HCMC University Of Technology
Faculty of Computer Science & Engineering

Lab 5 Synchronization
Course: Operating Systems

April 16, 2021

Goal: This lab helps student to practice with the synchronization in OS, and understand the reason why we need the synchronization.

Content In detail, this lab requires student practice with examples using synchronization techniques to solve the problem called **race condition**. The synchronization is performed on Thread, including the following techniques:

- Mutex
- Condition variable
- Semaphore

Result After doing this lab, student can understand the definition of synchronization and write a program which allows to solve the race condition using the techniques above.

Requirement Student need to review the theory of synchronization.

1 Problem

Let us imagine a simple example where two threads wish to update a global shared variable.

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

static volatile int counter = 0;

void *mythread(void *arg){
        printf("%s: begin\n", (char *) arg); int i;
        for (i = 0; i < 1e7; i++) {
                counter = counter + 1;
        }
        printf("%s: done\n", (char *) arg);
        return NULL;
}

int main(int argc, char *argv[])
{
        pthread_t p1, p2;
        printf("main: begin (counter = %d)\n", counter);
        pthread_create(&p1, NULL, mythread, "A");
        pthread_create(&p2, NULL, mythread, "B");

        // join waits for the threads to finish
        pthread_join(p1, NULL);
        pthread_join(p2, NULL);
        printf("main: done with both (counter = %d)\n", counter);
        return 0;
}
```

Finally, and most importantly, we can now look at what each worker is trying to do: add a number to the shared variable counter, and do so 10 million times (1e7) in a loop. Thus, the desired final result is: 20,000,000.
We now compile and run the program, to see how it behaves.

2 Synchronization

What we have demonstrated above (in section Problem) is called a **race condition**: the results depend on the timing execution of the code. With some bad luck (i.e., context switches that occur at untimely points in the execution), we get the wrong result. In fact, we may get a different result each time; thus, instead of a nice **deterministic** computation (which we are used to from computers), we call this result **indeterminate**, where it is not known what the output will be and it is indeed likely to be different across runs.

Because multiple threads executing this code can result in a **race condition**, we call this code a **critical section**. A **critical section** is a piece of code that accesses a shared variable (or more generally, a shared resource) and must not be concurrently executed by more than one thread.

What we really want for this code is what we call mutual exclusion. This property guarantees that if one thread is executing within the critical section, the others will be prevented from doing so.

3 Synchronization with Thread

3.1 Thread API

In the previous lab, we have studied the functions that allow to work with Thread.

- Thread creation

```
#include <pthread.h>
int pthread_create( pthread_t * thread,
                    const pthread_attr_t * attr,
                    void * (*start_routine)(void*), void * arg);
```

- Thread completion

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

3.2 Locks

The POSIX threads library provides mutual exclusion to protect the critical section via **locks**. The most basic pair of routines to use for this purpose is provided by this pair of routines:

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

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

Practice: with the example in Section Problem, we can use this method to solve the problem named **race condition**.

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

```
#include <pthread.h>

static volatile int counter = 0;
pthread_mutex_t lock;

void *mythread(void *arg){
        printf("%s\n", (char *)arg);

        pthread_mutex_lock(&lock);
        int i;
        for(i = 0; i < 1e7; i++)
                counter = counter + 1;
        pthread_mutex_unlock(&lock);

        printf("%s: done\n", (char *)arg);
        return NULL;
}

int main(int argc, char **argv)
{
        pthread_t p1, p2;
        int rc;
        pthread_mutex_init(&lock, NULL);
        printf("main: begin (counter - %d)\n", counter);

        rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
        rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);

        rc = pthread_join(p1, NULL); assert(rc == 0);
        rc = pthread_join(p2, NULL); assert(rc == 0);

