

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

Module No. # 05

Lecture No. # 21

This is lecture 21, of the course on High Performance Computing. In lecture 20, we had looked at some of the problems and solutions to those problems relating to concurrent programming. We saw how locks which are known as pin locks or busy weight locks, can be used to, and first, something called mutual exclusion through which shared variables between processes can be safely used.

We also saw how something called the semaphore can be used to synchronize processes to coordinate the progress of processes, in order to achieve the common objective in a concurrent program. Just you remind about an example that we saw relating to semaphores, the example that we had looked at, had the situation where they were three processes, process P 1, P 2 and P 3 which were cooperating towards the reading in multiplication and **outputting** of a matrix multiplication.

(Refer Slide Time: 01:19)

Semaphore Examples

- Semaphores can do more than mutex locks
- Example: Consider our concurrent program where process P1 reads 2 matrices; process P2 multiplies them & process P3 outputs the product

□ Semaphores $S_1 = 0$ $S_2 = 0$

<u>Process P1</u>	<u>Process P2</u>	<u>Process P3</u>
<u>Read A[], B[]</u>	<u>$C[] = A[] * B[]$</u>	<u>Write C[]</u>
<u>V(S₁)</u>	<u>V(S₂)</u>	

118

We saw how two semaphores could be used to synchronize the work as divided among the three processes. For example, process P 1 which is reading the matrices, process P 2 which is multiplying them and process P 3 which is writing the result, it could be set up, so that a semaphore initialized to 0 could be used to block process P 2, until process P 1 had finished the reading; and another semaphore which had been initialize to 0, could be used to block process P 3 from writing the result, until process P 2 had successfully multiplied the matrices. So, with a little bit of thought, one could use semaphores for different kinds of synchronization of processes which are cooperating towards a common objective.

(Refer Slide Time: 02:02)

Deadlock $L=1$ $\text{while}(\text{TestSet}(L));$

Consider the following process:

P1: AcquireLock(L); AcquireLock(L);

- Suppose that the first AcquireLock(L) succeeds
- P1 is then waiting for something (release of lock that it is holding) that will never happen
- This is a simple example of a general problem called **deadlock**
- Caused by a cycle of processes waiting for resources held by others while holding resources needed by others

Dead

140

Now, moving right ahead with some of the problems relating to concurrent programming, I have talked about potential problems that could arise due to the vigorous of shared variables and we saw that if one just wrote concurrent programs using shared variables, and did not identify and suitably protect the critical sections of the program, then the results could be somewhat unpredictable. The program may not do what you thought it was going to do, that is what the mutual exclusion problem and need to identify critical sections came up.

Now, other problem which could arise in concurrent programs is a problem known as deadlock and as a name suggest, it is somewhat serious problem. Now, to understand deadlock, let me just give you a very simple example of a deadlock situation. Let us

suppose that, I have a process which I am referring to as P 1 - I am just showing you a small part of the activity of that process - you will notice that it starts by acquiring a lock, a lock called L and the next thing that it does is to acquire the lock again.

It is not too clear what this process was written to achieve, but if we think about this little bit, we realize from our understanding of the implementation of acquire lock, that let suppose the process P 1 is lucky in his first call to acquire lock, let suppose that the lock is currently available. Then what will happen is the net effect of acquire lock, if it is implemented using let us say test and set of L is going to be that, the process is going to test and set L, so that the old value of L which would have been 0, will get changed to 1.

The process will then proceed to its next statement, which is acquiring lock L again, and we know that since the lock L is currently held in fact, by process P 1, the second call to acquire lock is just going to keep on executing, while test and set of L forever. We know that this is not a situation in which the value of L is going to get changed, because the only possibility for the value of L to get change is where this very same process executes, release lock. This process every time it executes, it just going to keep on executing acquire lock busy weighting on this loop inside the acquire lock function.

Now, this is in itself is not a dangerous situation, because we know that acquire lock involve busy weighting and after the process as used the CPU for some amount of time, it could be preempted and we Leal the CPU. But, they could be a most serious situation if the acquire lock function is not written using busy weighting, but is using the blocking idea, which I had talked about last time.

Let suppose it has a situation, where this is a blocking scenario; then as far as process P 1 is concerned, process P 1 is a situation where it is holding a lock and subsequently is trying to acquire the lock, but will be blocked until the lock becomes available. The only process which can release the lock is process P 1.

We know that is never going to release the lock because it is currently blocked, weighting for the lock to become available. Therefore, this is a situation which will never happen, we know that this will never happened and this is happen because of in correct use of the lock.

When we first looked at this piece of code, I suggest that is not clear what the programmer has in mind because it is not easy to understand how a single process could try to acquire a lock and while holding the lock, try to acquire the lock again, it does not make any sense, it is not logical. Therefore, this is very clearly a wrong piece of code, but this is just as simple illustration of a more general problem, which could arise if there are multiple processes and this situation is known as a deadlock.

The situation could arise if there are multiple processes, each of which is trying to acquire a resource may be using a semaphore and that resource is held by an another one of the processes, in such a way that there is a cycle of processes, each of which is waiting for something to happen that can only be achieved by the next in line, in that cycle of processes.

This ends up in what is known as a deadly embrace, where there is a group of processes which are in a sense blocked and locked, waiting for something that only one of the others in the group can do, which in turns waiting for something that one of the others in the group can do, guarantee that none of these processes can progress ever. So, this is much more dangerous in an infinite loop because that is a guarantee that the processes are not going to make any progress.

Now, deadlocks are a serious problem that could arise in writing concurrent programs and they are serious in that unless the deadlock is understood and eliminated, the program is not going to achieve its objective. We will talk about deadlocks again a little later; I wanted to introduce the idea.

(Refer Slide Time: 06:58)

Classical Problems

Producers-Consumers Problem

- Bounded buffer problem
- Producer process makes things and puts them into a fixed size shared buffer (array)
- Consumer process takes things out of shared buffer and uses them

Must ensure that producer doesn't put into full buffer or consumer take out of empty buffer

While treating buffer accesses as critical section

Since we are talking about potential problems, that one is to be aware of in concurrent programming. Now, just to tie in the concepts of concurrent programming, it is often a teaching technique that is used in concurrent programming classes to look at some standard problems and analyze how they could be solved using the constructs of concurrent programming. These are known as some of the classical problems of concurrent programming; one of these is known as the Producers-Consumers problem, I thought I would run through this.

This also sometimes known as the bounded buffer problem, the nature of the problem is the following. The situation is one where I have one or more processes, which are producing; I refer to them as a producer process.

So, the property of the producer process is that, it produces something and then puts the thing that has been produced into a fixed size array or buffer. So, when I use the word buffer, I am referring to something which we will represent as an array in the program. The property of the array is that, it is a shared array and that it is of some fixed size.

