

# **Signals**

# Signals

- **Can be termed as a way of Inter-process communication, but often it is not categorized as IPC (why!?)**
- **Processes can send each other “signals”, that is indicators of an event and receiver can “act” on the receipt of the signal**
  - **Integers are used to denote/signify each event**

# **Let's see signals in action using Linux command line**

- **Use of “kill” command**
  - **Does not mean “kill” literally every time**
  - **“kill” means send signal!**

# Signals

- **Signals are used in UNIX systems to notify a process that a particular event has occurred.**
- **Signal handling**
  - **Synchronous and asynchronous**
- **A signal handler (a function) is used to process signals**
  - **Signal is generated by particular event (asynchronous like segfault or synchronous like “kill” system call)**
  - **Signal is delivered to a process**
  - **Then, signal is “handled” by the handler**

# Signals

- **More about signals**
  - **Different signals are typically identified as different numbers**
  - **Operating systems provide system calls like kill() and signal() to enable processes to deliver and handle signals**
  - **sigaction()/signal()** - is used by a process to specify a “signal handler” – a code that should run on receiving a signal
  - **kill()** is used by a process to send another process a signal
  - **There are restrictions on which process can send which signal to other processes**

# Signals

- **Actions**
  - **Term** **Default action is to terminate the process.**
  - **Ign** **Default action is to ignore the signal.**
  - **Core** **Default action is to terminate the process and dump core (see core(5)).**
  - **Stop** **Default action is to stop the process.**
  - **Cont** **Default action is to continue the process if it is currently stopped.**

# Demo

- Let's see a demo of signals with respect to processes
- Let's see signal.h
  - /usr/include/signal.h
  - /usr/include/asm-generic/signal.h
  - /usr/include/linux/signal.h
  - /usr/include/sys/signal.h
  - /usr/include/x86\_64-linux-gnu/asm/signal.h
  - /usr/include/x86\_64-linux-gnu/sys/signal.h
- man 7 signal
- Important signals: **SIGKILL, SIGUSR1, SIGSEGV, SIGALRM, SIGCLD, SIGINT, SIGPIPE, ...**

# Signal handling by OS

```
Process 12323 {  
    signal(19, abcd);  
}
```

**OS: sys\_signal {**

**Note down that process 12323  
should handle signal number 19  
with function abcd**

```
}
```

```
Process P1 {  
    kill (12323, 19) ;  
}
```

**OS: sys\_kill {**

**Note down in PCB of process 12323 that  
signal number 19 is pending for you.**

```
}  
  
When process 12323 is scheduled, at that  
time the OS will check for pending signals,  
and invoke the appropriate signal handler for  
a pending signal.
```

# Sigaction: POSIX

```
#include <signal.h>
```

```
typedef void (*sighandler_t)(int);
```

```
sighandler_t signal(int signum, sighandler_t  
handler);
```

# Threads and Signals

- **Signal handling Options:**
  - **Deliver the signal to the thread to which the signal applies**
  - **Deliver the signal to every thread in the process**
  - **Deliver the signal to certain threads in the process**
  - **Assign a specific thread to receive all signals for the process**

# The “errno” And error conventions

# **Memory Management – Continued**

**More on Linking, Loading, Paging**

# **Review of last class**

- MMU : Hardware features for MM
- OS: Sets up MMU for a process, then schedules process
- Compiler : Generates object code for a particular OS + MMU architecture
- MMU: Detects memory violations and raises interrupt --> Effectively passing control to OS

# More on Linking and Loading

- **Static Linking:** All object code combined at link time and a big object code file is created
- **Static Loading:** All the code is loaded in memory at the time of exec()
- **Problem:** Big executable files, need to load functions even if they do not execute
- **Solution:** Dynamic Linking and Dynamic Loading

# Dynamic Linking

- **Linker is normally invoked as a part of compilation process**
  - **Links**
    - function code to function calls
    - references to global variables with “extern” declarations
- **Dynamic Linker**
  - **Does not combine function code with the object code file**
  - **Instead introduces a “stub” code that is indirect reference to actual code**
  - **At the time of “loading” (or executing!) the program in memory, the “link-loader” (part of OS!) will pick up the relevant code from the library machine code file (e.g. libc.so.6)**

# Dynamic Linking on Linux

```
#include <stdio.h>

int main() {
    int a, b;
    scanf("%d%d", &a, &b);
    printf("%d %d\n", a, b);
    return 0;
}
```

## PLT: Procedure Linkage Table

used to call external procedures/functions whose address is to be resolved by the dynamic linker at run time.

### Output of objdump -x -D

#### Disassembly of section .text:

```
0000000000001189 <main>:
```

```
11d4:    callq  1080 <printf@plt>
```

#### Disassembly of section .plt.got:

```
0000000000001080 <printf@plt>:
```

```
1080:    endbr64
```

```
1084:    bnd jmpq *0x2f3d(%rip)      # 3fc8
<printf@GLIBC_2.2.5>
```

```
108b:    nopl  0x0(%rax,%rax,1)
```

# Dynamic Loading

- **Loader**
  - Loads the program in memory
  - Part of exec() code
  - Needs to understand the format of the executable file (e.g. the ELF format)
- **Dynamic Loading**
  - Load a part from the ELF file only if needed during execution
  - Delayed loading
  - Needs a more sophisticated memory management by operating system – to be seen during this series of lectures

# Dynamic Linking, Loading

- Dynamic linking necessarily demands an advanced type of loader that understands dynamic linking
  - Hence called ‘link-loader’
  - Static or dynamic loading is still a choice
- Question: which of the MMU options will allow for which type of linking, loading ?

# **Continuous memory management**

# What is Continuous memory management?

- Entire process is hosted as one continuous chunk in RAM
- Memory is typically divided into two partitions
  - One for OS and other for processes
  - OS most typically located in “high memory” addresses, because interrupt vectors map to that location (Linux, Windows) !

# Hardware support needed: base + limit (or relocation + limit)

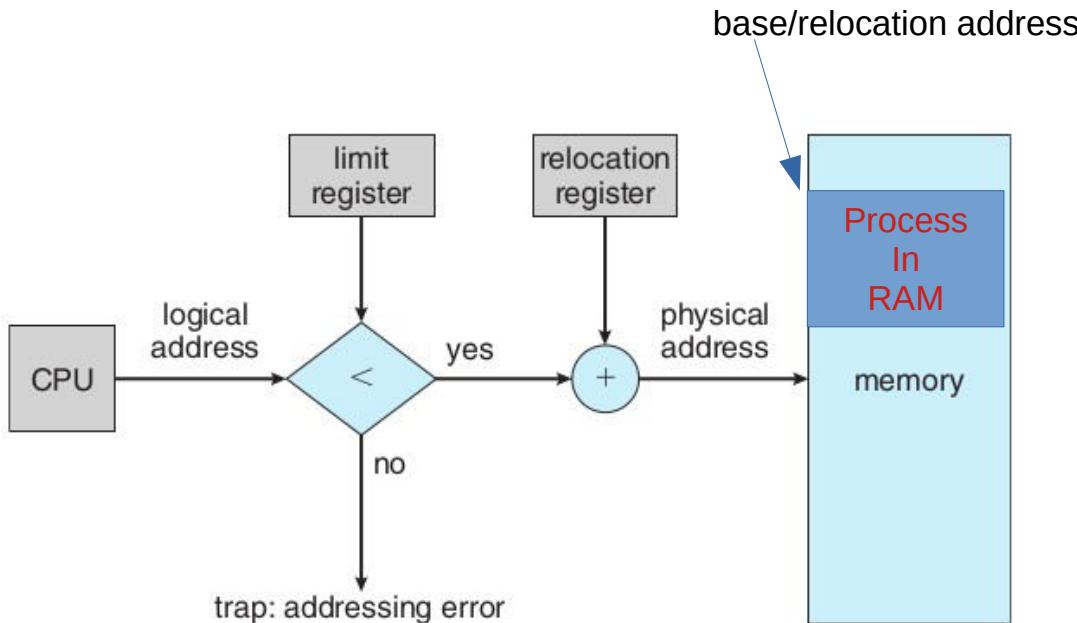


Figure 9.6 Hardware support for relocation and limit registers.

# Problems faced by OS

- Find a continuous chunk for the process being forked
- Different processes are of different sizes
  - Allocate a size parameter in the PCB
- After a process is over – free the memory occupied by it
- Maintain a list of free areas, and occupied areas
  - Can be done using an array, or linked list

# Variable partition scheme

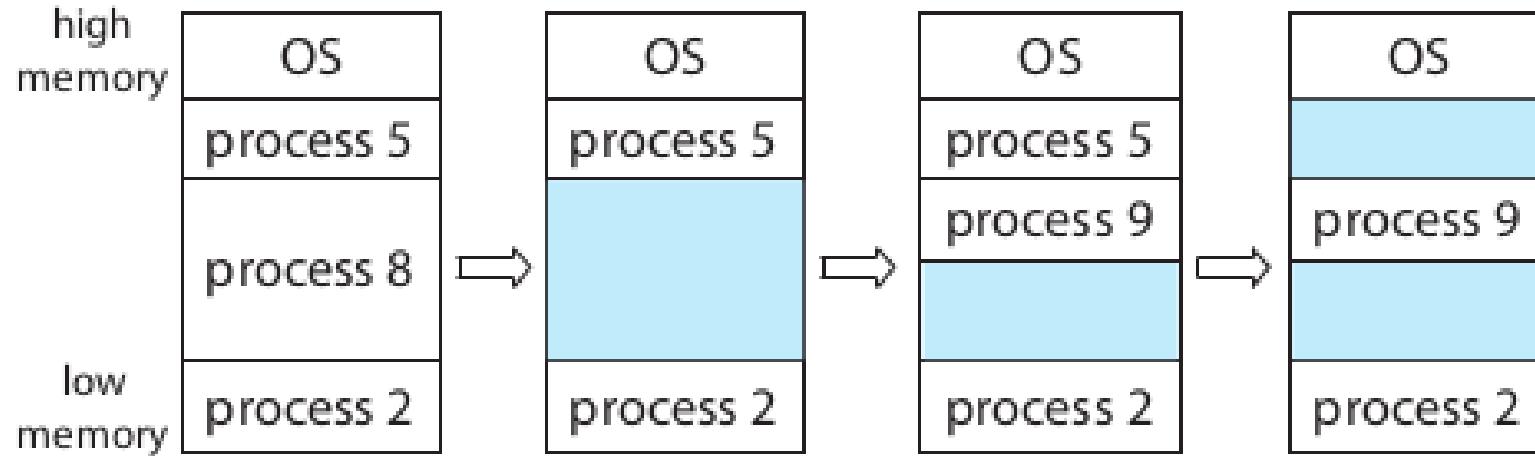


Figure 9.7 Variable partition.

# **Problem: how to find a “hole” to fit in new process**

- Suppose there are 3 free memory regions of sizes 30k, 40k, 20k
- The newly created process (during fork() + exec()) needs 15k
- Which region to allocate to it ?

# Strategies for finding a free chunk

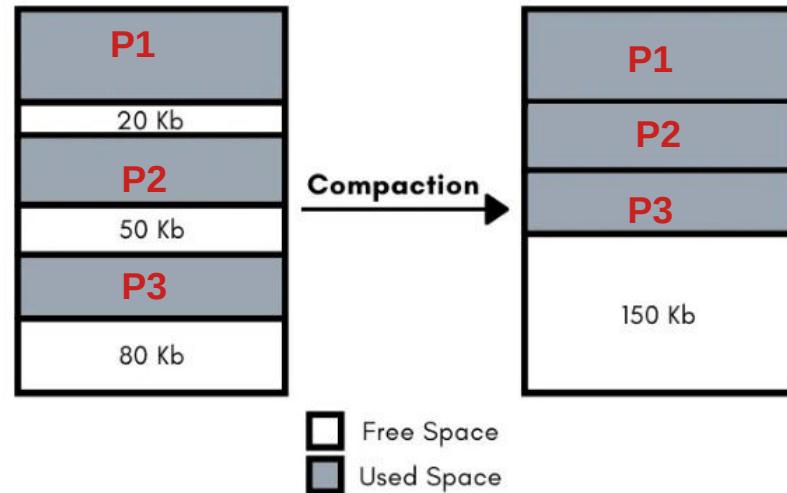
- 6k, 17k, 16k, 40k holes . Need 15k.
- **Best fit:** Find the smallest hole, larger than process. **Ans: 16k**
- **Worst fit:** Find the largest hole. **Ans: 40k**
- **First fit:** Find the “first” hole larger than the process. **Ans: 17k**

# Problem : External fragmentation

- Free chunks: 30k, 40k, 20k
- The newly created process (during `fork()` + `exec()`) needs 50k
- Total free memory:  $30+40+20 = 90\text{k}$ 
  - But can't allocate 50k !

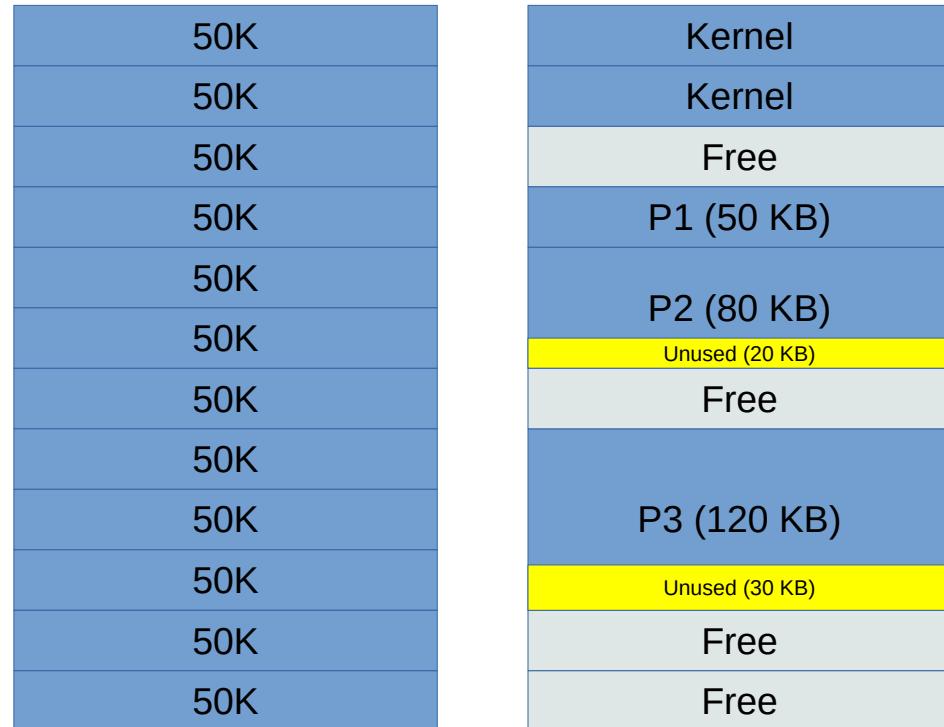
# Solution to external fragmentation

- Compaction !
- OS moves the process chunks in memory to make available continuous memory region
  - Then it must update the memory management information in PCB (e.g. base of the process) of each process
- Time consuming
- Possible only if the relocation+limit scheme of MMU is available



# Another solution to external fragmentation: Fixed size partitions

- Fixed partition scheme
- Memory is divided by OS into chunks of equal size: e.g., say, 50k
  - If total 1M memory, then 20 such chunks
- Allocate one or more chunks to a process, such that the total size is  $\geq$  the size of the process
  - E.g. if request is 50k, allocate 1 chunk
  - If request is 40k, still allocate 1 chunk
  - If request is 60k, then allocate 2 chunks
- Leads to internal fragmentation
  - space wasted in the case of 40k or 60k requests above



# Fixed partition scheme

- OS needs to keep track of
  - Which partition is free and which is used by which process
  - Free partitions can simply be tracked using a bitmap or a list of numbers
  - Each process's PCB will contain list of partitions allocated to it

# Solution to internal fragmentation

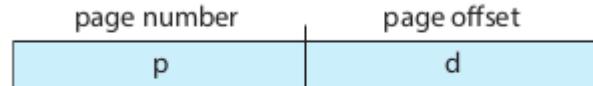
- Reduce the size of the fixed sized partition
- How small then ?
  - Smaller partitions mean more overhead for the operating system in allocating deallocating

# Paging

# An extended version of fixed size partitions

- Partition = page
  - Process = logically continuous sequence of bytes, divided in ‘page’ sizes
  - Memory divided into equally sized page ‘frames’
- Important distinction
  - Process need not be continuous in RAM
  - Different page sized chunks of process can go in any page frame
  - Page table to map pages into frames

# Logical address seen as



# Paging hardware

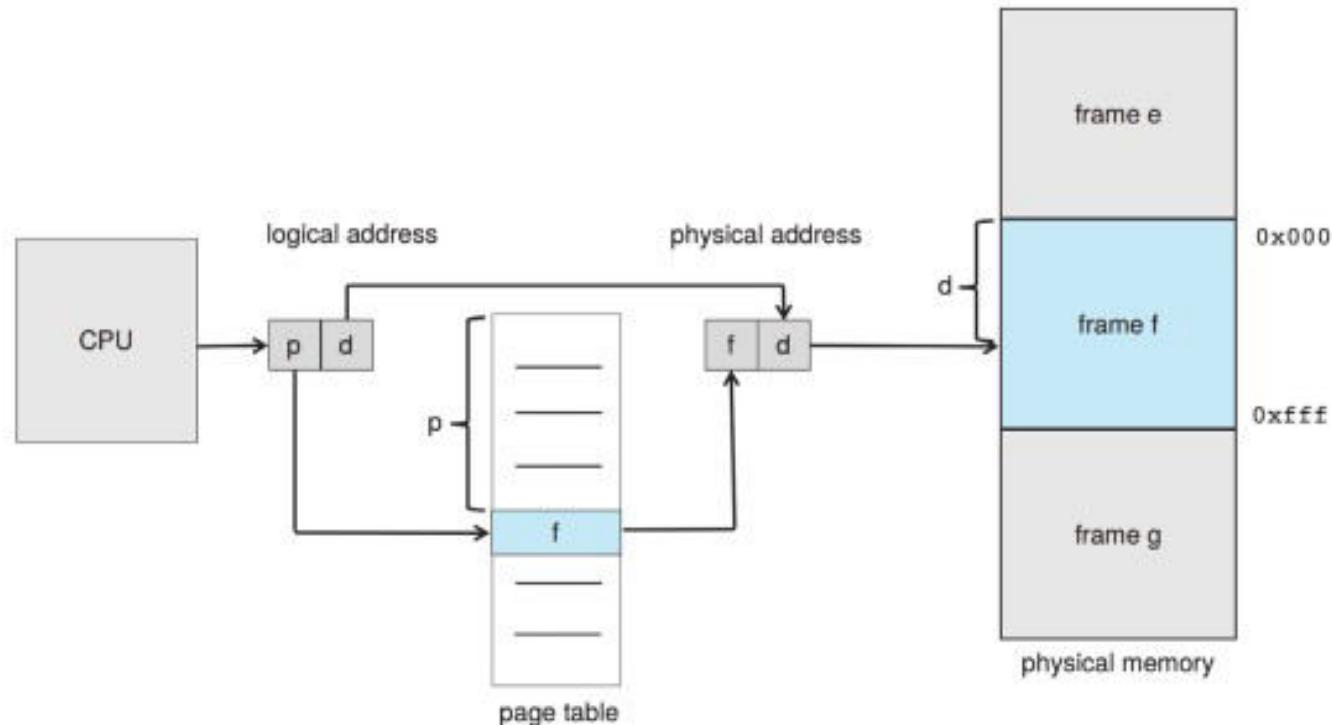


Figure 9.8 Paging hardware.

# MMU's job

To translate a logical address generated by the CPU to a physical address:

1. Extract the page number  $p$  and use it as an index into the page table.

(Page table location is stored in a hardware register

Also stored in PCB of the process, so that it can be used to load the hardware register on a context switch)

2. Extract the corresponding frame number  $f$  from the page table.

3. Replace the page number  $p$  in the logical address with the frame number  $f$ .

# Job of OS

- Allocate a page table for the process, at time of fork()/exec()
  - Allocate frames to process
  - Fill in page table entries
- In PCB of each process, maintain
  - Page table location (address)
  - List of pages frames allocated to this process
- During context switch of the process, load the PTBR using the PCB

# Job of OS

- **Maintain a list of all page frames**
  - Allocated frames
  - Free Frames (called frame table)
  - Can be done using simple linked list
  - Innovative data structures can also be used to maintain free and allocated frames list (e.g. xv6 code)
  -

free-frame list

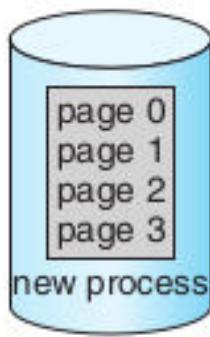
14

13

18

20

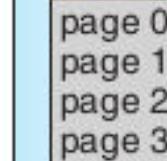
15



(a)

free-frame list

15



new-process page table

(b)

free-frame list

13 page 1

14 page 0

15

16

17

18 page 2

19

20 page 3

21

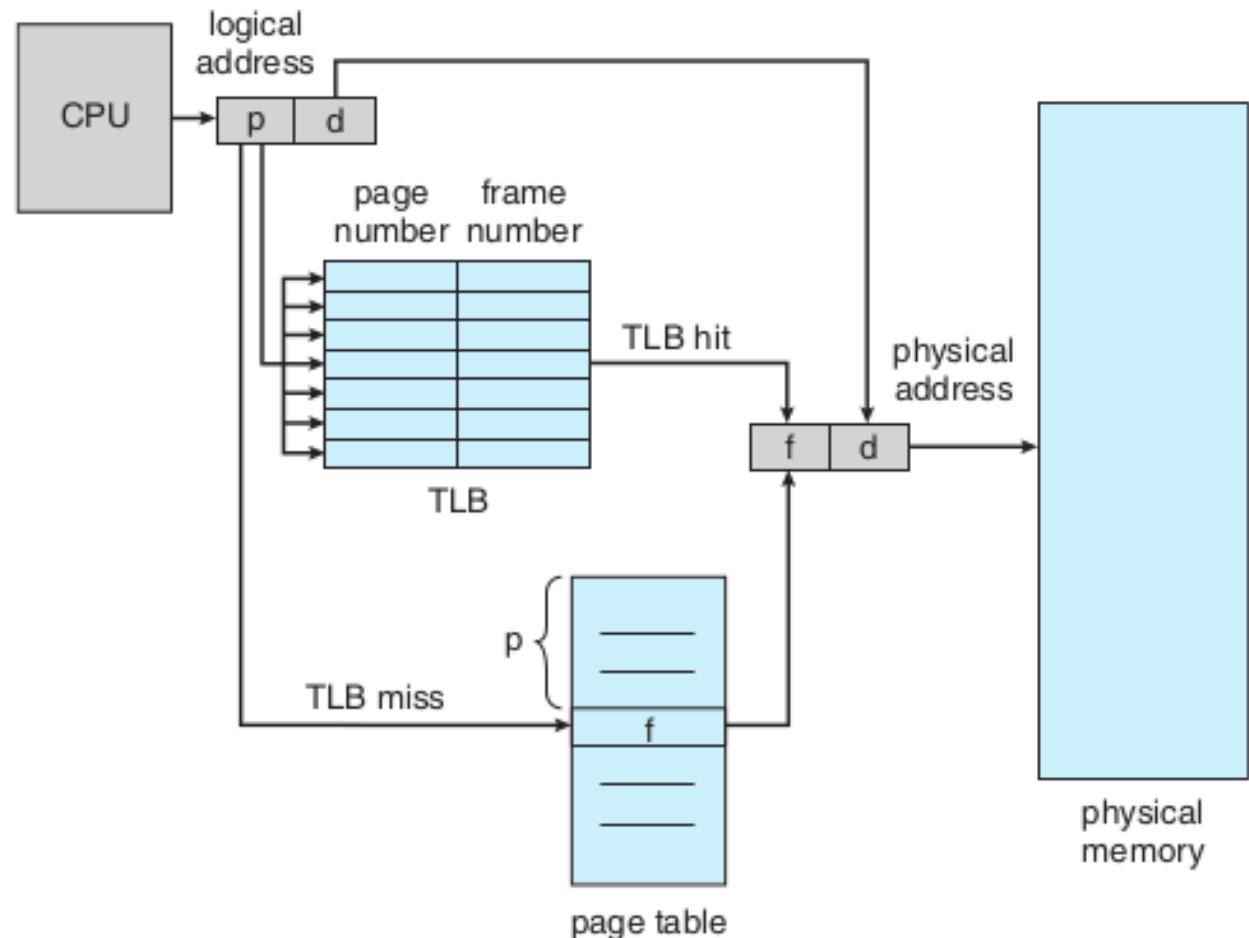
Figure 9.11 Free frames (a) before allocation and (b) after allocation.

# Disadvantage of Paging

- Each memory access results in two memory accesses!
  - One for page table, and one for the actual memory location !
  - Done as part of execution of instruction in hardware (not by OS!)
  - Slow down by 50%

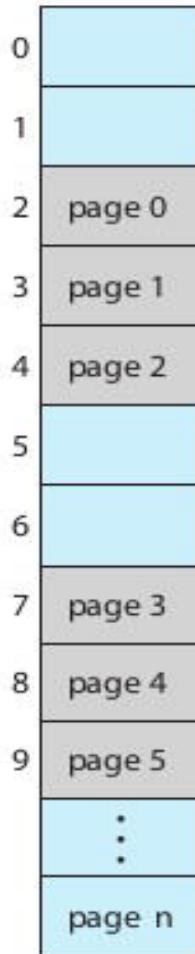
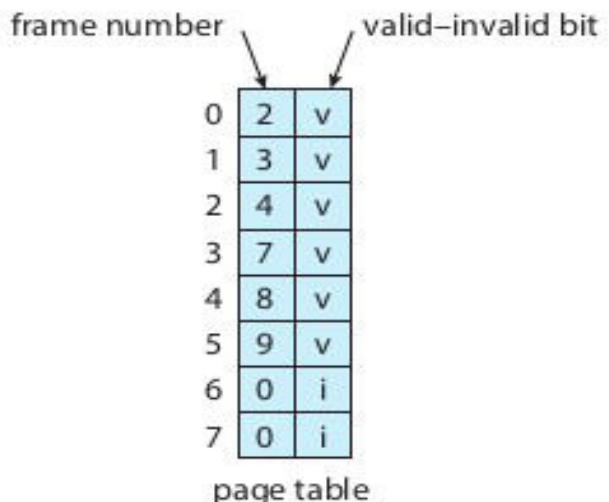
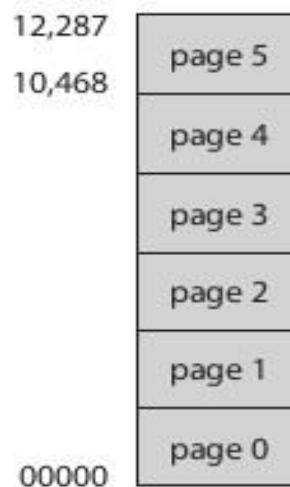
# Speeding up paging

- Translation Lookaside Buffer (TLB)
- Part of CPU hardware
- A cache of Page table entries
- Searched in parallel for a page number



# Speedup due to TLB

- Hit ratio
- Effective memory access time
  - = Hit ratio \* 1 memory access time + miss ratio \* 2 memory access time
- Example: memory access time 10ns, hit ratio = 0.8, then
  - effective access time =  $0.80 \times 10 + 0.20 \times 20$
  - = 12 nanoseconds



## Memory protection with paging

Figure 9.13 Valid (v) or invalid (i) bit in a page table.

# X86 PDE and PTE

31

Page table physical page number	A V L	G S T	P A D	0 A D	A C D	C W T	W U T	U W P	1 0 0
---------------------------------	-------------	-------------	-------------	-------------	-------------	-------------	-------------	-------------	-------------

PDE

31

Physical page number	A V L	G A T	P A D	0 A D	A C D	C W T	W U P	1 0 0
----------------------	-------------	-------------	-------------	-------------	-------------	-------------	-------------	-------------

PTE

P Present  
W Writable  
U User

WT 1=Write-through, 0=Write-back

CD Cache disabled

A Accessed

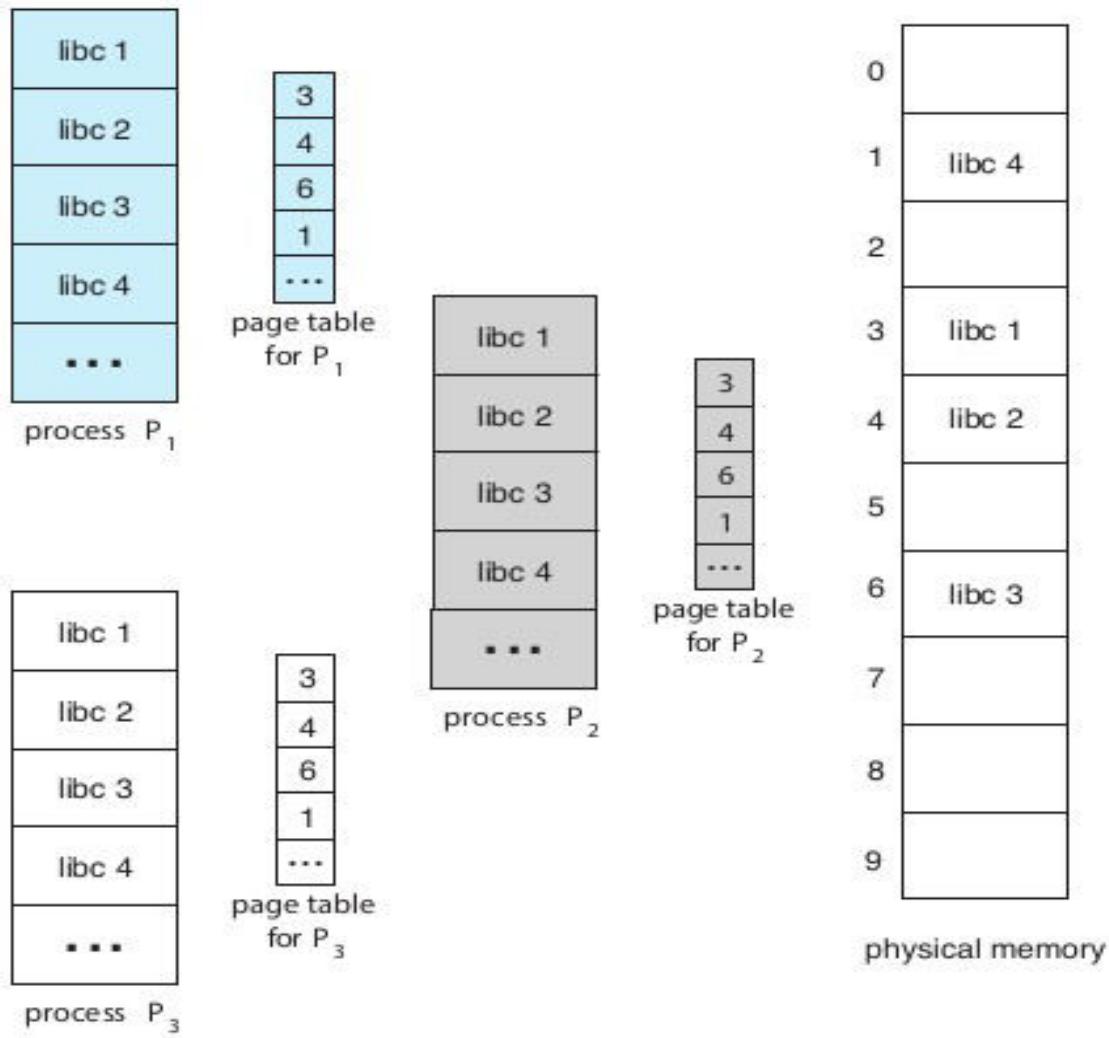
D Dirty

PS Page size (0=4KB, 1=4MB)

PAT Page table attribute index

G Global page

AVL Available for system use



# Shared pages (e.g. library) with paging

Figure 9.14 Sharing of standard C library in a paging environment.

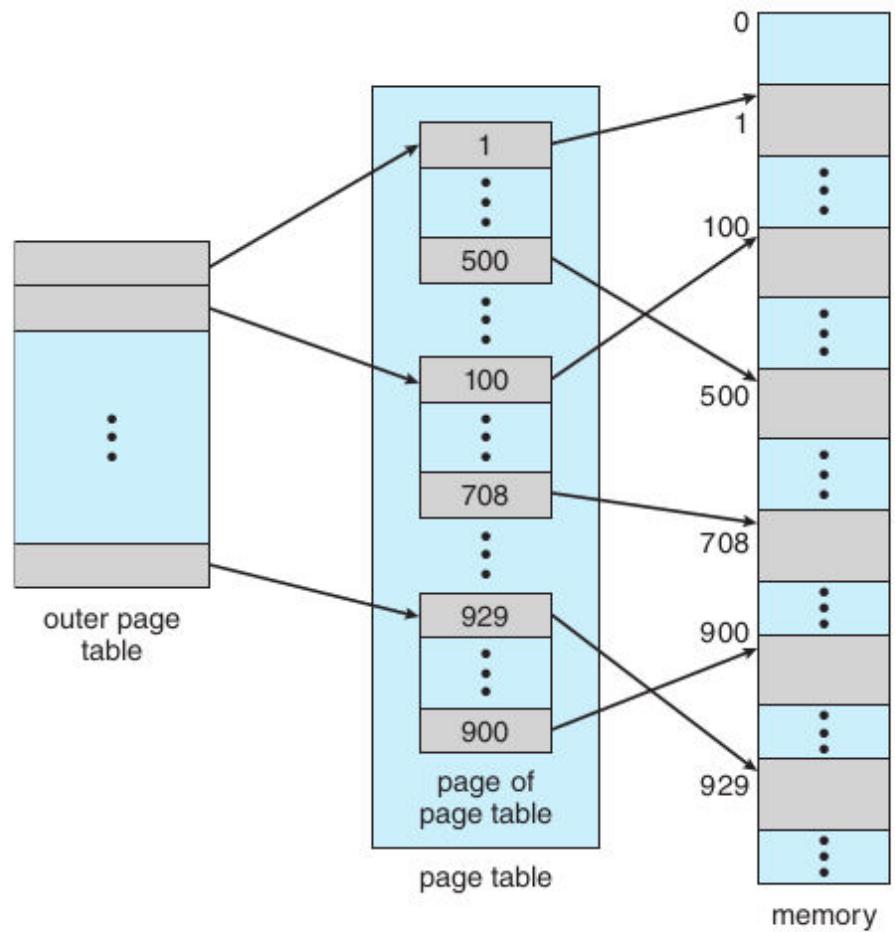
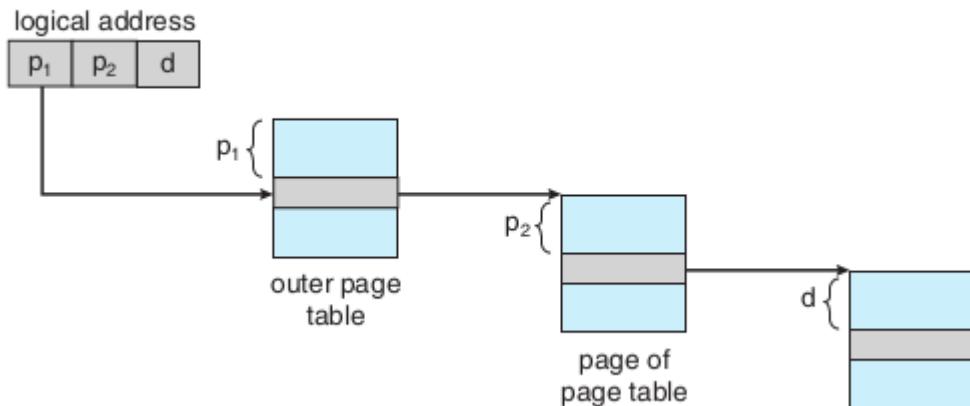
# Paging: problem of large PT

- 64 bit address
- Suppose 20 bit offset
  - That means  $2^{20} = 1 \text{ MB}$  pages
  - 44 bit page number:  $2^{44}$  that is trillion sized page table!
  - Can't have that big continuous page table!

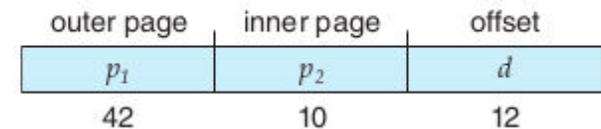
# Paging: problem of large PT

- 32 bit address
- Suppose 12 bit offset
  - That means  $2^{12} = 4 \text{ KB pages}$
  - 20 bit page number:  $2^{20}$  that is a million entries
  - Can't always have that big continuous page table as well, for each process!

## Hierarchical paging



**Figure 9.15** A two-level page-table scheme.



# More hierarchy

2nd outer page	outer page	inner page	offset
$p_1$	$p_2$	$p_3$	$d$
32	10	10	12

# Problems with hierarchical paging

- More number of memory accesses with each level !
  - Too slow !
- OS data structures also needed in that proportion

# Hashed page table

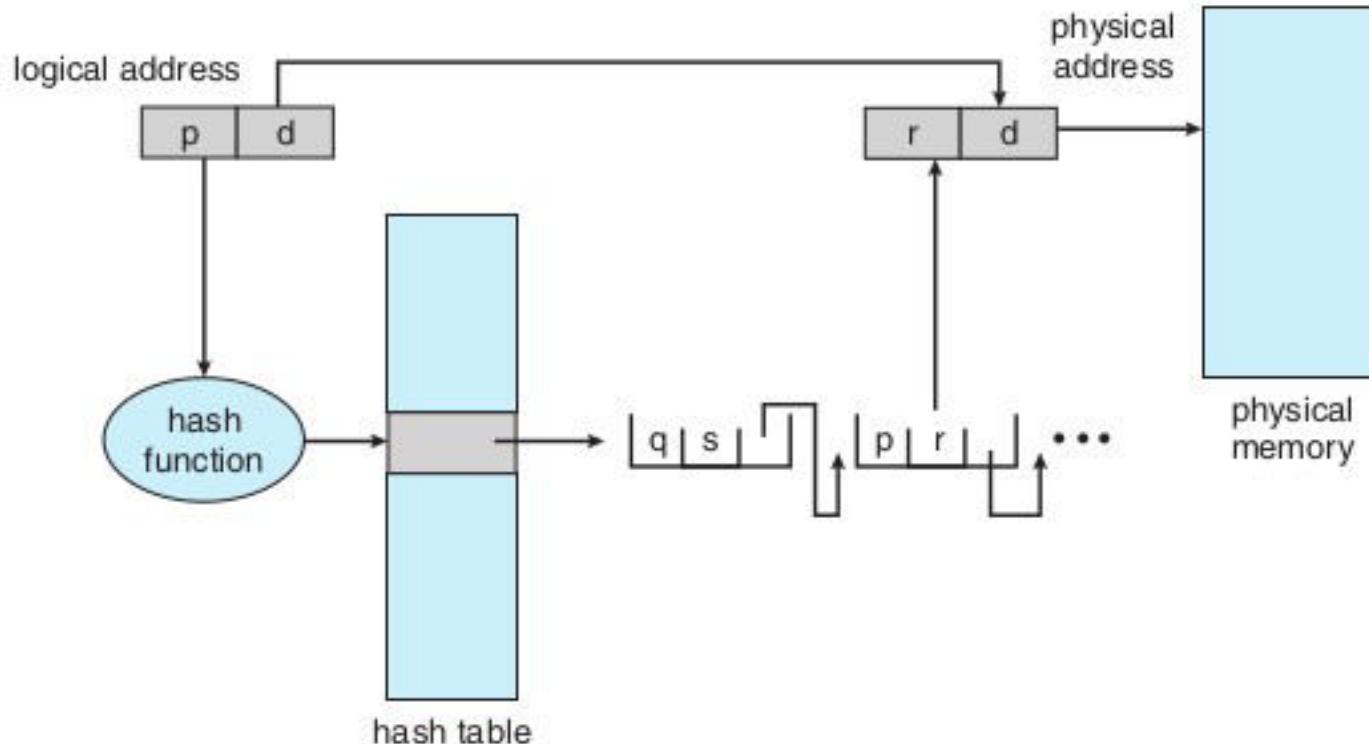


Figure 9.17 Hashed page table.

# Inverted page table

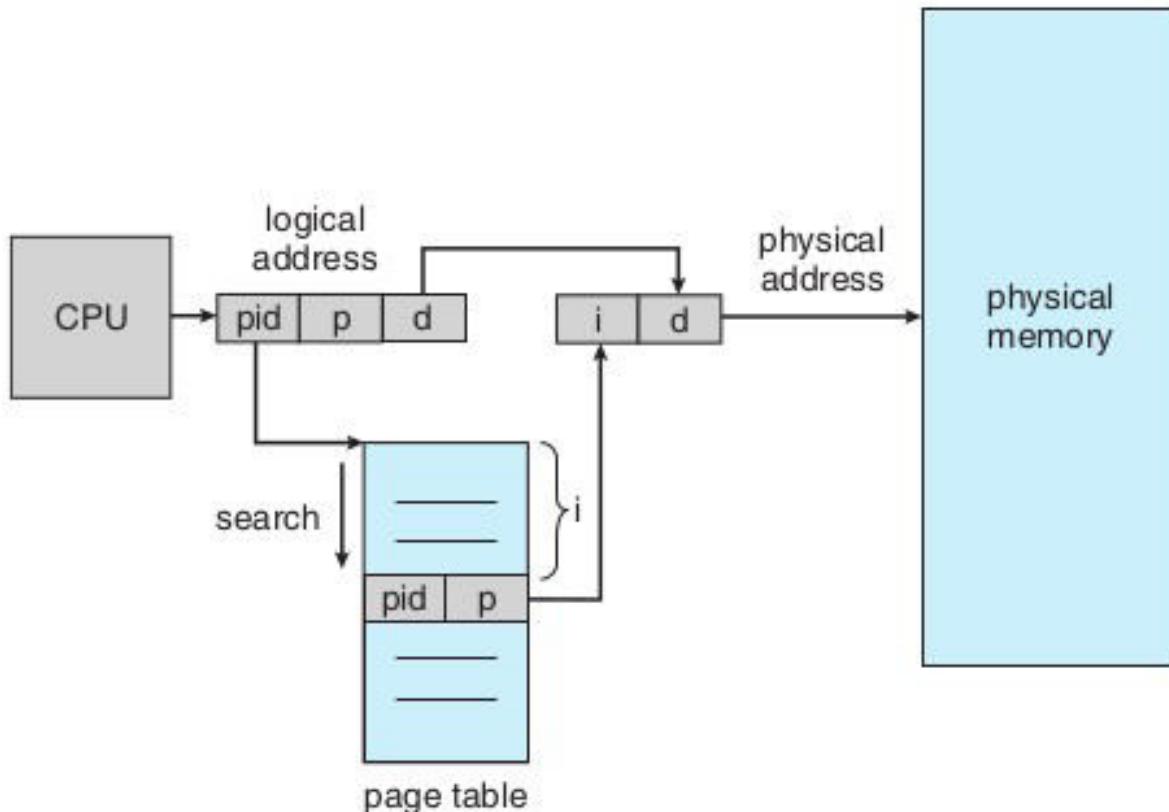


Figure 9.18 Inverted page table.

Normal page table – one per process --> Too much memory consumed

Inverted page table : global table – only one  
Needs to store PID in the table entry

Examples of systems using inverted page tables include the 64-bit Ultra SPARC and Power PC

virtual address consists of a triple:  
<process-id, page-number,  
offset>

# Case Study: Oracle SPARC Solaris

- 64 bit SPARC processor , 64 bit Solaris OS
- Uses Hashed page tables
  - one for the kernel and one for all user processes.
  - Each hash-table entry : base + span (#pages)
    - Reduces number of entries required

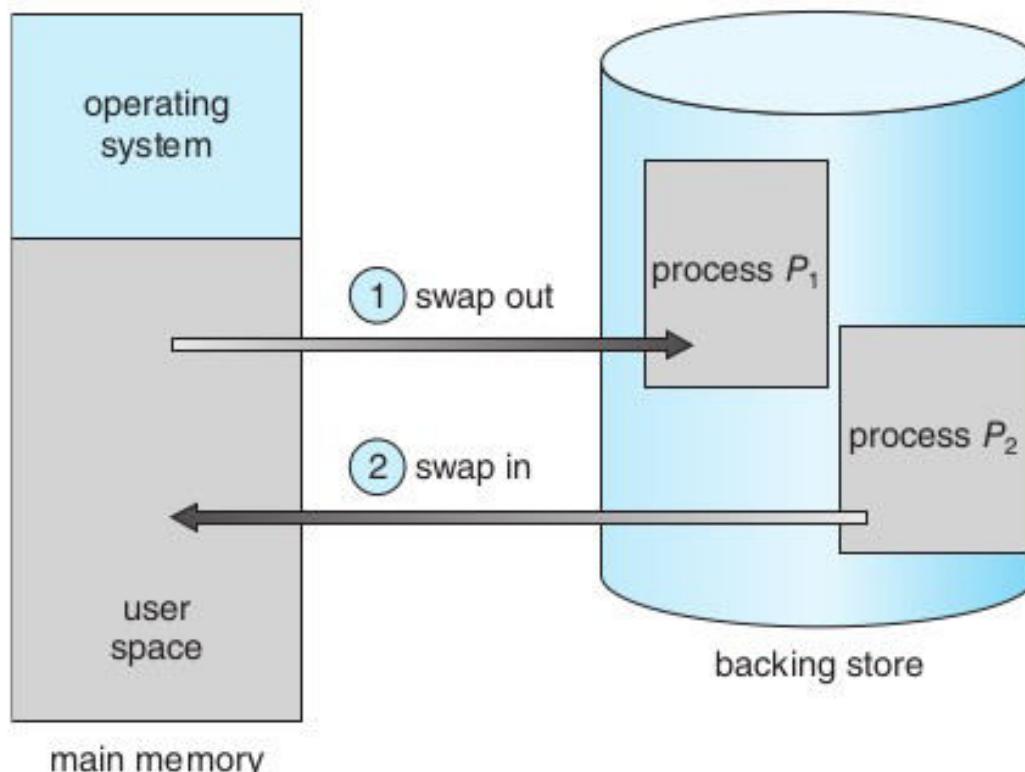
# Case Study: Oracle SPARC Solaris

- Caching levels: TLB (on CPU), TSB(in Memory), Page Tables (in Memory)
  - CPU implements a TLB that holds translation table entries ( TTE s) for fast hardware lookups.
  - A cache of these TTEs resides in a in-memory translation storage buffer (TSB ), which includes an entry per recently accessed page
  - When a virtual address reference occurs, the hardware searches the TLB for a translation.
  - If none is found, the hardware walks through the in memory TSB looking for the TTE that corresponds to the virtual address that caused the lookup

# Case Study: Oracle SPARC Solaris

- If a match is found in the TSB , the CPU copies the TSB entry into the TLB , and the memory translation completes.
- If no match is found in the TSB , the kernel is interrupted to search the hash table.
- The kernel then creates a TTE from the appropriate hash table and stores it in the TSB for automatic loading into the TLB by the CPU memory-management unit.
- Finally, the interrupt handler returns control to the MMU , which completes the address translation and retrieves the requested byte or word from main memory.

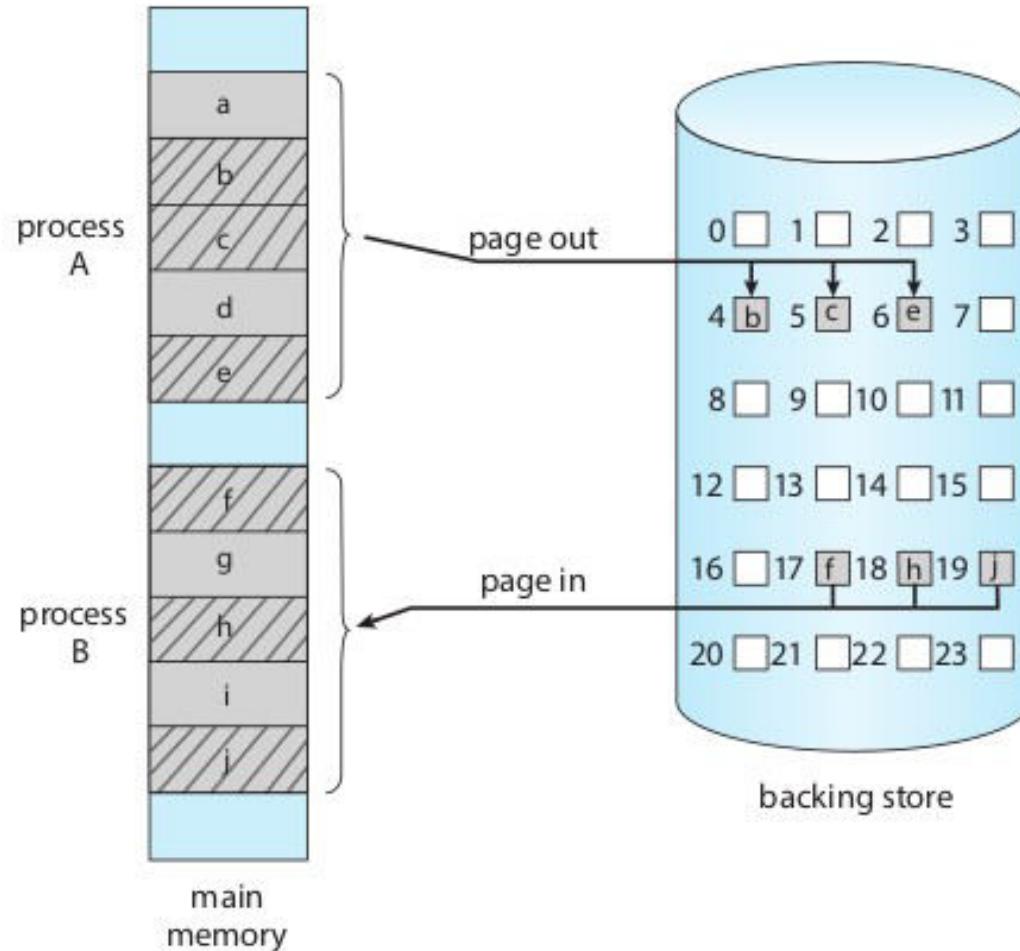
# Swapping



**Figure 9.19** Standard swapping of two processes using a disk as a backing store.

# Swapping

- Standard swapping
  - Entire process swapped in or swapped out
  - With continuous memory management
- Swapping with paging
  - Some pages are “paged out” and some “paged in”
  - Term “paging” refers to paging with swapping now



**Figure 9.20** Swapping with paging.

# Words of caution about ‘paging’

- Not as simple as it sounds when it comes to implementation
  - Writing OS code for this is challenging

# **Memory Management Basics**

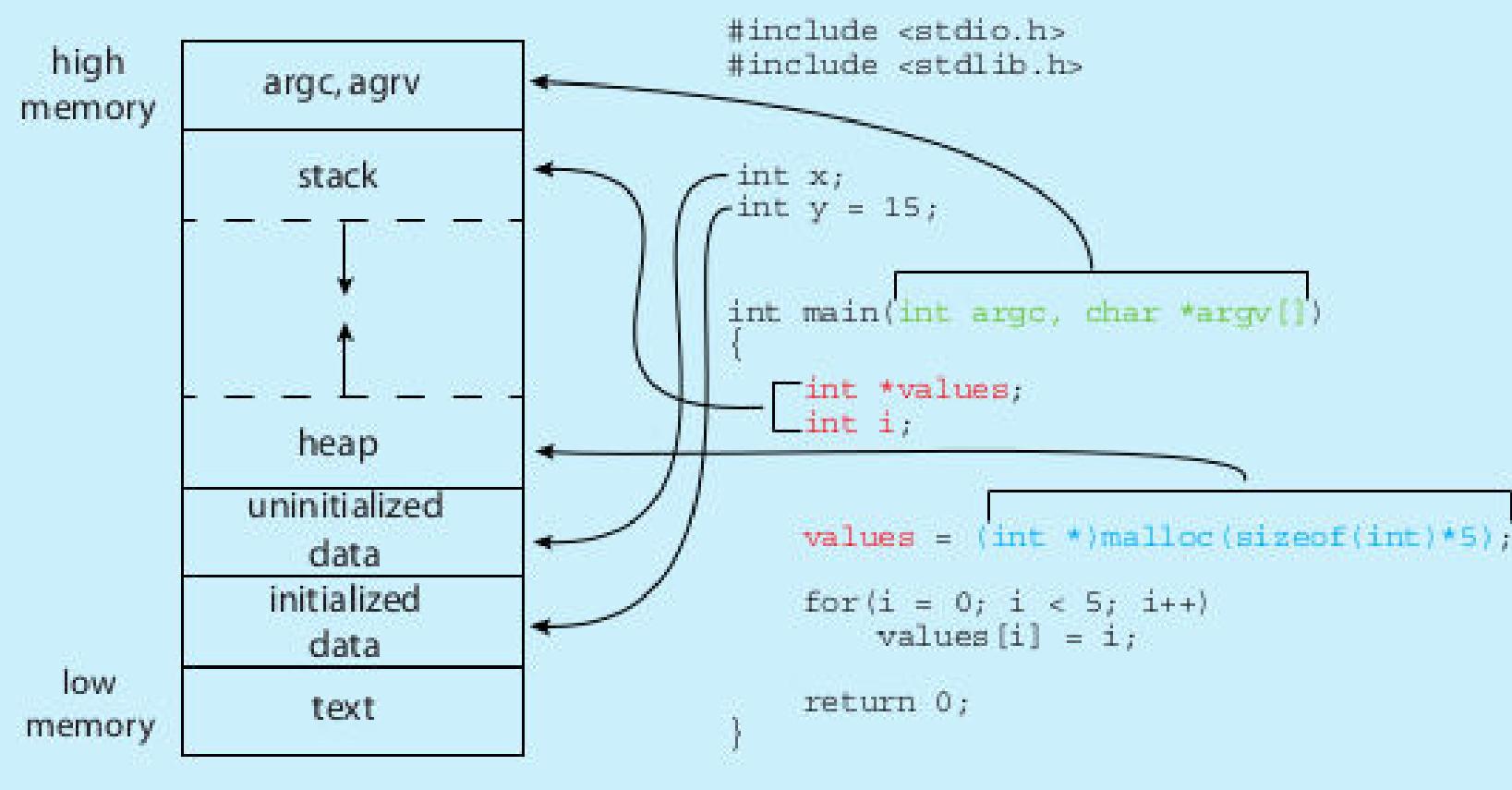
# Summary

- Understanding how the processor architecture drives the memory management features of OS and system programs (compilers, linkers)
- Understanding how different hardware designs lead to different memory management schemes by operating systems

# Addresses issued by CPU

- During the entire ‘on’ time of the CPU
  - Addresses are “issued” by the CPU on address bus
  - One address to fetch instruction from location specified by PC
  - Zero or more addresses depending on instruction
    - e.g. mov \$0x300, r1 # move contents of address 0x300 to r1 --> one extra address issued on address bus

# Memory layout of a C program



\$ size /bin/ls

text	data	bss	dec	hex	filename
128069	4688	4824	137581	2196d	/bin/ls

# Desired from a multi-tasking system

- Multiple processes in RAM at the same time (multi-programming)
- Processes should not be able to see/touch each other's code, data (globals), stack, heap, etc.
- Further advanced requirements
  - Process could reside anywhere in RAM
  - Process need not be continuous in RAM
  - Parts of process could be moved anywhere in RAM

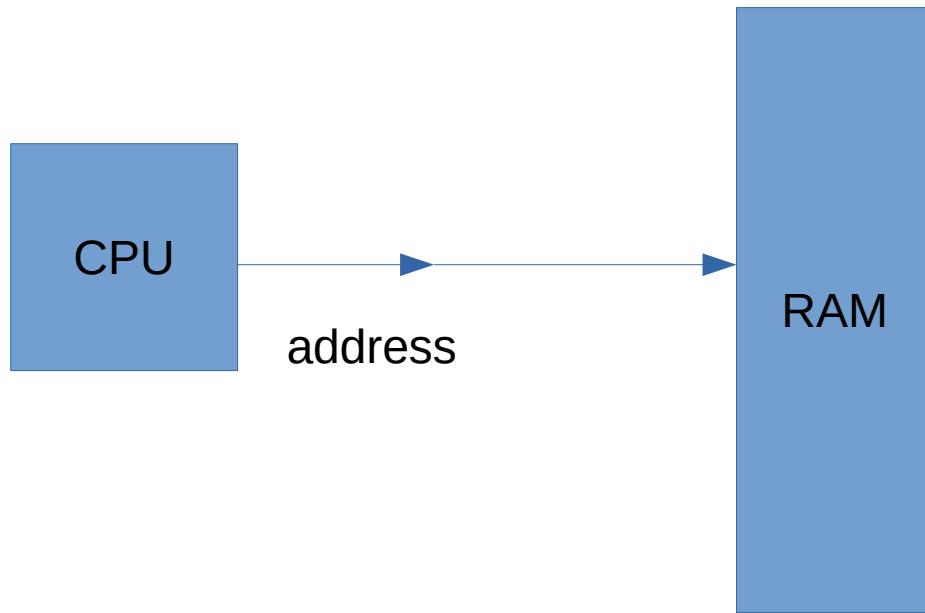
# Different ‘times’

- Different actions related to memory management for a program are taken at different times. So let's know the different ‘times’
- Compile time
  - When compiler is compiling your C code
- Load time
  - When you execute “./myprogram” and it's getting loaded in RAM by loader i.e. `exec()`
- Run time
  - When the process is alive, and getting scheduled by the OS

# Different types of Address binding

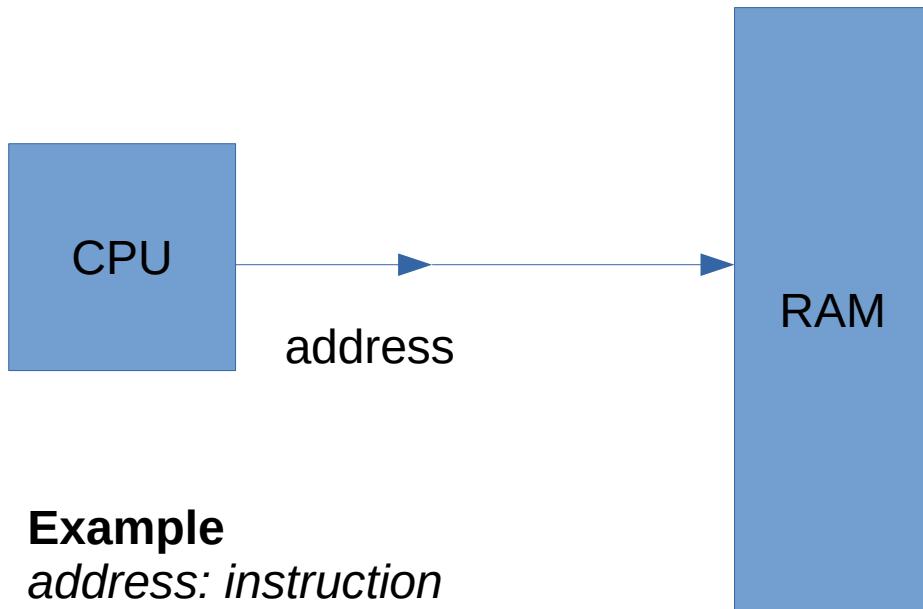
- Compile time address binding
    - Address of code/variables is fixed by compiler
    - Very rigid scheme
    - Location of process in RAM can not be changed ! Non-relocatable code.
  - Load time address binding
    - Address of code/variables is fixed by loader
    - Location of process in RAM is decided at load time, but can't be changed later
    - Flexible scheme, relocatable code
  - Run time address binding
    - Address of code/variables is fixed at the time of executing the code
    - Very flexible scheme , highly relocatable code
    - Location of process in RAM is decided at load time, but CAN be changed later also
- Which binding is actually used, is mandated by processor features + OS

# Simplest case



- Suppose the address issued by CPU reaches the RAM controller directly

# Simplest case



## Example

*address: instruction*

1000: mov \$0x300, r1

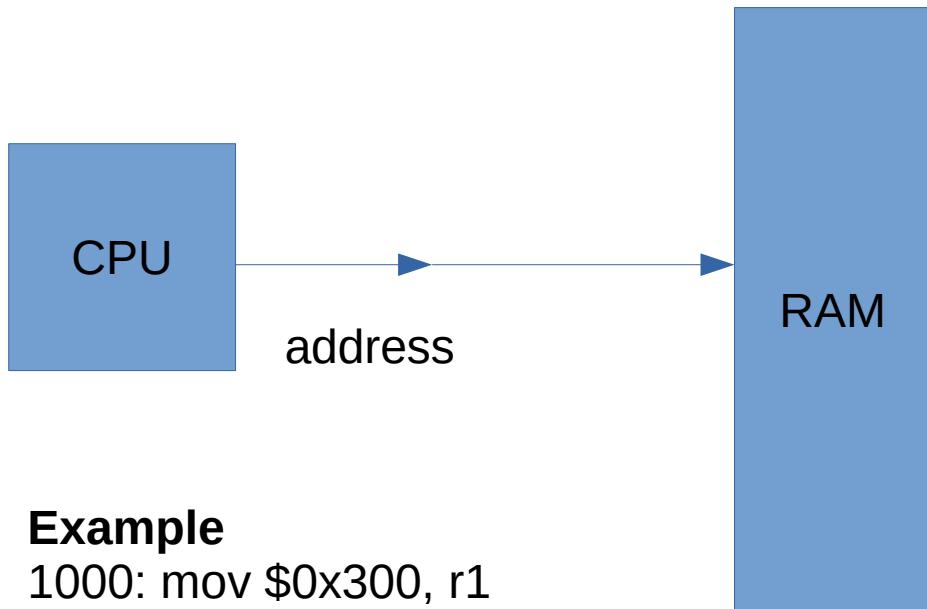
1004: add r1, -3

1008: jnz 1000

Sequence of addressed sent by CPU: 1000, 0x300, 1004, 1008, 1000, 0x300, ...

- How does this impact the compiler and OS ?
- When a process is running the addresses issued by it, will reach the RAM directly
- So exact addresses of globals, addresses in “jmp” and “call” must be part the machine instructions generated by compiler
  - How will the compiler know the addresses, at “compile time” ?

# Simplest case

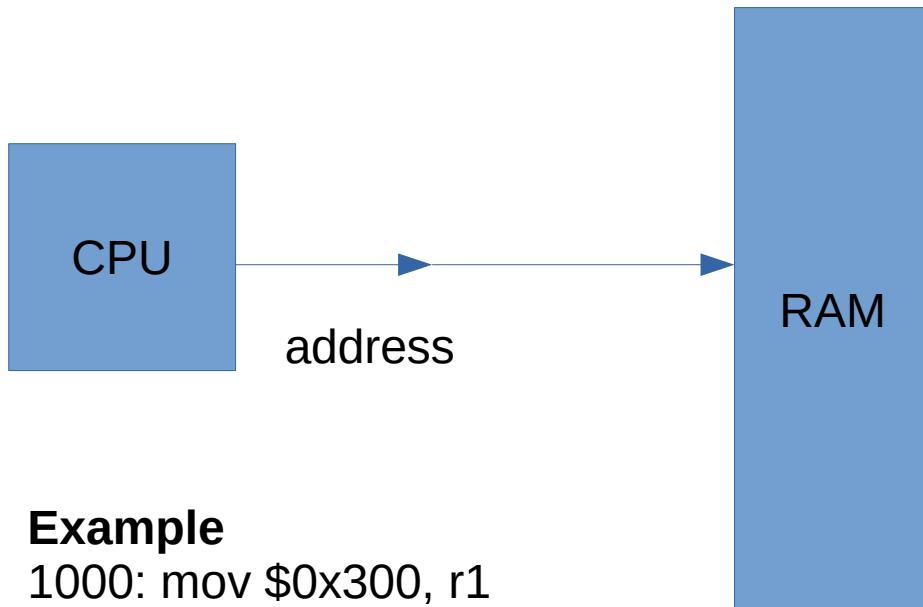


## Example

```
1000: mov $0x300, r1  
1004: add r1, -3  
1008: jnz 1000
```

- Solution: compiler assumes some fixed addresses for globals, code, etc.
- OS loads the program exactly at the same addresses specified in the executable file.  
**Non-relocatable code.**
- Now program can execute properly.

# Simplest case



## Example

```
1000: mov $0x300, r1  
1004: add r1, -3  
1008: jnz 1000
```

- Problem with this solution
  - Programs once loaded in RAM must stay there, can't be moved
  - What about 2 programs?
    - Compilers being “programs”, will make same assumptions and are likely to generate same/overlapping addresses for two different programs
    - Hence only one program can be in memory at a time !
    - No need to check for any memory boundary violations – all memory belongs to one process
- Example: DOS

# Base/Relocation + Limit scheme

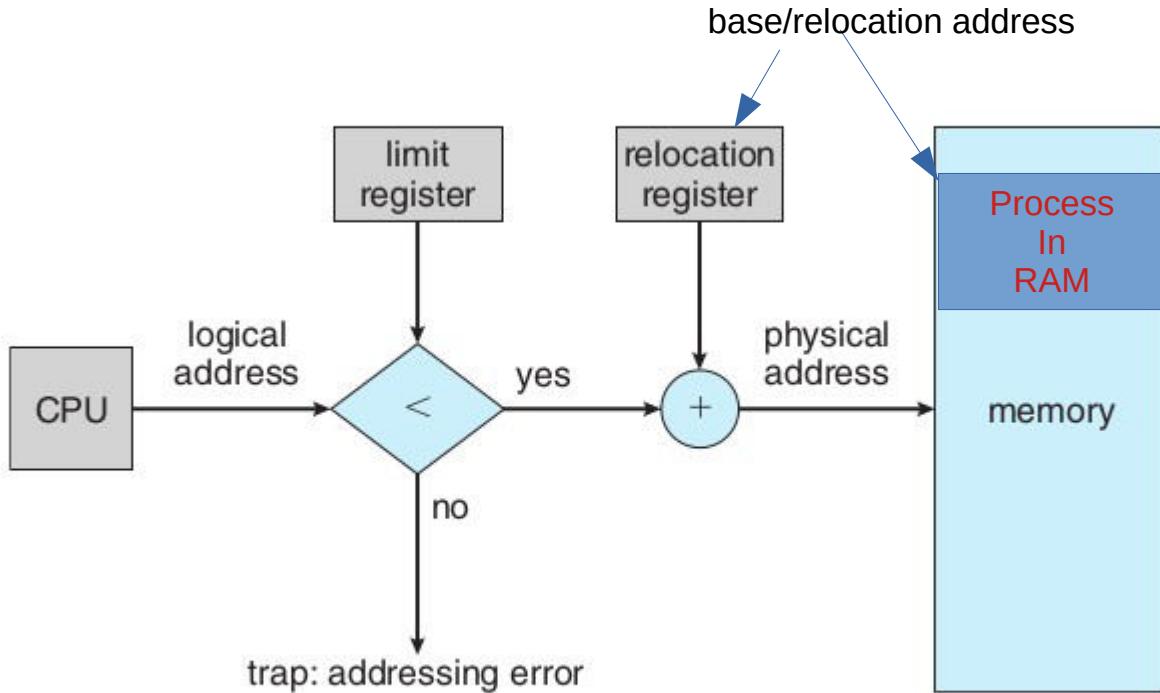
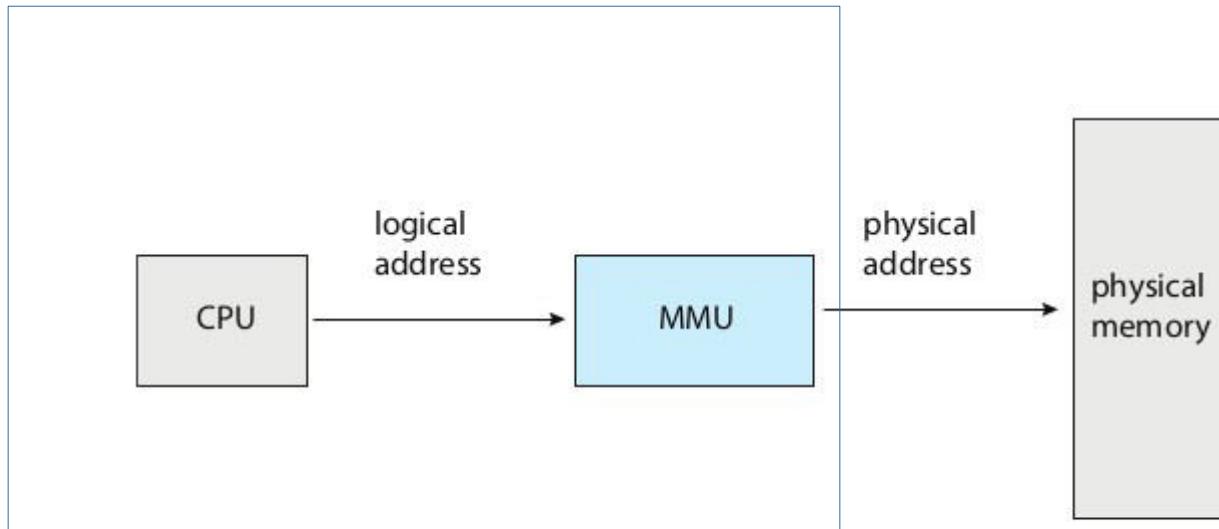


Figure 9.6 Hardware support for relocation and limit registers.

- Base and Limit are two registers inside CPU's Memory Management Unit
- 'base' is added to the address generated by CPU
- The result is compared with base+limit and if less passed to memory, else hardware interrupt is raised

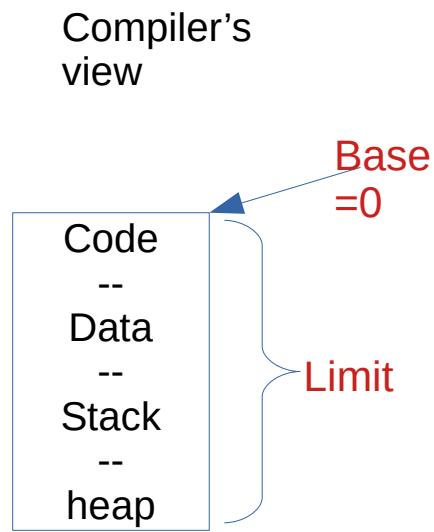
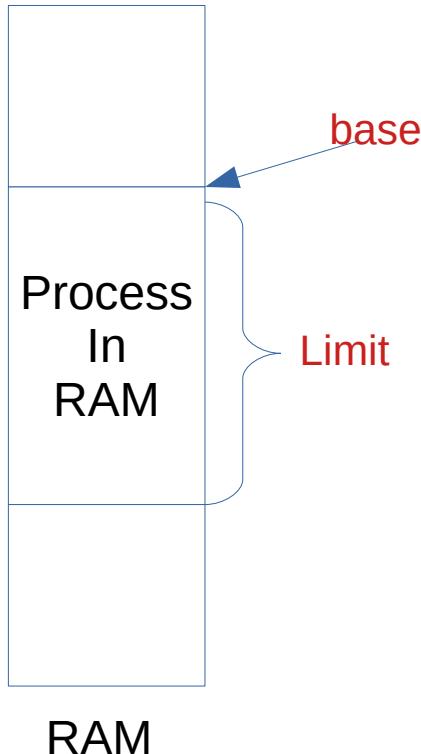
# Memory Management Unit (MMU)



**Figure 9.4** Memory management unit (MMU).

- Is part of the CPU chip, acts on every memory address issue by execution unit of the CPU
- In the scheme just discussed, the base, limit calculation parts are part of MMU

# Base/Relocation + Limit scheme



- Compiler's work
  - Assume that the process is one continuous chunk in memory, with a size limit
  - Assume that the process starts at address zero (!) and calculate addresses for globals, code, etc. And accordingly generate machine code

# Base/Relocation + Limit scheme

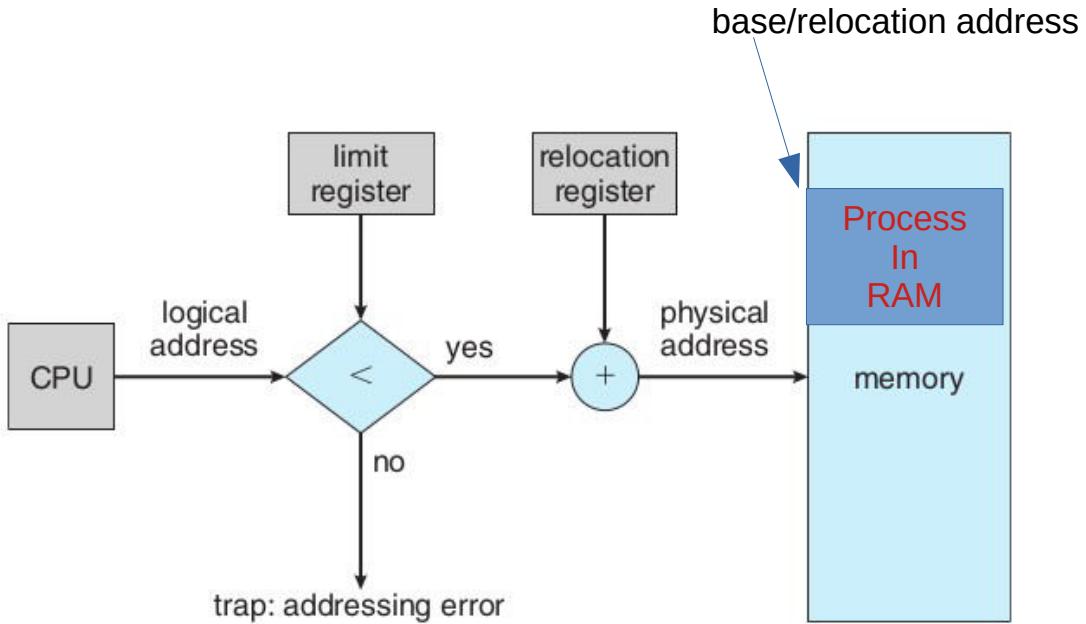


Figure 9.6 Hardware support for relocation and limit registers.

- OS's work

- While loading the process in memory – must load as one continuous segment
- Fill in the ‘base’ register with the actual address of the process in RAM.
- Setup the limit to be the size of the process as set by compiler in the executable file. *Remember the base+limit in OS’s own data structures.*

# Base/Relocation + Limit scheme

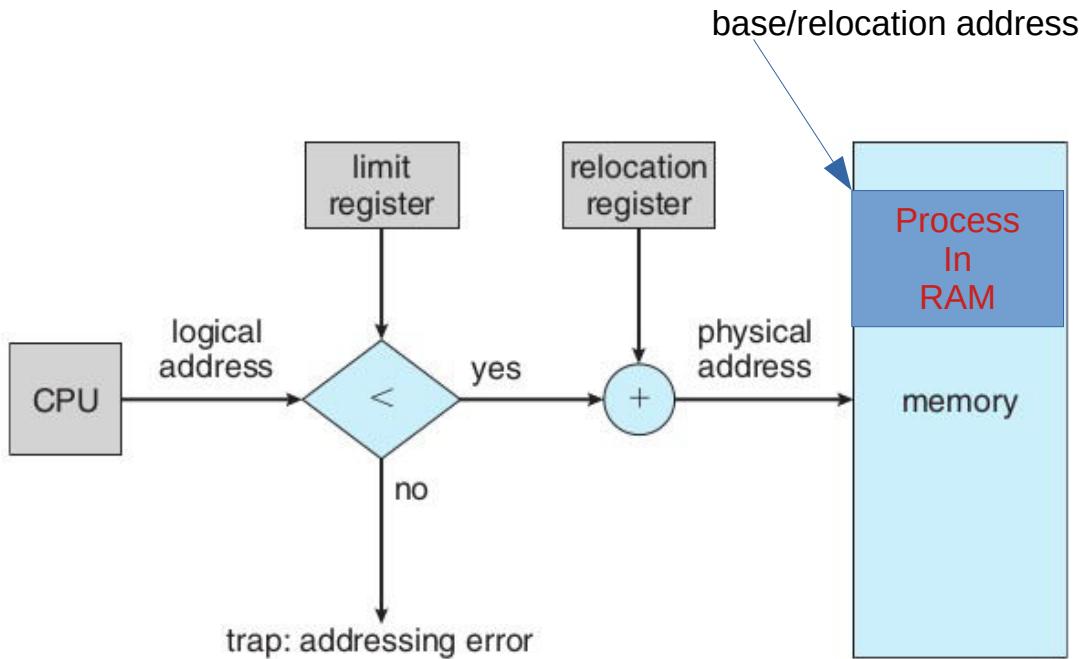
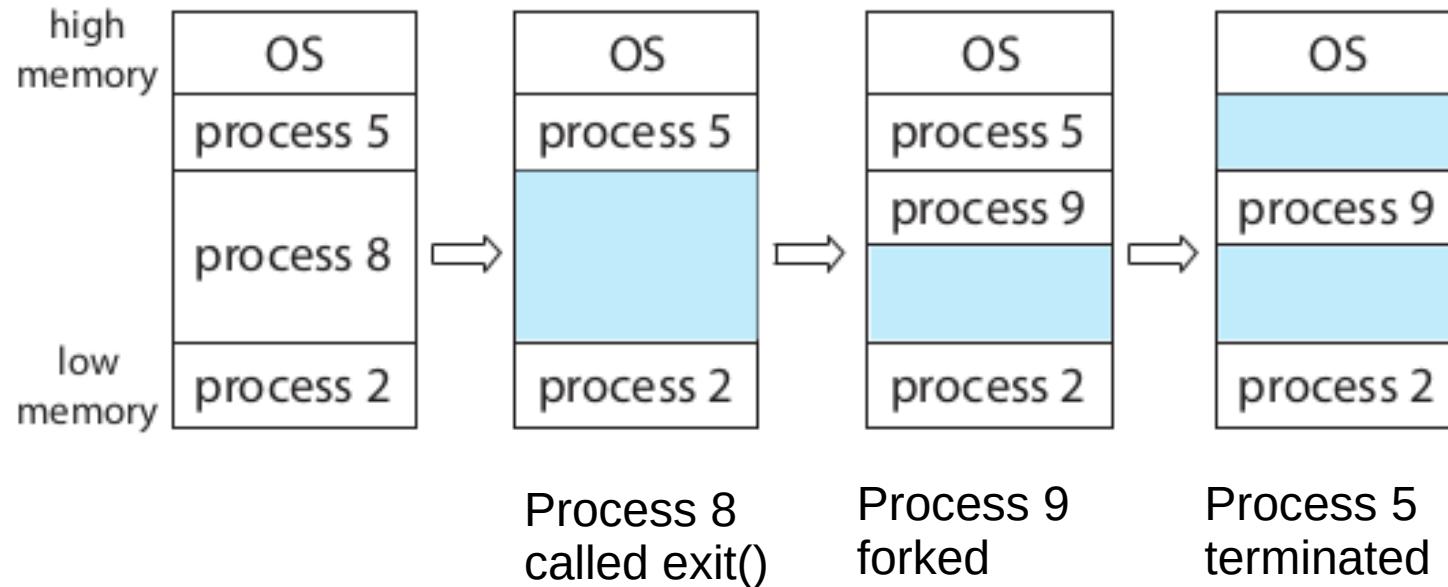


Figure 9.6 Hardware support for relocation and limit registers.

- Combined effect
  - “**Relocatable code**” – the process can go anywhere in RAM at the time of loading
  - Some memory violations can be detected – a memory access beyond base+limit will raise interrupt, thus running OS in turn, which may take action against the process

# Example scenario of memory in base+limit scheme

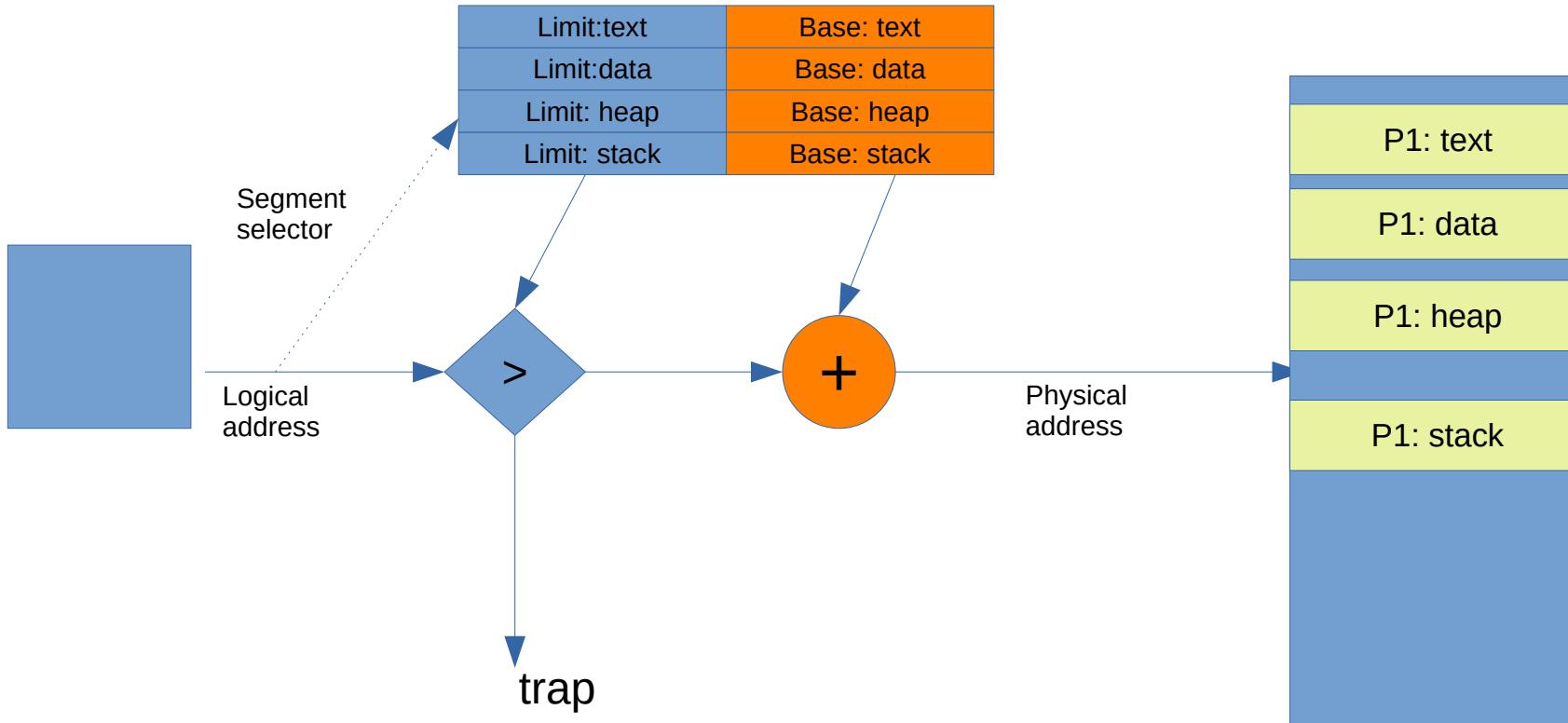


It should be possible to have relocatable code  
even with “simplest case”

By doing extra work during “loading”.

How?

