

Overview of Operating Systems (1)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems

8/21/2019



Major OS Components

- ❑ processes/threads
- ❑ memory
- ❑ I/O
- ❑ secondary storage
- ❑ file systems
- ❑ protection
- ❑ shells (command interpreter, or OS UI)
- ❑ GUI
- ❑ networking

Process management

- An OS executes many kinds of activities:
 - users' programs
 - batch jobs or scripts
 - system programs
 - print spoolers, name servers, file servers, network daemons, ...
- Each of these activities is encapsulated in a **process**
 - a process includes the execution **context**
 - PC, registers, VM, OS resources (e.g., open files), etc...
 - plus the program itself (code and data)
 - the OS's process module manages these processes
 - creation, destruction, scheduling, ...

Important: Processes vs. Threads

- Soon, we will separate the “thread of control” aspect of a process (program counter, call stack) from its other aspects (address space, open files, owner, etc.). And we will allow each {process / address space} to have multiple threads of control.
- But for now – for simplicity and for historical reasons – consider each {process / address space} to have a single thread of control.



Program/processor/process

- ❑ Note that a program is totally passive
 - ❑ just bytes on a disk that encode instructions to be run
- ❑ A process is an instance of a program being executed by a (real or virtual) processor
 - ❑ at any instant, there may be many processes running copies of the same program (e.g., an editor); each process is separate and (usually) independent
 - ❑ Linux: `ps -auwwx` to list all processes

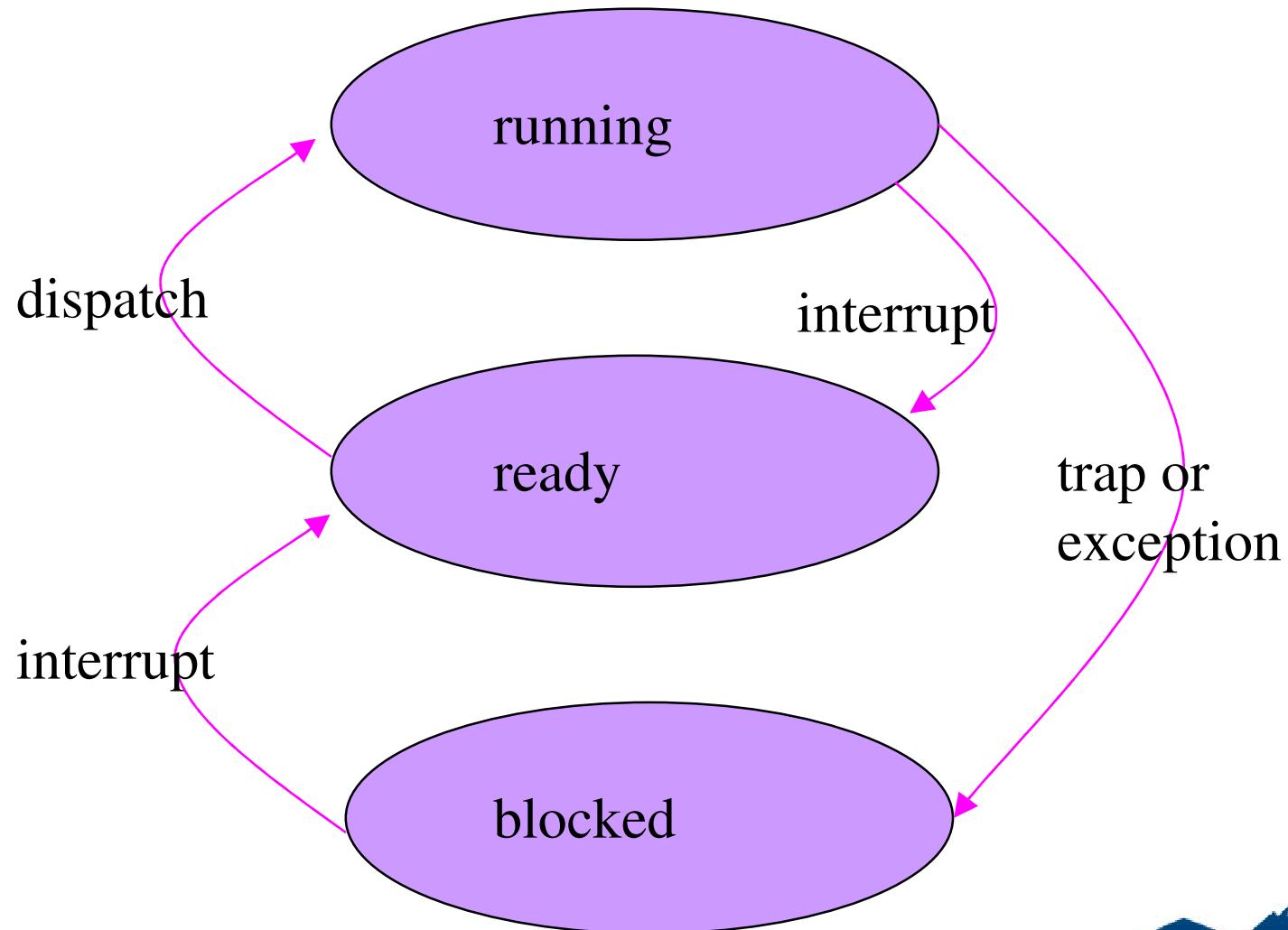
process A

code	page
stack	tables
PC	resources
registers	

process B

code	page
stack	tables
PC	resources
registers	

States of a user process



Process operations

- The OS provides the following kinds operations on processes (i.e., the process abstraction interface):
 - create a process
 - delete a process
 - suspend a process
 - resume a process
 - clone a process
 - inter-process communication
 - inter-process synchronization
 - create/delete a child process

Memory management

- ❑ The primary memory is the directly accessed storage for the CPU
 - ❑ programs must be resident in memory to execute
 - ❑ memory access is fast
 - ❑ but memory doesn't survive power failures
- ❑ OS must:
 - ❑ allocate memory space for programs
 - ❑ deallocate space when needed by rest of system
 - ❑ maintain mappings from physical to virtual memory
 - ❑ through **page tables**
 - ❑ decide how much memory to allocate to each process
 - ❑ a policy decision
 - ❑ decide when to remove a process from memory
 - ❑ also policy

I/O

- A big chunk of the OS kernel deals with I/O
 - hundreds of thousands of lines in Windows, Unix, etc.
- The OS provides a standard interface between programs (user or system) and devices
 - file system (disk), sockets (network), frame buffer (video)
- **Device drivers** are the routines that interact with specific device types
 - **encapsulates** device-specific knowledge
 - e.g., how to initialize a device, how to request I/O, how to handle interrupts or errors
 - examples: SCSI device drivers, Ethernet card drivers, video card drivers, sound card drivers, ...
- Note: Windows has ~35,000 device drivers!

Secondary storage

- Secondary storage (disk, FLASH, tape) is persistent memory
 - often magnetic media, survives power failures
- Routines that interact with disks are typically at a very low level in the OS
 - used by many components (file system, VM, ...)
 - handle scheduling of disk operations, head movement, error handling, and often management of space on disks
- Usually independent of file system
 - although there may be cooperation
 - file system knowledge of device details can help optimize performance
 - e.g., place related files close together on disk

PC Hardware and x86

Dr. Jun Zheng
CSE325 Principles of Operating
Systems
8/23/2019



A PC



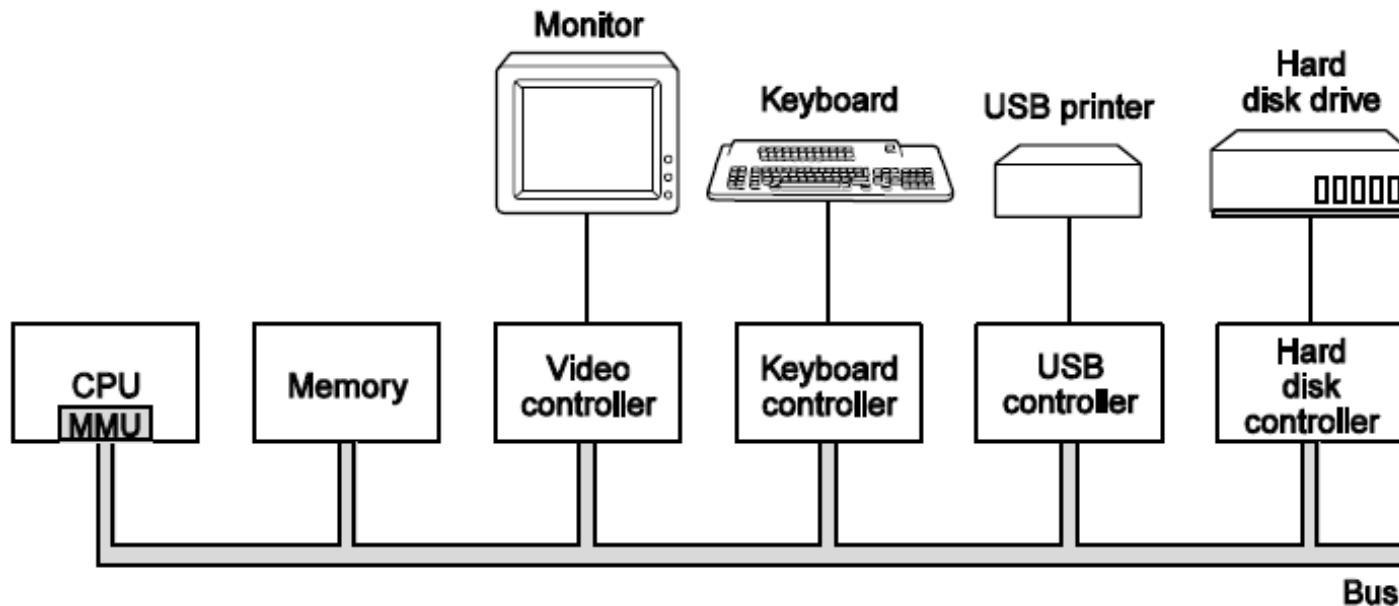
□ How to make it do something useful?

Outline

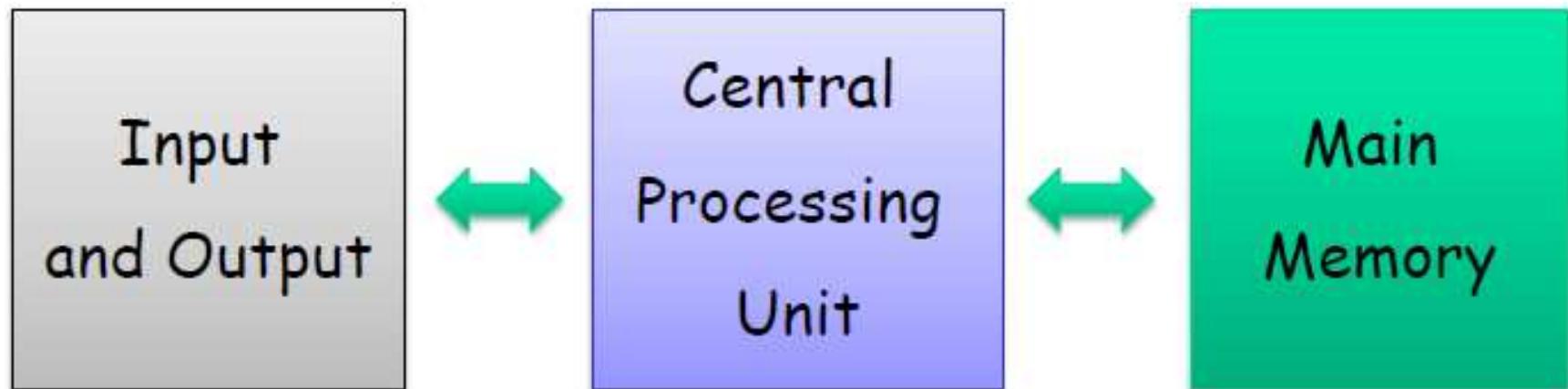
- ❑ PC architecture
- ❑ x86 Instruction set
- ❑ gcc calling conventions

PC Organization

- One or more CPUs, memory, and device controllers connect through system bus

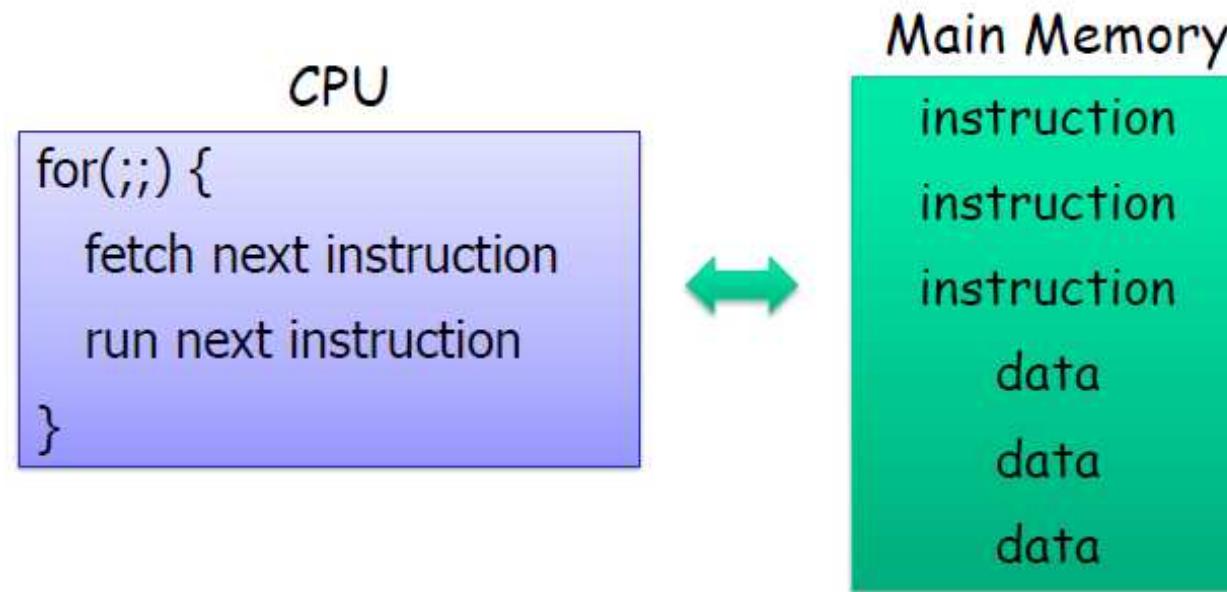


Abstract Model



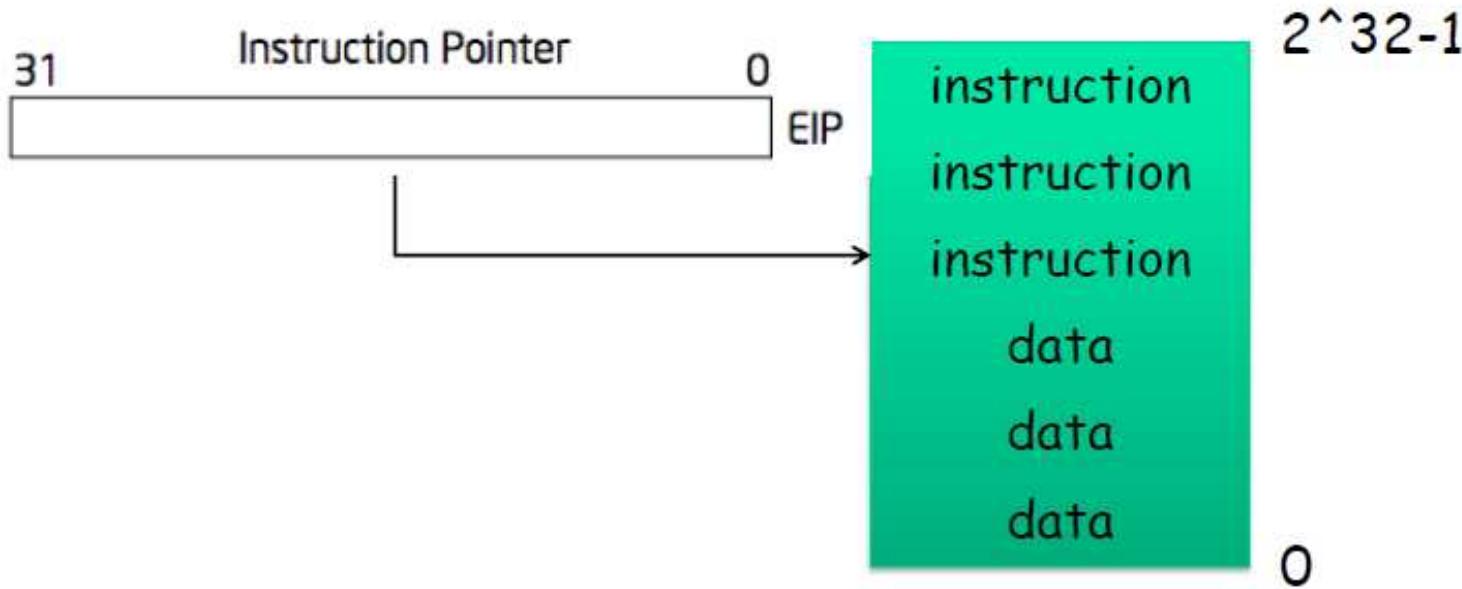
- I/O: communicating data to and from devices
- CPU: digital logic for doing computation
- Memory: N words of B bits

The Stored Program Computer



- ❑ Memory holds both instructions and data
- ❑ CPU interprets instructions
- ❑ Instructions read/write data

x86 Implementation



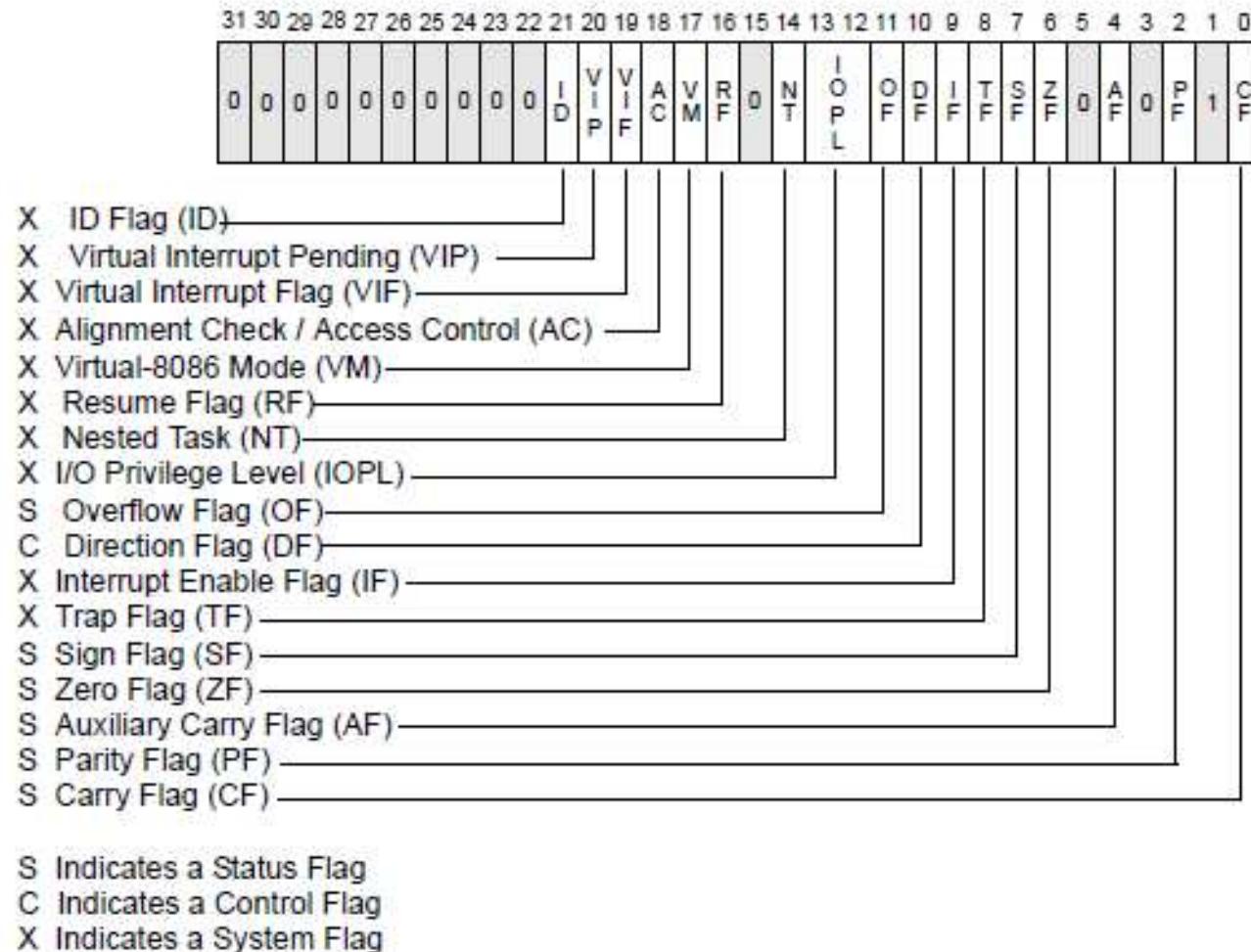
- ❑ EIP incremented after each instruction
- ❑ Variable length instructions
- ❑ EIP modified by CALL, RET, JMP, conditional JMP

Registers: Work Space

General-Purpose Registers				16-bit	32-bit	
31	16 15	8 7	0	AX	EAX	
	AH	AL		BX	EBX	ESP: stack pointer
	BH	BL		CX	ECX	EBP: frame base pointer
	CH	CL		DX	EDX	ESI: source index
	DH	DL			EBP	EDI: destination index
	BP				ESI	
	SI				EDI	
	DI				ESP	
	SP					

- 8, 16, and 32 bit versions
- Example: **ADD EAX, 10**
 - More: **SUB, AND, etc.**
- By convention some for special purposes

EFLAGS Register



☐ Track current CPU status

x86 Instruction Set

❑ Instruction syntax

- ❑ Intel manual Volume 2: op dst, src
- ❑ AT&T (gcc/gas): op src, dst
 - ❑ op uses suffix b, w, l for 8, 16, 32-bit operands
 - ❑ xv6, JOS

❑ Instruction classes

- ❑ Data movement: MOV, PUSH, POP, ...
- ❑ Arithmetic: TEST, SHL, ADD, AND, ...
- ❑ I/O: IN, OUT, ...
- ❑ Control: JMP, JZ, JNZ, CALL, RET
- ❑ String: MOVSB, REP, ...
- ❑ System: INT, IRET

Memory: Addressing Examples

Address	Value	Operand	Value	Comment
0x100	0xFF	%eax	0x100	Value in the register
0x104	0xAB	0x104	0xAB	Value is at the address
0x108	0x13	\$0x108	0x108	Value is the value (\$ means immediate, i.e. constant, value)
0x10C	0x11	(%eax)	0xFF	value is at the address stored in the register -> GTV@(reg)
		4(%eax)	0xAB	GTV@(4 + reg)
Register	Value	9(%eax, %edx)	0x11	GTV@(9 + reg + reg)
%eax	0x100	0xFC(,%ecx, 4)	0xFF	GTV@(0xFC + 0 + reg*4)
%ecx	0x1	(%eax, %edx, 4)	0x11	GTV@(reg + reg*4)
%edx	0x3			

- ❑ Red means memory address

Memory: Addressing Modes

movl %eax, %edx	edx = eax;	<i>register mode</i>
movl \$0x123, %edx	edx = 0x123;	<i>immediate</i>
movl 0x123, %edx	edx = *(int32_t*)0x123;	<i>direct</i>
movl (%ebx), %edx	edx = *(int32_t*)ebx;	<i>indirect</i>
movl 4(%ebx), %edx	edx = *(int32_t*)(ebx+4);	<i>displaced</i>

- Memory instructions: **MOV**, **PUSH**, **POP**, etc.
- More instructions can take a memory address

Stack Memory + Operations

Example instruction What it does

pushl %eax

subl \$4, %esp

movl %eax, (%esp)

popl %eax

movl (%esp), %eax

addl \$4, %esp

call 0x12345

pushl %eip (*)

movl \$0x12345, %eip (*)

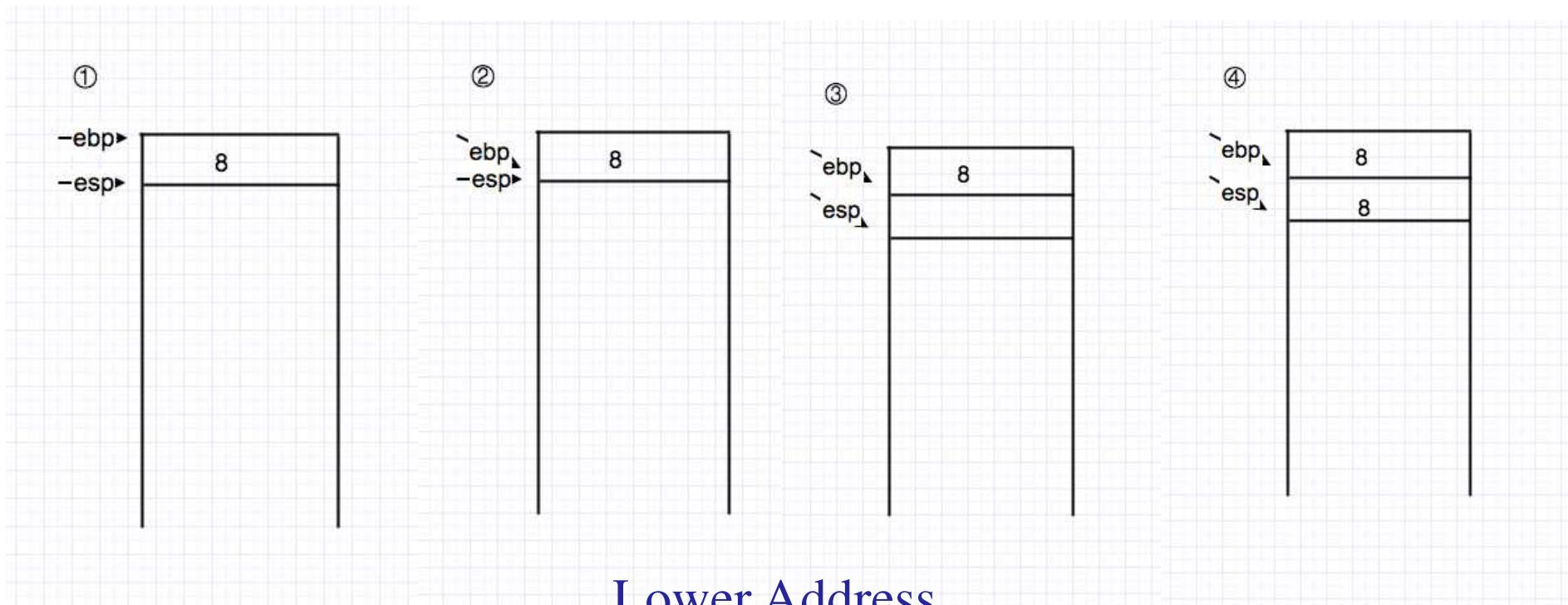
ret

popl %eip (*)

- For implementing function calls
- Stack grows “down” on x86

Stack Operation Example

```
pushl $8          ①  
movl %esp, %ebp ②  
subl $4, %esp    ③  
movl $8, (%esp) ④
```



More Memory

- 8086 16-bit register and 20-bit bus addresses
- These extra 4 bits come from segment register
 - **CS**: code segment, for EIP
 - Instruction address: **CS** * 16 + **EIP**
 - **SS**: stack segment, for **ESP** and **EBP**
 - **DS**: data segment for load/store via other registers
 - **ES**: another data segment, destination for string ops
- Make life more complicated
 - Cannot directly use 16-bit stack address as pointer
 - For a far pointer programmer must specify segment reg
 - Pointer arithmetic and array indexing across seg bound

And More Memory

- ❑ 80386: 32 bit register and addresses (1985)
- ❑ AMD k8: 64 bit (2003)
 - ❑ RAX instead of EAX
 - ❑ x86-64, x64, amd64, intel64: all same thing
- ❑ Backward compatibility
 - ❑ Boots in 16-bit mode; `bootasm.S` switches to 32
 - ❑ Prefix `0x66` gets 32-bit mode instructions
 - ❑ `MOVW` in 32-bit mode = `0x66 + MOVW` in 16-bit mode
 - ❑ `.code32` in `bootasm.S` tells assembler to insert `0x66`
- ❑ 80386 also added virtual memory addresses

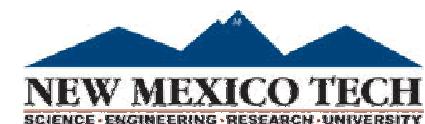
I/O Space and Instructions

```
#define DATA_PORT    0x378
#define STATUS_PORT   0x379
#define BUSY 0x80
#define CONTROL_PORT 0x37A
#define STROBE 0x01
void
lpt_putc(int c)
{
    /* wait for printer to consume previous byte */
    while((inb(STATUS_PORT) & BUSY) == 0)
        ;

    /* put the byte on the parallel lines */
    outb(DATA_PORT, c);

    /* tell the printer to look at the data */
    outb(CONTROL_PORT, STROBE);
    outb(CONTROL_PORT, 0);
}
```

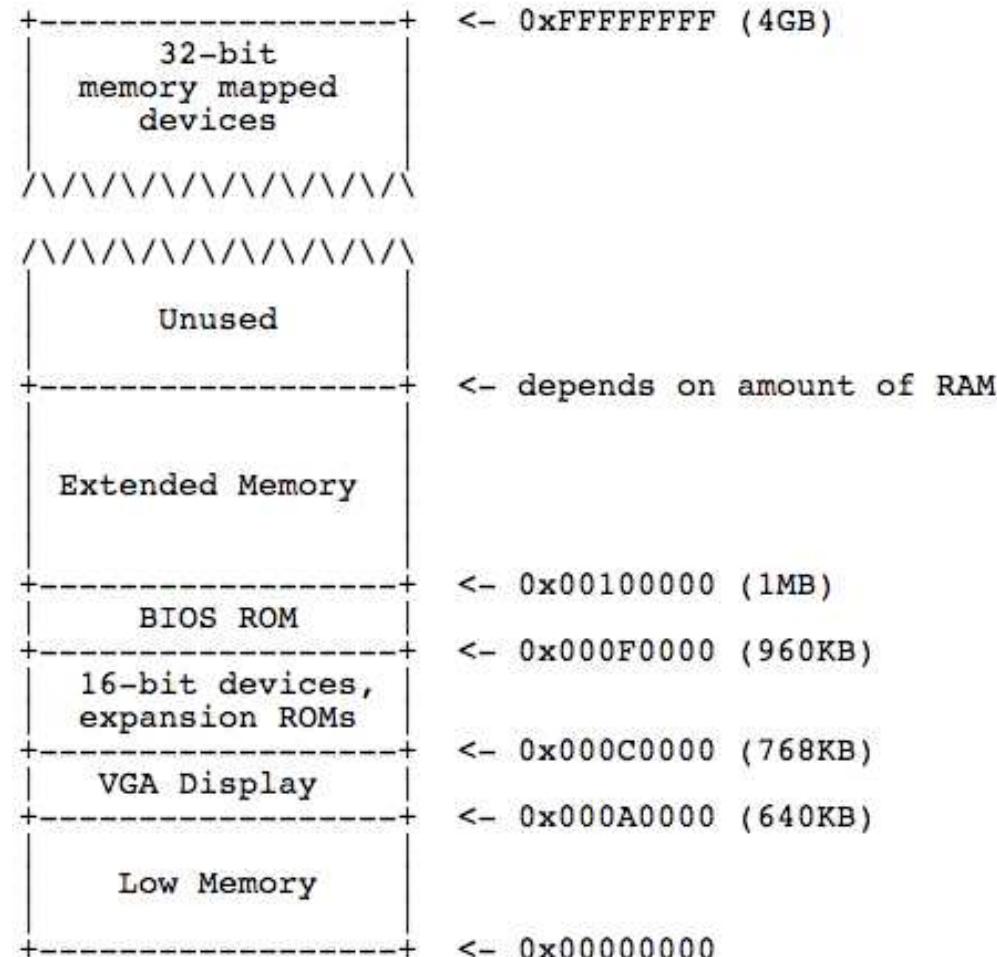
- 8086: only 1024 I/O addresses



Memory-mapped I/O

- ❑ Use normal addresses for I/O
 - ❑ No special instructions
 - ❑ No 1024 limit
 - ❑ Hardware routes to appropriate device
- ❑ Works like “magic” memory
 - ❑ I/O device addressed and accessed like memory
 - ❑ However, reads and writes have “side effects”
 - ❑ Read result can change due to external events

Memory Layout



gcc Inline Assembly

- Can embed assembly code in C code
 - Many examples in xv6
- Basic syntax: `asm ("assembly code")`
 - e.g., `asm ("movl %eax, %ebx")`
- Advanced syntax:

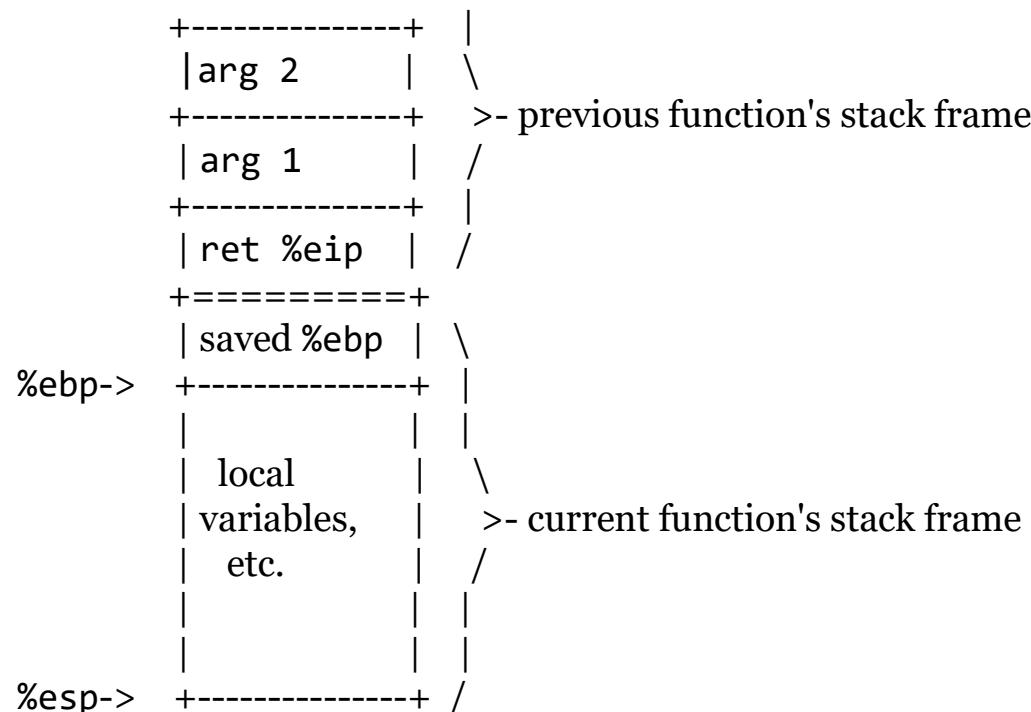
```
asm ( assembler template
      : output operands /* optional */
      : input operands /* optional */
      : list of clobbered registers /* optional */ );
```

e.g.,

```
int val;
asm ("movl %%ebp,%0" : "=r" (val));
```

gcc Calling Conventions

- Functions can do anything that doesn't violate contract. By convention, GCC does more:
 - each function has a stack frame marked by `%ebp`, `%esp`
 - `%esp` can move to make stack frame bigger, smaller
 - `%ebp` points at saved `%ebp` from previous function, chain to walk stack



gcc Calling Conventions (cont.)

- ❑ `%eax` contains return value, `%ecx`, `%edx` may be trashed
- ❑ 64 bit return value: `%eax + %edx`
- ❑ `%ebp`, `%ebx`, `%esi`, `%edi` must be as before call
- ❑ Caller saved: `%eax`, `%ecx`, `%edx`
- ❑ Callee saved: `%ebp`, `%ebx`, `%esi`, `%edi`

Overview of Operating Systems (2)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems
8/26/2019



File systems

- ❑ Secondary storage devices are crude and awkward
 - ❑ e.g., “write a 4096 byte block to sector 12”
- ❑ File system: a convenient abstraction
 - ❑ defines logical objects like **files** and **directories**
 - ❑ hides details about where on disk files live
 - ❑ as well as operations on objects like read and write
 - ❑ read/write byte ranges instead of blocks
- ❑ A **file** is the basic unit of long-term storage
 - ❑ file = named collection of persistent information
- ❑ A **directory** is just a special kind of file
 - ❑ directory = named file that contains names of other files and metadata about those files (e.g., file size)
- ❑ Note: Sequential byte stream is only one possibility!

File system operations

- The file system interface defines standard operations:
 - file (or directory) creation and deletion
 - manipulation of files and directories (read, write, extend, rename, protect)
 - copy
 - lock
- File systems also provide higher level services
 - accounting and quotas
 - backup (must be incremental and online!)
 - (sometimes) indexing or search
 - (sometimes) file versioning

Protection

- ❑ Protection is a general mechanism used throughout the OS
 - ❑ all resources needed to be protected
 - ❑ memory
 - ❑ processes
 - ❑ files
 - ❑ devices
 - ❑ CPU time
 - ❑ ...
 - ❑ protection mechanisms help to detect and contain unintentional errors, as well as preventing malicious destruction

Command interpreter (shell)

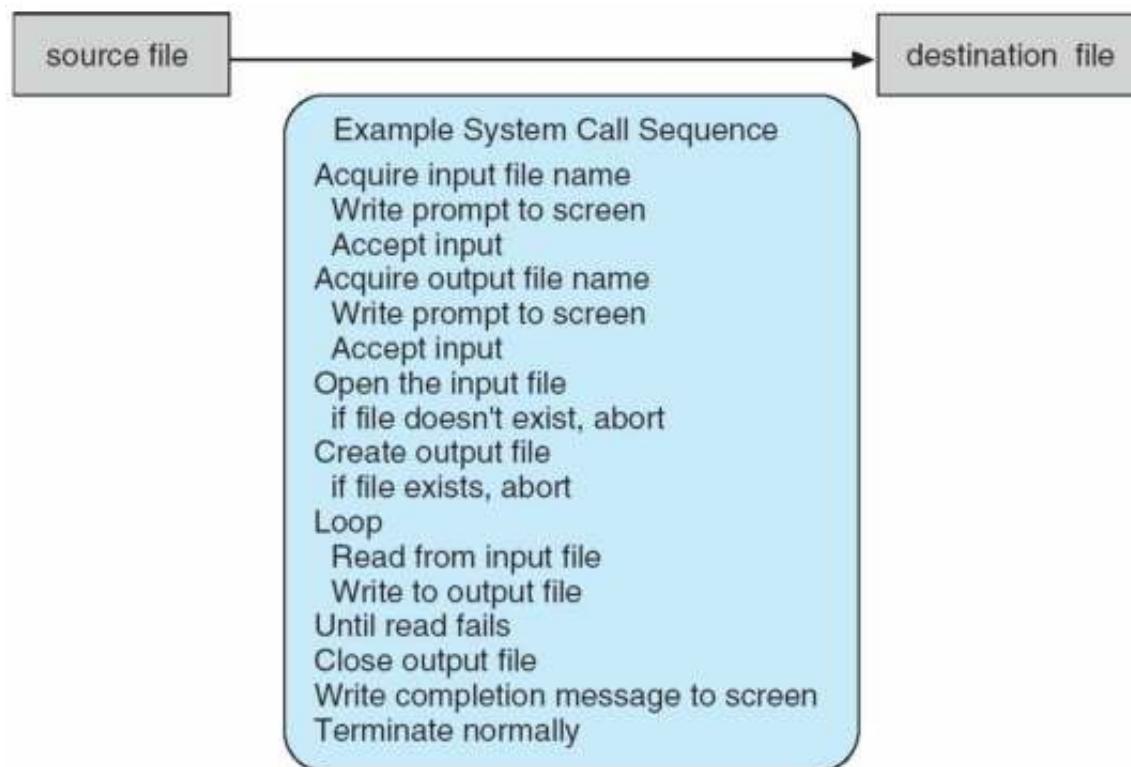
- A particular program that handles the interpretation of users' commands and helps to manage processes
 - user input may be from keyboard (command-line interface), from script files, or from the mouse (GUIs)
 - allows users to launch and control new programs
 - Sometimes multiple flavors implemented – **shells**
 - Sometimes commands built-in, sometimes just names of programs
 - If the latter, adding new features doesn't require shell modification