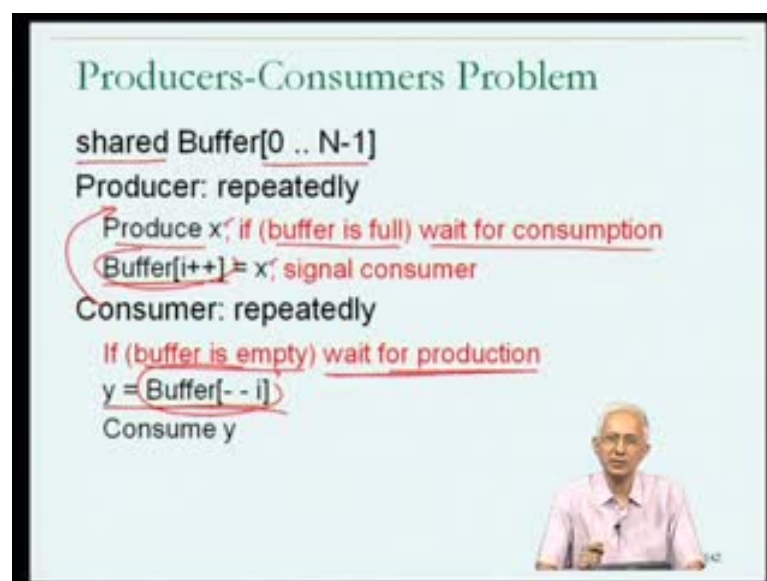
Now, the other entities in this concurrent program, but the other entity in this concurrent program is what I will refer to as a consumer and as the name suggest, what a consumer process does is, it take something out of the shared buffer, that is why it is called a consumer and it uses that whatever the thing was.

Now, the problem that could arise is that, if I write a concurrent program **to solve** to handle consumer producer like situations as two processes; one is a producer process, one is the consumer process. I just think about the code of the consumer process, the code of the consumer process is going to try to read something out of a shared buffer or a shared array and use that thing.

But unless, the producer process has already produced something and put it into the buffer, there may be nothing in the consumer process for the consumer process to consume. Similarly, **if the consumer process** if the producer processes has something which it has made, and would like to push it into the buffer, but up to now the consumer process has not removed anything from the buffer and the buffer is therefore full, then the producer process may not just be able to insert the newly created thing into the shared buffer.

Hence, there is a need to synchronize the activities of the producer and the consumer to make sure. For example, that the producer does not put into a full buffer or the consumer does not take out of an empty buffer. These are what may be viewed as synchronization requirements between the producer process and the consumer process. Therefore, in some sense, it is a useful example to try to write code for in terms of a concurrent programming exercise.

(Refer Slide Time: 10:10)



Producers-Consumers Problem

shared Buffer[0 .. N-1]

Producer: repeatedly

Produce x; if (buffer is full) wait for consumption

Buffer[i++] = x; signal consumer

Consumer: repeatedly

If (buffer is empty) wait for production

y = Buffer[--i]

Consume y

© 2012

The other aspect which is interesting here is that, the producer and the consumer not only have to be synchronizing, but they also have shared variables which they are both accessing. Specifically, the shared variable that they have is the shared buffer and therefore, we have to understand that in writing the producer consumer concurrent program, we have to treat accesses to the buffer has a critical section. Therefore, there are in addition to the synchronization requirements there is a mutual exclusion requirement. Therefore, this is a nice problem to understand these concepts more fully; just look at a possible encoding of the problem.

So, I describe the shared buffer by - some this is not exactly c code, it is not exactly any program, but a programming language, but will be added could for our purposes - the sudo code form.

So, I have a shared buffer and array which I describe as buffer; it is a fixed size, the size of this particular buffer is n . So, i number the elements from 0 through n minus 1 and I have a producer process, which is going to execute a loop in which it repeatedly produces a thing, which I will call x and then inserts that thing into the buffer.

So, associated with the buffer, I have an index variable call i which will let me know the current available slot in the buffer. So, I will initialize i to 0 and then subsequently whenever something is put into the buffer, increment i which is why there is i plus plus associated with the producer and the producer does this repeatedly.

What is the consumer do? The consumer once again repeatedly remove something from the buffer, in order to do that it decrements i and looks at the object inside that element of the buffer or array, put silent to a local variable call y and consumes y .

So, this is the basic activity which we have talked about, now there is a need to enforce the mutual exclusion requirement on the accesses to the shared variables. There is also a need to enforce the synchronization requirements of not making sure that the producer and the consumer do not produce and consume from empty or full buffers etcetera.

Now, one of the requirements that we have is that when the producer produces an object x , it must obviously, check to see if the buffer is full. If the buffer is full, then it has to wait for at least one consumption of an object to happen, that will ensure that one

element on the buffer is available; therefore, the producer can push the x that it has produce into the buffer, can insert the x it has produced into the buffer. So, that is one of our requirements.

The other requirement is that prior to thinking that it can take something out of the buffer, the consumer will have to check if the buffer is empty and if the buffer is empty, it will have to wait - in other words, it cannot just proceed after the if condition, it will have to wait - until at least one element has been produced. So, it has to wait for at least one production.

Now, we notice that there is the weighting for consumption and there is a weighting for a production. Now think about the consumption, when does something get consumed? Something gets consumed, when it is taken out of the buffer by the consumer. When does something get produced? It gets produced, when it is put into the buffer by the producer. Therefore, we can think of these requirements as being synchronization requirements between the processors, between the processes is producer and consumer, which could conceivably be handle using semaphores.

(Refer Slide Time: 13:10)

The slide is titled "Dining Philosophers Problem" in a green font. It contains a bulleted list of three items: "N philosophers sitting around a circular table with a plate of food in front of each and a fork between each 2 philosophers", "Philosopher does: repeatedly Eat (using 2 forks) Think", and "Problem: Avoid deadlock; be fair". To the right of the text is a diagram of a circular table with 10 seats, each with a plate and a fork. A speaker is visible at the bottom of the slide.

- N philosophers sitting around a circular table with a plate of food in front of each and a fork between each 2 philosophers
- Philosopher does: repeatedly
Eat (using 2 forks)
Think
- Problem: Avoid deadlock; be fair

Now, another problem which people often talk about as a concurrent programming problem is, what is called the dinning philosophers problems. It is a rather curious problem. Problem is stated as follows: let us suppose there is circular table and there are

n philosophers, n people sitting around the table, each has a plate of a food in front of him or her and there is a fork on the table between each two philosophers.

So, the setting is as over here (Refer Slide Time: 13:36). There are n philosophers, in this example n is equal to 5 and a plate in front of each with food on it. Unfortunately, there are not enough forks, we assume that each philosopher needs two forks to eat with, that is the suggestion; but, there are only n forks on the table, which means that the forks are shared resource which is scarce.

They call philosophers because all that the philosophers do is repeatedly, in other words, this is a loop; they think and then they eat. In order to eat, they have to require two forks, just eat - they have to get the fork on the left and the fork on the right, then eat and after having eaten, they put the forks back and think.

Now, the problem that one has to deal with here is that, it is possible that one could end up with some kind of a deadlock situation. For example, if this philosopher grabs this fork and this philosopher grabs this fork and this philosopher grabs this fork and so, on (Refer Slide Time: 14:28).

They could be a situation where each philosopher has one fork and therefore, all of the philosopher's processes will be deadlocked, unable to eat because they cannot get two forks. Therefore, there is actually a serious need to make sure that this philosophers or the code which is use to describe the activity of anyone of these philosophers is carefully written to ensure that, this deadlock does not arise or each philosopher ends up with one fork.

So, the problem is basically one of avoiding deadlock. **So, these 12 problems could be looked at of solutions.** So, the solutions to these 12 problems could be looked at to understand the need for or the use of the synchronization of mutual exclusion primitives, and they are treated very nicely in many of the text books.

