CS2106 Introduction to Operating Systems
Lab 3 –Synchronization Mechanisms

1. Introduction

In this lab we will look at synchronization mechanisms to help parallel programs operate correctly. In the first part we will look at the idea of lock variables, and of how to use semaphores. In the second part we will implement barriers using semaphores, and in the third part we will use barriers to help find the smallest and largest elements of a moderately sized array.

As an added challenge we will be using the synchronization mechanisms with processes rather than threads, and we will see how to correctly share mechanisms like semaphores between processes.

Deadline and Submission

a. You can do this lab alone or with a partner.
b. Ensure that both of you fill in your names, student IDs and group numbers in the answer book AxxxxxxY.docx.
c. Rename AxxxxxxY.docx to the student ID of the submitter before submitting.
d. Zip up the following files into a single ZIP file called AxxxxxxY.zip, where AxxxxxxY is your student ID:
    - Your answer book, properly renamed.
    - Your barrier.c and barrier.h
    - Your bigsmall-par.c

e. Upload your ZIP file, properly named, to Canvas by **11.59 pm, Sunday 30 March 2025.**
f. The folder has been set to close very shortly after midnight on Monday 31 March 2025. **Once the folder is closed no submissions will be accepted**.
g. This lab is worth 15 marks in total (11 marks for the report, 2 marks each for the two demos).

2. Activities

Part 1. Using Lock Variables and Semaphores

All files needed for this part can be found in the part1 directory. In this part we will look at two ways to do synchronization: Lock variables and semaphores. We begin by opening up lab3p1.c:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/shm.h>
#include <sys/wait.h>

#define NUM_PROCESSES 5

int main() {

    int i, j, pid;

    for(i=0; i<NUM_PROCESSES; i++)
    {
        if((pid = fork()) == 0) {
            break;
        }
    }

    if(pid == 0) {
        printf("I am child %d\n", i);

        for(j = i*10; j<i*10 + 10; j++){
            printf("%d ", j);
            fflush(stdout);
            usleep(250000);
        }

        printf("\n\n");
    }
    else {
        for(i=0; i<NUM_PROCESSES; i++)
            wait(NULL);
    }

}
```

This is a simple program that spawns 5 processes, which tries to produce the following output:

```
I am child 0
0 1 2 3 4 5 6 7 8 9

I am child 1
10 11 12 13 14 15 16 17 18 19

I am child 2
20 21 22 23 24 25 26 27 28 29

I am child 3
30 31 32 33 34 35 36 37 38 39

I am child 4
40 41 42 43 44 45 46 47 48 49
```

We compile and run the program using:

```
gcc lab3p1.c –o lab3p1
./lab3p1
```

We see an output similar to this:

```
I am child 0
0 I am child 1
10 I am child 2
20 I am child 3
30 I am child 4
40 1 11 21 41 31 12 2 22 32 42 3 13 23 43 33 24 14 4 44 34 5 15 25 35 45 26 16 6 46 36 17 27 47 7 37 18 28 8 48 38 29 19 9 49 39
```

Question 1.1 (1 mark)

Explain why all the "<some number> I am child X" (X is some number from 0 to 4) appears before most of the numbers. What does this suggest about the time quantum? Additionally, the numbers printed by each child are interleaved, as shown in the output. Explain why this interleaving happens and how the operating system's scheduling affects this behavior.

a. Using Lock Variables

We will now explore a mechanism called "lock variables" to try to synchronize the different processes. Lock variables are very simple and the algorithm below illustrates how they work:

1. Create a new shared variable "lock". Set it to 1
2. Spawn all processes
3. For each process:
   3.1 if lock is 0, goto 3.1 (Busy wait until lock is 1)
   3.2 Set lock to 0 (Note: we can only get here when lock is 1 and the loop at 3.1 exits)
   3.2 Execute my statements (Note: we can only get here once lock is 1

Part of the code for lab3p1-lock.c is shown below, with the relevant lock code circled:

```c
// We create a new shared variable for our lock
int *lock;
int shmid;

shmid = shmget(IPC_PRIVATE, sizeof(int), IPC_CREAT | 0600);
lock = shmat(shmid, NULL, 0);