System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

Example of System Calls

System call sequence to copy the contents of one file to another file



Example of Standard API

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t      read(int fd, void *buf, size_t count)
```

return value	function name	parameters
-----------------	------------------	------------

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

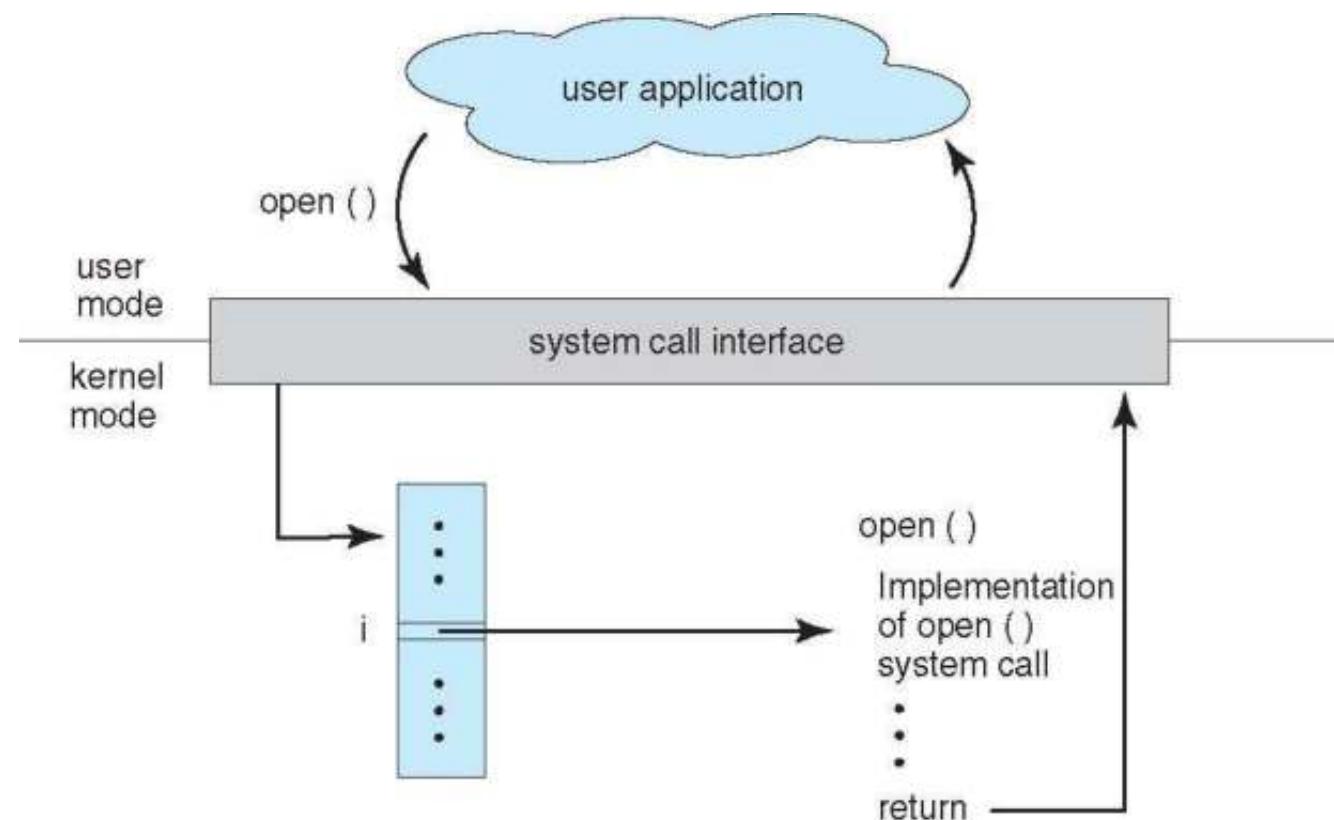
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns -1.

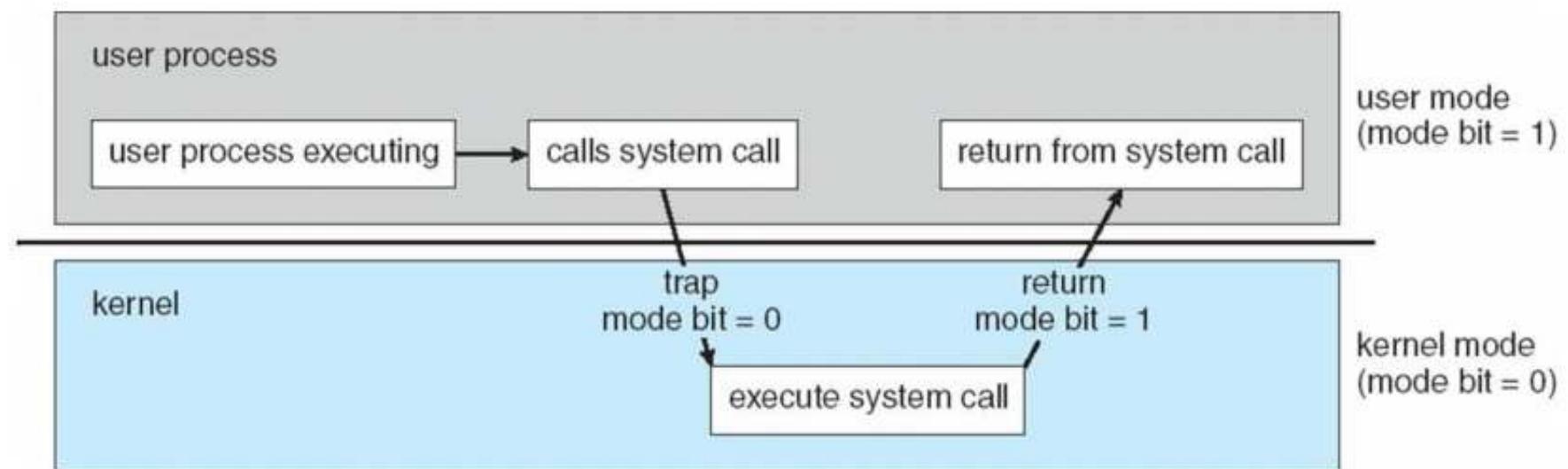
System Call Implementation

- Typically, a number associated with each system call
 - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
 - Just needs to obey API and understand what OS will do as a result call
 - Most details of OS interface hidden from programmer by API
 - Managed by run-time support library (set of functions built into libraries included with compiler)

API – System Call – OS Relationship



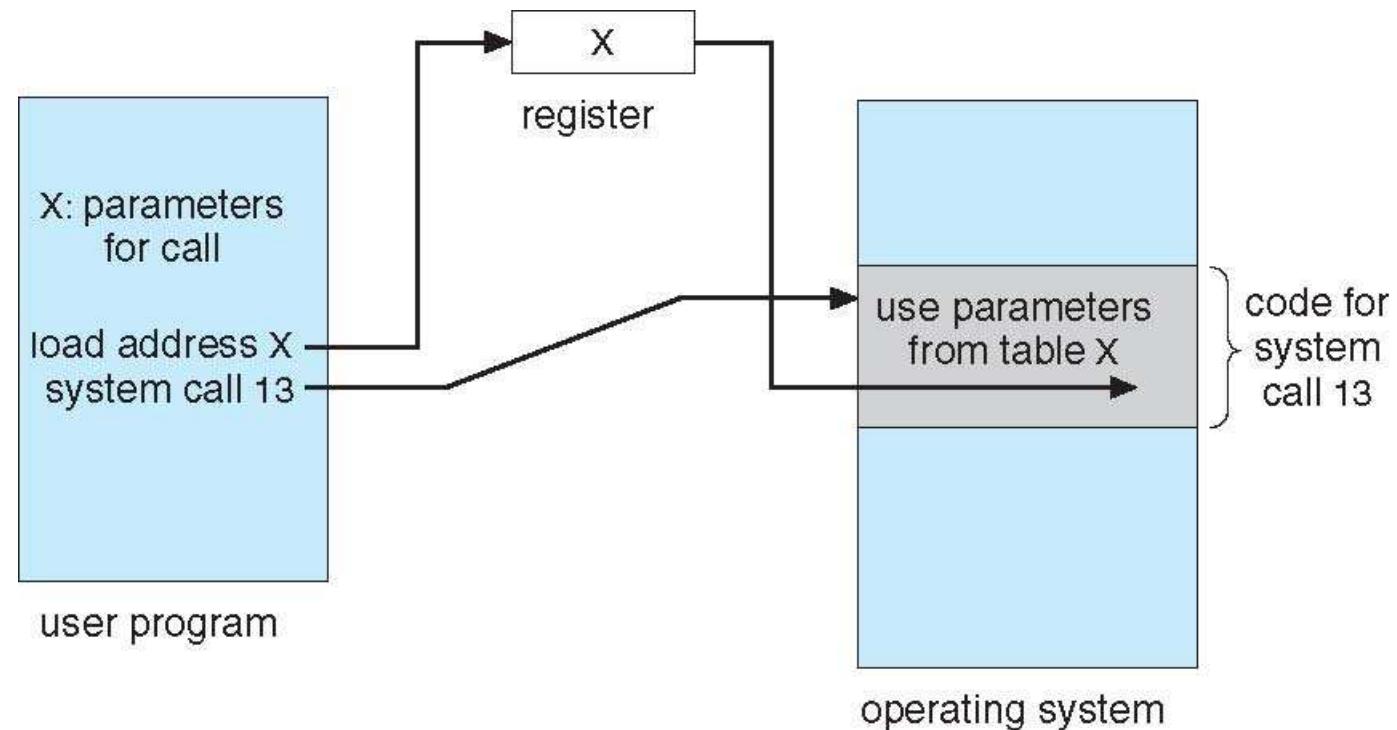
Mode Transfer



System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
 - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
 - Simplest: pass the parameters in registers
 - In some cases, may be more parameters than registers
 - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
 - This approach taken by Linux and Solaris
 - Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system
 - Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table

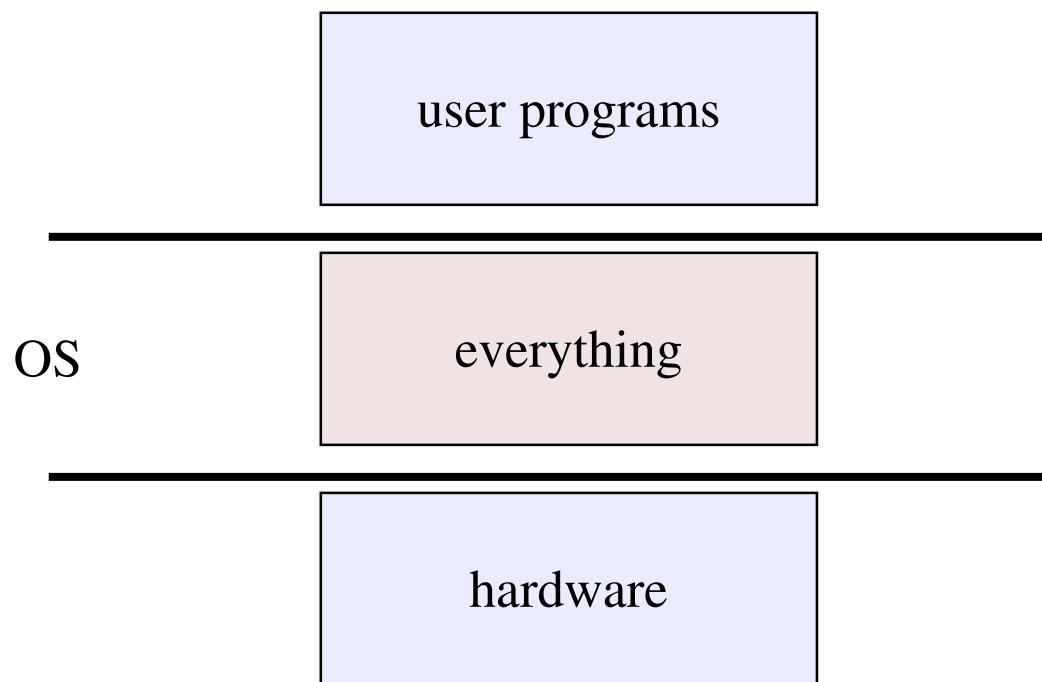


OS structure

- ❑ An OS consists of all of these components, plus:
 - ❑ many other components
 - ❑ system programs (privileged and non-privileged)
 - ❑ e.g., bootstrap code, the init program, ...
- ❑ Major issue:
 - ❑ how do we organize all this?
 - ❑ what are all of the code modules, and where do they exist?
 - ❑ how do they cooperate?
- ❑ Massive software engineering and design problem
 - ❑ design a large, complex program that:
 - ❑ performs well, is reliable, is extensible, is backwards compatible, ...

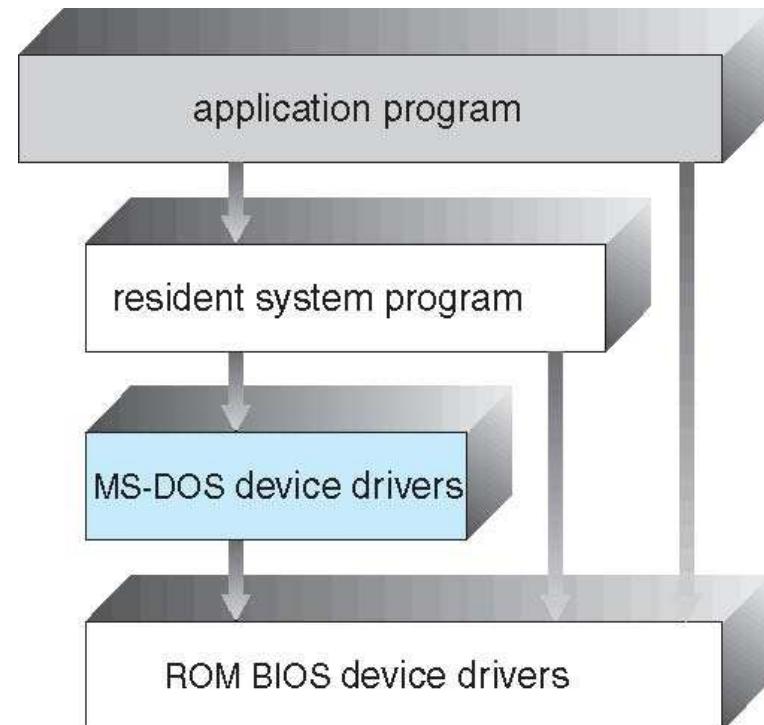
Early Structure: Monolithic

- ❑ Traditionally, OS's are built as a **monolithic** entity:
 - ❑ Single linked binary
 - ❑ Any function can call any other function
 - ❑ Example: MS-DOS, xv6



MS-DOS System Structure

- ❑ MS-DOS – written to provide the most functionality in the least space
- ❑ Not divided into modules
- ❑ Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated

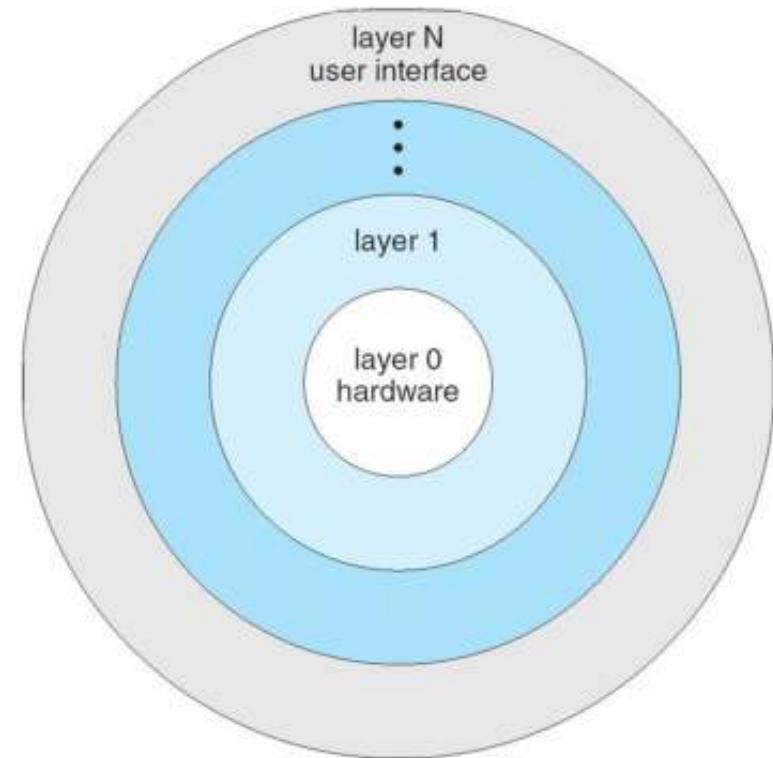


Monolithic Structure

- Major advantage:
 - cost of module interactions is low (procedure call)
- Disadvantages:
 - As system scales, it becomes:
 - Hard to understand
 - Hard to modify
 - Hard to maintain
 - Unreliable (no isolation between system modules)
- What is the alternative?
 - Find a way to organize the OS in order to simplify its design and implementation

Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers.
- The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers
- Each layer can be tested and verified independently

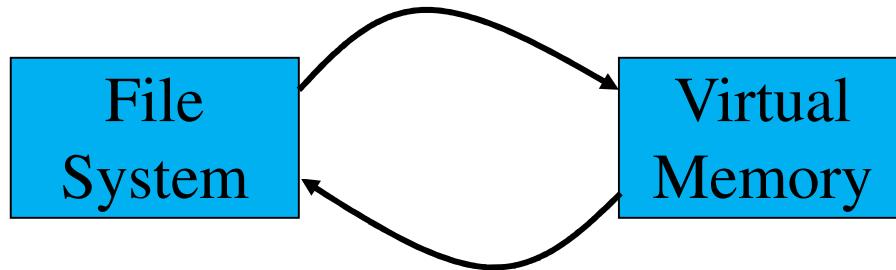


THE System

- The first description of layered approach was Dijkstra's THE system (1968!)
 - Layer 5: **Job Managers**
 - Execute users' programs
 - Layer 4: **Device Managers**
 - Handle devices and provide buffering
 - Layer 3: **Console Manager**
 - Implements virtual consoles
 - Layer 2: **Page Manager**
 - Implements virtual memories for each process
 - Layer 1: **Kernel**
 - Implements a virtual processor for each process
 - Layer 0: **Hardware**

Problems with Layering

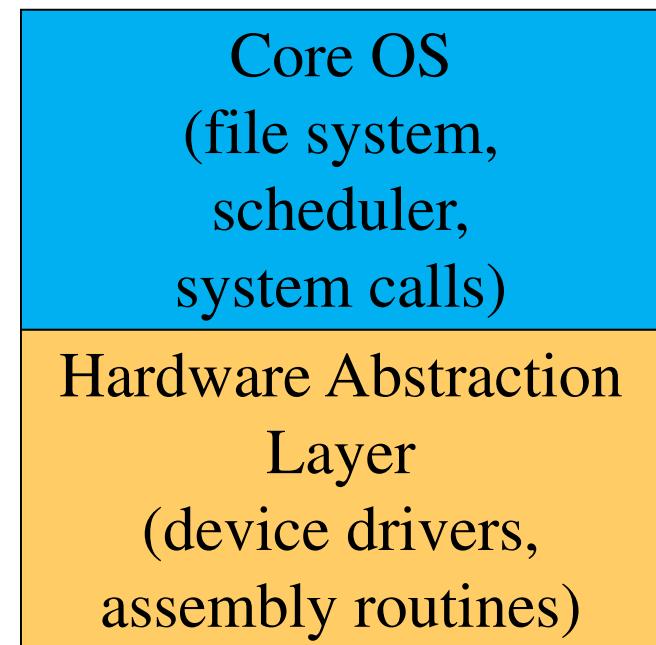
- Strict hierarchical structure is too inflexible
 - Real systems have “uses” cycles
 - File system requires virtual memory services (buffers)
 - Virtual memory would like to use files for its backing store



- Poor performance
 - Each layer crossing has **overhead** associated with it

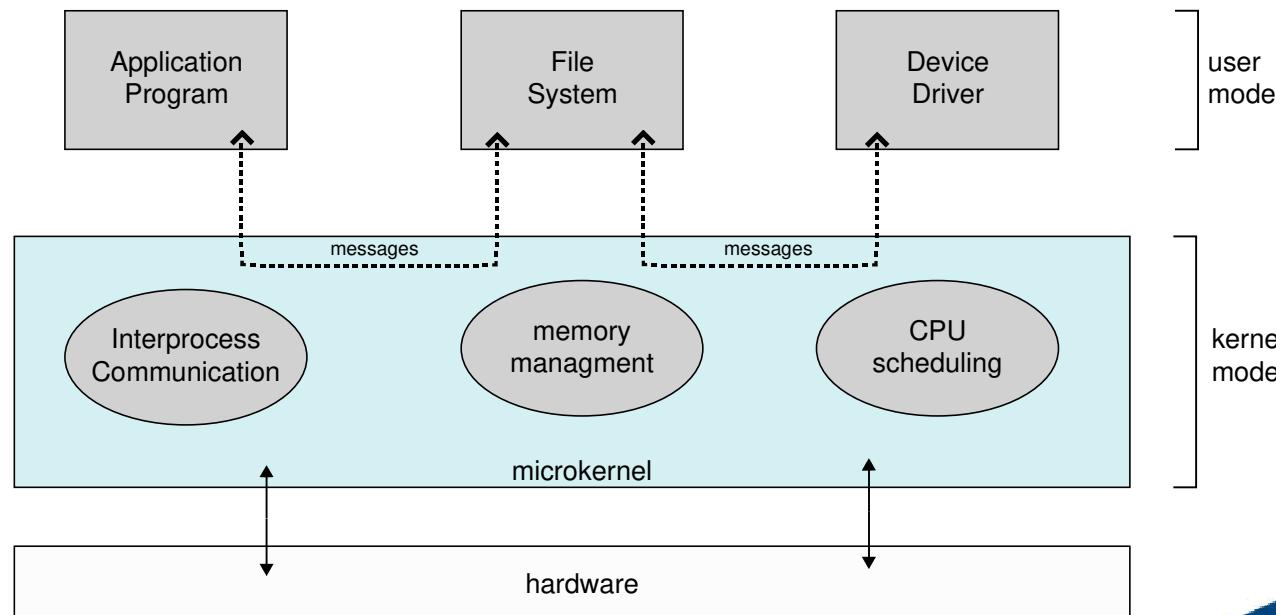
Hardware Abstraction Layer

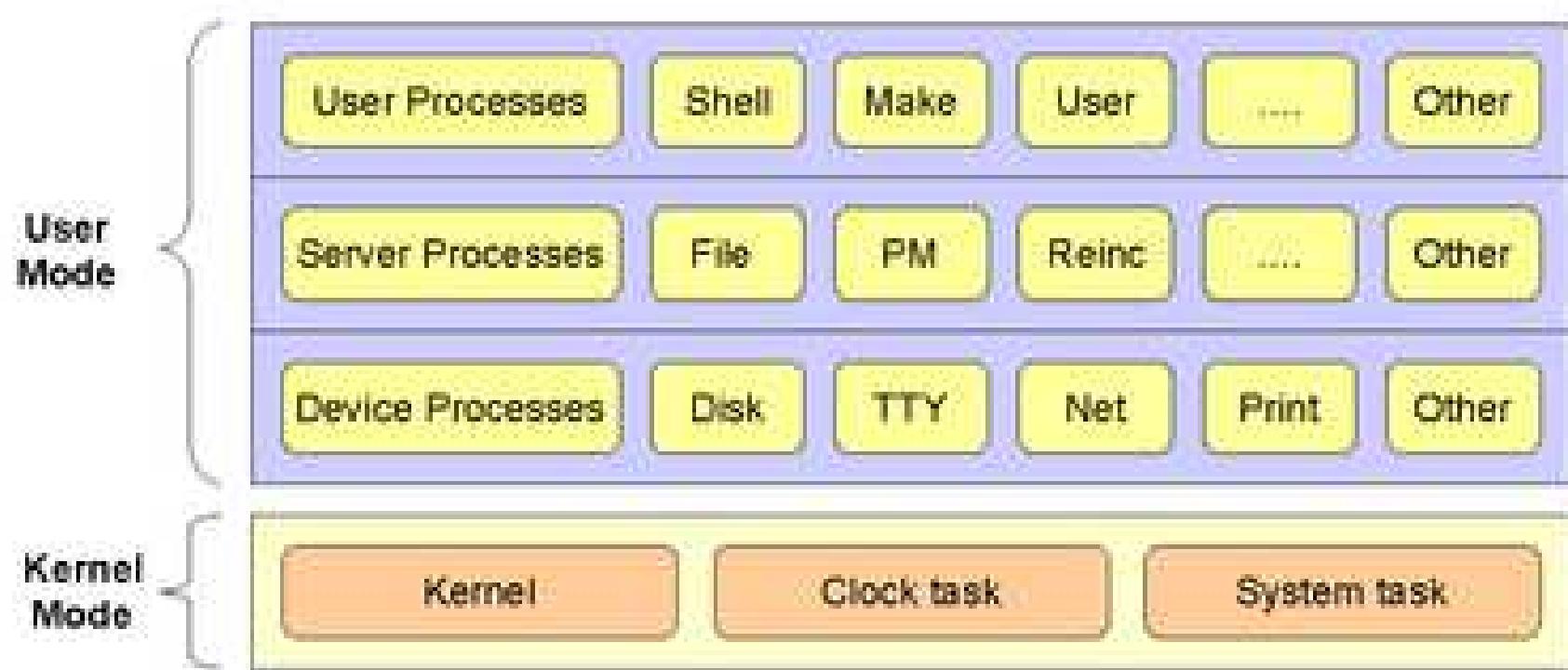
- An example of layering in modern operating systems
- Goal: separates hardware-specific routines from the “core” OS
 - Provides portability
 - Improves readability



Microkernel System Structure

- Moves as much from the kernel into user space
- **Mach:** example of **microkernel**
 - Mac OS X kernel (**Darwin**) partly based on Mach
- Communication takes place between user modules using **message passing**





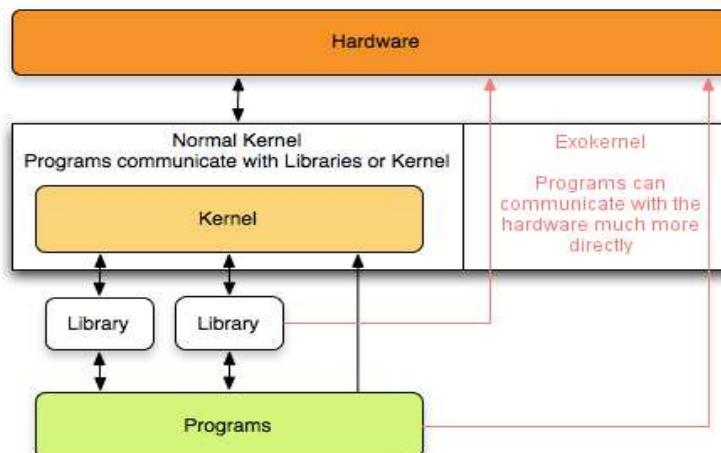
The MINIX 3 Microkernel Architecture

Microkernels: Pros and Cons

- Pros
 - Simplicity
 - Core kernel is very small
 - Extensibility
 - Can add new functionality in user-mode code
 - Reliability
 - OS services confined to user-mode programs
 - Cons
 - Poor performance
 - Message transfer operations instead of system call
- Tanenbaum–Torvalds debate
- https://en.wikipedia.org/wiki/Tanenbaum%20vs%20Torvalds_debate

Exokernel

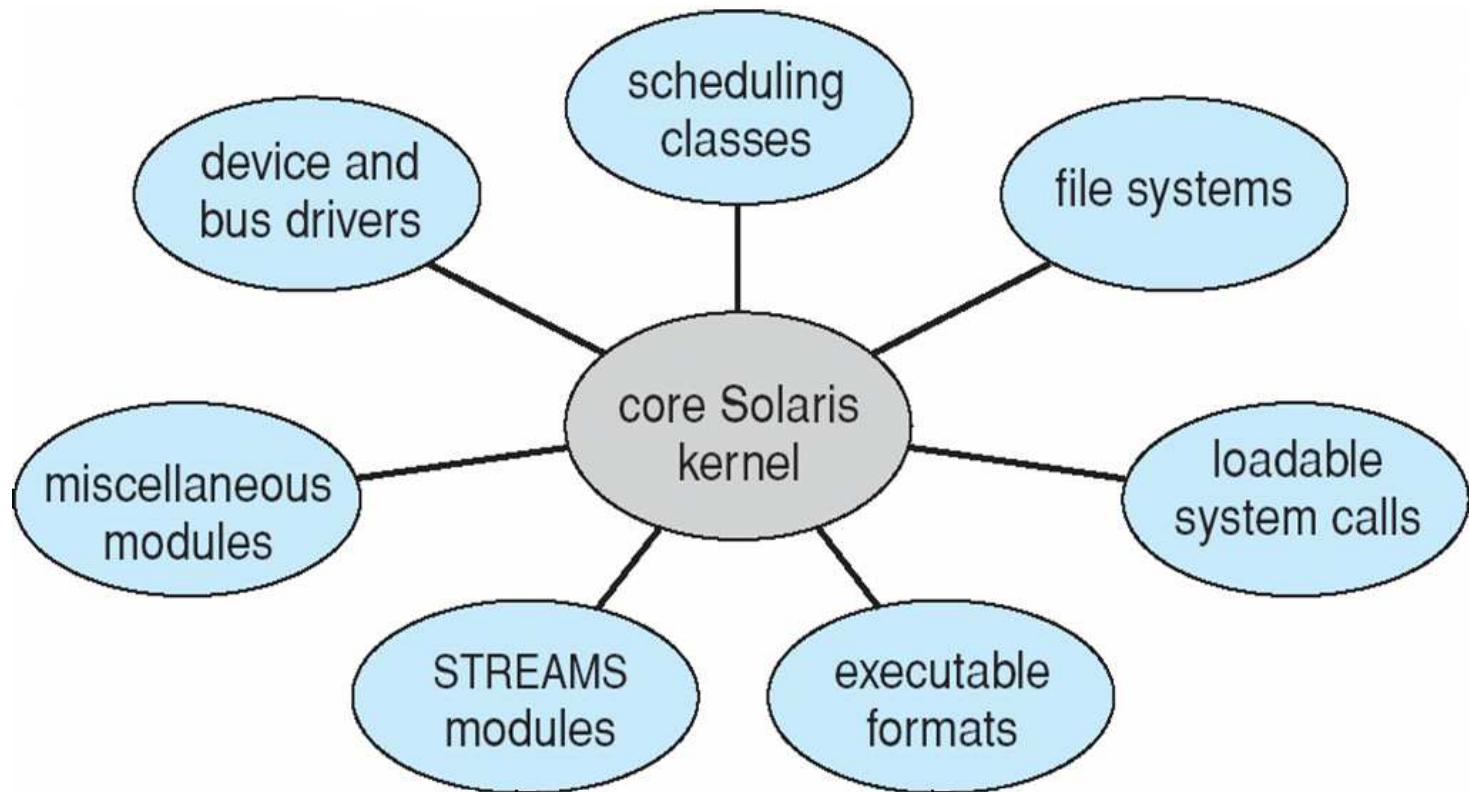
- Similar to microkernel in that only minimum functionality is in the kernel.
- Unlike the microkernel it exports hardware resources rather than emulating them.
- Physical resources are safely allocated to the application, where it can be managed.
- All abstractions are implemented in application level or as part of a library OS that is part of the application address space.
- Example: JOS



Modules

- Many modern operating systems implement **loadable kernel modules**
 - Uses object-oriented approach
 - Each core component is separate
 - Each talks to the others over known interfaces
 - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible
 - Linux, Solaris, etc

Solaris Modular Approach



Hybrid Systems

- Most modern operating systems are actually not one pure model
 - Hybrid combines multiple approaches to address performance, security, usability needs
 - Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
 - Windows mostly monolithic, plus microkernel for different subsystem ***personalities***
- Apple Mac OS X hybrid, layered, **Aqua** UI plus **Cocoa** programming environment
 - Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called **kernel extensions**)

Processes (1)

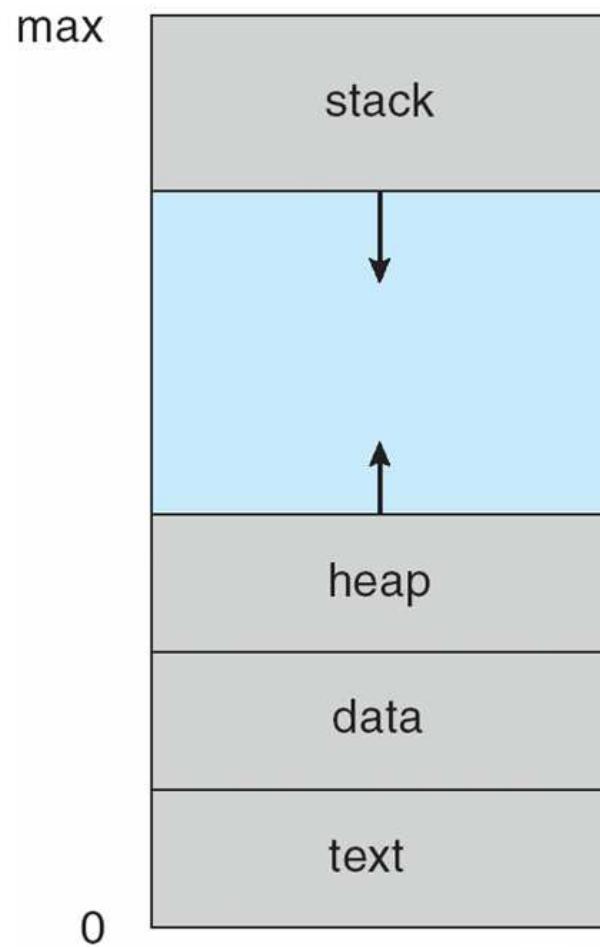
Dr. Jun Zheng
CSE325 Principles of Operating
Systems
8/30/2019



Process Concept

- An operating system executes a variety of programs:
 - Batch system – **jobs**
 - Time-shared systems – **user programs** or **tasks**
- Textbook uses the terms **job** and **process** almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion
- Multiple parts
 - The program code, also called **text section**
 - Current activity including **program counter**, processor registers
 - **Stack** containing temporary data
 - Function parameters, return addresses, local variables
 - **Data section** containing global variables
 - **Heap** containing memory dynamically allocated during run time

Process's Address Space (idealized)



Process Concept (Cont.)

- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
 - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
 - Consider multiple users executing the same program

Process Control Block (PCB)

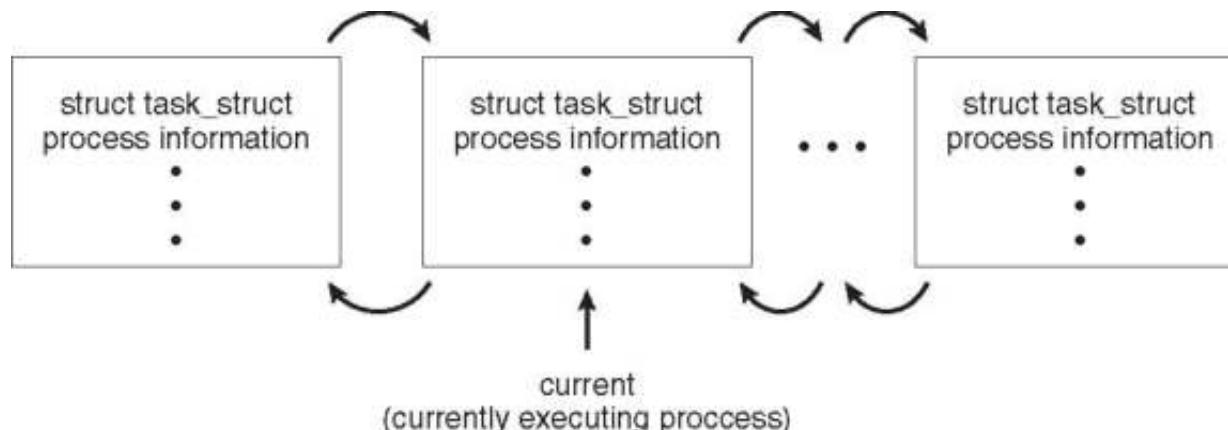
- ❑ Information associated with each process
(also called **task control block**)
- ❑ Process state – running, waiting, etc
- ❑ Program counter – location of instruction to next execute
- ❑ CPU registers – contents of all process-centric registers
- ❑ CPU scheduling information- priorities, scheduling queue pointers
- ❑ Memory-management information – memory allocated to the process
- ❑ Accounting information – CPU used, clock time elapsed since start, time limits
- ❑ I/O status information – I/O devices allocated to process, list of open files



Process Representation in Linux

⑩ Represented by the C structure task_struct

```
⑩ pid t_pid; /* process identifier */  
long state; /* state of the process */  
unsigned int time_slice /* scheduling information */  
struct task_struct *parent; /* this process's parent */  
struct list_head children; /* this process's children */  
struct files_struct *files; /* list of open files */  
struct mm_struct *mm; /* address space of this process */
```



Processes (2)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems

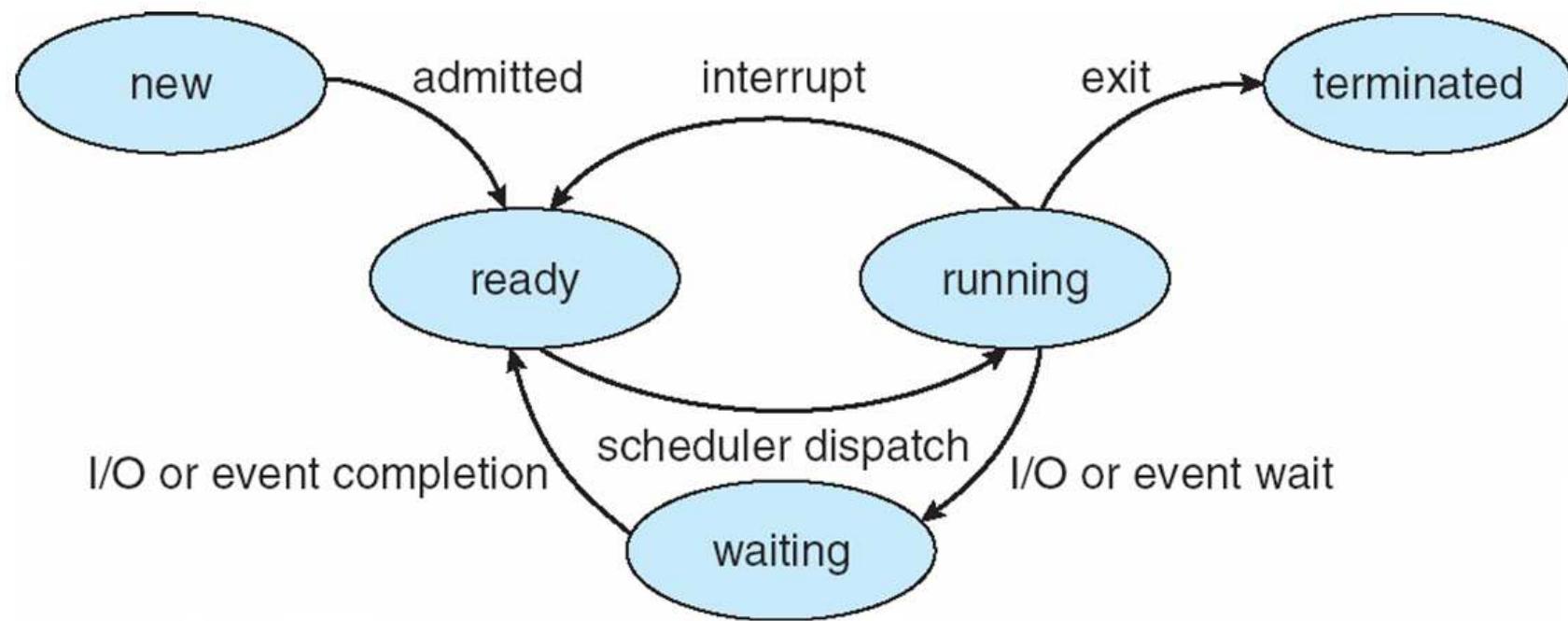
9/4/2019



Process State

- ❑ As a process executes, it changes **state**
 - ❑ **new**: The process is being created
 - ❑ **running**: Instructions are being executed
 - ❑ **waiting**: The process is waiting for some event to occur
 - ❑ **ready**: The process is waiting to be assigned to a processor
 - ❑ **terminated**: The process has finished execution

Diagram of Process State



Process Execution State

Example:

```
void main() {  
    printf("Hello World");  
}
```

State Sequence:

new

ready

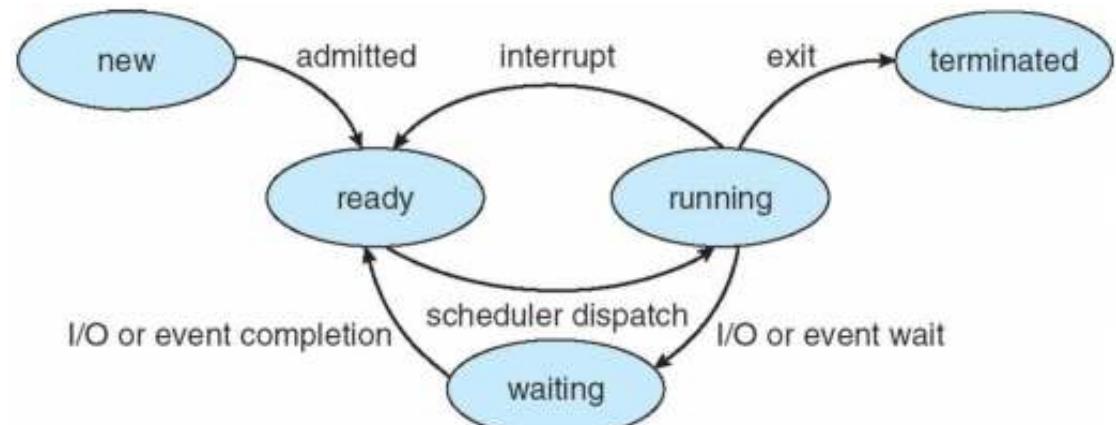
running

waiting (I/O)

ready

running

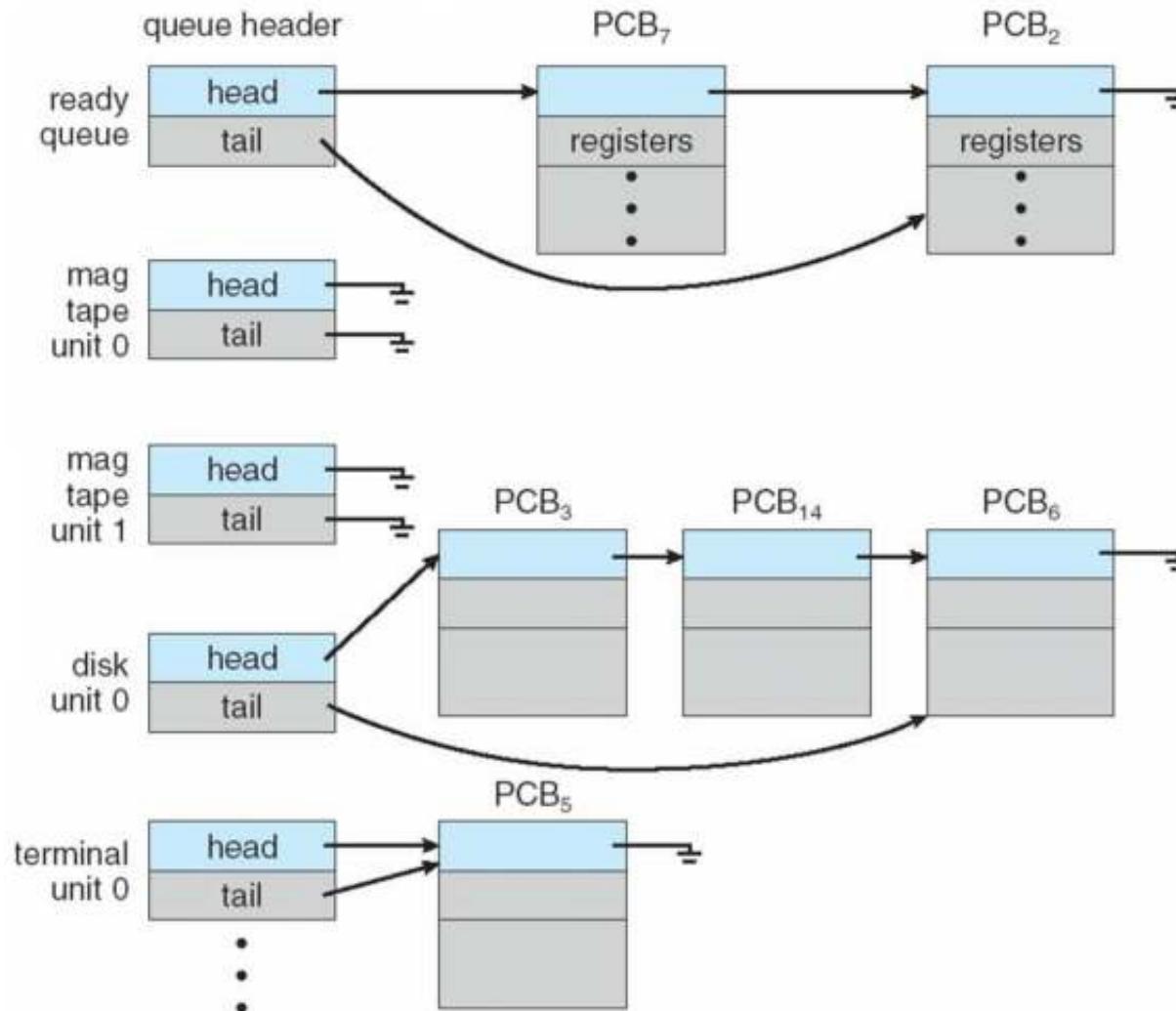
terminated



Process Scheduling

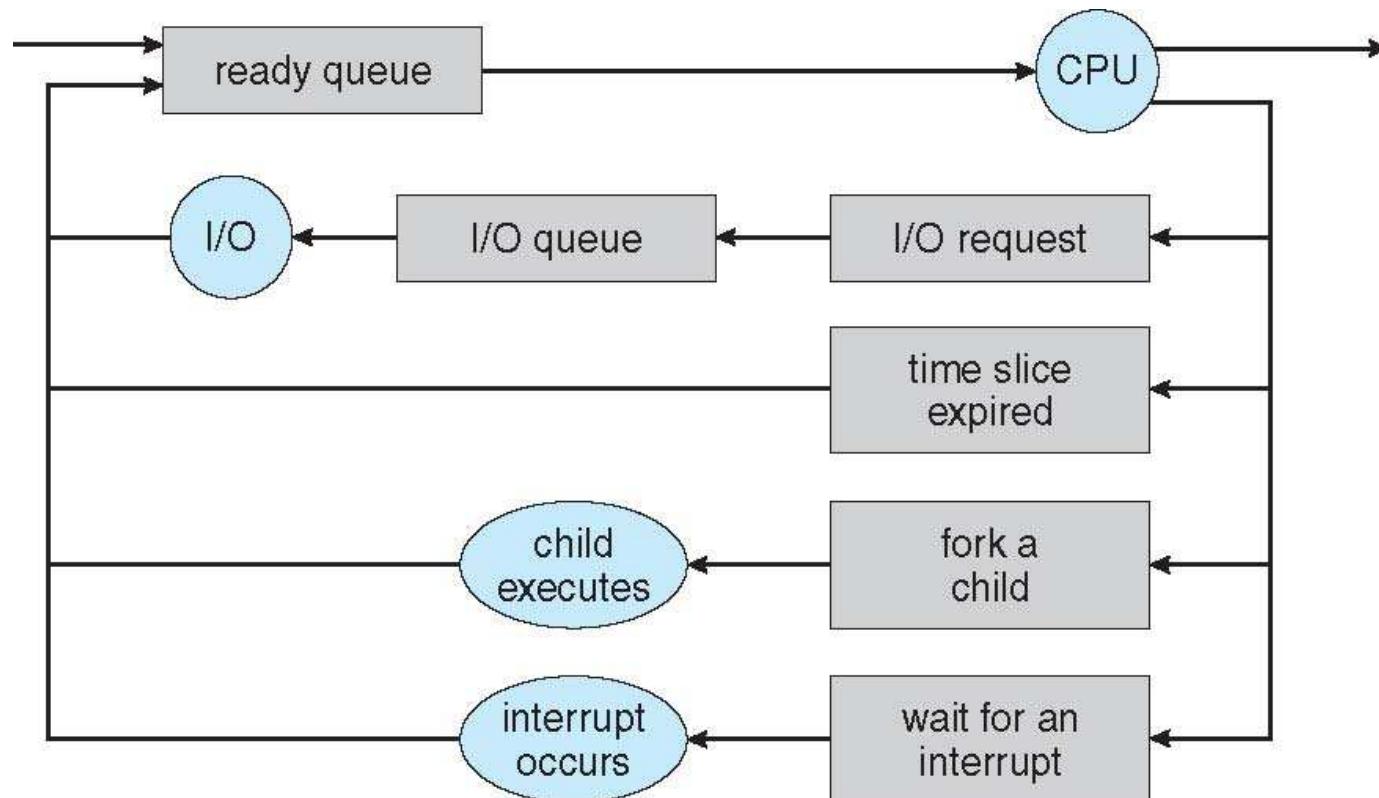
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
 - **Job queue** – set of all processes in the system
 - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
 - **Device queues** – set of processes waiting for an I/O device
 - Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling

- **Queueing diagram** represents queues, resources, flows

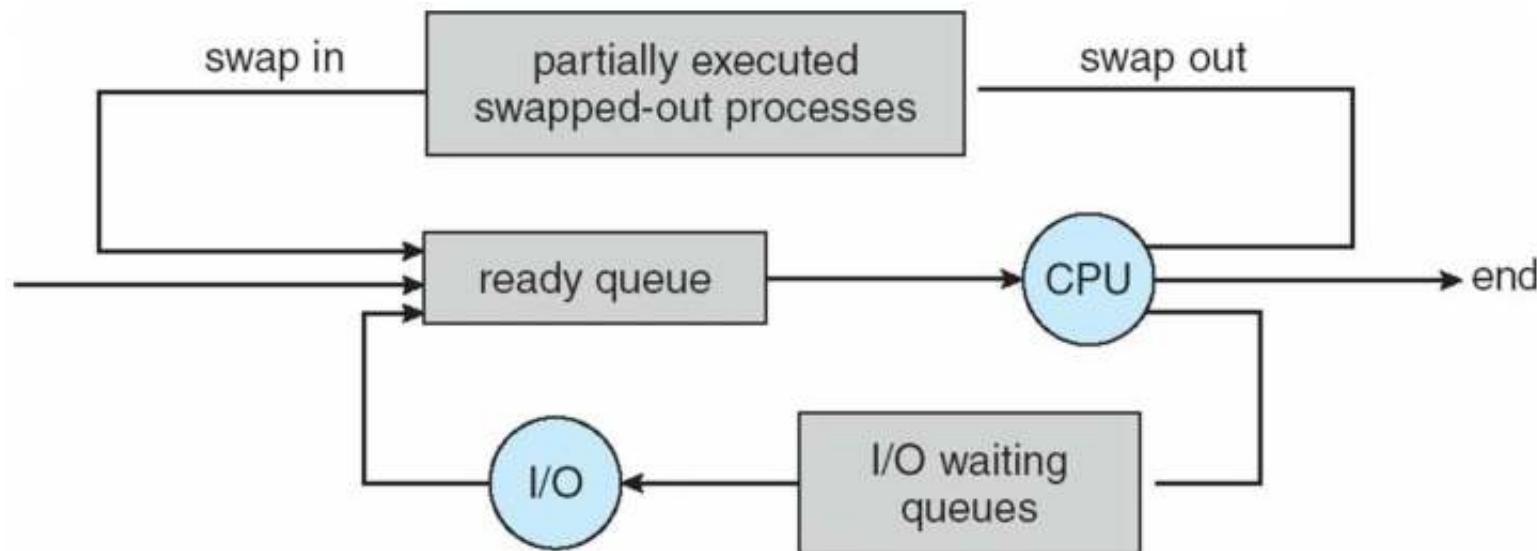


Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
 - Sometimes the only scheduler in a system
 - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
 - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
 - The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
 - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
 - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix***

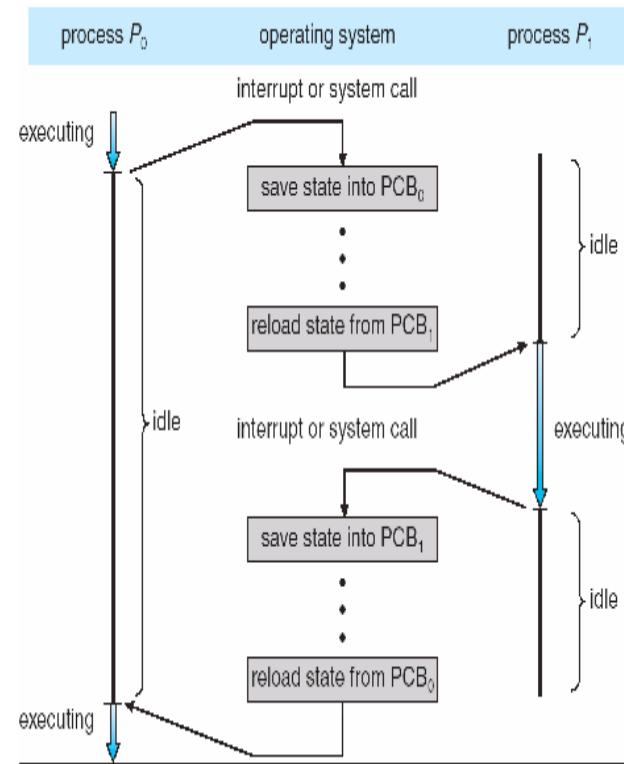
Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
 - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



Context Switch

- ❑ When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- ❑ **Context** of a process represented in the PCB
- ❑ Context-switch time is overhead; the system does no useful work while switching
 - ❑ The more complex the OS and the PCB → the longer the context switch
- ❑ Time dependent on hardware support
 - ❑ Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once



Operations on Processes

- ❑ System must provide mechanisms for:
 - ❑ process creation
 - ❑ process termination

Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier (pid)**
- Resource sharing options
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
- Execution options
 - Parent and children execute concurrently
 - Parent waits until children terminate

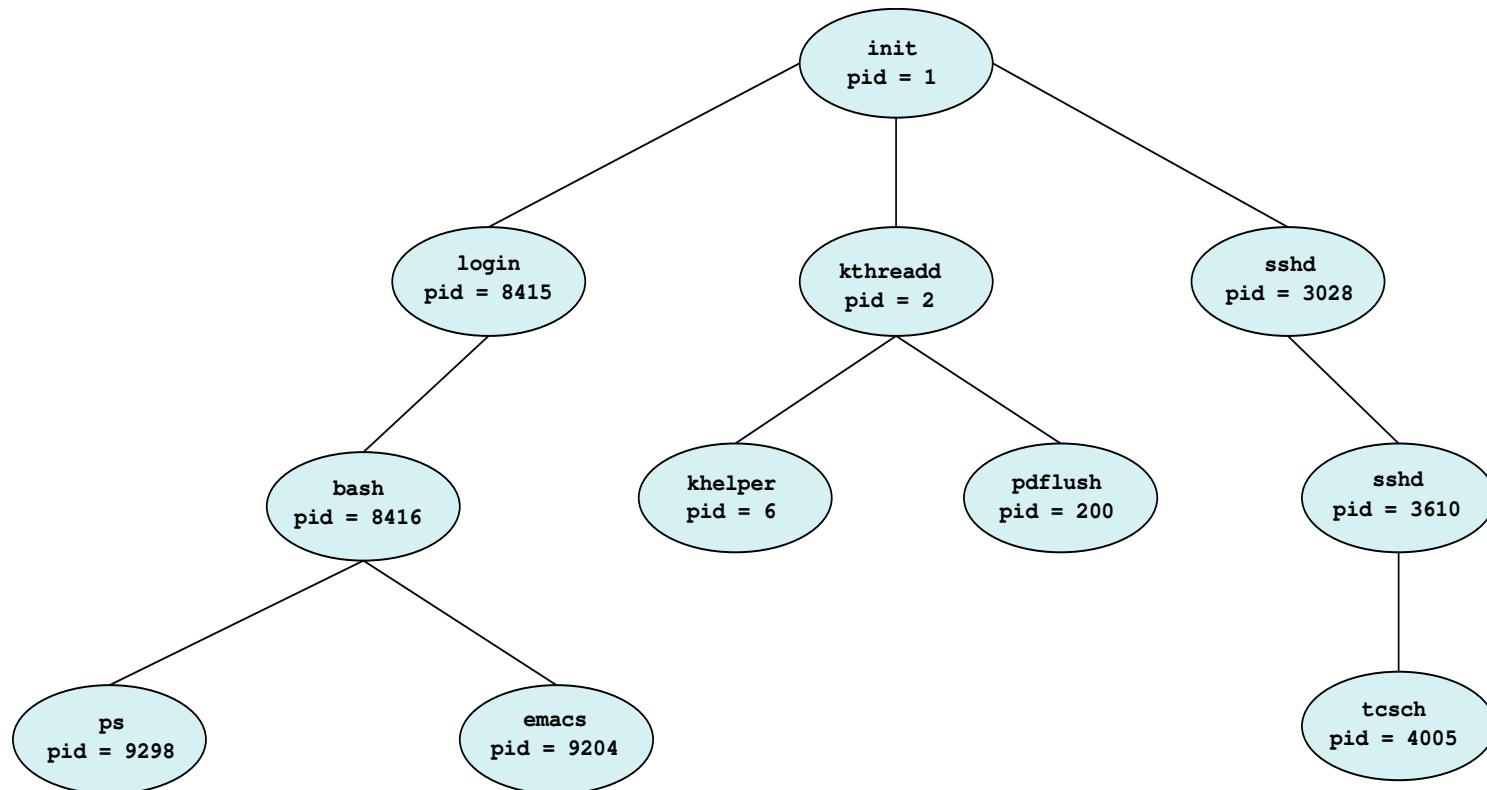
Processes (3)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems

9/6/2019

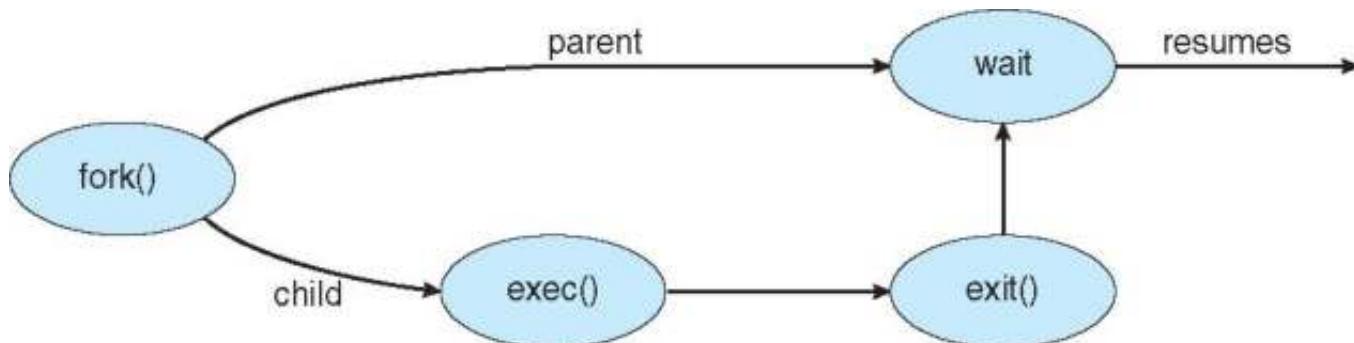


A Tree of Processes in Linux



Process Creation (Cont.)

- ❑ Address space
 - ❑ Child duplicate of parent
 - ❑ Child has a new program loaded into it
- ❑ UNIX examples
 - ❑ `fork()` system call creates new process
 - ❑ `exec()` system call used after a `fork()` to replace the process' memory space with a new program



Process Creation: Example

- When you log in to a machine running Unix, you create a shell process.
- Every command you type into the shell is a child of your shell process and is an implicit *fork* and *exec* pair.
- For example, you type emacs, the OS “*forks*” a new process and then “*exec*” (executes) emacs.
- If you type an & after the command, Unix will run the process in parallel with your shell, otherwise, your next shell command must wait until the first one completes.

C Program Forking Separate Process

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```

- In the parent process, fork returns the process id of the child
- In the child process, the return value is 0

Process Termination

- Process executes last statement and then asks the operating system to delete it using the `exit()` system call.
 - Returns status data from child to parent (via `wait()`)
 - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the `abort()` system call. Some reasons for doing so:
 - Child has exceeded allocated resources
 - Task assigned to child is no longer required
 - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Process Termination

- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
 - **cascading termination.** All children, grandchildren, etc. are terminated.
 - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **wait()** system call . The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

- If child process terminates but no parent waiting (did not invoke **wait()**), the child process is a **zombie**
- If parent terminated but child process remains running itself, the child process is an **orphan**

In-Class Work 1

How many processes are created by the following program including the parent process? What will be printed by the program? Assume children always print before their parent.

