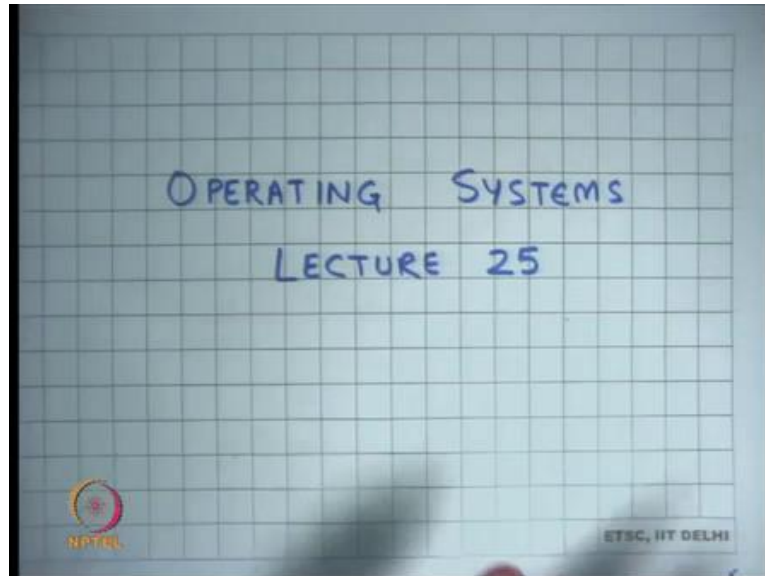


Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

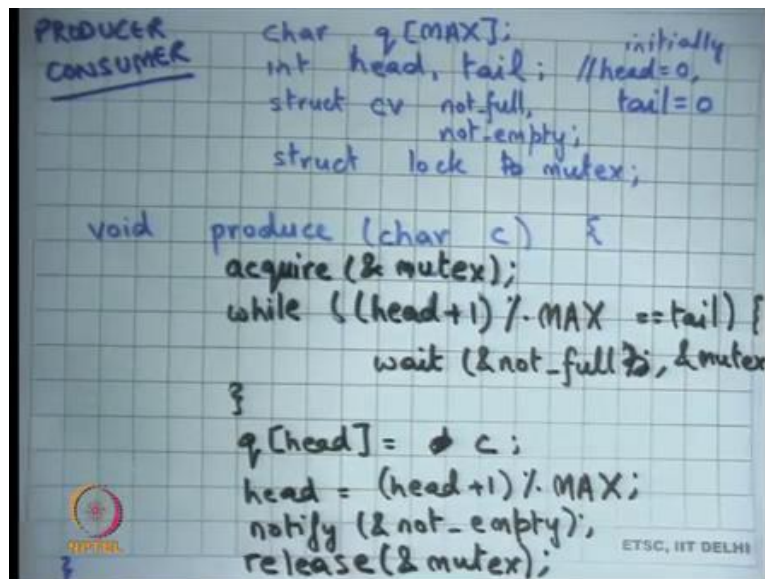
Lecture – 25
Multiple producers, multiple consumer queue; semaphores; monitors

(Refer Slide Time: 00:25)



Welcome to Operating Systems lecture 25, right.

(Refer Slide Time: 00:30)



So, last time, we were discussing a Producer Consumer example and we were you know the example that we had was there is a network thread and there is a server thread and there is an incoming queue of network packets and there could be multiple server threads for example, and similarly there is an outgoing queue and then, there could be multiple server threads and there is a there is a network threads that is picking packets from the outgoing network queue and putting them on the wire.

In any case this kind of a pattern where there is a producer and there is a consumer, so there are some producer threads and there is some consumer threads and there is a shared queue in the middle and the packets are being processed in the q in a 5 FIFO order; first in first out and the producer is supposed to produce elements and the consumer is supposed to consumer elements from right.

And we said that there are synchronization problems with that; of course, because number 1, the q itself is the shared structure. So, all accesses to the q and to the pointers in the q, have to be protected using locks because you know shared accesses to shared, I mean concurrent access has to shared data is dangerous. So, you should protect it with a lock. Not just that, you have to make sure that a producer does not produce to a full q and a consumer does not consume from an empty q right.

So, those are the two conditions that you have to satisfy. And so, not just not only do you have to maintain mutual exclusion, you have to also maintain these two invariants that you know that the producer should not produce to a full q and a consumer should not consume from an empty q, all right.

So, let me take go through this example in a little more detail. So, let us say this is my; this is my circular array that I am using to represent a q. It has MAX elements. I have 2 pointers: head and tail. Initially, I set head and tail to 0 and then, you know we looked at it last time and we said to ensure that a producer does not write to a full q, the producer should check for the condition and if the condition is not true, then it should wait on this condition variable called not full right.

And somebody will basically signal or notify this condition variables and so then, the producer can ensure you know then proceed. And similarly, the consumer should check if the q is empty and if it is empty, then it should wait on this condition variable called not-empty right and then, there should be a lock which basically does mutual exclusion

and today, I am going to I am calling this lock, I mean the name of this lock variable is mutex.

You could name it anything, where it is very common to name it mutex, to make it clear what the meaning of this lock is. This means this lock is basically for mutual exclusion and it is common; it is common practice to use the word mutex to represent a lock which is basically being used for mutual exclusion all right ok.

So, let us see how our producer works. The producer needs to first acquire the lock because it is going to operate on the read and write share data. So, it acquires a mutex and then, it checks whether the q is full and, in our case, you know because it is a circular buffer checking if the q is full involves checking if head plus 1 board MAX is 0 tail.

And notice because I have the lock, you know I can do this you know without having to worry about concurrent accesses, concurrent changes or reads or writes to head at all right. And if not, then I should wait on this condition variable called not full and the second argument to the to this wait is basically the mutex and the semantics of the wait are basically that, it will release the lock or release the mutex just before going to sleep in an atomic fashion and after it gets woken up, it the first thing it will do is try to reacquire the lock right.

So, basically the idea is that it will release the lock before sleeping and it will reacquire the lock, after it wakes up right and the condition that is waiting on or the condition variable, it is waiting on really is this is called not full right. Now, if it gets woken up by somebody. So, you know somebody calls notify or not full, what will happen?

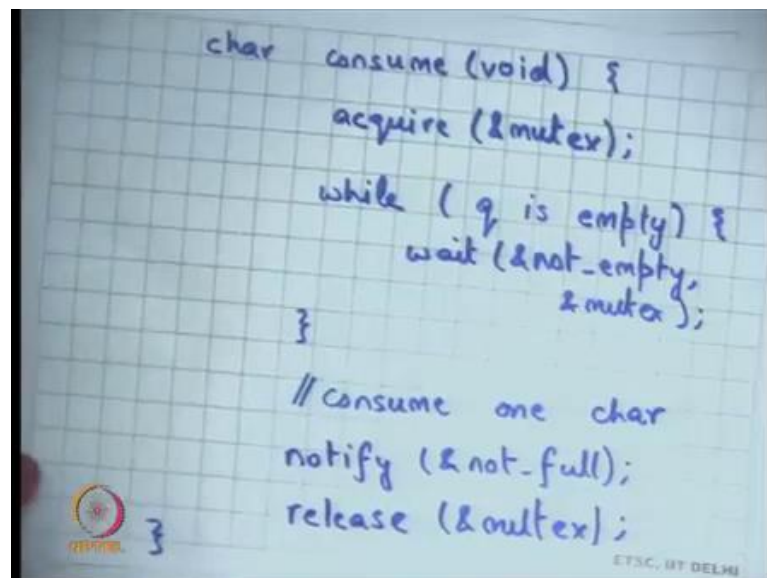
It will eventually, it will the next thing we will tried to do is try to reacquire the mutex and if it is able to reacquire the mutex, it will come out of the wait. In this code, the first thing you will do is you will check the condition again right. Why do I need to check the condition again? Let us hold that discussion for a moment.

