Laborator 11 - Elemente de sincronizare

1 Variabile condiție

În programele în care folosim paralelismul, cu fire de execuție concurente, o metodă de semnalizare/sincronizare e reprezentată de variabilele condiție (`pthread_cond_t`).

O variabilă condiție este utilizată împreună cu un mutex pentru:

- a testa o condiție;
- a pune firul de execuție în așteptare dacă condiția este falsă;
- a semnala firelor de execuție când condiția devine adevărată.

Dacă în cazul variabilelor condiție POSIX condiția logică asociată lor este implicită, în cazul implementării `std::cond_variable` aici condiția poate fi explicită.

Mai clar, variabilele condiție pun la dispoziție două categorii de funcții:

- `pthread_cond_wait`, care pune firul de execuție apelant în starea de *wait* până când se îndeplinește motivul pentru care facem *wait* – fie o condiție logică, fie altceva important într-un program, cum ar fi ajungerea și a ultimului fir de execuție la o barieră (care, dacă stați să vă gândiți, este tot o condiție logică `numar_fire_executie_ajunse_la_bariera == total_fire_executie`).
- `pthread_cond_signal`/`pthread_cond_broadcast` – funcții care trezesc din starea *wait* un fir de execuție sau toate firele de execuție care așteaptă pe variabila condiție pe care facem *signal* sau *broadcast*. Important de remarcat că, dacă în cazul semafoarelor un `sem_post` incrementează contorul intern și primul fir de execuție care vine și face `sem_wait` va trece decrementând același contor, `pthread_cond_signal`/`pthread_cond_broadcast` se pierde dacă nu se află nimeni care deja așteaptă pe variabila condiție.

1.1 Inițializare

```
pthread_cond_t cond;
pthread_cond_init(&cond, NULL);
```

1.2 Operații

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mtx);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

1.3 Comentariu despre `pthread_cond_wait`

`pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mtx)`:

Funcția este atipică, în sensul că, pentru a ne asigura că putem să ne punem în starea de *wait* doar dacă o condiție nu este îndeplinită, pașii pentru a apela funcția sunt:

- se face *lock* pe `mtx`;
- se verifică condiția care, dacă nu e îndeplinită, trebuie să ne ducă în starea de *wait*;
- se apelează funcția care, atomic, eliberează mutexul și pune firul de execuție în starea de *wait* pe variabila condiție;
- când firul de execuție este semnalizat, reia execuția doar după ce a reușit să facă din nou *lock* pe mutex-ul `mtx`.

În acest fel avem siguranța că, odată verificată condiția, ne putem pune în starea *wait* cu certitudinea că nimic nu a survenit asupra condiției protejate de mutexul `mtx` și că, odată treziți, putem găsi condiția în starea în care era când am fost semnalizați, pentru că mai întâi se asigură accesul exclusiv prin luarea mutexului `mtx` odată ce am fost treziți.

1.4 Exemplu minimal

```
asteptarea pe o variabila conditie:
pthread_mutex_lock(&mtx);
while (!ready)
    pthread_cond_wait(&cond, &mtx);
/* ready == 1 */
pthread_mutex_unlock(&mtx);

semnalizarea unui fir de executie pe o variabila conditie:
pthread_mutex_lock(&mtx);
ready = 1;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&mtx);

semnalizarea tuturor firelor de executie care asteapta pe o variabila conditie:
pthread_mutex_lock(&mtx);
ready = 1;
pthread_cond_broadcast(&cond);
pthread_mutex_unlock(&mtx);
```

2 Comparație cu semafoarele

Variabilele de condiție diferă de semafoare prin faptul că:

- nu au valoare internă; un semnal trimis unei variabile condiție pe care nu așteaptă nimeni în acel moment se pierde;
- nu mențin contor;
- depind de un mutex;

- sunt folosite exclusiv pentru așteptarea unei condiții logice sau producerii unui eveniment.

Tabel sumar:

Caracteristică	**Semafoare**	**Variabile condiție**
Stare	contor	nu
Blocare	S=0	condiție falsă
Semnalizare	persistentă	se pierde dacă nimeni nu așteaptă

3 Read-Write locks

Un *read-write lock* (RW-lock) permite acces concurent pentru mai mulți cititori sau acces exclusiv pentru un singur scriitor.

Dacă avem mai multe fire de execuție care doar vor să citească resurse partajate, nu avem nici un *race condition* atâta vreme cât nu există un fir de execuție care să dorească să scrie, astfel că putem oferi acces în paralel *reader*-ilor, dar trebuie să asigurăm acces exclusiv în cazul în care dorim să rescriem o resursă partajată.

3.1 Inițializare

```
pthread_rwlock_t rw;
pthread_rwlock_init(&rw, NULL);
```

3.2 Operații

Citire:

```
pthread_rwlock_rdlock(&rw);
/* ... */
pthread_rwlock_unlock(&rw);
```

Scriere:

```
pthread_rwlock_wrlock(&rw);
/* ... */
pthread_rwlock_unlock(&rw);
```

RW-lock-ul garantează:

- cel mult un scriitor simultan;
- oricâți cititori simultan, dacă nu există scriitor.

4 Exemplu minimal

```
#define N 16
int a[N];
pthread_rwlock_t rw;

void *reader(void *v)
{
    pthread_rwlock_rdlock(&rw);
    int x = a[0];
    pthread_rwlock_unlock(&rw);
    return NULL;
}

void *writer(void *v)
{
    pthread_rwlock_wrlock(&rw);
    a[0]++;
    pthread_rwlock_unlock(&rw);
    return NULL;
}
```

RW-lock-ul este util în programe în care operațiile de citire sunt mult mai frecvente decât cele de scriere.

5 Sarcini de laborator

1. Reimplementarea barierei din laboratorul 10 folosind:
    - o variabilă condiție.
2. Implementați un program în care:
    - există o variabilă globală `a`;
    - porniți M thread-uri cititoare și K thread-uri scriitoare cu un id de la 1 la K, în mod aleator, la o distanță de milisecunde între ele;
    - protejați variabila globală cu un `pthread_rwlock_t`;
    - cititorii citesc variabila globală și o afișează;
    - scriitorii modifică variabila globală cu id-ul lor.
3. Modificați programul anterior astfel încât scriitorii să aibă prioritate față de cititori, evitând blocarea lor pe termen lung (*starvation*).