National University of Computer and Emerging Sciences

Operating System Lab – 11

Lab Manual

Contents

Objective

In today's lab we will study about process synchronization, we will see concept of semaphores and how we can implement this. We will study process synchronization problems and we will see sleeping barber problem.

Introduction to Process Synchronization

When processes cooperate to perform a task, they need to communicate on a shared medium. Usually this shared medium is some resource allocated to any of the processes cooperating. When more than one process tries to write on it, this causes concurrency control problem since if two or more processes are able to write the shared medium simultaneously, then there is a strong chance that the data becomes inconsistent. To avoid this concurrency control problem, we often allow processes to lock the resources in order to avoid simultaneous access. There are two types of locks:

- Shared lock (used for reading purposes)
- Exclusive lock (used for writing purposes)

With shared lock more than one reading processes can be in critical section of the code where they access the shared resource, whereas in exclusive lock, not more than a single process is allowed to access the resource.

Mutual Exclusion as you may have studied in theory class, is used to enforce the process concurrency control avoiding deadlock conditions. Today our task is to study the usage of semaphores.

Concept of Semaphores

A semaphore is a counter that can be used to synchronize multiple threads. Linux guarantees that checking or modifying the value of a semaphore can be done safely, without creating a race condition.Each semaphore has a counter value, which is a non-negative integer.

A semaphore supports two basic operations:

- **A wait operation** decrements the value of the semaphore by 1. If the value is already zero, the operation blocks until the value of the semaphore becomes positive (due to the action of some other thread). When the semaphore's value becomes positive, it is decremented by 1 and the wait operation returns.
- **A post operation** increments the value of the semaphore by 1. If the semaphore was previously zero and other threads are blocked in a wait operation on that semaphore, one of those threads is unblocked and its wait operation completes (which brings the semaphore's value back to zero).

There are two types of semaphores:

- Named semaphores
- Unnamed semaphores

Named semaphores are powerful semaphores that are usually created to share among processes running from different files. This type of semaphores creates a temporary file under the directory /temp, that stores the value of semaphore.

Unnamed semaphores are less powerful semaphores that often work in a single file, they have no presence outside the file.

Implementation of Semaphores

Basics

A semaphore is represented by a variable whose type is ‘sem_t’ and is defined in ‘semaphore.h’ library, the following are some basic routine calls related to semaphores.

S.NO	Functions	Description
1	sem_init()	Initialize an unnamed semaphore
2	sem_destroy()	Destroy an unnamed semaphore
3	sem_wait()	Locks the semaphore referenced by sem_t by performing a semaphore lock operation on that semaphore.
4	sem_post()	Unlocks a semaphore

Libraries

The semaphores use semaphore.h to define the required data structures and functions. Some of them are defined in pthread.h as well, so at times semaphore programs give undefined reference error to sem_wait and sem_post. If you happen to encounter this problem, try adding pthread.h and compiling it as:

```
gcc -o output -pthread source.c
```

Declaration

The data structure that defines the semaphore is sem_t, so if you need to create a semaphore, you should write:

```
sem_t sem;
```

Initialization

The ‘sem_init()’ initializes an unnamed semaphore, the definition of this routine call is given below:

```
int sem_init(sem_t *sem, int p shared, unsigned int value)
```

The above initializes unnamed semaphore referenced to sem by the value. P shared indicates whether the semaphore is shared among the processes (pshared is zero the semaphore is not shared otherwise shared).

Returns 0 if success otherwise -1

Example

```
sem_t sem;
sem_init(&sem, 0, 1);
```

Waiting on a Semaphore

```
int sem_wait(sem_t *sem)
```

if the value of semaphore is greater than zero, sem_wait() will immediately decreases it by 1 and returns. Otherwise the process is blocked. This function returns 0 if success and -1 when error.

Example

```
sem_t sem;
…
if(sem_wait(&sem)) {
//do critical stuff
}
```

Posting a Semaphore

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

Increments the value of the semaphore by 1 pointed by the sem_t. If the value becomes greater than zero the process is unblocked. Returns 0 on success and -1 on error.