# Next scheme: Multiple base +limit pairs



# Next scheme: Segmentation

## Multiple base +limit pairs

- Multiple sets of base + limit registers
- Whenever an address is issued by execution unit of CPU, it will also include reference to some base register
  - And hence limit register paired to that base register will be used for error checking
- Compiler: can assume a separate chunk of memory for code, data, stack, heap, etc. And accordingly calculate addresses . Each “segment” starting at address 0.
- OS: will load the different ‘sections’ in different memory regions and accordingly set different ‘base’ registers
-

# **Next scheme: Multiple base +limit pairs, with further indirection**

- Base + limit pairs can also be stored in some memory location (not in registers). Question: how will the cpu know where it's in memory?
  - One CPU register to point to the location of table in memory
- Segment registers still in use, they give an index in this table
- This is x86 segmentation
  - Flexibility to have lot more “base+limits” in the array/table in memory

# Next scheme: Multiple base +limit pairs, with further indirection

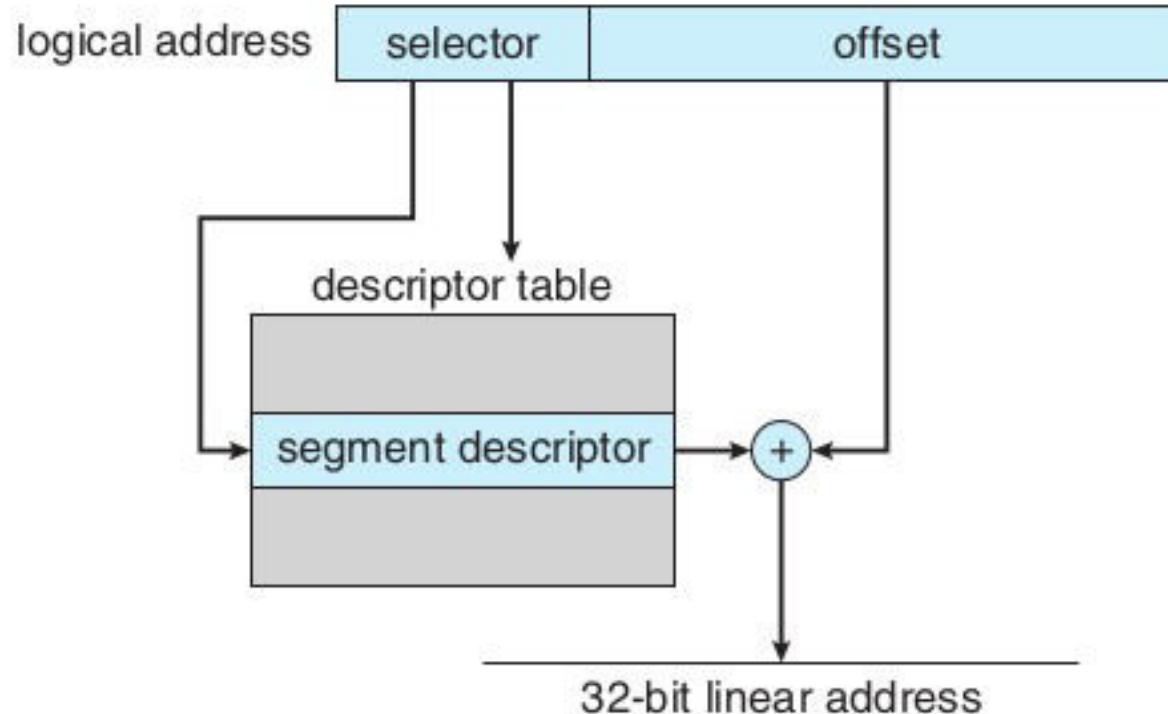


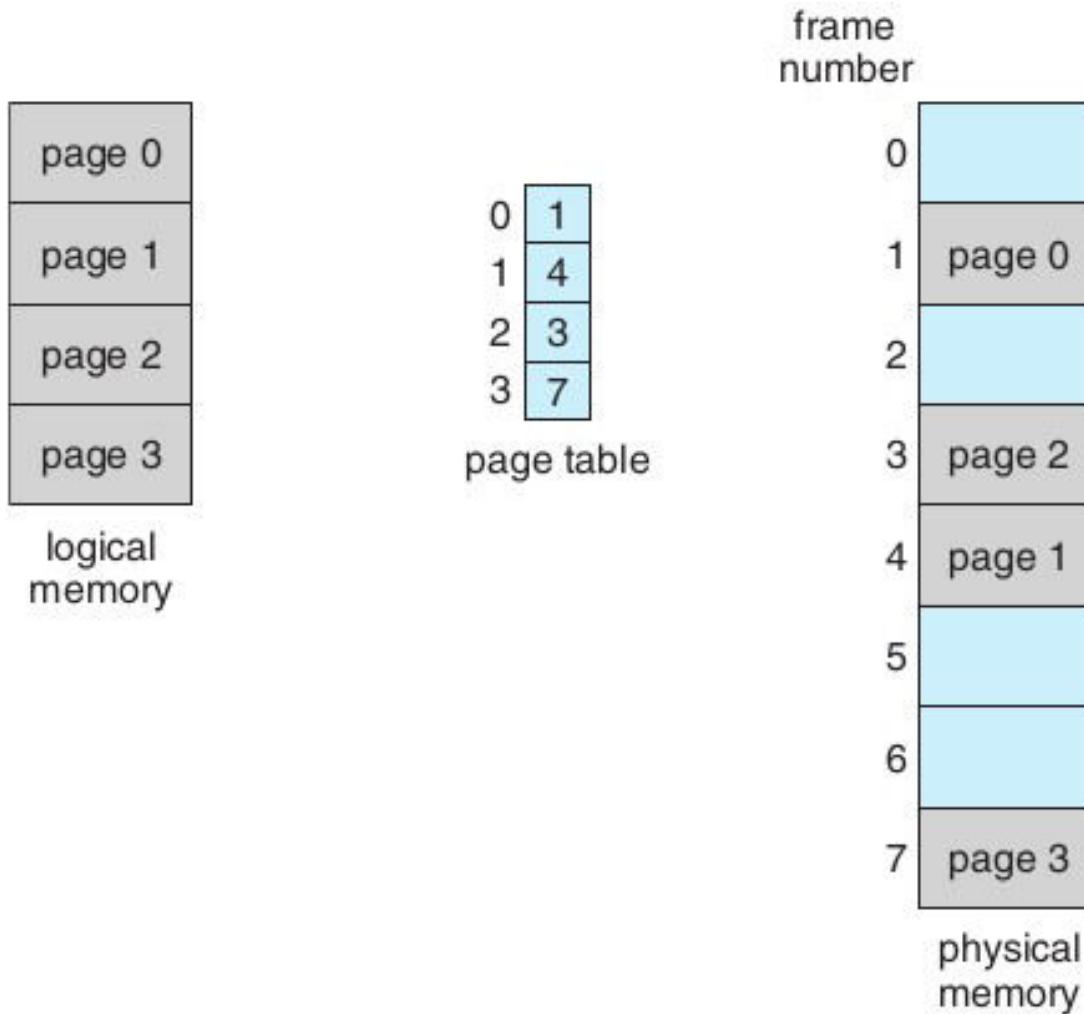
Figure 9.22 IA-32 segmentation.

# Problems with segmentation schemes

- OS needs to find a continuous free chunk of memory that fits the size of the “segment”
  - If not available, your exec() can fail due to lack of memory
- Suppose 50k is needed
  - Possible that among 3 free chunks total 100K may be available, but no single chunk of 50k!
  - **External fragmentation**
- Solution to external fragmentation: **compaction** – move the chunks around and make a continuous big chunk available. Time consuming, tricky.

# Solving external fragmentation problem

- Process should not be continuous in memory!
- Divide the continuous process image in smaller chunks (let's say 4k each) and locate the chunks anywhere in the physical memory
  - Need a way to map the *logical* memory addresses into *actual physical memory addresses*



**Figure 9.9** Paging model of logical and physical memory.

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

logical memory

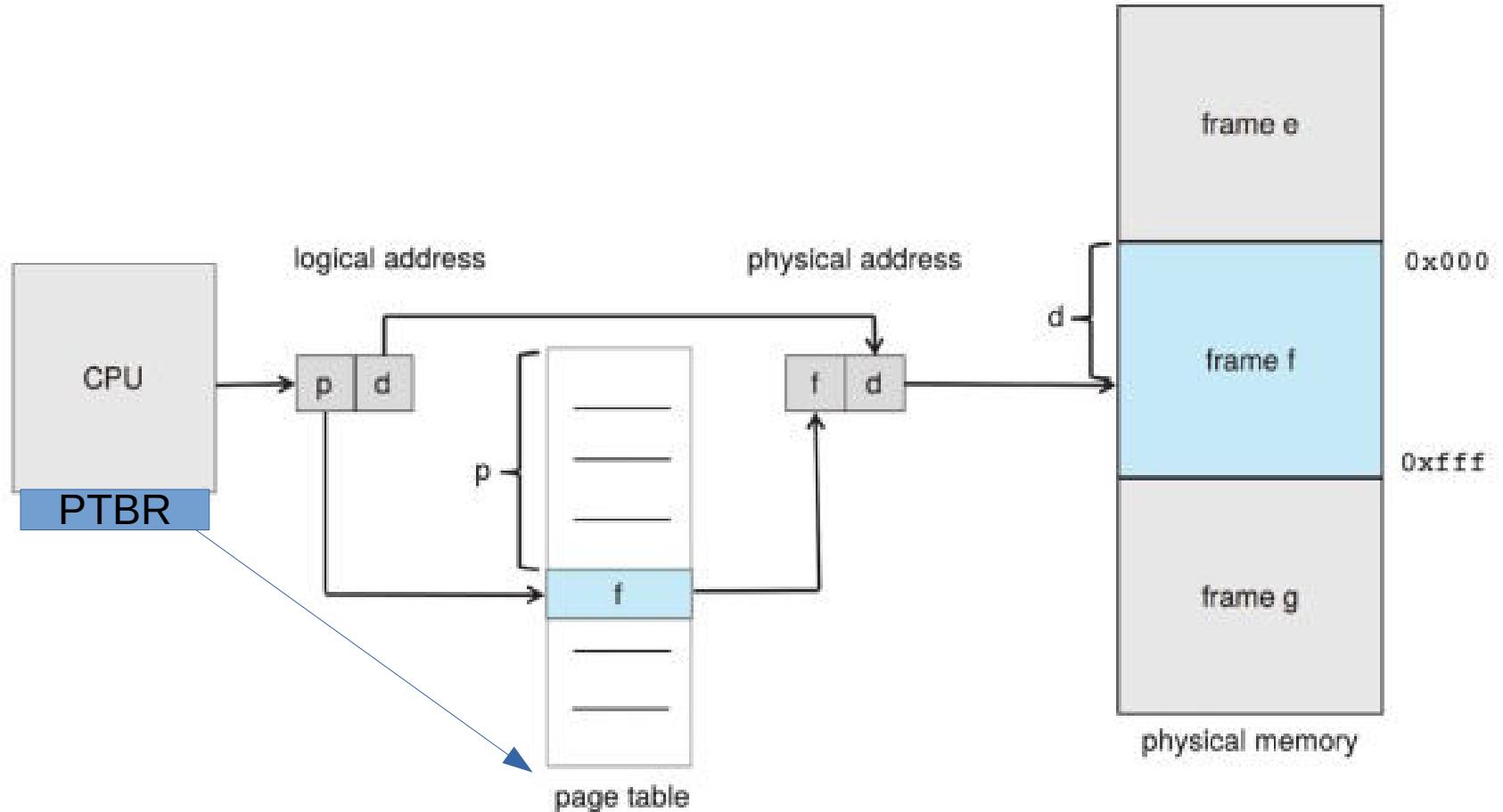
0	5
1	6
2	1
3	2

page table

0	
4	i j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	

physical memory

Figure 9.10 Paging example for a 32-byte memory with 4-byte pages.



**Figure 9.8** Paging hardware.

# Paging

- Process is assumed to be composed of equally sized “pages” (e.g. 4k page)
- Actual memory is considered to be divided into page “frames”.
- CPU generated logical address is split into a page number and offset
- A page table base register inside CPU will give location of an in memory table called page table
- Page number used as offset in a table called page table, which gives the physical page frame number
- Frame number + offset are combined to get physical memory address

# Paging

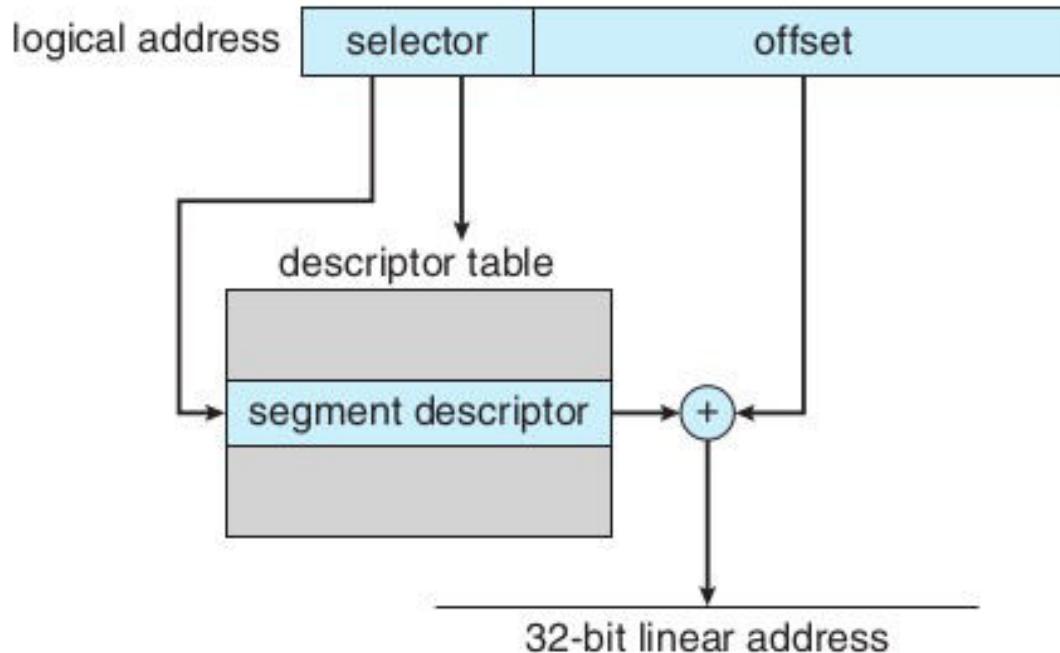
- Compiler: assume the process to be one continuous chunk of memory (!) . Generate addresses accordingly
- OS: at exec() time – allocate different frames to process, allocate a page table(!), setup the page table to map page numbers with frame numbers, setup the page table base register, start the process
- Now hardware will take care of all translations of logical addresses to physical addresses

# X86 memory management



**Figure 9.21** Logical to physical address translation in IA-32.

# Segmentation in x86



- The selector is automatically chosen using Code Segment (CS) register, or Data Segment (DS) register depending on which type of memory address is being fetched
- Descriptor table is in memory
- The location of Descriptor table (Global DT- GDT or Local DT – LDT) is given by a GDT-register i.e. GDTR or LDT-register i.e. LDTR
- 

**Figure 9.22** IA-32 segmentation.

# Paging in x86

- Depending on a flag setup in CR3 register, either 4 MB or 4 KB pages can be enabled
- Page directory, page table are both in memory

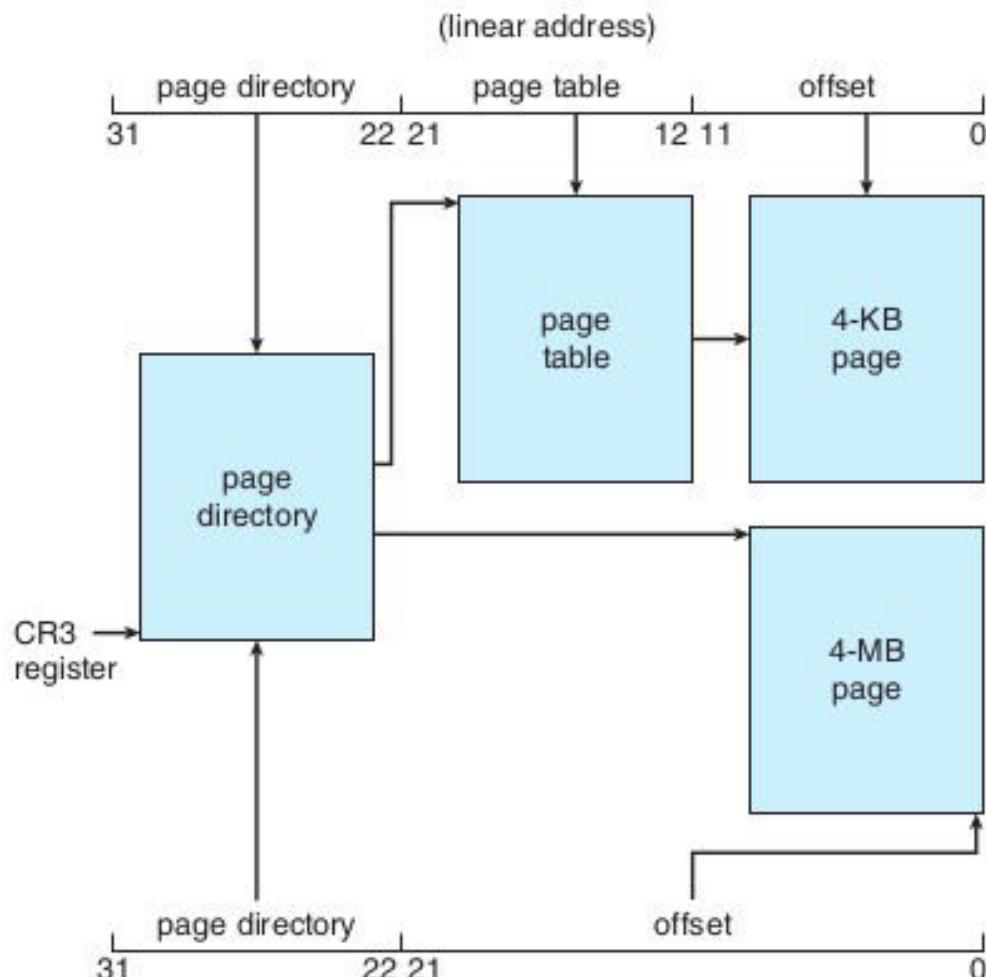


Figure 9.23 Paging in the IA-32 architecture.

# Scheduling

**Abhijit A.M.**  
**abhijit.comp@coep.ac.in**

**Credits: Slides from os-book.com**

# Necessity of scheduling

## Multiprogramming

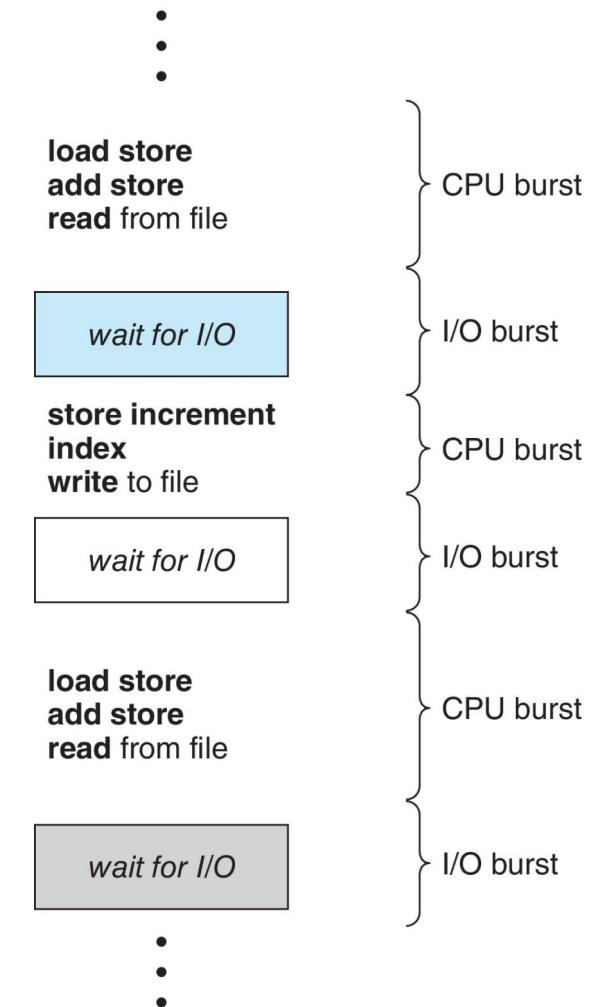
- Ability to run multiple programs at a time
- Increase use of CPU
  - CPU utilisation

# CPU Scheduling

- The task of selecting ‘next’ process/thread to execute on CPU and doing a context switch
- Scheduling algorithm
  - Criteria for selecting the ‘next’ process/thread and it’s implementation
- Why is it important?
  - Affects performance !
  - Affects end user experience !
  - Involves money!

# Observation: CPU, I/O Bursts

- Process can ‘wait’ for an event (disk I/O, read from keyboard, etc. )
- During this period another process can be scheduled
- CPU–I/O Burst Cycle:
  - Process execution consists of a **cycle** of CPU execution and I/O wait
  - **CPU burst followed by I/O burst**
  - **CPU burst distribution is of main concern**



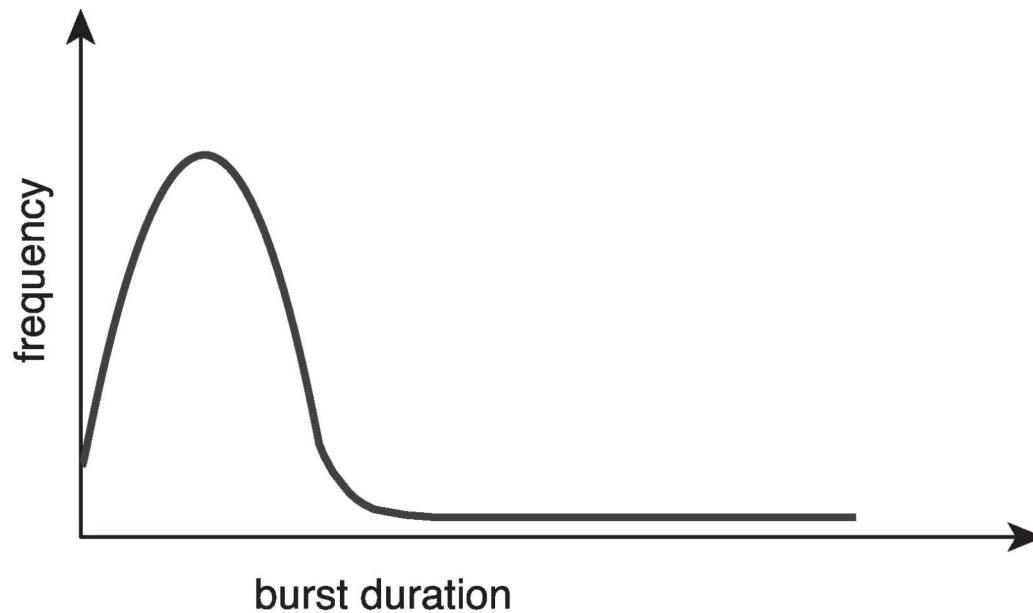
# Let's understand the problem

- Programs have alternate CPU and I/O bursts
  - Some are CPU intensive
  - Some are I/O intensive
  - Some are mix of both

A C code example:

```
f(int i, int j, int k) {  
    j = k * i; // CPU burst  
    scanf("%d", &i); // I/O burst  
    k = i * j;// CPU burst  
    printf("%d\n", k);// I/O burst  
    return k;  
}
```

# CPU bursts: observation



Large number of short bursts

Small number of longer  
bursts

# Scheduler, what does it do?

- **From a list of processes, ready to run**
  - Selects a process for execution
  - Allocates a CPU to the process for execution
  - Does “context switch”

# Context and Context Switch

- **Context**
  - Set of registers. Which ones?
  - All or some ?
  - Following registers on xv6 kernel: edi, esi, ebx, ebp, eip
    - Related to calling convention!
  - Linux kernel context: Much bigger!
  - Also includes: MMU setup
- **Context switch**
  - Process context -> kernel context
    - On interrupt, system call, or exception
  - Kernel context -> process context
    - Returning from system call, returning from interrupt handler, scheduling a process
  - In every switch, need to change to the set of registers “last in use” by that context, and also the MMU setup

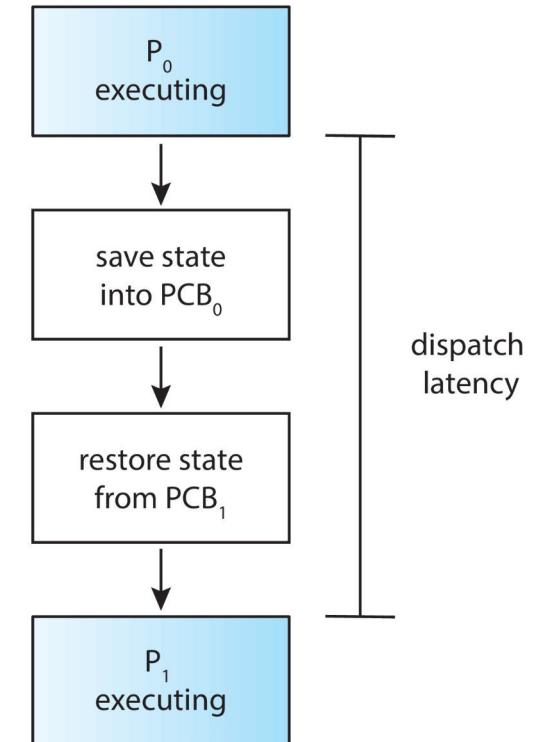
# When is scheduler invoked?

- 1) **Process Switches from running to waiting state**
  - Waiting for I/O, etc.
- 2) **Switches from running to ready state**
  - E.g. on a timer interrupt
- 3) **Switches from waiting to ready**
  - I/O wait completed, now ready to run
- 4) **Terminates**

- **Scheduling under 1 and 4 is nonpreemptive**
- **All other scheduling is preemptive**
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities

# Dispatcher: A part of scheduler

- Gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- Dispatch latency
  - time taken to stop one process and start another running
- Xv6: swtch(), some tail end parts of sched(), trap(), trapret()



# Dispatcher in action on Linux

- Run “**vmstat 1 3**”
  - Means run vmstat 3 times at 1 second delay
- In output, look at **CPU:cs**
  - Context switches every second
- Also for a process with pid 3323
  - Run “**cat /proc/3323/status**”
  - See **voluntary\_ctxt\_switches**
    - > Process left CPU
  - **nonvoluntary\_ctxt\_switche**
    - > Process was preempted

# Scheduling criteria

- **CPU utilization: Maximise**
  - keep the CPU as busy as possible. Linux: idle task is scheduled when no process to be scheduled.
- **Throughput : Maximise**
  - # of processes that complete their execution per time unit
- **Turnaround time : Minimise**
  - amount of time to execute a particular process
- **Waiting time : Minimise**
  - amount of time a process has been waiting in the ready queue
- **Response time : Minimise**
  - amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)
-

# To be studied later

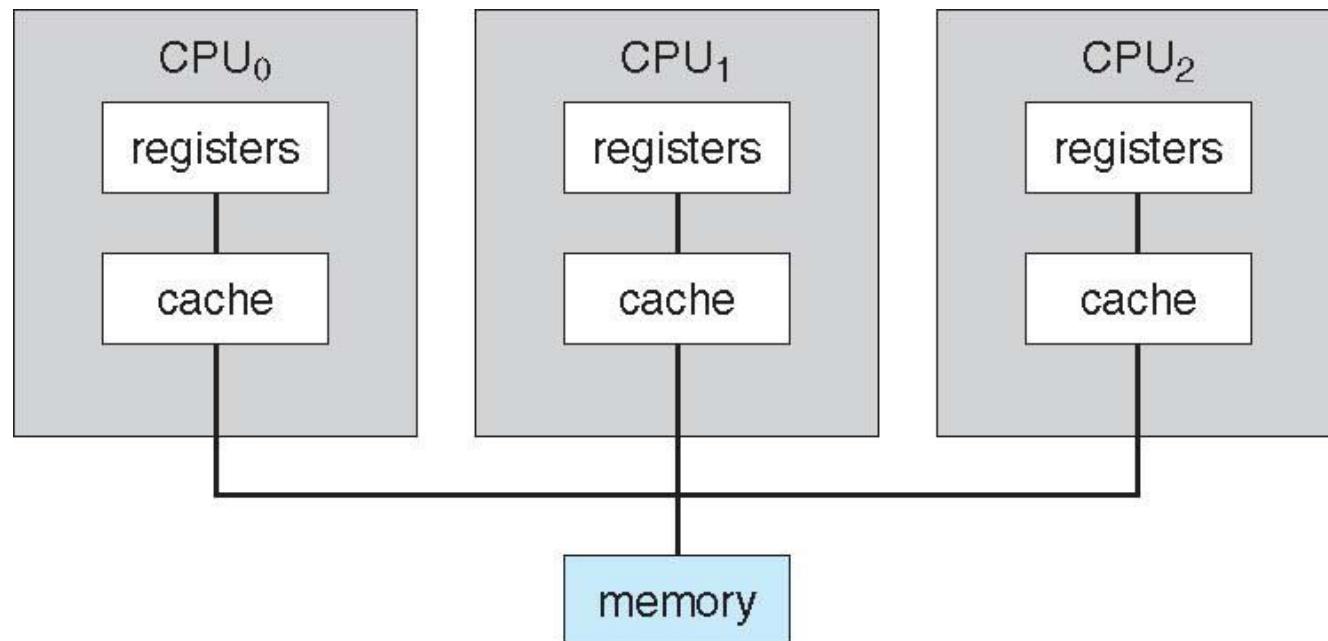
- **Evaluation of scheduling criteria**
- **Different scheduling algorithms, and their characteristics**
  - Round Robin, FIFO, Shortest Job First, Priority, Multi-level Queue, etc.
- **Thread scheduling**

# **Multi Processor Scheduling**

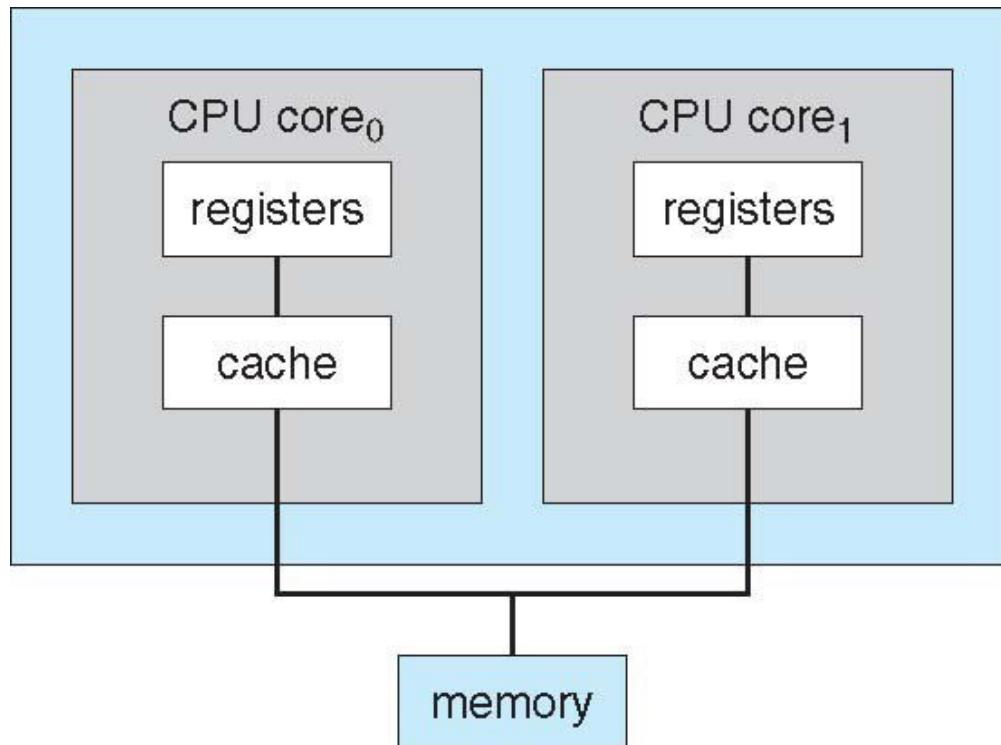
# Multiprocessor systems

- **Each processor has separate set of registers**
  - All: eip, esp, cr3, eax, ebx, etc.
- **Each processor runs independently of others**
- **Main difference is in how do they access memory**

# Symmetric multiprocessing (SMP)



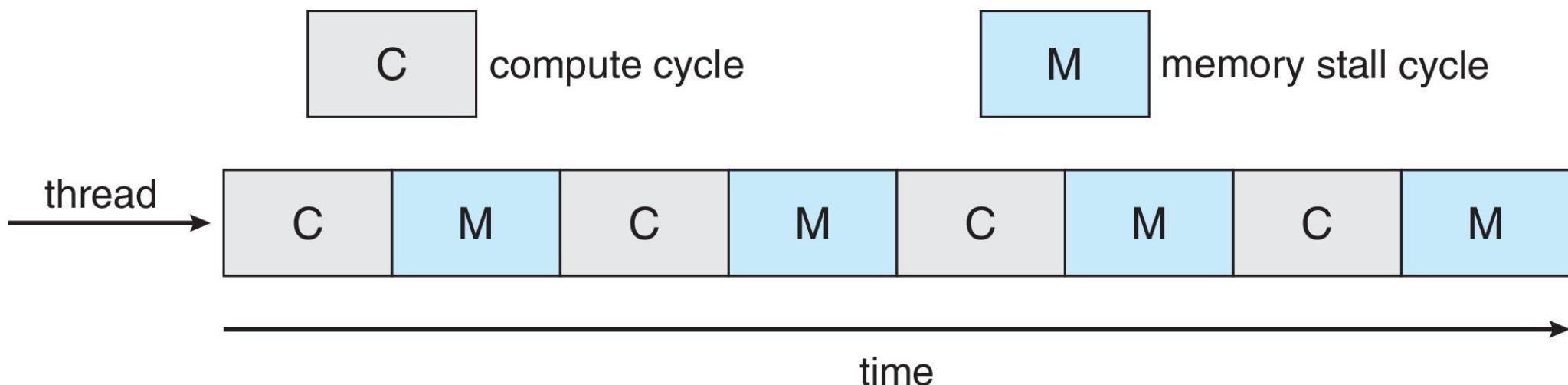
# Multicore systems (also SMP)



No difference from the perspective of OS. The hardware ensures that OS sees multiple processors and not multiple cores.

# Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens



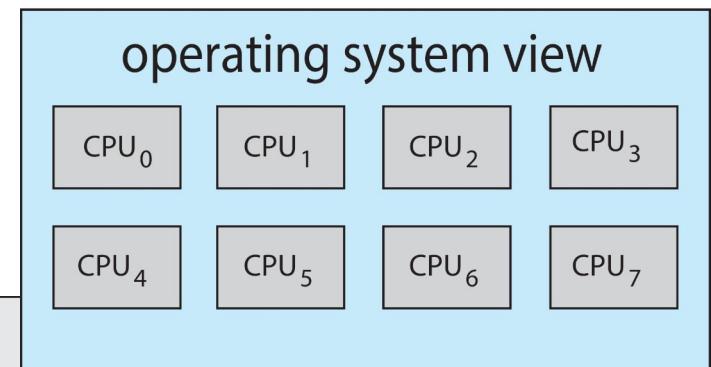
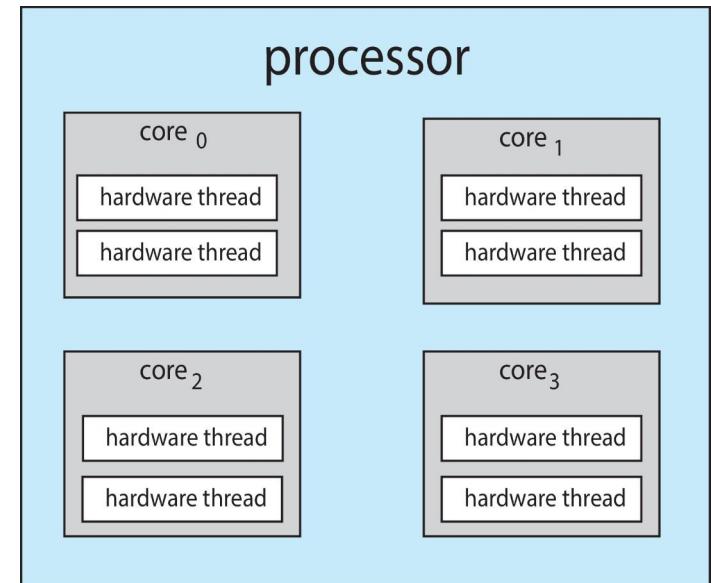
# Multithreaded Multicore System

Each core has > 1 hardware threads.

**Chip-multithreading** (CMT) assigns each core multiple hardware threads.  
(Intel refers to this as **hyperthreading**.)

On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.

If one thread has a memory stall, switch to another thread!



# More on SMP systems

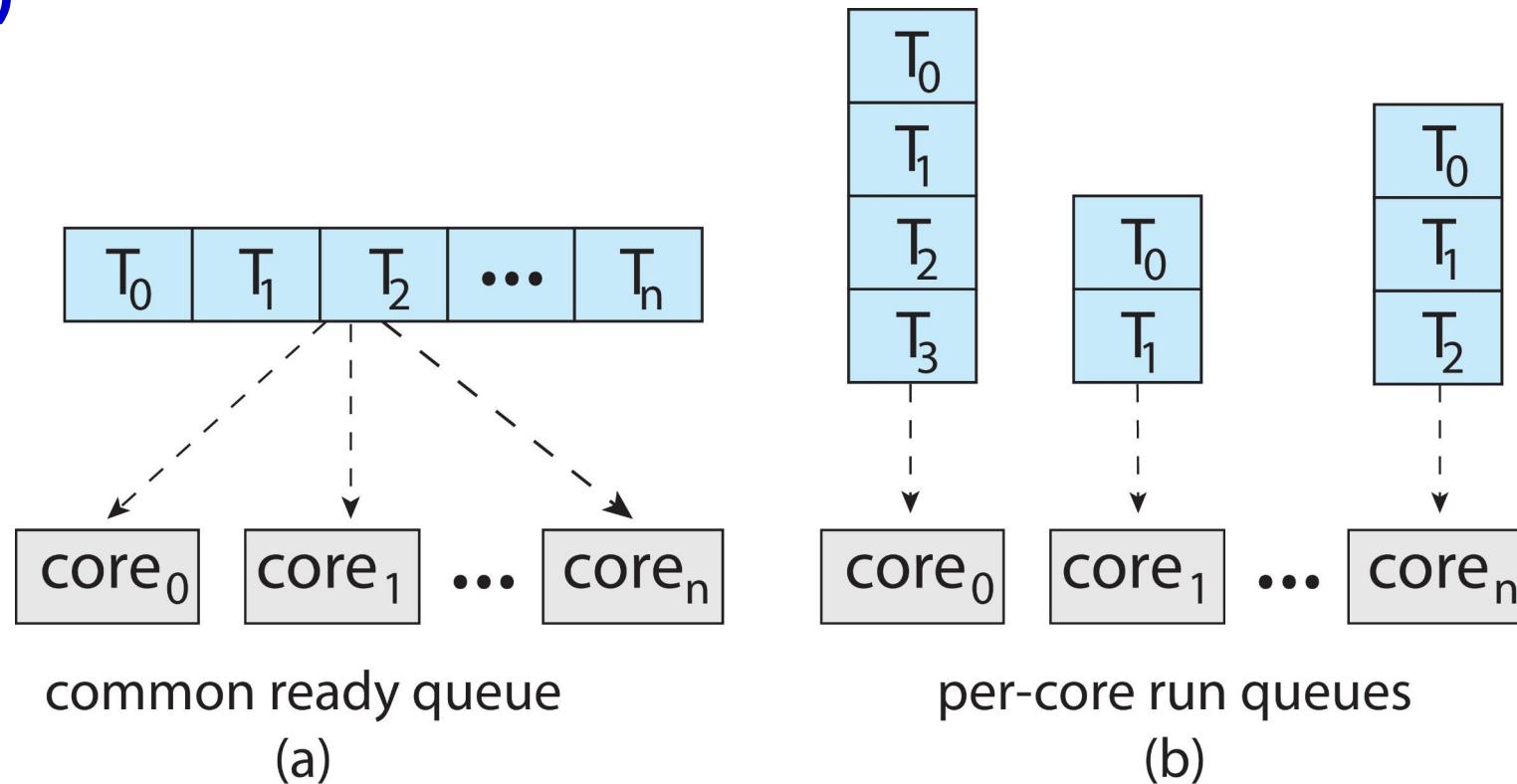
- **During booting – each CPU needs to be turned on**
  - Special I/O instructions writing to particular ports
  - See `lalicstartap()` in xv6
  - Need to setup CR3 on each processor
  - Segmentation, Page tables are shared (same memory for all CPUs)
- **All processors will keep running independently of each other**
- **Different interrupts on each processor – each needs IDT setup**
- **Each processor will be running processes, interrupt handlers, syscalls**
- **Synchronization problems !**
- **How to do scheduling ?**

# Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- Multiprocess may be any one of the following architectures:
  - Multicore CPUs
  - Multithreaded cores
  - NUMA systems
  - Heterogeneous multiprocessing

# Multiple-Processor Scheduling

- **Symmetric multiprocessing (SMP) is where each processor is self scheduling.**
- **All threads may be in a common ready queue (a)**
- **Each processor may have its own private queue of threads (b)**



# SMP in xv6

- Only one process queue
- No load balancing, no affinity (more later)
- A process may run any CPU-burst /allotted-time-quantum on any processor randomly

**End**

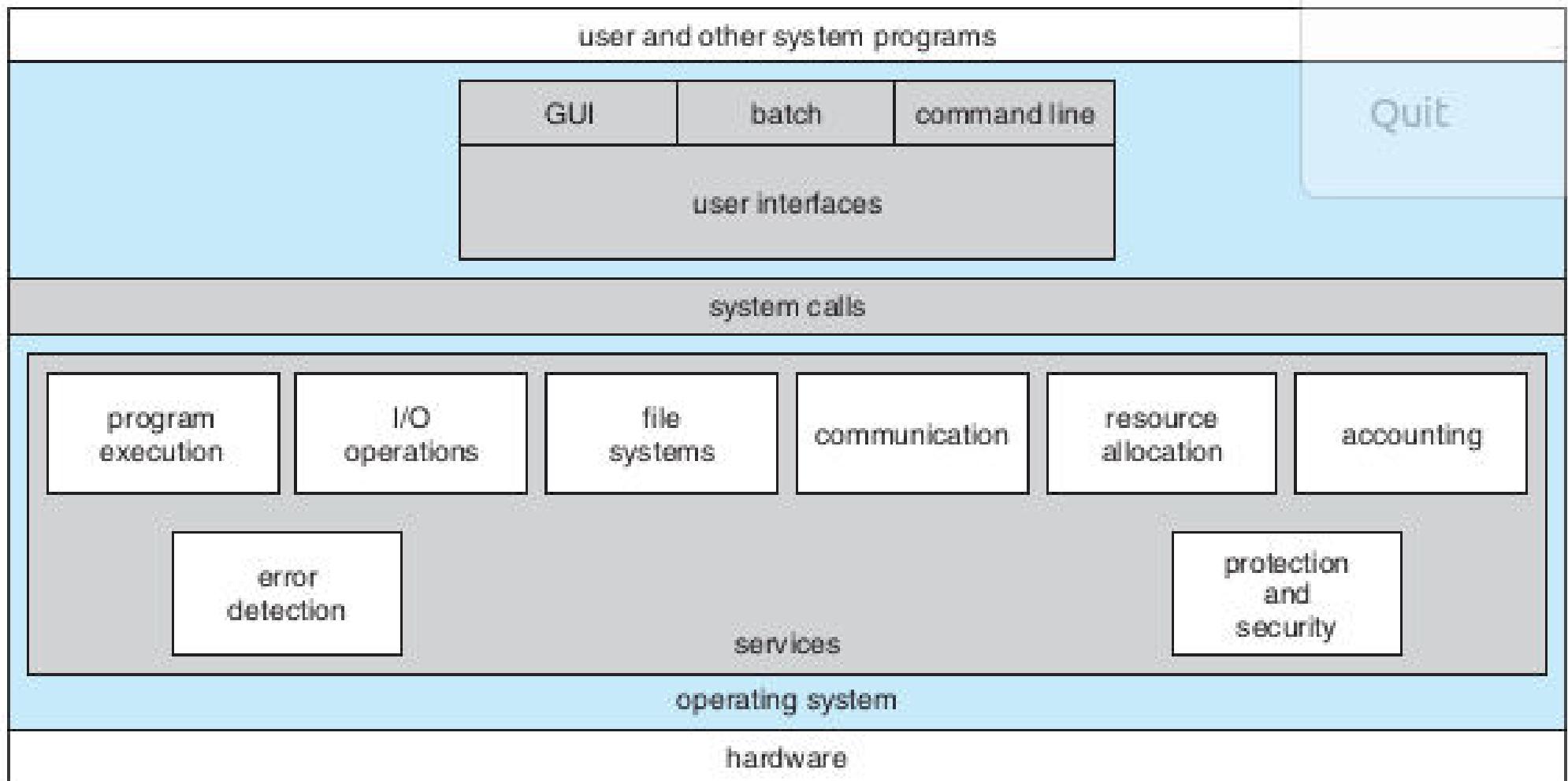
# **System Calls, fork(), exec()**

Abhijit A. M.  
[abhijit.comp@coep.ac.in](mailto:abhijit.comp@coep.ac.in)

(C) Abhijit A.M.

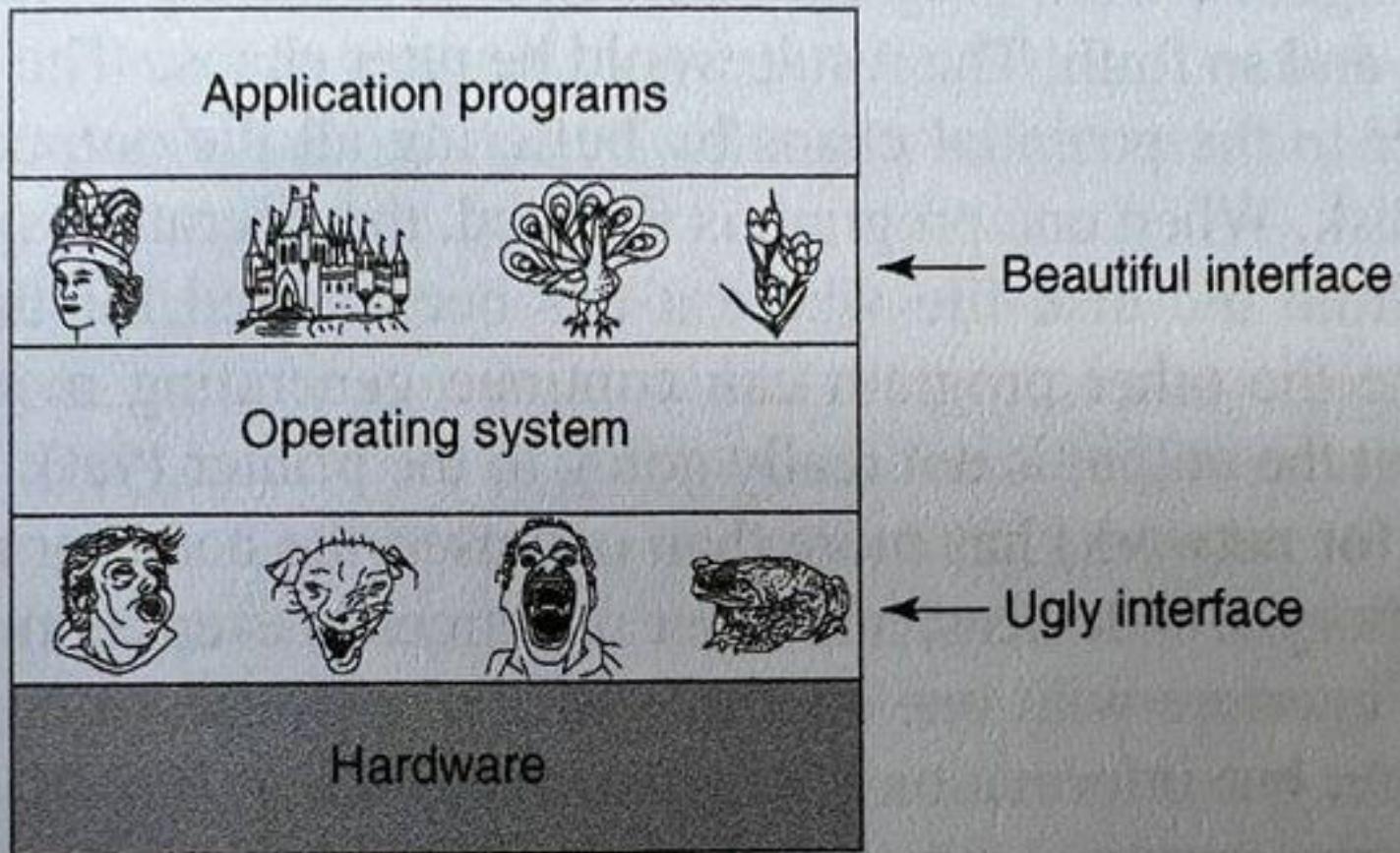
Available under Creative Commons Attribution-ShareAlike License V3.0+

Credits: Slides of “OS Book” ed10.



**Figure 2.1** A view of operating system services.

## WHAT IS AN OPERATING SYSTEM?



**Figure 1-2.** Operating systems turn ugly hardware into beautiful abstractions.

# System Calls

- **Services provided by operating system to applications**
  - Essentially available to applications by calling the particular software interrupt application
    - All system calls essentially involve the “INT 0x80” on x86 processors + Linux
    - Different arguments specified in EAX register inform the kernel about different system calls
- **The C library has wrapper functions for each of the system calls**
  - E.g. open(), read(), write(), fork(), mmap(), etc.

# Types of System Calls

- **File System Related**
  - Open(), read(), write(), close(), etc.
- **Processes Related**
  - Fork(), exec(), ...
- **Memory management related**
  - Mmap(), shm\_open(), ...
- **Device Management**
- **Information maintainance – time,date**
- **Communication between processes (IPC)**
- **Read man syscalls**

[https://linuxhint.com/list\\_of\\_linux\\_syscalls/](https://linuxhint.com/list_of_linux_syscalls/)

## EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix
<b>Process Control</b>	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
<b>File Manipulation</b>	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
<b>Device Manipulation</b>	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
<b>Information Maintenance</b>	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
<b>Communication</b>	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
<b>Protection</b>	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

```
int main() {
    int a = 2;
    printf("hi\n");
}

-----
C Library
-----

int printf("void *a, ...) {
    ...
    write(1, a, ...);
}

int write(int fd, char *, int len) {
    int ret;
    ...
    mov $5, %eax,
    mov ... %ebx,
    mov ..., %ecx
    int $0x80
    __asm__("movl %eax, -4(%ebp)");
    # -4ebp is ret
    return ret;
}
```

# Code schematic

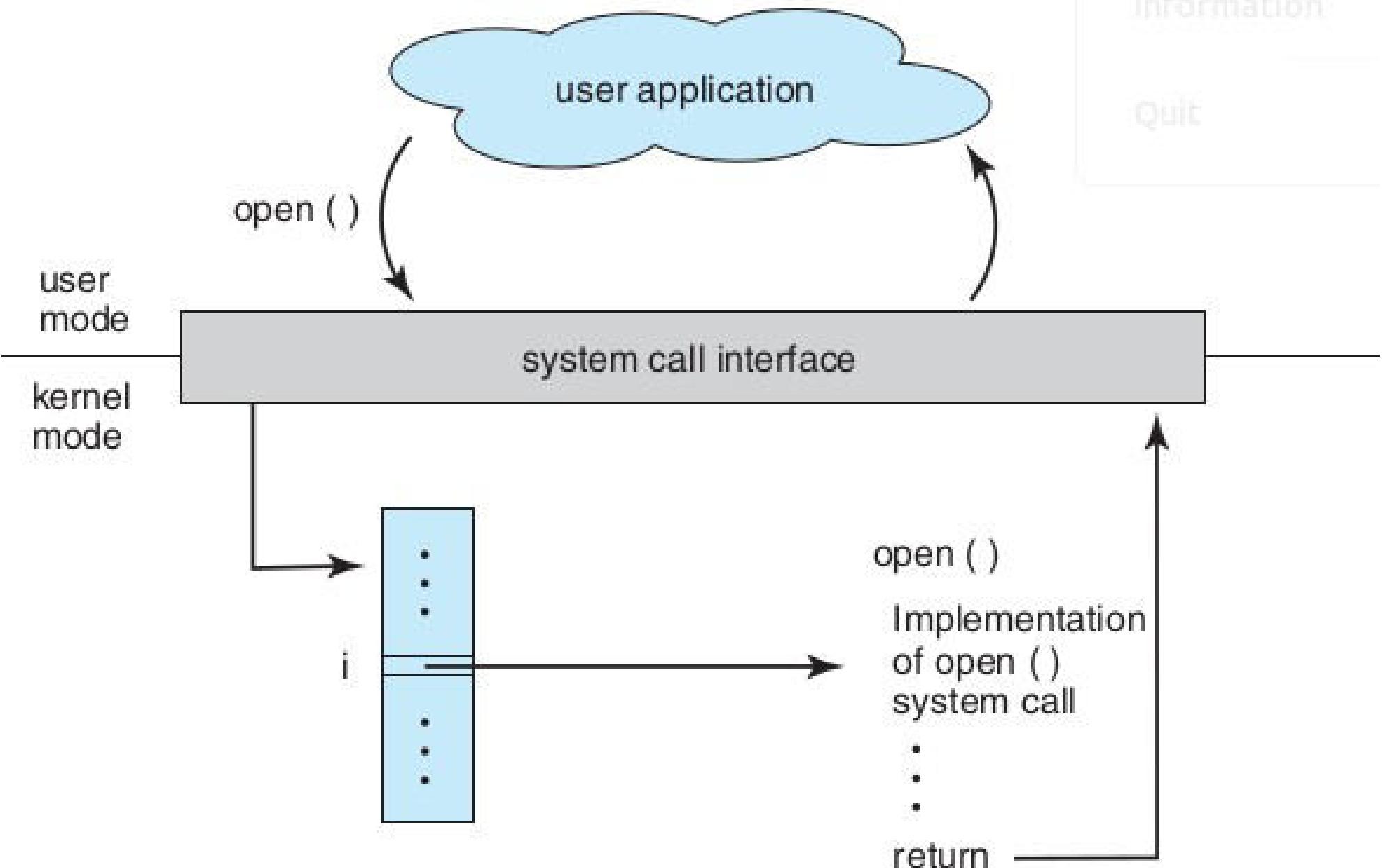
-----user-kernel-mode-boundary-----

//OS code

int sys\_write(int fd, char \*, int len) {

figure out location on disk  
where to do the write and  
carry out the operation,  
etc.

}



**Figure 2.6** The handling of a user application invoking the `open()` system call.

Two important system calls  
Related to processes

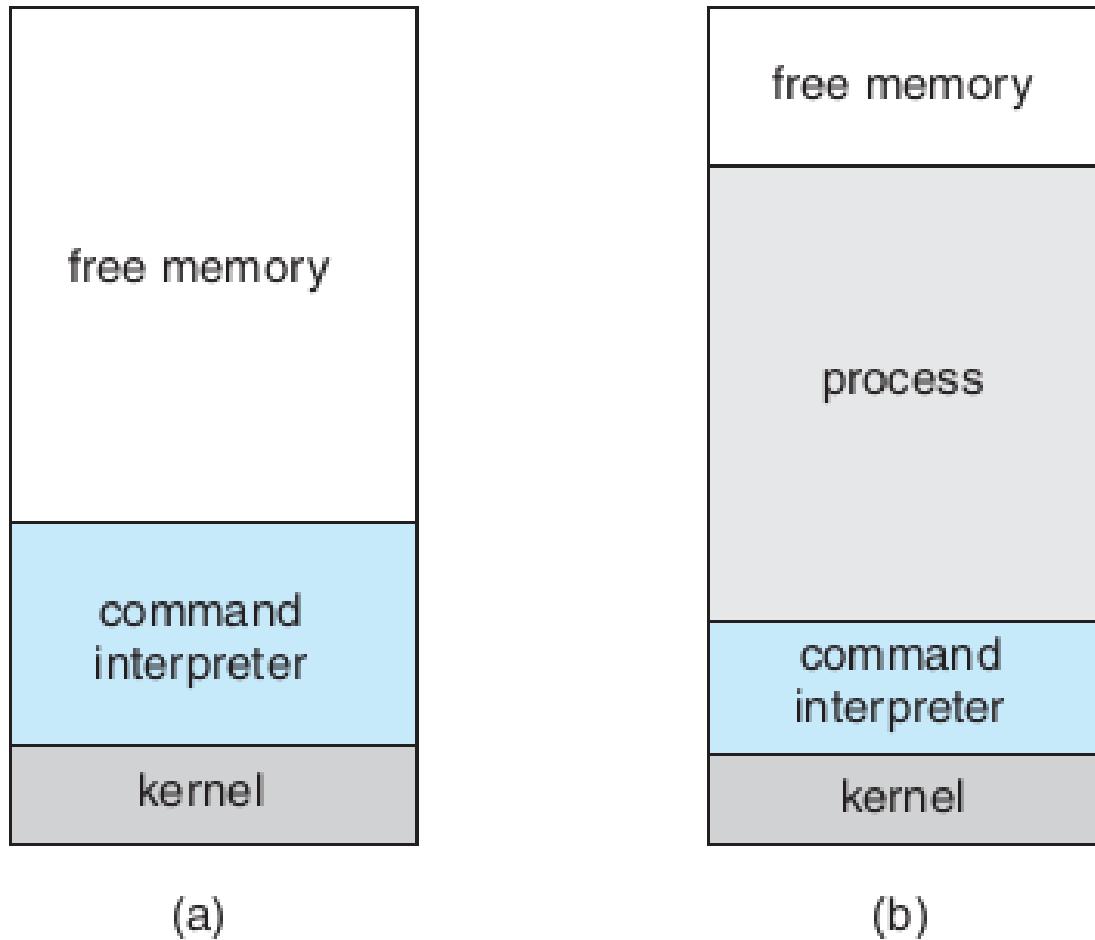
**fork() and exec()**

# Process

- **A program in execution**
- **Exists in RAM**
- **Scheduled by OS**
  - In a timesharing system, intermittantly scheduled by allocating a time quantum, e.g. 20 microseconds
- **The “ps” command on Linux**

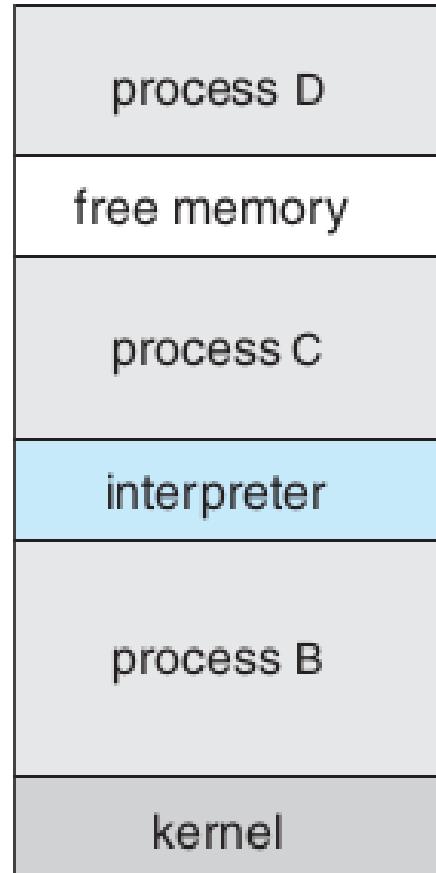
# Process in RAM

- **Memory is required to store the following components of a process**
  - **Code**
  - **Global variables (data)**
  - **Stack (stores local variables of functions)**
  - **Heap (stores malloced memory)**
  - **Shared libraries (e.g. code of printf, etc)**
  - **Few other things, may be**



**Figure 2.9** MS-DOS execution. (a) At system startup. (b) Running a program.

MS-DOS: a single tasking operating system  
Only one program in RAM at a time, and only one program can run at a time



A multi tasking system  
With multiple programs loaded in memory  
Along with kernel

(A very simplified conceptual diagram. Things are more complex in reality)

# **fork()**

- **A running process creates it's duplicate!**
- **After call to fork() is over**
  - Two processes are running
  - Identical
  - The calling function returns in two places!
  - Caller is called parent, and the new process is called child
  - PID is returned to parent and 0 to child

# **exec()**

- **Variants: execvp(), execl(), etc.**
- **Takes the path name of an executable as an argument**
- **Overwrites the existing process using the code provided in the executable**
- **The original process is OVER ! Vanished!**
- **The new program starts running overwritting the existing process!**

# Shell using fork and exec

- Demo
- The only way a process can be created on Unix/Linux is using `fork() + exec()`
- All processes that you see were started by some other process using `fork() + exec()` , except the initial “*init*” process
- *When you click on “firefox” icon, the user-interface program does a `fork() + exec()` to start firefox; same with a command line shell program*
- The “bash” shell you have been using is nothing but an advanced version of the shell code shown during the demo
- See the process tree starting from “*init*”
- Your next assignment

# The boot process, once again

- **BIOS**
- **Boot loader**
- **OS – kernel**
- **Init created by kernel by Hand(kernel mode)**
- **Kernel schedules init (the only process)**
- **Init fork-execs some programs (user mode)**
  - Now these programs will be scheduled by OS
- **Init -> GUI -> terminal -> shell**
  - One of the typical parent-child relationships

# **Event Driven kernel**

# **Multi-tasking, Multi-programming**

# Earlier...

- **Boot process**
  - BIOS -> Boot Loader -> OS -> “init” process
- **CPU doing**

```
for(;;) {  
    fetch from @PC;  
    deode+execute;  
    PC++/change PC  
}
```

# **Understanding hardware interrupts**

- **Hardware devices (keyboard, mouse, hard disk, etc) can raise “hardware interrupts”**
- **Basically create an electrical signal on some connection to CPU (/bus)**
- **This is notified to CPU (in hardware)**
- **Now CPU’s normal execution is interrupted!**
  - What’s the normal execution?
  - CPU will not continue doing the fetch, decode, execute, change PC cycle !
  - What happens then?

# Understanding hardware interrupts

- **On Hardware interrupt**
  - The PC changes to a location pre-determined by CPU manufacturers!
  - Now CPU resumes normal execution
    - What's normal?
    - Same thing: Fetch, Decode, Execute, Change PC, repeat!
    - But...
  - But what's there at this pre-determined address?
  - OS! How's that ?

# Boot Process

- **BIOS runs “automatically” because at the initial value of PC (on power ON), the manufacturers have put in the BIOS code in ROM**
  - CPU is running BIOS code . BIOS code also occupies all the addresses corresponding to hardware interrupts.
- **BIOS looks up boot loader (on default boot device) and loads it in RAM, and passes control over to it**
  - Pass control? - Like a JMP instruction
  - CPU is running boot loader code
- **Boot loader gets boot option from human user, and loads the selected OS in memory and passes control over to OS**
  - Now OS is running on the processor !

# Hardware interrupts and OS

- **When OS starts running initially**
  - It copies relevant parts of it's own code at all possible memory addresses that a hardware interrupt can lead to!
  - Intelligent, isn't' it?
- **Now what?**
  - Whenever there is a hardware interrupt – what will happen?
  - The PC will change to predetermined location, and control will jump into OS code
- **So remember: whenever there is a hardware interrupt, OS code will run!**
- **This is “taking control of hardware”**

# Key points

- **Understand the interplay of hardware features + OS code + clever combination of the two to achieve OS control over hardware**
- **Most features of computer systems / operating systems are derived from hardware features**
  - We will keep learning this throughout the course
  - Hardware support is needed for many OS features

# Time Shared CPU

- **Normal use: after the OS has been loaded and Desktop environment is running**
- **The OS and different application programs keep executing on the CPU alternatively (more about this later)**
  - **The CPU is time-shared between different applications and OS itself**
- **Questions to be answered later**
  - **How is this done?**
  - **How does OS control the time allocation to each?**
  - **How does OS choose which application program to run next?**
  - **Where do the application programs come from? How do they start running ?**
  - **Etc.**

# Multiprogramming

- **Program**
  - Just a binary (machine code) file lying on the **hard drive**. E.g. `/bin/ls`
  - Does not do anything!
- **Process**
  - A program that is executing
  - Must exist in **RAM** before it executes. **Why?**
  - One program can run as multiple processes. What does that mean?

# Multiprogramming

- **Multiprogramming**
  - A system where multiple processes(!) exist at the same time in the RAM
  - But only one runs at a time!
    - Because there is only one CPU
- **Multi tasking**
  - Time sharing between multiple processes in a multi-programming system
  - The OS enables this. How? To be seen later.

# Question

- **Select the correct one**
  - 1) A multiprogramming system is not necessarily multitasking
  - 2) A multitasking system is not necessarily multiprogramming

# Events , that interrupt CPU's functioning

- Three types of “traps” : Events that make the CPU run code at a pre-defined address
  - 1) Hardware interrupts
  - 2) Software interrupt instructions (trap)  
E.g. instruction “int”
  - 3) Exceptions
    - e.g. a machine instruction that does division by zero
    - Illegal instruction, etc.
    - Some are called “faults”, e.g. “page fault”, recoverable
    - some are called “aborts”, e.g. division by zero, non-recoverable
- The OS code occupies all memory locations corresponding to the PC values related to all the above events!

# Multi tasking requirements

- **Two processes should not be**
  - Able to steal time of each other
  - See data/code of each other
  - Modify data/code of each other
  - Etc.
- **The OS ensures all these things. How?**
  - To be seen later.

**But the OS is “always” “running”  
“in the background”  
Isn’t it?**

**Absolutely No!**

**Let's understand  
What kind of  
Hardware, OS interplay  
makes  
Multitasking possible**

# Two types of CPU instructions and two modes of CPU operation

- CPU instructions can be divided into two types
- Normal instructions
  - mov, jmp, add, etc.
- Privileged instructions
  - Normally related to hardware devices
  - E.g.
    - IN, OUT # write to I/O memory locations
    - INTR # software interrupt, etc.

# **Two types of CPU instructions and two modes of CPU operation**

- **CPUs have a mode bit (can be 0 or 1)**
- **The two values of the mode bit are called: User mode and Kernel mode**
- **If the bit is in user mode**
  - Only the normal instructions can be executed by CPU
- **If the bit is in kernel mode**
  - Both the normal and privileges instructions can be executed by CPU
- **If the CPU is “made” to execute privileged instruction when the mode bit is in “User mode”**
  - It results in a illegal instruction execution
  - Again running OS code!

# **Two types of CPU instructions and two modes of CPU operation**

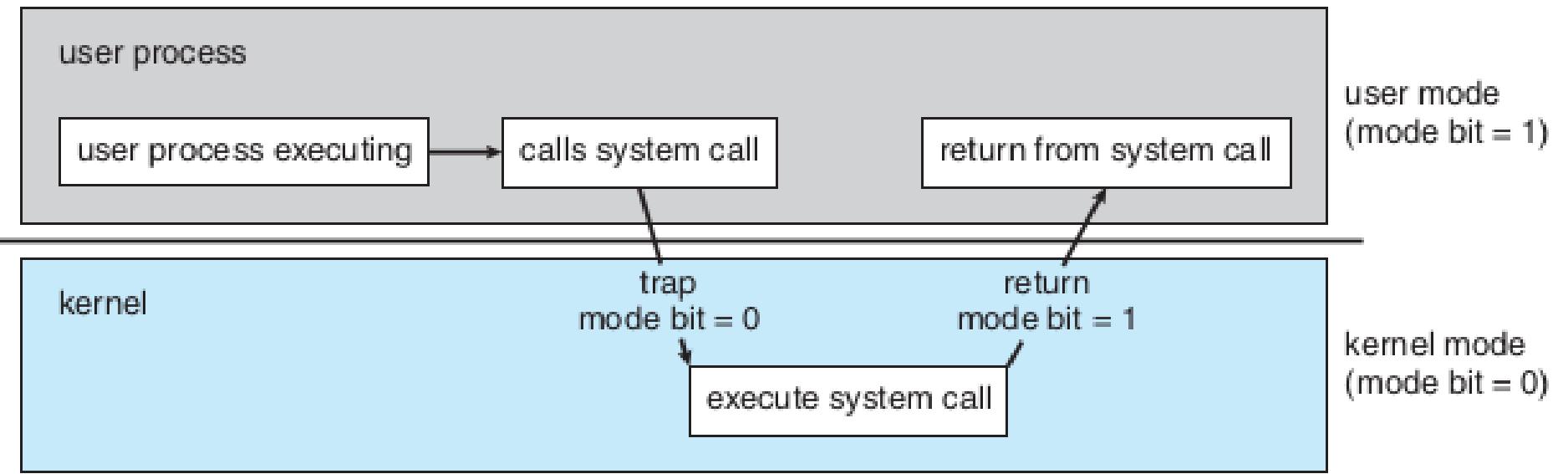
- **The operating system code runs in kernel mode.**
  - How? Wait for that!
- **The application code runs in user mode**
  - How? Wait !
  - So application code can not run privileged hardware instructions
- **Transition from user mode to kernel mode and vice-versa**
  - Special instruction called “software interrupt” instructions
  - E.g. INT instruction on x86

# Software interrupt instruction

- E.g. INT on x86 processors
- Does two things at the same time!
  - Changes mode from user mode to kernel mode in CPU
    - + Jumps to a pre-defined location! Basically changes PC to a pre-defined value.
      - Close to the way a hardware interrupt works. Isn't it?
  - Why two things together?
  - What's there are the pre-defined location?
    - Obviously, OS code. OS occupied these locations in Memory, at Boot time.

# Software interrupt instruction

- **What's the use of these type of instructions?**
  - An application code running INT 0x80 on x86 will now cause
    - Change of mode
    - Jump into OS code
  - Effectively a request by application code to OS to do a particular task!
  - E.g. read from keyboard or write to screen !
  - OS providing hardware services to applications !
  - A proper argument to INT 0x80 specified in a proper register indicates different possible request



**Figure 1.10** Transition from user to kernel mode.

# Software interrupt instruction

- **How does application code run INT instruction?**
  - C library functions like printf(), scanf() which do I/O requests contain the INT instruction!
  - Control flow
    - Application code -> printf -> INT -> OS code -> back to printf code -> back to application code

# Example: C program

```
int main() {  
    int i, j, k;  
    k = 20;  
    scanf("%d", &i); // This jumps into OS and returns back  
    j = k + i;  
    printf("%d\n", j); // This jumps into OS and returns back  
    return 0;  
}
```

# **Interrupt driven OS code**

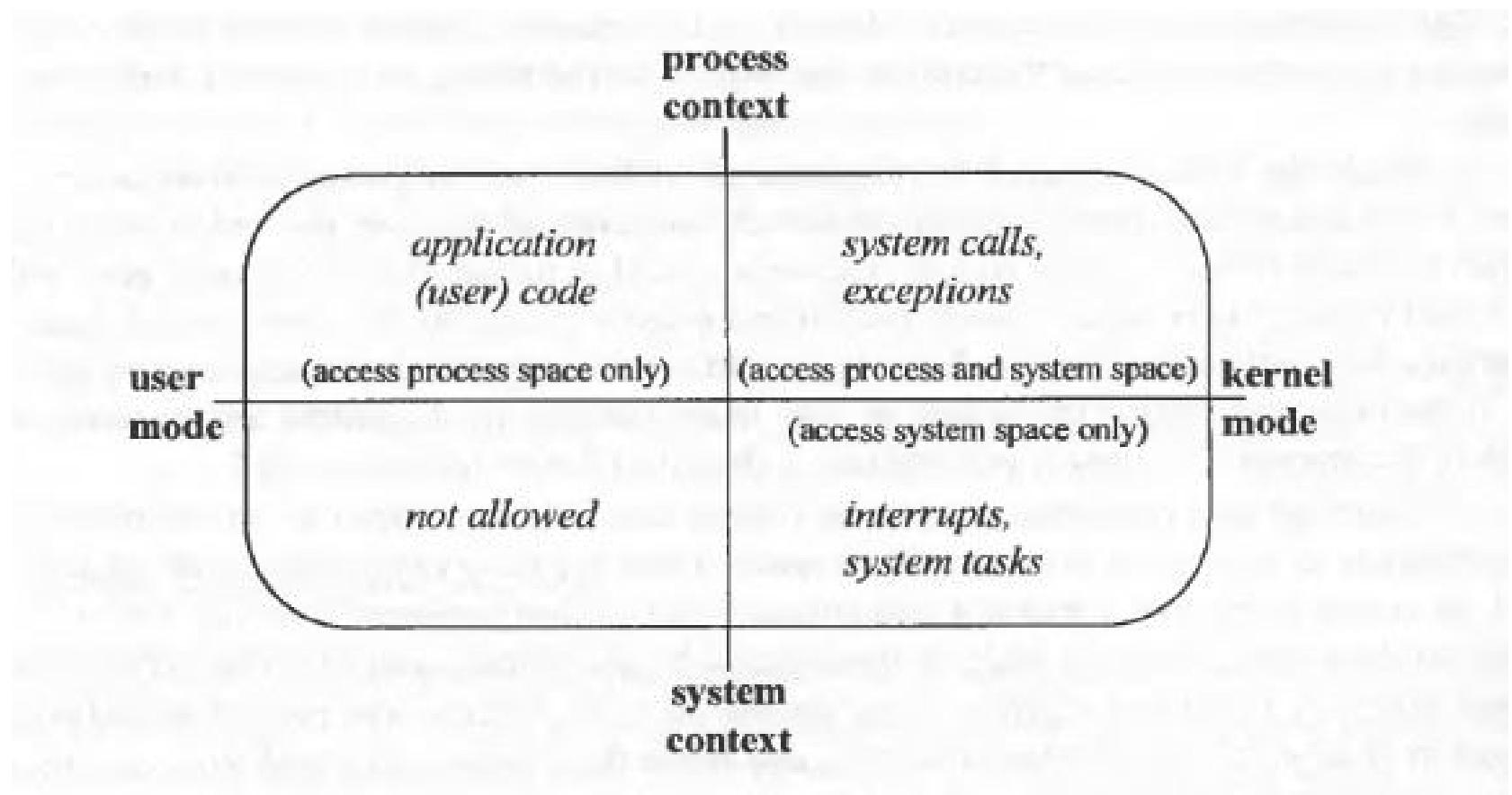
- **OS code is sitting in memory , and runs intermittantly . When?**
  - On a software or hardware interrupt or exception!
  - Event/Interrupt driven OS!
  - Hardware interrupts and exceptions occur asynchronously (un-predictedly) while software interrupts are caused by application code

# Interrupt driven OS code

- **Timer interrupt and multitasking OS**
  - Setting timer register is a privileged instruction.
  - After setting a value in it, the timer keeps ticking down and on becoming zero the timer interrupt is raised again (in hardware automatically)
  - OS sets timer interrupt and “passes control” over to some application code. Now only application code running on CPU !
  - When time allocated to process is over, the timer interrupt occurs and control jumps back to OS (hardware interrupt mechanism)
  - The OS code that runs here (the ISR) is called “scheduler”
  - OS can now schedule another program

# What runs on the processor ?

- 4 possibilities.



# Compilation, Linking, Loading

Abhijit A M

# Review of last few lectures

Boot sequence: BIOS, boot-loader, kernel

- Boot sequence: Process world
  - kernel->init -> many forks+execs() -> ....
- Hardware interrupts, system calls, exceptions
- Event driven kernel
- System calls
  - Fork, exec, ... open, read, ...

# What are compiler, assembler, linker and loader, and C library

System Programs/Utilities

Most essential to make a kernel really usable

# Standard C Library

- A collection of some of the most frequently needed functions for C programs
  - `scanf`, `printf`, `getchar`, system-call wrappers (`open`, `read`, `fork`, `exec`, etc.), ...
- An machine/object code file containing the machine code of all these functions
  - Not a source code! Neither a header file. More later.
- Where is the C library on your computer?
  - `/usr/lib/x86_64-linux-gnu/libc-2.31.so`

# Compiler

- application program, which converts one (programming) language to another
  - Most typically compilers convert a high level language like C, C++, etc. to Machine code language
- E.g. GCC /usr/bin/gcc
  - Usage: e.g.
  - \$ gcc main.c -o main
  - Here main.c is the C code, and "main" is the object/machine code file generated
- Input is a file and output is also a file.
- Other examples: g++ (for C++), javac (for java)

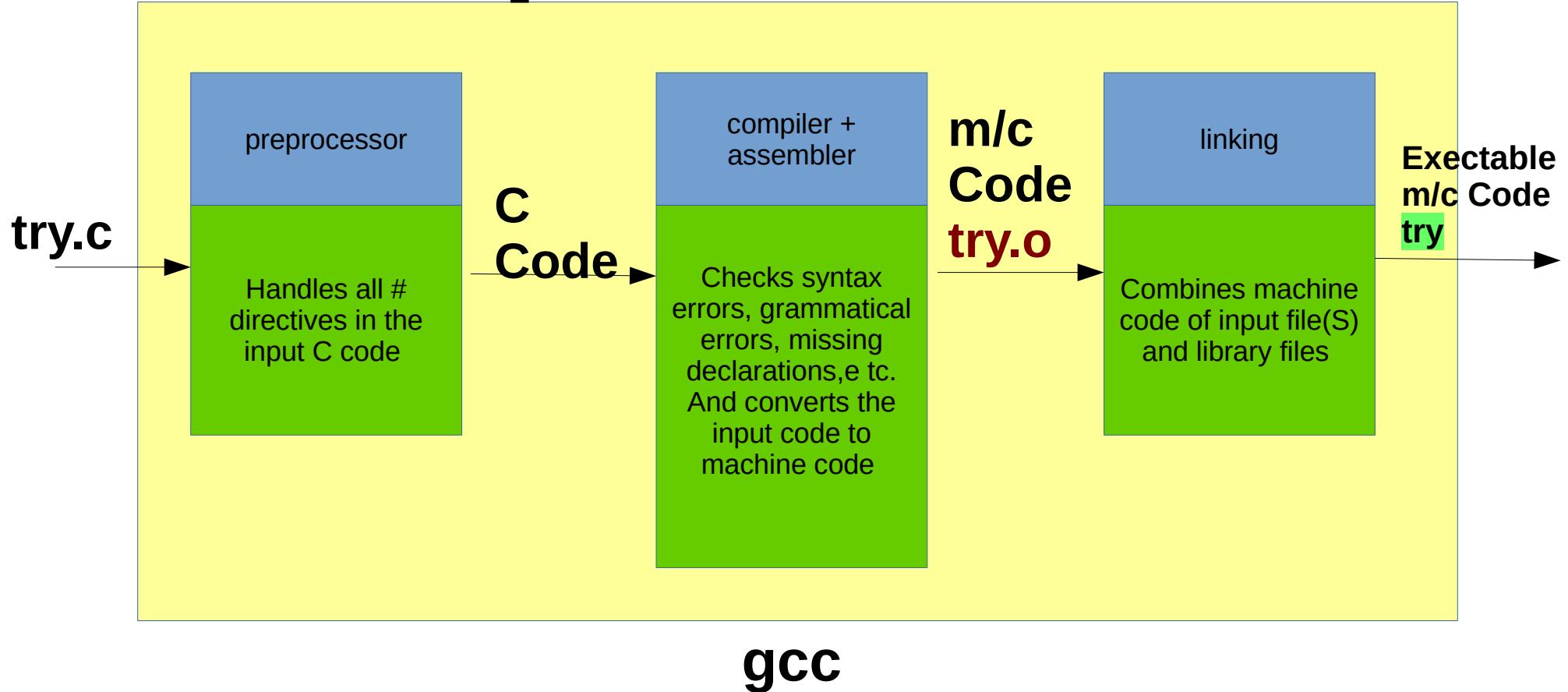


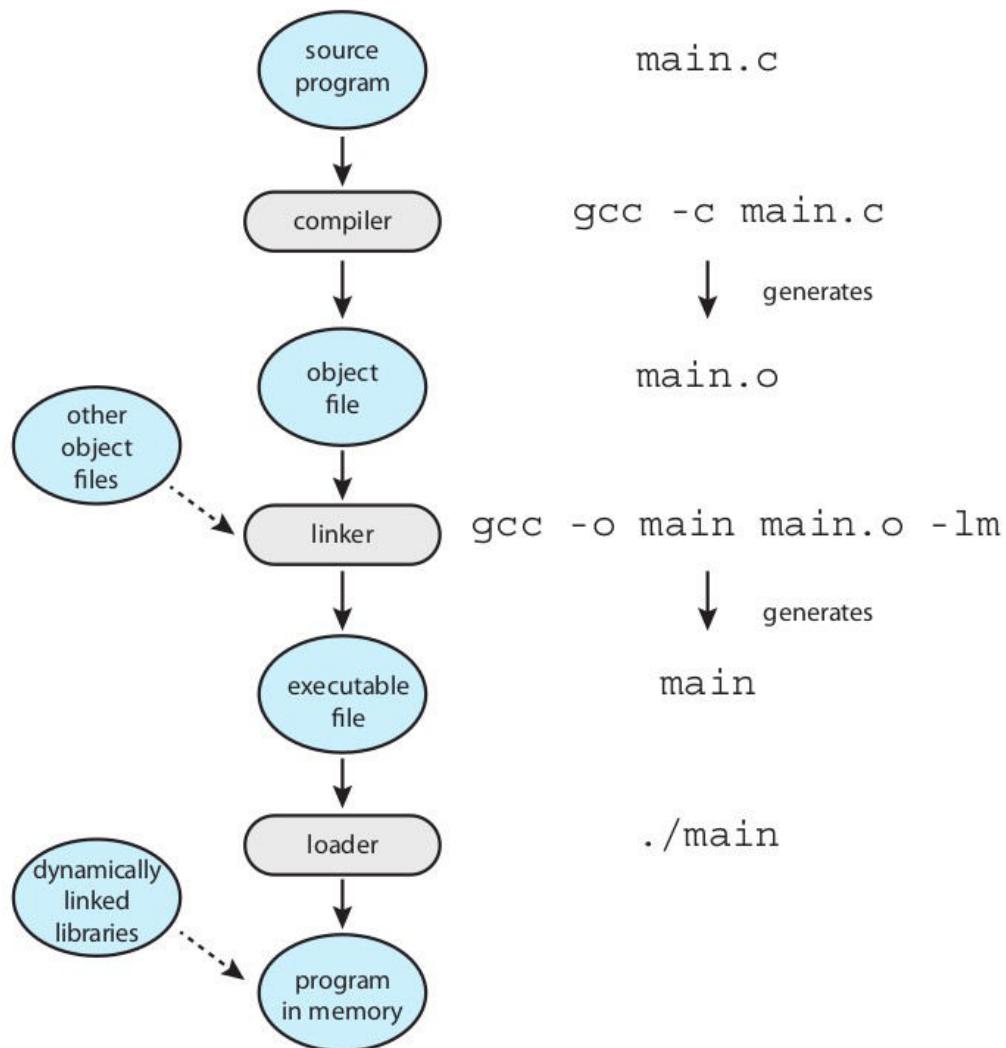
# Assembler

- application program, converts assembly code into machine code
- What is assembly language?
  - Human readable machine code language.
- E.g. x86 assembly code
  - mov 50, r1
  - add 10, r1
  - mov r1, 500
- Usage. eg..
  - \$ as something.s -o something



# Compilation Process





- From the textbook

**Figure 2.11** The role of the linker and loader.

# Example

**try.c**

```
#include <stdio.h>
#define MAX 30
int f(int, int);
int main() {
    int i, j, k;
    scanf("%d%d", &i, &j);
    k = f(i, j) + MAX;
    printf("%d\n", k);
    return 0;
}
```

**f.c**

```
int g(int);
#define ADD(a, b) (a + b)
int f(int m, int n) {
    return ADD(m,n) + g(10);
}
```

**g.c**

```
int g(int x) {
    return x + 10;
}
```

Try these commands, observe the output/errors/warnings, and try to understand what is happening

```
$ gcc try.c
$ gcc -c try.c
$ gcc -c f.c
$ gcc -c g.c
$ gcc try.o f.o g.o -o try
$ gcc -E try.c
$ gcc -E f.c
```

# More about the steps

- Pre-processor
  - `#define ABC XYZ`
    - cut ABC and paste XYZ
  - `# include <stdio.h>`
    - copy-paste the file stdio.h
    - There is no CODE in stdio.h, only typedefs, #includes, #define, #ifdef, etc.
- Linking
  - Normally links with the standard C-library by default
  - To link with other libraries, use the -l option of gcc

# Using gcc itself to understand the process

- Run only the preprocessor
  - cc -E test.c
  - Shows the output on the screen
- Run only till compilation (no linking)
  - cc -c test.c
  - Generates the “test.o” file , runs compilation + assembler
  - gcc -S main.c
  - One step before machine code generation, stops at assembly code
- Combine multiple .o files (only linking part)

# Linking process

- Linker is an application program
  - On linux, it's the "ld" program
  - E.g. you can run commands like `$ ld a.o b.o -o c.o`
  - Normally you have to specify some options to ld to get a proper executable file.
- When you run gcc
  - `$ cc main.o f.o g.o -o try`
  - the CC will internally invoke "ld" . ld does the job of linking

# Linking process

- The resultant file "try" here, will contain the codes of all the functions and linkages also.
- **What is linking?**
  - "connecting" the call of a function with the code of the function.
- What happens with the code of printf()
  - The linker or CC will automatically pick up code from the libc.so.6 file for the functions.