(Refer Slide Time: 15:21)

The slide is titled "THREADS" in green. Below the title, the word "Thread" is written in blue. A small red box contains a vertical wavy line representing a thread. The main content is a list of bullet points:

- Thread of control in a process
- 'Light weight process'
- Weight related to
 - Time for creation
 - Time for context switch
 - Size of context
- Recall: Process as a Data Structure

In the bottom right corner, there is a small video inset showing a man in a pink shirt speaking.

One additional concept which we were going to need later on and I will comment on that; what I mean by later on, in a few minutes is the concept of a thread. Therefore, I would like to talk about the thread before wrapping up on concurrent programming. Now, we have talked about processes and a thread is in some sense are related concept. We will think of a thread as being thread of control in a process, hence the name thread.

So, in general when I talked about a process, I talked about this worms could link to the process, which was a flow of control in along the same lines, we now talked about a thread of as being a thread of control within a process. Some people talk about a thread as being a special kind of process, but just lighter in weight and I will talk about that a little bit.

Now, when people talk about the weight of a thread as related to the weight of a process which is obviously, what is happening in referring to a thread as a process which has light weight, they must be referring to something about the thread which is simpler or less expensive or less time consuming, then the corresponding concept in the case of the process. Some aspects of threads, which may be relevant in this context, are that a thread is simpler than a process.

Therefore, text typical less time to create a thread as we are going to see is simpler from a process in important ways through which it takes much less time to switch from one

thread to another thread. Both of these are ramifications are a symptoms of the facts that, size of the context or the size of the amount of information data and operating system information or library information, associated with a thread is much smaller than that associated with the process.

So, now that to understand what a thread is, we need to understand the little bit about what we mean by the size of the thread or the context of a thread. Just to go back to our understanding of what a process is, let me remind you something that we saw in connection with our discussion of a process as a data structure. This will help us to understand what we mean by size of context.

(Refer Slide Time: 17:55)

Process as a Data Structure.

- What is the data manipulated by these process operations?
 1. Text, Data, Stack, Heap
 2. Data stored in hardware
 3. Other information maintained by the OS
 - Process parent and user identifiers
 - Memory management information: Page table
 - CPU time used by the process, in user/system
 - File related info: Open files, file pointers

Now, when I talked about the process as a data structure, we went through the exercise of listing the different operations that would be done on a process and those ultimately, what the different operations which were implemented a system calls. Subsequently, we have started to look at the different pieces of data, let for manipulated by those operations and we came up with a large collection of different pieces of data, that were be viewed as being important as far as the process is concerned. Therefore, the data associated with the process, there was a text data stack and heap which was fairly obvious.

There it contains of the hardware registers associated with the process, when it was running. Then there was a lot of operating system information which was associated with the process such as identifiers, memory management information such as page table etcetera, various species of information.

Now, all of this information is information that is associated with the process and could be described as the process context. In other words, the information that must be present or taken into account when that process is running; we saw that when there is a context switch from one process to another.

Some of the information that is listed over here had to be loaded into the hardware context. Some of the other pieces of information had to be taken into account by the operating system because, whenever there was a system call during the running of a process, the operating system had to be where which process is page table, which process is information about CPU time, which process is open files had to be refer to.

Therefore there was this notion that whenever there was a context switch not only the hardware state, but the operating system perspective had to be changed to that of the incoming process and the context of a process was fairly large, it filled one screen.

(Refer Slide Time: 15:21)

The slide is titled "THREADS" in green. Below the title is a blue heading "Thread" and a small red-outlined diagram of a thread. The main content is a list of bullet points:

- Thread of control in a process
- 'Light weight process'
- Weight related to
 - Time for creation
 - Time for context switch
 - Size of context
- Recall: Process as a Data Structure

In the bottom right corner, there is a small video inset showing a man in a pink shirt speaking.

Now, in saying that the context of or in saying that the time for context, which of a thread or the size of the context of a thread is going to be lighter than that of a process,

implication must be that, the thread is some kind of an entity a ((O)) to a process, but with much less information associated with it.

(Refer Slide Time: 19:45)

Process as a Data Structure:
PROCESS CONTEXT

- What is the data manipulated by these process operations?
 - Text, Data, Stack, Heap
 - Data stored in hardware
 - Other information maintained by the OS
 - Process, parent and user identifiers
 - Memory management information: Page table
 - CPU time used by the process, in user/system
 - File related info: Open files, file pointers

The slide features a red oval highlighting the list of data manipulated by process operations. A small inset image of a man in a pink shirt is visible in the bottom right corner.

(Refer Slide Time: 19:51)

Threads and Processes

- Thread context
 - Thread id
 - Stack
 - Stack pointer, PC, GPR values
- So, thread-context switching can be much faster than process context switch
- Many threads in the same process share parts of that process context
 - Virtual address space (other than stack)
- So, threads in the same process share variables that are not stack allocated

The slide includes a diagram on the right showing a circle labeled 'P' containing three smaller circles labeled 'T1', 'T2', and 'T3'. Red handwritten annotations include 'text', 'data', 'heap', and 'stack' with arrows pointing to the list items. A small inset image of a man in a pink shirt is visible in the bottom right corner.

Therefore, it should be possible for us to list for commonly occurring implementations of threads, what in the information might be contain within the thread context; it turns out that is this typically quite small, while it is obvious that each thread must have a unique identifier. Just like each process had to have a unique identifier, it is obvious because if there are multiple threads corresponding to the exclusion of a program, then it may be

necessary to switch from one to another and that therefore, they must clearly be distinguish from each other, for which reason each must have a unique identifier.

If they are separate threads of control, separate floors of control on a program, then they would clearly be able to make function calls independently of each other and must obviously, therefore, each have their own stack. Notice that, I do not say each must have their own text or each must have their own heap, they could actually have shared text data and heap but specifically, must have separate stacks because they are threads of control and may make separate function calls. Clearly, they must have separate stack pointers, if they have separate stacks and for that they must have separate special purpose and general purpose context associated with them.

If they are two threads and **they could** they are viewed as separate threads of control, exclusions inside the processor and I can switch from one to the other, very clearly the meaning of R 1 register 1 2, one of the threads is not going to always be the same as a value in R 1 from the perspective of one of other threads.

Similarly, as independent threads of control, flows of control they will be executing different instructions therefore, they must have separate hardware context of this kind, but that is pretty much it. So, the thread context is going to be very small, possibly separate stacks and stack pointer and a few of the general purpose registers, **I am sorry** the general purpose register file, as well as some of the special purpose registers.

But other than that, threads need not have separate context from each other. In fact, as a consequence of this, the amount of time that is going to take to switch from one thread to another is going to be much smaller than the amount of time that it takes to switch from one process to another. Flashily comes from the fact that, the process context which we listed on the previous slide is so large.

Now, the perspective that we could have is that, we could have a problem in that we to understand where the text data and heap associated with a thread are going to come from; it's well and good to say that, the thread context includes only these things, but if a thread is an active entity which could run on a processor, then it must have instructions associated with it.

I have mention the anything about its text or it is data or it is heap which is why the next point is we assuring, the picture we should have in mind is that, there could be many threads which are all part of the same process. In other words, they are all sharing many parts of the process context such as may be the text, the data and the heap.