So, let us say you know you check the condition again and you know assuming there was just 1 producer and 1 consumer and the consumer has just notified me, basically means that there must be some empty space in the q right; somebody has just consumed something and he has notified me.

So, there must be some empty space and so, this condition should be false right and so, you should get out of the loop and at this point, you can be sure that there is empty space in the q and head is pointing to that first empty slot and you produce in that empty slot right. After you produced in the empty slot, you may want to notify the consumer. In this case, I am actually notifying every time.

But you know you may want to be more efficient; you may want to say if you know if q has 1 element, only then notify right. Do not keep notifying unnecessarily, only if you are if you have just transitioned from 0 to 1 now or you are transitioned from an empty q to a non-empty q do you need to notify the consumer and consumer will have a very symmetric code right.

(Refer Slide Time: 05:52)



```
char consume(void) {  
    acquire(&mutex);  
    while (q is empty) {  
        wait(&not-empty,  
            &mutex);  
    }  
    // consume one char  
    notify(&not-full);  
    release(&mutex);  
}
```

The image shows a handwritten C code snippet for a consumer function. The code is written on a grid background. It starts with a function signature 'char consume(void) {'. Inside the function, it calls 'acquire(&mutex);'. Then there is a 'while (q is empty) {' loop. Inside the loop, it calls 'wait(¬-empty, &mutex);'. After the loop, there is a comment '// consume one char'. Then it calls 'notify(¬-full);' and 'release(&mutex);'. The function ends with a closing brace '}'.

So, if I just look at the consumer code, you know same thing you acquire the mutex, you check if the q is empty. This time I am not I am writing it in English instead of the actual condition right. I will wait on not-empty and mutex just like before at this if I come out of wait, I will recheck the condition and if I add just 1 producer and 1 consumer, this condition will always be false.

And so, I will come out of the loop and I have consumed 1 character, notify not-full release the mutex all right. So, now why do I need this loop? Right. Can I be sure that if the producer comes out of the wait? The q is definitely not-full all right. So, somebody, so, you said no, why?

Student: Because there are multiple producers.

So, yes. I mean if there are multiple producers, it is possible that even if you come out of the wait, the q is still full right. Why can it, how can it happen? Let us say you know there were 2 producers, both of them found the q to be full and both of them start waiting.

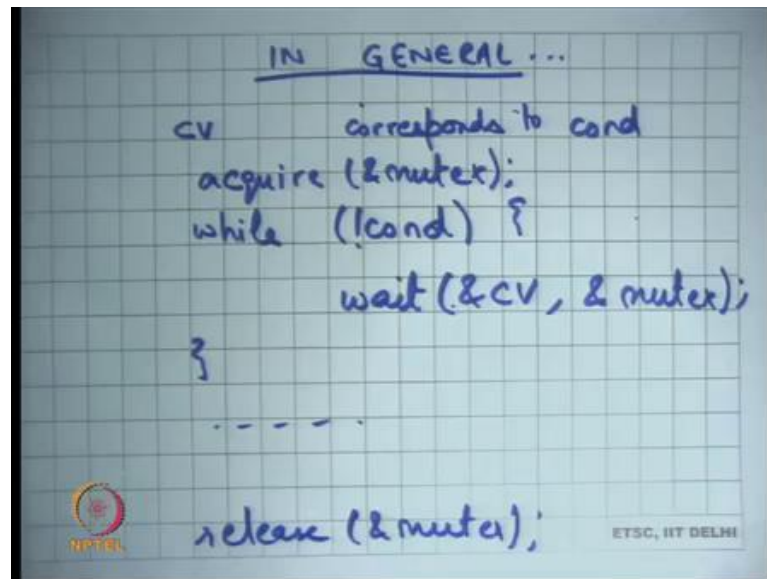
Now, one consumer comes along, and he consumes an element and then, he calls notify will have the effect of waking up both the producers let us say right. And so, both the producers will wake up; one of the producers will be able to get the mutex, the other producer will wait on the mutex. The producer who gets the mutex will you know check the condition; he will find it false; he can produce 1 element.

But at this point, he has made the q full. When he releases the mutex, the second producer will be able to get the mutex. He will come out of it and it is not ok for him to just go and start producing, it is he should again check the condition. So, the invariant is not that when you come out of it where the condition has definitely become false right. You went to wait because the condition was true, but it is not necessary that when you come out of wait, the condition is necessarily false.

So, hence you need to recheck the condition after you come out of wait right and so, you need a while loop instead of an if which we had last time. On the other hand, if you had only 1 producer and 1 consumer, if would have sufficed right, but it is always safer to you know use while I mean just to make sure, I mean then you are basically relying on more invariants in your code etcetera right.

So, if would have sufficed if you had just one. So, instead if you had to adjust 1 producer and 1 consumer, you could have replaced this if while within if ok, but if you have multiple producers, then you need a while. Similarly, if you have multiple consumers, then you need a while here right all right. So, this is a very common pattern. So, let us understand how a condition variable used in general right.

(Refer Slide Time: 08:39)



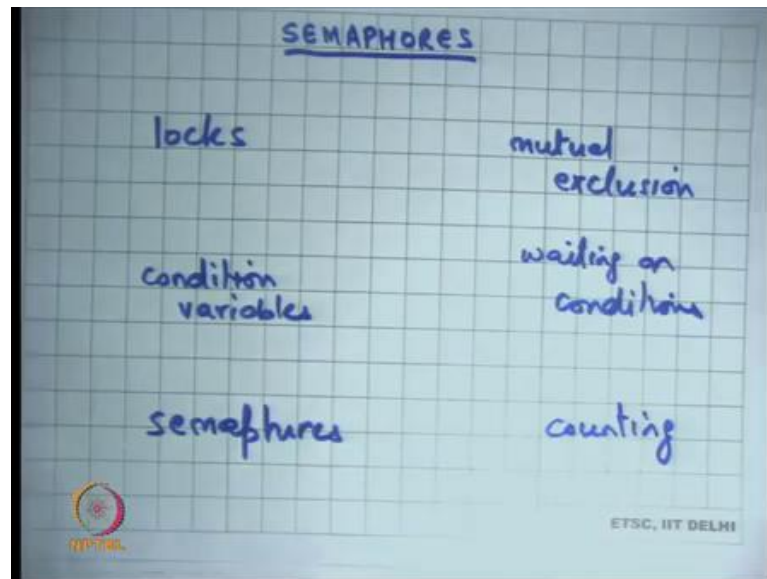
So, in general. So, I have taken an example of a producer consumer example to illustrate how condition variables are used, but in general a condition variable `cv` corresponds to some condition; let us say `cond` right. And you would typically want to check the condition, and you would typically want to check the condition in some kind of a loop.

So, while not `cond`, you will wait on `cv` all right. And because you know you want to make sure that the checking of the condition and the act of actually waiting should be atomic with respect to each other and also you want to make sure that there is mutual exclusion maintained in your checking of this condition.

You would typically want to also use a lock or a mutex right and that is the mutex you will pass as the second argument on this right. So, this will be the general pattern and of course, you will do something here and then, you will release the mutex right. And you may want to notify or not notify depending on whether you expect somebody else to be waiting on a condition or whether you are made some other condition true or not right.

So, this is a general pattern in which a condition variable is used. You take a mutex, you check a condition and if the condition is false, then you wait. But you do not, but you generally put the wait inside a while loop so that when you come out of it you recheck the condition right. Because the condition I mean may have become false by the time you actually came out of wait, just like in the producer consumer example right, so general pattern.