```
int main() {
    int i;
    int a = 3;
    if(fork() == 0) {
        a = a*4;
        if(fork() == 0)
            a = a - 2;
    }
    printf("%d\n", a);
}
```

Answer

3 processes

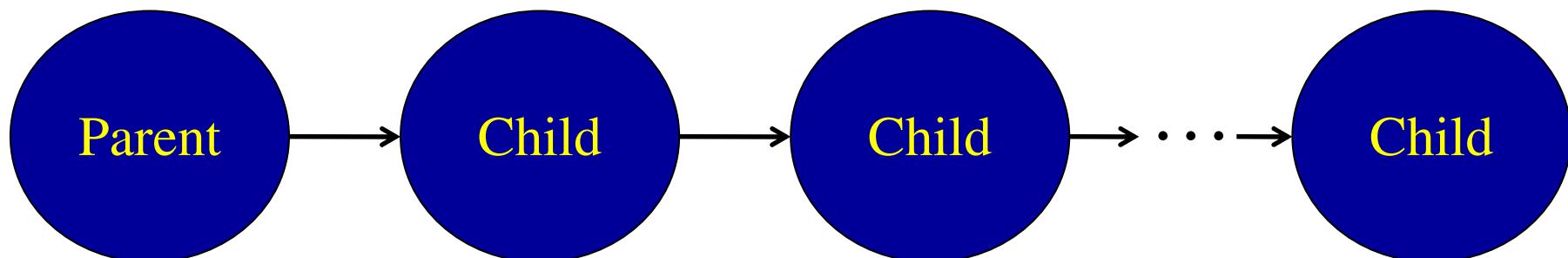
10

12

3

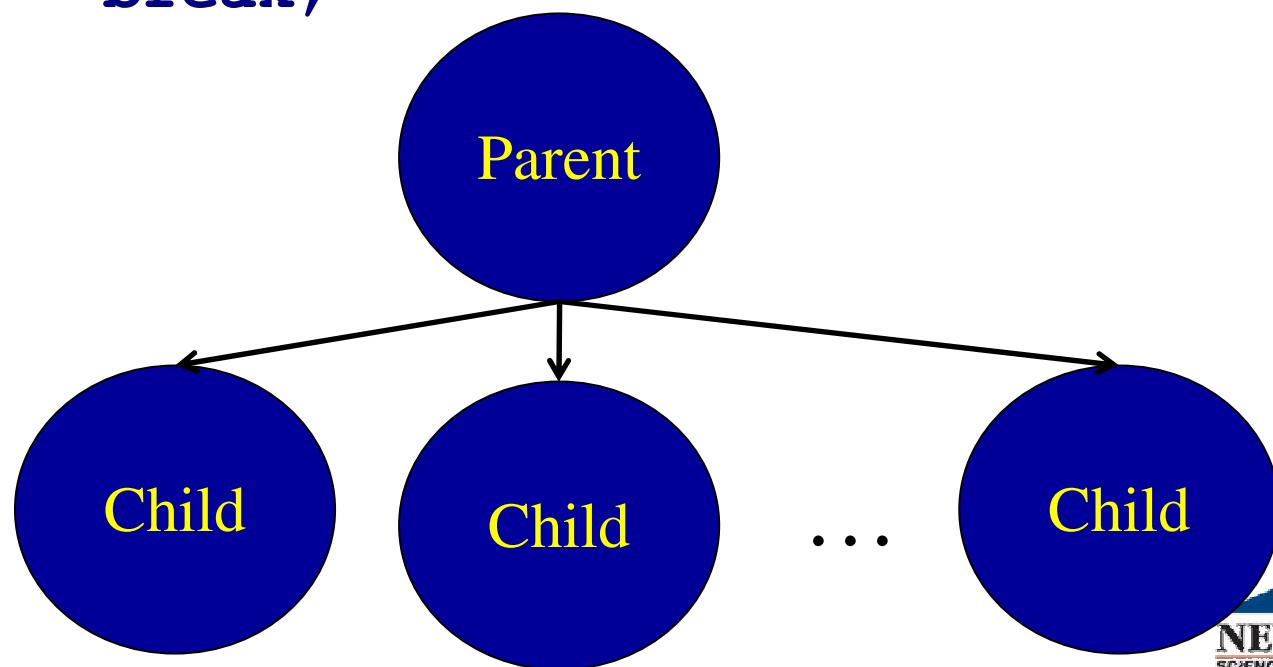
Process Chain

```
pid_t childpid = 0;  
  
for(i = 0; i < n; i++)  
    if(childpid = fork())  
        break;
```



Process Fan

```
pid_t childpid = 0;  
  
for(i = 0; i < n; i++)  
    if((childpid = fork()) <= 0)  
        break;
```



xv6 Process Management

In `sysproc.c`, `proc.c`

- ❑ `fork()` - create a new process,
- ❑ `exit()` - terminate current process
- ❑ `wait()` - wait for the child to exit
- ❑ `kill()` - set proc-killed to 1

In `proc.c`

- ❑ `sleep(chan)` - sleeps on a "channel", an address to name the condition we are sleeping on
- ❑ `wakeup(chan)` - wakeup all threads sleeping on chan (may be more than one)

In `sysfile.c`, `exec.c`

- ❑ `exec()` - replace memory of a current process with a memory image (of a program) loaded from a file

CPU Scheduling (1)

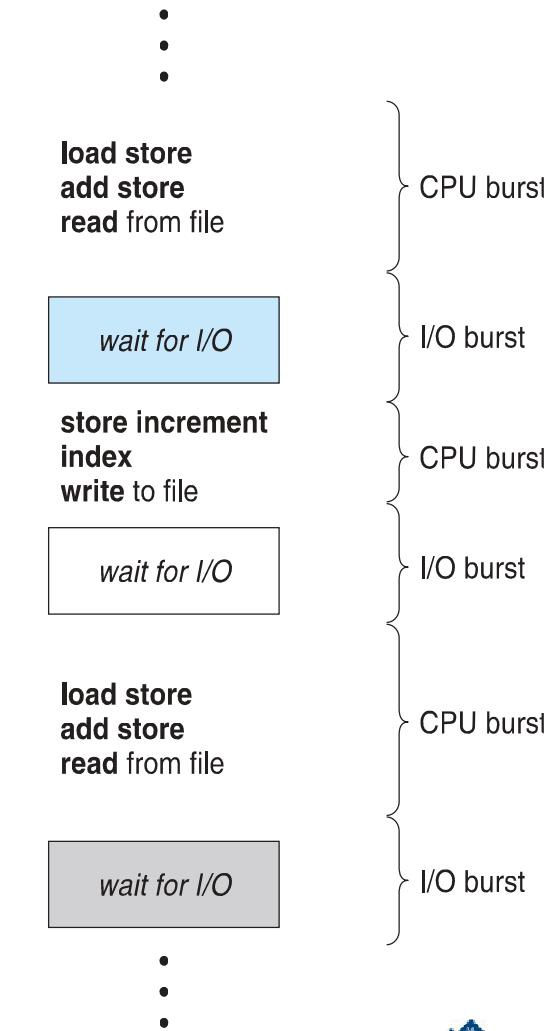
Dr. Jun Zheng
CSE325 Principles of Operating
Systems

9/9/2019

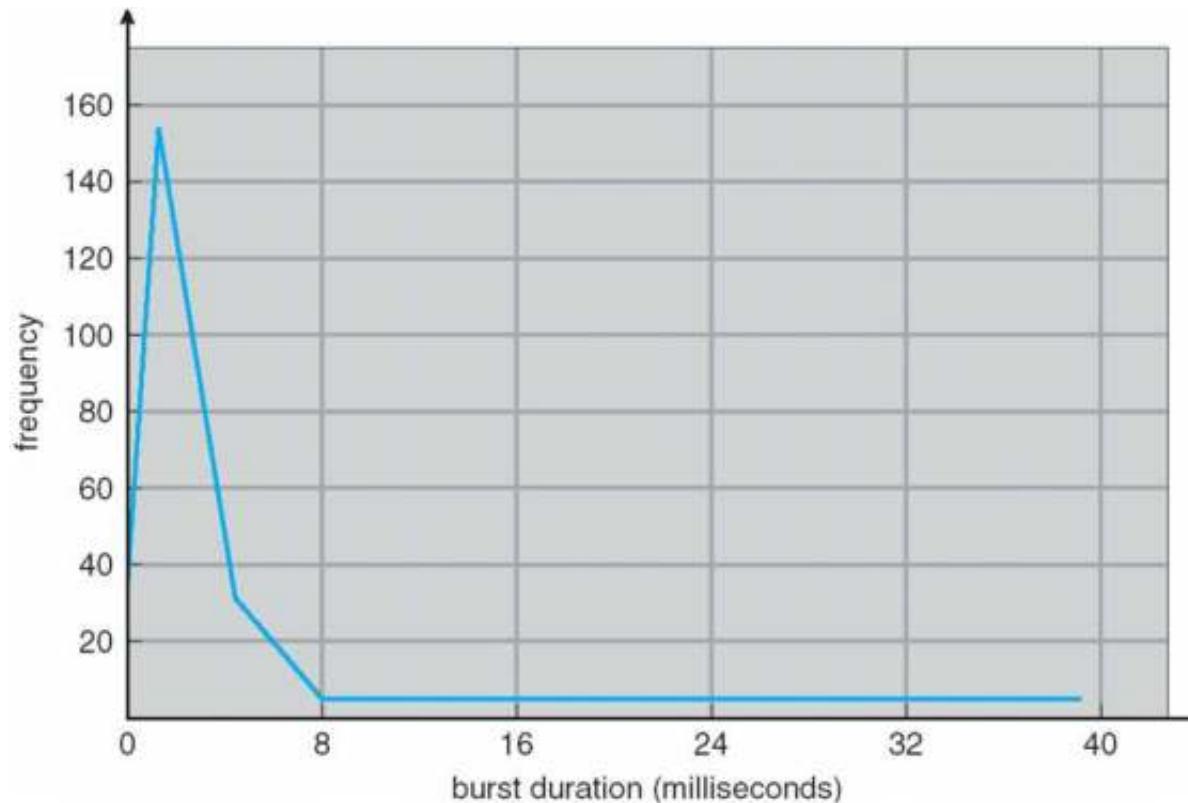


Basic Concepts

- ❑ Maximum CPU utilization obtained with multiprogramming
- ❑ 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



Histogram of CPU-burst Times



CPU Scheduler

- ❑ **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
 - ❑ Queue may be ordered in various ways
- ❑ CPU scheduling decisions may take place when a process:
 1. Switches from running to waiting state (e.g. I/O request)
 2. Switches from running to ready state (e.g. an interrupt)
 3. Switches from waiting to ready (e.g. completion of I/O)
 4. Terminates
- ❑ Scheduling under 1 and 4 is **nonpreemptive**
- ❑ All other scheduling is **preemptive**

Dispatcher

- Dispatcher module 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 it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Scheduling Algorithm Optimization Criteria

- ❑ Max CPU utilization
- ❑ Max throughput
- ❑ Min turnaround time
- ❑ Min waiting time
- ❑ Min response time

Gantt Chart

Illustrates how jobs are scheduled over time
on CPU

Example:



First- Come, First-Served (FCFS) Scheduling

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

- Suppose that the processes arrive in the order: P_1, P_2, P_3
The Gantt Chart for the schedule is:



- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2, P_3, P_1$$

- The Gantt chart for the schedule is:



- Waiting time for P₁ = 6; P₂ = 0; P₃ = 3
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- **Convoy effect** - short process behind long process
 - Consider one CPU-bound and many I/O-bound processes

Shortest-Job-First (SJF)

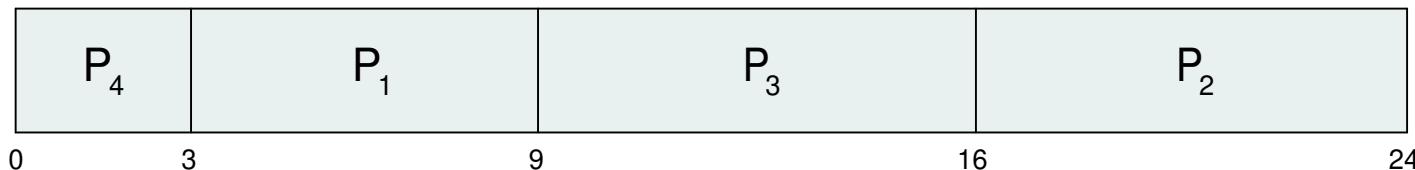
Scheduling

- Associate with each process the length of its next CPU burst
 - Use these lengths to schedule the process with the shortest time
- 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

Example of SJF

<u>Process</u>	<u>Burst Time</u>
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = $(3 + 16 + 9 + 0) / 4 = 7$

Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
 - Then pick process with shortest predicted next CPU burst
 1. t_n = actual length of n^{th} CPU burst
 2. τ_{n+1} = predicted value for the next CPU burst
 3. $\alpha, 0 \leq \alpha \leq 1$
 4. Define: $\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$.
 - Can be done by using the length of previous CPU bursts, using exponential averaging
 - Commonly, α set to $1/2$
 - Preemptive version called **shortest-remaining-time-first**

CPU Scheduling (2)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems

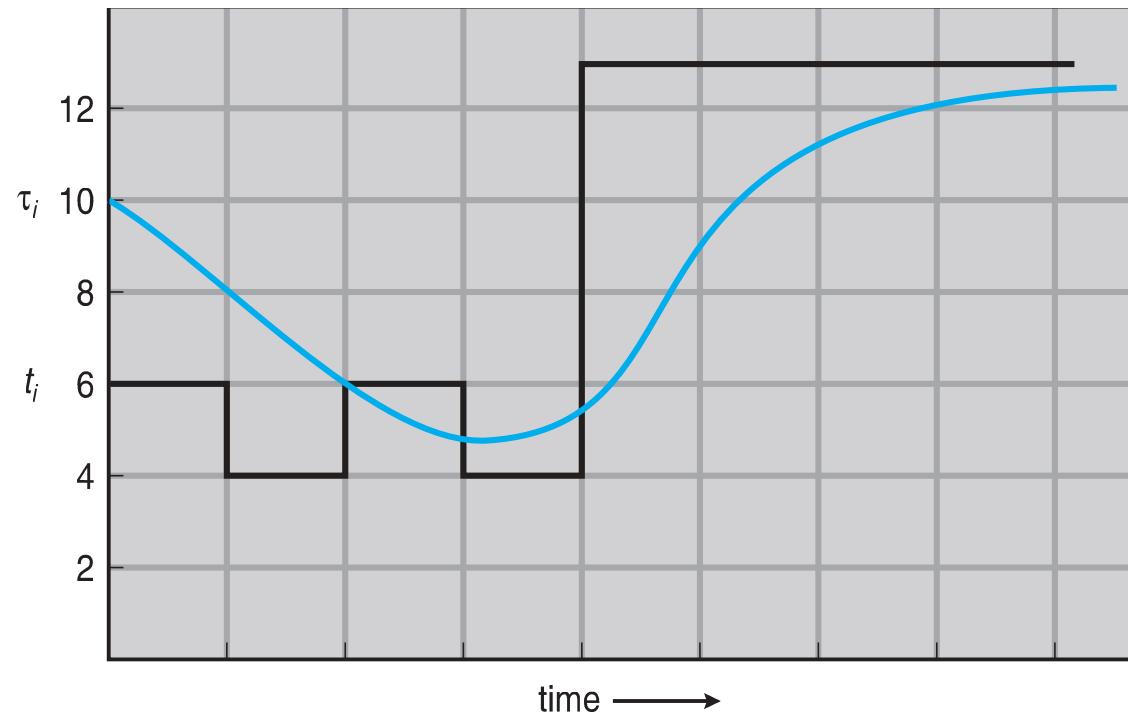
9/11/2019



Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
 - Then pick process with shortest predicted next CPU burst
 1. t_n = actual length of n^{th} CPU burst
 2. τ_{n+1} = predicted value for the next CPU burst
 3. $\alpha, 0 \leq \alpha \leq 1$
 4. Define: $\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$.
 - Can be done by using the length of previous CPU bursts, using exponential averaging
 - Commonly, α set to $1/2$
 - Preemptive version called **shortest-remaining-time-first**

Prediction of the Length of the Next CPU Burst



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

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:

$$\begin{aligned}\tau_{n+1} &= \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \dots \\ &\quad + (1 - \alpha)^j \alpha t_{n-j} + \dots \\ &\quad + (1 - \alpha)^{n+1} \tau_0\end{aligned}$$

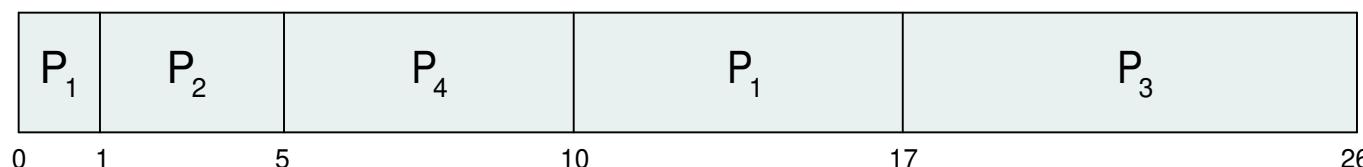
- ❑ Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive SJF Gantt Chart*



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

Priority Scheduling

- ❑ A priority number (integer) is associated with each process
- ❑ The CPU is allocated to the process with the highest priority
 - ❑ Preemptive
 - ❑ Nonpreemptive
- ❑ 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	10	3
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

Round Robin (RR)

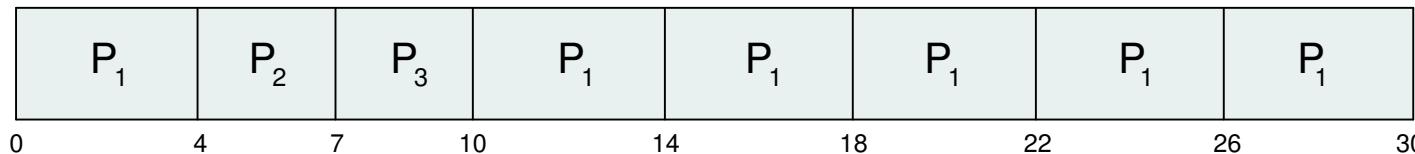
- Each process gets a small unit of CPU time (**time quantum** q), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q , then each process gets $1/n$ of the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

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

Average Waiting Time

$$P_1: (0-0) + (10 - 4) = 6$$

$$P_2: (4-0) = 4$$

$$P_3: (7 - 0) = 7$$

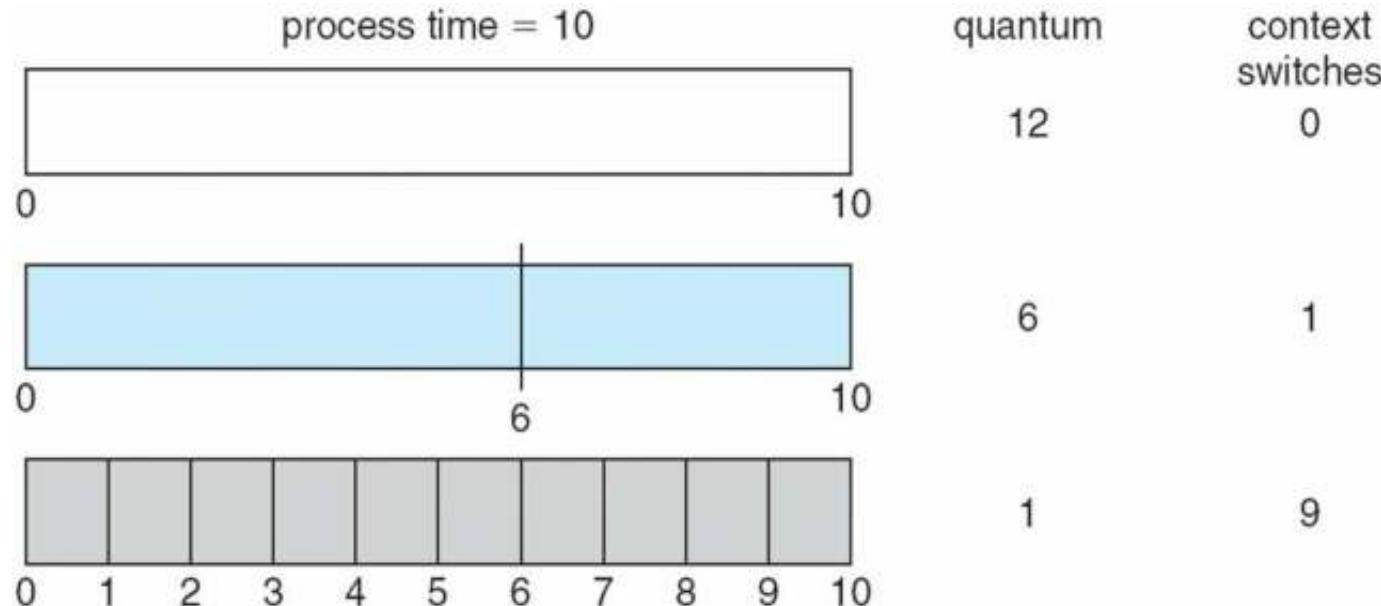
$$\text{Avg. waiting time: } (6 + 4 + 7)/3 = 17/3 \text{ ms}$$

CPU Scheduling (3)

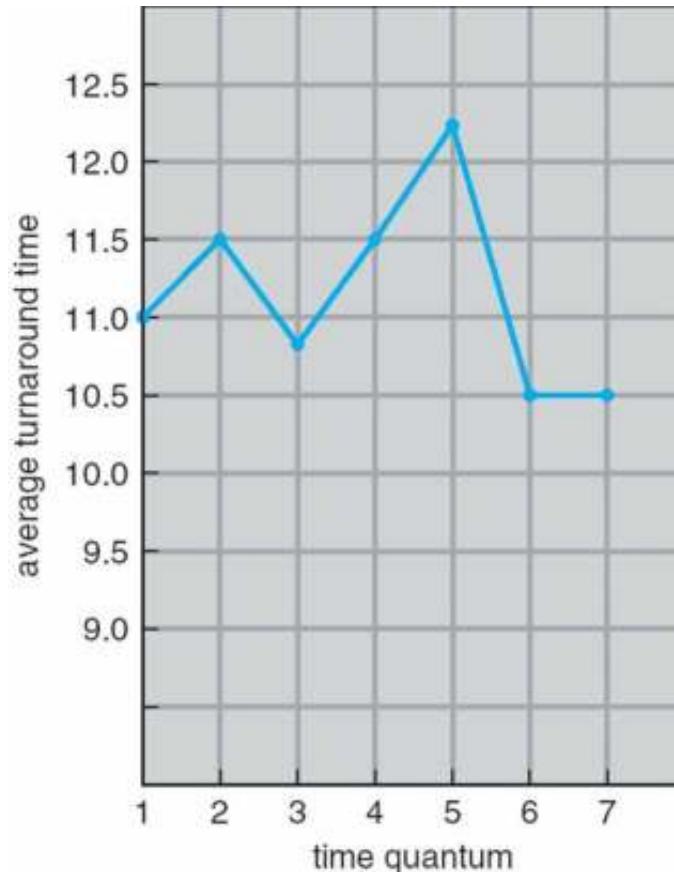
Dr. Jun Zheng
CSE325 Principles of Operating
Systems
9/13/2019



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 q

In-Class Work 2

Assume we have workload shown below. All five process arrive at time 0, in the given order.

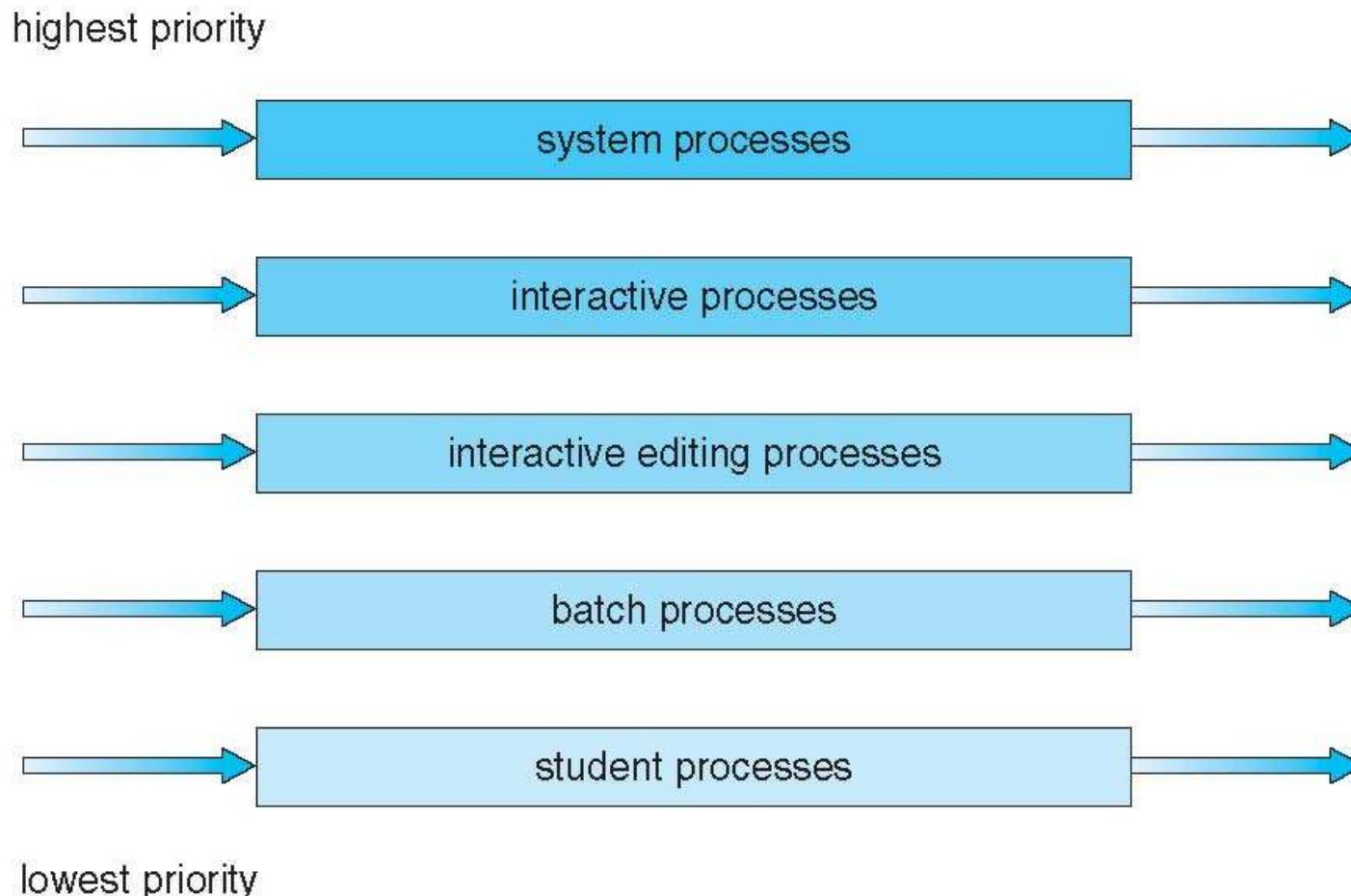
<u>Process</u>	<u>Burst Time (ms)</u>
P1	10
P2	29
P3	3
P4	7
P5	12

Consider the FCFS, SJF, and RR (quantum = 10 ms) scheduling algorithm for this set of processes. Which algorithm would give the minimum average waiting time? (use Gantt chart to solve this problem)

Multilevel Queue

- ❑ Ready queue is partitioned into separate queues, eg:
 - ❑ **foreground** (interactive)
 - ❑ **background** (batch)
- ❑ Process permanently in a given queue
- ❑ Each queue has its own scheduling algorithm:
 - ❑ foreground – RR
 - ❑ background – FCFS
- ❑ Scheduling must be done between the queues:
 - ❑ Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
 - ❑ Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR, 20% to background in FCFS

Multilevel Queue Scheduling



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

CPU Scheduling (4)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems

9/16/2019



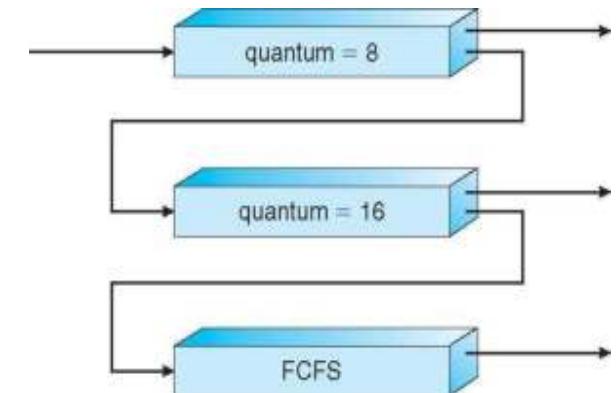
Example of Multilevel Feedback Queue

Three queues:

- Q_0 – RR with time quantum 8 milliseconds
- Q_1 – RR time quantum 16 milliseconds
- Q_2 – FCFS

Scheduling

- A new job enters queue Q_0 which is served FCFS
 - 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 is again served FCFS and receives 16 additional milliseconds
 - If it still does not complete, it is preempted and moved to queue Q_2



Question

- Many CPU-scheduling algorithms are parameterized. For example, the RR algorithm requires a parameter to indicate the time slice. Multilevel feedback queues require parameters to define the number of queues, the scheduling algorithms for each queue, the criteria used to move processes between queues, and so on. These algorithms are thus really sets of algorithms (for example, the set of RR algorithms for all time slices, and so on). One set of algorithms may include another (for example, the FCFS algorithm is the RR algorithm with an infinite time quantum). What (if any) relation holds between the following pairs of algorithm sets?
 - Priority and SJF
 - Multilevel feedback queues and FCFS
 - Priority and FCFS
 - RR and SJF

Answer

- The shortest job has the highest priority.
- The lowest level of MLFQ is FCFS.
- FCFS gives the highest priority to the job having been in existence the longest.
- None.

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
 - Currently, most common
- **Processor affinity** – process has affinity for processor on which it is currently running
 - **soft affinity**
 - **hard affinity**
 - Variations including **processor sets**

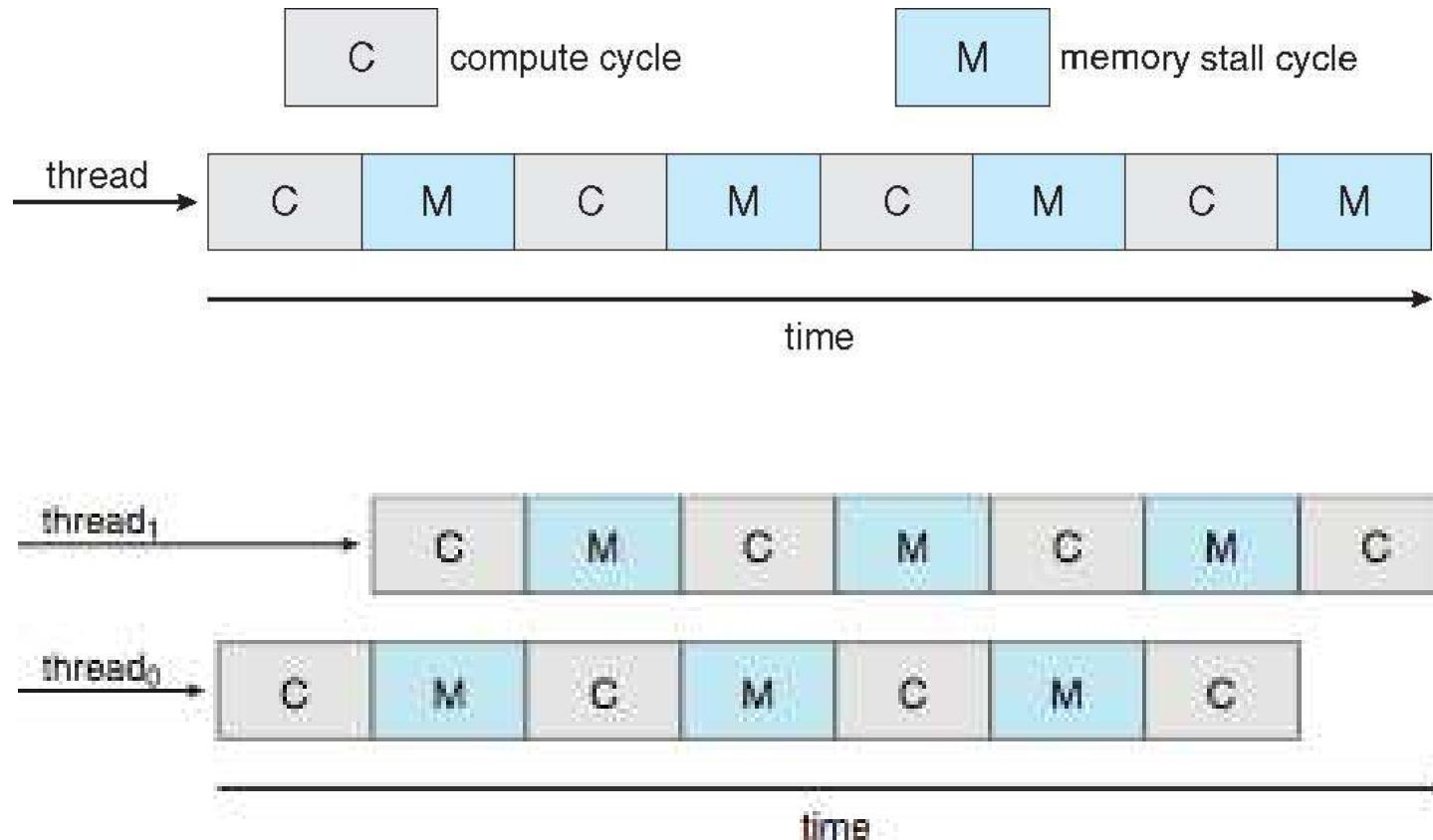
Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor

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

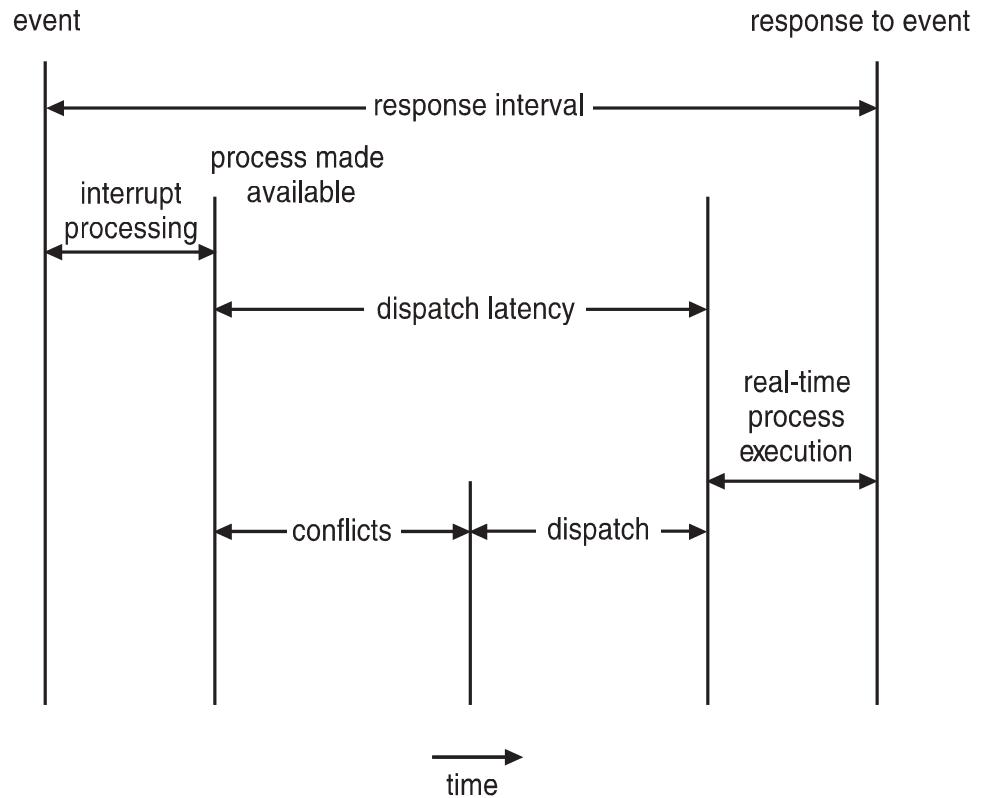


Real-Time CPU Scheduling

- ❑ Can present obvious challenges
- ❑ **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
- ❑ **Hard real-time systems** – task must be serviced by its deadline
- ❑ Two types of latencies affect performance
 - ❑ Interrupt latency – time from arrival of interrupt to start of routine that services interrupt
 - ❑ Dispatch latency – time for schedule to take current process off CPU and switch to another

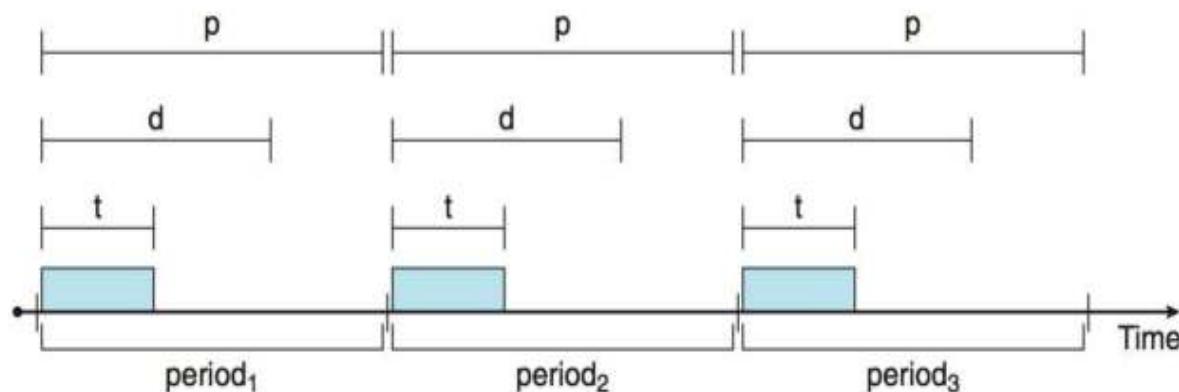
Real-Time CPU Scheduling (Cont.)

- ❑ Conflict phase of dispatch latency:
 - ❑ Preemption of any process running in kernel mode
 - ❑ Release by low-priority process of resources needed by high-priority processes



Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
- But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** - processes require CPU at constant intervals
- Has processing time t , deadline d , period p
- $0 \leq t \leq d \leq p$
- **Rate** of periodic task is $1/p$

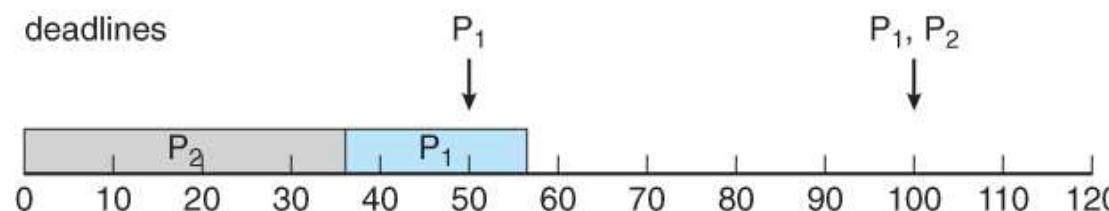


Rate Monotonic Scheduling

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority

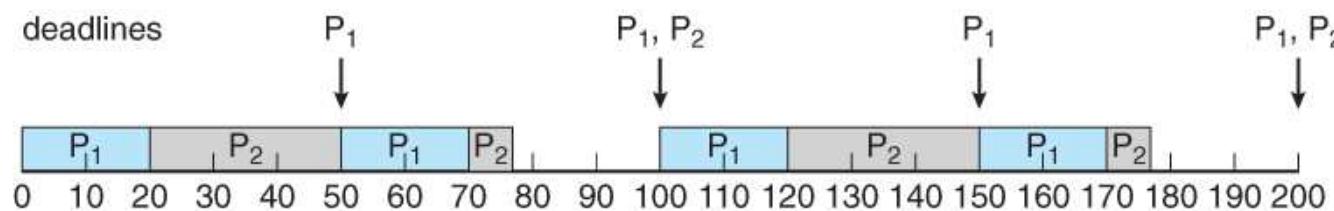
Example

- ❑ Suppose that process P₁ has a period of 50, an execution time of 20, and a deadline that matches its period (50).
- ❑ Similarly suppose that process P₂ has period 100, execution time of 35, and deadline of 100.
- ❑ The total CPU utilization time is $20 / 50 = 0.4$ for P₁, and $35 / 100 = 0.35$ for P₂, or 0.75 (75%) overall.
- ❑ However if P₂ is allowed to go first (FCFS), then P₁ cannot complete before its deadline:

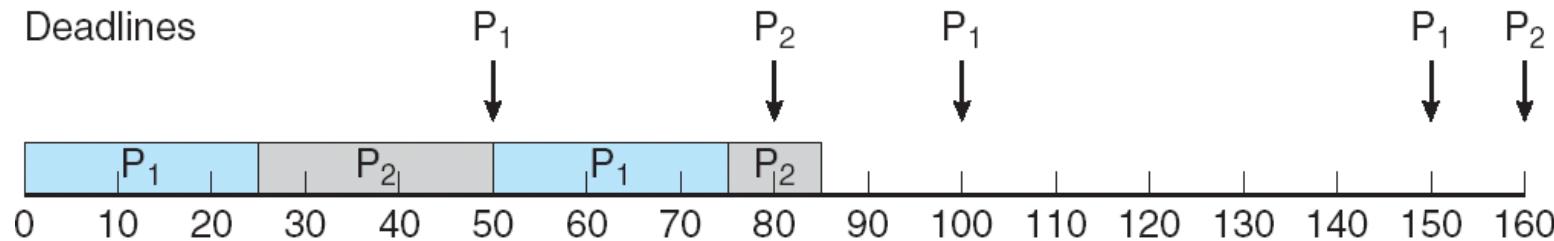


Example

- ❑ if P₁ is given higher priority, it gets to go first, and P₂ starts after P₁ completes its burst.
- ❑ At time 50 when the next period for P₁ starts, P₂ has only completed 30 of its 35 needed time units, but it gets pre-empted by P₁.
- ❑ At time 70, P₁ completes its task for its second period, and the P₂ is allowed to complete its last 5 time units.
- ❑ Overall both processes complete at time 75, and the cpu is then idle for 25 time units, before the process repeats.



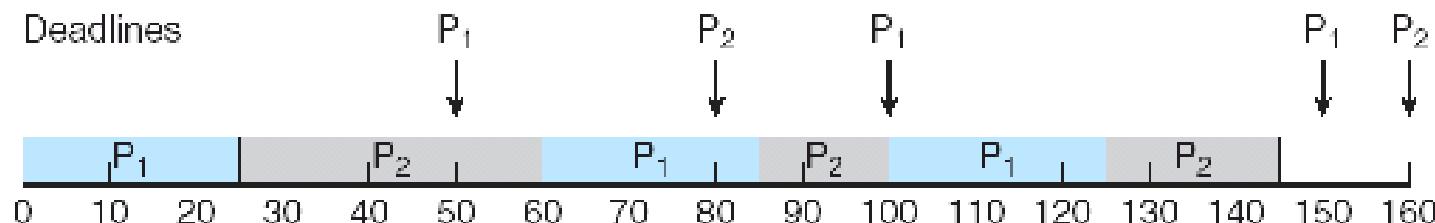
Missed Deadlines with Rate Monotonic Scheduling



$P_1 = 50$, $T_1 = 25$, $P_2 = 80$, $T_2 = 35$, and the deadlines match the periods.

Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:
 - the earlier the deadline, the higher the priority;
 - the later the deadline, the lower the priority
- Previous example:



$P_1 = 50$, $T_1 = 25$, $P_2 = 80$, $T_2 = 35$, and the deadlines match the periods.

Proportional Share Scheduling

- T shares are allocated among all processes in the system
- An application receives N shares where $N < T$
- This ensures each application will receive N / T of the total processor time
- Must work in conjunction with an admission-control policy to guarantee that an application receives its allocated share of time.

Threads

Dr. Jun Zheng
CSE325 Principles of Operating
Systems

9/18/2019



Processes

- ❑ A process includes many things
 - ❑ An address space (defining all the code and data pages)
 - ❑ OS resources (e.g., open files) and accounting information
 - ❑ Execution state (PC, SP, regs, etc.)
- ❑ Creating a new process is costly because of all of the data structures that must be allocated and initialized
 - ❑ Recall **struct task_struct** in Linux
 - ❑ ...which does not even include page tables, perhaps TLB flushing, etc.
- ❑ Communicating between processes is costly because most communication goes through the OS
 - ❑ Overhead of system calls and copying data

Multi-programming

- To execute parallel programs we need to
 - Create several processes that execute in parallel
 - Cause each to map to the same address space to share data
 - They are all part of the same computation
 - Have the OS schedule these processes in parallel
- This situation is **very inefficient**
 - **Space**: PCB, page tables, etc.
 - **Time**: create data structures, fork and copy addr space, etc.
- Solutions: possible to have more **efficient**, yet **cooperative** “processes”?

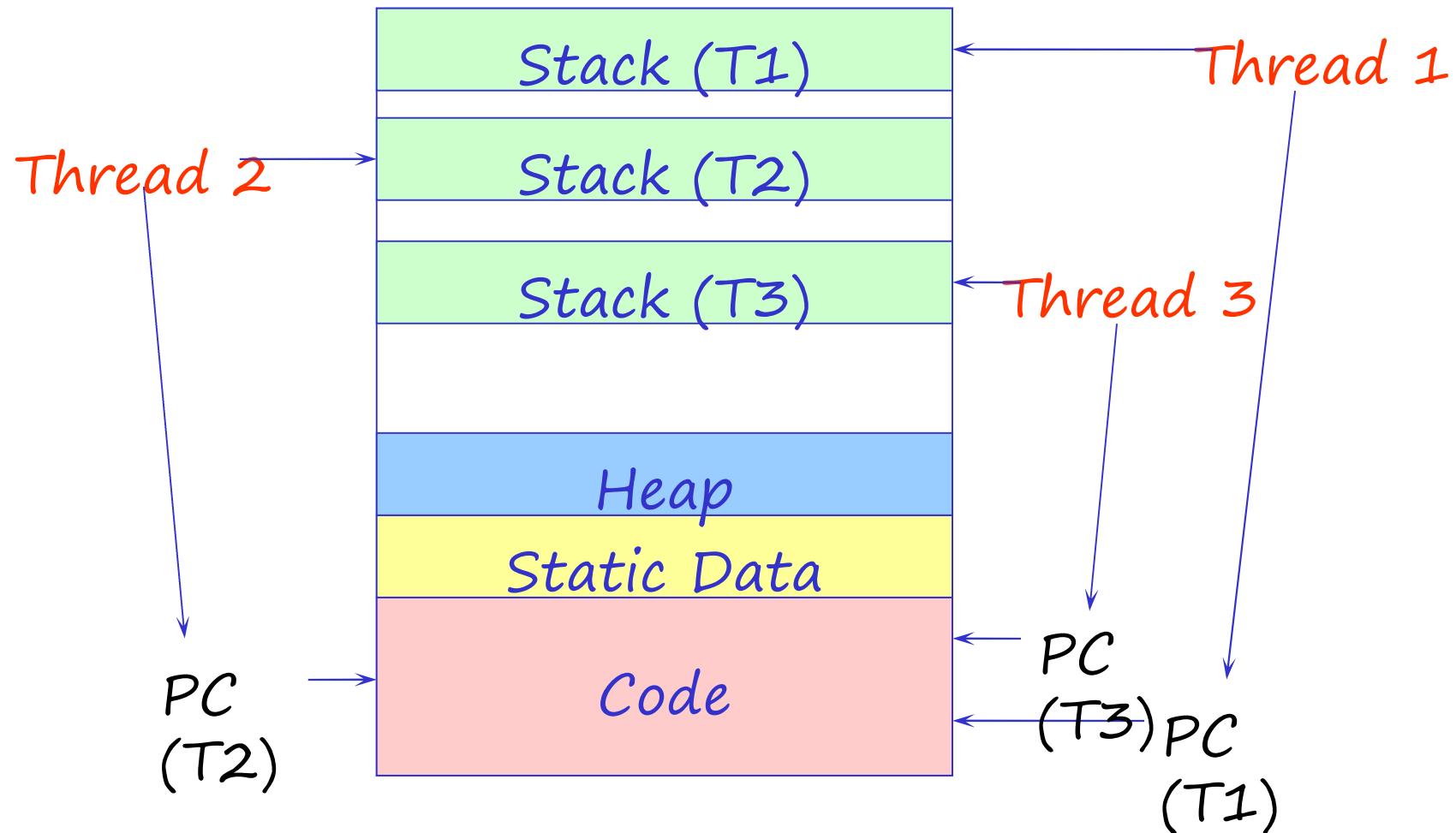
Rethinking Processes

- ❑ What is similar in these cooperating processes?
 - ❑ They all share the same code and data (address space)
 - ❑ They all share the same privileges
 - ❑ They all share the same resources (files, sockets, etc.)
- ❑ What don't they share?
 - ❑ Each has its own execution state: PC, SP, and registers
- ❑ **Key idea:** Why don't we separate the concept of a process from its execution state?
 - ❑ **Process:** address space, privileges, resources, etc.
 - ❑ **Execution state:** PC, SP, registers
- ❑ Exec state also called **thread of control**, or **thread**

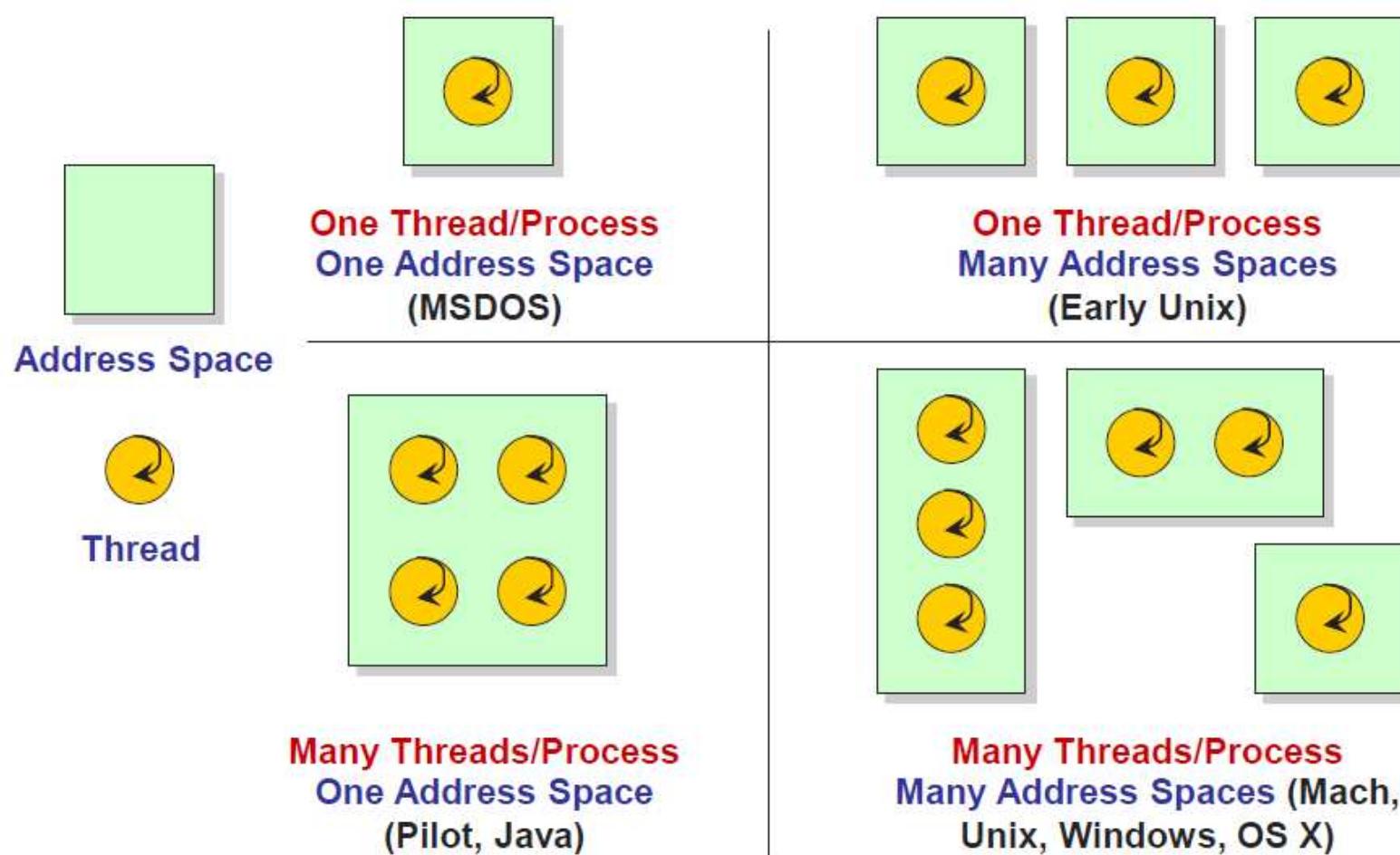
Threads

- Modern OSes (Mac, Windows, modern Unix) separate the concepts of processes and threads
 - The **thread** defines a sequential execution stream within a process (PC, SP, registers)
 - The **process** defines the address space and general process attributes (everything but threads of execution)
- A thread is bound to a single process
 - Processes, however, can have **multiple** threads
- Threads become the unit of scheduling
 - Processes are now the **containers** in which threads execute

Threads in a Process



Thread Design Space



Threads (2)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems
9/20/2019



Threads: Concurrent Servers

- Using **fork()** to create new processes to handle requests in parallel is overkill for such a simple task
- Web server example:

```
- while (1) {  
-     int sock = accept();  
-     if ((child_pid = fork()) == 0) {  
-         Handle client request  
-         Close socket and exit  
-     } else {  
-         Close socket  
-     }  
- }
```

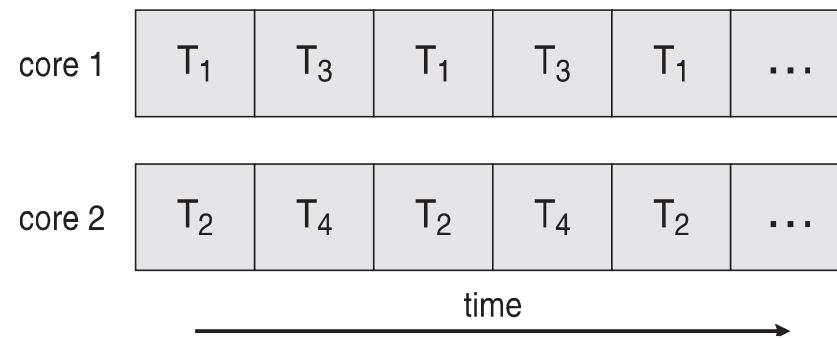
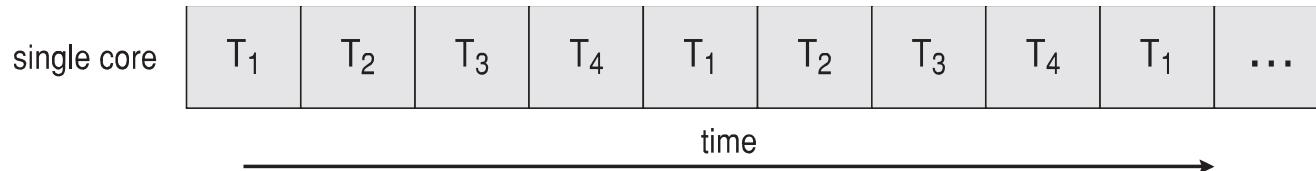
Threads: Concurrent Servers

- Instead, we can create a new thread for each request

```
⑩    web_server() {  
    - while (1) {  
    -     int sock = accept();  
    -     thread_create(handle_request, sock);  
    - }  
⑩    }  
  
