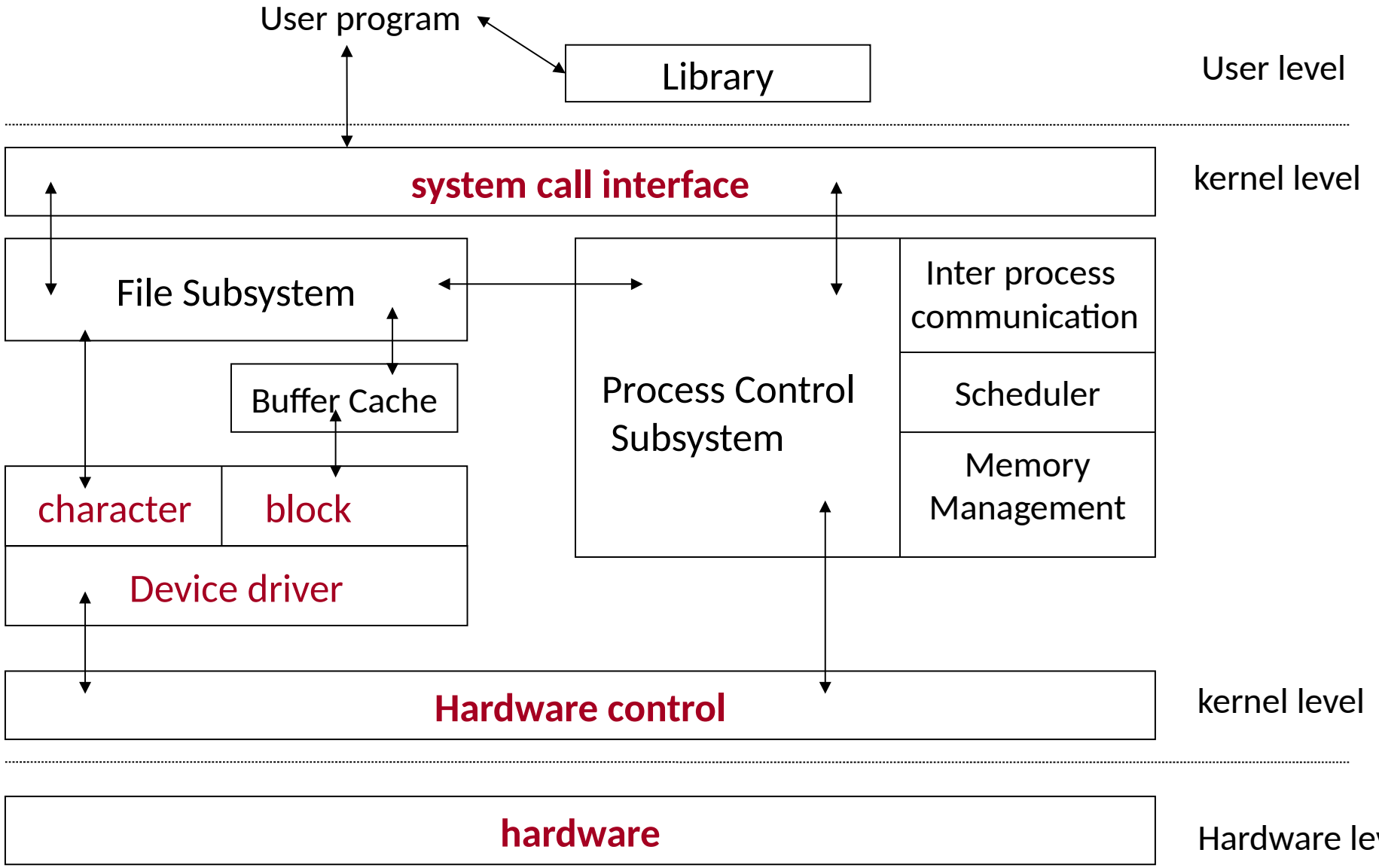
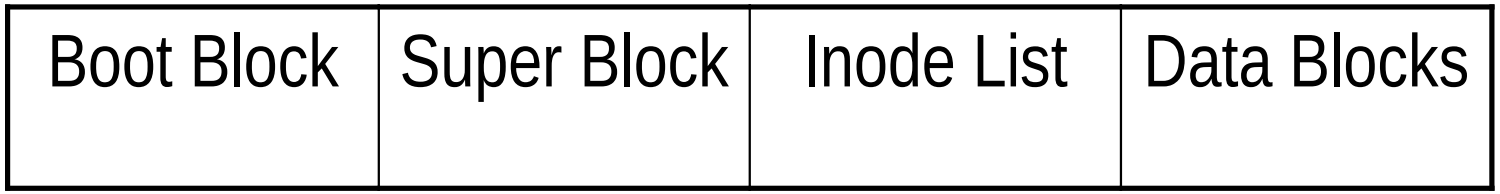
RECAP – COURSE OUTCOME-1:

Understood the internals of UNIX kernel architectures and explore design of File Subsystem, buffer cache, and File System Calls.

kernel Architecture (UNIX)



- A file system is consists of a sequence of logical blocks (512/1024 bytes)… A file system has the following structure:



Boot Block:

- The beginning of the file system
- Contains bootstrap code to load the os
- Initialize the operating system
- Typically occupies the first sector of the disk

Super Block

- Describes the state of a file system
- Describes the size of the file system
  - How many files it can store
- Where to find free space on the file system
- Other information

Inode List:

- Inodes are used to access disk files.
- Inodes maps the disk files
- For each file there is an inode entry in the inode list block
- Inode list also keeps track of directory structure

Data Blocks

- Starts at the end of the inode list
- Contains disk blocks
- An allocated data block can belong to one and only one file in the file system

COURSE OUTCOME-2

Understand the internals of system call and explore design of structure of processes, process control, process system calls and scheduling in UNIX systems

- Understand the internals of system call.
- Understand saving the context of a process, system calls for process control.

- Explore design of structure of processes and process control .
- Customize scheduling in UNIX systems and develop shell

- Analyze and Implement process system calls in UNIX systems

- Command-line interface
- To provide GUI

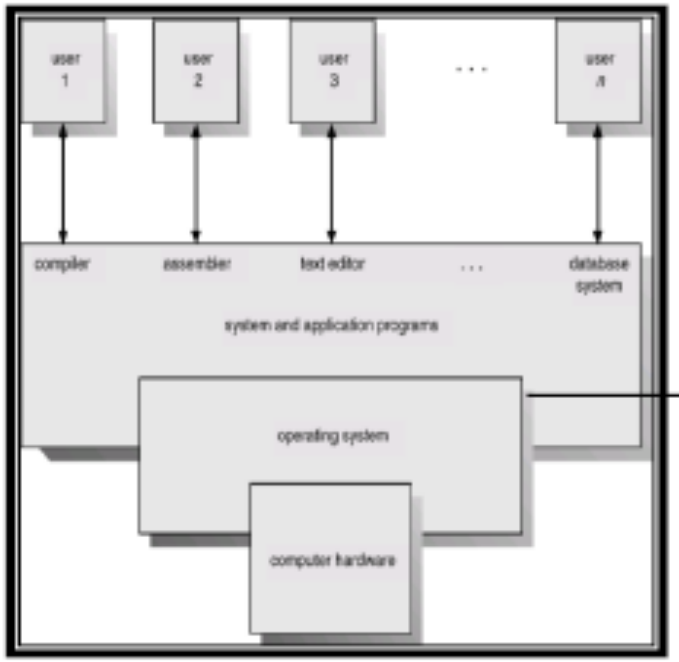
1

- Loading the program, executing, then storing the results back.

2

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

3



Abstract view of the components of a computer system

Operating System Services

1. User interface
2. Program execution
3. Process Management
4. Memory Management
5. I/O Device Management
6. File System Management
7. Disk Management
8. Error detection
9. Accounting.
10. Protection and Security

4

- Keeping track of which parts of memory are currently being used and by whom
- Deciding which processes (or parts thereof) and data to move into and out of memory
- Allocating and deallocating memory space as needed

5

- Managing the buffering, caching, and spooling
- A general device-driver interface
- Drivers for specific hardware devices

6

- Creating and deleting files
- Creating and deleting directories to organize files
- Supporting primitives for manipulating files and directories
- Mapping files onto secondary storage
- Backing up files on stable (non volatile) storage media

7

- Free-space management
- Storage allocation
- Disk scheduling

8

- Identifying different types of errors
- Rectifying them to ensure smooth execution

9

- Recording usage of resources
- Billing for used resources

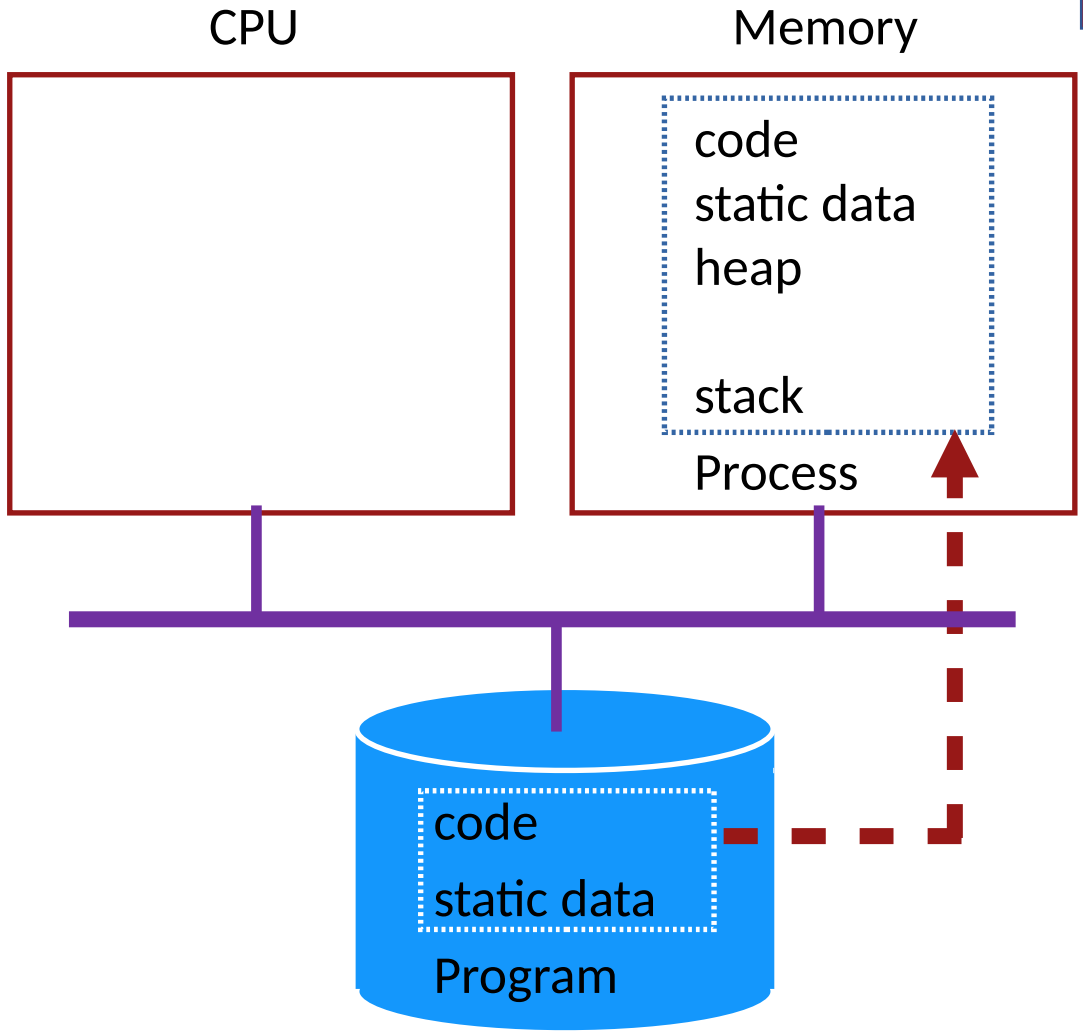
10

- To ensure that all access to system resources is controlled.
- Authentication mechanisms

Abstract view of the components of a Computer System with Operating System services

Operating system organization: Process overview

- A process is the execution of a program
- A process consists of text (machine code), data and stack segments.
- Many process can run simultaneously as kernel schedules them for execution
- Several processes may be instances of one program
- A process reads and writes its data and stack sections, but it cannot read or write the data and stack of other processes
- A process communicates with other processes and the rest of the world via system calls
- Kernel has a process table that keeps tract of all active processes
- Each entry in the process table contains pointers to the text, data, stack and the U Area of a process.
- All processes in UNIX system, except the very first process (process 0) which is created by the system boot code, are created by the fork system call



Operating system organization: Process overview

Process Context

- The context of a process is its state:
  - Text, data(variable), register
  - Process region table, U Area,
  - User stack and kernel stack
- When executing a process, the system is said to be executing in the context of the process.

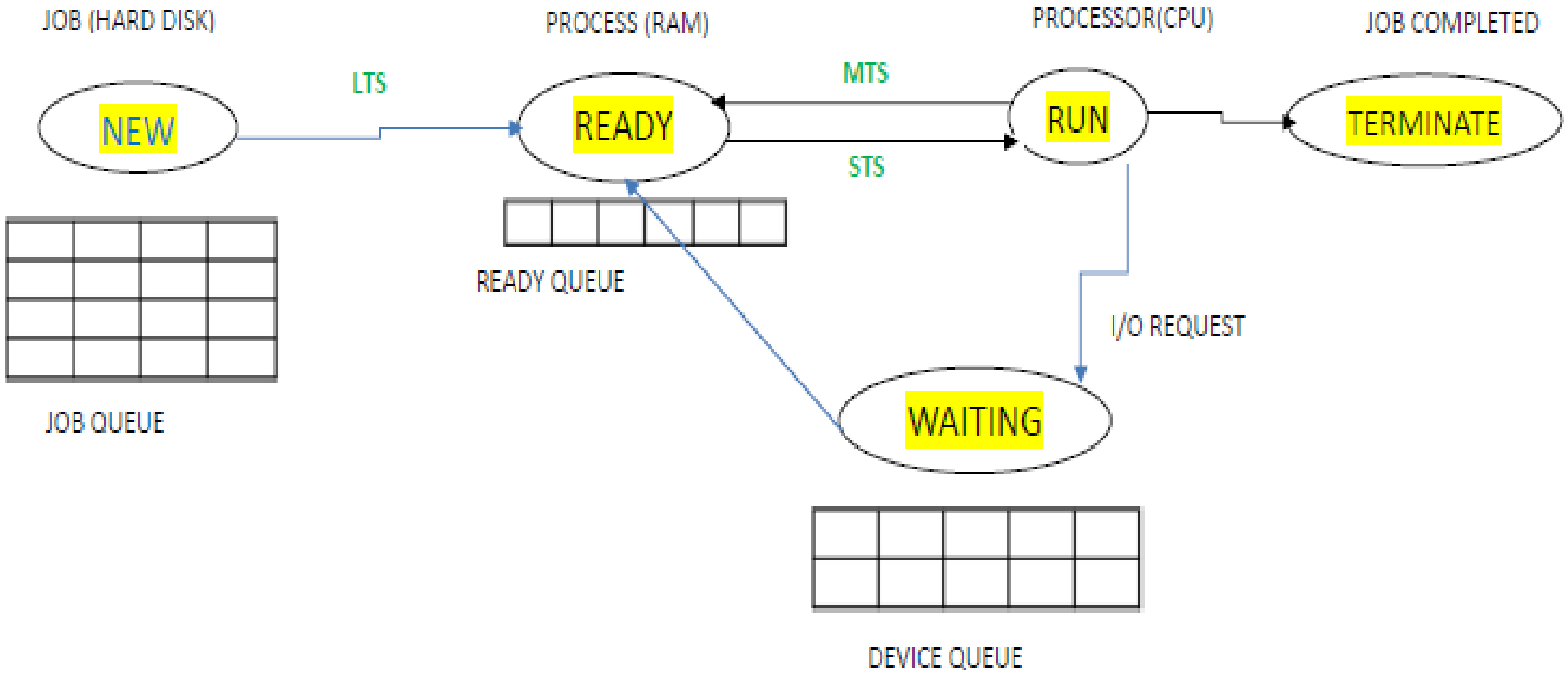
Context Switch

- When the kernel decides that it should execute another process, it does a context switch, so that the system executes in the context of the other process
- When doing a context switch, the kernel saves enough information so that it can later switch back to the first process and resume its execution.

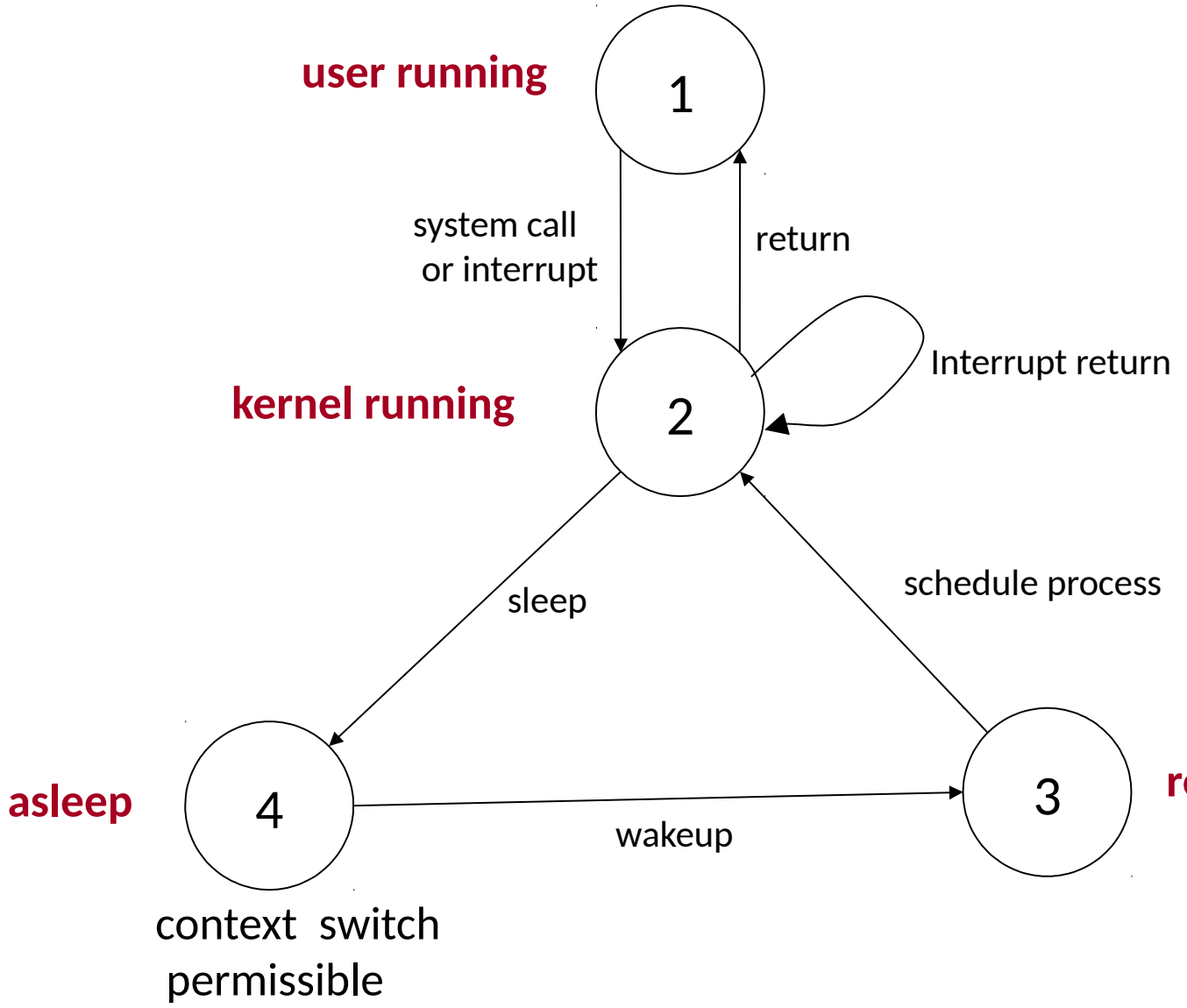
Mode of Process Execution

- The UNIX process runs in two modes:
  - **User mode**
    - Can access its own instructions and data, but not kernel instruction and data
  - **Kernel mode**
    - Can access kernel and user instructions and data
- When a process executes a system call, the execution mode of the process changes from user mode to kernel mode
- When moving from user to kernel mode, the kernel saves enough information so that it can later return to user mode and continue execution from where it left off.
- Mode change is not a context switch, just change in mode.

Process states and State Transition(Generic)



Process and State Transition(Specific to UNIX)

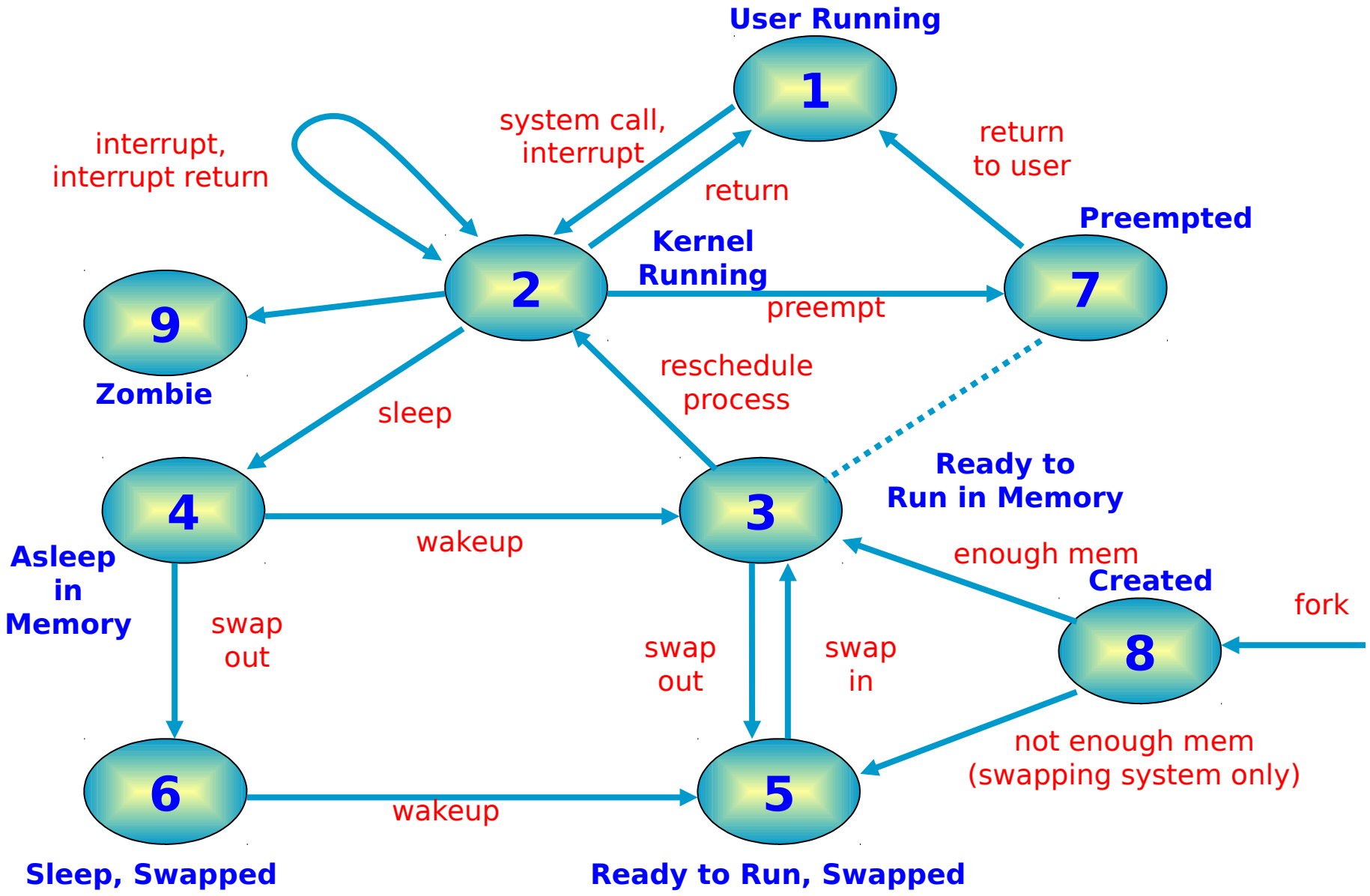


Process states are:

- The process is running in user mode
- The process is running in kernel mode
- The process is not executing, but it is ready to run as soon as the scheduler chooses it
- The process is sleeping
  - Such as waiting for I/O to complete

- **Process State Transition**
- The kernel allows a context switch only when a process moves from the state kernel running to the state asleep
- Process running in kernel mode cannot be preempted by other processes.

Process and State Transition in more elaborated way..



Process and State Transition

The complete set of process states

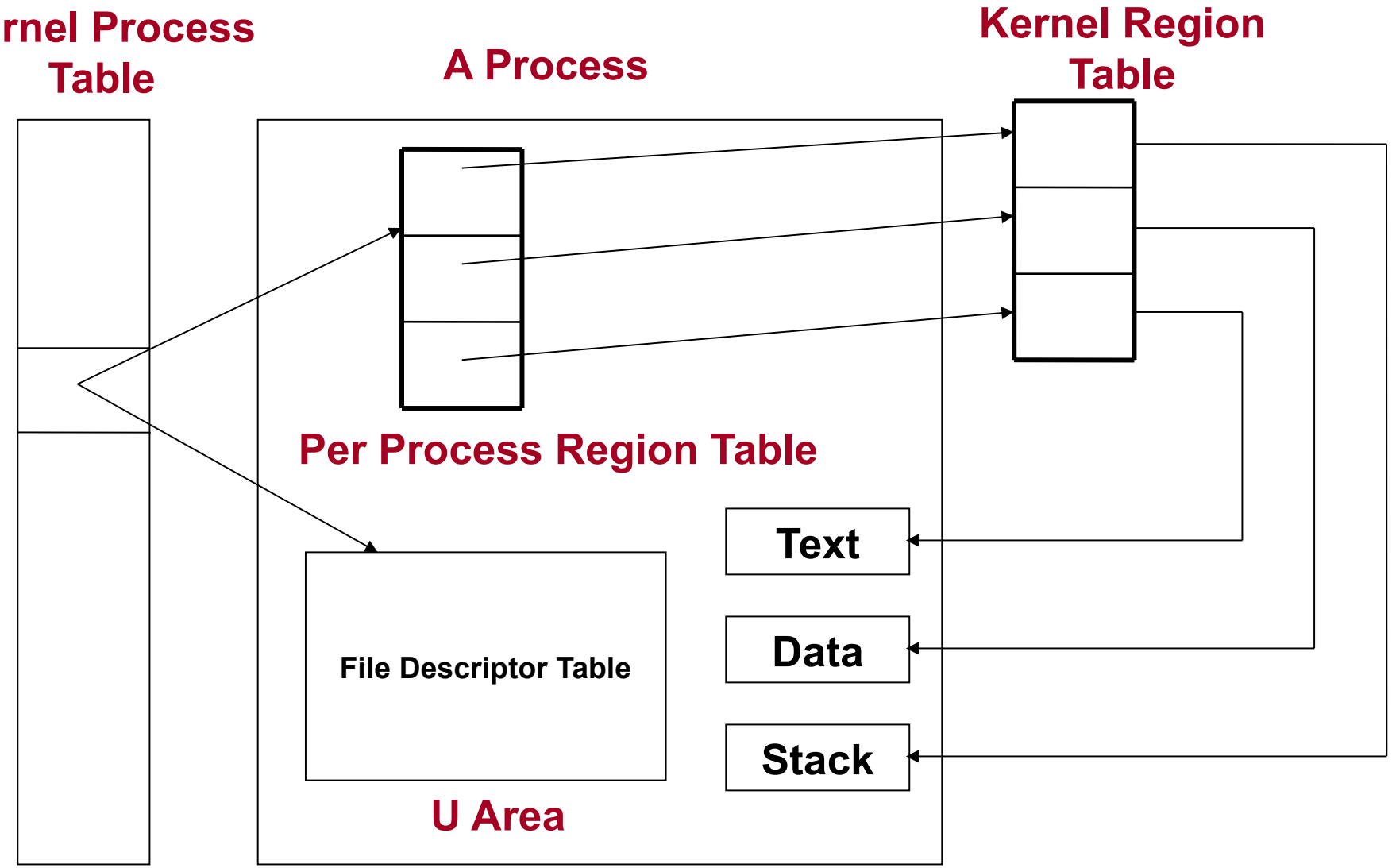
1. The process is executing in user mode
2. The process is executing in kernel mode
3. The process is not executing but is ready to run as soon as the kernel schedules it.
4. The process is sleeping and resides in main memory.
5. The process is ready to run, but the swapper (prcoess 0) must swap the process into main memory before the kernel can schedule it to execute.
6. The process is sleeping, and the swapper has swapped the process to secondary storage to make room for other processes in main memory.
7. The process is returning from the kernel to user mode, but the kernel preempts it and does a context switch to schedule another process. The distinction between this state and state 3 (" ready-to-run ") will be brought shortly.
8. The process is newly created and is in a transition state; the process exists, but it is not ready to run, nor is it sleeping. This state is the start state for all processes except process 0.
9. The process executed the exit system call and is in the zombie state. The process no longer exists, but it leaves a record containing an exit code and some timing statics for its parent process to collect. The zombie state is the final state of a process

Data Structures for Process

- **Two kernel data structures describe the state of a process: the process table entry and the uarea. The process table contains fields that must always be accessible to the kernel, but the uarea contains fields that need to be accessible only to the running process.**

- Therefore, the kernel allocates space for the uarea only when creating a process:

Kernel Support for Process:



Process Table:

Kernel has a process table that keeps track of all active processes
Each entry in the process table contains pointers to the text, data, stack and the U Area of a process.