        printf("main: finish with both (counter - %d)\n", counter);
        return 0;
}
```

Student implement the function above, then compile and execute it. Give the discussion about the method using **locks**.

3.3 Semaphore

Binary semaphore A binary semaphore can only be 0 or 1. Binary semaphores are most often used to implement a lock that allows only a single thread into a critical section. The semaphore is initially given the value 1 and when a thread approaches the critical region, it waits on the semaphore to decrease the value and "take out" the lock, then signals the semaphore at the end of the critical region to release the lock.

Binary Semaphore Example The canonical use of a semaphore is a lock associated with some resource so that only one thread at a time has access to the resource. In the example below, we have one piece of global data, the number of tickets remaining to sell, that we want to coordinate the access by multiple threads. In this case, a binary semaphore serves as a lock to guarantee that at most one thread is examining or changing the value of the variable at any given time.

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

#define NUM_TICKETS      35
#define NUM_SELLERS      4
#define true 1
#define false 0

static int numTickets = NUM_TICKETS;
static sem_t ticketLock;

void * sellTicket(void *arg);

int main(int argc, char **argv)
{
        int i;
        int tid[NUM_SELLERS];
        pthread_t sellers[NUM_SELLERS];

        sem_init(&ticketLock, 0, 1);
        for (i = 0; i < NUM_SELLERS; i++) {
           tid[i] = i;
           pthread_create(&sellers[i], NULL, sellTicket, (void *) tid[i]);
        }

        for(i = 0; i < NUM_SELLERS; i++)
                pthread_join(sellers[i], NULL);

        sem_destroy(&ticketLock);
        pthread_exit(NULL);
        return 0;
}

```

```c
void *sellTicket(void *arg){
        int done = false;
        int numSoldByThisThread = 0;
        int tid = (int) arg;
        while(!done){
//                sleep(1);
                sem_wait(&ticketLock);
                if(numTickets == 0)
                   done = true;
                else{
                   numTickets--;
                   numSoldByThisThread++;
                   printf("Thread %d sold one (%d left)\n", tid, numTickets);
                }
                sem_post(&ticketLock);
                sleep(1);
        }
        printf("Thread %d sold %d tickets\n", tid, numSoldByThisThread);
        pthread_exit(NULL);
}
```

General semaphores A general semaphore can take on any non-negative value. General semaphores are used for "counting" tasks such as creating a critical region that allows a specified number of threads to enter. For example, if you want at most four threads to be able to enter a section, you could protect it with a semaphore and initialize that semaphore to four. The first four threads will be able to decrease the semaphore and enter the region, but at that point, the semaphore will be zero and any other threads will block outside the critical region until one of the current threads leaves and signals the semaphore.

Generalized semaphore example The next synchronization problem we will confront in this chapter is known as the producer/consumer problem, or sometimes as the bounded buffer problem, which was first posed by Dijkstra [D72]. Imagine one or more producer threads and one or more consumer threads. Producers produce data items and wish to place them in a buffer; consumers grab data items out of the buffer consume them in some way.

This arrangement occurs in many real systems. For example, in a multi-threaded web server, a producer puts HTTP requests into a work queue (i.e., the bounded buffer); consumer threads take requests out of this queue and process them.

To solve the problem above, we use two semaphores, `empty` and `full`, which the threads will use to indicate when a buffer entry has been emptied or filled, respectively. Let's try with the first attempt below:

```c
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>

#define MAX_ITEMS 1
#define THREADS 1         // 1 producer and 1 consumer
#define LOOPS 2*MAX_ITEMS        // variable

// Initiate shared buffer
int buffer[MAX_ITEMS];
int fill = 0;
int use = 0;

sem_t empty;
sem_t full;

void put(int value);    // put data into buffer
int get();       // get data from buffer

void *producer(void *arg) {
        int i;
        int tid = (int) arg;
        for (i = 0; i < LOOPS; i++) {
                sem_wait(&empty);        // line P1
                put(i);                  // line P2
                printf("Producer %d put data %d\n", tid, i);
                sleep(1);
                sem_post(&full);         // line P3
        }
        pthread_exit(NULL);
}

void *consumer(void *arg) {
        int i, tmp = 0;
        int tid = (int) arg;
        while (tmp != -1) {
                sem_wait(&full);         // line C1
                tmp = get();             // line C2
                printf("Consumer %d get data %d\n", tid, tmp);
                sleep(1);
                sem_post(&empty);        // line C3
        }
        pthread_exit(NULL);
}

int main(int argc, char **argv){
        int i, j;
        int tid[THREADS];
        pthread_t producers[THREADS];
        pthread_t consumers[THREADS];

        sem_init(&empty, 0, MAX_ITEMS);
        sem_init(&full, 0, 0);

        for(i = 0; i < THREADS; i++){
            tid[i] = i;
            // Create producer thread
            pthread_create(&producers[i], NULL, producer, (void *) tid[i]);

            // Create consumer thread
            pthread_create(&consumers[i], NULL, consumer, (void *) tid[i]);
        }

        for(i = 0; i < THREADS; i++){
            pthread_join(producers[i], NULL);
            pthread_join(consumers[i], NULL);
        }

        sem_destroy(&full);
        sem_destroy(&empty);

        return 0;
}

void put(int value) {
        buffer[fill] = value; // line f1
        fill = (fill + 1) % MAX_ITEMS; // line f2
}

int get(){
        int tmp = buffer[use]; // line g1
        use = (use + 1) % MAX_ITEMS; // line g2
        return tmp;
}
```

