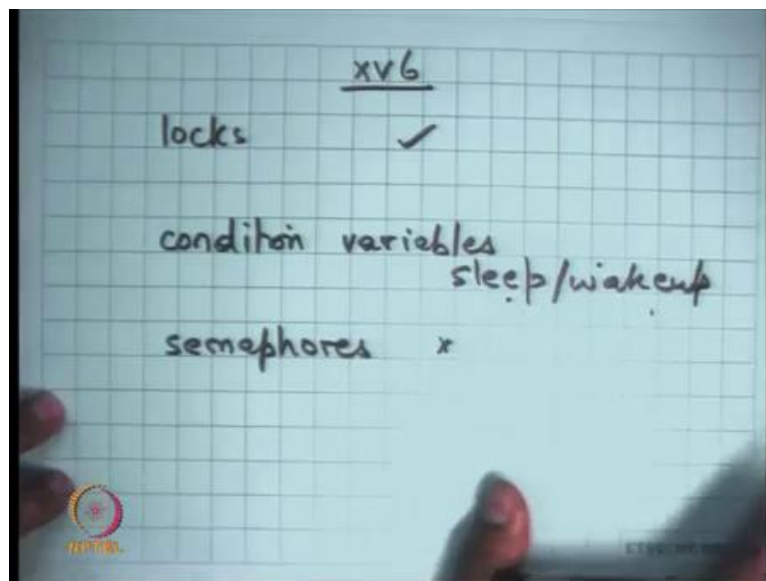**Operating Systems**
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 27
Synchronization in xv6: acquire/release, sleep/wakeup, exit/wait

Welcome to Operating Systems lecture 27 all right. So far, we have looked at locks condition variables semaphores, we also looked at lock free methods like transactions and one manifestation of the act called compare and swap right. So, these compare and swap instructions are also what are called lock free methods of dealing with synchronization and their advantages of doing lock free synchronization as we are going to discuss later in this course right.
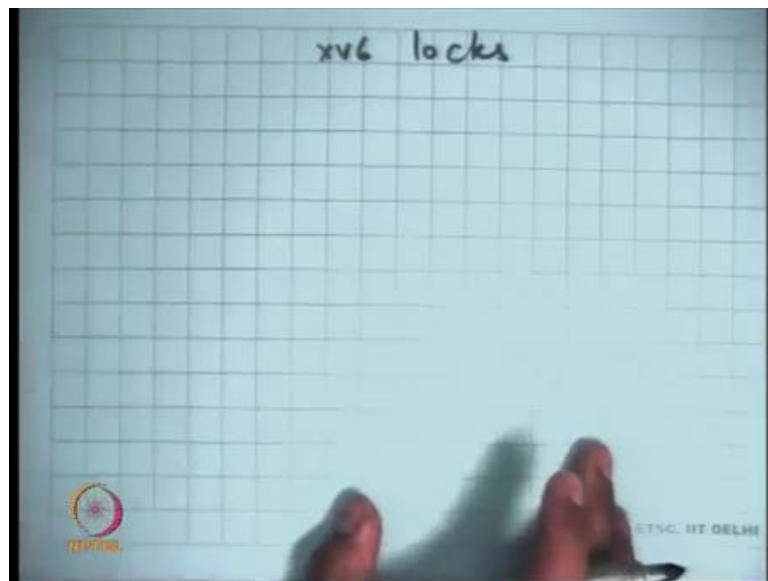
(Refer Slide Time: 00:31)

So, today I am going to discuss these in the context of xv6 and let us look at how an operating system like xv6 is using synchronization under it within itself. And so xv6 uses locks xv6 uses condition variables but you know in a different form. So, instead of calling them condition variables you call them sleep wake up all right. So, the semantics are similar sleep is conditional wait.

Student: Ok.

And wake up is notify. So, instead of calling them instead of calling them condition variables and wait and notify we calls them sleep and wake up. And we are going to see what the subtle differences between sleep are and wake up and wait and notify and does not use; does not use semaphores. So, that is fine semaphores I anyway (Refer Time: 01:42) you can always simulate anything that you need with semaphores using locks and condition variables all right.

(Refer Slide Time: 01:57)

So, let us look at how the acquire function is implemented in so let us look at xv6 locks all right. So, let us look at how xv6 is implementing locks. So, this is sheet fourteen on your listing ok.

```
1464   lk->name = name;
1465   lk->locked = 0;
1466   lk->cpu = 0;
1467 }
1468
1469 // Acquire the lock.
1470 // Loops (spins) until the lock is acquired.
1471 // Holding a lock for a long time may cause
1472 // other CPUs to waste time spinning to acquire it.
1473 void
1474 acquire(struct spinlock *lk)
1475 {
1476   pushcli(); // disable interrupts to avoid deadlock.
1477   if(holding(lk))
1478     panic("acquire");
1479
1480   // The xchg is atomic.
1481   // It also serializes, so that reads after acquire are not
1482   // reordered before it.
1483   while(xchg(&lk->locked, 1) != 0)
1484     ;
1485
1486   // Record info about lock acquisition for debugging.
1487   lk->cpu = cpu;
1488   getcallerpcs(&lk, lk->pcs);
1489 }
1490
```

That is the lock function, that is the acquire function basically, let see what the lock structure is before that. So, lock structure you know if you want to initialize a lock you basically have three fields in the lock one is the name, name is just for debugging purposes. If you want to know what lock it is so it is just for debugging purposes you do not really needed, but they are using it for debugging.

Then there is this lock variable that we know about that is the state of the lock and then also which CPU is holding it that is also again only for debugging you do not really need it strictly, but that is you know it helps in understanding what is going on all right.