⑩    handle_request(int sock) {  
⑩        Process request  
⑩        close(sock);  
⑩    }
```



Concurrency vs. Parallelism



Benefits of Multithreaded Programming

- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multiprocessor architectures

Question

- If a process exits and there are still threads of that process running, will they continue to run? Please justify your answer.

Answer

No. When a process exits, it takes everything with it—the process structure, the memory space, everything—including threads.

Kernel-Level Threads

- We have taken the execution aspect of a process and separated it out into threads
 - To make concurrency cheaper
- As such, the OS now manages threads *and* processes
 - All thread operations are implemented in the kernel
 - The OS schedules all of the threads in the system
- OS-managed threads are called **kernel-level threads** or **lightweight processes**
 - Windows: **threads**
 - Solaris: **lightweight processes (LWP)**
 - POSIX Threads (pthreads):
PTHREAD_SCOPE_SYSTEM

Kernel-level Thread Limitations

- Kernel-level threads make concurrency much cheaper than processes
 - Much less state to allocate and initialize
- However, for fine-grained concurrency, kernel-level threads still suffer from too much overhead
 - Thread operations still require system calls
 - Ideally, want thread operations to be **as fast as a procedure call**
- For such fine-grained concurrency, need even “cheaper” threads

User-Level Threads

- To make threads cheap and fast, they need to be implemented at user level
 - Kernel-level threads are managed by the OS
 - User-level threads are managed entirely by the run-time system (user-level library)
- User-level threads are small and fast
 - A thread is simply represented by a PC, registers, stack, and small thread control block (TCB)
 - Creating a new thread, switching between threads, and synchronizing threads are done via procedure call
 - No kernel involvement
 - User-level thread operations **100x faster** than kernel threads
 - pthreads: **PTHREAD_SCOPE_PROCESS**

User-level Thread Limitations

- ❑ But, user-level threads are not a perfect solution
 - ❑ As with everything else, they are a tradeoff
- ❑ User-level threads are **invisible** to the OS
 - ❑ They are not well integrated with the OS
- ❑ As a result, the OS can make poor decisions
 - ❑ Scheduling a process with idle threads
 - ❑ Blocking a process whose thread initiated an I/O, even though the process has other threads that can execute
- ❑ Solving this requires communication between the kernel and the user-level thread manager

Kernel- vs. User-level Threads

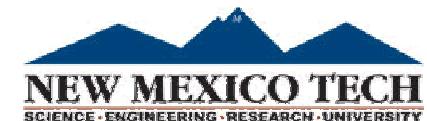
- ❑ Kernel-level threads
 - ❑ Integrated with OS (informed scheduling)
 - ❑ Slow to create, manipulate, synchronize
- ❑ User-level threads
 - ❑ Fast to create, manipulate, synchronize
 - ❑ Not integrated with OS (uninformed scheduling)
- ❑ Understanding the differences between kernel- and user-level threads is important

Question

- A disadvantage of ULTs is that when a ULT executes a blocking system call (e.g. a I/O call), not only is that thread blocked, but all of the threads within the process are blocked. Why?

Answer

- Because, with ULTs, the thread structure of a process is not visible to the operating system, which only schedules on the basis of processes.
- For a blocking system call like a I/O call, the control is transferred to the kernel. The kernel invokes the I/O action, places process in the blocked state, and switches to another process.

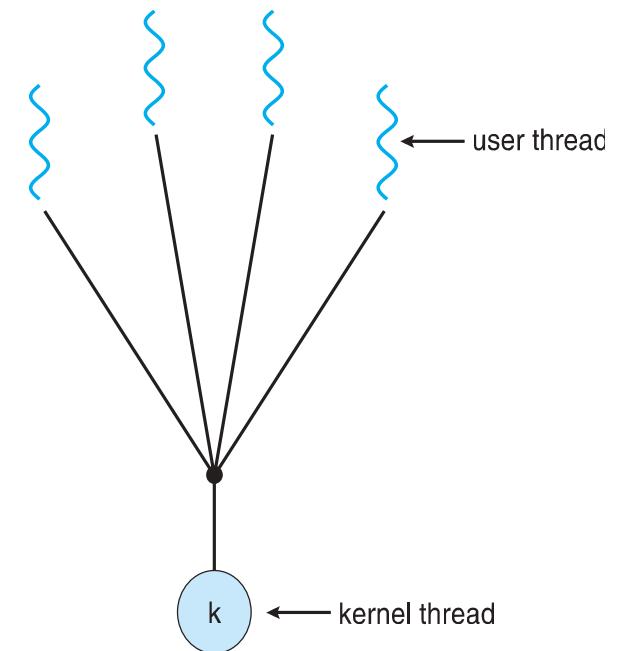


Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

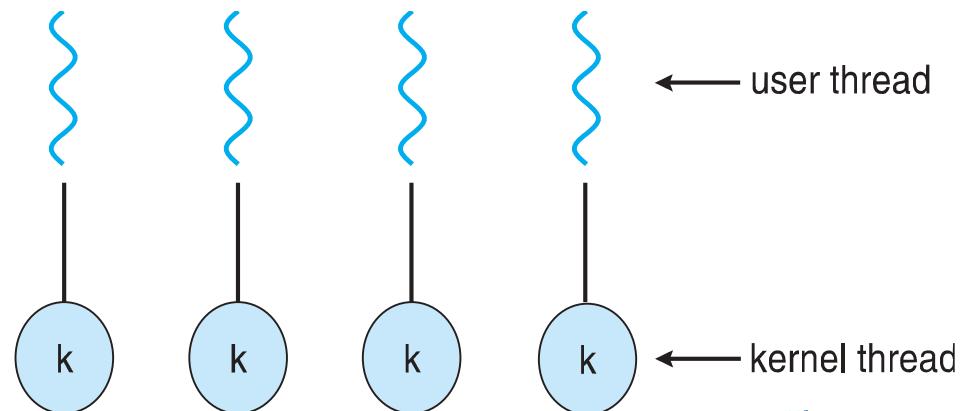
Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
 - **Solaris Green Threads**
 - **GNU Portable Threads**



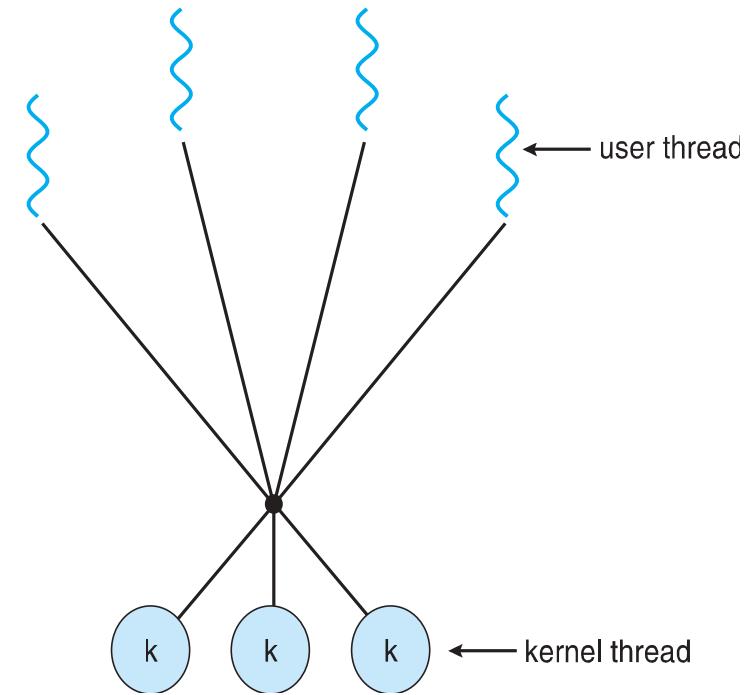
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
 - Windows
 - Linux
 - Solaris 9 and later



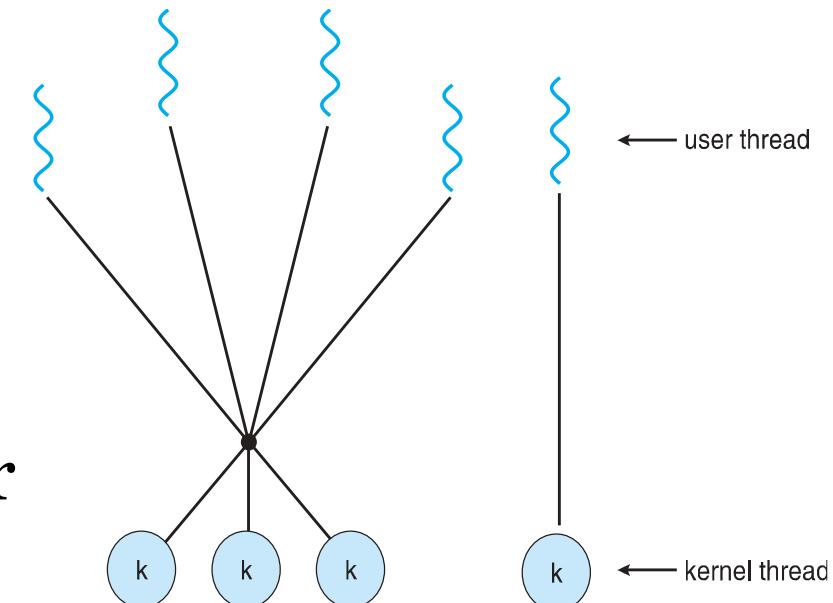
Many-to-Many Model

- 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 with the *ThreadFiber* package
- Extremely difficult to implement



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



Threads (3)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems

9/25/2019



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
- Three main thread libraries in use today
 - POSIX threads (Pthreads)
 - Win32
 - Java

Pthreads

- The standard interface for C threads
- <https://computing.llnl.gov/tutorials/pthreads/>
- To create threads
 - `int pthread_create(pthread_t *tid, pthread_attr_t *attr,
void *(*func)(void *), void *arg);`
 - **tid** is a pointer to an allocated (dynamic or otherwise) **pthread_t** (opaque type) that will have the thread ID of the new thread placed in it
 - **attr** is a pointer that can be used to change the attributes of the new thread (but we'll usually just use **NULL**)
 - **func** is a function pointer to the new thread's function to execute
 - **arg** is a pointer that will be passed to the new thread's routine when the thread is created; this is the way you pass arguments to a thread
 - returns **0** on success, **nonzero** on error
 - we can use structures to pass multiple values to the function executed by the thread



Thread Termination

- Threads can be terminated in one of four ways:
 - **Implicit termination**: thread routine returns; usually what we'll use
 - **Explicit termination**: the thread calls `pthread_exit()`
 - **Process exit**: any thread calls `exit()`, which terminates the process and all associated threads; maybe not what you really want
 - **Thread cancellation**: another thread calls `pthread_cancel()` to terminate a specific thread

Thread Reaping

- When a thread has terminated, information about it, including the thread return value, is still kept in memory until reaped by another thread
- Threads are reaped by **pthread_join()**:
 - **int pthread_join(pthread_t tid, void **val);**
 - Reaps thread with thread ID **tid**
 - Blocks until thread **tid** terminates
 - Frees memory resources held by thread **tid**
 - Returns 0 on success, nonzero on error
 - On success, ***val** is the return value of the terminated thread

Compiling Pthreads Code

- ❑ To compile the example programs, you need to tell **gcc** to use the pthread library when linking your executable
- ❑ To do this, use the **-l** switch to **gcc**:
 - ❑ **gcc -o threadex threadex.c -lpthread**
 - ❑ The **-l** switch is needed to use many other libraries
 - ❑ math functions, for example
 - ❑ If you forget this, your program will not compile, and you will get an error like this:
 - ❑ **/tmp/cc3VuzAe.o(.text+0x3a): In function `main': : undefined reference to `pthread_create'**
 - ❑ You can add this flag to the **LDFLAGS** variable in your **Makefile** to have it work correctly
- ❑ All of the functions we'll talk about that begin with "pthread_" require including **<pthread.h>**
- ❑ Example: **threads_basics.c, threads_ret_sqrt_value.c**

Several Threads

Example: `threads_several.c`

- Why do we create an array called **ids**? Why not just pass in **&i** as the argument?
 - The value of **i** changes; if it changes before the new thread can access the memory, it's a problem.
 - What will the output be? Do we know what the first line to be printed out will be? How about the last?
 - Most of the output will be interleaved

Example: `threads_no_sync.c`

Process Synchronization (1)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems

9/25/2019

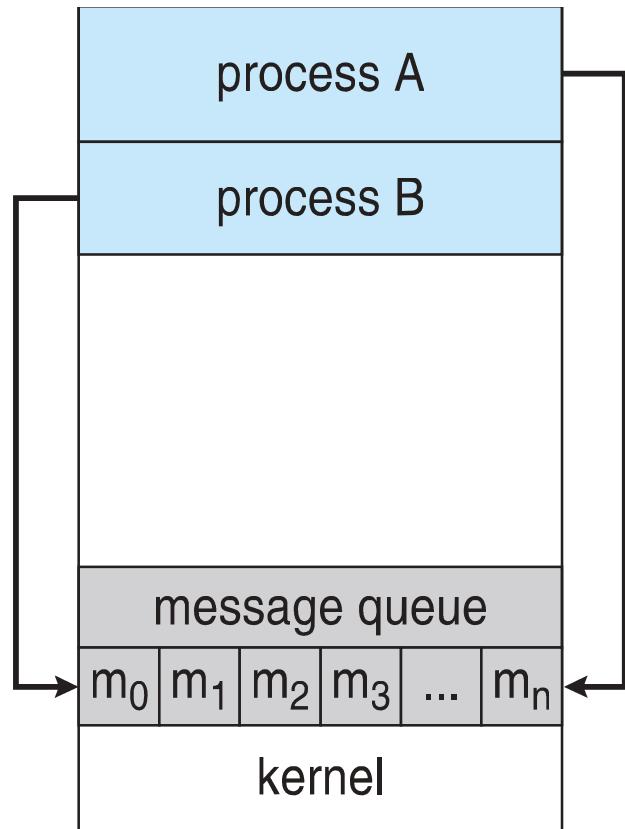


Interprocess Communication

- Processes within a system may be *independent* or *cooperating*
- Cooperating process can affect or be affected by other processes, including sharing data
- Cooperating processes need **interprocess communication (IPC)**
 - Message passing
 - Shared memory

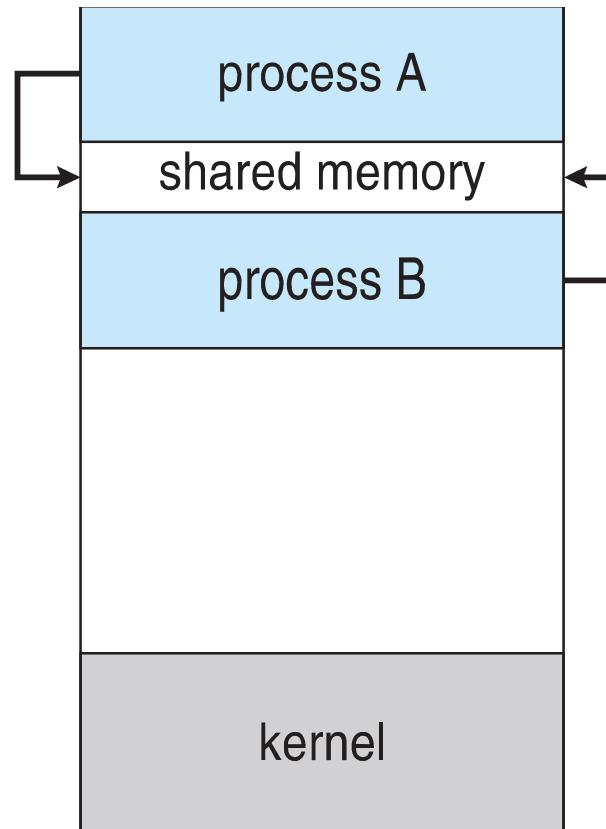
Two Communications Models

(a) Message passing.



(a)

(b) shared memory.



(b)

Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Concurrent access to shared data may result in data inconsistency

Producer-Consumer Problem

- A common paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
- **Shared memory (a buffer of items filled by the producer and emptied by the consumer)**
 - **unbounded-buffer** places no practical limit on the size of the buffer
 - **bounded-buffer** assumes that there is a fixed buffer size
- Producer and consumer must be synchronized

Bounded-Buffer – Shared-Memory Solution

Shared data

```
#define BUFFER_SIZE 10

typedef struct {

    . . .

} item;

item buffer[BUFFER_SIZE];
int in = 0; // next free position
int out = 0; // first full position
int counter = 0;
```

Empty: $\text{in} == \text{out}$, Full: $((\text{in}+1) \% \text{BUFFER_SIZE}) == \text{out}$



Producer

```
while (true) {  
    /* produce an item in next_produced */  
  
    while (counter == BUFFER_SIZE) ;  
    /* do nothing */  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```



Consumer

```
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
    /* consume the item in next consumed */  
}
```

- ❑ Does not consider the producer process and the consumer process attempt to access to the shared buffer concurrently

Race Condition

counter++ could be implemented as

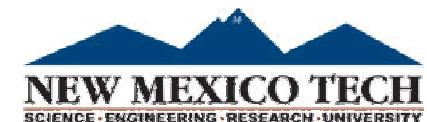
```
register1 = counter  
register1 = register1 + 1  
counter = register1
```

counter-- could be implemented as

```
register2 = counter  
register2 = register2 - 1  
counter = register2
```

Consider this execution interleaving with “count = 5” initially:

So: producer execute register1 = counter	{register1 = 5}
S1: producer execute register1 = register1 + 1	{register1 = 6}
S2: consumer execute register2 = counter	{register2 = 5}
S3: consumer execute register2 = register2 - 1	{register2 = 4}
S4: producer execute counter = register1	{counter = 6 }
S5: consumer execute counter = register2	{counter = 4}



Process Synchronization (2)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems
9/30/2019



Critical Section Problem

- Consider system of n processes $\{p_0, p_1, \dots p_{n-1}\}$
- 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**

Critical Section

General structure of process P_i

do {

entry section

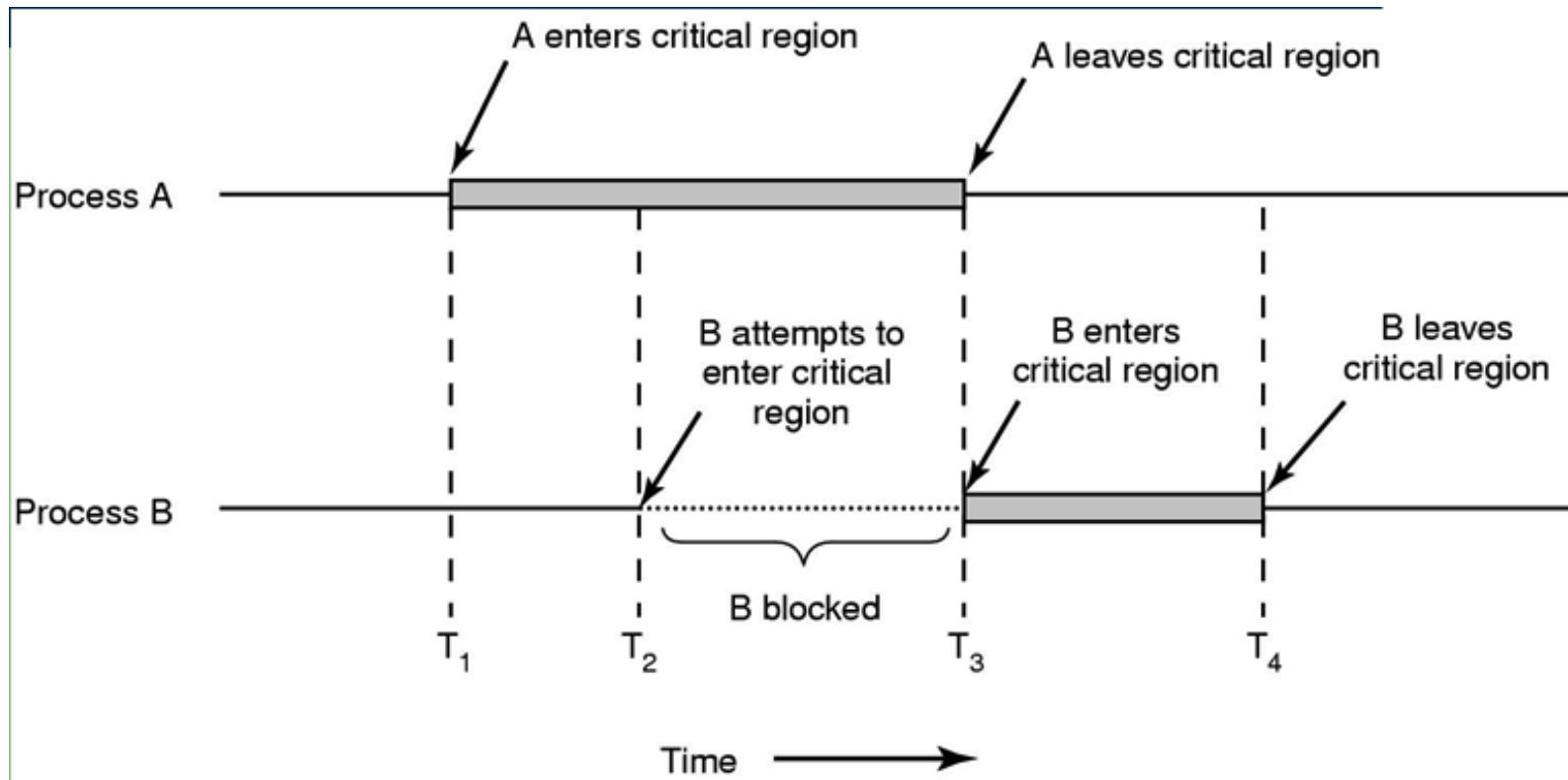
critical section

exit section

remainder section

} while (true);

Critical Region Illustrated



Solution to Critical-Section Problem

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

Peterson's Solution

- Good algorithmic description of solving the critical section problem
- Two process solution
- Assume that the **load** and **store** machine-language 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 **P_i** is ready!

Algorithm for Process P_i

```
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j); //busy wait  
        critical section  
    flag[i] = false;  
        remainder section  
} while (true);
```

Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:
 1. Mutual exclusion is preserved
 P_i enters CS only if:
either `flag[j] = false` or `turn = i`
 2. Progress requirement is satisfied
 3. Bounded-waiting requirement is met

Synchronization Hardware

- More solutions using techniques ranging from hardware to software-based APIs available to both kernel developer and application programmers.
- All solutions based on idea of **locking**
 - Protecting critical regions via locks
- Modern machines provide special atomic hardware instructions
 - **Atomic** = non-interruptible
 - Either test memory word and set value
 - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

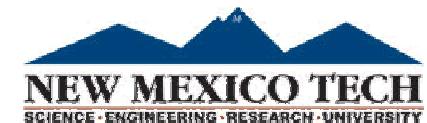
```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (TRUE);
```

test_and_set Instruction

Definition:

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

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.



Solution using test_and_set()

- Shared Boolean variable lock,
initialized to FALSE
- Solution:

```
do {  
    while (test_and_set (&lock))  
        ; /* do nothing */  
    /* critical section */  
    lock = false;  
    /* remainder section */  
} while (true);
```

Process Synchronization (3)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems

10/2/2019

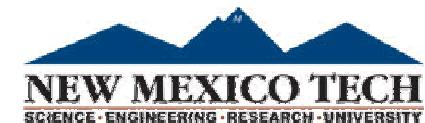


Mutex Locks

- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **acquire()** a lock then **release()** the lock
 - Boolean variable indicating if lock is available or not
- Calls to **acquire()** and **release()** must be atomic
 - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
 - This lock therefore called a **spinlock**

acquire() and release()

```
acquire() {  
    while (!available)  
        ; /* busy wait */  
    available = false;  
}  
  
release() {  
    available = true;  
}  
  
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (true);
```



Question

- ❑ Why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems?

Answer

- Spinlocks are not appropriate for single-processor systems because the condition that would break a process out of the spinlock can be obtained only by executing a different process. If the process is not relinquishing the processor, other processes do not get the opportunity to set the program condition required for the first process to make progress. In a multiprocessor system, other processes execute on other processors and thereby modify the program state in order to release the first process from the spinlock.

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations
 - **wait ()** and **signal ()**
 - Originally called **P ()** and **V ()**
- Definition of the **wait () operation**

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```

- Definition of the **signal () operation**

```
signal(S) {  
    S++;  
}
```

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
 - Same as a **mutex lock**
 - Can solve various synchronization problems
 - Consider P_1 and P_2 that require S_1 to happen before S_2
Create a semaphore “**synch**” initialized to 0

P1:

```
s1;  
signal(synch);
```

P2:

```
wait(synch);  
s2;
```

- Can implement a counting semaphore S as a binary semaphore

Semaphore Implementation

- Must guarantee that no two processes can execute the **wait ()** and **signal ()** on the same semaphore at the same time
- Thus, the 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
 - Note that applications may spend lots of time in critical sections and therefore busy waiting is not a good solution

Deadlock and Starvation

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

P_0

wait (S) ;

wait (Q) ;

...

signal (S) ;

signal (Q) ;

P_1

wait (Q) ;

wait (S) ;

...

signal (Q) ;

signal (S) ;

- **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

Priority Inversion Example

- Three jobs with three priorities (L, M, H)
- Assume that process H requires resource R , which is currently being accessed by process L .
- Ordinarily, process H would wait for L to finish using resource R . However, now suppose that process M becomes runnable, thereby preempting process L .
- Indirectly, a process with a lower priority process M has affected how long process H must wait for L to relinquish resource R .

Mars Pathfinder

- Mars Pathfinder, a NASA space probe that landed a robot, the Sojourner rover, on Mars in 1997 to conduct experiments. Shortly after the Sojourner began operating, it started to experience frequent computer resets. Each reset reinitialized all hardware and software, including communications. If the problem had not been solved, the Sojourner would have failed in its mission.
- The problem was caused by the fact that one high-priority task, “bc dist,” was taking longer than expected to complete its work. This task was being forced to wait for a shared resource that was held by the lower-priority “ASI/MET” task, which in turn was preempted by multiple medium-priority tasks. The “bc dist” task would stall waiting for the shared resource, and ultimately the “bc sched” task would discover the problem and perform the reset. The Sojourner was suffering from a typical case of **priority inversion**.
- How to solve it?

Priority Inheritance Protocol

- L executes its critical section at H's (high) priority. As a result, M will be unable to preempt L and will be blocked.
- When L exits its critical section, it regains its original (low) priority and awakens H (which was blocked by L).
- H, having high priority, preempts L and runs to completion.

In-Class Work 3

- The following figure shows the sequence of semaphore operations at the beginning and at the end of the tasks A, B.
- Determine for each of the 4 cases a, b, c and d, whether or in which sequence the tasks are executed, using the initializations of the semaphore variables given in Table 1.
- `wait()` is an atomic operation that waits for semaphore to become positive, then decrements it by 1, i.e. if the semaphore is 0, `wait()` is blocked.
- `signal()` is an atomic operation that increments the semaphore by 1, waking up a waiting `wait()` if any.
- SA, SB are the two semaphores corresponding to Tasks A and B.

In-Class Work 3

Task A

wait(SA)

wait(SA)

...

...

...

signal(SB)

END

Task B

wait(SB)

wait(SB)

...

...

...

signal(SA)

END

Table 1

	a	b	c	d
SA	2	1	0	2
SB	0	1	2	1

Process Synchronization (4)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems

10/4/2019



In-Class Work 3

- The following figure shows the sequence of semaphore operations at the beginning and at the end of the tasks A, B.
- Determine for each of the 4 cases a, b, c and d, whether or in which sequence the tasks are executed, using the initializations of the semaphore variables given in Table 1.
- `wait()` is an atomic operation that waits for semaphore to become positive, then decrements it by 1, i.e. if the semaphore is 0, `wait()` is blocked.
- `signal()` is an atomic operation that increments the semaphore by 1, waking up a waiting `wait()` if any.
- SA, SB are the two semaphores corresponding to Tasks A and B.

In-Class Work 3

Task A

wait(SA)

wait(SA)

...

...

...

signal(SB)

END

Task B

wait(SB)

wait(SB)

...

...

...

signal(SA)

END

Table 1

	a	b	c	d
SA	2	1	0	2
SB	0	1	2	1

Answer

a: task A

b: No task will be executed

c: task B

d: task A, task B

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
 - Bounded-Buffer Problem
 - Readers and Writers Problem
 - Dining-Philosophers Problem

Bounded-Buffer Problem

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



Bounded Buffer Problem (Cont.)

The structure of the producer process

```
do {  
    ...  
    /* produce an item in next_produced */  
    ...  
    wait(empty);  
    wait(mutex);  
    ...  
    /* add next produced to the buffer */  
    ...  
    signal(mutex);  
    signal(full);  
} while (true);
```

Bounded Buffer Problem (Cont.)

The structure of the consumer process

```
Do {  
    wait(full);  
    wait(mutex);  
    ...  
    /* remove an item from buffer to next_consumed */  
    ...  
    signal(mutex);  
    signal(empty);  
    ...  
    /* consume the item in next consumed */  
    ...  
} while (true);
```



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 considered – all involve some form of priorities
- Shared Data
 - Data set
 - Semaphore **rw_mutex** initialized to 1
 - Semaphore **mutex** initialized to 1
 - Integer **read_count** initialized to 0

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 the write ASAP
- Both may have starvation leading to even more variations

Readers-Writers Problem (Cont.)

The structure of a writer process

```
do {  
    wait (rw_mutex);  
  
    ...  
    /* writing is performed */  
  
    ...  
    signal (rw_mutex);  
} while (true);
```

Readers-Writers Problem (Cont.)

The structure of a reader process

```
do {
    wait(mutex);
    read_count++;          /* Protected by mutex */
    if (read_count == 1) /* First reader in */
        wait(rw_mutex); /* Lock out writers */

    signal(mutex);

    ...
    /* reading is performed */

    ...

    wait(mutex);
    read_count--; /* Protected by mutex */
    if (read_count == 0) /* Last out */
        signal(rw_mutex); /* Let in writers */

    signal(mutex);
} while (true);
```

Dining-Philosophers Problem



- Philosophers spend their lives alternating 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 Problem Algorithm

- 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?

Process Synchronization (5)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems
10/7/2019



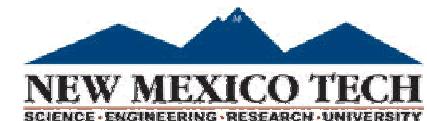
Dining-Philosophers Problem

Algorithm (Cont.)

- Deadlock handling
 - Allow at most 4 philosophers to be sitting simultaneously at the table.
 - Allow a philosopher to pick up the chopsticks only if both are available (picking must be done in a critical section).
 - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Problems with Semaphores

- Incorrect use of semaphore operations:
 - signal (mutex) wait (mutex)
 - wait (mutex) ... wait (mutex)
 - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.



Monitors

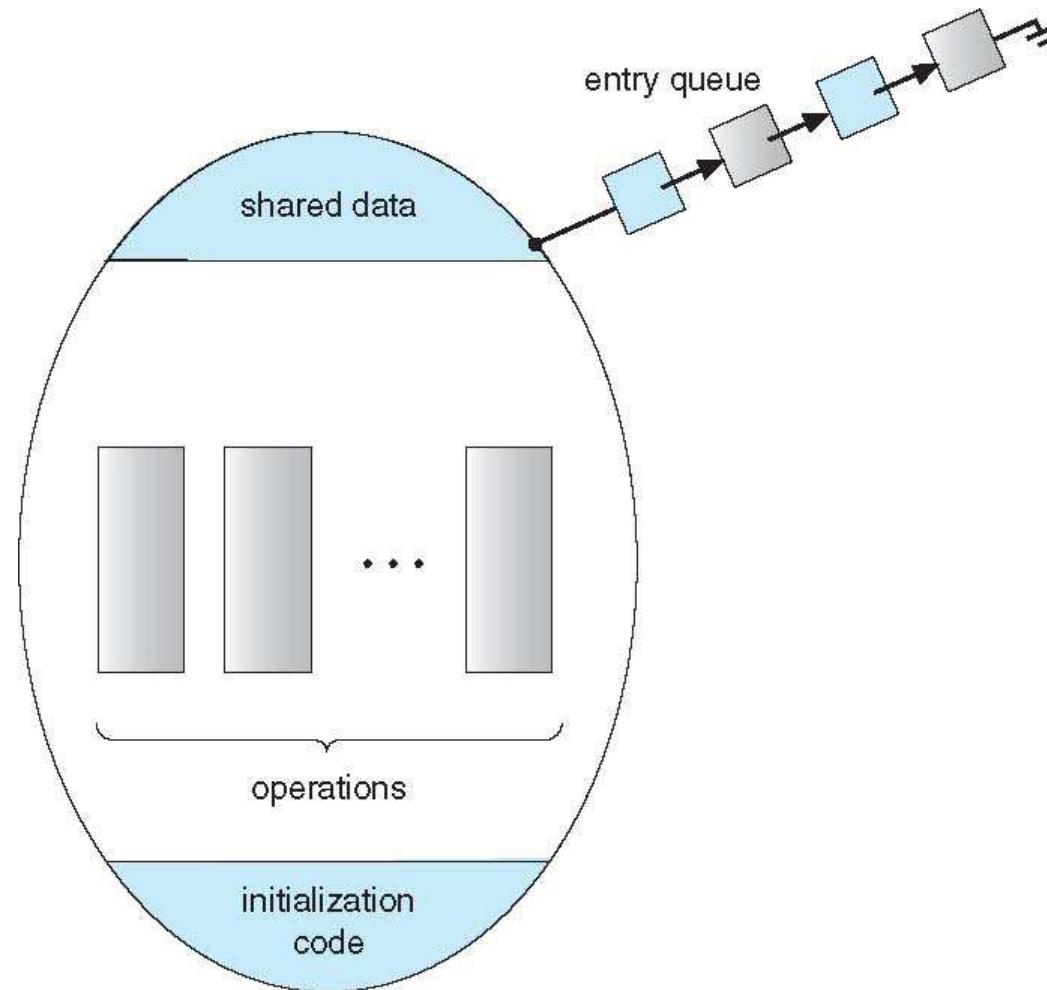
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { .... }

    procedure Pn (...) {.....}

    Initialization code (...) { ... }
}
}
```

Schematic View of a Monitor



But not powerful enough to model some synchronization schemes

Condition Variables

`condition x;`

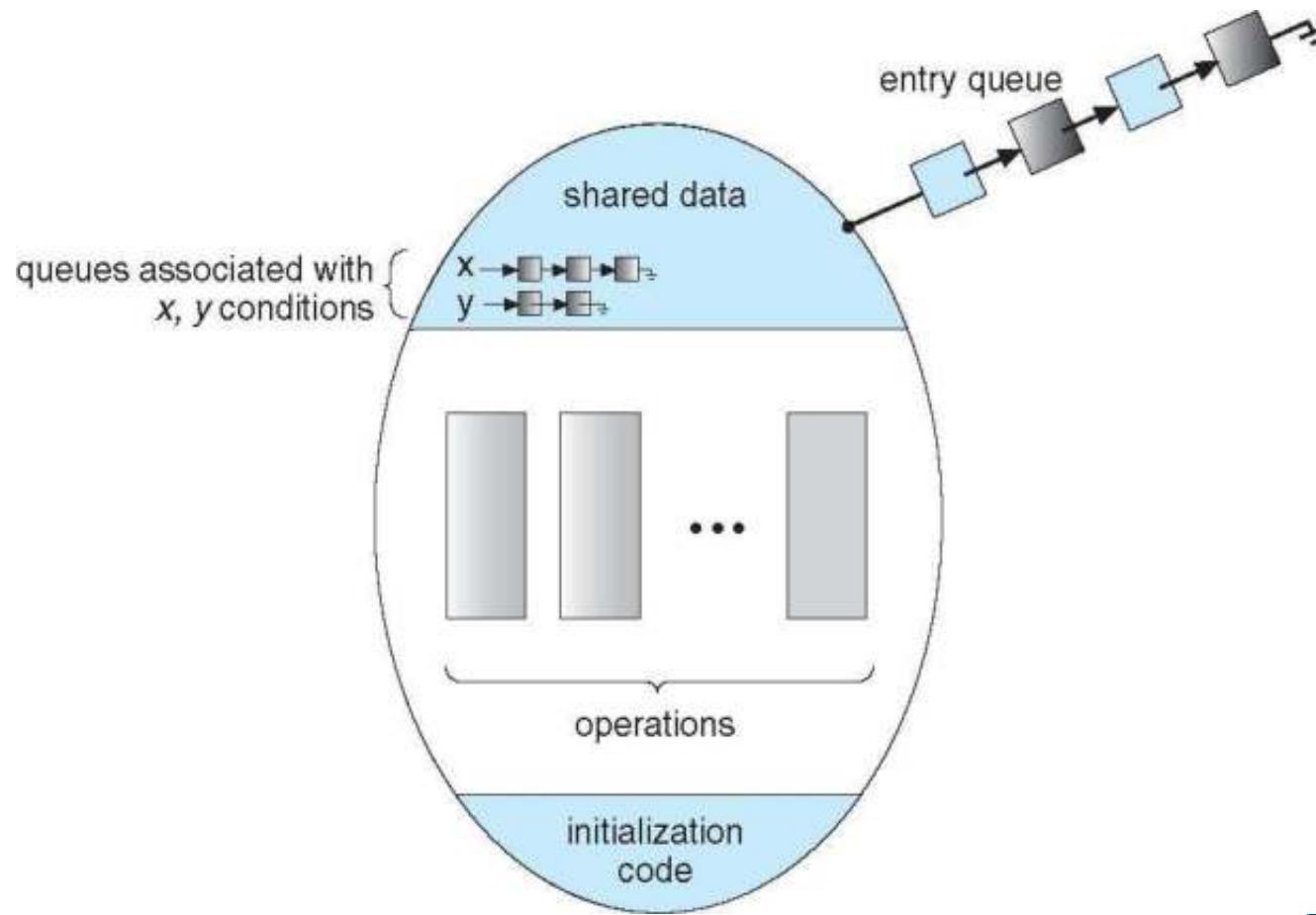
- Two operations are allowed on a condition variable:

`x.wait()` – a process that invokes the operation is suspended until `x.signal()`

`x.signal()` – resumes one of processes (if any) that invoked `x.wait()`

If no `x.wait()` on the variable, then it has no effect on the variable

Monitor with Condition Variables



Condition Variables Choices

- If process P invokes **x.signal()**, and process Q is suspended in **x.wait()**, what should happen next?
 - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include
 - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
 - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
 - Both have pros and cons – language implementer can decide
 - Monitors implemented in Concurrent Pascal compromise
 - P executing signal immediately leaves the monitor, Q is resumed
 - Implemented in other languages including Mesa, C#, Java

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
    enum { THINKING, HUNGRY, EATING} state [5];
    condition self [5];

    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self[i].wait;
    }

    void putdown (int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }
}
```

Solution to Dining Philosophers (Cont.)

```
void test (int i) {  
    if ((state[(i + 4) % 5] != EATING) &&  
        (state[i] == HUNGRY) &&  
        (state[(i + 1) % 5] != EATING)) {  
        state[i] = EATING ;  
        self[i].signal () ;  
    }  
}  
  
initialization_code() {  
    for (int i = 0; i < 5; i++)  
        state[i] = THINKING;  
}  
}
```



Solution to Dining Philosophers (Cont.)

- Each philosopher i invokes the operations **pickup ()** and **putdown ()** in the following sequence:

DiningPhilosophers.pickup (i) ;

EAT

DiningPhilosophers.putdown (i) ;

- No deadlock, but starvation is possible

Resuming Processes within a Monitor

- If several processes queued on condition x , and $x.signal()$ executed, which should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form
 $x.wait(c)$
 - Where c is **priority number**
 - Process with lowest number (highest priority) is scheduled next

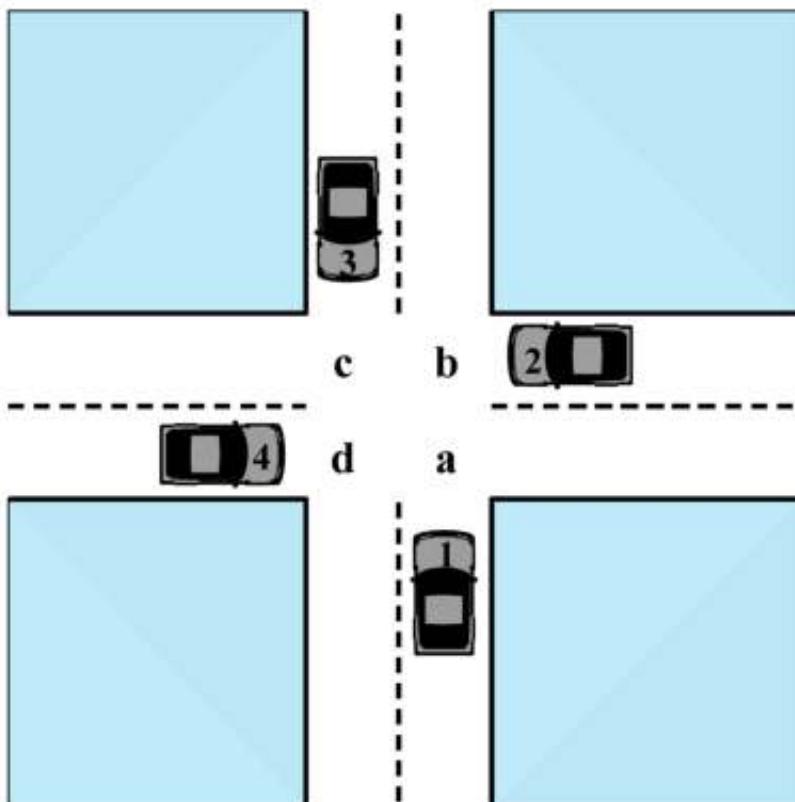
Deadlocks (1)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems
10/7/2019

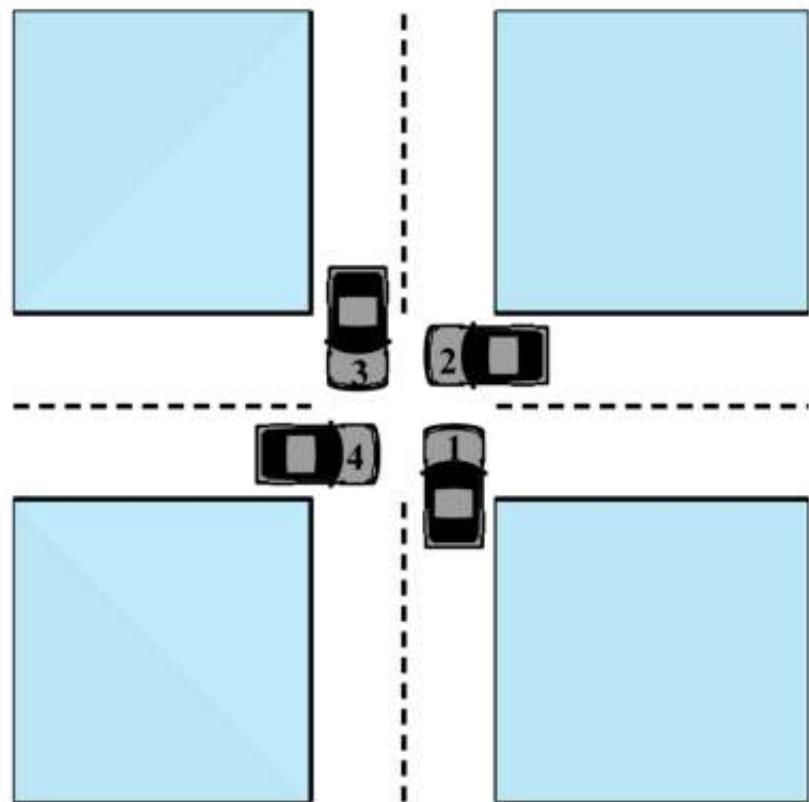


Deadlock

- The permanent blocking of a set of processes that either compete for system resources or communicate with each other
- A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set
- Permanent
- No efficient solution



(a) Deadlock possible



(b) Deadlock

Traffic Deadlock

Example 1: Reusable Resources

Process P

Step	Action
p ₀	Request (D)
p ₁	Lock (D)
p ₂	Request (T)
p ₃	Lock (T)
p ₄	Perform function
p ₅	Unlock (D)
p ₆	Unlock (T)

Process Q

Step	Action
q ₀	Request (T)
q ₁	Lock (T)
q ₂	Request (D)
q ₃	Lock (D)
q ₄	Perform function
q ₅	Unlock (T)
q ₆	Unlock (D)

Example 2: Memory Request

- Space is available for allocation of 200Kbytes, and the following sequence of events occur:

P1

...

Request 80 Kbytes;

...

Request 60 Kbytes;

P2

...

Request 70 Kbytes;

...

Request 80 Kbytes;

- Deadlock occurs if both processes progress to their second request

Deadlocks (2)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems
10/9/2019



System Model

- System consists of resources
- Resource types R_1, R_2, \dots, R_m
 - *CPU cycles, memory space, I/O devices*
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
 - **request**
 - **use**
 - **release**

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **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 .

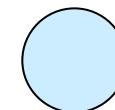
Resource-Allocation Graph

A set of vertices V and a set of edges E .

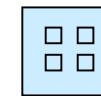
- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

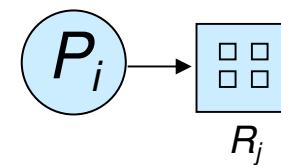
❑ Process



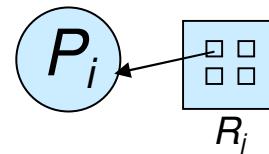
❑ Resource Type with 4 instances



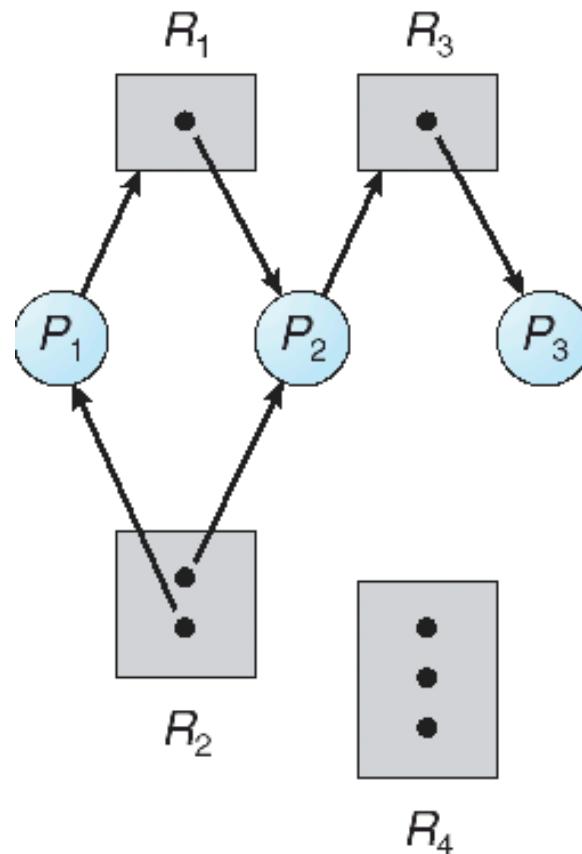
❑ P_i requests instance of R_j



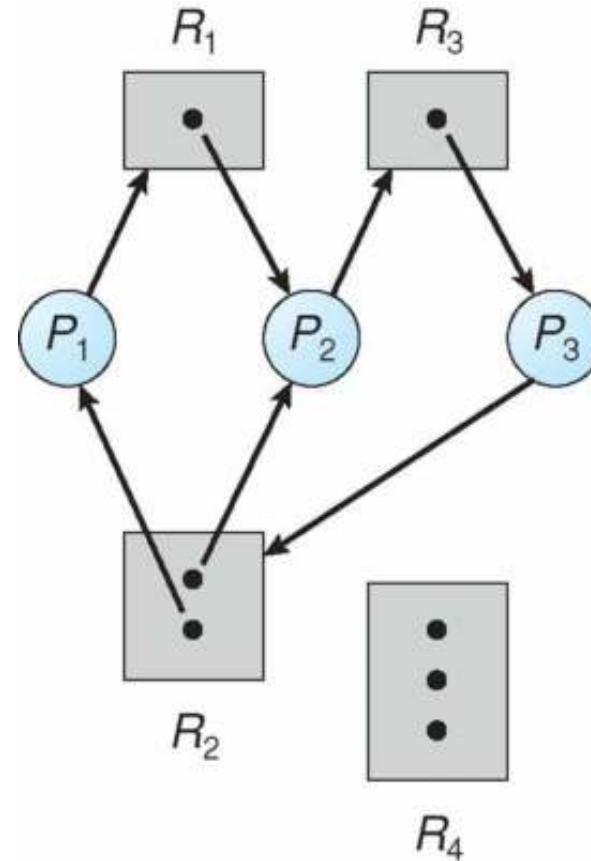
❑ P_i is holding an instance of R_j



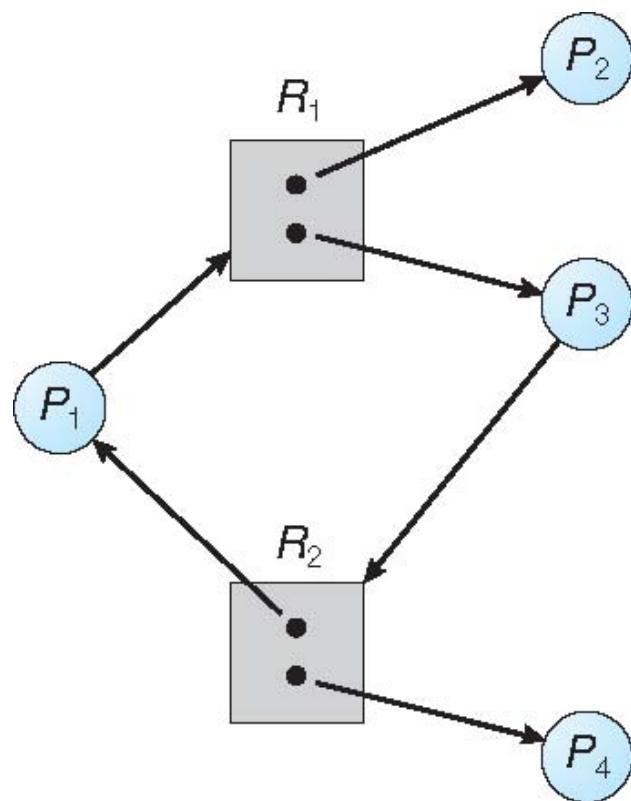
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Basic Facts

- If graph contains no cycles \Rightarrow 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:
 - ❑ Deadlock prevention
 - ❑ Deadlock avoidance
- ❑ Allow the system to enter a deadlock state and then detect and recover
- ❑ Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

Restrain the ways request can be made

- **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

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

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

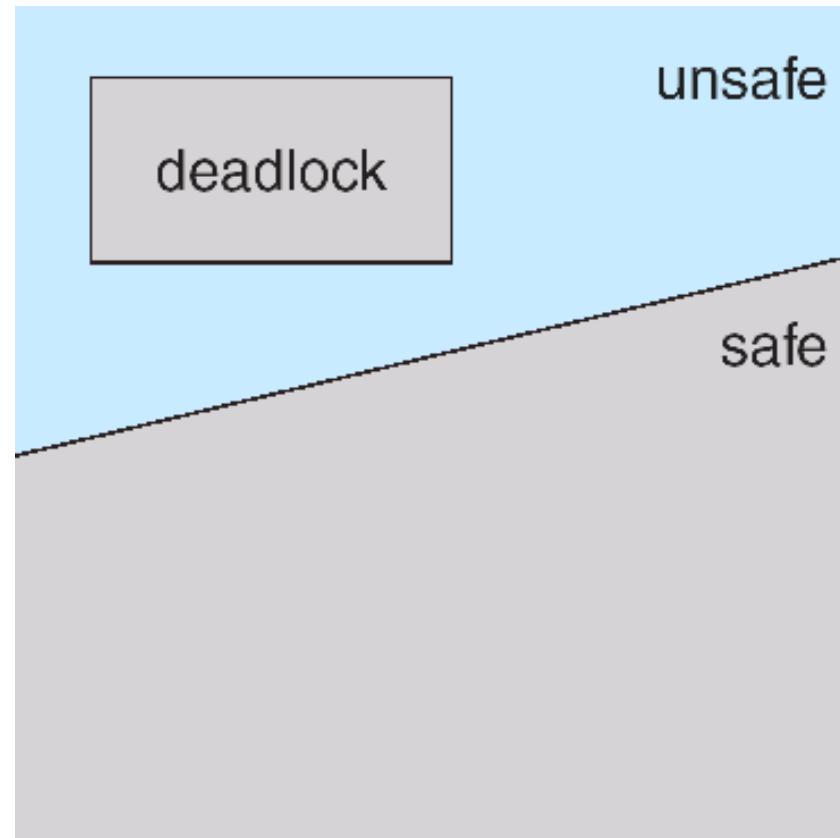
Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence $\langle P_1, P_2, \dots, P_n \rangle$ of ALL the processes in the systems such that for each P_i , the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j , with $j < i$
- That is:
 - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
 - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
 - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Basic Facts

- If a system is in safe state \Rightarrow no deadlocks
- If a system is in unsafe state \Rightarrow possibility of deadlock
- Avoidance \Rightarrow ensure that a system will never enter an unsafe state.