# Executable file format

- An executable file needs to execute in an environment created by OS and on a particular processor
  - Contains machine code + other information for OS
  - Need for a structured-way of storing machine code in it
- Different OS demand different formats
  - Windows: PE, Linux: ELF, Old Unixes: a.out, etc.
- ELF : The format on Linux.
- Try this
  - `$ file /bin/ls`
  - `$ file /usr/lib/x86_64-linux-gnu/libc-2.31.so`

# Exec() and ELF

- When you run a program
  - \$ ./try
  - Essentially there will be a fork() and exec("./try", ...)
  - So the kernel has to read the file "./try" and understand it.
  - So each kernel will demand its own object code file format.
  - Hence ELF, EXE, etc. Formats
- ELF is used not only for executable (complete machine code) programs, but also for partially compiled files e.g. main.o and library files like libc.so.6
- What is a.out?
  - "a.out" was the name of a format used on earlier Unixes.
  - It so happened that the early compiler writers, also created executable with default name 'a.out'

# Utilities to play with object code files

- objdump
  - \$ objdump -D -x /bin/ls
  - Shows all disassembled machine instructions and “headers”
- hexdump
  - \$ hexdump /bin/ls
  - Just shows the file in hexadecimal
- readelf
  - Alternative to objdump
- ar
  - To create a “statically linked” library file
  - \$ ar -crs libmine.a one.o two.o
- Gcc to create shared library
  - \$ gcc hello.o -shared -o libhello.so
- To see how gcc invokes as, ld, etc; do this
  - \$ gcc -v hello.c -o hello
  - /\*  
<https://stackoverflow.com/questions/1170809/how-to-get-gcc-linker-command>

# Linker, Loader, Link-Loader

- Linker or linkage-editor or link-editor
  - The “ld” program. Does linking.
- Loader
  - The exec(). It loads an executable in the memory.
- Link-Loader
  - Often the linker is called link-loader in literature. Because where were days when the linker and loader’s jobs were quite over-lapping.

# Static, dynamic / linking, loading

- Both linking and loading can be
  - Static or dynamic
  - More about this when we learn memory management
- An important fundamental:
  - memory management features of processor, memory management architecture of kernel, executable/object-code file format, output of linker and job of loader, are all interdependent and in-separable.
  - They all should fit into each other to make a system work
  - That's why the phrase “system programs”

# Cross-compiler

- Compiler on system-A, but generate object-code file for system-B (target system)
  - E.g. compile on Ubuntu, but create an EXE for windows
- Normally used when there is no compiler available on target system
  - see gcc -m option
- See [https://wiki.osdev.org/GCC\\_Cross-Compiler](https://wiki.osdev.org/GCC_Cross-Compiler)

# **Inter Process Communication**

# **Revision of process related concepts**

- **PCB, struct proc**
- **Process lifecycle – different states**
- **Queues/Lists of processes**
- **What is “Blocking”**
- **Event driven kernel**

# Before IPC, let's learn more about file related system calls

- **Redirection**

`ls > /tmp/file`

`cat < /etc/passwd`

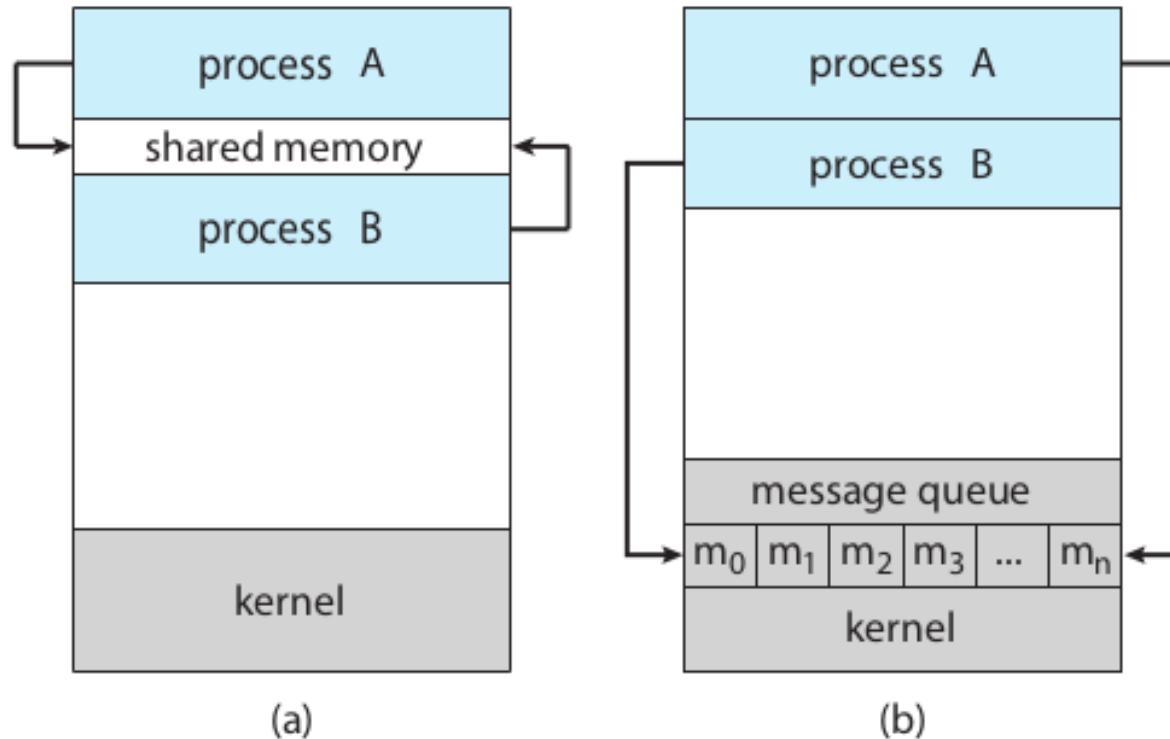
**how does this work?**

- **File descriptors are inherited across fork**

# IPC: Inter Process Communication

- Processes within a system may be independent or cooperating
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing, e.g. copy paste
  - Computation speedup, e.g. matrix multiplication
  - Modularity, e.g. chrome – separate process for display, separate for fetching data
  - Convenience,
- Cooperating processes need interprocess communication (IPC)
- Broadly Two models of IPC
  - Shared memory (examples: shared memory, pipes, ... )
  - Message passing (examples: send/recv on socket, send-recv messages, ...)

# Shared Memory Vs Message Passing



**Each requires OS to provide system calls for**

- Creating the IPC mechanism
- To read/write using the IPC mechanism
- Delete the IPC mechanism

**Note: processes communicating with each other with the help of OS!**

Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.

# Typical code of shared memory type of solutions

## Process P1

```
x = getshm(ID)
```

```
*x = 100;
```

## Process P2

```
x = getshm(ID)
```

```
y = *x
```

# Typical code of message passing type solutions

## Process P1

**send(P2, x)**

or

**broadcast(x)**

## Process P2

**y = receive(P1)**

or

**y = receive(anyone)**

# Example of co-operating processes: Producer Consumer Problem

- **Paradigm for cooperating processes, producer process produces information that is consumed by a consumer process**
  - unbounded-buffer places no practical limit on the size of the buffer
  - bounded-buffer assumes that there is a fixed buffer size

# Example of co-operating processes: Producer Consumer Problem

- Shared data

```
#define BUFFER_SIZE 10
```

```
typedef struct {
```

```
    . . .
```

```
} item;
```

```
item buffer[BUFFER_SIZE];
```

```
int in = 0;
```

```
int out = 0;
```

- Can only use BUFFER\_SIZE-1 elements

# Example of co-operating processes: Producer Consumer Problem

- **Code of Producer**

```
while (true) {  
    /* Produce an item */  
  
    while (((in = (in + 1) % BUFFER SIZE count) == out)  
        ; /* do nothing -- no free buffers */  
    buffer[in] = item;  
    in = (in + 1) % BUFFER SIZE;  
}
```

# Example of co-operating processes: Producer Consumer Problem

- **Code of Consumer**

```
while (true) {  
    while (in == out)  
        ; // do nothing -- nothing to consume  
  
    // remove an item from the buffer  
    item = buffer[out];  
    out = (out + 1) % BUFFER SIZE;  
    return item;  
}
```

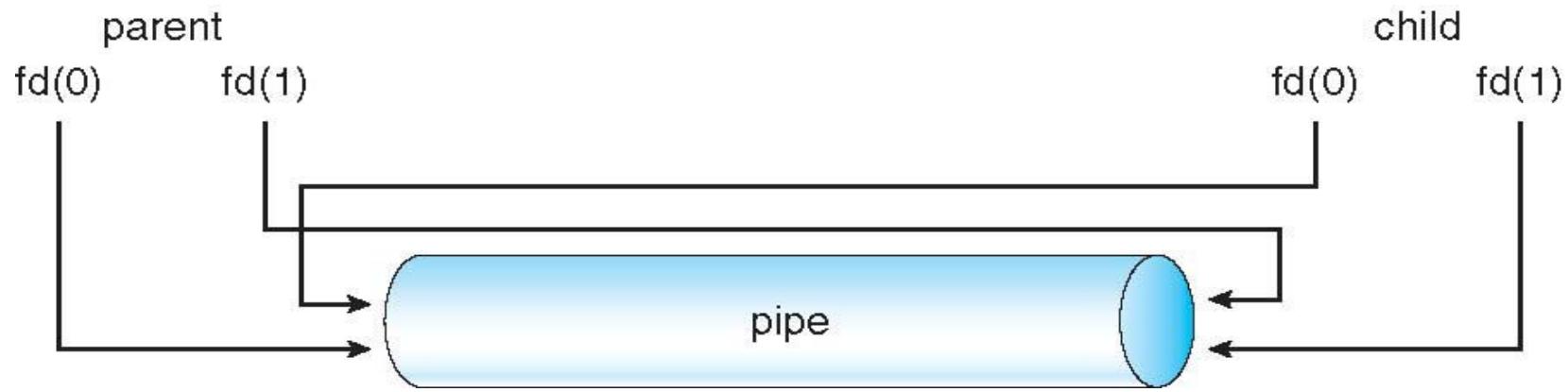
# Pipes

# Pipes for IPC

- **Two types**
  - Unnamed Pipes or ordinary pipes
  - Named Pipe

# Ordinary pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the write-end of the pipe)
- Consumer reads from the other end (the read-end of the pipe)
- Ordinary pipes are therefore unidirectional
- Requires a parent-child (or sibling, etc) kind of relationship between communicating processes



# Named pipes

- Also called FIFO
- Processes can create a “file” that acts as pipe. Multiple processes can share the file to read/write as a FIFO
- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems
- Is not deleted automatically by OS

# Named pipes

- **int mkfifo(const char \*pathname, mode\_t mode);**
- **Example**

# **Shared Memory**

# System V shared memory

- **Process first creates shared memory segment**  
`segment id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);`
- **Process wanting access to that shared memory must attach to it**  
`shared_memory = (char *) shmat(id, NULL, 0);`
- **Now the process could write to the shared memory**  
`sprintf(shared_memory, "Writing to shared memory");`
- **When done, a process can detach the shared memory from its address space**  
`shmdt(shared_memory);`

# **Example of Shared memory**

## **POSIX Shared Memory**

- **What is POSIX?**
  - Portable Operating System Interface (**POSIX**)
  - family of standards
  - specified by the IEEE Computer Society
  - for maintaining compatibility between operating systems.
  - API (system calls), shells, utility commands for compatibility among UNIXes and variants

# POSIX Shared Memory

- **shm\_open**
- **ftruncate**
- **Mmap**
- **See the example in Textbook**

# **Message passing**

# Message Passing

- **Message system – processes communicate with each other using send(), receive() like syscalls given by OS**
- **IPC facility provides two operations:**
  - send(message) – message size fixed or variable
  - Receive(message)
- **If P and Q wish to communicate, they need to:**
  - establish a communication link between them
  - exchange messages via send/receive
- **Communication link can be implemented in a variety of ways**

# Message Passing using “Naming”

- **Pass a message by “naming” the receiver**
  - **A) Direct communication with receiver**
    - **Receiver is identified by sender directly using it's name**
  - **B) Indirect communication with receiver**
    - **Receiver is identified by sender in-directly using it's 'location of receipt'**

# Message passing using direct communication

- **Processes must name each other explicitly:**
  - send (P, message) – send a message to process P
  - receive(Q, message) – receive a message from process Q
- **Properties of communication link**
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

# Message passing using IN-direct communication

- **Messages are directed and received from mailboxes (also referred to as ports)**
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- **Properties of communication link**
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

# Message passing using IN-direct communication

- **Operations**
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- **Primitives are defined as:**
  - `send(A, message)` – send a message to mailbox A
  - `receive(A, message)` – receive a message from mailbox A

# Message passing using IN-direct communication

- **Mailbox sharing**
  - P1, P2, and P3 share mailbox A
  - P1, sends; P2 and P3 receive
  - Who gets the message?
- **Solutions**
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

# Message Passing implementation: Synchronization issues

- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
  - Blocking send has the sender block until the message is received
  - Blocking receive has the receiver block until a message is available
- Non-blocking is considered asynchronous
  - Non-blocking send has the sender send the message and continue
  - Non-blocking receive has the receiver receive a valid message or null

# Producer consumer using blocking send and receive

## Producer

```
message next_produced;  
while (true) {  
/* produce an item in  
next_produced */  
send(next_produced);  
}
```

## Consumer

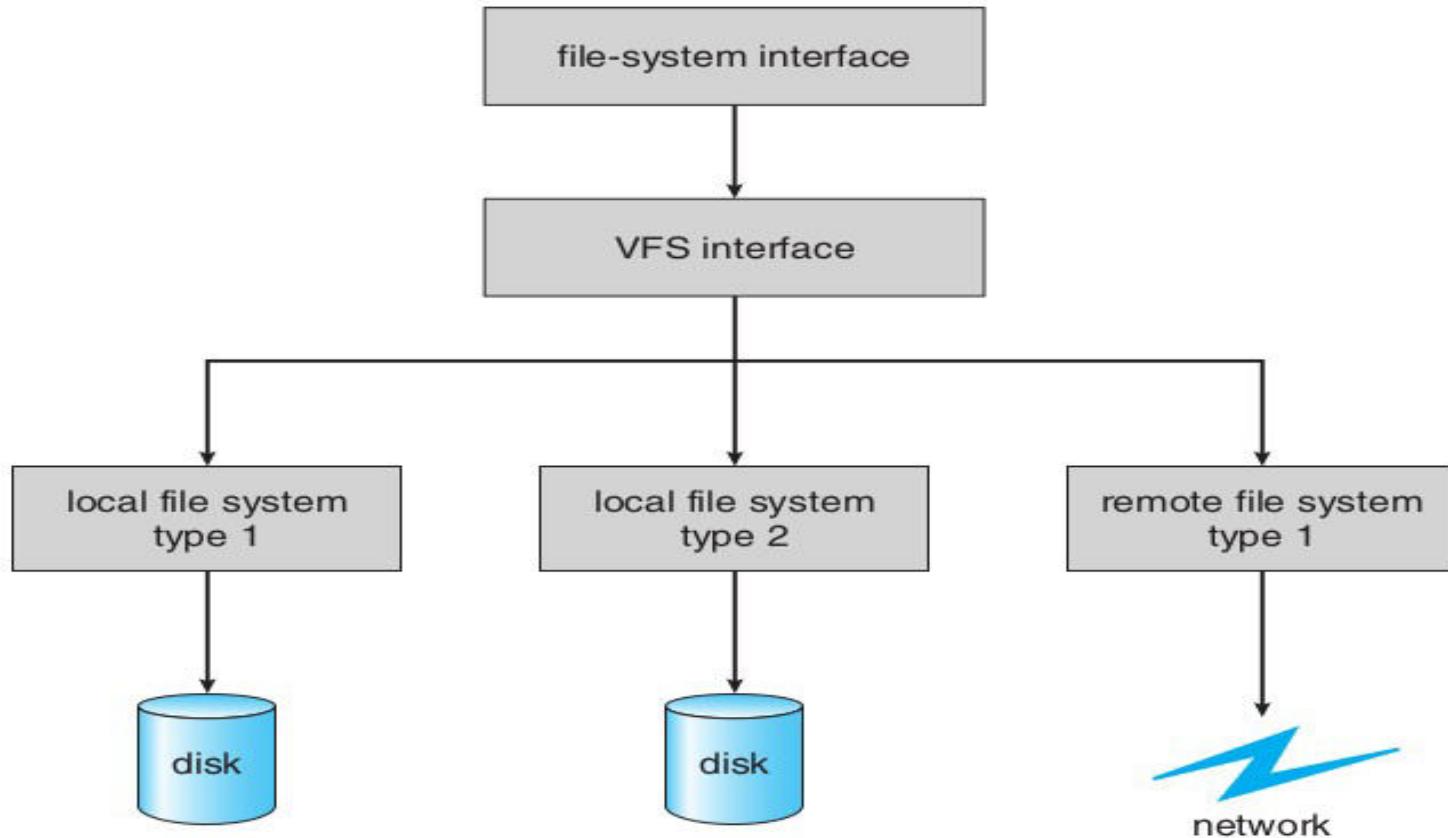
```
message  
next_consumed;  
while (true) {  
receive(next_consumed);  
}
```

# Message Passing implementation: choice of Buffering

- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages
    - Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of n messages
    - Sender must wait if link full
  3. Unbounded capacity – infinite length
    - Sender never waits



**VFS**



**Figure 15.5** Schematic view of a virtual file system.

# VFS

- Consider this

*/dev/sda1* is “/”

*/dev/sda2* is mounted on “/a/b” folder

How does this work in kernel?

`open(“/a/b/c/d”, O_RDONLY)`

- Consider xv6 code

- `sys_open` -> `namei` -> `namex` -> (`skipelem`, `dirlookup`, `ilock`)
- `Dirlookup()` of “c” in “/a/b” should return : Not the inode of “c” on */dev/sda1* but inode of “/” on */dev/sda2*

# VFS

- **Object Oriented Programming in C (let's see example of this)**
  - Clever use of function pointers
- **There is an “abstract” file system class (VFS), and there are concrete file system classes (ext2, vfat, ...)**
  - sys\_read → fileread → readi() becomes
  - sys\_read → fileread → (i->i\_ops->read)()
- **Inode is a generic inode**
  - Contains file system specific inode pointer
  - And file system specific inode operations
  - Fields setup during namei()

```
struct inode_operations {  
    int (*readi) (int, char *, int);  
    int (*writei) (int, char *, int);  
    ....  
}  
struct inode {  
    int mode,  
    Int uid;  
    ....  
    void *inode_specific;  
    struct inode_ops i_ops;  
}
```

**Efficiency and Performance  
(and the risks created  
while trying to achieve it!)**

# Efficiency

- Efficiency dependent on:
  - Disk allocation and directory algorithms
  - Types of data kept in file's directory entry
  - Pre-allocation or as-needed allocation of metadata structures
  - Fixed-size or varying-size data structures
  -

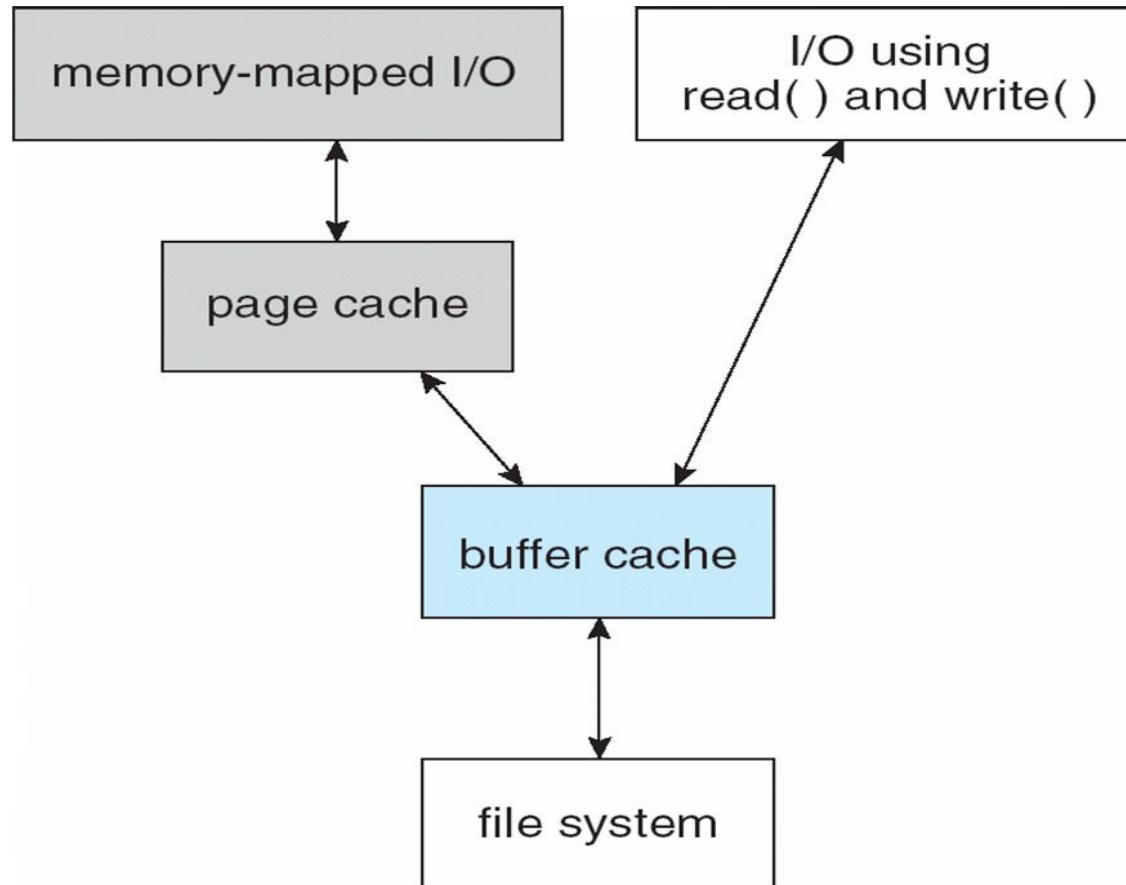
# Performance

- Keeping data and metadata close together
- Buffer cache – separate section of main memory for frequently used blocks
- Synchronous writes sometimes requested by apps or needed by OS
- No buffering / caching – writes must hit disk before acknowledgement
- Asynchronous writes more common, buffer-able, faster
- Free-behind and read-ahead – techniques to optimize sequential access
- Reads frequently slower than writes

# Page cache

- A page cache caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure

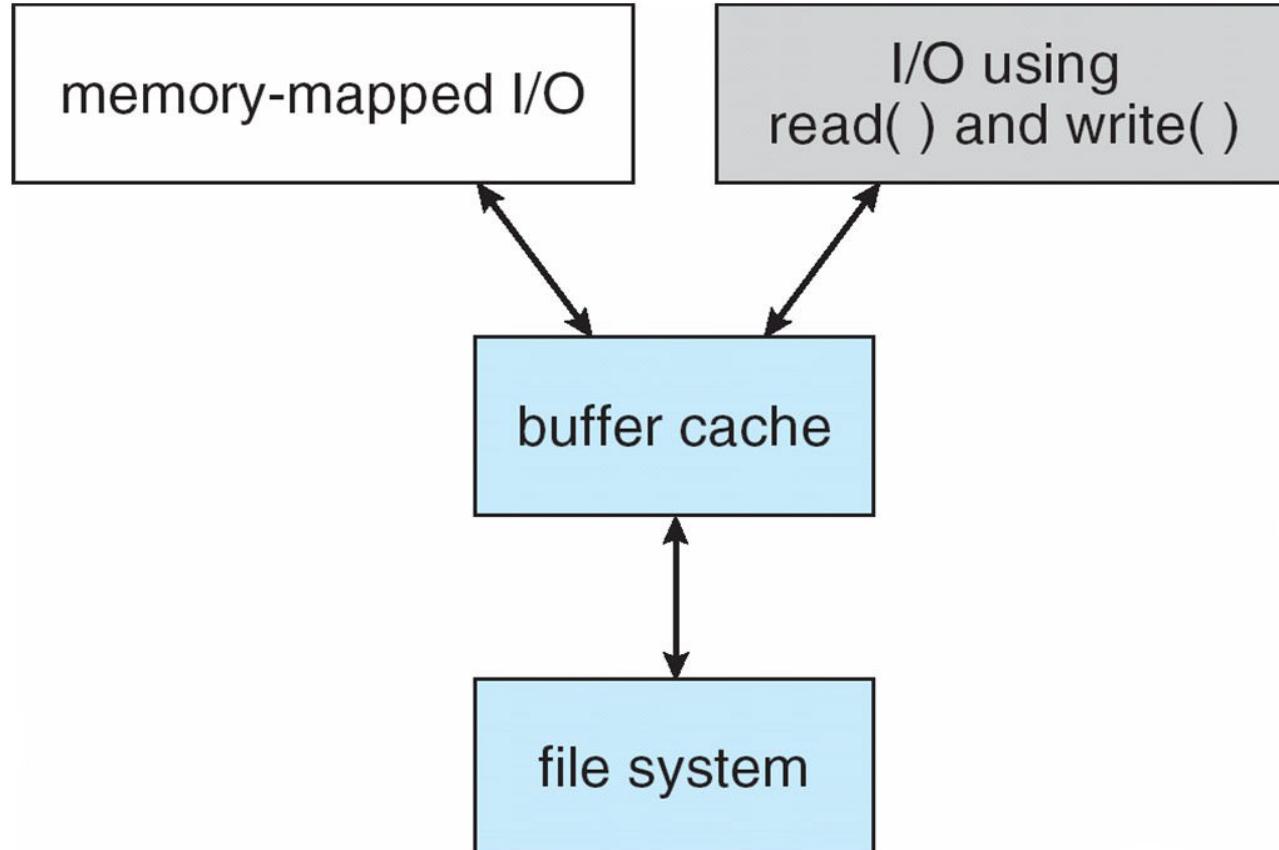
# I/O Without a Unified Buffer Cache



# Unified buffer cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid double caching
- But which caches get priority, and what replacement algorithms to use?

# I/O Using a Unified Buffer Cache



# Recovery

- **Problem. Consider creating a file on ext2 file system.**
  - Following on disk data structures will/may get modified
  - Directory data block, new directory data block, block bitmap, inode table, inode table bitmap, group descriptor, super block, data blocks for new file, more data block bitmaps, ...
  - All cached in memory by OS
- **Delayed write – OS writes changes in its in-memory data structures, and schedules writes to disk when convenient**
  - Possible that some of the above changes are written, but some are not
  - Inconsistent data structure! --> Example: inode table written, inode bitmap written, but directory data block not written

# Recovery

- **fsck: Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies**
  - Can be slow and sometimes fails
- **Use system programs to back up data from disk to another storage device (magnetic tape, other magnetic disk, optical)**
- **Recover lost file or disk by restoring data from backup**
- **Faster recovery? - “log structured file system” or “journaling file system” can help**

# Log structured file systems

- Log structured (or journaling) file systems record each metadata update to the file system as a transaction
- All transactions are written to a log
  - A transaction is considered committed once it is written to the log (sequentially)
  - Sometimes to a separate device or section of disk
  - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system structures
  - When the file system structures are modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- Faster recovery from crash, removes chance of inconsistency of metadata

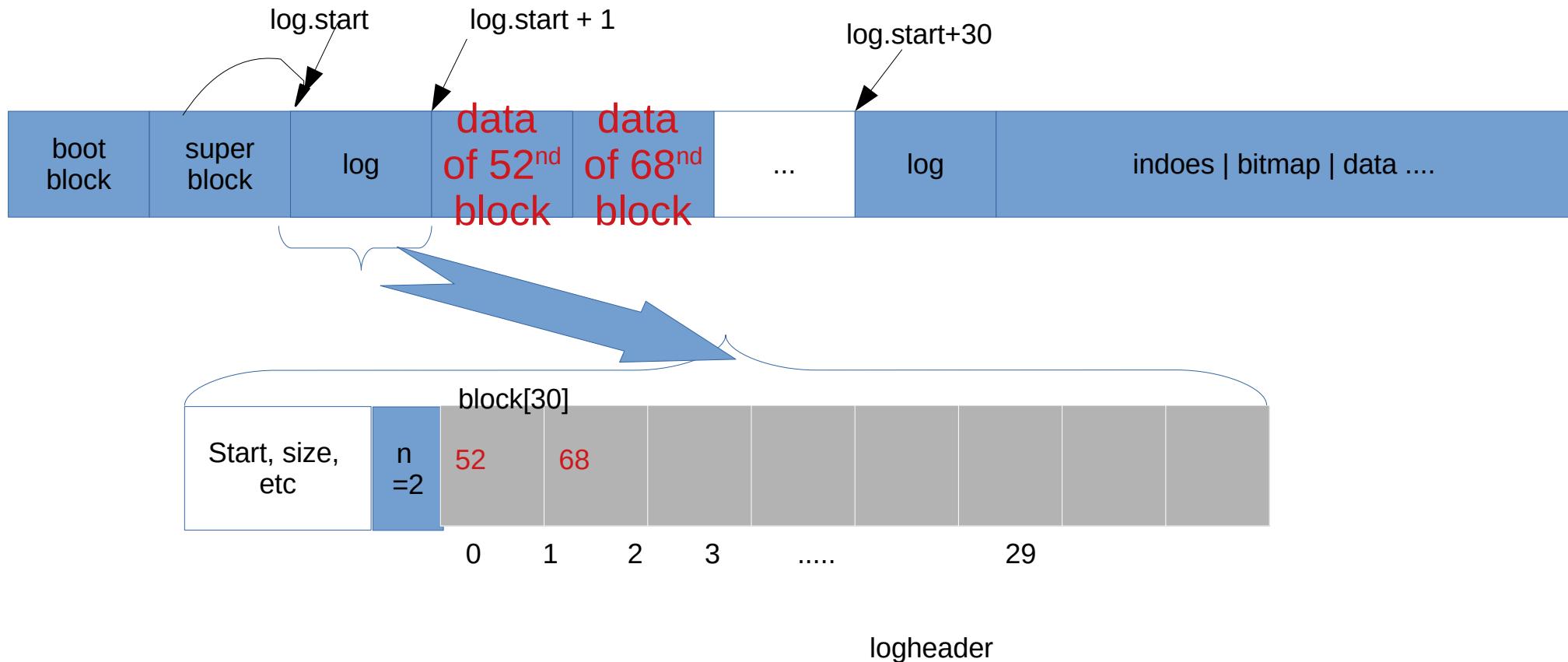
# Journaling file systems

- Veritas FS
- Ext3, Ext4
- Xv6 file system!

# log in xv6

- a mechanism of recovery from disk
- Concept: multiple write operations needed for system calls (e.g. ‘open’ system call to create a file in a directory)
  - some writes succeed and some don’t
  - leading to inconsistencies on disk
- In the log, all changes for a ‘transaction’ (an operation) are either written completely or not at all
- During recovery, completed operations can be “rerun” and incomplete operations neglected

# log on disk



# log in xv6

- **xv6 system call does not directly write the on-disk file system data structures.**
- **A system call calls begin\_op() at begining and end\_op() at end**
  - begin\_op() increments log.outstanding
  - end\_op() decrements log.outstanding, and if it's 0, then calls commit()
- **During the code of system call, whenever a buffer is modified, (and done with)**
  - log\_write() is called
  - This copies the block in an array of blocks inside log, the block is not written in its actual place in FS as of now
- **when finally commit() is called, all modified blocks are copied to disk in the file system**

# log

```
struct logheader { // ON DISK
    int n; // number of entries in use in block[] below
    int block[LOGSIZE]; // List of block numbers stored
};

struct log { // only in memory
    struct spinlock lock;
    int start; // first log block on disk (starts with logheader)
    int size; // total number of log blocks (in use out of 30)
    int outstanding; // how many FS sys calls are executing.
    int committing; // in commit(), please wait.
    int dev; // FS device
    struct logheader lh; // copy of the on disk logheader
};

struct log log;
```

# Typical use case of logging

```
/* In a system call code */  
begin_op();  
...  
bp = bread(...);  
bp->data[...] = ...;  
log_write(bp);  
...  
end_op();
```

prepare for logging. Wait if logging system is not ready or 'committing'. + **+outstanding**

read and get access to a data block – as a buffer

modify buffer

note down this buffer for writing, in log. proxy for bwrite(). Mark **B\_DIRTY**. Absorb multiple writes into one.

**Syscall done. write log and all blocks. --outstanding.**

**If outstanding = 0, commit().**

# Example of calls to logging

```
//file_write() code  
  
begin_op();  
  
ilock(f->ip);  
  
/*loop */ r = writei(f->ip, ...);  
  
iunlock(f->ip);  
  
end_op();
```

- each writei() in turn calls bread(), log\_write() and brelse()
  - also calls iupdate(ip) which also calls bread, log\_write and brelse
- Multiple writes are combined between begin\_op() and end\_op()

# Logging functions

- **Initlog()**
  - Set fields in global `log.xyz` variables, using FS superblock
  - Recovery if needed
  - **Called from first forkret()**
- **Following three called by FS code**
- **begin\_op(void)**
  - Increment `log.outstanding`
- **end\_op(void)**
  - Decrement `log.outstanding` and call `commit()` if it's zero
- **log\_write(buf \*)**
  - Remember the specified block number in `log.lh.block[]` array
  - Set the block to be dirty
- **write\_log(void)**
  - **Called only from commit()**
  - Use block numbers specified in `log.lh.block` and copy those blocks from memory to log-blocks
- **commit(void)**
  - **Called only from end\_op()**
  - `write_log()`
  - Write header to disk log-header
  - Copy from log blocks to actual FS blocks
  - Reset and write log header again













# **Scheduling Algorithms**

**Abhijit A.M.**  
**abhijit.comp@coep.ac.in**

**Credits: Slides from os-book.com**

# Calculations of different scheduling criteria

- If you want to evaluate an algorithm practically, you need a proper workload !
  - Processes with CPU and I/O bursts
  - Different durations of CPU bursts
  - Different durations of I/O bursts
    - How to do this programmatically?
    - How to ensure that after 2 seconds an I/O takes place?
  - Need periods when system will be “idle” – no process schedulable !

# Calculations of different criteria

- **CPU Utilization**

- % time spent in doing ‘useful’ work

- **What is useful work?**

- **On linux**

- there is an “idle” thread, scheduled when no other task is RUNNABLE
      - Not running idle thread is productive work
      - Includes process + scheduling time + interrupts

- **On other systems?**

- Need to define

- **On xv6**

- We can say that time spent in the loop selecting a process is idle work

# Calculations of different criteria

- **Throughput**
  - # processes that complete execution per unit time
  - Formula: total # processes completed / total time
  - Simply divide by your total workload that completed by the time taken
  - Depends on the workload as well. ‘long’ or ‘short’ processes.
  - If too many short processes , then throughput may appear to be high, like 10s of processes per second

# Calculations of different criteria

- **Turnaround time**
  - Amount of time required for one process to complete
  - For every process, note down the starting and ending time, difference is TA-time
  - For process P1 : (Time when process ended – time when process started)
    - = Sum of time spent in (ready queue + running + waiting for I/O)
  - One can find the average T.A. time

# Calculations of different criteria

- **Waiting time**
  - amount of time a process has been waiting in the *ready* queue.
  - To be minimised.
  - Part of Turn Around time
  - CPU scheduling does not affect waiting time in I/O queues, it affects time in ready queue

# Scheduling Criteria

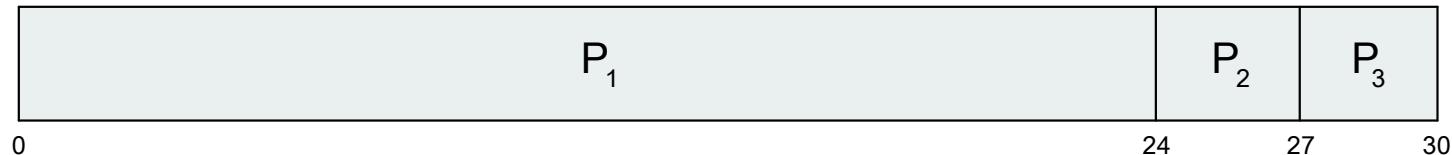
- **Response time**
  - amount of time it takes from when a request was submitted until the first response (not full output) is produced, (for time-sharing environment).
  - To be minimised.
  - E.g. time between your press of a key , and that key being shown in screen

# **Challenges in implementing the scheduling algorithms**

- **Not possible to know number of CPU and I/O bursts and the duration of each before the process runs !**
  - Although when we do numerical problems around each algorithm, we assume some values for CPU and I/O bursts, so the problems are solved in “hindsight” !
  -

# GANTT chart

- A timeline chart showing the sequence in which processes get scheduled
- Used for analysing a scheduling algorithm



# **Scheduling Algorithms**

# First- Come, First-Served (FCFS) Scheduling

Process    Burst Time

P1              24

P2              3

P3              3

Suppose that the processes arrive in the order: P1 , P2 , P3

The Gantt Chart for the schedule is:



Waiting time for P1 = 0; P2 = 24; P3 = 27

Average waiting time:  $(0 + 24 + 27)/3 = 17$

Non Pre-emptive algorithm

# FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

P<sub>2</sub> , P<sub>3</sub> , P<sub>1</sub>

The Gantt chart for the schedule is:



Waiting time for P<sub>1</sub> = 6; P<sub>2</sub> = 0; P<sub>3</sub> = 3

Average waiting time:  $(6 + 0 + 3)/3 = 3$

Much better than previous case

# FCFS: Convoy effect

- Consider one CPU-bound and many I/O-bound processes
- CPU bound process is scheduled, I/O bound processes are waiting in I/O queues
- I/O bound processes finish I/O and move to ready queue, and wait for CPU bound process to finish
- I/O devices Idle
- CPU bound process over, goes for I/O. I/O bound processes run quickly, move to I/O queues again
  - CPU idle
- CPU bound process will run when it's ready to run
- Same process will repeat
- --> Lower CPU utilisation
  - Better if I/O bound processes run first

# FCFS: further evaluation

- **Troublesome for interactive processes**
  - CPU bound process may hog CPU
  - Interactive process may not get a chance to run early and response time may be quite bad

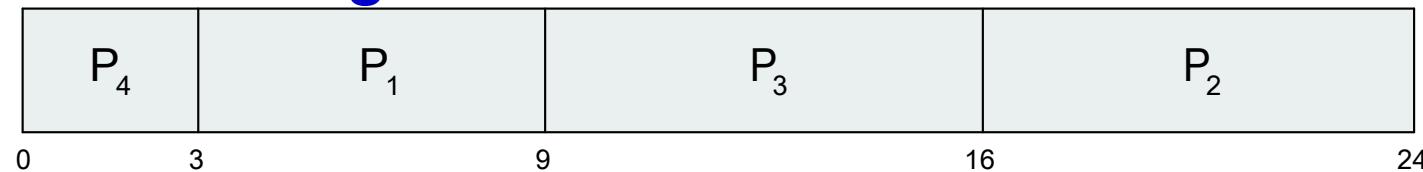
# **Shortest-Job-First (SJF) Scheduling**

- **Associate with each process the length of its next CPU burst**
  - Use these lengths to schedule the process with the shortest time. Better name – **Shortest Next CPU Burst Scheduler**
- **SJF is optimal – gives minimum average waiting time for a given set of processes**
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user – bad idea, unlikely to know!

# Example of SJF

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

SJF scheduling chart



$$\text{Average waiting time} = (3 + 16 + 9 + 0) / 4 = 7$$

# Determining Length of Next CPU Burst

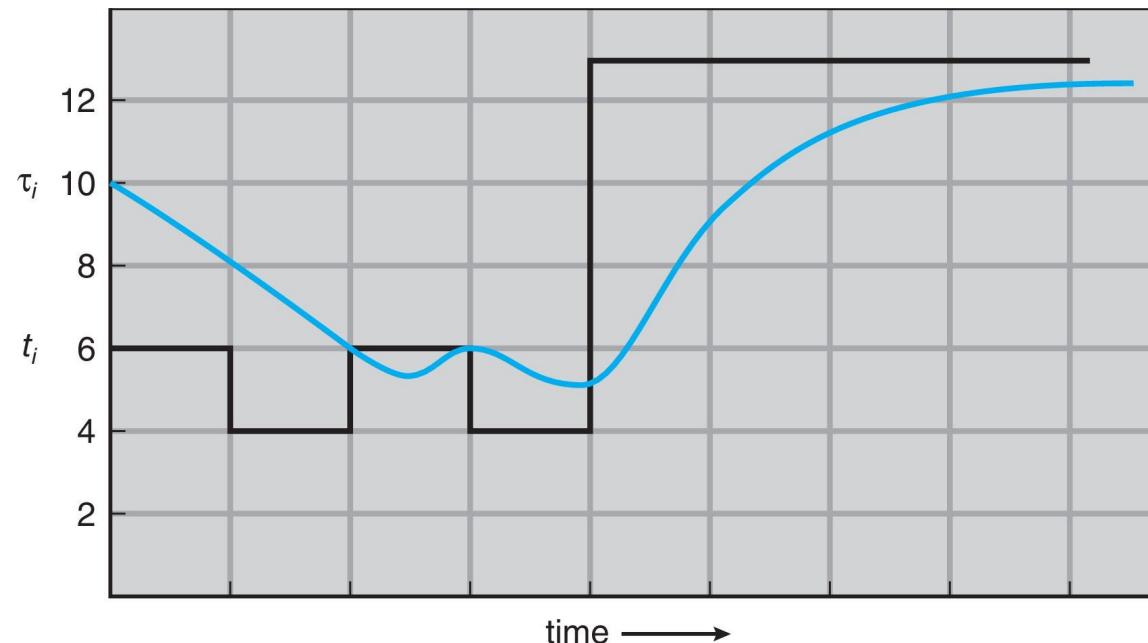
- Not possible to implement SJF as can't know “next” CPU burst. Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging
  1.  $t_n$  = actual length of  $n^{th}$  CPU burst
  2.  $\tau_{n+1}$  = predicted value for the next CPU burst
  3.  $\alpha$ ,  $0 \leq \alpha \leq 1$
  4. Define:  $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$ .
- Commonly,  $\alpha$  set to  $\frac{1}{2}$
- Preemptive version called shortest-remaining-time-first

# Examples of Exponential Averaging

- $\alpha = 0$ 
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$ 
  - $\tau_{n+1} = \alpha t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:
  - $$\begin{aligned}\tau_{n+1} &= \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots \\ &\quad + (1 - \alpha)^j \alpha t_{n-j} + \dots \\ &\quad + (1 - \alpha)^{n+1} \tau_0\end{aligned}$$
- Since both  $\alpha$  and  $(1 - \alpha)$  are less than or equal to 1, each successive term has less weight than its predecessor

# Prediction of the Length of the Next CPU Burst

- $\alpha = 1/2$  ,  $\tau_0 = 10$



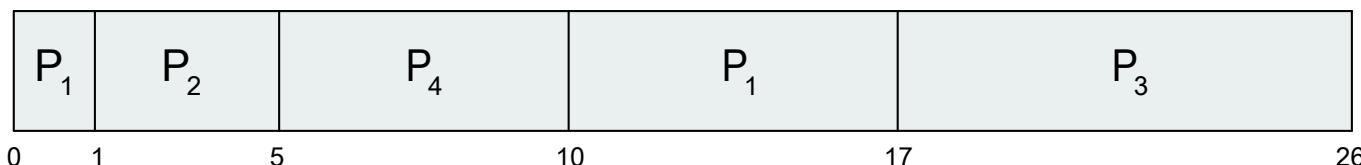
CPU burst ( $t_i$ )	6	4	6	6	4	13	13	13	...
"guess" ( $\tau_i$ )	10	8	6	6	5	9	11	12	...

# Example of Shortest-remaining-time-first

Preemptive SJF = SRTF. Now we add the concepts of varying arrival times and preemption to the analysis

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Preemptive SJF Gantt Chart



Average waiting time =  $[(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5 \text{ msec}$

# Round Robin (RR) Scheduling

- **Each process gets a small unit of CPU time (time quantum q), usually 10-100 milliseconds.**
  - After this time has elapsed, the process is preempted and added to the end of the ready queue.
- **If there are n processes in the ready queue and the time quantum is q, then each process gets  $1/n$  of the CPU time in chunks of at most q time units at once.**
  - No process waits more than  $(n-1)q$  time units.

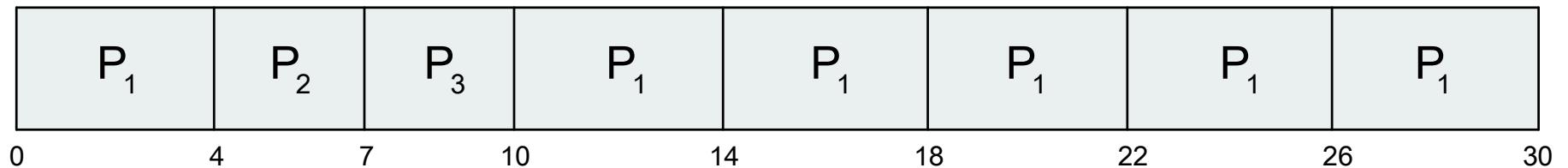
# Round Robin (RR) Scheduling

- Timer interrupts every quantum to schedule next process
- Performance
  - $q$  large  $\Rightarrow$  FIFO
  - $q$  small  $\Rightarrow$   $q$  must be large with respect to context switch, otherwise overhead is too high

# Example of RR with Time Quantum = 4

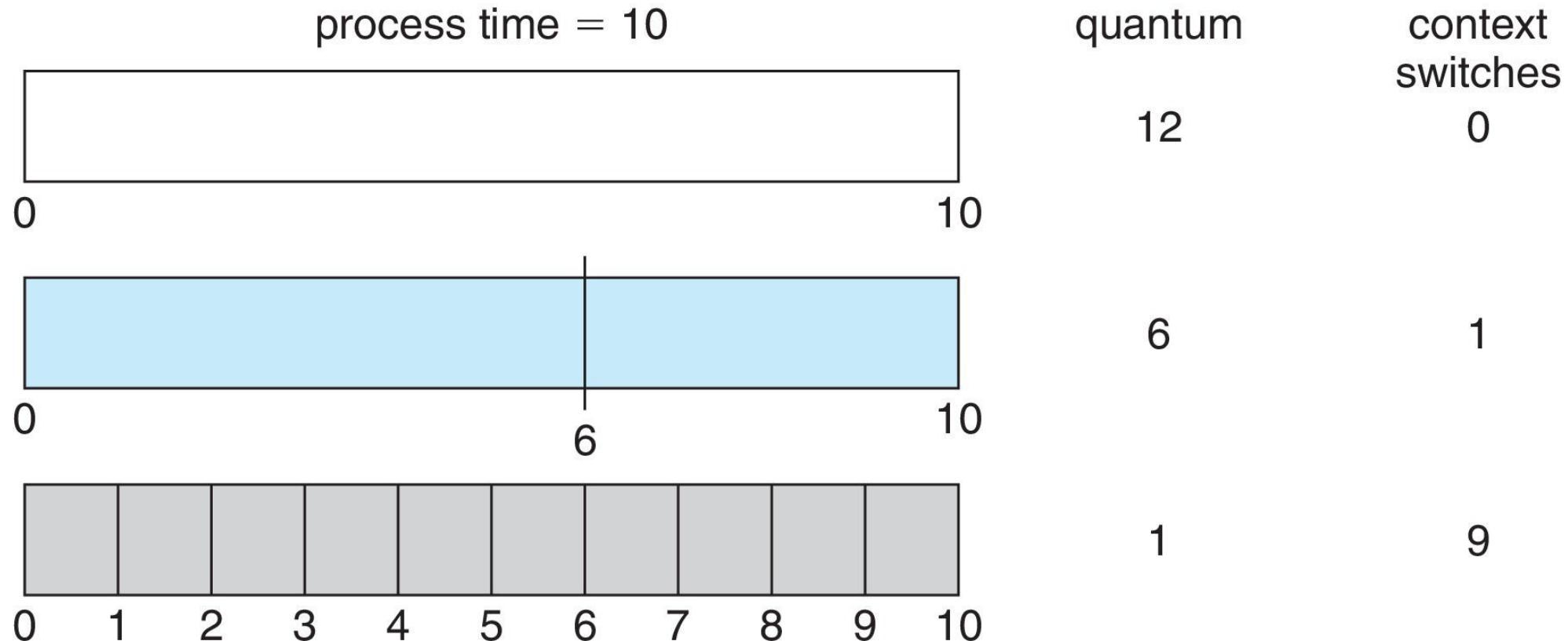
<u>Process</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

The Gantt chart is:

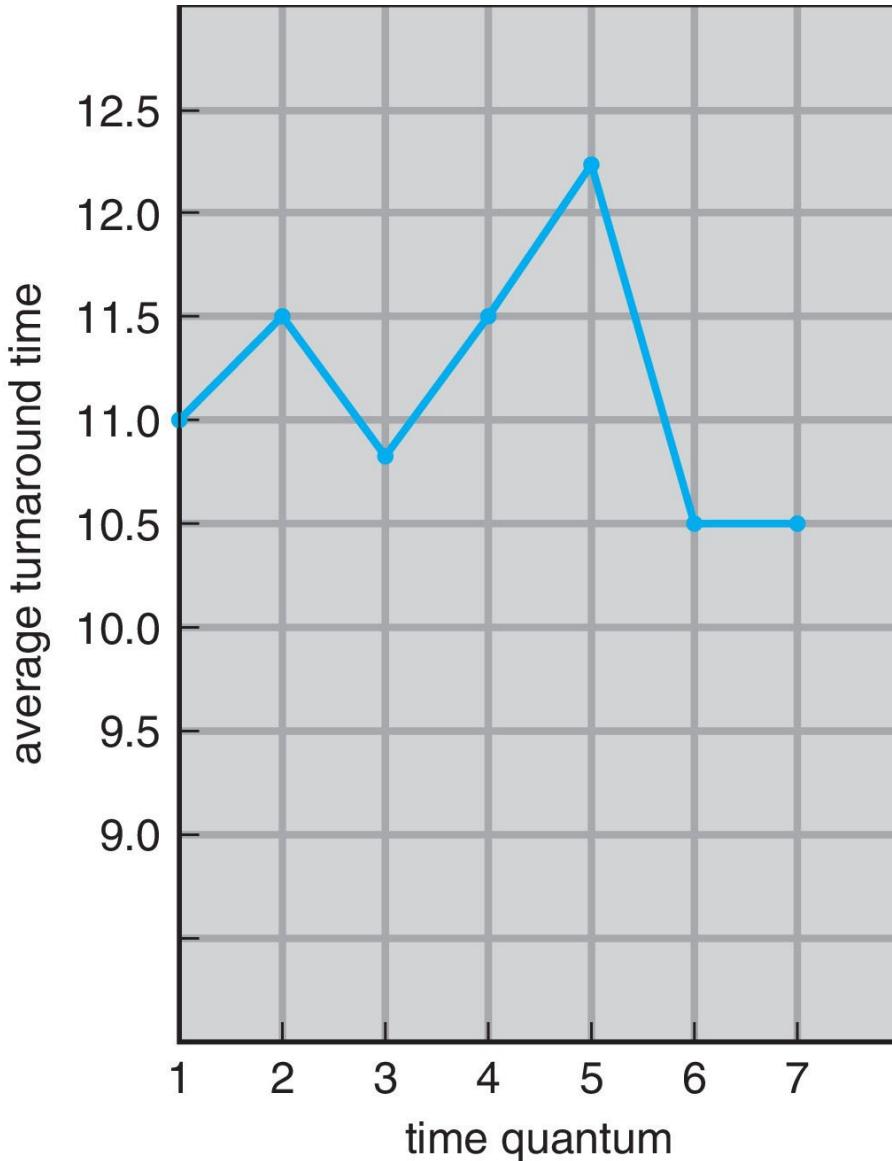


Typically, higher average turnaround than SJF, but better *response*  
q should be large compared to context switch time  
q usually 10ms to 100ms, context switch < 10 usec

# Time Quantum and Context Switch Time



# Turnaround Time Varies With The Time Quantum



process	time
$P_1$	6
$P_2$	3
$P_3$	1
$P_4$	7

80% of CPU bursts should be shorter than quantum

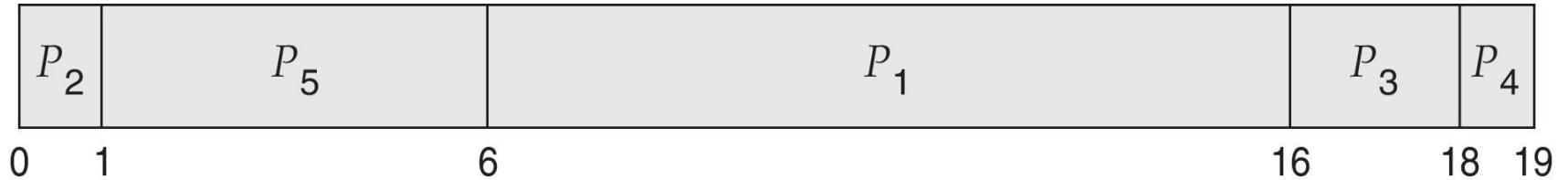
# Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive (timer interrupt, more time for more priority)
  - Nonpreemptive (no timer interrupt, just schedule process with highest priority)
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ Starvation – low priority processes may never execute
- Solution ≡ Aging – as time progresses increase the priority of the process
-

# Example of Priority Scheduling

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	3	10
$P_2$	1	1
$P_3$	2	4
$P_4$	1	5
$P_5$	5	2

- Priority scheduling Gantt Chart



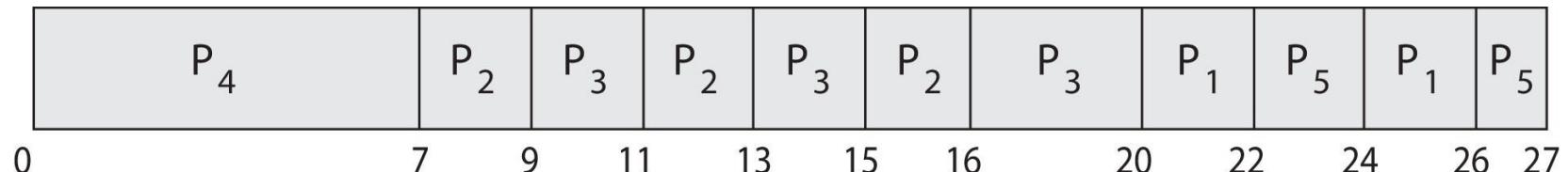
- Average waiting time = 8.2 msec

# Priority Scheduling with Round-Robin

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	4	3
$P_2$	5	2
$P_3$	8	2
$P_4$	7	1
$P_5$	3	3

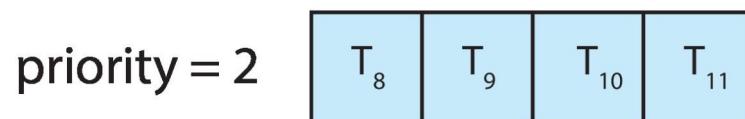
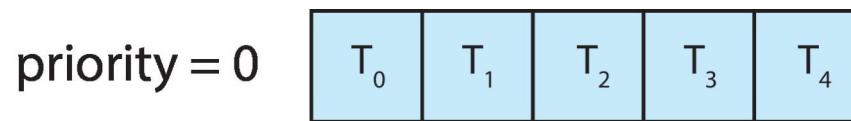
Run the process with the highest priority. Processes with the same priority run round-robin

Gantt Chart with 2 ms time quantum



# Multilevel Queue

- **With priority scheduling, have separate queues for each priority.**
- **Schedule the process in the highest-priority queue!**



•

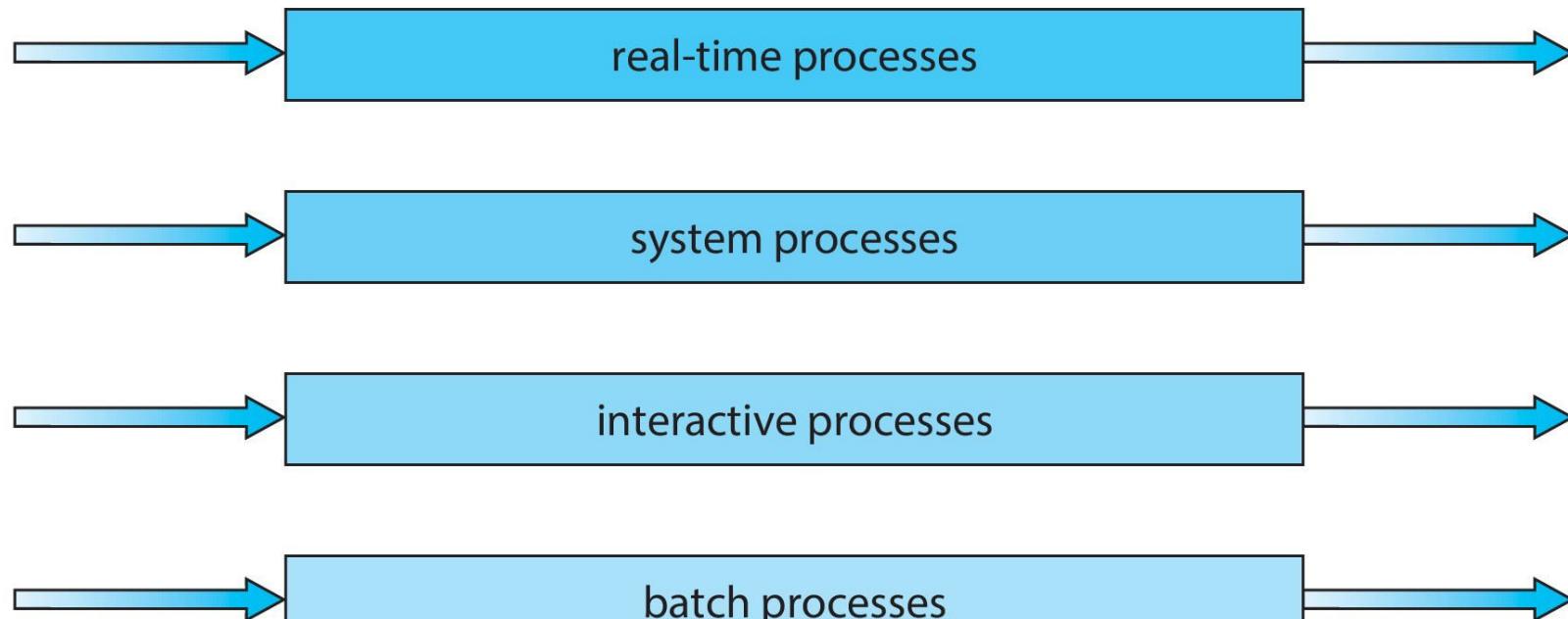
•

•



# Multilevel Queue

highest priority



lowest priority

# Implementing multilevel queue

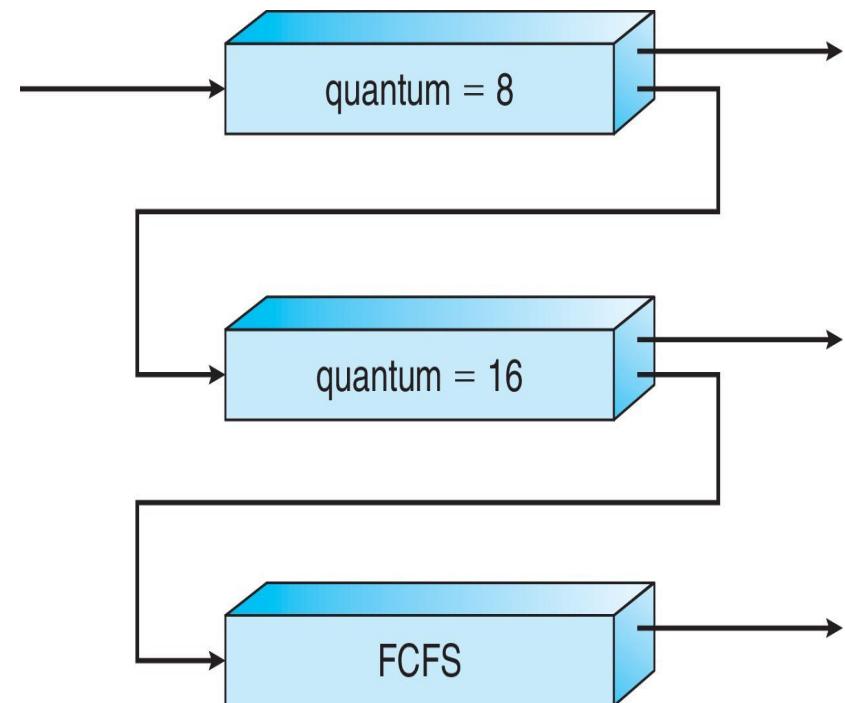
- **Processes need to have a priority**
  - Either modify fork()/exec() to have a priority
  - Or add a nice() system call to set priority
- **How to know the priority?**
  - The end user of the computer system needs to know this from needs of real life
  - E.g. on a database system, the database process will have a higher priority than other processes

# Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

# Example of Multilevel Feedback Queue

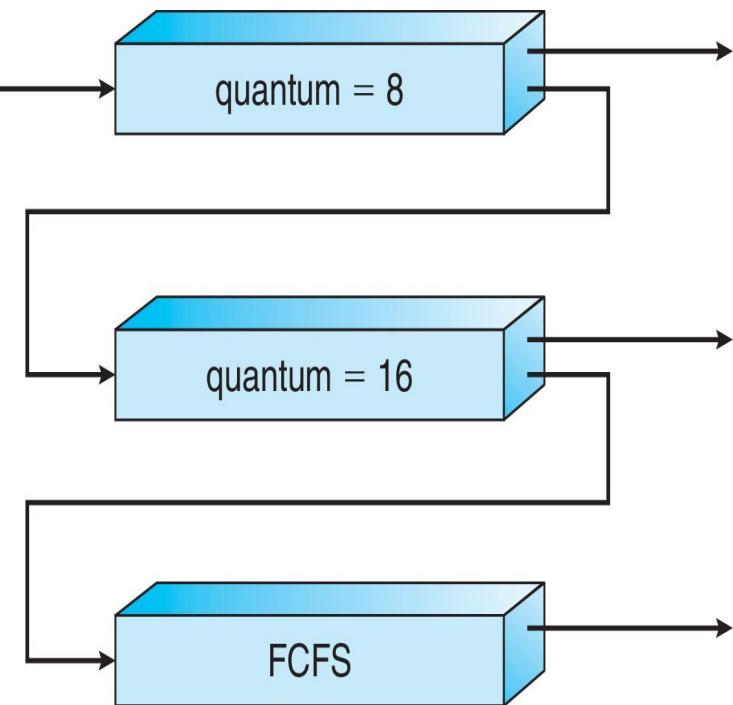
- **Three queues:**
  - $Q_0$  – RR with time quantum 8 milliseconds
  - $Q_1$  – RR time quantum 16 milliseconds
  - $Q_2$  – FCFS
- **Scheduling rules**
  - Serve all processes in  $Q_0$  first
  - Only when  $Q_0$  is empty, serve processes in  $Q_1$
  - Only when  $Q_0$  and  $Q_1$  are empty, serve processes in  $Q_2$



# Example of Multilevel Feedback Queue

- **Scheduling**

- A new job enters queue  $Q_0$ 
  - When it gains CPU, job receives 8 milliseconds
  - If it does not finish in 8 milliseconds, job is moved to queue  $Q_1$
- At  $Q_1$  job receives 16 additional milliseconds
  - If it still does not complete, it is preempted and moved to queue  $Q_2$
- To prevent starvation, move a process from lower-priority queue to higher priority queue after it has waited for too long



# Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as process-contention scope (PCS) since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is system-contention scope (SCS) – competition among all threads in system

# Pthread Scheduling

- **PTHREAD\_SCOPE\_PROCESS** schedules threads using PCS scheduling
- **PTHREAD\_SCOPE\_SYSTEM** schedules threads using SCS scheduling
- Linux and macOS only allow **PTHREAD\_SCOPE\_SYSTEM**
- Let's see a Demo using a program

# Multiple-Processor Scheduling – Load Balancing

- **Soft affinity** – the operating system attempts to keep a thread running on the same processor, but no guarantees.
- **Hard affinity** – allows a process to specify a set of processors it may run on.

**End**

# Device files Volume Manager

# Device Files

- See
  - \$ ls -l /dev
  - “c” character special device files
  - ‘b’ block device files
- Device file represents a hardware device
- Open() and then read(), write() on device files will read/write from the device (if supported)

# Device Files

- Major Number and Minor Number
  - The size field in inode is reused as major-minor number field for device files
  - Major number: type of device (identifies the device driver)
  - Minor number: device number of that type (tells the device driver, which device)
- Xv6
  - The sys\_read()->file\_read()->readi() will redirect the read operation to

```
return devsw[ip->major].read(ip, dst, n);  
where  
struct devsw {  
    int (*read) (struct inode*, char*, int);  
    int (*write) (struct inode*, char*, int);  
};
```

# Block Device Files

- For “block” devices
  - `read()`,`write()` happen in multiples of “block”
- E.g.
  - `/dev/sda1`
  - `/dev/hda1`
  - `/dev/nvme01`

# Issues with disk partitions

- Use of disk partitions

```
$ fdisk /dev/sdb # create partitions
```

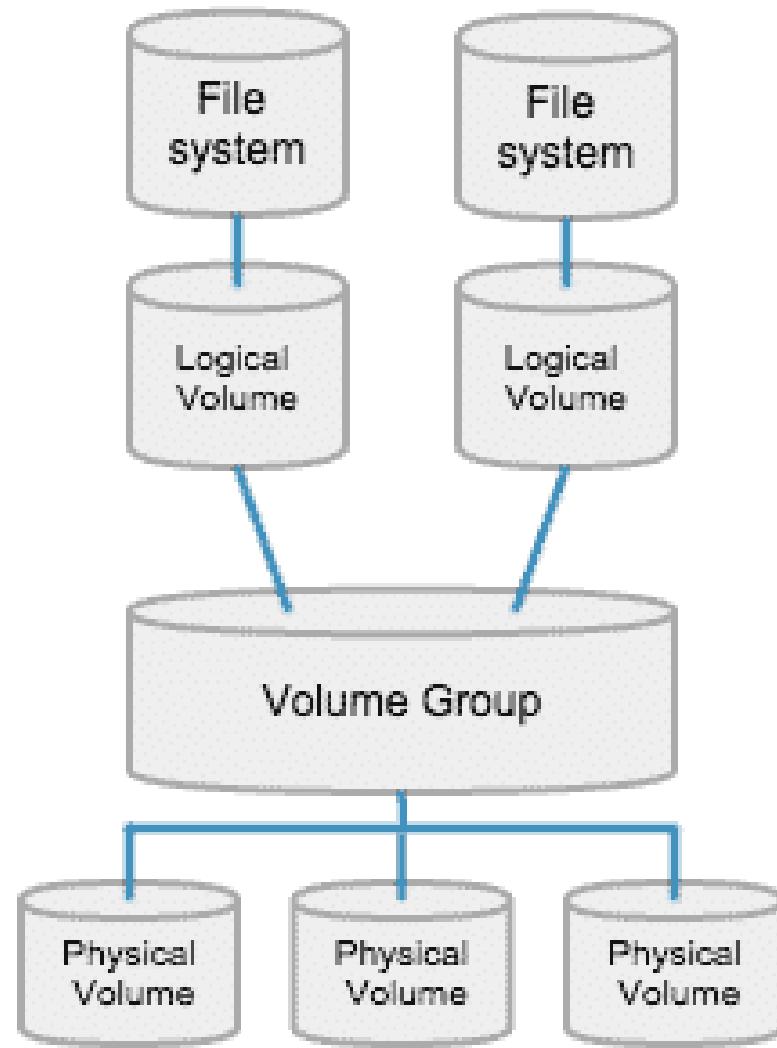
```
$ mkfs -t ext2 /dev/sdb1 # format
```

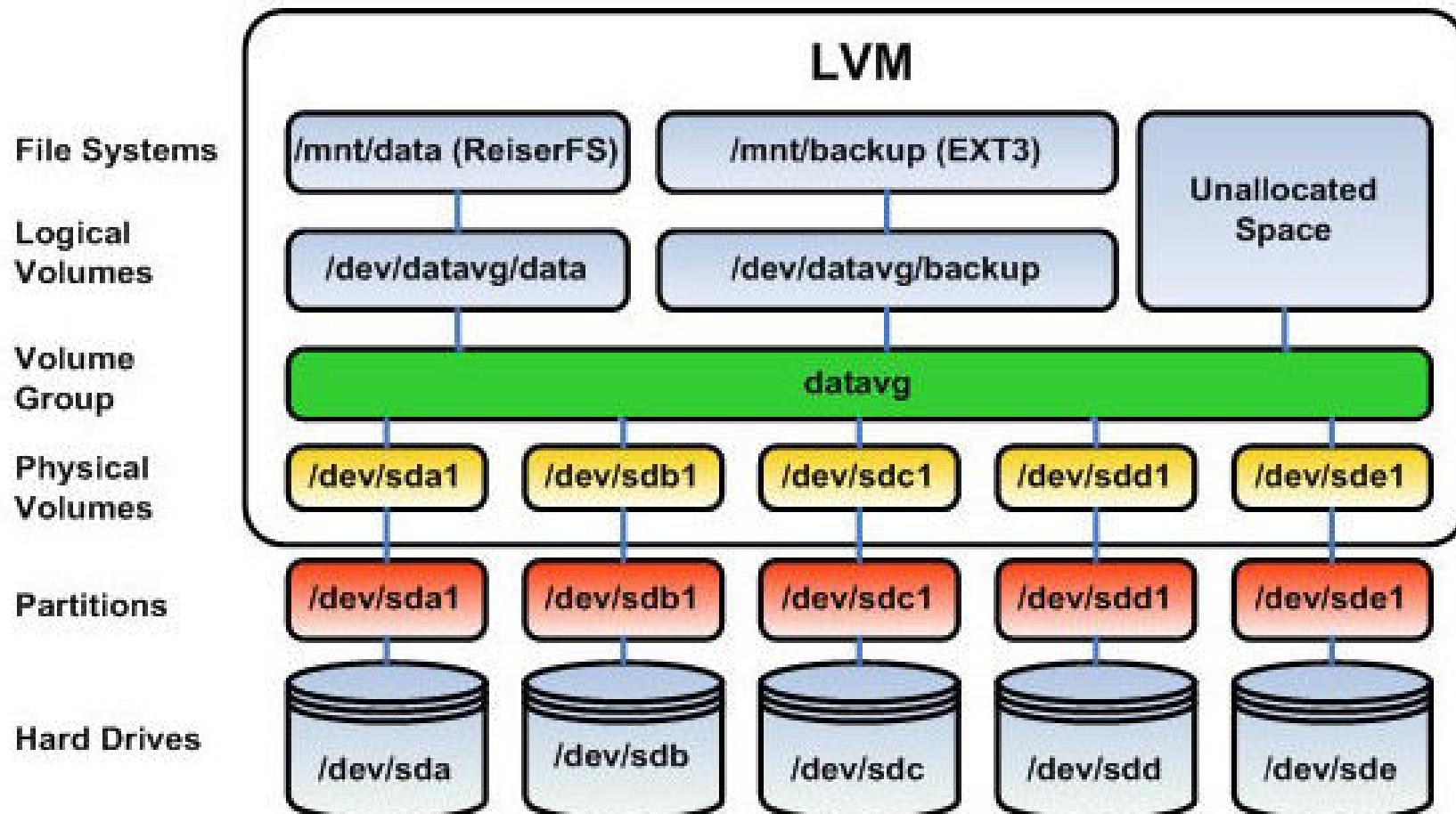
```
$ mount -t ext2 /dev/sdb1 /a/b # mounted on /a/b
```

- Fixed size
- Can't easily grow them
  - Without risk of losing file system
- Can't easily shrink them

# Logical Volume Manager **(LVM)**

- A layer (indirection) between physical partitions (physical volumes) and file system
  - Enables easy grouping , re-grouping, extending, shrinking, ...
- Logical Volume Group:
  - Parallel to a physical disk (but extendible)
- Logical Volume
  - Parallel to a physical partision (but extendible!)





# LVM

File Systems  
Sistema de arquivos

/home  
(ext4)

/data  
(xfs)

Logical Volume (LV)  
Volume Lógico

/dev/vgroup/lv\_home

/dev/vgroup/lv\_data

Volume Groups (VG)  
Grupos de volumes

vgroup

[www.linuxnaweb.com](http://www.linuxnaweb.com)

Physical Volumes (PV)  
Volumes físicos

/dev/sdb1

/dev/sdb2

/dev/sdc1

/dev/sdc2

Partitions  
Partições

/dev/sdb1  
8e LVM

/dev/sdb2  
8e LVM

/dev/sdc1  
8e LVM

/dev/sdc2  
8e LVM

Physical Drives  
Drivers físicos



/dev/sdb



/dev/sdc

# **Revision of Memory management concepts**

# Revision

- Memory layout of C program
- Address binding times: compile, link, load
- MMU Schemes
  - No MMU, Base+Relocation, Multiple Base+Relocation = Segmentation, Paging
    - Hierarchical paging
    - TLB,
  - External and Internal fragmentation
- X86 memory management
- Xv6 paging + segmentation

# **More on Linking, Loading**

# More on Linking and Loading

- Static Linking: All object code combined at link time and a big object code file is created
- Static Loading: All the code is loaded in memory at the time of exec()
- Problems
  - Static linking: Big executable files,
  - Static loading: need to load functions even if they do not execute
- Solution: Dynamic Linking and Dynamic Loading

# Dynamic Linking

- Linker is normally invoked as a part of compilation process
  - Links
    - function code to function calls
    - references to global variables with “extern” declarations
- Dynamic Linker
  - Does not combine function code with the object code file
  - Instead introduces a “stub” code that is indirect reference to actual code
  - At the time of “loading” (or executing!) the program in memory, the “link-loader” (part of OS!) will pick up the relevant code from the library machine code file (e.g. libc.so.6)

# Dynamic Linking on Linux

```
#include <stdio.h>  
  
int main() {  
    int a, b;  
    scanf("%d%d", &a, &b);  
    printf("%d %d\n", a, b);  
    return 0;  
}
```

## PLT: Procedure Linkage Table

used to call external procedures/functions whose address is to be resolved by the dynamic linker at run time.

### Output of objdump -x -D

#### Disassembly of section .text:

```
0000000000001189 <main>:  
    11d4:    callq  1080 <printf@plt>
```

#### Disassembly of section .plt.got:

```
0000000000001080 <printf@plt>:  
    1080:    endbr64  
    1084:    bnd jmpq *0x2f3d(%rip)      # 3fc8  
<printf@GLIBC_2.2.5>  
    108b:    nopl  0x0(%rax,%rax,1)
```

# Dynamic Loading

- Loader
  - Loads the program in memory
  - Part of exec() code
  - Needs to understand the format of the executable file (e.g. the ELF format)
- Dynamic Loading
  - Load a part from the ELF file only if needed during execution
  - Delayed loading
  - Needs a more sophisticated memory management by operating system – to be seen during this series of lectures

# Dynamic Linking, Loading

- Dynamic linking necessarily demands an advanced type of loader that understands dynamic linking
  - Hence called ‘link-loader’
  - Static or dynamic loading is still a choice
- Question: which of the MMU options will allow for which type of linking, loading ?

# **Virtual Memory**

# Introduction

- Virtual memory != Virtual address
  - Virtual address is address issued by CPU's execution unit, later converted by MMU to physical address
  - Virtual memory is a memory management technique employed by OS (with hardware support, of course)

# Unused parts of program

```
int a[4096][4096]
int f(int m[][4096]) {
    int i, j;
    for(i = 0; i < 1024; i++)
        m[0][i] = 200;
}
int main() {
    int i, j;
    for(i = 0; i < 1024; i++)
        a[1][i] = 200;
    if(random() == 10)
        f(a);
}
```

All parts of array a[] not accessed

Function f() may not be called

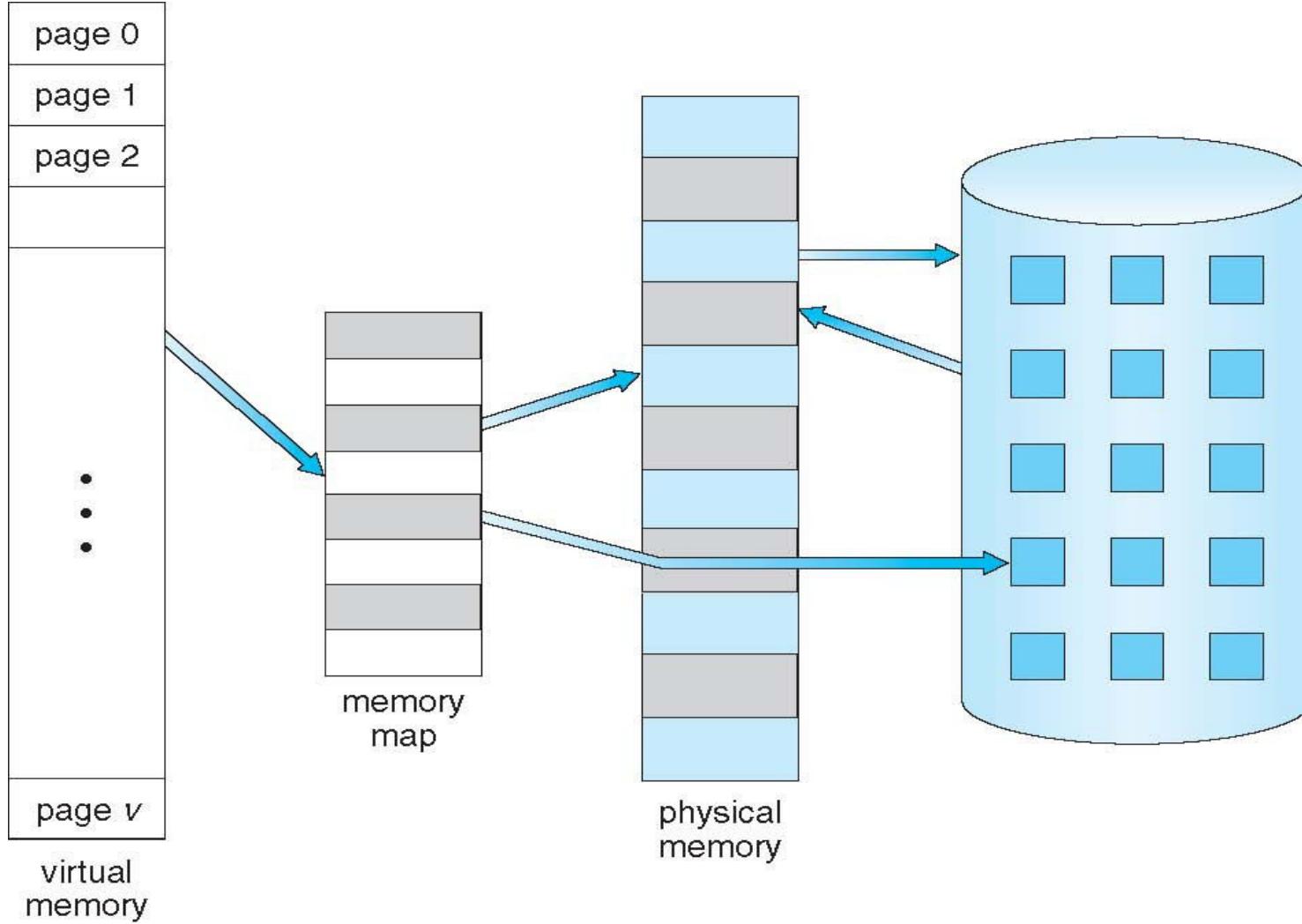
# **Some problems with schemes discussed so far**

- Code needs to be in memory to execute, But entire program rarely used
  - Error code, unusual routines, large data structures are rarely used
- So, entire program code, data not needed at same time
- So, consider ability to execute partially-loaded program
  - One Program no longer constrained by limits of physical memory
  - One Program and collection of programs could be larger than physical memory

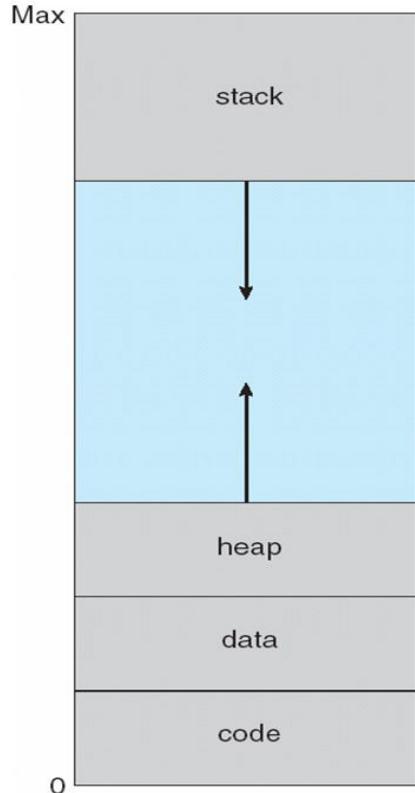
# What is virtual memory?

- Virtual memory – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

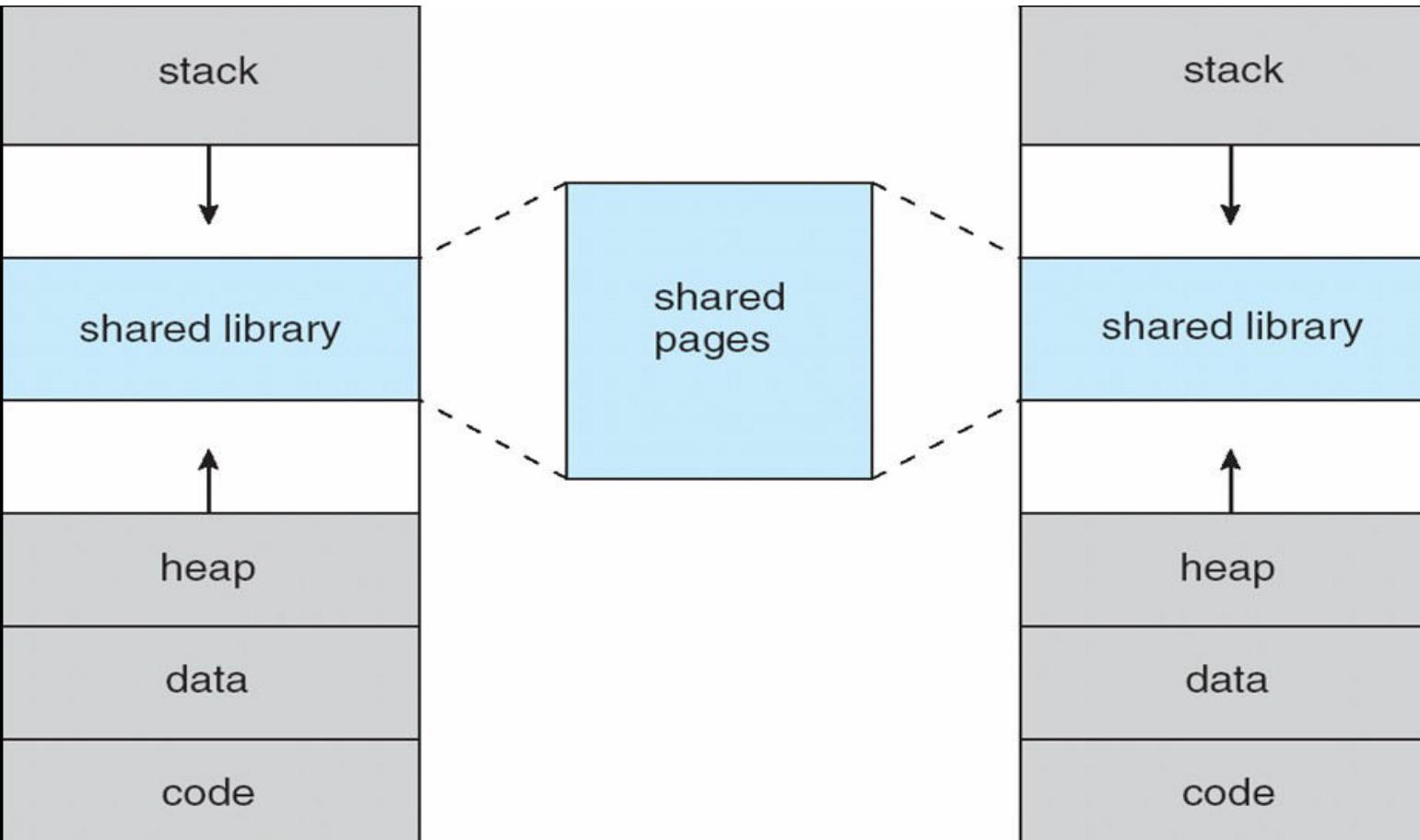
Virtual Memory  
Larger  
Than  
Physical  
Memory



# Virtual Address space



Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc



## Shared pages Using Virtual Memory

System libraries shared via mapping into virtual address space

Shared memory by mapping same page-frames into page tables of involved processes

Pages can be shared during `fork()`, speeding process creation (more later)

# **Demand Paging**

# Demand Paging

- Load a “page” to memory when it’s neded (on demand)
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
-

# Demand Paging

- Options:
  - Load entire process at load time : achieves little
  - Load some pages at load time: good
  - Load no pages at load time: pure demand paging

# New meaning for valid/invalid bits in page

Frame #	valid-invalid bit
	v
	v
	v
	v
....	i
	i
	i

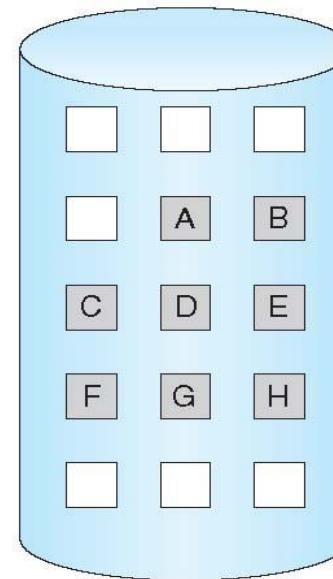
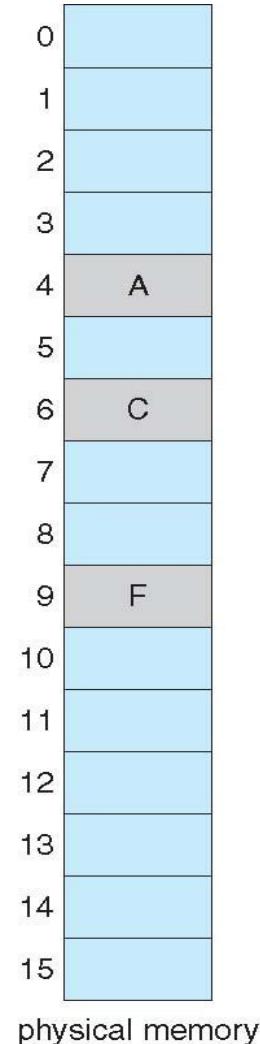
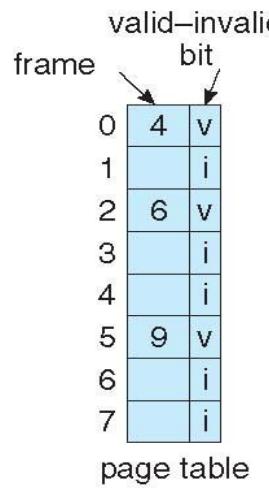
## table

With each page table entry a valid–invalid bit is associated

- v: in-memory – memory resident
- i : not-in-memory or illegal
- During address translation, if valid–invalid bit in page table entry is I : **raises trap called page fault**

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

logical memory



**Page**  
**Table**  
**With**  
**Some pages**  
**Not**  
**In memory**

# **Page fault**

# Page fault

- Page fault is a hardware interrupt
- It occurs when the page table entry corresponding to current memory access is “i”
- All actions that a kernel takes on a hardware interrupt are taken!
  - Change of stack to kernel stack
  - Saving the context of process
  - Switching to kernel code

# Important (not all) steps on a Page fault

1) Operating system looks at another data structure (table), most likely in PCB itself, to decide:

If it's Invalid reference -> abort the process (segfault)

Just not in memory -> Need to get the page in memory

2) Get empty frame (this may be complicated, may need evicting a frame if no free frames available!)

