1. Procesi

Deo programa koji je u stanju izvršavanja naziva se PROCES. Ukoliko imate dva terminal windows pokazana na vašem monitoru, onda je najverovatnije reč o izvšavanju istog terminalnog programa dvaputa – ali postoje dva terminalna procesa. Svaki terminalni prozor se najverovatnije izvršava u shellu. Svaki shell koji je u stanju izvršavanja je novi proces. Kada pozivate komandu iz shell-a odgovarajući program se izvršava u novom procesu; kada se proces završi shell proces nastavlja sa radom.

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, su slične onima iz drugih UNIX sistema. Većina njih je deklarisana u datoteci koja predstavlja zaglavlje <unistd.h>, ali treba proveriti uputstvo, za svaku od funkcija da biste bili sigurni.

1.1. Traženje procesa

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

1.1.1 Identifikator Procesa (ID)

Svaki proces na Linux sistemima identifikuje se pomoću svog jedinstvenog Identifikatora procesa (ID), ponekad označavanog sa PID. Identifikatori procesa su 16-to bitni brojevi koje sekvencijalno dodeljuje Linux, novonastalim procesima.

Svaki proces takođe ima svog "roditelja" (osim posebnog INIT procesa, opisanog u poglavlju 1.4.3.) Zbog toga, za procese u Linux sistemu se može reći da su organizovani kao stablo, gde proces init predstavlja koren (root). Identifikator roditeljskog procesa PPID, je jednostavno identifikator procesa pocesovog roditelja.

Kada se pominje Identifikator procesa u programskom jeziku C ili C++, uvek treba koristiti sintaksu pid_t, koja je definasana 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 primeru 1.1. možemo program koji prikazuje svoj Identifikator procesa i identifikator roditeljskog procesa.

Primer 1.1. (print-pid.c) Prokazivanje Identifikatora Procesa

```
#include <stdio.h>
#include <unistd.h>
int main ()
{
        printf ("Identifikator procesa je %d\n" ,  (int) getpid());
        printf ("Identifikator roditeljskog procesa je %d\n" ,  (int) getppid());
        return 0;
}
```

Primetite da ukoliko pozovete program nekoliko puta, dobićete različite vrednosti za Identifikator procesa, zato što je svaki novi poziv u nekom novom procesu. Međutim, ukoliko pravite pozive svaki put iz istog shell-a, Identifikator roditeljskog procesa (odnosno Identifikator proces za shell) je isti.

1.1.2. Pregled Aktivnih Procesa

Naredba ps prikazuje procese koji su u stanju izvršavanja na vašem sistemu. GNU/Linux verzija naredba 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 informacije će prikazati o svakom od tih procesa.

Standardno, pozivanjem komande ps, prikazuju se procesi koje kontrolišu terminal ili terminalni window-a u kojima je komanda ps pozvana. Na primer:

```
%       ps
        PID     TTY     TIME            CMD
21693   pts/8   00:00:00        bash
21694   pts/8   00:00:00        ps
```

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

Da biste dobili detaljniji uvid o tome šta je u stanju izvršavanja na vašem GNU/Lunux sistemu

pozovite sledeće:

% ps –e –o pid, ppid, command

Opcija –e nalaže naredbi ps da prikaže sve procese koji koji su trenutno u stanju izvršavanja u sistemu. 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.

PS IZLAZNI FORMATI

Pomoću –o opcije za naredbu ps, možete odrediti koje infomacije o procesima vi želite na izlazu kao listu odvojenu zarezima (comma-separator). Na primer, ps –o pid, user, start_time, command prikazuje Identifikator procesa, ime vlasnika procesa, vreme startovanja procesa i naredbu koja pokreće proces. Pogledajte uputstvo da bi videli sve opcije naredbe ps. Možete koristiti –f (kompletan listing), -l (dugački listing) ili –j (listing poslova) opciju umesto da dobijete tri različita formata tekućeg listinga.

Evo nekoliko prvih linija i nekoliko poslednjih izlaza ove naredbe na našem sistemu. Vi možete dobiti drugačiji prikaz, što zavisi od toga šta je u stanju izvršavanja na vašem sistemu.

