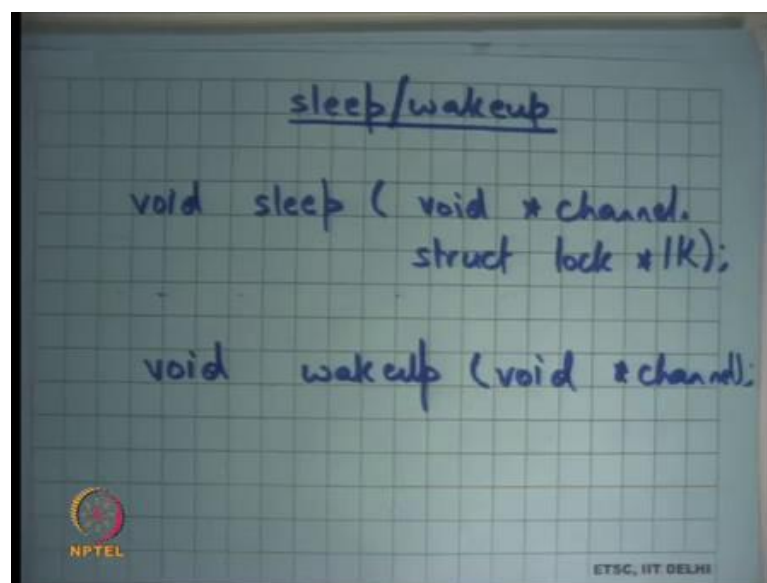


Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 28
More synchronization in xv6: kill, IDE device driver; Introduction to Demand
Paging

Welcome to Operating System lecture 28 right.

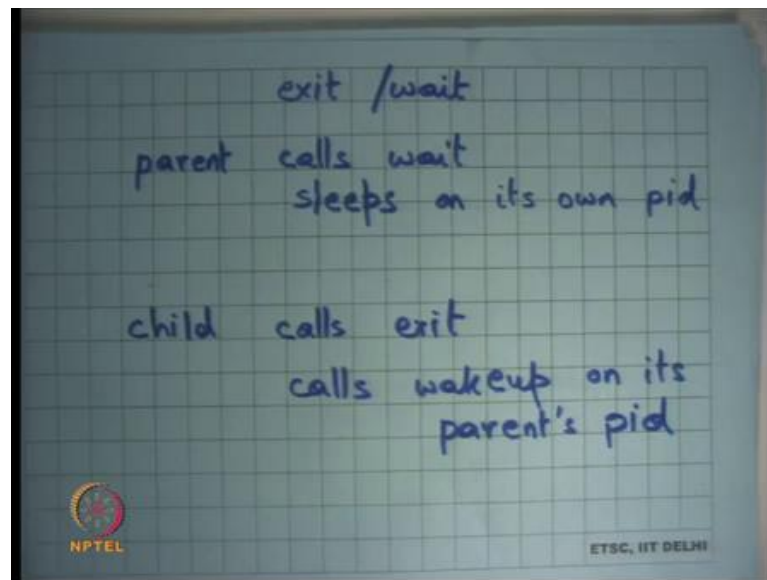
(Refer Slide Time: 00:29)



Last time we were discussing sleep wakeup on xv 6 and we said sleep wakeup is very is actually identical to condition variables except that we do not explicitly declare a condition variable we just sleep on some channel and that channel is some identifier. And, earlier in condition variable you would have declared a condition variable associated with that channel.

Now, you do not need to declare anything separately. So, that way you do not have because a condition variable has no state you do not need any space for the condition variable. So, really this channel is nothing, but an integer right any 32 bit integer that can be used to indicate that I am sleeping on this channel and then you use the same integer to indicate that I want to wake up all sleepers on this particular integer right. So, that is what it means.

(Refer Slide Time: 01:16)



And we said that you know we looked at some examples we said look there is exit and wait you know we understand this exit and wait system call in Unix and x v 6 implements the same. So, parent calls wait let say and if it finds the child that is currently not a zombie and running then or active then it is sleeps on it is own pid right. So, the channel is basically it is own pid and if a child calls exit thereafter then it calls wake up on it is parents pid. And, so that is how you basically ensure that you know you only wake up your own parents and nobody else.

And, we also said you know instead of this convention if we if I had a more general convention where instead of sleeping on my own pid I sleep on some global identifier that would have been also correct provided you are enclosing the wait enclosing the sleep inside in a loop which is checking the condition right. So, if a child calls wakeup on all the parents then all the parents are going all the waiting parents are going to come out of the sleep only to go back to sleep alright.

So, that is it is wasteful. So, it is better to have as fine-grained channels as possible and so this is a, this makes lot of sense. It is exactly as fine-grained as you can get in this particular example right.

Student: Sir.

Yes.

Student: Sir in a current case a when you are sleeping on your own pid and child is waking up the parents pid to be needed to check for the condition because at most only one process can be woken up by a child

Ok.

Student: So.

Alright. So, the question is you know in this case if child you know if I am using this convention that parent is waiting on a it is own pid and child is calling wake up on it is parents pid do I need to enclose the sleep in a while loop, can I just you know use if you know. So, I can be sure that if a parent has woken up from sleep then definitely one of it is children is zombie and so it can be collected right.

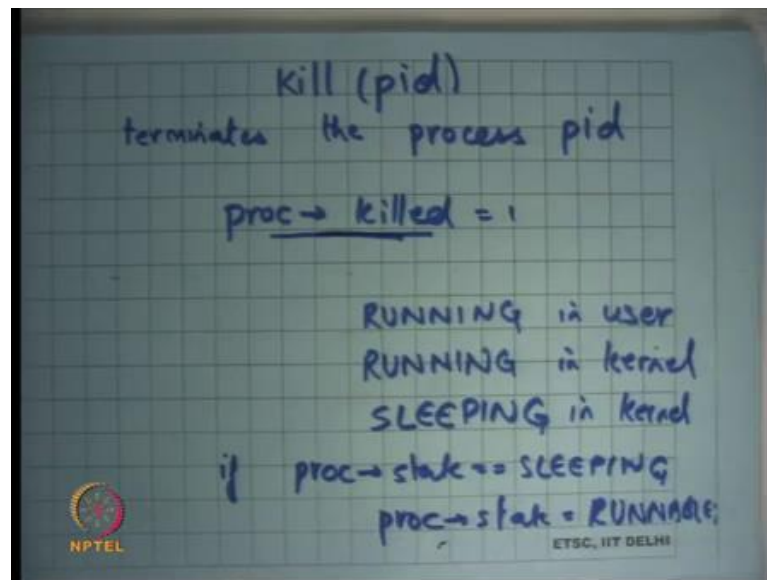
Well, I mean yes you can do such reasoning that is we have seen such a thing in single producer single consumer case, but you know doing these kinds of reasoning is usually error prone. You know in an as I am going to show you next there are other checks that you do make whether the process has been killed in the middle or things like that.

So, it is always better to you know put it in you know recheck after coming out of a wait that whether the condition is true or not and the overhead of rechecking is very small anyways right it just memory operation to basically do it also it is possible that you know. So, you basically said I want to I am going to use this convention for exit wait.

