Tutorial 6: **POSIX Threads Part II**

Faculty of Engineering and Applied Science

SOFE 3950U: Operating Systems | CRN: 74171

Due: March 4th, 2024

Group 8

Daniel Amasowomwan [100787640]
daniel.amasowomwan@ontariotechu.net

Stanley Watemi [100648403]
stanley.watemi@ontariotechu.net

Fayomi Toyin [100765921]
oluwatoyin.fayomi@ontariotechu.net

Conceptual Questions

1.) What is fork(), how does it differ from multi-threading (pthreads)?

fork(): A system call that duplicates the caller process to create a new process. The child process is a carbon replica of the parent process but with a unique process ID and parent process ID. Multi-threading (pthreads) enables many threads of execution to run concurrently inside the same process.

2.) What is inter-process communication (IPC)? Describe methods of performing IPC.

Inter-process communication (IPC) is the exchange of data or messages between computer processes. Pipes, sockets, message queues, shared memory, and signals are examples of IPC methods.

3.) Provide an explanation of semaphores, how do they work, how do they differ from mutual exclusion?

In a multi-process or multi-threaded context, semaphores are synchronization primitives that limit access to shared resources. Semaphores are useful for implementing mutual exclusion as well as coordinating the activity of several processes or threads.

4.) Provide an explanation of wait (P) and signal (V).

Wait (P) and signal (V): Wait (P) decrements the value of a semaphore, and the calling process or thread is halted if the result is negative. Signal (V) increases the value of a semaphore and, if present, unblocks one waiting process or thread.

5.) Research the main functions used for semaphores in <semaphore.h> and explain each function.

Semaphore routines include sem_init(), sem_destroy(), sem _ wait(), sem _ post(), sem _ trywait(), and sem _ getvalue() in semaphore.h> ():
sem_init(): initializes a semaphore.
sem_destroy(): destroys a semaphore.
sem_wait(): decrements a semaphore, blocking if necessary.
sem_post(): increments a semaphore, waking up a blocked process if any.
sem_trywait(): decrements a semaphore if it's available, otherwise returns immediately.
sem_getvalue(): gets the value of a semaphore.

Application Questions

All of your programs for this activity can be completed using the template provided, where you fill in the remaining content. A makefile is not necessary, to compile your programs use the following command in the terminal. **If you do not have clang then replace clang with gcc.**

clang -Wall -Wextra -std=c99 <program name>.c -o <program name>

Example:
clang -Wall -Wextra -std=c99 question1.c -o question1
You can then execute and test your program by running it with the following command.
./<program name>

Example:

./question1

Template

```
#define _XOPEN_SOURCE 700 // required for barriers to work

#include <stdlib.h>

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

int main(void)

{

}
```

1. Create a program that does the following.
    - Creates a master process with **two** child processes using **fork()**
    - The master process writes two files **child1.txt** containing the line **child 1** and **child2.txt** containing the line **child 2**
    - Each of the two child processes waits **one second** then reads the contents of their text file and prints the contents.
    - Reading and writing files between process is a simplified method of IPC.

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid1 = fork();
    if (pid1 == 0) {
        FILE *file1 = fopen("child1.txt", "w");
        fprintf(file1, "Child 1 is writing to the file \n");
        fclose(file1);

        sleep(1);
        char c[100];

        FILE *file1_read = fopen("child1.txt", "r");
        fgets(c, sizeof(c), file1_read);
        printf("Child 1 is reading from the file: %s\n", c);
        fclose(file1_read);
        exit(0);
```

```
22          exit(0);
23      } else if (pid1 > 0) {
24          pid_t pid2 = fork();
25          if (pid2 == 0) {
26              FILE *file2 = fopen("child2.txt", "w");
27              fprintf(file2, "Child 2 is writing to the file \n");
28              fclose(file2);
29              sleep(1);
30              char c[100]; // Corrected array declaration
31              FILE *file2_read = fopen("child2.txt", "r");
32              fgets(c, sizeof(c), file2_read);
33              printf("Child 2 is reading from the file: %s\n", c);
34              fclose(file2_read);
35              exit(0);
36          } else if (pid2 > 0) {
37              wait(NULL);
38              wait(NULL);
39              printf("Parent is reading from the files\n");
40              char c[100]; // Corrected array declaration
41              FILE *file1_read = fopen("child1.txt", "r");
42              fgets(c, sizeof(c), file1_read);
43              printf("Child 1 is reading from the file: %s\n", c);
44              fclose(file1_read);
45              FILE *file2_read = fopen("child2.txt", "r");
46              fgets(c, sizeof(c), file2_read);
47              printf("Child 2 is reading from the file: %s\n", c);
48              fclose(file2_read);
49          }
50      }
51
52      return 0;
```