So, the picture we may have in mind then is, I could have a single process in which there could be many threads and that is the common process context which includes text, data heap, open files etcetera, but each of the threads on its own may have a small amount of context of its own. And therefore, when I switch from one thread to another some of the process context between them could be shared.

(Refer Slide Time: 19:51)

Threads and Processes

- Thread context
 - Thread id
 - Stack
 - Stack pointer, PC, GPR values
- So, thread context switching can be much faster than process context switch
- ✎ Many threads in the same process share parts of that process context
 - Virtual address space (other than stack)
- So, threads in the same process share variables that are not stack allocated

In particular, other than the stack the virtual address space could be shared as a consequence of which the threads t 1, t 2, t 3 could actually have a large number of shared variables within the virtual memory frame work of any operating system.

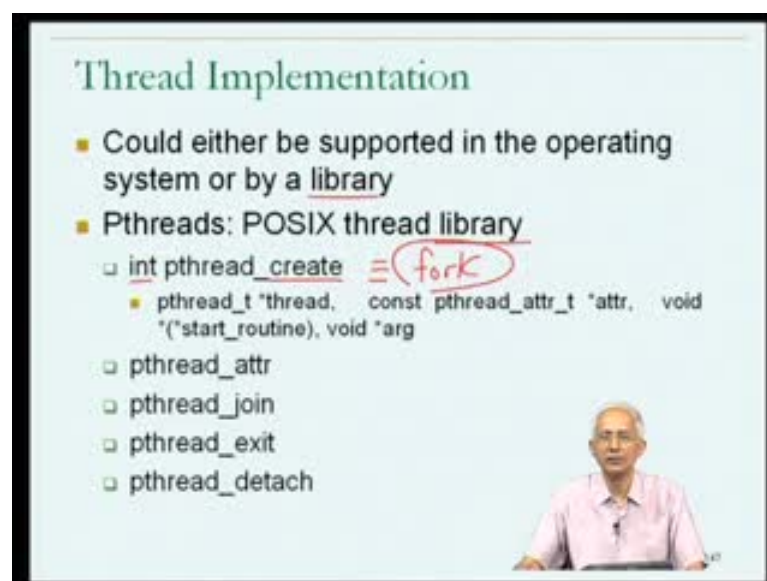
Therefore, a lot of what we talked about in connection with concurrent programming could actually find implementation in the form of threads rather than in implementation in the form of processes which are cooperating. Rather we could think about threads with in a process, cooperating towards the common objective, the threads would have to be synchronize, the threads would have to have our mutual exclusion on their access to shared variables.

But, those shared variables themselves could all be within a single process and therefore, not violating or understanding of the operating systems virtual memory in order to protect one process from another. This is a situation where the threads do not have to be protected by the operating system from each other. The threads being part of the same program would have been written to carefully and correctly utilize the shared variables in a manner that is good for their common objective. So, much about we have said, may make sense in the context of the one processor machines that we run on in the context of threads.

Now, we therefore have this notion, threads running in the same process having shared variables, but the stack allocated variables may not be need, may not be shared for reason of the stack being part of the separate context of the individual threads. In other words, each thread has its own stack and it is therefore, it is local variables and parameters are distinct from each other.

Now, the question which will arise now is, we talked about processes before we talked about operating systems. And subsequently, learn that the concept of processes created by the operating system, one of the operating system jobs is the management of processes. The operating systems schedule processes on to the CPU etcetera, what is the status of threads or threads also operating system level entities.

(Refer Slide Time: 25:12)



Thread Implementation

- Could either be supported in the operating system or by a library
- Pthreads: POSIX thread library
 - int pthread_create \equiv fork
 - pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine), void *arg
 - pthread_attr
 - pthread_join
 - pthread_exit
 - pthread_detach

(Note: The word 'fork' is circled in red in the original image.)

(A small inset image of a speaker is visible in the bottom right corner of the slide.)

Now, there are actually two possibilities. It is possible that the threads could be implemented or supported directly by the operating system, but on the other hand, it is possible to have implementation of threads which are made available through a library. In other words, not implemented inside the operating system but implemented outside the operating system through libraries of functions.

Some of the systems that you deal with will have hardware operating system support for threads and some of the others will not. In this case, you would use a threading library in order to deal with threads.

Now, one of the commonly available thread libraries is known as P threads; P threads stand for POSIX thread library and this is downloadable, available on many systems. What do you expect to find when you look at the thread library, what we mean understand by library is that it is a collection of functions and possibly associated data. What kind of functions would be expected to find in connection with the P threads or a threading library in general.

Now, very clearly we will expect that, they just like there is a system call to fork a new process, they must be some kind of a function inside the thread library to create a new thread. In the case of P threads, the name of that function is P thread underscore create, it returns an integer value, it takes various parameters; I would not go in to the details of this, but this is reassuring because this would be the parallel of fork, it is how one creates a new thread.

There are various other functions included in the library for doing various operations on threads, at the level of the process operations that we talked about. And I list some of them here, but will not going to the details. It becomes necessary to use P threads; one nearly has to understand the library.

Therefore, from our perspective you seen one function in P thread library which relates to fork, but we have seen that when we talked about writing concurrent programs using processes, there was the need for various synchronization in mutual exclusion primitive's came up. I suggested that the same requirements may exist if we are programming with threads, because threads do have shared variables, threads may have to synchronize with

each other. Therefore, locks and semaphores may be important for concurrent programming using threads.

(Refer Slide Time: 27:45)

Synchronization Primitives

Mutex locks *Acquire lock()*

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

If the mutex is already locked, the calling thread blocks until the mutex becomes available. Returns with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

```
pthread_mutex_unlock release lock()
```

Semaphores

```
sem_init /  
sem_wait P  
sem_post V
```

We would hope to see a several functions available in the thread library for operations relating to locks and semaphores. That is, in fact the case, we find out that there are functions for doing operations on locks. For example, there is the pthread mutex lock function and the P pthread mutex unlock, which serves the role of acquire lock and release lock in our earlier terminology. As far as semaphores are concerned, there is a mechanism to initialize a semaphore on many systems, a mechanism to do the P and the v operations on semaphores.

Whenever we see wait, we understand that would correspond to the P operation and post corresponds to the v operation - post of signal. Therefore, associated with these threading libraries, one does find support for the various issues that we saw in connection with concurrent programming.

(Refer Slide Time: 28:52)

Synchronization Primitives

Mutex locks *Acquire lock()*
`int pthread_mutex_lock(pthread_mutex_t *mutex)`
If the mutex is already locked, the calling thread blocks until the mutex becomes available. Returns with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

`pthread_mutex_unlock` *Release lock()*

Semaphores *Lock Semaphore* *Barrier*
`sem_init` */ P*
`sem_wait` */ P*
`sem_post` *\ V*

Now, I am going to stop the discussion of concurrent programming with this. You may be wondering why I talked about concurrent programming in some sort of a partial mode; for example, I talked about synchronization a little bit. I mention some other synchronization primitives, but did not go into them in detail at all. For example, I talked about lock and semaphore to some extent, but in earlier lecture, I had talked about something called the barrier. I never said anything to you about that, you will be wondering about this, you also be wondering about the relevance of this particular topic.