But let say completely unrelated part of the code wanted to use some other synchronization and it also decides to sleep on the pid of some process right. So, let us say there is some synchronization is going on due to some other completely unrelated part from exit wait exit right and so that also sleeps on pid.

And, so now they can sort of you know cross communicate and to protect from errors because of this cross communication you should always enclose your sleep within a while right. If you do not then you have to be very careful you know it become a global invariant then nobody else should be sleeping on the pid for example, right alright. So, today I am going to discuss another system call called kill right.

(Refer Slide Time: 04:36)



So, we have seen the kill system call in the context of Unix where we said that you can you know say kill on a pid and that will you know send a signal to that particular pid to that particular process. So, you can say kill pid with some signal and that signal (Refer time: 04:48) x v 6 does not implement signals.

So, kill pid does not send a signal to the pid, but just terminates that particular process. So, idea is if I send kill on a process id then that process gets terminated. So, how you think kill can be implemented? Can I just you know if some process calls kill can I just immediately acquire the p table lock, go to its process, and free all its structures does that make sense at all? Does that make sense? Yes.

Student: No.

No. Why not? One process called kill pid that that process just acquires a p table lock goes through the proc table finds the process for with that pid and freeze all its structure is that ok.

Student: (Refer time: 05:38).

What are that processes currently running? Right. It is in the middle of something and now you have just you know free deposit structures let say you know you freed up it is case stack and it was running in kernel mode you know suddenly there will be some bad

things suddenly happen. It could be in the middle of some operation you know it has not finish it is operation and now you free it or you completely terminate it you know.

Firstly, you know it is running. So, how can you just free it? If you free it then you have not told it to stop you have to you know some have some way of asking you to stop also it could be in the middle of something and you want it to come out of wherever it is and then exit gracefully right. And, so the way you would implement this kind of a thing is that you know you will basically have a flag inside the proc structure called killed right.

So, there is a structure called killed inside a proc and if one process calls kill pid then I will just iterate over the procs table find the process and set it is killed bit to 1 right. So, by default it is 0, but I will just set it to 1. Now, it is that process which will basically you know if it was in the middle of something then it will just you know when it comes out of it, it is that processes responsibility to know that it I have been killed and so it should exit alright.

So, if a process has been killed it should exit and exit will not free up it is data structures either it will be it is parent who will call wait and then that will free it free it up. In any case so, the point is that you know kill cannot be done immediately because the process may be running. So, the way it is run as you just set a bit in that processes structure and let that process run when the process you know finishes comes out of wherever it was, then it should really check whether I have been killed or not and then it will exit alright.

So, when you set proc dot killed you are accessing the shared proc structure. So, you should be holding the p table lock alright. So, you just hold the you use the p table lock to basically set this bit, but after that it is stack processes responsibility to basically exit after it has checked that it (Refer time: 07:53) alright.

Student: Sir.

Yes.

Student: Is any process allowed to kill any other process?

Is any process allowed to kill any other process well; if the in Unix if the processes belong to the same user then any you know a user is allowed to kill it is own process basically, but one user is not allowed to kill any another users process. Also, you know

modern operating systems like Linux have concept over root user which we all know that root user is super user. So, root user can kill any other users process for example.

So, you can have this access control mechanisms, but you know in x v 6 you just have one user; so, any process can kill any other process alright. So, I am saying that one process just sets the killed bit of the other process and I now expect that process to basically exit right. So, you know what if it does not exit, you know where what if some malicious process, how do I make sure that the that process will definitely exit in some bounded amount of time?

Student: (Refer time: 08:54) called.

So firstly, you do not trust the processes user space, but you definitely trust the processes kernel space right. Anytime the process is executing in the kernel mode it is your code that is running right, it is not the users codes that running it is your code that is running. So, you always trust the processes kernel space right because process in the kernel space is just a kernel thread and you complete trust that code. So, anytime the processes executing in the kernel you can be sure and because you have it in the kernel code you can be sure that it will call exit whenever kill bit is set.

Now, you can say that what if the process remains in the user mode and never comes in the kernel mode. But that is not going to happen because you know you have a timer interrupt that may make sure that every process periodically comes in the kernel mode.

So, whenever it comes in the kernel mode next it is going to check whether something you know whether it needs to exit or not. So, once again this orchestration has been done by the kernel programmer alright. Also let us say I said the killed bit of a process and it was sleeping when it was killed right.

So, let us take a few cases let say let say this process was running when it was killed and let say it was running in user space when it was killed. No problem. Eventually it will come to the kernel space either because it made a system call or because of the timer interrupt and as soon as it comes into the kernel space it should check whether I have been killed or not. And, if it has been killed you know right at the entry you should put a check that if it a have been killed then I should exit and that is a very safe place to exit

because you are not in the middle of any kernel operation; you have just come from user to kernel and you can just asking him to exit right there.

So, this is easy let say I was running in kernel mode alright. So, I am running in kernel mode, I could be in the middle of doing something of course, I am using the k stack and all that and so basically I should wait for the kernel to finish whatever it is doing and then come back quite likely. If it is running in kernel mode either it will return back to the user mode eventually. So, that may be nice place to check the killed flag. So, right just before reentering the user space you can check the killed flag and call exit before it reenters the user space right that is one thing.

If it is running in kernel mode, but it is not going to reenter the user space it is possible that you know the only other possibility is that it is going to sleep right or it is going to do something else. So, all these places where it could be going to sleep for example, before you do that, you should check that it is killed and you should check it all these sort of boundaries where you are not in the middle of something. So, at any place where you have showed that all the kernel data structures are consistent at that point you can pull put this check that if I have been killed then I should exit alright.

Now, let us say I was sleeping alright. So, sleeping of course, can only happen in kernel mode right. It does not make sense that a user is a process is sleeping in kernel mode in user mode you know. If it is in the user mode it must be running something it comes to if it wants to sleep it will come to the kernel mode and then it will call the sleep function to basically sleep.

So, let us it was sleeping in the kernel and you basically set proc dot killed is equal to 0 what will happen is equal to 1. What will happen? The process was sleeping, it will never get scheduled and so you know it will not get you know. So, killed is not going to have any effect, it just going to sleep forever may be. So, not just do not only do you set proc dot killed is equal to 1 you also say if it was sleeping then mark it as runnable alright.

So, here is that here is something sort of dangerous looking. I am saying any process who is sleeping if it has been killed just mark it runnable alright irrespective of you know what was the condition you know on which it was sleeping what is the channel it was

sleeping on does not matter just mark it as runnable it will get scheduled eventually and when it get scheduled it will get to run ok.

Now, it is the responsibility of the programmer to ensure that whenever a process comes out of the sleep it also checks whether the killed flag has been set or not alright and if so then it should exit ok. So, once again it is a kernel programmer who is making sure that you know. Firstly, you need to do this because otherwise you know a sleeping process will never get killed.

And if you need to do this you are violating some you may be violating some invariance because after all the process went to sleep on a some condition and that condition has not yet become true, but you are still made it runnable and so it will come back and it will basically, but it before you do start doing anything you should probably check whether it has been killed and if so it should exit alright.

