МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»

БАГАТОПОТОЧНІСТЬ В ОПЕРАЦІЙНІЙ СИСТЕМІ LINUX. СТВОРЕННЯ, КЕРУВАННЯ ТА СИНХРОНІЗАЦІЯ ПОТОКІВ.

МЕТОДИЧНІ ВКАЗІВКИ

до виконання лабораторної роботи №6
з дисципліни «Операційні системи»
для студентів першого (бакалаврського) рівня вищої освіти
спеціальності 121 «Інженерія програмного забезпечення»

Затверджено
на засіданні кафедри
програмного забезпечення.
Протокол №__від __.__.2022 р.

Львів – 2022

Багатопоточність в операційній системі Linux. Створення, керування та синхронізація потоків.: Методичні вказівки до лабораторної роботи №1 з дисципліни «Операційні системи» для студентів першого (бакалаврського) рівня вищої освіти спеціальності 121 – Інженерія програмного забезпечення / Укл.: О.Д.Грицай, –Львів: 2022. – 25 с.

Укладачі: Грицай О.Д., канд. фіз..-мат. наук,

Відповідальний за випуск Федасюк Д.В., д-р. техн. наук, проф.

Рецензенти

Лабораторна робота №6

Тема. Багатопоточність в операційній системі Linux. Створення, керування та синхронізація потоків.

Мета. Навчитися створювати потоки та керувати ними в операційній системі Linux. Ознайомитися з методами синхронізації потоків в операційній системі Linux. Навчитися реалізовувати багатопоточний алгоритм розв'язку задачі з використанням синхронізації в операційній системі Linux.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Потоки в операційній системі Linux. Поняття потоку, як одиниці виконання процесу в операційній системі (ОС) Linux, аналогічне як і у ОС Windows. Кожен процес можна представити як контейнер ресурсів і послідовність інструкцій виконання. Таким чином, можна сказати, що кожен процес містить хоча б один потік виконання, якому надані лічильник інструкцій, регістри і стек. Крім того, у кожному процесі можна створити додаткові гілки виконання – потоки, тоді такий процес називають багатопоточним.

Різниця між потоками в ОС Linux і в ОС Windows полягає у їх представленні в ядрі операційних систем. В ОС Windows потоки виконання у режимі користувача зіставляються з потоками у режим ядра. У перших версіях ядра Linux потоки користувача зіставлялись з процесами у ядрі. Створення потоку відбувалось з допомогою системного виклику clone(). Виклик clone(), як і fork() дозволяє створювати новий процес, але з певними відмінностями :

- одразу повне копіювання батьківського процесу
- створення власного стеку
- необхідно вказати спеціальний набір прапорців успадкування для того, щоб визначити, як будуть розподілятися ресурси (адресний простір, відкритті файли, обробники сигналів) між батьківським і дочірнім процесом.

Таким чином, створювався новий потік у режимі користувача, який відображається у процес ядра. Відображення здійснюється за моделлю 1:1. Оскільки керуючий блок процесу в Linux представлений в ядрі структурою *task_struct,* то і представлення потоку здійснювалось через *task_struct.* Фактично у ядрі потоки і процеси не розрізнялися. Але системний виклик clone() не підтримувався стандартом POSIX і тому розроблялись бібліотеки потоків, що дозволяли працювати з потоками, використовуючи clone() з різними атрибутами виконання.

У сучасних версіях Linux підтримуються спеціальні об'єкти ядра - потоки ядра, побудованні на зміненному і розширеному системному виклику clone(). Підтримка потоків здійснюється через POSIX-сумісну бібліотеку NPTL (Native POSIX Threads Library). Типи даних і функції, що застосовуються до потоків POSIX, мають префікс pthread_ і доступні через підключення phread.h.

Для компіляції програми, що використовує потоки необхідно вказати підключення бібліотеки pthread (Приклад 1).

Приклад 1. Запис командного рядка компіляції програми з потоками

```
user:~$ gcc -o main main.c -Wall -pthread
user:~$ ./main
```

Створення потоків.