Now, let me just warn you about things which are ahead. Later in this course, we are going to actually talk about some high performance systems, remember the title of this course is high performance computing. Many of the high performance systems today there is more than one processor; up to now, our assumption has been that, we are talking about computer system in which there is only one processor.

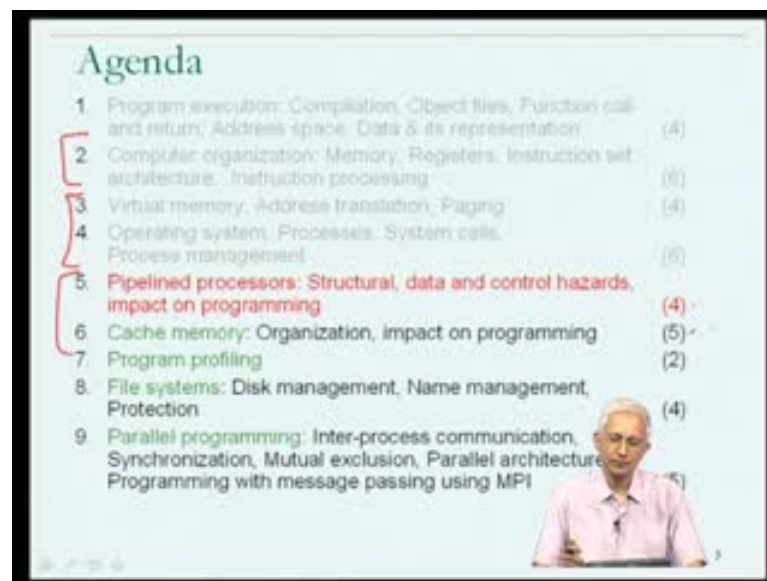
And hence only one process could be in execution at a time, only one thread could be in execution at a time, what I mean by that is only one process could be running on the processor at a time or only one thread could be running on the processor at a time, because there is only processor, only one program counter etcetera. But, if in high performance computing systems there is more than one processor, then there is a potential for a concurrent program to actually run with one process on one processor and one process on another processor. If that is the case, then it is no longer fair to call that a

concurrent program, because the activities on the two processors could actually be happening in parallel, in fact at the same time.

At exactly the instant in time, when process P 1 is running on processor 1, process P 2 might be running on processor 2. Therefore, there is no problem of concurrency there is actually a problem of events happening at the same time. This might in fact, we are most severe from the perspective of problem issues like, mutual exclusion and synchronization then, it was in the context of concurrent programming.

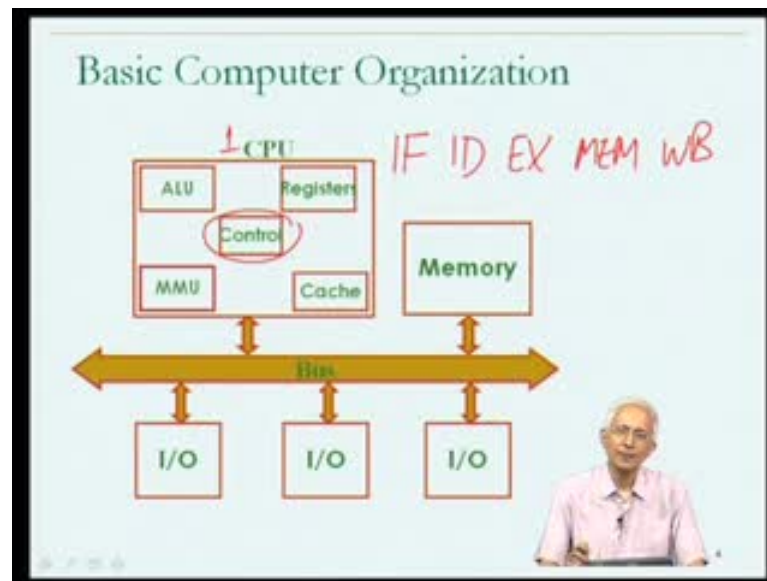
Therefore, later in this course, when we talk about high performance computer systems with multiple processors, we will come back to many of these issues in more detail, which is why I am rushing through them a little bit right now.

(Refer Slide Time: 30:50)



Now, coming back to our agenda, we are at a break point we have successfully completed our discussion of process management. About to move into the next topic, which is as you will see item number 5 label pipe line processors. Let me just switch to in that material, so we are now moving into the next topic which is line item number 5.

(Refer Slide Time: 31:50)



We have moved out of - if you looking back you will notice that, for the last 10 lectures we were on the software side of the computer systems prior to that, we were on the hardware side of computer systems briefly. We are now going back into the hardware side of computer systems for something like the next 9 lectures or so. You get to start off with this discussion of pipelined processors.

Again, let me just remind you, when we stop talking about hardware while back. We are still in the context of thinking about a computer system in which there is only one processor. Our discussion of the computer systems with more than one processor is a **little - several** lectures down the role. Our understanding of whether the processor operates is that, there is the need for the I O devices and the memory, but we looked a little bit at the control and had a picture in which we understood the various steps which had to happen in executing an instruction. There was a need to fetch the instruction from memory, there was a need to decode the instruction; in some instructions, they need to do a memory operation and in some instructions, they need to do a right back.

(Refer Slide Time: 32:47)

Performance of Processor

- Which is more important?
 - ☐ execution time of a single instruction
 - ☐ throughput of instruction execution
i.e., number of instructions executed per unit time
- Cycles Per Instruction (CPI) 2-4
- Current ideas: CPI between 3 and 5
- Pipelining CPI: 1
 - ☐ Why keep Fetch hardware idle while instruction is being decoded?
 - ☐ Inspired by petroleum pipelines?

We came up with simple hardware for each of the different steps in execution of an instruction. A question which you did not ask at that point in time was how fast can a processor be? You will recall that when we talked about implementation of a processor, we talked about how there was a general purpose register file, the correct operand may have to be fetched from memory; for that to happen conceivably the address may have to be calculated and so on, but we never talked about the performance of the processor, which is the title of the current slide.

Now, if we are to think about the processors that are available today, we assumed that they must be high performance processors and therefore, their hardware is not going to be along the lines of what we have seen.

It is going they are going to have some ideas within them to improve a speed with which they execute instructions. Now, an important question in this context which we need to understand to start things off is what is the correct priority? Which is more important? Is it important to build the hardware? So that, it can execute each instruction in as little time as possible. In other words, is it important to reduce the execution time of a single instruction or is it more important to increase the throughput of instruction execution? In other words, increase the number of instructions which are executed per unit time.

If you think about little bit, these two are distinct objectives, if it is my objective to reduce the execution time of a single instruction, I would actually try to optimize the piece of hardware for doing instruction fetch, instruction decode, etcetera. Then, put them all together to get the minimum possible amount of time for executing each instruction.

We saw that for some instruction, the amount of time to execute the instruction was 3 cycles, for other instructions the amount of time to execute the instruction was 4, for others it was 5. May be, where very careful design I could reduce this to 2 and 3 and 4, and that would be what would happen, if my objective was the first; first objective would try to reduce the times by very aggressive design possibly whereas, the objective in the second is saying do not worry too much about the individual instruction execution time, but try to improve the rate at which instructions execute, which is an interesting concept is not too clear at this point, how they could be done. But if you think about it little bit, let us assume that our objective is to make our programs run as fast as possible.