Student: Sir, but if it is sleeping, we can assume that after some time, it may wakeup and then we can exit.

If it is sleeping I can assume that after sometime it may wakeup and then it can exit well I mean so, let us let me think of an example where it may be sleeping and it may not wake up. So, well let see. So, let say I was a parent and I wanted to wait on my child to exit right and so my child is going to run for a long time and somebody wants to kill me right. So, should I wait for the child to you know finish it is execution before the parent gets killed maybe not right.

So, you want to basically kill it right then you know there should be some bound on the time it takes between the killed (Refer time: 14:54) between the kill operation and the actual act of getting killed right. Otherwise you know you have killed it, but it still appears on your process table that is not a good idea alright.

So, and it may not be always necessary that when you come out of sleep you always check the killed and you exit you know it really depends on a depends on the semantics under which you are sleeping and I am going to sleep you couple of examples to show when you need to exit on being killed and when you do not need to exit on being killed alright. So, let just let just look at the code. So, this is sheet 31 and this is the kill function alright.

(Refer Slide Time: 15:52)

```
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;
2628
2629     acquire(&ptable.lock);
2630     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2631         if(p->pid == pid){
2632             p->killed = 1;
2633             // Wake process from sleep if necessary.
2634             if(p->state == SLEEPING)
2635                 p->state = RUNNABLE;
2636             release(&ptable.lock);
2637             return 0;
2638         }
2639     }
2640     release(&ptable.lock);
2641     return -1;
2642 }
2643
2644
```

So, the first thing you do is acquire p table lock right then you check you iterate over the p table find a process whose pid is equal to the argument, set it is killed value to 1, if it is sleeping mark it runnable release the p table lock and return alright that is it. So, now let us see where the killed is being used. So, this is sheet 26 sorry not 31 sheet 26 alright.

(Refer Slide Time: 16:22)

```
3101 trap(struct trapframe *tf)
3102 {
3103     if(tf->trapno == T_SYSCALL){
3104         if(proc->killed)
3105             exit();
3106         proc->tf = tf;
3107         syscall();
3108         if(proc->killed)
3109             exit();
3110         return;
3111     }
3112     switch(tf->trapno){
3113     case T_IRQ0 + IRQ_TIMER:
3114         if(cpu->id == 0){
3115             acquire(&tickslock);
3116             ticks++;
3117             wakeup(&ticks);
3118             release(&tickslock);
3119         }
3120     }
3121     lapiceoi();
3122     break;
3123     case T_IRQ0 + IRQ_IDE:
3124         ideintr();

```

Let us look at sheet 31 where the killed flag is being checked. So, this is the trap function. Recall that the trap function gets called on every trap. A trap is both a system call or an external interrupt like a timer interrupt or any exception like a page fault right.

So, all these are basically traps and so the trap function gets called. Recall that the all traps function all traps assembly code used to call the trap function.

So, trap function gets called every time and you can see here that at entry you are checking if proc dot killed exit right. Similarly, after that you basically execute the system call and on exit you are saying if proc dot killed then exit right. And, so you will see similar sort of peppering of this code if proc dot killed exit at other places in the code for example, here is another example.

(Refer Slide Time: 17:13)

```
3171 // Force process to give up CPU on clock tick.
3172 // If interrupts were on while locks held, would need to check
3173 if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_T
3174     yield();
3175
3176 // Check if the process has been killed since we yielded
3177 if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3178     exit();
3179 }
3180
3181
3182
3183
3184
3185
3186
3187
3188
3189
3190
3191
3192
INTEL
3194
```

Let say if proc and proc dot killed, and I was executing in user mode then exit right. So, for example, this is the timer interrupt while I was executing in the user mode and the process has been killed since in the in the past then exit.

(Refer Slide Time: 17:29)

```
3154         tf->trapno, cpu->id, tf->eip, rcr2());
3155     panic("trap");
3156 }
3157 // In user space, assume process misbehaved.
3158 cprintf("pid %d %s: trap %d err %d on cpu %d "
3159         "eip 0x%x addr 0x%x--kill proc\n",
3160         proc->pid, proc->name, tf->trapno, tf->err, cpu->id,
3161         rcr2());
3162     proc->killed = 1;
3163 }
3164
3165 // Force process exit if it has been killed and is in user space.
3166 // (If it is still executing in the kernel, let it keep running
3167 // until it gets to the regular system call return.)
3168 if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3169     exit();
3170
3171 // Force process to give up CPU on clock tick.
3172 // If interrupts were on while locks held, would need to check
3173 if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
3174     yield();
3175
3176 // Check if the process has been killed since we yielded
3177 if(proc && proc->killed && (tf->cs&3) == DPL_USER)
```

Similarly, you know you will see something somewhere here right. So, so these are the places notice that he is basically identified some safe places where you should exit you should check and exit. Also, he has make sure that there is a bounded amount of time within which it will exit alright. Now, let us look at the wait and the exit code that we were looking at last time alright just to complete the discussion.

(Refer Slide Time: 18:12)

```
2401 // Return -1 if this process has no children.
2402 int
2403 wait(void)
2404 {
2405     struct proc *p;
2406     int havekids, pid;
2407
2408     acquire(&ptable.lock);
2409     for(;;){
2410         // Scan through table looking for zombie children.
2411         havekids = 0;
2412         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2413             if(p->parent != proc)
2414                 continue;
2415             havekids = 1;
2416             if(p->state == ZOMBIE){
2417                 // Found one.
2418                 pid = p->pid;
2419                 kfree(p->kstack);
2420                 p->kstack = 0;
2421                 freevm(p->pgdir);
2422                 p->state = UNUSED;
2423                 p->pid = 0;
2424                 p->parent = 0;
2425                 p->name[0] = 0;
```

So, here is let say the wait code right. This is sheet 24. I am looking at the wait code and recall that I was acquiring the p table lock iterating over the p table.

(Refer Slide Time: 18:26).

```
2417 // Found one.
2418 pid = p->pid;
2419 kfree(p->kstack);
2420 p->kstack = 0;
2421 freevm(p->pgdir);
2422 p->state = UNUSED;
2423 p->pid = 0;
2424 p->parent = 0;
2425 p->name[0] = 0;
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 wakeup call in proc_exit)
2439 sleep(proc, &ptable.lock);
2440 }
```

And, you know and if I find that there was some child if I have a, if this process has the child that has not that has not yet a zombie then I will go to sleep right. So, let say I was sleeping. So, let say this process was sleeping here and it is really waiting for one of it is children to exit.