- ✓ **State field:** user running, kernel running etc.
- ✓ **Fields** that allow the kernel to locate the process and u area. Requires while context switch
- ✓ **Process size :** kernel know how much space to allocate for the process.
- ✓ **User ID**
- ✓ **Process ID**
- ✓ **Event descriptor.**
  - Used when the process is in the "sleep" state.
- ✓ **Scheduling parameters.**
  - Allow the kernel to determine the order in which processes move to the states "kernel running" and "user running"
- ✓ **A signal field.**
  - keeps the signals sent to a process but not yet handled.
- ✓ **Various timers:** process execution time, resource utilization etc.

U area

U Area is the extension of process table entry.

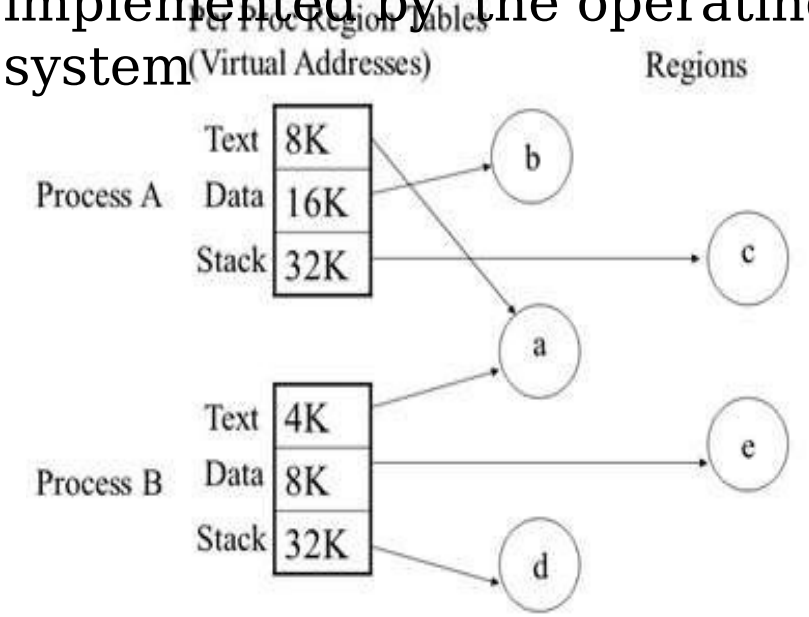
- ✓ Pointer to process table entry
- ✓ The current directory and current root
- ✓ User file descriptor table
- ✓ Limit fields
  - Restrict process size
  - Restrict size of the file it can write
- ✓ The control terminal field:
  - login terminal associated with the process, if one exists
- ✓ An array indicates how the process wishes to react to signal

Regions

- UNIX system consists of three logical sections: text, data, and stack.
- Region :-
  - is a contiguous area of the virtual address space of a process that can be treated as a distinct object to be shared or protected.
  - text, data, and stack usually form separate regions of a process. Several processes can share a region.
  - The kernel contains a region table and allocates an entry from the table for each active region in the system.
  - Shared regions may have different virtual addresses in each process
- Pregion :-
  - Each process contains a private per process region table, called a pregion.
  - Pregion entries may exist in the process table, the u area, or in a separately allocated area of memory, dependent on the implementation.
  - Each pregion entry points to a region table entry and contains the starting virtual address of the

Eg. Of Processes & Regions

The concept of the region is independent of the memory management policies implemented by the operating system



Region Table

Region table entries describes the attributes of the region, such as whether it contains text or data, whether it is shared or private

❑ Per Process Region Table (Pregion)

- The traditional kernel allocates several process regions, called **pregions,** for each process. The collection of pregions of a process form the **per process region table**.

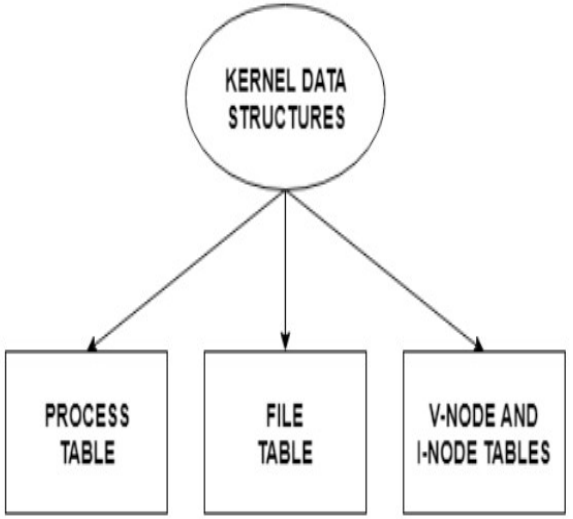
- The pregion data structure is used to record the following information:
  - ✓ The region types (e.g. text, data, stack, etc.)
  - ✓ Each pregion entry points to the kernel region table
  - ✓ Starting virtual (absolute) address of the region
  - ✓ Permission field: read-only, read-write, read-execute
- The extra level from the per process region table to kernel region table allows independent processes to share regions.

❑ Kernel Region table

- Kernel region table contains the pointer to the page table which keeps the physical memory address

Kernel Data Structures

The kernel data structures are very important as they store data about the current state of the system. For example, if a new process is created in the system, a kernel data structure is created that contains the details about the process.
Most of the kernel data structures are only accessible by the kernel and its subsystems. They may contain data as well as pointers to other data structures.



Kernel Components

The kernel stores and organizes a lot of information. So it has data about which processes are running in the system, their memory requirements, files in use etc. To handle all this, three important structures are used. These are process table, file table and v node/ i node information.

Process Table

The process table stores information about all the processes running in the system. These include the storage information, execution status, file information etc. When a process forks a child, its entry in the process table is duplicated including the file information and file pointers. So the parent and the child process share a file.

File Table

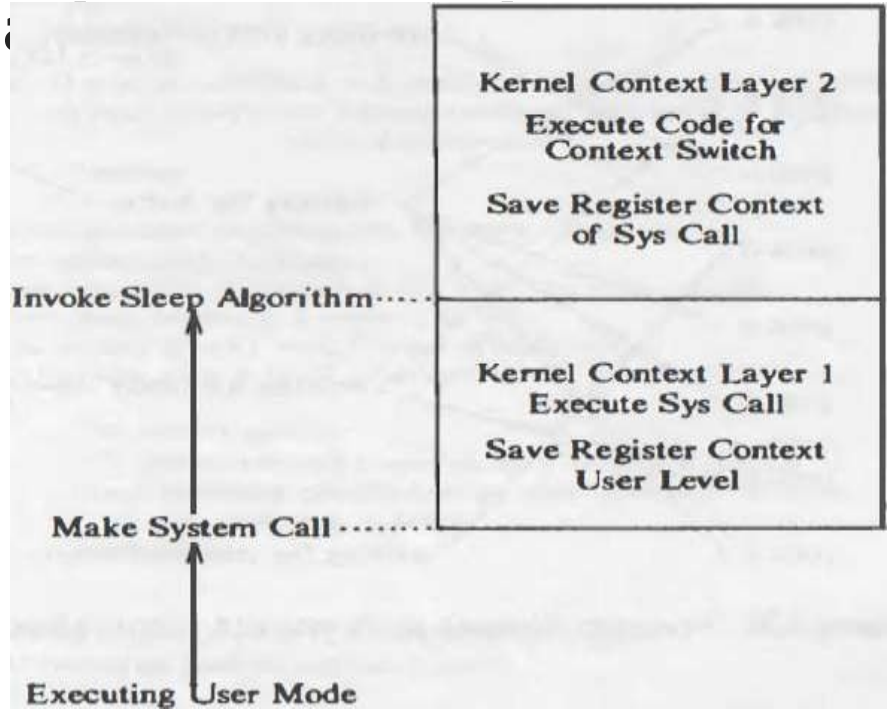
The file table contains entries about all the files in the system. If two or more processes use the same file, then they contain the same file information and the file descriptor number. Each file table entry contains information about the file such as file status (file read or file write), file offset etc. The file offset specifies the position for next read or write into the file. The file table also contains v-node and i-node pointers which point to the virtual node and index node respectively. These nodes contain information on how to read a file.

V-Node and I-Node Tables

Both the v-node and i-node are references to the storage system of the file and the storage mechanisms. They connect the hardware to the software. The v-node is an abstract concept that defines the method to access file data without worrying about the actual structure of the system. The i-node specifies file access information like file storage device, read/write procedures etc.

Slee
p

- The algorithms for sleep, which changes the process state from "kernel running" to "asleep in memory," and wakeup, which changes the process state from "a memory or swapped.



When a process goes to sleep, it typically does so during execution of a system call: The process enters the kernel (context layer I) when it executes an operating system trap and goes to sleep awaiting a resource. When the process goes to sleep, it does a context switch, pushing its current context layer and executing in kernel context layer 2 (Figure).

Processes also go to sleep when they incur page faults as a result of accessing virtual addresses that are not physically loaded; they sleep while the kernel reads in the contents of the

Sleep...

Continued...

Sleep Events and Addresses:

- Processes are said to sleep on an event, meaning that they are in the sleep state until the event occurs, at which time they wake up and enter a "ready-to-run" state (in memory or swapped out). Although the system uses the abstraction of sleeping on an event, the implementation maps the set of events into a set of (kernel) virtual addresses.
- The addresses that represent the events are coded into the kernel, and their only significance is that the kernel expects an event to map into a articular address.
- The abstraction of the event does not distinguish how many processes are awaiting the event, nor does the implementation. As a result, two anomalies arise. First, when an event occurs and a wakeup call is issued for

Slee p

```
algorithm sleep
input:  (1) sleep address
        (2) priority
output: 1 if process awakened as a result of a signal that process catches,
        longjump algorithm if process awakened as a result of a signal
                        that it does not catch,
        0 otherwise;
{
    raise processor execution level to block all interrupts;
    set process state to sleep;
    put process on sleep hash queue, based on sleep address;
    save sleep address in process table slot;
    set process priority level to input priority;
    if (process sleep is NOT interruptible)
    {
        do context switch;
        /* process resumes execution here when it wakes up */
        reset processor priority level to allow interrupts as when
            process went to sleep;
        return(0);
    }
    /* here, process sleep is interruptible by signals */
    if (no signal pending against process)
    {
        do context switch;
        /* process resumes execution here when it wakes up */
        if (no signal pending against process)
        {
            reset processor priority level to what it was when
                process went to sleep;
            return(0);
        }
    }
    remove process from sleep hash queue, if still there;

    reset processor priority level to what it was when process went to sleep;
    if (process sleep priority set to catch signals)
        return(1)
    do longjmp algorithm;
}
```

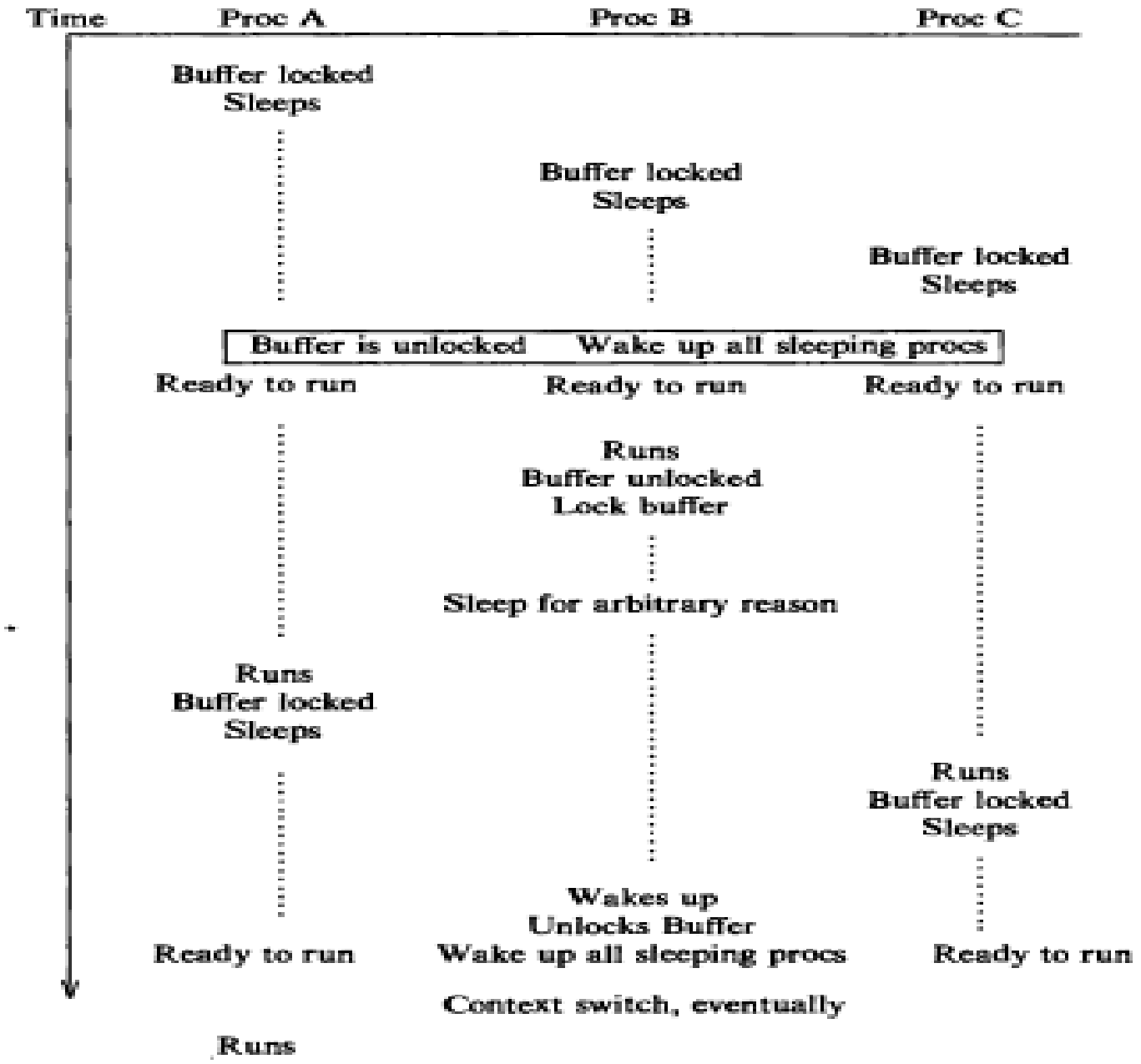


Figure 2.9. Multiple Processes Sleeping on a Lock

- Wakeup Algorithm
  - To wake up sleeping processes, the kernel executes the wakeup algorithm, either during the usual system call algorithms or when handling an interrupt.
  - For instance, the algorithm iput releases a locked inode and awakens all processes waiting for the lock to become free.
  - Similarly, the disk interrupt handler awakens a process waiting for I/O completion. The kernel raises the processor execution level in

```
algorithm wakeup              /* wake up a sleeping process */
input:  sleep address
output: none
{
        raise processor execution level to block all interrupts;
        find sleep hash queue for sleep address;
        for (every process asleep on sleep address)
        {
                remove process from hash queue;
                mark process state "ready to run";
                put process on scheduler list of processes ready to run;
                clear field in process table entry for sleep address;
                if (process not loaded in memory)
                        wake up swapper process (0);
                else if (awakened process is more elligible to run than
                                        currently running process)
                        set scheduler flag;
        }
        restore processor execution level to original level;
}
```

- To distinguish the types of sleep states, the kernel sets the scheduling priority of the sleeping process when it enters the sleep state, based on the sleep priority parameter. That is, it invokes the sleep algorithm with a priority value, based on its knowledge that the sleep event is sure to occur or not.
- If the priority is above a threshold value, the process will not wake up prematurely on receipt of a signal but will sleep until the event it is waiting for happens. But if the priority value is below the threshold value, the process will awaken immediately on receipt of the signal.

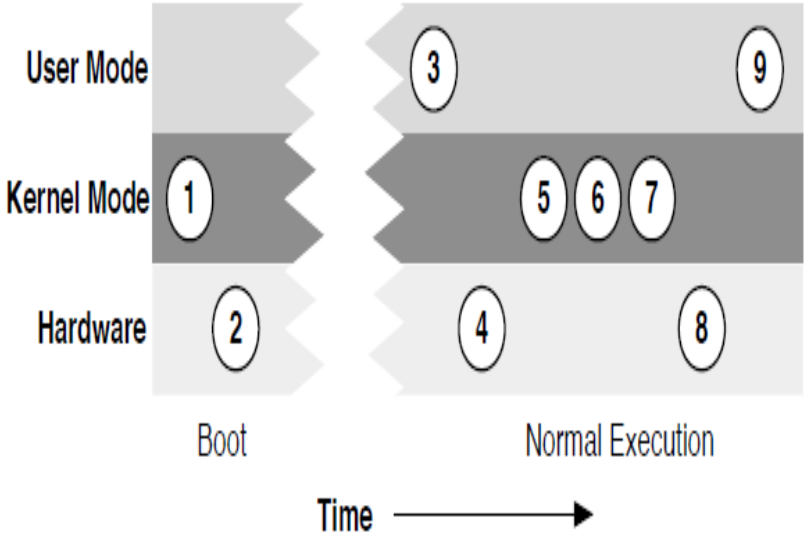
Thank You

Session Outcome:
Understand Under the Hood: The System Call

- Under the Hood system call
- **Xv6:** read(), tvinit, mpmain(), alltraps, traps, syscall()
- Understand usys.s, main.c, trap.c, mmu.h, vectors.s, trapasm.s, trap.c,traps.h, syscall.c

1. System Call Overview

- The xv6 kernel is a port of an old UNIX version 6
- In this session, we'll examine what goes on inside an os when system call is invoked.
- We'll specifically trace what happens in the code in order to understand a system call.



how can we build a system such that the OS is allowed access to all of the resources of the machine

the computer hardware(CPU) can be in two states(well several CPU architectures have more than two states). One is generally called User-mode (less privileged mode), and the other is called Kernel-mode (privileged mode).

- Generally all user processes begin with user-mode. When CPU is in user-mode, there are restrictions to what it can do, and cannot access hardware and other system resources.
- And when it needs access to system resources and hardware, it issues a request to the kernel. When the CPU is in kernel-mode, there is unrestricted access to all hardware and system resources.
- Upon this request, the scheduler pauses and saves the current context and then initiate to serve the request.
- After completion of the request the control comes back to the previous context and resumes its execution.

The operating system can switch the CPU from one mode to another mode as and when required.
This sort of pause and resume carried out by the operating system is also sometimes called as "Context Switch".

Difference between System Call and an Interrupt

A **hardware interrupt** is generated by a hardware attached to the system.

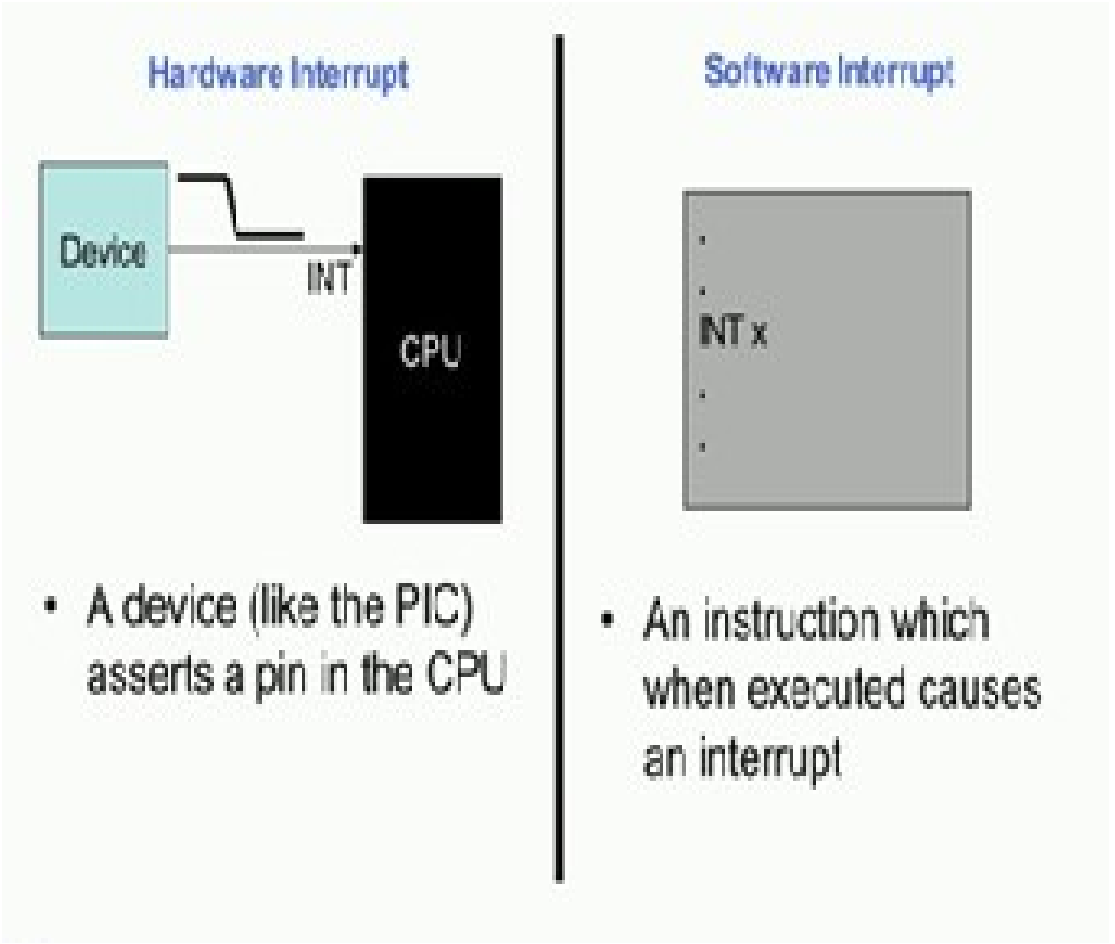
For example,

The printer attached to the system can generate an interrupt when it completes printing some document

device such as a keyboard or a network card could assert a particular signal in the CPU and this would cause the CPU to asynchronously execute an interrupt handler corresponding to the device.

So, this device would typically send a signal to the CPU through an intermediate device such as a PIC or a programmable interrupt controller.

In much the similar way, we have what is known as a Software Interrupt. A software interrupt is generated by a program running on the computer to get attention from the CPU.



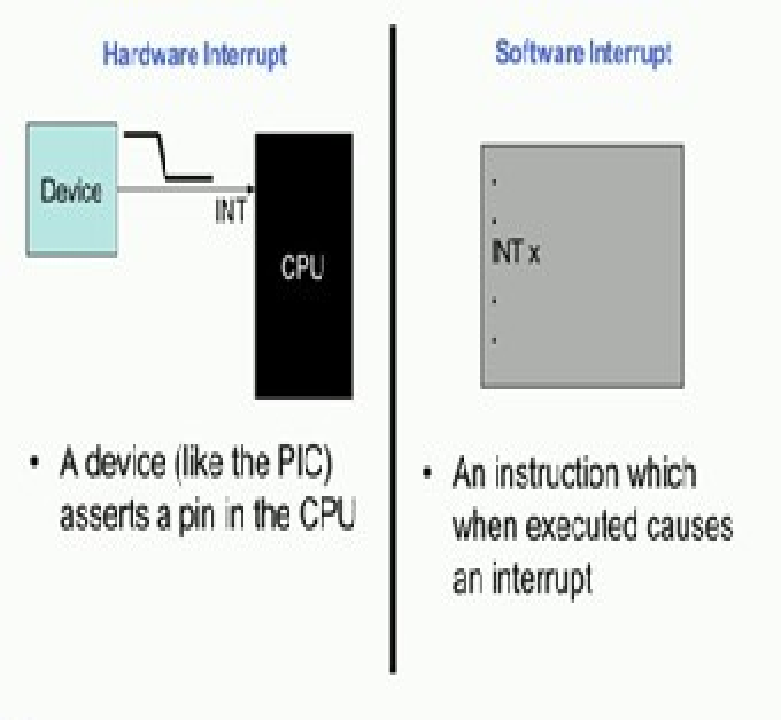
However, unlike having an external device which causes the interrupt, here an instruction in the program would trigger the interrupt. In this particular case, for example, an instruction such as INT would cause the interrupt to occur and the operating system to execute.

So, here the instruction is INT x, so x here is the interrupt number. It typically has a value less than 256, and it is used to specify or distinguish between software interrupts.

So when a particular interrupt gets triggered, from the options present in the entry for that interrupt, the CPU knows which privilege level it should set for executing that interrupt handler.

System call is generally a software interrupt. So one important thing to remember is this... **Both system calls and interrupts are numbered**

One major difference between software interrupt and hardware interrupt is the fact that hardware interrupt can be fired at any time. For example, a user typing something on the keyboard will trigger a hardware interrupt, but software interrupt can only be triggered by something which is currently being executed(ie: currently running) by the CPU.