(Refer Slide Time: 02:42)


```
1469 // Acquire the lock.
1470 // Loops (spins) until the lock is acquired.
1471 // Holding a lock for a long time may cause
1472 // other CPUs to waste time spinning to acquire it.
1473 void
1474 acquire(struct spinlock *lk)
1475 {
1476    pushcli(); // disable interrupts to avoid deadlock.
1477    if(holding(lk))
1478      panic("acquire");
1479
1480    // The xchg is atomic.
1481    // It also serializes, so that reads after acquire are not
1482    // reordered before it.
1483    while(xchg(&lk->locked, 1) != 0)
1484      ;
1485
1486    // Record info about lock acquisition for debugging.
1487    lk->cpu = cpu;
1488    getcallerpcs(&lk, lk->pcs);
1489 }
1490
1491
1492
1493
1494
1495
```

So, this is the acquire function it you know it disables interrupts. So, pushcli is going to disable the interrupts on the current processor. So, notice that the acquire function disables interrupts uses this loop to atomically set the locked variable to 1, sets the CPU to the current CPU value and the this is some debugging function which allows you to log exactly who called this lock and that is it.

So, what does it mean, when you get out of acquire these interrupts as still disabled right, because you called pushcli and you never called pop right you never enable re enabled interrupts? So, for the entire critical section in this spin lock the interrupts are disabled. So, why does xv6 need to disable interrupts in the entire critical section?

So, one answer is that if this timer interrupt within the critical section, then you know it can get soaked out and somebody else can get to run and atomicity you can get violated is that right. No because you know the other thread should also be taking the lock and it will not get the lock, because it has already called the exchange interaction here right.

Student: Ha.

So, it has set the lock variable to 0 to 1. So, any other thread will not be able to set you know we able to acquire the lock.

Student: Because esters and interrupt handler then basically. So, it may try to reacquire a lock, so it was acquired initially by requirement.

Right, so what if the interrupt handler also tries to access your data right. So, we have within the kernel and it is possible that whatever data you are accessing the interrupt handler also tries to access the same data all right. So, it is not about multiple threads, recall that we have been talking about mutual exclusion across threads.

Mutual exclusion across threads it is easily handled by this atomic exchange instruction right on multiple CPU. But what about mutual exclusion between a thread and the interrupt handler right. If an interrupt gets to run in the middle of a critical section then and the interrupt handler could touch the data that you have in the middle of a critical section, then we need a way to protect it right.

So, what are some ways to protect it you could say let the you know would not the interrupt handler also try to acquire the lock right. But if the interrupt handler tries to acquire the lock then you have a deadlock right there right. So, you got an interrupt, the interrupt handler gets to run and then interrupt handler tries to acquire the lock that is already held right.

So, that is not a good thing and interrupt handler will likely run with interrupt it saver, so that is not a good thing now you basically have a deadlock. So, basically to protect a thread from the interrupt handler from concurrent accesses by the interrupt handler, you disable the interrupts for the entire critical section.

The reason this is acceptable in xv6 is because you will expect that the critical sections are small and you know critical sections that are protected using this spinlock are small and those critical sections are indeed you know need to be protected against accessed from interrupts and we are going to see some examples right.

In general when you write your programs you would not worry about, so let say you have your own program or you know you write a own kernel thread and that thread has nothing to do with an interrupt handler. Then you know you will not use xv6 spinlock you would probably want to use your own spinlock that does not disable interrupts because you do not care about atomicity with respect to the interrupt handler.

So, the interrupts and disabled for the entire length of the critical section acquire leaves the interrupts disabled.

Why do I need to pushcli why cannot I just use cli? Just to ensure that if there are you know nested calls to acquire the nested locks. So, you acquired lock one and then you acquired lock two then you have a count of how many times cli was called right and so that is many times you have to call cli before you actually re enable the interrupts.

So, if you have you know nested locks then that is why you know. So, pushcli is nothing but it just increments a counter it is perceive counter that it increments, basically saying how many times cli has been called all right. So, that is acquire and let us look at the other things. So, notice that this holding function what is it doing it just checking that whether I am holding the lock already, whether this CPU is holding the lock already.

So, this is just a debugging thing really and the reason this is you can check this is because you also have a field called CPU right. So, holding is just going to check that you know if the same CPU try to acquire the same lock twice that is a bug and you want to you know just tell the programmer right away that there is bug in your program right rather than a (Refer Time: 07:38) have to find it in a roundabout way.

So, this is just a debugging aid right in general if you did not have the CPU field you won't have been able to make this check and that is program if you written your code correctly that still correct but this is a debugging aid for you.

Student: Sir.

Right, so I mean the other semantics in this sort of I mean this slightly subtle semantics are that a lock is actually held by a CPU right. So, I mean that falls out from the fact that you would disable interrupt. So, a thread and a CPU basically mean the same thing, because if disable the interrupt. So, the if you disable the interrupt the thread is now stuck to that CPU right it cannot now move anywhere.

So, now the lock is not really you know you can say that the lock is actually for the CPU and not for the thread, on the other hand if you did not disable the interrupts it is the lock should have been per thread. Because if the thread gets switched out and goes to another CPU, then that thread is holding the lock not the CPU that is hold in the lock right. But because you disable the interrupts now the thread is stuck to that CPU. So, you know you can call it a per CPU lock instead of calling at a per thread lock right but.

So, both have these are sort of interrelated, but against the point is that if you did not care about atomicity with respect to the interrupt handler you did not have to do pushcli all right. So, that is acquire right. So, let us look at the other things, so of course you know it try this is this loop we are very much familiar with just you know trying to atomically set it to 1.

And if it is not; if it is not if it does not find it to be 0 it just keeps spinning right that is a spin loop, we know this. And then it just says you know just sets a CPU variable to that I am holding this lock basically, is it to put l k dot CPU above while loop.

Student: This one not.

Firstly, until and unless you have acquired it you should not. Secondly, you know they can be racing accesses to cpu right, because this while loop is protecting. So, you can be show that only one thread or one CPU is within this region, but if you put this above it then you know there is a race condition on the cpu field of the lock right. So, because you put it after while you can be show that there is no racing and then let us look at release.

