# Sincronizare thread-uri

## Contents

# Fire de execuie (II) - Mecanisme de sincronizare

Pentru sincronizarea firelor de execuie avem la dispoziie:

- seciuni critice (excludere mutuală în cadrul aceluiai proces) doar Win32
- mutex: POSIX, Win32
- semafoare: POSIX, Win32
- variabile de condiie: POSIX, Win32 (începând cu Vista)
- evenimente: doar Win32
- timere: doar Win32.

Standardul POSIX specifică funcii de sincronizare pentru fiecare tip de obiect de sincronizare. API-ul Win32, fiind controlat de o singură entitate, permite ca toate obiectele de sincronizare să poată fi utilizate cu funciile standard de sincronizare: *WaitForSingleObject*, *WaitForMultipleObjects* sau *SignalObjectAndWait*.

# POSIX

## Mutex

Mutexurile sunt obiecte de sincronizare utilizate pentru a asigura accesul exclusiv la o seciune de cod în care se accesează date partajate între două sau mai multe fire de execuie. Un mutex are două stări posibile: ocupat i liber. Un mutex poate fi ocupat de un singur fir de execuie la un moment dat. Atunci când un mutex este ocupat de un fir de execuie, el nu mai poate fi ocupat de niciun altul. În acest caz, o cerere de ocupare venită din partea unui alt fir, în general va bloca firul până în momentul în care mutexul devine liber.

### Iniializarea/distrugerea unui mutex

Un mutex poate fi iniializat/distrus în mai multe moduri:

- folosind o macrodefiniie

```
// initializare statica a unui mutex cu atribute implicite
// NB: mutexul nu este eliberat, durata de viata a mutexului
//     este durata de viata a programului.
pthread_mutex_t mutex_static = PTHREAD_MUTEX_INITIALIZER;
```

- iniializat cu atribute implicite

```
// semnaturile functiilor de initializare si distrugere de mutex:
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);

void initializare_mutex_cu_atribute_implicite()
{
```

```
  pthread_mutex_t mutex_implicit;
  pthread_mutex_init(&mutex_implicit, NULL); // atrr=NULL -> atribute implicite

  // ... folosirea mutexului ...
  // ???

  // eliberare mutex
  pthread_mutex_destroy(&mutex_implicit);
}
```

- iniializare cu atribute explicite

```
// NB: functia pthread_mutexattr_settype si macro-ul PTHREAD_MUTEX_RECURSIVE
// sunt disponibile doar daca se defineste _XOPEN_SOURCE la o valoare >= 500 *INAINTE*
// de a include <pthread.h>. Pentru mai multe detalii consultai feature_test_macros(7).

#define _XOPEN_SOURCE 500
#include <pthread.h>

void initializare_mutex_recursiv()
{
  // definim atribute, le initializam si marcam tipul ca fiind recursiv.
  pthread_mutexattr_t attr;
  pthread_mutexattr_init(&attr);
  pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);

  // definim un mutex recursiv, il initializam cu atributele definite anterior
  pthread_mutex_t mutex_recursiv;
  pthread_mutex_init(&mutex_recursiv, &attr);

  // eliberam resursele atributului dupa crearea mutexului
  pthread_mutexattr_destroy(&attr);

  // ... folosirea mutexului ...
  // ???

  // eliberare mutex
  pthread_mutex_destroy(&mutex_recursiv);
}
```

**NB**: Mutexul trebuie să fie liber pentru a putea fi distrus. În caz contrar funcia va întoarce codul de eroare *EBUSY*. Întoarcerea valorii 0 semnifică succesul apelului.

### Tipuri de mutexuri

Folosind atributele de iniializare se pot crea mutexuri cu proprietăi speciale:

- activarea motenirii de prioritate (*priority inharitance*) pentru a preveni inversiunea de prioritate (*priority invesion*). Există trei protocoale de motenire a priorităii:
  - `PTHREAD_PRIO_NONE` nu se motenete prioritatea când deinem mutexul creat cu acest atribut
  - `PTHREAD_PRIO_INHERIT` dacă deinem un mutex creat cu acest atribut i dacă există fire de execuie blocate pe acel mutex se motenete prioritatea firului de execuie cu cea mai mare prioritate
  - `PTHREAD_PRIO_PROTECT` dacă firul de execuie curent deine unul sau mai multe mutexuri, acesta va executa la maximul priorităilor specificată pentru toi mutecii deinui.

```
#define _XOPEN_SOURCE 500
#include <pthread.h>
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t * attr, int * protocol);
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol);
```

- modul de comportare la preluări recursive ale mutexului
  - ♦ `PTHREAD_MUTEX_NORMAL` nu se fac verificări, preluarea recursivă duce la deadlock
  - ♦ `PTHREAD_MUTEX_ERRORCHECK` se fac verificări, preluarea recursivă duce la întoarcerea unei erori
  - ♦ `PTHREAD_MUTEX_RECURSIVE` mutexurile pot fi preluate recursiv, i trebuie eliberate de acelai număr de ori.

```
#define _XOPEN_SOURCE 500
#include <pthread.h>
pthread_mutexattr_gettype(const pthread_mutexattr_t * attr, int * protocol);
pthread_mutexattr_settype(pthread_mutexattr_t * attr, int protocol);
```

### Ocuparea/eliberearea unui mutex

Funciile de ocupare blocantă/eliberare a unui mutex:

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

Dacă mutexul este liber în momentul apelului, acesta va fi ocupat de firul apelant i funcia va întoarce imediat. Dacă mutexul este ocupat de un alt fir, apelul va bloca până la eliberarea mutexului. Dacă mutexul *este deja ocupat* de firul de execuie curent (lock recursiv), comportamentul funciei este dictat de tipul mutexului:

| Tip mutex | Lock recursiv | Unlock |
|---|---|---|
| `PTHREAD_MUTEX_NORMAL` | deadlock | eliberează mutexul |
| `PTHREAD_MUTEX_ERRORCHECK` | returnează eroare | eliberează mutexul |
| `PTHREAD_MUTEX_RECURSIVE` | incrementează contorul de ocupări | decrementează contorul de ocupări (la zero eliberează mutexul) |
| `PTHREAD_MUTEX_DEFAULT` | deadlock | eliberează mutexul |

Nu este garantată o ordine FIFO de ocupare a unui mutex. Oricare din firele aflate în ateptare la deblocarea unui mutex pot să-l acapareze.

