

Information Security-3
Prof. V Kamakoti
Department of Computer science and Engineering
Indian Institute of Technology Madras
Basics of Unix and Network Administration
Operating Systems
Mod02 Lecture 08
Module 8: Process Synchronization

So, welcome to module 8 and we will be talking about process synchronization. What you mean by synchronization? Synchronization is that when two processes want to access say, a resource a sheared resource, there should be certain discipline that needs to be enforced while accessing that resource and what is that discipline becomes a subject matter of synchronization and if that discipline is not adhered to then the process is basically can access that resource in such a way that there could be some inconsistency in the entire operation.

When you look at the entire operating system, there are lot of sheared resources for example, there is something called buffer cache, there is something called the super block in the file system there are many many sheared resources and these resources are sheared by multiple processes at given point of time and so process synchronization becomes the bread and butter of how to maintain consistency of access of these sheared resources among these processes whenever they try to access them. So that becomes the subject matter.


So we will introduce process process synchronization not from a file system point of not from examples derived from file system, but using some simple concepts like you know shearing a buffer etcetera and then you will introduce the concepts and understanding this concept is also very important from a security perspective, because of lot of leaks of information do come race conditions could be exploited to basically hack into your system and these are very important concepts that need to be understood specifically for the security community too.

(Refer Slide Time: 2:00)

Objectives

2

- About getting processes to coordinate with each other.
- About how processes work with resources that must be shared between them.
- About how we go about acquiring locks to protect regions of memory.
- About how synchronization is really used.



Operating Systems


So the objective of this particular module is about getting processes to coordinate with each other about how process work with resources that must be sheared between them and about how we go about acquiring locks to protect regions of memory. Why do you need to protect because if I do not protect two processed can access at the same time and basically go an make the content of that memory inconsistent with what the program is actually doing we will see some examples and how we synchronization is really used and it is really implemented. So that is these are all the objectives of this particular module.

(Refer Slide Time: 2:34)

But Why Synchronize?

3

- Multiple process compete for a common resource
 - At times they may co-operate to synchronize some activity or share a common resource.
 - Concurrent access to shared data may result in **data inconsistency if there is no controlled access**
 - Maintaining **data consistency** requires mechanisms to ensure the orderly execution of (**cooperating**) processes
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
 - Information sharing
 - Computation speed-up
 - Modularity
 - Convenience



Operating Systems

So but why should be synchronize? This is, because I am shearing the resources, when I am shearing the resources I need to adhere to some discipline if I do not have that discipline then

what will happen is that the state of your resource will become inconsistent, the data actually becomes inconsistent, we will give you some examples of that as we progress and there is we have to share resources, because we are looking at a cooperating environment where multiple processes try to work together to basically solve a particular problem, we have already seen why more than one process can work together and then we can do scheduling we have seen in the previous module.

In the context of scheduling, we found that if two processes are working together then I can basically merge the CPU time of one process with the IO time of another, essentially giving you a lot more computation speed up and a lot more hardware utilization etcetera. In addition, I could have certain things like modularity and convenience, for example, if I have a particular functionality, I would like to keep that functionality with respect to a particular module so that tomorrow when I upgrade software.

So software as five different functionalities, I will like to realize it has five different modules, because tomorrow when the software is actually upgraded and say functionality 3 changes, the modules corresponding functionality 1, 2, 4 and 5 can be retained and only the third module needs to be changed and this becomes very easy for me to release the next question. So modularity is also very important for creating these type of large scale software and that basically comes by isolating different loosely related modularity, separate threads or processes and building up a system and if that needs to be executed then basically what happens is that each module becomes a thread and or a process and these processes work in a cooperating way to solve the problem and then hence we will end up with (4:44) with a cooperative process environment.

(Refer Slide Time: 4:47)

EX:-

What may happen when process co-operate?

- Assume we have a process that **produces** information and another process **consume** this information and share a common fixed size **buffer** to store this information.
- We can have two types of buffer
 - ▣ **unbounded-buffer** places no practical limit on the size of the buffer
 - ▣ **bounded-buffer** assumes that there is a fixed buffer size
- The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

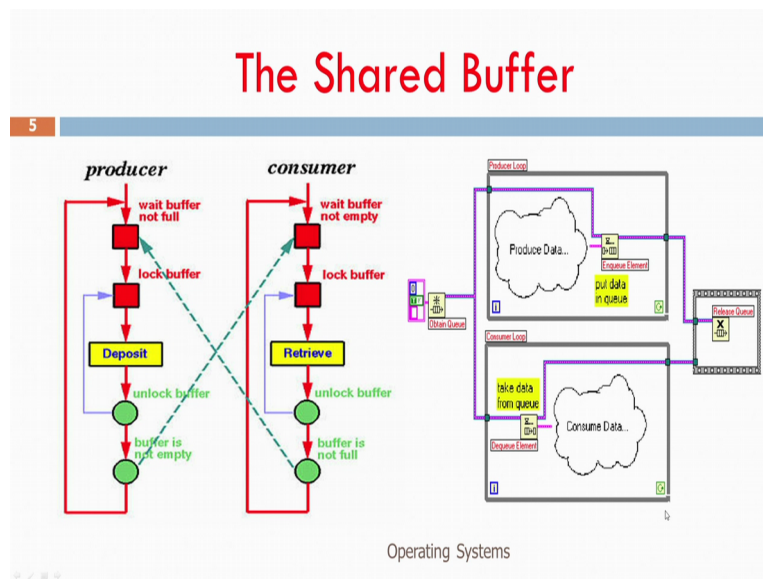
Operating Systems

So what may happen when process cooperate, right? So we will take up a very simple problem called buffer shearing problem. So when one and there is producer who produces into that buffer, there is a consumer who consume from the buffer and so let me say that there is a bounded buffer. So I have n locations. Now, a (())(5:10) producer should not produce into a buffer that is already full, because you lot (())(5:15) have location to store it and the consumer should not consume from a empty buffer, he (())(5:21) will basically land up (())(5:22) with null. So this is the access rule, right.

So if I have one buffer that I am going to shear. There is a consumer who consumes from the buffer, there is a producer who produces into that buffer, the producer should not produce into buffer that is full and a consumer should not try and consume or remove data from that buffer when the buffer is empty. This is a rule that we need to implement. So this is where I need synchronization. So when the produce when the buffer is full I should not allow the producer to add an entry there, when the buffer is empty I should not allow the consumer to consume from it.

So this rule has to be implemented and this is what we call as synchronization. So the producer is a process, a consumer is another process both of them are working together in a cooperative way and there is they are trying to shear a resource in this case a buffer and there is a discipline that I am basically enforcing in this context where you know the producer produces and the consumer consumes, producer should not produce into a full buffer, a consumer should not consume from an empty buffer. So this is what we are trying to achieve here.

(Refer Slide Time: 6:40)



So this is the entire diagram for this. So what will the producer do? So if you look at the extreme left, the producer will wait for the buffer to be not full if the buffer is not full he goes across the orange block then he will go and lock the the buffer, then he will deposit something then he will unlock the buffer and if the buffer is not empty he will go and trigger something, right and then he goes back. So the two green cross lines we will just discuss it a little later, but what does the producer to (7:18) it just two orange boxes, it will wait the first orange blocks will wait till the buffer is not full.