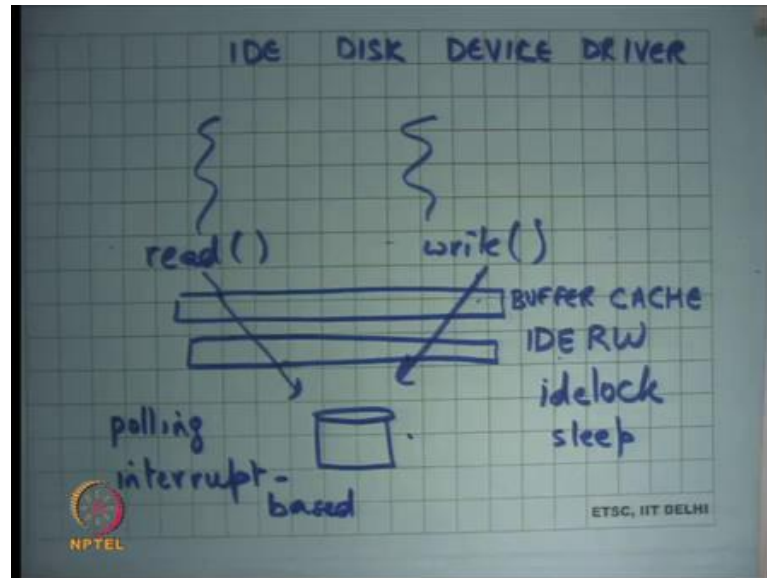
Now, somebody calls kill on this particular pid. So, what will happen it will be marked runnable it will come out of sleep immediately. Even though none of it is children have actually become zombie, yet it will come out of the sleep it will the nice thing now is I will go back and check the condition again right. So, here is an example where you know you actually even though the you have chosen the channel correctly you had to check the condition again because you there are other reasons why could have been woken up right.

So, you check the condition again and once again you find that you know there is nothing that as yet become a zombie there is none of my children have become a zombie I come here. But, this time I find that I have been killed and so I will release the p table lock and return minus 1 and return minus 1 will you know eventually go back to system call exit and there you are going to check whether it has been killed then it will it is going to call actually exist alright.

So, in this loop of checking here is an example where you basically you know if somebody has been woken up from a sleeping state unceremoniously not correctly then

you will basically the programmer is handling correctly and this is very important to handle it handle it correctly alright ok. Now, let us look at another example of how wait and notify is used and I am going to look at the IDE device driver the disk device driver.

(Refer Slide Time: 20:09)



So, IDE disk driver device driver. As you all know a computer system is made up of a CPU, main memory and a lot of devices right and one of these devices is the is the disk hard disk magnetic disk let say and IDE is just one interface one standard interface to communicate with the disk. So, it is a standard the IDE device manufacturer who confirm to that standard and the operating system developer will also confirm to the standard and so they can talk to each other alright.

So, let us see what is what happens basically there are multiple processes right and they may be calling read or write system calls and eventually you know the file descriptors of these read and write system calls maybe pointing to the disk alright. So, when they make a system call, they become kernel threads and these kernel threads will try to access the disk and read or write data from the disk.

So, what sets in the middle is this IDE driver alright on the xv 6 code this function is called IDE RW IDE read write alright. The multiple threads who are coming inside this trying to access the disk and they all go through this device driver to be able to access the disk. Also, because the disk is really slow as we as we already know what we also keep

here is what is called a cache right. So, if you read something from the disk you basically store it keep it in that cache.

So, that you know if some if somebody else wants to access it then it just gets satisfied from the cache and notice that this cache is the shared cache. It is shared across all the processes right. So, if one process tries to read something then it comes into the cache and another process tries to read the same thing then it does not need the disk access you save disk access because of a shared cache and so because it has to be a shared cache this cache has to be implemented inside the kernel right. If it was not a shared cache you could have implemented the cache at users space also I mean does not matter right, but because it is a shared cache it has to be a implement into the kernel and almost every kernel has it and this is call the buffer cache that is a common name for it alright.

So, buffer cache has many buffers and so it just reads data from the desk into the buffer and all subsequent request are checked against the buffer cache if you can satisfied from the cache then you just return it from the buffer cache otherwise you go to IDE RW to reach the disk alright. So, firstly there is just one disk and there are multiple logical processes right and multiple threads the threads could be logical threads or you know they could be logical concurrency or physical concurrency in either case you need to provide mutual exclusion on your access to disk. So, only one thread should be accessing the disk at any time right.

So, so that is needed. So, what basically what you do inside IDE RW you have some lock. So, on x v 6 you have an idelock. So, every thread before it acts to access the disk must acquire the lock and then access the (Refer time: 23:43). So, that make sure that only one thread is actually controlling the disk at anytime. So, one thread comes in talks to release gets it is data or writes the data reads or writes the data goes out then another thread comes in and so on. So, it is completely sequential in that sense also as we know that a disk is a really slow right.

So, if there are so, it is quite likely that there are you know lots of threads that are waiting for the disk to be available. It is this has a very slow thing then a queue get will usually get built up in front of that slow resource right and so the best thing to do would be to sleep as opposed to spin right because you the disk is basically milliseconds and you do not want to spin for milliseconds long. So, you would probably want to sleep. So,

you will basically sleep on something and so when the disk actually gets done then you will wake up the corresponding process alright.

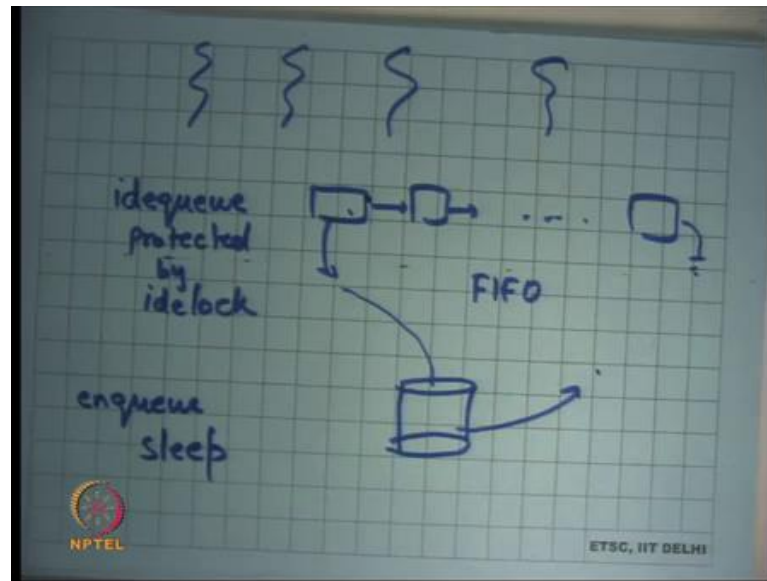
Also, device drivers are basically return in one or the two ways: one is called polling and the other is called interrupt based right. So, what this means is let say I request the disk to do something let say I ask the disk to say to read some data I want to read the sector number 10. Now, I can keep asking the disk. So, the disk is going to take some time and so I can keep asking the disk are you ready are you ready are you ready alright. So, that is called polling right you keep polling the disk for the data alright.

So, you basically just keep checking at some periodic interval whatever you think is valid you just keep polling the disk at periodic intervals and as soon as you get the data you give it back alright. Polling is alright. So, that is one way to do it the other way to do it is interrupt base where you ask the disk to do something and then you go to sleep alright and now the disk has been configured to generate an interrupt whenever it finishes.