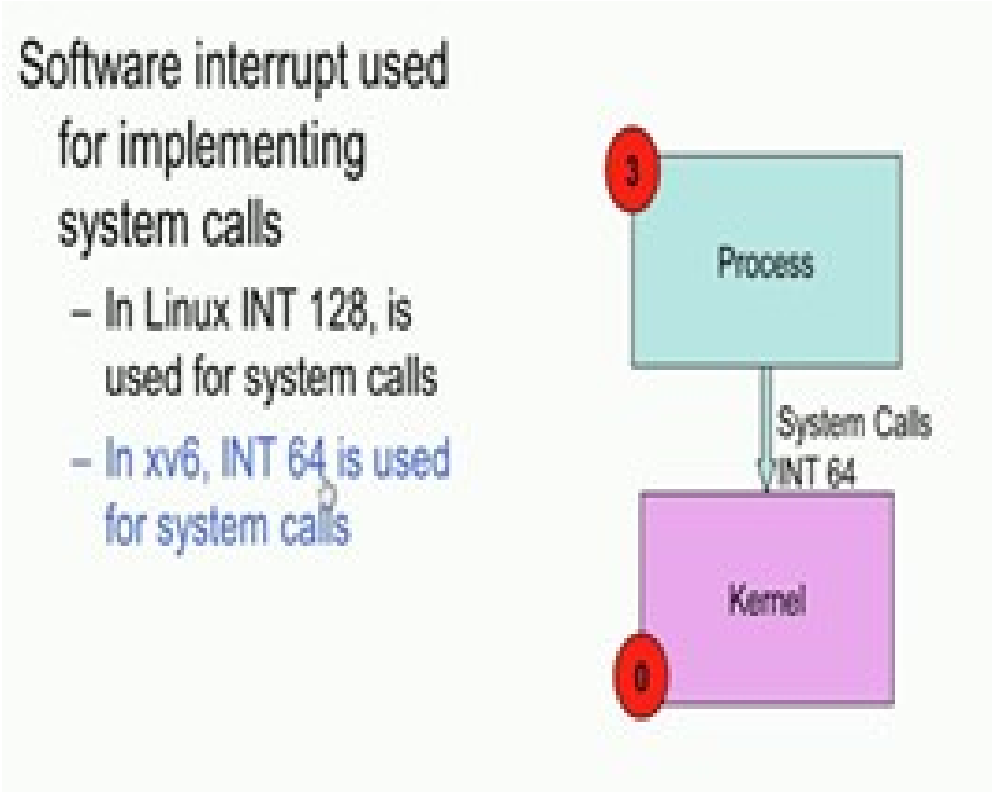


where is the software interrupt used?

- Software interrupts are used to implement system calls. So, as we know a user process could invoke a system call to perform some Kernel operation. For example, it could be to read a file or to write a file, to print something to a monitor, or to send a packet through the network and so on. More specifically, all operating systems implement system calls through one particular software interrupt.

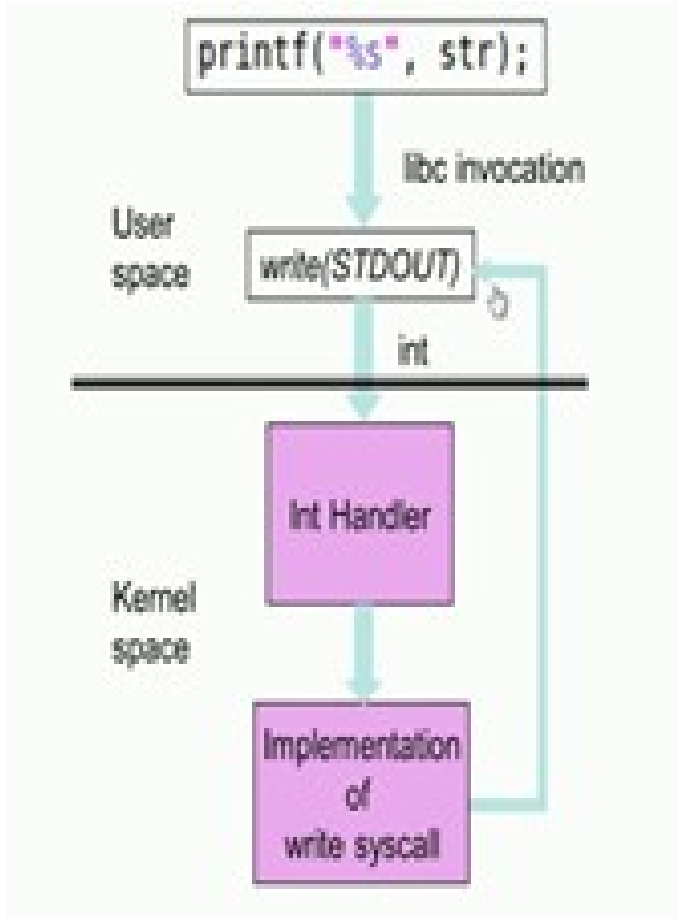
- For example, in the Linux operating systems, the software interrupt 128 is used to specify system calls. Therefore, in a **Linux OS,** if I have a **INT 128** which is executed in the user process, it would lead to an interrupt that occurs and cause the kernel or the operating system to execute, and thereafter the OS would execute code depending on the interrupt.

- In xv6, the software interrupt used to implement system calls is 64 or instruction like **INT 64** in the user process would be meant to implement a system call.



Software interrupt ..Example

- So to take an example, let us consider that our application has a printf statement present in it. So, printf would print this string ‘str’ (mentioned in above image) to the standard output, which typically is the monitor. Now printf is a function present in the library libc and it would cause the libc function to be invoked. Now in the libc function there is a call to the write system call with the specifier STDOUT i.e write(STDOUT). So, the STDOUT here is the file descriptor; and it is a special file descriptor which is meant for the standard output or the monitor.



- So in the write function, it would invoke INT 64 in xv6 or INT 128 in Linux and cause a software interrupt to occur. So, the software interrupt as we know would cause the transformation from the User space to the Kernel space and it would result in the operating system executing. The OS would then determine that the interrupt was in fact due to a system call and then it would determine what system call it was from; in this case, it was from a write system call and it was a write to the STDOUT - the standard output.

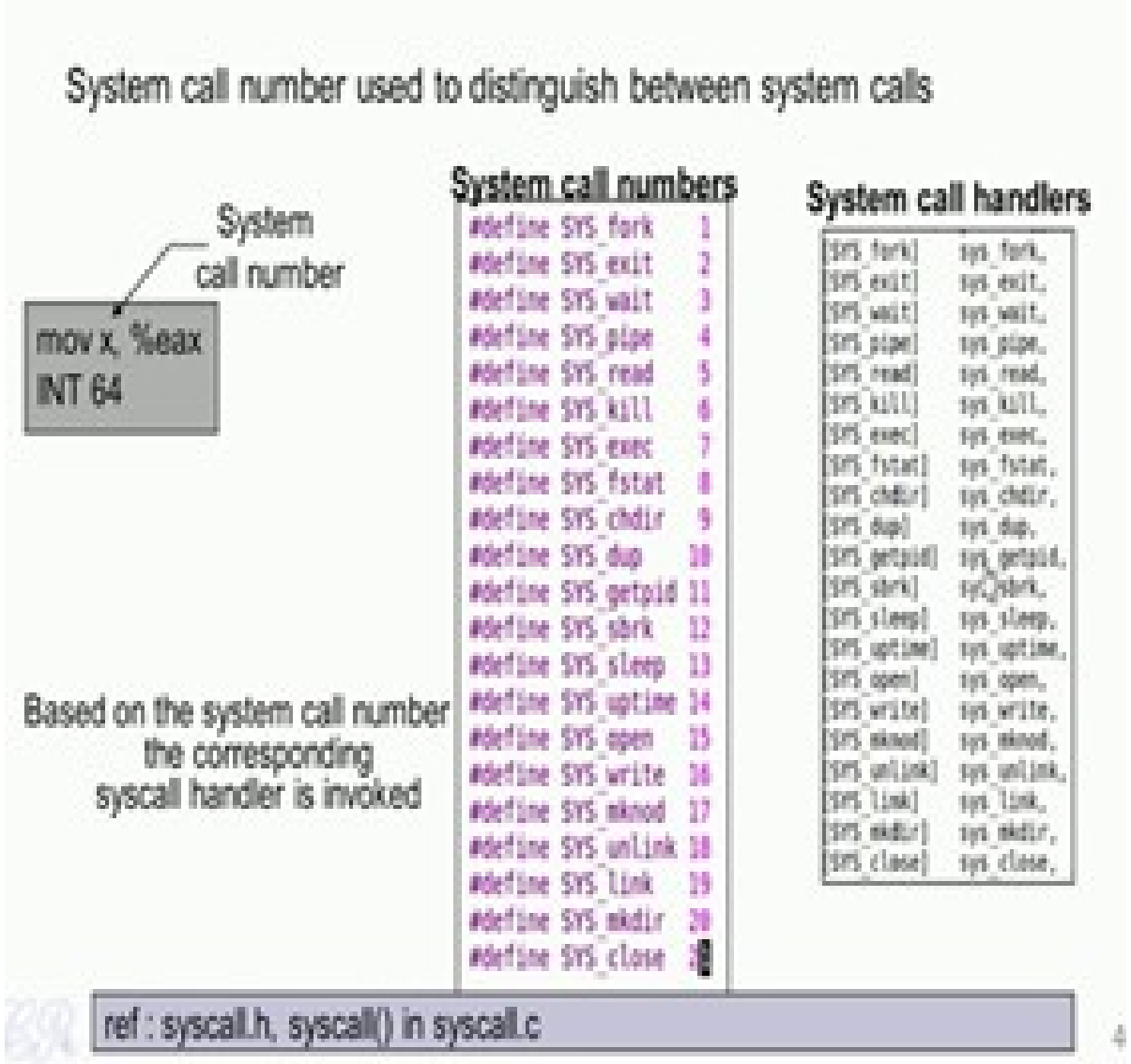
- So the operating system would then invoke the handler for the write system call, and this handler would take care of communicating with the various devices such as the video card to display the string onto the monitor. So, after this handler completes execution, the **IRET instruction** is executed which would result in a transformation back from Kernel space to the User space and the program will continue to execute.

Different types of system calls in XV6

- Now, typically operating systems support several different types of system calls. So, this particular table over here shows the various system calls supported by xv6. And you are also familiar with several types of system calls such as open(), read(), write(), close(), change directory chdir(), make directory mkdir() and so on. So, each of these system calls would be executed by having a software interrupt such as INT 64, because it is the xv6, so it is 64. So, each time any of these system calls are invoked by a user process, it would trigger the operating system to execute.

- Now, the next obvious question that one would ask is from the OS perspective, **how does the OS distinguish between the various system calls?** So, we mention that all the system calls will use either INT 64 for xv6 and INT 128 for Linux. So how does the OS determine whether the system call was with respect to fork(), wait(), sleep(), exit(), and so on. **Essentially this distinguisher comes from the user process itself.**

System call	Description
fork()	Create process
exit()	Terminate current process
wait()	Wait for a child process to exit
kill(pid)	Terminate process pid
getpid()	Return current process's id
sleep(n)	Sleep for n seconds
exec(filename, *argv)	Load a file and execute it
sbrk(n)	Grow process's memory by n bytes
open(filename, flags)	Open a file; flags indicate read/write
read(fd, buf, n)	Read n byes from an open file into buf
write(fd, buf, n)	Write n bytes to an open file
close(fd)	Release open file fd
dup(fd)	Duplicate fd
pipe(p)	Create a pipe and return fd's in p
chdir(dirname)	Change the current directory
mkdir(dirname)	Create a new directory
mknod(name, major, minor)	Create a device file
fstat(fd)	Return info about an open file
link(f1, f2)	Create another name (f2) for the file f1
unlink(filename)	Remove a file

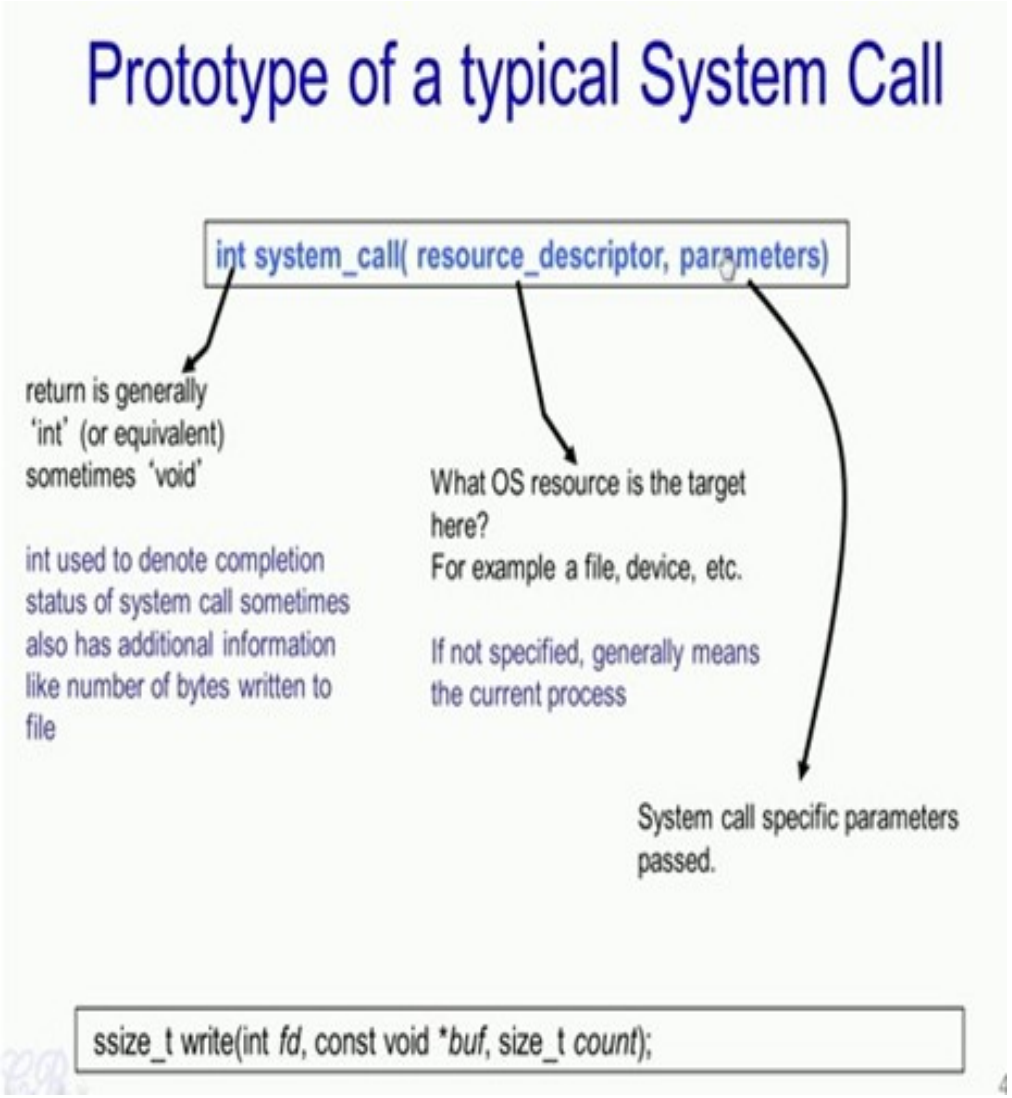


What happens is that, before the INT 64 instruction, the user process will move a system call number to the eax register. So, for example, instruction mov x, %eax will move the system call number to the eax register. Now each system call in the operating system will have a unique number, system call number. Now the operating system when triggered by the INT instruction would look up the eax register, and then determine what system call was invoked. For example, in xv6, if we look up these particular header files (mentioned in above image), we would see the various system call numbers defined. For example, we have each system call given a specific number, for example SYS_fork given 1, SYS_exit giving 2 and so on.

Now when the OS gets trigger due to the INT 64 instruction getting executed, the OS will determine the system call using this system call numbers and then invoke the corresponding system call handler. So, each of the system calls also have a corresponding system call handler corresponding to each of the system call numbers SYS_fork that is 1, SYS_exit it is 2 and so on.

So, these are system call functions (mentioned above in System call handler list) present in the operating system which gets triggered based on the type of the system call. For example, if eax had a value of 11, the operating system will look in to the eax register and determine that this corresponded to the getpid system call being invoked.

And then it would look into this particular table (mentioned above in System call handler list) and see that the getpid system call is handled by this function sys_getpid, and therefore it will then invoke this sys_getpid function.



Now, let us look at the typical prototype of a system call. It has a system call name that is a function name and then it is passed some resource descriptor and parameters and typically would return an integer. So, the resource descriptor specifies what operating system resource is the target here. For example, it could be a file or a device; and as we have seen in the previous slides it could also specify a particular monitor. For example, if the resource descriptor is STDOUT then the resource in use here is the monitor.

So, some system calls also do not specify this resource descriptor; in such a case, the system call is meant for that resource itself. For example, if we use the sleep system call, we only specify the time and no specific descriptor such as the file, a device and so on. This means that the current process wants to sleep for that given interval time.

The next part is the parameters. So, these parameters are specific for the system call. For example, if we invoke read, write, open or close or any other system call, the parameters specified here is going to be very specific for each of these system calls. For example, the write system call has the parameter *buf that is a void pointer and the count. So, the open or the close or any other system call would have different set of parameters. So, essentially these parameters are very specific to the type of the system call.

The return type is typically int or integer, and sometimes it is a void. So, ‘int’ is typically used because in this way the operating system will be able to send the completion status of the system call, whether it had executed successfully or it had failed and so on. So, sometimes the return is also used to specify certain specific information about the system call. For example, in write the return is ssize_t which in fact, is typdef to integer and it specifies number of bytes that have been written to the file, specified int fd. So, the return type could also vary depending on the type of system call.

The next thing what we will look at:

How are the parameters such as the resource descriptor and the other parameters passed to the kernel?

Passing Parameters in System Calls

```
ssize_t write(int fd, const void *buf, size_t count);
```

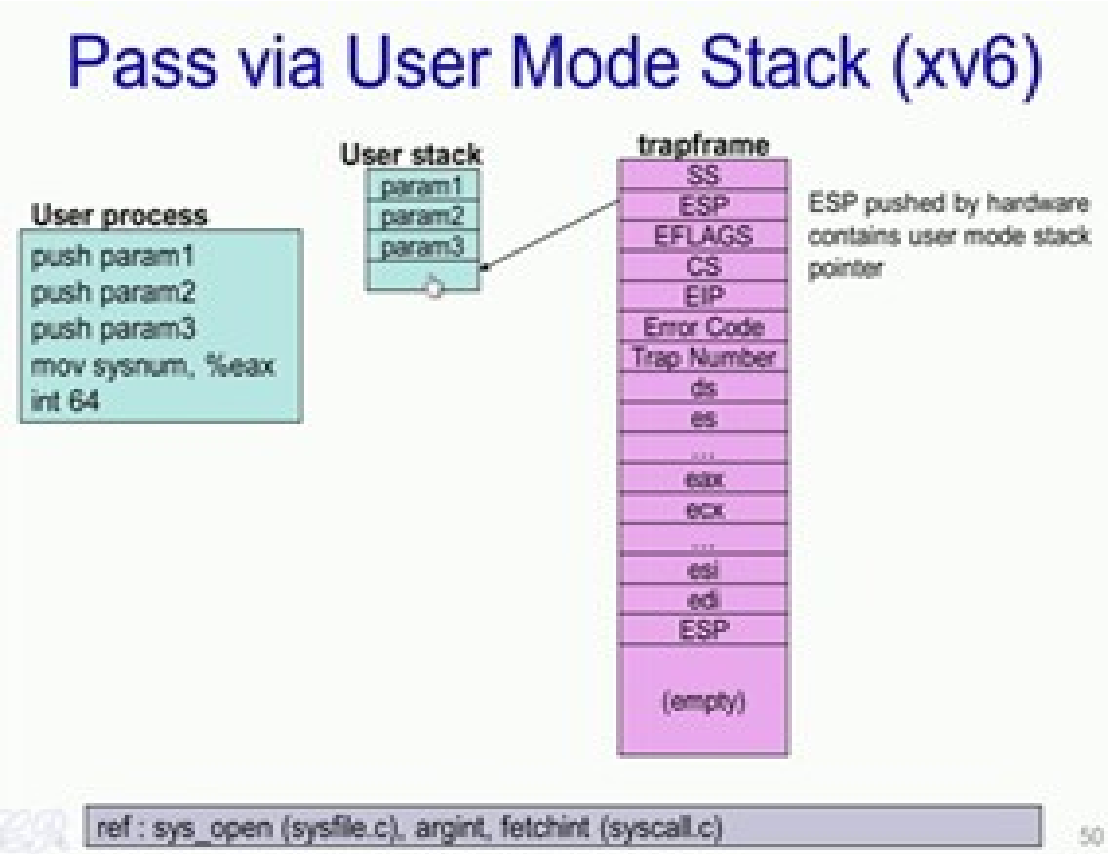
- Passing parameters to system calls not similar to passing parameters in function calls
  - Recall stack changes from user mode stack to kernel stack.
- Typical Methods
  - Pass by Registers (eg. Linux)
  - Pass via user mode stack (eg. xv6)
    - Complex
  - Pass via a designated memory region
    - Address passed through registers

Pass By Registers (Linux)

- System calls with fewer than 6 parameters passed in registers
  - %eax (sys call number), %ebx, %ecx,, %esi, %edi, %ebp
- If 6 or more arguments
  - Pass pointer to block structure containing argument list

So essentially, there are three ways of doing so.
The first is by pass by registers which is typically done in Linux; the second way is by passing through the user mode stack which is done in xv6; and
the third way is by passing through a designated memory region.
So, in this particular case (third case), what is done is that in the user process itself, a designated region most likely in the heap would be used to save the various parameters that is needed to be passed to the system call; and the address to this region in the heap is passed through the registers. So, we will look at the other two cases that is pass by registers and pass via user mode stack in more detail.

Now Pass by Registers which is used by Linux system calls would use the registers present in the processor to pass parameters to the kernel. So, we know we have already seen an example of this, of how the %eax register is used to pass the system call number from the user process to the operating system. In a similar way, other registers such as the %ebx, %ecx, %esi, %edi, and %ebp are used to pass the various parameters of the system call from the user process to the kernel. If the system call has more than 6 arguments then a pointer to a block structure containing the argument list is passed to the kernel.

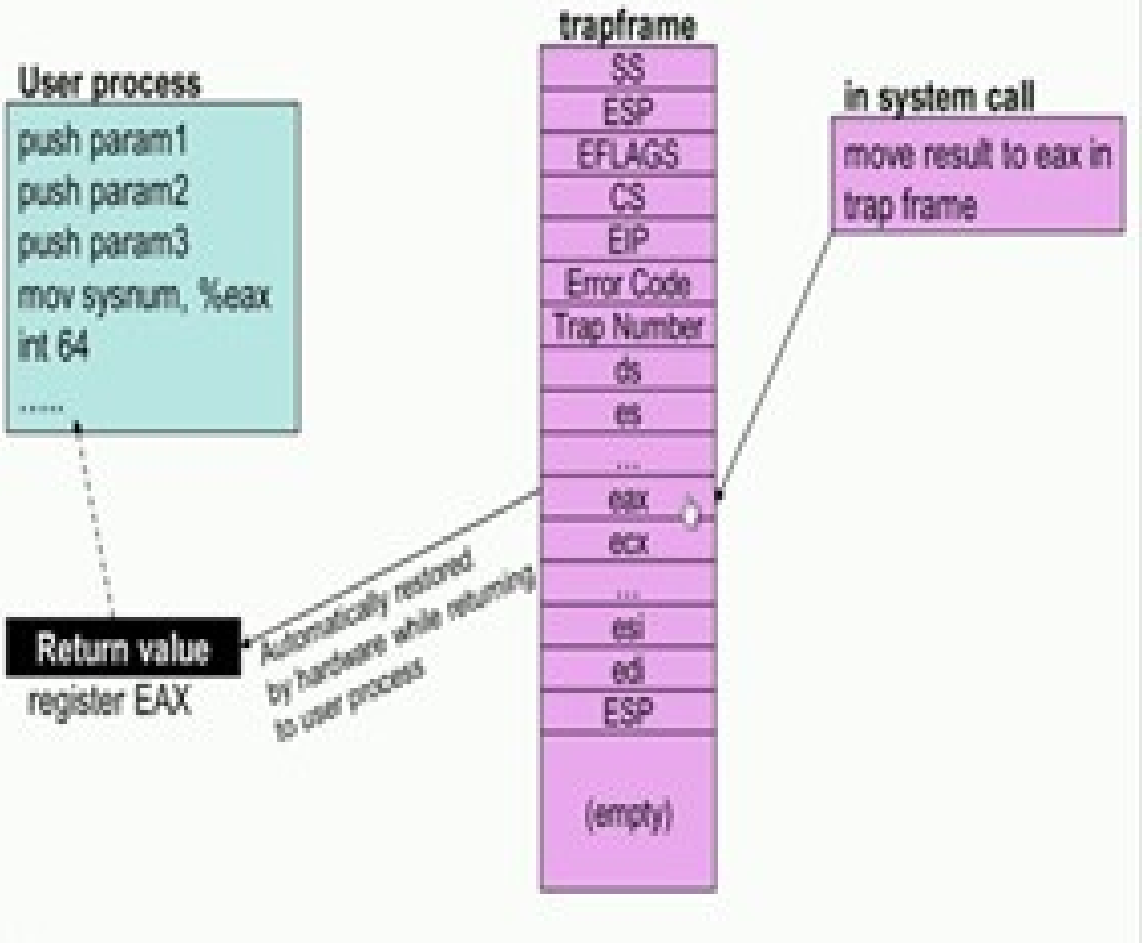


Now, the SS and ESP in trapframe box are the stack segment and the stack pointer, and these registers correspond to the user space stack. So, as we know before the INT 64 has been executed, the last known stack pointer was pointing to this particular location. And therefore, the contents of ESP will also point to this location in the user space stack. So, in this way, the kernel will then use the SS and the ESP from the trapframe to determine the various parameters for the system call.

Now, let us look at the second case that is Pass via the User Mode Stack and this is what is done in xv6. So, in this particular way, before the int 64 instruction is present various parameters in the system call are pushed onto the stack. For example, if the system call had 3 parameters; param 1, param 2, and param 3, so these three parameters are pushed into the user space stack and then the system call number moved into the %eax register. So, this here is the user space stack of the user process containing the 3 parameters.

Now, when the INT instruction is executed, as we know, it triggers an interrupt causing the switch from the user space into the kernel space. Also as a result of this interrupt execution, as we have seen, there is the switch in the stack from the user space stack to the kernel space stack. This kernel stack is used to create what is known as the **trapframe**. Some of the entries in the trapframe are pushed into the stack automatically by the CPU. So, in particular, these registers specified in capitals letters are all pushed onto the kernel stack that is on to the trapframe by the CPU.

Returns from System Calls



The next thing we will look at is how the return value is passed from the system call back to the user process. Again we will recollect that the entire reason for creating this trapframe in the kernel stack for the process is due to the reason that when the interrupt or the system call completes its execution, the entire state in the trapframe is restored back into the corresponding CPU registers, and it would result in the user process continuing to execute from where it had stopped, and also the context of the user process is restored with the help of the trapframe.

Now, in order to return a value from the system call which is executing in the kernel space back to the user space, so what is done is that the eax register in the trapframe is modified; essentially, we had seen that the eax register because of this particular instruction (mov sysnum, %eax) would contain the system call number.

Now this system call number is overridden by the return value of the system call. So, this could be a negative number like -1 or a positive number as we have seen in the earlier slide. So, now when the system call executes and completes its execution and the context is transferred back to the user process, the entire trapframe including the new value of eax in the trapframe is restored back in to the registers of the CPU. The process continues to execute from this particular instruction.