Для створення потоку використовують функцію **phread_create()**:

```
int pthread_create(pthead_t *thread, pthrad_attr_t *attr,
void*(*thread_func)(void *), void *arg);
```

thread - вказівник на структуру `pthead_t`, дескриптор потоку;
attr - вказівник на структуру атрибутів (розмір стеку, пріоритет, ...)
thread_func - вказівник на функцію потоку
arg- дані, що передаються у функцію.

Функція потоку має особливу сигнатуру `void*(*thread_func)(void *)`, що дозволяє передавати в неї аргументи будь-якого типу і повертати результат також будь-якого типу.

Приклад 2. Створення потоку і передача в нього декількох параметрів:

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

typedef struct {
     int age;
     char *name;
} th_param; //структура передачі параметрів

void *th_func(void *args) {
printf("age=%d  for  name  %s",  (int)((th_param*)args)->age,
(char *)((th_param*)args)->name );
return NULL;
}

int main(){
th_param param;
pthread_t th;
param.age=20;
param.name="Anna";
pthread_create(&th, NULL, &th_func, &param);
return 0;
}
```

Зауваження. Інколи доцільно наступне приведення до типу:

- Якщо функція вимагає int аргументу

  int pthread_create(..., void(*start_routine)(int), int arg);*

- Якщо функція повертає int

  *int pthread_create(..., int(*start_routine)(void*), void * arg);*

Очікування завершення потоків.

У попередньому прикладі важко побачити результат, оскільки батьківський потік виконається швидше і завершиться весь процес. Аналогічно, як у Windows потрібно дочекатися завершення дочірнього потоку. Це можна зробити за допомогою функції **pthread_join.**

`int` **`pthread_join`**`(pthread_t thread, void **value_ptr);`
thread – дескриптор потоку;
value_ptr – вказівник на повернений через pthread_exit або return результат виконання потоку; може бути NULL – тоді не очікуємо результат.
`void` **`pthread_exit`**`(void * retval)` функція, що дозволяє завершити виконання потоку з функції виконання потоку де retval - код повернення. Виклик цієї функції можливий лише для приєднаних потоків (joinable). За замовчуванням всі новостворені потоки приєднанні.

Приклад 3. Приклад очікування результату виконання потоку

```
#include<pthread.h>
void *thread_func (void * id_th){
   printf("i am %d thread\n", int(id_th));
pthread_exit (0);
}
void main(){
pthread_t thread;
int i=2;
void * value_ptr;
pthread_create(&thread, NULL, thread_func, (void *) i);
pthread_join(thread, &value_ptr)
}
```

Якщо нема необхідності очікувати повернення результату від потоку, то потік можна позначити як відокремлений. Тоді система отримає вказівку звільнити виділенні ресурси після завершення потоку. Зробити це можна з допомогою функції **pthread_detach.**

`int` **`pthread_detach`**`(pthread_t thread);` - функція позначає потік, ідентифікований потоком, як відокремлений. Коли відокремлений потік завершується, його ресурси автоматично звільняються назад до системи без необхідності приєднання іншого потоку до завершеного потоку.

Задати, що потік відокремлений можна задавши параметри структури атрибутів потоку.

Структура атрибутів потоку.

Параметри потоку можна змінити через структуру атрибутів потоку.
pthread_attr_t attr;
Ініціалізувати структуру можна з допомогою функції
pthread_attr_init(&attr);
Призначення атрибутів відбувається у наступному порядку:

1) Створити структуру атрибутів
2) Ініціалізувати її стандартними значеннями
3) Записати необхідні значення атрибутів (функції наведенні нище)
4) Передати вказівник на структуру при створенні потоків
5) Викликати pthread_attr_destroy() – для видалення об'єкту структури атрибутів з пам'яті, але сам об'єкт не видаляється, його можна проініціалізувати знову.

Для задання атрибутів існують різні функції.
Серед них:

- *вказати, що потік буде від'єднаним*:

int pthread_attr_setdetacstate(pthread_attr_t *attr, int detachstate);

detachstate - може мати два значення PTHREAD_CREATE_JOINABLE (приєднаний) або PTHREAD_CREATE_DETACHED (від'єднаний).

int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate); - повертає атрибут стану від'єднання атрибутів потоку у буфері, на який вказує detachstate.

- *встановити параметри планування потоків*

int **pthread_attr_setschedparam**(pthread_attr_t *attr, const struct sched_param *param); – встановлює параметри планування атрибутів в об'єкті атрибутів потоку на який посилається attr, до значень, зазначених у буфері, на який вказує param

int **pthread_attr_getschedparam**(const pthread_attr_t *attr,struct sched_param *param); – повертає параметри планування атрибутів в об'єкті атрибутів потоку

Параметри планування зберігаються в такій структурі:

