Starting to Synchronize

CSL301 - Operating Systems

Class Assignment - 8

भारतीय प्रौद्योगिकी संस्थान भिलाई

Indian Institute of Technology Bhilai

Instructions

- You are provided with reference C programs for today's lab questions *e.g.*, *semaphoresync.c*, *mutexdemo.c*, *etc*.

Understanding Semaphores in C

"Open The File semaphore_sync.c"

- `sem_t sem;`
  Declares a semaphore object handle. The actual state is managed internally by the OS or library.
- `sem_init(&sem, 0, 0);`
  Initializes a semaphore for thread synchronization.
  - `&sem`: Pointer to the semaphore object to initialize.
  - `0` (second parameter): Scope flag. `0` means the semaphore is shared only between threads in the same process.
  - `0` (third parameter): Initial value of the semaphore. Here, it starts at 0, so threads waiting on it will block until another thread increments it.

Understanding Semaphores in C

"Open The File semaphore_sync.c"

- `sem_post(&sem);`
  Increments (signals) the semaphore, potentially unblocking waiting threads.
- `sem_wait(&sem);`
  Decrements (waits) on the semaphore; blocks if the value is zero until signaled.

Question 1: Thread Synchronization Using Semaphore

Objective: To understand semaphores and control execution order.
Given:

- Use reference code `semaphore_sync.c` from the provided ZIP file.
- Use POSIX threads and semaphores to synchronize two threads.
- Thread A prints “Hello from A”.
- Thread B prints “Hello from B”.
- The given code ensures Thread B executes only after Thread A finishes.

Question 1: Thread Synchronization Using Semaphore

What to Do:

1. Run the reference code and record the output.
2. Modify it by removing semaphore logic and observe the output by running it multiple times.
3. Explain why the order changes without synchronization.
4. Add one more thread "threadC" which should execute only after threadB has done its execution. Note - You will need one more sem_t variable

Question 2: Rendezvous Between Two Threads

Objective: Learn how two threads synchronize at a rendezvous point.
Problem Statement:

- Thread A prints: A1 (Starting task), A2 (Ending task)
- Thread B prints: B1 (Starting task), B2 (Ending task)
- Synchronization requirements:
  - A1 must happen before B2
  - B1 must happen before A2
  - The order of A1 and B1 does not matter

Question 2: Rendezvous Between Two Threads

- Use two semaphores: `aArrived` and `bArrived`
- Thread A signals `aArrived` after printing A1 and waits on `bArrived` before printing A2
- Thread B signals `bArrived` after printing B1 and waits on `aArrived` before printing B2
- WAP to demonstrate this synchronization

Mutex Declaration and Initialization

Open - Reference code `mutex_demo.c`

Declaring a Mutex

- `pthread_mutex_t lock;` declares a mutex variable named `lock`.
- A mutex ensures only one thread can access a shared resource at a time.

Initializing a Mutex

- `pthread_mutex_init(&lock, NULL);` initializes the mutex before use.
- `&lock`: Address of the mutex variable.
- `NULL`: Uses default mutex attributes.
- After initialization, the mutex is unlocked and ready for use.

Mutex Locking and Unlocking

Locking a Mutex

- `pthread_mutex_lock(&lock);` locks the mutex before accessing the shared resource.
- If another thread holds the lock, this call blocks until the mutex is available.
- Only one thread can hold the lock at a time.

Unlocking a Mutex

- `pthread_mutex_unlock(&lock);` unlocks the mutex after use.
- Allows other threads waiting on the mutex to proceed.

Question 3: Thread Synchronization Using Mutex

Objective: Prevent multiple threads from accessing a critical section simultaneously.

Given:

- Use reference code `mutex_demo.c` from the ZIP file.
- Compile and run the program.
- Observe the counter values printed by both threads.
- Observe how mutex ensures correct output.

Question 3: Thread Synchronization Using Mutex

Objective: Understand race conditions and how mutex prevents inconsistent access.

Task:

- Implement a multithreaded counter upto 1000000 times with 10 threads.
- Part 1: Without mutex (observe race condition)
- Part 2: With mutex (observe consistent count)
- Record results and explain why race conditions occur without mutex.
- WAP to accomplish the above tasks.

Question 4: Multiplex / Bounded Semaphore

Objective: Limit threads simultaneously accessing a critical section.
Task:

- Use reference code `multiplex_semaphore.c` from the ZIP file.
- Compile and run the program.
- Observe how many threads are in the critical section simultaneously.
- Explain how the semaphore controls concurrent access and why it prevents more than the allowed number of threads from entering the critical section.

Question 5: Barrier Synchronization

Objective: Understand barriers and their proper implementation.
Task:

- Create N threads (e.g., $N = 5$). Each prints:

```
Thread X reached the barrier
Thread X passed the barrier
```

- Part A: Non-solution using mutex & semaphore (may deadlock)
- Part B: Reusable barrier using mutex & 2 semaphores (correct)
- Explain why Part A may fail and Part B succeeds.
- Write C program for each part to accomplish the above tasks.

Part A: Naive Barrier Implementation

Naive Barrier Steps: barrier.c

1. Given code -
2. Each thread increments a shared `count` variable (protected by a mutex).
3. If `count == N`, signal the barrier semaphore.
4. All threads wait on the barrier semaphore before proceeding.

Reference Pseudocode:

```
rendezvous
mutex.wait()
count = count + 1
mutex.signal()
if count == n: barrier.signal()
barrier.wait()
critical point
```

Part B: Reusable Barrier Solution

Reusable Barrier Steps:

1. Use a shared `count` variable, a mutex, and two semaphores (`turnstile` and `turnstile2`).
2. Ensure no thread passes until all have arrived; barrier is reusable.

Reference Pseudocode:

```
rendezvous
mutex.wait()
count = count + 1
mutex.signal()
if count == n: turnstile.signal()
turnstile.wait()
turnstile.signal()
critical point
```