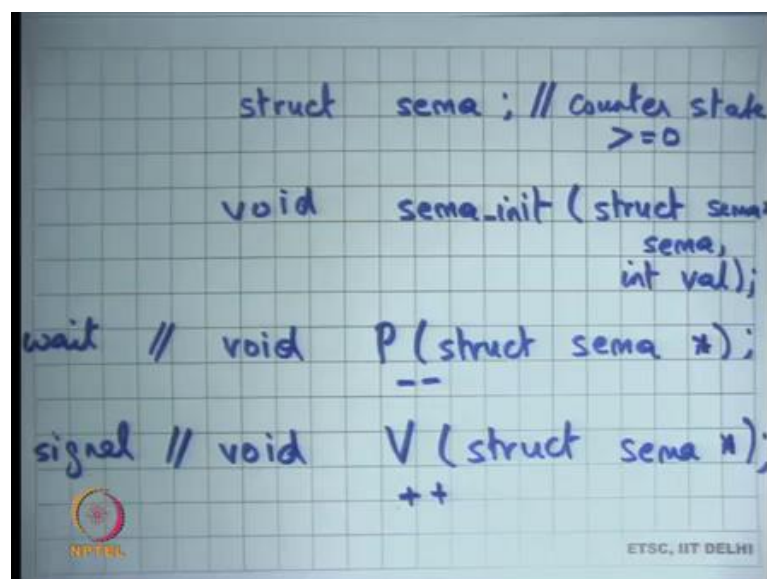
(Refer Slide Time: 10:30)



So, now I am going to talk about another abstraction which is Semaphores. So far, we have seen two abstractions to do synchronization; one is locks and locks were basically used for mutual exclusion, another was conditions, condition variables. These were basically associated with conditions. So, waiting on conditions all right.

Semaphores is yet another abstraction and they are basically you know the abstraction if you know more most concisely described as such something that enables you to count right and I am going to discuss exactly what semaphores mean right. So, we have looked at these two abstractions already. And today, I am discussing semaphores and let us see what semaphores are?

(Refer Slide Time: 11:30)



So, semaphores you know basically a common pattern in programming is basically like this programmer, producer consumer example involves counting of resources right. So, in the producer consumer example the resources were the slots in the q right. So, there were let us say MAX slots in the q and you make wanted to make sure that you know the you never go to $MAX + 1$. So, you never know; so, those are the number of sources and you are trying to count the number of resources.

So, the producer has to count the number of a full slots and similarly, the consumer has to count the number of empty slots and so on. And so, this is a very common pattern and in 1965 Dijkstra, whom you probably also know from a shortest path algorithm, you know proposed this subtraction called semaphores right. So, semaphores are basically in order to defined by a type, let us say it is a struct sema right.

Just like there were struct lock and struct cv, there is a type called struct sema and this is a state full type all right. Let us see and then there are 3 functions; there is void sema init that sema and int val. So, sema init takes the first argument as the semaphore variable sema and the second argument has an integer value which must be and basically it sets the counter inside sema to be equal to value.

So, it just initializes the semaphore to that counter called value right. So, basically semaphore has a state called counter right and the initial init variable will initialize the counter to value to val; assuming values greater than equal to 0. So, this counter should be greater than equal to 0 that is another invariant of a semaphore all right. Then, there is a function called P and there is a function called V.

The alphabets P and V are a you know are the first alphabets of the meanings of these functions in Dutch, you know which is basically Dijkstra's mother tongue, I guess. But you know the other names of this are wait sema wait or sema signal right, but we want to just call them P and V right. So, let us see what P and V mean? All right.

So, P's semantics are that it will decrement the counter of the semaphore by 1 right and V semantics are that it will increment the counter of the semaphore by 1 right. So, this is a minus-minus and operation and this is a plus-plus operation basically, all right. Except that minus-minus will only happen if the counter was greater than 0; if it was equal to 0, you will not you know so you will never allow the semaphore to become negative.

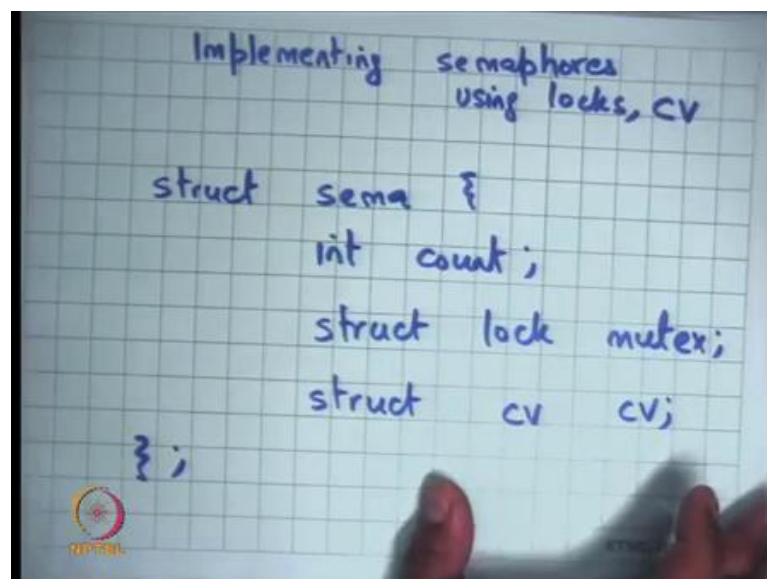
So, if it is equal to 0 and you call P on semaphore that is equal to 0, then you will wait and you will wait for it to become greater than 0 and only when it becomes greater than 0 value, do the decrement operation all right ok. Once again, init just initializes the counter to val. P decrements the counter except that it ensures that the semaphore never becomes negative.

If the semaphore is 0 and you know if somebody has called P that which means it may become negative, it will not actually decrement it, it will start waiting and we will start waiting for the semaphore to become greater than 0 and when it becomes greater than 0, at that point, it will decrement it and come out of P right.

And V is simply going to increment the semaphore. And of course, you know when it increment the semaphore, if it finds that somebody is waiting for it to become greater than 0, then it will wake it up and say you know why do not you know you can now decrement it is that right. So, those are the semantics.

Most importantly these operations, all these 3 operations are atomic with respect to each other right. So, this operation of decrementing and waiting is atomic with respect to increment and initialization and all you know. So, all these are atomic with respect to each other right. So, that is why this is these are these an interesting abstraction from a concurrency standpoint. So, P V and init are basically completely atomic with respect to each other, all right.

(Refer Slide Time: 16:25)

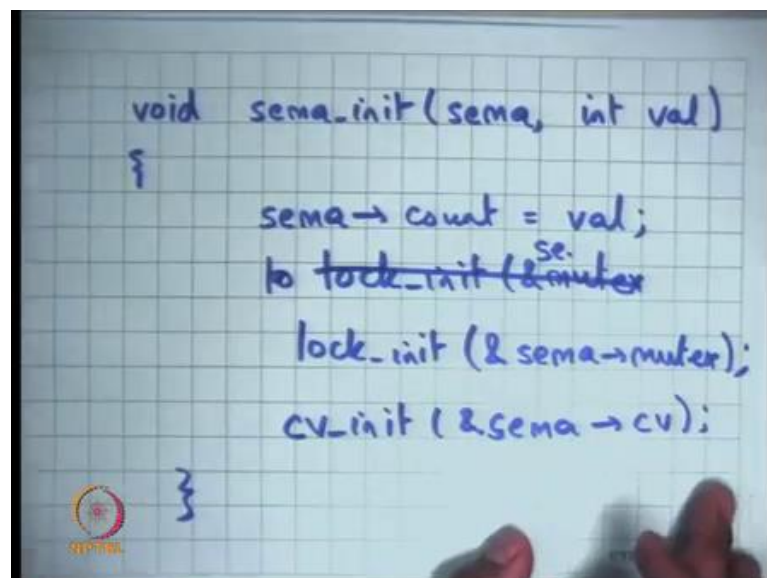


So, in order to make this clearer, I think it will help if we implement semaphores. you know the implementation of the semaphore will make it clear exactly what the semantics are the semaphores are in you know in concrete terms instead of just plain English and I am going to use locks and cvs to implement condition variables to implement semaphores right.

