Borislav Đorđević
Marko Carić
Dragan Pleskonjić
Nemanja Maček

OPERATIVNI SISTEMI 1
priručnik za laboratorijske vežbe

GNU/Linux sistemsko programiranje

Autori:	dr Borislav Đorđević
	Marko Carić
	mr Dragan Pleskonjić
	Nemanja Maček
Recenzenti:	dr Verica Vasiljević
	dr Slobodan Obradović
Izdavač:	Visoka škola elektrotehnike i računarstva
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

CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд

004.451.9LINUX(075.8)(076)
004.42:004.451(075.8)(076)

OPERATIVNI sistemi 1 : priručnik za laboratorijske vežbe. GNU/Linux sistemsko programiranje / Borislav Đorđević ... [et al.]. - Beograd : Visoka škola elektrotehnike i računarstva, 2007 (Beograd : MST Gajić). - VI, 202 str. : ilustr. ; 24 cm

Tiraž 200. - Bibliografija: str. 201-202.

ISBN 978-86-7982-009-9
1. Ђорђевић, Борислав [аутор]
a) Оперативни систем "Linux" - Програмирање - Вежбе
COBISS.SR-ID 145260812

Predgovor

Teorija i praksa operativnih sistema predstavlja jednu od fundamentalnih oblasti računarske tehnike koja u kombinaciji sa arhitekturom računara daje kvalitetnu osnovu za izučavanje svih oblasti računarske tehnike.

Jedan od najpopularnijih operativnih sistema je familija UNIX/Linux operativnih sistema, koja funkcioniše na velikom broju računara sa različitom procesorskom snagom, od mikroprocesora do mainframe mašina. UNIX je operativni sistem koga već 30 godina odlikuju visoke performanse i izražena stabilnost i koji je kao takav pogodan za izvršavanje različitih serverskih i klijentskih aplikacija. Većina velikih svetskih proizvođača ozbiljne računare opreme 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. 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, ”GNU/Linux sistemsko programiranje“ je prvenstveno namenjena kao priručnik za laboratorijske vežbe iz predmeta Operativni Sistemi 1 na Višoj elektrotehničkoj školi u Beogradu. Knjiga sadrži veliki broj praktičnih primera u C jeziku, na GNU/Linux sistemu, koji se odnose na procese, niti, rad sa datotekama i IPC mehanizme. Primeri pomažu studentima da bolje razumeju fundamentalne koncepte operativnih sistema. Knjiga može poslužiti kao koristan izvor informacija, svakom čitaocu koji se ozbiljnije bavi teorijom i praksom operativnih sistema.

Zahvalnost

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

Autori

Sadržaj

1. Uvod u sistemsko programiranje

U ovom poglavlju videćete kako da izvedete osnovne korake neophodne za kreiranje C ili C++ Linux programa. Naročito će u ovom poglavlju biti reči o tome kako da napišete i modifikujete C i C++ izvorni kôd, da prevedete taj kôd i da debagujete rezultujući program. U ovoj knjizi pretpostavljamo da poznajete C ili C++ programski jezik, kao i najosnovnije funkcije iz standardne C biblioteke. Primeri izvornog koda u ovom pratktikumu napisani su na programskom jeziku C, osim kada demonstriramo naročitu osobinu ili složenost C++ programiranja. Takođe pretpostavljamo da znate da izvedete osnovne operacije u Linux komandnoj liniji, kao što su kreiranje direktorijuma i kopiranje datoteka. Pošto su mnogi Linux programeri počinjali sa programiranjem u Windows okruženju, povremeno ćemo ukazivati na sličnosti i razlike između Windows i Linux operativnih sistema.

Otvaranje i izmena datoteke C ili C++ izvornog koda

Editor je program koji koristite za uređivanje teksta izvornog koda. Mnoštvo različitih editora je raspoloživo na Linux sistemu – vi, vim (vi improved), emacs, joe, jed, pico, nano i drugi. Ukratko ćemo opisati upotrebu editora vi, emacs, joe i jed.

Editor teksta vi

Vi editor je deo svakog UNIX sistema, počev od prvih verzija, koje su se pojavile ranih sedamdesetih godina. Iako je vi ekranski editor, jako je neugodan za korišćenje. U mnogim Linux distribucijama, vi editor je zamenjen editorom vim (vi improved). Koristi se isključivo za izmenu sadržaja tekstualnih datoteka i ne podržava formatiranje (na primer, masna slova i kurziv). Vi editor koristi ekranski prikaz datoteke (full-screen), ali ne dozvoljava upotrebu miša za pozicioniranje kursora na odgovarajući karakter – navigacija i sve modifikacije vrše se isključivo pomoću tastature. Izmene sadržaja datoteka se mogu snimiti na disk ili odbaciti.

Vi editor je deo svakog UNIX sistema i njegovo poznavanje je značajno za svakog ozbiljnog sistem administratora. Komande za modifikaciju datoteka kao što su `/etc/passwd` (`vipw`) i `/etc/group` (`vigr`) koriste vi za izmenu sadržaja ovih datoteka). Međutim, korisnik koji planira da postane sistem administrator treba da se upozna i sa drugim editorima teksta kao što su joe, jed, emacs i pico, koji su lakši i prijatniji za korišćenje.

Postoje tri osnovna režima rada vi editora: komandni, tekstualni i režim poslednje linije (linijski).

Komandni mod je režim u kome se korisnik nalazi kada pokrene vi. Korisnik može da unese komande za pozicioniranje kursora i komande za modifikaciju sadržaja datoteke. Iz ovog režima korisnik može da pređe u tekstualni ili linijski režim. Da bi prešao iz tekstualnog u linijski režim korisnik prvo mora preći u komandni režim. Taster Esc je uvek prečica za prelazak u komandni režim, tako da korisnik uvek može da ga iskoristi ukoliko nije siguran u kom se režimu nalazi. Dok se nalazi u komandnom režimu, korisnik može da vrši navigaciju po tekstu i zadaje određene komande.

Da bi unosio tekst korisnik mora preći u tekstualni režim komandama I (*input*), o (*open new line*) ili a (*apend*). Naravno, u ovom režimu uneti tekst neće biti protumačen kao komanda. Nakon završenog unosa, korisnik se može vratiti u komandni režim tasterom Esc.

U režim zadnje linije korisnik može preći zadavanjem karaktera dvotačka (:) u komandnom režimu. Tada se kursor pomera na dno ekrana. U ovom režimu korisnik može da snimi datoteku, snimi datoteku i napusti vi, napusti vi bez snimanja datoteke, traži i zameni određene nizove karaktera i konfiguriše vi.

Komande koje se koriste za prelazak iz jednog režima u drugi su:

- Komandni -> tekstualni: i (*input*), o (*open new lin*e), a (*append*)
- Tekstualni -> komandni: Esc
- Komandni -> režim zadnje linije: dvotačka (:)
- Režim zadnje linije -> komandni režim: Enter

Vi editor je je UNIX aplikacija koja se pokreće iz komandne linije. Ukoliko korisnik zada komandu vi bez argumenata, ili kao argument navede ime nepostojeće datoteke, vi će početi rad na novoj datoteci. Ukoliko se kao argument navede ime postojeće datoteke, vi će datoteku učitati, nakon čega korisnik može da vrši izmene. Sadržaj datoteke na disku se ažurira sadržajem modifikovanim vi editorom tek kada korisnik zada komandu za snimanje datoteke.

Sintaksa komande vi je:

```
$ vi [opcije] ime_datoteke
```

Najšešća sekvenca koraka u radu sa vi editorom je: otvaranje nove ili postojeće datoteke, unošenje novog ili modifikacija postojećeg teksta, snimanje izmena na disk i napuštanje vi editora.

Editor vi – interaktivne komande

Komande za prelazak iz komandnog u tekstualni režim rada:

- a unesi tekst posle kursora
- A unesi tekst na kraju linije
- i unesi tekst pre kursora
- o otvori novu liniju pre kursora

Komande za snimanje datoteka i napuštanje vi editora dostupne su iz režima poslednje linije:

- :w snimanje datoteke na disk pod tekućim imenom
- :w filename snimanje datoteke na disk pod imenom filename
- :wq snimanje datoteke i napuštanje vi editora
- :q! napuštanje vi editora bez snimanja izmena

Komande za navigaciju u komandnom (ne u tekstualnom) režimu (kursorski tasteri - strelice se mogu koristiti za navigaciju ako konkretni terminal to podržava):

- h pomera kursor na prethodni karakter (left arrow)
- l pomera kursor na sledeći karakter (right arrow, space)
- w pomera kursor na sledeću reč
- b pomera kursor na prethodnu reč
- k pomera kursor na prethodnu liniju (up arrow)
- j pomera kursor na sledeću liniju (down arrow)
- $ pomera kursor na kraj linije
- 0 pomera kursor na pocetak linije
- Return pomera kursor na početak sledeće linije

Komande za modifikaciju teksta dostupne su iz komandnog režima i režima zadnje linije:

- x briše karakter koji je obeležen kusorom
- dw brise reč desno od kursora
- dd briše liniju u kojoj se nalazi kursor
- yy kopira liniju u kojoj se nalazi kursor na clipboard (yank)
- p kopira liniju sa clipboarda ispod tekuće linije (paste)
- P kopira liniju sa clipboarda iznad tekuće linije (paste)
- u poništava prethodnu akciju (undo)

Emacs

Jedan od najfunkcionalnijih editora teksta je GNU Emacs. Emacs možete pokrenuti unošenjem teksta `emacs` u prozoru terminala i pritiskom na taster Enter. Kada se Emacs pokrene, možete koristiti

padajuće menije na vrhu da biste kreirali novu datoteku izvornog koda. Kliknite na Files meni, izaberite Open Files i unesite tekst imena datoteke koju želite da otvorite u "minibaferu" na dnu ekrana. Ako želite da kreirate datoteku sa C izvornim kodom, ime datoteke završite sa `.c` ili `.h`. Ako želite da kreirate datoteku sa C++ izvornim kodom, ime datoteke završite sa `.cpp`, `.hpp`, `.cxx`, `.hxx`, `.C` ili `.H`. Kada je datoteka otvorena, tekst možete unositi kao u bilo kom programu za obradu teksta. Da biste sačuvali datoteku izaberite Save Buffer sa padajućeg menija File. Kada završite sa radom u Emacsu, sa istog padajućeg menija File možete izabrati opciju Exit Emacs.

Ako više volite, možete da koristitite i prečice sa tastature da automatski otvarate i snimate datoteke, kao i da zatvorite Emacs. Za otvaranje datoteke pritisnite <Ctrl-x> <Ctrl-f>. (<Ctrl-x> znači da držite pritisnutim Control taster, a da zatim pritisnete taster x.) Da biste sačuvali datoteku, pritisnite <Ctrl-x><Ctrl-s>. Za zatvaranje Emacsa samo pritisnite <Ctrl-x><Ctrl-c>. Ako želite da se malo bolje upoznate sa Emacsom, izaberite Emacs Tutorial sa Help padajućeg menija. U ovom uputstvu ćete naći mnoštvo saveta kako da efikasno koristite Emacs.

Emacs – automatsko formatiranje

Ako ste navikli da programirate u integrisanom razvojnom okruženju (Integrated Development Environment – IDE), verovatno ste navikli da vam editor pomaže u formatiranju vašeg koda. Emacs vam omogućava to isto.

Ako otvorite C ili C++ datoteku izvornog koda, Emacs automatski zaključuje da ta datoteka sadrži izvorni kôd, a ne samo običan tekst. Ako pritisnete Tab taster u praznoj liniji, Emacs će pomeriti kurzor na odgovarajuću udaljenost od margine. Ako pritsnete Tab taster u liniji koja sadrži neki tekst, Emacs će uvući taj tekst. Na primer, pretpostavimo da ste uneli sledeći tekst:

```
int main ()
{
printf ("Hello, world!\n");
}
```

Ako pritisnete Tab taster u liniji koja sadrži poziv funkcije `printf`, Emacs će formatirati vaš kôd na sledeći način:

```
int main ()
{
  printf ("Hello, world!\n");
}
```

Primetite kako je linija pravilno uvučena.

Emacs – isticanje sintakse

Pored formatiranja, Emacs vam može olakšati i čitanje C i C++ koda bojenjem različitih elemenata sintakse. Na primer, Emacs će ključne reči prikazati u jednoj boji, ugrađene tipove kao što je int u drugoj, a komentare u trećoj boji. Upotrebom boja se u mnogome olakšava uočavanje nekih uobičajenih sintaksnih grešaka.

Najjednostavniji način da uključite isticanje sintakse je da u datoteku `~/.emacs` unesete sledeću liniju teksta:

```
(global-front-lock-mode t)
```

Snimite datoteku, ugasite Emacs i ponovo ga pokrenite. Nakon toga otvorite C ili C++ datoteku izvornog koda i uživajte!

Možda ste primetili da tekst koji ste uneli u vašu `.emacs` datoteku liči na kôd LISP programskog jezika. To je zato što to jeste LISP kôd! Većim delom Emacs je u stvari napisan na LISP programskom jeziku. Možete dodati nove funkcije Emacs-u pišući LISP kôd.

Editor teksta joe

Za razliku od vi editora, joe je krajnje prijatan za rad. Editor joe se pokreće sledećom komandom:

```
$ joe ime_datoteke
```

Ukoliko korisnik zada komandu joe bez argumenata, ili kao argument navede ime nepostojeće datoteke, joe će početi rad na novoj datoteci. Ukoliko se kao argument navede ime postojeće datoteke, joe će datoteku učitati, nakon čega korisnik može da vrši izmene. Sadržaj datoteke na disku ažurira se sadržajem modifikovanim editorom joe tek kada korisnik zada komandu za snimanje datoteke.

Ovaj editor ne koristi režime: tekst se jednostavno unosi (korisnik se nalazi u insert režimu), za navigaciju po tekstu koriste se kursorski tasteri (kao i PgUp i PgDn), a komande se zadaju korišćenjem kombinacije tastera <Ctrl-slovo>.

U svakom trenutku korisnik može dobiti pomoć kombinacijom tastera <Ctrl-K>, a zatim pritiskom na taster H.

Za pretraživanje teksta koriste se komande <Ctrl-K>F (nakon čega se zadaje tekst koji se traži) i <Ctrl-L> koja traži sledeće pojavljivanje teksta. Nakon zadavanja ove komande korisnik može zameniti pronađeni tekst novim.

Za rad sa blokovima (blok je obeležen ako se markiraju i početak i kraj) koriste se sledeće komande:

- <Ctrl-K> B markiranje početka bloka
- <Ctrl-K> K markiranje kraja bloka
- <Ctrl-K> M pomeranje bloka na tekuću poziciju
- <Ctrl-K> C kopiranje bloka na tekuću poziciju
- <Ctrl-K> F snimanje bloka u datoteku čije se ime navodi
- <Ctrl-K> Y brisanje bloka

Za brisanje karaktera, reči i linija mogu se koristiti sledeće komande, koje redom brišu:

- <Ctrl-D> tekući karakter
- <Ctrl-Y> tekuću liniju
- <Ctrl-W> niz karaktera od tekućeg karaktera do kraja reči
- <Ctrl-O> niz karaktera od početka reči do tekućeg karaktera
- <Ctrl-J> niz karaktera od tekućeg karaktera do kraja linije
- <Ctrl--> poništava izvršenje prethodne operacije

Za snimanje datoteke i napuštanje editora joe koriste se sledeće komande:

- <Ctrl-K> X snimanje datoteke na disk i napuštanje editora joe
- <Ctrl-C> napuštanje editora bez snimanja datoteke

Editor teksta jed

Editor jed je mode-less editor, prijatan za rad kao i joe. Kao novina u jed editoru se uvode i padajući meniji koji se mogu koristiti za zadavanje komandi. Editor Jed se pokreće kao i većina ostalih editora:

```
$ jed ime_datoteke
```

Isto što važi za vi i joe važi i u ovom slučaju: ukoliko korisnik zada komandu jed bez argumenata, ili kao argument navede ime nepostojeće datoteke, jed će početi rad na novoj datoteci. U suprotnom, jed će datoteku učitati, nakon čega korisnik može da vrši izmene. Sadržaj datoteke na disku ažurira se sadržajem modifikovanim editorom jed tek kada korisnik zada komandu za snimanje datoteke.

Korisnik u jed editoru jednostavno unosti tekst i putem padajućih menija koristi komande za rad sa tekstom. Padajući meni sa opcijama dobija se pritiskom na taster <F10> ili kombinacijom tastera <Alt-slovo> (na primer, <Alt-F> će otvoriti padajući meni File, a <Alt-E> padajući meni Edit. Opcije padajućih menija su intuitivne.

Prevođenje i povezivanje

Prevodilac prevodi izvorni kôd razumljiv čoveku u objektni kôd razumljiv mašini koji može da se izvršava. Prevodioci koji se koriste na Linux sistemima deo su GNU kolekcije prevodioca (GNU Compiler Collection), poznatije kao GCC. GCC sadrži i prevodioce za C, C++, Javu, Objektni C, Fortran i Chill. Ova knjiga se odnosi na C i C++ programiranje.

Pretpostavimo da imate projekat kao što je ovaj prikazan u Listingu 1.2 sa jednom C++ datotekom izvornog koda (`reciprocnavrednost.cpp`) i jednom C datotekom izvornog koda (`main.c`) kao u Listingu 1.1. Ove dve datoteke bi trebalo da se prevedu, a zatim povežu i tako naprave program `reciprocnavrednost`. Ovaj program računa recipročnu vrednost celog broja.

```
#include <stdio.h>
#include "reciprocnavrednost.hpp"

int main (int argc, char **argv)
{
  int i;
  i = atoi (argv[1]);
  printf ("Reciprocna vrednost od %d je %g\n", i,
           reciprocnavrednost (i));
  return 0;
}
```

Listing 1.1 (main.c)

```
#include <cassert>
#include "reciprocnavrednost.hpp"

double reciprocnavrednost (int i) {
  // argument i bi trebalo da bude ne-nula.
  assert (i != 0);
  return 1.0/i;
}
```

Listing 1.2 (reciprocnavrednost.cpp)

Postoji i jedna datoteka zaglavlja koja se zove `reciprocnavrednost.hpp` (videti Listing 1.3).

```
#ifdef __cplusplus
extern "C" {
#endif

extern double reciprocnavrednost (int i);

#ifdef __cplusplus
}
#endif
```

Listing 1.3 (reciprocnavrednost.hpp)

Prvi korak je da se C i C++ izvorni kôd prevede u objektni kôd.

Prevođenje pojedinačne datoteke izvornog koda

Naziv C prevodioca je gcc. Da bismo preveli C datoteku izvornog koda, koristimo –c opciju. Tako se, na primer, zadavanjem sledeće naredbe u komandnoj liniji prevodi `main.c` datoteka izvornog koda:

```
$ gcc –c main.c
```

Rezultujuća objektna datoteka se zove main.o.

C++ prevodilac se zove g++ i koristi se slično kao i gcc. Prevođenje `reciprocnavrednost.cpp` datoteke ostvaruje se unošenjem sledeće naredbe:

```
$ g++ –c reciprocnavrednost.cpp
```

Opcija `–c` govori g++ prevodiocu da prevede program samo u objektnu datoteku. Bez te opcije g++ bi pokušao da poveže program i napravi izvršnu datoteku. Pošto ste uneli ovu naredbu, dobićete objektnu datoteku `reciprocnavrednost.o`.

Da biste napravili iole veći program, verovatno će vam biti potrebne još neke opcije. Opcija –I se koristi da ukažemo GCC prevodiocu gde da traži datoteke sa zaglavljima. Podrazumevano, GCC ih traži u tekućem direktorijumu i u direktorijumima gde su instalirane datoteke sa zaglavljima za standardne biblioteke. Ako trebate da dodate datoteke sa zaglavljima koje se nalaze u nekom drugom direktorijumu, koristićete -I opciju. Na primer, pretpostavimo da u vašem projektu postoji direktorijum src za datoteke izvornog koda i jedan koji se zove ινχλυδε. Da biste ukazali g++ prevodiocu da koristi i `../include` direktorijum za pronalaženje `reciprocnavrednost.hpp` datoteke, `reciprocnavrednost.cpp` ćete prevoditi na sledeći način:

```
$ g++ –c –I ../include reciprocnavrednost.cpp
```

Ponekad ćete želeti da definišete makroe u komandnoj liniji. Na primer, u proizvodnom kodu ne želite da imate proveru unosa u `reciprocnavrednost.cpp` datoteci. Ona je tu samo da vam pomogne pri debagovanju. Proveru isključujete definisanjem makroa `NDEBUG`. Možete izričito da dodate #define u `reciprocnavrednost.cpp` datoteku, ali to bi zahtevalo izmenu koda. Lakše je da jednostavno definišete `NDEBUG` u komandnoj liniji na sledeći način:

```
$ g++ –c –D NDEBUG reciprocnavrednost.cpp
```

Ako ste hteli da definišete `NDEBUG` na neku određenu vrednost, uradili biste nešto slično ovom:

```
$ g++ –c –D NDEBUG=3 reciprocnavrednost.cpp
```

Ako stvarno radite na proizvodnom kodu, verovatno ćete želeti da GCC optimizuje taj kôd tako da se program izvršava što je brže moguće. To možete postići dodavanjem -02 opcije u komandnoj liniji. (GCC pruža više različitih nivoa optimizacije. Drugi nivo je primeren većini programa.) Sledeća naredba prevodi `reciprocnavrednost.cpp` sa uključenom optimizacijom:

```
$ g++ –c –02 reciprocnavrednost.cpp
```

Trebalo bi imati na umu da prevođenje sa uključenom optimizacijom može da oteža debagovanje vašeg programa. (videti odeljak "Debagovanje pomoću GDB-a") Takođe, u određenim slučajevima, prevođenje sa uključenom optimizacijom može da otkrije greške u vašem programu koje se ranije nisu javljale.

Možete proslediti još puno drugih opcija gcc i g++ prevodiocima. Najbolji način za dobijanje kompletne liste opcija je da pogledate postojeću dokumentaciju. To ćete uraditi unošenjem sledeće naredbe u komandnoj liniji:

```
$ info gcc
```

Povezivanje objektnih datoteka

Sada, kada ste preveli `main.c` i `reciprocnavrednost.cpp`, poželećete da ih povežete. Uvek koristite g++ prevodilac kada povezujete program koji sadrži C++ kôd , čak i onda kada on sadrži i C kod. Ako vaš program sadrži isključivo C kôd koristite gcc prevodilac. Kako ovaj program sadrži i C i C++ kôd , koristićete g++ prevodilac i to na sledeći način:

```
$ g++ –o reciprocnavrednost main.o reciprocnavrednost.o
```

Opcijom –o se definiše naziv datoteke koja će biti rezultat povezivanja. Sada možete pokrenuti reciprocnavrednost na sledeći način:

```
$ ./reciprocnavrednost 7
```

Program na ekranu ispisuje sledeći rezulta:

```
The reciprocnavrednost of 7 is 0.142857
```

Kao što vidite, g++ je automatski uključio standardnu C biblioteku koja sadrži implementaciju funkcije printf. Da je trebalo da uključite neku drugu biblioteku (kao što je skup alata za izradu grafičkog korisničkog interfejsa), naveli bi je pomoću opcije -l. Na Linux sistemu nazivi biblioteka skoro uvek počinju sa lib. Na primer, Pluggable Authentication Module (PAM) biblioteka se zove libpam.a. Da biste uključili libpam.a, koristićete sledeću naredbu:

```
$ g++ –o reciprocnavrednost main.o reciprocnavrednost.o -lpam
```

Prevodilac automatski dodaje lib na početak i .a na kraj.

Kao i sa datotekama zaglavlja, povezivač traži biblioteke na nekim podrazumevanim mestima, uključujući /lib i /usr/lib direktorijume u kojima se nalaze standardne sistemske biblioteke. Ako hoćete da povezivač pretraži i druge direktorijume, koristite -L opciju, koja se koristi isto kao i –I opcija koju smo ranije spomenuli. Na sledeći način možete uputiti povezivač da potraži biblioteke u /usr/local/lib/pam direktorijumu pre nego što ih potraži na podrazumevanim mestima:

```
$ g++ –o reciprocnavrednost main.o reciprocnavrednost.o –
L/usr/local/lib/pam -lpam
```

Iako ne morate da koristite –I opciju da ukažete predprocesoru da pretraži tekući direktorijum, to morate učiniti -L opcijom u slučaju povezivača. Naime, možete koristiti sledeću naredbu da ukažete povezivaču da potraži test biblioteku u tekućem direktorijumu:

```
$ gcc –o app app.o –L. –ltest
```

Automatizovanje procesa pomoću GNU Make-a

Ako ste programirali na Windows operativnom sistemu, verovatno ste to radili u nekom od integrisanih razvojnih okruženja (Integrated Development Environment – IDE). Dodali biste datoteke sa izvornim kodom u projekat i okruženje bi automatizovano izvršilo proces prevođenja celog vašeg projekta. Iako postoje integrisana razvojna okruženja i za Linux, ovde ih nećemo razmatrati. Umesto toga,

pokazaćemo vam kako da koristite GNU Make za automatizaciju prevođenja vašeg koda, a to je ono što i većina Linux programera radi.

Osnovna ideja koja stoji iza make-a je jednostavna. Kažete make-u koje ciljne objekte hoćete da obradi i date mu pravila po kojima da ih obradi. Takođe, definišete zavisnosti po kojima bi pojedini objekat eventualno trebalo ponovo obraditi.

Očigledno je da u našem primer projektu `reciprocnavrednost` postoje tri ciljna objekta: `reciprocnavrednost.o`, `main.o`, kao i sam `reciprocnavrednost`. U prethodnim primerima ste se već upoznali sa pravilima po kojima će se izvršiti obrada. Zavisnosti zahtevaju malo više pažnje. Jasno je da `reciprocnavrednost` zavisi od `reciprocnavrednost.o` i `main.o`, zato što ne možete da povežete ceo program dok ne prevedete svaku od objektnih datoteka. Objektne datoteke bi trebalo prevesti svaki put kada se izmene odgovarajuće datoteke izvornog koda. Takođe bi i promena `reciprocnavrednost.hpp` datoteke trebalo da izazove ponovno prevođenje obe objektne datoteke, zato što obe datoteke izvornog koda uključuju tu datoteku u svom zaglavlju.

Pored očiglednih ciljnih objekata, uvek bi trebalo da postoji i "očišćeni" (engl. clean) objekat. Ovaj ciljni objekat briše sve objektne datoteke i programe koji su generisani, tako da vam omogućava "čist" početak. Pravilo za ovaj objekat koristi rm naredbu za brisanje datoteka.

Sve ove informacije možete dostaviti make-u tako što ćete ih staviti u datoteku pod nazivom `Makefile`. Evo šta `Makefile` sadrži:

```
reciprocnavrednost: main.o reciprocnavrednost.o
 g++ $(CFLAGS) -o reciprocnavrednost main.o reciprocnavrednost.o
main.o: main.c reciprocnavrednost.hpp
 gcc $(CFLAGS) -c main.c
reciprocnavrednost.o: reciprocnavrednost.cpp
reciprocnavrednost.hpp
 g++ $(CFLAGS) -c reciprocnavrednost.cpp
clean:
 rm -f *.o reciprocnavrednost
```

Vidite da su ciljni objekti navedeni sa leve strane. Sledi dvotačka, a zatim zavisnosti, ako postoje. Pravilo za obradu je navedeno u sledećoj liniji. (Za sada ne obraćajte pažnju na `$(CFLAGS)`.) Linija sa pravilom mora biti uvučena jednim Tab karakterom, ili će se make zbuniti. Ako uređujete vašu `Makefile` datoteku u Emacs-u, on će vam pomoći u formatiranju.

Ako obrišete objektne datoteke koje ste već napravili i unesete

```
$ make
```

u komandnoj liniji, videćete sledeće:

```
gcc –c main.c
g++ –c reciprocnavrednost.cpp
g++ –o reciprocnavrednost main.o reciprocnavrednost.o
```

Vidite da je make automatski napravio objektne datoteke i zatim ih povezao. Ako sada promenite `main.c` i ponovo unesete make, videćete sledeće:

```
gcc –c main.c
g++ –o reciprocnavrednost main.o reciprocnavrednost.o
```

Vidite da je make znao da ponovo prevede `main.o` i da ponovo poveže program, ali da nije ponovo prevodio `reciprocnavrednost.cpp`, zato što se ni jedna od zavisnosti za `reciprocnavrednost.o` nije promenila.

`$(CFLAGS)` je make promenljiva. Možete je definisati u okviru `Makefile`-a ili u komandnoj liniji. GNU make će zameniti vrednost promenljive kada bude izvršavao pravilo. Tako biste, na primer, za ponovno prevođenje sa uključenom optimizacijom, uradili sledeće:

```
$ make clean
rm –f *.o reciprocnavrednost
$ make CFLAGS=-O2
gcc –O2 –c main.c
g++ –O2 –c reciprocnavrednost.cpp
g++ –O2 –o reciprocnavrednost main.o reciprocnavrednost.o
```

Vidite kako je `-02` postavljeno na mestu `$(CFLAGS)` u pravilima.

U ovom odeljku ste videli samo najosnovnije mogućnosti make-a. Više ćete saznati ako unesete:

```
info make
```

Debagovanje pomoću GNU debagera (GDB)

Debager je program koji koristite da biste otkrili zašto se vaš program ne ponaša onako kako vi mislite da bi trebalo. Ovo ćete često raditi. GNU debager (GDB) koristi većina Linux programera. Možete koristiti GDB da u koracima prođete kroz vaš kod, postavite tačke prekida i ispitate vrednosti lokalnih promenljivih.

Prevođenje sa informacijama za debagovanje

Da biste koristili GDB, morate pri prevođenju da uključite informacije za debagovanje. Uradite to dodavanjem `–g` opcije prevodiocu. Ako koristite Makefile kao što je prthodno opisano, možete samo da stavite da je `CFLAGS` jednako `–g` kada pokrenete make:

```
$ make CFLAGS=-g
gcc –g –c main.c
g++ –g –c reciprocnavrednost.cpp
g++ –g –o reciprocnavrednost main.o reciprocnavrednost.o
```

Kada prevodite sa `–g` opcijom, prevodilac ubacuje dodatnu informaciju u objektne i izvršne datoteke. Debager koristi ovu informaciju da bi saznao koje adrese odgovaraju kojim linijama koda u kojim datotekama, kako da ispiše lokalne promenljive i tako dalje.

Pokretanje GDB-a

Možete pokrenuti gdb unošenjem:

```
$ gdb reciprocnavrednost
```

Kada se gdb pokrene, trebalo bi da vidite GDB-ov odziv:

```
(gdb)
```

Prvi korak je da pokrenete vaš program unutar debagera. Samo unesite naredbu `run` i bilo koji programski parametar. Probajte da pokrenete program bez ikakvih parametara, ovako:

```
(gdb) run
Starting program: reciprocnavrednost
Program received signal SIGSEGV, Segmentation fault.
__strtol_internal (nptr=0x0, endptr=0x0, base=10, group=0)
at strtol.c:287
287  strtol.c: No such file or directory.
(gdb)
```

Problem je u tome što u main datoteci ne postoji kôd za proveru greške. Program očekuje jedan parametar, dok je u ovom slučaju program pokrenut bez ijednog parametra. Poruka `SIGSEGV` označava krah programa. GDB zna da se stvarni krah desio u funkciji koja se zove __strtol_internal. Ta funkcija se nalazi u standardnoj biblioteci i njen kôd ne postoji, što poruka `"No such file or directory"` i objašnjava. Stek možete videti korišćenjem `where` naredbe:

```
(gdb) where
```

```
#0   __strtol_internal (nptr=0x0, endptr=0x0, base=10, group=0)
     at strtol.c:287
#1   0x40096fb6 in atoi (nptr=0x0) at ../stdlib/stdlib.h:251
#2   0x804863e in main (argc=1, argv=0xbffff5e4) at main.c:8
```

Iz ovog prikaza možete videti da je main pozvala atoi funkciju sa NULL pokazivačem, što je i uzrok problema.

Možete otići dva nivoa naviše u steku dok ne dođete do `main` funkcije koristeći `up` naredbu:

```
(gdb) up 2
#2   0x804863e in main (argc=1, argv=0xbffff5e4) at main.c:8
8    i = atoi (argv[1]);
```

Primetite da gdb može da pronađe `main.c` kôd i da prikaže liniju gde se desio pogrešan poziv funkcije. Vrednosti promenljivih možete videti pomoću `print` naredbe:

```
(gdb) print argv[1]
$2 = 0x0
```

To potvrđuje da je zaista NULL pokazivač, prosleđen `atoi` funkciji, izazvao problem.

Tačku prekida možete postaviti koristeći `break` naredbu:

```
(gdb) break main
Breakpoint 1 at 0x804862e: file main.c, line 8.
```

Ova naredba postavlja tačku prekida na prvu liniju `main` koda. Sada probajte da ponovo pokrenete program sa parametrom, ovako:

```
(gdb) run 7
Starting program: reciprocnavrednost 7
Breakpoint 1, main (argc=2, argv=0xbffff5e4) at main.c:8
8    i = atoi (argv[1]);
```

Vidite da je debager stao kod tačke prekida.

Možete preskočiti poziv `atoi` funkcije pomoću `next` naredbe:

```
(gdb) next
9    printf ("Reciprocna vrednost od %d je %g\n", i,
reciprocnavrednost (i));
```

Ako želite da vidite šta se dešava unutar `reciprocnavrednost` funkcije, koristite `step` naredbu na sledeći način:

```
(gdb) step
```

```
reciprocnavrednost (i=7) at reciprocnavrednost.cpp:6
6    assert (i != 0);
```

Sada se nalazite u telu `reciprocnavrednost` funkcije.

Sistemski pozivi

Sistemski poziv je implementiran u kernelu Linux operativnog sistema. Kada program napravi sistemski poziv, argumenti se pakuju i predaju kernelu, koji preuzima izvršavanje programa sve dok se poziv ne završi. Sistemski poziv nije običan funkcijski poziv, tako da je potrebna specijalna procedura za predavanje kontrole kernelu. Međutim, GNU C biblioteka, tj. implementacija standardne C biblioteke, koja dolazi uz GNU/Linux sisteme, obuhvata funkcije sa Linux sistemskim pozivima tako da bi se oni lakše pozivali. I/O funkcije niskog nivoa kao što su `open` and `read` primeri su sistemskih poziva na Linux operativnom sistemu.

Skup Linux sistemskih poziva formira osnovni interfejs između programa i Linux kernela. Svaki poziv predstavlja osnovnu operaciju ili mogućnost. Neki sistemski pozivi su veoma moćni i mogu imati velikog uticaja na sistem. Na primer, neki sistemski pozivi vam omogućavaju da ugasite (*shut down*) Linux ili da alocirate sistemske resurse i sprečite druge korisnike da im pristupe. Ovi pozivi imaju restrikciju, da njih mogu pozovu samo procesi koje je kreirao korisnik sa superuser privilegijama (tj. programi koje je startovao `root`).

Funkcije mogu pozvati jednu ili više drugih funkcija ili sistemskih poziva u zavisnosti od toga kako su implementirane. Lista sistemskih poziva za vašu verziju kernela je data u `/usr/include/asm/unistd.h`. Neke od njih, operativni sistem koristi za internu upotrebu, a drugi se koriste samo pri implementaciji specijalnih funkcija.

U/I sistemski pozivi nižeg nivoa

C programeri u GNU/Linux sistemu imaju na raspolaganju dva skupa ULAZNO/IZLAZNIH funkcija. Standardna C biblioteka obezbeđuje U/I funkcije: printf, fopen, itd. Linux kernel obezbeđuje drugi skup U/I operacija, koje funkcionisu na nižem nivou od C funkcija.

Pošto je ova lekcija namenjena za ljude koji već poznaju jezik C, pretpostavljamo da ste se već sreli i da znate da koristite U/I funkcije C biblioteka.

Često postoje dobri razlozi za upotrebu Linux-ovih U/I funkcija nižeg nivoa. Mnoge od njih su sistemski pozivi kernela i one najvećim delom omogućavaju direktan pristup u osnovne mogućnosti sistema, koji su

raspoloživi za namenske programe. Ustvari standardne U/I rutine C biblioteka implementirane su u vrh nižeg nivoa Linux U/I sistemskih poziva. Korišćenje takvih funkcija uglavnom je najefikasniji (a ponekad i pogodniji) način da se izvedu ulazne i izlazne operacije.

Upotreba naredbe strace

Pre nego što počnemo da diskutujemo o sistemskim pozivima biće korisno da predstavimo komandu sa kojom možete naučiti više o sistemskim pozivima. Komanda `strace` prati izvršavanje nekog programa, listajući sve sistemske pozive koje program napravi i sve signale koje prima.

Kako bi pratili sve sistemske pozive i signale potrebno je jednostavno pozvati strace, navođenjem imena programa sa svojim argumentima u komandnoj liniji. Na primer, ukoliko želimo da pratimo sistemske pozive, koje poziva komanda `hostname` (komanda hostname pozvana bez parametara jednostavno štampa hostname računara na standardni izlaz) koristićemo sledeću komandu:

```
$ strace hostname
```

Ova komanda proizvodi nekoliko ekrana izlaznog teksta. Svaka linija se odnosi na samo jedan sistemski poziv. Za svaki sistemski poziv, ispisuje se ime tog sistemskog poziva, zatim njegovi argumenti (ili njihove skraćenice ukoliko su predugački) i njegova povratna vrednost. Gde je to moguće, `strace` prikazuje simbolička imena umesto numeričkih vrednosti za argumente i povratne vrednosti, kao i polja struktura koje su prosleđene pokazivačem u sistemski poziv. Napominjemo da `strace` ne može da prikaže pozive običnih funkcija.

U izlazu komande `strace hostname`, prva linija predstavlja `execve` sistemski poziv koji u stvari poziva `hostname` program:

```
execve("/bin/hostname", ["hostname"], [/* 49 vars */]) = 0
```

Prvi argument je ime programa koji se pokreće, drugi je njegova lista argumenata koja se sastoji od samo jednog elementa i treći je lista okruženja, koju strace izostavlja. Sledećih 30 redova deo su mehanizma koji učitava standardnu C biblioteku.

Pri kraju se nalaze sistemski pozivi koji pomažu da program završi posao. `Uname` je sistemski poziv koji se koristi da bi se pribavio hostname iz kernela.

```
uname({sys="Linux", node="myhostname", ...}) = 0
```

Primetite da `strace` obeležava polja (`sys` i `node`) u strukturi argumenata. Ovu strukturu popunjava sistemski poziv, popunjava se `sys`

polje sa imenom operativnog sistema, a `node` polje kao ime računara (hostname).

Na kraju, `write` sistemski poziv proizvodi izlaz. Deskriptor datoteke 1 predstavlja standardni izlaz. Treći argument je broj karaktera koji treba ispisati, a povratna vrednost predstavlja broj karaktera koji je u stvari ispisan.

```
write(1, "myhostname\n", 11) = 11
```

Kada pozovete `strace`, izlaz može biti teško čitljiv, zbog toga što se izlaz `hostname` programa može lako pomešati sa izlazom sitemskog poziva `strace`.

Ako program koji pratite proizvodi mnogo izlaza, ponekad je bolje preusmeriti izlaz u neku datoteku. Da bi ste ovo postigli koristite opciju `-o filename`.

Da bi se razumeo celokupan izlaz `strace -a`, potrebno je detaljno poznavanje Linux kernela, što prosečnom programeru i nije od presudne važnosti. Međutim, razumevanje je korisno za ispravljanje grešaka (debug) kod težih problema ili za razumevanje kako određeni programi funkcionišu.

Pronalaženje dodatnih informacija

Skoro svaka Linux distribucija dolazi sa mnoštvom korisne dokumentacije. Većinu onoga o čemu ćemo govoriti u ovoj knjizi možete naučiti čitajući dokumentaciju u okviru vaše Linux distribucije (mada bi vam to verovatno oduzelo više vremena). Dokumentacija nije uvek dobro organizovana, pa pronalaženje onoga što vam treba može da predstavlja problem. Takođe, dokumentacija je ponekad zastarela, tako da sve što čitate primite sa rezervom. Ako se sistem ne ponaša kao što stranice uputstva (*man pages*) kažu da bi trebalo, može da se desi da su stranice uputstva zastarele.

Da bismo vam pomogli u snalaženju, navešćemo vam najkorisnije izvore informacija o naprednom programiranju na Linux operativnom sistemu.

Stranice uputstva

Linux distribucije sadrže stranice uputstva (*man pages*, skraćeno od *manual pages*) za većinu osnovnih naredbi, sistemskih poziva i funkcija standardne biblioteke. Stranice uputstva su podeljene na sekcije označene brojevima, od kojih su za programere najznačajnije:

(1) Korisničke naredbe

(2) Sistemski pozivi

(3) Funkcije standardne biblioteke

(8) Sistemske/administrativne naredbe

Linux stranice uputstva dolaze instalirane na vašem sistemu, a pristupa im se naredbom man. Za pregled stranice uputstva jednistavno pozovite man ime, gde je ime naredba ili ime funkcije. U par slučajeva isto ime se javlja u više od jedne sekcije. Možete izričito navesti sekciju stavljanjem broja sekcije pre imena. Na primer, ako unesete sledeće, dobićete stranicu uputstva za `sleep` naredbu (u sekciji 1 Linux stranica uputstva):

```
$ man sleep
```

Da biste videli stranicu uputstva za `sleep` bibliotečku funkciju, koristite ovu naredbu:

```
$ man 3 sleep
```

Svaka stranica uputstva sadrži u jednij liniji teksta kratak opis naredbe ili funkcije. Naredba `whatis` ime prikazuje sve stranice uputstva (u svim sekcijama) za naredbu ili funkciju ime. Ako niste sigurni koju naredbu ili funkciju želite, možete da izvršite pretragu po ključnoj reči koristeći man `-k` ključna_reč.

Stranice uputstva sadrže veliki broj vrlo korisnih informacija i trebalo bi da budu prvo mesto na kome ćete potražiti pomoć. Stranica uputstva za naredbu opisuje opcije i parametre u komandnoj liniji, ulaz i izlaz, kodove grešaka, konfiguraciju i tome slično. Stranica uputstva za sistemske pozive ili bibliotečke funkcije opisuje parametre i povratne vrednosti, nabraja kodove grešaka i bočne efekte i navodi koju datoteku treba uključiti ako pozivate tu funkciju.

Info

Info dokumentacioni sistem poseduje detaljniju dokumentaciju za mnoge osnovne komponente GNU/Linux sistema, kao i za neke programe. Info stranice su dokumenti u hipertekst formatu, slični web stranicama. Za pokretanje tekstualnog Info pretraživača unesite info u komandnoj liniji. Prikazaće vam se meni Info dokumenata koji su instalirani na vašem sistemu. (Pritisnite <Ctrl-H> da biste prikazali tastere za navigaciju po Info dokumentu.)

Među najkorisnijim Info dokumentima su:

- `gcc` – gcc prevodilac
- `libc` – GNU C biblioteka, sadrži veliki broj sistemskih poziva

- `gdb` – GNU debager
- `emacs` – Emacs uređivač teksta
- `info` – sam Info sistem

Skoro svi standardni Linux alati za programiranje, kao što su `ld` povezivač, `as` asembler i `gprof` profajler) dolaze sa korisnim Info stranicama. Možete direktno pristupiti određenom Info dokumentu navodeći ime stranice u komandnoj liniji:

```
$ info libc
```

Datoteke zaglavlja

O sistemskim funkcijama koje su dostupne i o tome kako da ih koristite možete mnogo da naučite iz sistemskih datoteka zaglavlja (*header files*). One se nalaze u `/usr/include` i `/usr/include/sys` direktorijumima. Ako vam, na primer, prevodilac javlja greške zbog korišćenja sistemskog poziva, pogledajte da li je potpis funkcije u odgovarajućoj datoteci zaglavlja isti kao i u stranici uputstva.

Na Linux sistemima, veliki broj osnovnih i suštinskih detalja o tome kako sistemski pozivi rade, mogu se naći u datotekama zaglavlja koje se nalaze u `/usr/include/bits`, `/usr/include/asm` i `/usr/include/linux` direktorijumima. Recimo, numeričke vrednosti signala su definisane u datoteci `/usr/include/bits/signum.h`. Čitanje ovih datoteka zaglavlja će radoznalim umovima biti od koristi. Međutim, nemojte ove datoteke direktno uključivati u vaše programe. Uvek koristite datoteke zaglavlja iz `/usr/include` direktorijuma ili kako je već opisano u stranici uputstva za funkciju koju koristite.

Izvorni kôd

Ovde je izvorni kôd otvoren (engl. *open source*), zar ne? Poslednji izvor informacija o tome kako sistem radi je sam izvorni kôd sistema. Srećom po Linux programere, taj izvorni kôd je u potpunosti dostupan. Postoje šanse da uz vašu Linux distribuciju dolazi kompletan izvorni kod celog operativnog sistema i svih programa koji se uz nju isporučuju. U suprotnom, imate pravo da ga pod uslovima GNU General Public License zatražite od distributera. (Možda izvorni kôd samo nije instaliran na vašem disku. Potražite uputstva za njegovu instalaciju u dokumentaciji vaše distribucije.)

Izvorni kôd Linux kernela obično se nalazi u `/usr/src/linux` direktorijumu. Ako ova knjiga ne zadovoljava vašu radoznalost o tome kako procesi, deljena memorija i sistemski uređaji rade, uvek možete da učite direktno iz izvornog koda. Većina sistemskih funkcija koje su

opisane u ovoj knjizi su implementirane u GNU C biblioteci. Pogledajte u dokumentaciji vaše distribucije gde se nalazi izvorni kôd C biblioteke.

2. Preporuke za pisanje GNU/Linux softvera

Ovo poglavlje pokriva neke od tehnika koje većina GNU/Linux programera koristi. Prateći prikazane smernice, bićete u mogućnosti da pišete programe koji dobro rade u okviru GNU/Linux okruženja i koji ispunjavaju očekivanja GNU/Linux korisnika u pogledu rukovanja.

Komunikacija sa operativnim sistemom i okruženje

Kada ste prvi put učili C ili C++, naučili ste da je posebna main funkcija primarna ulazna tačka za program. Kada operativni sistem pokrene vaš program, ona automatski obezbeđuje određena sredstva koja pomažu programu da komunicira sa operativnim sistemom i korisnikom. Verovatno ste učili da postoje dva ulazna parametra main funkcije, uobičajeno nazivani argc i argv, koji prihvataju ulazne podatke u vaš program. Učili ste i o `stdin` i `stdout` funkcijama (ili `cin` i `cout` tokovima u C++-u) koje obezbeđuju konzolni ulaz i izlaz. Ove osobine obezbeđuju C i C++ jezici i one na određeni način interaguju sa GNU/Linux sistemom. GNU/Linux obezbeđuje i druge načine za interakciju sa operativnim okruženjem.

Argumenti komandne linije

Program pokrećete iz komandne linije unoseći ime programa. Opciono, programu možete proslediti dodatne informacije unoseći jednu ili više reči iza imena programa, odvojenih jednim praznim mestom. One se zovu argumenti komandne linije. (Takođe, možete proslediti i prazan karakter kao argument stavljajući ga pod znakove navodnika.) Uobičajeni naziv je lista argumenata programa, zato što oni ne moraju da poteknu iz komandne linije. U poglavlju o procesima videćete drugačiji način pokretanja programa u kojem jedan program može direktno da navede listu argumenata drugog programa.