(Refer Slide Time: 09:43)


```
1500 // Release the lock.
1501 void
1502 release(struct spinlock *lk)
1503 {
1504    if(!holding(lk))
1505       panic("release");
1506
1507    lk->pcs[0] = 0;
1508    lk->cpu = 0;
1509
1510    // The xchg serializes, so that reads before release are
1511    // not reordered after it.  The 1996 PentiumPro manual (Volume 3,
1512    // 7.2) says reads can be carried out speculatively and in
1513    // any order, which implies we need to serialize here.
1514    // But the 2007 Intel 64 Architecture Memory Ordering White
1515    // Paper says that Intel 64 and IA-32 will not move a load
1516    // after a store. So lock->locked = 0 would work here.
1517    // The xchg being asm volatile ensures gcc emits it after
1518    // the above assignments (and after the critical section).
1519    xchg(&lk->locked, 0);
1520
1521    popcli();
1522 }
1523
1524 // Record the current call stack in pcs[] by following the %ebp chain.
1525 void
```

So, this is the release function and once again there is just debugging aid which is if not holding lock then you know you panic that you trying to release a lock that you not holding that is fine and you set cpu to 0 and then you exchange lock with 0. So, once

again why do I need an exchange it, so I could have just set lock dot lock l k dot locked is equal to 0. But instead the programmer chooses to use the exchange instruction to do this why.

Student: (Refer Time: 10:10).

Because, so that there is a fence there is a barrier between, so that this neither the hardware nor the compiler is able to reorder these instructions in anyway all right. So, that these instructions are serialized with respect to each other. In other words, in other words exchange instruction act as an implicit fence or barrier right.

So, the exchange instruction acts as the barrier, so these instructions cannot get reordered with the other instructions because of the exchange instruction. If you had just written l k dot locked is equal to 0, the hardware was free at run time to just reorder these instructions right as we have discussed before all right.

Now, is it ok to put l k dot CPU after exchange l lock dot l k dot locked, I mean is it ok to swap these two instructions two these two statements 1508 and 1519.

Student: No.

No, because there be race conditions on the statement right as soon as you release the lock other people can now.

Student: (Refer Time: 11:13).

Well other people can now access the CPU variable because, another thread could have taken the lock and now it may be trying to write to the lock. And here you are trying to write the locks they are two concurrent access to the CPU variable and so now there are two concurrent access to the CPU variable.

So, you know quite obviously and smartly I am just with the that programmer is just using the same locked field to also protect the internal structures of the lock. Internal fields of the lock variable itself apart from the critical section the fields itself have been protected by this locked field all right.

(Refer Slide Time: 11:49)
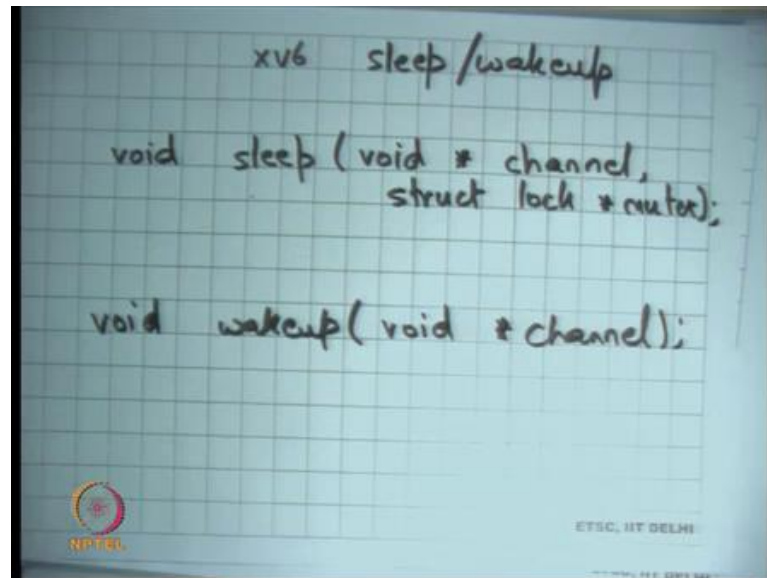
Now, let us look at sleep and wake up all right xv6 sleep and wake up this is just you can call it wait and notify. Now, these are just this is just an incarnation of condition variable and let us see how they are used. So, sleep the so the implementations are on sheet 25 of sleep and wake up sheet 25 and 26 and let us see what the idea behind sleep and wake up.

So, you have a function called sleep, let us look at the signatures of the function. So, there is a function called sleep that takes an argument void star channel, let us discuss what the channel is and struct lock *mutex all right and then there is wake up and wake up just says void star channel all right.

So, this is very similar to you know wait except that instead of a condition variable I am using this pointer called void star channel right. Instead of saying struct cv *cv I am saying void *channel. And similarly here I am saying void *channel right apart from it the semantics are exactly identical sleep is going to wait star waiting on the channel and release the lock release the mutex atomically and wake up is going to wake up in all the threads that are waiting on the channel all right so identical semantics.

Now, question is how I can just say void star channel why I do not need a struct cv here. Recall that unlike locks and semaphores the condition variables are a stateless abstraction, you do not need any state inside the condition variable you do not have any

field call logged or anything that is (Refer Time: 14:00) counter or anything, so it is a completely stateless abstraction.

So, you do not even actually need to have a variable specifically, all you need to say is that there is some number that I am going to sleep on right. That number in the condition variable examples was the address of that condition variable, instead of that address of the instead of actually wasting memory you know using memory to declare that condition variable.

If you just say I am going to sleep on some channel in the sum number that number may be tied to the logic of your program, you do not need to. So, earlier I was a declaring the condition variable for the logics. So, it is example I had not full and not empty, I am saying I do not need to declare these variable because these variables do not have any state anyways.

