# CIS 3110: Operating Systems

# Assignment 4: Synchronizing Threads with POSIX Semaphores

**Due Thursday, April 2, 2020 @ 11:59pm**

## 1. Objective

The objectives of this assignment is to familiarize you with basic process or thread synchronization concepts and primitives, and as well tools that are used to solve process or thread synchronization problems. In this assignment you will write a multi-threaded program that increments a shared counter. You are required to use C and pthreads. You are provided a program **ibadcnt.c**, but it produces unexpected results. You task is to use synchronization techniques, particularly, using Posix Semaphores, to ensure that the program always produce correct results. Also, your new program should accept a command-line parameter to specify how many times each thread increment the shared count.

## 2. Why Semaphores?

Posix semaphores are easy to use

- sem_init
- sem_wait
- sem_post
- sem_getvalue
- sem_destroy

Please refer to the following tutorials on how to use POSIX semaphores in C language

- How to use POSIX semaphores in C language
  https://www.geeksforgeeks.org/use-posix-semaphores-c/
- Example Semaphore
  https://os.itec.kit.edu/downloads/sysarch09-mutualexclusionADD.pdf

Now it is time to take a look at some code that does something a little unexpected. The program **ibadcnt.c** creates two new threads, both of which increment a global variable called cnt exactly NITER, with NITER = 1,000,000. But the program produces unexpected results.

From Programming Assignment #1, you already find out when running the program it gives a different result every time. Quite unexpected! Since cnt starts at 0, and both threads increment it

NITER times, we should see cnt equal to 2*NITER at the end of the program. However, the program doesn't behave as what is expected. And gives a different result every time it runs.

Threads can greatly simplify writing elegant and efficient programs. However, there are problems when multiple threads share a common address space, like the variable cnt in our earlier example.

To understand what might happen, let us analyze this simple piece of code:

```
THREAD 1            THREAD 2
a = data;           b = data;
a++;                b--;
data = a;           data = b;
```

Now if this code is executed serially (for instance, THREAD 1 first and then THREAD 2), there are no problems. However threads execute in an arbitrary order, so consider the following situation with "data = 0" initially:

| **Thread 1** | **Thread 2** | **data** |
|---|---|---|
| a = data; | --- | 0 |
| a = a+1; | --- | 0 |
| --- | b = data;    // 0 | 0 |
| --- | b = b-1; | 0 |
| data = a;    // 1 | --- | 1 |
| --- | data = b;    // -1 | -1 |

So unfortunately, data could end up +1, 0, -1, and there is NO WAY to know which value! It is completely non-deterministic!

The solution to this is to provide functions that will block a thread if another thread is accessing data that it is using.

Pthreads may use semaphores to achieve this.

## 3. Posix semaphores

All POSIX semaphore functions and types are prototyped or defined in semaphore.h. To define a semaphore object, use

```
sem_t sem_name;
```

**To initialize a semaphore, use sem_init:**

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- sem points to a semaphore object to initialize
- pshared is a flag indicating whether or not the semaphore should be shared with fork()ed processes. LinuxThreads does not currently support shared semaphores
- value is an initial value to set the semaphore to

Example of use:
```
sem_init(&sem_name, 0, 10);
```

**To wait on a semaphore, use sem_wait:**
```
int sem_wait(sem_t *sem);
```

Example of use:
```
sem_wait(&sem_name);
```

- If the value of the semaphore is negative, the calling process blocks; one of the blocked processes wakes up when another process calls sem_post.

**To increment the value of a semaphore, use sem_post:**
```
int sem_post(sem_t *sem);
```

Example of use:
```
sem_post(&sem_name);
```

- It increments the value of the semaphore and wakes up a blocked process waiting on the semaphore, if any.

**To find out the value of a semaphore, use**
```
int sem_getvalue(sem_t *sem, int *valp);
```

- gets the current value of sem and places it in the location pointed to by valp

Example of use:
```
int value;
sem_getvalue(&sem_name, &value);
printf("The value of the semaphors is %d\n", value);
```

**To destroy a semaphore, use**
```
int sem_destroy(sem_t *sem);
```

- destroys the semaphore; no threads should be waiting on the semaphore if its destruction is to succeed.

