



Programska podrška mjernih i procesnih sustava

vježba br. 5: **Threads**

1. UVOD

Thread (hrv. dretva ili nit, u dalnjem tekstu dretva) je uz procese još jedan mehanizam koji omogućava paralelno izvođenje programa. Dretve se izvode konkurentno (postoji scheduler) unutar jednog procesa. Svaki proces ima barem jednu dretvu izvođenja tako da možemo reći da dretve predstavljaju manju jedinicu izvođenja od procesa (zovu se i *lightweight process*). Za OS koji omogućava korištenje dretvi se kaže da podržava *multithreading* (većina današnjih ih podržava), slično kao *multitasking* za procese.

Kada se proces kreira u njemu se stvori jedna dretva koja onda sekvencijalno izvodi program. Ta dretva može kreirati nove dretve koje mogu izvršavati različite dijelove istog programa paralelno (kvaziparalelno). Tipičan primjer je spell checker u MS Word-u koji se izvršava paralelno (za vrijeme pisanja) unutar istog procesa (aplikacije).

Kada jedan proces pomoću *fork*-a stvori novi proces taj novi proces dobije svoj memorijski prostor sa kopijama varijabli, kopiraju se *file descriptori* i sl. Ako dijete zatvori *file descriptor* ili promijeni vrijednost varijable to nema nikakvog utjecaja na njegovog roditelja. Kada se pak kreira nova dretva ona se izvršava u istom memorijskom prostoru te se sve dijeli i ništa se ne kopira. Ona jednako pristupa svim varijablama i *file descriptorima* kao i bilo koja druga dretva unutar procesa. Ako jedna dretva promijeni iznos globalne varijable sve dretve će vidjeti promjenu tako da nema potrebe za klasičnim IPC mehanizmima za razmjenu podataka među dretvama.

Sve dretve unutar procesa izvršavaju isti program. Ako jedna dretva pozove *exec* sistemsku funkciju sve dretve se prekidaju i počinje izvođenje novog programa (koji naravno može ali i ne mora biti višedretven).

2. PTHREADS (POSIX Thread API)

Sve funkcije i strukture podataka potrebne za rad sa dretvama su deklarirane unutar `<pthread.h>` header datoteke. Također one nisu dio standardne C biblioteke već `libpthread` biblioteke te ih je potrebno prilikom linkanja uključiti sa `-lpthread` opcijom *compilera*.

Svaka dretva unutar procesa ima svoj *thread ID* te svaka dretva izvršava jednu funkciju. Drugim riječima kôd dretve se definira sa funkcijom u programu i dretva završava kad ta

funkcija završi sa izvođenjem. Moguće je i da više dretvi izvodi istu funkciju (ali obično s drugim parametrima).

Dretva se stvara pomoću:

```
void* pthread_create( pthread_t* thread_id, NULL, void* (*f) (void *), void *arg );
```

Prvi argument je pokazivač na ID novo kreirane dretve, drugim argumentom se mogu postaviti određeni atributi dretve (ovdje stavljeno na NULL), treći je adresa funkcije koju ta dretva izvodi (tipa `void *f(void *)`) te na kraju argument koji se toj funkciji šalje. Argument je samo jedan ali obično on pokazuje na strukturu koja sadrži više varijabli.

Nakon poziva `pthread_create` funkcije, glavna dretva nastavlja sa izvođenjem paralelno sa novonastalom dretvom. Scheduling tih dretvi je asinkron.

Završetak dretve nastupi ili kada funkcija koju ona izvršava dođe do `return` naredbe ili se eksplicitno prekine sa naredbom:

```
pthread_exit(void *ret_val);
```

Ekvivalent `wait()` poziva kod procesa, ovdje je:

```
pthread_join(pthread_t, void *ret_val);
```

prvi argument označava koju dretvu čekamo a drugi je njena povratna vrijednost. To je i jedini način da saznamo što je ta dretva/funkcija vratila.

Primjer programa u kojem se koristi nova dretva (prenose se i argumenti):

thread_print.c

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

typedef struct
{
    char znak; // znak koji se ispisuje
} print_parms;

/* funkcija koja se izvršava u posebnoj dretvi */
void* char_print (void* parametri)
{
    /* Treba castat sa void na pravilnu strukturu podataka */
    print_parms* p = (print_parms*) parametri;

    while(1)
        fputc (p->znak, stdout);

    return NULL;
}

int main ()
{
    pthread_t thread_id1;
    pthread_t thread_id2;

    print_parms thread_args1;
    print_parms thread_args2;
```

```

/* kreiraj dretvu koja ce printati 'x' */
thread_args1.znak = 'x';
pthread_create (&thread_id1, NULL, char_print, &thread_args1);

/* kreiraj dretvu koja ce printati 'o' */
thread_args2.znak = 'o';
pthread_create (&thread_id2, NULL, char_print, &thread_args2);