So if the buffer is full it will wait till the buffer is not full. So if the buffer is full, it will keep waiting there so and if the buffer is full, it will be waiting and who will go and say that you can proceed, it is waiting there meaning that process is in a suspended state. Now who is go into to go and say you can proceed. The consumer is go into go and say, so when the consumer consume something from the buffer, certainly there will be one location which is the not true or (7:59) not full because it has consumed one fellow from there, so the at least one location will be there. So that is the fellow who triggers an event to the producer saying (8:09) the buffer is not full for sheer. So that is a green line that goes from the consumer to the producer.

So when the producer comes it finds that the buffer is full, it waits on the first orange box till the green signal comes from the consumer saying hey (8:25) the buffer is not full now, you can go and produce, immediately it will go and lock the buffer and it will deposit and then unlock the buffer and then it will come back to he till now go and say to the consumer

buffer is not empty, because it has produced something. So now it will go and tell the consumer buffer is not empty and it will go back, right.

For the consumer, it will wait till the buffer is not empty, right because if the buffer is empty, it has to wait. So it will wait till the buffer becomes not empty. So if the buffer is empty it will wait there and there is a green signal coming from the producer which says (9:06) now I have produce one thing, certainly the buffer is not empty. So then it is go, it will lock the buffer, it will retrieve that data and then it will unlock the buffer, again it comes back and he says now, I have at least one location is empty there. So it will say buffer is not full tat signal it will send to the producer. So the two signals that pass from the consumer to the producer and from the producer to the consumer. These are the synchronizing action by having these two signals here we basically see to it, that the producer does not produce into a full buffer and the consumer does not consume from an empty buffer both will land up in a inconsistent state.

Now also note that whenever I am depositing whenever the producer is depositing something into the buffer or when the consumer is retrieving something from the buffer. It basically locks the buffer, because when I am producing into the buffer, I do not want the consumers to start consuming, I want to finish that deposition, I want to deposit the entire thing and then only allow the consumer to consume. So there is a lock and then unlock. So when there is when the buffer is locked what will happen? The consumer cannot cross that locked buffer when the producer has cross the lock buffer the consumer cannot cross the lock buffer, it will wait till the producer unlocks then only the consumer can go and start retrieving.

So at the same point of time the producer and the consumer cannot be inside the yellow blocks meaning deposit and retrieve when the producer is depositing the consumer cannot retrieve, when the consumer is retrieving the producer cannot deposit. So that is also taken care otherwise half the way he is depositing this fellow will start retrieving it then there could be an inconsistent state. So what you see on the right hand side is basically an implementation of this entire sheared infrastructure. So there is one buffer that is sheared by the producer and consumer and this is how the access to this is basically synchronized. So the two green lines basically tell you how we synchronize these two actions of not producing into a full buffer and not consuming from an empty buffer.

(Refer Slide Time: 11:32)

Solution?

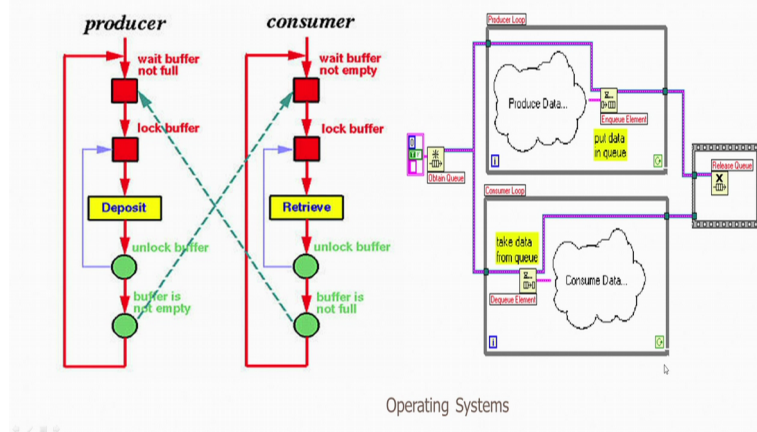
6

- Producer goes to sleep or discard data if the buffer is full.
- When consumer removes an item from the buffer, it notifies the producer who starts to fill the buffer again.
- Consumer goes to sleep if it finds the buffer to be empty.
- When the producer puts data into the buffer, it wakes up the sleeping consumer.

Operating Systems

The Shared Buffer

5



Operating Systems

So the solution essentially is the producer goes to sleep or discard data if the buffer is full, (())(11:37) in this case, it goes to sleep. So when that is the first rectangle on the producer side, the orange rectangle. What you mean by the wait it goes to sleep when consumer removes an item from the buffer it notifies the producer who starts to fill the buffer again and that notification goes through this green line from the consumer to the producer. Similarly, the consumer goes to sleep if it finds the buffer to be empty which is the first shell (())(12:03) red rectangle on the consumer side and whenever the producer puts data into the buffer it wakes up the sleeping consumer.

(Refer Slide Time: 12:12)

Ex – What may happen when we have concurrent process :- Race Condition

Assume a shared integer c, initialised to 42, and one process wanting to increase, the other to decrease the value.


```
P1 {  
    shared int c;  
    c++;  
}  
P2 {  
    shared int c;  
    c--;  
}
```

Implementation of c++, and c--:

```
load &c reg1  
inc reg1  
store reg1 &c  
load &c reg2  
dec reg2  
store reg2 &c
```

Interleaved operation of processes P1 and P2:

P1: load &c reg1	→	reg1 = 42
P1: inc reg1	→	reg1 = 43
P2: load &c reg2	→	reg2 = 42
P2: dec reg2	→	reg2 = 41
P2: store reg2 &c	→	c = 41
P1: store reg1 &c	→	c = 43



Operating Systems

Now let us go and make it much more interesting. So let us say, there are two programs p1 and p2 right. These are C program there is a both this program p1 and p2 share one integer call C like they share buffer disc shares an integer call C. The program p1 increment C the program p2 decrement C, assuming that p1 execute, so immediately followed by p2, the value of C should remain the same. Let us say before execution of p1 the value of C is 4 or in this case, let us say the value of C is 42 right when p1 executes and then p2 executes, the answer should again remain the 42, but what we will show now is that when p1 executes followed by p2 in some interleaved fashion, the answer can become 43 as what we are going to show now.

Now what do you mean by increment C plus plus? Please note that, that C plus plus that you see there in the red on your left hand side, essentially is three assembly instruction that is incrementing an integer is three assembly instructions the three assembly instructions that you see here, load the content of C into register 1 increment register 1 store the content of register one back to C, right. This is the syntax; let us keep this as syntax.

Similarly, for C minus minus load the content of the C on to register 2 decrement register 2 and store the content of register 2 back to C, correct. So this is how increment works, this is how decrement works. Now these are 3 assembly instructions. Now what will happen when I am going to execute this course? Let us say the p1 the first 2 instructions of this is executed. Say load and percent C register 1 that means register 1 gets the value 42 because C is initialize to 42.