IMPLEMENTATION THROUGH XV6:

1. System Call

We need to examine the path in and out of the kernel on a system call. It is much more complex than a simple procedure call, and requires a careful protocol on behalf of the OS and hardware to ensure that application state is properly saved and restored on entry and return.

2. Getting Into The Kernel: Issuing a Trap as specified in usys.S

- The first step in a system call begins at user-level with an application.
- The application that wishes to make a system call (such as read()) calls the relevant library routine.
- Here we can see that the read() library function actually doesn't do much at all; it moves the value 5 into the register %eax and issues the x86 trap instruction which is called **int** (short for "interrupt").
- The value in %eax is going to be used by the kernel to **vector** to the right systemcall, i.e., it determines which system call is being invoked. The int instruction takes one argument (here it is 64) which tells the hardware which trap type this is.
- In xv6, trap 64 is used to handle system calls.
- Any other arguments which are passed to the system call are passed on the stack.

```
.globl read;
read:
movl $5,
%eax;
int $64;
ret
```

usys.S

3. Kernel Side: Trap Tables

Once the **int** instruction is executed, the hardware takes over and does a bunch of work on behalf of the caller. One important thing the hardware does is to raise the **privilege level** of the CPU to kernel mode; on x86 this is usually means moving from a **CPL (Current Privilege Level)** of 3 (the level at which user applications run) to CPL 0 (in which the kernel runs). Yes, there are a couple of privilege levels in-between, but most systems do not make use of these.

The second important thing the hardware does is to transfer control to the **trap vectors** of the system. To enable the hardware to know what code to run when a particular trap occurs, the OS, when booting, must make sure to inform the hardware of the location of the code to run when

```
// FILE: main.c
int
main(void)
{
...
tvinit(); // trap vectors initialized here
...
}
```

The routine tvinit() is the relevant one here. Peeking inside of it, we see:

```
// FILE: trap.c
void tvinit(void)
{
int i;
for(i = 0; i < 256; i++)
SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
// this is the line we care about...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
vectors[T_SYSCALL], DPL_USER);
initlock(&tickslock, "time");
}
```

3. Kernel Side: Trap Tables..contd..

The SETGATE() macro is the relevant code here. It is used to set the idt array to point to the proper code to execute when various traps and interrupts occur. For system calls, the single SETGATE() call (which comes after the loop) is the one we're interested in. Here is what the macro does (as well as the gate
descriptor it sets):

```
// FILE: mmu.h
// Gate descriptors for interrupts and traps
struct gatedesc {
uint off_15_0 : 16; // low 16 bits of offset in segment
uint cs : 16; // code segment selector
uint args : 5; // # args, 0 for interrupt/trap gates
uint rsv1 : 3; // reserved(should be zero I guess)
uint type : 4; // type(STS_{TG,IG32,TG32})
uint s : 1; // must be 0 (system)
uint dpl : 2; // descriptor(meaning new) privilege level
uint p : 1; // Present
uint off_31_16 : 16; // high bits of offset in segment
};
```

```
// Set up a normal interrupt/trap gate descriptor.
// - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
// interrupt gate clears FL_IF, trap gate leaves FL_IF alone
// - sel: Code segment selector for interrupt/trap handler
// - off: Offset in code segment for interrupt/trap handler
// - dpl: Descriptor Privilege Level -
// the privilege level required for software to invoke
// this interrupt/trap gate explicitly using an int instruction.
#define SETGATE(gate, istrap, sel, off, d) \
{ \
(gate).off_15_0 = (uint) (off) & 0xffff; \
(gate).cs = (sel); \
(gate).args = 0; \
(gate).rsv1 = 0; \
(gate).type = (istrap) ? STS_TG32 : STS_IG32; \
(gate).s = 0; \
(gate).dpl = (d); \
(gate).p = 1; \
(gate).off_31_16 = (uint) (off) >> 16; \
}
```

As you can see from the code, all the SETGATE() macros does is set the values of an in-memory data structure. Most important is the off parameter, which tells the hardware where the trap handling code is. In the initialization code, the value vectors[T SYSCALL] is passed in; thus,whatever the vectors array points to will be the code to run when a system call takes place. There are other details (which are important too); consult an x86 hardware architecture manual for more information.

3. Kernel Side: Trap Tables..contd..

Note, however, that we still have not informed the hardware of this information, but rather filled a data structure. The actual hardware informing occurs a little later in the boot sequence; in xv6, it happens in the routine mpmain() in the file main.c:

```
static void
mpmain(void)
{
idtinit();
...
void
idtinit(void)
{
lidt(idt, sizeof(idt));
}
static inline void
lidt(struct gatedesc *p, int size)
{
volatile ushort pd[3];
pd[0] = size-1;
pd[1] = (uint)p;
pd[2] = (uint)p >> 16;
asm volatile("lidt (%0)" : : "r" (pd));
}
```

Here, you can see how (eventually) a single assembly instruction is called to tell the hardware where to find the **interrupt descriptor table (IDT)** in memory. Note this is done in mpmain() as each processor in the system must have such a table (they all use the same one of course).

3. Kernel Side: Trap Tables..contd..

```
struct trapframe {
// registers as pushed by pusha
uint edi;
uint esi;
uint ebp;
uint oesp; // useless & ignored
uint ebx;
uint edx;
uint ecx;
uint eax;
// rest of trap frame
ushort es;
ushort padding1;
ushort ds;
ushort padding2;
uint trapno;
// below here defined by x86 hardware
uint err;
uint eip;
ushort cs;
ushort padding3;
uint eflags;
// below here only when crossing rings, such as from user to kernel
uint esp;
ushort ss;
ushort padding4;
};
```

File: x86.h

The OS has carefully set up its trap handlers, and thus we are ready to see what happens on the OS side once an application issues a system call via the **int** instruction. Before any code is run, the hardware must perform a number of tasks. The first thing it does are those tasks which are difficult/impossible for the software to do itself, including saving the current PC (IP or EIP in Intel terminology) onto the stack, as well as a number of other registers such as the eflags register (which contains the current status of the CPU while the program was running), stack pointer, and so forth.

One can see what the hardware is expected to save by looking at the trapframe structure as defined in x86.h.

4. From Low-level To The C Trap Handler

As you can see from the bottom of the trapframe structure, some pieces of the trap frame are filled in by the hardware (up to the err field); the rest will be saved by the OS. The first code OS that is run is vector64() as found in vectors.S (which is automatically generated by vectors.pl).

```
.globl vector64
vector64:
pushl $64
jmp alltraps
```

File: vectors.S (generated by vectors.pl)

This code pushes the trap number onto the stack (filling in the trapno field of the trap frame) and then calls **alltraps()** to do most of the saving of context into the trapframe. The code in alltraps() pushes a few more segment registers (not described here, yet) onto the stack before pushing the remaining general purpose registers onto the trap frame via a pushal instruction. Then, the OS changes the descriptor segment and extra segment registers so that it can access its own (kernel) memory. Finally, the C trap handler is called.

5.The C Trap Handler

Once done with the low-level details of setting up the trap frame, the low-level assembly code calls up into a generic C trap handler called **trap()**, which is passed a pointer to the trapframe. This trap handler is called upon all types of interrupts and traps, and thus check the trap number field of the trap frame (trapno) to determine what to do. The first check is for the system call trap number (T_SYSCALL, or 64 as defined somewhat arbitrarily in traps.h), which then handles the system call, as you see here:

```
# vectors.S sends all traps here.
.globl alltraps
alltraps:
# Build trap frame.
pushl %ds
pushl %es
pushal
# Set up data segments.
movl $SEG_KDATA_SEL, %eax
movw %ax,%ds
movw %ax,%es
# Call trap(tf), where tf=%esp
pushl %esp
call trap
addl $4, %esp
```

File: trapasm.S

```
// FILE: trap.c
void
trap(struct trapframe *tf)
{
if(tf->trapno == T_SYSCALL){
if(cp->killed)
exit();
cp->tf = tf;
syscall();
if(cp->killed)
exit();
return;
}
... // continues
}
```

The code isn't too complicated. It checks if the current process (that made the system call) has been killed; if so, it simply exits and cleans up the process (and thus does not proceed with the system call). It then calls syscall() to actually perform the system call; more details on that below. Finally, it checks whether the process has been killed again before returning. Note that we'll follow the return path below in more detail.

```
Void syscall(void)
{
int num;
num = cp->tf->eax;
if(num >= 0 && num < NELEM(syscalls) && syscalls[num])
cp->tf->eax = syscalls[num]();
else {
cprintf("%d %s: unknown sys call %d\n", cp->pid, cp->name, num);
cp->tf->eax = -1;
}
}
```

File: syscall.c

6 Vectoring To The System Call

Once we finally get to the **syscall()** routine in **syscall.c,** not much work is left to do.

The system call number has been passed to us in the register %eax, and now we unpack that number from the trap frame and use it to call the appropriate routine as defined in the system call table syscalls[].

Pretty much all operating systems have a table similar to this to define the various system calls they support. After carefully checking that the system call number is in bounds, the pointed to routine is called to handle the call. For example, if the system call read() was called by the user, the routine sys read() will be invoked here.

The return value, you might note, is stored in %eax to pass back to the user.

```
static int (*syscalls[])(void) = {
[SYS_chdir] sys_chdir,
[SYS_close] sys_close,
[SYS_dup] sys_dup,
[SYS_exec] sys_exec,
[SYS_exit] sys_exit,
[SYS_fork] sys_fork,
[SYS_fstat] sys_fstat,
[SYS_getpid] sys_getpid,
[SYS_kill] sys_kill,
[SYS_link] sys_link,
[SYS_mkdir] sys_mkdir,
[SYS_mknod] sys_mknod,
[SYS_open] sys_open,
[SYS_pipe] sys_pipe,
[SYS_read] sys_read,
[SYS_sbrk] sys_sbrk,
[SYS_sleep] sys_sleep,
[SYS_unlink] sys_unlink,
[SYS_wait] sys_wait,
[SYS_write] sys_write,
};
```

The return path is pretty easy. First, the system call returns an integer value, which the code in syscall() grabs and places into the eax field of the trapframe. The code then returns into trap(), which simply returns into where it was called from in the assembly trap handler.

7 The Return Path

```
# Return falls through to trapret...
.globl trapret
trapret:
popal
popl %es
popl %ds
addl $0x8, %esp # trapno and errcode
iret
```

File: trapasm.S

This return code doesn't do too much, just making sure to pop the relevant values off the stack to restore the context of the running process. Finally, one more special instruction is called: **iret**, or the **return-from-trap** instruction. This instruction is similar to a return from a procedure call, but simultaneously lowers the privilege level back to user mode and jumps back to the instruction immediately following the int instruction called to invoke the system call, restoring all the state that has been saved into the trap frame. At this point, the user stub for read() is run again, which just uses a normal return-from-procedure-call instruction (ret) in order to return to the caller.

Session 13

CONTEXT OF A PROCESS

- In this session, I want to talk about the ideas behind context switching, then I want to look at the xv6 implementation of context switching.
- you all know that xv6 is an educational reimplementation of the Unix V6 kernel with a very well written and well documented source base.

Context of a Process

Why Context switch ?

- An operating system runs more processes than it has processors
- Needs some plan to time share the processors between the processes
- **Virtual Concurrency:** It's the practice of having multiple "processes", and switching between them really, really quickly to create the illusion that multiple things are happening at the same time.
- A common approach is to provide each process with a virtual processor – An illusion that it has exclusive access to the processor
- It is then the job of the OS to multiplex these multiple virtual processors on the underlying physical processors
- Context switching is the method an operating system employs to implement "multitasking
- It's also a good method for efficiently dealing with processes that need to wait for an IO request, to return. While one process is waiting, another can execute.

Implementation challenges

Q: How to switch from one process to another?

A: Context switching

Q: How to make context switching transparent?

A: Timer interrupts

Q: How to switch among processes running concurrently?

A: Locking

Q: How to coordinate processes?

A: Sleep on events (e.g., pipe, child exit)

Context of a Process

Context of a Unix process

1. user-level
2. register
3. system-level

1. User-level context

1. process text (instructions)
2. data
3. user stack
4. any shared memory

2. Register context

1. program counter - next instruction to run
2. processor status register
3. pointer to user or kernel stacks
4. general registers

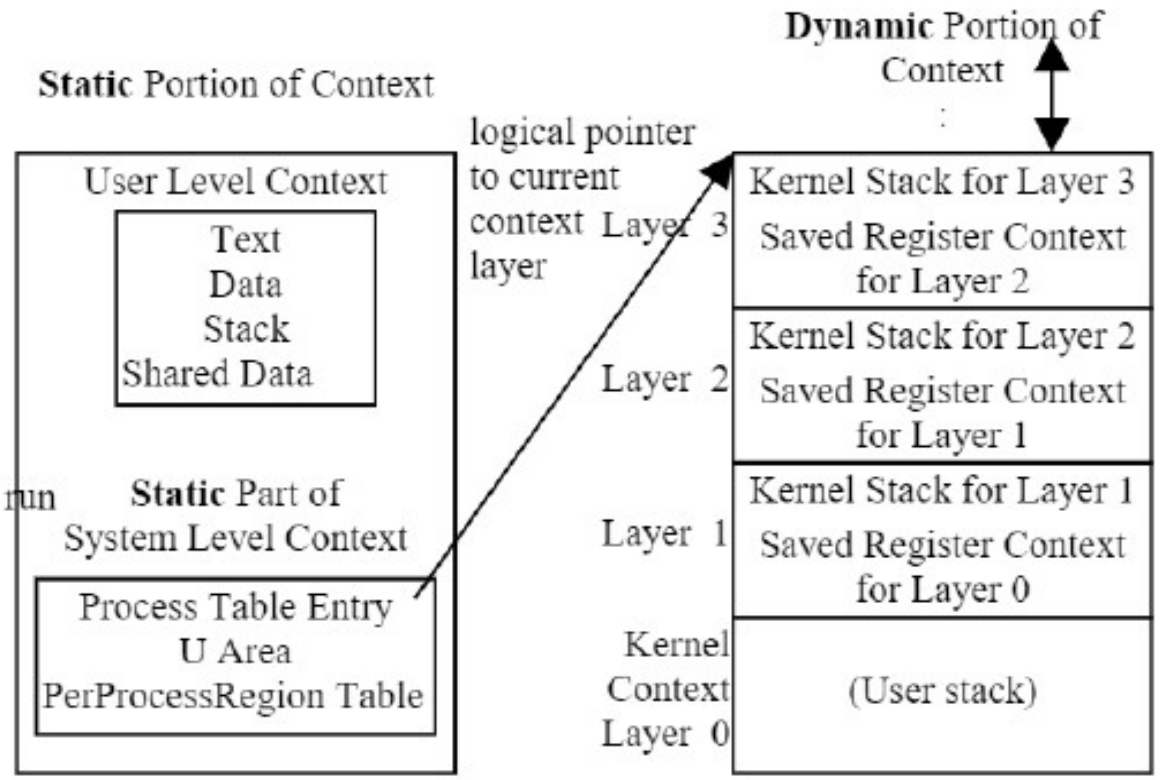
3. System-level context:

static part - for the P's lifetime

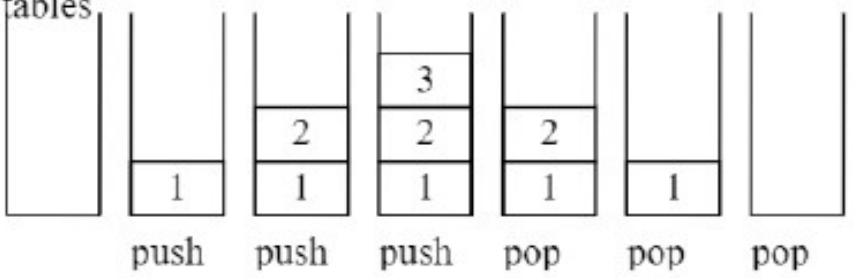
1. P table entry
2. u area
3. Per Process region (Pregion) entries, region tables, page tables

dynamic part - changes over P's lifetime

1. kernel stack of current layer
2. restorable registers of previous system context layers



Components of the Context of a Process



SAVING THE CONTEXT OF A POCESS

the kernel saves the context of a process whenever it pushes a new system context layer.

Saving of context layer happens when.....

1. the system receives an interrupt.
2. when a process executes a system call .
3. when the kernel does a context switch.

This session considers each case in detail. Interrupt, system call interface, context switch.

Interrupts and Exceptions:

- An event that requires the CPU to stop the current program execution and perform some service related to the event.
- **A simple analogy** – Reading a book and the phone rings – Stop reading and get the phone – Talk.. – Return to the book where one read and resume to read
- The phone call is an interrupt and the talk is an interrupt service routine (ISR) or an interrupt handler.
- The system is responsible for handling interrupts, whether they result from hardware (such as from the clock or from peripheral devices), from a programmed interrupt (execution of instructions designed to cause "software interrupt", or from exceptions (such as page faults).
- If the CPU is executing at a lower processor execution level than the level of the interrupt it accepts the interrupt before decoding the next instruction and raises the processor execution level, so that no other interrupts of the level (or lower level) can happen while it handles the current interrupt, preserving the integrity of the kernel data structure.

How The kernel handles the interrupt ?

1. It saves the current register context of the executing process and creates (pushes) a new context layer.
2. It determines the "source" or cause of the interrupt, identifying the type of interrupt (such as clock or disk) and the unit number of the interrupt, if applicable (such as which disk drive caused the interrupt).
3. When the system receives an interrupt, it gets a number from the machine that it uses as an offset into a table known as interrupt vector.
4. The contents of interrupt vector vary from machine to machine, but they usually contain the address of the interrupt handler for the corresponding interrupt source, and a way of finding a

INT Num	INT Handler
0	Clockintr
1	diskintr
2	ttyintr
3	devintr
4	softintr
5	Otherintr

Sample INT Vector

The kernel invokes the interrupt handler.

- The kernel stack for the new context layer is logically distinct from the kernel stack of the previous context layer.
- Some implementations use the kernel stack of the executing process to store the interrupt handler stack frames, and other implementations use a global interrupt stack to store the frames for the interrupt handlers that are guaranteed to return without switching context.
- The kernel executes a machine-specific sequence of instructions that restores the register context and kernel stack of the previous context layer, as they existed at the time of the interrupt and then resumes execution of the restored context layer.
- The behavior of the process may be affected by the interrupt handler, since the interrupt handler may have altered the global kernel data structures and awakened sleeping process.
- Usually the process continues execution as if the interrupt had never happened.

```
algorithm inthand /*handle interrupts*/
input : none
output : none
{
    save (push) current context layer;
    determine interrupt source;
    find interrupt vector;
    call interrupt handler;
    restore (pop) previous context layer;
}
```

Fig 10. Algorithm for Handling Interrupts

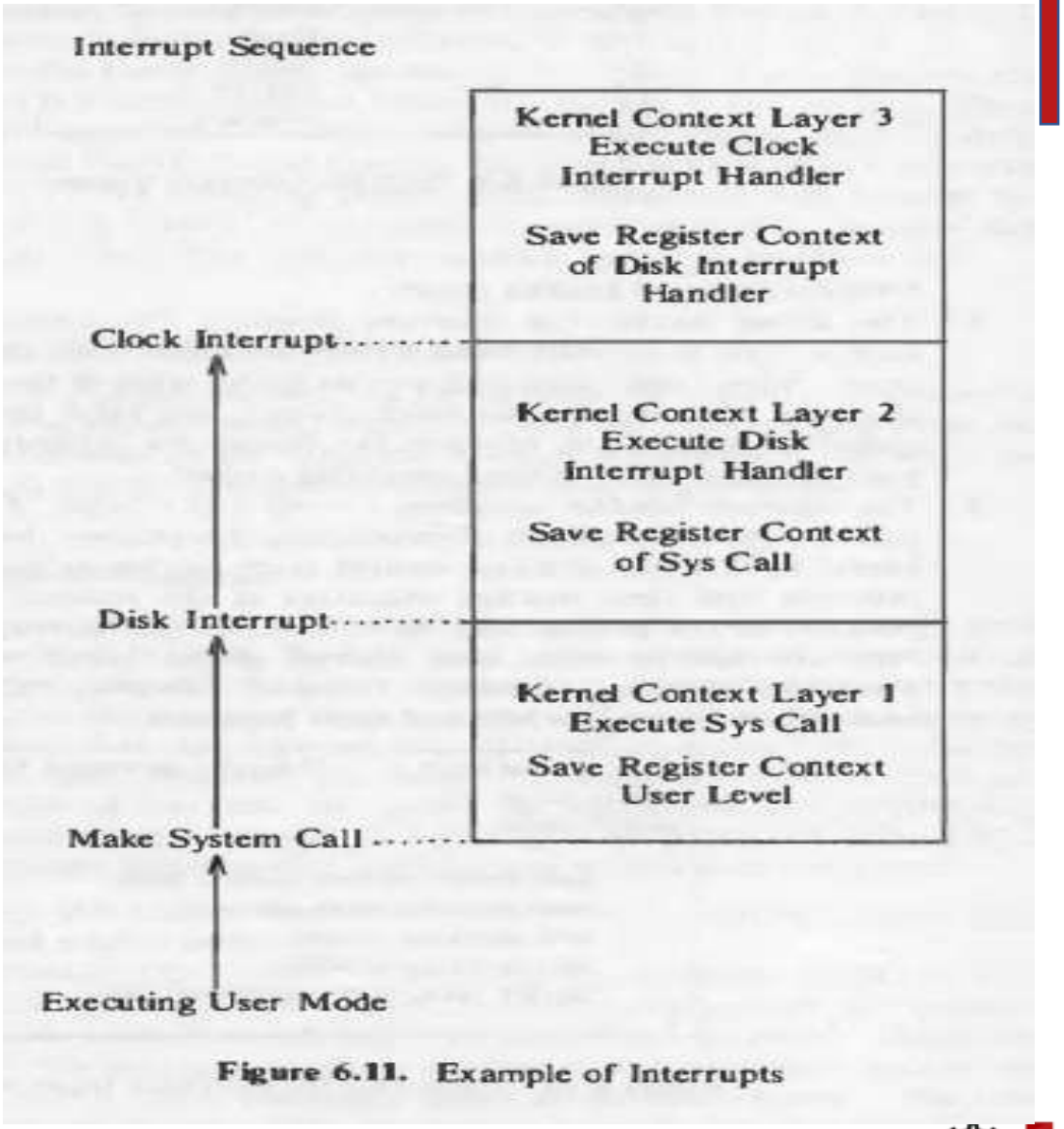


Figure 6.11. Example of Interrupts

System call Interface:

- The System call is a normal function call.
- The usual calling sequence cannot change the mode of a process from user to kernel.
- The C compiler uses a predefined library of functions that have the names of the system calls, thus resolving the system call references in the user program.
- The library functions typically invoke an instruction that changes the process execution mode to kernel mode and causes the kernel to start executing code for system calls.

```
algorithm syscall
input : system call number
output : result of system call
{

    find entry in system call table corresponding to system call number;
    determine number of parameters to system call;
    copy parameters from user address space to u area;
    save current context for abortive return;
    invoke system call code in kernel;
    if(error during execution of system call)
    {
        set register 0 in user saved register context to error number;
        turn on carry bit in PS register in user saved register context;
    }
    else
        set register 0,1 in user saved register context to return values from system call;
}
```

Fig 12. Algorithm for System Calls

Saving context for abortive returns

- Situations arise when the kernel abort its current execution sequence and immediately execute out of a previously saved context.

Mechanisms for executing a previous context.

The algorithm to save a context is **setjmp** and the algorithm to restore the context is **longjmp**

The method is identical to the function save context , except that save context pushes a new context layer where as setjmp stores the saved context in the **u area** and continues to execute in the old context layer.

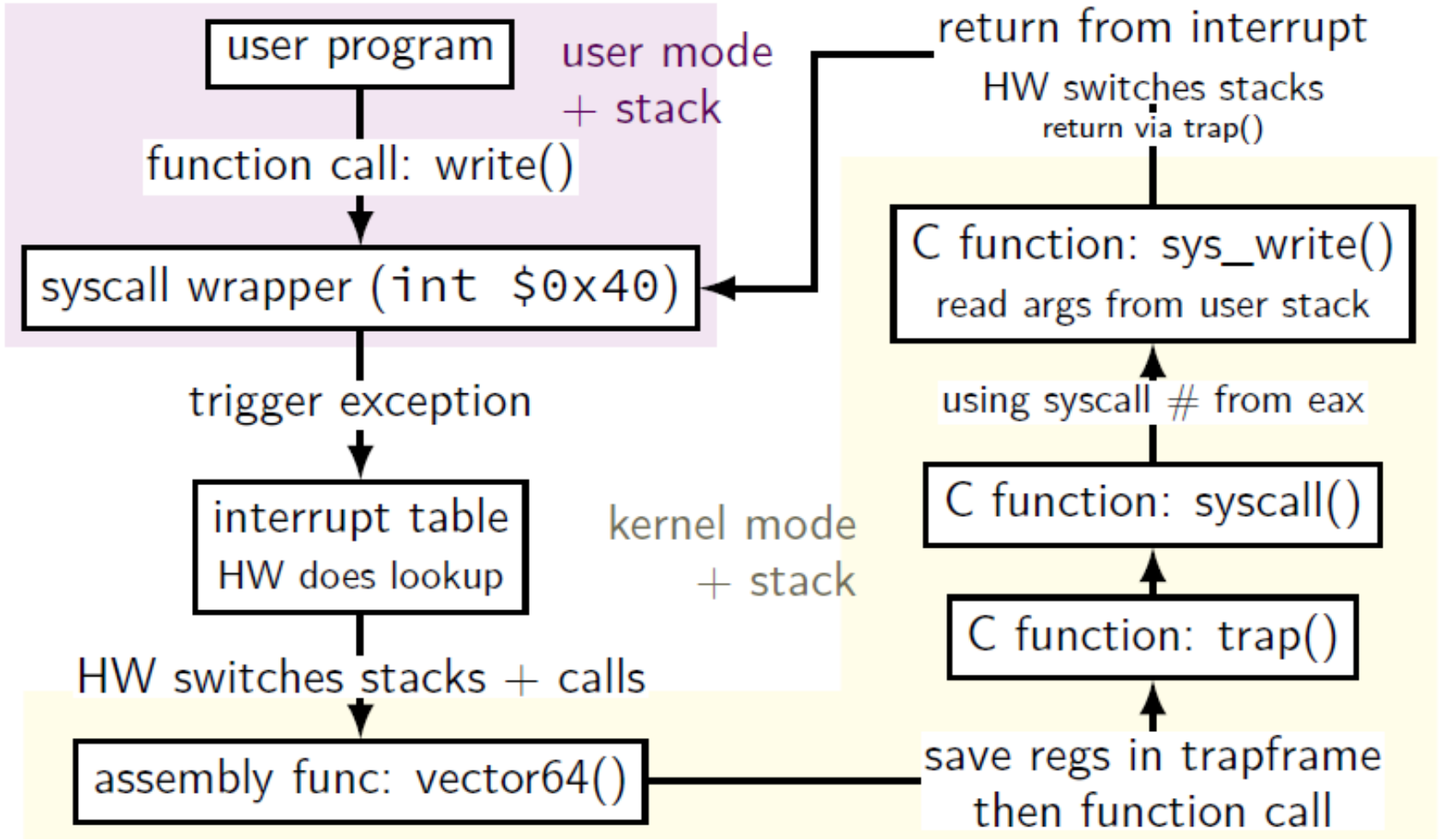
When the kernel wishes to resume the context it had saved in **setjmp** ,it does a longjmp, restoring its context from the u area and returning a 1 from setjmp.