Kada se program pokrene iz komandne linije, lista argumenata sadrži celu liniju komande uključujući naziv programa i svaki argument koji je eventualno naveden. Pretpostavimo da ste, na primer, pozvali ls naredbu u komandnoj liniji da biste prikazali sadržaj root direktorijuma, uključujući i podatke o veličini za svaku datoteku:

```
$ ls –s /
```

Lista argumenata koju program `ls` dobija ima tri elementa. Prvi je sam naziv programa, kako je navedeno u komandnoj liniji, `ls`. Drugi i treći elementi liste argumenata su dva argumenta komandne linije, `–s` i `/`.

Funkcija main vašeg programa može da pristupi listi argumenata preko `argc` i `argv` parametara (ako ih ne koristite, možete ih izostaviti). Prvi parametar, `argc`, je ceo broj i pokazuje koliko elemenata ima lista

argumenata. Drugi parametar, `argv`, je niz pokazivača na karaktere. Veličina niza je argc, a elementi niza pokazuju na elemente u listi argumenata kao nizovi karaktera završeni NULL-karakterom.

Korišćenje argumenata komandne linije je lako kao i ispitivanje sadržaja `argc` i `argv` parametara. Ako vas ne zanima samo ime programa, ne zaboravite da preskočite prvi element.

Listing 2.1 demonstrira kako se koriste `argc` i `argv` parametri.

```
#include <stdio.h>

int main (int argc, char* argv[])
{
  printf ("Ime ovog programa je '%s'.\n", argv[0]);
  printf ("Ovaj program je pozvan sa %d argumenata.\n",
           argc - 1);
  // Da li je naveden barem jedan argument komandne linije?
  if (argc > 1)
  {
    // Jeste, stampaj argumente.
    int i;
    printf ("Argumenti su:\n");
    for (i = 1; i < argc; ++i)
    printf (" %s\n", argv[i]);
  }
  return 0;
}
```

Listing 2.1 (arglist.c) Korišćenje argc i argv parametara

GNU/Linux propisi o komandnoj liniji

Skoro svi GNU/Linux programi ispunjavaju neke propise o tome kako se tumače argumenti komandne linije. Argumenti koje program očekuje spadaju u dve kategorije: opcije (ili flegovi) i ostali argumenti. Opcije utiču na ponašanje programa, dok ostali argumenti predstavljaju ulazne podatke (na primer, imena ulaznih datoteka).

Postoje dve forme opcija:

- Kratke opcije se sastoje od crtice i jednog karaktera (najčešće malo ili veliko slovo). Kratke opcije se brže unose.

- Duge opcije se sastoje od dve crtice iza kojih je ime sastavljeno od malih i velikih slova i crtica. Dugačke opcije se lakše pamte i lakše čitaju (u shell skriptovima, na primer).

Uglavnom, program obezbeđuje i kratku i dugu formu za većinu opcija koje podržava. Prvu za kratkoću, drugu za jasnoću. Na primer, većina programa razume opcije `–h` i `--help` i podjednako ih tretira. Standardno, kada se program pokreće iz komandne linije, bilo koja od željenih opcija dolazi odmah nakon imena programa. Neke opcije zahtevaju argument odmah u nastavku. Mnogi programi, na primer, tumače opciju `--output foo` tako da izlaz programa treba da se nađe u datoteci pod nazivom foo. Iza opcija se mogu nalaziti drugi argumenti komandne linije, najčešće ulazne datoteke ili ulazni podaci.

Na primer, naredba `ls –s /` prikazuje sadržaj root direktorijuma. Opcija `–s` menja podrazumevano ponašanje ls naredbe nalažući joj da prikaže veličinu (u kilobajtima) svake stavke. Argument `/` govori `ls` naredbi koji direktorijum da izlista. Opcija `--size` je sinonim za `–s`, pa se ista naredba mogla pozvati i kao `ls --size /`.

GNU standardi za kodiranje (GNU Coding Standards) sadrže listu imena uobičajeno korišćenih opcija komandne linije. Ako planirate da obezbedite bilo koju opciju sličnu ovima, dobra je ideja da koristite imena opisana u standardu za kodiranje. Vaš program će se ponašati slično drugim programima i korisnicima će biti lakši za učenje. Možete videti uputstva GNU standarda za kodiranje o opcijama komandne linije zadajući sledeću naredbu na većini GNU/Linux sistema:

```
$ info "(standards)User Interfaces"
```

Korišćenje getopt_long funkcije

Rasčlanjivanje opcija komandne linije je dosadan posao. Srećom, GNU C biblioteka nudi funkciju koju možete da koristite u C i C++ programima da biste ovaj posao učinili donekle jednostavnijim (mada i dalje pomalo neugodnim). Ova funkcija, `getopt_long`, razume i kratke i duge opcije. Ako koristite ovu funkciju, uključite `<getopt.h>` datoteku zaglavlja.

Pretpostavimo, na primer, da pišete program koji bi trebalo da prihvata sledeće tri opcije: prikazane u Tabeli 2.1.

Dodatno, program bi trebalo da prihvata eventualne argumente komandne linije koji predstavljaju imena ulaznih datoteka.

Da biste koristili `getopt_long`, morate obezbediti dve strukture podataka. Prva je niz karaktera koji sadrži validne kratke opcije, svaka sa po jednim slovom. Iza opcije koja zahteva argument stoji dvotačka.

Za vaš program, niz ho:v ukazuje da su validne opcije –h, –o i –v, s tim da druga opcija očekuje argument.

Kratka forma	Duga forma	Svrha
`–h`	`-- help`	Prikazuje kratko uputstvo
`–o ime_datoteke`	`-- output ime_datoteke`	Određuje ime izlazne datoteke
`–v`	`-- verbose`	Ispisuje opširne poruke

Tabela 2.1 Primer opcija programa

Da biste odredili dostupne duge opcije, pravite niz struct option elemenata. Svaki element odgovara jednoj dugoj opciji i ima četiri polja. U normalnim okolnostima, prvo polje je naziv duge opcije (kao niz karaktera, bez dve crtice). Drugo je 1 ako opcija prihvata argumente, a 0 ako ne prihvata. Treće je NULL, a četvrto je karakter konstanta koja predstavlja odgovarajuću kratku opciju. Poslednji element niza ima nule u svim poljima. Niz možete napisati ovako:

```
const struct option long_options[] = {
  { "help",     0, NULL, 'h' },
  { "output",   1, NULL, 'o' },
  { "verbose",  0, NULL, 'v' },
  { NULL,       0, NULL, 0   }
};
```

Funkciju `getopt_long` pozivate prosleđujući joj argc i argv argumente main funkcije, niz karaktera koji opisuje kratke opcije i niz struct option elemenata koji opisuje duge opcije.

- Svaki put kada pozovete `getopt_long`, ona izdvoji jednu opciju i vraća jednoslovnu oznaku te opcije ili –1 ako više nema opcija.
- Tipično, `getopt_long` ćete pozivati u petlji da biste prihvatili sve opcije koje je korisnik uneo i obrađivaćete ih ponaosob u okviru switch izraza.
- Ako getopt_long naiđe na nevažeću opciju (opciju koju niste naveli kao validnu kratku ili dugu opciju), ispisaće poruku o grešci i vratiće ? karakter (znak pitanja). Većina programa će u ovom slučaju prikazati kratko uputstvo i prekinuti sa radom.

- Kada se obrađuje opcija koja prihvata argumente, globalna promenljiva optarg pokazuje na tekst tog argumenta.
- Kada `getopt_long` završi sa obradom svih opcija, globalna promenljiva optind sadrži indeks prvog neopcionog argumenta (u `argv`-u).

Listing 2.2 prikazuje jedan primer kako možete da koristite `getopt_long` za obradu vaših argumenata.

```
#include <getopt.h>
#include <stdio.h>
#include <stdlib.h>

// Ime ovog programa
const char* program_name;

/* Štampanje korisnih informacija za ovaj program koristeći
STREAM (tipično stdout ili stderr) i izlaz iz programa
korišćenjem EXIT_CODE. */

void print_usage (FILE* stream, int exit_code)
{
  fprintf (stream,
    "Koriscenje:  %s opcije [ inputfile ... ]\n",
    program_name);
  fprintf (stream,
    " -h  --help             Prikaži ovu korisnu informaciju.\n"
    " -o  --output datoteka  Upiši izlaz u datoteku.\n"
    " -v  --verbose          Štampaj opširnu poruku.\n");
  exit (exit_code);
}

/* ARGC sadrži broj argumenata navedenih u komandnoj liniji
ARGV je niz pokazivača na argumente na komandnoj liniji*/

int main (int argc, char* argv[])
{
  int next_option;

  // Listanje validnih short opcija
  const char* const short_options = "ho:v";

  // Niz koji opisuje validne long opcije
  const struct option long_options[] = {
```

```
  { "help",     0, NULL, 'h' },
  { "output",   1, NULL, 'o' },
  { "verbose",  0, NULL, 'v' },
  { NULL,       0, NULL, 0   }
  // NULL je obavezan na kraju niza
};

// Ime datoteke za prihvatanje programskog izlaza ili
// NULL za standardni izlaz
const char* output_filename = NULL;

// Promenljiva koja reguliše ispis opširne poruke
int verbose = 0;

// Ime programa je uskladišteno u argv[0]
program_name = argv[0];
do {
  next_option = getopt_long (argc, argv, short_options,
    long_options, NULL);
  switch (next_option)
  {

  case 'h':   // -h ili --help
    /* Korisnik zahteva korisne informacije. Štampaj ih na
    standardni izlaz uz izlaz sa nulom (normalan završetak) */
    print_usage (stdout, 0);

  case 'o':   // -o ili --output
    /* Ova opcija zahteva argument – ime izlazne datoteke */
    output_filename = optarg;
    break;

  case 'v':   // -v ili --verbose
    verbose = 1;
    break;

  case '?':   // Korisnik je izabrao nevalidnu opciju
    /* Štampaj korisnu informaciju na standardni izlaz i
    izađi sa kodom jedan (nenormalan završetak) */
    print_usage (stderr, 1);

  case -1:    // Kraj opcija
    break;

  default:    // Neka nenavedena mogućnost
```

```
      abort ();
    }
  }
  while (next_option != -1);

  if (verbose) {
    int i;
    for (i = optind; i < argc; ++i)
      printf ("Argument: %s\n", argv[i]);
  }

  // Glavni program dolazi ovde

  return 0;
}
```

Listing 2.2 (getopt_long.c) Korišćenje getopt_long funkcije

Korišćenje `getopt_long` funkcije možda deluje komplikovano, ali kada biste sami pisali kôd koji obrađuje opcije komandne linije trebalo bi vam mnogo više vremena. Funkcija `getopt_long` je veoma sofisticirana i veoma fleksibilna u pogledu opisivanja vrste opcija koje prihvata. Ipak, bolje je ostaviti napredne mogućnosti sa strane i držati se osnovne strukture koja je ovde prikazana.

Standardni ulaz/izlaz

Standardna C biblioteka obezbeđuje standardne ulazne i izlazne tokove (`stdin` i `stdout`, redom). Njih koriste `scanf`, `printf` i druge bibliotečke funkcije. Po UNIX tradiciji, standardni ulaz i izlaz su uobičajeni za GNU/Linux programe. To omogućava ulančavanje više programa u pipeline korišćenjem | znaka (pipe - cev) i preusmeravanjem ulaza i izlaza. (Pogledajte stranicu uputstva (*man page*) za vaš komandni interpreter (shell) da biste naučili njegovu sintaksu.)

C biblioteka takođe obezbeđuje i `stderr`, standardni tok za greške. Programi bi trebalo da poruke o upozorenjima i greškama šalju kroz standardni tok za greške, a ne kroz standardni izlaz. Ovo omogućava korisnicima da razdvoje normalan izlaz i poruke o greškama, tako što će, na primer, preusmeriti standardni izlaz u datoteku, dok će se poruke o greškama ispisivati u komandnoj liniji. Funkcija `fprintf` se može koristiti za prikaz stderr, na primer:

```
fprintf (stderr, ("Error: ..."));
```

Ova tri toka su takođe dostupna pomoću osnovnih UNIX I/O naredbi (`read`, `write`, itd.) putem datotečkih indeksa. Ti indeksi su 0 za `stdin`, 1 za `stdout` i 2 za `stderr`.

Pri pokretanju programa, ponekad je korisno preusmeriti i standardni izlaz i standardni tok za greške u datoteku ili pipe. Sintaksa za ovo se razlikuje od jednog do drugog komandnog interpretera. Za Bourne tipove komandnih interpretera (uključujući bash, podrazumevani komandni interpreter na većini GNU/Linux distribucija) sintaksa je sledeća:

```
$ program > output_file.txt 2>&1
$ program 2>&1 | filter
```

Sintaksa `2>&1` označava da datotečki indeks 2 (`stderr`) treba da se priključi datotečkom indeksu 1 (`stdout`). Obratite pažnju na to da `2>&1` mora da stoji iza preusmeravanja (prvi primer), a ispred pipe-a (drugi primer).

Ne treba zaboraviti da je stdout baferovan. Podaci upisani u `stdout` se ne šalju komandnoj liniji (ili nekom drugom uređaju, ako su preusmereni) dok se bafer ne napuni, dok se program prirodno ne završi ili dok se stdout tok ne zatvori. Bafer možete eksplicitno isprazniti pozivajući sledeću naredbu:

```
fflush (stdout);
```

Sa druge strane, `stderr` nije baferovan. Podaci upisani u `stderr` idu direktno u komandnu liniju.

To može da proizvede neke iznenađujuće posledice. Na primer, ova petlja ne piše tačku svake sekunde, već se tačke čuvaju u baferu, a gomila tačaka se ispiše kada se bafer napuni.

```
while (1) {
  printf (".");
  sleep (1);
}
```

U ovoj petlji, međutim, tačke se pojavljuju svake sekunde:

```
while (1) {
  fprintf (stderr, ".");
  sleep (1);
}
```

Izlazni kodovi programa

Kada se program završi, on označava svoje stanje izlaznim kodom. Izlazni kôd je mali ceo broj. Po dogovoru izlazni kôd 0 (nula) označava uspešno izvršavanje, dok izlazni kodovi različiti od nule označavaju da se desila greška. Neki programi koriste različite vrednosti izlaznih kodova (različitih od nule) da bi istakli posebne greške.

U većini komandnih interpretera moguće je dobaviti izlazni kôd prethodno izvršenog programa koristeći posebnu `$?` promenljivu. Evo primera u kojem je ls naredba dva puta pozvana i u kojem je posle svakog poziva prikazan izlazni kôd. U prvom slučaju, naredba `ls` se ispravno izvršila i vratila izlazni kôd nula. U drugom slučaju, `ls` nailazi na grešku (zato što datoteka sa zadatim imenom ne postoji), pa prema tome vraća izlazni kôd različit od nule.

```
$ ls /
bin   coda  etc   lib         misc  nfs  proc  sbin  usr
boot  dev   home  lost+found  mnt   opt  root  tmp   var

$ echo $?
0

$ ls bogusfile
ls: bogusfile: No such file or directory

$ echo $?
1
```

C ili C++ program određuje svoj izlazni kôd vraćanjem njegove vrednosti iz main funkcije. Postoje i drugi načini da se obezbede izlazni kodovi. Posebni izlazni kodovi se dodeljuju programima koji se završavaju nepravilno (putem signala).

Okruženje

GNU/Linux svakom programu koji se izvršava obezbeđuje okruženje. Okruženje je kolekcija promenljiva/vrednost parova. I imena promenljivih okruženja, i njihove vrednosti su nizovi karaktera. Po dogovoru, imena promenljivih okruženja se pišu velikim slovima.

Verovatno ste već upoznati sa nekoliko opštih promenljivih okruženja. Na primer:

- `USER` sadrži vaše korisničko ime.
- `HOME` sadrži putanju do vašeg home direktorijuma.

- `PATH` sadrži listu direktorijuma odvojenih dvotačkom u kojima Linux traži naredbe koje vi pozivate.
- `DISPLAY` sadrži ime i broj ekrana X Window servera na kome se prikazuju prozori grafičkih X Window programa.

Vaš komandni interpreter, kao i svaki drugi program, ima svoje okruženje. Komandni interpreteri obezbeđuju metode za direktno pregledanje i izmenu okruženja. Za prikaz tekućeg okruženja u vašem komandnom interpreteru, pozovite program printenv. Različiti komandni interpreteri imaju drugačiju ugrađenu sintaksu za korišćenje promenljivih okruženja. U primerima koji slede koristićemo sintaksu Bourne tipova komandnih interpretera.

Komandni interpreter automatski pravi svoje promenljive za svaku promenljivu okruženja koju pronađe, tako da možete da pristupite vrednostima tih promenljivih pomoću `$ime_promenljive` sintakse. Na primer:

```
$ echo $USER
stojko

$ echo $HOME
/home/stojko
```

Možete da koristite `export` naredbu da biste izvezli promenljivu komandnog interpretera u okruženje. Na primer, za postavljanje `EDITOR` promenljive okruženja uradićete sledeće:

```
$ EDITOR=emacs
$ ex port EDITOR
```

Ili, ukratko:

```
$ export EDITOR=emacs
```

U programu, promenljivoj okruženja pristupate `getenv` funkcijom iz `<stdlib.h>`. Ova funkcija uzima ime promenljive, a vraća njenu vrednost kao niz karaktera ili NULL ako ta promenljiva nije definisana u okviru okruženja. Za postavljanje i brisanje promenljivih okruženja koristite `setenv` i `unsetenv` funkcije.

Nabrojati sve promenljive jednog okruženja zahteva malo veštine. Da biste to izveli, morate pristupiti globalnoj promenljivoj `environ`, koja je definisana u GNU C biblioteci. Ova promenljiva, tipa `char**`, je NULL-završen niz pokazivača na nizove karaktera. Svaki niz karaktera sadrži jednu promenljivu okruženja u formi `PROMENLJIVA=vrednost`.

Program u Listingu 2.3, na primer, jednostavno prikazuje celo okruženje prolazeći kroz `environ` niz u petlji.

```
#include <stdio.h>

// promenljiva environ sadrži podatke o okruženju
extern char** environ;

int main ()
{
  char** var;
  for (var = environ; *var != NULL; ++var)
    printf ("%s\n", *var);
  return 0;
}
```

Listing 2.3 (print-env.c) Prikazivanje izvršnog okruženja

Nemojte samostalno da menjate environ, već za to koristite setenv i unsetenv funkcije.

Obično, kada se novi program pokrene, on nasleđuje kopiju okruženja programa koji ga je pokrenuo (komandnog interpretera, ako je iz njega pokrenut). Tako, na primer, programi koje pokrećete iz komandne linije mogu ispitivati vrednosti promenljivih okruženja koje ste postavili u komandnom okruženju.

Promenljive okruženja se uglavnom koriste za saopštavanje konfiguracionih informacija programima. Recimo da pišete program koji se povezuje na Internet server da bi prikupio neke informacije. Mogli biste da napišete program tako da se ime servera navodi u komandnoj liniji. Međutim, pretpostavimo da ime servera korisnici neće često menjati. Možete koristiti posebnu promenljivu okruženja, na primer SERVER_NAME, za određivanje imena servera, a ako ta promenljiva ne postoji koristiće se podrazumevana vrednost. Deo vašeg programa bi mogao da izgleda kao što je prikazano u Listingu 2.4.

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
  char* server_name = getenv ("SERVER_NAME");
  if (server_name == NULL)
    /* SERVER_NAME promenljiva okruženja nije postavljena,
       pa se koristi njena podrazumevana vrednost */
    server_name = "server.my-company.com";
```

```
  printf ("pristupanje serveru %s\n", server_name);
  // Ovde se pristupa serveru
  return 0;
}
```

Listing 2.4 (client.c) Deo programa mrežnog klijenta

Recimo da je ovaj program preveden u izvršnu datoteku `client` koja se nalazi u tekućem direktorijumu. Pod pretpostavkom da niste postavili `SERVER_NAME` promenljivu, koristiće se podrazumevana vrednost za ime servera:

```
$ ./client
accessing server server.my-company.com
```

Ali se lako može navesti i drugo ime servera:

```
$ export SERVER_NAME=backup-server.elsewhere.net
$ ./client
accessing server backup-server.elsewhere.net
```

Korišćenje privremenih datoteka

Ponekad program treba da napravi privremenu datoteku za prosleđivanje podataka drugom programu ili za privremeno skladištenje velike količine podataka. Na GNU/Linux sistemima, privremene datoteke se čuvaju u `/tmp` direktorijumu. Kada koristite privremene datoteke, morate biti svesni sledećih zamki:

- Više od jedne instance vašeg programa se može simultano izvršavati (od strane istog ili nekog drugog korisnika). Instance bi trebalo da koriste različite privremene datoteke, da ne bi došlo do kolizije.
- Dozvole za pristup privremenoj datoteci bi trebalo postaviti tako da neautorizovani korisnici ne mogu da promene izvršavanje programa modifikovanjem ili zamenom privremene datoteke.
- Imena privremenih datoteka bi trebalo da se generišu na način koji je spolja nemoguće predvideti. U suprotnom, napadač može da iskoristi kašnjenje između testiranja da li je dato ime već u upotrebi i otvaranja nove privremene datoteke.

GNU/Linux nudi funkcije `mkstemp` i `tmpfile`, koje rešavaju ove probleme umesto vas (uz još par funkcija koje to ne rade). Koju ćete koristiti zavisi od toga da li planirate da privremenu datoteku isporučite

drugom programu, ili želite da koristite UNIX I/O (`open`, `write`, itd.) ili I/O funkcije C biblioteke (`fopen`, `fprintf`, itd.).

Korišćenje funkcije mkstemp

Funkcija `mkstemp` kreira jedinstveno ime privremene datoteke iz šablona imena datoteka, kreira datoteku sa dozvolama pristupa tako da samo tekući korisnik može da joj pristupi i otvara datoteku za čitanje i pisanje. Šablon imena datoteka je niz karaktera koji se završava sa "XXXXXX" (šest velikih slova X). Funkcija `mkstemp` na njihovo mesto stavlja karaktere tako da je ime datoteke jedinstveno. Povratna vrednost je datotečki indeks. Koristite `write` familiju funkcija za upis u privremenu datoteku.

Privremene datoteke kreirane od strane `mkstemp` funkcije ne brišu se automatski. Na vama je da obrišete privremenu datoteku kada za njom više nema potrebe. (Programeri bi trebalo da budu veoma pažljivi i da uklanjaju privremene datoteke, jer će se u suprotnom `/tmp` sistem datoteka na kraju prepuniti i učiniti sistem neupravljivim.) Ako je privremena datoteka samo za internu upotrebu i neće se prosleđivati drugom programu, nije loše da se nad njom odmah pozove unlink funkcija. Funkcija unlink uklanja vezu datoteke sa direktorijumom, ali pošto se datoteke logički nalaze u sistemu datoteka preko svojih referenci, ona se ne uklanja sve dok je i jedan datotečki indeks otvoren za tu datoteku. Na ovaj način, vaš program će moći da nastavi sa korišćenjem privremene datoteke, a ona će se automatski ukloniti čim zatvorite datotečki indeks. Pošto Linux zatvara datotečke indekse kada program završi sa radom, privremena datoteka će biti obrisana čak i ako se vaš program završi nepravilno.

Parom funkcija u Listingu 2.5 je prikazana upotreba `mkstemp` funkcije. Koristeći ih zajedno, ove funkcije olakšavaju upis prihvatne memorije (bafera) u privremenu datoteku (tako da memorija može da se oslobodi ili ponovo koristi) i njeno kasnije učitavanje.

```
#include <stdlib.h>
#include <unistd.h>

temp_file_handle write_temp_file (char* buffer, size_t length)
{
  /* Kreiranje datoteke i imena datoteke. XXXXXX će biti
  zamenjeno karakterima koji jednoznačno određuju datoteku */
  char temp_filename[] = "/tmp/temp_file.XXXXXX";
  int fd = mkstemp (temp_filename);
```

```
  unlink (temp_filename);

  // Prvo upiši broj bajtova u datoteku
  write (fd, &length, sizeof (length));

  // Upiši sada podatke
  write (fd, buffer, length);

  return fd;
}

/* Pročitaj sadržaj privremene datoteke TEMP_FILE koja je
kreirana funkcijom write_temp_file. Povratna vrednost je
novoalocirani bafer sa tim sadržajem koga korisnik mora
dealocirati naredbom free. *LENGTH predstavlja veličinu
sadržaja u bajtovima. Pomoćna datoteka je obrisana. */

char* read_temp_file (temp_file_handle temp_file,
                      size_t* length)
{
  char* buffer;
  int fd = temp_file;

  // Vrati se na početak datoteke
  lseek (fd, 0, SEEK_SET);

  // Učitaj veličinu podataka u privremenu datoteku
  read (fd, length, sizeof (*length));

  // Alociraj bafer i učitaj podatke
  buffer = (char*) malloc (*length);
  read (fd, buffer, *length);

  /* Zatvori deskriptor datoteke što će prouzrokovati
  nestanak privremene datoteke */
  close (fd);
  return buffer;
}
```

Listing 2.5 (temp_file.c) Korišćenje mkstemp funkcije

Korišćenje funkcije tmpfile

Ako koristite I/O funkcije C biblioteke i nemate potrebu da prosledite privremenu datoteku drugom programu, možete da koristite `tmpfile` funkciju. Ona kreira i otvara privremenu datoteku i vraća pokazivač na

nju. Privremena datoteka je već u startu nepovezana, kao u prethodnom primeru, tako da će se automatski obrisati kada se datotečki indeks zatvori ili kada program završi sa radom.

GNU/Linux vam nudi još nekoliko funkcija za generisanje privremenih datoteka i njihovih imena uključujući `mktemp`, `tmpnam` i `tempnam`. Ipak, nije dobro koristiti ove funkcije pošto pate od problema sa sigurnošću i pouzdanošću koje smo već pomenuli.

Pisanje pouzdanog koda

Pisanje programa koji rade ispravno pri normalnoj upotrebi nije lako. Pisanje programa koji se elegantno ponašaju pri nailaženju na greške još je teže. Ovaj odeljak opisuje neke tehnike pri pisanju programa za rano pronalaženje grešaka i za otkrivanje i oporavak od problema u programu koji se izvršava. Primeri koda prikazani kasnije u ovoj knjizi namerno ne sadrže opširnu proveru grešaka i kôd za oporavak zato što bi u tom slučaju osnovna funkcionalnost koja se prezentuje bila manje jasna.

Korišćenje assert makroa

Korisna smernica koju bi trebalo imati na umu kada se piše program, je da bi bagovi i neočekivane greške trebalo da uočljivo obore program što je ranije moguće. To će vam pomoći da ranije pronađete greške u fazi razvoja i testiranja. Greške koje se ne prikazuju tako uočljivo se često previde i ne javljaju se dok program ne stigne u ruke korisnika.

Jedna od najjednostavnijih metoda za proveru neočekivanih stanja je standardni C makro `assert`. Ulazni parametar ovog makroa je Boolean izraz. Ako je izraz netačan, program se prekida pošto se ispiše poruka o grešci koja sadrži datoteku izvornog koda, broj linije i tekst izraza. Makro assert je veoma koristan za različite provere postojanosti unutar programa. Na primer, za proveru validnosti parametara funkcije, za proveru stanja funkcije pre i posle poziva (ili poziva metode u C++ jeziku) i za proveru povratnih vrednosti.

Svako korišćenje `assert` makroa ne služi samo za proveru stanja pri izvršavanju, već i kao dokumentacija o radu programa unutar izvornog koda. Ako vaš program sadrži izraz `assert` (uslov) koji govori nekome ko čita vaš izvorni kôd da bi uslov uvek trebalo da bude tačan na tom mestu u programu i ako je uslov netačan, onda je to verovatno greška u programu.

Za kôd kod koga su performanse od kritičnog značaja, provere pri izvršavanju kao što je upotreba `assert` makroa mogu značajno da utiču na obaranje performansi. U ovim slučajevima možete da prevodite vaš

kôd definisanjem `NDEBUG` makroa dodavanjem `–DNDEBUG` opcije u komandnoj liniji prilikom prevođenja. Ako dodajete makro u sam kôd, onda on treba da bude dodat pre zaglavlja `<assert.h>`. Setovanjem `NDEBUG` makroa, svako pojavljivanje `assert` makroa će biti zanemareno pri prevođenju. Ipak, ovo bi trebalo raditi samo kada je neophodno zbog performansi i samo sa datotekama izvornog koda kod kojih su performanse od kritičnog značaja.

Pošto je moguće da se pri prevođenju zanemari upotreba `assert` makroa, vodite računa da bilo koji izraz koji koristite sa `assert` makroom ne pravi bočne efekte. Naročito, ne bi trebalo da pozivate funkcije unutar `assert` izraza, da definišete promenljive ili da koristite operatore kao što je `++`.

Recimo da pozivate funkciju `do_something` u petlji. Funkcija `do_something` vraća nulu pri uspešnom, a vrednost različitu od nule pri neuspešnom izvršenju. Ali, vi ne očekujete da će se ona ikada neuspešno izvršiti u vašem programu. Možda ćete poželeti da napišete:

```
for (i = 0; i < 100; ++i)
  assert (do_something () == 0);
```

Međutim, možda uvidite kako ova provera previše obara performanse pri izvršavanju i odlučite se da ponovo prevedete kôd sa definisanim `NDEBUG` makroom. To će u potpunosti isključiti pozivanje `assert` makroa, tako da se izraz nikad neće proveravati i `do_something` se nikada neće izvršiti. Umesto toga, trebalo bi da napišete ovo:

```
for (i = 0; i < 100; ++i) {
  int status = do_something ();
  assert (status == 0);
}
```

Još jedna stvar koju treba imati na umu je da `assert` ne bi trebalo da koristite za proveru unosa korisnika. Korisnici ne vole kada se aplikacije samo prekinu sa kritičnom porukom o grešci, čak i pri neodgovarajućem unosu. I pored toga, uvek bi trebalo da proveravate unos i pravite razumljive poruke o greškama kao odgovor. Koristite assert samo za unutrašnju proveru pri izvršavanju programa.

Neka od mesta na kojima je dobro koristiti `assert` su ova:

- Provera null pokazivača, na primer, kao neodgovarajućeg parametra funkcije. Poruka o grešci generisana od strane {`assert (pointer != NULL)`},

  ```
  Assertion 'pointer != ((void *)0)' failed.
  ```

je mnogo jasnija od poruke o grešci koja bi se prikazala da je vaš program opozvao null pokazivač:

```
Segmentation fault (core dumped)
```

- Provera vrednosti ulaznih parametara funkcije. Na primer, ako bi funkciju trebalo pozivati isključivo sa pozitivnom vrednošću za parametar `foo`, na početku tela funkcije koristite ovo:

  ```
  assert (foo > 0);
  ```

  Ovo će vam pomoći u otkrivanju pogrešnog poziva funkcije, a i veoma jasno ukazuje nekom ko čita kôd funkcije da postoji ograničenje u pogledu vrednosti parametra.

Ne uzdržavajte se, već slobodno koristite `assert` u vašim programima.

Greške pri sistemskim pozivima

Većina nas je prvobitno učila kako da piše programe koji se do kraja izvršavaju na tačno određen način. Program delimo na zadatke i podzadatke i svaka funkcija izvršava zadatak pozivajući druge funkcije za izvršenje odgovarajućih podzadataka. Pri odgovarajućem ulazu očekujemo da funkcija proizvede korektan izlaz i bočne efekte.

Realnost računarskog hardvera i softvera upada u ovaj idealizovani san. Računari imaju ograničene resurse, hardver otkazuje, više programa se izvršava istovremeno, korisnici i programi čine greške. Često se na granici između aplikacije i operativnog sistema pojavljuju ove realnosti. Prema tome, kada se koriste sistemski pozivi za pristup resursima sistema, za izvršenje I/O ili za neke druge potrebe, nije samo važno da se razume šta se dešava kada se poziv izvrši, već i kako i kada poziv može da ne uspe.

Sistemski pozivi mogu da se neuspešno izvrše iz više razloga. Na primer:

- Sistem može da ostane bez resursa (ili program može da premaši limite resursa koje mu sistem dodeljuje). Na primer, program može da pokuša da alocira previše memorije, da prekomerno piše po disku ili da istovremeno otvori previše datoteka.
- Linux može da blokira određeni sistemski poziv kada program pokušava da izvrši operaciju za koju nema dozvolu. Na primer, program može da pokuša da izvrši upis u datoteku koja je označena samo za čitanje (read-only), da pristupi memoriji drugog procesa ili da ubije program drugog korisnika.

- Argumenti sistemskog poziva mogu da budu neodgovarajući što zbog pogrešnog unosa korisnika, što zbog greške u programu. Recimo, program može da prosledi pogrešnu adresu memorije ili pogrešan datotečki indeks sistemskom pozivu. Ili, program može da pokuša da otvori direktorijum kao običnu datoteku ili može da prosledi ime datoteke sistemskom pozivu koji očekuje direktorijum.
- Sistemski poziv može da se ne izvrši iz razloga ne vezanih za program. To se najčešće dešava kada sistemski poziv pristupa hardveru. Uređaj može biti pokvaren ili može da ne podržava određenu operaciju, ili možda disk nije stavljen u čitač diskova.
- Sistemski poziv može ponekad biti prekinut od strane nekog spoljašnjeg događaja, kao što je isporuka signala. To možda ne znači da se desila stvarna greška, ali odgovornost je na programu da, ako je potrebno, ponovi sistemski poziv.

U dobro napisanom programu koji u velikoj meri koristi sistemske pozive, čest je slučaj da se više koda koristi za otkrivanje i obradu grešaka i ostalih izuzetaka nego za osnovni zadatak programa.

Kodovi grešaka za sistemske pozive

Većina sistemskih poziva vraća nulu ako se operacija uspešno izvršila ili vrednost različitu od nule ako se operacija nije uspešno izvršila. (Ipak, mnogi sistemski pozivi imaju drugačija pravila. Na primer, `malloc` vraća null pokazivač u slučaju greške. Uvek pažljivo čitajte stranice uputstva kada koristite sistemske pozive.) Iako su ove informacije dovoljne da bismo odredili da li bi program trebalo normalno da nastavi sa radom, one verovatno nisu dovoljne za pažljiv oporavak od grešaka.

Većina sistemskih poziva koristi posebnu promenljivu `errno` u koju smešta dodatne informacije u slučaju greške. Kada poziv ne uspe, sistem postavlja `errno` na vrednost koja opisuje zbog čega je do greške došlo. Pošto svi sistemski pozivi koriste istu `errno` promenljivu za čuvanje informacije o grešci, trebalo bi da odmah po neuspešnom pozivu iskopirate njenu vrednost u drugu promenljivu. Vrednost `errno` promenljive će biti obrisana kada sledeći put izvršite sistemski poziv.

Vrednosti koje opisuju greške su celobrojne, a moguće vrednosti su date u pretprocesorskim makroima. Po dogovoru njihovi nazivi se sastoje isključivo od velikih slova i počinju sa "E", kao na primer `EACCES` i `EINVAL`. Uvek koristite ove makroe za označavanje `errno` vrednosti, a ne njihove celobrojne vrednosti. U slučaju da koristite errno vrednosti, uključite `<errno.h>` zaglavlje.

GNU/Linux nudi zgodnu funkciju, `strerror`, koja vraća opis errno koda kao niz karaktera pogodnih za korišćenje kod pisanja poruka o grešci. Uključite `<string.h>` ako koristite `strerror`.

GNU/Linux takođe obezbeđuje i funkciju `perror`, koja upisuje opis greške direktno u `stderr` tok. Možete proslediti `perror` funkciji niz karaktera koji će se ispisivati pre opisa greške, koji bi obično trebalo da sadrži ime funkcije koja se nije pravilno izvršila. Ako koristite `perror`, uključite `<stdio.h>` zaglavlje.

U ovom delu koda pokušavamo da otvorimo datoteku. Ako otvaranje ne uspe, ispisujemo poruku o grešci i prekidamo izvršavanje programa. Obratite pažnju na to da `open` poziv vraća datotečki indeks ako je operacija uspela ili –1 ako nije.

```
fd = open ("inputfile.txt", O_RDONLY);
if (fd == -1) {
  /* Otvaranje datoteke nije uspelo.
  Ispiši poruku o grešci i izađi */
  fprintf (stderr, "greska pri otvaranju datoteke: %s\n",
    strerror (errno));
  exit (1);
}
```

U zavisnosti od vašeg programa i prirode sistemskog poziva odgovarajuća akcija u slučaju greške bi mogla da bude ispisivanje poruke o grešci, opoziv operacije, prekid programa, ponovni pokušaj ili čak zanemarivanje greške. Ipak, važno je da se obezbedi logika koja će na neki način da obradi sve moguće varijante grešaka.

Jedan od mogućih kodova grešaka na koji bi trebalo da obratite pažnju, pogotovo kod I/O funkcija, je `EINTR`. Neke funkcije, kao što su `read`, `select` i `sleep`, mogu da oduzmu značajno vreme za izvršenje. One se smatraju za blokirajuće funkcije zato što se izvršavanje programa blokira dok se poziv ne završi. Međutim, ako program primi signal dok je blokiran u jednom od ovih poziva, poziv se vraća ne završivši operaciju. U tom slučaju, errno je setovan na `EINTR`. Tada ćete obično želeti da ponovite sistemski poziv.

Podsetimo da u normalnim situacijama izvršna datoteka ne sadrži nikakve reference na originalni izvorni kôd. Izvršna datoteka je prosto niz mašinskih instrukcija koje su stvorene od strane prevodioca. Ovo nije dovoljno za debagovanje i znatno otežava traženje mesta eventualne greške u izvornoj datoteci. GCC obezbeđuje `-g debug` opciju koja omogućava vezu između mašinske instrukcije gde se dogodila greška i odgovarajuće linije u izvornom kodu. Takođe, kada program nenormalno završi rad, informacija o stanju programa u trenutku

prekida čuva se u takozvanoj *core* datoteci. Kombinacija core datoteke i `-g` opcije omogućava uspešno debagovanje programa. Evo dela koda koji koristi `chown` poziv da bi korisniku navedenom navedenog identifikatorom `user_id` dodelio vlasništvo nad datotekom navedenom parametrom `path`. Ako poziv ne uspe, program deluje u zavisnosti od vrednosti `errno` promenljive. Obratite pažnju da kada otkrijemo da verovatno postoji greška u programu, prekidamo program pomoću `abort` ili `assert` funkcija koje prouzrokuju generisanje core datoteke. To može da bude korisno za debagovanje po prekidu programa. Za ostale fatalne (neotklonjive) greške, kao što je nedostatak memorije, program prekidamo pomoću `exit` naredbe i izlazne vrednosti različite od nule, zato što ne bismo imali naročite koristi od core datoteke.

```
rval = chown (path, user_id, -1);
if (rval != 0) {
  // Sačuvaj errno jer će biti uništen u sledećem sist. pozivu
  int error_code = errno;
  // Operacija nije uspela; chown treba da vrati -1 zbog greške
  assert (rval == -1);

  // Proveri errno vrednost i pokreni odgovarajuću akciju
  switch (error_code) {
  case EPERM:         // Dozvola je odbijena.
  case EROFS:         // PATH je na read-only sistemu datoteka.
  case ENAMETOOLONG:  // PATH je previše dugačak.
  case ENOENT:        // PATH ne postoji
  case ENOTDIR:       // Deo promenljive PATH nije direktorijum
  case EACCES:        // Deo promenljive PATH nije dostupan
    // Nešto nije u redu sa datotekom. Izpiši poruku o grešci
    fprintf (stderr, "error changing ownership of %s: %s\n",
             path, strerror (error_code));
    break;

  case EFAULT:
    // PATH sadrži nelegalnu memorijsku adresu (verovatno "bug")
    abort ();

  case ENOMEM:
    // "Ispucana" memorija za jezgro
    fprintf (stderr, "%s\n", strerror (error_code));
    exit (1);

  default:
    /* Sve ostale, neočekivane greške. Pokušali smo da
    obradimo sve – ako smo neku propustili – to je "bug" */
```

```
      abort ();
    };
}
```

Mogli ste da koristite i ovaj kôd, koji se istovetno ponaša u slučaju uspešnog poziva:

```
rval = chown (path, user_id, -1);
assert (rval == 0);
```

Ali ako se poziv ne izvrši uspešno, ovo alternativno rešenje neće pokušati da izvesti, obradi ili da se oporavi od greške.

Da li ćete koristiti prvi oblik, drugi oblik ili nešto između, zavisi od detekcije grešaka i zahteva za oporavkom vašeg programa.

Greške i dodela resursa

Kada se sistemski poziv neuspešno izvrši često je prikladno opozvati tekuću operaciju ali i ne prekidati program, zato što postoji mogućnost oporavka od greške. Jedan od načina da se to izvede je da se vratimo iz tekuće funkcije, prosleđujući onom ko je poziva povratni kôd koji ukazuje na grešku.

Ako odlučite da se vratite negde iz sredine funkcije, važno je da se svi prethodno uspešno dodeljeni resursi u funkciji, prvo oslobode. Ovi resursi mogu da budu memorija, datotečki indeksi, pokazivači na datoteke, privremene datoteke, objekti za sinhronizaciju i tako dalje. U suprotnom, ako vaš program nastavi sa radom, resursi koji su dodeljeni pre nailaska na grešku će biti potrošeni.

Razmotrite, na primer, funkciju koja čita sadržaj datoteke i smešta ga u prihvatnu memoriju (bafer). Ta funkcija bi mogla da prati sledeće korake: 1. dodeli prihvatnu memoriju, 2. otvori datoteku, 3. čitaj sadržaj datoteke i smesti ga u prihvatnu memoriju, 4. zatvori datoteku, 5. vrati sadržaj prihvatne memorije.

Ako datoteka ne postoji, korak 2 neće uspeti. Podesna akcija bi mogla da bude da funkcija vrati NULL. Međutim, ako je prihvatna memorija već bila dodeljena u koraku 1, postoji rizik od curenja memorije. Ne smete da zaboravite da oslobodite prihvatnu memoriju na bilo kom mestu u toku programa sa koga se ne vraćate. Ako korak 3 ne uspe, ne samo da morate da oslobodite prihvatnu memoriju, već morate i da zatvorite datoteku.

Listing 2.6 prikazuje kako biste mogli da napišete ovu funkciju.

```
#include <fcntl.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

char* read_from_file (const char* filename, size_t length)
{
  char* buffer;
  int fd;
  ssize_t bytes_read;
  // Alociraj bafer.
  buffer = (char*) malloc (length);
  if (buffer == NULL)
    return NULL;
  // Otvori datoteku
  fd = open (filename, O_RDONLY);
  if (fd == -1) {
    // Otvaranje nije uspelo. Oslobodi bafer pre povratka
    free (buffer);
    return NULL;
  }

  // Pročitaj podatke
  bytes_read = read (fd, buffer, length);
  if (bytes_read != length) {
    // Čitanje nije uspelo. Oslobodi bafer i zatvori fd
    free (buffer);
    close (fd);
    return NULL;
  }

  // Sve je u redu. Zatvori datoteku i vrati bafer
  close (fd);
  return buffer;
}
```

Listing 2.6 (readfile.c) Oslobađanje resursa tokom nepravilnih stanja

Linux sam oslobađa dodeljenu memoriju, otvorene datoteke i većinu drugih resursa kada program završi sa radom, tako da nije neophodno oslobađati prihvatnu memoriju i zatvarati datoteke pre poziva exit naredbe. Ipak, možda ćete morati da ručno oslobodite druge deljene

resurse, kao što su privremene datoteke i deljena memorija, koji eventualno mogu da nadžive program.

Pisanje i korišćenje biblioteka

Činjenica je da su svi programi povezani sa jednom ili više biblioteka. Svaki program koji koristi C funkciju (kao što je printf ili malloc) će biti povezan sa izvršnom C bibliotekom. Ako vaš program ima grafički korisnički interfejs (Graphical User Interface – GUI), biće povezan sa prozorskim bibliotekama. Ako vaš program koristi bazu podataka, isporučilac baze će vam dati biblioteke pomoću kojih ćete jednostavnije pristupati podacima u bazi.

U svakom od ovih slučajeva morate da se odlučite da li ćete biblioteku povezati statički ili dinamički. Ako izaberete statičko povezivanje, vaši programi će biti veći i teži za nadogradnju, ali verovatno jednostavniji za razvijanje. Ako povezujete dinamički, vaši programi će biti manji, lakši za nadogradnju, ali zato teži za razvijanje. Ovaj odeljak opisuje kako se biblioteke povezuju statički i dinamički, detaljnije ispituje razlike i daje vam smernice kako biste lakše odlučili koji vid povezivanja je za vas bolji.

Arhive

Arhiva (ili statička biblioteka) je prosto skup objektnih datoteka sačuvanih u vidu jedne datoteke. (Arhiva otprilike odgovara `.LIB` datoteci na Windows sistemu.) Kada povezivaču dostavite arhivu, povezivač je pretražuje da bi pronašao objektne datoteke koje su mu potrebne, izvlači ih i povezuje sa vašim programom otprilike kao i kada biste vi direktno obezbedili te objektne datoteke.

Arhivu možete napraviti pomoću `ar` naredbe. Arhivske datoteke tradicionalno koriste `.a` nastavak umesto `.o` nastavka koje koriste obične objektne datoteke. Evo kako biste sjedinili `test1.o` i `test2.o` u pojedinačnu `libtest.a` arhivu:

```
% ar cr libtest.a test1.o test2.o
```

Opcija cr govori ar naredbi da napravi arhivu. Sada se možete povezati sa ovom arhivom koristeći `-ltest` opciju u gcc i g++ prevodiocima.

Kada povezivač naiđe na arhivu u komandnoj liniji, on u njoj traži sve definicije simbola (funkcija ili promenljivih) za koje postoje reference u objektnim datotekama koje je već obradio ali koje još nije definisao. Objektne datoteke koje definišu te simbole se izvlače iz arhive i uključuju u ciljnu izvršnu datoteku. Pošto povezivač pretražuje arhivu

onda kada na nju naiđe u komandnoj liniji, obično ima smisla navesti je na kraju. Na primer, pretpostavimo da `test.c` sadrži kôd iz listinga 2.7, a `app.c` sadrži kôd iz listinga 2.8.

