8. Kontrola procesa

8.2 Identifikatori procesa

Svaki proces ima jedinstven *proces ID*, nenegativan ceo broj. Kada se procesi završe, njihovi IDovi postaju kandidati za ponovnu upotrebu. Većina UNIX sistema implementiraju algoritme da odlože ponovnu upotrebu, tako da novokreirani procesi dobijaju IDove različite od skoro završenih procesa. Ovo smanjuje mogućnost pojave grešaka.

Postoje neki specijalni procesi. Proces koji ima proces ID 0 je *planer proces (scheduler)* i obično se naziva *razmenjivač (swapper)*. Ovaj proces je deo kernela. ID 1 ima proces koji se zove **init** i pokreće ga kernel pri kraju procedure startovanja sistema. Program za ovaj proces je /sbin/init. Ovaj proces se ne prekida i on je običan korisnički proces a ne sistemski proces u kernelu, iako ima superuser privilegije. On kreira druge procese koje pokrećemo.

Proces init pripada grupi procesa koje zovemo **demoni**. To su procesi koji se izvršavaju u pozadini, a ne pod direktnom kontrolom korisnika. Najčešće se pokreću pri startovanju sistema, i služe da konfigurišu hardver, reaguju na zahteve preko mreže itd.

```
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);

uid_t getuid(void);
uid_t geteuid(void);

gid_t getgid(void);
gid_t getegid(void);
```

8.3 fork funkcija

Postojeći proces može da kreira novi proces pozivom **fork** funkcije.

```
pid_t fork(void);
```

Novi proces se zove **dete proces**, a proces koji ga je kreirao **roditeljski proces**. Kaže se da se ova funkcija poziva jednom a vraća dvaput. To u stvari znači da se kod programa počev od prve naredbe posle *fork* izvršava dva puta – u detetu procesu i u roditeljskom procesu. Jedina razlika je da je povratna vrednost funkcije *fork* u detetu 0, a u roditeljskom procesu je proces ID deteta. Razlog za ovo je što roditelj može imati više od jednog deteta, a ne postoji funkcija koja omogućava procesu da dobije proces IDove svoje dece. U detetu se vraća 0 jer dete može imati samo jednog roditelja, i uvek može ID tog roditelja dobiti pomocu *getppid* funkcije.

I roditelj i dete nastavljaju izvršavanje od instrukcije koja sledi iza *fork*. Dete je kopija roditelja. Dete dobija kopiju roditeljskog prostora za podatke, hip i stek dok dele isti tekst segment. (Današnje implementacije ne rade potpuno kopiranje vec omogućavaju samo čitanje na zajedničkoj memoriji, a ukoliko je potrebno izmeniti neke podatke, onda se pravi kopija). Ako je memorija dinamički alocirana

pre poziva *fork*, tada i dete i roditelj imaju kopije dinamički alocirane memorije, pa je potrebno uraditi oslobađanje memorije u oba slučaja.

(PRIMER fork)

```
./fork
./fork > izlaz.txt
```

U opštem slučaju, nikada ne znamo da li se prvo izvršava roditelj ili dete. Ovo zavisi od algoritma koji koristi kernel. U prethodnom primeru čekamo 2 sekunde da bi bili sigurni da će se dete završiti pre roditelja. Standardna U/I biblioteka je baferisana. Standardni izlaz je linijski baferisan ukoliko je povezan sa terminalom, a inače je puno baferisan. U slučaju preusmeravanja, *printf* se poziva jednom, pre *fork*, ali linija ostaje u baferu i kopira se u dete proces. Funkcija *printf* na kraju programa nadovezuje novi sadržaj na sadržaj bafera, i kada se procesi završe njihovi baferi se konačno prazne. Tako da se pri preusmeravanju tekst koji se ispisuje pre *fork* poziva prikazuje 2 puta.

