Дисципліна «Операційні системи»

Лабораторна робота №13

Тема: «Програмування безпечної міжпроцесної та міжпотокової взаємодії в *Unix*-подібних ОС»

Викладач: Олександр А. Блажко,

доцент кафедри ІС Одеської політехніки, blazhko@ieee.org

Мета роботи: дослідити проблеми міжпроцесної та міжпотокової взаємодії та засоби їх безпечної взаємодії на прикладі семафорів та м`ютексів.

1 Теоретичні відомості

1.1 Загальні проблеми міжпроцесної взаємодії

Проблема взаємодії між процесами розбивається на три пункти:

1) передача інформації від одного процесу до іншого;

2) контроль за тим, щоб два процеси не перетиналися в критичних ситуаціях, наприклад, два процеси намагаються заволодіти однією і тією ж областю пам'яті;

3) синхронізація, наприклад, якщо процес *A* повинен поставляти дані, а процес *B* повинен почекати і не починати друкувати ці дані, поки вони не надійдуть від процесу *A*.

Перший пункт не стосується потоків, оскільки у потоків загальний адресний простір.

Процеси, які працюють спільно, можуть спільно використовувати деяке сховище даних, наприклад, файл загального доступу або ділянка оперативної пам'яті. Кожен з процесів може зчитувати дані із загального сховища даних і записувати їх туди.

Ситуація змагання – ситуація, в якій два та більше процесів зчитують або записують дані одночасно, і кінцевий результат залежить від того, який з цих процесів був першим.

Загальноприйнята взаємодія між різними процесами – асинхронна взаємодія, на противагу синхронній взаємодії між різними модулями послідовно виконуваної програми. Проблема виникає лише тоді, коли модифікації піддається об'єкт, що розділяється декількома процесами. Для виключення проблеми достатньо, щоб тільки один процес займався модифікацією, а всі інші враховували стан об'єкта.

Основний спосіб запобігання проблем в цій і іншій ситуації, пов'язаної зі спільним використанням пам'яті, файлів і чого-небудь ще, є заборона одночасного запису і читання розділених даних більш, ніж одним процесом через взаємне виключення, коли один процес використовує розділені дані, а іншому процесу це буде заборонено робити.

Частина програми, в якій є звернення до спільно використовуваних даних, називається критичною областю або критичної секцією. Якщо вдається уникнути

одночасного перебування двох процесів в критичних областях, можливо уникнути виникнення ситуації змагань. Для цього необхідно виконання чотирьох умов уникання виникнення ситуації змагань:

1) два процеси не повинні одночасно перебувати в критичних областях;
2) у програмі не повинно бути припущень про швидкість або кількість процесів;
3) процес, знаходячись поза критичної області, не може блокувати інші процеси;
4) неможлива ситуація, в якій процес вічно чекає попадання в критичну область.

1.2 Теоретичні ідеї синхронізації роботи процесів

1.2.1 Алгоритм «Змінні блокування» для взаємного виключення процесів/потоків

Основна ідея боротьби зі змаганнями – використання блокування:

– для контролю входу до критичної області використовується одна спільна змінна блокування – *змінна-прапорець* або *сигнальна змінна*;

– процеси перед входом у критичну область, перевіряють значення сигнальної змінної.

Розглянемо два процеси, програмний код яких представлено на рисунку 1.

```
// process PR1
while(1) {
  while (turn!=0); // loop
      turn=1;
      critical_region();
      turn=0;
      noncritical_region();
}
```

```
// process PR2
while(1) {
      while (turn!=0);// loop
      turn=1;
      critical_region ();
      turn=0;
      noncritical_region ();
}
```

Рис. 1 – Приклад програмного коду для двох процесів

Припустимо, що:

1) процес *PR1* зчитує змінну *turn*,
2) процес *PR1* визначає, що *turn* = 0,
3) перш ніж процес *PR1* встигає встановити *turn = 1*, управління отримує процес *PR2*
4) процес *PR2* встигає встановити *turn = 1* раніше ніж процес *PR1*
5) коли процес *PR1* знову отримає управління, він успішно встановить *turn = 1*
6) два процеси одночасно опиняться в критичних областях.

На жаль, саме завдяки простоті представлених прикладів програм вони не можуть забезпечити відсутність змагань:

– перевірка/зміна сигнальної змінної реалізуються двома різними операторами;

– в проміжку між різними операторами інший процес може отримати управління і також змінити значення сигнальної змінної.

Так зване «програмне вікно», в якому відбувається змагання, може становити всього 2-3 команди, але при попаданні обох процесів в це вікно, буде отримано саме те, чого слід було уникнути: обидва процеси можуть увійти у критичну секцію.

На рисунку 2 представлено приклад програми *lockvars.c*, яка імітує роботу двох потоків на основі алгоритму «Змінні блокування» для взаємного виключення потоків.