Example of use:
```
sem_destroy(&sem_name);
```

Note that when the semaphore is no longer required, and before the memory in which it is located is deallocated, the semaphore should be destroyed using sem_destroy.

## 4. Using semaphores - a short example

Consider the problem we had before and now let us use semaphores:

Declare the semaphore global (outside of any funcion):

```
sem_t mutex;
```

Initialize the semaphore in the main function:

```
sem_init(&mutex, 0, 1);
```

| Thread 1 | Thread 2 | data |
|---|---|---|
| sem_wait (&mutex); | --- | 0 |
| --- | sem_wait (&mutex); | 0 |
| a = data; | /* blocked */ | 0 |
| a = a+1; | /* blocked */ | 0 |
| data = a; | /* blocked */ | 1 |
| sem_post (&mutex); | /* blocked */ | 1 |
| /* blocked */ | b = data; | 1 |
| /* blocked */ | b = b - 1; | 1 |
| /* blocked */ | data = b; | 0 |
| /* blocked */ | sem_post (&mutex); | 0 |
| **[data is fine. The data race is gone.]** | | |

**Activity 1**.
Use the example above as a guide to fix the program i**badcnt.c**, so that the program always produces the expected output (the value 2*NITER). Make a copy of i**badcnt.c** into **igoodcnt.c** before you modify the code. Please do not edit the section of code that starts from:

//PLEASE DO NOT remove or modify the following code
And end at:
// End of code section

To compile a program that uses pthreads and posix semaphores, use

```
gcc -o xfilename filename.c -lpthread -lrt
```

Also, your new program should accept a command-line parameter to specify how many times each thread increments the shared count. In other words, the value of variable NITER should be entered by the user. Your program should be robust for bad arguments! If the argument is invalid or bad, your program should print a **usage statement**, and then exit graceful indicating an unsuccessful termination[4]. Further, your program should be robust to handle abnormal situations, for example, failing to create a thread.

There are several parts you must pay attention to about **usage statements** [1]

- The usage message: it always starts with the word "usage", followed by the program name and the names of the arguments. Argument names should be descriptive if possible, telling what the arguments refer to, like "NoofTimesEachThreadIncrements" in the example below. Argument names should not contain spaces! Optional arguments are put between square brackets, like "-l" for the Linux command *ls*. Do not use square brackets for non-optional arguments! Always print to stderr, not to stdout, to indicate that the program has been invoked incorrectly.
- The program name: always use argv[0] to refer to the program name rather than writing it out explicitly. This means that if you rename the program (which is common) you won't have to re-write the code.
- Exiting the program: use the exit function, which is defined in the header file <stdlib.h>. Any non-zero argument to exit (e.g. exit(1)) signals an unsuccessful completion of the program (a zero argument to exit (exit(0)) indicates successful completion of the program, but you rarely need to use exit for this). Or, you can simply use EXIT_FAILURE and EXIT_SUCCESS (which are defined in <stdlib.h>) instead of 1 and 0 as arguments to exit.

For example, the following is a code snippet, which prints a usage statement, and then exits the program by indicating an unsuccessful termination

```
fprintf(stderr, "usage: %s NoofTimesEachThreadIncrements\n", argv[0]);
exit(EXIT_FAILURE);
```

**Activity 2.**

**Q1)** In your modified program, do you think it is possible to become deadlocked? In other words, your program is in a situation in which two threads are waiting for each other to increment the counter, and thus neither ever does. Give Resource-Allocation Graph to help in your explanation. Note that in your Resource allocation graph, the thread is represented by a Circle node while the Semaphore or the Resource is represented by a rectangle node.

**Write up your answers to the above question and save them in a file called answers-a4.pdf.**

## 5. How to Test Your Program

Run your modified program many times to see if your program always gets the same output that the counter (or cnt) is equal to 2*NITER at the end of the program.