```
struct sched_param {
int sched_priority; / * Пріоритет планування */
};
```

int **pthread_attr_setschedpolicy**(pthread_attr_t *attr, int policy); – функція встановлює атрибут policy планування об'єкта атрибутів потоку, на який посилається attr, до значення, зазначеного в policy. Підтримувані значення для policy: SCHED_FIFO, SCHED_RR та SCHED_OTHER.

int **pthread_attr_getschedpolicy**(const pthread_attr_t *attr, int *policy);– повертає значення атрибуту policy

- *встановити параметри спорідненості з процесором*

int **pthread_attr_setaffinity_np**(pthread_attr_t *attr, size_t cpusetsize, const cpu_set_t *cpuset); – встановлює спорідненість з процесором

int **pthread_attr_getaffinity_np**(const pthread_attr_t *attr, size_t cpusetsize, cpu_set_t *cpuset);– повертає спорідненість з процесором

#include <sched.h>

int **sched_yield**(void); - викликає відмову потоку виклику від процесора. Потік зепереміщується до кінця черги готовності відносно статичного пріоритету, і запускається новий потік.

int **pthread_attr_setinheritsched**(pthread_attr_t *attr, int inheritsched);

int **pthread_attr_getinheritsched**(const pthread_attr_t *attr, int *inheritsched);

Функція pthread_attr_setinheritsched () встановлює атрибут inheritsched об'єкта атрибутів потоку, на який посилає attr, значення, зазначене в inheritsched.

Атрибут inheritsched визначає, чи потік, створений за допомогою об'єкта атрибутів потоку, буде успадковувати свої атрибути планування від потоку, що викликає, або візьме їх з атрибута atr. Наступні значення можуть бути вказані у inheritsched: PTHREAD_INHERIT_SCHED - потоки, створені за допомогою attr, успадковують атрибути планування з потоку створення; атрибути планування у attr ігноруються. PTHREAD_EXPLICIT_SCHED - потоки, створені за допомогою attr, мають свої атрибути планування зі значень, визначених об'єктом attributes.

int **pthread_attr_setscope**(pthread_attr_t *attr, int scope);
int **pthread_attr_getscope**(const pthread_attr_t *attr, int *scope);
встановити/отримати атрибут сфери спору в об'єкті атрибутів потоку. Атрибут scope визначає набір потоків, проти яких потік конкурує за такі ресурси, як центральний процесор.PTHREAD_SCOPE_SYSTEM потік конкурує за ресурси з усіма іншими потоками-процеси в системі, PTHREAD_SCOPE_PROCESS потік змагається за ресурси з усіма іншими потоками в тому самому процесі, в якому також був створений.

int **pthread_attr_setstacksize**(pthread_attr_t *attr, size_t stacksize); функція встановлює атрибут розміру стека об'єкта атрибутів потоку, на який посилається attr, до значення, зазначеного в stacksize. Атрибут розміру стека визначає мінімальний розмір (у байтах), який буде виділено для потоків, створених за допомогою об'єкта атрибутів потоків.
int **pthread_attr_getstacksize**(const pthread_attr_t *attr, size_t *stacksize);
Відміна виконання потоків.
Потік завершується коли виходить з функції потоку і викликає pthread_exit(). Якщо є необхідність відминити виконання потоку з іншого потоку викликають функцію **pthread_cancel.**
`int pthread_cancel(pthread_t thread);` – надсилає запит на скасування до потоку. Чи і коли цільовий потік реагує на запит на скасування, залежить від двох атрибутів, які перебувають під контролем цього потоку: його стану та типу скасування. Стан скасування потоку, визначений pthread_setcancelstate можна ввімкнути (за замовчуванням для нових потоків) або вимкнути.
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
Якщо потік вимкнув скасування, тоді залишається запит на скасування в черзі, поки потік не дозволить скасувати.
Тип скасування потоку, визначений pthread_setcanceltype. може бути або asynchronous асинхронним, або or deferred синхроним (відкладеним) (за умовчанням для нових потоків). Асинхронне скасування означає, що потік можна скасувати в будь -який час (зазвичай негайно, але система цього не гарантує). Синхроне скасування означає, що скасування буде затримуватися, поки наступний потік не викликає функцію, яка є точкою скасування.
Точка скасування створюється за допомогою функції pthread_testcancel(). Її порібно періодично викликати в місці, де потік можна завершити без втрати ресурсів.

int **pthread_kill**(pthread_t thread, int sig); -функція надсилає сигнал sig до потоку в тому ж процесі. Сигнал асинхронно направляється до потоку.

pthread_t **pthread_self**(void); повертає ідентифікатор потоку виклику.

Синхронізація потоків.

Потоки одного процесу можуть виконуватися як послідовно, так і паралельно. Потік є незалежним, якщо він не впливає на виконання інших потоків, інші потоки не впливають на нього і не мають з ним спільних даних. Результат виконання такого потоку однозначно залежить від вхідних даних. Всі інші потоки – взаємодіючі. Для отримання коректного результату необхідна синхронізація потоків, що взаємодіють між собою.

М'ютекс

М'ютекс дозволяє потокам керувати доступом до даних. При використанні м'ютексу тільки один потік в певний момент часу може заблокувати м'ютекс і отримати доступ до ресурсу («ліцензію» на його використання). При завершенні роботи з ресурсом потік повинен повернути «ліцензію», розблокувавши м'ютекс. Якщо який-небудь потік звернеться до вже заблокованого м'ютексів, то він буде змушений чекати розблокування м'ютекса потоком, що володіє їм.

Прототипи функцій для виконання операцій над м'ютексів описуються в файлі *pthread.h.* Нижче наводяться прототипи найбільш часто використовуваних функцій:

- для ініціалізації

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER; або
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // успішна ініціалізація!
```

