**CL2006 Operating**

**Systems Lab**

**LAB**

**-**

**09**

**Process Synchronization**

**(**

**Semaphore**

**)**

**NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES** **Spring 2023**

**National University of Computer and Emerging Sciences**

Operating System Lab – 09

*Lab Manual*

Contents

[Objective 2](#_Toc16605)

[Introduction to Process Synchronization 2](#_Toc16606)

[Concept of Semaphores 2](#_Toc16607)

[Implementation of Semaphores 3](#_Toc16608)

[Basics 3](#_Toc16609)

[Libraries 3](#_Toc16610)

[Declaration 3](#_Toc16611)

[Initialization 3](#_Toc16612)

[Waiting on a Semaphore 3](#_Toc16613)

[Posting a Semaphore 4](#_Toc16614)

[Destroying a Semaphore 4](#_Toc16615)

[Putting this altogether 4](#_Toc16616)

[Study Process Synchronization Problem: Sleeping Barber Problem 6](#_Toc16617)

Lab Activity............................................................................................................................................... 8

# 

# Objective

In today’s lab we will study process synchronization, we will see the 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 problems 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 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 and previous lab, 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 pshared, unsigned int value)

The above initializes unnamed semaphore referenced to sem by the value. Pshared 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);  } |

# Study Process Synchronization Problem: Sleeping Barber Problem

# The Sleeping Barber problem is a classic problem in process synchronization that is used to illustrate synchronization issues that can arise in a concurrent system. The problem is as follows:

# There is a barber shop with one barber and several chairs for waiting customers. Customers arrive at random times and if there is an available chair, they take a seat and wait for the barber to become available. If there are no chairs available, the customer leaves. When the barber finishes with a customer, he checks if there are any waiting customers. If there are, he begins cutting the hair of the next customer in the queue. If there are no customers waiting, he goes to sleep.

# The problem is to write a program that coordinates the actions of the customers and the barber in a way that avoids synchronization problems, such as deadlock or starvation.

One solution to the Sleeping Barber problem is to use semaphores to coordinate access to the waiting chairs and the barber chair. The solution involves the following steps:

Initialize two semaphores: one for the number of waiting chairs and one for the barber chair. The waiting chairs semaphore is initialized to the number of chairs, and the barber chair semaphore is initialized to zero.

Customers should acquire the waiting chairs semaphore before taking a seat in the waiting room. If there are no available chairs, they should leave.

When the barber finishes cutting a customer’s hair, he releases the barber chair semaphore and checks if there are any waiting customers. If there are, he acquires the barber chair semaphore and begins cutting the hair of the next customer in the queue.

The barber should wait on the barber chair semaphore if there are no customers waiting.

The solution ensures that the barber never cuts the hair of more than one customer at a time, and that customers wait if the barber is busy. It also ensures that the barber goes to sleep if there are no customers waiting.



Diagram

Description automatically generated

|  |
| --- |
| #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");  }  }  } |