disku koji ta datoteka okupira.
- Veličina datoteke.

Primer za disk inode strukturu

Jedan tipičan primer za disk inode strukturu prikazan je na slici 3.1.

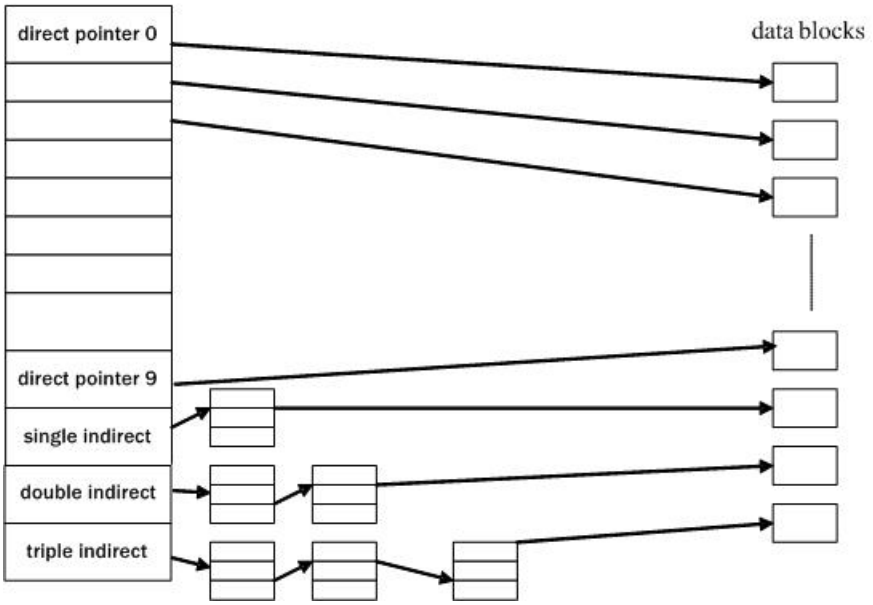


Slika 3.1. *Disk inode stuktura*

U ovom primeru imamo regularnu datoteku veličine 6030 bajtova čiji je vlasnik korisnik mjb sa pravima pristupa rwx. Datoteka pripada grupi os i svi članovi grupe imaju r i x bez w prava, dok svi ostali korisnici maju r i x bez w prava. Datoteci je poslednji put neko pristupao čitanjem i to 23. oktobra 2004 u 1:45 PM. Posledni put je neko upisao nešto u datoteku 22. oktobra 2004 u 10:30 AM. Inode je poslednji put promenjen 23. oktobra u 1:30 PM. Postoji razlika između upisa u datoteku i upisa u inode strukturu. Svaka promena u datoteci reflektuje se u inode strukturu, dok postoje promene koje idu samo u inode strukturu, a nemaju veze sa upisom u datoteku. To su recimo promena vlasništva, grupe, prava pristupa ili promena broja linkova.

Položaj na disku (layout) regularne datoteke

Svaki blok na disku ima unikatnu adresu i može pripadati samo jednoj datoteci. Svi blokovi koji pripadaju datoteci moraju se naći u inode strukturi te datoteke. Da bi inode struktura imala malu veličinu, a pri tom mogla efikasno opisivati i male i veoma velike datoteke, koristi se inode šema koja je sa 13 ulaza definisana na UNIX System V, koja je prikazana na slici 3.2. Broj ulaza može biti proizvoljan kao i broj stepena indirekcije.



Slika 3.2. *Pokazivači na blokove datoteke*

Evo primera kako se određuje maksimalna veličina datoteke za ovaj slučaj. Ako imamo 32 bitne pokazivače i sistemski blok od 1K, to znači da imamo 256 pokazivača po bloku. Prema šemi:

- sa 10 direktnih pokazivača adresiramo 10 direktnih 1K blokova = 10K
- sa jednim jednostrukin indirektnim pokazivačem adresiramo 256 blokova = 256K
- sa jednim dvostrukim indirektnim pokazivačem adresiramo 256^2 blokova = 64M
- sa jednim trostrukim indirektnim pokazivačem adresiramo 256^3 blokova = 16G

In-core inode struktura

In-core inode struktura sadrži dodatne informacije u odnosu na polja koja pripadaju disk inode strukturi:

- Status in-core inode strukture. Pokazuje: da li je inode struktura zaključana (locked), da li neki proces čeka da inode struktura postane otključana (unlocked), da li se in-core inode struktura razlikuje od svoje disk kopije kao rezultat promene polja u inode strukturi ili u samoj datoteci, da li je ta datoteka postala mount-point direktorjum.

- Opisivač sistema datoteka (FS descriptor). Zadaje se u vidu logičkog broja koji opisuje taj sistem datoteka, odnosno iz kog sistema datoteka je ta inode struktura.
- Broj Inode strukture (inode number). In-core inode struktura poseduje i ovo polje, zato što se na disku pozicija inode strukture određuje u polju fiksnog formata, preko pomeraja (offset). U memoriji, polje nije fiksno i mora se znati koja se inode struktura nalazi u in-core tabeli.
- Pokazivači na druge in-core inode struture. Kernel povezuje in-core inode strukture u heš liste (hash queue) i slobodne liste (free list), na sličan način kao kod baferskog keširanja. Heš lista se idetifikuje na osnovu sistema datoteka i inode broja. Kernel sadrži najviše jednu kopiju disk inode strukture, a ona može biti ili u heš listi ili u slobodnoj listi.
- Broj referenci, tj. RC (Reference Count). Broj referenci, pokazuje broj aktivnih instanci na tu datoteku, odnosno broj procesa koji su otvorili datoteku.

Komparacija in-core inode struktura u odnosu na baferski keš

Mnoga polja kod in-core inoda struktura su analogna poljima u baferskom zaglavlju i upravljanje je slično kao i kod baferskog keša. Kada se inode struktura zaključa, ostali procesi ne mogu da je otvore, ali oni postavljaju svoje zastavice (flag) da su zainteresovani za tu inode strukturu i da ih treba probuditi kada se inode struktura otključa. Kernel postavlja i druge zastavice koje ukazuju na razliku između disk inode strukture i njegove in-core kopije, kako bi na bazi tih flagova izjednačio stanje iste inode strukture u memoriji i na disku..

Glavna razlika između in-core inode strukture i baferskih zaglavlja ogleda se u broju referenci za in-core inode strukturu, koga uopšte nema kod baferskog zaglavlja. Svaki proces, koji pristupa in-core inode strukturi, povećava joj broj refernci, tako da inode struktura može biti u slobodnoj listi samo ako joj broj referenci padne na nulu. Samo takva inode struktura može napustiti memoriju, dok kod baferskog keša, bafer je u slobodnoj listi samo ako je otključan (unlocked).

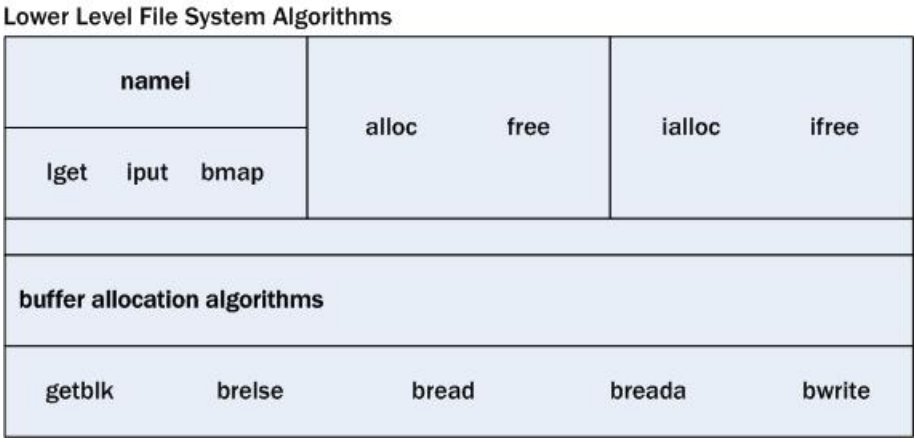
3.2. Osnovni algoritami niskog nivoa za sistem datoteka

Algoritmi niskog nivoa za sistem datoteka nalaze se na nivou iznad baferskog keša. To su:

- algoritam **iget**: vraća vrednost prethodno identifikovane in-core inode strukure, pri čemu je moguće čitanje iz inode tabele preko baferskog keša, ako se in-core inode struktura kreira po prvi put
- algoritam **iput**: otpušta in-core inode strukturu

- algoritam **bmap**: postavlja kernelske parametre za pristupanje datoteci
- algoritam **namei**: konvertuje ime datoteke u inode strukturu koristeći algoritme iget, iput i bmap.
- algoritmi **alloc** i **free**: alociraju i oslobađaju slobodne disk blokove za datoteku
- algoritmi **ialloc** i **ifree**: alociraju i oslobađaju slobodne disk inode strukture za datoteke

Slika 3.3 prikazuje algoritme sistema datoteka niskog nivoa.



Slika 3.3. *Pregled algoritama niskog nivoa za sistem datoteka*

Algoritam iget

Algoritam iget obavlja dve funkcije:

- kreira novu in-core inode strukturu, ako in-core inode struktura ne postoji
- uvećava broj referenci (reference count tj. RC) za postojeću in-core inode strukturu, ako in-core inode struktura već postoji

U prvom slučaju obavlja se čitanje iz inode tabele, preko baferskog keša.

Sledi prikaz pseudo koda algoritma za alokaciju in-core inodova, algoritma iget:

```
algorithm iget

input: file system and inode number
output: locked inode
```

```
{
  while (not done)
  {
    if(inode in inode cache)
    {
      if(inode locked)
      {
        sleep (event inode becomes unlocked);
        continue; /* loop back to while*/
      }
      /*special processing for mount point directory*/
      if (inode on inode free list)  remove from free list;
          /* if inode on the free list,
             in-core inode used by no processes */
      increment inode reference count;
      return (inode);
    }
    // inode not in inode cache
    if (no inodes on free list) return(error)
     // table is full, no chance
    /* creation of new in-core inode */
    remove new inode from free list;   // in inode cache
    reset  inode number in free list;  // in inode cache
    remove inode from old hash queue, place on new one;
    read inode from disk (algorithm bread);
    initialize inode (eg. reference count to 1)  (RC=1)
    return(inode)
  }
}
```

Pristupanje inode strukturama

Kernel identifikuje partikularnu inode strukturu preko broja sistema datoteka (FS number) i inode broja. Takođe, sistem datoteka alocira in-core inode strukturu preko iget algoritma koji veoma podseća na getblk algoritam. Kernel mapira broj sistema datoteka i inode broj u heš listu (hash queue) i traži inode u odgovarajućoj heš listi. U slučaju da ne može da ga nađe, alocira slobodnu inode strukturu iz slobodne liste, zaključava je i počinje čitanje disk inode stukture u svoju in-core kopiju.

Disk inode se određuje prema formuli:

- block num = ((inode number -1) / number of inodes per block) + start block of inode list)

i taj blok se čita pomoću algoritma bread.

Kako je inode tipična struktura od 64 ili 128 bajtova, pomeraj (offset) unutar bloka se određuje:

- offset = ((inode number -1) % number of inodes per block)) * size of disk inode

Kernel nezavisno manipuliše zaključavanjem inode strukture i njenim brojem referenci. Kernel zaključava inode strukturu svaki put kad treba da je zaštiti od drugih procesa. Što se broja referenci tiče, svaki sistemski poziv open inkrementira broj referenci, a svaki sistemski poziv close, dekrementira broj referenci. Obično se inode struktura ne zaključava sa posebnim sistemskim pozivom, zaključavanje i odključavanje inode strukture obavljaju se često u toku samog sistemskog poziva.

Oslobađanje inode strukture, algoritam iput

Kada kernel oslobađa inode strukturu, prvo se dekrementira broj referenci za tu in-core inode strukturu. Kada broj referenci padne na nulu, kernel upisuje in-core inode strukturu na disk inode strukturu, samo u slučaju da ima promena. Kernel takođe može osloboditi blokove podataka datoteke, što je u suštini brisanje datoteke, ako broj linkova padne na nulu. U svakom slučaju takva in-core inode struktura ide u slobodnu listu, a to znači da je In-core inode struktura i dalje u memoriji, ali se može izbaciti ako nema više mesta u inode keš memoriji.

Sledi algoritam za oslobađanje in-core inode struktura, iput algoritam:

```
algorithm iput /* release an in-core inode*/

input: pointer to in-core inode
output: none

{
  lock inode if not already locked;
  decrement inode reference count;
  if(reference count == 0)
  {
    if(link count == 0)
    {
      free disk blocks for file (algorithm free);
      set file type to 0;
      free inode (algorithm ifree)
    }
    if (file accessed or inode changed or file changed)
       update disk inode;
    put inode on free list;
    }
  release inode lock;
}
```

Konverzija logičkog u fizički blok, algoritam bmap

Algoritam bmap obavlja sledeću konverziju: na bazi datoteke, odnosno njene inode strukture i logičkog pomeraja (byte offset), određuje se blok u sistemu datoteka, koji odgovara tom logičkom pomeraju.

Demonstrirajmo kôd za algoritam bmap:

```
algorithm bmap
/* block map of logical file byte offset to file system block*/

input:  (1) inode
        (2) byte offset
output: (1) block number in file system
        (2) byte offset into block
        (3) bytes of I/O in block
        (4) read ahead block number
{
  calculate logical block number in file from byte offset;
  calculate start byte in block for I/O;          /* output 2*/
  calculate number of bytes to copy to user;      /* output 3*/
  check if read – ahead applicable, mark inode;   /* output 4*/
  determine level of indirection;
  while (not at necessary level of indirection)
  {
    calculate index into inode or indirect block from
      logical block number in file;
    get disk block number from inode or indirect block;
    release buffer from previous disk read, if any (brelse);
    if (no more levels of indirections) return (block number);
    read indirect disk block (algorithm bread)
    adjust logical block in file according to
      level of indirection;
  }
}
```

Moguće su null vrednosti pokazivača, što znači da proces nikada nije ništa upisao na tom pomeraju, tj. blokovi ostaju na nuli i ne troše prostor na disku.

BSD uvodi veće sistemske blokove kao što su 4K, 8K i uvodi pojam blok/fragment, pri čemu jedan sistemski blok može sadržati fragmente koji pripadaju različitim datotekama. To zahteva modifikaciju inode strukture.

Direktorijumi

To su specijalne datoteke koje imaju značajne uloge za UNIX stablo. Sastoje se od specijalnih struktura koje se nazivaju FCB (file control block), od kojih svaki FCB odgovara jednoj datoteci, a sadrži ime datoteke i inode strukturu koja joj pripada. Na primer, na UNIX System V postoji maksimum 14 bajtova za ime i 2 bajta za inode strukturu.

Na slici 3.4 je prikazan isečak etc direktorijuma, pri čemu prvi ulaz (.) predstavlja sam taj direktorijum, drugi ulaz predstavlja roditeljsku granu, a ostalo su grane i datoteke.

Byte offset in directory	Inode number (2 bytes)	File Names
0	83	.
16	2	..
32	1798	init
48	1276	fsck
64	85	clri

Slika 3.4. *Početak /etc direktorijuma*

Mada se direktorijumi tretiraju kao datoteke, kernel ne dozvoljava direktan upis u direktorijum, sem preko specijalnih sistemskih poziva kao što su creat, mknod, link i unlink. Program mkfs kreira poseban direktorijum koji ima prva dva ulaza koji dele istu inode strukturu, a to je root direktorijum.

Konverzija imena datoteke u inode strukturu, algoritam namei

Inicijalni pristup datoteci odvija se preko njenog punog imena (path name), a to je ime sa svim roditeljskim granama. Pristup preko punog imena se dešava u sistemskim pozivima kao što su open, chdir ili link. Kernel interno radi sa inode strukturama, pre nego sa punim imenima, zato što su inode strukture brojevi, a puna imena su nizovi karaktera različitih veličina. Algoritam namei konvertuje puno ime datoteke u njenu inode strukturu preko koje se pristupa datoteci. Algoritam namei razbija puno ime na jednu pojedinačnu komponentu iz imena, u jednom trenutku, konvertujući to ime u njegovu inode strukturu, tako što nalazi FCB strukturu tog imena u roditeljskom direktorijumu.

Svaki proces ima svoj tekući direktorijum, čija se inode struktura pamti u u-području za taj proces. Proces dobija tekući direktorijum od svog procesa roditelja, a može ga promeniti preko sistemskog poziva chdir. Sva puna imena datoteka počinju od tekućeg direktorijuma, osim ako ne počinju sa vodećim /, koja ukazuje da ime počinje od root

direktorijuma. Kernel identifikuje početak punog imena datoteke, pa na bazi tog početka za dalje pretraživanje koristi ili u-područje koje sadrži tekući direktorijum ili globalnu promenljivu u kojoj se čuva root inode struktura.

Algoritam namei koristi među-inode strukture ili srednje ili working inode strukture, a ima iterativnu petlju u čijem svakom prolazu se analizira jedna grana, za koju korisnik mora imati r i x pravo.

```
algorithm namei /* convert path name to inode*/

input: path name
output: locked inode

{
  if (path name starts from root)
    working inode = root inode (algorithm iget);
  else
    working inode = current directory inode (algorithm iget);
  while (there is more path name)
  {
    read next path name component from input;
    verify that working inode is of directory,
      access permissions OK;
    if (working inode is of root and component is "..") continue;
     /* loop back to while */
    read directory (working inode) by repeated use of
      algorithms bmap, bread and brelse
    if (component matches an entry in directory (working inode))
    {
      get inode number for matched component;
      release working inode (algorithm iput);
      working inode = inode of matched component (algorithm iget)
    }
    else
      return (no inode);
  }
  return(working inode);
}
```

Kernel obavlja linearno pretraživanje datoteka koje se nalaze u radnoj (working) inode strukturi, pokušavajući da nađe sledeću komponentu iz punog imena, sa početkom u pomeraju nula. Preko bmap algoritma određuje se blok na disku, koji se čita preko algoritma bread, a kada se blok dobije on se otpušta preko algoritma brelse. Potom se pročitani direktorijumski blok pretražuje za komponentu iz punog imena i ako se nađe, iz njene FCB strukture dobija se njen inode broj. Zatim se oslobađa stara radna inode struktura, a uzima se nova inoda struktura preko algoritmom iget. Svi direktorijumski blokovi moraju biti pročitani sve dok se ne nađe podudaranje sa novom komponentom iz

punog imena, pri čemu se može dogoditi da se ta komponenta uopšte ne nađe.

Alokacija i oslobađanje inode struktura i blokova na disku

Opisaćemo sada superblok strukturu, s obzirom da ona igra veliku ulogu za algoritme za alokaciju i oslobađanje inode struktura i slobodnih blokova na disku

Super Blok

Superblok se sastoji od sledećih polja:

- veličina sistema datoteka
- broj slobodnih blokova u sistemu datoteka
- lista slobodnih blokova raspoloživih u sistemu datoteka
- index-pokazivač na sledeći slobodan blok u slobodnoj listi blokova (free block list)
- veličina inode tabele
- broj slobodnih inode struktura u sistemu datoteka
- lista slobodnih inode struktura u sistemu datoteka
- index-pokazivač na sledeću slobodnu inode strukturu u slobodnoj inode listi
- polja za zaključavanje liste slobodnih blokova i za liste slobodnih inode struktura
- zastavica zaprljanosti (dirty flag) koji ukazuju da je superblok modifikovan i da mora da se sravni sa diskom

Dodeljivanje inode strukture za novu datoteku, algoritam ialloc

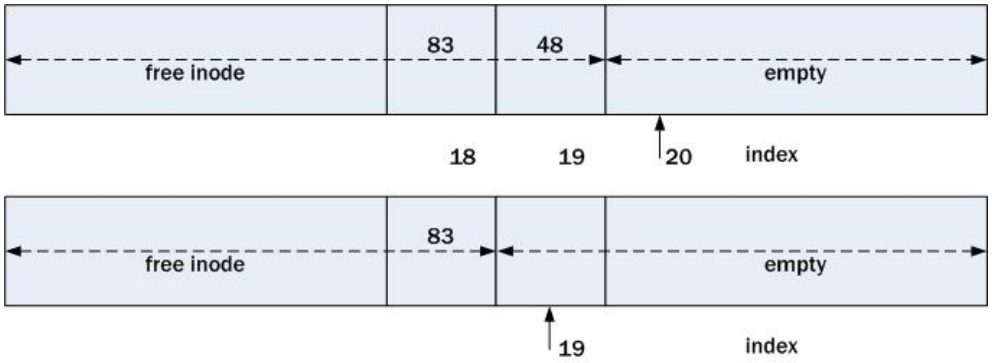
Algoritam ialloc obavlja dodeljivanje disk inode strukture za novo kreiranu datoteku. Sistem datoteka sadrži linearnu listu slobodnih inode struktura, pri čemu je inode struktura slobodna, ako je njen sadržaj nula. Kada se kreira nova datoteka mora se naći slobodan inode, a to zahteva intenzivno pretraživanje inode tabele što može dugo da traje zbog višestrukih disk čitanja. Da bi se to ubrzalo, superblok sadrži keširano polje, koje sadrži listu slobodnih inode struktura.

```
algorithm ialloc /* allocate inode */

input: FS
output: locked inode

{
  while (not done)
  {
    if (super block locked)
    {
      sleep (event super block becomes free);
```

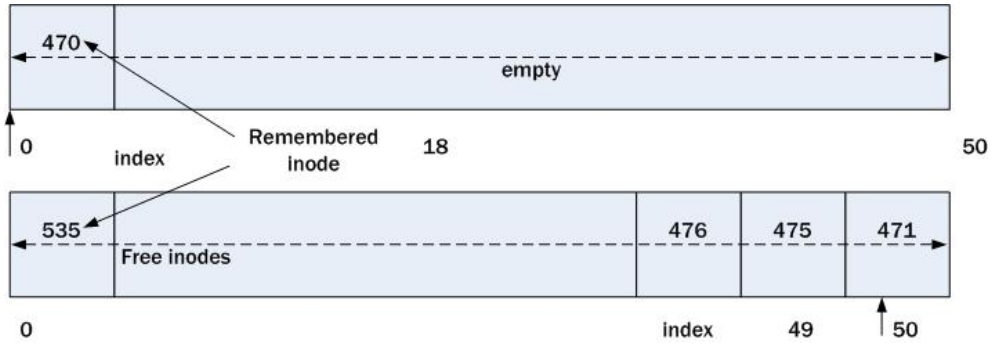
```
        continue; /*loop back to while*/
      }
      if (inode list in superblock is empty)
      {
        lock super block;
        get remembered inode for free inode search;
        search disk for free inodes until super block full,
          or no more free inodes (algorithms bread i brelse)
        unlock superblock;
        wake up (event superblock becomes free);
        if (no free inodes found on disk) return (no inode);
        set remembered inode for next free inode search;
      }
      /* there are inode in superblock inode list*/
      get inode number from super block list;
      get inode (algorithm iget);
      if (inode not free after all) /* !!! */
      {
        write inode to disk;
        release inode (algorithm iput);
        continue /* while loop*/
      }
      /* inode is free*/
      initialize inode;
      write inode to disk;
      decrement FS free inode count;
      return(inode);
    }
  }
```

Kernel prvo mora da proveri da li je superblok zaključan, zato što se u njemu nalazi slobodna inode lista. Ako lista nije prazna, kernel uzima sledeću inode strukturu, alocira novu in-core inode strukturu (algoritam iget), popunjava obe inode strukture (disk inode i in-core inode) i vraća zaključanu inode strukturu. Ako je superblok lista prazna, kernel mora da čita inode tabelu sa diska i da popunjava celu superblok listu, pri čemu pamti zadnju inode strukturu koju je detektovao i poslednje pročitani blok inode tabele (remembered inode, remembered block), da bi sledeći put pretraživanje počeo sa te tačke. Na slici 3.5 je dato dodeljivanje inode strukture iz sredine slobodne liste i iz potpuno prazne slobodne liste.

U ovom slučaju dodeljuje se inode struktura 48 koja je na indeksu 19 u listi slobodnih inode struktura.



Slika 3.5. *(a) dodeljivanje inode stukture iz sredine slobodne liste*



Slika 3.5. *(b) dodeljivanje inode stukture iz prazne slobodne liste*

U slučaju pod b, kreće se od zapamćene inode strukture koja u ovom slučaju iznosi 470. Sa diska se čita inode tabela i popunjava se svih 50 ulaza u superbloku. Index se pomera na zadnju poziciju, a zadnja zapamćena inode struktura ažurira se na broj 535, što će služiti za sledeće disk pretraživanje.

Algoritam ifree

Algoritam za oslobađanje inode strukture mnogo je jednostavniji.

```
algorithm ifree /* inode free */

input: FS inode number
output: none

{
  increment FS free inode count;
```

```
  if (super block locked) return;
  if (inode list full)
  {
    if ( inode number less than remebered inode for search)
      set remebered inode for search  = input inode number;
  }
  else
    store inode number in inode list;
  return;
}
```

Pravila za inode slobodne liste su:

- lista je sređena po opadajućim brojevima i osetno je manja nego inode tabela
- sadrži svoje ulaze koji se indeksiraju od najvišeg do najnižeg
- poslednji inode (index 1) uvek predstavlja zapamćeni (remembered) inode
- u slučaju potpuno prazne liste (sve inode strukture su slobodne), ukoliko se pojavi nova slobodna inode struktura, doći će do promene u super bloku, samo ako je novi inode broj manji od zapamćenog; u protivnom, novi broj se ne ubacuje u listu.

Primeri za ifree

Na slici 3.6 demonstriran je ifree algoritam, za slučaj pune liste.

Primećujemo da je zapamćena inode struktura 499, nije promenjena u novo-oslobođenu inode strukturu 601, zato što je 601 veći od 499.

Naravno, u svim ovim slučajevima moguća su stanja trke npr, kad kernel dodeljuje inode strukturu, kada kreira in-core inode strukturu itd. Mada se performanse smanjuju zbog zaključavanja superbloka i inode struktura, zaključavanja ipak sprečavaju stanja trke.

a) originalni super blok i lista slobodnih inode struktura

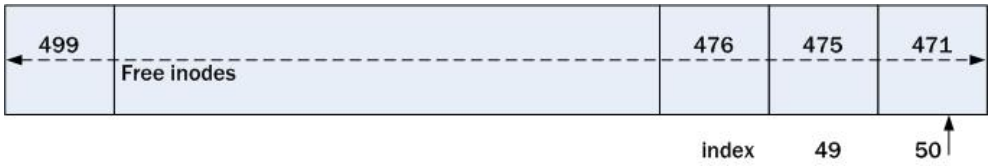
Slika 3.6. *(a) originalna slobodna lista*

b) oslobađanje inode strukture broj 499

Slika 3.6. *(b) inode se ubacuje u listu*

Primećujemo da je zapamćena inode struktura broj 535, promenjena u novo-oslobođeni 499, zato što je 499 veći od 535

c) oslobađanje inode strukture broj 601 (nema izmene liste slobodnih blokova u superbloku)



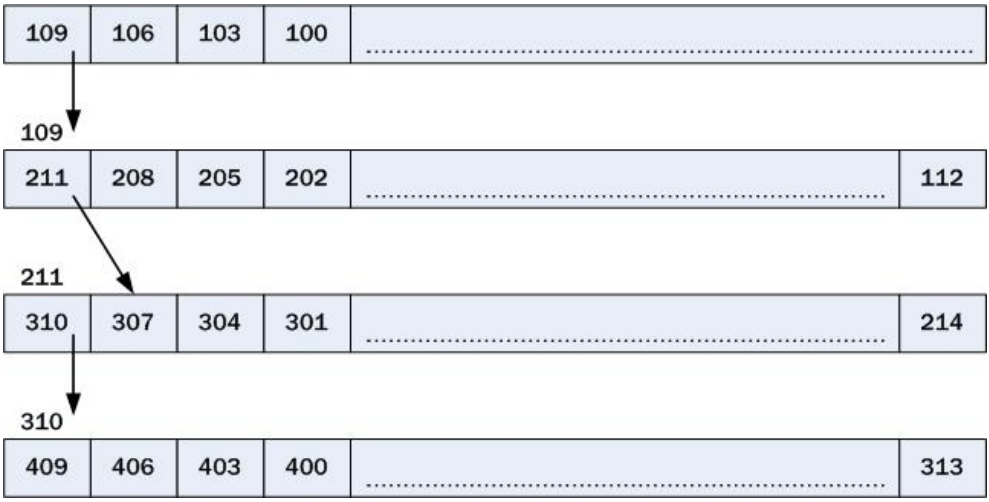
Slika 3.6. *(c) inode se ne ubacuje u listu*

Alokacije disk blokova, algoritam alloc

Svakoj datoteci mora biti prvo dodeljen blok iz liste slobodnih blokova, a svi njeni dodeljeni blokovi upisuju se u inode strukturu u direktne i indirektne pokazivače inode strukture. In-core superblok sadrži ograničenu listu slobodnih blokova sa pokazivačem na blokove na disku, koje sadrže listu sledećih slobodnih blokova. Program mkfs kreira ovu povezanu listu od slobodnih blokova čiji je početak u superbloku, kao na slici 3.7.

Kada kernel želi da dodeli blok iz sistema datoteka, on prvo analizira keširanu listu iz superbloka i uzima prvi slobodan blok. U slučaju da je to poslednji slobodan blok iz superblok liste, dodeliće se taj blok ali se prvo prelazi na listu sa diska, čita se blok iz povezane liste i popunjava superblok lista. Dodeljenom bloku dodeljuje se mesto u baferskom kešu sa getblk algoritmom i taj blok se puni nulama.

Program mkfs nastoji da liste slobodnih blokova sadrže blokove sa sličnim adresama, naravno u cilju performansi. Međutim, ta harmonija se brzo narušava jer se free blokovi često izbacuju i vraćaju u listu sasvim slučajno. Kernel pokušava povremeno da sortira slobodne liste.



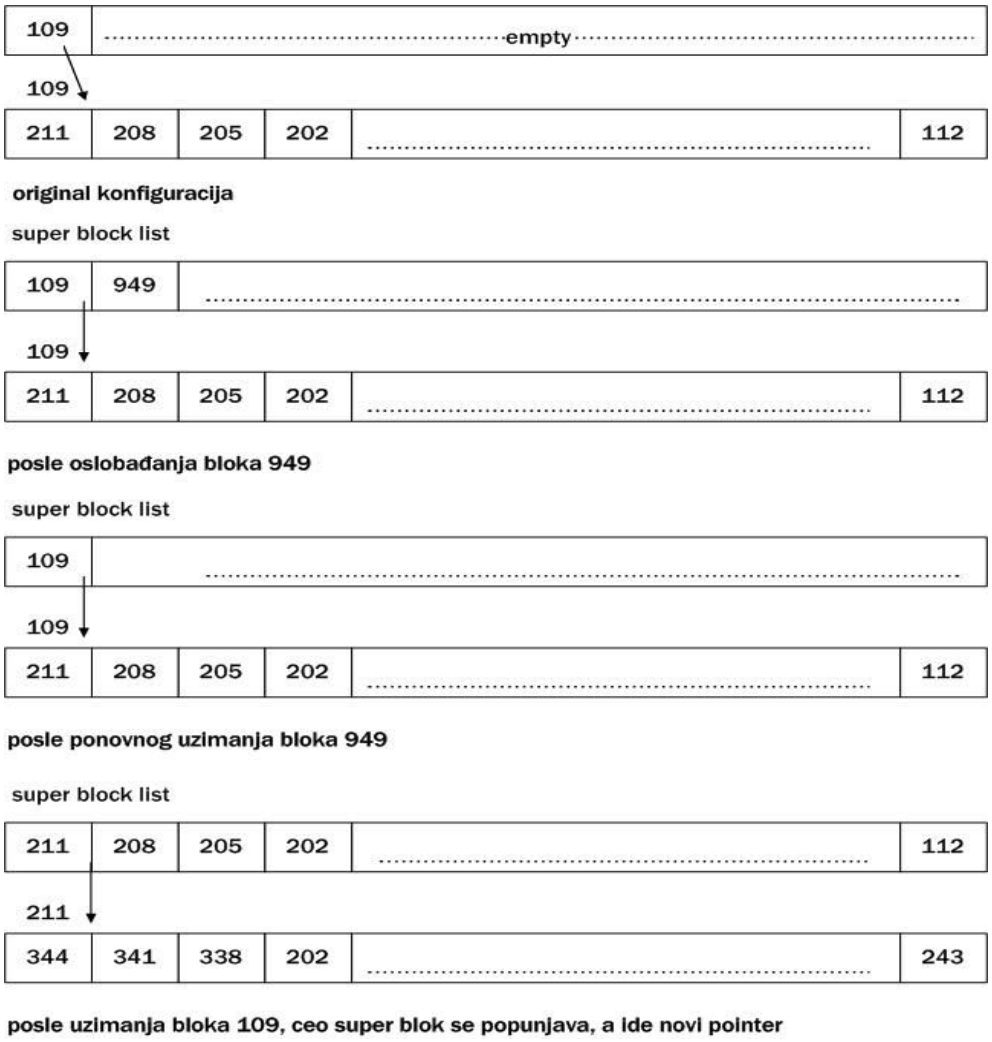
Slika 3.7. *Povezana lista slobodnih blokova*

Navodimo pseudo kôd za algoritam alloc:

```
algoritam alloc /* FS block allocation*/

input: FS number
output: buffer for new block

{
  while (super block locked)
    sleep (event super block not locked);
  remove block from super block free list
  if (removed last block from free list)
  {
    lock superblock;
    read block just taken from free list (algorithm bread);
    copy block numbers in block into super block;
    release block buffer (algorithm brelse);
    unlock superblock;
    wakeup processes (event superblock not locked);
  }
  get buffer for block removed from superblock list (getblk);
  zero buffer contents;
  decrement total count of free blocks;
  mark super block modified;
  return buffer;
}
```

Algoritam free

Algoritam free obavlja povratak bloka u slobodnu listu. Ako keširana superblok lista nije puna, blok se umeće u nju. Ako je puna, lista se upisuje u neki blok na disku, koji se stavlja u povezanu listu, superblok lista postaje prazna i u nju se smešta novo-oslobođeni blok. Taj slučaj je prikazan na slici 3.8.



Slika 3.8. *Primer za algoritam free*

Veoma je zanimljivo primetiti da se povezane liste ne primenjuju za slobodne inode strukture. Postoje tri glavna razloga za različit tretman slobodnih listi za blokove i slobodnih listi za inode strukture:

Kernel može odrediti da li je inode strukutra slobodna ispitivanjem njenog sadržaja: ako type polje slobodno i inode struktura je slobodna. Međutim, na osnovu sadržaja bloka, ne postoji način za određivanje da li je blok slobodan. Zato je neophodno da kernel podržava celu tj punu slobodnu listu za blokove.

Disk blokovi su veoma povoljni za povezane liste zato što jedan disk blok može sadržati veliki broj pokazivača na slobodne blokove. Na drugoj strani, mnogo je veći broj blokova nego broj inode struktura.

Procesi mnogo više teže da koriste slobodne blokove nego slobodne inode strukture, tako da je mnogo bitnije optimizovati performanse za liste slobodnih blokova nego za inode liste.

Drugi tipovi datoteke

UNIX sistem podržava još dva tipa datoteka: pipe i specijalne datoteke. Pipe datoteka koja se zove još i FIFO, razlikuje se od obične datoteke po tranzijentnim podacima: kada se podaci jednom pročitaju iz pipe datoteke, ne mogu se ponovo čitati. Takođe, podaci se čitaju onim redosledom kojim su upisivani, odnosno nema promene poretka. Kernel upisuje pipe datoteku na disk na isti način kao i običnu datoteku s tim što ne koristi indirektne pokazivače, već isključivo direktne.

Specijalne datoteke su blok specijalne i karakter specijalne datoteke koje specificiraju uređaje. Njihove inode strukture ne ukazuju na podatke na disku, već njihova inode struktura umesto veličine, sadrži dva broja: glavni (major) i sporedni (minor) broj. Glavni broj selektuje klasu uređaja kao što je disk, a sporedni broj ukazuje uređaj unutar klase.

Osnovni sistemski pozivi u radu sa datotekama

4.1. Pregled sistemskih poziva za rad sa datotekama

Počećemo od sistemskih poziva za postojeće datoteke kao što su open, read, write, lseek i close, a zatim demonstrirati sistemske pozive za kreiranje novih datoteka kao što su creat i mknod. Potom ćemo objasniti:

- sistemske pozive koji manipulišu inode strukturama i sistemom datoteka (FS) kao celinom kao što su: chdir, chroot, chown, chmod, stat, fsstat
- napredne sistemske pozive: pipe i dup koji su značajni za implementaciju pipeline komandi u komandnom interpreteru (shell).
- mount i umount sistemske pozive
- link i unlink sistemske pozive

Uvešćemo i kernelske strukture podataka:

- tabela datoteka FT (file table) kao globalna kernelska tabela otvorenih datoteka koja se odnosi na sve procese
- UFDT (user file descriptor table) kao korisnička tabela deskriptora datoteka koja se odnosi samo na taj proces
- tabela aktiviranih sistema datoteka MT (mount table), kao tabela aktivnih sistema datoteka

Sistemski pozivi se mogu klasifikovati u sledeće kategorije:

- Sistemski pozivi koji vraćaju deskriptore datoteka koje mogu da koriste drugi sistemski pozivi
- Sistemski pozivi koji koriste namei algoritam za razbijanje punog imena datoteka (path name)
- Sistemski pozivi koji dodeljuju i oslobađaju inode strukture preko algoritama ialloc i ifree
- Sistemski pozivi koji postavljaju ili menjaju atribute datoteke
- Sistemski pozivi koji obavljaju I/O operacije preko algoritama alloc, free i baferskih alokacionih algoritama
- Sistemski pozivi koji menjaju strukturu sistema datoteka
- Sistemski pozivi koji omogućavaju procesu da promeni svoj izgled stabla datoteka

Na slici 4.1 dat je pregled sistemskih poziva klasifikovanih po kategorijama.

<table>
<tr><th>Return File Descriptor</th><th>Use of namei</th><th>Assign Inodes</th><th>File Attributes</th><th>File I/O</th><th>File Sys Structures</th><th>Tree Manipulation</th></tr>
<tr><td>open
stat
dup
pipe
close</td><td>open stat
creat link
chdir unlink
chroot mknod
chown mount
chmod umount</td><td>creat
mknod
link
unlink</td><td>chown
chmod
stat</td><td>read
write
lseek</td><td>mount
umount</td><td>chdir
chown</td></tr>
<tr><td rowspan="4"></td><td colspan="5">Lower Level FS algorithm</td><td rowspan="4"></td></tr>
<tr><td colspan="2">namei
iget iput</td><td>ialloc
ifree</td><td colspan="2">alloc free bmap</td></tr>
<tr><td colspan="5">buffer allocation algorithm</td></tr>
<tr><td colspan="5">getblk brelse bread breada bwrite</td></tr>
</table>

Slika 4.1. *Pregled sistemskih poziva za sistem datoteka*

4.2. Sistemski pozivi za otvaranje i manipulaciju sa datotekama

Obradićemo sledeće sistemske pozive:

- open
- read
- write
- lseek
- close

Open

Sistemski poziv open je prvi korak koji proces mora obaviti da bi pristupio datoteci. Sintaksa za sistemski poziv open je:

```
fd = open (pathname, flags, modes)
```

Ulazni parametar **pathname** predstavlja ime datoteke, ulazni parametar **flags** ukazuju na tipove otvaranja (za čitanje ili upis) a ulazni parametar **modes** daje prava pristupa u slučaju da se kreira nova datoteka.

Sistemski poziv open na izlazu daje jednu celobrojnu vrednost koji se naziva korisnički deskriptor datoteke i koji se kasnije koristi za druge operacije nad datotekama kao što su:

- čitanje (reading)
- pisanje (writing)
- pozicioniranje (seeking)
- dupliranje file deskriptora (duplicating of file descriptors)
- postavljanje parametara datoteke (setting a file I/O parameters)
- određivanje statusa datoteke
- zatvaranje datoteke

Pseudo kôd za sistemski poziv open je sledeći:

```
algorithm  open

inputs: {filename,
         type of open,
         file permissions(for creation type of open)}
output: file descriptor

{
  convert file name to inode (algorithm namei)
    // iget produce in-core inode
  if(file does not exist  or not permitted access) return(error);
  allocate file table entry for inode, initialize count, offset;
    // #FT
  allocate user file descriptor entry, set pointer to file table
    entry; #UFTD
  if (type of open specifies truncate file) free all file blocks
    (algorithm free)
  unlock(inode);
  return(user file descriptor);
}
```

Kernel pretražuje sistem datoteka za zadatu datoteku preko algoritma namei. U slučaju da namei nađe datoteku, tako što prođe kroz sve grane iz njene putanje (pathname), proverava joj prava pristupa za proces, a zatim kreira ili modifikuje inode strukturu u memoriji i otvara jedan ulaz u tabeli datoteka FT za tu otvorenu datoteku. Taj ulaz u tabeli FT sadrži sledeće informacije:

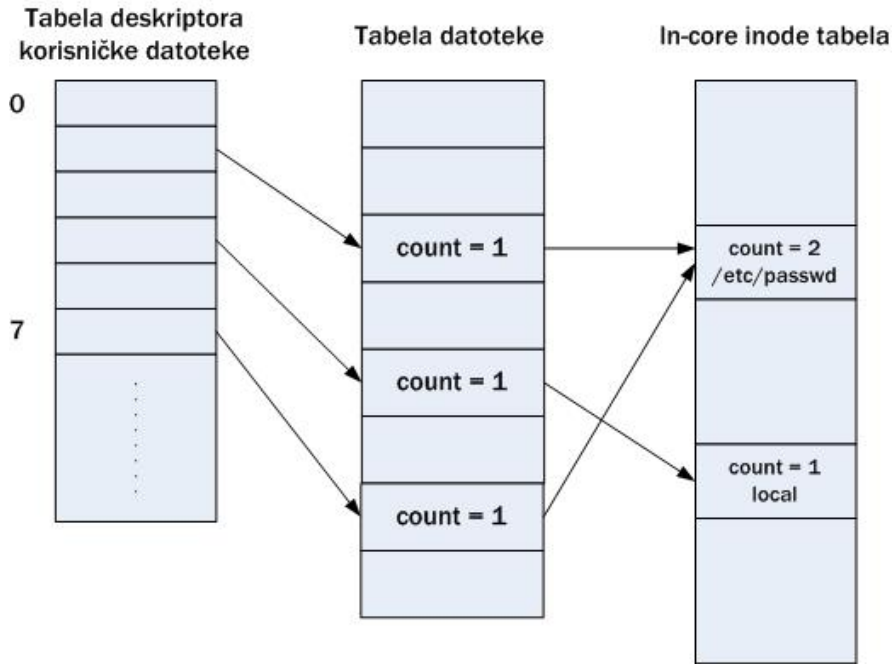
- pokazivač na in-core inode strukturu

- pomeraj unutar datoteke gde kernel očekuje da će se dogoditi sledeći upis ili čitanje. Prilikom sistemskog poziva open, kernel ovo polje postavlja na nulu, dok će kasnije akcije sa datotekom to polje modifikovati. Proces može otvoriti datoteku u "write-append" modu, a u tom slučaju kernel inicijalizuje pomeraj na kraj datoteke , odnosno pomeraj je jednak veličini datoteke (filesize). Pored globalne tabele datoteka FT, kernel inicijalizuje i tabelu UFDT, a pokazivač na nju postavlja u u-područje. Naravno, UDFT mora pokazivati na datoteku u tabeli datoteka FT.

Pretpostavimo pojavu tri uzastopna sistemska poziva open:

```
fd1 = open("/etc/passwd", O_RDONLY)
fd2 = open("/etc/local", O_RDWR)
fd3 = open("/etc/passwd", O_WRONLY)
```

Svaki sistemski poziv open, procesu vraća poseban deskriptor datoteka fd i kreira po jedan ulaz u UFTD tabeli u koji se upisuje fd i jedan ulaz u tabeli datoteka FT. Odgovarajući ulaz u tabeli UFDT pokazuje na odgovarajući ulaz u tabeli datoteka FT, čak i u slučaju da je datoteka otvorena više puta. Sve otvorene instance za istu datoteku ukazuju na jedinstvenu in-core inode strukturu. Situacija nakon ova tri sistemska poziva open prikazana je na slici 4.2.

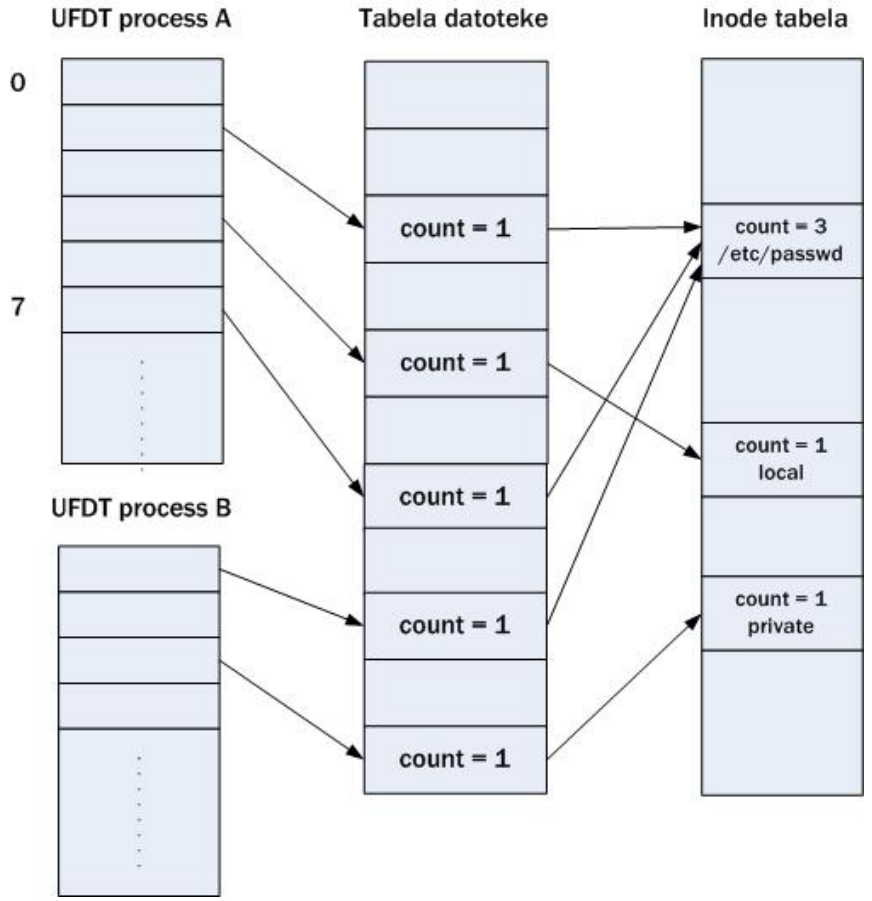


Slika 4.2. *Primer za jedan proces a 3 sistemska poziva open*

Pretpostavimo da drugi proces izvršava sledeći kôd

```
fd1 = open("/etc/passwd", O_RDONLY)
fd2 = open("private", O_RDONLY)
```

Situacija nakon ova dva dodatna sistemska poziva open prikazana je na slici 4.3.



Slika 4.3. *Primer za dva procesa a 5 sistemskih poziva open*

Sa slike vidimo da svaki open ima jedinstveni ulaz u tabeli UFDT za proces i jedinstveni ulaz u kernelovoj tabeli datoteka FT, a samo jednu in-core inode strukturu za svaku otvorenu datoteku.

UFDT ulazi bi mogli da sadrže pomeraj (offset) za sledeće I/O operacije nad datotekom i pokazivač na in-core inode strukturu, tako da tabela datoteka FT formalno

ne bi trebalo da postoji. Međutim njeno prisustvo je opravdano, jer omogućava deljenje pokazivača kao i deskriptora datoteke fd, što se koristi u sistemskom pozivu dup i fork. Prva tri korisnička fd su (0, 1 i 2) i predstavljaju standarni ulaz, standarni izlaz i izlaz za greške.

Sistemski poziv read

Sintaksa za sistemski pozivi read je:

```
number = read(fd, buffer, count)
```

pri čemu je:

- **fd** je file deskriptor dobijen od sistemskog poziva open
- **buffer** je adresa memorijskog bafera koje je proces dobio za čitanje
- **count** je broj bajtova koji se čita iz datoteke
- **number** je broj koji se upravo pročitao preko ovog sistemskog poziva read

```
algorirthm read

inputs: {user file descriptor,
         address of buffer in user process,
         number of byte to be read}
output: count of bytes copied into user space

{
  get file table entry from user file descriptor;
  check file accessibility;
  set parameters in u area for user address,
      byte count, I/O to user;
  get inode from FT;  // in-core inode
  lock inode;
  set byte offset in u area from file table offset;
  while (count not satisfied)
  {
    convert file offset to disk block (algorithm bmap)
    calculate offset into block, number of byte to read;
    if (number of bytes to read is 0) break; /* end of file*/
    read block (algorithm breada if with read ahead,
                bread otherwise);
    copy data from system buffer to user address;
    update u area fields for file byte offset,
       read count, address to write into user space;
    release buffer /*locked in bread*/
  }
  unlock inode;
```

```
  update FT offset for next read;
  return (total number of bytes read);
}
```

Na bazi deskriptora datoteke fd, kernel prvo uzima ulaz u tabeli UFTD. Tada se postavljaju I/O parametri u u-području, koji nisu prosleđeni kao funkcijski parametri. Konkretno, u slučaju sistemskog poziva read, postavljaju se sledeći parametri: prvo da je u pitanju ciklus čitanja (read I/O), zatim broj bajtova (count), zatim korisnička bafer adresa i na kraju pomeraj koji se čita iz ulaza u tabeli datoteka FT.

Polja koja se postavljaju u u-području su:

- **Mode**. Određuje čitanje ili pisanje (indicates read or write)
- **Count**. Broj bajtova za upis ili čitanje (count of bytes to be read or write)
- **Offset**. Bajtovski pomeraj unutar datoteka (byte offset in file).
- **Address**. Ciljna adresa za kopiranje podataka u korisničkoj ili kernelskoj memoriji (target address to copy data in user or kernel memory).
- **Flag**. Indikator da li se adresa nalazi u korisničkoj ili kernelskoj memoriji (indicates if address is in user or kernel memory)

Pored pomeraja, kernel iz tabele datoteka FT čita pokazivač na in-core inode strukturu, pronalazi tu inode strukturu, zaključava je i počinje da čita datoteku kroz kernelski keš

Čitanje se odvija u petlji na sledeći način:

- na bazi pomeraja određuje se broj bloka preko algoritma bmap
- blok se čita u keš preko algoritma bread
- modifikuju se polja count i pomeraj koja pripadaju u-području
- petlja se ponavlja dok se ne pročita zahtevani broj bajtova

Na kraju, postavlja se novi pomeraj za datoteku u tabeli datoteka FT, a korišćenjem sistemskog poziva lseek taj pomeraj se može modifikovati.

Posmatrajmo primer:

```
#include <fcntl.h>
main()
{
  int fd;
  char litlbuf[20], bigbuf[1024];
  fd =  open("/etc/passwd", O_RDONLY);
  read (fd, litlbuf, 20);
  read (fd, bigbuf, 1024);
  read (fd, litlbuf, 20);
```

```
}
```

- Sistemski poziv open otvara datoteku i postavlja pomeraj na nulu.
- Prvi sistemski poziv read, čita 20 bajtova i postavlja pomeraj na 20. Pri tome u kernelski keš se smešta 1K podataka, a iz keša se čita samo prvih 20 bajtova. U u-području se postavlja count na nulu (read je zadovoljen), pomeraj na 20, a potom se ukupan broj pročitanih bajtova postavlja na 20.
- Drugi sistemski poziv read čita 1K ali u pomeraju 20, i skoro sve se nalazi u kešu (ako je malo vremena proteklo između prvog i drugog čitanja). Međutim, tu postoji samo 1004 bajtova potrebnih podatka, pa se opet ide na inode strukturu za dobijanje sledećeg direktnog pokazivača na blok, sa diska se čita sledeći blok datoteke i sada se u kešu nalaze dva bloka datoteke. U baferu bigbuf, u dve iteracije prebacuju se 1024 bajta, a novi pomeraj za datoteku se postavlja na 1044.
- Treći sistemski poziv read čita 20 bajtova na pomeraju 1044, a ti bajtovi se najverovatnije nalaze u kešu i postavlja se finalni pomeraj na 1064.

Kernel će na bazi zahtevanih bajtova za čitanje (read count) da proceni da li će da koristi algoritam bread ili breada.

Ukoliko vrednost pokazivača u inode stukturi jednak nuli, tada nema čitanja sa diska već se korisnički bafer puni nulama.

Inode struktura je zaključana sve dok traje sistemski poziv read, jer bi moglo doći do inkonzistenicje podataka.

Pored toga uvodi se pojam zaključavanja zapisa, u cilju obezbeđivanja konzistentnosti podataka kao u primeru dva procesa koja istovremeno otvoraju istu datoteku, pri čemu jedan ima dva čitanja, a drugi proces dva upisa. Šta će pročitati prvi proces, zavisi od redosleda sistemskih poziva: na primer (read 1, read 2, write 1, write 2) ili (read 1, write 1, read 2, write 2), ili (read 1, write 1, write 2, read 2). Zato se pored otvaranja datoteke uvodi zaključavanje na zapis (record), tako da jedan proces ne može otvoriti zaključanu datoteku.

Posmatrajmo primer koji ilustruje prethodno opisani slučaj :

```
#include <fcntl.h>
/*process A*/
main()
{
  int fd;
  char buf[512];
  fd = open("/etc/passwd", O_RDONLY)
  read (fd, buf, sizeof(buf)) /*read 1*/
  read (fd, buf, sizeof(buf)) /*read 2*/
}
```

```
/*process B*/
main()
{
  int fd,i;
  char buf[512];
  for (i=0; i<sizeof(buf); i++) buf[i] = 'a';
  fd = open("/etc/passwd", O_WRONLY)
  write (fd, buf, sizeof(buf)) /*write 1*/
  write (fd, buf, sizeof(buf)) /*write 2*/
}
```

Na sledećem primeru proces može otvoriti datoteku dva puta i čitati je preko dva različita deskriptora sa nezavisnim pomerajima unutar datoteke.

```
#include <fcntl.h>
main()
{
  int fd1, fd2;
  char buf1[512], buf2[512];
  fd1 = open("/etc/passwd", O_RDONLY)
  fd2 = open("/etc/passwd", O_RDONLY)
  read (fd1, buf1, sizeof(buf1)) /*read 1*/
  read (fd2, buf2, sizeof(buf2)) /*read 2*/
}
```

write

Sintaksa za sistemski poziv write je:

```
number = write (fd, buffer, count)
```

pri čemu su parametri isti kao za sistemski poziv read: **fd** je deskriptor datoteke (file descriptor) dobijen od sistemskog poziva open, **buffer** je adresa memorijskog bafera koje je proces dobio za upis, **count** je broj bajtova koji se upisuje u datoteku, a **number** je broj koji je upravo upisan preko ovog sistemskog poziva write.

Algoritam za sistemski poziv write sličan je algoritmu za sistemski poziv read, izuzev ako datoteka ne sadrži blok koji odgovara pomeraju unutar datoteke za upis. Tada kernel alocira novi blok preko algoritma alloc i podešava inode strukturu. Ako pomeraj unutar datoteke odgovara indirektnom bloku, kernel mora dodeliti više blokova koji će se koristiti za indirektne blokove pokazivača i blokove podataka. Inode struktura se zaključava za vreme sistemskog poziva write, zato što u toku ovog sistemskog poziva kernel može promeniti inode strukturu. Kada se upis završi, kernel ažurira inode pokazivače i polje za veličinu datoteke ako je došlo do promene veličine.

Kada kernel obavlja upis, ponašanje je slično kao kod read ciklusa pri čemu su moguće određene situacije. Jedna potencijalna situacija je da se vrši upis celog bloka u

jednoj u iteraciji. Druga situacija je da se u nekoj iteraciji upisuje deo bloka: tada blok prvo mora da se učita u keš blok u memoriji, posle čega se prepisuje deo bloka i na kraju se ceo blok upisuje. Treća situacija se javlja kada je prvo potrebno alocirati indirektni blok, pa tek onda izvršiti upis.

Koristi se odloženi upis (delayed write) kroz kernelski keš koji je efikasan za pipe datoteke i datoteke koje se privremeno kreiraju i brišu se ubrzo posle toga, kao što su privremene datoteke koje kreira editor teksta i briše ih. Dakle, najveći uspeh postiže se za upis za datoteke koje će živeti samo u kešu i nikada neće dostići disk, zato što se obrišu.

Podešavanje pozicije u datoteci - lseek

Korišćenje sistemskih poziva read i write obično obezbeđuje sekvencijalni pristup datoteci, dok sistemski poziv lseek omogućava slučajan pristup.

Sintaksa za lseek sistemski poziv je sledeća:

```
position = lseek (fd, offset, reference)
```

pri čemu su parametri sledeći:

- **fd** je deskriptor datoteke dobijen od sistemskog poziva open
- **offset** je relativni bajt pomeraj
- **reference** predstavlja reper u odnosu na koji se posmatra pomeraj:
  - u odnosu na početak datoteke (0)
  - u odnosu na tekuću poziciju (1)
  - u odnosu na kraj datoteke (2)
- **position** je pomeraj u kome će sledeće čitanje ili upis da se dogode

Posmatrajmo primer:

```
#include <fcntl.h>
/*process A*/
main(argc, argv)
int argc;
char *argv[];
{
  int fd, skval;
  char c;
  if (argc != 2) exit();
  fd = open(argv[1], O_RDONLY);
  if (fd == -1) exit();
  while ((skval = read(fd, &c, 1)) == 1)
```

```
    {
      printf("char %c\n",c);
      skval = lseek(fd, 1023L, 1);
      printf("new seek val %d \n",skval);
    }
}
```

Program čita svaki 1024-ti bajt datoteke, tako što prvo pročita jedan bajt, onda obavi sistemski poziv lseek za 1023. Sistemski poziv lseek praktično ne radi ništa sa datotekom, on jednostavno podešava pomeraj u tabeli datoteka FT, koji će se korisititi u sledećoj read/write instanci.

Close

Proces zatvara otvorenu datoteku kada više ne želi da joj pristupa, preko sistemskog poziva close. Sintaksa za sistemski poziv close krajnje je jednostavna:

```
close (fd);
```

Kernel obavlja close operaciju tako što obavlja manipulacije u tabeli UFDT, tabeli datoteka FT i in-core inode strukturi. Ako je broj referenci u tabeli datoteka FT veći od jedan, zbog dup ili fork sistemskog poziva, kernel će dekrementirati broj referenci u tabeli datoteka FT. Ako taj broj posle dekrementa padne na nulu, kernel oslobađa ulaz u FT i otpušta in-core inode strukturu sa algoritmom iput, koji je originalno kreiran nakon sistemskog poziva open. Ako ostali procesi takođe koriste tu in-core inode strukturu, broj referenci se dekrementira. Nakon toga, briše se ulaz i u tabeli UFDT. Kada proces obavi sistemski poziv exit, sve njegove otvorene datoteke moraju se zatvoriti.

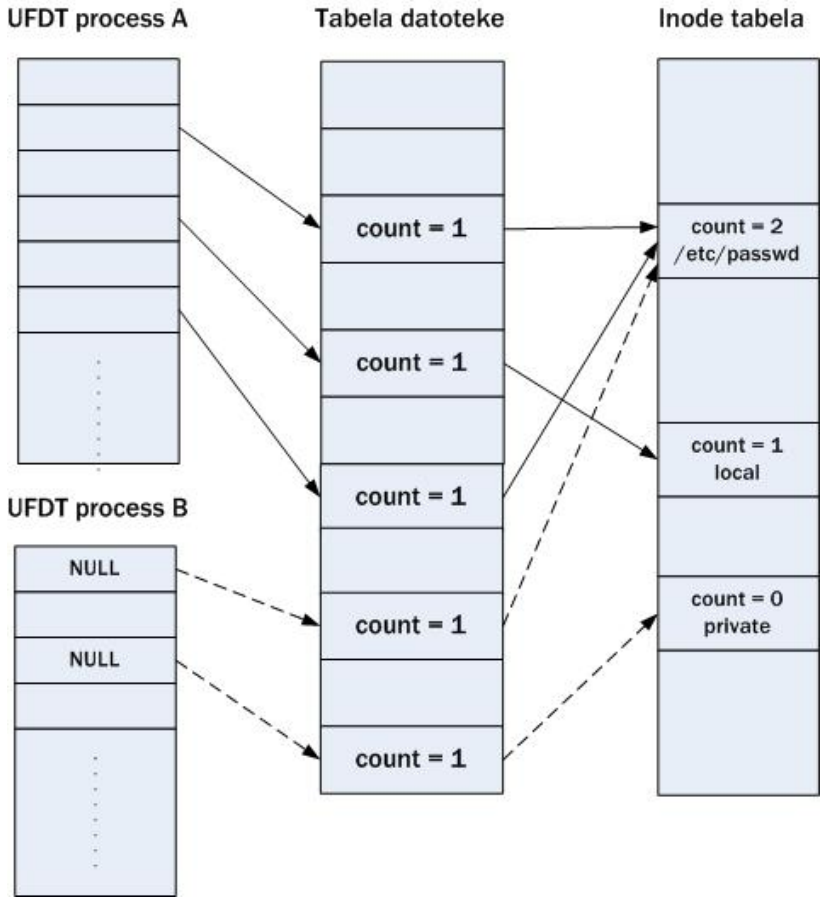
Na primer, posmatrajmo isti slučaj kao kod sistemskog poziva open, kada je proces B zatvorio sve svoje datoteke. Za in-core inode strukturu datoteke pod imenom count, broj referenci pada na nulu čime se otpušta ta inode struktura. Za /etc/passwd datoteku broj referenci pada na dva, potom se oslobađaju dva ulaza u FT koja su pripadala procesu B, a njegovi UFTD ulazi se anuliraju, kao na slici 4.4.

4.3. Kreiranje novih i specijalnih datoteka, rad sa direktorijumima i status datoteka

Obradićemo sledeće sistemske pozive:

- create
- mknod
- chdir, chroot
- chown, chmod

- stat, fsstat



Slika 4.4. *Situacija kada proces B zatvori svoje datoteke*

Kreiranje datoteka

Dok sistemski poziv open otvara postojeće datoteke, sistemski poziv create kreira novu datoteku. Sintaksa za sistemski poziv create je:

```
fd = create (pathname, modes)
```

gde svi parametri imaju isto značenje kao i kod sistemskog poziva open.

Ako takva datoteka ne postoji, kernel kreira novu datoteku nulte veličine, pod zadatim imenom i sa zadatim pravima. Ako datoteka postoji, kernel odseca datoteku na veličinu nula, pri čemu oslobađa sve blokove podataka.

```
algorithm  create

inputs: {filename,
         file permissions(for creation type of open)}
output: file descriptor

{
  get inode for file name (algorithm namei);
  if (file already exists)
  {
    if (not permitted access)
    {
      release inode (algorithm iput);
      return(error)
    }
  }
  else /*file does not exist yet*/
  {
    assign free inode from FS (algorithm ialloc);
    create new directory entry in parent directory:
      include new file name and newly assigned inode number
  }
  allocate file table entry for inode, initialize count, offset;
  if (file did exist at time of create)
    free all file blocks (algorithm free);
  unlock(inode);
  return(user file descriptor);
}
```

Sistemski poziv create može da se izvede preko sistemskog poziva open, koristeći dve zastavice: O_CREAT (create) i O_TRUNC (truncate).

Koristeći algoritam namei, kernel razvija putanju (pathname) po komponentama, dok ne naiđe na zadnju komponentu. Ukoliko u zadnjoj grani nađe odgovarajuće ime, to ime se po sadržaju mora anulirati. U protivnom, dogodiće se kreiranje novog objekta u direktorijumu, tj. novi FCB. Ukoliko u direktorijumskom bloku nema slobodnog mesta, direktorijum se mora proširiti novim blokom, što izaziva brojne akcije (alokaciju, inode promenu..). Potom se pronalazi slobodna inode struktura za novu datoteku (algoritam ialloc) koja se inicijalizuje i upisuje na disk preko bwrite algoritma. Ukoliko je datoteka postojala, kernel zahteva od procesa da ima pravo upisa (write) nad njom. U ovom slučaju proces može da je odseče na nulu, svi blokovi podataka (data) se oslobađaju (free algoritam), ali se ne menja vlasništvo i grupa datoteke.

Na kraju, inicijalizije se po jedan ulaz u tabelama UDFT i FT, kao u slučaju sistemskog poziva open.

Kreiranje specijalnih datoteka

Kreiranje specijanih datoteka, kao što su drajverske datoteke (device files), imenovane pipe datoteke (named pipes) i direktorijumi, odvija se preko sistemskog poziva mknod, koji ima sličnosti sa sistemskim pozivom create, jer dodeljuje novu inode strukturu za datoteku.

Sintaksa za sistemski poziv mknod je:

```
mknod (pathname, type and permissions, dev)
```

pri čemu je:

- **pathname** ime specijalne datoteke,
- **type** and permissions daju tip specijalne datoteke (node, pipe, directory) i prava pristupa koja joj se dodeljuju
- **dev** specificira glavni i sporedni broj za specijalne blok i karakter datoteke

```
algorithm  mknod

inputs: {filename,
         file type,
         permissions,
         major and minor device number}
output: none

{
  if (new node not name pipe and user not super user)
    return (error);
  get inode for file name (algorithm namei);
  if (new node already exist)
  {
    release parent inode (algorithm iput);
    return(error)
  }
  else  /*file does not exist yet*/
  {
    assign free inode from FS (algorithm ialloc);
    create new directory entry in parent directory:
      include new node name and newly assigned inode number;
    release parrent inode (algorithm iput);
  }
  if (new node is block or character special file)
```

```
    write major and minor numbers into inode structure;
  release new node inode (algorithm iput);
}
```

Kernel u direktorijumskoj strukturi traži ime specijalne datoteke koju treba da kreira. Ukoliko ime ne postoji, dodeljuje se nova inode struktura, a u direktorijumu se upisuje novi FCB. U inode strukturi upisuju se tip datoteke (blok ili karakter specijalna datoteka, pipe datoteka, direktorijum) i prava pristupa. Ukoliko je u ptanju blok ili karakter specijalna datoteka, u inode strukturu upisuju se glavni i sporedni broj. Za direktorijum se mora alocirati blok podataka koji se popunjava sa prva dva ulaza: sama ta grana (.) i roditeljska grana (..).

Promena direktorijuma

Prilikom podizanja sistema, prvi proces koji se kreira, uzima root direktorijum za svoj tekući direktorijum. Prvi proces to postiže pošto obavlja algoritam iget za korenski direktorijum, pa ga čuva u u-području kao svoj tekući direktorijum, a potom obavlja otključavanje inode strukture (algoritam iput). Kada proces sistemskim pozivom fork kreira svoje dete, novi proces nasleđuje tekući direktorijum od procesa roditelja, a kernel povećava broj referenci za inode strukturu tekućeg direktorijuma.

Algoritam chdir menja tekući direktorijum procesa. Sintaksa za sistemski poziv chdir je sledeća:

```
chdir (pathname);
```

gde je **pathname** putanja tj. ime direktorijuma koji će postati novi tekući direktorijum procesa. Kernel razvija putanju (pathname) po granama preko namei algoritma dok ne dođe do zadnje komponente, kada proverava prava pristupa za tu granu. Ako pravo pristupa postoji, uzima se inode struktura te grane (iget), inkremetira broj referenci za novi direktorijum, oslobađa se prethodni tekući direktorijum (algoritam iput), dekrementira se broj referenci za prethodni tekući direktorijum i upisuje se novi tekući direktorijum u u-područje procesa. Kada proces promeni tekući direktorijum, njegova inode struktura je polazna tačka za sva pretraživanja (namei) svih referenci koje ne počinju sa apsolutnom putanjom /....

Inode struktura za tekući direktorijum se otpušta i dekrementira se broj referenci samo ako nastupi novi chdir ili proces završi aktivnosti (exit).

```
algorithm change directory (chdir)

input: new directory name
output: none

{
  get inode for new directory name (algorithm namei);
  if (inode not that of directory or
      process not permitted access to file)
```

```
    {
      release inode (algorithm iput);
      return(error);
    }
    unlock inode;
    release "old" current directory inode (algorithm iput);
    place new inode into current directory slot in u-area
}
```

Promena korenskog direktorijuma (change root)

Procesi podrazumevaju da / označava root direktorijum celog UNIX stabla i to je globalna sistemska promenljiva koja sadrži inode strukturu root direktorijuma, koji se dobija preko sistemskog poziva iget prilikom podizanja UNIX-a.

Procesi mogu promeniti svoju oznaku za korenski (root) direktorijum preko sistemskog poziva chroot, što je korisno kada procesi simulairaju sopstvenu hijerarhiju sistema datoteka.

Sintaksa za sistemski poziv chroot je sledeća:

```
chroot (pathname);
```

gde je **pathname** ime direktorijuma koji će se tretirati kao korenski direktorijum procesa. Algoritmam za chroot praktično je isti kao i za chdir. Preko namei algoritma dobija se inode struktura novog korenskog direktorijima koji se upisuje u u-područje procesa. Nova inode struktura je logički root za sve reference koje počinju sa /.

Promena vlasničkih odnosa i prava pristupa

Promena vlasništva ili prava pristupa su operacije koje se obavljaju isključivo nad inode strukturom dok se sa datotekom ne radi ništa. Sintakse su sledeće:

```
chown (pathname, owner, group);
chmod (pathname, mode);
```

Da bi promenio vlasništvo, kernel konvertuje putanju (pathname) u inode strukturu preko namei algoritma. Za to je neophodno da proces ima root privilegiju ili da bude vlasnik datoteke. Nakon dobijanja inode strukture, kernel modifikuje UID i GID polje, briše set user i set group zastavice, a potom otpušta inode strukturu sa iput algoritmom.

Potpuno ista procedura radi se i za algoritam chmod, gde se modifikuju prava pristupa.

Sistemski pozivi stat i fsstat

Sistemski pozivi stat i fstat omogućavaju procesu da postavi upit za status datoteke, a upit će vratiti informaciju o tipu datoteke, vlasništvu, pravima pristupa, broju linkova, broju inode strukture i vremenima pristupa.

Sintakse su sledeće:

```
stat (pathname, statbuffer);
fsstat (fd, statbuffer);
```

gde je pathname putanja do datoteke, fd je deskriptor datoteke koji je vratio prethodni sistemski poziv open, a statbuffer je korisnički bafer za prijem podataka o statusu datoteke, pri čemu sistemski pozivi stat i fsstat pune podatke u taj bafer.

5

Napredni sistemski pozivi u radu sa datotekama

5.1. Sistemski pozivi pipe i dup

U ovoj lekciji, obradićemo napredne sistemske pozive tj. sistemske pozive koji se mogu smatrati za pozive višeg nivoa:

- sistemski pozivi pipe i dup koji su značajni za implementaciju pipeline komandi
- sistemski pozivi mount i umount
- sistemski pozivi link i unlink

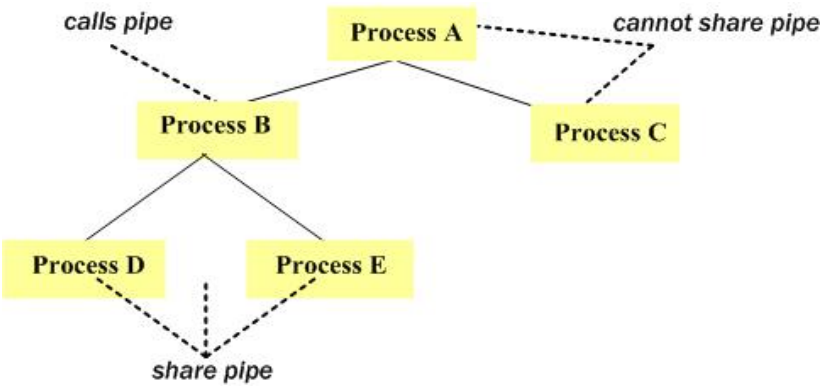
I u ovom slučaju, baziraćemo se na kernelskim strukturama podataka: tabela datoteka (FT, file table), korisnička tabela deskriptora datoteka (UFDT, user file descriptor table) i tabela aktivnih sistema datoteka (MT, mount table).

Pipe-datoteke

Pipe-datoteke dozvoljavaju prenos podataka između procesa na FIFO način, a takođe omogućavaju sinhronizaciju procesa. Postoje dve vrste pipe-datoteka:

- imenovane-pipe datoteke (named pipes)
- neimenovane-pipe datoteke (unamed pipes)

Obe vrste su identične, osim pri inicijalnom pristupu pipe-datoteci, pri čemu se za imenovane pipe-datoteke koristi sistemski poziv open kao read, write i close, dok se neimenovane pipe-datoteke kreiraju preko sistemskog poziva pipe. Pravilo je da samo deca procesa koji je otpočeo sistemski poziv pipe mogu deliti pristup neimenovanim pipe-datotekama.



Slika 5.1. *Odnosi procesa roditelja i dece u odnosu na pipe-datoteke*

Na ovoj slici 5.2 proces B poziva sistemski poziv pipe, a potom kreira dva nova procesa D i E, tako da sva tri procesa B, D i E mogu da dele pipe datoteku, dok procesi A i C ne mogu.

Na drugoj strani, svi procesi mogu pristupati imenovanoj pipe-datoteci bez obzira na njihove međusobne odnose.

Sistemski poziv Pipe

Sintaksa za sistemski poziv pipe je sledeća:

```
pipe (fdptr);
```

gde je **fdptr** pokazivač na celobrojno polje od dva elementa koje će sadržati dva deskriptora datoteka za čitanje i upis pipe-datoteke. Kako je pipe-datoteka u stvari datoteka koja ne postoji pre korišćenja, prilikom njenog kreiranja kernel mora dodeliti jednu inode strukturu. Kernel takođe dodeljuje par deskriptora datoteka (deskriptor za čitanje iz pipe-datoteke i deskriptor za upis u pipe-datoteku). Za razliku od obične datoteke gde ide samo jedan ulaz, za pipe-datoteku se dodeljuju dva ulaza u tabelu datoteka FT. Čim se kreira pipe-datoteka, svi sistemski pozivi (read, write) važe kao i u slučaju obične datoteke.

```
algorithm  pipe

input: none
output: {read file descriptor,
         write file descriptor}

{
  assign new inode from pipe device (algorithm ialloc);
  allocate FT entry for reading, another for writing;
  initalize FT entries to point to new inode;
  allocate 2 UDFT: for reading and for writing,
    initialize to point to respective FT entries;
  set inode RC to 2;
  initialize count of inode readers, writers to 1;
}
```

Kernel dodeljuje inode strukturu na pipe uređaju (pipe device) preko algoritma ialloc, pri čemu je pipe uređaj onaj sistem datoteka u kome se dodeljuje inode struktura i blokovi podataka za tu pipe-datoteku. Kada je pipe datoteka otvorena, kernel ne obavlja ponovnu dodelu (reassign) pipe inode strukture i blokova podataka. Kernel zatim dodeljuje dva ulaza u tabeli datoteka (FT) ulaza: jedan ulaz za read deskriptor i jedan ulaz za write deskriptor i ažurira informacije u in-core inode strukturi. Svaki FT ulaz čuva informaciju o broju otvorenih instanci za čitanje i upis pipe-datoteke, a inicijalno se taj broj postavlja na jedan. U in-core inode strukturi, brojač referenci pokazuje koliko puta je pipe-datoteka otvorena a inicijalno se postavlja na dva. Takođe, inode struktura čuva informaciju o bajt pomeraju koji određuje mesto sledećeg upisa ili čitanja kroz pipe-

datoteku. Tako, dolazimo do osnovne razlike između pipe datoteke i obične datoteke: kod pipe datoteke, pomeraj se nalazi u in-core inode strukturi, kod obične datoteke radi se o pomeraju u FT ulazu. Pošto sistemski poziv lseek ne može da se primenjuje na pipe-datoteku, proces ne može da podešava pomeraj kao u slučaju obične datoteke. Dakle, pristup podacima u pipe-datoteci ne može biti slučajan, već isključivo sekvencijalan, tj. na FIFO način.