- для захоплення і звільнення мютексу:

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

Приклад 4..

```
pthread_mutex_lock(&lock);
 x = x + 1; //
 pthread_mutex_unlock(&lock);
```

- неблокуючий мютекс:

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

Ці два виклики використовуються для захоплення мютексу. Версія trylock повертає помилку, якщо мютекс вже утримується; версія timedlock повертається після тайм -ауту або після захоплення мютексу, залежно від того, що станеться раніше. Таким чином, синхронізація з тайм -аутом нуля вироджується до випадку trylock.

Condition Variables

Умовна змінна дозволяє потокам очікувати виконання деякої умови (події), пов'язаної з даними що розділяються. Над умовними змінними визначено дві основні операції: *інформування* про настання події і *очікування* події. При виконанні операції «інформування» один з потоків, які очікують на умовній змінній, відновлює свою роботу.

Умовна змінна завжди використовується спільно з м'ютексом. Перед виконанням операції «очікування» потік повинен заблокувати м'ютекс. При виконанні операції «очікування» зазначений м'ютекс автоматично розблокується. Перед поновленням що очікує потоку виконується автоматичне блокування мьютекса, що дозволяє потоку увійти в критичну секцію, після критичної секції рекомендується розблокувати м'ютекс. При подачі сигналу іншим потокам рекомендується так само функцію «сигналізації» захистити м'ютексів.

Прототипи функцій для роботи з умовними змінними містяться в файлі *pthread. H.* Нижче наводяться прототипи функцій разом з поясненням їх синтаксису і виконуваних ними дій.

pthread_cond_init (pthread_cond_t * cond, const pthread _ condattr _ t * attr); ініціалізує умовну змінну *cond* із зазначеними атрибутами *attr* або з атрибутами за замовчуванням (при вказуванні 0 в якості *attr).*

int pthread _ cond _ destroy (pthread _ cond _ t * cond); - знищує умовну змінну *cond.*

int pthread _ cond _ signal (pthread _ cond _ t * cond); - інформування про настання події потоків, які очікують на умовній змінній *cond.*

int pthread _ cond _ broadcast (pthread _ cond _ t * cond); - інформування про настання події потоків, які очікують на умовній змінній *cond.* При цьому відновлені будуть всі очікують потоки.

