# Seminar 3: SURADNJA PROCESA I KRITIČNI ODSJEČCI

## PROGRAMI I PROCESI

Program je skup instrukcija i podataka koji se nalaze u datoteci na disku. U opisu datoteke ona je opisana kao izvršna i njen sadržaj je organiziran prema pravilima jezgre. Sve dok sadržaj datoteke odgovara pravilima i dok je označena kao izvršna, program može biti pokrenut. Da bi se pokrenuo novi program prvo treba (pozivom jezgre) kreirati novi proces koji je okolina u kojem se izvršava program.

Proces se sastoji od tri segmenta: segment instrukcija, segment korisničkih podataka i segment sistemskih podataka. Program inicijalizira segment instrukcija i korisničke podatke. Nakon inicijalizacije više nema čvste veze između procesa i programa koji on izvodi. Proces dobiva sredstva (više memorije, datoteke, itd.) koji nisu prisutni u samom programu, mijenja podatke, itd. Iz jednog programa može se inicijalizirati više procesa koji se paralelno izvode.

### sistemski poziv *fork*

Sistemskim pozivom *fork* zahtijeva se kreiranje novog procesa. Kada proces koji se trenutno izvodi pokrene novi proces, pokrenuti proces postaje "dijete" procesa "roditelja" koji ga je pokrenuo. Dijete dobija kopije segmenta instrukcija i segmenta podataka od roditelja. U stvari, pošto se segment instrukcija normalno ne mijenja, jezgra može uštediti vrijeme i memoriju tako da postavi taj segment kao zajednički za oba procesa (sve dok ga jedan od njih ne odluči inicijalizirati novim programom). Također, dijete nasljeđuje većinu sistemskih podataka od roditelja.

```
int fork(void) ;
```

U ovaj sistemski poziv ulazi jedan proces, a iz njega izlaze dva odvojena procesa ("dijete" i "roditelj") i dobivaju svaki svoju povratnu vrijednost. Proces dijete dobiva rezultat 0, a roditelj dobiva identifikacijski broj procesa djeteta. Ako dođe do greške, vraćena vrijednost je 1, a dijete nije ni kreirano. *fork* nema nikakvih argumenata, pa programer ne može biti odgovoran za grešku već je ona rezultat nemogućnosti jezgre da kreira novi proces zbog nedostatka nekog od potrebnih sredstava.

Dijete nasljeđuje većinu atributa iz segmenta sistemskih podataka kao što su aktualni direktorij, prioritet ili identifikacijski broj korisnika. Manje je atributa koji se ne nasljeđuju:

- Identifikacijski brojevi procesa djeteta i roditelja su različiti, jer su to različiti procesi.
- Proces dijete dobiva kopije otvorenih opisnika datoteka (*file descriptor*) od roditelja. Dakle to nisu isti opisnici datoteka, tj. procesi ih ne dijele. Međutim, procesi dijele kazaljke položaja u datotekama (*file pointer*). Ako jedan proces namjesti kazaljku položaja na određeno mjesto u datoteci, drugi proces će također čitati odnosno pisati od tog mjesta. Za razliku od toga, ako dijete zatvori svoj opisnik datoteke, to nema veze s roditeljevim opisnikom datoteke.
- Vrijeme izvođenja procesa djeteta je postavljeno na nula.

Dijete se može inicijalizirati novim programom (poziv *exec*) ili izvoditi poseban dio već prisutnog programa, dok roditelj može čekati da dijete završi ili paralelno raditi nešto drugo. Osnovni oblik upotrebe sistemskog poziva *fork* izgleda ovako:

```
if (fork() == 0) {
   posao djeteta
   exit(0);
}
```

```
nastavak rada roditelja (ili ništa)
wait(NULL);
```

### Sistemski pozivi *exit* i *wait*

```
void exit(int status) ;
```

završava izvođenje procesa koji ga je pozvao (ubija ga). Prje završetka, uredno se zatvaraju sve otvorene datoteke. Ne vraća nikakvu vrijednost jer iza njega nema nastavka procesa. Za *status* se obično stavlja 0 ako proces normalno završava, a 1 inače. Roditelj procesa koji završava pozivom *exit* prima njegov *status* preko sistemskog poziva *wait*.

```
int wait(int *statusp) ;
```

Ovaj sistemski poziv čeka da neki od procesa djece završi (ili bude zaustavljen za vrijeme praćenja), s tim da mu se ne može reći koji proces treba čekati. Vraća identifikacijski broj procesa djeteta koji je završio i sprema njegov status (16 bitova) u cijeli broj na koji pokazuje *statusp*, osim ako je taj argument `NULL`. U tom slučaju se status završenog procesa gubi. U slučaju greške (djece nema, ili je čekanje prekinuto primitkom signala) rezultat je 1.

