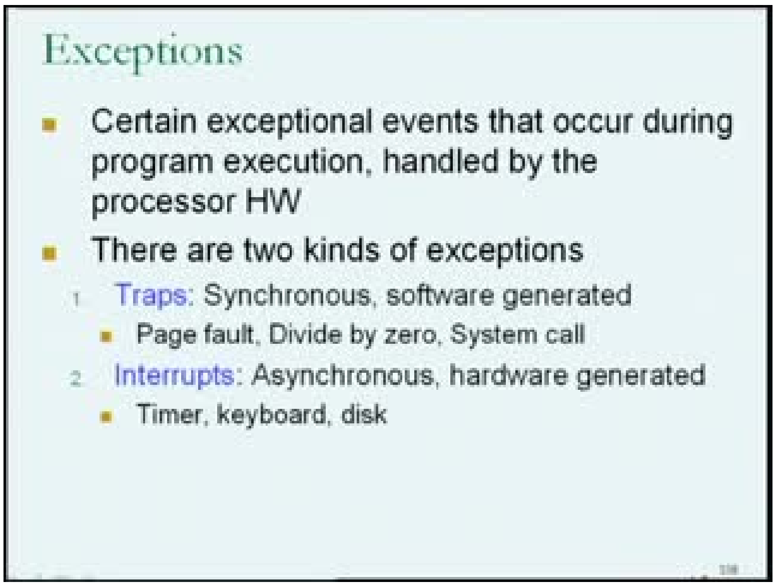
High Performance Computing

Prof. Matthew Jacob
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Module No. # 04
Lecture No. # 19

(Refer Slide Time: 00:23)

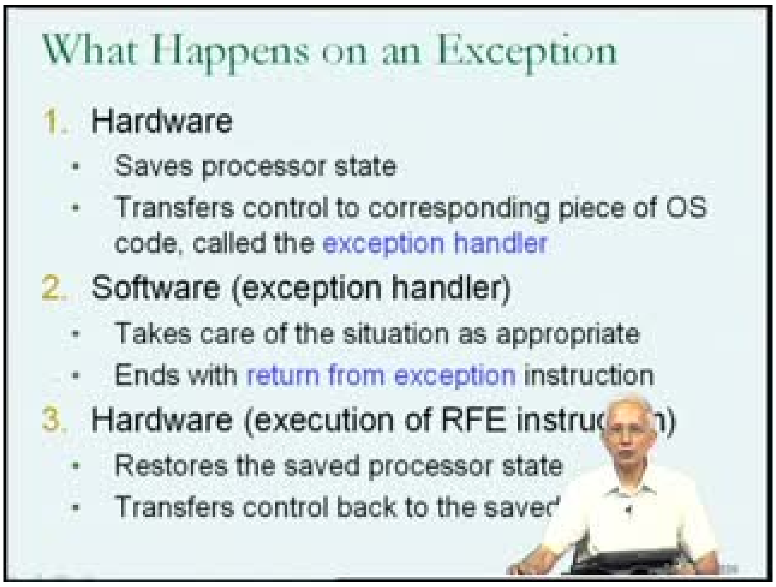


Exceptions

- Certain exceptional events that occur during program execution, handled by the processor HW
- There are two kinds of exceptions
  1. Traps: Synchronous, software generated
     - Page fault, Divide by zero, System call
  2. Interrupts: Asynchronous, hardware generated
     - Timer, keyboard, disk

This is lecture 19 of the course on high performance computing. In lecture 18, let me just remind you, we had seen about exceptions; the exceptions are these hopefully rare events that occur, but an important not only for the operating system to share the resources of a computer system, but also for interaction with the outside world.

(Refer Slide Time: 00:42)

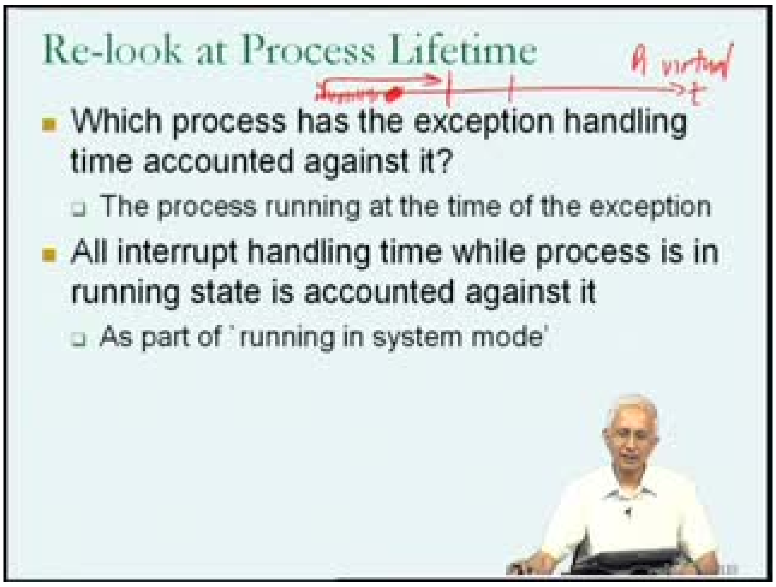


What Happens on an Exception

1. Hardware
   - Saves processor state
   - Transfers control to corresponding piece of OS code, called the exception handler
2. Software (exception handler)
   - Takes care of the situation as appropriate
   - Ends with return from exception instruction
3. Hardware (execution of RFE instruc[illegible]n)
   - Restores the saved processor state
   - Transfers control back to the saved

So, there are two kinds of exceptions: call traps and interrupts. In both cases, the handling of an exception requires hardware support. Therefore, I said that exceptions are essentially a hardware concept, which must be utilized by the operating system. And the hardware must be design to run through series of events, through which the operating system code can deal with the exception.

(Refer Slide Time: 01:20)



Re-look at Process Lifetime

- Which process has the exception handling time accounted against it?
  - The process running at the time of the exception
- All interrupt handling time while process is in running state is accounted against it
  - As part of 'running in system mode'

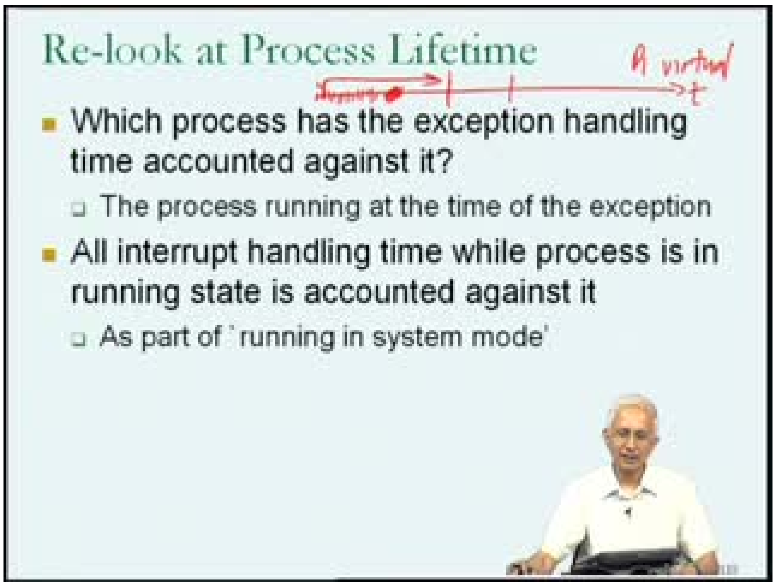
Therefore, the exceptional situation need not be problem, as far as the correct execution of programs are concerned, such as divide by 0. And further, that the exception mechanism can be used to the benefit of the operating system, in the sharing of the resources of a computer system among many programs in execution. Now with this concept of the exception, we actually need to re-look at our understanding of process lifetime.

Now, you will recall that we had this prospective of process lifetime, that we could think about real elapsed or wall clock time, which is taking into account, the entire history of events on a process, as far a processor; as far as use of the CPU is concerned, the different running processes at various intervals of time. We could look at process life, we could look at the life time in which there was the virtual time - the virtual CPU time - from the perspective of one of those processes. That one could further look at the breakup of the intervals of time on the virtual time line, in terms of time spent in user mode and time spent in system mode.

Exceptions rise interesting additional question to worry about. Essentially, when an exception occurs, the handling of the exception is going to take some amount of time, because a operating system code is going to have to be executed to handle the exception. And therefore, the time spent in handling the exception, in executing the exception handler must be accounted against one of the other process; could be the user process. User processes P 1 through p 100 or currently on the computer system. And this is basically a question, which we do need to know something about, because this main depending on how this question is answered. We may have to be a little careful in using CPU virtual time, as an indication of how much time our program is taking.

So, let me just go into the nature of the answer to this question on typical systems and then, just for the ramifications are. So, the question is this, which process when an exception occurs in the exception handler has to be executed? Exception handler - there is part of the operating system. Then, which process has the exception handling time accounted against it? Now, the answer is, that at the time that the exception occurred, the instant in time that the exception occurred, one of the processes, let us say - P 1 would have been the running process; and it is against process P 1, that the time for handling the exception handler is typically accounted.

(Refer Slide Time: 01:20)



Re-look at Process Lifetime

- Which process has the exception handling time accounted against it?
  - The process running at the time of the exception
- All interrupt handling time while process is in running state is accounted against it
  - As part of 'running in system mode'

Now, this means that all the interrupt handling time and strap handling time, which occurs while a processes in the running stage is accounted against it. Now, you should bear in mind, that this is a little bit of an unfair accounting, because not all of the exceptions that occur, when a processes running are actually the fault of the process.

For example, we saw that if I looked at the process P 1 - virtual time line and process P 1 is running for first got one second, then gets another second. I know that once let us say for millisecond, a timer interrupt occurs; and the timer interrupt handler may take few microseconds to execute are all less.

But all of that time is going to get accounted against process P 1, as time spent executing in not in user mode, but as time spent executing in system mode. Why is it accounted again system mode? Because whenever in exception occurs, control gets transfer to the operating system, timer interrupt handler in this case. And conceivably, if there is large amount of time spent in that way, it will get accounted against process p 1. Clearly, the timer interrupts are no fault of process P 1.

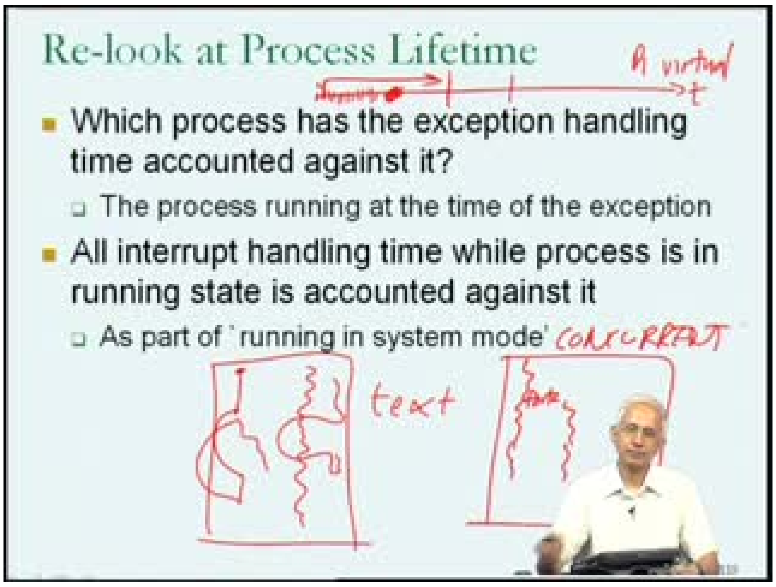
Similarly, when process P 1 is executing and there are other users logged into the same system, and let me just backup a little bit, while it is faired, it is correct to say that the timer interrupts are not due to process P 1's execution; it is also fair to say that process P 1, like all the processes must pay the overhead of supporting the operating system and that therefore, the timer interrupt handler should not be a problem with any process; they are essential for the correct running of the system.