Safe, Unsafe, Deadlock State



Avoidance Algorithms

- Single instance of a resource type
 - Use a resource-allocation graph
- Multiple instances of a resource type
 - Use the banker' s algorithm

Deadlocks (3)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems

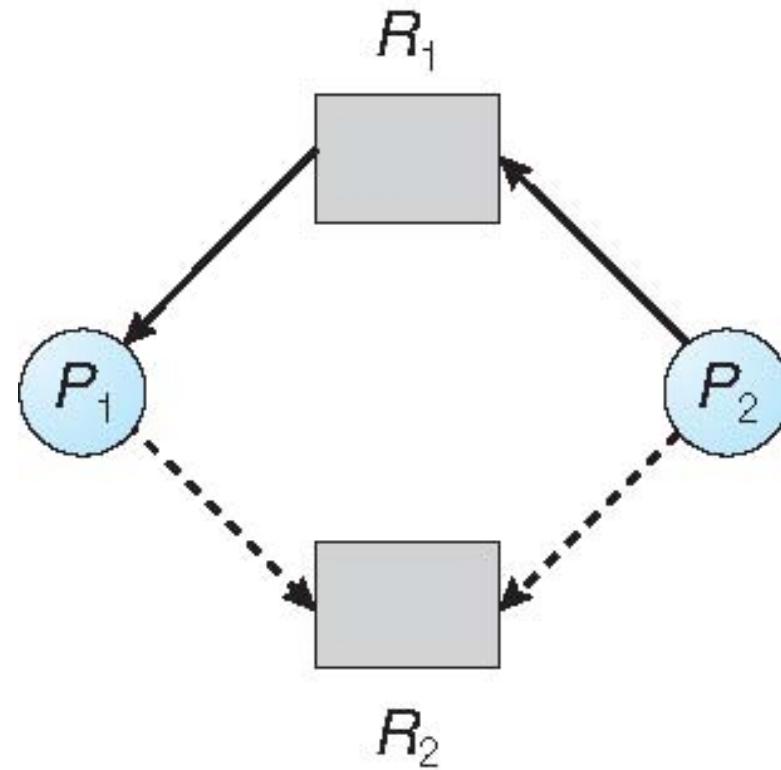
10/11/2019



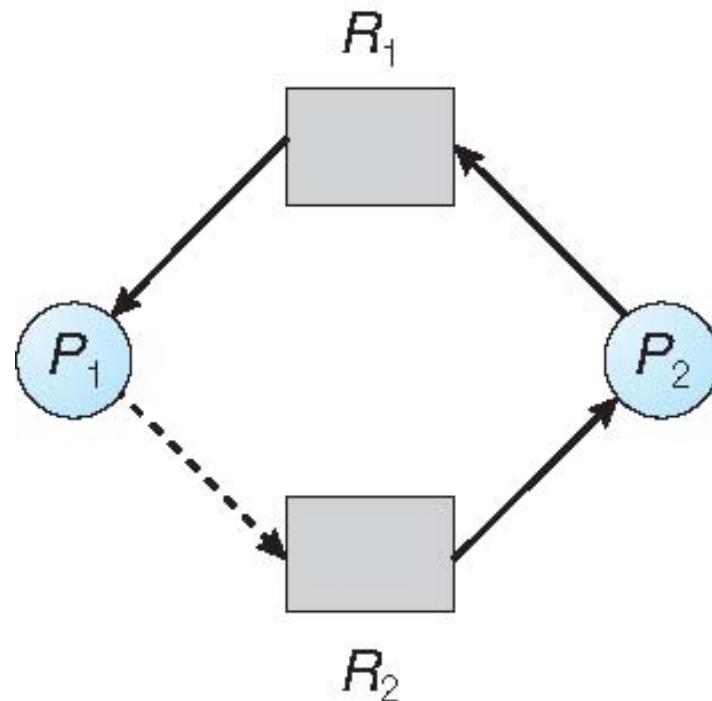
Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j ; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Banker's Algorithm

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Data Structures for the Banker' s Algorithm

Let n = number of processes, and m = number of resources types.

- **Available:** Vector of length m . If $\text{available}[j] = k$, there are k instances of resource type R_j available
- **Max:** $n \times m$ matrix. If $\text{Max}[i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation:** $n \times m$ matrix. If $\text{Allocation}[i,j] = k$ then P_i is currently allocated k instances of R_j
- **Need:** $n \times m$ matrix. If $\text{Need}[i,j] = k$, then P_i may need k more instances of R_j to complete its task
$$\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$$

Safety Algorithm

- ⑩ 1. Let Work and Finish be vectors of length m and n , respectively. Initialize:
 - $\text{Work} = \text{Available}$
 - $\text{Finish}[i] = \text{false}$ for $i = 0, 1, \dots, n-1$
- ⑩ 2. Find an i such that both:
 - ∅(a) $\text{Finish}[i] = \text{false}$
 - ∅(b) $\text{Need}_i \leq \text{Work}$
 - ∅If no such i exists, go to step 4
- ⑩ 3. $\text{Work} = \text{Work} + \text{Allocation}_i$
 $\text{Finish}[i] = \text{true}$
go to step 2
- ⑩ 4. If $\text{Finish}[i] == \text{true}$ for all i , then the system
is in a safe state

Resource-Request Algorithm for Process P_i

- ⑩ $\mathbf{Request}_i$ = request vector for process P_i . If $\mathbf{Request}_i[j] = k$ then process P_i wants k instances of resource type R_j
- ☞ 1. If $\mathbf{Request}_i \leq \mathbf{Need}_i$, go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
 - ☞ 2. If $\mathbf{Request}_i \leq \mathbf{Available}$, go to step 3. Otherwise P_i must wait, since resources are not available
 - ☞ 3. Pretend to allocate requested resources to P_i by modifying the state as follows:
 - $\mathbf{Available} = \mathbf{Available} - \mathbf{Request}_i;$
 - $\mathbf{Allocation}_i = \mathbf{Allocation}_i + \mathbf{Request}_i;$
 - $\mathbf{Need}_i = \mathbf{Need}_i - \mathbf{Request}_i;$
 - If safe \Rightarrow the resources are allocated to P_i
 - If unsafe $\Rightarrow P_i$ must wait, and the old resource allocation state is restored

Example of Banker' s Algorithm

5 processes P_0 through P_4 ;

3 resource types:

A (10 instances), B (5 instances), and C (7 instances)

Snapshot at time T_0 :

	<u>Allocation</u>			<u>Max</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Example (Cont.)

The content of the matrix ***Need*** is defined to be ***Max – Allocation***

	<u>Need</u>		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

The system is in a safe state since the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies safety criteria

Example: P_1 Request (1,0,2)

Check that Request \leq Available (that is, $(1,0,2) \leq (3,3,2)$ \Rightarrow true

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Executing safety algorithm shows that sequence $< P_1, P_3, P_4, P_0, P_2 >$ satisfies safety requirement

Can request for (3,3,0) by P_4 be granted?

Can request for (0,2,0) by P_0 be granted?

Deadlocks (4)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems

10/14/2019



Example of Banker' s Algorithm

5 processes P_0 through P_4 ;

3 resource types:

A (10 instances), B (5 instances), and C (7 instances)

Snapshot at time T_0 :

	<u>Allocation</u>			<u>Max</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Example (Cont.)

The content of the matrix ***Need*** is defined to be ***Max – Allocation***

	<u>Need</u>		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

The system is in a safe state since the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies safety criteria

Example: P_1 Request (1,0,2)

Check that Request \leq Available (that is, $(1,0,2) \leq (3,3,2)$ \Rightarrow true

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Executing safety algorithm shows that sequence $< P_1, P_3, P_4, P_0, P_2 >$ satisfies safety requirement

Can request for (3,3,0) by P_4 be granted?

Can request for (0,2,0) by P_0 be granted?

In-Class Work 4

Consider the following snapshot of a system:

<i>Allocation</i>	<i>Max</i>	<i>Available</i>
ABCD	ABCD	ABCD
P₀ 0012	0012	1520
P₁ 1000	1750	
P₂ 1354	2356	
P₃ 0632	0652	
P₄ 0014	0656	

Answer the following questions using the banker's algorithm:

- What is the content of the matrix *Need*?
- Is the system in a safe state?
- If a request from process P₁ arrives for (0,4,2,0), can the request be granted immediately?

Answer

- a. The values of **Need** for processes P_0 through P_4 respectively are $(0, 0, 0, 0)$, $(0, 7, 5, 0)$, $(1, 0, 0, 2)$, $(0, 0, 2, 0)$, and $(0, 6, 4, 2)$.
- b. Yes. With **Available** being equal to $(1, 5, 2, 0)$, either process P_0 or P_3 could run. Once process P_3 runs, it releases its resources, which allow all other existing processes to run.
- c. This results in the value of **Available** being $(1, 1, 0, 0)$. One ordering of processes that can finish is P_0, P_2, P_3, P_1 , and P_4 .

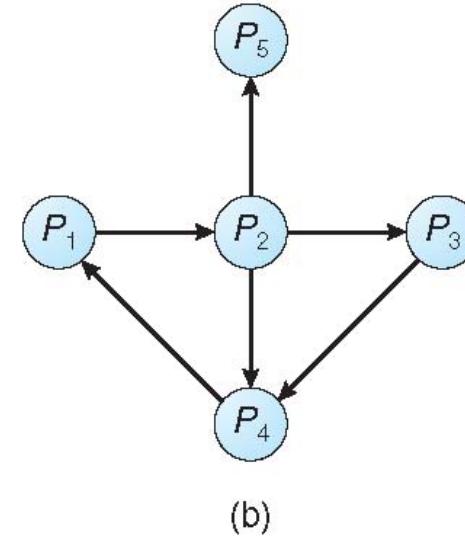
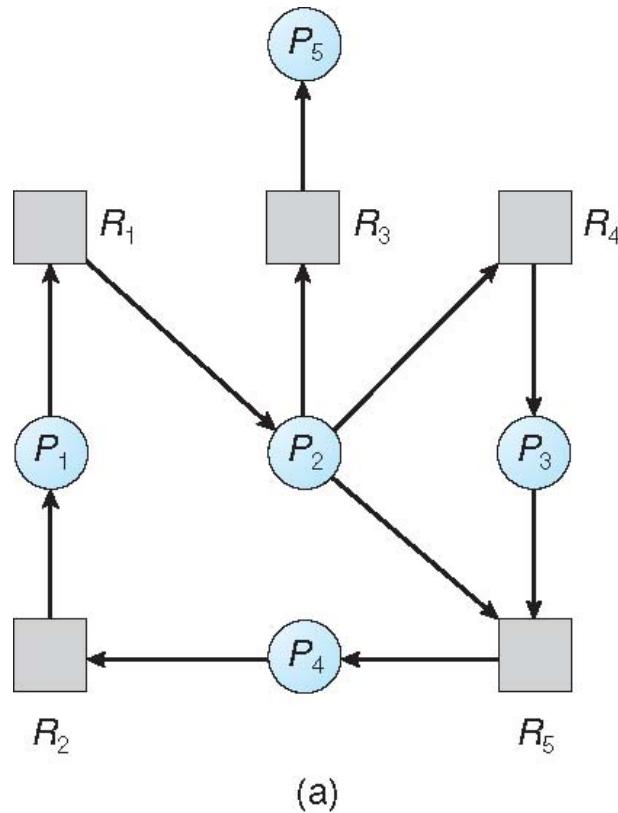
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance of Each Resource Type

- Maintain **wait-for** graph
 - Nodes are processes
 - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph

Corresponding wait-for graph

Several Instances of a Resource Type

- **Available**: A vector of length m indicates the number of available resources of each type
- **Allocation**: An $n \times m$ matrix defines the number of resources of each type currently allocated to each process
- **Request**: An $n \times m$ matrix indicates the current request of each process. If $\text{Request}[i][j] = k$, then process P_i is requesting k more instances of resource type R_j .

Detection Algorithm

1. Let \mathbf{Work} and \mathbf{Finish} be vectors of length m and n , respectively Initialize:
 - ☞(a) $\mathbf{Work} = \mathbf{Available}$
 - ☞(b) For $i = 1, 2, \dots, n$, if $\mathbf{Allocation}_i \neq \mathbf{0}$, then $\mathbf{Finish}[i] = \mathit{false}$; otherwise, $\mathbf{Finish}[i] = \mathit{true}$
2. Find an index i such that both:
 - ☞(a) $\mathbf{Finish}[i] == \mathit{false}$
 - ☞(b) $\mathbf{Request}_i \leq \mathbf{Work}$

☞ If no such i exists, go to step 4

Detection Algorithm (Cont.)

3. $Work = Work + Allocation_i$

$Finish[i] = true$

go to step 2

4. If $Finish[i] == false$, for some i , $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if $Finish[i] == false$, then P_i is deadlocked

Algorithm requires an order of $O(m \times n^2)$ operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

Five processes P_0 through P_4 ; three resource types A (7 instances), B (2 instances), and C (6 instances)

Snapshot at time T_0 :

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in $Finish[i] = \text{true}$ for all i

Deadlocks (5)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems

10/16/2019



Example of Detection Algorithm

Five processes P_0 through P_4 ; three resource types A (7 instances), B (2 instances), and C (6 instances)

Snapshot at time T_0 :

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in $Finish[i] = \text{true}$ for all i

Example (Cont.)

P_2 requests an additional instance of type C

Request

A B C

P_0 0 0 0

P_1 2 0 2

P_2 0 0 1

P_3 1 0 0

P_4 0 0 2

State of system?

Can reclaim resources held by process P_0 , but insufficient resources to fulfill other processes; requests

Deadlock exists, consisting of processes P_1 , P_2 , P_3 , and P_4

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
 - Priority of the process
 - How long process has computed, and how much longer to completion
 - Resources the process has used
 - Resources process needs to complete
 - How many processes will need to be terminated
 - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- **Selecting a victim** – minimize cost
- **Rollback** – return to some safe state,
restart process for that state
- **Starvation** – same process may always
be picked as victim, include number of
rollback in cost factor

Memory Management (1)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems
10/16/2019



Multiprogramming

- Simple uniprogramming with a single segment per process
- Uniprogramming disadvantages
 - Inefficient use of CPU time
 - Inflexibility of job scheduling
- Need multiprogramming

Memory Management Requirements

- The OS must fit multiple processes in memory
 - memory needs to be subdivided to accommodate multiple processes
 - memory needs to be allocated to ensure a reasonable supply of ready processes so that the CPU is never idle
 - memory management is an **optimization** task under **constraints**

Memory Management Wish-list

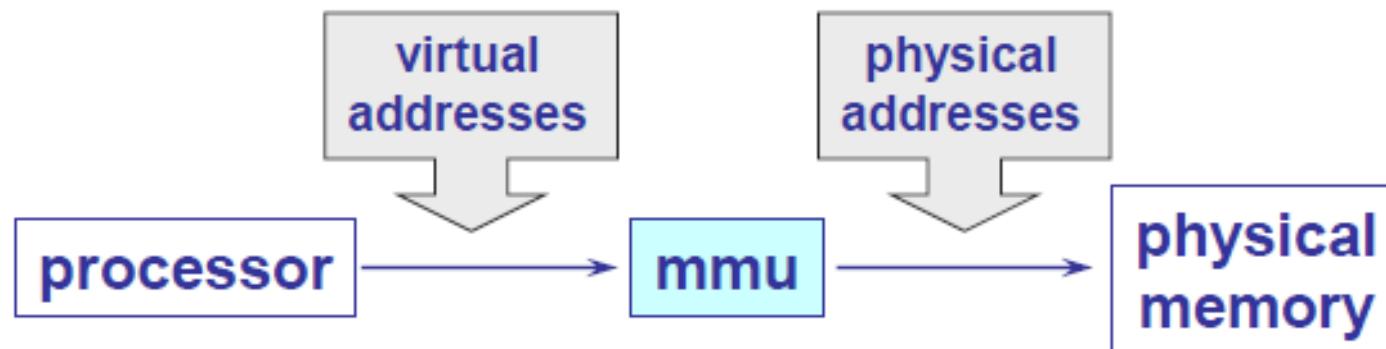
- ❑ Sharing
 - ❑ multiple processes **coexist** in main memory
- ❑ Transparency
 - ❑ Processes **are not aware** that memory is shared
 - ❑ Run **regardless of number/locations** of other processes
- ❑ Protection
 - ❑ **Cannot access** data of OS or other processes
- ❑ Efficiency: should have reasonable performance
 - ❑ Purpose of sharing is to increase efficiency
 - ❑ **Do not waste** CPU or memory resources

Virtual Addresses for Multiprogramming

- To make it easier to manage memory of multiple processes, make processes use virtual addresses (which is not what we mean by “virtual memory” today!)
- virtual addresses are independent of location in physical memory (RAM) where referenced data lives
 - OS determines location in physical memory
- instructions issued by CPU reference virtual addresses
 - e.g., pointers, arguments to load/store instructions, PC ...
- virtual addresses are translated by hardware into physical addresses (with some setup from OS)

Logical vs. Physical Address

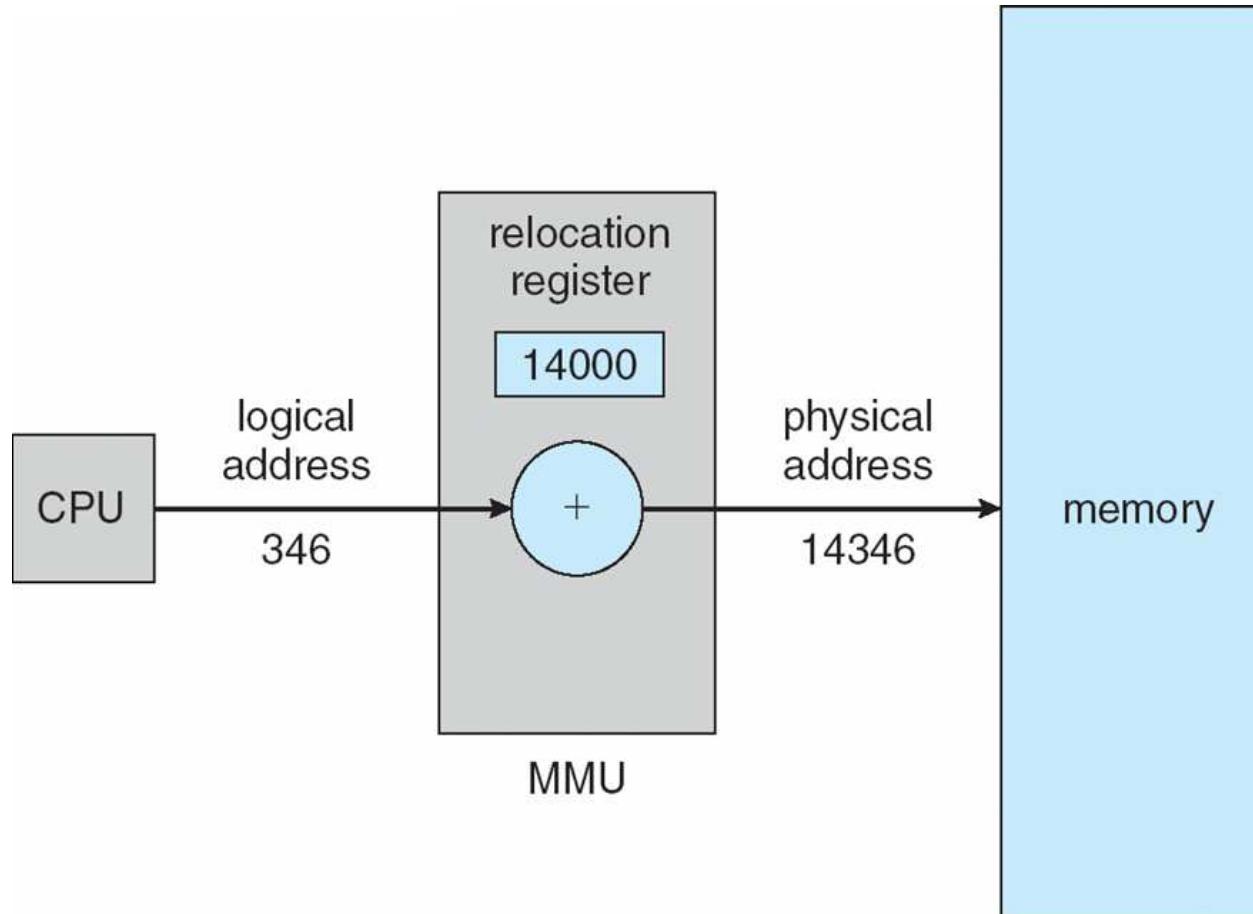
- **Logical address** – generated by the CPU; also referred to as **virtual address**
- **Physical address** – address seen by the memory unit



Old Technique #1: Fixed-Sized Partitions

- One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions
- Each partition may contain exactly one process.
- Degree of multiprogramming limited by number of partitions
- when a partition is free, a process is selected from the input queue and is loaded into the free partition.
- When the process terminates, the partition becomes available for another process.

Fixed-Sized Partitions

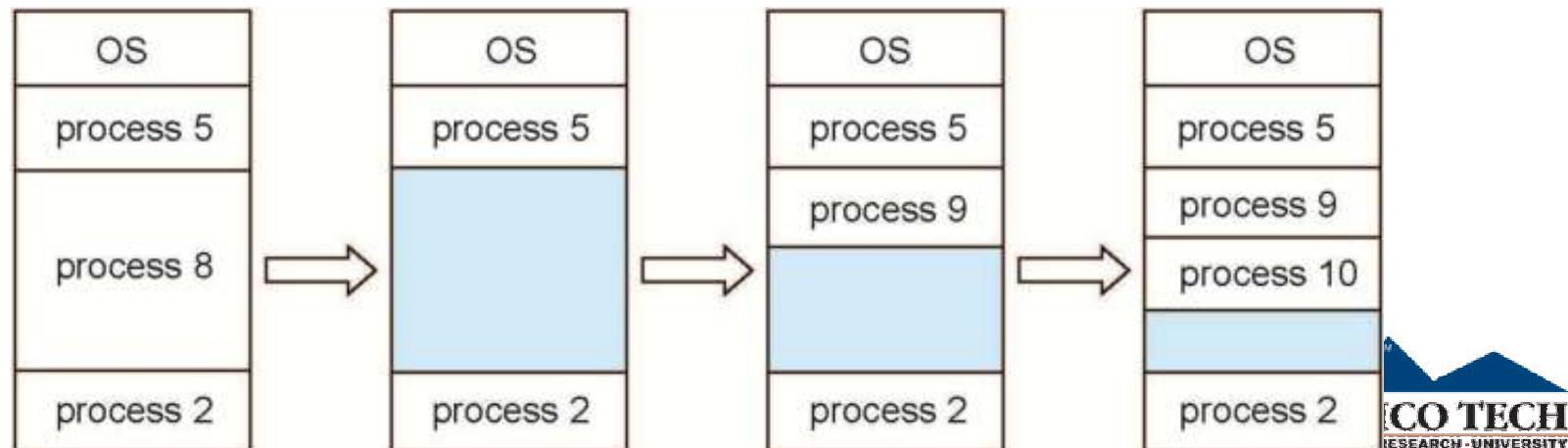


Memory Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- **Contiguous allocation** is one early method
- Main memory usually into two **partitions**:
 - Resident operating system, usually held in low memory with interrupt vector
 - User processes then held in high memory
 - Each process contained in single contiguous section of memory

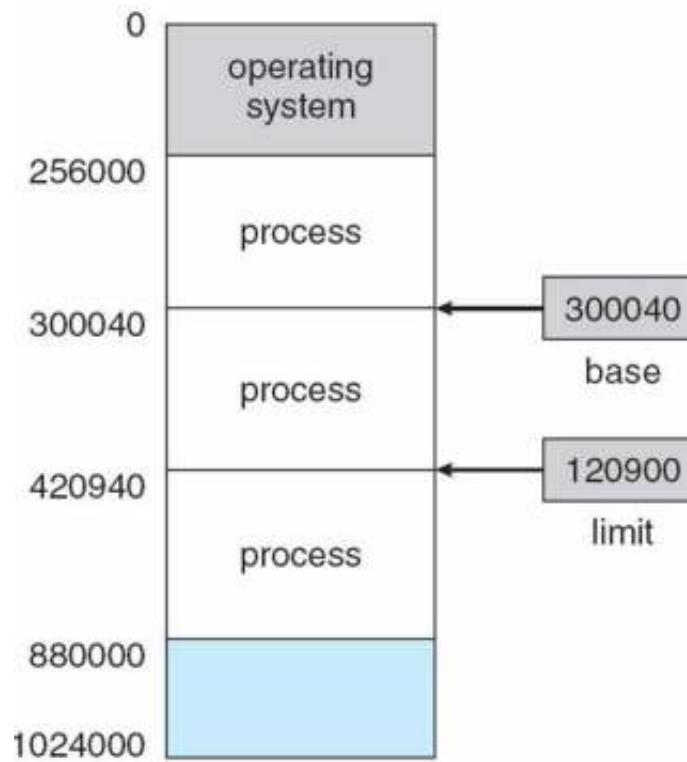
Old Technique #2: Variable Partitions

- **Variable-partition** sizes for efficiency (sized to a given process' needs)
- **Hole** – block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Process exiting frees its partition, adjacent free partitions combined
- Operating system maintains information about:
 - allocated partitions
 - free partitions (hole)

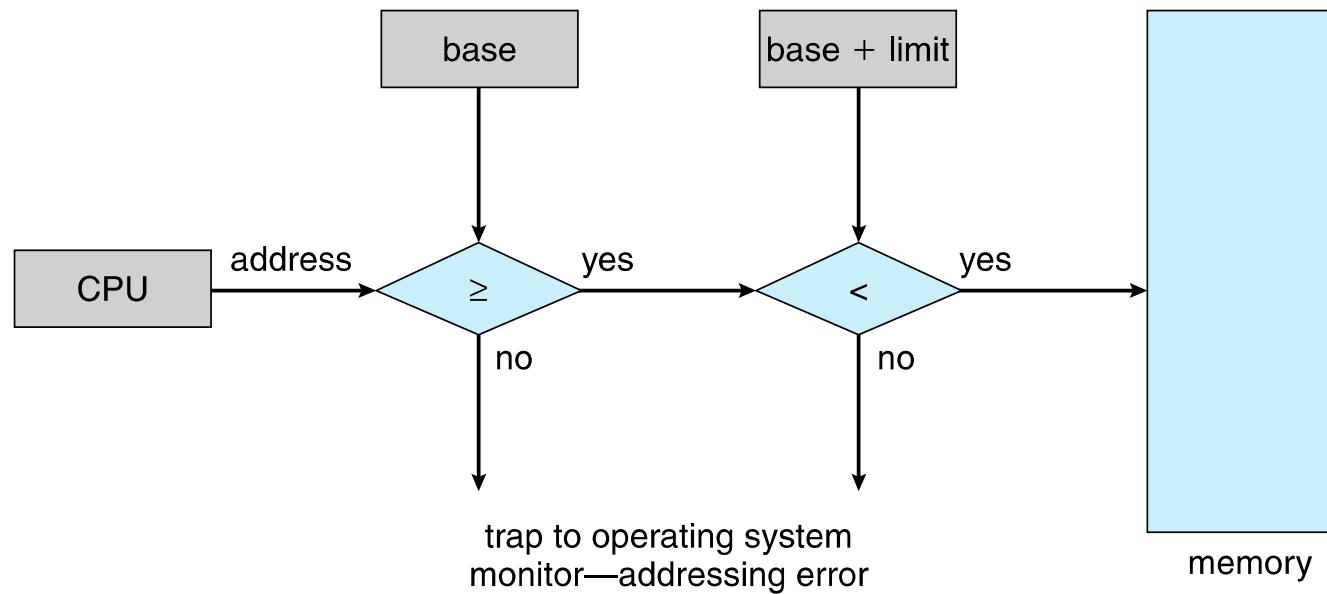


Variable Partitions

- A pair of registers provide address protection between processes:
 - **base register:** smallest legal address
 - **limit register:** size of the legal range



Hardware Address Protection



Dynamic Memory Allocation Problem

How to satisfy a request of size n from a list of free holes?

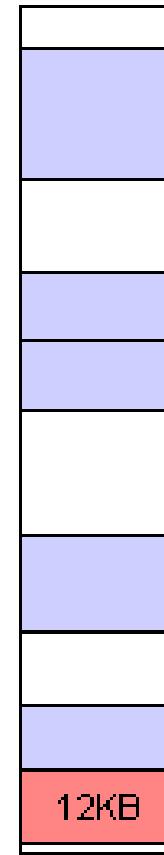
- **First-fit:** Allocate the *first* hole that is big enough
- **Best-fit:** Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
 - Produces the smallest leftover hole
- **Worst-fit:** Allocate the *largest* hole; must also search entire list
 - Produces the largest leftover hole

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

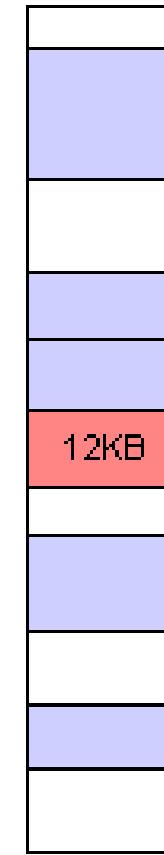
Allocation Example



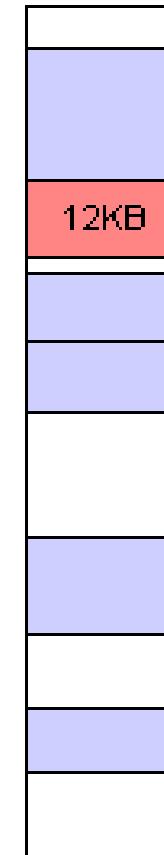
Memory



Best-fit



Worst-fit



First-fit

Memory Management (2)

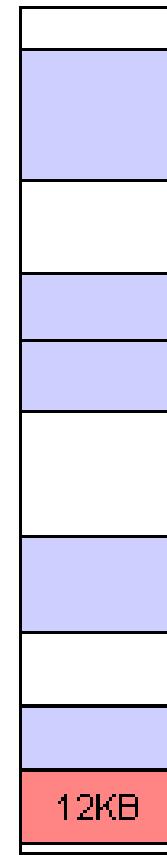
Dr. Jun Zheng
CSE325 Principles of Operating
Systems
10/21/2019



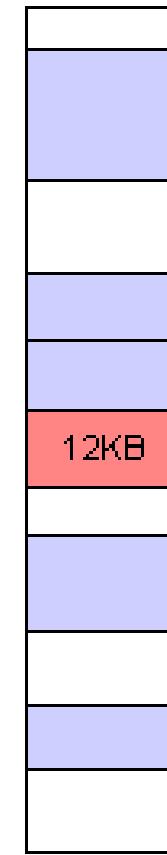
Allocation Example



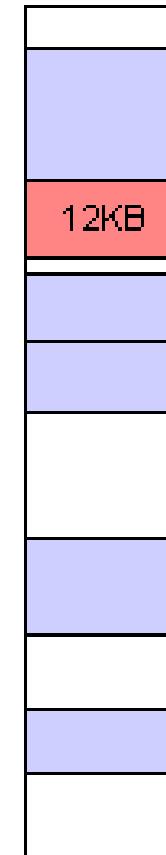
Memory



Best-fit



Worst-fit



First-fit

In-class Work 5

Give five memory partitions of 100KB, 500KB, 200KB, 300KB and 600KB (in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 212KB, 417KB, 112KB, and 426KB (in order)? Which algorithm makes the most efficient use of memory?

Fragmentation

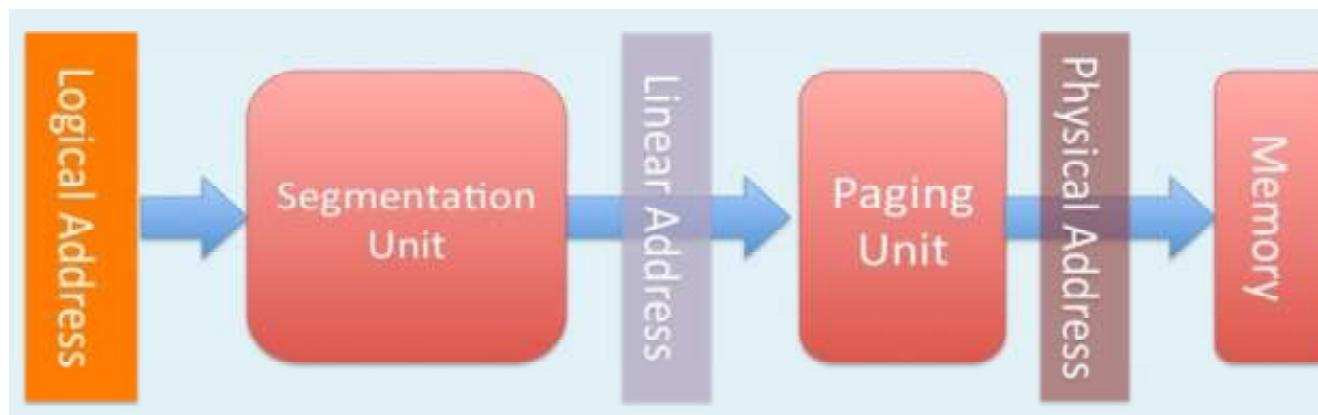
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

Fragmentation (Cont.)

- Reduce external fragmentation by **compaction**
 - Shuffle memory contents to place all free memory together in one large block
 - Compaction is possible *only* if relocation is dynamic, and is done at execution time
- Another solution is to allow the logical address space of the processes to be **noncontiguous**.

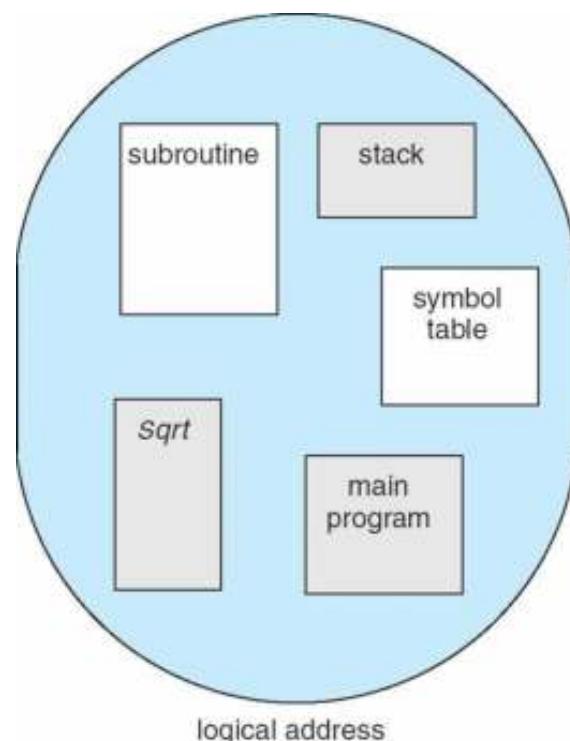
x86 Address Translation

- CPU generates virtual address (*seg, offset*)
 - Given to segmentation unit
 - Which produces linear addresses
 - Linear address given to paging unit
 - Which generates physical address in main memory

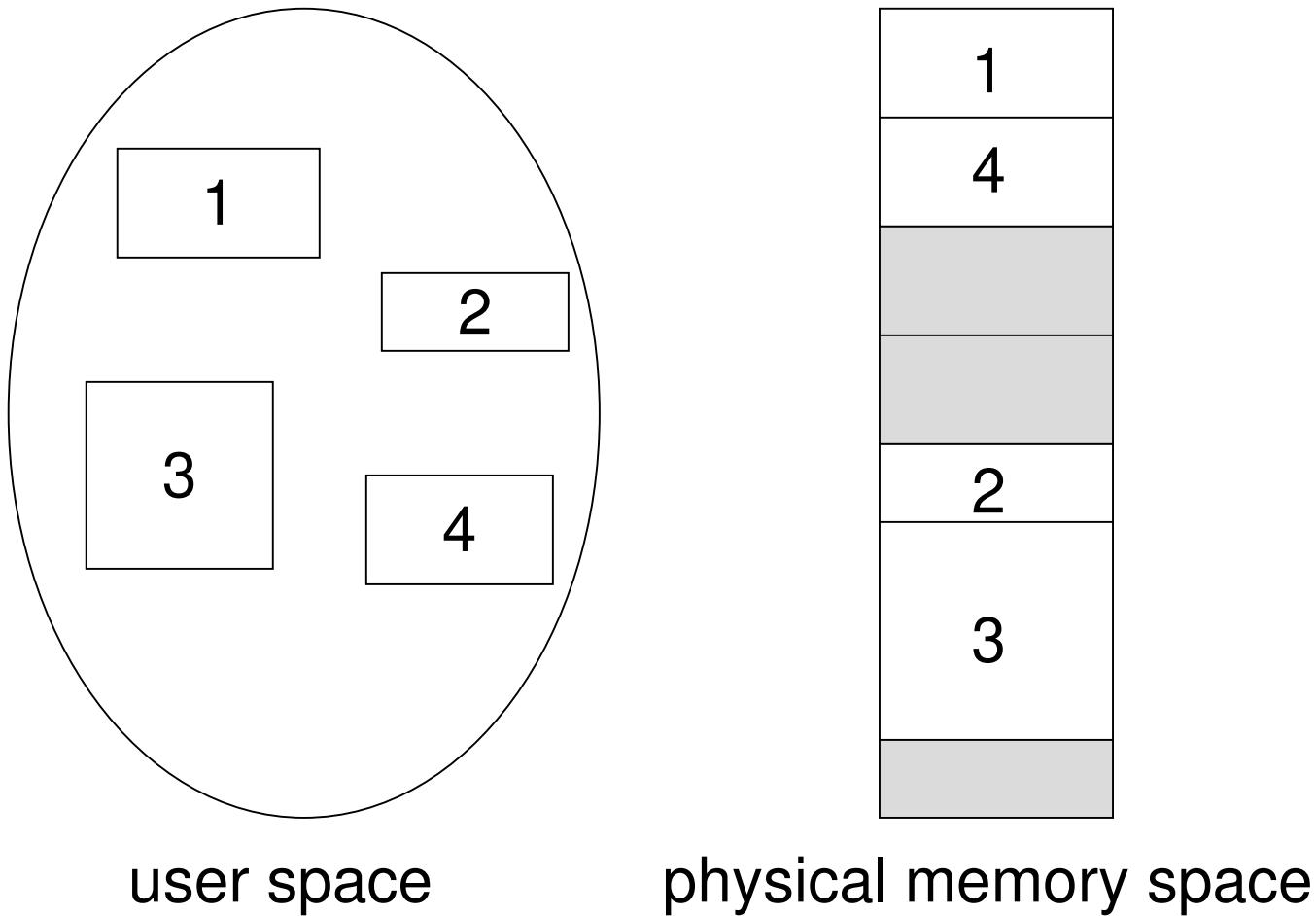


Segmentation

- Divide virtual address space into separate logical segments; each is part of physical memory.
- A natural extension of variable-sized partition
 - variable-sized partition = 1 segment/process
 - segmentation = many segments/process



Logical View of Segmentation



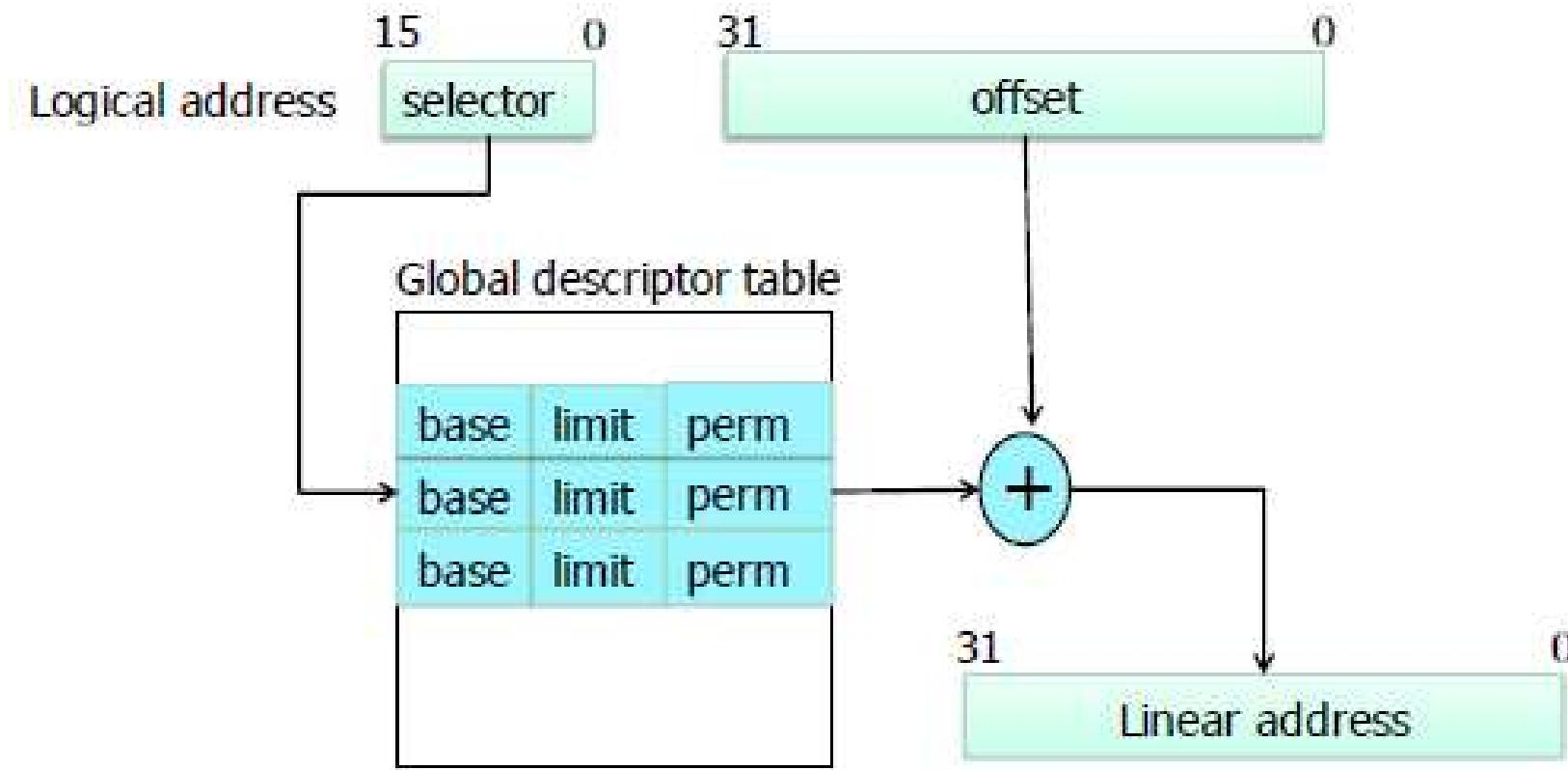
Segmentation Translation

- Virtual address: <segment-number,
offset>
- Segment table maps segment number to
segment information
 - **Base**: starting address of the segment in
physical memory
 - **Limit**: length of the segment
 - Additional metadata includes protection
bits (validation bit, r/w/e privileges etc.)
- Limit & protection checked on each access

x86 Segment Selector

- Logical address: segment selector + offset
- Segment selector stored in segment registers (16-bit)
 - `cs`: code segment selector
 - `ss`: stack segment selector
 - `ds`: data segment selector
 - `es, fs, gs`: extra
- Segment register can be implicitly or explicitly specified
 - Implicit by type of memory reference (`jmp`)
 - `mov $8049780, %eax` // implicitly use `ds`
 - Through special registers (`cs, ss, ds, es, fs, gs` on x86)
 - `mov %ss:$8049780, %eax` // explicitly use `ss`
- Support for segmentation removed in x86-64
 - `cs, ss, ds`, and `es` are forced to 0

x86 Segmentation Hardware



xv6 Segments

- ❑ `vm.c, ksegment()`
- ❑ Rely mainly on paging (like Linux)
- ❑ Kernel code: readable + executable in kernel mode
- ❑ Kernel data: writeable in kernel mode
- ❑ User code: readable + executable in user mode
- ❑ User data: writable in user mode
- ❑ These are all null mappings
 - ❑ Map to [0, 0xFFFFFFFF]
 - ❑ linear address = offset

Memory Management (3)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems
10/23/2019



Pros and Cons of Segmentation

❑ Advantages

- ❑ Segment sharing
- ❑ Easier to relocate segment than entire program
- ❑ Avoid allocating unused memory
- ❑ Flexible protection
- ❑ Efficient translation
 - ❑ Segment table small -> fit in MMU

❑ Disadvantages

- ❑ Segments have variable lengths -> How to fit
- ❑ External fragmentation: wasted memory

Paging

- ❑ Physical address space of a process can be noncontiguous; process is allocated fixed sized units of physical memory whenever the latter is available
 - ❑ **Avoids external fragmentation**
 - ❑ Avoids problem of varying sized memory chunks
- ❑ Still have Internal fragmentation

Paging

- Divide physical memory into fixed-sized blocks called **frames**
 - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size N pages, need to find N free frames and load program
- Set up a **page table** to translate logical to physical addresses

Address Translation Scheme

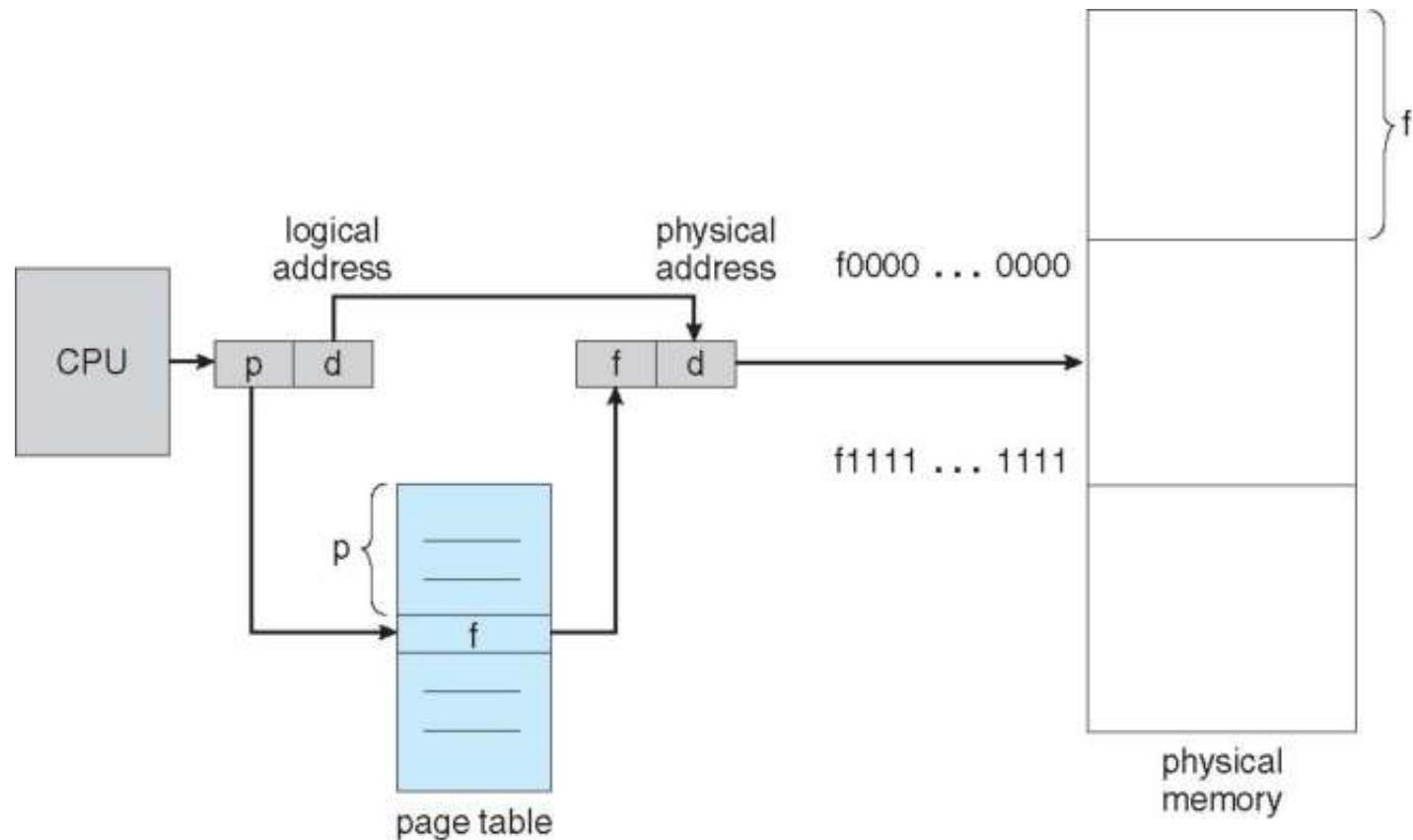
- Address generated by CPU is divided into:
 - **Page number (p)** – used as an index into a **page table** which contains base address of each page in physical memory
 - **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit

page number	page offset
p	d

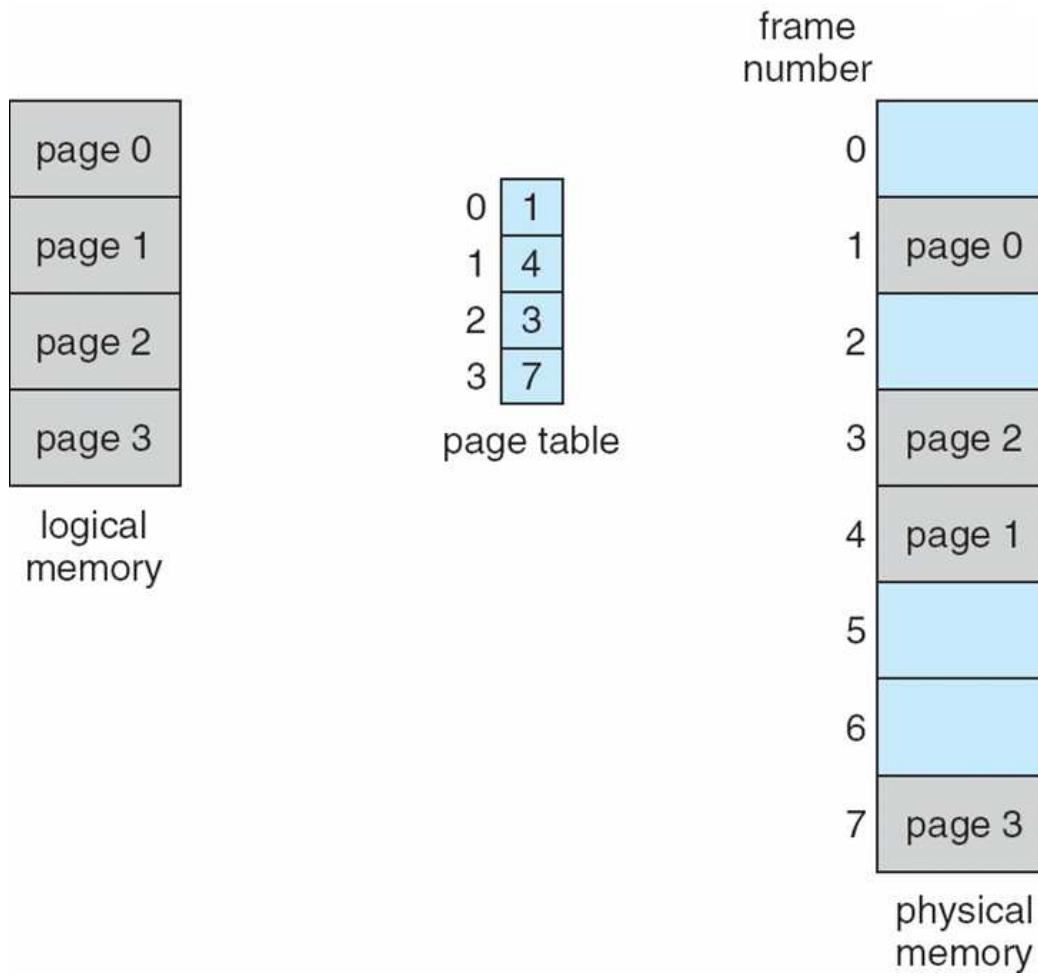
$m - n$ n

- For given logical address space 2^m and page size 2^n

Page Translation



Page Translation Example



Page Translation Exercise

- ❑ 8-bit virtual address, 10-bit physical address, and each page is 64 bytes
 - ❑ How many virtual pages?
 - ❑ How many physical pages?
 - ❑ How many entries in page table?
 - ❑ Given page table = [2, 5, 1, 8], what's the physical address for virtual address 241?
- ❑ m-bit virtual address, n-bit physical address, k-bit page size
 - ❑ What are the answers to the first three questions above?