So, I do not need to continuously keep asking it will call me back using an interrupt. So, it will generate the interrupt and the interrupt handler I when the interrupt gets generated. I can check whether my job has been completed and if so, I will do it take it right. So, there are two ways to deal with devices in general polling and interrupt ways slow devices are better dealt with in interrupt ways manner fastest devices are better dealt with in polling ways manner alright.

So, first so, the other thing is this IDE device has to be interrupt based alright. So, what is going to happen is that a thread is going to take the lock it is going to enqueue it is request into a list into a queue. And, then it is going to go to sleep whenever the device finishes it is going to generate an interrupt and the interrupt handler is going to call wake up right. So, that process can actually continue. So, who whosever job is been done that process will be called that you will basically wake up that particular process alright

(Refer Slide Time: 27:09)



So, the way it works on x v 6 is basically that you have a queue which it calls idequeue and this has all the buffers data waiting for disk access each and so there are multiple threads that try to add to this idequeue if you basically take. So, protected by idelock; so, if the multiple threads who want to access the disk, they actually trying to they first contained on idequeue. So, they contained an idequeue and enqueue they their request on this idequeue and this idequeue is processed in FIFO order by the disk right.

Now, the disk; so, now the driver basically picks up one request from here and asks the disk to do it the disk generates an interrupt when it is done and it basically. So, whoever was the process that is that was that requested this particular thing that particular process is woken up and that process can now go on it is way.

So, the process enqueues it is request on the idequeue and goes to sleep alright. So, enqueue and sleep and when the and the ide device takes one request at a time and when it is done then it wakes up the sleeping corresponding sleeping process alright. So, let us look at this code this is IDE RW on sheet 39 ok.

(Refer Slide Time: 28:47)

```
nc buf with disk.
B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
se if B_VALID is not set, read buf from disk, set B_VALID.

(struct buf *b)

uct buf **pp;

!(b->flags & B_BUSY)
anic("iderw: buf not busy");
(b->flags & (B_VALID|B_DIRTY)) == B_VALID)
anic("iderw: nothing to do");
p->dev != 0 && !havedisk1)
anic("iderw: ide disk 1 not present");

ire(&idelock);

ppent b to idequeue.
next;
pp=&idequeue; *pp; pp=(&pp)->qnext)
```

So, this is the function IDE RW. It takes an argument buffer which is you know some point into your buffer cache and semantics are that if the buffer is you know buffer the struct buf has the field called flags and if the B_DIRTY field is set in the flags then you should write treated as a write request.

So, you want to write this buffer to the disk else if B_VALID is not set then read the buffer. So, you know whether it is a right request or a read request is encoded within the buffer in it is flags whether if it is valid if it is not valid then it is a read request if it is dirty then it is a write request alright.

(Refer Slide Time: 29:31)

```
// Sync buf with disk.
// If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
// Else if B_VALID is not set, read buf from disk, set B_VALID.
void
iderw(struct buf *b)
{
    struct buf **pp;

    if(!(b->flags & B_BUSY))
        panic("iderw: buf not busy");
    if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
        panic("iderw: nothing to do");
    if(b->dev != 0 && !havedisk1)
        panic("iderw: ide disk 1 not present");

    acquire(&idelock);

    // Append b to idequeue.
    b->qnext = 0;
    for(pp=&idequeue; *pp; pp=(*pp)->qnext)
```

So, here are some debugging aids you know firstly alright. So, we can ignore this for a minute, and we say either it should be valid, or it should be dirty right cannot be that it is neither.

(Refer Slide Time: 29:47)

```
3959     panic("iderw: buf not busy");
3960     if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
3961         panic("iderw: nothing to do");
3962     if(b->dev != 0 && !havedisk1)
3963         panic("iderw: ide disk 1 not present");
3964
3965     acquire(&idelock);
3966
3967     // Append b to idequeue.
3968     b->qnext = 0;
3969     for(pp=&idequeue; *pp; pp=(*pp)->qnext)
3970         ;
3971     *pp = b;
3972
3973     // Start disk if necessary.
3974     if(idequeue == b)
3975         idestart(b);
3976
3977     // Wait for request to finish.
3978     while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
3979         sleep(b, &idelock);
3980     }
```

And, or I mean either it should be not valid, or it should be dirty basically ok. So, it should be either a read request write request. So, first thing you do is you acquire the idelock that is mutual exclusion the next thing you do is you append the block the buffer to the idequeue. So, idequeue is some shared structure as the global variable and you just

append the block to the idequeue you append it to the end of the idequeue because it is a FIFO. So, you append to the end of the idequeue.

So, you just iterate over the idequeue till you reach the end and then you append to the idequeue alright. So, just in this figure just go till the end and then put your new buffer here alright. Finally, we check if idequeue is equal to b. So, b is the buffer that was the argument to this function. So, this is the buffer that I want to read or write if idequeue is equal to b is checking what.

Student: (Refer time: 30:47).

If I am the first element in this queue that is what it means right; if I am the first element in this queue, then start the start the disk alright. So, basically it means that the queue was actually empty right now and this is the first request has been made to the disk. So, disk is actually not spinning right now it is not working right now. So, I need to start the disk. So, there is this function called idestart that is going to use in and out instruction to basically tell the disk to start there is some standard which is basically telling it to start.

(Refer Slide Time: 31:14)

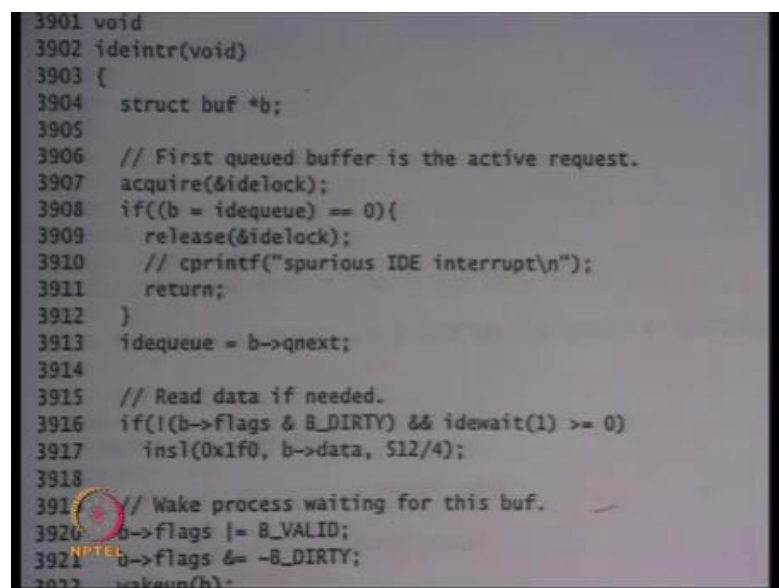
```
963     panic("iderw: ide disk 1 not present");
964
965     acquire(&idelock);
966
967     // Append b to idequeue.
968     b->qnext = 0;
969     for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
970         ;
971     *pp = b;
972
973     // Start disk if necessary.
974     if(idequeue == b)
975         idestart(b);
976
977     // Wait for request to finish.
978     while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
979         sleep(b, &idelock);
980     }
981
982     release(&idelock);
983 }
```

So, this disk has been started. If I am not the first element in the queue then I do not need to started it is already started right and if it is already started it will you know it will as we going to see it will basically just it is just serving one request and as soon as it is done serving that request it will pick the next request in the queue and so on right.

