Operating System Lab: CS341

LAB 7: Process Synchronization

Course Instructor: Dr Satendra Kumar

Key Headers

- **<pthread.h>**: This header provides the API for creating and managing threads in C. It includes functions like pthread_create, pthread_join, and synchronization primitives like mutexes and condition variables.

- **<semaphore.h>**:This header provides functions for working with semaphores, which are synchronization primitives that control access to shared resources. Semaphores are used here to ensure that only one philosopher can pick up a fork at a time.

- **<unistd.h>**: This header provides access to the POSIX operating system API, including functions like sleep(), which is used to simulate thinking and eating behavior in the philosophers.

- **<stdio.h>**: This is the standard I/O library in C, used for input and output operations. The function printf() is used to display messages indicating the state of each philosopher.

Important Functions

➢ **pthread_create:** Creates a new thread.

- **Syntax:** int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);

- **thread**: Pointer to pthread_t, which will store the thread identifier.
- **attr**: Thread attributes, or NULL for default attributes.
- **start_routine**: Function pointer to the routine that the thread will execute.
- **arg**: Argument passed to the start routine.

- **Example:** pthread_create(&thread_id[i], NULL, philosopher, &philosophers[i]);

- **pthread_join:** Waits for a thread to finish executing.

- **Syntax:** int pthread_join(pthread_t thread, void **retval);

- **thread**: The thread to wait for.
- **retval**: Pointer to the return value of the thread (can be NULL).

- **Example:** pthread_join(thread_id[i], NULL);

This waits for the philosopher thread to finish.

- **sem_init:** Initializes a semaphore.

- **Syntax:** int sem_init(sem_t *sem, int pshared, unsigned int value);

- **sem**: Pointer to the semaphore to initialize.
- **pshared**: 0 if the semaphore is used by threads in the same process.
- **value**: The initial value of the semaphore.

- **Example:** sem_init(&S[i], 0, 0);
- This initializes the semaphore for each philosopher with an initial value of 0, meaning they cannot access the forks until signaled.

➢ **sem_wait:** Decreases the value of semaphore by 1 (blocking if the value is 0).

- **Syntax:** int sem_wait(sem_t *sem);
- sem: Pointer to the semaphore.

- **Example:** sem_wait(&S[phil_num]);

- This makes the philosopher wait for permission to eat (the semaphore must be signaled with sem_post before proceeding).

➢ **sem_post:** Increases the value of semaphore by 1, potentially waking a waiting thread.

- **Syntax:** int sem_post(sem_t *sem);
- sem: Pointer to the semaphore.

- **Example:** sem_post(&S[phil_num]);
- This signals that the philosopher is allowed to eat by increasing the value of the semaphore.

➢ sem_wait and sem_post with Mutex

- **Mutex (sem_wait(&mutex) and sem_post(&mutex))**: Mutexes (mutual exclusions) are used to ensure that only one philosopher at a time can access the critical section where forks are picked up or put down.

- **Syntax:** sem_wait(&mutex); // Lock the critical section

  sem_post(&mutex); // Unlock the critical section

- **pthread_mutex_lock and pthread_mutex_unlock:** Lock and unlock a mutex to control access to shared data.

- **Syntax for Locking:** int pthread_mutex_lock(pthread_mutex_t *mutex);
- **Syntax for Unlocking:** int pthread_mutex_unlock(pthread_mutex_t *mutex);

- **Example:** pthread_mutex_lock(&mutex);

  pthread_mutex_unlock(&mutex);

- **pthread_mutex_destroy and sem_destroy:** Destroys the mutex or semaphore after their usage.

- **Syntax for mutex:** int pthread_mutex_destroy(pthread_mutex_t *mutex);
- **Syntax for semaphore:** int sem_destroy(sem_t *sem);

- **Example:** pthread_mutex_destroy(&mutex);

  sem_destroy(&wrt);