```
int f () {
  return 3;
}
```

Listing 2.7 (test.c) Sadržaj biblioteke

```
int main () {
  return f ();
}
```

Listing 2.8 (app.c) Program koji koristi bibliotečku funkciju

Sada pretpostavimo da je `test.o` sa još nekim objektnim datotekama sjedinjen u `libtest.a` arhivu. Sledeća naredba neće raditi:

```
% gcc -o app -L. -ltest app.o
app.o: In function 'main':
app.o(.text+0x4): undefined reference to 'f'
collect2: ld returned 1 exit status
```

Poruka o grešci ukazuje da, iako `libtest.a` sadrži definiciju funkcije f, povezivač je nije pronašao. To je zato što je povezivač pretražio arhivu kada je prvi put na nju naišao, a do tada on nije naišao ni na jednu referencu na f funkciju.

Sa druge strane, ako koristimo ovu naredbu, ni jedna poruka o grešci se neće pojaviti:

```
% gcc -o app app.o -L. –ltest
```

Razlog je što referenca na `f` funkciju u `app.o` datoteci prouzrokuje da povezivač uključi `test.o` objektnu datoteku iz `libtest.a` arhive.

Deljene biblioteke

Deljena biblioteka (poznata i kao deljeni objekat ili dinamički povezana biblioteka) je slična arhivi jer isto služi grupisanju objektnih datoteka. Ipak, tu postoji i mnogo značajnih razlika. Najbitnija razlika je u tome što kada se deljena biblioteka poveže sa programom, izvršna datoteka na kraju ne sadrži kôd koji je prisutan u deljenoj biblioteci. Ustvari, izvršna datoteka samo sadrži referencu na deljenu biblioteku. Ako je više programa na sistemu povezano sa istom deljenom

bibliotekom, oni će svi imati referencu na tu biblioteku, ali je neće sadržati. Prema tome, biblioteka je "deljena" od strane svih programa koji se povezuju na nju.

Druga važna razlika je ta da deljena biblioteka nije prosto skup objektnih datoteka od kojih povezivač bira one koje su potrebne za rešavanje nedefinisanih referenci. Ustvari, objektne datoteke koje obrazuju deljenu biblioteku su sjedinjene u jednu objektnu datoteku tako da programi koji su povezani sa njom uvek uključuju sav kôd iz te biblioteke, a ne samo one delove koji su potrebni.

Da biste kreirali deljenu biblioteku, morate prevesti objekte koji će obrazovati biblioteku pomoću `–fPIC` opcije, ovako:

```
% gcc –c –fPIC test1.c
```

Opcija `–fPIC` govori prevodiocu da će te test.o koristiti kao deo deljene biblioteke.

Poziciono nezavisan kod

PIC označava poziciono nezavisan kôd (Position-Independent Code). Funkcije u deljenim bibliotekama mogu da zauzimaju različite adrese u različitim programima, tako da kôd u deljenoj biblioteci ne sme da zavisi od adrese (ili pozicije) na koju se smešta. Ovo razmatranje nema mnogo dodira sa vama, kao programerom, sem što morate da zapamtite da koristite –fPIC opciju kada prevodite kôd koji će se koristiti u deljenoj biblioteci. Zatim spajate objektne datoteke u deljenu biblioteku na sledeći način:

```
% gcc -shared -fPIC -o libtest.so test1.o test2.o
```

Opcija `-shared` govori povezivaču da, umesto izvršne datoteke, napravi deljenu biblioteku. Deljene biblioteke koriste `.so` nastavak, koji označava deljeni objekat (shared object – so). Kao i kod statičkih arhiva, ime uvek počinje sa lib da bi se ukazalo na to da je ta datoteka biblioteka.

Povezivanje sa deljenom bibliotekom je isto kao i sa statičkom arhivom. Na primer, sledećom naredbom će se povezivanje izvršiti sa libtest.so bibliotekom ako je ona u tekućem direktorijumu, ili jednom iz podrazumevanih direktorijuma u kojima se nalaze biblioteke na sistemu:

```
% gcc -o app app.o -L. –ltest
```

Pretpostavimo da su i `libtest.a` i `libtest.so` dostupne. Tada povezivač mora da se odluči za samo jednu biblioteku. Povezivač pretražuje svaki direktorijum (prvo one naznačene `–L` opcijom, a zatim

one u podrazumevanim direktorijumima). Kada povezivač naiđe na direktorijum u kojem se nalazi bilo `libtest.a` ili `libtest.so`, on prestaje sa pretragom. Ako je dostupna samo jedna od ove dve datoteke, povezivač će izabrati nju. Inače, povezivač će izabrati deljenu biblioteku, sem ako izričito ne naglasite suprotno. Možete koristiti `–static` opciju ako zahtevate statičku arhivu. Na primer, sledeća naredba će koristiti `libtest.a` arhivu, iako je `libtest.so` deljena biblioteka takođe dostupna:

```
% gcc -static -o app app.o -L. –ltest
```

Naredba ldd prikazuje deljene biblioteke koje su povezane sa izvršnom datotekom. Te biblioteke moraju biti dostupne kada se program pokrene. Obratite pažnju na to da će ldd da izlista dodatnu biblioteku ld-linux.so, koja je deo mehanizma za dinamičko povezivanje GNU/Linux sistema.

Korišćenje LD_LIBRARY_PATH promenljive okruženja

Kada povežete program sa deljenom bibliotekom, povezivač ne stavlja celu putanju do deljene biblioteke u rezultujuću izvršnu datoteku. Ustvari, on stavlja samo ime te deljene biblioteke. Kada se program pokrene, sistem traži deljenu biblioteku i učitava je. Podrazumevano, sistem pretražuje samo `/lib` i `/usr/lib` direktorijume. Ako je deljena biblioteka koja je povezana sa vašim programom instalirana izvan tih direktorijuma, ona neće biti pronađena i sistem će odbiti da pokrene program.

Jedno rešenje za ovaj problem je da koristite `–Wl,-rpath` opciju kada povezujete program. Recimo da ste upotrebili sledeće:

```
% gcc -o app app.o -L. -ltest -Wl,-rpath,/usr/local/lib
```

U tom slučaju, kada se app pokrene, sistem će u `/usr/local/lib` direktorijumu da traži bilo koju od potrebnih deljenih biblioteka.

Drugo rešenje za ovaj problem je da postavite `LD_LIBRARY_PATH` promenljivu okruženja kada pokrećete program. Kao i `PATH` promenljiva okruženja i `LD_LIBRARY_PATH` je niz direktorijuma odvojenih dvotačkom. Na primer, ako u `LD_LIBRARY_PATH` promenljivoj stoji `/usr/local/lib:/opt/lib`, onda će se `/usr/local/lib` i `/opt/lib` pretraživati pre podrazumevanih `/lib` i `/usr/lib` direktorijuma. Trebalo bi da obratite pažnju i na to da ako ste postavili LD_LIBRARY_PATH, povezivač će pretraživati direktorijume koji su tamo navedeni uz one koje su navedeni –L opcijom kada bude pravio izvršnu datoteku.

Standardne biblioteke

Čak i ako niste naveli neku biblioteku kada ste povezivali vaš program, on gotovo sigurno koristi deljenu biblioteku. To je zato što GCC automatski povezuje standardnu C biblioteku `libc` za vas. Matematičke funkcije standardne C biblioteke se ne nalaze u `libc`, već u posebnoj biblioteci `libm` koju morate samostalno da navedete. Na primer, ako prevodite i povezujete program `compute.c` koji koristi trigonometrijske funkcije kao što su `sin` i `cos`, morate da izvršite sledeće:

```
% gcc -o compute compute.c –lm
```

Ako pišete C++ program i povezujete ga c++ ili g++ naredbama, takođe će se standardna C++ biblioteka libstdc++ dodati automatski.

Međusobne zavisnosti biblioteka

Jedna biblioteka će često zavisiti od druge biblioteke. Na primer, veliki broj GNU/Linux sistema poseduje `libtiff` biblioteku koja sadrži funkcije za čitanje i pisanje slika u TIFF formatu. Ova biblioteka koristi redom libjpeg (rutine za JPEG slike) i `libz` (rutine za kompresiju) biblioteke.

Listing 2.9 prikazuje veoma mali program koji koristi libtiff za otvaranje datoteke sa slikom u TIFF formatu.

```
#include <stdio.h>
#include <tiffio.h>

int main (int argc, char** argv)
{
  TIFF* tiff = TIFFOpen (argv[1], "r");
  TIFFClose (tiff);
  return 0;
}
```

Listing 2.9 (tifftest.c) Korišćenje libtiff biblioteke

Snimite ovu datoteku izvornog koda kao `tifftest.c`. Za prevođenje ovog programa i povezivanje sa `libtiff` bibliotekom, navedite `–ltiff` u komandnoj liniji:

```
% gcc -o tifftest tifftest.c –ltiff
```

Podrazumevano, ovim će se uključiti deljena varijanta `libtiff` biblioteke koja se nalazi u `/usr/lib/libtiff.so`. Kako `libtiff` koristi

libjpeg i libz, deljene varijante ovih biblioteka će takođe biti uključene (deljena biblioteka može da pokazuje na druge deljene biblioteke od kojih zavisi). Da biste ovo proverili, primenite ldd naredbu:

```
% ldd tifftest
    libtiff.so.3 => /usr/lib/libtiff.so.3 (0x4001d000)
    libc.so.6 => /lib/libc.so.6 (0x40060000)
    libjpeg.so.62 => /usr/lib/libjpeg.so.62 (0x40155000)
    libz.so.1 => /usr/lib/libz.so.1 (0x40174000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Statičke biblioteke, sa druge strane, ne mogu da pokazuju na druge biblioteke. Ako odlučite da povezivanje izvršite sa statičkom verzijom libtiff biblioteke, navodeći –static opciju u komandnoj liniji, susrešćete se sa nerazrešenim simbolima:

```
% gcc -static -o tifftest tifftest.c -ltiff
/usr/bin/../lib/libtiff.a(tif_jpeg.o):
In function 'TIFFjpeg_error_exit':
tif_jpeg.o(.text+0x2a): undefined reference to 'jpeg_abort'
...
```

Da biste statički povezali ovaj program, morate sami da navedete druge dve biblioteke:

```
% gcc -static -o tifftest tifftest.c -ltiff -ljpeg -lz
```

Ponekad će dve biblioteke biti međusobno zavisne. Drugim rečima, prva arhiva će referencirati simbole definisane u drugoj i obratno. Ova situacija se uglavnom javlja zbog lošeg dizajna, ali se i pored toga ponekad javlja. U tom slučaju, možete više puta da navedete istu biblioteku u komandnoj liniji. Povezivač će iznova pretražiti biblioteku kad god na nju naiđe. Na primer, ovom naredbom će libfoo.a biblioteka više puta da se pretražuje:

```
% gcc -o app app.o -lfoo -lbar –lfoo
```

Tako da, iako libfoo.a referencira simbole u libbar.a biblioteci i obratno, program će se uspešno povezati.

Za i protiv

Sada, kada znate sve o statičkim arhivama i deljenim bibliotekama, verovatno se pitate koju da koristite. Postoji par važnijih stvari koje bi trebalo uzeti u razmatranje.

Jedna od bitnih prednosti deljene biblioteke je što štedi prostor na sistemu gde je program instaliran. Ako instalirate 10 programa i ako

svaki od njih koristi istu deljenu biblioteku, onda korišćenjem deljene biblioteke štedite dosta prostora. Da ste umesto toga koristili statičku arhivu, ona bi bila uključena u svakom od tih programa. Znači, korišćenjem deljenih biblioteka se štedi prostor na disku. Takođe, ako se vaš program skida sa Web-a, skraćuje se vreme za skidanje programa.

Još jedna prednost vezana za deljene biblioteke je što korisnici mogu da nadograđuju biblioteke, bez da nadograđuju svaki program koji zavisi od njih. Na primer, recimo da proizvodite deljenu biblioteku koja upravlja HTTP konekcijama. Mnogi programi mogu da zavise od te biblioteke. Ako pronađete grešku u ovoj biblioteci, nadogradićete biblioteku. Momentalno će svi programi koji koriste ovu biblioteku biti popravljeni. Ne morate da ponovo povezujete svaki program kao što bi to radili u slučaju statičkih arhiva.

Ove prednosti bi mogle da vas navedu na pomisao kako bi uvek trebalo da koristite deljene biblioteke. Ipak, postoje i bitni razlozi koji idu u korist korišćenju statičkih arhiva. Činjenica da nadogradnja deljene biblioteke utiče na sve programe koji je koriste može biti i mana. Na primer, ako razvijate softver od koga se zahteva visoka pouzdanost, možda ćete radije da ga povežete sa statičkom arhivom tako da nadogradnja deljene biblioteke na sistemu neće uticati na rad vašeg programa. (U suprotnom, korisnici bi mogli da nadograde deljenu biblioteku, oštećujući time vaš program, a zatim da pozovu vašu tehničku podršku, kriveći vas!)

Ako ne budete bili u mogućnosti da instalirate vaše biblioteke u `/lib` ili `/usr/lib` direktorijumu, definitivno bi trebalo dvaput da razmislite o upotrebi deljene biblioteke. (Nećete moći da instalirate vaše biblioteke u tim direktorijumima ako očekujete da korisnici instaliraju vaš program bez administratorskih ovlašćenja.) Posebno, `-Wl,-rpath` trik neće raditi ako ne znate gde će se biblioteke na kraju nalaziti. A tražiti od vaših korisnika da postave `LD_LIBRARY_PATH` znači jedan korak više za njih. To pretstavlja znatan dodatni teret, pošto svaki korisnik to mora samostalno da uradi.

Moraćete da prosuđujete o ovim prednostima i manama za svaki program koji planirate da distribuirate.

Dinamičko učitavanje i otpuštanje

Ponekad ćete poželiti da učitate neki kôd u toku izvršavanja programa, a da ga prethodno niste izričito povezali. Uzmimo, na primer, aplikaciju koja podržava zamenljive (plug-in) module, kao što je Web pretraživač. Pretraživač omogućava trećim licima da razvijaju module koji će proširiti njegovu funkcionalnost. Ovi programeri prave deljene

biblioteke i stavljaju ih na podrazumevano mesto. Pretraživač zatim automatski učitava kôd iz tih biblioteka. Ova mogućnost je dostupna na Linux sistemu upotrebom `dlopen` funkcije. Mogli biste da otvorite deljenu biblioteku koja se zove `libtest.so` pozivajući `dlopen` na sledeći način:

```
dlopen ("libtest.so", RTLD_LAZY)
```

(Drugi parametar je oznaka kojom se određuje način povezivanja sa simbolima u deljenoj biblioteci. Ako želite više informacija, možete konsultovati stranice uputstva za `dlopen` funkciju, ali `RTLD_LAZY` je najčešće ono što želite.) Da biste koristili dinamičko učitavanje, uključite `<dlfcn.h>` datoteku zaglavlja i povežite sa `–ldl` opcijom da biste učitali `libdl` biblioteku. Povratna vrednost ove funkcije je `void *` koja se koristi za rukovanje deljenom bibliotekom. Ovu vrednost možete proslediti `dlsym` funkciji za dobijanje adrese funkcije koja je učitana sa deljenom bibliotekom. Na primer, ako `libtest.so` definiše funkciju koja se zove `my_function`, mogli biste da je pozovete na sledeći način:

```
void* handle = dlopen ("libtest.so", RTLD_LAZY);
void (*test)() = dlsym (handle, "my_function");
(*test)();
dlclose (handle);
```

Sistemski poziv `dlsym` se može upotrebiti i za dobijanje pokazivača na statičku promenljivu iz deljene biblioteke. I `dlsym` i `dlopen` vraćaju NULL ako se ne izvrše uspešno. U tom slučaju možete da pozovete `dlerror` funkciju (bez parametara) kojom se obezbeđuje poruka koja, na čoveku razumljiv način, opisuje problem. Funkcija `dlclose` otpušta deljenu biblioteku. Praktično, `dlopen` učitava biblioteku samo ako već nije učitana. Ako je biblioteka već učitana, `dlopen` samo uvećava broj referenci na biblioteku za jedan. Slično tome, `dlclose` smanjuje broj referenci, a otpušta biblioteku samo kada broj referenci padne na nulu.

Ako kôd za vašu deljenu biblioteku pišete u C++ jeziku, verovatno ćete želeti da funkcije i promenljive, kojima planirate da pristupate, deklarišete sa `extern "C"`. Na primer, ako je C++ funkcija `my_function` u deljenoj biblioteci i želite da joj pristupate pomoću `dlsym`, trebalo bi ovako da je deklarišete: `extern "C" void foo ();` što sprečava C++ prevodilac da izmeni ime funkcije. Inače, on bi promenio ime funkcije `foo` u neko drugo, čudno ime koje bi sadržalo dodatne kodirane informacije o funkciji. C prevodilac neće menjati imena, upotrebiće svako ime koje budete dali vašoj funkciji ili promenljivoj.

3. Procesi

Deo programa koji se nalazi u stanju izvršavanja naziva se proces. Kada zadate komandu u komandnom interpreteru (*shell*), odgovarajući program se izvršava u novom procesu, a kada se proces završi, proces u kome se izvršava komandni interpreter nastavlja sa radom. Kada otvorite program u grafičkom okruženju, na ekranu ćete dobiti novi prozor, tj proces u kome se izvršava program koji ste pokrenuli. Ukoliko imate dva prozora prikazana na vašem monitoru u kojima se izvršava ista aplikacija, onda je reč o izvršavanju istog programa dva puta u dva različita procesa.

Napredni programeri često koriste višestruko kooperativne procese u jednoj aplikaciji, da bi tako omogućili da aplikacija radi više od jedne stvari istovremeno, da bi povećali snagu aplikacije i da bi koristili već postojeće programe.

Većina funkcija koje služe za rad sa procesima, opisanih u ovom poglavlju, slične su onima iz drugih UNIX sistema. Većina njih deklarisana je u datoteci koja predstavlja zaglavlje `<unistd.h>`, ali je poželjno za svaku od funkcija proveriti uputstvo.

PID i pregledanje aktivnih procesa

Čak i dok sedite za računarom, ne radeći ništa, postoje procesi koji su u stanju izvršavanja. Svaki izvršni program koristi jedan ili više procesa. Počnimo tako, što ćemo pregledati procese koji se već nalaze u vašem računaru.

Identifikator procesa (PID) i procesa roditelja (PPID)

Svaki proces na Linux sistemima identifikuje se pomoću jedinstvenog identifikatora koji se označava sa PID (*Proces Identifier*). Identifikatori procesa su šesnaestobitni brojevi, koje Linux sekvencijalno dodeljuje novonastalim procesima.

Svaki proces, osim posebnog INIT procesa, ima svog "roditelja" čiji je identifikator PPID (*Parrent PID*). Zbog toga, za procese u Linux sistemu se može reći da su organizovani kao stablo, gde proces init predstavlja koren stabla.

Kada se radi sa identifikatorima procesa u programskom jeziku C ili C++, uvek treba koristiti tip `pid_t`, definisan u `<sys/types.h>`. Program može dobiti identifikator procesa za proces u kome se on izvršava pomoću sistemskog poziva `getpid()` i može dobiti identifikator roditeljskog procesa pomoću sistemskog poziva `getppid()`. Na primer, program dat u listingu 3.1 (`print-pid.c`) prikazuje identifikator procesa u kome se program izvršava i identifikator roditeljskog procesa.

Otkucajte u nekom od editora koji vam je na raspolaganju (vi, joe, jed ili emacs) tekst programa `print-pid.c`.

```
#include <stdio.h>
#include <unistd.h>

int main ()
{
  printf ("Identifikator procesa je %d\n", (int) getpid());
  printf ("Identifikator roditelja je %d\n", (int) getppid());
  return 0;
}
```

Listing 3.1 (print-pid.c) Pokazivanje identifikatora procesa

Prevedite zatim program komandom:

```
$ gcc –o print-pid print-pid.c
```

A zatim ga pokrenite nekoliko puta:

```
$ ./print-pid
Identifikator procesa je 2031
Identifikator roditeljskog procesa je 1478

$ ./print-pid
Identifikator procesa je 2049
Identifikator roditeljskog procesa je 1478
```

Primetite da ukoliko pozovete program nekoliko puta, dobićete različite vrednosti za identifikator procesa, zato što je svaki novi poziv ujedno novi proces. Međutim, ukoliko pravite pozive svaki put iz istog komandnog interpretera, identifikator roditeljskog procesa (odnosno identifikator proces u kom se izvršava taj komandni interpreter) je isti.

Pregled aktivnih procesa – komanda ps

Prijavite se na prvi terminal (tj. konzolu) Linux operativnog sistema (pritisnite <Ctrl+Alt+F1> pa unesite korisničko ime i lozinku) ili pokrenite program Terminal u grafičkom okruženju.

Naredba `ps` prikazuje procese koji su u stanju izvršavanja na vašem sistemu. GNU/Linux verzija za naredbu `ps` ima dosta opcija, zato što ona pokušava da bude kompatibilna sa verzijom naredbe `ps`, koja se nalazi na drugim UNIX sistemima. Ove opcije određuju koji će procesi biti prikazani i koje će se informacije prikazati o svakom od tih procesa.

Standardno, pozivanjem komande `ps`, prikazuju se procesi koje kontrolišu terminal ili terminalski prozor (t*erminal window*) iz koga je komanda `ps` pozvana. Na primer:

```
$ ps
  PID TTY          TIME CMD
 6506 pts/0    00:00:00 bash
 6523 pts/0    00:00:00 ps
```

Ovo pozivanje komande ps prikazuje dva procesa. Prvi, `bash`, je komandni interpreter (*shell*) koji je u stanju izvršavanja na ovom terminalu. Drugi je sam poziv komande `ps`. Prva kolona, sa natpisom PID, prikazuje identifikatore svih prikazanih procesa.

Da biste dobili detaljniji uvid o tome koji su procesi u stanju izvršavanja na vašem GNU/Lunux sistemu pozovite sledeće:

```
$ ps -e -o pid,ppid,command
  PID  PPID COMMAND
    1     0 /sbin/init
    2     0 [kthreadd]
    3     2 [migration/0]
 ...
 6503     1 gnome-terminal
 6505  6503 gnome-pty-helper
 6506  6503 bash
 6544  6506 ps -e -o pid,ppid,command
```

Opcija `–e` nalaže naredbi ps da prikaže sve procese koji su trenutno u stanju izvršavanja u sistemu. Pomoću `–o` opcije za naredbu `ps`, možete odrediti koje infomacije o procesima želite na izlazu, kao listu parametara odvojenu zarezima. U ovom slučaju, opcija `–o pid,ppid,command`, govori naredbi `ps` koje informacije da prikaže o svakom procesu, u ovom slučaju identifikator procesa, identifikator roditeljskog procesa i naredbu koja se izvršava u ovom procesu.

Primetite da roditeljski identifikator procesa naredbe `ps` (PPID= 6506) predstavlja identifikator procesa `bash`, odnosno komandnog interpretera iz kog smo pozvali naredbu `ps`. Primetite takođe da je roditelj procesa `bash` proces `gnome-terminal` (terminalski prozor u kome se izvršava komandni interpreter `bash`).

Možete koristiti i dodatne opcije za komandu `ps` pomoću kojih ćete dobiti tri različita formata tekućeg listinga.

- –f (kompletan listing)
- –l (dugačak listing)

- –j (listing poslova)

Raspoređivanje procesa i nice vrednosti

Linux kernel dodeljuje procesor procesima na osnovu prioriteta i vremena čekanja procesa na izvršenje. Procesor se dodeljuje svakom procesu na određeno vreme (time-slice), koji se u tom intervalu izvršava. Ukoliko korisnik pokrene neki interaktivni program (kao što je `vi`), imaće viši prioritet u odnosu na ove procese.

Na prioritet procesa utiče tzv. "*nice*" vrednost, koja se kreće u opsegu od 0 do 39 (podrazumevano 20). Ova vrednost može da se postavi prilikom pokretanja procesa i da se naknadno promeni. Pri tome, korisnik `root` (administrator sistema) može da smanji *nice* vrednost bilo kog procesa, tj. da mu povećava prioritet . Korisnik `root` takođe može da smanji prioritet bilo kog procesa. Obični neprivilegovani korisnici mogu samo da povećaju *nice* vrednosti samo svojih procesa, tj. da im smanje prioritet - obični korisnici ne mogu da povećavaju prioritete svojih procesa i da smanjuju prioritete tuđih procesa.

Komande nice i renice

Nice vrednost se prilikom pokretanja određuje komandom `nice`:

```
nice command &
```

Komanda `nice` bez dodatnih argumentata (osim komande koju pokreće) smanjuje prioritet novog procesa za 10.

Komandom `renice` vlasnik procesa ili `root` mogu promeniti nice vrednost aktivnog, postojećeg procesa. Sintaksa komande je:

```
renice priority [[-p] PID ...] [-u UID ...]
```

gde je:

- `priority` nova nice vrednost procesa
- `-p PID` PID procesa kome treba promeniti nice vrednost
- `-u UID` ID korisnika (menja se nice svih procesa koje je inicirao dati korisnik).

Na primer, komanda

```
renice 30 1500
```

će smanjiti prioritet procesa čiji je PID 1500 (pošto se nice vrednost povećava sa 20 na 30).

Kreiranje procesa

Postoje dve uobičajene tehnike za kreiranje novog procesa. Prva je relativno jednostavna ali treba da bude korišćena umereno, zato što nije efikasna i kod nje postoji pozamašna doza sigurnosnog rizika. Druga tehnika je složenija, ali pruža veću fleksibilnost, brzinu i sigurnost.

Korišćenje funkcije system

Funkcija system u standardnoj C biblioteci omogućuje, da se na lak način izvrši naredba unutar programa, na isti način kao da je naredba direktno uneta u shell. U stvari, funkcija sistem kreira podproces koristeći standardni Bourne shell (`/bin/sh`) i predaje naredbu tom podprocesu na izvršavanje. Na sledećem primeru prikazan je program koji poziva naredbu ls da bi prikazao sadržaj root direktorijuma, a efekat je isti, kao da ste otkucali komandu `ls –l` u shell-u.

Otkucajte tekst programa datog u listingu 3.2 i snimite ga u datoteku `system.c`.

```
#include <stdlib.h>
int main ()
{
  int return_value;
  return_value = system ("ls –l /");
  return return_value;
}
```

Listing 3.2 (system.c) Upotreba funkcije system

Prevedite i pokrenite program:

```
$ gcc –o system system.c
$ ./system
```

Funkcija `system` vraća izlazni status shell-naredbe. Ako se sam shell ne može izvršiti, funkcija system vraća 127; a ukoliko se dogodi neka druga greška, funkcija system vraća -1.

Pošto funkcija system koristi shell da bi pozvala vašu naredbu, ona je izložena ograničenjima i sigurnosnim propustima sistemskog shell-a. Ne možete se osloniti na raspoloživost bilo koje verzije Bourn shell-a. Na mnogim UNIX sistemima `/bin/sh` je simbolički link nekog drugog shella. Na primer, na većini GNU/Linux sistema /bin/sh ukazuje na bash (Bourne-Again Shell), a različite GNU/Linux distribucije koriste različite verzije za bash. Pozivajući program koji ima root privilegiju, funkcijom

system, možete imati različite rezultate na različitim GNU/Linux sistemima. Zbog toga, preporučuje se korišćenje `fork` i `exec` metoda za kreiranje procesa.

Korišćenje fork i exec metoda

Windows API sadrži "spawn" familiju funkcija. Ove funkcije uzimaju za argument ime programa koji treba da bude izvršen i stvaraju novi primerak procesa tog programa. Linux ne poseduje jednu funkciju koja radi sve to u jednom koraku. Umesto toga, Linux ima jednu funkciju `fork`, koja stvara dete-proces, koji je istovetna kopija svog roditeljskog procesa. Linux ima poseban skup funkcija, `exec` familiju funkcija, koje izazivaju da određeni proces prestaje da bude instanca jednog programa i umesto toga postaje instanca drugog programa. Da bi se izvršio novi proces, prvo se koristi `fork` da se napravi kopija tekućeg procesa. Potom se koristi `exec` da bi se jedan od ovih procesa transformisao u instancu programa, koga želite da izvršite.

Upotreba sistemskog poziva fork

Kada program pozove funkciju `fork`, stvara se duplikat procesa, koji se naziva dete-proces. Roditeljski proces, nastavlja sa izvršavanjem programa od mesta, gde je funkcija `fork` bila pozvana. Dete-proces takođe, izvršava program od istog mesta.

Po čemu su ta dva procesa različita? Kao prvo, dete-proces je novi proces i stoga ima i novi identifikator procesa, koji je različit od roditeljskog. Jedan od načina da program razlikuje da li se radi o roditeljskom procesu ili o dete-procesu je da pozove funkciju getpid. Međutim, `fork` funkcija prosleđuje različite povratne vrednosti za roditeljski proces i dete-proces. Jedan proces "ulazi" u `fork` poziv a dva procesa "izlaze", sa različitim povratnim vrednostima. Vrednost koja je vraćena za roditeljski proces, predstavlja vrednost identifikatora za dete-proces. Vrednost koja je vraćena detetu-procesu je nula. Zbog toga što ni jedan proces nikad ne može imati vrednost nula kao svoj identifikator procesa, to olakšava programu da odredi ko je roditelj a ko dete.

Sledeći primer (listing 3.2), koristi funkciju `fork` da bi napravio duplikat procesa. Primetite da se prvi blok, `if` deo, izvršava, samo u roditeljskom procesu, dok se `else` klauzula izvršava u dete-procesu.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
```

```
int main ()
{
  pid_t child_pid;
  child_pid = fork ();
  if (child_pid !=0) // ovo je proces roditelj
  {
    printf ("Ovo je proces roditelj: %d\n", (int) getpid ());
    printf ("Identifikator deteta: %d\n", (int) child_pid);
  }
  else // ovo je proces dete
    printf ("Ovo je dete proces: %d\n", (int) getpid ());
  return 0;
}
```

Listing 3.3 (fork.c) Korišćenje sistemskog poziva fork

Prevedite i pokrenite program:

```
$ gcc –o fork fork.c
$ ./fork
Ovo je proces roditelj: 1050
Identifikator deteta: 1051
Ovo je dete proces: 1051
```

Primetite: proces roditelj na osnovu povratne vrednosti iz sistemskog poziva fork, zna da je to PID novokreiranog procesa-deteta.

Upotreba exec familije funkcija

Funkcije familije exec zamenjuju program, koji se izvršava u tekućem procesu, sa drugim programom. Kada program pozove exec funkciju, proces momentalno prestaje da izvršava taj program i počinje sa izvršavanjem novog programa iz početka, pod pretpostavkom da poziv funkcije exec neće izazvati grešku.

Unutar exec familije funkcija, postoje funkcije koje su različite po svojim sposobnostima i po tome kako se pozivaju.

- Funkcije koje sadrže slovo p u svom imenu (execvp i execlp) prihvataju imena programa i vrše pretragu programa po imenu, sa tekuće izvršne putanje (PATH); funkcijama koje ne sadrže slovo p, mora se proslediti kompletna putanja programa koji će biti izvršen.

- Funkcije koje sadrže slovo v u svom imenu (`execv`, `execvp` i `execve`) prihvataju listu argumenata, za novi program, kao NULL-završeni niz pokazivača na nizove (strings).
- Funkcije koje sadrže slovo l (`execl`, `execlp` i `execle`) prihvataju listu argumenata koristeći `varargs` mehanizam programskog jezika C.
- Funkcije koje sadrže slovo e u svom imenu (`execve` i `execle`) prihvataju dodatni argument, a to je niz promenjivih u okruženju. Argument bi trebalo da bude NULL-završeni niz pokazivača na nizove karaktera. Svaki niz karaktera ima oblik "`VARIABLE=value`".

Zbog toga što funkcija exec zamenjuje pozvani program sa drugim, ona nikad ne vraća izvršenje programa na mesto iza poziva `exec`, osim ako se dogodi greška.

Lista argumenata koja se prosleđuje programu analogna je sa argumentima komandne linije koje definišete u programu, kada je pokrećete iz shella. Oni su raspoloživi preko argc i argv parametara u main funkciji. Zapamtite da, kada je program pokrenut iz shella, shell postavlja prvi element sa liste argumenata, `argv[0]`, kao ime tog programa, drugi element liste argumenata, `argv[1]`, kao prvi komandno-linijski argument i tako redom. Kada u vašim programima koristite funkciju `exec`, takođe bi trebalo da prosledite ime funkcije kao prvi element iz liste argumenata.

Upotreba funkcija fork i exec zajedno

Zajednička šema za pokretanje potprograma unutar programa radi tako, što prvo fork deluje na proces a potom exec na potprogram. Ovo omogućuje da pokrenuti program nastavi sa izvršavanjem u roditeljskom procesu, dok je pokrenuti program zamenjen potprogramom u dete-procesu. Otkucajte tekst programa `fork-exec.c` (listing 3.4), uočite sistemske pozive i pokušajte da objasnite način funkcionisanja celog programa. Program pravi dete-proces u kom se izvršava novi program. Promenljiva `program` je ime programa koji treba izvršiti; putanja za ovaj program će biti tražena. Promenljiva `arg_list` je lista sačinjena od nizova karaktera završiti NULL karakterom. Ta lista će biti prosleđena kao programska lista argumenata. Vrednost koja se vraća je identifikator procesa za proces koji se pojavljuje.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
```

```
int spawn (char* program, char** arg_list)
{
  pid_t child_pid;
  child_pid = fork (); // napravi duplikat procesa

  if (child_pid !=0) // ovo je roditeljski proces
    return child_pid;

  else
  { // ovo je dete proces
    execvp (program, arg_list); // pronađi program i izvrši ga
    // funkcija execvp vraća nešto samo ako se dogodi greška
    fprintf (stderr, "Greška u funkciji execvp\n");
    abort();
  }
}

int main ()
{
  // Lista argumenata koji se prosleđuju programu ls
  char* arg_list[] = {
    "ls", // argv[0], ime programa
    "-l",
     "/",
    NULL  // Lista argumenata mora da se završi sa NULL
  };

  /* Nastali dete proces izvršava ls komandu.
  Ignoriše vraćeni Identifikator procesa za dete porces. */
  spawn ("ls", arg_list)
  printf ("Glavni program \n ");

  return 0;
}
```

Listing 3.4 (fork-exec.c) Upotreba funkcija fork i exec zajedno

Ovaj program, kao i program dat u listingu 3.2 (`system.c`), lista sadržaj root direktorijuma koristeći naredbu `ls`.

Međutim, za razliku od programa `system`, koji poziva naredbu `ls` preko shell-a, program `fork-exec` poziva naredbu `ls` direktno, prosleđujući joj argumente komandne linije (`–l` i `/`).

Ukoliko prevedete i pokrenete program dobićete isti efekat kao i da ste zadali komandu `ls -l` u komandnom interpreteru (listanje sadržaja root direktorijuma aktivnog UNIX stabla).

```
$ gcc –o fork-exec fork-exec.c
$ ./fork-exec
```

4. Prekidanje, čekanje i zombi procesi

Normalno, proces se prekida na jedan od dva načina:

- program koji se izvršava poziva funkciju exit, ili
- povratkom iz programske funkcije main.

Svaki proces ima izlazni kôd: broj koji proces vraća svom roditeljskom procesu. Izlazni kôd je argument koji se prosleđuje funkciji exit, ili je to povratna vrednost iz funkcije main.

Prekidanje procesa

Proces se takođe može ukinuti na neuobičajan način, u zavisnosti od signala. Na primer, signali SIGBUS, SIGSEGV i SIGFPE izazivaju prekidanje procesa. Drugi signali se koriste da ukinu proces direktno. Signal SIGINT se šalje procesu, kada korisnik pokušava da ga ukine pomoću komande CTRL+C. Naredba `kill` podrazumevano šalje signal SIGTERM. Standardna obrada za oba ova signala jeste ukidanje procesa. Pozivanjem funkcije `abort`, proces šalje sebi signal SIGABRT, koji ukida proces i pravi core datoteku. Najmoćniji signal za ukidanje procesa je SIGKILL, koji trenutno ukida proces i ne može biti blokiran ili obrađen drugim programom.

Komanda kill

Bilo koji od ovih signala se može poslati korišćenjem naredbe kill određivanjem posebnog flaga u komandnoj liniji. Na primer, da bi se okončao problematični proces, tako što će biti poslat signal SIGKILL, treba pozvati jednu od sledeće dve navedene komande (argument pid predstavlja identifikator procesa):

```
$ kill -KILL pid
$ kill -s 9 pid
```

Da bi se poslao signal iz programa, treba koristiti funkciju `kill`. Prvi parametar je PID ciljanog procesa. Drugi parametar je broj signala; koristite SIGTERM da bi simulirali standardno ponašanje naredbe `kill`. Na primer, kada `child_pid` sadrži identifikator procesa za dete_proces, možete koristiti `kill` funkciju da prekinete dete-proces iz roditeljskog procesa na sledeći način:

```
kill (child_pid, SIGTERM);
```

Uključite `<sys/types.h>` i `<signal.h>` zaglavlja datoteke ako koristite kill funkciju.

Izlazni status

Po konvenciji, izlazni kôd se koristi da pruži informaciju da li je program dobro izvršen. Nula kao izlazni kôd ukazuje da je izvršavanje bilo dobro, dok kôd koji je različit od nule ukazuje na grešku. U ovom drugom slučaju, vraćena vrednost nam može dati neke podatke vezane za prirodu nastale greške. Dobra ideja je držati se ove konvencije u vašim programima zato što drugi delovi GNU/Linux sistema podrazumevaju takvo ponašanje. Na primer, shell prihvata tu konvenciju kada povežete više programa sa operatorima && (logičko I) i || (logičko ILI) operatorima. Stoga, trebalo bi eksplicitno da vratite nulu iz vaše main funkcije, osim ukoliko se dogodi greška.

Kod većine shell-ova, moguće je dobiti izlazni kôd većine nedavno izvršenih programa, pomoću specijalne $? promenjive. Ovde je primer u kojem je naredba ls pozvana dva puta i njen izlazni kôd je prikazan nakon svakog poziva. U prvom slučaju, naredba ls se izvršava dobro i vraća nulu kao izlazni kôd. U drugom slučaju, ls nailazi na grešku (zato što je ime datoteke navedene u komandnoj liniji nepostojeće) i zbog toga vraća vrednost različitu od nule.

```
$ ls /
bin   coda etc   lib        misc nfs   proc sbin usr
boot  dev  home  lost+found mnt  opt   root tmp  var

$ echo $?
0

$ ls bougusfile
ls: bogusfile: No such file or directory

$echo $?
1
```

Primetite da bez obzira što je int tip parametra funkcije exit i bez obzira što funkcija main vraća int vrednost, Linux ne koristi punih 32 bita vraćenog statusa. Zapravo, trebalo bi da koristite samo kodove između nule i 127. Izlazni kodovi iznad 128 imaju specijalno značenje – kada je proces ukinut preko signala, njegov kôd je 128 plus broj signala.

Čekanje na završetak procesa

Ako ste ukucali i pokrenuli primer za `fork` i `exec`, možda ste primetili da se izlaz iz ls programa često pojavljuje nakon što je glavni program već završio sa radom. To se dešava zato što je dete-proces, u kojem se ls izvršava, nezavisno raspoređen u odnosu na roditeljski proces. Zbog toga što je Linux multi-tasking operativni sistem, oba procesa se

izvršavaju simultano, a ne može se predvideti da li će `ls` program izvršavati pre ili posle roditeljskog procesa.

Ipak, u nekim situacijama, poželjno je da roditeljski proces čeka jedan ili više dete-procesa da završe. Ovo se može izvesti pomoću `wait` familije sistemskih poziva. Ove funkcije omogućavaju da sačekate da drugi proces završi sa izvršavanjem i da roditeljski proces dobije informacije o ukidanju dete-procesa. Postoje četiri različita sistemska poziva u `wait` familiji, a možete odabrati da li ćete dobiti malo ili puno informacija o procesu koji je okončan, a možete birati da li vodite računa o tome koji je dete-proces okončan.

Sistemski poziv wait

Najjednostavnija takva funkcija se jednostavno naziva wait, koja blokira proces koji je poziva dok se jedno od njegovih dete-procesa ne završi (ili se dogodi greška). Funkcija `wait` vraća celobrojnu vrednost iz koje se može zaključiti kako je dete-proces okončao (deklaracija funkcije `wait` je `pid_t wait(int *stanje_deteta)`, a PID je celobrojna vrednost).

Dodatno, mogu se koristiti makroi WIFEXITED i WEXITSTATUS. Makro WEXITSTATUS vraća izlazni kôd za dete-proces. Makro WIFEXITED vraća nenulu ako se dete proces završio normalno (preko funkcije `exit` ili povratkom iz funkcije `main()`, tj. glavnog programa) ili je prekinut nekim signalom. U slučaju da je utvrđeno da je izvršenje deteta prekinuto signalom, makro WTERMSIG određuje broj signala pomoću kojeg je dete proces okončan.

U listingu 4.1 imamo sličnu `main()` funkciju koju smo koristili u primeru sa `fork` i `exec` (listing 3.4, listanje sadržaja root direktorijuma). Ovog puta, roditeljski proces poziva wait funkciju da bi sačekao dete-proces, u kojem se izvršava naredba ls, da završi sa radom.

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <unistd.h>

int main ()
{
  int child_status;
```

```
  // Lista argumenata koji se prosleđuju programu ls
  char* arg_list[] = {
    "ls", // argv[0], ime programa
    "-l",
    "/",
    NULL  // Lista argumenata mora da se završi sa NULL
  };

  /* Stvara se dete proces koji izvršava naredbu "ls".
  Ignoriše se vraćeni identifikator za dete-proces */
  spawn ("ls", arg_list);

  wait (&child_status); // čeka se da se dete-proces izvrši

  if (WIFEXITED (child_status))
    printf ("Dete-proces okoncan normalno, sa izlaznom kodom",
             " %d \n", WIFEXITED (child_status));
  else
    printf ("dete-proces nije okoncan normalno \n");

  return 0;
}
```

Listing 4.1 (wait.c) Primer sistemskog poziva wait

Prevedete i pokrenete program:

```
$ gcc –o wait wait.c
$ ./wait
```

Uporedite programe `wait.c` i `fork-exec.c`. U Linuxu je raspoloživo nekoliko sličnih sistemskih poziva, koji su fleksibilniji ili pružaju više informacija o završetku dete-procesa. Funkcija `waitpid` može se koristiti kako bi se sačekalo da se zadati dete-proces završi, umesto bilo kog drugog dete-procesa. Funkcija `wait3` daje statističke podatke o upotrebi procesora za dete-proces, a funkcija `wait4` dozvoljava definisanje dodatnih opcija o tome koji proces treba da se čeka.

Zombi procesi

Ako se dete proces završi dok je roditeljski proces u funkciji `wait`, dete proces nestaje i njegov izlazni status se vraća roditelju preko poziva `wait`. Šta se dešava kada se dete-proces završi a roditeljski proces nije pozvao `wait`? Da li jednostavno nestaje? Ne, zato što, bi onda informacije o njegovom ukidanju (kao što su da li je okončan normalno i

koji je njegov izlazni status) bile izgubljene. Umesto toga, kada se dete-proces okonča, ono postaje zombi proces. Zombi proces je proces koji se završio aktivnosti, ali koji još nije obrisan. Na roditeljskom procesu je da obriše svoje zombi decu-procese. To obavlja funkcija `wait`, tako da tada nije potrebno pratiti da li se vaš dete-proces još uvek izvršava.

Pretpostavimo, na primer, da je program obavio `fork` i kreirao dete proces, da je izvršio neka druga računanja, a zatim poziva `wait`. Ako dete proces još uvek nije završilo na ovom mestu, roditeljski proces će biti blokiran u `wait` pozivu sve dok se dete-proces ne završi. Ako se dete-proces završi pre nego što roditeljski proces pozove `wait`, dete proces postaje zombi proces. Kada roditeljski proces pozove `wait`, počinje ukidanje zombi dete-procesa, dete-proces se briše, a `wait` poziv završava trenutno. Šta se dešava ako roditeljski proces ne ukloni decu-procese? Oni ostaju u sistemu kao zombi procesi.

Program u sledećem sistemskim pozivom `fork` kreira dete-proces koji se momentalno ukida. Roditelj spava jedan minut i ne briše dete proces.

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main ()
{
  pid_t child_pid;
  child_pid = fork (); // stvara se dete proces
  if (child_pid >0)
    sleep (60); // roditeljski proces spava jedan minut
  else exit (0); // ovo je dete proces - izlaz je trenutan
return 0;
}
```

Listing 4.2 (make-zombie.c) Kreiranje zombi procesa

Prevedite i pokrenite ovaj program.

```
$ gcc –o make-zombie make-zombie.c
$ ./make-zombie
```

Dok se program `make-zombie` izvršava otvorite novi terminal prozor i izlistajte procese na sistemu sledećom komandom:

```
$ ps –e –o pid,ppid,stat,cmd
```

Komanda prikazuje listu identifikatora procesa (parametar `pid`), identifikator procesa roditelja (parametar `ppid`), status procesa

(parametar `stat`) i komandnu liniju koja je inicirala nastanak procesa (parametar `cmd`). Primetite da se kao dodatak roditeljskom `make-zombie` procesu, pojavljuje još jedan `make-zombie` proces. To je dete-proces, a identifikator njegovog roditelja je PID glavnog `make-zombie` procesa. Dete-proces je označen kao `<defunct>` i njegov statusni kôd je `Z`, kao zombi.

Šta se dešava kada proces roditelj završi? Da li zombi proces ostaje tu gde jeste? Ne. Pokušajte da pokrenete komandu `ps` ponovo (nakon minut vremena) i primetite da su oba `make-zombie` procesa nestala. Kada proces roditelj završi sa radom, njegovu decu nasleđuje poseban proces `init` (proces čiji je PID=1; to je prvi proces koji se pokreće prilikom podizanja Linux operativnog sistema). Proces `init` automatski uklanja bilo koji zombi dete-proces koji nasledi.

Vežbe

[1] Otkucajte naredbu `ps –af`. Šta se dobilo kao izlaz?

Otkucajte naredbu `ps –ax`. Koji su procesi sada u pitanju?

[2] Napravite kratak program za listanje sistemskih procesa, koji daje isti izlaz kao i konzolna naredba `ps –ax`.

Uputstvo: u `main()` funkciji pozvati funkciju `system("ps –ax")`.

[3] Kakva se razlika postiže ako kao argument funkcije system navedemo narebu `"ps –ax &"`? Da li sada moramo čekati da se cela naredba ps završi pre nego što se iz poziva funkcije `ps` vratimo u glavni program?

Uputstvo: Ovo se najbolje vidi ako posle poziva naredbe system napravimo kratak ispis nečega.

[4] Unesite u editoru program fork1.c prikazan u listingu 4.3.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
  pid_t pid;
  char *poruka;
  int n;
  print("Program fork1 pocinje…\n");
  pid = fork();
```

```
  switch(pid) {
  case -1:
    perror("fork nije uspeo!"); exit(1);
  case 0:
    poruka = "Ovaj proces je dete proces";
    n = 5; break;
  default:
    poruka = "Ovaj proces je proces roditelj";
    n = 3; break;
  }
  for(; n>0; n--) {
    puts(poruka);
    sleep(1);
  }
exit(0);
}
```

Listing 4.3 (fork1.c)

Prevedite i pokrenite program. Kakav je izlaz i zašto? Stavite sada naredbu `sleep(1)` pod komentar (`//sleep(1);`)

Da li se promenio redosled ispisa poruka i zašto? Šta se dešava ako stavimo `sleep(2)`?

U programu `fork1.c`, u naredbi `switch` zamenite vrednosti promenljive `n` (tamo gde je bilo 3 stavimo 5 i obrnuto). Da li se pojavio zombi proces? Proverite to naredbom `ps –al` - imate 2 sekunde vremena da vidite zombija :)

[5] U prethodnom primeru, roditelj proces nije čekao da dobije statusne informacije o kraju rada deteta procesa. Čekanje i dobijanje tih informacija postiže se naredbom wait. Za dobijanje statusnih informacija mogu se koristiti makroi WIFEXITED i WEXITSTATUS. Kao što je već rečeno, WIFEXITED vraća nenulu ako se dete proces završio normalno, dok WEXITSTATUS vraća izlazni kôd za dete-proces.

Izmenite prethodni program u program `wait1.c` (listing 4.4).

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
```

```
#include <sys/wait.h>

int main()
{
  pid_t pid;
  char *poruka;
  int n, exit_code;
  print("Program wait1 pocinje…\n");
  pid = fork();
  switch(pid) {
  case -1:
    perror("fork nije uspeo!"); exit(1);
  case 0:
    poruka = "Ovaj proces je proces dete";
    n = 5; exit_code = 37; break;
  default:
    poruka = "Ovaj proces je proces roditelj";
    n = 3; exit_code = 0; break;
  }

  for(; n>0; n--) {
    puts(poruka);
    sleep(1);
  }
  if(pid!=0)
  {
    int stat_val;
    pid_t child_pid;
    child_pid = wait(&stat_val);
    printf("Dete proces je zavrsio: PID = %d\n", child_pid);
    if(WIFEXITED(stat_val))
      printf("Dete proces je zavrsio rad sa kodom %d\n",
             WEXITSTATUS(stat_val));
    else
      printf("Dete proces je zavrsio sa radom neregularno\n");
  }
  exit(exit_code);
}
```

Listing 4.4 (wait1.c)

5. Niti

Niti, kao i procesi, su mehanizam koji omogućava da program radi više od jedne stvari istovremeno. Kao i kod procesa, niti se izvršavaju istovremeno. Linux kernel ih raspoređuje asinhrono, prekidajući svaku nit sa vremena na vreme, da bi ostale imale priliku da se izvrše.