Example

```
Sem_t sem;
…
if(sem_wait(&sem)) {
//some critical stuff
}
//after critical stuff
sem_post(&sem);
```

Destroying a Semaphore

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

Destroys an unnamed semaphore pointed by the variable sem_t, if there are process waiting for this semaphore or the semaphore is already destroyed. The program may produce an undefined behavior. This function returns 0 if success or -1 if error.

Example

```
sem_t sem;
…
sem_destroy(&sem);
```

Putting this altogether

The below code demonstrates the use of semaphore in a counter variable in a multithreaded environment.

```
/* Includes */
#include <unistd.h>     /* Symbolic Constants */
#include <sys/types.h> /* Primitive System Data Types */
#include <errno.h>      /* Errors */
#include <stdio.h>      /* Input/Output */
#include <stdlib.h>     /* General Utilities */
#include <pthread.h>    /* POSIX Threads */
#include <string.h>     /* String handling */
```

```
#include <semaphore.h> /* Semaphore */

/* prototype for thread routine */
void handler ( void *ptr );

/* global vars */
/*
semaphores are declared global so they can be accessed
in main() and in thread routine, here, the semaphore is
used as a mutex
*/
sem_t mutex;
int counter; //shared variable

int main() {
        int i[2] = {0, 1};
        pthread_t thread_a, thread_b;
        counter = 0;
        sem_init(&mutex, 0, 1);
        /*Initialize mutex to 1 and 2nd param zero means mutex is local*/
        /*Note: you can check for successful initialization by evaluating the return value of semaphore
and pthreads*/

        //Initializing and creating threads
        pthread_create(&thread_a, 0, (void *) &handler, (void *) &i[0]);
        pthread_create(&thread_b, 0, (void *) &handler, (void *) &i[1]);
        pthread_join(thread_a,NULL);
        pthread_join(thread_b,NULL);
        sem_destroy(&mutex);
        return 0;
}

void handler (void *ptr) {
        int x = *((int*)ptr);

        printf("sem [INFO] Thread %d Waiting to enter in critical region. \n", x);
        sem_wait(&mutex);
        //Critical Region Starts
                printf("sem [INFO] Thread %d Enters in Critical Region. \n", x);
                printf("sem [INFO] Thread %d Value of Counter is %d.\n",x,counter);
                printf("sem [INFO] Thread %d Increamenting The Value of counter\n",x);
                counter++;
                printf("sem [INFO] Thread %d New value of counter is: %d\n",x, counter);
                printf("sem [INFO] Thread %d Exiting Critical Region.\n", x);
        //Critical Region Ends
        sem_post(&mutex);
        pthread_exit(0);
}
```

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

sem_t s;

void handler(int signal)
{
	sem_post(&s); /* Release the Kraken! */
	printf("OK BAAAYE!\n");

}

void *singsong(void *param)
{
	sem_wait(&s);
	printf("I had to wait until your signal released me!\n");
}

int main()
{
	int ok = sem_init(&s, 0, 1 /* Initial value of zero*/);
	if (ok == -1) {
		perror("Could not create unnamed semaphore");
		return 1;
	}
	signal(SIGINT, handler); // Too simple! See note below

