Synchronization

Learning Outcomes

At the end of this session, the students should be able to:

1. Explain the synchronization problem of threads; and
2. Create C program which uses mutexes and semaphores

Content

Required Packages

1. Ubuntu 16.04
2. gcc
3. bash

Synchronization

Multi-threading allows your program to run in parallel and at the same time share the global resources of your program.

Threads may access (read or write) the same data at the same time which is unsafe if allowed.

	Thread 1	Shared Resources	Thread 2	
		`x = 5`		
this will get 5 →	`getX()`		`getX()`	← this will get 5
	`putX(12)`			
		`x = 12`	`getX()`	← this will get 12
	`putX(15)`		`putX(17)`	

`What will be the value of x after?`
`Assuming x = 17,`

	`X = 17`	
`putX(16)`		`getX()`

`What value of x will Thread 2 get?`

CC BY-NC-SA 4.0

To solve these problems, we need to protect the shared resources whenever we access them.

Mutual Exclusion

Also known as mutex, this method allows your thread to broadcast its access to a shared resource. A mutex is like a logical lock that you can attach to a resource

Thread 1	Shared Resources	Thread 2
	`x = 5`	
`lock_mutex()`		
`putX(12)`		`lock_mutex()`
`addX(5)`	`x = 12`	`(wait)`
`unlock_mutex()`	`x = 17`	`(wait)`
	`x = 17`	`getX()`
		`unlock_mutex()`

Think of it like this:
You have one lock, and every time you access a shared resource, you must first take hold of that lock. If the lock is being used by others, you should wait. Once the lock is free, you can get it and proceed to the shared resource.

Mutexes in C

To use mutex in C, you need a thread mutex variable of type pthread_mutex_t

Initializing a Mutex

This mutex can be initialized in two ways:

a. Calling the `pthread_mutex_init` function

```
int pthread_mutex_init( pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *a );
```

Where `mutex` is the address of the mutex variable, and
a is the address of the attribute of the mutex.

b. Initializing it using the `PTHREAD_MUTEX_INITIALIZER` macro

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Once initialized, you can use your `mutex` to block threads from accessing shared resources if a thread is already using it.

Locking a Mutex

```
pthread_mutex_lock(pthread_mutex_t *mutex)
```

where `mutex` is the address of the mutex lock.

This checks if the lock is already being used. If it is, then it will wait for it to be unlocked. Else, it will proceed to the critical region.

Unlocking a Mutex

```
pthread_mutex_unlock(pthread_mutex_t *mutex)
```

where `mutex` is the address of the mutex lock.

(C) Institute of Computer Science | UPLB
CC BY-NC-SA 4.0

This unlocks the lock that is currently being used by the calling function in order to make the lock available for other threads to use.

Semaphores

Semaphores are another way of synchronizing shared resources in multi-threading applications. A *semaphore* is a value that threads can check whenever they want to access a certain shared resource.

Semaphores are classified into two (2) types -- binary and counting:

A *binary semaphore* has only two (2) possible value: 0 or 1. It acts just like how mutexes acts. If the value of the semaphore is 0, it can mean that a thread is accessing a certain shared resource associated with that semaphore. If it is 1, then the resource can be accessed and the semaphore must be updated.

A *counting semaphore* can have additional values and allows a thread to access a shared resource as long as it is within the specified limit. The counting semaphore counts the number of available slots for accessing a resource.

Semaphores in C

To use semaphores in C, you need a thread semaphore variable of type **sem_t**.

Initializing a Semaphore

```
int sem_init(sem_t *sem, int shared, int value);
```

where *sem* is the address of the semaphore to be initialized,
shared is a flag to whether the semaphore will be shared to processes,
and *value* is the initial value of the semaphore

This creates a semaphore with initial value equal to the third argument.

Waiting a Semaphore

```
int sem_wait(sem_t *sem);
```

here sem is the address of the semaphore that will be locked until successful return or interrupt.

Checks the value of the semaphore. If it is less than or equal to zero, the calling thread will be blocked.

Increasing the value of a Semaphore

```
int sem_post(sem_t *sem);
```

where sem is the address of the semaphore to be incremented.

Increases the value of the semaphore, thus, increasing the number of available slots for thread execution.

Learning Experiences

Students will be given sample codes for demonstration

Assessment Tool

A programming exercise using mutex and semaphores

References

[1] "Multithreaded Programming (POSIX pthreads Tutorial)" <https://randu.org/tutorials/threads/>

[2] "Synchronization Threads with POSIX Semaphores "
<http://www.csc.villanova.edu/~mdamian/threads/posixsem.html>

[3] "Locks and Condition Variables"
<https://web.stanford.edu/~ouster/cgi-bin/cs140-spring14/lecture.php?topic=locks>

(C) Institute of Computer Science | UPLB
CC BY-NC-SA 4.0