## 3.2. Інтерфейс прикладного програмування POSIX Threads (Pthreads)

### 3.2.1. Використання POSIX Threads

POSIX Threads (Pthreads) – це стандарт POSIX, який визначає програмний інтерфейс для створення та використання програмних потоків [15]. Існує кілька реалізацій цього стандарту, зокрема

1) LinuxThreads – перша реалізація Pthreads в ОС Linux; починаючи з glibc 2.4 не підтримується.

2) Native POSIX Thread Library (NPTL) – сучасна реалізація Pthreads в ОС Linux, яка у порівнянні з LinuxThreads більш повно відповідає стандарту POSIX Threads.

3) GNU Portable Threads (GNU Pth);

4) Open Source POSIX Threads for Win32 (pthreads-w32) та ін.

В ОС Linux за допомогою команди getconf можна визначити реалізацію POSIX Threads та її версію, наприклад:

```
$ getconf GNU_LIBPTHREAD_VERSION
NPTL 2.28
```

Pthreads визначає набір типів, функцій та констант для роботи з програмними потоками на мові програмування С. Для роботи з Pthreads необхідно мати файл pthread.h та відповідну бібліотеку функцій. Один процес може містити кілька потоків, усі вони виконують одну і ту ж програму. Ці потоки мають однакову глобальну пам’ять (сегменти даних та область пам’яті, що виділяється динамічно), але кожен потік має власний стек.

Стандартом Pthreads визначено більше 70-ти функцій, які за призначенням можна поділити на наступні основні групи (в дужках вказані префікси у назвах відповідних функцій):

1) Функції управління потоками (pthread_);

2) Функції управління атрибутами потоків(pthread_attr_, pthread_get<paramname>, pthread_set<paramname>);

3) Функції управління м’ютексами (pthread_mutex_, pthread_mutexattr_);

4) Функції управління умовними змінними (pthread_cond_, pthread_condattr_);

5) Функції бар’єрної синхронізації (pthread_barrier_, pthread_barrierattr_);

6) Функції блокування читання-запису (pthread_rwlock_, pthread_rwlockattr_)

7) Функції спін-блокування (spinlock) (pthread_spin_).

Більшість функцій Pthreads повертають 0 в разі успіху або номер помилки в протилежному випадку. Функції pthreads не встановлюють значення змінної помилки errno.

Важливим питанням при програмуванні на рівні потоків є використання потоко-безпечних функцій (thread-safe functions). Потоко-безпечна функція – це функція, яку безпечно викликати у декількох потоках одночасно (тобто вона буде давати однакові результати незалежно від того, в якому потоці вона була викликана). Стандарти POSIX.1-2001 та POSIX.1-2008 визначають перелік стандартних функцій, які мають бути потоко-безпечними.

### 3.2.2. Створення та ідентифікація програмних потоків

Для створення нового програмного потоку використовується функція pthread_create() (рис.3.5). В параметрі thread (покажчик на ідентифікатор потоку типу pthread_t) повертається ідентифікатор створеного потоку; параметр attr – це вказівник на атрибутний об’єкт (змінна типу pthread_attr_t), яким задаються атрибути нового потоку; параметр start_routine – це назва початкової функції, яку буде виконувати потік; параметр arg – це вказівник на аргумент (типу void), який передається функції start_routine.

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

Рис.3.5. Опис функції pthread_create().

В разі успішного виконання функція pthread_create() повертає 0, а в разі помилки повертає не нульове значення, яке вказує на тип помилки. Також в разі успішного виконання через параметр thread функція поверне ідентифікатор створеного потоку. За допомогою другого параметру attr можна змінити атрибути нового потоку, що встановлені по замовченню. Як правило, замість цього другим аргументом вказують NULL, користуючись атрибутами потоку, які встановлені по замовченню. Щойно створений потік починає виконання з функції start_routine. Ця функція приймає єдиний аргумент, arg, який є нетипізованим вказівником. Якщо необхідно передати функції start_routine значний обсяг інформації, то її слід зберегти у вигляді структури й передати вказівник на структуру в аргументі arg. При створенні нового потоку не можна заздалегідь припускати, хто першим отримає керування – щойно створений потік або потік, що викликав функцію pthread_create.

Згідно стандарту POSIX.1 програмні потоки розділяють (мають однакові значення) наступних атрибутів процесу:

- ідентифікатор процесу (process ID),
- ідентифікатор батьківського процесу (parent process ID);
- ідентифікатори групи процесів та сесії (process group ID and session ID);
- термінал запуску процесу (controlling terminal);
- ідентифікатори користувача та групи користувачів (user and group IDs);
- дескриптори відкритих файлів (open file descriptors);
- замки на записи (record locks);
- диспозиції сигналів (signal dispositions);
- маска режиму створення файлу (file mode creation mask);
- поточна і коренева директорія (current and root directory);
- інтервальні таймери (interval timers) і таймери POSIX (POSIX timers);
- значення поправки до статичного пріоритету (nice value);
- ресурсні обмеження (resource limits);
- виміри споживання процесорного часу та системних ресурсів (measurements of the consumption of CPU time and resources).

Разом зі стеком потоку, стандарт POSIX.1 визначає наступні унікальні для кожного програмного потоку атрибути:

- ідентифікатор потоку (thread ID);
- маска сигналів (signal mask);
- змінна помилки (the errno variable);
- альтернативний стек сигналів (alternate signal stack);
- алгоритм диспетчеризації в режимі реального часу та відповідний пріоритет (real-time scheduling policy and priority);
- можливості здійснення операцій з точки зору прав доступу (capabilities);
- прив’язка до процесора або ядер процесора (CPU affinity).

На рис.3.6. наведено приклад створення програмних потоків управління. В програмі створюються п’ять нових потоків (окрім основного потоку еквівалентного самому процесу) за допомогою функції pthread_create(). Кожний зі створених потоків виводить повідомлення «Hello World!» та свій ідентифікатор (функція PrintHello()).

```
// This program creates five new threads,
// each of which prints its thread number to standard output

#include <pthread.h>
```

```
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;

   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;

   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);

      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

Рис.3.6. Приклад створення п’яти потоків, кожний з яких виводить свій номер.

Кожен з потоків у процесі має свій унікальний ідентифікатор (зберігається у змінній типу pthread_t). Цей ідентифікатор повертається викликом pthread_create(), і, крім цього, потік може отримати власний ідентифікатор викликом функції pthread_self(). Унікальність ідентифікаторів потоків гарантується лише в межах відповідного процесу. У всіх функціях Pthreads, де в якості параметру використовується ідентифікатор, він відноситься до потоку в цьому ж процесі. Система може повторно використовувати ідентифікатор потоку після приєднання завершеного потоку або завершення від’єднаного потоку. В стандарті POSIX зазначено: «Якщо програма намагається використати ідентифікатор потоку, термін дії якого закінчився, то відповідна поведінка не визначена».

Функція pthread_self() (рис.3.7) повертає викликаючому програмному потоку його ідентифікатор. Ця функція є прямим аналогом функції getpid() для процесів. За допомогою функції pthread_equal() (рис.3.8) можна перевірити рівність двох заданих ідентифікаторів потоків (t1 і t2). Якщо ідентифікатори рівні, то функція pthread_equal() поверне ненульове значення, інакше поверне 0.

```
#include <pthread.h>

pthread_t pthread_self(void);
```

Рис.3.7. Опис функції pthread_self().