Konceptualno, nit postoji unutar procesa. Niti su finije izvršne jedinice od procesa. Kada pozivate program, Lunux kreira novi proces i u tom procesu se stvara jedna nit koja izvršava program sekvencijalno. Ta nit može da stvori dodatne niti, a sve ove niti izvršavaju isti program u istom procesu, ali svaka nit može da izvršava različit deo programa u ma kom trenutku.

Videli smo kako program može da izvrši funkciju `fork` i napravi dete proces. Dete proces inicijalno izvršava svoj roditeljski proces, sa njegovom roditeljskom virtualnom memorijom, deskriptorima datoteka, i tako dalje. Dete proces može da modifikuje memoriju, zatvori deskriptore datoteka, i tome slično, bez uticaja na roditeljski proces, a važi i obrnuto. Kada program kreira novu nit ništa nije kopirano. Stvarajuća i stvorena nit dele isti memorijski prostor, deskriptore datoteka, i druge sistemske resurse kao i original. Ako jedna nit promeni vrednost promenjive, druga nit će potom videti izmenjenu vrednost. Slično tome, ako jedna nit zatvori deskriptor, druge niti neće moći da pročitaju ili da upišu u taj deskriptor datoteke. Zbog toga što proces i sve njegove niti mogu da izvršavaju samo jedan program istovremeno, ako ma koja nit unutar procesa pozove jednu od exec funkcija, sve ostale niti se ukidaju (novi program može, na primer, da stvori nove niti). GNU/Linux koristi POSIX standard za niti (poznat kao `PThread`). Sve funkcije niti i tipovi podataka su deklarisane u header datoteci `<pthread.h>`. Pthread funkcije nisu uključene u standardnu C biblioteku. Umesto toga, one se nalaze u libthread, tako da treba da dodate `-lpthread` u komandnoj liniji kada pozivate vaš program.

Kreiranje niti

Svaka nit u procesu se identifikuje pomoću identifikatora niti (thread ID). Kada je u pitanju Identifikator niti u C ili C++ programima, koristite tip `pthread_t`.

Čim se završi kreiranje, svaka nit izvršava nitsku funkciju. Ovo je obična funkcija i sadrži kôd koji nit treba da izvrši. Na GNU/Linux sistemu, nitske funkcije uzimaju jedan parametar, tipa void*, a imaju povratnu vrednost tipa `void`*. Parametar je argument niti, a GNU/Linux prosleđuje njegovu vrednost do niti, bez gledanja u njega. Vaš program može da koristi ovaj parametar da prosledi podatke novoj niti. Slično tome, vaš program može da koristi povratnu vrednost da bi prosledio podatke iz niti koja se završava ka njenom tvorcu.

Funkcija `pthread_create` kreira novu nit.

Ulaz za ovu funkciju je:

- Pokazivač na `pthread_t` promenjivu, u kojoj se smešta identifikator niti za novu nit.
- Pokazivač na atribut niti, objekat. Ovaj objekat kontroliše detalje o tome kako nit vrši interakciju sa ostatkom programa. Ako prosledite vrednost NULL kao atribut niti, nit će biti kreirana sa uobičajenim-default atributima niti.
- Pokazivač na funkciju niti. Ovo je običan pokazivač na funkciju, tipa: `void* (*) (void*)`
- Vrednost argumenta niti, tipa `void*`. Šta god da prosleđujete, to se jednostavno prosleđuje kao argument funkcije niti, kada nit počne da se izvršava.

Poziv `pthread_create` povlači trenutan povratak u originalnu nit koja nastavlja sa izvršavanjem instrukcija iza poziva `pthread_create`. U međuvremenu, nova nit počinje sa izvršavanjem funkcije niti. Linux vrši raspoređivanje obe niti asinhrono, a vaš se program ne sme oslanjati na relativan raspored po kome se instrukcije izvršavaju u okviru te dve niti.

Primer kreiranja niti

Program u listingu 5.1 stvara nit koja u kontinuitetu prikazuje karakter "x" na standardnom izlazu za greške (`stderr`). Nakon pozivanja `thread_create`, glavna nit u kontinuitetu prikazuje karakter "o" na standardnom izlazu za greške.

```
#include <pthread.h>
#include <stdio.h>

/* Ispisuje karakter "x" na standardni izlaz stderr.
Parametar unused nije korišćen. */

void* print_xs (void* unused)
{
  while (1) fputc ('x', stderr);
  return NULL;
}

int main()
{
  // Stvara novu nit. Nova nit izvršava print_xs funkciju
```

```
  pthread_t thread_id;
  pthread_create(&thread_id, NULL, &print_xs, NULL);

  // Ispisuje "o" u kontinuitetu na standardni izlaz stderr.
  while (1) fputc('o', stderr);
  return 0;
}
```

Listing 5.1 (thread-create.c) Kreiranje niti

Prevedite i pokrenite program:

```
$ gcc –o thread-create thread-create.c –lpthread
$ ./thread-create
```

Pokušajte da izvršite program da biste videli šta se dešava. Linux raspoređuje procesor dvema nitima koje naizmenično ispisuju karaktere "x" i "o" – jedna nit nastaje kao posledica izvršavanja glavnog programa, a druga nastaje pozivom funkcije `pthread_create`. Primetite da je pojava karaktera "x" i "o" nepredvidiva.

Pod normalnim okolnostima, nit završava rad na jedan od dva načina. Jedan način, kako je već pokazano, jeste povratkom iz funkcije niti. Povratna vrednost iz funkcije niti predstavlja povratnu vrednost iz niti. Na drugi način, nit može da završi-izađe eksplicitno, pozivanjem sistemskog poziva `pthread_exit`. Ovaj sistemski poziv može biti pozvan od strane funkcije niti ili neke druge funkcije; ta druga funkcija može biti pozvana direktno ili indirektno od strane funkcije niti. Ulazni argument za funkciju `pthread_exit` je povratna vrednost niti.

Prosleđivanje podataka nitima

Argument niti obezbeđuje pogodan metod za prosleđivanje podataka u nit. Zbog toga što je tip argumenta tipa `void`, ne može se proslediti puno podataka direktno preko argumenta. Umesto toga koristite argument niti, da biste prosledili pokazivač na neku strukturu ili na niz podataka. Jedna, uobičajena tehnika je da se definiše struktura za svaku funkciju niti, koja sadrži "parametre" koje funkcija niti očekuje.

Korišćenjem argumenta niti, na lak način se može ponovo upotrebiti ista funkcija niti za mnoge niti. Sve ove niti izvršavaju isti kôd, ali nad različitim podacima.

Program u listingu 5.2 je sličan prethodnom primeru. Program kreira dve nove niti, jednu koja ispisuje "x", a druga koja ispisuje "o". Umesto beskonačnog ispisa karaktera, svaka nit ispisuje utvrđen broj karaktera

i potom izlazi, vraćajući se iz funkcije niti. Istu funkciju niti, char_print, koriste obe niti, ali svaka je konfigurisana drugačije, korišćenjem strukture char_print_parms.

```
#include <pthread.h>
#include <stdio.h>

// Sledeća struktura je parametar char_print funkcije
struct char_print_parms {
  char character; // Karakter koji se ispisuje
  int count; // Broj puta koliko se ispisuje
};

// Ispisuje count karaktera na standardni izlaz stderr
void* char_print (void* parameters)
{
  // Konvertuje generički pokazivač u odgovarajući tip
  struct char_print_parms* p = (struct char_print_parms*)
parameters;
  int i;
  for (i = 0; i < p->count; ++i) fputc(p->character, stderr);
  return NULL;
}

// Glavni program
int main ()
{
  pthread_t thread1_id, thread2_id;
  struct char_print_parms thread1_args;
  struct char_print_parms thread2_args;

  // Stvara novu nit koja ispisuje 30.000 puta X
  thread1_args.character = 'x';
  thread1_args.count = 30000;
  pthread_create(&thread1_id, NULL, &char_print, &thread1_args);

  // Stvara novu nit koja ispisuje 20.000 puta O
  thread2_args.character = 'o';
  thread2_args.count = 20000;
  pthread_create(&thread2_id, NULL, &char_print, &thread2_args);

  return 0;
}
```

Listing 5.2 (thread-create2.c) Stvaranje dve niti

Prevedite i pokrenite program:

```
$ gcc –o thread-create2 thread-create2.c –lpthread
$ ./thread-create2
```

Primetite da je pojava x i o karaktera nepredvidiva, kada Linux alternativno raspoređuje dve niti, a može se dogoditi da se na ekranu ne pojavi ništa.

Pazite! Program u ovom primeru, ima ozbiljnu grešku u sebi. Glavna nit (koja izvršava funkciju `main`) kreira strukture parametara niti (`thread1_args` i `thread2_args`) kao svoje lokalne promenjive, i onda prosleđuje pokazivače na ove strukture nitima koje kreira. Šta sprečava Linux da raspoređuje tri niti na takav način da `main` završi izvršavanje pre nego što se bilo koja od druge dve niti završe? Ništa! Ali ako se to dogodi, memorija koja sadrži strukture parametra niti će biti vraćena, dok druge dve niti još uvek pokušavaju da joj pristupe.

Čekanje da niti završe rad

Kako obezbediti da glavna nit (`main`) sačeka dok druge dve niti završe. Potrebna nam je funkcija koja je slična funkciji `wait`; funkcija `wait` je čekala da se završi proces, a tražena funkcija treba da sačeka da se završi nit. Ta funkcija je `pthread_join`, koja uzima dva argumenta: Identifikator niti za nit koja se čeka, i pokazivač na `void*` promenljivu, koja će primiti povratnu vrednost niti koja završava aktivnost. Ukoliko vas ne zanima povratna vrednost niti, prosleđuje se NULL kao drugi argument.

Ispravka greške prethodnom programu

Sledeći primer pokazuje ispravljenu main funkciju iz prethodnog pogrešnog primera. U ovom slučaju, main ne završava, sve dok obe niti koje ispisuju "x" i "o" ne završe sa radom, tako da više ne koriste argumente strukture. U listingu 5.3 data je ispravljena `main()` funkcija programa iz listinga 5.2.

```
#include <pthread.h>
#include <stdio.h>

int main ()
{
  pthread_t thread1_id;
  pthread_t thread2_id;
  struct char_print_parms thread1_args;
```

```
  struct char_print_parms thread2_args;

  // Stvara novu nit koja ispisuje 30.000 puta X
  thread1_args.character = 'x';
  thread1_args.count = 30000;
  pthread_create (&thread1_id, NULL, &char_print,
thread1_args);

  // Stvara novu nit koja ispisuje 20.000 puta O */
  thread2_args.character = 'o';
  thread2_args.count = 20000;
  pthread_create (&thread2_id, NULL, &char_print,
thread2_args);

  // Provera da je prva nit završena
  pthread_join (thread1_id, NULL);

  // Provera da je druga nit završena
  pthread_join (thread2_id, NULL);

  // Sada je vraćanje bezbedno
  return 0;
}
```

Primer 5.3 (thread-create2r) Čekanje da niti završe rad

Prevedite i pokrenite program:

```
$ gcc –o thread-create2 thread-create2.c –lpthread
$ ./thread-create2
```

Pojava karaktera "x" i "o" je nepredvidiva zbog raspoređivanja niti. Karakter "x" će se pojaviti 30 000 puta, a karakter "o" će se pojaviti 20 000 puta.

Povratne vrednosti niti

Ako je drugi argument, koji prosleđujete funkciji `pthread_join` različit od NULL, povratna vrednost niti će biti smeštena na lokaciji na koju ovaj argument pokazuje. Povratna vrednost niti, kao i argument niti, je tipa `void*`. Ako želite da vratite neku celobrojnu vrednost (`int`) ili neki drugi mali broj, možete to lako da uraditi tako što ćete tu vrednost konvertovati u `void*`, a onda je nakon pozivanja funkcije `pthread_join` ponovo konvertovati u vrednost odgovarajućeg tipa.

Program `primes.c` u listingu 5.4 izračunava n-ti prosti broj, u zasebnoj niti. Ta nit vraća željeni prosti broj, kao svoju povratnu vrednost niti. Glavna nit je, u međuvremenu, slobodna za izvršavanje drugog koda. Primetite da je uzastopni algoritam za deljenje, korišćen u ovom programu prilično neefikasan; konsultujte neku knjigu koja govori o numeričkim algoritmima za izračunavanje prostih brojeva.

```
#include <pthread.h>
#include <stdio.h>

/* Izračunavanje uzastopnih prostih brojeva
(vrlo neefikasno). Vraćanje n-tog prostog broja,
gde je N vrednost ukazana u *ARG */

void* compute_prime (void* arg)
{
  int candidate  = 2;
  int n = *((int*) arg);
  while (1) {
    int factor;
    int is_prime = 1;

    // Testiranje prostih brojeva pomoću deljenja
    for (factor = 2; factor < candidate; ++factor)
      if (candidate % factor ==0 )
      {
        is_prime = 0;
        break;
      }

    // Da li je ovo prost broj koji tražimo?
    if (is_prime)
    {
      if(--n == 0)
      // Vrati željenji prost broj kao povratnu vrednost niti
      return (void*) candidate;
    }
    ++candidate;
  }
return NULL;
}

int main ()
{
```

```
  pthread_t thread;
  int which_prime=5000;
  int prime;
  int i = 0;

  // Počinje izračunavanje niti, sve do 5000tog prostog broja
  pthread_create (&thread, NULL, &compute_prime, &which_prime);

  // Obavljanje nekog posla glavne niti...
  for(i=0;i<10000;i++) printf("Ja, glavna nit, pišem ovo...");

  // Čekanje da se nit završi, a potom da se uzme rezultat
  pthread_join(thread, (void*) &prime);

  // Ispisuje se najveći prost broj koji je izračunat
  printf("%d.prost broj je %d.\n", which_prime, prime);

  return 0;
}
```

Listing 5.4 (primes.c) Izračunavanje n-tog prostog broja pomoću posebne niti

Prevedite i pokrenite program:

```
$ gcc –o primes primes.c -lpthread
$ ./primes
```

Podaci specifični za nit

Za razliku od procesa, sve niti u jednom programu dele isti adresni prostor. Ovo znači da ako jedna nit modifikuje lokaciju u memoriji (na primer, globalna promenjiva), promena je vidljiva za sve ostale niti. Ovo omogućuje višestrukim nitima da rade sa istim podacima bez upotrebe interprocesnih komunikacionih mehanizama.

Međutim, svaka nit ima sopstveni stek. Ovo dozvoljava svakoj niti da izvršava različit kôd i da poziva i vraća iz podrutina na uobičajen način. Kao i u jedno-nitskim programima, svako pozivanje podrutine u svakoj niti ima svoj sopstveni skup lokalnih promenjivih, koje su smeštene na steku niti.

Međutim, ponekad je poželjno kopirati određenu promenjivu tako da svaka nit ima zasebnu kopiju. GNU/Linux podržava ovo tako što svaku nit snabdeva nitski-specificiranim područjem podataka. Promenjive smeštene u ovom području su kopirane za svaku nit, a svaka nit može da menja svoju kopiju promenjive, bez uticaja na druge niti. Zbog toga

što sve niti dele isti memorijski prostor, nitski-specificiranim podacima se ne može pristupiti, korišćenjem normalnih referenci na promenljive. GNU/Linux obezbeđuje specijalne funkcije za podešavanja i vraćanja vrednosti iz nitski-specificiranog područja podataka. Možete da napravite koliko god želite nitski-specificiranih objekata podataka, svaki je tipa `void*`. Svaki objekat referenciran je preko ključa. Da biste stvorili novi ključ, a samim tim i novi objekat podataka za svaku nit, koristite funkciju `pthread_key_create`. Prvi argument je pokazivač na promenljivu tipa `pthread_key_t`. Tu vrednost ključa može koristiti bilo koja nit za pristup sopstvenoj kopiji odgovarajućeg objekta podataka. Drugi argument za `pthread_key_t` je funkcija za čišćenje. Ako ovde prosledite pokazivač funkcije, GNU/Linux automatski poziva tu funkciju kada svaka nit izađe, prosleđujući nitski-specificiranu vrednost, odgovarajuću za taj ključ. Ovo je prilično zgodno zbog toga što se funkcija za čišćenje poziva čak i kada je nit otkazana u nekoj proizvoljnoj tački izvršavanja. Ako je nitski-specificirana vrednost NULL, funkcija za čišćenje niti se ne poziva. Ako vam funkcija za čišćenje nije potrebna, možete proslediti NULL na mesto pokazivača funkcije.

Nakon što ste kreirali ključ, svaka nit može postaviti svoju nitski-specificiranu vrednost koja odgovara tom ključu, pozivanjem funkcije pthread_setspecific. Prvi argument je ključ, a drugi je `void*` nitski-specificirana vrednost. Da biste povratili nitski-specificiran objekt podataka, pozovite funkciju `pthread_getspecific`, prosleđujući joj ključ kao argumenat.

Pretpostavimo, na primer, da vaša aplikacija deli poslove između višestrukih niti. U demonstrativnu svrhu, neka svaka nit ima zasebnu log datoteku, u koju se snimaju poruke o zadacima niti. Nitski-specificirano područje podataka je pogodno mesto za smeštanje pokazivača za log datoteku svake zasebne niti.

Primer `tsd.c` u listingu 5.5 pokazuje vam kako da ovo izvršite. Main funkcija u ovom jednostavnom programu, stvara ključ za nitski-specificirani pokazivač datoteke a onda ga smešta u `thread_log_key`. Zbog toga što je ovo globalna promenjiva, nju dele sve niti. Kada svaka nit počne da izvršava svoju funkciju niti, ona otvara svoju log datoteku i smešta pokazivač na datoteku pod tim ključem. Kasnije, bilo koja od ovih niti može pozvati funkciju `write_to_thread_log`, da bi pisala poruku u nitski-specificiranu log datoteku. Ta funkcija vraća pokazivač za log datoteku iz nitski-specificiranih podataka a zatim se upisuje poruka. U našem primeru, u svaku log datoteku smo upisali poruku "Početak niti" i konkretno ime log datoteke za svaku nit posebno.

```
#include <malloc.h>
#include <pthread.h>
```

```
#include <stdio.h>

// Ključ za vezivanje pokazivača log datoteke sa svakom niti
static pthread_key_t thread_log_key;

// Ispis PORUKE log datoteke za tekuću nit
void write_to_thread_log (const char* message)
{
  FILE* thread_log=(FILE*) pthread_getspecific(thread_log_key);
  fprintf(thread_log, "%s\n", message);
}

// Zatvaranje pokazivača log datoteke THREAD_LOG
void close_thread_log (void* thread_log)
{
  fclose ((FILE*) thread_log);
}

void* thread_function (void* args)
{
  char thread_log_filename[20];
  FILE* thread_log;

  // Generisanje imena datoteke za ovu log datoteku niti
  sprintf (thread_log_filename, "thread%d.log",
    (int) pthread_self ());

  // Otvaranje log datoteke
  thread_log = fopen (thread_log_filename, "w");

  /* Smeštanje pokazivača datoteke u nitski-određenim podacima
  koji odgovaraju ključu thread_log_key */
  thread_setspecific (thread_log_key, thread_log);

  write_to_thread_log("Pocetak niti");
  write_to_thread_log(thread_log_filename);
  // Radite posao ovde

  return NULL;
}

int main ()
{
  int i;
  pthread_t threads [5];
```

```
  /* Napravite ključ da biste pridružili pokazivač nitske log
  datoteke za nitski određene podatke. Koristite
  close_thread_log da biste obrisali pokazivače datoteka */
  pthread_key_create (&thread_log_key, close_thread_log);

  // Stvaranje niti da biste uradili neki posao
  for (i = 0; i < 5; i++)
  pthread_create (&(threads[i]), NULL, thread_function, NULL);

  // čekanje da sve niti završe
  for (i = 0; i < 5; i++) pthread_join (threads[i], NULL);
return 0;
}
```

Listing 5.5 (tsd.c) Podaci specifični za nit

Prevedite i pokrenite program:

```
$ gcc -o tsd tsd.c -lpthread
$ ./tsd
```

Primetite da thread_function ne mora da zatvori log datoteku. To je zbog toga što, kada se kreira ključ log datoteke, `close_thread_log` je određen kao funkcija za čišćenje tog ključa. Uvek kada nit izlazi, GNU/Linux poziva tu funkciju, prosleđujući joj nitski-specificiranu vrednost za ključ log datoteke niti. Ova funkcija vodi računa o zatvaranju log datoteke.

6. Sinhronizacija niti

Nalik procesima i niti se prirodno izvršavaju asinhrono. To znači da je u opštem slučaju trenutak njihovog prekida nepredvidiv. Raspoređivanje niti je u nadležnosti procesora – ne smemo praviti nikakve pretpostavke o njihovim relativnim brzinama. U jednom trenutku procesor opslužuje jednu nit, tako da se izvršavanje niti odvija pseudo-paralelno.

Sinhronizacija i kritične sekcije

Problem asinhronosti dobija na značaju kada se zna da niti mogu raditi kooperativno, odnosno da mogu deliti zajedničke podatke. Pristup toj deljenoj strukturi podataka mora biti kontrolisan na način da se ne može desiti utrkivanje niti (*race conditions*). Ukoliko više niti naizmenično menjaju deljene podatke (utrkuju se), to dovodi do grubih grešaka. Ideja kritične sekcije zasniva se na zabrani da dve ili više niti istovremeno budu u kritičnoj sekciji. Nit može biti prekinuta u kritičnoj sekciji, ali se ne sme dogoditi da taj prekid iskoristi druga nit za ulazak u kritičnu sekciju.

Razmotrimo sledeći primer: niz poslova čeka u redu čekanja pri čemu je svaki predstavljen jednom niti. Preciznije, red čekanja je predstavljen jednostruko povezanom listom čiji elementi sadrže dva polja: strukturu (*job*) i pokazivač na sledeći element liste. "Posao" je predstavljen jednim elementom liste. Dakle, više konkurentnih niti proverava red čekanja u cilju dobijanja informacije da li je "posao dostupan" odnosno da li se nalazi u redu čekanja (povezanoj listi). U ovom primeru uzimanje elementa iz liste svodi se na uzimanje sa njenog početka. Podsetimo, jednostuko povezana lista uvek mora imati pokazivač na njen početak, odnosno na prvi element liste. Ako uzmemo jedan element sa početka liste, pokazivač na početak liste (`job_queue`) mora pokazivati na prvi sledeći element.

Primer koji sledi specijalno je napravljen za slučaj kada se lista sastoji od jednog elementa. Takođe, specijalno je iskorišćen sistemski poziv `sched_yield()`, koji prouzrokuje da se nit koja se izvršava odrekne procesora za neko vreme. Znači, posle poziva `sched_yield()`, druga nit preuzima procesor i nastavlja da radi sa stanjem koji je ostavila prethodna nit. Primetite da nisu korišćeni nikakvi mehanizmi sinhronizacije, tako da ni sa čim nije sprečeno da obe niti istovremeno uđu u svoju kritičnu sekciju. Ako bi recimo uslov ulaska u kritičnu sekciju bila provera da li je lista neprazna, moguće je da bi obe niti prošle taj uslov. Kritična sekcija u ovom primeru bio bi deo koda u kome se pristupa elementu liste:

```
struct job* next_job = job_queue;
job_queue = job_queue->next;
```

Ukoliko bi jedna nit izvršila prvu naredbu, novostvoreni pokazivač `next_job` bi pokazivao na početak liste (jer je `job_queue` pokazivač na početak liste). Ako bi sada druga nit preuzela procesor i takođe izvršila prvu naredbu, i ona bi napravila pokazivač `next_job` koji takođe pokazuje na početak liste. Ovo je već opasna situacija, jer dva pokazivača koji pripadaju različitim nitima pokazuju na isti element liste. Ako bi sada prva nit ponovo preuzela procesor, nastavila bi da radi tamo gde je stala, što znači da bi izvršila sledeću naredbu, a to je pomeranje pokazivača koji pokazuje na početak liste (`job_queue`) na sledeći element liste. Ako bi se lista sastojala samo od jednog elementa, ovo bi dovelo do dodeljivanja NULL vrednosti pokazivaču `job_queue` (jer polje poslednjeg elementa liste za pokazivač na sledeći element je NULL) a to znači da bi lista u ovom trenutku bila prazna. Kada bi se procesor ponovo dodelio drugoj niti kojoj je za izvršenje ostala druga naredba kritične sekcije, ta nit bi naišla na praznu listu što bi izazvalo grešku u segmentaciji (*segmentation fault*). Program u listingu 6.1 demonstrira upravo tu situaciju.

```
#include<malloc.h>
#include<pthread.h>
#include<stdio.h>

struct job
{
  struct job *next;
  int br;
};

struct job* job_queue = NULL;

void process_job(struct job* x)
{
  printf("Broj je: %d\n",x->br);
}

void* thread_function(void* arg)
{
  while(job_queue != NULL)
  {
    struct job* next_job = job_queue;
    sched_yield();
    job_queue = job_queue->next;
    process_job(next_job);
    free(next_job);
```

```
  }
  return NULL;
}

int main()
{
  pthread_t thread[2];
  int i;

  struct job *pom = malloc(sizeof(struct job));
  pom->next = job_queue;
  pom->br = 7;
  job_queue = pom;

  for(i=0;i<2;i++)
    pthread_create(&thread[i],NULL,&thread_function,&job_queue);
  for(i=0;i<2;i++)
    pthread_join(thread[i],NULL);

  return 0;
}
```

Listing 6.1 (job-queue1.c) Sinhronizacija i kritične sekcije

Prevedite i pokrenite program:

```
$ gcc job-queue1.c -o job-queue1 -lpthread
$ ./job-queue1
```

Rešenje pomoću MUTEX semafora

Kako rešiti prethodni problem? Logična ideja je da jedna nit čeka dok druga ne završi svoju kritičnu sekciju. Dakle, jedna nit treba da čeka, dok druga ne obavi svoje operacije sa listom, tako da to ne omete nit koja je čekala da posle toga uradi isto za svoje potrebe. Implementacija ovog mehanizma može se uraditi korišćenjem binarnog semafora. Generičko ime za binarni semafor je mutex (*MUTual EXclusion*). Binarni semafor se može naći u stanju 0 ili 1. Kada nit naiđe na stanje 0, mora da čeka; u suprotnom prolazi i postavlja vrednost binarnog semafora na 0, čime "zatvara vrata za sobom". Dok nit koja je zaključala vrata u jednom trenutku sama ne otvori bravu (postavljanjem binarnog semafora na 1), dotle sve druge niti koje žele prolaz kroz semafor moraju čekati. Zamislite da samo jedan auto može proći semafor i da se odmah posle njegovog prolaska pali crveno svetlo. Kada taj auto završi

vožnju, na semaforu se pali zeleno svetlo, ali opet, po istom principu, semafor može proći samo jedan auto.

Jedan od načina za stvaranje mutex-a postiže se naredbom:

```
pthread_mutex_t job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;
```

Nit zaključava mutex pozivom funkcije `pthread_mutex_lock`, a otključava naredbom `pthread_mutex_unlock`. U programu koji sledi kritična sekcija je zaštićena mutex-om `job_queue_mutex`. Zbog poređenja sa prethodnim programom, napravljena je lista od jednog elementa, ali se to lako može promeniti u for petlji. Ovaj put, napravljeno je pet niti, da bi se pokazalo da više njih može da čeka na semaforu mutex. Takođe, izvršene su manje promene u nitskoj funkciji thread_function, ali to ne menja suštinu programa: ako je lista prazna ne izlazimo trenutno iz `while` petlje; u tom slučaju bi mutex ostao zaključan za sva vremena, sprečavajući bilo koju drugu nit da pristupi listi. Napomenimo da funkcija `process_job` obavlja neku nebitnu aktivnost konkretne niti; u ovom slučaju ispisuje nulu. Izlaz programa datog u listingu 6.2 potpuno je regularan.

```
#include<malloc.h>
#include<pthread.h>
#include<stdio.h>

struct job {
  struct job *next;
  int br;
};

struct job* job_queue = NULL;
pthread_mutex_t job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;

void process_job(struct job* x)
{
  printf("Broj je: %d\n",x->br);
}

void* thread_function(void* arg)
{
  while(1)
  {
    struct job *next_job;

    pthread_mutex_lock(&job_queue_mutex);
    if(job_queue == NULL) next_job = NULL;
```

```
    else
    {
      next_job = job_queue;
      job_queue = job_queue->next;
    }
    pthread_mutex_unlock(&job_queue_mutex);

    if(next_job == NULL) break;
    process_job(next_job);
    free(next_job);
  }
  return NULL;
}

int main()
{
  pthread_t thread[5];
  int i;
  for(i=0;i<1;i++)
  {
    struct job *pom = malloc(sizeof(struct job));
    pom->next = job_queue;
    pom->br = i;
    job_queue = pom;
  }

  for(i=0;i<5;i++)
    pthread_create(&thread[i],NULL,&thread_function,&job_queue);
  for(i=0;i<5;i++)
    pthread_join(thread[i],NULL);

  return 0;
}
```

Listing 6.2 (job-queue2.c) Rešenje pomoću MUTEX semafora

Prevedite i pokrenite program:

```
$ gcc job-queue2.c -o job-queue2 -lpthread
$ ./job-queue2
```

Rešenje pomoću brojačkog semafora

Glavna nit (`main`) funkcioniše tako što čeka da se završe niti koje je proizvela (ukoliko je pozvana funkcija `pthread_join`) i onda i sama završava rad. Nedostatak ovakvog pristupa ogleda se u izostanku provere da li su proizvedene niti prebrzo izašle, pre nego što su dobile podatke za obradu. Recimo, u prethodnom primeru, red čekanja (povezana lista) mora biti ili napunjen unapred, ili se mora puniti barem onoliko brzo koliko niti potražuju njegove poslove (elemente liste). Ako bi stvorene niti radile prebrzo, naišle bi na praznu listu i izašle pre nego što bi uzele i jedan element. Zato nam je potreban mehanizam koji će blokirati niti dok je red čekanja prazan. Kada stigne jedan posao, biće odblokirana jedna nit i tako redom. Ovo možemo postići upotrebom brojačkog semafora. Za razliku od binarnog semafora koji uzima vrednosti nula ili jedan, brojački semafor uzima različite celobrojne vrednosti. U slučaju da mu je vrednost nula ili manja od nule, izaziva blokiranje nadolazećih niti. Prva vrednost koja izaziva blokiranje je vrednost nula i nit koja dolazi prelazi u blokirano stanje spuštajući vrednost brojačkog semafora na -1. Dakle kada je vrednost brojačkog semafora -N, to znači da se N niti nalazi u redu čekanja. Kada neka slobodna nit poveća vrednost semafora, onda će jedna od niti koje čekaju proći semafor, i ući u svoju kritičnu sekciju. Dok je ona u kritičnoj sekciji, ukoliko je vrednost semafora nepozitivna, ni jedna druga nit neće moći da uđe u kritičnu sekciju. Na prvi pogled predstavljeni mehanizam ostavlja utisak da brojački semafor uzima samo nenegativne vrednosti, budući da blokiranje na semaforu počinje kada njegova vrednost postane nula; ipak, teorijski, brojački semafor može uzeti sve moguće celobrojne vrednosti, a njegova negativna vrednost daje informaciju o broju blokiranih niti u redu čekanja. Pod Linuxom, brojački semafor posmatramo kao semafor sa nenegativnim vrednostima. Kada je vrednost semafora nula, sve nadolazeće niti čekaju; kada neka druga nit poveća vrednost semafora, jedna nit iz reda čekanja prolazi i vrednost semafora opet pada na nulu. U našem primeru početna vrednost semafora je nula.

Semafor poseduje dve osnovne operacije: *wait* i *post*. *Wait* operacija snižava vrednost semafora za jedan: ako je vrednost već bila jednaka nuli, nadolazeća nit se blokira; u suprotnom, vrednost semafora se smanjuje za jedan, ali nit prolazi semafor. *Post* operacija povećava vrednost semafora za jedan. Niti koje čekaju na semaforu, mogu ga proći samo u slučaju njegove pozitivne vrednosti. Za rad sa semaforima uključite zaglavlje `<semaphore.h>`. Semafor je predstavljen promenljivom tipa `sem_t`. Pre upotrebe, semafor se inicijalizuje `sem_init` funkcijom. Prvi parametar ove funkcije je pokazivač na promenljivu tipa `sem_wait`, drugi argument je nula, a treći je inicijalna vrednost semafora. Za smanjenje vrednosti semafora koristi se operacija `sem_wait`, a za

povećanje vrednosti operacija `sem_post`. Razmotrimo program `job-queue3.c` dat u listingu 6.3.

```
#include<malloc.h>
#include<pthread.h>
#include<stdio.h>
#include<semaphore.h>

struct job
{
  struct job *next;
  int br;
};

struct job* job_queue = NULL;
pthread_mutex_t job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;
sem_t job_queue_count;

void initialize_job_queue()
{
  job_queue = NULL;
  sem_init(&job_queue_count,0,0);
}

void process_job(struct job* x)
{
  printf("Broj je: %d\n",x->br);
}

void* thread_function(void* arg)
{
  while(job_queue!=NULL)
  {
    struct job *next_job;
    sem_wait(&job_queue_count);

    pthread_mutex_lock(&job_queue_mutex);
    next_job = job_queue;
    job_queue = job_queue->next;
    pthread_mutex_unlock(&job_queue_mutex);

    process_job(next_job);
    free(next_job);
  }
```

```
  return NULL;
}

void enqueue_job(int i)
{
  struct job* new_job=(struct job*) malloc(sizeof(struct job));
  pthread_mutex_lock(&job_queue_mutex);
  new_job->next = job_queue;
  new_job->br = i;
  job_queue = new_job;
  sem_post(&job_queue_count);
  pthread_mutex_unlock(&job_queue_mutex);
}

int main()
{
  pthread_t thread[5];
  int i;

  initialize_job_queue();

  for(i=0;i<5;i++)
    pthread_create(&thread[i], NULL, &thread_function,
      &job_queue);

  for(i=0;i<10;i++) enqueue_job(i);

  for(i=0;i<5;i++)
    pthread_join(thread[i],NULL);

  return 0;
}
```

Listing 6.3 (job-queue3.c) Rešenje pomoću brojačkog semafora

Prevedite i pokrenite program.

```
$ gcc job-queue3.c -o job-queue3 -lpthread
$ ./job-queue3
```

Primetite da je ispravljen nedostatak prethodnog programa (`job-queue2.c`, listing 6.2). Pre uzimanja posla (elementa liste) sve niti prvo čekaju na semaforu `job_queue_count`. Da bi se uzeo element iz liste, mora se prvo proći operacija `sem_wait` (koja se nalazi u funkciji `thread_function`). To se može desiti jedino ako se dogodi operacija `sem_post` (koja se nalazi u funkciji `enqueue_job`) posle završenog

dodavanja elementa u listu. Na ovaj način, postiže se sinhronizacija jer niti moraju sačekati da se lista napuni barem jednim elementom, pa tek posle obrade novostvorene liste mogu završiti rad.

Vežbe

[1] Prevedite, pokrenite i objasnite program `thread1.c` iz listinga 6.4.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

char message[] = "Zdravo svete!";

void *thread_function(void *arg)
{
  printf("Nitska funkcija se izvrsava...\n");
  sleep(3);
  strcpy(message, "Do vidjenja!");
  pthread_exit("Hvala za procesorsko vreme.");
}

int main()
{
  pthread_t a_thread;
  void *thread_result;
  pthread_create(&a_thread, NULL, thread_function,
    (void *)message);
  printf("Cekanje da nit zavrsi...\n");
  pthread_join(a_thread, &thread_result);
  printf("Nit je zavrsila rad i vraca vrednost %s\n",
    (char *)thread_result);
  printf("Poruka je sada %s\n", message);
}
```

Listing 6.4 (thread1.c)

[2] Prevedite, pokrenite i objasnite program `thread2.c` iz listinga 6.5.

```
#include <stdio.h>
#include <unistd.h>
```

```
#include <stdlib.h>
#include <pthread.h>
int run_now = 1;
char message[] = "Zdravo svete!";

void *thread_function(void *arg)
{
  int print_count2 = 0;
  while(print_count2++ < 20)
  {
    if (run_now == 2)
    {
      printf("2");
      run_now = 1;
    }
    else sleep(1);
  }
}

int main() {
  pthread_t a_thread;
  void *thread_result;
  int print_count1 = 0;
  pthread_create(&a_thread, NULL, thread_function,
    (void *)message);
  while(print_count1++ < 20)
  {
    if (run_now == 1)
    {
      printf("1");
      run_now = 2;
    }
    else sleep(1);
  }
  printf("\nCekanje da nit zavrsi...\n");
  pthread_join(a_thread, &thread_result);
  printf("Nit je zavrsila rad zahvaljujuci naredbi
pthread_join\n");
}
```

Listing 6.5 (thread2.c)

[3] Prevedite, pokrenite i objasnite program `thread3.c` iz listinga 6.6.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <string.h>

sem_t sem;
#define WORK_SIZE 1024
char work_area[WORK_SIZE];

void *thread_function(void *arg)
{
  sem_wait(&sem);
  while(strncmp("end", work_area, 3) != 0) {
    printf("Uneli ste %d karaktera\n", strlen(work_area) -1);
    sem_wait(&sem);
  }
}

int main()
{
  pthread_t a_thread;
  void *thread_result;
  sem_init(&sem, 0, 0);
  pthread_create(&a_thread, NULL, thread_function, NULL);
  printf("Unesi neki tekst. Unesite 'end' za kraj\n");
  while(strncmp("end", work_area, 3) != 0) {
    fgets(work_area, WORK_SIZE, stdin);
    // čeka na unos sa standardnog ulaza
    sem_post(&sem);
  }
  printf("\nCekanje da nit zavrsi...\n");
  pthread_join(a_thread, &thread_result);
  printf("Nit je zavrsila rad zahvaljujuci naredbi
pthread_join\n");
  sem_destroy(&sem);
}
```

Listing 6.6 (thread3.c)

[4] Prevedite, pokrenite i objasnite program `thread4.c` iz listinga 6.7.

Napomena: u jednom trenutku uneti sa tastature reč "BRZO" i prokomentarisati izlaz. Kolika je u tom slučaju vrednost brojačkog semafora sem? Obratiti pažnju da dok još ništa nije uneto, glavna nit (`main`) verovatno dolazi do funkcije `fgets`, gde zatim čeka na dalji unos sa tastature. U međuvremenu druga nit dobija procesor, ali njena prva naredba `sem_wait` je blokira, jer je početna vrednost semafora sem jednaka nuli. Glavna nit ponovo dobija procesor i ukoliko unesete "BRZO", nastavlja rad od tačke gde je stala (funkcija `fgets`). Glavna nit će ponovo biti blokirana (i prepustiti procesor) tek kada ponovo dođe do funkcije `fgtets` koja čeka na unos sa tastature. Šta se može reći o semaforu `sem`? Na koju vrednost će doći, pre nego što druga nit dobije procesor?

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <string.h>

sem_t sem;
#define WORK_SIZE 1024
char work_area[WORK_SIZE];

void *thread_function(void *arg)
{
  sem_wait(&sem);
  while(strncmp("end", work_area, 3) != 0)
  {
    printf("Uneli ste %d karaktera\n", strlen(work_area) -1);
    sem_wait(&sem);
  }
}

int main()
{
  pthread_t a_thread;
  void *thread_result;
  sem_init(&sem, 0, 0);
  pthread_create(&a_thread, NULL, thread_function, NULL);
  printf("Unesite neki tekst. Unesite 'end' za kraj\n");
  while(strncmp("end", work_area, 3) != 0)
```

```
  {
    if (strncmp(work_area, "BRZO", 4) == 0)
    {
      sem_post(&sem);
      strcpy(work_area, "Ha..ha...");
    }
    else
      fgets(work_area, WORK_SIZE, stdin);
    }
    sem_post(&sem);
  }
  printf("\nCekanje da nit zavrsi...\n");
  pthread_join(a_thread, &thread_result);
  printf("Nit je zavrsila rad zahvaljujuci naredbi
pthread_join\n");
  sem_destroy(&sem);
}
```

Listing 6.7 (thread4.c)

7. IPC: deljiva i mapirana memorija

U poglavlju 4 razmotrili smo sistemski poziv `wait`, pomoću koga jedan proces-roditelj može dobiti izlazni (exit) status od procesa deteta. To je najjednostavnija forma komunikacije između dva procesa, ali ne i najmoćnija. Mehanizam `fork`, `exec`, `wait`, `exit`, ne omogućava nijedan vid komunikacije roditelja i deteta izuzev preko argumenata komandne linije i varijabli okoline, niti omogućava bilo koji način komunikacije deteta sa roditeljem izuzev preko detetovog izlaznog statusa. Nijedan od ovih mehanizama ne pruža način ili sredstvo za komunikaciju sa procesom detetom dok je ono u stanju izvršavanja, niti ovi mehanizmi dozvoljavaju komunikaciju sa procesima izvan odnosa roditelj-dete.

Ovo poglavlje opisuje sredstva za interprocesnu komunikaciju koja zaobilaze ova ograničenja. Predstavićemo različite načine komunikacije između roditelja i deteta, između "nepovezanih" procesa, pa čak i između procesa na različitim mašinama.

Interprocesna komunikacija (IPC) je prenos podataka između procesa. Na primer, Web pretraživač može tražiti Web stranicu sa Web servera, koji zatim šalje HTML podatke. Ovaj prenos podataka obično koristi soket mehanizam u vidu telefonske konekcije. U drugom primeru, želite da štampate imena datoteka u direktorijumu koristeći komandu ls | lpr. Shell (komandni interpreter) stvara ls proces i poseban lpr proces, povezujući ova dva procesa pomoću pipe mahanizma-datoteke, predstavljenog simbolom " | " . Pipe mehanizam dozvoljava jednosmernu komunikaciju između dva srodna procesa. Proces ls upisuje podatke u pipe, a proces lpr ih čita iz pipe.

U sledeća dva poglavlja razmotrićemo pet tipova interprocesne komunikacije:

- Deljiva memorija (*shared memory*) dozvoljava procesima da komuniciraju prostim čitanjem i upisivanjem u određenu memorijsku lokaciju.
- Mapirana memorija je slična deljivoj memoriji, izuzev što je povezana sa datotekom u sistemu datoteka
- Pipe datoteke dozvoljavaju sekvencijalnu komunikaciju jednog procesa sa srodnim procesom.
- FIFO datoteke su slični kao pipe, izuzev što i nepovezani procesi mogu komunicirati jer FIFO datoteka ima dodeljeno ime u sistemu datoteka
- Soketi podržavaju komunikaciju između nepovezanih procesa čak i na različitim računarima.

Ovi tipovi IPC razlikuju se po sledećim kriterijumima:

- Da li ograničavaju komunikaciju na srodne procese (procese sa zajedničkim pretkom), na nepovezane procese koji dele isti sistem datoteka, ili na bilo koji računar povezan u mrežu.
- Da li je komunikacioni proces ograničen samo na upisivanje podataka ili samo na čitanje podataka.
- Broju procesa kojima je dozvoljeno da komuniciraju.
- Da li su komunikacioni procesi sinhronizovani pomoću IPC - na primer proces čitanja je u stanju čekanja sve dok podaci ne postanu dostupni za čitanje.

U ovom poglavlju zanemaričemo diskusiju o IPC dozvoli za komunikaciju samo određeni broj puta, kao što je komunikacija preko izlaznog statusa deteta.

Deljiva memorija

Jedan od najjednostavnijih metoda interprocesne komunikacije je upotreba deljive memorije. Deljiva memorija omogućava, da dva ili više procesa pristupe istoj memoriji ako pozovu `malloc` i ako svi vraćeni pokazivači ukazuju na istu memoriju. Kada jedan proces promeni neku memorijsku reč, svi ostali procesi vide promenu.

Brza lokalna komunikacija

Deljiva memorija je najbrža forma interprocesne komunikacije zato što svi procesi dele isti deo memorije. Pristup ovoj deljivoj memoriji je isto toliko brz kao i pristup procesovoj nedeljivoj memoriji i ne zahteva sistemski poziv ili pristup kernelu. Takođe se izbegava nepotrebno kopiranje podataka.

Ozbirom da kernel ne sinhornizuje pristupe deljivoj memoriji, potrebno je da obezbedite svoju sopstvenu sinhronizaciju. Na primer, proces ne bi trebao da čita iz memorije dok se ne završi upis podataka u nju, niti bi dva procesa trebala da upisuju u istu memorijsku lokaciju u isto vreme.

Česta strategija za izbegavanje ovih stanja trke je upotreba semafora.

Naši ilustrativni programi, ipak, prikazuju kako samo jedan proces pristupa memoriji da bi se fokusirali na mehanizam deljive memorije i kako bi izbegli da iskomplikujemo kôd primera sa sinhronizacionom logikom.

Memorijski model

Da bi koristio deljivi memorijski segment, jedan proces mora alocirati segment. Zatim svaki proces koji želi da pristupi segmentu mora priključiti (*attach*) segment. Nakon završetka korišćenja segmenta, svaki proces mora isključiti (*detach*) segment. Na kraju, bar jedan proces mora osloboditi (*deallocate*) segment.

Razumevanje Linux memorijskog modela pomaže da se objasni alokacija i priključivanje procesa. Pod Linux-om, svaka virtuelna memorija procesa podeljena je na stranice. Svaki proces mora voditi računa o mapiranju svojih memorijskih adresa u virtualne memorijske stranice, koje sadrže aktuelne podatke. Iako svaki proces poseduje sopstvene adrese, višestruko procesno mapiranje može ukazivati na istu stranicu, dozvoljavajući deljenje memorije.

Alokacija novog deljivog memorijskog segmenta, prouzrokuje kreiranje virtuelnih memorijskih stranica. Obzirom da svi procesi žele da pristupe istom deljivom segmentu, samo bi jedan proces trebao da alocira novi deljivi segment. Alociranjem postojećeg segmenta ne kreiraju se nove stranice, već se vraća identifikator za već postojeće stranice. Da bi dozvolio procesu da koristi deljivi memorijski segment, proces ga priključuje u svoj adresni prostor, što dodaje mapirane ulaze u njegovu virtualnu memoriju, koje odgovaraju stranicama deljivog segmenta. Kada završimo sa upotrebom segmenta, ovi mapirani ulazi se uklanjaju. Kada više nema procesa koji bi koristili deljivi memorijski segment, tačno jedan proces mora delocirati virtuelne memorijske stranice segmenta.

Svi deljivi memorijski segmenti alocirani su kao celobrojni umnošci veličine sistemske stranice, koja je zadata kao broj bajtova u stranici memorije. Na Linux sistemima, veličina stranice iznosi 4KB, ali vi možete odrediti ovu vrednost pozivajući funkciju getpagesize.

Alokacija

Proces alocira deljivi memorijski segment korišćenjem `shmget` ("*SHared Memory GET*"). Njegov prvi parametar je celobrojni ključ, koji određuje koji segment se kreira. Nesrodni procesi mogu pristupiti istom deljivom segmentu pomoću specifiranja iste vrednosti ključa. Na nesreću, drugi procesi mogu takođe odabrati isti fiksni ključ, a što može voditi do konflikta. Upotrebom specijalne konstante `IPC_PRIVATE` kao vrednosti ključa, garantuje da će se kreirati potpuno novi memorijski segment.

Njegov drugi parametar definiše broj bajtova u segmentu. Obzirom da su segmenti alocirani korišćenjem stranica, broj stvarno alociranih bajtova je zaokružen na ceo broj stranica.

Treći parametar je jedan bit ili vrednost flega koji određuje opcije za shmget. Vrednosti flega uključuju sledeće:

- `IPC_CREAT` — Ovaj fleg ukazuje na potrebu za kreiranjem novog segmenta. Ovo omogućava kreiranje novog segmenta na bazi vrednosti ključa.
- `IPC_EXCL` — Ovaj fleg koji se uvek koristi sa IPC_CREAT, prouzrokuje neuspeh u izvršenju shmget, u slučaju da je zadati ključ segmenta koji već postoji. Zbog toga, on uređuje da pozvani proces uvek dobije poseban-ekskluzivan-nov segment. U slučaju da ovaj fleg nije zadat a koristi se ključ već postojećeg segmenta, shmget vraća postojeći segment umesto da kreira novi.
- Mode flags — Ova vrednost je sačinjena od 9 bitova koji ukazuju na prava dodeljena vlasniku, grupi i ostatku sveta (others) za kontrolu pristupa segmentu. Izvršni bitovi se zanemaruju. Jednostavan način da se odrede prava je korišćenje konstanti definisanih u `<sys/stat.h>`. Na primer, `S_IRUSR` i `S_IWUSR` određuju prava čitanja i upisivanja za vlasnika deljivog memorijskog segmenta, a `S_IROTH` i `S_IWOTH` određuju prava čitanja i upisivanja za ostale (*others*).

Na primer, ovakvo pozivanje `shmget`, kreira novi deljivi memorijski segment (ili pristup već postojećem u slučaju da je `shm_key` već upotrebljen) koji je čitljiv i upisljiv za vlasnika, ali ne i za druge korisnike.