### Încercarea neblocantă de ocupare a unui mutex

Pentru a încerca ocuparea unui mutex fără a atepta eliberarea acestuia în cazul în care este deja ocupat, se va apela funcia:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

```
int rc = pthread_mutex_trylock(&mutex);
if (rc == 0) {
   // mutexul era liber i l-am ocupat cu succes.
} else if (rc == EBUSY) {
   // mutexul este deinut de altcineva i nu l-am putut ocupa
```

```
   // dar în loc să mă blochez ca la un apel pthread_mutex_lock(&mutex)
   // am întors eroarea EBUSY.
} else {
   // a avut loc o altă eroare
}
```

### Exemplu de utilizare a mutexurilor

Un exemplu de utilizare a unui mutex pentru a serializa accesul la variabila globală `global_counter`:

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

#define NUM_THREADS 5

// mutex global
pthread_mutex_t mutex;
int global_counter = 0;

void *thread_routine(void *arg)
{
  // preluam mutexul global
  pthread_mutex_lock(&mutex);

  // afisam si modificam valoarea variabilei globale 'global_counter'
  printf("Thread %d says global_counter=%d\n", (int) arg, global_counter);
  global_counter ++;

  // eliberam mutexul pentru a acorda acces altui fir de executie
  pthread_mutex_unlock(&mutex);
  return NULL;
}

int main(void)
{
  int i;
  pthread_t tids[NUM_THREADS];

  // mutexul este initializat o singura data si folosit de toate firele de executie
  pthread_mutex_init(&mutex, NULL);

  // firele de executie vor executa codul functiei 'thread_routine'
  // in locul unui pointer la date utile, se trimite in ultimul argument
  // un intreg - identificatorul firului de executie
  for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tids[i], NULL, thread_routine, (void *) i);

  // asteptam ca toate firele de executie sa se termine
  for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tids[i], NULL);

  // eliberam resursele mutexului
  pthread_mutex_destroy(&mutex);
  return 0;
}

$ gcc -Wall mutex.c -lpthread
$ ./a.out
Thread 1 says global_counter=0
Thread 2 says global_counter=1
```

```
Thread 3 says global_counter=2
Thread 4 says global_counter=3
Thread 0 says global_counter=4
```

### Futexuri

Mutexurile din firele de execuie POSIX sunt implementate cu ajutorul *futexurilor*, din considerente de performană. Numele de *futex* vine de la Fast User-space muTEX. Ideea de la care a plecat implementarea futexurilor a fost aceea de a optimiza operaia de ocupare a unui mutex în cazul în care acesta nu este deja ocupat. Dacă mutexul nu este ocupat, el va fi ocupat fără ca procesul care îl ocupă să se blocheze. În acest caz, nefiind necesară blocarea, nu este necesar ca procesul să intre în kernel-mode (pentru a intra într-o stare de ateptare). Optimizarea constă în testarea i setarea atomică a valorii mutexului (printr-o instruciune de tip test-and-set-lock) în user-space, eliminându-se trap-ul în kernel în cazul în care nu este necesară blocarea.

Futexul poate fi orice variabilă dintr-o zonă de memorie partajată între mai multe fire de execuie sau procese. Aadar, operaiile efective cu futexurile se fac prin intermediul funciei *do_futex*, disponibilă prin includerea headerului *linux/futex.h*. Signatura ei arată astfel:

```
long do_futex(unsigned long uaddr, int op,
              int val, unsigned long timeout, unsigned long uaddr2, int val2);
```

În cazul în care este necesară blocarea, *do_futex* va face un apel de sistem - *sys_futex*. Futexurile pot fi utile (i poate fi necesară utilizarea lor explicită) în cazul sincronizării proceselor, alocate în variabile din zone de memorie partajată între procesele respective.

## Semafor

Semafoarele sunt obiecte de sincronizare ce reprezintă o generalizare a mutexurilor prin aceea că salvează numărul de operaii de eliberare (incrementare) efectuate asupra lor. Practic, un semafor reprezintă un întreg care se incrementează/decrementează atomic. Valoarea unui semafor nu poate scădea sub 0. Dacă semaforul are valoarea 0, operaia de decrementare se va bloca până când valoarea semaforului devine strict pozitivă. Mutexurile pot fi privite, aadar, ca nite semafoare binare.

Operaiile care pot fi efectuate asupra semafoarelor POSIX sunt:

```
/* SEMAFOARE CU NUME */

// deschiderea unui semafor identificat prin nume.
// folosit pentru a sincroniza procese diferite
sem_t* sem_open(const char *name, int oflag);
sem_t* sem_open(const char, *name, int oflag, mode_t mode, unsigned int value);

// închiderea unui semafor cu nume
int sem_close(sem_t *sem);

// stergerea din sistem a unui semafor cu nume
int sem_unlink(const char *name);

/* SEMAFOARE FARA NUME */
/**
 * initializarea unui semafor fara nume
 * sem - semaforul nou creat
 * pshared - 0 daca semaforul nu este partajat decat
```

```
              de firele de executie ale procesului curent
            - non zero: semafor partajat cu alte procese
              in acest caz 'sem' trebuie alocat intr-o zona
              de memorie partajata
 * value - valoarea initiala a semaforului
 */
int sem_init(sem_t *sem, int pshared, unsigned int value);

// distrugerea unui semafor fara nume
int sem_destroy(sem_t *sem);


/* OPERATII PE SEMAFOARE */

// incrementarea (V)
int sem_post(sem_t *sem);

// decrementarea blocantă (P)
int sem_wait(sem_t *sem);

// decrementarea neblocantă
int sem_trywait(sem_t *sem);