На рисунку 3 представлено приклад виконання програми *lockvars,* де видно одночасну появу двох процесів у критичній області.

```c
// компіляція з підключенням бібліотеки -lpthread
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

int turn = 0; // сигнальна змінна

int main(void) {
    pthread_t f2_thread, f1_thread;
    void *PR1(), *PR2(); // функції конкуруючих потоків
    pthread_create(&f1_thread, NULL, PR1, NULL);
    pthread_create(&f2_thread, NULL, PR2, NULL);
    pthread_join(f1_thread, NULL);
    pthread_join(f2_thread, NULL);
    return 0;
}

void *PR1() {
    while(1) {
        while (turn != 0);
        sleep(2); // затримка PR1, пов'язана з операціями
        turn = 1;
        printf("PR1: Critical Region.\n");
        turn = 0;
        printf("PR1: Noncritical Region.\n");
    }
}

void *PR2() {
    while(1) {
        sleep(1); // PR2 стартує пізніше PR1
        while (turn != 0);
        turn = 1;
        printf("PR2: Critical Region.\n");
        sleep(3); // затримка PR1, пов'язана з операціями
        turn = 0;
        printf("PR2: Noncritical Region.\n");
    }
}
```

Рис. 2 – Приклад програми *lockvars.c* з імітацією роботи двох потоків з використанням алгоритму «Змінні блокування» для взаємного виключення потоків

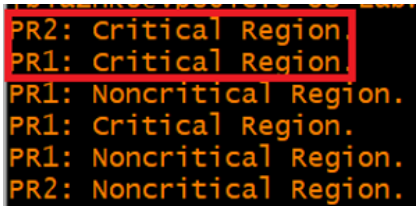

Рис. 3 – Приклад виконання програми *lockvars*

1.2.2 Алгоритм «Строге чергування» для взаємного виключення процесів/потоків

Перетворимо алгоритм керування процесами «Змінні блокування» в алгоритм «Строге чергування».

Розглянемо два процеси, програмний код яких представлено на рисунку 4.

```
// PR1
while(1){
        while (turn!=0);// loop
        critical_region();
        turn=1;
        noncritical_region();
}
```

```
// PR2
while(1){
        while (turn==0);// loop
        critical_region();
        turn=0;
        noncritical_region();
}
```

Рис. 4 – Приклад програмного коду алгоритму «Строге чергування» для двох процесів

Припустимо, що спочатку *turn = 0.*

Процес *PR1* спочатку перевіряє значення *turn*, зчитує 0 і входить у критичну область.

Процес *PR2* також перевіряє значення *turn*, зчитує 0 і входить у нескінченний цикл, чекаючи коли ж значення *turn* зміниться.

Процес *PR1* змінює значення *turn = 1* і входить у некритичну область.

Процес *PR2* перевіряє значення *turn*, зчитує 1 і входить у критичну область.

Постійна перевірка значення сигнальної змінної називається *активним очікуванням*.

Недолік алгоритму блокування – безцільна витрата процесорного часу.

Spinlock (spin – кружляння), циклічне блокування, – блокування, яке використовує активне очікування, коли процес нічого корисного не робить, але використовує ресурси системи.

Наявність циклічного блокування порушує третє із сформульованих умов уникання виникнення ситуації змагань: один процес, заблокований іншим, не перебуває у критичній області.

На рисунку 5 представлено приклад програми *strictwatching.c*, яка імітує роботу двох потоків з використанням алгоритму «Змінні блокування» для взаємного виключення потоків.

На рисунку 6 представлено приклад виконання програми *strictwatching,* де не видно одночасної появи двох процесів у критичній області.

```c
// компіляція з підключенням бібліотеки -lpthread
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

int turn = 0; // сигнальна змінна

int main(void) {
    pthread_t f2_thread, f1_thread;
    void *PR1(), *PR2(); // функції конкуруючих потоків
    pthread_create(&f1_thread, NULL, PR1, NULL);
    pthread_create(&f2_thread, NULL, PR2, NULL);
    pthread_join(f1_thread, NULL);
    pthread_join(f2_thread, NULL);
    return 0;
}

void *PR1() {
    while(1) {
        while (turn != 0);
        sleep(2); // затримка PR1, пов'язана з операціями
        printf("PR1: Critical Region.\n");
        turn = 1;
        printf("PR1: Noncritical Region.\n");
    }
}

void *PR2() {
    while(1) {
        sleep(1); // PR2 стартує пізніше PR1
        while (turn == 0);
        printf("PR2: Critical Region.\n");
        sleep(3); // затримка PR1, пов'язана з операціями
        turn = 0;
        printf("PR2: Noncritical Region.\n");
    }
}
```

Рис. 5 – Приклад програми *strictwatching.c* з імітацією роботи двох потоків з використанням алгоритму «Строге чергування» для взаємного виключення потоків

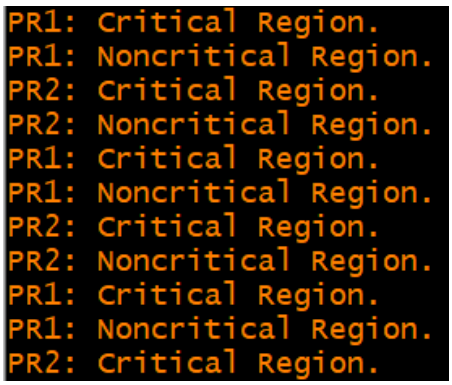

Рис. 6 – Приклад виконання програми *strictwatching*

1.2.3 Алгоритм Петерсона для взаємного виключення потоків

В 1981 році Гаррі Петерсон розробив алгоритм паралельного програмування для взаємного виключення потоків виконання коду, який спочатку був сформульований для двопотокового випадку, але в подальшому був узагальнений для довільної кількості потоків.

На рисунку 7 наведено приклад програмного коду алгоритму Петерсона.

```
// PR1
while(1) {
  flag[0] = 1;
  turn = 1;
  while (flag[1] == 1 && turn ==1 ) {
  // очікування
  }
   critical_region();
 flag[0] = 0;
  noncritical_region();
}
```

```
// PR2
while(1) {
  flag[1] = 1;
  turn = 0;
  while (flag[0] == 1 && turn == 0 ) {
  // очікування
  }
 critical_region();
 flag[1] = 0;
  noncritical_region();
}
```

Рис. 7 – Приклад програмного коду алгоритму Петерсона.