So, why I am wasting some memory may be instead of doing that let us just use some channel which is just a number right. So, condition variable was also just being used as a number as an address. So, why do not we just use it an as an address all right this is fine.
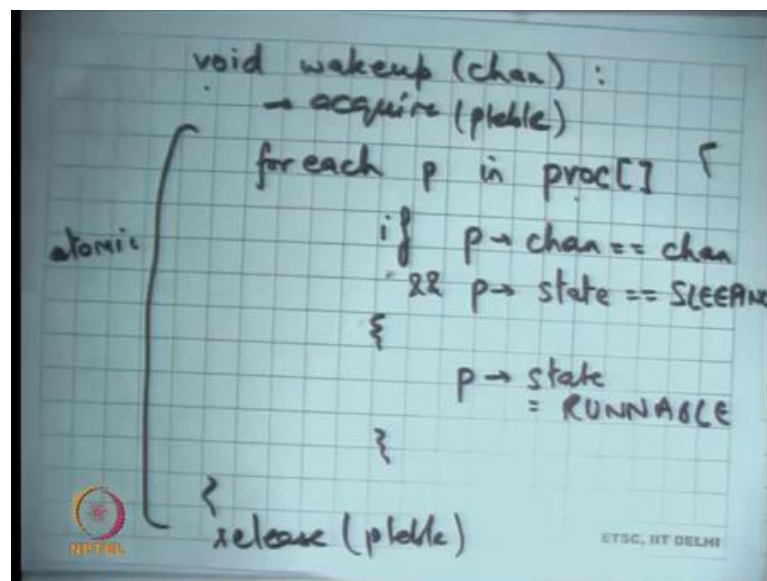
(Refer Slide Time: 15:04)
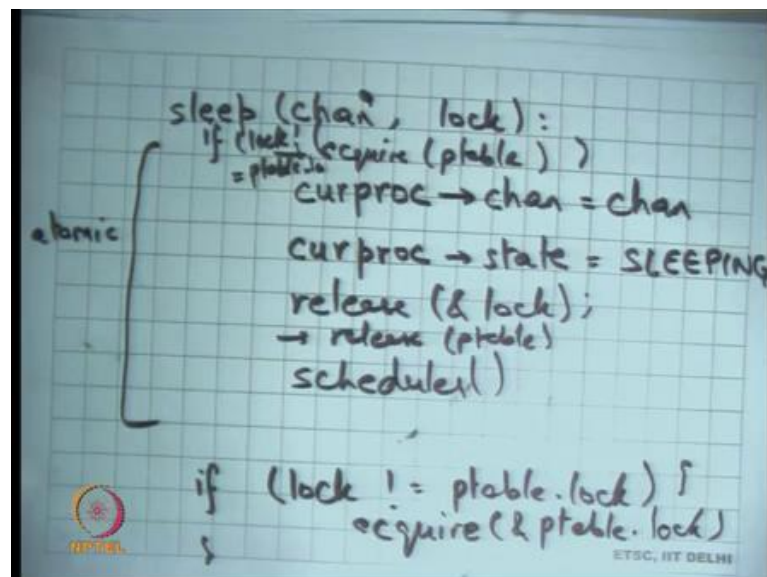
So, let us see I mean at a high level what does a sleep do sleep is going to. So, it takes two arguments I am just going to say let say whoever whichever processor process calls sleep is going to see curproc.channel = channel and curproc.state = sleeping and it is

going to release the lock right and then it will call the scheduler. So, that somebody else can get to run now.

So, he is made my it is own state as sleeping and now when it runs a schedule as the scheduler as not going to schedule this process, because this process is state has become sleeping right. So, I am just trying to develop the implementation of sleep, how it would be implemented under the covers we have discuss the semantics of condition variables already all right.

Now, let us look at wake up and this whole process has to be atomic right. What does atomic mean no other process should be able to inspect curproc while I am doing all this right. For example, if somebody calls wake up then he should not be able to you know inspect curproc while I was sort of in the middle of doing sleep right. So, how I am going to make it make it atomic let us refer the discussion but let just say this is atomic.

(Refer Slide Time: 16:36)

And then let us see how wake up is implemented, a wake up is just this it will just for each proc for each p in the proc table right. So, I have a global list of all the proc or a process if p.channel = channel right and p.state = sleeping then do what?

Student: (Refer Time: 17:30).

P dot state is equal to Runnable right. So, do not we change p arrow channel it will does not matter I mean channel is only valid if the state was sleeping. If the state is runnable

and there is no, I mean channel is meaningless, do I set all the processes to be runnable or only one.

Student: All.

All right that is the semantics of wake up or notify that all the processes are going to wake up, that is the semantic that we have discuss the different semantics in literature. But you know that semantics that we are working that wake up is going to wake up all the threads all right ok. Once again, this whole thing needs to be atomic.

What does it mean for this whole thing you need to be atomic? While I am doing this check no other process should be able to read or write the stuck proc structure right. This proc table should be sort of exclusive to me right or for example this one is also touching curproc is also part of the proc table right.

Student: Yes.

Curproc is just a pointer inside your proc table. So, even this should be atomic with respect to that right. So, in other words you know it should not happen then the in the middle of while I am in the middle of this code gets to run right. So, sleep and wake up need to be atomic with respect to each other. How do you ensure atomicity?

Student: By a lock.

By a lock right, so what do you think should is the easiest option here just have a lock for the entire p table.

Student: (Refer Slide Time: 18:56).

Right got enough, so just have a lock for the entire ptable have acquired ptable here and release ptable somewhere here right.

Student: (Refer Slide Time: 19:10).

Let us say acquire ptable release ptable acquire ptable by a, when I say acquire ptable I am releasing acquire ptable dot lock right and then release ptable ok. So, it makes it sort of atomic is this correct, is there a problem of deadlocks; there will be a problem of deadlocks?