Some problems issued:

- In this example, the producer first waits for a buffer to become empty in order to put data into it, and the consumer similarly waits for a buffer to become filled before using it. Let us first imagine that MAX_ITEMS=1 (there is only one buffer in the array), and see if this works.

- You can try this same example with more threads (e.g., multiple producers, and multiple consumers). It should still work.

- Let us now imagine that MAX_ITEMS is greater than 1 (say MAX_ITEMS = 10). For this example, let us assume that there are multiple producers and multiple consumers. We now have a problem. Do you see where it occurs?

A Solution: Adding Mutual Exclusion
The filling of a buffer and incrementing of the index into the buffer is a critical section, and thus must be guarded carefully. Now we have added some locks around the entire `put()/get()` parts of the code, as indicated by the `NEW LINE` comments. That seems like the right idea, but it also doesn't work. Why? Deadlock.

- Why does deadlock occur?

Avoiding Deadlock

```
...
#define
#define MAX_ITEMS 10
#define THREADS 2          // 2 producers and 2 consumers
#define LOOPS 2*MAX_ITEMS          // variable

...

sem_t empty;
sem_t full;
sem_t lock;

...

void *producer(void *arg) {
        int i;
        int tid = (int) arg;
        for (i = 0; i < LOOPS; i++) {
                sem_wait(&lock);          // line P0          (NEW LINE)
                sem_wait(&empty);         // line P1
```

```
            put(i);                           // line P2
            printf("Producer %d put data %d\n", tid, i);
            sleep(1);
            sem_post(&full);          // line P3
            sem_post(&lock); // line P4 (NEW LINE)
        }
        pthread_exit(NULL);
}

void *consumer(void *arg) {
        int i, tmp = 0;
        int tid = (int) arg;
        while (tmp != -1) {
            sem_wait(&lock);          // line C0 (NEW LINE)
            sem_wait(&full);          // line C1
            tmp = get();              // line C2
            printf("Consumer %d get data %d\n", tid, tmp);
            sleep(2);
            sem_post(&empty);         // line C3
            sem_post(&lock);          // line C4 (NEW LINE)
        }
        pthread_exit(NULL);
}

int main(int argc, char **argv){

        ...
        sem_init(&empty, 0, MAX_ITEMS);
        sem_init(&full, 0, 0);
        sem_init(&lock, 0, 1);
        ...
}
```

Imagine two threads, one producer and one consumer. The consumer gets to run first. It acquires the `lock` (`line C0`), and then calls `sem_wait()` on the `full` semaphore (`line C1`); because there is no data yet, this call causes the consumer to block and thus yield the CPU; importantly, though, the consumer still holds the lock.

A producer then runs. It has data to produce and if it were able to run, it would be able to wake the consumer thread and all would be good. Unfortunately, the first thing it does is call `sem_wait()` on the binary `lock` semaphore (`line P0`). The lock is already held. Hence, the producer is now stuck waiting too.

A Working Solution

To solve this problem, we simply must reduce the scope of the `lock`. We simply move the `lock` acquire and release to be just around the critical section; the `full` and `empty` wait and signal code is left outside.
Let's try it.

4 Condition variables

The other major component of any threads library, and certainly the case with POSIX threads, is the presence of a condition variable. Condition variables are useful when some kind of signaling must take place between threads, if one thread is waiting for another to do something before it can continue. Two primary routines are used by programs wishing to interact in this way:

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

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

To use a condition variable, one has to in addition have a lock that is associated with this condition. When calling either of the above routines, this lock should be held.

The first routine, `pthread_cond_wait()`, puts the calling thread to sleep, and thus waits for some other thread to signal it, usually when something in the program has changed that the now-sleeping thread might care about.

Practice: to illustrate the situation using **condition variables** with Thread. We write a program name *cond_usg.c* which creates 3 threads, where 2 threads is used to increase a global variable named **count**, the last thread is used to cause the condition, it has to wait the signal from other process to continue.

