Dekkerov postupak međusobnog isključivanja

Uz pomoć Dekkerovog algoritma moguće je sinkronizirati dvije dretve ili dva procesa koji primjerice izgledaju ovako:

```
proces proc(i){          /* i  [0,1] */
   ponavljaj{
      uđi u kritični odsječak
      K.O.;
      izađi iz kritičnog odsječka
      N.K.O.;
   }dok je zadovoljen(uvjet);
}
```

Dekkerov algoritam:

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

funkcija uđi_u_kritični_odsječak(i,j)
{
   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
      }
   }
}

funkcija izađi_iz_kritičnog_odsječka(i,j)
{
   PRAVO = j
   ZASTAVICA[i] = 0
}
```

Dodatne upute:

Ako se program rješava s procesima tada 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 može se usporiti izvršavanje programa sa:

```
sleep(1);
```

na kraju kritičnog odsječka (K.O.).

Lamportov postupak međusobnog isključivanja

Uz pomoć Lamportovog algoritma moguće je sinkronizirati dvije ili više dretvi, odnosno dva ili više procesa koji primjerice mogu izgledati ovako:

```
proces proc(i){          /* i  [0..n] */
   ponavljaj{
      uđi u kritični odsječak;
      K.O.;
      izađi iz kritičnog odsječka;
      N.K.O.;
   }dok je zadovoljen(uvjet);
}
```

Lamportov algoritam:

```
zajedničke varijable: ULAZ[0..n-1], BROJ[0..n-1]
funkcija uđi_u_kritični_odsječak(i)
{
   ULAZ[i] = 1;
   BROJ[i] = max(BROJ[0],...,BROJ[n-1]) + 1;
   ULAZ[i] = 0;

   za j = 0 do n-1 čini
      dok je ULAZ[j] <> 0 čini
         ništa;
      dok je BROJ[j] <> 0 && (BROJ[j] < BROJ[i] || (BROJ[j] == BROJ[i] && j <
i)) čini
         ništa;
}

funkcija izađi_iz_kritičnog_odsječka(i)
{
   BROJ[i] = 0;
}
```

Dodatne upute:

Ako se program rješava s procesima tada 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 može se usporiti izvršavanje programa sa:

```
sleep(1);
```

na kraju kritičnog odsječka (K.O.).

Problemi zbog izvođenja instrukcija "preko reda"

Za ispravan rad Dekkerovog, Petersonovog i Lamportovog algoritma pretpostavlja se da se instrukcije programa izvode zadanim redoslijedom. Međutim, neki procesori, kako bi ubrzali izvođenje programa, dozvoljavaju izvođenje instrukcija i "preko reda" (engl. *out-of-order*). Takvo ponašanje može uzrokovati greške u algoritmu međusobnog isključivanja.

Primjerice, prilikom izvođenja Lampotovog algoritma može se dogoditi da se sljedeći niz instrukcija ne izvodi zadanim redoslijedom.

```
1:   ULAZ[i] = 1;
2:   najveci = max(BROJ[0],...,BROJ[n-1]);
3:   BROJ[i] = najveci + 1;
4:   ULAZ[i] = 0;
```

Procesor koji omogućuje izvođenje instrukcija "preko reda" bi mogao ustanoviti da su instrukcije u trećoj i četvrtoj liniji međusobno nezavisne i mogao bi ih izvesti proizvoljnim redosljedom, npr. liniju 4 prije linije 3 (razlog za to može biti da se element `ULAZ[i]` nalazi u priručnom spremniku, a `BROJ[i]` ne). U tom se slučaju ne radi više o Lamportovom algoritmu.

Kako bi se izbjeglo proizvoljno izvođenje instrukcija trebalo bi se jezičnom procesoru naznačiti da je pristup spornim varijablama točno određen programskim kodom.

Jedan od načina da se to napravi jest da se varijabla proglasi tipom podataka `atomic`:

```
#include <stdatomic.h>
atomic_int ULAZ[N], BROJ[N];
```

Drugi način je da se koriste atomarne operacije. Atomarne operacije, osim što će navedenu operaciju napraviti kao atomarnu (nedjeljivu), još osiguravaju ispravan redoslijed obavljanja operacija (instrukcije prije moraju biti gotovo prije, instrukcije poslije ne smiju započeti prije nego li je ova operacija gotova). Primjer s njihovim korištenjem za postavljanje i čitanje varijabli u Lamportovom algoritmu je u nastavku.