// citirea valorii
int sem_getvalue(sem_t *sem, int *pvalue);
```

Semafoarele POSIX au fost prezentate în cadrul laboratorului de comunicare inter-proces.

## Variabila de condiie

Variabilele condiie pun la dispoziie un sistem de notificare pentru fire de execuie, permiându-i unui fir să se blocheze în ateptarea unui semnal din partea unui alt fir. Folosirea corectă a variabilelor condiie presupune un protocol cooperativ între firele de execuie.

Mutexurile (*mutual exclusion locks*) i semafoarele permit blocarea *altor* fire de execuie. Variabilele de condiie se folosesc pentru a bloca firul *curent* de execuie până la îndeplinirea unei condiii.

Variabilele condiie sunt obiecte de sincronizare care-i permit unui fir de execuie să-i suspende execuia până când o condiie (predicat logic) devine adevărată. Când un fir de execuie determină că predicatul a devenit adevărat, va semnala variabila condiie, deblocând astfel unul sau toate firele de execuie blocate la acea variabilă condiie (în funcie de cum se dorete).

O variabilă condiie trebuie întotdeauna folosită împreună cu un mutex pentru evitarea race-ului care se produce când un fir se pregătete să atepte la variabila condiie în urma evaluării predicatului logic, iar alt fir semnalizează variabila condiie chiar înainte ca primul fir să se blocheze, pierzându-se astfel semnalul. Aadar, operaiile de semnalizare, testare a condiiei logice i blocare la variabila condiie trebuie efectuate având ocupat mutexul asociat variabilei condiie. Condiia logică este testată sub protecia mutexului, iar dacă nu este îndeplinită, firul apelant se blochează la variabila condiie, eliberând atomic mutexul. În momentul deblocării, un fir de execuie va încerca să ocupe mutexul asociat variabilei condiie. De asemenea, testarea predicatului logic trebuie făcută într-o buclă, pentru că dacă sunt eliberate mai multe fire deodată, doar unul va reui să ocupe mutexul asociat condiiei. Restul vor atepta ca acesta să-l elibereze, însă este posibil ca firul care a ocupat mutexul să schimbe valoarea predicatului logic pe durata deinerii mutexului. Din acest motiv celelalte fire trebuie să testeze din nou predicatul pentru că altfel i-ar începe execuia presupunând predicatul adevărat, când el este, de fapt, fals.

### Iniializarea/distrugerea unei variabile de condiie

```
// initializare statica a unei variabile de condiie cu atribute implicite
// NB: variabila de conditie nu este eliberata,
//     durata de viata a variabilei de condiie este durata de viata a programului.
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

// semnaturile functiilor de initializare si eliberare de variabile de condiie:
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Ca i la mutexuri:

- dacă parametrul `attr` este nul se folosesc atribute implicite
- trebuie să nu existe nici un fir de execuie în ateptare pe variabila de condiie atunci când aceasta este distrusă, altfel se întoarce `EBUSY`.

### Blocarea la o variabilă condiie

Pentru a-i suspenda execuia i a atepta la o variabilă condiie, un fir de execuie va apela:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

Firul de execuie apelant trebuie să fi ocupat deja mutexul asociat, în momentul apelului. Funcia *pthread_cond_wait* va elibera mutexul i se va bloca, ateptând ca variabila condiie să fie semnalizată de un alt fir de execuie. Cele două operaii sunt efectuate atomic. În momentul în care variabila condiie este semnalizată, se va încerca ocuparea mutexului asociat, i după ocuparea acestuia, apelul funciei va întoarce. Observai că firul de execuie apelant poate fi suspendat, după deblocare, în ateptarea ocupării mutexului asociat, timp în care predicatul logic, adevărat în momentul deblocării firului, poate fi modificat de alte fire. De aceea, apelul *pthread_cond_wait* trebuie efectuat într-o buclă în care se testează valoarea de adevăr a predicatului logic asociat variabilei condiie, pentru a asigura o serializare corectă a firelor de execuie. Un alt argument pentru testarea în buclă a predicatului logic este acela că un apel *pthread_cond_wait* poate fi întrerupt de un semnal asincron (vezi laboratorul de semnale), înainte ca predicatul logic să devină adevărat. Dacă firele de execuie care ateptau la variabila condiie nu ar testa din nou predicatul logic, i-ar continua execuia presupunând greit că acesta e adevărat.

### Blocarea la o variabilă condiie cu timeout

Pentru a-i suspenda execuia i a atepta la o variabilă condiie, nu mai târziu de un moment specificat de timp, un fir de execuie va apela:

```
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
                           const struct timespec *abstime);
```

Funcia se comportă la fel ca *pthread_cond_wait*, cu excepia faptului că dacă variabila condiie nu este semnalizată mai devreme de *abstime*, firul apelant este deblocat, i după ocuparea mutexului asociat, funcia se întoarce cu eroarea *ETIMEDOUT*. Parametrul *abstime* este absolut i reprezintă numărul de secunde trecute de la 1 ianuarie 1970, ora 00:00.

### Deblocarea unui singur fir blocat la o variabilă condiie

Pentru a debloca un singur fir de execuie blocat la o variabilă condiie se va semnaliza variabila condiie astfel:

```
int pthread_cond_signal(pthread_cond_t *cond);
```

Dacă la variabila condiie nu ateaptă niciun fir de execuie, apelul funciei nu are efect i semnalizarea se va pierde. Dacă la variabila condiie ateaptă mai multe fire de execuie, va fi deblocat doar unul dintre acestea. Alegerea firului care va fi deblocat este făcută de planificatorul de fire de execuie. Nu se poate presupune că firele care ateaptă vor fi deblocate în ordinea în care i-au început ateptarea. Firul de execuie apelant trebuie să deină mutexul asociat variabilei condiie în momentul apelului acestei funcii.

Exemplu:

```
pthread_mutex_t count_lock;
pthread_cond_t  count_nonzero;
unsigned count;

decrement_count()
{
pthread_mutex_lock(&count_lock);
while (count == 0)
pthread_cond_wait(&count_nonzero, &count_lock);
      = count - 1;
pthread_mutex_unlock(&count_lock);
}

increment_count()
{
pthread_mutex_lock(&count_lock);
if (count == 0)
pthread_cond_signal(&count_nonzero);
      = count + 1;
pthread_mutex_unlock(&count_lock);
}
```

### Deblocarea tuturor firelor blocate la o variabilă condiie

Pentru a debloca toate firele de execuie blocate la o variabilă condiie, se semnalizează variabila condiie astfel:

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Dacă la variabila condiie nu ateaptă niciun fir de execuie, apelul funciei nu are efect i semnalizarea se va pierde. Dacă la variabila condiie ateaptă fire de execuie, toate acestea vor fi deblocate, dar vor concura pentru ocuparea mutexului asociat variabilei condiie. Firul de execuie apelant trebuie să deină mutexul asociat variabilei condiie în momentul apelului acestei funcii.

### Exemplu de utilizare a variabilelor de condiie

În următorul program se utilizează o barieră pentru a sincroniza firele de execuie ale programului. Bariera este implementată cu ajutorului unei variabile de condiie.