But, let us look at another example. Suppose, on the computer system, where this timeline is being drawn, there are several users logged in and one of the users is typing something at the keyboard. Now what is that mean? That means, that when process P 1 is running in this interval of time, whenever a keyboard interrupt happens, the keyboard interrupt handler has to be executed; and that interval of time too would get accounted against process P 1, as time spent executing in system mode.

Now, the time spent in handling the keyboard interrupt, generated by another user has nothing to do with process P 1; and therefore, one might say that, this is in unfair accounting. Now, one should also bear in mind, that its slightly that these pieces of code are going to be return, so that they are very small and efficient - they are the interrupt handlers and the exception handlers in general.

And therefore, the amount of time spent in handling, they may not be very high. But if there is a very high load on a system, in other words, a larger of users doing a lot of activity, I say a lot of typing, a lot of mouse movement, etcetera, then the amount of time spent in this activity could end of being non-trivial. So, it is something that one has to be aware of, and I will raise this issue once again, when we come up against it. But we should note that, as part of a component of time that is reported back to a process, if it asks of the operating system how much CPU time was spent was I running, then part of the time spent running in system mode, could well be time spent handling exceptions; Some of which may be x interrupts, that are really nothing to do with the process in question; that is the, may be a fact of life on the system that you are dealing with.

(Refer Slide Time: 06:55)


Re-look at Process Lifetime

- Which process has the exception handling time accounted against it?
  - The process running at the time of the exception
- All interrupt handling time while process is in running state is accounted against it
  - As part of 'running in system mode'

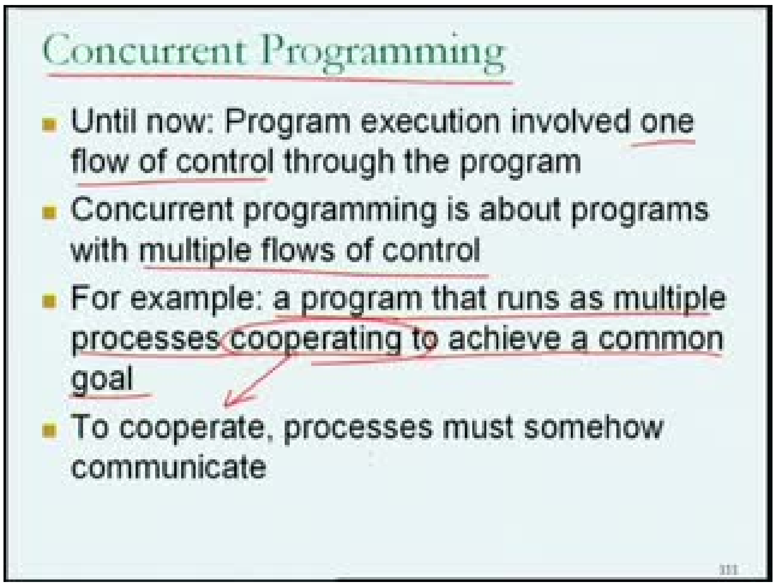
Now, I am going to switch from talking about the operating system side of process management to, in fact, the programmer side of process management. This is a topic which I will title as concurrent programming; and it is essentially a new perspective on programming which may arise. Now, that we know, that we could write programs which run as multiple processes.

Now, let me just remind you that, until now in this course, whenever we talked about program execution, for the most part we were talking about program in which there was one flow of control. And what I mean by that is, if I looked at the program as, let us say just it is text, then one flow of control, means that the program execution starts at some point, then goes to the next instruction, then it goes to next instruction; then maybe there is a control transfer goes to a function call; it executes the function call, then may be a transfer is control back.

And I could think of the execution of a process within this particular program, in terms of this worm which is worming its way through the instructions of the program. It changes its direction, whenever there is a control transfer and so on, but there is only one worm going through the text of the program. In other words, we were for the most part talking about one flow of control through the program.

Now, that we know about the fork system call, and we realize that operating system support multiple processes; many of which could even be from the same program. This raises the possibility for us to write programs, which are concurrent programs or programs in which there is more than one flow of control. In other words, going back to a diagram, I could write a program which starts as one of those worms - flows of control - but at that point in time, it might do a fork.

(Refer Slide Time: 08:58)



Concurrent Programming

- Until now: Program execution involved one flow of control through the program
- Concurrent programming is about programs with multiple flows of control
- For example: a program that runs as multiple processes cooperating to achieve a common goal
- To cooperate, processes must somehow communicate

So, that the parent flow of control continues, but a child flow of control starts. And from this point in time onwards, they are two flows of controls or two worms working their way through this concurrent program. So, that is going to require possibly a new perspective on, what it means to write a program with the understanding, that they could be multiple processes, which come into existence as a result of the execution of that program.

So, in a briefer side, we will try to understand some of the important concepts related to concurrent programming. So, we are still talking about operating system related issues; this is we are still talking about issues relating to the way that the operating system manages processes, but we just trying to see, whether the something we need to know from the programming side, in order to leverage or get a benefit out of this knowledge.

So, we will briefly look at what concurrent programming might involve. What we have learned now, we understand that a currentor concurrent program might be one way to setup a concurrent program; in other words, the program which has multiple source of control might be to set it up, as a program that runs as multiple processes cooperating to achieve a common goal. In general, whenever we wrote a program, the objective of writing the program was to achieve a goal.

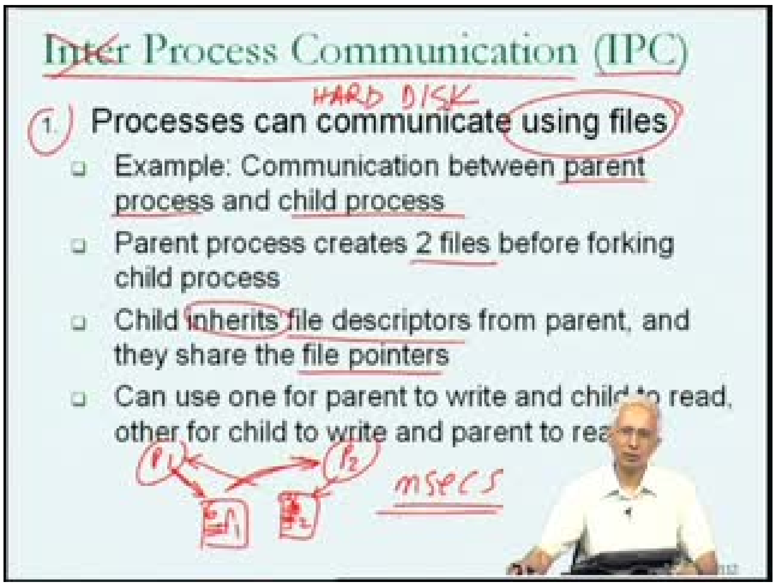
We have always talked about, I defined a program as algorithms and data structures, which we are put together in order to achieve some objective. We are now in a situations where we are going to have a program, that runs as multiple processes and they must be trying to achieve that same or some objective. But since they are multiple processes, we will talk about how all of those processes cooperate, and rather than talking about the objective, we will talk about that as there common goal.

So, there is this notion of cooperation, which enters the picture. Until now, when we were writing our ordinary program which had a single flow of control, they was not really notion of cooperation, since we were managing only one control flow. Now, we are talking about programs in which there are multiple control flows, possibly multiple processes, and we do have to worry about how these processes could cooperate.

Now, in some sense, one could start by saying, that in order to co-operate processes must somehow communicate. And you could actually think about this problem in the more general setting of how do human beings cooperate. Is it possible for two human beings to cooperate with each other, if they do not communicate with each other? The answer is, it may be possible if the two human beings have a long history, they have cooperated with each other many times in the past.

Therefore, now they are in a situation, with they do not have to communicate in order to cooperate, they know very well, how they suppose to cooperate. But one could argue that, what is happening there is some kind of intersect communication. The situation that we are in over here is where there are processes, which cannot have any kind of preexisting knowledge gain by experience. Human beings can gain by experience, these processes we suspect cannot, they have to be programmed to do whatever they do. And therefore, this seems like a reasonable starting point; processes must somehow communicate in order to cooperate. When human beings cooperate, they communicate in various forms; some of you make cooperate with others by talking, others may cooperate with others by passing note to each other; if you are not in the same room, you might cooperate by talking on the telephone, you might cooperate by E-mail to each other and so on; these are all ideas, which are related to communication. And we need to understand something about how processes can communicate, before we can talk about what it means to do concurrent programming.

(Refer Slide Time: 12:27)



~~Inter~~ Process Communication (IPC)

1. Processes can communicate using files
   - Example: Communication between parent process and child process
   - Parent process creates 2 files before forking child process
   - Child inherits file descriptors from parent, and they share the file pointers
   - Can use one for parent to write and child to read, other for child to write and parent to read

So, the immediate task is to talk about how processes can communicate; and this is often the topic of how processes can communicate is often described as inter process communication. Of course, the word inter is not really, you may wonder why it is there; you could actually scratch it out. We are talking about the communication between processes. There is little concept of communication inside a process anyway, and this is sometimes abbreviated as IPC - Inter Process Communication. We are going to look at various possible ways, that processes could communicate with each other. And for the most part, the ways are we are going to look at are, supported on the operating systems that we have been talking about directly.

Now, the situation that we are in is, I have 2 processes - say process P 1 and process p 2, I know that they have their separate virtual address spaces, therefore, it is not like one of them can write a variable, which the other can read; so, they cannot readily communicate in that way.

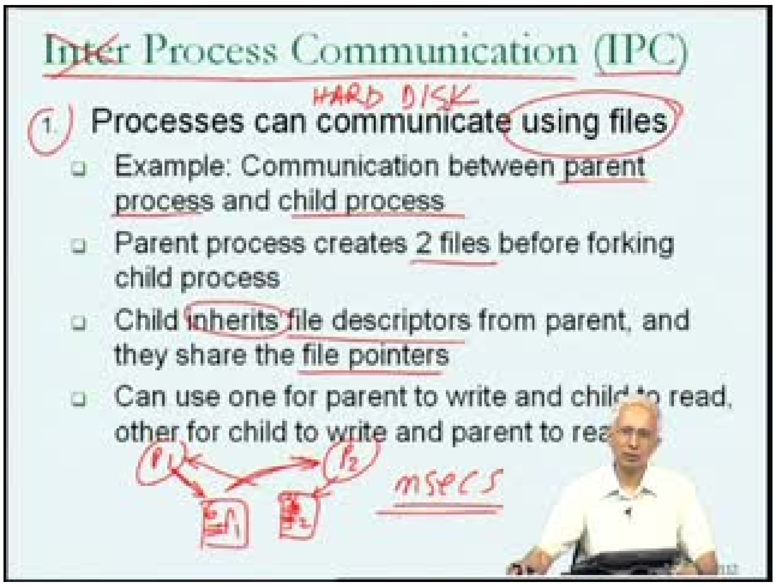
So, we have to think little bit about, how they could communicate in the absence of that kind of a facility. And one idea which might come to mind is, we know that the systems, that we deal with support this concept of files, and that files or entities which from, what I have mentioned, seem to exist on a hard disk; and as long as process can open a file and has sufficient privileges to do so, it can communicate with another process conceivably through that file.

