Mutex and Semaphores

Mutex

- The hardware-based solutions are complicated and not accessible to application programmers
- We need to provide a higher level abstraction
- Achieved through mutexes, semaphores etc.

Mutex

- Variables -
  - A shared boolean variable named *available*
- Functions -
  - aquire() and release()
- Implementation

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;
}

release() {
   available = true;
}
```

```
while (true) {

    acquire lock

        critical section

    release lock

        remainder section

}
```

Mutex

P1	P2
acquire()	
available = false and acquired critical section	
	acquire() available = false and busy waiting
release() available = true	
	available = true Break while loop available = false and acquired critical section

Mutex : Advantages and Disadvantages

- Disadvantage :
  - Busy waiting : Wastes CPU cycle
- Advantage :
  - Busy Waiting! : Reduces context switching time
  - Ideally critical sections have short duration
  - Hence, uninterrupted busy waiting can be ideal

Spinlock

- Type of locks where the waiting process spins (uses a loop) to wait
- Behaves differently for two types of process:
  - Kernel process : All kernel processes trust each other, so, all spinlocks are uninterrupted
  - User process : Should not be uninterrupted always as not trusted. Usually, most OS (like Solaris, Mac OS X and FreeBSD) provide hybrid mechanism. Upto a time threshold, process is uninterrupted but can be interrupted after that limit.

Semaphore

- Mutex enable mutual exclusion
- What if several processes (> *n*) want to access *n* resources and one process can use one resource at a time?
- Cannot be achieved with binary mutex

Semaphore

- Two types -
  - Binary Semaphore : Works like mutex
  - Counting Semaphore : Used to control access to resources
- Implementation :

```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

```
signal(S) {
    S++;
}
```

Semaphore : Example

- Consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2 . Suppose we require that S2 be executed only after S1 has completed.
- Solution -

```
S1;
signal(synch);

 wait(synch);
 S2;
```

Semaphore : Disadvantage

- Suffers from busy waiting
- Solution : Put waiting process to sleep, wake up a sleeping process only when another process has relinquished the resource

Semaphore Implementation

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
            add this process to S->list
            sleep();
        }
}
```

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
            remove a process P from S->list;
            wakeup(P);
        }
}
```

Semaphore Implementation

- Number of processes : 4
- Number of resources : 2
- Consider FIFO waiting queue
- Consider the following order of processes to execute wait() : P1, P3, P2, P4
- P1, P2, P3 and P4 relinquish their resource after 3, 5, 3 and 2 timesteps respectively
- Using a semaphore, how many timesteps does it take for all process to finish?

Semaphore Implementation

- Class task : Implement Mutex with compare and swap operation
- Class task : Implement semaphore for producer and consumer process (Hint: You need three semaphores)