Department of Computer and Information Sciences
Towson University
Laboratory for Operating Systems Course

Process Synchronization Lab

Background:

All POSIX semaphore functions and types are prototyped or defined in semaphore.h

#include <semaphore.h>

To compile you code which uses thread and semaphores:

>> gcc *filename.c* -o *filename* –lpthread -lrt

To declare a semaphore:

sem_t *sem_name*;

Example:

sem_t mutex;

To Initialize a semaphore:

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: the initial value

Example:

sem_init(&mutex, 0, 1);

To wait on a semaphore:

int sem_wait(sem_t *sem);

Example:

sem_wait(&mutex);

To signal a semaphore:

int sem_post(sem_t *sem);

Example:

sem_post(&mutex);

To get a value of a semaphore:

int sem_getvalue(sem_t *sem, int *valp);

Example:

```
int val;
sem_getvalue(&mutex, &val);
```

To destroy a semaphore:

int sem_destroy(sem_t *sem);

Example:

```
sem_destroy(&mutex);
```

---

1. **(5 points)** The producer-consumer problem is a classic example of multi-thread/ multi-process synchronization problem. In this code a producer produces a letter and puts it in a buffer and a consumer retrieves a letter and prints it. **Compile and run the program as given below and explain the output generated by the program.**

```
#include <pthread.h>
#include <stdio.h>
#define BUFF_SIZE 20

char buffer[BUFF_SIZE];
int nextIn = 0;
int nextOut = 0;

void Put(char item)
{
  buffer[nextIn] = item;
  nextIn = (nextIn + 1) % BUFF_SIZE;
  printf("Producing %c ...\n", item);
}
void * Producer()
{
    int i;
    for(i = 0; i < 15; i++)
    {
      sleep(rand()%9);
      Put((char)('A'+ i % 26));
    }
}

void Get(char item)
{
  item = buffer[nextOut];
  nextOut = (nextOut + 1) % BUFF_SIZE;
```

```
  printf("Consuming %c ...\n", item);
}
void * Consumer()
{
   int i;
   char  item;
   for(i = 0; i < 15; i++)
   {
     sleep(rand()%4);
     Get(item);
   }
}
void main()
{
            pthread_t pid, cid;
            pthread_create(&pid, NULL, Producer, NULL);
            pthread_create(&cid, NULL, Consumer, NULL);
            pthread_join(pid, NULL);
            pthread_join(cid, NULL);
}
```

2. **(15 pints)** Using semaphores modify your code to allow proper synchronization between multiple producers and multiple consumers running concurrently. Include the thread ID of the producers and consumers in the print statements. Pthread_self( ) function returns the pthread ID of the calling thread. **Submit a copy of your source code and a screenshot of your output.**

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

#define BUFF_SIZE 20
char buffer[BUFF_SIZE];
int nextIn = 0;
int nextOut = 0;

//used for synchronization – producer should be blocked if the buffer is full
sem_t empty_slots;
//used for synchronization – consumer should be blocked if the buffer is empty
sem_t full_slot;
//used for mutual exclusion – enforces mutual exclusion on the shared buffer
sem_t mutex;


void Put(char item)
{
  ...
  buffer[nextIn] = item;
  nextIn = (nextIn + 1) % BUFF_SIZE;
  printf("Producing %c ...\n", item);
  ...
}
void * Producer()
{
    int i;
    for(i = 0; i < 15; i++)
    {
      sleep(rand()%6);
      Put((char)('A'+ i % 26));
    }
}
void Get(char item)
{
  ...
  item = buffer[nextOut];
  nextOut = (nextOut + 1) % BUFF_SIZE;
  printf("Consuming %c ...\n", item);
```

```
   ...
}
void * Consumer()
{
   int i;
   char  item;
   for(i = 0; i < 15; i++)
   {
     sleep(rand()%6);
     Get(item);
   }
}

void main()
{
```

/* Initialize semaphores */

```
        pthread_t pid, cid;
        pthread_create(&pid, NULL, Producer, NULL);
        pthread_create(&cid, NULL, Consumer, NULL);
```

// create more consumer & producer threads.

```
        pthread_join(pid, NULL);
        pthread_join(cid, NULL);
}
```

Synchronize Unrelated Processes:

3. (30 points) Programs below simulate the producer-consumer problem as independent processes which do not share anything by default. In this version, processes share data via a shared memory segment and access the same semaphores by using their unique ID (name).
   (1) Compile and run the programs given below as separate processes.
   (2) Modify the code for both producer & consumer to include the PID of the process in the output statements.
   (3) Start multiple producers and multiple consumers. Do you have to make any changes to the code to run properly? Explain your answer.
   (4) modify the code given below by adding a third type of process which processes the data generated by the producer, before it's used and printed by the consumer. In this example, the "Processing process" changes the case if the item generated by the producer from upper-case to lower-case and stores it back in the buffer before the consumer can retrieve and print it. Submit a copy of your program and snapshot of your output.

The sem_open( ) function creates a new or opens an existing named semaphore and returns a semaphore pointer that can be used on subsequent operations.

sem_t *sem_open(const char *_name_, int _oflag_);
sem_t *sem_open(const char *_name_, int _oflag_,
mode_t _mode_, unsigned int _value_);

//code for Producer.c