3) Swap page into frame via scheduled disk/IO operation

4) Reset tables to indicate page now in memory.

5) Set validation bit = v

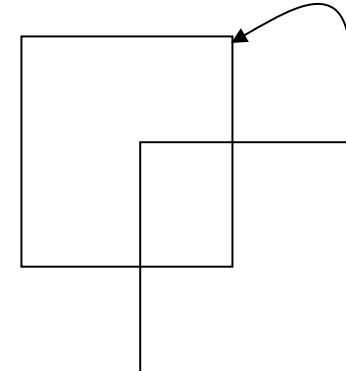
6) Restart the instruction that caused the page fault

# Issues with page fault handling

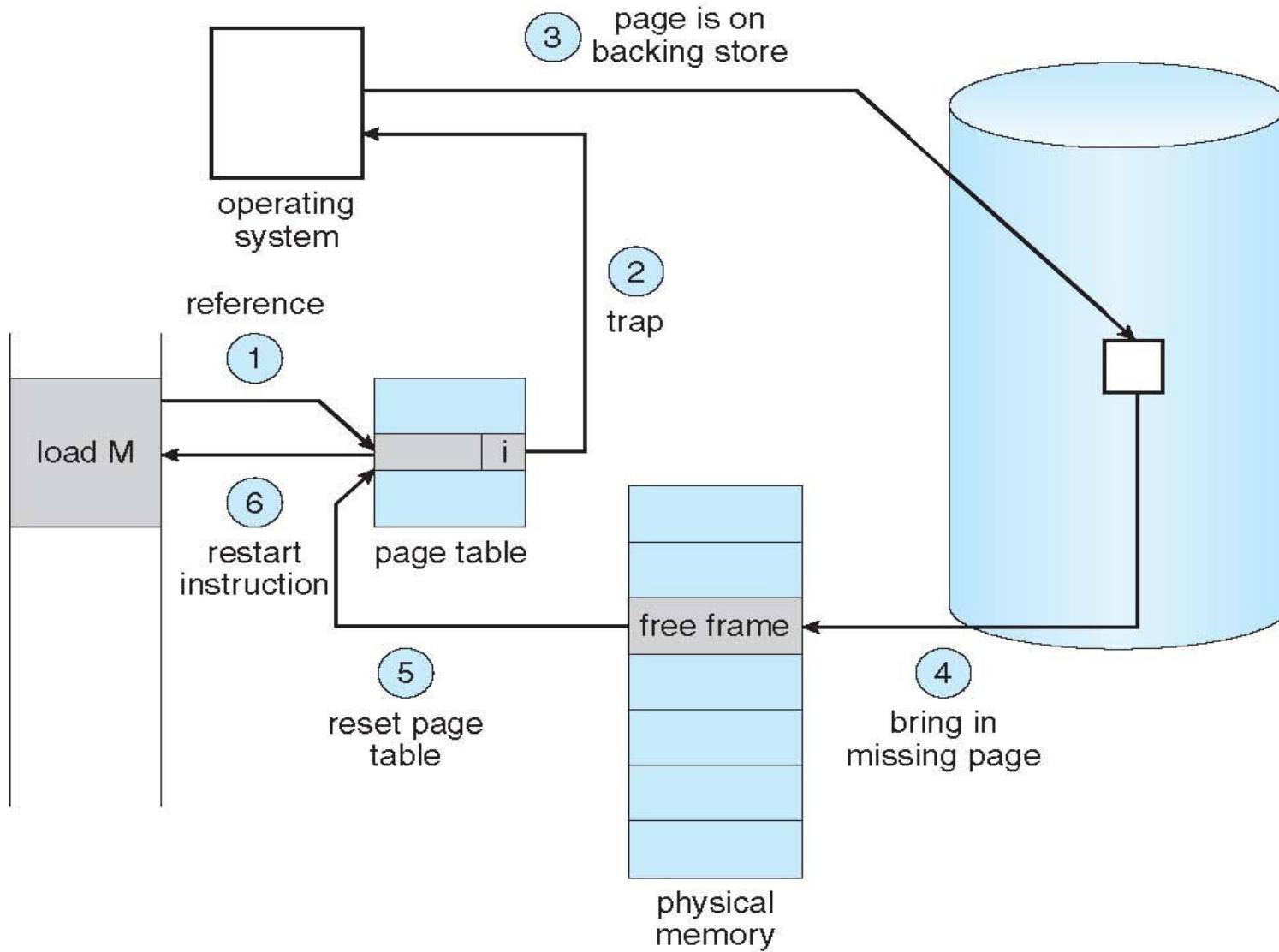
- Extreme case – start process with *no pages in memory*
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging. Less response time ?**
- Actually, a given instruction could access multiple pages -> multiple page faults
  - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

# Problem with Instruction restart

- A critical Problem
- Consider an instruction that could access several different locations
  - movarray 0x100, 0x200, 20
  - # copy 20 bytes from address 0x100 to address 0x200
  - movarray 0x100, 0x110, 20
  - # what to do in this case?



# Handling A Page Fault



# Page fault handling (detailed)

1)Trap to the operating system

2)Default trap handling():

Save the process registers and process state

Determine that the interrupt was a page fault. Run page fault handler.

3)Page fault handler(): Check that the page reference was legal and determine the location of the page on the disk. If illegal, terminate process.

4)Find a free frame. Issue a read from the disk to a free frame:

Process waits in a queue for disk read. Meanwhile many processes may get scheduled.

Disk DMA hardware transfers data to the free frame and raises interrupt in end

# Page fault handling (detailed)

- 6) (as said on last slide) While waiting, allocate the CPU to some other process
- 7) (as said on last slide) Receive an interrupt from the disk I/O subsystem (I/O completed)
- 8) Default interrupt handling():
  - Save the registers and process state for the other user
  - Determine that the interrupt was from the disk
- 9) Disk interrupt handler():
  - Figure out that the interrupt was for our waiting process
  - Make the process runnable
- 10) Wait for the CPU to be allocated to this process again
  - Kernel restores the page table of the process, marks entry as “v”
  - Restore the user registers, process state, and new page table, and then resume the interrupted instruction

# Performance of demand paging

**Page Fault Rate  $0 \leq p \leq 1$**

if  $p = 0$  no page faults

if  $p = 1$ , every reference is a fault

**Effective (memory) Access Time (EAT)**

$EAT = (1 - p) * \text{memory access time} +$

$p * (\text{page fault overhead} // \text{Kernel code execution time})$

$+ \text{swap page out} // \text{time to write an occupied frame to disk}$

$+ \text{swap page in} // \text{time to read data from disk into free frame}$

$+ \text{restart overhead}) // \text{time to reset process context, restart it}$

# Performance of demand paging

Memory access time = 200 nanoseconds

Average page-fault service time = 8 milliseconds

$$EAT = (1 - p) \times 200 + p \text{ (8 milliseconds)}$$

$$\begin{aligned} &= (1 - p) \times 200 + p \times 8,000,000 \\ &= 200 + p \times 7,999,800 \end{aligned}$$

If one access out of 1,000 causes a page fault, then

$$EAT = 8.2 \text{ microseconds.}$$

This is a slowdown by a factor of 40!!

If want performance degradation < 10 percent

$$220 > 200 + 7,999,800 \times p$$

$$20 > 7,999,800 \times p$$

$$p < .0000025$$

< one page fault in every 400,000 memory accesses

# An optimization: Copy on write

## The problem with `fork()` and `exec()`.

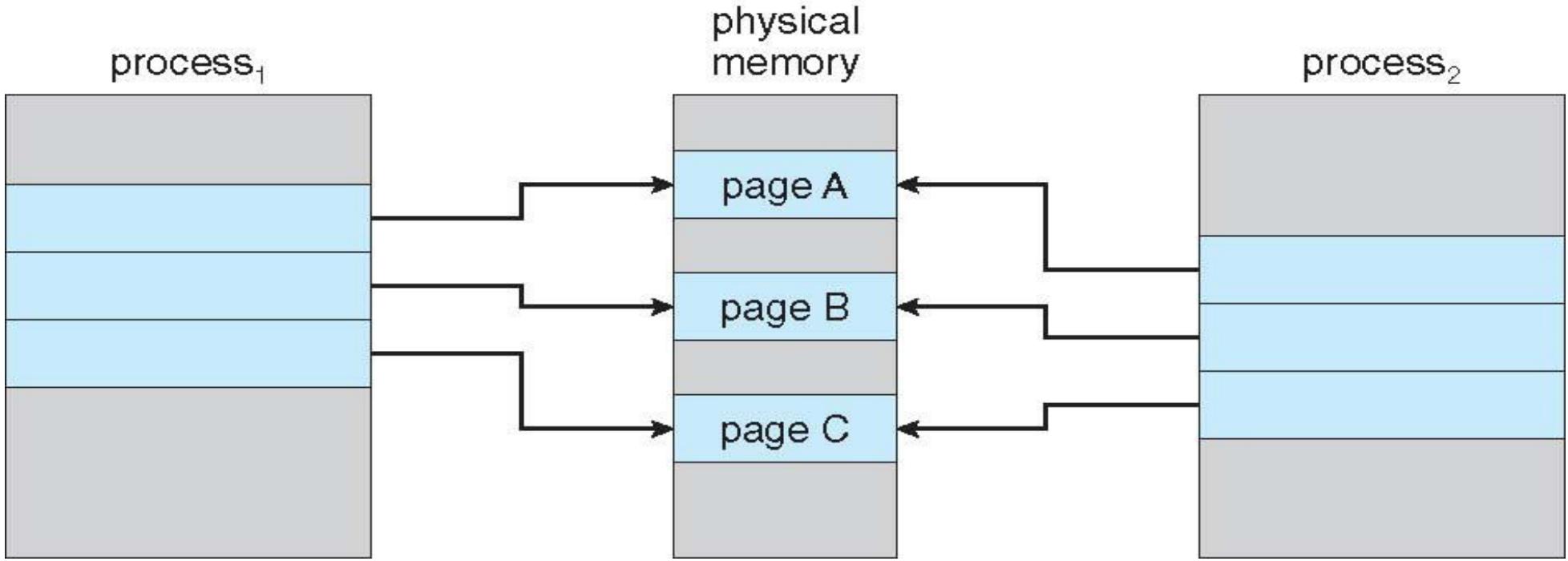
### Consider the case of a shell

```
scanf("%s", cmd);
if(strcmp(cmd, "exit") == 0)
    return 0;
pid = fork(); // A->B
if(pid == 0) {
    ret = execl(cmd, cmd, NULL);
    if(ret == -1) {
        perror("execution
failed");
        exit(errno);
    }
} else {
    wait(0);
}
```

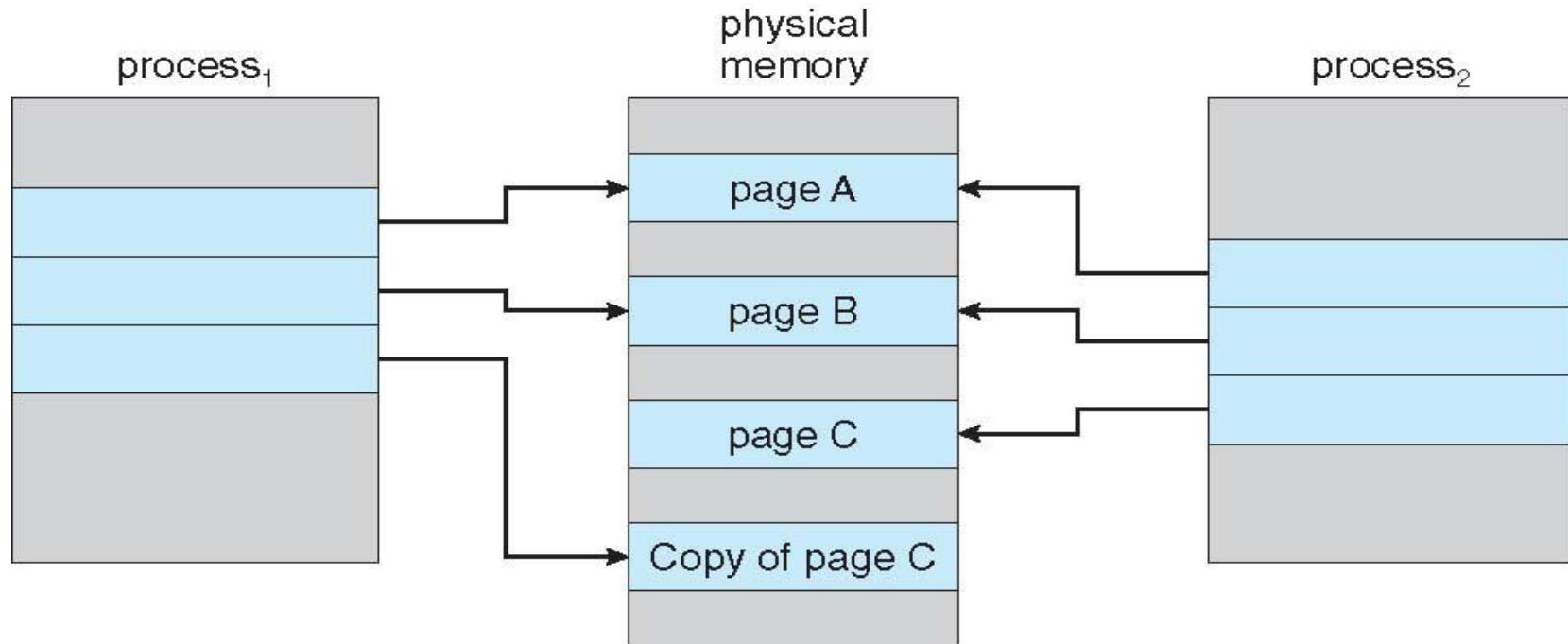
- During `fork()`
  - Pages of parent were duplicated
  - Equal amount of page frames were allocated
  - Page table for child differed from parent (as it has another set of frames)
- In `exec()`
  - The page frames of child were taken away and new frames were allocated
  - Child's page table was rebuilt!
  - Waste of time during `fork()` if the `exec()` was to be called immediately

# An optimization: Copy on write

- Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- vfork() variation on fork() system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call exec()
  - Very efficient



# Before Process 1 Modifies Page C



# After Process 1 Modifies Page C

# Challenges and improvements in implementation

- Choice of backing store
  - For stack, heap pages: on swap partition
  - For code, shared library? : swap partition or the actual executable file on the file-system?
  - If the choice is file-system for code, then the page-fault handler needs to call functions related to file-system
- Is the page table itself pageable?
  - If no, good
  - If Yes, then there can be page faults in accessing the page tables themselves! More complicated!
- Is the kernel code pageable?
  - If no, good
  - If yes, life is very complicated ! Page fault in running kernel code, interrupt handlers, system calls, etc.

# **Page replacement**

# Review

- Concept of virtual memory, demand paging.
- Page fault
- Performance degradation due to page fault: Need to reduce #page faults to a minimum
- Page fault handling process, broad steps: (1) Trap (2) Locate on disk (3) **find free frame** (4) schedule disk I/O (5) update page table (6) resume
- More on (3) today

# List of free frames

- Kernel needs to maintain a list of free frames
- At the time of loading the kernel, the list is created (kinit1, knit2 in xv6)
- Frames are used for allocating memory to a process
  - But may also be used for managing kernel's own data structures also (in xv6, all kernel data is statically allocated)
- More processes --> more demand for frames

# **What if no free frame found on page fault?**

- Page frames in use depends on “Degree of multiprogramming”
  - More multiprogramming -> overallocation of frames
  - Also in demand from the kernel, I/O buffers, etc
  - How much to allocate to each process? How many processes to allow?
- Page replacement – find some page(frame) in memory, but not really in use, page it out
  - Questions : terminate process? Page out process? replace the page?
  - For performance, need an algorithm which will result in minimum number of page faults
- Bad choices may result in same page being brought into memory several times

# Need for Page Replacement

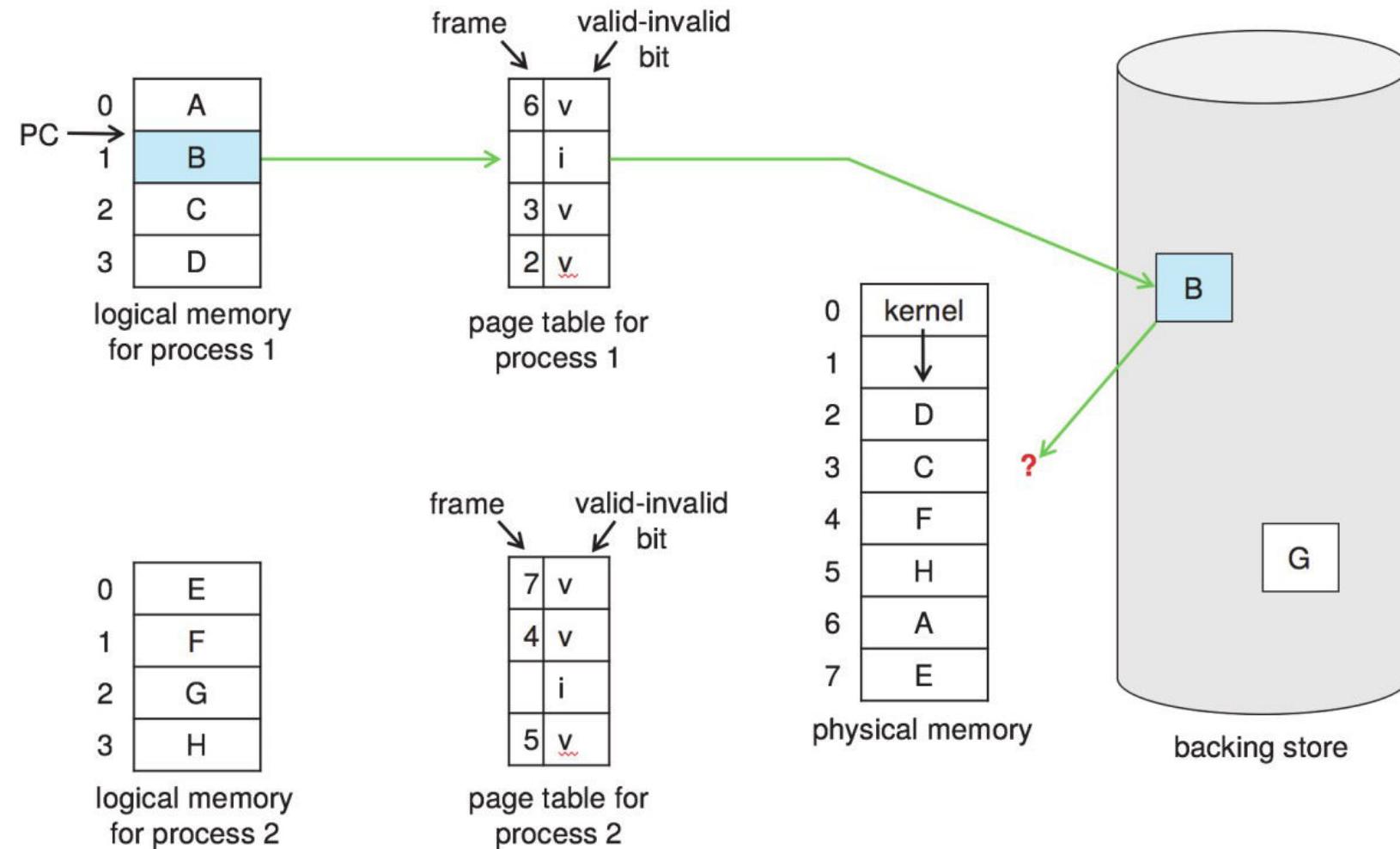


Figure 10.9 Need for page replacement.

# Page replacement

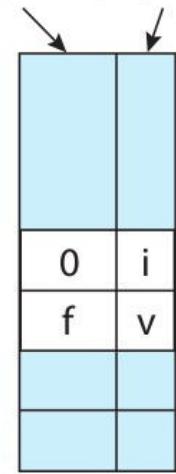
- Strategies for performance
  - Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
  - Use modify (dirty) bit in page table. To reduce overhead of page transfers – only modified pages are written to disk. If page is not modified, just reuse it (a copy is already there in backing store)

# Basic Page replacement

- 1) Find the location of the desired page on disk
- 2) Find a free frame:
  - If there is a free frame, use it
  - If there is no free frame, use a page replacement algorithm to select a victim frame & write victim frame to disk if dirty
- 3) Bring the desired page into the free frame; update the page table of process and global frame table/list
- 4) Continue the process by restarting the instruction that caused the trap
  - Note now potentially 2 page transfers for page fault – increasing EAT

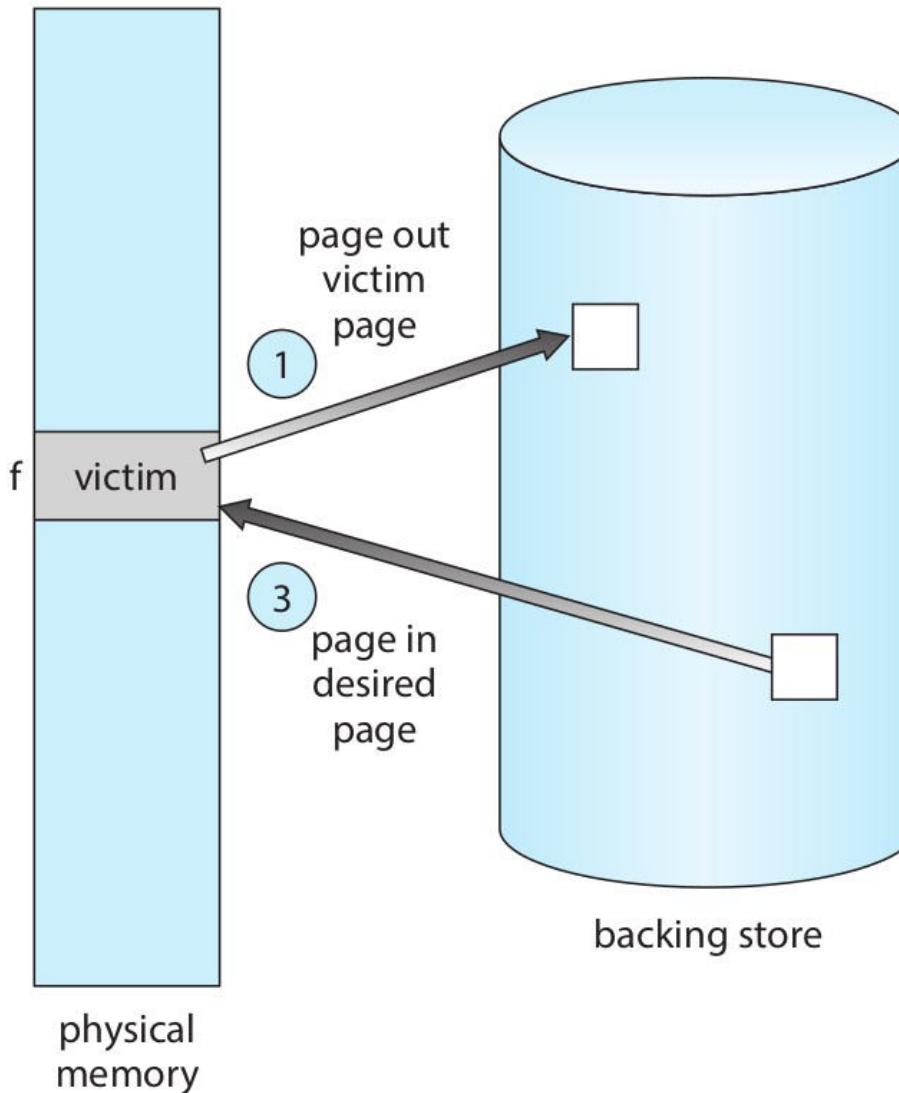
# Page Replacement

frame      valid-invalid bit



2 change to invalid

4 reset page table for new page

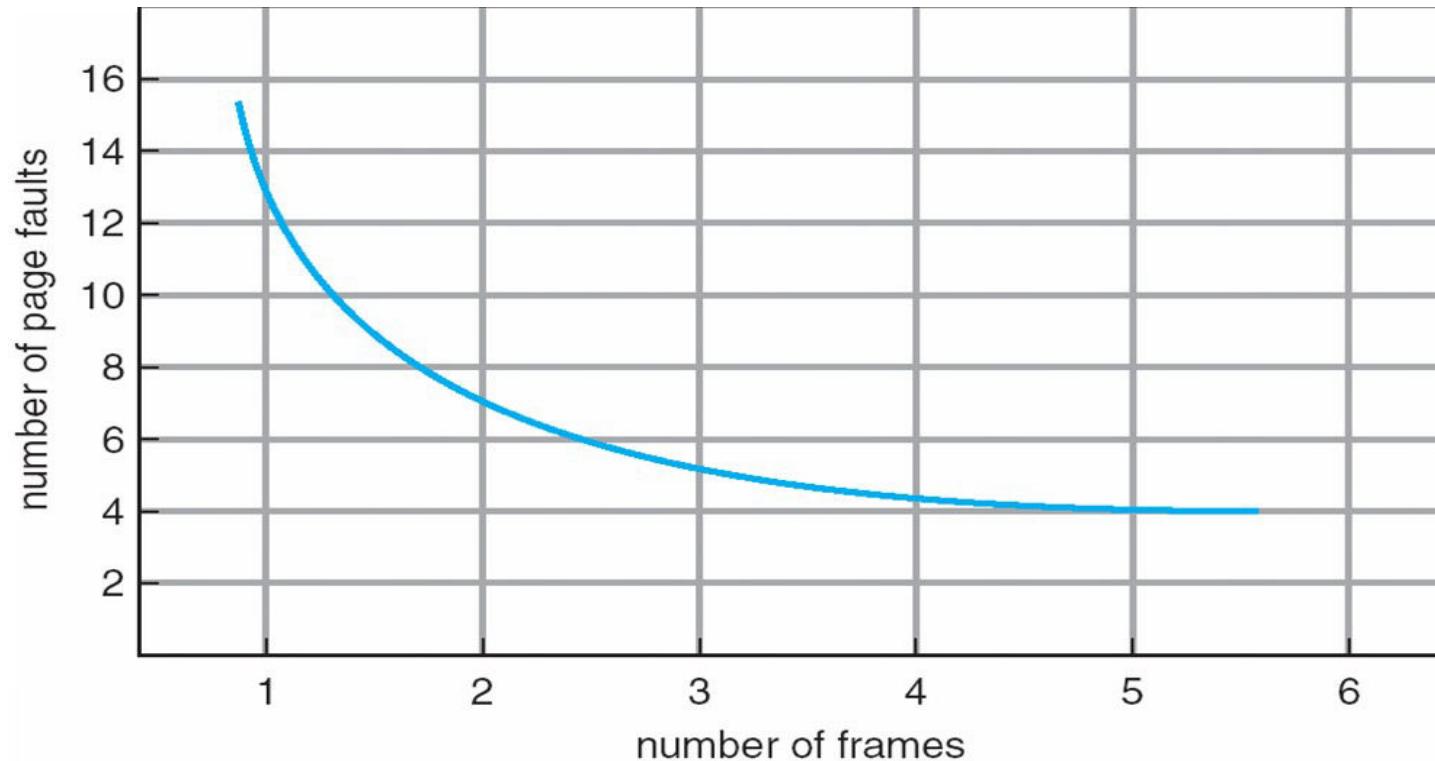


# Two problems to solve

- Frame-allocation algorithm determines
  - How many frames to give each process
  - Which frames to replace
- Page-replacement algorithm
  - Want lowest page-fault rate on both first access and re-access

# Evaluating algorithm: Reference string

- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just *page numbers*, not full addresses
  - Repeated access to the same page does not cause a page fault
  - In all our examples, the reference string is 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1



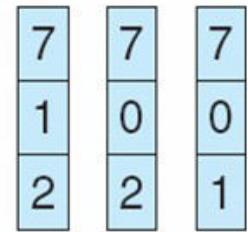
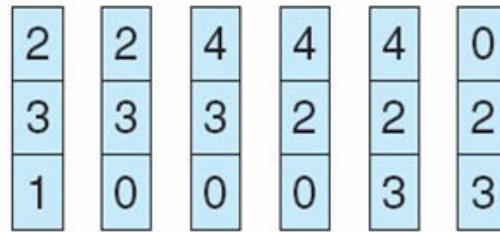
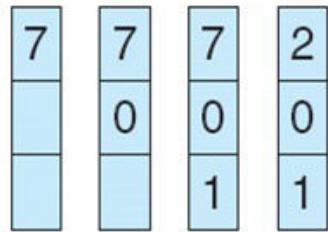
**An  
Expectation**

**More page  
Frames  
Means less  
faults**

# FIFO Algorithm

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

15 page faults

# FIFO Algorithm

1	5	1 4 5
2	3	2 1 3
3	4	3 2 4

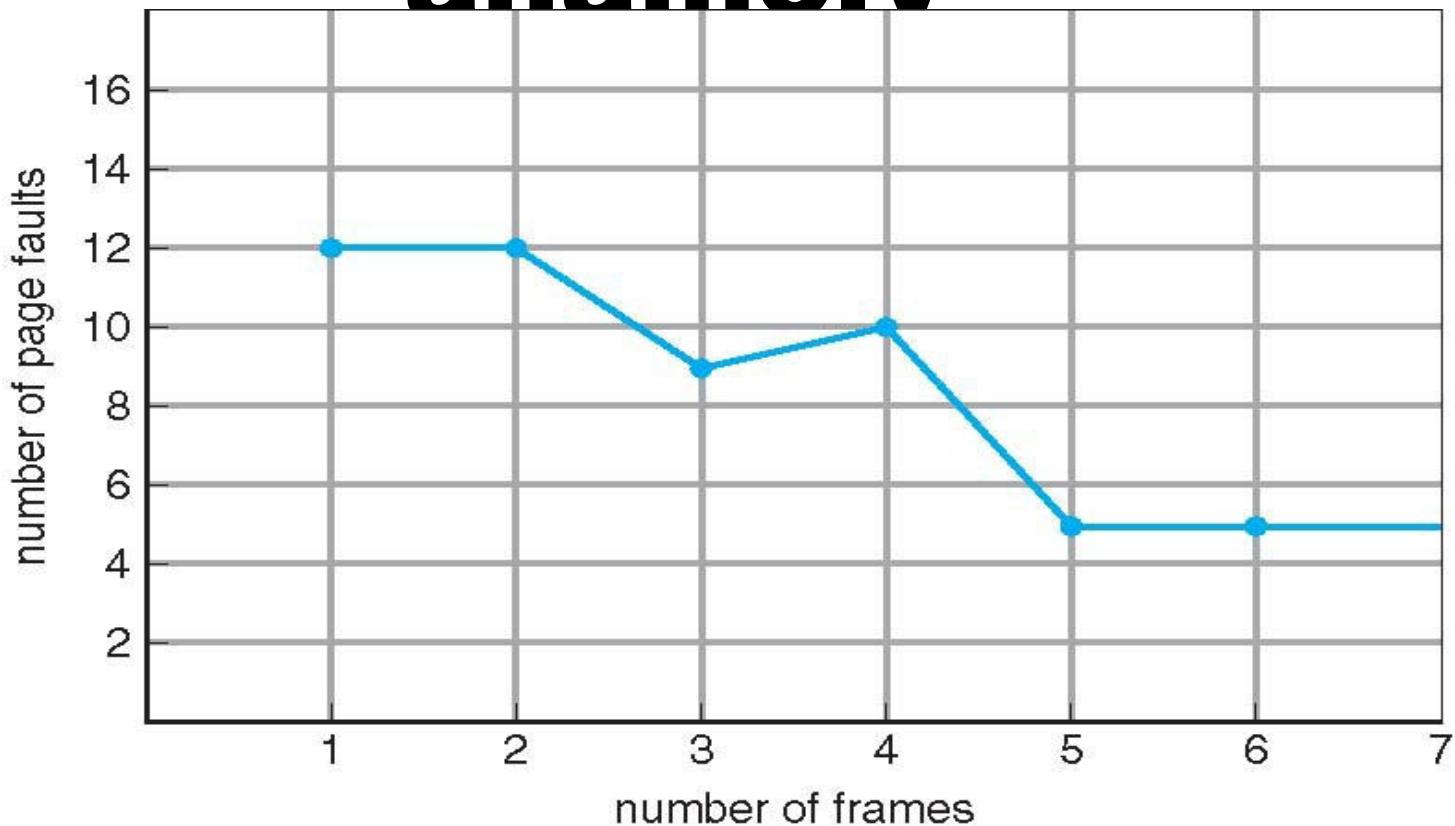
9 PF

1	4	1 5 4
2	5	2 1 5
3	2	3 2
4	3	4 3

10 PF

- **Belady's Anomaly**
- Adding more frames can cause more page faults!
  - Can vary by reference string: consider  
1,2,3,4,1,2,5,1,2,3,4,5

# FIFO Algorithm: Balady's anamolv



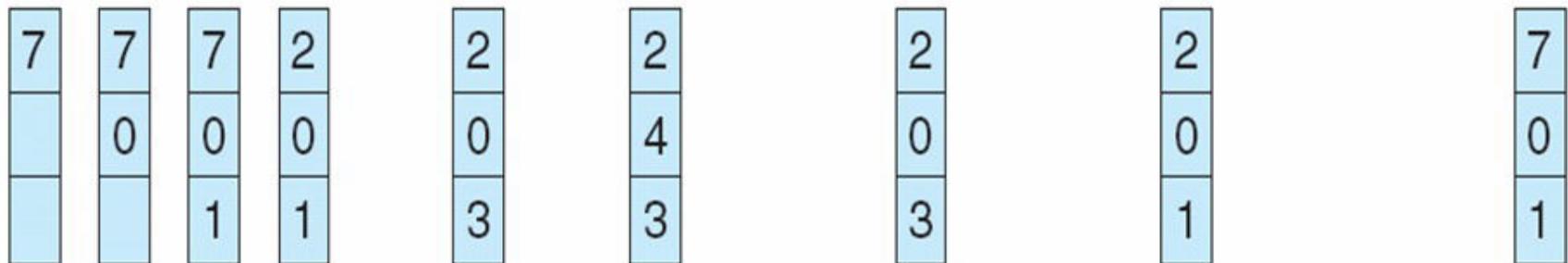
# Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal #replacements for the example on the next slide
  - How do you know this?
    - Can't read the future
  - Used for measuring how well your algorithm performs

# Optimal page replacement

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

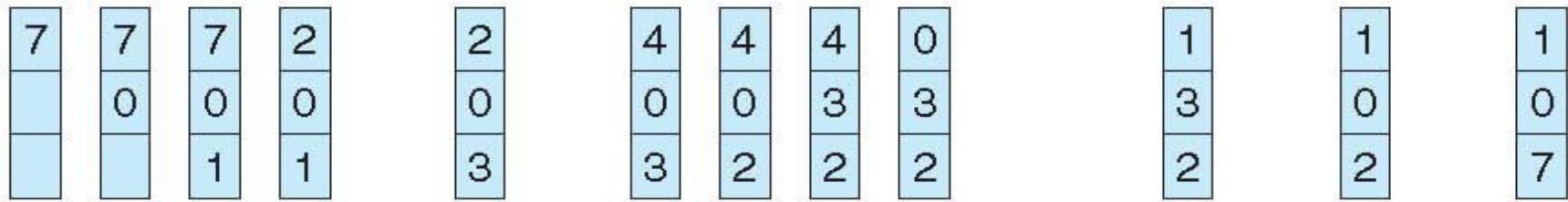


page frames

# Least Recently Used: an approximation of optimal

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

Use past knowledge rather than future

Replace page that has not been used in the most amount of time

Associate time of last use with each page

12 faults – better than FIFO but worse than OPT

Generally good algorithm and frequently used

But how to implement?

# LRU: Counter implementation

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed

# LRU: Stack implementation

- Keep a stack of page numbers in a double link form:
- Page referenced: move it to the top
  - requires 6 pointers to be changed and
  - each update more expensive
  - But no need of a search for replacement

reference string

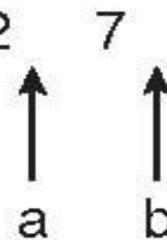
4 7 0 7 1 0 1 2 1 2 7 1 2

2
1
0
7
4

stack  
before  
a

7
2
1
0
4

stack  
after  
b



**Use of a  
Stack to  
Record  
The  
Most  
Recent  
Page**

# Stack algorithms

- An algorithm for which it can be shown that the set of pages in memory for  $n$  frames is always a subset of the set of pages that would be in memory with  $n + 1$  frames
- Do not suffer from Balady's anomaly
- For example: Optimal, LRU

# **LRU: Approximation**

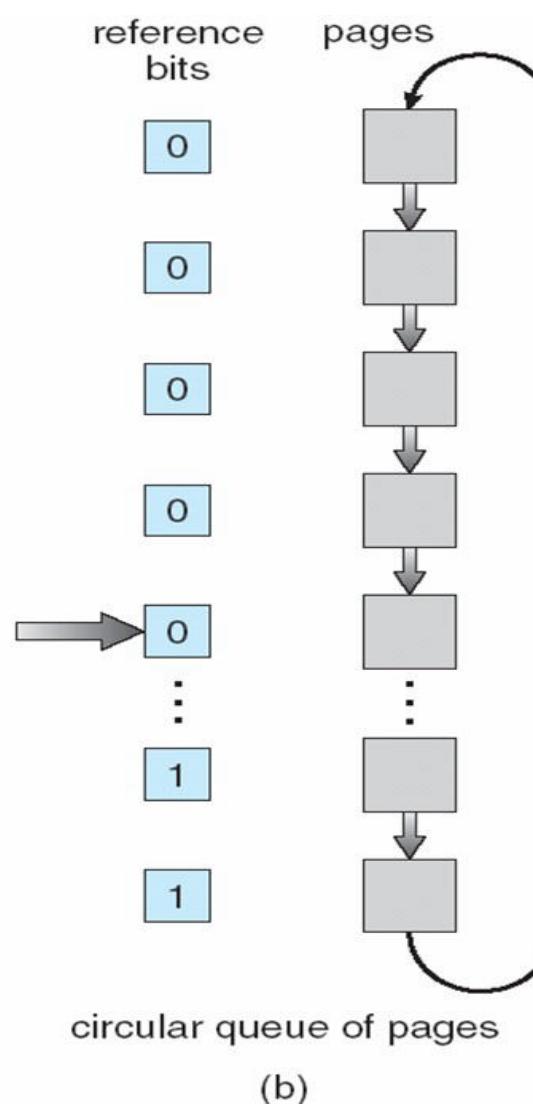
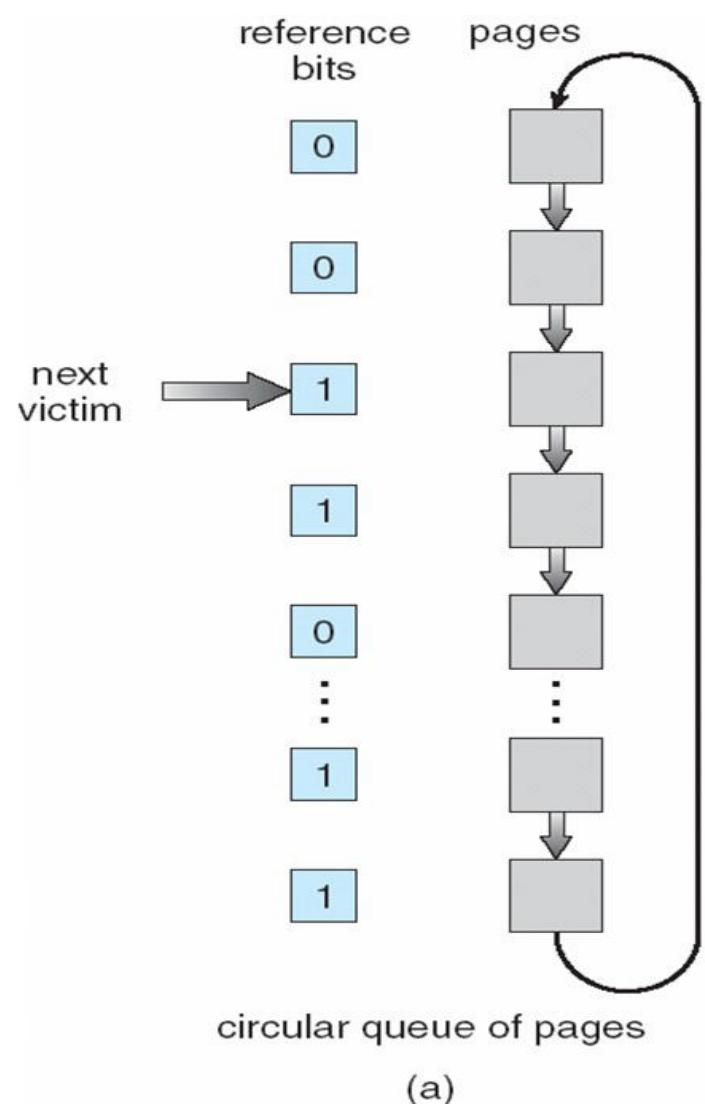
## **algorithms**

- LRU needs special hardware and still slow
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
  - We do not know the order, however

# LRU: Approximation algorithms

- Second-chance algorithm
  - FIFO + hardware-provided reference bit. If bit is 0 select, if bit is 1, then set it to 0 and move to next one.
- An implementation of second-chance: Clock replacement
  - If page to be replaced has
  - Reference bit = 0 -> replace it
  - reference bit = 1 then:
    - set reference bit 0, leave page in memory
    - replace next page, subject to same rules

# Second- Chance (clock) Page- Replacement Algorithm



# Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common
  - **LFU Algorithm**: replaces page with smallest count
  - **MFU Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

# Page buffering algorithms

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

# Major and Minor page faults

- Most modern OS refer to these two types
- Major fault
  - Fault + page not in memory
- Minor fault
  - Fault, but page is in memory
  - For example shared memory pages; second instance of fork(), page already on free-frame list,
- On Linux run

```
$ ps -eo min_flt,maj_flt,cmd
```

# Special rules for special applications

- All of earlier algorithms have OS guessing about future page access
- But some applications have better knowledge – e.g. databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can give direct access to the disk, getting out of the way of the applications
  - Raw disk mode
  - Bypasses buffering, locking, etc

# Allocation of frames

- Each process needs *minimum* number of frames
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- Maximum of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

# Fixed allocation of frames

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool
- Proportional allocation – Allocate according to the size of process

$s_i$  = size of process  $i$

$$S = \sum s_i$$

$m$  = total number of frames

$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

$$\frac{s_1}{S} = \frac{10}{137}$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

# Priority Allocation of frames

- Use a proportional allocation scheme using priorities rather than size
- If process  $P_i$  generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

# Global Vs Local allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput so more common
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

# **Virtual Memory – Remaining topics**

# Agenda

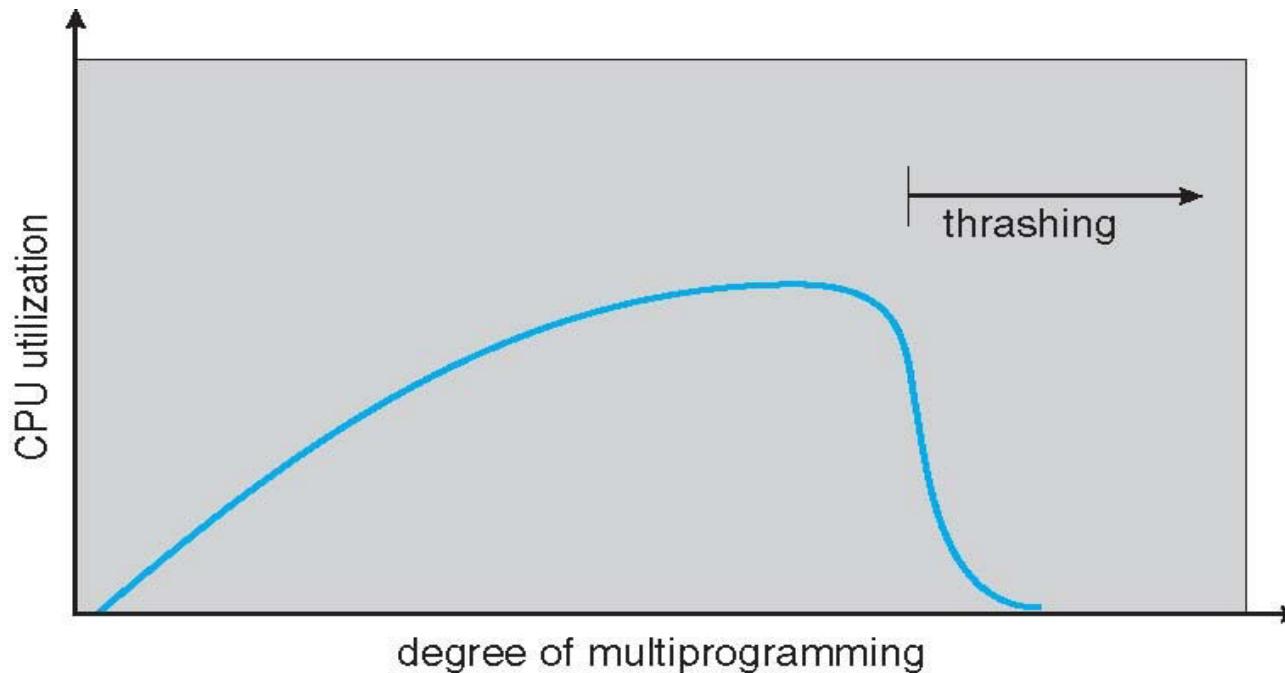
- Problem of Thrashing and possible solutions
- Mmap(), Memory mapped files
- Kernel Memory Management
- Other Considerations

# Thrashing

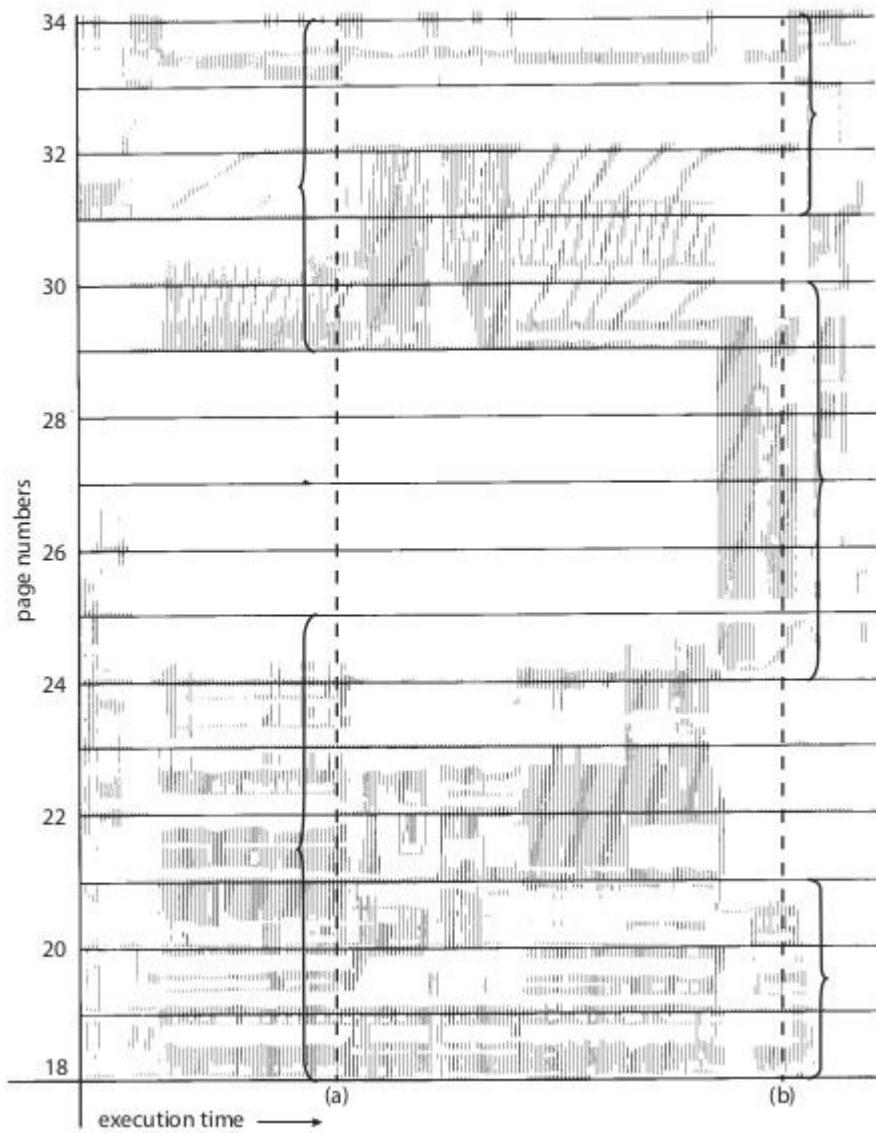
# Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
- This leads to:
  - Low CPU utilization
  - Operating system thinking that it needs to increase the degree of multiprogramming
  - Another process added to the system
- Thrashing : a process is busy swapping pages in and out

# Thrashing



# Locality In A Memory-Reference Pattern



# Demand paging and thrashing

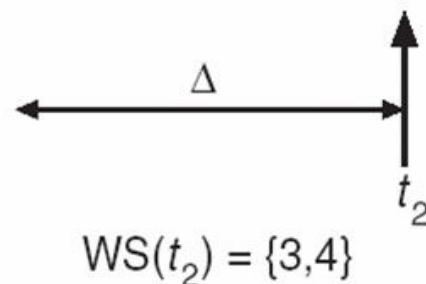
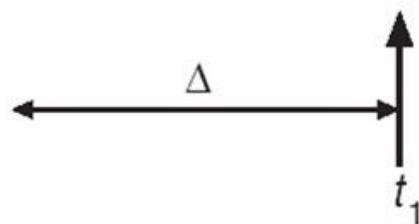
- Why does demand paging work?
  - Locality model
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?
  - size of locality > total memory size
  - Limit effects by using local or priority page replacement

# Working set model

- $\Delta \equiv$  working-set window  $\equiv$  a fixed number of page references
  - Example: 10,000 instructions
- Working Set Size,  $WSS_i$  (working set of Process  $P_i$ ) =
  - total number of pages referenced in the most recent  $\Delta$  (varies in time)
  - if  $\Delta$  too small will not encompass entire locality
  - if  $\Delta$  too large will encompass several localities
  - if  $\Delta = \infty \Rightarrow$  will encompass entire program

page reference table

... 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...

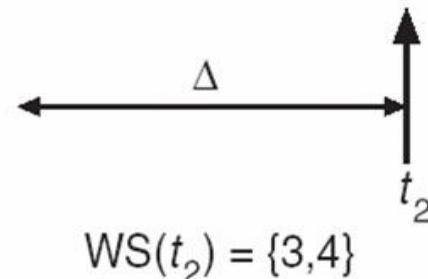
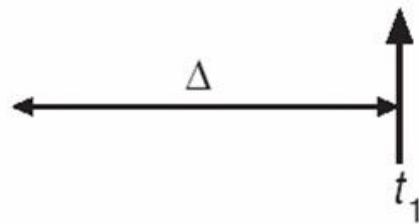


# Working set model

- $D = \sum WSS_i \equiv$  total demand frames
  - Approximation of locality
  - if  $D > m$  (total available frames)  $\Rightarrow$  Thrashing
  - Policy if  $D > m$ , then suspend or swap out one of the processes

page reference table

... 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...



# Keeping Track of the Working Set

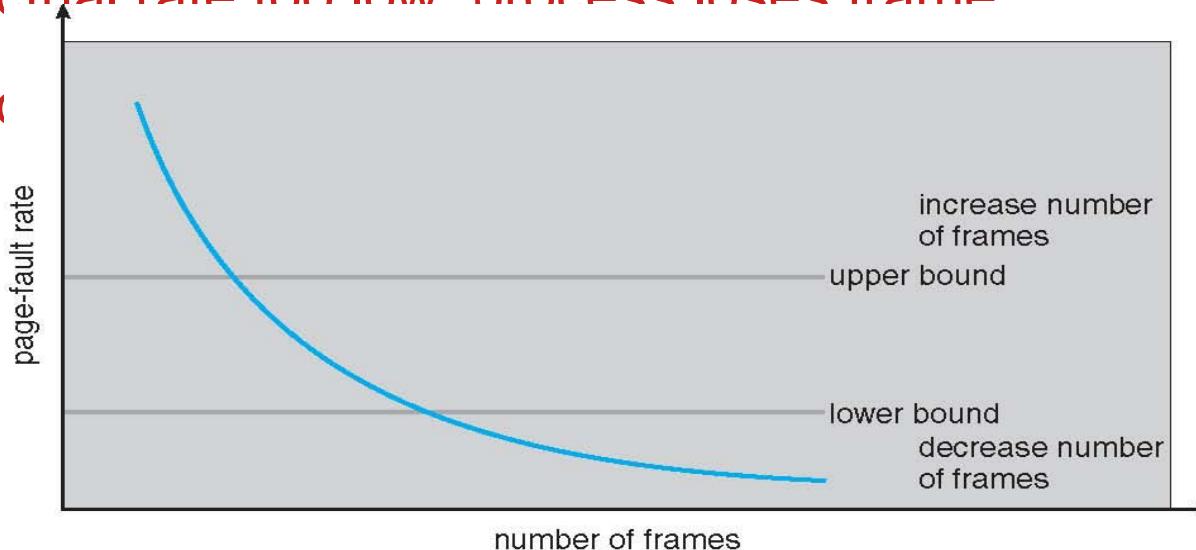
- Approximate with interval timer + a reference bit
- Example:  $\Delta = 10,000$ 
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupt occurs copy (to memory) and sets the values of all reference bits to 0
  - If one of the bits in memory = 1  $\Rightarrow$  page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

# Page fault frequency

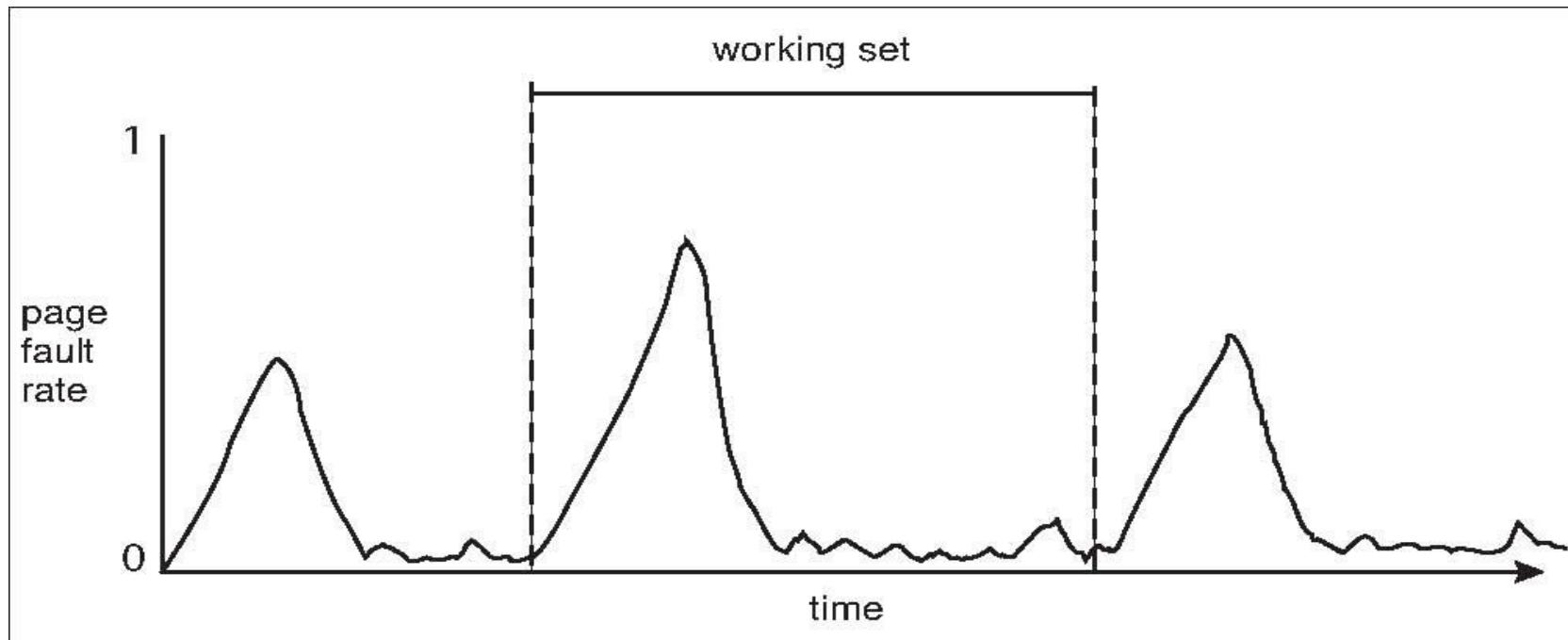
- More direct approach than WSS
- Establish “acceptable” page-fault frequency rate and use local replacement policy

- If actual rate too low process loses frame

- If actual rate too high process gets frame



# Working Sets and Page Fault Rates

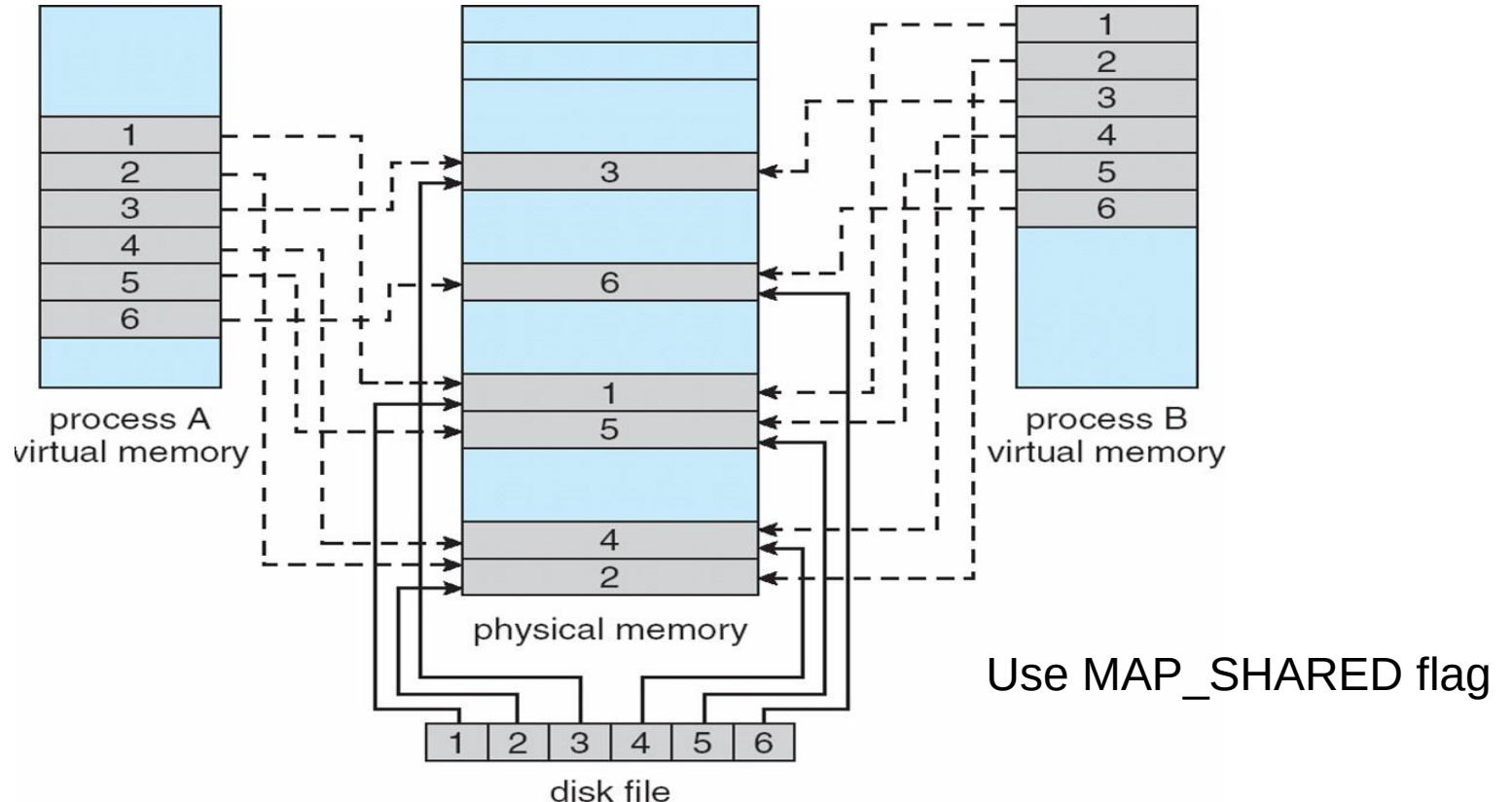


# **Memory Mapped Files**

# Memory-Mapped Files

- First, let's see a demo of using `mmap()`

# Memory-Mapped Files



# Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory
- A file is initially read using demand paging
  - A page-sized portion of the file is read from the file system into a physical page
  - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies and speeds file access by driving file I/O through memory rather than read() and write() system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared
- But when does written data make it to disk?
  - Periodically and / or at file close() time
  - For example, when the pager scans for dirty pages

# Memory-Mapped Files

- Some OSes uses memory mapped files for standard I/O
- Process can explicitly request memory mapping a file via `mmap()` system call
  - Now file mapped into process address space
- For standard I/O (`open()`, `read()`, `write()`, `close()`), `mmap` anyway
  - But map file into kernel address space
  - Process still does `read()` and `write()`
    - Copies data to and from kernel space and user space
  - Uses efficient memory management subsystem
    - Avoids needing separate subsystem
- COW can be used for read/write non-shared pages
- Memory mapped files can be used for shared memory (although again via separate system calls)

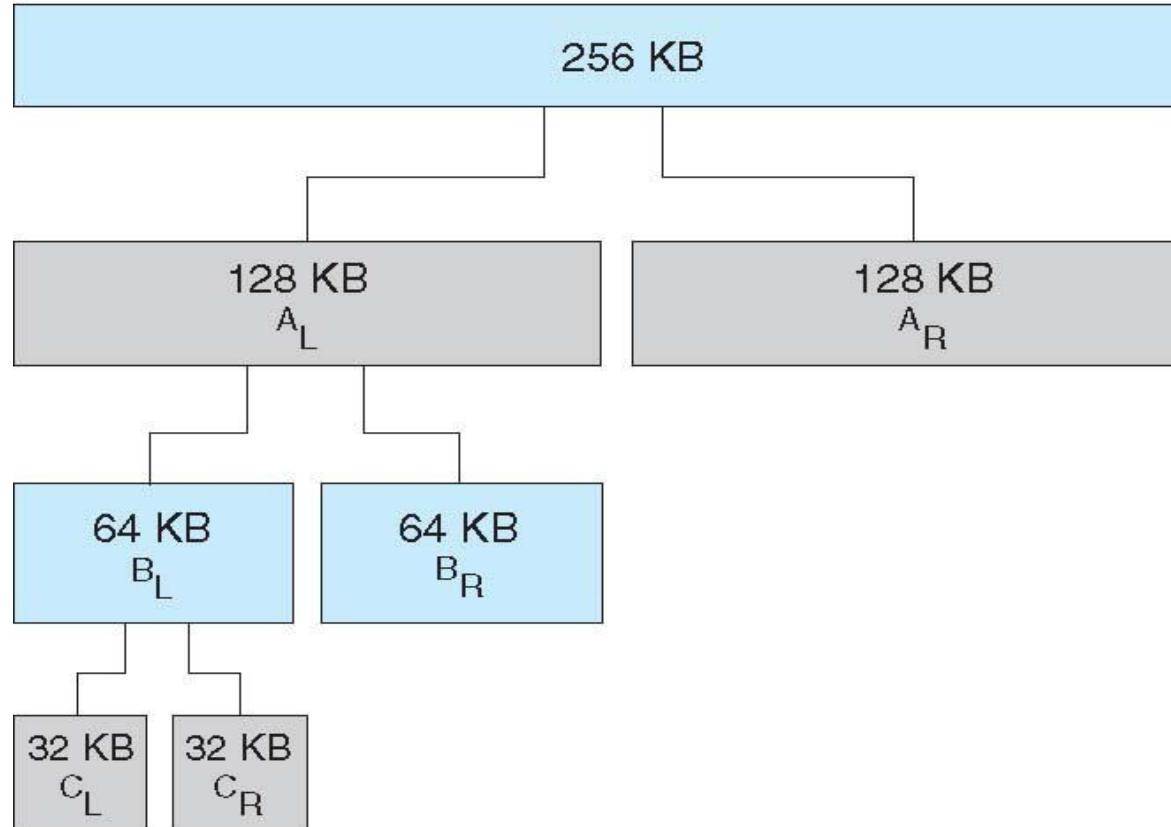
# **Allocating Kernel Memory**

# Allocating kernel memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
  - I.e. for device I/O
-

# Buddy Allocator

physically contiguous pages



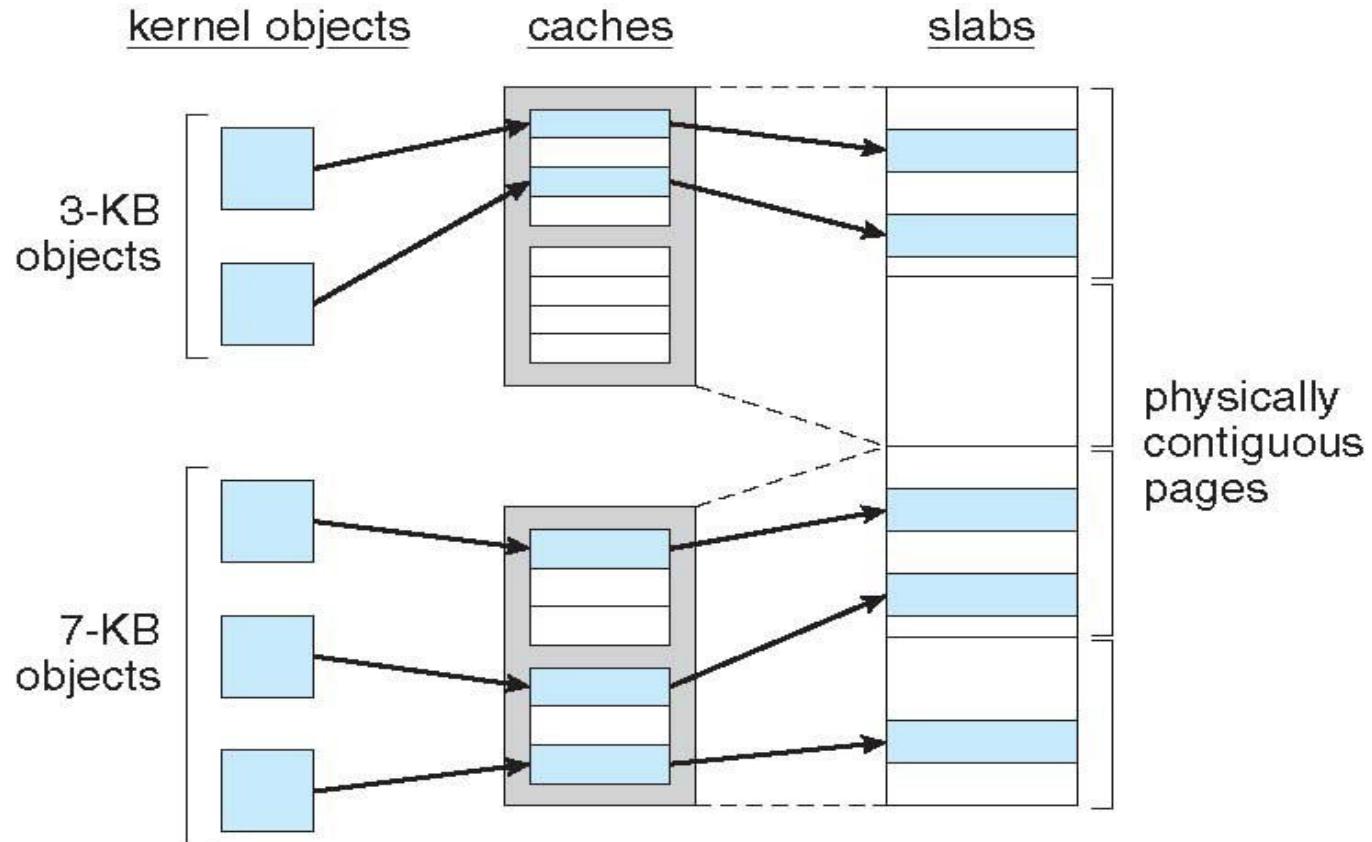
# Buddy Allocator

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using power-of-2 allocator
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
  - Continue until appropriate sized chunk available

# Buddy Allocator

- Continue until appropriate sized chunk available
- For example, assume 256KB chunk available, kernel requests 21KB
  - Split into AL and Ar of 128KB each
    - One further divided into BL and BR of 64KB
      - One further into CL and CR of 32KB each – one used to satisfy request
- Advantage – quickly coalesce unused chunks into larger chunk
- Disadvantage - fragmentation

# Slab Allocator



# Slab Allocator

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **free**
- When structures stored, objects marked as **used**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction

# **Other considerations**

# Other Considerations -- Prepaging

- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Assume  $s$  pages are prepaged and  $\alpha$  of the pages is used
  - Is cost of  $s * \alpha$  save pages faults  $>$  or  $<$  than the cost of prepaging  $s * (1 - \alpha)$  unnecessary pages?
  - $\alpha$  near zero --> prepaging loses

# Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - **Resolution**
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Always power of 2, usually in the range  $2^{12}$  (4,096 bytes) to  $2^{22}$  (4,194,304 bytes)
- On average, growing over time

# TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- $\text{TLB Reach} = (\text{TLB Size}) \times (\text{Page Size})$
- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults
- Increase the Page Size
  - This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

# Program Structure

- Program structure
- `Int[128,128] data;`
- Each row is stored in one page
- Program 1

```
for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
        data[i,j] = 0;
```