Student: Deadlock (Refer Slide Time: 19:59) ptable (Refer Slide Time: 16:00).

Sure. Firstly, if the lock that we are trying to acquire is a ptable lock, then you know the same thread. So, if this argument is also equal to ptable lock then there is a problem. So firstly, I need to handle this special case because I am inside the kernel it is possible that somebody wants to sleep and the mutex that is protecting that critical section is actually the ptable lock itself.

So, I could just say if lock is not equal to ptable lock, then acquire ptable lock right. Let me write it clearly if lock is not equal to ptable dot lock acquire ptable dot lock right. Similarly, when I release the lock here. I do not need to do two releases I just need to do one release. If lock was equal to p dot ptable.lock is this am I safe now? Interrupt handlers also acquire ptable lock is that a problem?

Student: Because we have (Refer Slide Time: 21:09).

No because we know the acquire function will disable interrupt completely. So, if I am within this region, I can be show that interrupts are disabled. So, no interrupt handler will get to run, so that is the that is a whole idea by we had pushcli inside acquire right. Do not we need to worry about one more thing to ensure that there are no deadlocks.

Student: Order of acquire.

Order of acquisition of locks, do not we see that there are two locks here one is the lock that is already held before we called lock sleep, and another is ptable dot lock. So, there are two locks that are going to be acquired, assuming lock was not equal to ptable lock. So, you know the somebody the caller called acquire ptable lock then you called sleep is going to acquire ptable dot lock.

So, now you have two locks and whenever you hold there is any point you can hold two locks you have to worry about the order, otherwise they can be a deadlock right. So, s this is a; this is a real example of you know why locking kills modularity you have to worry about these things ok.

So, the convention in the operating system is that ptable lock will always be the lowest lock that will ever be taken. So, the ptable has the least priority in some sense right. So, it will be always be the it will be the always be the last lock that is ever taken. So, if you

make that assumption then this code is correct right. But that assumption has to be obeyed by all other parts of the code, that you will never acquire in another lock after you hold the ptable lock.

So, ptable lock will always be the innermost lock in other words right. So, you have to have this global and invariant across your code basically all right. Let us look at so this is you know I have sketched the implementation but let us look at the real implementation on sheet 25.

(Refer Slide Time: 22:52)

```
2551 // Reacquires lock when awakened.
2552 void
2553 sleep(void *chan, struct spinlock *lk)
2554 {
2555   if(proc == 0)
2556     panic("sleep");
2557
2558   if(lk == 0)
2559     panic("sleep without lk");
2560
2561   // Must acquire ptable.lock in order to
2562   // change p->state and then call sched.
2563   // Once we hold ptable.lock, we can be
2564   // guaranteed that we won't miss any wakeup
2565   // (wakeup runs with ptable.lock locked),
2566   // so it's okay to release lk.
2567   if(lk != &ptable.lock){
2568     acquire(&ptable.lock);
2569     release(lk);
```

So, this is sleep on channel and spinlock lock right that is the and once again there are there are these debugging aids, that the current process should not be 0 proc is a global variable which is indicating. What is the current process that running on the CPU it is a per CPU variable if l k is equal to 0? So, lock should not be equal to null I should not be sleeping without the lock. So, lock should not be equal to null these are just debugging aids you can ignore it.

```
2558    if(lk == 0)
2559      panic("sleep without lk");
2560
2561    // Must acquire ptable.lock in order to
2562    // change p->state and then call sched.
2563    // Once we hold ptable.lock, we can be
2564    // guaranteed that we won't miss any wakeup
2565    // (wakeup runs with ptable.lock locked),
2566    // so it's okay to release lk.
2567    if(lk != &ptable.lock){
2568      acquire(&ptable.lock);
2569      release(lk);
2570    }
2571
2572    // Go to sleep.
2573    proc->chan = chan;
2574    proc->state = SLEEPING;        release(lk)
2575    sched();
```

Here is a check if lock is not equal to ptable lock acquire ptable lock and release the real lock, the lock that was used to call it ok. What is happening? So, what did I have I had something different I said if lock I said if lock is not equal to ptable lock acquired ptable lock, else do not acquired ptable lock you know you already have ptable.

Here he is saying if lock is not equal to ptable lock acquired ptable lock that is the same. But he also releasing the lock here, in my code I said you know let us I was releasing lock here does not matter if I release the lock here as supposed to release in the lock here.

So, I was releasing the lock here he is releasing the lock here does not matter does not matter, because you know you already held ptable lock. So now, you have created mutual exclusion between you know between the data. So, the data that you are going to access is mutually exclusive because, so is so this area is mutually exclusive because of ptable lock ok. Because you all you are doing is accessing the proc structure and assuming that you hold the invariant, that whenever you touch the proc structure you will hold the ptable lock this area is anyways protected from race conditions.

So, you do not really need to hold the lock for all that long you could have held it, but you know it is also to just release it a priory all right. So, let say I have released the lock here and I am somewhere here. Let say in other thread calls wake up at this point what will happen, another threads call wake up. So, wake up will try to acquire the ptable lock

and you will not get it right, it will wait for this whole thing to complete the scheduler to get called.

The scheduler recall is going to cause the ptable lock to get released whoever gets to run next is going to release the ptable lock. Recall that the schedulers in variant was that whoever call the scheduler should have held the ptable lock and then call the scheduler and the schedulers going to schedule somebody and that somebody is going to release the lock for you

So, this was sort of very very different kind of the structure where one thread acquires the lock and another thread released the lock right. So, this was the different pattern and so that is fine the ptable lock will be released at this point and, but you know you have ensured mutual exclusion with respect to the wakeup function.

So, if we had another acquisition of a lock here, then it would be much more prudent to release the lock as soon as possible. So, that you do not you do not have more ordering dependencies right. Because if you have another acquire here then you know you would have had to had have an ordering dependency between the l k that you have as an argument and this particular lock also, but you know.