That is the ultimate bottom line; the manufacture of a processor knows that the processor will sell well, if it is able to execute programs faster than the competitors. Therefore, in some sense, whichever of these objectives will achieve that ultimate objective is probably the important one. Little bit of thought one can actually argue and convince people that the throughput is where one is going to get faster program execution, not necessarily from the faster in individual instruction execution time.

Therefore, we are actually going to start looking at an idea in which the objective is in fact, the second. **Trying to reduce I am sorry** trying to increase the rate at which instructions are executed, not paying too much attention to the amount of time that it takes to execute any one instruction.

In fact, they are not being too concerned if the amount of time to execute any one instruction is actually higher than it could have been, rather trying to increase the rate at which instructions are executed. Now, a commonly use terminology for the rate at which an instructions are executed is to talk about the number of cycles per instruction and that is sometimes abbreviated as CPI-Cycles Per Instruction. The general ideas that, if I have a program which is going to execute, I can count the total number of clock cycles it takes to execute the whole program.

(Refer Slide Time: 32:47)

Performance of Processor

- Which is more important? — 3 4 5
- execution time of a single instruction
- throughput of instruction execution
i.e., number of instructions executed per unit time
- Cycles Per Instruction (CPI) 2-4
- Current ideas: CPI between 3 and 5
- Pipelining CPI: 1
 - Why keep Fetch hardware idle while instruction is being decoded
 - Inspired by petroleum pipelines?

I can also count the number of instructions that it took to execute the whole program and then, I can divide the total number of cycles divided by the total number of instructions and come up with the cycles per instruction number for the execution of a particular program on a particular piece of hardware.

Now, for the kinds of design ideas that we have been looking at so far, you may ask what would the cycles per instruction for a program be. Arguing along the lines that I suggested at the top, you will remember that for using the ideas that we have so far, where some instructions take 3 cycles, some instructions take 4 cycles, some instructions take 5 cycles.

You will notice that, when I execute a program on a machine which has these characteristics, some of the instructions may take 5 cycles and some of the other may take 3, but no instruction is going to take less than 3 cycles, no instruction is going to take more than 5 cycles.

Therefore, however many instructions I execute - in other words, however many instructions have to be executed - as part of running a program that I am interested in, the CPI is going to end up somewhere between 3 and 5, it cannot be less than 3 it cannot be more than 5, it is going to be somewhere between 3 and 5 using the current ideas and a certain simplifying assumptions of course.

Now, we are going to look at an idea called pipelining, which is going to concentrate on the rate at which instructions are executed. It is not going to try to reduce the 3 to 2, reduce to 4 to 3, reduce to 5 to 4, the effect of doing that might have been to reduce the CPI to something like the range 2 to 4 and that is not very aggressive. The pipelining is going to try to reduce the CPI to something like 1, which is substantially better than the cycles per instruction that we were talking about for the ideas that we seen so far, so it is a substantially new approach.

Now, the basis of the idea of pipelining in essence from the context of the hardware that we have seen is to try to get an answer to this question. In building the hardware that we had, we had the hardware to fetch an instruction and increment the program counter, we had the hardware to decode an instruction and access the operands, then we had a hardware to execute the ALU etcetera, the hardware related to the access to memory if its load or store etcetera. Then, we just put these pieces of hardware one after the other and expected that, you would be use to execute instructions.

We never ask the question we still over here, the question is after the piece of hardware which was meant to fetch the instructions from memory had fetched the current instruction, why do I keep it idle when that particular instruction is being decoded?

Why cannot I use that fetch hardware for something else? This is the fundamental question which answers to which provide the idea of pipelining, but before getting into the mechanics of pipelining, I will just quickly suggest where the idea pipelining may have been motivated from.

Now, the suggestion from the name pipelining you might guess that the motivation for the idea of pipelining came from petroleum pipelines **sorry** in a quick - I am just going to hypothesis, how by this may be reasonable guess about where the inspiration for the word pipelining came from.

Let me just say talk a little bit about pipelines, I am not a chemical engineer, some of you may be and will have better ideas about pipelines then I have, therefore this is a just general rough kind of perspective on pipelines.

(Refer Slide Time: 32:47)

Performance of Processor

- Which is more important?
 - execution time of a single instruction
 - throughput of instruction execution
i.e., number of instructions executed per unit time
- Cycles Per Instruction (CPI) 2-4
- Current ideas: CPI between 3 and 5
- Pipelining CPI: 1
 - Why keep Fetch hardware idle while instruction is being decoded
 - Inspired by petroleum pipelines?

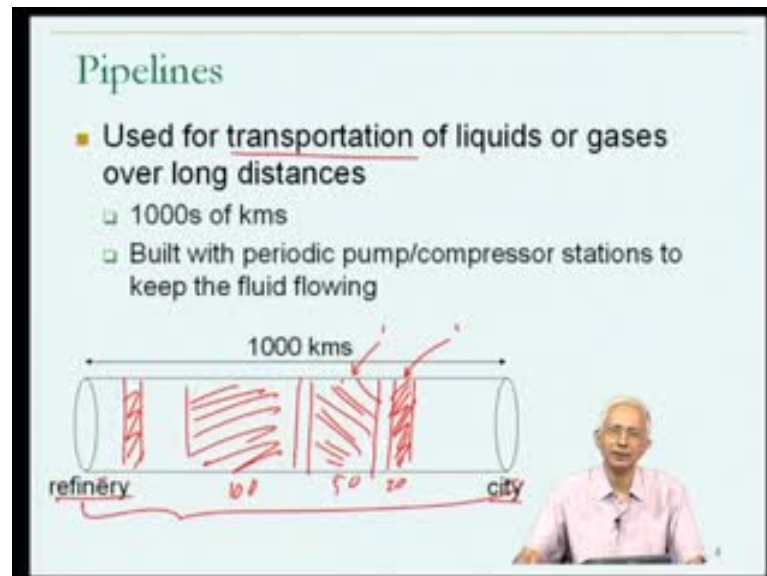
Now, pipelines are actually physically objects, they are used to transport fluids. The fluids could be liquids or gases over long distances and when I talk about long distances, I could be talking about 1000s of kilometers; for example, some of you may be aware about pipelines. For example, there is well known pipeline which stretches from somewhere in your Mumbai in India to somewhere near Delhi and that is 1000s of kilometers.

These are substantially large distances and they are used to transport fluids could be liquids or gases. This sometimes arises in the context of transporting petroleum products from a refinery to a point from which they could be distributed. What I mean by petroleum products I will elaborate on a little bit, but as far as a pipeline is concerned one should have a picture of some kind of a pipe, I am not sure the diameter these pipes will be.

In some case, the pipelines could be underground and on the cases, for cost reasons they may be over ground. What is important is that they must have some periodic stations, periodic pieces of equipment which could be used to pump or compress depending on whether we are talking about liquids or gases, because unless there is a pump to push the fluid through the pipeline, then the whole pipeline would have to have some kind of a gradient in order for the fluid to flow. The objective here is transportation of the fluid and therefore, they may be a need for these pumps periodically.