Pri forkovanju svi fajl deskriptori koji su otvoreni kod roditelja dupliraju se za dete (kao da je pozvana *dup* funkcija za svaki deskriptor). (Slika 8.2 – APUE) Važno je da roditelj i dete dele isti ofset fajla. To omogućava da roditelj sačeka dete, dete nešto ispiše i posle toga roditelj nadoveže ispis na detetov. Ukoliko roditelj i dete pišu po istom deskriptoru, bez sinhronizacije, njihov izlaz će biti izmešan.

Dva glavna razloga da *fork* ne uspe su: a) već ima previše procesa na sistemu b) maksimalan broj procesa za ovaj real user ID je prekoračen.
Funkcija *fork* se koristi u dva slučaja:

- Kada proces hoće da duplira sebe kako bi roditelj i dete obavljali različite sekcije koda u isto vreme – ovako funkcionišu serveri.
- Kada proces hoće da izvrši drugi program – ovako funkcioniše shell.

8.5 exit funkcije

Nezavisno od toga kako se proces završi, u kernelu se izvršava isti kod. Prvo se zatvaraju svi otvoreni deskriptori za proces, oslobađa memorija koja je korišćena i slično. Roditelj se obaveštava o načinu prekida deteta procesa pomoću exit statusa, a ukoliko je došlo do neuobičajenog prekida, kernel generiše status prekida koji roditelj može dobiti pomoću *wait* ili *waitpid* funkcija.

Šta se dešava ukoliko se roditelj završi pre deteta? Proces *init* nasleđuje to dete (**siroče**), tj. postaje njegov novi roditelj. Ono što se dešava je da bilo kad da se proces završi, kernel prolazi kroz sve aktivne procese da vidi da li je proces koji se završio roditelj nekog procesa koji još postoji. Ukoliko takav dete proces postoji, njegov roditeljski proces ID se postavlja na 1. Ovako se garantuje da svaki proces ima roditelja.

Kada se dete završi pre roditelja, pošto kernel čuva informacije o svakom procesu koji se završava u tabeli procesa, roditelj može da dobije ove informacije pozivom *wait* ili *waitpid*. Proces koji se završio, ali čiji roditelj još nije čekao na njega (nije pozvao *wait* ili *waitpid*) se zove **zombi** (neformalno, postojanje zombija označava "nezainteresovanost" roditelja da sazna kako je završen dete proces). Iako se taj proces završio, informacije o njemu još uvek će postojati u tabeli procesa. Ako neki zombi postoji duži period vremena, onda najverovatnije postoji bag u roditeljskom programu. Ako bi se

roditeljski proces završio a zombi i dalje postojao, to bi značilo da postoji greška u operativnom sistemu. Trebalo bi uvek obezbediti da roditeljski proces čeka na dete proces. Prisustvo par zombija ne ugrožava sistem, ali prisustvo većeg broja može uzrokovati da ne postoje slobodni proces IDovi za nove procese (do problema sa memorijom ne dolazi jer je jedini prostor koji je potreban onaj u tabeli procesa).
(PRIMER init)

8.6 wait i waitpid funkcije

Kada se proces završi, bilo regularno ili neregularno, kernel šalje SIGCHLD signal njegovom roditelju. Podrazumevana akcija za ovaj signal je ignorisanje. Proces koji poziva *wait* ili *waitpid* funkciju može da:

- blokira, ako su sva deca i dalje aktivna
- se vrati momentalno sa statusom prekida deteta, kada je dete završilo sa radom
- se vrati momentalno sa greškom, ukoliko proces nema nijedan dete proces

Ukoliko pozivamo wait jer dolazi do SIGCHLD signala, očekujemo da se wait momentalno vrati. Ali ukoliko je zovemo u proizvoljnom trenutku vremena, ona može da blokira dalje izvršavanje programa.

```
pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int options);
```

Razlike između ove dve funkcije su:

- *wait* funkcija može da blokira pozivaoca dok se dete proces ne završi, dok waitpid ima opciju koja sprečava blokiranje
- *waitpid* ne čeka dete koje se prvo završava, već opcije određuju decu na koje čeka

Ukoliko se dete već završilo i postalo zombi, *wait* se momentalno vraća sa statusom deteta. Inače, blokira pozivaoca dok se dete ne završi. Ukoliko ima više dece, *wait* se vraća kada se bilo koje završi.