So, this may be a starting point, the idea that processes can communicate using files. So, let me just give an example; suppose, as a situation where I have two processes, they happen to be a parent process and a child process. In other words, when I program started running, there was one process; subsequently, it did a fork system call, at which point I call it the parent process and the newly created processes what I call the child process; and I want for some reason, these two processes to communicate.

Now, for them to communicate, it may be possible, that in that particular situation, that I mean I need one of them to be able to communicate a value to the other. So, let say that, the parent process has computed the value 6 and it wants to communicate the value 6 to the child process. Hence, the question then becomes, how can the parent process communicate the value 6 to the child process?

Then, first idea is one could do this using files. So, one possible way of doing this is out line here. Now, I am going to set this up using two files, because it is conceivable, that I want the parent to be able to send data to the child and the child might have to send data to the parent; and rather than using one file for both, I will use one file for the parent to write into and the child to read from, and the other file for the child to write into and the parent to read from. So, before the parent process does the fork, it could create those two files: one for it to write into, the other for the child to write into.

(Refer Slide Time: 12:27)


~~Inter~~ Process Communication (IPC)

1. Processes can communicate using files
   - Example: Communication between parent process and child process
   - Parent process creates 2 files before forking child process
   - Child inherits file descriptors from parent, and they share the file pointers
   - Can use one for parent to write and child to read, other for child to write and parent to read

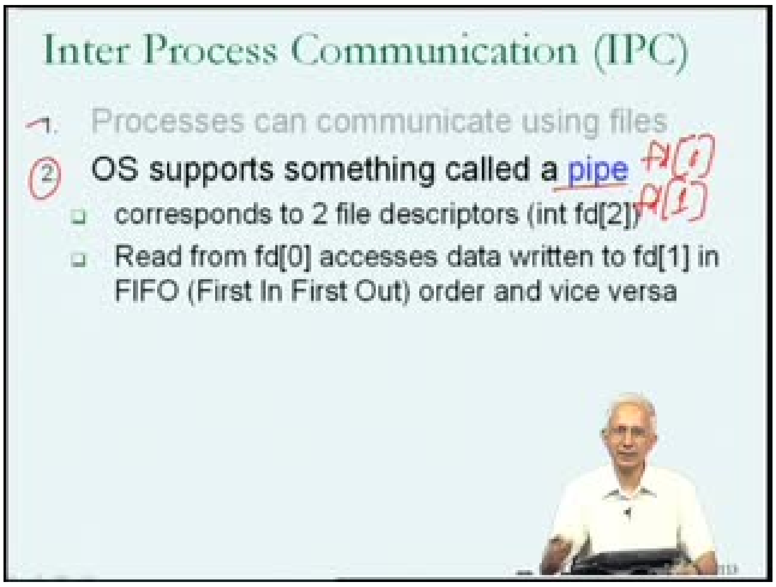
Now, we have seen that when the fork system call occurs, the child inherits a lot of things from its parents such as, its text, its data, its stack and its heap. It also as it happens inherits information about files from its parents; so, we will learn more about file descriptors and file pointers later. But for the moment, it is enough to understand that as a part of the fork system call, the child will inherit enough information about the two files that its parents had created.

So that subsequently, the parent on the child can be written to communicate by using one of the files for the parent to write and the child to read, and the other for the child to write and the parent to read. So, the idea is, there is process P 1 which is the parent process, p 2 which is the child, and there are 2 files: file 1 and file 2. So after the forking, if the parent wants to communicate the value 6 to the child, it writes a value 6 in file 1. The child in order to see what the parent has communicated with it, so the parent writes into file 1, the child reads from file 1. If in response the child wants to communicate, suppose the child wants to communicate was to calculate 6 multiplied by 6, it would do so and to communicate that back to the parent, it would write 36 in the file p 2, which the parent would read; and therefore, communication could happen.

So, all of the contains of file P 1 will be things written by the parent, which can be read by the child; and all of the things in file p 2 could be things written by the child, to be read by the parent and therefore they can communicate; so, this is possible. And the only problem with this kind of a mechanism is that it uses files.

We have seen that anything involving files will involve something, which is as slow as a hard disk. And therefore, anyone of these pieces of communication between the parent and the child is going to involve, one write to the hard disk, and one read from the hard disk, which we have seen could involve a large number of milliseconds if not seconds. And therefore, one can understand why I have put this down is the first idea an inter process communication, because they are certainly they are better, be better idea is then this. Since, this is going to be a very high overhead way for one process to communicate with another process, in terms of the amount of time that it will take for a single value to be communicated from the parent to the child, conceivably several milliseconds if not seconds; and that is a enormity of time on the time scale of the processor, as we have seen several orders of magnitude.

(Refer Slide Time: 17:53)



Inter Process Communication (IPC)

1. Processes can communicate using files
2. OS supports something called a pipe
   - corresponds to 2 file descriptors (int fd[2])
   - Read from fd[0] accesses data written to fd[1] in FIFO (First In First Out) order and vice versa

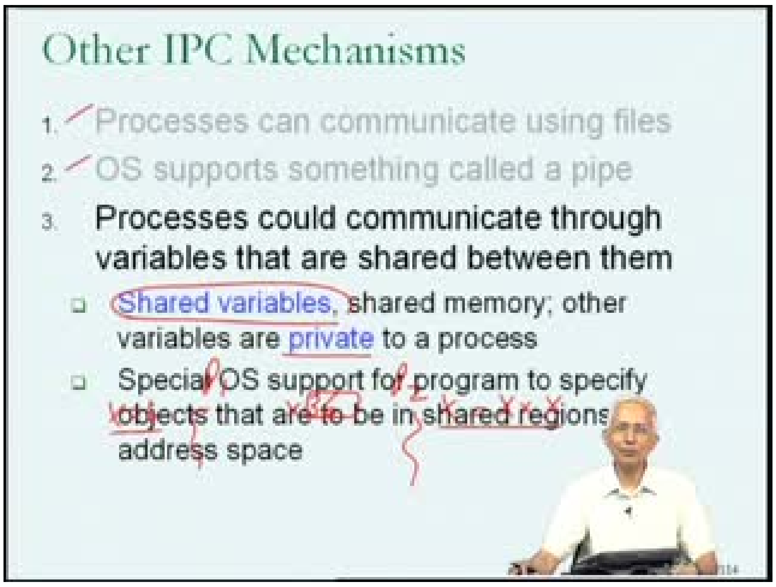
So, it is possible to setup programs, so the processes communicate using files. But we want some better ideas, and one kind of an idea which is supported by almost all operating systems, it is a idea of something called a pipe - that is why I put it in blue, the technical term - and it turns out that the pipe look superficially very much like, the mechanism that we were using just now using files. In a sense, what the pipe provides is two file descriptors; in other words, two mechanisms that could have been used to read and write into files; but rather than using them to read and write to files, it short circuits the mechanism by passes the disk and directly does a communication through the operating system. Therefore, this mechanism will not involve the overheads of hard disks, but will provide the same kind of facility for a parent to write into a file descriptor, subsequently the child can read from that file descriptor.

Similarly, the child could use the second child file descriptor to write, and the parent would read from that, to communicate values one way on the other, without the disk being involved - without a hard disk being involved.

So, the general ideas they are two file descriptors, which would be declared something like int fd2. So, the two file descriptors would be fd 0 and fd1, as far as the pipe is concerned; and one could be used for reading and writing between the parent and the child, etcetera, along lines about we had seen for doing with the file. And the main difference is, this will not involve the overheads of dealing with the file stored on a hard disk.

So, this becomes a mechanism which would actually be feasible in terms of the time scale of a processor; it will involve system calls, it will involves some amount of time for the transfer of data, but it not involved the milliseconds of time we were worried about, as far as the hard disk is concerned. So, this is some form of a rationalization in terms of overhead of the idea one, but must be supported by the operating system; it must be provided as an operating system facility.

(Refer Slide Time: 19:54)

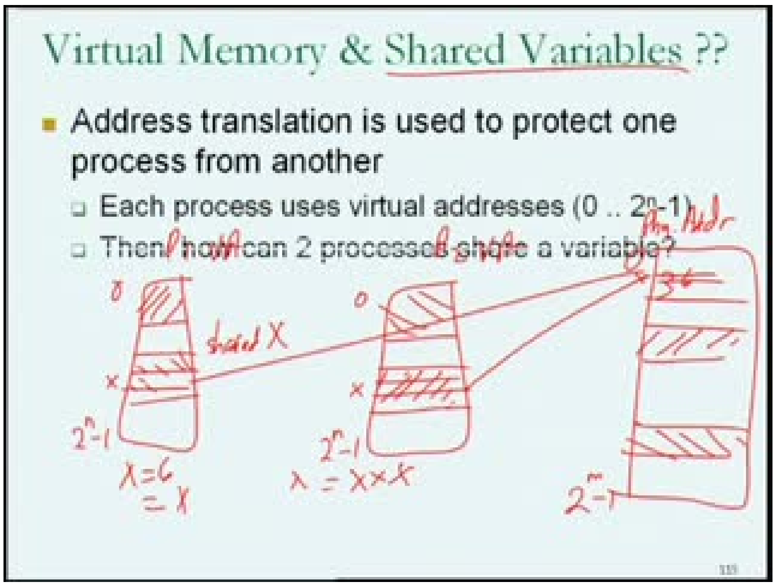

Other IPC Mechanisms

1. Processes can communicate using files
2. OS supports something called a pipe
3. Processes could communicate through variables that are shared between them
   - Shared variables, shared memory; other variables are private to a process
   - Special OS support for program to specify objects that are to be in shared regions of address space

Now, there are several other ideas. Now, one of them strangely enough is the processes could be made to communicate through variables, that are shared between them. Now, I had explicitly said that, this is unlikely to be the kind of thing that can be done, because of the way that we have looked at processes up to now, but let us suppose for the moment that it is possible. So, the idea here is, that there will be some variables, which can be used between process P 1 and p 2 for the purposes of communication and we will refer to those variables as shared variables; and the variables of the process which are not shared, we will refer to as private variables. So, it possible that I have process P 1 and process p 2, and they may have one variable that is shared and all there other variables are not shared or private.

Now, this is a bit of (( )), but we get clearly understand that if this facility is there, then if process P 1 wants to communicate the value 6 to process P 2, the program could be written, so that process P 1 write 6 into that variable. In other words, they just sets x equal to 6, and process p 2 in order to read the variable would read x, and if it was suppose to square it, we can multiply x by x and put the value back in x; as a result of which x will become 36, which can then be read by the parent.

(Refer Slide Time: 21:38)



Virtual Memory & Shared Variables ??

- Address translation is used to protect one process from another
  - Each process uses virtual addresses (0 .. 2^n-1)
  - Then how can 2 processes share a variable?