So, you know what I showed you in the previous slide was the semantics or the abstraction of a semaphores. Well, let us look at how a semaphore could be implemented, and I am going to give you one example implementation of a semaphore right. So, let us say struct sema will have a 1 field which is the count right. This is an integer field and let us say it has a an associated mutex lock right and an associated condition variable ok.

So, that is let us say that is how I define my semaphore and implement my semaphore, it has 3 fields account variable which is used to maintain this account and then, I have these two other fields that are basically there are to ensure atomicity and of my access with respect to each other and also in you know implementing semantics and along allowing waiting right on a condition.

(Refer Slide Time: 18:03)

A photograph of a hand-drawn code snippet on a grid background. The code defines a function 'sema_init' that takes a pointer to a semaphore structure and an integer value. Inside the function, it sets the 'count' field of the semaphore to the given value, then calls 'lock_init' with the address of the semaphore's 'mutex' field, and finally calls 'cv_init' with the address of the semaphore's 'cv' field. There is a small correction in the code where 'to lock_init' is crossed out and 'lock_init' is written instead, with a 'se.' above it.

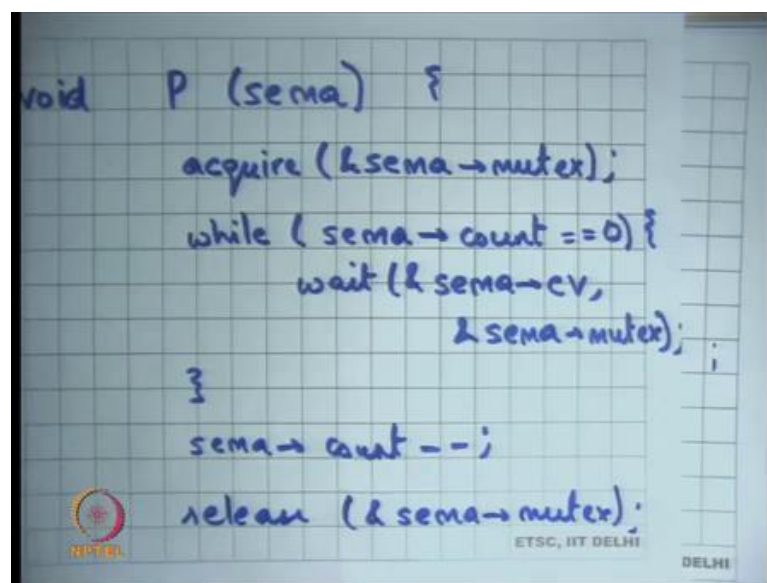
```
void sema_init(sema, int val)
{
    sema->count = val;
    to lock_initse. (&sema->mutex)
    lock_init (&sema->mutex);
    cv_init (&sema->cv);
}
```

So, let us say I do this and then, I need to implement sema init. There are two arguments sema and int val and what I am going to do is say sema.count is equal to val. But before I

and also going to say lock or lock init. So, let us say I you know there is a function which initializes a lock and let us say and initializes it to you know unlock state.

So, it just sema dot mutex all right. And let us say there is another function called cv init, I mean just initializing the corresponding fields right. So, that is the that is an initialization function just initialize all these all these repeats; nothing great I mean I just initialize the count, and these are just initialization functions other corresponding components right.

(Refer Slide Time: 19:09)



```
void P (sema) {  
    acquire (&sema->mutex);  
    while (sema->count == 0) {  
        wait (&sema->cv,  
              &sema->mutex);  
    }  
    sema->count --;  
    release (&sema->mutex);  
}
```

The image shows a handwritten C code snippet for the P (wait) operation of a semaphore. The code is written on a grid background. It starts with a function definition 'void P (sema) {'. The first line inside the function is 'acquire (&sema->mutex);'. This is followed by a 'while' loop: 'while (sema->count == 0) {'. Inside the loop, there is a 'wait' function call: 'wait (&sema->cv, &sema->mutex);'. The loop is closed with a closing brace '}'. After the loop, the count is decremented: 'sema->count --;'. Finally, the mutex is released: 'release (&sema->mutex);'. The function is closed with a closing brace '}'. In the bottom left corner, there is a logo for 'NPTEL'. In the bottom right corner, there is text that reads 'ETSC, IIT DELHI' and 'DELHI'.

Let us see P. So, I want to say P on sema, I could say I would say acquire. So, I want to make them atomic with respect to each other. So, I will say acquire sema.mutex right, that is ensured and nobody else could be operating on the semaphore at this time. So, all these operations should do acquire on sema.mutex and then, what should I check? I need to decrement the count right, but I also want to check that the count is greater than 0. So, what should I do? While; while what?

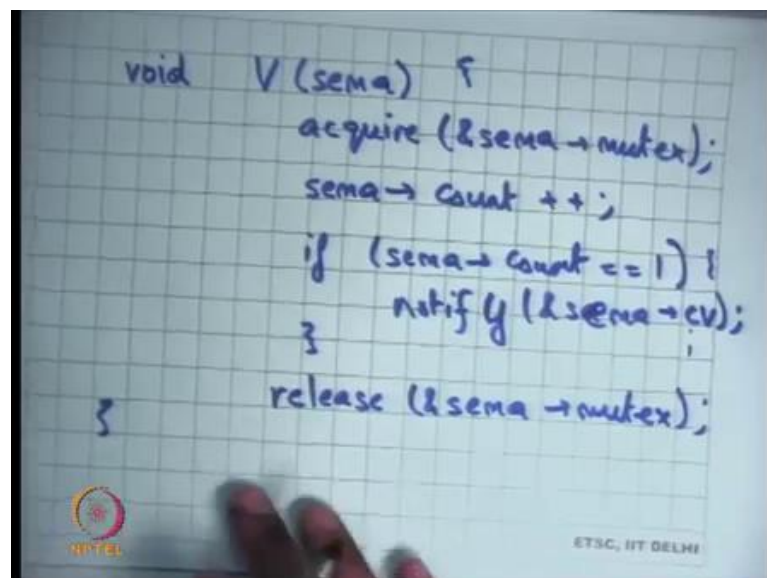
Student: Count is while count.

While count is equal to 0 right. So, I am using the condition variable to basically wait on the condition that discount becomes greater than 0 right. So, while count is not greater than 0 or while count is equal to 0 other in other words, I would say while (sema.count == 0) wait on sema.cv sema.mutex right ok.

And then, if I have if I come out of the loop, I can be sure that `sema.count > 0` right and at this point, I can just say `sema.count--` all right and then release.

Make sense, basically I wanted to do this, but I also wanted to do this only if this condition is false right and so, till this condition is true, I need to wait and I am using a condition variable to wait and I will and as you can imagine what I will do is in the V function, I will call notify on this condition wait right and of course, I use the mutex, I release the mutex before I go to wait and I do this in an atomic way just like all the other pattern we have seen so far, all right ok.

(Refer Slide Time: 21:36)



```
void V(sema) {
    acquire(&sema->mutex);
    sema->count++;
    if (sema->count == 1) {
        notify(&sema->cv);
    }
    release(&sema->mutex);
}
```

The image shows a handwritten code snippet for the V function on a grid background. The code is written in blue ink. It starts with a function signature `void V(sema) {`. The first line inside the function is `acquire(&sema->mutex);`. The second line is `sema->count++;`. The third line is an if statement: `if (sema->count == 1) {`. Inside the if block, the fourth line is `notify(&sema->cv);`. The if block is closed with a closing brace `}`. The final line of the function is `release(&sema->mutex);`, followed by a closing brace `}` for the function. In the bottom left corner, there is a small logo with the word 'NPTEL' and a circular emblem. In the bottom right corner, the text 'ETSC, IIT DELHI' is visible.

Now, let us write V.

Student: Sir.

Yes.

