Laboratorul 7

Sincronizare

1 Excluziune mutuală

Adesea în programele cu mai multe fire de execuție și procese avem nevoie ca o singură entitate să execute un număr de instrucțiuni la un moment de timp dat. Această zonă în care are voie un singur proces sau thread se numește zonă critică (critical section).

La laborator vom folosi exclusiv pentru paralelism fire de execuție, dar conceptele folosite se pot aplica la fel de bine pentru situația în care avem de-a face cu mai mult procese.

Obiectul cel mai des folosit pentru asigurarea accesului exclusiv într-o zonă critică este mutex-ul (prescurtat de la **mut**ual **ex**clusive). În bibliotecile de sistem ce implementează standardul POSIX, un obiect de tip mutex este creat cu ajutorul funcției `pthread_mutex_init(3)`

```
int pthread_mutex_init(pthread_mutex_t *mutex,
 const pthread_mutexattr_t *attr);
```

care inițializează un `mutex` cu atributele `attr` (la laborator se va folosi `NULL` pentru a obține atributele implicite similar cum am procedat pentru `pthread`). De obicei variabila mutex se pune fie în memoria globală, fie într-o structură accesibilă tuturor firelor de execuție după care se inițializează în `main()` sau undeva înainte de a fii folosit.

```
pthread_mutex_t mtx;
int main()
{
        /* ... */
        if (pthread_mutex_init(&mtx, NULL)) {
                perror(NULL);
                return errno;
        }
        /* ... */
}
```

Un mutex se poate afla în două stări: fie închis (locked), fie deschis (unlocked). Când mutexul este închis înseamnă că un thread deține dreptul exclusiv de execuție asupra zonei critice până ce decide să renunțe la acest drept.

Pentru a obține un mutex (i.e. pentru a-l închide) se folosește funcția `pthread_mutex_lock(3)` iar pentru a-l elibera se folosește `pthread_mutex_unlock(3)`. Ele sunt folosite tipic în tandem, demarcând zona critică

```
pthread_mutex_lock(&mtx);
count++;        /* critical section */
pthread_mutex_unlock(&mtx);
```

Atenție, funcția `pthread_mutex_lock(3)` nu se termină de executat până nu obține mutexul, blocând astfel firul de execuție ce a apelat-o!

La sfârșit, când nu mai avem nevoie de obiectul de tip mutex, eliberăm resursele ocupate cu `pthread_mutex_destroy(3)`

```
pthread_mutex_destroy(&mtx);
```

2 Semafoare

Semafoarele sunt similare obiectelor de tip mutex, dar pot face față unor scenarii mai sofisticate de sincronizare.

În esență, un semafor este o variabilă S inițializată cu o valoare întreagă care este manipulată exclusiv cu ajutorul a două funcții: `wait` și `post` (sau `signal`). În principiu, ele scad, respectiv cresc, cu o unitate valoarea lui S.

Condiția principală ca obiectele de tip semafor să funcționeze corect este ca funcțiile de mai sus să se execute fără a fii întrerupte (să se execute atomic). Astfel putem observa că un obiect de tip mutex este defapt un caz particular de semafor în care $S = 1$.

În mediile POSIX semafoarele sunt inițializate cu `sem_init(3)`

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

care setează valoarea inițială a semaforului `sem` cu $S =$ `value`. Variabila `pshared` este folosită pentru a semnala dacă vrem să folosim semaforul în cadrul mai multor procese. Nefiind cazul în cadrul laboratorului, vom folosi tot timpul `pshared` $= 0$. Apelul tipic este similar cu cel în cazul obiectelor mutex

```
sem_t sem;
int main()
{
        /* ... */
        if (sem_init(&sem, 0, S)) {
                perror(NULL);
                return errno;
        }
        /* ... */
}
```

unde S este valoarea inițială aleasă.

Funcția `sem_wait(3)` scade valoarea lui S cu o unitate și poate fii executată cu siguranță în medii de lucru paralele.

```
if (sem_wait(&sem)) {
        perror(NULL);
        return errno;
}
```

Atenție, dacă $S = 0$ atunci funcția așteaptă ca valoarea să crească înainte de a o scădea blocând astfel firul apelant!

Similar, pentru a crește valoarea lui S este folosită funcția `sem_post(3)`

```
if (sem_post(&sem)) {
        perror(NULL);
        return errno;
}
```

care, după incrementare $S = S + 1$, verifică dacă sunt thread-uri blocate de semafor și eliberează thread-ul care așteaptă de cel mai mult timp în coadă. Acesta va relua execuția din punctul în care a apelat `sem_wait(3)`.

Când nu mai este folosit, semaforul este eliberat cu ajutorul funcției `sem_destroy(3)`

```
sem_destroy(&sem);
```

3 Sarcini de laborator

1. Scrieți un program care gestionează accesul la un număr finit de resurse. Mai multe fire de execuție pot cere concomitent o parte din resurse pe care le vor da înapoi o dată ce nu le mai sunt utile. Fie numărul maxim de resurse dat

   ```
   #define MAX_RESOURCES 5
   int available_resources = MAX_RESOURCES;
   ```

   Când un thread dorește să oțină un număr de resurse acesta apelează `decrease_count`.

   ```
   int decrease_count(int count)
   {
           if (available_resources < count)
                   return -1;
           else
                   available_resources -= count;
           return 0;
   }
   ```

   iar când resursele nu-i mai sunt necesare apelează `increase_count`

```
int increase_count(int count)
{
        available_resources += count;
        return 0;
}
```

Funcțiile de mai sus prezintă mai multe defecte într-un mediu de execuție paralel printre care și un **race condition**. Modificați funcțiile și rezolvați race condition-ul folosind obiecte de tip mutex. Arătați că modificările dumneavoastră sunt corecte cu ajutorul unui program care pornește mai multe thread-uri ce consumă un număr diferit de resurse fiecare. De exemplu:

```
$ ./count
MAX_RESOURCES=5
Got 2 resources 3 remaining
Released 2 resources 5 remaining
Got 2 resources 3 remaining
Released 2 resources 5 remaining
Got 1 resources 4 remaining
Released 1 resources 5 remaining
Got 3 resources 2 remaining
Got 2 resources 0 remaining
Released 3 resources 3 remaining
Released 2 resources 5 remaining
```

2. Scrieți un program care să sincronizeze execuția a N fire de execuție construind un obiect de tip barieră. Bariera va fii inițializată folosind `init(N)` și fiecare thread va apela `barrier_point()` când va ajunge în dreptul barierei. Când funcția este apelată a N-a oară, aceasta pornește execuția tuturor firelor în așteptare.
   Verificați rezultatele dumneavoastră cu ajutorul unui program care pornește mai multe thread-uri ce se folosesc de barieră pentru a-și sincroniza execuția.

   Funcția executată de fiecare fir poate avea următoarea formă

```
void * tfun(void *v)
{
        int *tid = (int *)v;

        printf("%d reached the barrier\n", *tid);
        barrier_point();
        printf("%d passed the barrier\n", *tid);

        free(tid);
        return NULL;
}
```

unde `tid` este numărul threadului pornit. Astfel, o instanță cu 5 fire de execuție ar afișa

```
$ ./barrier
NTHRS=5
0 reached the barrier
1 reached the barrier
2 reached the barrier
3 reached the barrier
4 reached the barrier
4 passed the barrier
0 passed the barrier
1 passed the barrier
3 passed the barrier
2 passed the barrier
```

Indiciu, pentru a implementa obiectul de tip barieră folosiți un mutex pentru contorizarea firelor ajunse la barieră și un semafor pentru a aștepta la barieră.