P1
P2
Phy Addr
0
shared X
x
2^n-1
x=6
= X
x = x×x
36
2^m-1

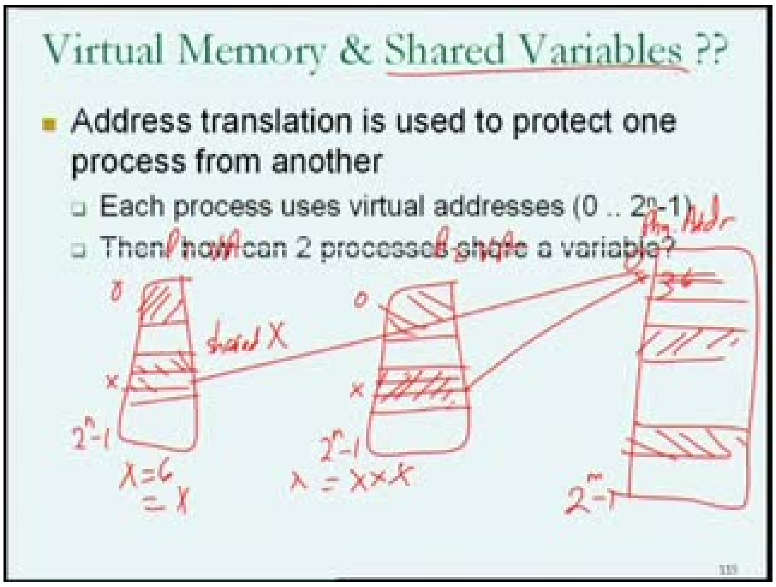
So, this is definitely we preferable to either of the previous ideas we have seen, which involved file descriptors, which may have been fairly, may have involve a lot of system calls. Here, we are clearly talking about something, that might involve only memory accesses; notice here, there is no question of having to make a system call. The only problem that we have with this idea at all is, because we have until now been talking about virtual memory, in which the objective of having virtual memory and address translation was to protect one process from another; and we were explicitly trying to make sure that, one process could not access to variable of another process.

So, we had setup things are, understanding was that the operating system setup things, so that each process uses its own virtual addresses. So, if there was process P 1 and there was process p 2, each had its own virtual address space; and the virtual address space went from 0 to 2 power n minus 1 and these were two virtual address spaces. And in reality, there was physical memory which was a physical address space, which went from 0 up to some 2 power n minus 1. And at any given point in time, some of the contents, for example, this page may have been over here, this page may be over here, at some point in time during the execution of these programs. But our whole idea as far as virtual memory was concerned, was to keep these two processes apart.

Now, we are not told that it may be useful, if the operating system can somehow provide support, so that there is a shared variable. In other words, if the operating system could allow us a settings up, so that this variable x is actually such that, if it is in memory, the mapping for the page, suppose this is the page inside process P 1 containing the variable x, and this is the page inside process p 2 containing the variable x is setup, so that both of these pages map to the same physical address space. So, in theory we understand that we wanted to keep processes separate, as far as their ability to address variables of the other word concerned, but in practice, now that we have understood that a shared variable might be useful mechanism for one process to communicate with another in during concurrent programming.

We see that it may not actually be that difficult for an operating system to help out. It basically just has to provide the mechanism by which, if I indicate in my program, that the variable x is suppose to be shared between process P 1 and process p 2, then the operating system could conceivably set things up. So, that the page containing the variable x in the virtual address space of p 1, and the page containing variable x in the virtual address space of p 2 actually have the same mapping to a physical address page, in which case they will be talking about the same variable.

(Refer Slide Time: 21:38)



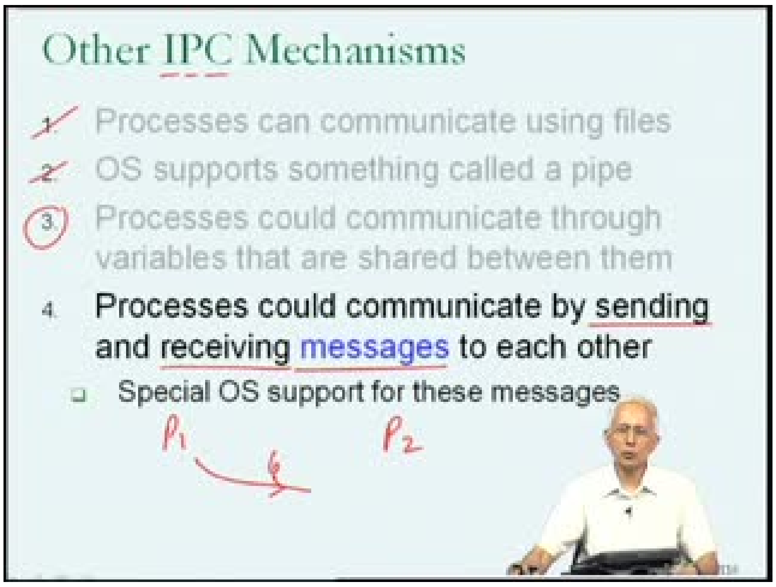
Virtual Memory & Shared Variables ??

- Address translation is used to protect one process from another
  - Each process uses virtual addresses (0 .. 2^n-1)
  - Then how can 2 processes share a variable?

And when process P1 writes into x and process p 2 reads x, they will get the same value; and therefore, they can communicate with each other just using ordinary memory operations. By some kind of operating system support, through the way that the mapping between physical addresses and virtual addresses is done. So, this can be supported by the operating system.

The idea of shared variables: the idea that they could be variables, which are shared between two processes, even though all the other variables, and all the text, and all the data, and all the other stack and heap of these two processes may be distinct from each other, in order to protect one process from the other.

(Refer Slide Time: 24:50)



Other IPC Mechanisms

1. ~~Processes can communicate using files~~
2. ~~OS supports something called a pipe~~
3. Processes could communicate through variables that are shared between them
4. Processes could communicate by sending and receiving messages to each other
   - Special OS support for these messages

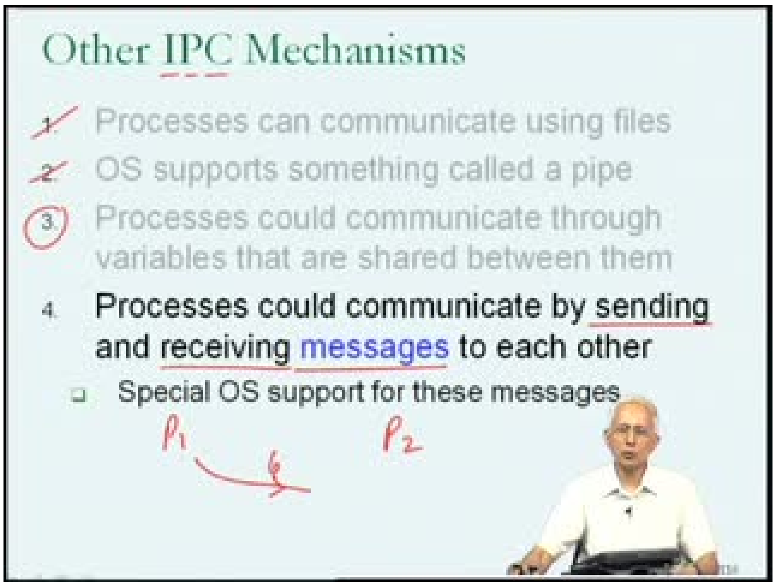
So, this is something which can be done, processes communicating through variables that are shared between them, but it requires explicit support from the operating system. By default, the operating systems - virtual memory management mechanisms are going to protect processes from each other. And therefore, this would have to be some kind of a special mechanism, which should be made available by specially declaring reasons of memory, as shared through various kinds of system call mechanisms.

So, let us assume that, this is possible; and on the systems that we are dealing with, we could find out how to set this up. So, this is a third possibility; and among three that we have seen, it is the only one which might involve a very low small number of system calls, in order to setup the communication. In the case of files, we have seen that for every read and write, they would be a system call and huge disk IO activities; in the case of using pipes, we have seen that every read or write would have to happen through file descriptive, which might involve system calls. And the reasons that I am worried about system calls is, because whenever a system call occurs, remember that is essentially a trap; and therefore, it will involve saving the state of the process.The exception handling mechanism that we had seen at the end of the previous lecture, and therefore, any IPC mechanism that involves a lot of system calls is not as good as an IPC mechanism, that involves very few system calls.

So, among the three, the third seems to involve very few system calls; they may be the need for a system call to setup the shared region of memory, but subsequently the accesses to the variable x will happen using ordinary read and write operations. In other words, by including them the variable x in the right hand side of an expression or in the left hand side of in expression, depending on whether it is a read or a write. Therefore, this is, among the three, the superior one which we have from this perspective. Now, there are other possibilities, as far as IPC - Inter Process Communication are concerned. And one which is came to something that we have seen, when we talked about human communication, we will talk about next, the idea that processes could communicate by sending and receiving something called messages to each other.

So, this is a completely new concept to us. Communication by a process is doing the activity of sending a message, at the other end the message being received. So, very clearly this is a new kind of a facility, which the operating system has to provide. And we suspect that if the operating system does provide this kind of a facility, then they will be a system calls or at library functions, which might be call send and receive, through which one process can package a piece of data within a message and send it to the other process. And similarly, the other process can explicitly indicate that it which is to receive the message which has been send by the first process.

(Refer Slide Time: 24:50)

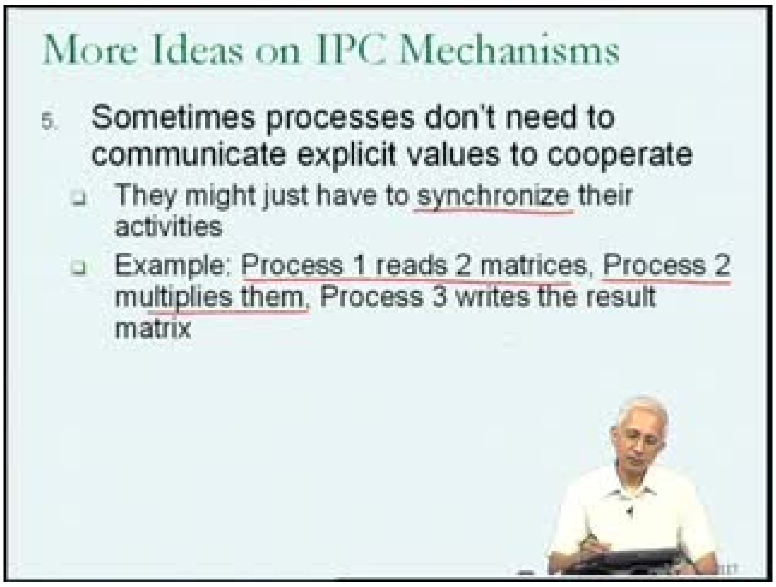

Other IPC Mechanisms

1. Processes can communicate using files
2. OS supports something called a pipe
3. Processes could communicate through variables that are shared between them
4. Processes could communicate by sending and receiving messages to each other
   - Special OS support for these messages

So, we have this idea of process P 1 and process P 2 communicating by explicitly sending these things called messages to each other. Now, with these four major ideas, we are going to see more about message sending and receiving later on. I am just introducing it has a possibility right now, then the actual kinds of facilities which might be available to do this, we will talk about later.

Now, all four of these IPC mechanisms have the property that they can used to communicate explicit data values. For example, I talked about how the current problem might be that, I want to communicate the value 6 from process P 1 to process P 2. Now, this kind of communication is not, in general, what may be required for all types of cooperation between processes?