Початково обидва процеси знаходяться за межами критичної області.

Алгоритм має два типу змінних: *flag[]* – масив прапорців зацікавленості потоків у вході до критичної секції, *turn* – сигнальна змінна.

Якщо відповідний елемент масиву *flag* встановлений в 1, тоді і відповідний процес намагається увійти до критичної секції.

Вхід до критичної секції надається процесу *P1*, якщо *P2* не намагається увійти до своєї критичної секції або *P2* надав пріоритет P1 встановленням *turn* в 0.

Алгоритм Петерсона також має недолік наявності циклічного блокування, але завдяки масиву прапорців час такого блокування зменшується, що є основною перевагою алгоритму Петерсона у порівнянні з алгоритмом «Строге чергування».

На рисунку 8 представлено приклад програми *peterson.c*, яка імітує роботу двох потоків з використанням алгоритму Петерсона для взаємного виключення потоків.

```c
// компіляція з підключенням бібліотеки -lpthread
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

int flag[2]; /* масив прапорців зацікавленості потоків
                у вході до критичної секції */
int turn; // сигнальна змінна

int main(void) {
    pthread_t f2_thread, f1_thread;
    void *PR1(), *PR2(); // функції конкуруючих потоків
    pthread_create(&f1_thread, NULL, PR1, NULL);
    pthread_create(&f2_thread, NULL, PR2, NULL);
    pthread_join(f1_thread, NULL);
    pthread_join(f2_thread, NULL);
    return 0;
}

void *PR1() {
    while(1) {
        flag[0] = 1;
        turn = 1;
        while (flag[1] ==1 && turn == 1); // очікування
        sleep(2); // затримка PR1, пов'язана з операціями
        printf("PR1: Critical Region.\n");
        flag[0] = 0;
        printf("PR1: Noncritical Region.\n");
    }
}
void *PR2() {
    while(1) {
        sleep(1); // PR2 стартує пізніше PR1
        flag[1] = 1;
        turn = 0;
        while (flag[0] ==1 && turn == 0);
        printf("PR2: Critical Region.\n");
        sleep(3); // затримка PR1, пов'язана з операціями
        flag[1] = 0;
        printf("PR2: Noncritical Region.\n");
    }
}
```

Рис. 8 – Приклад програми *peterson.c* з імітацією роботи двох потоків з використанням алгоритму Петерсона для взаємного виключення потоків

1.3 Сучасні засоби синхронізації роботи процесів через семафори та м’ютекси

1.3.1 Синхронізація роботи процесів на основі семафорів

У 1965 році Едсгер Дейкстра запропонував використовувати цілу змінну для підрахунку сигналів запуску процесів – *семафор* (*semaphore*), значення якого може бути нулем або деяким позитивним числом.

Змінювати значення семафору можуть дві операції: операція *down* та операція *up.*

Операція *down* порівнює значення семафора з нулем:

– якщо значення семафора > 0, тоді *down* його зменшує і повертає керування;

– якщо значення семафора = 0, тоді процедура *down* не повертає керування процесу, а переводить його у стан очікування.

Операції перевірки/зміни значення семафора або переведення процесу в стан очікування виконуються як єдина і неподільна дія, яка є *атомарною*. Таким чином, за рахунок атомарності, гарантується, що після початку операції жоден процес не отримає доступу до семафора до закінчення або блокування операції.

Операція *up* збільшує значення семафора на 1. Якщо з цим семафором пов'язані один або декілька процесів, що очікують, які не можуть завершити більш ранню операцію *down*, один з них вибирається системою. Таким чином, після виконання операції *up* над семафором, пов'язаним з декількома процесами, що очікують значення семафора, так і залишиться рівним 0, але число процесів, що очікують зменшиться на одиницю. Жоден процес не може бути блокований під час виконання операції *up*.

Намагаючись пройти через семафор, процес намагається відняти 1 від значення семафора. Якщо значення семафора >= 1, семафор відкритий і процес проходить крізь семафор. Якщо значення = 0 (семафор закритий), процес зупиняється і стає в чергу.

Закриття семафора відповідає захопленню ресурсу, доступ, до якого контролюється цим семафором. Якщо ресурс захоплений, тоді інші процеси змушені чекати його звільнення. Вийшовши з критичної секції, процес збільшує значення семафора на одиницю, відкриваючи його. При цьому перший, з тих процесів, що стояли в черзі, активізується, віднімає зі значення семафора одиницю, знову закриваючи семафор. Якщо ж черга була порожня, тоді нічого не відбувається, а просто семафор залишається відкритим.

Семафори можуть приймати будь-які невід'ємні значення, ще називаються лічильниками (*counting semaphore*). Це відповідає випадку, коли декілька процесів можуть працювати з ресурсом одночасно, бо ресурс складається з декількох незалежних, але рівноцінних частин, наприклад, кількох однакових принтерів.

Якщо лічильник може приймати лише два значення – 0 та 1, тоді семафор називається двійковим (бінарним) семафором.

1.3.2 Синхронізація роботи процесів на основі м’ютексів

М'ютекси (*MutEx – Mutual Exclusion* – взаємне виключення) використовуються коли з ресурсом в кожен момент часу може працювати тільки один процес/потік.