```
stanley@stanley-VirtualBox:~/Tutorial6$ gcc -o question1 question1.c
stanley@stanley-VirtualBox:~/Tutorial6$ ./question1
Child 1 is reading from the file: Child 1 is writing to the file

Child 2 is reading from the file: Child 2 is writing to the file

Parent is reading from the files
Child 1 is reading from the file: Child 1 is writing to the file

Child 2 is reading from the file: Child 2 is writing to the file

stanley@stanley-VirtualBox:~/Tutorial6$
```

2. Create a program that does the following.
    - Creates a parent and child process using **fork().**
    - The child process sleeps for 1 second and prints “Child process”.
    - The parent process instead of immediately executing, **waits** for the child process to terminate using the **wait()** function before printing “Parent process”.
    - The parent process must check the return status of the child process after it has finished waiting.
    - See the following for more information on forking and waiting: http://www.cs.rutgers.edu/~pxk/416/notes/c-tutorials/

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <sys/wait.h>


int main() {
    pid_t pid1 = fork();
    if (pid1==0)
    {
        sleep(1);
        printf("This is the child process \n");
        exit(0);


    }
    else if (pid1>0)
    {
        int status;
        wait( &status);

        if (WIFEXITED(status))
        {
            printf("This is the parent process\n");

        }
        else {
            printf("The child process 1 did not terminate normally\n");
        }

```

```
stanley@stanley-VirtualBox:~/Tutorial6$ gcc -o question2 question2.c
stanley@stanley-VirtualBox:~/Tutorial6$ ./question2
This is the child process
This is the parent process
stanley@stanley-VirtualBox:~/Tutorial6$
```

3. Create a program that does the following.
    - Create a **global** array length five, **moving_sum**, and initialize it to zeros.
    - Prompts a user for fives numbers
    - For each number creates a thread
    - Each thread executes a function **factorial** which takes a struct containing the **number** and **index** of the number entered (0 - 4) and does the following:
        - Calculates the factorial (e.g. 5! = 5x4x3x2x1 = 120, 0! = 1).
        - **Using a semaphore**, gets the previous value in the **moving_sum[index-1]** if the value at that **index is > 0**. If the value is retrieved it is added to the factorial calculated and the sum is added to **moving_sum[index].**
        - Until the value in **moving_sum[index-1]** is > 0, performs an infinite loop, each time it must perform signal and wait to allow other threads access to the critical section.
    - After all threads finish (using **join()**) print the contents of **moving_sum**

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

#define arraySize 5

typedef struct {
    int num;
    int arr_index;
} data;

int moving_sum[arraySize] = {0};
sem_t sem[arraySize] = {0};

int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

void* thread_function(void* arg) {
    data* d = (data*)arg;
    int num = d->num;
    int arr_index = d->arr_index;
    int fact = factorial(num);

    while (1) {
        if (arr_index == 0 || moving_sum[arr_index - 1] != 0) {
            sem_wait(&sem[arr_index]);
            if (arr_index > 0) {
                fact += moving_sum[arr_index - 1];
            }
```

```c
int main() {
    pthread_t threads[arraySize];
    data d[arraySize];
    for (int i = 0; i < arraySize; i++) {
        sem_init(&sem[i], 0, 0);
    }
    for (int i = 0; i < arraySize; i++) {
        d[i].num = i;
        d[i].arr_index = i;
        pthread_create(&threads[i], NULL, thread_function, (void*)&d[i]);
    }
    for (int i = 0; i < arraySize; i++) {
        pthread_join(threads[i], NULL);
    }
    for (int i = 0; i < arraySize; i++) {
        printf("The moving sum of %d is %d\n", i, moving_sum[i]);
    }
    return 0;
}
```