Copying data between system and user address space

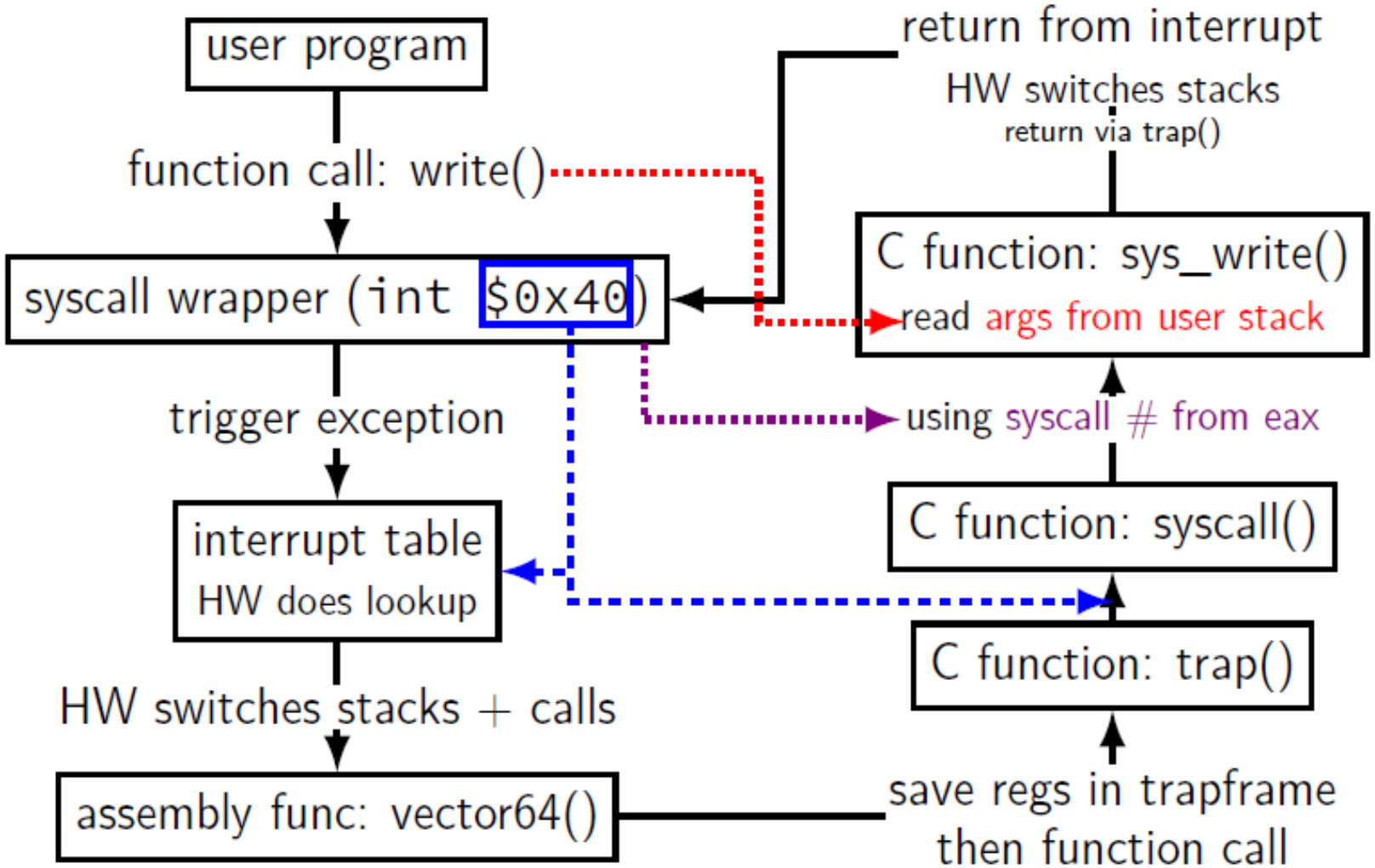
- A process executes in kernel mode or in user mode with no overlap of modes.
- Many system calls move data between kernel and user space, such as when copying system call parameters from user to kernel space or when copying data from I/O buffers in the read system call.
- Many machines allow the kernel to reference addresses in user space directly.
- The kernel must ascertain, that the address being read or written is accessible as if it had been executing in user mode;
- Otherwise it could override the ordinary protection mechanisms and inadvertently read or write addresses outside the user address space (possibly kernel data structure).

XV6 DESIGN AND IMPLEMENTTION OF CONTEXT SWITCH

write syscall in xv6



write syscall in xv6



Several CPU registers are referenced in the discussion of operating system concepts. Here are some names of x86 registers that we will find useful.

1. EAX, EBX, ECX, EDX, ESI, and EDI are general purpose registers used to store variables during computations.
2. The general purpose registers EBP and ESP store the base and top of the current stack frame.
3. The program counter (PC) is also referred to as EIP (instruction pointer).
4. The segment registers CS, DS, ES, FS, GS, and SS store pointers to various segments (e.g., code segment, data segment, stack segment) of the process memory.
5. The control registers like CR0 hold control information. For example, the CR3 register holds the address of the page table of the current running process, which is used to translate virtual addresses to physical addresses.

Of these registers, EAX, ECX, and EDX are called **caller-save registers**, and the rest (EBX, ESI, EDI, ESP, EBP, EIP) are **callee-save registers.**

The callee-save registers must be preserved across function calls and context switches. Suppose an executing process pushes a new stack frame onto a stack during a function call, or the CPU moves away to another process. When the function call returns, or the CPU is switched back to the process, the callee-save registers must have the same values as before the event. The ESP register should be pointing to the old stack as before, and EIP should contain the return address from where the process resumes execution. The EAX register is used to pass arguments and return values.

The caller save registers may have changed, and no assumptions would be made of them.

The Basics

Context switching mechanisms are platform specific. Depending on what processor architecture you're using, your context switching will look very different. Here we are covering x86..

The general idea is that your processor has a state, which is the current contents of all of its registers. Each process will contain a copy of the state that it's in when it gets swapped off the processor, and when it later gets swapped back on that state will get restored. This allows processes to be paused and resumed at a later time.

Your processor doesn't have any knowledge of processes. The processor only really executes one set of instructions at a time in a (mostly) linear fashion, so how do we convince it to jump between multiple processes? We need to have a list of processes, each containing processor state. We also need to have a means of jumping out of a running process and doing all of the logic necessary to swap it off the processor and swap its successor on.

What is a process?

From a high level, we all know that a process is a running program in the system. It contains some binary executable code, some data, it can be sent signals, it can spawn children and it can die. But what data makes this possible? Let's take a look at what xv6 considers enough information to represent a process:

```
2336 // Per−process state
2337 struct proc {
2338 uint sz; // Size of process memory (bytes)
2339 pde_t* pgdir; // Page table
2340 char *kstack; // Bottom of kernel stack for this process
2341 enum procstate state; // Process state
2342 int pid; // Process ID
2343 struct proc *parent; // Parent process
2344 struct trapframe *tf; // Trap frame for current syscall
2345 struct context *context; // swtch() here to run process
2346 void *chan; // If non−zero, sleeping on chan
2347 int killed; // If non−zero, have been killed
2348 struct file *ofile[NOFILE]; // Open files
2349 struct inode *cwd; // Current directory
2350 char name[16]; // Process name (debugging)
2351 };
```

Lots of interesting things in there. A lot of them are pretty obvious, such as the process ID, the size of the process memory, the parent process and so on. Other things are a little less obvious. Why does it need a kernel stack, for example? And what's a trap frame?

Kernel Stacks

Every process in xv6 needs a kernel stack for handling system calls. If, for example, a process gets stopped mid-way through a system call and a new process starts, it would be a really bad idea if the new process started in the middle of a shared kernel stack. It would be equivalent to all user processes sharing one big stack, and because sharing stacks is a bad idea, each process has its own user and kernel stack.

Another reason it's really useful to have a kernel stack for each user-mode process is that even if the user mangles/destroys its user-mode stack, the kernel stack will be intact and the operating system will be able to handle the failure gracefully.

Interrupts and Trap Frames

An integral part of the x86 architecture is its interrupt system. The processor can, at any time and for a variety of reasons, be interrupted and asked to do something else. Examples of types of interrupts include divide by zero, access to memory that doesn't belong to you or is protected, IO requests returning, keyboard events and a special "timer" interrupt that happens periodically. This is how "pre-emptive process scheduling" is implemented.

There is a piece of hardware inside your computer called a "programmable interval timer"*, which can be told to generate an interrupt after a given amount of time. In xv6, it is told to fire every 100 milliseconds, and xv6 has set the PIT interrupt number to point at its process scheduling code, which handles context switches.

The trap frame stored in the **proc** struct, however, is concerned only with system calls. xv6 implements its system calls in a similar way to Linux, by having the software interrupt number 64 be handled as a system call. The user stores the system call number into the **eax** register, fires an **int 64**, the processor looks up the handler for that interrupt, and then the OS transitions into kernel mode.

Part of the kernel mode transition involves creating a "trapframe", which looks like this (**x86.h** file in xv6):

```
0600 // Layout of the trap frame built on the stack by the
0601 // hardware and by trapasm.S, and passed to trap().
0602 struct trapframe {
0603 // registers as pushed by pusha
0604 uint edi;
0605 uint esi;
0606 uint ebp;
0607 uint oesp; // useless & ignored
0608 uint ebx;
0609 uint edx;
0610 uint ecx;
0611 uint eax;
0612
0613 // rest of trap frame
0614 ushort gs;
0615 ushort padding1;
0616 ushort fs;
0617 ushort padding2;
0618 ushort es;
0619 ushort padding3;
0620 ushort ds;
0621 ushort padding4;
0622 uint trapno;
0623
0624 // below here defined by x86 hardware
0625 uint err;
0626 uint eip;
0627 ushort cs;
0628 ushort padding5;
0629 uint eflags;
0630
0631 // below here only when crossing rings,
such as from user to kernel
0632 uint esp;
0633 ushort ss;
0634 ushort padding6;
0635 };
```

This is built on the stack for all interrupts that happen and serves as a way for the operating system to tell the processor where to jump back to after the interrupt has been handled.

When xv6 handles a system call, the current process's **proc->tf** is set to the interrupt trap frame, which contains the register set that will get restored to the processor when the system call retuns.

This way, system calls can interact with the register set and give return values back to user mode.

System call code in xv6, found in **syscall.c**:

```
3700 void
3701 syscall(void)
3702 {
3703 int num;
3704 struct proc *curproc = myproc();
3705
3706 num = curproc−>tf−>eax;  // Get the syscall number from eax
  // Check that the syscall number is greater than 0, inside the range of total
  // syscalls and that a function pointer exists for that syscall number.
3707 if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    // Set eax to the return value of the syscall.
3708 curproc−>tf−>eax = syscalls[num]();
3709 } else {
    // Else print an error and return -1 to the user program.
3710 cprintf("%d %s: unknown sys call %d\n",
3711 curproc−>pid, curproc−>name, num);
3712 curproc−>tf−>eax = −1;
3713 }
3714 }
```

Disabling and re-enabling interrupts

Sometimes we don't want the processor to be interrupted because we may leave really important data structures in an inconsistent state. x86 offers two commands to deal with this called **cli** and **sti**, clear interrupts and set interrupts respectively. These two commands manipulate a bit inside the **eflags** register to tell the processor whether or not we are currently interested in handling interrupts.

When we've cleared interrupts, we stop handling them but they do get queued up internally by the processor. When we restore interrupts, the queued up interrupts start getting processed again. This makes sense. You wouldn't want to completely ignore them otherwise you would have processes waiting on IO results that have been thrown away.

In **spinlock.c** there are definitions for **pushcli()** and **popcli()**, which are a "matched" interface to the x86 **cli** and **sti** instructions. So it takes two calls to **popcli()** to restore interrupts after two calls to **pushcli()**. This makes it easier to handle the clearing of interrupts when multiple parts of the code base are calling **cli** and **sti**. Without them, you could end up with functions turning interrupts back on before other parts of the code base are ready for them.

spinlocks are declared in kern/spinlock.h, and support the following four operations:

1. void spinlock_init(spinlock *_lk_): initialize spinlock _lk_ to the unlocked state.
2. void spinlock_acquire(spinlock *_lk_): acquire spinlock _lk_, spinning if necessary until it becomes available.
3. void spinlock_release(spinlock *_lk_): release spinlock _lk_.
4. int spinlock_holding(spinlock *_lk_): return true (nonzero) if this CPU is holding spinlock _lk_, false (zero) otherwise.
5. pushcli :Disables interrupts, Increments a "nested call count", Remembers whether interrupts were originally enabled when ncli was 0
6. Popcli: Decrements nested call count. If now zero, enables interrupts if they were originally enabled.

0379 // spinlock.c
0380 void acquire(struct spinlock*);
0381 void getcallerpcs(void*, uint*);
0382 int holding(struct spinlock*);
0383 void initlock(struct spinlock*, char*);
0384 void release(struct spinlock*);
0385 void pushcli(void);
0386 void popcli(void);

If that's still unclear, imagine the following code:

```
void func1(void) {
  cli();
  // important stuff
  func2();
  // more important stuff
  sti();
  return;
}

void func2(void) {
  cli();
  // important stuff
  sti();
  return;
}

func1();
```

There is a gigantic problem in the above code. **func2()** is calling **sti()**, but when it returns, **func1()** still expects interrupts to be off and does some important things. To get around that problem, we would rewrite the above code like so:

```
void func1(void) {
  pushcli();
  // important stuff
  func2();
  // more important stuff
  popcli();
  return;
}

void func2(void) {
  pushcli();
  // important stuff
  popcli();
  return;
}
func1();
```

This way, interrupts would only be turned back on when the **popcli()** inside of **func1()** has been called.

Detailed steps for context switching

Let's envisage a scenario. Process 1 is running, process 2 is waiting to run. The PIT(programmable interval timer) throws an interrupt. This causes the processor to stop what it's doing, abandon process 1, create a trap frame and transition into kernel mode. When an interrupt is caught, there is a generic catch-all that sets up the trap frame and any trap information the processor has given us and then calls the C function **trap()**. Here's the assembler for that:

vector.S

The comment at the top of the above assembler snippet mentions **vector.S**, but you probably won't find that file inside the xv6 repo. Why?

```
3300 #include "mmu.h"
3301
3302 # vectors.S sends all traps here.
3303 .globl alltraps
3304 alltraps:
3305 # Build trap frame.
3306 pushl %ds
3307 pushl %es
3308 pushl %fs
3309 pushl %gs
3310 pushal
3311
3312 # Set up data segments.
3313 movw $(SEG_KDATA<<3), %ax
3314 movw %ax, %ds
3315 movw %ax, %es
3316
3317 # Call trap(tf), where tf=%esp
3318 pushl %esp
3319 call trap
3320 addl $4, %esp
3321
3322 # Return falls through to trapret...
3323 .globl trapret
3324 trapret:
3325 popal
3326 popl %gs
3327 popl %fs
3328 popl %es
3329 popl %ds
3330 addl $0x8, %esp # trapno and errcode
3331 iret
```

The reason is that x86 doesn't make interrupt management easy, and to handle all interrupts in a way that allows you to identify what interrupt happened you need to set up 256 interrupt handlers in assembly. xv6 handles this with a Perl script to generate the entry points that the IDT entries point to. Each entry pushes an error code if the processor didn't, pushes the interrupt number, and then jumps to alltraps. :

The important bit to note is that everything jumps into the **alltraps** assembler label seen above, which passes control into the **trap()** function in C.

```
3250 #!/usr/bin/perl −w
3251
3252 # Generate vectors.S, the trap/interrupt entry points.
3253 # There has to be one entry point per interrupt number
3254 # since otherwise there's no way for trap() to discover
3255 # the interrupt number.
3256
3257 print "# generated by vectors.pl − do not edit\n";
3258 print "# handlers\n";
3259 print ".globl alltraps\n";
3260 for(my $i = 0; $i < 256; $i++){
3261 print ".globl vector$i\n";
3262 print "vector$i:\n";
3263 if(!($i == 8 || ($i >= 10 && $i <= 14) || $i == 17)){
3264 print " pushl \$0\n";
3265 }
3266 print " pushl \$$i\n";
3267 print " jmp alltraps\n";
3268 }
3269
3270 print "\n# vector table\n";
3271 print ".data\n";
3272 print ".globl vectors\n";
3273 print "vectors:\n";
3274 for(my $i = 0; $i < 256; $i++){
3275 print " .long vector$i\n";
3276 }
```

Alltraps continues to save processor registers:
it pushes %ds, %es, %fs, %gs, and the general-purpose registers.
The result of this effort is that the kernel stack now contains a struct trapframe containing the processor registers at the time of the trap.
The processor pushes ss, esp, eflags, cs, and eip. The processor or the trap vector pushes an error number, and alltraps pushes the rest.

The trap frame contains all the information necessary to restore the user mode processor registers when the kernel returns to the current process, so that the processor can continue exactly as it was when the trap started. In the case of the first system call, the saved eip is the address of the instruction right after the int instruction. cs is the user code segment selector. eflags is the content of the eflags register at the point of executing the int instruction.

As part of saving the general-purpose registers, alltraps also saves %eax, which contains the system call number for the kernel to inspect later

Continuing the example...

We're now in process 1's kernel stack in the trap handler and we need to find a new process to run. Because this is a timer interrupt, the following bit of code inside of the **trap()** function in **trap.c** will run:

```
3471 // Force process to give up CPU on clock tick.
3472 // If interrupts were on while locks held, would need to check nlock.
3473 if(myproc() && myproc()−>state == RUNNING &&
3474 tf−>trapno == T_IRQ0+IRQ_TIMER)
3475 yield();
```

Let's dig into what **yield()** is doing (**proc.c**):

```
2826 // Give up the CPU for one scheduling round.
2827 void
2828 yield(void)
2829 {
2830 acquire(&ptable.lock);
2831 myproc()−>state = RUNNABLE;
2832 sched();
2833 release(&ptable.lock);
2834 }
```

So first we're acquiring a lock on the process table. This is to avoid race conditions when multiple processors are running scheduling code. Then we set the current process's state to **RUNNABLE**, and we call a function called **sched()** (in **proc.c**):

panic is the kernel's last resort: the impossible has happened and the kernel does not know how to proceed.

```
2800 // Enter scheduler. Must hold only ptable.lock
2801 // and have changed proc->state. Saves and restores
2802 // intena because intena is a property of this
2803 // kernel thread, not this CPU. It should
2804 // be proc->intena and proc->ncli, but that would
2805 // break in the few places where a lock is held but
2806 // there's no process.
2807 void
2808 sched(void)
2809 {
2810 int intena;
2811 struct proc *p = myproc();
2812
2813 if(!holding(&ptable.lock))  // If we aren't holding the ptable lock, panic
2814 panic("sched ptable.lock");
2815 if(mycpu()->ncli != 1)   // If we aren't 1 pushcli level deep, panic
2816 panic("sched locks");
2817 if(p->state == RUNNING)  // If the current process is in the running state, panic
2818 panic("sched running");
2819 if(readeflags()&FL_IF)   // If the processor can be interrupted, panic
2820 panic("sched interruptible");
2821 intena = mycpu()->intena;
2822 swtch(&p->context, mycpu()->scheduler);
2823 mycpu()->intena = intena;
2824 }
```

The majority of **sched()** is concerned with making sure it's safe to schedule a new process. First it checks that we're holding the process table lock, then it checks to make sure we're only one **pushcli()** level deep, then it makes sure the process being swapped off is not still running, then it checks if interrupts have been cleared. The last check feels a lot like a "just in case", because in theory the **cpu->ncli** check should cover it.

If any of the above is true, we cannot safely schedule a new process and the kernel panics.

The **cpu->intena** is a variable that stores whether or not interrupts were enabled before a **pushcli** call. The only place in the xv6 code that **cpu->intena** used in **spinlock.c**

The next important call inside of **sched()** is the call to **swtch()**. This is the central/key of the operation, where the actual context switching happens. It passes in a pointer to the current process's context, so that the current registers can be saved, and it passes in the scheduler's context to be switched to. It makes sense, in that case, that this part is implemented in assembler (**swtch.S**):

```
3050 # Context switch
3051 #
3052 # void swtch(struct context **old, struct context *new);
3053 #
3054 # Save the current registers on the stack, creating
3055 # a struct context, and save its address in *old.
3056 # Switch stacks to new and pop previously−saved registers.
3057
3058 .globl swtch
3059 swtch:
3060 movl 4(%esp), %eax
3061 movl 8(%esp), %edx
3062
3063 # Save old callee−saved registers
3064 pushl %ebp
3065 pushl %ebx
3066 pushl %esi
3067 pushl %edi
3068
3069 # Switch stacks
3070 movl %esp, (%eax)
3071 movl %edx, %esp
3072
3073 # Load new callee−saved registers
3074 popl %edi
3075 popl %esi
3076 popl %ebx
3077 popl %ebp
3078 ret
```

SWTCH

void swtch(struct context, struct context*);**

- Saves and restores *contexts*
- Takes two arguments: `struct context **old` and `struct context *new`
  - Replaces the former with the latter
- Each time a process has to give up the CPU, its kernel thread invokes `swtch` to save its own context and switch to the scheduler context
- Context is a struct context*, stored on the kernel stack
- CPU pushed onto stack and saves stack pointer to *old
- Copies new to %esp, pops previous registers, and returns

This is what we've been looking for. **swtch()** takes two context structs, that look like this (**proc.h**):

It contains only the registers necessary for performing a context switch, and you can see in **swtch()** that all we need to do is push the current context values, switch the stack pointer, then pop the context values from the new stack into the appropriate registers. Magic.

The rest of the process is less concerned with context switching and more concerned with picking a new process to run, which is a completely different kettle of fish.

```
2316 // Saved registers for kernel context switches.
2317 // Don't need to save all the segment registers (%cs, etc),
2318 // because they are constant across kernel contexts.
2319 // Don't need to save %eax, %ecx, %edx, because the
2320 // x86 convention is that the caller has saved them.
2321 // Contexts are stored at the bottom of the stack they
2322 // describe; the stack pointer is the address of the context.
2323 // The layout of the context matches the layout of the stack in swtch.S
2324 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
2325 // but it is on the stack and allocproc() manipulates it.
2326 struct context {
2327 uint edi;
2328 uint esi;
2329 uint ebx;
2330 uint ebp;
2331 uint eip;
2332 };
```

THANK YOU

SLEEP & WAKEUP

Session 14

In this session, we will understand and explore the design of sleep and wakeup algorithms

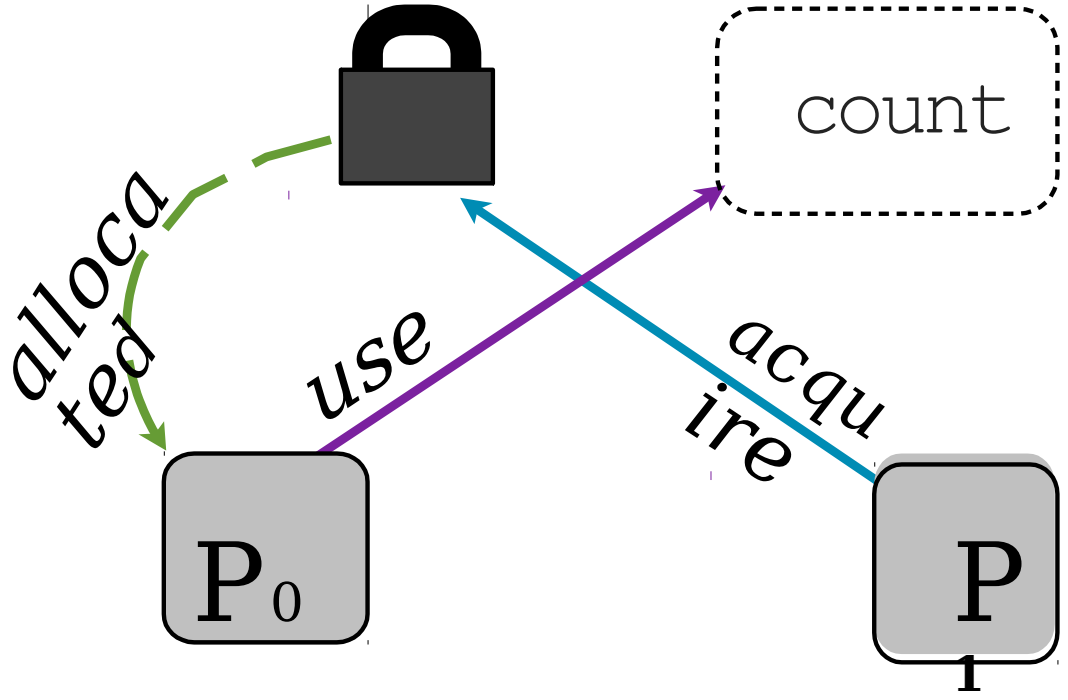
Synchronization

Program 0

```
a1  count = count + 1
```

Program 1

```
b1  count = count - 1
```



Motivating Scenario

shared variable

```
int counter=5;
```

program 0

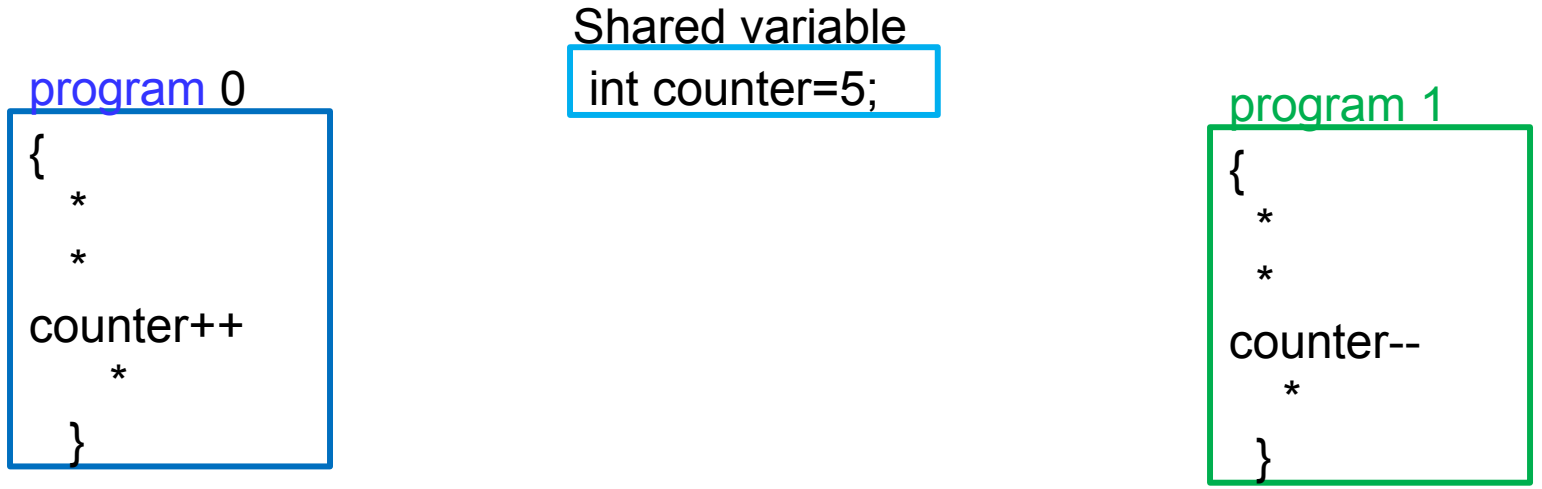
```
{
  *
  *
counter++
    *
  }
```

program 1

```
{
  *
  *
counter--
  *
}
```

- Single core
  - Program 1 and program 2 are executing at the same time but sharing a single core

Motivating Scenario



- What is the value of counter?
  - expected to be 5
  - but could also be 4 and 6

Motivating Scenario

Shared variable

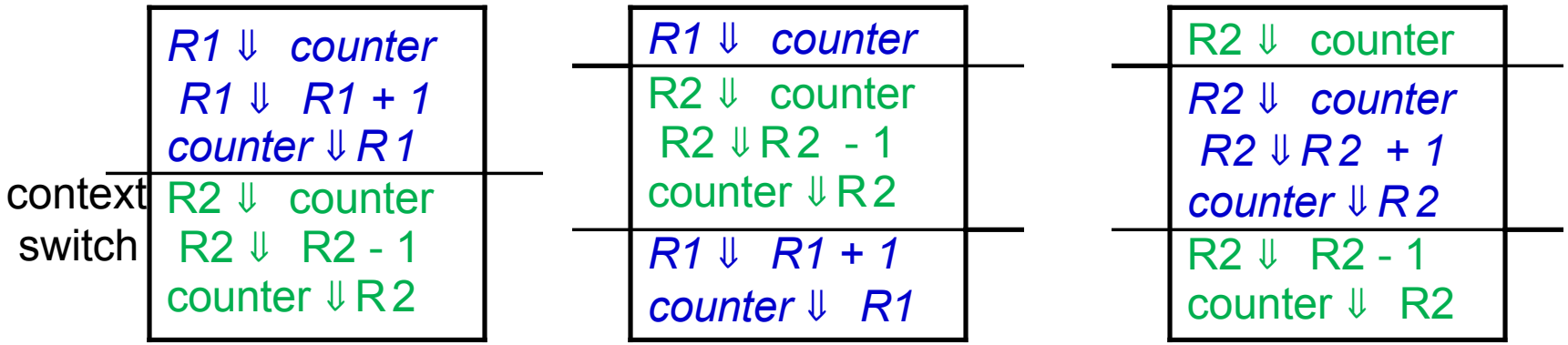
```
int counter=5;
```

program 0

```
{
   *
   *
counter++
   *
}
```

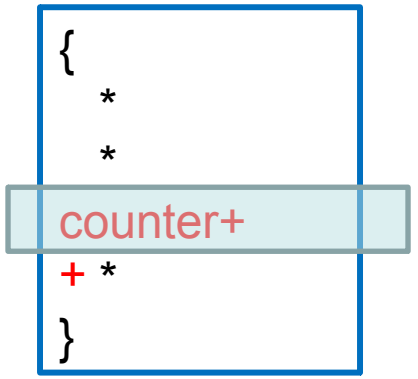
program 1

```
{
  *
  *
counter--
   *
  }
```



Race Conditions

- Race conditions
  - A situation where several processes access and manipulate the same data (*critical section*)
  - The outcome depends on the order in which the access take place
  - Prevent race conditions by synchronization
    - Ensure only one process at a time manipulates the critical data



critical section

No more than one process should execute in critical section at a time

Critical Section

- Requirements
  - **Mutual Exclusion :** No more than one process in critical section at a given time
  - **Progress :** When no process is in the critical section, any process that requests entry into the critical section must be permitted without any delay
  - **No starvation (bounded wait):** There is an upper bound on the number of times a process enters the critical section, while another is waiting.