Imenovani pipe

Imenovana pipe-datoteka je datoteka čija je semantika potpuna ista kao kod neimenovane pipe-datoteke, osim što ima FCB u direktorijumu i pristupa joj se po putanji (pathname). Imenovane pipe-datoteke se otvaraju kao obične datoteke i permanetno postoje u sistemu datoteka, brišu se sistemskim pozivom unlink, dok su neimenovane pipe-datoteke tranzijentne tj. traju sve dok ima procesa koji ih koriste. Kada svi procesi završe sa korišćenjem pipe-datoteka, kernel oslobađa inode strukturu odnosno briše neimenovanu pipe-datoteku.

Algoritam za otvaranje imenovanih pipe-datoteka je skoro isti kao u slučaju obične datoteke. Međutim, pre završetka sistemskog poziva open, kernel inkremetira read i write broj referenci u inode strukturi, ukazujući na broj procesa koji su otvorili imenovanu pipe-datoteku za čitanje i upis. Proces koji otvara imenovanu pipe-datoteku za čitanje ostaje uspavan sve dok drugi proces ne otvori pipe-datoteku za upis i obrnuto. Kernel će probuditi proces koji je uspavan na pipe-datoteci, kada se dogodi neki događaj upisa ili čitanja zavisno od uzroka uspavljivanja.

Čitanje i upis kroz pipe

Proces pristupa podacima u pipe-datoteci na algoritmom FIFO, što znači da podaci moraju da se čitaju u onom poretku u kome su se upisali. Nije neophodno da u pipe-datoteci broj čitalaca bude jednak broju pisaca; ako je broj pisaca i čitalaca veći od jedan, mora se obezbediti sinhronizacija.

Kernel pristupa podacima u pipe-datoteci kao i u slučaju obične datoteke kada se blokovi dodeljuju i pune u toku sistemskog poziva write. Razlika u alokaciji prostora između pipe i obične datoteke je u tome što pipe-datoteka koristi isključivo direktne pokazivače, što povećava performanse ali ograničava veličinu pipe-datoteke.

Kernel manipuliše direktnim blokovima pipe-datoteke kao sa kružnim redom čekanja (queue), kontrolišući da interni read i write pokazivači poštuju FIFO poredak, što je prikazano na slici 5.2.

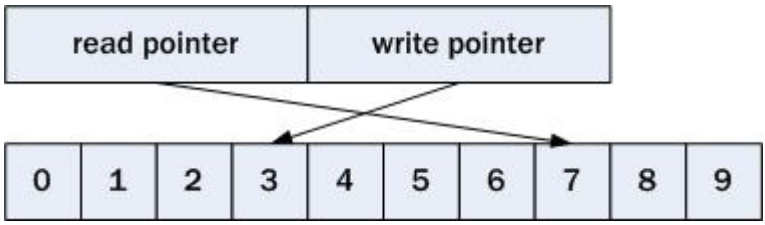
Postoje sledeća četri slučaja čitanja i upisa u pipe-datoteku:

[1] upis u pipe-datoteku kada ima mesta za upis

[2] čitanje iz pipe-datoteke kada se u njoj nalaze svi potrebni podaci koje čitaoc zahteva

[3] čitanje iz pipe-datoteke kada nema svih traženin podataka

[4] upis u pipe-datoteku kada nema mesta za upis



Slika 5.2. *Pokazivači za čitanje i upis za pipe datoteku*

Prvi slučaj nastupa u slučaju kada proces upisuje u pipe-datoteku pri čemu je suma podataka koji se upisuje zajedno sa količinom podataka koja se već nalazi u pipe-datoteci manja od kapaciteta pipe-datoteke. Kernel tada upisuje podatke kao u slučaju obične datoteke. Za svaki takav upis inkremetira se veličina pipe-datoteke i write pointer, a bude se svi read procesi koji čekaju da se nešto upiše u pipe-datoteku. Kod obične datoteke veličina se ažurira samo ako se upis dešava iza kraja datoteke. Kada se dođe do kraja pipe-datoteke, tj. kada se iscrpe svi direktni pointeri, ide se na početak pipe-datoteke (byte offset 0). Upis je uvek u rastućem poretku.

Kada proces čita pipe-datoteku, proverava se da li je ona prazna. Ako nije prazna, čitanje se dešava ali uz strogo poštovanje pomeraja u inode strukturi pipe-datoteke. Posle čitanja svakog bloka, kernel dekremetira veličinu pipe-datoteke, ažurira read pokazivač (read offset) i budi writer procese koji su uspavani i čekaju na prazno mesto u pipe-datoteci. Proces koji čita pipe-datoteku će se uspavati ako je pipe-datoteka prazna ili nema dovoljno podataka. Postoje i opcije čitanja i upisa bez kašnjenja (no delay) koje odmah vraćaju rezultat bez čekanja ako read ili write ne mogu trenutno da se zadovolje.

Ako proces želi da nešto upiše u pipe-datoteku, a pri tome nema mesta, mora da ide na spavanje dok reader proces ne isprazni pipe-datoteku. Ukoliko proces upisuje podatke koji prevazilaze kapacitet pipe-datoteke, proces ne može da čeka jer nikada neće dočekati. Tada upisuje samo onaj deo koji može da stane, u najboljem slučaju to je kapacitet pipe-datoteke, pa čeka da se ona potpuno ili delimično isprazni.

Da ponovimo, pipe datoteka liči na regularnu datoteku ali read i write pomeraji nisu kontrolabilni od strane procesa već se kontrolišu pomoću kernela, nalaze se u inode strukturi a ne u tabeli datoteka FT i to su zajednički pomeraji za sve procese koji dele pipe-datoteke.

Zatvaranje pipe datoteka

Kada se pipe-datoteka zatvara, proces obavlja sličnu proceduru, ali kernel mora obaviti specijalnu proceduru prilikom zatvaranja pipe-datoteke, odnosno otpuštanja

inode strukture te pipe-datoteke. Kernel dekrementira broj čitalaca (readers) i pisaca (writers) pipe-datoteke na svako zatvaranje. Ako broj pisaca padne na nulu, a ima uspavanih čitaoca, kernel in budi i vraća im sistemski poziv read bez pročitanih podataka. Ako broj čitalaca padne na nulu, a ima uspavanih pisaca, kernel ih budi i šalje im signal greške. U oba slučaja čekalo bi se na nešto što nije sigurno da će se dogoditi. U slučaju imenovanih pipe-datoteka mogao bi se pojaviti novi proces koji spašava situaciju.

Ako nema ni reader ni writer procesa, kernel oslobađa sve blokove podataka pipe-datoteke i markira inode strukturu da je pipe-datoteka prazna.

Pipe primeri

Ovaj program ilustruje veštačko korišćenje pipe-datoteka. Proces kreira pipe-datoteku, ide u beskonačnu petlju, upisuje string „hello“ u pipe-datoteku i čita ga iz pipe-datoteke. Kernel ne kontroliše stanje ako isti proces čita ili piše u pipe-datoteku.

```
char string [] = "hello"
main
{
  char buf[1024]; char *cp1, cp2*;
  int fds[2]; cp1 = string; cp2 = buf;
  while (*cp1)
  *cp2++ = *cp1++; /* popuna bafera "buf" sa stringom hello */
  pipe(fds);
  for (;;)
  {
    write(fds[1], buf, 6);
    read(fds[0], buf, 6);
  }
}
```

Sledeći primer ilustruje imenovanu pipe datoteku:

```
#include <fcntl.h>
char string [] = "hello"
main(argc, argv)
int argc; char *argv[]
{
  int fd; char buf[256]
  /*create named pipe with read/write permission for all users*/
  mknod ("fifo", 010777,0)
  if (argc == 2) fd = open ("fifo", O_WRONLY);
  else fd = open ("fifo", O_RDONLY);
  for (;;)
  if (argc == 2) write(fd,string,6);
  else rd(fd,buf,6);
}
```

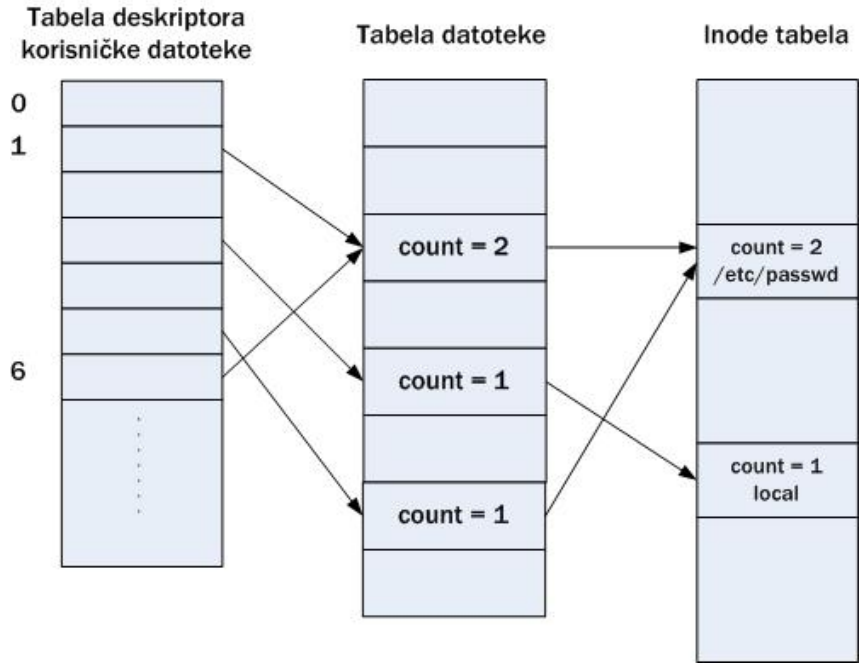
Proces kreira pipe-datoteku, a zatim ako je pozvan sa dva argumenta, kontinualno upisuje string hello u pipe-datoteku pod imenom fifo. Ako se pozove bez drugog argumenta on kontinualno čita pomenutu fifo pipe-datoteku. Ako se od ovog programa kreiraju dva procesa oni će komunicirati kroz istu pipe-datoteku, pri čemu se više procesa može uključiti na isti način.

DUP

Sistemski poziv dup kopira descriptor datoteke u prvi slobodan ulaz (slot) u UFDT strukturi, pri čemu korisniku vraća novi descriptor. Ovaj sistemski poziv radi za sve tipove datoteka i ima sledeću sintaksu:

```
newfd = dup(fd);
```

pri čemu je **fd** deskriptor datoteke koji se duplira, a newfd je novi deskriptor datoteke. S obzirom da sistemski poziv dup duplira deskriptor, inkrementiraće se broj referenci odgovarajućeg ulaza tabele datoteka FT, koji sada ima još jedan ulaz iz UFDT tabele koji na njega pokazuje.



Slika 5.3. *Primer za dup sistemski poziv*

Na slici 5.3, UFDT(1) tj. fd1 je otvorio datoteku /etc/passwd, UFDT(3) tj. fd3 je otvorio datoteku local, UFDT(5) tj. fd5 je ponovo otvorio /etc/passwd. Potom je obavljen sistemski poziv dup, newfd=dup(fd1) i fd1 se upisuje u ulaz 6.

Sistemski poziv dup ne izgleda previše elegantno, ali je zgodan za razne sofisticirane zadatke u programiranju kao što je protočna obrada komandi (shell pipeline).

Primer za sistemski poziv dup

Posmatrajmo sledeći program:

```
#include <fcntl.h>
main()
{
  int i,j;
  char buf1[512], buf2[512];
  i = open("etc/passwd", O_RDONLY)
  j = dup (i);
  read (i, buf1, sizeof(buf1));
  read (j, buf2, sizeof(buf2));
  close(i);
  read (j, buf2, sizeof(buf2));
}
```

Najpre je urađen sistemski poziv open na datoteku /etc/passwd, dobijen je deskriptor datoteke fd = i, kreiran je jedan ulaz u tabeli UFDT i jedan ulaz u tabeli datoteka FT sa bajt pomerajem koji je jednak nuli. Sistemskim pozivom dup kreiran je još jedan deskriptor datoteke fd = j, još jedan ulaz u UFDT koji ukazuje na isti FT ulaz, što znači sa istim bajt pomerajem. Prva dva sistemska poziva read pročitaće prva dva disk bloka datoteke, a onda se zatvara jedan deskriptor datoteke, u ovom slučaju i, ali se datoteka ne zatvara već samo jedan ulaz u UFDT nestaje. Ali tu je drugi deskriptor datoteke fd = j sa kojim se nastavlja čitanje iz datoteke. Ovo se može raditi i sa standardnim deskriptorima datoteke 0, 1, 2, koji se mogu se duplirati, zatvarati itd.

5.2. Aktiviranje i deaktiviranje sistema datoteka

Fizička disk jedinica sastoji se od više logičkih sekcija, i svaka sekcija ima svoje ime (device file name). Proces može pristupati podacima u sekciji kao sa datotekom, prvo se otvori device filename, a potom se disk sekcija tretira kao datoteka, po njoj se čita i piše. Sistemski poziv mount povezuje stablo i sistem datoteka koji se nalazi u odgovarajućoj sekciji na disku, dok sistemski poziv umount izbacuje sistem datoteka iz stabla. Sistemski poziv mount omogućava korisnicima da pristupaju sekciji diska kao sistem datoteka, a ne kao sekvenci disk blokova.

Aktiviranje sistema datoteka - mount

Sintaksa za sistemski poziv mount je:

```
mount(special pathname, directory pathname, options);
```

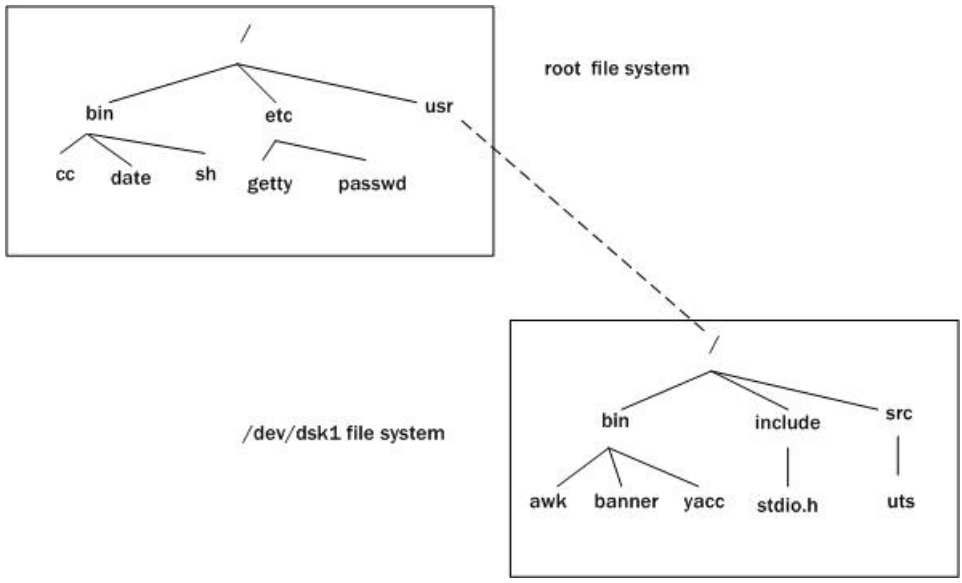
pri čemu je:

- **special pathname** - specijalna datoteka koja refencira na deo diska koji sadrži sistem datoteka koji će biti aktiviran
- **directory pathname** - direktorijum u postojećem stablu gde će sistem datoteka biti aktiviran (mount-point directory)
- **options** - skup dodatnih opcija koji ukazuje na koji način će sistem datoteka biti aktiviran (na primer, read-only opcija blokira svaki create ili write sistemski poziv u tom sistemu datoteka)

Na primer, ako proces pošalje sledeći sistemski poziv:

```
mount("/dev/dsk1", "/usr", 0)
```

kernel će priključiti sistem datoteka koji se nalazi u delu diska koji se naziva /dev/dsk1 na direktorijum /usr u postojećem UNIX stablu. Ilustracija ovog aktiviranja data je na slici 5.4.



Slika 5.4. *Primer za aktiviranje sistema datoteke*

Datoteka /dev/dsk1 je blok specijalna datoteka i po pravilu predstavlja deo diska, a da bi taj deo diska mogao da se aktivira, on mora da sadrži sistem datoteka, odnosno da sadrži superblok, inode listu i root direktorijum. Kada se kompletira sistemski poziv mount, root direktorijum aktiviranog sistema datoteka je raspoloživ preko MPD (mount point directory), /usr u ovom slučaju. Proces pristupa datotekama na aktiviranom sistemu datoteka sasvim normalno bez obzira što su one uklonjive. Jedino sistemski poziv link kontroliše tu činjenicu jer ne dozvoljava linkovanje između dva sistema datoteka.

Kernel održava tabelu aktivnih sistema datoteka MT (Mount table, MT) sa ulazom za svaki aktivirani sistem datoteka.

Svaku ulaz u tabeli aktivnih sistema datoteka sadrži:

- broj uređaja (device number) koji identifikuje aktivirani sistem datoteka
- pokazivač na bafer koji sadrži superblok za taj sistem datoteka
- pokazivač na root inode strukturu aktiviranog sistem datoteka (/ of /dev/dsk1)
- pokazivač na inode strukturu za MPD (usr of root FS)

Asocijacija MPD inode strukture i root inode strukture aktiviranog sistema datoteka, dozvoljava kernelu da prostiru (traverse) sistem datoteka vrlo efikasno.

Algoritam mount

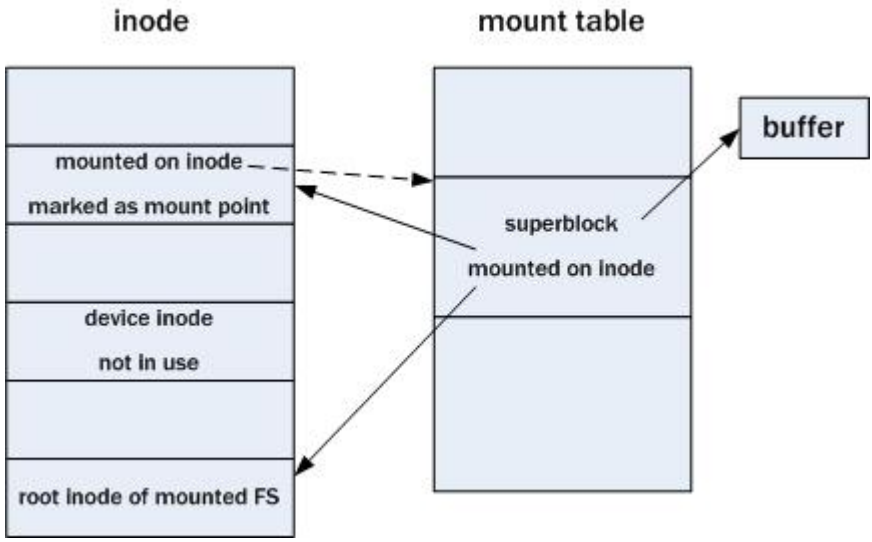
Kernel dozvoljava samo procesu sa root privilegijama da obavi sistemske pozive mount i umount, kako se ne bi izazvao haos u sistemu. Kernel pronalazi inode strukturu za blok specijalnu datoteku koja predstavlja sistem datoteka za aktiviranje, izvlači glavni i sporedni broj, pomoću kojih se identifikuje sekcija na disku koja sadrži taj sistem datoteka i nalazi inode strukturu za MPD. Broj referenci za taj MPD ne sme da bude veći od jedan, jer se ne sme aktivirati više sistema datoteka na isti MPD. Kernel zatim alocira slobodan ulaz u tabeli aktivnih sistema datoteka (MT, Mount Table), markira ulaz da je zauzet i dodeljuje broj uređaja nove datoteke (device number). Potom se poziva open procedura za blok-specijalnu datoteku, inicijalizuju se drajverske strukture podataka i šalju komande hardveru preko disk drajvera. Kernel alocira slobodni blok u bafer kešu (getblk) i preko bread algortima učitava superblok u njega. Kernel memoriše pokazivač na inode strukturu za MPD. Kernel pronalazi inode strukturu za root direktorijum sistema datoteka koji će biti aktiviran i memoriše ga u MT tabeli. Za korisnika, MPD i root direktorijum aktiviranog sistema datoteka su logički ekvivalentni i kernel uspostavlja njihovu ekvivalenciju u istom ulazu u MT. Iza aktiviranja, procesi više ne pristupaju inode strukturi za MPD.

Kernel inicijalizuje polja u superbloku, brišući zaključavanje za listu slobodnih blokova i za listu slobodnih inode strukura i postavlja broj slobodnih inode struktura na nulu. To se radi da bi se sprečilo oštećenje sistema datoteka prilikom aktiviranja sistema datoteka posle nasilnog pada sistema, što će naterati kernel da pretraži slobodne inode strukture algoritmom ialloc. Nažalost, ako su povezane liste slobodnih blokova oštećene, kernel ih

ne popravlja, već se to postiže preko programa fsck.

Ukoliko sistemski poziv mount sadrži read-only opciju, kernel će postaviti read-only zastavicu u superbloku.

Na kraju, kernel će markirati inode strukturu za MPD kao tačku aktiviranja (mountpoint), tako da svi procesi mogu da to identifikuju, što je ilustrovano na slici 5.5.



Slika 5.5. *Relacija između tabele MT, MPD direkttorijuma i root direktorijuma*

```
algorithm mount

input: {file name of block special file,
        MPD,
        options]
output: none

{
  if(not superuser)  return (error);
  get inode of block special file (algorithm namei);
  make legality check;
  get inode of MPD (algorithm namei);
  if (not directory, or reference count > 1)
  {
    release inodes (algorithm iput);
    return (error);
  }
  find empty slot in MT;
```

```
    invoke block device driver open routine;
    get free buffer from buffer cache;
    read superblock into free buffer;
    initialize superblock fields;
    get root inode of mounted device (algorithm iget),
      save in mount table;
    mark inode of MPD as mount point;
    release special file inode (algorithm iput);
    unlock inode of MPD;
}
```

Prolazak kroz MPD (Crossing mount points in file path names)

Razmatrimo šta se dešava ako putanja (pathname) prolazi preko MPD i kako se ponašaju algoritmi namei i iget. Postoje dva slučaja prelaska preko MPD:

- prelazak iz MPD na aktivirani sistem datoteka, tj. u dubinu (komanda 2)
- prelazak iz aktiviranog sistema datoteka na MPD, tj. iz dubine (komanda 3)

```
[1] mount /dev/dsk1 /usr

[2] cd /usr/src/ust

[3] cd ../../..
```

Prva komanda obavlja sistemski poziv mount koji aktivira sistem datoteka (/dev/dsk1) na MPD (/usr). Sledeća komanda cd, obavlja sistemski poziv chdir koji na svom putu prolazi preko MPD direktorijuma, /usr. Druga komanda cd ide na tri roditeljske grane, prolazeći opet MPD direktorijum /usr.

U prvom slučaju, za prelazak iz MPD na aktivirani sistem datoteka (u dubinu), koristimo modifikovani iget algoritam, koji je sličan osnovnom algoritmu iget, osim što proverava da li je inode struktura istovremeno MPD. Za svaku takvu inode strukturu, pronalazi se ulaz u tabeli MT, iz koga se čita broj sistema datoteka (device number) i inode struktura root direktorijuma, na osnovu kojih može pristupiti root direktorijumu aktiviranog sistema datoteka. U prvoj cd komandi, kernel pristupa inode strukturi direktorijuma /usr i detektuje da je to MPD, a onda iz MT tabele čita broj sistema datoteka (device number) i inode strukturu root direktorijuma.

Modifikivani iget algoritam

```
algorithm iget

input: file system and inode number
output: locked inode

{
```

```
  while(not done)
  {
    if(inode in inode cache)
    {
      if(inode locked)
      {
        sleep (event inode becomes unlocked);
        continue; /* loop back to while*/
      }
      /*special processing for mount point directory*/
      if(inode a mount point)
      {
        find mount table entry for mount point;
        get new FS number from MT;
        use  root  inode  number in search;
        continue
      }
      if(inode on inode free list)  remove from free list;
      increment inode reference count;
      return (inode);
      /* inode not in inode cache*/
      if(no inodes on free list) return(error)
      remove new inode from free list;
      reset inode number in free list;
      remove inode from old hash queue, place on new one;
      read inode from disk(algorithm bread)
      initialize inode (eg. reference count to 1)
      return(inode)
    }
  }
}
```

Modifikivani namei algoritam

Za drugi slučaj (cd ../../..) posmatajmo modifkovani namei algoritam.

```
algorithm namei /* convert path name to inode*/

input: path name
output: locked inode

{
  if (path name starts from root)
    working inode = root inode (algorithm iget);
  else
```

```
    working inode = current directory inode (algorithm iget);
  while (there is more path name)
  {
    read next path name component from input;
    verify that working inode is of directory,
      access permissions OK;
    if (working inode is of root and component is "..") continue;
      /*loop back to while*/
    read directory (working inode) by repeated use of
      algorithms bmap, bread and brelse
    if (component matches an entry in directory (working inode))
    {
      get inode number for matched component;
      if (found inode of root and working inode is root
          and component name is ..)
         /*crossing mount point*/
      {
        get MT entry for working inode;
        release working inode (algorithm iput);
        working inode = mounted on inode; (MPD inode)
        lock MPD inode;
        increment reference count of working inode;
        go to component search for ..;
      }
      release working inode (algorithm iput);
      working inode = inode of matched component (algorithm iget)
    }
    else
      return (no inode);
  }
  return(working inode);
}
```

Za svaku inode strukturu iz putanje , kernel proverava da li je to root direktorijum i ako je sledeća komponenta iz pitanje (..), tada je to sigurno prolazak kroz MPD. Tada se tekuća inode struktura (working inode) zamenjuje inode strukturom za MPD koji se čita iz tabele MT.

Postoje dve inode strukture za svaki MPD direktorijum, njegova originalna inode struktura i root inode struktura od aktiviranog sistema datoteka i kada je algoritam namei prolazi preko MPD, mora doći do zamene tekuće inode strukture.

Deaktiviranje sistema datoteka

Sintaksa za umount sistemski poziv je sledeća:

```
umount (special filename)
```

gde **special filename** ukazuje na sistem datoteka koga treba deaktivirati. Kernel mora prvo proveriti da nema otvorenih datoteka i blokova sa odloženim upisom (DW, Delayed write blocks) u kešu za sistem datoteka koji će se deaktivirati. Pretražuje se tabela datoteka FT, za sve potencijalne datoteke koje pripadaju tom sistemu datoteka.

Algoritam za umount

```
algorithm  umount

input:  file name of block special file
output:  none

{
  if(not superuser)  return (error);
  get inode of block special file (algorithm namei);
  extract major, minor number of FS;
  ger mount table entry, based on major and minor number;
  release special file inode (algorithm iput);
  remove shared text entries from region tables
    for files belonging FS;
  update superblock, inodes, flush buffer;
  if (files from FS still in use) return (error);
  get root inode of mounted FS from MT;
  lock inode;
  release inode (algorithm iput); /*iget was in mount*/
  invoke close routine for special device;
  invalidate buffers in pool from unmounted FS;
  get inode of MPD from MT;
  lock inode;
  clear flag marking it as mountpoint;
  release inode (algorithm iput); /*iget was in mount*/
  free buffer used for super block;
  free mount table slot;
}
```

Kernel uzima inode strukturu za blok specijalnu datoteku koja odgovara sistemu datoteka, ažurira superblok i inode strukturu, zatim sravnjuje memorijsku sliku sa stanjem na disku za superblok i inode strukturu (flush). Na kraju oslobađa inode strukturu za root direktorijum, zatvara uređaj, uzima MPD, briše mu mountpoint zastavicu, i oslobađa inode strukturu za bivši MPD.

Na kraju briše ulaz u tabeli MT i bafer za superblok.

5.3. Sistemski pozivi link i unlink i konzistencija podataka

Sistemski poziv link povezuje datoteku sa novim imenom u stablu, tako što kreira novi FCB sa postojećom inode strukturom. Sistemski poziv unlink uklanja FCB datoteke iz direktorijuma.

Link

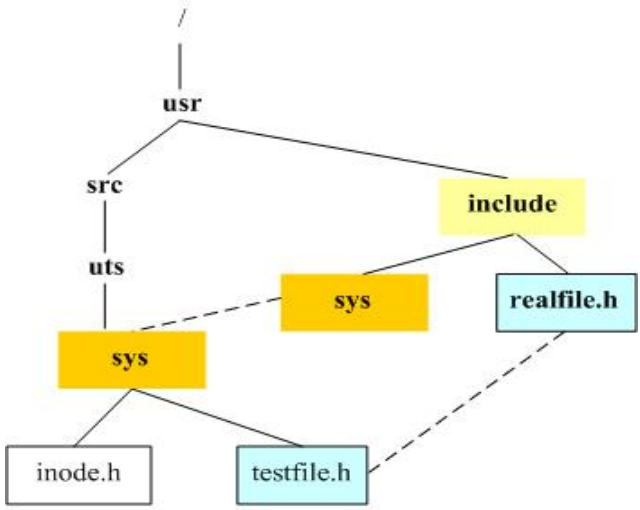
Sintaksa za sistemski poziv link je:

```
link (source file name, target file name);
```

gde je **source file name** ime postojeće datoteke, a **target file name** nova datoteka koja će se kreirati ako uspe sistemski poziv link. Sistem datoteka sadrži imena za sve linkove koje datoteka ima, i proces može pristupati datoteci po bilo kom od tih imena. Kernel ne zna koja je prva odnosno originalna datoteka, tako da su sva imena ravnopravna.

Na primer, posle izvršenja sledeća dva sistemska poziva, dogodiće se situacija kao na slici 5.6.

```
link ("/usr/src/uts/sys",  "/usr/inlcude/sys")
link ("/usr/inlcude/realife.h","/usr/src/uts/sys/testfile.h")
```



Slika 5.6. *Primer za sistemski poziv link*

Tri različita imena ukazuju na istu datoteku:

- `/usr/include/realife.h`
- `/usr/src/uts/sys/testfile.h`
- `/usr/include/sys/testfile.h`

Kernel dozvoljava isključivo root korisniku da linkuje direktorijume, zato što postoji mogućnost nastanka beskonačnih petlji kada korisnik linkuje direktorijume koji su na istoj grani, a ide se unazad. Potreba za linkovanjem direktorijuma postojala je na starim UNIX sistemima, zato što je mkdir komanda radila preko sistemskog poziva link. Kasnije je to razrešeno uvođenjem sistemskog poziva mkdir, čime nestaje potreba za linkovanjem direktorijuma.

Algoritam za link

Sledi prikaz algoritma link.

```
algorithm link

input: {existing file name,
        new file name}
output: none

{
  get inode for existing file name (algorithm namei);
  if (too many links on file or
      linking directory without super user permission)
  {
    release inode (algorithm iput);
    return (error);
  }
  increment link count on inode;
  update disk copy of inode;
  unlock inode;
  get parent inode for directory to contain
    new file name (algorithm namei);
  if (new file name already exists or existing/new
    file name on different FS)
  {
    undo update done above;
    return (error);
  }
  create new FCB-directory entry in parent directory of
    new file name: FCB includes new file name, but inode
    number of existing file;
```

```
    release parent directory inode (algorithm iput);
    release inode of existing file (algorithm iput);
}
```

Kernel prvo locira inode strukturu originalne datoteke, preko algoritma namei, zatim inkrementira broj linkova (link count), ažurira disk kopiju inode strukture (iz razloga konzistencije), a na kraju otključava inode strukturu. Zatim se traži odredišna datoteka (target file), ako ona postoji sistemski poziv link će otkazati, a tada se broj linkova dekrementira jer je prethodno inkremetiran. Ako odredišna datoteka ne postoji, dodeljuje se nova FCB struktura u odredišnom direktorijumu, upisuje se ime odredišne datoteke i broj inode strukture originalne datoteke i oslobađa se inode struktura roditeljskog direktorijuma za odredišnu datoteku preko algoritma iput. Potom se oslobađa inode struktura postojeće datoteke čiji je broj linkova uvećan za jedan.

Zastoj (deadlock) u linkovanju datoteka

Prilikom linkovanja datoteka mogući su zastoji. Obradićemo dve moguće situacije za zastoj i obe potiču zbog neoslobađanja inode strukture nakon inkrementiranja broja linkova. Treba naglasiti da se uvek zaključava inode struktura originala. Ako kernel ne oslobađa inode strukturu, dva procesa bi mogla da uđu u zastoj, kao na primeru dva simultana procesa:

```
proces A: link("a/b/c/d", "e/f/g");
proces B: link("e/f ", "a/b/c/d/ee");
```

Pretpostavimo da proces A nalazi inode strukturu za "a/b/c/d" u isto vreme kada proces B nalazi inode strukturuza "e/f " i oba postaju blokirana. Kada proces A pokušava da nađe inode strukturu za "e/f", on mora da čeka da inode struktura za direktorijum e postane slobodna. Na drugoj strani, proces B mora da čeka da inode struktura za direktorijum d postane slobodna. Na taj način, oba procesa su uletela u zastoj. Sve se to sprečava tako što kernel otpušta inode strukturu nakon inkrementiranja broja linkova.

Unilnk

Sistemski poziv unlink uklanja FCB datoteke iz direktorijuma. Sintaksa za sistemski poziv unlink je sledeća:

```
unlink (pathname);
```

gde je **pathname** ime koje će biti izbrisano iz UNIX stabla. Na primer u sledećoj sekvenci, sistemski poziv open će otkazati jer je datoteka prethodno obrisana:

```
unlink ("myfile");
fd = open ("myfile", O_RDONLY);
```

Ako se sistemski poziv unlink primeni na ime datoteke koja je poslednji ili jedini link na tu datoteku, unlink će osloboditi inode strukturu i sve blokove datoteke. Ako postoji

više linkova, nestaje samo to ime i broj linkova se dekrementira, dok je datoteka raspoloživa preko svih ostalih imena.

Algoritam za unlink

```
algorithm  unlink

input: file name
output: none

{
  get parent inode for directory to contain file to be
    unlinked (algorithm namei);
  /* if unlinking the current directory…*/
  if (last component of file name ".")
    increment inode reference count;
  else
    get inode of file to be unlinked (algorithm iget);
  if (file is directory but user is not super-user)
  {
    release inodes (algorithm iput);
    return (error);
  }
  if (shared text file and link count currently 1)
    remove from region table;
  write parent directory: zero inode number of unlinked file;
  release inode parent directory (algorithm iput);
  decrement file link count;
  release file inode (algorithm iput);
  /* iput checks if link count is 0: if so, release file blocks
     (algorithm free) and frees inode (algorithm ifree) */
}
```

Kernel prvo preko algoritma namei nalazi inode strukturu roditeljskog direktorijuma, a ne inode strukturu datoteke koju briše. Potom se pristupa in-core inode strukturi datoteke koja će biti obrisana, preko algoritma iget. Posle toga se obavljaju testovi, da li to deljiva tekst datoteka (shared text) i da li je broj linkova za nju jednak jedinici. Tada se deljivi tekst region uklanja iz region tabele, a potom kernel briše FCB datoteke iz direktorijuma, tako što se na mestu inode broja upisuje nula, a to se sinhrono upisuje na disk i inode struktura roditeljskog direktorijuma se oslobađa preko algoritma iput. Za datoteku koja se briše broj linkova se dekrementira i oslobađa se in-core inode struktura. Ako je prilikom otpuštanja in-core inode strukture broj linkova jednak nuli, ta datoteka više neće postojati na disku. Iput algoritam tada oslobađa sve blokove podataka sa algoritmom free, tako što oslobađa sve direktne i sve indirektne blokove; takođe oslobađa inode strukturu sa ifree algoritmom, tako što u inode strukturi upisuje polje filetype da bude jednako nuli.

Konzistencija sistema datoteka

Kernel obavlja svoje upise u poretku koji bi minimizovao verovatnoću oštećenja sistema datoteka u slučaju otkaza sistema. Na primer, kada se briše datoteka iz svog direktorijuma, to se sinhrono upisuje na disk u roditeljski direktorijum, pre nego što se oslobode blokovi datoteke i inode struktura. Ako se dogodi havarija sistema (system crash) dok se oslobađaju blokovi podataka, to nije opasna situacija, zato što nema FCB koji ukazuje na tu inode strukturu. U sličaju da unos nije sinhron a dogodi se havarija, FCB koji nije upisan na disk može ukazivati na potpuno praznu ili pogrešnu inode strukturu što predstavlja ozbiljnije oštećenje sistema datoteka. Mnogo je lakše očistiti inode strukturu za koju ne postoji FCB, nego ispraviti pogrešan FCB.

Primer za konzistenciju sistema datoteka

Pretpostavimo da imamo dva linka sa imenima a i b i obavimo unlink a. Kernel poštuje poredak write operacija, pa prvo anulira FCB u direktorijumu za link a i upiše ga na disk. Ako tada sistem doživi havariju (crach), broj linkova (LC) je dva, a ostao je samo b, tako da će sistem imati problema jedino ako se briše b, a u svim ostalim slučajevima sve je ispravno.

Pretpostavimo drugi poredak write operacija. Kernel prvo dekrementira broj linkova i događa se havarija pre nego što je obrisan FCB. Posle podizanja sistema, imamo dva FCB za istu inode strukturu, a broj linkova je jednak jedan. Ako se obavi brisanje a ili b, datoteka će fizički nestati, a ostaje FCB koji postaje neregularan, jer ukazuje ili na praznu inode strukturu ili na inode strukturu koja će se kasnije dodeliti nekoj drugoj datoteci, koja će opet imati broj linkova jednak jedan, a dva imena.

Kernel takođe oslobađa inode strukturu i blokove podataka u poretku; mogu se prvo oslobađati blokovi podataka pa tek onda inode struktura ili obrnuto, ali uticaj havarije sistema je sasvim različit. Pretpostavimo da kernel oslobađa disk blokove i da se dogodi havarija (crash). Nakon podizanja sistema imamo inode koji ukazuje na oslobođene blokove, a to je prilično neregularna situacija. Ako se prvo oslobodi inode struktura, pa sistem pukne, blokovi koji su pripadali datoteci postaju neupotrebljivi jer nisu vraćeni u slobodnu listu, ali sve ostalo je sasvim regularno, nema inode struktura i nema pogrešnog pristupa. Program fsck lakše oslobađa disk blokove nego što ih čisti.

Stanje trke

Stanje trke (race condition) je prisutno u sistemskom pozivu unlink, osobito kada se brišu direktorijumi naredbom rmdir. Direktorijum se može obrisati samo ako je prazan, a to će se konstatovati samo ako svi direktorijumski ulazi-FCB imaju vrednost inode strukture nula. Naredba rmdir se ne obavlja atomski, što znači da se može dogoditi promena konteksta (context switch) između čitanja za proveru direktorijumskih blokova i brisanja samog direktorijuma. Na primer, neki drugi proces obavlja sistemski poziv create, a ovaj prvi je konstatovao da je već prazan. Jedini način sprečavanja prethodne

situacije je zaključavanje zapisa (record locking). Kada proces izvršava sistemski poziv unlink, nijedan drugi proces ne može da pristupa roditeljskom direktorijumu jer mu je inode struktura zaključana zbog algoritma unlink.

Podsetimo da algoritam za sistemski poziv link oslobađa inode stukturu pre kraja poziva, ali ako drugi proces pokušava da obriše datoteku, to je moguće zato što je sistemski poziv link oslobodio inode strukturu, dok brisanje samo dekrementira broj linkova, a datoteka ostaje.

Drugo stanje trke dešava se kada jedan proces konvertuje ime datoteke u inode strukturu preko namei algoritma, a drugi proces uklanja granu u toj putanji. Pretpostavimo da proces A razvija putanju "a/b/c/d" i odlazi na spavanje sve dok se ne oslobodi in-core inode struktura za direktorijum c. Proces B na primer želi da obavi unlink c i on odlazi na spavanje. Kada se direktorijum c oslobodi i ako kernel dâ prednost procesu B, tada će B obrisati granu c, naravno ukoliko je prazna. Kada proces A dobije procesor, algoritam namei više ne nalazi in-core strukturu za direktorijum c, jer je ona u međuvremenu obrisana a proces A je na nju čekao i tada sistemski poziv namei mora da se završi sa porukom o grešci. Može da se dogodi i treća situacija u kojoj novi proces C dobija inode strukturu koja je pripadala direktorijumu c, pa stvara novu granu ili datoteku, a da proces A nastavlja sa izmenjenom siutacijom, pa će pristupiti pogrešnoj datoteci, što nije dobro.

Proces može obrisati datoteku čak i kada je otvorena od strane nekog drugog procesa. Pošto sistemski poziv open, oslobađa inode strukturu na kraju, sistemski poziv unlink će biti uspešan. Kernel realizuje sistemski poziv unlink kao da datoteka nije otvorena i uklanja FCB, tako da datoteci više niko ne može da pristupi. Međutim kako je sistemski poziv open inkrementirao broj referenci, kernel ne briše datoteku oslobađajući blokove podataka i inode strukturu, u iput algoritmu, koga poziva sistemski poziv unlink. Svi procesi koji su otvorili datoteku rade sa njenim deskriptorom datoteke (FD). Tek kada se dogodi sistemski poziv close, kernel preko iput algoritma briše sadržaj datoteke. Ovo se često dešava, proces kreira privremenu datoteku, pa je tako otvorenu odmah obriše, ali se pravo brisanje događa tek na zadnji sistemski poziv close.

Primer za stanje trke

Evo primera: prvo se otvara datoteka, zatim se briše, sistemski poziv stat otkazuje jer ide datoteku tj. na ime koga više nema, ali sistemski poziv fstat radi jer ide preko file deskriptora.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
main(argc, argv)
int argc;
char *argv[];
{
  int fd;
```

```
  char buf[1024];
  struct stat, statbuf;
  if(argc != 2) exit (); /*need parameter*/
  fd = open(argv[1], O_RDONLY)
  if (fd == -1) exit (); /*open fails */
  if(unlink(argv[1], == -1) exit (); /*unlink just opened */
  if( (stat(argv[1],&statbuf == -1)  /*stat the file by name */
    printf("stat %s fails as it should\n", argv[1]);
  else
    printf("stat %s succeded!!!!! \n", argv[1]);
  if( (fstat(fd,&statbuf == -1)  /*fstat the file by fd */
    printf("fstat %s fails !!!!!\n", argv[1]);
  else
    printf("fstat %s succeded as it should\n", argv[1]);
  while (read(fd, buf, sizeof(buf)>0) /*read open/unlinked file*/
    printf("%1024s n", buf); /*prints 1K byte field*/
)
```

Apstakcije sistema datoteka

Weinberg je uveo tipove sitema datoteka da bi obezbedio podršku za njegov mrežni sistem datoteka (network filesystem), što je implemetirano na UNIX System V. Zahvaljujući tipovima sistema datoteka, kernel može simultano podržavati mrežne sisteme datoteka ili sisteme datoteka različitih operativnih sistema. Procesi koriste normalan sistemski poziv open, a kernel mapira generički skup I/O operacija u operacije za specifični sistem datoteka.

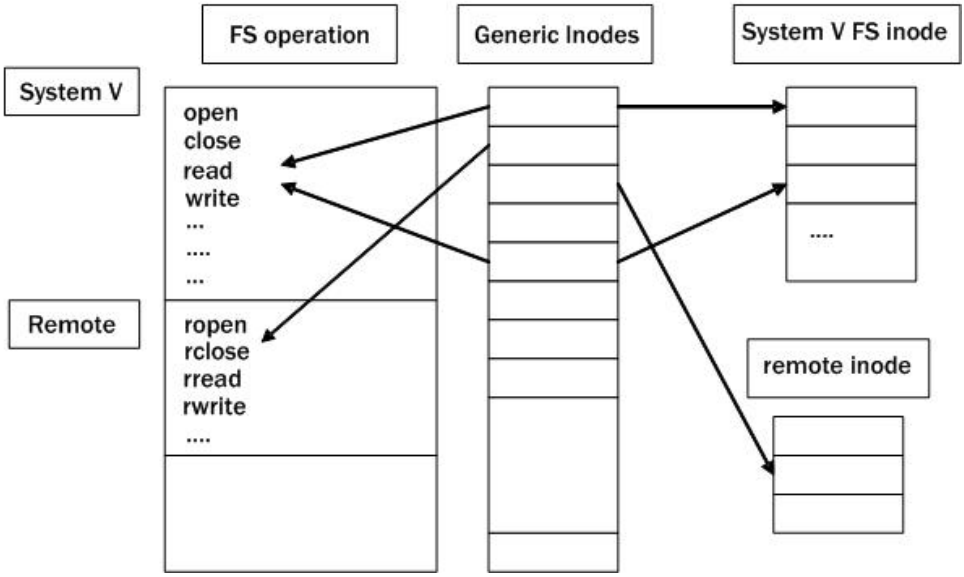
Inode struktura je interfejs između apstraktnog i specifičnog sistema datoteka. Generička in-core inode struktura sadrži podatke koji su nezavisni od specifičnog sistema datoteka i ukazuju na file-specifičnu inode strukturu koja sadrži specifične podatke za datoteku.

File-specifična inode struktura uključuje informacije kao što su kontrola prava pristupa, položaj datoteke na disku (block layout) itd.

Generička inode struktura sadrži informacije kao što su broj uređaja (device number), broj inode strukture (inode number), tip datoteke (file type), veličina (size), vlasništvo i broj referenci. Drugi podaci koji su specifični za taj sistem datoteka sadržani u superbloku i direktorijumskim strukturama.

Na slici 5.7 je prikazana generička in-core inode tabela i dve tabele sa file-specifičnim inode strukturama, jedna za sistem datoteka kod sistema UNIX System V, a druga za udaljeni sistem datoteka (remote), koja sadrži dovoljno informacija da identifikuje datoteku na udaljenom sistemu datoteka. Taj udaljeni sistem datoteka možda i nema inode baziranu strukturu, međutim, preko generičke inode tabele i file-specifične tabele, sve se to mapira u strukturu koja podseća na inode strukturu (inode-like). Mapiranje se vrši tako da se sve radi apstraktno. Sa svim datotekama se apstraktno radi sa

sistemskim pozivima open, close, read i write. Apstraktna manipulacija sa inode strukturama obavlja se preko algoritma namei,iget i iput. Svi sistemi datoteka se apstraktno aktiviraju i deaktiviraju preko sistemskih poziva mount i umount.



Slika 5.7. *Apstrakcija sistema datoteka, generička inode tabela i file specifične tabele*

Održavanje sistema datoteka

Kernel održava konzistenciju sistema datoteka za vreme normalne operacije, ali neobične okolnosti kao što je nestanak napajanja (power-fail), mogu sistem datoteka ostaviti u nekonzistetnom stanju. Komanda fsck proverava sistem datoteka i ispravlja nepravilnosti, pristupajući preko blok ili karakter interfejsa, zaobilazeći regularni pristup datoteci.

Za disk blokove, pravila su jasna; prvo su svi blokovi slobodni, potom se blok izbaci iz liste a dodeli samo jednoj inode strukturi, a da bi se dodelio drugoj mora se prvo vratiti u listu. Prema tome, disk blok ili je u slobodnoj listi ili je u samo jednoj inode strukturi. Tipične neregularnosti su da disk blokovi pripadaju više od jednoj inode strukturi ili se istovremeno nalaze i u listi slobodnih blokova i u inode strukturi. Tipična situacija koja se dešava nastaje kada kernel oslobodi blok iz datoteke, upiše ga in-core slobodnu listu superbloka i dodeli ga u in-core strukturu druge datoteke. Potom treba ažurirati superblok na disku i obe inode strukture i ako se pre tih ažuriranja dogodi havarija, blok se može naći u inode strukturama obe datoteke, a takođe može završiti i u slobodnoj listi i u inode strukturi.

Postoji još jedna neregularna situacija kada blok nije ni u slobodnoj listi ni u datoteci. Na primer, blok koji se izbacuje iz slobodne liste upisuje se u in-core inode strukturu, ali se ne ažurira a dogodi se havarija, pa ga nema nigde.

Takođe, inode strukture mogu imati broj linkova različit od nule, a da ta inode struktura ne postoji u nijednom FCB. To se dešava u slučaju havarije, kada se dodeljuje inode struktura za pipe ili novokreiranu datoteku, pri čemu se ne upisuje FCB u direktorijum. Slična situacija može da se dogoditi ako se neregulano obriše grana koja nije prazna.

I druga polja u inode strukturi mogu biti neregularna, na primer tip datoteke u slučaju da se aktivira nepravilno formatirani sistem datoteka.

Neregularnosti se pojavljuju u slobodnim i zauzetim inode strukturama, jer se zbog havarije, inode struktura može naći i u FCB i u listi slobodnih inode struktura.

Neregularnost se javlja kada broj slobodnih blokova i broj slobodnih inode struktura u superbloku ne odgovaraju realnom stanju na disku, zato što kompletna informacija u superbloku mora biti veoma tačna.

6

Struktura UNIX procesa

6.1. Struktura procesa

U uvodnim poglavljima već formulisali smo karakteristike procesa. Ovde će se te karakteristike bolje objasniti preko prebacivanja konteksta (context switch) i objašnjenja kako kernel identifikuje i locira procese, sa detaljnim prikazom stanja procesa i tranzicija stanja.

Kernel sadrži tabelu procesa koju ćemo zvati tabelu procesa PT (process table), kod koje svaki ulaz opisuje stanje jednog aktivnog procesa u sistemu, dok u-područje sadrži dodatne informacije koje kontrolišu operaciju procesa. Ulaz u tabeli procesa PT i u-područje su delovi konteksta procesa. Ono po čemu se procesi najviše razlikuju jesu njihovi adresni prostori.

U ovom poglavlju obradićemo:

- stanja procesa i tranzicije
- principe memorijskog upravljanja za procese i kernel i kako se upravlja virtuelnom merorijom
- kontekst procesa i algoritme niskog nivoa (low-level) koji manipulišu sa kontekstom procesa
- način na koji kernel čuva kontekst procesa za vreme prekida, sistemske pozive i kako nastavlja prekinuti proces
- algoritme za manipulaciju procesnim adresnim prostorom
- algoritme za sistemske pozive za uspavljivanje (sleep) i buđenje procesa (wakeup)

Stanja procesa i tranzicije

U uvodnim lekcijama, data je osnovna šema stanja procesa i tranzicija, a ovde će biti data potpuna šema. Sledeća lista sadrži kompletno stanje procesa:

[1] **User Running**. Proces se izvršava u korisničkom modu

[2] **Kernel Running**. Proces se izvršava u kernelskom modu

[3] **Ready to run in memory**. Proces se ne izvršava ali je spreman potpuno čim ga kernel prozove

[4] **Asleep in memory**. Proces je uspavan u memoriji

[5] **Ready to run, Swapped**. Proces je spreman za rad, ali je na swap prostoru, pa proces swapper mora da ga prebaci u memoriju pre nego što ga kernel prozove

[6] **Sleep, Swapped**. Proces je uspavan na swap prostoru, prvo uspavan pa prebačen na swap prostor

[7] **Preempted**. Proces se vraća iz kernelskog moda u korisnički mod, ali kernel mu oduzima procesor (preemtion) i obavlja prebacivanje konteksta da bi aktivirao drugi proces. Iz stanja 7 vrlo brzo će preći u stanje 3.

[8] **Created**. Proces je novo kreiran i nalazi se u tranzicionom stanju, praktično proces postoji ali nije ni spreman za rad, a nije ni uspavan, ovo je praktično početno stanje svih procesa izuzev za proces 0.

[9] **Zombie**. Proces izvršava sistemski poziv exit i nalazi se u zombi stanju. Proces više ne postoji, ali ostavlja izlazni status (exit_code) i neke vremenske statističke podatke za svoje procese roditelje koji ih sakupljaju. Zombie stanje je finalno stanje procesa.

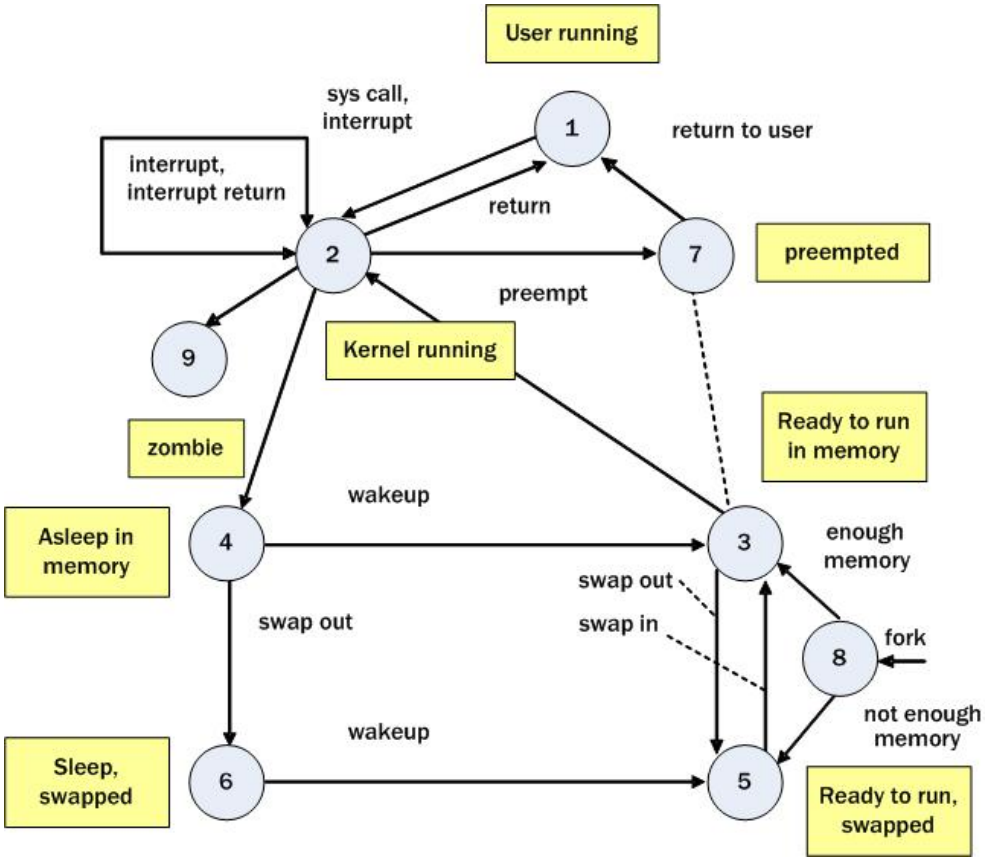
Dijagram stanja procesa

Na slici 6.1 je prikazan puni dijagram stanja procesa – kompletna slika procesnih stanja i tranzicija. Razmarajmo tipičan proces koji prolazi kroz svoja stanja. Proces ulazi u model stanja u stanje 8 (created), kada proces roditelj obavlja sistemski poziv fork i iza tog stanja prelazi u stanje gde je spreman za rad 3 ili 5. Ako je u stanju 3, raspoređivač procesa (scheduler) ga može izabrati za rad kada prvo ulazi u stanje 2 kernel running, kada kompletira svoj deo sistemskog poziva fork.

Kada proces kompletira sistemski poziv, može se prebaciti u stanje 1 (user running) kada se izvršava u korisničkom modu. Može se dogoditi prekidni signal kao na primer prekidni signal časovnika (clock interrupt), kada proces ulazi u kernelski mod, sačuva se njegov kontekst, izvrši se interrupt handler, a potom kernel može da vrati proces ponovo u stanje 1 (user running), ali kernel može da mu oduzme procesor (preemtion) da bi aktivirao neki drugi proces. Kernel ga prebacuje u stanje 7 (preemted) koje je skoro isto kao i stanje 3 (ready to run), ali su ona razdvojena da bi se označilo da proces jedino može da izgubi procesor samo kad se prebacuje iz kernelskog u korisnički mod. Takav proces (preempted process) može da se prebaci u swap prostor, a isprekidana linija ukazuje da su stanja 3 i 7 praktično ekvivalentna.

Kada proces izvršava sistemski poziv, on napušta stanje 1 (user running) i prelazi u stanje 2 (kernel running). Ako proces tom prilikom izvršava I/O operaciju na koju mora da čeka, on će ući u stanje 4, a to je stanje uspavan u memoriji (asleep in memory), iz koga može da se prebaci na swap prostor (stanje 6), a iz oba stanja uspavanosti (stanja 4 i 6) se može probuditi preko prekidnog signala, koji označava da je I/O završen. Kernel će probuditi sve uspavane procese koji čekaju na taj događaj.

Algoritam za buđenje (wakeup) prebacuje proces iz stanja 4 (Asleep in memory) u stanje 3 (Ready to run in memory) ili iz stanja 6 (Sleep, swapped) u stanje 5 (Ready to run, swapped).



Slika 6.1. *Dijagram stanja procesa*

Ukoliko postoji preveliki broj procesa, tako da je Unix u stanju deficita memorije, sistemski proces swapper može neki od procesa prebaciti iz stanja 3 (Ready to run in memory) u stanje 5 (Ready to run, swapped) i tako povećati količinu raspoložive fizičke memorije.

Proces može sam kontrolisati neke tranzicije na korisničkom nivou (user level).

Prva korisnički-kontrolabilna tranzicija, je kada proces može da kreira novi proces, preko sistemskog poziva fork, a stanje novog procesa će biti ili stanje 3 (Ready to run in memory) ili stanje 5 (Ready to run, swapped) zavisno od kernela i od količine slobodne memorije, što ne zavisi od samog procesa.

Druga korisnički-kontrolabilna tranzicija, je pozivanje bilo kog sistemskog poziva. Proces može uvek sebe prebaciti iz stanja 1 u 2 tako što pozove sistemski poziv, ali za obrnutu transformaciju iz stanja 2 u stanje 1 proces nema kontrolu, ta tranzicija je pod

kontrolom kernela. Ta tranzicija se ne mora dogoditi, u slučaju da se pojavi signal za završetak (signalTermination) i proces prelazi iz stanja 2 u stanje 9 (zombie stanje).

Treća korisnički-kontrolabilna tranzicija, nastaje kada proces izvršava sistemski poziv exit, koji ga prebacuje iz stanja 1 u stanje 2, pa u stanje 9 (zombie stanje), čime se praktično završava aktivnost procesa.

Sve druge tranzicije su pod kontrolom kernela, zavise od spoljnih događaja, količine memorije, pri čemu su neka pravila strogo definisana: na primer, ni jedan proces ne može nasilno izgubiti procesor dok je u kernelskom modu, tj. u stanju 2. Proces može nasilno izgubiti procesor samo kada se obavlja tranzicija iz stanja 2 u stanje 1.

Kernelske strukture podataka vezane za proces

Dve strukture podataka opisuju stanje procesa:

- ulaz u tabeli procesa PT: koji sadrži polja koja su uvek raspoloživa kernelu
- **u-područje** (u-area) sadrži polja koja su potrebna jedino procesu koji se izvršava

Zato kernel alocira prostor u u-područje svaki put kada se kreira novi proces, tako da samo popunjeni ulazi u tabeli procesa PT imaju svoje alokacije za u-područja.

PT ulazi

Polja u ulazu tabele procesa PT su:

[1] Polje stanje (**state field**), identifikuje stanje procesa

[2] Polje za pokazivače na u-područje (**u-area pointers**). Pokazivači omogućaju kernelu da lociraju u-područje u memoriji ili eventualno na disku. Kernel koristi ove informacije za obavljanje prebacivanja konteksta , kad se proces prebacuje iz stanja 3 (ready to run in memory) u 2 (kernel running) ili iz stanja 7 (preemted) u stanje 1 (user running). Takođe, kernel koristi ove infomacije kada obavlja swap operacije za procese između 2 memorijska stanja i 2 odgovarajuća swap stanja, stanje 3 u stanje 5 ili stanje 4 u stanje 6.

[3] Polje za veličinu procesa (**proces size**). Polje ukazuje koliko je veličina procesa, kako bi kernel mogao da alocira dovoljno memorije za proces

[4] Polje za korisničke identifikatore (**user IDs**, **UIDs**). Jedan ili nekoliko korisničkih identifikatora koji određuju privilegije za proces. Na primer UID polja određuju skup procesa koji mogu međusobno da razmenjuju signale između sebe.

[5] Polje za identifikator procesa (**process IDs**, **PIDs**). Jedan ili nekoliko proces identifikatora koji specificiraju relacione odnose između samih procesa i ta polja se postavljaju prilikom sistemskog poziva fork, kada proces ulazi u stanje 8 (created).

[6] Polje za opisivanje događaja (**event descriptor**). Polje koje je značajno kad je proces uspavan

[7] Polje za prioritet (**scheduling parameters**). Parametri ovog polja omogućavaju kernelu da odrede poredak u kome se procesi pomeraju u stanja 2 (kernel running) i stanja 1 (user running)

[8] Polje za signale (**signal field**). Polje označava signale koji su poslati procesu a nisu još obrađeni

[9] Polje za vremenske parametre (**timer parameters**). Parametri u ovom polju označavaju vreme izvršavanja i korišćenje kernelskih resursa koji omogućavaju vođenje statistike (accounting) procesa i određivanje prioriteta jednog procesa.

U-područje (U-area)

U-područje sadrži sledeća polja koja dopunski karakterišu stanja jednog procesa:

[1] Pokazivač na tabelu procesa PT koji identifukuje ulaz iz PT tabele, koji odgovara u-području

[2] Rrealni i efektivni korisnički ID koji opisuju prava pristupa koje ima proces

[3] Vremensko polje koje zapisuje vreme procesa i njegove dece koji su proveli izvršavajući se u korisničkom i kernelskom modu

[4] Polje za signale (**signal array**) koje pokazuje kako će proces reagovati na svaki signal pojedinačno

[5] Kontrolni terminal (**control terminal field**) koji je pridružen procesu

[6] Polje za greške (**errors field**) opisuje greške koji se dogodile u toku sistemskog poziva

[7] Polje za povratnu vrednost (**return value field**) sadrži povratnu vrednost, odnosno rezultat sistemskog poziva

[8] Ulazno-izlazni parametri (**IO parameters**) opisuju količinu podataka za transfer, izvornu i odredišnu adresu u korisničkom prostoru, pomeraj unutar datoteke itd.

[9] Tekući direktorijum i tekući root direktorijum su dva parametra koji opisuju okolinu sistema datoteka za procese

[10] UFDT tabela, koja opisuje sve datoteke koje je proces otvorio

[11] Polja za ograničenja (**limit fields**) koja određuju maximalnu veličinu procesa i datoteka koje proces može napraviti

[12] Polje za pravo pristupa (**permission modes field**) koje definiše inicijalna prava pristupa za novu datoteku

Svaka tranzicija procesa ima poseban tretman sa fizičkom i virtulenom memorijom.

Raspored sistemske memorije

Pretpostavimo da je fizička memorija u opsegu adresa od nule do neke maksimalne vrednosti koja predstavlja ukupnu količinu memorije. Kao što je već naglašeno, proces na UNIX-u se sastoji od tri logičke sekcije: kôd (text) sekcija, sekcija podataka (data) i stek (stack) sekcija. Kod sekcija sastoji se od skupa instrukcija u kojima nalaze programske adrese (skokovi i pozivi procedura), adrese za podatke (za pristup globalnim promenljivama) i adrese za stek (za pristup lokalnim podacima u procedurama).

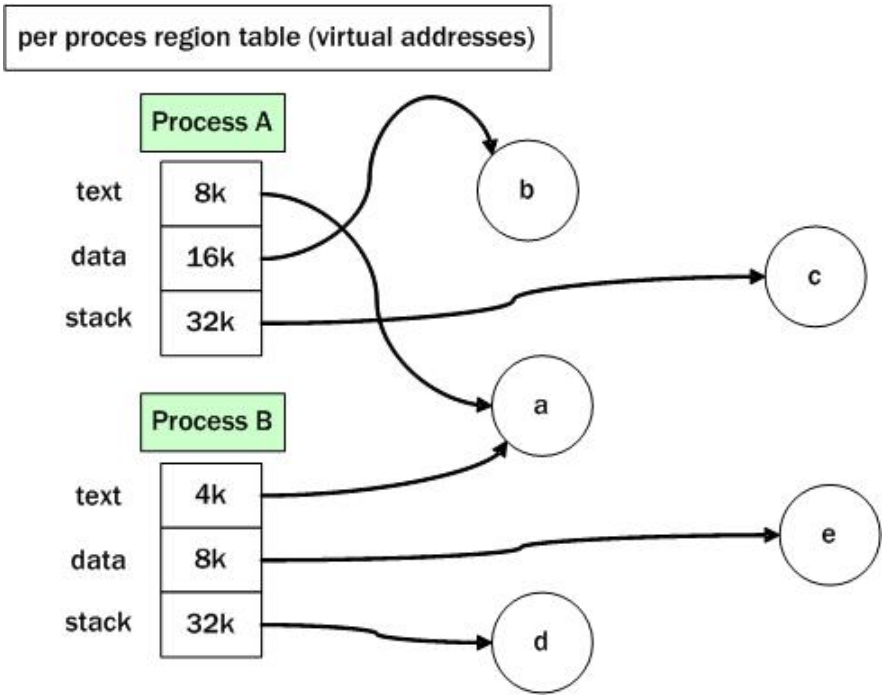
Umesto da se bave fizičkim adresama, prevodioci koriste princip virtuelnih adresa, i oni ne treba da znaju gde će kernel njihove programe napuniti za vreme izvršavanja. Takođe više varijanti istog programa mogu postojati u memoriji, a da svi imaju iste virtualne ali različite fizičke adrese. Deo kernela i hardvera koji obavlja translaciju virtulenih u fizičke adrese naziva se MMU (memory management unit).

Regioni

Kernel kod operativnog sistema UNIX System V deli virtuelni adresni prostor procesa u logičke regione. Region je kontinualno područje virtulenih adresa procesa koje se tretiraju kao poseban objekat koji se može deliti (shared) ili biti privatan (private). To znači da su kôd sekcija, sekcija podataka i stek sekcija posebni regioni procesa. Više procesa mogu da dele jedan region, na primer više procesa koji izvršavaju isti program deliće kôd region, a moguće je da više procesa imaju zajedničke podatke pa će imati deljivi region podataka.

Kernel sadrži tabelu regiona koju ćemo zvati RT (region table) i za svaki region postoji jedan ulaz, koji treba da definiše njegovu virtulenu i fizičku adresu. Takođe, za svaki proces postoji posebna tabela koju ćemo zvati PPRT (per proces region table), a ona se u UNIX terminologiji takođe naziva pregion. Pregion ulazi se mogu nalaziti u tabeli procesa PT, u u-području ili u posebnom delu memorije, zavisno od implementacije operativnog sistema UNIX. Svaki pregion ukazuje na ulaz u RT tabelu i na početnu virtuelni adresu regiona. Takođe, pregion, sadrži zaštitne atribute, kao što su read-only, read-write, read-execute. Deljivi regioni mogu imati različite virtulene adrese za različite procese. Pregion i region tabela RT, podsećaju na tabelu FT (file table) i inode strukturu, što znači da više procesa mogu da dele regione, kao što više procesa mogu da pristupaju datoteci preko tabele FT i inode tabele.

Na slici 6.2 data su dva procesa A i B, sa njihovim regionima, pregionima i virtulenim adresama na koje su regioni priključeni. Oba procesa dele kôd region pod imenom a, koji za jedan proces počinje na virtulenoj adresi 8K, a za drugi proces na virtulenoj adresi 4K, dok su region podataka i stek region privatni.



Slika 6.2. *Primer regiona procesa*

Koncept regiona je nezavistan od memorijske šeme koju primenjuje operativni sistem: da li se primenjuje straničenje (paging) ili segmentacija.

Straničenje i tabele stranica

Priča koja sledi je opšta tako da nije strogo referncirana na operativni sistem UNIX. U memorijskim šemama baziranim na straničenju (paging), MMU deli fizičku memoriju u blokove iste veličine koje se nazivaju stranice, sa tipičnim veličinama od 512 bajtova do 4K i to se definiše preko hardvera.

Svaka adresibilna lokacija u memoriji adresira se preko dve komponente ili uređenog para: (page number, byte offset in page), gde je parametar page number broj stranice, a parametar byte offset in page bajtovski pomeraj unutar stranice.

Na primer ako je fizička memorija 2^{32} a veličina stranice 1K, tada se svaka fizička 32bitna adresa tretira kao par koga obrazuju 22 bita kao broj stranice i 10 bitova kao pomeraj (offset) unutar stranice, kao na sledećem primeru gde imamo heksadecimalnu adresu 58432. Prvo ćemo obaviti njenu binarnu interpretaciju, a potom ćemo 32bitnu adresu razdvojiti na dva dela u razmeri: gornjih 22 i donjih 10 bitova.

- Hexadecimalna adresa: 58432
- Binarna: 0101 1000 01 00 0011 0010
- Broj stranice, pomeraj: 01 0110 0001, 00 0011 0010
- Hexadecimalno: 161 32

Straničenje (Paging) omogućava kernelu da virtulene adrese ne moraju da budu kontinualne u fizičkoj memoriji, mogu da budu razbacane kao blokovi datoteka na disku, ali se teži da stranice jednog regiona budu po mogućtstvu što bliže.

Kernel mapira virtulene u fizičke adrese preko tabele stranica, tako što mapira logičke stranice regiona u fizičke stranice. Pošto je region kontinualni opseg virtuelnih adresa, logička stranica je index u tabeli stranica koja pored mapiranja može sadržati i neke zaštitne bitove, a svaki ulaz u RT sadrži pokazivač na tabelu stranica (PageTable). Tabela stranica je jedna od tipičnih kernelskih struktura podataka. Evo jednog primera mapiranja između logičkih i fizičkih stranica:

Logički broj stranice	Fizički broj stranice
0	177
1	54
2	209
3	17

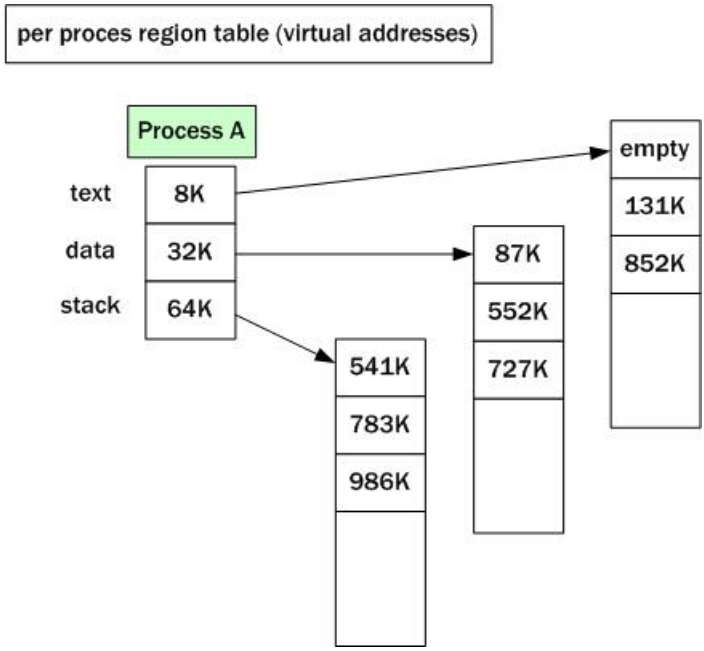
Uzmimo slučaj sa slike 6.3.

Pretpostavimo da je veličina stranice 1K, i pretpostavimo da proces pristupa virtuelnoj adresi 68.432. Pregion ukazuje da je to virtuelna adresa u stek regionu koja počine na 64K (65536) i pretpostavimo da stek raste na gore. Kada oduzmemo ove dve adrese, 68432 -65536, dobijamo pomeraj od 2896 bajtova unutar regiona. Pošto imamo 1K stranice, to je adresa stranice broj 2 (stranice se broje od nule) sa pomerajem od 848 bajtova unutar stranice, a iz tabele stranica čita se fizička adresa stranice i vidimo da je to 986K.

Moderne mašine koriste različite hardverske registre i keš da ubrzaju straničenje.

Pretpostavimo sledeći memorijski model: memorija je organizovana u stranicam veličine 1K, a sistem sadrži skup tripleta memorijskih registara, od koji prvi registar sadrži adresu tabele stranica u memoriji, drugi registar sadrži početnu virtuelnu adresu regiona, a treći sadrži kontrolne informacije, kao što je ukupan broj stranica u tabeli stranica, zatim prava pristupa za te stranice itd. Ovaj model odgovara regionima, a kada kernel priprema proces za izvršavanje , on puni ove triplete na bazi pregiona.

Ako proces generiše adresu koja prevazilazi njegov adresni prostor, pojaviće se izuzetak (exception), a hardver mora da reaguje izvršenjem rutine za obradu izuzetka (exception handler). Takođe, ista situacija će se dogoditi ako proces pokušava da piše po read-only stranici. Ovakve izuzetne situacije obično izazivaju prekid procesa (exit).

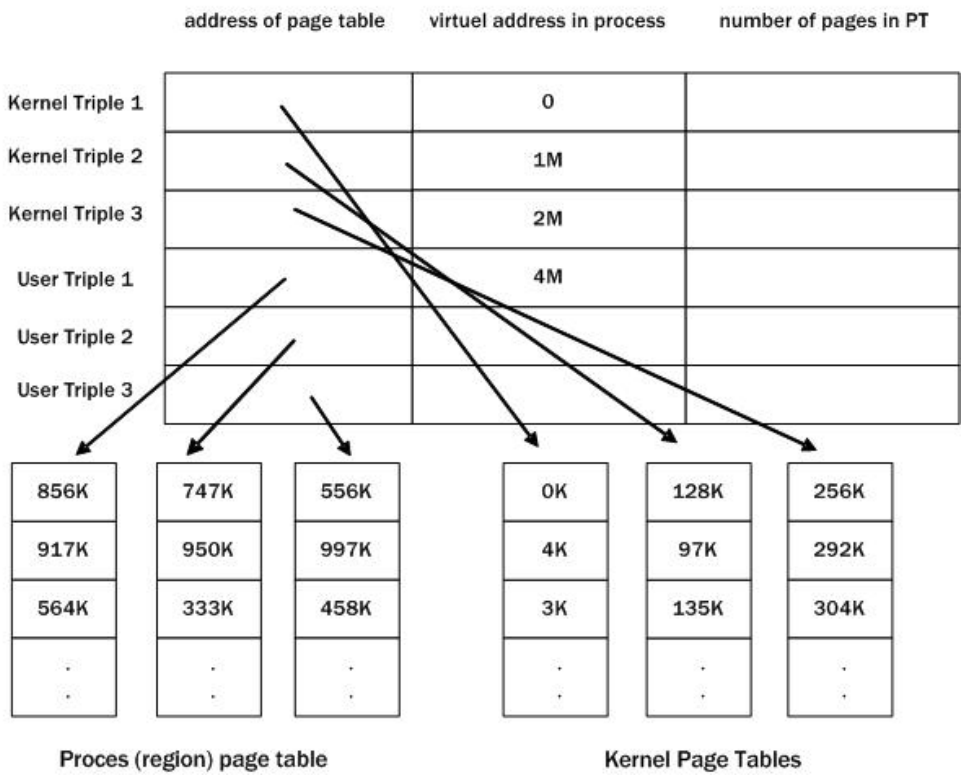


Slika 6.3. *Primer regiona sa tabelama stranica*

Memorijska slika kernela

Mada se kernel izvršava u kontekstu procesa, virtuelna memorija dodeljena kernelu nazavisna je od svih procesa. Kôd i podaci za kernel ostaju u sistemu permanentno, i svi procesi ih dele. Prilikom podizanja sistema, kernelski kôd se puni u memoriju, a postavljaju se potrebne tabele i registri da mapiraju virtuelne adrese u fizičke adrese. Kernelska tabela stranica (PageTable) analogna je tabeli stranica PageTable koja je dodeljena procesima i mehanizam za adresnu translaciju je sličan kao kod korisničkih adresa. Kod mnogih mašina, virtuleni prostor se deli na više klasa, uključujući sistemsku i korisničku klasu, a svaka ima sopstvenu tabelu stranica. Kada se proces prebaci u kernelski mod, dozvoljava mu se pristup kernelskim adresama, a blokira mu se pristup ako je u korisničkom modu. To se postiže tako što operativni sistem sarađuje sa hardverom ili se postavljaju specijalni registri ili biti u CPU (podsetimo, proces prelazi u kernelski mod samo preko sistemskih poziva ili ako se dogodi prekid)

Evo jednog primera na slici 6.4, gde je virtuleni prostor kernela od 0 od 4MB-1, a korisnički prostor preko 4M. Postoje dva skupa registarskih tripleta, od koji prvi skup definiše kernelske adrese, a drugi korisničke. Svaki triplet ukazuje na tabelu stranica (PageTable) koja definiše vezu između virtulenih i fizičkih adresa. Naravno proces će moći da pristupi kernelskim adresama i kernelskim tripletima samo u kernelskom modu.



Slika 6.4. *Registarslki tripleti za kernelski i korisnički adresni prostor*

Neki sistemi organizuju kernelsku memoriju tako da je većina virtuelnih adresa identična sa svojim fizičkim, ali u-područje zahteva virutuelno mapiranje i za kernel.

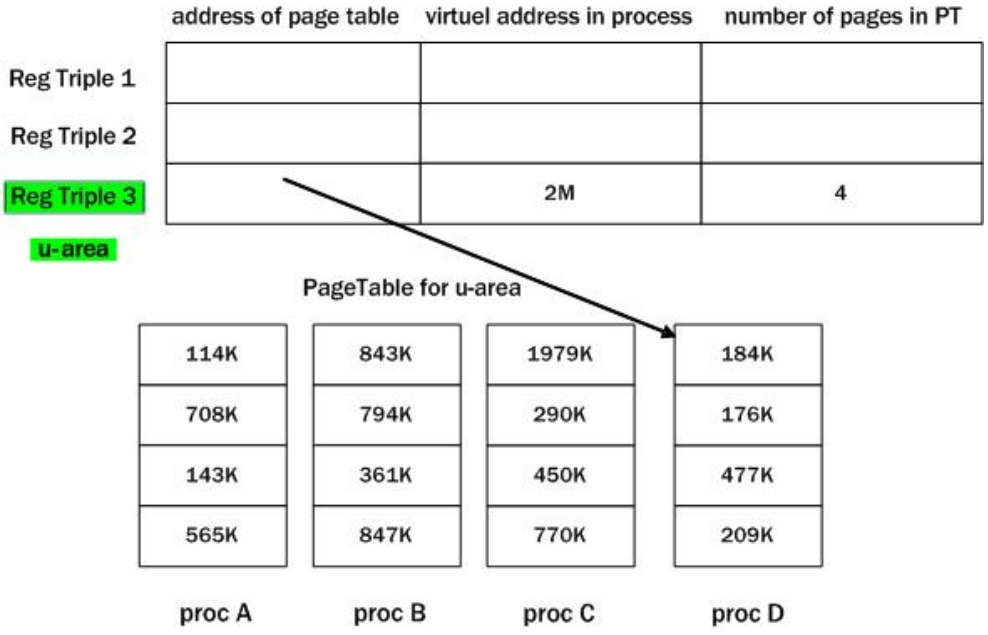
U područje (U area)

Svaki proces ima svoje privatno u-područje, ali kernel toj strukturi pristupa kao da postoji jedno jedino u-područje u sistemu i to od procesa koji se trenutno izvršava. Kernel menja svoju mapu za adresnu translaciju u saglasnosti sa procesom koji se izvršava, pristupajući u-području tog procesa. Kada se kernel operatvnog sistema UNIX kreira, tj prvo se prevede i linkuje, a potom napuni u memoriju, punilac-loader dodeljuje posebnu sistemsku promenljivu pod imenom u, na fiksnu virtulenu adresu. Vrednost ove promenljive i njena adresa su poznati ostalim delovima kernela, a posebno je to značajno za modul koji obavlja prebacivanje konteksta. (context switch).

Adresa u-područja je različita za svaki proces, ali kernel joj pristupa preko iste virtulene adrese.

Svi procesi pristupaju svom u-području isključivo kada su kernelskom modu, a nikako kada su u korisničkom (user) modu. Kernel preko svoje virtulene adrese može pristupati samo jednom u-području i to onom koji pripada aktivnom procesu.

Na primer, pretpostavimo da se u-područje veličine 4K nalazi se na kernelskoj virtulenoj memoriji 2M, kao na slici 6.5. Za kernel su prikazana tri tripleta, prvi triplet se odnosi na kôd kernela, drugi triplet se odnosi na podatke (data) kernela, a treći se odnosi na u-područje (u-area pointer). Treći triplet trenutno ukazuje na u-područje procesa D. Ako bi kernel želeo da pristupa u-području procesa A, kernel kopira odgovarjuću informaciju o tabeli stranice za u-područje procesa A u svoj treći registarski triplet. U ovakvoj šemi, uvek treći triplet mora da ukazuje na u-područje aktivnog procesa. Prilikom prebacivanja konteksta, prva dva tripleta kernela se ne menjaju jer svi procesi dele kernelski kôd i podatke.



Slika 6.5. *Primer za u-područje na 2MB*

6.2. Kontekst procesa

Kontekst procesa sastoji se od sadržine adresnog prostora procesa, sadržine hardverskih registara i kernelskih struktura podataka koje pripadaju procesu. Formalno, kontekst procesa je unija njegovih korisničkog (user-level) konteksta , registarskog konteksta i sistemskog (system-level) konteksta.

Korisnički (user-level) kontekst

Korisnički kontekst sastoji se od: kôd regiona, regiona podataka, regiona steka i deljive memorije (shared memorije), tj od svih memorijskih regiona, koji okupiraju virtuleni adresni prostor procesa

Delovi virtuelnog adresnog prostora procesa mogu biti delimično u memoriji, a delimično na swap prostoru

Registarski kontekst

Registarski kontekst sastoji se od sledećih komponenti:

{1} PC (program counter) specificira adresu sledeće instrukcje koju će CPU izvršavati (adresa je virtulena u kernelskom ili korisničkom prostoru)

{2} PSW (processor status register) sadrži status hardvera. Po pravilu delovi PSW registra sadrže informacije vezane za status zadnje izvršene CPU instrukcije, na primer da li je rezultat pozitivan ili negativan. Drugi delovi registra mogu ukazivati na prekoračenje (carry flag). Veoma bitnu infomaciju predstavlja režim u kom se proces izvršava (kernel mode, user mode), što govori da li proces može izvršavati privilegovane instrukcije i da li može pristupati kernelskim podataka stukturama.

{3} SP (stack pointer) sadrži tekuću adresu sledećeg ulaza u korisničkom ili kernelskom steku. Naravno, CPU arhitektura diktira da li SP ukazuje na prvu slobodnu lokaciju na steku ili na zadnju zauzetu, kao i smer u kome stek memorija raste, na gore ili na dole.

{4} GPRs (general-purpose registers), registri opšte namene sadrži podatke koji se generišu u procesu za vreme izvršavanja

Sistemski (system-level) kontekst

Sistemski kontext procesa ima statički i dinamički deo, pri čemu proces ima jedan statički deo za vreme svog izvršavanja, dok može imati više dinamičkih delova. Dinamički deo sistemskog konteksta trebalo bi da se vidi kao stek nivoa konteksta (stack of contex

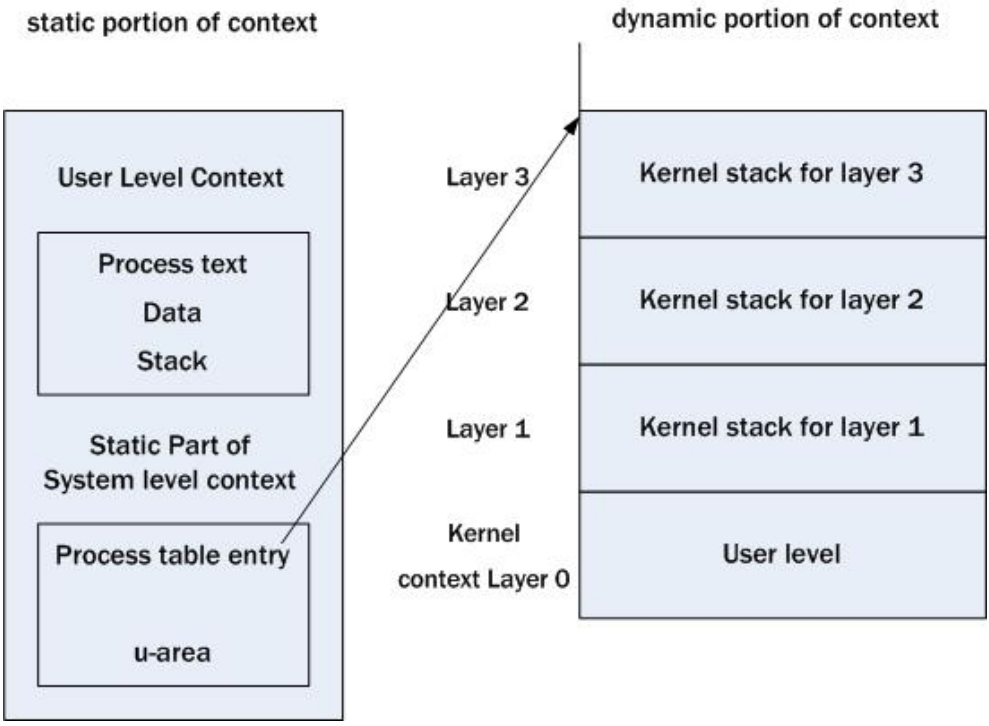
leyers), koje kernel gura i skida sa steka na bazi različitih događaja. Sistemski kontekst sastoji od sledećih komponenti:

- ulaz u tabelu procesa (PT entry) za taj proces, definiše stanje procesa i sadrži kontrolne informacije kojima kernel uvek može pristupiti
- u-područje procesa sadrži kontrolne informacije koje su jedino potrebne u kontekstu procesa. Generalne informacije kao što su prioritet procesa čuvaju se u procesnoj tabeli, pošto se njima pristupa izvan konteksta procesa
- Pregion ulazi, region tabele i tabele stranica, definišu mapiranje između virtuelnih i fizičkih adresa, tj definišu kôd region, region podataka i stek region procesa. Ako više procesa dele iste regione, ti regioni su sastavni delovi konteksta svakog procesa, zato što svaki proces manipuliše regionom nezavisno. Takođe, poseban deo memorijskog upravljanja predstavlja posao (task) koji označava koji deo adresnog prostora nije u memoriji, odnosno nalazi i na swap prostoru.
- Kernelski stek sadrži stek okvire (stack-frames) kernelskih procedura kada se proces izvršava u kernelskom modu. Mada svi procesi izvršavaju identičan kernelski kôd, svi imaju privatne kopije kernelskog steka, koji opisuje specifično pozivanje kernelskih funkcija. Na primer, jedan proces može pozvati sistemski poziv creat i otići na spavanje, dok kernel ne dodeli novu inode stukturu za njega, a drugi proces može pozvati sistemski poziv read i otići na spavanje dok se obavi transfer podataka. Oba procesa izvršavaju kernelske funkcije, ali imaju posebne kernelske stekove. Kernel mora da bude sposoban da obnovi sadržaj kernelskog steka i poziciju SP registra da bi nastavio izvršavanje procesa u kernelskom modu. Mnogi sistemi postavljaju kernelski stek u u-području, mada može da postoji i kao nezavisna celina u memoriji. Kernelski stek se prazni kada se proces vrati u korisnički (user) mod.
- Nivoi konteksta (Context Layers): Dinamički deo sistemskog konteksta procesa sastoji se od skupa nivoa konteksta (layer), koji rade na principu LIFO steka. Svaki nivo sistemskog konteksta sadrži informacije da rekonstruiše prethodni nivo konteksta, a glavni deo čini inforamcije registerski kontekst prethodnog sloja.

Kernel gura (push) novi nivo konteksta na stek kada se: 1. dogodi prekid, 2. kada proces obavi sistemski poziv, ili 3. kada se obavi prebacivanje konteksta. Kernel skida sa steka (pop) nivo konteksa , kada se: 1. obavlja povratak iz obrade prekida, ili 2. kada se završi sistemski poziv ili 3. kada se obavlja prebacivanje konteksta. Prebacivanje konteksta obuhvata push i pop operaciju systemskog nivoa konteksta: kernel gura na stek nivo konteksta starog procesa a skida sa steka nivo konteksta novog procesa. Ulaz u procesnoj tabeli (PT entry) čuva informacije neophodne da se rekonstruiše tekući nivo konteksta.

Na slici 6.6 prikazane su komponente konteksta procesa. Leva strana slike sadrži statičku deo konteksta, koja se sastoji od korisničkog (user-level) konteksta (text, data, stack, shared memory), statičkog dela sistemskog konteksta (ulaz u tabeli procesa , u-područje, pregion ulazi). Na desnoj strani nalazi se dinamički deo sistemskog konteksta,

tj kerneslki stek koji se sastoji od više stek okvira, gde svaki okvir sadrži sačuvani registarski kontekst prethodnog nivoa. Nulti nivo na slici predstavlja stek za korisnički kontekst (user-level), tj korisnički stek .



Slika 6.6. *Statički i dinamički deo konteksta procesa*

Svaki proces se izvršava precizno u okviru tekućeg nivoa konteksta . Broj nivoa konteksta je ograničen na osnovu broja nivoa prekida koje mašina podržava. Na primer ako mašina podržava pet prekidnih nivoa (softverski prekid, terminal, disks, all other peripherals, clock), tada proces može sadržati maksimalno sedam nivoa konteksta: jedan za svaki prekid, jedan za sistemske pozive, jedan za korisnički nivo (user level). Ovakav koncept omogućava da proces podržava najgoru sekvencu da mu se dogodi svih pet prekida.

Iako kernel uvek izvršava kontekst nekog procesa, logička funkcija koja se pri tome dešava ne mora biti vezana za sam proces, kao na primer disk koji prekine proces koji nema nikakve veze sa njim, jer se dogodi prekid pa proces prelazi u kernelski mod, gde se pozove prekidna rutina za disk, a sve je to posledica prebacivanja podataka za neki drugi proces. Rutina za obradu prekida nema uticaja na statički deo konteksta procesa.

Čuvanje konteksta procesa

Kao što je već istaknuto, kernel čuva kontekst procesa tako što gura (push) postojeći kontekst na novi sistemski nivo konteksta (context layer), a to se dešava u sledećim situacijama: 1. kada se dogodi prekid, 2. kada proces izvršava sistemski poziv ili 3. kada kernel obavlja prebacivanje konteksta.

Prekidi (interrupts) i izuzeci (exceptions)

Sistem je odgovoran za upravljanje prekidima, bez obzira da li su to: 1. hardveski prekidi, 2. softverski prekidi (to su instrukcije koje izazivaju softverski prekid) ili 3. izuzeci (exceptions) kao što je na primer greška u staničenju (page fault). Ako procesor radi na CPU nivou izvršavanja koji prihvata prekide, on će prihvati prekid i odskočiti na rutinu za obradu prekida (interrupt handler), pri čemu će podići CPU nivo izvršavanja, tako da niži prekidi ne mogu da prekinu CPU, i na taj način se štite kernelske strukture podataka. Kernel upravlja prekidom sa sledećom sekvencom operacija:

{1} kernel čuva registarski kontekst procesa koji se izvršava i kreira (push) novi nivo konteksta (layer)

{2} kernel određuje izvor uzroka prekida, identifikuje tip prekida i hardversku jedinicu ako ih ima više, na primer ako imate više diskova, treba odrediti ko je izazvao prekid. Kada se primi hardverski prekid, hardver vraća broj koji se naziva prekidni vektor (interrupt vector) koji se koristi kao pokazivač u tabeli, koja se naziva prekidna vektor tabela IVT (Interrupt Vector Table), iz koje se selektuje odgovarajuća prekidna rutina. Na primer ako terminal generiše prekid, a IVT izgleda kao na tabeli 6.1, tada će mašina vratiti prekidni vektor broj 2, pomoću koga će se iz tabele pozvati prekidna rutina (interrupt handler), ttyintr.

Interrupt Number	Interrupt handler
0	clockintr
1	diskintr
2	ttyintr
3	devintr
4	softintr
5	otherintr

Tabela 6.1. Prekidna vektor tabela IVT

[3] Kernel poziva odgovarajuću prekidnu rutinu (interrupt handler). Kernelski stek za novi nivo konteksta (layer) se logički razlikuje od kernelskog steka za prethodni nivo konteksta. Neke implementacije operartivnog sistema UNIX, koriste kernelski stek procesa koji se izvršava, da sačuvaju stek okvire za prekidnu rutinu, a druge implementacije koriste globalni prekidni stek (interrupt stack) da sačuvaju stek okvire za prekidne rutine, nakon čijeg završetka i povratka iz njih, za proces se ne obavlja prebacivanje konteksta .

[4] Prekidna rutina (Interrupt handler) kompletira rad i obavlja povratak. Kernel obavlja sekvencu koja obnavlja registarski kontekst i kernelski stek prethodnog nivoa konteksta, koji je bio u trenutku kada se dogodio prekid i nastavlja izvršavanje obnovljenog konteksta. Prekidna rutina može uticati na proces, zato što menja kernelske strukture i budi uspavane procese, mada u suštini prekinuti proces nastavlja izvršavanje kao da se prekid nije dogodio.

Funkcionisanje prekidne rutine je prikazano sledećim algoritmom.