Значення м'ютексу встановлюється двома процедурами:

– *mutex_lock* – встановити блокування, коли процес/потік збирається увійти в критичну область, і якщо м'ютекс не заблокований, запит виконується, а процес/потік, який його викликає, може зайти до критичної області;

– *mutex_unlock* – зняти блокування і, якщо м'ютекс закритий, то процес/потік, який намагається увійти до критичну секцію, блокується.

Треба пом`ятати, що м'ютекс, забезпечуючи контроль двох станів ресурсу, НЕ є особливим випадком двійкового семафору, тому що:

– м'ютекс може бути звільнений, тобто значення м'ютексу може бути змінено, тільки процесом/потоком, який вперше визначив його значення;

– значення двійкового семафору може змінюватися будь-яким процесом/потоком.

Семафори більше підходять для вирішення деяких проблем синхронізації, таких як «Постачальник-Споживач», модель взаємодії якого має наступні особливості:

– декілька процесів використовують буфер обмеженого розміру;

– процеси як процеси-постачальники, поміщають в буфер дані

– процеси як процеси-споживачі зчитує дані з буферу;

– потрібна змінна *count* для відстеження кількості елементів в буфері;

– кожен з процесів повинен перевіряти значення *count*:

  - процес-постачальник – щоб не переповнити буфер;
  - процес-споживач – щоб перейти до стану очікування, якщо *count* = 0.

1.3.3 Синхронізація роботи процесів на основі ф’ютексів

В 2003 році колективом авторів у складі Уберту Франке (*IBM Thomas J. Watson Research Center*), Мэттью Кирквудом, Инго Молнаром (*Red Hat*) та Расти Расселом (*IBM Linux Technology Center*) було запропоновано *futex* (англ. *Fast Userspace muTEX*) – низькорівневий механізм синхронізації процесів, на основі якого в подальшому стали створювати інші механізми, такі як м'ютекси та семафори.

Ф’ютекс описується:

– фрагментом у спільній пам'яті для кількох процесорів у режимі користувача;

– чергою очікування у режимі ядра.

Значення змінної, що розділяється між процесами, може бути збільшено або зменшено на одиницю за одну асемблерну інструкцію. Процеси, «зав'язані» на цей ф'ютекс, чекають, коли це значення стане позитивним. Всі операції з ф'ютексами практично

повністю проводяться в просторі користувача (за відсутності змагання, відповідні функції ядра задіяні лише в обмеженому наборі спірних випадків, що дозволяє підвищити ефективність використання синхронізуючих примітивів, оскільки більшість операцій не використовує арбітраж, а отже, і уникнути використання відносно дорогих системних викликів у режимі ядра.

1.3.4 Програмування семафорів

Для створення семафору використовується функцію *sem_open* з наступним форматом виклику функції:

```
sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
```

Функція має наступні аргументи:

– *name* – назва семафору, в якій на початку має бути символ "/", а довжина не повинна перевищувати 251 знаки;

– *oflag* – керуючий прапорець роботи із семафором:

  - *O_CREATE* – для створення семафору;
  - *O_EXCL* – для використання вже створеного семафору;

– *mode* – права доступу, які визначаються за форматом команди *chmod*;

– *value* – початкове значення семафору.

Якщо початкове значення = 1 та в подальшому воно приймає лише два значення – 0 та 1, тоді семафор стає бінарним (двійковим) семафором.

Функція повертає адресу семафору, або *SEM_FAILED* – при помилці.

Для збільшення значення семафору (процедура *up*) викликається функція *sem_post*:

```
int sem_post(sem_t *sem);
```

Виклик функції може відновити виконання будь-яких процесів/потоків, які очікують зміни значення семафора. Функція повертає значення 0 у разі успішного виконання.

Функція *sem_wait* зменшує значення семафору (процедура *down*):

```
int sem_wait(sem_t *sem);
```

Якщо значення семафору > 0, тоді виконується зменшення значння і функція відразу завершується. Якщо значення семафора = 0, тоді виклик блокується до тих пір, поки не стане можливим виконати це зменшення.

Функція *sem_trywait* подібна *sem_wait* за винятком того, що якщо зменшення не можливо виконати відразу, тоді виклик завершується помилкою, а не блокується.

Функція *sem_close* завершує роботу семафору:

```
int sem_close(sem_t *sem);
```

Опис вказаних функцій міститься у *header*-файлі *semaphore.h*

На рисунку 9 представлено приклад програми *semaphore.c* використання семафору для синхронізації роботи двох процесів:

– 1-й процес створює семафор з початковим значенням лічильника = 0;

– 1-й процес намагається отримати доступ до семафору, але починає чекати, доки значення лічильника не стане > 0;

– 2-й процес відкриває вже створений семафор;

– 2-й процес збільшує значення лічильника на 1;

– 1-й процес отримує доступ до семафору, зменшуючи значення лічильника на 1.

```c
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
#include <stdio.h>

#define SEMAPHORE_NAME "/blazhko_sync_semaphore"

int main(int argc, char ** argv) {
    sem_t *sem;
    if ( argc < 2 ) {
        if ((sem = sem_open(SEMAPHORE_NAME,
                O_CREAT, 0777, 0)) == SEM_FAILED ) {
            fprintf(stderr,"sem_open error");
            return 1;
        }
        printf("PR1: semaphore is open. I'm waiting for a signal...\n");
        if (sem_wait(sem) < 0 )
            fprintf(stderr,"sem_wait error");
        printf("PR1: I got the signal!\n");
        if ( sem_close(sem) < 0 )
            fprintf(stderr,"sem_close error");
        return 0;
    }
    else {
        printf("PR2: I opened semaphore to send a signal...\n");
        if ( (sem = sem_open(SEMAPHORE_NAME, 0)) == SEM_FAILED ) {
            fprintf(stderr,"sem_open error");
            return 1;
        }
        sem_post(sem);
        printf("PR2: I sent the signal!\n");
        return 0;
    }
}
```