```
#include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);
```

Рис.3.8. Опис функції pthread_equal().

### 3.2.3. Завершення виконання програмних потоків

Створений програмний потік може завершити своє виконання одним з наступних способів:

1) викликом функції pthread_exit();

2) звичайним завершенням стартової функції <start_routine>;

3) завершитись у відповідь на запит на завершення, отриманий від іншого потоку (функція pthread_cancel());

4) викликом функції exit() в будь-якому потоці, який також зупинить сам процес разом з усіма іншими потоками.

Функція pthread_exit() (рис.3.9) завершує виконання викликаючого програмного потоку. Ця функція ніколи не повертає керування. В параметрі retval можна передати код завершення потоку, який буде отриманий іншим потоком за допомогою функції очікування на завершення потоку pthread_join().

```
#include <pthread.h>

void pthread_exit(void *retval);
```

Рис.3.9. Опис функції pthread_exit().

Функція pthread_cancel() (рис.3.10) надсилає запит на завершення вказаному потоку (thread). Відправивши запит на завершення потоку, виклик pthread_cancel() негайно завершується, не чекаючи на завершення вказаного потоку (thread). Як потік відреагує на цей запит (коли і як він завершиться), залежить від стану завершення (cancellation state) та типу завершення (cancellation type), що встановлені для цього потоку.

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

Рис.3.10. Опис функції pthread_cancel().

Функція pthread_testcancel() (рис.3.11) перевіряє наявність запиту на завершення, який знаходиться в черзі очікування на надходження (pending cancellation request). Виклик pthread_testcancel() створює точку завершення (cancellation point) у викликаючому потоці, внаслідок чого потік, який до цього часу виконував код без точок завершення, відповість на запит на завершення у разі його наявності.

Якщо можливість завершення по запиту відключена (за допомогою функції pthread_setcancelstate()), або запит на завершення відсутній, то виклик phread_testcancel() не має ефекту.

```
#include <pthread.h>

void pthread_testcancel(void);
```

Рис.3.11. Опис функції pthread_testcancel().

За допомогою функції pthread_setcancelstate() (рис.3.12) можна встановити один з двох станів реагування на надходження запитів на завершення:

1) PTHREAD_CANCEL_DISABLE – в цьому стані потік не реагує на запити на завершення; якщо такі запити надходять, то вони ставляться в чергу очікування на надходження;

2) PTHREAD_CANCEL_ENABLE – в цьому стані потік реагує на запити на завершення (цей стан встановлено за замовчуванням).

Стан, який потрібно встановити, задається в параметрі state. Попередній стан повертається в параметрі oldstate. Тимчасове відключення реагування на запити на завершення (PTHREAD_CANCEL_DISABLE) корисно, коли потік виконує ділянку коду, в якій повинні бути виконані всі кроки.

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
```

Рис.3.12. Опис функції pthread_setcancelstate().

За допомогою функції pthread_setcanceltype() (рис.3.13) можна встановити один з двох типів реагування на запит на завершення (для стану PTHREAD_CANCEL_ENABLE):

1) PTHREAD_CANCEL_ASYNCHRONOUS - у відповідь на запит, потік завершується в будь який момент часу, в тому числі негайно після надходження запиту на завершення;

2) PTHREAD_CANCEL_DEFERRED - у відповідь на запит, потік відкладає своє завершення до точки завершення (cancellation point) (цей тип реагування на запит встановлено за замовчуванням).

Тип реагування, який потрібно встановити, задається в параметрі type. Попередній тип реагування повертається в параметрі oldtype. Точка завершення (cancellation point) - це виклик будь-якої стандартної функції з набору функцій, визначених реалізацією стандарту POSIX Threads.

```
#include <pthread.h>

int pthread_setcanceltype(int type, int *oldtype);
```

Рис.3.13. Опис функції pthread_setcanceltype().

### 3.2.4. Очікування на завершення програмних потоків

Для очікування на завершення іншого потоку використовується функція pthread_join() (рис.3.14). Функція pthread_join() призупиняє виконання викликаючого потоку до моменту завершення виконання іншого потоку з ідентифікатором thread. Якщо покажчик retval не нульовий, то через нього повертається код завершення відповідного потоку. Якщо потік був примусово завершений (у відповідь на запит на завершення), то за адресою retval буде записано значення PTHREAD_CANCELED. Усі потоки в процесі є одноранговими: будь-який потік може приєднатись (join) до будь-якого іншого потоку в процесі. Функція pthread_join() є прямим аналогом функції waitpid() для процесів.

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);
```

Рис.3.14. Опис функції pthread_join().

Функція pthread_detach() (рис.3.15) робить вказаний потік від’єднаним, тобто таким на завершення якого не може очікувати жоден інший потік (викликом pthread_join). Від’єднаний потік є віддаленим аналогом процеса-демона, який виконується у фоновому режимі. Після «самостійного» завершення від’єднаного потоку всі захоплені ним системні ресурси автоматично звільняються. По замовченню усі потоки створюються як приєднані. При необхідності змінити власний статус потоку параметром функції pthread_detach() вказується функція pthread_self(): pthread_detach(pthread_self()).

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

Рис.9.15. Опис функції pthread_detach().

Якщо нас не цікавить момент часу і код завершення деякого потоку, ми можемо звернутися до функції pthread_detach(), щоб дозволити операційній системі утилізувати ресурси, захоплені цим потоком, після його завершення. Будь-який потік в кінці кінців в момент завершення має “виходити” на pthread_join() чи pthread_detach(). Виклик pthread_detach() може використовуватись для виключення тупикових ситуацій (deadlocks), коли, наприклад, два потоки очікують на завершення один одного.

### 3.2.5. Управління атрибутами програмних потоків

Атрибути потоку задаються у вигляді атрибутного об’єкту (змінна типу pthread_attr_t). Атрибутний об’єкт представляє собою структуру (рис.3.16), яка дозволяє зберігати наступні атрибути потоку:

1) stackaddr – місце розташування стеку потоку (за замовчуванням адресу стеку обирає система);

2) stacksize – розмір стеку потоку (за замовчуванням новий потік має встановлене у системі максимальне значення розміру стеку);

3) detachstate - тип потоку: від’єднаний чи приєднаний (за замовчуванням потік є приєднаним – PTHREAD_CREATE_JOINABLE);

4) inheritsched – режим наслідування параметрів диспетчеризації (за замовчуванням потік наслідує параметри диспетчеризації батьківського потоку – PTHREAD_INHERIT_SCHED);

5) scope – область конкуренції потоку за час процесора (contention scope);

6) schedpolicy – стратегія диспетчеризації (планування паралельного виконання потоку, значення за замовчуванням – SCHED_OTHER);

7) priority – пріоритет потоку (за замовчуванням потік наслідує пріоритет основного потоку свого процесу).

```
typedef struct __pthread_attr pthread_attr_t;

struct __pthread_attr
{
  struct sched_param __schedparam;
  void *__stackaddr;
  size_t __stacksize;
  size_t __guardsize;
  enum __pthread_detachstate __detachstate;
  enum __pthread_inheritsched __inheritsched;
  enum __pthread_contentionscope __contentionscope;
  int __schedpolicy;
};
```

Рис.3.16. Структура pthread_attr_t.

Реалізація стандарту POSIX Threads в системі ОС Linux (Native POSIX Thread Library, NPTL) за замовчуванням визначає значення атрибуту stacksize рівним встановленому у системі максимальному значенню розміру стеку ("stack size" resource limit). Це значення можна дізнатись командою ulimit:

```
$ ulimit -s
8192
```

В наведеному прикладі це значення складає 8M (0x800000 байт).

Атрибут scope – область конкуренції потоку за час процесора (contention scope) – показує спосіб відображення потоку рівня користувача у потік рівня ядра (модель Many-to-One або модель One-to-One) і може приймати одне з двох значень:

1) PTHREAD_SCOPE_PROCESS – локальна область конкуренції в межах процесу, коли потоки одного процесу конкурують за доступ до процесора лише між собою (модель Many-to-One);

2) PTHREAD_SCOPE_SYSTEM – глобальна область конкуренції в межах всієї системи, коли потоки одного процесу конкурують за доступ до процесора з усіма іншими потоками в системі (модель One-to-One).

Значення атрибуту scope визначається типом моделі відображення потоку рівня користувача у потік рівня ядра, яка реалізована у відповідній бібліотеці функцій потоків. Зокрема в Native POSIX Thread Library (NPTL) реалізована модель One-to-One. Відповідно scope має значення PTHREAD_SCOPE_SYSTEM.

Для початкової ініціалізації та знищення атрибутного об’єкту використовуються функції int pthread_attr_init() та pthread_attr_destroy() (рис.3.17)

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

Рис.3.17. Опис функцій pthread_attr_init() та pthread_attr_destroy().

Після ініціалізації атрибутного об’єкту значення окремих атрибутів можна
1) подивитися за допомогою функцій типу pthread_attr_get<attr_name>(), та
2) встановити за допомогою функцій типу pthread_attr_set<attr_name>().
Для отримання значень атрибутів потоку використовуються наступні функції:

```
int pthread_attr_getdetachstate(const pthread_attr_t *, int *);
int pthread_attr_getguardsize(const pthread_attr_t *, size_t *);
int pthread_attr_getinheritsched(const pthread_attr_t *, int *);
int pthread_attr_getschedparam(const pthread_attr_t *, struct sched_param *);
int pthread_attr_getschedpolicy(const pthread_attr_t *, int *);
int pthread_attr_getscope(const pthread_attr_t *, int *);
int pthread_attr_getstackaddr(const pthread_attr_t *, void **);
int pthread_attr_getstacksize(const pthread_attr_t *, size_t *);
```

Для встановлення значень атрибутів потоку використовуються наступні функції:

```
int pthread_attr_setdetachstate(pthread_attr_t *, int);
int pthread_attr_setguardsize(pthread_attr_t *, size_t);
int pthread_attr_setinheritsched(pthread_attr_t *, int);
int pthread_attr_setschedparam(pthread_attr_t *,const struct sched_param *);
int pthread_attr_setschedpolicy(pthread_attr_t *, int);
int pthread_attr_setscope(pthread_attr_t *, int);
int pthread_attr_setstackaddr(pthread_attr_t *, void *);
int pthread_attr_setstacksize(pthread_attr_t *, size_t);
```

Як правило, встановлення необхідних значень атрибутів потоку у атрибутному об’єкті (pthread_attr_t) виконується до виклику функції pthread_create() з метою створити новий потік з потрібними атрибутами (вказавши адресу атрибутного об’єкту другим аргументом функції pthread_create()) (рис.3.18).

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
void *start_routine;
void arg
int ret;

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);
ret = pthread_attr_setdetachstate(&tattr,PTHREAD_CREATE_DETACHED);
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

Рис.3.18. Приклад створення “від’єднаного” потоку
(з атрибутом PTHREAD_CREATE_DETACHED).

## 3.3. Синхронізація програмних потоків в POSIX Threads

### 3.3.1. Синхронізація програмних потоків за допомогою м’ютексів

М’ютекс (lock, mutex, mutex lock) - (від англ. **mut**ual **ex**clusion — взаємне виключення) об'єкт синхронізації паралельних задач (процесів, потоків) для організації взаємного виключення при здійсненні операцій доступу до даних (запис, читання). М’ютекс може перебувати в одному з двох станів: закритий (lock) та відкритий (unlock). Відповідно над м’ютексом можна здійснити дві операції, які змінюють його стан на протилежний (рис.3.19): закрити (lock) та відкрити (unlock).

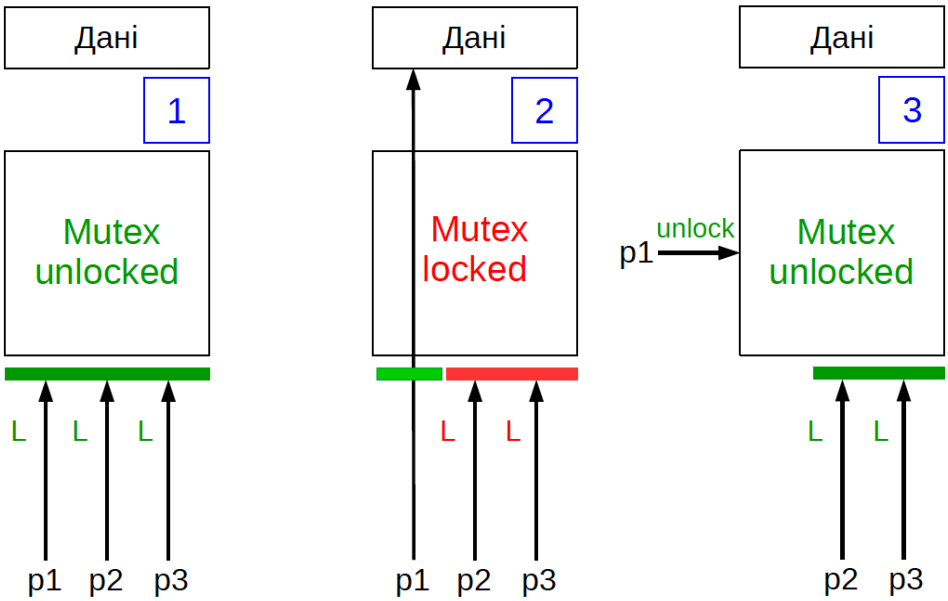


Рис.3.19. Схема використання м’ютексу.

Спроба потоку виконати операцію lock над закритим (захопленим іншим потоком) м’ютексом призводить до блокування потоку до моменту часу, коли м’ютекс буде відкритий (звільнений іншим потоком). На відміну від семафору, логіка використання м’ютексу передбачає, що його може відкрити (звільнити) лише той потік, який його перед цим закрив (захопив). Як правило, у програмі м’ютекс оголошується як глобальна змінна і використовується для “захищеного” доступу потоків до відповідного об’єкту даних. Основне використання м’ютексів - це синхронізація доступу програмних потоків до спільних даних.

Взаємне виключення можна розглядати, як метод організації послідовного (узгодженого) доступу потоків до спільних ресурсів. Приклади ситуацій, які потребують використання взаємного виключення: 1) один потік модифікує змінну, значення якої в цей же час модифікується іншим потоком; 2) потік використовує змінну, значення якої знаходиться в процесі оновлення іншим потоком, відтак отримуючи її старе значення.

Механізм взаємного виключення дозволяє програмісту створити власний протокол послідовного (узгодженого) доступу потоків до спільних даних або ресурсів. При цьому м’ютекс - це свого роду замок, яким можна віртуально захистити якийсь ресурс. Якщо потік хоче змінити або прочитати значення спільного ресурсу, він повинен спочатку отримати до нього доступ, закривши (захопивши) відповідний м’ютекс. Після цього він може виконувати будь які операції зі спільним ресурсом, не турбуючись про те, що інші потоки отримають до нього доступ, оскільки інші потоки будуть чекати своєї черги на доступ. Після того, як потік завершить роботу зі спільним ресурсом, він відкриє (звільнить) м’ютекс, тим самим дозволивши іншим потокам отримати доступ до ресурсу (рис.3.20).

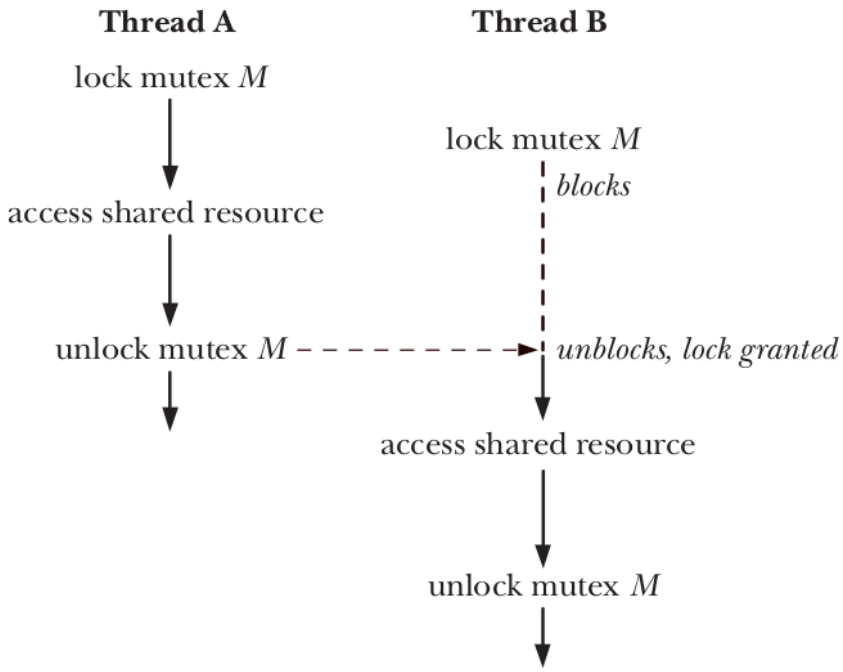


Рис.3.20. Організація послідовного (узгодженого) доступу потоків до спільного ресурсу з використанням м'ютекса.

Наведений протокол послідовного (узгодженого) доступу має бути реалізований у всіх потоках, які виконують операції зі спільним ресурсом. Якщо протокол порушується (наприклад, потік модифікує спільний ресурс без попереднього захоплення м'ютекса), то це, як правило, призводить до помилки. Аналіз програми на наявність помилок такого типу є достатньо складною задачею (складність тим більше, чим більш складну структуру має спільний ресурс, та чим більшу кількість м'ютексів використано у програмі).

В POSIX Threads (Pthreads) об'єкт синхронізації "м'ютекс" створюється у вигляді змінної типу pthread_mutex_t. Після оголошення змінної, м'ютекс потрібно ініціалізувати: або 1) присвоївши їй значення константи PTHREAD_MUTEX_INITIALIZER (рис.3.21), якщо це статична змінна, або 2) викликавши функцію pthread_mutex_init() (рис.3.22), якщо м'ютекс зберігається в області пам'яті, що виділяється динамічно. В останньому випадку перед завершенням програми потрібно звільнити відповідний системний ресурс, викликавши функцію pthread_mutex_destroy() (рис.3.22). Після ініціалізації м'ютекс знаходиться в стані "відкритий" (незахоплений). Як об'єкт даних, м'ютекс може бути розташований або у локальній пам'яті процесу, або у ділянці спільної пам'яті (shared memory) двох чи більше процесів.

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Рис.3.21. Ініціалізація м'ютексу як статичної змінної.