int pthread _ cond _ wait (pthread _ cond _ t * cond, pthread _ mutex _ t * mutex); - очікування події на умовній змінній *cond.*

Типове використання виглядає так:

Приклад 5 а

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
Pthread_mutex_lock(&lock);
while (ready == 0)
Pthread_cond_wait(&cond, &lock);
Pthread_mutex_unlock(&lock);
```

У цьому коді, після ініціалізації відповідного мютексу та умови, потік перевіряє, чи змінна ready ще не встановлена щось крім 0. Якщо ні, потік просто викликає процедуру очікування, щоб перейти в режим сну, поки інший потік не збудить її.

Код для розбудження потоку, який буде працювати в іншому потоці, виглядає так:

Приклад 5 б

```
Pthread_mutex_lock(&lock);
 ready = 1;
 Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);
```

виклик очікування приймає мютекс як другий параметр, тоді як виклик сигналу приймає лише умову. Причина цієї різниці полягає в тому, що виклик очікування, крім того, що він переводить поток виклику в режим сну, звільняє мютекс, коли переводить зазначеного абонента в режим сну. перед тим, як повернутися після пробудження, pthread cond wait () знову набуває мютекс, тим самим гарантуючи, що в будь-який час, коли потік очікування працює між блокуванням на початку послідовності очікування та звільненням мютексу в кінці, він тримає замок.

Безпечніше розглядати пробудження як натяк на те, що щось могло змінитися, а не як на абсолютний факт

Приклад 6.

```
int done = 0;
 pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
 pthread_cond_t c = PTHREAD_COND_INITIALIZER;

 void thr_exit() {
 Pthread_mutex_lock(&m);
 done = 1;
  Pthread_cond_signal(&c);
 Pthread_mutex_unlock(&m);
 }

 void *child(void *arg) {
 printf("child\n");
 thr_exit();
 return NULL;
 }

 void thr_join() {
 Pthread_mutex_lock(&m);
 while (done == 0)
 Pthread_cond_wait(&c, &m);
 Pthread_mutex_unlock(&m);
 }

 int main(int argc, char *argv[]) {
 printf("parent: begin\n");
 pthread_t p;
 Pthread_create(&p, NULL, child, NULL);
 thr_join();
```

```
printf("parent: end\n");
return 0;
}
```

Семафор

Семафор призначений для синхронізації потоків по діям і за даними. Семафор - це захищена змінна, значення якої можна опитувати і міняти тільки за допомогою спеціальних операцій *P* і *V* та операції ініціалізації. Семафор може приймати ціле невід'ємне значення. При виконанні потоком операції *P* над семафором *S* значення семафора зменшується на 1 при *S*> 0 або потік блокується, «чекаючи на семафорі», при *S* = 0. При виконанні операції *V (S)* відбувається пробудження одного з потоків, які очікують на семафорі *S,* а якщо таких немає, то значення семафора збільшується на 1 Як випливає з вищесказаного, при вході в критичну секцію потік повинен виконувати операцію *P,* а при виході з критичної секції операцію *V.*

Прототипи функцій для маніпуляції з семафора описуються в файлі *semaphore. H.* Нижче наводяться прототипи функцій разом з поясненням їх синтаксису і виконуваних ними дій.

int sem_init (sem_t * sem, int pshared, unsigned int value); - ініціалізація семафора *sem* значенням *value. Я*к *pshared* завжди необхідно вказувати 0.

int sem _ wait (sem _ t * sem); - «очікування на семафорі». Виконання потоку блокується до тих пір, поки значення семафора не стане позитивним. При цьому значення семафора зменшується на 1.

int sem _ post (sem _ t * sem); - збільшує значення семафора *sem.*

int sem_destroy (sem_t * sem); - знищує семафор *sem.*

int sem_trywait (sem_t * sem); - неблокуючий варіант функції *sem_wait.* При цьому замість блокування що викликав потоку функція повертає управління з кодом помилки в якості результату роботи.

Приклад 7.

