

Storage Systems
Dr. K. Gopinath
Department of Computer Science and Engineering
Indian Institute of Science, Bangalore

Storage Interfaces and Device Drivers

Lecture - 18

Device Drivers Part 2: Spinlock synchronization mechanism in Device Driver, Linux Concurrency Model, Introduction to Block Devices and the Block layer

Welcome again to the NPTL course on Storage Systems. In the previous class we were starting to look at a device driver, and what is functionality the device driver was to count the number of interrupts. So, we took a look at some aspects of that particular program.

(Refer Slide Time: 00:44)

```
ssize_t ints_read(struct file *file, char *buf, size_t
size, loff_t *poff) {
    int bufsize;
    if(!evtbuf) return -EINVAL;
    if(recording) {
        cli();
        recording=0;
        sti();
    }
    bufsize=MIN(sizeof(struct event_t)*
(nextevt-evtbuf)-*poff, size);
    if(bufsize) {
        if(copy_to_user(buf,((char *)evtbuf
+*poff, bufsize))
return -EFAULT;
    }
    (*poff)+=bufsize;
    return bufsize;
}

ssize_t ints_write(struct file *file, const char *buf,
size_t size, loff_t *poff) {
    char c;
    NPTL char kbuf[32];
    int ret,nrents,ssize;

    if(get_user(c,buf) || size<2)
return -EFAULT;

    switch(c) {
        case 's':
        case 'S':
            ssize=MIN(sizeof(kbuf),size-1);
            if(copy_from_user(kbuf,buf+1,ssize))
return -EFAULT;
            kbuf[ssize]=0;
            ret=sscanf(kbuf,"%d",&nrents);
            if(ret!=1 || !nrents)
return -EINVAL;

            if(recording) {
                cli();
                recording=0;
                sti();
            }
            if(evtbuf) {
                vfree(evtbuf);
                evtbuf=0;
            }
    }
}
```

And for example, we went through the right case, right? In the write case what are we doing? We are essentially determining the size of the buffer that has to be allocated in the kernel right.

We also did some just did some sanity checking out here.

(Refer Slide Time: 01:09)

```
    evtbuf=(struct event_t *)
    vmalloc(nrents*sizeof(struct event_t));
    if(!evtbuf)
        return -ENOMEM;
    nextevt=evtbuf;
    lastevt=evtbuf+nrents;

    cli();
    recording=1;
    sti();
    break;
    default: return -EINVAL;
}

(*poff)+=size;
return size;
}

int ints_open(struct inode *inode, struct file *file)
{
    return 0;
}

int ints_release(struct inode *inode, struct file *file)
{
    return 0;
}

int init_module(void) {
    spin_lock_init(&evtbuf_lk);
    cli();
    penter_irq=enter_irq;
    pleave_irq=leave_irq;
    sti();
    register_chrdev(233, "ints", &ints_fops);
    return 0;
}

void cleanup_module(void) {
    if(recording) {
        cli();
        recording=0;
        sti();
    }

    if(evtbuf) vfree(evtbuf);
    penter_irq=0;
    pleave_irq=0;
    unregister_chrdev(666, "ints");
}
```

Then we allocated it here, right. Then we set the recording on. And then what would happen is that now that the recording is on, every time there is an interrupt, we will enter.

(Refer Slide Time: 01:22)

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/vmalloc.h>
#include <linux/string.h>
#include <asm/uaccess.h>
#include <linux/errno.h>
#include "intevts.h"

struct event_t *evtbuf, *nextevt, *lastevt;
int recording=0;
spinlock_t evtbuf_lk;

extern void (*penter_irq)(int irq, int cpu);
extern void (*pleave_irq)(int irq, int cpu);

ssize_t ints_read(struct file *, char *, size_t, loff_t *);
ssize_t ints_write(struct file *, const char *, size_t, loff_t *);
int ints_open(struct inode *, struct file *);
int ints_release(struct inode *, struct file *);

static struct file_operations ints_fops = {
    .read:    ints_read,
    .write:   ints_write,
    .open:    ints_open,
    .release: ints_release,
};

void enter_irq(int irq, int cpu) {
    int flags;
    spin_lock_irqsave(&evtbuf_lk, flags);
    if(recording && nextevt==lastevt) {
        rdtscll(nextevt->time);
        nextevt->event=
            MKEVENT(irq, E_ENTER);
        nextevt->cpu=cpu;
        nextevt++;
    }
    spin_unlock_irqrestore(&evtbuf_lk, flags);
}