Рис. 9 – Приклад програми *semaphore.c* із семафором для синхронізації роботи процесів

В 1-му псевдотерміналі запускається програма без параметрів, імітуючи роботу 1-го процесу, а у 2-му псевдотерміналі запускається програма з параметром 1, імітуючи роботу 2-го процесу після чого на екран виводиться повідомлення, як показано на рисунку 10.

```
[blazhko@vps6iefe OS-LabWork13-Examples]$ ./semaphore
PR1: semaphore is open. I'm waiting for a signal...
PR1: I got the signal!
[blazhko@vps6iefe OS-LabWork13-Examples]$ ./semaphore 1
PR2: I opened semaphore to send a signal...
PR2: I sent the signal!
```

Рис. 10 – Приклад виконання двох програм-процесів, які використовують семафор

На рисунку 11 представлено приклад програми *semaphore2.c*, яка імітує роботу двох потоків з використанням двійкового семафору для взаємного виключення потоків.

```c
#include <fcntl.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>
#define SEMAPHORE_NAME "/blazhko_semaphore7"
int main(int argc, char ** argv) {
    sem_t *sem;
    pthread_t f2_thread, f1_thread;
    void *PR1(), *PR2();
    if ((sem = sem_open(SEMAPHORE_NAME,O_CREAT, 0777, 1)) == SEM_FAILED ) {
        fprintf(stderr,"sem_open error");
        return 1;
    }
    pthread_create(&f1_thread, NULL, PR1, sem);
    pthread_create(&f2_thread, NULL, PR2, sem);
    pthread_join(f1_thread, NULL);
    pthread_join(f2_thread, NULL);
    if ( sem_close(sem) < 0 ) {
        fprintf(stderr,"sem_close error");
        return 1;
    }
    return 0;
}
void *PR1(sem_t *sem) {
    for (int i=1;i<=5;i++) {
        if (sem_wait(sem) != 0 ) fprintf(stderr,"PR1: sem_wait error!\n");
        printf("PR1: Critical Region.\n");
        if (sem_post(sem) !=0 ) fprintf(stderr,"PR1: sem_post error!\n");
        printf("PR1: Noncritical Region.\n");
        sleep(1); // затримка, пов'язана із операціями PR1
    }
}
void *PR2(sem_t *sem) {
    sleep(1); // PR2 стартує пізніше PR1
    for (int i=1;i<=5;i++) {
        if (sem_wait(sem) != 0 ) fprintf(stderr,"PR2: sem_wait error!\n");
        printf("PR2: Critical Region.\n");
        if (sem_post(sem) !=0 ) fprintf(stderr,"PR2: sem_post error!\n");
        printf("PR2: Noncritical Region.\n");
        sleep(1); // затримка, пов'язана із операціями PR2
    }
}
```

Рис. 11 – Приклад програми *semaphore2.c*, яка імітує роботу двох потоків з використанням двійкового семафору для взаємного виключення потоків

1.3.5 Програмування м`ютексів

Для використання м`ютекса необхідно викликати функцію *pthread_mutex_init*:

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

Функція отримує в якості аргументів:

– *mutex* – змінна *mutex*-типу;

– *mutexattr* – значення змінної, якщо *mutexattr* = *NULL*, то м'ютекс ініціалізується значенням за замовчуванням.

У разі успішного виконання функції (код возрата 0), м'ютекс вважається ініціалізованим і «вільним».

Якщо достатньо використати атрибути м'ютексу за замовчуванням, тоді замість виклику функції *pthread_mutex_init* можна статично ініціалізувати м'ютекс-змінну із використанням макросу *PTHREAD_MUTEX_INITIALIZER*, наприклад:

```
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Щоб зайняти або звільнити м'ютекс, використовуються функції:

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

Функція *pthread_mutex_lock* займає м'ютекс, якщо він не зайнятий, стає його власником і відразу ж виходить. Якщо м'ютекс зайнятий, то функція блокує подальше виконання процесу і чекає звільнення м'ютексу.

Коди повернення для функції *pthread_mutex_lock*:

EINVAL – *mutex* неправильно ініціалізовано;

EDEADLK – м'ютекс вже зайнятий поточним процесом.

Функція *pthread_mutex_trylock* ідентична поведінці функції *pthread_mutex_lock*, але вона не блокує процес, якщо м'ютекс зайнятий, а повертає *EBUSY* код.

Фунція *pthread_mutex_unlock* звільняє зайнятий м'ютекс.

Коди повернення для *pthread_mutex_unlock*:

EINVAL – м'ютекс неправильно ініціалізовано;

EPERM – процес не є володарем м`ютекса.

На рисунку 12 наведено приклад програми, яка імітує роботу двох потоків з використанням м'ютексу для взаємного виключення потоків.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <errno.h>

static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int main(int argc, char ** argv) {
    pthread_t f2_thread, f1_thread;
    void *PR1(), *PR2();
    pthread_create(&f1_thread, NULL, PR1, NULL);
    pthread_create(&f2_thread, NULL, PR2, NULL);
    pthread_join(f1_thread, NULL);
    pthread_join(f2_thread, NULL);
    return 0;
}
void *PR1() {
    while(1) {
        if (pthread_mutex_lock(&mutex) == EDEADLK )
            fprintf(stderr,"PR1: вже зайнятий поточним процесом!\n");
        printf("PR1: Critical Region.\n");
        if (pthread_mutex_unlock(&mutex) == EPERM )
            fprintf(stderr,"PR1: EPERM - процес не є власником mutex!\n");
        printf("PR1: Noncritical Region.\n");
        sleep(1); // затримка, пов'язана із операціями PR1
    }
}
void *PR2() {
    sleep(1); // PR2 стартує пізніше PR1
    while(1) {
        if (pthread_mutex_lock(&mutex) == EDEADLK )
            fprintf(stderr,"PR2: вже зайнятий поточним процесом!\n");
        printf("PR2: Critical Region.\n");
        if (pthread_mutex_unlock(&mutex) == EPERM )
            fprintf(stderr,"PR2: EPERM - процес не є власником mutex!\n");
        printf("PR2: Noncritical Region.\n");
        sleep(1); // затримка, пов'язана із операціями PR1
    }
}
```