```
algorithm  inthand    /*handle interrupts*/

input: none
output: none

{
  save (push) current context layer;
  determine interrupt source;
  find interrupt vector;
  call interrupt handler;
  restore (pop) previous context layer;
}
```

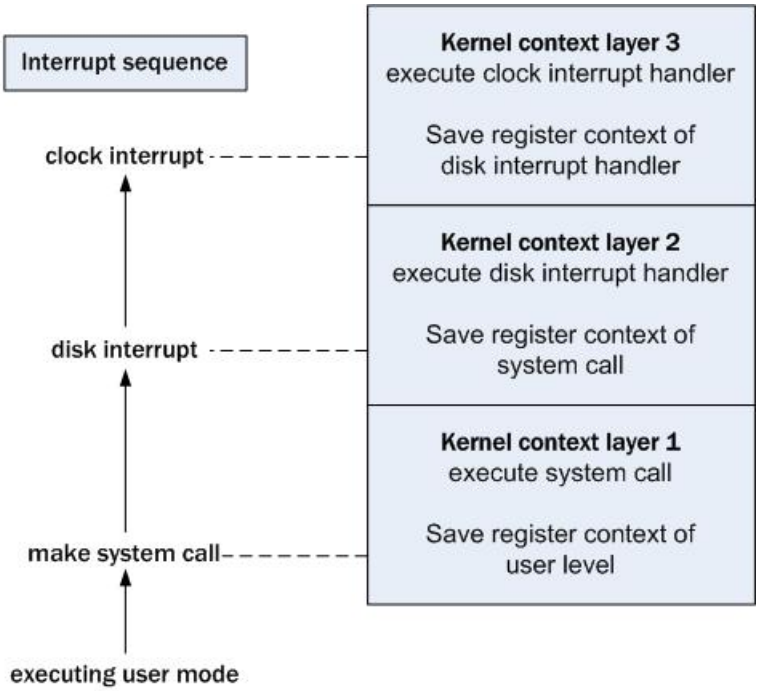
Na mnogim arhitekturama se deo obrade prekida obavlja u samom hardveru zbog ubrzavanja operacija.

Na slici 6.7 prikazana je situacija kada proces obavi sistemski poziv, a dogodi se disk prekid, a tada se dogodi prekidni signal od časovnika. Svaki put kada sistem primi prekidni signal ili obavi sistemski poziv, kreira se novi nivo konteksta i čuva se registarski kontekst prethodnog nivoa.

Interfejs sistemskih poziva

Sistemski poziv liči na običan funkcijski poziv (function call), ali obična funkcija ne može promeniti mod procesa od korisničkog ka kernelskom. C prevodioc koristi unapred definisanu biblioteku sa imenima sistemskih poziva, koje sadrže instrukcije koje menjaju mod u kernelski a te instrukcije se nazivaju zamke operativnog sistema (operating system trap). Rutine biblioteke se izvršavaju u korisničkom modu, ali sistemski poziv je neka vrsta prekidne rutine, tj podseća na nju. Funkcije biblioteke prosleđuju kernelu jedinstveni broj za svaki sistemski poziv, na način koji je zavistan od CPU arhitekture, na

sledeći način: ili 1. kao parametar zazamku operativnog sistema (OS trap), ili 2. u CPU registru ili 3. na steku, a kernel na osnovu zadatog broja određuje koji se sistemski poziv zahteva .



Slika 6.7. *Kernelski stek*

Algoritam za sistemske pozive (syscall)

Sledi prikaz algoritma za sistemske pozive, syscall:

```
algorithm syscall /*algorithm for invocation of SC */

input: system call number
output: result of system call

{
  find entry in SC table corresponding to SC number;
  determine number of parameters to SC;
  copy parameters from user address space to u-area;
  save current context for abortive return;
  invoke SC code in kernel;
```

```
    if (error durring SC)
    {
      set register 0 in user saved register context
         to error number;
      turn on carry bit in PS register in
        user saved register context;
    }
    else
    set register 0, 1 in user saved register contex
      to return values from SC;
}
```

Kada se uđe u zamku operativnog sistema (OS trap), kernel uzima broj sistemskog poziva, na osnovu kojeg pronalazi odgovarajuću kernelsku rutinu za sistemski poziv i uzima ulazne parametre za sistemski poziv. Kernel izračunava adresu (korisničku) prvog parametra za sistemski poziv na korisničkom steku, tako što SP registru doda ili oduzme odgovarajući broj parametra za sistemski poziv. Na kraju, kernel kopira sve parametre za sistemski poziv u u-području. Kada se obavi kôd za sistemski poziv, kernel određuje da li se dogodila greška i ako se dogodila, uzima se lokacija u sačuvanom registarskom kontekstu, setuje se carry zastavica (carry flag) i kôd greške se upisuju u registarsku lokaciju broj nula. Ukoliko greške nije bilo , kernel briše carry flag i kopira odgovajuće povratne vrednosti sistemskog poziva u registre broj nula i broj jedan u sačuvanom registarskom kontekstu. Zatim kernel obavlja povratak iz zamke operativnog sistema u korisnički mod , odnosno u funkciju biblioteke, preciznije u prvu istrukciju iza zamke operativnog sistema, a korisnički kontekst se obnavlja sa povratnim vrednostima, koje opisujuju status obavljenog sistemskog poziva.

Prebacivanje konteksta

Ako se setimo dijagrama procesa stanja, kernel omogućava prebacivanje konteksta (Context switch) u četiri slučaja:

- kada proces samog sebe uspava
- kada proces završava aktivnosti, obavlja exit
- kada se vrati iz sistemskog poziva (kernelski mod) u korisnički mod, pri čemu nije najpogodniji proces koji bi nastavio rad (ima prioritetnijih)
- kada se vrati iz prekida u korisnički mod, ali nije najpogodniji proces koji bi nastavio rad (ima prioritetnijih)

Kernel čuva integtritet svojih struktura sprečavajući proizvoljno prebacivanje konteksta, a mora obezbediti da stanje podataka bude konzistetno pre nego što odobri prebacivanje konteksta: na primer da li su podaci ažurirani, da li su redovi čekanja povezani korektno, da li su brave (locks) postavljeni ili korektno skinuti.

Procedura prebacivanja konteksta podseća na obradu prekida (interrupt handling) ili na sistemski poziv, sa izuzetkom što kernel obnavlja nivo konteksta nekog drugog procesa umesto sa obnavlja prethodni nivo konteksta istog procesa.

Koraci prilikom prebacivanje konteksta su sledeći:

[1] Odrediti da li je na tom mestu moguće realizovati prebacivanje konteksta

[2] Sačuvati kontekst starog procesa

[3] Naći najbolji proces za izvršenje tj nastavak

[4] Obnoviti kontekst novog procesa

Kôd koji implementira prebacivanje konteksta na UNIX-u nije baš lako razumeti, zato što nema klasičnog povratka kao kod sistemkog poziva. Kernel čuva kontekst u jednoj tački procesa, a onda izvršava prebacivanje konteksta i raspoređivanje (scheduling algoritme) u kontekstu starog procesa. Kada kasnije obnovi kontekst starog procesa , on će nastaviti izvršavanje u odnosu na prethodno sačuvani kontekst.

Sledi prikaz pseudo loda (pseudo-code) za prebacivanje konteksta:

```
if (save_context()) /*save context of executing process*/
{
  /* pick another process to run*/
  ….
  ….
  ….
  resume_context (new_process);
  /* never gets here! */
}
/* ressuming process executes from here*/
```

Funkcija save_context obavlja čuvanje konteksta tekućeg procesa i ako bude uspešna, vraća vrednost 1. Od brojnih vrednosti, kernel čuva vrednost PC registra na lokaciji nula u steku i to se koristi kao povratna vrednost iz funkcije save_context(). Kernel nastavlja da izvršava kontekst starog procesa A, tako što bira novi proces B i poziva funkciju resume_context koja će obnoviti kontekst novog procesa B. Kada se obnovi njegov kontekst, CPU izvršava proces B, dok proces A ostaje u svom sačuvanom kontekstu. Njega će aktivirati neki drugi proces, koji će obaviti prebacivanje konteksta i izabrati njega, a on će nastaviti u tački ispod komentara /*ressuming ….*/

Čuvanje konteksta za abortivne povratke

Postoje situacije kada kernel mora prekinuti tekuću sekvencu i neposredno izvrštiti prethodno sačuvani kontekst. U sekcijama ove glave obradićemo situacije kada se proces uspavljuje (sleeping) ili mu se pošalju signali koji nateraju proces da iznenada promeni svoj kontekst. Algoritam koji čuva kontekst je setjmp, dok algoritam koji obnavlja kontekst je longjmp. Metodi su identični kao u funkciji save_context, izuzev što

save_context gura novi nivo konteksta na kernelski stek, dok algoritam setjmp memoriše sačuvani nivo konteksta na u-područje i nastavlja da izvršava stari nivo konteksta . Kada kernel obnavlja kontekst koga je sačuvao algoritam setjmp, on obavlja algoritam longjmp koji obnavlja kontekst iz u-područja.

Kopiranje podataka između kernelskog i korisničkog adresnog prostora

Do sada smo rekli, procesi se izvršavaju u kernelskom ili korisničkom modu ne preklapajući se. Ali videli smo da mnogi sistemski pozivi pomeraju podatke između kernelskog i korisničkog prostora, kao što je kopiranje parametara za sistemski poziv iz korisničkog u kernelski prostor ili kada se kopiraju podaci iz kernel I/O bafera u korisnički prostor. Mnoge mašine dozvoljavaju kernelu da direktno pristupa korisničkom prostoru, što nije baš jednostavno i mora da se kontroliše validnost, odnosno da se prilikom transfera podatka ne prekorači opseg, pa kopiranje zahteva više od jedne operacije jer mora da se proverava validost adresa.

6.3. Algoritmi za manipulaciju adresnim prostorom procesa

Do sada nismo uključivali priču o virtulenoj memoriji i regionima u okviru sistemskih poziva, prebacivanje konteksta , međutim to je jako bitno jer se memorija procesa može dinamički menjati.

Podsetimo da ulaz u tabeli regiona RT sadrži sledeće informacije koje opisuju region:

- pokazivač na inode strukturu datoteke koja je napunjena u region
- tip regiona (text, shared memory, private data, stack)
- veličina regiona
- lokacija regiona u fizičkoj memoriji
- status regiona koji može biti kombinacija:
  - zaključan (locked)
  - traži se (in demand)
  - puni se (in process of being loaded into memory)
  - validan (valid, loaded into memory)
- broj referenci (Reference Count), koji predstavlja broj procesa koji imaju referencu na taj region

Operacije sa regionima

Operacije koje deluju na region su:

- zaključavanje (lock)
- otključavanje (unlock)
- alociranje regiona (allocate)
- priključivanje (attach) regiona u memorijski prostor procesa
- promena veličine (change size)
- punjenje (load) regiona iz datoteke
- oslobađanje (free)
- izbacivanje (detach) regiona iz memorijskog prostora procesa
- duplikacija sadržine regiona

Na primer, kada se izvršava sistemski poziv exec, koji prepisuje korisnički adresni prostor sa sadržajem izvršne datoteke, izbacuju se (detach) se stari regioni, oslobađa se adresni prostor procesa osim ako regioni nisu deljivi, alociraju se novi regioni, priključuju se (attach) se regioni u adresni prostor procesa i regioni pune se sa sadržajem datoteke. Opisaćemo sve region operacije u detalje.

Zaključavanje i otključavanje regiona

Kernel ima mogućnost da obavi zaključavanje i otključavanje (lock i unlock) regiona a da ga ne oslobađa (slično kao kod inode strukture, sa iget i iput algoritmima). Kernel zaključava region da bi sprečio druge procese da manipulišu a njim, a potom ga oslobodja.

Alokacija regiona

Kernel alocira novi region (alogoritma allocreg) za vreme sistemskih poziva fork, exec i shmget (shared memory get). Kernel sadrži tabelu regiona RT (Region Table), čiji ulazi su: ili 1. u povezanoj slobodnoj listi ili 2. u aktivnoj povezanoj listi, slično kao kod bafertskog ili inode keša. Kada se alocira RT ulaz, kernel uklanja prvi raspoloživi ulaz iz slobodne liste, stavlja ga u aktivnu listu, zaključava taj region i markira njegov tip (shared, private). Uz par izuzetaka, svaki proces se udružuje sa izvršnom datotekom, kao rezultat sistemskog poziva exec. Sistemski poziv exec, poziva algoritam allocreg, koji setuje inode polje u RT ulazu, da ukazuje na datoteku sa kojom će se region povezati i kasnije napuniti. Ovako upisana inode struktura identifukuje region u kernelu, tako drugi procesi mogu da dele taj region. Za svaki proces koji deli region, kernel će inkrementirati broj referenci RC (reference count) i sprečiti da se region oslobodi sve dok ga procesi koriste.

Allocreg vraća zaključani alocirani region.

```
algortithm allocreg /* allocate a region data structure*/

input:  (1) inode pointer
        (2) region type
output:
```

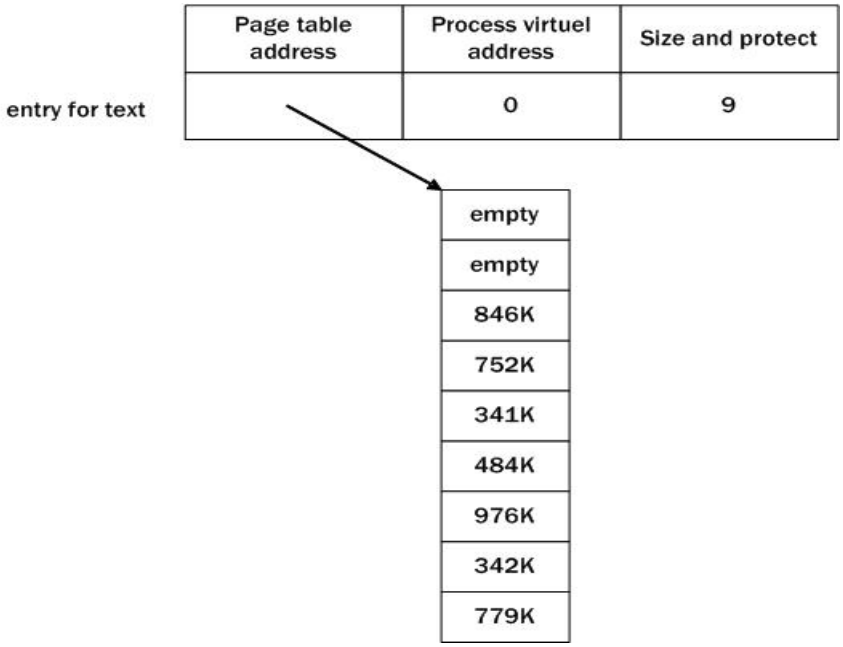
```
{
  remove region from linked list of free regions;
  assign region type;
  assign region inode pointer;
  if (inode pointer not null) increment inode RC;
  place region on linked list of active region;
  return (locked region);
}
```

Priključivanje regiona u proces

Kernel priključuje (attach) region (algoritam attachreg) za vreme sistemskih poziva fork, exec i shmget (shared memory get), pri čemu se region konektuje, tj priključuje u virtuelni adresni prostor procesa. Region može biti novo alocirani region ili neki od postojećih regiona, koga proces deli sa drugim procesima. Kernel alocira slobodan pregion ulaz, postavlja type polje na odgovarajuću vrednost (text, data, shared memory, stack) i upisuje virtulenu adresu na poziciji gde će se region locirati u adresnom prostoru procesa. Proces ne sme mora da proširuje limite za svoju najvišu virtulenu adresu, a virtuelne adrese novog regiona ne smeju da se preklapaju sa postojećim regionima procesa. Na primer, ako proces ima najvišu virtulenu adresu od 8MB, ne može mu se priključiti region od 1M, na virtuelnu adresu od 7.5MB Ako je sve legalno prilikom priključivanja regiona, kernel inkrementira veličinu u ulazu tabele procesa (PT entry), saglasno sa novo priključenim regionom i povećava broj referenci za taj region.

```
algortithm attachreg /* attach a region data structure*/

input:
  (1) pointer to (locked) region is being attached
  (2) process to which region is being attached
  (3) virtuel address in process where region will be attached
  (4) region type
output:  per process region table entry

{
  allocate per process region table entry for process;
  initialize per process region table entry;
    set pointer to region to be attached;
    set type field;
    set virtual address field;
  check legality of virtuel address, region size;
  increment region Reference Count;
  increment process size according to attached region;
  initialize new hardware register triple for process;
  return(per process region table entry)
}
```

Algoritam attachreg zatim inicijalizuje novi skup registarskih tripleta (memory management register triplet) za proces: ako region nije već priključen u neki drugi proces, kernel alocira tabele stranica za region koje će se kasnije popuniti u sistemskom pozivu growreg. Ako je region već priključen u neki drugi proces, koristiće se postojeće page tabele (stranica). Na kraju, attachereg vraća pokazivač na pregion ulaz za novo priključeni region. Na primer, pretpostavimo da kernel želi da priključi postojeći deljivi kod (shared text) region od 7K na virtuelnu adresu nula: kernel alocira novi triplet i inicijalizuje triplet sa pokazivačem na tabelu stranica za region, procesovom početnom virtuelnom adresom nula i veličinom tabele stranica od 9 ulaza, kao na slici 6.8.



Slika 6.8. *Primer pregiona*

Promena veličine regiona

Proces može proširiti ili smanjiti svoj virtuleni adresni prostor sa sistemskim pozivom sbrk. Slično, stek procesa se automaski proširuje ako se poveća dubina ugnježdenih poziva procedura. Interno, kernel poziva algoritam growreg koji će promeniti veličinu regiona. Prilikom ekspanzije regiona kernel mora da obezbedi da se virtuelne adrese proširenog regiiona ne preklapaju sa drugim regionima i da se ne prokorači maksimalna dozvoljena virtuelna adresa procesa. Kernel nikada neće menjati veličinu shared regiona koga deli više procesa.

Algoritam growreg se koristi u dva slučaja:

- kada se primeni sistemski poziv sbrk na region podataka (data region)
- automatsko povećanje korisničkog stek regiona

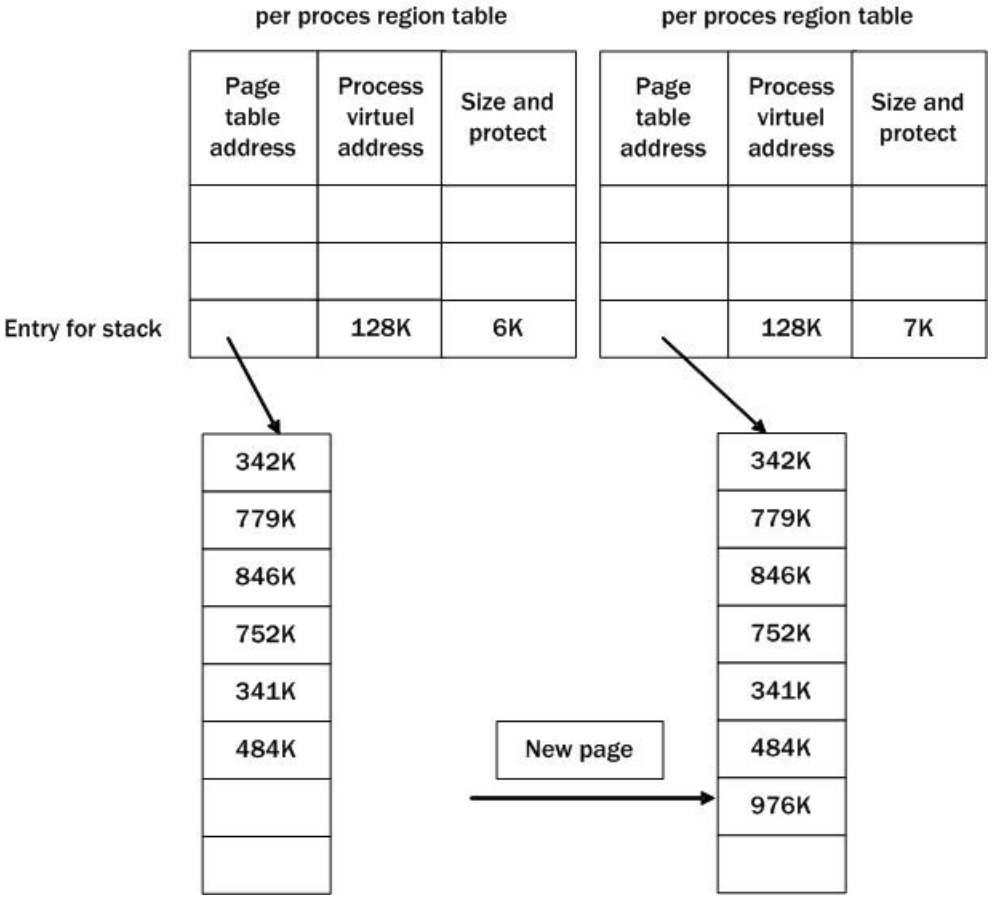
```
algorithm growreg /* change size of a region */

input:  (1) pointer to per proces region table entry
        (2) change in size of region (+ or -)
output:  none

{
  if(region size increasing)
  {
    check legality of new region size;
    allocate auxilliary page tables;
    if(not system supporting demand paging)
    {
      allocate physical memory;
      initialize auxiliary tables, as necessary;
    }
  }
  else /* region size decreasing*/
  {
    free physical memory;
    free auxiliary tables
  }
  do (other) initialize auxiliary tables, as necessary;
  set size field of process table;
}
```

Zapaža se da su oba regiona (data i stack) privatna.

Kernel alocira tabelu stranica ili proširuje postojeću tabelu stranica, da se prilagode novoj veličini regiona. Na sistemu bez DP tehnike (Demand Paging), tj straničenja po zahtevu, mora se alocirati direktno fizička memorija, koja naravno mora biti raspoloživa.

U slučaju smanjivanja regiona, kernel prosto oslobađa memoriju iz regiona. U oba slučaja, podešava se veličina procesa i veličina regiona i modifikuju se pregion ulaz i registarski triplet da odražava promenu.

Na primer, pretpostavimo sa stek region za proces startuje na adresi 128K i da trenutno sadrži 6K i kernel želi da ga poveća za 1K (za jednu stranicu). Ako se poštuju pravila ne preklapanja i svi zahtevi su legalni, veličina procesa će porasti sa 134K na 135K, a stek region će porasti za 1K, a tabela stranica dobija još jedan aktivan ulaz, kao na slici 6.9.



Slika 6.9. *Primer uvećanja steka za novu stranicu*

Punjenje regiona

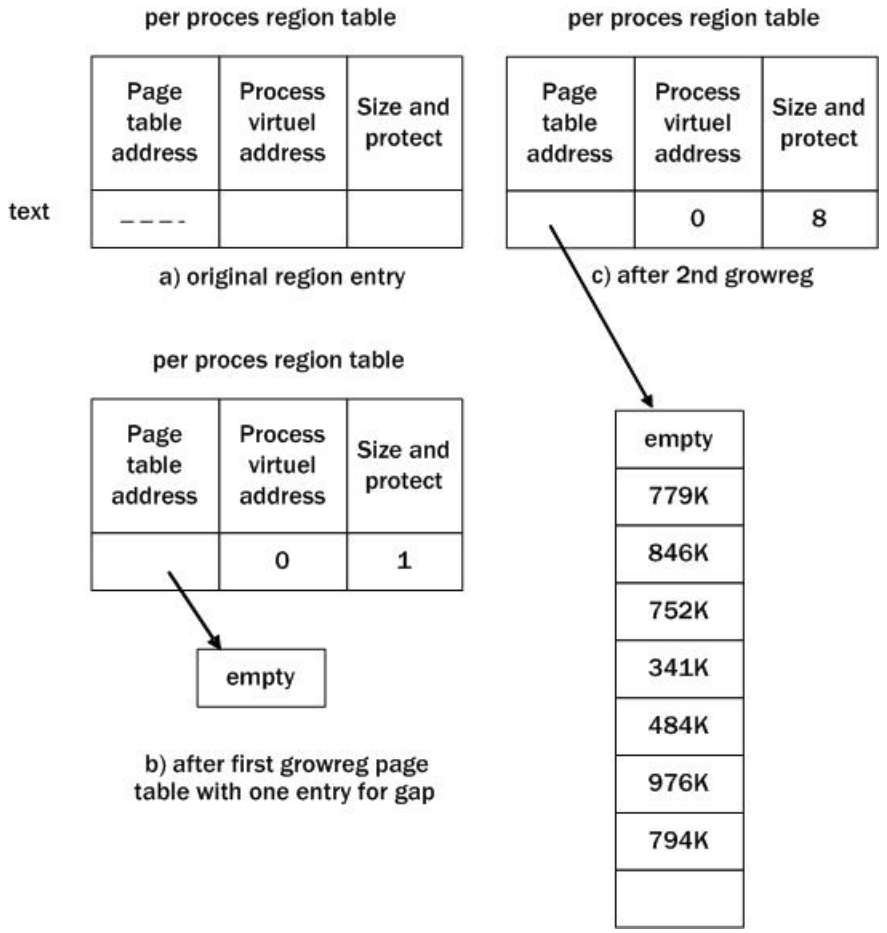
U sistemu koji podržavaju DP tehniku, kernel može mapirati datoteku u procesov adresni prostor za vreme sistemskog poziva exec, organizujući odloženi pristup fizičkim stranicama, na DP bazi tj kada se stranica zatraži. Ako sistem ne podržava DP tehnku, (straničenje po zahtevu), cela izvršna datoteka se mora kopirati u fizičku memoriju. U principu, može se priključiti region na različitim virtulenim adresama u odnosu na one gde je napunjena izvršna datoteka, praveći rupe (gaps) u tabeli stranica. Ova osobina se koristi da se napravi greška u memorijskim referencama (memory faults(kada korisnički program pristupa adresi koja se nalazi u rupi, a na taj način se proces štiti od ilegalnog korišćenja ili od napadača . Gap (praznina) se vrlo često ostavlja.

Da bi se napunila datoteka u region, korisiti se algoritam loadreg, koji obračunava razliku između virtulene adrese regiona i početne virtuelne adrese podataka regiona i onda sledi podešavanje. Tada se region postavlja u stanje „being loaded into memory“ i počinje punjenje regiona iz datoteke, preko read algoritma.

Ako kernel koristi punjenje kod (text) regiona koji će deliti više procesa, moguće je da drugi proces pokuša pristup regionu koji nije još napunjen, zato što prvi proces, tj onaj koji je otpočeo punjenje spava, dok se čita datoteka. Ovde ne može da se primeni zaključavanje (lock), zbog sintakse sistemskog poziva exec. Kernel ovu situaciju razrešava, tako što proveri da li region napunjen, pa ako nije napunjen, proces koji ga traži ide na spavanje, a probudiće ga prekidni signal od diska.

```
algortithm loadreg /* load a portion of file into a region */

input: (1) pointer to per process region table entry
       (2) virtuel address to load region
       (3) inode pointer of file for loading region
       (4) byte offset in file for start of region
       (5) byte count for amount of data loaded
output:  none

{
  increase region size according to eventual size of region
    (algorithm growreg);
  mark region state: being loaded into memory;
  unlock region;
  setup u area parameters for reading file;
    target virtual address where data is read to
    start offset value for reading file
    count of byte to read from file
  read file into region (algorithm read)
  lock region;
  mark region state: completely loaded into memory;
  awaken all process waiting for region to be loaded;
}
```

Na primer, pretpostavimo da kernel puni kod iz datoteke večine 7K u region koji je priključen na virtuelnu adresu nuli, ali želi se rupa od 1K na početku regiona. Kernel prvo mora da alocira RT ulaz preko algoritma allocreg, a zatim se obavi priključenje (attach), preko algoritma attachreg na virtuelnu adresu nula. Zatim se poziva algoritam loadreg, koji će pozvati niži algortiam growreg dva puta, prvi put za rupu od 1K sa praznim tj nevažećim ulazom u tabeli stranica, a drugi put za kod region od 7K sa važećim ulazima u tabeli stranica. Nakon dodele fizičke memorije, kernel puni datoteku u region na virtulenu adresu 1K, kao na slici 6.10.



Slika 6.10. *Primer punjena regiona*

Oslobađanje regiona

Kada region više nije priključen (attached) ni jednom procesu, kernel oslobađa region i vraća ga u listu slobodnih regiona. Ako je regionu pridružena inode struktura, kernel je otpušta preko algoritma iput. Kernel otpušta fizičke resurse vezane za region, kao što su tabela stranica i stranice.

```
algortithm freereg /* free an alocated region */

input:  pointer to a locked region
output:  none
```

```
{
  if (region RC non zero)
  {
    /* some process still using region*/
    release region lock;
    if (region has associated inode) release inode lock;
    return;
  }
  if (region has associated inode) release inode (iput);
  free physical memory still associated with region;
  free aux tables asociated with region;
  clear region fields;
  place region on the free list;
  unlock region;
}
```

Na primer ako kernel želi da oslobodi stek region sa slike 6.9 i ako je broj referenci za region pao na nulu, tada će se osloboditi 7 stranica fizičke memorije i sama tabela stranica za taj stek region.

Izbacivanje regiona

Kernel izbacuje (detach) regione u sistemskim pozivima exec, exit i shmdt (detach shared memory detach). Kernel ažurira pregion ulaz i više stranica fizičke memorije, tako što poništi triplet. Kernel dekrementira broj referenci za region i size polje u ulazu tabele procesa (PT entry) za proces saglasno veličini regiona. Ako je broj referenci jednak nuli, kernel će tada osloboditi region preko algoritma freereg, a ako je broj referenci različit od nule, otpustiće se brava (lock) za region i za inode strukturu, ali će region ostati alociran za druge procese koji ga još uvek koriste.

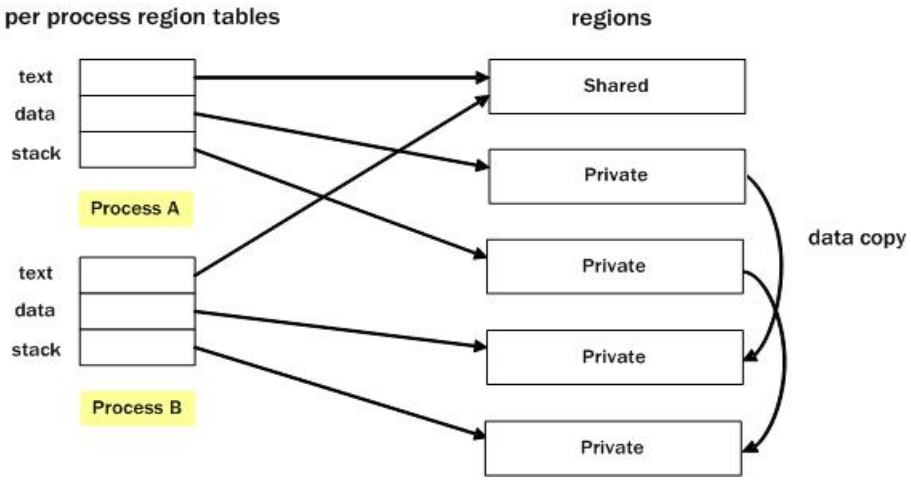
```
algortithm detachreg /* detach a region from a process */

input:  pointer to per process region table entry
output:  none

{
  get aux memory management tables for process,
    release as appropriate;
  decrement process size;
  decrement region RC;
  if (region RC=0) free region (algorrithm freereg)
  else
  {
    free inode lock (if inode associated with region)
    free region lock;
  }
}
```

Dupliranje regiona

Sistemski poziv fork zahteva da kernel duplicira regione procesa. Ako je region deljiv (shared) za procese, kernel nema potreba da fizički kopira region, već se samo inkrementira broj referenci, dozvoljavajući i procesu roditelju i procesu detetu da dele region. Ako region nije deljiv, kernel mora u sitemskom pozivu fork da kopira region, dodeljujući novi ulaz u region tabeli, novu tabelu stranica i fizičku memoriju za region.

```
algortithm dupreg /* duplicate an existing region */

input:  pointer to region table entry
output:  pointer to region that looks identical to input region

{
  if(region type is shared) return(input region pointer)
  /* caller will increment RC*/
  allocate new region (algorithm allocreg);
  setup aux memory structures as exist in iput region
  allocate physical memory for region contents;
  copy region contents from input region to
    newly alocated region;
  return(pointer to allocated region)
}
```

Na slici 6.11 proces A kreira (fork) proces B, kao svoje dete i duplicira svoje regione, pri čemu se text region deli, ali data i stek regioni su privatni. Tako se kreiraju novi regioni koji su kopije regiona procesa A, mada ne mora da se pravi fizička kopija uvek, zahvaljujući tehnici straničenja.

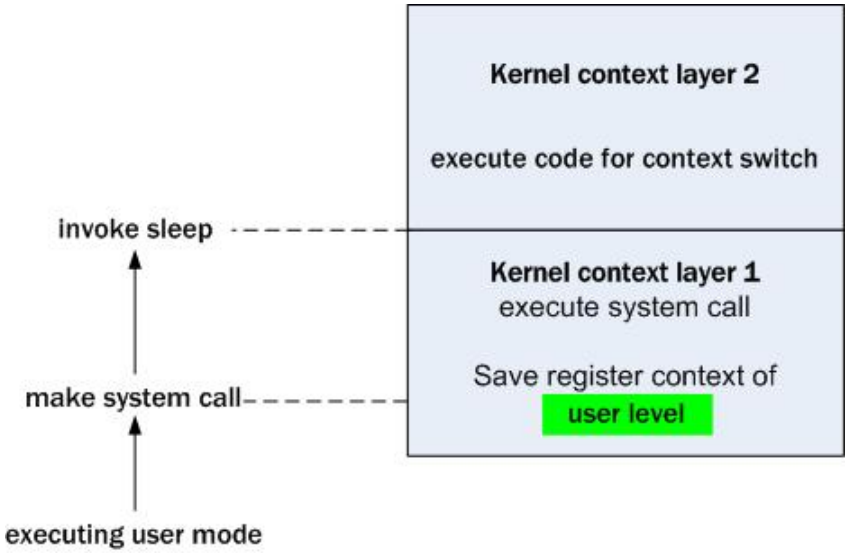


Slika 6.11. *Primer dupliciranja regiona*

Sleep algoritam

Do sada smo obradili niže (low-level) funkcije koje se izvršavaju prilikom tranzicija u stanje 2 i iz stanja 2 (kernel running) osim funkcija za uspavljivanje i buđenje procesa. Objasnićemo algoritam sleep koji menja proces iz stanja 2 (kernel running) u stanje 4 (asleep in memory) i algoritam wakeup koji menja proces iz stanja 4 (asleep in memory) u stanje 3 (ready to run) u memoriji. Takođe postoje 2 slična stanja na swap prostoru.

Kada proces odlazi na spavanje, to se po pravilu dešava preko sistemskog poziva koji to dozvoljava: proces ulazi u kernelski mod, sa nivom konteksa broj 1 (context layer 1,) kada izvršava zamka operativnog sistema (OS trap). U okviru sistemskog poziva proces odlazi na spavanje čekajući na neki događaj. Kada proces ode na spavanje, on obavlja prebacivanje konteksta, gurajući tekući kontekst na stek i izvršavajući nivo konteksta broj 2, (kernel context layer 2), kao na slici 6.12. Proces takođe ide na spavanje kada mu se dogodi PF (greška u straničenju), ako rezultat virtulene adrese koja nije u fizičkoj memoriji i spava dok kernel ne pročita sadržinu stranice.



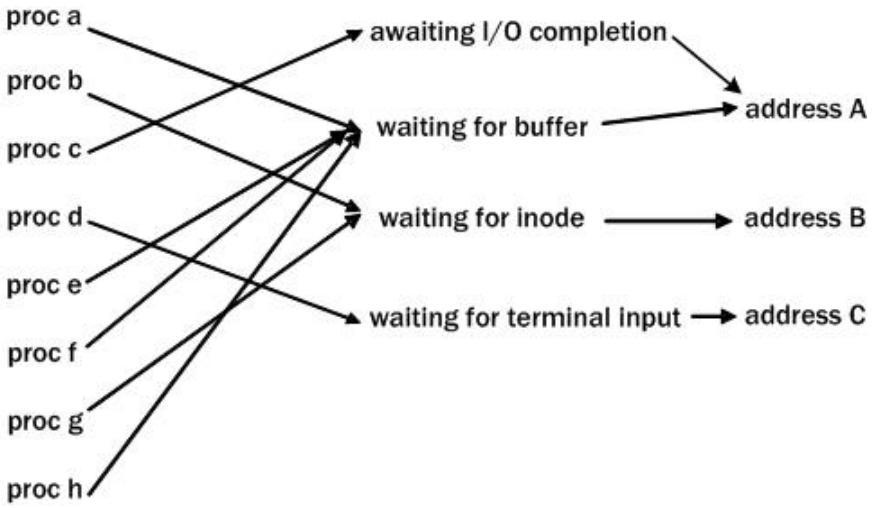
Slika 6.12. *Primer za uspavljivanje (sleep)*

Sleep događaji i adrese

Podsetimo da se kaže da proces ide na spavanje na događaj što znači da spava dok se događaj ne desi, a kada se događaj desi, proces se budi i ulazi u stanje 3 (ready to run), koje može da bude u memoriji ili na swap prostoru.

Mada sistem koristi apstrakciju spavanja na događaj, implementacija mapira skup događaja na skup kernelovih virtulenih adresa. Adrese koje reprezentuje događaje koji se koduju u kernel, i njihovo jedino značenje je to da kernel očekuje događaje, koje mapira na partikularnu adresu. Apstrakcija događaja ne razlikuju koliko procesa čekaju na događaj, a kao rezultat dve anomalije mogu da se dogode i. Prva anomalija je što kernel budi odjedanput sve procese koji čekaju na taj događaj i oni prelaze stanje „ready to run“. Kernel ih ne budi jedan po jedan.

Druga anomalija je što više događaja mogu da se mapiraju na jednu istu adresu. U primeru sa slike 6.13, događaji kao "waiting for the buffer" i "awaiting I/O completion" mapiraju se na adresu bafera ("addr A"). Kada se I/O za bafer kompletira, kernel budi sve procese koji čekaju na obe vrste događaja, zato što proces koji je inicirao I/O i čeka na završetak IO ciklusa, ali je pri tome proces je zaključao bafer, dok drugi procesi čekaju da se bafer oslobodi, tako da praktično dve vrste događaja čekaju na istu stvar.



Slika 6.13. *Primer sleep adresa*

```
algorithm sleep  /* sleep a process*/
input:  (1) sleep address
        (2) priority
output:
  1, if process awakened as a result of signal that
      process catches,
  longjmp algorithm, if process awakened as a result of signal
                     that is not catch
  0, otherwise
```

```
{
  raise processor execution level to block all interrupts;
  set process state "sleep"
  put process on sleep hash queue based on sleep address;
  save sleep address in process table slot;
  set process priority level to input priority;
  if (process sleep is NOT interruptible)
  {
    do context switch;
    /* process resumes execuiton here when if wake up*/
    reset processor execution level to allow inerrupts as
      when process went to sleep;
    return(0);
  }
  /* here process sleep is interuptibile by signal*/
  if (no signal pending against process)
  {
    do context switch;
    /* process resumes execuiton here when if wake up*/
    if(no signal pending against process)
    {
      reset processor execution level to allow inerrupts
        as when process went to sleep;
      return(0);
    }
  }
  remove process from sleep hash queue, if still there;
  reset processor execution level to allow inerrupts
    as when process went to sleep;
  if (process sleep priority set to catch signals) return(1)
  do longjmp algorithm;  /* wakeup*/
}
```

Opis algoritma za sleep je sledeći. Kernel podiže CPU nivo izvršavanja da blokira sve prekide, tako da ne može da se dogodi stanje trke (race condition), kada se manipuliše sa sleep redovima čekanja, a potom se sačuva informacija o starom CPU nivou izvršavanja, da bi se stanje kasnije rekonstruisalo. Nakon toga, stanje procesa se označi kao uspavan (asleep), sačuva se sleep adresa i prioritet u tabeli procesa (PT entry), a potom se proces postavi u hash listu (hash-queue) uspavanih procesa. U prostom slučaju, kada sleep algoritam ne može da se prekida, proces obavlja prebacivanje konteksta i bezbedno je uspavan. Kada se proces probudi, kernel ga kasnije može izabrati za izvršavanje. Proces se restauira iz njegovog konteksta u sleep algoritmu, obnavlja se CPU nivo izvršavanja na vrednost koju je proces imao kada je ulazio u uspavljivanje i obavlja povratak iz sleep algoritma.

```
algorithm wakeup  /* wake up a sleeping process*/
```

```
input: sleep address
output: none

{
  raise processor execution level to block all interrupts;
  find sleep hash queue for sleep address;
  for (every process asleep on sleep address)
  {
    remove process from hash queue;
    mark process state "ready to run"
    put process on scheduller list of process ready to run;
    clear field in PT entry for sleep address;
    if(process not loaded in memory) wakeup swapper process (0);
    else if (awakened process is more elliglible to
             run than currently running process)
      set scheduller flag;
  }
  restore processor execution level to original level;
}
```

Da bi probudio uspavane procese, kernel izvršava wakeup algoritam ili preko sistemskog poziva ili preko prekidne rutine (interrupt handler). Na primer algoritam iput oslobađa zaključanu inode strukturu (zaključanu inode strukturu), i budi sve uspavane procese koji čekaju da se brava (lock) skine. Slično, prekidna rutina za disk (disk interupt handler) budi sve procese koji čekaju na I/O završetak. Kernel podiže CPU nivo izvršavanja da blokira sve prekide dok traje budjenje wakeup. Zatim, za svaki proces koji spava na toj sleep adresi, markira se stanje procesa na "ready to run", proces se uklanja iz povezane liste uspavanih procesa, plasira se u listu za raspoređivanje, briše polje tog procesa u ulazu tabele procesa (PT entry), koje markira njegovu sleep adresu. Ako probuđeni proces nije u memoriji, kernel će probuditi swapper proces da obavi swap-in procesa u memoriju. Takođe, ako je proces koji se probudio većeg prioritete od tekućeg procesa, kernel će postaviti scheduling flag tako da će to izazvati ponovno raspoređivanje. Na kraju kernel obnavlja CPU nivo izvršavanja .

Obe diskusije su uprošćeni sleep i wakeup algoritmi, zato što pretpostavlaju da proces spava dok se ne dogodi događaj. Nekada su ti događaji sigurni, a nekada neizvesni pa kernel može probuditi proces slanjem signala, koji može omogućiti selektivno buđenje procesa.

Na primer, ako proces posalje sistemski poziv read za terminal, onda se čeka dok korisnik ne otkuca nešto sa terminala. To može trajati veoma dugo, a moguće je da se neće ni dogoditi, zato što je neki drugi korisnik prosto ugasio terminal. Može se reći da uspavani pocesi nisu opasni; oni imaju slot u tabeli procesa, ali ne troše procesorsko vreme.

Da bi razlikovao tipove uspavanih stanja, kernel postavlja prioritet raspoređivanja (scheduling) uspavanog procesa, kada ovaj prelazi u uspavano stanje, jer se sleep

algoritam uvek poziva sa parametrom za prioritet. Ako je prioritet iznad neke probojne (threshold) vrednosti, proces se ne budi na signal, već čeka da se desi događaj, a ako je manji od threshold vrednosti, proces se budi na signal.

Ako je signal postavljen kada proces ulazi u sleep algoritam, uspavljivanje (sleep) će se dogoditi ili neće. Ako je prioritet procesa iznad neke probojne (threshold) vrednosti, proces ide na spavanje i eksplicitno čeka sistemski poziv za buđenje (wakeup). Ako je prioritet procesa manji od threshold vrednosti, proces ne ide na spavanje, već prekida sleep algoritam.

Kada se proces probudi kao rezultat signala ili ako nikada nije otišao u sleep zbog prisustva signala, kernel može da obavi longjmp algoritam, zavisno od uzroka uspavljivanja procesa. Kernel obavlja algoritam longjmp, koji obnavlja prethodni sačuvani kontekst, i to u slučaju da nema drugi način da se okonča sistemski poziv ili da se probudi proces. Slično kao u pomenutom primeru, jedan korisnik zadaje čitanje read sa terminala, dok drugi fizički ugasi terminal. Čitanje se se neće obaviti ali se mora javiti informacija o grešci, jer se proces ne budi normalno, nego kernel prvo sačuva kontekts procesa na početku sistemskog poziva sa setjmp algortimom, a potom obavi longjmp, kao rezultat signala koji budi proces.

Postoje situacije kada kernel želi da probudi proces na signal, ali bez longjmp algoritma. Kernel poziva sleep algoritam sa specijalnim prioritetnim parametrom koji izaziva da sleep algoritam vrati vrednost jedan. To je efikasnije nego da prvo obavi setjmp neposredno pre samog uspavjivanja, pa da poziva longjmp da ga budi. Sve je ovo urađeno sa svrhom da kernel sam može da pročisti lokalne strukture podataka. Na primer, drajverski program, može alocirati privatne strukture podataka i otići na spavanje sa neprekidljivim prioritetom, pa ako se probudi zbog signala, trebalo bi da oslobodi te strukture, a onda se po potrebi obavlja algoritam longjmp. Korisnik nema kontrolu kada se obavlja algoritam longjmp, to zavisi od uzroka uspavljivanja procesa i da li kernelske strukture podataka imaju potrebu da se modifikuju, pre nego što se proces vrati iz sistemskog poziva.

7

Kontrola procesa

7.1. Uvod u kontrolu procesa i mehanizam fork

U prethodnim lekcijama, definisali smo kontekst procesa i algoritme za manipulaciju sa kontekstom procesa. Ova lekcija opisuje korišćenje i implementaciju sistemskih poziva koji kontrolišu kontekst procesa.

- fork sistemski poziv kreira novi proces
- exit sistemski poziv završava izvršavanje programa
- wait sistemski poziv omogućava roditelju da sinhroniše svoje izvršavanje sa exitom procesa deteta
- signali informišu procese o asinhronim događajima; kernel sinhroniše izvršavanje sistemskih poziva wait i exit preko signala, pa će oni biti obrađeni prvi
- exec sistemski poziv dozvoljava procesu da pozove novi program prepisujući svoj adresni prostor sa izvršnom datotekom
- brk sistemski poziv dozvoljava procesu da alocira memoriju dinamički, kao što operativni sistem dozvoljava da korisnički stek (user stack) raste dinamički, koristeći sličan mehanizam kao brk.

Na kraju će biti objašnjene glavne petlje u shell i init procesima. Na slici 8.1 su dati odnosi između sistemskih poziva za procese i memorijskih algoritama opisanih u prethodnoj glavi. Svi sistemski pozivi uglavnom koriste i sleep i wakeup algoritam, omogućavajući procesima da se uspavljuju i bude, dok sistemski poziv exec koristi i sistemske pozive za datoteke.

System calls dealing with **memory management**				System calls dealing with **syncronization**			Miscellanous	
fork	**exec**	**brk**	**exit**	**wait**	**signal**	**kill**	**setpgrp**	**setuid**
dupreg attachreg	detachreg allocreg attachreg growreg loadreg mapreg	growreg	detachreg					

Slika 7.1. *Pregled sistemskih poziva za procese*

Kreiranje procesa – fork mehanizam

Jedini način za kreiranje novog procesa na UNIX operativnom sistemu obavlja se preko sistemskog poziva fork. Proces koji poziva sistemski poziv fork naziva se roditeljski proces, dok se novokreirani proces naziva dete proces. Sintaksa za sistemski poziv fork je:

```
pid=fork();
```

Na povratku iz sistemskog poziva fork, dva procesa imaju potpuno identične korisničke (user-level) kontekste, izuzev za pid, koji predstavlja povratnu vrednost sistemskog poziva fork. Za roditeljski proces povratna vrednost pid jednaka je PID (process identifikator) vrednosti za novo kreirani proces dete (child PID), dok je za dete proces povratna vrednost pid jednaka nuli. Proces sa PID jednakim nula, koji interno kreiran od kernela, jedini je proces koji se ne kreira preko sistemskog poziva fork.

Kernel obavlja sledeću sekvencu operacija za sistemski poziv fork:

- Alocira novi ulaz u tabeli procesa (PT entry) za novi proces-dete
- Dodeljuje novi jedinstveni ID za proces dete
- Pravi logičku kopiju konteksta roditeljskog procesa. Pošto izvesni delovi procesa, kao što su kôd (text) region, mogu da se dele između procesa, kernel inkrementira broj referenci za takav region umesto da kopira ceo region na novu fizičku lokaciju u memoriji
- Inkrementira broj referenci za ulaze u tabeli datoteka FT i inode tabelu za datoteke dodeljene procesu
- Vraća ID broj procesa deteta roditeljskom procesu i nultu vrednost za proces dete

Implementacija sistemskog poziva fork nije trivijalna, zato što se proces dete pojavljuje na početku izvršavanja, sama realizacija zavisi od operativnog sistema UNIX, da li primenuje DP ili swaping tehniku. No pretpostavimo, da sistem ili dovoljno fizičke memorije za proces dete, tj za kompletnu memoriju potrebnu za proces-dete.

Algoritam fork

```
algorithm fork
input:  none
output: to parent process child PID number, to child process 0

{
  check for available kernel resources;
  get free proc table slot, unique PID number;
  check that user not running too many processes;
  mark child state "being created";
```

```
    copy data from parent ProcessTable slot to new child slot;
    increment counts on current directory inode and changed root
      (if applicable);
    increment open file counts in FileTable;
    make copy of parrent context (u area, text, stack) in memory;
    push dummy system level context layer onto
      child system level context;
    dummy context contains data allowing child process
      to recognize itself, and start running from here
      when schedulled
    if(executing process is parent process)
    {
      change child state to "ready to run";
      return(child ID);
    }
    else /* executing process is child process*/
    {
      initialize u area timing fields;
      return(0); /*to user*/
    }
  }
```

Algoritam fork – opis i ilustracija

Kernel prvo mora da proveri raspoloživost resursa da bi se fork uspešno obavio. Na swapping sistemima, potrebno je imati dovoljno prostora ili u memoriji ili na swap prostoru (disku) za proces-dete. Na sistemima sa straničenjem (paging) potrebno je samo alocirati memoriju za pomoćne tabele kao što su tabele stranica. Naravno, ukoliko su resursi neraspoloživi, sistemski poziv fork će otkazati.

Kernel prvo nalazi ulaz u tabeli procesa (PT entry) da bi otpočeo konstrukciju konteksta za proces dete, ali mora proveriti da tabela nije već puna, tj. nije veliki ili maksimalni broj procesa već u izvršavanju. Zatim se nalazi jedinstveni ID broj za novi proces, obično za jedan veći od poslednjeg PID-a . Ako drugi proces već ima taj PID, kernel ide na sledeći veći broj, a ako se dostigne neki maksimalni broj, počinje se od nule. Mnogi procesi će trajati kratko u memoriji, pa će obaviti exit i fork će u svakom slučaju naći slobodan PID.

UNIX mora odrediti ograničenje za maksimalni broj procesa koje korisnik može da izvršava simultano. Takođe, ako su svi ulazi u tabeli procesa PT zauzeti, proces mora da čeka. Na drugoj strani, superuser root može naterati svaki proces da prekine aktivnost, odnosno da nasilno obavi sistemski poziv exit. Za superusera kao što je root, jedino ograničenje je maksimalni broj procesa.

Kernel potom inicijalizuje ulaz u tabeli procesa (PT entry) za proces dete, kopirajući razna polja iz roditeljskog slota. Na primer, dete nasleđuje od roditelja realni i efektivni

korisnički ID, roditeljsku procesnu grupu, roditeljsku nice vrednost koja se koristi za izračunavanje prioriteta procesa. Kernel upisuje roditeljski PID u ulaz u tabeli procesa (PT entry) deteta, kako bi dete bilo u stanju da odredi svog roditelja. Zatim se proces-dete ubacuje u stablo procesa i inicijalizuju mu se razni parametri za raspoređivanje, kao što je inicijalni prioritet, inicijalno CPU korišćenje i drugi vremenski (timing) parametri, čime se završava realizacija stanja procesa pod nazivom "being created".

Kernel zatim prilagođava broj referenci za datoteke koje se detetu automatski dodeljuju, to su sve datoteke koje je roditelj otvorio. Prvo će proces dete dobiti isti tekući direktorijumu kao i roditeljski proces. Broj procesa koji koriste tekući direktorijum se povećava za jedan, što se mora upisati u inode strukturu (in-core inode), u polju za broj referenci.

Drugo, ako je proces roditelj ili bilo koji od njegovih predaka ikada promenio svoj root direktorijum (sistemski poziv chroot), dete će naslediti taj promenjeni root i za njegovu inode strukturu se mora inkrementirati broj referenci. Na kraju, kernel pretražuje UFDT za otvorene datoteke roditelja i inkrementira broj referenci za ulaze u globalnoj tabeli datoteka FT, za svaku otvorenu datoteku koja pripada procesu roditelju. Ne samo da dete nasleđuje prava pristupa za datoteku, već se takođe i svi pristupi i pokazivači na datoteke takođe dele između roditelja i deteta, zato što oba procesa imaju identične ulaze u FT. Efekat sistemskog poziva fork sličan je sistemskom pozivu dup za otvorene datoteke: novi ulaz u UFDT ukazuje na isti ulaz u glavnoj FT za otvorenu datoteku, samo što su kod sistemskog poziva dup ulazi u UFDT svi vezani za jedan proces, a kod sistemskog poziva fork, ti UFDT ulazi su potpuno isti, ali pripadaju različitim procesima.

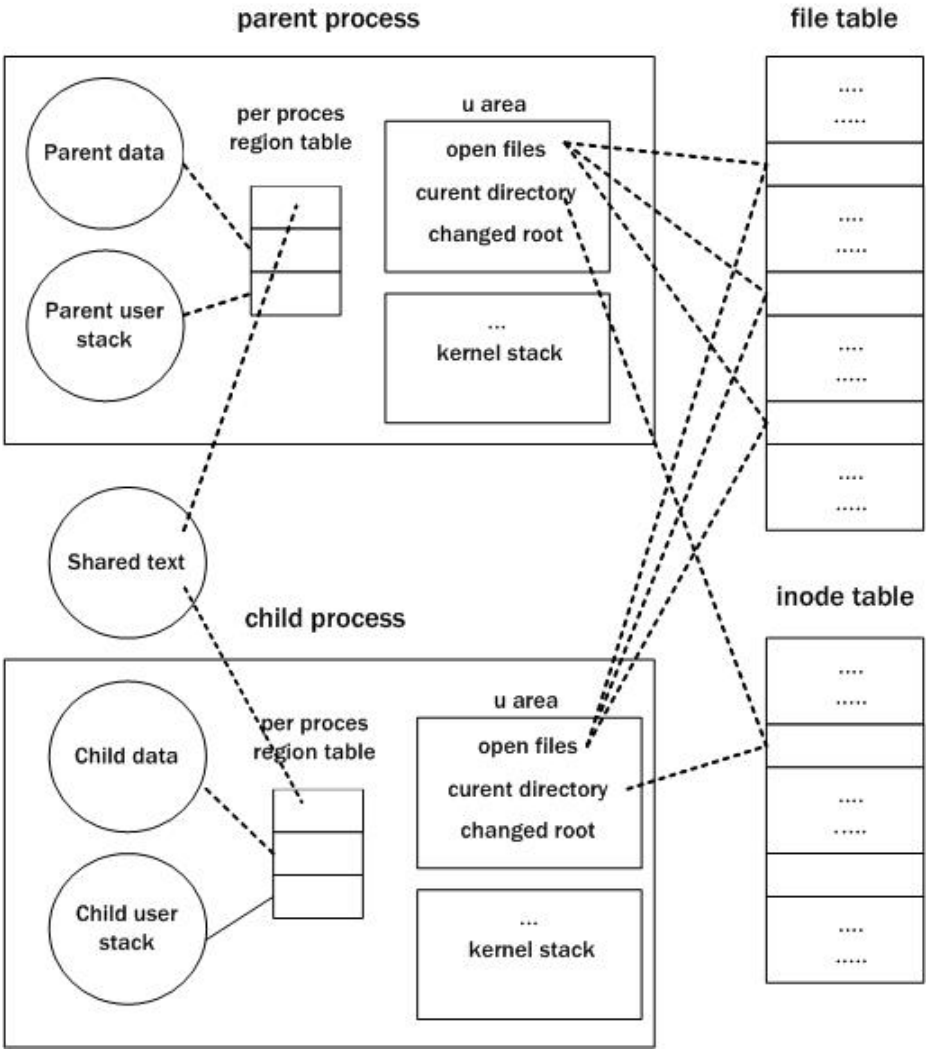
Relacija između procesa roditelja i procesa deteta prikazana je na slici 8.2.

Primer 1 za sistemski poziv fork

Analizirajmo program koji je primer deljenja datoteka preko sistemskog poziva fork. Korisnik će pozvati program sa dva ulazna parametra: ime postojeće datoteke i nove datoteke koja će biti kreirana. Proces otvara postojeću datoteku, kreira novu datoteku i ako nema grešaka, preko sistemskog poziva fork kreira proces dete. Interno, kernel pravi kopije roditeljskog konteksta za proces dete, a potom se roditeljski proces izvršava u jednom adresnom prostoru a dete u drugom. Svaki proces pristupa privatnim kopijama globalnih promenljivih fdrd, fdwt i c i privatnim kopijama stek promenljivih argc i argv, ali samo svojim kopijama, tu nema mešanja. Međutim, kernel kopira u-područje originalnog procesa za vreme sistemskog poziva fork, pa dete nasleđuje pristup roditeljskim datotekama.

I roditelj i dete nezavisno zovu funkciju rdwrt, koja izvršava petlju, čita jedan bajt izvorne datoteke i upisuje u odredišnu (target) datoteku. Funkcija se završava kada sistemski poziv read dostigne kraj datoteke. U ovom slučaju svaki proces ima sopstvene deskriptore datoteka, a to su fdrd i fdwr, ali oni pokazuju na iste ulaze u tabeli datoteka FT, pa svaki proces modifikuje pomeraj za čitanje (read offset) na svako čitanje (read) i pomeraj za upis (write offset) na svaki upis (write). Oba procesa roditelj i dete, nikada neće imati isti pomeraj, pa nema prepisivanja. Efekat je kopiranje iste izvorne datoteke u

istu odredišnu datoteku, preko 2 procesa, ali sadržaj odredišne datoteke zavisi od redosleda izvršavanja procesa, odnosno sistemskih poziva read i write. Na primer dva bajta "ab" u originalu mogu završiti u kopiji ili kao "ab", ako proces obavi read/write ali i kao "ba" ako bude read, read, write, write.



Slika 7.2. *Relacija između procesa roditelja i procesa deteta*

```
#include <fcntl.h>
int fdrd, fdwt;
```

```
main (argc, argv)
int argc;
char *argv[]
{
  if(argc != 3) exit(1);
  if((fdrd = open(argv[1], O_RDONLY)) == -1) exit(1);
  if((fdwt = creat(argv[2], 0666)) == -1) exit(1);
  fork();
  /*both processes execute same code*/
  rdwrt();
  exit(0);
}

rdwrt()
{
  for(;;)
  {
    if(read(fdrd, &c, 1) != 1) return;
    write(fdwt, &c, 1);
  }
}
```

Primer 2 za sistemski poziv fork

Analizirajmo sledeći program u kome dete nasleđuje deskriptore datoteka 0 i 1 (standard input i output) od svog roditelja. Izvršavanje svakog sistemskog poziva pipe alocira dva deskriptora datoteka u polju to_par i to_chil, redom. Proces obavlja sistemski poziv fork i time kopira svoj kontekst: svaki proces pristupa svojim privatnim podacima. Proces roditelj zatvara svoj standardni izlaz (output) (file-descriptor 1) i duplira (sa dup) write-deskriptor koga je dobio od sistemskog poziva pipe(to_chil). Pošto se prvi slobodan ulaz u roditeljskoj UFDT kreira preko sistemskog close(1), kernel preko sistemskog poziva dup(to_chil[1]) kopira pipe write descriptor u ulaz 1 u UFDT, tako da ukazuje na pipe datoteku to_chil. Roditeljski proces obavlja sličnu operaciju, da bi postavio svoj standardni ulaz sa fd=0 na read pipe deskriptor to_par[0], odnosno ulaz uzima iz pipe datoteke to_par.

Slično, proces dete zatvara svoj standarni ulaz (fd=0) i duplira pipe read descriptor, od pipe datoteke to_chil, tako da ulaz uzima iz svoje pipe datoteke, to_chil. Proces dete obavlja sličan skup operacija da bi napravio svoj standardni izlaz, i to će biti pipe write descriptor za datoteku to_par. Oba procesa zatvaraju file-deskriptore koje im je vratio sistemski poziv pipe, što je dobra programerska praksa. Kao rezultat toga, kada roditelj upisuje nešto na svoj standardni izlaz, to se upisuje na pipe datoteku to_chil, odnosno podaci se šalju procesu detetu, koji čita pipe kao svoj standardni ulaz. Kada dete upisuje nešto na svoj standardni izlaz, to se upisuje na pipe datoteku to_par, odnosno podaci se šalju procesu roditelju, koji čita taj pipe kao svoj standardni ulaz. Procesi tako razmenjuju poruke preko ove dve pipe datoteke.

Rezultat ovog programa je invarijantnost, efekat je isti bez obzira kojim se redosledom roditelj i dete izvršavaju iza sistemskog poziva fork. Na primer ako dete obavi svoj sistemski poziv read pre nego što roditelj napravi sistemski poziv write, dete će se uspavati sve dok roditelj ne pošalje nešto na pipe datoteku.

Ako roditelj obavi sistemski poziv write, a dete to ne pročita, on će da čeka na write, tj. upis u pipe. Ovde je poredak izvršenja fiksan: svaki proces kompletira sistemske pozive read i write i ne može da kompletira svoj sledeći read dok drugi proces ne kompletira svoje sistemske pozive read i write. Preciznije, prvo roditelj upisuje, pa dete čita, pa dete upisuje, pa roditelj čita, pa roditelj upisuje itd.

Roditeljski proces izlazi posle 15 iteracija, a dete zatim čita EOF (end of file) zato što pipe datoteka nema više proces writer i završava (exit). Ako je dete upisalo nešto u pipe datoteku, nakon što je roditelj izašao, primiće signal da je upisao na pipe datoteku, za koju nema ni jednog reader procesa.

Dobra programerska praksa podrazumeva zatvaranje suvišnih deskriptora datoteka iz tri razloga. Prvo, time se smanjuje opasnost od sistemskog ograničenja za ukupan broj deskriptora datoteka. Drugo, ako proces dete obavi sistemski poziv exec, deskriptori datoteka (fd) ostaju dodeljeni novom kontekstu. Zatvaranje nepotrebnih deskriptora datoteka omogućava programima da se izvršavaju u čistom okruženju sa jedino otvorenim deskriptorima fd 0, 1 i 2 . Na kraju, čitanje iz pipe datoteke vraća status EOF, ako nema više procesa koji pišu u pipe datoteku. Ako proces čitaoc-reader drži otvoren pipe deskriptor za upis (write-descriptor), nikada se neće znati da li je proces-pisac zatvorio njegov kraj pipe datoteke. Prethodni primer ne radi dobro ako dete ne zatvori svoj write pipe deskriptor, pre ulaska u svoju petlju.