```
int segment_id = shmget (shm_key, getpagesize(),
                         IPC_CREAT | S_IRUSR | S_IWUSER);
```

U slučaju da poziv uspe, `shmget` vraća identifikator segmenta. Ako deljivi memorijski segment već postoji, vrši se verifikacija prava pristupa i proverava se da li je segment označen za uništenje.

Pridruživanje i izbacivanje

Da bi deljivi memorijski segment učinio dostupnim, proces mora koristiti poziv shmat, "*SHared Memory ATach*." Ovom pozivu treba proslediti identifikator deljivog memorijskog segmenta koji se dobija pomoću `shmget`. Drugi argument je pokazivač, koji određuje gde u adresnom prostoru procesa, želite da mapirate deljivu memoriju, a ako odrediti vrednost NULL, Linux će odabrati raspoloživu adresu. Treći argument je fleg, koji može uključivati sledeće:

- SHM_RND ukazuje da adresa koja je određena za drugi parametar, treba da bude zaokružena na umnožak veličine stranice. Ako ne postavite ovaj fleg, potrebno je lično podesiti drugi argument za shmat.
- SHM_RDONLY ukazuje da će segment biti samo čitljiv, ne i upisiv.

Ako poziv shmat uspe, on vraća adresu pridruženog deljivog segmenta. Deca kreirana fork pozivom nasleđuju pridružene deljive segmente, a ako to žele, ona mogu izbaciti deljive memorijske segmente iz svog prostora.

Kada ste završili sa deljivim memorijskim segmentom, segment je potrebno izbaciti koristeći shmdt ("*SHared Memory DeTach*"). Za shmdt treba koristiti adresu dobijenu pomoću shmat. Ako se segment oslobodi i ako je ovo bio poslednji proces koji ga je koristio, on se uklanja. Sistemski poziv exit ili pozivanje bilo kog člana exec familije, automatski izbacuje segment.

Kontrola i oslobađanje deljive memorije

Poziv shmctl ("*SHared Memory ConTroL*" – kontrola deljive memorije) vraća informaciju o deljivom memorijskom segmentu i može ga modifikovati. Prvi parametar je identifikator deljivog memorijskog segmenta.

Da bi dobili informaciju o deljivom memorijskom segmentu potrebno je proslediti: IPC_STAT kao drugi argument i pokazivač na strukturu shmid_ds.

Za uklanjanje segmenta potrebno je proslediti: IPC_RMID kao drugi argument i NULL kao treći argument. Segment se uklanja kada ga i poslednji proces koji ga je priključio (*attach*) konačno izbaci (*detach*) iz svog adresnog prostora.

Svaki deljivi memorijski segment trebalo bi eksplicitno osloboditi (deallocate) koristeći shmctl kada se završi sa njim, da bi se izbeglo narušavanje sistemskog ograničenja u pogledu ukupnog broja deljivih memorijskih segmenata. Sistemski pozivi exit i exec izbacuju (*detach*) memorijski segment, ali ga ne oslobađaju.

Pogledajte shmctl man stranicu za opis drugih operacija koje se mogu obaviti nad deljivim memorijskim segmentima.

Program shm.c u listingu 7.1 ilustruje upotrebu deljive memorije.

```
#include <stdio.h>
#include <sys/shm.h>
```

```
#include <sys/stat.h>

int main () {
  int segment_id;
  char* shared_memory;
  struct shmid_ds shmbuffer;
  int segment_size;
  const int shared_segment_size = 0x6400;

  // Alocira deljivi memorijski segment
  segment_id = shmget (IPC_PRIVATE, shared_segment_size,
    IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);

  // Pridružuje deljivi memorijski segment
  shared_memory =  (char*)  shmat  (segment_id, 0, 0);
  printf ("deljiva memorija pridružena na adresi %p\n",
    shared_memory);

  // Određuje veličinu segmenta
  shmctl (segment_id, IPC_STAT, &shmbuffer);
  segment_size = shmbuffer.shm_segsz;
  printf ("veličina segmenta: %d\n", segment_size);

  // Upisuje string u deljivi memorijski segment
  sprintf (shared_memory, "Hello, world.");

  // Odbacivanje deljivog memorijskog segmenta
  shmdt (shared_memory);

  // Ponovno pridruživanje deljivog segmenta na drugoj adresi
  shared_memory = (char*) shmat (segment_id,
    (void*) 0x5000000, 0);
  printf ("deljiva memorija ponovo pridružena na adresi %p\n",
    shared_memory);

  // Ispisivanje stringa iz deljive memorije
  printf ("%s\n", shared_memory);

  // Odbacivanje deljivog memorijskog segmenta
  shmdt (shared_memory);

  // Oslobađanje (dealokacija) deljivog memorijskog segmenta
  shmctl (segment_id, IPC_RMID, 0);

return 0;
```

```
}
```

Listing 7.1 (shm.c) Deljiva memorija

Prevedite i pokrenite program:

```
$ gcc –o shm shm.c
$ ./shm
```

Potencijalni izlaz programa je:

```
deljiva memorija pridružena na adresi  0xb7fc1000
veličina segmenta:  25600
deljiva memorija ponovo pridružena na adresi  0x5000000
Hello, world.
```

Otklanjanje grešaka

Komanda `ipcs` pruža informacije o osobinama interprocesne komunikacije, uključujući i deljive segmente. Korišćenjem flega `–m` dobijaju se informacije o deljivoj memoriji.

Na primer, ovaj kôd ilustruje da je u upotrebi jedan deljivi memorijski segment sa brojem 1627649:

```
$ ipcs  -m
------------------ Shared Memory Segments ------------------
key          shmid     owner   perms   bytes   nattch   status
0x00000000   1627649   user    640     25600   0
```

Ako je ovaj memorijski segment greškom zaostao iza programa, moguće ga je ukloniti komandom ipcrm.

```
$ ipcrm shm 1627649
```

Za i protiv deljive memorije

Deljivi memorijski segmenti dozvoljavaju brzu dvosmernu komunikaciju između više procesa. Svaki korisnik može i čitati i upisivati, ali program mora uspostaviti i poštovati neke protokole kako bi sprečio trku, kao što je prepisivanje informacije pre čitanja. Na nesreću, Linux ne garantuje striktno ekskluzivan pristup, čak iako kreirate novi deljivi segment koristeći `IPC_PRIVATE`. Takođe, da bi više procesa koristili isti deljivi segment, potrebno je da se usaglase da koriste isti ključ.

Mapirana memorija

Mapirana memorija dozvoljava različitim procesima da komuniciraju preko deljene datoteke. Mada možda gledate na mapiranu memoriju kao na upotrebu deljivog memorijskog segmenta sa imenom, morate biti svesni da postoje tehničke razlike. Mapirana memorija može se koristiti za interprocesnu komunikaciju ili kao jednostavan način za pristup sadržaju datoteke.

Mapirana memorija formira vezu između datoteke i memorije procesa. Linux deli datoteke na delove koji su veličine stranice i onda ih kopira u stranice virtualne memorije kako bi ih učinio dostupnim u adresnom prostoru procesa. Zato, procesi mogu čitati sadržaj datoteke pomoću običnog memorijskog pristupa. Takođe, proces može menjati sadržaj datoteke upisujući u memoriju. Ovo omogućava brz pristup datotekama.

Na mapiranu memoriju možete gledati kao, prvo na alokaciju bafera da drži čitav sadržaj datoteke, onda na čitanje datoteke u bafer i na kraju upisivanje bafera nazad u datoteku, ako je došlo do promena. Linux rukuje operacijama čitanja i upisivanja za vas.

Mapiranje obične datoteke

Da bi se mapirala obična datoteka u memoriju procesa, koristite sistemski poziv `mmap` ("Memory MAPped," izgovara se "em-map"). Prvi argument je adresa na koju želite da Linux mapira datoteku u vašem procesnom adresnom prostoru. Vrednost NULL dozvoljava Linuxu da odabere pogodnu početnu adresu. Drugi argument je dužina mape u bajtovima. Treći argument određuje zaštitu mapiranog adresnog opsega. Ova zaštita se sastoji od bita "ili" simbola `PROT_READ`, `PROT_WRITE`, i `PROT_EXEC`, koje odgovaraju pravima za čitanje, pisanje i izvršavanje respektivno. Četvrti argument je vrednost flega koja određuje dodatne opcije. Peti argument je deskriptor datoteke koji opisuje datoteku koja treba da se mapira. Poslednji argument je pomeraj (offset) od početka datoteke odakle počinje mapiranje. Odgovarajućim izborom početnog pomeraja i odgovarajuće dužine, moguće je da mapirate celu datoteku ili deo datoteke u memoriji. Vrednost flega je jedna "ili" kombinacija od ovih konstanti:

- `MAP_FIXED` — Ako postavimo ovaj fleg, Linux koristi zatraženu adresu da mapira datoteku umesto da je sam određuje. Ova adresa mora biti stranično podešena (page-aligned).
- `MAP_PRIVATE` — Upisi u memorijski opseg se ne upisuju u mapiranu datoteku, nego u privatnu kopiju te datoteke. Nijedan drugi proces

ne vidi ove upise. Ovaj mod ne bi trebalo koristiti u kombinaciji sa `MAP_SHARED` flegom.

- `MAP_SHARED` — Upisi se odmah prosleđuju u mapiranu datoteku, a ne samo u bafer. Koristite ovaj mod kad koristite mapiranu memoriju za IPC. Ovaj mod ne bi trebalo koristiti sa flegom `MAP_PRIVATE`.

Ako poziv uspe, on vraća pokazivač na početak memorije. Pri neuspehu, vraća `MAP_FAILED`. Kada ste završili sa korišćenjem mapirane memorije, oslobodite je koristeći munmap. Pokrenite komandu navodeći početnu adresu i dužinu mapiranog memorijskog prostora. Linux automatski demapira mapirani prostor nakon završetka procesa.

Upis u memorijski mapiranu datoteku

Da bi ilustrovali korišćenje mapiranog memorijskog prostora osmotrimo dva programa za čitanje i upis u datoteku. Prvi program, `mmap-write.c` (listing 7.2), generiše slučajne brojeve i upisuje ih u memorijski-mapiranu datoteku. Drugi program `mmap-read.c` (listing 7.3), čita brojeve, štampa ih i zamenjuje ih u mapiranom memorijski-mapiranoj datoteci, sa njihovom dvostrukom vrednošću. Oba programa koriste argumente komandne linije kao ime datoteke koja se mapira.

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <time.h>
#include <unistd.h>
#define FILE_LENGTH 0x100

// Vraća ujednačene slučajne brojeve u opsegu [low,high].
int random_range (unsigned const low, unsigned const high)
{
  unsigned const range = high - low + 1;
  return low+(int) (((double) range)*rand()/(RAND_MAX+1.0));
}

int main  (int argc,  char* const argv[])
{
  int fd;
  void* file_memory;
```

```
  // Generator slučajnih brojeva
  srand  (time  (NULL));

  /* Priprema dovoljno veliku datoteku za prihvatanje
  neoznačenog celog broja */
  fd = open  (argv[1], O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
  lseek (fd, FILE_LENGTH+1, SEEK_SET);
  write (fd, "", 1);
  lseek (fd, 0, SEEK_SET);

  // Kreira mapiranu memoriju
  file_memory = mmap(0,FILE_LENGTH,PROT_WRITE,MAP_SHARED,fd,0);
  close (fd);

  // Upisuje slučajan ceo broj u mapirani memorijski prostor
  sprintf((char*) file_memory, "%d\n", random_range(-100, 100));

  // Oslobađa memoriju (nepotrebno jer se program završava)
  munmap (file_memory, FILE_LENGTH);

  return 0;
}
```

Listing 7.2 (mmap-write.c) Upis slučajnog broja u memorijski-mapiranu datoteku

Prevedite i pokrenite program:

```
$ gcc –o mmap-write mmap-write.c
$ ./mmap-write /tmp/integer-file
```

Program `mmap-write` otvara datoteku, kreirajući je, ako to prethodno nije urađeno. Treći argument za open određuje da je datoteka otvorena za čitanje i upisivanje. Obzirom da ne znamo dužinu datoteke, koristimo lseek da bi se uverili da je datoteka dovoljno velika, da smesti ceo broj i onda se se vraćamo na početnu poziciju u datoteci (`lseek`).

Program mapira datoteku u memoriju i zatim zatvara deskriptor datoteke zato što više nije potreban. Program zatim upisuje slučajan ceo broj u mapiranu memoriju, a time i u datoteku, a zatim demapira memoriju. Poziv munmap je nepotreban zato što će Linux automatski izvršiti unmap datoteke, nakon završetka programa.

Čitanje iz memorijski mapirane datoteke

Program mmap-read.c dat u listingu 7.3 čita integer iz mapirane memorijske datoteke i udvostručuje ga.

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
#define FILE_LENGTH 0x100

int main (int argc, char* const argv[])
{
  int fd;
  void* file_memory; int integer;

  // Otvaranje datoteke
  fd = open (argv[1], O_RDWR, S_IRUSR | S_IWUSR);

  // Stvaranje mapirane memorije
  file_memory = mmap (0, FILE_LENGTH, PROT_READ | PROT_WRITE,
    MAP_SHARED, fd,  0);
  close (fd);

  // Čitanje celobrojne vrednosti, štampanje i udvostručavanje
  sscanf (file_memory, "%d", &integer);
  printf ("value: %d\n", integer);
  sprintf ((char*) file_memory, "%d\n", 2*integer);

  // Oslobađanje memorije (nepotrebno jer se program inače
  // završava)
  munmap  (file_memory, FILE_LENGTH);
  return 0;
}
```

Listing 7.3 (mmap-read.c) Čitanje iz memorijski-mapirane datoteke

Prevedite program:

```
$ gcc –o mmap-read mmap-read.c
```

Program mmap-read čita broj iz datoteke i zatim upisuje u datoteku udvostručenu vrednost tog broja. Prvo, on otvara datoteku i mapira je u memoriju za čitanje i upisivanje. Obzirom da možemo predpostaviti da

je datoteka dovoljno velika da prihvati neoznačeni ceo broj (integer), nije potrebno da koristimo `lseek`, kao u prethodnom programu. Program čita vrednosti iz memorije tj. datoteke koristeći `sscanf` naredbu, a zatim ih formatira i upisuje dvostruke vrednosti koristeći `sprintf`.

Ovo je primer izvršenja ovog programa. On mapira datoteku `integer-file`.

```
$ ./mmap-write integer-file
$ cat integer-file
42
$ ./mmap-read integer-file
value: 42
$ cat integer-file
84
```

Primetite da je tekst 42 zapisan na disk bez pozivanja sistemskog poziva write, i ponovo je pročitan bez sistemskog poziva read. Zapazite da ovaj program-primer upisuje i čita ceo broj kao string (koristeći `sprintf` i `sscanf`) u demonstrativnu svrhu - nije neophodno da sadržaj memorijski mapirane datoteke bude tekst. Možete smestiti i ponovo uzeti proizvoljne binarne cifre iz memorijski mapirane datoteke.

Deljivi pristupi datotekama

Različiti procesi mogu komunicirati, koristeći mapirane memorijske oblasti povezane istom datotekom. Postavite fleg `MAP_SHARED`, tako da bilo koji upis u ove memorijske regione, bude odmah prenet u mapiranu datoteku i bude vidljiv drugim procesima. Ako ne postavite ovaj fleg, Linux može baferovati upise (cache), pre nego što ih prenese u datoteku.

Alternativno, možete prinuditi Linux da prebaci baferovane upise u datoteku na disku pozivajući msync. Njegova prva dva parametra određuju mapirani memorijski prostor, kao i kod munmap. Treći parameter je fleg sa sledećim vrednostima:

- `MS_ASYNC` — Ažuriranje je predviđeno, ali ne mora bezuslovno da se izvrši pre kraja poziva.
- `MS_SYNC` — Ažuriranje je trenutno; poziv `msync` se blokira dok se potpuno ne izvrši. Nije moguće zajedno koristiti `MS_SYNC` i `MS_ASYNC`.
- `MS_INVALIDATE` — Sva druga mapiranja datoteke se poništavaju, kako bi i ta mapiranja mogla videti ažurirane vrednosti.

Na primer: da bi ažurirali deljivu datoteku, mapiranu na adresi `mem_addr` i dužine `mem_length` bajtova, pozovimo sledeće:

```
msync (mem_addr, mem_length, MS_SYNC | MS_INVALIDATE);
```

Kao i kod deljivih memorijskih segmenata, korisnici mapiranih memorijskih oblasti moraju uspostaviti i pridržavati se protokola za izbegavanje stanja trke. Na primer, semafor se može koristiti da spreči da u jednom trenutku jednoj mapiranoj memoriji pristupa više od jednog procesa. Alternativno, možete koristiti `fcntl` da postavite zabranu čitanja ili upisivanja datoteka.

Privatno mapiranje

Postavljanjem `MAP_PRIVATE` kod poziva `mmap`, kreira se oblast sa osobinom "kopiraj pri upisu" (*copy-on-write*). Bilo koji upis u ovu oblast se reflektuje samo na ovu memoriju procesa, a drugi procesi koji mapiraju istu datoteku, neće videti promene. Umesto da upisuje direktno na stranicu koja je deljiva sa svim procesima, proces upisuje na privatnu kopiju svoje stranice. Svako naredno čitanje i pisanje ovog procesa, koristi ovu kopiranu stranicu.

Druge upotrebe mmap

Pored interprocesne komunikacije, poziv mmap se može koristiti i u druge svrhe. Jedna uobičajena upotreba je zamena za sistemske pozive read i write. Na primer, umesto da eksplicitno pročita sadržaj datoteke u memoriju, program će radije da mapira datoteku i skenira je koristeći memorijsko čitanje. Za neke programe ovo je pogodnije i takođe se može izvršiti brže nego eksplicitne U/I operacije datoteka.

Jedna napredna i moćna tehnika, koju koriste neki programi je kreiranje strukture podataka (npr. obične strukture) u memorijski mapiranoj datoteci. Pri kasnijem pozivu, program ponovo mapira tu datoteku u memoriju, a strukture podataka se vraćaju u svoje prethodno stanje. Zapazite, da će pokazivači na ove strukture podataka biti pogrešni, izuzev ako svi ne pokazuju u granicama istog memorijski mapiranog prostora i ako se program ne pobrine da se datoteka ponovo mapira u isti adresni prostor koji je ranije zauzimala.

Druga pogodna tehnika je mapiranje specijalne `/dev/zero` datoteke u memoriju. Ova datoteka, se ponaša kao beskonačno duga datoteka popunjena sa nulama. Program kome je potreban izvor nula, može koristiti mmap na datoteku `/dev/zero`. Upisivanja u `/dev/zero` se odbacuju, tako da se mapirana memorija može koristiti u bilo koju svrhu. Uobičajeni memorijski alokatori često mapiraju `/dev/zero` da bi dobili delove preinicijalizovane memorije, popunjene nulama.

8. IPC: pipe i FIFO

Pipe je komunikacioni uređaj koji dozvoljava jednosmernu komunikaciju. Podatak upisan na upisni kraj pipe datoteke ("*write end*"), čita se sa kraja za čitanje pipe datoteke ("*read end*"). Pipe datoteke su serijski uređaji, podaci se uvek čitaju iz pipe datoteke u istom redosledu kako su upisani. Obično se pipe koristi za komunikaciju između dve niti u jednom procesu ili između procesa roditelja i procesa deteta.

Neimenovani pipe

U shell programu, simbol | kreira pipe. Na primer ova shell komanda prouzrokuje da shell stvori dva procesa deteta, jedna za ls i jedan za less:

```
$ ls | less
```

Shell takođe stvara pipe datoteku, spajajući standardni izlaz ls procesa, sa standardnim ulazom less procesa. Imena datoteka izlistana pomoću ls se šalju ka `less` u istom onom poretku, u kojem bi bili poslani direktno ka terminalu.

Kapacitet podataka pipe datoteke je ograničen. Ako proces pisac piše brže nego što proces čitaoc može da pročita podatke, i ako pipe ne može da sačuva višak podataka, proces pisac se blokira dok veći kapacitet ne postane dostupan. Ako proces čitaoc pokuša da pročita, a ne postoje raspoloživii podaci, on se blokira sve dok podaci ne postanu dostupni. Zato, pipe automacki sinhronizuje dva procesa.

Kreiranje pipe datoteka

Da bi kreirali pipe, moramo koristiti sistemski poziv `pipe`. Sistemski poziv `pipe` zahteva na ulazu celobrojni niz veličine 2. Poziv pipe smešta deskriptor čitanja datoteke na poziciju 0 u nizu, a deskriptor upisa datoteke na poziciju 1. Na primer, osmotrimo sledeći kôd:

```
int pipe_fds[2];
int read_fd;
int write_fd;
pipe (pipe_fds);
read_fd = pipe_fds[0];
write_fd = pipe_fds[1];
```

Podaci upisani preko write deskriptora `write_fd` se mogu pročitati pomoću read deskriptora `read_fd`.

Komunikacija između procesa roditelja i procesa deteta

Poziv pipe kreira dva deskriptora datoteke, koji su validni samo za taj procesa i za njegovu decu. Deskriptori datoteka procesa ne mogu se deliti između dva nepovezana procesa. Kada proces pozove `fork`, deskriptori datoteka se kopiraju na novi proces dete. Zato, `pipe` može spajati jedino povezane procese.

U programu `pipe.c` (listing 8.1) `fork` stvara proces dete. Dete nasleđuje deskriptore pipe datoteke. Roditelj upisuje string u pipe, a proces dete ga čita. Jednostavan program konvertuje ove deskriptore datoteke u `FILE*` stream strukture, koristeći `fdopen`. Obzirom da radije koristimo stream strukture neko deskriptore datoteke, možemo koristiti ulazno/izlazne funkcije višeg reda iz standardne C biblioteke kao što su `printf` i `fgets`.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

/* Upisuje COUNT kopije iz PORUKE u STREAM, između svake
praveći pauzu od jedne sekunde. */
void writer (const char* message, int count, FILE* stream)
{
  for (; count > 0; --count)
  {
    // Upisuje poruku u niz, i odmah ih šalje dalje...
    fprintf (stream, "%s\n", message);
    fflush (stream);

    sleep (1); // Čeka malo
    }
}

// Čita proizvoljne stringove iz niza što je duže moguće.
void reader (FILE* stream)
{
  char buffer[1024];
  /* Čita dok ne stignemo do kraja niza. fgets čita do
  nove linije ili do kraja.  */
  while(!feof (stream) && !ferror (stream) && fgets (buffer,
        sizeof (buffer), stream) != NULL)
   fputs (buffer, stdout);
}
```

```
int main ()
{
  int fds[2];
  pid_t pid;

  // Kreira pipe. Fajl deskriptori oba kraja se smeštaju u fds
  pipe(fds);

  // Stvara se proces dete.
  pid = fork ();
  if (pid == (pid_t) 0)
  {
    FILE* stream;
    /* Ovo je proces dete. Zatvara našu kopiju kraja za upis
    fajl deskriptora */
    close (fds[1]);
    /* Konvertuje file-deskriptor za čitanje u objekat
    tipa FILE, i čita iz njega. */
    stream = fdopen  (fds[0],   "r");
    reader (stream);
    close (fds[0]);
  }
  else
  {
    // Ovo je proces roditelj.
    FILE* stream;
    // Zatvara našu kopiju kraja za čitanje datoteka deskripora
    close (fds[0]);
    /* Konvertuje file-descriptor upisa u datoteka object
    i upisuje u njega */
    stream = fdopen (fds[1], "w");
    writer ("Hello, world.", 5, stream);
    close (fds[1]);
  }

return 0;
}
```

Listing 8.1 (pipe.c) Korišćenje pipe za komunikaciju sa procesom detetom

Prevedite i izvršite program pipe.c i objasnite izlazne rezultate.

```
$ gcc –o pipe pipe.c
$ ./pipe
```

Na početku `main` funkcije, `fds` se deklariše kao celobrojni niz veličine 2. Pipe poziv kreira pipe datoteku i smešta deskriptore za upis i čitanje u taj niz. Program tada preko poziva `fork`, kreira proces dete. Nakon zatvaranja kraja za čitanje pipe datoteke, proces roditelj počinje da upisuje podatke u pipe datoteku. Nakon zatvaranja kraja za upis pipe, dete čita podatke iz pipe datoteke.

Zapazite da nakon upisivanja u writer-funkciji, roditelj obavlja `flush` funkciju, kako bi sravnio stanje iz keš memorije u pipe datoteku. U suprotnom, može se desiti da string ne bude poslat odmah kroz pipe.

Kada pozivate komandu `ls | less`, dešavaju se dva poziva `fork`: jedan za `ls` dete proces i jedan `less` dete proces. Oba ova procesa nasleđuju pipe file-deskriptore kako bi mogli komunicirati koristeći pipe datoteku. Ako želite da ostvarite komunikaciju između nepovezanih procesa koristite FIFO, a što je opisano u odeljku, "FIFO."

Preusmeravanje standardnog ulaza, izlaza i izlaza za greške

Često ćete želeti da kreirate proces dete i da postavite jedan kraj pipe kao njegov standardni ulaz ili standardni izlaz. Korišćenjem poziva `dup2`, možete izjednačiti jedan file-deskriptor sa drugim. Na primer, da bi se preusmerio standardni ulaz procesa na file-deskriptor `fd`, koristite sledeću liniju:

```
dup2 (fd, STDIN_FILENO);
```

Simbolička konstanta `STDIN_FILENO` predstavlja file-deskriptor za standardni ulaz, koji ima vrednost 0. Poziv zatvara standardni ulaz i ponovo ga otvara kao duplikat fd-a kako bi se mogao koristiti za razmenjivanje. Izjednačeni file-deskriptori dele istu poziciju datoteka i iste file-status flegove. Zato se karakteri pročitani pomoću fd, ne čitaju ponovo (reread) kroz standardni ulaz.

Primer preusmeravanja izlaza

Program `dup2.c` (listing 8.2) koristi sistemski poziv `dup2`, da pošalje sadržaj iz pipe datoteke ka komandi za sortiranje. Nakon kreiranja pipe, program obavlja `fork`. Proces roditelj ispisuje podatke kroz pipe datoteku. Proces dete pridružuje file-deskriptor za čitanje pipe na svoj standardni ulaz koristeći `dup2`; tada izvršava program za sortiranje.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main ()
{
  int fds[2]; pid_t pid;
  pipe(fds); // Kreira pipe
  pid = fork (); // Stvara se proces dete.
  if (pid == (pid_t) 0)
  { // Ovo je proces dete.
    // Zatvara se naša kopija kraja za upis file-deskriptora
    close (fds[1]);

    // Povezuje se kraj za čitanje pipe na standardni ulaz.
    dup2 (fds[0], STDIN_FILENO);

    // Zamenjuje se proces dete sa programom za sortiranje.
    execlp ("sort", "sort", 0);
  }
  else
  { // Ovo je proces roditelj.
    FILE* stream;
    // Zatvara se naša kopija kraja za čitanje file-deskriptora
    close (fds[0]);
    /* Kovertuje se datoteka deskriptor za upis u datoteka
    object, i upisuje se u njega. */
    stream = fdopen (fds[1], "w");
    fprintf (stream, "This is a test.\n");
    fprintf (stream, "Hello, world.\n");
    fprintf (stream, "My dog has fleas.\n");
    fprintf (stream, "This program is great.\n");
    fprintf (stream, "One fish, two fish.\n");
    fflush (stream);
    close (fds[1]);
    waitpid (pid, NULL, 0); // Čeka se da proces dete završi
  }
return 0;
}
```

Listing 8.2 (dup2.c) Preusmeravanje izlaza pipe pomoću dup2

Prevedite i izvršite program pipe.c i objasnite izlazne rezultate.

```
$ gcc –o dup2 dup2.c
$ ./dup2
```

Proces dete sortira sadžaj pipe datoteke, koju puni proces roditelj sa stingovima:

```
This is a test.
Hello, world.
My dog has fleas.
This program is great.
One fish, two fish.
```

popen i pclose

Uobičajena upotreba pipe datoteke je slanje ili primanje podataka od programa, koji se izvšrava u drugom procesu. Funkcije `popen` i `pcclose` olakšavaju ovu upotrebu (paradigmu) eliminišući potrebu za upotrebom `pipe`, `fork`, `dup2`, `exec` i `fdopen`. Uporedite program `popen.c` (listing 8.3), koji koristi `popen` i `pclose`, sa prethodnim primerom `dup2.c` (listing 8.2).

```
#include <stdio.h>
#include <unistd.h>

int main ()
{
  FILE* stream = popen ("sort", "w");
  fprintf (stream, "This is a test.\n");
  fprintf (stream, "Hello, world.\n");
  fprintf (stream, "My dog has fleas.\n");
  fprintf (stream, "This program is great.\n");
  fprintf (stream, "One fish, two fish.\n");
  return pclose (stream);
}
```

Listing 8.3 (popen.c) Primer upotrebe popen

Poziv popen kreira proces dete koje izvršava komandu za sortiranje, zamenjujući pozive `pipe`, `fork`, `dup2`, i `execlp`. Drugi argument `"w"`, ukazuje da ovaj proces hoće da upisuje u proces dete. Povratna vrednost poziva `popen` je jedan kraj pipe datoteke, a drugi kraj je povezan na standardni ulaz procesa deteta. Nakon završetka upisivanja, pclose zatvara stream strukturu procesa deteta, čeka da se proces završi i vraća njegovu statusnu vrednost.

Prvi argument od poziva `popen` se izvršava kao shell komanda u procesu koga pokreće `/bin/sh`. Shell koristi `PATH` variablu na uobičajeni način kako bi pronašao programe da bi ih izvršio. Ako je drugi argument `"r"`, funkcija vraća procesu detetu standardni stream izlaz, kako bi roditelj mogao pročitati taj izlaz. Ako je drugi argument `"w"`, funkcija vraća procesu standardni stream ulaz, kako bi roditelj mogao da pošalje podatke. Ako se dogodi greška, popen vraća null (nulti) pokazivač.

Pozovite pclose, da bi se zatvorila stream struktura, vraćena pomoću popen. Nakon zatvaranja određene stream strukture, `pclose` čeka na završetak procesa deteta.

FIFO

FIFO datoteka je pipe koji ima ime u sistemu datoteka. Bilo koji proces može otvoriti ili zatvoriti FIFO, procesi ne moraju biti srodni jedan sa drugim. FIFO se takođe zove i imenovani pipe.

Možete kreirati FIFO koristeći mkfifo komandu. Odredite putanju do FIFO datoteke na komandnoj liniji. Na primer, kreirajte FIFO u `/tmp/fifo` koristeći sledeće:

```
% mkfifo /tmp/fifo
% ls -l /tmp/fifo
prw-rw-rw-  1  stojko  users  0  Jan 16 14:04  /tmp/fifo
```

Prvi karakter izlaza iz `ls` je `p`, koji ukazuje da je datoteka ustvari FIFO (imenovani pipe).

U jednom prozoru čitajte iz FIFO koristeći:

```
$ cat < /tmp/fifo
```

U drugom prozoru, upisujte u FIFO koristeći:

```
$ cat > /tmp/fifo
```

Zatim ukucajte nekoliko linija teksta. Svaki put kada pritisnete Enter, linija teksta se šalje kroz FIFO i pojavljuje se u prvom prozoru. Zatvorite FIFO pritiskanjem `Ctrl+D` u drugom prozoru. Uklonite FIFO komandom

```
$ rm /tmp/fifo
```

Kreiranje FIFO

Kreirajte FIFO programski koristeći `mkfifo` funkciju. Prvi argument je putanja na kojoj se kreira FIFO. Drugi parametar određuje prava pristupa za pipe datoteku (za vlasnika, grupu, ostale). 0bzirom da pipe

mora posedovati i čitača (reader) i upisivača (writer), dozvole moraju uključivati i pravo čitanja i pravo upisivanja.

Ako pipe ne može biti kreiran (na primer, ako datoteka sa tim imenom već postoji), `mkfifo` vraća —1.

Uključite `<sys/types.h>` i `<sys/stat.h>` ako koristite `mkfifo`.

Pristup FIFO datoteci

Pristup FIFO datoteci je isti kao i običnoj datoteci. Za komunikaciju kroz FIFO, potrebno je da ga jedan program otvori za upis, a drugi da ga otvori za čitanje. Koriste se U/I funkcije nižeg reda (`open`, `write`, `read`, `close`,) ili U/I funkcije iz C biblioteke (`fopen`, `fprintf`, `fscanf`, `fclose`, itd.).

Na primer, za upis bafera podataka u FIFO datoteku, koristeći U/I rutine nižeg reda, možete koristiti sledeći k**ô**d:

```
int fd = open (fifo_path, O_WRONLY);
write (fd, data, data_length);
close (fd);
```

Za čitanje podataka iz FIFO datoteke, koristeći U/I funkcije C biblioteke, možete koristiti ovaj k**ô**d:

```
FILE* fifo = fopen (fifo_path, "r");
fscanf (fifo, "%s", buffer);
fclose (fifo);
```

FIFO može imati više čitača (readers) ili upisivača (writers). Bajtovi sa svakog upisivača se atomski-nedeljivo upisuju do maksimalne veličine `PIPE_BUF` (4KB na Linux-u). Delovi sa simultanih upisivača se mogu preplitati. Slična pravila se mogu primeniti i na simultane čitače.

Razlike sa Windows imenovanim pipe datoteka

Pipe datoteke u Win32 operativnim sistemima su veomo slične Linux pipe datotekama (pogledati Win32 dokumentaciju za tehničke detalje). Glavna razlika kod imenovanih pipe datoteka je u tome, što kod Win32 sistema oni više funkcionišu kao soketi. Win32 imenovani pipe mogu povezati procese na različitim računarima povezanim preko mreže. Na Linux-u u ovu svrhu se koriste soketi. Takođe, Win32 dozvoljava više konekcija tipa čitač-upisivač na imenovanoj pipe datoteci bez preplitanja, a pipe datoteke se mogu koristiti za dvosmernu komunikaciju.

9. IPC: soketi

Soket (priključnica) je bidirekcioni uređaj koji se može koristiti za komunikaciju sa drugim procesom na istoj mašini ili sa procesom na drugim mašinama. Soketi su jedina interprocesna komunikacija koju ćemo osmatrati u ovoj knjizi koja dozvoljava komunikaciju između procesa na različitim računarima. Internet programi kao što su Telnet, rlogin, FTP, talk, i World Wide Web koriste sokete.

Na primer, možete dobiti WWW stranicu sa Web servera, koristeći Telnet program, zato što oba procesa koriste sokete za mrežnu komunikaciju. Da bi otvorili konekciju ka WWW serveru na www.mumbojumbo.com, koristite telnet www.mumbojumbo.com 80. Broj porta 80 određuje konekciju ka Web serveru i programsko pokretanje www.mumbojumbo.com umesto nekog drugog procesa. Pokušajte da ukucate GET / nakon što je konekcija uspostavljena. Ova komanda šalje poruku kroz soket ka Web serveru, koji odgovara šaljući HTML kôd za home page (početne strane) i zatim zatvara konekciju, na primer:

```
$ telnet www.mumbojumbo.com 80
Trying 166.61.6.1...
Connected to jimmyjones.mumbojumbo.com (166.61.6.1). Escape
character is  '"]'.
GET /
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1 ">
..........
```

Obično ćete koristiti telnet da povežete Telnet server sa udaljenim pristupima (logins). Ali telnet možete koristiti takođe za konekciju sa serverom druge vrste i onda ukucavati komentare direktno na njega.

Koncept soketa

Kada kreirate soket, morate odrediti tri parametra:

- stil komunikacije
- namespace (prostor imena)
- protokol.

Tip ili stil komunikacije kontroliše kako soketi tretiraju poslate podatke i određuje broj partnera za komunikaciju. Kada je podatak poslat kroz soket, on se grupiše u delove koji se nazivaju paketi. Stil

komunikacije određuje kako se postupa sa ovim paketima i kako se oni adresiraju od pošiljaoca do primaoca.

- Konekcijski stil garantuje isporuku svih paketa u poretku kako su i poslani. Ako su paketi izgubljeni ili je promenjen njihov redosled usled problema u mreži, primalac automatski zahteva od pošiljaoca njihovo ponovno slanje. Konekcijski stil soket je kao telefonski poziv: Adrese pošiljaoca i primaoca su fiksne na početku komunikacije i tokom konekcije.
- Datagram stil ne garantuje poredak paketa za isporuku ili prijem. Paketi mogu biti izgubljeni ili preorganizovani tokom transporta kroz mrežu u zavisnosti od mrežnih grešaka ili drugih uslova. Svaki paket mora biti obeležen zajedno sa svojom destinacijom i nije garantovano da će biti isporučen. Sistem garantuje samo "najbolji trud", tako da paketi mogu nestati ili biti preorganizovani tokom transporta.

Soketi sa datagram stilom se ponašaju više kao poštanska pisma. Pošiljalac specifira adresu primaoca za svaku poruku pojedinačno.

Namespace soketa određuje kako se piše adresa soketa. Adresa soketa identifikuje jedan kraj soket konekcije. Na primer, soket adrese u "lokalnom prostoru imena" su obična imena datoteka. U "Internet prostoru imena" soket adrese se sastavljaju od Internet adrese (takođe poznate kao Internet protokol adrese - Internet Protocol address ili IP adrese-IP address) hosta prikačenog na mrežu i broja porta. Port brojevi su različiti za svaki soket na istom hostu.

Protokol određuje kako se prenose podaci. Neki protokoli su

- TCP/IP, primarni protokol koji se koristi kod Interneta
- AppleTalk mrežni protokol
- UNIX lokalni komunikacioni protokol

Nisu podržane sve kombinacije stilova, imena prostora i protokola.

Sistemski pozivi

Soketi su fleksibilnijji od prethodno osmotrenih komunikacionih tehnika. Ovo su sistemski pozivi koji su vezani za sokete:

- `socket` - kreira soket
- `close` - uništava soket
- `connect` - kreira konekciju između dva soketa
- `bind` - obeležava serverski soket sa adresom

- `listen` - konfiguriše soket da prihvati uslove
- `accept` - prihvata konekciju i kreira novi soket za konekciju

Soketi se predstavljaju pomoću file-deskriptora.

Kreiranje i uništavanje soketa

Funkcije `socket` i `close` kreiraju i uništavaju sokete, respektivno. Kada kreirate sokete specifirajte tri soket parametra: namespace, komunikacioni stil i protokol.

Za namespace parametar koristite konstante koje počinju sa `PF_` (skraćenica "familija protokola"). Na primer,

- `PF_LOCAL` ili `PF_UNIX` određuje lokalni namespace
- `PF_INET` određuje Internet namespace.

Za komunikacioni stil, koristite konstante koje počinju sa `SOCK_`. Koristite

- `SOCK_STREAM` soket sa konekcijskim stilom
- `SOCK_DGRAM` za soket sa datagram stilom

Treći parametar, protokol, određuje mehanizam nižeg reda za slanje i primanje podataka. Svaki protokol je odgovarajući za određenu kombinaciju namespace-stil. Obzirom da obično postoji jedan najbolji protokol za svaki takav par, postavljanjem 0 obično se dobija dobar protokol.

Ako je `soket` poziv bio uspešan, on vraća file-deskriptor za soket. Možete čitati, upisivati u soket isto kao i kod drugih file-deskriptora.

Kada ste završili sa soketom, pokrenite `close` kako biste ga uklonili.

Sistemski poziv connect

Da bi kreirali konekciju između dva soketa, klijent soket poziva connect, specifirajući adresu serverskog soketa na koji se konektuje. Klijent je proces koji pokreće konekciju, a server je proces koji čeka da prihvati konekcije. Klijent proces poziva connect kako bi odpočeo konekciju od lokalnog soketa do soketa servera, specifiranog drugim argumentom. Treći argument je dužina u bajtovima adresne strukture, na koju se ukazuje drugim argumentom. Formati adresa soketa se razlikuju u zavisnosti od namespace soketa.

Slanje informacija

Bilo koja tehnika za upis u deskriptor datoteke može se koristiti da bi se upisivalo u soket. Pogledajte diskusiju o Linux-ovim U/I funkcijama nižeg reda i o nekim posledicama njihove upotrebe.

Funkcija `send`, koja je posebna za soket deskriptore datoteka, omogućava alternativan upis sa nekoliko dodatnih mogućnosti, a za informacije o komandi `send`, pogledajte man stranu.

Serveri

Životni ciklus servera sastoji se od kreiranja soketa konekcijskog stila, dodeljivanja adrese soketu, postavljanje poziva za osluškivanje (`listen`), koji omogućava konekciju sa soketima, postavljanje poziva za prihvatanje dolazne konekcije, i na kraju zatvaranje soketa. Podaci se ne čitaju ili upisuju direktno kroz server soket, već svaki put kada program prihvati novu konekciju, Linux kreira poseban soket koji se koristi za transfer podataka preko te konekcije. U ovom odeljku, upoznaćemo se sa `bind`, `listen` i `accept` pozivima.

Sistemski poziv bind

Adresa mora biti dodeljena serverskom soketu, korišćenjem `bind`. Prvi argument je soket file- deskriptor. Drugi argument je pokazivač na soket adresnu strukturu, a format zavisi od familije soket adrese. Treći argument je dužina adresne strukture u bajtima.

Sistemski poziv listen

Kada je adresa dodeljena soketu sa konekcijskim stilom, on mora pokrenuti osluškivanje kako bi ukazivao da je server. Prvi argument je soket file-deskriptor. Drugi argument određuje koliko nerešenih konekcija stoji u redu za čekanje. Ako je red za čekanje pun, dodatne konekcije će biti odbijene. Ovo ne ograničava ukupan broj konekcija koje server može da opsluži; ovo ograničava samo broj konekcija koje pokušavaju da se konektuju, a koje još nisu prihvaćene.

Sistemski poziv accept

Server prihvata konekciju od klijenta pokretanjem poziva `accept`. Prvi argument je soket file-deskriptor. Drugi argument ukazuje na soket adresnu strukturu, koja se popunjava sa soket adresom klijenta. Treći argument je dužina u bajtima soket adresne strukture. Server može koristiti adresu klijenta da bi odlučio da li stvarno želi da komunicira sa

klijentom. Poziv accept, kreira novi soket, za komunikaciju sa klijentom i vraća se novi, odgovarajući file-deskriptor. Originalni server soket nastavlja da prihvata nove klijentske konekcije.

Da bi pročitali podatak sa soketa, ali bez uklanjanja podataka iz reda za čekanje, koristite poziv recv. On koristi iste argumente kao read, plus dodatni FLAG argument. Fleg `MSG_PEEK` prouzrokuje da podatak bude pročitan, ali ne i uklonjen sa ulaznog reda za čekanje (queue).

Lokalni soketi

Soketi koji spajaju procese na istom računaru mogu koristiti lokalni namespace predstavljen sinonimima `PF_LOCAL` i `PF_UNIX`. Oni se zovu lokalni soketi ili UNIX-domain soketi. Njihove soket adrese, određene imenom datoteka, se koriste samo kada se kreiraju konekcije. Soket ime je određeno u strukturi sockaddr_un. Potrebno je postaviti sun_family polje na `AF_LOCAL`, ukazujući time da je ovo lokalni namespace. Polje sun_path određuje ime datoteka koji se koristi, a može biti dugo najviše 108 bajtova. Stvarna dužina strukture sockaddr_un trebalo bi da se obračuna korišćenjem makroa `SUN_LEN`.

Može se koristiti bilo koje datoteka ime, ali proces mora imati prava upisa nad direktorijumom, koje dozvoljava dodavanje datoteka u direktorijum. Da bi se konektovao na soket, proces mora imati pravo čitanja soket datoteke. I pored toga što različiti računari mogu deliti isti datoteka sistem, samo procesi koji se izvršavaju na istom računaru mogu komunicirati preko lokalnih namespace soketa.

Jedini dopustljiv protokol za lokalni namespace je 0.

Pošto se nalazi u datoteka sistemu, lokalni soket je prikazan kao datoteka.

Na primer, zapazite inicijal `s`:

```
$ ls  -l /tmp/socket
srwxrwx--x  1  user  group  0 Nov 13 19:18  /tmp/socket
```

Pozovite `unlink` za uklanjanja lokalnog soketa kada ste završili sa njim.

Primer korišćenja lokalnog namespace soketa

Ilustrovaćemo sokete sa dva programa: `socket-server.c` (listing 9.1) i `socket-client.c` (listing 9.2). Server program u `socket-server.c`, kreira lokalni namespace soket i čeka na konekciju preko njega. Kada primi konekciju, on čita tekstualnu poruke sa konekcije i ispisuje ih sve do zatvaranja konekcije. Ako je jedna od ovih poruka "quit" serverski

program uklanja soket i završava. Server soket program uzima putanju do soketa, kao argument sa komandne linije.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

/* Čita poruke sa soketa i prikazuje ih. Nastavlja do zatvaranja
soketa. Vraća nonzero ako klijent pošalje "quit" poruku, a nulu
u drugom slučaju. */
int server (int client_socket)
{
  while (1)
  {
    int length;
    char* text;

    /* Prvo, čita dužinu tekst poruke sa soketa. Ako čitanje
    vrati nulu, klijent zatvara konekciju. */

    if (read (client_socket, &length, sizeof (length)) == 0)
      return 0;

    // Alocira se bafer da bi prihvatio poruku.
    text = (char*) malloc (length);

    // Čita sam tekst, i prikazuje ga.
    read (client_socket, text, length); printf ("%s\n", text);

    // Oslobađa bafer.
    free (text);

    // Ako klijent pošalje poruku "quit," sve je završeno.
    if (!strcmp (text, "quit")) return 1;
  }
}

int main  (int argc,  char* const argv[])
{
  const char* const socket_name = argv[1];
  int socket_fd;
  struct sockaddr_un name;
```

```
  int client_sent_quit_message;

  // Kreira soket.
  socket_fd = socket   (PF_LOCAL,  SOCK_STREAM,  0);

  // Ukazuje da je ovo server.
  name.sun_family = AF_LOCAL;

  strcpy (name.sun_path, socket_name);
  bind (socket_fd, &name, SUN_LEN (&name));

  // Čeka – osluškuje konekcije.
  listen (socket_fd, 5);

  /* Ponavlja prihvatanje konekcija, čineći da jedan server
  radi sa svakim klijentom. Nastavlja dok klijent ne pošalje
  "quit" poruku. */
  do
  {
    struct sockaddr_un client_name;
    socklen_t client_name_len;
    int client_socket_fd;

    // Prihvata konekciju.
    client_socket_fd = accept (socket_fd, &client_name,
      &client_name_len);

    // Rukuje sa konekcijom.
    client_sent_quit_message = server (client_socket_fd);

    // Zatvara naš kraj konekcije.
    close (client_socket_fd);
  } while (!client_sent_quit_message);

  // Uklanja soket datoteka.
  close (socket_fd);
  unlink (socket_name);

return 0;
}
```

Listing 9.1 (socket-server.c) Local namespace soket server

Klijentski program `socket-client.c` (listing 9.2), konektuje se na lokal namespace soket i šalje poruku. Putanja do soketa i poruka su određeni na komandnoj liniji.