%ps –e –o pid, ppid, command

PID	PPID	COMMAND
1	0	init[5]
2	1	[kflushd]
3	1	[kupdate]
...		
21725	21693	xtrem
21727	21725	bash
21728	21727	ps –e –o pid, ppid, command

Primetite da roditeljski Identifikator procesa naredbe ps, čiji je broj 21727, predstavlja Identifikator procesa bash-a, odnosno shell-a iz kojeg smo pozvali naredbu ps. Roditeljski Identifikator procesa bash-a je 21725, odnosno identifikator procesa za xtem.

1.1.3. Ukidanje procesa

Proces koji je u stanju izvršavanja možete ukinuti naredbom kill. Jednostavno, u komandnoj liniji odredite Identifikator procesa za proces koji će biti ukinut.

Naredba kill radi tako što procesu prosleđuje SIGTERM, odnosno signal za ukidanje.[1] Ovo uzrokuje ukidanje procesa, osim ako to izvršni program eksplicitno radi ili postoji maskiranje za SIGTERM signal. Signali su opisani u poglavlju 1.3. SIGNALI.

1.2. Kreiranje procesa

Postvoje dve uobičajene tehnike za kreiranje novog procesa. Prva je relativno jednostavna ali ona 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.

1.2.1. Korišćenje funkcije system

Funkcija system u standardnoj C biblioteci omogućuje da se na lak način izvrši naredbe 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 shellu na izvešavanje. Na primeru 1.2. prikazan je program koji poziva naredbu ls da bi prikazao sadržaj root (korenog) direktorijuma isto kao da ste ukucali ls –l u shell.

Primer 1.2. (system.c) Upotreba funkcije system

```
#include <stdlib.h>

int main ()
{
int return_value;
return_value = system („ls –l /“);
return return_value;
}
```

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

Zbog toga što funkcija system koristi shell da bi pozvala vašu naredbu, ona je izložena svojstvima, ograničenjima i sigurnosnim propustima sistemskog shella. Ne možete se osloniti na raspoloživost bilo koje verzije Bournovog shella. 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 ratličite verzije bash-a. Pozivajući program koji ima root privilegiju, funkcijom system, na primer, može imati različite rezultate na različitim GNU/Linux sistemima. Zbog toga, preporučuje se korišćenje **fork** i **exec** metoda za kreiranje procesa.

1.2.2. Korišćenje fork i exec metoda

DOS i Windows API sadrže 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 pravi dete-proces koji je istovetna kopija svog roditeljskog procesa. Linux ima drugi skup funkija, exec familija funkcija, koje izazivaju da određeni proces prestaje da bude primerak jednog programa i umesto toga postaje primerak drugog programa. Da bi se stvorio 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 primerak programa koji želite da stvorite.

Pozivanje metoda 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.

Pa 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 i dete-procese - jedan „ulazi“ u fork poziv a dva procesa „izlaze“, sa različitim vraćenim vrednostima. Vrednost koja je vraćena za roditeljski proces predstavlja vrednost Identifikator procesa za dete-proces. Vrednost koja je vraćena u dete-proces je nula. Zbog toga što ni jedan proces nikad ne može imati vrednost nula kao svoj Identifikator procesa, ovo je programu lako bilo da se izvršava kao roditeljski ili dete-proces. Primer 1.3. koristi funkciju fork da bi napravio duplikat programskog procesa. Primetite da se prvi blok, if deo, izvršava, samo u roditeljskom procesu, dok else klauzula izvršava u dete-procesu.

Primer 1.3. (fork.c) Korišćenje funkcije fork za dupliciranje programskog procesa

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

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

int main ()
{
        pid_t child_pid;

printf ("Identifikator procesa glavnog programa je: %d\n" , (int) getpid ());