In addition to this, there may be the need for some protection mechanisms periodically such as valves, things like that periodically, but in a sense the abstraction of the pipeline that I will work with is one something like this. There is a pipe has been out running for potentially 1000s of kilometers and one example is, it might be running from a refinery to a city.

(Refer Slide Time: 42:02)



Now, a refinery is a place where crude petroleum is taken in and it is refined into many products. For example, it is cracked in one gets diesel, kerosene, airline fuel, etcetera; various different kinds of things will be obtained from crude petroleum and it is conceivable that all of them have to be transported to the city to the other end.

Now, one could actually think about, let suppose that one can describe the amount the volume of each of the different kinds of petroleum products that one has in terms of kilometers. Let suppose that at some point in time, I have the equivalent of 20 kilometers in terms of volume. The volume would be defined by the area of the pipe multiplied by the link that particular volume occupies, if it is pumped through the system. Let suppose that I have 20 kilometers of airline fuel, which I want to transport from the refinery to the city.

Then, one thing could be done is to just pump the 20 kilometers through the pipeline. Now, if that is the case then much of the effort of using the pipe line is going to be spent

in pumping air through the pipe line, because any given point in time there is 20 kilometers. Now, the 20 kilo meters of airline fuel here and it slowly moves to the pipeline as some point to time is over here and in pumping it, you are basically pumping air, in order to push the petroleum fraction that you have through the pipeline.

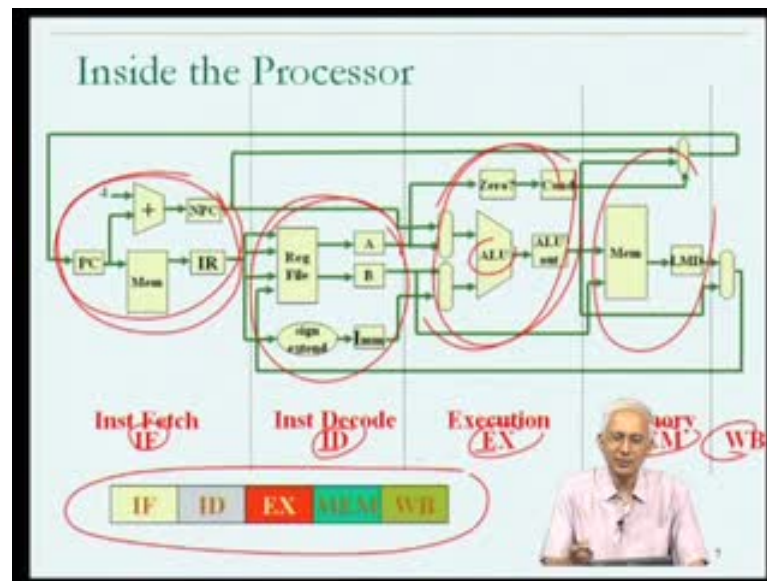
So this is not a very effective use of the pipe line, must more effective use of the pipeline might be that after having pump the 20 kilometers of airline fuel, I conceivably have to leave a little bit of a gap, but I could then pump may be 50 kilometers of some other fluid; may be, diesel through the same pipeline. After a little bit of a gap I could pump 100 kilometers of some other fraction of petroleum. In a sense by trying to keep the pipeline full, I will be increasing the utilization of the pipeline and in fact, I will be reducing the amount of time that it takes for anyone of these fractions to reach if I look at the average amount of time across all of the fractions.

In other words, here what we are trying to do is to improve the throughput of the pipeline. This is an idea for petroleum pipelines of improving the throughput and the key idea is rather than just concentrating on one fraction, I concentrate on many fractions; I try to get many fractions into the pipeline, so that pipeline is full of many different fractions at the same time.

Now, if you want to translate this into our processor context, I just have to replace the term petroleum by the term instructions and talk about, this has being one instruction, this has potentially being another instruction and so on (Refer Slide Time: 44:14). The general idea being that if I have a piece of hardware which is capable of executing instructions, then rather than trying to use all of the hardware for just one instruction at a time, in other words that 20 kilometers of airline fuel, I try to actually use some of the hardware for airline fuel, one of the instructions.

Some of the other hardware for kerosene, in other words, one of the other instructions and so on, in other words tries to use all of the hardware at any given point in time for various instructions, so that none of the hardware is idle. The net result will be that I might actually increase the throughput in other words, the rate at which instructions are completed.

(Refer Slide Time: 45:12)



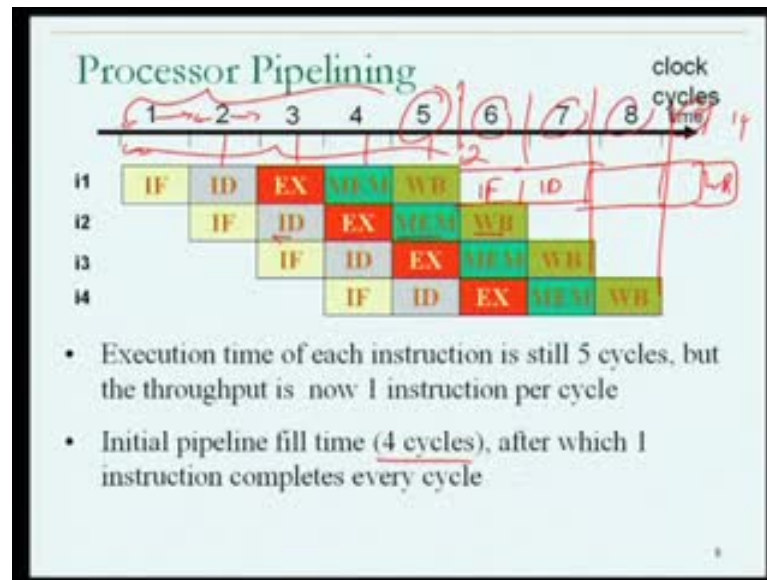
This seems like a reasonable guess as to where the motivation for pipelines came from. Now, if I look at the hardware that I had for the processor, this is what it look like, as I said there was the hardware to do fetching of instructions and incrementing the program counter, there was a hardware to decode instructions and fetch the operands, there was a hardware to do various ALU related operations whether they be in connection with arithmetic instructions or load store instructions or branch instructions.

In all these cases, the ALU may have **to you have** been involved and then there was a hardware to do operations on reading from memory such as load and store instructions. Up to now, our picture was that we can just use all of these for one instruction at a time. The time that we when through this description I had introduced a label for each of the five steps in instruction execution, the labels were IF, ID, EX, MEM and WB standing for Instruction Fetch, Instruction Decode, Instruction Execution, Memory Access and Write back.

Now, in the lectures to follow, I do not want to repeatedly reproduce this detail diagram; in fact, do not have to look at the details of the individual components of the hardware too often. Therefore, I will try to use some of kind in abbreviated notation which may in fact just looks something like this (Refer Slide Time: 46:30). Rather than drawing the instruction fetch hardware, I may just draw a yellow block labeled IF for instruction fetch.

Similarly, for instruction decodes a grey block labeled ID, instead of this instruction execute a red block labeled EX and so on. So, I will use this abbreviated form and when in doubt one in some situations, I may refer back to the full diagram, but typically the abbreviated form the surface.