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

// Upisuje u soket TEXT dat putem deskriptora SOCKET_FD
void write_text (int socket_fd, const char* text)
{
  // Upisuje broj bajtova u string, uključujući NULL završetak
  int length = strlen (text) + 1;
  write (socket_fd, &length, sizeof (length));

  // Upisuje string
  write (socket_fd, text, length);
}

int main (int argc, char* const argv[])
{
  const char* const socket_name = argv[1];
  const char* const message = argv[2];
  int socket_fd; struct sockaddr_un name;

  // Kreira soket
  socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);

  // Čuva ime servera u adresi soketa
  name.sun_family = AF_LOCAL;
  strcpy (name.sun_path, socket_name);

  // Konektuje soket
  connect (socket_fd, &name, SUN_LEN (&name));

  // Upisuje tekst sa komandne linije u soket
  write_text (socket_fd, message);

  close (socket_fd);
  return 0;
}
```

Listing 9.2 (socket-client.c) Local namespace soket klijent

Pre nego što klijent pošalje tekst poruke, on šalje dužinu teksta u bajtovima (varijabla `length`). Takođe, server prvo čita dužinu teksta poruke iz soketa. Ovo omogućava serveru da alocira odgovarajuću veličinu bafera za prihvatanje tekst poruke, pre čitanja sa soketa.

Prevedite i izvršite program `socket-server.c` i `socket-client.c` i objasnite izlazne rezultate. Da bi probali ovaj primer, pokrenite server program u jednom prozoru. Odredite putanju do soketa – na primer, `socket`.

```
$ ./socket-server socket
```

U drugom prozoru, pokrenite klijenta nekoliko puta, određujući istu soket putanju, i poruke da ih pošaljete klijentu:

```
$ ./socket-client socket "Hello, world."
$ ./socket-client socket "This is a test."
```

Program server prihvata i ispisuje ove poruke. Da bi zatvorili server, pošaljite poruku `"quit"` sa klijenta:

```
$ ./socket-client socket "quit"
Program server se zatvara.
```

Internet-domain soketi

UNIX-domain soketi mogu se koristiti samo za komunikaciju između dva procesa na istom računaru. Za razliku od prethodno navedenih, Internet-domain soketi, se mogu koristiti za konekciju procesa na različitim mašinama, povezanih mrežom.

Soketi koji povezuju procese preko interneta, koriste Internet namespace, koji se predstavlja pomoću `PF_INET`. Najčećši protokoli za njega su TCP/IP. Internet protokol (IP), protokol nižeg reda, prenosi pakete kroz Internet, deleći ih i ponovo ih spajajući ako je to potrebno. On garantuje samo "best-effort" (najbolji trud) isporuku, tako da paketi mogu nestati ili biti reorganizovani tokom transporta. Svaki računar učesnik je određen korišćenjem IP broja. Transmission Control Protocol (TCP-protokol kontrole prenosa), oslanjajući se na IP, pruža pouzdan i uredan transport. On dozvoljava uspostavu telefonske konekcije između dva računara i osigurava da će prenos podataka biti siguran i uredan.

DNS imena

Obzirom da je lakše zapamtiti imena nego brojeve, Domain Name Service (DNS) povezuje imena kao što je `www.mumbojumbo.com` sa jedinstvenim kompjuterskim IP adresom. DNS je implementiran pomoću svetske hijararhije imena servera, s tim što vi ne morate da razumete DNS protokol, kako bi koristili host imena u svojim programima.

Internet soket adrese sadrže dva dela: broj mašine-IP i broj porta. Ove informacije se čuvaju u strukturi `sockaddr_in` promenljive. Postavite sin_family polje na `AF_INET` da bi ukazali da je ovo Internet namespace

adresa. Polje `sin_addr`, čuva Internet adresu željene mašine u obliku 32-bitnog IP broja. Port pomaže da se razlikuju različiti soketi koji su dodeljeni računaru. Obzirom da različite mašine čuvaju višebajtne vrednosti u različitom bajt poretku, koristite `htons` da bi konvertovali port broj u mrežni bajt poredak (*network byte order*). Pogledajte ip man stranu za više informacija.

Da bi konvertovali ljudima čitljiva host imena, ili brojeve u standardnom dot-označavanju (kao što je 10.0.0.1) ili DNS imena (kao što je `www.mumbojumbo.com`) u 32-bitne IP brojeve, možete koristiti funkciju `gethostbyname`, koja vraća pokazivač na stukturu `hostent` structure; polje `h_addr field` sadrži host IP broj.

Primer upotrebe Internet-domain soketa

Program `socket-inet.c` (listing 9.3) ilustruje upotrebu Internet-domain soketa. Program dobija home page (početnu stranu) sa Web servera čije je host ime određeno na komandnoj liniji.

```
#include <stdlib.h>
#include <stdio.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/socket.h>
#include <unistd.h>
#include <string.h>

/* Prikazuje sadržaj home page-a sa server soketa. Vraća
indikator uspeha. */
void get_home_page (int socket_fd)
{
  char buffer[10000];
  ssize_t number_characters_read;

  // Šalje HTTP GET komandu za home page
  sprintf (buffer, "GET /\n");
  write (socket_fd, buffer, strlen (buffer));

  /* Čita sa soketa. Poziv za čitanje možda ne može vratiti
  sve podatke u jednom trenutku, tako da nastavlja da pokušava
  dok ga ne prekinemo. */
  while (1)
  {
    number_characters_read = read (socket_fd, buffer, 10000);
```

```
    if (number_characters_read == 0) return;

    // Ispisuje podatke na standardni izlaz
    fwrite (buffer,sizeof(char),number_characters_read,stdout);
  }
}

int main (int argc, char* const argv[])
{
  int socket_fd;
  struct sockaddr_in name;
  struct hostent* hostinfo;

  // Kreira soket.
  socket_fd = socket (PF_INET, SOCK_STREAM, 0);

  // Čuva ime servera u adresu soketa.
  name.sin_family = AF_INET;

  // Konvertuje iz stringa u brojeve.
  hostinfo = gethostbyname (argv[1]);
  if (hostinfo == NULL) return 1;
  else
  name.sin_addr = *((struct in_addr *) hostinfo->h_addr);

  // Web serveri koriste port 80.
  name.sin_port = htons (80);

  // Konektuje se na Web server
  if (connect(socket_fd,&name,sizeof(struct sockaddr_in))== -1)
  {
    perror ("connect");
    return 1;
  }
  // Traži server home page (početnu stranu)
  get_home_page (socket_fd);
  return 0;
}
```

Listing 9.3 (socket-inet.c) Čitanje sa WWW servera

Ovaj program uzima hostname Web servera sa komandne linije (ne URL, tj. bez "http://"). On poziva gethostbyname da bi preveo ime hosta u numeričku IP adresu i da bi zatim konektovao stream (TCP) soket na port broj 80 za taj računar. Web serveri koriste Hypertext Transport

Protocol (HTTP), tako da program izdaje `HTTP GET` komandu i server odgovara šaljući tekst svoje home stranice.

Prevedite i izvršite program `socket-inet.c` i objasnite izlazne rezultate.

```
$ gcc –o socket-inet socket-inet.c
```

```
$ ./socket-inet www.codesourcery.com
<html> <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1">
.....
```

Standardni port brojevi

Po dogovoru, Web serveri čekaju na konekcije na portu 80. Većina Internet mrežnih servisa je povezana pomoću standardnih port brojeva. Na primer, sigurni Web serveri koji koriste SSL čekaju na konekciju na portu 443, a e-mail serveri (koji koriste SMTP) koriste port 25. Na GNU/Linux sistemima, veza između imena protokola/servisa i standardnih port brojeva je navedena u datoteci /etc/services. Prva kolona je ime protokola ili ime servisa. Druga kolona navodi broj porta i vrstu konekcije: tcp za konekcijski orjentisanu, ili udp za datagram. Ako implementirate uobičajene mrežne servise koristeći Internet-domain sokete, koristite portove veće od 1024.

Soket parovi

Kao što smo videli ranije, pipe funkcija kreira dva deskriptora datoteka za početak i kraj pipe. Pipe-ovi su ograničeni zato što deskriptore datoteka moraju koristiti samo srodni procesi i zato što je komunikacija unidirekciona (jednosmerna). Funkcija socketpair, kreira dva deskriptora datoteke, za dva povezana soketa na istom računaru. Ovi file-deskriptori dozvoljavaju dvosmernu komunikaciju između srodnih procesa. Njegova prva tri parametra su ista kao i ona kod soket poziva: oni određuju domen (*domain*), stil konekcije, i protokol. Poslednji parametar je niz sa dva celobrojna člana, koji je popunjen sa deskriptorima datoteke za dva soketa, slično kao kod pipe. Kada pozivate socketpair, morate specifirati PF_LOCAL kao domen.

10. Osnovni sistemski pozivi za rad sa datotekama

Prva U/I funkcija sa kojom ste se susreli kada ste počeli da učite jezik C verovatno je `printf`. Ona formatira tekstualni niz (string) i onda ga ispisuje na standardni izlaz. Uopštena verzija, `fprintf`, može da ispiše tekst u datoteku (stream) drugačiju nego standardni izlaz. Datoteka (stream) predstavljena je kao `FILE*` pokazivač. Kada otvorite datoteku sa `fopen`, dobijate `FILE*` pokazivač. Kada završite sa radom, možete je zatvoriti sa `fclose`. Pored `fprint`, možete koristiti funkcije kao sto su `fputc`, `fputs` i `fwrite` koje upisuju podatke u datoteku (stream) ili `fscanf`, `fgetc`, `fgets` i `fread` koje učitavaju podatke.

Sa Linux-ovim operacijama nižeg nivoa, koristimo postupak nazvan deskriptor datoteke umesto `FILE*` pokazivača. Deskriptor datoteke je celobrojna vrednost koja upućuje na određenu datoteku u pojedinačnom pristupu. Deskriptor može biti otvoren za čitanje, za pisanje ili i za čitanje i za pisanje. Deskriptor datoteke ne mora da upućuje na otvorenu datoteku; može predstavljati vezu sa drugom sistemskom komponentom koja može da prima i šalje podatke. Na primer veza između uređaja predstavljena je pomoću deskriptora datoteke, kao otvorena utičnica, ili kao jedan kraj cevovoda (pipe).

Pomenuli smo ranije da su U/I funkcije standardne C biblioteke implementirane na vrhu ovih U/I funkcija niskog nivoa. Ponekad je zgodno:

- da se koriste standardne funkcije sa deskriptorima datoteke,
- ili da se koriste U/I funkcije niskog nivoa, na `*FILE` stream strukturi standardne bibliotke.

GNU/Linux vam omogućuje da koristite oboje.

Ako ste otvorili datoteku koristeći `fopen`, možete dobiti deskriptor datoteke koristeći `fileno` funkciju. Funkcija fileno uzima argument `FILE*` i vraća deskriptor datoteke. Na primer, ako otvorimo datoteku sa standardnom `fopen` funkcijom (funkcija visokog nivoa), upis u nju pomoću funkcije `writev` (funkcija niskog nivoa) možete postići uz pomoć ovog kôda:

```
FILE* stream = fopen (filename, "w");
int file_descriptor = fileno (stream);
writev (file_descriptor, vector, vector_length);
```

Zapamtite da `FILE*` stream i `file_descriptor` odgovaraju istoj otvorenoj datoteci. Ako pozovete sledeću liniju, više ne možete da upisujete u tu datoteku tj. u file_descriptor:

```
fclose (stream);
```

Slično, ako pozovete sledeću liniju, više ne možete da upisujete u stream `FILE*`:

```
close (file_descriptor);
```

Da bi išli obrnuto, od deskriptora datoteke do stream strukture (datoteke) FILE*, koristi se fdopen funkcija. Ona konstruiše stream strukturu FILE*, koja odgovara deskriptoru datoteke. Funkcija fdopen uzima argument deskriptora datoteke i string argument, koji specificira način (mod) na koji se kreira stream struktura. Sintaksa ovog mod argumenta ista je kao i drugog argumenta funkcije fopen, i mora biti kompatibilan sa deskriptorom datoteke. Na primer, naznačite r za čitanje deskriptora datoteke ili w za pisanje u deskriptor datoteke. Kao i kod funkcije fileno, datoteka (stream) i deskriptor datoteke upućuju ka istoj otvorenoj datoteci, pa ako zatvorite jedan, ni onaj drugi nećete moći da koristite.

Otvaranje datoteke

Da bi otvorili datoteku i naveli deskriptor datoteke za pristup toj datoteci, koristite funkciju open. Ona uzima za argumente ime putanje datoteke koja treba da se otvori u vidu slovnog niza i indikator koji opisuje kako da se otvori. Možete koristiti open da kreirate novi datoteku; u tom slučaju, dodajte argumente koji opisuju koja će pristupna prava imati nova datoteka.

Ako je drugi argument O_RDONLY, datoteka je otvorena samo za čitanje; ako budete pokušali da upisujete u nju pojaviće se greška. Slično, O_WRONLY stvara da deskriptor bude samo za upis. Sa O_RDWR, deskriptor datoteke se navodi da bi datoteka bila upotrebljena i za upis i za čitanje. Znajte da ne mogu sve datoteke biti otvorene na sva tri načina. Na primer, dozvole nad datotekom mogu da zabrane određenom procesu da je otvori za čitanje ili za upis; datoteka na uređaju koji je predviđen samo za čitanje kao CD-ROM, ne može biti otvorena za upis.

Možete navesti dodatne funkcije, korišćenjem bita operacija ili sa ovom vrednošću sa jednim ili više indikatora. Ovo su najčešće korišćene vrednosti:

- Navedite O_TRUNC da odbacite otvorenu datoteku, ako prethodno postoji. Podaci upisani u deskriptor datoteke će zameniti prethodni sadržaj datoteke.
- Navedite O_APPEND da dodate u postojeću datoteku. Podaci upisani u deskriptor datoteke biće dodati na kraj datoteke.
- Navedite O_CREAT da kreirate novu datoteku. Ako ime datoteke koje navedete ne postoji, nova datoteka će biti kreirana, pod pretpostavkom da postoji direktorijum koji sadrži datoteku i da proces ima prava da kreira datoteku u tom direktorijumu. Ako datoteka već postoji, onda će se samo otvoriti.

- Navedite `O_EXCL` sa `O_CREAT` da nasilno kreirate novu datoteku. Ako datoteka već postoji, poziv open će biti bezuspešan.

Ako pozovete open sa `O_CREAT`, dodajte treći dodatni argument zadajući dozvole za novu datoteku.

Na primer program `create-file.c` (listing 10.1) kreira novu datoteku sa imenom zadatim u komandnoj liniji. Koristi `O_EXCL` indikator sa open, pa ako datoteka već postoji, dolazi do greške. Novoj datoteci su data prava čitanja i pisanja, vlasniku i grupi vlasnika, i pravo čitanja ostalima. (ako je umask podešen na ne-nulte vrednosti, aktuelne dozvole mogu biti restriktivnije).

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  // Putanja gde ce se kreirati nova datoteka.
  char* path = argv[1];

  // Dozvole za novu datoteku.
  mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH;

  // Kreira datoteku.
  int fd = open (path, O_WRONLY | O_EXCL | O_CREAT, mode);
  if (fd == -1)
  {
    // Dolazi do greške. Ispisuje se poruka o gresci.
    perror ("open");
    return 1;
  }

  return 0;
}
```

Listing 10.1 (create-file.c) Kreiranje nove datoteke

Prevedite program i pokrenite ga navodeci ime nove datoteke kao argument:

```
$ gcc –o create-file create-file.c
$ ./create-file testfile
```

```
$ ls -l testfile
-rw-rw-r--  1   marko   users   0 Feb 1   22:47   testfile
```

Pokušajte ponovo da kreirate istu datoteku:

```
$ ./create-file testfile
open: File exists
```

Primetite da je veličina nove datoteke 0 zato što program nije upisao nikakve podatke u nju.

Umask

Kada kreirate novu datoteku koristeći `open`, neki bitovi prava pristupa koje ste predhodno namestili mogu biti isključeni. To je zato što je vaša umaska podešena na ne-nulte vrednost. Procesov `umask` određuje bitove koji su maskirani van svih novo-kreiranih dozvola. Aktuelna korišćena prava pristupa su kombinacija bita pristupnih prava koja odredimo za otvaranje i bita dopunjenih sa `umask`.

Da bi promenili vaš umask iz komandnog interpretera, koristite `umask` komandu, i odredite numeričku vrednost maske, u oktalnom sistemu. Da bi promenili umask za tekući proces, koristite funkciju `umask`, prosleđujući mu željenu vrednost, kako bi je koristili za sledeću open funkciju.

Na primer, pozivanjem ovog reda

```
umask (S_IRWXO | S_IWGRP);
```

u programu, ili pozivajući ovu komandu

```
$ umask 027
```

određujete dozvolu za pisanje za članove grupe i čitanje, pisanje i pristup drugima će uvek biti maskiran van novih dozvola.

Čitanje podataka

Odgovarajuća komanda za čitanje podataka je `read`. Kao i `write`, on zahteva deskriptor datoteke, pokazivač ka bafer i brojač. Brojač broji koliko je bajtova pročitano iz deskriptora datoteke u bafer. Komada `read` vraca –1 ako je u pitanju greška ili onoliko bajtova koliko je u stvari pročitao. Taj broj može biti manji od broja bajtova koji su zahtevani, na primer, ako nema više bajtova u datoteci.

Čitanje DOS/Windows tekstualnih datoteka

Vaš program će verovatno ponekad trebati da pročita neki DOS ili Windows program. Bitno je primeiti jednu važnu razliku u strukturi datoteka ove dve platforme.

U GNU/Linux tekstualnim datotekama, svaki red je odvojen od sledećeg sa karakterom za novi red. Karakter za novi red je konstanta "\n", koja ima ASCII kôd 10. U Windows sistemu, redovi su separatisani sa kombinacijom 2 karaktera: CR karakter (karakterakter je "\r", ciji je ASCII kôd 13), koji je propraćen sa karakterom za novi red.

Neki GNU/Linux tekst editori prikazuju ^M na kraju svakog reda kod čitanja Windows textualne datoteke – to je CR karakter. Emacs prikazuju Windows tekstualne datoteke pravilno, ali označava ih pokazivanjem (DOS) u statusnoj (mode) liniji na dnu bafera. Neki Windows editori, kao na primer Notepad, prikazuju sav text u GNU/Linux tekst datoteci u jednom redu jer očekuju CR karakter na kraju svakog reda. Drugi programi, koji su i za GNU/Linux i Windows i rade sa tekstom mogu da prijave dosta grešaka, kada im se ubaci pogrešan format teksta.

Ako vaš program obavlja čitanje tekstualne datoteke napravljene sa strane Windows programa, verovatno ćete želeti da promenite sekvencu "/r/n" sa jednim novim redom. Slično, ako vaš program piše tekstualne datoteke, koje trebaju da budu pročitane od strane Windows programa, zamenite karakter za novi red sa "/r/n" kombinacijom. Morate ovo uraditi bilo da koristite niži nivo U/I funkcija koji su objašnjeni u ovom dodatku, bilo da koristite standardne C U/I funkcije.

Listing 10.2 (hexdump.c) prikazuje prostu demonstraciju funkcije read. Program ispisuje heksadecimalne ostatke sadržaja datoteke precizirane na komandnoj liniji. Svaki red prikazuje pomeraj u datoteci i sledećih 16 bajtova.

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  unsigned char buffer[16];
  size_t offset = 0;
  size_t bytes_read;
```

```
  int i;

  // Otvara datoteku za čitanje
  int fd = open (argv[1], O_RDONLY);

  /*čita iz datoteke, jedan po jedna karakter. Nastavlja
  dok čitanje "ne postane kratko", a to je, kada čitanje
  manje nego sto smo mi trazili. To pokazuje da smo
  stigli do kraja datoteke.*/

  do
  {
    // Čitanje bafera
    bytes_read = read (fd, buffer, sizeof (buffer));
    // Pisanje pomeraja u datoteku, pracen je sa samim bajtovima
    printf ("0x%06x : ", offset);
    for (i = 0; i<bytes_read; ++i) printf("%02x ", buffer[i]);
    printf ("\n");
    // Pamti nasu poziciju u datoteci
    offset += bytes_read;
  }
  while (bytes_read == sizeof (buffer));
  // Sve gotovo

  return 0;
}
```

Listing 10.2 (hexdump.c) Ispis heksadecimalnog ostatka datoteke

Sa komandnog prompta prevesti i pokrenuti program. Program prikazuje ispis ostataka izvrsne datoteke `hexdump`, tj same sebe.

```
$ gcc –o hexdump hexdump.c
$ ./hexdump hexdump
0x000000 : 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
0x000010 : 02 00 03 00 01 00 00 00 c0 83 04 08 34 00 00 00
0x000020 : e8 23 00 00 00 00 00 00 34 00 20 00 06 00 28 00
0x000030 : 1d 00 1a 00 06 00 00 00 34 00 00 00 34 80 04 08
...
```

Vaš izlaz može izgledati drugačije, u zavisnosti od prevodioca koji koristite da prevedete hexdump i prevedenih (kompajliranih) indikatora (flags) za prevođenje koje ste selektovali.

Upisivanje podataka

Podsetimo se sistemskog poziva `write`. Upisivanje podataka u deskriptor datoteke se obavlja korišćenjem poziva `write`. Za poziv `write` potrebno je obezbediti sledeće ulazne argumente

- deskriptor datoteke
- pokazivač na bafer podataka
- broj bajtova za upis

Deskriptor datoteke mora biti otvoren za upis. Podaci upisani u datoteku ne moraju biti karakteri, poziv `write` kopira proizvoljne bajtove iz bafera u deskriptor datoteke. Upišite podatke u deskriptor datoteke koristeći poziv `write`. Obezbedite deskriptoru datoteke, pokazivač na bafer podataka i broj bajtova za upis. Deskriptor datoteke mora biti otvoren za upis. Podaci upisani u datoteku ne moraju biti string, tj. write kopira proizvoljne bajtove iz bafera u deskriptor datoteke.

Program `timestamp.c` (listing 10.3) dodaje tekuće vreme u datu datoteku na komandnu liniju. Ako datoteka ne postoji, kreiraće se. Ovaj program takođe koristi `time`, `localtime` i `asctime` funkcije za dobijanje i formatiranje tekućeg vremena.

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>

// Vraca string koji predstavlja tekuci datum i vreme.
char* get_timestamp ()
{
  time_t now = time (NULL);
  return asctime (localtime (&now));
}

int main (int argc, char* argv[])
{
  // Datoteka u koju treba da se doda vremenska oznaka
  char* filename = argv[1];

  // Uzima tekucu vremensku oznaku
```

```
  char* timestamp = get_timestamp ();

  // Otvara datoteku za upis. Ako postoji, pristupa joj.
  int fd = open(filename, O_WRONLY | O_CREAT | O_APPEND, 0666);

  // Racuna duzinu timestamp stringa
  size_t length = strlen (timestamp);

  // Upisuje timestamp u datoteku
  write (fd, timestamp, length);

  // Sve gotovo
  close (fd);

return 0;
}
```

Listing 10.3 (timestamp.c) Dodavanje vremenske oznake datoteci

Program timestamp dodaje tekuce vreme u datu datoteku sa komandne linije. Ako datoteka ne postoji, kreirace se. Prevedite i izvršite ovaj program, a zatim prikažite sadržaj datoteke koju ste naveli kao argument.

```
$ gcc –o timestamp timestamp.c
$ ./timestamp blackadder
$ cat blackadder
Thu Nov  1 21:25:11 2001
```

Pokušajte ponovo da pokrenete program navodeći istu datoteku kao argument.

```
$ ./timestamp blackadder
$ cat blackadder
Thu Nov  1 21:25:11 2007
Thu Nov  1 21:25:38 2007
```

Zapamtite da prvi put kad pozovemo `timestamp`, on kreira datotekui `blackadder`, dok kod drugog poziva samo dodajemo u datoteku `blackadder`.

Kretanje po datoteci

Deskriptor datoteke pamti svoju poziciju u datoteci. Dok čitate iz njega ili pišete po njemu, njegova pozicija napreduje paralelno sa brojem bajtova koje čitamo ili pišemo. Ponekad, moraćete da se krećete po datoteci bez čitanja ili pisanja podataka. Na primer, želite da pišete

po sredini datoteke bez menjanja početka datoteke, ili želite da se vratite na početak datoteke i ponovo da pročitate nesto a da ne morate ponovo da otvarate datoteku i krećete iz početka.

Funkcija `lseek` vam omogućava da pozicionirate deskriptor datoteke u datoteci. Prosledite ga deskriptoru datoteke i dva dodatna argumenta određujući novu poziciju.

- Ako je treći argument `SEEK_SET`, `lseek` tumači drugi argument kao poziciju, u bajtovima, od početka datoteke.
- Ako je treći argument `SEEK_CUR`, `lseek` tumači drugi argument kao pomeraj, koji može biti pozitivan ili negativan, od trenutne pozicije.
- Ako je treći argument `SEEK_END`, `lseek` tumači drugi argument kao pomeraj od kraja datoteke. Pozitivna vrednost pokazuje poziciju izvan kraja datoteke.

Funkcija `lseek` vraća novu poziciju, kao pomeraj od početka datoteke.

Vrsta pomeraja je `off_t`. Ako se pojavi greška, `lseek` vraća -1; `lseek` ne možete koristiti sa nekim vrstama deskriptora datoteka, kao što je utičnica (socket) datoteke.

Ako želite da nađete poziciju deskriptora datoteke u datoteci a da ga ne menjate, odredite pomeraj 0 od trenutne pozicije – na primer:

```
off_t position = lseek (file_descriptor, 0, SEEK_CUR);
```

Linux vam omogućava da `lseek` funkcijom pozicionirate deskriptor datoteke izvan datoteke. Normalno, ako je deskriptor datoteke pozicioniran na kraju datoteke i vi nešto upisujete u njega, Linux ce automatski da proširi datoteku, kako bi napravio mesta za nove podatke. Ako pozicionirate deskriptor datoteke izvan datoteke i onda nesto upisujete u njega, Linux prvo proširuje datoteku kako bi prilagodio rupu koju ste napravili sa lseek operacijom i onda upisuje na kraj datoteke. Ova praznina ne zauzima mesto na disku, naprotiv, Linux samo pravi poruku koliko je to veliko. Ako kasnije pokušate da to pročitate iz datoteke, vašem programu će izgledati kao da je ta praznina popunjena sa 0 bajtovima.

Korišćenjem ovakvog ponašanja `lseek`, moguće je kreirati veoma velike datoteke koje ne zauzimaju skoro nikakav prostor na disku. Program `lseek-huge` (listing 10.4) upravo radi to. On uzima kao komandnu liniju ime datoteke i veličinu tražene datoteke u MB. Program otvara novu datoteku, pozicionira se iza kraja datoteke koristeci `lseek`, a onda upiše jedinstveni 0 bajt pre nego što zatvori datoteku.

```
#include <fcntl.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  int zero = 0;
  const int megabyte = 1024 * 1024;

  char* filename = argv[1];
  size_t length = (size_t) atoi (argv[2]) * megabyte;

  // Otvaranje nove datoteke
  int fd = open (filename, O_WRONLY | O_CREAT | O_EXCL, 0666);

  // Prelazimo na bajt manje od mesta gde hocemo da bude kraj
  lseek (fd, length - 1, SEEK_SET);

  // Pišemo jedinstven 0 bajt
  write (fd, &zero, 1);

  // Sve gotovo
  close (fd);

  return 0;
}
```

Listing 10.4 (lseek-huge.c) Kreiranje velike datoteke koristeći lseek

Prevedite program i iskoristite ga da napravite datoteku od 1GB (1024MB). Proverite slobodno mesto na disku pre i posle operacije.

```
$ gcc –o lseek-huge lseek-huge.c

$ df -h .

Filesystem   Size   Used   Avail  Use%   Mounted on
/dev/hda5    2.9G   2.1G   655M   76%    /
$ ./lseek-huge bigfile 1024
$ ls -l bigfile
-rw-r-----  1 marko  users  1073741824  Feb  5  16:29  bigfile
$ df -h .
Filesystem   Size   Used   Avail  Use%   Mounted on
/dev/hda5    2.9G   2.1G   655M   76%    /
```

Neprimetna količina memorije na disku je potrošena, uprkos ogromnoj veličini bigfile. Opet, iako otvorimo bigfile i čitamo iz njega, on će izgledati kao da je popunjen sa nulama u vrednosti od 1GB. Na primer, možemo da testiramo njegov sadržaj sa hexdump programom `hexdump` (listing 10.2).

```
$ ./hexdump bigfile | head -10
0x000000 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000010 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000020 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000030 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000040 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000050 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
...
```

Ako ovo sami startujete, verovatno ćete hteti da ga prekinete sa Ctrl +C, radije nego da gledate kako on ispisuje preko 2^{30} "0" bajtova.

Zapamtite da su ove magične praznine u datotekama specijalne karakteristike ext2 sistema datoteka koji je tipično korišćen za GNU/Linux diskove. Ako pokušate da koristite `lseek-huge` za stvaranje datoteke na nekim drugim tipovima sistema datoteka, kao što je fat ili vfat sistem datoteka, koji su korišćeni kao postavka za DOS i Windwos particije, otkrićete da datoteka koju dobijete kao rezultat ustvari stvarno popunjava određenu količinu prostora na disku.

Linux vam ne dozvoljava da se pozicionirate sa `lseek` pre početka datoteke.

11. Napredni sistemski pozivi za rad sa datotekama

U prethodnoj vežbi obradili smo primere za sistemske pozive `open`, `read`, `write`, `lseek`. U ovom predavanju daćemo dopunu sa primerima za sledeće sistemske pozive za:

- vektorsko čitanje i pisanje (`writev`, `readv`)
- brzi prenos podataka (`sendfile`)
- čitanje sadržaja simboličkih linkova (`readlink`)
- zaključavanje datoteka (`fcntl`)
- prikazivanje informacija iz i-node strukture datoteke (`stat`)
- proveru prava pristupa (`access`)
- rad sa direktorijumima
- "brisanje" i pomeranje, odnosno promena imena

Vektorsko čitanje i pisanje

Poziv `write` uzima kao argumente

- deskriptor datoteke u koju se upisuje
- pokazivač na početak bafera podataka
- dužinu tog bafera

Poziv `write` upisuje kontinualnu oblast memorije u deskriptor datoteke. Program će ponekad trebati da zapiše nekoliko stavki podataka, a svaka se nalazi u različitim delovima memorije. Da bi koristio jedan `write` poziv, program mora ili da kopira sve stavke podatka u kontinualni deo memorije, a to dovodi do dodatnih CPU-memorijskih ciklusa, ili bi program morao da napravi više `write` poziva, što je takođe neefikasno.

Za neke aplikacije, višestruki `write` pozivi su neefikasni ili nepoželjni. Na primer, kada pišemo na mrežni soket, dve write funkcije mogu da pruzrokuju da dva paketa budu poslata kroz mrežu, a ti isti podaci, mogu biti poslati u jednom paketu, preko jednog poziva `write`.

Poziv `writev` omogućava vam da više diskontinualnih regiona memorije upišete u deskriptor datoteke u jednoj operaciji. Ovo se naziva vektorsko pisanje (*vector write*). Ono što treba dodatno da uradimo, prilikom korišćenja poziva `writev`, je to da moramo da podesimo strukturu podataka, specificiranjem početka i dužine svakog dela memorije. Ova struktura podataka se naziva niz (array) elemenata strukture `iovec`. Svaki element specificira jedan deo memorije za upis, pri čemu polja `iov_base` i `iov_len` određuju početnu adresu dela

memorije i njenu dužinu. Ako unapred znate kolika vam oblast treba, možete prosto da deklarišete niz promenljivih strukture `iovec`; ako broj regiona može da varira, onda morate da dodelite dinamički niz.

Ulazni parametri funkcije `writev` su:

- deskriptor datoteke u koju se upisuje
- niz elemenata strukture `iovec`
- broj elemenata u nizu

Povratna vrednost je ukupan broj upisanih bajtova.

Program `write-arg.c` (listing 11.1) upisuje argumente sa komandne linije u datoteku, koristeći jedan `writev` poziv. Prvi argument je ime datoteke, a drugi i svi ostali argumenti se upisuju u datoteku sa tim imenom, i to svaki u posebnom redu. Program alocira niz sa elementima structure iovec koji je ima duplo više elemenata od broja argumenata koji se upisuju – za svaki argument program upisuje tekst svojih argumenata, kao i karakter za novi red. Pošto ne znamo unapred broj argumenata, niz se alocira korišćenjem funkcije `malloc`.

```
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  int fd;
  struct iovec* vec;
  struct iovec* vec_next;
  int i;

  /* Trebaće nam "bafer" koji sadrži karaktere za novi red.
  Koristite uobičajene char promenljive za to */
  char newline = '\n';

  // Prvi argument u komandnoj liniji je ime datoteke
  char* filename = argv[1];

  /* Preskačemo prva dva elementa sa liste argumenata.
  Element 0 je ime programa, a element 1 je ime datoteke */
```

```
  argc -= 2;
  argv += 2;

  /* Dodeljivanje niza elemenata iovec. Trebaju nam po dva
  za svaki element liste argumenata, jedan za sam text,
  a drugi za novi red */
  vec = (struct iovec*) malloc (2*argc*sizeof (struct iovec));

  // Petlja preko liste argumenata, pravljenje iovec upisa
  vec_next = vec;
  for (i = 0; i < argc; ++i)
  {
    // Prvi element je tekst samog argumenta
    vec_next->iov_base = argv[i];
    vec_next->iov_len = strlen (argv[i]);
    ++vec_next;

    /* Drugi red je jedan karakter za novi red. U redu je
    ako višestruki elementi iz niza struktura iovec pokazuju
    na isti deo memorije*/
    vec_next->iov_base = &newline;
    vec_next->iov_len = 1;
    ++vec_next;
  }

  // Pisanje argumenata u datoteku
  fd = open (filename, O_WRONLY | O_CREAT);
  writev (fd, vec, 2 * argc);
  close (fd);
  free (vec);

  return 0;
}
```

Listing 11.1 (write-arg.c) Pisanje liste argumenata u datoteku koristeći writev

Prevedite program i pokrenite ga navodeći output file kao argument.

```
$ gcc –o write-arg  write-arg.c
$ ./write-arg outputfile "bora" "marko" "dragan" "nemanja"
```

Deklarišemo izlaznu datoteku pod imenom outputfile i tri tekstualna niza koji su ulazni argumenti za writew funkciju

Rezultat izvršavanja programa je:

```
$ cat outputfile
bora
marko
dragan
nemanja
```

Brzi prenosi podataka

Sistemski poziv `sendfile`, omogućava efikasan mehanizam kopiranja podataka iz jednog deskriptora datoteke u drugi. Deskriptori mogu biti otvoreni za disk datoteke, sokete ili druge uređaje.

Da bi se kopiralo iz jednog deskriptora datoteke u drugi, program dodeljuje bafer fiksne veličine, kopira neke podatke iz jednog deskriptora u bafer, a zatim iz istog bafera, upisuje u drugi deskriptor, a to ponavlja sve dok svi podaci ne budu kopirani. Ovo je neefikasno i po pitanju vremena i prostora, zato što zahteva dodatnu memoriju za bafer i pravi suvišnu kopiju podataka u baferu.

Korišćenjem funkcije `sendfile`, posrednički bafer se eliminiše.

Poziv `sendfile` ima sledeće ulazne parametre:

- deskriptor datoteke u koji treba da se upisuje
- deskriptor datoteke sa koga se čita
- pokazivač za promenljivu pomeraja (offset), koja ukazuje odakle treba da počne iz ulazne datoteke;
- broj bajtova koje treba prebaciti.

Pomeraj (offset) promenljiva sadrži pomeraj u ulaznoj datoteci od koga čitanje treba da počne (0 označava početak datoteke) i ova promenljiva se ažurira se na poziciju u datoteci, nakon transfera. Vrednost koju sendfile vraća je broj prebačenih bitova. Uključite `<sys/sendfile.h>` u vaš program ako se koristi funkcija sendfile.

Program `copy.c` (listing 11.2) je jednostavna ali veoma efikasna primena kopiranja datoteka.

Kada se ovaj program pozove sa dva imena za datoteke na komandnoj liniji, program kopira sadržaj prve datoteke u drugu. Program koristi fstat da odredi veličinu izvorne tj. prve datoteke u bajtovima.

```
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/sendfile.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>


int main (int argc, char* argv[])
{
  int read_fd;
  int write_fd;
  struct stat stat_buf;
  off_t offset = 0;

  // Otvaranje ulazne datoteke
  read_fd = open (argv[1], O_RDONLY);

  // Dobijanje veličine ulazne datoteke
  fstat (read_fd, &stat_buf);

  /* Otvaranje izlazne datoteke za upis, sa istim dozvolama
  kao za ulaznu datoteku */
  write_fd=open(argv[2], O_WRONLY | O_CREAT, stat_buf.st_mode);

  // Brzo kopiranje
  sendfile (write_fd, read_fd, &offset, stat_buf.st_size);

  // Zatvaranje datoteka
  close (read_fd);
  close (write_fd);

  return 0;
}
```

Listing 11.2 (copy.c) Kopiranje datoteke pomoću sendfile

Prevedite i pokrenite program.

```
$ gcc –o copy copy.c
$ ./copy /etc/termcap /tmp/termcap.1
```

Deklarišemo ulaznu datoteku `/etc/termcap` i izlaznu datoteku `/tmp/termcap.1`. Prva datoteka predstavlja nešto veću datoteku na

operativnom sistemu Linux, a `/tmp` direktorijum je izabran zato što korisnik tamo ima pravo da upisuje svoje datoteke.

Rezultat izvršavanja programa je:

```
$ ls –l /tmp/termcap.1
```

Videćete novokreiranu datoteku `termcap.1` koju je kreirao sistemski poziv sendfile. Uporedite vremena. Prvo ćemo obrisati datoteku:

```
$ rm /tmp/termcap.1
```

Merimo vreme za klasičnu `cp` komandu

```
$ time cp /etc/termcap /tmp/termcap.1
```

Potom ćemo ponovo obrisati datoteku

```
$ rm /tmp/termcap.1
```

Merimo vreme za klasičnu copy program koji koristi sendfile sistemski poziv

```
$ time ./copy /etc/termcap /tmp/termcap.1
```

Uporedite ta dva vremena.

Sistemski poziv `sendfile` može biti iskorišćen na mnogim primenama da bi se kopiranje učinilo efikasnijim. Jedan dobar primer je na Web serveru ili nekom drugom mrežnom daemonu, koji prenosi sadržaj datoteke, preko mreže do klijent programa. Zahtev se obično prima sa soketa klijent-kompjutera. Server program otvara lokalnu datoteku da bi dobio podatke i upisuje sadržaj datoteke na mrežni soket. Korišćenje funkcije sendfile može znatno da ubrza ovu operaciju. Treba preduzeti i druge korake da bi se mrežni transfer učinio što efikasnijim, kao što je pravilno postavljanje soket parametara. Ali ovo je van teme ove lekcije.

Čitanje sadržaja simboličkih linkova

Sistemski poziv `readlink` obezbeđuje ime objekta na koji pokazuje simbolički link. Uzima tri argumenta:

- putanju do simboličkog linka
- bafer koji prima putanju do datoteke na koju pokazuje link, original datoteke (target)
- dužinu tog bafera.

Poziv readlink ne stavlja null karakter (‘\0’) koji označava kraj stringa, u ovom slučaju u bafer, koji sadrži ime originalne datoteke (target). Ali umesto toga, readlink vraća broj karaktera koji upisuje u

bafer, tako da je upis oznake kraja bafera krajnje jednostavan (samo dodate nulu).

Ako prvi argument za readlink pokazuje na datoteku koja nije simbolički link, readlink postavlja vrednost `errno` na `EINVAL` i vraća vrednost -1.

Program `print-symlink.c` (listing 11.3) prikazuje put do datoteke na koju pokazuje simbolički link (simbolički link se unosi preko komandne linije).

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  char target_path[256];
  char* link_path = argv[1];

  // Pokušaj da se pročita ono na šta pokazuje simbolički link
  int len = readlink (link_path, target_path,
    sizeof (target_path));

  if (len == -1)
  { // Poziv je propao
    if (errno == EINVAL) // Ovo nije simbolički link
      fprintf (stderr, "%s nije simbolicki link\n", link_path);

    else // U pitanju je drugi problem; ispisi generičku poruku
      perror ("readlink");

    return 1;
  }
  else
  {
    // Niz target_path se mora završiti NULL karakterom
    target_path[len] = '\0';

    printf ("%s\n", target_path); // Ispišite je
    return 0;
  }
}
```

Listing 11.3 (print-symlink.c) Prikaz puta do datoteke na koju pokazuje simbolički link

Prevedite program:

```
$ gcc –o print-symlink print-symlink.c
```

Evo primera kako možete napraviti simbolički link i iskoristiti program print-symlink da ga pročitate:

```
$ ln -s /usr/bin/wc my_link
$ ./print-symlink my_link
/usr/bin/wc
```

Zakjučavanje

Sistemski poziv `fcntl` je način da se pristupi nekim naprednim operacijama na deskriptorima datoteka. Prvi argument za `fcntl` je deskriptor otvorene datoteke, a drugi je vrednost koja označava koja operacija treba da bude izvršena. Za neke operacije, fcntl uzima dodatni argument. Ovde ćemo opisati jednu od najkorisnijih fcntl operacija, zaključavanje datoteka (*file locking*). Pogledajte fcntl man stranu, za informacije o ostalim operacijama.

Fcntl sistemski poziv omogućava programu da postavi zaključavanje za čitanje (*read-lock*) ili zaključavanje za upis (*write-lock*) na datoteku, što je u nekoj meri slično mutex semaforima. Zaključavanje za čitanje (*read lock*) se postavlja na deskriptor datoteke sa koga se može čitati, a zaključavanje za upis *(write-lock*) se postavlja na deskriptor datoteke u koji se može pisati. Više od jednog procesa može držati zaključavanje za čitanje (*read-lock*) na istoj datoteci u isto vreme, ali samo jedan proces može držati zaključavanje za upis (*write-lock*). Ista datoteka ne može biti zatvorena (zaključan) i za čitanje i za pisanje. Zapamtite da zaključavanje zapravo ne sprečava ostale procese od otvaranja datoteke, čitanja iz nje ili pisanja po njoj, sem ako i oni takođe ne traže zaključavanje sa `fcntl`.

Da bi se postavilo zaključavanje (lock) na datoteku, prvo napravite i anulirajte strukturu `flock`. Postavite `l_type` polje strukture na `F_RDLCK` za zaključavanje za čitanje (read-lock) ili `F_WRLCK` za zaključavanje za upis (write-lock). Zatim pozovite `fcntl`, sa tri ulazna argumenta:

- deskriptor te datoteke
- `F_SETLCKW` je operacijski kôd
- pokazivač na struct flock promenljivu.

Ako neki drugi proces drži zaključavanje koje sprečava primenu novog zaključavanja, `fcntl` se blokira dok se to zaključavanje ne ukloni.

Program `lock-file.c` (listing 11.4) otvara datoteku za pisanje, a njeno ime se daje u komandnoj liniji, a onda se zaključavanje za upis (write-lock) postavlja na tu datoteku. Program čeka da korisnik pritisne taster <enter>, a onda otključava i zatvara datoteku.

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  char* file = argv[1];
  int fd;
  struct flock lock;

  printf ("Otvaranje datoteke %s\n", file);
  // Otvaranje deskriptora datoteke.
  fd = open (file, O_WRONLY);

  printf ("Zakljucavanje\n");
  // Inicijalizacija flock strukture.
  memset (&lock, 0, sizeof(lock));

  lock.l_type = F_WRLCK;
  // Postavljanje zabrane pisanja (write lock) na datoteku.
  fcntl (fd, F_SETLKW, &lock);
  printf ("Zakljucano! Pritisni Enter za otkljucavanje... ");

  // Čeka na  Enter.
  getchar ();
  printf ("Otključavanje\n");

  // Otključavanje
  lock.l_type = F_UNLCK;
  fcntl (fd, F_SETLKW, &lock);
  close (fd);

  return 0;
}
```

Listing 11.4 (lock-file.c) Postavlja zabranu pisanja korišćenjem fcntl

Prevedite i pokrenite program navodeći ime datoteke `/tmp/test-file` kao argument:

```
$ gcc -o lock-file lock-file.c
$ touch /tmp/test-file
$ ./lock-file /tmp/test-file
Otvaranje datoteke /tmp/test-file
Zakljucavanje
Zakljucano! Pritisni Enter za otkljucavanje...
```

Onda u drugom prozoru pokušajte da pokrenete `lock-file`, ponovo na istoj datoteci.

```
$ ./lock-file /tmp/test-file
Otvaranje datoteke /tmp/test-file
Zakljucavanje
```

Primetimo da je druga instanca blokirana dok pokušava da zaključa datoteku. Vratite se na prvi prozor i pritisnite Enter. Dobićete poruku:

```
Otključavanje
```

Program koji je otvoren u drugom prozoru odmah dobija šansu da obavi zaključavanje za upis (*write-lock*).

Ako želite da se `fcntl` ne blokira, ako poziv ne može da dobije zaključavanje koji ste tražili, koristite `F_SETLK` umesto `F_SETLKW`. Ako se zaključavanje ne može postići, `fcntl` odmah vraća –1

Linux obezbeđuje još jednu primenu zaključavanja datoteka pozivom `flock`. Primena `fcntl` ima jednu veliku prednost: radi sa NFS datotekama (ukoliko je NFS server relativno nov i ispravno konfigurisan). Ako imate pristup preko dve mašine koje aktiviraju isti sistem datoteka preko NFS, možete ponoviti prethodni primer koristeći dve različite mašine. Pokrenite `lock-file` na jednoj mašini, specificirajući datoteku na NFS i pokrenite ga ponovo na drugoj mašini, specificirajući istu datoteku. NFS budi drugi program kada se zaključavanje na upis oslobodi, od strane prvog programa.

Informacije iz i-node strukture datoteke

Korišćenjem poziva `open` i `read`, možete videti sadržaj datoteke. Ali šta je sa ostalim informacijama? Na primer, pozivanje `ls -l` prikazuje, informacije kao što su veličina datoteke, poslednje promene u datoteci, dozvole i vlasnika datoteke.

Funkcija `stat` nam daje te informacije o datoteci. Funkcija `stat` ima dva ulazna argumenta:

- putanja (path) do datoteke koji vas zanima
- pokazivač na promenljivu tipa struct stat

Ako stat odradi sve uspešno, onda vraća vrednost 0 i popunjava polja sa informacijama o datoteci, u suprotnom, stat vraća vrednost -1.