(Refer Slide Time: 28:30)



More Ideas on IPC Mechanisms

5. Sometimes processes don't need to communicate explicit values to cooperate
   - They might just have to synchronize their activities
   - Example: Process 1 reads 2 matrices, Process 2 multiplies them, Process 3 writes the result matrix

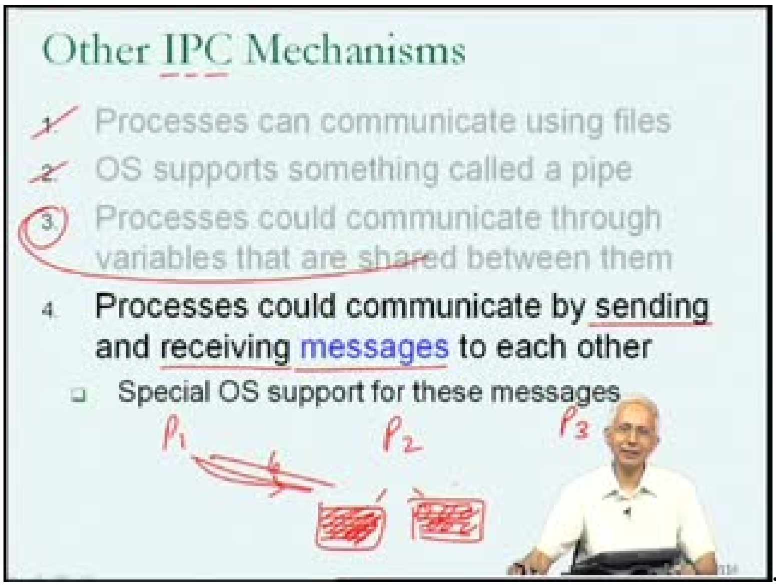
It is possible sometimes processes may need to cooperate, but may not need to communicate explicitly explicit values. So in order to cooperate, they need not need to communicate explicit values; it might just something less than that may be all that is required for the cooperation to be facilitated. And let me just give you an example; one example is that the processes may not have to receive values from each other, they may just have to synchronize their activities.

In other words, this is probably best illustrated through an example, they were synchronizes an important word here. Let me just give you a very simple somewhat trivial little example. Let suppose that the problem that I have is, they need to multiply two matrices; matrices are some form of large data structures. And let suppose that problem that I have is, I want to multiply two such matrices and since I now know about concurrent programming, I have decided that I will achieve this through a program that runs as three processes

For in my initial experiments with concurrent programming, I decided I would like to do this. So, I decided to set things up, so that first process P 1 reads in the 2 matrices, then process P 2 multiplies them and process P 3 writes the result, writes out the result matrix through the output. I am going to assume that the matrices themselves are in shared or in a form of memory which is shared between the processes such as, using the shared memory we had talked about over here.

So, I am going to worry about the data within the matrices themselves. I am going to worry about the fact that, if you think about this cooperation between these processes, the first thing that must happen is process P 1 must read in the 2 matrices, after that process P 2 must multiply them. And it is important the process P 2 does not start doing the multiplication, before process P 1 has finished reading in the 2 matrices; think about it this way. Now, there is process P 1, there is process P 2, and there is process p 3, and they have shared memory in which the matrices to be multiplied are available.

(Refer Slide Time: 30:37)

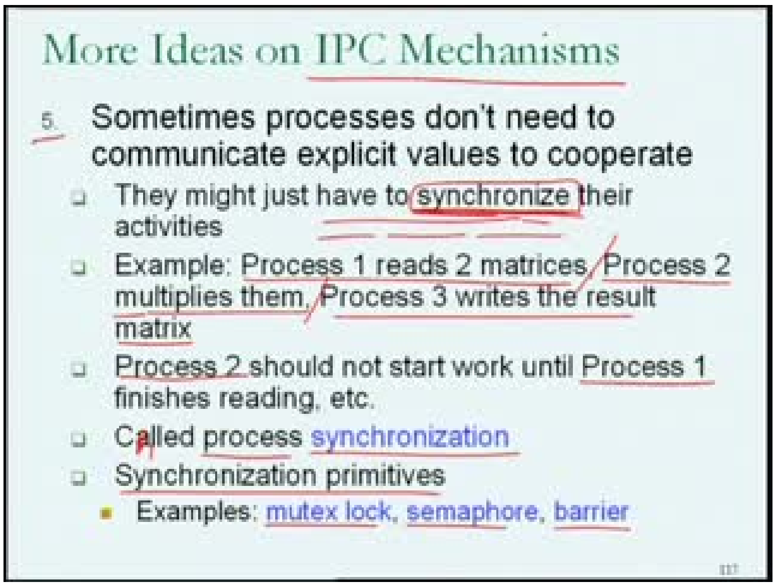


Other IPC Mechanisms

1. ~~Processes can communicate using files~~
2. ~~OS supports something called a pipe~~
3. Processes could communicate through variables that are shared between them
4. Processes could communicate by sending and receiving messages to each other
   - Special OS support for these messages

So, process P 1 is reading in the matrix, which the first matrix is to be multiplied, is reading in the second matrix which is to be multiplied. Now, what would happen if process P 2 started doing the multiplication, before process P 1 finish reading in? It could be the case, that process P 1 has read in all of the first matrix we multiplied, but it is not finish reading in the second matrix we multiplied, in which case the values within the second matrix could be garbage values; if you are lucky, they may be all zeros.

(Refer Slide Time: 31:20)



More Ideas on IPC Mechanisms

5. Sometimes processes don't need to communicate explicit values to cooperate
   - They might just have to synchronize their activities
   - Example: Process 1 reads 2 matrices, Process 2 multiplies them, Process 3 writes the result matrix
   - Process 2 should not start work until Process 1 finishes reading, etc.
   - Called process synchronization
   - Synchronization primitives
     - Examples: mutex lock, semaphore, barrier

So, if process P 2 starts doing the multiplication on the data, before all the data has arrived, then the result will be meaningless on the perspective of the task at hand. Therefore, it is important here, the process P 2 does not start doing the multiplication until process P 1 has finished doing the reading of the matrices. Similarly, the job of process P 3 is nearly to write the result of the matrices; and clearly process P 3 should not start doing its work, until process P 2 has finished doing its work.

So, this is a situation where we actually need the process is to synchronize their activities. In other words, process P 2 must not start, until process P 1 has finished reading the matrices; and process P 3 must not start, until process P 2 has finished the multiplication. We notice that in this case, the process is P 1, P 2 and P 3 are not sending values to each other. There is no need for them to communicate values between each other at all; there is no need for like, the value 6 which had to go between the parent and the child in the previous case. Here, all that we have to ensure is, the process P 2 does not start, until process P 1 has finish, and that process P 3 does not start, until process P 2 has finished its task.

Now, this may look like a trivial example, since it is fairly clear that by setting up my concurrent program in this way, I may get no benefit. In that process P 1 will finish, then process P 2 will finish; then process P 3 will finish, and even if I have done all the activity in just process p 1, it could have taken possibly the same or less time. But this was just an example of an extreme case of where process, I my setup a concurrent program, but very clearly in example where there is no need for data communication, but nearly for the activities of process is to be synchronized with each other.

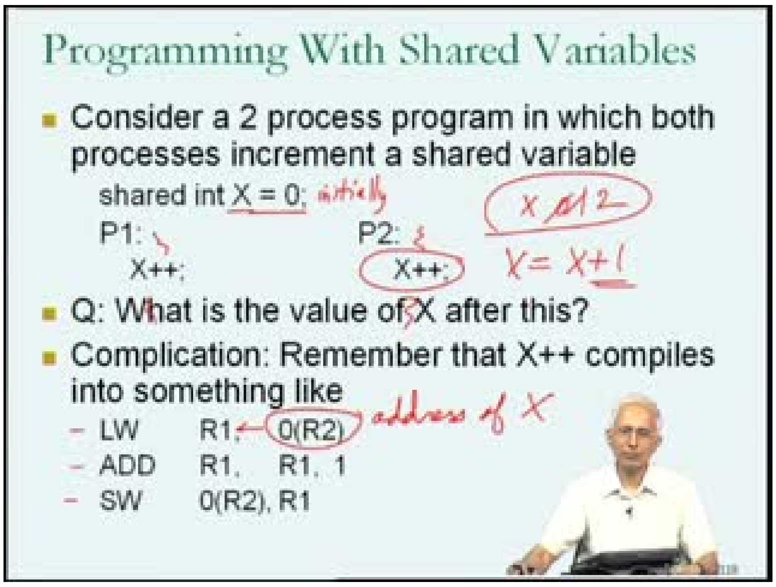
So, if this is the case, then they are likely to be IPC mechanisms - Inter Process Communication mechanisms, which do not have to handle the transfer of data, but nearly have to handle synchronization. In other words, coordination between processes to make sure that one does not proceed beyond a certain point in its activity, until one of the others has reach, let say, a certain point in its activity.

In this particular example, we did not want process P 2 to start its work, until process P 1 had finished its work and so on. So, this is what we refer to as process synchronization; the idea that 1 process that the 2 processes P 1 and P 2 may have to be setup, so that P 2 waits for P 1 or that they are synchronize with each other not to proceed in definitely, until the synchronization had been achieved.

Now, in practice, to do synchronization of this kind, some kind of mechanisms would have to be provided; typically, they are provided by in operating system. Some of the mechanisms can be implemented by users using mechanisms, that are available in user mode alone. But in general, people may talk about synchronization primitives; the word primitive is use to suggest that they are simple operations which are provided, they do not have to be implemented, but they are provided, and therefore, they are refer to as primitives - synchronization primitives or primitive synchronization operations. Some of the names of, some of these synchronization primitives are mutex lock, semaphore, barrier, we will see each of these in a little bit more detail; some in this class, some of the classes to follow.

So, just to recap, we are on the situation where we have talking about inter process communication, situation where it is not necessary for explicit values to be communicated, but nearly for synchronization between processes; so, that is the fifth possibility. Just to remind, you have seen four others prior to this and they all related to how explicit values could be communicated between processes.

(Refer Slide Time: 34:55)



Programming With Shared Variables

- Consider a 2 process program in which both processes increment a shared variable

```
shared int X = 0;
P1:                P2:
   X++;               X++;
```

- Q: What is the value of X after this?
- Complication: Remember that X++ compiles into something like
  - LW R1, 0(R2)
  - ADD R1, R1, 1
  - SW 0(R2), R1

Now, moving right along, let suppose that we are in a situation, where the assumption is that, processes that we have are communicating with each other and explicitly communicating values with each other using shared variables; this was the third of the options that we had in the list. The first was through files, the second was through pipes, the third was through shared variables, the fourth was through message passing, and the fifth was near idea of synchronization.