Locks and Unlocks

shared variable

```
int counter=5;
 lock_t L;
```

program 0

```
{
   *
   *
lock(L)
counter++
 unlock(L)
      *
   }
```

program 1

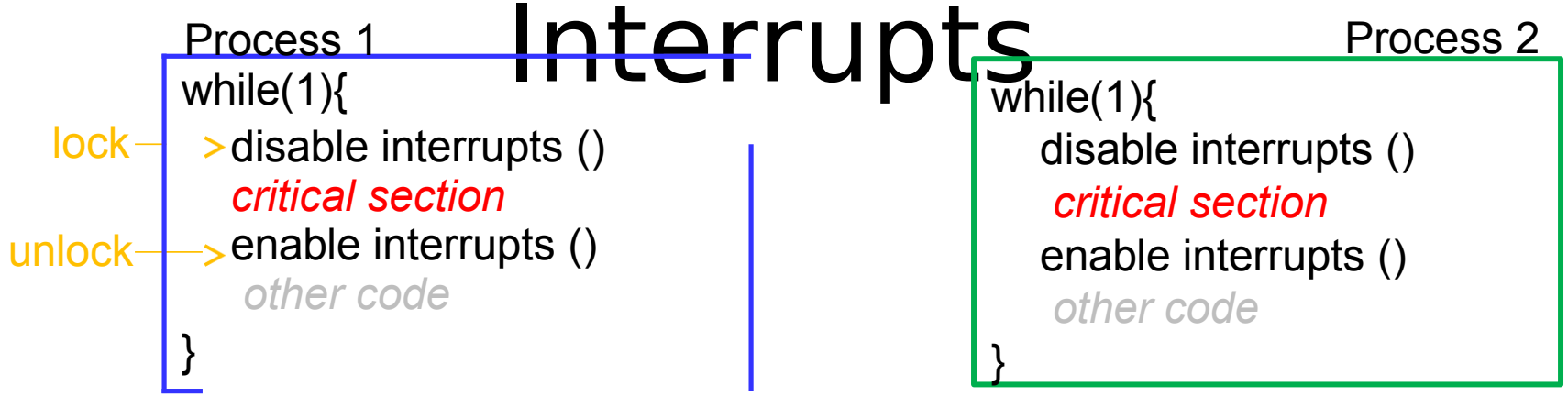
```
{
  *
  *
lock(L)
counter--

unlock(L)
    *
  }
```

- lock(L) : acquire lock L exclusively
  - Only the process with L can access the critical section
- unlock(L) : release exclusive access to lock L
  - Permitting other processes to access the critical section

How to Implement Locking

Using Interrupts



- Simple
  - When interrupts are disabled, context switches won't happen
- Requires privileges
  - User processes generally cannot disable interrupts
- Not suited for multicore systems

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations
  - wait (S) {

```
        while S <= 0
                ; // no-op
          S--;
    }
```

  - signal (S) {

```
      S++;
    }
```

Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion
  - Semaphore S; // initialized to 1
  - wait (S);

    Critical Section

    signal (S);

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list

- Two operations:
  - Sleep/block – place the process invoking the operation on the appropriate waiting queue.
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue.

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

```
wait (S){
    value--;
    if (value < 0) {
        add this process to waiting queue
        sleep()/block();  }
}
```

- Implementation of signal:

```
Signal (S){
    value++;
    if (value <= 0) {
        remove a process P from the waiting queue
        wakeup(P);  }
}
```

Deadlock and Starvation

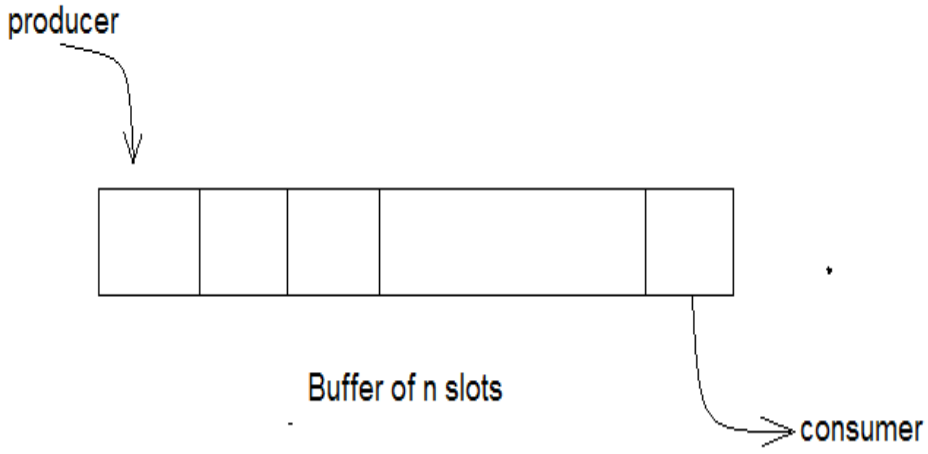
- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Bounded-Buffer Problem

- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.



The structure of the producer process

```
while (true)  {

        //   produce an item

    wait (empty);
    wait (mutex);

        //  add the item to the  buffer

    signal (mutex);
    signal (full);
}
```

The structure of the consumer process

```
while (true) {
        wait (full);
        wait (mutex);

            //  remove an item from  buffer

        signal (mutex);
        signal (empty);

            //  consume the removed item
}
```

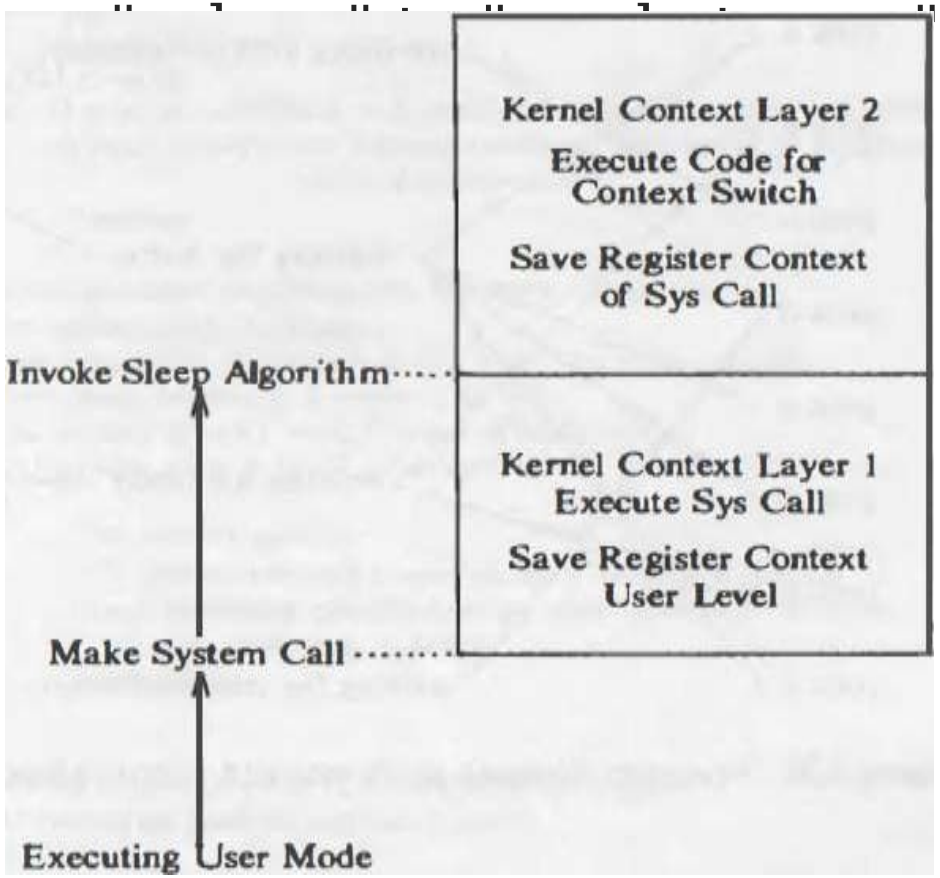
Sleep and Wakeup
(Producer Consumer problem)

```
1.#define N 100 //maximum slots in buffer
2.#define count=0 //items in the buffer
3.void producer (void)
4.{
5.    int item;
6.    while(True)
7.    {
8.        item = produce_item(); //producer produces an item
9.        if(count == N) //if the buffer is full then the producer will sleep Sleep();
10.         insert_item (item); //the item is inserted into buffer
11.         countcount=count+1;
12.         if(count==1) //The producer will wake up the
13.         //consumer if there is at least 1 item in the buffer
14.         wake-up(consumer);
15.    }
16.}
```

```
1.void consumer (void)
2.{
3.    int item;
4.    while(True)
5.    {
6.        {
7.            if(count == 0) //The consumer will sleep if the buffer is empty.sleep();
8.            item = remove_item();
9.            countcount = count - 1;
10.             if(count == N-1) //if there is at least one slot available in the buffer
11.             //then the consumer will wake up producer
12.             wake-up(producer);
13.             consume_item(item); //the item is read by consumer.
14.       }
15.   }
16.}
```

Slee
p

- The algorithms for sleep, which changes the process state from "kernel running" to "asleep in memory," and wakeup, which changes the process state from [illegible] in memory or swapped.



When a process goes to sleep, it typically does so during execution of a **system call**: The process enters the kernel (context layer I) when it executes an operating system trap and goes to sleep awaiting a resource.

When the process goes to sleep, it does a **context switch**, pushing its current context layer and executing in kernel context layer 2.

Processes also go to sleep when they [illegible] **page faults** as a result of

Sleep

```
algorithm sleep
input:  (1) sleep address
        (2) priority
output: 1 if process awakened as a result of a signal that process catches,
        longjump algorithm if process awakened as a result of a signal
                        that it does not catch,
        0 otherwise;
{
    raise processor execution level to block all interrupts;
    set process state to sleep;
    put process on sleep hash queue, based on sleep address;
    save sleep address in process table slot;
    set process priority level to input priority;
    if (process sleep is NOT interruptible)
    {
        do context switch;
        /* process resumes execution here when it wakes up */
        reset processor priority level to allow interrupts as when
                process went to sleep;
        return(0);
    }
    /* here, process sleep is interruptible by signals */
    if (no signal pending against process)
    {
        do context switch;
        /* process resumes execution here when it wakes up */
        if (no signal pending against process)
        {
            reset processor priority level to what it was when
                process went to sleep;
            return(0);
        }
    }
    remove process from sleep hash queue, if still there;

    reset processor priority level to what it was when process went to sleep;
    if (process sleep priority set to catch signals)
        return(1)
    do longjmp algorithm;
}
```

Steps of sleep() algorithm

1. Kernel first raises the process execution level to block all interrupts.
2. Marks the state of process as "ASLEEP" and makes appropriate entry in process table.
3. Kernel maintains a hash queue of currently slept process. So kernel keeps this newly process on the hash queue.
4. Kernel saves the "sleep address" in process table and also sets and saves the process priority level to "input priority" allows the process to catch signals if process's sleep is going to be interruptible by signals.
5. If process's sleep state is not interruptible by any signals.
   In such case process is going to sleep and hence "Context switch" is allowed.
   So kernel saves context layer of this process (by setjmp) and does a context switch to start context layer of another, scheduled process.
   Now process safely sleeps and when awakes enter in to ready to run state.
   After coming back to sleep algorithm kernel restores process previous execution level to allow usual interrupts and returns 0 as its exit status of sleep algorithm.
6. If process's sleep state is interruptible .Kernel always check for pending signals for this sleeping process.
   If no signals is pending for the process then..
   So kernel saves context layer of this process (by setjmp) and does a context switch to start context layer of another, scheduled process.
   Now process safely sleeps and when awakes enter in to ready to run state.
   After coming back to sleep algorithm kernel again checks for pending signals .If still signals are not pending for the process then….
   restores process previous execution level to allow usual interrupts and returns 0 as its exit status of sleep algorithm.
7. If process's sleep state is interruptible .Kernel always check for pending signals for this sleeping process.
   If signal is pending for the process then..
   a. if "priority" is set above the threshold then process will not think about the signal and will wake up only when required event occurs. So it will wakeup by kernel with explicit wakeup call.
   b. But if "sleep priority" is below the "threshold" then process actually does not go to sleep but respond the arrived signals as if it is sleeping

Wakeup Algorithm

- To wake up sleeping processes, the kernel executes the wakeup algorithm, either during the usual system call algorithms or when handling an interrupt.
- For instance, the algorithm iput releases a locked inode and awakens all processes waiting for the lock to become free.
- Similarly, the disk interrupt handler awakens a process waiting for I/O completion. The kernel raises the processor execution level in

```
algorithm wakeup              /* wake up a sleeping process */
input:  sleep address
output: none
{
        raise processor execution level to block all interrupts;
        find sleep hash queue for sleep address;
        for (every process asleep on sleep address)
        {
                remove process from hash queue;
                mark process state "ready to run";
                put process on scheduler list of processes ready to run;
                clear field in process table entry for sleep address;
                if (process not loaded in memory)
                        wake up swapper process (0);
                else if (awakened process is more elligible to run than
                                        currently running process)
                        set scheduler flag;
        }
        restore processor execution level to original level;
}
```

Locks (i.e., spinlocks) in xv6 are implemented using the xchg atomic instruction. The function to acquire a lock disables all interrupts, and the function that releases the lock re-enables them.

spinlocks are declared in kern/spinlock.h, and support the following four operations:

1. void spinlock_init(spinlock *lk): initialize spinlock *lk* to the unlocked state.
2. void spinlock_acquire(spinlock *lk): acquire spinlock *lk*, spinning if necessary until it becomes available.
3. void spinlock_release(spinlock *lk): release spinlock *lk*.
4. int spinlock_holding(spinlock *lk): return true (nonzero) if this CPU is holding spinlock *lk*, false (zero) otherwise.
5. pushcli :Disables interrupts, Increments a "nested call count", Remembers whether interrupts were originally enabled when ncli was 0
6. Popcli: Decrements nested call count. If now zero, enables interrupts if they were originally enabled.

```
1500 // Mutual exclusion lock.
1501 struct spinlock {
1502 uint locked; // Is the lock held?
1503
1504 // For debugging:
1505 char *name; // Name of lock.
1506 struct cpu *cpu; // The cpu holding the
lock.
1507 uint pcs[10]; // The call stack (an array
of program counters)
1508 // that locked the lock.
1509 };
0379 // spinlock.c
0380 void acquire(struct
spinlock*);
0381 void getcallerpcs(void*,
uint*);
0382 int holding(struct spinlock*);
0383 void initlock(struct spinlock*,
char*);
0384 void release(struct
```

So, let us look at how the acquire function is implemented in so let us look at xv6 locks all right. So, let us look at how xv6 is implementing locks. That is the lock function, that is the acquire function basically, let see what the lock structure is before that. So, lock structure you know if you want to initialize a lock you basically have three fields in the lock one is the name, name is just for debugging purposes Then there is this lock variable that we know about that is the state of the lock and then also which CPU is holding it that is also again only for debugging.

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

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

So, basically to protect a thread from the interrupt handler from concurrent accesses by the interrupt handler, you disable the interrupts for the entire critical section. And acquire leaves the interrupts disabled.
So, notice that this holding function what is it doing it just checking that whether I am holding the lock already, whether this CPU is holding the lock already.

So, this is the release function and once again there is just debugging aid which is if not holding lock then you know you panic that you trying to release a lock that you not holding that is fine and you set cpu to 0 and then you exchange lock with 0. So, once again why do I need an exchange it, so I could have just set lock dot lock l k dot locked is equal to 0. But instead the programmer chooses to use the exchange instruction to do this why

So, let us look at how the acquire function is implemented in.. So, let us look at how xv6 is implementing locks.

```
1550 // Mutual exclusion spin locks.
1551
1552 #include "types.h"
1553 #include "defs.h"
1554 #include "param.h"
1555 #include "x86.h"
1556 #include "memlayout.h"
1557 #include "mmu.h"
1558 #include "proc.h"
1559 #include "spinlock.h"
1560
1561 void
1562 initlock(struct spinlock *lk, char
*name)
1563 {
1564 lk−>name = name;
1565 lk−>locked = 0;
1566 lk−>cpu = 0;
1567 }
1568
```

```
1569 // Acquire the lock.
1570 // Loops (spins) until the lock is acquired.
1571 // Holding a lock for a long time may cause
1572 // other CPUs to waste time spinning to acquire it.
1573 void
1574 acquire(struct spinlock *lk)
1575 {
1576 pushcli(); // disable interrupts to avoid deadlock.
1577 if(holding(lk))
1578 panic("acquire");
1579
1580 // The xchg is atomic.
1581 while(xchg(&lk−>locked, 1) != 0)
1582 ;
1583
1584 // Tell the C compiler and the processor to not move
loads or stores
1585 // past this point, to ensure that the critical section's
memory
1586 // references happen after the lock is acquired.
1587 __sync_synchronize();
1588
1589 // Record info about lock acquisition for debugging.
1590 lk−>cpu = mycpu();
1591 getcallerpcs(&lk, lk−>pcs);
```

```
1600 // Release the lock.
1601 void
1602 release(struct spinlock *lk)
1603 {
1604 if(!holding(lk))
1605 panic("release");
1606
1607 lk−>pcs[0] = 0;
1608 lk−>cpu = 0;
1609
1610 // Tell the C compiler and the processor to not move loads
or stores
1611 // past this point, to ensure that all the stores in the critical
1612 // section are visible to other cores before the lock is
released.
1613 // Both the C compiler and the hardware may re−order
loads and
1614 // stores; __sync_synchronize() tells them both not to.
1615 __sync_synchronize();
1616
1617 // Release the lock, equivalent to lk−>locked = 0.
1618 // This code can’t use a C assignment, since it might
1619 // not be atomic. A real OS would use C atomics here.
1620 asm volatile("movl $0, %0" : "+m" (lk−>locked) : );
```

```
1650 // Check whether this cpu is holding
the lock.
1651 int
1652 holding(struct spinlock *lock)
1653 {
1654 int r;
1655 pushcli();
1656 r = lock−>locked && lock−>cpu ==
mycpu();
1657 popcli();
1658 return r;
1659 }
1660
1661
```

```
1662 // Pushcli/popcli are like cli/sti except that they are matched:
1663 // it takes two popcli to undo two pushcli. Also, if interrupts
1664 // are off, then pushcli, popcli leaves them off.
1665
1666 void
1667 pushcli(void)
1668 {
1669 int eflags;
1670
1671 eflags = readeflags();
1672 cli();
1673 if(mycpu()−>ncli == 0)
1674 mycpu()−>intena = eflags & FL_IF;
1675 mycpu()−>ncli += 1;
1676 }
1677
1678 void
1679 popcli(void)
1680 {
1681 if(readeflags()&FL_IF)
1682 panic("popcli − interruptible");
1683 if(−−mycpu()−>ncli < 0)
1684 panic("popcli");
1685 if(mycpu()−>ncli == 0 && mycpu()−>intena)
1686 sti();
1687 }
```

```
2871 // Atomically release lock and sleep on chan.
2872 // Reacquires lock when awakened.
2873 void
2874 sleep(void *chan, struct spinlock *lk)
2875 {
2876 struct proc *p = myproc();
2877
2878 if(p == 0)
2879 panic("sleep");
2880
2881 if(lk == 0)
2882 panic("sleep without lk");
2883
2884 // Must acquire ptable.lock in order to
2885 // change p−>state and then call sched.
2886 // Once we hold ptable.lock, we can be
2887 // guaranteed that we won't miss any wakeup
2888 // (wakeup runs with ptable.lock locked),
2889 // so it's okay to release lk.
2890 if(lk != &ptable.lock){
2891 acquire(&ptable.lock);
2892 release(lk);
2893 }
2894 // Go to sleep.
2895 p−>chan = chan;
2896 p−>state = SLEEPING;
2897
2898 sched();
2899
2900 // Tidy up.
2901 p−>chan = 0;
2902
2903 // Reacquire original lock.
2904 if(lk != &ptable.lock){
2905 release(&ptable.lock);
2906 acquire(lk);
2907 }
2908 }
```

```
2950 // Wake up all processes sleeping on chan.
2951 // The ptable lock must be held.
2952 static void
2953 wakeup1(void *chan)
2954 {
2955 struct proc *p;
2956
2957 for(p = ptable.proc; p <
&ptable.proc[NPROC]; p++)
2958 if(p−>state == SLEEPING && p−>chan ==
chan)
2959 p−>state = RUNNABLE;
2960 }
2961
```

```
2962 // Wake up all processes
sleeping on chan.
2963 void
2964 wakeup(void *chan)
2965 {
2966 acquire(&ptable.lock);
2967 wakeup1(chan);
2968 release(&ptable.lock);
2969 }
```

Thank You

Session -15 Process Control

- RECAP – Session14
- System Calls Related to the Process
- Understanding the fork() System Call
- Understanding the kill() System Call
- Understanding the exit() System Call
- Understanding the wait() System Call

Recap of Session-14

We have learned the following things in the previous session

- **Rae Conditions**
- **Critical Sections**
- **Locks and Unlocks**
- **Interrupts**
- **Semaphore**
- **Deadlocks**

System Calls for the Process

1. Fork
   - Creates a New Process
2. Kill
   - Used to Send Signal to a Process OR Group of Processes
3. Exit
   - Used to Terminate the Process Execution
4. wait
   - Wait for the Process to change its State

Fork System Call

- Fork System Call Helps the User to Create a New Process, this is the only way in Unix to create a Process.
- The Process from which Fork is Invoked is Called as the Parent Process
- The Newly Process is Known as the Child Process.
- Syntax of Fork System Call is ☾ Pid = fork()
- Fork System call does not Take any Arguments.
- After Fork is Executed Successfully both the Processes will have the Same Copies of User Level Context
- But both the Processes will Differ in their Return Values of PId.
- In the Parent Process Pid will have its Child PID and int Child Pid will have 0.

Fork System Call Contd…

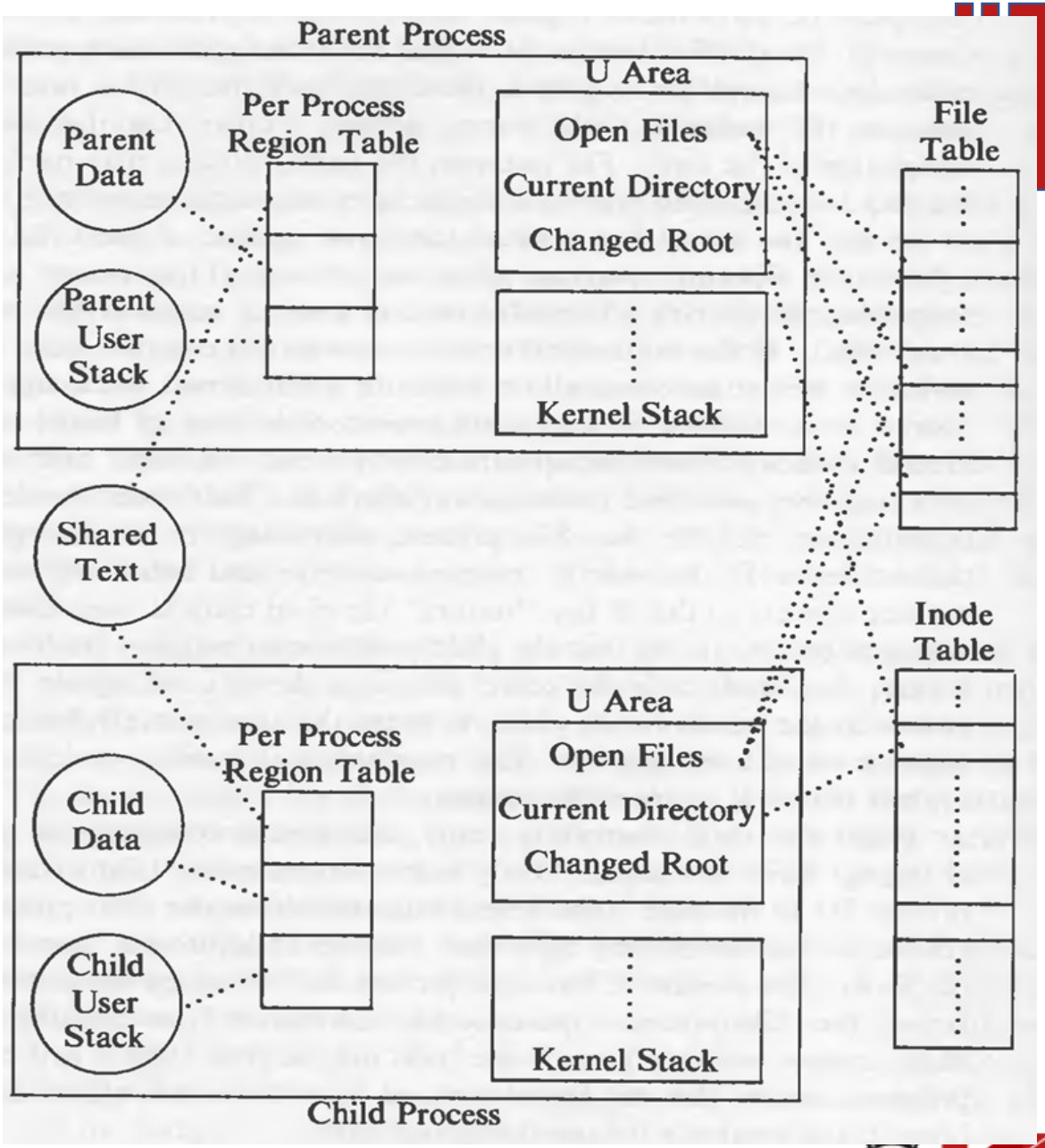
- **Kernel will Perform the Following Tasks for the Fork System Call**
  1. It allocates a slot in the process table for the new process.
  2. It assigns a unique ID number to the child process.
  3. It makes a logical copy of the context of the parent process.
     - Since certain portions of a process, such as the text region, may be shared between processes, the kernel can sometimes increment a region reference count instead of copying the region to a new physical location in memory
  4. It increments file and mode table counters for files associated with the process.
  5. It returns the ID number of the child to the parent process, and a 0 value to the child process.