$128 \times 128 = 16,384$  page faults

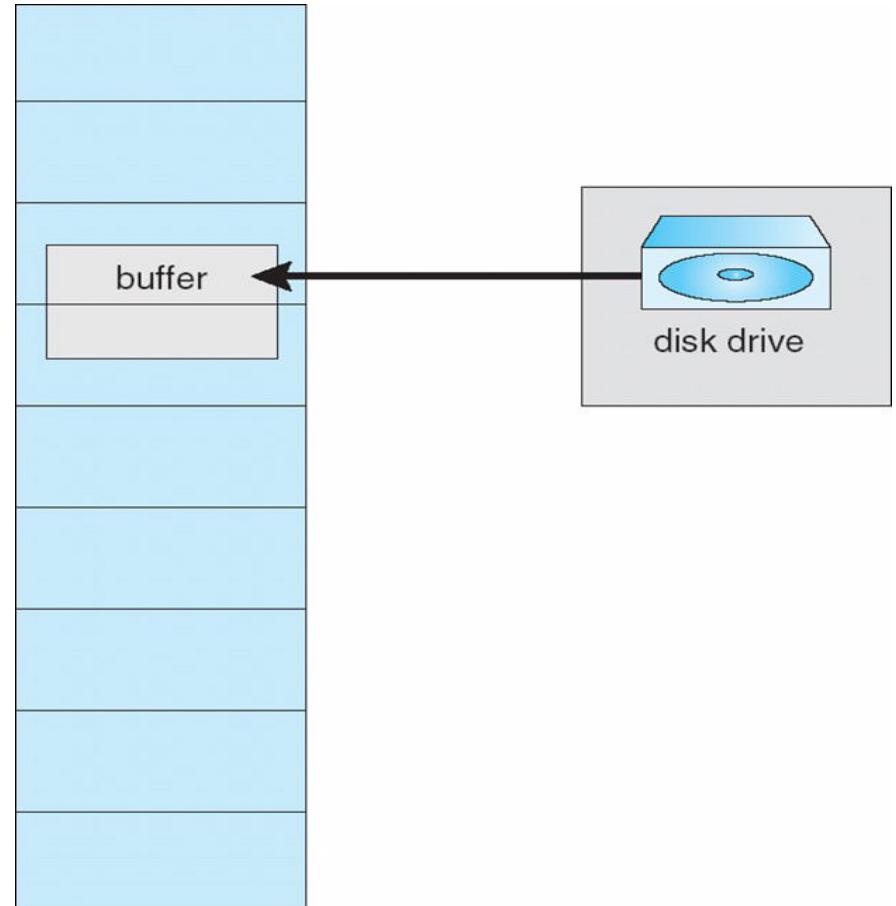
- Program 2

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;
```

128 page faults

# I/O Interlock

- **I/O Interlock** – Pages must sometimes be locked into memory
- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm



# Threads

Abhijit A M  
abhijit.comp@coep.ac.in

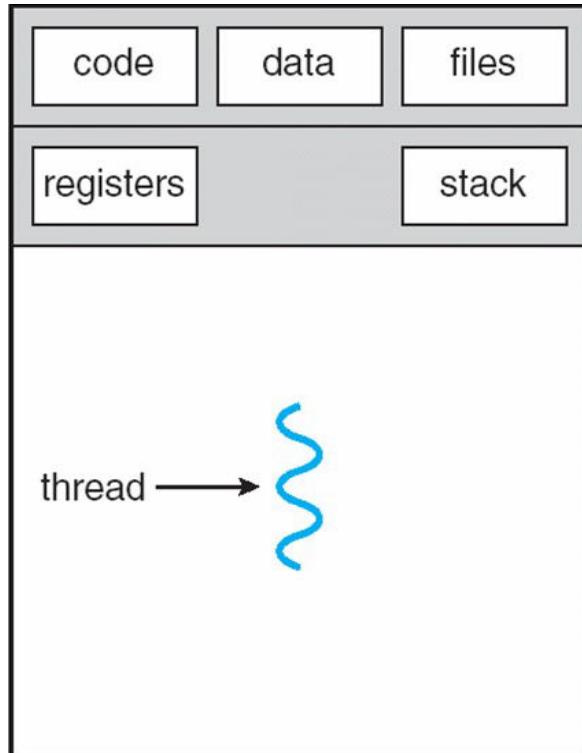
# Threads

- **thread — a fundamental unit of CPU utilization**
  - A separate control flow within a program
  - set of instructions that execute “concurrently” with other parts of the code
  - Note the difference: Concurrency: progress at the same time, Parallel: execution at the same time
- **Threads run within application**
  - An application can be divided into multiple parts
  - Each part may be written to execute as a threads
- **Let's see an example**

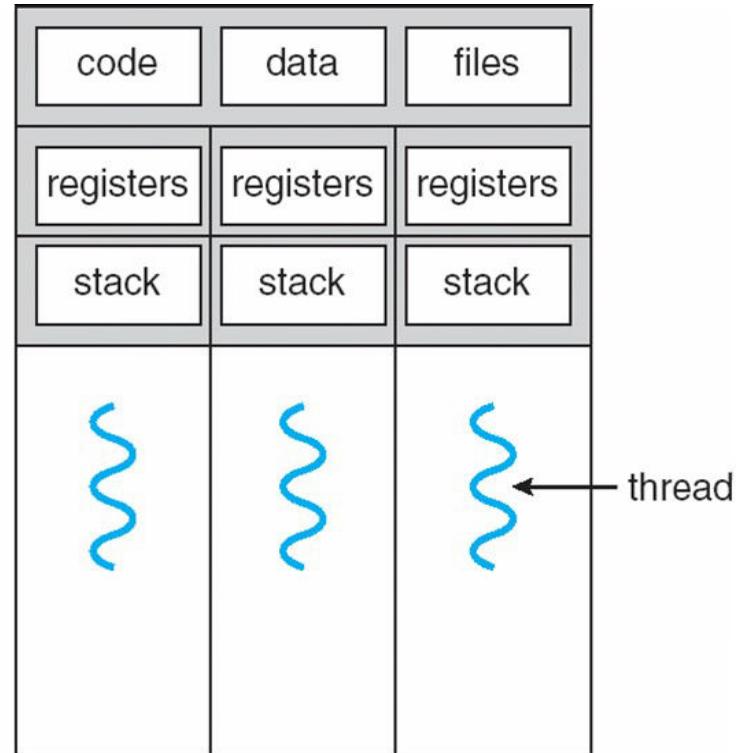
# Threads

- **Multiple tasks with the application can be implemented by separate threads**
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- **Process creation is heavy-weight while thread creation is light-weight, due to the very nature of threads**
- **Can simplify code, increase efficiency**
- **Kernels are generally multithreaded**

# Single vs Multithreaded process

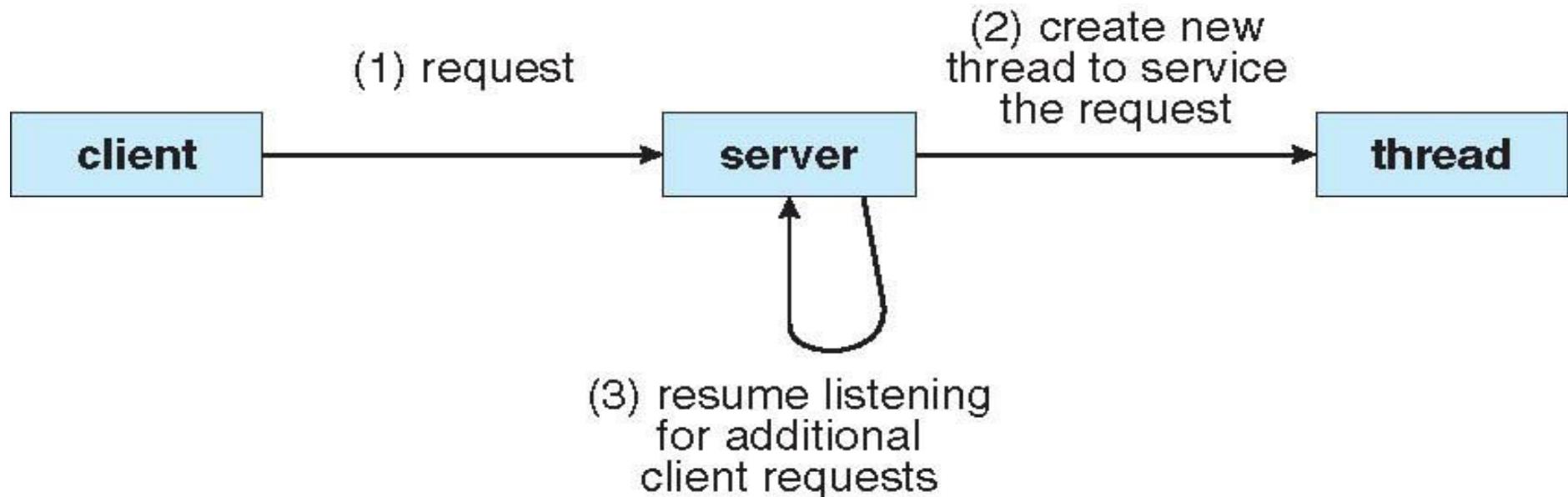


single-threaded process



multithreaded process

# A multithreaded server

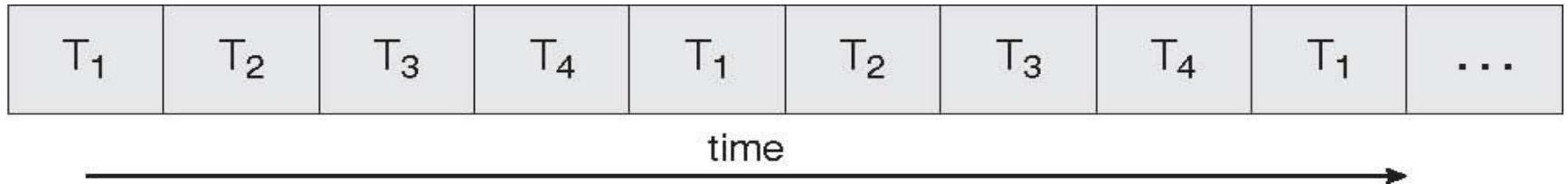


# Benefits of threads

- **Responsiveness**
- **Resource Sharing**
- **Economy**
- **Scalability**

# Single vs Multicore systems

single core



**Single core : Concurrency possible**

**Multicore : parallel execution possible**

core 1



core 2



time

# Multicore programming

- Multicore systems putting pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging
  -

# User vs Kernel Threads

- **User Threads:** Thread management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads
- **Kernel Threads:** Supported by the Kernel
- **Examples**
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

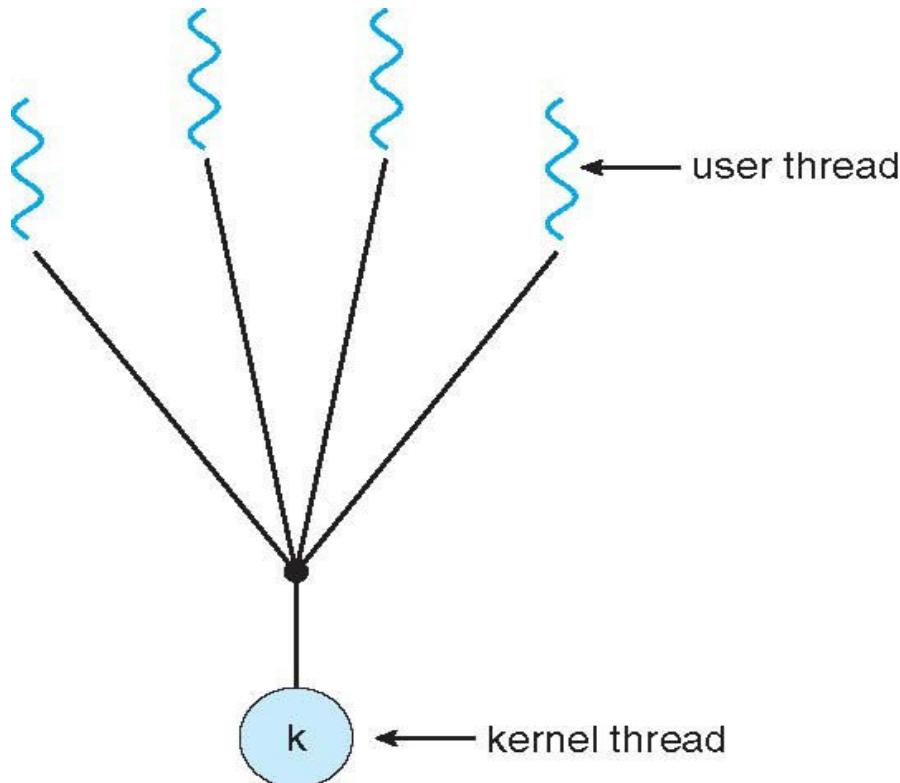
# User threads vs Kernel Threads

- **User threads**
  - User level library provides a “typedef” called threads
  - The scheduling of threads needs to be implemented in the user level library
  - Need some type of timer handling functionality at user level execution of CPU
    - OS needs to provide system calls for this
  - Kernel does not know that there are threads!
- **Kernel Threads**
  - Kernel implements concept of threads
  - Still, there may be a user level library, that maps kernel concept of threads to “user concept” since applications link with user level libraries
  - Kernel does scheduling!

# Multithreading models

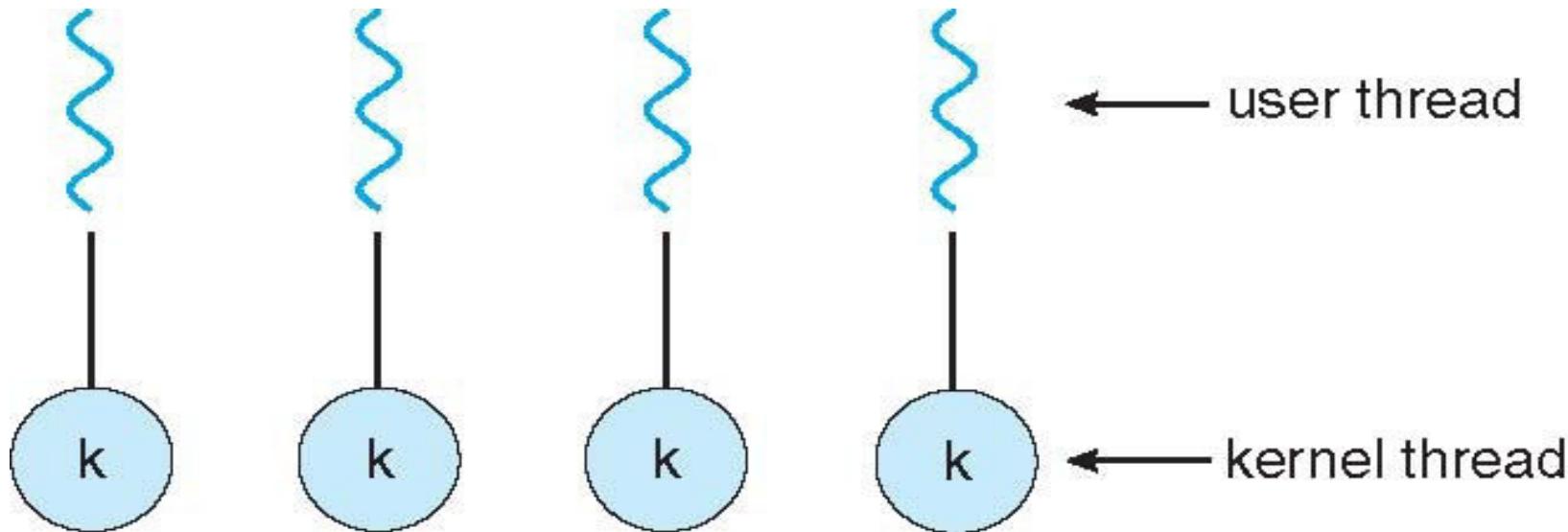
- **How to map user threads to kernel threads?**
  - Many-to-One
  - One-to-One
  - Many-to-Many
- **What if there are no kernel threads?**
  - Then only “one” process. Hence many-one mapping possible, to be done by user level thread library
  - Is One-One possible?

# Many-One Model



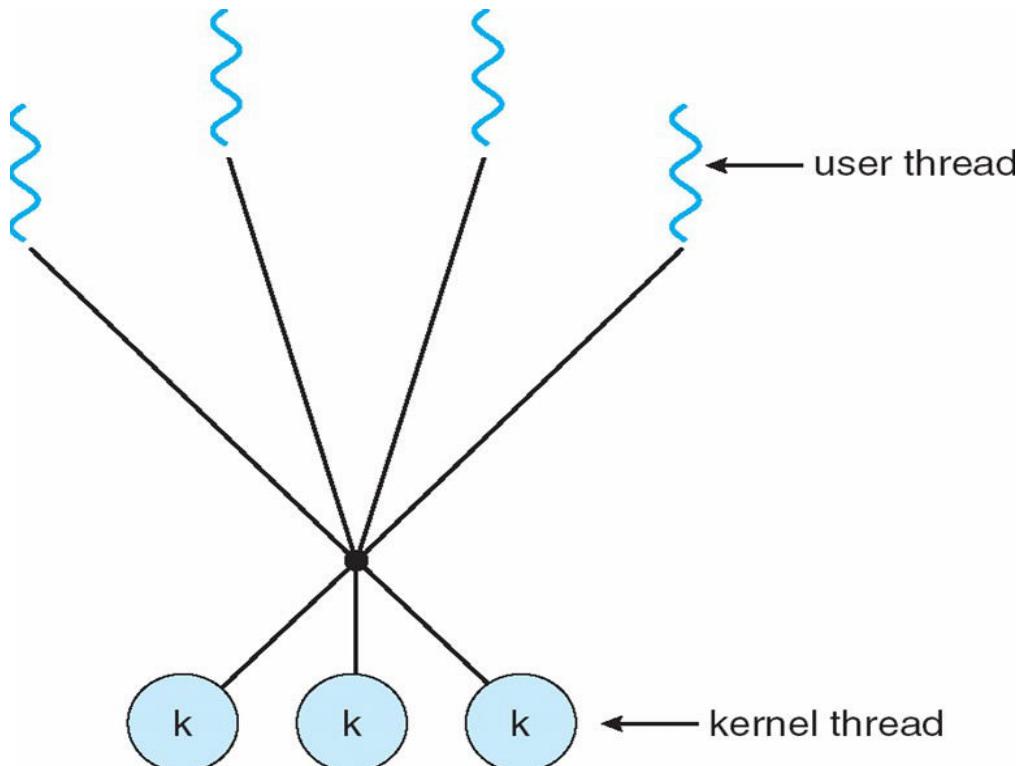
- **Many user-level threads mapped to single kernel thread**
- **Examples:**
  - Solaris Green Threads
  - GNU Portable Threads

# One-One Model



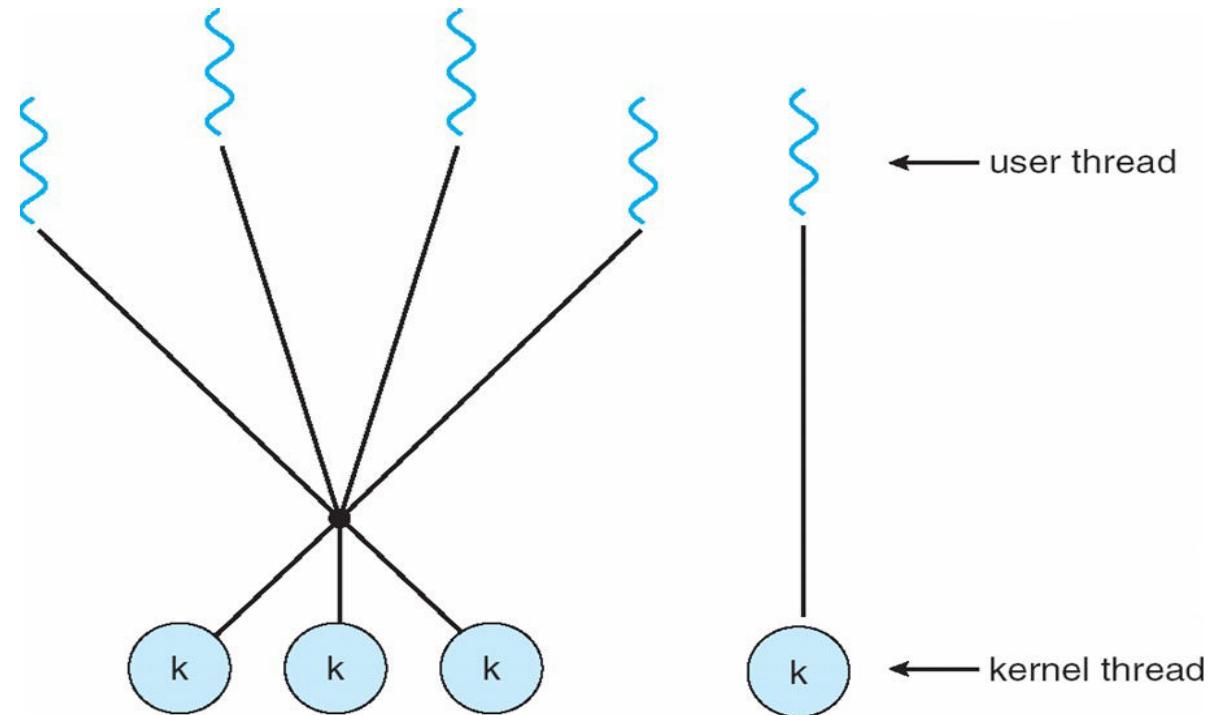
- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

# Many-Many Model



- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the ThreadFiber package

# Two Level Model



- Similar to M:M, except that it allows a user thread to be bound to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

# **Thread Libraries**

# Thread libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

# pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)
-

# **Demo of pthreads code**

**Demonstration on Linux – see the code,  
compile and execute it.**

# Other libraries

- **Windows threading API**
  - `CreateThread(...)`
  - `WaitForSingleObject(...)`
  - `CloseHandle(...)`
- **Java Threads**
  - The `Threads` class
  - The `Runnable` Interface

# Issues with threads

- **Semantics of fork() and exec() system calls**
  - Does fork() duplicate only the calling thread or all threads?
- **Thread cancellation of target thread**
  - Terminating a thread before it has finished
  - Two general approaches:
    - Asynchronous cancellation terminates the target thread immediately.
    - Deferred cancellation allows the target thread to periodically check if it should be cancelled.

# **More on threads**

# Thread pools

- Some kernels/libraries can provide system calls to :  
**Create a number of threads in a pool where they await work, assign work/function to a waiting thread**
- **Advantages:**
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool

# Thread Local Storage (TLS)

- Thread-specific data, Thread Local Storage (TLS)
  - Not local, but global kind of data for all functions of a thread, more like “static” data
  - Create Facility needed for data private to thread
  - Allows each thread to have its own copy of data
  - Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
  - gcc compiler provides the storage class keyword `thread` for declaring TLS data

```
static __thread int threadID;
```

```
int arr[16];  
int f() {  
    a(); b(); c();  
}  
int g() {  
    x(); y();  
}  
int main() {  
    th_create(...,f,...);  
    th_create(...,g,...);  
}  
//arr is visible to all of them!  
//need data for only f,a,b,c  
//need data for only g,x,y
```

# Thread Local Storage (TLS) in pthreads

- Functions
  - `pthread_key_create`
  - `pthread_key_delete`
  - `pthread_setspecific`
  - `pthread_getspecific`
- See
  - `thrd_specific.c` file

# Scheduler activations for threads

- **Scheduler Activations**
  - Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
  - Scheduler activations provide upcalls - a communication mechanism from the kernel to the thread library
  - This communication allows an application to maintain the correct number kernel threads

# Scheduler activations for threads

## Library

```
--  
upcall_handler() {  
    Create one more LWP and schedule threads on  
    this;  
}  
th_setup(int n) {  
    max_LSW = n;  
    curr_LWP = 0;  
    register_upcall(upcall_handler);  
}  
th_create(..., fn,...) {  
    if(curr_LWP < max_LWP)  
        create LWP;  
    schedule fn on one of the LWP;  
}
```

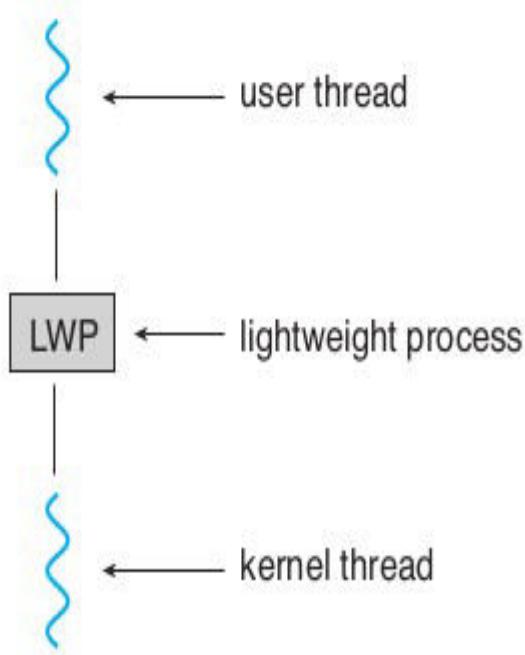
## application

```
--  
f0 {  
    scanf();  
}  
g0 {  
    recv();  
}  
h0 {...}; i0 {...}  
main()  
    th_setup(2);  
    th_create(...,f,...);  
    th_create(...,g,...);  
    th_create(...,h,...);  
    th_create(...,i,...);  
}
```

## Kernel

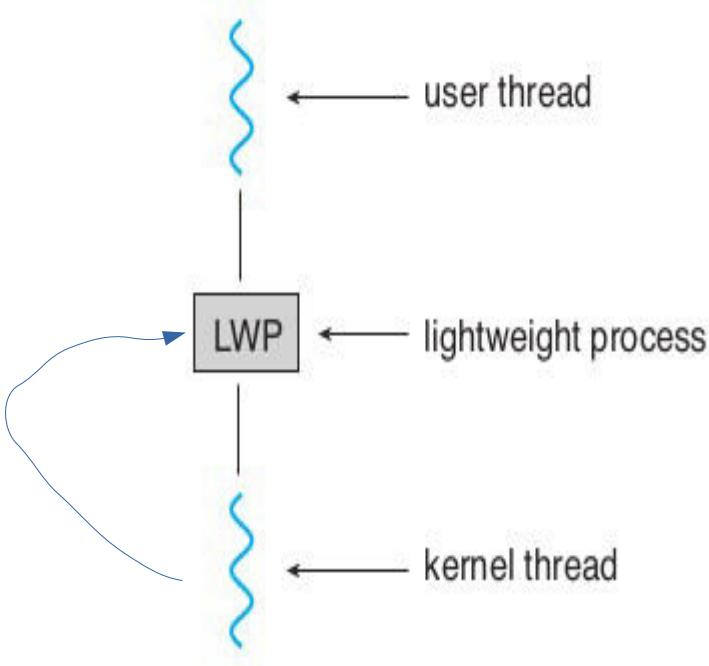
```
--  
register_upcall(function f) {  
    proc->upcall = f;  
}  
sys_write() {  
    // before calling sleep() going  
    to block  
    myproc()->upcall(); // tricky!  
}
```

# Issues with threads



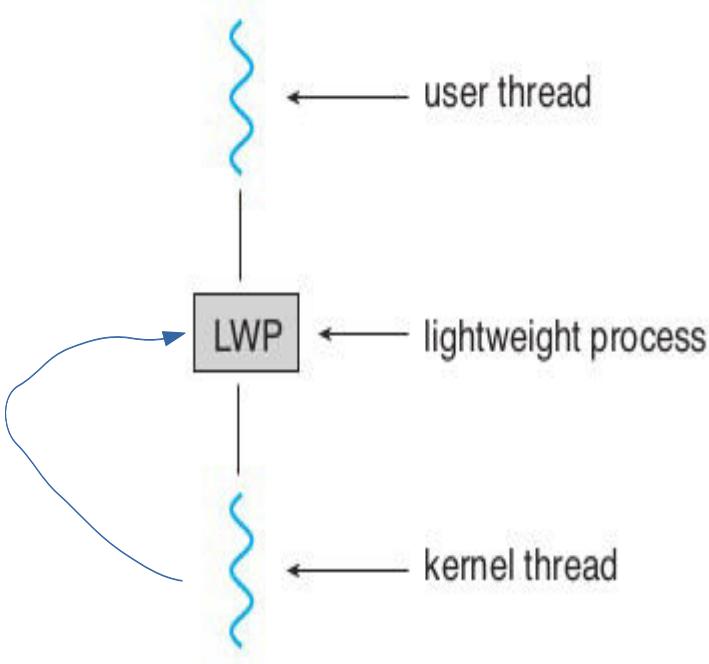
- **Scheduler Activations: LWP approach**
  - An intermediate data structure LWP
  - appears to be a virtual processor on which the application can schedule a user thread to run.
  - Each LWP attached to a kernel thread
  - Typically one LWP per blocking call, e.g. 5 file I/Os in one process, then 5 LWPs needed

# Issues with threads



- **Scheduler Activations: LWP approach**
  - Kernel needs to inform application about events like: a thread is about to block, or wait is over : this is ‘upcall’
  - This will help application relinquish the LWP or request a new LWP

# Issues with threads



- **Example NetBSD**
  - **“An Implementation of Scheduler Activations on the NetBSD Operating System”**
  - <https://web.mit.edu/nathanw/www/usenix/freenix-sa/freenix-sa.html>

# Linux threads

- Only threads (called task), no processes!
- Process is a thread that shares many particular resources with the parent thread
- `Clone()` system call to create a thread

# Linux threads

- **clone() takes options to determine sharing on process create**
- **struct task\_struct points to process data structures (shared or unique depending on clone options)**
- **fork() is a wrapper on top of clone()**
  - Use ‘strace’ to see this.

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

# Issues in implementing threads project

- How to implement a user land library for threads?
- How to handle 1-1, many-one, many-many implementations?
- Identifying the support required from OS and hardware
- Identifying the libraries that will help in implementation

# Issues in implementing threads project

- **Understand the clone() system call completely**
  - Try out various possible ways of calling it
  - Passing different options
  - Passing a user-land buffer as stack
- **How to save and restore context?**
  - C: setjmp, longjmp
  - Setcontext, getcontext(), makecontext(), swapcontext() functions
- **Sigaction is more powerful than signal**
  - Learn SIGALRM handling for timer and scheduler, timer\_create() & timer\_stop() system calls
- **Customized data structure to store threads, and manage thread-lists for scheduling**

# **Signals**

# Signals

- Signals are used in UNIX systems to notify a process that a particular event has occurred.
- Signal handling
  - Synchronous and asynchronous
- A signal handler (a function) is used to process signals
  - Signal is generated by particular event
  - Signal is delivered to a process
  - Then, signal is “handled” by the handler

# Signals

- **More about signals**
  - Different signals are typically identified as different numbers
  - Operating systems provide system calls like `kill()` and `signal()` to enable processes to deliver and receive signals
  - `Signal()` - is used by a process to specify a “signal handler” – a code that should run on receiving a signal
  - `Kill()` is used by a process to send another process a signal
  - There are restrictions on which process can send which signal to other processes

# Demo

- Let's see a demo of signals with respect to processes
- Let's see signal.h
  - /usr/include/signal.h
  - /usr/include/asm-generic/signal.h
  - /usr/include/linux/signal.h
  - /usr/include/sys/signal.h
  - /usr/include/x86\_64-linux-gnu/asm/signal.h
  - /usr/include/x86\_64-linux-gnu/sys/signal.h
- man 7 signal
- Important signals: SIGKILL, SIGUSR1, SIGSEGV, SIGALRM, SIGCLD, SIGINT, SIGPIPE, ...

# Signal handling by OS

```
Process 12323 {  
    signal(19, abcd);  
}
```

```
OS: sys_signal {  
    Note down that process 12323  
    wants to handle signal number  
    19 with function abcd  
}
```

```
Process P1 {  
    kill (12323, 19) ;  
}
```

```
OS: sys_kill {
```

Note down in PCB of process 12323 that signal number 19 is pending for you.

When process 12323 is scheduled, at that time the OS will check for pending signals, and invoke the appropriate signal handler for a pending signal.

# Threads and Signals

- **Signal handling Options:**
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process

# Synchronization

# My formulation

- OS = data structures + synchronization
- Synchronization problems make writing OS code challenging
- Demand exceptional coding skills

# Race problem

```
long c = 0, c1 = 0, c2 = 0, run = 1;  
void *thread1(void *arg) {  
    while(run == 1) {  
        c++;  
        c1++;  
    }  
}  
void *thread2(void *arg) {  
    while(run == 1) {  
        c++;  
        c2++;  
    }  
}
```

```
int main() {  
    pthread_t th1, th2;  
    pthread_create(&th1, NULL, thread1,  
NULL);  
    pthread_create(&th2, NULL, thread2,  
NULL);  
    //fprintf(stdout, "Ending main\n");  
    sleep(2);  
    run = 0;  
    fprintf(stdout, "c = %ld c1+c2 = %ld  
c1 = %ld c2 = %ld \n", c, c1+c2, c1, c2);  
    fflush(stdout);  
}
```

# Race problem

- On earlier slide
  - Value of c should be equal to  $c1 + c2$ , but it is not!
  - Why?
- There is a “race” between thread1 and thread2 for updating the variable c
- thread1 and thread2 may get scheduled in any order and *interrupted* any point in time
- The changes to c are not atomic!
  - What does that mean?

# Race problem

- C++, when converted to assembly code, could be

```
mov c, r1  
add r1, 1  
mov r1, c
```

- Now following sequence of instructions is possible among thread1 and thread2

```
thread1: mov c, r1  
thread2: mov c, r1  
thread1: add r1, 1  
thread1: mov r1, c  
thread2: add r1, 1  
thread2: mov r1, c
```

- What will be value in c, if initially c was, say 5?

- It will be 6, when it is expected to be 7. Other variations also possible.

# Races: reasons

- **Interruptible kernel**
  - If entry to kernel code does not disable interrupts, then modifications to any kernel data structure can be left incomplete
  - This introduces concurrency
- **Multiprocessor systems**
  - On SMP systems: memory is shared, kernel and process code run on all processors
  - Same variable can be updated parallelly (not concurrently)
- **What about non-interruptible kernel on multiprocessor systems?**
- **What about non-interruptible kernel on uniprocessor systems?**

# Critical Section problem

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (TRUE);
```

Figure 6.1 General structure of a typical process  $P$ .

# Critical Section Problem

- Consider system of n processes {p<sub>0</sub>, p<sub>1</sub>, ... p<sub>n-1</sub>}
- Each process has critical section segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- Critical section problem is to design protocol to solve this
- Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section
- Especially challenging with preemptive kernels

# Expected solution characteristics

- **1. Mutual Exclusion**
  - If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections
- **2. Progress**
  - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
- **3. Bounded Waiting**
  - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
    - Assume that each process executes at a nonzero speed
    - No assumption concerning relative speed of the  $n$  processes

# suggested solution - 1

```
int flag = 1;  
void *thread1(void *arg) {  
    while(run == 1) {  
        while(flag == 0)  
            ;  
        flag = 0;  
        c++;  
        flag = 1;  
        c1++;  
    }  
}
```

- **What's wrong here?**
- **Assumes that**  
**while(flag ==) ; flag**  
**= 0**  
**will be atomic**

# suggested solution - 2

```
int flag = 0;  
  
void *thread1(void *arg) {  
    while(run == 1) {  
        if(flag)  
            c++;  
        else  
            continue;  
        c1++;  
        flag = 0;  
    }  
}
```

```
void *thread2(void *arg) {  
    while(run == 1) {  
        if(!flag)  
            c++;  
        else  
            continue;  
        c2++;  
        flag = 1;  
    }  
}
```

# Peterson's solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process Pi is ready!

# Peterson's solution

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j)
        ;
    critical section
    flag[i] = FALSE;
    remainder section
} while (TRUE);
```

- Provable that
  - Mutual exclusion is preserved
  - Progress requirement is satisfied
  - Bounded-waiting requirement is met

# Hardware solution – the one actually implemented

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
  - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words
  - Basically two operations (read/write) done atomically in hardware

# Solution using test-and-set

```
lock = false; //global
```

```
do {  
    while ( TestAndSet (&lock ) )  
        ; // do nothing  
    // critical section  
    lock = FALSE;  
    // remainder section  
} while (TRUE);
```

Definition:

```
boolean TestAndSet (boolean  
*target)  
{  
    boolean rv = *target;  
    *target = TRUE;  
    return rv;  
}
```

# Solution using swap

```
lock = false; //global  
  
do {  
    key = true  
    while ( key == true))  
        swap(&lock, &key)  
        // critical section  
    lock = FALSE;  
    // remainder section  
} while (TRUE);
```

# Spinlock

- A lock implemented to do ‘busy-wait’
- Using instructions like T&S or Swap
- As shown on earlier slides

```
spinlock(int *lock){  
    While(test-and-set(lock))  
        ;  
}  
  
spinunlock(lock *lock) {  
    *lock = false;  
}
```

# Bounded wait M.E. with T&S

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;
    // critical section
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // remainder section
} while (TRUE);
```

# **sleep-locks**

- **Spin locks result in busy-wait**
- **CPU cycles wasted by waiting processes/threads**
- **Solution – threads keep waiting for the lock to be available**
  - Move thread to wait queue
  - The thread holding the lock will wake up one of them

# Sleep locks/mutexes

```
//ignore syntactical issues  
typedef struct mutex {  
    int islocked;  
    int spinlock;  
    waitqueue q;  
}mutex;  
wait(mutex *m) {  
    spinlock(m->spinlock);  
    while(m->islocked)  
        Block(m, m->spinlock)  
    lk->islocked = 1;  
    spinunlock(m->spinlock);  
}
```

```
Block(mutex *m, spinlock *sl) {  
    currprocess->state = WAITING  
    move current process to m->q  
    spinunlock(sl);  
    Sched();  
    spinlock(sl);  
}  
release(mutex *m) {  
    spinlock(m->spinlock);  
    m->islocked = 0;  
    Some process in m->queue  
    =RUNNABLE;  
    spinunlock(m->spinlock);  
}
```

# Some thumb-rules of spinlocks

- **Never block a process holding a spinlock !**

- **Typical code:**

```
while(condition)
    { Spin-unlock()
      Schedule()
      Spin-lock()
    }
```

- **Hold a spin lock for only a short duration of time**

- **Spinlocks are preferable on multiprocessor systems**
- **Cost of context switch is a concern in case of sleep-wait locks**
- **Short = < 2 context switches**

# Locks in xv6 code

# struct spinlock

// Mutual exclusion lock.

```
struct spinlock {
```

```
    uint locked;      // Is the lock held?
```

// For debugging:

```
    char *name;      // Name of lock.
```

```
    struct cpu *cpu; // The cpu holding the lock.
```

```
    uint pcs[10];    // The call stack (an array of program counters)
```

```
                           // that locked the lock.
```

```
};
```

# spinlocks in xv6 code

```
struct {  
    struct spinlock lock;  
    struct buf buf[NBUF];  
    struct buf head;  
} bcache;  
  
struct {  
    struct spinlock lock;  
    struct file file[NFILE];  
} ftable;  
  
struct {  
    struct spinlock lock;  
    struct inode inode[NINODE];  
} icache;  
  
struct sleeplock {  
    uint locked;      // Is the lock held?  
    struct spinlock lk;
```

```
static struct spinlock idelock;  
  
struct {  
    struct spinlock lock;  
    int use_lock;  
    struct run *freelist;  
} kmem;  
  
struct log {  
    struct spinlock lock;  
    ...}  
  
struct pipe {  
    struct spinlock lock;  
    ...}  
  
struct {  
    struct spinlock lock;  
    struct proc proc[NPROC];  
} ptable;  
  
struct spinlock tickslock;
```

```

static inline uint
xchg(volatile uint *addr, uint newval)
{
    uint result;
    // The + in "+m" denotes a read-modify-
    // write operand.
    asm volatile("lock; xchgl %0, %1" :
        "+m" (*addr), "=a" (result) :
        "1" (newval) :
        "cc");
    return result;
}

struct spinlock {
    uint locked;      // Is the lock held?

    // For debugging:
    char *name;      // Name of lock.
    struct cpu *cpu; // The cpu holding the
                     // lock.

    uint pcs[10];    // The call stack (an array
                     // of program counters) that locked the lock.
};

```

# Spinlock in xv6

```

void acquire(struct spinlock *lk)
{
    pushcli(); // disable interrupts to
               // avoid deadlock.

    // The xchg is atomic.
    while(xchg(&lk->locked, 1) != 0)
        ;

    //extra debugging code
}

void release(struct spinlock *lk)
{
    //extra debugging code
    asm volatile("movl $0, %0" :
        "+m" (lk->locked) : );
    popcli();
}

```

```

Void acquire(struct spinlock *lk)
{
    pushcli(); // disable interrupts to avoid deadlock.
    if(holding(lk))
        panic("acquire");
    .....
void pushcli(void)
{
    int eflags;
    eflags = readeflags();
    cli();
    if(mycpu()->ncli == 0)
        mycpu()->intena = eflags & FL_IF;
    mycpu()->ncli += 1;
}
static inline uint
readeflags(void)
{
    uint eflags;
    asm volatile("pushfl; popl %0" : "=r" (eflags));
    return eflags;
}

```

# spinlocks

- **Pushcli() - disable interrupts on that processor**
- **One after another many acquire() can be called on different spinlocks**
  - Keep a count of them in mycpu()->ncli

```

void
release(struct spinlock *lk)
{
...
    asm volatile("movl $0, %0" : "+m" (lk->locked) : );
    popcli();
}

.

Void popcli(void)
{
    if(readeflags()&FL_IF)
        panic("popcli - interruptible");
    if(--mycpu()->ncli < 0)
        panic("popcli");
    if(mycpu()->ncli == 0 && mycpu()->intena)
        sti();
}

```

# spinlocks

- **Popcli()**

- **Restore interrupts if last popcli() call restores ncli to 0 & interrupts were enabled before pushcli() was called**

# spinlocks

- Always disable interrupts while acquiring spinlock
  - Suppose **iderw** held the **idelock** and then got interrupted to run **ideintr**.
  - **Ideintr** would try to lock **idelock**, see it was held, and wait for it to be released.
  - In this situation, **idelock** will never be released
  - Deadlock
- General OS rule: if a spin-lock is used by an interrupt handler, a processor must never hold that lock with interrupts enabled
- Xv6 rule: when a processor enters a spin-lock critical section, xv6 always ensures interrupts are disabled on that processor.

# sleeplocks

- **Sleeplocks don't spin. They move a process to a wait-queue if the lock can't be acquired**
- **XV6 approach to “wait-queues”**
  - Any memory address serves as a “wait channel”
  - The sleep() and wakeup() functions just use that address as a ‘condition’
  - There are no per condition process queues! Just one global queue of processes used for scheduling, sleep, wakeup etc. --> Linear search everytime !
    - **costly, but simple**

```
void  
sleep(void *chan, struct spinlock *lk)  
{  
    struct proc *p = myproc();  
    ....  
    if(lk != &ptable.lock){  
        acquire(&ptable.lock);  
        release(lk);  
    }  
    p->chan = chan;  
    p->state = SLEEPING;  
    sched();  
    // Reacquire original lock.  
    if(lk != &ptable.lock){  
        release(&ptable.lock);  
        acquire(lk);  
    }  
}
```

# sleep()

- At call must hold lock on the resource on which you are going to sleep
- since you are going to change p-> values & call sched(), hold ptable.lock if not held
- p->chan = given address remembers on which condition the process is waiting
- call to sched() blocks the process

# Calls to sleep() : examples of “chan” (output from cscope)

0 console.c

consoleread 251

sleep(&input.r, &cons.lock);

2 ide.c iderw

169 sleep(b, &idelock);

3 log.c begin\_op

131 sleep(&log, &log.lock);

6 pipe.c piperead

111 sleep(&p->nread, &p->lock);

7 proc.c wait

317 sleep(curproc,  
&phtable.lock);

8 sleeplock.c

acquiresleep 28

sleep(lk, &lk->lk);

9 sysproc.c

sys\_sleep 74

sleep(&ticks, &tickslock);

```

void wakeup(void *chan)
{
    acquire(&ptable.lock);
    wakeup1(chan);
    release(&ptable.lock);
}

static void wakeup1(void *chan)
{
    struct proc *p;

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

```

# Wakeup()

- Acquire ptable.lock since you are going to change ptable and p-> values
- just linear search in process table for a process where p->chan is given address
- Make it runnable

# sleeplock

// Long-term locks for processes

struct sleeplock {

    uint locked; // Is the lock held?

    struct spinlock lk; // spinlock protecting this sleep lock

// For debugging:

    char \*name; // Name of lock.

    int pid; // Process holding lock

};

# Sleeplock acquire and release

```
void
acquiresleep(struct sleeplock *lk)
{
    acquire(&lk->lk);
    while (lk->locked) {
        /* Abhijit: interrupts are not disabled in
         sleep !*/
        sleep(lk, &lk->lk);
    }
    lk->locked = 1;
    lk->pid = myproc()->pid;
    release(&lk->lk);
}
```

```
void
releasesleep(struct sleeplock
*Lk)
{
    acquire(&lk->lk);
    lk->locked = 0;
    lk->pid = 0;
    wakeup(lk);
    release(&lk->lk);
}
```

# Where are sleeplocks used?

- **struct buf**
  - waiting for I/O on this buffer
- **struct inode**
  - waiting for I/o to this inode
- Just two !

# Sleeplocks issues

- sleep-locks support yielding the processor during their critical sections.
- This property poses a design challenge:
  - if thread T1 holds lock L1 and has yielded the processor (waiting for some other condition),
  - and thread T2 wishes to acquire L1,
  - we have to ensure that T1 can execute
  - while T2 is waiting so that T1 can release L1.
  - T2 can't use the spin-lock acquire function here: it spins with interrupts turned off, and that would prevent T1 from running.
- To avoid this deadlock, the sleep-lock acquire routine (called `acquiresleep`) yields the processor while waiting, and does not disable interrupts.

Sleep-locks leave interrupts enabled, they cannot be used in interrupt handlers.

# More needs of synchronization

- Not only critical section problems
- Run processes in a particular order
- Allow multiple processes read access, but only one process write access
- Etc.

# 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++;  
}  
--> Note this is Signal() on a semaphore, different froms signal system call
```

# Semaphore for synchronization

- **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 mutex; // initialized to 1
do {
    wait (mutex);
        // Critical Section
    signal (mutex);
    // remainder section
} while (TRUE)
```

# Semaphore implementation

```
Wait(sem *s) {  
    while(s <=0)  
        block(); // could be ";"  
    s--;  
}  
  
signal(sem *s) {  
    s++;  
}
```

- **Left side – expected behaviour**
- **Both the wait and signal should be atomic.**
- **This is the semantics of the semaphore.**

# Semaphore implementation? - 1

```
struct semaphore {  
    int val;  
    spinlock lk;  
};  
sem_init(semaphore *s, int initval) {  
    s->val = initval;  
    s->sl = 0;  
}  
wait(semaphore *s) {  
    spinlock(&(s->sl));  
    while(s->val <=0)  
    ;  
    (s->val)--;  
    spinunlock(&(s->sl));  
}
```

```
signal(semaphore *s) {  
    spinlock(*(s->sl));  
    (s->val)++;  
    spinunlock(*(s->sl));  
}  
- suppose 2 processes trying wait.  
val = 1;  
Th1: spinlock                   Th2: spinlock-waits  
Th1: while -> false, val-- => 0; spinunlock;  
Th2: spinlock success; while() -> true, loops;  
Th1: is done with critical section, it calls signal. it calls spinlock() -> wait.  
Who is holding spinlock-> Th2. It is waiting for val > 0. Who can set value > 0 , ans: Th1, and Th1 is waiting for spinlock which is held by Th2.  
circular wait. Deadlock.  
None of them will proceed.
```

# Semaphore implementation? - 2

```
struct semaphore {  
    int val;  
    spinlock lk;  
};  
sem_init(semaphore *s, int initval) {  
    s->val = initval;  
    s->sl = 0;  
}  
signal(semaphore *s) {  
    spinlock(*(s->sl));  
    (s->val)++;  
    spinunlock(*(s->sl));  
}
```

```
wait(semaphore *s) {  
    spinlock(&(s->sl));  
    while(s->val <=0) {  
        spinunlock(&(s->sl));  
        spinlock(&(s->sl));  
    }  
    (s->val)--;  
    spinunlock(&(s->sl));  
}
```

**Problem:** race in spinlock of while loop and signal's spinlock.  
Bounded wait not guaranteed.  
Spinlocks are not good for a long wait.

# Semaphore implementation? - 3, idea

```
struct semaphore {  
    int val;  
    spinlock lk;  
};  
sem_init(semaphore *s, int initval) {  
    s->val = initval;  
    s->sl = 0;  
}  
block() {  
    put this current process on wait-q;  
    schedule();  
}  
  
wait(semaphore *s) {  
    spinlock(&(s->sl));  
    while(s->val <=0) {  
        Block();  
    }  
    (s->val)--;  
    spinunlock(&(s->sl));  
}  
signal(seamphore *s) {  
    spinlock(*(s->sl));  
    (s->val)++;  
    spinunlock(*(s->sl));  
}
```

# Semaphore implementation? - 3a

```
struct semaphore {  
    int val;  
    spinlock lk;  
    list l;  
};  
sem_init(semaphore *s, int initval) {  
    s->val = initval;  
    s->sl = 0;  
}  
block(semaphore *s) {  
    listappend(s->l, current);  
    schedule();  
}  
problem is that block() will be called  
without holding the spinlock and the  
access to the list is not protected.  
Note that - so far we have ignored changes  
to signal()
```

```
wait(semaphore *s) {  
    spinlock(&(s->sl));  
    while(s->val <=0) {  
        spinunlock(&(s->sl));  
        block(s);  
    }  
    (s->val)--;  
    spinunlock(&(s->sl));  
}  
signal(seamphore *s) {  
    spinlock(*(s->sl));  
    (s->val)++;  
    spinunlock(*(s->sl));  
}
```

# Semaphore implementation? - 3b

```
struct semaphore {  
    int val;  
    spinlock lk;  
    list l;  
};  
sem_init(semaphore *s, int initval) {  
    s->val = initval;  
    s->sl = 0;  
}  
block(semaphore *s) {  
    listappend(s->l, current);  
    spinunlock(&(s->sl));  
    schedule();  
}  
wait(semaphore *s) {  
    spinlock(&(s->sl));  
    while(s->val <=0) {  
        block(s);  
    }  
    (s->val)--;  
    spinunlock(&(s->sl));  
}  
signal(semaphore *s) {  
    spinlock(*(s->sl));  
    (s->val)++;  
    x = dequeue(s->sl) and enqueue(readyq, x);  
    spinunlock(*(s->sl));  
}  
Problem: after a blocked process comes out  
of the block, it does not hold the spinlock and  
it's going to change the s->sl;
```

# Semaphore implementation? - 3c

```
struct semaphore {  
    int val;  
    spinlock lk;  
    list l;  
};  
sem_init(semaphore *s, int initval) {  
    s->val = initval;  
    s->sl = 0;  
}  
block(semaphore *s) {  
    listappend(s->l, current);  
    spinunlock(&(s->sl));  
    schedule();  
}  
  
wait(semaphore *s) {  
    spinlock(&(s->sl)); // A  
    while(s->val <=0) {  
        block(s);  
        spinlock(&(s->sl)); // B  
    }  
    (s->val)--;  
    spinunlock(&(s->sl));  
}  
signal(semaphore *s) {  
    spinlock(*(s->sl));  
    (s->val)++;  
    x = dequeue(s->sl) and enqueue(readyq, x);  
    spinunlock(*(s->sl));  
}  
  
Question: there is race between A and B. Can we guarantee bounded wait ?
```

# 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 in Linux

```
struct semaphore {  
    raw_spinlock_t    lock;  
    unsigned int      count;  
    struct list_head  wait_list;  
};  
  
static noinline void __sched  
__down(struct semaphore *sem)  
{  
    __down_common(sem,  
    TASK_UNINTERRUPTIBLE,  
    MAX_SCHEDULE_TIMEOUT);  
}  
  
void down(struct semaphore *sem)  
{  
    unsigned long flags;  
  
    raw_spin_lock_irqsave(&sem->lock, flags);  
    if (likely(sem->count > 0))  
        sem->count--;  
    else  
        __down(sem);  
    raw_spin_unlock_irqrestore(&sem->lock,  
    flags);  
}
```

# Semaphore in Linux

```
static inline int __sched
__down_common(struct semaphore
*sem, long state, long timeout)
{
    struct task_struct *task = current;
    struct semaphore_waiter waiter;
    list_add_tail(&waiter.list, &sem->wait_list);
    waiter.task = task;
    waiter.up = false;
```

```
    for (;;) {
        if (signal_pending_state(state, task))
            goto interrupted;
        if (unlikely(timeout <= 0))
            goto timed_out;
        __set_task_state(task, state);
        raw_spin_unlock_irq(&sem->lock);
        timeout = schedule_timeout(timeout);
        raw_spin_lock_irq(&sem->lock);
        if (waiter.up)
            return 0;
    }
....
```

# **Different uses of semaphores**

# For mutual exclusion

**/\*During initialization\*/**

**semaphore sem;**

**initsem (&sem, 1);**

**/\* On each use\*/**

**P (&sem);**

**Use resource;**

**V (&sem);**

# Event-wait

```
/* During initialization */
semaphore event;
initsem (&event, 0); /* probably at boot time */

/* Code executed by thread that must wait on event */
P (&event); /* Blocks if event has not occurred */
/* Event has occurred */
V (&event); /* So that another thread may wake up */
/* Continue processing */

/* Code executed by another thread when event occurs */
V (&event); /* Wake up one thread */
```

# Control countable resources

**/\* During initialization \*/**

semaphore counter;

initsem (&counter, resourceCount);

**/\* Code executed to use the resource \*/**

P (&counter); /\* Blocks until resource is available \*/

Use resource; /\* Guaranteed to be available now \*/

V (&counter); /\* Release the resource \*/

# Drawbacks of semaphores

- Need to be implemented using lower level primitives like spinlocks
- Context-switch is involved in blocking and signaling – time consuming
- Can not be used for a short critical section

# **Deadlocks**

# 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

P0

**wait (S);**

**wait (Q);**

· ·  
· ·  
· ·

**signal (S);**

**signal (Q);**

P1

**wait (Q);**

**wait (S);**

**signal (Q);**

**signal (S);**

# Example of deadlock

- Let's see the pthreads program : `deadlock.c`
- Same programme as on earlier slide, but with `pthread_mutex_lock();`

# Non-deadlock, but similar situations

- **Starvation – indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion**
  - Scheduling problem when lower-priority process holds a lock needed by higher-priority process (so it can not pre-empt lower priority process), and a medium priority process (that does not need the lock) pre-empts lower priority task, denying turn to higher priority task
  - Solved via priority-inheritance protocol : temporarily enhance priority of lower priority task to highest

# Livelock

- **Similar to deadlock, but processes keep doing ‘useless work’**
- **E.g. two people meet in a corridor opposite each other**
  - Both move to left at same time
  - Then both move to right at same time
  - Keep Repeating!
- **No process able to progress, but each doing ‘some work’ (not sleeping/waiting), state keeps changing**

# Livelock example

```
#include <stdio.h>
#include <pthread.h>
struct person {
    int otherid;
    int otherHungry;
    int myid;
};
int main() {
    pthread_t th1, th2;
    struct person one, two;
    one.otherid = 2; one.myid = 1;
    two.otherid = 1; two.myid = 2;
    one.otherHungry = two.otherHungry = 1;
    pthread_create(&th1, NULL, eat, &one);
    pthread_create(&th2, NULL, eat, &two);
    printf("Main: Waiting for threads to get over\n");
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    return 0;
}
```

```
/* thread two runs in this function */
int spoonWith = 1;
void *eat(void *param)
{
    int eaten = 0;
    struct person person= *(struct person *)param;
    while (!eaten) {
        if(spoonWith == person.myid)
            printf("%d going to eat\n", person.myid);
        else
            continue;
        if(person.otherHungry) {
            printf("You eat %d\n", person.otherid);
            spoonWith = person.otherid;
            continue;
        }
        printf("%d is eating\n", person.myid);
        break;
    }
}
```

# More on deadlocks

- Under which conditions they can occur?
- How can deadlocks be avoided/prevented?
- How can a system recover if there is a deadlock ?

# **System model for understanding deadlocks**

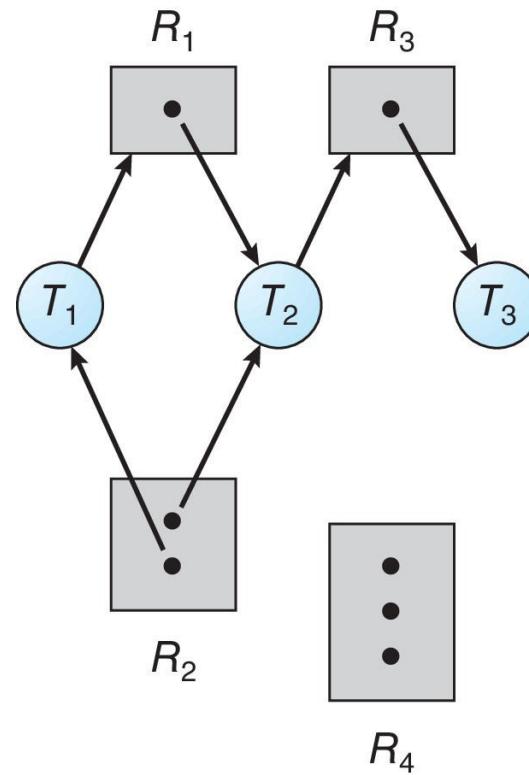
- **System consists of resources**
- **Resource types R<sub>1</sub>, R<sub>2</sub>, . . . , R<sub>m</sub>**
  - CPU cycles, memory space, I/O devices
  - Resource: Most typically a lock, synchronization primitive
- **Each resource type R<sub>i</sub> has W<sub>i</sub> instances.**
- **Each process utilizes a resource as follows:**
  - request
  - use
  - release

# Deadlock characterisation

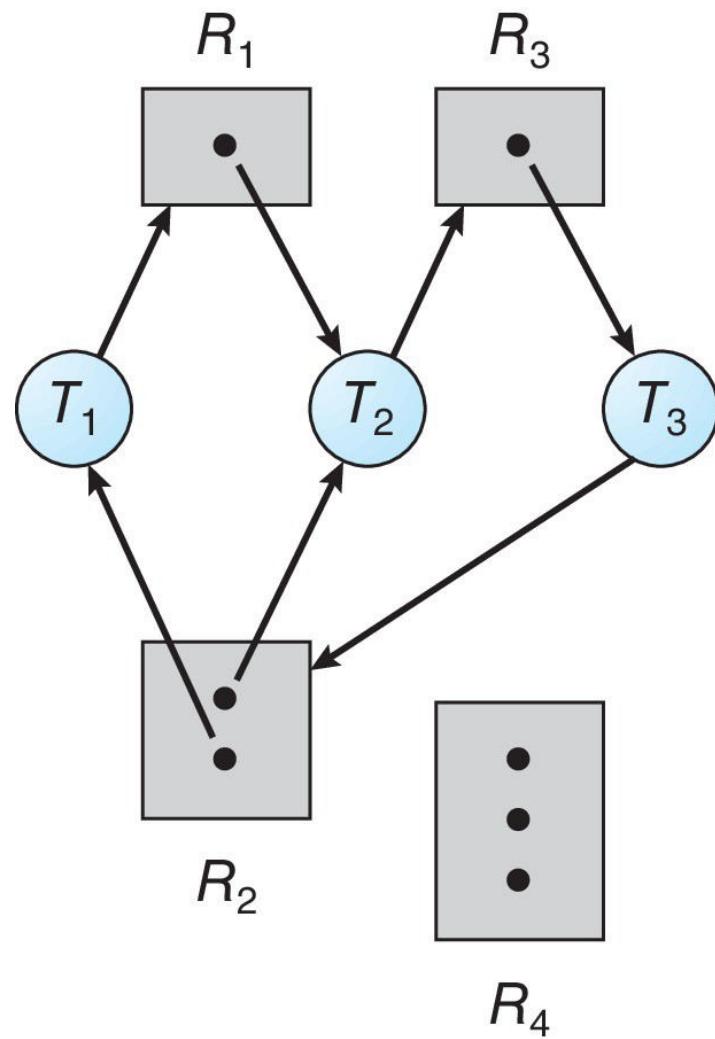
- **Deadlock is possible only if ALL of these conditions are TRUE at the same time**
  - **Mutual exclusion:** only one process at a time can use a resource
  - **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
  - **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
  - **Circular wait:** there exists a set  $\{P_0, P_1, \dots, P_n\}$  of waiting processes such that  $P_0$  is waiting for a resource that is held by  $P_1$ ,  $P_1$  is waiting for a resource that is held by  $P_2, \dots, P_{n-1}$  is waiting for a resource that is held by  $P_n$ , and  $P_n$  is waiting for a resource that is held by  $P_0$ .

# Resource Allocation Graph Example

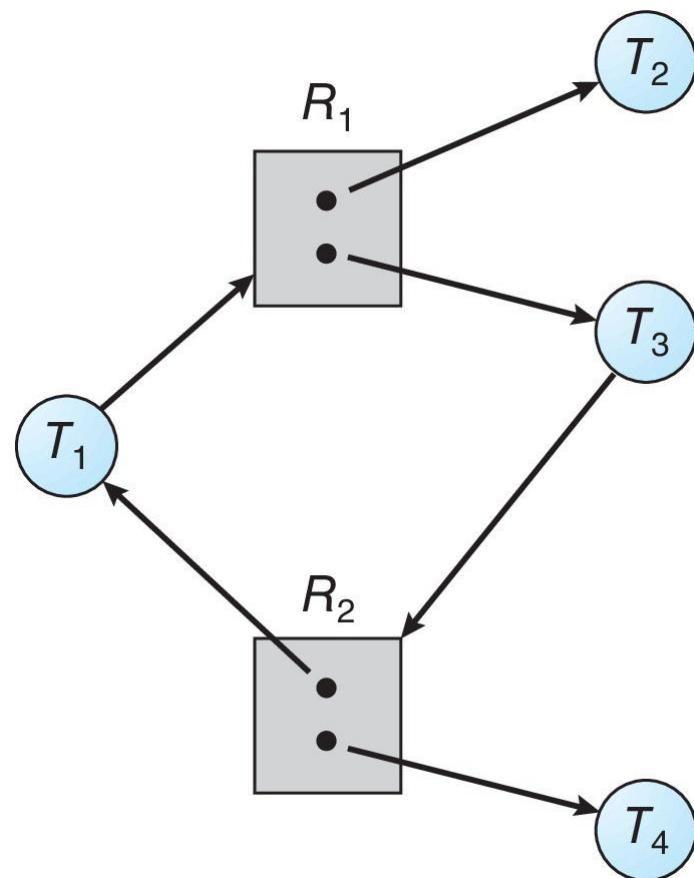
- One instance of R1
- Two instances of R2
- One instance of R3
- Three instances of R4
- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
- T3 is holding one instance of R3



# Resource Allocation Graph with a Deadlock



# Graph with a Cycle But no Deadlock



# Basic Facts

- **If graph contains no cycles -> no deadlock**
- **If graph contains a cycle :**
  - **if only one instance per resource type, then deadlock**
  - **if several instances per resource type, possibility of deadlock**

# Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state:
  - 1) Deadlock prevention
  - 2) Deadlock avoidance
  - 3) Allow the system to enter a deadlock state and then recover
  - 4) Ignore the problem and pretend that deadlocks never occur in the system.

# (1) Deadlock Prevention

- **Invalidate one of the four necessary conditions for deadlock:**
- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
  - Low resource utilization; starvation possible

# (1) Deadlock Prevention (Cont.)

- **No Preemption:**
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait:**
  - Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

# (1) Deadlock prevention: Circular Wait

- **Invalidate the circular wait condition is most common.**
- **Simply assign each resource (i.e., mutex locks) a unique number.**
- **Resources must be acquired in order.**
- **If:**

**first\_mutex = 1**

**second\_mutex = 5**

**code for thread\_two could not be written as follows:**

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);

    pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);

    pthread_exit(0);
}
```

# (1) Preventing deadlock: cyclic wait

- **Locking hierarchy : Highly preferred technique in kernels**
  - Decide an ordering among all ‘locks’
  - Ensure that on ALL code paths in the kernel, the locks are obtained in the decided order!
  - Poses coding challenges!
  - A key differentiating factor in kernels
  - Do not look at only the current lock being taken, look at all the locks the code may be holding at any given point in code!

# **(1) Prevention in Xv6: Lock Ordering**

- **lock on the directory, a lock on the new file's inode, a lock on a disk block buffer, idelock, and ptable.lock.**

## (2) Deadlock avoidance

- **Requires that the system has some additional a priori information available**
  - Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need
  - The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
  - Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes

## (2) Deadlock avoidance

- Please see: concept of safe states, unsafe states, Banker's algorithm

# (3) Deadlock detection and recovery

- How to detect a deadlock in the system?
- The Resource-Allocation Graph is a graph. Need an algorithm to detect cycle in a graph.
- How to recover?
  - Abort all processes or abort one by one?
  - Which processes to abort?
    - Priority ?
    - Time spent since forked()?
    - Resources used?
    - Resources needed?
    - Interactive or not?
    - How many need to be terminated?

# **“Condition” Synchronization Tool**

# What is condition variable?

- A variable with a sleep queue
- Threads can sleep on it, and wake-up all remaining

Struct condition {

    Proc \*next

    Proc \*prev

    Spinlock \*lock

}

Different variables of this type can be used as different  
'conditions'

# Code for condition variables

```
//Spinlock s is held before calling wait
void wait (condition *c, spinlock_t *s)
{
    spin_lock (&c->listLock);
    add self to the linked list;
    spin_unlock (&c->listLock);
    spin_unlock (s); /* release
spinlock before blocking */
    swtch(); /* perform context switch */
    /* When we return from swtch, the
event has occurred */
    spin_lock (s); /* acquire the spin
lock again */
    return;
}
```

```
void do_signal (condition *c)
/*Wakeup one thread waiting on the condition*/
{
    spin_lock (&c->listLock);
    remove one thread from linked list, if it is nonempty;
    spin_unlock (&c->listLock);
    if a thread was removed from the list, make it
        runnable;
    return;
}
void do broadcast (condition *c)
/*Wakeup all threads waiting on the condition*/
{
    spin_lock (&c->listLock);
    while (linked list is nonempty) {
        remove a thread from linked list;
        make it runnable;
    }
    spin_unlock (&c->listLock);
}
```

# Semaphore implementation using condition variables?

- Is this possible?
- Can we try it?

```
typedef struct semaphore {
```

```
    //something  
    condition c;  
}semaphore;
```

- Now write code for semaphore P() and V()

# **Classical Synchronization Problems**

# Bounded-Buffer Problem

- **Producer and consumer processes**
  - N buffers, each can hold one item
- **Producer produces ‘items’ to be consumed by consumer , in the bounded buffer**
- **Consumer should wait if there are no items**
- **Producer should wait if the ‘bounded buffer’ is full**

# Bounded-Buffer Problem: solution with semaphores

- **Semaphore mutex initialized to the value 1**
- **Semaphore full initialized to the value 0**
- **Semaphore empty initialized to the value N**

# Bounded-buffer problem

The structure of the producer process

```
do {  
    // produce an item in nextp  
    wait (empty);  
    wait (mutex);  
    // add the item to the buffer  
    signal (mutex);  
    signal (full);  
} while (TRUE);
```

The structure of the Consumer process

```
do {  
    wait (full);  
    wait (mutex);  
    // remove an item from  
    // buffer to nextc  
    signal (mutex);  
    signal (empty);  
    // consume item in nextc  
} while (TRUE);
```

# Bounded buffer problem

- Example : pipe()
- Let's see code of pipe in xv6 – a solution using sleeplocks

# Readers-Writers problem

- **A data set is shared among a number of concurrent processes**
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write
- **Problem – allow multiple readers to read at the same time**
  - Only one single writer can access the shared data at the same time
- **Several variations of how readers and writers are treated – all involve priorities**
- **Shared Data**
  - Data set
  - Semaphore mutex initialized to 1
  - Semaphore wrt initialized to 1
  - Integer readcount initialized to 0

## The structure of a writer process

```
do {  
    wait (wrt) ;  
    // writing is performed  
    signal (wrt) ;  
} while (TRUE);
```

## The structure of a reader process

```
do {  
    wait (mutex) ;  
    readcount ++ ;  
    if (readcount == 1)  
        wait (wrt) ;  
    signal (mutex)  
    // reading is performed  
    wait (mutex) ;  
    readcount -- ;  
    if (readcount == 0)  
        signal (wrt) ;  
    signal (mutex) ;  
} while (TRUE);
```

# Readers-Writers problem

# **Readers-Writers Problem Variations**

- **First variation – no reader kept waiting unless writer has permission to use shared object**
- **Second variation – once writer is ready, it performs write asap**
- **Both may have starvation leading to even more variations**
- **Problem is solved on some systems by kernel providing reader-writer locks**

# Reader-write lock

- A lock with following operations on it
  - Lockshared()
  - Unlockshared()
  - LockExcl()
  - UnlockExcl()
- Possible additions
  - Downgrade() -> from excl to shared
  - Upgrade() -> from shared to excl

# Code for reader-writer locks

```
struct rwlock {  
    int nActive; /* num of active  
readers, or -1 if a writer is  
active */  
  
    int nPendingReads;  
    int nPendingWrites;  
    spinlock_t sl;  
    condition canRead;  
    condition canWrite;  
};
```

```
void lockShared (struct rwlock *r)  
{  
    spin_lock (&r->sl);  
    r->nPendingReads++;  
    if (r->nPendingWrites > 0)  
        wait (&r->canRead, &r->sl ); /*don't starve  
writers */  
    while ({r->nActive < 0) /* someone has  
exclusive lock */  
        wait (&r->canRead, &r->sl);  
    r->nActive++;  
    r->nPendingReads--;  
    spin_unlock (&r->sl);  
}
```

# Code for reader-writer locks

```
void unlockShared (struct rwlock
*r)
{
    spin_lock (&r->sl);
    r->nActive--;
    if (r->nActive == 0) {
        spin_unlock (&r->sl);
        do signal (&r->canWrite);
    } else
        spin_unlock (&r->M);
}
```

```
void lockExclusive (struct rwlock
*r)
{
    spin_lock (&r->sl);
    r->nPendingWrtes++;
    while (r->nActive)
        wait (&r->canWrite, &r->sl);
    r->nPendingWrites--;
    r->nActive = -1;
    spin_unlock (&r->sl);
}
```

# Code for reader-writer locks

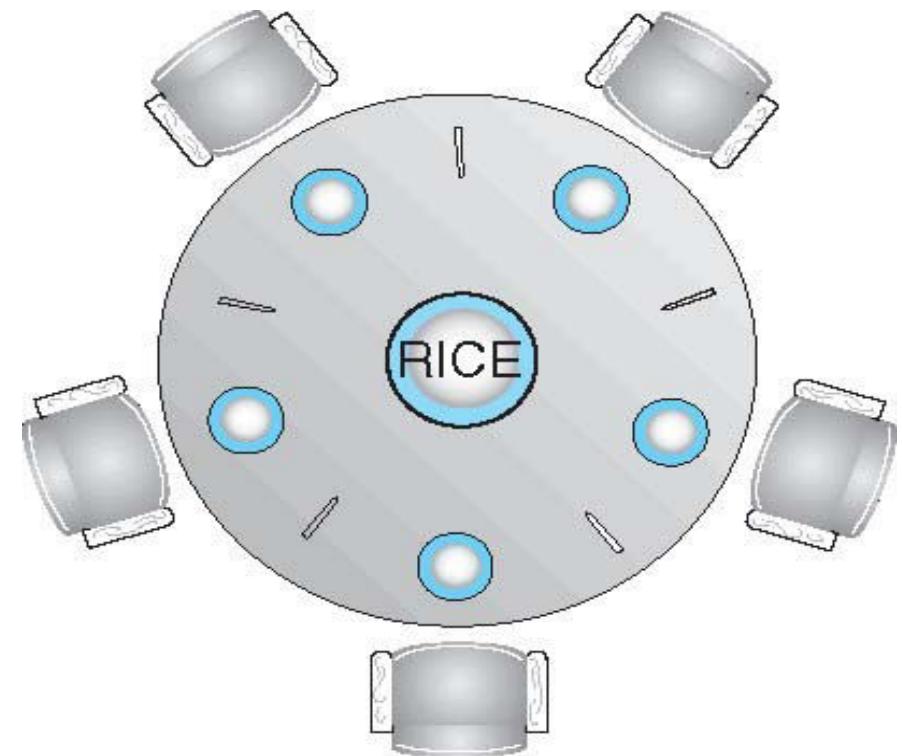
```
void unlockExclusive (struct rwlock *r){  
    boolean t wakeReaders;  
    spin_lock (&r->sl);  
    r->nActive = 0;  
    wakeReaders = (r->nPendingReads != 0);  
    spin_unlock (&r->sl);  
    if (wakeReaders)  
        do broadcast (&r->canRead); /* wake  
allreaders */  
    else  
        do_signal (&r->canWrite);  
/*wakeupsinglewriter */  
}
```

Try writing code for  
downgrade and  
upgrade

Try writing a reader-  
writer lock using  
semaphores!

# Dining-Philosophers Problem

- Philosophers spend their lives thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1



# Dining philosophers: One solution

The structure of Philosopher i:

```
do {  
    wait ( chopstick[i] );  
    wait ( chopStick[ (i + 1) % 5] );  
    // eat  
    signal ( chopstick[i] );  
    signal (chopstick[ (i + 1) % 5] );  
    // think  
} while (TRUE);
```

What is the problem with this algorithm?

# Dining philosophers: Possible approaches

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available
  - to do this, she must pick them up in a critical section
- Use an asymmetric solution
  - that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick
  - whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick.

# Other solutions to dining philosopher's problem

- Using higher level synchronization primitives like 'monitors'
-

# **Practical Problems**

# Lost Wakeup problem

- **The sleep/wakeup mechanism does not function correctly on a multiprocessor.**
- **Consider a potential race:**
  - Thread T1 has locked a resource R1.
  - Thread T2, running on another processor, tries to acquire the resource, and finds it locked.
  - T2 calls sleep() to wait for the resource.
  - Between the time T2 finds the resource locked and the time it calls sleep(), T1 frees the resource and proceeds to wake up all threads blocked on it.
  - Since T2 has not yet been put on the sleep queue, it will miss the wakeup.
  - The end result is that the resource is not locked, but T2 is blocked waiting for it to be unlocked.
  - If no one else tries to access the resource, T2 could block indefinitely.
  - This is known as the lost wakeup problem,
- **Requires some mechanism to combine the test for the resource and the call to sleep() into a single atomic operation.**

# Lost Wakeup problem

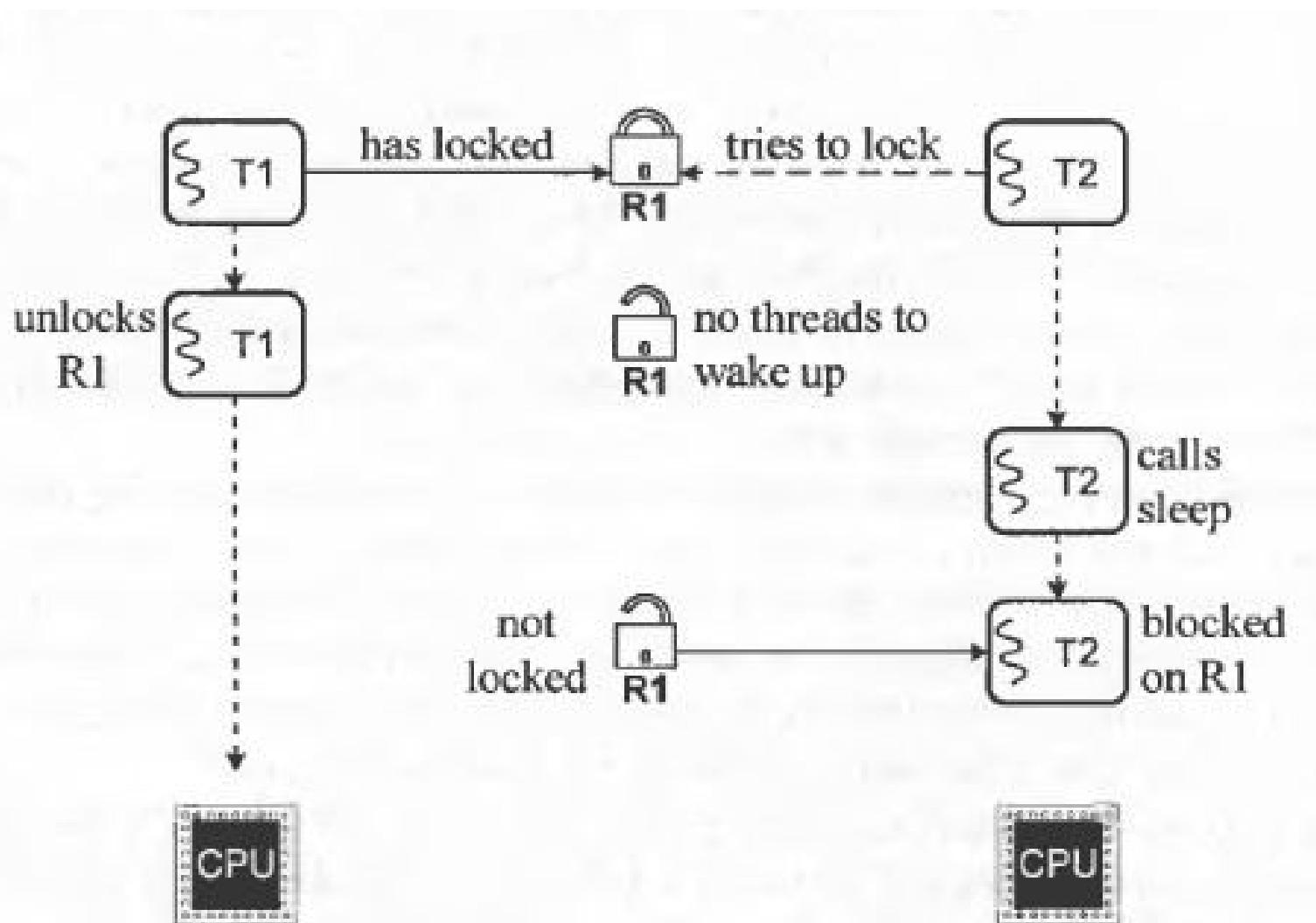


Figure 7-6. The lost wakeup problem.

# Thundering herd problem

- **Thundering Herd problem**
  - On a multiprocessor, if several threads were locked the resource
  - Waking them all may cause them to be simultaneously scheduled on different processors
  - and they would all fight for the same resource again.
- **Starvation**
  - Even if only one thread was blocked on the resource, there is still a time delay between its waking up and actually running.
  - In this interval, an unrelated thread may grab the resource causing the awakened thread to block again. If this happens frequently, it could lead to starvation of this thread.
  - This problem is not as acute on a uniprocessor, since by the time a thread runs, whoever had locked the resource is likely to have released it.

# **Case Studies**

# Linux Synchronization

- Prior to kernel Version 2.6, disables interrupts to implement short critical sections
- Version 2.6 and later, fully preemptive
- Linux provides:
  - semaphores
  - spinlocks
  - reader-writer versions of both
  - Atomic integers
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

# Linux Synchronization

- **Atomic variables**

- atomic\_t** is the type for atomic integer

- **Consider the variables**

- atomic\_t counter;**

- int value;**

| <i>Atomic Operation</i>        | <i>Effect</i>          |
|--------------------------------|------------------------|
| atomic_set(&counter,5);        | counter = 5            |
| atomic_add(10,&counter);       | counter = counter + 10 |
| atomic_sub(4,&counter);        | counter = counter - 4  |
| atomic_inc(&counter);          | counter = counter + 1  |
| value = atomic_read(&counter); | value = 12             |

# Pthreads synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spinlocks

# **Synchronization issues in xv6 kernel**

# Difference approaches

- **Pros and Cons of locks**
  - Locks ensure serialization
  - Locks consume time !
- **Solution – 1**
  - One big kernel lock
  - Too inefficient
- **Solution – 2**
  - One lock per variable
  - Often un-necessary, many data structures get manipulated in once place, one lock for all of them may work
- **Problem: ptable.lock for the entire array and every element within**
  - Alternatively: one lock for array, one lock per array entry

# Three types of code

- **System calls code**
  - Can it be interruptible?
  - If yes, when?
- **Interrupt handler code**
  - Disable interrupts during interrupt handling or not?
  - Deadlock with iderw ! - already seen
- **Process's user code**
  - Ignore. Not concerned with it now.

# Interrupts enabling/disabling in xv6

- **Holding every spinlock disables interrupts!**
- **System call code or Interrupt handler code won't be interrupted if**
  - The code path followed took at least once spinlock !
  - Interrupts disabled only on that processor!
- **Acquire calls pushcli() before xchg()**
- **Release calls popcli() after xchg()**

# Memory ordering

- Compiler may generate machine code for out-of-order execution !
- Processor pipelines can also do the same!
- This often improves performance
- Compiler may reorder 4 after 6 --> Trouble!
- Solution: Memory barrier
  - `__sync_synchronize()`, provided by GCC
  - Do not reorder across this line
  - Done only on acquire and release()

- Consider this

- 1) `l = malloc(sizeof *l);`
- 2) `l->data = data;`
- 3) `acquire(&listlock);`
- 4) `l->next = list;`
- 5) `list = l;`
- 6) `release(&listlock);`

# Lost Wakeup?

- **Do we have this problem in xv6?**
- **Let's analyze again!**
  - The race in `acquiresleep()`'s call to `sleep()` and `releasesleep()`
- **T1 holding lock, T2 willing to acquire lock**
  - Both running on different processor
  - Or both running on same processor
  - What happens in both scenarios?
- **Introduce a T3 and T4 on each of two different processors. Now how does the scenario change?**
- **See page 69 in xv6 book revision-11.**

# Code of sleep()

```
if(lk != &ptable.lock){  
    acquire(&ptable.lock);  
    release(lk);  
}
```

- Why this check?
- Deadlock otherwise!
- Check: wait() calls with ptable.lock held!

# Exercise question : 1

Sleep has to check lk != &ptable.lock to avoid a deadlock

Suppose the special case were eliminated by replacing

```
if(lk != &ptable.lock){  
    acquire(&ptable.lock);  
    release(lk);  
}
```

with

```
release(lk);  
acquire(&ptable.lock);
```

Doing this would break sleep. How?

`

# bget() problem

- **bget() panics if no free buffers!**
- **Quite bad**
- **Should sleep !**
- **But that will introduce many deadlock problems. Which ones ?**

# **iget() and ilock()**

- **iget() does no hold lock on inode**
- **Illock() does**
- **Why this separation?**
  - Performance? If you want only “read” the inode, then why lock it?
- **What if iget() returned the inode locked?**

# Interesting cases in namex()

```
while((path = skipel(path, name)) != 0){
    ilock(ip);
    if(ip->type != T_DIR){
        iunlockput(ip);
        return 0;
    }
    if(nameiparent && *path == '\0'){
        // Stop one level early.
        iunlock(ip);
        return ip;
    }
    if((next = dirlookup(ip, name, 0)) == 0){
        iunlockput(ip);
        return 0;
    }
    iunlock(ip);
    ip = next;
}
--> only after obtaining next from
dirlookup() and iget() is the lock
released on ip;
-> lock on next obtained only after
releasing the lock on ip. Deadlock
possible if next was “.”
```

Xv6

Interesting case of holding and releasing  
ptable.lock in scheduling

**One process acquires, another releases!**

# Giving up CPU

- A process that wants to give up the CPU
  - must acquire the process table lock ptable.lock
  - release any other locks it is holding
  - update its own state (proc->state),
  - and then call sched()
- Yield follows this convention, as do sleep and exit
- Lock held by one process P1, will be released another process P2 that starts running after sched()
  - remember P2 returns either in yield() or sleep()
  - In both, the first thing done is releasing ptable.lock

# Interesting race if ptable.lock is not held

- Suppose P1 calls yield()
- Suppose yield() does not take ptable.lock
  - Remember yield() is for a process to give up CPU
- Yield sets process state of P1 to RUNNABLE
- Before yield's sched() calls swtch()
- Another processor runs scheduler() and runs P1 on that processor
- Now we have P1 running on both processors!
- P1 in yield taking ptable.lock prevents this

# Homework

- **Read the version-11 textbook of xv6**
- **Solve the exercises!**

# Notes on reading xv6 code

Abhijit A. M.

[abhijit.comp@coep.ac.in](mailto:abhijit.comp@coep.ac.in)

Credits:

xv6 book by Cox, Kaashoek, Morris

Notes by Prof. Sorav Bansal

# **Introduction to xv6**

## **Structure of xv6 code**

## **Compiling and executing xv6 code**

# About xv6

- Unix Like OS
- Multi tasking, Single user
- On x86 processor
- Supports some system calls
- Small code, 7 to 10k
- Meant for learning OS concepts
- **No** : demand paging, no copy-on-write fork, no shared-memory, fixed size stack for user programs

# Use cscope and ctags with VIM

- Go to folder of xv6 code and run

```
cscope -q *. [chS]
```

- Also run

```
ctags *. [chS]
```

- Now download the file

[http://cscope.sourceforge.net/cscope\\_maps.vim](http://cscope.sourceforge.net/cscope_maps.vim) as  
.cscope\_maps.vim in your ~ folder

- And add line “source ~/.cscope\_maps.vim” in your  
~/.vimrc file

- Read this tutorial

[http://cscope.sourceforge.net/cscope\\_vim\\_tutorial.html](http://cscope.sourceforge.net/cscope_vim_tutorial.html)

# Use call graphs (using doxygen)

- Doxygen – a documentation generator.
- Can also be used to generate “call graphs” of functions
- Download xv6
- Install doxygen on your Ubuntu machine.
- cd to xv6 folder
- Run “doxygen -g doxyconfig”
- This creates the file “doxyconfig”

# Use call graphs (using doxygen)

>Create a folder “doxygen”

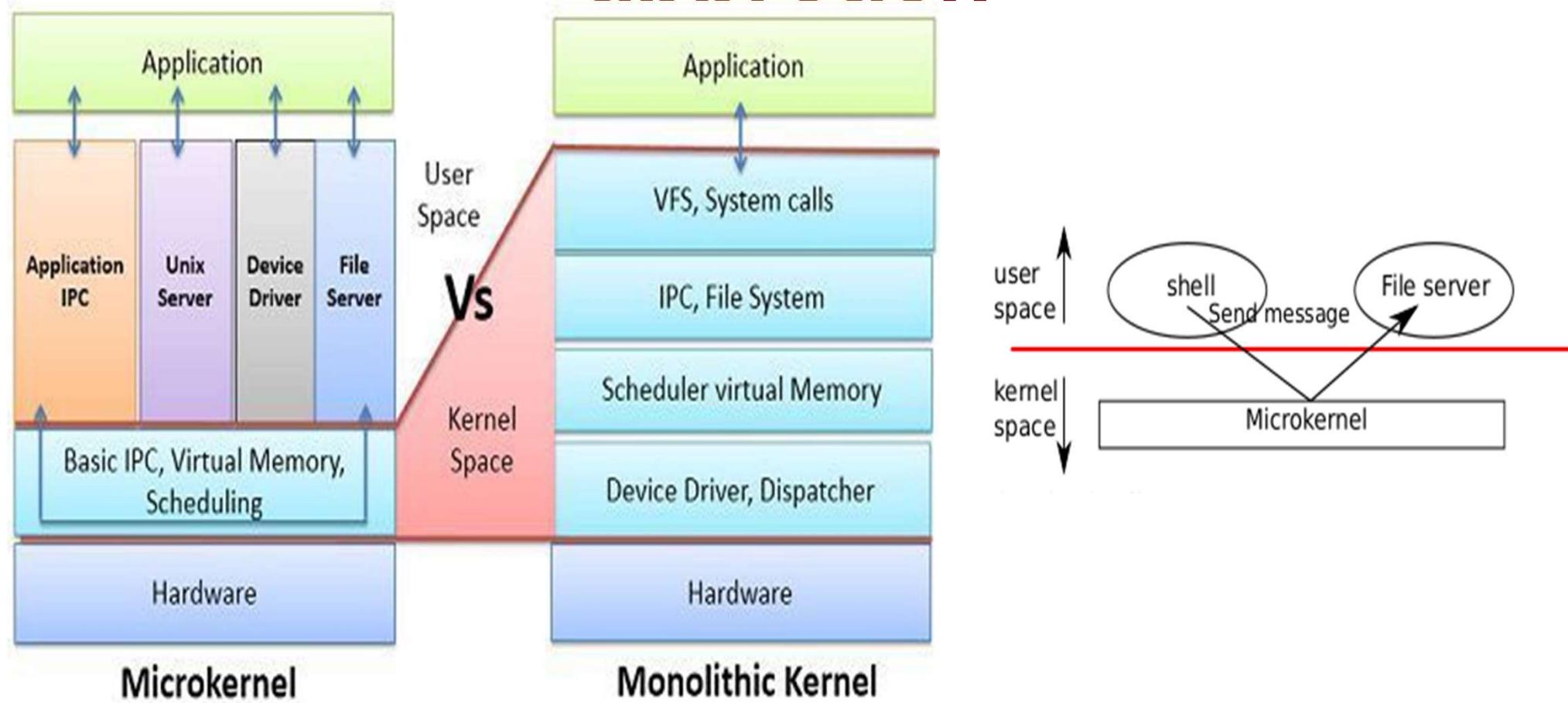
Open “doxyconfig” file and make these changes.

```
PROJECT_NAME          = "XV6"
OUTPUT_DIRECTORY      = ./doxygen
CREATE_SUBDIRS        = YES
EXTRACT_ALL           = YES
EXCLUDE               = usertests.c cat.c yes.c echo.c forktest.c grep.c
init.c kill.c ln.c ls.c mkdir.c rm.c sh.c stressfs.c wc.c zombie.c
CALL_GRAPH            = YES
CALLER_GRAPH          = YES
```

Now run “doxygen doxyconfig”

Go to “doxygen”/html and open “firefox index.html” --> See call graphs in files -> any file

# Xv6 follows monolithic kernel approach



# qemu

- A virtual machine manager, like Virtualbox

- Qemu provides us

- BIOS

- Virtual CPU, RAM, Disk controller, Keyboard controller

- IOAPIC, LAPIC

- Qemu runs xv6 using this command

```
qemu -serial mon:stdio -drive  
file=fs.img,index=1,media=disk,format=raw -drive  
file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
```

- Invoked when you run “make qemu”

# qemu

## ■ Understanding qemu command

■ **-serial mon:stdio**

■ the window of xv6 is also multiplexed in your normal terminal.

■ Run “make qemu”, then Press “Ctrl-a” and “c” in terminal and you get qemu prompt

■ **-drive file=fs.img,index=1,media=disk,format=raw**

■ Specify the hard disk in “fs.img”, accessible at first slot in IDE(or SATA, etc), as a “disk”, with “raw” format

■ **-smp 2**

■ Two cores in SMP mode to be simulated

■ **-m 512**

■ Use 512 MB ram

# About files in XV6 code

- `cat.c echo.c forktest.c grep.c init.c  
kill.c ln.c ls.c mkdir.c rm.c sh.c  
stressfs.c usertests.c wc.c yes.c  
zombie.c`

- **User programs for testing xv6**

- **Makefile**

- **To compile the code**

- **dot-bochsrc**

- **For running with emulator bochs**

# About files in XV6 code

- bootasm.S entryother.S entry.S  
initcode.S swtch.S trapasm.S  
usys.S

- Kernel code written in Assembly. Total 373 lines

- kernel.ld

- Instructions to Linker, for linking the kernel properly

- README Notes LICENSE

- Misc files

# Using Makefile

- ❑ **make qemu**
  - ❑ Compile code and run using “qemu” emulator
- ❑ **make xv6.pdf**
  - ❑ Generate a PDF of xv6 code
- ❑ **make mkfs**
  - ❑ Create the mkfs program
- ❑ **make clean**
  - ❑ Remove all intermediary and final build files

# Files generated by Makefile

- ❑ .o files
  - ❑ Compiled from each .c file
  - ❑ No need of separate instruction in Makefile to create .o files
  - ❑ %: %.o \$(ULIB) line is sufficient to build each .o for a \_xyz file

# Files generated by Makefile

## asm files

Each of them has an equivalent object code file or C file. For example

```
bootblock: bootasm.S bootmain.c  
          $(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c  
bootmain.c  
          $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S  
          $(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o  
bootblock.o bootasm.o bootmain.o  
          $(OBJDUMP) -S bootblock.o > bootblock.asm  
          $(OBJCOPY) -S -O binary -j .text bootblock.o  
bootblock  
          ./sign.pl bootblock
```

# Files generated by Makefile

- ❑ `_ln`, `_ls`, etc

- ❑ Executable user programs

- ❑ Compilation process is explained after few slides

# Files generated by Makefile

- **xv6.img**
- Image of xv6 created
- **xv6.img: bootblock kernel**
  - **dd if=/dev/zero of=xv6.img count=10000**
  - **dd if=bootblock of=xv6.img conv=notrunc**
  - **dd if=kernel of=xv6.img seek=1 conv=notrunc**

# Files generated by Makefile

bootblock

```
bootblock: bootasm.S bootmain.c
          $(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c
bootmain.c
          $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c
bootasm.S
          $(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o
bootblock.o bootasm.o bootmain.o
          $(OBJDUMP) -S bootblock.o > bootblock.asm
          $(OBJCOPY) -S -O binary -j .text bootblock.o
bootblock
          ./sign.pl bootblock
```

# Files generated by Makefile

**kernel**

```
kernel: $(OBJS) entry.o entryother initcode kernel.1d  
        $(LD) $(LDFLAGS) -T kernel.1d -  
o kernel entry.o $(OBJS) -b binary  
initcode entryother  
        $(OBJDUMP) -S kernel >  
kernel.asm  
        $(OBJDUMP) -t kernel | sed  
'1,/SYMBOL TABLE/d; s/ .* //; /^$$/d'  
> kernel.sym
```

# Files generated by Makefile

- **fs.img**

- A disk image containing user programs and README

- **fs.img: mkfs README \$(UPROGS)**

- ./mkfs fs.img README \$(UPROGS)

- **.sym files**

- Symbol tables of different programs

- E.g. for file “kernel”

- **\$ (OBJDUMP) -t kernel | sed '1,/SYMBOL  
TABLE/d; s/ .\* / /; /^\$\$/d' > kernel.sym**

# Size of xv6 C code

- `wc *[ch] | sort -n`
- **10595 34249 278455 total**
- Out of which
- **738 4271 33514 dot-bochssrc**
- **wc cat.c echo.c forktest.c grep.c init.c kill.c ln.c ls.c mkdir.c rm.c sh.c stressfs.c usertests.c wc.c yes.c zombie.c**
- **2849 6864 51993 total**
- So total code is  $10595 - 2849 - 738 = 7008$  lines

# List of commands to try (in given order)

**usertests #** Runs lot of tests and takes upto 10 minutes to run

**stressfs #** opens , reads and writes to files in parallel

**ls #** out put is filetyp, inode number, type

**cat README**

**ls;ls**

**cat README | grep BUILD**

**echo hi there**

**echo hi there | grep hi**

**echo "hi there**

# List of commands to try (in this order)

`echo README | grep Wa`

`ls ../ # works from inside test`

`echo README | grep Wa |  
grep ty # does not work`

`cd # fails`

`cat README | grep Wa |  
grep bl # works`

`cd / # works`

`ls > out # takes time!`

`wc README`

`mkdir test`

`rm out`

`cd test`

`ls . test # listing both  
directories`

`ls # fails`

`In cat xyz; ls`

`rm xyz; ls`

# User Libraries: Used to link user land programs

## ↳ Ulib.c

↳ **Strcpy, strcmp,strlen, memset, strchr, stat, atoi, memmove**

↳ **Stat uses open()**

↳ **Usys.S -> compiles into usys.o**

↳ **Assembly code file.** Basically converts all calls like open() (e.g. used in ulib.c) into assembly code using “int” instruction.

↳ **Run following command see the last 4 lines in the output**

```
objdump -d usys.o
```

↳ **00000048 <open>:**

|       |                |     |            |
|-------|----------------|-----|------------|
| ↳ 48: | b8 0f 00 00 00 | mov | \$0xf,%eax |
| ↳ 4d: | cd 40          | int | \$0x40     |
| ↳ 4f: | c3             | ret |            |

# User Libraries: Used to link user land programs

- **printf.c**
- **Code for printf()!**
- **Interesting to read this code.**
- **Uses variable number of arguments. Normal technique in C is to use va\_args library, but here it uses pointer arithmetic.**
- **Written using two more functions: printint() and putc() - both call write()**
- **Where is code for write()?**

# User Libraries: Used to link user land programs

## ❑ umalloc.c

- ❑ This is an implementation of malloc() and free()
- ❑ Almost same as the one done in “The C Programming Language” by Kernighan and Ritchie
- ❑ Uses sbrk() to get more memory from xv6 kernel

# Understanding the build process in more details

¶Run

```
make qemu | tee make-output.txt
```

¶You will get all compilation commands in  
**make-output.txt**

# Compiling user land programs

Normally when you compile a program on Linux

You compile it for the same ‘target’ machine (= CPU + OS)

The compiler itself runs on the same OS

To compile a user land program for xv6, we don’t have a compiler on xv6,

So we compile the programs (using make, cc) on Linux , for xv6

Obviously they can’t link with the standard libraries on Linux

# Compiling user land programs

```
ULIB = ulib.o usys.o printf.o umalloc.o

_%: %.o $(ULIB)

$(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^

$(OBJDUMP) -S $@ > $*.asm

$(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^\$/d' > $*.sym
```

\$@ is the name of the file being generated

\$^ is dependencies . i.e. \$(ULIB) and %.o in this case

# Compiling user land programs

```
gcc -fno-pic -static -fno-builtin -fno-strict-
aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-
frame-pointer -fno-stack-protector -fno-pie -no-pie
-c -o cat.o cat.c

ld -m      elf_i386 -N -e main -Ttext 0 -o _cat cat.o
ulib.o usys.o printf.o umalloc.o

objdump -S _cat > cat.asm

objdump -t _cat | sed '1,/SYMBOL TABLE/d; s/ .* / /;
/^$/d' > cat.sym
```

# Compiling user land programs

Mkfs is compiled like a Linux program !