```
#include <string.h>
char string [] = "hello word"
main()
{
  int count,i;
  int to_par[2], to_chil[2]
  char buf[256];
  pipe(to_par); pipe(to_chil);
  if(fork() == 0)
  {
    /*child process executes here*/
    close(0); /*close old standard input*/
    dup(to_chil[0]);   /*dup pipe read to standard input*/
    close(1);          /*close old standard output*/
    dup(to_par[1]);    /*dup pipe write to standard output*/
    close(to_par[1]);  /*close unnecessary pipe descriptors*/
    close(to_chil[0]);
    close(to_par[0]);
    close(to_chil[1]);
    for (;;)
```

```
    {
      if((count == read(0, buf, sizeof(buf) == 0) exit();
      write(1, buf, count); }
    }
    /*parent process executes here*/

      close(1); /*rearrange standard in, out */
      dup(to_chil[1]);
      close(0);
      dup(to_par[0]);
      close(to_chil[1]);
      close(to_par[0]);
      close(to_chil[0]);
      close(to_par[1]);
      for ( i = 0; i<15, i++)
      {
         write(1, string, strlen(string));
         read(0, buf, sizeof(buf));
      }

}
```

7.2. UNIX Signali

Signali informišu procese da su se dogodili asinhroni događaji. Procesi mogu jedan drugome slati signale sa sistemskim pozivom kill, ili kernel to obavlja interno. U sistemu UNIX System V postoje 19 signala koji se mogu klasifikovati na sledeći način:

- signali koji prekidaju procese, šalju se kada proces obavlja exit ili kada proces poziva sistemski poziv sa "death of child" parametrom
- signali vezani za exception uslove (adresa izvan virtuelnog opsega, write u read only područje..)
- signali vezani za nepopravljive uslove za vreme sistemskih poziva, kao na primer nedostatak sistemskih resursa
- signali izazvani neočekivanom greškom za vreme sistemskih poziva, kao što je pogrešan ulazni parametar za sistemski poziv, upis u pipe datoteku koji nema reader proces
- signali generisani od procesa u korisničkom modu, kad proces želi da primi alarm signal posle izvesnog perioda vremena
- signali vezani za terminalsku komunikaciju

- signali za praćenje izvršavanja procesa

Obrada signala ima više aspekata, od načina kako se signali šalju, potom kako proces upravlja signalom i kako proces kontroliše svoje reakcije na signal. Da bi poslao signal procesu, kernel postavlja bit u signal polju u ulazu procesne tabele (PT entry), a svakom tipu signala odgovara po je jedan bit. Ako je proces uspavan, kernel će ga probuditi prilikom slanja signala. Proces može zapamtiti više tipova signala ali nema informaciju koliko puta je primio isti signal. Na primer, ako proces primi signale hangup i kill, postaviće oba bita u signal polju svog PT ulaza, ali nema tragova koliko je puta primio te signale.

Kernel proverava da li je primljen signal kada je proces spreman da se vrati iz stanja 2 u 1 ili kada se proces budi. Proces upravlja signalima jedino kada se proces vraća iz stanja 2 u 1, tako da signal nema efekta u kernelskom modu. Ako je proces u korisničkom modu i pošalje mu se signal, slanje signala će napraviti prekid, a signal će se obraditi po povratku iz prekida kada je proces spreman da napusti kernel mod. Znači obrada signala je isključivo u kernelskom režimu.

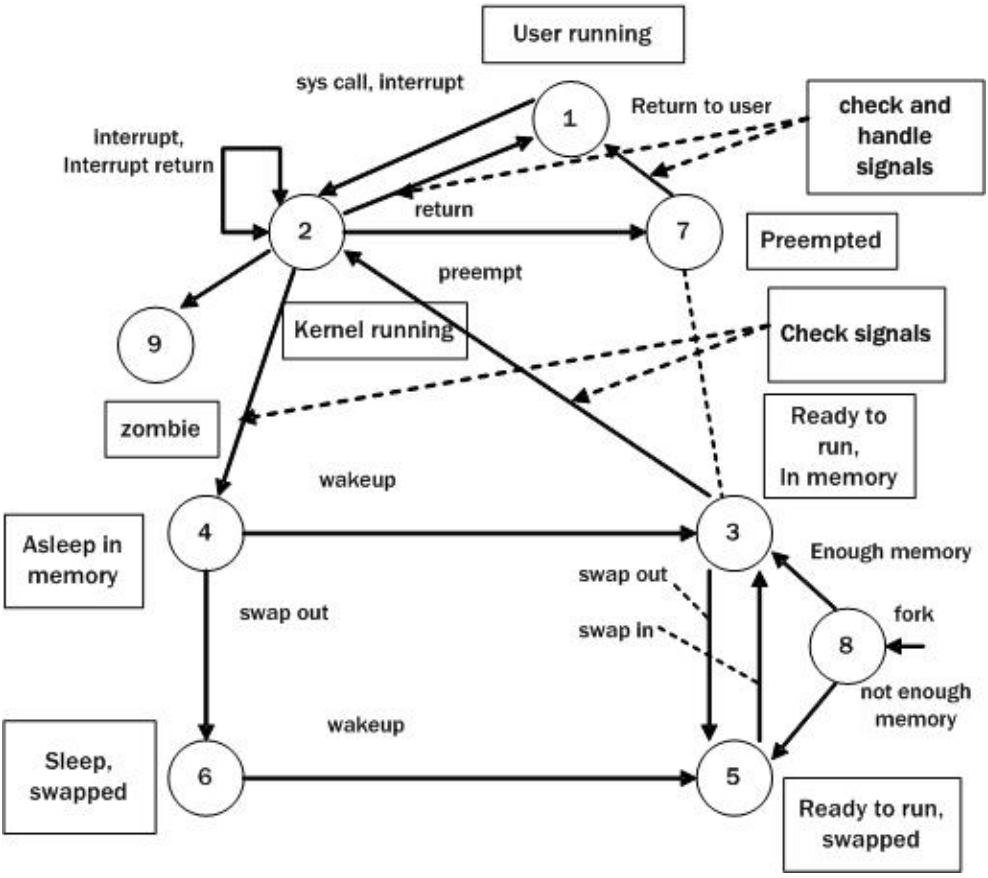
Pozicija za detekciju i obradu signala u dijagramu stanja procesa je prikazana na slici 7.3.

Detekcija signala – algoritam issig

Demonstriraćemo kernelski algoritam issig, koji se određuje da li je proces primio signal (za sada ne obrađujemo signal "death of child", a naglasimo da proces može izabrati da ignoriše signale koristeći sistemski poziv signal, u kome kernel prosto isključuje indikaciju onih signala koje proces želi da ignoriše, ali prisustvo signala kill se ne ignoriše.

```
algorithm issig  /* test for receipt of signal*/

input: none
output: true, if process received signals that it does not ignore
        false otherwise

{
  while(received signal field in PT entry not 0)
  {
    find a signal number sent to process;
    if(signal is death of child)
    {
      if(ignoring death of child signal)
        free PT entries of zombie children;
      else if (caching death of child signals) return(true);
    }
    else if (not ignoring signal) return(true);
    turn off signal bit in received signal field in PT entry;
```

```
    }
    return(false);
}
```



Slika 7.3. *Pozicija za testiranje i obradu signala u dijagramu stanja procesa*

Upravljanje signalima

Kernel upravlja signalima u kontekstu procesa koji ih prima, tako da proces mora da obavlja upravljanje signalima. Postoje tri slučaja za upravljanje signalima:

- proces se prekida po prijemu signala
- proces ignoriše signal
- proces izvršava posebnu (korisničku) funkciju po prijemu signala

Podrazumevana akcija po prijemu signala je prelazak u kernelski mod, ali proces može specificirati specijalnu akciju koja se preuzima po prijemu signala, a to se postiže preko sistemskog poziva signal. Sintaksa za sistemski poziv signal je:

```
oldfunction = signal(signum, function);
```

pri čemu je signum signal za koji se specificira akcija, function je adresa korisničke funkcije koja se preuzima po prijemu signala, a povratna vrednost oldfunction je vrednost koju vraća sistemski poziv signal, a predstavlja prethodno definisanu vrednost funkcija za obradu tog signala. Proces može postaviti 0 ili 1 umesto funkcije, pri čemu 1 ignoriše signal, a 0 ga prihvata i prebacuje proces u kernelski mod. U-područje sadrži vektorsko polje od rutina za obradu signala (signal-handler), po jednu rutinu za svaki signal koji postoji na UNIX operativnom sistemu. Kernel čuva adresu korisničke funkcije u polju koje odgovara signalu, a svaka rutina za jedan tip signala je nezavisna od drugih. Upravljanje signalom je prikazano u psig algoritmu.

```
algorithm psig
/* handle  signals after recognizing their existance */

input: none
output: none

{
  get signal number set in PT entry;
  clear signal number set in PT entry;
  if(user had called signal SC to ignore this signal) return;
    /* done */
  if(user specified function to handle the signal)
    /*signal catcher*/
  {
    get user virtual address of signal catcher stored in u-area;
    /* the next statement has undesirable side-effects*/
    clear u area entry that stored address in signal catcher;
    modify user level context;
    artificially create user stack frame to mimic call to
      signal catcher function;
    call to signal catcher function;
    modify system level context;
    write address of signal catcher into program counter field
      of user saved register context;
    return;
   }
    if(signal is type that system should dump
       core image of process)
    {
      create file named "core" in current directory;
      write contents of user level context to file "core";
```

```
        }
        invoke exit algorithm immediately;
    }
```

Kada obrađuje signal, kernel određuje tip signala i ukida odgovarajući signal bit u PT ulazu koji se postavlja kada se signal primi. Ako je signal-handling funkcija postavljena na nulu (default), kernel će sačuvati memorijsku sliku procesa "core dump" za izvesne tipove signala pre završetka procesa. Ta slika je pogodna za programere i omogućava im da ispravljaju greške u svom programu. Kernel kreira sliku za one signale koji su poslati u slučaju da nešto nije u redu sa procesom, na primer kad izvršava ilegalnu instrukciju ili ako prekorači svoju virtuelnu adresu. Kernel ne pravi sliku (dump) za one signale koje ne uključuju neku programsku grešku. Na primer, kada korisnik pošalje break taster sa terminala, to znači da korisnik želi da prekine proces i prekidni signal znači da je sa procesom sve u redu, pa se ne pravi slika (core dump). Na drugoj strani, quit signal će uzrokovati da se generiše slika (core dump) tekućeg procesa, što se postiže pritiskom control-vertical-bar na tastaturi, a jako je pogodan za zaglavljene procese, koji se tada prekidaju sa core dumpom.

Kada proces primi signal koji je prethodno rešio da ignoriše, proces nastavlja izvršenje kao da se signal nikada nije desio, a kernel ne resetuje polje u u-području koje definiše ignorisanje signala; proces će ignorisati svaku novu pojavu signala. Ako proces primi signal koga je prethodno odlučio da hvata (catch), izvršiće se korisničku funkcija neposredno po povratku u korisnički mod, s tim što će kernel pre toga da obavi sledeće akcije:

- kernel pristupa sačuvanom korisničkom registarskom kontekstu, nalazi PC i SP koji su sačuvani za povratak u korisnički mod.
- briše signal handler polje u u-području, postavljajući ga u default (podrazumevano) stanje
- kernel kreira novi stek okvir (frame) na korisničkom steku, u koga upisuje PC i SP iz korisničkog konteksta i to će se korisititi ako se zbog signala pozove korisnička funkcija (signal catcher function)
- kernel menja stanje sačuvanog korisničkog registarskog konteksta: PC se postavlja na adresu signal catcher funkcije, a SP se podešava da povratak bude u pravi korisnički kontekst

Po povratku u korisnički mod, proces će prvo izvršiti signal catcher funkciju, a potom se vraća u svoj originalni kontekst

Na primer, slika sadrži program koji hvata prekidni signal (SIGINT) i šalje sebi prekidni signal.

```
#include <signal.h>
main()
{
```

```
  extern catcher();
  signal(SIGINT, catcher);
  kill(0, SIGINT)
}
catcher() {}
```

a externa rutina je data u VAX asembleru

```
_catcher()
104:
106: ret
107: halt
dok je assembleslerska rutina za kill
_kill()
108:
   # next line traps into kernel
10a: chmk $0x25 #change to kernel mode
10c: bgequ 0x6 <0x114>
10e: jmp 0x14(pc)
114: clrl  r0
116: ret
```

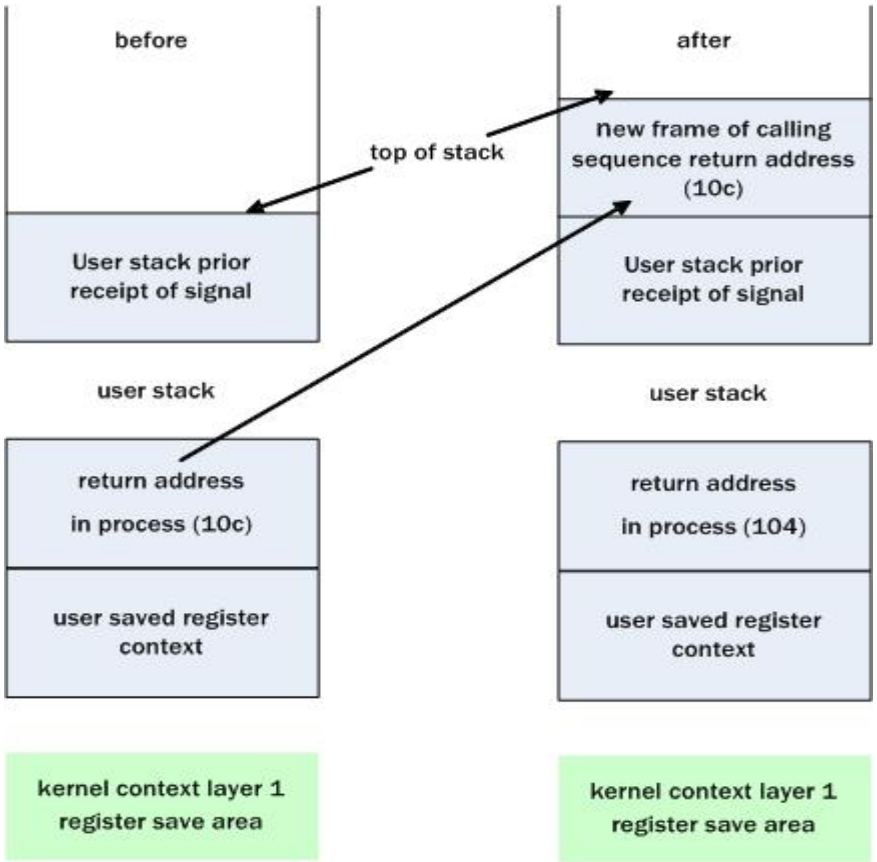
Izgled korisničkog i kernelskog steka pre i posle prijema signala (SIGINT) dat je na slici 7.4. Zapazite da je povratak is sistemkog poziva kill na adresi 10c, a da catcher funkcija počinje na adresi 104, pri čemu sistemski poziv kill, generiše signal SIGINT.

Anomalije u algoritmima za tretiranje signala

Postoji više anomalija u opisanim algoritmima za tretiranje signala. Prva i najznačajnija nastaje kada proces obrađuje signal, a kernel pre povratka u korisnički mod briše polje u u-području koje sadrži adresu za signal catcher funkciju, tako da za nove signale proces mora ponovo pozvati sistemski poziv signal, a to nije zgodno jer može doći do stanja trke (race condition), zato što se novi signal može pojaviti pre nego što proces pozove sistemski poziv signal. Sledeći program ilustruje stanje trke .

```
#include <signal.h>
sigcatcher()
{
  printf("PID %d caught one", getpid()); /* print process id*/
  signal(SIGINT, sigcatcher); /*again*/
}
main()
{
  int ppid;
  signal(SIGINT, sigcatcher)
  if(fork() ==0)
```

```
  { /* give enough time for both process to set up*/
    sleep(5)            /*lib function to delay 5 secs*/
    ppid = getpid();    /* get parent id*/
    for(;;)
    if (kill(ppid, SIGINT) == -1)  exit();
  }
  /* lower priority, greater chance of exhibiting race*/
  nice(10);
  for(;;);
}
```



Slika 7.4. *Primer za signal-cacher funkciju*

Proces zove sistemski poziv signal da bi uhvatio prekidni signal SIGINT i izvršio funkciju sigcatcher(). Proces kreira dete proces pozivom fork, a zatim poziva sistemski poziv nice da bi smanjio svoj prioritet i odlazi u beskonačnu petlju (for(;;)). Proces dete

spava pet sekundi, dajući šansu roditelju da obavi nice, a potom ide u petlju u kojoj roditelju šalje prekidni signal za vreme svake iteracije. Ako se sistemski poziv kill završi sa greškom, na primer zato što roditelj više ne postoji, proces dete obavlja sistemski poziv exit. Ideja je da roditeljski proces treba da pozove signal catcher svaki put kad dobije prekidni signal, koji će štampati poruku na ekranu i sa sistemskim pozivom signal ponovo postaviti signal-catcher, tako da bi roditelj po toj zamisli nastavio da izvršava beskonačnu petlju.

Moguća je sledeća sekvenca događaja:

- proces-dete šalje prekidni signal roditelju
- proces-roditelj hvata signal i poziva signal catcher, ali kernel nasilno oduzima procesor roditelju (preemption) i obavlja prebacivanje konteksta pre nego što roditelj ponovo obavi sistemski poziv signal
- proces dete dobija CPU i šalje novi prekidni signal
- proces roditelj više nema postavljen signal catcher, pa kad dobije CPU obaviće exit

Ovo će se veoma verovatno dogoditi, jer je proces roditelj sistemskim pozivom nice smanjio svoj prioritet i proces dete će se birati mnogo češće (podsetimo da oba izvršavaju beskonačnu petlju). Ukupan efekat je da će se oba procesa završiti, ali se ne može odrediti kada će se to dogoditi.

Signali su u prvim UNIX-ima bili zaduženi da prekinu proces ili da se ignorišu, pa im za stanje trke (race condition) nije bila posvećena pažnja. Međutim, sa mogućnošću hvatanja signala, stanje trke postaje ozbiljan problem, koji se na prvi način može rešiti tako što kenel ne briše signalsko polje po prijemu signala. Ovo rešenje će napraviti novi problem: ako signali koji dolaze budu uhvaćeni, korisnički stek može da naraste mnogo i da probije svoj opseg, jer svaki signal poziva signal catcher funkciju, čiji se parametri čuvaju na korisničkom steku. Alternativno, kernel bi mogao da resetuje vrednost signal-handling funkcije da ignoriše signale tog tipa sve dok korisnik ponovo ne specificira šta da se radi sa tim signalom. Takvo rešenje unosi gubitak informacija, zato što proces tada ne zna koliko je signala primio, što je manje opasno nego da se probije korisnički stek. Na kraju, BSD dozvoljava procesu da blokira ili odblokira prijem signala sa posebnim sistemskim pozivom. Kada proces odblokira signal, kernel šalje sve neobrađene (pending) signale koje je proces već bio blokirao. Kada proces primi signal, kernel automatski blokira prijem novih signala dok se signal catching funkcija ne kompletira, a na sličan način radi sa hardverskim prekidom, blokiraju se novi prekidi dok ne opsluži prethodni prekid.

Druga anomalija predstavlja tretiranje signala koji dolaze dok je proces u sistemskom pozivu za spavanje. Signal će izazvati da proces obavi longjmp iz svog spavanja, da se vrati u korisnički mod i da pozove signal catching funkciju. Kada se vrati iz ove funkcije, procesu izgleda da se vratio iz sistemskog poziva sa greškom koja ukazuje da je sistemski poziv prekinut, što korisnik može proveriti i restartovati sistemski poziv, ali je bolje da kernel automatski restartuje sistemski poziv, kao što je to kod operativnog

sistema free BSD.

Treća anomalija postoji u slučaju da proces ignoriše signal. Ako signal dođe kada je proces uspavan, proces će se probuditi ali neće obaviti longjmp, zato što kernel shvata da proces ignoriše taj signal, a već je probudio proces. Mnogo bolja politika bi bila da proces ostavi signal da spava, ali kernel mora da sačuva adresu signal funkcije u u-području procesa. Međutim u-područje može biti nedostupno ako proces spava. Rešenje je da se adresa signal funkcije čuva u ulazu u procesnoj tabeli, a onda kernel može da odluči da li da budi proces po prijemu signala. Alternativno, proces bi mogao neposredno da se vrati na spavanje preko sistemskog poziva sleep, ako proceni da se nije trebao buditi.

Na kraju, kernel ne tretira "death of child" signale kao sve druge signale. Kada proces prepozna "death of child", on postavlja notifikaciju signala u signal polju ulaza u procesnoj tabeli PT u podrazumevano stanje, i ponaša se da ništa od signala nije primio. Efekat ovog signala je buđenje uspavanog procesa. Ako proces hvata ovaj signal, on poziva signal-catching funkciju kao i za sve druge signale. Operaciju koju kernel radi ako proces ignoriše ovaj signal diskutujemo u opisu za wait sistemski poziv.

Grupe procesa

Mada se procesi na UNIX sistemu identifikuju po jedinstvenom PID, poželjno je da sistem može da grupiše procese tako da se identifikuju po grupi. Na primer proces shell je roditelj svim korisničkim procesima sa tog terminala i oni se mogu grupisati da primaju istovremeno signale. Kernel koristi grupni ID da identifikuje grupu povezanih procesa, koji treba da dobiju zajednički signal. Grupni ID se čuva u tabeli procesa PT.

Postoji poseban sistemski poziv koji služi da inicijalizuje PGID i setuje na istu vrednost kao i PID. Sintaksa za sistemski poziv setgrp je:

```
grp = setgrp();
```

gde je grp novi PGID. Proces dete nasleđuje od svog roditelja PGID preko sistemskog poziva fork.

Slanje signala iz procesa

Procesi koriste sistemski poziv kill da bi slali signale. Sintaksa za sistemski poziv kill je

```
kill(pid, signum)
```

gde je pid skup procesa koji primaju signal, a signum je broj signala koji se šalje. Sledeća lista prikazuje korespodenciju između vrednosti pid i skupa procesa

- ako je pid pozitivan broj, kernel šalje signal procesu sa PID=pid

- ako je pid jednak nuli, kernel šalje signal svim procesima koji su sa procesom koji šalje signal u istoj grupi
- ako je pid jednak jedan, kernel šalje signal svim procesima čiji je realni UID jednak efektivnom UID procesa koji šalje signal. Ako proces koji šalje ima efektivni UID=root, kernel šalje signal svim procesima osim za PID=0 i PID=1
- ako je pid negativan broj koji nije -1, kernel šalje signale svim procesima u grupi čiji je PGID jednak apsolutnoj vrednosti pid.

U svim slučajevima, ako proces koji šalje signal nema efektivnu vrednost UID=root, ako pošalje signal procesima čija realna ili efektivna vrednost ne odgovara primajućim procesima, sistemski poziv kill će otkazati.

Evo jednog primera za setgrp.

```
#include <signal.h>
main()
{
  register int i;
  setgrp();
  for(i=0; i<10; i++)
  {
    if(fork() == 0)
    { /* child proc*/
      if (i & 1) setgrp();
      printf("pid %d pgrp =%d \n", getpid(), getgrp());
      /* print proc id*/
      pause(); /* SC to suspend execution*/
    }
  }
  kill(0, SIGINT);
}
```

U ovom programu proces postavlja svoj sopstveni PGID i kreira desetoro dece, od kojih svako dete ima isti PGID kao roditelj, ali procesi za vreme neparnih iteracija, postavljaju svoje sopstvene PGID. Sistemski poziv getpid i getgrp vraćaju PID i PGID tekućeg procesa, dok sistemski poziv pause suspenduje izvršavanje procesa dok proces ne primi signal. Na kraju, roditelj izvršava kill i šalje SIGINT svi procesima iz svoje grupe, tako da će pet parnih procesa primiti taj signal, a pet neparnih će ostati u uspavanom (paused) stanju.

Sistemski pozivi exit, wait i exec

Uništenje procesa

Procesi na UNIX operativnom sistemu završavaju svoje aktivnosti preko sistemskog poziva exit. Proces koji završava aktivnosti ulazi u zombie stanje, otpuštajući sve svoje resurse. Sintaksa je

```
exit(status)
```

gde je status vrednost koja se vraća procesu roditelju.

Procesi mogu pozvati exit eksplicitno ili implicitno na kraju programa, rutina koja je startovala program, na kraju, po povratku iz glavnog programa pozvaće sistemski poziv exit. Kernel može eksplicitno pozvati sistemski poziv exit ako proces primi signal koji ne hvata i tada će status programa biti broj signala.

Operativni sistem ne odlučuje koliko će proces trajati (procesi swaper i init postoje sve dok je aktivan UNIX, a na drugoj strani neki procesi postoje kratko posle čega izčezavaju kao npr. getty).

Algoritam exit

Objasnimo algoritam za sistemsk i poziv exit:

```
algorithm exit /* */

input:  return code for parrent process
ouput: none

{
  ignore all signals;
  if(process group leader with associated control terminal)
  {
    send hangup signal to all member of process group;
    reset process group for all members to 0;
  }
  close all open files(algorithm close(internal version))
  release current directory (algorithm iput);
  release current (changed) root, if exist (algorithm iput);
  free regions, memory associated with process
    (algorithm freereg);
  write accounting record;
  make process state zombie;
  assign parent PID of all child processes to be init process;
```

```
    if any children were zombie, send death of child
      signal to init;
    send death of child signal to parent process;
    context switch;
}
```

Kernel prvo blokira obradu signala za proces koji završava, zato što za njega signali više neće imati smisla. Ako je proces vođa grupe procesa na pridruženom terminalu, tada će se prekinuti svi procesi u grupi. Na primer, ako korisnik prosledi <ctrl-d> karakter shell procesu, shell je vođa i svi procesi sa njim u grupi će se okončati. Kernel postavlja takođe PGID svih procesa iz te grupe na nulu, da ne bi došlo do haosa ako neki novi proces dobije PID od vođe koji je upravo uradio exit, pa on postaje novi vođa za one koji se nisu završili. Kernel zatim analizira otvorene deskriptore datoteka koje zatvara pomoću internog close algoritma i oslobađa inode strukturu za tekući direktorijum i za promenjeni root, preko iput algoritma.

Zatim, kernel oslobađa svu korisničku memoriju, oslobađajući regione preko algortima detachreg i menja stanje procesa u stanje zombie. Kernel čuva izlazni status za zombie dete (exit status code) i akumulira i upisuje statističke informacije o vremenima koje je dete radilo u korisničkom modu, u kernelskom modu, iskorišćenost procesora (CPU usage), iskorišćenost memorije itd.

Na kraju, kernel izbacuje proces iz stabla aktivnih procesa, a proces init će usvojiti svu procesovu živu decu-procese. Ako je bilo koje dete zombie, proces koji završava poslaće procesu init "death of child" signal tako init izbacuje sve zombie procese iz proces tabele. Proces koji završava aktivnosti, takođe šalje svom roditelju "death of child" signal.

Po tipičnom UNIX scenariju, roditeljski proces izvršava sistemski poziv wait za sinhronizaciju sa decom koja završavaju rad. Proces koji završava (now-zombie) još radi prebacivanje konteksta, tako da kernel može da izabere drugi proces za rad, jer kernel nikada neće izabrati proces u zombie stanju.

Primer za sistemski poziv exit

U sledećem primeru, proces kreira proces dete, koje štampa svoj PID i obavlja sistemski poziv pause, blokirajući sebe dok ne primi neki signal. Roditelj proces štampa dečiji PID i završava vraćajući detetov PID kao izlazni status. Proces-dete će postojati sve dok ne primi signal, bez obzira da li proces-roditelj postoji ili ne.

```
main()
{
  int child;
  if ((child = fork()) = = 0)
  {
    /* child process*/
    printf("child PID %d \n", getpid());
```

```
    pause();
  }
  /*parrent process*/
  printf("child PID %d \n", child);
  exit(child);
}
```

Sistemski poziv wait (čekanje na uništenje)

Proces može sinhronizovati svoje izvršavanje sa završetkom procesa-deteta preko sistemskog poziva wait. Sintaksa za ovaj sistemski poziv je:

```
pid = wait(stat_addr)
```

gde je pid proces ID deteta koje treba da postane zombie, stat_addr je integer adresa u korisničkom prostoru koja sadrži izlazni status deteta.

Sledi prikaz pseudo koda algoritma za sistemski poziv wait.

```
algorithm  wait

input:  address of variable to store status of exiting process
output: child ID, child exit code

{
  if(waiting process has no child processes) return(error);
  for (;;)
  {
    if(waiting process has zombie child)
    {
      pick arbitrary zombie child;
      add child CPU usage to parent;
      free child PT entry;
      return(child ID, child exit code);
    }
    if(process has no child processes)  return error;
    sleep at interruptible priority (event child process exits);
  }
}
```

Kernel traži zombie decu procesa i ako nema nijednog deteta, vraća se greška. Ako se nađe bilo koje zombie dete, ekstrakuju se PID takvog deteta i izlazni status. To su parametri koje je generisao sistemski poziv exit takvog deteta, a to mogu da budu različite vrednosti koje opisuju šta se dogodilo sa detetom.

Kernel uzima akumulisano vreme koje je dete radilo u korisničkom i kernelskom modu i dodaje u u-područje procesa roditelja, a na kraju otpušta ulaz u tabeli procesa PT koji je zauzimalo dete.

Ako proces obavlja sistemski poziv wait i pri tome ima decu procese ali nisu u zombie stanju (nijedno dete), proces će se uspavati sve dok se ne pojavi signal. Kernel nema poseban sistemski poziv za buđenje procesa koji spava u sistemskom pozivu wait, takav proces se jedino budi na prijem signala. Za signal "death of child", proces će reagovati na sledeći način:

- u podrazumevanom slučaju, proces će se probuditi u svom spavanju; u sistemskom pozivu wait i sleep će tada pozvati algoritam issig da proveri prisustvo signala. Ako issig detektuje signal "death of child" tada vraća "false". Kernel ne obavlja longjmp iz sleep algoritma ali se vraća u sistemski poziv wait, gde će pronaći barem jedno zombie dete i vratiti se iz sistemskog poziva wait.
- ako proces hvata signal "death of child", kernel poziva user-handler rutinu kao i za druge signale.
- ako proces ignoriše signal "death of child", kernel ulazi u wait petlju, oslobađa ulaze u tabeli procesa PT za zombie decu i traži odnosno čeka na novu zombie decu..

Primeri za wait

Uzmimo primer kada se sledeći program pozove sa ili bez argumenata.

```
#include <signal.h>
main(argc, argv)
int argc;
char *argv[];
{
  int i, ret_val, ret_code;
  if(argc >= 1) signal(SIGCLD, SIG_IGN);
    /* ignore death of children*/
  for(i=0; i<15; i++)
  {
    if(fork() == 0)
    {
      /* child proc*/
      printf("child proc %x \n", getpid()); /* print proc id*/
      exit(i); /* SC to suspend execution*/
    }
    ret_val = wait (&ret_code);
    printf("wait ret_val %x ret_code %x\n", ret_val, ret_code);
  }
}
```

Uzmimo slučaj da korisnik pozive program bez argumenata, (argc = 1, ime programa). Proces roditelj kreira 15 procesa dece koja eventualno završavaju sa izlaznim kodom a to je vrednost brojača kad je proces kreiran. Kernel izvršava sistemski poziv wait za proces

roditelj, pronalazi zombi-dete proces koje je završilo aktivnost, uzima njegov PID i exit status, ali se ne zna koje će to dete da bude.

Ako se program pozove sa argumentom (argc >1) roditeljski proces signalizira da ignoriše signal "death of child". Pretpostavimo da roditelj spava, a neko njegovo dete obavlja sistemski poziv exit, koji postavlja signal "death of child" koga roditelj ignoriše. Roditelj praktično uklanja ulaz u tabeli procesa PT za to zombie dete, ali nastavlja wait kao da se signal nije dogodio. Svaki put se obavlja ova procedura za svako dete koje je završilo aktivnosti, i onda se dolazi do procesa roditelja koji nema više dece, kada se sistemski poziv wait završava sa exit kodom =-1.

Vidimo da u prvom slučaju roditeljski proces čeka bilo koje dete da završi, a u drugom slučaju roditelj čeka svu decu da završe.

Stare verzije UNIX operativnog sistema implementiraju sistemske pozive exit i wait bez signala "death of child", a umesto tog signala sistemski poziv exit budi roditeljski proces. Ako je roditeljski proces zaspao u sistemskom pozivu wait, sve je u redu, sistemski poziv exit deteta će ga probuditi, ali ako roditelj nije uspavan u sistemskom pozivu wait, buđenje nema efekat, ali će roditelj sledeći put kad uđe u sistemski poziv wait naći zombie decu.

Problem sa ovakvim implementacijama je u tome što je nemoguće očistiti zombie procese osim ako njihov roditelj ne obavi sistemski poziv wait. Ako proces kreira više dece, ali nikada ne izvrši wait, tabela procesa PT će biti puna zombie dece.

Posmatrajmo dispečer program:

```
#include <signal.h>
main(argc, argv)
{
  char buf[256];
  int i, ret_val, ret_code;
  if(argc >= 1)    signal(SIGCLD, SIG_IGN);
  /* ignore death of children*/
  while(read(0, buf, 256))
  {
    if(fork() == 0)
    {
      /* child proc here typically does something with buffer */
      printf("child proc %x \n", getpid()); /* print proc id*/
      exit(0); /* SC to suspend execution*/
    }
  }
}
```

Proces čita svoj standarni ulaz dok se ne dogodi EOF (end of file), kreirajući proces dete za svako čitanje (read) sa terminala tj. standardnog ulaza. Međutim u ovom programu roditelj proces nikad ne obavlja wait, zato što želi da najbrže moguće dispečuje

procese decu. Ako proces roditelj preko sistemskog poziva signal ignoriše signal "death of child", kernel će otpuštati sve zombie procese automatski, preko procesa init. U protivnom, zombie procesi će neprekidno boraviti u tabeli procesa PT.

Sistemski poziv exec (pozivanje drugih programa)

Sistemski poziv exec poziva drugi program, prepisujući memorijski prostor procesa kopijom izvršne datoteke. Sadržina korisničkog (user-level) konteksta koji je postojao pre sistemskog poziva exec nije više dostupna, osim parametara za exec sistemski poziv, koje kernel kopira iz starog u novi adresni prostor. Sintaksa za sistemski poziv exec je:

```
execve(filename, argv, envp)
```

gde je filename ime izvršne datoteke koja se poziva, argv je pokazivač na polje pokazivača koji su parametri za izvršnu datoteku, a envp je pokazivač na okruženje (enviroment) izvršnog programa. Postoji više bibliotečkih funkcija koje pozivaju sistemski poziv exec kao što su execl, execv, execle itd. Kada program koristi komandne linijske parametre kao

```
main(argc, argv)
```

polje argv je kopija argv parametra sistemskog poziva exec. Što se okoline tiče, ona se sastoji od karakter nizova tipa "name=value" i pruža korisne informacije za program kao što su korisničke promenljive HOME ili PATH. Procesi mogu pristupati njihovoj okolini preko globalne promenljive environ, koju inicijalizuje C startrup rutina.

Algoritam za sistemski poziv exec

Sledi opis algoritma za sistemski poziv exec.

```
algorithm  exec

input: (1) file name
       (2) parameter list
       (3) enviroment variables list
output: none

{
  get file inode (algorithm namei)
  verify file executable, user has permission to execute;
  read file headers, check that it is a load module;
  copy exec parameters from old address space to system space;
  for (every region attached to process)
    detach all old region (allgorithm detach);
  for (every region specified in load module)
  {
    allocate new regions (algorithm allocreg);
```

```
      attach the region (algorithm attachreg);
      load region into memory if appropriate (algorithm loadreg);
    }
    copy exec parameters into new user stack region;
    special processing for setuid programs, tracing;
    initialize user register save area for return to user mode;
    release inode of file (algorithm iput);
}
```

Sistemski poziv exec prvo pristupa datoteci, tako što joj pronalazi inode strukturu preko algoritma namei i proverava da li je to obična datoteka sa x pravima i da li korisnik koji je vlasnik procesa ima pravo da izvršava program. Kernel zatim čita zaglavlje (header) datoteke da bi odredio logički sadržaj (layout) izvršne datoteke.

Format izvršne datoteke

Na slici 7.5 je prikazan logički format izvršne datoteke koju generiše asembler ili loader i koja se sastoji od četirii dela:

- Primarno zaglavlje (header) opisuje koliko ima sekcija u datoteci, početnu virtuelnu adresu za izvršenje procesa i magični broj koji daje tip izvršne datoteke
- Zaglavlja sekcija opisuju svaku sekciju u datoteci, opisujući veličinu sekcije (section size), virtuelnu adresu sekcije koju sekcija treba da zaizme u vreme izvršavanja i druge informacije
- Sekcije koje sadrže podatke kao što je text ili kôd programa koje se inicijalno pune u adresni prostor procesa
- Razne sekcije sa tipičnim osobinama sekcija raznih podataka: simboličke tabele i druge vrste podataka

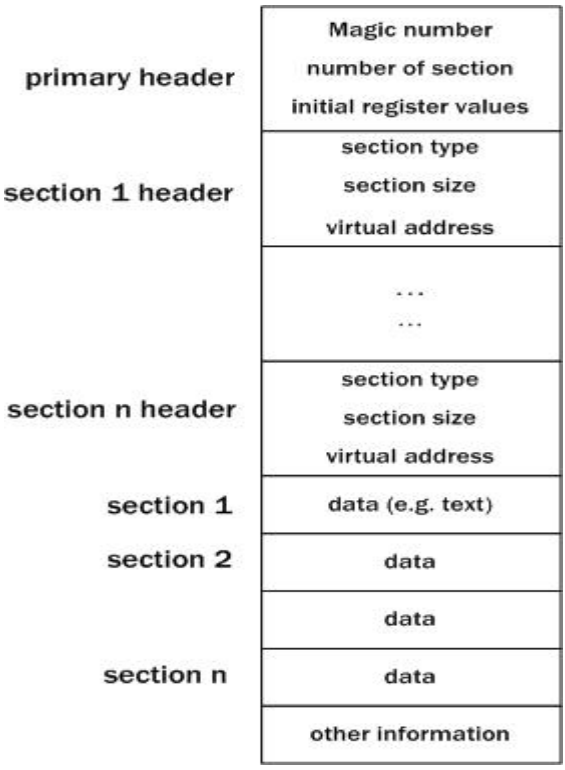
Mnogi formati su se menjali, ali sve izvršne datoteke sadrže primarno zaglavlje sa magičnim brojevima, pri čemu magični brojevi mogu da definišu neke osobine izvršne datoteke vezane za procesor, a magični brojevi su veoma važni u sistemu straničenja.

Kernel prvo pristupa inode strukturi izvršne datoteke i proverava da li može da je izvršava. Iza toga će se prepisati korisnički kontekst procesa, pa prvo što treba uraditi je kopiranje parametara novog programa koji su u starom kontekstu. Kernel ih prvo kopira u privremeni bafer sve dok ne priključi (attach) regione za korisnički kontekst.

Parametri za sistemski poziv exec su adrese za parametre, pa kernel prvo kopira adrese parametara a potom i same parametre. Kernel može izabrati više lokacija za privremeni smeštaj parametara, kao što je kernelski stek, nealocirane stranice, swap prostor.

Naprostiji način kopiranja parametara je novi korisnički kontekst u kernelskom steku, ali parametri mogu biti dugački, a kernelski stek ograničen. Zato se prelazi na druge

metode, kao što su alociranje stranica u memoriji, a swap prostor se izbegava jer je spor.



Slika 7.5. *Format izvršne datoteke*

Posle kopiranja exec parametara, kernel izbacuje (detach) stare regione procesa preko detachreg algoritma. Text regioni imaju poseban tretman. U ovom trenutku proces nema korisnički kontekst, tako da ako mu se dogodi greška on se prekida. Kernel alocira i priključuje (attach) regione za text sekcije i sekcije podataka, a onda se oni pune iz sekcija izvršne datoteke (algoritmi allocreg, attachreg, loadreg). Region podataka procesa deli se na dva dela, podaci koji se inicijalizuju u vreme prevođenja i podaci koji se ne inicijalizju u vreme prevođenja (bss). Prvo se alociraju i priključuju (attach) inicijalizovani podaci, a kernel potom obavlja uvećanje regiona podataka sa growreg algoritmom za bss sekciju, koga inicijalizuje nulama. Na kraju se alocira stek region, priključuje se procesu i alocira se memorija za exec parametre koji se kopiraju u korisnički stek region.

Kernel briše adrese korisničkih signal catcher funkcija iz u-područja, zato što ona više nemaju značenje za novi korisnički kontekst. Potom se setuju registri za novi korisnički kontekst od kojih su najznačajniji PC i SP. PC se puni iz zaglavlja datoteke. Kernel uzima specijalnu akciju za setuid programe i za proces tracing, a to ćemo obraditi

kasnije. Na kraju se otpušta inode struktura izvršne datoteke, preko algoritma iput. Praktično, sistemski poziv exec u odnosu na datoteku, obavlja skoro sve što i sistemski poziv open, sem što nema ulaza u tabeli datoteka FT. Kada se vrati iz sistemskog poziva exec, proces počinje da izvršava kôd novog programa. Mada je proces potpuno promenjen po kodu, njegov PID i mesto u proces stablu ostaje isto, jedino mu se promenio korisnički (user-level) kontekst.

Primer za sistemski poziv exec

Na primer, sledeći program kreira proces dete koje obavlja sistemski poziv exec.

```
main()
{
  int status;
  if(fork() == 0) execl("/bin/date", "date", 0);
  wait(&status);
}
```

Kada se obavi sistemski poziv fork, roditelj i dete izvršavaju isti kôd. Pre nego što dete obavi sistemski paziv exec, njegov text se sastoji od gornjeg programa, region podataka sadrži nizove "/bin/date", i "date", a stek sadrži stek okvire za realizaciju sistemskog poziva exec. Kernel pronalazi /bin/date i uverava se da je to izvršna datoteka koju korisnik može da obavi. Kernel kopira parametre sistemskog poziva exec ("/bin/date", "date") negde privremeno, a potom oslobađa sve regione koji su pripadali procesu, a potom alocira nove, u novi text region kopira text sekciju, a sekcije podataka kopira u region podataka. U alocirani stek region kopira ulazne parametre. Posle sistemskog poziva exec, proces dete ne izvšava više gornji program, dete se više nikada neće vratiti na wait, njegov kôd je kôd programa /bin/date. Kada dete obavi svoje, roditelj dobija exit status i završava wait.

Algoritam xalloc

Text i data sekcije su po pravilu razdvojeni u izvršnim datoteka što im donosi dve velike prednosti: zaštitu i deljenje. Na primer ako su razdvojeni, text region je nepromenljiv i može se deliti između više procesa. Kako je UNIX uveo strogo razdvajanje text i data sekcija, uveden je novi algoritam xalloc za dodelu text regiona, pri čemu kernel mora da proveri u magičnim brojevima da li je text sekcija deljiva i ako jeste, da li je već u memoriji od strane nekog drugog procesa.

```
algorithm xalloc /* allocate and initialize text region*/

input: inode of executable file
output: none

{
  if(executable file does not have separate text region) return;
  if(text region associated with text of inode)
```

```
{
  /* text region exist .....attach to it*/
  lock region;
  while(contents of region not ready yet)
  {
    /* manipulation of refernce count prevents total
       removal of region */
    increment region RC;
    unlock region;
    sleep(event contents of region ready)
    lock region;
    decrement region RC;
  }
  attach region to process (algorithm attachreg);
  unlock region;
  return;
}
/* no such text region---create one*/
allocate text region (algorithm allocreg); /* region locked*/
if(inode mode has stickey bit) turn on region stickey flag;
attach region to virtual address indicated by inode file header
  (algorithm attachreg);
if(file specially formated for paging system) /* lesson 9*/
else /*not formated for paging system*/
read file text into region (algorithm loadreg);
change region protection in per process region table
  to read-only;
unlock region;
}
```

Kernel u xalloc algoritmu pretražuje aktivnu region listu za text region izvršne datoteke, tražeći region sa odgovarajućom inode strukturom, istom kao za traženu datoteku. Ako takav region ne postoji, kernel mora da alocira novi region (allocreg), zatim da ga priključi procesu (attachreg), onda sledi punjenje u memoriju (loadreg) i menja mu se zastavicu na read-only režim.

Ako je region pronađen u listi aktivnih regiona postoje dve situacije: prva situacija se odnosi kada je region već u memoriji, a druga situacija je da je počelo punjenje tekst regiona, a nije završeno, pa proces ide na spavanje, dok se region ne napuni. Kernel otključava region kada završi algoritam xalloc, i dekrementira broj referenci kasnije, tek kada obavi izbacivanje (detach) za vreme sistemskog poziva exit.

Podsetimo da prilikom alokacije regiona, inkrementira se broj referenci za region i broj referenci za inode strukturu, tako da je broj referenci jednak jedinici, tako da niko ne može da obriše datoteku, pa čak i da se dogodi sistemski poziv unlink, datoteka će ostati tu.

Prepostavimo da /bin/date ima razdvojene text i data sekcije. Prvi put kada proces izvrši /bin/date, kernel napravi ulaz u region tabeli RT za text region i postavlja inode broj referenci za incore inode strukturu na jedan, kada se exec kompletira. Kada /bin/date završi, algoritmi detachreg i freereg dekrementiraju broj referenci za inode na nulu. Ako se broj referenci za incore inode strukturu ne bi inkrementirao kada se prvi put izvršava sistemski poziv exec, exec nekog drugog programa bi mogao da namesti drugi program u in-core inode tabeli, pa bi exec mogao da izvrši pogrešan program. Zato je broj referenci inode za shared text minumum jedan.

Mogućnost da se dele text regioni može da se ubrza preko stickey bita, koji se setuje preko chmod komande. Stickey bit izaziva da kernel ne otpušta memoriju lociranu za stickey-text region, čak i ako broj referenci za region padne na nulu. Kernel će ostaviti stickey-text region u memoriji sa inode strukturom koja ima broj referenci jednak jedan, čak i kad nema više aktivnih procesa za njega. Zato će svaki novi sistemski poziv exec za taj stickey-text imati malo vreme postavljanja (startup time), jer je već u memoriji, pa čak i da je swapovan taj text region, brže će se napuniti iz swap prostora nego iz datoteke.

Kernel će ukloniti ulaze za stickey-bit text regione u sledećim slučajevima:

- ako proces otvori datoteku za write, čime se menja text region
- ako proces promeni prava pristupa (permision mode) tj. skine stickey bit
- ako proces obriše datoteku (unlink), a nema ni jednog procesa koji radi sa datotekom
- ako proces deaktivira (umount) sistem datoteka
- ako nestane mesta na swap prostoru

U prva dva slučaja menja se sadržaj datoteke, odnosno ukida se stickey bit. U ostalim slučajevima, ne menja se datoteka, ali nema ni jednog procesa koji je koristi, a više i nema uslova da je neko koristi.

Prvi operativni sistemi UNIX imaju razdvojene sistemske pozive fork i exec zbog male memorije, ali razdvajanje sistemskih poziva fork i exec je kasnije postalo jako zgodno zato što procesi mogu nezavisno da manipulišu sa deskriptorima datoteka i sa pipe datotekama mnogo bolje nego kada bi fork i exec bili spojeni u jedan sistemski poziv.

UID procesa

Kernel pridružuje procesu dva korisnička identifikatora (user IDs), nezavisno od procesovog PID, a to su realni UID i efektivni UID ili setuid (set user ID). Realni korisnički ID-ovi identifikuju korisnika koji je odgovoran za proces koji se izvršava. Efektivni korisnički ID se koristi pri dodeli vlasništva novokreirane datoteke, pri proveri prava pristupa za datoteku i pri proveri prava pristupa prilikom slanja signala procesima preko sistemskog poziva kill. Kernel će dozvoliti procesu da promeni svoj efektivni korisnički ID kada izvršava setuid program ili kad proces eksplicitno pozove sistemski poziv setuid.

Setuid sintaksa

Setuid program je izvršna datoteka sa setovanim setuid bitom u svom polju za prava pristupa (permission). Kada proces izvršava setuid program, kernel setuje polje za efektivni UID u procesovoj tabeli PT, kao i u u-području, ali ne ID od korisnika koji je pokrenuo proces već od vlasnika setuid programa. Da bi smo razlikovali realno i efektivno UID polje u PT, objasnimo sistemski poziv setuid.

Sintaksa za sistemski poziv setuid je

```
setuid(uid)
```

gde je uid novi korisnički ID (UID), a rezultat zavisi od tekuće vrednosti efektivnog UID-a. Ako je efektivni UID procesa koji poziva sistemski poziv setuid jednak ID od superusera, kernel tada postavlja realno i efektivno UID polje na dva mesta, u procesnoj tabeli PT i u u-području i to na vrednost uid. Međutim, ako efektivni UID procesa koji poziva setuid nije jednak superuseru, kernel tada postavlja efektivni UID polje samo u u-području na vrednost uid. Naravno uid mora da bude realni korisnik, u protivnom, sistemski poziv javlja grešku. Generalno, proces nasleđuje svoj realni i efektivni korisnički ID od roditelja za vreme sistemskog poziva fork i održava njihove vrednosti u toku sistemskog poziva exec.

Primer za setuid

Sledeći program demo demonstrira sistemski poziv setuid.

```
-rwsr-xr-x   maury   group   demo    uid(maury)=8319
-r--------   maury   group   maury   uid(mjb)=5088
-r--------   mjb     group   mjb
```

```
include <fcntl.h>
main()
{
  int uid, euid, fdmjb, fdmaury;
  uid = getuid();
  euid = geteuid();
  printf("uid %d euid %d\n", uid, euid);
  fdmjb = open("mjb", O_RDONLY);
  dmaury = open("maury", O_RDONLY);
  printf("fdmjb %d fdmaury %d\n", fdmjb, fdmaury);
  setuid(uid)
  printf("after setuid (%d): uid %d euid %d\n",
          uid, getuid(), setuid());
  fdmjb = open("mjb", O_RDONLY);
  fdmaury = open("maury", O_RDONLY);
  printf("fdmjb %d fdmaury %d\n", fdmjb, fdmaury);
  setuid(euid);
```

```
  printf("after setuid (%d): uid %d euid %d\n",
          uid, getuid(), setuid());
}
```

Pretpostavimo da je izvršna datoteka kreirana prevođenjem programa čiji je vlasnik "maury" (UID=8139), setuid bit mu je postavljen a svi korisnici imaju pravo da ga izvršavaju. Dalje, pretpostavimo da korisnici "mjb" sa UID=5088 i korisnik maury imaju po jednu datoteku koja se zove isto kao i korisnik i one su read-only samo za svoje vlasnike.

Kada korisnik mjb, izvršava demo program (program je vlasništvo korisnika maury i ima setuid bit), rezultat će biti sledeći:

```
uid 5088 euid 8319
fdmjb -1 fdmaury 3
after setuid(5088): uid 5088 euid 5088
fdmjb 4 fdmaury -1
after setuid(5088): uid 5088 euid 8139
```

Sistemski poziv getuid i geteuid vraćaju realni i efektivni korisnički ID procesa i za korisnika mjb to su uid 5088 euid 8319. Proces neće moći da otvori svoju datoteku mjb, jer je njegov euid 8139 pa nema read pravo, ali može da otvori tuđu datoteku maury. Posle poziva sistemskog poziva setuid, efektivni UID se vraća na 5088 i to prikazuje printf naredba (printf("after setuid (%d): uid %d euid %d\n"....).

Sada proces sa novim efektivnim UID može da otvori svoju datoteku, ali ne može datoteku maury. Onda se efektivni UID ponovo vraća na 8319 sa setuid(euid). U ovom primeru smo videli da proces može izvršavati setuid program i menjati efektivni UID između njegovog realnog UID-a i vlasnika setuid datoteke.

Kada korisnik maury izvrši ovaj program rezultat će biti:

```
uid 8319 euid 8319
fdmjb -1 fdmaury 3
after setuid(8139): uid 8319 euid 8319
fdmjb -1 fdmaury 4
after setuid(8319): uid 8319 euid 8139
```

Primećujemo da su realni i efektivni UID za korisnika maury uvek 8319 i da njegov proces ne može nikada otvoriti mjb datoteku. Efektivni uid se upisuje u u-područje na svaki sistemski poziv setuid ili kada se izvršava setuid program. Sačuvani UID u tabeli procesa PT dozvoljava procesu da postavi svoj efektivni korisnički UID preko sistemskog poziva setuid.

Login program je tipičan program koji obavlja sistemski poziv setuid. Login program ima setuid na korisnika root, pa je njegov efektivni UID jednak root, a program traži od korisnika da unese svoje ime i lozinku a onda postavlja realni i efektivni korisnički ID za tog korisnika, i na kraju login obavlja sistemski poziv exec za program shell koji se izvršava sa realnim i efektivnim UID od tog korisnika.

Takođe mkdir komanda je tipičan setuid program, jer samo proces sa efektivnim UID jednako root, može kreirati direktorijum. Da bi to mogli da rade obični korisnici, mkdir je vlasništo privilegovanog korisnika root i ima setuid bit, tako da kada se izvršava program mkdir, proces nastupa sa efektivnim UID jednako root.

Promena veličine procesa

Proces može povećati ili smanjiti veličinu svog regiona podataka, korišćenjem sistemskog poziva brk, čija je sintaksa:

```
brk (endds)
```

gde je endds vrednost najviše virtuelne adrese regiona podataka procesa (break value).

Postoji još jedna alternativa

```
oldends=sbrk (increment)
```

gde increment vrednost povećava tekuću break vrednost za zadati broj bajtova, a oldendds je break vrednost prethodnog sistemskog poziva brk. Sistemski poziv sbrk je rutina iz C biblioteke koja poziva brk. Ako se region uvećava, novi virtuelni prostor je takođe kontinulan sa starim prostorom, odnosno stari adresni prostor se proširuje. Kernel mora da proveri da li je zahtev legalan, da li to može da se realizuje i da se ne preklapa sa drugim regionima. Ako sve provere prođu, kernel poziva growreg algoritam da alocira pomoćnu (aux) memoriju kao što su tabele stranica za novi region i uvećava polje za veličinu procesa u tabeli procesa PT. Na swapping sistemima, takođe se pokušava da se alocira novi prostor u memoriji i nova memorija se puni nulama. Ako nema dovoljno mesta u memoriji neki od procesa mora da se prebaci na swap prostor (swap out).

Algoritam brk

Ako se sistemski poziv brk koristi za smanjenje regiona, kernel će osloboditi memoriju i ažurirati tabelu procesa PT i tabele stranica.

```
algorithm  brk

input: new break value
output: old break value

{
  lock process data region;
  if(region size increasing)
  if(new region size is illegal)
  {
    unlock data region;
    return(error);
```

```
  }
  change region size (algorithm growreg);
  zero out in new space;
  unlock process data region;
}
```

Primer za sistemski poziv brk

Slika prikazuje program koji koristi sistemski poziv brk.

```
#include <signal.h>
char *cp; int callno;
main()
{
  char *sbrk();
  extern catcher();
  signal(SIGSEGV, catcher);
  cp = sbrk((0));
  print("original brk value %u",cp)
  for(;;)
  *cp++=1
}
catcher(signo)
int signo;
{
  callno++;
  printf (caught sig %d %dth call at addr %u", signo, callno, cp)
  sbrk(256)
  signal(SIGSEGV, catcher);
}
```

Pošto podesi signal-catcher funkciju za SIGSEGV (segmentation violation) signale, proces poziva sbrk, pa prikazuje početnu break vrednost. Tada se obavlja petlja, inkrementira se karakter pokzaivač cp i upisuje se vrednost u njega, sve dok ne udari u kraj data regiona, kada se generiše signal SIGSEGV, a njegova signal-catcher funkcija pozove sistemski poziv sbrk(256), koji povećava data region za 256 bajtova. Petlja se nastavlja dalje.

Sledeći prikaz dešava se na sistemu sa straničenjem (paging) 3B20, na VAX procesoru koji je prilagođen tehnici straničenja:

```
original brk value 140924
caught sig 11 1th call at addr 141312
caught sig 11 2th call at addr 141312
caught sig 11 3th call at addr 143360
……(same address printed out to 10h call)
caught sig 11 10th call at addr 143360
```

```
caught sig 11 11th call at addr 145408
……(same address printed out to 18h call)
caught sig 11 18th call at addr 145408
caught sig 11 19th call at addr 145408
```

Kernel automatski proširuje korisnički stek region kada se dogodi prekoračenje (overflow), koristeći sličan algoritam kao brk. Proces originalno sadrži dovoljan korisnički stek da čuva exec parametre, ali u toku izvršenja može doći do prekoračenja, što izaziva grešku u memoriji (memory fault), što generiše prekidni signal, u kome kernel detektuje da se dogodilo prekoračenje na steku (stack overflow) . Tada se poredi zahtevana veličina steka i tekuća veličina stek regiona pa se povećava preko sistemskog poziva brk.

Komandni intrepreter – shell

Program shell je kompleksniji nego što će ovde biti opisano. Pažnja će biti posvećena glavnoj shell petlji u kojoj je demonstrirano asinhrono izvršavanje, redirekcija izlaza i pipeline tehnika za komande. Glavna shell petlja je prikazana na donjem algoritmui.

Shell čita komandnu liniju sa standardnog ulaza koju zatim interpretira po fiksnim pravilima. Standardni ulazni i izlazni deskriptor datoteke za shell obično je terminal na kome se korisnik loguje. Ako shell prepozna ulazni string kao neku unutrašnju komandu (cd, for, while..), shell izvršava komandu interno bez kreiranja novog procesa, u protivnom ulazni niz je ime izvršne datoteke.

Najprostije komandne linije sadrže ime programa i neke parametre kao na primer:

```
who
grep –n include *.c
ls –l
```

Shell obavlja sistemi poziv fork i kreira proces dete koje će izvršiti program zadat u komandnoj liniji. Roditeljski proces, shell, čeka da dete završi, a onda se ponovo vrti u petlji čekajući novu komandu.

Da bi se proces startovao asinhrono (in background) potrebno je upotrebiti & kao u primeru

```
$ nroff –mm bigdocument &
```

pri čemu shell setuje svoju internu promenljivu amper koja se postavlja nakon analize komandne linije. Ako nađe &, shell ne izvršava sistemski poziv wait, nego se opet vraća na petlju za sledeću komandu.

Proces dete u shell-u ima kopiju komandne linije posle sistemskog poziva fork. Redirekcija standarne izlazne datoteke obavlja se kao u primeru

```
$ nroff –mm bigdocument > output
```

kada proces dete kreira izlaznu datoteku specificiranu u komandnoj liniji, pri čemu ako sistemski poziv creat otkaže (zbog nedostatka prava pristupa), proces dete će realizovati exit neposredno. Ali ako sistemski poziv creat uspe, proces-dete zatvara svoj standardni izlaz, duplira file-deskriptor fd za novu output datoteku, čime standarni izlaz postaje nova datoteka. Proces dete zatvara fd nove datoteke jer više nije potreban.

Redirekcija standardnog ulaza i standardne greške je slična.

Algoritam za shell

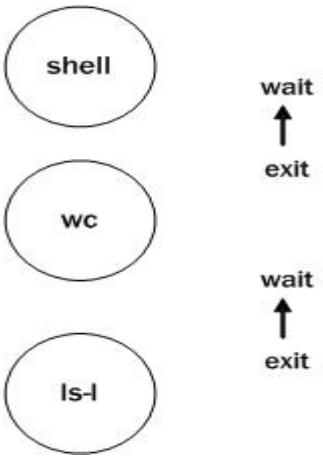
```
/*read commnad line until "end of file*/
while(read(stdin, buffer, numchars))
{
  /* parse command line/*
  if(/* command line contain & /* )
    amper=1;
  else amper=0;
  /* for commands not part of shell command language*/
  if(fork()==0)
  {
    /* redirection of I/O? /*
    if(/* redirection output */)
    {
      fd=creat(newfile, mask);
      close(stout);
      dup(fd);
      close(fd);
      /* stdout is now redirected*/
    }
    if(/* piping */)
    {
      pipe(fildes);
      if(fork()==0)
      {
        /* first component od command line*/
        close(stout);
        dup(fildes[1]);
        close(fildes[1]);
        close(fildes[0]);
        /*stdout now goes to pipe*/
        /*child process does, command*/
        execlp(command1, command1,0)
       }
      /*2nd command line component of command line*/
      close(stdin);
      dup(fildes[0]);
```

```
      close(fildes[0]);
      close(fildes[1]);
      /* standard input now comes to pipe*/
    }/* end of pipe*/
    execve(command2, command2, 0);
  }/* end of child*/
  /*parent continue here…., wait for child to exit if required */
  if(amper == 0) retid = wait(&status);
}
```

Posmatrajmo kôd za pipeline situaciju u kojoj imamo jednostruki komandni pipeline:

```
$ ls –l | wc –l
```

Pošto proces shell kao roditelj obavi fork i kreira dete koje upravlja drugim delom komande linije (wc u ovom slučaju), to dete kreira pipe datoteku, a zatim dete obavlja fork i stvara svoje dete proces (unuče u odnosu na shell) koje će upravljati prvim delom komandne linije. Unuče, koje je nastalo kao drugi fork izvršavaće prvu komponentu komande linije, a to je ls. Unuče upisuje u pipe datoteku, zatvara svoj standarni izlaz, duplira pipe write deskriptor i zatvara pipe-write deskriptor, a zatim se izvršava prvi deo komandne linije, ls -l. Roditelj unučeta je proces koji će obaviti wc, a to je je dete shell procesa, kao na slici. Ovaj proces zatvara svoj standardni ulaz, duplira pipe-read deskriptor, a zatim zatvara pipe-read descriptor i izvršava drugi deo komandne linije. Oba procesa se izvršavaju asinhrono, ali se sinhronišu preko pipe datoteke, tako što je izlaz jedne komande ulaz za drugu. Glavni roditelj čeka svoje dete da završi (wc), a to dete čeka svoje dete ls –l i cela linija se završava kada wc obavi exit, a prvo se završava ls, kao na slici 7.6.



Slika 7.6. *Relacija između procesa u pipeline tehnici*

Boot i init proces

Da bi se sistem inicijalizovao, administrator obavlja bootstrap sekvencu, odnosno podiže sistem: podizanje zavisi od mašina, ali je poenta u učitavanju operativnog sistema u memoriju i njegovo preuzimanje kontrole. Za UNIX sistem, obično se pročita boot-blok (block 0) u memoriju i preda mu se kontrola, a zadatak mu je da locira UNIX kernel i da ga pročita u memoriju. Kada je kernel u memoriji, boot-blok odskači na početnu adresu kernela i kernel startuje svoje izvršavanje (algoritam start)

Algoritam start

```
algorithm start /* system start procedure*/

input: none
output: none

{
  initialize all kernel data structure;
  pseudo-mount of root;
  hand-craft enviroment of process 0;
  fork process 1;
  {
    /* process 1 here*/
    alocate region;
    attach region to init address space;
    grow region to accommodate code about to copy in;
    copy code from kernel space to init user space to exec init;
    change mode: return from kernel to user mode;
    /*
      init never gets here, as result of above change mode;
      init exec's /etc/initttab and becomes a normal user
      process with respect to invocation od system calls
    */
  }
  /* proc 0 continues here */
  fork kernel process;
  /*
    proces 0 invokes the swapper to manage the allocation
    of process address space to main memory  and swap space;
    this is an infinite loop; proces 0 usually sleeps in the
    loop unless there is work for it to do*/
  /*
  execute code for swapper algorithm;
}
```

Kernel inicijalizuje svoje interne strukture podataka. Na primer, kernel konstruiše povezane liste slobodnih bafera i inodova, konstruiše hash-queue liste za bafere i inode strukture, inicijalizuje region strukture, tabele stranica. Kada kompletira inicijalizaciju, kernel mountuje (aktivira) root FS (sistem datoteka) na root direktorijum. Zatim se postavlja okolina za proces nula, postavlja mu se u-područje, inicijalizuje se prvi ulaz u tabeli procesa PT, i postavlja tekući direktorijum za proces nula.

Kada mu postavi okolinu, kernel startuje proces 0. Proces 0, obavlja fork i kreira svoje dete, pozivajući fork direktno iz kernela jer se proces 0 izvršava u kernelskom modu. Novi proces, proces 1, radeći u kernel modu, kreira svoj korisnički kontekst, alocirajući region podataka i priključuje ga u svoj adresni prostor. U region podataka se kopira kernelski kôd, a on po potrebi raste. Taj kopirani kôd formira korisnički kontekts procesa 1. Proces 1 postavlja svoj sačuvani registarski kontekst i formalno se vraća u korisnički mod, izvršavajući kôd koji je upravo kopirao iz kernela. Proces 1 je korisnički proces, za razliku od procesa 0, koji je kernelski proces i izvršava se uvek u kernelskom modu. Text za proces 1, sastoji se od koda za sistemski poziv exec za datoteku /etc/init, a proces 1 to obavlja na normalan način kao i svi drugi procesi. Proces 1 se naziva i init, jer je on odgovoran za inicijalizaciju svih ostalih procesa.

Proces init

Init proces je proces dispečer, izvršavajući procese koji omogućavaju korisnicima da se uloguju (prijave) na sistem.

