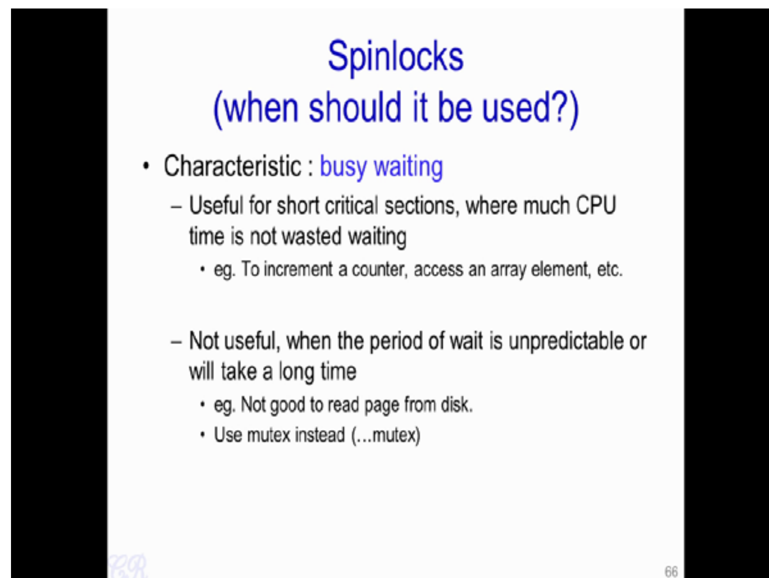
Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 06
Lecture – 28
Mutexes

Hello. In this video we will look at a Mutexes, which is a construct used to solve the critical section problem.

(Refer Slide Time: 00:25)



We will start with where we stopped off in the last video, with Spinlocks. Essentially, the main characteristic of spinlocks is that it uses busy waiting that is, we had seen that in order to have a lock there was a while loop and in that while loop the exchange instruction was continuously invoked and the while loop would only exit when the exchange instruction returned a 0.

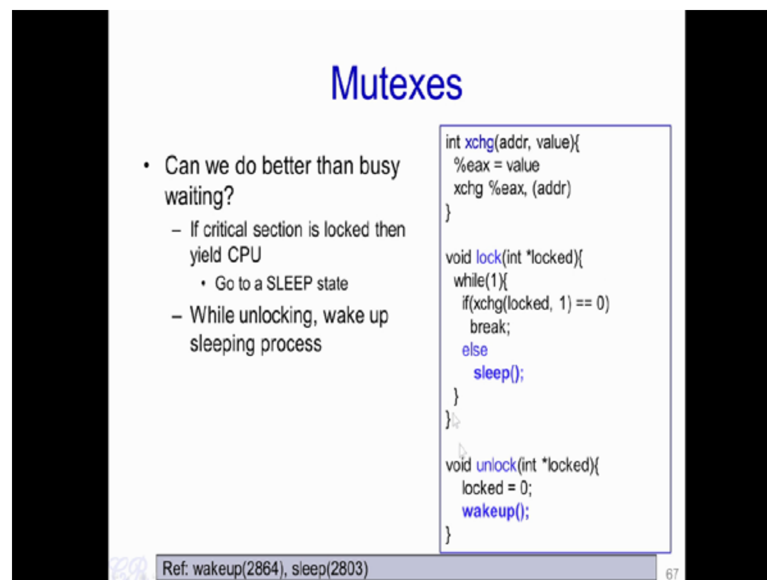
This busy waiting is not ideally what is required. Essentially, busy waiting causes the CPU cycled to be wasted, leading to may be things like performance degradation as well as huge wastage of memory.

So, where would we actually use Spinlocks? Spinlocks are useful when we have short critical sections and we know that we do not have to waste too much time in waiting. For

instance, if we just want to increment a shared counter then we could possibly use a spinlock or another thing is to access an array element then a spinlock would be preferred. Essentially, these things we assume could will not have too much of overheads. Therefore, even if another process is accessing the counter we are certain that the process is going to spend too much time incrementing the counter.

Therefore, the waiting process will not have to waste too many cycles waiting to enter into the critical section. However, spinlocks are not useful when the period of waiting is unpredictable or it will take a very long time. For instance, if there is page fault and resulting in a page of memory which is loaded from the hard disk into the main memory. So, this would take a considerable very long time and you do not want your process to be actually wasting CPU cycles during this entire operation. In such a case, we use a different construct called a Mutex.

(Refer Slide Time: 02:43)



This over here shows how typically the mutex is implemented. So, it again relies considerably on the exchange instruction which is used and just like the spinlock we have the lock and unlock and memory location which is shared between all processes.

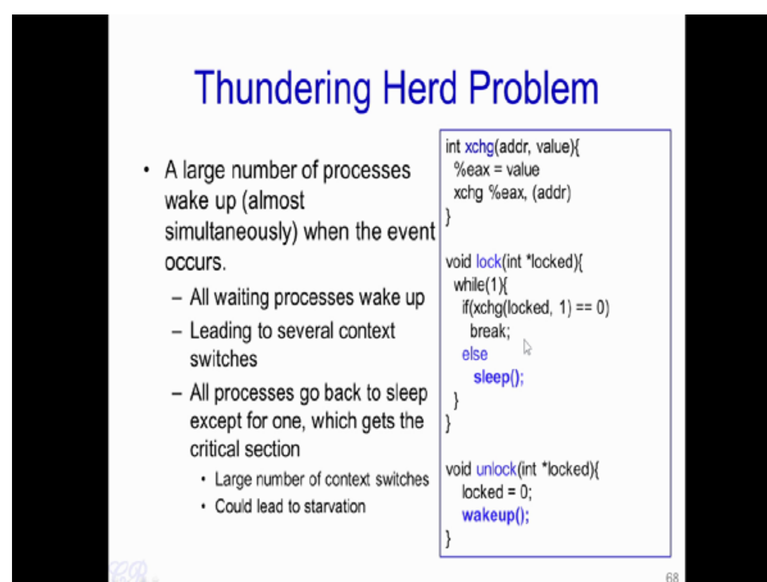
Now, in order to obtain the critical section process would need to invoke a lock and in this lock function we have a while loop. So, as in this spinlock case the exchanged instruction is invoked in the while loop and this instruction would either return a value of 0 or something not equal to 0 or typically 1. If the value is equal to 0, then we break from

this loop and that process would then have acquire the lock and execute in the critical section. However, if the exchange returns a value which is not 0 then we go into this else part and execute this function called sleep.

