ECSE 427 / COMP 310
Programming Assignment #2: Processes and threads
Due date: Check My Courses

This assignment has two parts. The first part consists of short answer questions that relate to the lectures on processes and threads, and the second part consists of a c program.

Part 1: Short answer questions

P1Q1. Consider the code below. Show a situation where deadlock can occur. Do so by describing a sequence of steps, showing the values of all variables at every step. (5 marks)

```
/* process 0 */
.
.
flag[0] = true;
while (flag[1]);
/* critical section */
flag[0] = false;
.
.
```

```
/* process 1 */
.
.
flag[1] = true;
while (flag[0]);
/* critical section */
flag[1] = false;
.
.
```

step 0: flag[0]=true, flag[1]=False
step 1: process 0 runs, flag[0]=true, flag[1]=true
...

P1Q2. Consider the code below. Show a situation where it does not provide mutual exclusion. Do so by describing a sequence of steps, showing the values of all variables at every step. (5 marks)

```
/* process 0 */
.
.
while (turn != 0);
/* critical section */
turn = 1;
.
.
```

```
/* process 1 */
.
.
while (turn != 1);
/* critical section */
turn = 0;
.
.
```

Part 2: Reservation system (90 marks)

General requirements

You are to implement a reservation system for a gala. Reservations can be made one table at a time; and there are 2 sections, section A and section B, each with 10 tables

available (numbered 100 to 110 for section A and 200 to 210 for section B). The reservation system will keep a record of all the reserved table numbers and the name of the person who made the reservation. You will create this application with multiple processes. The processes should be independently launchable from the shell. You will use shared memory for creating the inter-process communication. Because processes do not share memory, you will create shared memory among the processes and setup the appropriate synchronization. Manipulation of shared variables such as the buffer should be protected using a semaphore.

Specific requirements

1- Every time a person wants to reserve a table, the command to execute within your program is:
>>reserve <person_name> <section> <table_number(optional)>

Your system should:

a- Check to see if there is a table available in the requested section. If a table number was not specified, your system must find a table by checking the reservation structure in increasing order of table number. If none are available (or the requested table is taken), send a message telling that to the user.
b- If a table is found to be available, the system should reserve it and update the reservation structure with the table name, the person's name, and the table's reservation status.

2- Other commands that the system should be able to handle are:
>>init
to reinitialize the database. This should cause all the reservations to get deleted.
>>status
This should print the current reservation status.

>>exit
This should exit the system.

3- The program should be able to accept these commands and should also be able to take as input a list of commands from a text file, one command per line.

4- Your program should handle invalid commands by sending an error message to the user (errors include an invalid table number).

5- Your program should handle memory leak issues

Implementation details

Your first step would be to create a shared memory space and store a data structure holding the current reservation information. The reservation information consists of the

table number, its reservation status, and the name of the person who reserved it.

To setup the shared memory, you need to use the POSIX shared memory mechanisms. POSIX shared memory has some known issues with Mac OS. Therefore, this assignment should be attempted in Linux (any Linux distribution with a recent kernel should be fine).

In POSIX shared memory a memory region that needs to be shared is referred to as the memory object. To create a memory object, you need to use the following library routine. The total size of all the shared memory objects that can be created in a single machine is limited by the size of the tmpfs file system setup by the system administrator. I don't think this limit would be a problem unless large number of students use the same machine for running their programs. You can check the /dev/shm directory to see shared memory objects in a machine. You need to do two steps to setup a shared memory object.

1. Use shm_open() function to open an object with the specified name. This is similar to the open() for files.

2. Pass the file descriptor obtained in the above step to a mmap() that specifies the MAP_SHARED in the flags argument.

The name argument of shm_open() identifies shared memory object to be created or opened (i.e., something already created). This function can include specific flag values and mode values to define the configuration required from the shared memory object.

```
#include <fcntl.h>           /* Defines O_* constants */
#include <sys/stat.h>        /* Defines mode constants */
#include <sys/mman.h>

int shm_open(const char *name, int oflag, mode_t mode);
                                Returns file descriptor on success, or –1 on error
```