```
gcc -Werror -Wall -o mkfs mkfs.c
```

# How to read kernel code ?

- Understand the data structures
- Know each global variable, typedefs, lists, arrays, etc.
- Know the purpose of each of them
- While reading a code path, e.g. exec()
- Try to ‘locate’ the key line of code that does major work
- Initially (but not forever) ignore the ‘error checking’ code
- Keep summarising what you have read
- Remembering is important !
- To understand kernel code, you should be good with concepts in OS , C, assembly, hardware

# Pre-requisites for reading the code

- Understanding of core concepts of operating systems
- Memory Management, processes, fork-exec, file systems, synchronization, x86 architecture, calling convention , computer organization
- 2 approaches:
  - 1) Read OS basics first, and then start reading xv6 code
    - Good approach, but takes more time !
  - 2) Read some basics, read xv6, repeat
    - Gives a headstart, but you will always have gaps in your understanding of the code, until you are done with everything
    - We normally follow this approach
    - Good knowledge of C, pointers, function pointers particularly
    - Data structures: doubly linked lists, queues, structures and pointers

# Processes in xv6 code

# Process Table

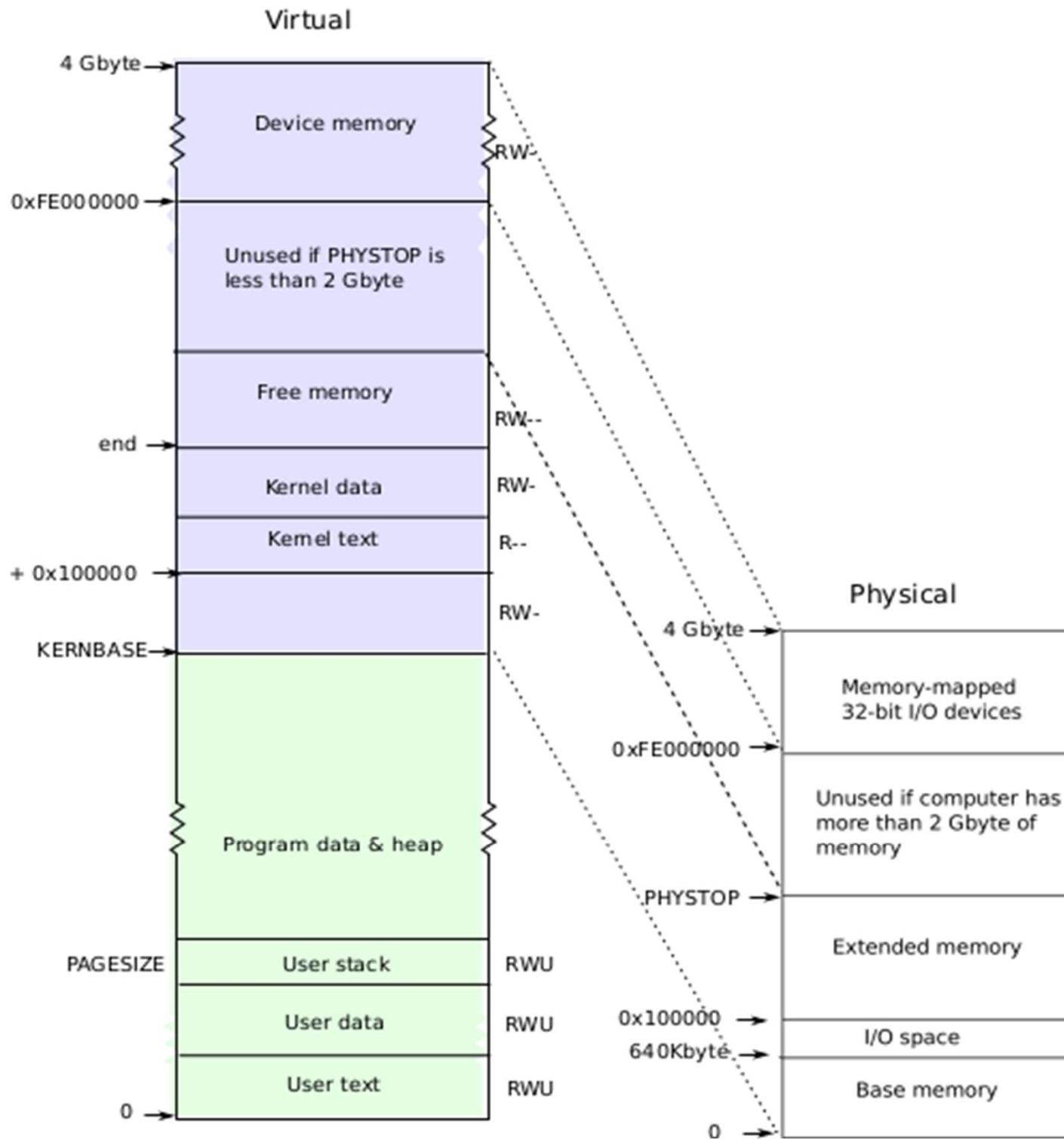
```
struct {  
    struct spinlock lock;  
    struct proc proc[NPROC];  
} ptable;
```

- One single global array of processes
- Protected by `ptable.lock`

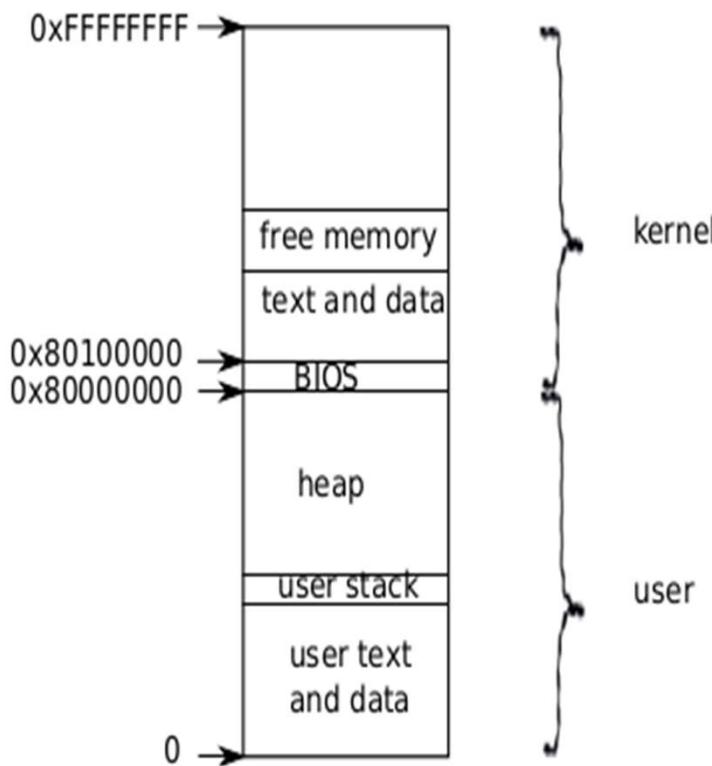
## Layout of process's VA space

xv6  
schema!

different  
from Linux



# Logical layout of memory for a process



- Address 0: code

- Then globals

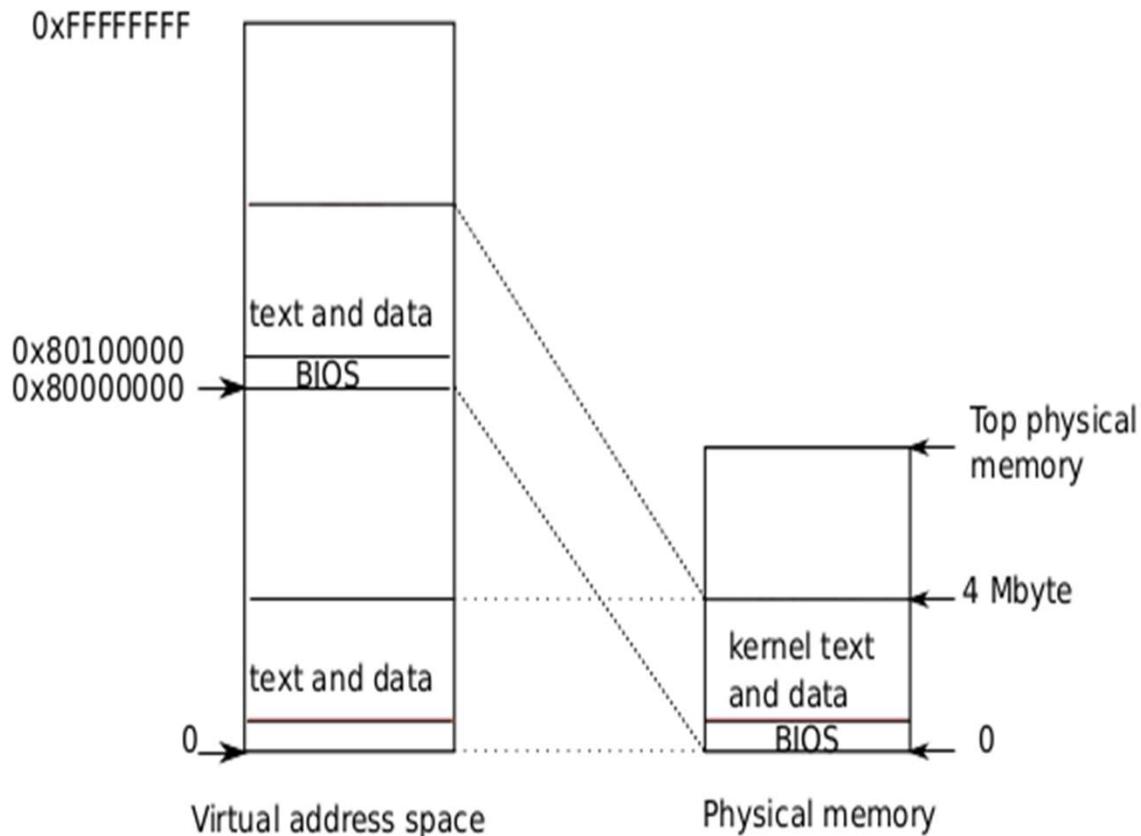
- Then stack

- Then heap

- Each process's address space maps kernel's text, data also --> so that system calls run with these mappings

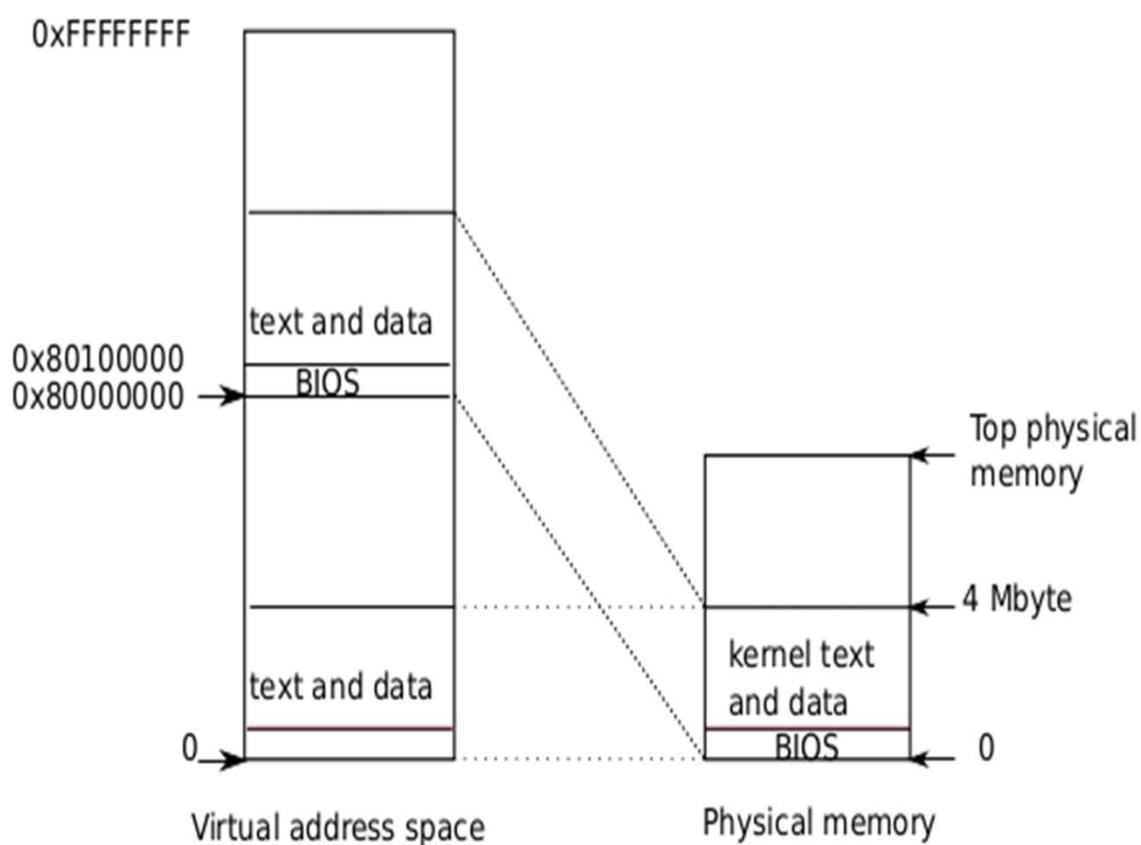
- Kernel code can directly access user memory now

# Kernel mappings in user address space actual location of kernel



- Kernel is loaded at 0x100000 physical address
- PA 0 to 0x100000 is BIOS and devices
- Process's page table will map
- VA 0x80000000 to PA 0x00000 and
- VA 0x8010000 to 0x100000

# Kernel mappings in user address space actual location of kernel



Kernel is not loaded at the PA 0x80000000 because some systems may not have that much memory

0x80000000 is called KERNBASE in xv6

# Imp Concepts

- A process has two stacks
  - user stack: used when user code is running
  - kernel stack: used when kernel is running on behalf of a process
- Note: there is a third stack also!
- The kernel stack used by the scheduler itself
- Not a per process stack

# Struct proc

```
// Per-process state

struct proc {

    uint sz;           // Size of process memory (bytes)

    pde_t* pgdir;     // Page table

    char *kstack;      // Bottom of kernel stack for this process

    enum procstate state; // Process state. allocated, ready to run, running, wait-
                          // ing for I/O, or exiting.

    int pid;          // Process ID

    struct proc *parent; // Parent process

    struct trapframe *tf; // Trap frame for current syscall

    struct context *context; // swtch() here to run process. Process's context

    void *chan;        // If non-zero, sleeping on chan. More when we discuss sleep, wakeup

    int killed;        // If non-zero, have been killed

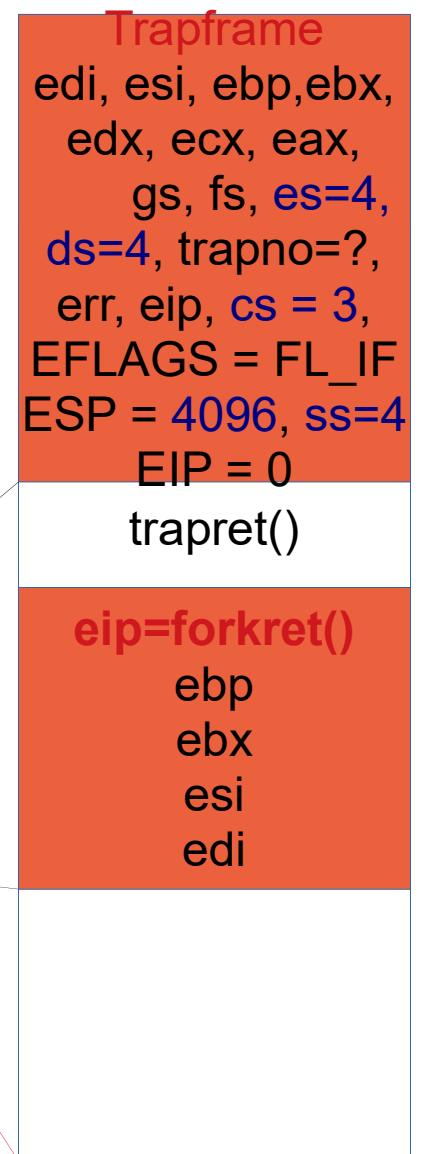
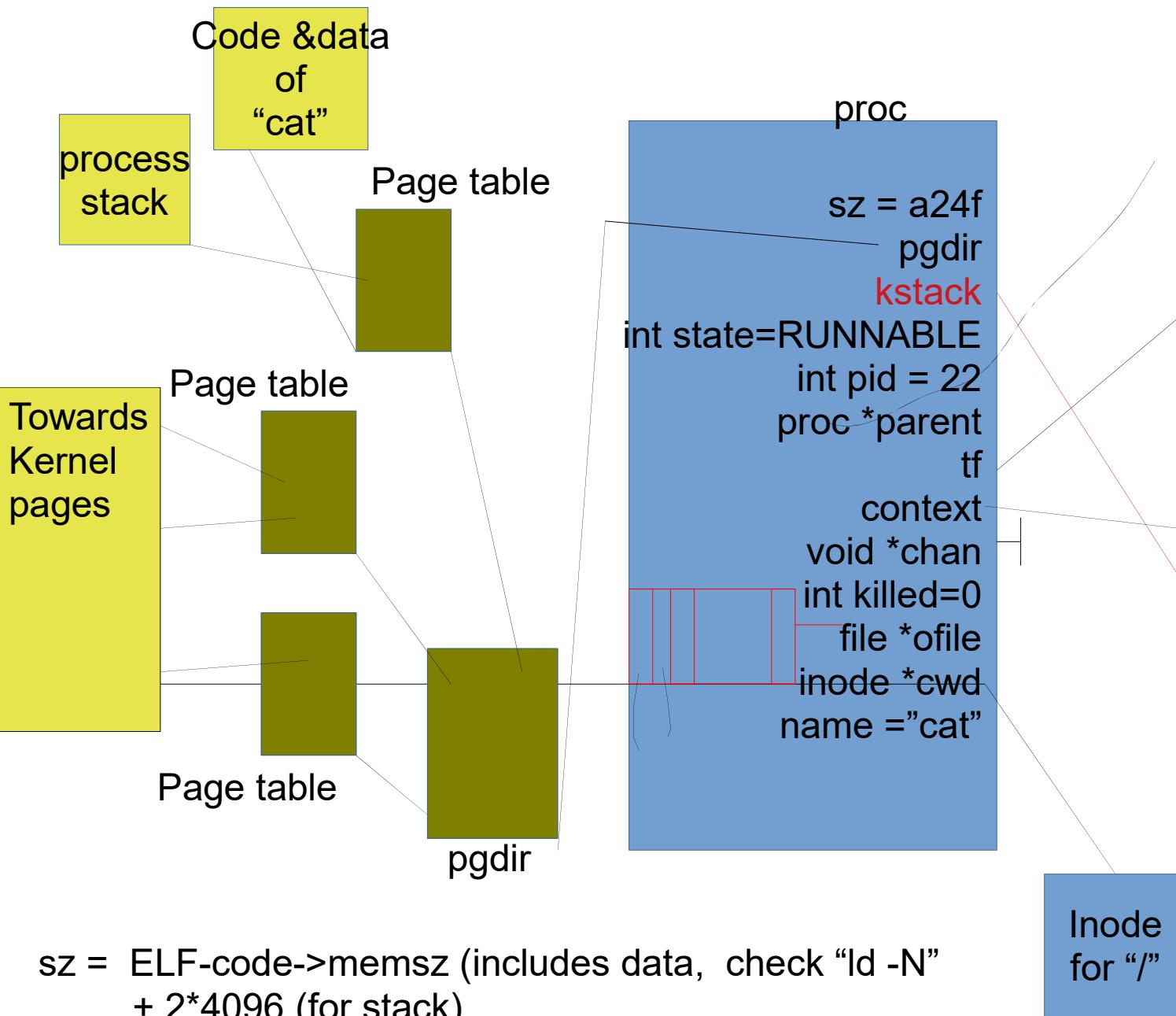
    struct file *ofile[NFILE]; // Open files, used by open(), read(),...

    struct inode *cwd; // Current directory, changed with "chdir()"

    char name[16];    // Process name (for debugging)

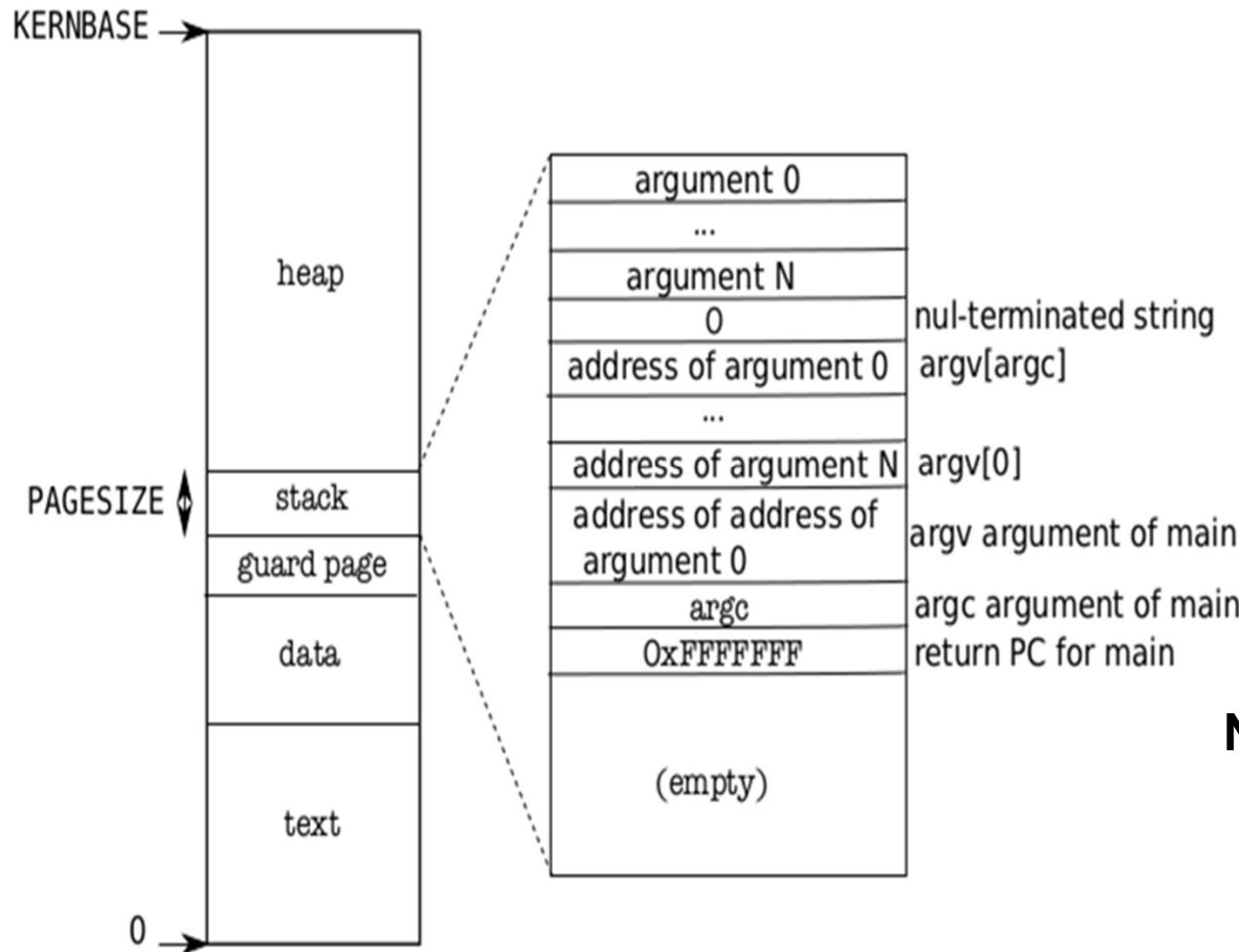
};
```

# struct proc diagram: Very imp!



In use only when you are in kernel on a “trap” = interrupt/syscall. “tf” always used. trapret,forkret used during fork()

## Memory Layout of a user process



## Memory Layout of a user process

After exec()

Note the argc, argv on stack

The “guard page” is just a mapping in page table.  
No frame allocated. It’s marked as invalid. So if stack grows (due to many function calls), then

OS will detect it with an exception

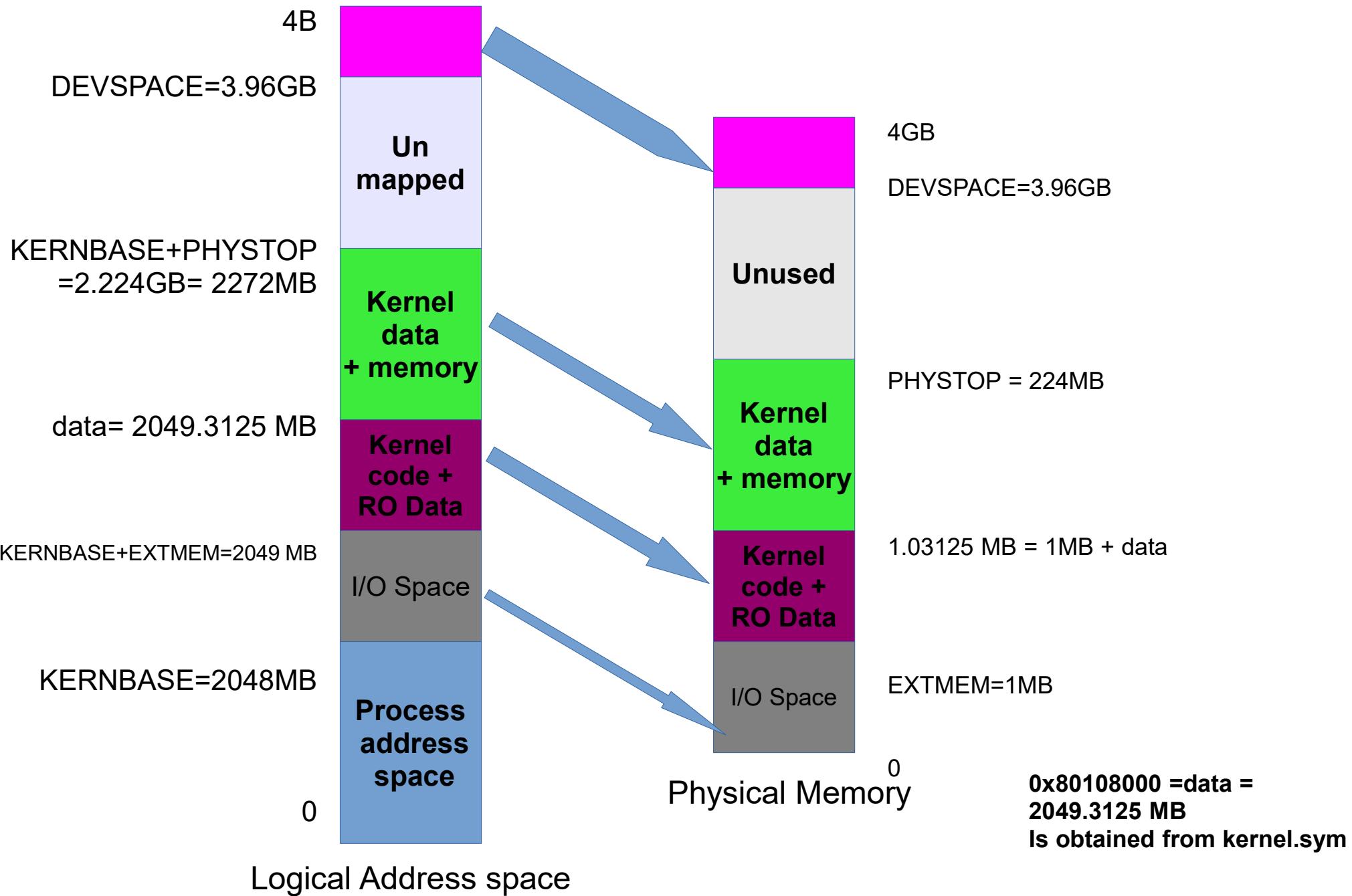
# Handling Traps

# Some basic steps

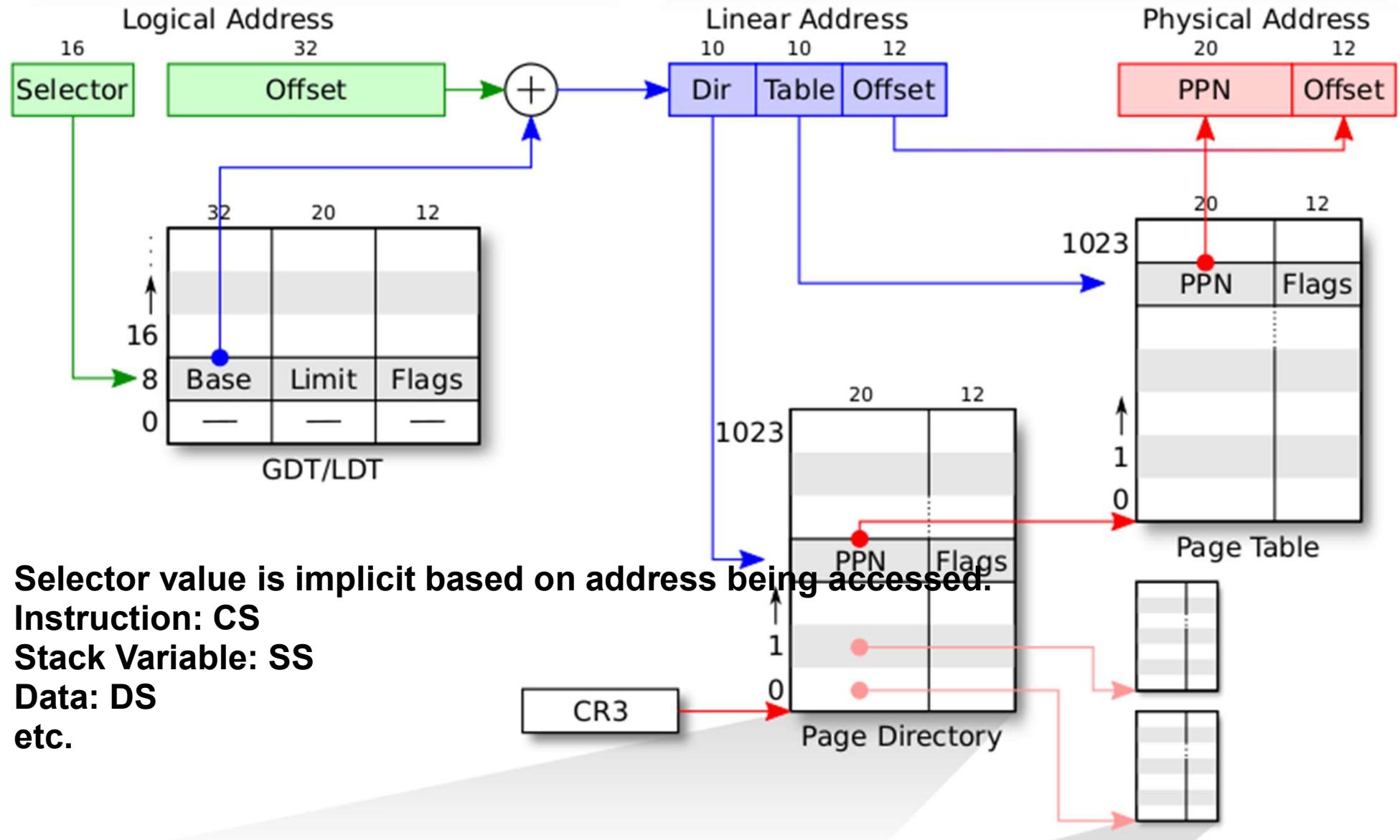
- **Xv6.img is created by “make”**
- **Contains bootsector, kernel code, kernel data**
- **QEMU boots using xv6.img**
- **First it runs bootloader**
- **Bootloader loads kernel of xv6 (from xv6.img)**
- **Kernel starts running**
- **Kernel running..**
- **Kernel calls main() of kernel (NOT a C application!) & Initializes:**
  - memory management data structures
  - process data structures
  - file handling data structures
  - Multi-processors
  - Multi-processor data structures
  - Interrupt Descriptor Table
  - ...
  - Then creates init()
  - Init() fork-execs shell

kmap[] mappings done in kvmalloc().

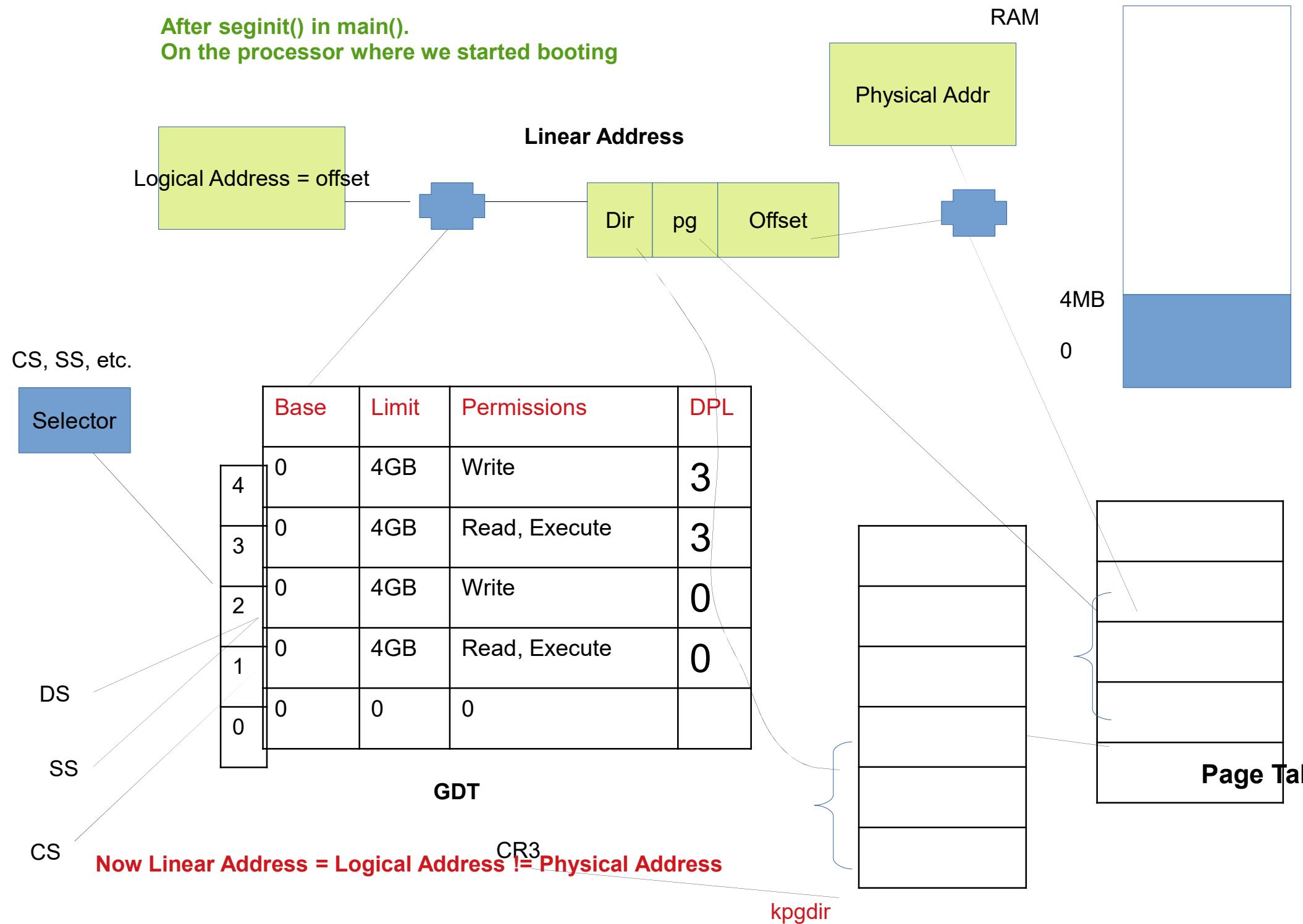
This shows segmentwise, entries are done in page directory and page table for corresponding VA-> PA mappings



# Segmentation + Paging



After seginit() in main().  
On the processor where we started booting



# Handling traps

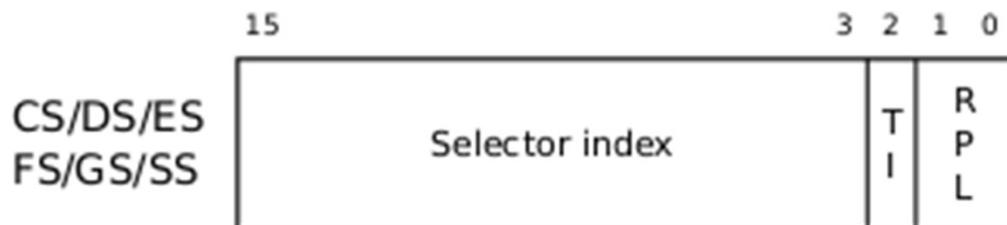
- ❑ Transition from user mode to kernel mode
  - ❑ On a system call
  - ❑ On a hardware interrupt
- ❑ User program doing illegal work (exception)
  - ❑ Actions needed, particularly w.r.t. to hardware interrupts
  - ❑ Change to kernel mode & switch to kernel stack
  - ❑ Kernel to work with devices, if needed
  - ❑ Kernel to understand interface of device

# Handling traps

- Actions needed on a trap
  - Save the processor's registers (context) for future use
  - Set up the system to run kernel code (kernel context) on kernel stack
  - Start kernel in appropriate place (sys call, intr handler, etc)
  - Kernel to get all info related to event (which block I/O done?, which sys call called, which process did exception and what type, get arguments to system call, etc)

# Privilege level

- The x86 has 4 protection levels, numbered 0 (most privilege) to 3 (least privilege).
- In practice, most operating systems use only 2 levels: 0 and 3, which are then called kernel mode and user mode, respectively.
- The current privilege level with which the x86 executes instructions is stored in %cs register, in the field CPL.



TI      Table index (0=GDT, 1=LDT)  
RPL    Requester privilege level

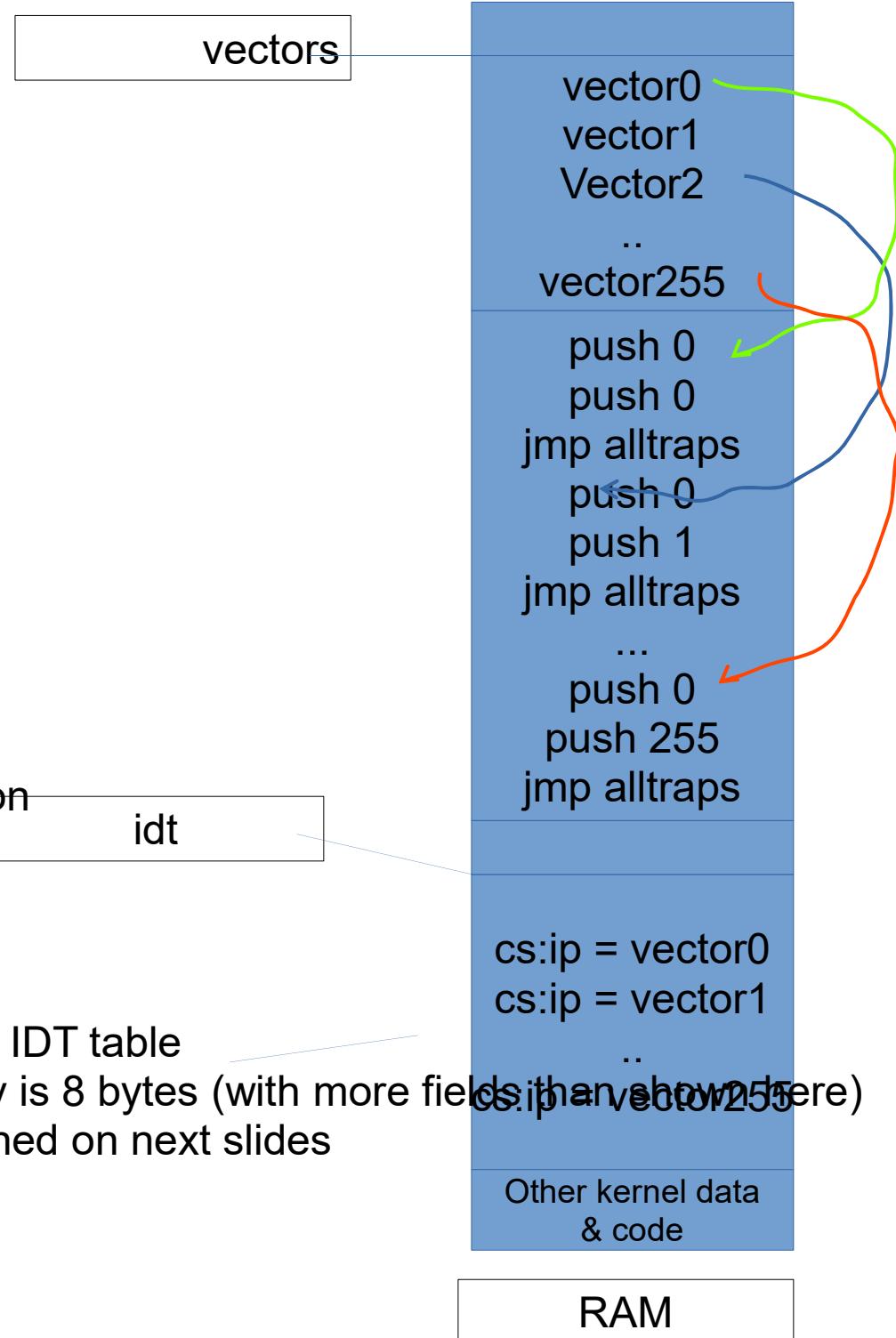
# Privilege level

- Changes automatically on
  - “int” instruction
  - hardware interrupt
  - exception
- Changes back on
  - iret
- “int” 10 --> makes 10<sup>th</sup> hardware interrupt. S/w interrupt can be used to ‘create hardware interrupt’
- Xv6 uses “int 64” for actual system calls

# Interrupt Descriptor Table (IDT)

- IDT defines interrupt handlers
  - Has 256 entries
  - each giving the %cs and %eip to be used when handling the corresponding interrupt.
- Mapping
- Interrupts 0-31 are defined for software exceptions, like divide errors or attempts to access invalid memory addresses.
  - Xv6 maps the 32 hardware interrupts to the range 32-63
  - and uses interrupt 64 as the system call interrupt

IDT setup  
done by tvinit() function



The array of “vectors”  
And the code of “push, ..jmp”  
Is part of kernel image (xv6.img)

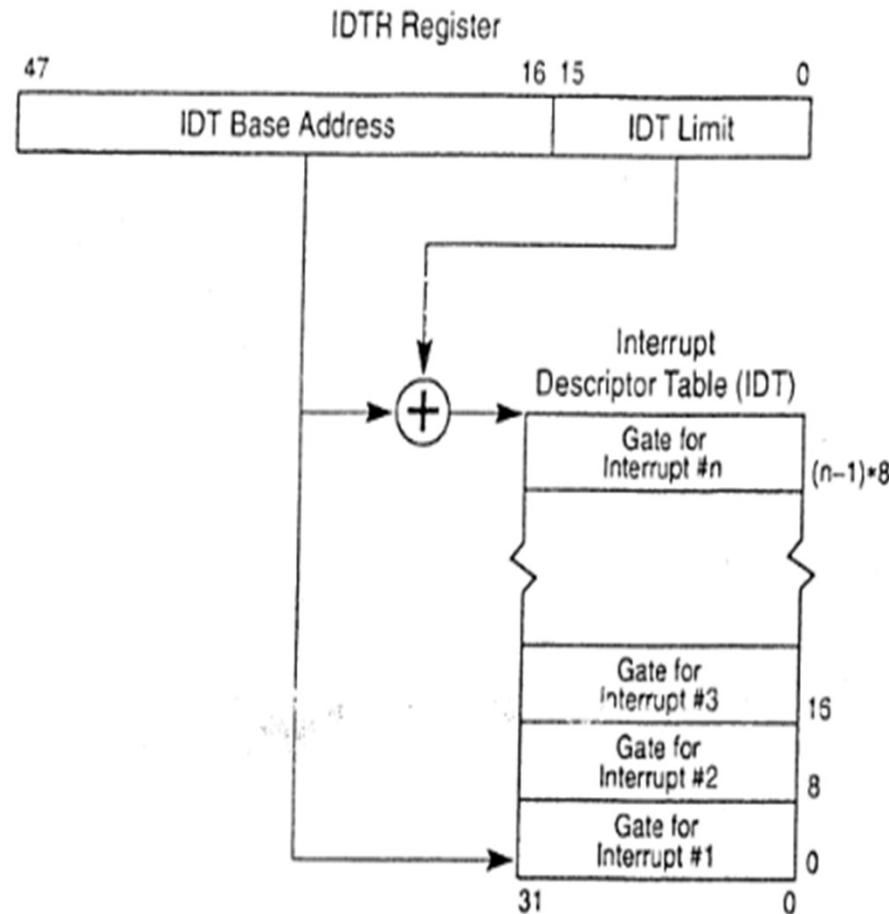
The tvinit() is called during kernel initialization

This is the IDT table  
Each entry is 8 bytes (with more fields than shown here)  
as mentioned on next slides

Other kernel data & code

RAM

# IDTR and IDT



IDT is in RAM

IDTR is in CPU

# Interrupt Descriptor Table (IDT) entries (in RAM)

```
// 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_{IG32,TG32})
    uint s : 1; // must be 0 (system)
    uint dpl : 2; // descriptor(new) privilege level
    uint p : 1; // Present
    uint off_31_16 : 16; // high bits of offset in segment
};
```

# Setting IDT entries

```
void
tvinit(void)
{
    int i;
    for(i = 0; i < 256; i++)
        SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
             vectors[T_SYSCALL], DPL_USER);
    /* value 1 in second argument --> don't disable interrupts
       * DPL_USER means that processes can raise this
       interrupt. */
    initlock(&tickslock, "time");
}
```

# Setting IDT entries

```
#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; \
}
```

# Setting IDT entries

**Vectors.S**

```
# generated by vectors.pl - do not
edit

# handlers

.globl alltraps

.globl vector0

vector0:
    pushl $0
    pushl $0
    jmp alltraps

.globl vector1

vector1:
    pushl $0
    pushl $1
    jmp alltraps
```

**trapasm.S**

```
#include "mmu.h"

# vectors.S sends all traps
here.

.globl alltraps

alltraps:
    # Build trap frame.

    pushl %ds
    pushl %es
    pushl %fs
    pushl %gs

    Pushal
    ....
```

**How will interrupts be handled?**

# On int instruction/interrupt the CPU does this:

- Fetch the n'th descriptor from the IDT, where n is the argument of int. (IDTR->idt[n])
- Check that CPL in %cs is <= DPL, where DPL is the privilege level in the descriptor.
- Temporarily save %esp and %ss in CPU-internal registers, but only if the target segment selector's PL < CPL.
- Switching from user mode to kernel mode. Hence save user code's SS and ESP
- Load %ss and %esp from a task segment descriptor.
- Stack changes to kernel stack now. TS descriptor is on GDT, index given by TR register. See switchuvm()

- Push %ss. // optional
- Push %esp. // optional (also changes ss,esp using TSS)
- Push %eflags.
- Push %cs.
- Push %eip.
- Clear the IF bit in %eflags, but only on an interrupt.
- Set %cs and %eip to the values in the descriptor.

# After “int” ‘s job is done

- IDT was already set

- Remember vectors.S

- So jump to 64<sup>th</sup> entry in vector’s  
vector64:

- pushl \$0

- pushl \$64

- jmp alltraps

- So now stack has ss, esp, eflags, cs, eip, 0 (for error code),  
64

- Next run alltraps from trapasm.S

```
# Build trap frame.  
pushl %ds  
pushl %es  
pushl %fs  
pushl %gs  
pushal // push all gen purpose  
regs  
# Set up data segments.  
movw $(SEG_KDATA<<3), %ax  
movw %ax, %ds  
movw %ax, %es  
# Call trap(tf), where tf=%esp  
pushl %esp # first arg to trap()  
call trap  
addl $4, %esp
```

## alltraps:

- Now stack contains
  - ss, esp, eflags, cs, eip, 0 (for error code), 64, ds, es, fs, gs, eax, ecx, edx, ebx, oesp, ebp, esi, edi
- This is the struct trapframe !
- So the kernel stack now contains the trapframe
- Trapframe is a part of kernel stack

```
void  
trap(struct trapframe *tf)  
{  
    if(tf->trapno == T_SYSCALL){  
        if(myproc()->killed)  
            exit();  
        myproc()->tf = tf;  
        syscall();  
        if(myproc()->killed)  
            exit();  
        return;  
    }  
    switch(tf->trapno){  
        ....
```

# trap()

- Argument is trapframe
- In all traps
- Before “call trap”, there was “push %esp” and stack had the trapframe
- Remember calling convention --> when a function is called, the stack contains the arguments in reverse order (here only 1 arg)

# trap()

- Has a switch
- switch(tf->trapno)
- Q: who set this trapno?
- Depending on the type of trap
- Call interrupt handler

- Timer
  - wakeup(&ticks)
- IDE: disk interrupt
  - Ideintr()
- KBD
  - KbdINTR()
- COM1
  - Uatrintr()
- If Timer
  - Call yield() -- calls sched()
- If process was killed (how is that done?)
  - Call exit()!

# when trap() returns

```
#Back in alltraps  
  
call trap  
  
addl $4, %esp  
  
  
# Return falls through to trapret...  
  
.globl trapret  
  
trapret:  
  
    popal  
  
    popl %gs  
    popl %fs  
    popl %es  
    popl %ds  
  
    addl $0x8, %esp # trapno and errcode  
  
    iret
```

## Stack had (trapframe)

ss, esp,eflags, cs, eip, 0 (for error code), 64, ds, es, fs, gs, eax, ecx, edx, ebx, oesp, ebp, esi, edi, esp

add \$4 %esp

esp

popal

eax, ecx, edx, ebx, oesp, ebp, esi, edi

Then gs, fs, es, ds

add \$0x8, %esp

0 (for error code), 64

iret

ss, esp,eflags, cs, eip,

# XV6 bootloader

Abhijit A. M.

[abhijit.comp@coep.ac.in](mailto:abhijit.comp@coep.ac.in)

Credits:

xv6 book by Cox, Kaashoek, Morris

Notes by Prof. Sorav Bansal

# A word of caution

- ❑ We begin reading xv6 code
- ❑ But it's not possible to read this code in a “linear fashion”
- ❑ The dependency between knowing OS concepts and reading/writing a kernel that is written using all concepts

# **What we have seen ....**

- ❑ Compilation process, calling conventions**
- ❑ Basics of Memory Management by OS**
- ❑ Basics of x86 architecture**
- ❑ Registers, segments, memory management unit, addressing, some basic machine instructions,**
- ❑ ELF files**
- ❑ Objdump, program headers**
- ❑ Symbol tables**

# Boot-process

- Bootloader itself
  - gets loaded by the BIOS at a fixed location in memory and BIOS makes it run
  - Our job, as OS programmers, is to write the bootloader code
- Bootloader does
  - Pick up code of OS from a ‘known’ location and loads it in memory
  - Makes the OS run
- Xv6 bootloader: bootasm.S bootmain.c (see Makefile)

# **bootasm.S bootmain.c: Steps**

- 1) Starts in “real” mode, 16 bit mode. Does some misc legacy work.**
- 2) Runs instructions to do MMU set-up for protected-mode & only segmentation (0-4GB, identity mapping), changes to protected mode.**
- 3) Reads kernel ELF file and loads it in RAM, as per instructions in ‘kernel’ ELF file**
- 4) Sets up paging (4 MB pages)**
- 5) Runs main() of kernel (which later on sets up 4KB pages)**

# bootloader

- BIOS Runs (automatically)

- Loads boot sector into RAM at 0x7c00

- Starts executing that code

- How to make sure that your bootloader is loaded at 0x7c00 ?

- Makefile has

```
bootblock: bootblock.S bootmain.c
```

```
$(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S .....
```

```
...
```

```
$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o  
bootmain.o
```

Results in:

00007c00 <start>: in bootblock.asm

# Processor starts in real mode

- Processor starts in real mode – works like 16 bit 8088
- eight 16-bit general-purpose registers,
- Segment registers %cs, %ds, %es, and %ss --> additional bits necessary to generate 20-bit memory addresses from 16-bit registers.

$\text{addr} = \text{seg} \ll 4 + \text{addr}$



**Effective memory translation in the beginning  
At \_start in bootasm.S:**

**%cs=0 %ip=7c00.**

**So effective address = 0\*16+ip = ip**

# bootloader

- First instruction is ‘cli’
- disable interrupts
- So that until your code loads all hardware interrupt handlers, no interrupt will occur

# Zeroing registers

```
# Zero data segment registers DS, ES, and  
ss.  
  
xorw %ax,%ax          # Set %ax to zero  
movw %ax,%ds          # -> Data Segment  
movw %ax,%es          # -> Extra Segment  
movw %ax,%ss          # -> Stack Segment
```

- zero ax and ds, es, ss
- BIOS did not put in anything perhaps

# A not so necessary detail

## Enable 21 bit address

seta20.1:

```
inb    $0x64,%al          # Wait for not busy  
testb   $0x2,%al  
jnz     seta20.1  
  
movb    $0xd1,%al          # 0xd1 -> port 0x64  
outb    %al,$0x64
```

seta20.2:

```
inb    $0x64,%al          # Wait for not busy  
testb   $0x2,%al  
jnz     seta20.2  
  
movb    $0xdf,%al          # 0xdf -> port 0x60  
outb    %al,$0x60
```

▪ Seg:off with 16 bit segments can actually address more than 20 bits of memory. After 0x100000 (=2<sup>20</sup>), 8086 wrapped addresses to 0.

▪ 80286 introduced 21<sup>st</sup> bit of address. But older software required 20 bits only. BIOS disabled 21<sup>st</sup> bit. Some OS needed 21<sup>st</sup> Bit. So enable it.

▪ Write to Port 0x64 and 0x60 -> keyboard controller

▪ to enable 21<sup>st</sup> bit out of address translation

▪ Why? Before the A20, i.e. 21<sup>st</sup> bit was introduced, it belonged to keyboard controller

▪ For more details see  
[https://en.wikipedia.org/wiki/A20\\_line](https://en.wikipedia.org/wiki/A20_line)  
[https://en.wikipedia.org/wiki/A20\\_Line](https://en.wikipedia.org/wiki/A20_Line)

**After this**

**Some instructions are run  
to enter protected mode**

**And further code runs in protected mode**

# Real mode Vs protected mode

- Real mode 16 bit registers

- Protected mode

- Enables segmentation + Paging both

- No longer seg\*16+offset calculations

- Segment registers is index into segment descriptor table. But segment:offset pairs continue

- **mov %esp, \$32 # SS will be used with esp**

- More in next few slides

- Other segment registers need to be explicitly mentioned in instructions

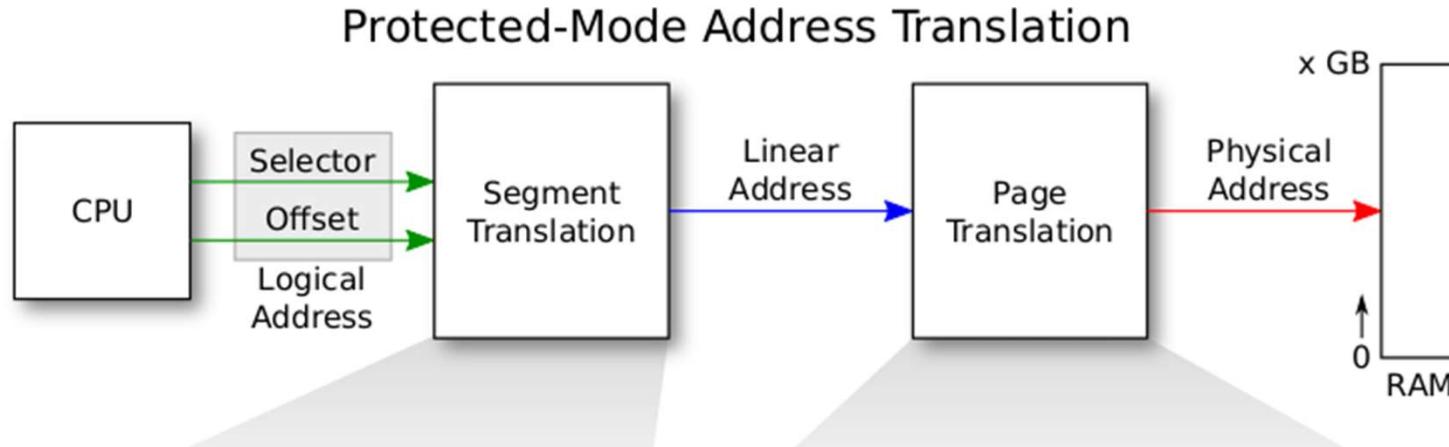
- **Mov FS:\$200, 30**

- 32 bit registers

- can address upto  $2^{32}$  memory

- Can do arithmetic in 32 bits

# X86 address : protected mode address translation



**Both Segmentation and Paging are used in x86**  
**X86 allows optionally one-level or two-level paging**  
**Segmentation is a must to setup, paging is optional (needs to be enabled)**  
**Hence different OS can use segmentation+paging in different ways**

# X86 segmentation

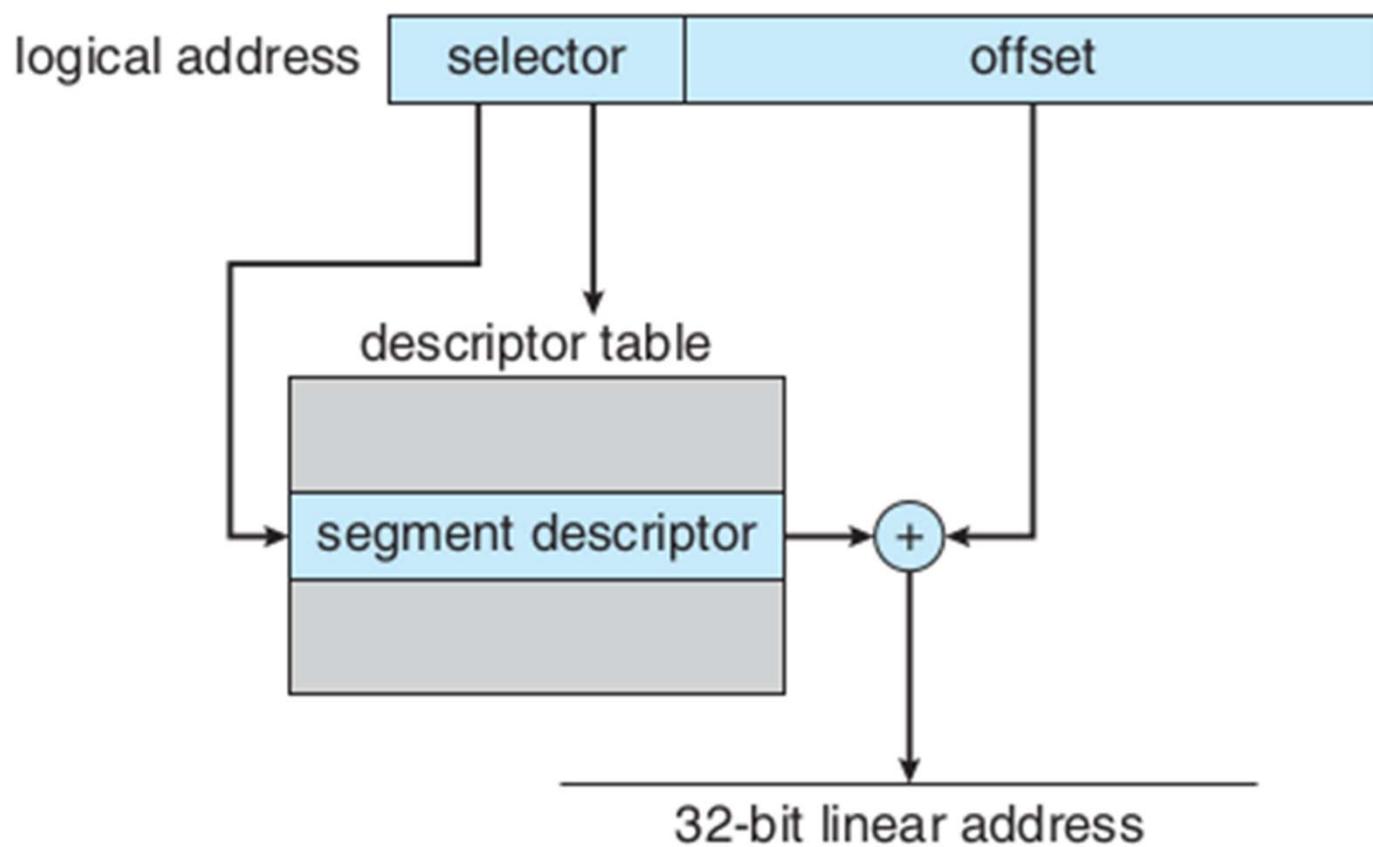


Figure 8.22 IA-32 segmentation.

# Paging concept, hierarchical paging

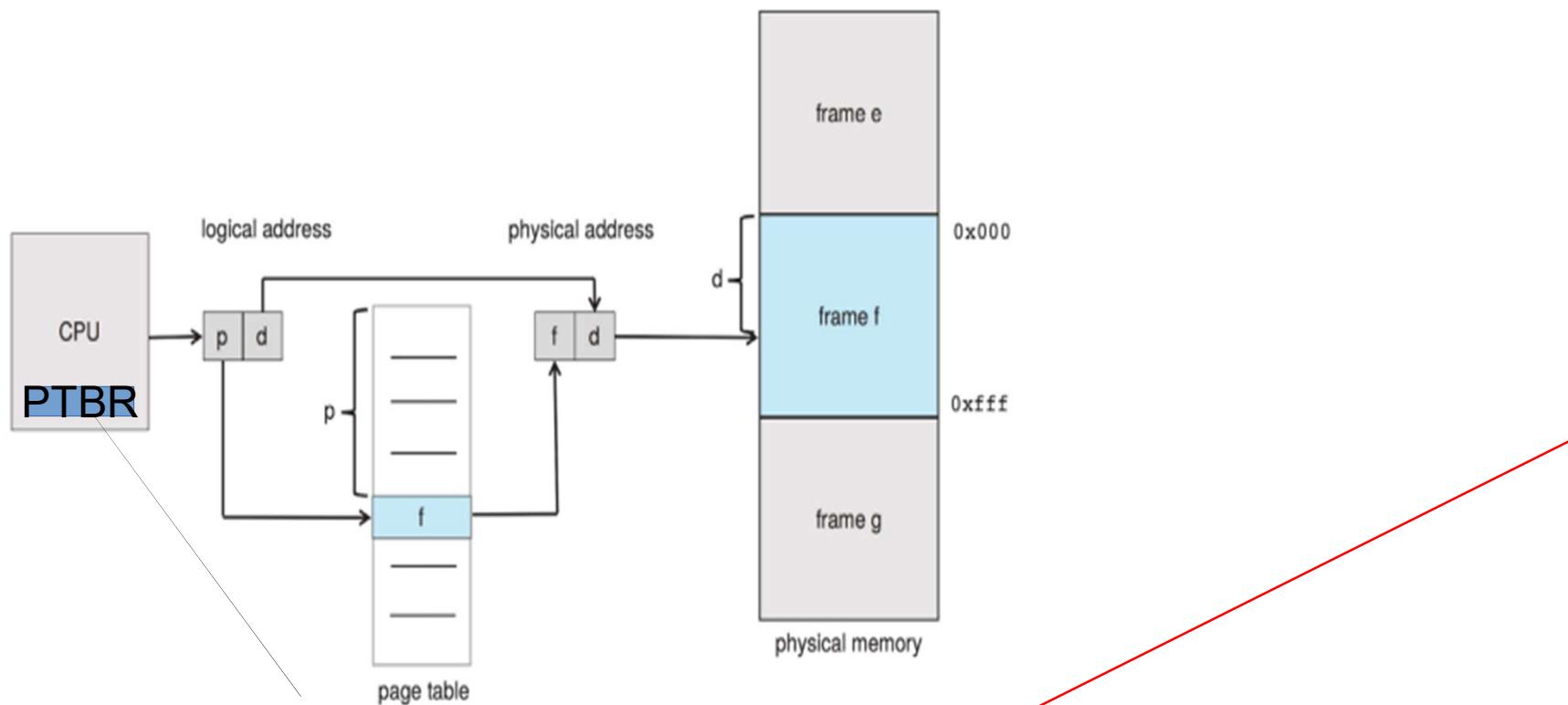


Figure 9.8 Paging hardware.

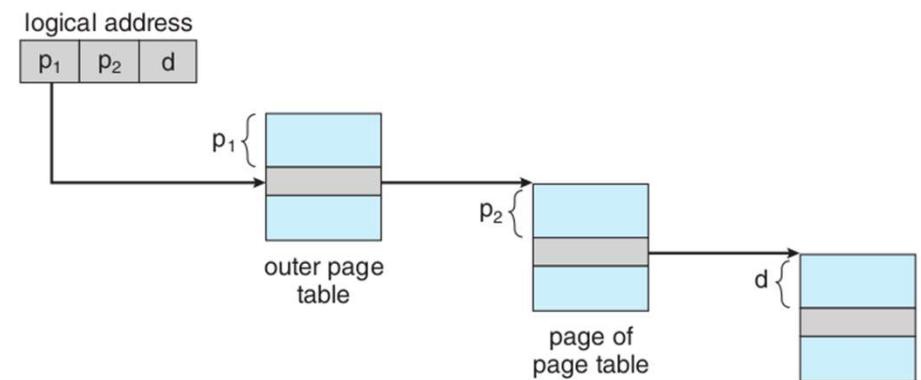
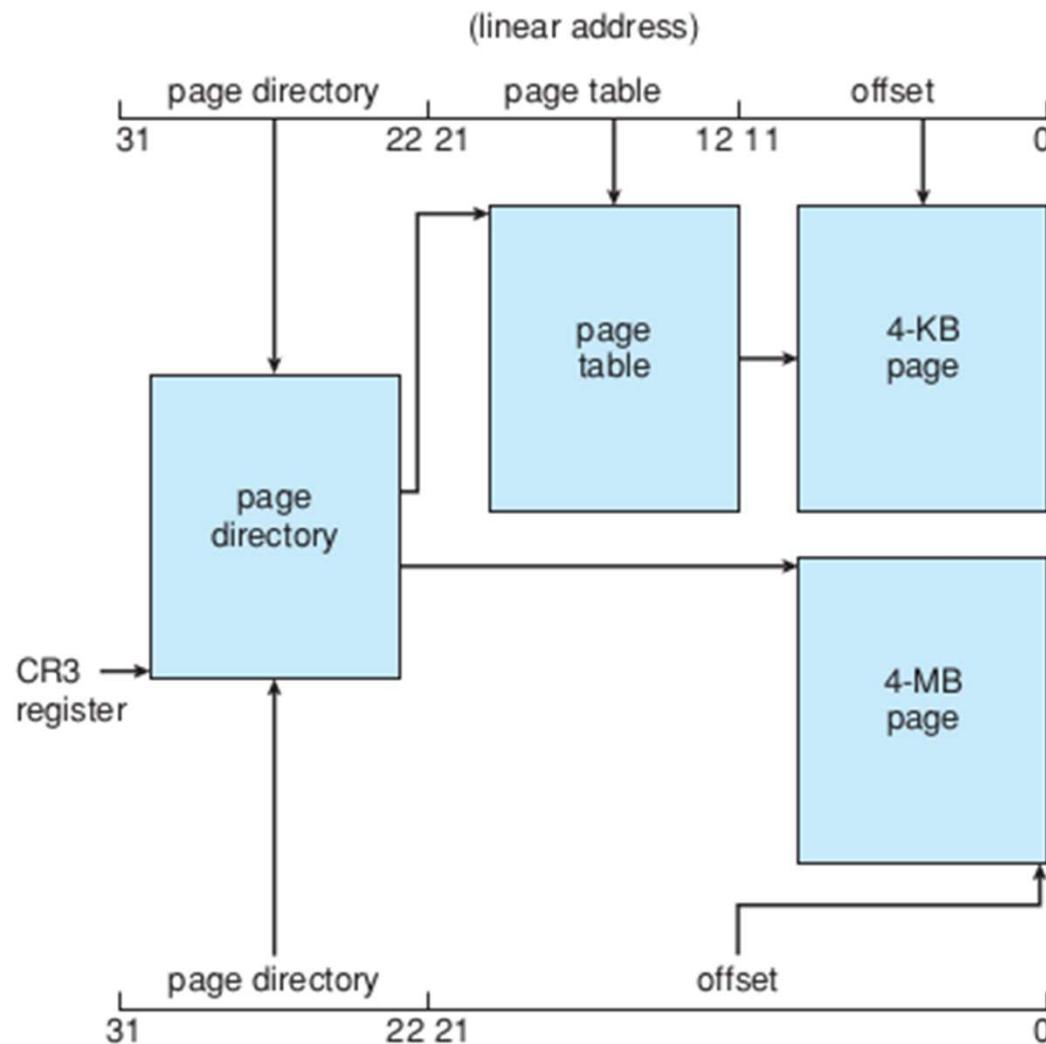


Figure 9.16 Address translation for a two-level 32-bit paging architecture.

# X86 paging



**Figure 8.23** Paging in the IA-32 architecture.

# Page Directory Entry (PDE) Page Table Entry (PTE)

31

| Page table physical page number | 12 11 10 9 8 7 6 5 4 3 2 1 0                                                         |
|---------------------------------|--------------------------------------------------------------------------------------|
|                                 | A<br>V<br>L      G<br>S      P<br>0      A<br>C<br>W<br>D<br>T      U<br>U<br>W<br>P |

PDE

P Present

W Writable

U User

WT 1=Write-through, 0=Write-back

CD Cache disabled

A Accessed

D Dirty

PS Page size (0=4KB, 1=4MB)

PAT Page table attribute index

G Global page

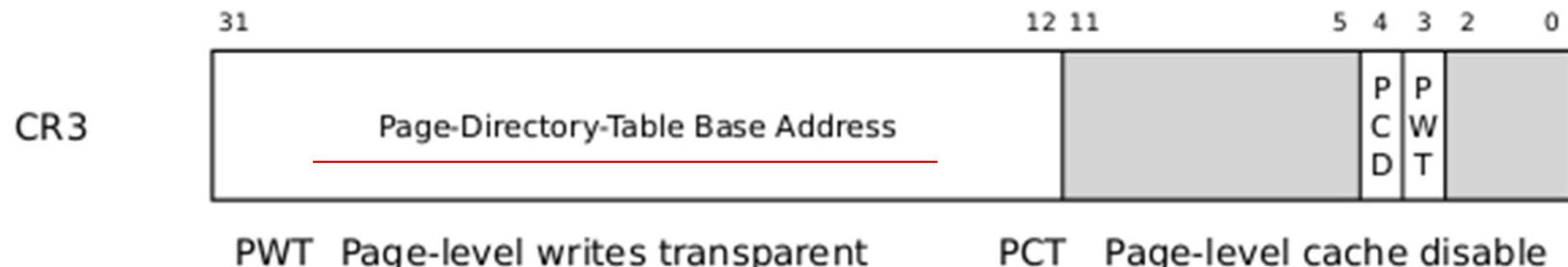
AVL Available for system use

31

| Physical page number | 12 11 10 9 8 7 6 5 4 3 2 1 0                                                                   |
|----------------------|------------------------------------------------------------------------------------------------|
|                      | A<br>V<br>L      G<br>A<br>T      P<br>D<br>A      0<br>C<br>W<br>D<br>T      U<br>U<br>W<br>P |

PTE

# CR3

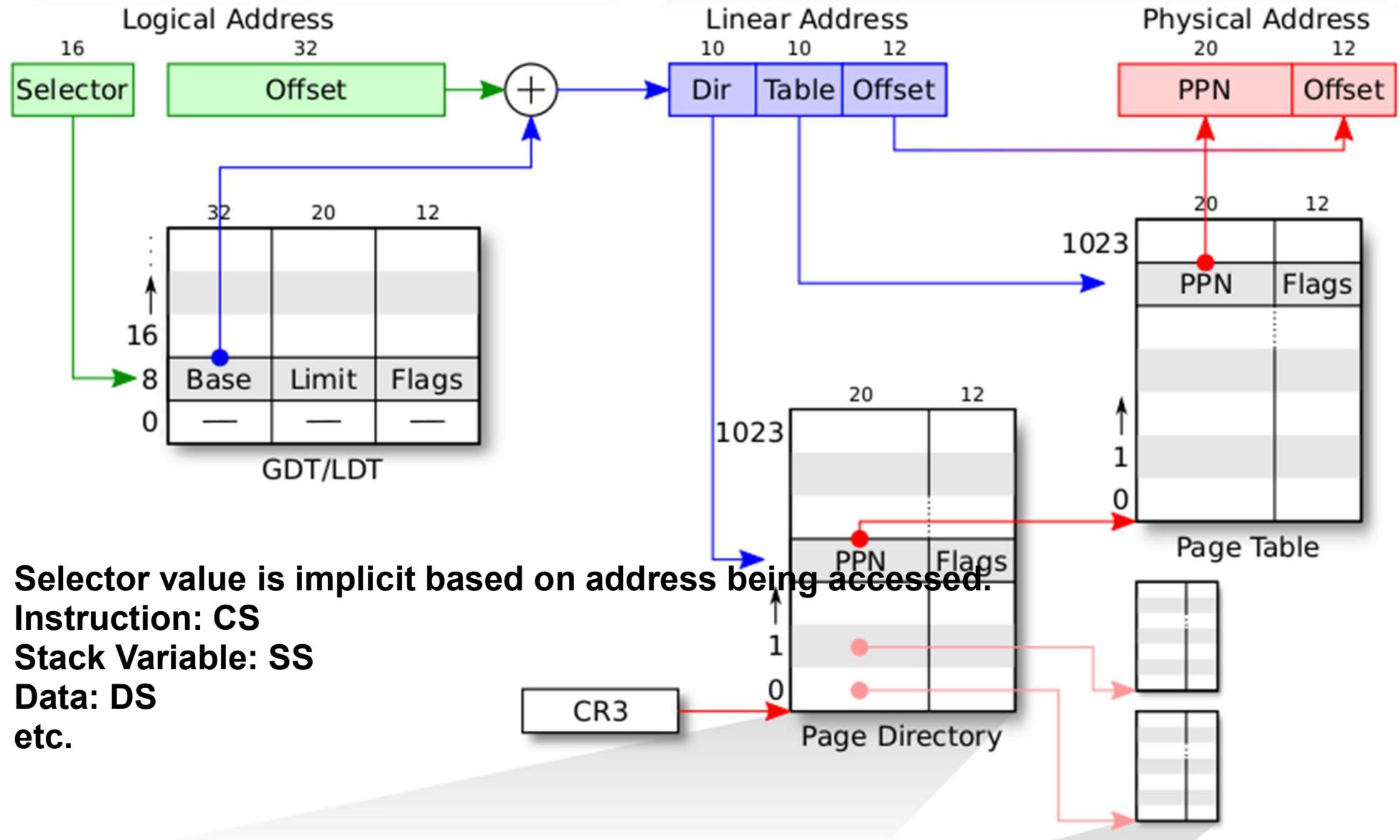


# CR4

CR4

|                                                     |                                   |        |                               |
|-----------------------------------------------------|-----------------------------------|--------|-------------------------------|
| 31                                                  | 11 10 9 8 7 6 5 4 3 2 1 0         |        |                               |
| O S X M O S C E X P P M G E P A S D T S V P V M I E |                                   |        |                               |
| VME                                                 | Virtual-8086 mode extensions      | MCE    | Machine check enable          |
| PVI                                                 | Protected-mode virtual interrupts | PGE    | Page-global enable            |
| TSD                                                 | Time stamp disable                | PCE    | Performance counter enable    |
| DE                                                  | Debugging extensions              | OSFXSR | OS FXSAVE/FXRSTOR support     |
| PSE                                                 | Page size extensions              | OSXMM- | OS unmasked exception support |
| PAE                                                 | Physical-address extension        | EXCPT  |                               |

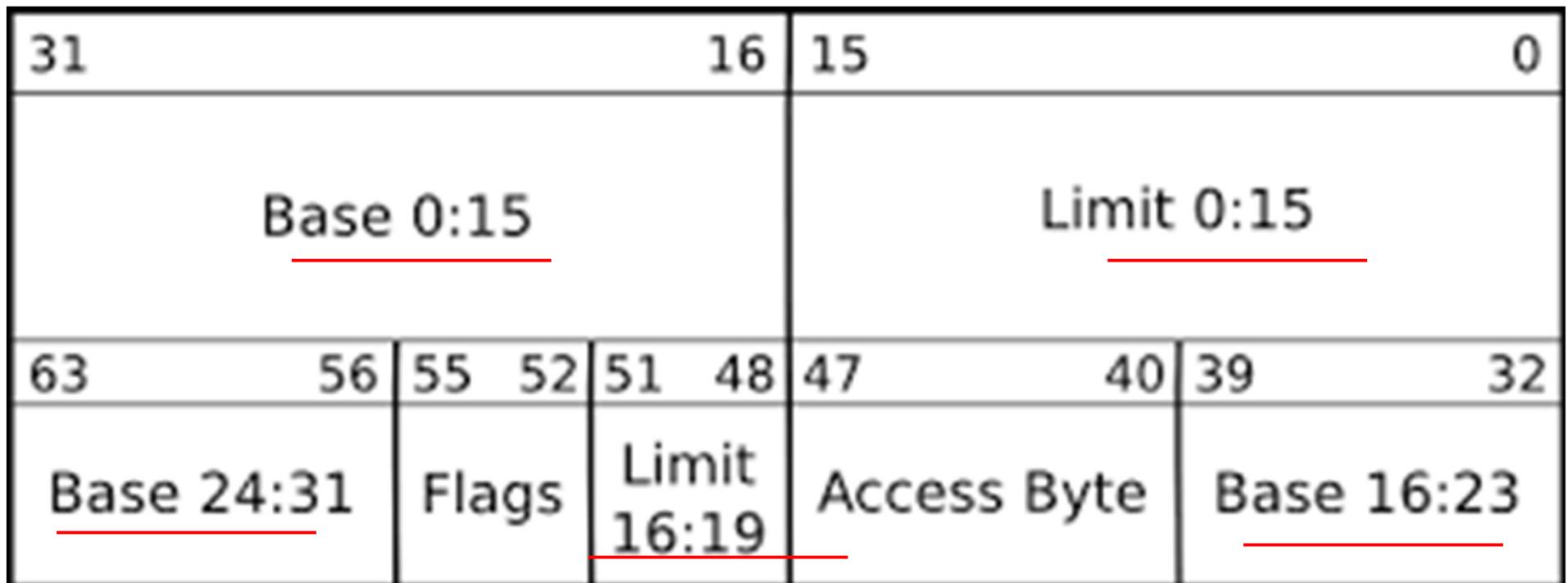
# Segmentation + Paging



# Segmentation + Paging setup of xv6

- xv6 configures the segmentation hardware by setting  
Base = 0, Limit = 4 GB
- translate logical to linear addresses without change,  
so that they are always equal.
- Segmentation is practically off
- Once paging is enabled, the only interesting address  
mapping in the system will be linear to physical.
- In xv6 paging is NOT enabled while loading kernel
- After kernel is loaded 4 MB pages are used for a while
- Later the kernel switches to 4 kB pages!

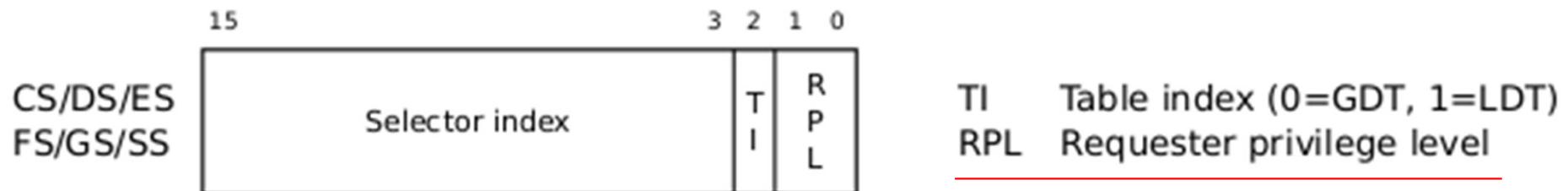
# GDT Entry



## asm.h

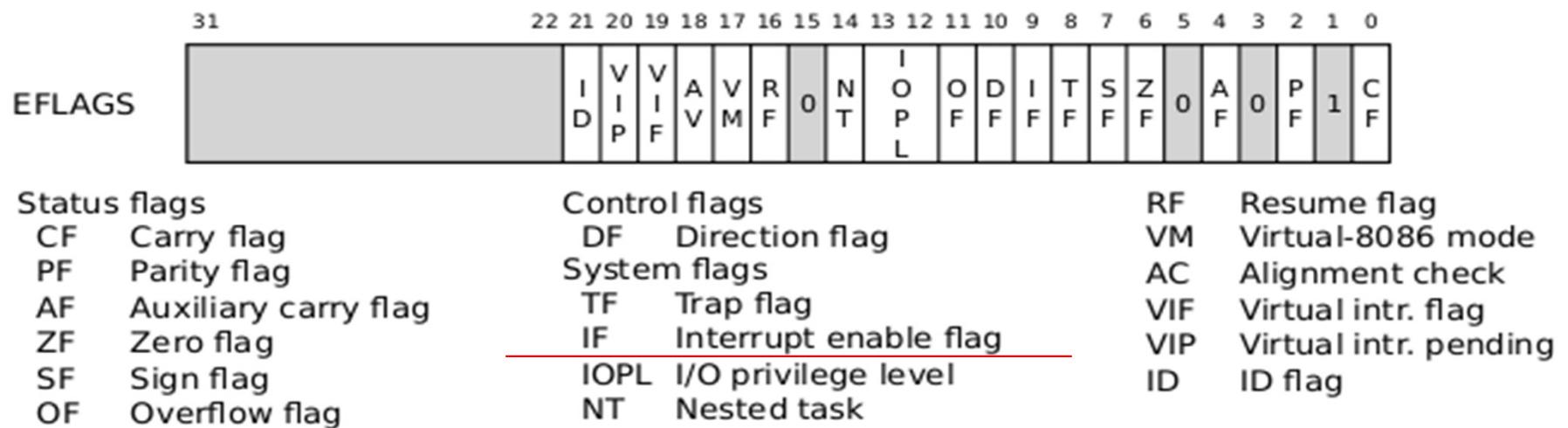
```
#define SEG_ASM(type,base,lim) \
    .word (((lim) >> 12) & 0xffff), ((base) & 0xffff); \
    .byte (((base) >> 16) & 0xff), (0x90 | (type)), \
          (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
```

# Segment selector



**Note in 16 bit mode, segment selector is 16 bit, here it's 13 bit + 3 bits**

# EFLAGS register



```
lgdt gdtdesc
```

...

