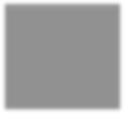
AIR UNIVERSITY, ISLAMABAD



Department of Computer

Science

FACULTYOF COMPUTING AND ARTIFICIAL INTELLIGENCE

Lab Manual : 10 Topic: Semaphore

Lab Instructor:

**Instructions:**

**Plagiarism:** Plagiarism cases will be dealt with strictly. If found plagiarized, both the involved parties will be awarded zero marks in this assignment, all of the remaining assignments, or even an F grade in the course. Copying from the internet is the easiest way to get caught!

**Deadline:** Late submission with marks deduction will be accepted according to the course policy shared earlier. Correct and timely submission of the assignment is the responsibility of every student; hence no relaxation will be given to anyone.

**Tip:** For timely completion of the assignment, start as early as possible. Furthermore, work smartly - as some of the problems can be solved using smarter logic.

1. Note: Follow the given instructions to the letter, failing to do so will result in a zero.

**Objectives:**

In this lab, you will learn:

* About Semaphore
* Types of semaphore
* Semaphore System calls
* Practical examples of Semaphore with implementation

Semaphore

A Semaphore S is an integer variable used for Controlling the synchronization. Semaphore is simply a variable that is non-negative and shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.

It can that can be accessed only through two *atomic* operations. i.e. these operation can not be interrupted and can be treated as an indivisible step.

**Operation on Semaphore**

**Wait / Down P**

**Signal /UP V**

* Semaphores are used to avoid race condition. Therefore, Semaphores are control mechanism.
* Race condition occurs when two or more processes are in their critical section.
* Critical section is a part of code in which a process accesses a shared resource.
* So whenever two or more processes in their critical sections, their may be chances that they might go in race condition.
* A semaphore is a data structure, which can be shared by a group of processes.
* When several processes compete for same resource (e.g. a file, a message queue, or a shared memory), their operations may be synchronized with the use of semaphores, so that they do not interfere with each other.

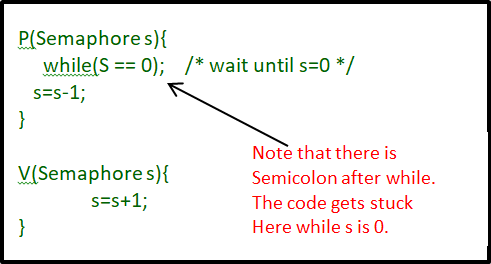
Semaphores are of two types:

1. **Binary Semaphore –**

This is also known as mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes.

1. **Counting Semaphore –**

Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.



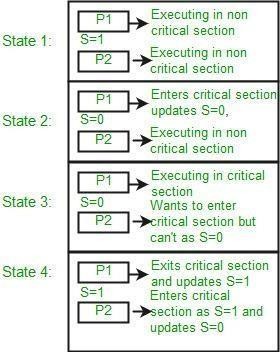
A semaphore is variable S which takes one of two values:

* + 0 when a resource is lock (protected), and should not be accessed by other processes,
  + 1 when the resource is unlocked (released).
  + Such semaphore, which could take two values 0 or 1, is called binary semaphore.

But in Linux implementation the two values are

* + –1, which is our P operation to wait for a semaphore to become available

 +1, which is our V operation to signal that a semaphore is now available.



# Semaphore System Calls:

#include <semaphore.h>

* + int sem\_init();
  + int sem\_wait();
  + int sem\_trywait();
  + int sem\_post();
  + int sem\_destroy();

# Create a semaphore:

int **sem\_init** (sem\_t\* sem, int pshared, unsigned int value);

* + **sem\_t** - the semaphore to be initialized
  + **pshared :** This argument specifies whether or not the newly initialized semaphore is shared between processes or between threads. A non-zero value means the semaphore is shared between processes and a value of zero means it is shared between threads.
  + **value** - the initial value of the semaphore

All semaphore functions return zero on success and non- zero on failure.

# Semaphore Operations:

int **sem\_post**(sem\_t \* *sem*);

* + This will increase the value of the semaphore by one.

int **sem\_wait** (sem\_t\* sem);

* + This will return immediately if the value of the semaphore is greater than zero and block the thread otherwise. It decreases the semaphore by one.

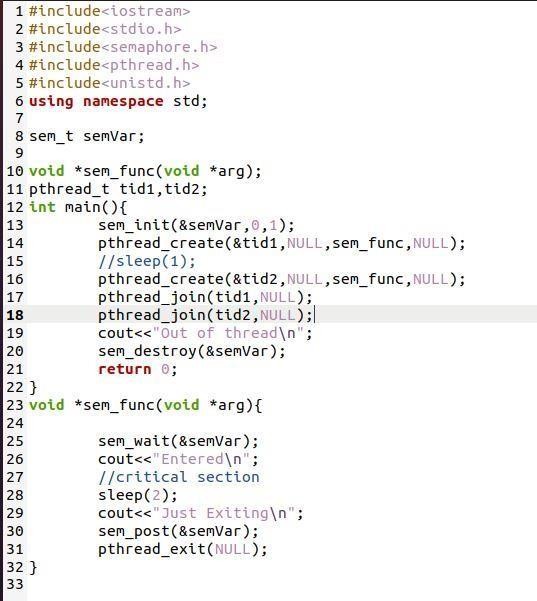
int **sem\_trywait**(sem\_t \* *sem*);

* + Version of the sem\_wait which does not block. Decreases the semaphore by one if the semaphore does not equal to zero. If it is zero it does not block, returns zero with error code **EAGAIN**

int **sem\_destroy**(sem\_t \* *sem*);

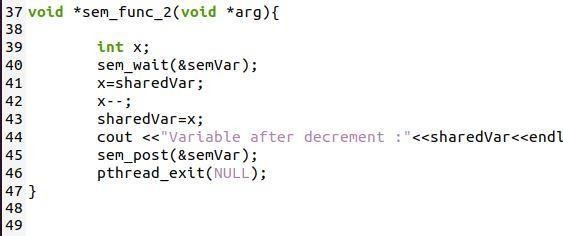
It releases the resources that semaphore has and destroys it.

**Semaphores Example 1:**



**Semaphores Example 2:**





**Semaphores Example 3:**





**Your Task:**

Write a program that creates a certain number of sellerthreads that attempt to sell all the available tickets. There is a global variable numTickets which tracks the number of tickets remaining to sell.

We will create many threads that all will attempt to sell tickets until they are all gone. Each thread will exit after all the tickets have been sold.

E.g. if we have 4 tickets and 2 sellerthreads then output should be like

Seller #1 sold one (3 left)

Seller #0 sold one (2 left)

Seller #1 sold one (1 left)

Seller #0 sold one (0 left)

Seller #1 noticed all tickets sold! (I sold 2 myself)

Seller #0 noticed all tickets sold! (I sold 2 myself)

Done