void leave_irq(int irq, int cpu) {
    int flags;
    spin_lock_irqsave(&evtbuf_lk, flags);
    if(recording && nextevt==lastevt) {
        rdtscll(nextevt->time);
        nextevt->event=
            MKEVENT(irq, E_LEAVE);
        nextevt->cpu=cpu;
        nextevt++;
    }
    spin_unlock_irqrestore(&evtbuf_lk, flags);
}
```

We will it on the entry of the interrupt we will do this, at exist we do this, because we have defined penter irq to with this, penter pleave irq to be this, and this is an extern its. So, it is an extern function pointer; that means it is actually linked with this other outside code. Where that is getting called on interrupt right on interrupts these 2 things.

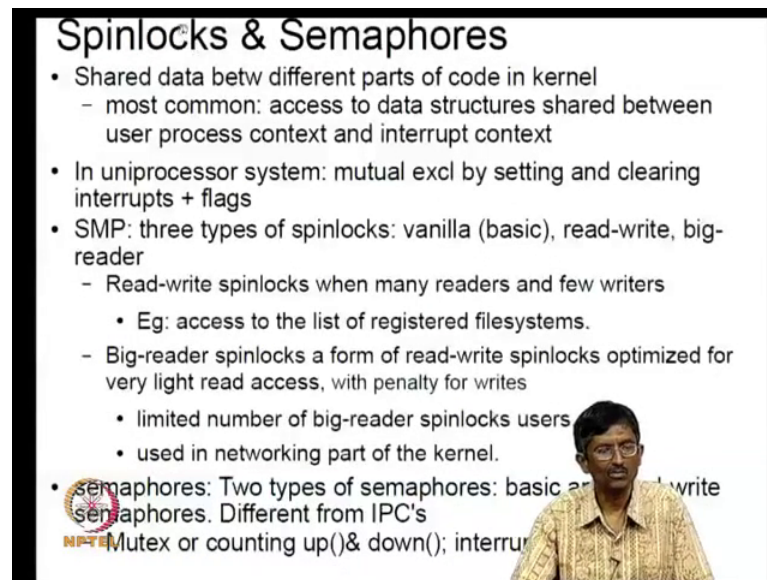
So, I think we looked at some part of it last time. So, we look at this enter irq one more time. But assuming that wherever it is being called this penter irq, it comes with the request routine and the CPU and that basically what is wrote here. And first what you do is please spin lock irq square. Now the thing is where do we do else a spin lock, because normally these things are very fast operations. If you try to put somebody to sleep, it is too costly. Basically, because you have to context switch, it is too costly.

So, for example, it might take a context switch about, possibly a good for good part of a millisecond. Where is the spin lock it is just sitting and executing instructions. Instruction can be about few nanoseconds, 10 tens of nanoseconds. So, if you got able to get a job done quickly, within tens of nanoseconds trying to sleep is going to be about almost 4 5 orders of magnitude more costly. So, I do not want to do sleep if it is possible. So, that is one thing an addition on the interrupt handlers are not supposed to sleep this one, there is no we will come to back.

So, what is this guy doing? It is basically trying to ensure that you save the state of the interrupts, because there could be multiple interrupts, some interrupt may be on some interrupts may be off you want to save the state and that is being done through flags. The flags is the thing that is going to keep track of the save state. And the log that you are taking is this one. You are spinning on this lock. This lock was initialized somewhere here. We remember that when I loaded the module, it was initialize out here.

So, may I think we need to study a bit about the kind of mechanisms are available for mutual exclusion slash synchronization in the kernel. So, we will take a look at that one. I think some of it already we have done, but let us take a look.

(Refer Slide Time: 04:42)



Spinlocks & Semaphores

- Shared data betw different parts of code in kernel
 - most common: access to data structures shared between user process context and interrupt context
- In uniprocessor system: mutual excl by setting and clearing interrupts + flags
- SMP: three types of spinlocks: vanilla (basic), read-write, big-reader
 - Read-write spinlocks when many readers and few writers
 - Eg: access to the list of registered filesystems.
 - Big-reader spinlocks a form of read-write spinlocks optimized for very light read access, with penalty for writes
 - limited number of big-reader spinlocks users
 - used in networking part of the kernel.
- semaphores: Two types of semaphores: basic and read-write semaphores. Different from IPC's
NPTM Mutex or counting up()& down(); interr

So, basically what is the issue? It turns out that there is often shared data between different parts of code in kernel. Some are accessed in the user process context and some in interrupt context. I think it should be clear about what is user process context means, does not mean executing in user space. It means that you could start a user space once you come into the system call, you still enter into the supervisor mode, but you still have the same context, because you got called through a user process.

That means, that you still have essentially the same thread is running starting from your user code what you written, and then you walk into the kernel you are still having the same context in some sense. In some sense if inside the kernel I would say something about that my current process is this, it is still valid. I am talking about something which is out there which is corresponding to the code is I am running inside the kernel also. There is a correspondence between what I am going to do inside the kernel and outside world, because the user process, whereas interrupts context is a different thing.

Basically, what is interrupt context mean? It means that you are executing some piece of code, which has more relation to what was just being executed and it is got interrupted. Because interrupts can come any time, I am doing something and some unrelated interrupt for example, a disk interrupt or a anything can come which is unrelated to my work or a tape interrupt. Let us say I am not dealing with any tapes at all. I can still get a

tape interrupt. So, there is nothing to with my current process, but I said I am doing a user process what I mean is I have a system high initiate system call in user space.

So that means, there is a connection between my user space program, that context from which I came and all through the system call then I am executing inside the kernel. There is a connection. So, what happens is that sometimes there is some parts of cold in the kernel (Refer Time: 06:52) shared, even though one is user process context and interrupt context. A good example is suppose they have a dc driver, and in the user process context we have doing something about allocating buffers; using buffers whatever. So, you might be manipulating some qs. Qs Corresponding to buffers.

Now, interrupt context is also can deal with the same thing. Why because suppose you initiated some activity of reading some block; when it is finishes the interrupt, context will has to say that this particular block was read, now I can insert it into the list of valid buffers. So, what will happen is that both the context that came through the system call inside the kernel, and then at completely autonomous interrupt context, they can refer to the same qs they might actually manipulating the same qs. Because they can be manipulating the same qs, they can be a problem if while the system call is doing something with respect to the q links right, the pointers.

Is interrupt comes in end it might find it in a inconsistent state, because the interpreting come ends at any instruction boundary. Because normally qs have a certain invariant for example, let us say I have a w linked list. So, you might have an invariant about that forward if I go this direction there should be point in the reverse direction. So, I am at a modified only one part, the reverse might not to be modified right. So, the invariant can be spoiled. So, you need to have some way of doing mutual exclusion. So, in uniprocessor system, you can do it by setting and clearing interrupt and flags, there is some way to do it.

In SMP, normally you have many there are many methods of doing it here, we just talk about spin locks. There are 3 types of spin locks. You might have what is called the basic one, read write, spin locks and big reader. So, again lot of this keeps changing with Linux versions; this just a simplified version in the slightly the older version. For example, nowadays you has something called read copy update RCU, which is

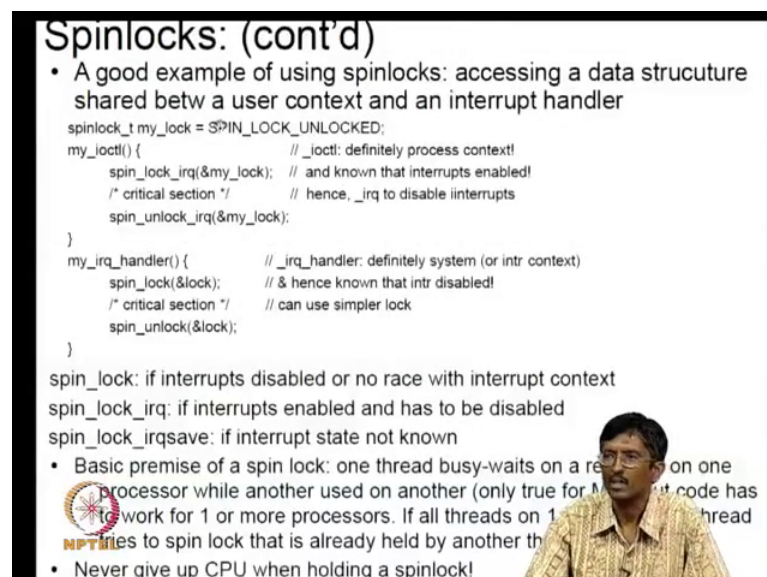
extensively used. So, whatever I am saying is not really widely used currently, but it is used here in there.

So, everything I am saying you should examine it carefully in the context of a current Linux kernels, because things are changing all the time. But anyway, read write locks basically when there are you have many readers and few writers, it turns out that you might use the read write spin locks. The vanilla one actually has no preference further read or write. We have read writes spin locks. Basically, if you have to preponderance, the readers you might want to keep preference to readers than writers.

For examples we have a system with lots of file systems. So, the file system so are in a linked list; and most often times you are walking it to get some information, but you are not modifying a file systems. The time when this information on the link list changes is when you unmount and mount file systems. And that is usually a rare event. So, most assign your link list is not changing; that means, that read operations can be many more compared to write operations.

So, someone this spin locks can be optimized for reads compare writes. Because those writes happen very, very infrequently mounting time unmount and mount a file system for example. So, they are also other thing called big reader spin locks, it is a type of read write spin locks. It is even more optimized for reverse and there is a specific penalty for writes. So, there also some things called semaphores, we will not get into it write now.

(Refer Slide Time: 11:01)



Spinlocks: (cont'd)

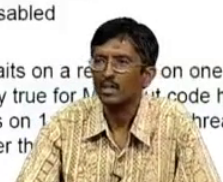
- A good example of using spinlocks: accessing a data structure shared between a user context and an interrupt handler

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
my_ioctl() {
    // _ioctl: definitely process context!
    spin_lock_irq(&my_lock); // and known that interrupts enabled!
    /* critical section */ // hence, _irq to disable interrupts
    spin_unlock_irq(&my_lock);
}
my_irq_handler() {
    // _irq_handler: definitely system (or intr) context
    spin_lock(&lock); // & hence known that intr disabled!
    /* critical section */ // can use simpler lock
    spin_unlock(&lock);
}
```

spin_lock: if interrupts disabled or no race with interrupt context
spin_lock_irq: if interrupts enabled and has to be disabled
spin_lock_irqsave: if interrupt state not known

- Basic premise of a spin lock: one thread busy-waits on a resource on one processor while another used on another (only true for multiprocessor code has to work for 1 or more processors. If all threads on 1 processor, a thread can never give up CPU when holding a spinlock!

