Lab-13
Semaphores

What is a Semaphore?

A semaphore is a tool in programming that helps manage multiple tasks (or threads) trying to use the same resource (like memory or a file). Think of it like a traffic light that controls how many cars (threads) can enter a bridge (critical section) at a time.

- **Binary Semaphore:** This semaphore is like a traffic light that only lets one car go at a time. It allows one thread into the critical section while blocking others until it finishes.
- **Counting Semaphore:** This semaphore is like a toll booth with multiple lanes. It allows a specific number of threads to access a resource at the same time. For instance, if it's set to 3, then up to three threads can enter, but a fourth one must wait until one of the others finishes.

Why Use Semaphores?

Semaphores are used in programs to:

- **Control access to resources:** To avoid errors when multiple threads try to read or write to the same resource simultaneously.
- **Prevent deadlocks:** By making sure that threads don't get stuck waiting indefinitely for resources.
- **Manage conditions:** For example, one thread waits for another to signal before it starts working.

Basic Operations with Semaphores

Semaphores have four main functions in C:

- `sem_init`: Initializes the semaphore with a given value (like setting up a traffic light with "n" green lights).
- `sem_wait`: Decreases the semaphore count by 1. If it's zero, the thread waits until it becomes greater than zero.
- `sem_post`: Increases the semaphore count by 1. This is like saying, "I'm done!" so that another waiting thread can proceed.
- `sem_destroy`: Cleans up the semaphore when it's no longer needed.

Let's go over these with examples!

Example 1: Binary Semaphore (One at a Time)

In this example, only one thread can enter the critical section at a time:

Listing 1: Binary Semaphore Example

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

sem_t semaphore;

void* thread_function(void* arg) {
    sem_wait(&semaphore);  // "Locks" the semaphore, decreasing count by 1
    printf("Thread␣%ld␣is␣entering␣the␣critical␣section.\n", (long)arg);

    // Simulate some work
    sleep(1);

    printf("Thread␣%ld␣is␣leaving␣the␣critical␣section.\n", (long)arg);
    sem_post(&semaphore);  // "Unlocks" the semaphore, increasing count by 1
    return NULL;
}

int main() {
    pthread_t threads[3];
    sem_init(&semaphore, 0, 1);  // Initialize semaphore with 1 (binary)

    for (long i = 0; i < 3; i++) {
        pthread_create(&threads[i], NULL, thread_function, (void*)i);
    }

    for (int i = 0; i < 3; i++) {
        pthread_join(threads[i], NULL);
    }

    sem_destroy(&semaphore);  // Cleanup
    return 0;
}
```

Explanation:

- `sem_init(&semaphore, 0, 1)` sets up the semaphore to allow only one thread at a time.
- `sem_wait` locks the semaphore, letting only one thread enter the critical section.
- `sem_post` unlocks it, letting the next waiting thread enter.

Example 2: Counting Semaphore (Multiple at a Time)

This example allows up to 3 threads to access the critical section simultaneously:

Listing 2: Counting Semaphore Example

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

```
#include <pthread.h>
#include <semaphore.h>

sem_t semaphore;

void* thread_function(void* arg) {
    sem_wait(&semaphore);  // Locks (or waits) for available access
    printf("Thread %ld is accessing the resource.\n", (long)arg);

    // Simulate some work
    sleep(2);

    printf("Thread %ld is done using the resource.\n", (long)arg);
    sem_post(&semaphore);  // Unlocks, allowing another thread to access
    return NULL;
}

int main() {
    pthread_t threads[5];
    sem_init(&semaphore, 0, 3);  // Allows up to 3 threads at a time

    for (long i = 0; i < 5; i++) {
        pthread_create(&threads[i], NULL, thread_function, (void*)i);
    }

    for (int i = 0; i < 5; i++) {
        pthread_join(threads[i], NULL);
    }

    sem_destroy(&semaphore);  // Cleanup
    return 0;
}
```

Explanation:

- `sem_init(&semaphore, 0, 3)` allows up to 3 threads to enter the critical section at once.
- `sem_wait` decreases the semaphore count by 1, blocking the fourth thread when 3 are already in the critical section.
- `sem_post` increases the count, letting a waiting thread access the resource once another thread exits.

Example 3: Binary Semaphore as a Signal

In this example, one thread waits for a signal from another before proceeding:

Listing 3: Binary Semaphore as a Signal

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

sem_t semaphore;
```

```
void* thread1_function(void* arg) {
    printf("Thread 1: Waiting for signal...\n");
    sem_wait(&semaphore);  // Waits until sem_post is called
    printf("Thread 1: Received signal, proceeding...\n");
    return NULL;
}

void* thread2_function(void* arg) {
    sleep(1);  // Simulate some work before signaling
    printf("Thread 2: Sending signal...\n");
    sem_post(&semaphore);  // Signal to thread 1
    return NULL;
}

int main() {
    pthread_t thread1, thread2;
    sem_init(&semaphore, 0, 0);  // Initial value of 0, so thread 1 waits

    pthread_create(&thread1, NULL, thread1_function, NULL);
    pthread_create(&thread2, NULL, thread2_function, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    sem_destroy(&semaphore);  // Cleanup
    return 0;
}
```

Explanation:

- `sem_init(&semaphore, 0, 0)` makes the semaphore start at 0, so `thread1_function` waits.
- `thread2_function` calls `sem_post`, signaling to `sem_wait` in `thread1_function` that it can now proceed.
- This lets `thread1_function` finish after receiving the signal from `thread2_function`.

Key Takeaways

- `sem_wait` and `sem_post` control access to shared resources.
- Binary semaphores are used for exclusive access (one thread at a time).
- Counting semaphores allow a set number of threads to enter a critical section.
- Semaphores can act as signals between threads, letting one thread wait until another gives the go-ahead.

Semaphores make concurrent programming safer by helping control the flow of threads that share resources.