Fork Algorithm

```
/*  Algorithm: fork
 *  Input: none
 *  Output: to parent process, child PID number
 *          to child process, 0 */
{
        check for available kernel resources;
        get free proc table slot, unique PID number;
        check that user not running too many processes;
        mark child state "being created";
        copy data from parent proc table slot to new
child slot;
        increment counts on current directory inode and
changed root (if applicable);
        increment open file counts in file table;
        make copy of parent context (u-area,text, data, stack) in
memory;
        push dummy system level context layer onto child system level
context;
        // dummy context contains data allowing child process to
        // recognize itself and start running from here when scheduled
        if (executing process is parent process)
        {
                change child state to "ready to run";
                return (child ID);      // from system to user
        }
        else  // executing process is the child process
        {
                initialize u-area timing fields;
                return (0);
        }
```

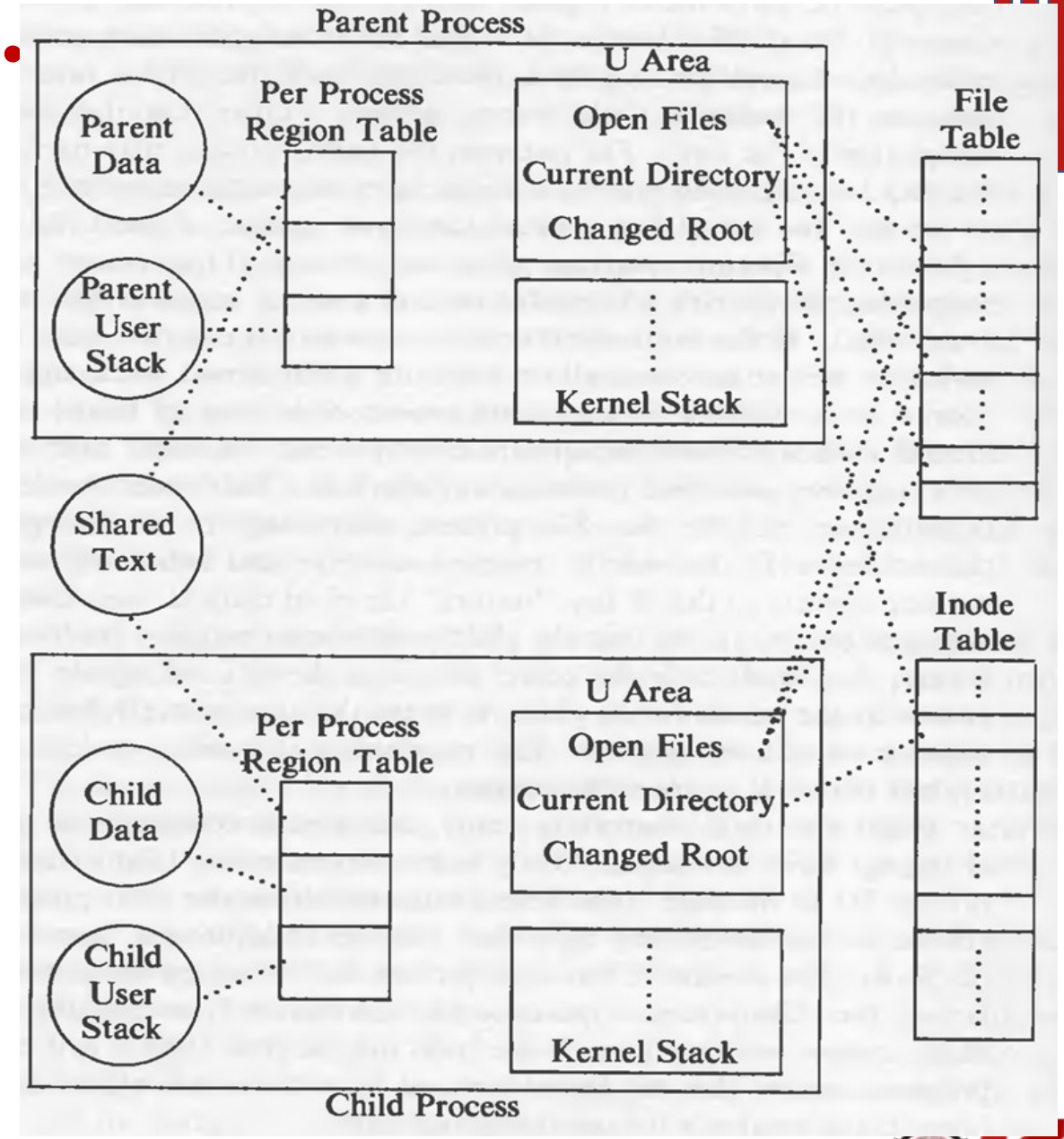
Fork Algorithm Contd…

- Making a copy of a process is called forking.
  - Parent (is the original)
  - child (is the new process)
- When fork is invoked,
  - child is an exact copy of parent
    - When fork is called all pages are shared between parent and child
    - Easily done by copying the parent's page tables



Fork Algorithm Contd…

- Making a copy of a process is called forking.
  - Parent (is the original)
  - child (is the new process)
- When fork is invoked,
  - child is an exact copy of parent
    - When fork is called all pages are shared between parent and child
    - Easily done by copying the parent's page tables



Fork Algorithm Contd...

- The kernel first checks if it has enough resources to create a new process.
  - In a swapping system, it needs space either in memory or on disk to hold the child process.
  - On a paging system, it has to allocate memory for auxiliary tables such as page tables.
- The kernel assigns a unique ID to a process.
  - It is one greater than the previously assigned ID.
  - If another process has the proposed ID number, the kernel attempts to assign the next higher ID number.
- When ID numbers reach the maximum value, assignment starts from 0 again.
  - Since most processes execute for a short time, most ID numbers are not is use when ID assignment wraps around.

Fork Algorithm Contd...

- The system imposes a (configurable) limit on the number of processes a user can simultaneously execute so that no user can steal many process table slots. Similarly, ordinary users cannot create a process that would occupy the last remaining slot in the process table. On the other hand, a super user can create as many processes as it wants (limited by the size of the process table.)
- The kernel assigns parent process ID in the child slot, putting the child in process tree structure, and initialize various scheduling parameters, such as the initial priority value, initial CPU usage, and other timing fields. The initial state of the process is "being created".
- The kernel duplicates every region in the parent process using algorithm *dupreg*, and attaches every region to the child process using algorithm *attachreg*. In a swapping system, it copies the contents of regions that are not shared into a new area of main memory.

Fork Algorithm Contd...

The kernel copies the parent context layer 1, containing user saved register context and the kernel stack frame of the *fork* system call.

If the implementation is one where the kernel stack is part of the u-area, the kernel automatically creates the child kernel stack when it creates the child u-area. Otherwise, the parent process must copy its kernel stack to a private area of memory associated with the child process.

The kernel then creates a dummy context layer 2 for the child process, containing the saved register context for context layer 1.

It sets the program counter and other registers in the saved register context so that it can "restore" the child context, even though it had never executed before, and so that the child process can recognize itself as the child when it runs. For instance, if the kernel code tests the value of register 0 to decide if the process is the parent or the child, it writes the appropriate value in the child saved register context in layer 1.

Fork Algorithm Contd...

When the child context is ready, the parent completes its part of *fork* by changing the child state to "ready to run (in memory)" and by returning the child process ID to the user. The kernel later schedules the child process for execution via the normal scheduling algorithm, and the child process "completes" its part of the *fork*. The context of the child process was set up by the parent process; to the kernel, the child process appears to have awakened after awaiting a resource. The child process executes part of the code for the *fork* system call, according to the program counter that the kernel restored from the saved register context in context layer 2, and returns a 0 from the system call.

The figure give a logical view of the parent and child processes and their relationship with the kernel data structures immediately after completion of the *fork* system call:

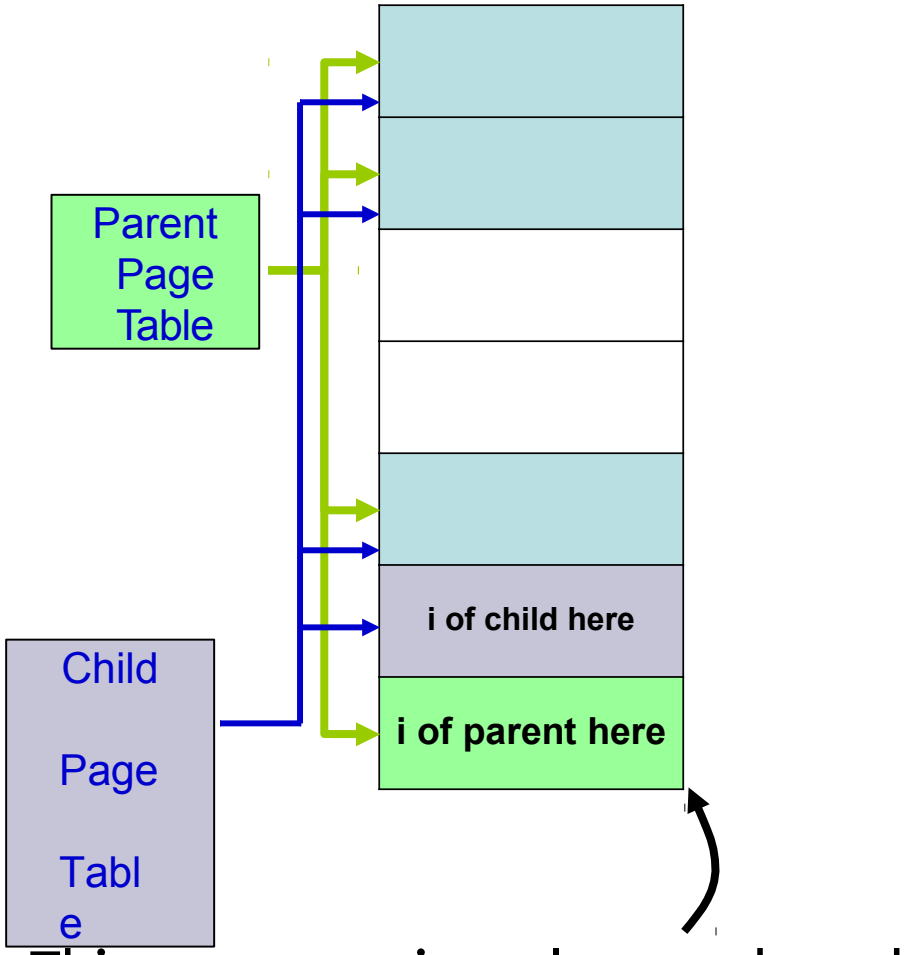
Fork Algorithm Contd…

Copy on Write (COW)

- When data in any of the shared pages change, OS intercepts and makes a copy of the page. Thus, parent and child will have different copies of this page

Why?

- A large portion of executables are not used. Copying each page from parent and child would incur significant disk swapping and huge performance penalties. Postpone coping of pages as much as possible thus optimizing performance



Fork Algorithm Contd...

- Each process has a page table which maps its virtual addresses to physical addresses.
- When the fork() operation is performed, then each process has a new page table created in which each entry is marked with a ‘‘copy-on-write’’ flag.
- this is also done for the caller’s address space. When the contents of memory are to be updated, the flag is checked. If it is set, a new page is allocated, the data from the old page copied, the update is made on the new page, and the ‘‘copy-on-write’’ flag is cleared for the new page. Thus, unexpected changes to shared state do not occur, as independent copies are created ‘‘on demand’’.
- This is very effective in the special case of the shell, where almost no copying has to be done before an exec() replaces the address space.
- If the resources are unavailable, the fork call fails. The kernel finds a slot in the process table to start constructing the context of the child process and makes sure that the user doing the fork does not have too many processes already running.

Fork Algorithm in XV6

```
2579 int
2580 fork(void)
2581 {
2582    int i, pid;
2583    struct proc *np;
2584    struct proc *curproc =
myproc();
2585
2586 // Allocate process.  1
2587 if((np = allocproc()) == 0){
2588 return -1;
2589 }
2590
```

```
2591 // Copy process state from proc.
2592 if((np->pgdir = copyuvm(curproc->pgdir,
curproc->sz)) == 0){
2593 kfree(np->kstack);
2594 np->kstack = 0;
2595 np->state = UNUSED;
2596 return -1;
2597 }
2598 np->sz = curproc->sz;
2599 np->parent = curproc;
```

Fork Algorithm in XV6 contd…

```
2600 *np->tf = *curproc->tf;
2601
2602 // Clear %eax so that fork returns 0 in the child.
2603 np->tf->eax = 0;
2604
2605 for(i = 0; i < NOFILE; i++)
2606 if(curproc->ofile[i])
2607 np->ofile[i] = filedup(curproc->ofile[i]);
2608 np->cwd = idup(curproc->cwd);
2609
2610 safestrcpy(np->name, curproc->name,
sizeof(curproc->name));
2611
2612 pid = np->pid;
2613
2614 acquire(&ptable.lock);
2615
2616 np->state = RUNNABLE;
2617
2618 release(&ptable.lock);
2619
2620 return pid;
2621 }
```

Process Structure in XV6

```
2334 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

2337 struct proc {
2338 uint sz; // Size of process memory (bytes)
2339 pde_t* pgdir; // Page table
2340 char *kstack; // Bottom of kernel stack for this process
2341 enum procstate state; // Process state
2342 int pid; // Process ID
2343 struct proc *parent; // Parent process
2344 struct trapframe *tf; // Trap frame for current syscall
2345 struct context *context; // swtch() here to run process
2346 void *chan; // If non-zero, sleeping on chan
2347 int killed; // If non-zero, have been killed
2348 struct file *ofile[NOFILE]; // Open files
2349 struct inode *cwd; // Current directory
```

Allocating the Process in XV6

```
2468 // Look in the process table for an UNUSED proc.
2469 // If found, change state to EMBRYO and initialize
2470 // state required to run in the kernel.
2471 // Otherwise return 0.
2472 static struct proc*
2473 allocproc(void)
2474 {
2475 struct proc *p;
2476 char *sp;
2477
2478 acquire(&ptable.lock);
2479
2480 for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2481 if(p->state == UNUSED)
2482 goto found;
2483
2484 release(&ptable.lock);
2485 return 0;
```

```
2486
2487 found:
2488 p->state = EMBRYO;
2489 p->pid = nextpid++;
2490
2491 release(&ptable.lock);
2492
2493 // Allocate kernel stack.
2494 if((p->kstack = kalloc()) == 0){
2495 p->state = UNUSED;
2496 return 0;
2497 }
2498 sp = p->kstack + KSTACKSIZE;
```

Signals

- *Signals* inform processes of the occurrence of asynchronous events.
- Processes may send each other *signals* with the *kill* system call, or the kernel may send signals internally.
- There are 19 signals in the System V (Release 2) UNIX system that can be classified as follows:
  - Signals having to do with the termination of a process, send when a process exits or when a process invokes the signal system call with the death of child parameter.
  - Signals having to do with process induced exceptions such as when a process accesses an address outside its virtual address space, when it attempts to write memory that is read-only (such as program text), or when it executes a privileged instruction or for various hardware errors.
  - Signals having to do with the unrecoverable conditions during a system call, such as running out of system resources during *exec* after the original address space has been released

Signals Contd...

- Signals originating from a process in user mode, such as when a process wishes to receive an *alarm* signal after a period of time, or when processes send arbitrary signals to each other with the *kill* system call.
- Signals related to terminal interaction such as when a user hands up a terminal (or the "carrier" signal drops on such a line for any reason), or when a user presses the "break" or "delete" keys on a terminal keyboard.
- Signals for tracing execution of a process.
- When a kernel or a process sends a signal to another process, a bit in the process table entry of that process is set, with respect to the type of signal received. If the process is asleep at an interruptible priority, the kernel awakens it. A process can remember different types of signals but it cannot remember how many times a signal of a particular type was received.

Handling Signals

- The kernel handles signals in the context of the process that receives them so a process must run to handle signals.
- There are three cases for handling signals: the process *exits* on receipt of the signal, it ignores the signal, or it executes a particular (user) function on receipt of the signal.
- The default action is to call *exit* in kernel mode, but a process can specify special action to take on receipt of certain signals with the *signal* system call.
  - old function = signal (signum, function);

Handling Signals

- where signum is the signal number the process is specifying the action for, function is the address of the (user) function the process wants to invoke on receipt of the signal, and the return value old function was the value of function in the most recently specified call to *signal* for *signum*.
- The process can pass the values 1 or 0 instead of a function address: The process will ignore future occurrences of the signal if the parameter value is 1 and *exit* in the kernel on receipt of the signal if its value is 0 (default value).
- The u-area contains an array of signal-handler fields, one for each signal defined in the system. The kernel stores the address of the user function in the field that corresponds to the signal number.

Processes Termination

- Voluntary : exit(status)
  - OS passes exit status to parent via wait(&status)
  - OS frees process resources

- Involuntary : kill(pid, signal)
  - Signal can be sent by another process or by OS
  - pid is for the process to be killed
  - signal a signal that the process needs to be killed
    - Examples : SIGTERM, SIGQUIT (ctrl+\), SIGINT (ctrl+c), SIGHUP

Process Termination Exit

- Processes on the UNIX system exit by executing the exit system call. When a process exits, it enters the zombie state, relinquishes all of its resources, and dismantles its context except for its process table entry.
- The exit() system call ends a process and returns a value to it parent. The prototype for the exit() system call is: exit (status);
- where status is an integer between 0 and 255.
- This number is returned to the parent via wait() as the exit status of the process. By convention, when a process exits with a status of zero that means it didn't encounter any problems; when a process exit with a non-zero status that means it did have problems.
- exit() is actually not a system routine
- it is a library routine that call the system routine _exit(). exit() cleans up the standard I/O streams before calling _exit(), so any output that has been buffered but not yet actually written out is flushed.
- Calling _exit() instead of exit() will bypass this clean up procedure. exit() does not return.

Process Termination Exit

```
/*  Algorithm: exit  Input: return code for parent process  Output: none */
{
      ignore all signals;
      if (process group leader with associated control terminal)
      {
            send hangup signal to all members of the process group;
            reset process group for all members to 0;
      }
      close all open files (internal version of algorithm close);
      release current directory (algorithm: iput);
      release current (changed) root, if exists (algorithm: iput);
      free regions, memory associated with process (algorithm: freereg);
      write accounting record;
      make process state zombie;
      assign parent process ID for all child processes to be init process (1);
            if any children were zombie, send death of child signal to init;
      send death of child signal to parent process;
      context switch;
}
```

Exit in XV6

```
2623 // Exit the current process. Does not return.
2624 // An exited process remains in the zombie state
2625 // until its parent calls wait() to find out it exited.
2626 void
2627 exit(void)
2628 {
2629 struct proc *curproc = myproc();
2630 struct proc *p;
2631 int fd;
2632
2633 if(curproc == initproc)
2634 panic("init exiting");
2635
2636 // Close all open files.
```

```
2637 for(fd = 0; fd < NOFILE; fd++){
2638 if(curproc->ofile[fd]){
2639 fileclose(curproc->ofile[fd]);
2640 curproc->ofile[fd] = 0;
2641 }
2642 }
```

Processes Termination Kill

- Processes use the *kill* system call to send signals.
  - kill (pid, signum);
- where pid identifies the set of processes to receive the signal, and signum is the signal number being sent. The following list shows the correspondence between values of pid and sets of processes.
- If pid is a positive integer, the kernel sends the signal to the process with process ID pid.
- If pid is 0, the kernel sends the signal to all processes in the sender's process group.
- If pid is -1, the kernel sends the signal to all processes whose real user ID equals the effective user ID of the sender (real and effective user IDs are studied later). If the sending process has effective user ID of superuser, the kernel sends the signal to all processes except processes 0 and 1.
- If pid is a negative integer but not -1, the kernel sends the signal to all processes in the process group equal to the absolute value of pid.

Kill in XV6

```
2971 // Kill the process with the given pid.
2972 // Process won't exit until it returns
2973 // to user space (see trap in trap.c).
2974 int
2975 kill(int pid)
2976 {
2977 struct proc *p;
2978
2979 acquire(&ptable.lock);
2980 for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2981 if(p->pid == pid){
2982 p->killed = 1;
2983 // Wake process from sleep if necessary.
2984 if(p->state == SLEEPING)
2985 p->state = RUNNABLE;
2986 release(&ptable.lock);
2987 return 0;
2988 }
2989 }
2990 release(&ptable.lock);
2991 return -1;
2992 }
```

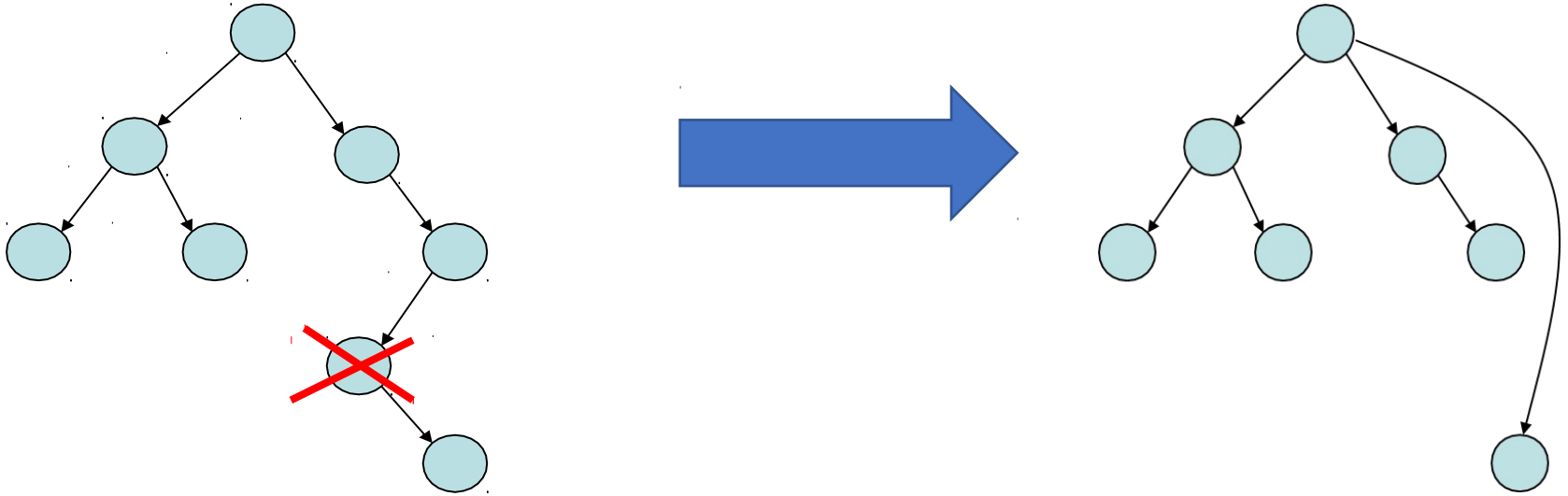
- When a child exits, it does not die immediately. Instead, it switches to the ZOMBIE process state until the parent calls wait to learn of the exit.
- The parent is then responsible for freeing the memory associated wit h the process and preparing the struct proc for reuse.
- If the parent exits before the child, the init process adopts the child and waits for it, so that every child has a parent to clean up after it.
- Exit acquires ptable.lock and then wakes up any process sleeping on a wait channel equal to the current process's parent proc(2651); if there is such a process, it will be the parent in wait.
- This may look premature, since exit has not marked the current process as a ZOMBIE yet, but it is safe: although wakeup may cause the parent to run, the loop in wait cannot run until exit releases ptable.lock by calling sched to enter the scheduler, so wait can't look at the exiting process until after exit has set its state to ZOMBIE(2663). Before exit yields the processor, it reparents all of the exiting process's children, passing them to the initproc(2653-2660). Finally, exit calls sched to relinquish the CPU.

Zombies

- When a process terminates it becomes a zombie (or defunct process)
  - PCB in OS still exists even though program no longer executing
  - Why?
  - So that the parent process can read the child's exit status (through wait system call)
- When parent reads status zombie entries removed from OS… process reaped!
- Suppose parent does not read status Zombie will continue to exist infinitely … a resource leak
  - These are typically found by a reaper process

Orphans

- When a parent process terminates before its child Adopted by first process (/sbin/init)



- Unintentional orphans
  - When parent crashes
- Intentional orphans
  - Process becomes detached from user session and runs in the background
  - Called daemons, used to run background services
  - See nohup

Wait system call

- The wait() system call suspends execution of the calling process until one of its children terminates. The call wait(&status) is equivalent to: waitpid(-1, &status, 0);
- The waitpid() system call suspends execution of the calling process until a child specified by pid argument has changed state. By default, waitpid() waits only for terminated children, but this behaviour is modifiable via the options argument, as described below.
- The Value of PId can be one from the following
  - < -1 ☾ Wait for any child process whose group Id is equal to the obsolute value of Pid
  - -1 ☾ Wait for any child Process
  - 0 ☾ wait for any child process whose process group ID is equal to that of the calling process.
  - > 0 ☾ meaning wait for the child whose process ID is equal to the value of pid

Wait system call code in xv6

```
2668 // Wait for a child process to exit and return its pid.
2669 // Return -1 if this process has no children.
2670 int
2671 wait(void)
2672 {
2673struct proc *p;
2674int havekids, pid;
2675   struct proc *curproc = myproc();
2676
2677 acquire(&ptable.lock);
2678   for(;;){
2679// Scan through table looking for exited children.
2680havekids = 0;
2681   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2682      if(p->parent != curproc)
2683 continue;
2684      havekids = 1;
2685       if(p->state == ZOMBIE){
2686         // Found one.
2683        pid = p->pid;
2684         kfree(p->kstack);
2685         p->kstack = 0;
2686         freevm(p->pgdir);
2687         p->pid = 0;
2688         p->parent = 0;
2689         p->name[0] = 0;
2690         p->killed = 0;
2691        p->state = UNUSED;
2692        release(&ptable.lock);
2693         return pid;
2694       }
2695    }
```

Wait system call code in xv6

```
// No point waiting if we don't have any children.
if(!havekids || curproc->killed){
release(&ptable.lock);
return -1;
}

// Wait for children to exit. (See wakeup1 call in proc_exit.)
sleep(curproc, &ptable.lock);
}
}
```

Thank You

Session -16 Process Control

- RECAP – Session15
- System Calls Related to the Process
- Understanding the exec() System Call
- Understanding the shell
- Understanding the init process

Recap of Session-15

- In Session15 we have learned the following
- Fork System Call
- Working Mechanism of Fork
- Process Structure
- Kill System Call
- Exit System Call
- Zombies
- Orphans

System Calls for the Process

1. Exec
   - Used to Invoke other Programs
2. Shell
   - Acts as an Interface between User Applications and Kernel
3. Init Process
   - The Second Process of the System

Exec System Call

- We fork to create a process, but more often than not, we follow it with an exec to run a separate program in the address space of the child.
- Exec replaces this address space (the text, data, and stack) with that of the new program, which then starts running by executing its main function.
- Since no new process is created, the PID doesn't change across an exec. Because the stack is replaced with a new one, the call to exec doesn't return unless it results in an error.