Ovo su najkorišćenija polja u strukturi stat:

- st_mode sadrži prava pristupa te datoteke.
- kao dodatak na prava pristupa, st_mode koduje tip datoteke u svojim višim bitovima.
- st_uid i st_gid sadrze ID vlasnika datoteke i grupe kojoj datoteka pripada.
- st_size sadrži veličinu datoteke, u bajtovima.
- st_atime sadrži vreme kada je poslednji put bilo pristupano toj datoteci (pisanje ili čitanje)
- st_mtime sadrži vreme kada je ta datoteka poslednji put bila promenjena.

Ovi makroi proveravaju vrednost polja st_mode da bi odredili koju vrstu tj. tip datoteke je analizirao vaš sistemski poziv stat. Makro ocenjuje da je to istina (true, tj. 1), ako je datoteka određenog tipa.

- S_ISBLK (mode) - blok uređaj
- S_ISCHR (mode) - karakter uređaj
- S_ISDIR (mode) - direktorijum
- S_ISFIFO (mode) - fifo
- S_ISLNK (mode) - simbolički link
- S_ISREG (mode) - regularna datoteka
- S_ISSOCK (mode) - soket

Polje st_dev sadrži glavni (major) broj i sporedni (minor) broj hardverskog uređaja, preciznije drajvera, kojima blok ili karakter datoteka odgovara. Glavni (major) broj uređaja se pomera ulevo za osam bitova, a sporedni (minor) broj uređaja, obuhvata osam najmanje značajnih bitova. Polje st_ino sadrži inode broj datoteke, pri čemu inode locira datoteku u sistemu datoteka.

Ako pozovete stat na simbolički link, stat prati link i vi možete dobiti informacije o datoteci na koji taj link pokazuje, ali ne i o samom simboličkom linku. Zbog takvog rada, S_ISLINK nikada neće biti istinit (true) kao rezultat stat. Koristite lstat funkciju, ako ne zelite da pratite simbolički link, zato što ova funkcija dobija informacije o samom linku a ne o njenom originalu. Ako pozovete lstat na datoteku, koja nije simbolički link, onda ona ima isti rezultat kao stat. Pozivanjem funkcije

stat, na prekinuti link (link koji pokazuje na nepostojeću ili nepristupačnu originalnu datoteku), kao rezultat se pojavljuje greška, dok se pozivanjem funkcije lstat greška ne prijavljuje.

Ako već imate otvorenu datoteku za čitanje ili pisanje, pozovite `fstat` umesto `stat`. Poziv `fstat` uzima deskriptor datoteke kao njegov prvi argument umesto putanje.

Program `read-file.c` (listing 11.5) prikazuje funkciju koja alocira dovoljno veliki bafer, da prihvati sadržaj datoteke i onda ga učitava u bafer. Funkcija koristi fstat da odredi veličinu bafera, koja je potrebna za alociranje, a takođe se proverava da li je to regularna datoteka.

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

/* Učitava sadržaj FILENAME u novododeljeni bafer.
Veličina bafera je sačuvana u *LENGTH. Vraća bafer, koji
korisnik mora da isprazni. Ako makro ne odgovori da je
datoteka regularna, on vraća NULL vrednost. */

char* read_file (const char* filename)
{
  int fd;
  struct stat file_info;
  char* buffer;
  size_t* length = malloc(sizeof(size_t));

  // Otvara datoteku
  fd = open (filename, O_RDONLY);

  // Uzimanje informacija o datoteci
  fstat (fd, &file_info);
  *length = file_info.st_size;

  // Obratite pažnju da li je u pitanju regularna datoteka
  if (!S_ISREG (file_info.st_mode))
  {
    // Nije, znaci odustani
    close (fd);
    return NULL;
```

```
  }

  /* Dodeljivanje dovoljno velikog bafera da primi sadržaj
  datoteke */
  buffer = (char*) malloc (*length);

  // Učitavanje datoteke u bafer
  read (fd, buffer, *length);

  // Završavanje
  close (fd);
  return buffer;
}

int main(int argc, char *argv[])
{
  char *filename = argv[1];
  char *buffer = read_file(filename);
  printf("Bafer je:\n%s\n", buffer);
}
```

Listing 11.5 (read-file.c) Učitavanje datoteke u bafer

Prevedite program `read-file.c`

```
$ gcc -o read-file read-file.c
```

Pokrenite program na test datoteci `/etc/hosts`:

```
$ ./read-file /etc/hosts
Bafer je:
127.0.0.1       localhost
172.20.3.100    linux.site linux
```

Izlistajte sadržaj datoteke `/etc/hosts` koristeći naredbu `cat` i uporedite sadržaj:

```
$ ./read-file /etc/hosts
127.0.0.1       localhost
172.20.3.100    linux.site linux
```

Provera prava pristupa datoteci

Sistemski poziv `access` određuje da li pozivni proces ima prava pristupa datoteci. Može da proveri bilo koju kombinaciju read, write i execute prava, a takođe proverava postojanje datoteke.

Poziv access ima dva ulazna argumenta.

- Prvi je putanja do datoteke koji treba da se proveri.
- Drugi je bit (bitwise) ili simbolička vrednost `R_OK`, `W_OK`, i `X_OK`, koja odgovara read, write, i execute dozvoli.

Povratna vrednost je 0 ako proces ima sve specificirane dozvole. Ako datoteka postoji, ali pozivni proces nema specificirane dozvole, access vraća –1 i postavlja errno na vrednost `EACCES` (ili na vrednost `EROFS`, ako se dozvola za upis proverava za datoteku u read-only sistemu datoteka).

Ako je drugi argument `F_OK`, access prosto proverava postojanje datoteke. Ako ona postoji, vrednost koja se vraća je 0; ako ne postoji, ta vrednost je –1 i errno je postavljena na vrednost `ENOENT`. Zapamtite da errno može biti postavljen na `EACCES`, ako se bilo kom direktorijumu u putanji datoteke ne može pristupiti.

Program `check-access.c` (listing 11.6) koristi sistemski poziv `access` da proveri postojanje datoteke i proveri read i write dozvole. Potrebno je u komandnoj liniji dati ime datoteke, koja se proverava.

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  char* path = argv[1];
  int rval;

  // Provera postojanja datoteka
  rval = access (path, F_OK);
  if (rval == 0) printf ("%s postoji\n", path);
  else
  {
    if (errno == ENOENT)printf ("%s ne postoji\n", path);
    else if (errno == EACCES)
      printf ("ne mogu da pristupim %s \n", path);
    return 0;
  }

  // Provera read dozvola
  rval = access (path, R_OK);
  if (rval == 0) printf ("%s moze da se cita\n", path);
```

```
  else printf ("%s ne moze da se cita\n", path);

  // Proverava write dozvolu
  rval = access (path, W_OK);
  if (rval == 0) printf ("U %s moze da se pise\n", path);
  else if (errno == EACCES)
    printf ("U %s ne moze da se pise (nema w dozvole)\n", path);
  else if (errno == EROFS)
    printf ("U %s ne moze da se pise (read-only FS)\n", path);

  return 0;
}
```

Listing 11.6 (check-access.c) Provera dozvole pristupa datoteci

Prevedite program `check-access.c`

```
$ gcc -o check-access check-access.c
```

Pokrenite program na test datoteci `/etc/hosts`:

```
$ ./check-access /etc/hosts
/etc/hosts postoji
/etc/hosts moze da se cita
U /etc/hosts ne moze da se pise (nema w dozvole)
```

Na primer, da bi se proverile pristupne dozvole za datoteku `README` na CD-ROM medijumu, primenite sledeće:

```
$ ./check-access /mnt/cdrom/README
/mnt/cdrom/README postoji
/mnt/cdrom/README moze da se cita
U /mnt/cdrom/README ne moze da se pise (read-only FS)
```

Rad sa direktorijumima

Par operacija nad direktorijumima koje mogu da budu korisne su:

- `getcwd` nam daje trenutni radni direktorijum. Ima dva argumenta, bafer i dužinu tog bafera. getcwd kopira putanju trenutnog radnog direktorijuma u bafer.
- `chdir` menja radni direktorijum u kojem se trenutno nalazite, na putanju koju stavite kao ulazni argument.
- `mkdir` kreira novi direktorijum. Njegov prvi argument je putanja gde će se nalaziti taj novi direktorijum. Drugi argument su prava

pristupa za taj novi direktorijum. Interpretacija prava pristupa je ista kao kod trećeg argumenta za funkciju open i modifikuju se na isti način preko umask procesa.

- `rmdir` briše direktorijum. Njegov argument je putanja do direktorijuma.

Čitanje sadržaja direktorijuma

GNU/Linux obezbeđuje funkcije za čitanje sadržaja direktorijuma. Iako one nisu direktno vezane za U/I funkcije nižeg nivoa, objašnjavamo ih jer su često korisne u raznim aplikativnim programima.

Da bi pročitali sadržaj direktorijuma, pratite sledeće korake:

- Pozovite `opendir` funkciju, zadajući putanju direktorijuma koji želimo da ispitamo. Funkcija opendir vraća `DIR*` identifikator, koji ćete da koristite za pristup sadržaju direktorijuma. Ako se pojavi neka greška, funkcija vraća NULL.
- Pozovite `readdir` sa ponavljanjem, prosleđući `DIR*` identifikator koji ste dobili sa opendir. Svaki put kada pozovete `readdir`, on vraća pokazivač na strukturu `dirent` koji odgovara sledećem direktorijumskom ulazu. Kada stignete do kraja sadržaja tog direktorijuma, readdir vraća NULL vrednost. Struktura `dirent` koju dobijate od funkcije `readdir`, ima polje `d_name`, koje sadrži ime direktorijumskog ulaza.
- Pozovite `closedir`, prosleđujući `DIR*` identifikator, da bi završili operaciju listanja.

Uključite `<sys/types.h>` i `<dirent.h>` ako koristite ove funkcije u svom programu.

Zapamtite da ako želite da sadržaj direktorijuma bude poređan u određenom redosledu, to morate sami da uradite.

Program `listdir.c` (listing 11.7) ispisuje sadržaj direktorijuma. Direktorijum može biti zadat u komandnoj liniji, ali ako nije zadat, program koristi trenutni radni direktorijum. Program prikazuje tip i putanju svake datoteke, tj svake stavke u direktorijumu koristeći sistemski poziv `lstat` funkciji `get_file_type`.

```
#include <assert.h>
#include <dirent.h>
#include <stdio.h>
#include <string.h>
```

```
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

/* Vraća string koji opisuje tip datoteke */
const char* get_file_type (const char* path)
{
  struct stat st;
  lstat (path, &st);
  if (S_ISLNK (st.st_mode)) return "simbolicki link";
  else if (S_ISDIR (st.st_mode)) return "direktorijum";
  else if (S_ISCHR (st.st_mode)) return "karakter uredjaj";
  else if (S_ISBLK (st.st_mode)) return "blok uredjaj";
  else if (S_ISFIFO (st.st_mode)) return "fifo";
  else if (S_ISSOCK (st.st_mode)) return "socket";
  else if (S_ISREG (st.st_mode)) return "regularna datoteka";
  else
  // Neocekivano. Mora da bude jedan od navedenih tipova.
  assert (0);
}

int main (int argc, char* argv[])
{
  char* dir_path;
  DIR* dir;
  struct dirent* entry;
  char entry_path[PATH_MAX + 1];
  size_t path_len;

  if (argc >= 2)
    // Ako je direktorijum zadat kao argument, koristite ga
    dir_path = argv[1];
  else
    // U suprotnom koristi trenutni direktorijum
    dir_path = ".";

    // Kopira putanju direktorijuma u entry_path
  strncpy (entry_path, dir_path, sizeof (entry_path));
  path_len = strlen (dir_path);

  /* Ako se putanja direktorijuma ne završava kosom crtom,
  ona se dodaje */
```

```
  if (entry_path[path_len - 1] != '/')
  {
    entry_path[path_len] = '/';
    entry_path[path_len + 1] = '\0';
    ++path_len;
  }

  // Pocetak operacije listanja direktorijuma
  dir = opendir (dir_path);

  while ((entry = readdir (dir)) != NULL)
  {
    const char* type;

    strncpy (entry_path + path_len, entry->d_name,
    sizeof (entry_path) - path_len);

    // Određivanje tipa datoteke (stavke u direktorijumu)
    type = get_file_type (entry_path);

    // Ispisivanje tipa i punog imena sa putanjom
    printf ("%-18s: %s\n", type, entry_path);
  }

  // Sve gotovo
  closedir (dir);

  return 0;
}
```

Listing 11.7 (listdir.c) Ispis liste direktorijuma

Prevedite i izvršite program listdir.c, deklarišite ulaznu datoteku i demonstrirajte sistemske pozive za direktorijume.

Prevedite program `listdir.c` i pokrenite ga navodeći direktorijum `/dev` kao argument.

```
$ gcc -o listdir listdir.c
$ ./listdir /dev
directory : /dev/.
directory : /dev/..
socket : /dev/log
character device : /dev/null
regular file : /dev/MAKEDEV
fifo : /dev/initctl
```

```
character device : /dev/agpgart
...
```

Da bi ovo proverili, možete koristiti komandu `ls` u istom direktorijumu. Navedite –u infikator da bi naložili ls da ne sortira, i specificirajte –a indikator da bi uzrokovali da trenutni direktorijum (.) i roditeljski direktorijum (..) budu uračunati.

```
$ ls -lUa /dev
total 124
drwxr-xr-x 7 root root 36864 Feb 1 15:14 .
drwxr-xr-x 22 root root 4096 Oct 11 16:39 ..
srw-rw-rw- 1 root root 0 Dec 18 01:31 log
crw-rw-rw- 1 root root 1, 3 May 5 1998 null
-rwxr-xr-x 1 root root 26689 Mar 2 2000 MAKEDEV
prw------- 1 root root 0 Dec 11 18:37 initctl
crw-rw-r-- 1 root root 10, 175 Feb 3 2000 agpgart
...
```

Prvi karakter svakog reda koji prikazuje komada `ls` je tip ulaza.

Brisanje, pomeranje i promena imena

Sistemski poziv `unlink` briše datoteku. Njegov argument je putanja do datoteke. Ova funkcija takođe može biti korišćena za brisanje drugih objekata sistema datoteka, kao što su imenovane pipe datoteke (named pipes) i uređaja (`/dev`). U stvari, `unlink` ne mora da obriše sadržaj datoteke. Kao što njegovo ime govori, on raskida link datoteke i direktorijuma u kome se nalazi. Datoteka se više ne ispisuje u tom direktorijumu kada ga listamo, ali ako bilo koji proces ima otvoren deskriptor datoteke za tu datoteku, sadržaj datoteke se ne briše sa diska. Samo kada ni jedan proces nema otvoren deskriptor datoteke onda se sadržaj datoteke briše. Tako da, ako neki process otvori datoteku da bi je čitao ili pisao u nju, a onda drugi proces uradi unlink nad tom datotekom i kreira novu datoteku sa istim imenom, prvi proces će videti stari sadržaj datoteke a ne novi, osim ako ne zatvori prethodno, pa ga ponovo otvori.

Sistemski poziv `rename` preimenuje ili pomera datoteku. Njegova dva argumenta su njegova stara putanja i nova putanja do datoteke. Ako su putanje ka različitim direktorijumima, rename pomera datoteku, samo ako su obe u različitim sistemima datoteka. Možete takođe koristiti `rename` da pomerate direktorijume ili druge objekte sistema datoteka.

12. Uređaji

Linux kao i većina operativnih sistema, vrši interakciju sa hardverskim uređajima putem modularizovanih softverskih komponenata koji se nazivaju drajveri uređaja. Drajver uređaja sakriva detalje komunikacionih protokola hardverskog uređaja od operativnog sistema i dozvoljava sistemu da vrši interakciju sa uređajem putem standardizovanog interfejsa.

Pod Linuxom, drajveri uređaja su deo jezgra i mogu biti ili povezani (linkovani) statički u jezgro ili se pozivati po potrebi kao moduli jezgra. Drajveri uređaja se izvršavaju kao deo jezgra i nisu direktno pristupni kao korisnički procesi. Ipak, Linux obezbeđuje mehanizam putem koga procesi mogu da komuniciraju sa drajverom uređaja - a preko njega i sa samim hardverskim uređajem - putem objekata koji su slični datotekama. Ovi objekti su prikazani u sistemu datoteka i programi ih mogu otvoriti, isčitavati iz njih i upisivati u njih, praktično kao da su standardne datoteke. Koristeći ili Linuxove ulazno/izlazne operacije niskog nivoa ili standarde ulazno/izlazne operacije iz C biblioteke, vaši programi mogu da komuniciraju sa hardverskim uređajima preko ovih kvazi-fajl objekata.

Linux takođe omogućava nekoliko kvazi-fajl objekata koji mogu da komuniciraju direktno sa jezgrom, pre nego sa drajverima uređaja. Oni nisu povezani sa hardverskim uređajima; umesto toga, oni omogućavaju razne vrste specijalnih ponašanja koje mogu biti od koristi aplikacijama i sistemskim programima.

Tipovi uređaja

Nodovi za uređaje nisu obične datoteke - one ne predstavljaju delove podataka na disk-baziranim sistemima fajlova. Umesto toga, podaci koji se isčitavaju ili upisuju u nod za uređaje povezani su sa odgovarajućim drajverom uređaja, a posle toga i sa samim uređajem. Postoje dva tipa uređaja:

- Karakter uređaj, tj uređaj "bajt-po-bajt" predstavlja hardverski uređaj koji isčitava ili upisuje serijski tok bajtova podataka. Serijski i paralelni portovi i terminali su neki od primera za uređaj tipa "bajt-po-bajt".
- Blok uređaj predstavlja hardverski uređaj koji isčitava ili upisuje podatke u blokovima podataka fiksirane veličine. Za razliku od karakter uređaja, blok uređaj obezbeđuje slučajan pristup skladištenim podacima. Čvrsti disk je primer blok uređaja.

Tipični aplikativni programi nikada neće koristiti blok uređaje. Dok je čvrsti disk predstavljen kao blok uređaj, sadržaj svake particije na disku obično sadrži sistem datoteka, i taj sistem datoteka je montiran

(mounted) na koreni sistem datoteka GNU/Linux-a. Samo izvršni kod jezgra koji implementira sistem datoteka mora da pristupa direktno blok uređaju; aplikativni programi pristupaju sadržaju diska preko standardnih datoteka i direktorijuma.

Opasnosti vezane za blok uređaje

Blok uređaji obezbeđuju direktan pristup podacima na čvrstom disku. Iako je većina GNU/Linux sistema konfigurisana tako da spreči ne-kernelske procese da pristupaju ovakvim uređajima direktno, kernelski procesi mogu da izazovu teška oštećenja prilikom promene podataka na disku. Program koji radi sa diskom kao sa običnim blok uređajem, može da izmeni ili uništi kontrolne informacije sistema datoteka, a čak i particije diska, boot zapis i MBR, time uništavajući podatke, a u najgorem slučaju i ceo sistem. Uvek pristupajte ovim diskovima s mnogo pažnje.

Označavanje uređaja

Linux identifikuje uređaje koristeći dva broja: glavni broj uređaja i sporedni broj uređaja. Glavni broj uređaja određuje kom drajveru odgovara koji uređaj. Ta povezanost glavnih brojeva uređaja i drajvera je fiksna i deo je jezgarnih komponenti Linuxa. Uočite da isti glavni broj uređaja može biti povezan sa dva različita drajvera, jednim za uređaj "bajt-po-bajt" i drugim za blok uređaj.

Sporedni brojevi uređaja razlikuju individualne uređaje ili komponente kojima upravlja jedan drajver. Značenje sporednog broja uređaja zavisi od drajvera uređaja.

Na primer, glavni uređaj pod brojem 3 odgovara primarnom IDE kontroleru sistema. IDE kontroler može imati dva uređaja (čvrsti, trakasti ili CD-ROM drajv) povezana na njega; "glavni" (*master*) uređaj kao broj sporednog uređaja ima 0, dok "sporedni" (*slave*) uređaj kao broj sporednog uređaja ima 64. Individualne particije na master uređaju (ako uređaj podržava particije) su predstavljene sporednim brojevima uređaja 1, 2, 3... itd. Individualne particije na slave uređaju su predstavljene sporednim brojevima uređaja 65, 66, 67, itd.

Glavni brojevi uređaja su zavedeni u dokumentaciji o kernelskim komponentama Linuxa. Na mnogim Linux distribucijama dokumentacija se može naći u datoteci `/usr/src/linux/Documentation/devices.txt`. Specijalani nod `/proc/devices` prikazuje glavne brojeve uređaja koji odgovaraju aktivnim drajverima uređaja koji su trenutno učitani u jezgro (opisano u poglavlju o /proc sistemu).

Nodovi za uređaje

Nod za uređaj je na mnogo načina isti kao i obična datoteka. Možete ga pomerati koristeći mv komandu i brisati ga koristeći rm komandu. Ako, doduše, pokušate da iskopirate nod za uređaje koristeći cp komandu, isčitaćete bajtove sa uređaja (ako uređaj podržava isčitavanje) i upisaćete ih u odredišnu datoteku. Ako pokušate da prepišete nod za uređaje, umesto toga ćete upisati bajtove na odgovarajući uređaj.

Možete kreirati nod za uređaje u sistemu datoteka koristeći `mknod` komandu (pozovite `man 1 mknod` za stranicu sa uputstvom) ili `mknod` sistemski poziv (pozovite `man 2 mknod` za stranicu sa uputstvom). Kreiranje noda za uređaje u sistemu datoteka ne znači automatski da je odgovarajući drajver uređaja ili hardverski uređaj prisutan ili raspoloživ; Nod za uređaj je samo portal za komunikaciju sa drajverom, ako je ovaj prisutan. Samo procesi superuser-a (administratora sistema) mogu da naprave blok uređaje ili uređaje "bajt-po-bajt" koristeći mknod komandu ili mknod sistemski poziv.

Da biste kreirali uređaj koristeći mknod komandu, kao prvi argument naznačite putanju gde bi nod trebalo da se u sistemu datoteka pojavi. Kao drugi argument, naznačite b za blok uređaj ili c za uređaj "bajt-po-bajt". Obezbedite glavne i sporedne brojeve uređaja kao treći i četvrti argument, respektivno. Na primer, ova komanda pravi uređaj "bajt-po-bajt" pod imenom lp0 u tekućem direktorijumu. Uređaj ima glavni broj uređaja 6 i sporedni broj 0. Ovi brojevi odgovaraju prvom paralelnom portu na Linux sistemu.

```
$ mknod ./lp0 c 6 0
```

Zapamtite da samo superuser procesi mogu da stvaraju blok i uređaje "bajt-po-bajt", tako da morate biti prijavljeni na sistem kao root da biste uspešno pozvali ovu komandu.

Komanda ls posebno je namenjena za prikaz nodova za uređaje. Ako pozovete komandu `ls` sa `-l` ili `-o` opcijama, prvi karakter na svakoj liniji izlaza predstavljaće tip noda. Setite se da `-` (povlaka) označava normalnu datoteku, dok `d` predstavlja direktorijum. Slično, `b` prestavlja blok uređaj, a `c` uređaj "bajt-po-bajt". Za druga dva, `ls` izlistava glavne i sporedne brojeve uređaja gde bi inače stajala veličina obične datoteke. Na primer, možemo prikazati upravo kreirani blok uređaj:

```
$ ls -l lp0
crw-r-----  1  root  root  6, 0  Mar 7  17:03  lp0
```

U programu , možete odrediti da li je nod sistema datoteka blok ili uređaj "bajt-po-bajt" i zatim dobiti brojeve uređaja koristeći `stat`.

Da bi se nod obrisao, koristi se `rm`. Ova komanda ne briše uređaj ili broj uređaja; ona jednostavno sklanja nod uređaja iz sistema datoteka.

```
$ rm ./lp0
```

Direktorijum /dev

Konvencionalno, na svakom GNU/Linux sistemu postoji direktorijum `/dev` koji sadrži sve Linuxu poznate nodove blok i uređaja "bajt-po-bajt". Nodovi u direktorijumu `/dev` su standardizovana imena koja odgovaraju glavnim i sporednim brojevima uređaja. Na primer, glavni uređaj koji je priključen na primarni IDE kontroler, koji ima glavne i sporedne brojeve 3 i 0, ima standardizovano ime `/dev/hda`. Ako ovaj uređaj podržava particije, prva particija na njemu, koja ima sporedni broj uređaja 1, ima standardizovano ime `/dev/hda1`. Možete proveriti da li je to tako i na vašem sistemu:

```
$ ls -l /dev/hda /dev/hda1
brw-rw----  1  root  disk  3, 0  May 5 1998  /dev/hda
brw-rw----  1  root  disk  3, 0  May 5 1998  /dev/hda1
```

Slično tome, /dev direktorijum sadrži nod za uređaj "bajt-po-bajt" na paralelnom portu koji smo ranije koristili:

```
$ ls -l /dev/lp0
crw-rw----  1  root  daemon 6, 0 May 5 1998  /dev/lp0
```

U većini sličajeva, ne bi trebali da koristite mknod komadnu za kreiranje sopstvenih nodova za uređaje. Umesto toga, koristite nodove u /dev direktorijumu. Korisnički programi nemaju izbora već moraju da koriste postojeće nodove uređaja, jer ne mogu da naprave sopstveni. Tipično, samo sistemski administratori i razvojni programeri koji koriste specijalizovane hardverske uređaje će osetiti potrebu da kreiraju sopstvene nodove uređaja. Većina GNU/Linux distribucija sadrži uputstva koja pomažu sistemskim administratorima da pravilno kreiraju nodove uređaja.

Pristup uređajima putem otvaranja datoteka

Kako koristiti ove uređaje? U slučaju uređaja "bajt-po-bajt", to može biti vrlo jednostavno: Otvoriti uređaj kao da je obična datoteka i potom iz njega čitati ili u njega pisati. Čak možete koristiti regularne komande za rad sa datotekama, kao što su cat, ili vašu redirekcionu sintaksu, da biste poslali podatke na ili sa uređaja.

Na primer, ako imate štampač povezan na prvi paralelni port vašeg računara, možete štampati datoteke šaljući ih direktno na `/dev/lp0`. Da biste štampali sadržaj datoteke `document.txt`, pozovite sledeće:

```
$ cat document > /dev/lp0
```

Morate imati dozvolu da upisujete na nod za uređaje da bi ovo radilo; na mnogim GNU/Linux sistemima, dozvole su podešene tako da samo root i daemon sistemskog štampača (`lpd`) mogu da upisuju nešto u datoteku. Takođe, ono što izađe iz vašeg štampača zavisi od toga kako štampač tumači podatke koje mu pošaljete. Neki štampači će štampati običan tekst koji se pošalje na njih, dok drugi neće. PostScript štampači će obraditi i štampati PostScript datoteke koje pošaljete na njih.

U programu, slanje podataka na uređaje takođe je vrlo jednostavno. Na primer, ovaj deo koda koristi U/I funkcije niskog nivoa da pošalje sadržaj bafera na `/dev/lp0`.

```
int fd = open ("/dev/lp0", 0_WRONGLY);
write (fd, buffer, buffer_length);
close (fd);
```

Hardverski uređaji

Neki uobičajeni blok uređaji su izlistani u tabeli 12.1. Brojevi uređaja za slične uređaje prate očigledan šablon (na primer, druga particija na prvom SCSI drajvu je `/dev/sda2`). Nekada je korisno znati kom uređaju odgovara koje ime uređaja, kada se pregledaju montirani sistemi datoteka u `/proc/mounts`.

Uređaj	Ime	Glavni	Sporedni
Prvi floppy drajv	`/dev/fd0`	2	0
Drugi floppy drajv	`/dev/fd1`	2	1
Primarni IDE kontroler, master uređaj	`/dev/hda`	3	0
Primarni IDE kontroler, master uređaj, prva particija	`/dev/hda1`	3	1
Primarni IDE kontroler, slave uređaj	`/dev/hdb`	3	64
Primarni IDE kontroler, slave uređaj, prva particija	`/dev/hdb1`	3	65
Sekund. IDE kontroler, master uređaj	`/dev/hdc`	22	0
Sekund. IDE kontroler, slave	`/dev/hdd`	22	64

Uređaj	Ime	Glavni	Sporedni
uređaj			
Prvi SCSI drajv	`/dev/sda`	8	0
Prvi SCSI drajv, prva particija	`/dev/sda1`	8	1
Drugi SCSI drajv	`/dev/sdb`	8	16
Drugi SCSI drajv, prva particija	`/dev/sdb1`	8	17
Prvi SCSI CD-ROM drajv	`/dev/scd0`	11	0
Drugi SCSI CD-ROM drajv	`/dev/scd1`	11	1

Tabela 12.1 Deo listinga uobičajenih blok uređaja

Uređaj	Ime	Glavni	Sporedni
Paralelni port 0	`/dev/lp0` `/dev/par0`	6	0
Paralelni port 1	`/dev/lp1` `/dev/par1`	6	1
Prvi serijski port	`/dev/ttyS0`	4	64
Drugi serijski port	`/dev/ttyS1`	4	65
IDE trakasti drajv	`/dev/ht0`	37	0
First SCSI trakasti drajv	`/dev/st0`	9	0
Drugi SCSI trakasti drajv	`/dev/st1`	9	1
Sistemska konzola	`/dev/console`	5	1
Prvi virtuelni terminal	`/dev/tty1`	4	1
Drugi virtuelni terminal	`/dev/tty2`	4	2
Trenutni terminalni uređaj procesa	`/dev/tty`	5	0
Zvučna kartica	`/dev/audio`	14	4

Tabela 12.2 Deo listinga uobičajenih "bajt-po-bajt" uređaja

Možete pristupiti pojedinim hardverskim komponentama preko više od jednog uređaja "bajt-po-bajt"; često, drugi uređaj "bajt-po-bajt" obezbeđuje drugu semantiku. Na primer, kada koristite IDE trakasti uređaj `/dev/ht0`, Linux automatski premotava traku u drajvu kada zatvorite deskriptor datoteka. Možete koristiti uređaj `/dev/nht0` da bi pristupili istom trakastom drajvu, samo što tada Linux neće automatski premotati traku kada zatvorite deskriptor datoteka. Ponekad možete videti programe kako koriste `/dev/cua0` i slične uređaje; ovo su stariji interfejsi ka serijskim portovima kao što je `/dev/ttyS0`.

Povremeno, poželećete da upisujete podatke direktno na uređaj "bajt-po-bajt" - na primer:

- Terminalski program može pristupiti modemu direktno preko serijskog porta uređaja. Podaci upisani ili isčitani sa uređaja su poslati preko modema na udaljeni računar.
- Bekap (sigurnosna kopija) program trakastog uređaja može upisivati podatke direktno na trakasti uređaj. Bekap program može implementirati sopstvene formate kompresije i provere grešaka.
- Program može upisivati direktno na prvi virtuelni terminal pišući podatke na `/dev/tty1`. (terminalski prozori koji su pokrenuti u grafičkom okruženju, ili sesije udaljenih pristupnih terminala, nisu povezani sa virtualnim terminalima; umesto toga, oni su povezani sa kvazi-terminalima, o kojima se govori pred kraj ovog poglavlja.)
- Programu je ponekad potrebno da pristupi terminalskom uređaju sa kojim je povezan. Na primer, vaš program će možda pitati korisnika za lozinku. Iz bezbednosnih razloga, možete želeti da ignorišete redirekciju standardnih ulaza i izlaza i uvek isčitavate lozinku sa terminala, bez obzira kako je korisnik pozvao komandu. Jedan od načina da se ovo uradi je da se otvori `/dev/tty`, koji uvek odgovara terminalskom uređaju koji je povezan sa procesom koji ga je otvorio. Upišite odzivnu poruku na taj uređaj i isčitajte lozinku sa njega. Ignorišući standardni ulaz i izlaz, ovo sprečava korisnika da ubacuje lozinku u vaš program koristeći osnovnu sintaksu kao što je ova:

```
$ secure_program < my-password.txt
```

- Program može da reprodukuje zvuke preko sistema zvučne kartice tako što šalje audio podatke na `/dev/audio`. Primetite da audio podaci moraju biti u Sun audio formatu (obično je povezan sa .au ekstenzijom). Na primer:

```
$ cat /usr/share/sndconfig/sample.au > /dev/audio
```

Specijalni uređaji

Linux takođe obezbeđuje nekoliko uređaja "bajt-po-bajt" koji nisu u relaciji sa hardverskim uređajima. Svi ovi nodovi koriste glavni broj uređaja 1, koji je povezan sa uređajem Linux-ovog jezgra umesto sa drajverom uređaja.

/dev/null

Nod `/dev/null`, null uređaj, je vrlo koristan. On ima dve namene; vrlo verovatno ste upoznati sa najmanje jednom od njih:

- Linux odbacuje bilo koje podatke upisane na `/dev/null`. Uobičajen trik je da se `/dev/null` naznači kao izlazna datoteka u kontekstu kada je izlaz nepoželjan.Na primer, da biste izvršili komandu i odbacili njen standardni izlaz (bez štampanja ili upisivanja u datoteku), preusmerite izlaz na `/dev/null`:

```
$ verbose_command > /dev/null
```

- Čitanje iz `/dev/null` uvek rezultira kao EOF (end-of-file, kraj datoteke). Na primer, ako biste otvorili opis datoteke u `/dev/null` koristeći open i zatim pokušali da read (isčitate) iz deskriptora datoteke, read neće pročitati nijedan bajt i vratiće 0. Ako kopirate iz `/dev/null` u drugu datoteku, odredište će biti datoteka dužine (veličine) 0:

```
$ cp /dev/null empty-file
$ ls -l empty-file
-rw-rw-----  1  stojko  users  0  Mar 8  00:27  empty-file
```

/dev/zero

Nod za uređaje `/dev/zero` ponaša se kao beskrajno duga datoteka ispunjena sa 0 bajtova. Koliko god podataka pokušate da isčitate iz `/dev/zero`, Linux «generiše» dovoljno 0 bajtova.

Da bi ovo prikazali, pokrenimo program `hexdump` program navodeći nod `/dev/zero` kao argument (program štampa sadržaj datoteke u heksadecimalnom obliku, izvorni kod `hexdump.c` dat u listingu 10.2).

```
$ ./hexdump /dev/zero
0x000000 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000010 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000020 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000030 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
...
```

Pritisnite CTRL+C kada budete uvereni da će nastaviti u beskonačnost.

/dev/full

Datoteka `/dev/full` se ponaša kao da je datoteka na sistemu datoteka koji nema više prostora na sebi. Upis na `/dev/full` ne uspeva i postavlja `errno` na `ENOSPC`, što obično ukazuje na to da je uređaj na koji se vrši upis pun.

Na primer, možete probati da upisujete na `/dev/full` koristeći `cp` komandu:

```
$ cp /etc/fstab /dev/full
cp: /dev/full: No space left on device
```

Datoteka `/dev/full` je izuzetno korisna za testiranje kako se vaš program ponaša kada mu nestane prostora na disku dok upisuje u datoteku.

Generatori slučajnih brojeva

Specijalni uređaji `/dev/random` i `/dev/urandom` obezbeđuju pristup generatoru slučajnih brojeva jezgra Linuxa.

Većina softverskih funkcija za generisanje slučajnih brojeva, kao što je rand funkcija u standardnoj C biblioteci, u stvari generišu kvazi-slučajne brojeve. Iako ovi brojevi zadovoljavaju neke osobine slučajnih brojeva, oni se mogu reprodukovati: Ako počnete sa istom "seme" vrednošću (početna vrednost), svaki put ćete dobiti istu sekvencu kvazi-slučajnih brojeva. Ovakvo ponašanje je neminovno jer su računari u suštini predvidivi. Doduše, za određene aplikacije ovo ponašanje je nepoželjno; na primer, ponekad je moguće razbiti kriptografski algoritam ako možete da pribavite niz slučajnih brojeva koji on koristi.

Da bi se dobila bolja "slučajnost" brojeva u kompjuterskim programima, potreban je spoljašnji izvor slučajnosti. Jezgro Linuxa uzima naročito dobar izvor slučajnosti: VAS! Mereći vremenske razmake između vaših akcija unosa, kao što su otkucaji na tastaturi i pomeranje miša, Linux je u stanju da generiše nepredvidiv niz visoko kvalitetnih slučajnih brojeva. Možete pristupiti ovom nizu čitajući iz `/dev/random` i `/dev/urandom`. Podaci koje isčitate su niz slučajno generisanih bajtova.

Razlika između ova dva uređaja se sama otkriva kada Linux istroši svoju "zalihu slučajnosti". Ako pokušate da isčitate veliki broj bajtova iz `/dev/random`, a pritom niste generisali bilo kakve akcije unosa (niste kucali, pomerali miša, ili slično), Linux će blokirati operaciju čitanja.

Samo kada obezbedite neku "slučajnost", Linux će generisati neke slučajne bajtove i vratiti ih vašem programu.

Na primer, pokušajte da prikažete sadržaj `/dev/random` uređaja koristeći komandu od. Svaki red izlaza prikazuje 16 slučajnih bajtova. Koristimo program `od` umesto `hexdump` programa, iako u principu rade jednu te istu stvar, jer `hexdump` ugasi sam sebe kada mu nestane podataka, dok `od` čeka dok još podataka ne bude dostupno. Opcija `-t x1` govori od programu da štampa sadržaj datoteke u heksadecimalnom obliku.

```
$ od -t x1 /dev/random
0000000 2c 9c 7a db 2e 79 3d 65 36 c2 e3 1b 52 75 1e 1a
0000020 d3 6d 1e a7 91 05 2d 4d c3 a6 de 54 29 f4 46 04
0000040 b3 b0 8d 94 21 57 f3 90 61 dd 26 ac 94 c3 b9 3a
0000060 05 a3 02 cb 22 0a bc c9 45 dd a6 59 40 22 53 d4
```

Broj linija izlaza koje vidite će varirati; može ih biti poprilično, ali izlaz će eventualno pauzirati kada Linux istroši zalihu "slučajnosti". Sada pokušajte da pomerite miša ili da kucate na tastaturi i vidite kako se dodatni slučajni brojevi pojavljuju. Za još bolji izbor slučajnosti, možete se igrati sa tasterima.

Unos sa `/dev/urandom`, u kontrastu sa prethodnim, nikada neće blokirati. Ako Linux istroši zalihu "slučajnosti", koristiće kriptografske algoritme da generiše kvazi-slučajne bajtove iz prošlog niza slučajnih brojeva. Iako su ovi bajtovi dovoljno slučajni za mnoge upotrebe, oni neće proći toliko testova slučajnosti kao oni generisani iz `/dev/random`.

Na primer, ako pozovete sledeće slučajni bajtovi će proletati u beskonačnost, dok ne zatvorite program sa Ctrl+C:

```
% od -t x1 /dev/urandom
0000000 62 71 d6 3e af dd de 62 c0 42 78 bd 29 9c 69 49
0000020 26 3b 95 bc b9 6c 15 16 38 fd 7e 34 f0 ba ce c3
0000040 95 31 e5 2c 8d 8a dd f4 c4 3b 9b 44 2f 20 d1 54
...
```

Koristeći slučajne brojeve iz `/dev/random` u programu je takođe lako. Listing `random-number.c` prezentuje funkciju koja generiše slučajne brojeve koristeći bajtove isčitane iz `/dev/random`.

Zapamtite da `/dev/random` blokira isčitavanje dok ne postoji dovoljno "slučajnosti" da zadovolji generator; umesto toga, možete koristiti `/dev/urandom` ako je važnije brzo izvođenje i ako vam ne smeta potencijalno lošiji kvalitet slučajnih brojeva.

```
#include <assert.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>

/* Vraća slučajan ceo broj između MIN i MAX, uključujući i
te dve vrednosti. Dobavlja "slučajnost" iz /dev/random. */
int random_number (int min, int max)
{
  /* Skladišti deskriptor datoteke otvoren za /dev/random
  u statičku promenljivu. Na ovaj način, ne moramo da
  otvaramo datoteku svaki put kada je ova funkcija pozvana */
  static int dev_random_fd = -1;
  char* next_random_byte;
  int bytes_to_read;
  unsigned random_value;

  // Utvrditi da je MAX veći od MIN.
  assert (max > min);

  /* Ako je ovo prvi put da je ova funkcija pozvana,
  otvoriti deskriptor datoteke u /dev/random. */
  if (dev_random_fd == -1)
  {
    dev_random_fd = open ("/dev/random", O_RDONLY);
    assert (dev_random_fd != -1);
  }

  /* Pročitati dovoljno slučajnih bajtova da bi se napunila
  celobrojna vrednost */
  next_random_byte = (char*) &random_value;
  bytes_to_read = sizeof (random_value);

  /* Držati u petlji dok ne pročitamo dovoljno bajtova.
  Jer je /dev/random popunjen korisnički-generisanim
  akcijama, isčitavanje se može blokirati i vraćati, jedan
  po jedan, samo pojedinačne slučajne bajtove. */
  do
  {
    int bytes_read;
    bytes_read = read (dev_random_fd, next_random_byte,
```

```
      bytes_to_read);
    bytes_to_read -= bytes_read;
    next_random_byte += bytes_read;
  } while (bytes_to_read > 0);


  // Izračunati slučajan broj u pravilnom opsegu.
  return min + (random_value % (max - min + 1));
}

int main(int argc, char* argv[])
{
  int i;
  for (i=1; i<=10; i++) printf ("%d\n",
    random_number (atoi(argv[1]), atoi(argv[2])));
  return 0;
}
```

Listing 12.1 (random_number.c) Generisanje slučajnih brojeva

Prevedite i pokrenite program navodeći gornju i dodju granicu. Program će generisati 10 slučajnih brojeva u tom opsegu.

```
$ gcc -o random_number random_number.c
$ ./random_number 1 10000
7465
8801
799
9678
1348
2888
1289
105
6389
9557
```

loopback uređaji

Uređaj povratne petlje vam omogućava da simulirate blok uređaj koristeći običnu datoteku na disku. Zamislite čvrsti disk na koga su podaci upisivani i isčitavani iz datoteke koja je nazvana disk-image, umesto iz traka i sektora stvarnog fizičkog diska ili disk particije. (Naravno, datoteka disk-image mora postojati na stvarnom disku, koji

mora biti većeg kapaciteta od simuliranog diska). Uređaj povratne petlje omogućava korišćenje datoteke na ovaj način.

Uređaji povratne petlje su nazvani `/dev/loop0`, `/dev/loop1`, itd. Svaki ponaosob može biti iskorišćen da simulira jedan blok uređaj u jednom trenutku. Primetite da samo superuser može da podesi uređaj povratne petlje.

Uređaj povratne petlje može biti korišćen na isti način kao i svaki drugi blok uređaj. Možete konstruisati sistem datoteka na uređaju i onda montirati taj sistem datoteka kao što biste ga montirali na običnom disku ili particiji. Takav sistem datoteka koji je u celosti nastanjen u običnoj datoteci na disku se naziva virtuelni sistem datoteka.

Da bi se konstruisao virtuelni sistem datoteka i montirao kao uređaj povratne petlje ispratite sledeće korake:

1. Napravite praznu datoteku koja će poslužiti za skladištenje virtuelnog sistema datoteka. Veličina datoteke će, očigledno, biti veličina uređaja povratne petlje posle montiranja. Lakši način kreiranja datoteka fiksne veličine postiže se dd komadnom. Ova komanda kopira blokove (standardno od 512 bajtova svaki) iz jedne datoteke u drugu. Pogodan izvor bajtova za kopiranje je iz `/dev/zero`. Da bi se napravila datoteka veličine 10MB nazvana `disk-image`, pozovite sledeće:

```
$ dd if=/dev/zero of=/tmp/disk-image count=20480
20480+0 records in
20480+0 records out
$ ls -l /tmp/disk-image
-rw-rw----  1 root root 10485760 Mar 8 01:56 /tmp/disk-image
```

2. Datoteka koju ste upravo kreirali je ispunjena 0 bajtovima. Pre nego što je montirate, morate konstruisati sistem datoteka. To podešava različite kontrolne strukture potrebne za organizaciju i skladištenje datoteka i gradi koreni direktorijum. Možete kreirati bilo koji tip sistema datoteka u svom `disk-image`-u. Da bi se kreirao ext2 sistem datoteka (najčešće korišćeni tip na Linux diskovima) koristi se `mke2fs` komanda. Jer se obično pokreće na blok uređajima, a ne običnim datotekama, tražiće potvrdu:

```
$ mke2fs -q /tmp/disk-image
mke2fs 1.18, 11-Nov-1999 for EXT2 FS 0.5b, 95/08/09
disk-image is not a block special device.
Proceed anyway? (y,n) y
```

Opcija `-q` onemogućava prikazivanje informacija o novo-kreiranom sistemu datoteka.

Sada disk-image sadrži nov sistem datoteka, kao da je sveže inicijalizovan čvrsti disk od 10MB.

3. Montirajte sistem datoteka koristeći uređaj povratne petlje. Da biste uradili ovo koristite mount komandu, određujući datoteku `disk-image` kao montirani uređaj. Takođe odredite `loop=loopback-device` kao opciju montiranja, koristeći `-o` opciju da biste rekli `mount` komandi koji uređaj povratne petlje da koristi.

Na primer, da biste montirali naš disk-image sistem datoteka, pozovite sledeće komande. Zapamtite, samo superuser može koristiti uređaj povratne petlje. Prva komanda kreira direktorijum `/tmp/virtual-fs`, koji se koristi kao tačka montiranja za virtuelni sistem datoteka.

```
$ mkdir /tmp/virtual-fs
$ mount -o loop=/dev/loop0 /tmp/disk-image /tmp/virtual-fs
```

Sada je disk-image montiran kao da je običan čvrsti disk od 10MB.

```
$ df -h /tmp/virtual-fs
Filesystem        Size   Used  Avail  Use%   Mounted on
/tmp/disk-image   9.7M   13k    9.2M    0%   /tmp/virtual-fs
```

Možete ga koristiti kao bilo koji drugi disk:

```
$ cd /tmp/virtual-fs
$ echo ‘Hello, world!’ > test.txt
$ ls -l
total 13
drwxr-xr-x  2  root  root  12288 Mar  8 02:00  lost+found
-rw-rw----  1  root  root     14 Mar  8 02:12  test.txt
$ cat test.txt
Hello, world!
```

Uočite da je direktorijum `lost+found` automatski dodat od strane mke2fs. Ako sistem datoteka ikada bude oštećen i neki podaci budu obnovljeni, ali ne i povezani sa datotekom, oni se smeštaju u direktorijum `lost+found`.

Kada ste završili, de-montirajte virtuelni sistem datoteka.

```
$ cd /tmp
$ umount /tmp/virtual-fs
```

Možete izbrisati `disk-image` ako želite, ili ga kasnije montirati da biste pristupili datotekama na virtuelnom sistemu datoteka. Takođe, možete ga kopirati na drugi računar i tamo ga montirati - celokupan sistem datoteka koji ste stvorili ostaće nepromenjen.

Umesto kreiranja sistema datoteka "od nule", možete kopirati jedan direktno sa uređaja. Na primer, možete napraviti image (identičnu kopiju) sadržaja CD-ROM-a jednostavnim kopiranjem sa CD-ROM uređaja.

Ako posedujete IDE CD-ROM uređaj koristite odgovarajuće ime uređaja, kao što je `/dev/hda`, opisan prethodno. Ako imate SCSI CD-ROM uređaj ime uređaja će biti `/dev/scd0`, ili slično. Vaš sistem takođe može imati i simbolički link /dev/cdrom koji pokazuje na odgovarajući uređaj. Konsultujte vašu `/dev/fstab` datoteku da biste odredili koji uređaj odgovara vašem CD-ROM drajvu.

Jednostavno iskopirajte taj uređaj u datoteku. Rezultujuća datoteka će biti kompletna kopija diska sistema datoteka na CD-ROM drajvu - na primer:

```
$ cp /dev/cdrom /tmp/cdrom-image
```

Ovo može potrajati nekoliko minuta zavisno od CD-ROM-a koji kopirate i brzine vašeg drajva. Rezultujući image može biti poprilične veličine - onolike kolike je i sadržaj na CD-ROM-u.