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

```
#include <pthread.h>


#define NUM_THREADS 5


// implementarea unei bariere *nereentrante* cu variabile de conditie
struct my_barrier_t {
  // mutex folosit pentru a serializa accesele la datele interne ale barierei
  pthread_mutex_t lock;
  // variabila de conditie pe care se astepta sosirea tuturor firelor de executie
  pthread_cond_t  cond;
  // numar de fire de executie care trebuie sa mai vina pentru a elibera bariera
  int nr_still_to_come;
};

struct my_barrier_t bar;

void my_barrier_init(struct my_barrier_t * bar, int nr_still_to_come)
{
  pthread_mutex_init(&bar->lock, NULL);
  pthread_cond_init(&bar->cond, NULL);
  // cate fire de executie sunt asteptate la bariera.
  bar->nr_still_to_come = nr_still_to_come;
}

void my_barrier_destroy(struct my_barrier_t * bar)
{
  pthread_cond_destroy(&bar->cond);
  pthread_mutex_destroy(&bar->lock);
}


void *thread_routine(void *arg)
{
  int thd_id = (int) arg;

  // inainte de a lucra cu datele interne ale barierei trebuie sa preluam mutexul
  pthread_mutex_lock(&bar.lock);

  printf("thd %d: before teh barrier\n", thd_id);

  // suntem ultimul fir de executie care a sosit la bariera?
  int is_last_to_arrive = (bar.nr_still_to_come == 1);
  // decrementam numarul de fire de executie asteptate la bariera
  bar.nr_still_to_come --;

  // cat timp mai sunt threaduri care nu au ajuns la bariera, asteptam.
  while (bar.nr_still_to_come != 0)
    // lockul se elibereaza automat inainte de a incepe asteptarea
    pthread_cond_wait(&bar.cond, &bar.lock);

  // ultimul thread ajuns la bariera va semnaliza celelalte threaduri
  if (is_last_to_arrive) {
    printf("   let the flood in\n");
    pthread_cond_broadcast(&bar.cond);
  }


  printf("thd %d: after teh barrier\n", thd_id);

  // la iesirea din functia de asteptare se preia automat mutexul, trebuie eliberat.
```

```
  pthread_mutex_unlock(&bar.lock);
  return NULL;
}


int main(void)
{
  int i;
  pthread_t tids[NUM_THREADS];

  my_barrier_init(&bar, NUM_THREADS);

  for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tids[i], NULL, thread_routine, (void *) i);

  for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tids[i], NULL);

  my_barrier_destroy(&bar);
  return 0;
}
```

```
$ gcc -Wall cond_var.c -pthread
$ ./a.out
thd 0: before teh barrier
thd 2: before teh barrier
thd 3: before teh barrier
thd 4: before teh barrier
thd 1: before teh barrier
   let the flood in
thd 1: after teh barrier
thd 2: after teh barrier
thd 3: after teh barrier
thd 4: after teh barrier
thd 0: after teh barrier
```

Din execuia programului se observă:

- ordinea în care sunt planificate firele de execuie nu este numaidecât cea a creării lor
- ordinea în care sunt trezite firele de execuie ce ateaptă la o variabilă de condiie nu este numaidecât ordinea în care acestea au intrat în ateptare.

## Bariera

Standardul POSIX definete i un set de funcii i structuri de date de lucru cu bariere. Aceste funcii sunt disponibile dacă se definete macro-ul `_XOPEN_SOURCE` la o valoare >= 600.

Cu bariere POSIX, programul de mai sus poate fi simplificat:

```
#define _XOPEN_SOURCE 600
#include <pthread.h>
#include <stdio.h>

pthread_barrier_t barrier;
#define NUM_THREADS 5

void *thread_routine(void *arg)
```

```
{
  int thd_id = (int) arg;
  int rc;

  printf("thd %d: before teh barrier\n", thd_id);

  // toate firele de executie asteapta la bariera.
  rc = pthread_barrier_wait(&barrier);
  if (rc == PTHREAD_BARRIER_SERIAL_THREAD) {
    // un singur fir de execuie (posibil ultimul) va intoarce PTHREAD_BARRIER_SERIAL_THREAD
    // restul firelor de execuie întorc 0 în caz de succes.
    printf("   let the flood in\n", thd_id);
  }
  printf("thd %d: after teh barrier\n", thd_id);
  return NULL;
}


int main()
{
  int i;
  pthread_t tids[NUM_THREADS];

  // bariera este initializata o singura data si folosita de toate firele de executie
  pthread_barrier_init(&barrier, NULL, NUM_THREADS);

  // firele de executie vor executa codul functiei 'thread_routine'
  // in locul unui pointer la date utile, se trimite in ultimul argument
  // un intreg - identificatorul firului de executie
  for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tids[i], NULL, thread_routine, (void *) i);

  // asteptam ca toate firele de executie sa se termine
  for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tids[i], NULL);

  // eliberam resursele barierei
  pthread_barrier_destroy(&barrier);
  return 0;
}

$ gcc -Wall barrier.c -lpthread
$ ./a.out
thd 0: before teh barrier
thd 2: before teh barrier
thd 1: before teh barrier
thd 3: before teh barrier
thd 4: before teh barrier
   let the flood in
thd 4: after teh barrier
thd 2: after teh barrier
thd 3: after teh barrier
thd 0: after teh barrier
thd 1: after teh barrier
```

### Iniializarea/distrugearea unei bariere

```
// pentru a folosi funciile de lucru cu bariere e nevoie să se definească
// _XOPEN_SOURCE la o valoare >= 600. Pentru detalii consultai feature_test_macros(7).
#define _XOPEN_SOURCE 600
```

```
#include <pthread.h>

// attr    -> un set de adribute, poate fi NULL (se folosesc atribute implicite)
// count   -> numărul de fire de execuie care trebuie să ajungă
//            la barieră pentru ca aceasta să fie eliberată
int pthread_barrier_init(pthread_barrier_t * barrier,
                         const pthread_barrierattr_t * attr,
                         unsigned count);

// trebuie să nu existe fire de execuie în ateptare la barieră
//   înainte de a apela funcia _destroy,
// altfel, se întoarce EBUSY i nu se distruge bariera.
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

### Ateptarea la o barieră

```
#define _XOPEN_SOURCE 600
#include <pthread.h>
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

Dacă bariera a fost creată cu `count=N`, primele `N-1` fire de execuie care apelează `pthread_barrier_wait` se blochează. Când sosete ultimul (al `N`-lea), va debloca toate cele `N-1` fire de execuie. Funcia `pthread_barrier_wait` întoarce trei valori:

- `EINVAL` în cazul în care bariera nu este iniializată (singura eroare definită)
- `PTHREAD_BARRIER_SERIAL_THREAD` în caz de succes, un singur fir de execuie va întoarce valoarea aceasta nu e specificat care este acel fir de execuie (nu e obligatoriu să fie ultimul ajuns la barieră)
- `0` valoare întoarsă în caz de succes de celelalte `N-1` fire de execuie.