NPT



So, let us just look at an example of a spin lock, and you will try to understand how it is used. So, so for example, let us say that I define a spin lock underscore t that is, but the name of the lock is my lock. And I initially initialize it to spin lock unlock, it is unlock.

Now, I say my ioctl. Now what is ioctl? Ioctl is a I O control often provided by device drivers. What is I O control? It is some kind of a way of controlling the behavior of a device or some file system. It is got essentially it is got all the remaining functionality that is not mean only be handled by some other mails typically.

let us say there is some specific functionality read write etcetera, those things are handled through specific interfaces. There could be some other residual functionality which has not been completely characterized or proper it is varying across this lot of variations. So, you do not want to write down to saying the all this variation should be there in my in my list of functional functionalities. I will just put it into the ioctl. Basically, ioctl is a way in which I provide some control operations on this I O or I O devices I O whatever.

So, normally what happens is that I call ioctl sitting from user space. So, there is a similar thing for files, it is called a fnctl. You might have seen it, if you have might have seen it in Posix, this only call fnctl. That is also something which does gives you an ability to control information regarding particular file. For example, you might speci specify that fnctl 3 fnctl, that it has got what is called the there are some file systems allow you to specify what call an extent; that is, a how contiguous allocations you are requiring anytime you allocate anything to the file. Instead of standard 4 kilobytes, let us say I have multimedia file 4 calibrate is too small for multimedia files. It will for in in one-megabyte chunks.

Then you can use an fnctl you make a call using fnctl and say, for this particular file media multimedia file, please ensure that you allocated only in terms of one-megabyte chunks. Similarly, in ioctl also for example, once upon a time you had floppies which had one point 4 some megabytes or 2.88 megabytes, single, double all those things right. So, thing is that you might have you may want to set ioctl saying that now I want you to write it assuming a double density, or single density, whatever it is.

So, I can make a call. Since I am calling from a user space, when I and it turns out to be ioctl will be essentially it will come here to a system call. There is ioctl call, system call

and once it comes it comes to appropriate driver routine. Now essentially from user space a came through system call, and now I am in my driver part by ioctl. So, there is where I am definitely in a process context, because I came through a system call. I still have connection with the party who initiated it. And now I since I am coming from cross context, it is known that interrupts are enabled. Because this is (Refer Time: 14:46) cross context it means it is not absolutely critical a user stuff can be always be interrupted.

Therefore, he is going to say irq to tell it please disable interrupts, this driver space spin lock irq. There other varieties like spin lock if I know that interrupts already disabled, or no race with interrupt context; that means, that I am doing something, and whatever I am doing none of interrupt handlers ever touch. So, was I am able to figure it out. Then that can I can use this more simpler surplus spin lock.

So now that you know interruption an able to have to say spin lock irq, and then I do whatever I have to do so and then I spin unlock irq. So, basically this has interrupts get disabled for me here at this point. So, I am in a state situation there is no question of these conditions (Refer Time: 15:44) whereas, suppose I have a my irq handler, this gives an interrupt handler. There is there is a disk block which got completed, and there is a call back after the disk block has been read. Now the handler is basically this something similar to this. And this came in once sometime in the pass somebody did something. And now that context is different from the context that is the interrupt handler, because this interrupt handler is happening only because an interrupt. This one has no connection with what is currently executing.

So, it is in the interrupt context and because in interrupt context it is not it is known that interrupts are disabled. So, for that reason you may it can take spin lock without having to do this part. Sometimes it turns out that you are coming from multiple levels of calls, the multiple subsystems, and you might not know exactly whether interrupts are disabled or enabled. If you are in that unknowing state not knowing what is going on. You can just say irq save it will figure out. It will actually make it. So, that you do not have to worry about the state to be received everything will be fine, and then you do the reverse part of when you exit.

So, what is the basic permission of a spin lock? There are some interesting things that have to worry about in spin locks. I have mentioned already your spinning to get access

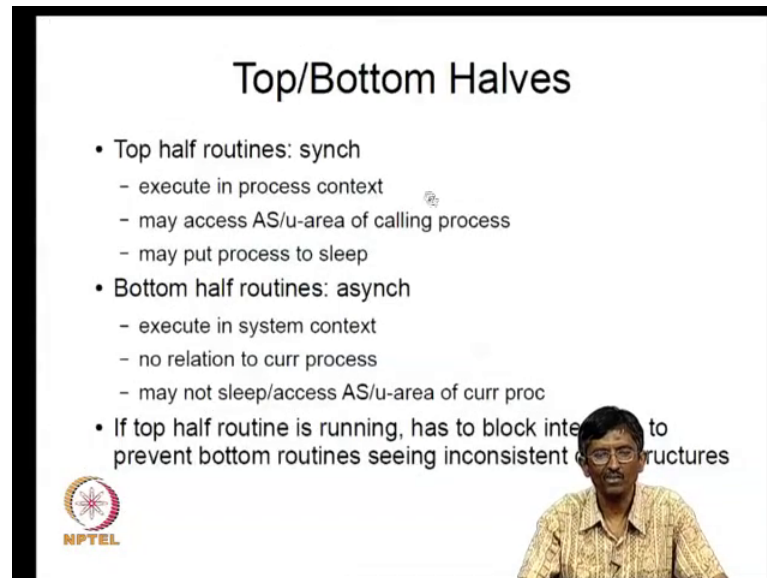
to resource. If somebody has locked it, you expect the other party will unlock it, since you are keeping on checking whether lock is available or not. When that party unlocks it you will get access to it also. There will be a contention multiple parties can try to get this at the same time somebody (Refer Time: 17:45), but you will notice that there was some additional thing that you have to be careful about my programming these things in the kernel.

One thread, basically busy waits on a resource on one processor while another used on another. This is true for multiprocessor. But notice that your code even though may write it for multiprocessor. It should run even on a single processor, where n equal to 1. If you are if you are saying that this particular thing only runs for n equal 2, it never will touch this kernel there is not something acceptable. Ice power most specifications, whatever could you run it can run on a multiprocessor, it should also run on a single processor.

Now, suppose if code has to work for example, of all threads on one processor. Let us say, that I have allocate all the threads from on one processor. And then if a thread tries to spin lock that is already held by another thread (Refer Time: 18:48) because the single processor, we find spinning let me other guys are not on the CPU the spinning. Other guys are there other guy can cannot can never get access to the CPU, because hand spinning continuously therefore, the other guy who has a lock can never be woken up and we cannot give access to the lock with this guy spinning on therefore, the divided lock.