```
#include <semaphore.h>
 sem_t s;
 sem_init(&s, 0, 1);

int sem_wait(sem_t *s) {
 decrement the value of semaphore s by one
 wait if value of semaphore s is negative
 }

 int sem_post(sem_t *s) {
 increment the value of semaphore s by one
 if there are one or more threads waiting, wake one
 }
```

Блокуючі шлюзи читання/запису

Блокування читання-запис можна назвати інтелектуальним засобом синхронізації, оскільки вони роблять відмінність між читачами і

письменниками. У більшості випадків колективні дані частіше читають, ніж змінюють (дійсно, навіщо писати те, що ніхто не читатиме?), І це робить блокування читання-запис вельми вживаними.

Типовий приклад використання блокувань читання-запис - синхронізація доступу до буферизованная в пам'яті фрагментам файлової системи, особливо до каталогів (чим ближче до кореня, тим частіше їх читають і рідше змінюють). Важливо відзначити в цьому зв'язку, що, на відміну від блокувань, що реалізуються функцією fcntl (), блокування читання-запис можуть застосовуватися і там, де файлова система відсутня (наприклад, в мінімальних конфігураціях, що функціонують під управлінням відповідної подпрофілю стандарту POSIX операційної системи реального часу).

Блокування читання-запису (read-write locks) схожі на мютекс, але відрізняються від них тим, що мають два режими захоплення - для читання і для запису. Блокування для читання можуть утримувати кілька ниток одночасно. Блокування для запису може утримувати тільки одна нитка; при цьому ніяка інша нитка не може утримувати цю ж блокування для читання. API для роботи з блокуваннями читання-запису в цілому схожий на API для роботи з мютекс і включає в себе наступні функції:

- pthread_rwlock_init (3C)
- pthread_rwlock_rdlock (3C)
- pthread_rwlock_tryrdlock (3C)
- pthread_rwlock_timedrdlock (3C)
- pthread_rwlock_wrlock (3C)
- pthread_rwlock_trywrlock (3C)
- pthread_rwlock_timedwrlock (3C)
- pthread_rwlock_unlock (3C)
- pthread_rwlock_destroy (3C)
- pthread_rwlockattr_init
- pthread_rwlockattr_destroy
- pthread_rwlockattr_getpshared (3C)
- pthread_rwlockattr_setpshared (3C)

Набір атрибутів pthread_rwlockattr_t істотно біднішими, ніж у мютекс, і включає тільки атрибут pshared, керуючий областю дії блокування.

Застосування блокувань читання-запису досить очевидно. Вони використовуються для захисту структур даних, які читають значно частіше, ніж модифікують.

Бар’єри.

Ідея бар’єру полягає у блокуванні потоків до того часу, поки кількість потоків що очікують не досягне певного значення лічильнику бар’єру, після чого потоки продовжують свою роботу.

Оголошення бар’єру barrier_t barrier_cs;

Ініціалізація бар’єру barrier_init (barrier_t barrier_cs, int n) ;

Очікування на бар’єрі barrier_wait(barrier_cs); В результатті виконання значення лічильника збільується на 1. Якщо він менший від n, то цей потік блокується. Якщо досягаються максимальне значення, то робота всіх потоків

відновлюється. Значення лічильника для всіх потоків стає 0, а для одного з них -1. В результаті, цей потік, що отримає -1, робить підсумкову роботу.

Приклад

```
pthread_barrier_t barrier_cs;
void thread_fun()
{ int res;
step1(); //перший етап виконання
res = barrier_wait(barrier_cs);
// очікування завершення першого етапу
if (res == -1) step1_done();
step2(); // другий етап
res = barrier_wait(barrier_cs);
// очікування завершення другого етапу
if (res == -1) step2_done();
// і т.д.
}
//
void run ()
{ thread_t threads[n];
barrier_init(barrier_cs, n):
for (i=0; i<n; i++)
threads[i]=thread_create(thread_fun);
for (i=0; i<n; i++)
thread_join(threads[i]);
 }
```

Деякі додаткові можливості роботи з потоками

Сигнали:

```
#include stdio.h>
#include stdlib.h>
#include signal.h>
void term_handler(int i) // функція обробки сигналу
{
  printf ("Terminating\n");
  exit(EXIT_SUCCESS);
}
int main(int argc, char ** argv)
{
    struct sigaction sa; /* спец. структура що
використовується як параметр системного виклику
sigaction() */
    sigset_t newset; //набір сигналів
    sigemptyset(&newset); // ініціалізує набір
сигналів newset, і очищує його від всіх сигналів
    sigaddset(&newset, SIGHUP); //додає сигнал SIGHUP
в набір
```

```
    sigprocmask(SIG_BLOCK, &newset, 0); // додаєм
набір newset в группу заблококованих
    sa.sa_handler = term_handler; // додаєм в
структуру типу  sigaction вказівник на функцію
обробника сигналів
    sigaction(SIGTERM, &sa, 0);
    printf("My pid is %i\n", getpid());
    printf("Waiting...\n");

    while(1) sleep(1);
    return EXIT_FAILURE;
}
```

Поняття монітора

Умовні змінні не використовують окремо від мютексів, при цьому є правила взаємодії між цими примітивами. Ці правила є основою для поняття монітору – синхронізації вищого рівня. Монітором називається набір функцій, які використовують один загальний мютекс і нуль або більше умовних змінних для керування доступу до спільновикористовуваних даних згідно правил.

Приклад . Використання монітору для операцій над лічильником

monitor.cpp

home > oksana > forlabos > monitor.cpp > Monitor

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

class Monitor{
    private:
        int counter;
        pthread_mutex_t mutex;
    public:

        int get_counter (){
            return counter;
        }
        void inc_counter(int plus){
            pthread_mutex_lock(&mutex);
            counter+=plus;
            pthread_mutex_unlock(&mutex);
             }
        void dec_counter(int min){
            pthread_mutex_lock(&mutex);
            counter+=min;
            pthread_mutex_unlock(&mutex);
        }
    Monitor (){
        counter=0;
       mutex=PTHREAD_MUTEX_INITIALIZER;
    }
    ~Monitor(){
        pthread_mutex_destroy(&mutex);
    }
};
Monitor balance;
```

monitor.cpp

home > oksana > forlabos > monitor.cpp > balance

```
void * bal_inc(void * param)
{ int a=10;
    for (int i=0; i<10000; i++){
        for(int j=0;j<1000;j++)
        {
            balance.inc_counter(a);
        }
    }
    return NULL;
}
void * bal_dec(void * param)
{ int b=3;
    for (int i=0; i<10000; i++){
        for(int j=0;j<1000;j++)
        {
            balance.dec_counter(b);
        }

    }
    return NULL;
}

int main (){
    pthread_t th1, th2;
    pthread_create(&th1, NULL, &bal_inc,NULL);
    pthread_create(&th2, NULL, &bal_dec,NULL);
    pthread_join(th1,NULL);
    pthread_join(th2, NULL);
    std::cout<<balance.get_counter();
    return 0;
}
```

Крім того поряд з мютексом в моніторах використовують умовні змінні. Внісши деякі корективи у попередній приклад отримаємо монітор з умовними змінами:

```
// всі функції знаходяться в моніторі
// з кожним потоком пов’язанв умовна змінна finished_cond
// і прапорець finished - спільно використовувані дані
void thread_init()
{ mutex_lock(mutex);
// на початку виконання потоку умова не виконується
this_thread.finished = 0;
mutex_unlock(mutex);
}
void thread_exit()
{ mutex_lock(mutex);
this_thread.finished = 1;
// розбудити потік, який очікує, якщо він є
signal(this_thread.finished_cond, mutex);
mutex_unlock(mutex);
}
void thread_join(thread_t thread)
{ mutex_lock(mutex);
// очікування завершення потоку thread
while (! thread.finished)
wait(thread.finished_cond, mutex);
mutex_unlock(mutex);
}
```