```
algorithm init /* init proces, proces 1 of the system*/

input: none
output: none

{
  fd = open ("/etc/inittab", O_RDONLY);
  while(line_read(fd, buffer))
  { /*read every line of file*/
    if(invoked state != buffer state) continue;
      /* loop back to while*/
    /*state mached*/
    if(fork() == 0)
    {
      execl("process specified in buffer")
      exit();
    }
    /*init process does not wait */
    /* loop back to while*/
  }
  while ((id = wait((int *) 0)) != -1)
  {
    /* check here if a spawned child died*/
```

```
      /*consider respawning */
      /*otherwise, just continue*/
    }
  }
```

Proces init čita datoteku /etc/inittab u cilju dobijanja instrukcija za izbor procesa koje treba da izvršava. Ova datoteka sadrži linije koje imaju id, identifikator stanja sistema (single user, multiuser), akciju i programsku specifikaciju, kao u primeru:

format: identifier, state, action, process specification

polja su razdvojena znakom :, a komentar počinje sa #

```
co:.respawn:/etc/getty console console #console in machine room
46:2:respawn:/etc/getty -t 60 tty46 4800H #comment here
```

Proces init čita datoteku i ako stanje u kome je init pozvan odgovara identifikatoru stanja u liniji, kreira se proces na bazi programa datog u liniji. Na primer, kada se pozove init u višekorisničkom (multiuser) stanju, init tipično izvršava getty proces, koji upravlja terminalskim linijama. Uspešna login procedura se odvija preko tri procesa, prvo getty proces, pa login proces i na kraju shell proces. U međuvremenu, init izvršava sistemski poziv wait, upravljajući završecima svojih procesa i procesa koji više nemaju roditelje (orphans).

Procesi na UNIX sistemu se dele u tri vrste:

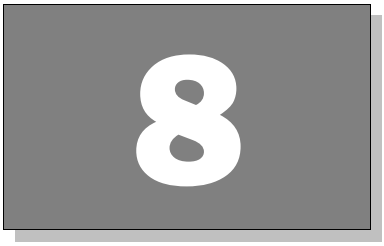
1. korisnički procesi

2. deamon procesi

3. kernel procesi

Većina procesa su korisnički procesi sa pridruženim terminalom. Deamon procesi nisu pridruženi ni jednom korisniku, već obavljaju sistemske funkcije, kao što je administracija i kontrola mreža, vremenski kontrolisane aktivnosti, funkcije za štampu (printer spooling). Demon procese kreira init, a mogu ih pokrenuti i korisnici, liče na korisničke programe koji rade u korisničkom modu a obavljaju sistemske pozive za pristup sistemskim servisima.

Kernel procesi izvršavaju se isključivo u kernelskom modu. Proces 0 kreira kernelske procese, kao što su proces vhand (page-reclaiming process vhand), a zatim postaje swapper proces. Kernelski procesi su slični demon procesima u tome što obezbeđuju sistemske servise, ali imaju veću kontrolu i prava nad kernelskim strukturama i prioritet pošto su deo kernela, pa mogu pristupati direktno kernelskim strukturama podataka bez sistemskih poziva, pa su veoma brzi. Na drugoj strani demoni su mnogo fleksibilniji, jer se lako menjaju, dok za promenu kernelskih procesa mora da se obavi prevođenje kernela.



Raspoređivanje procesa i vremenske funkcije

8.1. UNIX raspoređivanje procesa

U sistemima sa deljenim vremenom (time sharing systems), kernel alocira CPU procesu za period vremena koji se zove vremenski interval ili vremenski kvantum (time slice ili time quantum), a nakon tog intervala procesu se nasilno oduzima procesor (preempted) i raspoređuje se drugi proces. UNIX koristi relativno vreme izvršavanja kao parametar za određivanje koji se proces raspoređuje sledeći. Svaki proces ima prioritet za raspoređivanje (scheduling priority). Kada kernel obavlja prebacivanje konteksta, tada se se bira proces sa najvećim prioritetom. Kernel ponovo izračunava prioritet kada se proces vrati iz kernelskog u korisnički mod i periodično podešava prioritet za procese u stanju 3 (ready-to-run), tj. koji su spremni za rad.

Neki procesi imaju potrebu da znaju količinu vremena. Na primer, komanda time meri vreme izvršavanja UNIX komandi, a komanda date prikazuje datum i vreme. Različiti sistemski pozivi omogućavaju procesima da setuju ili dobijaju kernelske vremenske podatke.

Sistem održava vreme preko sistemskog časovnika, koji šalje prekidni signal u regularnim trenucima i ti trenuci se nazivaju otkucaji sata (time ticks).

Obradićemo raspoređivanje procesa (process scheduling) i sistemske pozive vezane za vremenske funkcije.

UNIX je poznat kao operativni sistem koji ima RoundRobin raspoređivanje sa više nivoa i sa povratnom spregom (RoundRobin with multilevel feedback), što znači da kernel alocira CPU procesima za jedan vremenski kvantum. Pri isteku kvantuma, procesu se nasilno oduzima procesor i vraća se u neki od prioritetnih redova čekanja (queue). Proces bi mogao da napravi više iteracija kroz povratnu (feedback) petlju pre nego što završi svoje aktivnosti. Kada kernel obavi prebacivanje konteksta, on obnavlja kontekst procesa tako da proces uvek nastavlja tamo gde je stao, tj. gde je bio suspendovan.

Algoritam za rapoređivanje

U okviru svakog prebacivanja konteksta kernel mora obaviti algoritam raspoređivanja koji će izabrati proces iz reda čekanja spremnih procesa (ready queue) i to proces sa najvećim prioritetom. To važi samo za procese koji su memoriji tj. spremni su za rad; ne može se selektovati proces koji je u swap prostoru. Ako više procesa ima isti prioritet, bira se onaj koji je najduže čekao, poštujući RoundRobin algoritam. Ako nema procesa u redu čekanja spremnih procesa, CPU će biti besposlen (idle), čeka se prvi otkucaj časovnika (timer tick), a onda se ponovo pokušava raspoređivanje (scheduling).

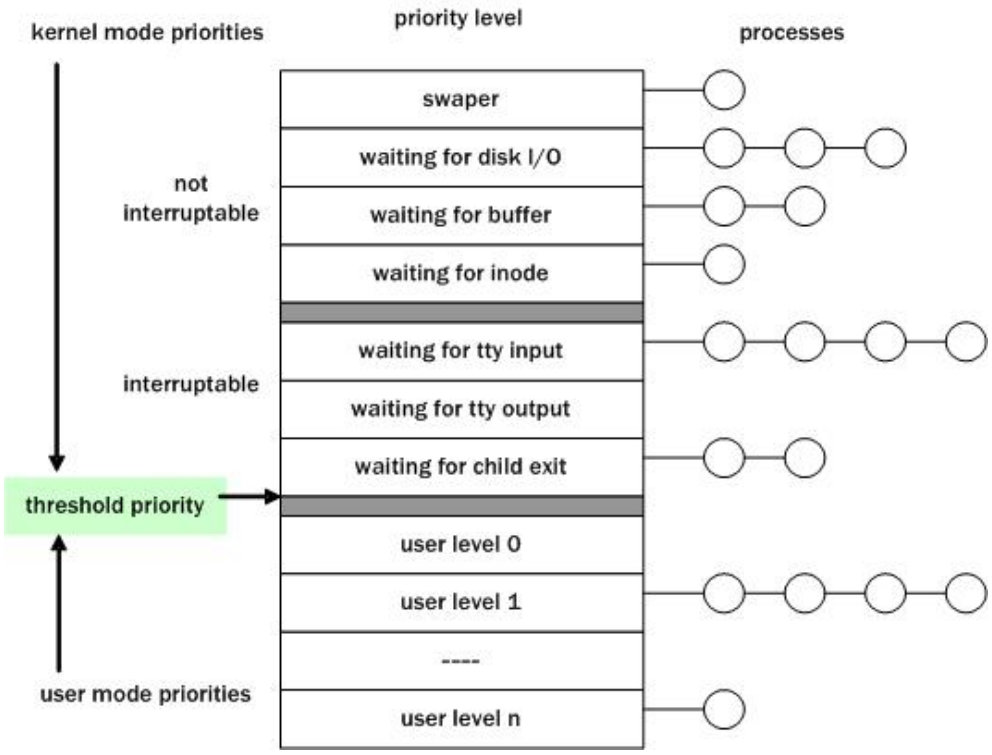
```
algorithm  schedule_process

input: none
```

```
output: none

{
  while(no process picked to execute)
  {
    for (every process on run queue)
    pick highest priority process that is loaded in memory;
    if(no process eglible to execute)
      idle the machine;
      /*interupt takes machine out of idle state*/
  }
  remove chosen process from run queue;
  switch context to that of chosen process, resume its execution;
}
```

Parametri za raspoređivanje

Svaki ulaz u tabeli procesa PT ima polje za prioritet, pri čemu je prioritet procesa u korisničkom modu funkcija njegovog nedavnog CPU korišćenja procesa. Opseg procesovih prioriteta može se podeliti u dve klase kao na slici: korisnički prioriteti i kernelski prioriteti. Svaka klasa sadrži više prioritetnih vrednosti i svaki prioritet ima red čekanja (queue) za procese koji se logički dodeljuju u taj red čekanja. Procesi sa korisničkim (user-level) prioritetima mogu nasilno da izgube CPU (preemption) na njihovom povratku iz kernelskog u korisnički mod, a procesi sa kernelskim prioritetima, svoje prioritete dobijaju u sleep algoritmu. Korisnički prioriteti su ispod probojne (threshold) vrednosti, a kernelski prioriteti su iznad te probojne vrednosti. Kernelski prioriteti se dalje dele na klase: procesi sa nižim kernelskim prioritetima bude se po prijemu signala, dok procesi sa višim prioritetom nastavljaju da spavaju.

Slika 8.1 prikazuje probojni (threshold) prioritet između korisničkih prioriteta i kernelskih prioriteta kao dupla linija između dva prioriteta "waiting for child exit" i "user level 0". Prioriteti "swapper", "waiting for disk I/O", "waiting for buffer" "waiting for inode" su visoki i neprekidljivi (koji ne dozvoljavaju prekide) sistemski prioriteti, sa 1, 3, 2 i jednim procesom u redu čekanja. Kernelski prioriteti "waiting for tty input", "waiting for tty output"i "waiting for child exit" su niži prioriteti koji su prekidljivi (dozvoljavaju prekide), sa 4, 0, i 2 procesa u redu čekanja. Slika razdvaja korisničke prioritete koji se ovde nazivaju "user level 0", "user level 1".. "user level n", sa 0, 4 i 1 procesom u redu čekanja.



Slika 8.1. *Prioriteti procesa*

Izračunavanje prioriteta procesa

Kernel kalkuliše prioritet procesa u specifičnom stanju procesa.

Kernel dodeljuje prioritet procesu koji treba da ode na spavanje, pri čemu je prioritet vezan za uzrok uspavljivanja procesa. Prioritet ne zavisi od karakteristika procesa u odnosu na CPU i IO zahteve (I/O bound, CPU bound). Procesi koji spavaju u nižim algoritmima teže da izazovu mnogo više uskih grla u sistemu ako su neaktivni, pa im se daje veći prioritet. Na primer proces koji spava i čeka na disk I/O ima veći prioritet od procesa koji čeka na bafer iz više razloga:

- prvo, proces koji čeka na disk I/O već ima bafer i kad se probudi ima šanse da obavi procesiranje i potom oslobodi bafer i druge resurse, što je dobro za sistem.
- drugo: proces koji čeka da oslobodi bafer, možda čeka bafer koji drži proces koji čeka na disk I/O i kad se disk I/O završi oba procesa se uskoro bude, jer spavaju na istoj adresi-baferu. U protivnom, javlja se zastoj (deadlock); jedan proces drži

bafer a čeka ga proces većeg prioriteta koji čeka njegov bafer.

Kernel podešava prioritet procesa kada se ovaj vraća iz kernelskog u korisnički mod. Proces može prethodno da uđe u uspavano stanje, kada mu se menja prioritet iz korisničkog u kernelski mod, a onda mu se prioritet opet menja kada se vrati u korisnički mod. Kernel pamti koliko se koji proces izvršava na CPU i nastoji da bude korektan prema svima procesima.

Prekidna rutina za časovnik (clock handler) prilagođava prioritet svih procesa u korisničkom modu, i to svake sekunde (RoundRobin na 1 sekundu) i izaziva da kernel obavi algoritam raspoređivanja da bi sprečio da neki proces monopolizuje CPU.

Časovnik (clock) može prekinuti proces više puta za vreme njegovog vremenskog kvantuma, a na svaki otkucaj časovnika, prekidna rutina časovnika (clock handler) inkrementira polje u tabeli procesa PT koji memoriše informaciju o CPU korišćenju. Svake sekunde, prekidna rutina časovnika podešava isto polje na bazi funkcije slabljenja (decay function), pri čemu se slabljenje odnosi na prioritet procesa:

- decay(CPU) = CPU /2; (UNIX system V)

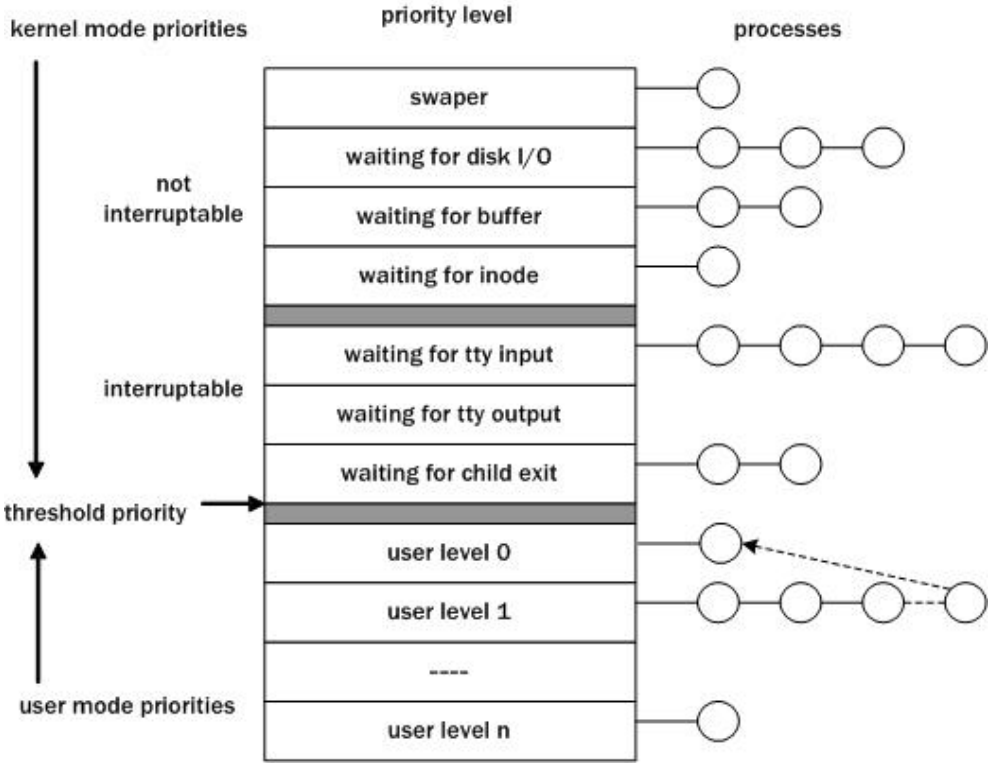
a zatim se ponovno izračunava prioritet za svaki proces i stanju 3 "preempteed but ready to run" po sledećoj formuli:

- priority = ("recent CPU usage"/2) + (base level user priority)

gde je "base level user priority" probojni prioritet između kernelskog i korisničkog moda. Što je broj manji to je prioritet procesa efektivno veći. Analizirajući formule za decay i prioritet, dolazimo do zaključka da što više proces radi na CPU, to ima veći broj (usage) a manji prioritet, a takođe što duže čeka to ima manji broj (usage) a veći prioritet

Efekat jedno-sekundne (one-second) rekalkulacije čini da se korisnički procesi pomeraju između korisničkih redova čekanja (user-queue), što je ilustrovano na slici 9.2, gde proces menja svoje prioritetne redove (queue) iz reda 1 u red 0. Na realnom sistemu, svi korisnički procesi menjaju svoje redove, ali se samo jedan proces selektuje za rad. Kernel ne menja prioritete procesima u kernelskom modu, niti dozvoljava korisničkom procesu da pređe u kernelski prioritet zbog druge formule. (jedino ako obavi sistemski poziv).

Kernel svake sekunde pokušava da ponovno izračuna prioritet svih korisničkih procesa, ali taj interval može da varira. Na primer, ako kernel izvršava svoje kritične sekcije, otkucaj sata se neće dogoditii, pa zato kernel pamti da je trebao da obavi rekalkulaciju. Kernel podešava inicijani prioritet na visoke vrednosti za neke procese kao što su editori teksa, koji imaju visok procenat slabog iskorišćenja CPU (high level of idle_time_to_CPU_usage ratio), pa samim tim oni imaju niski prioritet.



Slika 8.2. *Pomeranje korisničkih procesa između prioritetnih redova čekanja*

Postoje implementacije sistema UNIX gde vremenski kvantum dinamički menja vrednost između 0 i 1 sekunde, zavisno od opterećenja sistema. Takvi sistemi daju brži odziv, jer ne moraju da čekaju jednu sekundu, ali kernel ima više prebacivanja konteksta, a to smanjuje performanse.

Primeri raspoređivanja procesa

Primer 1

Slika 8.3 sadrži jedan prioritetni red za tri procesa A, B i C na sistemu UNIX Sistem V, pod sledećim uslovima: kreirani su simultano sa inicijalnim prioritetom 60 (threshold), a to je maksimalni broj za korisnički prioritet, otkucaj sata (timer tick) se dešava na jednu sekundu, nema sistemskih poziva i nema drugih procesa u stanju 3 (ready to run).

time	proc A		proc B		proc C	
0	priority	cpu count	priority	cpu count	priority	cpu count
	60	0	60	0	60	0
		1				
		2				
		..				
		60				
1	75	30	60	0	60	0
				1		
				2		
				..		
				60		
2	67	15	75	30	60	0
						1
						2
						..
						60
3	63	7	67	15	75	30
		8				
		9				
		..				
		67				
4	76	33	63	7	67	15
				8		
				9		
				..		
				67		
5	68	16	67	33	63	7

Slika 8.3. *Primer raspoređivanja za 3 procesa*

Kernel kalkuliše (izračunava) funkciju slabljenja (decay) preko formule:

decay(CPU) = CPU /2; (UNIX system V)

a zatim se ponovno izračunava prioritet za svaki proces i stanju 3 "preempteed but ready to run" po sledećoj formuli:

priority = ("recent CPU usage"/2) + 60

Pretpostavimo da je A prvi proces za izvršavanje i startuje da radi na vremenski kvantum od jedne sekunde. Za vreme od jednog vremenskog kvantuma, 60 puta se dogodi otkucaj časovnika (timer tick), tako da se polje za CPU korišćenje uveća na 60 za proces A. Potom ističe vremenski kvantum, kernel obavlja prebacivanje konteksta, rekalkuliše prioritete za sve procese (A=75, B=60, C=60) i bira proces B za izvršavanje. Opet se 60 puta događa otkucaj časovnika, čime ističe novi vremenski kvantum i opet se vrši rekalkulacija proriteta.

Primer 2

Posmatrajmo slučaj na slici 8.4 sa prioritetima i pretpostavimo da su to jedini procesi u sistemu. Kernel može više puta oduzeti CPU (preempt) za proces A i vratiti ga u stanje 3 (ready-to-run), a kao posledica više vremenskih kvantuma koji su se dogodili, njegov prioritet pada (slika a). Zatim u sistem ulazi proces B čiji inicijalni prioritet može biti veći od a (slika b). Ako postoje i drugi procesi, A i B mogu ostati neko vreme u redu čekanja spremnih procesa (ready-queue), tako da im se prioriteti približavaju ili izjednačavaju (slika c i d). Takođe, može se dogoditi da A bude selektovan jer duže čeka u istom redu čekanja (queue).

Podsetimo da kernel bira proces na završetku prebacivanja konteksta; proces mora da obavi prebacivanje konteksta kada ide na spavanje ili kad obavi sistemski poziv exit, a postoji mogućnost da obavi prebacivanje konteksta kada se vraća iz kernelskog moda u korisnički mod (preemption) i to se dešava ako u redu čekanja spremnih procesa postoji proces većeg prioriteta. To se dešava ako kernel probudi proces visokog prioriteta odnosno svaki probuđeni kernelski proces ima veći prioritet; drugi slučaj je kada prekidna rutina za časovnik (clock handler) obavi reviziju prioriteta za sve u ready-queue, pa tekućem procesu padne prioritet a drugome procesu se poveća.

Kontrolisanje prioriteta procesa

Procesi mogu sami da utiču na svoj prioritet preko sistemskog poziva nice:

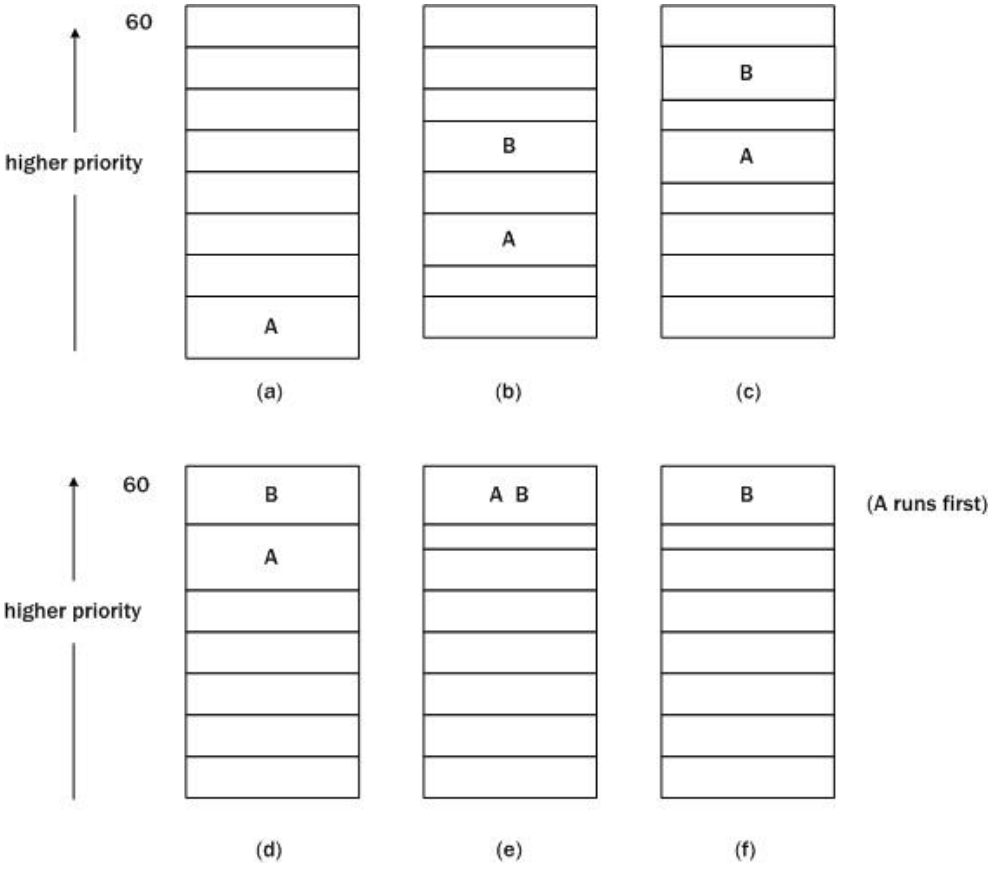
```
nice(value);
```

gde je value vrednost koja se dodaje na formulu

priority = ("recent CPU usage"/constant) + (base level user priority) +(nice value)

Sistemski poziv nice može inkrementirati ili dekrementirati nice polje u tabeli procesaa PT za vrednost datu u sistemskom pozivu nice, mada samo super-korisnik root može povećavati prioritet procesa. Običan korisnik može smanjiti svoj prioritet. Procesi nasleđuju nice vrednost od svog roditelja. Sistemski poziv nice radi samo na procesu koji

se izvršava (proces samom sebi menja nice i nijednom drugom procesu). To znači da administrator ne može da smanji prioritet procesima koji zauzimaju dugo CPU, osim da ih nasilno prekine sistemskim pozivom kill.



Slika 8.4. *Promena prioriteta za 2 procesa u vremenu*

FSS (Fair Share Scheduler)

Do sada opisan algoritam za raspoređivanje ne razlikuje klase korisnika. No pojavila se potreba za uvođenjem mehanizma za favorizovanje grupe korisnika, kako bi im se obezbedio bolji odziv. Šema se naziva Fair Share Scheduler. Princip FSS se ogleda u deljenju svih korisnika na fair-grupe (fair share groups), fair grupe su takve da su u njima svi procesi ravnopravni, tako da članovi grupe imaju obično RR raspoređivanje za svoju grupu, a sistem alocira CPU proporcionalno svakoj grupi bez obzira na broj procesa u

grupi. Na primer, pretpostavimo da postoje četiri fair-grupe u sistemu i da svaka grupa dobije 25% CPU i da grupe sadrže jedan, dva i četiri procesa, koji uglavnom imaju CPU korišćenje (CPU bound process) i koji nikada ne završavaju. Na primer, svaki predstavlja beskonačnu petlju (endless loop). Imamo deset procesa i ako bi se koristio običan RR, svaki bi proces imao 10% CPU iskorišćenja (postoji 10 procesa). Ali ako se koristi FSS, proces u grupi 1 imaće dva puta više vreme za CPU koršćenje, nego u grupi 2, tri puta više nego u grupi 3 i četri puta više nego u grupi 4, dok za istu grupu procesi imaju ravnomerno dodeljen CPU.

Implementacija ove šeme je jednostavna, a može joj se dodati grupni prioritet "fair share group priority". Svaki proces ima novo polje u svom u-području, koje ukazuje na grupno CPU korišćenje, nazvaćemo ga FSG (fair share CPU usage) i to grupno polje je zajedničko za sve procese iz grupe. Svaki otkucaj časovnika (timer tick) inkrementira polje FSG CPU, kao što inkrementira polje za CPU korišćenje za tekući proces. Svake sekunde izračunava se decay vrednost za sve procese i decay za celu grupu. Kada se izračunava prioritet procesa, nova komponenta koja se dodaje u formulu je grupno CPU korišćenje, a to predstavlja CPU vreme dodeljeno grupi i koje uvećava svaki proces iz grupe, koji koristi CPU.

Na primer, uzmimo opet ona tri procesa, ali tako da je A u jednoj grupi a B i C u drugoj grupi. Neka se prvo izvršava A, što će inkrementirati CPU polje i grupno polje u toku jednog vremenskog kvantuma . Kada se ponovno izračunava prioritet, procesi B i C su većeg prioriteta, i neka kernel izabere B. B se izvršava, ažirira se njegovo CPU polje i grupno polje zajedničko za B i za C. Kada se u drugoj sekundi izračunava prioritet, za proces C će da bude 75, a za proces A 74, pa je sekvenca procesa A, B, A, C, A itd, kao na slici 8.5.

Procesiranje u realnom vremenu

Real time procesiranje implicira mogućnost obezbeđivanja trenutnog odgovora na eksterne događaje, tako što se izabere odgovarajući proces kao odziv na događaj, ali u ograničenom intervalu vremena. Na primer, automatski (life-suport) sistem u bolnici mora reagovati na svaku promenu stanja bolesnika. Na drugoj strani editori teksta nisu real-time, korisnik očekuje brz odgovor, ali ako sačeka par sekundi, to nije strašno.

Prethodni algoritmi za raspoređivanje nisu pogodni za real-time sisteme, zato što ne mogu da garantuju da će startovati neki proces u okviru fiksnog vremenskog ograničenja. Druga nepovoljnost za real-time kod UNIX operativnog sistema je što je kernelski mod non-preemptive, kernel ne može izabrati real-time proces koji je u korisničkom modu ako već izvršava proces koji je u kernelskom modu, osim ako se UNIX osetno ne promeni.

U principu, programeri bi mogli ubaciti real-time procese u kernel da bi postigli real-time odziv. Pravo rešenje je dozvoliti real-time procesima da postoje dinamički (ne u kernelu), ali da imaju mehanizam da obaveste kernel o njihovim real-time ograničenjima.

Time	proces A			proces B			proces C		
0	priority	CPU	Group	priority	CPU	Group	priority	CPU	Group
	60	0	0	60	0		60	0	
		1	1						
		2	2						
		..	..						
		60	60						
1	90	30	30	60	0	0	60	0	0
					1	1			1
					2	2			2
					..	..			..
					60	60			60
2	74	15	15	90	30	30	75	0	30
		16	16						
		17	17						
		75	75						
3	96	37	37	74	15	15	67	0	15
						16		1	16
								2	
					15	75		60	75
4	78	18	18	81	7	37	93	30	37
		19	19						
		20							
		78	78						
5	98	39	39	70	3	18	76	15	18

Slika 8.5. *Primer za FSS*

8.2. Sistemski pozivi za vreme

Postoji više sistemskih poziva za vremenske funkcije: stime, time, times i alarm; stime i time se bave globalnim sistemskim vremenom, a times i alarm se bave vremenom za individualne procese.

Sistemski poziv stime

Sistemski poziv stime dozvoljava superuseru root da setuje globalnu kernelsku promenljivu koja sadrži vrednost za tekuće sistemsko vreme:

```
stime(pvalue);
```

gde je argument pvalue tipa long integer koji daje vreme mereno u sekundama u odnosu na Januar 1, 1970 GMT. Prekidna rutina za časovnik inkrementira ovu promenljivu svake sekunde.

Sistemski poziv time

Sistemski poziv time uzima sistemsko vreme kao

```
time(tloc);
```

gde je tloc korisnička lokacija u koju poziv upisuje vreme. Komande tipa date koriste sistemski poziv time da bi dobili tekuće vreme.

Sistemski poziv times

Sistemski poziv times dobija kumulativno vreme koje je pozvani proces proveo izvršavajući se u korisničkom i kernelskom modu i kumulativna vremena koja su sva njegova zombie deca provela izvršavajući se u korisničkom i kernelskom modu. Sintaksa za times je:

```
times(tbuffer)
struct tms *tbuffer;
```

gde je struktura tms definisana na sledeći način:

```
struct tms
{
  /* time_t is data structure for time*/
  time_t tms_utime;  /* user time of process*/
  time_t tms_stime;  /* kernel time of process*/
  time_t tms_cutime; /* user time of children*/
```

```
  time_t tms_cstime; /* kernel time of children*/
}
```

Times vraća proteklo vreme iz proizvoljne tačke iz prošlosti, obično od trenutka podizanja sistema.

Primer sistemskih poziva za vreme

Dat program koji kreira desetoro dece, i svako dete obavlja petlju 10000 puta. Proces roditelj poziva sistemski poziv times, pre kreiranja dece i kada sva deca završe (exit), a proces dete zove times pre i posle svoje petlje. Neko će naivno pomisliti da su roditeljska vremena (user time of children i kernel time of children) jednaka zbiru vremena korisničkih i kernelskih parametara dece, ali deca uopšte ne računaju sistemski poziv fork kao i svoj sistemski poziv exit. Takođe, sva vremena se menjaju zbog prekida i prebacivanja konteksta.

```
#include <sys/types.h>
#include <sys/times.h>
extern long times();
main()
{
  int i;
  struct tms pb1, pb2;
  long pt1, pt2;
  pt1 = times(&pb1); /*start time*/
  for (i=0; i<10; i++)
  if (fork()==0) child(i);
  for (i=0; i<10; i++) wait((int *) 0 );
  pt2 = times(&pb2); /*finished time*/
  printf("parent real %u user %u sys %u cuser %u csys %u",
         pt2-pt1, pb2.tms_utime-pb1.tms_utime,
         pb2.tms_stime-pb1.tms_stime,
         pb2.tms_cutime-pb1.tms_cutime,
         pb2.tms_cstime-pb1.tms_cstime);
}
child(n)
int n;
{
  int i;
  struct tms cb1, cb2;
  long t1, t2;
  t1 = times(&cb1); /*start time*/
  for (i=0; i<10.000; i++) ;
  t2 = times(&cb2); /*finished time*/
  printf("child %d real %u user %u sys %u cuser %u csys %u", n,
```

```
        t2-t1, cb2.tms_utime-cb1.tms_utime,
        cb2.tms_stime-cb1.tms_stime);
  exit();
}
```

Sistemski poziv alarm

Korisnički procesi mogu da pozovu alarm signale preko sistemskog poziva alarm. Na primer, sledeći program proverava vreme pristupa datoteci svakog minuta i prikazuje poruku da je datoteka imala pristup. Da bi se to uradilo, generiše se beskonačna petlja: za vreme svake iteracije poziva se sistemski poziv stat koji obaveštava o vremenu pristupa datoteci i ako je u toku zadnjeg minuta došlo do pristupa, štampa se poruka. Proces zatim postavlja sistemski poziv signal, da postavi signal cacther funkciju za alarm signal, poziva sistemski poziv alarm da emituje alarm signal za 60 sekundi i zove sistemski poziv pause da supenduje svoju aktivnost dok ne primi signal. Posle 60 sekundi, pojavljuje se alarm signal, kernel setuje korisnički stek za proces da pozove signal catcher funkciju; to je wakeup u ovom slučaju, koji vraća proces na stanje iza sistemskog poziva pause, a to je for petlja.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/signal.h>
main(argc, argv)
int argc; char *argv[]
{
  extern unsigned alarm();
  extern wakeup();
  struct stat statbuf;
  time_t axtime;
  if(argc != 2)
  {
    printf("only 1 arg");
    exit();
  }
  axtime = (time_t) ->0;
  for(;;)
  { /* find out file access time*/
    if(stat(argv[1], &statbuf ) == -1)
    {
      printf("file %s not there", argv[1]);
      exit();
    }
    if(axtime != statbuf.st_atime )
    {
      printf("file %s accessed", argv[1]);
```

```
        axtime  = statbuf.st_atime;
      }
      signal(SIGALRM, wakeup)
      alarm(60); /* wakeup after 60 sec)
      /*slanje alarm, signala procesu na zahtev*/
      pause(); /* go to sleep*/
    }
}
wakeup()
{
}
```

Zajednička osobina svih sistemskih poziva time je oslanjanje na sistemski časovnik, a kernel manipuliše različitim vremenskim brojačima kada upravlja prekidima za časovnik i inicira odgovarajuće akcije.

Clock

Funkcije za prekidnu rutinu za časovnik (clock interrupt handler) su:

- restart časovnika, tj. ponovno startovanje časovnika
- raspoređivanje pozivanja internih kernel funkcija baziranih na unutrašnjim tajmerima
- obezbeđivanje profiling funkcije za kernelske i korisničke procese
- skupljanje account statistike za sistem i proces
- čuvanje zapisa o vremenu
- slanje alarm signala procesima na zahtev
- periodično buđenje swapper procesa
- kontrola process-CPU raspoređivanja

Neke operacije se rade na svaki prekid tj. otkucaj časovnika, dok se neki izvršavaju nakon više otkucaja. Prekidna rutina za časovnik (clock handler) izvršava se na visokom CPU nivou izvršavanja, sprečavajući mnoge prekide da se obrađuju dok se izvršava prekidna rutinu za časovnik. Prekidna rutina za časovnik (clock handler) mora zato biti brza, zato što to kritično vreme kad su prekidi blokirani mora da bude veoma kratko.

Algoritam za časovnik-clock

Algoritam za časovnik clock je:

```
algorithm  clock

input: none
output: none
```

```
{
  restart clock; /* so that it will interrupt again*/
  if(callout table not empty)
  {
    adjust callout time;
    schedule callout function if time elapsed;
  }
  if(kernel profiling on)
    note program counter at time of interrupt;
  if(user profiling on)
    note program counter at time of interrupt;
  gather system statistics;
  gather statistics per process;
  adjust measure of process CPU utilization;
  if(1 second or more since last here and
     interupt not in critical region of code)
  {
    for (all process in the system)
    {
      adjust alarm time if active;
      adjust measure of process CPU utilization;
      if(process to execute in user mode)
        adjust process priority;
    }
    wakeup swapper process if is necessary;
  }
}
```

Restartovanje časovnika

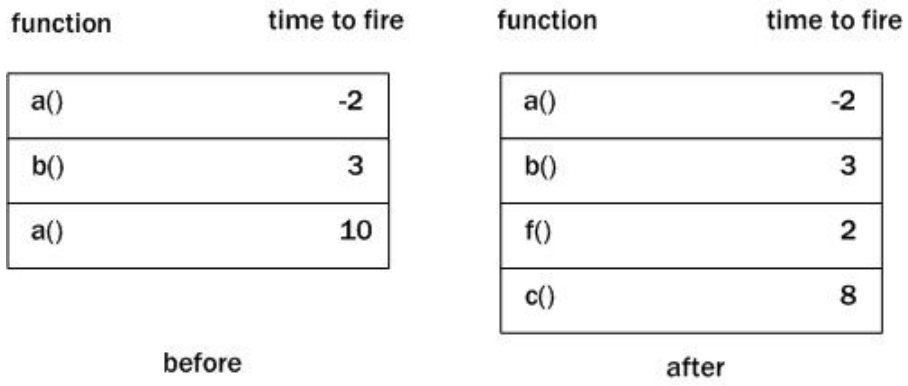
Kada se dogodi otkucaj časovnika (clock interrupt), mnoge mašine zahtevaju, da isti prekid može ponovo da se dogodi, a takve instrukcije zavise od hardvera i nećemo ih ovde diskutovati.

Interni sistemski tajmauti

Neke kernel operacije, a posebno drajveri i mrežni protokoli, zahtevaju pozivanje kernel funkcija na real-time osnovi. Na primer, proces može postaviti terminal u neobrađeni (raw) mod, tako da kernel zadovoljava read zahteve u fiksnim vremenskim intervalima, umesto da čeka korisnika da otkuca CR karakter (enter). Kernel čuva sve potrebne informacije u callout tabeli (viditi algoritam za clock), koja se sastoji od funkcije koju treba izvršiti kada istekne vreme, parametar za funkciju i broj otkucaja časovnika (timer tikova) koji treba da se dogodi pre nego što se funkcija pozove.

Korisnik nema direktnu kontrolu nad ulazima callout tabele, ali različiti kernel algoritmi mogu da ih kreiraju po želi korisnika. Kernel sortira ulaze callout tabele prema parametru "time to fire", odnosno vremenu kada funkcija treba da bude startovana, nezavisno od redosleda po kojem su pristigli u tabelu. Zbog vremenskog poretka polje za "time to fire" se namešta u odnosu na prethodni ulaz (previous time to fire). Totalni time to fire je zbir svih elemenata ispred ulaza uključujući sam ulaz.

Slika 8.6 prikazuje jednu callout tabelu i dodavanje novog ulaza za funkciju f koju treba izvršiti posle 10 otkucaja časovnika. Zato se funkcija f ubacuje između b i c, a za c se menja realni broj u osam.



Slika 8.6. *Primer za callout tabelu*

Kernel može koristiti povezane liste za svaki ulaz tabele ili može podesiti sve ulaze, što je bolja varijanta kada se callout tabela ne koristi mnogo.

Na svaki otkucaj sata, prekidna rutina za časovnik (clock handler) proverava da li ima ulaza u callout tabeli i ako ih ima dekrementira se time-to-fire polje prvog ulaza. To praktično znači da su i svi ostali ulazi dekremntirani, jer kernel sabira brojeve svih prethodnih. Ako je vremensko polje prvog ulaza manje ili jednako nula, tada se poziva odgovarajuća funkcija. Prekidna rutina za časovnik (clock handler) ne poziva funkciju direktno, jer bi mogao da se blokira (na primer rutina za časovnik bi se blokirala ako poziva funkciju koja traje duže od jednog otkucaja sata, pa bi taj otkucaj bio izgubljen. Umesto toga, prekidna rutina za časovnik poziva callout funkciju u vidu softverskog prekida, a on je na CPU nivou izvršavanja koji dozvoljava hardverske prekide. Dok se izvršava funkcija (software interrupt) mnogi se otkucaju časovnika mogu dogoditi, pa prvo polje može imati negativne vrednosti. Kada se softverski prekid na kraju završi, uklanjaju se svi ulazi iz callout tabele koji su istekli a potom se pozivaju njihove funkcije.

Kada je polje prvog ulaza negativno, prekidna rutina za časovnik (clock handler) mora naći prvi pozitivan ulaz i dekrementirati ga. Na slici, vrednost prvog ulaza je -2, što znači da su prošla dva otkucaja časovnika od kada funkcija a trebala da se izvrši, pa kernel

preskače polje a i dekremenira polje za b.

Profiling funkcija

Kernel profiling funkcija obavlja merenje vremena, koliko je sistem radio u kernelskom a koliko u korisničkom modu i koliko su se pojedine rutine izvršavale u kernelu. Kernel profile driver monitoriše relativne performanse kernelskih modula, odabirajući sistemske aktivnosti kada se dogoditi tajmerski prekid tj. otkucaj časovnika. Profile drajver ima listu kernelskih adresa za odabiranje (sampling), to su obično adrese kernelskih funkcija. Proces je prethodno preuzeo te adrese upisom u profile drajver. Ako je kernelski profiling uključen, prekidna rutina za časovnik (clock handler) poziva prekidnu rutinu (interrupt handler) za profile drajver, koji prvo određuje koji je mod (korisnički/kernel) bio pre otkucaja časovnika. Ako je bio korisnički mod, profiler inkrementira interni brojač koji odgovara vrednosti PC registra. Korisnički procesi mogu čitati profile drajver da dobiju kernelske brojače i obave statistička merenja. Na primer, sledeća tabela prikazuje adresu kernelskih rutina:

Algorithm	Address	count
bread	100	5
breada	150	0
bwrite	200	0
brelse	300	2
getblk	400	1
user	-	2

Ako je sekvenca vrednosti registra PC, odabranih u deset otkucaja sata: 110, 330, 145, 125, 440, 130, 320, 104, onda pratite sliku iznad. Na prvi pogled, sistem je proveo 20% u korisničkom modu, a 50% u algoritmu bread.

Ako se profiling obavlja za duži period vremena slika korišćenja sistema je realnija. Ali profiler ne uključuje koliko je vremena sistem proveo u prekidnoj rutini za časovnik (clock handler) i kritične delove koda, zato što se profile ne poziva tada, a kritični delovi koda zauzimaju značajni deo profile funkcije.

Korisnik može obaviti profiling za procese u korisničkom modu pozivajući sistemski poziv profil:

```
profil(buff, bufsize, offset, scale);
```

gde je buff adresa polja (array) u korisničkom prostoru, bufsize je veličina polja, offset je virtuelna adresa korisničkog potprograma, i scale je faktor koji mapira korisničke vituelne adrese u polje (array). Kernel tretira scale parametar kao binarnu frakciju (fiksne dužine). Kada časovnik prekine program u korisničkom modu, prekidna rutina za časovnik (clock handler) određuje vrednost registra PC u vreme prekida, komparira offset, inkrementira lokaciju u buf polju na bazi parametara bufsize i scale.

Na primer, posmatrajmo program za profiling, koji poziva dve funkcije u beskonačnoj petlji. Proces prvo poziva signal da ubaci funkciju theend kao signal catcher funkciju za SIGINT, zatim kalkuliše opseg text adresa koju želi da profiluje, proširujući adresu funkcije main sa adresom funkcije theend i na kraju zove sistemski poziv profil, da informiše kernel da želi profil svog izvršavanja. Na prvi prekid sa tastature se generiše rezultat na ekranu (buf display).

```
#include <signal.h>
int buffer[4096]
main()
{
  int offset, endof, scale, eff, gee, text;
  extern theend(), f(), g();
  signal(SIGINT, theend)
  endof = (int) theend;
  offset = (int) main;
  /* calculate number of words in program text*/
  text = (endof-offset+sizeoff(int)-1)/sizeoff(int);
  scale=0xffff;
  printf("offset %d endof  %d text  %d ", offset, endof, text)
  eff = (int) f;
  gee = (int) g;
  printf("f %d g %d fdiff %d gdiff %d", eff, gee,
    eff-offset, gee-offset)
  profil(buffer, sizeoff(int)*text, offset, scale);
  for(;;)
  {
    f();
    g();
  }
}
theend()
{
  int i;
  for(i=0, i<4096; i++)
  if (buffer[i]) printf("buf[%d] = %d", i, buffer[i]);
  exit();
}
```

Izlaz bi mogao da bude sledeći:

```
offset 212 endof 440 text 57
f 416 g 428 fdiff 204 gdiff 216
buf[46] = 40
buf[48] = 8585216
buf[49] = 151
buf[51] = 12189799
buf[53] = 65
buf[54] = 10682455
buf[56] = 67
```

Adresa f je 204 veća od početne (0th) profiling adrese, pa se f funkcija mapira na bafer ulaze 51, 52, 53. Slično, g funkcija se mapira na 54, 55 i 56. Adrese 46, 48 i 49 su adrese glavne petlje u main funkciji.

Obračun i statistika

Kada časovnik prekine sistem, sistem može imati izvršavanje u kernelskom ili u korisničkom modu ili može da bude besposlen. Ako je besposlen (idle), svi procesi su uspavani. Kernel čuva interne brojače za sva stanja procesora i podešava ih uvek za vreme otkucaja časovnika tj. clock prekida. Korisnik procesi mogu kasnije da izvuku te infomacije iz kernela i naprave statistiku.

Svaki proces ima dva polja u svom u-području u kome kernel upusuje (log) korišćenje memorije od strane procesa (memory usage). Kada se dogodi otkucaj časovnika (clock prekid), kernel izračunava totalnu memoriju za privatne regione procesa i proporcionalno korišćenje deljivih (shared) regiona. Na primer, ako proces deli text region od 50K sa četiri druga procesa, a koristi svoj privatni region podataka i region steka veličine 25 i 40K, kernel tereti proces sa 75K (50/5 + 25 +40). Za sistem straničenja, kalkuliše se ukupan broj validnih stranica u svakom regionu, tako što sabira sve validne stranice u privatnim regionima sa proporcionalnim brojem deljivih (shared) stranica. Kernel upisuje ove informacije u zapis za statistiku (accounting record) kada proces obavi sistemski poziv exit i to se može koristiti za naplaćivanje usluga korisnicima.

Održavanje vremena

Kernel inkrementira sistemsku timer promenljivu na svaki otkucaj časovnika (timer tick), čuvajući vreme u broju otkucaja od podizanja UNIX sistema. Kernel koristi ovu promenljivu da vrati vreme na zahtev sistemskog poziva time i izračunava ukupno vreme (real) izvršavanja procesa. Kernel čuva procesovo početno vreme (start time) u u-području procesa još u sistemkom pozivu fork i oduzima vreme iz sistemske timer promenljive kada proces završi aktivnosti (exit) i tako dobije realno vreme izvršavanja procesa. Druga vremenska promenljiva ažurira se svake sekunde i služi za sistemski poziv stime, a opisuje kalendarsko vreme.

9

Upravljanje memorijom

9.1. Uvod u upravljanje memorijom na UNIXu

Algoritmi za CPU raspoređivanje (scheduling) imaju snažan uticaj na tehnike za upravljanje memorijom (memory management policies). Deo računarskog sistema koji upravlja memorijom naziva se MMU (memory management unit). Da bi proces mogao da počne izvršavanje, barem neki deo procesa se mora nalaziti u memoriji, tj. CPU ne može izvršavati proces koji je potpuno na swap prostoru. Zahvaljujući swap konceptu, uvek može biti više aktivnih procesa nego što bi se oni mogli naći u memoriji istovremeno. To je jedan od glavnih zadataka MMU, da odluči koji će proces da bude delimično ili potpuno u memoriji a koji na swap prostoru . MMU monitoriše zauzeće memorije i povremeno neke procese upisuje na swap uređaj, a povremeno neke vraća sa swap prostora u glavnu memoriju.

Prvi UNIX sistemi transferišu cele procese između swap prostora i glavne memorije, pri čemu jedini delimičan transfer može biti transfer deljivog (shared) text regiona, a takva MMU se naziva swaping MMU. Prvi UNIX sistemi imali su malu maksimalnu veličinu procesa od 64K, što je uslovljavala mala količina RAM memorije. BSD UNIX počevši od release 4.0 je uveo DP (demand paging) MMU, baziranu na straničenju po zahtevu. U DP tehnici, MMU transferiše stranice memorije između memorije i swap prostora, a ne cele procesi. Nakod BSD UNIX sistema, DP tehniku je kasnije uveo i UNIX System V. Straničenje (paging) je uvelo puno poboljšanja, dovoljno je da jedna stranica procesa bude u memoriji da bi on počeo da se izvršava, a sve ostale stranice se dobijaju preko DP tehnike. Prednost koji donosi DP je u tome što omogućava veliku fleksibilnost u mapiranju između virtuelne i fizičke memorije, dozvoljavajući da veličina procesa bude veća od fizičke memorije, a i više procesa mogu da budu u redu čekanja spremnih procesa (ready queue), što znači da su simultano prisutni u memoriji.

Swaping MMU je lakša za realizaciju i ima manji sistemski overhead, tj. gubitak vremena na opsluživanje sistemskih funkcija. U ovoj lekciji objasnićemo obe MMU tehnike.

Swapping tehnika

Swapping algoritam ima tri funkcionalna dela ili komponente:

- upravljanje swap uređajem
- swap-out (prebacivanje celog procesa iz memorije na swap prostor)
- swap-in (vraćanje celog procesa sa swap prostora u memoriju)

Alokacija swap prostora

Swap uređaj je blok uređaj u konfigurabilnoj sekciji diska. Kad su u pitanju datoteke, kernel datotekama dodeljuje jedan blok sa diska u vremenu. Kada je u pitanju swap, swap prostor se alocira u grupama kontinualnih blokova. Prostor alociran za datoteke koristi se statički, pošto blok može pripadati datoteci veoma dugo, pa alokaciona politika mora da bude optimizovana da smanji fragmentaciju datoteke na disku. Alokacija blokova na swap prostoru je tranzijenta i zavisi od trenutnog stanja izvršavanja procesa. Proces koji je na swap prostoru će napustiti swap prostor i osloboditi blokove koje je zauzimao. Zbog sporosti disk I/O uređaja, poželjeno da transferi sa diska budu veći, pa je zato alokacija blokova u swap prostoru za jedan proces uvek kontunualna bez obzira na fragmentaciju, tako da ceo proces može da se transferiše na swap ili sa swap prostora u jednom disk IO transferu.

Pošto je swap alokacija drugačija u odnosu na alokaciju u sistemu datoteka (FS), stuktura podataka za vođenje evidencije o slobodnom prostoru je takođe različita. Kernel podržava listu slobonih blokova za sistem datoteka kao povezanu listu slobodnih blokova kojoj se pristupa preko super-blok strukture za taj sistem datoteka. Za swap se kreira posebna struktura u memoriji koja se naziva swap mapa (in-core swap-map), koja radi na principu first-fit alokacije.

Mapa za swap je polje sa ulazima gde se svaki ulaz sastoji od adrese alokatibilnog resursa i od broja jedinica koji su tu raspoloživi (free). Kernel interpretira adresu i broj jedinica nagore, tj. prema vrhu swap mape. Inicijalno, swap mapa sadrži jedan ulaz koji ukazuje na adresu i totalni broj resursa. Na primer, kernel tretira svaku jedinicu swap mape kao grupu disk blokova i tretira adresu kao pomeraj u blokovima (block offset) od početka swap područja. Slika 9.1 ilustruje inicijalnu swap mapu, koja se sastoji od 10.000 blokova, a blokovi su veličine 1K, što daje ukupan swap prostor od 10MB, koji počinju na adresi 1.

Slika 9.1. *Inicijalna swap mapa*

Kada kernel dodelju i oslobađa swap resurse, ažurira se swap mapa da opisuje novi slobodni prostor.

Algoritam malloc

Navodimo algoritam malloc za alokaciju prostora na bazi swap mape.

```
algorithm malloc /* algorithm to allocate map space */

input:  (1) map address / *indicate which map to use */
        (2) requested number of units
output: address if successful, 0 otherwise

{
  for(every map entry)
  {
    if(current map can it fit requested units)
    {
      if(requested units == number of units in entry)
        delete entry from map;
      else
        adjust start address of entry;
      return(original adress of entry);
    }
  }
  return(0);
}
```

Primeri za malloc

Kernel pretražuje swap mapu za prvi ulaz koji sadrži dovoljno slobodnog prostora da zadovolji zahtev. Ako taj zahtev dobije sve blokove sa tog ulaza, taj ulaz se briše iz swap mape i ona se komprimuje za jedan ulaz manje. Ako je ulaz u swap mapi veći od onoga što se traži, ulaz se modifikuje, početna adresa se uvećava za broj dodeljenih blokova, a broj slobodnih jedinca se smanjuje za broj dodeljenih blokova (start address = start address + requested_size, unit = unit – requested_size).

Slika 9.2 sadrži sliku swap mape, za slučaj pod (b) kada se dodeljuju 100 jedinica, za slučaj pod (c) kada se dodeljuju još 50 jedinica i za slučaj pod (d) kada se dodeljuju još 100 jedinica. Kernel prilagođava swap mapu, koja sada prikazuje da je prvih 250 jedinica alocirano i da sada sadrži 9750 jedinica na početnoj adresi 251.

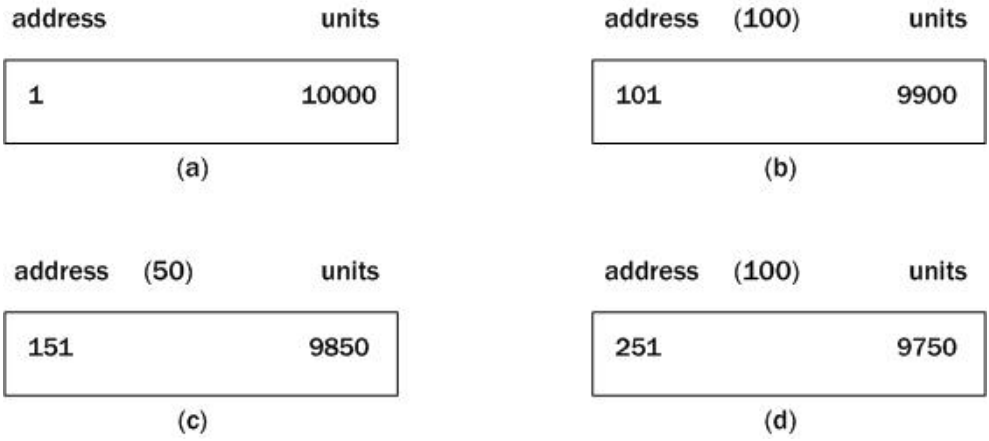
Kada se oslobađaju swap resursi, kernel nalazi njihove pozicije u mapi i tada su sledeće tri situacije moguće:

[1] U ovoj situaciji, oslobođeni resursi prave potpunu (full) šupljinu u swap mapi (sa obe strane), šupljina je kontinualna sa ulazima čiji adrese neposredno predhode novoj šupljini i prate je dalje u mapi. U ovom slučaju, kernel kombinuje novu šupljinu sa dva susedna ulaza i kreira jednu veliku šupljinu sa jednim ulazom u mapi, a broj ulaza u mapi se smanjuje za jedan.

[2] U ovoj situaciji, oslobođeni resursi delimično prave šupljinu u mapi (samo sa jedne strane). To je slučaj kada se nova šupljina naslanja na šupljinu sa donje strane (adrese joj prethode) ili se naslanja na šupljinu sa gornje strane (adrese joj slede). U tom slučaju, podešava se samo taj ulaz sa kojim se nova šupljina spaja, a broj

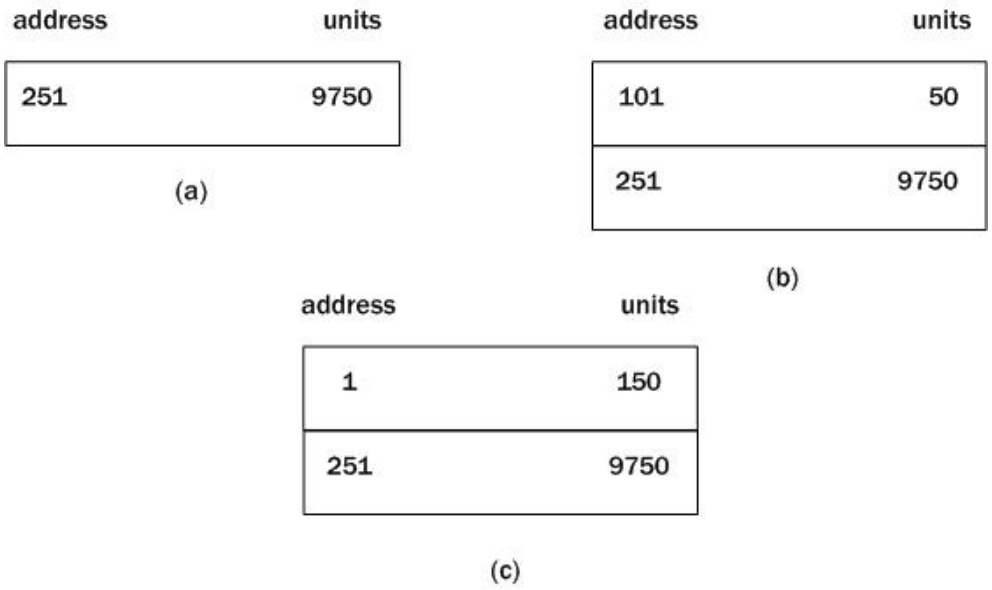
ulaza u mapi se ne menja

[3] U ovoj situaciji, oslobađanjem resursa dobija se nova šupljina, ali ona nije susedna sa nijednom postojećom šupljinom. Kernel ubacuje novi ulaz u mapu, koji opisuje novu šupljinu, tako da se broj ulaza u mapi povećava za jedan.



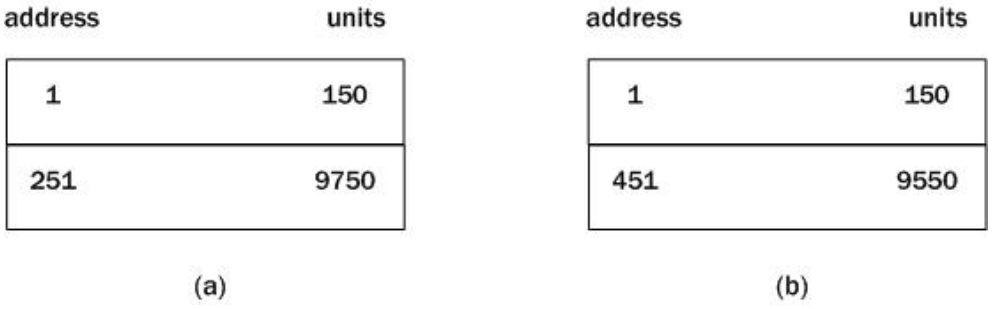
Slika 9.2. *Primer alokacije swap prostora*

Demonstrirajmo ove slučajeve, na slici 9.3, koja se nastavlja na prethodni primer, u kome je prvo došlo do alokacije swap blokova. Ako kernel oslobodi 50 swap jedinica na adresi 101 u swap prostoru, nova šuplljina se ne dodiruje sa postojećim šupljinama, u swap mapu se uvodi novi ulaz kao na slici pod (b). Ako potom kernel oslobodi 100 jedinica na adresi 1, onda će se šupljine spojiti sa gornje strane, pa imamo situaciju kao na slici pod (c).



Slika 9.3. *Povratak resursa u swap mapu*

Pretpostavimo da sada kernel zahteva novih 200 jedinica iz swap prostora. Pošto prvi ulaz sadrži samo 150 jedinica, kernel će zadovoljiti zahtev iz drugog ulaza mape, kao na slici 9.4 pod (b).



Slika 9.4. *Ponovno traženje resursa*

Na kraju, ako kernel oslobodi 350 jedinica, koji počinje na adresi 151, to su resursi koji su se dodeljivali odvojeno, ali kernel ih sada oslobođa odjednom, pa će prvih 300 jedinica swap prostora, napraviti šupljinu između prva ulaza, koja je spojena sa njima sa obe strane, pa će se tri ulaza u mapi spojiti u jedan ulaz.

Tradicionalni UNIX koristi samo jedan swap prostor, ali novi UNIX System V dozvoljava više swap prostora ili swap uređaja. Kernel bira swap uređaj po RoundRobin šemi, tako da obezbeđuje uvek dovoljno slobodnog swap prostora, a administrator može kreirati ili izbaciti swap prostor dinamički. Kada se neki swap uređaj ili prostor izbacuje, njegovi podaci moraju da se sačuvaju; prvo se prebacuju u memoriju, pa eventualno na neki drugi swap prostor, pa se tek onda izbacuje swap prostor.

Swap-out tehnika

Kernel obavlja swap-out procesa kada mu nedostaje fizička memorija, a to može da se dogodi u sledećim situacijama:

[1] Sistemski poziv fork mora alocirati prostor za novi proces-dete

[2] Sistemski poziv brk povećava veličinu procesa

[3] Veličina procesa postaje veća, zato što mu je stek region porastao

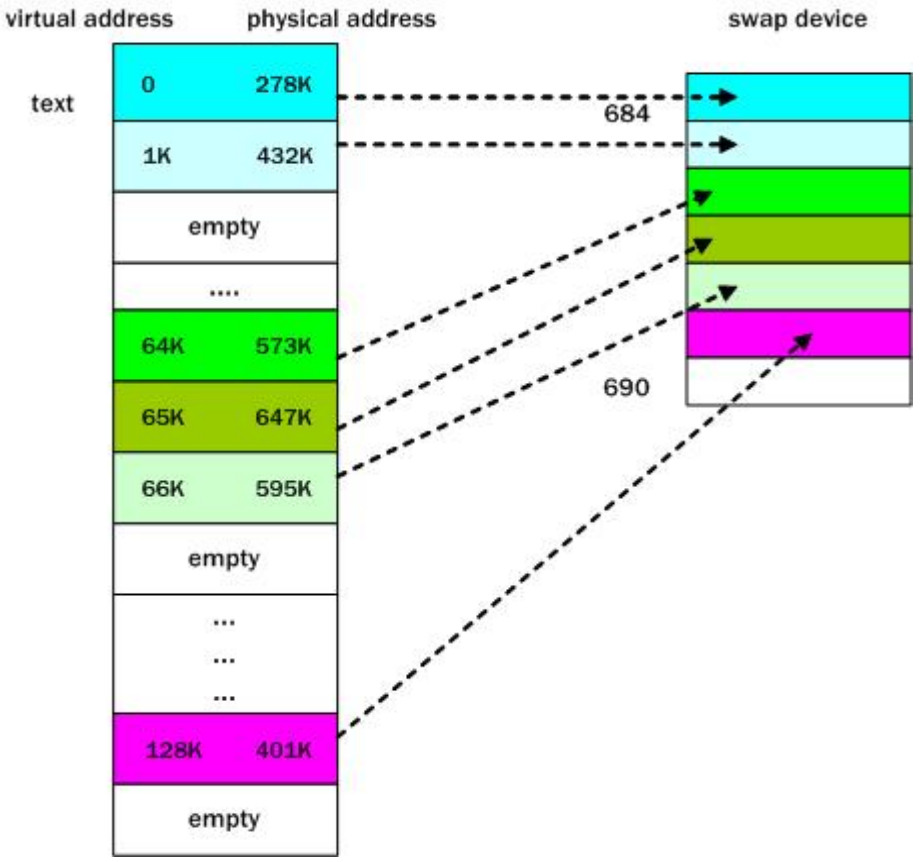
[4] Kernel želi da oslobodi prostor u memoriji za procese koji su prethodno na swap prostoru tj. u swap-out stanju, da bi ih vratio u memoriju tj. obavio swap-in funkciju

Prvi slučaj je unikatan (fork), jer to je jedini slučaj kada se prethodno zauzeta memorija ne oslobađa, zato što proces-roditelj drži svoje resurse a traži kompletno iste resurse za dete.

Kada kernel izabere proces za koga će obaviti swap-out, kernel dekrementira broj referenci za svaki region procesa i obavlja swap out za sve regione čiji je broj referenci pao na nulu. Kernel alocira prostor na swap uređaju i zaključava (lock) proces u memoriji (za slučajeve od 1-3). Na taj način štiti proces od kernelskog procesa swapper, dok se ne obavi tekuća swap-out operacija. Kernel čuva swap adresu regiona u region tabeli, tj. u ulazu region tabele za taj region.

Kernel obavlja swap-out sa maksimalnom količinom podataka koliko je to moguće po jednom disk I/O zahtevu i to direktno između korisničkog adresnog prostora i diska, preskačući baferski keš (by-pass). Ako hadrver ne može da prebaci više stranica odjednom, kernel u iteracijama prebacuje stranicu po stranicu. Sama tehnika transfera zavisi od disk kontrolera, od memorijske organizacije itd. Na primer kod sistema sa straničenjem (paging), virtuelne adrese su kontinualne, ali fizičke adrese su diskontinualne, što će uzrokovati više disk zahteva. Proces swapper, koji uglavnom obavlja ove funkcije, čeka da se kompletira jedna disk I/O operacija pre nego što otpočne nova.

Nije neophodno da kernel prebaci kompletan virtuelni adresi prostor procesa na swap prostor. Kopiraju se samo dodeljene fizičke adrese, dok nedodeljene virtuelne adrese ostaju neiskopirane. Kada kasnije kernel vraća proces u fizičku memoriju, kernel zna virtuelnu adresnu mapu procesa, tako da može da ponovo dodeli proces na korektne virtuelne adrese.

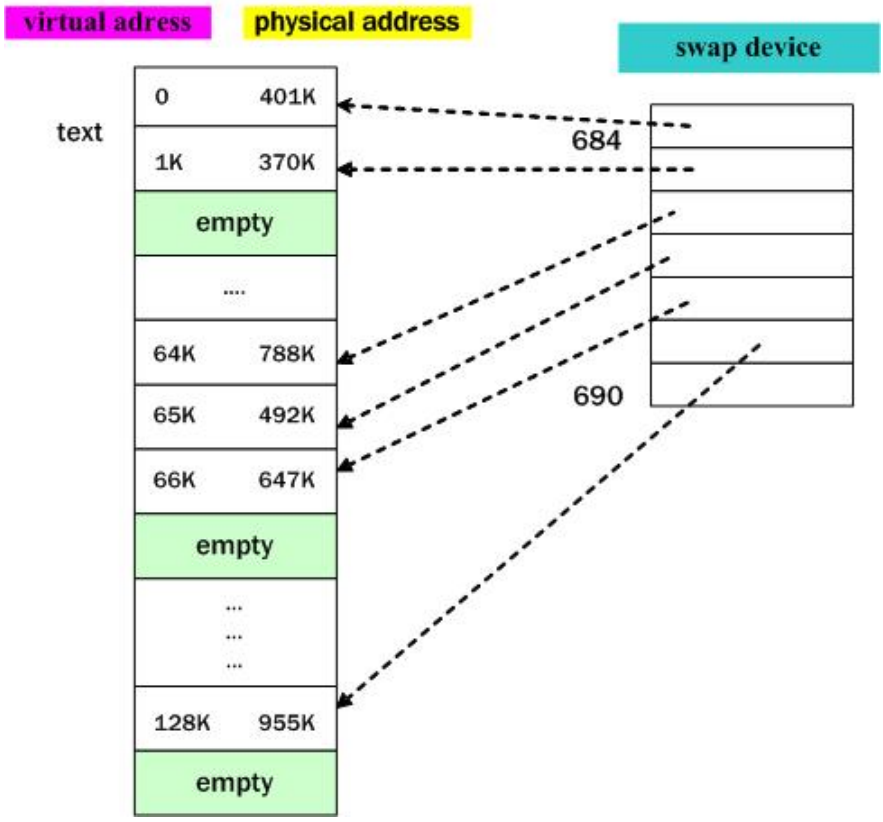


Slika 9.5. *Primer za swap-out operaciju*

Na slici je dat primer mapiranja memorijske slike procesa (in-core image) procesa na swap uređaj. Proces sadrži tri regiona za text, data i stek. Text region se završava na virtuelnoj adresi 2K, a region podataka startuje na virtuelnoj adresi 64K, ostavljajući prazninu (gap) od 62K u virtuelnom adresnom prostoru. Kada kernel obavlja swap-out za proces, kernel obavlja swap-out samo za „žive“ stranice procesa, a to su stranice na adresi 0, 1K, 64K, 65K, 66K i 128K. Praznine u virtuelnom adresnom prostoru ne prebacuje na swap prostor, a to je 62K između kraja text regiona i početka regiona podataka (data), kao i 61K između kraja regiona podataka i početka stek regiona. Kao što slika 9.5 prikazuje, swap prostor se uvek puni kontinualno.

Swap in operacija

Kada kernel obavlja swap-in operaciju, na osnovu procesove memorijske mape zna da proces ima šupljinu od 62K i na taj način dodeljuje fizičku memoriju u saglasnosti sa mapom. Taj slučaj je prikazan na slici 9.6. Zapažamo da fizičke adrese pre i posle swap operacija nisu iste, ali proces se bavi virtuelnim adresama, tako da se korisnički (user-level) kontekst nije ništa promenio, jer je virtuelni prostor ostao isti.



Slika 9.6. *Primer swap-in operacije*

Teorijski, i ostatak memorijskog prostora okupiranog procesom kao što je u-područje ili kernelski stek, poželjno je da se takođe upusuje na swap prostor. Kernel može privremeno da zaključa (lock) region u memoriji dok se obavlja osetljiva operacija. Praktično, kernel ne obavlja swap operaciju za u-područje, ako u-područje sadrži tabelu za translaciju adrese za procese. Različite implementacije takođe diktiraju da li proces

može da obavi swap-out operaciju za samog sebe ili mora da zahteva od drugog procesa da mu obavi swap-out operaciju.

Fork algoritam pomoću swap tehnike (Fork swap)

Opis fork algoritma podrazumeva da roditeljski proces ima dovoljno memorije da napravi dete proces (child context). U suprotnom, kernel obavlja swap-out procesa-roditelja, ali bez oslobađanja memorije koju proces-roditelj zauzima (to je kopija roditelja, a dete nije dobilo svoju memoriju). Kada je swap kompletan, dete proces postoji na swap prostoru, a roditelj ga postavlja u stanje 5 (ready to run in swap), a potom se proces roditelj vraća u korisnički mod (user-mod). Proces swaper može prebaciti proces-dete swap-in operacijom u stanje 3 (ready to run), pa kad ga proces raspoređivač (scheduler) izabere, proces-dete će okončati svoj deo sistemskog poziva fork i potom se proces-dete vraća u korisnički mod.

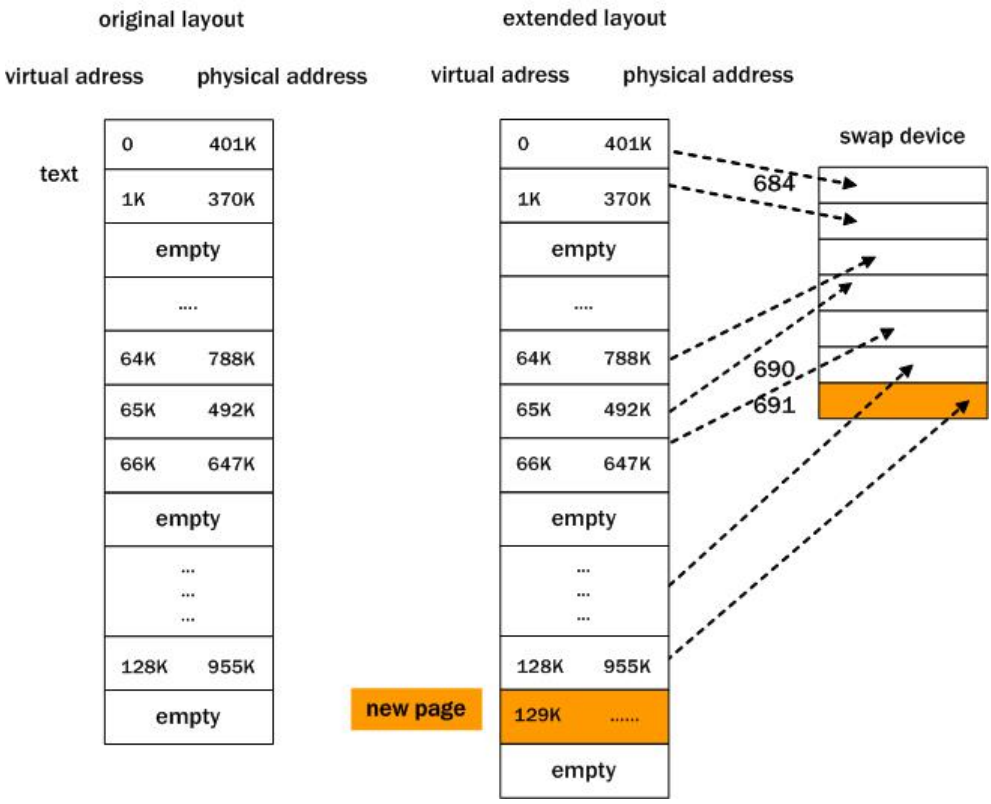
Proširenje procesa preko swap prostora

Ako proces zahteva više fizičke memorije od one koja mu je trenutno alocirana (kao rezultat rasta korisničkog steka ili usled sistemskog poziva brk), a ta veličina nije trenutno raspoloživa u fizičkoj memoriji, kernel obavlja proširenje swap prostora za proces. Kernel rezerviše mesto na swap prostoru, da sadrži kompletan prostor procesa uključujući i proširenje koje se zahteva, tako što se uspostavlja korespodencija između nove virtuelne adrese i swap prostora, a novoj virtuelnoj adresi se ne dodeljuje fizička adresa jer nije trenutno raspoloživa. Na kraju se za ceo proces obavlja swap-out operacija, a novi prostor tj. proširenje se puni nulama. Kasnije, swap-in operacija za proces, će dodeliti procesu fizičku adresu za novi prostor, što je prikazano na slici 9.7.

Swapper proces

Proces 0, swapper, je jedini proces koji obavlja operacije swap-in procesa sa swap uređaja u memoriju. Kada se operativni sistem UNIX podigne, proces swapper odlazi u beskonačnu petlju i njegov jedini posao je da obavlja swap operacije (swap-out, swap-in). Proces swapper ide na spavanje ako nema posla (slučaj kada nema ni jednog procesa za swap-in ili swap-out operacija). Kernel bira swapper kao i sve druge procese, ali to je proces koji se izvršava isključivo u kernelskom modu. Proces swapper ne koristi sistemske pozive, ali koristi interne kernelske finkcije.

Kao što smo već rekli, prekidna rutina za časovnik (clock handler) meri vreme koje svaki proces provodi u fizičkoj memoriji i na swap prostoru. Prilikom buđenja, proces swapper proverava sve procese koji su stanju 5 (ready to run, swapped) i bira jedan proces, koji je najduže na swap prostoru (swapped). Ako ima dovoljno memorije, swapper će obaviti swap-in operaciju za taj proces: alocira fizičku memoriju, čita proces sa swap uređaja i oslobađa swap prostor koji je zauzimao.



Slika 9.7. *Swap bazirano proširenje*

Posle uspešne swap-in operacije, swapper pretražuje i ostale procese stanju 5 (ready to run, swapped) i obavlja swap-in operaciju za njih. Jedna od sledećih sitacija može da mu se dogodi:

[1] nema procesa u stanju 5 (ready to run, swapped). Swapper odlazi na spavanje, dok se ne pojavi proces na swap uređaju

[2] swapper nalazi proces je u stanju 5 (ready to run, swapped) ali nema dovoljno fizičke memorije za swap-in operaciju. Tada proces pokušava da obavi swap-out operaciju za neki drugi proces iz stanja 3 (ready to run), čime se oslobađa memorija, pa ako uspe, swapper pokušava prethodnu swap-in operaciju za proces, koja je bila neuspešna.

Algoritam swapper

```
algorithm swapper
```

```
/* swap in, swap out process,
   swap out other processes to make room */

input:  none
output: none

{
loop:
  for(all swapped out process that are ready to run)
    pick a process swapped out longest;
  if(no such process)
  {
    sleep(event must swap in);
    goto loop;
  }
  if(enough room in main memory for process)
  {
    swap-in process;
    goto loop;
  }
/* loop 2: revised algorithm */
loop2:
  for(all processes loaded in main memory,
      not zombie and not locked in memory)
  {
    if(there is a sleeping process) choose process such that
      priority + residence time is numerically highest;
    else /*no sleeping process*/
    choose process such that priority + residence time is
      numerically highest;
  }
  if(chosen proces not sleeping or residency requirements
    not satisfied) sleep (event must swap process in)
  else
    swap out process;
  goto loop2;
}
```

Ako proces swapper mora da obavi swap-out, on ispituje sve procese u memoriji. Za zombie procese se ne obavlja swap-out operacija, zato što ne zauzimaju fizičku memoriji. Za procese koji su na neki način zaključani (locked) u memoriji, na primer procesi koji obavljaju region operacije, takođe se ne obavlja swap-out operacija. Kernel radi swap-out operacije za uspavane procese radije nego spremne procese (ready to run), zato što oni imaju veliku šansu da budu izabrani od raspoređivača . Izbor uspavanog procesa za swap-out se obavlja na bazi prioriteta i vremena koje je proces proveo u memoriji. Ako nema uspavanih procesa u memoriji, proces se bira između procesa u stanju 3 (ready to run), na bazi prioriteta, nice vrednosti i vremena koje je proces proveo u memoriji.

Uvodi se UNIX dvo-sekundno pravilo (two-seconds rule): Proces u stanju 3 (ready to run) mora biti u memoriji bar dve sekunde, pre nego što doživi swap out operaciju, a takođe neće se obaviti ni swap-in operacija procesa koji nije na swap prostoru proveo bar dve sekunde.

Ako nijedan proces ne ispunjava gornje uslove za swap-out opraciju, swapper odlazi na spavanje na događaj, koji znači da hoće da vrati proces u memoriju ali nema mesta za njega (want a swap process into memory but cannot find room for it). Prekidna rutina za časovnik (clock handler) budiće swappper na svaku sekundu. Takođe, pri svakom uspavljivanju nekog procesa, kernel će probuditi swapper. Swapper se uvek budi na početku svog algoritma.

Primeri za swapper

Na slici 9.8 dato je pet procesa i vreme provedeno u memoriji i na swap prostoru. Pretpostavimo da su svi procesi CPU orijentisani (CPU-bound) i da ne obavljaju ni jedan sistemski poziv, pa se prebacivanja konteksta dešavaju po RounnRobin algoritmu na jednu sekundu, ali tada se uvek izvršava i visoko prioritetni swapper. Pretpostavimo da su procesi iste veličine i da sistem ima mesta za samo dva procesa u memorji. Inicijalno, procesi A i B su u memoriji, dok su C, D i E na swap prostoru. Swapper ne može obaviti swap-out operaciju ni za jedan proces, za prve dve sekunde, zato što nema slobodne memorije i nijedan proces nije zadovoljio dvo-sekundno pravilo. Ali posle dve sekunde, swapper će obaviti swap-out operacije za procese A i B, a potom će obaviti swap-in operaciju za procese C i D (za proces E se ne obavlja swap-in zato što nema mesta u memoriji) i počeće da izvršava C. Posle treće sekunde, proces E bi mogao da ima swap-in operaciju, ali procesi C i D nisu proveli dve sekunde u memoriji. Tek u četvrtoj sekundi, proces swapper obavlja swap-out operacije za procese C i D, a potom obavlja swap-in operacije za proces E i A.

Za swap-in operaciju, swapper bira procese na bazi vremena provedenog od trenutka kad je proces doživeo swap-out operaciju, a drugi kriterijum je prioritet, jer će takvi procesi imati bolju šansu da se izvrše uskoro.

Algoritam za izbor swap-out procesa u slučaju da se u memoriji pravi mesto (make a room) za novi proces je kompleksniji. Prvo, swapper će obaviti swap-out operaciju procesa na bazi prioriteta, vremena provedenog u memoriji (memory-residence time), i nice vrednosti. Mada swapper obavlja swap-out operaciju nekog procesa da bi napravio mesto za proces za koji treba obaviti swap-in operaciju, može da se obavi i swap-out procesa, čijim se prebacivanjem ne obezbeđuje dovoljno memorije za dolazeće procese. Na primer, ako swapper pokušava da obavi swap-in operaciju za proces koji okupira 1MB, sistem nema dovoljno slobodne memorije, a procesi u memoriji su mali, tada je alternativa da se obavi swap-out operacija za grupu procesa, dok se ne obavi zahtev.