Za obe funkcije se u argument *statloc* smešta status prekida procesa. Pojedinačni bitovi označavaju različite stvari, a razloge prekida određujemo preko makroa.

(PRIMER status)
Napomena: Ukoliko funkcija *exit* vrati 0, onda kažemo da se taj proces uspešno završio. Vraćanje neke druge vrednosti označava neuspeh (najčešće 1). Ovo treba razlikovati od normalnog i nenormalnog završetka procesa: svaki završetak procesa pozivom exit funkcije smatra se normalnim završetkom, dok se nenormalnim smatra prekid procesa usled primanja nekog signala.

Makroe možemo videti u TABELI 8.4.

U zavisnosti od argumenta *pid waitpid* funkcija čeka na različite procese:

- pid == -1 čeka na bilo koje dete proces. U ovom slučaju ekvivalentna je *wait* funkciji
- pid > 0 čeka na dete proces ciji ID je jednak pid
- pid == 0 čeka na bilo koje dete proces ciji je proces grup ID jednak onom od pozivajućeg procesa
- pid < -1 čeka na bilo koje dete čiji je proces grup ID jednak apsolutnoj vrednosti *pid*

Argument *options* dalje kontroliše *waitpid*. On je ili 0 ili se dobija bitskim *ili* konstanti iz tabele.
TABELA 8.7

Funkcija *waitpid* omogućava neblokirajuću verziju wait funkcije.

8.9 Race Conditions (Trke za resurse)

RC nastaje kada više procesa pokušava da nešto uradi sa zajedničkim podacima, a ishod zavisi od redosleda kojim se procesi izvršavaju. Pri forkovanju dolazi do ovakvih situacija, kada se ne zna da li će se roditelj ili dete prvo izvršavati. Potencijalni RC je postojao u programu zombie. Ni pozivanje sleep funkcije ništa ne garantuje, jer sistem može biti preopterećen. Ovakvi problemi se teško otkrivaju jer programi u većini slučajeva dobro rade.

Ukoliko želimo da u detetu čekamo na roditelja, to možemo postići naredbom:

```
while (getppid() != 1)
        sleep(1);
```

Problem sa ovakvom vrstom petlje (POLLING) je da troši procesorsko vreme. Da bi se izbegao ovaj problem kao i RC koriste se signali.

(PRIMER race)
Ako se prime race pokrene više puta, trebalo bi da se u nekim slučajevima poruke ispišu pomešano, što označava RC.

Zadatak 1. Napisati program *nadovezivanje.c* koji kreira dete proces i čeka na njega. Roditelj proverava status završetka deteta i samo ako se dete normalno završilo sa statusom 0, roditelj nastavlja sa radom. U detetu se u fajl izlaz.txt ispisuje proces ID deteta, a po završetku deteta se iz roditeljskog procesa vrši ispis proces IDa roditelja u isti fajl. Obavezno je korišćenje open sistemskog poziva.

Zadatak 2. Napisati program *make_zombie.c* koji treba da demonstrira nastajanje zombi procesa na sistemu. Program kreira dete proces koji se momentalno završava. Roditelj NE poziva ni *wait* ni *waitpid* već se uspavljuje na 10 sekundi. Program pokrenuti u pozadini sa *./make_zombie&* i tokom rada programa izvršiti komandu *ps.* Programi koji su postali zombiji uz svoje ime treba da sadrže string *<defunct>*.

Zadatak 3. Napisati program *server_sqrt.c* koji na ulazu prikazuje znak % i učitava cele brojeve. Svaki put kada pročita ceo broj, server kreira dete proces koje pomoću funcije *sqrt* izračunava koren broja i upisujue ga u fajl x...x.txt, gde je x...x identifikator deteta procesa. Roditelj čeka na dete i po završetku deteta iz odgovarajućeg fajla čita rezultat izračunavanja i ispisuje ga na ekran.