Рис. 12 – приклад програми, яка імітує роботу двох потоків з використанням м’ютексу для взаємного виключення потоків

1.4 Програмна реалізація протоколів блокування операцій транзакцій

Припустимо, що існує база даних із таблицею банківських рахунків *bill*.

Для проведення експериментів над механізмами взаємного виключення декількох потоків створено програму з описом структури даних, яка співпадає зі структурою таблиці *bill* за прикладом з рисунку 13.

```
// структура "банківський рахунок"
struct bill {
    int n_bill;
    int bill_count;
    char name[20];
};
struct bill bill[2] = { {1, 100, "Ivanenko" } };
```

Рис. 13 – Опис структури даних, яка співпадає зі структурою таблиці *bill*

Припустимо також, що існують дві транзакції *T1* та *T2*, які виконують *SQL*-операції над таблицею *bill*. В таблиці 1 наведно приклад квазіпаралельного виконання *SQL*-операцій цих транзакцій.

Таблиця 1 – Приклад квазіпаралельного виконання *SQL*-операцій двох транзакцій

№	*SQL*-операція транзакції *T1*	*SQL*-операція транзакції *T2*
1.	*START TRANSACTION;*	
2.		*START TRANSACTION;*
3.	*SELECT bill_count FROM bill WHERE n_bill =1;*	
4.		*SELECT bill_count FROM bill WHERE n_bill =1;*
5.	*UPDATE bill SET bill_count = 10 WHERE n_bill =1;*	
6.		*UPDATE bill SET bill_count = 20 WHERE n_bill =1;*
7.	*COMMIT;*	
8.		*COMMIT;*

Для проведення експериментів над механізмами взаємного виключення декількох потоків з використанням алгоритму «Змінні блокування» створено програму *db_lockvars.c*, приклад якої представлено на рисунку 14.

В програмному коді виконано наступні заміни для *SQL*-операцій з таблиці 1:

– *SELECT*-операцію замінено на операцію читання значння елементу масива;

– *UPDATE*-операцію замінено на операцію зміни значння елементу масива.

Програмний код функції *T1, T2* майже співпадає.

Головна відмінність: в програмному коді функції *T1* для імітації затримки між оператором активного очікування (*spinlock*) включено функцію затримку *sleep(2)*, що відвищує ймовірність потрапляння 2-го потоку до критичної секції одночасно із 1-м потоком.

Для чередування операцій транзакцій також додано декілька функцій затримки *sleep(1).*

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <errno.h>
int turn = 0;
struct bill {
    int n_bill;
    int bill_count;
    char name[20];
};
struct bill bill[2] = { {1, 100, "Ivanenko" } };
void main(void) {
    pthread_t T1_thread, T2_thread;
    void *T1(), *T2();
    pthread_create(&T1_thread, NULL, T1, NULL);
    pthread_create(&T2_thread, NULL, T2, NULL);
    pthread_join(T1_thread, NULL);
    pthread_join(T2_thread, NULL);
}
void *T1() {
    for (int i=1;i<=3;i++) {
        while (turn != 0);
        sleep(2);
        turn = 1;
        printf("T1: Critical Region\n");
        printf("T1: Read[bill_count]=%d\n",bill[0].bill_count);
        sleep(1);
        bill[0].bill_count = 10;
        printf("T1: Write[bill_count]=%d\n",bill[0].bill_count);
        sleep(1);
        turn = 0;
        printf("T1: Noncritical Region\n");
        sleep(1);
    }
}
void *T2() {
    sleep(1); // T2 стартує пізніше T1
    for (int i=1;i<=3;i++) {
        while (turn != 0);
        turn = 1;
        printf("T2: Critical Region\n");
        printf("T2: Read[bill_count]=%d\n",bill[0].bill_count);
```

Рис. 14 – Приклад програми *db_lockvars.c*

На рисунку 15 наведено приклад результату роботи програми *db_lockvars*, де видно про паралельний вхід транзакцій до критичних секцій.