So, why important thing is that we should never give CPU, when holding a spin lock because I absolutely I said rule; that means, that whenever you have a spin lock in your hands never give for CPU if you do that your system can handled lock. So, these are some other interesting things that you have to know when you are programming in the kernel.

(Refer Slide Time: 19:34)



Top/Bottom Halves

- Top half routines: synch
 - execute in process context
 - may access AS/u-area of calling process
 - may put process to sleep
- Bottom half routines: asynch
 - execute in system context
 - no relation to curr process
 - may not sleep/access AS/u-area of curr proc
- If top half routine is running, has to block interrupts to prevent bottom routines seeing inconsistent data structures

Now, similar to that let us also talk a bit about top and bottom halves. So, some of you might already hold of it, what is the top half? Executing process context, the same thing; for example, this is an example of a top half. You are coming from the process context you are in the kernel that is a top half. And because you are connected with the system call whatever you are execute something in the kernel you are still connected with the (Refer Time: 20:00) you can access the address space or in older units is called u area of the calling process; that means, you still have access to the context or the process that made the system call, you can still refer to it.

For example, you can look at how long it has been executing all those kind of you temperate. There are some there is some information about the time taken etcetera, or the resources you have to work and all those things you can normally there is a resource structure it tells you, how much you can use how many files can be open and all those kind of things. Now inside this top of routine you can actually look onto the all those things. They make sense because you are executing on behalf of process, which us called a system call and you are connected with it. And you can also put the process to sleep.

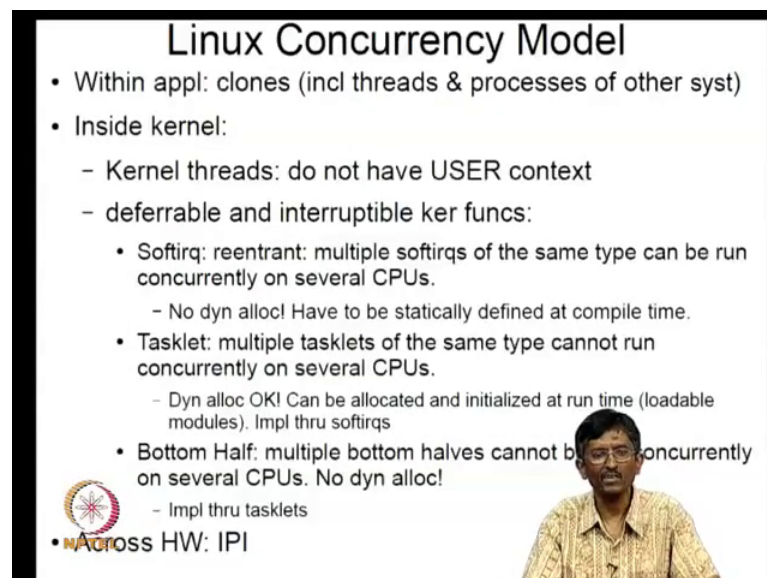
Basically, because essentially, you are putting that particular process, which worked in through system call to sleep is perfectly fine there is nothing upsetting about it. Whereas, use a bottom of routines these are asynchronous. These are call because of things like device completions some read etcetera devices can a completed, they called

asynchronous sleep. They executed system context, and there is no relation to the current process. Because there was a current process, and because of their interrupt that interrupt handled started running, and then it called the bottom of routine. And the thing is this bottom halves it has nothing to do with the process that was executing there was being there was interrupted.

Therefore, you cannot access the address space or rear of the current process. Whatever is called a current process it is not possible. It may not sleep also, because either you can get in deadlock or you are creating delays you are creating (Refer Time: 21:44) if there is multiple stack of interrupts, if you sleep then other guys they can get all stuck till the topmost level interrupt can find that also is possible.

So, nothing is if top of routine is running has to block interrupts to prevent bottom routines, seeing inconsistent data structures. I already discussed this basically if top of routine is running, right. Basically, that guy could be manipulate if similar structures. It might have just done some part of the thing, when it is thing there is a inconsistent, on this interrupts comes in on the bottom of another ones and that would see the inconsistent part you have to block it.

(Refer Slide Time: 22:28)



Linux Concurrency Model

- Within appl: clones (incl threads & processes of other syst)
- Inside kernel:
 - Kernel threads: do not have USER context
 - deferrable and interruptible ker funcs:
 - Softirq: reentrant: multiple softirqs of the same type can be run concurrently on several CPUs.
 - No dyn alloc! Have to be statically defined at compile time.
 - Tasklet: multiple tasklets of the same type cannot run concurrently on several CPUs.
 - Dyn alloc OK! Can be allocated and initialized at run time (loadable modules). Impl thru softirqs
 - Bottom Half: multiple bottom halves cannot be concurrently on several CPUs. No dyn alloc!
 - Impl thru tasklets
- Across HW: IPI

WACUP

WACUP

Now, this is a way most unique us is used to be in the early 80's, you will see that Linux has a slightly different model it is slightly adds to certain additional things inside. We will skip this part let us look at the kernel part of it. Now the kernel threads are those

which do not have user context. These are things or run independent of the user for example, you have something called housekeeping activities. There is some stuff which is kernel memory which is cached which is caching some stuff in that disk and you kept it cached, and it is not yet returned to disk.

Now, you can have what is called a thread a kernel a kernel daemon which will once in a while wake up. And ensure that part of it is flushed to disk. Because if you do not flush it regular times it power fuels whatever happens? You might lose whatever that is there in the kernel that dirty buffers that are there, you may not be able to flush into disk. Because of power fuels and you might have the disk might not have the current copy of what was supposed to be there can be we will do some banking transaction. And you updated a thing suppose you are sitting in memory and power fuels. Essentially it is equal into your bank transaction not going through.

So, there is daemon which keeps firing in, and this independent of any user process context. Because it doing for everybody. In sense as you can call it house maintenance operations housekeeping operations and these are done for every process. So, it is nothing not done to any particular process that is why do not have user context, there also what are called deferrable and interruptible kernel functions. And bottom half art I talked about previously is one type of it. This is how Linux also started with, but then they added these things.

Now what is the bottom half in the Linux, within how we started out? We can have basically multiple bottom halves cannot be run concurrently on several CPU's, there has to be only one thing at a time that was all big problem. Another thing is that the bottom half was particular subsystem for example, a disk will have one particular bottom half network in subsystem will have one particular bottom half (Refer Time: 24:38) tty will have one etcetera. So, this is all fixed also that is you had fixed number of bottom halves. So, because basically depending on the type of devices, that were there that is how it started out. That is how there is no dynamic allocation. And you cannot run or you can only run one thing at a time it is extremely restrictive.

Then you came with something called tasklets, multiple tasklets same type cannot run concurrently on several CPUs. For example, I cannot have the same disk tasklets running on the same things. I can have a disk running on one CPU and the tape running on

something else on a I can have some other, let us say the network is stop running some other thing because they usually or 3 different subsystems they usually do not interact usually. So, that by definition they do not have any things to they do not club or each others stuff. They can be (Refer Time: 25:34) and it turns out that you can have dynamic allocated, and they can be this slightly more flexible kernel model. And again, I have not go into too much detail about this one.