Now see the program how the program is executing of the bottom of your screen here, first I will load C into register 1. So 42 gets into register 1, I will increment, so register 1 becomes 43, before I could store please note that the p2 starts executing, because let us say, it is round robin algorithm. So there is a context switch. So before so the time quantum allotted to p1 is over now I go to p2. Now p2 load C reg2 happen, so reg2 again gets 42, because C is not updated C is still 42 so reg2 will get 42 and now decrement reg2, so reg2 gets 41 then I store the value of reg2 on to C, so C gets 41.

Now the context switch happens when the context switch happens back to p1 what happens it will now? It is in the third instruction, the first two instructions are already done, the third instruction is store reg1 and percent C so C will get 43. So C plus plus C minus minus executed one after another should maintain the value of C as 42, but when I compile that code C plus plus and C minus minus, I get 3 assembly instructions for C plus plus and 3 assembly instructions for C minus minus and when I am started executing this program and if they get interleaved into in between then what happens I get an inconsistent value for C. do you follow? This is the problem of shearing data and this is race condition, because who (()) (15:46) finish first, 2 fellows are racing, right p1 and p2 are racing with each other and because of that this entire problem has happen.

Now what is the solution to this? When p1 is executing these three instructions, p2 should not be allowed to execute those 3 instructions, when p2 is executing those 3 instructions, p1 is should not be allowed to execute these 3 instructions if we had done C plus plus completely and then C minus minus or if we have done C minus minus completely and then C plus plus then this problem should not have happen, the point is the to ensure that what should I do? I should execute p1 first that 3 assembly instructions corresponding to p1 first, I have to finish that then I have to start the assembly instructions with free (())(16:39) p2 or I should finish of the assembly instructions with respect to p2 first and then start assembly instructions with respect to p1, right that means these 3 instructions and these 3 instructions of p1 and p2, they form the critical section for p1 and p2.

So if we look at some of the modern books, critical section means one single code which is executed by multiple programs that is the feel you get, in some of the operating system books that is not the case, right. There is a critical section for every program and they are different, but they are all bound by one level. So for the purpose of executing, this multi-threaded program which essentially has two threads namely p1 and p2. There is one critical section

spawning between those (17:36) program, the critical section of p1 is these 3 instructions that you see on the left, the critical section for p2 are the 3 instructions you see on the right and these two critical sections are linked, in the sense that when p1 is executing the critical its critical section p2 should not go into the corresponding associated critical section when p2 is executing its critical section p1 should not go into its associated critical section you are getting this point.

So when you look at all these modern books they will be one block called critical section and almost every student I have interviewed so far in the last 16 years as old (18:22) that there is one critical section, which two fellows will shear, no the code is different, again I repeat p1s critical section has the code on your left hand side, p2s critical section has the code on the right hand side when p1 is executing its code, which is different from the code of p2, p2 should not go and executes its associated code and vice versa and if that is ensured then this type of an inconsistency or race conditions will not happen, right and that is what we meant by locking and unlocking.

So imagine that very simply that before coming into this p1, these 3 instructions it will go and lock and the end of this it will go and unlock. Similarly, p2 will also put a lock and unlock before and after this C code. So when I lock, so p1 locks and then when p2 also tries to lock, it will say somebody else has locked and it will ask to wait and when I unlock then I go and release the lock. So p1 can now lock and go in, but when I unlock what is happened I have completely finished executing those 3 instructions, I unlock only after I finish the 3 instruction by this only one fellow will be allow to go into the critical section while the other fellow will be waiting on its critical section, please note right and when it comes and executes its critical section this fellow should be out of that critical section and you have to wait it if you want to enter here again he has to wait.

So the notion of allowing only one process to execute on its (20:06) critical section while the other process has to wait and vice versa and it extends to n processes if all n processes shear one resource each one will have its own critical section and exactly one can be in each critical section to other others have to be out of their respective critical section. This property is called mutual exclusion, right and so the entire process synchronization revolves around how do we achieve this mutual exclusion.

(Refer Slide Time: 20:38)

Solution?

8

- **Mutual exclusion(Through Locks)**
 - ▣ A synchronization mechanism to avoid race conditions by ensuring exclusive execution of **critical sections**.
 - ▣ This ensures that only one process is doing certain things at one time.
- **Critical section:** a section of code which reads or writes shared data.
- This is the section of code which is common to ***n* cooperating processes**, the processes ***may be accessing common variables***

Operating Systems

Ex – What may happen when we have concurrent process :- Race Condition

7

Assume a shared integer *c*, initialised to 42, and one process wanting to increase, the other to decrease the value.

```
P1 {                                P2 {
  shared int c;                    shared int c;
  c++;                              c--;
}
```

Implementation of *c++*, and *c--*:

```
load &c reg1                        load &c reg2
inc reg1                           dec reg2
store reg1 &c                      store reg2 &c
```

Interleaved operation of processes P1 and P2:

```
P1: load &c reg1    → reg1 = 42
P1: inc reg1       → reg1 = 43
P2: load &c reg2    → reg2 = 42
P2: dec reg2       → reg2 = 41
P2: store reg2 &c  → c = 41
P1: store reg1 &c  → c = 43
```

Operating Systems

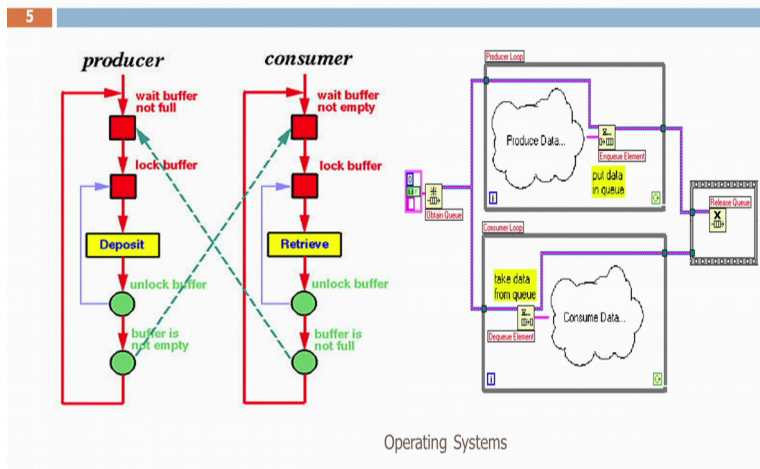
Solution?

6

- Producer goes to sleep or discard data if the buffer is full.
- When consumer removes an item from the buffer, it notifies the producer who starts to fill the buffer again.
- Consumer goes to sleep if it finds the buffer to be empty.
- When the producer puts data into the buffer, it wakes up the sleeping consumer.

Operating Systems

The Shared Buffer



So one immediately we have seen how we achieve this mutual exclusion? We have seen the notion of lock in the previous one but one slide. So I lock the buffer and I unlock I get this mutual exclusion, right but it is not so easy. now we will see how to achieve this mutual exclusion but the need for a mutual exclusion is that I need to have a synchronization mechanism to avoid race conditions by ensuring exclusive execution of critical section that is motive of mutual exclusion and this exclusive execution of critical section where in only one fellow can enter and the other fellow should not enter. This is among n cooperating processes I need to get method for that.