```
T2: Critical Region
T2: Read[bill_count]=100
T1: Critical Region
T1: Read[bill_count]=100
T2: Write[bill_count]=10
T1: Write[bill_count]=10
T2: Noncritical Region
T1: Noncritical Region
```

Рис. 15 – Приклад результату роботи програми *db_lockvars*

Для проведення експериментів над механізмами взаємного виключення декількох потоків на основі м’ютексів створено програму *db_mutex.c*, приклад якої представлено на рисунку 16.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <errno.h>
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
struct bill {
    int n_bill;
    int bill_count;
    char name[20];
};
struct bill bill[2] = { {1, 100, "Ivanenko" } };
void main(void) {
    pthread_t T1_thread, T2_thread;
    void *T1(), *T2();
    pthread_create(&T1_thread, NULL, T1, NULL);
    pthread_create(&T2_thread, NULL, T2, NULL);
    pthread_join(T1_thread, NULL);
    pthread_join(T2_thread, NULL);
}
void *T1() {
    for (int i=1;i<=3;i++) {
        pthread_mutex_lock(&mutex);
        printf("T1: Critical Region\n");
        printf("T1: Read[bill_count]=%d\n",bill[0].bill_count);
        sleep(1);
        bill[0].bill_count = 10;
        printf("T1: Write[bill_count]=%d\n",bill[0].bill_count);
        pthread_mutex_unlock(&mutex);
        printf("T1: Noncritical Region\n");
        sleep(1);
    }
}
void *T2() {
    sleep(1); // T2 стартує пізніше T1
    for (int i=1;i<=3;i++) {
        pthread_mutex_lock(&mutex);
        printf("T2: Critical Region\n");
        printf("T2: Read[bill_count]=%d\n",bill[0].bill_count);
        sleep(1);
        bill[0].bill_count = 20;
        printf("T2: Write[bill_count]=%d\n",bill[0].bill_count);
```

Рис. 16 – Приклад програми *db_mutex.c*

2 Завдання до лабораторної роботи

Виконати наступні дії з підготовки до виконання завдань роботи:

1) встановити з'єднання з *Linux*-сервером з *IP*-адресою = 46.175.148.116;
2) перейти до каталогу *Git*-репозиторія;
3) створити нову *Git*-гілку з назвою «*Laboratory-work-13»*;
4) перейти до роботи зі створеною гілкою;
5) створити каталог з назвою «*Laboratory-work-13»*;
6) перейти до каталогу «*Laboratory-work-13»;*
7) в каталозі «*Laboratory-work-13»* створити файл *README.md* та додати до файлу рядок тексту із темою лабораторної роботи *«Програмування безпечної міжпроцесної та міжпотокової взаємодії в Unix-подібних ОС»* як заголовок 2-го рівня *Markdown*-форматування;
8) виконати операції з оновлення *GitHub*-репозиторію змінами *Git*-репозиторія через послідовність *Git*-команд *add, commit* із коментарем «*Changed by Local Git*» та *push*;
9) на веб-сервісі *GitHub* перейти до створеної гілки «*Laboratory-work-13»;*
10) перейти до каталогу «*Laboratory-work-13»* та розпочати процес редагування файлу *README.md* ;

в подальшому за результатами рішень кожного наступного розділу завдань до файлу *README.md* додавати рядки як заголовки 3-го рівня *Markdown*-форматування з назвами розділу та знімки екрані з підписами до знімків екранів з описом пунктів завдань.

2.1 Аналіз наявності змагань у міжпотоковій взаємодії на основі алгоритму «Змінні блокування»

2.1.1 В лабораторній роботі №10 було використано таблицю реляційної бази даних. Для проведення експериментів над механізмами взаємного виключення декількох потоків під час паралельного доступу до структурної змінної з використанням алгоритму «Змінні блокування» створити *C*-програму за прикладом на рисунку 14 та назвою за шаблоном «*db_lockvars_ваше_прізвище.c*», яка містить:

- опис структурної змінної, який відповідає структурі реляційної таблиці;
- налаштування роботи двох потоків через окремі функції з назвою *T1,T2*;
- структури даних роботи алгоритму «Змінні блокування» в програмному коді потокових функцій;
- кожна функція повинна:
  - спочатку виконати операцію читання значення будь-якого елемента структури;
  - в подальшому виконати операцію зміни значення вказаного елемента структури;

– функції затримки для імітації паралельної роботи потоків.

2.1.2 Скомпілювати програму та перевірити її роботу. Зробити висновок щодо наявності або відсутності змагань потоків.

2.2 Аналіз наявності змагань у міжпотоковій взаємодії на основі алгоритму «Строге чергування»

2.2.1 Для проведення експериментів над механізмами взаємного виключення декількох потоків під час паралельного доступу до структурної змінної з використанням алгоритму «Строге чергування» створити *C*-програму з назвою за шаблоном «*db_strictwatching_ваше_прізвище.c*», яка містить:

– опис структурної змінної, який відповідає структурі реляційної таблиці;

– налаштування роботи двох потоків через окремі функції з назвою *T1,T2*;

– структури даних роботи алгоритму «Строге чергування» в програмному коді потокових функцій;

– кожна функція повинна:

o спочатку виконати операцію читання значення будь-якого елемента структури;

o в подальшому виконати операцію зміни значення вказаного елемента структури;

– функції затримки для імітації паралельної роботи потоків.

2.2.2 Скомпілювати програму та перевірити її роботу. Зробити висновок щодо наявності або відсутності змагань потоків.

2.3 Аналіз наявності змагань у міжпотоковій взаємодії на основі алгоритму Петерсона

2.3.1 Для проведення експериментів над механізмами взаємного виключення декількох потоків під час паралельного доступу до структурної змінної з використанням алгоритму Петерсона створити *C*-програму з назвою за шаблоном «*db_peterson_ваше_прізвище.c*», яка містить:

– опис структурної змінної, який відповідає структурі реляційної таблиці;

– налаштування роботи двох потоків через окремі функції з назвою *T1,T2*;

– кожна функція повинна:

o спочатку виконати операцію читання значення будь-якого елемента структури;

o в подальшому виконати операцію зміни значення вказаного елемента структури;

– структури даних роботи алгоритму Петерсона в програмному коді потокових функцій;

– функції затримки для імітації паралельної роботи потоків.