```
# Bootstrap GDT
```

```
.p2align 2 # force 4 byte  
alignment
```

```
gdt:
```

```
SEG_NULLASM # null seg
```

```
SEG_ASM(STA_X|STA_R, 0x0,  
0xffffffff) # code seg
```

```
SEG_ASM(STA_W, 0x0,  
0xffffffff)
```

```
# data seg
```

```
gdtdesc:
```

```
.word (gdtdesc - gdt - 1)
```

```
# sizeof(gdt) - 1
```

```
.long gdt
```

# lgdt

- load the processor's (GDT) register with the value gdtdesc which points to the table gdt.

- table gdt : The table has a null entry, one entry for executable code, and one entry to data.

- all segments have a base address of zero and the maximum possible limit

- The code segment descriptor has a flag set that indicates that the code should run in 32-bit mode

- With this setup, when the boot loader enters protected mode, logical addresses map one-to-one to physical addresses.

- At gdtdesc we have this data

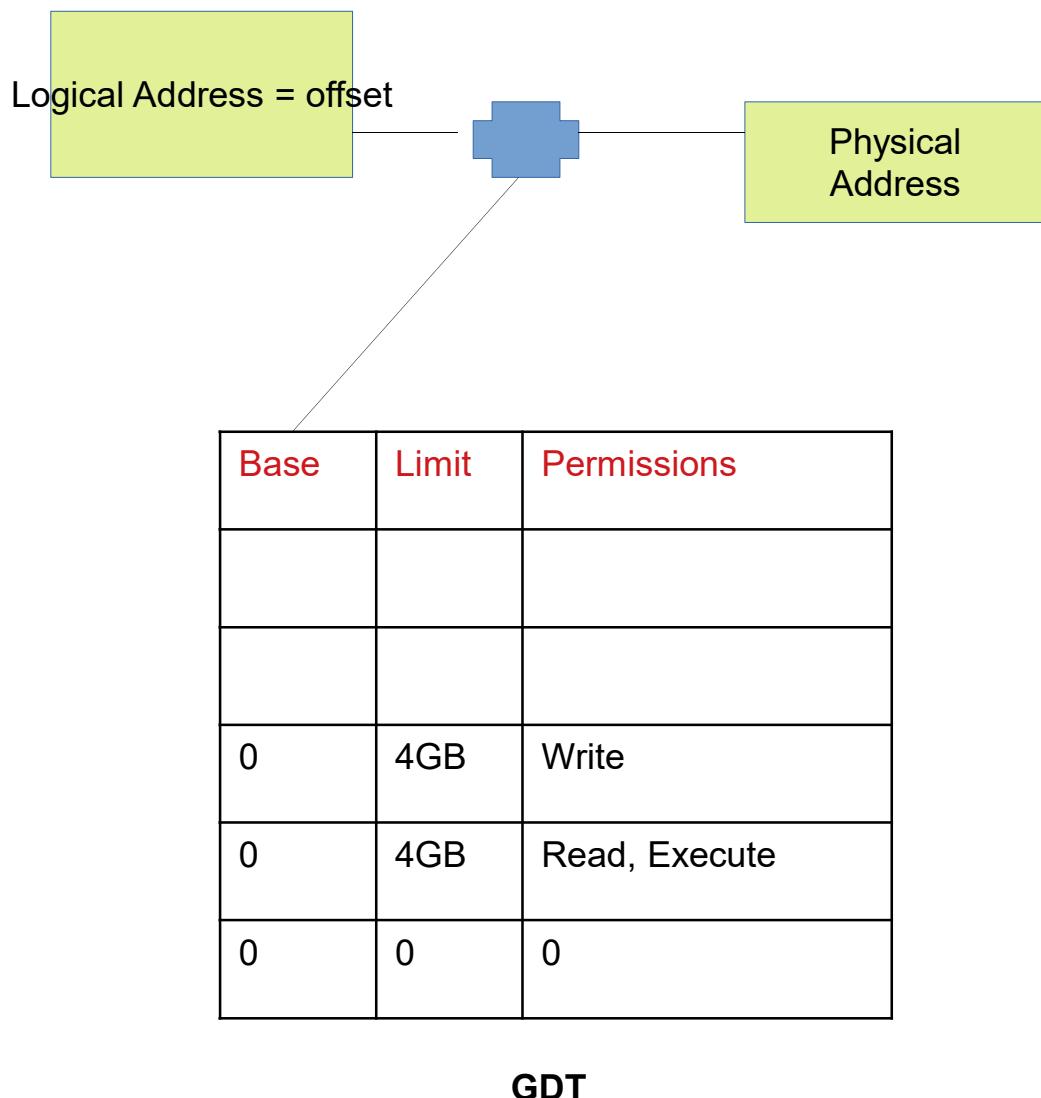
- 2, <4 byte addr of gdt>

- Total 6 bytes

- GDTR is : <address 4 byte>, <table limit 2 byte>

- So lgdt gdtdesc loads these two values in GDTR

**bootasm.S after “lgdt gdtdesc”  
till jump to “entry”**



Still  
Logical Address = Physical  
address!

But with GDT in picture and  
Protected Mode operation

**During this time,**  
**Loading kernel from  
ELF into physical memory**

**Addresses in “kernel” file  
translate to same physical  
address!**

# Prepare to enable protected mode

- ② Prepare to enable protected mode by setting the 1 bit (CR0\_PE) in register %cr0

```
movl  %cr0, %eax  
orl  $CR0_PE, %eax  
movl  %eax, %cr0
```

# CR0

| CR0 | 31 30 29 28         | 19 18 17 16 15 | 6 5 4 3 2 1 0                |
|-----|---------------------|----------------|------------------------------|
|     | P C N<br>G D W      | A M<br>W P     | N E T S E M P<br>E T S M P E |
| PE  | Protection enabled  | ET             | Extension type               |
| MP  | Monitor coprocessor | NE             | Numeric error                |
| EM  | Emulation           | WP             | Write protect                |
| TS  | Task switched       | AM             | Alignment mask               |

**PG: Paging enabled or not**

**WP: Write protection on/off**

**PE: Protection Enabled --> protected mode.**

# Complete transition to 32 bit mode

`Ijmp $(SEG_KCODE<<3), $start32`

Complete the transition to 32-bit protected mode by using a long jmp

to reload %cs (=1) and %eip (=start32).

Note that ‘start32’ is the address of next instruction after Ijmp.

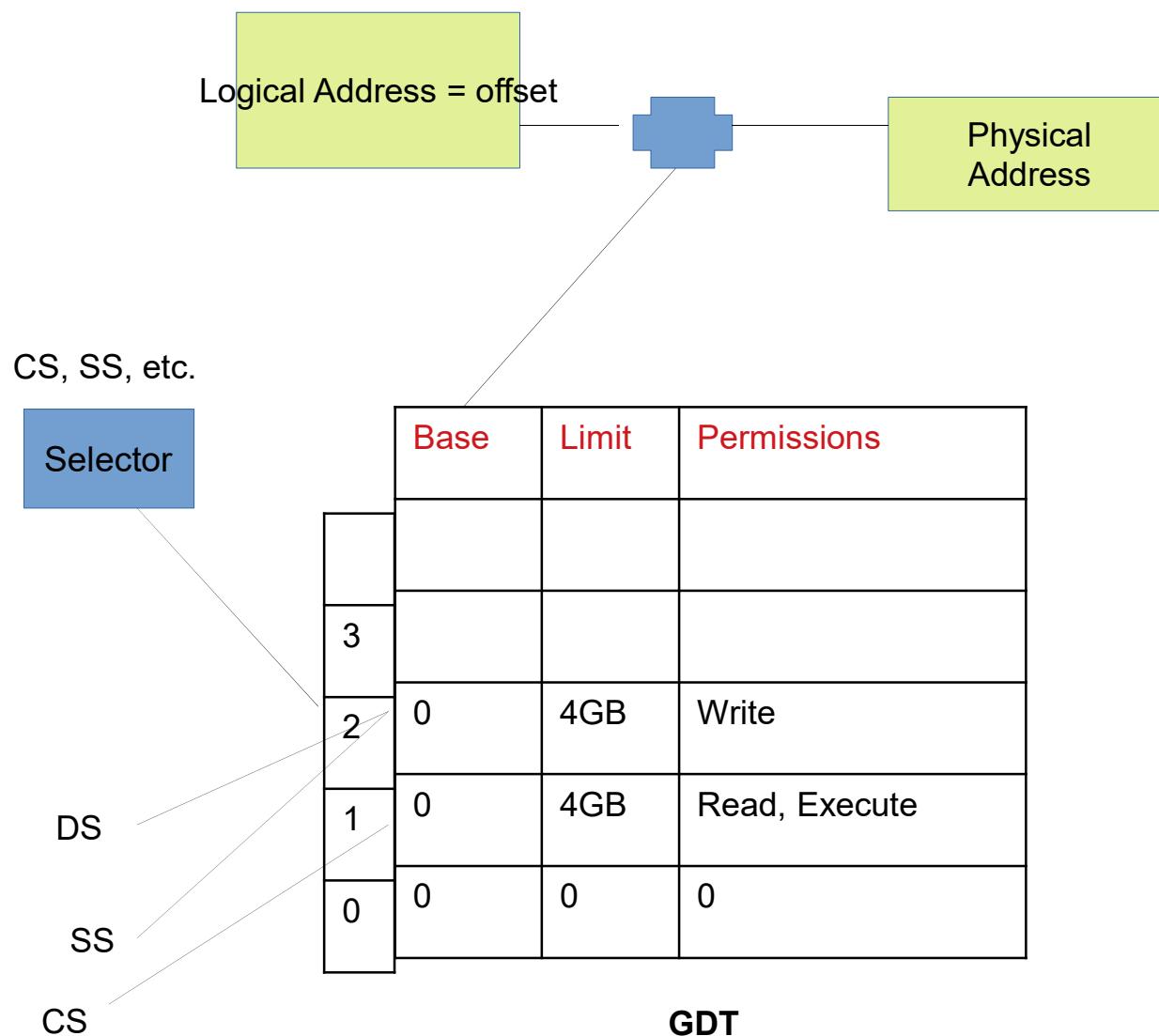
Note: The segment descriptors are set up with no translation (that is 0-4GB setting), so that the mapping is still the identity mapping.

# Jumping to “C” code

```
movw $(SEG_KDATA<<3), %ax # Our data segment  
selector  
  
movw %ax, %ds      # -> DS: Data Segment  
movw %ax, %es      # -> ES: Extra Segment  
movw %ax, %ss      # -> SS: Stack Segment  
movw $0, %ax      # Zero segments not ready  
for use  
  
movw %ax, %fs      # -> FS  
movw %ax, %gs      # -> GS  
  
# Set up the stack pointer and call into C.  
  
movl $start, %esp  
call bootmain
```

- ❑ Setup Data, extra, stack segment with SEG\_KDATA (=2), FS & GS (=0)
- ❑ Copy “\$start” i.e. 7c00 to stack-ptr
- ❑ It will grow from 7c00 to 0000
- ❑ Call bootmain() a C function
  - ❑ In bootmain.c

## Setup now



# bootmain(): already in memory, as part of ‘bootblock’

bootmain.c , expects to find a copy of the kernel executable on the disk starting at the second sector (sector = 1).

Why?

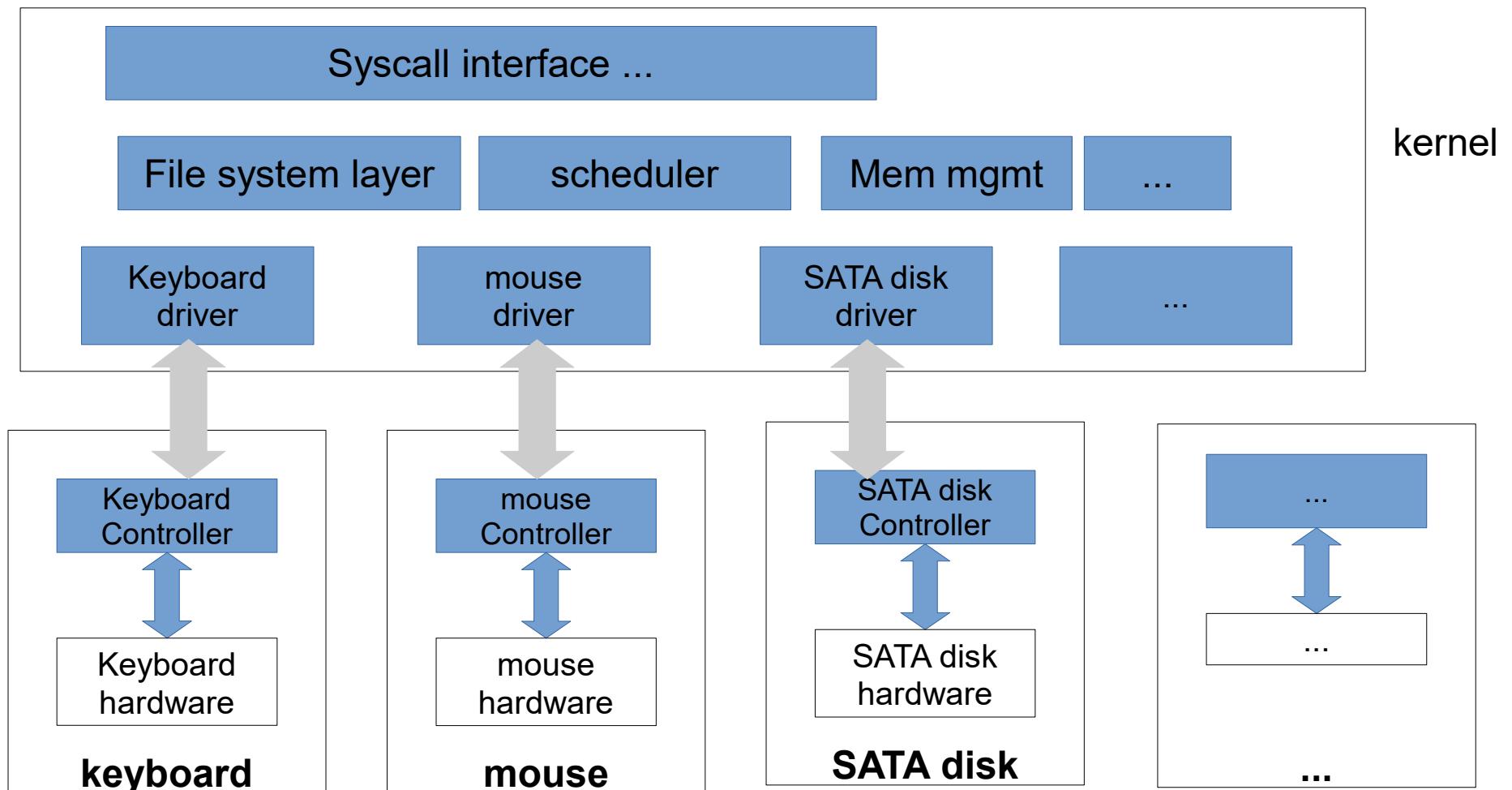
The kernel is an ELF format binary

Bootmain loads the first 4096 bytes of the ELF binary. It places the in-memory copy at address 0x10000

readseg() is a function that runs OUT instructions in particular IO ports, to issue commands to read from Disk

```
void  
bootmain(void)  
{  
    struct elfhdr *elf;  
    struct proghdr *ph, *eph;  
    void (*entry)(void);  
    uchar* pa;  
  
    elf = (struct elfhdr*)0x10000; // scratch space  
  
    // Read 1st page off disk  
    readseg((uchar*)elf, 4096, 0);
```

# Drivers and Controllers



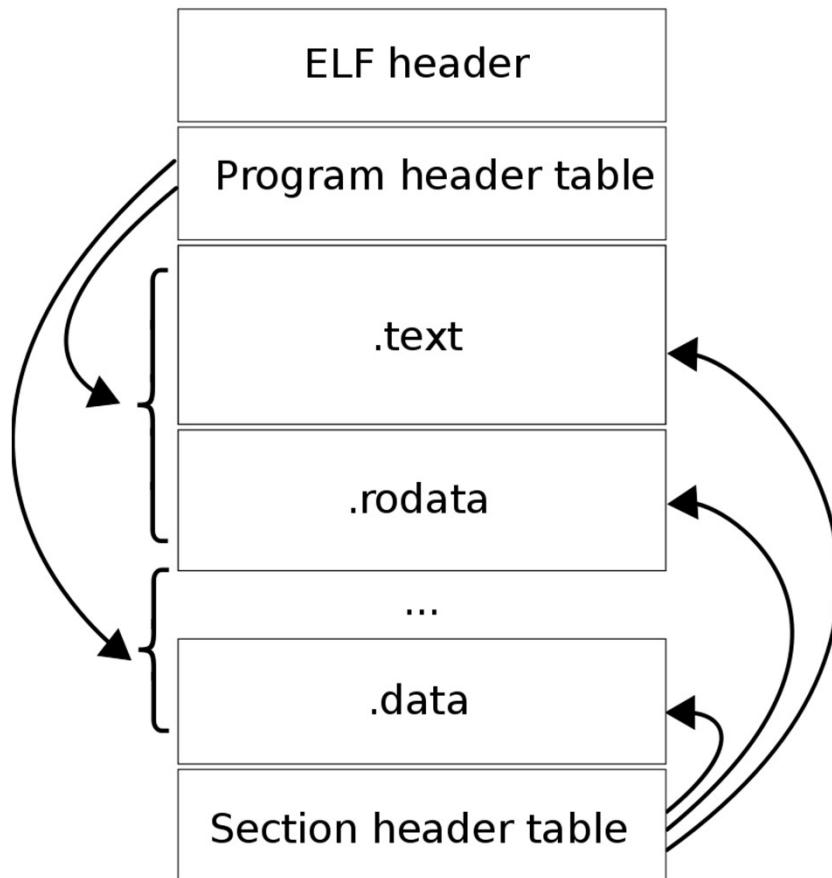
# bootmain()

② Check if it's really ELF  
or not

③ Next load kernel code  
from ELF file “kernel”  
into memory

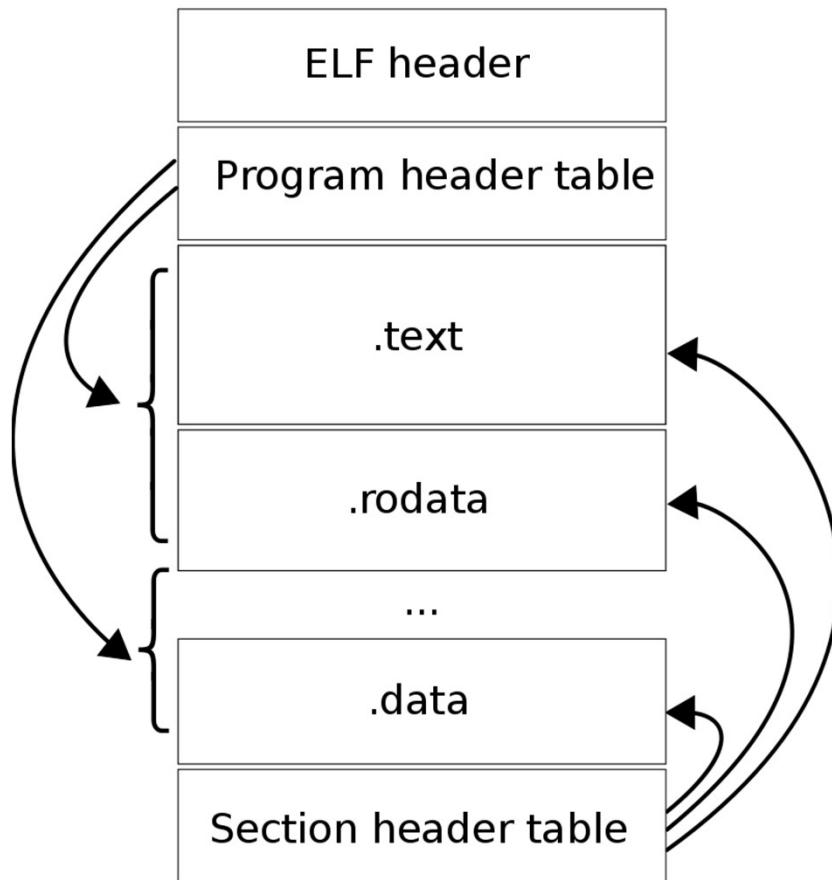
```
// Is this an ELF
executable?  
  
if(elf->magic != ELF_MAGIC)  
  
    return; // let bootasm.S  
handle error
```

# ELF



```
struct elfhdr {  
    uint magic; // must equal ELF_MAGIC  
    uchar elf[12];  
    ushort type;  
    ushort machine;  
    uint version;  
    uint entry;  
    uint phoff; // where is program header table  
    uint shoff;  
    uint flags;  
    ushort ehsiz;  
    ushort phentsize;  
    ushort phnum; // no. Of program header entries  
    ushort shentsize;  
    ushort shnum;  
    ushort shstrndx;  
};
```

# ELF



// Program header

struct proghdr {

    uint type; // Loadable segment ,  
    Dynamic linking information ,  
    Interpreter information , Thread-  
    Local Storage template , etc.

    uint off; //Offset of the segment in  
    the file image.

    uint vaddr; //Virtual address of the  
    segment in memory.

    uint paddr; // physical address to  
    load this program, if PA is relevant

    uint filesz; //Size in bytes of the  
    segment in the file image.

    uint memsz; //Size in bytes of the  
    segment in memory. May be 0.

    uint flags;

    uint align;

};

# Run ‘objdump -x -a kernel | head -15’ & see this

kernel: file format elf32-i386

kernel

architecture: i386, flags 0x00000112:

EXEC\_P, HAS\_SYMS, D\_PAGED

start address 0x0010000c

Code to be loaded at KERNBASE + KERNLINK

Program Header:

LOAD off 0x00001000 vaddr 0x80100000 paddr 0x00100000 align 2\*\*12  
filesz 0x0000a516 memsz 0x000154a8 flags rwx

STACK off 0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2\*\*4  
filesz 0x00000000 memsz 0x00000000 flags rwx

Diff between  
memsz & filesz, will be filled with  
zeroes in memory

Stack : everything zeroes

# Load code from ELF to memory

```
// Load each program segment (ignores ph flags).  
  
ph = (struct proghdr*)((uchar*)elf + elf->phoff);  
  
eph = ph + elf->phnum;  
  
// Abhijit: number of program headers  
  
for(; ph < eph; ph++) {  
  
    // Abhijit: iterate over each program header  
  
    pa = (uchar*)ph->paddr;  
  
    // Abhijit: the physical address to load program  
    /* Abhijit: read ph->filesz bytes, into 'pa',  
       from ph->off in kernel/disk */  
  
    readseg(pa, ph->filesz, ph->off);  
  
    if(ph->memsz > ph->filesz)  
        stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz); //  
        Zero the remainder section*/  
  
}
```

# Jump to Entry

```
// Call the entry point from the ELF header.  
// Does not return!  
/* Abhijit:  
 * elf->entry was set by Linker using kernel.1d  
 * This is address 0x80100000 specified in  
kernel.1d  
 * See kernel.asm for kernel assembly code).  
 */  
entry = (void(*)(void))(elf->entry) ;  
entry() ;
```

# Before code of entry(), Reminder: PDE looks like this

31

| Page table physical page number | A<br>V<br>L | G<br>S | P<br>0 | O<br>A | C<br>D | W<br>T | U<br>U | W<br>W | P<br>P |
|---------------------------------|-------------|--------|--------|--------|--------|--------|--------|--------|--------|
|---------------------------------|-------------|--------|--------|--------|--------|--------|--------|--------|--------|

PDE

P Present  
W Writable  
U User

WT 1=Write-through, 0=Write-back

CD Cache disabled

A Accessed

D Dirty

PS Page size (0=4KB, 1=4MB)

PAT Page table attribute index

G Global page

AVL Available for system use

31

| Physical page number | A<br>V<br>L | G<br>T | P<br>A | D<br>D | A<br>A | C<br>D | W<br>T | U<br>U | W<br>W | P<br>P |
|----------------------|-------------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
|----------------------|-------------|--------|--------|--------|--------|--------|--------|--------|--------|--------|

PTE

To understand  
further  
code

Remember: 4  
MB pages  
are possible

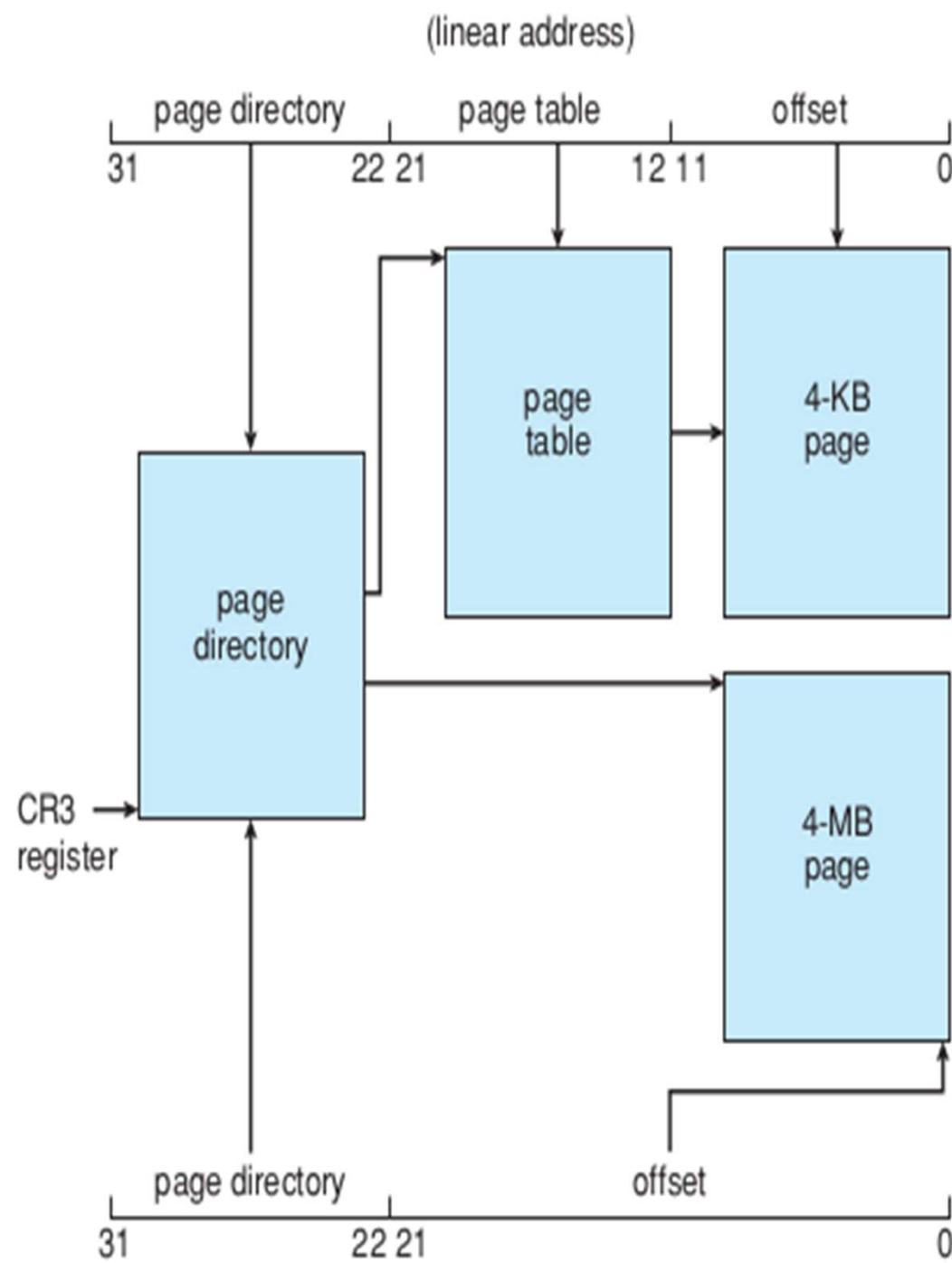
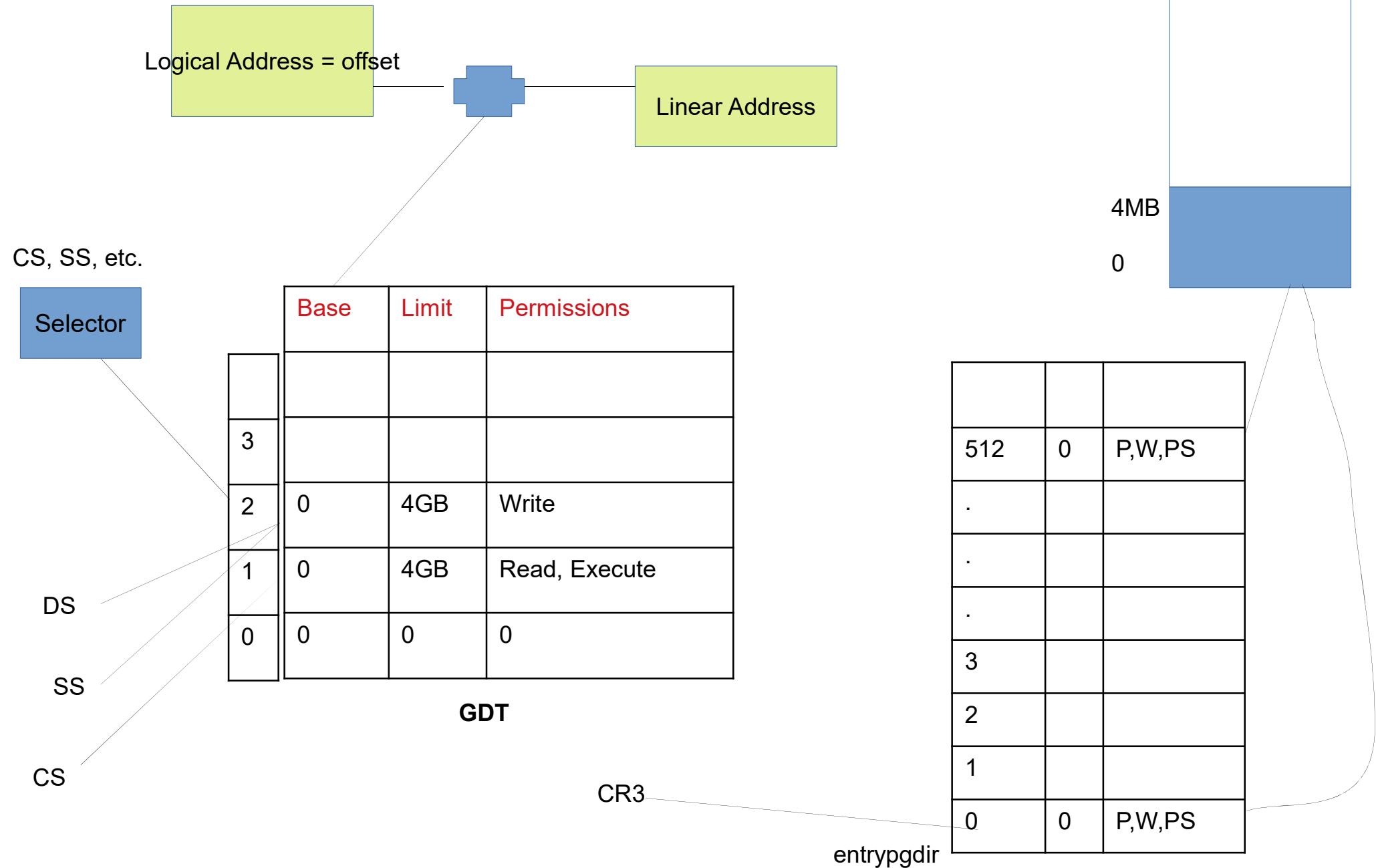


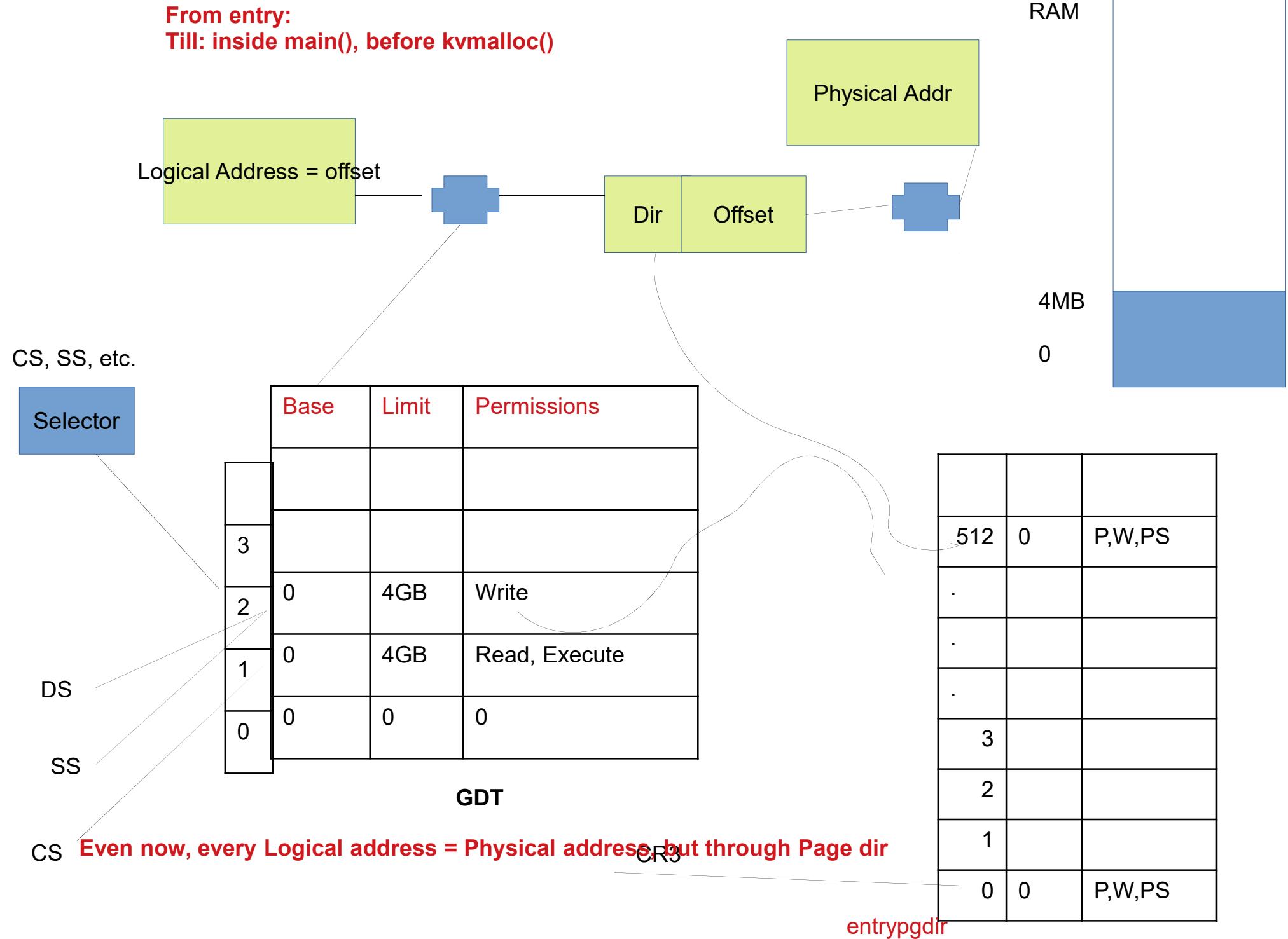
Figure 8.23 Paging in the IA-32 architecture.

**From entry:  
Till: inside main(), before kvmalloc()**



From entry:

Till: inside main(), before kvmalloc()



# entrypgdir in main.c, is used by entry()

```
__attribute__((aligned(PGSIZE)))  
  
pde_t entrypgdir[NPDENTRIES] = {  
  
    // Map VA's [0, 4MB) to PA's [0, 4MB)  
    [0] = (0) | PTE_P | PTE_W | PTE_PS,  
  
    // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB). This is entry 512  
    [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,  
  
};
```

```
#define PTE_P      0x001 // Present  
#define PTE_W      0x002 // Writeable  
#define PTE_U      0x004 // User  
#define PTE_PS     0x080 // Page Size  
#define PDXSHIFT   22    // offset of  
                      // PDX in a linear address
```

This is entry page directory during entry(), beginning of kernel

Mapping 0:0x400000 (i.e. 0: 4MB) to physical addresses 0:0x400000. is required as long as entry is executing at low addresses, but will eventually be removed.

This mapping restricts the kernel instructions and data to 4 Mbytes.

# entry() in entry.S

entry:

```
movl %cr4, %eax  
orl $(CR4_PSE), %eax  
movl %eax, %cr4  
movl $(V2P_WO(entrypgdir)),  
%eax  
movl %eax, %cr3  
movl %cr0, %eax  
orl $(CR0_PG|CR0_WP), %eax  
movl %eax, %cr0  
movl $(stack + KSTACKSIZE),  
%esp  
mov $main, %eax  
jmp *%eax
```

- ② # Turn on page size extension for 4Mbyte pages
- ② # Set page directory. 4 MB pages (temporarily only. More later)
- ② # Turn on paging.
- ② # Set up the stack pointer.
- ② # Jump to main(), and switch to executing at high addresses. The indirect call is needed because the assembler produces a PC-relative instruction for a direct jump.

# More about entry()

```
movl $(V2P_WO(entrypgdir)), %eax  
movl %eax, %cr3
```

-> Here we use physical address using V2P\_WO because paging is not turned on yet

■ **V2P is simple:**  
**subtract 0x80000000**  
**i.e. KERNBASE from address**

# More about entry()

```
movl %cr0, %eax  
orl $(CR0_PG|CR0_WP), %eax  
movl %eax, %cr0
```

This turns on paging

After this also, entry() is running and processor is executing code at lower addresses

But we have already set 0'th entry in pgdir to address 0

So it still works!

# entry()

```
movl $(stack +  
KSTACKSIZE), %esp
```

```
mov $main, %eax  
jmp *%eax
```

```
.comm stack,  
KSTACKSIZE
```

# Abhijit: allocate here 'stack' of size  
= KSTACKSIZE

- # Set up the stack pointer.
- # Abhijit: +KSTACKSIZE is done as stack grows downwards
- # Jump to main(), and switch to executing at high addresses. The indirect call is needed because the assembler produces a PC-relative instruction for a direct jump.

# **bootasm.S bootmain.c: Steps**

- 1) Starts in “real” mode, 16 bit mode. Does some misc legacy work.**
- 2) Runs instructions to do MMU set-up for protected-mode & only segmentation (0-4GB, identity mapping), changes to protected mode.**
- 3) Reads kernel ELF file and loads it in RAM, as per instructions in ELF file**
- 4) Sets up paging (4 MB pages)**
- 5) Runs main() of kernel**

Code from bootasm.S bootmain.c is over!

Kernel is loaded.

Now kernel is going to prepare itself

# main() in main.c

- Initializes “free list” of page frames
- In 2 steps. Why?
- Sets up page table for kernel
- Detects configuration of all processors
- Starts all processors
- Just like the first processor
- Creates the first process!

- Initializes
- LAPIC on each processor, IOAPIC
- Disables PIC
- “Console” hardware (the standard I/O)
- Serial Port
- Interrupt Descriptor Table
- Buffer Cache
- Files Table
- Hard Disk (IDE)

# main() in main.c

```
int main(void) {
    kinit1(end,
P2V(4*1024*1024)); // phys
page allocator

    kvmalloc();    // kernel page
table

    void kinit1(void *vstart, void
*vend) {
        initlock(&kmem.lock,
"kmem");

        kmem.use_lock = 0;

        freerange(vstart, vend);

    }
}
```

# main() in main.c

```
void
```

```
freerange(void *vstart, void *vend)
```

```
{
```

```
    char *p;
```

```
    p =  
        (char*)PGROUNDUP((uint)vstart);
```

```
    for(; p + PGSIZE <= (char*)vend; p  
        += PGSIZE)
```

```
        kfree(p);
```

```
}
```

```
kfree(char *v) {  
    struct run *r;  
    if((uint)v % PGSIZE || v <  
end || V2P(v) >= PHYSTOP)  
        panic("kfree");  
    // Fill with junk to catch  
    dangling refs.  
    memset(v, 1, PGSIZE);  
    if(kmem.use_lock)  
        acquire(&kmem.lock);  
    r = (struct run*)v;  
    r->next = kmem.freelist;  
    kmem.freelist = r;  
    if(kmem.use_lock)  
        release(&kmem.lock); }
```

lock  
uselock  
run \*freelist

run \*next

Allocated frame

run \*next

Allocated frame

Allocated frame

run \*next

run \*next

Allocated frame

RAM –  
divided  
into frames

## Free List in XV6 Obtained after main() -> kinit1()

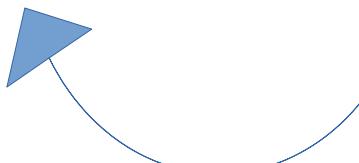
Pages obtained Between

**end = 801154a8 = 2049 MB to P2V(4MB) = 2052 MB**

Remember

Right now Logical = Physical address.

Actually like  
this in memory



lock  
uselock  
run \*freelist

kmem

Seen independently

run \*

run \*

run \*

# Back to main()

int

**main(void) {**

    kinit1(end,  
P2V(4\*1024\*1024)); //  
**phys page allocator**

    kvmalloc();    //  
**kernel page table**

// Allocate one page  
table for the machine  
for the kernel address  
// space for scheduler  
processes.

**void**

**kvmalloc(void)**  
{

**kpgdir = setupkvm();**

**switchkvm();**

}

# Back to main()

```
int  
main(void) {  
  
    kinit1(end,  
P2V(4*1024*1024)); //  
phys page allocator  
  
    kvmalloc(); //  
kernel page table
```

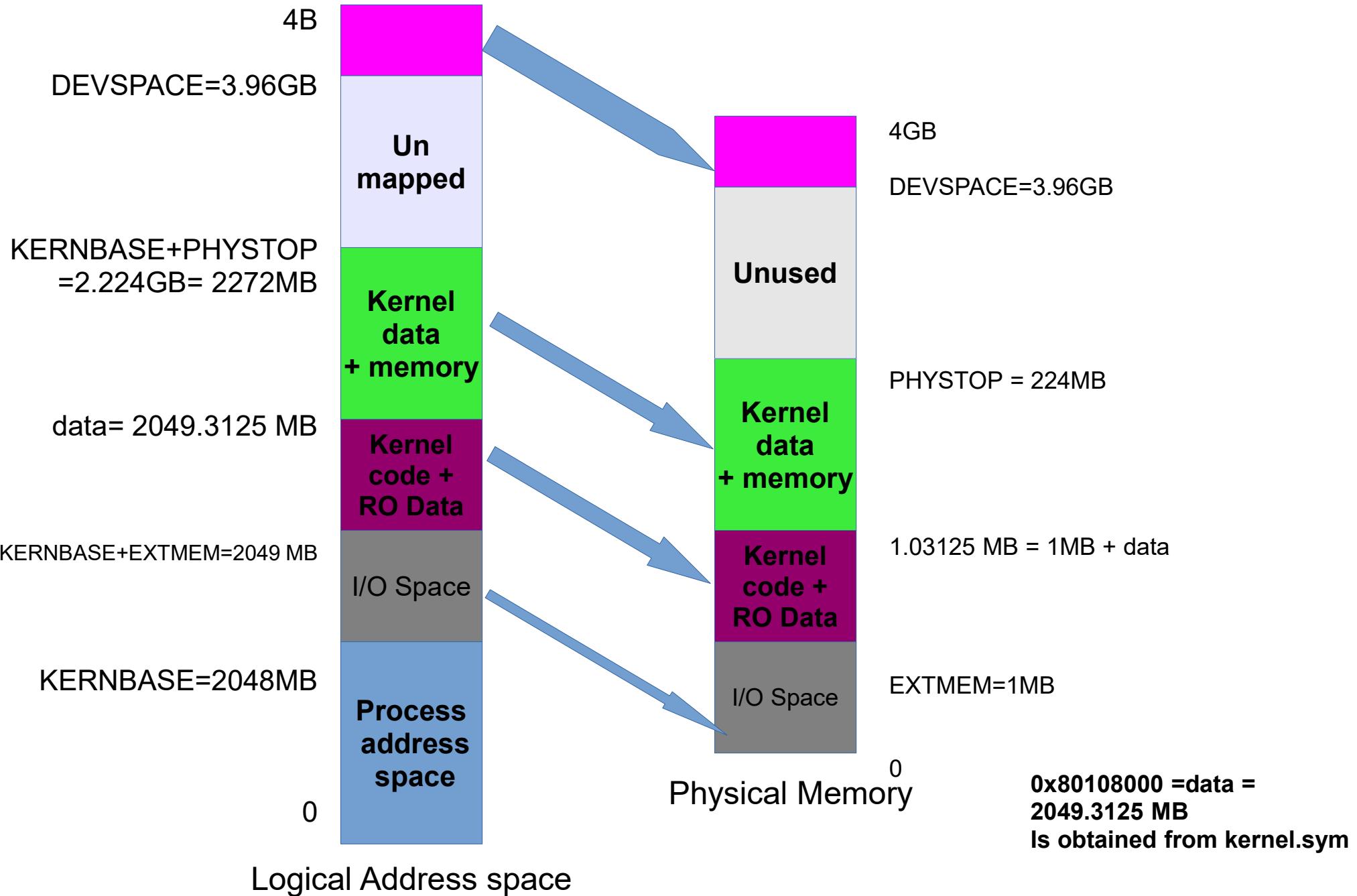
```
// Allocate one page  
table for the machine  
for the kernel address  
  
// space for scheduler  
processes.  
  
void  
kvmalloc(void)  
{  
  
    kpgdir = setupkvm();  
    // global var kpgdir  
  
    switchkvm();  
}
```

```
pde_t*
setupkvm(void)
{
    pde_t *pgdir;
    struct kmap *k;
    if((pgdir = (pde_t*)kalloc()) == 0)
        return 0;
    memset(pgdir, 0, PGSIZE);
    if (P2V(PHYSTOP) > (void*)DEVSPACE)
        panic("PHYSTOP too high");
    for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
        if(mappages(pgdir, k->virt, k->phys_end - k-
>phys_start,
            (uint)k->phys_start, k->perm) < 0) {
            freevm(pgdir);
            return 0;
        }
    return pgdir;
}
```

```
static struct kmap {  
    void *virt;  
    uint phys_start;  
    uint phys_end;  
    int perm;  
} kmap[] = {  
    { (void*)KERNBASE, 0,           EXTMEM,  PTE_W}, // I/O space  
    { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, // kern text+rodata  
    { (void*)data,   V2P(data),   PHYSTOP,  PTE_W}, // kern data+memory  
    { (void*)DEVSPACE, DEVSPACE, 0,      PTE_W}, // more devices  
};
```

kmap[] mappings done in kvmalloc().

This shows segmentwise, entries are done in page directory and page table for corresponding VA-> PA mappings



# Remidner: PDE and PTE entries

31

| Page table physical page number | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---------------------------------|----|----|----|---|---|---|---|---|---|---|---|---|---|
|                                 | A  | G  | P  | S | 0 | A | C | W | U | W | U | W | P |

PDE

P Present  
W Writable  
U User

WT 1=Write-through, 0=Write-back

CD Cache disabled

A Accessed

D Dirty

PS Page size (0=4KB, 1=4MB)

PAT Page table attribute index

G Global page

AVL Available for system use

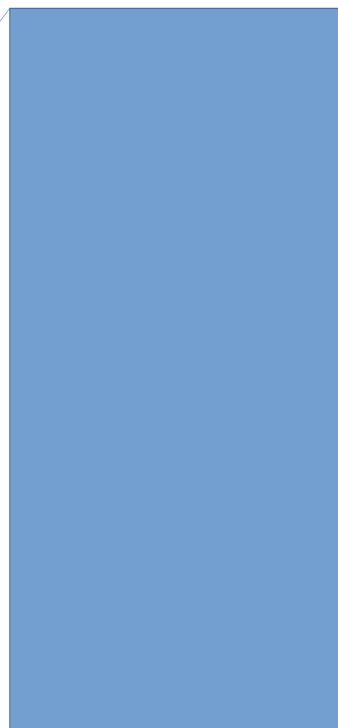
31

| Physical page number | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----------------------|----|----|----|---|---|---|---|---|---|---|---|---|---|
|                      | A  | G  | P  | A | D | A | C | W | U | W | U | W | P |

PTE

**Before mappages()**

pgdir



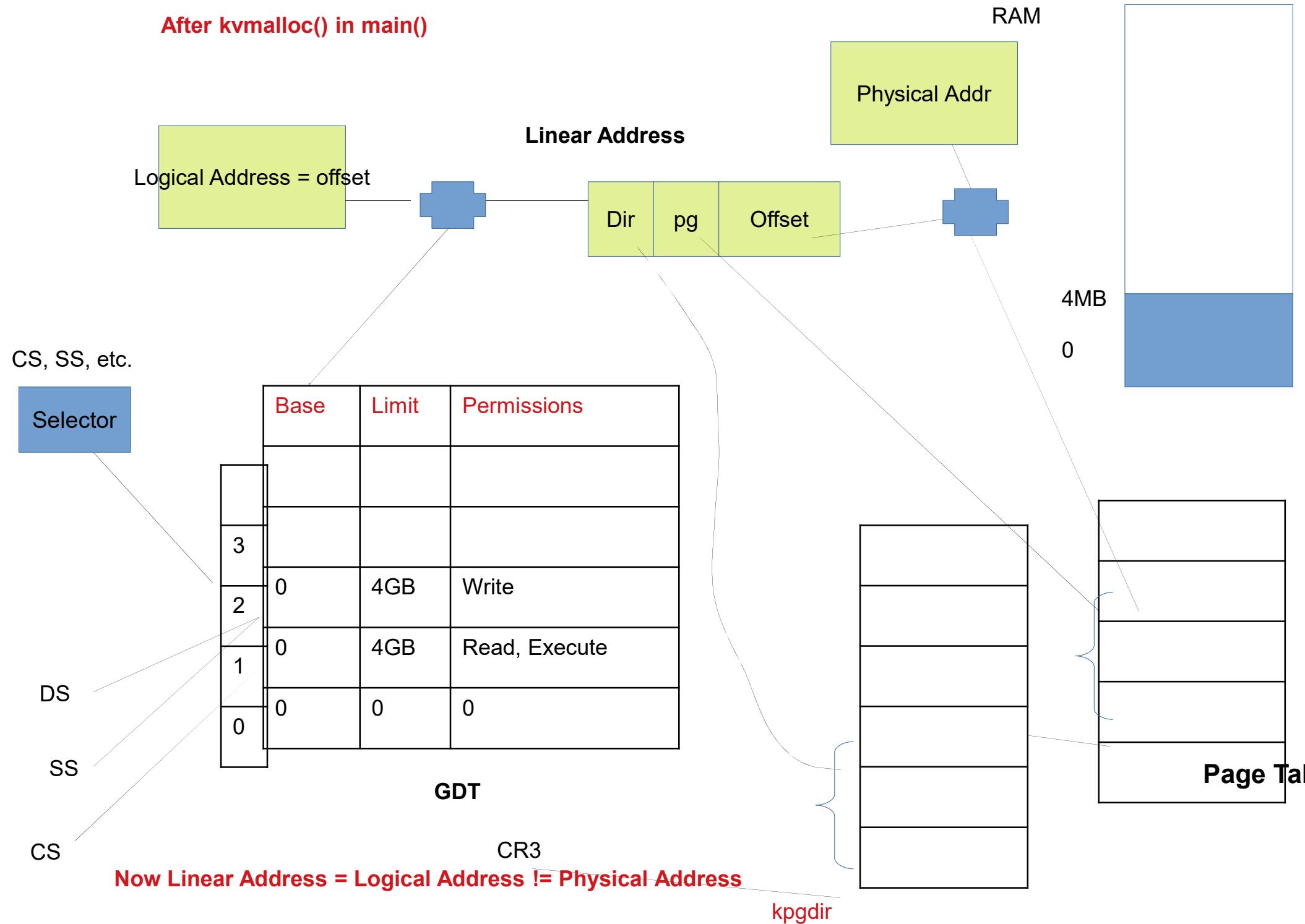
pgdir

Page-no,UWP  
Page-no,UWP  
Page-no,UWP  
Page-no,UWP  
Page-no,UWP  
Page-no,UWP  
Page-no,UWP  
Page-no,UWP  
Page-no,UWP



**What mappages()  
Does**

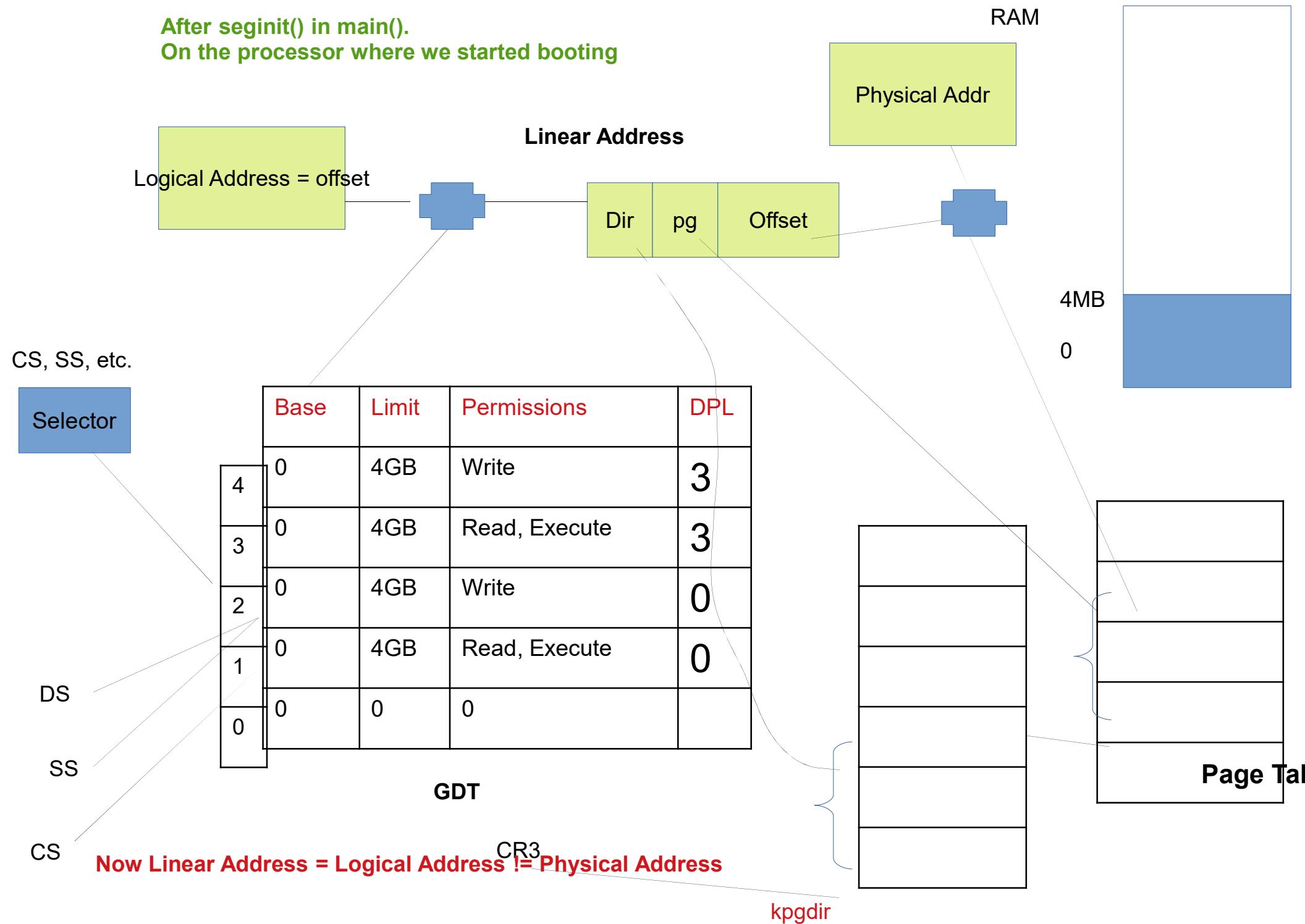
After kvmalloc() in main()



# **main() -> seginit()**

- Re-initialize GDT
- Once and forever now
- Just set 4 entries
- All spanning 4 GB
- Differing only in permissions and privilege level

After seginit() in main().  
On the processor where we started booting



# After seginit()

- While running kernel code, necessary to switch CS, DS, SS to index 1,2,2 in GDT
- While running user code, necessary to switch CS, DS, SS to index 3,4,4 in GDT
- This happens automatically as part of “trap” handling (covered separately)

# Scheduler

# Scheduler – in most simple terms

- Selects a process to execute and passes control to it !
  - The process is chosen out of “READY” state processes
  - Saving of context of “earlier” process and loading of context of “next” process needs to happen
- Questions
- What are the different scenarios in which a scheduler called ?
  - What are the intricacies of “passing control”
  - What is “context” ?

# Steps in scheduling

## scheduling

▪ Suppose you want to switch from P1 to P2 on a timer interrupt

▪ P1 was doing

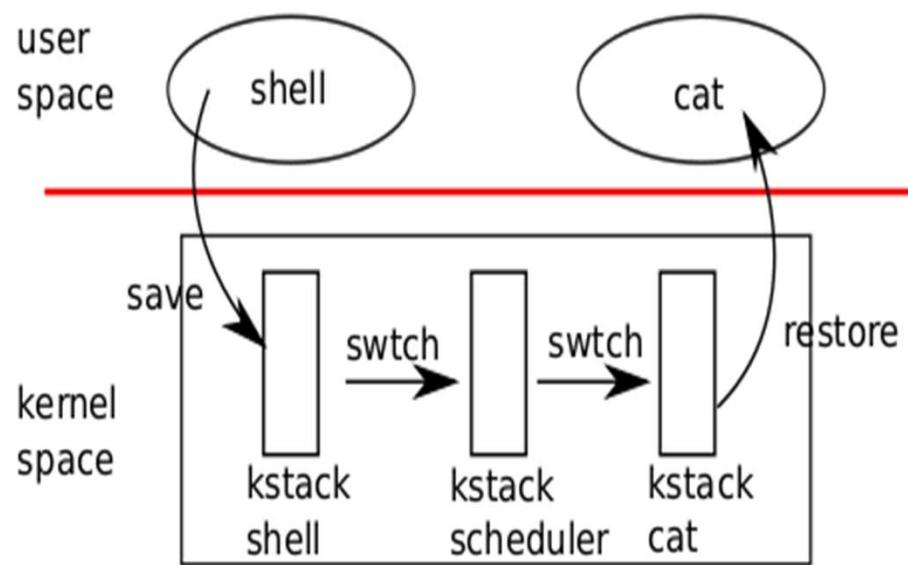
F() { i++; j++; }

▪ P2 was doing

G() { x--; y++; }

▪ P1 will experience a timer interrupt, switch to kernel (scheduler) and scheduler will schedule P2

# 4 stacks need to change!



- User stack of process ->
- kernel stack of process
- Switch to kernel stack
- The normal sequence on any interrupt !
- Kernel stack of process ->
- kernel stack of scheduler
- Why?
- Kernel stack of scheduler ->
- kernel stack of new process . Why?
- Kernel stack of new process ->
- user stack of new process

# **scheduler()**

- Enable interrupts
- Find a **RUNNABLE** process. Simple round-robin!
- **c->proc = p**
- **switchuvm(p) : Save TSS and make CR3 to point to new process pagedir**
- **p->state = RUNNING**
- **swtch(&(c->scheduler), p->context)**

# swtch

swtch:

```
    movl 4(%esp), %eax
```

```
    movl 8(%esp), %edx
```

```
# Save old callee-saved registers
```

```
    pushl %ebp
```

```
    pushl %ebx
```

```
    pushl %esi
```

```
    pushl %edi
```

```
# Switch stacks
```

```
    movl %esp, (%eax)
```

```
    movl %edx, %esp
```

```
# Load new callee-saved registers
```

```
    popl %edi
```

```
    popl %esi
```

```
    popl %ebx
```

```
    popl %ebp
```

```
    ret
```

# **scheduler()**

- **swtch(&(c->scheduler), p->context)**
- Note that when scheduler() was called, when P1 was running
- After call to swtch() shown above
  - The call does NOT return!
  - The new process P2 given by ‘p’ starts running !
  - Let’s review swtch() again

# swtch(old, new)

■ The magic function in swtch.S

■ Saves callee-save registers of old context

■ Switches esp to new-context's stack

■ Pop callee-save registers from new context

■ ret

■ where? in the case of first process – returns to forkret() because stack was setup like that !

■ in case of other processes, return where?

■ Return address given on kernel stack. But what's that?

■ The EIP in p->context

■ When was EIP set in p->context ?

# **scheduler()**

↳ Called from?

↳ **mpmain()**

↳ No where else!

↳ **sched() is another scheduler function !**

↳ Who calls sched() ?

↳ **exit()** - a process exiting calls sched ()

↳ **yield()** - a process interrupted by timer calls yield()

↳ **sleep()** - a process going to wait calls sleep()

```

void
sched(void)
{
    int intena;
    struct proc *p = myproc();
    if(!holding(&ptable.lock))
        panic("sched ptable.lock");
    if(mycpu()->ncli != 1)
        panic("sched locks");
    if(p->state == RUNNING)
        panic("sched running");
    if(readeflags()&FL_IF)
        panic("sched interruptible");
    intena = mycpu()->intena;
    swtch(&p->context, mycpu()->scheduler);
    /*A*/ mycpu()->intena = intena;
}

```

# sched()

▪ get current process

▪ Error checking code (ignore as of now)

▪ get interrupt enabled status on current CPU (ignore as of now)

▪ call to swtch

▪ Note the arguments' order

▪ p->context first, mycpu()->scheduler second

▪ swtch() is a function call

▪ pushes address of /\*A\*/ on stack of current process p

▪ switches stack to mycpu()->scheduler. Then pops EIP from that stack and jumps there.

▪ when was mycpu()->scheduler set?  
Ans: during scheduler()!

# **sched() and scheduler()**

```
 sched() {  
 ...  
 swtch(&p->context, mycpu()->scheduler); /* X */  
 }
```

```
 scheduler(void) {  
 ...  
 swtch(&(c->scheduler), p-  
 >context); /* Y */  
 }
```

- **scheduler() saves context in c->scheduler, sched() saves context in p->context**
- **after swtch() call in sched(), the control jumps to Y in scheduler**
- **Switch from process stack to scheduler's stack**
- **after swtch() call in scheduler(), the control jumps to X in sched()**
- **Switch from scheduler's stack to new process's stack**
- **Set of co-operating functions**

# **sched() and scheduler() as co-routines**

- In sched()**

```
swtch(&p->context, mycpu()->scheduler);
```

- In scheduler()**

```
swtch(&(c->scheduler), p->context);
```

- These two keep switching between processes**

- These two functions work together to achieve scheduling**

- Using asynchronous jumps**

- Hence they are co-routines**

# To summarize

- On a timer interrupt during P1

- trap() is called. Stack has changed from P1's user stack to P1's kernel stack

- trap()->yield()

- yield()->sched()

- sched() -> swtch(&p->context, c->scheduler())

- Stack changes to scheduler's kernel stack.

- Switches to location “Y” in scheduler().

- Now the loop in scheduler()

- calls switchkvm()

- Then continues to find next process (P2) to run

- Then calls switchuvvm(p): changing the page table to the P2's page tables

- then calls swtch(&c->scheduler, p2's->context)

- Stack changes to P2's kernel stack.

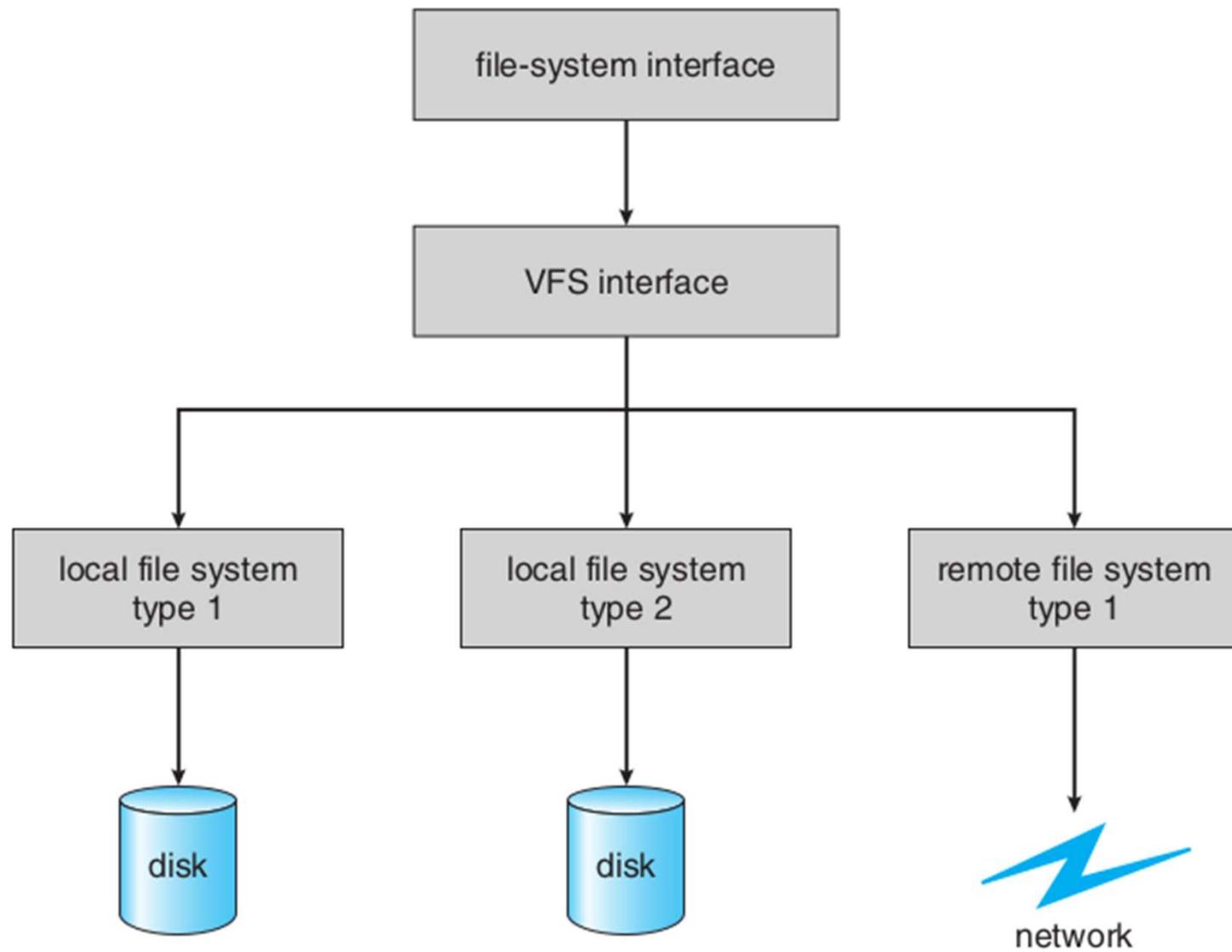
- P2 runs the last instruction it was in ! Where was it?

- mycpu()->intena = intena; in sched()

- Then returns to the one who called sched() i.e. exit/sleep, etc

- Finally returns from its own “TRAP” handler and returns to P2's user stack and user code

**VFS**



**Figure 15.5** Schematic view of a virtual file system.

# **Illustrative example of VFS implementation**

# **File System Code**

**open, read, write, close, pipe, fstat, chdir,  
dup, mknod, link, unlink, mkdir,**

**Files, Inodes, Buffers**

# What we already know

- File system related system calls

- deal with ‘fd’ arrays (**ofile** in xv6). **open()** returns first empty index. **open** should ideally locate the inode on disk and initialize some data structures
- maintain ‘offsets’ within a ‘file’ to support sequential read/write
- **dup()** like system calls duplicate pointers in fd-array
- read/write like system calls, going through ‘**ofile**’ array, should locate data of file on disk
- We need functions to read/write from disk – that is **disk driver**
- cache data of files in OS data structures for performance : **buffering**
- Need to handle on disk data structures as well
- Faster recovery (like journaling in ext3) is desired

# xv6 file handling code

- Is a very good example in ‘design’ of a layered and modular architecture
- Splits the entire work into different modules, and modules into functions properly
- The task of each function is neatly defined and compartmentalized

# Layers of xv6 file system code

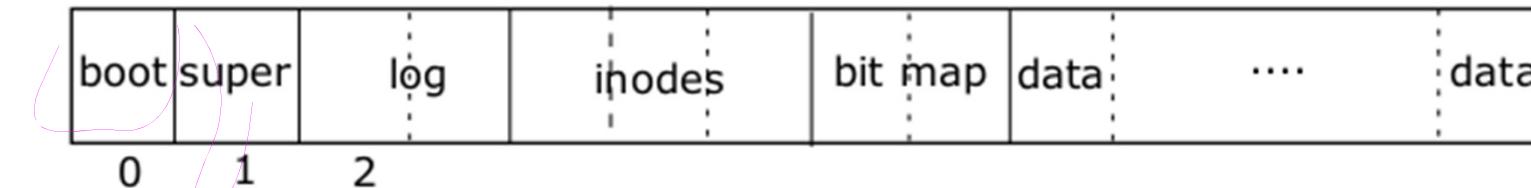
|                 |                                                                                                                      |
|-----------------|----------------------------------------------------------------------------------------------------------------------|
| System Calls    | open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,                                       |
| File descriptor | <b>file.c</b> fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,                                |
| Pathname        | <b>fs.c</b> namex, namei, nameiparent, skipellem                                                                     |
| Directory       | <b>fs.c</b> dirlookup, dirlink                                                                                       |
| Inode           | <b>fs.c</b> iiinit, ialloc, iupdate, igit, idup, ilock, unlock, iput, unlockput, itrunc, stati, readi, writei, bmap, |
| Logging         | Block allocation on disk: <b>balloc, bfree</b><br><b>log.c</b> : begin_op, end_op, initlog, commit,                  |
| Buffer cache    | <b>bio.c</b> binit, bget, bread, bwrite, brelse                                                                      |
| Disk            | <b>ide.c</b> : idewait, ideinit, idestart, ideintr, iderw                                                            |

Normally, any upper layer can call any lower layer below

**Abhijit: Block allocator should be considered as another Layer!**

# Layout of xv6 file system

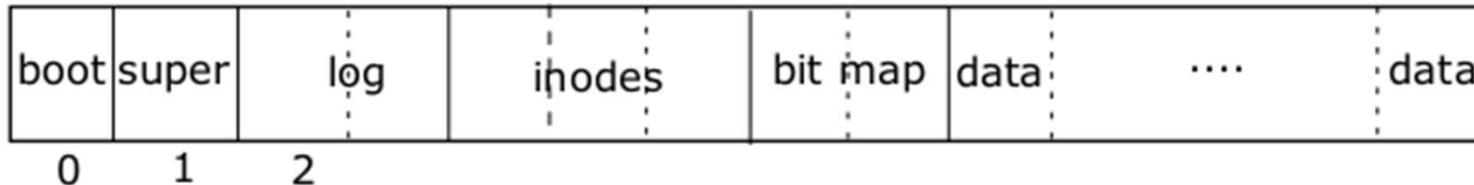
Pointer shown are concept



May see the code of mkfs.c to get insight into the layout

```
struct superblock {  
    uint size;      // Size of file system image (blocks)  
    uint nblocks;   // Number of data blocks  
    uint ninodes;   // Number of inodes.  
    uint nlog;      // Number of log blocks  
    uint logstart;  // Block number of first log block  
    uint inodestart; // Block number of first inode block  
    uint bmapstart; // Block number of first free map block  
};  
#define ROOTINO 1 // root i-number  
#define BSIZE 512 // block size
```

# Layout of xv6 file system

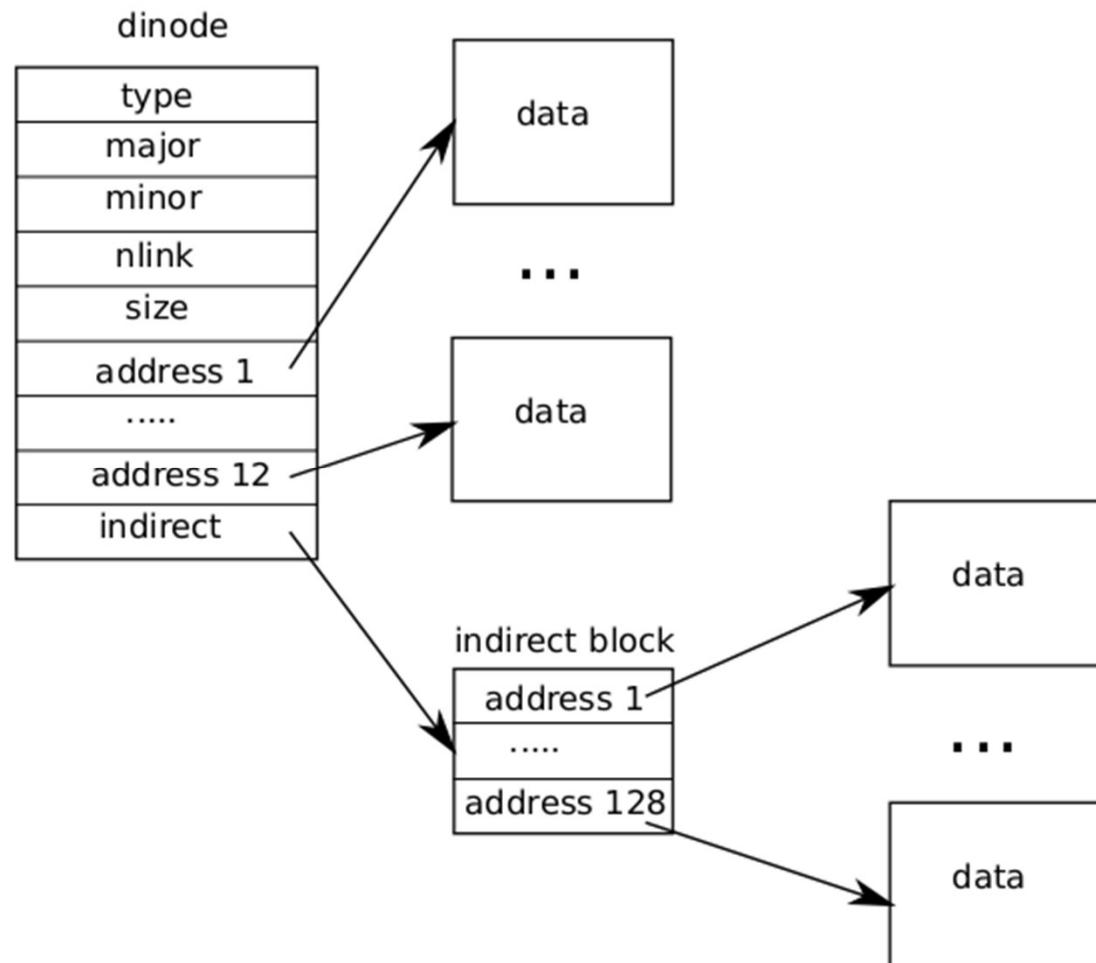


```
#define NDIRECT 12
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT + NINDIRECT)
// On-disk inode structure
struct dinode {
    short type;          // File type
    short major;         // Major device number (T_DEV only)
    short minor;         // Minor device number (T_DEV only)
    short nlink;         // Number of links to inode in file system
    uint size;           // Size of file (bytes)
    uint addrs[NDIRECT+1]; // Data block addresses
};
```

```
#define DIRSIZ 14
```

```
struct dirent {
    ushort inum;
    char name[DIRSIZ];
};
```

# File on disk



# Let's discuss lowest layer first

|                 |   |                                                                                                                         |
|-----------------|---|-------------------------------------------------------------------------------------------------------------------------|
| System Calls    |   | open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,                                          |
| File descriptor | → | <b>file.c</b> fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,                                   |
| Pathname        | → | <b>fs.c</b> namex, namei, nameiparent, skipellem                                                                        |
| Directory       | → | <b>fs.c</b> dirlookup, dirlink                                                                                          |
| Inode           | → | <b>fs.c</b> iiinit, ialloc, iupdate, igeet, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap, |
| Logging         | → | Block allocation on disk: <b>balloc, bfree</b><br><b>log.c</b> : begin_op, end_op, initlog, commit,                     |
| Buffer cache    | → | <b>bio.c</b> binit, bget, bread, bwrite, brelse                                                                         |
| Disk            | → | <b>ide.c</b> : idewait, ideinit, idestart, ideintr, iderw                                                               |

Normally, any upper layer can call any lower layer below

# ide.c: idewait, ideinit, idestart, ideintr, iderw

static struct spinlock idelock;

static struct buf \*idequeue;

static int havedisk1;

■ ideinit

■ was called from main.c: main()

■ Initialized IDE controller by writing to certain ports

■ havedisk=1 setup

■ Initialize idelock

■ idewait

■ BUSY loop waiting for IDE to be ready

# ide.c: idewait, ideinit, idestart, ideintr, iderw

- **void idestart(buf \*b)**

- **static void idestart(struct buf \*b)**

- Calculate sector number on disk using b->blockno

- Issue a read/write command to IDE controller.

- (This is the first buf on **idequeue**)

- **ideintr**

- Take **idelock**. Called on IDE interrupt (through alltraps()->trap())

- Wakeup the process waiting on first buffer in **buffer \*idequeue**;

- call **idestart()**. Release **idelock**.

- **iderw(buf \*b)**

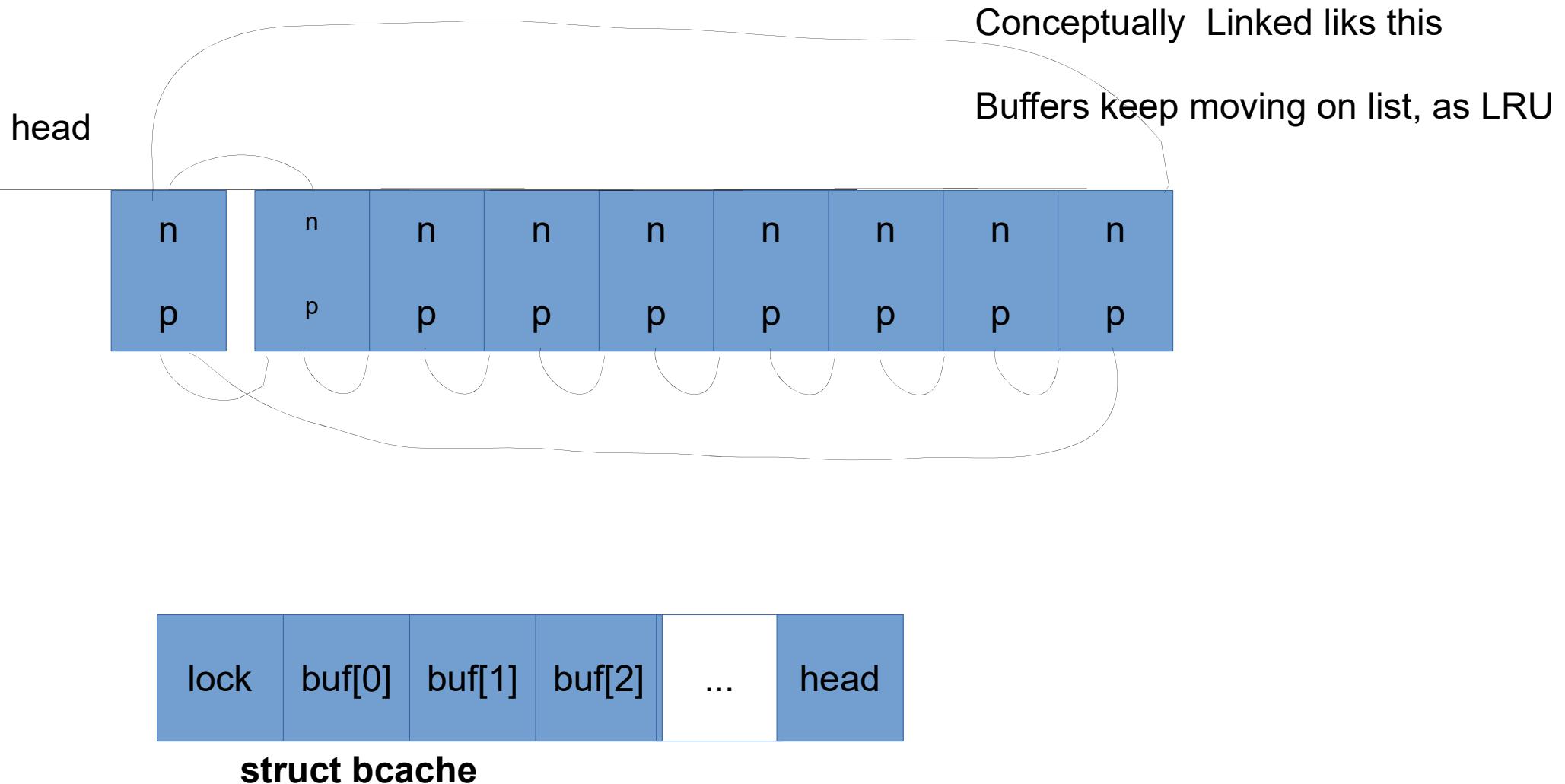
- Move **buf b** to end of **idequeue**

- Call **idestart()** if not running, sleep on **idelock**

# Let's see buffer cache layer

|                                                          |   |                                                                                                                        |
|----------------------------------------------------------|---|------------------------------------------------------------------------------------------------------------------------|
| System Calls                                             |   | open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,                                         |
| File descriptor                                          | → | <b>file.c</b> fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,                                  |
| Pathname                                                 | → | <b>fs.c</b> namex, namei, nameiparent, skipellem                                                                       |
| Directory                                                | → | <b>fs.c</b> dirlookup, dirlink                                                                                         |
| Inode                                                    | → | <b>fs.c</b> iiinit, ialloc, iupdate, igit, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap, |
| Logging                                                  | → | Block allocation on disk: <b>balloc, bfree</b><br><b>log.c</b> : begin_op, end_op, initlog, commit,                    |
| Buffer cache                                             | → | <b>bio.c</b> binit, bget, bread, bwrite, brelse                                                                        |
| Disk                                                     | → | <b>ide.c</b> : idewait, ideinit, idestart, ideintr, iderw                                                              |
| Normally, any upper layer can call any lower layer below |   |                                                                                                                        |

# Reminder: After main() -> binit()



# struct buf

```
struct buf {  
    int flags; // 0 or B_VALID or B_DIRTY  
    uint dev; // device number  
    uint blockno; // seq block number on device  
    struct sleeplock lock; // Lock to be held by process using it  
    uint refcnt; // Number of live accesses to the buf  
    struct buf *prev; // cache list  
    struct buf *next; // cache list  
    struct buf *qnext; // disk queue  
    uchar data[BSIZE]; // data 512 bytes  
};  
#define B_VALID 0x2 // buffer has been read from disk  
#define B_DIRTY 0x4 // buffer needs to be written to disk
```

# buffer cache:

## static struct buf\* bget(uint dev, uint blockno)

- The bcache.head list is maintained on Most Recently Used (MRU) basis
- head.next is the Most Recently Used (MRU) buffer
- hence head.prev is the Least Recently Used (LRU)
- Look for a buffer with b->blockno = blockno and b->dev = dev
- Search the head.next list for existing buffer (MRU order)
- Else search the head.prev list for empty buffer
- panic() if found in-use or empty buffer
- Increment b->refcnt ; Returns buffer locked
- Does not change the list structure, just returns a buf in use

# buffer cache:

## struct buf\* bread(uint dev, uint blockno)

```
struct buf*
bread(uint dev, uint blockno)
{
    struct buf *b;
    b = bget(dev, blockno);
    if((b->flags & B_VALID) == 0) {
        iderw(b);
    }
    return b; // locked buffer
}
```

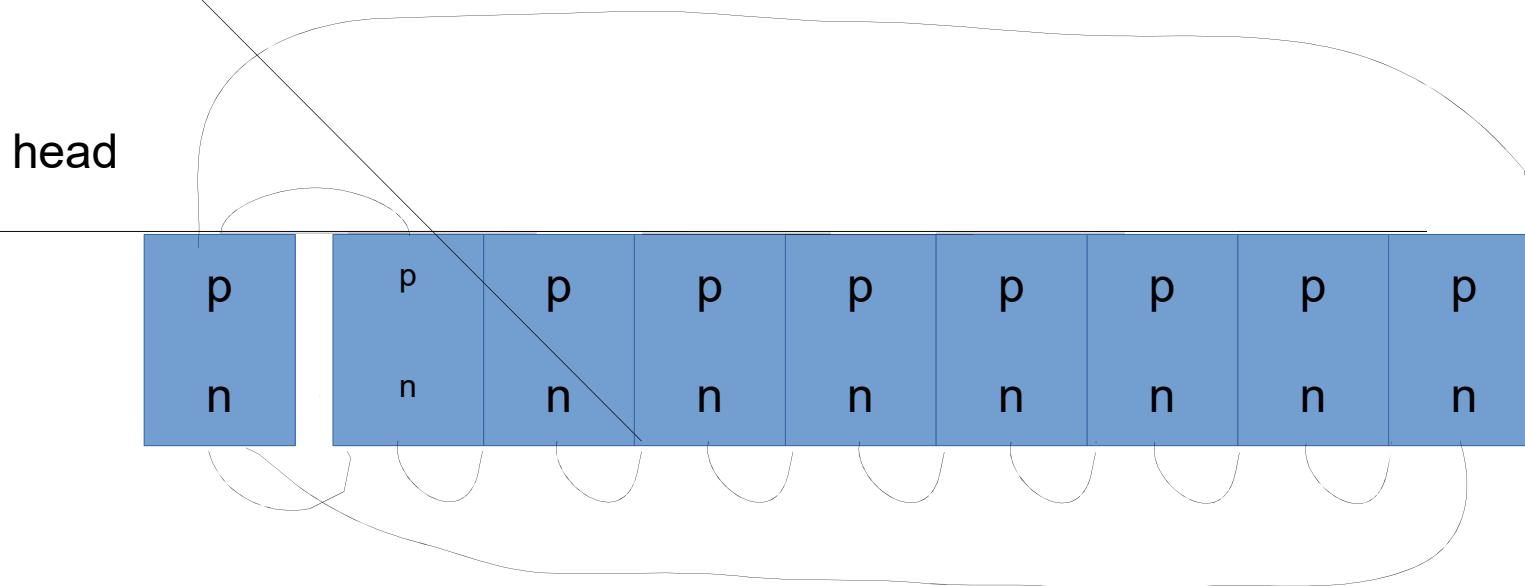
```
void
bwrite(struct buf *b)
{
    if(!holdingsleep(&b->lock))
        panic("bwrite");
    b->flags |= B_DIRTY;
    iderw(b);
}
```

Recollect: `iderw` moves buf to tail of `idequeue`, calls `idestart()` and `sleep()`

## buffer cache: **void brelse(struct buf \*b)**

- ❑ release lock on buffer
- ❑ **b->refcnt = 0**
- ❑ If **b->refcnt = 0**
  - ❑ Means buffer will no longer be used
  - ❑ Move it to front of the front of **bcache.head**

# Overall in this diagram



Buffers keep moving to the front of the list and around  
The list always contains **NBUF=30** buffers  
**head.next** is always the MRU and **head.prev** is always LRU buffer

# File descriptor layer code

|                 |                                                                                                                                 |
|-----------------|---------------------------------------------------------------------------------------------------------------------------------|
| System Calls    | open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,                                                  |
| File descriptor | <b>file.c</b> fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,                                           |
| Pathname        | <b>fs.c</b> namex, namei, nameiparent, skipelem                                                                                 |
| Directory       | <b>fs.c</b> dirlookup, dirlink                                                                                                  |
| Inode           | <b>fs.c</b> iiinit, ialloc, iupdate, igeet, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, <b>bmap</b> , |
| Logging         | Block allocation on disk: <b>balloc</b> , <b>bfree</b>                                                                          |
| Buffer cache    | <b>log.c</b> : begin_op, end_op, initlog, commit,                                                                               |
| Disk            | <b>bio.c</b> binit, bget, bread, bwrite, brelse                                                                                 |
|                 | <b>ide.c</b> : idewait, ideinit, idestart, ideintr, iderw                                                                       |

# data structures related to “file” layer

```
struct file {  
    enum { FD_NONE, FD_PIPE, FD_INODE } type;  
    int ref; // reference count  
    char readable;  
    char writable;  
    struct pipe *pipe; // used only if it  
    works as a pipe  
    struct inode *ip;  
    uint off;  
};  
// interesting no lock in struct file !
```

```
struct proc {  
    ...  
    struct file *ofile[NOFILE]; // Open files per  
    process  
    ...  
}  
struct {  
    struct spinlock lock;  
    struct file file[NFILE];  
} ftable; //global table from which ‘file’ is  
allocated to every process  
  
Lock is used to protect updates to every entry in  
the array
```

# Multiple processes accessing same file.

- ☞ Each will get a different ‘struct file’
- ☞ but share the inode !
- ☞ different offset in struct file, for each process
- ☞ Also true, if same process opens file many times
- ☞ File can be a PIPE (more later)
- ☞ what about STDIN, STDOUT, STDERR files ?
- ☞ Figure out!
- ☞ ref
  - ☞ used if the file was ‘duped’ or process forked . in that case the ‘struct file’ is shared

# file layer functions

▫ **filealloc**

▫ find an empty struct  
file in ‘ftable’ and  
return it

▫ set ref = 1

▫ **filedup(file \*)**

▫ simply ref++

▫ **fclose**

▫ **--ref**

▫ if ref = 0

▫ free struct file

▫ iput() / pipeclose()

▫ note – transaction if iput()  
called

▫ **filestat**

▫ simply return fields from  
inode, after holding lock. on  
inodes for files only.

# file layer functions

- **fileread**
  - **call readi() or piperead()**
  - **readi() later calls device-read or inode read (using bread())**
  
  - **filewrite**
  - **call pipewrite() or writei()**
  - **writei() is called in a loop, within a transaction**
- **Why does readi() call read on the device , why not fileread() itself call device read ?**

# Reading Directory Layer

|                 |   |                                                                                                                                |
|-----------------|---|--------------------------------------------------------------------------------------------------------------------------------|
| System Calls    |   | open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,                                                 |
| File descriptor | → | <b>file.c</b> fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,                                          |
| Pathname        | → | <b>fs.c</b> namex, namei, nameiparent, skipelem                                                                                |
| Directory       | → | <b>fs.c</b> dirlookup, dirlink                                                                                                 |
| Inode           | → | <b>fs.c</b> iiinit, ialloc, iupdate, igit, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, <b>bmap</b> , |
| Logging         | → | Block allocation on disk: <b>balloc</b> , <b>bfree</b>                                                                         |
| Buffer cache    | → | <b>log.c</b> : begin_op, end_op, initlog, commit,                                                                              |
| Disk            | → | <b>bio.c</b> binit, bget, bread, bwrite, brelse                                                                                |
|                 |   | <b>ide.c</b> : idewait, ideinit, idestart, ideintr, iderw                                                                      |

# directory entry

```
#define DIRSIZ 14
```

```
struct dirent {  
    ushort inum;  
    char name[DIRSIZ];  
};
```

Data of a directory file is a sequence of such entries. To find a name, just get all the data blocks and search the name

How to get the data for a directory? We already know the ans!