(Refer Slide Time: 21:30)

From O.S POV

- 9
- Utilize Semaphores (or) Monitors
 - Synchronized counting variables.
 - a shared integer variable that cannot drop below zero.
 - Formally, a semaphore comprises:
 - An integer value
 - Two operations: P() and V()
 - P() (Decrement value)
 - While value == 0, sleep
 - V() (Increment value)
 - If there are any threads sleeping waiting for value to become non-zero, wakeup at least 1 thread
- Operating Systems

From the OS point of view, this is achieved using something call semaphores or monitors. What is a semaphore? Semaphore is a sheared entity. It has an integer value and it has, it can

be decremented, it can be incremented, when I decrement a semaphore and its value becomes zero, I sleep when increment is just go and increase its value; I said there is an integer value right. It is increase its value by 1, right and the when the when I am incrementing before I increment, I check if it is zero. If it is zero then I go and wake up all the processes that are waiting for the semaphore, okay.

So let us understand a semaphore has two operations. It has an integer value and it has two operations P and V, P means decrement V means increment, decrement what I do I decrement reduced and the integer value I reduce it by one and if that value becomes zero I put that process which is trying to decrement to sleep. Similarly, when a increment means I go and increment that integer value and before incrementing I check if it is zero, if it is zero then there will be some processes that are sleeping, I wake the all of them and then I go in (()) (22:43). So these are the three operations and semaphores can be used extensively for achieving this synchronization that we have done in the past.

(Refer Slide Time: 22:53)

Semaphores

10

- Binary semaphores can be used to implement mutual exclusion
 - Initialize counter to 1
 - P == lock acquire
 - V == lock release
- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
 - Also known as *mutex locks*

Operating Systems

There also a binary semaphores, which basically does not have a value, but it is basically the integer values in the range zero and one. it does not have an integer value; it has a bit as that value. So it basically uses zero and one and these binary semaphores are also called as mutual exclusion locks. A lock is something like you have locked or unlocked, there is no value for it.

(Refer Slide Time: 23:17)

Critical Section

11

A section of code, common to n cooperating processes, in which the processes may be accessing common variables.

A Critical Section Environment contains:

- **Entry Section**:- Code requesting entry into the critical section.
- **Critical Section** :- ,Code in which only one process can execute at any one time.
- **Exit Section** :- The end of the critical section, releasing or allowing others in.
- **Remainder Section** :- Rest of the code AFTER the critical section.

```
do {  
    while ( turn ^≠ i );  
    /* critical section */  
    turn = i;  
    /* remainder section */  
} while(TRUE);
```

Entry Section
Critical Section
Exit Section
Remainder Section

Semaphores

10

- Binary semaphores can be used to implement mutual exclusion
 - Initialize counter to 1
 - $P ==$ lock acquire
 - $V ==$ lock release
- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1; can be simpler to implement
 - Also known as *mutex locks*

Operating Systems

From O.S POV

9

- Utilize Semaphores (or) Monitors
 - Synchronized counting variables.
 - a shared integer variable that cannot drop below zero.
- Formally, a semaphore comprises:
 - An integer value
 - Two operations: $P()$ and $V()$
- $P()$ (Decrement value)
 - While value $== 0$, sleep
- $V()$ (Increment value)
 - If there are any threads sleeping waiting for value to become non-zero, wakeup at least 1 thread

Operating Systems

Solution?

8

- **Mutual exclusion(Through Locks)**
 - ▣ A synchronization mechanism to avoid race conditions by ensuring exclusive execution of **critical sections**.
 - ▣ This ensures that only one process is doing certain things at one time.
- **Critical section:** a section of code which reads or writes shared data.
- This is the section of code which is common to ***n* cooperating processes**, the processes ***may be accessing common variables***

Operating Systems

Ex – What may happen when we have concurrent process :- Race Condition

7

Assume a shared integer *c*, initialised to 42, and one process wanting to increase, the other to decrease the value.

```
P1 {                               P2 {
    shared int c;                  shared int c;
    c++;                           c--;
}
```

Implementation of *c++*, and *c--*:

```
load &c reg1                       load &c reg2
inc reg1                           dec reg2
store reg1 &c                      store reg2 &c
```

Interleaved operation of processes P1 and P2:

```
P1: load &c reg1    → reg1 = 42
P1: inc reg1       → reg1 = 43
P2: load &c reg2    → reg2 = 42
P2: dec reg2       → reg2 = 41
P2: store reg2 &c  → c = 41
P1: store reg1 &c  → c = 43
```

Operating Systems


Now as we see that a critical section as in 3 parts as we saw in the program also in the previous thing. A critical section whatever we see as 3 instructions is the critical section for the p1 and also on the left for p2, but there something before it and then there is a entry into the critical section then there is exit in the from the critical section and then there can be reminder, so the in a huge code there will be some part which is a critical section and there will be entry criteria for the critical section, there will be some exit criteria for the critical section and then they will be entry criteria for the critical section, they will be some exit criteria for the critical section and then they will be remaining portion of the code, so that is what we have seen here, so they will be an entry section for the critical section then the critical section itself, then an exit section for the critical section and they will be something called reminder section, which is the reminder of all the code. The entire code minus the critical section is the reminder section.

(Refer Slide Time: 24:11)

Reader's Writers Problem

12

- A data set is shared among a number of concurrent processes
 - Readers – only read the data set; they do not perform any updates
 - Writers – can both read and write
- Problem – allow multiple readers to read at the same time; only one single writer can access the shared data at the same time
- Shared Data
 - Data set
 - Semaphore `mutex` initialized to 1
 - Semaphore `wrt` initialized to 1
 - Integer `readcount` initialized to 0



Operating Systems

Now we will explain this synchronization true what we call as the readers writers problem which is very very interesting. So there is a sheared data and multiple readers can read from that data, right at the same point of time multiple readers can read from the data, but only one fellow can write into then data, right at any point of time and when somebody is writing into that, nobody can read. So the three conditions are at any point of time, multiple readers can be reading no one can be writing should write and at any point, one fellow can write, more than one cannot write only one fellow can write and when that fellow is writing nobody else can read, right.

So these are the 3 types of criteria that we want to put and let us see, how synchronization how do you achieve this. So this is called the readers writers problem. So what are there, there is a data which people will read and write may be one integer variable or whatever and then we will be using 3 semaphores, one is called mutex initialize to 1, wrt which is initialize to 1 and then we will have an integer call read count which is initialize to 0, so these are all the shared data.

So I have a dataset which people will read and write from, the processes will read and write from. There is a semaphore call mutex which is initialize to 1, there is a semaphore call wrt which is initialize to 1 and there is a integer which is called read count which is initialize to zero. The read count actually it tells, how many readers are currently reading the sheared data. Read count essentially says, how many readers are currently reading the sheared data.


(Refer Slide Time: 25:57)

Reader's Writers Problem

12

- A data set is shared among a number of concurrent processes
 - Readers – only read the data set; they do not perform any updates
 - Writers – can both read and write
- Problem – allow multiple readers to read at the same time; only one single writer can access the shared data at the same time
- Shared Data
 - Data set
 - Semaphore `mutex` initialized to 1
 - Semaphore `wrt` initialized to 1
 - Integer `readcount` initialized to 0

Operating Systems



Reader Writers Problem

13

Writer:

```
wait(wrt);
...
perform writing;
...
signal(wrt);
```

Reader:

```
wait(mutex);
readcount := readcount + 1;
if readcount = 1 then wait(wrt);
signal(mutex);
...
perform reading;
...
wait(mutex);
readcount := readcount - 1;
if readcount = 0 then signal(wrt);
signal(mutex);
```

- A file is to be shared among several processes.
- Some processes read, others write to the file.
- Concurrent reading is allowed, but concurrent writing or reading/writing is not.