```
#include <pthread.h>

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

int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

Рис.3.22. Опис функцій pthread_mutex_init() та pthread_mutex_destroy().

Для здійснення операцій lock та unlock над м'ютексом використовуються функції pthread_mutex_lock() та pthread_mutex_unlock() (рис.3.23). В якості параметру вказується м'ютекс, над яким здійснюється операція. За допомогою функції pthread_mutex_trylock() (рис.3.23) можна виконати спробу захоплення м'ютекса в режимі без блокування (в асинхронному режимі). Якщо на момент виклику pthread_mutex_trylock() м'ютекс вже захоплений іншим потоком, то функція завершиться негайно і поверне код помилки EBUSY, інакше функція захопить м'ютекс і поверне 0. За допомогою функції pthread_mutex_timedlock() (рис.3.23) можна виконати спробу захоплення м'ютексу на протязі заданого проміжку часу (abs_timeout). Приклад використання м'ютексів для забезпечення послідовного (узгодженого) доступу потоків до глобального лічильника наведено на рис.3.24.

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

int pthread_mutex_lock (pthread_mutex_t *mutex);

int pthread_mutex_trylock (pthread_mutex_t *mutex);

int pthread_mutex_timedlock (pthread_mutex_t *mutex,
                             const struct timespec *abs_timeout);

int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

Рис.3.23. Опис функцій pthread_mutex_lock(), pthread_mutex_trylock(), pthread_mutex_timedlock() та pthread_mutex_unlock().

```
#include <pthread.h>

pthread_mutex_t count_mutex;
int count = 0;

void increment_count() {
    pthread_mutex_lock(&count_mutex);
    count = count + 1;
    pthread_mutex_unlock(&count_mutex);
}

int get_count() {
    int c;

    pthread_mutex_lock(&count_mutex);
    c = count;
    pthread_mutex_unlock(&count_mutex);
    return (c);
}
```

Рис.3.24. Приклад використання м'ютексів для забезпечення послідовного (узгодженого) доступу потоків до глобального лічильника count.

POSIX Threads (Pthreads) надає можливість створювати м'ютекси із заданими атрибутами. Для цього використовується відповідний атрибутний об'єкт - змінна типу pthread_mutexattr_t. Перед використанням, атрибутний об'єкт потрібно ініціалізувати за допомогою функції pthread_mutexattr_init() (рис.3.25). Після завершення роботи з атрибутним об'єктом потрібно звільнити відповідний ресурс викликом pthread_mutexattr_destroy() (рис.3.25). Для отримання значень атрибутів використовуються функції, наведені на рис.3.26. Для встановлення значень атрибутів використовуються функції, наведені на рис.3.27. Типова послідовність кроків по використанню атрибутного об'єкту: створити атрибутний об'єкт,

встановити потрібні значення атрибутів, передати атрибутний об'єкт другим параметром функції pthread_mutex_init () для створення м'ютексу з відповідними атрибутами.

```
#include<pthread.h>
int pthread_mutexattr_init (pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy (pthread_mutexattr_t *attr);
```

Рис.3.25. Опис функцій pthread_mutexattr_init() та pthread_mutexattr_destroy().

```
#include<pthread.h>

int pthread_mutexattr_getpshared (const pthread_mutexattr_t *attr, int *pshared);
int pthread_mutexattr_gettype (const pthread_mutexattr_t  *attr,  int *type);
int pthread_mutexattr_getprotocol (const pthread_mutexattr_t *attr, int *protocol);
int pthread_mutexattr_getprioceiling (const pthread_mutexattr_t *attr, int *prioceiling);
int pthread_mutexattr_getrobust (const pthread_mutexattr_t *attr, int *robust);
int pthread_mutexattr_getrobust_np (const pthread_mutexattr_t *attr, int *robust_np);
```

Рис.3.26. Функції для отримання значень атрибутів м'ютекса.

```
#include <pthread.h>

int pthread_mutexattr_setpshared (pthread_mutexattr_t *attr, int pshared);
int pthread_mutexattr_settype (pthread_mutexattr_t *attr, int type);
int pthread_mutexattr_setprotocol (pthread_mutexattr_t *attr, int protocol);
int pthread_mutexattr_setprioceiling (pthread_mutexattr_t *attr, int prioceiling);
int pthread_mutexattr_setrobust (pthread_mutexattr_t *attr, int robust);
int pthread_mutexattr_setrobust_np (pthread_mutexattr_t *attr, int robust_np);
```

Рис.3.27. Функції для встановлення значень атрибутів м'ютекса.

За допомогою атрибуту pshared (process_shared) визначається область використання м'ютекса. Цей атрибут може мати два значення:

1) PTHREAD_PROCESS_PRIVATE – м'ютекс використовується потоками одного процесу (це значення встановлено за замовчуванням).