// If lock is 1, we get to run our code.
lock[0] = 1;

for(i=0; i<NUM_PROCESSES; i++)
{
    if((pid = fork()) == 0) {
        break;
    }
}

if(pid == 0) {

    // Spin lock until lock is 1
    while(lock[0] == 0);

    // Claim to lock
    lock[0] = 0;
    printf("I am child %d\n", i);

    for(j = i*10; j<i*10 + 10; j++){
        printf("%d ", j);
        fflush(stdout);
        usleep(250000);
    }

    printf("\n\n");

    // Release the lock
    lock[0] = 1;
}
```

Note that while the lock variable doesn't control which process gets to run first. All we want at this point is to get it to do something like the following:

```
I am child 0
0 1 2 3 4 5 6 7 8 9

I am child 3
31 32 33 34 35 36 37 38 39
etc.
```

That is, we want one process to complete running before the next one starts.

Compile and run lab3p1-lock.c by using:

```
gcc lab3p1-lock.c -o lab3p1lock
./lab3p1lock
```

Question 1.2 (1 mark)

If we look at the result of our program, it's clear we have not achieved our objective. Explain why the lock variable failed to coordinate the processes.

b. Using Turn Variables

You've already seen the concept of "turn variables" in Tutorial 4 Question 4. To summarize:

1. Create a shared variable called "turn". Set turn to 0.
2. For each process i from 0 to n-1:
   2.1 while(turn != i); // Do nothing while it's not our turn.
   2.2 Now it is my turn. Execute my code.
   2.3 turn = turn + 1; // Pass turn to next process

You are given a file called lab3p1-shm.c. Modify this program to implement turn variables.

Question 1.3 (1 mark)

Modify the code to implement turn variables and explain your changes. Why might turn variables be less efficient on multi-core systems?

(No credit will be given without explanations. Marks will be deducted if you forget to detach or free your shared memory).

c. Using Semaphores

We will now make an attempt to use semaphores. Open up sema-wrong.c (yes, the solution is indeed wrong) and you will see a straightforward but naive attempt to use semaphores to coordinate between processes:

```
int main() {
    sem_t sem;
    int pid;

    sem_init(&sem, 1, 0);

    if((pid = fork()) != 0) {
        printf("Parent! Making my child wait for 1 second.\n");
        sleep(1);
        sem_post(&sem);
    }
    else
    {
        sem_wait(&sem);
        printf("Child! Waited 1 second for parent.\n");
    }

    if(pid != 0)
    {
        wait(NULL);
        sem_destroy(&sem);
    }
}
```

We compile and run using:

```
gcc sema-wrong.c -o sema-wrong -lpthread
./sema-wrong
```

(Note: The `-lpthread` is important to bring in the semaphore library)

Question 1.4 (1 mark)

Explain each parameter of sem_init, and what sem_wait and sem_post do. Note: If you just say "sem_wait" waits for a semaphore and "sem_post" posts to a semaphore, you will not receive any credit. Your answer must show understanding of what semaphores are and how they work.

Question 1.5 (1 mark)

There is no shared memory between the parent and child process. Explain why this causes the program to hang.

We will now see how to correctly share semaphores between processes. Open up sema-right.c and you will see:

```
int shmid, pid;
sem_t *sem;

shmid = shmget(IPC_PRIVATE, sizeof(sem_t), IPC_CREAT | 0600);
sem = (sem_t *) shmat(shmid, NULL, 0);

sem_init(sem, 1, 0);

if((pid = fork()) != 0) {
    printf("Parent!. Making my child wait for 1 second.\n");
    sleep(1);
    sem_post(sem);
    wait(NULL);
    sem_destroy(sem);
    shmctl(shmid, IPC_RMID, 0);
}
else
{
    sem_wait(sem);
    printf("Child! Waited 1 second for parent.\n");
}
```

Notice what we've done here:

a. Our semaphore sem is of type sem_t * instead of sem_t.
b. We create a segment of shared memory using shmget of size sem_t.
c. We attach the shared memory to sem using shmat.
d. From then on we treat sem as an ordinary semaphore. Note that now we pass in sem instead of &sem to sem_wait and sem_post, since sem is now of type sem_t * instead of sem_t.
e. We destroy the semaphore, then detach and release the shared memory once it's no longer needed.

We compile and run as usual:

```
gcc sema-right.c -o sema-right -lpthread
./sema-right
```

We now see that the program runs correctly; the parent pauses for a second before releasing the child and we see:

```
Parent!. Making my child wait for 1 second.
Child! Waited 1 second for parent.
```

You are now given lab3p1-sem.c, which at this time is identical to lab3p1.c. Using what you've learnt about semaphores, modify lab3p1-sem.c to produce this output, with the correct ordering of processes from 0 to 4 (Hint: Create an array of semaphores):

```
I am child 0
0 1 2 3 4 5 6 7 8 9

I am child 1
10 11 12 13 14 15 16 17 18 19

I am child 2
20 21 22 23 24 25 26 27 28 29

I am child 3
30 31 32 33 34 35 36 37 38 39

I am child 4
40 41 42 43 44 45 46 47 48 49
```

Question 1.6 (1 mark)

Briefly explain how your program synchronizes the processes to produce the output above. There is no need to cut-and-paste code.

What challenges might arise if more processes are added, and how would you modify the semaphore-based synchronization to handle an arbitrary number of processes efficiently?

DEMO 1 (2 marks)

Compile your lab3p1-sem.c and demonstrate to your TA that it works correctly.

Part 2. Creating Barriers with Semaphores

In this part we will create a barrier using semaphores. Recall that a barrier is a structure that is called by an expected number of processes. All processes except the last will block at the barrier, until the last process calls the barrier, after which all processes are released. All files needed can be found in the part2 directory.

When we create barriers using semaphores, we are relying on the following property of semaphores:

1. We start with sem=0
2. If *n* processes call wait(sem), all *n* processes will block.
3. If another process calls signal(sem), ONE of the *n* processes will be picked to unblock.
4. If each of the *n* processes also calls signal(sem) after being unblocked, eventually everyone will be unblocked.

Using the principle above and the idea we learn in the Tutorial, we can design our barrier using the following pseudo-code (note: This is incomplete and just gives you an idea of what to do). However, this barrier can only be used one time, and we want to make it reusable.

```
int nproc = 0, count = 0;
sem_t barrier;

// Initializes the barrier
void init_barrier(int num_proc) {
        nproc = num_proc;
        count = 0;
        Initialize barrier to 0.
}
```

```
// Every process calls this to "reach" the barrier. However, this is a
one-time barrier. How to make it reusable?

void reach_barrier() {
        count++;
        if(count == nproc) {
                // Release the semaphore
                signal(barrier);
        }
        else {
                // We are not the last process. So we wait at the
                // semaphore until we are freed.
                wait(barrier);

                // Now that we are freed, we free the next process
                signal(barrier);
        }
}
```

You are given three files barrier.c, barrier.h and test_barrier.c. The barrier.c file contains three functions:

init_barrier: Takes one argument – The number of processes that will call the barrier. This function should create any shared memory required, as well as the semaphores you need. Note that in the algorithm above count is a shared variable that is updated by multiple processes and can thus create race conditions. You will need one more semaphore to act as a mutex to protect this variable.

reach_barrier: Does not take any arguments. Follows the algorithm above and make it reusable.

destroy_barrier: If the caller is a parent process, destroy all semaphores and detach and release all shared memory.

The barrier.h file contains prototypes for barrier.c, and test_barrier.c contains a test program that basically creates 6 children that will sleep a random amount of time of up to 1 second, then call reach_barrier for a number of **NUM_TESTS** times. When all children reach the barrier, the parent will print "All the children have returned." and exit. **init_barrier()** and **destroy_barrier()** will only be called once in **test_barrier.c** program so make your barrier **reusable**.

The children too will output "Child X has slept for Y seconds and has now reached the barrier".

After completing your barrier.c, you can compile and run using:

```
gcc barrier.c test_barrier.c –o test_barrier –lpthread
./test_barrier
```

If your barrier works correctly, you should see an output similar to this (sleep times will differ):

```
nknight@legion ~/soln copy/part2 ./test_barrier
**      Test #1      **
**Parent waiting for children**

        Child 5 slept for 0.15 seconds and has now reached the barrier
        Child 2 slept for 0.22 seconds and has now reached the barrier
        Child 3 slept for 0.39 seconds and has now reached the barrier
        Child 4 slept for 0.51 seconds and has now reached the barrier
        Child 1 slept for 0.66 seconds and has now reached the barrier
        Child 0 slept for 0.98 seconds and has now reached the barrier