Postoje tri načina kako može završiti proces: pozivom *exit*, primitkom signala ili padom sustava (nestanak napajanja ili slično). Na koji je način proces završio možemo pročitati iz statusa na koji pokazuje *statusp* osim ako se radi o trećem slučaju (vidi `man wait`).

Ako proces roditelj završi prije svog procesa djeteta, djetetu se dodjeljuje novi roditelj - proces *init* s identifikacijskim brojem 1. *init* je važan prilikom pokretanja sustava, a u kasnijem radu većinom izvodi *wait* i tako "prikuplja izgubljenu djecu" kada završe.

Ako proces dijete završi, a roditelj ga ne čeka sa *wait*, on postaje proces-zombi (*zombie*). Otpuštaju se njegovi segmenti u memoriji, ali se zadržavaju njegovi podaci u tablici procesa. Oni su potrebni sve dok roditelj ne izvede *wait* kada proces-zombi nestaje. Ako roditelj završi, a da nije pozvao *wait*, proces-zombi dobiva novog roditelja (*init*) koji će ga prikupiti sa *wait*.

### Pokretanje paralelnih procesa

U ovoj vježbi trebat će pokrenuti više procesa tako da rade paralelno. To se može izvesti sa dvije petlje. U prvoj se kreiraju procesi djeca pozivom *fork* a svako dijete poziva odgovarajuću funkciju. Iza poziva funkcije treba se nalaziti *exit* jer samo roditelj nastavlja izvršavanje petlje. Nakon izlaska iz prve petlje, roditelj poziva *wait* toliko puta koliko je procesa djece kreirao.

```
for (i = 0; i < N; i++)
   if (fork() == 0) {
      funkcija koja obavlja posao djeteta i
      exit(0);
   }

while (i--)
   wait (NULL);
```

# ZAJEDNIČKA MEMORIJA

Nakon kreiranja novog procesa sa *fork*, procesi roditelj i dijete dijele segment s podacima koji se sastoji od stranica. Sve dok je stranica nepromjenjena oba procesa je mogu čitati. Ali čim jedan proces pokuša pisati na stranicu procesi dobiva odvojene kopije podataka. Tada niti globalne varijable nisu zajedničke za sve procese, pa ako jedan proces promjeni neku varijablu drugi to neće primjetiti. To je jedan od razloga za korištenje zajedničke memorije. Varijable koja trebaju biti zajedničke za sve procese moraju se nalaziti u zajedničkoj memoriji koju prethodno treba zauzeti.

Zajednička memorija je najbrži način komunikacije među procesima. Ista memorija je priključena adresnim prostorima dva ili više procesa. Čim je nešto upisano u zajedničku memoriju, istog trenutka je dostupno svim procesima koji imaju priključen taj dio zajedničke memorije na svoj adresni prostor. Za sinkronizaciju čitanja i pisanja u zajedničku memoriju mogu se upotrijebiti semafori, poruke ili posebni algoritmi.

Blok zajedničke memorije se kraće naziva segment. Može biti više zajedničkih segmenata koji su zajednički za različte kombinacije aktivnih procesa. Svaki proces može pristupiti k više segmenata. Segment je prvo kreiran izvan adresnog prostora bilo kojeg procesa, a svaki proces koji želi pristupiti segmentu izvršava sistemski poziv kojim ga veže na svoj adresni prostor. Broj segmenata je određen sklopovskim ograničenjima, a veličina segmenta može također biti ograničena.