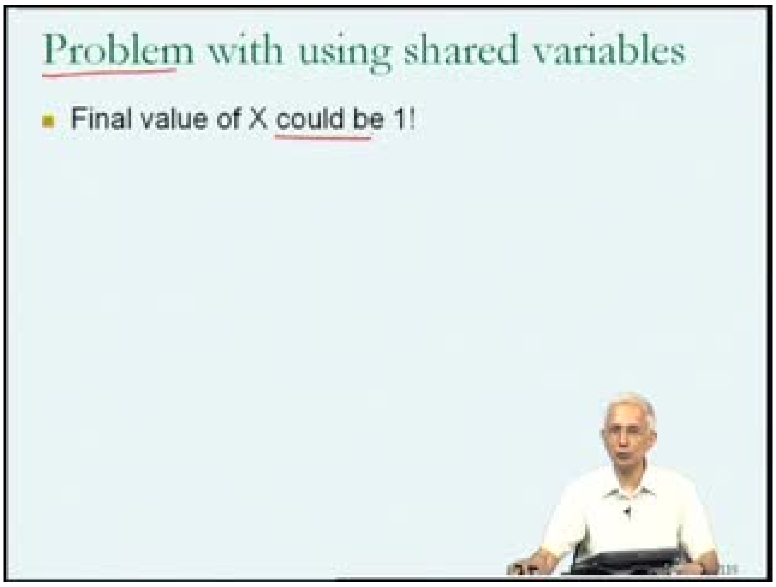
So, we are going to back to the idea 3. We are going to spend a little bit of time talking about concurrent programming using shared variables. So, the idea something like this we have seen, if there was a, let us suppose a two process program and we look at a very simple example, because where the objective of this slide is to illustrate a problem which arises, which we must be aware of if you want to do concurrent programming with shared variables.

So, let us suppose, we have a two process program, a program which is running as two processes, possibly parent and child, in which both the processes increment a shared variable. In other words, there is one shared variable called x, which happens to initially be 0, and what each of these processes does is to increment x and I indicate that by the C X plus plus. So, in addition to other things which they may be doing, so they may be doing other activity as well, but each of the processes is going to increment the shared variable X.

Now, when you look at this concurrent program, you will ask the question what should be the final value of the variable x; and it should be clear in your mind, that since x was initially 0, so this is the initial value of x and x was incremented twice, we suspect that the final value of the shared variable x should be 2. We ask the question, what should the final value of the variable x be after this, and we suspect that the final value of the variable x should be 2, because one of these increments is going to make it, is going to change it from 0 to 1, the other increment is going to change it from 1 to 2. So, we suspect that the final value should be 2.

However, there is a complication and the complication arises from the following; you will remember that, when I say increments the variable using x plus plus, that is actually a little bit more complicated than just incrementing the variable. If I think about something like the mips instruction set, because I know that x plus plus is really x equals x plus 1; and in order to do x equals x plus 1, the first thing that is going to happen is to load the value of the variable x into a register, so this might be the address of the variable x - the shared variable x - subsequently I loaded into the register R1, subsequently I increment register R1, and finally I update the variable x using a store word instruction. So, this operation of incrementing the variable x is actually going to being the mips, if the processor on which these two concurrent programs are running is a mips one processor, then it is going to compiling to this sequence of three instructions.

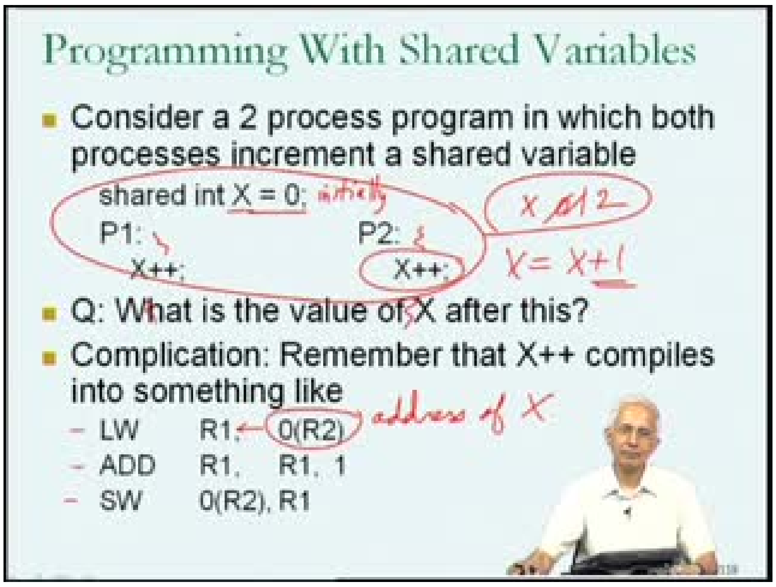
(Refer Slide Time: 38:08)



Problem with using shared variables

- Final value of X could be 1!

(Refer Slide Time: 38:42)



Programming With Shared Variables

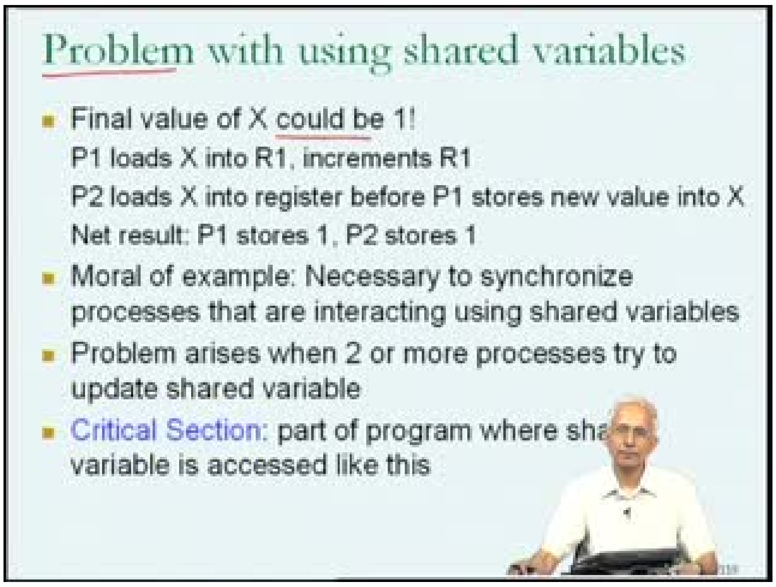
- Consider a 2 process program in which both processes increment a shared variable

```
  shared int X = 0;  initially
  P1:                      P2:
     X++;                     X++;
```

X = 2
X = X+1
address of X

- Q: What is the value of X after this?
- Complication: Remember that X++ compiles into something like
  - LW R1, 0(R2)
  - ADD R1, R1, 1
  - SW 0(R2), R1

(Refer Slide Time: 38:50)



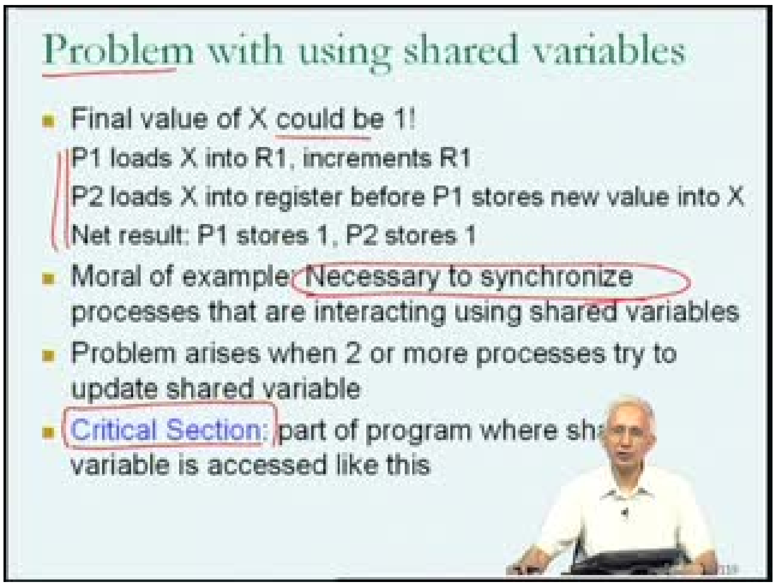
Problem with using shared variables

- Final value of X could be 1!
  P1 loads X into R1, increments R1
  P2 loads X into register before P1 stores new value into X
  Net result: P1 stores 1, P2 stores 1
- Moral of example: Necessary to synchronize processes that are interacting using shared variables
- Problem arises when 2 or more processes try to update shared variable
- Critical Section: part of program where sha variable is accessed like this

Now, let us just see, what could happen when these two processes end up running on that processor. So, I am now formally describing this as a problem with using shared variables, so you must at this point anticipate something is going to go wrong. But let me just point out right now, that unfortunately the final value of the variable x could end up being 1; there is no guarantee that is going to end up at 2; it could end up being 2, but on the other hand, it could end up being 1; it cannot end up being 0, but could end up being either 1 or 2. And this is not a good piece of news, because typically when you write a program that looks like this, your idea may have been that, you wanted the variable x to end up with the variable 2, that is why you incremented it twice.

But you now told, that if you wrote the concurrent program which runs as a on a single processor, using that kind of code you may end up with a variable x having a value one, and the way that this could happen is the following. It is possible that, well, we know that the sequence of events as described by that mips - machine language instructions - was the P 1 loads x into R 1 and then increments R 1; it is possible that P 2 then loads x into register, before P 1 stores a new value into the variable x, and the net result with then be the P 1 stores the value 1, P 2 stores the value 1. Let me just how trying this a little bit more detail.

(Refer Slide Time: 39:33)



Problem with using shared variables

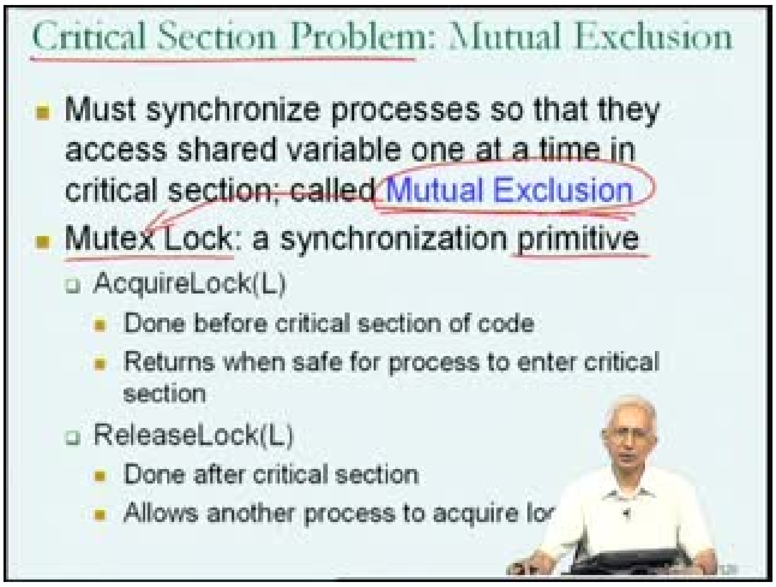
- Final value of X could be 1!
  - P1 loads X into R1, increments R1
  - P2 loads X into register before P1 stores new value into X
  - Net result: P1 stores 1, P2 stores 1
- Moral of example: Necessary to synchronize processes that are interacting using shared variables
- Problem arises when 2 or more processes try to update shared variable
- Critical Section: part of program where sh variable is accessed like this

So, that the consequence is clear; we will come back to that slide shortly. Well, let me just, may be go to through slide in order and they will come to this particular example in more detail in a few seconds. Now, if this is the case and as I said we will run through this a little bit more detail shortly. This is a very dangerous situation, because we have to be careful in writing these kinds of concurrent programs. We must be aware that, it may be necessary to synchronize the processes that are interacting using shared variables. And the problem seems to have a reason, because process P 1 and process P 2 are sharing one processor, only one of them can be running at a time and the processor the operating system may switch from process P 1 to process P 2, somewhere in between, the sequence of events that is going to happen.