Operating Systems

Now let us do this, the code for writer is, note that there could be in multiple readers and multiple writers, there will be n readers and n writers, not one reader or one writer, there can be n readers and n writers. So one example for that is a file is to be shared among several processes, many processes can read at the same time, many processes will try to write, but when it is writing only one fellow should be allowed, after that other fellow can be allowed, but what point of time one point of time only one fellow can be allowed to write, but many fellows can be allowed to read the file, right. So there is a direct translation of the readers writers problem to a practical problem, okay.

Now this is the code, I have `wrt` as a semaphore, which will , so what I do is when I want to write, I just say `wait wrt`, `wrt` is a semaphore, so when I say `wait wrt`, it is initialize to 1, so

when I say wait it may become 0, okay. Now if somebody wants to write, he will also execute wait wrt when wrt is 0 he will go to sleep, so I will perform writing and then I will do signal wrt, when I do signal wrt what will happen that will be incremented from 0 to 1, since it is incremented from 0 to 1 if somebody is waiting, those fellows will also be woken up, right.

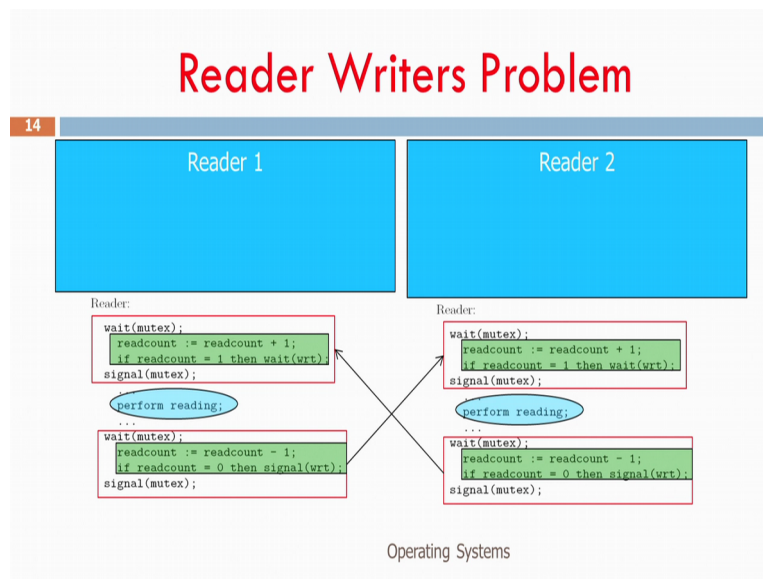
So wait wrt and signal wrt will ensure that only one fellow can write at any point of time; whoever executes wait wrt first will execute that fellow, he will see that wrt is one and he reduce it to zero and that fellow will come and start performing writing, the other fellows who will (())(27:51) to execute wait wrt later, they will all wait, after this fellow finishes he (())(27:57) signals wrt then those fellows will start executing, okay by this what just by putting wait wrt and signal wrt, I have ensured that only one process can write, more than one process cannot write at the same time, right.

Now this wait wrt and signal wrt are atomic instructions, atomic instruction is that when I am executing wait, it will be completely executed before something else can get executed. So when I start executing wait, I will finish the execution of wait before anybody else can do anything, so by this 2 processes who want to write into the sheared data, cannot execute wait at the same time and get access into the perform writing part, right.

So only one fellow, so if two fellows who want to execute write the first fellow the moment he says, wait write, he will finish of wait write then only he can, he will give the control to the next fellow. So I will do wait write, the other fellow want to do wait write, when I get (())(29:08) of wait write, I will start executing wait write, I will finish that wait write then only I will give the control to the other fellow, because wait write wait is a atomic instruction that is what we mean by atomic instructions, so when I say wait, so essentially I do that wait and if write is 0 wrt is one, I make it 0 and I start performing the writing while other fellow, executes wait after me and you will find that wrt is 0, so essentially (())(29:38) go and sleep. Now when I finish writing then I say signal then I allow him to go and continue, right. So this is how, so I have ensured that no two fellows can write at the same time, right.

Now the other thing that we need to ensure is when I am writing, no fellow should read and when somebody is reading no fellow should be allowed to write, when I am writing, the writer alone will be there, no other fellow will write no other fellow will read also and when I am reading, no other fellow will be allowed to write, correct you are getting this. So now how do we achieve this, the reader problem that is shown below, right we will go ahead and go to the next slide and see how it is achieved.

(Refer Slide Time: 30:26)



Reader Writers Problem

13

Writer:

```
wait(wrt);
...
perform writing;
...
signal(wrt);
```

Reader:

```
wait(mutex);
readcount := readcount + 1;
if readcount = 1 then wait(wrt);
signal(mutex);
...
perform reading;
...
wait(mutex);
readcount := readcount - 1;
if readcount = 0 then signal(wrt);
signal(mutex);
```

- A file is to be shared among several processes.
- Some processes read, others write to the file.
- Concurrent reading is allowed, but concurrent writing or reading/writing is not.

Operating Systems

First and foremost if you look at the reader code down, there is something call read count, what will read count tell you? Read count basically tells that number of readers that are currently reading, right and please, note that when I want to read, first I increment read count, then I perform the reading then I decrement read count and go off, correct. So that is what I am doing say, read count equal to read count plus 1, perform reading, read count equal to read count minus 1 get away, right. Since multiple fellows can read at the same time, please note that when one fellow is doing read count equal to read count plus 1, another fellow can do read count equal to read count minus 1, one fellow does read count plus plus at the same time another fellow can do read count minus minus and if a same variable incremented by one process and it is decremented by another process, already we have seen that there is race condition.

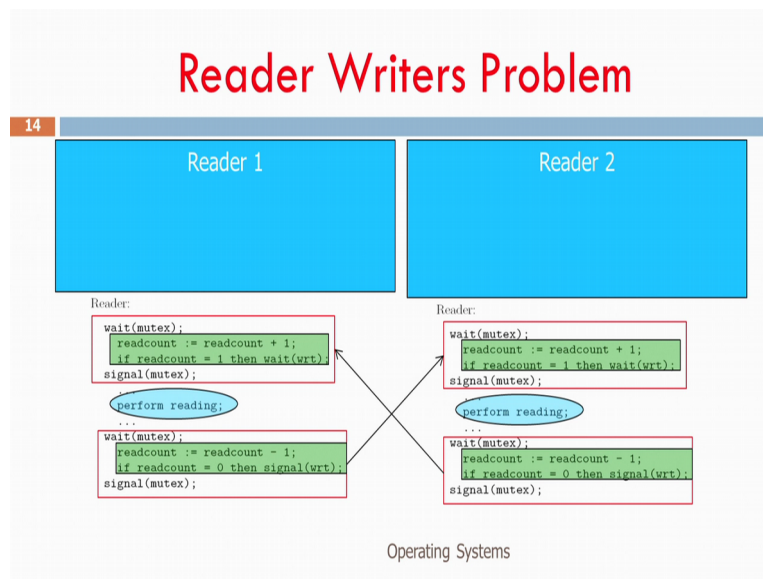
So whenever I am incrementing read count or whenever I am decrementing read count I want to protect it so that only one fellow can increment read count and only one fellow can decrement read count, so what I do there? So I use that semaphore called mutex, so I say wait mutex read count equal to read count plus 1 signal mutex, perform reading, wait mutex read count equal to read count minus 1 signal mutex. So there are multiple readers, if any of the reader is in one either incrementing or decrementing, he would have made mutex to 0 then only he can enter.