```
//umjesto: ULAZ[i] = 1;
__atomic_store_n (&ULAZ[i], 1, __ATOMIC_RELEASE);

//umjesto: ULAZ[i] = 0;
__atomic_store_n (&ULAZ[i], 0, __ATOMIC_RELEASE);

//umjesto: BROJ[i] = najveci + 1;
__atomic_store_n (&BROJ[i], najveci + 1, __ATOMIC_RELEASE);
```

```
//umjesto: while (ULAZ[j] == 1) ;
do __atomic_load (&ULAZ[j], &ulazJ, __ATOMIC_ACQUIRE);
while (ulazJ == 1);

//umjesto: while (BROJ[j] != 0 && itd) ;
do __atomic_load (&BROJ[j], &brojJ, __ATOMIC_ACQUIRE);
while (brojJ != 0 && itd);

//umjesto: BROJ[i] = 0;
__atomic_store_n (&BROJ[i], 0, __ATOMIC_RELEASE);
```

Više informacija o ovakvim operacijama na Atomic GCC te Atomic GCC: Memory model.

Operacijski sustavi
Druga laboratorijska vježba

1. Motivacija

Cilj ove laboratorijske vježbe je praktično proučavanje i upoznavanje s jezgrenim pozivima vezanim uz stvaranje i upravljanje procesima. U okviru vježbe potrebno je ostvariti jednostavnu ljusku `fsh` *(Fer SHell)*.

2. Zadatak

Zadatak je podijeljen u tri glavna dijela, a popis funkcionalnosti koje je potrebno ostvariti je dan u nastavku:

1. Osnovno pokretanje programa - **2 boda**
2. Ugrađene naredbe `cd` i `exit` te propagacija signala `SIGINT` - **1 bod**
3. Prošireno pokretanje programa pomoću varijabli okoline - **1 bod**

Ljusku je potrebno **ostvariti u programskom jeziku C ili C++** (bilo koji standard) te u bilo kojem **UNIX okruženju** (npr. GNU/Linux, FreeBSD, itd.). Dodatno je dozvoljeno korištenje bilo kojih funkcija koje pruža standardna C biblioteka (npr. `glibc`, `musl`) **osim funkcija** **`system`, `execvp`, `execlp` i `execvpe`**.

2.1. Pristupanje dokumentaciji jezgrenih poziva i bibliotečnih funkcija

Tijekom ove laboratorijske vježbe susrest ćete se s raznim složenim jezgrenim pozivima čije je ponašanje potrebno detaljno proučiti kako bi ljuska ispravno radila. Glavni izvor informacija pri radu sa naredbama ili funkcijama koje pruža operacijski sustav je naredba `man`. Naredba `man` kao argument prima ime programa ili funkcije čijoj se dokumentaciji želi pristupiti. Bitno je napomenuti da je dokumentacija cijelog sustava podijeljena u zasebne sekcije grupirane po vrsti sadržaja kojeg opisuju. Primjerice, treća sekcija opisuje bibliotečne funkcije, druga sekcija opisuje jezgrene pozive, a prva opisuje programe ili ugrađene naredbe ljuske. Pozivanje naredbe `man` u obliku `man <ime_programa>` slijedom pretražuje sekcije i vraća prvu dokumentaciju koja odgovara imenu programa. Nažalost, neke funkcije i naredbe dijele imena, te je moguće da vam naredba `man` vrati pogrešnu dokumentaciju, kao što je prikazano u slijedećem primjeru.

Pristupanje odgovarajućoj dokumentaciji bibliotečne funkcije sleep.