Drugo, ako swapper spava zato što nije našao dovoljno memorije da obavi swap-in procesa, on će ponovo pretraživanje swap prostora za izbor novog procesa za swap-in, iako je u prethodnoj iteraciji izabrao jedan. Razlog je što drugi procesi u swap prostoru mogu da se probude i da budu bolji za swap-in operaciju, nego prethodno izabrani

proces. Neka mala šansa da prethodno izabrani proces bude ponovo izabran ipak postoji. U nekim implementacijama, swapper pokušava da obavi swap-out više manjih procesa da bi napravio mesto za veći proces, a tek onda radi izbor za swap-in.

time	Proc A	Proc B	Proc C	Proc D	Proc E
0	0	0	swap out	swap out	swap out
	runs		0	0	0
1	1	1	1	1	1
		runs			
2	2	2	2	2	2
	swap out	swap out	swap in	swap in	
	0	0	0	0	
			runs		
3	1	1	1	1	3
				runs	
4	2	2	2	2	4
	swap in		swap out	swap out	swap in
	0		0	0	0
					runs
5	1	3	1	1	1
	runs				
6	2	4	2	2	2
	swap out	swap in	swap in		swap out
	0	0	0		0
		runs			

Slika 9.8. *Primer za swapper*

time	Proc A	Proc B	Proc C	Proc D	Proc E
0	0	0	swap out	nice 25	swap out
	runs		0	swap out	0
				0	
1	1	1	1	1	1
		runs			
2	2	2	2	2	2
	swap out	swap out	swap in	swap in	
	0	0	0	0	
			runs		
3	1	1	1	**1**	3
				swap out	swap in
				0	runs
					0
4	2	2	2	1	1
	swap in		swap out		
	0		0		
	runs				
5	1	3	1	2	2
		swap in			swap out
		0			0
		runs			
6	2	1	2	**3**	1
	swap out			**swap in**	
	0			**0**	
				runs	

Slika 10.9. *Primer za swap out, neposredno posle swap-in*

Treće, ako swapper izabere proces u stanju 3 (ready to run) za swap-out operaciju, moguće je da je to proces koji se nije nikada izvršavao jer je prethodno bio prebačen na swap prostor. Na slici 9.9 prikazan je jedan sličan slučaj, kada kernel obavlja swap-in

operaciju za proces D u drugoj sekundi, bira proces C na izvršavanje. Zatim obavlja swap-out operaciju za proces D u trećoj sekundi da bi favorizovao proces E, zato što proces D ima visoku nice vrednost, bez obzira što se proces D nikada nije izvršavao. Takav raspored događaja (trashing) nije poželjan.

Pomenućemo i glavnu opasnost. Ako proces pokušava da obavi swap-out procesa, a nema prostora u swap prostoru, može se dogoditi zastoj (deadlock), ako su ispunjeni sledeći uslovi:

- svi procesi u memoriji su uspavani,
- svi spremni procesi (ready to run) su imali swap-out operaciju,
- nema mesta na swap prostoru za nove procese,
- nema mesta ni u memoriji za nadolazeće procese.

Sve ovo se izbegava preko DP (Demand Paging) implementacije.

9.2. Straničenje po zahtevu (Demand Paging, DP)

Računari čije su memorijske arhitekture bazirane na stranicama i čiji CPU imaju restartabilne instrukcije (ako se u delu instrukcije dogodi greška u straničenju PF, a CPU restartuje instrukciju nakon rutine za obradu greške u straničenju tj. PF handlera) mogu podržavati kernelski DP algoritam, koji obavlja swap operacija za stranice (pages) između memorije i swap uređaja. DP sistemi oslobađaju procese limitacija koje nameće veličina fizičke memorije. Na primer mašina može sadržati 1 ili 2MB fizičke memorije a da izvršava procese veličine 4 ili 5MB. Kernel ispoljava ograničenje u pogledu virtualne veličine procesa, zavisno od veličine virtuelne memorije koju sistem podržava. Na primer, ako proces ne može da stane u fizičku memoriju, kernel će puniti dinamički delove procesa u memoriju i izvršavati proces bez obzira što neki njegovi delovi nisu u memoriji nego na swap uređaju. DP tehnika je transparentno za korisnika, osim za maksimalnu virtuelnu veličinu procesa.

Princip lokalnosti

Procesi teže da izvršavaju instrukcije u malim porcijama njihovog text prostora kao što su programske petlje i često pozivani potprogrami. Memorijske reference teže da se grupišu u male podskupove prostora podataka za proces, a to je poznato pod nazivom princip lokalnosti. Denning je uveo pojam radnog skupa procesa (working set procesa), koji obuhvata skup svih stranica kojima je proces pristupao u poslednjih n memorijskih referenci, pri čemu se n zove prozor radnog skupa (working set window). Kako je trenutni radni skup stranica samo deo ukupne memorije procesa, to znači da više procesa mogu simultano biti u memoriji nego na swaping sistemu, što povećava performanse. Kada se pojavi adresa za stranicu koja nije u memoriji, dogodi se greška u straničenju PF (page

fault), rutina za obradu greške u straničenju (PF handler) reguliše grešku PF, nakon čega se stranica dovodi u memoriju, a radni skup procesa se obnavlja.

Slika 9.10 pokazuje sekvencu referenci na stranice koju bi proces mogao napraviti, a takođe prikazuju se radni skupovi procesa za različite prozore radnog skupa. Za zamenu stranica u fizičkoj memoriji primenjuje se LRU algoritam. Kako se proces izvršava, menja se njegov radni skup, zavisno od memorijskih referenci, pri čemu širi prozor obuhvata veći radni skup, što znači da neće praviti PF greške tako često. Nije praktično koristiti jednostavni model radnog skupa (pure working set model), jer je preskupo po performanse da se prati poredak svih referenci za stranice. Umesto toga, sistem aproksimira model radnog skupa, postavljajući čuveni R bit (reference bit), svaki put kada se sistem obrati stranici, a potom se periodično obavlja memorijska analiza sistema (sampling). Ako je stranica bila skoro referencirana ona će biti u radnom skupu, u protivnom ona je zastarela (ages) i može se po potrebi izbaciti iz memorije.

	working sets/window sizes			
page references	2	3	4	5
24	24	24	24	24
15	15 24	15 24	15 24	15 24
18	18 15	18 15 24	18 15 24	18 15 24
23	23 18	23 18 15	23 18 15	23 18 15
24	24 23	24 23 18	23 18 15 24	23 18 15 24
17	17 24	17 24 23	17 24 23 18	17 24 23 18 15
18	18 17	18 17 24	....	...
24	24 18	....	....	...
18	18 24	....	....	...
17	17 18	....	....	...
17	17	....	....	...
15	15 17	15 17 18	15 17 18 24	...
24	24 15	24 15 17	....	...
17	17 24	....	....	...
24	24 17	....	....	...
18	18 24	18 24 17	....	...

Slika 10.10. *Primer radnog skupa u zavisnosti od veličine procesa*

Kada proces pristupa stranici koja nije iz radnog skupa, takva referenca će izazvati grešku u straničenju po pitanju validnosti (validity PF). Kernel suspenduje, tj. uspavljuje proces sve dok se stranica ne dovede u memoriju. Kada se stranica dovede u memoriju,

proces restartuje instrukciju koja je izazvala grešku PF, tako da implementacija straničenja u DP tehnici ima 2 dela: swap-out operacije za retko korišćene stranica i opsluživanje greške u straničenju PF, kada se obavlja swap-in operacija za stranicu koja nedostaje (faulted page). Ovo uglavnom važi za sve operativne sisteme koji koriste straničenje, a sada ćemo opisati specifičnosti za UNIX System V.

Strukture podataka za DP tehniku (Data structure for demand paging)

Kernel sadrži četiri glavne strukture podataka za podršku DP tehnike :

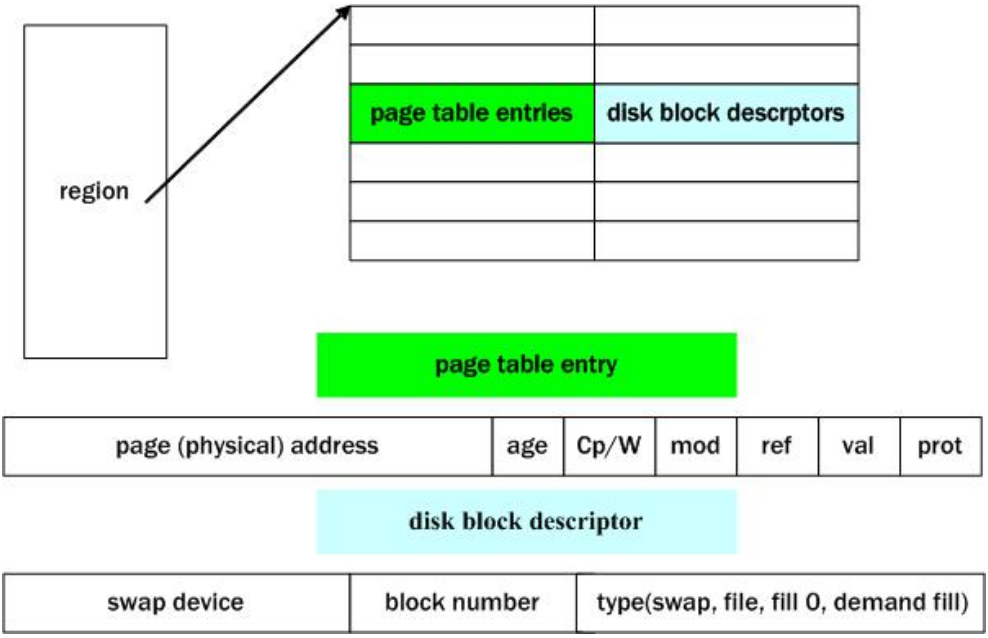
- tabela stranica (page table)
- disk blok deskriptori,
- tabela okvira fizičke memorije (page frame data tabela), koja se na UNIX operativnom sistemu naziva pfdata i
- tabela korišćenja swap prostora (swap-use table).

Kernel alocira prostor za pfdata tabelu samo jednom prilikom podizanja UNIX sistema, a preostale tri strukture se kreiraju dinamički. Strukture podataka za DP su prikazane na slici 9.11.

Podsetimo se da region sadrži tabelu stranica za pristup fizičkoj memoriji. Svaki ulaz u tabeli stranica sadrži:

- fizičku adresu stranice
- zaštitne bite koji pokazuju da li proces može da čita, piše ili izvršava stranicu
- statusna polja ili bitove koja podržavaju DP tehniku:
    - bit validnosti (valid bit)
    - bit za referencu (reference bit)
    - bit za modifikaciju (modify bit)
    - cow bit (copy on write bit)
    - bitovi starosti (age bit)

Kernel postavlja bit validnosti kada je sadržina stranice legalna, ali ako je bit validnosti nula, to ne znači da je referenca na stranicu obavezno ilegalna. Bit za referencu pokazuje da li je proces nedavno referencirao stranicu, a bit za modifikaciju ukazuje da je proces modifikovao sadržaj stranice. COW bit (copy on write) se koristi u fork sistemskom pozivu i pokazuje da kernel mora da formira novu kopiju stranice, kada proces pokušava da je promeni tj. upiše nešto u nju. Na kraju, kernel manipuliše sa bitovima starosti (age bits) koji pokazuju koliko dugo je stranica bila član radnog skupa procesa. Kernel manipuliše bitom validnosti, cow bitom, i bitovima starosti, a hardver setuje bit za referencu i bit za modifikaciju u ulazu tabele. stranice.



Slika 10.11. *Stukture podataka za DP tehniku*

Svaki ulaz u tabeli stranica ima pridruženi disk blok deskriptor, koji opisuju položaj tj. adresu kopije virtuelne stranice, koja se nalazi na swap prostoru . Procesi koji dele region pristupaju zajedničkim ulazima u tabeli stranica i disk blok deskriptorima.

Kopija virtuelne stranice može se nalaziti:

[1] Na swap prostoru. Ako je stranica na swap prostoru, disk blok deskriptor sadrži dva broja:

- broj uređaja (device number) koji identifikuje swap prostor na disku
- broj bloka (blok number) na swap prostoru koji sadrži stranicu.

[2] U izvršnoj datoteci. Ako je stranica u izvršnoj datoteci, disk blok deskriptor sadrži broj logičkog bloka iz datoteke, koji sadrži stranicu, a kernel može lako da konvertuje taj broj u disk adresu.

[3] Nije na swap prostoru. Disk blok deskriptor može da sadrži i dva specijalna uslova za vreme sistemskog poziva exec:

- zahteva se punjenje stranice (demand fill)
- zahteva se upis nula u stranicu (demand zero)

Pfdata tabela opisuje fizičku memoriju svake stranice, a tabela se indeksira po broju stranice. Polja u ulazu pfdata tabele su:

[1] Polje koje opisuje stanje stranice (page state). Polje pokazuje da li je stranica na swap prostoru ili u izvršnoj datoteci, koji je DMA kanal zadužen za stranicu, da li stranica može ponovo da se dodeli (reassigned) itd.

[2] Broj referenci, opisuje broj procesa koji referencira tu stranicu. Broj referenci je preciznije jednak broju validnih ulaza tabele stranica koji referenciraju tu stranicu. Mada se ne očekuje, broj referenci se može razlikovati od broja procesa koji dele tu stranicu, što ćemo objasniti na primeru algoritma fork u DP sistemu.

[3] Logički uređaj (swap ili sistem datoteka FS) i blok u njemu koji sadrži stranicu

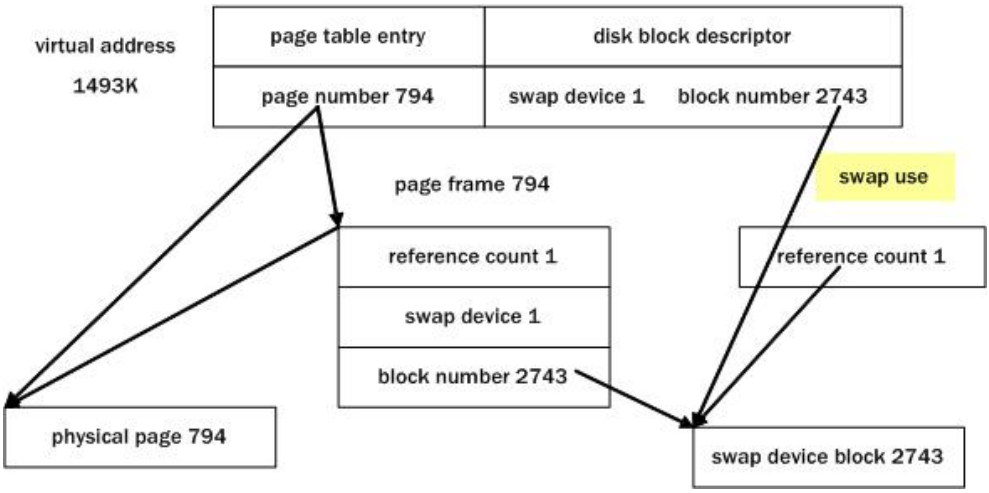
[4] Pokazivači na druge ulaze u pfdata tabeli, na listu slobodnih stranica i na red čekanja (hash queue) stranica

Kernel povezuje ulaze pfdata tabele u dve liste: na listu slobodnih stranica i na hash listu, što je analogno povezanim listama kod baferskog keša. Lista slobodnih stranica (to ne moraju da budu prazne stranice, već kao kod baferskog keša, to su trenutno slobodne stranice za korišćenje) je lista stranica koje su raspoložive za dodeljivanje i slobodna lista olakšava pretraživanje. Kernel alocira nove stranice iz slobodne liste na bazi LRU algoritma. Kernel takođe obavlja hash funkciju za ulaze pfdata tabele, na bazi njegove swap adrese i tako kernel može lako da locira stranicu ako je u memoriji. Da bi dodelio fizičku stranicu regionu, kernel uklanja stranicu sa početka slobodne liste, ažurira oba broja za swap adresu (device i blok number) i postavlja stranicu u odgovarajuću hash-queue listu.

Tabela korišćenja swap prostora (Swap-use table) sadrži ulaze za svaku stranicu na swap uređaju. Ulaz ove tabele, sadrži broj referenci o tome koliko ulaza iz tabele stranica ukazuju na tu stranicu u swap prostoru.

Na slici 9.12 prikazana je veza između ulaza iz tabele stranica, disk blok deskriptora, ulaza pfdata tabele i ulaza swap-use tabele.

Virtuelna adresa 1493K procesa se mapira na ulaz iz tabele stranica, koji pokazuje na fizičku stranicu 794; disk blok deskriptor za taj ulaz iz tabele stranica ukazuje da kopija stranice postoji na swap uređaju broj 1 na disk-bloku 2743. Ulaz pfdata tabele za fizičku stranicu broj 794, takođe ukazuje da kopija te stranice postoji na swap uređaju broj 1 i na disk-bloku 2743 i da je njegov broj referenci jednak jedan. Videćemo kasnije, zašto se swap adresa duplira i u pfdata tabeli i u disk blok deskriptoru. Broj referenci u swap-use tabeli za virtuelnu stranicu je jedan, što znači da samo jedan ulaz iz tabele stranica ukazuje na swap kopiju.

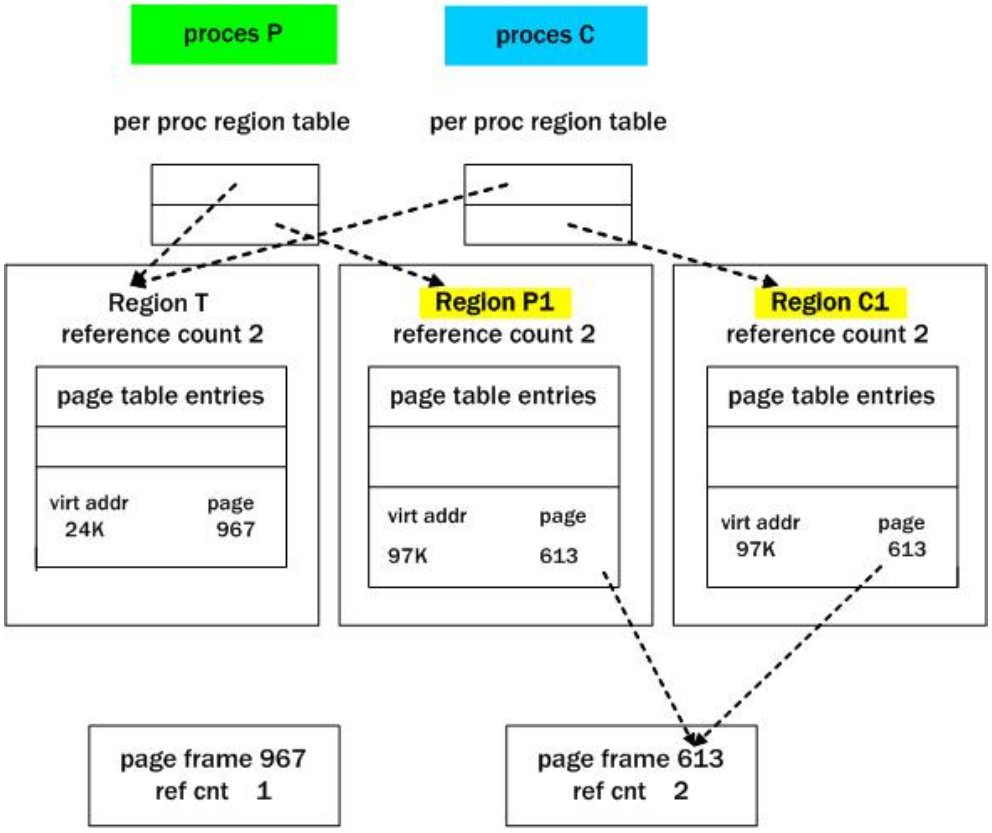


Slika 9.12. *Primer veze između tabele stranica, disk blok deskriptora, pfdata tabele i swap-use tabele*

Mehanizam fork u sistemima sa straničenjem

Kernel duplira svaki region roditeljskog procesa za vreme sistemkog poziva fork i dodeljuje ga procesu detetu. Tradicionalno, za swaping sisteme, kernel pravi fizičku kopiju roditeljskog adresnog prostora, a to je nekorisno jer procesi zovu sistemski poziv exec odmah posle sistemskog poziva fork, a sistemski poziv exec oslobođa svu memoriju od procesa deteta, pa traži novu memoriju. Na sistemu UNIX system V sa straničenjem, kernel izbegava kopiranje stranica preko manipulacije sa region tabelama, ulazima tabele stranica i ulazima pfdata tabele. Kernel inkrementira broj referenci za deljive regione (text region). Za privatne regione (data i stack), kernel će inicirati novi ulaz u region tabeli, i novu tabelu stranica, ali tada se ispituje svaki ulaz roditeljske tabele stranica: ako je stranica validna, inkrementira se broj referenci u ulazu pfdata tabele, koji ukazuje na broj procesa, koji dele stranicu preko različitih regiona (za razliku od broja procesa koji dele stranicu preko deljenja čitavog regiona). Ako stranica postoji na swapu, inkrementira se broj referenci za stranicu u swap-use tabeli.

Na ovaj način stranici može da se pristupa iz oba regiona (roditelj, dete), sve dok neko od procesa ne poželi upis u nju. Kernel tada kopira stranicu tako da svaki region dobija svoju privatnu kopiju. Da bi se ovo dešavalo, kernel postavlja cow (copy on write) bit za svaku stranicu privatnog regiona roditelja i deteta za vreme fork sistemkog poziva. Ako drugi proces zatraži upis, dogodiće se greška u zaštiti (protection fault) za tu stranicu i kernel će napraviti kopiju stranice za proces koji je tražio upis. U suštini, fizičko kopiranje stranica se odlaže do realne potrebe.



Slika 9.13. *fork u sistemu sa DP tehnikom*

Slika 9.13 daje izgled struktura kada proces obavlja sistemski poziv fork i stvara svoje dete proces. Procesi dele pristup tabeli stranica za deljivi (shared) tekst region T, tako da je njegov broj referenci za region jednak dva, a broj referenci svih stranica tog regiona T, u pfdata tabeli biće jednak jedan. Kernel alocira novi data-region za proces dete C1, koji je kopija data-regiona P1 procesa roditelja. Ulazi u tabelama stranica za oba regiona su identični, a prikazan je ulaz sa VA=97K. Ovaj ulaz tabele stranica ukazuje na pfdata ulaz broj 613, čiji je broj referenci jednak dva, što govori da dva različita regiona ukazuju na istu fizičku stranicu.

BSD vfork mehanizam

Kod BSD UNIX sistema, sistemski poziv fork pravi fizičke kopije stranica roditelja, a da bi ubrzao performanse izbegavanjem tog kopiranja, BSD je napravio sistemski poziv vfork, koji pretpostavlja da će dete odmah iza vfork obaviti sistemski poziv exec. Sistemki

poziv vfork ne kopira stranice za dete, tako da je vfork brži nego System V fork, ali se dete proces u vorku izvršava u fizičkom adresnom prostoru kao roditeljski proces, osim ako proces dete ne obavi sistemski poziv exec ili exit. U takvoj situaciji proces dete može prepisati delove roditeljskog regiona podataka ili stek regiona. To je opasna situacija, ako programer koristi sistemski poziv vfork neoprezno.

Na primer, analizirajmo sledeći program. Posle sistemskog poziva vfork, proces dete ne izvrši sistemski poziv exec, nego resetuje promenljive global i local, pa obavlja sistemski poziv exit. Sa sistemskim pozivom wait, roditelj može suspendovati svoje aktivnosti dok proces dete (child) ne obavi sistemski poziv exit. Kada roditelj nastavi izvršavanje, nalazi da mu promenljive nisu iste kao što su bile pre sistemskog poziva vfork, a mnogo je opasnije ako proces dete poremeti roditeljski stek.

```
int global;
main()
{
  int local;
  if(vfork() ==0)
  {
    /* child */
    global = 2; /*write parent data space*/
    local = 3;  /*write parent stack*/
    _exit();
  }
  printf("global %d local %d\n, global, local)
}
```

Sistemski poziv exec u sistemima sa straničenjem

Kada proces pozove sistemski poziv exec, kernel bi trebalo da učita celu izvršnu datoteku iz sistema datoteka u memoriju. Na sistemu sa DP tehnikom, izvršna datoteka može biti prevelika da cela stane u raspoloživu fizičku memoriju. Međutim, zahvaljući DP tehnici, to nije problem, kernel ne obavlja neku reorganizaciju memorije, on jednostavno poziva punjenje datoteke, a memorija se dodelju prema potrebi. Prvo se dodeljuje tabela stranica i disk blok desktiptori se postavljaju da pokazuju na izvršnu datoteku, ulazi tabele stranica za non-bss područje podataka se markiraju kao "demand fill", a ulazi tabele stranica za bss podatke se markiraju kao "demand zero". Preko read algoritma, proces će napraviti grešku u straničenju PF, prilikom čitanja svake stranice. Kada rutina za grešku u straničenju (PageFault handler), detektuje da je status stranice "demand fill", to ima značenje da stranicu neposredno treba napuniti sa delom izvršne datoteke, a ako je stranica markirana sa "demand zero" statusom, to znači da sadržina stranice mora da se popuni nulama. U opisu rutine za greške u straničenju (validity fault handler), biće objašnjeno kako se to radi. Ako proces ne može ceo da stane u memoriju, page-stealer proces periodično obavlja swap-out stranica drugih procesa u swap prostor, da bi se oslobodila memorija za nadolazeće stranice procesa koji se izvršava.

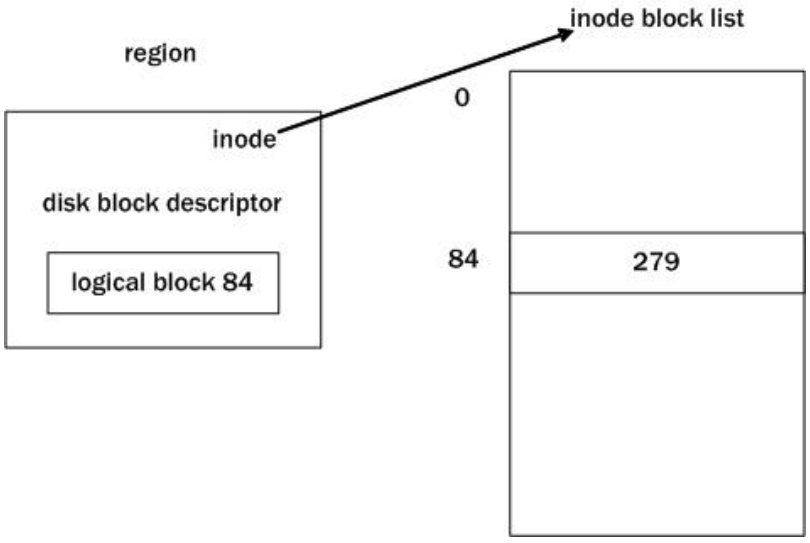
Postoje ozbiljni nedostaci ove šeme.

- proces za svaku stranicu izaziva grešku u straničenju PF, bez obzira da li će proces ikada da koristi tu stranicu ili ne.
- page-stealer proces može obaviti swap-out za stranice pre nego što se one izvrše, što izaziva dve extra swap operacije po stranici. Na primer ako proces prerano zatraži stranicu, kao rezultat istrukacija poziva procedure ili skoka (call or jump).

Da bi učinio sistemski poziv exec efikasnijim, kernel može da obavlja DP tehniku direktno iz izvršne datoteke pod uslovom da su podaci i instrukcije propisno dodeljeni na stranice, na šta ukazuje magični broj (magic number) u zaglavlju datoteke, mada korišćenje standarnih algoritama kroz keš bafer nije baš pogodno za DP tehniku.

Za straničenje direktno iz izvršne datoteke, kernel pronalazi sve brojeve disk blokova koji pripadaju izvršnoj datoteci, kada se obavlja sistemki poziv exec i priključuje ih (atach) u listu u in-core inode strukturi. Kada se postavljaju ulazi tabele stranica za takvu izvršnu datoteku, kernel označava disk blok deskriptore sa logičkim blok brojem bloka na disku koji se sadrži u stranici, a rutina za grešku u straničenju (validity fault handler) posle toga koristi ove informacije, kada puni stranicu iz datoteke.

Na slici 9.14 imamo jedno takvo tipično aranžiranje, gde disk blok deskriptor ukazuje na stranicu koja predstavlja 84-ti blok datoteke. Kernel prati pokazivač iz regiona u inode strukturu, u kojoj traži odgovarujuće blokove diska.



Slika 9.14*. Ubrzavanje exec sistemskog poziva u DP tehnici*

PS proces (Page stealer process)

PS proces (page stealer) možemo prevesti kao kradljivac stranica, a u daljem tekstu PS proces. PS proces obavlja swap-out stranica procesa koje nisu više deo radnog skupa procesa, tj onih stranica kojima proces nije dugo pristupao. Kernel kreira PS proces za vreme sistemske inicijalizacije i poziva ga sve dok je UNIX aktovan, i to uvek kada je malo raspoloživih slobodnih stranica. PS proces ispituje svaki aktivni, nezaključani (unlocked) region, preskačući zaključane (locked)regione, i inkrementira polje starosti (age) svih validnih stranica. Kernel zaklučava region, kada se dogodi greška u straničenu PF u tom regionu, tako PS proces ne može ukrasti stranicu iz takvog regiona.

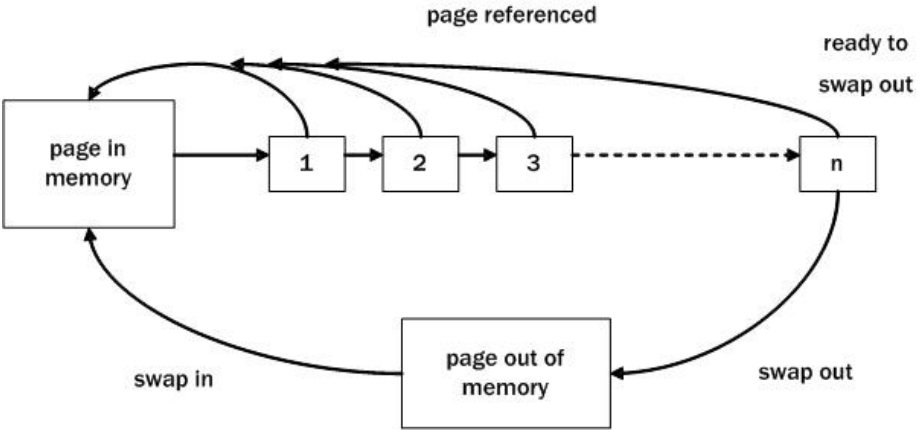
Postoje dva stanja za stranice u memoriji:

[1] stranica nije zastarela, stranica je nedavno imala pristup i nije još poželjna za swap-out operaciju

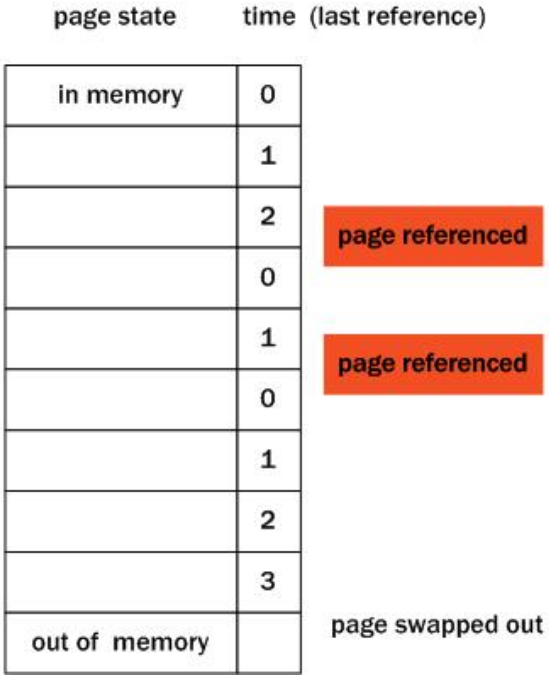
[2] stranica je zastarela, nedavno nije imala pristup i poželjna je za swap-out operaciju, koja će omogućiti da ta fizička stranica bude raspoloživa drugim procesima

Prvo stanje je kada proces nedavno pristupao stranici, pa je ona u njegovom radnom skupu. Neke mašine setuju bit za referencu kada se proces obraća stranici, a ako to ne podržava hardver, to se može simulirati softverski. PS proces gasi bit za referencu za takve stranice, ali pamti koliko je ispitivanja prošlo od poslednje reference za tu stranicu. Prvo stanje se sastoji od više nižih stanja, koje odgovaraju broju prolazaka PS procesa, pre nego što stranica postane poželjna za swap-out operaciju. Kada broj probije probojna (threshold) vrednost za broj prolazaka, kernel stavlja stranicu u drugo stanje, tj. proglašava da je zastarela (ready to swapped). Maksimalni period tj. broj prolazaka za koji stranica postaje stara (age), zavisi od implementacije, a ograničena je brojem raspoloživih stranica. Princip funkcionisanja PS procesa prikazan je na slici 9.15.

Na slici 10.16 prikazana je interakcija između procesa koji pristupaju stranici i ispitivanja koje obavlja proces PS. Stranica se nalazi u memoriji i slika prikazuje broj ispitivanja PS uzmeđu memorijskih referenci. Proces referencira stranicu posle drugog ispitivanja, obarajući njene bite starosti (age) na nulu. U nastavku, proces ponovo referencira stranicu posle još jednog PS ispitivanja tj. prolaska. Na kraju, proces PS ispituje stranicu tri puta bez promene njenog bita za referencu i obavlja swap-out za tu stranicu tj. stavlja je u listu za swap-out.



Slika 9.15. *Princip funkcionisanja PS procesa*



Slika 9.16. *Primer PS procesa sa 3 prolaska*

Ako dva ili više procesa dele isti region, oni ažuriraju bit za referencu za iste ulaze u tabeli stranica. Stranice mogu biti deo radnog skupa za više procesa, ali to ne zanima proces PS. Ako je stranica deo radnog skupa bilo kog procesa ona ostaje u memoriji, ali ako ne pripada ni jednom radnom skupu, poželjna je za izbacivanje. Nije bitno da li region ima više stranica u memoriji od drugih, proces PS ne pokušava da obavi swap-out za jednak broj stranica iz svih aktivnih regiona.

Kernel će probuditi proces PS, kada količina slobodnih stranica padne ispod donje granice (low-water mark). U tom slučaju proces PS obavlja swap-out za sve zastarele stranice sve dok količina slobodne memorije ne dostigne gornju granicu (high-water mark). Korišćenje donje i gornje granice smanjuje mogućnost za čuveni trashing efekat . Na primer ako bi kernel koristio samo jednu granicu, PS proces će prazniti stranice iznad potrebnih vrednosti, pa je moguće da se one opet vrate u memoriju i da opet naprave trashing efekat. Zato se swap-out obavlja do gornje granice (high-water mark) i ne obavlja više sve dok broj slobodnih stranica ne padne ispod donje granice, tako da PS se ne poziva često, odnosno ne obavlja swap-out često. Administratori mogu da podese donju i gornju granicu (low i high water mark).

Ako proces PS odluči da obavi swap-out stranice, analizira se da li kopija stranica postoji na swap uređaju. Postoje tri moguće situacije:

[1] ako nema kopije stranice na swap uređaju, kernel bira stranicu za swap-out: proces PS postavlja stranicu na listu stranica za koje će se obaviti swap-out i nastavlja da radi, čime je swap-out logički završen. Kada lista stranica za koje treba da se obavi swap-out dostigne maksimum, kernel upisuje te stranice na swap uređaj, naravno u većim grupama, a ne pojedinačno

[2] ako kopija stranice već postoji na swap uređaju, a nijedan proces joj nije modifikovao sadržaj, što znači da je bit za modifikaciju jednak nuli, kernel tada briše bit validnosti za stranicu, dekrementira broj referenci u pfdata ulazu i postavlja ulaz te stranice na slobodnu listu za buduće alokcije

[3] ako je kopija stranice na swap uređaju, ali je memorijski sadžaj stranice modifikovan, kernel bira stranicu za swap-out, kao u slučaju pod 1, a oslobađa njen prostor na swapu, zato što će se upisati na nekom drugom mestu

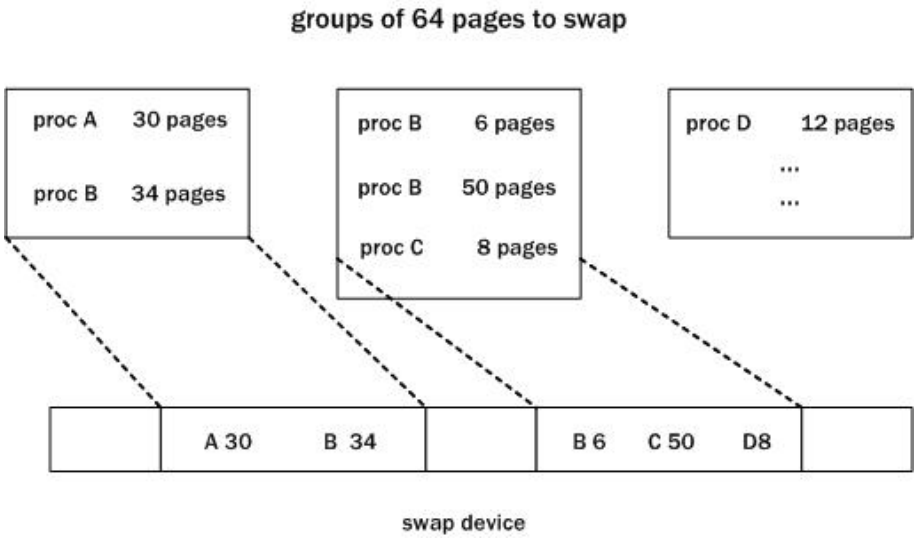
Proces PS kopira stranicu na swap prostor samo u slučaju 1 i 3. Objasnimo razliku između ovih slučajeva. Pretpostavimo da je stranica na swap uređaju i da je ušla u memoriju nakon greške u straničenu PF. Eventualno, proces PS može kasnije odlučiti da ponovo obavi swap-out iste stranice. Ako nijedan proces nije modifikovao stranicu, memorijska kopija je identična disk kopiji pa nema potrebe da se ponovo upisuje na swap. Ako je proces modifikovao stranicu, kernel mora da upiše stranicu na swap, ali to se ne radi sa tehnikom prepisivanja (overwrite), nego se oslobađa prostor na swapu koji je stranica prethodno zauzimala, a stranica će se ponovo upisati na drugom mestu na swap prostoru, tako da se swap prostor održava kontinualnim, što daje bolje performanse.

Proces PS puni listu stanica za koje će obaviti swap-out operacija. Moguće je da te stranice potiču iz različitih regiona, a PS proces ih prebacuje na swap, samo kada je lista puna. Sve stranice procesa ne moraju biti prebače na swap: neke stranice nemaju dovoljno starosti (age) i to je glavna razlika između paging i swapping sistema kod kojih se ceo proces prebacuje na swap. Ako swap uređaj nema dovoljno mesta, tada kernel obavlja swap-out za jednu po jednu stranicu u vremenu. Uopšte, postoji veća fragmentacija na swap prostoru kod paging sistema u odnosu na swaping sisteme, zato što kernel prebacuje manji broj stranica.

Kada kernel upisuje stranicu na swap, on isključuje bit validnosti u ulazu tabele stranice i dekrementira broj referenci u pfdata tabeli. Ako broj referenci padne na nulu, tada se plasira ta fizička stranica na slobodnu listu, ali ona ipak ostaje još uvek tu (u kešu), sve dok se ne dodeli nekom drugom procesu. Ako broj referenci nije nula, to znači da drugi procesi dele stranicu, (na primer kao rezultat sistemskog poziva fork), ali kernel će svakako obaviti swap-out za stranicu, zato što je zastarela. Na kraju, kernel alocira swap prostor, upisuje swap adresu u disk blok deskriptoru i inkrementira broj referenci u swap-use tabeli (use-count) za stranicu. Ako proces napravi grešku u straničenu PF dok je stranica još uvek na slobodnoj listi (free listi) a nije dodeljena drugome, kernel će obnoviti stranicu u memoriji umesto da je obnavlja sa swap prostora.

Primer za PS proces

Na primeru datom na slici 9.17, PS odlučuje da obavi swap-out za 30, 40, 50 i 20 stranica od procesa A, B, C i D, a zatim upisuje po 64 stranice u jednoj disk operaciji.



Slika 9.17. *Primer za swap-out koji obavlja PS proces*

Slika prikazuje sekvencu swap-out operacija koje se dešavaju kada PS ispituje redom procese A, B, C i D. PS dodeljuje prostor na swap uređaju za 64 stanice, pa zatim upisuje 30 stranica procesa A i 34 stranice procesa B. Zatim alocira prostor za još 64 stranice na swap uređaju i tu upisuje 6 stranica procesa B, 50 stranica procesa C i 8 stranica za proces D. Dva područja na swap uređaju u toku dve swap-out operacije ne moraju da budu kontinualna. PS proces drži ostalih 12 stranica procesa u listi za swap-out, ali nema upisa na swap dok se ne popuni lista od 64 stranice. Kada procesi uzimaju svoje stranice sa swap uređaja, a to se dešava kada se dogodi greška u straničenu PF ili kada obave sistemki poziv exit, ažurira se lista slobodnih blokova na swap uređaju.

Obavimo rezime. Postoje dve faze u swap-out proceduri za stranice. Prvo, proces PS nalazi stranice podesne za swap-out i plasira ih u listu stranica za swap-out. Drugo, kernel kopira stranicu na swap prostor kada mu je povoljno, ukida se valid bit, dekrementira se broj referenci u pfdata tabeli i plasira se pfdata ulaz na kraj slobodne liste, ako broj referenci padne na nulu. Sadržaj fizičke stranice u memoriji je validan sve dok se stranica ne dodeli nekom drugom.

9.3. Algoritmi za greške u straničenju (Page faults)

Sistem sa DP tehnikom može obuhvatiti dva tipa grešaka u straničenu PF:

- greške zbog validnosti (validity faults)
- greške zbog zaštite (protection faults)

Kako rutine za obradu grešaka u straničenju (fault handlers) mogu da čitaju stranicu sa diska u memoriju i za to vreme obično spavaju za vreme disk I/O operacije, rutine za obradu grešaka u straničenju su jedini izuzeci kod prekidnih rutina (interrupt handler), zato što prekidne rutine po pravilu ne mogu na spavanje. Međutim rutina za obradu grešaka u straničenju spava u kontekstu procesa koji je izazvao greški PF, rutina za obradu grešaka u straničenju je povezana sa tekućim procesom, u suštini zajedno spavaju.

Rutina za obradu greške zbog validnosti u straničenju

Ako proces pokušava da priđe stranici čiji bit validosti nije postavljen, dogodiće se greška zbog validnosti (validity fault) i kernel će prozvati rutinu VFH ((Validity fault handler)). Bit validnosti nije postavljen za stranice izvan virtuelnog adresnog prostora procesa ili za stranice koje jesu u virtuelnom prostoru procesa, ali još nemaju dodeljenu fizičku stranicu.

```
algorithm vfault /*handler for validity faults*/

input: address where proces faulted
output: none
```

```
{
  find region, PT entry, disk block descrtiptor corresponding
    to faulted address, lock region;
  if(address outside virtual address space)
  {
    send signal(SIGSEGV: segementation violation) to process;
    goto out;
  }
  if(address now valid) /* process may have slept above*/
  {
    goto out;
  }
  if(page in cache)
  {
    remove page from cache;
    adjust page table entry;
    while(page contents not valid)
      /*another process faulted first*/
      sleep(event contents become valid)
  }
  else /* page not in cache*/
  {
    assign new page to region;
    put new page in cache, update pfdata entry;
    if(page not previously loaded and page "demand zero")
      clear assigned page to 0;
    else
    {
      read virtual page from swap device or exec file;
      sleep(event I/O done);
    }
    awaken processes (event page contents valid);
  }
  set page valid bit;
  clear page modify bit, page age;
  recalculate proces priority;
  out: unlock region;
}
```

Hardver obaveštava kernel koja je virtuelna adresa izazvala gresku zbog validnosti. Kernel nalazi ulaz tabele stranica i disk blok deskriptor za tu stranicu. Kernel zaključava region koji sadrži taj ulaz tabele stranica, tako da spreči stanje trke (race condition), koji bi mogao da se napravi ako proces PS pokuša da obavi swap-out za stranicu. Ako disk blok deskriptor nema zapis za tu stranicu, pokušana memorijska referenca je pogrešna (invalid) i kernel šalje "segmentation violation" signal procesu koji je poslao tu referencu. Ako je memorijska referenca bila legalna, kernel alocira stranicu memorije u koju će se

pročitati sadržina swap uređaja ili sadržina iz izvršne datoteke.

Stanja stranice koja je napravila PF

Stranica koja je izazvala grešku zbog validnosti PF, može da se nađe u jednom od pet stanja:

[1] nalazi se samo u swap uređaju a ne i u memoriji

[2] nalazi se u slobodnoj listi (free page list) u memoriji

[3] nalazi se u izvršnoj datoteci

[4] označena je kao "demand zero"

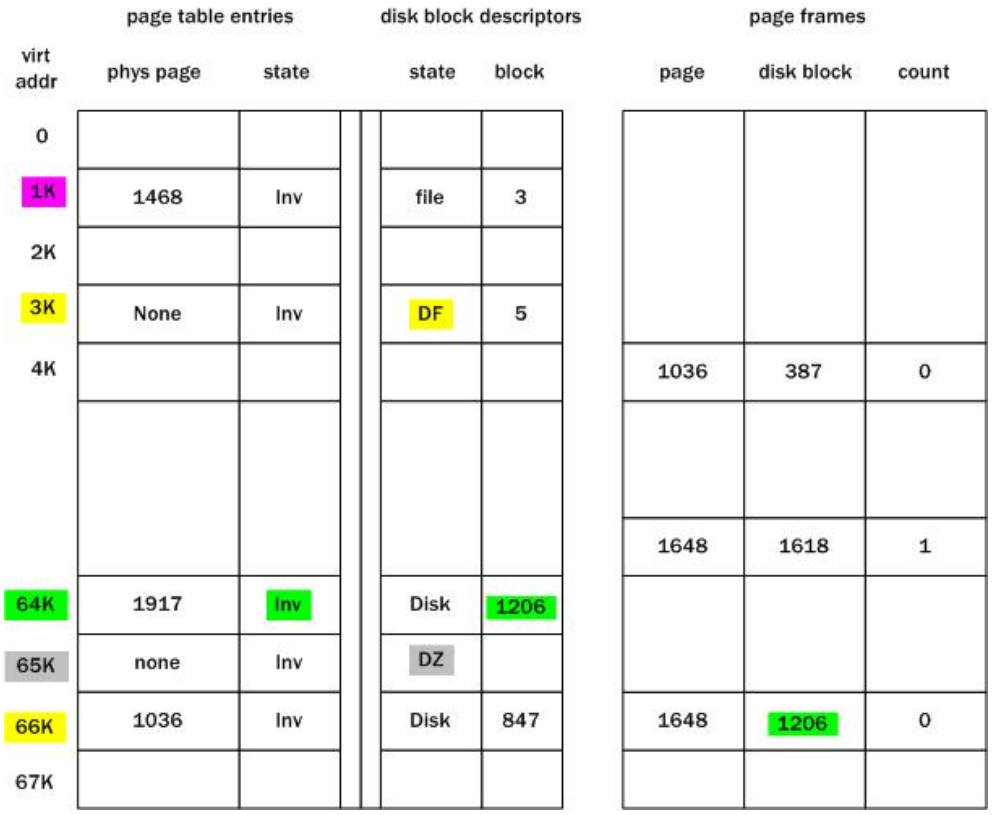
[5] označena je kao "demand fill"

Slučaj 1

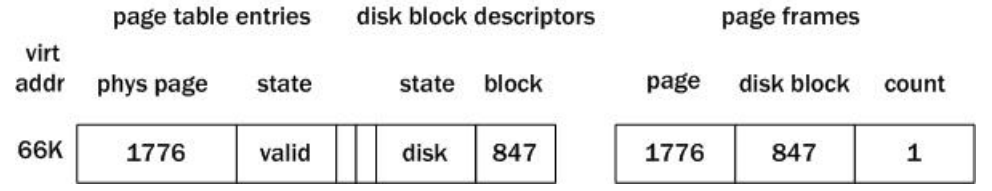
U slučaju 1, stranica je samo na swap uređaju a nije u memoriji. Možda je nekada bila u memoriji ali je proces PS prebacio na swap prostor. Iz disk blok deskriptora, kernel nalazi swap uređaj i broj bloka u njemu, a treba da proveri da li je ta stranica još uvek u memoriji, tj. u kešu stranica (page cache). Ako stranica nije u kešu stranica, kernel ažurira ulaz tabele stranica tako da ukazuje na stranicu koju treba pročitati, nalazi slobodnu fizičku stranicu, podešava ulaz pfdata tabele na odgovarajuću hash listu, kako bi ubrzali operaciju za VHF, a potom se čita stranica sa swap uređaja. Proces koji je napravio grešku u straničenu spava sve dok se ne završi I/O, kada kernel budi sve procese koji su čekali na tu stranicu.

Uzmimo za primer ulaz iz tabele stranica za virtuelnu adresu 66K na slici 9.18.

Ako proces izazove grešku zbog validnosti (validity fault), VFH ispituje disk blok desktriptor i detektuje da se stranica nalazi na swap uređaju u bloku 847. Virtuelna adresa je legalna, pa VHF pretražuje keš stranica (page cache) tj. pfdata tabelu i tamo ne nalazi nijedan ulaz za disk blok 847. Ako nema kopije u fizičkoj memoriji ili u kešu stranica, VFH mora pročitati stranicu sa diska (swap). Kernel dodeljuje fizički stranicu 1776, čita sadržaj sa swap uređaja u nju, ažurira ulaz tabele stranica da ukazuje na stranicu 1776. Na kraju, kernel ažurira disk blok deskriptor da ukazuje na stranicu koja još uvek ostaje na swap uređaju i ažurira ulaz pfdata tabele za stranicu 1776 da pokazuje na blok 847 na swap uređaju, koji sadrži duplikat virtuelne stranice, kao na slici 9.19.



Slika 9.18. *Slučajevi za VFH*

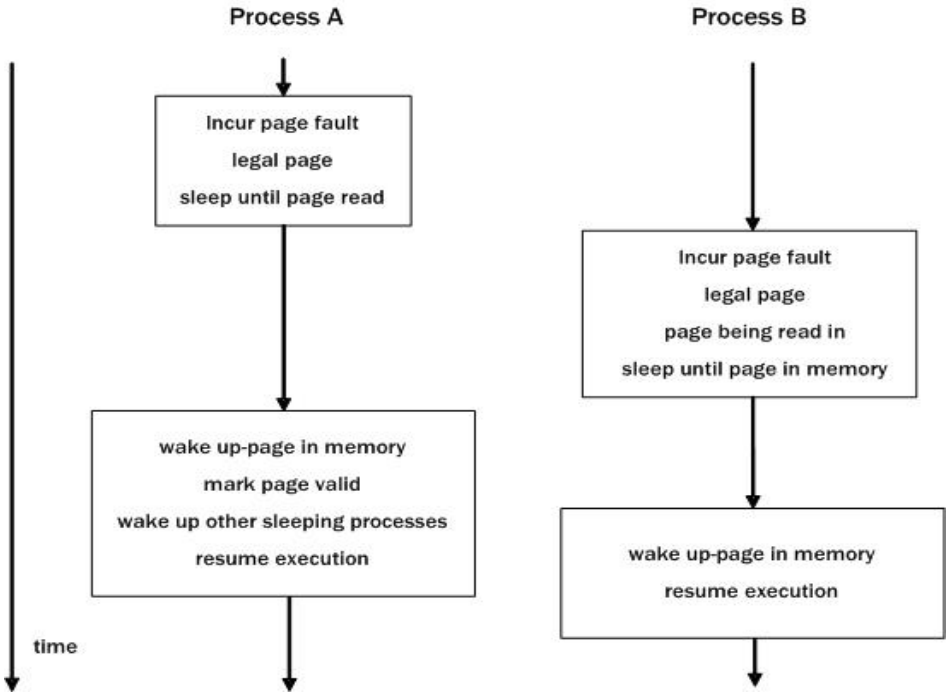


Slika 9.19. *Slučaj 1, situacija nakon dodele stranice*

Slučaj 2

Kernel ne obavlja uvek I/O operaciju kada se dogodi greška zbog validnosti (validity fault), bez obzira što disk blok deskriptor ukazuje na stranicu koja je na swap prostoru. Postoji slučaj kada VFH nalazi stranicu u kešu stranica, a detektuje je po bloku zadatom u disk blok desktriptoru. Kernel ponovo dodeljuje ulaz tabele stranica da ukazuje na nađenu stranicu, inkrementira njen broj referenci, uklanja je iz slobodne liste. Ovaj slučaj imamo kada proces pristupa virtuelnoj adrsesi 64K. Tražeći po kešu stranica, po disk bloku 1206, kernel nalazi da fizička stranica (page frame) 1861 ima disk blok desktriptor 1206. Kernel postavlja ulaz tabele stranica za virtuelnu adresu 64K da ukazuje na fizičku stranicu 1861 i postavlja bit validosti. Zato se disk blok upisuje u obe tabele: u tabelu stranica i u pfdata tabelu.

Takođe, VFH ne mora da čita stranicu u memoriju, ako je drugi proces već napravio grešku u straničenu za istu fizičku stranicu, ali je nije još pročitao. VFH nalazi region koji sadrži ulaz tabele stranica koji je zaključan drugom instancom od VFH. On tada spava dok prethodna instanca ne pročita stranicu i oglasi je validnom, kao na slici 9.20.



Slika 9.20. *Slučaj kada 2 procesa izazovu grešku za istu stranicu*

Slučaj 3

Ako kopija stranice ne postoji na swap prostoru nego u originalnoj izvršnoj datoteci, kernel čita stranicu iz originalne datoteke. VFH ispituje disk blok deskriptor, nalazi logički blok datoteke koji sadrži stranicu i nalazi inode strukturu pridruženu sa ulazom u region tabeli. Koristi se logički blok kao pokazivač na polje disk blokova datoteke čije su adrese priključene (attach) u inode strukturi za vreme sistemskog poziva exec. Poznavajući broj disk bloka preko inode strukture, stranica se čita sa diska u memoriju. Na primer disk blok deskriptor za virtuelne adresu 1K prikazuje da ta stranica nalazi u logičkom bloku broj tri u izvršnoj datoteci.

Slučajevi 4 i 5

Ako proces izazove grešku u straničenju za stranicu označenu kao "demand fill" ili "demand zero", kernel alocira slobodnu fizičku stranicu u memoriji i ažurira odgovarujući ulaz u tabeli stranica. Na zahtev "demand zero" upisuju se nule u stranicu. Na kraju se brišu flagovi "demand zero" ili "demand fill". Stranica se proglašava za validnu i njen sadržaj se ne nalazi dupliran na swap uređaju ili u sistemu datoteka. Ovo će se dogoditi za virtuelne adrese 3K i 65K. Nijedan proces nije pristupao ovim stranicama nakon što se obavio sistemski poziv exec.

VFH se završava postavljanjem bita validnosti i brisanjem bita modifikacije, a zatim se rekalkuliše prioritet procesa, pošto proces može biti uspavan u VFH, a to je kernelski prioritet, što bi mu dalo nepravedno veliki prioritet kada se vrati u korisnički mod. Na kraju, pri povratku u korisnički mod, proverava da li je bilo nekih signala dok se obrađivala greška u straničenju.

Rutina za obradu greške zbog zaštite u straničenju

Druga vrsta memorijske greške u straničenju koju proces može izazvati je greška zbog zaštite (protection fault), koja znači da proces pristupa validnoj stranici, ali zaštitini biti pridruženi stranici ne dozvoljavaju pristup. Proces može takođe izazvati ovu vrstu greške (protection fault), kada pokušava upis u stranicu koja ima postavljen cow bit (copy-on-write), koji se obično za vreme sistemskog poziva fork. Kernel mora odrediti da li su prava zabranjena zato što se upis zahteva u stranici sa cow bitom, što je relativno legalno ili se nešto zaista ilegalno dogodilo.

Hardver će obezbediti za PFH virtuelnu adresu gde se dogodila greška, a PFH (Protection fault handler) će odrediti odgovarajući region i ulaz u PT.

```
algorithm pfault /*handler for protection faults*/

input: address where proces faulted
output: none

{
```

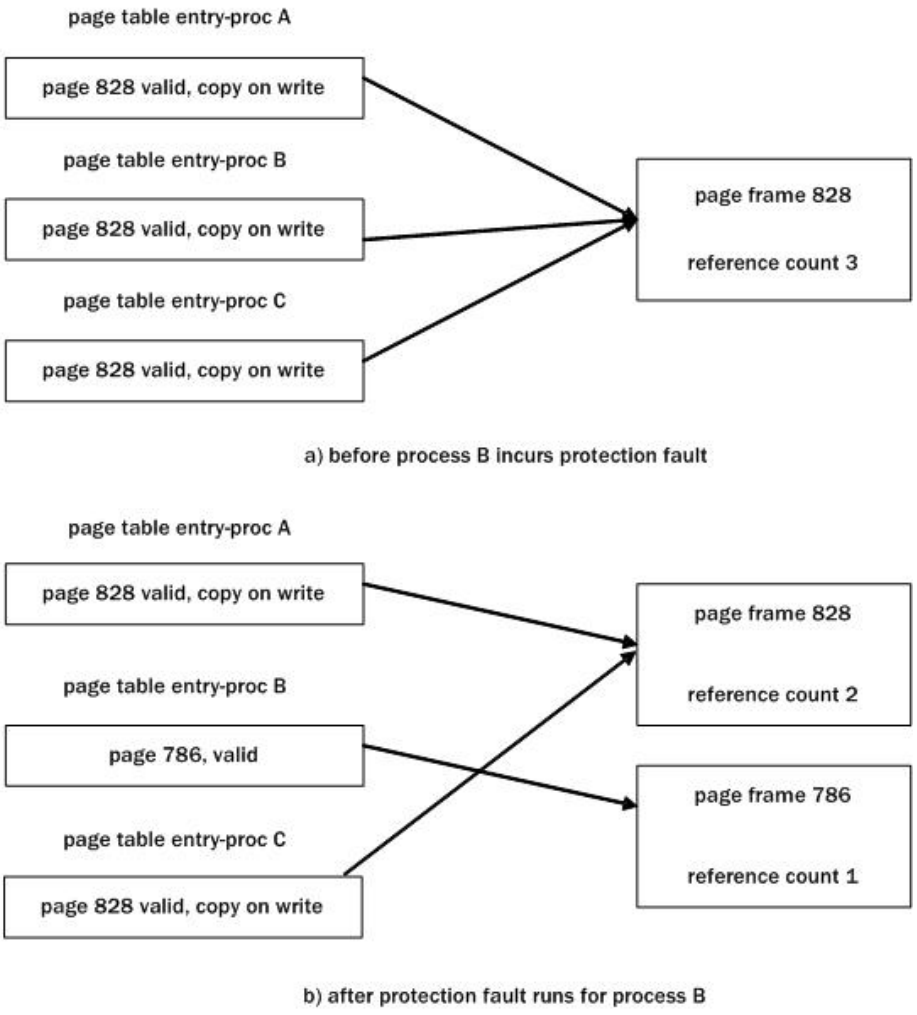
```
  find region, PT entry, disk block descrtiptor corresponding
    to faulted address, lock region;
  if(page not valid in memory) goto out;
  if(copy on write bit not set) /* real program error-signal*/
    goto out;
  if(page frame reference count > 1)
  {
    allocate a new physical page;
    copy contents of old page to new page;
    decrement old page frame RC;
    update page table entry to point to new physical page;
  }
  else /* steal page, since nobady else is using it */
  {
    if(copy of page exists on swap device)
      free space on swap device, break page association;
    if(page is on page hash queue)
      remove from hash queue;
  }
  set modify bit, clear copy on write bit in PT entry;
  recalculate proces priority;
  check for signals;
  out: unlock region;
}
```

PFH će zaključati region, tako da PS ne može ukrasti stranicu dok nju obrađuje PFH. Ako PFH odredi da je grešku izazvao cow (copy-on-write) bit, kernel alocira novu stranicu i kopira sadržaj stare stranice u nju. Drugi procesi zadržaće reference na staru cow stranicu. Posle kopiranja stranice i ažuriranja ulaza tabele stranica sa novom stranicom, kernel dekrementira broj referenci pfdata ulaza za staru stranicu.

Primer za PFH

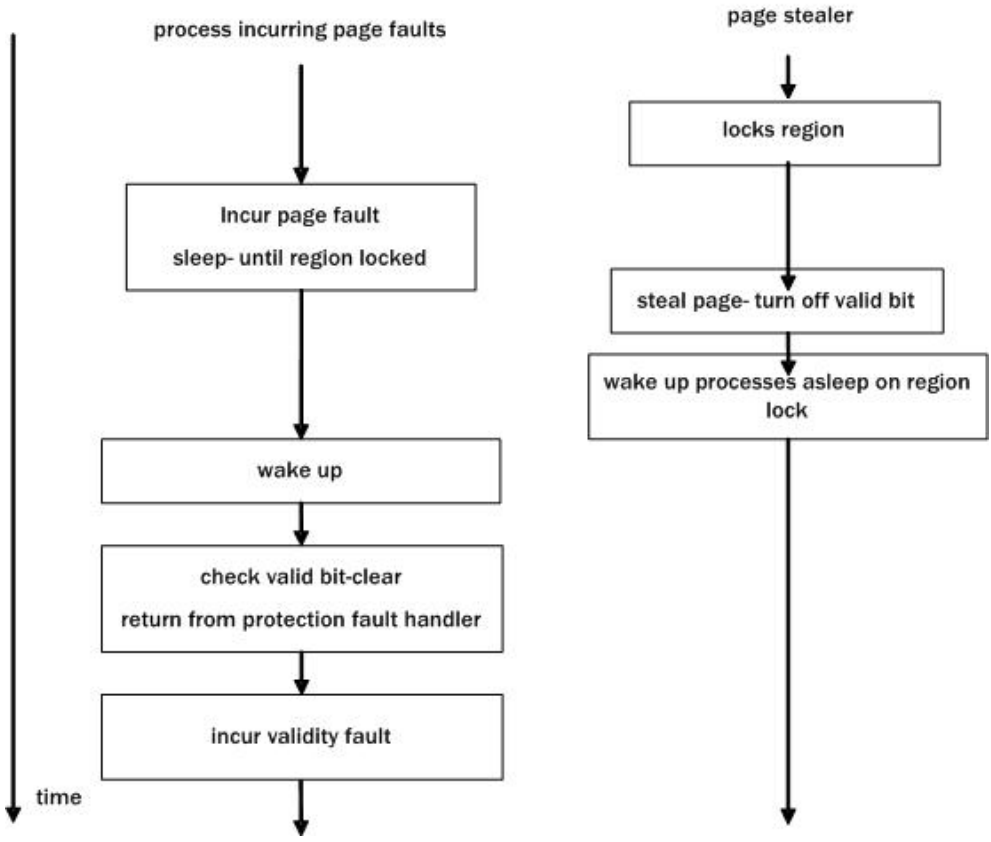
Slika 9.21 pokazuje scenario: tri procesa dele fizičku stranicu 828. Proces B pokušava upis u stranicu, ali to izaziva grešku zbog zaštite (protection fault) zbog cow (copy-on-write) bita. PFH alocira novu stranicu 786, kopira sadržaj stare stranice 828, dekrementira broj referenci za staru stranicu 828 i ažurira ulaz tabele stranica za proces B da ukazuje na novu stranicu 768.

Ako je cow (copy-on-write) bit postavljen ali nijedan drugi proces ne koristi tu stranicu, kernel će dozvoliti procesu da obavi upis u fizičku stranicu. Ukida se cow bit, razdvaja se stranica od svoje disk kopije (ako postoji) jer drugi procesi mogu da dele disk kopiju. Zatim se ukljanja pfdata ulaz iz hash liste, zato što nova kopija virtuelne stranice nije na swap prostoru. Zatim, se dekrementira broj referenci za swap-use, i ako broj referenci padne na nulu, oslobađa se swap prostor.



Slika 9.21. *CoW bit sa tri procesa*

Ako je ulaz tabele stranica invalidan, a postavljen je i cow (copy-on-write) bit, da bi izazvao grešku zbog zaštite (protection fault), pretpostavimo da sistem prvo proverava validnost pa tek onda zaštitne bite. U svakom slučaju PFH mora proveriti validnost, zato što on zaključava region, a PS proces može u međuvremenu da obavi swap-out stranice, tako što joj prvo ukine validnost. Ako je stranica invalidna, PFH završava trenutno a poziva se VFH, ali proces će izazvati i protection fault. Slika 9.22 prikazuje sekvencu događaja.



Slika 9.22. *Obe rutine u akciji PFH i VHF*

Kada PFH završi exekuciju, za novu stranicu se postavljaju bit modifikacije i drugi zaštitni bitovi, a briše se cow (copy-on-write) bit, rekalkuliše se prioritet procesa i proverava se da li je bilo signala.

Hibridni sistemi

Mada DP tretira sistem mnogo fleksibilnije od swaping sistema, moguće su situacije kada proces PS i rutina VFH izazivaju trash-efekat zvog nedostatka memorije. Ako je zbir radnih skupova svih procesa veća od fizičke memorije, VHF obično spava, zato što ne može da alocira stranice za procese. Proces PS ne može da krade stranice dovoljno brzo, zato što su sve stranice u radnom skupu, tako je performanse sistema drastično padaju, jer kernel provodi mnogo vremena u disk IO zahtevima.

System V kernel izvršava i swapping i paging algoritme da bi sprečio trashing efekat. Kada kernel više ne može da dodeljuje stranice za procese, budi se proces swapper i gura ceo novi proces u stanje 5 "ready to run but swappped". Više procesa mogu biti u takvom stanju simultano. Swapper radi swap-out operaciju za cele procese sve dok slobodna memorija ne pređe gornju granicu (high-level mark). Svaki proces koji se ceo prebaci na swap prostor, postaje proces u stanju 5 "ready to run but swappped". Iza toga se radi normalna DP tehnika, a postoje razne hibridne kombinacije.

10

UNIX ulazno/izlazni sistem

10.1. Uvod u I/O sistem

I/O podsistem dozvoljava procesu da komunicira sa periferijskim uređajima, kao što su diskovi, trake, terminali, štampači, mrežni uređaji a kernelski moduli koji kontrolišu uređaje nazivaju se device drajveri. Obično postoji jedan na jedan (one-to-one) korespodencija između drajvera i tipa uređaja: UNIX može sadržati jedan disk drajver da kontroliše sve diskove u sistemu, jedan terminal drajver da kontroliše sve terminale. Ako imate uređaje iste klase ali od različitih proizvođača drajveri za njih mogu biti različiti, zato što uređaji imaju različiti komandni skup. Na drugoj strani drajveri mogu biti prilično univerzalni, da podržavaju različite uređaje koje kontrolišu.

UNIX podržava "softverske uređaje" koji nemaju pridruženi fizički uređaj. Na primer, memorija se tretira kao I/O uređaj, iako to nije I/O uređaj. Drajveri upisuju statističke informacije u kernelske strukture podataka, kao i zapise za praćenje (trace) koji su pogodni za ispravljanje grešaka u programima (debbuging).

Ova lekcija opisuje interfejs između procesa, I/O sistema i drajvera. Objasnićemo generalnu strukturu i funkciju drajvera, a detaljno ćemo objasniti disk i terminal drajvere. Zaključićemo sa novom metodom za realizaciju drajvera koja se naziva streams mehanizam.

Drajverski interfejs

UNIX sistem sadrži dva tipa uređaja, blok uređaje i karakter uređaje (raw). Blok uređaji, kao što su diskovi, optički uređaji i trake izgledaju kao uređaji sa slučajnim-direktnim pristupom (random-access storage device). Karakter uređaji su: terminali, štampači, mrežni uređaji. Blok uređaji, pored svog blok interfejsa mogu takođe imati interfejs za karakter uređaje.

Korisnički interfejs za sve uređaje prolazi kroz sistem datoteka. Svaki uređaj ima posebnu datoteku u sistemu datoteka sa imenom koje podseća na uređaj. Specijalne datoteke za uređaje imaju svoju inode strukturu i svoj FCB u UNIX direktorijumu. Ove datoteke se razlikuju od običnih datoteka po tipu datoteke, one su blok specijalne ili karakter specijalne. Ako uređaj ima obe vrste interfejsa, tada ima obe drajverske datoteke (blok i karakter). Sistemski pozivi koji važe za obične datoteke imaju posebno značenje kada rade sa specijalnim datotekama. Takođe, sistemski poziv ioctl, dozvoljava procesima da kontrolišu-podešavaju karakter uređaje, a takav poziv se ne primenjuje za regularne datoteke. Međutim, svaki drajver ne mora da podržava interfejs sistemskih poziva, kao što je drajver za praćenje (trace driver), koji čita zapise koje su kernelskim strukturama ostavili drugi drajveri.

Sistemska konfiguracija

Sistemska konfiguracija je procedura pomoću koje administratori specificiraju parametre koji su zavisni od instalacije. Neki parametri specificiraju veličine kernelskih tabela, kao što su tabela procesa PT, inode tabela, tabela datoteka FT, broj bafera alociranih za gomilu bafera u baferskom kešu (buffer cache pool). Drugi parametri specificiraju konfiguraciju uređaja, objašnjavajući kernelu koji uređaji su uključeni u instalaciju i njihove adrese. Na primer, konfiguracija može specificirati koja je mrežna kartica instalirana u server.

Postoje tri faze u kojima se specificira konfiguracija uređaja.

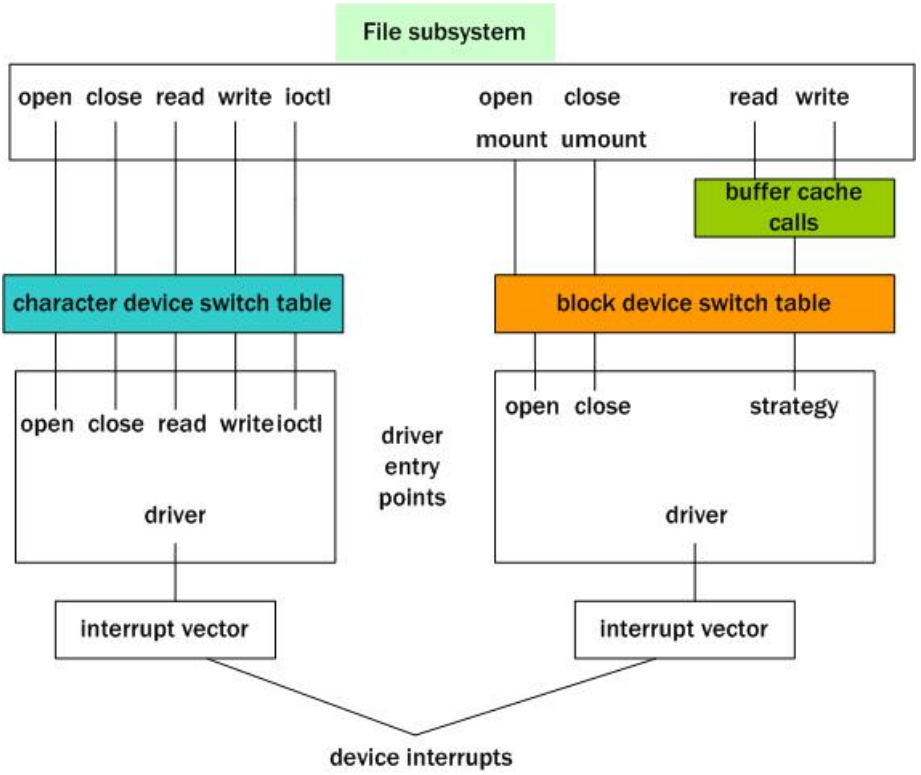
- administratori mogu da ubace konfiguracione podatke koji se prevode i linkuju kada je formira kernelski kôd. Konfiguracioni podaci se tipično specificiraju u jednostavnom formatu i konfiguracioni program konvertuje ih u datoteku pogodnu za kompajliranje.
- administratori mogu da ubace konfiguracione podatke nakon što se UNIX podigne, a kernel ažurira svoje interne konfiguracione tabele dinamički.
- postoje "self-identifying" uređaji koji omogućavaju kernelu da prepozna koji je uređaj instaliran. U tom slučaju, kernel čita hardverske prekidače-informacije i automatski konfiguriše sam sebe

Prekidačka (switch) tabela

Kernelski drajver interfejs je opisan preko dve prekidačke tabele: prekidačke tabele za blok uređaje (blok device switch table) i prekidačke tabele za karakter uređaje (character device switch table), kao na slici 10.1.

- Svaki tip uređaja ima ulaze u tabeli, koji direktno upućuju kernel na odgovarajući drajver, u saglasnosti sa sistemskim pozivom.
- Sistemski pozivi open i close za drajverske (device datoteke) prolaze kroz dve prekidačke (switch) tabele, saglasno tipu datoteka.
- Sistemski pozivi read, write i ioctl za karakter specijalne datoteke prolaze kroz odgovarajuće procedure u karakter prekidačkoj tabeli.
- Sistemski pozivi mount i umount pozivaju device open i device close procedure za blok uređaje.
- Sistemski pozivi read i write za blok uređaje pozivaju algoritme za baferski keš, koji poziva strategijsku (strategy) proceduru za uređaj. Neki drajveri pozivaju strategijsku proceduru interno, iz svojih read i write procedura.

Drajvere ćemo objasniti detaljno.



Slika 10.1. *Prekidačke tabela za drajvere*

Hardver u drajverskom interfejsu sastoji se od kontrolnih I/O instrukcija za manipulaciju uređaja i prekidnih vektora: kad se dogodi prekid, sistem određuje uređaj koji je izazvao prekid i poziva odgovarajuću prekidnu rutinu (interrupt handler). Softverski uređaji, kao što je kernelski profiler-drajver nemaju hardverski interfejs, ali druge prekidne rutine mogu pozvati softverske prekidne rutine (software interrupt handler). Na primer prekidna rutina za časovnik, može pozvati kernelski profiler-drajver.

Administrator kreira specijalne drajverske datoteke sa mknod komandom, čiji ulaz predstavljaju prapametri: 1. tip datoteke (block, character), 2. glavni i pomoćni broj (major i minor) itd. Komanda mknod poziva sistemski poziv mknod, koji kreira drajversku datoteku. Na primer komandna linija:

```
mknod /dev/tty13 c 2 13
```

gde je /dev/tty13 je ime karakter datoteke, c predstavlja karakter datoteku sa brojem 2 kao glavnim (major) brojem i sa brojem 13 kao sporednim (minor) brojem. Glavni broj ukazuje na tip uređaja koji odgovara ulazu u blok ili karakter prekidačkoj tabeli, a sporedni broj ukazuje na jedinicu uređaja. Ako proces otvara blok datoteku /dev/dsk1 sa

glavnim brojem nula, kernel poziva rutinu gdopen na ulazu broj nula u blok karakter prekidačkoj tabeli kao na slici 10.2.

block device switch table			
entry	**open**	**close**	**strategy**
0	gdopen	gdclose	gdstrategy
1	gtopen	gtclose	gtstrategy

Slika 10.2. *Blok prekidačka tabela*

Ako proces čita karakter datoteku /dev/mem koja ima glavni broj tri, kernel poziva rutinu mmread na trećem ulazu karakter prekidačke tabele, kao na slici 10.3. Rutina nulldev je prazna rutina, koristi se kada nema potrebe za posebnim drajverskim funkcijama.

character device switch table					
entry	**Open**	**close**	**read**	**write**	**loctl**
0	conopen	conclose	conread	conwrite	conioctl
1	dzbopen	dzbclose	dzbread	dzbwrite	dzbioctl
2	syopen	nulldev	syread	sywrite	syioctl
3	nulldev	nulldev	mmread	mmwrite	nodev
4	gdopen	gdclose	gdread	gdwite	nodev
5	gdopen	gdclose	gdread	gdwite	nodev

Slika 10.3. *Karakter prekidačka tabela*

Mnogi perferijski uređaji imaju isti glavni broj, a sporedni broj ih razlikuje između sebe. Drajverske (device) datoteke ostaju stalno na direktorijumu /dev, ne moraju da se kreiraju svaki put kad se UNIX podiže, one se menjaju, a nove se dodaju samo ako se promeni hardverska konfiguracija ili se instaliraju novi uređaji.