So when I want to enter either read count equal to read count plus 1 or read count equal to read count minus 1, one of those two sections either on the top before reading or on the bottom after reading, if you want to enter my mutex value should be 1, right and the moment I enter, I will make the mutex to 0. So the all the fellows who want to enter either of these sections, they may all the readers can be in any either one of these 3 sections, it can be either on the entry part (32:38) where it is incrementing read count or it can be actually reading or it can be on the exit part where it is decrementing reader 1.

So if any of the process wants to enter any of these 2 mutual exclusion section, they cannot enter unless I finish my operation, if I get hold of mutex, no other process can increment or decrement read count until I finish, the incrementing or the decrementing of the read count, by this read count, there is a serialization of the way by which I keep on increasing or decreasing the read count, if I do not put that mutex before and after signal wait and signal mutex before read count equal to read count plus 1 and after read count equal to read count minus 1 or again if I do not put wait and signal before in between and read count equal to read count minus 1, if I do not put that wait and signal there, then what will happen, they will be a race condition on the way read count is managed.

So the read count will become inconsistent, read count should tell me how many fellows are reading, but if I do not put that wait and signal mutex on both ends then read count can become an inconsistent value as we had demonstrated earlier (33:57) green you are either following this.

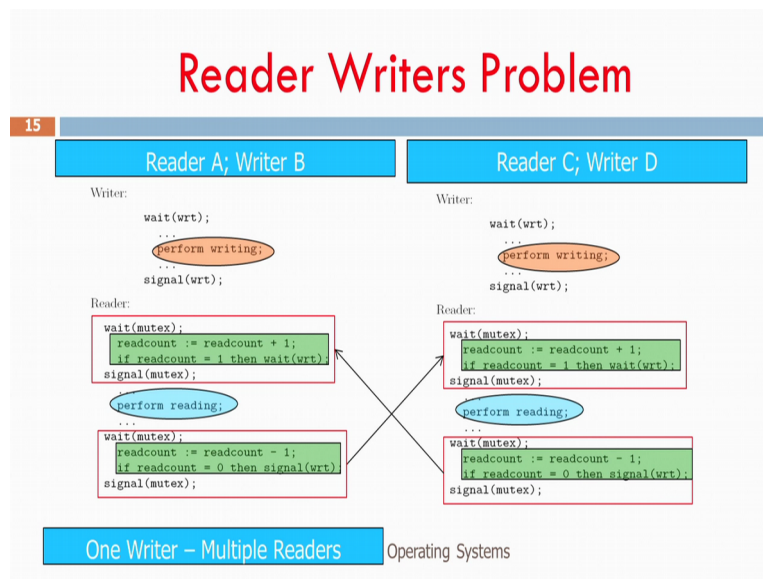
(Refer Slide Time: 33:59)



So what happens here is that there is, so when (34:02) 2 readers are there, so reader 1 wants to do reading, it first gets wait mutex, right and then it increments read count, just forget the net statement if read count equal to 1 we will explain little later, then it will signal mutex, it has incremented read count and now, it is doing reading, after it signals mutex reader 2 can now go and increment read count and he (34:33) can also do signal mutex and he can also do reading. So at the same time 2 fellows can do the reading, but 2 fellows cannot increment read count at the same time, if they are allowed 2 then we will land up with the inconsistency we have already seen in this lecture and again when they finish the reading, they want to decrement read count again there they will be a serialization.

So reader 1 will finish then reader 2 or reader 2 will finish and then reader 1, okay (35:05). So please note that I if I put a wait mutex in the beginning and signal mutex in the last and I do not have any signal and wait in between then which property is (35:16) the property that multiple readers can read at the same time that property gets (35:21). So I have to put a wait and signal before and after read count equal to read count plus 1 and a wait and signal before read count equal to read count minus 1, before and after, so then what happens is multiple readers can read at the same time, but they cannot increment or decrement read count the same time, so that is the basic thing, please understand, this is how it works. So this is about readers.

(Refer Slide Time: 35:49)



Now so what are the 2 properties that we have ensured, now let us go here, I have ensured that two writers cannot write at the same time, so only one writer can enter. Multiple readers can read at the same time, right and read count is consistently incremented and decremented, these 2 I have ensured, but what I am not ensured is that when a writer is writing no reader can read and when a reader is reading, no writer can write and that is basically achieved by those 2 statements if read count equal to 1 then wait wrt, so what when will read count be equal to 1 when I am the first reader, correct right, because I have just increment a read count equal to read count plus 1 and then immediately I am executing if read count equal to 1 then wait wrt, please note the reader code, right.

Now when read count equal to 1 means I am the first reader, then I will go and say wait wrt, if already somebody has entered and they are doing the writing, then what will happen I have to wait, if there is writer already in, I have to wait, correct. If no fellow is writing there then wrt will be 1 I would have made a 0 and I will start reading, correct. So if another writer want to enter he will start waiting, because if I am the first reader and I go and wait on wrt what will happen I will and nobody is writing then I will get that wrt as 1, I will make wrt to 0, right and since, wrt is zero no other writer also can enter into it, correct.

So the first reader who gets wrt as 1, he will make wrt 0 and till wrt becomes 1, no writer can enter. So when a reader as entered no writer can enter, when a writer has entered no reader can enter so that we have taken care off, right we will perform reading and then so similarly, please note that when a writer is writing, the first reader goes and tries to read, he would have

increment the read count and now read count is 1 then he will go and wait in wrt, right. So he has to wait till (38:23) till the writer finishes, he has to wait till the writer finishes.

Now please also note that the mutex is also with him, the mutex is also with the reader, he has not still release the mutex. So any other reader who wants to execute on read from this, they will come and wait where they will wait on mutex itself. So reader please note that writer 1 is writing, so it has wrt, reader 1 reader A is the first one to start reading, now he will find that read count is 1, he will go and wait on wrt. So reader A has mutex and he is waiting on wrt.

Now reader B wants to or reader C wants to read here, now he will go and wait on mutex itself, because mutex is currently held by writer A. so writer B is writing, so wrt is writer B reader A wants to read, he gets mutex wait mutex he will make mutex zero he will get mutex, he will enter that code, he will make reader read count equal to read count plus 1 and now, he will find out read count is 1, because he is the only reader. Now, he will go and wait on wrt but wrt is held by writer B. Now, reader C wants to read he will now wait in mutex itself, because it is there (39:47), okay.