So, I do not need to do anything if it is already started it knows what to do next as the long as I have appended something to the queue I am fine alright and then I keep waiting on this condition. The condition is that whatever I wanted to do read or write if it is happened if it till it has not happened keep sleeping right. The mutex is idelock and the channel is the buffer on which you wanted to do this operation alright.

Let us look at you know may be it will become clearer if you look at what happens if the disk finishes.

(Refer Slide Time: 32:21)



```
3901 void
3902 ideintr(void)
3903 {
3904     struct buf *b;
3905
3906     // First queued buffer is the active request.
3907     acquire(&idelock);
3908     if((b = idequeue) == 0){
3909         release(&idelock);
3910         // cprintf("spurious IDE interrupt\n");
3911         return;
3912     }
3913     idequeue = b->qnext;
3914
3915     // Read data if needed.
3916     if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
3917         insl(0x1f0, b->data, 512/4);
3918
3919     // Wake process waiting for this buf.
3920     b->flags |= B_VALID;
3921     b->flags &= ~B_DIRTY;
3922     wakeup(b);
3923 }
```

So, when the disk finishes the IDE interrupt function gets called the interrupt handler and what is going to do it is going to again acquire the idelock right once again. So, the interrupt handler; so, the disk is finished it has called the IDE interrupt handler it will acquire the idelock and it will check if the idequeue is now if the idequeue is null that basically mean I do not know why I got this interrupt it is possible that device creates a spurious interrupt. So, I just say you know it is a spurious interrupt do not worry about it and usually if I got an interrupt it basically means that they must have been something in my idequeue right that is why I was working and that is why I generated the interrupt.

But, if the device for some reason generated a bad interrupt, I should tolerate it. So, the programmer is tolerating it. Otherwise basically say you basically say that you know the whatever the top of the queue has been has been addressed has been serviced. So, you

basically move the top of the queue to it is next, you read the data using the in instruction from the disk into the buffers data b dot data and so b b is the top of the queue right.

So, you are serving the top of the queue. You check you read the data into the b these buffer you set it is flags to say that now it is valid or you know it is not dirty anymore it depending on whether the read or write request and then you call wake up on b right. Why do you call wake up on b because the process who was who made this request must be sleeping on b.?

So, now you call wakeup on b. So, that is how you basically coordinate between a process who made the request and the disk who completed the request for that particular process alright and finally, if the queue is still not empty which means there are more request to do then you restart the device for the next buffer alright. So, you know details aside I mean let us forget about how exactly the disk is being you know what is the interface you know what does insl mean and what does how does IDE start work these are all sort of very you know too much detail that we do not really need, but what is important is how is synchronization happening right.

So, there is a process who append something to the queue goes to sleep on that on the buffer that he wanted to actually read or write the disk when it goes to sleep it also releases the idelock. The disk when it finishes generates an interrupt the interrupt handler also needs to acquire the idelock because it needs to operate on the it needs to manipulate the idequeue for example, it will move the top of the idequeue to it is next point right.

So, it will operate on the idequeue. So, it is needed to take the lock and then it calls wake up to on any process that was waiting on that idequeue. So, basically wakeup make makes it runnable and so that process can now return from wherever it was good.

(Refer Slide Time: 35:39)

```
3963     panic("iderw: ide disk 1 not present");
3964
3965     acquire(&idelock);
3966
3967     // Append b to idequeue.
3968     b->qnext = 0;
3969     for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
3970         ;
3971     *pp = b;
3972
3973     // Start disk if necessary.
3974     if(idequeue == b)
3975         idestart(b);
3976
3977     // Wait for request to finish.
3978     while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
3979         sleep(b, &idelock);
3980     }
3981     release(&idelock);
3982 }
```

So, what if a disk generates an interrupt while. So, before that what if a disk generates an interrupt while I was somewhere here. So, I have started the disk and before I go to sleep the disk has finish. So, let say the disk is really fast it just finishes immediately alright.

Student: (Refer time: 35:48) the lock. So, interrupt handler cannot be evoked.

Right, so basically because I have acquired the idelock interrupts are disabled at this point this is the great example to understand why we need to disable the interrupts when we are holding a lock right. If we did not disable interrupts the interrupt handler could have run here, and bad things could have happened because I am holding the idelock this entire region the interrupts are disabled.

Interrupts get re-enabled only when you sleep, and you release the lock consequently alright. So, this entire region that interrupts a disabled alright; so, there is no; so, if an interrupts occurs what happens is that the interrupt gets buffered by the hardware and as soon as you re-enable the interrupts the hardware basically gives that interrupt to you.

Student: But that buffer is very small.

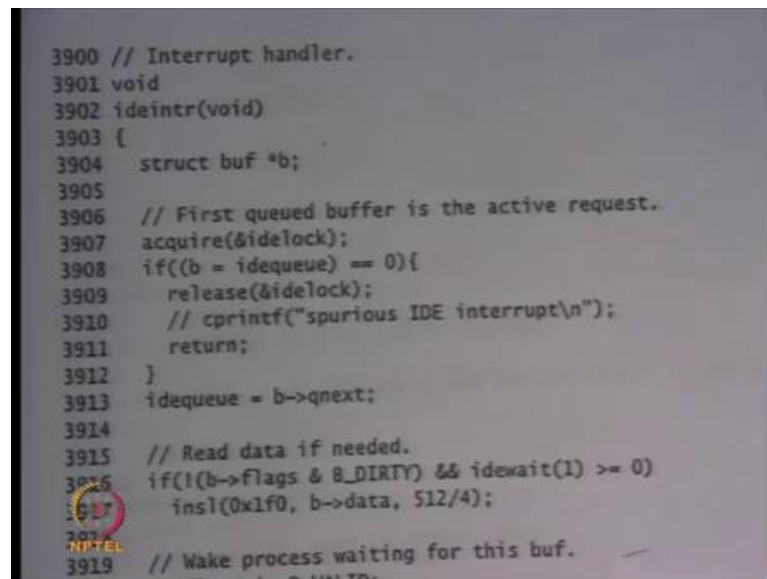
That buffer is very small you know 1 or 2 interrupts it is right. You know you probably worrying about the situation what if the interrupt gets lost and no future interrupt comes. I think we have discuss this before. So, let say you know in usually the protocol is that if a device makes an interrupt it expects for an acknowledgement from the CPU. So, if it

has not receive the acknowledgement, we will retry the interrupt. So, there is you know that kind of a protocol going on alright. So, if an interrupt occurs here no problem.

Student: Ok.

Interrupt will be served only as soon as the idelock gets released and so the interrupts get re-enabled.