Student: (Refer Time: 26:06).

But you do not have any acquisition that is true all right and let us look at the wake-up function.

(Refer Slide Time: 26:14)


```
2613 void
2614 wakeup(void *chan)
2615 {
2616    acquire(&ptable.lock);
2617    wakeup1(chan);
2618    release(&ptable.lock);
2619 }
2620
2621 // Kill the process with the given pid.
2622 // Process won't exit until it returns
2623 // to user space (see trap in trap.c).
2624 int
2625 kill(int pid)
2626 {
2627    struct proc *p;
28      acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC];
          if(p->pid == pid){
```

Well here is the wake up function, the first thing it does is acquire ptable lock calls this help a function called wakeup 1 and releases the ptable lock all right and what is wakeup 1 doing let us see wakeup 1 is just above it.

(Refer Slide Time: 26:26)


```
14:35 2012  xv6/proc.c  Page 10

/ Wake up all processes sleeping on chan.
/ The ptable lock must be held.
static void
wakeup1(void *chan)

  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;


// Wake up all processes sleeping on chan.
void
wakeup(void *chan)
```

Wake up 1 is just iterating over the ptable and checking if state is equal to sleeping and channels equal to chan then state is equal to RUNNABLE right. So, this whole logic is being called with ptable lock held all right ok. So, we understand how sleep and wakeup can be implemented under the covers in a and recall that you know.

In doing this we have make sure that the sleep is not just going to sleep putting the process to sleep in sleeping state. But also releasing the lock and is doing it in an atomic fashion in the sense that no other process can see the state of a process within this whole thing and the construct that is providing you this atomicity is what ptable lock right.

So, ptable lock is ensuring that there is atomicity in going to sleep and releasing the lock all right. So, let us look at how sleep and wakeup are used inside the xv6 kernel all right. So, we have seen before; we have seen before this the system calls called exit and wait all right. In our discussion on unique system calls and also in your lab you may have come across exit and wait.

(Refer Slide Time: 27:59)
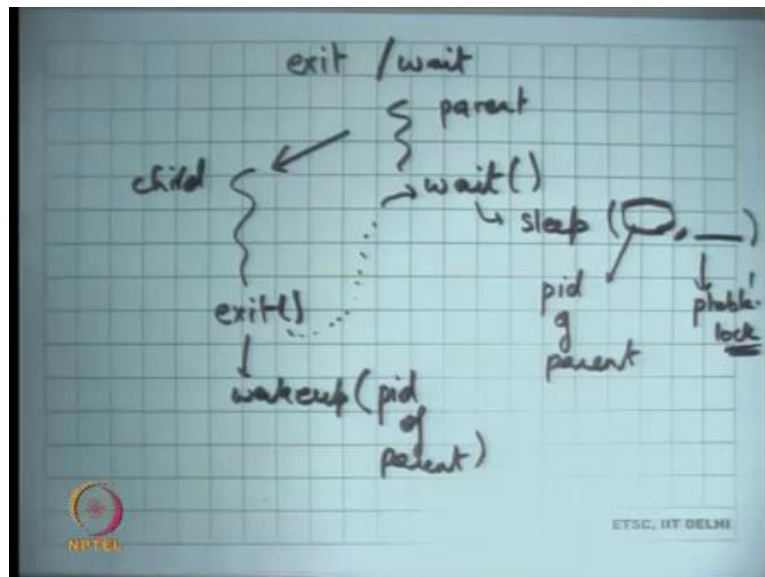
What are the semantics of the exist and wait let us just revise that? So, let us this is the process that calls exit and here is a parent process. So, parent child so if a parent calls wait, then the wait will never return till one of it is children exits right. So, the semantics of wait is that it is going to wait till one of the children have exited. So, the dependency is that you know you will come out of it only after this as called exit right.

So, how does how will an OS implement let us say something of this sort. So, what are the semantics that I have to put myself to sleeping state till some of one of my children exits and so he will put me in the runnable state right. One way to do this is using condition variables or using sleep wake up.

So, the idea is wait internally will call sleep, it will check a condition the condition is has a you know do I have any children and if so, are they running and if so, I will start sleeping on some channel. And then exit should call wake up on that particular channel right. So, that if there was a parent sleeping on that channel then he will wake up and then he can actually come out of the wait.

Student: So, why we are using this semantics rather than child one or signals, why we use signals to wake up the parent (Refer Time: 29:32) condition variables.

No so the signal is just an abstraction at the process level right. In either case you this is I am talking about inside the kernel I am not talking about at the user level. So, inside the kernel you know let us say one process as called wait right. So, exit wait is completely independent of the sick child right. Sick child was there if the parent has not called wait and yet you want to basically inform the parent that I have you know my child has your child has basically exited.

Student: Semantics same type of thing over same process.

So, I mean Unix has both exit wait and sick child you know signal mechanism. So, I am right now talking about exit wait right. So, how is exit wait implemented, even you know signals will require some kind of synchronization it may not require sleep and wake up necessarily. We will require some kind of synchronization which will be very similar basically. But let us I mean if this is a simpler thing to worry about signals is more complicated to implement right and actually explicit does not support signals at all.

In any case I mean even Unix or Linux whatever will have exit wait without any sick child or anything right. So, if a parent is waiting then and a process exits then it is going to wake up the parent. So, that it comes out of the wait ok. So, and I am saying that you know the waiting can be implemented using sleep and telling the parent that I have existed can be implemented using wakeup. I can use what is the lock what is the mutex I should use here inside my sleep and wakeup.

If I want to do it one could mutex to use this ptable dot lock right, after all I am putting myself to sleep which basically you know I could potentially use ptable dot lock. And so let say you know that is easy I can use ptable dot lock I need any mutex that is going to

ensure that you know anything that is going to it is going to give me mutual exclusion with respect to accesses to my proc structure.