child_pid = fork ();
if (child_pid !=0) {
printf ("ovo je roditeljski proces, ciji je identifikator procesa: %d\n" , (int) getpid ());
printf ("identifikator procesa za dete proces je: %d\n" , (int) child_pid);
}
else
printf ("ovo je dete proces, ciji je identifikator procesa: %d\n" , (int) getpid ());
return 0;
}
```

Upotreba exec familije funkcija

Exec funkcije zamenjuju program koji se izvršava u procesu drugim programom. Kada program pozove exec funkciju, proces momentalno prestaje da se 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 u manjoj meri po njihovim 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; 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-izvršeni niz pokazivača na stringove. 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, niz promenjivih u okruženju. Argument bi trebalo da bude NULL-izvršeni niz pokazivača na karakter stringove. Svaki karakter string bi trebalo da ima oblik "VARIABLE=value".

Zbog toga što funkcija exec zamenjuje pozvani program sa drugim, ona nikad ne vraća vrednost osim ako se dogodi greška.

Lista argumenata koja se prosleđuje programu je analogna sa argumentima komandne -linije koje vi definišete u programu kada je pokrećete iz shella. Oni su raspoloživi preko argc i argv parametara u main-u. 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, vi bi takođe, trebalo da prosledite ime funkcije kao prvi element 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.

Program u primeru 1.4., poput onog u primeru 1.2., lista sadržaj root (korenog) direktorijuma koristeći ls naredbu. Mada, za razliku od prethodnog primera on poziva naredbu ls direktno, prosleđujući joj komandne linije –l i / radije nego da je poziva preko shella.

Primer 1.4. (fork-exec.c) Upotreba funkcija fork i exec zajedno

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

/* Pojavljuje se dete-proces koje izvršava novi progam. PROGRAM je ime programa koji treba izvršiti;
putanja za ovaj program će biti tražena. ARG_list je NULL-izvršena lista sačinjena od karakter
stringova koja će biti prosleđena kao programska lista argumenata. Vrednost koja se vraća je
Identifikator procesa za proces koji se pojavljuje. */
```

```
int spawn (char* program, char** arg_list)
{
pid_t child_pid;

/* Pravi duplikat ovog procesa */
child_pid = fork ();
if (child_pid !=0)
/* Ovo je roditeljski proces */
return child_pid;
else     {
/* Sada izvrši PROGRAM, traži ga u putanji */
execvp (program, arg_list);
/*funkcija execvp vraća samo ako se dogodi greška */
fprintf (stderr, "an error occured in execvp\n");
abort();
}
}
int main ()
{
        /*Lista argumenata se prosleđuje komandi 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 ("završeno sa  main funkcijom\n ");

return 0;
```

}

1.4. Ukidanje procesa

Normalno, proces se ukida na jedan od dva načina. Ili program koji se izvršava poziva funkciju exit, ili se programska funkcija main vraća. Svaki proces ima izlazni kod: broj koji proces vraća svom roditeljskom procesu. Izlazni kod je argument koji se prosleđuje funkciji exit, ili vrednost vraćena iz funkcije main.

Proces se takođe može ukinuti na neuobičajen način, u zavisnosti od signala. Na primer, SIGBUS, SIGSEGV i SIGFPE signali, ranije pomenuti, izazivaju ukidanje 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 šalje signal SIGTERM. Standardno dispoziranje za oba ova 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.

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 sledeće, gde pid predstavlja identifikator tog procesa:

%kill – KILL pid

Da bi se poslao signal iz programa, treba koristiti funkciju kill. Prvi parametar je Identifikator procesa 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 ukinete dete-proces iz roditeljskog procesa na sledeći način:

kill (child_pid, SIGTERM);

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

Po konvenciji, izlazni kod se koristi da pruži informaciju da li je program dobro izvršen. Nula kao izlzni kod ukazuje da je izvršavanje bilo dobro, dok kod 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 &&(logičko i) i || (logičko ili) operatorima. Stoga, vi bi trebalo eksplicitno da vratite nulu iz vaše main funkcije, osim ukoliko se dogodi greška.

Sa većinom shellova, moguće je dobiti izlazni kod većine nedavno izvršenih programa, pomoću specijalne $? promenjive. Ovde je primer u kojem je naredba ls pozvana dva puta i njen izlazni kod je prikazan nakon svakog poziva. U prvom slučaju, naredba ls se izvršava dobro i vraća nulu kao izlazni kod. 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 čak iako je tip parametra funkcije exit int i da iako funkcija main vraća int, Linux ne sadrži punih 32 bita vraćenog koda. 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 kod je 128 plus broj signala.

