BITS, Pilani - Hyderabad Campus
Operating Systems (CS F372)

Tutorial – 8

In this tutorial, you will learn about synchronization mechanism - Semaphores.

Semaphores:

A semaphore is used to control the number of processes/threads that can use a resource simultaneously. A process (or a thread) waits for permission to proceed by waiting for the integer to become > 0. On success, the process signals this by decrementing the integer by 1. When it is finished using the resource, it adds 1 to semaphore value. A semaphore is as an object with an integer value that we can manipulate with two routines. Because the initial value of the semaphore determines its behaviour, before calling any other routine to interact with the semaphore, we must first initialize it to some value, as this code below does:

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1);
```

Notice the initialization function sem_init() which has the folowing prototype:

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

Waiting on a semaphore :

```
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
```

Incrementing a semaphore :

```
int sem_post(sem_t *sem);
```

Getting the value of a semaphore :

```
int sem_getvalue(sem_t *sem, int *sval);
```

Destroying a semaphore :

```
int sem_destroy(sem_t *sem);
```

Now let us look at a simple thread synchronization mechanism using binary semaphore where it works as a mutex lock:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <semaphore.h>
sem_t s;
int counter; /* shared variable */
/* prototype for thread routine */
void handler (void *ptr) {
      int x;
      x = *((int *) ptr);
      printf("Thread %d: Waiting to enter critical region...\n", x);
      sem_wait(&s);
      // CRITICAL REGION
      printf("Thread %d: Now in critical region...\n", x);
      printf("Thread %d: Counter Value: %d\n", x, counter);
      printf("Thread %d: Incrementing Counter...\n", x);
      counter++;
      printf("Thread %d: New Counter Value: %d\n", x, counter);
      printf("Thread %d: Exiting critical region...\n", x);
      //END OF CRITICAL REGION
```

```
	sem_post(&s);
	pthread_exit(0);
}

int main() {
	int i[2];
	pthread_t thrd_a;
	pthread_t thrd_b;
	i[0] = 0; i[1] = 1;
	sem_init(&s, 0, 1);
	pthread_create (&thrd_a, NULL, (void *) &handler, (void *) &i[0]);
	pthread_create (&thrd_b, NULL, (void *) &handler, (void *) &i[1]);
	pthread_join(thrd_a, NULL);
	pthread_join(thrd_b, NULL);
	sem_destroy(&mutex);
	exit(0);
}
```

Producer Consumer problem:

Producer/consumer problem is sometimes called *Bounded buffer problem*.

Imagine one or more producer threads and one or more consumer threads. Producers produce data items and wish to place them in a buffer; consumers grab data items out of the buffer and consume the data in some way.

This scenario occurs in many places within real life systems. For example, in a multithread web server, a producer puts HTTP requests into a work queue (i.e., the bounded buffer); a thread pool of consumers each take a request out of the work queue and process the request. Because the bounded buffer is a shared resource, we must require synchronized access to it, otherwise a race condition will occur.

A sample producer consumer program:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include </usr/include/semaphore.h>
#include <unistd.h>
#define BUFF 5 /* total number of slots */
#define NP 3 /* number of producers */
#define NC 3 /* number of consumers */
#define N 4 /* number of items produced/consumed */
typedef struct
{
	int buf[BUFF]; /* shared var */
	int in; /* buf[in%BUFF] is the first empty slot */
	int out; /* buf[out%BUFF] is the first full slot */
	sem_t full; /* keep track of the number of full spots */
	sem_t empty; /* keep track of the number of empty spots */
	pthread_mutex_t m; /* enforce mutual exclusion to shared data */
} sbuf_t;
sbuf_t shared;
void *Producer(void *arg)
{
	int item;
	long i;
	i = (long)arg;
	for (i=0; i < N; i++) {
		// Produce an item
		item = i;
		// First wait for an empty slot
		sem_wait(&shared.empty);
		// We are here since there is an empty slot; wait if lock is not
		// available for read/write
		pthread_mutex_lock(&shared.m);
		shared.buf[shared.in] = item;
```