2.3.2 Скомпілювати програму та перевірити її роботу. Зробити висновок щодо наявності або відсутності змагань потоків.

2.4 Аналіз наявності змагань у міжпотоковій взаємодії на основі двійкового семафору

2.4.1 Для проведення експериментів над механізмами взаємного виключення декількох потоків під час паралельного доступу до структурної змінної з використанням двійкового семафору створити *C*-програму з назвою за шаблоном «*db_semaphore2_ваше_прізвище.c*», яка містить:

- опис структурної змінної, який відповідає структурі реляційної таблиці;
- налаштування роботи двох потоків через окремі функції з назвою *T1,T2*;
- кожна функція повинна:
  - спочатку виконати операцію читання значення будь-якого елемента структури;
  - в подальшому виконати операцію зміни значення вказаного елемента структури;
- назва семафору співпадає з вашим прізвищем транслітерацією;
- права доступу при відкритті семафору = можна читати, писати та виконувати власнику та групі власника;
- функції керування семафором;
- функції затримки для імітації паралельної роботи потоків.

2.4.2 Скомпілювати програму та перевірити її роботу. Зробити висновок щодо наявності або відсутності змагань потоків.

2.5 Аналіз наявності змагань у міжпотоковій взаємодії на основі м’ютексу

2.5.1 Для проведення експериментів над механізмами взаємного виключення декількох потоків під час паралельного доступу до структурної змінної з використанням м’ютексу створити *C*-програму з назвою за шаблоном «*db_mutex_ваше_прізвище.c*»:

- опис структурної змінної, який відповідає структурі реляційної таблиці;
- налаштування роботи двох потоків через окремі функції з назвою *T1,T2*;
- кожна функція повинна:
  - спочатку виконати операцію читання значення будь-якого елемента структури;
  - в подальшому виконати операцію зміни значення вказаного елемента структури;
- функції керування м’ютексом;
- функції затримки для імітації паралельної роботи потоків.

2.5.2 Скомпілювати програму та перевірити її роботу. Зробити висновок щодо наявності або відсутності змагань потоків.

2.6 Підготовка процесу *Code Review* для надання рішень завдань лабораторної роботи на перевірку викладачем

Примітка: Рішення на завдання 2.1-2.2 можуть бути надані під час *Online*-заняття для отримання відповідних балів. За межами *Online*-заняття виконуються всі рішення.

На веб-сервісі *GitHub* зафіксувати зміни у файлі *README.md*.

Скопіювати файли, які було створено у попередніх розділах завдань, в каталог «*Laboratory-work-13*» *Git*-репозиторію.

Оновити *Git*-репозиторій змінами нової гілки «*Laboratory-work-13*» з *GitHub*-репозиторію.

Оновити *GitHub*-репозиторій змінами нової гілки «*Laboratory-work-13*» *Git*-репозиторію. Виконати запит *Pull Request*.

Примітка: Увага! Не натискайте кнопку «Merge pull request»!

Це повинен зробити лише рецензент, який є вашим викладачем!

Може бути виконано два запити *Pull Request*: під час *Online*-заняття та за межами *Online*-заняття.

Якщо запит *Pull Request* було зроблено під час *Online*-заняття, тоді рецензент-викладач перегляне рішення, надасть оцінку та закриє *Pull Request* без операції *Merge*.

Коли буде зроблено остаточний запит *Pull Request*, тоді рецензент-викладач перегляне ваше рішення та виконає злиття нової гілки та основної гілки через операцію *Merge*. Якщо рецензент знайде помилки, він повідомить про це у коментарях, які з’являться на сторінці *Pull request.*

2.5 Оцінка результатів виконання завдань лабораторної роботи

Оцінка	Умови
+2 бали	під час *Online*-заняття виконано правильні рішення завдань 2.1-2.2 або на наступному *Online*-занятті чи на консультації отримано правильну відповідь на два запитання, які стосуються рішень
+2 бали	1) всі рішення роботи відповідають завданням 2) *Pull Request* представлено не пізніше найближчої консультації після офіційного заняття із захисту лабораторної роботи
-0.5 балів за кожну омилку	в рішенні є помилка, про яку вказано в *Code Review*
-1 бал	*Pull Request* представлено пізніше часу завершення найближчої консультації після офіційного заняття із захисту лабораторної роботи за кожний тиждень запізнення
+2 бали	на наступному *Online*-занятті або на консультації отримано правильну відповідь на два запитання, які стосуються рішень

Література

1. Блажко О.А. Відео-запис лекції «Програмування безпечної міжпроцесної та міжпотокової взаємодії в *Unix*-подібних ОС». URL: https://youtu.be/5ZEFa3jb3UM

Контрольні запитання

1. За яких умов виникає ситуація змагання?
2. Що таке критична область або секція?
3. Наведіть приклади двох умов уникання виникнення ситуації змагань?
4. Що таке сигнальна змінна?
5. Що таке циклічне блокування?
6. В чому перевага алгоритма Петерсона для взаємного виключення потоків у порівнянні з алгоритмом «Строге чергування»?
7. Що таке семафор?
8. Чому атомарність процесу-транзакції гарантує відсутність змагань?
9. Що таке двійковий семафор?
10. Що таке м’ютекс?
11. Чим двійковий семафор відрізняється від м’ютекса?
12. Що таке ф’ютекс?