The soft irqs are the most general ones. You can have the same type running on different CPU for example, I can have I have let us say 700 disks on my system. I can have I want to run all of those things there are they are doing all these operations at the same time. All right all this disk running at the same time. So, a few number of them can be finished some operational disk around the same time. I want to do the interrupt crossing about the fact that difference reading something or writing something. It will be very restrictive for me given that I am somewhere in a disks right, to say that we are (Refer Time: 26:27) in sequence, one after another even there are so many CPUs right.

So, these things are basically reentry; that means that they ensure that the way the code is written itself. They ensure that they take all the locks to ensure that even if the same type same code is running. And so, many CPUs because the way that take the locks, they are able to keep the data structures completely let us say consistent across each of these parties were running at the same time. So, because it is the most general part it turns out that they cannot be dynamic and allocated ok.

Again, we have to going to be data that understand why that is the case. I think we can I am not going to go into details here. So, this is the kernel model in Linux. So, I hope I given you some free for this part of it is spin lock irq save etcetera. And so, what is happening on entering? I checking if it is recording. If it is recording you do not know I will again I am not going to do it. And I am also checking next event is not equal to last event. Basically, what is a what is happening? I have a set of buffers right. I did the first one second one I came to the last one. I am trying to seek my pointer e said the last one.

If I already come to the last one; that means, I do not have anywhere to write (Refer Time: 27:57) that is not the case. Then I am saying read time stamp call. And then I am writing it into this particular variable. I think I discussed it last time this is the read this is the assembly language code that does it. And then I am creating an event. The simple

straightforward thing it is basically it is creating some kind data structures nothing more than that. And then I am saying I am ready for the next event I have set the pointer to the next place in the buffer, I have to do that this is basically implementing with to the next location about.

And then once I am done with this, I am going to play this. Again, a music and say `irqsave` because I do not know what state I could be in some this making sure that I save this state. This is an could be that there are so many interrupt possible, because I can have stacking of interrupts. And I am not very clear about exactly what state I am in current at any point in time therefore, I am doing this.

Again, living also is something similar, I have basically get a lock, and then I have basically record that event, I am also going to say that this is going to be the living event here. I said that it is a entering event this is going to be a living event, and then I increment to the next buffer allocation. I am also recording for example, CPU data very straightforward, and then once I am done with I exit.

So, what have we done right now? You first in this code in the right code basically you allocated buffers, you are start the recording on, and then this interrupts are happening, and then this read times stag called this assembly language code started recording, which interact with CPU what time all those things right. It keeps doing it till the whole buffer is form I give it a certain buffer size right this much and then it completely has (Refer Time: 30:10). Once it comes to the end of the buffer then it stops.

So, that is what is going to happen. So, whens it is completed, it stops and essentially what is happening is that even if the interacts happen, you still getting into this code, but you are not doing it. It is a slightly when one can imagine writing better piece of code, as soon as you let us say come to end right, you may want to not do all this `irq save` all the kind of stuff all right. You may want to do it make it a null function, it is possible right. That is something you may want to think about. So, but is just doing nothing here it is just getting this thing and coming out. That it is sitting there out of there.

So now, the trays of all the interrupts I have been captured sitting in kernel memory, I gets actually virtual the pageable kernel memory as for as sitting there, till you somebody tries to read it. What is it read? Read a out here. How is the read being done? Read is being done through this user code.

(Refer Slide Time: 31:23)

User code

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <assert.h>
#include <stdlib.h>
#include "intevts.h"

#define rdtscll(val) \
    __asm__ __volatile__ ("rdtsc": "=A" (val))
#define STACKNR 32

int fd1,fd2;
int cpustack[2][STACKNR],stktop[2]={-1,-1};
unsigned long long cpustart[2];

void push(int cpu,int val) {
    assert(stktop[cpu]<STACKNR-1);
    cpustack[cpu][++stktop[cpu]]=val;
}

int peek(int cpu) {
    assert(stktop[cpu]>=0);
    return cpustack[cpu][stktop[cpu]-];
}

int peek(int cpu) {
    assert(stktop[cpu]>=0);
    return cpustack[cpu][stktop[cpu]-];
}

int peek(int cpu) {
    assert(stktop[cpu]>=0);
    return cpustack[cpu][stktop[cpu]-];
}

void die(char *func) {
    perror(func);
    exit -1;
}

void closefiles(void) {
    close(fd1);
    close(fd2);
}

int initfiles(void) {
    fd1=open("out.cpu1",O_CREAT|O_WRONLY,0666);
    if(fd1<0) die("open");

    fd2=open("out.cpu2",O_CREAT|O_WRONLY,0666);
    if(fd2<0) die("open");
}
```

So, what is happening out here? You have this is the main of the user code. This is a regular user space code.

(Refer Slide Time: 31:31)

```
void output(int cpu,long long x, int y) {
    char buffer[32];
    sprintf(buffer,"%lld %d\n",x,y+(cpu?0:20));
    if(cpu) write(fd1,buffer,strlen(buffer));
    else write(fd2,buffer,strlen(buffer));
}

main() {
    int ret,fd=0;
    struct event_t e;
    if(!initfiles()) return -1;

    push(0,-1);
    push(1,-1);
    while((ret=read(fd,&e,sizeof(e)))==sizeof(e)) {
        printf("Event record:%lld,%d - [%s,%d]\n",
            e.time, e.cpu,
            EVTYP(e.event)==0 ? "E_ENTER":"E_LEAVE",
            EVTNR(e.event));
        assert(e.cpu==1||e.cpu==0);
    }

    if(cpustart[e.cpu]) {
        cpustart[e.cpu]=e.time;
        e.time=0;
    }
    else { e.time=e.time-cpustart[e.cpu];
        e.time=e.time*1000*1000/
            (1263*1000*1000);
    }
    if(EVTYP(e.event)==E_ENTER) {
        output(e.cpu,e.time,peek(e.cpu));
        output(e.cpu,e.time,EVTNR(e.event));
        push(e.cpu,EVTNR(e.event));
    }
    else if(EVTYP(e.event)==E_LEAVE) {
        assert(EVTNR(e.event)==peek(e.cpu));
        output(e.cpu,e.time,peek(e.cpu));
        pop(e.cpu);
        output(e.cpu,e.time,peek(e.cpu));
    }
    else assert(0);
}