(Refer Slide Time: 37:18)

A screenshot of a slide showing C code for an IDE interrupt handler. The code is numbered from 3900 to 3919. It defines a function 'ideintr' that takes a void pointer. Inside, it declares a 'struct buf *b;'. It then enters a loop where it acquires a lock, checks if the queue is empty, and if so, releases the lock and returns. Otherwise, it moves to the next buffer in the queue, reads data if needed, and wakes up the process waiting for the buffer. A watermark 'NPTEL' is visible on the left side of the code block.

```
3900 // Interrupt handler.
3901 void
3902 ideintr(void)
3903 {
3904     struct buf *b;
3905
3906     // First queued buffer is the active request.
3907     acquire(&idelock);
3908     if((b = idequeue) == 0){
3909         release(&idelock);
3910         // cprintf("spurious IDE interrupt\n");
3911         return;
3912     }
3913     idequeue = b->qnext;
3914
3915     // Read data if needed.
3916     if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
3917         insl(0x1f0, b->data, 512/4);
3918     // Wake process waiting for this buf.
3919     waken(b->proc);
```

The IDE interrupt when it gets to run, we will try to acquire the idelock. You can be sure that if it was only one CPU then it will get this IDE idelock right. The only reason that the interrupt handler may not get the idelock is because another CPU has is holding it, but the same CPU could not be holding this idelock because after all the interrupt got to run right.

So, the interrupt got to run that CPU could not have been holding the idelock if it were holding then interrupts would have been disabled. So, that the whole reason the whole the whole fact that the interrupt handler got to run means that the interrupt the CPU was not holding any lock. So, you know you will that basically means that you will eventually get idelock. So, there is no deadlock problem here right.

(Refer Slide Time: 38:07)

```
3911     return;
3912 }
3913 idequeue = b->qnext;
3914
3915 // Read data if needed.
3916 if(!b->flags & B_DIRTY) && idewait(1) >= 0)
3917     insl(0x1f0, b->data, 512/4);
3918
3919 // Wake process waiting for this buf.
3920 b->flags |= B_VALID;
3921 b->flags &= ~B_DIRTY;
3922 wakeup(b);
3923
3924 // Start disk on next buf in queue.
3925 if(idequeue != 0)
3926     idestart(idequeue);
3927
3928 release(&idelock);
3929
3930
3931
3932
```

And, then you will perform this operation and then you will wake up the particular process. When you call wake up that process may not necessarily run immediately; let say it was a uniprocessor system this interrupt handler is running. So, you know how can that process run you have also disabled interrupts completely here, but when you release the idelock and then you return from the interrupt, the interrupts get enabled again and you know because you have woken it up it has become runnable whenever the schedule gets run next it is going to get picked up and it will it can now proceed alright.

So, let say the process wakes up from sleep, it checks the condition again this time it finds it to be true false and no it can go on it is way alright. Notice that here inside the loop when I am saying while some conditions sleep on this. I have not really checked for p dot killed right. I said that usually when you come out of sleep you should now because you are doing this thing that any sleeping process will be made runnable is it incorrect to not check for sleep dot p dot killed right.

Student: (Refer Time: 39:10).


Well I mean it does not matter if it is going to check the condition even if the process was killed you still want that the buffer has been read lock has been released and now you can go on and check later alright. So, it is in this case. So, it is not necessary that every sleep loop needs to be needs to have the killed condition check killed check you

know some need some do not example and one has to reason carefully about what means it and what does not.

Student: Sir, why do not we needed in the space.

(Refer Slide Time: 39:49)

```
968 b->qnext = 0;
969 for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
970     ;
971 *pp = b;
972
973 // Start disk if necessary.
974 if(idequeue == b)
975     idestart(b);
976
977 // Wait for request to finish.
978 while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
979     sleep(b, &idelock);
980 }
981
982 release(&idelock);
983 }
984
985
986
987
988
```



So, one way to think about it is that eventually it is going to come out right. So, it is not it is not a very long time that it is going to it is going to stay there. Now, I am not waiting for some child to exit for example, I am just waiting for the disk to finish alright.

So, it is better to you know let it finish and then release the idelock and then basically let it is caller call. So, make let the things be in a more consistent state you know. I do not know whether the caller is leaving things in an inconsistent state or not. So, you know let it reach some boundary this may not be the right boundary this may be the innermost sort of thing and if I just sort of started returning minus 1 from here that is not that may not be the right thing to do.

So, just an example sometimes you may want to check it sometimes you may not want to check it, but the thing is that within a limited time frame you should basically be exiting that particular process alright. Here is another this example is also another an interesting illustration of why recursive locks are bad idea right.

So, recursive locks are bad idea here why because notice that mutual exclusion is required between the thread and the interrupt handler right and the mutual exclusion is

being done using the idelock instead of you know instead of making idelock like the wait is if I made them made the idelock recursive and the interrupt handler was actually able to get the idelock also then bad things can happen because you know you can get take a lock you could be middle of something and now in the interrupt handler gets called now it takes the idelock and so it and it assume some invariants which are not guarantee to be true right.

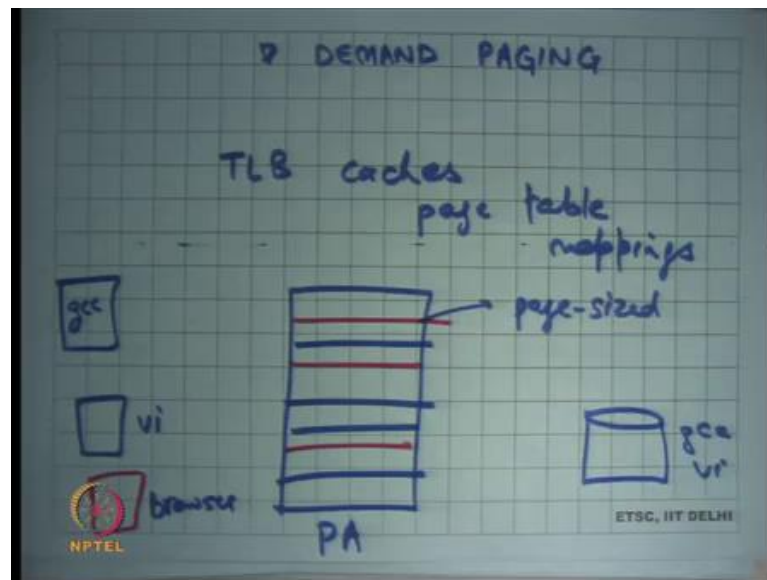
So, recursive locks specially for in presence of interrupts are definitely a bad idea in general also you know it encourage bugs, but if it if the you know you are providing mutual exclusion with respect to interrupts then I then recursive locks are a bad idea alright. Also, because I have you know I am holding a lock I am trying to acquire idelock I need to basically ensure that there is some kind of ordering.

So, any caller if the it is possible that the caller also holding certain locks. So, this idelock needs to be ordered with respect to whatever locks the caller is holding. So, you know one invariant could be that the idelock will always be the innermost lock you will never hold idelock and then try to acquire another lock. So, that way you can prevent deadlocks right.