So now what happens when one writer is writing and any number of readers who want to read, everybody will start waiting in mutex they getting this everybody will start writing waiting in mutex. So when one writer is writing, no other reader can enter the corresponding critical section. Now when the writer finishes signal write then it will when it signals what will happen you are the reader A will get that signal. Now he will go and signal mutex, so reader C will now do read count equal to read count plus 1 and then he will now wait on write, now read count becomes 2, so when read count is 1 only 1 wait on writes, you will not execute the second statement there, the reader C will not execute the second statement there he will signal mutex and then correspondingly every reader can enter.

So when the writer finishes then all the readers who are waiting will start entering into it, so multiple readers can enter. So the only the first reader will check if there is a writer if there is a writer inside, he will stop everybody else he will hold mutex and you will stop everybody else from entering and when the writer finishes, immediately he will enter and once he enters, he will allow everybody else to enter, because he will release mutex. So the first reader alone will check whether there is a writer and the other readers will not check, so that is why we have put if read count equal to 1 then wait for it.

Now after we perform the reading then what happens again when I do read count equal to read count minus 1, if read count equal to 0 that means I am the last reader then I will go and say signal wrt, right. So the moment I start reading the first reader will hold wrt, right and then he will enable second reader, third reader, fourth reader, now the fourth reader is the one who is going to finish say, let us say they have doing it in you know it takes that same amount of time, so the second reader will finish first reader, second reader, third reader, fourth reader is the fellow who finish that fellow will be responsible for returning (0)(42:03) the wrt, right. So that is why we say if read count equal to zero then signal wrt, right.

So by just using two semaphores, we are now in a position to see that 2 things, one semaphore mutex is basically used to increment and decrement read count, in addition it is all mutex is also used to stop readers from entering when there is a writer as I explained you and the there is a wrt which ensures that exactly one writer enters and when that writer inside (0)(42:43) no reader will be there and many readers can enter and when there is a reader no writer can enter. So these two are basically achieved, so this is an interesting example of how we can use wait and signal to basically you know do this synchronization. So this is called the codes section, so perform writing perform writing perform reading. This is called the code section and that code section is a critical section. So for a writer process perform writing is a critical section for a reader process perform reading is a critical section.

Now I could have reader A reader reader C to readers, they need not do the same reading, they can do different reading, but that reading part (0)(43:26) is a associated critical section. Similarly, writer B and writer D as you see can they can perform different writing, they need not do the same code as you see in contemporary text books, okay. So this can be different code, so I have reader A reader B reader A writer B reader C writer D four fellows here, each will do different writing and different reading respectively that code can be different but all these four sections which I have marked as ellipses here, they are associated and they need to follow this access principle that at any point of time exactly one writer can be there and no reader can be there, multiple readers can be there, but then no writer can be there and this is what we need to achieve here, right.

(Refer Slide Time: 44:13)

Mutual Exclusion(Locks)-Rules

16

- ❑ No assumptions on hardware: speed, # of processors
- ❑ Mutual exclusion is maintained – that is, only one thread at a time can be executing inside a Code Section
- ❑ Execution of Code Section takes a finite time
- ❑ A thread/process not in CS cannot prevent other threads/processes to enter the Code Section
- ❑ Entering Code Section cannot be delayed indefinitely: no deadlock or starvation.

Deadlock: Permanent blocking of threads/process

Livelock: Execution but no progress

Starvation: One or more threads/process denied resources

Operating Systems

What must be done during Critical Session?

17

- ❑ To prevent race conditions, we must guarantee that certain sections of the cooperating programs are not interleaved.
 - ❑ **Mutual Exclusion** :- If a process is executing in its critical section, then no other processes can be executing in their critical sections
 - ❑ **Progress** :- If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
 - ❑ **Bounded Waiting** :- A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
 - Each process executes at a nonzero speed.
 - No assumption concerning the relative speed of processes.

Operating Systems

So the thing is that there are multiple things that we need to achieve in this critical sections or critical session or whatever you call here. So one thing is mutual exclusion that we have seen that exactly one fellow should be there or multiple depending on the access rule, the second thing is progress. What do you mean by progress that when nobody is there in the critical section and I want to enter and I should be allow to enter that is called progress and the third important thing about here is bounded waiting, I want to access that critical section I should access it within a bounded amount of time, I should not infinitely wait for that, right.

So these 3 aspects are very very important while we implement this critical section, one thing is mutual exclusion another this progress, mutual exclusion we have seen progress is that when I am when nobody is there in the critical section and I want to access it I should be

allow to access is called progress. Bounded waiting when I want to access a critical section, I express a wish to access a critical section at time t within some time $t + \Delta$ that Δ should be quantified, I should be allowed to access that there should be a guaranty that I should access.

(Refer Slide Time: 45:31)

Mutual Exclusion(Locks)-Rules

16

- ❑ No assumptions on hardware: speed, # of processors
- ❑ Mutual exclusion is maintained – that is, only one thread at a time can be executing inside a Code Section
- ❑ Execution of Code Section takes a finite time
- ❑ A thread/process not in CS cannot prevent other threads/processes to enter the Code Section
- ❑ Entering Code Section cannot be delayed indefinitely: no deadlock or starvation.
 - Deadlock: Permanent blocking of threads/process*
 - Livelock: Execution but no progress*
 - Starvation: One or more threads/process denied resources*

Operating Systems

These three are very very important, these three conditions one is called deadlock where the processes can permanently be blocked live lock where execution is done, but no progress, a starvation where one or more threads are processor denied the resources. One of these three conditions can basically occur.

(Refer Slide Time: 45:49)

Mutual Exclusion(Locks)-Rules

16

- ❑ No assumptions on hardware: speed, # of processors
- ❑ Mutual exclusion is maintained – that is, only one thread at a time can be executing inside a Code Section
- ❑ Execution of Code Section takes a finite time
- ❑ A thread/process not in CS cannot prevent other threads/processes to enter the Code Section
- ❑ Entering Code Section cannot be delayed indefinitely: no deadlock or starvation.
 - Deadlock: Permanent blocking of threads/process*
 - Livelock: Execution but no progress*
 - Starvation: One or more threads/process denied resources*

Operating Systems

What must be done during Critical Session?

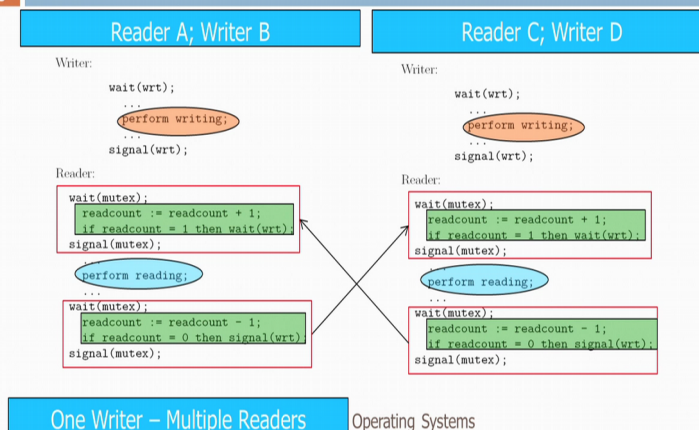
17

- To prevent race conditions, we must guarantee that certain sections of the cooperating programs are not interleaved.
 - **Mutual Exclusion** :- If a process is executing in its critical section, then no other processes can be executing in their critical sections
 - **Progress** :- If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
 - **Bounded Waiting** :- A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
 - Each process executes at a nonzero speed.
 - No assumption concerning the relative speed of processes.