## Sistemski pozivi za kreiranje i rad sa zajedničkom emorijom

```
typedef key_t int;
int shmget(key_t key, int size, int flags) ;
```

Ovaj sistemski poziv pretvara ključ (*key*) nekog segmenta zajedničke memorije u njegov identifikacijski broj ili kreira novi segment. Novi segment duljine barem *size* bajtova će biti kreiran ako se kao ključ upotrijebi `IPC_PRIVATE`. U devet najnižih bitova *flags* se stavljaju dozvole pristupa (na primjer, oktalni broj 0600 znači da korisnik može čitati i pisati, a grupa i ostali ne mogu). *shmget* vraća identifikacijski broj segmenta koji je potreban u *shmat* ili -1 u slučaju greške.

Proces veže segment na svoj adresni prostor sa *shmat*:

```
char *shmat(int segid, char *addr, int flags) ;
```

Ako segment treba vezati na određenu adresu treba je staviti u *addr*, a ako je *addr* jednako `NULL` jezgra će sama odabrati adresu (moguće ako se kasnije ne koristi dinamičko zauzimanje memorije sa *malloc* ili slično). *flags* također najčešće može biti 0. *segid* je identifikacijski broj segmenta dobijen pozivom *shmget*. *shmat* vraća pokazivač na zajedničku memoriju duljine tražene u shmget ili 1 ako dođe do greške. Dohvaćanje i spremanje podataka u segmente obavlja se na uobičajen način.

Segment se može otpustiti sistemskim pozivom *shmdt*:

```
int shmdt(char *addr) ;
```

Sama zajednička memorija ostaje nedirnuta i može joj se opet pristupiti tako da se ponovo veže na adresni prostor procesa, iako je moguće da pri tome dobije drugu adresu u njegovom adresnom prostoru. *addr* je adresa segmenta dobivena pozivom *shmat*.

Uništavanje segmenta zajedničke memorije izvodi se sistemskim pozivom *shmctl*:

```
int shmctl(int segid, int cmd, struct shmid_ds *sbuf) ;
```

Za uništavanje segmenta treba za *segid* staviti identifikacijski broj dobiven sa *shmget*, *cmd* treba biti `IPC_RMID`, a *sbuf* može biti `NULL`. Greška je uništiti segment koji nije otpušten iz adresnog prostora svih procesa koji su ga koristili. *shmctl*, kao i *shmdt* vraća 0 ako je sve u redu, a -1 u slučaju greške. (Detaljnije o ovim pozivima u: `man shmget`, `man shmop`, `man shmctl`)

## Struktura programa sa paralelnim procesima i zajedničkom memorijom

```
definiranje kazaljki na zajedničke varijable


proces k
   početak
      proces koji koristi zajedničke varijable
      ...
   kraj.
...
```

```
glavni program
   početak
      zauzimanje zajedničke memorije
      pokretanje paralelnih procesa
      oslobađanje zauzete zajedničke memorije
   kraj.
```

**VAŽNO**: Varijablama u zajedničkoj memoriji se nužno pristupa korištenjem kazaljki.

## Primjer programa sa paralelnim procesima i zajedničkom memorijom

Ovo je trivijalan primjer korištenja zajedničke memorije. Koristi se jedna cjelobrojna zajednička varijabla. Kreiraju se dva paralelna procesa, od kojih jedan upisuje vrijednost (različitu od 0) u tu varijablu, a drugi čeka da ona bude upisana.

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

int * ZajednickaVarijabla;

void Pisac(int i)
{
   * ZajednickaVarijabla = i;
}

void Citac(void)
{
   int i;

   do {
      i = * ZajednickaVarijabla;
   } while (i == 0);

   printf("Procitano je: %d\n", i);
}

int main(void)
{
   int Id;      /* identifikacijski broj segmenta */

   /* zauzimanje zajedničke memorije */
   Id = shmget(IPC_PRIVATE, sizeof(int), 0600);
   if (Id == -1)
      exit(1);  /* greška - nema zajedničke memorije */
   ZajednickaVarijabla = (int *) shmat(Id, NULL, 0);

   /* pokretanje paralelnih procesa */
   if (fork() == 0) {
      Citac();
      exit(0);
   }
   if (fork() == 0) {
      Pisac(123);
      exit(0);
   }
   (void) wait(NULL);
   (void) wait(NULL);