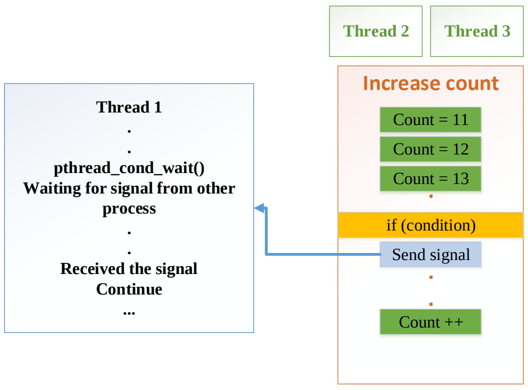

Figure 4.1: Condition variable.

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

#define NUM_THREADS 3
#define TCOUNT 100
#define COUNT_LIMIT 20

int count = 10;
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *tid){
int i;
long my_id = (long) tid;
for(i = 0; i < TCOUNT; i++){
pthread_mutex_lock(&count_mutex);
count++;
if(count == COUNT_LIMIT){
printf("inc_count(): thread %ld, count = %d,
threshold reached.\n", my_id, count);
pthread_cond_signal(&count_threshold_cv);
printf("Just sent signal \n");
}

printf("inc_count(): thread %ld, count = %d,
uncloking mutex\n", my_id, count);
pthread_mutex_unlock(&count_mutex);
sleep(1);
}
pthread_exit(NULL);
}

void *watch_count(void *tid){
long my_id = (long) tid;
printf("Starting watch_count(): thread %ld\n", my_id);

pthread_mutex_lock(&count_mutex);
while(count < COUNT_LIMIT){
printf("watch_count(): thread %ld, count = %d,
waiting...\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 count value...\n", my_id);
count += 80;
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);
pthread_exit(NULL);
}

int main(int argc, char **argv)
{
int i, rc;
pthread_t p1, p2, p3;
long t1 = 1, t2 = 2, t3 = 3;
pthread_attr_t attr;

printf("main: begin\n");
pthread_mutex_init(&count_mutex, NULL);
pthread_cond_init(&count_threshold_cv, NULL);

thread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

pthread_create(&p1, &attr, watch_count, (void *)t1);
pthread_create(&p2, &attr, inc_count, (void *)t2);
pthread_create(&p3, &attr, inc_count, (void *)t3);

rc = pthread_join(p1, NULL); assert(rc == 0);
rc = pthread_join(p2, NULL); assert(rc == 0);
rc = pthread_join(p3, NULL); assert(rc == 0);

printf("main: finish, final count = %d\n", count);

pthread_attr_destroy(&attr);
pthread_mutex_destroy(&count_mutex);
pthread_cond_destroy(&count_threshold_cv);
pthread_exit(NULL);

return 0;
```

```
}
```

5 Exercise (Required)

Problem 1 (3 points): Race conditions are possible in many computer systems. Consider a banking system that maintains an account balance with two functions: `deposit` (amount) and `withdraw` (amount). These two functions are passed the amount that is to be deposited or withdrawn from the bank account balance. Assume that a husband and wife share a bank account. Concurrently, the husband calls the `withdraw()` function and the wife calls `deposit()`. Write a short essay listing possible outcomes we could get and pointing out in which situations those outcomes are produced. Also, propose methods that the bank could apply to avoid unexpected results.

Problem 1 (2 points) Write a new program *nosynch.c* by copying the program *cond_ usg.c* (Section 4) and then removing all entry and exit sections. Write your remarks about the displayed outputs when executing these two programs *nosynch.c* and *cond_ usg.c.*

Problem 2 (5 points): In the Lab 3, we wrote a simple multi-thread program for calculating the value of pi using Monte-Carlo method. In this exercise, we also calculate pi using the same method but with a different implementation. We create a shared (global) count variable and let worker threads update on this variable in each of their iteration instead of on their own local count variable. To make sure the result is correct, remember to avoid race conditions on updates to the shared global variable by using mutex locks. Compare the performance of this approach with the previous one in Lab 5.

Put your answers to the questions in **Problem 1** and **Problem 2** to a single PDF file name `ex.pdf`. In the **Problem 3**, reuse the file you have written for Lab 3 and only modify parts you think it need to be. You must submit your ZIP file to BKEL.