National University of Computer and Emerging Sciences

Lab Manual 7
(Operating Systems)

Department of Computer Science
FAST-NU, Lahore

Semaphores

OBJECTIVE: To understand and learn about semaphores in threads

Semaphores are used in concurrent programming to control access to shared resources or critical sections by multiple threads or processes. They serve several important purposes:

1. **Synchronization**: Semaphores are commonly used to synchronize access to shared resources among multiple threads or processes. They ensure that only one thread or a limited number of threads can access a critical section of code or a shared resource at any given time. This helps prevent race conditions and ensures that data integrity is maintained.

2. **Mutual Exclusion**: Semaphores can enforce mutual exclusion, which means they prevent multiple threads from accessing a critical section simultaneously. By controlling access using semaphores, you can ensure that only one thread can execute the critical section at a time, thereby avoiding conflicts and inconsistencies.

Semaphore Library: " semaphore.h"

Variables:

1. **semaphore:**
    - This variable represents the semaphore itself, which is of type sem_t.
    - Semaphores are special variables used for controlling access to shared resources among multiple threads or processes.
    -

Functions

1. **sem_init(sem_t *sem, int pshared, unsigned int value):**
    - sem_init initializes the semaphore sem with the specified attributes.
    - sem: A pointer to the semaphore variable (sem_t) that needs to be initialized.
    - pshared: This parameter determines whether the semaphore should be shared between threads (pshared != 0) or processes (pshared == 0). For thread synchronization, typically set to 0.
    - value: The initial value of the semaphore, which specifies the number of threads or processes that can simultaneously access the shared resource without blocking. If value is greater than 1, it's known as a counting semaphore.
    - In the provided code, sem_init(&semaphore, 0, 1); initializes the semaphore semaphore with an initial value of 1, meaning only one thread can access the critical section at a time.
2. **sem_wait(sem_t *sem):**
    - sem_wait decrements (waits for) the semaphore pointed to by sem.
    - If the semaphore's value is greater than zero, it decrements the value immediately and allows the calling thread to proceed.
    - If the semaphore's value is zero, sem_wait will block (put the calling thread to sleep) until the semaphore's value becomes positive (indicating that the critical section is available).
    - In the provided code, sem_wait(&semaphore); is used by threads to attempt to acquire (decrement) the semaphore before entering the critical section.
3. **sem_post(sem_t *sem):**
    - sem_post increments (signals) the semaphore pointed to by sem.
    - This function is used to release the semaphore after a thread has finished using the shared

resource or critical section.

- If there are other threads waiting for the semaphore (sem_wait), sem_post will wake up one of the waiting threads.
- In the provided code, sem_post(&semaphore); is used by threads to release (increment) the semaphore after completing their work in the critical section, allowing another thread to acquire the semaphore.

Demo Code:

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
 sem_t semaphore;
void *thread_function(void *arg) {
    int thread_id = *((int *)arg);
  printf("Thread %d is trying to acquire the semaphore...\n", thread_id);
      sem_wait(&semaphore)
       printf("Thread %d has acquired the semaphore!\n", thread_id);
       sleep(2);
       sem_post(&semaphore);
       printf("Thread %d has released the semaphore.\n", thread_id);
       pthread_exit(NULL);
}

int main() {
   pthread_t threads[3];
   int thread_ids[3] = {1, 2, 3};
   sem_init(&semaphore, 0, 1); // Initial value of semaphore is 1 (available)
   for (int i = 0; i < 3; i++) {
      pthread_create(&threads[i], NULL, thread_function, &thread_ids[i]);
   }
   for (int i = 0; i < 3; i++) {
      pthread_join(threads[i], NULL);
   }
   sem_destroy(&semaphore);

   return 0;
}
```

Output:

```
Thread 1 has acquired the semaphore!

Thread 1 has released the semaphore!

Thread 2 has acquired the semaphore!

Thread 2 has released the semaphore!

Thread 3 has acquired the semaphore!

Thread 3 has released the semaphore!
```

Question 1:

Reader-Writer Problem: Write a C program to solve the reader-writer problem using semaphores. Implement functions start_read, end_read, start_write, and end_write to coordinate concurrent access to a shared resource where multiple readers can access simultaneously but exclusive access is required for writers.

Hint: Use counting semaphores for readers

Question 2:

Traffic Light Simulation with Pedestrian Crossing

Design a multithreaded simulation to manage a four-way intersection with traffic lights for vehicles (North-South and East-West) and pedestrian signals for crossing each direction. Use semaphores to coordinate the state transitions of traffic lights and pedestrian signals, ensuring safe and synchronized behavior.

Tasks:

1. Initialize semaphores (**semaphore_NS**, **semaphore_EW**, **semaphore_ped**) and any necessary global variables.
2. Implement functions for traffic light controllers (**traffic_light_NorthSouth()**, **traffic_light_EastWest()**) and pedestrian crossing (**pedestrian_crossing()**) using semaphore synchronization.
3. Write a main simulation loop that orchestrates the behavior of traffic lights and pedestrian signals based on predefined timings and intervals.