# Mecanisme de sincronizare a firelor de execuie Windows

Pentru sincronizarea firelor de execuie Windows sunt disponibile o serie de obiecte de sincronizare la care firele de execuie pot atepta prin apelarea uneia din funciile de ateptare (*WaitForSingleObject*, *WaitForMultipleObjects* ori *SignalObjectAndWait*).

Obiectele de sincronizare Semaphore, Mutex, Event i WaitableTimer pot fi folosite atât pentru sincronizarea proceselor cât i a firelor de execuie. Ele au fost deja introduse în laboratoarele trecute.

În Windows mai există un mecanism de sincronizare care este disponibil doar pentru firele de execuie ale aceluiai proces, i anume *CriticalSection*. Se recomandă folosirea *CriticalSection* pentru excluderea mutuală a firelor de execuie ale aceluiai proces, fiind mai eficient decât *Mutex* sau *Semaphore*.

Win32 API pune la dispoziie un mecanism de acces sincronizat la variabile partajate între fire de execuie prin intermediul funciilor interlocked (*Interlocked Variable Access*), precum i operaii atomice de inserare i tergere în liste simplu înlănuite (*Interlocked Singly Linked Lists*).

## Mutex Win32

Subiectul a fost tratat în laboratorul de comunicaie inter-proces

```
// crează un mutex
HANDLE CreateMutex(LPSECURITY_ATTRIBUTES lpMutexAttributes, BOOL bInitialOwner, LPCTSTR lpName );

// deschide un mutex (identificat prin nume)
HANDLE OpenMutex(DWORD dwDesiredAccess, BOOL bInheritHandle, LPCTSTR lpName);

// eliberează un mutex ocupat
BOOL ReleaseMutex(HANDLE hMutex);
```

## Semafor Win32

Subiectul a fost tratat în laboratorul de comunicaie inter-proces

```
// crează un semafor
HANDLE CreateSemaphore(LPSECURITY_ATTRIBUTES semattr, LONG initial_count,
                       LONG maximum_count, LPCTSTR name);

// deschide un semafor existent
HANDLE OpenSemaphore(DWORD dwDesiredAccess, BOOL bInheritHandle, LPCTSTR name);

// incrementeare contor semafor cu 'lReleaseCount'
BOOL ReleaseSemaphore(HANDLE hSemaphore, LONG lReleaseCount, LPLONG lpPreviousCount);
```

## Seciune critică

Obiectele *CriticalSection* sunt echivalente mutexurilor POSIX de tip *RECURSIVE*. Obiectele *CriticalSection* sunt folosite pentru excluderea mutuală a accesului firelor de execuie ale aceluiai proces la o seciune critică de cod care conine operaii asupra unor date partajate. Un singur fir de execuie va fi activ la un moment dat în interiorul seciunii critice, i dacă mai multe fire ateaptă să intre, nu este garantată ordinea lor de intrare, totui sistemul va fi echitabil faă de toate.

Operaiile care se pot efectua asupra unei seciuni critice sunt: intrarea, intrarea neblocantă, ieirea din seciunea critică, iniializarea i distrugerea.

Pentru serializarea accesului la o seciune de cod critică, fiecare fir de execuie va trebui să intre într-un obiect *CriticalSection* la începutul seciunii i să-l părăsească la sfâritul ei. În acest fel, dacă două fire de execuie încearcă să intre în *CriticalSection* simultan, doar unul dintre ele va reui, i îi va continua execuia în interiorul seciunii critice, iar celălalt se va bloca pînă când obiectul *CriticalSection* va fi părăsit de primul fir. Aadar, la sfâritul seciunii, primul fir trebuie să părăsească obiectul *CriticalSection*, permiându-i celuilalt intrarea.

Pentru excluderea mutuală se pot folosi atât obiecte *Mutex*, cât i obiecte *CriticalSection*; dacă sincronizarea trebuie făcută doar între firele de execuie ale aceluiai proces este recomandată folosirea *CriticalSection*, fiind mai un mecanism mai eficient. Operaia de intrare în *CriticalSection* se traduce într-o singură instruciune de asamblare de tip test-and-set-lock (TSL). *CriticalSection* este echivalentul futexului din Linux.

### Iniializarea/distrugerea unei seciuni critice

Alocarea memoriei pentru o seciune critică se face prin declararea unui obiect `CRITICAL_SECTION`. Acesta nu va putea fi folosit, totui, înainte de a fi iniializat.

```
// initializează o seciune critică cu un contor de spin implicit = 0
void InitializeCriticalSection(LPCRITICAL_SECTION pcrit_sect);

// permite specificarea contorului de spin
BOOL InitializeCriticalSectionAndSpinCount(LPCRITICAL_SECTION pcrit_sect, DWORD dwSpinCount);

// modifică contorul de spin al unei seciuni critice
DWORD SetCriticalSectionSpinCount(LPCRITICAL_SECTION pcrit_sect, DWORD dwSpinCount);

// distrugerea unei seciuni critice
void DeleteCriticalSection(LPCRITICAL_SECTION pcrit_sect);
```

Un obiect `CRITICAL_SECTION` nu poate fi copiat ori modificat după iniializare. De asemenea, un obiect `CRITICAL_SECTION` nu trebuie iniializat de două ori, în caz contrar, comportamentul său fiind nedefinit.

Contorul de spin are sens doar pe sistemele multiprocesor (SMP) (este ignorat pe sisteme uniprocesor). Contorul de spin reprezintă numărul de cicli pe care îi petrece un fir de execuie pe un procesor în busy-waiting, înainte de a-i suspenda execuia la un semafor asociat seciunii critice în ateptarea eliberării acesteia. Scopul ateptării unui număr de cicli în busy-waiting este evitarea blocării la semafor în cazul în care seciunea critică se eliberează în intervalul respectiv, deoarece blocarea la semafor are impact asupra performanelor. Folosirea contorului de spin este recomandată mai ales în cazul unei seciuni critice scurte, accesate foarte des.