closefiles();
}
```

So, I am going to say read fd etcetera. So, fd is 0; that means, that you are doing some redirections. Basically, what you are doing is you are going to say that this particular program is going to have this less than followed by slash temp ints; that means, that it is slash temp ints will have fd of 0. That is how when you say read fd etcetera, it is going to the read of this device which I have created; that means, that it is going to call this. You

will see that there is you can see that the numbers of parameters are different. Here is the regular read call as you are used to you got 3 parameters, fd; it has got the size. And it is got the address right.

So, you are supposed to fill in that stuff here all right. It is supposed to read something and put it here right. Now whereas, here we can see, it has got how many parameters. 1 parameter, 2 parameters, 3 parameter, then 4 parameters; that means, that there is something in between this system call I am where the driver gets it; so like z or some other intermediate thing will be sitting there and doing it.

Why should at their case? Please you can notice that there is an offset also. Remember that in the case of read the offset is simply sit, where I am reading it. Because I am keeping on adding to and keeping on incrementing the implicit offset, every time I do a read write that is going on. So, I do not mention that pointer. It is implicit in the it is kept in the file descriptor somewhere that information is kept. And some piece of code is coming in between that call and here which is actually putting that is all it is offset here.

So, that is one part of it and also you can see it started with file descriptor there it has been converted into a struct file star. While this is happening, you do not have worry about it there is solid infrastructure in the systems can taken over this. So, what is it doing now. So, there is no reading stuff. So, what is it saying? It saying read from slash tempt ints, and read about what is size of e what is he? Is an event; so I wanted to read as many bytes as the size of the event? I think if you note if you notice what are you writing when we what is the information of collecting for an event, that I think we have seen already. Where is that event? This is the information, or the if you look at the event of the data structure. It will have all that that piece of information.

So, that is what is supposed to read. Again, it is being some sanity checking out here, if not event buffer it should not happen, but by share bad luck for some reason, who knows there could be a bit flip as I told you. You can your memory can lie your disk can lie all kinds a bus can lie, you could what I mean by saying that some arose can take place on system. So, if you are good OS kernel hacker, you never trust whatever you see. You check everything for a good here. What you are doing is just checking, if you can go for this not somebodys asked to read some stuff, you are worrying whether is there any buffer at all in the first place, that is what we worrying about.

Most likely all these things usually we are it should the case at the event buffer. So, you should never get into the situation, but if it happens you report it. Again, if you if recording. So, by if some recording is still going on. So, I want to make sure that it cannot record it. Even this it could be that the right as soon it the interrupt routine right. The minute it came to the end of it right. You should probably start recording equal to 0, but it is not done it here. One can add that code. So, that recording becomes 0. So, it is being done here.

So, again you are taking this cli sti as I mentioned these things are no longer available as far as I know in the current linuxs you have to do something else here. And what you are doing right now is; you are going to you are now supposed to read from the event buffer, and copy into user buffer user buffer. Is given by here and e that event e this is a user space you given to the address and you have to copied from the kernel buffer right into this one.

As I mentioned to you, you have to be careful about copying stuff from in and out of kernel that is why you are doing what is called copy to user. And before that of course, you have to do something about you finding the minimum size. And you basically you are trying to read some number of bytes and it is trying to figure out what is actually available and that is what is a big part. And then it is going to copy from user to the event from the event buffer to the user buffer, and it is going through instead of being a straight then copy, it is going to copy to user and the copy to user will carefully check everything whether the user buffer. So, actually legitimate buffer, is it is not a booby-trapped buffer which when you touch it explodes.

Why is that the case? Because I say no I said best as we discussed last time also. If there is anything that a trap happens the kernel result to pick it up, right. The kernel itself actually well it is touching something it itself gets a trap, there is nobody to look after it. Because the in some sense you know you can not keep on putting one more level kernel 0 kernel one kernel 2 see everybody watching for everybody else that does not work out. So, the thing is that is say and why has to be careful that is why you use copy to user. It is ensuring that the buffer that user has given where the information has the event has to be copied into that is legitimate; and as I mention again there implicit pointer that has to be updated. And then we do a read and write once you read so many bytes are right so many bytes implicit pointer changes on that is continuous.

And you always when you do a read we always return whatever you read the size of it. Because you remember the int there is usually the value that returned after read is int that is basically what it is. So, these all the read happens. So, what we have done right now is; I keep reading one event at a time exactly one event write at a time. Once a read one event, then I am going to let us say, populate my data structure e time and e CPU. Event dot time event dot CPU, and then I am checking whether it is entry into or the interrupt or the exit of the interrupt and the number also event number.

And I am also doing some sanity checks for example; it should not be because I am only in this particular example. There only a 2 CPUs it is checking whether 2 CPU server. This is some straightforward tracking of the time. In case I was just starting about when I am just starting about, it tries to if it is 0 then I am starting. And so, is taking care of that part. Once I have already looked at the first event, then this is the code. If you look at this some there is some peculiar thing here I would like you to look at it closely. This 1263 is coming because if I remember right, my student was looking at 1.263 gigahertz machine in those days.

So, the 1263 is coming from the 1263 megahertz that is what is coming through. Basically, these things are clock ticks. And if you really want time, you have to look at the clock frequency and convert it into microseconds or milliseconds. So, that is what is going on. Again, he is checking the sanity of the whole thing, he is basically if it is enter he is pushing it onto a stack, and then it is to the corresponding exit also and this should I will match. And so, that is what basically is doing. If some of these things do not work then mail is something went wrong. So, is that is this stuff has already been captured is checking the sanity of or what if already been captured.

Basically, then interrupt should nest properly. The nesting is what is doing. That is why steps some stacks is got some 2 CPU stacks, and this is this stack cannot is the number of interrupt levels. So, that is what is doing whatever. So, I think this more or less and then finally, he is outputting it into some files. Basically, he has got 2 files, out dot CPU one these are all the interrupts. All the trace of all the interrupts on CPU 1, this is the trace of all the interrupts at were handled by CPU 2. So, he is basically writing to all those things, depending on which CPU it is right into fd 1 or fd 2, all right.

