IPC and Synchronization

Module 08

Reading

The module covers Section 2.3.1 - 2.3.7

We will not cover 2.3.8 - 2.3.10

The problem...

```
newml-image:code sfrees$ echo "1000" > account.txt
newml-image:code sfrees$ more account.txt
1000
newml-image:code sfrees$ ./rob
Theif stole total of 32862
Theif stole total of 3193
Theif stole total of 21233
Theif stole total of 2922
newml-image:code sfrees$ ./rob
Theif stole total of 17972
Theif stole total of 33343
Theif stole total of 20053
Theif stole total of 21283
newml-image:code sfrees$ ./rob
Theif stole total of 999
Theif stole total of 999
Theif stole total of 999
Theif stole total of 999
```

The output of this program is unpredictable

Unpredictable = bad.

Why is it unpredictable?

- In about a week, we'll learn about the **scheduler.**
    - The schedule interrupts running processes, and schedules others to run, using a **deterministic** model
    - While its deterministic from the OS's perspective - its **completely random** from an individual user program's perspective
- When sharing data, we must be aware that we could be interrupted at ***any time***

Is this really that bad?

- Its extremely likely we'll be interrupted while reading from disk (it's inherently a blocking call)
- But it's possible (less likely) to be interrupted anywhere..

```
a++;
```

```
a = a + 1;
```

```
LOAD  $r1, a
ADD   $r1, 1, $r2
STORE $r1, a
```

Like here.

Compound C++ statements

Actual assembly code...

Race Conditions

Any situation where the outcome of execution depends on the order in which multiple processes or threads execute.

- Race conditions don't crash programs - but when ***may*** lead to incorrect results
- Sometimes incorrect results crash programs.
    - Race conditions while deleting/creating memory

You must handle race conditions using synchronization mechanisms

Critical Section

- A critical section is an area of code where **shared data is manipulated**.
- It is an area in code where you do not want two or more threads to simultaneously be executing
- To do this, we provide ***synchronized*** access to this area of the code.
  - Methods to do this are called **critical section solutions.**

Requirements for Critical Section Solutions

1. No two processes may be simultaneously inside their critical regions. (Mutual Exclusion)
2. No assumptions may be made about speeds or the number of CPUs. (Ass u me)
3. No process running outside its critical region may block other processes. (Progress)
4. No process should have to wait forever to enter its critical region. (Bounded Wait)

Simple Solution

- Disable Interrupts on the CPU
    - OK for User Programs?
    - Long Critical Sections?
    - What about multiple processors?

Some possible solutions

- Lock Variables
- Turn Taking (spin lock)
- Peterson's Solution
- TSL Instructions (Hardware)

Some possible solutions

- ~~Lock Variables~~
- Turn Taking (spin lock)
- Peterson's Solution
- TSL Instructions (Hardware)

Some possible solutions

- ~~Lock Variables~~
- ~~Turn Taking (spin lock)~~ (OK sometimes)
- Peterson's Solution
- TSL Instructions (Hardware)

Some possible solutions

- ~~Lock Variables~~
- ~~Turn Taking (spin lock)~~ (OK sometimes)
- Peterson's Solution (a bit cumbersome)
- TSL Instructions (Hardware)

Some possible solutions

- ~~Lock Variables~~
- ~~Turn Taking (spin lock)~~ (OK sometimes)
- Peterson's Solution (a bit cumbersome)
- TSL Instructions (Hardware)

We need Abstractions

Using Peterson's solution, or figuring out if TSL is supported, is a pain.

Luckily, we can create abstractions (an API) to provide synchronization!

Mutex

- Mutex stands for **mut**ual **ex**clusion.
- Idea:

```
Mutex m;        // shared between threads
m.lock()        // employs a CS solution
critical work
m.unlock()      // employs a CS solution
non critical work
```

Avoiding Busy Waiting

- There's one problem with Peterson's solution and TSL:
- Waiting means constantly checking if we can stop waiting.
- Eats processor time!

Sleep and Wake

```
class Mutex {
    private boolean lock;
    private List asleep;

    public void lock() {
        if (lock) {
            asleep.push(this thread);
            sleep(this thread);
        }
        lock = true;
    }
```

```
    public void unlock() {
        lock = false;
        if (asleep.length > 0 ) {
            wakeup(asleep[0]);
            asleep.pop();
        }
```

Mutex in POSIX

```
#include <pthread.h>
pthread_mutex_t mutex;

pthread_mutex_init(&mutex, NULL);

pthread_mutex_lock (&mutex);
pthread_mutex_unlock (&mutex);

pthread_mutex_destroy(&mutex);
```

Mutex in Windows

```
#include <windows.h>
HANDLE CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,
  BOOL                  bInitialOwner,
  LPCTSTR               lpName
);

WaitForSingleObject(HANDLE mutex, INFINITE);  // down

BOOL ReleaseMutex(HANDLE mutex);

CloseHandle(mutex);
```

Mutex in C++ 11

```
#include <mutex>

// constructor initializes as unlocked
std::mutex my_mutex;



my_mutex.lock();

my_mutex.unlock();
```

Semaphore - a generalization

- You can think of a mutex as a **boolean** that can be atomically changed/read.
- A semaphore is an **integer** that can be atomically incremented and decremented

```
Semaphore s;
s.up();  // increments s, never blocks/sleeps
s.down();  // sleeps if s is 0, woken up when s >=0
```

Why Semaphore?

Semaphores can be used for mutual exclusion, but that's not really why they are useful...

Lots of problems can be modeled with semaphores - *read the producer/consumer segment of the textbook!*

Semaphores in POSIX

```
#include <semaphore.h>

sem_t sem;

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_post(sem_t *sem);  // up
int sem_wait(sem_t *sem); // down
int sem_destroy(sem_t *sem);
```

Semaphores in Windows

```
#include <windows.h>
HANDLE semaphore;
semaphore = CreateSemaphore(
                  NULL,              // default security attributes
                  MAX_SEM_COUNT,   // initial count
                  MAX_SEM_COUNT,   // maximum count
                  NULL);             // unnamed semaphore

WaitForSingleObject(semaphore, INFINITE);  // down
ReleaseSemaphore(semaphore,  // handle to semaphore
                 1,          // increase count by one
                 NULL);      // pointer to int if you want the previous value.
```

Creating a Semaphore in C++ 11

C++ doesn't provide a Semaphore structure, but you can create one using a combination of C++ synchronization primitives.

Outside the scope of this class - but take a look: [*http://en.cppreference.com/w/cpp/thread*](http://en.cppreference.com/w/cpp/thread)

Monitors

- Mutexes seem easy… but in practice they are not!
- The ordering of unlock/lock is crucial
- When using multiple mutexes, gets even trickier
- A Monitor is a **class** which synchronizes access to its private member variables

Monitor objects in C++

Some languages provide monitor support (Java, C#) - but it's not too terrible to implement in C++...

- Create a private mutex in class
- At the beginning of each public method, acquire the lock
- Release the lock before returning.

The value here is that now a user may work with your object without concern for race conditions on the object’s member variables.

Up Next...

We see the other side of things… scheduling

Please read section 2.4