```
            shared.in = (shared.in+1)%BUFF;
            printf("[P%ld] Producing %d ...\n", i, item);
            fflush(stdout);
            // Release lock
            pthread_mutex_unlock(&shared.m);
            //done, so increment number of full slot by one (you just wrote)
            sem_post(&shared.full);
            //Interleave execution of producer and consumer
            if (i % 2 == 1) sleep(1);
      }
      return NULL;
}
void *Consumer(void *arg){
      int item;
      long i;
      i = (long)arg;
      for (i=N; i > 0; i) {
            sem_wait(&shared.full);
            pthread_mutex_lock(&shared.m);
            item=i;
            item=shared.buf[shared.out];
            shared.out = (shared.out+1)%BUFF;
            printf("[C%ld] Consuming %d ...\n", i, item);
            fflush(stdout);
            // Release lock
            pthread_mutex_unlock(&shared.m);
            // done reading, so increment the number of empty slots
            sem_post(&shared.empty);
            // Interleave
            if (i % 2 == 1) sleep(1); // randomize sleep time
      }
      return NULL;
}

int main() {
      pthread_t p, c;
      long i;
      sem_init(&shared.full, 0, 0);
      sem_init(&shared.empty, 0, BUFF);
      pthread_mutex_init(&shared.mutex, NULL);
      for (i = 0; i < NP; i++) {
            // Create new producers
            pthread_create(&p[i], NULL, Producer, (void *)i);
      }
      // Create new consumers
      for(i=0; i<NC; i++) {
            pthread_create(&c[i], NULL, Consumer, (void *)i);
      }
      pthread_exit(NULL);
      return 0;
}
```

Readers-writers problem:

This problem refers to the multiple threads eaders/writers to memory. There are a few constraints:

- While a thread is writing, no other thread can read or write to that shared resource.
- Multiple threads can read from the shared resource simultaneously.
- If a thread is reading, no thread can write to that shared resource.

It has three main variations depending on which among the reader and writer threads has preference.

1. Reader threads get preference i.e. no reader should be kept waiting if a reader thread is already reading. While a thread R is reading and a writer thread W is waiting for R to finish, any other reader threads get preference over W. In this case, the writer W may starve.

2. Writer gets preference i.e. a writer thread should not be kept waiting longer than absolutely necessary. No writer thread waits for readers if a writer thread is already running i.e. the first writer thread blocks out all readers to give writers the preference. Note that the actual write is performed exclusively, so the critical section must be secured before writing is performed. Here, readers could starve.
3. No thread should starve i.e. the solution is fair to both readers and writers.

The following is the code for the 1st variation. Note how the first reader thread is using *blockw* semaphore to block writer threads so that all readers can jump ahead and read.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <unistd.h>
#include <time.h>
sem_t blockw;
sem_t rmutex;
int readcount=0;
int data=0; //shared resource
void *writer_func(void *arg) {
    long f = (long)arg;
    sem_wait(&blockw);
    data++;
    printf("writer thread %d writes %d\n",f,data);
    usleep(1000);
    sem_post(&blockw);
}

void *reader_func(void *arg) {
    long f = (long)arg;
    sem_wait(&rmutex);
    readcount++;
    if (readcount == 1)
        sem_wait(&blockw);
    sem_post(&rmutex);
    printf("reader thread %d reads %d\n", f, data);
    usleep(1000);

    sem_wait(&rmutex);
    readcount--;
    if (readcount == 0)
        sem_post(&blockw);
    sem_post(&rmutex);
 }
int main()
{
    long i,b;
    pthread_t rtid[10],wtid[10];
    sem_init(&blockw,0,1);
    sem_init(&rmutex,0,1);
    for(i=0;i<=10;i++){
        pthread_create(&wtid[i],NULL,writer_func,(void *)i);
        pthread_create(&rtid[i],NULL,reader_func,(void *)i);
    }
    for(i=0;i<=2;i++){
        pthread_join(wtid[i],NULL);
        pthread_join(rtid[i],NULL);
    }
    return 0;
}
```

You can read more about the readers-writers problem and its implementation at:
https://en.wikipedia.org/wiki/Readers-writers_problem

Homework:

1. Implement the 2nd and 3rd variations of readers-writers problem for 10 concurrent readers/writers.

2. Consider the following scenario. There is a ticket booking counter that sells or cancels tickets for a train berth. Initially, there are a total of 10 berths available numbered from 101-110.
   Only one person can buy or cancel a ticket at a time. A person gets the first berth available from the numbered berths. If a person cancels a bought ticket, that berth will be made available. Every person who buys a ticket gets a ticket number and the booked berth number. The first ticket is numbered 1001 and for every successful buy, the number increases by 1, continuously.
   Implement a program (write two different functions ticket_buy() and ticket_cancel() to be called from main()) for this scenario when there are 20 people (1 to 20) who are standing in a queue in any order to buy a ticket. 10 of these persons from the queue are initially successful in getting a ticket (ticket number 1001 to 1010). Then tickets for berth number 101, 102, and 105 are cancelled, so three next persons from the queue will get the tickets (number 1011 to 1013).
   Implement while considering what happens in real life scenario when multiple people want to buy a ticket at the same time and how it is handled.

3. Implement “Dining Philospher’s problem” using an array of Mutexes such that there is no deadlock.