ЗАВДАННЯ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ

1. Реалізувати заданий (згідно варіанту) алгоритм в окремому потоці.
2. Виконати розпаралелення заданого алгоритму на 2, 4, 8, 16 потоків.
3. Реалізувати можливість зміни/встановлення пріоритету потоку (для планування потоків) або встановлення відповідності виконання на ядрі.
4. Реалізувати можливість зробити потік від’єднаним.
5. Реалізувати можливість відміни потоку.
6. Реалізувати синхронізацію потоків за допомогою вказаних методів (згідно варіанту)
7. Порівняти час виконання задачі відповідно до кількості потоків і методу синхронізації (чи без синхронізації).
8. Результати виконання роботи оформити у звіт

ВАРІАНТИ ЗАВДАНЬ

(згідно порядкового номеру у підгрупі)

1. Реалізувати циклічний посимвольний вивід: прізвище ім'я, номер студентського. (кількість ітерацій >10000) (Синхронізація: умовна зміна, бар'єри)
2. Обчислити суму елементів заданого масиву (кількість елементів >10000, елементи рандомні) (Синхронізація: семафор, спінблокуавння)
3. Бінарний пошук заданого елементу масиву (кількість елементів >10000, елементи рандомні). Вивести значення та індекс. (Синхронізація: м'ютекс, умовні змінні)
4. Лінійний пошук мінімального елементу непосортованого масиву (кількість елементів >10000, елементи рандомні). Вивести значення та індекс. (Синхронізація: м'ютекс, бар'єри)
5. Сортування заданого масиву методом quick sort (кількість елементів >10000, елементи рандомні). Вивести посортований масив. (Синхронізація: семафор, спінлок) (Синхронізація: м'ютекс, бар'єри)
6. Обчислити інтеграл функції f(x) = x*sin(x/2) на проміжку [0, 100] правилом трапеції (кількість розбитів 10000) (Синхронізація: семафор, бар'єри)
7. Пошук заданого слова у файлі і вивід рядка з тим словом (кількість рядків у файлі > 1000, текст довільна наукова стаття) (Синхронізація: спінлок, умовні змінні)
8. Підрахунок кількості заданих символів у файлі. (кількість рядків у файлі > 1000, текст довільна наукова стаття) (Синхронізація: rwlock, умовні змінні)
9. Вивід слів з файлу, що розпочинаються на задану літеру (кількість рядків у файлі > 1000, текст довільна наукова стаття) (Синхронізація: спінлок, умовні змінні)
10. Обчислити суму елементів заданого масиву (кількість елементів >10000, елементи масиву задаються формулою a(0)=2, a(i)=a(i-1)*i+exp(i), і -індекс елементу) (Синхронізація: м'ютексб спінлок)
11. Пошук найдовшого слова у файлі (кількість рядків у файлі > 1000, текст довільна наукова стаття). Вивести слово і кількість символів. (Синхронізація: семафор, rwlock)
12. Обчислення n-наступних елементів ряду чисел Фібоначі, що розпочинається з f(0)=13, через рекурентні формули (n>10). (Синхронізація: м'ютекс, умовні змінні)
13. Вивід найменшого слова в кожному рядку. (кількість рядків у файлі > 1000, текст довільна наукова стаття). (Синхронізація: бар'єр, умовні змінні)
14. Піднести задану квадратну матрицю до n-го ступеню (елементи матриці випадкові, n -ввести з клавіатури). (Синхронізація: м'ютекс, бар'єри).

Якщо буде необхідність, можна використати допоміжні методи синхронізації, або об'єднати наявні.
Полегшений варіант завдання: варіант 1 – синхронізація м'ютекс, семафор. (оцінка на 1 бал менше)

КОНТРОЛЬНІ ЗАПИТАННЯ

СПИСОК ЛІТЕРАТУРИ

НАВЧАЛЬНЕ ВИДАННЯ

БАГАТОПОТОЧНІСТЬ В ОПЕРАЦІЙНІЙ СИСТЕМІ LINUX. СТВОРЕННЯ, КЕРУВАННЯ ТА СИНХРОНІЗАЦІЯ ПОТОКІВ.

МЕТОДИЧНІ ВКАЗІВКИ

до виконання лабораторної роботи №6
з дисципліни «Операційні системи»
для студентів першого (бакалаврського) рівня вищої освіти
спеціальності 121 «Інженерія програмного забезпечення»

Укладачі: Грицай Оксана Дмитрівна, канд. фіз.-мат. наук,