2) PTHREAD_PROCESS_SHARED – м'ютекс може бути використаний потоками двох чи більше процесів, що взаємодіють за допомогою механізму спільної пам'яті (shared memory), яка містить цей м'ютекс (така можливість має бути реалізована в операційній системі).

За допомогою атрибуту type визначається тип м'ютекса, який може бути одним з наступних:

1) PTHREAD_MUTEX_NORMAL – "звичайний" м'ютекс, при звертанні до якого не здійснюється перевірка наявності помилок або тупикових ситуацій (deadlocks). Якщо потік намагається захопити м'ютекс, який він вже захопив раніше, то виникає тупикова ситуація (deadlock). Звільнення м'ютекса, який не захоплений або захоплений іншим потоком, має невизначений результат (згідно стандарту Pthreads). У ОС Linux обидві ці операції успішні для цього типу м'ютекса.

2) PTHREAD_MUTEX_ERRORCHECK – м'ютекс, при звертанні до якого здійснюється перевірка наявності помилок або тупикових ситуацій (deadlocks). У всіх трьох ситуаціях (повторне захоплення, звільнення незахопленого м'ютекса, звільнення м'ютекса захопленого іншим потоком) відповідна функція Pthreads повертає помилку. Цей тип м'ютексів, як правило, повільніший за звичайний м'ютекс, але може бути корисним як інструмент відлагодження паралельної програми (наприклад, для того, щоб виявити, де програма порушує правила використання м'ютексів).

3) PTHREAD_MUTEX_RECURSIVE – рекурсивний м'ютекс, який підтримує підрахунок кількості захоплень (locks). Коли потік вперше захоплює м'ютекс, кількість захоплень встановлюється у 1. Кожна наступна операція захоплення м'ютекса тим самим потоком збільшує кількість захоплень, а кожна операція звільнення зменшує цю кількість. М'ютекс звільняється (тобто стає доступним для захоплення іншими потоками) лише тоді, коли

кількість захоплень падає до 0. Звільнення незахопленого м’ютексу не вдається, як і звільнення м’ютексу, який на даний момент захоплений іншим потоком.

4) PTHREAD_MUTEX_DEFAULT - тип м’ютекса, реалізований у системі та встановлений за замовчуванням (спеціальний тип, визначений у стандарті Pthreads задля можливості реалізації м’ютексів інших типів). У ОС Linux м’ютекс PTHREAD_MUTEX_DEFAULT еквівалентний м’ютексу PTHREAD_MUTEX_NORMAL.

### 3.3.2. Синхронізація програмних потоків за допомогою умовних змінних

Умовна змінна (condition variable) – це об’єкт синхронізації, за допомогою якого один потік сигналізує іншому про подію, що відбулася. Цього сигналу може очікувати один або декілька потоків (в заблокованому стані) від деякого іншого потоку. Тобто умовна змінна дозволяє одному потоку інформувати інші потоки про зміну стану деякої спільної змінної (або іншого спільного ресурсу), а також дозволяє іншим потокам очікувати (з блокуванням) на отримання такого повідомлення. Основне використання умовних змінних - це узгодження деякої послідовності операцій, що виконуються різними потоками.

Умовні змінні дозволяють синхронізувати потоки на основі фактичного значення спільних даних. Наприклад, у варіанті програми без умовної змінної потоку потрібно було б постійно перевіряти в режимі опитування (polling), чи не виконалась деяка умова, щоб продовжити своє виконання. На таку перевірку (скажімо у циклі) витрачається багато системних ресурсів, оскільки потік буде постійно зайнятий цією “роботою”. Умовна змінна - це спосіб досягнення цієї ж мети без опитування.

Механізм умовної змінної передбачає використання власне умовної змінної та відповідного м’ютексу для забезпечення атомарності операцій переходу до очікування сигналу, виходу з очікування та перевірки і сигналізації про виконання умови. Типова послідовність дій по використанню умовної змінної у двох потоках наступна (рис.3.28):

1) Потік **A** (очікує на сигнал): виконує обчислення до моменту, коли потрібно дочекатись виконання певної умови; захоплює м’ютекс M; переходить до очікування виконання умови (pthread_cond_wait()), тобто очікує на сигнал від потоку **B** про виконання умови (після виклику функція pthread_cond_wait() звільняє м’ютекс M); очікує, перебуваючи в заблокованому (призупиненому) стані; після отримання сигналу про виконання умови, потік **A** розблоковується (перед завершенням функція pthread_cond_wait() знову захоплює м’ютекс M); звільняє м’ютекс M; продовжує виконання.

2) Потік **B** (сигналізує): виконує обчислення до моменту, коли внаслідок його “дій” змінюється умова; захоплює м’ютекс M; виконує дії, що призводять до зміни умови; перевіряє умову, і якщо вона виконується, то сигналізує про це потоку **A** (pthread_cond_signal()); звільняє м’ютекс M; продовжує виконання.

| **Thread A** | | **Thread B** |
|---|---|---|
| • Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)<br>• Lock associated mutex and check value of a global variable<br>• Call pthread_cond_wait() to perform a blocking wait for signal from Thread-B. Note that a call to pthread_cond_wait() automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.<br>• When signalled, wake up. Mutex is automatically and atomically locked.<br>• Explicitly unlock mutex<br>• Continue | | • Do work<br>• Lock associated mutex<br>• Change the value of the global variable that Thread-A is waiting upon.<br>• Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A.<br>• Unlock mutex.<br>• Continue |

Рис.3.28. Типова послідовність дій по використанню умовної змінної у двох потоках.

Умовну змінну можна використати, наприклад, для узгодження виконання двох потоків, один з яких (потік А) записує дані у спільну чергу, а інший (потік В) читає дані з цієї черги (рис.3.29). В цьому прикладі потік А сигналізує потоку В про те, що в черзі з'явилися дані, які можна прочитати. У потоці В перед переходом до очікування на сигнал від потоку А виконується перевірка (не)виконання умови (while(queue is empty)), щоб уникнути нескінченного блокування на cond_wait(C,M), оскільки, якщо чекати на умову, яка вже виконалась, то є всі шанси, що про її виконання ніхто ніколи не просигналізує.

*Thread A*

1. Read data [B]
2. Lock *M* [b]
3. Put item on queue
4. cond_signal(*C*)
5. Unlock *M*
6. Goto step 1

*Thread B*

1. Lock *M* [b]
2. while (queue is empty) {
   cond_wait(*C*, *M*) [B]
   }
3. Remove item; update database
4. Unlock *M*
5. Goto step 1

Рис.3.29. Приклад узгодження виконання двох потоків, які записують та читають дані зі спільної черги, за допомогою умовної змінної С.

При використанні умовних змінних потрібно зважати на можливість помилкового сигналізування про виконання умови (spurious wakeup). В цьому випадку потік, який дочекався сигналу, розблоковується, але перед його подальшим виконанням варто додатково перевірити виконання умови. У наведеному прикладі (рис.3.29) така перевірка реалізована у вигляді циклу while (while(queue is empty)). Якщо умова все ж таки не виконана, то потік знову переходить до очікування (cond_wait(C,M)). Причиною помилкового сигналізування (spurious wakeup) може бути те, що за час передавання сигналу якийсь інший потік встиг виконати дії, які змінили умову, про виконання якої було просигналізовано.