So, this is roughly the code corresponding to a device driver that is counting interrupts. The one thing which I have not covered is what happens when you want to unload this module. This is what will get executed again if it is still recording for whatever reasons you said recording is equal to 0. You free the event buffer because it you allocated kernel virtual memory, and this kernel is running all the time. Any time you do not free then basically something like you lose that virtual address space. So, you have to free it is what is called soft might how what a called memory leaks. If you do not do it then the kernel has memory leaks one did the kernel will choke up and die.

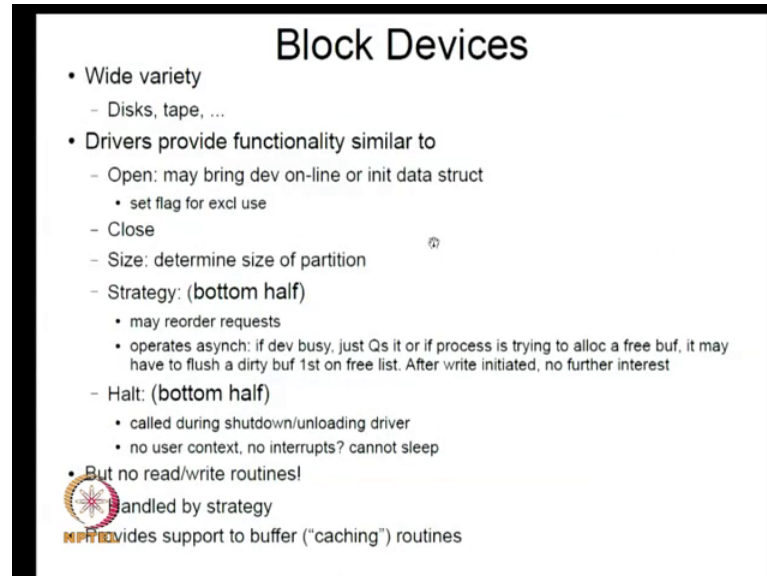
Basically, because the even though it has got gigabytes of virtual memory, you are running so many of these things and a kernel is supposed to run for days and days together. So, every time we are lose few megabytes, if you just do it enough times then you finally, will exhaust address space you are dead. And you also do not want to call this again because remember this p enter irq if it is 0 means you are not calling any of my driver. If it is set to these values all right enter irq or leave irq here right, where is it may be sit it oh, it is here see only because I said this is that I am getting controlled into these 2.

So, I am going to set to 0 saying that please do not call my driver at this point. And then I am also unregister in, I am saying that this device I know I want to release the major number. So, anybody can use it this seems to a bug here this 233 and 6 6 6 I do not know what the stories, so this roughly the whole of this device driver. So now this is an example of what is called typically called a character device driver. It is not based on blocks it there is no boundaries you can did here. There is events or the some kind of boundaries you might calling, but this is not something hard and (Refer Time: 43:53) up it turns out to be the let us say the data structure, the event has been defined to me.

So, that there is some kind of structure to it, but these are not at the level of what you normally considered blocks. And there is no caching going on typically in in block devices, there is some kind of caching going on. So, here this would be typically considered a character device driver. Of course, this distinction between what is called character device driver and block device driver has been now always even a fuzzy thing. So, you do not have to give it anything you can just skip it, but ok.

So, we will move on to the next part which is I will just briefly discuss block devices, we will go into more detail later.

(Refer Slide Time: 44:50)



Block Devices

- Wide variety
 - Disks, tape, ...
- Drivers provide functionality similar to
 - Open: may bring dev on-line or init data struct
 - set flag for excl use
 - Close
 - Size: determine size of partition
 - Strategy: (bottom half)
 - may reorder requests
 - operates asynch: if dev busy, just Qs it or if process is trying to alloc a free buf, it may have to flush a dirty buf 1st on free list. After write initiated, no further interest
 - Halt: (bottom half)
 - called during shutdown/unloading driver
 - no user context, no interrupts? cannot sleep
- But no read/write routines!
 - handled by strategy
 - provides support to buffer ("caching") routines

So, what a block devices? The good typical good examples are things like disk tape CD ROM's all those kind of things. Now you notice that in the previous case the character devices provided certain functionality. You had things like what are the functionalities, we had open close they are read write open. At least often in other unix it will call closely these in the Linux it call release. So, you have read right open etcetera. Whereas, see we will notice this is the kind of functionality. Open, close possibly size strategy, hard usually do not have read write routines.

So, this one major difference we just go through it open, what is it is basically can bring the device online or initialize some data structures correspond to device. It may also set the flag for exclusive use. For example, it take device only one user is using it. Because it is you can have the rewind operations going on right. So, you want to ensure that only one party says if 2 are 2 parties, comments a rewind it is going to be big close. So, you may want to set it by closing this also. You might have a close let me said done with the device. You can size you may want to look at the device and say what is the size of the partition and those kind formation, how many partitions in their etcetera. You can have at called a strategy.

What is the strategy? Basically, if you want do any reading and writing you go through strategy, and basically this bottom half, what it means is that this is not being initiated because of your system calls. This being this is being done as part of other activities, basically what is happening is the device is certain q, you q the requests to them. And then they are these requests are managed through strategy. Of course, the word strategy is not used in Linux, but it is the world uniques name for it.

Basically, the reason why you have a strategy is because the minute to give a request. It may not be right and to do the request immediately. Why is that? Because as I mentioned, in block devices especially disks you want to do what is called clustering. You want to get instead of doing the request immediately as soon as you somebody gives a request, you want to wait for few more requests. And hopefully you have instead of just doing a 4 kilobyte or one kilobyte request. We want to do some 100s of kilobytes is possible. You want to cluster them.

That is even why the guys make a request and they disappear. The user process context walks into the kernel makes a request and the runs away, goes away. And the strategy routine is a guy which wakes up and sees that it does all this clustering etcetera, and tells it is now I have seen enough stuff. Now I can use my disk to some reasonable efficiency level.

Now, I will do it. So, none of the users are doing it parse. It is being done by the strategy routine that is why it is a bottom of routine. It has no context it is no it does no user context. And because it has got so many requests, it can reorder requests. Because on a disk if you go all over the place randomly. So, I am going to be a costly. If it is possible for you to go in up a more disciplined fashion through the disk.

So, that you go through in a certain order the request, that I that you have seen it might make sense of course, the reordering requires can create problems for the application, in case they have to be done with particular order. And in that case, you have to do what is called you might want to insert what you called barriers. So, the barrier will essentially ensure that you force, the system to completely finish certain operation for you take the next ones. But in general, the idea here is you are doing a request most of the unix kind of programs or synchronously written programs.

What it means is that; you are not really doing is synchronous I O; that means, you write and then you wait; that means, this nothing this is going on right. Typically, a single thread programs single threaded only when you have multi-threaded programs then you have a problem. So, your single thread programs, it make a request, your process is struck anywhere. And then there are other parties in the system, they do not usually have anything to do what I am doing. They also make requests. So, another thing is I can easily take your requests and reorder them without any worry, because they are all unrelated requests. They coming from different people they are doing probably different things. And so, I can reorder them without too much trouble.

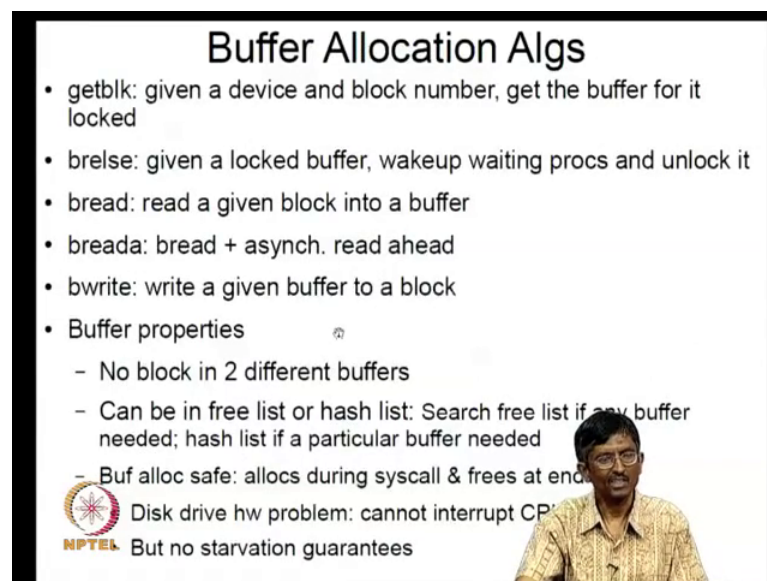