```
stanley@stanley-VirtualBox:~/Tutorial6$ gcc -o question3 question3.c
stanley@stanley-VirtualBox:~/Tutorial6$ ./question3
The moving sum of 0 is 1
The moving sum of 1 is 1
The moving sum of 2 is 2
The moving sum of 3 is 6
The moving sum of 4 is 24
stanley@stanley-VirtualBox:~/Tutorial6$
```

4. The **producer/consumer problem** is a classic problem in synchronization, create a program that does the following.
    - Create a global array **buffer** of **length 5**, this is shared by producer and consumers and initialized to zero.
    - Prompts the user for **ten numbers** (store in an array use #define NUMBERS 10 for the size)
    - Creates two threads, one a producer, the other a consumer
    - The **producer thread** calls the function **producer** which takes the array of numbers from the users as an argument and does the following:
        - Loops until all ten items have been added to the buffer, each time with a **random delay** before proceeding
        - **Using semaphores** gets access to the critical section (buffer)
        - For each number added to **buffer** prints “Produced <number>”, to indicate the number that has been added to the buffer
        - If the **buffer is full,** it waits until a number has been consumed, so that another number can be added to the buffer
    - The **consumer thread** calls the function **consumer** and does the following:
        - Loops until ten items have been consumed from the buffer, each time with a **random delay** before proceeding
        - **Using semaphores** gets access to the critical section (buffer)
        - For each number **consumed** from the **buffer**, sets the buffer at that index to **0,** indicating that the value has been consumed.
        - For each number consumed, also prints “Consumed <number>”, to indicate the number that has been consumed from the buffer
        - If the **buffer is empty,** it waits until a number has been added, so that another number can be consumed from the buffer
    - The program waits for both threads to finish using **join()**, and then prints the contents of **buffer**, the contents of buffer should be all zeros.

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

#define BUFFER 5
#define N 10

int buffer[BUFFER] = {0};
sem_t empty, full, mutex;

void *producer(void *arg)
{
    int *num = (int *)arg;
    int index = 0;
    for (int i = 0; i < N; i++)
    {
        usleep(rand() % 1000);
        sem_wait(&empty);
        sem_wait(&mutex);
        buffer[index] = num[i];
        printf("Produced: %d\n", num[i]);
        index = (index + 1) % BUFFER;
        sem_post(&mutex);
        sem_post(&full);
    }
    pthread_exit(NULL);
}

void *consumer(void *arg)
{
```

```
34      int consumed = 0;
35      while (consumed < N)
36      {
37          usleep(rand() % 1000);
38          sem_wait(&full);
39          sem_wait(&mutex);
40          int index = consumed % BUFFER;
41          int value = buffer[index];
42          buffer[index] = 0;
43          printf("Consumed: %d\n", value);
44          consumed++;
45          sem_post(&mutex);
46          sem_post(&empty);
47      }
48      pthread_exit(NULL);
49  }
50
51  int main()
52  {
53      srand(time(NULL));
54      pthread_t producer_thread, consumer_thread;
55      sem_init(&empty, 0, BUFFER);
56      sem_init(&full, 0, 0);
57      sem_init(&mutex, 0, 1);
58      int num[N];
59      printf("Enter %d numbers:\n", N);
60      for (int i = 0; i < N; i++)
61      {
62          scanf("%d", &num[i]);
63      }
64      pthread_create(&producer_thread, NULL, producer, (void *)num);
65      pthread_create(&consumer_thread, NULL, consumer, NULL);
```

```
66      pthread_join(producer_thread, NULL);
67      pthread_join(consumer_thread, NULL);
68
69      printf("Contents of buffer:\n");
70      for (int i = 0; i < BUFFER; i++)
71      {
72          printf("%d\n", buffer[i]);
73      }
74      sem_destroy(&empty);
75      sem_destroy(&full);
76      sem_destroy(&mutex);
77      return 0;
```

```
stanley@stanley-VirtualBox:~/Tutorial6$ ./question4
Enter 10 numbers:
12
18
22
33
44
56
78
35
68
59
Produced: 12
Consumed: 12
Produced: 18
Consumed: 18
Produced: 22
Consumed: 22
Produced: 33
Produced: 44
Consumed: 33
```

```
Consumed: 18
Produced: 22
Consumed: 22
Produced: 33
Produced: 44
Consumed: 33
Consumed: 44
Produced: 56
Consumed: 56
Produced: 78
Consumed: 78
Produced: 35
Consumed: 35
Produced: 68
Consumed: 68
Produced: 59
Consumed: 59
Contents of buffer:
0
0
0
0
0
```