Exec System Call

- The exec system call invokes another program, overlaying the memory space of a process with a copy of an executable file.
- The contents of the user-level context that existed before the exec call are no longer accessible afterward except for exec's parameters, which the kernel copies from the old address space to the new address space.

Exec Algorithm

```
algorithm exec
input: (1) file name (2) parameter list
       (3) environment variables list
output: none
{
   get file mode (algorithm namei);
   verify file executable, user bas permission to execute;
   read file headers, check that it is a bad module;
   copy exec parameters from old address space to system space;
   for (every region attached to process)
      detach all old regions (algorithm detach);
      for (every region specified in laad module)
      {
      allocate new regions (algorithm allocreg);
      attach the regions (algorithm attachreg);
      load region into memory if appropriate (algorithm
loadreg);
       }
      copy exec parameters into new user stack region;
      special processing for setuid programs, tracing;
      initialize user register save area for return to user
                  mode;
      release mode of file (algorithm iput);
}
```

Exec System Call Contd…

- Following attributes that are inherited from fork will remain same when exec is executed.
  - Previous program's file descriptors.
  - Current and root directory.
  - Umask settings
  - Global environment.
- However, a programmer can change the exec'd environment in two ways.
  - By closing one or more file descriptors, so files opened before or after a fork can't be read directly in the exec'd process.
  - By passing a separate environment to the exec'd process instead of the default environment maintained in the global environ variable. Two exec functions (execve and execle) are designed to work this way

Exec XV6 Code

```
6600 #include "types.h"
6601 #include "param.h“
6602 #include "memlayout.h"
6603 #include "mmu.h"
6604 #include "proc.h"
6605 #include "defs.h"
6606 #include "x86.h“
6607 #include "elf.h“
6608
6609 int
6610 exec(char *path, char **argv)
6611 { 6612 char *s, *last;
6613 int i, off;
6614 uint argc, sz, sp, ustack[3+MAXARG+1];
6615 struct elfhdr elf;
6616 struct inode *ip;
6617 struct proghdr ph;
6618 pde_t *pgdir, *oldpgdir;
6619 struct proc *curproc = myproc();
6620
6621 begin_op();
6622
6623 if((ip = namei(path)) == 0){
6624 end_op();
6625 cprintf("exec: fail\n");
6626 return −1;
6627 }
6628 ilock(ip);
6629 pgdir = 0;
6630
6631 // Check ELF header
6632 if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
6633 goto bad;
6634 if(elf.magic != ELF_MAGIC)
6635 goto bad;
6636
6637 if((pgdir = setupkvm()) == 0)
6638 goto bad;
6639
6640 // Load program into memory.
6641 sz = 0;
6642 for(i=0, off=elf.phoff; i< ph.filesz)
6648      goto bad;
6649 if(ph.vaddr + ph.memsz < ph.vaddr)
```

Exec XV6 Code

```
6650 goto bad;
6651 if((sz = allocuvm(pgdir, sz, ph.vaddr +
ph.memsz)) == 0)
6652 goto bad;
6653 if(ph.vaddr % PGSIZE != 0)
6654 goto bad;
6655 if(loaduvm(pgdir, (char*)ph.vaddr, ip,
ph.off, ph.filesz) < 0)
6656 goto bad;
6657 }
6658 iunlockput(ip);
6659 end_op();
6660 ip = 0;
6661
6662 // Allocate two pages at the next page
boundary.
6663 // Make the first inaccessible. Use the
second as the user stack.
6664 sz = PGROUNDUP(sz);
6665 if((sz = allocuvm(pgdir, sz, sz +
2*PGSIZE)) == 0)
6666 goto bad;
6667 clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
6668 sp = sz;
6669
6670 // Push argument strings, prepare rest of stack in ustack.
6671 for(argc = 0; argv[argc]; argc++) {
6672 if(argc >= MAXARG)
6673 goto bad;
6674 sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
6675 if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
6676 goto bad;
6677 ustack[3+argc] = sp;
6678 }
6679 ustack[3+argc] = 0;
6680
6681 ustack[0] = 0xffffffff; // fake return PC
6682 ustack[1] = argc;
6683 ustack[2] = sp - (argc+1)*4; // argv pointer
6684
6685 sp -= (3+argc+1) * 4;
6686 if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
6687 goto bad;
6688
```

Exec Xv6 Code

```
6689 // Save program name for debugging.
6690 for(last=s=path; *s; s++)
6691 if(*s == '/')
6692 last = s+1;
6693 safestrcpy(curproc->name, last, sizeof(curproc->name));
6694
6695 // Commit to the user image.
6696 oldpgdir = curproc->pgdir;
6697 curproc->pgdir = pgdir;
6698 curproc->sz = sz;
6699 curproc->tf->eip = elf.entry; // main
6700 curproc->tf->esp = sp;
6701 switchuvm(curproc);
6702 freevm(oldpgdir);
6703 return 0;
6704
```

```
6705 bad:
6706 if(pgdir)
6707 freevm(pgdir);
6708 if(ip){
6709 iunlockput(ip);
6710 end_op();
6711 }
6712 return -1;
6713 }
6714
```

Exec Working

- Exec is the system call that creates the user part of an address space. It initializes the user part of an address space from a file stored in the file system.
- Exec (6610) opens the named binary path using namei (6623). Then, it reads the ELF header.
- Xv6 applications are described in the widely-used ELF format, defined in elf.h.
- An ELF binary consists of an ELF header, struct elfhdr (0905), followed by a sequence of program section headers, struct proghdr (0924).
- Each proghdr describes a section of the application that must be loaded into memory; xv6 programs have only one program section header, but other systems might have separate sections for instructions and data.
- The first step is a quick check that the file probably contains an ELF binary.
- An ELF binary starts with the four-byte ‘‘magic number’’ 0x7F, ’E’, ’L’, ’F’, or ELF_MAGIC (0902).
- If the ELF header has the right magic number, exec assumes that the binary is well-formed. Exec allocates a new page table with no user mappings with setupkvm (6637), allocates memory for each ELF segment with allocuvm (6651), and loads each segment into memory with loaduvm (6655).
- allocuvm checks that the virtual addresses requested is below KERNBASE.
- loaduvm (1903) uses walkpgdir to find the physical address of the allocated memory at which to write each page of the ELF segment, and readi to read from the file

Exec Working

- The program section header's filesz may be less than the memsz, indicating that the gap between them should be filled with zeroes (for C global variables) rather than read from the file.
- For /init, filesz is 2240 bytes and memsz is 2252 bytes, and thus allocuvm allocates enough physical memory to hold 2252 bytes, but reads only 2240bytes from the file /init.
- Now exec allocates and initializes the user stack. It allocates just one stack page. Exec copies the argument strings to the top of the stack one at a time, recording the pointers to them in ustack.
- Exec places an inaccessible page just below the stack page, so that programs that try to use more than one page will fault. This inaccessible page also allows exec to deal with arguments that are too large; in that situation, the copyout(2118) function that exec uses to copy arguments to the stack will notice that the destination page is not accessible, and will return −1.
- During the preparation of the new memory image, if exec detects an error like an invalid program segment, it jumps to the label bad, frees the new image, and returns −1.Exec must wait to free the old image until it is sure that the system call will succeed: if the old image is gone, the system call cannot return−1 to it. The only error cases in exec happen during the creation of the image.
- Once the image is complete, exec can install the new image(6701) and free the old one(6702). Finally, exec returns 0.

Using Exec may be Risky and Complicated

- Exec loads bytes from the ELF file into memory at addresses specified by the ELF file.
- Users or processes can place whatever addresses they want int o an ELF file.
- Thus exec is risky, because the addresses in the ELF file may refer to the kernel, accidentally or on purpose.
- Notice that when exec copies the program segments, it makes sure that the data being loaded into memory fits in the declared size ph.memsz (5069-5070).
- Without this check, a malformed ELF binary could cause exec to write past the end of the allocated memory image, causing memory corruption and making the operating system unstable.
- The boot sector neglected this check both to reduce code size and because not checking doesn't change the failure mode: either way the machine doesn't boot if given a bad ELF image. In contrast, in exec this check is the difference between making one process fail and making the entire system fail.

- Exec is the most complicated code in xv6 in and in most operating systems. It involves pointer translation (in sys_exec too), many error cases, and must replace one running process with another. Real world operating systems have even more complicated exec implementations. They handle shell scripts more complicated ELF binaries, and even multiple binary formats.

The Shell

- The Shell acts as an interface between the User / User Applications and the Kernel.
- The shell is both a command interpreter and a programming language. It is also a process that creates an environment to work in.
- The shell reads a command line from its standard input and interprets it according to a fixed set of rules. The standard input and standard output file descriptors for the login shell are usually the terminal on which the user logs in.
- If the shell recognizes the input string as a built-in command (for example, commands cd, for, while and others), it executes the command internally without creating new processes; otherwise, it assumes the command is the name of an executable file.
- The simplest command lines contain a program name and some parameters, such as who, grep —n, Is —l The shell forks and creates a child process, which execs the program that the user specified on the command line.
- The parent process, the shell that the user is using, waits until the child process exits from the command and then loops back to read the next command.

The Shell Contd…

- UNIX Operating System offers a variety of shells for you to choose from. Over time, shells have become more powerful by the progressive addition of new features. The shells we consider in this text can be grouped into two categories
  - The Bourne family comprising the Bourne shell (/bin/sh) and its derivatives—the Korn shell (/bin/ksh) and Bash (/bin/bash).
  - The C Shell (/bin/csh) and its derivative, Tcsh (/bin/tcsh).
- The shell's own pathname is stored in SHELL, but its PID is stored in a special "variable", $$.
- To know the PID of your current shell, type $ echo $$

Shell Commands - Sequence of Steps

- **Parsing:** The shell first breaks up the command line into words using spaces and tabs as delimiters, unless quoted. All consecutive occurrences of a space or tab are replaced here with a single space.
- **Variable evaluation:** All words preceded by a $ are evaluated as variables, unless quoted or escaped.
- **Command substitution:** Any command surrounded by backquotes is executed by the shell, which then replaces the standard output of the command in the command line.
- **Redirection:** The shell then looks for the characters >, > to open the files they point to.
- **Wild-card interpretation:** The shell finally scans the command line for wild cards (the characters *, ?, [and]). Any word containing a wild card is replaced with a sorted list of filenames that match the pattern. The list of these filenames then forms the arguments to the command.
- **PATH evaluation:** The shell finally looks for the PATH variable to determine the sequence of directories it has to search in order to hunt for the command.

Init Process

- The PPID of every login shell is always 1. This is the init process: the second process of the system.
- init is a very important process and, apart from being the parent of users' shells, it is also responsible for giving birth to every service that's running in the system—like printing, mail, Web, and so on.
- Even though no one may be using the system, a number of system processes keep running all the time.
- They are spawned during system startup by init (PID 1), the parent of the login shell. The ps -e command lists them all.
- System processes that have no controlling terminal are easily identified by the ? in the TTY column.
- A process that is disassociated from the terminal can neither write to the terminal nor read from it. Such processes are also known as daemons. Many of these daemons are actually sleeping (a process state) and wake up only when they receive input.
- Examples of Daemons are: lpsched, sendmail, inetd etc.

Init Process Contd...

- To initialize a system from an inactive state, an administrator goes through a "bootstrap" sequence: The administrator "boots" the system. Boot procedures vary according to machine type, but the goal is common to all machines.
- The init process is a process dispatcher, spawning processes that allow users to log in to the system, among others.
- Init reads the file "tetchnittab" for instructions about which processes to spawn.
- The file "/etc/inittab" contains lines that contain an "id," a state identifier (single user, multi-user, etc.), an "action" and a program specification.
- Init reads the file and, if the state in which it was invoked matches the state identifier of a line, creates a process that executes the given program specification

Booting Algorithm

```
algorithm start /* system startup procedure */
input: none
output: none
{
   initialize all kernel data structures;
   pseudo-mount of root;
   hand-craft environment of process 0;
   fork process 1: /* process 1 in here */
   allocate region;
   attach region to init address space;
   grow region to accommodate code about to copy in;
   copy code from kernel space to init user space to exec init;
   change mode: return from kernel to user mode;
    /* init never gets here---as result of above change mode,
```

Booting Algorithm Contd...

```
   * init exec's ietc/init and becomes a "normar user process * with respect to invocation
            of system calls * /
 }
 */ proc 0 continues here */
  fork kernel processes;
  /* process 0 invokes the swapper to manage the allocation of
   * process address space to main memory and the swap devices,
   * This is an infinite loop; process 0 usually sleeps in the
   * loop unless there is work for it to do. */
  execute code for swapper algorithm;
}
```

Init Algorithm

```
algorithm init
/* init process, process I of the system */
 input; none
 output: none
{
    fd = open("/etc/inittab", O_RDONLY);
   while (line_read(fd,   buffer))
   {
        /* read every line of file */
       if (invoked state !, buffer state)
            continue;   /* loop back to while */
       /* state matched */
     if (fork() == 0)
     {
       execl("process specified in buffer");
       exit();
     }
      /* init process does not wait  */
     /* loop back to while */
   while ((id == wait((int *) 0)) !=1)
   {
     /* check here if a spawned child died;
      * consider respawning it */
     /* otherwise, just continue */
   }
}
```

Thank You

Session -17 process scheduling

- RECAP – Session16
- Understanding the Scheduling process
- Understanding the Multiplexing
- Understanding the Context Switching
- Understanding the Sleep and Wakeup
- Understanding the my proc and my cpu

Recap of Session-16

We have learned the following things in the previous session

- Exec System Call
- Internal working of Exec
- Exec code in XV6
- Risk og using Exec
- The Shell Command
- Init Process

Scheduling-Introduction

- An operating system runs more processes than it has processors
- Needs some plan to time share the processors between the processes
- A common approach is to provide each process with a virtual processor – An illusion that it has exclusive access to the processor
- It is then the job of the OS to multiplex these multiple virtual processors on the underlying physical processors

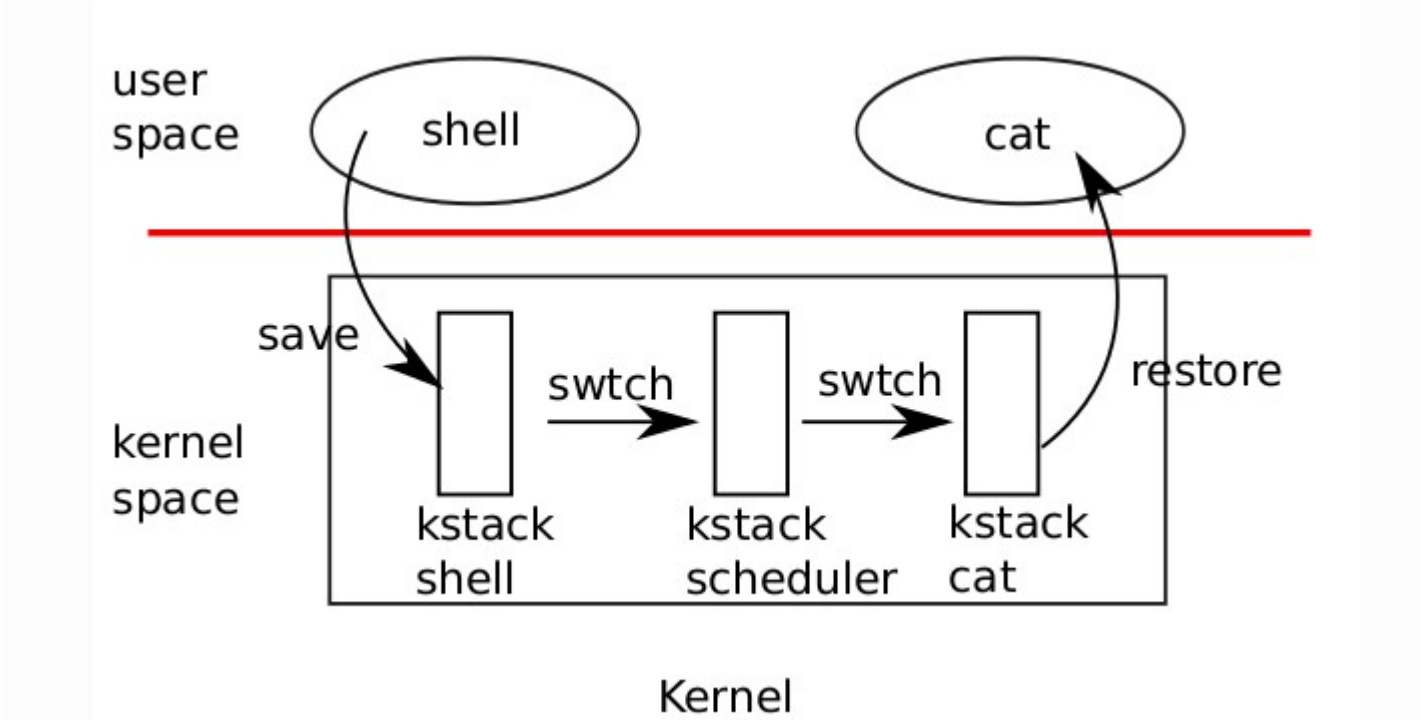
Multiplexing

- A process does I/O, put it to sleep, and schedule another process
- Use timer interrupts to stop running on a processor after a fixed time quantum (100 msec)
- Implementing multiplexing has a few challenges.
  - First, how to switch from one process to another?
  - Second, how to do context switching transparently?
  - Third, many CPUs may be switching among processes concurrently, and a locking plan is necessary to avoid races.
  - Fourth, when a process has exited its memory and otherresources must be freed, but it cannot do all of this itself because (for example) it can’tfree its own kernel stack while still using it.
- Xv6 uses the standard mechanism of context switching

Context Switching

- Used to achieve multiplexing
- Internally two low-level context switches are performed
  - Process's kernel thread to the current CPU's scheduler thread
  - Scheduler's thread to a process's kernel thread
- No direct switching from one user-space process to another
- Each process has its own kernel stack and register set (its context)
- Each CPU has its own scheduler thread
- Context switch involves saving the old thread's CPU registers and restoring previously-saved registers of the new thread (enabled by swtch)

Context Switching



Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

swtch

- Saves and restores contexts
- Takes two arguments: struct context **old and struct context *new
- Replaces the former with the latter
- Each time a process has to give up the CPU, its kernel thread invokes swtch to save its own context and switch to the scheduler context
- Flow in case of an interrupt:

1 trap handles the interrupt and the calls yield

2 yield makes a call to sched

3 sched invokes swtch(&proc->context, cpu->scheduler)

4 Control returns to the scheduler thread

Scheduling

- Each process that wants to give up the processor:
  1 Acquires ptable.lock (process table lock)
  2 Releases any other locks that it is holding
  3 Updates proc->state (its own state)
  4 Calls sched
- Mechanism followed by yield, and sleep and exit
- sched ensures that these steps are followed

sched

```
1 void sched(void) {
2 int intena;
3
4 if(! holding (& ptable.lock ))
5 panic("sched ptable.lock");
6 if(cpu−>ncli != 1)
7 panic("sched locks");
8 if(proc−>state == RUNNING)
9 panic("sched running");
10 if( readeflags ()& FL_IF)
11 panic("sched interruptible");
12 intena = cpu−>intena;
13 swtch (&proc−>context , cpu−>scheduler );
14 cpu−>intena = intena;
15 }
```

yield

```
1 void yield(void) {
2 acquire (& ptable.lock );
3 proc−>state = RUNNABLE;
4 sched ();
5 release (& ptable.lock );
6 }
```

Scheduling

- Why must a process acquire ptable.lock before a call to swtch?
- Breaks the convention that the thread that acquires a lock is also responsible for releasing the lock
- Without acquiring ptable.lock, two CPUs might want to schedule the same process because they can access ptable

Scheduler

- Simple loop: find a process to run, run it until it stops, repeat
- Acquires and releases ptable.lock, and enables interrupts on every iteration. Why?
- If CPU is idle (no RUNNABLE) 1 Idle looping while holding a lock would not allow any other CPU to access the process table 2 Idle looping (all processes are waiting for I/O) while interrupts are disabled would not allow any I/O to arrive
- The first process with p->state == RUNNABLE is selected
- The process is assigned to the per-CPU proc
- The process's page table is switched to via switchuvm
- The process is marked as RUNNING
- swtch is called to start running it

Scheduler

```
1 void scheduler (void) {
2 struct proc *p;
3 for(;;){
4 sti ();
5 acquire (& ptable.lock );
6 for(p = ptable.proc; p < &ptable.proc[NPROC ]; p++){
7 if(p->state != RUNNABLE)
8 continue;
9 proc = p;
10 switchuvm (p);
11 p->state = RUNNING;
12 swtch (&cpu->scheduler , proc->context );
13 switchkvm ();
14 proc = 0;
15 }
16 release (& ptable.lock );
17 }
```

Sleep and wakeup

sleep and wakeup enable an IPC mechanism

- Enable sequence coordination or conditional synchronization
- sleep allows one process to sleep waiting for an event
- wakeup allows another process to wake up processes sleeping on an event

Queue

```
100 struct q {
101   void *ptr;
102 };
103
104 void*
105 send(struct q *q, void *p)
106 {
107   while(q->ptr != 0)
108     ;
109   q->ptr = p;
110 }
111
112 void*
113 recv(struct q *q)
114 {
115   void *p;
116
117   while((p = q->ptr) == 0)
118     ;
119   q->ptr = 0;
120   return p;
121 }
```

Queue with sleep and wakeup

```
201 void*
202 send(struct q *q, void *p)
203 {
204   while(q->ptr != 0)
205     ;
206   q->ptr = p;
207   wakeup(q); /* wake recv */
208 }
209
210 void*
211 recv(struct q *q)
212 {
213   void *p;
214
215   while((p = q->ptr) == 0)
216     sleep(q);
217   q->ptr = 0;
218   return p;
219 }
```

Queue with sleep and wakeup and locking

```
300 struct q {
301   struct spinlock lock;
302   void *ptr;
303 };
304
305 void*
306 send(struct q *q, void *p)
307 {
308   acquire(&q->lock);
309   while(q->ptr != 0)
310     ;
311   q->ptr = p;
312   wakeup(q);
313   release(&q->lock);
314 }
```

```
315
316 void*
317 recv(struct q *q)
318 {
319   void *p;
320
321   acquire(&q->lock);
322   while((p = q->ptr) == 0)
323     sleep(q);
324   q->ptr = 0;
325   release(&q->lock);
326   return p;
327 }
```

Queue with sleep and wakeup and implicit locking

```
400 struct q {
401   struct spinlock lock;
402   void *ptr;
403 };
404
405 void*
406 send(struct q *q, void *p)
407 {
408   acquire(&q->lock);
409   while(q->ptr != 0)
410     ;
411   q->ptr = p;
412   wakeup(q);
413   release(&q->lock);
414 }
415
```

```
416 void*
417 recv(struct q *q)
418 {
419   void *p;
420
421   acquire(&q->lock);
422   while((p = q->ptr) == 0)
423     sleep(q, &q->lock);
424   q->ptr = 0;
425   release(&q->lock);
426   return p;
427 }
```

My proc and CPU

```
2300 // Segments in proc->gdt.
 2301 #define NSEGS 7
2302
2303 // Per-CPU state
 2304 struct cpu {
2305 uchar id; // Local APIC ID; index into cpus[] below
2306 struct context *scheduler; // swtch() here to enter
scheduler
2307 struct taskstate ts; // Used by x86 to find stack for
interrupt
2308 struct segdesc gdt[NSEGS]; // x86 global descriptor
table 2309 volatile uint started; // Has the CPU started?
2310 int ncli; // Depth of pushcli nesting.
2311 int intena; // Were interrupts enabled before pushcli?
2312
2313 // Cpu-local storage variables; see below
2314 struct cpu *cpu;
2315 struct proc *proc; // The currently-running process.
2316 };
2317
2318 extern struct cpu cpus[NCPU];
2319 extern int ncpu; 2320
```

```
2321 // Per-CPU variables, holding pointers to the
2322 // current cpu and to the current process.
2323 // The asm suffix tells gcc to use "%gs:0" to
refer to cpu
2324 // and "%gs:4" to refer to proc. seginit sets up
the
2325 // %gs segment register so that %gs refers to
the memory
2326 // holding those two variables in the local
cpu's struct cpu.
2327 // This is similar to how thread-local variables
are implemented
2328 // in thread libraries such as Linux pthreads.
2329 extern struct cpu *cpu asm("%gs:0"); //
&cpus[cpunum()]
2330 extern struct proc *proc asm("%gs:4"); //
cpus[cpunum()].proc
```

```
2331 2332
 2333 // Saved registers for kernel context switches.
2334 // Don't need to save all the segment registers (%cs, etc), 2335 // because they are constant across kernel contexts.
2336 // Don't need to save %eax, %ecx, %edx, because the 2337 // x86 convention is that the caller has saved them.
2338 // Contexts are stored at the bottom of the stack they 2339 // describe; the stack pointer is the address of the context.
2340 // The layout of the context matches the layout of the stack in swtch.S
2341 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
2342 // but it is on the stack and allocproc() manipulates it. 2343 struct context {
2344 uint edi;
2345 uint esi;
2346 uint ebx;
2347 uint ebp;
2348 uint eip;
 2349 };
```

```
2350 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
2351
2352 // Per-process state
2353 struct proc {
2354 uint sz; // Size of process memory (bytes)
2355 pde_t* pgdir; // Page table
2356 char *kstack; // Bottom of kernel stack for this process
 2357 enum procstate state; // Process state
 2358 int pid; // Process ID
2359 struct proc *parent; // Parent process
2360 struct trapframe *tf; // Trap frame for current syscall
2361 struct context *context; // swtch() here to run process
2362 void *chan; // If non-zero, sleeping on chan
2363 int killed; // If non-zero, have been killed
2364 struct file *ofile[NOFILE]; // Open files
 2365 struct inode *cwd; // Current directory
2366 char name[16]; // Process name (debugging)
2367 };
 2368
2369 // Process memory is laid out contiguously, low addresses first:
2370 // text 2371 // original data and bss
2372 // fixed-size stack
2373 // expandable heap
```

Thank you