1.4.1. Čekanje na ukidanje procesa

Ako ste ukucali i pokrenuli Primer 1.4. 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 zbog što je dete-proces, u kojem se ls izvršava, nezavisno raspoređen u odnosu na roditeljski proces. Zbog toga što je Linux multitasking operativni sistem, oba procesa se izvršavaju simultano, i ne može se predvideti da li će ls program moći da se izvršavati pre ili posle roditeljskog procesa.

Ipak, 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 vam omogućavaju da sačekata da proces da završi sa izvršavanjem, i omogućuju da roditeljski proces dobije informacije o ukidanju dete-procesa. Postoje četiri različita sistemska poziva u wait familiji; vi možete odabrati da li ćete dobiti malo ili puno informacija o procesu koji je okončan; i možete birati da li vodite računa o tome koji je dete-proces okončan.

1.4.2. Sistemski poziv wait

Najjednostavnija takva funkcija se jednostavno naziva wait. Ona blokira proces koji je pozvan dok se jedno od njegovih dete-procesa ne izvrši (ili se dogodi greška). Ona vraća statusni kod preko intager pokazivačkog argumenta, iz kojeg možete izvući informaciju o tome kako se dete-proces okončao. Na primer, WEXITSTATUS makro izvlači izlazni kod za dete-proces.

Možete koristiti WIFEXITED makro da bi iz dete-procesa odredili izlazni status bilo da se proces završio normalno (preko exit funkcije ili povratkom iz main-a) ili da je ukinut zbog neobrađenog signala. U ovom drugom slučaju, koristite WTERMSIG makro da bi dobili iz njegovog izlaznog statusa broj signala pomoću kojeg okončan.

Ovde imamo main funkciju koju smo koristili u primeru sa fork i exec. Ovog puta, roditeljski proces poziva wait funkciju da bi sačekao dok dete-proces, u kojem se izvršava naredba ls, završi sa radom.

```
int main ()
{
        int child_status;

        /* Lista argumenata koja se prosleđuje naredbi ls*/
        char* arg_list [] = {
        "ls"            /* argv [0], ime programa*/
        "-l"
        "/"
        NULL            /*lista argumenata se završava sa NULL*/
};

/* Stvara se dete proces koji izvršava naredbu "ls". Ignoriše se vraćeni Identifikator procesa za dete-proces */

spawn ("ls", arg_list);

/* Čeka se da se dete-proces izvrši*/
wait (&child_status);
if (WIFEXITED (child_status))
        printf ("dete-proces je okoncan normalno, sa izlaznom kodom  %d \n",
```

```
        WIFEXITED (child_status));
else
        printf (("dete-proces nije okoncan normalno \n");
return 0;
}
```

Nekoliko sličnih sistemskih poziva je raspoloživo u Linuxu, koji su fleksibilniji ili pružaju više informacija o završetku dete-procesa. Funkcija waitpid može se koristiti kako bi se sačekalo da se ciljani dete-proces završi umesto bilo kog drugog dete-procesa. Funkcija wait3 daje statističke podatke o upotrebi procesora (CPU) za dete-proces, a funkcija wait4 vam dozvoljava definisanje dodatnih opcija o tome koji proces treba da se čeka.

1.4.3. "Zombi" procesi

Ako se dete proces završi dok je roditeljski proces u funkciji wait, dete proces nestaje i njegov izlazni status je vraćen roditelju preko poziva wait. Ali šta se dešava kada se dete-proces izvrši 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, ako je tako koji, je njegov izlazni status – bile izgubljene. Umesto toga, kada se dete-proces okonča, ono postaje zombi proces. Zombi proces je proces koji se izvršio ali koji još nije obrisan. Na roditeljskom procesu je da obriše svoje zombi decu-procese. Funkcija wait radi ovo, takođe, tako da nije potrebno da se prati da li se vaš dete-proces se još uvek izvršava pre nego što se čeka na njega. Predpostavimo, na primer, da program koji vrši fork nad dete-procesom, vrši neka druga računanja, a zati poziva wait. Ako dete proces još uvek nije ukinut na ovom mestu, roditeljski proces će biti blokiran u wait pozivu sve dok se dete-proces ne izvrši. Ako se dete-proces izvrši pre nego što roditeljski proces pozove wait, dete proces postaje zombi proces. Kada roditeljski proces pozove wait, ukidanje zombi dete-procesa počinje, dete-proces se briše, i wait poziv se uklanja momentalno.