(Refer Slide Time: 47:05)



The fundamental idea of pipe lining is that, we are interested in trying to maximize the rate at which instructions are executed, so we will keep steady track of time. Now, when I draw this time line, I just to make sure that there is no confusion about our recent use of time line - our recent use of time line involved either process virtual time or it involve real time. In this particular context, we are back to the hardware level and therefore we are talking about the hardware time line which is more accurate to the real time line, but in which we are actually concerned about the individual cycles that are going by as the activities proceed.

If I consider, I could therefore label that time line as clock cycles. In fact, since I know that the clock cycles discretize the time line, I could hence forth know the time line has being this discrete version, where there is the first clock cycle which is the time between here and here, this is followed by the second clock cycle and so on.

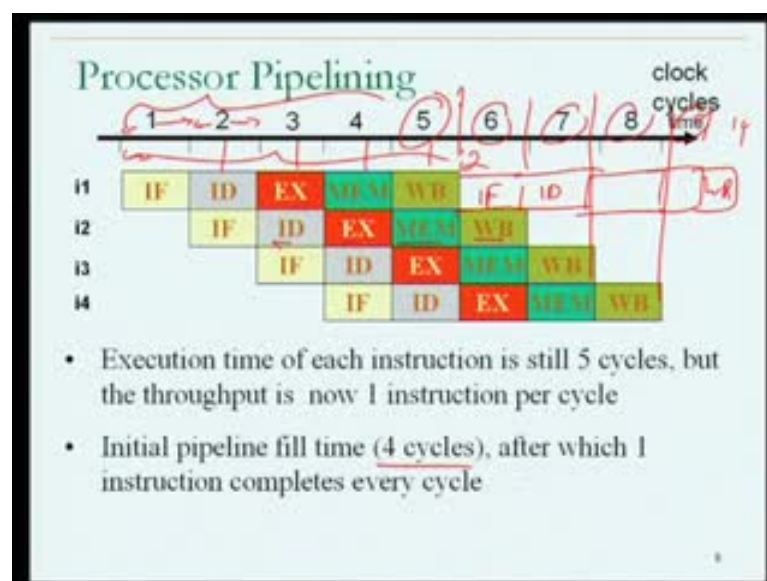
From now on, we will actually have a discretized version of the time line clock cycle 1, clock cycle 2, etcetera. In most cases, we will not be too concerned about the various

activities that happen within clock cycle 1 for example; in some cases, we may have to work now that important for the moment. What happens as instructions execute? Now, our current picture is that, the first instruction executes we will execute by spending 1 clock cycle being fetched, 1 clock cycle being decoded, 1 clock cycle being executed, etcetera. If it is an instruction which requires all the 5 aspects of the hardware, what about the second instruction execution?

Now, our current assumption is that the second instruction will start executing when the first instruction finishes execution. In other words, it will actually use the IF hardware in the sixth cycle, it will use the ID hardware in the second cycle, I am referring to the second instruction and so on.

Based on our current understanding, the second instruction would finish execution in the cycle numbered 10; it would be finish right back in the cycle number 10, but just based on the question that was just asked, the question which was asked in the slide before a last was, after the first instruction has used the IF or the Instruction Fetch hardware why do I keep it idle? Why instruction fetch hardware is kept idle for 4 cycles? Why cannot I use the instruction fetch hardware to fetch the second instruction right away? In other words, why not fetch the second instruction in cycle number 2, rather than waiting until cycle number 6 to fetch the second instruction and so on.

(Refer Slide Time: 47:05)



The second instruction can be fetched in cycle number 2, it can be decoded in cycle number 3, it can be executed in cycle number 4, it can use the MEM hardware in cycle number 5 and it can be written back in cycle number 6, rather than actually finishing on the cycle number 10. This is in a sense, what process of pipe lining will try to do. Note that, as far as process pipe lining is concerned, the fact that each instruction took 5 cycles is hardly concerned at all, because what we have achieved is more important. So, the third instruction can follow and use IF hardware starting from cycle number 3 and so on.

What is the net effect? The net effect is that just like in our previous simple perspective of the hardware, the amount of time that it takes to execute a single instruction is still 5 cycles, but with this new perspective on reusing the hardware as soon as possible, the rate at which instructions can complete is now 1 instruction per cycle, how do we see this? We see this if we look at the time that instruction 1 completes; instruction 1 completes in cycle 5, instruction 2 completes in cycle 6, instruction 3 completes in cycle 7 and so on.

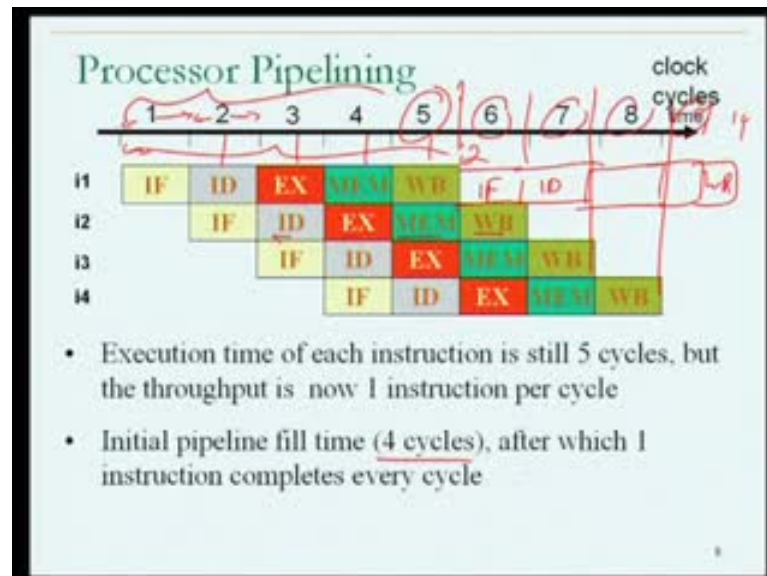
In other words, even though it takes 5 cycles for anyone instruction to execute, every cycle in instruction seems to be completing in this diagram. Therefore, the throughput or the rate at which instructions are execute is one instruction per cycle or one cycle per instruction using our CPI notation. Now, one thing I would want to point out from the perspective of the diagram, since the diagram is upon the screen is that, it is true that an instruction is completing every cycle starting with cycle number 5. In cycle number 5 instruction 1 completes in cycle number 6 instruction 2 completes and so on, but if I look at the first 4 cycles, no instruction completed. Therefore, I do have to know that there was this initial time of 4 cycles during which the pipeline was filling up when no instruction completed, but after that the throughput of this pipeline is 1 instruction per cycle.

Therefore, this simple idea if it can be implemented is very successfully increasing the rate at which instructions are completing and can therefore successfully reduce the amount of time that it takes to execute a program.

Now, we will look into the details of pipe lining and the problems that arise in the design of the hardware to achieve the objectives of pipe lining in the lectures that follow. I will stop today by just reminding you that we have now, in this lecture we have begin the

transition from software side of computer organization back to the hardware side of computer organization. We are now going to look in a little bit more detail at one of the more recent developments and computer hardware where pipe lining is used. This is in fact, a development which happened as more than 20 years ago, but the earlier ideas that we had seen predated even that.

(Refer Slide Time: 47:05)



So, this is an important stepping stone for us to understand the nature of processors today, because on the process of today we can definitely expect the pipe lining is a widely used hardware phenomenon and that is something which as programmers we may well have to be aware off, thank you.