Now, this sleep function would cause the process to go from the running state into the block state. Essentially, the process is waiting for a particular operation to arrive. So, until this operation arrives the process will not get any CPU time. Now, this event which the sleep is waiting for is the wake up event. So, when other process invokes wake up, it will result in the sleeping process to be woken up from the block state and put on to the ready queue. Now, if it is lucky when it executes the exchanged instruction again it would get 0 and it would get into the critical section.

On the other hand, if it is unlucky it would execute the exchange instruction and get something which is non-zero and it would go back to sleep. And it would continue to sleep until woken up by another process. Essentially, we see over here that instead of doing a busy waiting as was done in spinlocks, in mutexes we put the entire process into a sleep state. The process will continue to be in a sleep state until it is woken up and when it woken up, it is going to try the lock again and if it achieves a lock then it enters into the critical section.

(Refer Slide Time: 05:34)



One issue with mutex is what is known as a Thundering Herd Problem. So, the thundering herd problem occurs when we have large number of processes. So, each of

these processes let us assume is using the same critical section and invokes the lock in order to enter into the critical section, and at the end of the critical section would invoke unlock which would then wake up another process waiting for the critical section. So, it would happen if we have large number of processes that, there are several processes present in the sleep mode while one process enters into the critical section.

So, when that process executes unlock it invokes wake up. This results in all the processes which are sleeping to be woken up. So, all these processes would then go from the block state into the ready queue and the scheduler would then sequentially execute each of these processes. So, each process is then going to continue its while loop and executes the exchange function. Out of all these processes because of the atomic nature of the exchange instruction only one process would acquire the lock and all other processes would go back into sleep state. So, this continues every time.

Every time, whenever an unlock is invoked by a process just completing its critical section, it will invoke wake up and it will result in all processes waiting on that mutex to be woken up and all, except one would go back to sleep. There would be exactly one process which would gain the lock and enter into the critical section. So, as a result of what we see that whenever there is wake up invoked there would result in several context which is occurring, in order that all the processes execute and check the exchange instruction again. Now, by the way exchanges implemented in the hardware all processes except one will enter into the critical section.

So, the issue over here and why it is called a Thundering Herd Problem is every time there is a wake up, there is a huge avalanche of context switching that occurs because a large number of processes are entering into the ready queue and this could lead to starvation.

(Refer Slide Time: 08:19)

Thundering Herd Problem

- The Solution
  - When entering critical section, push into a queue before blocking
  - When exiting critical section, wake up only the first process in the queue

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void lock(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
    else{
      // add this process to Queue
      sleep();
    }
  }
}

void unlock(int *locked){
    locked = 0;
    // remove process P from queue
    wakeup(P)
}
```

69

One solution to the Thundering Herd Problem is to modify the way mutexes are implemented by incorporating queues. In this implementation of a mutex, whenever the lock is invoked and the exchange instruction return something which is non-zero, the process gets added into a queue and then goes to sleep.

Now, when a process invokes unlock the sleeping process is removed from the queue and a wake up specifically for only that process is invoked. So, unlike the previous cases where all processes are woken up, in this case there is exactly one process which is woken up, so this process P which is specified here would wake up from sleep and since it is the only process which as woken up, it would typically go into this while loop, check the exchange and most likely it would get the lock and execute the critical section.

Similarly, when it unlocks, it would pick out the next process which is waiting in the queue and it will wake up only that process. Now, this second process would then enter into the critical section.

(Refer Slide Time: 09:42)

Locks and Priorities

- What happens when a high priority task requests a lock, while a low priority task is in the critical section
  - Priority Inversion
  - Possible solution
    - Priority Inheritance

Interesting Read : Mass Pathfinder
http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/mars_pathfinder.html

71

So, when we are talking about synchronization primitive such as a spinlocks and mutexes, it is important to also consider the case when a priority based scheduling algorithm is used in the operating system. Let us consider this particular scenario.

So, let us say we have a high priority task and a low priority task which share the same data and have critical section. Now, let us say that the low priority task is executing in the critical section and at this particular time, the high priority task request for the lock in order to enter into the critical section. So, the scenario we are facing here is that the low priority task is executing in the critical section, while during this time the high priority task invokes something like a lock and wants to enter into the critical section.

Now, the dilemma we are facing here is that we have a high priority task which is waiting for a low priority task to complete. So, this is known as the Priority Inversion Problem. Essentially, we have something important - a task which is important and given a high priority and it is waiting for a lower priority task to complete its execution. And if you look at this particular link present here, you will see quite an interesting case where such a priority inversion problem had occurred, essentially with a path finder.

One possible solution for the priority inversion problem is known as the Priority Inheritance. So, essentially in this solution whenever a low priority task is executing in the critical section a high priority task request for that critical section, what happens is that the low priority task is escalated to a high priority. Essentially, the priority of the

low priority task becomes equal to that of the high priority task. The low priority task then would execute with this high priority until it releases the critical section. So, this would ensure that the high priority task would execute relatively quickly.

Thank you.