Internal Fragmentation

- Calculating internal fragmentation
 - Page size = 2,048 bytes
 - Process size = 72,766 bytes
 - 35 pages + 1,086 bytes
 - Internal fragmentation of $2,048 - 1,086 = 962$ bytes
 - Worst case fragmentation = 1 frame – 1 byte
 - On average fragmentation = $1 / 2$ frame size
- So small frame sizes desirable?
 - But each page table entry takes memory to track
 - Page sizes growing over time
 - Solaris supports two page sizes – 8 KB and 4 MB

Page Protection

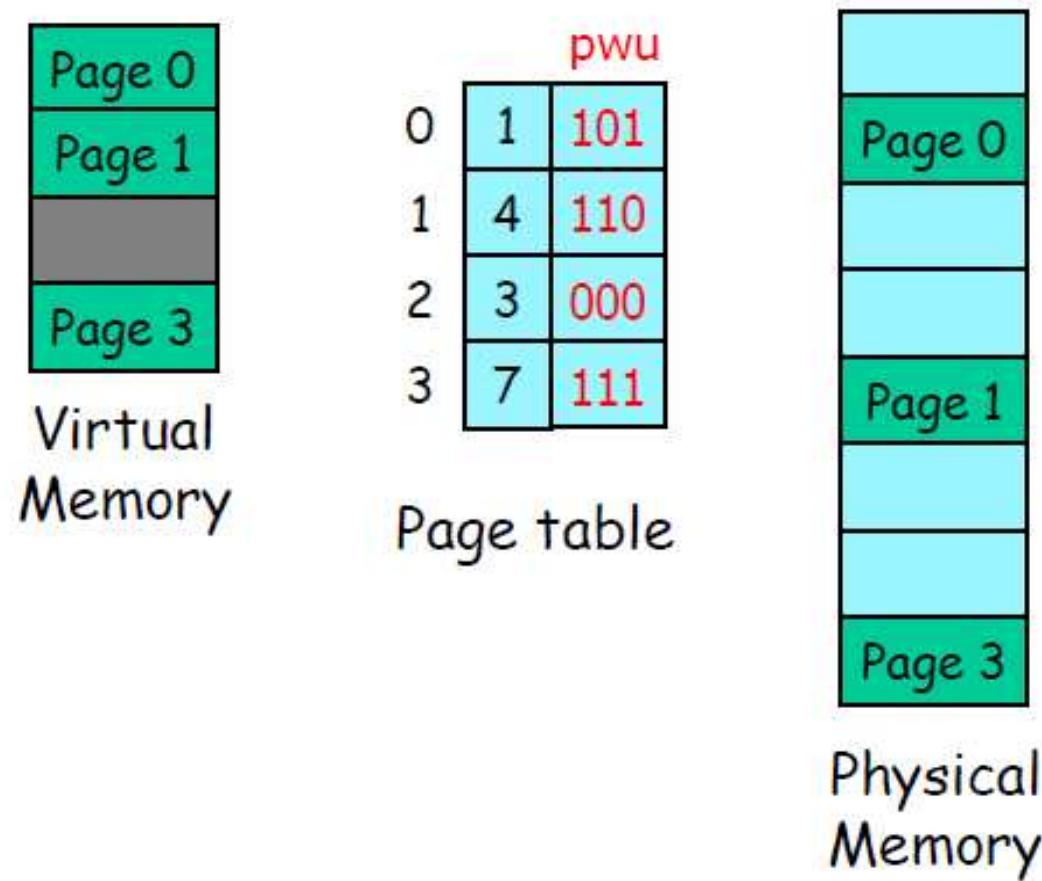
- Implemented by associating protection bits with each virtual page in page table
- Protection bits
 - present bit: map to a valid physical page?
 - read/write/execute bits: can read/write/execute?
 - user bit: can access in user mode?
 - x86: PTE_P, PTE_W, PTE_U
- Checked by MMU on each memory access

Memory Management (4)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems
10/25/2019



Page Protection Example



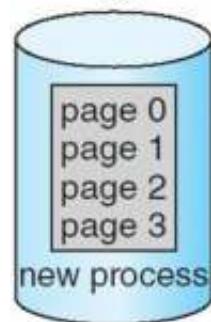
Page Allocation

- ❑ Free frame management
 - ❑ E.g., can put page on a free list
- ❑ Allocation policy
 - ❑ E.g., one page at a time, from head of free list
- ❑ xv6: `kalloc.c`

Page Allocation

free-frame list

14
13
18
20
15

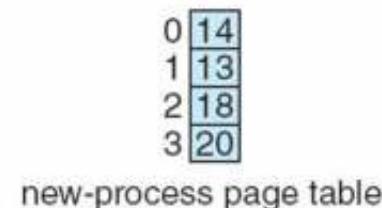
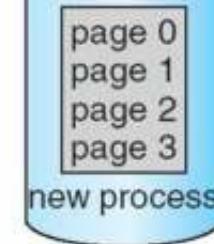


(a)

Before allocation

free-frame list

15



new-process page table

(b)

After allocation

Implementation of Page Table

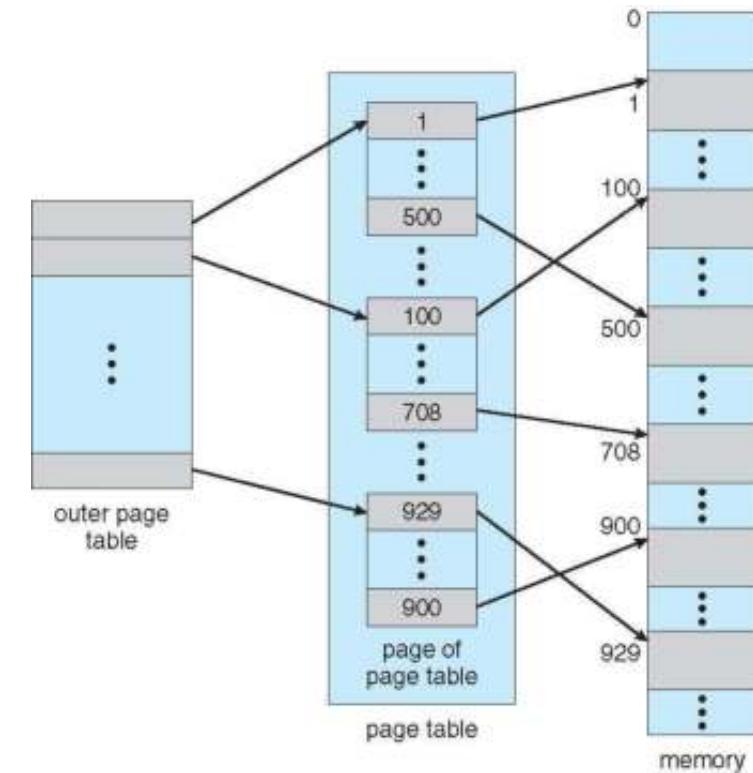
- Page table is kept in main memory
 - **Page-table base register (PTBR)** points to the page table
 - x86: **cr3**
 - **Page-table length register (PTLR)** indicates size of the page table
 - OS stores base in process control block (PCB)
 - OS switches PTBR on each context switch
- In this scheme every data/instruction access requires two memory accesses
 - One for the page table and one for the data / instruction
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**

Structure of the Page Table

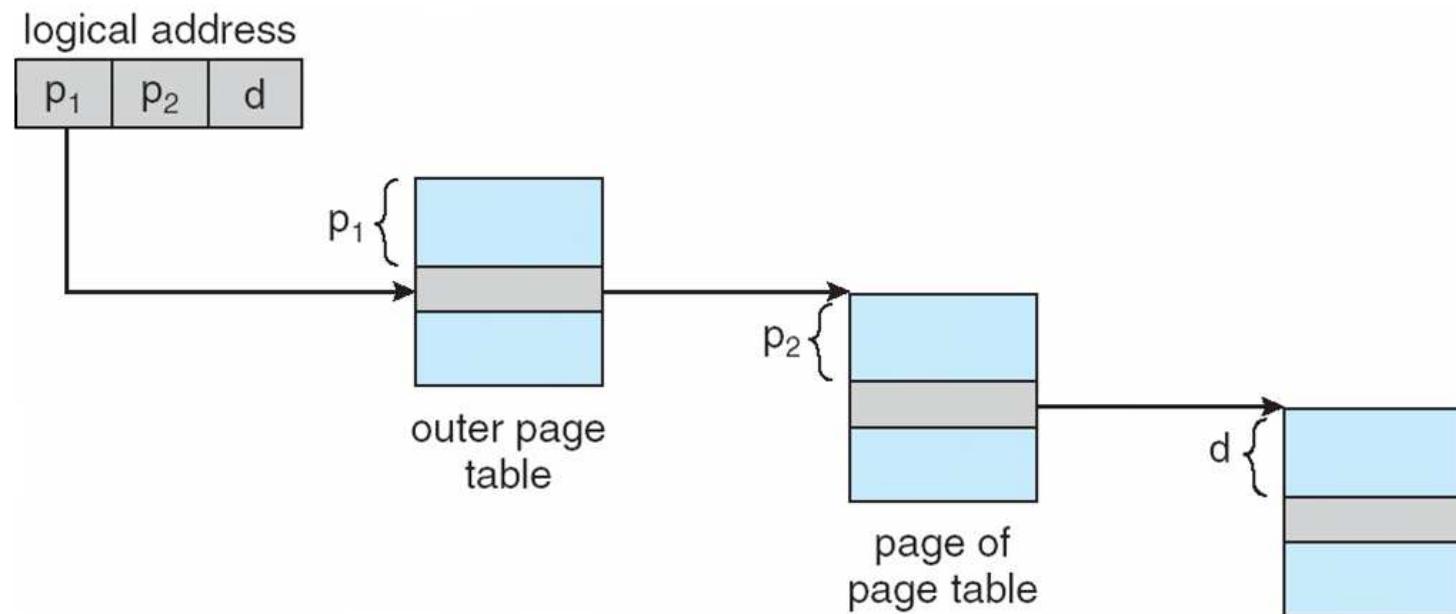
- Memory structures for paging can get huge using straight-forward methods
 - Consider a 32-bit logical address space as on modern computers
 - Page size of 4 KB (2^{12})
 - Page table would have 1 million entries ($2^{32} / 2^{12}$)
 - If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
 - That amount of memory used to cost a lot
 - Don't want to allocate that contiguously in main memory
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table



Address Translation with Hierarchical Page Table

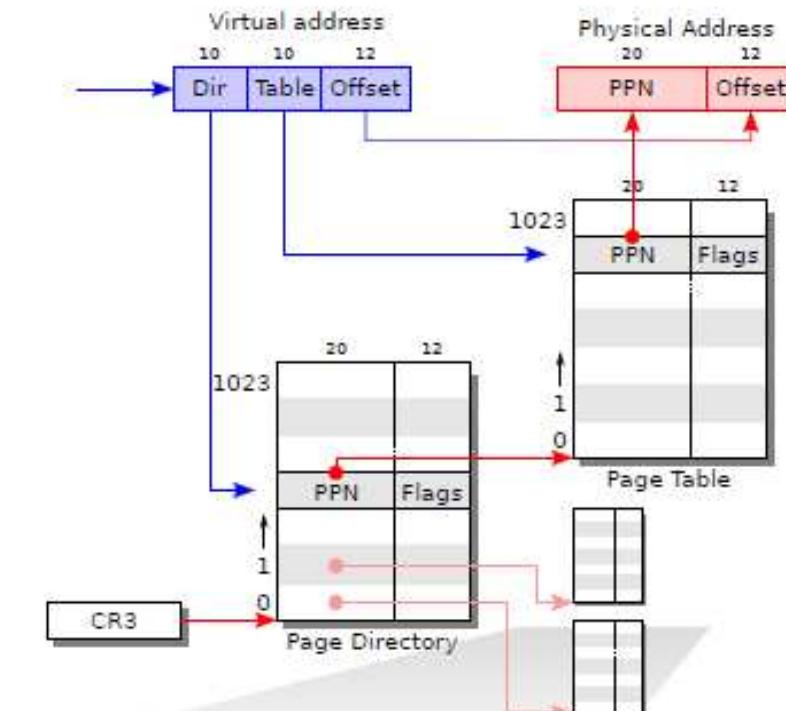
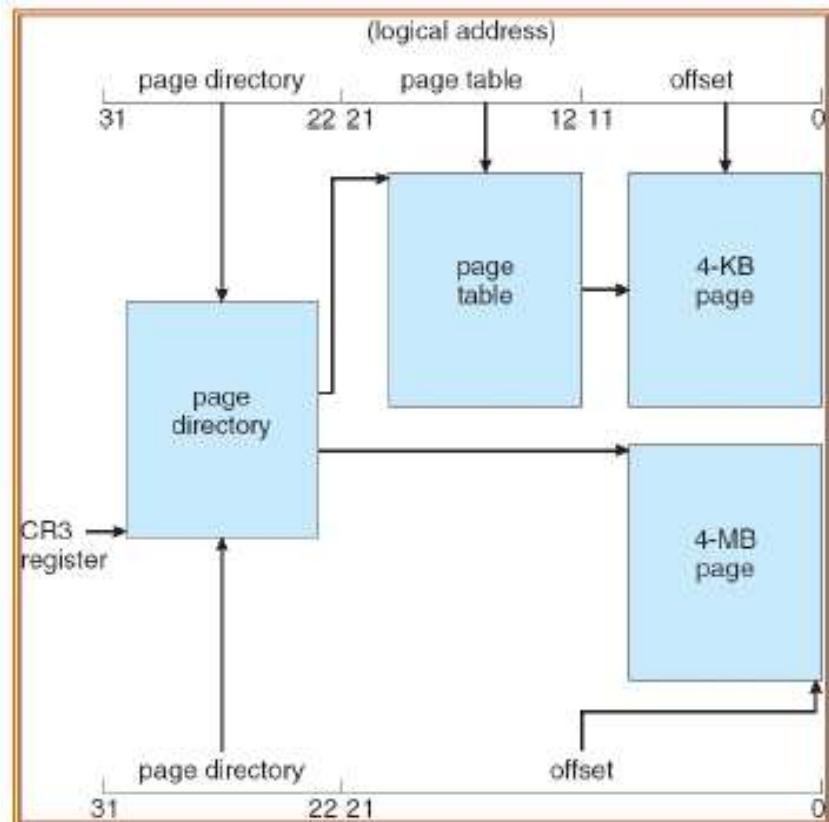


x86 Page Translation

- ❑ 32-bit address space, 4 KB page
 - ❑ 4KB page -> 12 bits for page offset
- ❑ How many bits for 2nd-level page table?
 - ❑ Desirable to fit a 2nd-level page table in one page
 - ❑ $4\text{KB}/4\text{B} = 1024 \rightarrow 10$ bits for 2nd-level page table
- ❑ Address bits for top-level page table: $32 - 12 - 10 = 10$

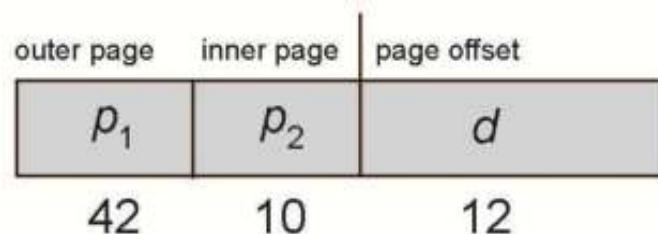
page number	page offset
p_1	p_2
10	10 12

x86 Paging Architecture



64-bit Logical Address Space

- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
 - Then page table has 2^{52} entries
 - If two level scheme, inner page tables could be 2^{10} 4-byte entries
 - Address would look like

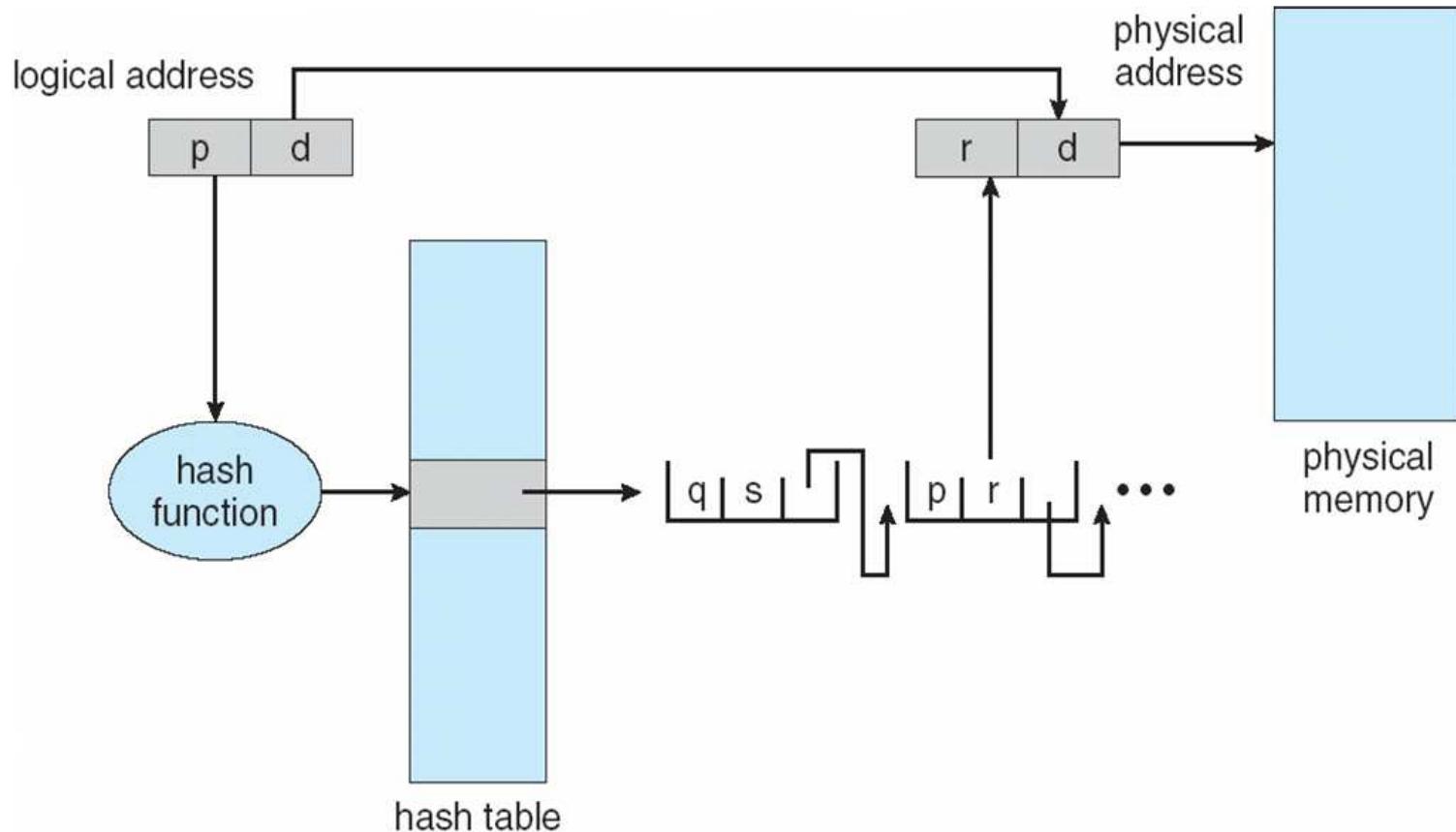


- Outer page table has 2^{42} entries or 2^{44} bytes
- One solution is to add a 2nd outer page table
- But in the following example the 2nd outer page table is still 2^{34} bytes in size
 - And possibly 4 memory access to get to one physical memory location

Hashed Page Tables

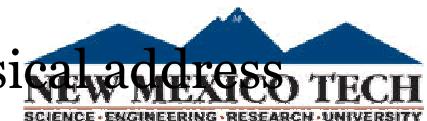
- Common in address spaces > 32 bits
- The logical page number is hashed into a page table
 - This page table contains a chain of elements hashing to the same location
- Each element contains (1) the logical page number (2) the value of the mapped page frame (3) a pointer to the next element
- Logical page numbers are compared in this chain searching for a match
 - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
 - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
 - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

Hashed Page Table

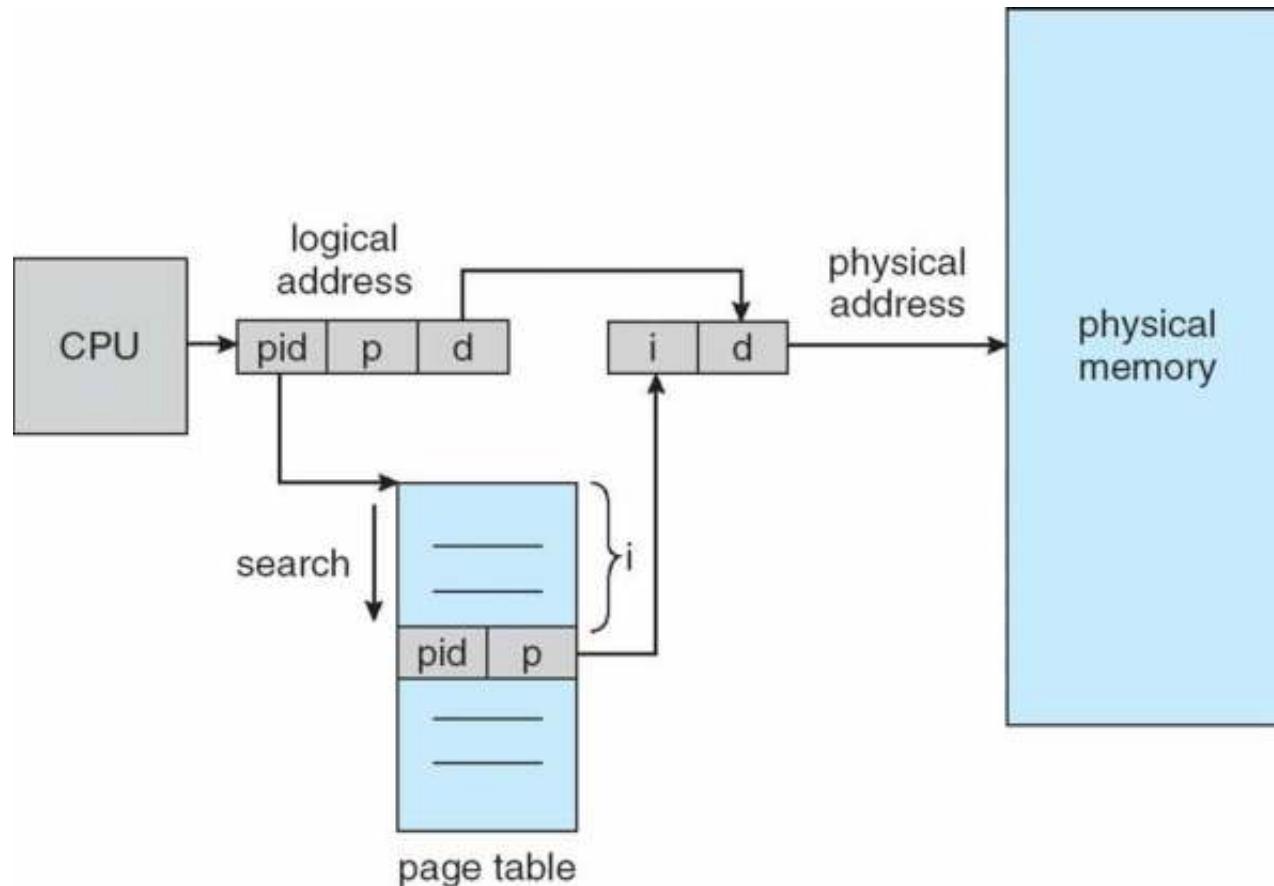


Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
 - TLB can accelerate access
- But how to implement shared memory?
 - One mapping of a logical address to the shared physical address



Inverted Page Table Architecture

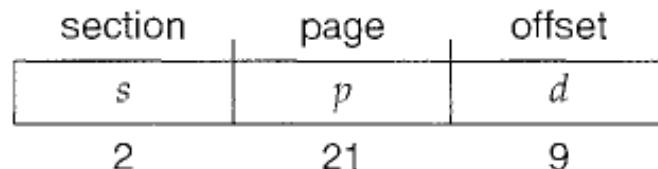


Example

The VAX architecture supports a variation of two-level paging.

The VAX is a 32-bit machine with a page size of 512 bytes.

The logical address space of a process is divided into four equal sections, each of which consists of 2^{30} bytes. Each section represents a different part of the logical address space of a process. The first 2 high-order bits of the logical address designate the appropriate section. The next 21 bits represent the logical page number of that section, and the final 9 bits represent an offset in the desired page.



How many memory operations are performed when a user program executes a memory-load operation?

Example

Consider a computer system with a 32-bit logical address and 4-KB (2^{12} B) page size. The system supports up to 512MB (2^{29} B) of physical memory. How many entries are there in each of the following?

- (1) A conventional single-level page table
- (2) An inverted page table

Answer

(1) # of pages= # of entries =????

Size of logical address space = 2^m = # of pages × page size

»» 2^{32} = # of pages × 2^{12}

of pages = $2^{32} / 2^{12} = 2^{20}$ pages

(2)

Size of physical address space = # of frames × frame size

(frame size = page size)

For inverted page table, # of frames = # of entries

2^{29} = # of frames × 2^{12}

of frames = $2^{29} / 2^{12} = 2^{17}$

of entries = 2^{17}

Memory Management (5)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems
11/4/2019

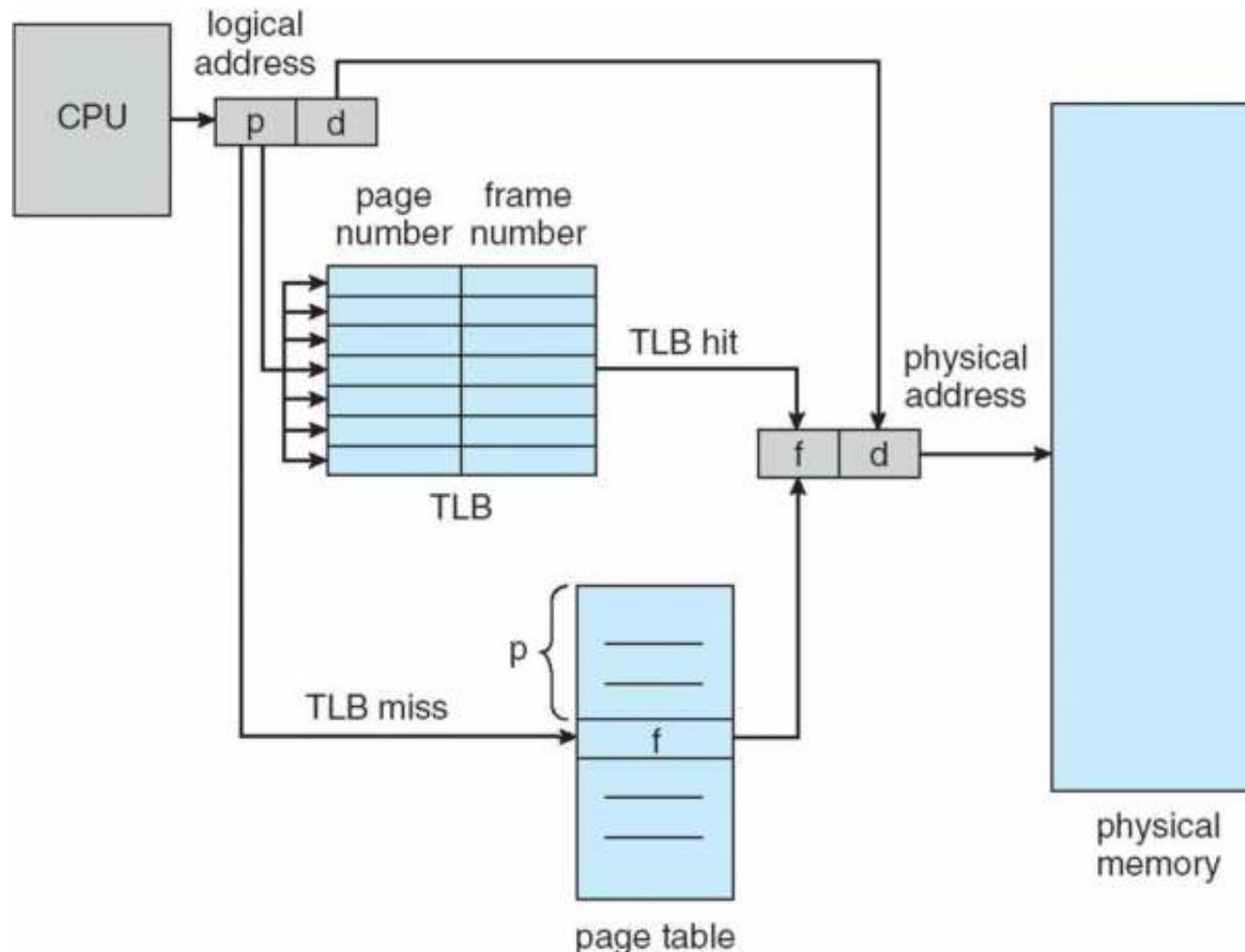


TLB

- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process
 - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
 - Replacement policies must be considered
 - Some entries can be **wired down** for permanent fast access



Paging Hardware With TLB



Effective Access Time

- Associative Lookup = ε time unit
 - Can be < 10% of memory access time
- Hit ratio = α
 - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Consider $\alpha = 80\%$, $\varepsilon = 20\text{ns}$ for TLB search, 100ns for memory access
- **Effective Access Time (EAT)**
$$\text{EAT} = (100 + \varepsilon) \alpha + (200 + \varepsilon)(1 - \alpha) = 200 + \varepsilon - 100\alpha$$
- Consider $\alpha = 80\%$, $\varepsilon = 20\text{ns}$ for TLB search, 100ns for memory access
 - $\text{EAT} = 0.80 \times 120 + 0.20 \times 220 = 140\text{ns}$
- Consider more realistic hit ratio -> $\alpha = 99\%$, $\varepsilon = 20\text{ns}$ for TLB search, 100ns for memory access
 - $\text{EAT} = 0.99 \times 120 + 0.01 \times 220 = 122\text{ns}$

Virtual Memory (1)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems

11/4/2019



Background

- Code needs to be in memory to execute, but entire program rarely used
 - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
 - Program no longer constrained by limits of physical memory
 - Each program takes less memory while running -> more programs run at the same time
 - Increased CPU utilization and throughput with no increase in response time or turnaround time
 - Less I/O needed to load or swap programs into memory -> each user program runs faster

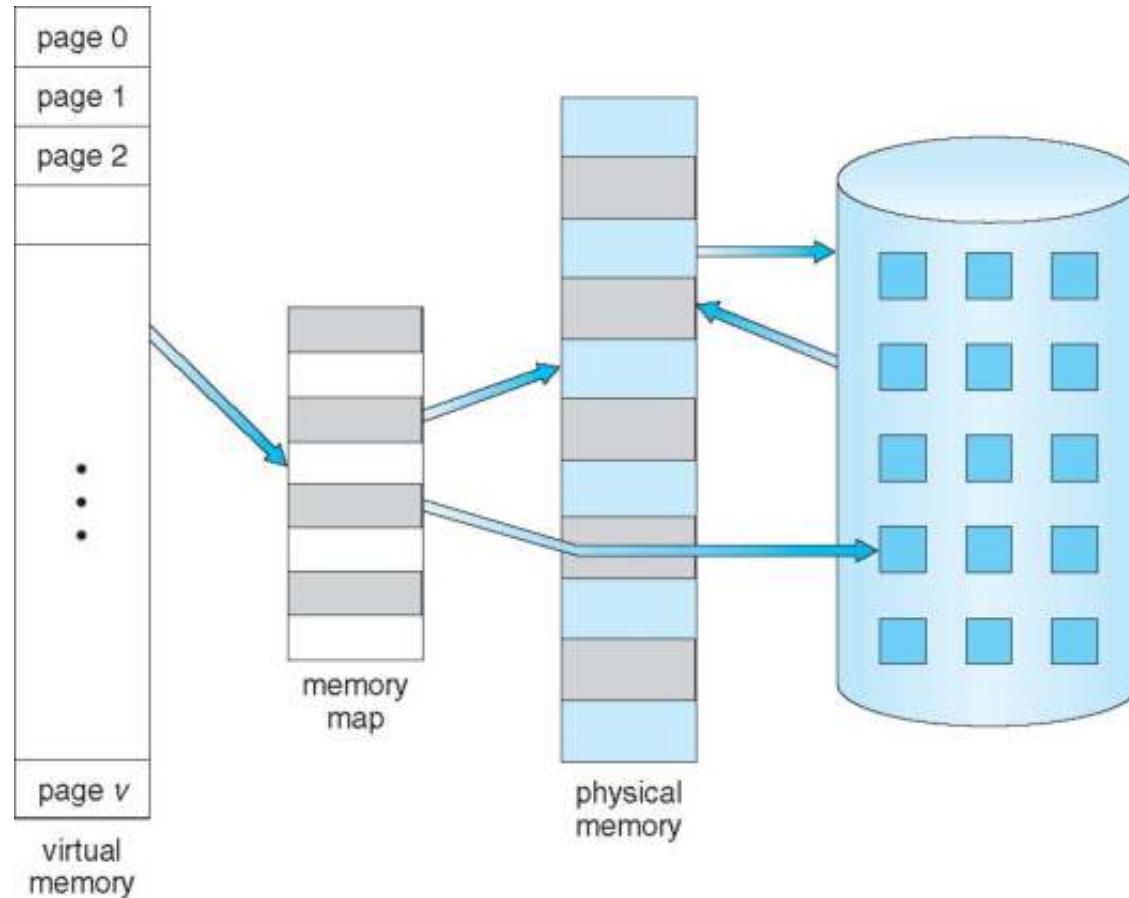
Background (Cont.)

- **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

Background (Cont.)

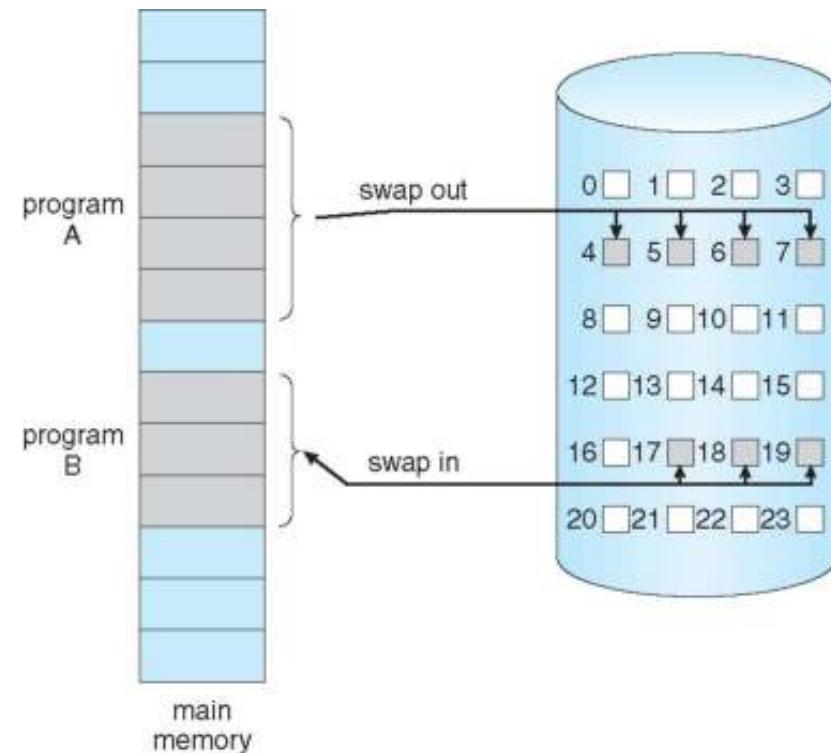
- **Virtual address space** – logical view of how process is stored in memory
 - Usually start at address 0, contiguous addresses until end of space
 - Meanwhile, physical memory organized in page frames
 - MMU must map logical to physical
- Virtual memory can be implemented via demand paging: only bring in pages actually used

Virtual Memory That is Larger Than Physical Memory



Demand Paging

- ❑ Bring a page into memory only when it is needed
 - ❑ Less I/O needed, no unnecessary I/O
 - ❑ Less memory needed
 - ❑ Faster response
 - ❑ More users
- ❑ **Lazy swapper** – never swaps a page into memory unless page will be needed
 - ❑ Swapper that deals with pages is a **pager**



Basic Concepts

- ❑ With swapping, pager guesses which pages will be used before the process is swapped out again
 - ❑ Instead of swapping in a whole process, pager brings in only those pages into memory
- ❑ How to determine that set of pages?
 - ❑ Need new MMU functionality to implement demand paging
- ❑ If pages needed are already **memory resident**
 - ❑ No difference from non demand-paging
- ❑ If page needed and not memory resident
 - ❑ Need to detect and load the page into memory from storage
 - ❑ Without changing program behavior
 - ❑ Without programmer needing to change code

Valid-Invalid Bit

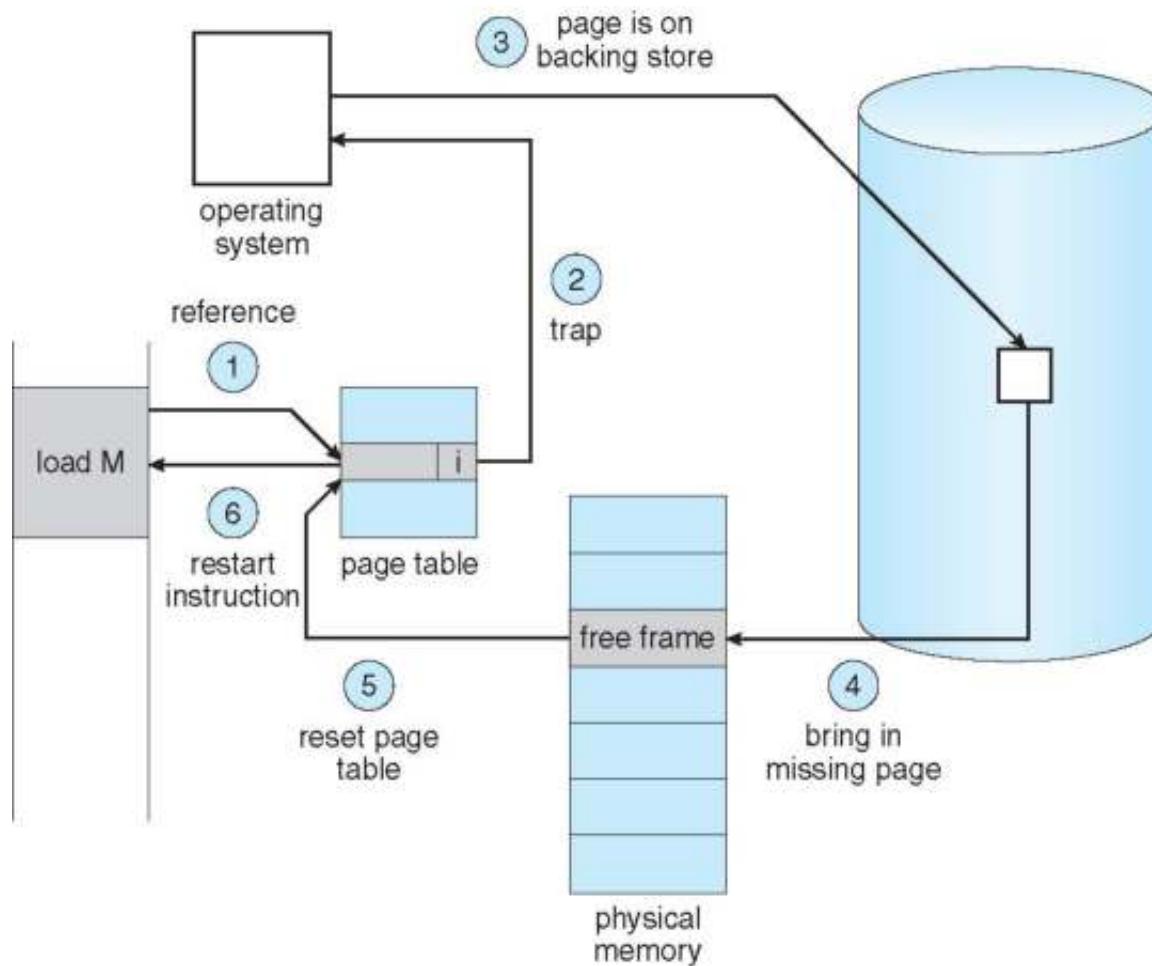
- With each page table entry a valid–invalid bit is associated ($v \Rightarrow$ in-memory – **memory resident**, $i \Rightarrow$ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

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

page table

- During MMU address translation, if valid–invalid bit in page table entry is **i** \Rightarrow **page fault**

Steps in Handling a Page Fault



Aspects of Demand Paging

- 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**
- Actually, a given instruction could access multiple pages -> multiple page faults
 - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
 - 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

Virtual Memory (2)

Dr. Jun Zheng

CSE325 Principles of Operating
Systems

11/6/2019



Performance of Demand Paging

Page Fault Rate

- $0 \leq p \leq 1.0$
- if $p = 0$ no page faults
- if $p = 1$, every reference is a fault

Effective Access Time (EAT)

$$\begin{aligned} EAT = & (1 - p) \times \text{memory access} \\ & + p (\text{page fault overhead} \\ & \quad + \text{swap page out} \\ & \quad + \text{swap page in} \\ & \quad + \text{restart overhead}) \end{aligned}$$

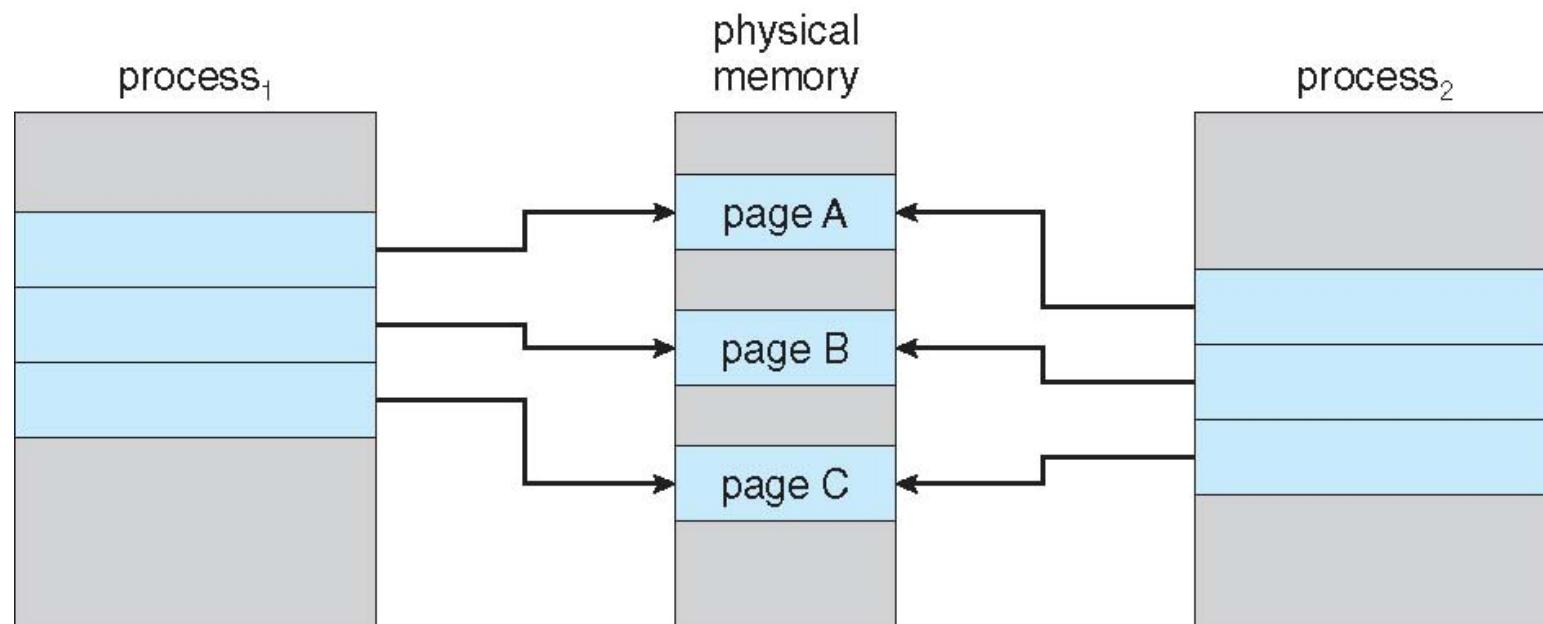
Demand Paging Example

- ❑ Memory access time = 200ns
- ❑ Average page fault service time = 8ms
- ❑ $EAT = (1-p)*200 + p*8,000,000 = 200 + p*7,999,800$
- ❑ If one access out of 1000 causes a page fault, then $EAT = 8.2\mu s$
- ❑ A slowdown by a factor of 40

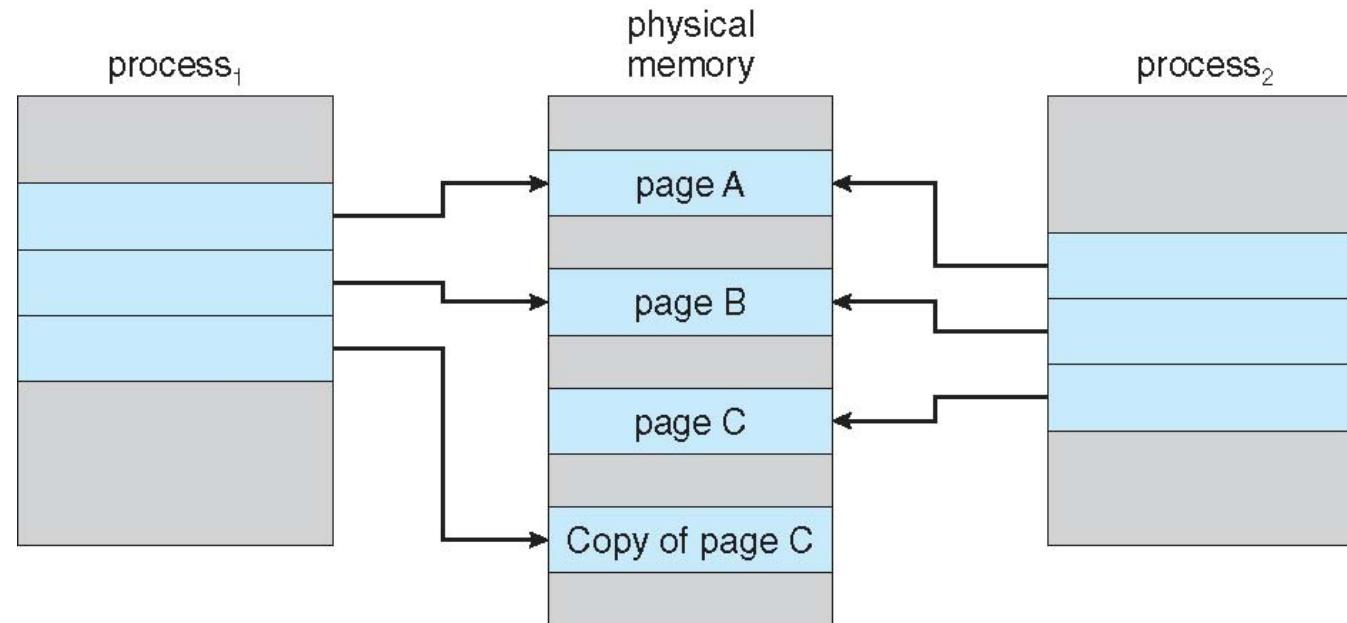
Copy-on-Write

- **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



What Happens if There is no Free Frame?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out
 - Performance – want an algorithm which will result in minimum number of page faults

Page Replacement

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

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 (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

Page Replacement Algorithms

- Want lowest page-fault rate on both first access and re-access
- 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
 - Results depend on number of frames available
- In the examples if not specified, the **reference string** of referenced page numbers is

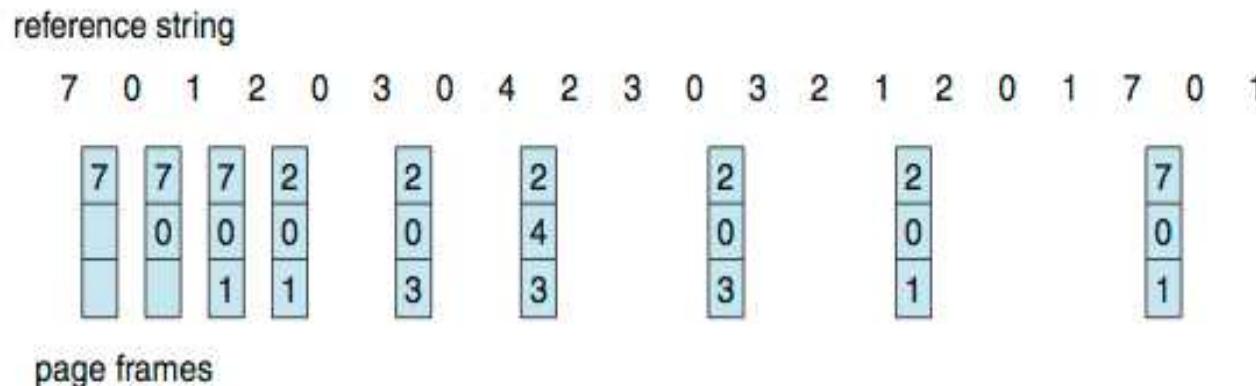
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Random Replacement

- Replaced page is chosen from m loaded frames with probability $1/m$
- Easy to implement but does not perform well

Optimal Algorithm

- Replace page that will not be used for longest period of time



- How do you know this?
 - Can't read the future
- Used for measuring how well your algorithm performs

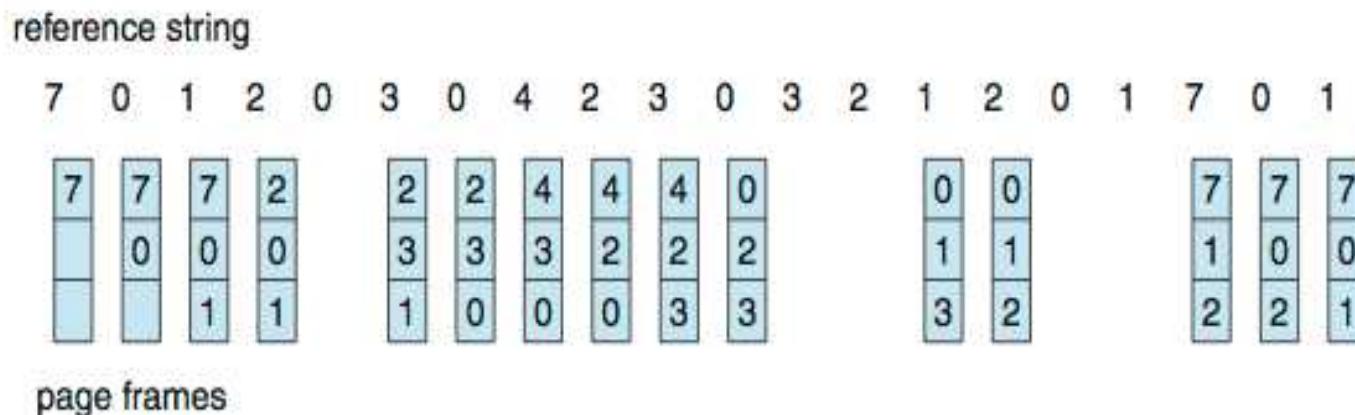
First-In-First-Out (FIFO)

Algorithm

- Reference string:

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

- 3 frames (3 pages can be in memory at a time per process)

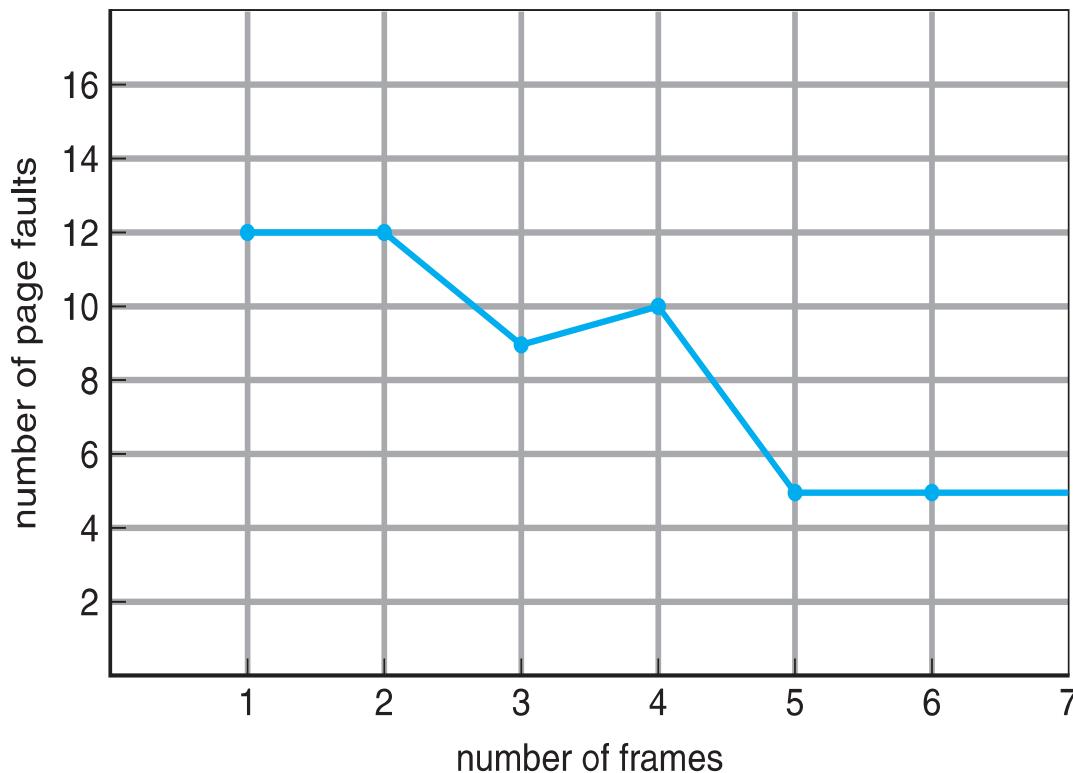


15 page faults

- What if we have 4 frames?
- Consider reference string: 1 2 3 4 1 2 5 1 2 3 4 5

Belady's Anomaly

- Adding more frames can cause more page faults!