```
$ man sleep
SLEEP(1)   User Commands
SLEEP(1)

NAME
       sleep - delay for a
specified amount of time
SYNOPSIS
       sleep NUMBER[SUFFIX]...
       sleep OPTION
...
```

```
$ man 3 sleep
sleep(3)   Library Functions
Manual   sleep(3)

NAME
       sleep - sleep for a
specified number of seconds

LIBRARY
       Standard C library (libc,
-lc)

SYNOPSIS
       #include <unistd.h>
       unsigned int
sleep(unsigned int seconds);
...
```

Taj problem rješavamo tako da naredbi `man` eksplicitno zadamo sekciju koju treba pretražiti. Tako se dokumentaciji bilo kojeg jezgrenog poziva pristupa naredbom `man 2 <ime_poziva>`. a dokumentaciji funkcija standardne C biblioteke pristupa se naredbom `man 3 <ime_funkcije>`.

2.2. Osnovno pokretanje programa

Pseudokod 1 ugrubo opisuje rad ljuske. Prije čekanja na upisivanje naredbe, vaša ljuska treba ispisati "`fsh>` ". Svaka naredba sastoji se od imena naredbe i više argumenata koji su odvojeni proizvoljnim brojem razmaka. Nije potrebno ostvariti "brisanje" znakova prilikom upisivanja naredbe.

Primjer 1: Pseudokod osnovnog rada ljuske.

```
while(1){
    ispisi "fsh> ";

    cekaj naredbu;
    procitaj znakovni niz i obradi;

    naredba = pronadi_naredbu();
    if(naredba nije ugradena){
       stvori novi proces;
       u djetetu pokreni program s unesenim argumentima;
       cekaj zavrsetak djeteta;
    }
}
```

Primjer 2: Primjer ispravnog rada ljuske.

```
fsh> /usr/bin/pwd
/home/student
fsh> /usr/bin/echo "pozdrav"
pozdrav
fsh> /usr/bin/ls -lh
# ispis sadrzaja direktorija
fsh> /bin/asdf
fsh: Unknown command: /bin/asdf
```

2.2.1. Preporučeni tijek rješavanja

1. Napisati osnovni kostur ljuske,
2. Ostvariti funkciju koja ulazni niz znakova razbija na ime naredbe i argumente (ako postoje),
3. Detaljno proučiti jezgrene pozive `fork`, `wait`, i `execve`,
4. Ostvariti funkciju koja prepoznaje "ugrađene" naredbe (potrebno za drugi podzadatak),
5. Ostvariti pokretanje programa pomoću jezgrenih poziva navedenih u trećoj točki.

2.3. Osnovne ugrađene naredbe i propagacija signala

2.3.1. Naredba `cd`

Ugrađena naredba `cd` treba korisniku omogućiti osnovnu navigaciju po datotečnom sustavu, za što preporučujemo koristiti funkciju `chdir()` (`man 3`

`chdir`). Primjer ispravno ostvarene naredbe dan je u Primjeru 3. Posebnu pozornost potrebno je obratiti na sljedeće slučajeve:

- Za sve greške (npr. nepostojeće direktorije) potrebno je ispisati odgovarajuću poruku na `stderr`

`Primjer 3: Primjer ispisa prilikom korištenja ispravno ostvarene naredbe cd`