/* moram cekati da dretve zavrse iz dva razloga:
1. jer cu inace dealocirati strukturu kojoj dretva pristupa
   (zbog 'call by reference').
2. Kada glavna dretva zavrsti i proces zavrsi skupa sa svim dretvama
*/
pthread_join(thread_id1,NULL); // povratne vrijednosti nema, zato NULL
pthread_join(thread_id2,NULL); // povratne vrijednosti nema, zato NULL

return 0;
}

```

- Proučite i pokrenite program, uočite kako scheduler mijenja redoslijed izvođenja dretvi.

Tipičan primjer matematičke operacije koja se može paralelizirati je množenje matrica. Naime operacije množenja pojedinih redaka sa stupcima su potpuno nezavisne. Tako će brzina izvođenja linearno rasti sa količinom procesora koji se koriste za takvo računanje. Kostur kôda za paralelno računanje množenja dviju matrica 2x2 je dan ispod:

thread_matmult.c

```

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

typedef struct
{
    int redak[2];
    int stupac[2];
} thread_parms;

// funkcija koja se izvršava u posebnoj dretvi
void* mult(void* parametri)
{
    int rezultat;

    // Treba castat sa void na pravilnu strukturu podataka
    thread_parms* p = (thread_parms*) parametri;

    /* mnozi readak sa stupcem */

    return (void *)rezultat;
}

/* The main program. */
int main ()
{
    pthread_t thread_id1;
    thread_parms thread_args1;
    int thread_rez1; // mjesto za jedan medjurezultat

    // Definicija matrica koje treba pomnoziti
    int a[2][2]={{1,2},
                 {2,3}};

```

```

int b[2][2]={{5,5},
             {6,6}};

/* kreiraj argument za funkciju */

// kreiraj jednu dretvu koja racuna medjurezultat
pthread_create (&thread_id1, NULL, mult, &thread_args1);

//pokupi medjurezultat
pthread_join(thread_id1,(void *) &thread_rez1);

printf("rezultat je %d\n",thread_rez1);

return 0;
}

```

- Nadopuni postojeći program tako da paralelno izračuna umnožak te dvije matrice. Sve dretve moraju izvršavati istu funkciju (naravno s različitim argumentima).

3. DRETVE I KRITIČNI ODSJEĆCI

Dretve se izvršavaju konkurentno, znači da se ne zna unaprijed kada će scheduler odabrati pojedinu dretvu. S obzirom da je i puno resursa zajedničko puno češće dolazi do *race conditiona* kao što je bilo simulirano na prošloj vježbi sa dijeljenom memorijom.

Osim semafora za sinkronizirani pristup dijeljenim podatcima koristi se *mutex* (MUTual EXclusion locks). Uloga mu je slična kao binarni semafor i jednostavniji je za korištenje. Taj resurs ima ulogu ključa. Tko prvi zaključa ušao je u kritični odsječak i na izlasku otključa da drugi mogu uči. Svi koji pokušaju uči dok je zaključano će ostati blokirani i nasumce odabrani iz reda čekanja će uči nakon što se otključa.

Mutex se deklarira kao varijabla tipa `pthread_mutex_t` (Definirano u `<pthread.h>`).

Prije upotrebe potrebno ga je inicijalizirati sa:

```
pthread_mutex_init(pthread_mutex_t *mutex, NULL);
```

Bitna su samo još dva funkcija poziva (za zaključavanje i otključavanje):

```
pthread_mutex_lock(pthread_mutex_t *mutex);
pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Primjer programa (s jedinom dretvom) gdje se oni koriste:

thread_sync.c

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

int global; // deklaracija globalne varijable
pthread_mutex_t mymutex; // deklaracija mutexa

int main()
{
    pthread_mutex_init(&mymutex, NULL); // inicijalizacija mutexa
    global = 5; // inicijalizacija globalne varijable
```

```

pthread_mutex_lock(&mymutex);
global = 6;           /* KRITICNI ODSJECAK */
pthread_mutex_unlock(&mymutex);

printf('Globalna varijabla iznosi %d\n',global);

return 0;
}

```

- Kreirati dvije nove dretve gdje se simulira *race condition* kod pristupa globalnoj varijabli kao u prethodnoj vježbi (tamo je umjesto globalne varijable bila dijeljena memorija) te ga riješiti upotrebom mutexa. Prekid simulirajte `sleep()` sistemskim pozivom.

4. DEADLOCK

Do potpunog zastoja može doći kada se treba zauzeti skup resursa. Ako se resursi ne zauzimaju na jednak način i ako se pojavi prekid dok nisu svi resursi zauzeti može lako doći do potpunog zastoja.

- Zadatak je stvoriti potpuni zastoj dvije dretve prilikom zauzimanja (lockanja) dva *mutexa* (zauzimaju međusobno obrnutim redoslijedom). Prekid simulirajte `sleep()` sistemskim pozivom.