В POSIX Threads (Pthreads) об'єкт синхронізації "умовна змінна" створюється у вигляді змінної типу pthread_cond_t. Після оголошення змінної, умовну змінну потрібно ініціалізувати: або 1) присвоївши їй значення константи PTHREAD_COND_INITIALIZER, якщо це статична змінна, або 2) викликавши функцію pthread_cond_init() (рис.3.30), якщо умовна змінна зберігається в області пам'яті, що виділяється динамічно. В останньому випадку перед завершенням програми потрібно звільнити відповідний системний ресурс, викликавши функцію pthread_cond_destroy() (рис.3.30). Як об'єкт даних, умовна змінна може бути розташована або у локальній пам'яті процесу, або у ділянці спільної пам'яті (shared memory) двох чи більше процесів. Разом з умовною змінною створюється відповідний м'ютекс, який необхідний для роботи з нею (рис.3.31).

```
#include <pthread.h>

int pthread_cond_init (pthread_cond_t *cond, pthread_condattr_t *attr);

int pthread_cond_destroy (pthread_cond_t *cond);
```

Рис.3.30. Опис функцій pthread_mutex_init() та pthread_mutex_destroy().

```
int     count = 0;
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;
```

```
int main(int argc, char *argv[])
{
.  .  .

  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init(&count_threshold_cv, NULL);

.  .  .

  /* Clean up and exit */
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit (NULL);
}
```

Рис.3.31. Приклад створення, ініціалізації та знищення умовної змінної.

Для очікування сигналу про виконання умови використовується функція pthread_cond_wait() (рис.3.32). Першим параметром вказується умовна змінна cond, а другим параметром (mutex) вказується відповідний м'ютекс. Для очікування сигналу про виконання умови на заданому проміжку часу (timeout) використовується функція pthread_cond_timedwait() (рис.3.32). Функція pthread_cond_wait() блокує викликаючий потік доти, доки якийсь інший потік не просигналізує про виконання умови за допомогою функції pthread_cond_signal(). Функція pthread_cond_wait() викликається лише після захоплення м'ютекса mutex. Сама функція автоматично і атомарно звільняє цей м'ютекс на час очікування. Після надходження сигналу про виконання умови, потік розблоковується і при виході з функції pthread_cond_wait() mutex автоматично захоплюється знову. Тому після розблокування потоку потрібно звільнити цей м'ютекс викликом pthread_mutex_unlock() (рис.3.33).

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

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

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

Рис.3.32. Опис функцій pthread_cond_wait() та pthread_cond_timedwait().

```
  /*
  Lock mutex and wait for signal.  Note that the pthread_cond_wait routine
  will automatically and atomically unlock mutex while it waits.
  Also, note that if COUNT_LIMIT is reached before this routine is run by
  the waiting thread, the loop will be skipped to prevent pthread_cond_wait
  from never returning.
  */
  pthread_mutex_lock(&count_mutex);
  while (count < COUNT_LIMIT) {
    printf("watch_count(): thread %ld Count= %d. Going into wait...\n",
my_id,count);
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count(): thread %ld Condition signal received.
Count= %d\n", my_id,count);
    printf("watch_count(): thread %ld Updating the value of count...\n",
my_id,count);
    count += 125;
```

```
    printf("watch_count(): thread %ld count now = %d.\n", my_id, count);
    }
  printf("watch_count(): thread %ld Unlocking mutex.\n", my_id);
  pthread_mutex_unlock(&count_mutex);
```

Рис.3.33. Приклад використання функції pthread_cond_wait()
для очікуваня сигналу про виконання умови.

Для надсилання сигналу про виконання умови використовується функція pthread_cond_signal() (рис.3.34), параметром якої вказується відповідна умовна змінна cond. Виклик функції pthread_cond_signal() розблокує один потік, що очікує на сигнал (тобто заблокований викликом pthread_cond_wait()). Якщо надходження цього сигналу також очікують інші потоки, то вони залишаться заблокованими. Для надсилання сигналу одночасно усім потокам, які чекають на виконання відповідної умови, використовується функція pthread_cond_broadcast() (рис.3.34). Перед викликом pthread_cond_signal() також потрібно спочатку захопити, а після завершення виклику звільнити м'ютекс, прив'язаний до умовної змінної (рис.3.35).

```
#include <pthread.h>

int pthread_cond_signal (pthread_cond_t *cond);

int pthread_cond_broadcast (pthread_cond_t *cond);
```

Рис.3.34. Опис функцій pthread_cond_signal() та pthread_cond_broadcast().

```
    pthread_mutex_lock(&count_mutex);
    count++;
    /*
    Check the value of count and signal waiting thread when condition is
    reached. Note that this occurs while mutex is locked.
    */
    if (count == COUNT_LIMIT) {
      printf("inc_count(): thread %ld, count = %d  Threshold reached. ",
             my_id, count);
      pthread_cond_signal(&count_threshold_cv);
      printf("Just sent signal.\n");
      }
    printf("inc_count(): thread %ld, count = %d, unlocking mutex\n",
         my_id, count);
    pthread_mutex_unlock(&count_mutex);
```

Рис.3.35. Приклад використання функції pthread_cond_signal() для
надсилання сигналу про виконання умови.

POSIX Threads (Pthreads) надає можливість створювати умовну змінну із заданими атрибутами. Для цього використовується відповідний атрибутний об'єкт - змінна типу pthread_condattr_t. Перед використанням, атрибутний об'єкт потрібно ініціалізувати за допомогою функції pthread_condattr_init() (рис.3.36). Після завершення роботи з атрибутним об'єктом потрібно звільнити відповідний ресурс викликом pthread_condattr_destroy() (рис.3.36). Для отримання та встановлення значень атрибутів використовуються функції, наведені на рис.3.36. Типова послідовність кроків по використанню атрибутного об'єкту: створити атрибутний об'єкт, встановити потрібні значення атрибутів, передати атрибутний об'єкт другим параметром функції pthread_cond_init() для створення умовної змінної з відповідними атрибутами. Атрибут pshared у випадку умовної змінної має те саме значення, що відповідний атрибут м'ютекса.

```
#include<pthread.h>
int pthread_condattr_init (pthread_condattr_t *attr);
int pthread_condattr_destroy (pthread_condattr_t *attr);

int pthread_condattr_getpshared (const pthread_condattr_t *attr, int *pshared);
int pthread_condattr_getclock (const pthread_condattr_t *attr, clockid_t *clock_id);

int pthread_condattr_setpshared (pthread_condattr_t *attr, int pshared);
int pthread_condattr_setclock (pthread_condattr_t *attr, clockid_t clock_id);
```

Рис.3.36. Опис функцій для роботи з атрибутами умовної змінної.

### 3.3.3. Блокування читання-запису (readers–writer lock)

Блокування читання-запису (readers–writer lock) - це механізм синхронізації, який дозволяє одночасне читання та ексклюзивний запис захищених спільних даних. Блокування читання-запису реалізовано у вигляді відповідного об'єкту синхронізації rwlock, який можна захопити або в режимі читання (read lock), або в режимі запису (write lock). Щоб змінити спільні дані, потік повинен спочатку отримати ексклюзивний доступ для запису (захопити rwlock в режимі запису). Ексклюзивне захоплення для запису не дозволяється, поки не будуть звільнені всі захоплення даного rwlock в режимі читання.

Блокування читання-запису (readers–writer lock) використовується у випадках, коли на відміну від використання м'ютексів, є можливість розмежувати операції читання та запису (за умов що дозволено одночасне читання захищених даних декількома потоками). За рахунок цього досягається прискорення паралельного виконання (одночасні операції читання не блокуються), тобто досягається вищий ступінь паралелізму. В деяких паралельних програмах дані зчитуються частіше, ніж змінюються, тому такі програми виграють у швидкості виконання, якщо в них використовуються блокування читання-запису замість м'ютексів.

На відміну від м'ютекса, який може мати два стани (відкритий, закритий), об'єкт блокування читання-запису rwlock може мати три стани: блокування читання (закритий для операцій читання), блокування запису (закритий для операцій запису) і відсутність блокування (відкритий). При цьому між захопленням rwlock на читання чи запис є різниця. Зокрема тут діють наступні правила:

1) Будь-яка кількість потоків може заблокувати ресурс для зчитування (read lock), якщо жоден інший потік не заблокував його для запису.