```
fsh> /usr/bin/pwd
/home/student
fsh> cd Documents
fsh> /usr/bin/pwd
/home/student/Documents
fsh> cd /nepostojec
cd: The directory '/nepostojec' does not exist
```

2.3.2. Naredba `exit`

Ugrađena naredba `exit` treba korisniku omogućiti izlazak iz ljuske kada ju korisnik upiše ili kada korisnik pošalje prazan niz znakova kao naredbu.

2.3.3. Slanje signala

Vaša ljuska mora korisniku omogućiti da ručno prekine izvođenje pomoću kombinacije tipki `CTRL-C`, odnosno slanjem signala `SIGINT`. Pri tome je bitno da poslani signal **ne prekine rad ljuske, nego programa koji se trenutno izvodi**. Ako korisnik pošalje `SIGINT` dok nijedan program nije pokrenut, potrebno je samo ispisati novi red. Za ostvarenje ove funkcionalnosti preporučujemo porodicu funkcija `sigaction` (`man 3 sigaction`). Vaša ljuska bi trebala "uhvatiti" signal i odlučiti što s njim treba napraviti.

2.3.4. Bitne napomene

Koncept *procesnih grupa*[1] i standard *POSIX* propisuju da svi signali poslani preko tipkovnice moraju biti propagirani svim procesima koji su članovi tzv. *foreground process* grupe. Kako je naša ljuska član te grupe, u našem slučaju to vodi do neispravne propagacije signala `SIGINT` jer se isti propagira svim procesima pokrenutim iz ljuske, a ne samo ljusci. Rješenje za ovaj problem je smjestiti novostvoreni proces dijete u zasebnu grupu **pozivom funkcije `setpgid(0,0)` (`man 3 setpgid`) u djetetu prije poziva `execve()`**. Pritiskom kombinacije tipki `CTRL+C` Vaša ljuska **ne dobiva nikakav niz znakova na standardni ulaz**, već jezgra operacijskog sustava prepoznaje tu kombinaciju i šalje signal `SIGINT`. Stoga nije potrebno provjeravati je li ljuska primila znakove `"^C"` kao ulaz.

2.3.5. Preporučeni tijek rješavanja

1. Dodati `cd` i `exit` u popis ugrađenih naredbi,
2. Dodati funkcionalnosti za prethodno navedene naredbe,
3. Izmijeniti pokretanje djeteta tako da se dijete smjesti u zasebnu procesnu grupu,
4. Ostvariti funkciju za obradu signala `SIGINT`.

2.4. Prošireno pokretanje programa pomoću varijabli okoline

U dosadašnjim primjerima za sve naredbe bilo je potrebno navesti njihovu punu putanju. Kako je taj cijeli postupak nezgrapan i dugotrajan, ljuske obično pružaju mogućnost upisivanja samo imena programa. Nakon što je upisano ime naredbe, ljuska traži program naredbe u prethodno definiranim dijelovima datotečnog sustava koji su dostupni preko varijable okoline `PATH` [2]. Varijabla `PATH` sadrži niz putanja odvojenih znakom "**:**", a primjer sadržaja varijable dan je u Primjeru 4.

`Primjer 4: Primjer sadržaja varijable PATH`

```
> echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/sbin
```

Vaš zadatak je proširiti osnovnu funkcionalnost ljuske tako da omogućite pokretanje programa preko njihovog imena. Vaša ljuska treba dohvatiti sadržaj varijable `PATH` korištenjem bibliotečnog poziva `getenv (man 3 getenv)`, obraditi ga i prilikom svakog pokretanja programa pretražiti sve navedene direktorije kako bi našla odgovarajući program.

`Primjer 5: Korištenje poziva getenv.`

```
obradi_varijablu_path() {
    ...
    char *sadrzaj = getenv("PATH");
    ...
}
```

2.4.1. Bitne napomene

- Za pronalazak programa u direktoriju preporučujemo koristiti poziv `access` (i jezgreni poziv i bibliotečna funkcija su dozvoljene)
- Postupak pretrage se ne smije pokretati ako korisnik upiše putanju (tj. ako ime naredbe počinje s '/' ili '.')
- Potrebno je ispisati poruku greške ako program ne postoji

2.4.2. Preporučeni tijek rješavanja

1. Proučiti poziv `getenv()`,
2. Ostvariti dohvaćanje i obradu sadržaja varijable okoline `PATH` pri pokretanju ljuske,
3. Dodati traženje programa naredbe prije pokretanja.