Student: Sir when we call (Refer Time: 21:40) pass the mutex is the argument.

Ok.

Student: So, the wave function that release the mutex on it is own, so then, why do we need to release it again? As in general, we happened on this but.

No. So, the wait function releases the mutex, but also reacquires it after waking up. It does not just release; it also reacquires it ok. So, at this point or any you know anytime you are outside wait, you would hold the mutex in this code right. Similarly, there is V on sema. What do I do here? First, I acquire the lock right.

So, I acquire sema.mutex and I need to do that to maintain mutual exclusion between P and V and between multiple V's and multiple you know. So, mutex is basically ensuring mutual exclusion between a P and a V or between multiple P's or between multiple V's basically right. So, it just making sure that things are completely mutually exclusive all right, and what do I write here?

Student: Increment.

I just increment right, I do not need to wait on any condition, I can just you know V the semantics of V as I have told you is just to increment the semaphores count. I just say sema.count++. Do I need to do anything else?

Student: Notify (Refer Time: 23:18).

Notify and one thing is I could always notify, or a more efficient thing could be to check if the count is equal to 1, only then notify right. If it had just become 1 from 0, only then there is a likelihood that anybody is waiting right. So, I basically say if sema.count = 1 notify sema.cv right ok. So, that so this implementation obeys the semantics of semaphores and I am using lots and condition variables to implement semaphores.

But of course, you know you could, you know you could, you may not, you may just want to implement semaphores using assembly instructions, like you may want to implement semaphores using the atomic exchange instruction, that we have seen right. That may be a more efficient way of implementing semaphores and I would like you to think about it at home right. How will you implement semaphores using just assembly, instead of using these high-level abstractions of locks and condition variables?

Definitely possible, after all these locks themselves are implemented using assembly instructions right and so, our condition variables. So, let us look at abstraction once again. So, we have a type called sema. It is a stateful type, in the sense that it has a state

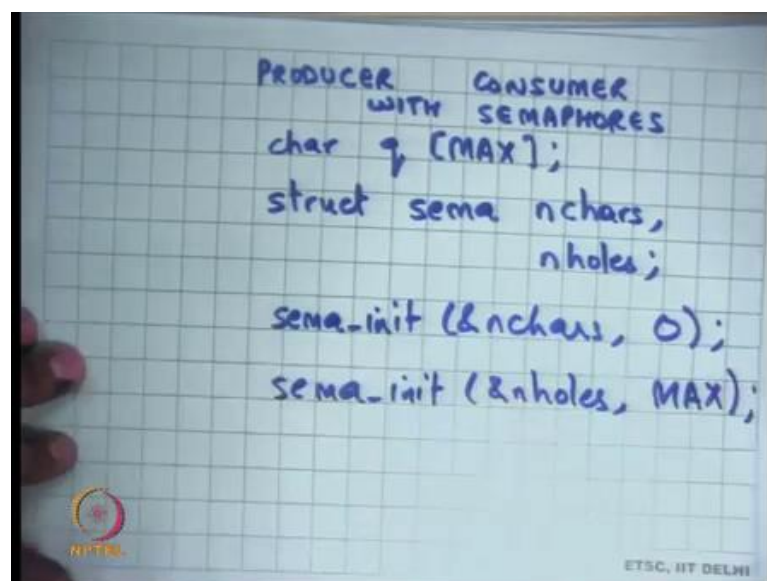
and the state is the counter right and the counter must be greater than equal to 0, non-negative.

And then, these are the 3 functions that you can use on the sema and this is a contrast to, so in many ways semaphores and condition variables look a little similar. Because condition variables also had a wait function and a notify function and one may be tempted to say or P looks very similar to wait and V looks very similar to notify right. After all, P is just saying let us wait on something and V is just saying let us signal something; except that we have now the difference between a semaphore and a condition variable as that semaphore also has state.

A condition variable did not have any state. If you call notify if somebody is waiting right now, he will wake up; if nobody is waiting right now, nothing will happen. If somebody goes to wait later, you know it says problem you somebody else you call him call to call notify from him. So, if a notify occurred before wait no they do not match up. On the other hand, with the semaphore if a V occurs before P, then it is because V were have incremented and so, P will not have to wait right.

So, in that sense, they do not there is because of the state they do not need to happen together, they can happen at different points of time. So, you know that this may become clearer, when we are even I talk about some example uses of a semaphores all right.

(Refer Slide Time: 26:15)

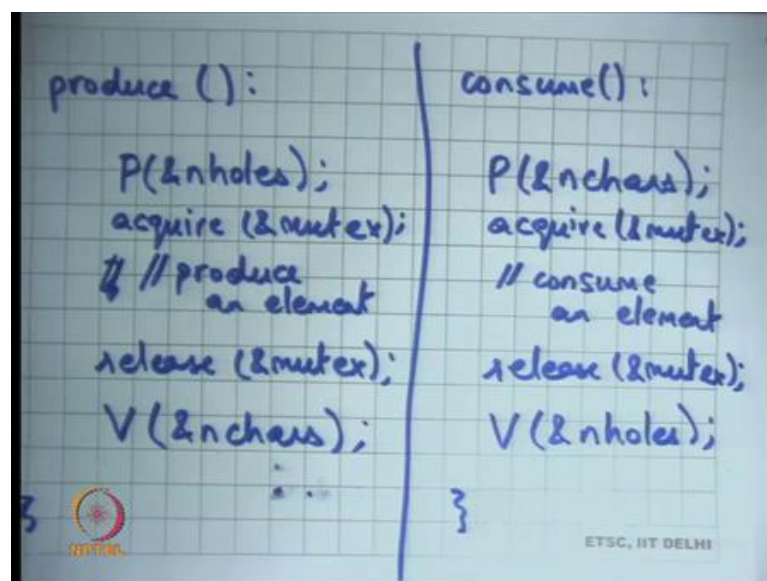


So, let us look at Producer Consumer with Semaphores. So, we have seen producer consumer with the locks and condition variables, but let us see producing consumers with semaphores and let us say we again have `char q[MAX]` and let us say I also have head and tail, but I am going to just you know abstract them and let us say now, I am as I basically I need to what I am going to do is I need to count the resources right, I need to make sure that a producer does not produce to a full q and a consumer does not consume from an empty q.

So, what I will do is I will maintain two counters; one for all for the number of elements that are present in the q and one for the number of elements that are or number of slots that are available in the q all right. So, I will have two counters and each counter will be of type sema; so struck sema, one is you know how many elements there are in the q? So, let us see n chars that is the how many elements there are in the q.

So, it is just a counter right structure, but I am using it semaphores to represent that counter and then, there is another counter called n holes which is how many empty slots there are in the q and the invariant will be typically that n holes plus n chars will be equal to MAX basically right. And so, I initialize n chars to 0, let us say the q was initially empty and I initialize n holes to MAX right.

(Refer Slide Time: 28:07)



```
produce():  
    P(&nholes);  
    acquire(&mutex);  
    // produce  
    // an element  
    release(&mutex);  
    V(&nchars);  
    ...  
consume():  
    P(&nchars);  
    acquire(&mutex);  
    // consume  
    // an element  
    release(&mutex);  
    V(&nholes);  
    ...
```

ETSC, IIT DELHI

Now, let us see how you write a producer. Let us say this is produce and let us say this is consume. All right ok. So, recall that in the previous case what I did was I acquired a

mutex, then I checked for a condition inside a while loop and then I waited and then, when I came out of the loop, then I consume etcetera.

But when I use semaphores, I do not have to do any of that; all I need to do is if I want to produce, I call P on n holes right. It basically means I am going I need you know decrement n holes by 1 an atomic manner, if I am and I because n holes are non-negative number, if I come out of P has to be non-negative number, when if I come out of P, I am sure that there is an empty slot available and the q right. And then, I produce an element.