```
struct inode*
dirlookup(struct inode *dp, char *name, uint *poff)
```

- Given a pointer to directory inode (dp), name of file to be searched
- return the pointer to inode of that file (NULL if not found)
- set the ‘offset’ of the entry found, inside directories data blocks, in poff
- How was ‘dp’ obtained? Who should be calling dirlookup? Why is poff returned?
- During resolution of pathnames?
- Code: call readi() to get data of dp, search name in it, name comes with inode-num, iget() that inode-num

```
int  
dirlink(struct inode *dp, char *name, uint inum)
```

- „Create a new entry for ‘name’ \_ ‘inum’ in directory given by ‘dp’
- „inode number must have been obtained before calling this. How to do that?
- „Use dirlookup() to verify entry does not exist!
- „Get empty slot in directory’s data block
- „Make directory entry
- „Update directory inode! writei()

# namex

- Called by namei(), or nameiparent()
- Just iteratively split a path using “/” separator and get inode for last component
- iget() root inode, then
  - Repeatedly calls
  - split on “/”, dirlookup() for next component

# races in namex()

- Crucial. Called so many times!
- one kernel thread is looking up a pathname another kernel thread may be changing the directory by calling unlink
- when executing dirlookup in namex, the lookup thread holds the lock on the directory and dirlookup() returns an inode that was obtained using iget.
- Deadlock? next points to the same inode as ip when looking up "..". Locking next before releasing the lock on ip would result in a deadlock.
- namex unlocks the directory before obtaining a lock on next.

# Let's see Inode Layer

|                 |   |                                                                                                                         |
|-----------------|---|-------------------------------------------------------------------------------------------------------------------------|
| System Calls    |   | open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,                                          |
| File descriptor | → | <b>file.c</b> fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,                                   |
| Pathname        | → | <b>fs.c</b> namex, namei, nameiparent, skipellem                                                                        |
| Directory       | → | <b>fs.c</b> dirlookup, dirlink                                                                                          |
| Inode           | → | <b>fs.c</b> iinit, ialloc, iupdate, iguret, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap, |
| Logging         | → |                                                                                                                         |
| Buffer cache    | → | Block allocation on disk: <b>balloc, bfree</b>                                                                          |
| Disk            | → | <b>log.c</b> : begin_op, end_op, initlog, commit,<br><b>bio.c</b> binit, bget, bread, bwrite, brelse                    |
|                 |   | <b>ide.c</b> : idewait, ideinit, idestart, ideintr, ide rw                                                              |

# On disk & in memory inodes

```
struct {  
    struct spinlock lock;  
    struct inode inode[NINODE];  
} icache;
```

// On-disk inode structure

```
struct dinode {  
    short type;      // File type  
    short major;    // T_DEV Major device number  
    short minor;    // Minor device number  
    short nlink;    // Number of links  
    uint size;       // Size of file (bytes)  
    uint addrs[NDIRECT+1]; /  
};
```

```
// in-memory copy of an inode  
struct inode {  
    uint dev;        // Device number  
    uint inum;       // Inode number  
    int ref;         // Reference count  
    struct sleeplock lock; // protects everything  
                           // below here  
    int valid;       // been read from disk?  
  
    short type;      // copy of disk inode  
    short major;  
    short minor;  
    short nlink;  
    uint size;  
    uint addrs[NDIRECT+1];  
};
```

# In memory inodes

- Kernel keeps a subset of on disk inodes, those in use, in memory
  - as long as ‘ref’ is >0
- The **iget** and **iput** functions acquire and release pointers to an inode, modifying the **ref** count.
- See the caller graph of **iget()**
  - all those who call **iget()**
- Sleep lock in ‘inode’ protects
  - fields in inode
  - data blocks of inode

# iget and iupdate

## iget

■ searches for an existing/free inode in icache and returns pointer to one

■ if found, increments ref and returns pointer to inode

■ else gets empty inode , initializes, ref=1 and return

■ No lock held after iget()

■ Code must call ilock() after iget() to get lock

■ During lookup (later), many processes can iget() an inode, but only one holds the lock

## iupdate(inode \*ip)

■ read on disk block of inode

■ get on disk inode

■ modify it as specified in 'ip'

■ modify disk block of inode

■ log\_write(disk block of inode)

# **itrunc , iput**

```
②iput(ip)
③if ref is 1
④itrunc(ip)
⑤type = 0
⑥iupdate(ip)
⑦i->valid = 0 // free in
memory
⑧else
⑨ref--
```

```
②itrunc(ip)
③write all data blocks of
inode to disk
④using bfree()
⑤ip->size = 0
⑥Inode is freed from use
⑦iupdate(ip)
⑧called from iput() only
when 'ref' becomes zero
```

# race in iput ?

- A concurrent thread might be waiting in ilock to use this inode and won't be prepared to find the inode is not longer allocated
- This is not possible. Why?
- no way for a syscall to get a ref to a inode with ip->ref = 1

```
void  
iput(struct inode *ip)  
{  
    acquireSleep(&ip->lock);  
    if(ip->valid && ip->nlink == 0){  
        acquire(&icache.lock);  
        int r = ip->ref;  
        release(&icache.lock);  
        if(r == 1){  
            // inode has no links and no other references:  
            // truncate and free.  
            itrunc(ip);  
        }  
    }  
}
```

# buffer and inode cache

- to read an **inode**, its block must be read in a buffer
- So the buffer always contains a copy of the on-disk **dinode**
- duplicate copy in in-memory **inode**
- The inode cache is write-through, code that modifies a cached inode must immediately write it to disk with **iupdate**
- Inode may still exist in the buffer cache

# allocating inode

- `ialloc(dev, type)`
  - Loop over all disk inodes
  - read inode (from its block)
  - if it's free (note inum)
    - zero on disk inode
    - write on disk inode (as zeroes)
  - return `iget(dev, inum)`
  - panic if no free inodes
- 
- `ilock`
    - code must acquire ilock before using inode's data/fields
    - **Ilock reads inode if it's already not in memory**

# Trouble with `iput()` and crashes

- ❑ `iput()` doesn't truncate a file immediately when the link count for the file drops to zero, because
- ❑ some process might still hold a reference to the inode in memory: a process might still be reading and writing to the file, because it successfully opened it.

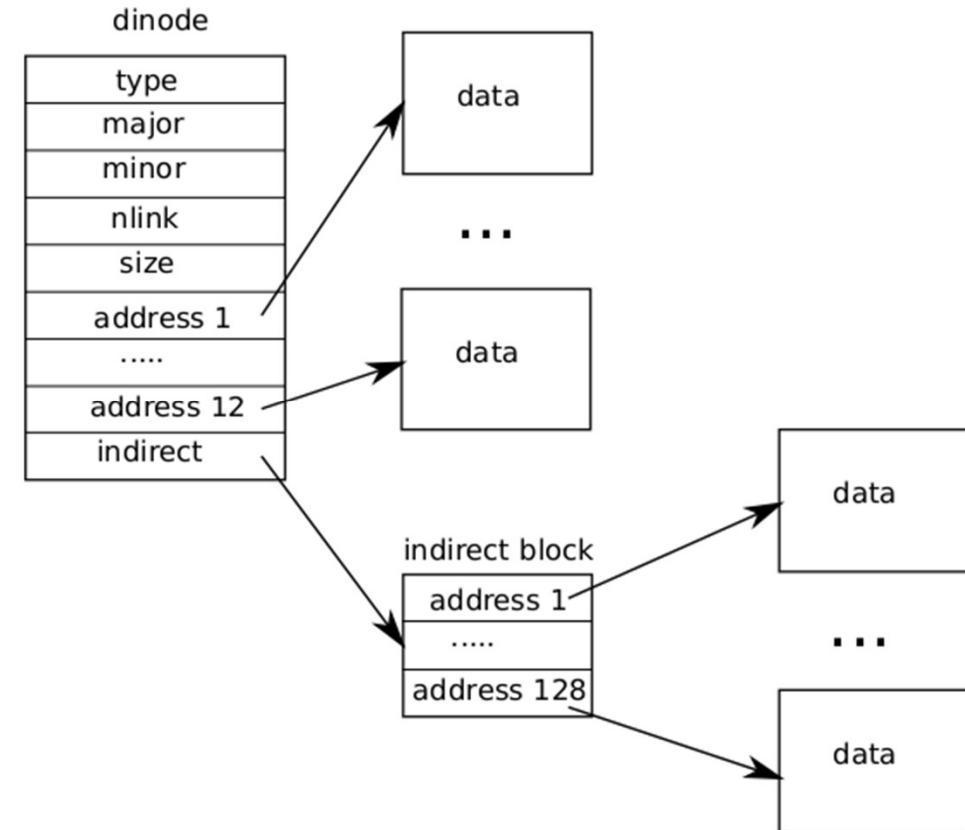
- ❑ if a crash happens before the last process closes the file descriptor for the file, then the file will be marked allocated on disk but no directory entry points to it

❑ Unsolved problem.

❑ How to solve it?

# Get Inode data: bmap(ip, bn)

- Allocate ‘bn’th block for the file given by inode ‘ip’
- Allocate block on disk and store it in either direct entries or block of indirect entries
- allocate block of indirect entries if needed using balloc()



# writing/reading data at a given offset in file

**readi(struct inode \*ip,  
char \*dst, uint off, uint  
n)**

**writei(struct inode \*ip,  
char \*src, uint off, uint  
n)**

- Calculate the block number in file where ‘off’ belongs
- Read sufficient blocks to read ‘n’ bytes
  - using bread(), brelse()
- Call devsw.read if inode is a device Inode.
- Writei() also updates size if required

# Let's see block allocation layer

|                 |                                                                                                                       |
|-----------------|-----------------------------------------------------------------------------------------------------------------------|
| System Calls    | open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,                                        |
| File descriptor | <b>file.c</b> fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,                                 |
| Pathname        | <b>fs.c</b> namex, namei, nameiparent, skipellem                                                                      |
| Directory       | <b>fs.c</b> dirlookup, dirlink                                                                                        |
| Inode           | <b>fs.c</b> iinit, ialloc, iupdate, igit, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap, |
| Logging         | <b>Block allocation on disk: balloc, bfree</b>                                                                        |
| Buffer cache    | <b>log.c</b> : begin_op, end_op, initlog, commit,<br><b>bio.c</b> binit, bget, bread, bwrite, brelse                  |
| Disk            | <b>ide.c</b> : idewait, ideinit, idestart, ideintr, iderw                                                             |

Normally, any upper layer can call any lower layer below

**Abhijit: Block allocator should be considered as another Layer!**

# allocating & deallocating blocks on DISK

## ■ **balloc(devno)**

■ looks for a block whose bitmap bit is zero, indicating that it is free.

■ On finding updates the bitmap and returns the block.

■ balloc() calls bread()->bget to get a block from disk in a buffer.

■ Race prevented by the fact that the buffer cache only lets one process use any one bitmap block at a time.

■ Calls log\_write(bp);

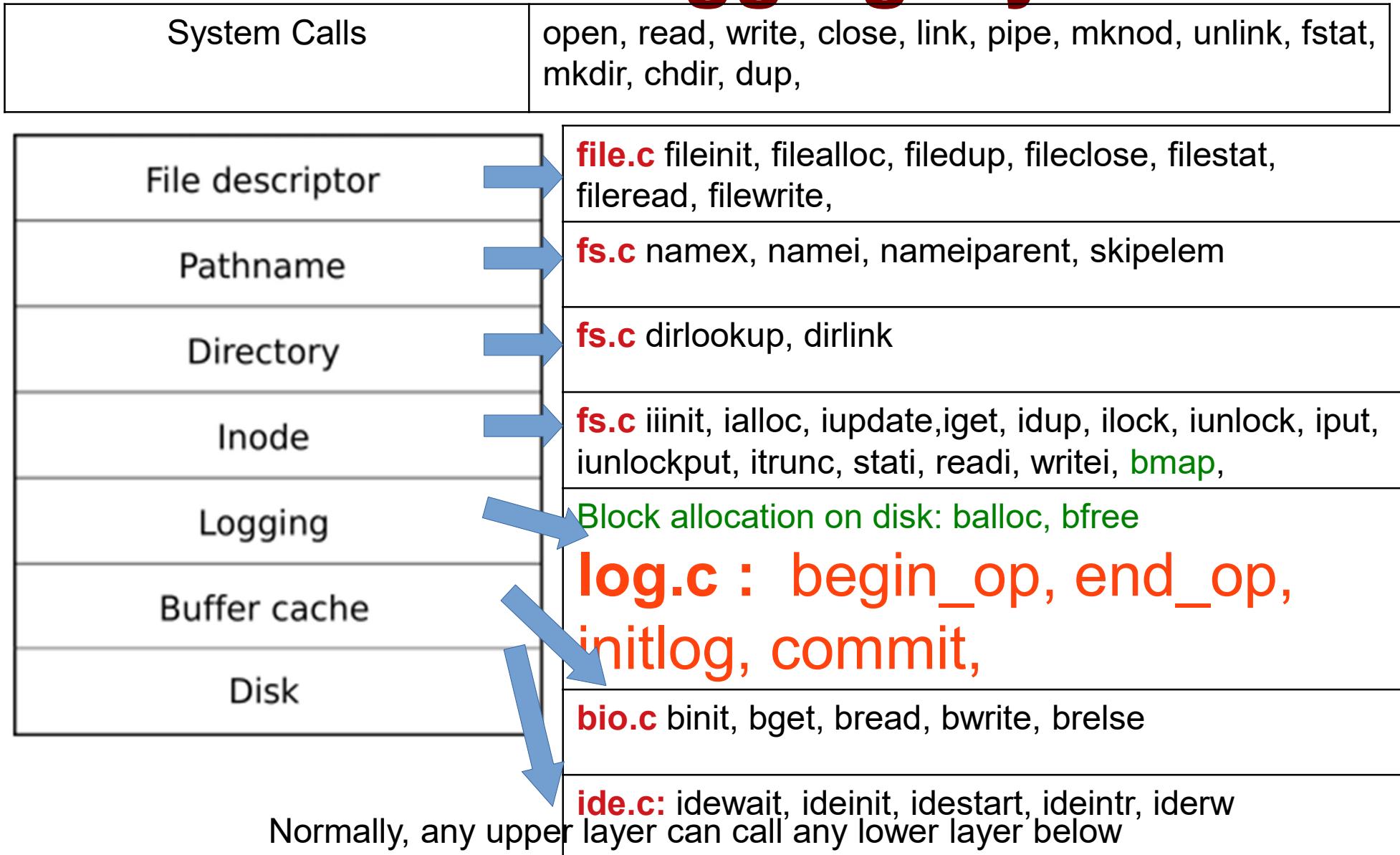
■ Thus writes to bitmap blocks are also logged

## ■ **bfree(devno, blockno)**

■ finds the right bitmap block and clears the right bit.

■ Also calls log\_write()

# Let's see logging layer



# Recovery

- Problem. Consider creating a file on ext2 file system.
- Following on disk data structures will/may get modified
  - Directory data block, new directory data block, block bitmap, inode table, inode table bitmap, group descriptor, super block, data blocks for new file, more data block bitmaps, ...
  - All cached in memory by OS
  - Delayed write – OS writes changes in its in-memory data structures, and schedules writes to disk when convenient
  - Possible that some of the above changes are written, but some are not
  - Inconsistent data structure! --> Example: inode table written, inode bitmap written, but directory data block not written

# Recovery

- **Consistency checking (e.g. fsck command) – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies**
- **Can be slow and sometimes fails**
- **Use system programs to back up data from disk to another storage device (magnetic tape, other magnetic disk, optical)**
- **Recover lost file or disk by restoring data from backup**

# Recovery

- Is a critical problem!
- Downtime is un-desired!
- A attempt at the solution: log structured / journaling file systems, e.g. ext3

# Log structured file systems

- Log structured (or journaling) file systems record each metadata update to the file system as a transaction
- All transactions are written to a log
- A transaction is considered committed once it is written to the log (sequentially)
- Sometimes to a separate device or section of disk
- However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system structures
- When the file system structures are modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- Faster recovery from crash, removes chance of inconsistency of metadata

# Journaling file systems

▪ Veritas FS

▪ Ext3, Ext4

▪ Xv6 file system!

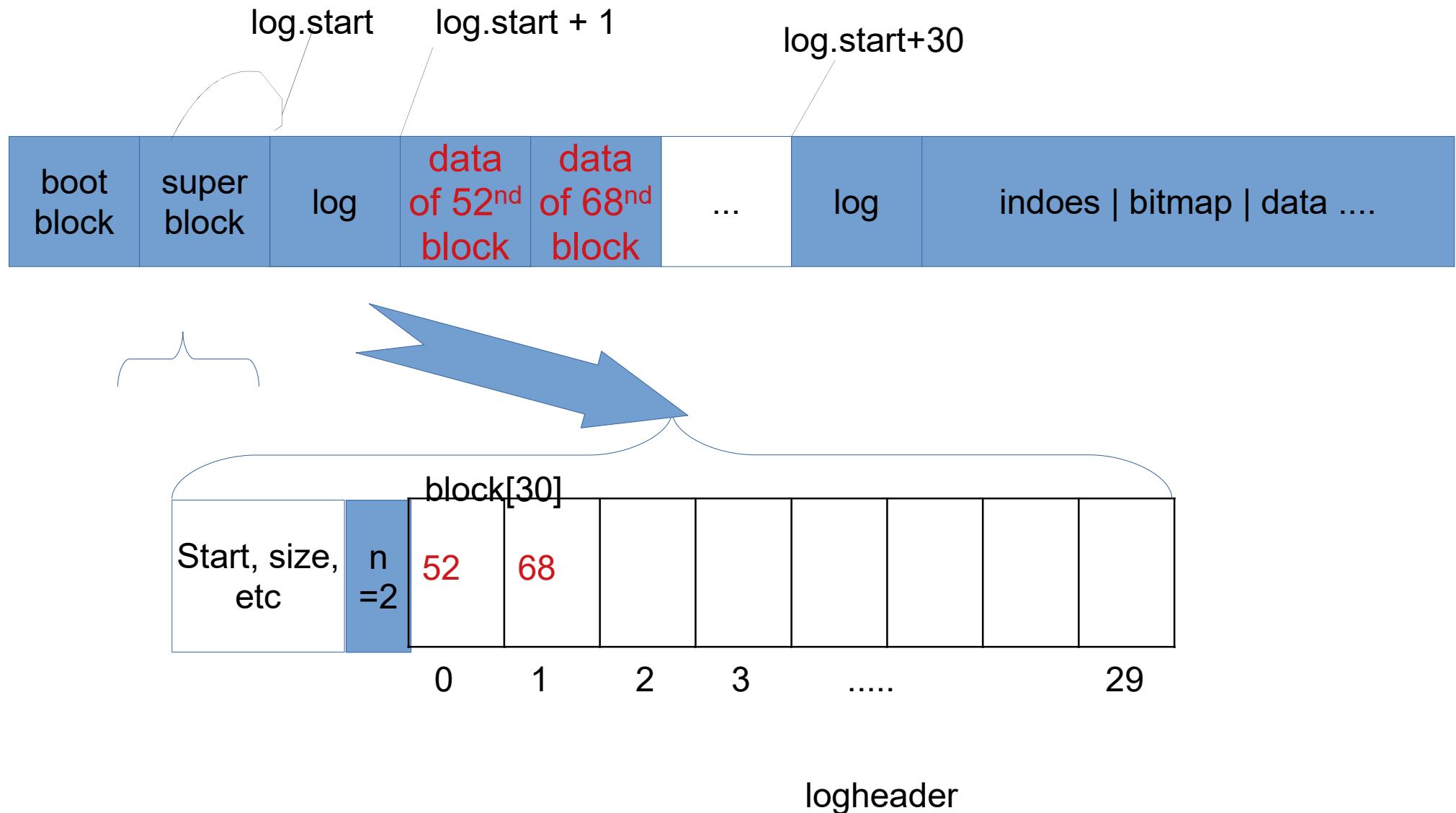
# log in xv6

- a mechanism of recovery from disk
- Concept: multiple write operations needed for system calls (e.g. ‘open’ system call to create a file in a directory)
- some writes succeed and some don’t
- leading to inconsistencies on disk
- In the log, all changes for a ‘transaction’ (an operation) are either written completely or not at all
- During recovery, completed operations can be “rerun” and incomplete operations neglected

# log in xv6

- xv6 system call does not directly write the on-disk file system data structures.
- A system call calls begin\_op() at begining and end\_op() at end
- begin\_op() increments log.outstanding
- end\_op() decrements log.outstanding, and if it's 0, then calls commit()
- During the code of system call, whenever a buffer is modified, (and done with)
  - log\_write() is called
  - This copies the block in an array of blocks inside log, the block is not written in it's actual place in FS as of now
  - when finally commit() is called, all modified blocks are copied to disk in the file system

# log on disk



# log

```
struct logheader { // ON DISK
    int n; // number of entries in use in block[] below
    int block[LOGSIZE]; // List of block numbers stored
};

struct log { // only in memory
    struct spinlock lock;
    int start; // first log block on disk (starts with logheader)
    int size; // total number of log blocks (in use out of 30)
    int outstanding; // how many FS sys calls are executing.
    int committing; // in commit(), please wait.
    int dev; // FS device
    struct logheader lh; // copy of the on disk logheader
};

struct log log;
```

# Typical use case of logging

*/\* In a system call code \*/*

**begin\_op();**

...

**bp = bread(...);**

**bp->data[...] = ...;**

**log\_write(bp);**

...

**end\_op();**

prepare for logging. Wait if logging system is not ready or ‘committing’. **++outstanding**

read and get access to a data block – as a buffer

modify buffer

note down this buffer for writing, in log. proxy for bwrite(). Mark B\_DIRTY. Absorb multiple writes into one.

**Syscall done. write log and all blocks. --outstanding.**

**If outstanding = 0, commit().**

# Example of calls to logging

```
//file_write() code  
begin_op();  
  
ilock(f->ip);  
  
/*loop */ r = writei(f->ip, ...);  
  
iunlock(f->ip);  
  
end_op();
```

- each writei() in turn calls bread(), log\_write() and brelse()
- also calls iupdate(ip) which also calls bread, log\_write and brelse
- Multiple writes are combined between begin\_op() and end\_op()

# Logging functions

## `initlog()`

- Set fields in global `log.xyz` variables, using FS superblock

- Recovery if needed

- Called from first `forkret()`

- Following three called by FS code

## `begin_op(void)`

- Increment `log.outstanding`

## `end_op(void)`

- Decrement `log.outstanding` and call `commit()` if it's zero

## `log_write(buf *)`

- Remember the specified block number in `log.lh.block[]` array

- Set the block to be dirty

## `write_log(void)`

- Called only from `commit()`

- Use block numbers specified in `log.lh.block` and copy those blocks from memory to log-blocks

## `commit(void)`

- Called only from `end_op()`

## `write_log()`

- Write header to disk log-header

- Copy from log blocks to actual FS blocks

- Reset and write log header again

# pipes

```
struct pipe {  
    struct spinlock lock;  
    char data[PIPE_SIZE];  
    uint nread;  
    // number of bytes read  
    uint nwrite;  
    // number of bytes written  
    int readopen;  
    // read fd is still open  
    int writeopen;  
    // write fd is still open  
};
```

functions  
pipealloc  
pipeclose  
piperead  
pipewrite

# pipes

- „pipealloc
- „allocate two struct file
- „allocate pipe itself using kalloc (it's a big structure with array)
- „init lock
- „initialize both struct file as 2 ends (r/w)
- „pipewrite
- „wait if pipe full
- „write to pipe
- „wakeup processes waiting to read
- „piperead
- „wait if no data
- „read from pipe
- „wakeup processes waiting to write
- „Good producer consumer code !

# **Further to reading system call code now**

- ❑ Now we are ready to read the code of system calls on file system**
- ❑ sys\_open, sys\_write, sys\_read , etc.**
- ❑ Advise: Before you read code of these, contemplate on what these functions should do using the functions we have studied so far.**
- ❑ Also think of locks that need to be held.**

# Synchronization

# My formulation

- OS = data structures + synchronization
- Synchronization problems make writing OS code challenging
- Demand exceptional coding skills

# Race problem

```
long c = 0, c1 = 0, c2 = 0, run = 1;  
  
void *thread1(void *arg) {  
    while(run == 1) {  
        c++;  
        c1++;  
    }  
}  
  
void *thread2(void *arg) {  
    while(run == 1) {  
        c++;  
        c2++;  
    }  
}
```

```
int main() {  
    pthread_t th1, th2;  
    pthread_create(&th1, NULL,  
    thread1, NULL);  
  
    pthread_create(&th2, NULL,  
    thread2, NULL);  
  
    //fprintf(stdout, "Ending main\n");  
  
    sleep(2);  
  
    run = 0;  
  
    fprintf(stdout, "c = %ld c1+c2 =  
    %ld c1 = %ld c2 = %ld \n", c, c1+c2,  
    c1, c2);  
  
    fflush(stdout);  
}
```

# Race problem

- On earlier slide
- Value of c should be equal to  $c_1 + c_2$ , but it is not!
- Why?
- There is a “race” between thread1 and thread2 for updating the variable c
- thread1 and thread2 may get scheduled in any order and *interrupted* any point in time
- The changes to c are not atomic!
- What does that mean?

# Race problem

- C++, when converted to assembly code, could be

**mov c, r1**

**add r1, 1**

**mov r1, c**

- Now following sequence of instructions is possible among thread1 and thread2

**thread1: mov c, r1**

**thread2: mov c, r1**

**thread1: add r1, 1**

**thread1: mov r1, c**

**thread2: add r1, 1**

**thread2: mov r1, c**

- What will be value in c, if initially c was, say 5?

- It will be 6, when it is expected to be 7. Other variations also possible.

# Races: reasons

- **Interruptible kernel**

- **If entry to kernel code does not disable interrupts, then modifications to any kernel data structure can be left incomplete**

- **This introduces concurrency**

- **Multiprocessor systems**

- **On SMP systems: memory is shared, kernel and process code run on all processors**

- **Same variable can be updated parallelly (not concurrently)**

- **What about non-interruptible kernel on multiprocessor systems?**

- **What about non-interruptible kernel on uniprocessor systems?**

# Critical Section problem

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (TRUE);
```

Figure 6.1 General structure of a typical process  $P_i$ .

# Critical Section Problem

- Consider system of n processes {p<sub>0</sub>, p<sub>1</sub>, ... p<sub>n-1</sub>}
- Each process has critical section segment of code
- Process may be changing common variables, updating table, writing file, etc
- When one process in critical section, no other may be in its critical section
- Critical section problem is to design protocol to solve this
- Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section
- Especially challenging with preemptive kernels

# Expected solution characteristics

## 1. Mutual Exclusion

- If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections

## 2. Progress

- If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

## 3. Bounded Waiting

- A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the  $n$  processes

# suggested solution - 1

```
int flag = 1;  
void *thread1(void *arg) {  
    while(run == 1) {  
        while(flag == 0)  
        ;  
        flag = 0;  
        c++;  
        flag = 1;  
        c1++;  
    }  
}
```

- What's wrong here?
- Assumes that  
**while(flag ==) ; flag = 0**
- will be atomic

# suggested solution - 2

```
int flag = 0;  
  
void *thread1(void *arg) {  
    while(run == 1) {  
        if(flag)  
            c++;  
        else  
            continue;  
        c1++;  
        flag = 0;  
    }  
}
```

```
void *thread2(void *arg) {  
    while(run == 1) {  
        if(!flag)  
            c++;  
        else  
            continue;  
        c2++;  
        flag = 1;  
    }  
}
```

# Peterson's solution

- Two process solution

- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted

- The two processes share two variables:

- int turn;

- Boolean flag[2]

- The variable turn indicates whose turn it is to enter the critical section

- The flag array is used to indicate if a process is ready to enter the critical section.  $\text{flag}[i] = \text{true}$  implies that process Pi is ready!

# Peterson's solution

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j)  
        ;  
    critical section  
    flag[i] = FALSE;  
    remainder section  
} while (TRUE);
```

- Provable that
  - Mutual exclusion is preserved
  - Progress requirement is satisfied
  - Bounded-waiting requirement is met

# Hardware solution – the one actually implemented

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
- Currently running code would execute without preemption
- Generally too inefficient on multiprocessor systems
- Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words
  - Basically two operations (read/write) done atomically in hardware

# Solution using test-and-set

```
lock = false; //global
```

```
do {  
    while ( TestAndSet (&lock ) )  
          ; // do nothing  
    // critical section  
    lock = FALSE;  
    // remainder section  
} while (TRUE);
```

Definition:

```
boolean TestAndSet (boolean  
*target)  
{  
    boolean rv = *target;  
    *target = TRUE;  
    return rv;  
}
```

# Solution using swap

```
lock = false; //global  
  
do {  
    key = true  
    while ( key == true))  
        swap(&lock, &key)  
        // critical section  
    lock = FALSE;  
    // remainder section  
} while (TRUE);
```

# Spinlock

- A lock implemented to do ‘busy-wait’

- Using instructions like T&S or Swap

- As shown on earlier slides

```
▪ spinlock(int *lock){  
    While(test-and-set(lock))
```

```
    ;
```

```
}
```

```
▪ spinunlock(lock *lock) {  
    *lock = false;  
}
```

# Bounded wait M.E. with T&S

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;
        // critical section
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;
        // remainder section
} while (TRUE);
```

# **sleep-locks**

- Spin locks result in busy-wait
- CPU cycles wasted by waiting processes/threads
- Solution – threads keep waiting for the lock to be available
- Move thread to wait queue
- The thread holding the lock will wake up one of them

# Sleep locks/mutexes

```
//ignore syntactical issues

typedef struct mutex {
    int islocked;
    int spinlock;
    waitqueue q;
}mutex;

wait(mutex *m) {
    spinlock(m->spinlock);
    while(m->islocked)
        Block(m, m->spinlock)
    lk->islocked = 1;
    spinunlock(m->spinlock);
}

Block(mutex *m, spinlock *sl) {
    currprocess->state = WAITING
    move current process to m->q
    spinunlock(sl);
    Sched();
    spinlock(sl);
}

release(mutex *m) {
    spinlock(m->spinlock);
    m->islocked = 0;
    Some process in m->queue =RUNNABLE;
    spinunlock(m->spinlock);
}
```

# Some thumb-rules of spinlocks

- Never block a process holding a spinlock !

- Typical code:

```
while(condition)
```

```
{ Spin-unlock()
```

```
Schedule()
```

```
Spin-lock()
```

```
}
```

- Hold a spin lock for only a short duration of time

- Spinlocks are preferable on multiprocessor systems

- Cost of context switch is a concern in case of sleep-wait locks

- Short = < 2 context switches

# Locks in xv6 code

# struct spinlock

// Mutual exclusion lock.

```
struct spinlock {
```

```
    uint locked;      // Is the lock held?
```

// For debugging:

```
    char *name;      // Name of lock.
```

```
    struct cpu *cpu; // The cpu holding the lock.
```

```
    uint pcs[10];    // The call stack (an array of program counters)
```

```
                    // that locked the lock.
```

```
};
```

# spinlocks in xv6 code

```
struct {  
    struct spinlock lock;  
    struct buf buf[NBUF];  
    struct buf head;  
} bcache;  
  
struct {  
    struct spinlock lock;  
    struct file file[NFILE];  
} ftable;  
  
struct {  
    struct spinlock lock;  
    struct inode inode[NINODE];  
} icache;  
  
struct sleeplock {  
    uint locked;      // Is the lock held?  
    struct spinlock lk;
```

```
static struct spinlock idelock;  
  
struct {  
    struct spinlock lock;  
    int use_lock;  
    struct run *freelist;  
} kmem;  
  
struct log {  
    struct spinlock lock;  
}  
...}  
  
struct pipe {  
    struct spinlock lock;  
}  
...}  
  
struct {  
    struct spinlock lock;  
    struct proc proc[NPROC];  
} ptable;  
  
struct spinlock tickslock;
```

```

static inline uint
xchg(volatile uint *addr, uint newval)
{
    uint result;
    // The + in "+m" denotes a read-modify-write
    // operand.

    asm volatile("lock; xchgl %0, %1" :
        "+m" (*addr), "=a" (result) :
        "1" (newval) :
        "cc");
    return result;
}

struct spinlock {
    uint locked;      // Is the lock held?

    // For debugging:
    char *name;       // Name of lock.
    struct cpu *cpu;  // The cpu holding the lock.
    uint pcs[10];     // The call stack (an array of program
                      // counters) that locked the lock.
};

```

# Spinlock in xv6

```

void acquire(struct spinlock *lk)
{
    pushcli(); // disable interrupts to
    // avoid deadlock.

    // The xchg is atomic.

    while(xchg(&lk->locked, 1) != 0)
        ;
    //extra debugging code
}

void release(struct spinlock *lk)
{
    //extra debugging code

    asm volatile("movl $0, %0" : "+m"
               (lk->locked) : );
    popcli();
}

```

```
Void acquire(struct spinlock *lk)
{
    pushcli(); // disable interrupts to avoid deadlock.

    if(holding(lk))
        panic("acquire");

    .....

void pushcli(void)
{
    int eflags;

    eflags = readeflags();

    cli();

    if(mycpu()->ncli == 0)
        mycpu()->intena = eflags & FL_IF;
    mycpu()->ncli += 1;
}

static inline uint
readeflags(void)
{
    uint eflags;

    asm volatile("pushfl; popl %0" : "=r" (eflags));

    return eflags;
}
```

# spinlocks

- Pushcli() - disable interrupts on that processor

- One after another many acquire() can be called on different spinlocks

- Keep a count of them in mycpu()->ncli

```
void  
release(struct spinlock *lk)  
{  
...  
    asm volatile("movl $0, %0" : "+m" (lk->locked) : );  
    popcli();  
}  
. . .  
Void popcli(void)  
{  
    if(readeflags()&FL_IF)  
        panic("popcli - interruptible");  
    if(--mycpu()->ncli < 0)  
        panic("popcli");  
    if(mycpu()->ncli == 0 && mycpu()->intena)  
        sti();  
}
```

# spinlocks

## Popcli()

Restore interrupts if last popcli() call restores ncli to 0 & interrupts were enabled before pushcli() was called

# spinlocks

- Always disable interrupts while acquiring spinlock
- Suppose **iderw** held the **idelock** and then got interrupted to run **ideintr**.
  - **Ideintr** would try to lock **idelock**, see it was held, and wait for it to be released.
  - In this situation, **idelock** will never be released
- Deadlock
  - General OS rule: if a spin-lock is used by an interrupt handler, a processor must never hold that lock with interrupts enabled
  - Xv6 rule: when a processor enters a spin-lock critical section, xv6 always ensures interrupts are disabled on that processor.

# sleeplocks

- Sleeplocks don't spin. They move a process to a wait-queue if the lock can't be acquired
- XV6 approach to “wait-queues”
- Any memory address serves as a “wait channel”
  - The sleep() and wakeup() functions just use that address as a ‘condition’
  - There are no per condition process queues! Just one global queue of processes used for scheduling, sleep, wakeup etc. --> Linear search everytime !
  - costly, but simple

```
void  
sleep(void *chan, struct spinlock *lk)  
{  
    struct proc *p = myproc();  
    ....  
    if(lk != &ptable.lock){  
        acquire(&ptable.lock);  
        release(lk);  
    }  
    p->chan = chan;  
    p->state = SLEEPING;  
    sched();  
    // Reacquire original lock.  
    if(lk != &ptable.lock){  
        release(&ptable.lock);  
        acquire(lk);  
    }  
}
```

# sleep()

- At call must hold lock on the resource on which you are going to sleep
- since you are going to change p-> values & call sched(), hold ptable.lock if not held
- p->chan = given address remembers on which condition the process is waiting
- call to sched() blocks the process

# Calls to sleep() : examples of “chan” (output from cscope)

0 console.c

consoleread 251

sleep(&input.r, &cons.lock);

2 ide.c iderw

169 sleep(b, &idelock);

3 log.c begin\_op

131 sleep(&log, &log.lock);

6 pipe.c piperead

111 sleep(&p->nread, &p->lock);

7 proc.c wait

317 sleep(curproc,  
&ptable.lock);

8 sleeplock.c

acquiresleep 28  
sleep(lk, &lk->lk);

9 sysproc.c

sys\_sleep 74  
sleep(&ticks,  
&tickslock);

```
void wakeup(void *chan)
{
    acquire(&ptable.lock);
    wakeup1(chan);
    release(&ptable.lock);
}

static void wakeup1(void *chan)
{
    struct proc *p;

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

## Wakeup()

- Acquire ptable.lock since you are going to change ptable and p->values
- just linear search in process table for a process where p->chan is given address
- Make it runnable

# sleeplock

```
// Long-term locks for processes
struct sleeplock {
    uint locked;      // Is the lock held?
    struct spinlock lk; // spinlock protecting this sleep lock

    // For debugging:
    char *name;      // Name of lock.
    int pid;         // Process holding lock
};
```

# Sleeplock acquire and release

```
void
acquiresleep(struct sleeplock *lk)
{
    acquire(&lk->lk);
    while (lk->locked) {
        /* Abhijit: interrupts are not disabled in sleep
       */
        sleep(lk, &lk->lk);
    }
    lk->locked = 1;
    lk->pid = myproc()->pid;
    release(&lk->lk);
}
```

```
void
releasesleep(struct sleeplock *lk)
{
    acquire(&lk->lk);
    lk->locked = 0;
    lk->pid = 0;
    wakeup(lk);
    release(&lk->lk);
}
```

# Where are sleeplocks used?

- ❑ **struct buf**
    - ❑ waiting for I/O on this buffer
  - ❑ **struct inode**
    - ❑ waiting for I/o to this inode

# Sleeplocks issues

- sleep-locks support yielding the processor during their critical sections.
- This property poses a design challenge:
  - if thread T1 holds lock L1 and has yielded the processor (waiting for some other condition),
  - and thread T2 wishes to acquire L1,
  - we have to ensure that T1 can execute
  - while T2 is waiting so that T1 can release L1.
- T2 can't use the spin-lock acquire function here: it spins with interrupts turned off, and that would prevent T1 from running.
- To avoid this deadlock, the sleep-lock acquire routine (called `acquiresleep`) yields the processor while waiting, and does not disable interrupts.
- Sleep-locks leave interrupts enabled, they cannot be used in interrupt handlers.

# More needs of synchronization

- ❑ Not only critical section problems
- ❑ Run processes in a particular order
- ❑ Allow multiple processes read access, but only one process write access
- ❑ Etc.



# 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++;

- }

- --> Note this is Signal() on a semaphore, different froms signal system call

# Semaphore for synchronization

- **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 mutex; // initialized to 1
  ▪ do {
    ▪ wait (mutex);
      ▪ // Critical Section
    ▪ signal (mutex);
  ▪ // remainder section
  ▪ } while (TRUE)
```

# Semaphore implementation

```
Wait(sem *s) {  
    while(s <=0)  
        block(); // could be ";"  
    s--;  
}  
  
signal(sem *s) {  
    s++;  
}
```

- Left side – expected behaviour
- Both the wait and signal should be atomic.
- This is the semantics of the semaphore.

# Semaphore implementation? - 1

```
struct semaphore {  
    int val;  
    spinlock lk;  
};  
  
sem_init(semaphore *s, int initval) {  
    s->val = initval;  
    s->sl = 0;  
}  
  
wait(semaphore *s) {  
    spinlock(&(s->sl));  
    while(s->val <=0)  
    ;  
    (s->val)--;  
    spinunlock(&(s->sl));  
}
```

```
signal(semaphore *s) {  
    spinlock(*s->sl));  
    (s->val)++;  
    spinunlock(*s->sl));  
}
```

- suppose 2 processes trying wait.

val = 1;

Th1: spinlock                  Th2: spinlock-waits  
Th1: while -> false, val-- => 0; spinunlock;  
Th2: spinlock success; while() -> true, loops;  
Th1: is done with critical section, it calls signal. it calls spinlock() -> wait.

Who is holding spinlock -> Th2. It is waiting for val > 0. Who can set value > 0 , ans: Th1, and Th1 is waiting for spinlock which is held by Th2.

circular wait. Deadlock.

None of them will proceed.

# Semaphore implementation? - 2

```
struct semaphore {  
    int val;  
    spinlock lk;  
};  
  
sem_init(semaphore *s, int initval) {  
    s->val = initval;  
    s->sl = 0;  
}  
  
signal(semaphore *s) {  
    spinlock(*(s->sl));  
    (s->val)++;  
    spinunlock(*(s->sl));  
}
```

```
wait(semaphore *s) {  
    spinlock(&(s->sl));  
    while(s->val <=0) {  
        spinunlock(&(s->sl));  
        spinlock(&(s->sl));  
    }  
    (s->val)--;  
    spinunlock(&(s->sl));  
}
```

**Problem:** race in **spinlock** of **while** loop and signal's **spinlock**. Bounded wait not guaranteed.

**Spinlocks are not good for a long wait.**

# Semaphore implementation? - 3, idea

```
struct semaphore {  
    int val;  
    spinlock lk;  
};  
sem_init(semaphore *s, int initval) {  
    s->val = initval;  
    s->sl = 0;  
}  
block() {  
    put this current process on wait-q;  
    schedule();  
}  
  
wait(semaphore *s) {  
    spinlock(&(s->sl));  
    while(s->val <=0) {  
        Block();  
    }  
    (s->val)--;  
    spinunlock(&(s->sl));  
}  
signal(semaphore *s) {  
    spinlock(*(s->sl));  
    (s->val)++;  
    spinunlock(*(s->sl));  
}
```

# Semaphore implementation? - 3a

```
struct semaphore {  
    int val;  
    spinlock lk;  
    list l;  
};  
  
sem_init(semaphore *s, int initval) {  
    s->val = initval;  
    s->sl = 0;  
}  
  
block(semaphore *s) {  
    listappend(s->l, current);  
    schedule();  
}  
  
problem is that block() will be called without holding  
the spinlock and the access to the list is not protected.  
  
Note that - so far we have ignored changes to signal()
```

```
wait(semaphore *s) {  
    spinlock(&(s->sl));  
    while(s->val <=0) {  
        spinunlock(&(s->sl));  
        block(s);  
    }  
    (s->val)--;  
    spinunlock(&(s->sl));  
}  
  
signal(semaphore *s) {  
    spinlock(*(s->sl));  
    (s->val)++;  
    spinunlock(*(s->sl));  
}
```

# Semaphore implementation? - 3b

```
struct semaphore {  
    int val;  
    spinlock lk;  
    list l;  
};  
  
sem_init(semaphore *s, int initval) {  
    s->val = initval;  
    s->sl = 0;  
}  
  
block(semaphore *s) {  
    listappend(s->l, current);  
    spinunlock(&(s->sl));  
    schedule();  
}  
  
wait(semaphore *s) {  
    spinlock(&(s->sl));  
    while(s->val <=0) {  
        block(s);  
    }  
    (s->val)--;  
    spinunlock(&(s->sl));  
}  
  
signal(semaphore *s) {  
    spinlock(*(s->sl));  
    (s->val)++;  
    x = dequeue(s->sl) and enqueue(readyq, x);  
    spinunlock(*(s->sl));  
}  
  
Problem: after a blocked process comes out of the block,  
it does not hold the spinlock and it's going to change  
the s->sl;
```

# Semaphore implementation? - 3c

```
struct semaphore {  
    int val;  
    spinlock lk;  
    list l;  
};  
  
sem_init(semaphore *s, int initval) {  
    s->val = initval;  
    s->sl = 0;  
}  
  
block(semaphore *s) {  
    listappend(s->l, current);  
    spinunlock(&(s->sl));  
    schedule();  
}  
  
wait(semaphore *s) {  
    spinlock(&(s->sl)); // A  
    while(s->val <=0) {  
        block(s);  
        spinlock(&(s->sl)); // B  
    }  
    (s->val)--;  
    spinunlock(&(s->sl));  
}  
  
signal(semaphore *s) {  
    spinlock(*(s->sl));  
    (s->val)++;  
    x = dequeue(s->sl) and enqueue(readyq, x);  
    spinunlock(*(s->sl));  
}  
  
Question: there is race between A and B. Can we  
guarantee bounded wait ?
```

# 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:
  - 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

```
wait(semaphore *S) {  
    S->value--;  
    if (S->value < 0) {  
        add this process  
        to S->list;  
        block();  
    }  
}  
  
signal(semaphore *S) {  
    S->value++;  
    if (S->value <= 0) {  
        remove a process P  
        from S->list;  
        wakeup(P);  
    }  
}
```

# Semaphore in Linux

```
struct semaphore {  
    raw_spinlock_t    lock;  
    unsigned int      count;  
    struct list_head  wait_list;  
};  
  
static inline void __sched  
__down(struct semaphore *sem)  
{  
    __down_common(sem,  
    TASK_UNINTERRUPTIBLE,  
    MAX_SCHEDULE_TIMEOUT);  
}  
  
void down(struct semaphore *sem)  
{  
    unsigned long flags;  
  
    raw_spin_lock_irqsave(&sem->lock, flags);  
    if (likely(sem->count > 0))  
        sem->count--;  
    else  
        __down(sem);  
    raw_spin_unlock_irqrestore(&sem->lock,  
    flags);  
}
```

# Semaphore in Linux

```
static inline int __sched
__down_common(struct semaphore
*sem, long state, long timeout)
{
    struct task_struct *task = current;
    struct semaphore_waiter waiter;
    list_add_tail(&waiter.list, &sem->wait_list);
    waiter.task = task;
    waiter.up = false;
```

```
for (;;) {
    if (signal_pending_state(state, task))
        goto interrupted;
    if (unlikely(timeout <= 0))
        goto timed_out;
    __set_task_state(task, state);
    raw_spin_unlock_irq(&sem->lock);
    timeout = schedule_timeout(timeout);
    raw_spin_lock_irq(&sem->lock);
    if (waiter.up)
        return 0;
}
....
```

# **Different uses of semaphores**

# For mutual exclusion

**/\*During initialization\*/**

**semaphore sem;**

**initsem (&sem, 1);**

**/\* On each use\*/**

**P (&sem);**

**Use resource;**

**V (&sem);**

# Event-wait

```
/* During initialization */

semaphore event;

initsem (&event, 0); /* probably at boot time */

/* Code executed by thread that must wait on event */

P (&event); /* Blocks if event has not occurred */

/* Event has occurred */

V (&event); /* So that another thread may wake up */

/* Continue processing */

/* Code executed by another thread when event occurs */

V (&event); /* Wake up one thread */
```

# Control countable resources

```
/* During initialization */  
semaphore counter;  
initsem (&counter, resourceCount);  
/* Code executed to use the resource */  
P (&counter); /* Blocks until resource is available */  
Use resource; /* Guaranteed to be available now */  
V (&counter); /* Release the resource */
```

# Drawbacks of semaphores

- ❑ Need to be implemented using lower level primitives like spinlocks
- ❑ Context-switch is involved in blocking and signaling – time consuming
- ❑ Can not be used for a short critical section

# **Deadlocks**

# 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

|   | P0                 | P1                 |
|---|--------------------|--------------------|
| ▪ | <b>wait (S);</b>   | <b>wait (Q);</b>   |
| ▪ | <b>wait (Q);</b>   | <b>wait (S);</b>   |
| ▪ | .                  | .                  |
| ▪ | .                  | .                  |
| ▪ | .                  | .                  |
| ▪ | <b>signal (S);</b> | <b>signal (Q);</b> |
| ▪ | <b>signal (Q);</b> | <b>signal (S);</b> |

# Example of deadlock

- Let's see the pthreads program : `deadlock.c`
- Same programme as on earlier slide, but with `pthread_mutex_lock();`

# Non-deadlock, but similar situations

## ■ Starvation – indefinite blocking

■ A process may never be removed from the semaphore queue in which it is suspended

## ■ Priority Inversion

■ Scheduling problem when lower-priority process holds a lock needed by higher-priority process (so it can not pre-empt lower priority process), and a medium priority process (that does not need the lock) pre-empts lower priority task, denying turn to higher priority task

■ Solved via priority-inheritance protocol : temporarily enhance priority of lower priority task to highest

# Livelock

- Similar to deadlock, but processes keep doing ‘useless work’
  - E.g. two people meet in a corridor opposite each other
    - Both move to left at same time
    - Then both move to right at same time
    - Keep Repeating!
  - No process able to progress, but each doing ‘some work’ (not sleeping/waiting), state keeps changing

# Livelock example

```
#include <stdio.h>                                /* thread two runs in this function */

#include <pthread.h>

struct person {

    int otherid;
    int otherHungry;
    int myid;
};

int main() {

    pthread_t th1, th2;

    struct person one, two;

    one.otherid = 2; one.myid = 1;

    two.otherid = 1; two.myid = 2;

    one.otherHungry = two.otherHungry = 1;

    pthread_create(&th1, NULL, eat, &one);

    pthread_create(&th2, NULL, eat, &two);

    printf("Main: Waiting for threads to get over\n");

    pthread_join(th1, NULL);

    pthread_join(th2, NULL);

    return 0;
}

/* thread two runs in this function */

int spoonWith = 1;

void *eat(void *param)

{

    int eaten = 0;

    struct person person= *(struct person *)param;

    while (!eaten) {

        if(spoonWith == person.myid)

            printf("%d going to eat\n", person.myid);

        else

            continue;

        if(person.otherHungry) {

            printf("You eat %d\n", person.otherid);

            spoonWith = person.otherid;

            continue;

        }

        printf("%d is eating\n", person.myid);

        break;

    }

}
```

# More on deadlocks

- ❑ Under which conditions they can occur?
- ❑ How can deadlocks be avoided/prevented?
- ❑ How can a system recover if there is a deadlock ?

# System model for understanding deadlocks

- System consists of resources
  - Resource types  $R_1, R_2, \dots, R_m$
  - CPU cycles, memory space, I/O devices
- Resource: Most typically a lock, synchronization primitive
- Each resource type  $R_i$  has  $W_i$  instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release

# Deadlock characterisation

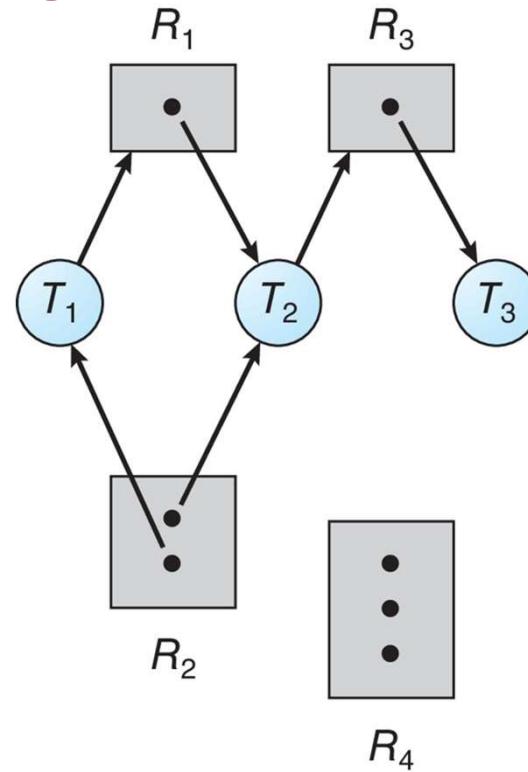
- Deadlock is possible only if ALL of these conditions are TRUE at the same time
  - Mutual exclusion: only one process at a time can use a resource
  - Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes
  - No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task
  - Circular wait: there exists a set  $\{P_0, P_1, \dots, P_n\}$  of waiting processes such that  $P_0$  is waiting for a resource that is held by  $P_1$ ,  $P_1$  is waiting for a resource that is held by  $P_2$ , ...,  $P_{n-1}$  is waiting for a resource that is held by  $P_n$ , and  $P_n$  is waiting for a resource that is held by  $P_0$ .

# Resource-Allocation Graph

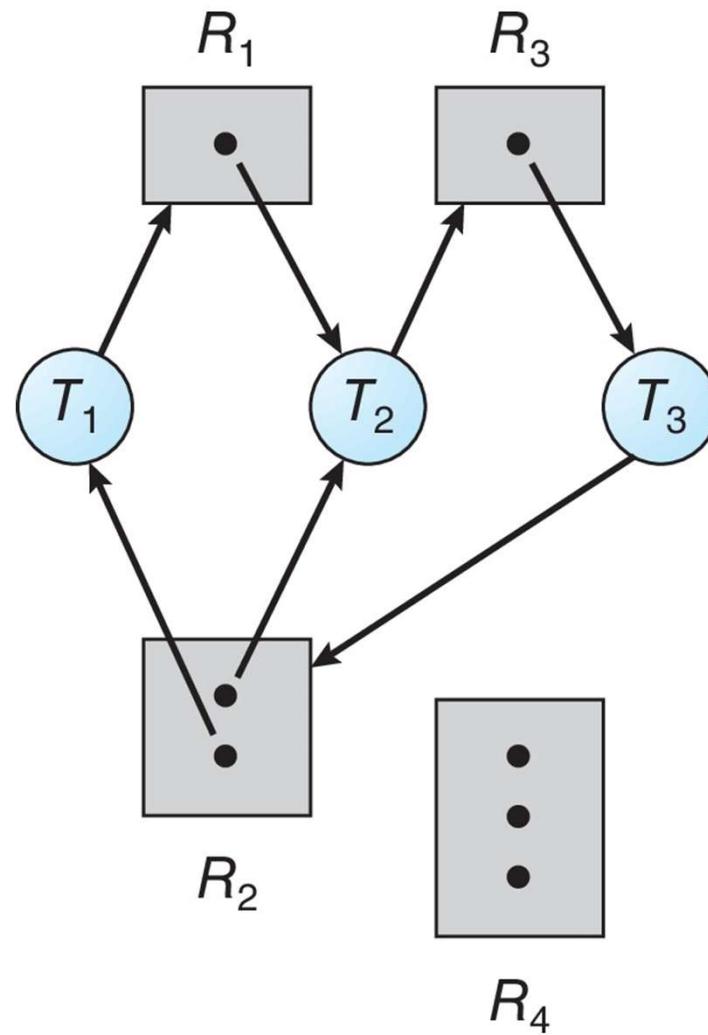
- A set of vertices  $V$  and a set of edges  $E$
- $V$  is partitioned into two types:
  - $P = \{P_1, P_2, \dots, P_n\}$ , the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \dots, R_m\}$ , the set consisting of all resource types in the system
- request edge – directed edge  $P_i \rightarrow R_j$
- assignment edge – directed edge  $R_j \rightarrow P_i$

# Resource Allocation Graph Example

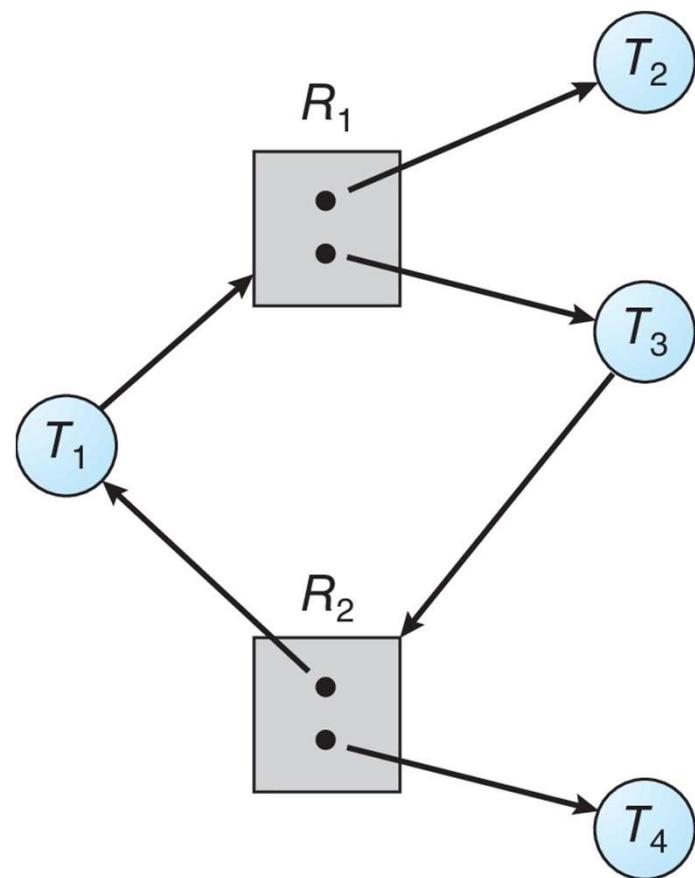
- One instance of R1
- Two instances of R2
- One instance of R3
- Three instances of R4
- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
- T3 is holding one instance of R3



# Resource Allocation Graph with a Deadlock



# Graph with a Cycle But no Deadlock



# Basic Facts

- ② If graph contains no cycles -> no deadlock
- ② If graph contains a cycle :
  - ② if only one instance per resource type, then deadlock
  - ② if several instances per resource type, possibility of deadlock

# Methods for Handling Deadlocks

- ❑ Ensure that the system will never enter a deadlock state:
  - 1) Deadlock prevention
  - 2) Deadlock avoidance
- ❑ 3) Allow the system to enter a deadlock state and then recover
- ❑ 4) Ignore the problem and pretend that deadlocks never occur in the system.

# (1) Deadlock Prevention

- Invalidate one of the four necessary conditions for deadlock:
  - Mutual Exclusion – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
  - Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
  - Low resource utilization; starvation possible

# (1) Deadlock Prevention (Cont.)

## ❑ No Preemption:

❑ If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released

❑ Preempted resources are added to the list of resources for which the process is waiting

❑ Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

## ❑ Circular Wait:

❑ Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

# (1) Deadlock prevention: Circular Wait

- Invalidating the circular wait condition is most common.
- Simply assign each resource (i.e., mutex locks) a unique number.
- Resources must be acquired in order.
- If:
  - first\_mutex = 1
  - second\_mutex = 5
- code for thread\_two could not be written as follows:

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);

    pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);

    pthread_exit(0);
}
```

# (1) Preventing deadlock: cyclic wait

- Locking hierarchy : Highly preferred technique in kernels
- Decide an ordering among all ‘locks’
- Ensure that on ALL code paths in the kernel, the locks are obtained in the decided order!
- Poses coding challenges!
- A key differentiating factor in kernels
- Do not look at only the current lock being taken, look at all the locks the code may be holding at any given point in code!

# (1) Prevention in Xv6: Lock Ordering

lock on the directory, a lock on the new file's inode, a lock on a disk block buffer, idelock, and ptable.lock.

## (2) Deadlock avoidance

- Requires that the system has some additional a priori information available
- Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes

## (2) Deadlock avoidance

>Please see: concept of safe states, unsafe states, Banker's algorithm

# (3) Deadlock detection and recovery

▪ How to detect a deadlock in the system?

▪ The Resource-Allocation Graph is a graph. Need an algorithm to detect cycle in a graph.

▪ How to recover?

▪ Abort all processes or abort one by one?

▪ Which processes to abort?

▪ Priority ?

▪ Time spent since forked()?

▪ Resources used?

▪ Resources needed?

▪ Interactive or not?

▪ How many need to be terminated?

# **“Condition” Synchronization Tool**

# What is condition variable?

- A variable with a sleep queue
- Threads can sleep on it, and wake-up all remaining

Struct condition {

    Proc \*next

    Proc \*prev

    Spinlock \*lock

}

Different variables of this type can be used as different 'conditions'

# Code for condition variables

```
//Spinlock s is held before calling wait

void wait (condition *c, spinlock_t *s)
{
    spin_lock (&c->listLock);
    add self to the linked list;
    spin_unlock (&c->listLock);
    spin_unlock (s); /* release
spinlock before blocking */
    swtch(); /* perform context switch */

    /* When we return from swtch, the event
has occurred */

    spin_lock (s); /* acquire the spin lock again
*/
    return;
}
```

```
void do_signal (condition *c)

/*Wakeup one thread waiting on the condition*/

{
    spin_lock (&c->listLock);

    remove one thread from linked list, if it is nonempty;

    spin_unlock (&c->listLock);

    if a thread was removed from the list, make it
        runnable;

    return;
}

void do_broadcast (condition *c)

/*Wakeup al lthreads waiting on the condition*/

{
    spin_lock (&c->listLock);

    while (linked list is nonempty) {

        remove a thread from linked list;

        make it runnable;
    }

    spin_unlock (&c->listLock);
}
```

# Semaphore implementation using condition variables?

❑ Is this possible?

❑ Can we try it?

```
typedef struct semaphore {
```

```
    //something
```

```
    condition c;
```

```
}semaphore;
```

❑ Now write code for semaphore P() and V()

# **Classical Synchronization Problems**

# Bounded-Buffer Problem

- Producer and consumer processes
- N buffers, each can hold one item
- Producer produces ‘items’ to be consumed by consumer , in the bounded buffer
- Consumer should wait if there are no items
- Producer should wait if the ‘bounded buffer’ is full

# **Bounded-Buffer Problem: solution with semaphores**

- ❑ **Semaphore mutex initialized to the value 1**
- ❑ **Semaphore full initialized to the value 0**
- ❑ **Semaphore empty initialized to the value N**

# Bounded-buffer problem

The structure of the producer process

```
do {  
    // produce an item in nextp  
    wait (empty);  
    wait (mutex);  
    // add the item to the buffer  
    signal (mutex);  
    signal (full);  
} while (TRUE);
```

The structure of the Consumer process

```
do {  
    wait (full);  
    wait (mutex);  
    // remove an item from  
    // buffer to nextc  
    signal (mutex);  
    signal (empty);  
    // consume item in nextc  
} while (TRUE);
```

# Bounded buffer problem

- ❑ Example : pipe()
- ❑ Let's see code of pipe in xv6 – a solution using sleeplocks

# Readers-Writers problem

- A data set is shared among a number of concurrent processes
- Readers – only read the data set; they do not perform any updates
- Writers – can both read and write
- Problem – allow multiple readers to read at the same time
- Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are treated – all involve priorities
- Shared Data
- Data set
  - Semaphore mutex initialized to 1
  - Semaphore wrt initialized to 1
  - Integer readcount initialized to 0

## The structure of a writer process

```
do {  
    wait (wrt) ;  
    // writing is performed  
    signal (wrt) ;  
} while (TRUE);
```

# Readers-Writers problem

## The structure of a reader process

```
do {  
    wait (mutex) ;  
    readcount ++ ;  
    if (readcount == 1)  
        wait (wrt) ;  
    signal (mutex)  
    // reading is performed  
    wait (mutex) ;  
    readcount -- ;  
    if (readcount == 0)  
        signal (wrt) ;  
    signal (mutex) ;  
} while (TRUE);
```

# Readers-Writers Problem

## Variations

- ❑ First variation – no reader kept waiting unless writer has permission to use shared object
- ❑ Second variation – once writer is ready, it performs write asap
- ❑ Both may have starvation leading to even more variations
- ❑ Problem is solved on some systems by kernel providing reader-writer locks

# Reader-write lock

- A lock with following operations on it
  - Lockshared()
  - Unlockshared()
  - LockExcl()
  - UnlockExcl()
- Possible additions
  - Downgrade() -> from excl to shared
  - Upgrade() -> from shared to excl

# Code for reader-writer locks

```
struct rwlock {  
    int nActive; /* num of  
active readers, or -1 if a writer  
is active */  
  
    int nPendingReads;  
  
    int nPendingWrites;  
  
    spinlock_t sl;  
  
    condition canRead;  
  
    condition canWrite;  
};
```

```
void lockShared (struct rwlock *r)  
{  
    spin_lock (&r->sl);  
    r->nPendingReads++;  
  
    if (r->nPendingWrites > 0)  
        wait (&r->canRead, &r->sl ); /*don't starve  
writers */  
  
    while {r->nActive < 0) /* someone has  
exclusive lock */  
        wait (&r->canRead, &r->sl);  
    r->nActive++;  
    r->nPendingReads--;  
    spin_unlock (&r->sl);  
}
```

# Code for reader-writer locks

```
void unlockShared (struct rwlock *r)    void lockExclusive (struct rwlock *r)
{
    spin_lock (&r->sl);
    r->nActive--;
    if (r->nActive == 0) {
        spin_unlock (&r->sl);
        do signal (&r->canWrite);
    } else
        spin_unlock (&r->M);
}

(
    spin_lock (&r->sl);
    r->nPendingWrtes++;
    while (r->nActive)
        wait (&r->canWrite, &r->sl);
    r->nPendingWrites--;
    r->nActive = -1;
    spin_unlock (&r->sl);
}
```

# Code for reader-writer locks

```
void unlockExclusive (struct rwlock *r){  
    boolean t wakeReaders;  
    spin_lock (&r->sl);  
    r->nActive = 0;  
    wakeReaders = (r->nPendingReads != 0);  
    spin_unlock (&r->sl);  
    if (wakeReaders)  
        do broadcast (&r->canRead); /* wake  
allreaders */  
    else  
        do_signal (&r->canWrite);  
    /*wakeasinglewri r */  
}
```

**Try writing code for  
downgrade and  
upgrade**

**Try writing a reader-  
writer lock using  
semaphores!**

# Dining-Philosophers Problem

- Philosophers spend their lives thinking and eating

- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl

- Need both to eat, then release both when done

- In the case of 5 philosophers

- Shared data

- Bowl of rice (data set)

- Semaphore chopstick [5] initialized to 1



# Dining philosophers: One solution

The structure of Philosopher i:

```
do {  
    wait ( chopstick[i] );  
    wait ( chopStick[ (i + 1) % 5] );  
    // eat  
    signal ( chopstick[i] );  
    signal (chopstick[ (i + 1) % 5] );  
    // think  
} while (TRUE);
```

What is the problem with this algorithm?

# Dining philosophers: Possible approaches

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available
  - to do this, she must pick them up in a critical section
- Use an asymmetric solution
  - that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick
  - whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick.

# Other solutions to dining philosopher's problem

- Using higher level synchronization primitives like 'monitors'

# **Practical Problems**

# Lost Wakeup problem

- The sleep/wakeup mechanism does not function correctly on a multiprocessor.
- Consider a potential race:
  - Thread T1 has locked a resource R1.
  - Thread T2, running on another processor, tries to acquire the resource, and finds it locked.
  - T2 calls sleep() to wait for the resource.
  - Between the time T2 finds the resource locked and the time it calls sleep(), T1 frees the resource and proceeds to wake up all threads blocked on it.
  - Since T2 has not yet been put on the sleep queue, it will miss the wakeup.
  - The end result is that the resource is not locked, but T2 is blocked waiting for it to be unlocked.
  - If no one else tries to access the resource, T2 could block indefinitely.
  - This is known as the lost wakeup problem,
  - Requires some mechanism to combine the test for the resource and the call to sleep() into a single atomic operation.

# Lost Wakeup problem

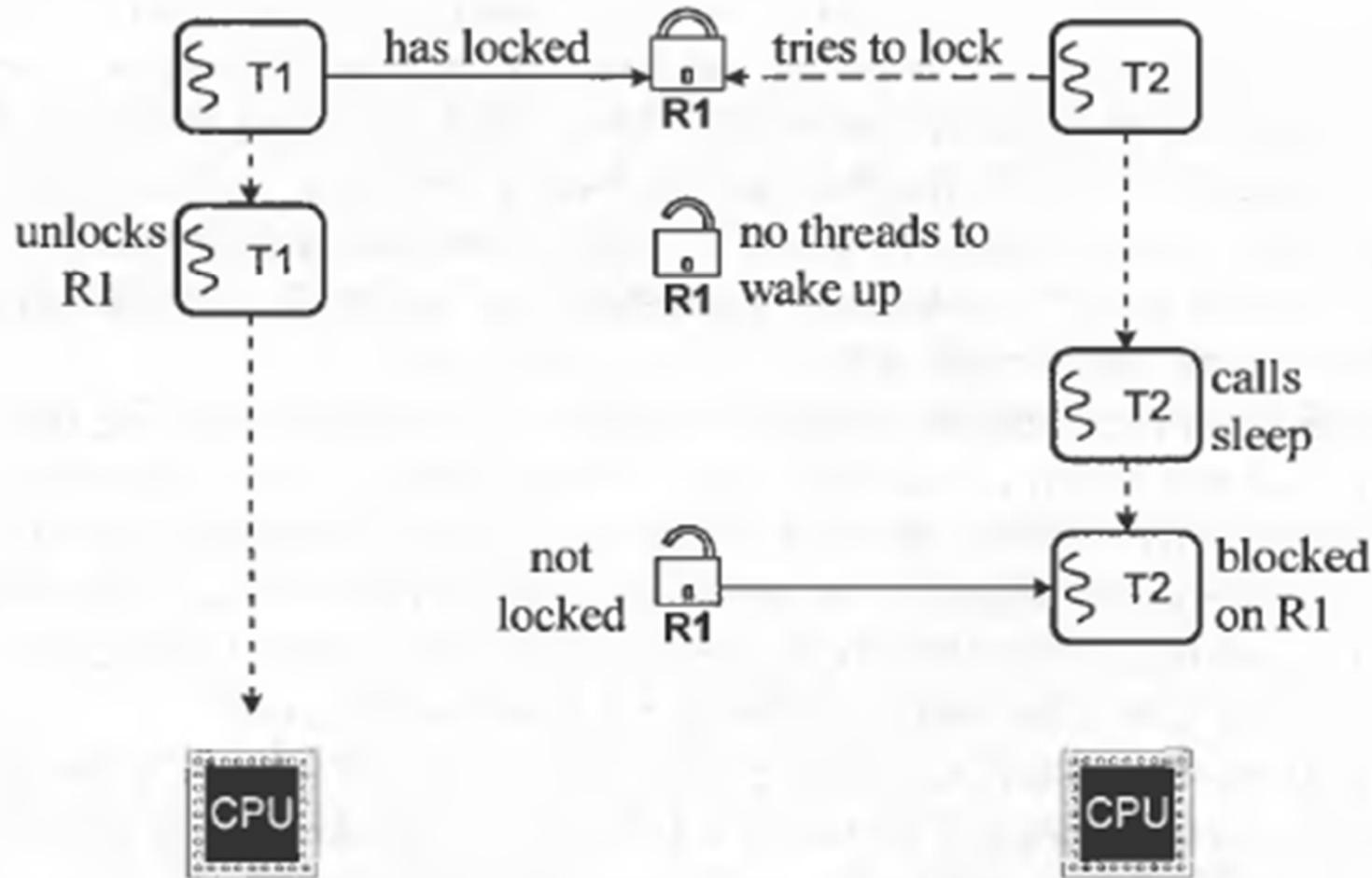


Figure 7-6. The lost wakeup problem.

# Thundering herd problem

- Thundering Herd problem

- On a multiprocessor, if several threads were locked the resource
- Waking them all may cause them to be simultaneously scheduled on different processors
- and they would all fight for the same resource again.

- Starvation

- Even if only one thread was blocked on the resource, there is still a time delay between its waking up and actually running.
- In this interval, an unrelated thread may grab the resource causing the awakened thread to block again. If this happens frequently, it could lead to starvation of this thread.
- This problem is not as acute on a uniprocessor, since by the time a thread runs, whoever had locked the resource is likely to have released it.

# **Case Studies**

# Linux Synchronization

- Prior to kernel Version 2.6, disables interrupts to implement short critical sections
- Version 2.6 and later, fully preemptive
- Linux provides:
  - semaphores
  - spinlocks
  - reader-writer versions of both
  - Atomic integers
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

# Linux Synchronization

## ④ Atomic variables

**atomic\_t** is the type for atomic integer

## ④ Consider the variables

**atomic\_t counter;**

**int value;**

| <i>Atomic Operation</i>        | <i>Effect</i>          |
|--------------------------------|------------------------|
| atomic_set(&counter,5);        | counter = 5            |
| atomic_add(10,&counter);       | counter = counter + 10 |
| atomic_sub(4,&counter);        | counter = counter - 4  |
| atomic_inc(&counter);          | counter = counter + 1  |
| value = atomic_read(&counter); | value = 12             |

# Pthreads synchronization

- ❑ Pthreads API is OS-independent

- ❑ It provides:

- ❑ mutex locks

- ❑ condition variables

- ❑ Non-portable extensions include:

- ❑ read-write locks

- ❑ spinlocks

# Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments
- Starts as a standard semaphore spin-lock
  - If lock held, and by a thread running on another CPU, spins
  - If lock held by non-run-state thread, block and sleep waiting for signal of lock being released

# Solaris Synchronization

- Uses condition variables
- Uses readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
- Turnstiles are per-lock-holding-thread, not per-object
- Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile

# **Synchronization issues in xv6 kernel**

# Difference approaches

- Pros and Cons of locks
- Locks ensure serialization
- Locks consume time !
- Solution – 1
  - One big kernel lock
  - Too inefficient
- Solution – 2
  - One lock per variable
    - Often un-necessary, many data structures get manipulated in once place, one lock for all of them may work
  - Problem: ptable.lock for the entire array and every element within
  - Alternatively: one lock for array, one lock per array entry

# Three types of code

- System calls code
- Can it be interruptible?
- If yes, when?
- Interrupt handler code
  - Disable interrupts during interrupt handling or not?
  - Deadlock with iderw ! - already seen
- Process's user code
  - Ignore. Not concerned with it now.

# Interrupts enabling/disabling in xv6

- Holding every spinlock disables interrupts!
- System call code or Interrupt handler code won't be interrupted if
- The code path followed took at least one spinlock !
- Interrupts disabled only on that processor!
- Acquire calls pushcli() before xchg()
- Release calls popcli() after xchg()

# Memory ordering

- Compiler may generate machine code for out-of-order execution !
- Processor pipelines can also do the same!
- This often improves performance
- Compiler may reorder 4 after 6 --> Trouble!
- Solution: Memory barrier
  - `sync_synchronize()`, provided by **GCC**
- Do not reorder across this line
- Done only on acquire and release()

## ▪ Consider this

- 1) `I = malloc(sizeof *I);`
- 2) `I->data = data;`
- 3) `acquire(&listlock);`
- 4) `I->next = list;`
- 5) `list = I;`
- 6) `release(&listlock);`

# Lost Wakeup?

- Do we have this problem in xv6?

- Let's analyze again!

- The race in `acquiresleep()`'s call to `sleep()` and `releasesleep()`

- T1 holding lock, T2 willing to acquire lock

- Both running on different processor

- Or both running on same processor

- What happens in both scenarios?

- Introduce a T3 and T4 on each of two different processors. Now how does the scenario change?

- See page 69 in xv6 book revision-11.

# Code of sleep()

```
if(lk != &ptable.lock){  
    acquire(&ptable.lock);  
    release(lk);  
}
```

- Why this check?

- Deadlock otherwise!

- Check: wait() calls with ptable.lock held!

# Exercise question : 1

Sleep has to check lk != &ptable.lock to avoid a deadlock

Suppose the special case were eliminated by replacing

```
if(lk != &ptable.lock){  
    acquire(&ptable.lock);  
    release(lk);  
}
```

with

```
release(lk);  
acquire(&ptable.lock);
```

Doing this would break sleep. How?

'

# bget() problem

- bget() panics if no free buffers!
- Quite bad
- Should sleep !
- But that will introduce many deadlock problems. Which ones ?

# **iget() and ilock()**

- ❑ **iget() does no hold lock on inode**
- ❑ **ilock() does**
- ❑ **Why this separation?**
- ❑ **Performance? If you want only “read” the inode, then why lock it?**
- ❑ **What if iget() returned the inode locked?**

# Interesting cases in namex()

```
while((path = skipel(path, name)) != 0){  
    ilock(ip);  
    if(ip->type != T_DIR){  
        iunlockput(ip);  
        return 0;  
    }  
    if(nameiparent && *path == '\0'){  
        // Stop one level early.  
        iunlock(ip);  
        return ip;  
    }  
    if((next = dirlookup(ip, name, 0)) == 0){  
        iunlockput(ip);  
        return 0;  
    }  
    iunlockput(ip);  
    ip = next;  
}  
--> only after obtaining next from  
dirlookup() and iget() is the lock  
released on ip;  
-> lock on next obtained only after  
releasing the lock on ip. Deadlock  
possible if next was ".">
```

# Xv6

Interesting case of holding and releasing  
ptable.lock in scheduling

**One process acquires, another releases!**

# Giving up CPU

- A process that wants to give up the CPU
  - must acquire the process table lock ptable.lock
  - release any other locks it is holding
  - update its own state (proc->state),
  - and then call sched()
- Yield follows this convention, as do sleep and exit
  - Lock held by one process P1, will be released another process P2 that starts running after sched()
  - remember P2 returns either in yield() or sleep()
  - In both, the first thing done is releasing ptable.lock

# Interesting race if ptable.lock is not held

- Suppose P1 calls yield()
- Suppose yield() does not take ptable.lock
- Remember yield() is for a process to give up CPU
- Yield sets process state of P1 to RUNNABLE
- Before yield's sched() calls swtch()
- Another processor runs scheduler() and runs P1 on that processor
- Now we have P1 running on both processors!
- P1 in yield taking ptable.lock prevents this

# Homework

- Read the version-11 textbook of xv6
- Solve the exercises!