**All the children have returned**

**      Test #2      **
**Parent waiting for children**

        Child 5 slept for 0.15 seconds and has now reached the barrier
        Child 2 slept for 0.22 seconds and has now reached the barrier
        Child 3 slept for 0.39 seconds and has now reached the barrier
        Child 4 slept for 0.51 seconds and has now reached the barrier
        Child 1 slept for 0.66 seconds and has now reached the barrier
        Child 0 slept for 0.98 seconds and has now reached the barrier

**All the children have returned**

**      Test #3      **
**Parent waiting for children**

        Child 5 slept for 0.15 seconds and has now reached the barrier
        Child 2 slept for 0.22 seconds and has now reached the barrier
        Child 3 slept for 0.39 seconds and has now reached the barrier
        Child 4 slept for 0.51 seconds and has now reached the barrier
        Child 1 slept for 0.66 seconds and has now reached the barrier
        Child 0 slept for 0.98 seconds and has now reached the barrier

**All the children have returned**

**      Test #4      **
**Parent waiting for children**

        Child 5 slept for 0.15 seconds and has now reached the barrier
        Child 2 slept for 0.22 seconds and has now reached the barrier
        Child 3 slept for 0.39 seconds and has now reached the barrier
        Child 4 slept for 0.51 seconds and has now reached the barrier
        Child 1 slept for 0.66 seconds and has now reached the barrier
        Child 0 slept for 0.98 seconds and has now reached the barrier

**All the children have returned**
```

Question 2.1 (1 mark)

Explain what values should the semaphores be initialized to, and why? Cut and paste your code for init_barrier and describe the steps your init_barrier function takes to set up the synchronization.

Question 2.2 (1 mark)

Cut and paste your code for reach_barrier and explain it.

DEMO 2 (2 marks)
Compile and execute test_barrier and show your TA that it works.

Part 3. Finding Largest and Smallest Element in Parallel

In this part we will search through a moderately sized 2,000,000 element array to find the largest and smallest elements. You are given two programs. Both generate the same 2,000,000 element array of pseudo-random numbers with the same starting seed. Most of what you need can be found in the part3 directory, although you will need to copy over the barrier.* files from the previous part to complete your program.

a. The sequential version bigsmall.c. Compile and run this program:

```
gcc bigsmall.c -o bigsmall
./bigsmall
```

You will see:

```
Number of items: 2000000
Smallest element is 752
Largest element is 2147474537
Time taken is 0.01 seconds
```

The random number generator is seeded to 24601, so this program will always produce the same smallest and largest elements. This helps you to check if your program is working correctly.

b. The (incorrectly implemented) parallel version, bigsmall-par.c. This program splits the 2,000,000 element array between 8 processes (250,000 integers per process). There are two arrays “largest” and “smallest” where each process will store its results (e.g. process 1 will store to largest[1] and smallest[1], etc.). The parent is supposed to wait for all child processes to finish before looking through “largest” and “smallest” to find the largest and smallest integers amongst all those found by the children, and print those out.

Compile and execute using:

```
gcc bigsmall-par.c
./bigsmall-par.c
```

You will see the following output:

```
Number of items: 2000000
Smallest element is 0
Largest element is 0
Time taken is 0.00s
```

The random number generator is again seeded to 24601, so this output is clearly incorrect.

Now modify bigsmall-par.c to produce the correct answer, subject to the following rules:

a. The program will still distribute the search between 8 processors.
b. You must use barriers to coordinate.
c. You must produce the same smallest and largest element as bigsmall.c.
d. Leave the code that measures the time taken to do the search in its current place in the parent's code.

Answer the following questions:

Question 3.1 (1 mark)

Explain why the original bigsmall-par.c code produced the wrong results.

Question 3.2 (1 mark)

Explain the changes that you made to make it work correctly. You may cut-and-paste SMALL REVELEVANT FRAGMENTS to illustrate the changes that you made. Do not cut-and-paste the entire code here.

Question 3.3 (1 mark)

If you left the code to measure time taken in its original place, you will find that the parallel code shows a faster timing. However, this way of measuring timing may not be fair. Explain why.