2) Блокування ресурсу для запису (write lock) може бути встановлено, лише коли жоден інший потік не заблокував ресурс для читання або для запису.

Іншими словами, довільна кількість потоків може зчитувати спільні дані, якщо жоден потік не змінює їх у поточний момент. Спільні дані можуть бути змінені лише за умови, що ніхто інший їх не зчитує й не змінює.

Якщо блокування читання-запису встановлено в режимі для запису, всі потоки, які будуть намагатися також встановити блокування для запису, будуть призупинені доти, поки блокування не буде знято. Якщо блокування читання-запису встановлено в режимі для читання, всі потоки, які будуть намагатися також встановити блокування для читання, отримають доступ до ресурсу, але якщо який-небудь потік спробує встановити блокування для запису, він буде призупинений доти, поки не буде знято останнє блокування для читання.

В POSIX Threads (Pthreads) об'єкт синхронізації rwlock створюється у вигляді змінної типу pthread_rwlock_t. Після оголошення змінної, rwlock потрібно ініціалізувати одним з двох способів:

1) присвоївши змінній значення константи PTHREAD_RWLOCK_INITIALIZER, якщо це статична змінна, або

2) викликавши функцію pthread_rwlock_init() (рис.3.37), якщо rwlock зберігається в області пам'яті, що виділяється динамічно.

В останньому випадку перед завершенням програми потрібно звільнити відповідний системний ресурс, викликавши функцію pthread_rwlock_destroy() (рис.3.37). Після ініціалізації rwlock знаходиться в стані “відкритий” (незахоплений). Як об’єкт даних, rwlock може бути розташований або у локальній пам’яті процесу, або у ділянці спільної пам’яті (shared memory) двох чи більше процесів.

```
#include <pthread.h>

int pthread_rwlock_init (pthread_rwlock_t *rwlock,
                         const pthread_rwlockattr_t *attr);

int pthread_rwlock_destroy (pthread_rwlock_t *rwlock);
```

Рис.3.37. Опис функцій pthread_rwlock_init() та pthread_rwlock_destroy().

Для здійснення операцій read lock, write lock та unlock над rwlock використовуються функції pthread_rwlock_rdlock(), pthread_rwlock_wrlock() та pthread_rwlock_unlock() (рис.3.38). В якості параметру вказується rwlock, над яким здійснюється операція. За допомогою функцій pthread_rwlock_tryrdlock() та pthread_rwlock_trywrlock() (рис.3.38) можна виконати спробу захоплення rwlock в режимі без блокування (в асинхронному режимі). Якщо на момент виклику цих функцій rwlock вже захоплений іншим потоком (у відповідному режимі), то функція завершиться негайно і поверне код помилки EBUSY, інакше функція захопить rwlock і поверне 0. За допомогою функцій pthread_rwlock_timedrdlock() та pthread_rwlock_timedwrlock() (рис.3.38) можна виконати спробу захоплення rwlock на протязі заданого проміжку часу (abs_timeout).

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

int pthread_rwlock_rdlock (pthread_rwlock_t *rwlock);

int pthread_rwlock_wrlock (pthread_rwlock_t *rwlock);

int pthread_rwlock_tryrdlock (pthread_rwlock_t *rwlock);

int pthread_rwlock_trywrlock (pthread_rwlock_t *rwlock);

int pthread_rwlock_timedrdlock (pthread_rwlock_t *rwlock,
                                const struct timespec *abs_timeout);

int pthread_rwlock_timedwrlock (pthread_rwlock_t *rwlock,
                                const struct timespec *abs_timeout);

int pthread_rwlock_unlock (pthread_rwlock_t *rwlock);
```

Рис.3.38. Опис функцій для здійснення операцій над rwlock.

POSIX Threads (Pthreads) надає можливість створювати rwlock із заданими атрибутами. Для цього використовується відповідний атрибутний об’єкт – змінна типу pthread_rwlockattr_t. Перед використанням, атрибутний об’єкт потрібно ініціалізувати за допомогою функції pthread_rwlockattr_init() (рис.3.39). Після завершення роботи з атрибутним об’єктом потрібно звільнити відповідний ресурс викликом pthread_rwlockattr_destroy() (рис.3.39). Для отримання та встановлення значень атрибутів використовуються функції, наведені на рис.3.39. Типова послідовність кроків по використанню атрибутного об’єкту: створити атрибутний об’єкт, встановити потрібні значення атрибутів, передати атрибутний об’єкт другим параметром функції pthread_rwlock_init() для створення rwlock з відповідними атрибутами. Атрибут pshared у випадку rwlock має те саме значення, що відповідний атрибут м’ютекса.

```
#include<pthread.h>
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);

int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *attr, int *pshared);
int pthread_rwlockattr_getkind_np(const pthread_rwlockattr_t *attr, int *pref);

int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);
int pthread_rwlockattr_setkind_np(pthread_rwlockattr_t *attr, int pref);
```

Рис.3.39. Опис функцій для роботи з атрибутами rwlock.

### 3.3.4. Спін-блокування (spinlock)

Спін-блокування (spinlock) використовуєтья в тих самих ситуаціях, де потрібен м’ютекс, але блокування (призупинка) потоку на виклику операції lock неприпустиме. Наприклад, обробник переривання у ядрі ОС ніколи не повинен блокуватись. Якщо це станеться, система зависне (вийде з ладу). Якщо в обробнику переривання потрібно вставити елемент даних у загальнодоступний зв'язаний список, то для цього захоплюється відповідний спінлок, вставляється елемент даних, звільняється спінлок.

Спінлок (spinlock) - це механізм синхронізації низького рівня, який, головним чином, використовується у мультипроцесорних системах зі спільною пам'яттю (shared memory multiprocessors). Коли викликаючий потік намагається захопити спінлок, який вже захоплений іншим потоком, він не призупиняється, а виконує спеціальний цикл, в якому весь час перевіряє, чи спінлок не звільнився. Коли спінлок захоплено, його слід “утримувати” лише короткий час, оскільки виконання потоків, що в циклі очікують на звільнення спінлоку, витрачає процесорний час. Перед тим, як потік з якоїсь причини призупиняється, він повинен звільнити усі захоплені спінлоки, щоб надати іншим потокам можливість їх захопити.

При виконанні будь-якого блокування виникає дилема між 1) витратою ресурсів процесора для налаштування блокування потоку на м’ютексі та 2) витратою ресурсів процесора на виконання потоку в циклі, коли він заблокований на спінлоку. Спінлоки вимагають небагато ресурсів для налаштування, після чого виконується простий цикл, в якому повторюється атомарна операція захоплення, поки її виконання не стане можливим. Тобто під час очікування потік продовжує споживати ресурси процесора. В цілому спінлоки витрачають менше системних ресурсів в разі короткочасного блокування потоків, а м’ютекси витрачають менше системних ресурсів в разі порівняно тривалого блокування потоків.

В POSIX Threads (Pthreads) об’єкт синхронізації “спінлок” створюється у вигляді змінної типу pthread_spinlock_t. Після оголошення змінної, спінлок потрібно ініціалізувати викликом функції pthread_spin_init() (рис.3.40). Другим параметром pshared задається значення відповідного атрибуту спінлоку (process_shared). Атрибут pshared у випадку спінлоку має те саме значення, що і відповідний атрибут м’ютекса. Після завершення роботи зі спінлоком потрібно звільнити відповідний системний ресурс, викликавши функцію pthread_spin_destroy() (рис.3.40).