Interfejs sistemskih poziva i drajverski interfejs

Ova sekcija opisuje interfejs između kernela i drajverskih programa. Za sistemske pozive se koriste deskriptori datoteka (file-descriptors), kernel prati pokazivače iz UFDT tabele, pokazivače iz tabele datoteka FT i inode strukturu, u kojoj se ispituju tip datoteke i pristup blok ili karakter prekidačkoj tabeli. Izvlače se glavni i sporedni brojevi iz inode strukture, glavni broj se koristi kao indeks u odgovarajućoj prekidačkoj tabeli i poziva se odgovarajuća drajverska funkcija u saglasnosti sa parametrima sistemskog poziva kojima se dodaju glavni i pomoćni broj, koji su dobijeni iz inode strukture. Važna razlika između sistemkih poziva za drajverske i regularne datoteke je da se inode struktura specijalne datoteke ne zaključava (lock) kada kernel izvršava drajverski program. Drajveri često spavaju, čekajući na hardverku konekciju ili na transfer podataka, tako da kernel ne može da odredi koliko će drajver da spava. Ako bi se inode struktura zaključavala, mnogi procesi bi takođe bili uspavani, jer je prvi proces uspavan u drajverskom kodu, a samim tim bi bila zaključana inode struktura za drajver.

Drajver interpretira parametre sistemskog poziva kao parametre za uređaj. Drajver održava strukture podataka koje opisuju stanje uređaja koga drajver kontroliše. Drajverske funkcije i prekidne rutine izvršavaju se u saglasnosti sa stanjem drajvera i akcijama koje se obavljaju (data input, data output). Opisaćemo detaljno svaki interfejs.

Sistemski poziv open za drajvere

Kernel prati istu proceduru za otvaranje uređaja kao za otvaranje regularnih datoteka, alocira inode strukturu u memoriji, inkrementira broj referenci, dodeljuje ulaz u tabeli datoteka FT i ulaz u UFDT tabelu u koji upisuje deskriptor datoteke (user file deskriptor). Kernel, vraća deskriptor datoteke procesu, tako da otvoreni uređaj liči na otvorenu datoteku. Međutim, kernel poziva i specifičnu (device-specific) open proceduru pre povratka u korisnički mod.

```
algorithm open /* for device drivers*/

input:  pathname
        openmode
output: file descriptor

{
  convert pathame to inode, increment RC, allocate entry in FT,
  user fd, kao kod open regularne datoteke;
  get major, minor number from inode;
  save context (algorithm setjmp) in case of
    long jump from driver;
  if(block device)
  {
    use major number as index to block device switch table;
    call driver open procedure for index;
```

```
    pass minor number, open modes;
  }
  else
  {
    use major number as index to character device switch table;
    call driver open procedure for index;
    pass minor number, open modes;
  }
  if(open fails in driver) decrement file table, inode count;
}
```

Za blok uređaje, poziva se open procedura kodirana u blok prekidačkoj tabeli, a za karakter uređaje poziva se open procedura u karakter prekidačkoj tabeli. Ako uređaj ima obe varijante (block & character) kernel će pozvati open proceduru iz one tabele zavisno od vrste datoteke koju je korisnik otvorio. Dve open procedure mogu biti čak i identične, zavisno od drajvera.

Specifična (Device-specific) open procedura uspostavlja konekciju između procesa i otvorenog uređaja, a takođe inicijalizuje privatnu drajversku struktruru podataka. Za terminal, na primer, open procedura može uspavati proces sve dok mašina ne detektuje hardverski (carrier) signal koji ukazuje da korisnik pokušava login proceduru. Zatim se inicijalizuju drajverske strukture podataka u saglasnosti sa setovanjem terminala (na primer terminal baud rate). Za softverske uređaje kao što je sistemska memorija, open procedura nema šta da inicijalizuje.

Ako proces mora da spava zbog nekog spoljašnjeg razloga kada se otvara uređaj, moguće je da se događaj, koji treba da probudi proces, nikada ne desi. Na primer, ako nema korisnika da obavi login proceduru na terminalu, getty proces koji je otvorio terminal može spavati dugo vremena. Kernel mora da bude stanju da probudi proces i da prekine njegov sistemski poziv open tako što pošalje procesu signal. Tada se mora resetovati inode struktura, ulaz u tabeli datoteka, deskriptor datoteke fd za koga je proces dodelio promenjivu ali nije dobio fd, jer je sistemski poziv open otkazao. Međutim, kernel čuva kontekst procesa korišćenjem algoritma setjmp, pre nego što uđe u specifičnu (device-specific) open proceduru. Potom, ako se proces probudi zbog signala, kernel obnavlja kontekst procesa u stanje pre ulaska u drajver, preko longjmp algoritma i oslobađa strukture podataka alocirane za sistemski poziv open. Drajver može da uhvati signal (signal caching), pa da očisti svoju privatnu strukturu podataka, ako je potrebno. Kernel ponovo podešava strukture podataka kada drajver naiđe na grešku; na primer sistemski poziv open otkazuje za uređaj koji nije konfigurisan.

Procesi mogu specificirati različite opcije da kvalifikuju uređaj za device-open proceduru. Najčešća opcija je "no delay", koja znači da proces ne ide na spavanje za vreme open procedure ako uređaj nije spreman. U tom slučaju sistemski poziv open se završava gotovo trenutno i tada proces ne zna da li je kontakt sa hardverom uspostavljen ili ne. Otvaranje uređaja sa "no delay" opcijom takođe ima uticaj na semantiku sistemkog poziva read što ćemo objasniti.

Ako je uređaj otvoren više puta, kernel manipuliše sa UFDT tabelom, inode strukturom i ulazima u tabeli datoteka FT, kao kod običnih datoteka, s tim što se za svaki open-uređaja poziva i specifična (device-specific) open procedura. Drajver može brojati koliko puta je uređaj otvoren, a ako je broj neodgovarajući, sistemski poziv open može otkazati. Na primer, ako to ima smisla, može da se dozvoli da više procesa otvore terminal za upis, tako da korisnici mogu da razmenuju poruke. Ali nema smisla dozvoliti da više procesa otvori štampač za simultani upis, pošto oni mogu da prepisuju jedan drugome podatke.

Sistemski poziv close za drajvere

Proces zatvara već otvoreni I/O uređaj, preko sistemskog poziva close. Kernel poziva specifičnu (device-specific) close proceduru samo ako je to poslednji sistemski poziv close za taj uređaj, što znači da nema više procesa koji drže taj uređaj otvorenim, zato što ta specifična close procedura završava hardversku konekciju, što znači da mora da čeka da nijedan proces pristupa uređaju. Zato što kernel poziva specifičnu open proceduru za vreme svakog sistemskog poziva open, a specifičnu close proceduru samo jedanom, drajver ne može uvek biti siguran koliko procesa koristi uređaj. Drajveri lako mogu da izgube kontrolu ako nisu pažljivo napisani: ako spavaju u close proceduri, a drugi proces otvara uređaj pre nego što se specifična close procedura kompletira, uređaj može postati neupotrebljiv, ako kombinacija open i close procedura dovede drajver u nedefinisano stanje.

Algoritam za zatvaranje uređaja, close, sličan je algoritmu za close regularne datoteke.

```
algorithm close /* for device*/

input:  file descriptor
output: none

{
  do regular close algorithm;
  if(file table RC not 0) go to finish;
  if(there is another open file and its major,
     minor numbers are the same as device being closed)
    go to finish; /*not last close after all*/
  if(character device)
  {
    use major number as index to character device switch table;
    call driver close procedure: parameter minor number;
  }
  if(block device)
  {
    if(device mounted) goto finish;
    write device blocks in buffer cache to device;
    use major number as index to block device switch table;
    call driver close procedure: parameter minor number;
```

```
      invalidate device blocks still in buffer cache;
    }
    finish:
    release inode;
  }
```

Međutim, pre nego što kernel otpusti inode strukturu uređaja, obaviće se operacije specifične za drajverske datoteke:

[1] Kernel pretražuje tabelu datoteka FT da bi bio siguran da nijedan drugi proces ne drži uređaj otvorenim. Nije dovoljno bazirati se na broju referenci u ulazu tabele datoteka, zato što više procesa mogu pristupati uređaju preko različitih ulaza u tabeli datoteka FT. Nije pouzdano bazirati se na broju referenci u incore inode strukturi, zato što više drajverskih datoteka mogu da specificiraju isti uređaj. Na primer, rezultat sledeće ls –l komande, prikazuje dve različite karakter datoteke (po imenu) ali njihovi glavni i sporedni brojevi (major, minor) su isti pa obe ukazuju na isti uređaj

```
crw--w--w- 1  root  vis     9,1  Aug 6  1984          /dev/tty01
crw--w--w- 1  root  unix    9,1  May 3  15:02        /dev/fty01
```

Ako proces otvara dve datoteke nezavisno, on pristupa različitim inode strukturama, ali koristi isti uređaj.

[2] Za karakter uređaje, kernel poziva specifičnu close proceduru i vraća se u korisnički mod. Za blok uređaje, kernel traži tabelu aktivnih sistema datoteka MT (mount table), da bi bio siguran da uređaj ne sadrži aktivirani sistem datoteka (mounted FS). Ako se u uređaju nalazi aktivirani sistem datoteka, kernel ne može pozvati device-close proceduru, zato što to nije poslednji close za uređaj. Čak i da na uređaju nema aktiviranih sistema datoteka, baferski keš može sadržati "delayed write" blokove vezane za taj uređaj, koji se još nisu upisali na disk. Kernel mora da pretraži baferski keš i upiše takve blokove pre poziva device-close procedure. Kada zatvori uređaj, kernel ponovo prolazi kroz bafer keš i poništava validnost (invalidate) za sve bafere koji sadrže blokove za sistem datoteka koji se nalazi na zatvorenom uređaju. Na taj način, kernel prazni baferski keš.

[3] Kernel otpušta inode za device-file.

Device-close procedura prekida vezu sa uređajem, reinicijalizuje drajverske strukture podataka i hardver, tako da kernel može da obavi ponovno otvaranje uređaja.

Procedure za čitanje i upis

Kernelski algoritam za čitanje i upis (read /write) uređaja, sličan je kao i za regularnu datoteku. Ako proces čita ili piše karakter uređaj, kernel poziva drajversku read/wite proceduru. Postoje slučajevi kada kernel prebacuje podatke direktno između korisničkog adresnog prostora i uređaja, iako drajver po pravilu interno baferiše podatke. Na primer, terminal drajveri koriste c-liste da baferišu podatke. U takvim slučajevima, drajver alocira

bafer, kopira podatke iz korisničkog adresnog prostora za vreme write procedure i šalje podatke iz svog bafera na uređaj. Drajver usklađuje brzine: ako korisnički proces generiše podatke brže nego što uređaj može da upisuje, write procedura mora da uspava proces dok uređaj ne bude spreman da primi nove podatke. Za proceduru read, drajver prima podatke u svoj bafer, a onda ih iz bafera prosleđuje u korisnički adresni prostor.

Metod po kome drajver kominicira sa uređajem zavisi od hardvera, pri čemu se koristi separatni I/O ili memorijski mapirani I/O, kao na VAX-u.

Neke mašine koriste programirani I/O, koji znači da mašina sadrži instrukcije koje kontrolišu uređaje (IO instructions, not memory).

Pošto je interface između drajvera i hardvera mašinski zavistan, ne postoji standardni interfejs na ovom nivou. Za memorijski mapirani IO (memory-mapped) i za programirani IO (programmed I/O), drajver može koristiti DMA kanal, za transfer podataka između uređaja i memorije, pri čemu DMA radi ceo transfer paralelno sa CPU, a na kraju uređaj postavlja prekid koji ukazuje da je transfer završen. Drajver postavlja fizičke adrese u DMA na bazi virtuelnih adresa i stranične organizacije (paging map).

Brzi uređaji mogu direktno kopirati podatke između uređaja i korisničkog adresnog prostora, bez upotrebe kernelskog bafera, a to će ubrzati transfer, jer imamo jednu operaciju kopiranja manje (device to kernel buffer), pri čemu veličina trasfera nije limitirana veličinom bloka kernel bafera. Takav transfer se naziva neobrađeni ili "raw" IO i obično se poziva iz karakter read i write procedura.

Strategijski interfejs

Kernel koristi strategijski interfejs (strategy interface) za prenos podataka između baferskog keša i uređaja, po pravilu za blok uređaje. Mada read i write procedure karakter uređaja ponekad koriste njihove strategijske procedure za transfer podataka, direktno između uređaja i korisničkog adresnog prostora. Strategijska procedura po pravilu I/O poslove postavlja u red čekanja (queue) za uređaj, a posle toga otpočinje sofisticirano I/O raspoređivanje (IO scheduling). Kernel prosleđuje adresu baferskog zaglavlja (headera) u drajversku strategijsku proceduru, a zaglavlje sadrži listu adresa (pages) i veličinu za prenos podataka sa uređaja ili na uređaj. Za baferski keš, kernel prenosi podatke sa jedne memorijske adrese, a kada obavlja swap operacije, kernel prebacuje podatke sa više memorijskih adresa (pages). Ako se podaci kopiraju sa ili iz korisničkog adresnog prostora, driver mora zaključati proces u memoriji ili bar stranice koje se kopiraju dok se I/O transfer ne okonča.

Na primer, posle aktiviranja sistema datoteka (FS mounting), kernel identifikuje svaku datoteku u tom sistemu datoteka, preko broja uređaja za taj sistem datoteka i broja inode strukture za datoteku. Broj uređaja za sistem datoteka se koduje kao glavni i sporedni broj (major i minor number). Kada kernel pristupa bloku iz datoteke, on kopira broj uređaja i broj bloka na bafersko zaglavlje header. Kada keš algoritmi (bread ili bwrite) pristupaju disku, oni pozivaju strategijsku proceduru na bazi glavnog i sporednog broja uređaja. Strategijska procedura koristi polja za glavni i sporedni broj kao i broj

bloka u baferskom zaglavlju, da identifikuju gde se nalaze podaci na uređaju i prenosi podatke u taj bafer. Takođe, ako proces pristupa blok uređaju direktno, (ako proces otvara blok uređaj i čita ga i upisuje), on koristi keš algoritme koji rade na upravo opisani način.

Sistemski poziv Ioctl

Sistemski poziv ioctl je generalizacija terminalskih sistemskih poziva stty (set terminal settings) i gtty SC (get terminal settings), a to su sistemski pozivi raspoloživi na ranim verzijama UNIX operativnog sistema. Ioctl obezbeđuje generalnu ulaznu tačku za komande samog uređaja, dozvoljavajući procesu da postavlja hardverske opcije za uređaj i softverske opcije za drajver. Akcije koje omogućava ioctl zavise od uređaja, a definisane su od strane drajvera. Program koji koristi ioctl mora da zna tip datoteke sa kojom će da se bavi, zato što su one specifične za dati uređaj. Ovo je jedini izuzetak generalnog pravila da UNIX ne pravi razliku između tipova datoteka. Videćemo detaljno korišćenje sistemskog poziva ioctl za terminale, a sada da objasnimo sintaksu:

```
ioctl(fd, command, arg)
```

gde je fd deskriptor datoteke koji je vratio sistemski poziv open, command je zahtev drajveru da obavi neku akciju, a arg je parametar (moguće pointer na strukturu) za command. Komande su drajverski specifične, međutim svaki drajver interpretira komande prema internim specifikacijama, a format strukture arg zavisi od same komande. Drajveri mogu čitati arg iz korisničkog prostora na bazi unapred definisanog formata ili mogu upisati trenutno setovanje uređaja u arg strukturu. Na primer, ioctl omogućava korisniku da postavi baud-rate za terminal, da premotaju traku, da postave mrežnu adresu na mrežne kartice itd.

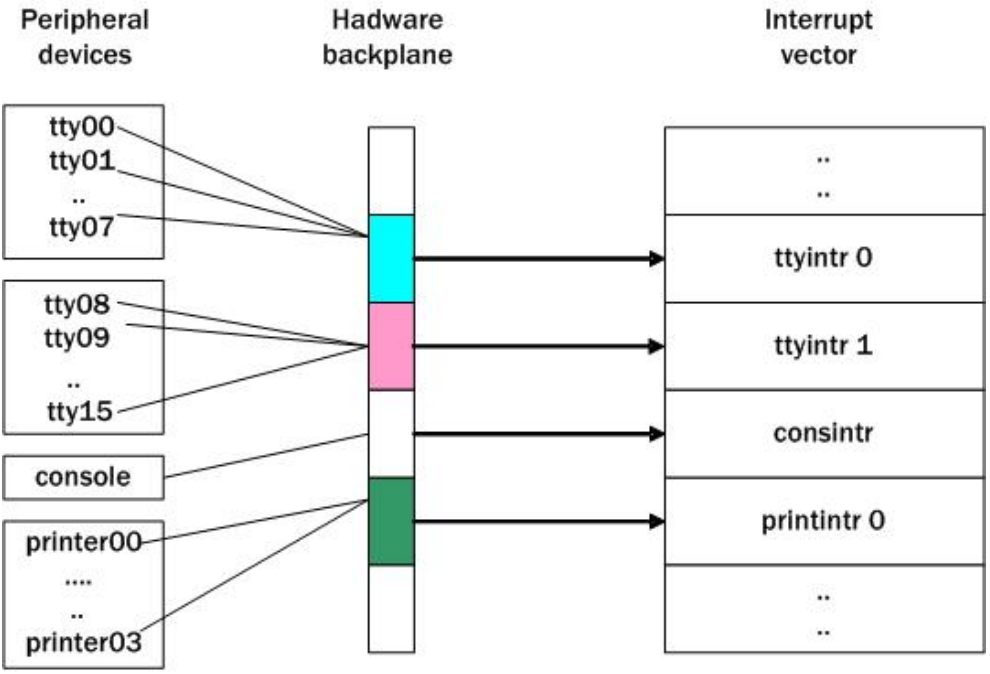
Ostali sistemski pozivi za sistem datoteka

Sistemski pozivi za sistem datoteka, kao što su stat i chmod rade sa uređajima kao da su regularne datoteke. Oni manipulišu inode strukturom bez pristupanja drajveru. Čak i sistemki poziv lseek radi sa uređajima, na primer ako proces obavi lseek na traku, kernel ažurira pomeraj u tabeli datoteka ali ne obavlja drajversku (driver-specific) operaciju. Kada proces kasnije čita ili upisuje na traku, kernel kopira pomeraj u tabeli datoteka u u-područje, kao za običnu datoteku, a uređaj se fizički pomera na korektan pomeraj na koji ukazuje u-područje.

Prekidne rutine

Pojava prekida izaziva kernel da izvršava prekidnu rutinu (interrupt handler), koji se bazira na korelaciji uređaja koji je izazvao prekid i pomeraja u prekidnoj vektor tabeli IVT. Kernel poziva prekidnu rutinu, prosleđujući joj broj uređaja ili druge parametre, pomoću kojih može da se identifikuje uređaj koji je izazvao prekid. Na primer, slika 10.4 prikazuje dva ulaza u IVT za upravljanje terminal prekidima ("ttyintr"), gde svaki ulaz upravlja

prekidima za osam terminala. Ako terminal tty09 postavi prekid, kernel će pozvati odgovarajuću prekidnu rutinu, na bazi tabele IVT i pozicije hardverskog uređaja koji je postavio prekid. Više uređaja mogu da budu na istom IVT ulazu (ovde osam), pa drajver mora da identifikuje tačan uređaj.



Slika 10.4. *Primer za IVT*

Na slici 10.4, dva ulaza u IVT tabeli za terminale (ttyintr) su označena kao 0 i 1, mada je moguće da oni zovu istu prekidnu rutinu, a da se 0 i 1 prosleđuju kao parametar za tu prekidnu rutinu. Prekidna rutina mora da ima mehanizam, da odredi koji je terminal postavio prekid, pa se onda prekid obrađuje na odgovarajući način.

Disk drajveri

Istorijski, UNIX je uvek dozvoljavao da se disk deli na više delova koji sadrže sisteme datoteka. Na primer, ako disk sadrži četiri sistema datoteka, administrator može postaviti jedan od tih sistema datoteka da bude neaktiviran, jedan od njih da se aktivira samo za čitanje (read-only), a dva sistema datoteka da se aktiviraju normalno tj. i za čitanje i za upis. Mada su svi sistemi datoteka delovi istog diska, korisnik ne može pristupati datotekama u neaktiviranom sistemu datoteka, niti može pisati u read-only sistemu datoteka.

Disk drajver translira adresu u sistemu datoteka koja se sastoji od broja uređaja i broja bloka u partikularni blok na disku. Drajver dobija adresu na jedan od dva načina:

- strategijska procedura koristi bafer iz baferskog keša, a bafersko zaglavlje sadrži i broj uređaja i broj bloka
- read i write procedurama se prosleđuje broj uređaja kao parametar, a procedure konvertuju tekući pomeraj u bajtovima, koji je sačuvan u u-području, u odgovarajuću disk blok adresu.

Disk drajver koristi broj uređaja da identifikuje fizički disk i sistem datoteka , preko interne tabele u kojoj je označen početak svakog sistema datoteka. Na bazi te tabele i pomeraja u sistemu datoteka, dodaje se pomeraj na početak sistema datoteka i dobija se traženi sektor.

Evo jednog primera podele diska, koji je dat na slici 10.5.

section	start block	length in blocks
size of block = 512 bytes		
0	0	64000
1	64000	944000
2	168000	840000
3	336000	672000
4	504000	504000
5	672000	336000
6	840000	168000
7	0	1008000

Slika 10.5. *Primer podele diska*

Pretpostavimo da su disk blok datoteke /dev/dsk0, /dev/dsk1, /dev/dsk2 i /dev/dsk3, prve četiri sekcije na disku sa glavnim brojem nula i sporednim brojem od 0 do 3. Pretpostavimo da je veličina sistemskog bloka jednaka veličini bloka na disku tj. 512 bajtova. Ako sistem pokušava sa se obrati bloku broj 940 na sistemu datoteka /dev/dsk3, disk drajver će konvertovati tu adresu u adresu bloka na disku koja iznosi 336940 = 336000 + 940.

Veličina sistema datoteka na disku može da varira, a određuju je administratori sistema. Za ceo disk postoji posebna datoteka, u ovom slučaju to je /dev/tty07. Korišćenjem fiksnih sekcija na disku, smanjuje se fleksibilnost. Zato se informacija o disk

particijama ne čuvaju u disk drajveru, nego u konfigurabilnim tabelama na samom disku, mada je iz razloga kompatibilnosti teško definisati univerzalnu poziciju za tu tabelu na disku. Na primer boot blok je početak svakog diska i obično je u njemu master volume tabela, a superblok je početak svakog UNIX sistema datoteka. Bez obzira na sve, disk drajver zna gde je master volume tabela za dati disk.

Mnogo pažnje je posvećeno disk performansama, tako da su mnoge funkcije kao što je disk raspoređivanje (sheduling), data transferi itd, uglavnom prebačene iz drajvera u sam disk kontroler itd.

Sistemski programi za obradu diska, mogu koristiti ili raw ili blok interfejs za pristup disku direktno bez sistemskih poziva vezanih za sistem datoteka. Dva značajna programa za diskove su mkfs i fsck. Program mkfs kreira sisteme datoteka u particiji, tako što kreira supeblok strukturu, inode listu, povezanu listu slobodnih disk blokova, root direktorijum na novom sistemu datoteka. Program fsck proverava konzistenciju postojećeg sistema datoteka i ispravlja moguće greške.

Analizirajmo sledeći program ci. Prethodno pogledajmo blok i karakter datoteku za /dev/dsk15 dobijenu sa komandom:

```
$ ls –l /dev/dsk15 /dev/rdsk15
br-------- 2  root  root  0,  21 Feb 12  15:40  /dev/dsk15
crw-rw---- 2  root  root  7,  21 Mar 7   09:29  /dev/rdsk15
```

Blok specijalna datoteka /dev/dsk15, vlasnik je root, za blok datoteku samo root ima pravo čitanja i ima glavni broj nula, a sporedni broj 21. Karakter specijalna datoteka /dev/rdsk15, vlasnik je root, za ovu datoteku i vlasnik i grupa imaju read i write pravo, a ima glavni broj sedam, a sporedni broj 21.

```
#include "fcntl.h"
main()
{
  char buf1[4096], buf2[4096];
  int fd1, fd2, i;
  if((fd1=open("/dev/dsk15", O_RDONLY)) = = -1) ||
    ((fd2=open("/dev/rdsk15", O_RDONLY)) = = -1))
  {
    printf("failure on open");
    exit();
  }
  lseek (fd1, 8192L,0)
  lseek (fd2, 8192L,0)
  if((read(fd1, buf1, sizeof(buf1)) = = -1) ||
     (read(fd2, buf2, sizeof(buf2)) = = -1)) = = -1))
  {
    printf("failure on read");
    exit();
  }
```

```
    for(i=0; i< sizeoff(buf1); i++)
    {
      if( buf1[i] != buf2[i])
      {
        printf("different at offset %d",i);
        exit();
      }
      printf("reads match");
    }
}
```

Proces otvara datoteku /dev/dsk15, pristupa disku preko blok prekidačke tabele, a datoteci /dev/rdsk15 preko karakter prekidačke tabele. Kako su sporedni brojevi isti za obe datoteke, obe datoteke ukazuju na istu particiju na disku, pa će proces izvršiti open proceduru istog drajvera dva puta (ali preko različitih prekidačkih tabela), zatim se obavlja lseek na isti pomeraj i čitaju se podaci sa istih blokova diska, što bi trebalo da proizvede isti rezultat.

Programi koji čitaju i pišu direktno su opasni jer mogu da prepišu osetljive podatke. Administratori moraju da zaštite i blok i neobrađeni (raw) pristup disku, kao na u datom primeru, gde su i blok i karakter datoteke vlasništvo korisnika root i samo on može da ih čita ili piše.

Programi koji čitaju i pišu direktno na disk, mogu da oštete konzistenciju sistema datoteka jer mogu da prepišu direktorijume, inode blokove itd.

Razlika između blok i karakter disk interfejsa je u tome da li koriste baferski keš ili ne.

- Kada proces otvara blok specijalnu datoteku (block interface), sve je isto kao kod obične datoteke, osim što posle konvertovanja bajt pomeraja u blok pomeraj (algoritam bmap), kernel tretira logički blok pomeraj kao fizički broj bloka u sistemu. Zatim pristupa podacima preko baferskog keša preko strategijske procedure.

- Kada proces otvara karakter specijalnu datoteku (raw interface), kernel ne konvertuje byte pomeraj u pomeraj datoteke, nego prosleđuje pomeraj direktno drajveru u u-području, a onda drajverske read/write rutine obavljaju konvertovanje bajt pomeraja u blok pomeraj, i kopiraju podatke direkno sa diska ili na disk, ali bez bafeskog keša (bypass cache).

Ako jedan proces piše u blok datoteku, a drugi proces čita isti blok ali kroz karakter datoteku, ovaj drugi proces sa raw pristupom, možda neće pročitati ono što je prvi proces upisao, zato što su mu podaci u kešu a ne na disku (delayed write), a da je čitao kroz blok interfejs, dobio bi iz keša sveže podatke.

Korišćenje raw interfejsa može izazvati čudno ponašanje. Ako proces čita ili piše na raw device u jedinicama manjim od bloka na disku, rezultati zavise od samog drajvera.

Na primer, ako pošaljete jednobajtne upise na traku, svaki bajt može se pojaviti na različitom bloku trake.

Prednost korišćenja raw interfejsa je brzina, ako nema ponovnog čitanja, kada bi keš imao veću prednost. Procesi koji pristupaju po blok interfejsu su ograničeni na transfere veličine sistemskog bloka, pri čemu jedna disk operacija prenosi jedan sistemski blok podataka (1K na primer).

Ako se koristi raw disk interfejs, veličina bloka je ograničena disk kontrolerom. Funkcionalno, čitanje će biti isto, s tim što je raw interfejs mnogo brži. U prethodnom programu proces će 4K podataka (1K system block) pročitati u četiri iteracije, dok će čitanje kroz raw disk interfejs da se zadovolji u jednoj disk operaciji. Takođe, nema ni duplog transfera podataka sa diska u keš, pa iz keša u korisnički prostor.

Terminal drajveri

Teminal drajveri imaju funkciju da kontrolišu prenos podataka sa terminala i u terminal. Terminali su specijalni uređaji, jer predstavljaju korisnički interfejs. Da bi omogućili interaktivno korišćenje UNIX operativnog sistema, terminal drajveri koriste unutrašnji interfejs u module za disciplinu linije, koji interpretiraju ulaz i izlaz. U kanoničkom modu, disciplina linije konvertuje neobrađenu (raw data) sekvencu podataka otkucanu na tastaturi u kanoničku formu, pre nego što se podaci pošalju procesu koji ih prima. Disciplina linije takođe konvertuje neobrađenu izlaznu (raw output) sekvencu podataka koju je pripremio proces za upis na terminal u formatu koji korisnik očekuje. U neobrađenom "raw" modu, disciplina linije preskače konverziju, podaci se prosleđuju onakvi kakvi jesu.

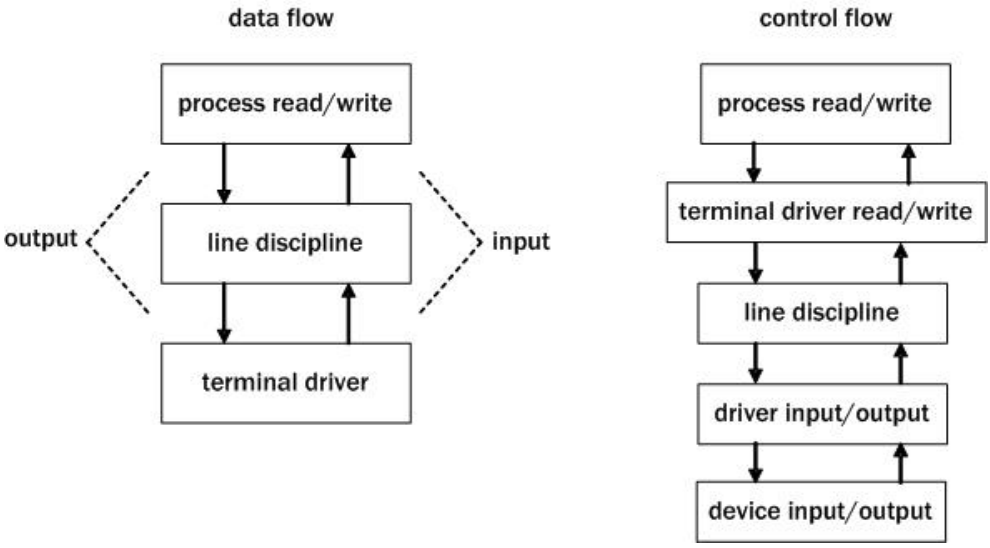
Na primer, ako kucate liniju pa pogrešite, otkucajte erase karakter, a terminal će poslati sve što ste otkucali uključujući i erase karakter. U kanoničnoj formi, disciplina linije baferiše podatke u linije (linija je sekvenca karaktera sve do znaka za novi red) i procesira erase karaktere interno, pre nego što pošalje promenjenu sekvencu procesu.

Funkcije discipline linije su:

- podeliti ulazni niz u linije
- obraditi erase karaktere
- obraditi "kill" karakter koji znači da treba poništiti sve do tada otkucane karaktere u liniji
- obaviti echo (write) primljenih karaktera sa terminala
- proširiti karaktere kao na primer tab karakter (kao sekvencu praznina)
- generisati signale procesima za terminalski prekid (hangup), line breaks ili kao odgovor na delete taster
- dozvoliti raw mode koji ne interpretira specijalne karaktere kao erase, kill ili CR

Da bi se podržavao raw mode, podrazumeva se upotreba asinhronog terminala, tako da proces može da čita svaki karakter koji se otkuca, a u normalnom slučaju proces čeka da korisnik otkuca taster CR t.j enter.

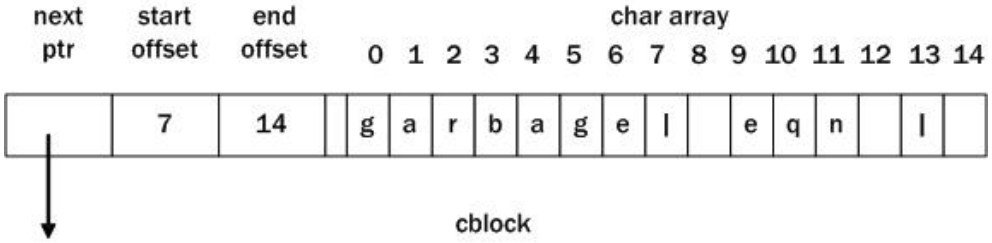
Prvi UNIX sistemi nisu imali disciplinu linije u kernelu što je kasnije urađeno, jer je disciplina linije potrebna za mnoge programe a ne samo za shell i editor, pa je ugrađena u kernel. Mada disciplina linije izvršava funkcije koje su smeštene između terminal drajvera i ostatka kernela, kernel ne poziva disciplinu linije direktno, nego kroz terminal drajver. Na slici 10.6 je prikazan logički tok podataka kroz terminal drajver, disciplina linije i tok kontrole kroz terminal drajver. Korisnici mogu da specificiraju koju disciplinu linije hoće da koriste preko sistemskog poziva ioctl, ali je teško realizovati da jedan uređaj koristi istovremeno više disciplina linije simultano, gde svaki modul za disciplinu linije poziva sledeći modul da procesira obrađene podatke.



Slika 10.6. *Dijagram toka podataka i kontrole za terminal*

C-liste

Disciplina linije manipuliše podacima u c-listama (clist). C-lista (character list), je povezana lista c-blokova promenljive dužine, sa promenljivim brojem karaktera u listi. C-blok sadrži pokazivač na sledeći c-blok u povezanoj listi, polje podataka (obično je malo) i dva pomeraja, koji ukazuju na validne podatke u polju, kao na slici 10.7. Početni pomeraj (start offset) ukazuje početak validnih podataka. Krajnji pomeraj (end offset) ukazuje na prvi non-valid podatak.



Slika 10.7. *Primer za c-blok (cblock)*

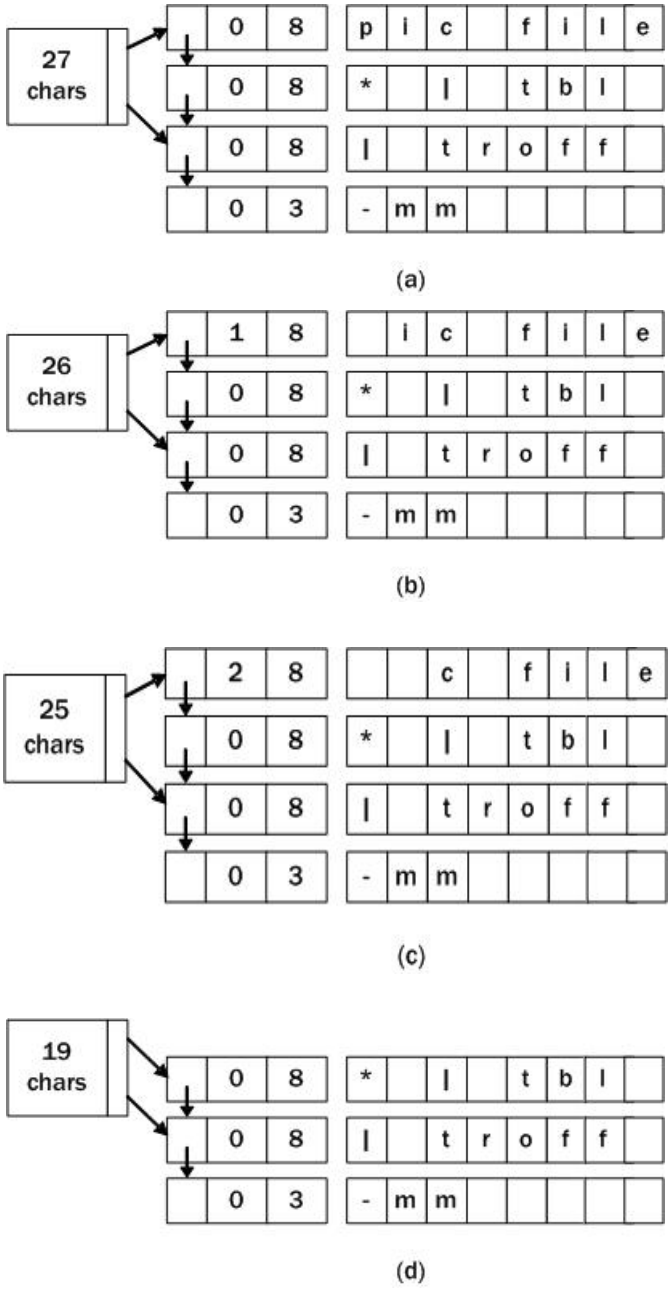
Kernel podržava povezanu listu slobodnih c-blokova i ima šest operacija nad c-listama i c-blokovima:

- operacija dodele c-bloka iz slobodne liste u drajver
- operacija povratka c-bloka u slobodnu listu
- prosleđivanje prvog karaktera iz c-liste. Kernel može proslediti prvi karakter iz c-liste: on uklanja prvi karakter iz prvog c-bloka, podešava broj karaktera u c-listi i podešava prvi pomeraj u prvom c-bloku, tako da sledeća operacija neće biti prosleđivanje istog karaktera, već sledećeg. Ako se prosleđivanje odnosi na zadnji karakter u c-bloku, taj c-blok postaje prazan, pa kernel plasira taj prazan c-blok na slobodnu listu i podešava c-list pokazivače. Ako c-lista ne sadrži karatere kada se prosleđivanje zahteva, kernel vraća null karakter.
- dodavanje novog karaktera na kraj c-liste. Kernel može postaviti karakter na kraju c-liste, nalazeći zadnji c-blok, stavljući karakter u njega i podešavajući pomeraje. Ako je c-block pun, alocira se novi cblok iz slobodne liste, ubacuje se u c-listu a u njega se dodaje karakter.
- Kernel može da ukloni grupu karaktera sa početka c-liste i to jedan po jedan c-block u jednom trenutku, što je ekvivalentno uklanjanju svih karaktera u c-bloku jedan po jedan.
- Kernel može plasirati popunjeni c-blok na kraj cliste

Cliste obezbeđuju prosti baferski mehanizam, zgodan za male količine podataka i spore uređaje kao što je terminal. One dozvoljavaju manipulaciju podataka na bazi jednog po jednog karakter u grupi c-blokova.

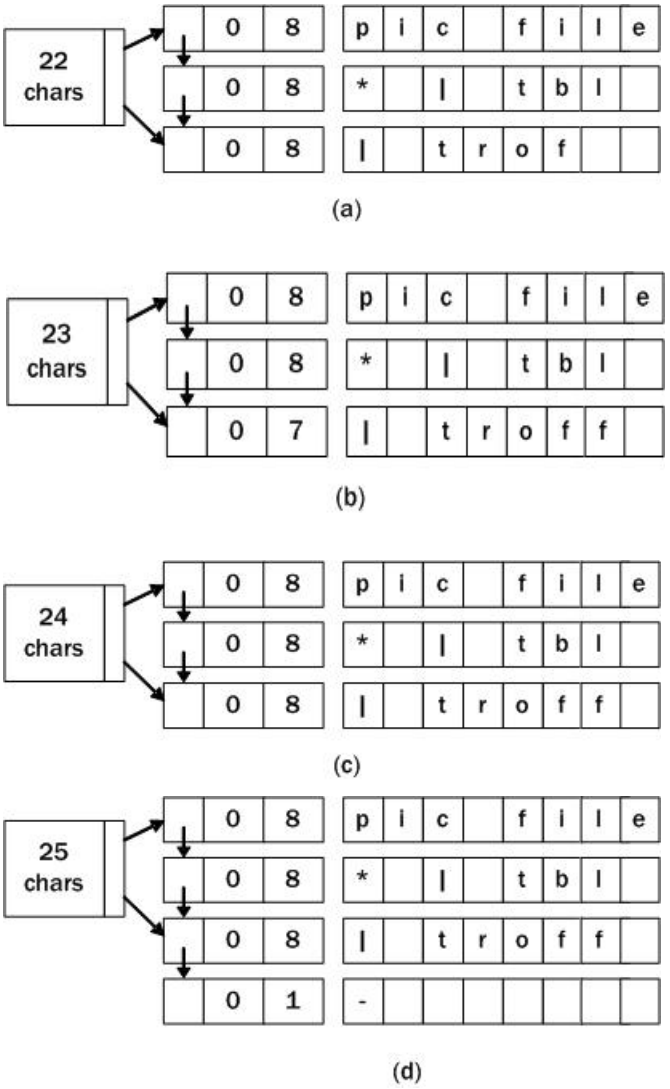
Primeri C-liste

Na slici 10.8 je dat primer ukljanjanja karaktera iz c-liste, pri čemu kernel ukljanja jedan po jedan karakter iz prvog c-bloka kao na slikama od (a) do (c), sve dok nema više karaktera u c-bloku kao na slici (d), kada se c-block izbacuje.



Slika 10.8. *Primer uzimamanja karaktera u c-listu*

Slično slika 10.9 prikazuje dodavanje karaktera u c-listu (cblock sadrži 8 karaktera) i kada nema više mesta, kernel linkuje novi c-block kao na slici (d).



Slika 10.9. *Dodavanje karaktera u c-listu*

Terminal drajver u kanoničnom modu

Strukture podataka za terminal drajvere imaju tri c-liste pridružene sa njima:

- c-lista koja čuva podatke za izlaz na terminal (output)
- c-lista koja čuva neobrađene "raw" ulazne input podatke koje obezbeđuje prekidna rutina za tastaturu, kada korisnik otkuca karaktere na tastaturi
- c-lista koja čuva obrađene "cooked" ulazne podatke, pošto disciplina linije konvertuje specijalne karaktere u "raw" listi, kao što je erase ili kill karakter.

algoritam upis na terminal terminal_write

```
algorithm terminal_write
{
  while(more data to be copied from user space)
  {
    if(tty flooded with output data)
    {
      start write operation to hardware with
        data on output clist;
      sleep (event: tty can accept  more data);
      continue; /*back to while loop*/
    }
    copy cblock size of data from user space to output clist;
    line discipline convert tab characters, etc;
  }
  start write operation to hardware with data on output clist;
 }
```

Kada proces upisuje na terminal, terminal drajver poziva disciplinu linije (LD modul). Disciplina linije ima petlju, čita izlazne karaktere iz korisničkog adresnog prostora i ubacuje ih izlaznu c-listu sve dok ima mesta. Disciplina linije procesira izlazne podatke, proširuje tab karaktere u seriju space karaktera, itd. Ako broj karaktera u izlaznoj c-listi postane veći od gornje granice (high-water mark), disciplina linije poziva drajversku proceduru za prenos podataka iz izlazne c-liste na terminal i postavlja proces koji je otpočeo upis na terminal da ide na spavanje. Kada količina podataka u izlaznoj c-listi padne ispod donje granice (low-water mark), prekidna rutina za tastaturu budi sve procese uspavane na događaju da terminal može da prihvati još podataka. Disciplina linije završava svoju petlju, kopira sve podatke iz korisničkog adresnog prostora u izlaznu c-listu i poziva drajverske procedure da prebace podatke na terminal.

Ako više procesa pokušava upis na terminal istovremeno, oni prate opisanu proceduru nezavisno. Izlaz može biti čudan, podaci raznih procesa mogu se preklapati na terminalu tj. ekranu. Ovo se može desiti zato što proces može pisati na terminal koristeći

više sistemskih poziva za upis na terminal. Kernel može obaviti prebacivanje kontesta dok je proces u korisničkom modu, između sukcesivnih sistemskih poziva za upis na terminal. Drugi i aktivni proces može takođe pisati na terminal dok početni proces spava. Izlazni podaci mogu biti izmešani na terminalu, zato što proces koji piše može zaspati u sredini sistemskih poziva za upis na terminal dok čeka da se upišu prethodni podaci. Kernel može izabrati drugi proces koji piše na terminal dok je prvi uspavan. Kernel ne garantuje da će sadržina bafera za izlaz na terminal biti kontinualna na terminalu, zbog mogućnosti simultanih upisa.

Primer za algoritam terminal_write

Analizirajmo sledeći program:

```
char form[] = "this is a sample output string from child"
main
{
  char output[128];
  int i;
  for (i=0, i<18; i++)
  {
    switch (fork())
    {
      case -1: exit(); /* error = = = hit max processes*/
      default: /*parent process*/
      break;
      case 0; /*child process*/
      /*format output string in variable output*/
      sprintf(output, "%s%d\n%s%d\n", form, i, form, i);
      for(;;) write(1, output, sizeof(output));
    }
  }
}
```

Proces roditelj kreira 18-oro dece, svako dete-proces formatira niz (preko funckije spintf) u polju output, sprintf uključuje poruku i vrednost, koja je redni broj sistemskog poziva fork(), a samim tim redni broj deteta, a zatim dete u petlji upisuje string na terminal (fd = 1). Namerno je podešeno da output string bude veći od 64 bajta, da ne može da stane u jedan c-block koji je na sistemu UNIX System V = 64 bajta. Kada terminal drajver traži više od 64 bajta za upis na terminal, output može postati izmešan, kao u sledećem primeru:

```
this is a sample output string from child 1
this is a sample outthis is a sample output string from child 0
```

Čitanje se sa terminala

Čitanje podataka sa terminala u kanoničnoj formi mnogo je kompleksnija operacija. Sistemski poziv za terminal read specificira broj bajtova koje proces želi da pročita, ali disciplina linije će zadovoljiti sistemski poziv read sa terminala uvek kada korisnik otkuca enter tj. CR, bez obzira što broj bajtova nije zadovoljen. To je praktično s obzirom da proces ne može da predvidi koliko će karaktera korisnik otkucati na tastaturi, pa nije bitno koliko ima karaktera. Na primer, za korisnika nije bitno da li je otkucana komanda prosta kao date ili ls, ili je složena kao

```
pic file* | tbl | eqn | troff – mm –Taps | apsend
```

Terminal drajver ne zna ko čita sa terminala, da li je to shell, editor ili nešto drugo, disciplina linije reaguje na svaki enter i zadovoljava sistemski poziv read odmah posle enter.

algoritam terminal_read

```
algorithm terminal_read

{
  if(no data on canonical clist)
  {
    while(no data on raw clist)
    {
      if(tty opened with no delay option) return;
      {
        if(tty in raw mode based on timer and timer not active)
          arrange for timer wakeup (callout table);
        sleep(event: data arrives from terminal);
      }/* there is data on raw clist*/
    if(tty in raw mode) copy all data from raw
      clist to cannonical clist;
    else
    {
      while(characters on raw clist)
      {
        copy one character at time from raw clist to canonical
          clist: do erase, kill processing;
        if(char is CR or end-of-file) break;
          /*out of while loop*/
      }
    }
    start write operation to hardware with data on output clist;
  }
  while(characters on canonical list and
```

```
        read count not satisfied)
  copy from cblocks on canonical list to user adress space;
}
```

Pretpostavimo da je teminal u kanoničkom modu. Ako nema podataka na ulaznoj c-listi, proces koji čita sa terminala spava sve dok ne dođu podaci sa terminala. Kada se podaci unesu, terminalska prekidna rutina poziva prekidnu rutinu iz discipline linije, koji plasira podatke na raw c-listu za ulaz procesa koji čita i na izlaznu c-listu za prikazivanje echo karaktera na terminalu. Ako ulazni niz sadrži karakter enter, prekidna rutina će probuditi sve uspavane procese koji čitaju terminal. Kada se proces koji čita terminal izvršava, drajver izbacuje karaktere iz raw c-liste, obavlja erase i kill procesiranje i plasira karaktere u kanoničnu c-listu. Zatim se kopiraju karakteri u korisnički adresni prostor sve dok se ne pojavi enter ili se ne zadovolji broj bajtova u terminal_read zahtevu (zavisno koji je broj manji). Međutim, probuđeni proces može da detektuje da podaci zbog kojih je ustao više ne postoje, zato što drugi procesi mogu pročitati terminal i ukloniti podatke iz raw c-liste pre nego što probuđeni proces dobije kontrolu. Slična sitacija se dešava kada više procesa čita iz pipe datoteke.

Karakter procesiranje u ulaznom i izlaznom smeru je asimetrično, praćeno sa dve ulazne c-liste i jednom izlaznom c-listom. Disciplina linije izbacuje podatke iz korisničkog prostora, procesira je i ubacuje izlaznu c-listu. Da bi bilo simetrično, trebalo bi da postoji samo jedna ulazna c-lista. Ali to bi značilo da prekidna rutina u procesiranju erase i kill karaktera mora biti kompleksniji i da blokira druge prekide u kritičnoj sekciji. Korišćenjem dve ulazne c-liste, znači da prekidna rutina može da upisuje karaktere prosto u raw c-listu i budi uspavane procese koji čitaju terminal. Prekidna rutina takođe gura ulazni karakter neposredno na izlaznu c-listu, tako da posle minimalnog kašnjenja korisnik vidi otkucani karakter na terminalu.

Primer za algoritam terminal_read

Slika prikazuje program gde se kreira puno dece koja čitaju svoj standarni ulaz, a to je terminal koji je prespor da zadovolji sve procese koji čitaju terminal, tako da će procesi provesti mnogo vremena spavajući u terminal-read algoritmu.

```
char input[256]
main
{
  register int i;
  for (i=0, i<18; i++)
  {
    switch (fork())
    {
      case -1: /* error = = = hit max processes*/
      printf("error cannot fork");
      exit();
      default: /*parent process*/
```

```
        break;
        case 0; /*child process*/
        for(;;)
        {
          read(0, input, 256); /*read line*/
          printf("%d read %s \n", i, input);
        }
      }
    }
  }
```

Kada korisnik unese liniju podataka, terminalska prekidna rutina budi sve procese koji čitaju sa terminala, pošto oni svi spavaju na tom istom prioritetnom nivou, svi su poželjni. Ne može da se predvidi koji će proces da se izvršava, tj. da čita liniju, a to će sam program da prikaže na terminalu, tj. proces-dete ispisuje svoj indeks (i) i liniju koju je pročitao. Ostali procesi će dobiti CPU, ali verovatno neće imati ulaze podatke na ulaznoj c-listu i idu opet na spavanje. Procedura se ponavlja na unos svake nove linije, a ne može se garantovati da jedan proces neće potrošiti sve ulazne podatke.

Situacija sa višestrukim čitanjem terminala je komlikovana, ali kernel mora dozvoliti višestrukim procesima da čitaju terminal simultano. Ako ne bi tako bilo, tj. ako bi samo jedan proces mogao čitati terminal, tada proces koga je startovao shell i koji očekuje neki ulaz sa tastature neće moći da se izvršava, zato što shell takođe pristupa standarnom ulazu. Zato procesi moraju sinhronizovati svoj pristup terminalu na korisničkom nivou.

Kada korisnik otkuca "end of file" karakter (ctrl-d), disciplina linije zadovoljava terminal_read sistemski poziv, puneći ulazni niz sve do EOF, ali se EOF ne uključuje. Ako je EOF jedini otkucani karakter, povratna vrednost za read sistemski poziv biće „no data“ tj. povratna vrednost nula. Proces koji je pozvao sistemski poziv read, mora da prepozna EOF i da okonča svoj sistemski poziv read sa terminala. Na primer, shell tada završava svoju petlju na EOF, dobije nulu i shell obavlja sistemski poziv exit.

Ova sekcija obuhvata prost terminalski hardver koji prenosi podatke za terminal jedan po jedan, kako ih korisnik kuca. Inteligentni terminali obrađuju svoj ulaz interno, oslobađajući glavni CPU. Terminal drajver za njih liči na drajvere za proste terminale, ali funkcije discipline linije se prilagođavaju osobinama inteligentnih terminala.

Terminal drajver u raw modu

Korisnik može da setuje terminalske parametre kao što je erase i kill karakter i dobija setovane vrednosti preko sistemskog poziva ioctl. Slično, ioctl kontroliše da li terminal izbacuje echo za svoj ulaz, setuje terminal baud rate, prazni U/I request, ručno postavlja start i stop karakter. Terminal drajver ima strukture podataka koje čuvaju različito terminalsko setovanje, a disciplina linije prima parametre preko sistemskog poziva ioctl i postavlja ili čita polja iz terminalskih struktura podataka.

Kada proces setuje terminalske parametre, to važi za sve procese koji koriste terminal. Setovanje se ne ukida automatski, kada proces obavi exit.

Procesi mogu da postave terminal u raw mode, gde disciplina linije prebacuje karaktere onako kako ih je korisnik otkucao, nikakvo ulazno procesiranje se ne obavlja. Kernel mora da zna kako tj kada da zadovolji sistemski poziv read, u slučaju da se karakter CR tj <enter>, tretira kao običan karakter. To se radi tako što se sistemski poziv read zadovolji posle minimalnog broja ulaznih karaktera ili posle nekog fiksnog vremena od prijema prvog karaktera sa terminala, a to vreme kernel podešava u callout tabeli. Oba kriterijuma se setuju preko sistemskog poziva ioctl. Kada se neki kriterijum ispuni, prekidna rutina za discipline linije budi uspavane procese. Drajver prebacuje sve karaktere iz raw liste u kanoničku listu i radi sve kao u algoritmu za kanonički mod. Raw mod je značajan za screen orijentisane aplikacije, kao što je vi editor, koji ima mnoge komande koje se završavaju sa enter.

Primer za raw mode

Na slici je dat program koji obavlja ioctl da sačuva tekući terminalsko postavljanje (settings) za fd jednak nuli, (a to je standardni ulaz). Komanda ioctl TCGETA daje nalog drajveru za dobijanje setovanja za terminal i to se čuva u strukturi savetty u korisničkom prostoru. Ova komada se koristi da odredi da li je datoteka terminal ili ne, a ne menja ništa u sistemu: u slučaju da ioctl otkaže to onda nije bio terminal. Ovde proces obavlja i drugi ioctl sistemski poziv koji postavlja terminal u raw mod. Ukida se echo i aranžira da se zadovolji terminal read posle minimalno pet karaktera primljenih sa terminala, ili kad prođe deset sekundi od prijema prvog. Kada dobije prekidni signal, proces resetuje terminal u originalno setovanje i završava.

```
#inlcude <signal.h>
#inlcude <termio.h>
struct termio savetty;
main()
{
  extern sigcatch();
  struct termio newtty;
  int nrd;
  char buf[32];
  signal(SIGINT, sigcatch)
  if(ioctl(0, TCGETA, &savetty) == -1)
  {
    printf("ioctl failed: not a tty")
    exit();
  }
  newtty=savetty;
  newtty.c_lflag        &= ~ICANON   /* turn off canonical mode*/
  newtty.c_lflag        &= ~ECHO     /* turn off echo */
  newtty.c_cc[VMIN]     = 5          /* minimum 5 characters */
```

```
  newtty.c_cc[VTIME]    = 100               /* 10 sec interval */
  if(ioctl(0, TCSETAF, &newtty) == -1)
  {
    printf("cannot put tty into raw mode")
    exit();
  }
  for (;;)
  {
    nrd = read(0, buf, sizeoff(buf));
    buf[nrd] =0;
    printf("read %d chars %s",nrd, buf)
  }
}
```

Terminal poliranje

Ponekad je zgodno polirati uređaj, što znači čitati ako su podaci prisutni, a ako nisu prisutni, treba nastavitii regularno dalje procesiranje. Program prikazuje taj slučaj:

```
#include <fctl.h>
main
{
  register int i, n;
  int fd;
  char buf[256];
  /*open terminal read-only with no-delay option*/
  if((fd = open("/dev/tty", O_RDONLY | O_NDELAY)) == -1) exit();
  n=1;
  for(;;) /*forever*/
  {
    for(i=0; i<n; i++);
    if(read(fd,buf, sizeof(buf) >0)
    {
      printf("read at n%d",n)
      n--;
    }
    else n++; /*no data to read; returns due to no-delay*/
  }
}
```

Proces otvara terminal sa "no delay" opcijom, tako da sistemki poziv read koji sledi neće spavati ako podaci nisu spremni, nego se odmah vraća iz sistemskog poziva read. Takav metod radi ako proces monitoriše uređaje, proces otvara svaki takav uređaj sa "no delay" opcijom i polira ih, čekajući na ulaz sa nekoga od njih, sa napomenom da poliranje troši CPU vreme.

BSD ima sistemski poziv select koji dozvoljava poliranje uređaja. Sintaksa je:

```
select (nfds, rfds, wfds, efds, timeout)
```

gde je nfsd broj deskiptora datoteke koji se selektuju, rfds, wfds i efds ukazuju na bitove maske koji selektuju otvorene deskriptore datoteka. Timeout pokazuje koliko će vremena select spavati, čekajući da podaci dođu, a ako podaci dođu za bilo koji deskriptor fd u sistemskom pozivu select pre isteka timeout vremena, select se završava, pokazajući u masci descriptor koji je selektovan tj. koji ima podatke. Na primer, ako korsinik ide na spavanje dok ne primi nešto na deskriptor 0, 1 ili 2, tada rfds treba da ukazuje na bit masku 7, kada se select vrati, a maska će biti prepisana za onaj deskriptor koji ima spremne podatke. Maska wfds radi slično za write deskriptore, a efds se odnosi na izuzetne (exception) uslove koji postoje na nekom deskriptoru, a to je veoma korisno u mrežama.

Kontrolni terminal

Kontrolni terminal je onaj terminal na kome se korisnici loguju na sistem i kontrolišu procese koji su otpočeli rad sa terminala. Kada proces otvori terminal, terminal drajver otvara disciplinu linije. Ako je proces, proces-vođa (leader), a vođa je postao kao rezultat prethodnog sistemskog poziva setgrp i ako proces nema dodeljeni kontrolni terminal, disciplina linije pravi od otvorenog terminala kontrolni terminal za proces. Pamti se glavni i sporedni broj za taj terminal u u-području i čuva se procesna grupa u strukturama terminal drajvera. Proces koji otvara terminal je kontrolni proces, tipično to je login shell.

Kontrolni terminal igra zapaženu ulogu u upravljanju signalima. Kada korisnik pritisne delete, break, rubout ili quit taster, prekidna rutina poziva disciplinu linije, koja šalje signal svim procesima u procesovoj grupi. Slično ako prekidna rutina terminala primi hangup signal od hardvera, disciplina linije šalje signal svim procesima u procesovoj grupi. Na ovaj način, procesi na posebnom terminalu primaju hangup signal i većina procesa na ovaj signal obavlja sistemski poziv exit. Na ovaj način se ubijaju procesi ukoliko korisnik naglo isključi terminal. Prekidna rutina nakon svega izbacuje kontrolni terminal iz procesne grupe, tako da ti procesi više ne primaju signale generisane na terminalu.

Indirektni terminal drajver

Procesi često imaju potrebu da čitaju i da pišu direktno u kontrolni terminal, čak i kada su standarni ulaz i izlaz redirektovani u druge datoteke. Na primer, shell script može poslati urgentnu poruku direktno na terminal, mada su deskriptori 1 i 2 redirektovani negde drugde. UNIX obezbeđuje indirektni terminal pristup preko drajverske datoteke /dev/tty koja je označena kao kontrolni terminal za svaki proces koji ga ima. Korisnici logovani na različitim terminalima mogu pristupati /dev/tty ali će pristupati različitim terminalima.

Postoje dve implementacije pomoću kojih kernel nalazi kontrolni terminal preko /dev/tty. Prvo, kernel može definisati specijalni broj uređaja za indirektni terminal u specijalnom ulazu u prekidačkoj karakter tabeli. Kada se prozove indirektni terminal, drajver uzima glavni i sporedni broj kontrolnog terminala iz u-područja i poziva realni terminal drajver preko prekidačke tabele.

Drugo, kontrolni terminal se nalazi tako što se testira ime /dev/tty pre open procedure, a taj test treba da vrati ime realnog terminala.

10.2. Login procedura, streams struktura

Proces login (prijavljivanje na sistem)

Proces init se izvršava u beskonačnoj petlji, čita /etc/inittab i izvršava linije prevodeći sistem iz jednog stanja u drugo. U višekorisničkom modu, init mora da omogući korisnicima login proceduru. Init izvršava proces getty (get terminal) i čuva informaciju o tome koji getty proces otvara koji terminal. Svaki getty proces postavlja svoju procesnu grupu preko sistemskog poziva setgrp, radi open za terminalsku liniju i obično spava u sistemkom pozivu open, dok mašina hardverski ne konektuje terminal. Kada se open završi, getty obavi sistemski poziv exec za login proces koji zahteva od korisnika da se identifikuje sa login imenom i lozinkom. Ako se korisnik loguje uspešno, login obavlja sistemski poziv exec za shell u kome korisnik počinje da radi. Taj shell se zove login shell. Shell proces ima isti PID kao i getty i zato je on proces-vođa (process leader). Ako se korisnik ne konektuje uspešno, login pravi exit, zatvara (close) terminalsku liniju, a init ponovo izvršava novi getty. Init pauzira dok ne primi "death of child" signal, onda se budi i izvršava novi getty koji ponovo otvara isti terminal.

Algoritam login

```
algorithm login /* procedure fo logging in*/

{
  getty process executes;
  set process group (setgrp SC);
  open tty line; /*sleep until opened*/
  if(open successful)
  {
    exec login program;
    prompt for user name;
    turn off echo, prompt for password;
    if(successful)
    {
```

```
        put tty in cannonical mode (ioctl);
        exec shell
      }
      else
        count login attempts, try again up to a point;
    }
  }
```

Streams

Razne šeme za realizaciju drajvera pate od raznih nedostataka. Različiti drajveri teže da dupliraju funkcionalnost, posebno drajveri koji realizuju mrežne protokole, koji uključuju komponentu za hardver i komponentu za protokol. Takođe, c-liste su korisne za baferske funkcije, ali su preskupe zbog manipulacije karakter po karakter. Učinjen je pokušaj za povećanjem performansi putem uvođenja modularnosti u I/O drajvere. Nedostaju zajedničke procedure na korisničkom nivou, gde više komandi mogu da obavljaju zajedničku funkciju, ali za različite uređaje.

Ritchie je implementirao šemu koja se naziva streams i koja obezbeđuje veću modularnost i fleksibilnost za I/O sisteme. Stream je full-duplex konekcija između procesa i device-drajvera. Sastoji se od skupa linearno povezanih queue-parova, a u paru imamo deo za ulaz i deo za izlaz. Kada proces upisuje podatke u streams, kernel šalje podatke u izlazne redove (output queues). Kada drajver prima ulazne podatke on ih šalje po ulaznim redovoma (input queues) do procesa koji čita. Redovi prosleđuju poruke sa susednim redovima na bazi precizno definisanog interfejsa.

Svaki par redova (queue pair) ima pridružen kernelski modul, kao što je drajver, disciplina linije ili protokol, a moduli manipulišu podacima prosleđujući ih kroz parove redova.

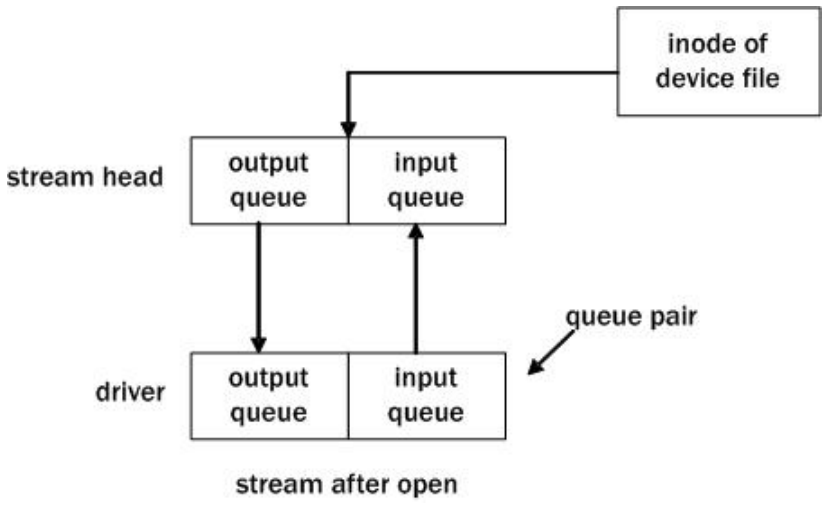
Svaki par redova je struktura podataka koja sadrži sledeće elemente:

- open proceduru (poziva se za vreme sistemskog poziva open)
- close proceduru (poziva se za vreme sistemskog poziva close)
- put proceduru (poziva se da prosledi poruku u red (queue))
- service proceduru (poziva se kad se red-queue izabere za izvršenje)
- pokazivač na sledeći red (queue) u streams strukturi
- pokazivač na listu poruka koje čekaju servis
- flagove, gornju i donju granicu (high-water mark i low-water mark), koriste se za kontrolu toka (flow-control), raspoređivanje, održavanje stanja redova

Kernel alocira parove redova koji su susedni u memoriji tako da jedan član para reda može lako da nađe svog drugog člana para.

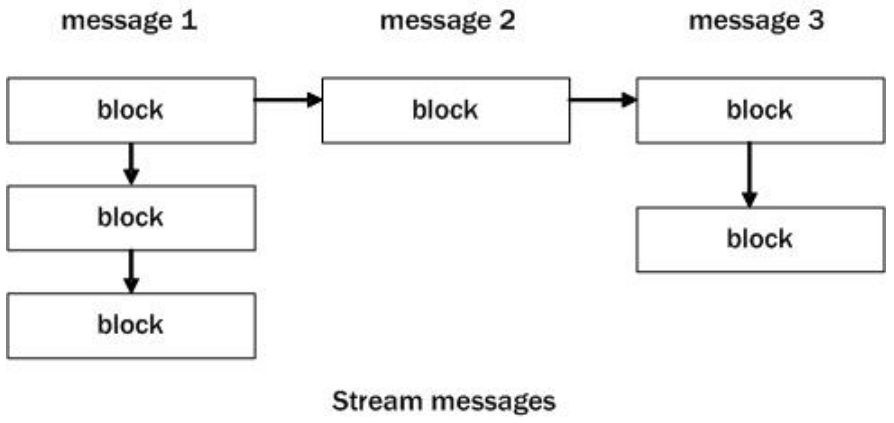
Streams je namenjen za karakter uređaje. Postoji specijalno polje u karakter prekidačkoj tabeli koje ukazuje na streams inicijalizacionu strukturu koja sadrži adresu rutine i gornju i donju granicu (high-water mark i low-water mark). Kada kernel izvršava sistemski poziv open i otkrije da je to karakter specijalna datoteka, ispituje se karakter switch tabela, da li je reč o streams drajveru ili ne, a na bazi polja za taj ulaz. Ako nije stream drajver, kernel prati običnu proceduru za karakter uređaje. Pri prvom open pozivu za streams drajver, kernel alocira par redova, jedan par za početak streams strukture (streams-head) i drugi par za drajver. Početni par (streams-head) je modul identičan za sve streams strukture: imaji put i service proceduru i interfeje ka višim I kernelskim modulima koji implementiraju sistemske pozive read, write i ioctl. Kernel inicijalizuje drajversku queue strukturu, dodeljujući queue-pokazivače i kopirajući adrese drajverskih rutina iz drajverskih inicijalizacionih struktura. Na kraju, drajver poziva open proceduru. Drajver open procedura obavlja običnu inicijalizaciju ali čuva informaciju da pozove red sa kojim je pridružen. Na kraju, kernel dodeljuje specijalan pokazivač na in-core inode strukturu da ukazuje na početni (stream-head) par redova, što je prikazano na slici 11.10. Kada drugi proces otvara uređaj, kernel nalazi prethodno alocirani streams preko inode pokazivača i poziva open proceduru svih modula u streams strukturi.

Moduli komuniciraju prosleđivanjem poruka sa susednim modulima. Poruka se sastoji od povezanih lista zaglavlja za poruke, MBH (message block headers). Svaki MBH ukazuje na početnu i krajnu lokaciju blokova podataka. Postoje dva tipa poruka: kontrolne i data-poruke, a tip se nalazi u identifikatoru MBH. Kontrolne poruke su rezultat sistemskog poziva ioctl ili specijalnih uslova, kao što je teminal hangup, a data poruke su posledica sistemskog poziva write ili read, kada imamo dolazak podataka sa uređaja.



Slika 10.10. *Stream drajver posle otvaranja*

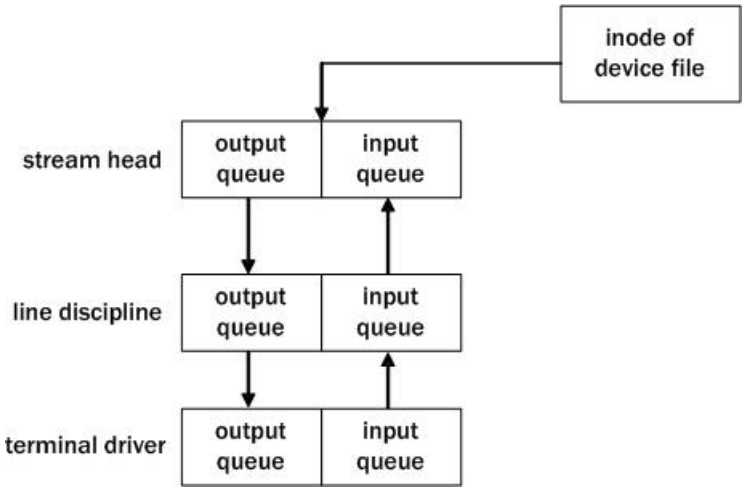
Kada proces upisuje u streams, kernel kopira podatke iz korisničkog prostora u blokove poruke koje su alocirane za početni par redova (stream-head). Stream-head modul poziva "put" proceduru od sledećeg queue modula, koji može procesirati poruku, proslediti je direktno u sledeći queue ili je izbaciti iz queue za kasnije procesiranje, kada modul linkuje MBH na povezanu listu. Na taj način formira se dvostruko (two-way) povezana lista poruka, kao na slici 10.11.



Slika 10.11. *Dvostruko povezana lista poruka u streams drajveru*

Zatim se setuje zastavica u queue data strukturi tog modula, da ukazuje da postoje podacie za obradu i servisiranje. Modul plasira red na povezanu listu redova, koji zahtevaju servis i poziva mehanizam za raspoređivanje, a taj raspoređivač poziva servisne procedure za svaki red u listi. Kernel može izabrati modul preko softverskog prekida, slično kao što poziva callout tabelu, a softverski prekidni handler poziva individualne servisne procedure.

Proces može gurnuti modul na otvoreni streams, preko sistemskog poziva ioctl. Kernel ubacuje modul ispod stream-head i konektuje pokazivače na redove (queue pointers) da čuvaju strukturu dvostruko povezane liste. Niži moduli streams strukture ne obraćaju pažnju da li komuniciraju sa početnim ili umetnutim modulom zato što je interfejs "put" procedura od sledećeg reda na streamsu, a sledeći red (queue) postaje ubačeni modul. Na primer, proces može ubaciti modul discipline linije u terminalski streams drajver da obrađuje kill i erase karakter, kao na slici 10.12.



Slika 10.12. *Disciplina liije kao modul*

Na ovakav način disciplina linije nema isti interfejs kao prethodno opisana, ali joj je funkcija ista. Bez modula discipline linije, terminal drajver ne procesira ulazne karaktere tako da oni neizmenjeni stižu do stream-head reda. Program koji otvara terminal i ubacuje modul discipline linije može da izgleda ovako:

```
fd=open ("/dev/ttyxy", O_RDWR);
ioctl(fd, PUSH, TTYLD);
```

Nema ograničenja koliko se modula može ubaciti (pushed) u streams. Proces može izbaciti modul iz streams (pop) u LIFO poretku, koristeći drugi sistemski poziv ioctl

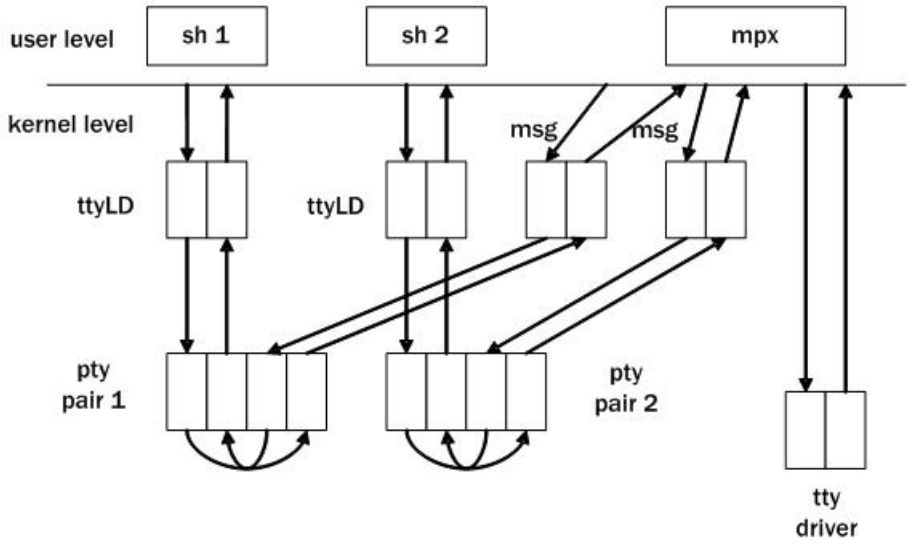
```
ioctl(fd, POP, 0);
```

Modul discipline linije se može primenjivati i na mrežne interfejse, umesto na terminale, tako da modul ispod nje može da bude mrežni drajver.

Primer za streams

Pick je opisao šemu za realizaciju virtuelnih terminala koristeći streams. Korisnik vidi više virtuelnih terminala, svaki zauzima poseban prozor na fizičkom terminalu. Pickova šema radi na grafičkim terminalima, ali je primenjljiva na običnim terminalima. Svaki window može okupirati čitav ekran, korisnik ima kontrolnu sekvencu za prebacivanje između virtuelnih prozora.

Slika 10.13 prikazuje uređenje procesa i kernelskih modula.



Slika 10.13. *Proces mpx*

Korisnik poziva proces mpx, koji kontroliše fizički terminal. Mpx čita fizičku terminalsku liniju i čeka na notifikaciju kontrolnih događaja kao što je kreiranje novog prozora, prebacivanje na drugi prozor itd. Kada se primi notifikacija da korisnik želi da kreira novi prozor, mpx kreira proces za kontrolu novog prozora i za komunikaciju sa pseudo-terminalom (pty). Pty je softverski uređaj koji radi u parovima, izlaz koji je upućen na jednog člana para se šalje na ulaz drugog člana u paru, dok se ulaz šalje ka višem modulu. Da bi postavio prozor, mpx alocira pty par i otvara (open) jedan član od njega, uspostavljajući streams ka njemu. Mpx obavlja fork, i novi proces otvara drugi član u pty paru. Mpx ubacuje message modul na njegov pty streams za konverziju kontrolnih poruka u data poruke, a dete proces gura modul discipline linije u svoj pty streams pre nego što obavi sistemski poziv exec za shell. Shell se sada izvršava na virtuelnom terminalu.