Flag	Description
O_CREAT	Create object if it doesn't already exist
O_EXCL	With O_CREAT, create object exclusively
O_RDONLY	Open for read-only access
O_RDWR	Open for read-write access
O_TRUNC	Truncate object to zero length

The program listing below shows a small example program where the string provided as the first argument is copied into the shared memory. The shared memory is created under the name "myshared." Obviously, you need to use a unique name that is related to your login name so that there are no name conflicts even if you happen to use a lab machine for running the program. The mmap() is used to map the shared memory object starting at an address. By specifying a NULL value for the first argument of the mmap(), we are letting the kernel pick the starting location for the shared memory object in the virtual address space.

The ftruncate() call is used to resize the shared memory object to fit the string (first argument). As the last statement we copy the bytes into the shared memory region.

The example below shows a program for reading the contents in the shared memory object and writing it to the standard output. Obviously, the name of the shared memory object should match the one in the above program. The fstat() system call allows us to determine the length of the shared memory object. This length is used in the mmap() so that we can map only that portion into the virtual address space.

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

int main()
{
    struct stat s;

    int fd = shm_open("myshared", O_RDWR, 0);
    if (fd < 0)
        printf("Error.. opening shm\n");

    if (fstat(fd, &s) == -1)
        printf("Error fstat\n");

    char *addr = mmap(NULL, s.st_size, PROT_READ, MAP_SHARED, fd, 0);
    close(fd);

    write(STDOUT_FILENO, addr, s.st_size);
}
```

Another problem to consider is synchronization. When multiple processes access the shared memory region, we will run into the synchronization problem. At the very least, we need to take care of the mutually exclusive access needed for race free update of the shared structures. The UNIX operating system (Linux in our case), provides different variations of semaphore: System V, POSIX, etc. In this assignment, you need to use the POSIX semaphores. POSIX semaphores can be created in two different ways: named and anonymous. You need to use the named semaphore in this project. With the named semaphores you don't need to put them in a shared memory. Different processes can share a semaphore by simply using the same name. Because we have shared memory objects, we could have used the anonymous semaphores as well.
To create a named semaphore, use the following library function.

```
#include <fcntl.h>           /* Defines O_* constants */
#include <sys/stat.h>        /* Defines mode constants */
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag, ...
                /* mode_t mode, unsigned int value */ );
                Returns pointer to semaphore on success, or SEM_FAILED on error
```

Once a semaphore is successfully created, two types of operations can be performed on them: wait() and post() (we referred to the post as signal in the lectures). The POSIX semaphore also provides additional APIs to obtain the current value and try waiting. The figure below shows the library function for waiting on a semaphore. It decrements the semaphore and if the value is greater than or equal to 0, it returns immediately. Otherwise, it gets blocked. Whether the value of the semaphore goes to negative or not is implementation dependent. In Linux, the value does not go negative. The process is blocked until the value becomes greater than 0 and then the value is decremented.

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
                                        Returns 0 on success, or −1 on error
```

The post() operation shown below increments the semaphore value. If a process is waiting on the semaphore, it will be woken and allowed to decrement the semaphore value. If multiple processes are waiting on the semaphore, one process is arbitrarily woken up and allowed to decrement the semaphore value. That is, only one process is let go when a post() operation happens among the waiting processes, however, the order in which the waiting processes are let go is undefined.

```
#include <semaphore.h>

int sem_post(sem_t *sem);
                                        Returns 0 on success, or −1 on error
```

One of the important problems to solve here is synchronization: readers and writers (R&W) problem. See Section 2.5.2 of the textbook. You can fit the algorithm provided there to achieve the synchronization in your system. In the R&W problem, readers overlap with other readers. However, there cannot be any overlap with writers. A writer needs mutually exclusive access to the database. Also, when a writer is updating the database, readers need to wait as well. Otherwise, readers would get invalid (partially updated) values.

Work to be submitted

- c file containing your code. You file should be called a2_fall2017.c
- 2 text files with test cases that aim at verifying that the requirements of this document have been met.
- pdf with the answers to part 1 of the assignment, and a trace of your output after running your program using the 2 test text files concurrently.