```
#include<pthread.h>

int pthread_spin_init (pthread_spinlock_t *lock, int pshared);
int pthread_spin_destroy (pthread_spinlock_t *lock);
```

Рис.3.40. Опис функцій pthread_spin_init() та pthread_spin_destroy().

Для здійснення операцій lock та unlock над спінлоком використовуються функції pthread_spin_lock() та pthread_spin_unlock() (рис.3.41). В якості параметру вказується спінлок, над яким здійснюється операція. За допомогою функції pthread_spin_trylock() (рис.3.41) можна

виконати спробу захоплення спінлока в режимі без блокування (в асинхронному режимі). Якщо на момент виклику pthread_spin_trylock() спінлок вже захоплений іншим потоком, то функція завершиться негайно і поверне код помилки EBUSY, інакше функція захопить спінлок і поверне 0.

```
#include <pthread.h>

int pthread_spin_lock (pthread_spinlock_t *lock);

int pthread_spin_trylock (pthread_spinlock_t *lock);

int pthread_spin_unlock (pthread_spinlock_t *lock);
```

Рис.3.41. Опис функцій pthread_spin_lock(), pthread_spin_trylock() та pthread_spin_unlock().

### 3.3.5. Бар’єрна синхронізація програмних потоків

Бар’єрна синхронізація використовується у випадках, коли потрібно дочекатися завершення виконання обчислень у декількох програмних потоках, перш ніж продовжити виконання програми одночасно у всіх цих потоках (рис.3.42). У стандарті POSIX Threads (Pthreads) визначено відповідний об’єкт синхронізації barrier та функції з префіксом pthread_barrier для роботи з ним. Ці функції дозволяють створити бар'єр, вказавши кількість потоків, які мають зібратися на бар'єрі, і реалізувати у потоках очікування моменту часу, коли зрештою всі потоки “підійдуть до бар’єру”. Коли останній потік “підходить до бар'єру”, всі потоки відновлюють виконання.

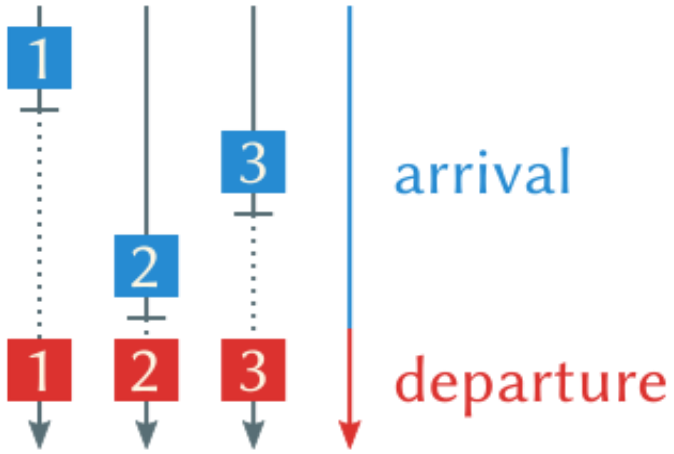


Рис.3.42. Схема бар’єрної синхронізації трьох потоків.

В POSIX Threads (Pthreads) об’єкт синхронізації barrier створюється у вигляді змінної типу pthread_barrier_t. Після оголошення змінної, бар’єр потрібно ініціалізувати викликом функції pthread_barrier_init() (рис.3.43). Третім параметром count задається кількість потоків, які будуть збиратися у бар’єра. Для бар’єрної синхронізації потоку потрібно викликати функцію pthread_barrier_wait() (рис.3.44). Коли count потоків зрештою викличуть функцію pthread_barrier_wait() для заданого бар’єру, то усі вони одночасно розблокуються і продовжать своє виконання. Після завершення роботи з бар’єром потрібно звільнити відповідний системний ресурс, викликавши функцію pthread_barrier_destroy() (рис.3.43).

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

```
int pthread_barrier_init (pthread_barrier_t *restrict barrier,
                          const pthread_barrierattr_t *restrict attr,
                          unsigned count);

int pthread_barrier_destroy (pthread_barrier_t *barrier);
```

Рис.3.43. Опис функцій pthread_mutex_init() та pthread_mutex_destroy().

```
#include <pthread.h>

int pthread_barrier_wait (pthread_barrier_t  *barrier);
```

Рис.3.44. Опис функції pthread_barrier_wait().

POSIX Threads (Pthreads) надає можливість створювати бар'єри із заданим атрибутом pshared (process_shared). Для цього використовується відповідний атрибутний об'єкт - змінна типу pthread_barrierattr_t. Перед використанням, атрибутний об'єкт потрібно ініціалізувати за допомогою функції pthread_barrierattr_init() (рис.3.45). Після завершення роботи з атрибутним об'єктом потрібно звільнити відповідний ресурс викликом pthread_barrierattr_destroy() (рис.3.45). Для отримання та встановлення значення атрибуту pshared використовуються функції pthread_barrierattr_getpshared() та pthread_barrierattr_setpshared() (рис.3.45). Типова послідовність кроків по використанню атрибутного об'єкту: створити атрибутний об'єкт, встановити потрібне значення атрибуту pshared, передати атрибутний об'єкт другим параметром функції pthread_barrier_init() для створення бар'єру з відповідним атрибутом. Атрибут pshared у випадку бар'єра має те саме значення, що відповідний атрибут м'ютекса.

```
#include<pthread.h>

int pthread_barrierattr_init(pthread_barrierattr_t *attr);
int pthread_barrierattr_getpshared(const pthread_barrierattr_t *attr,
                                   int *pshared);
int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr, int pshared);
int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
```

Рис.3.45. Опис функцій для роботи з атрибутним об'єктом бар'єру.

## Контрольні питання

1. Які основні переваги розпаралелювання обчислень на рівні програмних потоків у порівнянні з розпаралелюванням на рівні обчислювальних процесів?
2. Для чого призначений інтерфейс прикладного програмування POSIX Threads?
3. Яка модель відображення потоків рівня користувача у потоки рівня ядра реалізована в Native POSIX Thread Library (NPTL)?
4. Яку функцію можна вважати потоко-безпечною (thread-safe)?
5. Яка функція призначена для створення нового програмного потоку в POSIX Threads і що вказується у її параметрах?
6. Яка функція POSIX Threads надсилає вказаному програмному потоку запит на завершення?
7. Як в POSIX Threads організовано очікування на завершення програмного потоку?
8. За допомогою якої функції POSIX Threads вказаний потік робиться від'єднаним, тобто таким на завершення якого не може очікувати жоден інший потік?
9. Які функції POSIX Threads використовуються для управління атрибутами потоків?
10. Які основні атрибути має програмний потік в POSIX Threads?
11. Які функції POSIX Threads використовуються для управління м'ютексами?
12. Який тип м'ютекса в POSIX Threads передбачає перевірку наявності помилок або ситуацій взаємного блокування (deadlocks) при звертанні до м'ютекса?
13. Який тип м'ютекса в POSIX Threads підтримує підрахунок кількості операцій захоплення (locks) та звільнення (unlocks) м'ютекса?
14. Для чого використовуються умовна змінна (conditional variable)?
15. Як організовано синхронізацію програмних потоків за допомогою умовної змінної?
16. Які функції POSIX Threads використовуються для управління умовними змінними?
17. Яка функція POSIX Threads призначена для очікування сигналу про виконання умови при використанні умовної змінної?
18. В яких станах може знаходитись об'єкт синхронізації "блокування читання-запису" (readers–writer lock)?
19. Яка функція POSIX Threads використовуються для здійснення операції «блокування запису» над об'єктом синхронізації readers–writer lock (rwlock)?
20. Для чого використовується засіб синхронізації спінлок (spinlock) і чим він відрізняється від м'ютекса?
21. Які функції POSIX Threads використовуються для бар'єрної синхронізації програмних потоків?