Synchronization

Lab 5: Operating Systems

Marzieh Babaeianjelodar

Why we need Synchronization?

Concurrent access to shared data may result in data inconsistency!

Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes!

Mutex

A **mutex** object only allows one thread into a controlled section, forcing other threads which attempt to gain access to that section to wait until the first thread has exited from that section.

lock(mutex)

Do work

unlock(mutex)

Lock each thread at a time. There is ownership. Every thread owns the lock and mutex only allows one thread to access resource.

Show a simple example of Mutex lock

Declare: pthread_mutex_t lock

Initialization: pthread_mutex_init(&lock, NULL)

Acquire: pthread_mutex_lock(&lock)

Critical Section

Release: pthread_mutex_unlock(&lock)

Destroy: pthread_mutex_destroy(&lock)

Why do we need Semaphores?

Locks only provide mutual exclusion!

Ensures only one thread is in critical section at a time.

SEMAPHORES

Semaphore is protected integer variable that can facilitate and restrict access to shared resources in a multi-processing environment.

- **wait()**: is called when a process wants access to a resource. (when semaphore variable is negative, the process wait is blocked.)
- **signal()**: is called when a process is done using a resource.

For example, four bathrooms are available and four identical keys are available. Five people want to use the bathrooms. Four of those people obtain those four keys and use the four bathrooms. The last person has to wait(). If any of the people has finished using the bathrooms, signal()s. There is no ownership mechanism.

Show a simple example of Semaphore

Declare the semaphore global (outside of any function): sem_t sem;

Initialize the semaphore in the main function: sem_init(&sem, 0, 1);

sem_wait(&sem)

sem_post(&sem)

Consumer/Producer Problem

The producer–consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem.

The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue.

The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time.

The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.