   /* oslobađanje zajedničke memorije */
   (void) shmdt((char *) ZajednickaVarijabla);
   (void) shmctl(Id, IPC_RMID, NULL);

   exit(0);
}
```

# ZADATAK

Ostvariti suradnju paralelnih procesa korištenjem zajedničke memorije. Struktura procesa dana je slijedećim pseudokodom:

```
proces proc(i)          /* i  [0..n-1] */
   za k = 1 do 5 čini
      uđi u kritični odsječak
      za m = 1 do 10 čini
          ispiši (i, k, m)
      izađi iz kritičnog odsječka
kraj.
```

Suradnju ostvariti na slijedeće načine:

1. za nekoliko procesa bez međusobnog isključivanja,
2. za dva procesa međusobnim isključivanjem po Dekkerovom algoritmu,
3. za više procesa međusobnim isključivanjem po Lamportovom algoritmu.

**Dekkerov algoritam:**

```
zajedničke varijable: PRAVO, ZASTAVICA[0..1]

proces proc(i)
   za k = 1 do 5 čini
      ZASTAVICA[i] = 1
      dok je ZASTAVICA[j]<>0 čini
         ako je PRAVO=j onda
            ZASTAVICA[i] = 0
            dok je PRAVO=j čini
               ništa
            ZASTAVICA[i] = 1
      za m = 1 do 10 čini
         ispiši (i, k, m)
      PRAVO = j
      ZASTAVICA[i] = 0
kraj.
```

**Lamportov algoritam:**

```
zajedničke varijable: TRAŽIM[0..n-1], BROJ[0..n-1]

proces proc(i)
   za k = 1 do 5 čini
      TRAŽIM[i] = 1
      BROJ[i] = max(BROJ[0],...,BROJ[n-1]) + 1
      TRAŽIM[i] = 0
      za j = 0 do n-1 čini
         dok je TRAŽIM[j]<>0 čini
            ništa
         dok je BROJ[j]<>0 ^ (BROJ[j],j)<(BROJ[i],i) čini
            ništa
      za m = 1 do 10 čini
            ispiši (i, k, m)
      BROJ[i] = 0
kraj.
```

# UPUTA

Oznaka `(a1,b1)<(a2,b2)` upotrijebljena u Lamportovom algoritmu označava da treba uspoređivati `a1` i `a2`, a ako su oni jednaki odluku donosi usporedba `b1` i `b2`.

Za slučaj bez međusobnog isključivanja nije potrebno koristiti zajedničke varijable. Kod druga dva programa treba zajedničke varijable tako organizirati da se prostor za njih zauzme odjednom i podijeli među njima.

Ovo je nužno zbog ograničenog broja segmenata i velikog broja korisnika.

Ovisno o opterećenju računala i broju procesa koji se pokreću, a da bi se vidjele razlike prilikom izvođenja programa 1, 2. i 3. može biti potrebno usporiti izvršavanje sa:

```
sleep(1);
```

nakon `ispisi (i, k, m)`.

Ako se segment zajedničke memorije ne uništi, zajednička memorija ostaje trajno zauzeta i nakon završetka svih procesa koji je koriste, pa čak i nakon što korisnik koji ju je kreirao napusti računalo (`logout`). Pošto je broj segmenata ograničen, to ubrzo može izazvati nemogućnost rada programa koji koriste zajedničku memoriju. (Isto vrijedi i za ostala sredstva za međuprocesnu komunikaciju: skupove semafora i redove poruka.) Podaci o upotrijebljenim sredstvima za međuprocesnu komunikaciju mogu se dobiti naredbom: ipcs. Naredbom *ipcrm* mogu se uništavati pojedina sredstva (vidi: man ipcrm, man ipcs). Za lakše uništavanje zaostalih segmenata zajedničke memorije (kao i skupova semafora i redova poruka) može poslužiti jednostavan program brisi:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/msg.h>
#include <values.h>


int main ( void )
{
   int i;

   for (i = 1; i < MAXLONG; i++) {
      if (shmctl(i, IPC_RMID, NULL) != -1)
         printf("Obrisao zajednicku memoriju %d\n", i);
      if (semctl(i, 0, IPC_RMID, 0) != -1)
         printf("Obrisao skup semafora %d\n", i);
      if (msgctl(i, IPC_RMID, NULL) != -1)
         printf("Obrisao red poruka %d\n", i);
   }

   return 0;
}
```