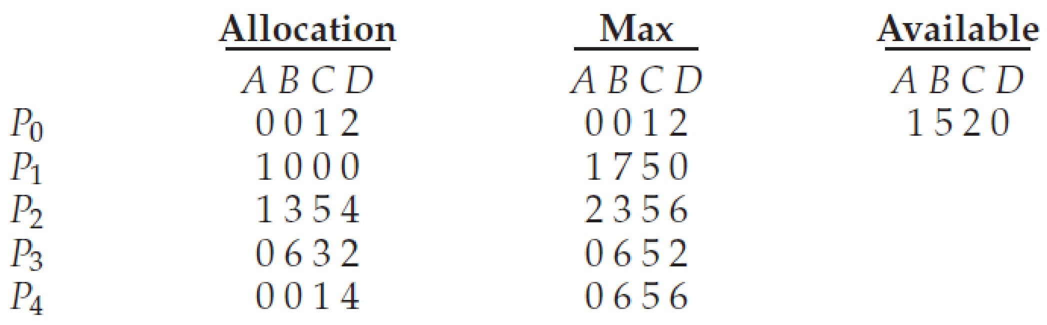
Also, you need to test the following abnormal situations to see whether your program is robust to handle these abnormal situations, including

- Invalid number of arguments
- Invalid number of times which each thread counts, for example, negative counter

## 6. Theoretical questions

In order to further help you better understand and solve the deadlock issue, you need to solve the following theoretical questions. Write up your answers to the following question and add them in the file answers-a4.pdf.

**Activity 3.**
**Q2)** Consider the following snapshot of a system:



| | Allocation | Max | Available |
|---|---|---|---|
| | *A B C D* | *A B C D* | *A B C D* |
| $P_0$ | 0 0 1 2 | 0 0 1 2 | 1 5 2 0 |
| $P_1$ | 1 0 0 0 | 1 7 5 0 | |
| $P_2$ | 1 3 5 4 | 2 3 5 6 | |
| $P_3$ | 0 6 3 2 | 0 6 5 2 | |
| $P_4$ | 0 0 1 4 | 0 6 5 6 | |

Answer the following questions using the banker's algorithm:
a. What is the content of the matrix **Need**?
b. Is the system in a safe state? Please explain why.

## 7. Grading Scheme

| | |
|---|---|
| Code quality (e.g., CODE Style & Documentation) | 0.5 point |
| Activity 1 (Modified code) | 3 points |
| Activity 2 (Q1) | 2 points |
| Activity 3 (Q2) | 3 points |
| Stability and reliability testing (e.g., not have memory leaks or memory violations, proper error handling) | 1.5 points |

## 8. Submission

- If you have any problems in the development of your programs, contact the teaching assistant (TA) assigned to this course.

- You are encouraged to discuss this project with fellow students. However, you are **not** allowed to share code and your brief with any student.
- If your TA is unable to run/test your program, you should present a demo arranged by your TA's request.
- Please only submit the source code files plus any files required to build your program as well as any files requested in the assignment, including
  - the complete source code, igoodcnt.c;
  - the Makefile; and
  - your answers to the above two questions, **Q1** and **Q2** (**answers-a4.pdf**)
- How to name your programming assignment projects: For any assignment, zip all the files required to a zip file with name: CIS3110_<assignment_number>_XXX.zip, where <assignment_number> is the assisgnement number you are solving (e.g., a4 for Programming Assignment 4) and XXX is your University of Guelph's email ID (Central Login ID). This naming convention facilitates the tasks of marking for the instructor and course TAs. It also helps you in organizing your course work. Failure to follow the requirements will result in mark reduction.

  Note: to zip and unzip files in Unix:

  ```
  $zip -r filename.zip files
  $unzip filename.zip
  ```

**Please only submit the C source code files and Makefile file plus any files requested in an assignment. You are required to develop software in C on VM provided by the textbook.**

**Acknowledgements**

**This lab has incorporated the materials developed by Mirela Damian at Villanova University. The copyright of these materials belongs to them.**

References:
[1] How to use POSIX semaphores in C language
https://www.geeksforgeeks.org/use-posix-semaphores-c/
[2] Example Semaphore
https://os.itec.kit.edu/downloads/sysarch09-mutualexclusionADD.pdf
[3] POSIX Semaphores
http://web.cs.iastate.edu/~cs352/notes/Ch6-7.pdf
[4] Overview of POSIX semaphores
https://linux.die.net/man/7/sem_overview