So, if you have to going to be using concurrent programming using shared variables, it may end of being necessary to synchronize the processes to avoid these kinds of wrong updates of shared variables. And the problem seems to arise at least in this example, because I had two or more processes which are trying to update a shared variable. And therefore, we will call parts of the concurrent programs where shared variables are accessed like this, we call them critical sections of the program. They are the important parts of the concurrent program, which may have to be handle carefully, maybe using some kind of synchronization primitives, in order to cause a program to do what we think it is going to do.

So, in short, if we program using shared variables, then we may be have to treat the regions of the program which deal with the shared variables carefully; and hence, we will refer to them as critical sections. Once again this is in blue, so just think that it is a commonly used technical term.

(Refer Slide Time: 41:20)



Critical Section Problem: Mutual Exclusion

- Must synchronize processes so that they access shared variable one at a time in critical section; called Mutual Exclusion
- Mutex Lock: a synchronization primitive
  - AcquireLock(L)
    - Done before critical section of code
    - Returns when safe for process to enter critical section
  - ReleaseLock(L)
    - Done after critical section
    - Allows another process to acquire lo

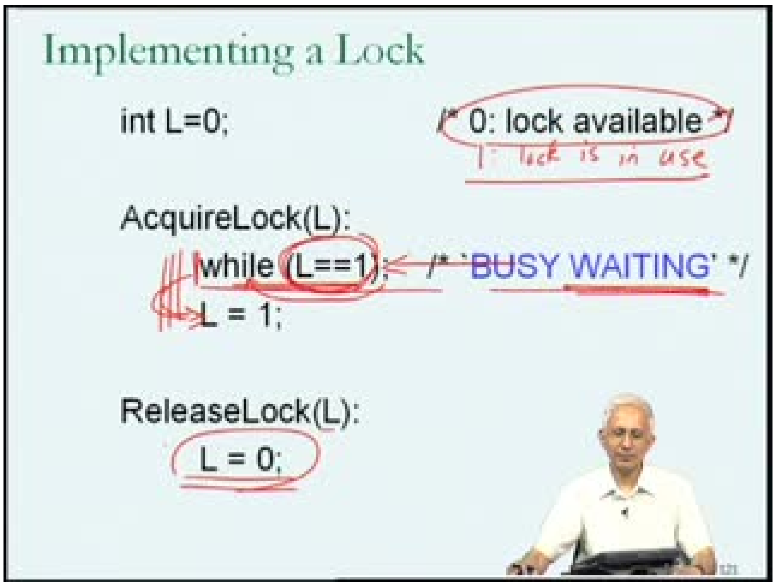
Now, the problem which I will take about next is, explicitly, what has to be done in a critical section to ensure that the result of the concurrent program is what I may have thought. In other words, this example, conceptually I thought that the result of incrementing x twice should be that x ends up with 2. Now, what is the problem with the critical section? The problem with the critical section was that, I did not seem to be properly synchronizing the activity of process P 1 and process P 2 in such a way that, they could safely update the variable x. Now, in general, it may be the case that within these kinds of critical sections, the synchronization must be done, so that the accesses to shared variables are done one at a time. In other words, this notion of allowing one process to access the shared variable mutually exclusively of the other process.

In other words, do not allow the two processes to actually be inside these critical sections, during reasons of time that the others are also in the critical section. The name given for a synchronization primitive that can be used in such a scenario, is to describe it as a mutex lock. Mutex is the abbreviation for mutual exclusion and lock is the name for the mechanism suggesting that it is going to ensure that the problem does not arise, the problem of strange values ending up in the shared variables. Now, the general idea behind how one could use a mutex lock is, the mutex lock we can view it again as a data structure, possibly provided by the operating system; the operations to this data structure are provided by the operating system.

I will describe the two operations on the mutex lock: acquire lock and release lock. And the general idea is that, before entering a critical section, in other words, the regions of my concurrent program that are critical sections, I must put acquire lock call before the critical section, I release lock call after the critical section; and in a sense, lock the critical section code to solve this problem and make sure that only one process is executing inside that piece of code at a time.

So, the idea is that one we call acquire lock before the critical section. And it will provide some kind of a guarantee, so that controllable transfer back from acquire lock, only when it is safe for the process to enter the critical section. And similarly, release lock will be called after the critical section and will allow another process to acquire the lock. This is a general mode of operation or the mutual exclusion primitive called the mutex lock.

(Refer Slide Time: 43:52)



Implementing a Lock

```
int L=0;                /* 0: lock available */
                           1: lock is in use

AcquireLock(L):
    while (L==1);  <-  /* 'BUSY WAITING' */
    L = 1;

ReleaseLock(L):
    L = 0;
```

Now, let me suggest how one could implement a lock and this will give us a better understanding of the need for synchronization. So basically, I have to talk about the implementation of a function called acquire lock, and the implementation of a function called release lock; so, let us just do that. I will think of the lock as I said as being some kind of a data structure.

So, the acquire lock takes a lock as a parameter, release lock takes a lock as a parameter; these are not value parameters, but reference parameters. Suppose, that the lock variable is an integer variable, and initially it has the value 0, and the way that this implementation is going to work is that, I will initialize the lock to 0 and the meaning of the lock being 0 is that the lock is available.

If lock has a value of 1, that is going to mean that the lock is not available or the lock is in use. So, as far as using the lock is concerned, the lock initially has a value of 0 and we understood that the release lock operation was going to be used to make the lock available for the next process to acquire. Therefore, we suspect all that has to happen in the release lock primitive is to set the lock variable to 0, this will make the lock available, which was if you look back at the previous slide, that is what we wanted the release lock function to do; we wanted it to allow another process to acquire the lock, which means that it should make the lock available and it will do so by setting the variable L to 0.

Now, what should the acquire lock primitive do? Basically, it will call, before a process enters the critical section; and what is just supposed to do is, it should return, only when it is safe for the process to enter the critical section. In other words, it should return, only when the value of the lock variable is 0 and also it should set the lock variable to 1, so to prevent other processes from entering the critical section.

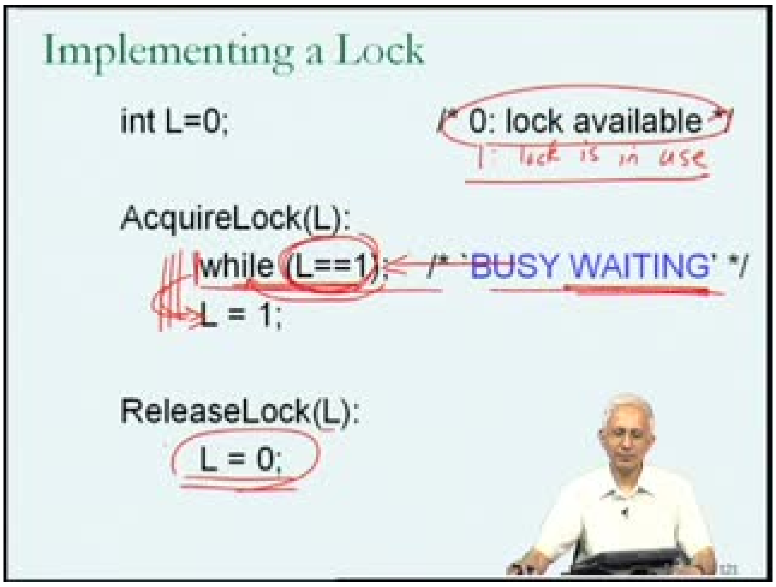
So, in a sense what they acquire lock function primitive is going to have to do is, as long as L is equal to 1, in other words, as long as the lock is not available, it has to do nothing, it has to just wait. So, what I am showing you over here is an example of a loop. As long as L is equal to 1, it has to loop; and therefore, this loop just keeps executing while L equal to 1, until L becomes equal to 0. How could L become to 0? L can become equal to 0, because some other process executes set L equal to 0.

Now, after the acquire lock, the process which is executing acquire lock, comes out of the while loop, that means, the L is equal to 0. And then the process which wanted to acquire lock can set L equal to 1, thereby preventing other processes from observing that the lock is available; and therefore, this could be viewed as being a implementation of acquire lock.

Remember the objective of acquire lock was, to keep a process which is trying to get the lock waiting, until it is safe for the process to acquire the lock; and we understood that the lock is available if L equals to 0, which is why we kept waiting until L became 0. As soon as L became 0, we got out of the waiting loop and set L equal to 1, which meant informs other processes who may be checking that the lock is now in use.

Now, unfortunately, this is not a correct implementation, but before I come to that, this idea of looping, executing the checking on L equal to 1 repeatedly, until L becomes equal to 0 is what I will prefer to as busy waiting, since it is a waiting loop. But this is the waiting loop in which some activity is happening, to the process which is executing this loop will be fetching and decoding and executing instructions.

(Refer Slide Time: 43:52)



Implementing a Lock

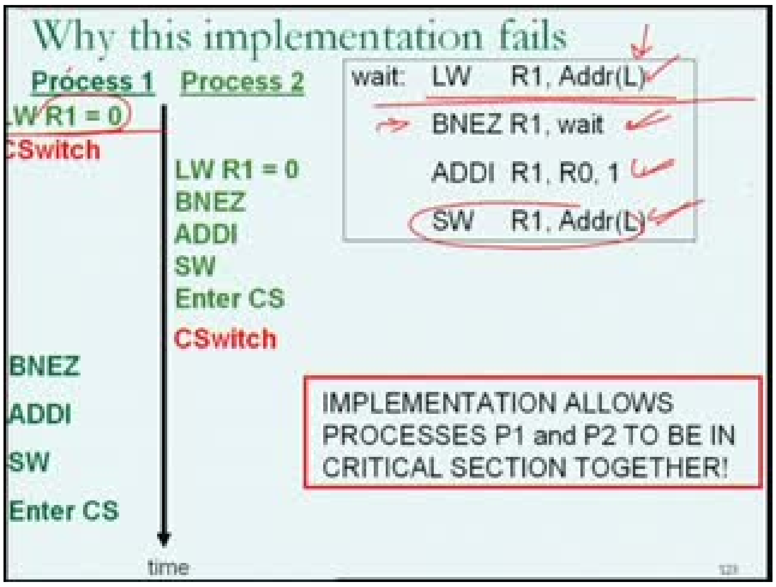
```
int L=0;                    /* 0: lock available */
                               1: lock is in use

AcquireLock(L):
    while (L==1);           /* 'BUSY WAITING' */
    L = 1;

ReleaseLock(L):
    L = 0;
```

In other words, it will be keeping the processor busy and which is why it is known as busy waiting. It is not a form of waiting, whether operating system could say this process is waiting, it is doing nothing, and therefore should be put into the waiting state. Rather, it is a kind of waiting in which the process is successfully executing instructions, but the instructions are such that it is just waiting for something to happen. It is indefinitely check, it is keeping on executing this check on the value of the variable L, until L becomes equal to 0, and that will happens soon or later, when some other process releases the lock L, specifically if there is a preemptive scheduling policy.

(Refer Slide Time: 48:15)


Why this implementation fails

```
wait:  LW    R1, Addr(L)
       BNEZ  R1, wait
       ADDI  R1, R0, 1
       SW    R1, Addr(L)
```

```
Process 1      Process 2
LW R1 = 0
CSwitch
               LW R1 = 0
               BNEZ
               ADDI
               SW
               Enter CS
               CSwitch
BNEZ
ADDI
SW
Enter CS
                   time
```

IMPLEMENTATION ALLOWS PROCESSES P1 and P2 TO BE IN CRITICAL SECTION TOGETHER!

