OPERATING SYSTEMS

Module3_Part3

Textbook : Operating Systems Concepts by Silberschatz

Mutex Locks

- The hard ware based solution Test and Set Lock for critical section problem is complicated and inaccessible to programmers.
- Another software tool used to solve critical section problem is Mutex Locks.
- We use murex locks to protect critical regions and avoid race conditions
- Process must acquire lock before entering the critical region and release the lock when it exits.
- The acquire function() acquire the lock and release function() releases the lock

Mutex locks

- Mutex has a boolean variable available whose value indicates lock is available or not

if lock is available call to acquire() succeeds; the lock is then considered unavailable

the process attempt to acquire unavailable lock is blocked until the lock is available

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

The definition of release() as follows

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

Mutex locks

- do {

  Acquire lock

  Critical section

  Release lock

  noncritical section

  } while(true);

Call to either acquire() or release() must be performed atomically

Solution to critical section problem using mutex locks

Mutex locks

The main disadvantage of mutex locks is **busy waiting**

while a process is in its critical section, any other process that tries to enter

its critical section loops continuously in the call to acquire()

This type mutex lock is also called spin lock ,since the process spins while

waiting for the lock to become available

The spin locks do have advantage that **no context switch** is required when

process wait on a lock(context switch may take considerable time)

When locks are expected to be held for short times spin locks are useful

Process synchronization using semaphore

- A **semaphore** is a synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities..
- A semaphore S is an integer variable that, apart from initialization, is accessed only

  through two standard atomic operations: wait () and signal ().

  wait () operation was originally termed P;

  signal() was originally called V.

Process synchronization using semaphore

The Definition of wait () is as follows:

```
wait(S) {
        while S <= 0
        ; // no-op
        s--;
        }
```

The definition of signal() is as follows:

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

semaphore

- All modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly.
- That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
- In addition, in the case of wait (S), the testing of the integer value of S (S <= 0), as well as its possible modification (S--), must be executed without interruption

semaphore

Usage

Two types of semaphores

counting semaphore: The value can range over an unrestricted domain.

binary semaphore:The value can range only between 0 and 1.

binary semaphores are known as mutex locks, as they are locks that provide

mutual exclusion.

Binary semaphore

- We can use binary semaphores to deal with the critical-section problem £or multiple processes. Then processes share a semaphore, mutex, initialized to 1. Each process *Pi* is organized as shown below

```
do {
   wait(mutex);

      // critical section

   signal(mutex);

      // remainder section
} while (TRUE);
```

Mutual-exclusion implementation with semaphores.

Counting semaphore

- Counting semaphores can be used to control access to a given resource consisting of a finite number o£ instances.
- The semaphore is initialized to the number of resources available.
- Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count).
- When a process releases a resource, it performs a signal() operation (incrementing the count).
- When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

Semaphore

- We can also use semaphores to solve various synchronization problems.
- For 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.

We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0, and by

inserting the statements

S1;

signal(synch) ;

in process P1

and the statements

wait(synch);

S2; in process P2.

Because synch is initialized to 0, P2 will execute S2 only after P1

has invoked signal (synch), which is after statement S1 has been executed.