Of course, if have a single program with multi-threading. Like java, for example, or using Posix threads whatever. Then they might be some requirements for different threads might make requests and they might have to follow certain order to get certain guarantees with respect to what really happens you want to do it consistently what gets updated to disk in that case you are do something special. So, the you not say that this particular design is for the old Unix in the 1970's Unix where synchronous I O was more or less standard. No threading was there. And multi programming was doing a lot of people are sharing the same system.

In that case ordering requests is perfectly fine. And this operates asynchronously. Synchronous means with respect to the process who initiate the request (Refer Time: 50:41) this is no connection with that particular ordering. So, when somebody wants to start a new request, if the device is busy you just q it. Or it could also be the case that there is a slightly different situation. It may be that you need a free buffer, we will come to that soon there are some buffer caching routines are there. And you need to allocate a buffer for the one particular block. And if it is not freely available, what you might want to do is you might want to flush a dirty block to make that buff one buffer available.

So, in a sense what is happening is that you are doing a right not because user wanted it. You are doing a right because, somebody wanted just a free buffer, but you are doing a write unrelated write, which is nothing to with my request so that I gives a free buffer. So, the study does all these kind of things, that is why it is a bottom half it is nothing to with any process context. So now, we have a halt and this basically done not because a user it is done usually because of some system requirement.

So, usually these block devices, they provide support to the buffer caching routines. Few with block devices typically have buffering going on typically. There is caching extensive caching going on reason why of course, is typically block devices have been slow. In the tapes and disk etcetera, because they have been slow, the caching has been a central part of their design it shows very big (Refer Time: 52:22); that means, that block devices the device drivers have to essentially provide support to buffer caching routines. That is something they have to do that is a critical part of it.


(Refer Slide Time: 52:24)



Buffer Allocation Algs

- getblk: given a device and block number, get the buffer for it locked
- brelse: given a locked buffer, wakeup waiting procs and unlock it
- bread: read a given block into a buffer
- breada: bread + asynch. read ahead
- bwrite: write a given buffer to a block
- Buffer properties
 - No block in 2 different buffers
 - Can be in free list or hash list: Search free list if any buffer needed; hash list if a particular buffer needed
 - Buf alloc safe: allocs during syscall & frees at end

Disk drive hw problem: cannot interrupt CPU
NPTEL But no starvation guarantees



So, I just mentioned some of these things quickly you can have things like buffer allocation algorithms, get block. You can have the buffer release. You can have buffer read buffer read synchronous buffer write etcetera. Basically, it is because the caching, what is the thing that you want to guarantee is a following; there is a block, it has to be there is exactly in one single buffer, it can not make 2 different buffers. Different 2 buffers the inconsistency can raise: that is why there is a strict requirement. And then it can be in the free list or hash list.

So, now the interesting thing about the system is that; suppose I have a file, which I am reading. I got it to the cache. I did not modify it. It is quite possible that somebody else some other user you also may also want the same file. So, even than done with my use, I can since it is cached I can make it available to the other second user much more rapidly much more less late and see further processor. So, the thing is what happens is that I get

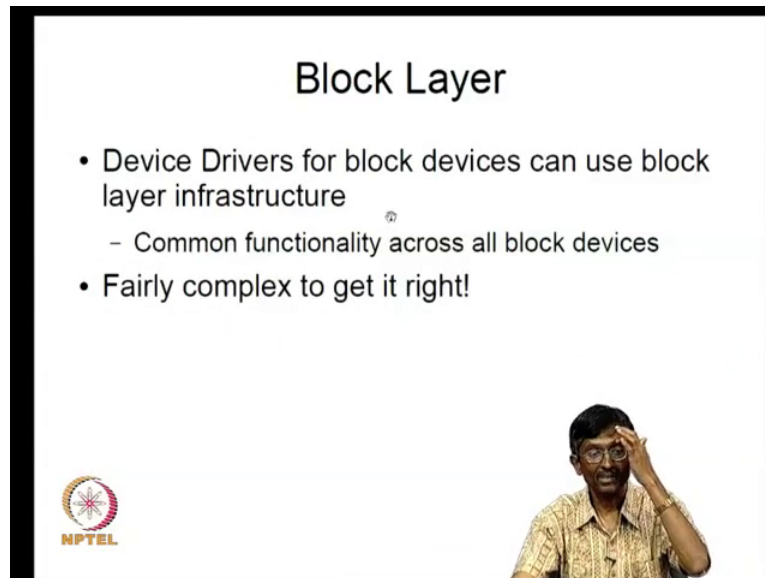
some stuff and done with it, since I am done with it I can put it on a what is called a free list, but it is still valid content.

If by share chance somebody else comes along and wants the same stuff. And look at the free list if it is there I will give it him directly, because nothing has been modified. So, the system essentially what it is doing is that, if first part it takes the payment getting it. The other parties can free ride on it. So, the idea basically is that you keep a free list, and the manager feel it is in such a way that, you always pull out from the free list, which either totally junk value you observe more content what is over, or does not been used for quite some time.

So, what I will do is I have a free list, and what I will do is if somebody got some file into the system, some block into the system, and then I am done with my use I will put it on the end of the list. And somebody wants a free block sorry, free buffer; I pull it to the front; that means that my effort is bringing it from a slow device into the kernel memory, since it is sitting at the end of the buffer guest, right? Till I am completely done if all these things, I do not have to be it will not be touched. So, there is chance that you can get reused that is apply.

That is why it can be in free list or in hash list. Hash list means I am still using it in some sense free list means I used it and I am done with it typically. So, typically all this sub you will notice that this buffer allocation algorithms they usually call this I can bread for example, we will call a make a call to the read of the block device. And write will also do a right on the block device. So, these caching routines depend upon this block (Refer Time: 55:58).

(Refer Slide Time: 55:59)



The slide is titled "Block Layer" and contains the following text:

- Device Drivers for block devices can use block layer infrastructure
 - Common functionality across all block devices
- Fairly complex to get it right!

In the bottom left corner, there is a circular logo with a star-like pattern and the text "NPTEL" below it. In the bottom right corner, there is a small inset image of a man with glasses, wearing a patterned shirt, with his hand to his forehead in a thoughtful or stressed pose.

The direction of these things it turns out most systems I what is called the block layer. So, there is a device driver. There are locks or device driver as possible. Now you can have a device driver for a tape, you can have a device driver for disk etcetera. Or you can have it for CD ROM etcetera. Now it turns out that many of them might have some commonality. So, because they are all talking intrude of reading in size some certain block sizes. They are talking about how to take care of interrupts etcetera. There is some commonality of functions.

So, most operating systems typically provide a block layer also. So, the device drivers use the block layer infrastructure to get their job done. So, in a sense you can see a hierarchy developing. You are something like a file system on top typically. Then you have a buffer cache layer next to it. Then below it, you might see a device block device driver. And bellow it might say a block layer, you will see. So, with lot of functionality that is plate a cost. And basically, this abstracts away the common functionality. So, that each block device driver does not have to reinvent the whole thing, but because it is doing it across so many different (Refer Time: 57:28) devices getting it right is not easy, but so, I will see that a Linux there has been lot of changes in block layer design quite a bit.

So, this has been changing quite a bit. And getting it right is non-trivial. We will at some point in later you might actually take a look at this block layer design there is (Refer

Time: 57:50) saying it is because, one of our one students subarna in in our department; she has designed the 2.5 Linux, 2.6 block layer design. She has done good portion of it. We will talk about it later.

So, basically if you think about this whole system, you will see that lot of interfaces like, Posix, device driver interface, and you have to very be careful how to use the infrastructure. If you do it try then it is easy, but there is a lot of functionality at different places, you have to an exact what you use.

I think we will stop here for the time being, and then we will continue from next class.