So, one lock that gives you mutual exclusion with respect to accesses to your proc structure it is basically ptable dot lock. So, that is easy.

Student: Sir, but that cannot ensure that only (Refer Time: 31:52) child can be include anybody else any process can be.

No that is just the mutex that is just for the mutex.

Student: (Refer Time: 31:58).

Right, now the question is what is the channel what is the channel I should use?

Student: Process id of parent.

Process id of parent right I can say you know I can say parents process id let say does that make sense.

Student: (Refer Time: 32:14).

Why does this channel make sense let us say I have ten children all of them are going to call wakeup on one process id of parent right? So, that is the; that is the channel that I am sleeping on right. And so, I actually make sense because pid of parent is basically you know it is assuming that your convention is that you are going to use pid of parent only for this exit wait business. Then you know another person's child cannot wake you right.

So, one a process cannot be woken by another processes child, you will only be woken up by your own children right. If you use this convention, what if you used some you know instead of using p id of parent let say I just used one constant number 1. So, I am always going to sleep on identifier on this constant called 1 let say.

Student: Any process with exit will (Refer Time: 33:13).

So, any process that exits is going to call wakeup on all the processes that are waiting is that a correctness problem?

Student: Yes.

That may not be a correctness problem because you recall that usually you would, when you come out of sleep you check the condition again. And so, what will happen is if you are sleeping on some global channel that shared across all the processes, then you know you will wake up all these other processes. But all these processes if you have written your code correctly again going to check the condition and everybody accept your own parent is going to find the condition to be false.

And so, all of them going to go back to sleep your parent is going to come out it is wasteful it is not a; it is not incorrect right. So, choosing the channel identifier is just a matter of you know figuring out.

Student: (Refer Time: 34:00).

What are you know what are my; what are my communication patterns and then figuring out the channel the identified that is common to this to the pattern? So, in this case it is a parent id. But if you something which was more course grained right like I took the extreme example of a global identifier called 1. Assuming you have enclosed your sleep loop with a while condition and a check that still correct, but it is wasteful all right. So, let us look at the exit and wait implementations the real ones on sheet 23 and 24.

(Refer Slide Time: 34:39)

```
353 void                    parent calls wait() to find out it exited.
354 exit(void)
355 {
356     struct proc *p;
357     int fd;
358
359     if(proc == initproc)
360         panic("init exiting");
361
362     // Close all open files.
363     for(fd = 0; fd < NOFILE; fd++){
364         if(proc->ofile[fd]){
365             fileclose(proc->ofile[fd]);
366             proc->ofile[fd] = 0;
367         }
368     }
369
370     put(proc->cwd);
371     proc->cwd = 0;
372
373     acquire(&ptable.lock);
```

So, that is your exit function, this function will be called if you if a process makes the existing system call and well it just says you know here some debugging aid the first

process the init proc should not get ever exit. So, that just debugging it. Close all the open files right we have seen this every process has file every table file table and so, xv6 also implements these Unix semantics and so if a process is exiting just close all the files ok. So, we are that is fine that is easy.

(Refer Slide Time: 35:14)

```
acquire(&ptable.lock);

// Parent might be sleeping in wait().
wakeup1(proc->parent);

// Pass abandoned children to init.
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->parent == proc){
    p->parent = initproc;
    if(p->state == ZOMBIE)
      wakeup1(initproc);
  }
}

// Jump into the scheduler, never to return.
proc->state = ZOMBIE;
sched();
panic("zombie exit");
}
```

Also, you know this is some so the process that was open that was currently running, it is current working directory that cwd is also sort of in the file system knows that this process is accessing this particular directory. So, that file so nobody can actually delete that directory from under its feed. So, for that reason you know there is a way to lock that directory and we are going to discuss this when we discuss file system.

So, it is basically unlocking the directory saying that I am going to exit. So, I can unlock the directory if some other process wants to delete that directory is free to do that ok. And I said my cwd to 0 current working directory and now is my logic to do implement exit no to do exit wait synchronization.

So, I do acquire ptable.lock and I wake up on proc dot parent. So, instead of the pid of the parent, I am using the pointer of the pc. So, here I am actually using the pid itself. So, proc dot parent actually the pid of actually it is the pointed to the proc structure of the parent right.

So, instead of using the pid just use the pointer and the and the proc structure that is equally good right there is a one to one correspond pid in the proc structure. So, proc dot parent you just wake it up right, notice that I am not calling wakeup I am calling wakeup one because I have already the ptable lock right.

So, here is another example why locks skill modularity, I have to worry about which function will acquire the lock and which will not acquire the lock what locks I already have you know. So, it is not nice ok. So, I will just wake up the proc dot parent and just fall through and if I have any children then, so then I just iterate over the proc table notice that I have the ptable lock. So, I can safely assume that these accesses are atomic.

So, ptable.proc to p so look at all the processes, if I am the parent of that process right which means that this process is my child, then do what I am going to you know I am going to exit. So, basically, he is you know the offend process has to be connected to the init process, recall that this is something this sort of semantics that we discuss.

That if the parent exits then all it is children are connected to the init proc and init proc is going to call wait. So, that there are no zombies remaining right. And if p dot state is equal to zombie then wake up init proc what is happening here. So, init proc is probably already call is already in the wait loop right. So, init proc is just constantly calling wait, just to collect all the sort of zombie processes and now.

So now, I was the process and I had a child process I forgot to call wait on the child process. So, that child processes just languishing as zombie there right. Now I am going to I am exiting. So, what I can do is all my zombie process if I see any zombie process, I can tell the init proc to wake up, because I have attached a zombie process to him.

So, now his wait you know wait is going to return, so that is what I am doing. So, I am waking up the init proc, so that his wait returns if he is inside the wait right. This is to wait to ensure that your parent returns. So, a process that is exiting you are making sure that your parent returns from a wait, this is to make sure that init proc returns from the wait if any if it is inside right, because you just attaches zombie process to the init proc is right.