So, let us say you know produce an element which may mean you know $q[\text{head}] = c$, $\text{head} = (\text{head} + 1) \% \text{MAX}$ and then, and let us say that is it right. And then in consume, I do P n chars I decrement the number of characters and I consume them consume an element. So, both of them are just decrementing; somebody also needs to increment it right. So, where do you increment?

Student: After (Refer Time: 29:49).

Right. So, I will increment here. What will our increment?

Student: N chars.

N chars right. So, you decrement n holes before you increment n chars after because you know at this point you have added a character and you can basically say you know you increment n chars by 1 and similarly, you can increment n holes here. That's it right? Semaphores go ahead.

Student: Do not we need mutex?

Do not we need mutex. Great question. So, that is it for making sure that you do not produce to a full q and it is also making sure that you do not produce from an empty q, but it does not make sure that this these operations produce an element and consume an element are atomic with respect to each other and so, you may want to have another. So, you will want to have another thing we let us say you have an acquire mutex here just for the mutual exclusion part and then, you have a release mutex.

Student: Sir, should not acquire (Refer Time: 30:50) as an outside P and V?

All right let us see. So, the question is should not we acquire before P and release after V; what do you think? So, firstly, you know we said that P itself is atomic. So, it does not need to be provide protected by a mutex right and the. So, you know right this code is correct firstly, all right you need to convince yourself that this code is correct.

You have decremented n holes atomically and at this point you can be sure that you are free to consume or free to produce here right. Similarly, your decremented n chars you come out, so you are going to be sure that you are free to consume right. Mutex is just making sure that they are mutually exclusive with each other.

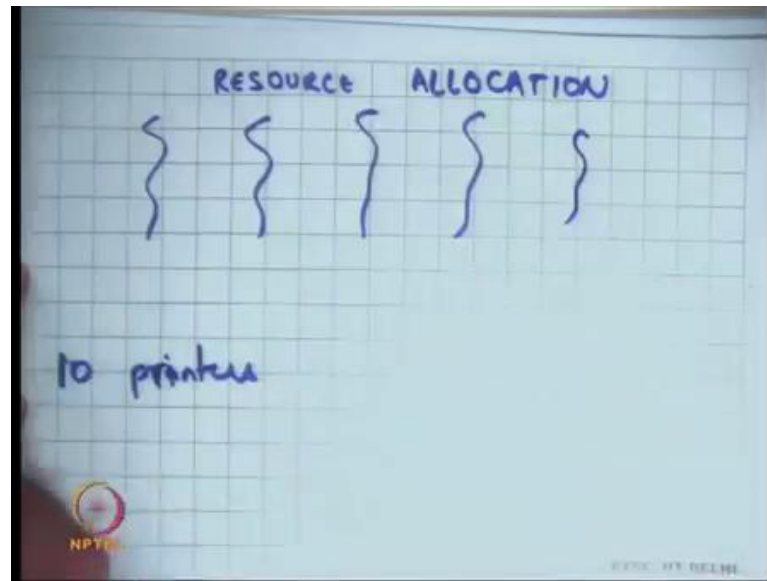
If you put acquire before P, you know you may actually run into problems because you know you here is the situation where you know that two things, there are two resources there is a mutex and there are n holes and you are going to, you want to acquire both of them at once and now, you have to worry about order and all that and so, you know you have to worry about deadlocks etcetera.

But if you do code in this way, then you know there is absolutely no problem of deadlock. Now, acquire and release a sort of mutex are within the inner loop and so, it can never happen that you wait on a mutex and then, you wait on something else. You are hold a mutex and you wait on something else that is not possible and so, you will never have a deadlock in this case right.

And of course, you also want to make your critical section as small as possible. So, that is you know that is producer consumer with some work with semaphores. This code is much simpler than the code that we had seen earlier right. What is going on? Well, what is going on as simple that you know we have noticed that there is a pattern, the pattern is that of counting; both of in both cases, we were counting, we were just counting the number of elements and we were waiting and so, we have subsumed that logic of counting within the P function itself right that is all we have done.

And so the our higher level code looks much simpler right and so, semaphores are a very useful abstraction, it allows us to capture this very common paradigm where you are counting the sources using P and V and your code actually looks much simpler and much better to understand, much easier to understand in that sense all right. So, this is an example of producer consumer with semaphores.

(Refer Slide Time: 33:24)

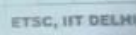



Let me give you some how another example. Let us say Resource Allocation all right. So, let us say you know I have 10 printers and there are you know many threads let us say 100s of threads that are trying to print on these 10 printers and I can I say that because there are only 10 printers, there should be utmost 10 concurrent requests in my printer, in my print q at any at any time let us say right. So, this is a problem of resource allocation.

You have multiple threads, you have hundreds of threads and a, but you have a few number of resources and you want to say that you know I want to give only, you know I want to make sure that the concurrency factor is limited to by the number of resources. So, let us say the resource is a printer, then utmost 10 concurrent print requests can be given, others should have to wait right.

(Refer Slide Time: 34:22)

```
sema_int (&nprinters, 10);  
void print () {  
    P(&nprinters)  
  
    }  
  
    V(&nprinters);  
}
```

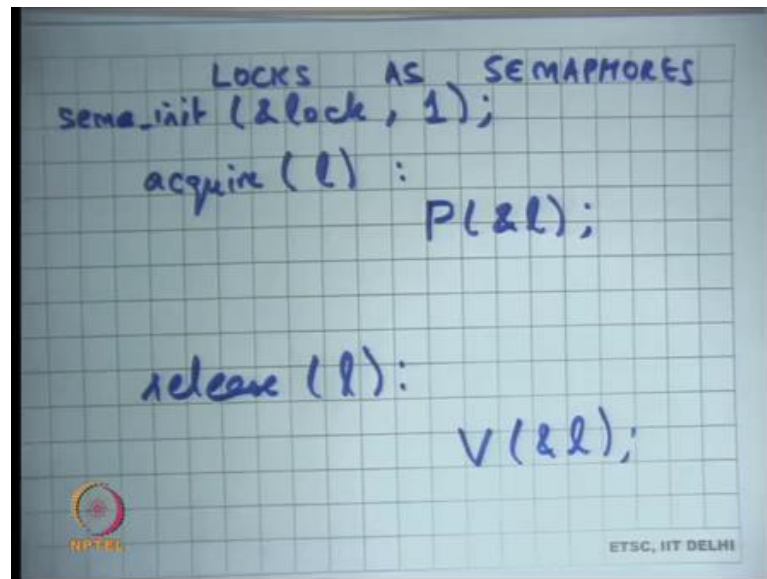


So, very easily handled by semaphores, all you need to do is let us say this is your print function, inside the print function you will basically say P on let us say n printers, you know call the print command and then, call V on n printers. Of course, you will first initialize the int printers variable semaphore to 10.

Let us say if you know you had only 10 printers or whatever is the number of printers you initialize the counter to the number of resources and then, whoever wants to consume the resource should or access the resource should enclose it is access logic within a P and a V and that is it right.

So, if there are a 100 threads depending on who came first, some of them will get it and as the threads are done now as threads exit they will immediately call V, which will cause another thread to enter right. So, it will allow you to do scheduling across and resource allocation across multiple threads all right. The semaphores for resource allocation all right.

(Refer Slide Time: 35:36)



And let us see another sort of interesting example is Locks, can be implement it as semaphore right simply. So, what is a lock attraction? You basically have a state right. So, lock unlike condition variables are stateful is a stateful abstraction right. The condition variables are a stateless abstraction, semaphores are stateful abstraction and lock, it is also a stateful abstraction because the lock variable has the state called whether it is locked or not right.