	pthread_t tid;
	pthread_create(&tid, NULL, singsong, NULL);
	pthread_exit(NULL); /* Process will exit when there are no more threads */
}
```

Study Process Synchronization Problem: Sleeping Barber Problem

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h> // The maximum number of customer threads.
#define MAX_CUSTOMERS 25 // Function prototypes...

void *customer(void *num);
void *barber(void *);

//Define the semaphores.
// waitingRoom Limits the # of customers allowed to enter the waiting room at one time.
sem_t waitingRoom;
// barberChair ensures mutually exclusive access to the barber chair.
sem_t barberChair;
// barberPillow is used to allow the barber to sleep until a customer arrives.
sem_t barberPillow;
// seatBelt is used to make the customer to wait until the barber is done cutting his/her hair.
sem_t seatBelt;
// Flag to stop the barber thread when all customers have been serviced.
int allDone = 0;

int main(int argc, char *argv[])
{
 pthread_t btid;
 pthread_t tid[MAX_CUSTOMERS];
 int i, x, numCustomers, numChairs; int Number[MAX_CUSTOMERS];
 printf("Maximum number of customers can only be 25. Enter number of customers and chairs.\n");
 scanf("%d",&x);
 numCustomers = x;
 scanf("%d",&x);
 numChairs = x;
 if (numCustomers > MAX_CUSTOMERS) {
 printf("The maximum number of Customers is %d.\n", MAX_CUSTOMERS);
 return 0;
 }
```

```
 printf("A solution to the sleeping barber problem using semaphores.\n");
for (i = 0; i < MAX_CUSTOMERS; i++) {
 Number[i] = i;
 }
 // Initialize the semaphores with initial values...
 sem_init(&waitingRoom, 0, numChairs);
 sem_init(&barberChair, 0, 1);
 sem_init(&barberPillow, 0, 0);
sem_init(&seatBelt, 0, 0);

 // Create the barber.
 pthread_create(&btid, NULL, barber, NULL);

 // Create the customers.
 for (i = 0; i < numCustomers; i++) {
 pthread_create(&tid[i], NULL, customer, (void *)&Number[i]);
 }
 // Join each of the threads to wait for them to finish.
for (i = 0; i < numCustomers; i++) {
 pthread_join(tid[i],NULL);
 }
 // When all of the customers are finished, kill the barber thread.
 allDone = 1;
 sem_post(&barberPillow); // Wake the barber so he will exit.
 pthread_join(btid,NULL);
return 0;
}
void *customer(void *number) {
 int num = *(int *)number; // Leave for the shop and take some random amount of time to arrive.
 printf("Customer %d leaving for barber shop.\n", num);
 sleep(5);
 printf("Customer %d arrived at barber shop.\n", num); // Wait for space to open up in the waiting
room...
 sem_wait(&waitingRoom);
 printf("Customer %d entering waiting room.\n", num); // Wait for the barber chair to become free.
 sem_wait(&barberChair); // The chair is free so give up your spot in the waiting room.
 sem_post(&waitingRoom); // Wake up the barber...
 printf("Customer %d waking the barber.\n", num);
 sem_post(&barberPillow); // Wait for the barber to finish cutting your hair.
 sem_wait(&seatBelt); // Give up the chair.
 sem_post(&barberChair);
 printf("Customer %d leaving barber shop.\n", num);
}

void *barber(void *junk)
{
```

```
// While there are still customers to be serviced... Our barber is omnicient and can tell if there are customers still on the way to his shop.
while (!allDone) { // Sleep until someone arrives and wakes you.
printf("The barber is sleeping\n");
sem_wait(&barberPillow); // Skip this stuff at the end
if (!allDone)
{ // Take a random amount of time to cut the customer's hair.
 printf("The barber is cutting hair\n");
 sleep(3);
 printf("The barber has finished cutting hair.\n"); // Release the customer when done cutting
 sem_post(&seatBelt);
}
else {printf("The barber is going home for the day.\n");}}}
```

Lab Activity

Task1:

a. Create an icecream eating contest problem protected by a semaphore lock. Use global variables icecreamremaining. Create 3 threads for 3 persons to eat icecreams until all are finished. But only one person will be given icecreamcone at a time by the salesman. So restrict access to icecreams so that only one thread can decrement it by using a semaphor. So use sem_wait and Sem_post in thread .

b. get each person to count money from his wallet for the icecream payment which can take one to 2 seconds during which other person can acquire the salesman to sell him icecream. Redo the coding to accommodate this condition in your threads as well.

Task 2:

You need to synchronize customers at boarding lounge of an airport using semaphore. where there are 10 customers, each needs to weight his luggage, get it checked and get a boarding pass. During each task passengers are too bored that they sleep, weighting luggage takes 4 seconds sleep, security check for luggage needs 7 seconds sleep and getting boarding pass needs 3 seconds sleep.