And then you make your state as zombie right, notice that I am not cleaned up my virtual memory, I will not cleaned up my case stack I will not de-allocated my virtual memory

my page table still remains all the data that I have I am not freed it right. I have just mark my state as zombie and call the scheduler what is going to happen is that the my parent process when it is going to run is going to call wait and that is going to a de allocate my page table and my case stack ok. Why cannot I just de allocate my page table here.

Student: (Refer Time: 9:22).

Because I am Standing on that page table right, that is the page table that is currently loaded. This exit function is being called in the context of the process that is currently running.

Student: Yeah.

And so, I cannot just de allocate the page table because that page table is actually the one that is loaded into the hardware right. Now you only change your state to zombie and now when the other process gets loaded now the you know now you are of the CPU and now you can be de allocated.

Similarly, case stack cannot be de allocated because this entire function is executing of that case stack. You cannot just de allocate the case stack it is your parent who will de allocate this case stack (Refer Time: 39:58).

Student: Sir, but we can de allocate the user page table as an.

You can de allocate the user side of the page table that is an optimization and you know the real operating system will perhaps do that. But you know xv6 just makes it simpler and says the parent will de allocate everything ok.

Student: Sir in case of (Refer Time: 40:19) these scheduler functions was de allocating the case stack as an. So, is that also possible optimization as in the case stack once we will reach the scheduler function and you have seen that the process from which you have just come into that has not exited. So, what I do is it de allocate.

Right, so the question is instead of waiting for the parent to de allocate the stack, is it possible to just you know on every context which check if my previous process has become zombie. And if you know on the context switch as soon as your context switch.

Student: (Refer Time: 40:52).

So, you have figured out that the previous process that just got context switched out as become zombie then you can just de allocate it is structures. But you know only some of it is structures you still have to maintain it is proc structure, because the parent may call wait. You have to for example, store the exit status return value exit code right.

So those are all possibilities, but you know that there is a tradeoff in doing that. On every context which you are going to check whether the previous process context which process has become zombie or not right that itself has a cost some cost to it. So, you know whether it is a worth file thing to do or not i is something that has a that is (Refer Time: 41:27) basically a design decision ok.

(Refer Slide Time: 41:33)


```
struct proc *p;
int havekids, pid;

acquire(&ptable.lock);
for(;;){
  // Scan through table looking for zombie children.
  havekids = 0;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent != proc)
      continue;
    havekids = 1;
    if(p->state == ZOMBIE){
      // Found one.
      pid = p->pid;
      kfree(p->kstack);
      p->kstack = 0;
      freevm(p->pgdir);
      p->state = UNUSED;
      p->pid = 0;
      p->parent = 0;
      p->name[0] = 0;
      p->killed = 0;
```

Let us look at wait, so here is wait I just go through my so. Firstly, I acquire the ptable lock, once again what is wait going to do it is going to go through the entire ptable and it is going to check if I am the parent of this particular process and if it is zombie, then I can just return right I called wait on something that is already existed. So, I can just return right because that is already zombie. So, I can just return from there.

So, I just sort of go through the ptable, if I am not the parent of that proc of that p then continue. Otherwise I am the parent I just check if it is a zombie. I found the zombie my wait does not need to sleep my wait can just return. So, I found one I can just free all it is things. So, I can free it is case stag, I can free it is page directory, I can set it is state to unused right.

```
if(p->parent != proc)
  continue;
havekids = 1;
if(p->state == ZOMBIE){
  // Found one.
  pid = p->pid;
  kfree(p->kstack);
  p->kstack = 0;
  freevm(p->pgdir);
  p->state = UNUSED;
  p->pid = 0;
  p->parent = 0;
  p->name[0] = 0;
  p->killed = 0;
  release(&ptable.lock);
  return pid;
}
}
// No point waiting if we don't have any children.
if(!havekids || proc->killed){
  release(&ptable.lock);
```

And I am doing this under the protection of ptable lock recall and I just release the ptable lock and I return the process id of the process that I just released right that the that is the one that was zombie. So, return the pid.

Student: Sir do we need to actually put set these field to 0 because anyways once you have drafted and use, we are not going to access (Refer Time: 42:42)

Sure. I mean you do you need to set it to 0, because you have set it to unused, I mean those small things can be done right. I am not sure you know what other parts of the code are relying on you know how what invariants they maintain but you could do that sure all right.

```
2426          p->killed = 0;
2427          release(&ptable.lock);
2428          return pid;
2429        }
2430      }
2431
2432      // No point waiting if we don't have any children.
2433      if(!havekids || proc->killed){
2434        release(&ptable.lock);
2435        return -1;
2436      }
2437
2438      // Wait for children to exit.  (See wakeup1 call in p
2439      sleep(proc, &ptable.lock);
2440    }
2441  }
2442
2443
2444
2445
2446
```

Now if I do not have kids right which means I do not have any children and I called wait I do not need to wait for anything right because I do not have any children. So, and I am calling wait. So, I just return with a minus 1 status that is all right.

And then there is a field called proc dot killed I am going to discuss this next time let us wait on that. But at this point I know that I have some children this process has some children and I have some children that are not yet zombie. So, what I have to do I have to wait for them to become zombie. So, I just sleep on proc and this is the ptable dot lock that is protecting that right. So, that is basically exit wait.

So, now this process has become will go into a sleeping state as soon as somebody exits, he is going to call wake up and the channel is going to be his parent. I slept on my own id right he is going to wake up on his parents id. So, that is how this synchronizing happening ok.

Let us stop here and I am going to discuss the kill system call and how the killing system call is implemented using these mechanisms and how the id device driver. However, disk device driver uses synchronization and how device driver works internally to do this thing. And the also we are going to discuss virtual memory and we are going to start the topic of virtual memory and paging in the next lecture.