Operating Systems

Reader Writers Problem

15



One Writer – Multiple Readers

Operating Systems

Hardware help?

18

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
 - Currently running code would execute without preemption
 - Generally too inefficient on multiprocessor systems
 - OS using this not broadly scalable
- Modern machines provide special atomic hardware instructions
 - Atomic = non-interruptible
 - Either test memory word and set value
 - Or swap contents of two memory words

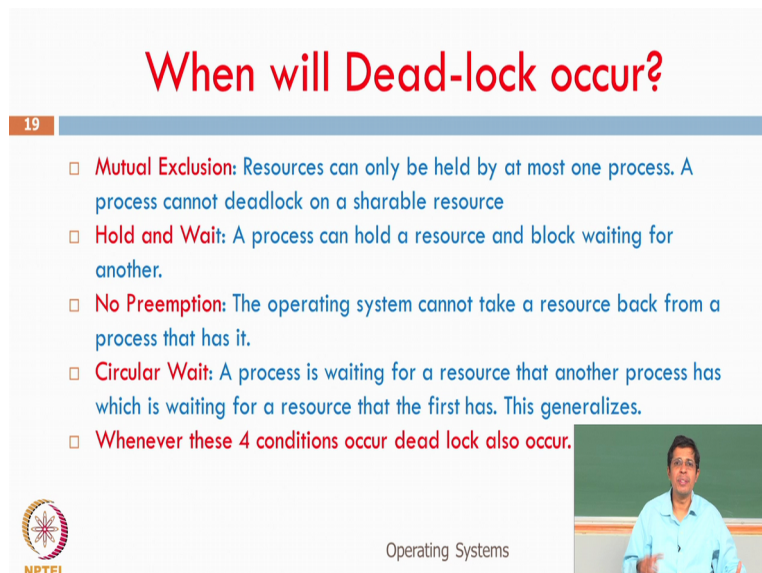


Operating Systems

So when we do process synchronization whenever we talk off mutual exclusion as we have described here we need to take care of progress and bounded waiting. So who will ensure this progress and bounded waiting, the way wait and signal is implemented that will ensure that we will have progress and bounded waiting we will have all the three mutual exclusion, progress and bounded waiting while we will not go into the exact implementation of these three, but we know that there are modern processes which can help us by having these atomic instructions.

So there are lot of atomic instructions for example, something call the test instruction, there something call swap instructions where you can swap the values of two bits, test instruction test and set instructions is called, so I can test value of bit and set it immediately as a single operation you can. So there are implementations of this wait and signal of a semaphore that basically use as this swap and test and set instructions which are atomic if you look at the instructions set of many of the contemporary microprocessors that have come in the last 2 decades you will find these type of atomic instruction that are uninterruptable and which can be used tom implement this wait and signal operations and once I could do that I can basically ensure they will be mutual exclusion and I can ensure (())(43:26) they will be progress and bounded waiting.

(Refer Slide Time: 47:17)




When will Dead-lock occur?

19

- **Mutual Exclusion:** Resources can only be held by at most one process. A process cannot deadlock on a sharable resource
- **Hold and Wait:** A process can hold a resource and block waiting for another.
- **No Preemption:** The operating system cannot take a resource back from a process that has it.
- **Circular Wait:** A process is waiting for a resource that another process has which is waiting for a resource that the first has. This generalizes.
- **Whenever these 4 conditions occur dead lock also occur.**

NPTEL

Operating Systems



So one has to be extremely careful about deadlocks, first thing is that when there is mutual exclusion we need mutual exclusion, because we need to shear resources along with mutual exclusion if I have hold and wait that is a process is holding a semaphore or a resource by which it is blocking another process from entering and there is no preemption that is the



operating system cannot take resource back from a process that are set, right I have given that resource, I cannot take it back and then you will end up (())(47:49) with a circular wait A is holding a resource and it is waiting for B to release, B is holding a resource which A wants but it is waiting for C, C is holding a resource which a wants and the so a is waiting for B, B is waiting for C, C is waiting for A. if (())(47:51) circular wait comes then if all these four conditions are true then deadlock can occur.

(Refer Slide Time: 48:17)

Hardware help?

18

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
 - Currently running code would execute without preemption
 - Generally too inefficient on multiprocessor systems
 - OS using this not broadly scalable
- Modern machines provide special atomic hardware instructions
 - Atomic = non-interruptible
 - Either test memory word and set value
 - Or swap contents of two memory words






Operating Systems

What must be done during Critical Session?

17

- To prevent race conditions, we must guarantee that certain sections of the cooperating programs are not interleaved.
 - **Mutual Exclusion** :- If a process is executing in its critical section, then no other processes can be executing in their critical sections
 - **Progress** :- If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
 - **Bounded Waiting** :- A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
 - Each process executes at a nonzero speed.
 - No assumption concerning the relative speed of processes.

Operating Systems

When will Dead-lock occur?

19

- **Mutual Exclusion:** Resources can only be held by at most one process. A process cannot deadlock on a sharable resource
- **Hold and Wait:** A process can hold a resource and block waiting for another.
- **No Preemption:** The operating system cannot take a resource back from a process that has it.
- **Circular Wait:** A process is waiting for a resource that another process has which is waiting for a resource that the first has. This generalizes.
- **Whenever these 4 conditions occur dead lock also occur.**



Operating Systems



So when we do process synchronization, one of the things is that we implement such that these three conditions are satisfied mutual exclusion, progress and bounded waiting, in additions we also see there a circular wait does not happen and that is very very important. So when you look at a complex system like the file system where lot of synchronization needs to be done, because many processes shear many files. Now to basically handle deadlock and the ensure that a deadlock free (48:39) come that becomes a big challenge. Deadlock can also be denial of service and so there is also some sort of a security implications for this.

(Refer Slide Time: 48:50)

Synchronization in Linux

20

- Prior to kernel Version 2.6, disables interrupts to implement short critical sections
- Version 2.6 and later, fully preemptive
- Linux provides:
 - semaphores
 - spin locks



Operating Systems



So synchronization in Linux prior to kernel version 2.6, the synchronization happens like this, I want to have an atomic instruction that means it is uninterruptable, so how do I (49:01) an uninterruptable instruction go and disable interrupt when it starts executing, enable

interrupt after it completes execution, but there is a very (49:07) way of doing it, when version 2.6 and later it has become fully preemptive and Linux actually provides semaphores and spin locks, okay.

So this is all that I wanted to talk about process synchronization to some depth about race conditions and how do you implement mutual exclusion etcetera, right. So this will form some basis for you to appreciate operating system security and administration at a later point of time. So I also guide you to some of the textbooks the standard text books like Galvin to basically look into these chapters in detail, the objective of this course is to actually teach you administration of operating systems and the administration of network security networks basically from security perspective.

Nevertheless these are some titles, which needs to be understood in great detail and that is why we gave glimpse of it, it is not a complete education on process synchronization but whatever is necessary is given in a nutshell, but there are also mooc courses on operating systems plus standard textbooks and other NPTEL lectures which you are welcome to read to understand more about process synchronization. So we will meet in the new module, thank you.