So, you have to worry about these things also alright because; so, another example of why locking and modularity do not yeah because, yeah you have to worry about ok. So, I am taking idelock here I should not be taking idelock somewhere else where it is possible that you take the idelock and try to take idelock somewhere you know another lock after that good.

So, with that I think you know we have done lot of synchronization and synchronization was you know by far one of one of the most sort of important practical topics in programming in general and in operating systems.

(Refer Slide Time: 43:04)



Now, I am going to talk about demand paging alright. So, we know that we know we have we have looked at virtual memory and we saw virtual memory using segmentation where there was a base and limit we also saw virtual memory using paging and we said you know paging is more flexible. Although paging has more overhead because you have lots of page size you need to maintain a page table and so, but we said then there is a cache called TLB right.

So, there is a TLB that caches page table mappings right and assuming that this cache has a very high hit rate and this cache is very fast by fast I mean it is you know sub nanosecond. So, and the time it takes to actually dereference the TLB is roughly equivalent to the time it takes to dereference that is just right. If it is that fast, then you know paging makes a lot of practical sense

And so, you know hardware developers. So, TLB is actually work in practice because usually programs have a lot of locality lot of spatial and temporal locality which means that the same locations are likely to be accessed over and over again also if you access the location then very likely you are going to access locations close to that. So, because programs exhibit a lot of spatial and temporal locality the hit rates of TLB caches are usually very high and you know on the order of 99 point let say 9 percent or something and so you basically you know do not pay the cost of paging that right.

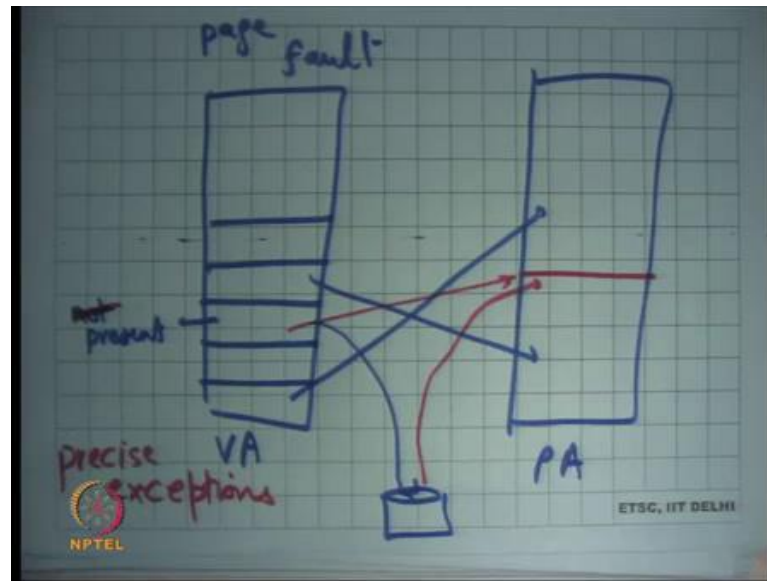
So, the cost of paging was basically this dereference of page table, which is costly so, but that gets eliminated because of TLB cache and so usually. So, so here is how your memory will look like. So, let say this is the physical address space and these are multiple virtual address spaces let me draw this with the different color let say this is gcc and this is let say vi and and so on this is browser then some pages the pages are strewn across like this in the physical address space ok.

And these lines are page sized ok. So, we understand all this. Now in the simple word that we are discuss. So, far we said that whenever you load a program the entire memory of that program the executable and whatever other data it needs is basically loaded from the disk into the physical memory and load time right. So, basically initially the program lives on disk let say gcc and vi including it is code and data and as soon as you load it we say that the entire you know there is some format called a dot out executable.

And, so the a dot out format is fast and the loader basically creates an address space, allocates pages in the physical address space creates an address space and loads the entire contents of the executable into physical memory alright. It is quite possible that you know the process was just started to just stop immediately or it is not going to access all that memory that it actually has in the executable.

So, it is going to access only your fraction of the memory that is going to do it. So, it does not make sense to actually pull all the things at once and so what you can do is you can pull things from disk on demand and that is what is called demand paging right. So, the idea is that you basically at load time you just create an address space alright.

(Refer Slide Time: 47:02)



So, let say this is an address space. This is the virtual address space and let say this is the physical address space and let say this is the disk then you have created the address space. Let say these you know these slots have pages. So, some of these pages are currently mapped inside your physical address space and others are not currently loaded and are actually pointing to the disk at some disk block right.

So, it is not actually a loaded. So, the page is not loaded immediately, but in the page tables you have stored this information that this page corresponds to this particular disk block alright. So, and you say that it is not present alright. So, this particular page is not present. So, the present bit is off in the page table and you have stored this information that this particular page actually lives on the disk at this particular offset.

So, when you run this process you are going to you know if you if you do not touch this page and you exit you know you have saved a lot of work. If you touch this page, then you basically take an exception right. An exception happens because you try to access a page that is not present right and so this exception is called a page fault. If you try to access a virtual address that is not currently mapped in the page table or it is not currently present in the page table, you take an exception that is called a page fault.

The page fault will cause the operating system to run the operating system page fault handler to run on the kernel stack of that particular process and in the previous discussion you were saying that this the page fault handler may want to kill the process

or it may want to send a signal to the process depending on whether the operating system implement signals or not. But, in this case you may want to do one more thing which is to check if this page is actually mapped to a disk location and if so it should not do any of these, it should not kill the process. In fact, you know this is operating system playing tricks under the carpet the process was actually you know doing everything in good intentions if the operating system that is playing tricks under the carpet.

So, what it should do is it should allocate a page here. Let say it allocates a page here loads from the disk block to here marks this present and create some mapping like this alright. So, that is called demand paging and demand paging is a very very sort of useful optimization because you know for the common case you do not need all parts of the executable are not going to get accessed only some parts of it is going to get accessed.

And, so you save a lot of disgreeds alright and also you reduced the pressure on your memory you do not you do not need to allocate that many pages on your memory. So, there is more free memory available on your system right. In general, you know your system may be running only on a small amount of memory let say your system is running on 512 megabytes of memory, yet you can your operating system will allow you to load larger processes.

So, for example, your operating system will allow you to load you know gigabyte size process on a machine of size 512 MB. It is done because of the demand paging running underneath the covers right yeah.

Student: Sir a page fault occurs, and we load the missing page (Refer time: 50:27) into the pageble now we want to re-do the command which caused the page fault how is it.

Yeah, good question we are going to discuss it very soon. So, basically there is some instruction that try to access that particular address and so a page fault occurred. The operating system is going to get to run and it is going to load the page and now you will need to restart that instruction alright and so that is how it is run basically you just restart that instruction.

What you need to make sure is that if you run the instruction twice or you know the first if an instruction causes an exception it does not cause any partial execution of its logic

right. So, either the instruction completes successfully, or it does not do anything at all and causes an exception alright.

So, this has an, this is the property of the hardware or the architecture and this property is called precise exceptions alright. So, if an architecture supports precise exceptions, if there was an exception at an instruction it is safe for the operating system to restart that instruction and basically assume that nothing happened in the previous execution of that instruction alright.

So, we will continue this discussion next lecture.