Šta se dešava ako roditeljski proces ne ukloni decu-procese? Oni ostaju u sitemu kao zombi procesi. Program u primeru 1.6. radi fork nad dete-procesom, koji se momentalno ukida i biva uspavan na jedan minut, bez brisanja dete-procesa.

Primer 1.6. (zombie.c) Pravljenje zombi procesa

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

```
int main ()
{
pid_t child_pid;

/* stvara se dete porces */

child_pid = fork ();
if (child_pid >0) {
/*Ovo je roditeljski proces. Spava jedan minut */
sleep (60);
}
else {
/*Ovo je dete proces. Izlaz trenutno */
                exit (0); }
return 0;
}
```

Pokušajte da kompajlirate ovu datoteku, koja je izvršnog tipa, nazvanu make-zombie. Pokrenite je i dok se ona još uvek izvršava listajte procese na sistemu pozivanjem sledeće komande u novom prozoru:

```
%ps –e –o pid, ppid, stat, cmd
```

Ovo prikazuje listu Identifikatora procesa, roditeljski Identifikator procesa, status procesa i komandnu liniju procesa. Primetite da se kao dodatak roditeljskom make-zombie procesu, pojavljuje još jedan make-zombie proces. To je dete-proces; primetite da je Identifikator procesa njegovog roditelja je Identifikator procesa glavnog make-zombie procesa. Dete-proces je označen kao <defunct>, i njegov statusni kod je Z, kao Zombi.

Šta se dešava kada glavni make-zombie program završi kada roditeljski proces izađe, uopšte bez pozivanja funkcije wait? Da li zombi proces ostaje tu gde jeste? Ne – pokušajte da pokrenete ps program ponovo i primetite da su oba make-zombie procesa nestala. Kada program izađe, njegovu decu nasleđuje poseban proces, init program, koji za vreme izvršavanja uvek ima vrednost 1 kao Identifikator procesa (to je prvi proces koji se startuje prilikom podizanja Linuxa). Proces init automatski uklanja bilo koji zombi dete-proces koji nasledi.

Vežbe:

Otkucajte naredbu ps –af. Šta se dobilo kao izlaz?

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

Koji proces je označen preko PID = 1?

Napravite krartak program za listanje sistemskih procesa, koji dakle daje isti izlaz kao i konzolna naredba ps –ax. (Uputstvo: u main() funkciji pozvati system(“ps –ax”);

Kakava se razlika postiže ako zadamo narebu ps –ax &. Da li sada moramo čekati da se cela naredba ps završi pre nego što se iz poziva funkcije ps vratimo u glani program?

(Ovo se najbolje vidi ako posle poziva naredbe system napravimo kratak ispis nečega).

Ispišite sledeći program fork1.c:

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

int main()
{
pid_t pid;
char *poruka;
int n;

print(“fork program 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 roditelj process”;
n = 3;
break;
}

for(; n>0; n--)
{
puts(poruka);
sleep(1);
}
exit(0);
}
```

Pokrenite program. Kakav je izlaz? Stavite sada naredbu sleep pod komentar (// sleep(1)).

Da li se promenio redosled ispisa poruka i zašto?

Šta se dešava ako stavimo sleep(2)?

U prethodnom primeru, roditelj proces nije čekao da dobije statusne informacije o kraju rada deteta procesa. To se postiže naredbom wait. Izmeniti predhodni program u wait1.c:

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

int main()
{
pid_t pid;
char *poruka;
int n;
int exit_code;

print(“fork program pocinje…\n”);
```

```
pid = fork();
switch(pid)
{
case -1:
perror("fork nije uspeo!");
exit(1);
case 0:
poruka = "Ovaj proces je dete proces";
n = 5;
exit_code = 37;
break;
default:
poruka = "Ovaj proces je roditelj proces";
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);
```

U programu fork1.c, u naredbi switch zamenimo vrednosti promenljive n (tamo gde je bilo 3 stavimo 5 i obrnuto). Da li se pojavio zombi proces?

Proverimo to naredbom ps –al.