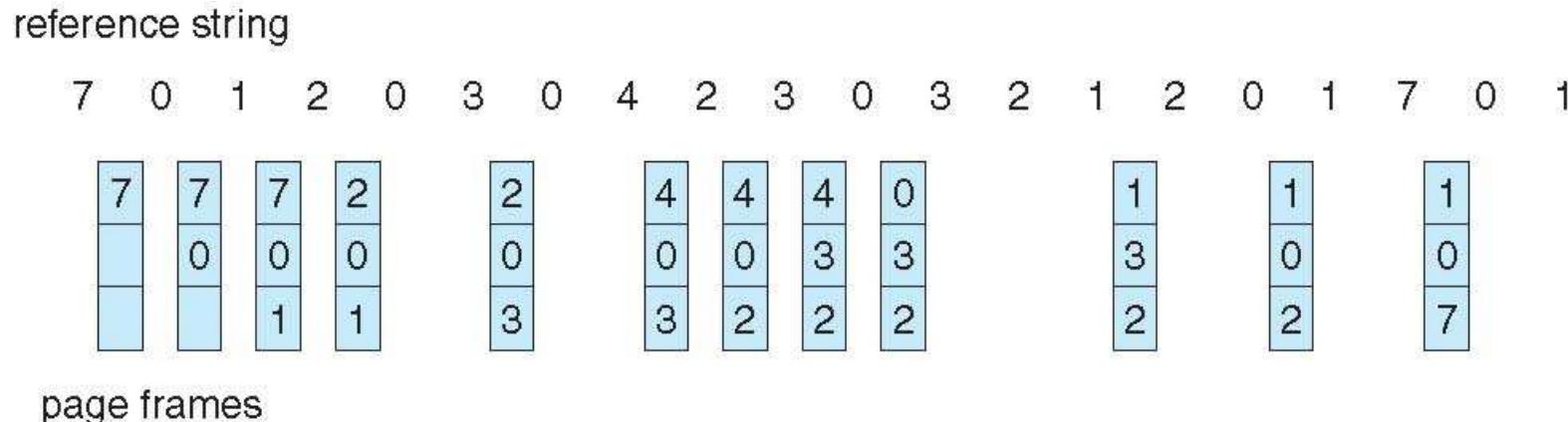
Virtual Memory (3)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems
11/8/2019



Least Recently Used (LRU) Algorithm

- 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 Algorithm (Cont.)

- Counter implementation
 - Every page-table entry has a time-of-use field and the CPU has a logic clock or counter;
 - The clock is incremented for every time a page is referenced and the content of the clock register is copied to the time-of-use field in the page-table entry
 - When a page needs to be replaced, look at the counters to find smallest value
 - Search through table needed
 - Overflow of the clock must be considered

LRU Algorithm (Cont.)

- Stack implementation
 - Keep a stack of page numbers (use a doubly linked list)
 - Page referenced, move it to the top
 - The LRU page is always at the bottom of the stack
 - Update is more expensive (at worst changing six pointers) but no search for replacement
- LRU and OPT are cases of **stack algorithms** that don't have Belady's anomaly
 - A stack algorithm is 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.

Use Of A Stack to Record Most Recent Page References

reference string

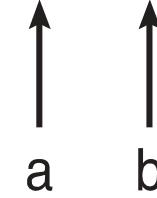
4 7 0 7 1 0 1 2 1 2 7 1 2



stack
before
a



stack
after
b



In-Class Work 6

Consider the following page reference string:

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6

How many page faults would occur for the following replacement algorithms, assuming **three frames**?

Remember that all frames are initially empty, so your first unique pages will cost one fault each.

- LRU replacement
- FIFO replacement
- Optimal replacement

Virtual Memory (4)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems

11/11/2019



In-Class Work 6

Consider the following page reference string:

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3,
6

How many page faults would occur for the
following replacement algorithms, assuming
three frames?

Remember that all frames are initially empty, so
your first unique pages will cost one fault each.

- LRU replacement
- FIFO replacement
- Optimal replacement

LRU Approximation Algorithms

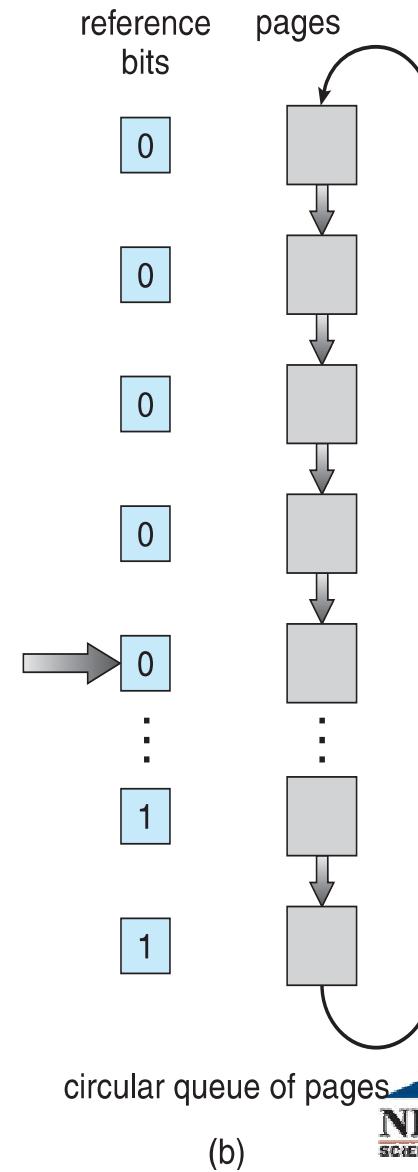
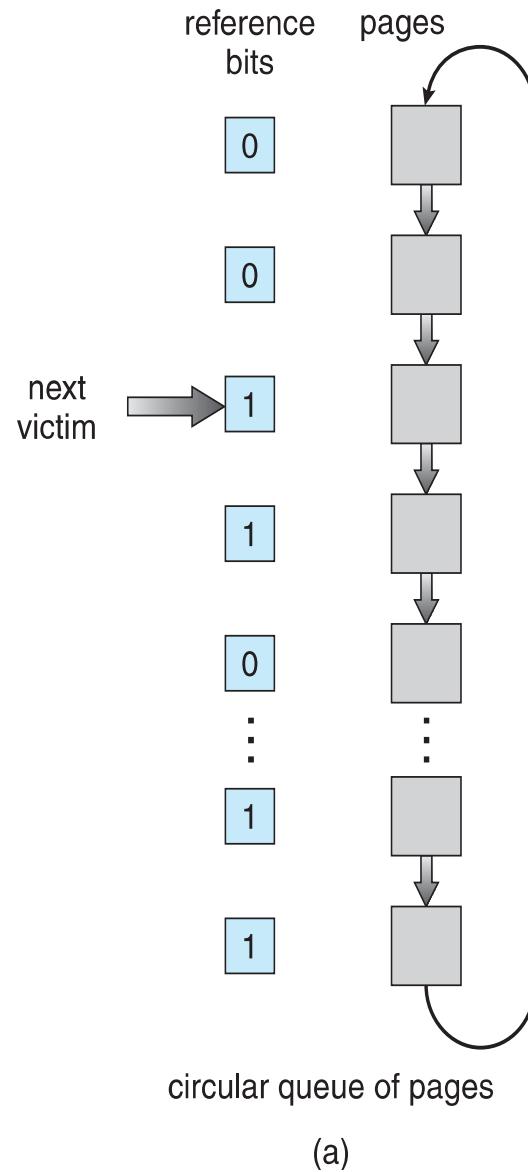
- LRU needs special hardware and still slow
- **Reference bit**
 - With each page associate a bit, initially = 0
 - When page is referenced bit set to 1
 - Replace any with reference bit = 0 (if one exists)
 - We do not know the order, however
 - Basis for many LRU approximation algorithms

LRU Approximation Algorithms

□ Second-chance algorithm

- Generally FIFO, plus hardware-provided reference bit
- 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



Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available)
- Take ordered pair (reference, modify)
 - (0, 0) neither recently used nor modified – best page to replace
 - (0, 1) not recently used but modified – not quite as good, must write out before replacement
 - (1, 0) recently used but clean – probably will be used again soon
 - (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement
- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
 - Might need to search circular queue several times



Frame Allocation

- ❑ Each process needs minimum number of frames
 - ❑ E.g. IBM 370 – 6 pages to handle MVC instruction
- ❑ Maximum that a process can get of course is total frames in the system
- ❑ Two major allocation schemes
 - ❑ Fixed allocation
 - ❑ Priority allocation
- ❑ Many variations

Fixed Allocation

- Equal allocation
 - E.g. if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
- Proportional allocation – allocate according to the size of process
 - Dynamic as degree of multiprogramming, process sizes change

s_i = size of process p_i

$$S = \sum s_i$$

m = total number of frames

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



Priority Allocation

- ❑ 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

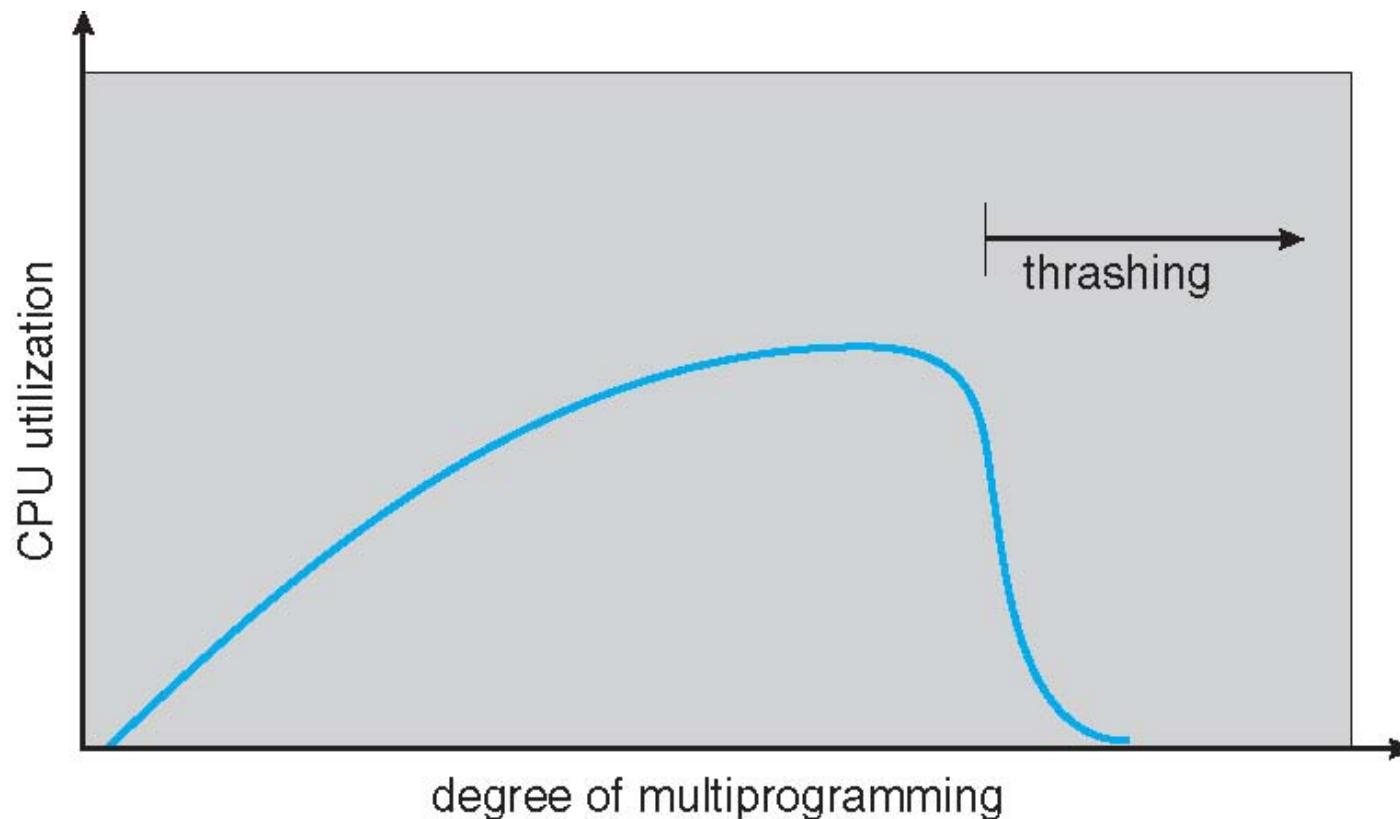
Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
 - A process cannot control its own page-fault rate
 - 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

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 (Cont.)



Demand Paging and Thrashing

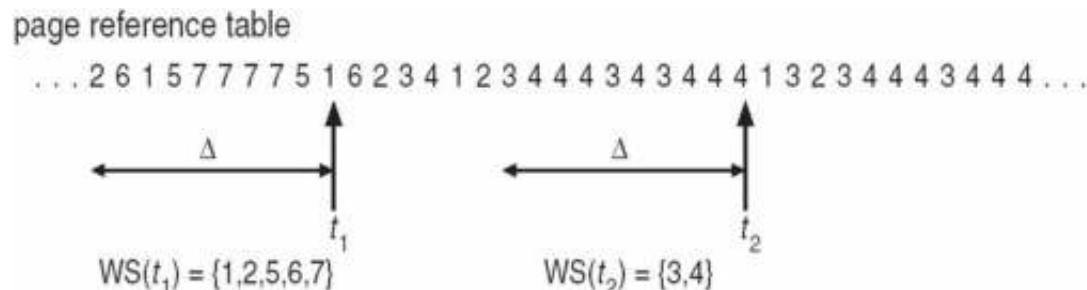
- Why does demand paging work?

Locality model

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

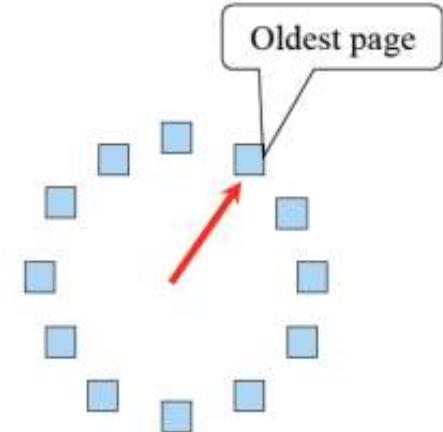
Working Set

- Main idea
 - Define a working set as the set of pages in the most recent K page references to approximate the program's locality
 - Keep the working set in memory will reduce page faults significantly
- Approximate working set
 - The set of pages of a process used in the last T seconds



WSClock

- Follow the clock hand
- If the reference bit is 1
 - Set reference bit to 0
 - Set the current time for the page
 - Advance the clock hand
- If the reference bit is 0, check “time of last use”
 - If the page has been used within δ , go to the next
 - If the page has not been used within δ and modify bit is 1
 - Schedule the page for page out and go to the next
 - If the page has not been used within δ and modify bit is 0
 - Replace this page



"WSCLOCK—a simple and effective algorithm for virtual memory management" by Richard W. Carr and John L. Hennessy, SOSP'81.

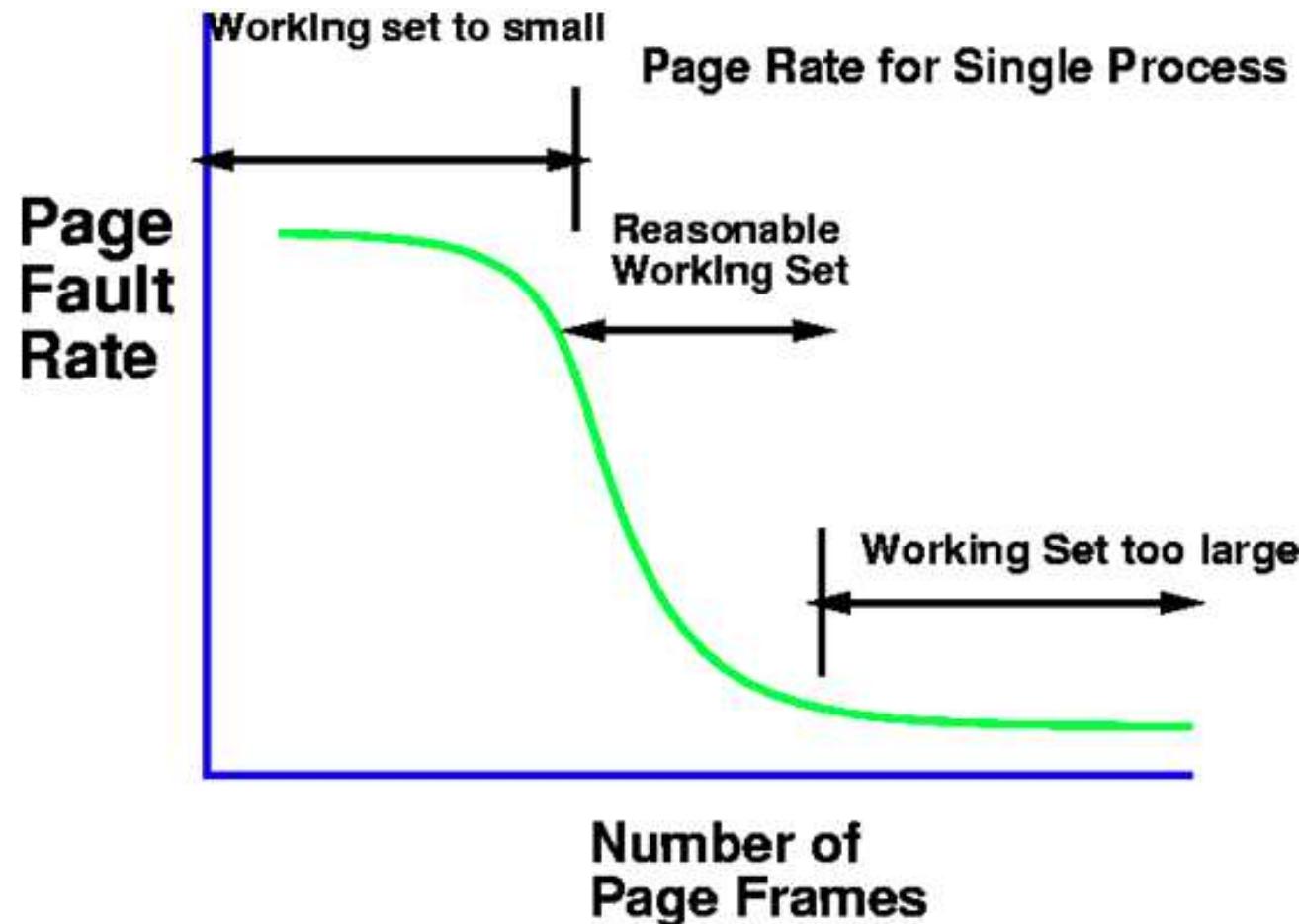
Virtual Memory (5)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems

11/13/2019

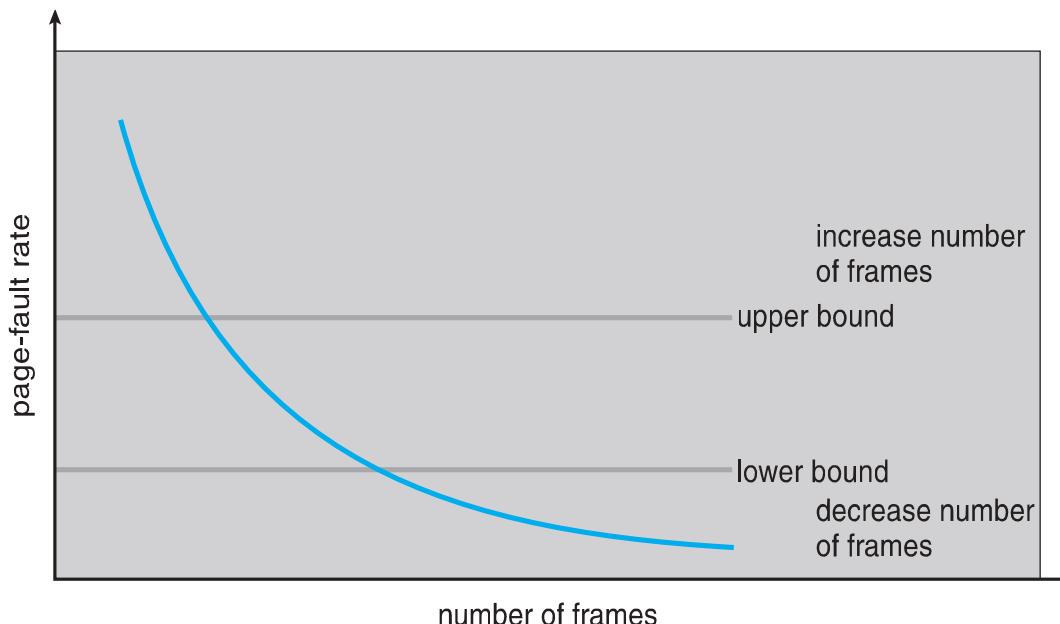


Working Set Size vs. Page Fault Rate



Page-Fault Frequency

- More direct approach than working-set
- Establish “acceptable” **page-fault frequency (PFF)** rate and use local replacement policy
 - If actual rate too low, process loses frame
 - If actual rate too high, process gains frame



Example: Windows

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page
- Processes are assigned **working set minimum** and **working set maximum**
 - Working set minimum is the minimum number of pages the process is guaranteed to have in memory
 - A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory
 - Working set trimming removes pages from processes that have pages in excess of their working set minimum



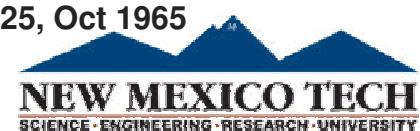
Allocating Kernel Memory

- ❑ 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 Allocation (1)

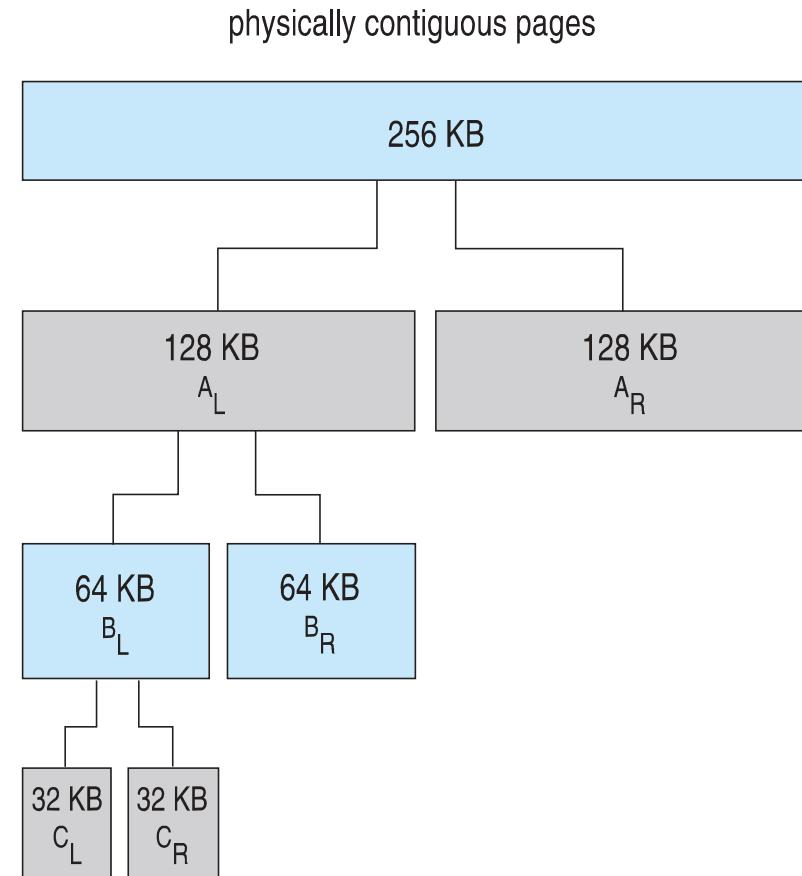
- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
 - Satisfies requests in units sized as power of 2
 - Request rounded up to next highest power of 2
 - When smaller allocation needed than is available, current chunk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

Kenneth C. Knowlton. A Fast storage allocator. *Communications of the ACM* 8(10):623-625, Oct 1965



Buddy Allocation (2)

- For example, assume 256KB chunk available, kernel requests 21KB
- Advantage – quickly **coalesce** unused chunks into larger chunk
- Disadvantage - fragmentation



In-class Work 7

- Consider a system with 1MB of available memory and requests for 42KB, 396KB, 10KB, and 28KB. Show the amount of memory allocated for each request and the state of memory after each request.
- How much internal fragmentation exists in this scenario?
- How much external fragmentation exists in this scenario?

Answer

Original: 1MB

42KB -> 64KB: 64KB(x) 64KB 128KB 256KB 512KB

396KB->512KB: 64KB(x) 64KB 128KB 256KB 512KB(x)

10KB->16KB: 64KB(x) 16KB(x) 16KB 32KB 128KB 256KB 512KB(x)

28KB->32KB: 64KB(x) 16KB(x) 16KB 32KB(x) 128KB 256KB 512KB(x)

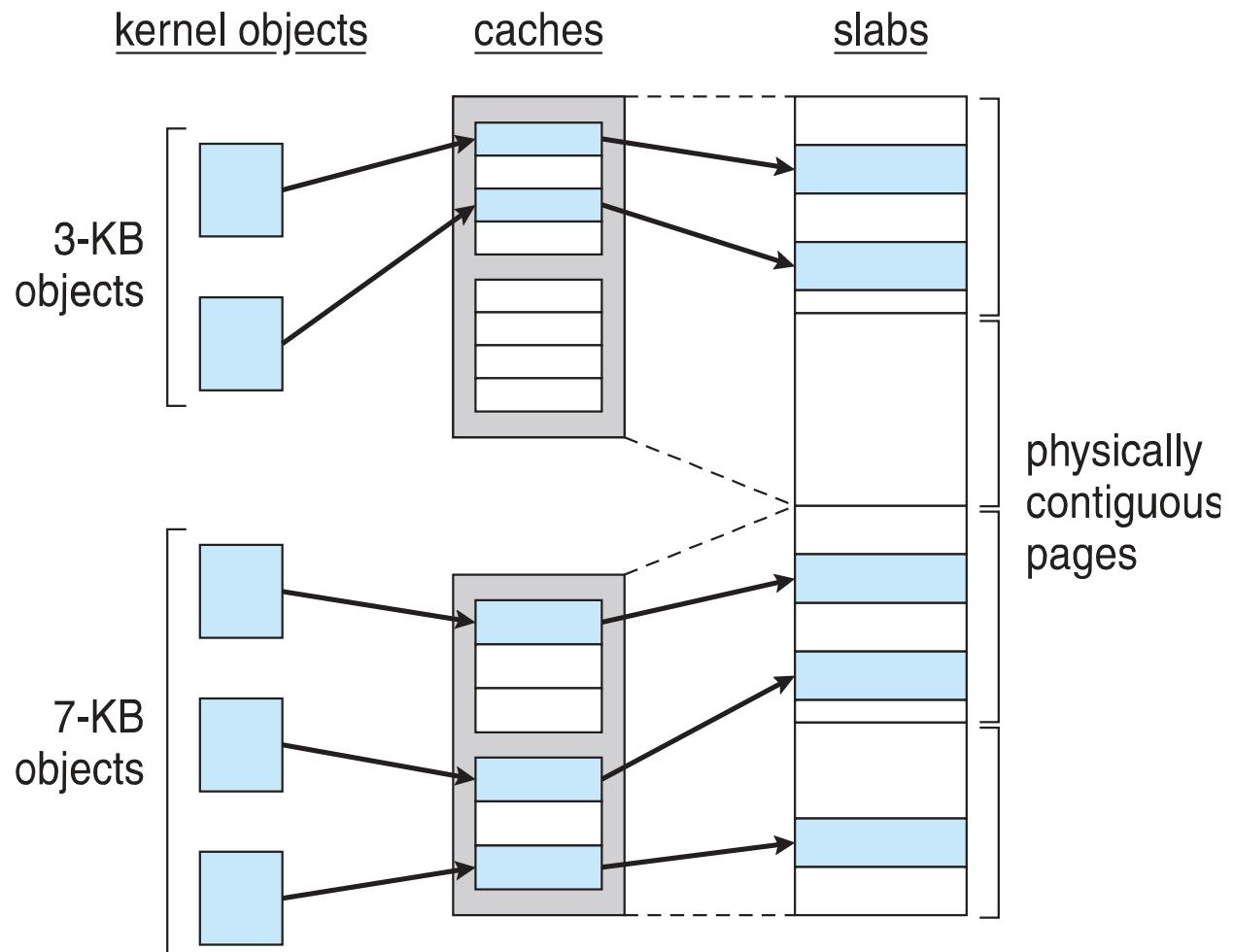
Internal fragmentation: $(64-42)+(512-396)+(16-10)+(32-28) = 148\text{KB}$

External fragmentation: $16+128+256 = 400\text{KB}$

Slab Allocation (1)

- Motivation
 - Frequent (de)allocation of certain kernel objects
 - **e.g. file struct and inode**
 - Other allocators: overly general; assume variable size
- **Slab: cache of slots**
 - slot size = object size
 - free memory management = bitmap
 - allocate: set bit and return slot
 - Free: clear bit
- Used in FreeBSD and linux, implemented on top of buddy page allocator, for objects smaller than a page

Slab Allocation (2)



File Systems (1)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems

11/15/2019



File Systems

- Most visible aspect of an OS
- Implement an abstraction (**files**) for secondary storage
 - A file is a named collection of related information that is recorded on secondary storage.
 - Data can NOT be written to secondary storage unless they are within a file.
- Organize files logically (**directories**)
- Permit sharing of data between processes, users, and machines
- Protect data from unwanted access (**security**)

File Structure

- A file has a certain defined **structure** which depends on its types:
 - A text file is a sequence of characters organized into lines.
 - A source file is a sequence of subroutines and function.
 - An object file is a sequence of bytes organized into blocks understandable by the system's linker.
 - An executable file is a series of code sections that the loader can bring into memory and execute.

File Attributes

- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk

File Operations

- **Create**
- **Write** – at **write pointer** location
- **Read** – at **read pointer** location
- **Reposition within file - seek**
- **Delete**
- **Truncate**
- **Open(F_i)** – search the directory structure on disk for entry F_i , and move the content of entry to memory
- **Close (F_i)** – move the content of entry F_i in memory to directory structure on disk

Access Methods

Sequential access

- Read all bytes/records from the beginning
- Cannot jump around
 - May rewind or back up, however
- Convenient when medium was magnetic tape
- Often useful when whole file is needed

Random access

- Bytes (or records) read in any order
- Essential for database systems
- Read can be ...
 - Move file marker (seek), then read or ...
 - Read and then move file marker

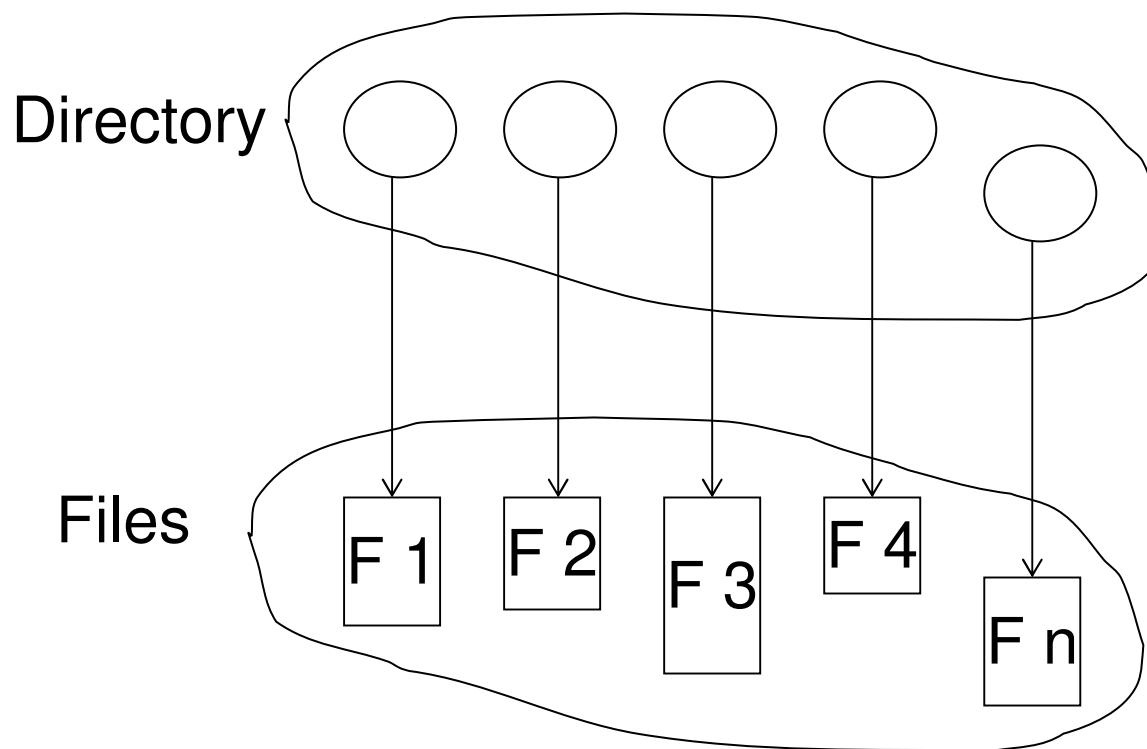


Directories

- ❑ Naming is nice, but limited
- ❑ Humans like to group things together for convenience
- ❑ File systems allow this to be done with *directories*

Directory Structure

A collection of nodes containing information about all files



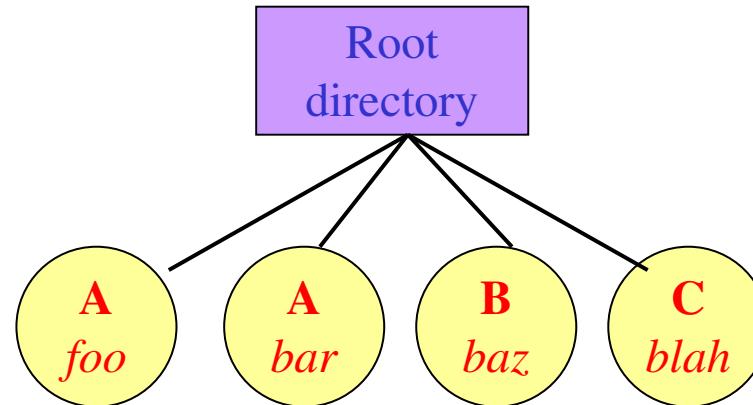
Both the directory structure and the files reside on disk

Directory Organization

The directory is organized logically to obtain

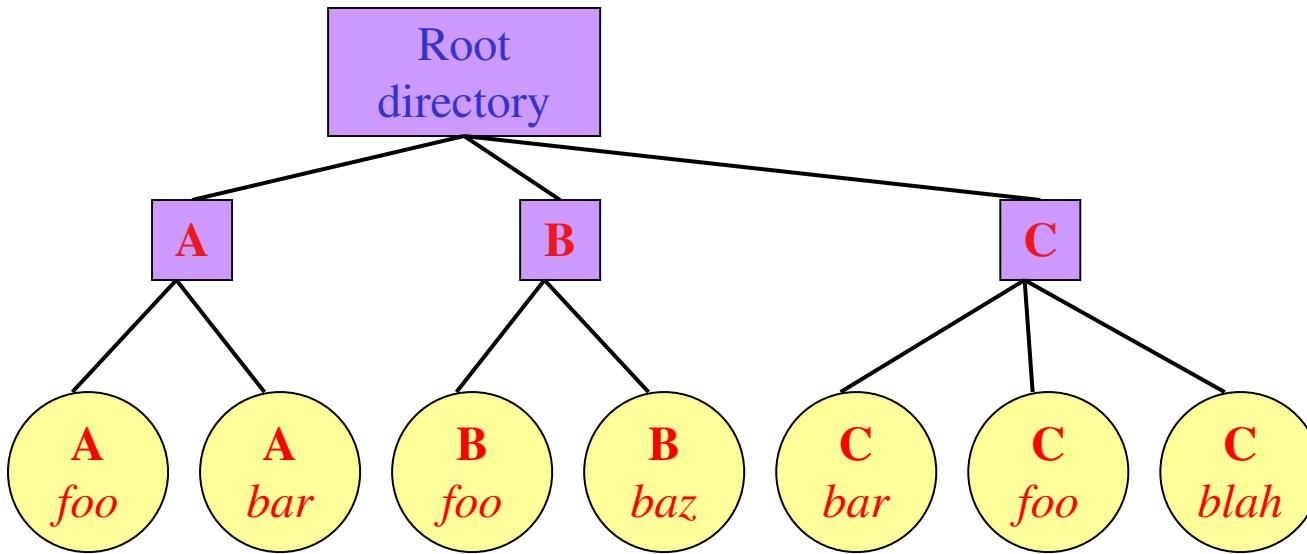
- ❑ **Efficiency** – locating a file quickly
- ❑ **Naming** – convenient to users
 - ❑ Two users can have same name for different files
 - ❑ The same file can have several different names
- ❑ **Grouping** – logical grouping of files by properties, (e.g., all Java programs, all games, ...)

Single-Level Directory



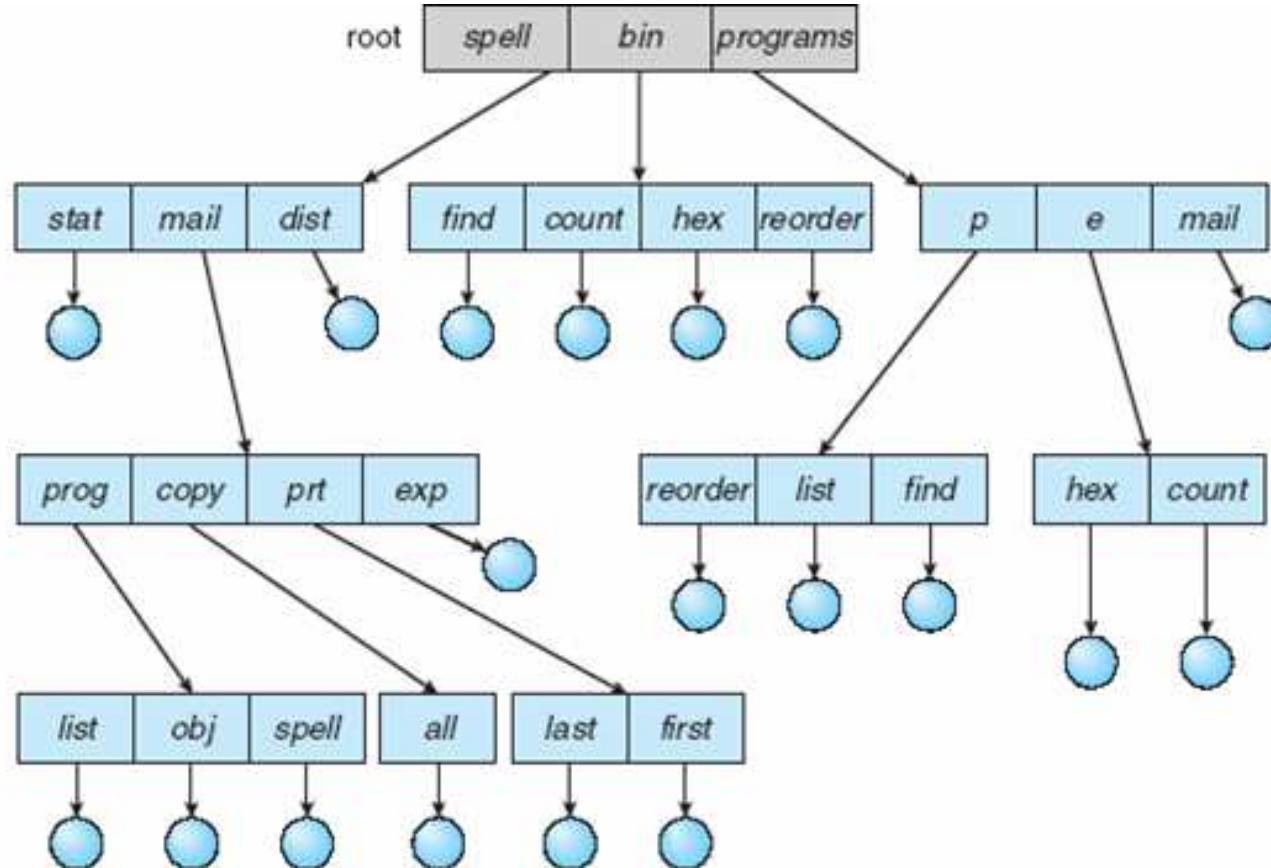
- ❑ One directory in the file system
- ❑ Example directory
 - ❑ Contains 4 files (*foo*, *bar*, *baz*, *blah*)
 - ❑ owned by 3 different people: A, B, and C (owners shown in red)
- ❑ Problem: what if user B wants to create a file called *foo*?
- ❑ Naming problem, grouping problem

Two-Level Directory



- Solves naming problem:** each user has her own directory
- Multiple users can use the same file name
- By default, users access files in their own directories
- Extension: allow users to access files in others' directories
- No grouping capability**

Tree-Structured Directories



Tree-Structured Directories

- **Absolute** or **relative** path name
- Creating a new file is done in current directory
- Delete a file

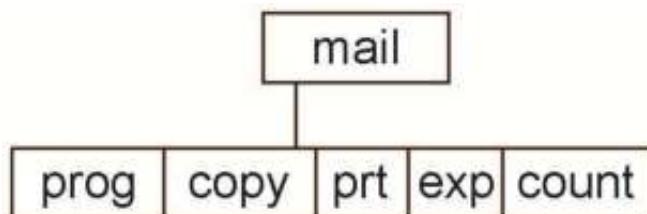
rm <file-name>

- Creating a new subdirectory is done in current directory

mkdir <dir-name>

- Example: if in current directory **/mail**

mkdir count



Deleting “mail” ⇒ deleting the entire subtree rooted by “mail”

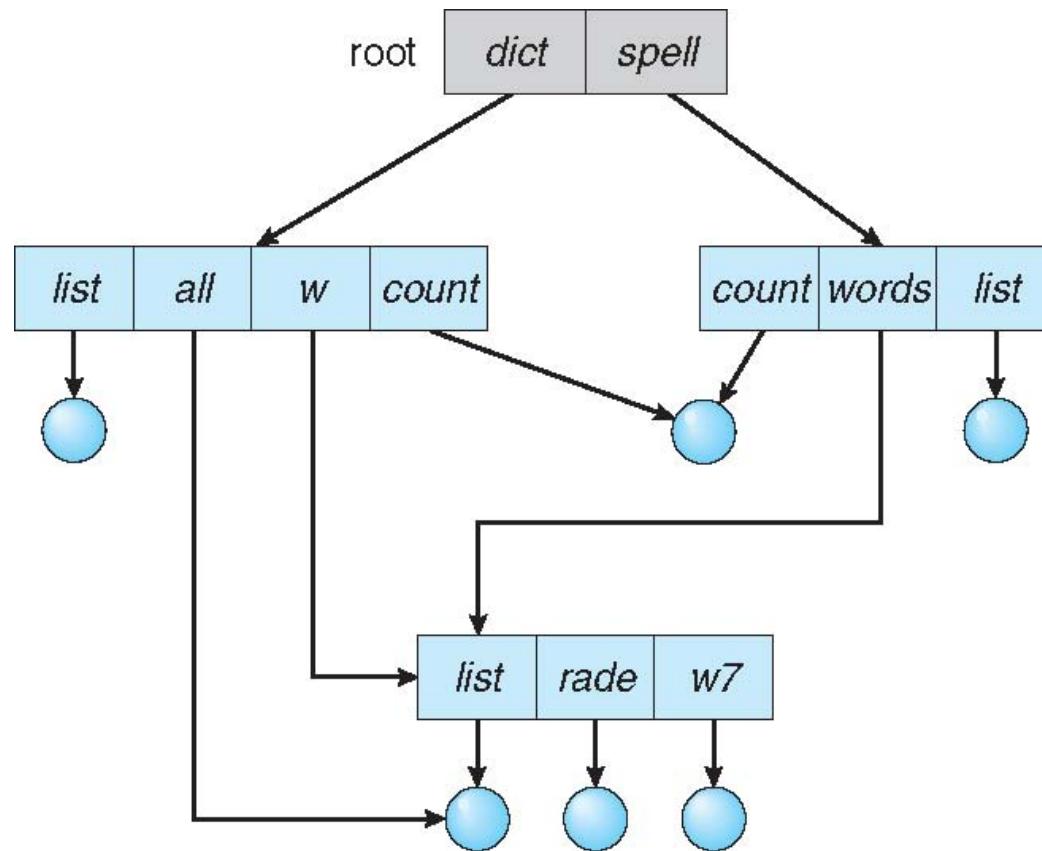
Tree-Structured Directories

- ❑ Efficient searching
- ❑ Grouping Capability
- ❑ Current directory (working directory)



Acyclic-Graph Directories

Have shared subdirectories and files



Acyclic-Graph Directories

- Two different names (aliasing)
- New directory entry type
 - **Link** – another name (pointer) to an existing file
 - **Resolve the link** – follow pointer to locate the file
- If ***dict*** deletes ***count*** ⇒ dangling pointer
 - How to solve this problem?

Solutions to Dangling Pointer Problem

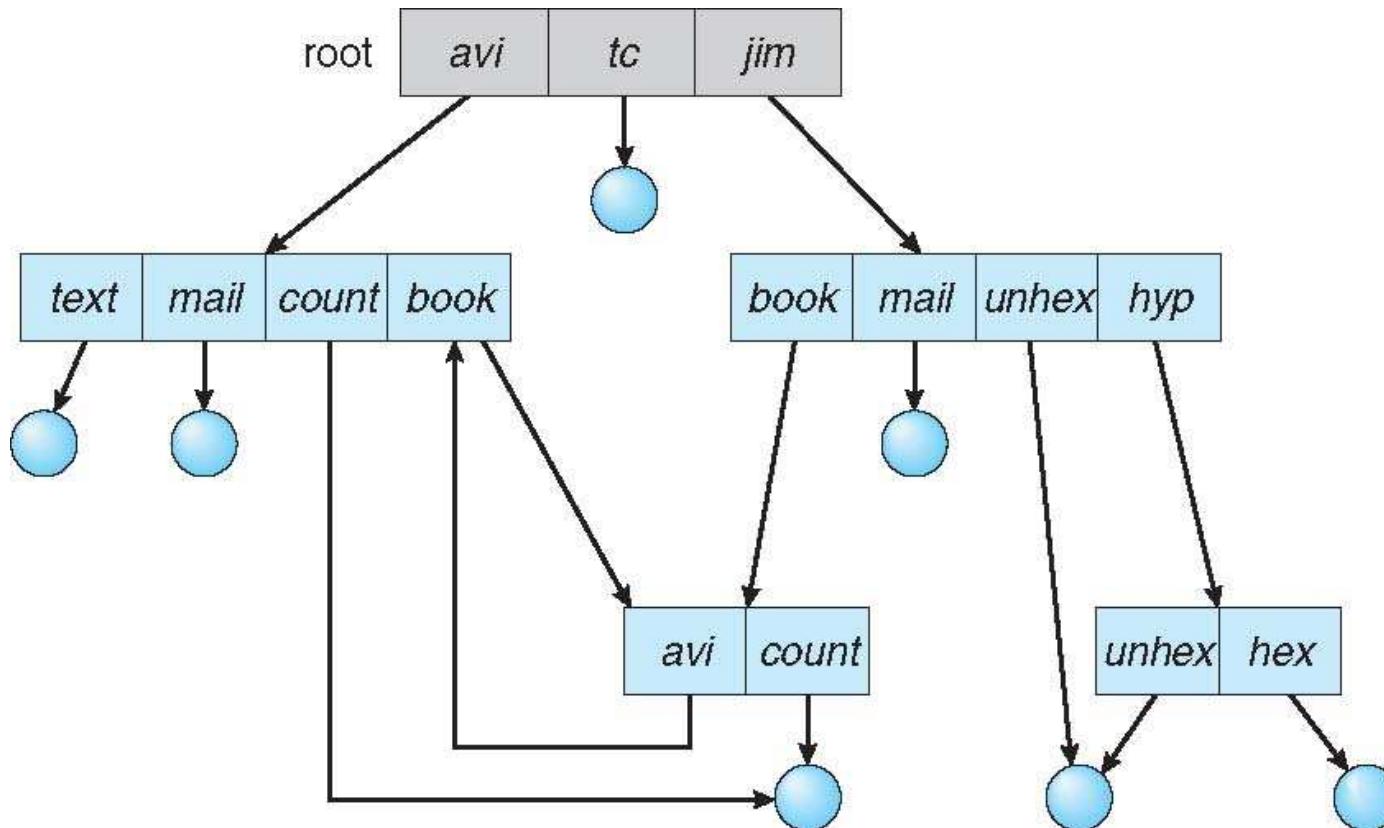
- Backpointers, so we can delete all pointers
Variable size records a problem
- Entry-hold-count solution

File Systems (2)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems
11/18/2019



General Graph Directory



Cycles?

General Graph Directory

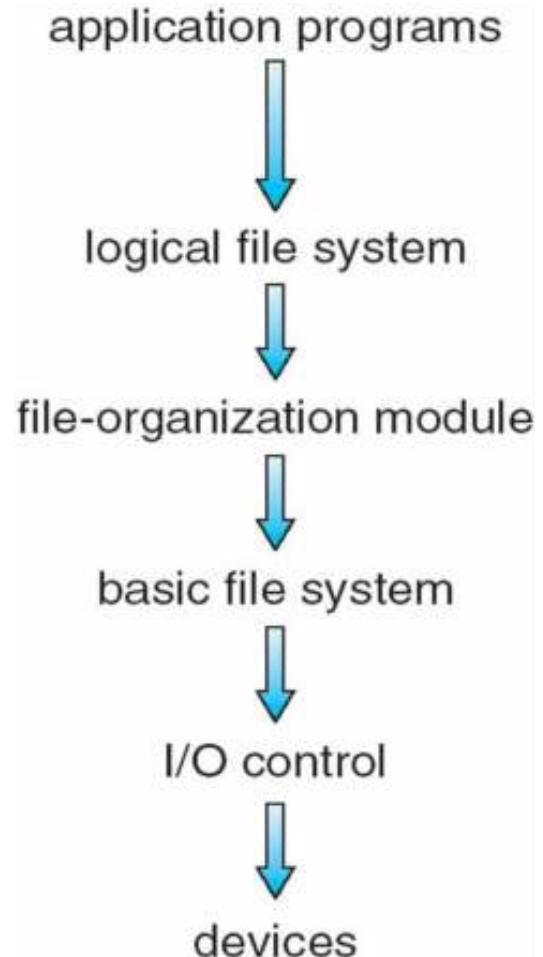
□ How do we guarantee no cycles?