Proces mpx je multiplekser, on prosleđuje izlaz sa virtuelnih terminala na fizički terminal, a demultipleksira ulaz sa fizičkog terminala na odgovarjući pseudo terminal. Mpx čeka na dolazak podataka sa bilo koje linije (pty i fizički) preko select sistemskog poziva. Kada podaci dođu sa fizičkog terminala, mpx određuje da li je to kontrolna poruka za kreiranje novog pseudo terminala ili brisanje postojećeg, ili je to data-poruka poslata procesu da čita ili piše na terminal, a u njoj postoji zaglavlje za koji se pty odnosi. Na bazi toga, mpx prosleđuje poruku na odgovarajući pty stream. Pty drajver rutira podatke kroz disciplinu linije do procesa koji čita. Inverzna procedura nastaje kada proces upisuje u pseudo terminal, mpx će preko streams, dobiti podatke i poslati na fizički terminal.

Ako proces pošalje ioctl na virtuelni terminal, to se u modulu discipline linije tog streamsa postavlja nezavisno.

```
mpx
/*assume file desctriptors 0 i 1 already refer to physical tty*/

for(;;) /*loop*/
  select(input); /*wait for some line with input*/
  read input line;
  switch(line with input data)
  {
  case physical tty; /* input on physical tty line*/
    if(control command) /* eg. create new window*/
    {
      open a free pseudo-tty;
      fork a new process;
      if(parent)
      {
        push a msg discipline on mpx side;
        continue; /* back to for loop*/
      }
      /*child here*/
      close unnecesarry file descriptor;
      open other member of pseudo-tty pair,
      get stdin, stdout, stderr;
      push tty LD;
      exec shell; /*looks like virtual tty*/
      }
    /*regular data from tty coming up for virtual tty*/
    demultiplex data read from physical tty,
    strip off headers and write to approupriate pty;
    continue; /*back to loop*/
  case logical tty; /* a virtual tty is writing a window */
    encode header indicating what window data is for;
    write header and data to physical tty;
    continue;  /* back to for loop*/
  }
}
```

Analiza streams struktura

Moduli za steams se realizuju kao softverski prekidi, a ne kao posebni procesi pa ne prolaze kroz CPU raspoređivanje. Oni su uvek u kontekstu svog procesa, ne mogu spavati sami za sebe, nego mogu uspavati svoj proces. Moduli čuvaju svoje stanje interno.

Neke anomalije se mogu pojaviti i u realizaciji streams struktura:

- process accounting je dosta težak sa streams strukturama
- bez streams struktura, korisnik može lako prebaciti terminal u raw mod i vratiti se brzo ako nema raspoloživih podataka, što nije lako realizovati u steams drajveru
- streams su linearne konekcije i nije lako dozvoliti multiplkesiranje u kernelu (mpx je multipleksiranje na korisničkom nivou)

Bez obzira na anomalije, streams su moderan i kvalitetan koncept u realizaciju device-drajvera.

11

UNIX IPC

11.1. Uvod u UNIX IPC

IPC (Interprocess communication) komunikacioni mehanizam dozvoljava procesima da razmenjuje podatke i da sinhronišu izvršavanje. Već smo razmatrali razne forme IPC kao što su pipe datoteke, imenovane (named) pipe datoteke i signali. Pipe datoteke (unamed) imaju nedostatak pošto za njih znaju samo procesi koji su naslednici procesa koji je obavio sistemski poziv pipe, ostali procesi ne mogu pristupati toj pipe datoteci, nema komunikacije. Iako imenovane pipe datoteke dozvoljavaju procesima koji nisu međusobno povezani da komuniciraju sa pipe datotekom, oni se ne mogu generalno koristiti preko mreže, kao što ne dozvoljavaju višestruke IPC komunikacije. Nemoguće je naterati imenovanu pipe datoteku da obezbedi privatni kanal za par komunikacionih procesa. Procesi mogu komunicirati slanjem signala preko sistemskog poziva kill, ali cela poruka je samo signal koji je u suštini samo jedan broj.

Ovo poglavlje opisuje druge forme IPC.

Počinje sa ispitivanjem mehanizma za praćenje procesa (tracing), gde jedan proces prati (trace) i kontroliše izvršavanje drugog procesa, a zatim se objašnjava System V IPC paket: poruke, shared memory i semafori.

Objašnjavaju se tradicionalni metodi za IPC na mreži i na kraju se daje korisnički (user-level) pregled za BSD socket mehanizam. Socket se može prevesti kao utičnica, ali mi ćemo koristiti izvorni engleski naziv.

Ovde se ne diskutuju mrežni protokoli, adresiranje i razni mrežni servisi.

Process tracing

UNIX sistem obezbeđuje primitivnu formu IPC za praćnje (tracing) procesa, koji je koristan za ispravljanje grešaka (debugging). Debuger ili kontrolni program, kao što je sdb, izvršava proces koga će da prati i kontroliše njegovo izvršavanje sa sistemskim pozivom ptrace. Preko ovog sistemskog poziva postavlja se i briše tačka prekida BP (break point) i čitaju se i upisuju podaci u virtuelni adresni prostor za oba procesa. Praćenje procesa (tracing) se sastoji od sinhronizacije kontrolnog (debuger) procesa i praćenog (traced) procesa i kontrole izvršavanja praćenog programa.

Primer za sistemski poziv ptrace

Navodimo strukturu kontrolnog (debuger) procesa:

```
if((pid= fork()) == 0)
{
/* child = traced process */
  ptrace(0,0,0,0);
```

```
  exec("name of traced  process here");
}
/* debugger process continues here*/
for(;;)
{
  wait ((int *) 0);
  read(input for tracing instructions)
  ptrace(cmd, pid, ….);
  if(quitting trace) break;
}
```

Ovaj kôd prikazuje tipičnu strukturu kontrolnog (debugger) programa. Kontrolni program izvršava proces dete, koje će pozvati sistemski poziv ptrace i kao rezultat toga, kernel postavlja trace bit u ulazu tabele procesa PT, za proces dete. Iza toga, dete izvršava program koji će biti praćen (traced). Na primer, ako korisnik želi da kontroliše program a.out, proces dete će pozvati program a.out sa sistemskim pozivom exec. Kernel izvršava sistemski poziv exec kao i obično, ali detektuje da je trace-bit setovan i šalje detetu trap signal. Kernel proverava signale kada se vrati iz sistemskog poziva exec, kao što proverava signale posle svakog sistemskog poziva, detektuje trap signal koji je sam sebi poslao i izvršava kôd za praćenje procesa (proces tracing) kao specijalan slučaj upravljanja signalom. Proces dete sa trace bitom, budi roditelja (debbuger proces) koji spava u sistemskom pozivu wait, a dete ulazi u specijalno trace stanje koje je slično sleep stanju i obavlja prebacivanje konteksta.

Tipično, roditelj (debbuger process) ulazi u korisničku (user-level) petlju, čekajući da ga probudi proces dete (traced program). Kada dete tj. praćeni program probudi roditelja tj. kontolni program (debugger), roditelj napušta sistemski poziv wait, čita ulazne komande i konvertuje ih u seriju sistemskhi poziva ptrace koji kontrolišu dete (traced program). Sintaksa za sistemski poziv ptrace je:

```
ptrace(cmd, pid, addr, data);
```

gde cmd predstavlja različite komande kao što su čitanje podataka, upis podataka, nastavak izvršavanja itd, pid je PID za praćeni program, addr je virtulena adresa u procesu detetu za čitanje ili upis, data je celobrojna vrednost odnosno broj za upis.

Kada se izvršava sistemski poziv ptrace, kernel proverava da li kontrolni program (debugger) ima proces dete sa tom pid vrednošću, da li je dete u praćenom (traced) stanju, a potom se koristi globalna trace struktura za prenos podataka između ta dva procesa. Ta trace struktura se zaključava, da bi se zaštitila od drugih tracing procesa, kopiraju se parametri cmd, pid, addr, data u strukturu podataka, budi se dete proces i postavlja u stanje spremnosti (ready-to-run), a zatim kontrolni proces (debbuger) spava dok se proces dete ne oglasi.

Kada dete nastavi izvršavanje (u kernelskom modu), proces dete obavlja odgovarujuću trace komandu, upisuje svoj odgovor u trace strukturu, a zatim budi roditelja (debuger). Zavisno od tipa komande, dete može ući ponovo u debug stanje i čekati na novu komandu ili dete može da nastavi izvršavanje. Kada proces roditelj

(debugger) nastavi izvršavanje, kernel čuva povratnu adresu od praćenog procesa, otključava trace strukturu i vraća kontrolu roditeljskom procesu.

Ako roditeljski proces (debugger) ne spava u sistemskom pozivu wait kada dete ulazi u trace stanje, on neće otkriti svoj praćeni proces dete (traced child), sve dok ne obavi sistemski poziv wait .

Analizirajmo sledeća dva programa koji se zovu trace (praćeni program) i debug (kontrolni program).

```
trace              -(a traced process)
int data[32];
main()
{
  int i;
  for(i=0; i <32; i++) printf("data[%d]=%d\n", i, data[i])
  printf("ptrace data addr 0x%x\n",data);
}

debug              - a tracing process
#define TR_SETUP 0
#define TR_WRITE 5
#define TR_RESUME 7
int addr;
main(argc, argv)
int argc;
char *argv[];
{
  int i, pid;
  sscanf(argv[1], "%x",&addr);
  if((pid=fork()) = = 0)
  {
    ptrace(TR_SETUP, 0, 0, 0)
    execl("trace", "trace",0)
    exit();
  }
  for(i=0; i <32; i++)
  {
    wait((int *) 0);
    /* write value of i into address addr in proc pid*/
    if(ptrace(TR_WRITE, pid, addr, i) = = -1) exit();
    addr + = sizeof(int);
  }
  /*traced process should resume execution*/
  ptrace(TR_RESUME, pid, 1, 0)
}
```

Redosled izvršavanja za ova dva programa je sledeći. Prvo se izbrši program trace na terminalu bez praćenja, polje data će imati vrednost nula. Proces prikazuje adresu za polje data i izlazi. Sada izvršimo program debug, sa parametrima koje je prikazao program trace. Proces debug čuva ulazni parametar u promenljivoj addr, kreira proces dete koje poziva sistemski poziv ptrace da učini sebe pogodnim za praćenje, a potom obavlja sistemski poziv exec da izvrši program "trace". Kernel šalje signal SIGTRAP detetu (proces trace) i na kraju sistemskog poziva exec proces dete ulazi u trace stanje, čekajući na komandu od procesa roditelja (proces debug). Ako proces debug spava u sistemskom pozivu wait, budi se i nalazi svoje praćeno proces-dete i vraća se iz sistemskog poziva wait. Program debug zatim poziva sistemski poziv ptrace, u svakom koraku petlje upisuje vrednost indeksa petlje na adresi addr, pa inkrementira promenljivu addr, a addr je adresa ulaza u "data" polju.

Poslednji sistemski poziv ptrace koji poziva debug proces, izaziva da proces dete nastavi aktivnost, a u to vreme data polje sadrži vrednosti od 0 do 31. Debugeri kao UNIX program sdb imaju pristup simboličkoj tabeli za praćene procesa iz koje mogu da se odrede adrese koje se koriste kao parametri za sistemske pozive ptrace .

Korišćenje ptrace za praćenje procesa je primitiva i pati od više nedostataka.

[1] Kernel mora da obavi četiri prebacivanja konteksta da prenese podatak između kontrolnog (debugger) i praćenog programa (traced): kernel obavlja prebacivanje konteksta u kontrolnom programu u sistemskom pozivu ptrace sve dok praćeni proces ne odgovori na upit. Zatim imamo prebacivanje konteksta u praćeni proces i prebacivanje konteksta iz praćenog procesa. Na kraju imamo prebacivanje konteksta u kontrolni proces (debugger) sa odgovorom na sistemski poziv ptrace. Overhead je neophodan, zato što kontrolni proces (debugger) nema načina da dobije pristup u virtuelnom prostoru praćenog procesa i zato je praćenje (tracing) spora tehnika.

[2] Debugger proces može da obavi praćenje (tracing) za više dece istovremeno, ali to se retko koristi. Debugger može da obavi praćenje samo svoje dece, a ako dete uradi fork, debugger nema kontrolu na proces unuče. Ako dete obavi sistemski poziv exec, program koji se će izvršavati je još uvek u debug stanju, ali mu debugger možda ne zna ime, pa je simboličko praćenje otežano.

[3] Debugger ne može da obavi praćenje (tracing) procesa koji se već izvršava, ako taj proces ne obavi sistemski poziv ptrace, tako da neki procesi treba da se prekinu i ponovo restartuju u trace modu.

[4] Nemoguće je da se obavi praćenje za setuid program, zato što korisnici mogu da ugroze bezbednost upisom svog adresnog prostora preko sistemskog poziva ptrace i obaviti ilegalnu operaciju. Na primer, ako setuid program obavi sistemski poziv exec za programa "privatefile", pametan korisnik može preko sistemskog poziva ptrace da obavi prepis imena datoteke u /bin/sh i da izvrši shell sa neautorizivanim privilegijama. Sistemski poziv exec ignoriše setuid bit ako se proces prati, da bi se zaštitio od korisnika, koji pokušava da prepiše adresni prostor setuid programa.

Killian je opisao drugačiju šemu za praćenje procesa, zasnovanu na sistemskom pozivu switch. Administrator aktivira (mount) pseudo sistem datoteka proc na /proc i korisnici identifikuju svoje procese preko PID i tretiraju ih kao datoteke od sistema datoteka proc . Kernel za sistem datoteka proc daje prava pristupa na bazi UID-a. Korisnici ispituju procese, čitajući datoteku u sistemu datoteka proc, a mogu da postave tačke prekida (breakpoint) upisom u proc datoteku. Sistemski poziv stat vraća razne statističke informacije o procesu. Sistem datoteka proc uklanja sve nedostatke sistemskog poziva ptrace. Prvo, debugger može preneti mnogo više podataka po jednom sistemskom pozivu nego ptrace. Drugo, debuger može da obavi praćenje za bilo koji proces, a ne samo za svoje dete. Treće, debugger može da obavi praćenje za bilo koji postojeći proces, a ne samo za one koji su se prethodno pripremili za praćenje. Samo superuser može da prati setuid programe.

System V IPC

UNIX System V IPC paket sastoji se od tri mehanizma:

[1] Poruke dozvoljavaju procesima da šalju formatirane nizove procesima

[2] Deljiva memorija (shared memory) dozvoljava procesima da dele delove svojih adresnih prostora

[3] Semafori dozvoljavaju procesima da sinhronišu svoja izvršavanja

Svaki mehanizam sadrži tabelu čiji ulazi opisuju sve instance mehanizma.

Svaku ulaz sadrži numerički ključ (key), čije je ime izabrano od strane korisnika.

Svaki mehanizam sadrži sistemski poziv get za kreiranje novog ulaza ili za dobijanje postojećeg ulaza i parametara za sistemski poziv koji uključuju ključ i zastavice (flags). Kernel pretražuje odgovarajuću tabelu za ulaz koji sadrži zadati ključ. Procesi mogu pozvati sistemski poziv get sa ključem IPC-PRIVATE da obezbede potpuno novi neiskorišćeni ulaz. Procesi mogu postaviti IPC_CREAT kao zastavicu (flag), da kreiraju novi ulaz ako takav sa zadatim ključem ne postoji. Takođe, mogu da forsiraju objavu greške postavljenjem IPC_EXCL i IPC_CREAT zastavica, ako takav ulaz već postoji. Sistemski poziv get vraća deskriptor koji se bira od strane kernela, a služi za korišćenje u drugim sistemskim pozivima i get je analogan sistemskim pozivima open i creat, kod sistema datoteka.

Za svaki IPC mehanizam, kernel koristi sledeću formulu da nađe indeks u tabeli sa strukturama podataka za odgovarajući deskrtiptor

- index = descriptor modulo (number of entries in the table)

Na primer, ako tabela za strukturu poruka sadrži 100 ulaza, deskriptori za ulaz 1 su 1, 101, 201. Kada proces uklanja ulaz, kernel uvećava deskriptor za broj ulaza u tabeli: inkrementirana vrednost postaje novi deskriptor za ulaz kada se ponovo pozove sistemski poziv get. Procesi koji pokušavaju da priđu ulazu sa svojim starim

desktriptorom dobijaju informaciju o grešci (fail). Na primer, ako je deskriptor dodeljen sa (ulazom 1) ima vrednost 201, pa se ukloni, kernel dodeljuje novi deskriptor sa vrednošću 301.

Svaki IPC ulaz ima zaštitnu strukturu, koja uključuje korisnički ID i grupni ID procesa koji je kreirao ulaz, a UID, GID i prava pristupa rwx se postavljaju preko sistemskog poziva control.

Svaku ulaz sadrži druge statusne informacije, kao što je PID procesa koji je zadnji ažurirao ulaz (send a message, receive a message, ..) i vreme kad se ažuriranje dogodilo.

Svaki mehanizam ima sistemski poziv "control", za postavljanje upita na ulaz, za postavljanje statusnih informacije ili za uklanjanje ulaza iz sistema. Kada proces postavi upit za status ulaza, kernel proverava da li proces ima r pravo i kopira podatke sa tog ulaza u korisnički prostor. Slično, da bi se postavili parametri ulaza, što zahteva upis, kernel proverava da li je to UID procesa kreatora ili je to UID=root, zato što upis može da obavi samo vlasnik ili root. Kernel kopira korisničke podatke u ulaz, postavljajući UID, GID, prava i druga polja zavisno od mehanizma. Kernel ne menja vlasnika ulaza, a samo vlasnik ili root mogu ukloniti ulaz.

IPC: poruke

Postoje četiri sistemska poziva za poruke:

- msget: vraća (ili kreira) deskriptor poruke MD (message descriptor) koji određuje red čekanja za poruke MQ (message queue) za korišćenje u drugim sistemskim pozivima
- msgctl: ima opcije za setovanje i čitanje parametra pridruženih deskriptoru poruke MD, a ima i opciju za uklanjanje deskriptora poruke MD
- msgsnd: šalje poruku
- msgrcv: prima poruku

msgget

Sintaksa za sistemski poziv msgget je:

```
msgqid = msgget(key, flag);
```

gde je msgqid deskriptor koji vraća sistemski poziv, ključ (key) i zastavice (flag) imaju semantiku opisanu kao za sistemski poziv get.

Kernel čuva poruke na povezanoj listi tj. redu čekanja (queue) po deskriptoru poruka i koristi msgqid kao indeks u polju redova zaglavlja poruka MQH (mesage queue headers). U dodatku IPC polja za prava pristupa, struktura reda čekanja sadrži sledeća polja:

- pokazivač na prvu i poslednju poruku u povezanoj listi
- broj poruka i ukupan broj data bajtova u povezanoj listi
- maksimalni broj bajtova podataka koji mogu biti u povezanoj listi
- PID za poslednje procese koji su slali ili primali poruke
- vremensku oznaku TS (time stamps) poslednjih msgsnd, msgrcv i msgctl operacija

Kada korisnik pozove sistemski poziv msgget za stvaranje novog deskriptora, kernel pretražuje red čekanja za poruke (message queues) da bi se odredilo da postoji neki ulaz taj ključ. Ako nema ulaza za zadati ključ, kernel alocira novu strukturu u redu čekanja, inicijalizuje je i vraća korisniku identifikator msgqid. U protivnom, ako postoji ulaz sa tim ključem, proveravaju se prava pristupa i vraća se postojeći identifikator.

msgsnd

Proces koristi sistemski poziv msgsnd za slanje poruke:

```
msgsnd(msgqid, msg, count, flag);
```

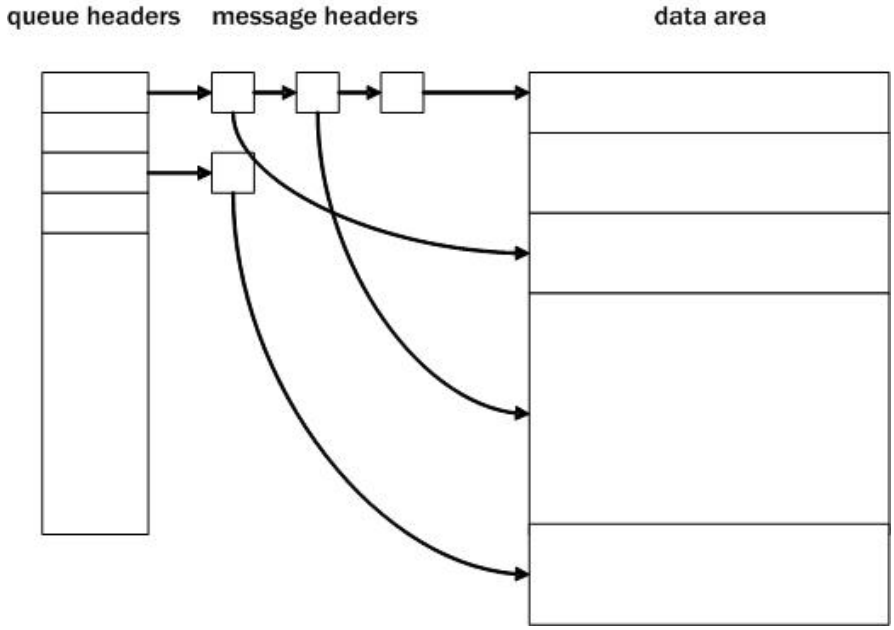
gde je msgqid deskriptor koji vraća sistemski poziv msgget, msg je pokazivač na strukturu koja se sastoji od korisnički definisane strukture koja ima integer tip poruke i polje od karaktera, count daje veličinu poruke, a zastavica flag specificira akciju koju kernel preuzima ako se prekorači veličina bafera.

```
algorithm msgsend /* send a message*/

input: (1) message queue descriptor
       (2) address of message structure
       (3) size of message
       (4) flags
output: number of byte sent

{
  check legality of descriptors, permission;
  while(not enough space to store messages)
  {
    if(flags specify not to wait) return;
    sleep(until event enough space available);
  }
  get message header;
  read message text from user space to kernel;
  adjust data structure: enqueue message header;
  message header points to data, count, time stamps, PID;
  wakeup all processes waiting to read message from queue;
}
```

Kernel proverava da li proces koji šalje poruku ima w pravo na deskriptor poruke MD, da li dužina poruke ne prevazilazi sistemsko ograničenje, da li red čekanja poruka MQ

(message queue) ne sadrži suviše bajtova i da li je tip poruke +integer. Ako svi testovi prođu, kernel alocira prostor za poruku iz mape poruka i kopira podatke iz korisničkog prostora. Kernel alocira zaglavlje poruke MH i stavlja ga na kraj povezane liste tj. reda čekanja za zaglavlja MQH. Zatim upisuje tip poruke i veličinu u zaglavlje poruke MH, postavlja zaglavlje poruke MH da ukazuje na samu poruku i ažurira različita statistička polja (broj poruka i bajtova u MQ, TS i uid sendera) u zaglavlju reda čekanja (queue header). Kernel zatim budi procese koji spavaju, čekajući da poruke stignu u red čekanja poruka MQ. Ako broj bajtova prekorači granicu za taj red čekanja (queue limit), proces spava sve dok se druge poruke ne izbace iz reda čekanja. Ako proces specificira flag IPC_NOWAIT, on se vraća sa porukom o grešci, ali bez uspavljivanja.

Slika 11.1 prikazuje poruke na redu čekanja MQ (message queue), prikazuje zaglavlja poruka MQH (message queue headers), povezanu listu zaglavja MH (message headers), pokazivače iz zaglavlja poruka MH u područje podataka tj. samih poruka.



Slika 11.1. *Sistem poruka*

Primer za msgsnd

Posmatrajmo sledeći program: proces poziva sistemki poziv msgget za dobijanje deskriptora za MSGKEY. Zatim se setuje poruka dužine 256 bajtova, mada se koristi samo prvi integer, kopira se PID u text poruke, dodeljuje se tip poruke d abude 1 i poziva se sistemski poziv msgsnd da pošalje poruku.

```
client process
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MSGKEY  75
struct msgform
{
  long mtype;
  char  mtext[256];
} msg;
main()
{
  struct msgform msg;
  int msgid, pid, *pint;
  msigid = msgget(MSGKEY, 0777);
  pid = getpid();
  pint = (int *) msg.mtext;
  *pint=pid; /*copy pid into message text*/
  msg.mtype=1;
  msgsnd(msgid, &msg, sizeoff(int),0);
  msgrcv(msgid, &msg, 256, pid,0) /*pid is used as msg type*/
  printf("client: receive from pid %d", *pint);
}
```

msgrcv

Proces prima poruku preko sistemskog poziva msgrcv

```
count = msgrcv(id, msg, maxcount, type, flag)
```

gde je id deskriptor poruke, msg je pokazivač na korisničku strukturu koja će sadržavati primljenu poruku, maxcount je veličina polja podatka u msg strukturi, a zastavica flag specificira akciju koju kernel preuzima ako poruka nije u redu čekanja MQ. Povratna vrednost, count, je broj bajtova koje je korisnik zaista primio.

Kernel proverava da li korisnik ima potrebna prava redu čekanja za poruke MQ. Ako je polje type jednako nula, kernel nalazi i uzima prvu poruku u povezanoj listi. Ako je veličina poruke manja ili jednaka veličini maxcount, tada kernel kopira poruku u korisnički prostor tj. u strukturu msg, a podešava red čekanja poruka MQ: dekremetira broj poruka u redu čekanja, setuje vreme prijema i PID procesa koji je primio poruku, podešava povezanu listu i oslobađa prostor koji je zauzimala poruka. Iza toga kernel budi sve procese koji su čekali da se oslobodi mesto u redu čekanja poruka MQ. Ako je poruka veća od maxcount, kernel vraća grešku u sistemskom poziv, i ostavlja poruku i dalje u redu čekanja . Ako proces ignoriše ograničenja za veličinu, (bit MSG_NOERROR je setovan u flag), kernel odseca poruku, vraća zahtevani broj bajtova i uklanja celu poruku iz reda čekanja poruka MQ.

```
algorithm msgrcv /* receive a message*/

input:  (1) message descriptor
        (2) address of data array for incoming message
        (3) size of data array
        (4) requested message type
        (5) flags
output: number of byte in returned message

{
  check permissions;
  loop:
  check legality of message descriptor;
  /*find message to return to user*/
  if(requested message type = = 0)
    consired first message on queue;
  else if(requested message type > 0)
    consired first message on queue with given type;
  else if(requested message type < 0)
    consired first of lowest message on queue;
    such that its type is <= absolute value of requested type;
  if(there is a message)
  {
      adjust message size or return error if user size is too
small;
    copy message type, text from kernel space to user space;
    unlink message from queue;
    return;
  }
  /*no message*/
  if(flag specify not to sleep) return with error;
  sleep(event message arrives on queue);
  goto loop;
}
```

Proces može primiti poruku odgovarujućeg tipa, postavljanjem type polja u formatu za sistemski poziv msgrcv. Ako je tip pozitivni ceo broj (+integer), kernel vraća prvu poruku datog tipa. Ako je tip negativni ceo broj (–integer), kernal nalazi najniži tip svih poruka u redu čekanja poruka MQ, čiji je tip manji od apsolutne vrednosti zadatog polja type. Na primer, ako MQ sadrži tri poruke čiji su tipovi 3, 1 i 2, a proces hoće prijem poruke sa -2, tada će uzeti poruku tipa 1. Ako nijedna poruka ne zadovolji zahtev za prijem, kernel gura proces na spavanje osim ako nije postavljen flag IPC_NOWAIT.

Analizirajmo programe client proces i server proces. Program server proces, prikazuje strukturu servera koji obezbeđuje servis klijent procesima. Na primer, mogu se primati zahtevi od klijentskih procesa za database informacije. Server proces je jedna od tačaka za pristup u bazu podataka. Server kreira strukturu za poruke postavljanjem IPC_CREAT

flaga u sistemskom pozivu mssget i prima sve poruke čiji je tip 1 od klijentskih procesa. Zatim čita text poruke, nalazi PID klijentskog procesa, i setuje tip povratne poruke za klijentski proces sa tim PID-om. Na taj način, serverski proces prima samo poruke od klijentskog procesa, a klijentski proces prima samo poruke koje je baš za njega poslao serverski proces. Procesi u tom slučaju idu na na isti red čekanja poruka MQ.

```
server  process
#include <sys/types.h> #include <sys/ipc.h>
#include <sys/msg.h> #define MSGKEY  75
struct msgform
{
  long mtype;
  char mtext[256];
} msg;
main()
{
  int i, pid, *pint;
  extern cleanup();
  for(i=0; i<20; i++)
  signal(i, cleanup);
  msigid = msgget(MSGKEY, 0777 | IPC_CREAT);
  for(;;)
  {
    msgrcv(msgid, &msg, 256, 1,0)
    pint = (int *) msg.mtext;
    pid = *pint;
    printf("server. receive from pid %d", pid);
    msg.mtype = pid;
    *pint = getpid();
    msgsnd(msgid, &msg, sizeoff(int),0)
  }
}
cleanup()
{
  msgctl(msgid, IPC_RMID,0);
  exit();

}
```

Poruke se formatiraju u "type-data" parovima. Type je prefiks koji dozvoljava procesima da selektuju poruke posebnog tipa. Procesi mogu da ekstrakuju poruke partikularnog tipa iz reda čekanja poruka MQ u poretku u kome su stigle, a kernel održava poredak. Mada je moguće realizovati sistem poruka (message passing) na korisničkom nivou u sistemu datoteka, poruke su kernelske memorijske strukture, brže su i fleksibilniije za IPC.

Proces može zahtevati upit za status deskriptora poruke MD, da postavi status ili da ukloni poruku iz reda čekanja poruka MQ preko sistemskog poziva msgctl, čija je sintaksa:

```
msgctl(id, cmd, mstatbuf)
```

gde je id deskriptor poruke, cmd specificira tip komande, mstatbuf je adresa korisničke strukture koja sadrži kontrolne parametre ili mesto za smeštanje rezultata upita.

Vratimo se na program serverski proces, proces hvata signal i poziva funkciju cleanup da ukloni red čekanja poruka MQ iz sistema. Jedino ako primi SIGKILL koga ne može da uhvati, red čeknja poruka MQ će ostati u sistemu čak i kad nema više poruka i procesa koji ukazuju na njega.

11.2. IPC: Deljiva memorija

Procesi mogu komunicirati direktno jedan sa drugim deljenjem dela njihovog virtuelnog adresnog prostora, a zatim čitanjem i pisanjem u tu deljivu memoriju.

Sistemski pozivi za deljivu memoriju (SharedMemory) su slični sistemskim pozivima za sistem poruka. To su:

[1] shmget: kreira novi region deljive memorije ili vraća jedan postojeći region deljive memorije

[2] shmat: logički priključuje (attach) region u virtuelni adresni prostor procesa

[3] sgmdt: izbacuje region (detach) region iz virtuelnog adresnog prostora procesa

[4] shmctl: manipuliše različitim parametrima za deljivu memoriju

Procesi pristupaju deljivoj memoriji istim memorijskim instrukcijama kao i za običnu memoriju. Posle priključivanja (attach) regiona deljive memorije ona postaje deo virtuelnog adresnog prostora procesa, nema posebnih sistemskih poziva za pristup deljivoj memoriji.

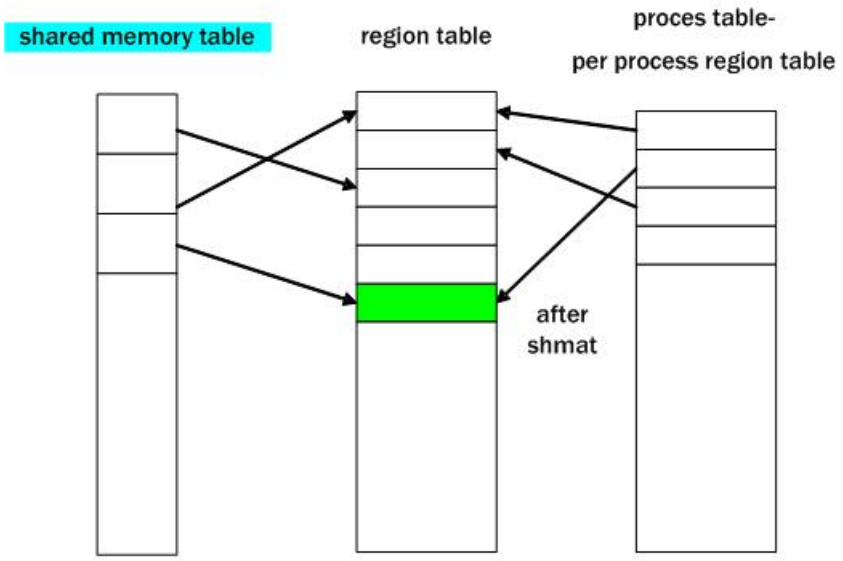
shmget

Sintaksa za sistemski poziv shmget je:

```
shmid = shmget(key, size, flag);
```

gde je size broj bajtova u regionu. Kernel pretražuje tabelu deljive memorije SMT (shared memory table) za zadati ključ. Ako nađe ključ i prava pristupa su u redu, proces će dobiti vrednost shmid kao deskriptor za region deljive memorije. Ako ne nađe ključ, a korisnik je setovao IPC_CREAT flag da kreira novi region, kernel verifikuje da je zadata

veličina između minimalne i maksimalne vrednosti koju UNIX dozvoljava, zatim alocira region podataka preko algoritma allocreg. Kernel čuva prava pristupa, veličinu i pokazivač na ulaz u tabeli regiona RT u tabeli deljive memorije SMT i postavlja zastavicu-flag da ukazuje da još nema memorije koja je dodeljena regionu Memorija se alocira tj. tabele stranica se kreiraju kada se region priključi u proces (attach). Na slici 11.2 prikazane su strukture podataka za deljivu memoriju.



Slika 11.2. *Strukure podataka za deljivu memoriju*

Kernel takođe setuje flag u ulazu region tabele RT da ukazuje da se region ne oslobađa kada proces obavi sistemski poziv exit. Zato, regioni deljive memorije ostaju alocirani čak i kad više nema procesa vezanih za njih.

shmat

Proces priključuje (attach) region deljive memorije u svoj adresni prostor za sistemski poziv shmat :

```
virtaddr = shmat(id, addr, flags);
```

Parametar id je id vraćen od prethodnog sistemskog poziva shmget, id identifikuje region deljive memorije, addr je virtuelna adresa u koju korisnik želi da priključi (attach)

region deljive memorije, zastavice flags specificiraju da li region read-only i da li kernel treba da zaokruži (round-off) korisnički-specificiranu adresu. Povratna vrednost, virtaddr, je virtuelna adresa gde kernel priključio region, a ne mora da bude jednaka zadatoj adresi addr.

Kada se izvršava sistemski poziv shmat, kernel proverava da li proces ima potrebna prava za pristup regionu. Potom se ispituje addr i ako je taj parametar nula kernel bira podesnu virtuelnu adresu.

```
algorithm shmat /* attach shared memory */

input:  (1) shared memory descriptor
        (2) virtual address to attach memory
        (3) flags
output: virtual address where memory was attached

{
  check validity of descriptors, permissions;
  if(user specified virtual address)
  {
    round off virtual address, as specified by flags;
    check legality of virtual address, size of region;
  }
  else /* user wants kernel to find good address*/
      kernel picks virtual address: error if none available;
    attach region to process address space (algortihm attachreg);
    if(region being attached for the first time)
      allocate page tables, memory for region (alortigm growreg);
    return(virtual address where atached);
}
```

Region deljive memorije ne sme da se preklapa sa drugim regionima u procesovom virtuelnom adresnom prostoru, a mora se pažljivo birati da drugi regione ne porastu u region deljive memorije. Na primer, proces može uvećati svoj region podataka sistemskim pozivom brk, a novi region se nastavlja na stari, pa se zato region deljive memorije ne dodeljuje iznad regiona podataka ili iznad stek regiona. Ako stek raste na gore, povoljno je region deljive memorije postaviti odmah ispod steka.

Kernel proverava da li deljivi region SM leži u adresnom prostoru procesa, a potom priključuje (attach) region u proces. Ako prozvani proces prvi priključuje taj deljivi region, kernel alocira neophodne tabele, zatim poziva growreg, podešava polje ulaza u SM tabeli, upisujući vreme priključenja, "last time attached" i vraća virtuelnu adresu u kojoj je priključen region.

shmdt

Proces izbacuje (detach) region deljive memorije iz adresnog prostora procesa preko sistemskiog poziva shmdt.

```
shmdt(addr)
```

gde je addr virtuelna adresa koju je prethodno vratio sistemski poziv shmat, a virtuelna adresa se koristi jer proces može imati više regiona deljive memorije u svom adresnom prostoru. Kernel pretražuje procesov region priključen na toj virtuelnoj adresi i skida ga sa detachreg algoritmom, a takođe se koristi i tabela deljive memorije SM.

Primeri za deljivu memoriju

Analizirajmo sledeći program. Proces kreira region deljive memorije od 128K, obavlja priključivanje (attach) dva puta u svoj adresni prostor, ali na različitim adresama. Proces upisuje podatke u prvi region deljive memorije, a čita iz drugog regiona deljive memorije.

Primer1.Priključivanje deljive memorije 2 puta u isti proces

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHMKEY  75
#define K        1024
int shmid;
main()
{
  int i, *pint;
  char *addr1, *addr2;
  extern char *shmat();
  extern cleanup();
  for(i=0; i<20; i++) signal(i, cleanup);
  shmid = shmget(SHMKEY, 128*K, 0777 | IPC_CREAT);
  addr1 = shmat(shmid,0,0);
  addr2 = shmat(shmid,0,0);
  printf("addr1 0x%x addr2 0x%x ", addr1, addr2);
  pint = (int *) addr1;
  for(i=0; i<256; i++) *pint++ = i;
  pint = (int *) addr1
  *pint = 256;
  pint = (int *) addr2;
  for(i=0; i<256; i++) printf("index %d value %d", i,*pint++ );
  pause();
}
cleanup()
{
  shmctl(shmid, IPC_RMID,0);
  exit();
}
```

Drugi proces (primer2) priključuje (attach) isti region deljive memorije, ali uzima samo 64K, što pokazuje da proces može da priključi različite veličine regiona deljive memorije.

Drugi proces čeka dok prvi proces ne upiše nenultu (non-zero) vrednost u prvu reč regiona deljive memorije i tada čita deljivu memoriju. Prvi proces, pauzira i daje šansu drugom procesu da se izvršava.

Kada prvi proces uhvati signal, on uklanja region deljive memorije.

Primer 2. Deljiva memorija među procesima

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define SHMKEY 75
#define K 1024
int shmid;
main()
{
  int i, *pint;
  char *addr;
  extern char *shmat();
  char  mtext[256];
  shmid = shmget(SHMKEY, 64*K, 0777);
  addr = shmat(shmid,0,0);
  pint = (int *) addr;
  while (*pint == 0);
  pint = (int *) addr;
  *pint = 256;
  for(i=0; i<256; i++) printf("%d", *pint++ );
}
```

Sistemski poziv shmctl

Proces koristi sistemski poziv shmctl, da traži upit statusa ili da postavi parametre za region deljive memorije:

```
shmctl(id, cmd, shmstatbuf);
```

gde id identifikuje ulaz u tabelu deljive memorije, cmd specificira tip komande, shmstatbuf je adresa korisničke strukture koja sadrži statusne informacije ulaza u tabeli deljive memorije SM, kada se pita ili postavlja status za taj region deljive memorije. Kernel tretira komande za upit statusa i promenu vlasništva slično kao kod poruka. Kada se ukljanja region deljive memorije, kernel analizira ulaz u tabeli regiona RT, pa ako nema drugih procesa sa priključenim tim regionom, oslobađa se RT ulaz sa svim resursima, preko algoritma freereg. Ako neki proces ima priključen region deljive memorije, broj referenci biće različit od nule, kernel briše flag koji ukazuje da region ne

treba da se oslobodi kada ga poslednji proces izbaci iz svog adresnog prostora (deatch). Procesi koji još uvek koriste region mogu još da ga koriste, ali drugi procesi ne mogu više da ga priključe, zato što je jedan proces tražio njegovo brisanje. Tek kada svi procesi otkače region, on se oslobađa (slično kao open pa unlink, odmah).

Semafori

Sistemski poziv semafor omogućava procesu da sinhroniše izvršavanje pomoću skupa atomskih operacija na skupu semafora. Pre implementacije semafora, koristio se princip zaključavanja datoteke. Proces treba da kreira datoteku za zaključavanje (lock file) sa sistemskim pozivom creat, ako želi da zaključa resurs, a sistemski poziv create će otkazati ako datoteka već postoji, a zbog toga će proces pretpostaviti da je drugi proces već zaključao resurs. Glavni nedostatak ove šeme je što proces ne zna kada da pokuša ponovo, a datoteke za zaključavanje mogu da ostanu u sistemu posle sistemske havarije.

Semafor ima dve atomske operacije P i V. P (wait) operacija dekrementira vrednost semafora, samo ako je njegova vrednost veća od nule. V (signal) operacija inkrementira vrednost semafora. Kako su P i V atomske, samo jedna opracija može da se obavlja u jednom trenutku. P i V operacije su atomske za kernel, nijedan drugi proces ne može ništa da radi sa semaforom dok drugi ne obavi operaciju, a ako ne može da je obavi operaciju, proces ide na spavanje.

Semafor na UNIX System V sastoji se od sledećih elemenata:

- vrednost semafora
- PID zadnjeg procesa koji manipuliše semaforom
- broj procesa koji čekaju da se poveća vrednost semafora
- broj procesa koji čekaju da vrednost semafora postane nula

Sistemski pozivi za semafore su:

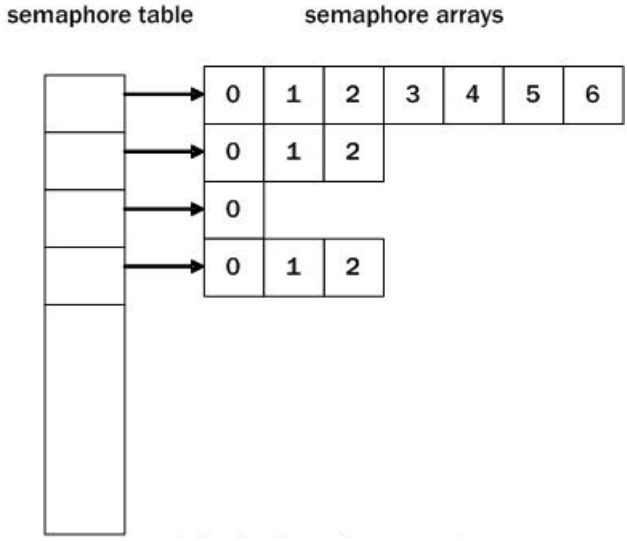
- semget: služi za kreiranje i dobijanje pristupa skupu semafora
- semctl: obavlja različite kontrolne operacije nad semaforima
- semop: manipuliše vrednostima semafora

Sistemski poziv semget

Sistemski poziv semget kreira polje semafora:

```
id=segmet(key, count, flag);
```

gde su key, count i flag parametri kao kod deljive memorije. Kernel alocira ulaz koji pokazuje na polje semaforskih struktura sa brojem od count elemenata, kao na slici 11.3.



Slika 11.3. *Strukture podataka za semafore*

Ulaz takođe specificira: broj semafora u polju, vreme poslednjeg sistemskog poziva semop, vreme poslednjeg sistemskog poziva semctl.

Sistemski poziv semop

Proces manipuliše semaforom preko sistemskog poziva semop.

```
oldval = semop(id, oplist, count);
```

gde je id deskriptor koji je vratio sistemski poziv semget, oplist je pokazivač na polje semaforskih operacija, a count je veličina polja. Povratna vrednost oldval je vrednost poslednjeg semafora nad kojim je operisano.

Format za svaki element opliste je:

- broj semafora koji identifikuje ulaz polja semafora nad kojim će se izvoditi operacija
- operacija
- zastavice (flags)

Kernel čita polje semaforskih operacija oplist, iz korisničkog adresnog prostora i verifikuje da li su semaforski brojevi legalni i da li proces ima pravo da čita ili da menja semafor.

```
algorithm semop /*semaphore operations*/
```

```
input: (1) semaphor descriptor
       (2) array of semaphor operation
       (3) number of elements in array
output: start value of last semphore operated on

{
  check legality of semaphore descriptor;
  start:
  read array of semaphore operations from user to kernel space;
  check permissions for all semaphore operations;
  for(each semaphore operation in array)
  {
    if(semaphore operation is positive)
    {
      add "operation" to semaphore value;
      if(UNDO flag set on semaphore operation)
        update process undo structure;
      wakeup all processes sleeping
        (event semaphore value increase);
    }
    else if (semaphore operation is negative)
    {
      if("operation" + semaphore value >= 0)
      {
        add "operation" to semaphore value;
        if(UNDO flag set on semaphore operation)
          update process undo structure;
        if (semaphore value 0) wakeup all processes sleeping
          (event semaphore value becomes 0;
        continue;
      }
      reverse all semaphore oeprations already
        done this SC(previous iterations);
      if(flags specify not to sleep) return with error;
      sleep (event semaphore value increase);
      goto start; /* loop from begginning*/
    }
    else /* semaphore operation is zero */
    {
      if(semaphore value non 0)
      {
        reverse all semaphore oeprations already
          done this SC(previous iterations);
        if(flags specify not to sleep) return with error;
        sleep  (event semaphore value increase);
        goto start; /* loop from begginning*/
```

```
        }
      }
    }/* for loop ends here*
    /* semaphore operations all succeeded*/
    update timestamps, PID;
    return value of last semaphore operated
      on before call succeeded;
}
```

Ako kernel mora da ide na spavanje dok radi listu semaforskih operacija, on vraća vrednosti semafora kao na početku sistemskog poziva, pa ide na spavanje dok se ne desi događaj, a onda se restartuje sistemski poziv semop. Zato što kernel čuva semaforske operacije u globalnom polju, iz koga čita parametre ako mora da restartuje sistemski poziv semop. Operacije su atomske, ili sve ili ništa.

Kernel menja vrednosti semafora u skladu sa vrednošću operacije:

- ako je vrednost pozitivna (+), to inkremntira semafor i budi sve procese koji čekaju da vrednost semafora poraste.
- ako je operacija nula, kernel proverava vrednost semafora, pa ako jeste nula, nastavlja sa drugim operacijama u polju, u protivnom inkrementira broj procesa koji spavaju čekajući da vrednost semafora postane nula, pa ide na spavanje.
- ako je vrednost negativna (-) i apsolutna vrednost je manja ili jednaka vrednosti semafora, kernel dodaje vrednost operacije (negativnu) na vrednost semafora. Ako je rezultat nula, kernel budi sve procese koji čekaju da rezultat postane nula. Ako je vrednost semafora manja od apsolutne vrednosti operacije, kernel gura proces na spavanje, a događaj koji ga budi nastaje kada se inkrementira vrednost semafora. Kada proces ode na spavanje u semafor operaciji, budi se na signal.

Primer za semafor

Analizirajmo program za " locking and unlocking semaphore operations" i nazovimo ga a.out i zatim izvršimo program tri puta na sledeći način:

[1] a.out &

[2] a.out a &

[3] a.out b &

Na primer, sistemski poziv semget u sledećem programu će kreirati semafor sa dva elementa.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#define SEMKEY  75
```

```
int semid;
unsigned int count;
/*definition of sembuf in file sys/sem.h
*struct sembuf {
*  unsigned shortsem_num;
*  short sem_op;
*  short sem_flg;
}*/
struct sembuf psembuf, vsembuf; /* ops for P and V*/
main(argc, argv)
int argc;
char *argv;
char *argv[];
{
  int i, first, second;
  short initarray[2], outarray[2];
  extern cleanup();
  if(argc == 1)
  {
    for(i=0; i<20; i++) signal(i, cleanup);
    semid = semget(SEMKEY, 2, 0777| IPC_CREAT);
    initarray[0]= initarray[1]=1;
    semctl(semid, 2, SETALL, initarray);
    semctl(semid, 2, GETALL, outarray);
    printf("sem init vals %d %d", outarray[0], outarray[1]);
    pause(); /* sleep until awakned by signal*/
  }
  else if (argv[1][0] == a)
  {
    first =0;
    second=1;
  }
  else
  {
    first =1;
    second=0;
  }
  semid = semget(SEMKEY, 2, 0777);
  psembuf.sem_op= -1;
  psembuf.sem_flg= SEM_UNDO;
  vsembuf.sem_op= 1;
  vsembuf.sem_flg= SEM_UNDO;
  for(count =0; ; count ++)
  {
    psembuf.sem_num= first;
    semop(semid, &psembuf,1);
```

```
      psembuf.sem_num= second;
      semop(semid, &psembuf,1);
      rintf("proc %d count %d", getpid(), count);
      vsembuf.sem_num= second;
      semop(semid, &vsembuf,1);
      vsembuf.sem_num= first;
      semop(semid, &vsembuf,1);
    }
  }
  cleanup()
  {
    semctl(semid, IPC_RMID,0);
    exit();
  }
```

U prvom slučaju program se pozove bez argumenata, tako da proces kreira semaforski set sa dva ulaza i inicijalizuje njihove vrednosti na 1. Zatim proces pauzira i spava, sve dok ga ne probudi signal, kada preko funkcije cleanup() briše semaforski skup.

Kada se program izvršava sa argumentom a, proces radi četiri semaforske operacije u petlji, dekremetira se vrednost semafora 0, dekremetira se vrednost semafora 1, izvršava se print naredba, a zatim se inkrementira semafor 1, pa semafor 0. Proces ide na spavanje ako pokušava da dekrementira vrednost semafora koja je 0, tako da se semafor tada smatra da je zaključan. Kako su semafori postavljeni na 1 i nema drugih procesa koji koriste semafor, proces sa argumentom a neće spavati nijedanom, i vrednost oba semafora će oscilovati, između 0 i 1.

Kada se program izvršava sa argumentom b, proces radi četiri semaforske operacije u petlji, dekremetira se vrednost semafora 0 i 1 u inverznom poretku od procesa sa argumentom a. Međutim ako procesi sa argumentom a i b rade simultano, situacija može da bude sledeća: proces sa argumetom a zaključa semafor 0, a hoće da zaključa semafor 1, ali je proces sa argumetom b je već zaključao semafor 1, a hoće da zaključa semafor 0 koji je već zaključan . Oba procesa idu na spavanje i nikada se ne bude. To je zastoj (deadlock), sve dok proces ne primi signal.

Da bi se izbegavali ovakvi problemi, procesi treba da koriste sledeće strukture i kôd (za ovaj primer):

```
struct sembuf psembuf[2]; /* ops for P and V*/
  psembuf[0].sem_num = 0;
  psembuf[1].sem_num = 1;
  psembuf[0].sem_op  = -1;
  psembuf[1].sem_op  = -1;
semop(semid, psembuf,2);
```

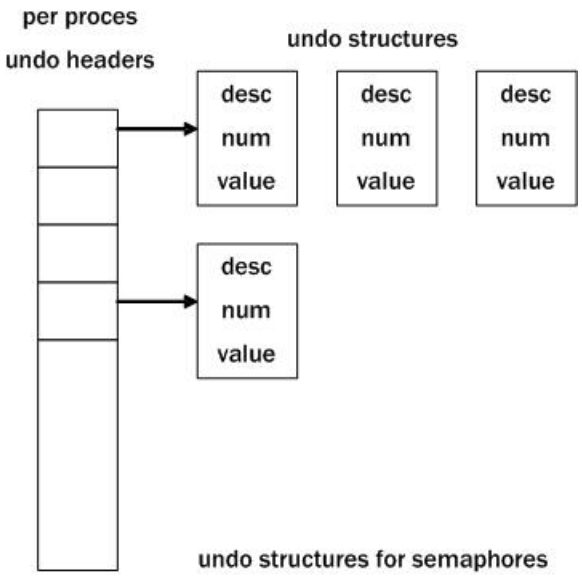
Psembuf je polje semaforskih operacija koje dekrementiraju semafore 0 i 1 istovremeno. Ako bilo koja operacija ne može da se izvrši, proces spava sve dok se obe

ne obave, a pre spavanja vraća sve vrednosti semafora koje je promenio. Na primer, ako je vrednost semafora 0 jednaka 1, a vrednost semafora 1 jenaka 0, kernel će ostaviti obe ove vrednosti sve dok ne bude mogao da dekrementira obe vrednosti.

Proces može setovati IPC_NOWAIT flag, pa ako proces mora da čeka na semaforu, trebalo bi da ide na spavanje, ali zbog ove zastavice kernel će se vratiti iz sistemskog poziva sa porukom o grešci. Zato je moguće implementirati uslovni semafor, gde proces ne ide na spavanje ako ne može da dobije atomsku akciju.

SEM UNDO strukture

Nepovoljne situacije mogu da se dogode, ako proces obavlja semaforsku operaciju, na primer zaključa semafor i obavi sistemski poziv exit a ne resetuje semaforsku vrednost. Takve situacije su programerska greška ili je signal ubio proces pre vremena. Ako u programu za "locking and unlocking semaphore operations", proces dobije kill signal nakon što dekrementira semaforske vrednosti, nema šanse da ih inkrementira ponovo jer kill signal ne može da se uhvati, a semafori ostaju zaključani. Da bi se izbegavale takve situacije, procesi mogu da setuju SEM_UNDO flag u sitemski poziv semop, a kada se on okonča kernel poništava efekte svake semaforske operacije koju je proces obavio. Da bi to mogao da radi, kernel ima tabelu za svaki proces u sistemu, gde svaku ulaz ukazuje na undo strukturu. Svaki semafor koji proces koristi ima svoju undo strukturu. Ta struktura se sastoji od tripleta koga čine semafor ID, semaforski broj u skupu i vrednost, kao na slici 11.4



Slika 12.4. *NDO strukture za semafore*

Kernel alocira undo strukturu dinamički, kada proces izvršava prvi sistemski poziv semop sa SEM_UNDO zastavicom. Na sledeći sistemski poziv semop sa SEM_UNDO flagom, kernel pretražuje tabelu za undo strukturu za semafor sa tim IDom i tada oduzima semaforsku operaciju od vrednosti u tabeli, tako da undo struktura sadrži negativan zbir svih semaforskih operacija koje su obavljene sa zastavicom SEM_UNDO. Ako undo-struktura ne postoji, kernel je kreira, a kada vrednost u undo-strukturi padne na nulu, kernel ukljanja tu undo-strukturu. Kada proces obavi exit, kernel poziva specijalnu strukturu koja obavlja akciju na semaforu.

Vratimo se opet na program za "locking and unlocking semaphore operations", gde imamo sukcesivne operacije P i V za isti semafor. Kernel kreira undo strukturu svaku put kad proces dekrementira semaforsku vrednost, a ukljanja strukturu svaki put kad proces inkrementira semaforsku vrednost, zato što je tada vrednost u undo strukturi jednaka nuli. Slika prikazuje izgled undo strukture za proces sa argumentom a. Posle prve operacije P, proces ima triplet za semafor 0 i vrednost jednaku 1. Posle druge operacije P, uvodi se novi triplet za semafor 1, sa vrednošću 1. Ako bi proces u tom trenutku iznenada završio, krenel bi dodao svakom semaforu vrednost 1 i tako im vratio vrednosti na početne. U regularnom slučaju, kernel dekrementira podešenu undo vrednost za semafor 1 za vreme treće operacije, zato što inkrementira semafor i taj triplet postaje 0 pa se ukljanja, a isto se dešava sa tripletom za semafor 0 posle četvrte operacije.

Ranije opisano polje operacija na semaforu, omogućava da se izbegava deadlock za 2 semafora, ali undo struktura za to polje je komlikovana. Problem zastoja (deadlock) može da se rešava na korisničkom nivou.

Sistemski poziv semctl

Sistemski poziv semctl sadrži brojne operacije za semafor:

```
semctl(id, number, cmd, arg)
```

Arg se deklariše kao unija:

```
union semunion
{
  int val;
  struct semid_ds *semstat;
  unsigned short *array;
} arg;
```

Kernel interpretira arg na bazi komande cmd, kao kod sistemskog poziva ioctl. Očekivane akcije su setovanje ili čitanje kontrolnih parametara, setovanje ili čitanje semaforskih vrednosti. Na primer, za komandu koja briše semafor, IPC_RMID, kernel nalazi sve procese koji imaju undo strukturu za semafore i ukljanjaju njihove triplete. Zatim se reinicijalizuje semaforska struktura podataka i bude se svi uspavani procesi koji čekaju na neki događaj na semaforu, koji kada nađu da semafor više ne postoji vraćaju informaciju o grešci.

Generalni zaključci

Postoje sličnosti između sistema datoteka i IPC mehanizama. Sistemski poziv get (IPC) sličan je sistemskom pozivu open za sistem datoteka, a sistemski pozivi tipa "control" koji ukljanjaju deskriptore iz sistema slični su sistemskom pozivu unlink. Jedino sistemski poziv close nema sličnu operaciju u IPC, jer kernel ne vodi zapis o svim procesima koji koriste IPC. Proces može korisiti IPC bez sistemskog poziva get, ako uspe da pogodi ID i ima dovoljno prava. Kernel ne može obrisati nekorišćene IPC strukture automatski, zato što ne zna da li su i dalje potrebne. Pogrešan proces može ostaviti nepotrebne IPC strukture. Kernel može čuvati IPC informacije u svojim strukturama, kada proces obavi sistemski poziv exit , ali je bolje da to čuvanje obavi u log datoteci.

IPC uvodi ključeve, umesto imena datoteka, ali teško je proširiti takav IPC na mrežu, zato što je potrebno opisati različite objekte na različitim mašinama, tako da ovaj IPC važi za jednu mašinu. Korišćenjem ključeva umesto imena datoteka ubrzava se IPC.

11.3. Mrežni IPC

Programi kao što su mail, ftp, remote-login, koji hoće da komuniciraju sa drugim mašinama imaju svoje ad-hoc metode za uspostavljanje konekcije i za razmenu podataka. Na primer, mail program čuva poštu u datoteci na /usr/mail/mjb za korisnika mjb. Kada korisnik šalje poštu preko iste mašine, mail program dodaje poruku na adresiranu datoteku koju drugi korisnik otvara i čita. Da bi se poslala pošta na drugu mašinu, mail program mora da pronađe odgovarajuću mail datoteku na drugoj mašini. Kako ne može direktno da pristupa drugoj mašini, nego samo preko mreže, postavljaju se dva agenta na obe mašine koji će im uspostaviti komunikaciju. Lokalni proces se zove klijent a udaljeni se naziva serverski proces.

Zato što UNIX kreira nove procese preko sistemskog poziva fork, serverski proces mora postojati pre nego što klijentski proces uspostavi konekciju. Bilo bi loše da udaljeni kernel kreira novi proces u trenutku kada se po mreži pojavljuje novi zahtev za konekcijom. Mnogo je bolje da serverski proces postoji i da stalno proverava prisustvo zahteva po mreži. To obično radi init proces koji kreira serverski proces koji čita komunikacioni kanal sve dok ne dobije neki zahtev za servis i iza toga se poštuje neki protokol. Klijentski i serverski programi obično biraju mrežni uređaj i protokole.

Na primer, uucp program omogućava ftp servis po mreži i rex proceduru (remote execution of comands). Klijentski proces daje upit u bazi svog servera za adresu i informacije o rutiranju (na primer br. Telefona). Potom otvara modemski uređaj (sistemski poziv open), pa obavlja sistemski poziv write ili ioctl na otvorenom uređaju i poziva udaljenu mašinu. Na udaljenoj mašini proces init će izvršiti proces getty na telefonskoj liniji kad se uspostavi hardverska veza između modema, pa se klijentski proces loguje i dobija specijalan shell pod nazivom uucico.

Mrežne komunikacije izazivaju problem za UNIX, zato što poruke često uključuju i delove sa podacima i kontrolne delove. Kontrolne informacije obično imaju adresu koja opisuje odredište poruke, a format adrese zavisi od protokola, tako da bi proces morao da zna sve detalje protokola, a to ugrožava UNIX princip o nezaviosti tipa datoteke tj. uređaja.

UNIX je stalno poboljšavao svoju mrežnu funkciju, a streams strukture su doneli elegantan metod za mrežnu podršku, zato što protokol moduli mogu fleksibilno da se dodaju u streams strukturu. Sada ćemo opisati BDS soket mehanizam.

Soketi (Sockets)

Procesi mogu komunicirati po mreži, ali način komunikacije zavisi od protokola i mrežnih uređaja. Da bi se obezbedila IPC preko raznih mrežnih protokola, BSD je obezbedio mehanizam koji se naziva soket. Ovde ćemo objasniti korisnički aspekt soket mehanizma, koji je prikazan na slici 11.5.

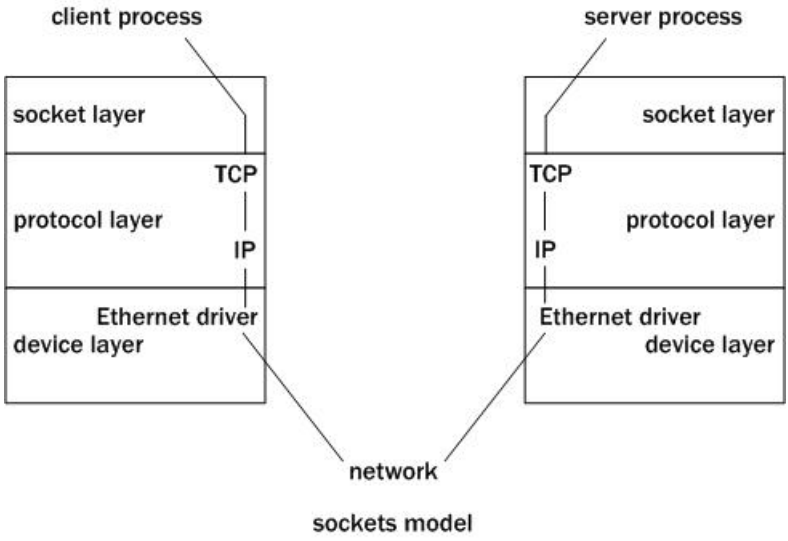
Kernelska soket struktura sastoji se od tri nivoa (layers): soket nivo, protokol nivo i nivo uređaja (device layer). Soket nivo obezbeđuje interfejs između sistemskih poziva i nižih slojeva, protokol nivo sadrži protokolske module korišćene u komunikaciji (TCP, IP), a nivo uređaja sadrži device drajvere za mrežne uređaje. Legalna kombinacija protokola i drajvera se specificira prilikom konfiguracije sistema. Procesi komuniciraju koristeći klijent-server model, server proces osluškuje svoj soket (listen), koji je jedan od krajeva dvo-komunikacione putanje. Klijentski proces komunicira sa serverskim procesom preko svog soketa koji je drugom kraju komunikacione putanje. Kernel održava interne konekcije i rutira podatke od klijenta do servera.

Soketi koji dele zajedničke komunikacione osobine (naming, protocols address format) se grupišu u domene. BSD uvodi dva domena: UNIX domen za procese koji komuniciraju na istoj mašini i Internet domen za procese koji komuniciraju na mreži koristeći DARPA komunikacione protokole. Svaki soket ima tip koji može biti

1. datagram

2. virtuelno kolo VC (virtual circuit).

VC dozvoljava sekvenciranje i pouzdan prenos podataka. Datagrami nemaju sekvenciranje, ni pouzdan prenos, ali su jednostavniji za realizaciju od VC, zato što nemaju dugotrajne setup operacije. Datagrami su podesni za neke vrste komunikacija. UNIX Sistem V ima podrazumevani (default) protokol, za svaki domen-socket tip. Na primer TCP protokol je za VC, dok je UDP datagram protokol u Internet domenu.



Slika 11.5. *Socket mehanizam*

Sistemski pozivi za soket mehanizam

Soket mehanizam sadrži više sistemskih poziva.

- Sistemski poziv socket uspostavlja krajnu tačku (end-point) komunikacionog linka.

  ```
  sd=socket(format, type, protocol);
  ```

  Format specificira komunikacioni domen (UNIX system domain or Internet domain), type ukazuje na tip komunikacije (VC or datagram) i protokol opisujuje partikularni protokol koji kontroliše komunikaciju. Procesi će koristiti soket deskriptor sd u drugim sistemskim pozivima.

- Sistemski poziv close zatvara soket.

- Sistemski poziv bind udružuje ime sa soket deskriptorom

  ```
  bind(sd, address, length);
  ```

  gde je sd soket deskriptor, address ukazuje na strukturu koja specificira identifikatore za komunikacioni domen i protokol definisan u soket sistemskom pozivu. Length je dužina adresne strukture, na primer to je ime datoteke u UNIX domenu. Server procesira bind adrese u soketima i postavlja njihova imena da bi identifikovao sebe kod klijentskih procesa.

- Sistemski poziv connect omogućava kernelu, tj. klijentskom procesu da obavi konekciju u postojeći soket:

  ```
  connect(sd, address, length);
  ```

  a semantika je ista kao za sistemski poziv bind, ali je adresa u ovom slučaju adresa target soketa, koji formira drugi kraj komunikacione linije. Oba soketa moraju imati isti komunikacioni domen i protokol, a kernel uređuje da se komunikacioni link postavi korektno. Ako je tip definisan kao datagram, sistemski poziv connect informiše kernel o adresi koja će se koristiti za naredne sistemskog poziva send, koji služi za slanje poruka preko soketa.

- Sistemski poziv listen. Kada serverski proces uređuje prijem konekcije preko VC, kernel mora ubaciti dolazeće zahteve u red čekanja (queue), pre nego što ih servisira. Sistemski poziv listen specificira maksimalnu veličinu reda čekanja (queue):

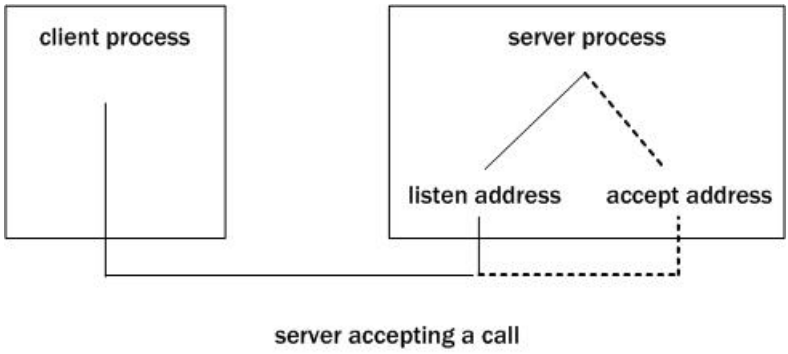
  ```
  listen(sd, qlength);
  ```

  gde je sd soket deskriptor, a qlength je maksimalni broj zahteva koji čekaju na obradu.

- Sistemski poziv accept prima dolazeće zahteve za konekciju sa serverskim procesom:

  ```
  nsfd=accept(sd, address, addrlen);
  ```

  gde je sd soket deskriptor, address ukazuje na korisničko polje, koje kernel puni sa povratnim adresama klijenata koji se povezuju, addrlen ukazuje na dužinu za polje address. Kada sistemski poziv accept završi aktivnost, kernel prepisuje sadržinu addrlen sa brojem, koji ukazuje na količinu prostora koju je zauzelo polje address. Sistemski poziv accept vraća novi soket deskriptor, različit od soket deskrtiptora sd. Server može nastaviti da osluškuje na soketu, dok komunicira sa klijentskim procesom, preko odvojenog komunikacionog kanala, kao na slici 11.6.



Slika 11.6. *Sistemski poziv accept*

- Sistemski poziv send. Sistemski pozivi send i recv prenose podatke preko konektovanog soketa:

  ```
  count = send(sd, msg, length, flags);
  ```

  gde je sd soket deskriptor, msg je pokazivač na poruku koja će biti poslata, length je dužina poruke, count je broj bajtova koji je zaista poslat. Flags parametar može biti setovan na vrednost SOF_OOB, a ukazuje da se podaci šalju na "out-of-band" način. A to znači da podaci koji se šalju, nisu deo regularane sekvence podataka koja se šalje između procesa. Na primer, za udaljeni login, karakter <delete> se šalje sa ovim flagom.

- Sistemski poziv recv. Sintaksa za recv sistemski poziv je

  ```
  count = recv(sd, buf, length, flags);
  ```

  gde je sd soket deskriptor, buf je data polje za dolazeće podatke, length je očekivana dužina poruke, count je broj bajtova koji je zaista primljen. Flags mogu biti setovani da pokupe dolazeću poruku i ispitaju joj sadžaj bez uklanjanja iz queue, ili na bazi "out of data". Datagram opcije za sistemski poziv recv, su sendto i recvfrom, a imaju dodatne parametre za adresiranje. Procesi mogu da koriste sistemske pozive read i write na otvorenim soketima umesto send i recv, kao da su soketi obične datotke. (naravno posle uspostave konekcije)

- Sistemski poziv shutdown zatvara soket konekciju:

  ```
  shutdown(sd, mode);
  ```

  gde mode ukazuje da li se radi o predaji (sending end), prijemu (receiving end) ili se na oba kraja više ne dozvoljava prenos podataka. Ovaj sistemski poziv infomiše protokole da zatvore mrežnu komunikaciju, pri čemu sd i dalje postoji i jedino će sistemski poziv close osloboditi soket deskriptor

- Sistemski poziv getsockname uzima ime soketa koje je vezano preko prethodnog sistemskog poziva bind.

```
getsockname(sd, name, length);
```

- Sistemski pozivi getsockopt i setsockopt vraćaju ili postavljaju različite opcije za soket, u saglasnosti sa domenom i protokolom.

Primeri za socket

Analizirajmo sledeći program koji predstavlja serverski proces. Proces kreira stream soket u UNIX sistem domenu i vezuje na taj soket ime "sockname", a to će biti datoteka. Zatim se poziva sistemski poziv listen da specificira interni red čekanja (queue) za dolazeće poruke i ulazi u petlju, čekajući dolazeće zahteve. Sistemski poziv accept spava sve dok protokol ne objavi da je zahtev za konekciju upućen ka tom soketu. Zatim, sistemski poziv accept vraća novi deskriptor za dolazeće zahteve. Serverski proces obavlja sistemski poziv fork, koji kreira proces da komunicira sa klijentskim procesom: roditelj i dete zatvaraju njihove deskriptore tako da ne učestvuju u komunikacionom saobraćaju drugog procesa. Proces dete obavlja svoju konverzaciju sa klijentskim procesom, a završava aktivnosti posle povratka iz sistemskog poziva read. Server proces ostaje u petlji i čeka na novi zahtev za konekciju u sistemski poziv accept.

Serverski proces

```
server proces in the UNIX System Domain

#include <sys/types.h>
#include <sys/socket.h>
main()
{
  int sd,ns;
  char buf[256];
  struct sockaddr sockaddr;
  int formlen;
  sd=socket(AF_UNIX, SOCK_STREAM, 0);
  /*bind name- null char is not part of name*/
  bind(sd, "sockname", sizeof("sockname"-1);
  listen(sd,1);
  for(;;)
  {
    ns=accept(sd, &sockaddr, &fromlen);
    if(fork() = = 0)
    {
      /*child*/
      close(sd);
      read(ns, buf, sizeof(buf);
```

```
      printf("server read %s", buf);
      exit();
    }
    close(ns);
  }
}
```

Klijentski proces

Analizirajmo klijentski proces koji sarađuje sa serverskim procesom. Klijent kreira soket u istom domenu kao serverski proces i realizuje sistemski poziv connect za datoteku sockname, koja je povezana za soket u serverskom procesu. Kada se sistemski poziv connect vrati, klijent proces ima VC ka serverskom procesu. U ovom primeru, on upisuje jednu poruku i izlazi tj. obavlja sistemski poziv exit.

```
client proces in the UNIX System Domain
#include <sys/types.h>
#include <sys/socket.h>
main()
{
  int sd,ns;
  char buf[256];
  struct sockaddr sockaddr;
  int formlen;
  sd=socket(AF_UNIX, SOCK_STREAM, 0);
  /*connect to name- null char is not part of name*/
  if(connect(sd, "sockname", sizeof("sockname"-1) = = -1) exit();
  write(sd, "hi guy",6)
}
```

Internet domen

Ako server proces želi da opslužuje procese na mreži, mora da specificira Internet domen u sistemskom pozivu socket, kao

```
socket(AF_INET, SOCK_STREAM, 0);
```

i obavi vezivanje (bind) mrežne adrese dobijene od servera za imenovanje (name server). BSD ima biblioteku za ove funkcije. Takođe, drugi parametar u klijentovom sistemskom pozivu connect treba da sadrži adresnu informaciju, potrebnu za identifikaciju mašine na mreži (adresu rutiranja za slanje poruka na odredišnu mašinu preko roter mašine) i dodatne informacije koje identifikuju partikularni soket na odredišnoj (destination) mašini. Ako server želi da osluškuje (listen) na mreži i lokalne procese, on treba da koristi dve vrste soketa, a sistemski poziv select određuje koji klijent realizuje konekciju.

Literatura

[1] D. E. Knuth, "*The Art of Computer Programming, Volume 1: Fundamental Algorithms*", Second Edition, Addison-Wesley, 1973.

[2] M. J. Bach, "*The Design of the UNIX Operating System*", Prentice Hall, 1987.

[3] M. Mitchell, J. Oldham, A. Samuel " Advanced Linux Programming", New Riders Publishing, ISBN 0-7357-1043-0, June 2001

[4] A. M. Lister, R. D. Eager, "*Fundamentals of Operating Systems*", Fifth Edition, The Macmillan Press Ltd, 1993.

[5] S. Rago, "*UNIX System V Network Programming*", Addison-Wesley, 1993.

[6] A. S. Tannenbaum, A. S. Woodhull, "*Operating System Design and Implementation*", Second Edition, Prentice Hall, 1997

[7] J. Gray, "*Interprocess Communication in UNIX*", Prentice Hall, 1997.

[8] N. Rhodes, J. McKeehan, "*Understanding the Linux Kernel*", O'Reily and Associates, 1999.

[9] W. Stallings, "*Operating Systems*", Fourth Edition, Prentice Hall, 2000.

[10] M. Bar, "*Linux Internals*", McGraw-Hill, 2000.

[11] A. Danesh, "*Red Hat Linux*", Mikro knjiga, 2000.

[12] A. S. Tannenbaum, "*Modern Operating Systems*", Prentice Hall, 2001.

[13] W. Stanfield, R. D. Smith, "*Linux System Administration*", Second Edition, Sybex 2001.

[14] R. W. Smith, "*Linux+ Study Guide*", Sybex, 2001.

[15] D. P. Bovet, M. Cesati, "*Understanding the Linux Kernel*", O'Reily and Associates, 2001.

[16] J. Mauro, R. McDougall, "*Solaris Internals: Core Kernel Architecture*", Prentice Hall, 2001.

[17] G. Mourani, "*Securing and Optimizing Linux: The Ultimate Solution*", Open Network Architecture Inc, 2001.

[18] T. Collings, K. Wall, "*Red Hat Linux Networking & System Administration*", Hungry Minds Inc., 2002.

[19] A. Silberschatz, P. B. Galvin, G. Gagne, "*Operating System Concepts*", Sixth Edition (Windows XP Update), John Wiley & Sons, Inc, 2003.

[20] S. Figgins, E. Siever, A. Weber, "*Linux in a Nutshell*", 4th Edition, O'Reilly & Associates, Inc., 2003.

[21] B. Đorđević, D. Pleskonjić, N. Maček, "*Operativni sistemi: UNIX i Linux*", Viša elektrotehnička škola, Beograd, 2004.

[22] B. Đorđević, D. Pleskonjić, N. Maček, "*Operativni sistemi: koncepti*", Viša elektrotehnička škola, Beograd, 2004.

[23] B. Đorđević, D. Pleskonjić, N. Maček, "*Operativni sistemi: zbirka rešenih zadataka*", Viša elektrotehnička škola, Beograd, 2005.

[24] B. Đorđević, D. Pleskonjić, N. Maček, "*Operativni sistemi: teorija, praksa i rešeni zadaci*", Mikro knjiga, 2005.