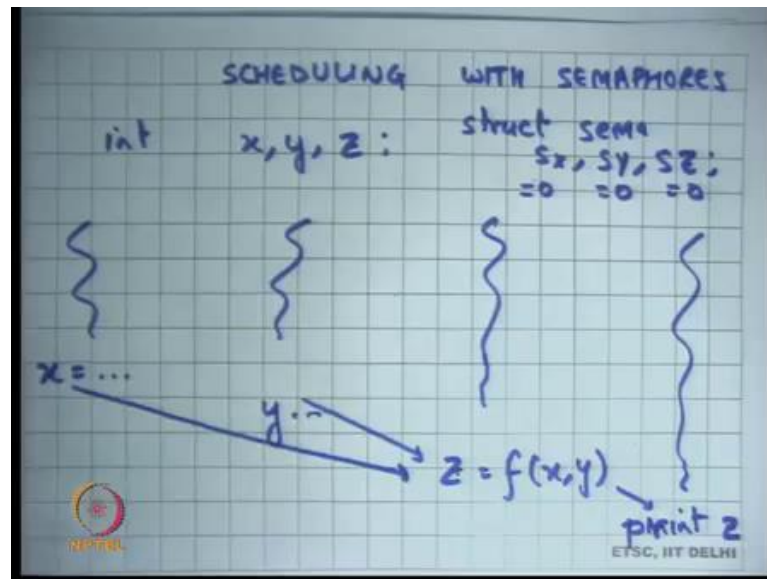
The lock variable has 1 bit of state the semaphore has an integer worth of state right ok. So, locks as semaphores. Well, what is a lock? It has acquire l; well, you can implement acquire l by you know putting use having a semaphore inside l and you can just say you know you can you can say initial you can initialize the semaphore as 1 right.

So, a lock has done nothing but a resource with 1 instance and so, if the multiple threads we want to access this resource only 1 instance can go in it right. So, a lock can be model as a resource and you can use semaphores to do it. So, acquire l is nothing but what?

Student: P.

P. That is it and release l is V l. Simplistic use of a semaphore you know semaphore who can do much more, but if you wanted to use the semaphore as a lock very easy we initialize it to 1, acquired becomes P and release becomes V.

(Refer Slide Time: 37:18)



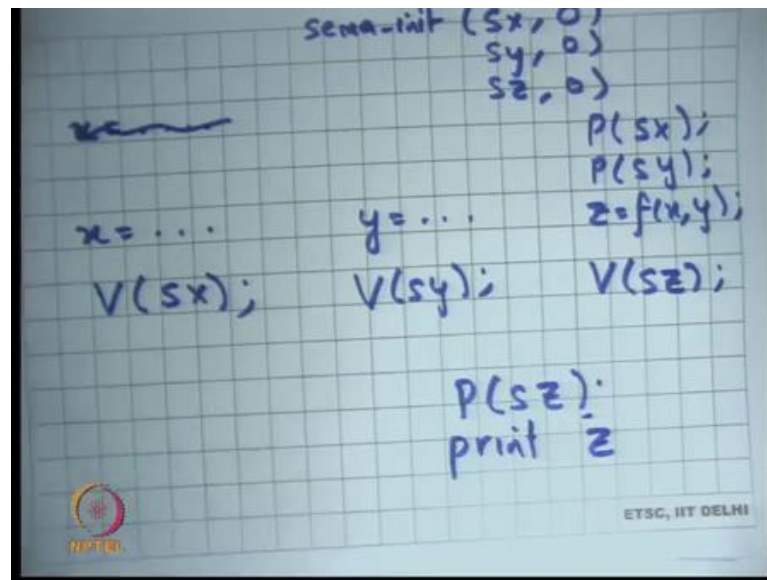
Let us see Scheduling with Semaphores. So, let us say; let us say there are you know these variables called x, y, z right. Let us see these are integers and somebody is going to compute x , somebody else is going to compute y and then, a third thread is going to use x and y to compute z and then, once z is computed you want to print the value of z right.

So, let us say there are you know 3 threads; one of them compute x , then another thread computes y , then yet another thread compute z as a function of x and y all right and let us say yet another thread, you know prints z . But you want to make sure that this computation happens only after this and this has happened, and you want to make sure that this print happens only after this has happened right. So, you want some kind of dependency.

So, let us say these are the dependency you want that you know z should happen after this has happened and this has happened, and this should happen after this has happened. So, this kind of some kind of dependency graph that you have and you are doing computation and different threads are going to compute this thing and you want to schedule these threads or you want to synchronize these threads in this way.

One way to do this is using semaphores. So, you can say struct sema sx sy and sz , you associate each us; 1 semaphore with each variable denoting whether that variable has been computed or not all right. So, sx and you initialize it to 0. So, all these semaphores I initialized to 0. So, you use you call the sema init function to initialize them to 0 and then, you write codes something like this.

(Refer Slide Time: 39:18)



So, when you say $x = \dots$ you say, so initially all of them are 0. So, `sema_init(sx, 0)`, `(sy, 0)` and `(sz, 0)` right. Then, you say $x = \dots$ and then you say what?

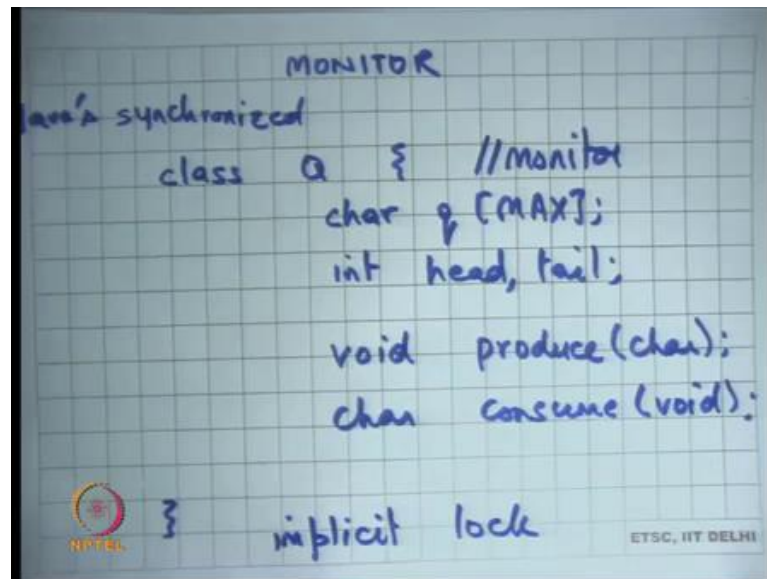
Student: (Refer Time: 39:40).

You say `V(sx)` right and you say $y = \dots$ you say `V(sy)`. Basically, indicating that x has been computed. So, you put up your push up the flag by 1. Y has been computed, so you push a y 's flag by 1 and for $z = f(x, y)$ what do you do? Before it you say `P(sx)` and `P(sy)`. So, `P` is going to have the effect of waiting for somebody to have called `V`.

So, only if somebody has called `V` would I ever be able to come out of `P` right because I initialize it to 0. Means so, this is an example of how you will do scheduling and then, after your computed z you may say `P(sz)` `V(sz)` and of course, in you know before the print z you may want to say `P(sz)` right.

So, if you initialize the semaphore to 0 and use it to represent whether something has been computed or not or some condition is true or not, you can use that to make sure that things are happening one after another in a you know in an ordered way in a scheduled way, good.

(Refer Slide Time: 40:54)



Let me talk about another abstraction. So, we have seen 3 abstraction so far locks, conditioned variables and semaphores right and you have seen how you know locks are absolutely integral to do mutual exclusion and notice that these abstractions actually can be implemented, they are all actually roughly equally powerful abstractions. So, you can implement semaphores using locks and conditioned variables. You can implement locks using semaphores. So, you know it is not like one is less powerful than the other etcetera; but of course, you know that there is some difference.

Of course, you know you have to lock condition, you need both locks and condition variables to implement semaphores for example all right ok. Let me talk about something else now. So far you know all our abstractions have been of the type where there is some data that there is some data type that we declare like a lock or a semaphore or a condition variable and then, we have some functions and then it is the responsibility of the programmer to enclose his code or you know instrument his code with these functions appropriately. In general, it is very error prone.