```
#include <stdio.h>
#include <sys/types.h>
#include <semaphore.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/shm.h>

#define BUFF_SIZE 20


typedef struct {
char buffer[BUFF_SIZE];
int nextIn;
int nextOut;
} shared_data;

shared_data *shm, *s;
char sem_name1[] = "mutex";
char sem_name2[] = "empty_slots";
char sem_name3[] = "full_slots";

sem_t *empty_slots;
sem_t *full_slots;
sem_t *mutex;
```

```
void Put(char item)
{
  sem_wait(empty_slots);
  sem_wait(mutex);
  s->buffer[s->nextIn] = item;
  s->nextIn = (s->nextIn + 1) % BUFF_SIZE;
  sem_post(mutex);
  printf("Producing %c ...\n", item);
  sem_post(full_slots);
}

void Producer()
{
    int i;

    for(i = 0; i < 10; i++)
    {
      sleep(rand()%3);
      Put((char)('A'+ i % 26));
    }
}


void main()
{
       //Create and initialize the semaphores

       mutex=sem_open(sem_name1, O_CREAT,0644, 1);
       full_slots=sem_open(sem_name3, O_CREAT,0644, 0);
       empty_slots=sem_open(sem_name2, O_CREAT,0644, 10);

       //Create a key for the segment
       key_t key;
       key = 1234;   //NOTE:  you can use ftok() function to generate the key

       //create the segment
       int shmid;

       if ((shmid = shmget(key, sizeof(shared_data), IPC_CREAT |0666)) <0)
       {
              perror("Shmget");
              exit(1);
       }

       //attach to the segment
       if ((shm = (shared_data *) shmat(shmid, NULL, 0))==(shared_data *) -1)
       {
              perror("Shmat");
              exit(1);
       }

       s=shm;
       s->nextIn = 0;

       Producer( );
       //detach from the segment
       shmdt((void *) shm);
```

```
}
```

//code for Consumer.c

```
#include <pthread.h>
#include <stdio.h>
#include <sys/types.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/shm.h>

#define BUFF_SIZE 20

char buffer[BUFF_SIZE];
int nextIn = 0;
int nextOut = 0;

char sem_name1[] = "mutex";
char sem_name2[] = "empty_slots";
char sem_name3[] = "full_slots";

sem_t *empty_slots;
sem_t *full_slots;
sem_t *mutex;

typedef struct {
char buffer[BUFF_SIZE];
int nextIn;
int nextOut;
} shared_data;

shared_data *shm, *s;

void Get(char item)
{
  sem_wait(full_slots);
  sem_wait(mutex);
  item = s->buffer[s->nextOut];
  s->nextOut = (s->nextOut + 1) % BUFF_SIZE;
  sem_post(mutex);
  printf("Consuming %c ...\n", item);
  sem_post(empty_slots);
}

void Consumer()
{
    int i;
    char  item;
    for(i = 0; i < 10; i++)
    {
      sleep(rand()%9);
      Get(item);
    }
}

void main()
{
       //Open exisitng semaphores
```

```
        mutex=sem_open(sem_name1, O_CREAT,0644, 1);
        full_slots=sem_open(sem_name3, O_CREAT,0644, 0);
        empty_slots=sem_open(sem_name2, O_CREAT,0644, 10);

        //Create a key for the segment
        key_t key;
        key = 1234;   //NOTE:  you can use ftok() function to generate the key

        //locate the segment
        int shmid;
        if ((shmid = shmget(key, sizeof(shared_data),0666)) <0)
        {
                perror("Shmget");
                exit(1);
        }
        //attach to the segment
        if ((shm = (shared_data *) shmat(shmid, NULL, 0))==(shared_data *) -1)
        {
                perror("Shmat");
                exit(1);
        }

        s=shm;
        s->nextOut = 0;
        Consumer();
}
```

3. **(50 points)** Our Department has a teaching assistant who helps undergraduate students with their programming assignments during regular office hours. The TA's office is rather small and has room for only one desk with a chair and computer. There are three chairs in the hallway outside the office where students can sit and wait if the TA is currently helping another student. When there are no students who need help during office hours, the TA sits at the desk and takes a nap. If a student arrives during the office hours and finds the TA sleeping, the student must awaken the TA (using a semaphore) to ask for help, if a student arrives and finds the TA currently helping another student, the student sits on one of the chairs in the hallway and waits. If no chairs are available, the student will go back to programming and come back at a later time. When the TA finishes helping a student, the TA must check to see if there are students waiting for help in the hallway. If so, the TA must help each of these students in turn.

Using Pthreads, begin by creating 10 students. Each will run as a separate thread. The TA will run as a separate thread as well. Students thread will alternate between programming for a period of time and seeking help from the TA. To simulate students programming – as well as the TA providing help to a student – is to have the appropriate thread to sleep for a random period of time. Each student thread should print its state (programming, waiting to get help, getting help, TA not available back to programing) along with its threadID. Similarly, the TA should print its state (helping, getting the next student, napping).

Submit a copy of your source code and a screenshot of your output.