Sada možete montirati ovaj CD-ROM image bez potrebe za posedovanjem originalnog CD-ROM-a u drajvu. Na primer, da biste ga montirali na /mnt/cdrom, koristite ovu komandu:

```
$ mount -o loop=/dev/loop0 /tmp/cdrom-image /mnt/cdrom
```

Pošto je image na čvrstom disku performanse će biti mnogo bolje nego na samom CD-ROM-u. Uočite da većina CD-ROM-ova koristi iso9660 sistem datoteka.

Kvazi-terminali (PTY)

Ako izvršite mount komandu bez argumenata komandne linije, koja prikazuje sisteme datoteka montirane na vašem sistemu, primetićete liniju koja izgleda otprilike ovako:

```
none on /dev/pts type devpts (rw,gid=5,mode=620)
```

Ovo govori da je specijalni tip sistema datoteka, `devpts`, montiran na `/dev/pts`. Ovaj sistem datoteka, koji nije povezan ni sa jednim hardverskim uređajem, je takozvani "magični" sistem datoteka koji je napravljen od strane Linux-ovog jezgra. Sličan je `/proc` sistemu datoteka

Kao i `/dev` datoteka, `/dev/pts` sadrži nodove koji odgovaraju uređajima. Ali za razliku od `/dev`, koji je običan direktorijum, `/dev/pts` je specijalni direktorijum koji je kreiran dinamički od strane jezgra Linux-

a. Sadržaj direktorijuma može da varira s vremenom i u skladu je sa stanjem sistema na kome je.

Nodovi u `/dev/pts` odgovaraju kvazi-terminalima (ili kvazi-TTY, ili PTY). Linux kreira PTY za svaki novi terminalski prozor koji otvorite i prikazuje odgovarajući nod u `/dev/pts`. PTY uređaj se ponaša kao terminalski uređaj - on prihvata unos sa tastature i prikazuje tekstualni izlaz iz programa koji su na njemu pokrenuti. PTY uređaji su numerisani i PTY broj je ime odgovarajućeg noda u `/dev/pts`.

Možete prikazati terminalski uređaj povezan sa procesom koristeći ps komandu. Odredite tty kao jedno od polja proizvoljnog formata sa `-o` opcijom. Da biste prikazali ID (identifikacija) procesa, TTY i komandnu liniju svakog procesa koji dele isti terminal, pokrenite komandu:

```
$ ps -o pid,tty,cmd
PID    TT     CMD
28832  pts/4  bash
29287  pts/4  ps -o pid,tty,cmd
```

Konkretno ovaj terminalski prozor je pokrenut u PTY 4.

PTY ima odgovarajući nod u /dev/pts:

```
$ ls -l /dev/pts/4
crw–w----  1  stojko  tty  136, 4  Mar 8  02:56  /dev/pts/4
```

Uočite da je to uređaj "bajt-po-bajt" i da je njegov vlasnik takođe vlasnik i procesa za koji je kreiran.

Možete izčitati ili upisati na PTY uređaj. Ako izčitavate, otećete unos sa tastature koji bi inače bio poslat programu koji je pokrenut u PTY-u. Ako upišete na njega podaci će se pojaviti u tom prozoru.

Pokušajte da otvorite novi terminalski prozor i da odredite njegov PTY broj tako što ćete pozvati ps -o pid, tty, cmd. Iz drugog prozora, upišite neki tekst na PTY uređaj. Na primer, ako je PTY broj novog prozora 7, pozovite sledeću komandu u drugom prozoru:

```
$ echo ‘Hello, other window!’ > /dev/pts/7
```

Izlaz će se pojaviti u novom terminalskom prozoru. Ako zatvorite novi terminalski prozor, unos 7 u `/dev/pts` će nestati.

Ako povovete ps komandu da biste odredili TTY iz tekstualnog moda virtuelnog terminala (pritisnite CTRL+ALT+F1 da biste se prebacili na prvi virtuelni terminal, na primer), videćete da je pokrenut u običnom terminalskom uređaju umesto u PTY-u (kvazi-terminalski uređaj):

```
$ ps -o pid,tty,cmd
```

```
PID    TT    CMD
29325  tty1  bash
29353  tty1  ps -o pid,tty,cmd
```

Sistemski poziv ioctl

Sistemski poziv `ioctl` je univerzalni interfejs za kontrolu hardverskih uređaja. Prvi argument za ioctl je deskriptor datoteke koji bi trebalo otvoriti za uređaj koji želite da kontrolišete. Drugi argument je šifra pristupa koja ukazuje na operaciju koju želite da izvršite. Razne šifre pristupa su raspoložive za različite uređaje. Zavisno od šifre pristupa, mogu postojati dodatni argumenti koji dostavljaju podatke za `ioctl`.

Mnoge od raspoloživih šifara pristupa za razne uređaje su izlistane u ioctl_list stranicama uputstva. Korišćenje `ioctl` generalno zahteva detaljno razumevanje drajvera uređaja koji odgovara hardverskom uređaju koji želite da kontrolišete. Većina ovih je vrlo specijalizovana i van opsega materijala u ovoj knjizi. Međutim, pokazaćemo vam jedan primer da bismo vam nagovestili za šta se koristi `ioctl`.

```
#include <fcntl.h>
#include <linux/cdrom.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
   /* Otvoriti deskriptor datoteke na uređaj koji je
   specificiran na komandnoj liniji. */
   int fd = open (argv[1], O_RDONLY);

   ioctl (fd, CDROMEJECT); // Izbaciti CD-ROM.

   close (fd); // zatvoriti deskriptor datoteke

   return 0;
}
```

Listing 12.2 (cdrom-eject.c) Izbacivanje CD-ROM medijuma

Listing 12.2 prezentuje kratak program koji izbacuje disk u CD-ROM drajvu (ako sam drajv ovo podržava). Potreban je samo jedan argument komandne linije, CD-ROM uređaj. On otvara deskriptor datoteke na uređaj i poziva `ioctl` sa pristupnom šifrom `CDROMEJECT`. Ovaj zahtev,

definisan u zaglavlju `<linux/cdrom.h>`, daje instrukciju uređaju da izbaci disk.

Prevedite program:

```
$ gcc –o cdrom-eject cdrom-eject.c
```

Ako na vašem sistemu postoji IDE CD-ROM uređaj povezan kao master uređaj na sekundarnom IDE kontroleru, odgovarajući uređaj je `/dev/hdc`. Da biste izbacili disk iz drajva, pozovite sledeće:

```
$ ./cdrom-eject /dev/hdc
```

Prilagodite argument vašem sistemu (pogledajte tabelu 12.1) i isprobajte ovaj program.

13. Sistem datoteka /proc

Probajte pozvati komandu mount bez argumenata – ovo prikazuje sisteme datoteka montirane na vaš GNU/Linux računar. Videćete jednu liniju koja izgleda ovako.

```
none on /proc type proc (rw)
```

Ovo je specijalni /proc sistem datoteka. Primetićete da prvo polje none, znači da ovaj sistem datoteka nije povezan ni sa kakvim hardverskim uređajem kao što je npr. hard disk. Datoteke u /proc ne odgovaraju stvarnim datotekama na fizičkom uređaju. Umesto toga, to su "magični" objekti koji se ponašaju kao datoteke ali omogućavaju pristup parametrima, strukturi podataka i statističkim informacijama iz kernela. "Sadržaj" ovih datoteka nisu uvek fiksni blokovi podataka, kao što je slučaj kod običnih datoteka. Umesto toga, njih u letu generiše Linux kernel kada čitate iz datoteke. Konfiguraciju pokrenutog kernela možete izmeniti upisivanjem u određene datoteke u /proc.

Pogledajmo primer:

```
$ ls -l /proc/version
-r--r--r--  1  root  root  0  Jan 17  18:09  /proc/version
```

Primetićete da je veličina datoteke nula; to je posledica činjenice da sadržaj datoteke generiše kernel, pa princip rada veličine datoteke nije primenljiv. Takođe, ako pokrenete sami ovu komandu, primetićete da je vreme poslednje izmene datoteke u stvari tekuće vreme.

Šta je u ovoj datoteci? Sadržaj /proc/version sadrži string koji opisuje broj verzije Linux kernela. Sadrži informacije o verziji koje se mogu dobiti pomoću sistemskog poziva "uname", kao i dodatne informacije kao što je verzija kompajlera koja je korišćena za kompajliranje kernela. Možete čitati iz /proc/version kao iz bilo koje druge obične datoteke. Npr. jednostavan primer prikazivanja sadržaja /proc datoteke dobija se korišćenjem cat komande.

```
$ cat /proc/version
Linux version 2.2.14-5.0 (root@porky.devel.redhat.com) (gcc
version egcs-2.91.
66 19990314/Linux (egcs-1.1.2 release)) #1 Tue Mar 7 21:07:39 EST
2000
```

Različiti upisi u /proc sistem datoteka detaljno su opisani u proc man stranici; da biste ih pogledali pozovite sljedeću komandu.

```
$ man 5 proc
```

U ovom poglavlju, opisaćemo neke mogućnosti /proc sistema datoteka koja mogu biti korisna programerima aplikacija i pokazaćemo

neke primere njihovog korišćenja. Neke mogućnosti `/proc` su korisne i za debagovanje.

Ako ste zainteresovani da saznate više informacija o principu rada `/proc`, pogledajte u izvorni kôd Linux-ovog kernela pod `/usr/src/linux/fs/proc`.

Dobijanje informacija iz /proc

Većina upisa u `/proc` obezbeđuje informacije čitljive ljudima, ali je format ispisa dovoljno jednostavan da bi se mogao parsirati. Npr, `/proc/cpuinfo` sadrži informacije o CPU. Izlaz je tabela vrednosti sa opisom vrednosti kao i kolonom sa vrednostima za date opise.

Npr, ispis bi mogao izgledati ovako:

```
$ cat /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 5
model name : Pentium II (Deschutes)
stepping : 2
cpu MHz : 400.913520
cache size : 512 KB
fdiv_bug : no
hlt_bug : no
sep_bug : no
f00f_bug : no
coma_bug : no
fpu : yes
fpu_exception : yes
cpuid level : 2
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep
mtrr pge mca cmov pat pse36 mmx fxsr
bogomips : 399.77
```

Primer čitanja /proc informacija iz programa

Jednostavan način da dobijete vrednosti iz ovog izlaza je da pročitate podatke iz datoteke i upišete ih u bafer a nakon toga ih parsirate pomoću funkcije `sscanf`. Listing 13.1 (`clock-speed.c`) prikazuje primer kako se to može uraditi. Program sadrži funkciju `get_cpu_clock_speed` koja čita iz `/proc/cpuinfo` u memoriju i daje prvu brzinu CPU časovnika.

```
#include <stdio.h>
#include <string.h>

/* Vraća brzinu sistemskog časovnika procesora u Mhz na osnovu
informacije iz /proc/cpuinfo. Na višeprocesorskoj mašini to se
odnosi na prvi procesor. U slučaju greške vraća se nula. */
float get_cpu_clock_speed ()
{
  FILE* fp;
  char buffer[1024];
  size_t bytes_read;
  char* match;
  float clock_speed;

  // Učitaj ceo sadržaj /proc/info u bafer.
  fp = fopen ("/proc/cpuinfo", "r");
  bytes_read = fread (buffer, 1, sizeof (buffer), fp);
  fclose (fp);

  // Prekini ako čitanje nije uspelo ili ako je bafer mali
  if (bytes_read==0 || bytes_read==sizeof(buffer)) return 0;

  // Stvaranje NULL-završenog stringa buffer
  buffer[bytes_read] = '\0';

  // Lociranje linije koja počinje sa "cpu MHz".
  match = strstr (buffer, "cpu MHz");
  if (match == NULL) return 0;

  /* Učitavanje prethodno dobijenog sadržaja linije u
  promenljivu clock_speed */
  sscanf (match, "cpu MHz : %f", &clock_speed);
  return clock_speed;
}

int main ()
{
```

```
    printf ("CPU: %4.0f MHz\n", get_cpu_clock_speed ());
    return 0;
}
```

Listing 13.1 (clock-speed.c) Dobijanje brzine CPU iz /proc/cpuinfo

Prevedite i pokrenite ovaj program.

```
$ gcc –o clock-speed clock-speed.c
$ ./clock-speed
```

Morate znati da semantika, imena i izlazni formati upisa u sistemu datoteka `/proc` mogu biti promenjene u zavisnosti od revizije Linux kernela. Ako ih koristite u programu, morate biti sigurni da će se program ponašati normalno ako neki od upisa u `/proc` nedostaje ili je formatiran na drugi način.

Upisi procesa

Sistem datoteka `/proc` sadrži direktorijum za svaki proces koji je pokrenut na GNU/Linux sistemu. Naziv svakog direktorija je ID procesa za odgovarajući proces . Ovi direktorijumi nastaju i nestaju dinamički kako se procesi pokreću i završavaju na sistemu. Svaki direktorijum sadrži nekoliko upisa koji obezbeđuju pristup informacijama o pokrenutom procesu. Zbog ovih direktorijuma procesa sistem datoteka `/proc` je i dobio svoje ime.

Svaki proces sadrži sledeće upise:

- `cmdline` sadrži listu argumenata za proces
- `cwd` je simbolički link koji pokazuje na tekući radni direktorijum procesa
- `environ` sadrži okruženje procesa
- `exe` je simbolički link koji pokazuje na memorijsku sliku procesa koji se izvršava
- `fd` je poddirektorijum koji sadrži upise za deskriptore datoteka koje je proces otvorio
- `maps` prikazuje informacije o datotekama mapiranim u adresama procesa. Za svaku mapiranu datoteku, maps prikazuje rang adresa u adresnom prostoru procesa u kojem je datoteka mapirana, dozvole na ovim adresama, naziv datoteke i druge informacije. Tabela maps za svaki proces prikazuje izvršne datoteke pokrenute

u procesu, učitane deljene biblioteke kao i druge datoteke koje je proces mapirao.

- `root` je simbolički link ka root direktoriju svakog procesa. Obično je to simbolički link ka /, sistemskom root direktorijumu. Root direktoriumj za proces može biti promenjen korišćenjem `chroot` poziva ili komande `chroot` .
- `stat` sadrži dosta statusnih i statističkih informacija o procesu. Ovo su isti podaci koji su prikazani u upisu `status` ali u izvornom numeričkom formatu u jednoj liniji. Ovaj format je teško čitljiv, ali može biti pogodan za parsiranje u programima. Ako želite da koristite upis `stat` u vašim programima pogledajte proc man stranu koja opisuje njegov sadržaj.
- `statm` sadrži podatke o memoriji korišćenoj u procesu.
- `status` sadrži dosta statusnih i statističkih informacija prilagođenih čitanju za ljude.
- procesorski ulaz (`cpu entry`) se pojavljuje samo na SMP Linux kernelima. Sadrži padove procesnog vremena (korisničkog i sistemskog) iz procesora.

Primetićete da zbog sigurnosnih razloga, dozvole za neke ulaze su postavljene tako da samo korisnik koji je vlasnik procesa (ili superuser) može da im pristupi.

/proc/self

Jedan dodatni ulaz u `/proc` sistemu datoteka olakšava programima put do pronalaženja informacija o sopstvenim procesima. Ulaz `/proc/self` je simbolički link ka `/proc` direktoriju koji odgovara tekućem procesu. Odredište `/proc/self` linka zavisi od procesa koji ga gleda; svaki proces vidi svoj direktorijum kao odredište linka.

Na primer, program u listingu 13.2 (`get_pid.c`) čita sadržaj `/proc/self` linka da odredi svoj ID procesa (ovo ćemo uraditi samo ilustracije radi; pozivanjem getpid funkcije, ID procesa se mnogo lakše dobija). Ovaj program koristi readlink sistemski poziv, da izvadi odredište simboličkog linka.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

// Uzima PID procesa iz linka /proc/self
```

```
pid_t get_pid_from_proc_self()
{
  char target[32];
  int pid;

  // Čita odredište linka (ono na šta link pokazuje)
  readlink ("/proc/self", target, sizeof (target));

  // Odredište je direktorijum nazvan po odgovarajućem PID-u
  sscanf (target, "%d", &pid);

  return (pid_t) pid;
}

int main ()
{
  printf ("PID dobijen iz /proc/self: %d\n",
    (int) get_pid_from_proc_self ());
  printf ("PID koji vraca funkcija getpid(): %d\n",
    (int) getpid ());

  return 0;
}
```

Listing 13.2 (get-pid.c) Dobijanje ID procesa iz /proc/self

Prevedite i pokrenite program:

```
$ gcc –o get-pid get-pid.c
$ ./get_pid
```

Lista argumenata procesa

Upis `cmdline` sadrži listu argumenata procesa. Argumenti su predstavljeni kao jedan karakter string sa argumentima odvojenim NUL karakterima. Većina string funkcija očekuje da se kompletan karakter string završava sa jednim NUL i da neće imati NUL unutar stringa, tako da morate da tretirate sadržaj na poseban način.

Koristeći `cmdline` upise u `/proc` sistemu datoteka, možemo implementirati program koji ispisuje argumente drugog procesa. Listing 13.3 (`print-arg-list.c`) je takav program; ispisuje listu argumenata procesa za koji je dat ID. S obzirom da može biti nekoliko NUL u sadržaju `cmdline`, pre nege jedan na kraju, ne možemo odrediti dužinu stringa sa `strlen` (koji jednostavno broji karaktere dok ne naiđe na

NUL). Umesto toga, odredićemo dužinu `cmdline` pomoću `read`, koji kao rezultat vraća broj bajtova koji su pročitani.

> Napomena: NUL je karakter sa bpojčanom vrijednošću 0. Ovo se razlikuje od NULL, koji predstavlja deskriptor sa vrednošću 0. U C-u, karakter string se obično prekida sa NUL karakterom. Na primer, karakter string "Hello, world!" zauzima 14 bajtova, zbog toga što se nalazi NUL nakon uzvičnika, koji predstavlja kraj stringa. NULL, sa druge strane, je deskriptor sa vrednošću za koju možete definitivno sigurni da neće odgovarati stvarnom adresnom prostoru u vašem programu. U C-u i C++, NUL se predstavlja kao karakter konstanta '\0' ili (char) 0. Definicija NUL-a zavisi od operativnog sistema; na Linuxu, definiše se kao ((void*)0) ili jednostavno 0 u C++.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

// Ispisuje listu argumenata procesa na osnovu datog PID-a
void print_process_arg_list (pid_t pid)
{
  int fd;
  char filename[24];
  char arg_list[1024];
  size_t length;
  char* next_arg;

  // Stvaranje imena cmdline datoteke procesa
  snprintf (filename, sizeof (filename), "/proc/%d/cmdline",
    (int) pid);

  // Čitanje sadržaja datoteke
  fd = open (filename, O_RDONLY);
  length = read (fd, arg_list, sizeof (arg_list));
  close (fd);

  // Čitanje ne stvara NUL-završen bafer, dodajemo NUL
  arg_list[length] = '\0';

  // Prolazak kroz argumente koji su odvojeni NUL karakterima
```

```
  next_arg = arg_list;
  while (next_arg < arg_list + length)
  {
    /* Štampanje argumenata. Sada su svi NUL-završeni pa se
    tretiraju kao običan string */
    printf ("%s\n", next_arg);

    /* Prelazak na sledeći argument. Pošto je svaki argument
    NUL-završen, strlen broji dužinu sledećeg argumenta,
    a ne cele liste argumenata */
    next_arg += strlen (next_arg) + 1;
  }
}

int main (int argc, char* argv[])
{
  pid_t pid = (pid_t) atoi (argv[1]);
  print_process_arg_list (pid);
  return 0;
}
```

Listing 13.3 (print-arg-list.c) Ispisuje listu argumenata pokrenutog procesa

Prevedite i pokrenite program. Pod pretpostavkom da je Na proces čiji je PID 372 sistemski program za logovanje, syslogd:

```
$ gcc –o print-arg-list print-arg-list.c
$ ps 372
PID TTY STAT TIME COMMAND
372 ? S 0:00 syslogd -m 0
$ ./print-arg-list  372
syslogd
-m
0
```

U ovom slučaju, syslogd je pozvan sa argumentima `–m 0`.

Okruženje procesa

Upis `environ` sadrži okruženje procesa. Kao i u `cmdline`, pojedinačne promenljive okruženja su odvojene NUL karakterom. Format svakog elementa je isti kao onaj koji je korišćen u promenljivoj `environ` nazvanoj `VARIABLE=value`.

Ova verzija ID broj procesa sa komandne linije i ispisuje okruženje procesa, čitajući ga iz upisa `/proc`.

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

/* Ispisivanje okruženja procesa datog njegovim PID-om:
svaku promenljivu okruženja u posebnoj liniji */
void print_process_environment (pid_t pid) {
  char filename[24]; char environment[8192];
  int fd; size_t length; char* next_var;

  // Generisanje imena datoteke okruženja (environ file)
  snprintf (filename, sizeof (filename), "/proc/%d/environ",
    (int) pid);

  // Čitanje sadržaja datoteke
  fd = open (filename, O_RDONLY);
  length = read (fd, environment, sizeof (environment));
  close (fd);

  // Čitanje ne stvara bafer NUL-završenim, pa to radimo ovde
  environment[length] = '\0';

  // Prolazak petljom kroz promenljive odvojene NUL karakterom
  next_var = environment;
  while (next_var < environment + length) {
    /* Ispis promenljivih. Svaka je NUL-završena, pa je
    tretiramo kao običan string */
    printf ("%s\n", next_var);

    /* Prelazak na sledeću promenljivu. Pošto je svaka
    promenljiva NUL-završena, strlen broji dužinu
    promenljive a ne dužinu čutavog niza promenljivih */
    next_var += strlen (next_var) + 1;
  }
}

int main (int argc, char* argv[]) {
```

```
    pid_t pid = (pid_t) atoi (argv[1]);
    print_process_environment (pid);
    return 0;
}
```

Listing 13.4 (print-environment.c) Prikaz okruženja procesa

Otkucajte naredbu ps, prevedite i pokrenite program:

```
$ ps
```

Iskoristite neki od prikazanih identifikatora procesa (pid) za prvi argument u komandnoj liniji prilikom pokretanja programa.

```
$ gcc –o print-enviroment print-enviroment.c
$ ./print-enviroment pid
```

Izvršne datoteke procesa

Upis exe pokazuje na izvršne datoteke pokrenute u procesu. Već smo objasnili da je obično izvršno ime programa prosleđeno kao prvi argument u listi argumenata. Primetićete, međutim da je ovo konvencionalno, program može biti pozvan bez liste argumenata. Korištenje exe upisa u sistemu datoteka mnogo je sigurniji način da odredite koje su izvršne datoteke pokrenute.

Jedna od korisnih tehnika je izdvajanje putanje do izvršnih datoteka iz /proc sistema datoteka. Za većinu programa, programske datoteke su instalirane u direktorijumima relativnim putanjama poznatim izvršnim datotekama glavnog programa, pa je neophodno odrediti gdje se te izvršne datoteke nalaze. Funkcija get_executable_path u Listingu 13.5 (get-exe-path.c) određuje putanju izvršnih datoteka u pozvanim procesima, ispitujući simbolički link /proc/self/exe.

```
#include <limits.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

/* Nalaženje putanje izvršne datoteke trenutno pokrenutog
procesa. Putanja je smeštena u promenljivoj BUFFER koja ima
dužinu LEN. Funkcija vraća broj karaktera u putanji ili -1
u slučaju greške */
```

```
size_t get_executable_path (char* buffer, size_t len)
{
  char* path_end;
  // Čita odredište linka /proc/self/exe.
  if (readlink("/proc/self/exe", buffer, len) <= 0) return -1;
  /* Pronađi poslednje pojavljivanje znaka '/'
  (separatora putanje) */
  path_end = strrchr (buffer, '/');
  if (path_end == NULL) return -1;
  // Pomeranje na karakter posle poslednjeg separatora putanje
  ++path_end;
  /* Dobijanje direktorijuma koji sadrži program dodavanjem
  znaka za stvaranje stringa posle posledenjeg znaka '/' */
  *path_end = '\0';
  // Dužina putanje je broj karaktera do zadnjeg znaka '/'
  return (size_t) (path_end - buffer);
}
int main ()
{
  char path[PATH_MAX];
  get_executable_path (path, sizeof (path));
  printf ("Ovaj program je u direktorijumu %s\n", path);
  return 0;
}
```

Listing 13.5 (get-exe-path.c) Dobijanje putanje izvršne datoteke trenutno pokrenutog programa

Prevedite i pokrenite program:

```
$ gcc –o get-exe-path get-exe-path.c
$ ./get-exe-path
```

Deskriptori datoteka u procesima

Upis `fd` je poddirektorijum, koji sadrži upise za deskriptore datoteka koje je pokrenuo proces. Svaki upis je simbolički link ka datoteci ili uređaju za koji je otvoren deskriptor datoteke. Možete čitati ili pisati u ove simboličke linkove; ovo upisuje ili čita iz odgovarajuće datoteke ili

drajvera kojeg je otvorio proces. Upisi u `fd` poddirektorijumu su nazvani po broju deskriptora:

```
$ ps
PID   TTY    TIME      CMD
1261  pts/4  00:00:00  bash
2455  pts/4  00:00:00  ps
```

U ovom slučaju, shell (`bash`) je pokrenut u procesu 1261. Sada otvorite drugi prozor, i pogledajte sadržaj `fd` poddirektorijuma za taj proces.

```
$ ls -l /proc/1261/fd
total 0
lrwx------ 1 stojko stojko 64 Jan 30 01:02 0 -> /dev/pts/4
lrwx------ 1 stojko stojko 64 Jan 30 01:02 1 -> /dev/pts/4
lrwx------ 1 stojko stojko 64 Jan 30 01:02 2 -> /dev/pts/4
```

(Može se desiti da bude još neka linija pri ispisu koja odgovara drugim deskriptorima datoteka). Deskriptori datoteka 0, 1 i 2 su respektivno inicijalizovani za standardni ulaz, izlaz i grešku. Stoga, upisivanjem u `/proc/1261/fd/1`, možete upisati u drajver priključen na stdout za shell proces – u ovom slučaju pseudo TTY u prvom prozoru. U drugom prozoru probajte upisati poruku u tu datoteku:

```
$ echo "Hello, world." >> /proc/1261/fd/1
Tekst će se pojaviti u prvom prozoru.
```

Deskriptor datoteka pored standardnog ulaza, izlaza i greške se pojavljuje u fd poddirektorijumu. U listingu 13.6 dat je program `open-and-spin.c` koji jednostavno otvara deskriptor datoteke za datoteku definisanu u komandnoj liniji i ponavlja tu operaciju u krug.

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  const char* const filename = argv[1];
  int fd = open (filename, O_RDONLY);
  printf ("in process %d, file descriptor %d is open to %s\n",
  (int) getpid (), (int) fd, filename);
```

```
  while (1);
  return 0;
}
```

Listing 13.6 (open-and-spin.c) Otvori datoteku i pročitaj

Prevedite program:

```
$ gcc –o open-and-spin open-and-spin.c
```

Probajte da pokrenete program u jednom prozoru.

```
$ ./open-and-spin /etc/fstab
in process 2570, file descriptor 3 is open to /etc/fstab
```

U drugom prozoru, pogledajte u `fd` poddirektorijum koji odgovara ovom procesu u `/proc`:

```
$ ls -l /proc/2570/fd
total 0
lrwx------ 1 stojko stojko 64 Jan 30 01:30 0 -> /dev/pts/2
lrwx------ 1 stojko stojko 64 Jan 30 01:30 1 -> /dev/pts/2
lrwx------ 1 stojko stojko 64 Jan 30 01:30 2 -> /dev/pts/2
lr-x------ 1 stojko stojko 64 Jan 30 01:30 3 -> /etc/fstab
```

Primetićete da je ulaz za deskriptor datoteke, povezan na datoteku `/etc/fstab` koja je otvorena na ovom deskriptoru. Deskriptori datoteka mogu biti otvoreni na soketima i pajpovima. U takvim slučajevima odredište simboličkog linka koji odgovara deskriptoru datoteke će označiti soket ili pajp umesto pokazivanja na običnu datoteku ili uređaj.

Statistika memorije u procesima

Ulaz `statm` sadrži listu od sedam brojeva, odvojenih prazninom. Svaki broj predstavlja broj strana u memoriji koje proces koristi u određenoj kategoriji. Kategorije koje su poređane po pojavljivanju prikazane su ovde:

- Ukupna veličina procesa
- Veličina procesa rezidentna u fizičkoj memoriji
- Memorija deljena sa drugim procesima, to je memorija mapirana sa ovim procesom i bar sa još jednim procesom (kao što su deljene biblioteke ili nedirnute kopirane strane)
- Veličina teksta u procesu – tj. veličina učitanog izvršnog koda

- Veličina deljenih biblioteka mapiranih u proces
- Memorija korišćena u procesu za stek
- Broj "prljavih" strana – tj. strana u memoriji koje je izmenio program

Upis statm sadrži razne informacije o procesu, formatirane tako da budu razumljive ljudima. Ove informacije sadrže ID procesa i ID procesa roditelja, stvarne i efektivne korisničke i grupne ID-e, memoriju, maske bitova koji definišu koji signali su uhvaćeni, ignorisani ili blokirani.

Informacije o hardveru

Nekoliko drugih upisa u /proc sistem datoteka obezbeđuju pristup informacijama o hardveru sistema. Iako su one uglavnom interesantne za konfiguratore i administratore sistema, te informacije mogu povremeno biti od koristi takođe i za programere aplikacija. Ovde ćemo predstaviti neke od interesantnijih.

Informacije o centralnom procesoru (CPU)

Kao što je ranije prikazano, `/proc/cpuinfo` sadrži informacije o CPU ili o procesorskom vođenju GNU/Linux sistema. Procesorsko polje izlistava procesorski broj; to je 0 za jedno-procesorske sisteme. Prodavac, CPU porodica, model i polja sa koracima omogućavaju vam da odredite tačan model i reviziju CPU-a. Izuzetno korisna polja sa zastavicom (flag) pokazuju koje su CPU zastavice postavljene, što ukazuje na karakteristike dostupne u ovom CPU. Na primer "mmx" ukazuje na dostupnost proširenih MMX instrukcija.

Većina informacija koja se vrati iz `/proc/cpuinfo` je izvedena iz cpuid x86 skupa instrukcija. Ova instrukcija ja mehanizam niskog nivoa pomoću koga program dobija informacije o CPU. Za bolje razumevanje učinka `/proc/cpuinfo` pogledajte dokumentaciju o cpuid instrukcijama u Intelovom "IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference". Ovaj vodič je dostupan na `http://developer.intel.com/design`.

Poslednji element, bogomips, je vrednost specifična za Linux. To je mera procesorskog brzog okretanja u tesnoj (uskoj) petlji i zbog toga je prilično loš (siromašan) indikator ukupne brzine procesora.

Informacije o uređaju

Datoteka /proc/devices izlistava glavne brojeve uređaja za karakter i blok uređaje dostupne sistemu.

Informacije o PCI magistrali

Datoteka /proc/pci izlistava kratak pregled uređaja povezanih sa PCI magistralom ili magistralama. Ovo su stvarne PCI ekspanzione ploče (karte) i one takođe mogu da sadrže uređaje ugrađene u sistemsku matičnu ploču, plus AGP grafičku kartu. Izlistavanje uključuje i tip uređaja; osnovne informacije o uređaju i prodavcu; ime uređaja, ako je dostupno; informacije o karakteristikama koje uređaj nudi i informacije o PCI resursima koje koristi uređaj.

Informacije o serijskom portu

Datoteka /proc/tty/driver/serial izlistava informacije o konfiguraciji i statistiku vezanu za serijski port. Serijski portovi su numerisani od 04. Informacije o konfiguraciji serijskih portova mogu se takođe dobiti, kao i modifikovati, koristeći setserial komandu. Ipak, /proc/tty/driver/serial pokazuje dodatne podatke (statistiku) o svakom prekinutom brojanju serijskog porta.

Na primer, ovaj red iz /proc/tty/driver/serial bi mogao opisati serijski port 1 (što bi bio COM2 pod Windows-om):

```
1: uart:16550A port:2F8 irq:3 baud:9600 tx:11 rx:0
```

Ovo ukazuje da serijski port pokreće UART tipa 16550A, da koristi I/O port 0x2f8 i IQR 3 za komunikaciju i radi na 9600 baud-a. Serijski port je primetio 11 prekida u prenosu i 0 prekida u prijemu.

Informacije o kernelu

Mnogi od upisa u /proc obezbeđuju pristup informacijama o pokretanju konfiguracije kernela i njegovom stanju. Neki od ovih upisa su na najvišem nivou /proc; drugi su pod /proc/sys/kernel.

Informacije o verziji

Datoteka /proc/version sadrži dugački niz koji opisuje kernelov serijski broj (broj distribucije) i verziju izrade. Takođe uključuje informacije o tome kako je kernel izrađen: korisnik koji ga je kompajlirao, mašina na kojoj je kompajliran, datum kada je kompajliran, distribuciju kompajlera (verziju) koja je korištena – na prijmer:

```
$ cat /proc/version
Linux version 2.2.14-5.0 (rootŽporky.devel.redhat.com) (gcc
version egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)) #1 Tue
Mar 7 21:07:39 EST 2000
```

Ovo ukazuje na to da sistem koristi 2.2.14 verziju (distribuciju) Linux kernela, koja je kompajlirana sa EGCS verzijom 1.1.2 (EGCS, Eksperimentalni GNU kompajler sistem, preteča je trenutnog GCC projekta.)

Najznačajniji pojmovi u ovom izlazu, ime operativnog sistema i verzija i revizija kernela dostupni su takođe i u odvojenim `/proc` unosima. To su `/proc/sys/kernel/ostype`, `/proc/sys/kernel/osrelease`, i `/proc/sys/kernel/version` .

```
$ cat /proc/sys/kernel/ostype
Linux
$ cat /proc/sys/kernel/osrelease
2.2.14-5.0
$ cat /proc/sys/kernel/version
#1 Tue Mar 7 21:07:39 EST 2000
```

Hostname i ime domena

Unosi `/proc/sys/kernel/hostname` i `/proc/sys/kernel/domainname` sadrže, redom, ime hosta i ime domena kompjutera.

Upotreba memorije

Unos `/proc/meminfo` sadrži informacije o upotrebi memorije u sistemu. Informacije su predstavljene kako za fizičku memoriju, tako i za swap prostor. Prva tri reda predstavljaju ukupnu memoriju, u bajtovima, a redovi ispod, sažetak tih informacija u kilobajtima – na primer:

```
$ cat /proc/meminfo
total: used: free: shared: buffers: cached:
Mem: 529694720 519610368 10084352 82612224 10977280 82108416
Swap: 271392768 44003328 227389440
MemTotal: 517280 kB
MemFree: 9848 kB
MemShared: 80676 kB
Buffers: 10720 kB
Cached: 80184 kB
BigTotal: 0 kB
```

```
BigFree: 0 kB
SwapTotal: 265032 kB
SwapFree: 222060 kB
```

Ovo pokazuje 512 MB fizičke memorije, od koje je otprilike 9 MB slobodno, i 258 MB swap prostora, od kojeg je 216 MB slobodno. U redu koji je vezan za fizičku memoriju, predstavljene su i tri druge vrednosti:

- Kolona `shared` prikazuje ukupnu deljenu (shared) memoriju trenutno dodeljenu sistemu.
- Kolona `buffers` predstavlja memoriju koju Linux koristi kao bafere za blok uređaje.
- Kolona `cached` predstavlja memoriju dodeljenu od strane Linux-a za keš stranica. Ova memorija se koristi za keš pristup mapiranim datotekama.

Možete koristiti `free` komandu za ispis ovih istih podataka o memoriji.

Diskovi i sistemi datoteka

Sistem datoteka `/proc` takođe sadrži informacije o diskovima prisutnim u sistemu i sistemima datoteka postavljenih za njih.

Sistemi datoteka

Ulaz `/proc/filesystems` prikazuje tipove sistema datoteka koje su poznate kernelu.

Treba uzeti u obzir da ova lista nije jako korisna zato što nije potpuna. Sistemi datoteka mogu biti 'loaded' i 'unloaded' dinamično kao kernel moduli. Sadržaj `/proc/filesystem` izlistava samo one tipove sistema datoteka koji su ili statički povezani (linkovani) sa kernelom ili su trenutno učitani. Ostali tipovi sistema datoteka mogu biti dostupni na sistemu kao moduli ali još uvek ne mogu biti učitani.

Diskovi i particije

Sistem datoteka `/proc` sadrži informacije o uređajima povezanim kako sa IDE kontrolorima, tako i sa SCSI kontrolorima. (ako su sadržani u sistemu).

Na tipičnim sistemima poddirektorijum `/proc/ide` može da sadrži jedan ili oba od dva direktorijuma `ide0` i `ide1`, korespondirajući sa

primarnim i sekundarnim IDE kontrolorima na sistemu. Oni sadrže dalje podirektorijume koji korespondiraju sa fizičkim uređajima povezanim sa kontrolorima. Direktorijimi kontrolora ili uređaja mogu biti odsutni ako Linux nije prepoznao ni jedan uređaj koji je povezan.

Pune staze koje korespondiraju sa 4 moguća IDE uređaja predstavljene su u tabeli 13.1.

Uređaj	Poddirektorijum
Primarni Master	`/proc/ide/ide0/hda`
Primarni Slave	`/proc/ide/ide0/hdb`
Sekundarni Master	`/proc/ide/ide1/hdc`
Sekundarni Slave	`/proc/ide/ide1/hdd`

Tabela 13.1. Pune staze koje odgovaraju IDE uređajima

Svaki direktorijum IDE uređaja sadrži nekoliko upisa koji obezbeđuju pristup informacijama o identifikaciji i konfiguraciji uređaja. Nekoliko najkorisnijih prikazani su ovde:

- `model` sadrži identifikacijsku vezu modela uređaja
- `media` sadrži media tip uređaja. Moduće vrednosti su `disc`, `cdrom`, `tape`, `floppy` i `UNKNOWN`
- `capacity` sadrži kapacitet uređaja u 512-bajtnim (byte) blokovima. Treba uzeti u obzir da će za CD ROM uređaje vrednost biti 231 – 1, a ne kapacitet diska u drajvu. Treba uzeti u obzir da vrednost u capacity predstavlja kapacitet celog fizičkog diska; kapacitet sistema datoteka sadržanih u particijama diska će biti manja.

Na primer, ove komande pokazuju kako odrediti tip medija i identifikaciju uređaja za master uređaj na sekundarnom IDE kontroloru. U ovom slučaju, to je Toshiba CD ROM drajv.

```
$ cat /proc/ide/ide1/hdc/media
cdrom
$ cat /proc/ide/ide1/hdc/model
TOSHIBA CD-ROM XM-6702B
```

Ako u sistemu postoje SCSI uređaji, /proc/scsi/scsi sadrži kratak pregled njihovih identifikacionih vrednosti. Na primer, sadržaj bi mogao izgledati ovako:

```
$ cat /proc/scsi/scsi
```

```
Attached devices:
Host: scsi0 Channel: 00 Id: 00 Lun: 00
Vendor: QUANTUM model: ATLAS_V__9_WLS  Rev: 0230
Type: Direct_Access         ANSI SCSI revision: 03
Host: scsi0 Channel: 00 Id: 04 Lun: 00
Vendor: QUANTUM model: QM39100TD-SW  Rev: N491
Type: Direct_Access         ANSI SCSI revision: 02
```

Ovaj kompjuter sadrži jedan jednokanalni SCSI kontroler (označen `scsi0`), sa kojim su povezana dva Quantum disk drajva sa SCSI identifikacijama uređaja 0 i 4.

Upis `/proc/partitions` prikazuje particije prepoznatih disk uređaja. Za svaku particiju, najveći i najmanji broj uređaja, broj 1024-bajtnih (byte) blokova i ime uređaja koji je u vezi sa tom particijom.

Upis `/proc/sys/dev/cdrom/info` prikazuje različite informacije o mogućnostima CD ROM drajvova. Polja su laka za razumevanje:

```
$ cat /proc/sys/dev/cdrom/info
CD-ROM information, Id: cdrom.c 2.56 1999/09/09
Drive name: hdc
Drive speed: 48
Drive # of slots: 0
Can close tray: 1
Can open tray: 1
Can change speed: 1
Can select disc: 0
Can read multisession: 1
Can read MCN: 1
Reports media changed: 1
Can play audio: 1
```

Šta je gde montirano?

Datoteka `/proc/mounts` daje sažetak montiranih sistema datoteka. Svaka linija (red) korespondira sa jednim mount deskriptorom i izlistava montirane uređaje, tačku montaže i druge informacije. Treba uzeti u obzir da `/proc/mounts` sadrži iste informacije kao i obična datoteka `/etc/mtab`, koji se automatski menja sa `mount` komandom.

Ovo su elementi mount deskiptora:

- prvi element u liniji je montirani uređaj. Za specijalne sisteme datoteka kao što je `/proc` sistem datoteka to je none.

- Drugi element je tačka montiranja - mesto u root sistemu datoteka na kojem se pojavljuje sadržaj sistema. Za sam rut sistem, tačka montiraja je izlistana kao / . Za swap drajvove, tačka montiranja je izlistana kao `swap`.
- Treći element je tip sistema datoteka. Trenutno, većina GNU/Linux sistema koristi ext2 sistem datoteka za disk drajvove, ali DOS ili Windows drajvovi mogu biti montirani sa drugim tipovima sistema datoteka, kao što su fat ili vfat. Većina CD-ROM-ova sadrži iso9660 sistem datoteka.
- Četvrti element izlistava flegove montiranja. Ovo su opcije koje su određene kada je mount dodat.

U `/proc/mounts` poslednja dva elementa su uvek 0 i nemaju značenje.

Zaključane datoteke

U poglavlju 11 opisano je kako se koristi `fcntl` sistemski poziv da bi se manipulisalo zaključavanjem datoteka, tj. bravama za čitanje i pisanje (*read and write locks*) na datotekama. Upis `/proc/locks` opisuje sve brave datoteka koje se trenutno ističu u sistemu. Svaki red u ispisu korespondira sa jednom bravom.

Za brave kreirane sa `fcntl`, prva dva upisa u liniji su POSIX ADVISORY. Treći je WRITE ili READ, zavisno od tipa brave. Sledeći broj je proces identifikacija procesa koji drži bravu. Sledeća tri broja, odvojena sa dvotačkama su glavni i sporedni brojevi uređaja na kojem se nalazi datoteka i inode broj, koji locira datoteku u sistemu datoteka. Ostatak linije izlistava vrednosti unutar kernela koje nisu od generalne koristi.

Pretvaranje sadržaja `/proc/locks` u korisnu informaciju zahteva nešto istraživačkog rada. Možete gledati `/proc/locks` u akciji, tako što ćete pokrenuti program iz listinga 11.4 (`lock-file.c`) da bi se kreirala brava za pisanje (write lock) na datoteci `/tmp/test-file`.

```
$ touch /tmp/test-file
$ ./lock-file /tmp/test-file
file /tmp/test-file
opening /tmp/test-file
locking
locked; hit enter to unlock...
```

U drugom prozoru, pogledajte sadržaj `/proc/locks`

```
$ cat /proc/locks
1: POSIX  ADVISORY  WRITE 5467 08:05:181288 0 2147483647
```

```
00000000 dfea7d40 00000000 00000000
```

Može da bude i drugih linija ispisa koje takođe korespondiraju bravama koje drže drugi programi. U ovom slučaju, `5467` je identifikacija procesa `lock-file` programa.

```
$ ps 5467
PID   TTY      STAT  TIME  COMMAND
5467  pts/28   S     0:00  ./lock-file /tmp/test-file
```

Zaključana datoteka `/tmp/test-file` nalazi se na uređaju koji ima sporedni i glavni broj uređaja, redom, 8 i 5. Ovi brojevi korespondiraju sa `/dev/sda5`.

```
$ df /tmp
Filesystem  1k-blocks  Used     Available  Use%  Mounted on
/dev/sda5   8459764    5094292  2935736    63%
$ ls –l /dev/sda5
brw-rw----  1  root  disk  8, 5  May 5 1998  /dev/sda5
```

Sama datoteka `/tmp/test-file` je na inode 181,288 na tom uređaju.

```
$ ls --inode /tmp/test-file
181288 /tmp/test-file
```

Statistika sistema

Dva upisa u `/proc` sadrže korisnu statistiku sistema. Datoteka `/proc/loadavg` sadrži informacije o opterećenju sistema (*system load*). Prva tri broja predstavljaju broj aktivnih zadataka na sistemu – procesi koji trenutno teku – prosečno u proteklih 1,5 do 15 minuta. Sledeći unos pokazuje trenutni tekući broj pokrenutih zadataka – procesa koji su trenutno planirani da se pokrenu pre nego da budu blokirani u sistemskom pozivu – i ukupan broj procesa u sistemu. Poslednji unos je identifikacija procesa koji je poslednji bio u toku.

Datoteka `/proc/uptime` sadrži dužinu vremena od kada je sistem podignut, kao i vreme u okviru tog vremena u kojem sistem nije bio aktivan (*idle*). Oba vremena su data kao 'floating-point' vrednosti, u sekundama.

```
$ cat /proc/uptime
3248936.18 3072330.49
```

Program `print-uptime.c` u listingu 13.7 izvlači proteklo vreme od podizanja sistema (*uptime*) i vreme neaktivnosti (*idle time*) iz sistema i prikazuje ih u razumljivim jedinicama.

```
#include <stdio.h>

/* Promenljiva time predstavlja količinu vremena u sekundama,
a promenljiva label predstavlja opisnu labelu */
void print_time (char* label, long time)
{
  // Konvertovanje konstanti
  const long minute = 60;
  const long hour = minute * 60;
  const long day = hour * 24;

  // Stvaranje izlaza
  printf ("%s: %ld days, %ld::%02ld:%02ld\n", label,
          time / day, (time % day) / hour,
          (time % hour) / minute, time % minute);
}

int main ()
{
  FILE* fp;
  double uptime, idle_time;

  // Čitanje vremena uptime i idle time iz /proc/uptime
  fp = fopen ("/proc/uptime", "r");
  fscanf (fp, "%lf %lf\n", &uptime, &idle_time);
  fclose (fp);

  // Ispis
  print_time ("uptime" , (long) uptime);
  print_time ("idle time", (long) idle_time);

  return 0;
}
```

Listing 13.7 (print-uptime.c) Ispisivanje vremena uptime i idle time

Prevedite i izvršite program:

```
$ gcc –o print-uptime print-uptime.c
$ ./print-uptime
```

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] A. M. Lister, R. D. Eager, "*Fundamentals of Operating Systems*", Fifth Edition, The Macmillan Press Ltd, 1993.

[4] S. Rago, "*UNIX System V Network Programming*", Addison-Wesley, 1993.

[5] A. S. Tannenbaum, A. S. Woodhull, "*Operating System Design and Implementation*", Second Edition, Prentice Hall, 1997

[6] J. Gray, "*Interprocess Communication in UNIX*", Prentice Hall, 1997.

[7] N. Rhodes, J. McKeehan, "*Understanding the Linux Kernel*", O'Reily and Associates, 1999.

[8] W. Stallings, "*Operating Systems*", Fourth Edition, Prentice Hall, 2000.

[9] M. Bar, "*Linux Internals*", McGraw-Hill, 2000.

[10] A. Danesh, "*Red Hat Linux*", Mikro knjiga, 2000.

[11] M. Mitchell, J. Oldham, A. Samuel, "*Advanced Linux Programming*", New Riders Publishing, 2001.

[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: teorija, praksa i rešeni zadaci*", Mikro knjiga, 2005.