This kind of thing is very error prone; you know he puts in acquire in the beginning and then, there are multiple code paths that are going from his function and then, now does he put a release on all those code paths or not. You know that is a very common bug, you have missed a putting a release somewhere for example, right.

Also, a compiler cannot check anything right. So, it is these abstractions are not part of the language per say. The C language has nothing to do with you know whether you are

using a lock or not using a lock whether or anything of that sort or you know so, these abstractions are completely independent of the language and that is not necessarily a good thing because the compiler will never be able to tell you if you made an error right.

So, example if you are did not enclose a you know if you are not bracketing and acquire with the release always, a compiler could have should have told you; but it will not be able to tell you because you know it knows nothing about what you are doing. For him, it is just function call called acquire right. You could have called it something else and you know you would not know.

So, there is a. So, to solve this problem, they there is yet there is an abstraction called a monitor which is a first-class support inside the language, inside the programming language to do a mutual exclusion right. So, the idea for monitor is basically that you know you basically use object orientation or etcetera and you basically define a class which will be called it is monitor and then, you find in you know and this class will have some share data.

Let us say in our case it has the shared data called q and these pointers head and tail and so on and it will have some functions which will operate on the share data all right. So, there is a function called let us say void produce right and char consume all right.

So, and it is the responsibility of the programming language to ensure that there is mutual exclusion between all the functions that are defined inside this class right. So, it is a regular object-oriented programming, there is encapsulation which means these variables are private; only these functions can access these variables. But moreover, these functions will always execute in a mutually exclusive way.

So, if there is some thread which is executing produce no other thread will be able to execute either produce or consume right. So, only one thread could be inside the monitor at any time. So, this class is also called a monitor, a class of this type right is also called a monitor and the idea is that only one thread will be inside the monitor at any time right. So, it is a language that provides you a construct to define mutual exclusion. So far, we have been defining mutual exclusion using our own data type called lock.

Now, here is an abstraction that the high. This is the high-level abstraction provided by the language which allows you to basically say that these functions need to execute in a

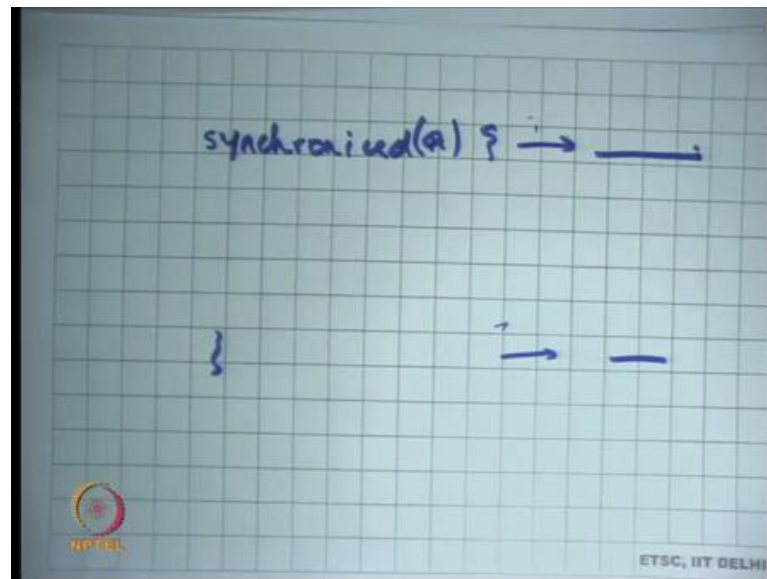
mutual exclusive way. It also provides you encapsulation because it defines the data which is likely to be shared and it says that these because these this data has only available to these functions, it automatically insures correctness because this data has not global in the true sense.

Then, you know it cannot because this data cannot be an accessed by anybody else and because these functions are mutually exclusive by the definition of the monitor, you know your code should be correct. So, it makes things easy to reason about for a programmer, moreover the compiler can now you know has some idea about what this code is doing.

So, he knows that this is the data that is only being accessed by this, so it can do some optimisations in that sense right. It also knows that this function and this function are mutually exclusive. So, it does not need to worry about you know it can do it can freely to optimisations within produce and within consume because it knows that these accesses are going to be visually exclusive etcetera all right. So, for example, I could have implemented my q, the producer consumer my example using a monitor.

So, languages some languages support monitors, C does not support a monitor, but let us say Java supports monitor. So, the synchronised keywords. So, this is there is the keyword called synchronized in Java which allows you to say that this area needs to be mutually exclusive. It is the compiler which will put which is ensured that things are mutually exclusive for you fine. So, if you do not want to use explicit locks, you can just put code inside. You know you can.

(Refer Slide Time: 47:11)



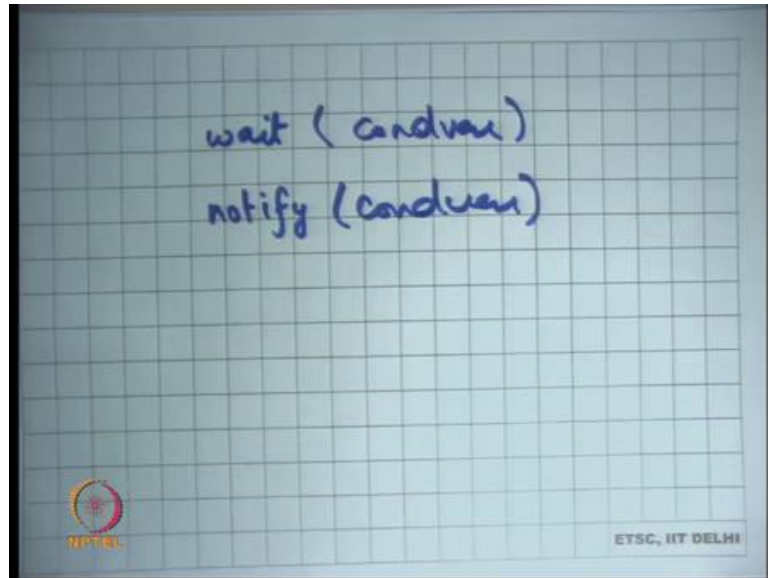
So, Java allows you to do things like you know a class is synchronised so that makes means the monitor or it can mean, or you can also say synchronised blocks. So, you can just say synchronised on you know and write some code here. The compiler will ensure that you know it basically amounts to saying acquired lock here and release the lock here and you can also give an argument to this.

So, this is the lock that gets acquired. So, that you know reduces the chances of you making errors like you know a release is not bracketed within an acquire. Now, because you are using static language construct, you are sure that you know you do not you are not running into errors of that type. So, that is a monitor ok. Basically, what the compiler does is that before and after the calls of these functions, it itself adds calls to acquire and release.

So, there is an implicit lock. In all our previous cases, there was an explicit lock that we had defined. Inside a monitor, there is an implicit lock that the compiler will insert for you right and also you know let us say there were multiple in sensations of this class q, then each of them will have the separate implicit lock for example, fine.

Monitors also need, so, the code within the produce just like you know we have seen the producer code. The producer code may need to wait on the condition. So, what is the; what is the counter part of a condition variable inside in the monitor world?

(Refer Slide Time: 48:50)



So, monitors also have these functions called wait and notify or you know you can call them by different names wait and signal or anything else and you can say some condition variable. Except that when you use wait and signal notify within a monitor, you do not need to use the mutex as a second argument of wait right.

The mutex of as it is which is the second argument of wait is also implicit if the wait has been called inside the code of a monitor. Then, it knows that this is the lock is basically the implicit lock that needs to be released right. So, the monitors wait and see notify do not need to care you know already know which is the mutex that needs to be released. It is the implicit lock of the monitor ok.

Let us stop here and continue our discussion tomorrow.