### Utilizarea seciunilor critice

Seciunile critice Windows au comportamentul mutexurilor POSIX de tip *RECURSIVE*. Un fir de execuie care se află deja în seciunea critică **nu se va bloca** dacă apelează din nou *EnterCriticalSection*, însă va trebui să părăsească seciunea critică de un număr de ori egal cu cel al ocupărilor, pentru a o elibera.

```
// similar cu pthread_mutex_lock() pentru mutexuri RECURSIVE
void EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);


// similar cu pthread_mutex_unlock() pentru mutexuri RECURSIVE
void LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection);


// pentru a folosi TryEnterCriticalSection trebuie definit
//  _WIN32_WINNT >= 0x0400 înainte de a include prima dată <windows.h>
#define _WIN32_WINNT 0x0400
#include <windows.h>

// similar cu pthread_mutex_trylock() pentru mutexuri RECURSIVE
// - întoarce FALSE (=0) dacă seciunea critică este ocupată de alt
//   fir de execuie i NU blochează firul curent de execuie
// - întoarce o valoarea nenulă dacă seciunea critică era liberă
//   sau ocupată tot de acest fir de execuie
BOOL TryEnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

În cadrul unui fir de execuie, numărul apelurilor `LeaveCriticalSection` trebuie să fie egal cu numărul apelurilor `EnterCriticalSection`, pentru a elibera în final seciunea critică. Dacă un fir de execuie care nu a intrat în seciunea critică apelează `LeaveCriticalSection`, se va produce o eroare care va face ca firele care au apelat `EnterCriticalSection` să atepte pentru o perioadă nedefinită de timp.

### Exemplu seciuni critice

```
// seciune critică globală
CRITICAL_SECTION CriticalSection;


DWORD ThreadProc( LPVOID * param )
{
  // blochează firul de execuie până la intrarea în seciunea critică
  // similar cu un apel pthread_mutex_lock(&mutex) pentru un mutex de tip RECURSIVE
  EnterCriticalSection(&CriticalSection);

  // folosirea datelor protejate de seciunea critică
  // ...

  // ieirea din seciunea critică
  // similar cu un apel pthread_mutex_unlock(&mutex) pentru un mutex de tip RECURSIVE
  LeaveCriticalSection(&CriticalSection);
}

int main()
{
  // seciunea critică trebuie iniializată o singură dată i
  // este folosită de toate firele de execuie
  InitializeCriticalSection(&CriticalSection);

  // execuie fire de execuie
  // ...

  // eliberare seciune critică
  DeleteCriticalSection(&CriticalSection);
  return 0;
}
```

## Operaii atomice cu variabile partajate (Interlocked Variable Access)

Funciile `interlocked` pun la dispoziie un mecanism de sincronizare a accesului la variabile partajate între mai multe fire de execuie. Funciile pot fi apelate de fire de execuie ale unor procese diferite, pentru variabile aflate într-un spaiu de memorie partajată. Funciile `interlocked` reprezintă cel mai simplu mod de evitare a race-ului care apare când două fire de execuie modifică aceeai variabilă.

Operaiile atomice asupra variabilelor partajate:

- incrementare/decrementare

```
// incrementează valoarea indicată de 'lpAddend' i întoarce valoarea incrementată.
LONG InterlockedIncrement(LONG volatile *lpAddend);

// decrementează valoarea indicată de 'lpDecend' i întoarce valoarea decrementată.
```

```
LONG InterlockedDecrement(LONG volatile *lpDecend);
```

- atribuirea atomică a unei valori unei variabile partajate.

```
// funciile următoare vor întoarce vechea valoare a variabilei.

// va scrie valoarea întreagă 'Value' în zona indicată de 'Target'
LONG InterlockedExchange(LONG volatile *Target, LONG Value);

// va adăuga 'Value' la 'Addend'
LONG InterlockedExchangeAdd(LPLONG volatile Addend, LONG Value);

// va atribui pointerul 'Value' pointerului indicat de pointerul 'Target'.
PVOID InterlockedExchangePointer(PVOID volatile *Target,PVOID Value);
```

- atribuirea atomică după o condiie a unei valori variabilei partajate

```
// Compare & Exchange LONG
LONG InterlockedCompareExchange(LONG volatile * dest, LONG exchange, LONG comp);

// Compare & Exchange Pointer
PVOID InterlockedCompareExchangePointer(PVOID volatile * dest, PVOID exchange, PVOID comp);
```

`InterlockedCompareExchange` va compara `dest` cu `comp`; dacă sunt egale îi va atribui lui `dest` valoarea `exchange`. Testul i atribuirea vor fi executate într-o singură operaie atomică. Pentru variabile de tip pointer se va folosi `InterlockedCompareExchangePointer`. Comportamentul este echivalent cu:

```
 atomicly_do {                      // execută atomic tot blocul următor
    tmp = *dest;                    // copiază valoarea din *dest
    if (tmp == comp) {              // dacă e egală cu valoarea lui 'comp'
       * destination = exchange;    // atunci scrie valoarea 'exchange' în *dest
    }
 }
```

# Windows Thread Pooling

Pentru a facilita dezvoltarea de aplicaii eficiente bazate pe fire de execuie, sistemul de operare Windows pune la dispoziie mecanismul *thread pooling*. Utilizarea *thread pooling* este benefică în cazul unei aplicaii bazată pe fire de execuie care au de îndeplinit taskuri relativ scurte. Prin utilizarea *thread pooling*, fiecare task de efectuat va fi atribuit unui fir de execuie din pool, eliminându-se calculele suplimentare impuse de crearea i terminarea unui fir de execuie. Prin task se înelege o procedură executată de un fir de execuie din *thread pool*.

Există două modalităi prin care o aplicaie poate specifica taskurile care dorete să fie executate de fire de execuie din *thread pool*.

- se pot adăuga taskuri ce vor fi executate imediat ce se eliberează un fir de execuie din *thread pool*
- se pot adăuga operaii de ateptare care au asociată o funcie *callback* ce urmează a fi executată la sfâritul unui timeout de unul din firele de execuie din *thread pool*. Din această categorie fac parte operaiile de ateptare a terminării unei intrări/ieiri asincrone, operaiile de ateptare a expirării unui *TimerQueue Timer* i funciile de ateptare înregistrate.

Dacă vreuna din funciile executate într-un *thread-pool* apelează `TerminateThread` comportamentul nu este definit.

## Adăugarea de taskuri la *thread pool*

### Ateptarea unei operaii de intrare/ieire asincrone

Pentru a adăuga la *thread pool* un task care se va executa la finalul unei operaii de intrare/ieire asincrone pe un anumit *FileHandle*, se va apela funcia:

```
// înregistrează o funcie ce va fi chemată când se încheie o
// operaie de IO asincron pe fiierul identificat prin FileHandle.
// Pot fi înregistrate mai multe funcii i vor fi chemate toate
// când se încheie operaia IO asincronă. Ordinea în care sunt apelate
// nu este specificată.
BOOL BindIoCompletionCallback(
HANDLE                          FileHandle,
        LPOVERLAPPED_COMPLETION_ROUTINE Function,
ULONG                           Flags);

// semnătura funciei înregistrate să fie executată la încheierea operaia AIO
VOID CALLBACK FileIOCompletionRoutine(
DWORD dwErrorCode,
DWORD dwNumberOfBytesTransfered,
LPOVERLAPPED lpOverlapped);
```

### Adăugarea unui task pentru execuie imediată

Pentru a adăuga la *thread pool* un task care să fie executat imediat se va apela funcia:

```
BOOL QueueUserWorkItem(
        LPTHREAD_START_ROUTINE Function, // funcia de executat
PVOID Context, // pointer ce va fi pasat funciei ca argument
ULONG Flags);  // tipul rutinei (IO, NON-IO, funcia ateaptă mult, etc.)

// Semnătura funciei e identică cu semnătura funciilor executate cu CreateThread
DWORD WINAPI ThreadProc(LPVOID param);
```

## Timer Queues

Obiectele `TimerQueue` reprezintă cozi de timere. Ele conin obiecte `TimerQueueTimer` care au asociată o funcie `callback` ce va fi executată de un fir de execuie din `thread pool` la expirarea timerului.

### Crearea/distrugerea unei cozi de timere

```
#define _WIN32_WINNT 0x0500
#include <windows.h>

// întoarce un handle la coada de timere
HANDLE CreateTimerQueue(void);


// marchează coada pentru tergere, dar *NU* ateaptă
```

```
// ca toate callbackurile asociate cozii să se termine
BOOL DeleteTimerQueue(HANDLE TimerQueue);

/**
 * CompletionEvent = NULL - marchează coada pentru tergere i iese imediat (ca DeleteTimerQueue)
 * CompletionEvent = INVALID_HANDLE_VALUE - funcia ateaptă să se încheie toate callbackurile.
 * CompletionEvent = un handle de tip Event - un obiect Event care va fi
 *                   trecut în starea SIGNALED când se încheie toate callbackurile.
 */
BOOL DeleteTimerQueueEx(HANDLE TimerQueue, HANDLE CompletionEvent);
```

### Crearea unui timer

Pentru crearea unui timer se va apela funcia:

```
BOOL CreateTimerQueueTimer(
PHANDLE phNewTimer, // aici întoarce un HANDLE la timerul nou creat
HANDLE TimerQueue,  // coada la care este adăugat timerul.
                            // Dacă e NULL se folosete o coadă implicită.
        WAITORTIMERCALLBACK CallBack, // callback de executat
PVOID Parameter,    // parametru trimis callbackului
DWORD DueTime,      // timerul va expira prima dată după 'DueTime' milisec.
DWORD Period,       // apoi timerul va expira periodic după 'Period' milisec.
ULONG Flags);       // tipul callbackului: IO/NonIO, EXECUTEONLYONCE, .a.


// semnătura unui callback
VOID WaitOrTimerCallback(PVOID lpParameter, BOOLEAN TimerOrWaitFired);


// modificarea timpului de expirare al unui timer
BOOL ChangeTimerQueueTimer(
HANDLE TimerQueue, // coada la care este adăugat timerul.
                           // Dacă e NULL se folosete o coadă implicită.
HANDLE Timer,  // HANDLE la timerul de modificat
ULONG DueTime, // timerul va expira prima dată după 'DueTime' milisec.
ULONG Period); // apoi timerul va expira periodic după 'Period' milisec.


// dezactivarea unui timer
BOOL CancelTimerQueueTimer(HANDLE TimerQueue,HANDLE Timer);

// dezactivarea I distrugerea unui timer.
// CompletionEvent e similar cu cel din DeleteTimerQueueEx.
BOOL DeleteTimerQueueTimer(HANDLE TimerQueue, HANDLE Timer,HANDLE CompletionEvent);
```

## Registered Wait Functions

Funciile de ateptare înregistrate sunt funcii de ateptare executate de un fir de execuie din *thread pool*. În momentul în care obiectul de sincronizare după care se ateaptă trece în starea *signaled*, se va executa rutina *callback* asociată funciei de ateptare înregistrate, de un fir de execuie din *thread pool*. În mod implicit, funciile de ateptare înregistrate se rearmează automat i rutinele *callback* sunt executate de fiecare dată când obiectul de sincronizare după care se ateaptă trece în starea *signaled*, sau intervalul de timeout expiră. Acest lucru se repetă până când înregistrarea funciei de ateptare este anulată. Se poate seta, însă, ca funcia de ateptare înregistrată să se execute o singură dată.

### Înregistrarea unei funcii de ateptare

Pentru înregistrarea în *thread pool* a unei funcii de ateptare se va apela funcia:

```
BOOL RegisterWaitForSingleObject(
PHANDLE phNewWaitObject,
HANDLE hObject,
        WAITORTIMERCALLBACK Callback,
PVOID Context,
ULONG dwMilliseconds,
ULONG dwFlags
);
```

De fiecare dată când *hObject* trece în starea *signaled*, i la fiecare *dwMilliseconds*, rutina *Callback* va fi executată cu parametrul *Context* de un fir de execuie din *thread pool*. Rutina *Callback* trebuie să nu apeleze *TerminateThread* i să aibă următoarea signatură:

```
VOID CALLBACK WaitOrTimerCallback(
PVOID lpParameter,
BOOLEAN TimerOrWaitFired
);
```

Parametrul *TimerOrWaitFired* va specifica dacă execuia rutinei *Callback* s-a declanat în urma trecerii în starea *signaled* a obiectului de sincronizare, sau în urma expirării intervalului de timeout specificat.

Prin intermediul parametrului *dwFlags* se pot transmite caracteristici ale firului de execuie care va executa rutina *Callback*, precum i dacă funcia de ateptare trebuie să se execute doar o singură dată. Funcia va întoarce, prin parametrul *phNewWaitObject*, un handle ce va fi folosit pentru deînregistrarea funciei de ateptare.

### Deînregistrarea unei funcii de ateptare

Pentru a anula înregistrarea unei funcii de ateptare se va apela una din funciile:

```
BOOL UnregisterWait(HANDLE WaitHandle);
BOOL UnregisterWaitEx(HANDLE WaitHandle,HANDLE CompletionEvent);
```

Orice funcie de ateptare înregistrată va trebui deînregistrată prin apelul uneia din funciile de mai sus.

Funcia *UnregisterWaitEx* va semnaliza eventul *CompletionEvent* în cazul în care se termină cu succes i rutina de callback s-a terminat cu succes. Dacă valoarea lui *CompletionEvent* nu este NULL, atunci funcia va atepta finalizarea operaiei de ateptare i terminarea rutinei asociate.

# Quiz

Pentru autoevaluare raspundei la întrebările din acest quiz.

# Exercitii pre-laborator

1. Se da urmatorul program:

```
int a;
int main(){
  a++;
  return 0;
}
```

- Obtineti fisierul in limbaj de asamblare corespunzator. ( folositi optiunea -S la compilarea cu gcc ).
- Identificati secventa de cod care realizeaza incrementarea.
- Cate operatii in limbaj de asamblare sunt necesare?

2. Care este diferenta intre un deadlock si un livelock?

## Prezentare

Pentru a urmări mai uor noiunile expuse la începutul laboratorului folosii această prezentare (pdf) (odp).

# Exerciii

Arhiva de sarcini a laboratorului.

## Linux

1. 1 punct `unsafe`
   - Creai `nr=SINGLE_INC_THREADS_NO` fire de execuie care apelează funcia `single_increment` i `nr=DOUBLE_INC_THREADS_NO` fire de execuie care apelează funcia `double_increment`.
   - Este necesară protejarea variabilei `a` dacă funcia `single_increment` este rulată din mai multe fire de execuie? Dar variabila `b`?
   - Corectai (dacă este cazul) funciile `single_increment` i `double_increment` folosind muteci POSIX.
   - Hints
     - Folosii `make test` sau `./test.sh` pentru a testa programul.
2. 1 punct `blocked`
   - Inspectai fiierul `blocked.c`, compilai i executai binarul (repetai până detectai blocarea programului). Programul crează două fire de execuie care caută un număr magic, fiecare în intervalul propriu.
   - Fiecare fir de execuie, pentru fiecare valoare din intervalul propriu verifică dacă este valoarea căutată:
     - dacă da, marchează un câmp `found` pentru a întiina i celălalt thread că a găsit numărul căutat,
     - dacă nu, verifică câmpul `found` al structurii celuilalt fir de execuie pentru a vedea dacă acesta a găsit deja numărul căutat.
   - Determinai cauza blocării i reparai programul i *explicai* soluia.
   - HINT: putei utiliza `helgrind` unul din toolurile `valgrind` pentru a detecta problema: `valgrind --tool=helgrind ./blocked`.
3. 2.5 puncte `h2o`
   - Formula de generare a apei este `2H + O = H2O`. Atomii sunt reprezentai de fire de execuie.

- Creai `H_THREADS` fire de execuie care apelează funcia `hReady`, i `O_THREADS` fire de execuie care apelează funcia `oReady`.
- Un `H` nu se combina (thread-ul corespunzător lui nu se termină) până când nu există un alt `H` i un `O`. De asemenea, un `O`, pentru a se combina are nevoie de doi `H`.
- Completai funciile `hReady` i `oReady` astfel încât să obinei apă.
- HINTS: utilizai semafoare.

4. 2.5 puncte `prodcons`
   - Rezolvai problema producător-consumator folosind variabile de condiie i muteci. Producătorul va produce un anumit număr de întregi pe care îi va pune într-un buffer de lungime limitată de unde consumatorul Îi va lua.
   - Completai funciile `consumer_fn` i `producer_fn`.
   - HINTS
     - Folosii tipul `buffer_t` pentru a reprezenta buffer-ul. Câmpul `count` reprezintă poziia curentă de inserat în buffer.
     - Folosii funciile `init_buffer`, `insert_item`, `remove_item`, `is_buffer_full` i `is_buffer_empty` pentru a lucra cu buffer-ul.

## Windows

1. 1 punct `unsafe`
   - Creai `nr=SINGLE_INC_THREADS_NO` fire de execuie care apelează funcia `single_increment` i `nr=DOUBLE_INC_THREADS_NO` fire de execuie care apelează funcia `double_increment`.
   - Este necesară protejarea variabilei `a` dacă funcia `single_increment` este rulată din mai multe fire de execuie? Dar variabila `b`?
   - Corectai (dacă este cazul) funciile `single_increment` i `double_increment` folosind seciuni critice.
2. 1 puncte `interlocked`
   - Creai `THREAD_NO` fire de execuie care incrementează circular o variabilă (când se ajunge la o limită se resetează la 0).
   - Folosii `Interlocked Variables` deoarece mecanismul e mai rapid decât o incrementare normală protejată cu `Mutex` sau `CRITICAL_SECTION`.
   - Incrementarea circulară se va face în funcia `threadfn`.
   - HINT: Folosii `InterlockedCompareExchange`
3. 2 puncte `timerQueue`
   - Creai un `TimerQueueTimer` a cărui rutină callback să fie declanată de exact 3 ori, din secundă în secundă. După 3 declanări se va dezactiva timerul i se vor distruge toate resursele create.
   - HINT: Trebuie să sincronizai rutina timer-ului cu rutina care îl dezactivează; pentru aceasta putei folosi un semafor.
4. 2 puncte `registeredWait`
   - Rezolvai problema precedentă folosind funcii de ateptare înregistrate. Mai exact rutina callback a funciei de ateptare să se declaneze de exact 3 ori din secundă în secundă.
   - HINT: Trebuie să sincronizai funcia înregistrată cu rutina care o dezînregistrează; pentru aceasta putei folosi un semafor.

# Soluii

Soluții exerciții laborator 9