Now, I had suggested that this does not work; and let me try to explain, why this implementation fails. Now, in understanding why this implementation fails, we will get a better understanding of the nature of the critical section problem itself.

Now, the implementation that we had for acquire lock look like this; while L was equal to 1, we wanted to busy wait, so do nothing and this is the busy wait loop; as soon as L becomes equal to 0, set L equal to 1. Now, just like we looked at the increment of x in terms of the mips instructions that it might compile into, let us look at this piece of code in terms of the mips instructions that it might compiled into, and what will end up is conceivably something like this.

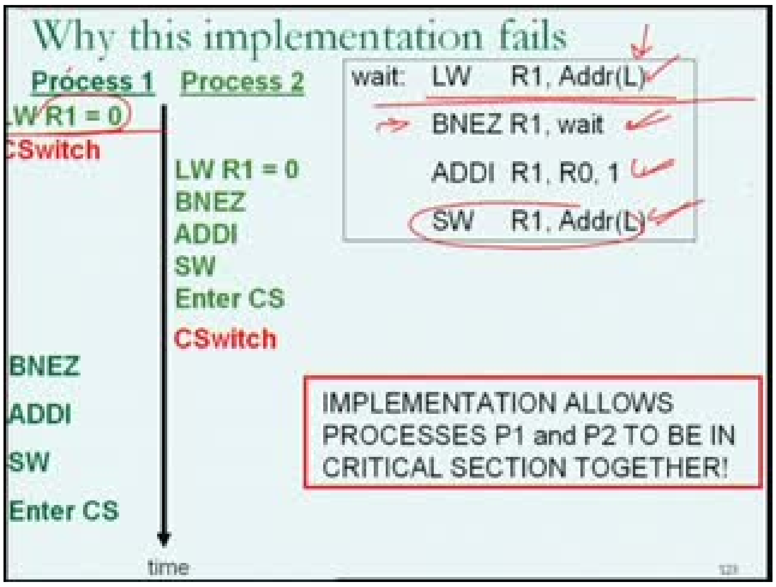
So, I have a load word instruction. So the idea here is, I am going to load the value of L into a register, then I am going to compare that with 0 and if it is not equal to 0 a loop. So, the loop that you see over here is, the loop above; the instruction that you see before are setting R 1 equal to 1 and then storing that value into the lock variable. So, this is the L equal to 1.

So, the first two mips instructions relate to the busy wait loop; the second two mips instructions relate to certain L equal to 1. So, this looks like a reasonable compilation of the sequence of the C code, that we see on the left.

Now, the situation that we are in is, let me just get out of this particular mark up. The situation that we are in is, we have this piece of code, and we understand that this is what is going to happen, when a process executes the acquire lock function; and we need no longer worry about the C instructions on the left, so I will get rid of the C instructions on the left. And suppose that there are 2 processes: process P 1 and P 2, sorry. Let me assume that the lock is currently available, in other words, L is equal to 0 and that there are two processes P 1 and P 2 which are trying to acquire the lock.

So, the two processes which want to acquire the lock, because both of them want to access the shared variable. So, I am going to show you time lines for process P 1 and P 2; I am going to draw the time lines vertically, because I am going to mark the time lines with instructions that are being executed.

(Refer Slide Time: 48:15)



Why this implementation fails

Process 1	Process 2
LW R1 = 0	
CSwitch	
	LW R1 = 0
	BNEZ
	ADDI
	SW
	Enter CS
	CSwitch
BNEZ	
ADDI	
SW	
Enter CS	

```
wait:  LW    R1, Addr(L)
       BNEZ  R1, wait
       ADDI  R1, R0, 1
       SW    R1, Addr(L)
```

IMPLEMENTATION ALLOWS PROCESSES P1 and P2 TO BE IN CRITICAL SECTION TOGETHER!

So let suppose that, currently process P 1 is executing, and therefore it is starts by executing the first instruction of this, it is executing in the acquire lock function and it executes the first instruction, so it loads the value of the lock into the register R 1; so, it gets R 1 equals 0. Let suppose that at this point in time, there is a context switch. Now, remember, that the operating system at the end of the time slice of a process will switch from that process to another one of the other ready processes.

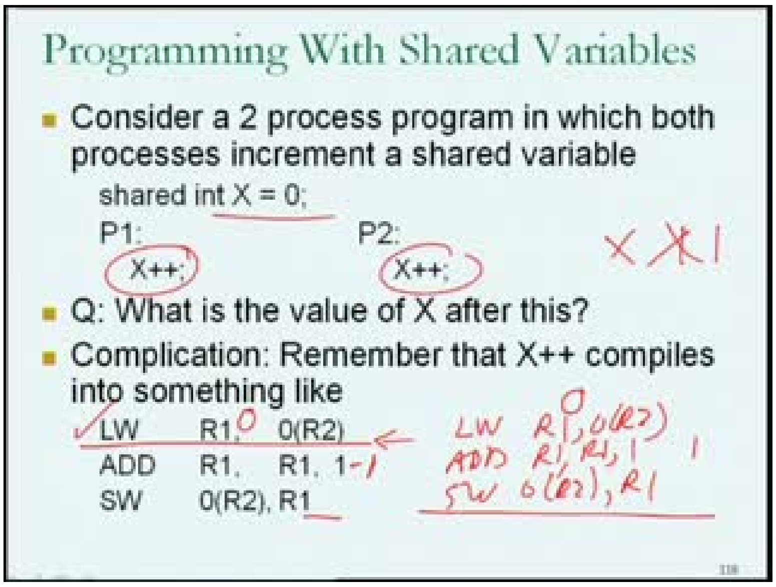
Now it is quite possible that a context switch could happen, after process P 1 has executed this first instruction; so, this is the perfectly legitimate possibility. Let suppose that the operating system now makes process P 2 the running process, process P 1 has been put into the ready queue. Process P 2 also is executing the sequence of instructions. Let suppose, that its successfully executes all the four instructions, but it is going to find out is, that it will read into its register R 1, the value L equal to 0, just like process P 1 did.

Note that process P 1 had read the value of L equal to 0, but did not go to the subsequent statements and modify the value of L. Consequently, when process P 2 started executing, it too reads the value of L equal to 0, but goes ahead and modifies the value of L, and therefore, enters its critical section. So, process P 2 has successfully acquire the lock and is executing in its critical section.

Let suppose that at this point in time, process P 2 suffers the context switch, and that process P 2 becomes the ready process, and process P 1 becomes the running process. Now, when process P 1 resumes its execution, it would be at a point where its PC value will be such that, it will execute the branch if not equal to 0 instructions. So, it does that, in fact, it goes from the branch if not equal to 0, it checks the value of R 1; as far as it is concerned R 1 is equal to 0, because that I have successfully loaded R 1 with the value of L at that point in time; and therefore, it too successfully executes these instructions and enters the critical section.

So, we are now in a situation, where we see that this implementation can allow two processes to be executing the critical section at the same time; and therefore, it does not provide the kind of guarantees that we were looking for.

(Refer Slide Time: 53:15)



Programming With Shared Variables

- Consider a 2 process program in which both processes increment a shared variable

```
shared int X = 0;
P1:              P2:
  X++;             X++;
```

- Q: What is the value of X after this?
- Complication: Remember that X++ compiles into something like

```
LW     R1,  0(R2)
ADD    R1,  R1, 1
SW     0(R2), R1
```

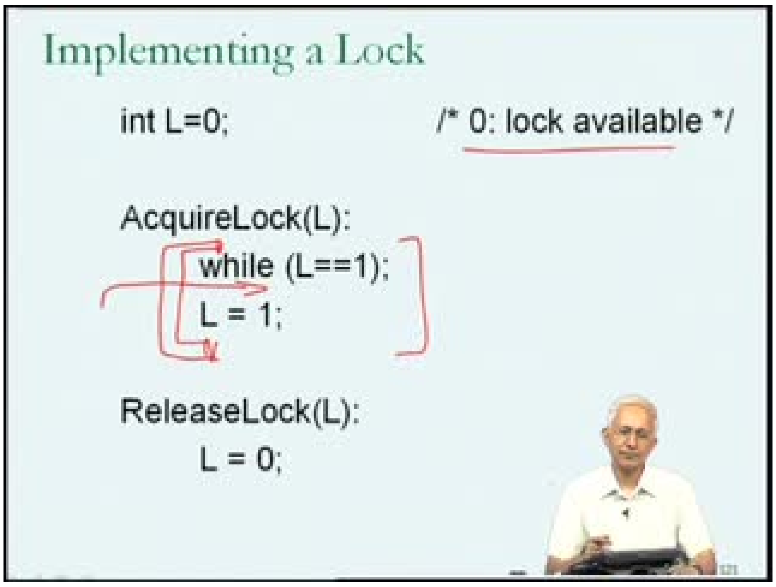
X X 1
0
-1
LW R1, 0(R2)
ADD R1, R1, 1
SW 0(R2), R1
0
1

This in fact give us some idea of why the earlier example that we had seen, was also has this example. Let me just go back to the earlier example. So, we had a situation, where process P 1 and process P 2 are both executing this piece of code. And let us suppose that process P 1 executes initially X is equal to 0, process P 1 executes load word, so that as far as it is concerned, its register R 1 contains the value 0.

Let suppose, that at that point in time, there is a context switch and process P 2 starts executing, so process P 2 also loads from the variable X, which can still contains the value 0; so, we store the value 0 into R 1; it increments its R 1 by 1 to make R 1 equal to 1; it stores the value of R 1 into the variable X, so that X becomes equal to 1.

Let suppose at this point in time there is a context switch back to process P 1; P 1 is at a point where it has a value of 0 in its register R 1; it increments R 1 by 1 to come up with the value of R 1 equal to 1 and its stores this value of R 1 into the variable X, overwriting the value 1 which was written there by process P 2 with a new value 1; which is how it was possible for two processes to implement the variable X, but still end up with the situation, where the value the variable X was only 1 and not 2. Now, this was the problem with the variable X, and that was the reason which we wanted to have a solution to the problem in the form of an operating system support, such as a lock; and we define the idea of the critical section.

(Refer Slide Time: 54:54)



Implementing a Lock

```
int L=0;                    /* 0: lock available */

AcquireLock(L):
    while (L==1);
    L = 1;

ReleaseLock(L):
    L = 0;
```

But in doing so apparently, we came up with an implementation, which was inadequate. This particular implementation is inadequate and if you look at this implementation we realize why. We realize that in this implementation, this itself is a critical section. This is a region of code, where if there is a context switch in between, the consequences can be that the overall effect is not what we would have wanted; that by trying to implement operating system primitive in which there is a source of problems in itself, we are not solving any problem at all.

Consequently, very clearly, we need to have solutions or implementations of things like the lock, which go beyond this, seemingly correct interpretation of how to safely make a lock available and possibly requiring hardware support to do the same.

We will be continuing in the next lecture. We will look at a specific implementation, which requires hardware support in order to implement a mutex lock, but is correct, and can therefore be used as the basis for writing programs - concurrent programs - which use shared variables, in order to facilitate the communication; and therefore the cooperation of multiple processes in working towards a common objective.

Thank you.