5. Create a program that does the following.
    - A master process which prompts a user for five numbers and writes the five numbers to a file called **numbers.txt**
    - The master process forks and creates a child process and then **waits for the child process to terminate**
    - The child process reads the five numbers from **numbers.txt** and creates five threads
    - Each thread executes a function **factorial**, which takes the number as an argument and does the following:
        - Calculates the factorial (e.g. 5! = 5x4x3x2x1 = 120).
        - Adds the factorial calculated to a global variable **total_sum** using the **+=** operator
        - The **total_sum** must be incremented in a thread-safe manner using **semaphores**
        - Prints the current factorial value and the calculated factorial
    - The child process has a **join** on all threads and after all threads have completed writes the **total_sum** to a file called **sum.txt** and terminates
    - After the child process has terminated the parent process reads the contents of **sum.txt** and prints the total sum.
    - Reading and writing files between processes is one of the simplest methods of IPC.

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
#include <semaphore.h>
#include <pthread.h>

#define N 5

int sum = 0;
sem_t sem;

void *factorial(void *arg) {
    int *n = (int *)arg;
    int f = 1;
    for (int i = 1; i <= *n; i++) {
        f *= i;
    }
    sem_wait(&sem);
    sum += f;
    sem_post(&sem);

    printf("Factorial of %d is %d\n", *n, f);
    printf("Sum is %d\n", sum);
    pthread_exit(NULL);
}

int main(){
    sem_init(&sem, 0, 1);
    printf("Enter numbers\n");
    int num[N];
    FILE *file = fopen("input.txt", "w");
    if (file == NULL) {
        printf("Error opening file!\n");
        exit(1);
```

```c
int main(){
    sem_init(&sem, 0, 1);
    printf("Enter numbers\n");
    int num[N];
    FILE *file = fopen("input.txt", "w");
    if (file == NULL) {
        printf("Error opening file!\n");
        exit(1);
    }
    for (int i = 0; i < N; i++) {
        scanf("%d", &num[i]);
        fprintf(file, "%d\n", num[i]);
    }
    fclose(file);

    pid_t pid = fork();
    if (pid == 0) {

        FILE *file = fopen("input.txt", "r");
        if (file == NULL) {
            printf("Error opening file!\n");
            exit(1);
        }

        pthread_t threads[N];
        for (int i = 0; i < N; i++) {
            pthread_create(&threads[i], NULL, factorial, &num[i]);
        }
        fclose(file);
        for (int i = 0; i < N; i++) {
            pthread_join(threads[i], NULL);
        }
        FILE *file2 = fopen("output.txt", "w");
        if (file2 == NULL) {
```

```
56          fclose(file);
57          for (int i = 0; i < N; i++) {
58              pthread_join(threads[i], NULL);
59          }
60          FILE *file2 = fopen("output.txt", "w");
61          if (file2 == NULL) {
62              printf("Error opening file!\n");
63              exit(1);
64          }
65          fprintf(file2, "Sum is %d\n", sum);
66          fclose(file2);
67          exit(0);
68
69      } else if (pid > 0) {
70          wait(NULL);
71          FILE *file = fopen("output.txt", "r");
72          if (file == NULL) {
73              printf("Error opening file!\n");
74              exit(1);
75          }
76          fscanf(file, "%d", &sum);
77          fclose(file);
78          printf("Sum is %d\n", sum);
79      }
80      else {
81          printf("Error\n");
82          exit(1);
83      }
84      sem_destroy(&sem);
85      return 0;
86
87
88  }
```

```
stanley@stanley-VirtualBox:~/Tutorial6$ gcc -o question5 question5.c
stanley@stanley-VirtualBox:~/Tutorial6$ ./question5
Enter numbers
25
3
4
5
6
Factorial of 3 is 6
Sum is 2076181326
Factorial of 5 is 120
Sum is 2076181326
Factorial of 6 is 720
Sum is 2076181326
Factorial of 25 is 2076180480
Sum is 2076181326
Factorial of 4 is 24
Sum is 2076181350
Sum is 0
stanley@stanley-VirtualBox:~/Tutorial6$
```