- Allow only links to file not subdirectories
- Every time a new link is added use a cycle detection algorithm to determine whether it is OK

File-System Structure

- **File system** resides on secondary storage (disks)
 - Provided user interface to storage, mapping logical to physical
 - Provides efficient and convenient access to disk by allowing data to be stored, located retrieved easily
- Disk provides in-place rewrite and random access
 - I/O transfers performed in **blocks** of **sectors** (32B – 4,096B, usually 512B)
- **File control block** – storage structure consisting of information about a file
- **Device driver** controls the physical device
- File system organized into layers

Layered File System



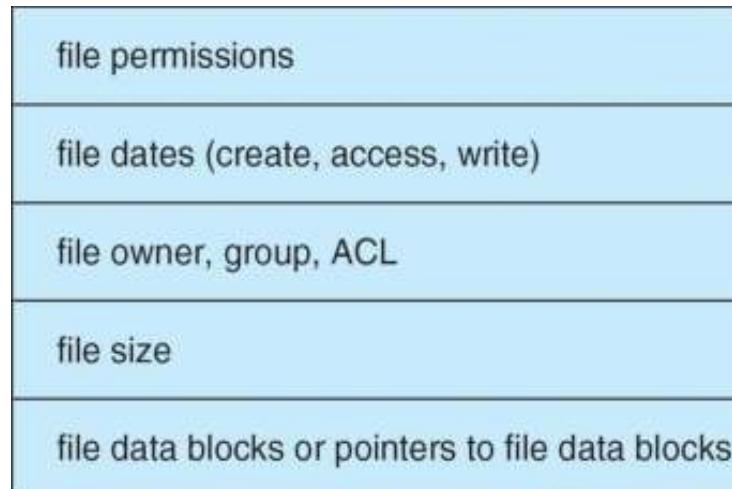
File-System Implementation

(1)

- We have system calls at the API level, but how do we implement their functions?
 - On-disk and in-memory structures
- Boot control block contains info needed by system to boot OS from that volume
 - Needed if volume contains OS, usually first block of volume
- Volume control block (UFS: superblock, NTFS: master file table) contains volume details
 - Total # of blocks, # of free blocks, block size, free block pointers or array
- Directory structure organizes the files
 - UFS: Names and inode numbers, NTFS: master file table

File-System Implementation (2)

- Per-file **File Control Block (FCB)** contains many details about the file
 - UFS: inode number, permissions, size, dates
 - NTFS: stores into in master file table using relational DB structures



In-Memory File System Structures

- An in-memory **mount table** contains information about each mounted volume.
- An in-memory directory-structure cache holds the directory information of recently accessed directories. (For directories at which volumes are mounted, it can contain a pointer to the volume table.)
- The **system-wide open-file table** contains a copy of the FCB of each open file, as well as other information.
- The **per-process open-file table** contains a pointer to the appropriate entry in the system-wide open-file table, as well as other information.
- Buffers hold file-system blocks when they are being read from disk or written to disk.

Directory Implementation

- **Linear list** of file names with pointer to the data blocks
 - Simple to program
 - Time-consuming to execute
 - Linear search time
 - Could keep ordered alphabetically via linked list or use B+ tree
- **Hash Table** – linear list with hash data structure
 - Decreases directory search time
 - **Collisions** – situations where two file names hash to the same location
 - Only good if the number of entries is fixed, or use chained-overflow method

Allocation Methods

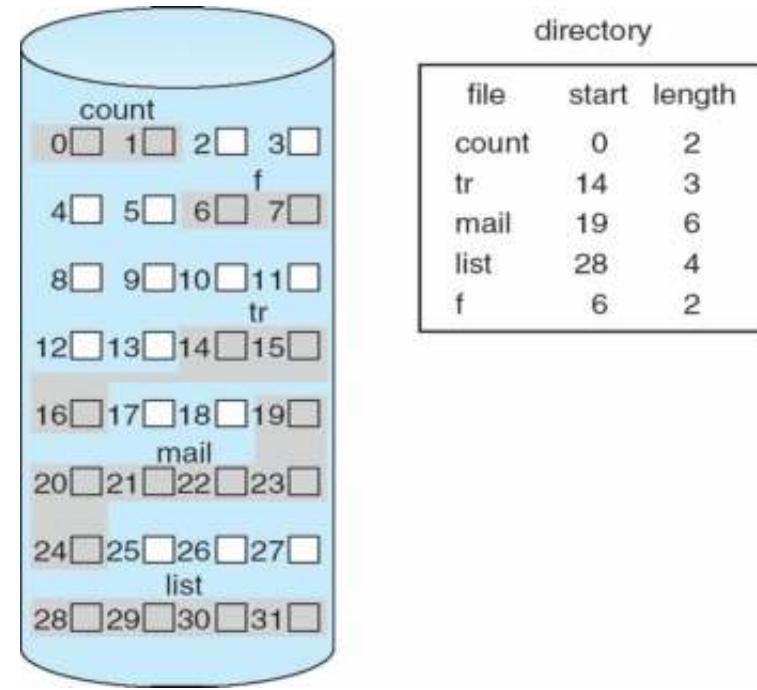
- An allocation method refers to how disk blocks are allocated for files:
- Three methods
 - Contiguous allocation
 - Linked allocation
 - Indexed allocation

Contiguous Allocation

- Each file occupies set of contiguous blocks
- Best performance in most cases
- Pros:
 - Simple – only starting location (block #) and length (number of blocks) are required
 - Direct access

Contiguous Allocation

- ❑ Mapping from logical address to physical address



Block to be accessed = Q + starting address
Displacement into block = R

Cons

- ❑ Waste of space
- ❑ Difficult to support dynamic file sizes (files cannot grow)

File Systems (3)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems
11/20/2019

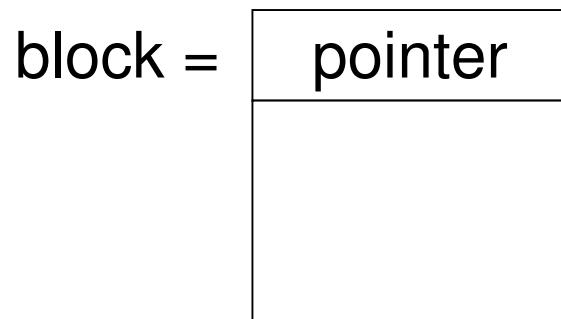


Extent-Based Systems

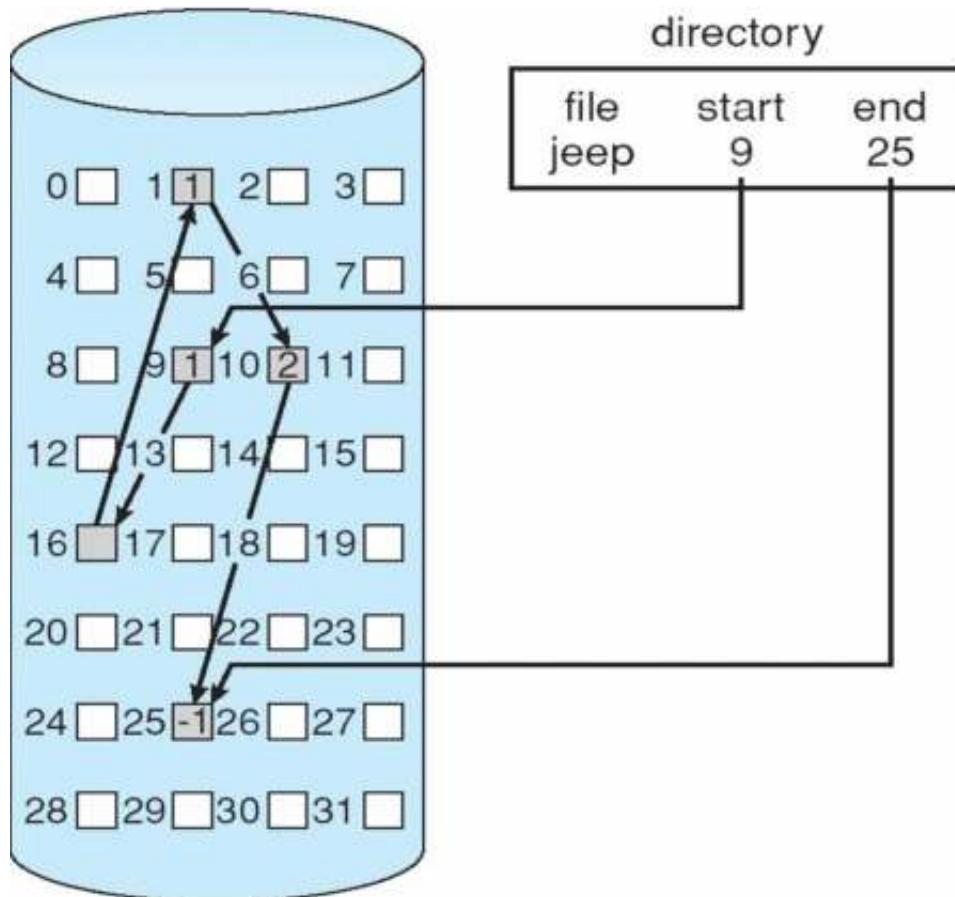
- Many newer file systems (i.e., Veritas File System) use a modified contiguous allocation scheme
- Extent-based file systems allocate disk blocks in extents
- An **extent** is a contiguous area of disks (a range of blocks)
 - Extents are allocated for file allocation
 - A file consists of one or more extents

Linked Allocation

- Each file is a linked list of disk blocks
 - Blocks may be scattered anywhere on the disk
- Each block contains pointer to next block



Linked Allocation



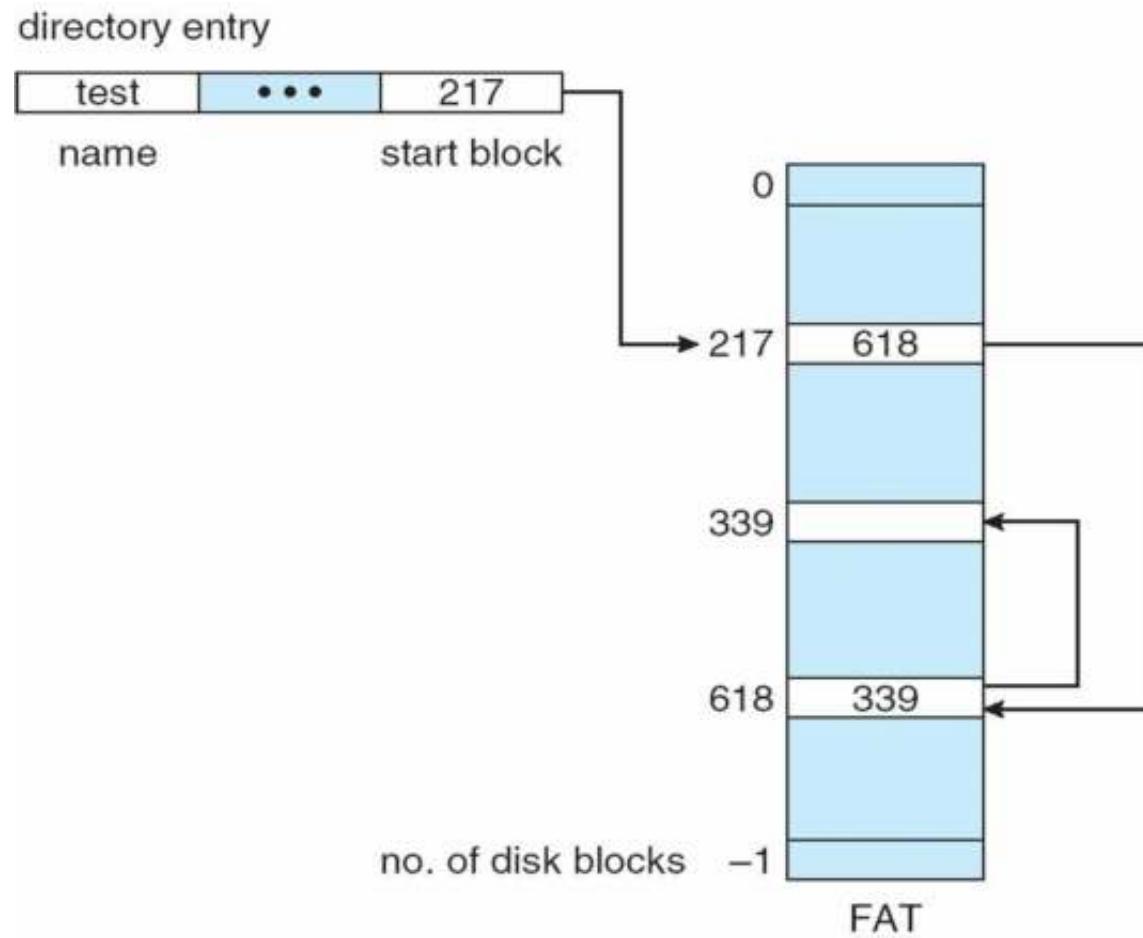
Pros

- Simple – need only starting address
- No external fragmentation

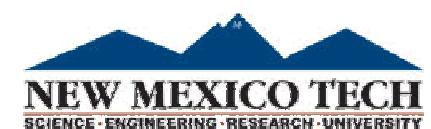
Cons

- No random access
- Seek can be slow
- Space required for pointer
- Reliability

File-Allocation Table (FAT)

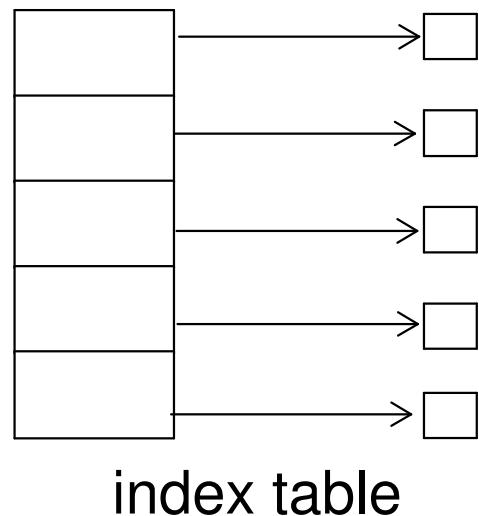


MS-DOS

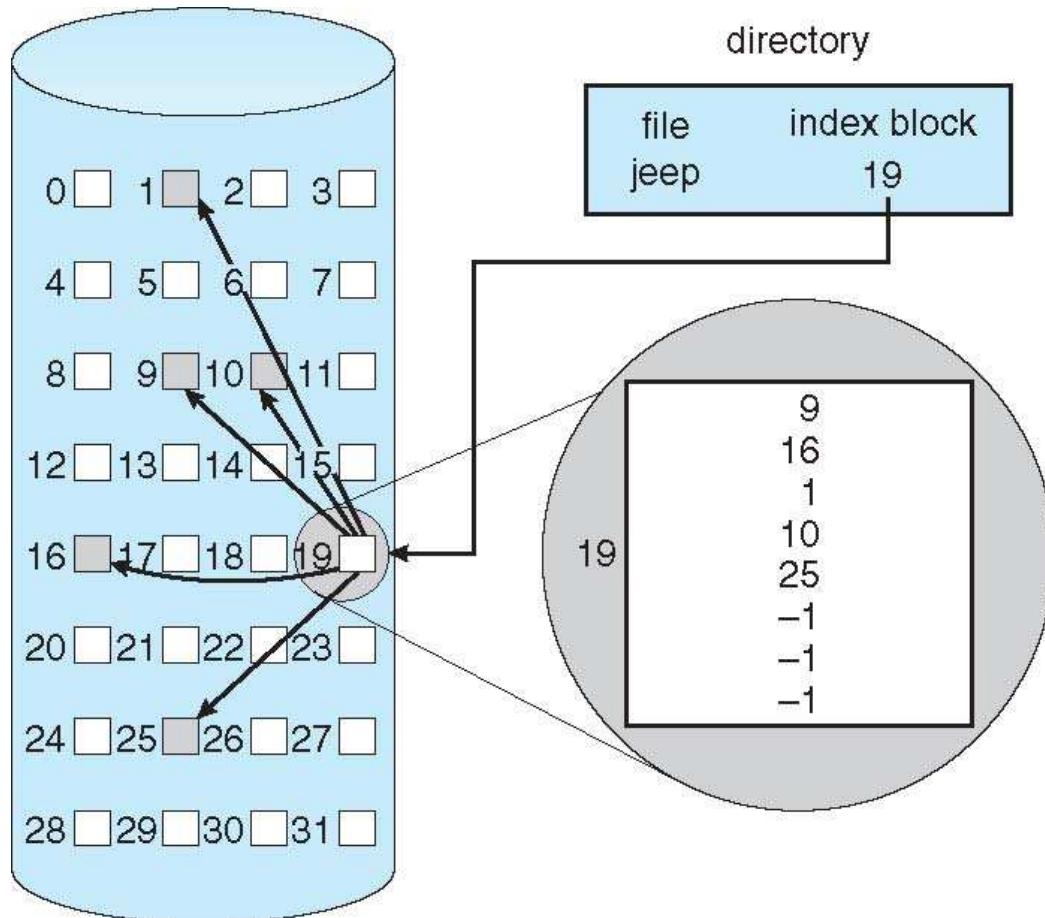


Indexed Allocation

- Each file has its own **index block**(s) of pointers to its data blocks
- Logical view



Example of Indexed Allocation



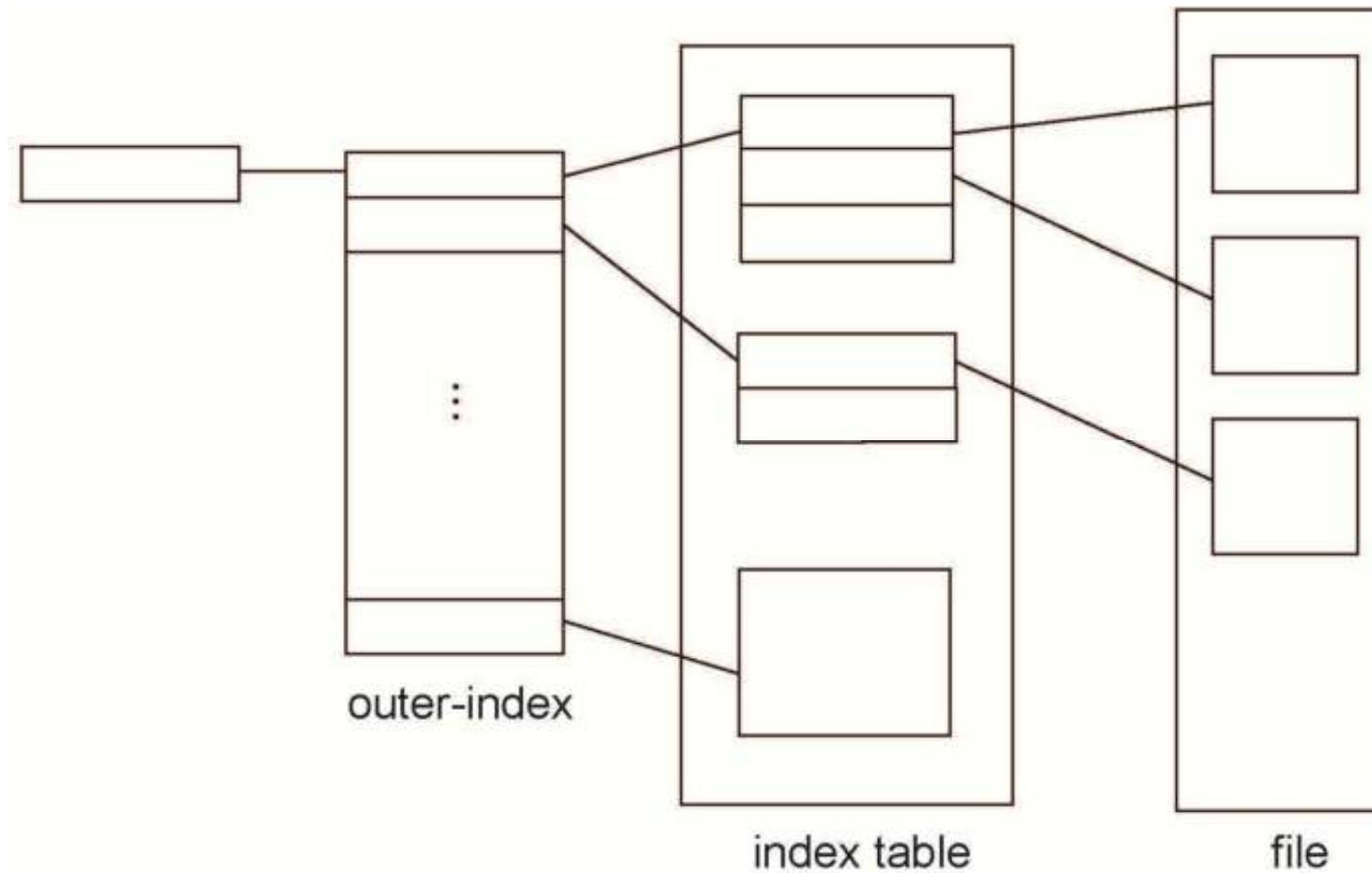
Pros

- Simplify seeks
- Random access
- Dynamic access without external fragmentation
- May link several index blocks together (for large files)

Cons

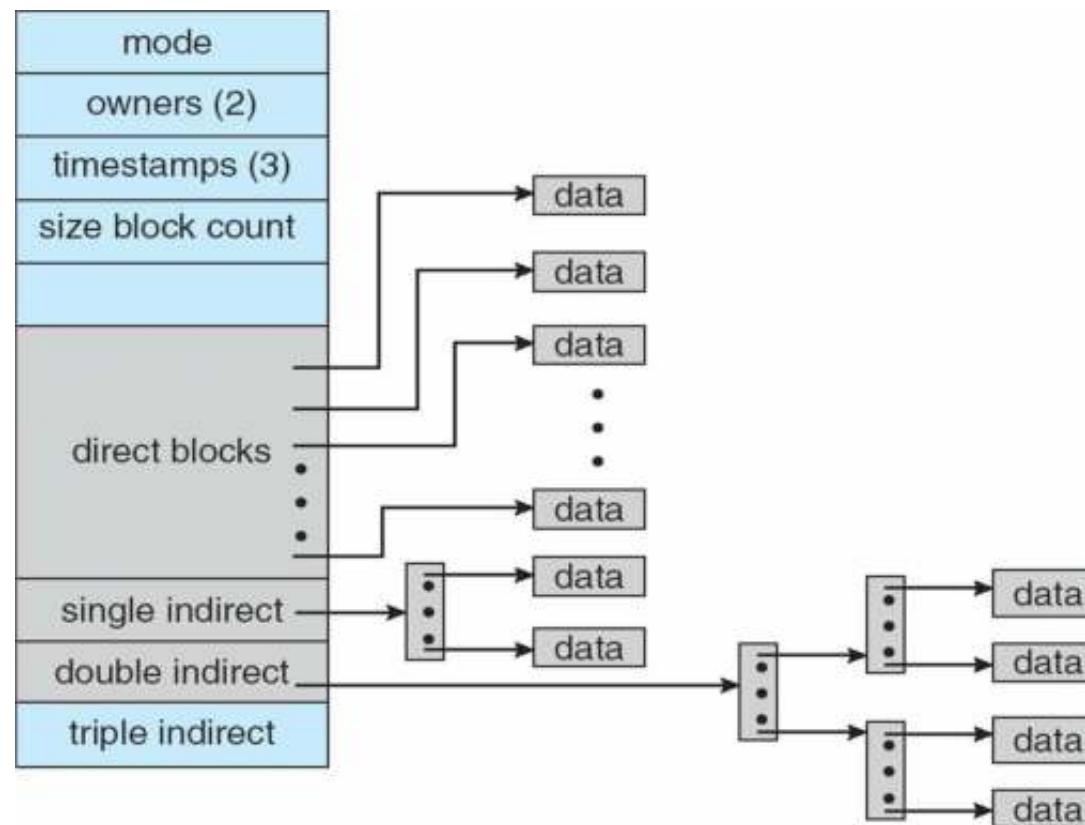
- Overhead of index block

Indexed Allocation – Mapping



Combined Scheme: UNIX UFS

- ❑ 4K bytes per block, 32-bit addresses



Question

- Consider a file system that uses inodes to represent files. Disk blocks are 8 KB in size, and a pointer to a disk block requires 4 bytes. This file system has 12 direct disk blocks, as well as single, double, and triple indirect disk blocks. What is the maximum size of a file that can be stored in this file system?

Answer

12 direct disk blocks *8KB = 96KB

1 single disk block = $2048 * 8KB = 16384KB$

1 double disk block = $2048^2 * 8 KB = 33554432 KB$

1 triple disk block = $2048^3 * 8 KB = 68719476736 KB$

Total = 64.03 TB

In-Class Work 8

Consider a UNIX file system with 10 direct pointers, 1 indirect pointer, 1 double-indirect pointer, and 1 triple-indirect pointer in the i-node. Assume that disk blocks are 4K bytes and that each pointer to a disk block requires 4 bytes.

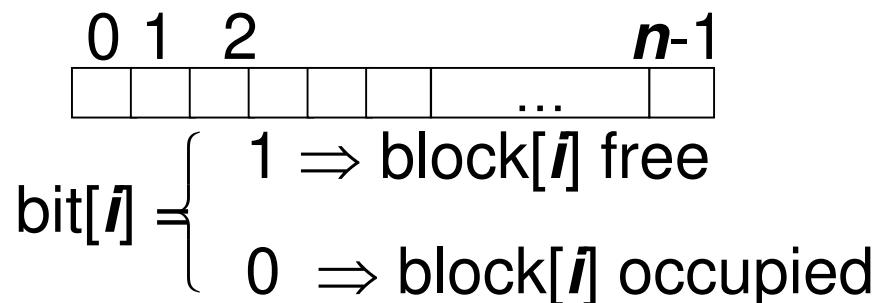
- (1) What is the largest possible file that can be supported with this design? (show your work as expression, no need to calculate the numeric result)
- (2) Assume that the operating system has already read your file into the main memory. How many disk reads are required to read the data block 800 into memory? Explain your answer.

Answer

- (1) $10 * 4\text{KB} + 1024 * 4\text{KB} + 1024 * 1024 * 4\text{KB} + 1024 * 1024 * 1024 * 4\text{KB}$
- (2) Data block number 800 falls in the set of blocks accessible from the single indirect block (block number 10-1033). One disk read is required to read the indirect block, one disk read is required to read the actual data, for a total of two disk reads.

Free-Space Management (1)

- File system maintains **free-space list** to track available blocks/clusters
 - (Using term “block” for simplicity)
- **Bit vector** or **bit map** (n blocks)



Block number calculation (first free block)

$$\begin{aligned} & (\text{number of bits per word}) * \\ & (\text{number of 0-value words}) + \\ & \text{offset of first 1 bit} \end{aligned}$$

CPUs have instructions to return offset within word of first “1” bit

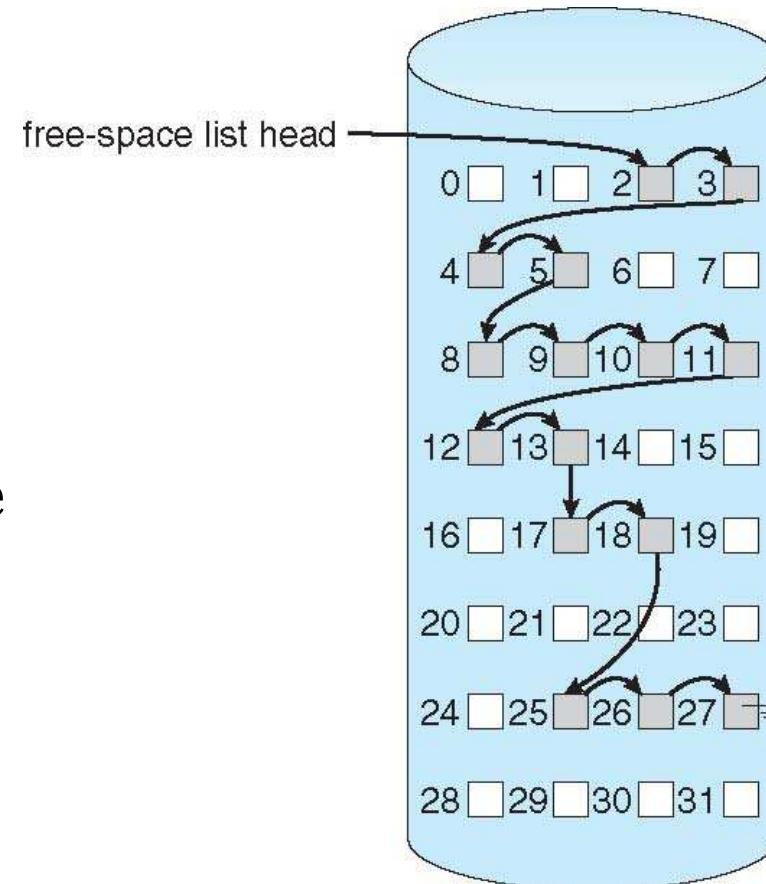
Free-Space Management (2)

- Bit map requires extra space
 - Example:
 - block size = 4KB = 2^{12} bytes
 - disk size = 2^{40} bytes (1 terabyte)
 - $n = 2^{40}/2^{12} = 2^{28}$ bits (or 256MB)
 - if clusters of 4 blocks -> 32MB of memory
- Easy to get contiguous files

Linked Free Space List on Disk

Linked list (free list)

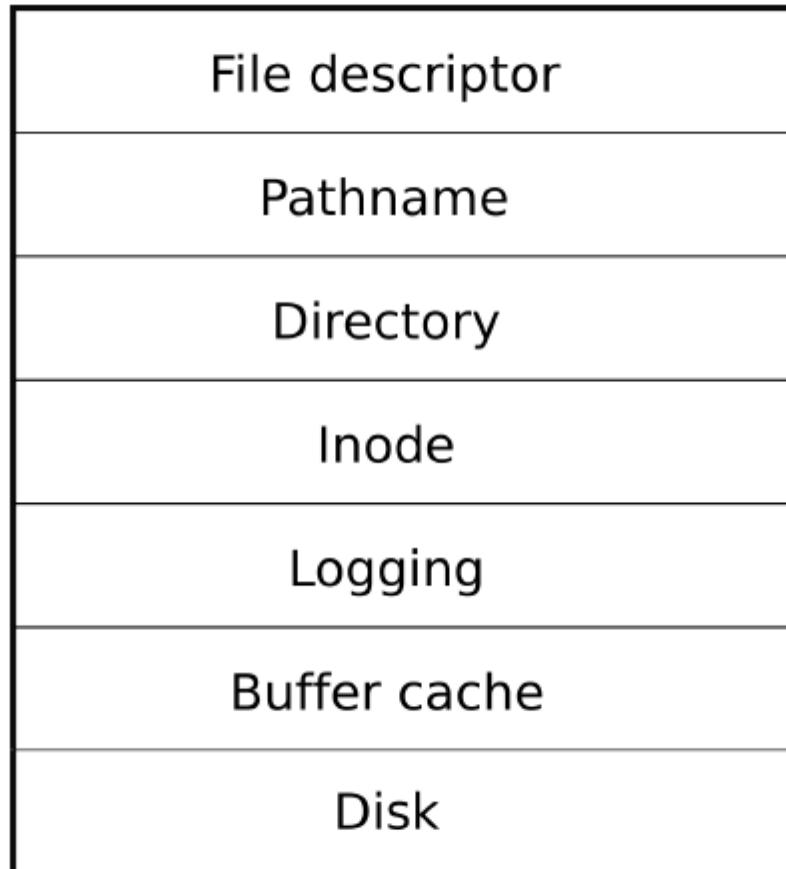
- ❑ Cannot get contiguous space easily
- ❑ No waste of space
- ❑ No need to traverse the entire list (if # free blocks recorded)



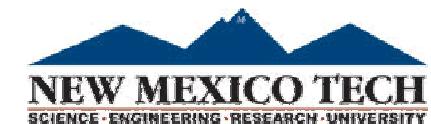
Free-Space Management (3)

- Grouping
 - Modify linked list to store address of next $n-1$ free blocks in first free block, plus a pointer to next block that contains free-block-pointers (like this one)
 - Find the addresses of a large number of free blocks quickly
- Counting
 - Because space is frequently contiguously used and freed, with contiguous-allocation algorithm, extents, or clustering
 - Keep address of first free block and count of following free contiguous blocks
 - Free space list then has entries containing addresses and counts
 - The entries can be stored in a balanced tree for efficient lookup, insertion, and deletion.

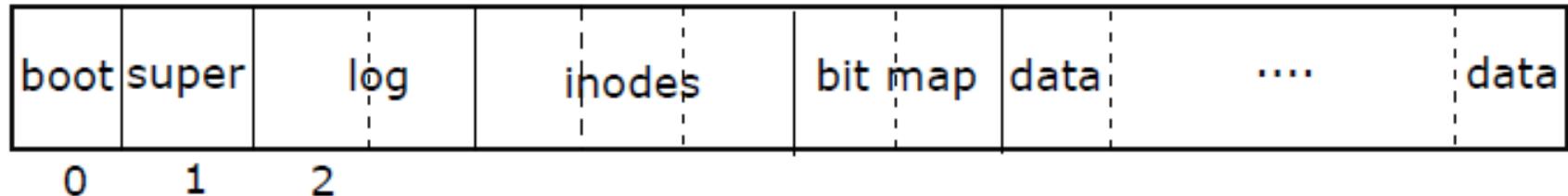
Layers of xv6 File System



Chapter 6 of xv6 book



xv6 File System Structure



fs.h, fs.c, mkfs.c

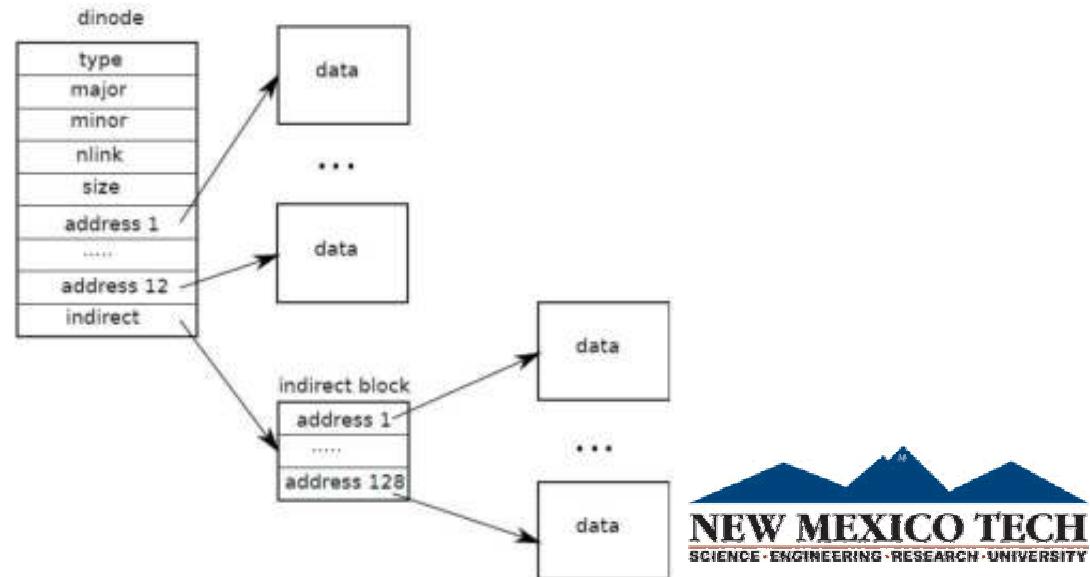
```
struct superblock {  
    uint size;          // Size of file system image (blocks)  
    uint nblocks;       // Number of data blocks  
    uint ninodes;       // Number of inodes.  
    uint nlog;          // Number of log blocks  
    uint logstart;      // Block number of first log block  
    uint inodestart;    // Block number of first inode block  
    uint bmapstart;     // Block number of first free map block  
};
```

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  
};
```

NDIRECT = 12

NINDIRECT = BSIZE/4 =
512/4 = 128



Storage Systems (1)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems
11/22/2019



Magnetic Disk

❑ Purpose

- ❑ Long-term, nonvolatile storage
- ❑ Large, inexpensive, slow level in the storage hierarchy

❑ Characteristics

- ❑ Seek Time
 - ❑ positional latency
 - ❑ rotational latency

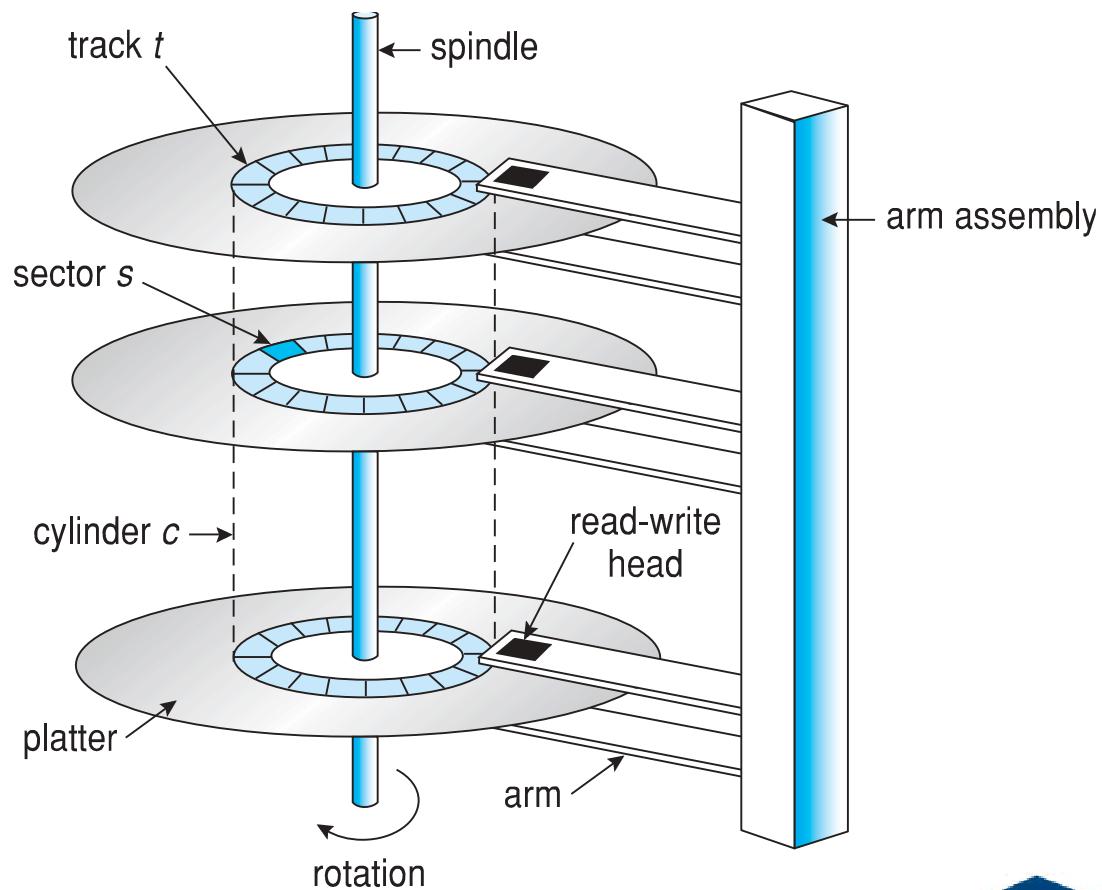
❑ Rotational rate

- ❑ 60 to 200 times per second

❑ Capacity

- ❑ Terabytes
- ❑ Quadruples every 3 years

Moving-head Disk Mechanism



Disk Structure

- Disk drives are addressed as large 1-dimensional arrays of *logical blocks*, where the logical block is the smallest unit of transfer.
- The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially.
 - Sector 0 is the first sector of the first track on the outermost cylinder.
 - Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

Disk Scheduling

- The operating system is responsible for using hardware efficiently – for the disk drives, this means having a fast access time and disk bandwidth
 - Minimize seek time
 - Seek time \approx seek distance
 - Disk **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer

Disk Scheduling (Cont.)

- Note that drive controllers have small buffers and can manage a queue of I/O requests (of varying “depth”)
- Several algorithms exist to schedule the servicing of disk I/O requests
- The analysis is true for one or many platters
- We illustrate scheduling algorithms with a request queue (0-199)

98, 183, 37, 122, 14, 124, 65, 67

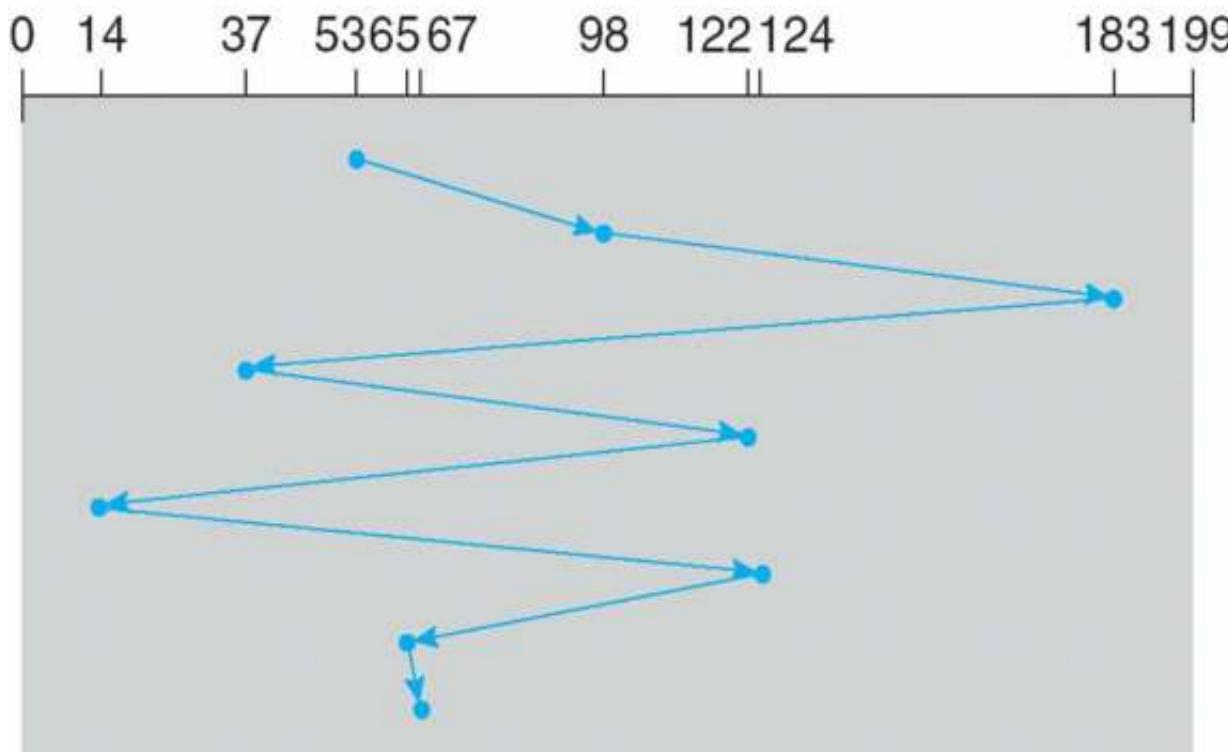
Head pointer 53

FCFS

Illustration shows total head movement of 640 cylinders

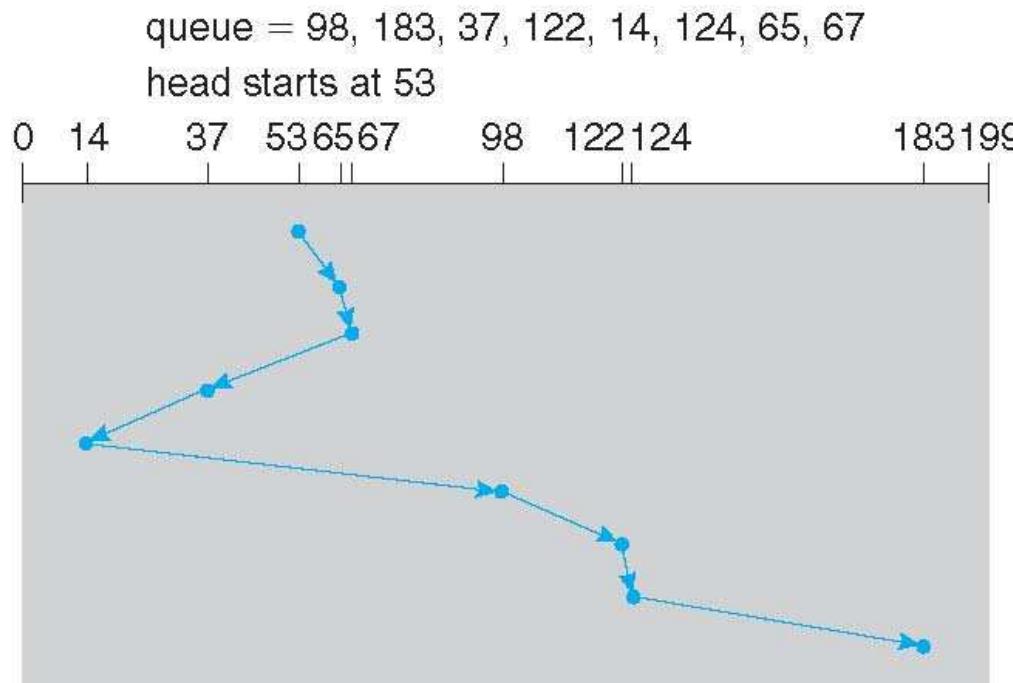
queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



SSTF

- Shortest Seek Time First selects the request with the minimum seek time from the current head position
- SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests
- Illustration shows total head movement of 236 cylinders



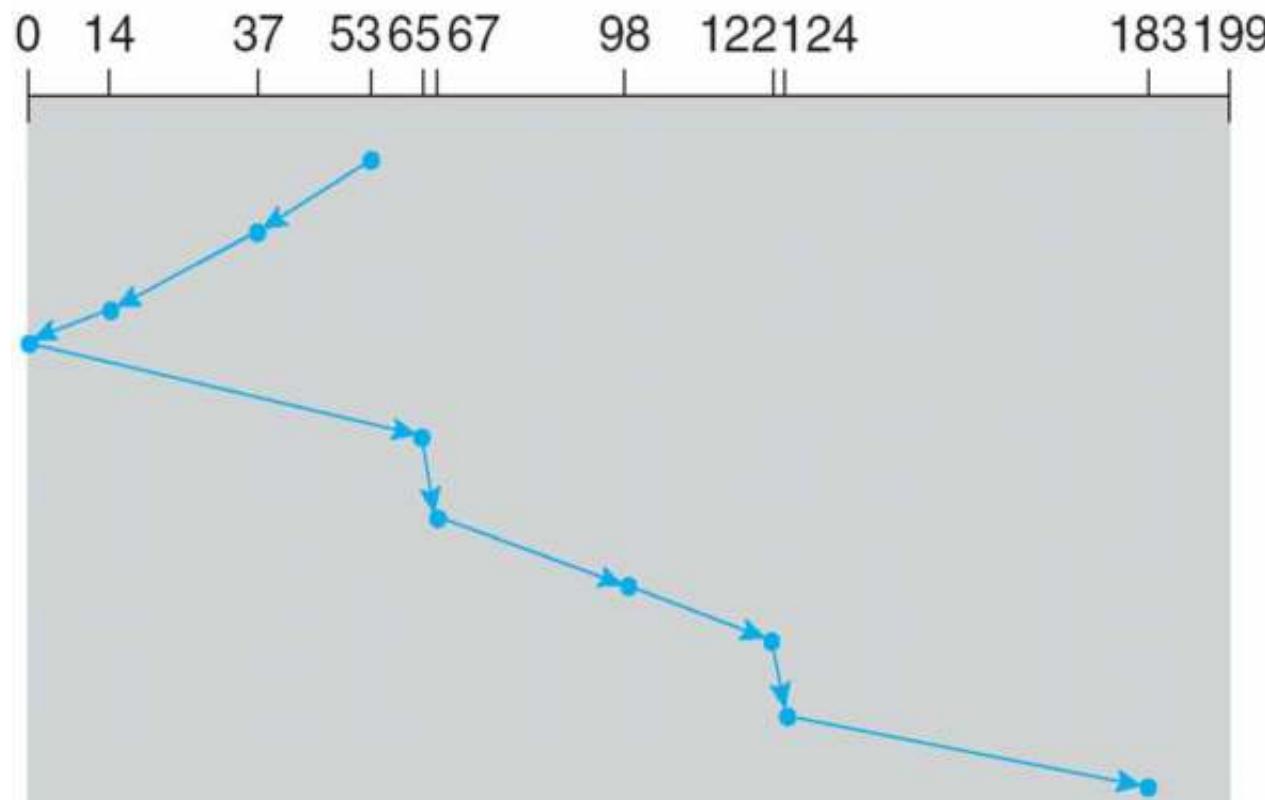
SCAN

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- **SCAN algorithm** sometimes called the **elevator algorithm**

SCAN (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



What if requests are uniformly dense and the head reaches one end and reverses direction?

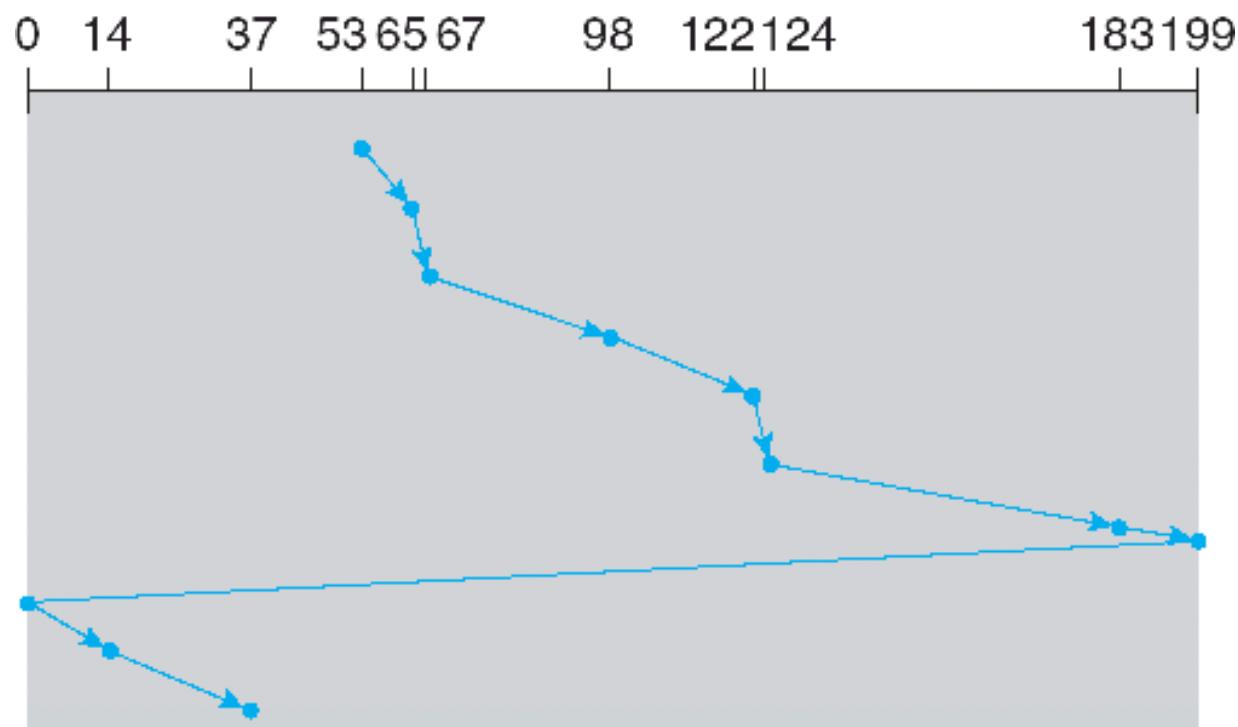
C-SCAN

- Provides a more uniform wait time than SCAN
- The head moves from one end of the disk to the other, servicing requests as it goes
 - When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip
- Treats the cylinders as a circular list that wraps around from the last cylinder to the first one

C-SCAN (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



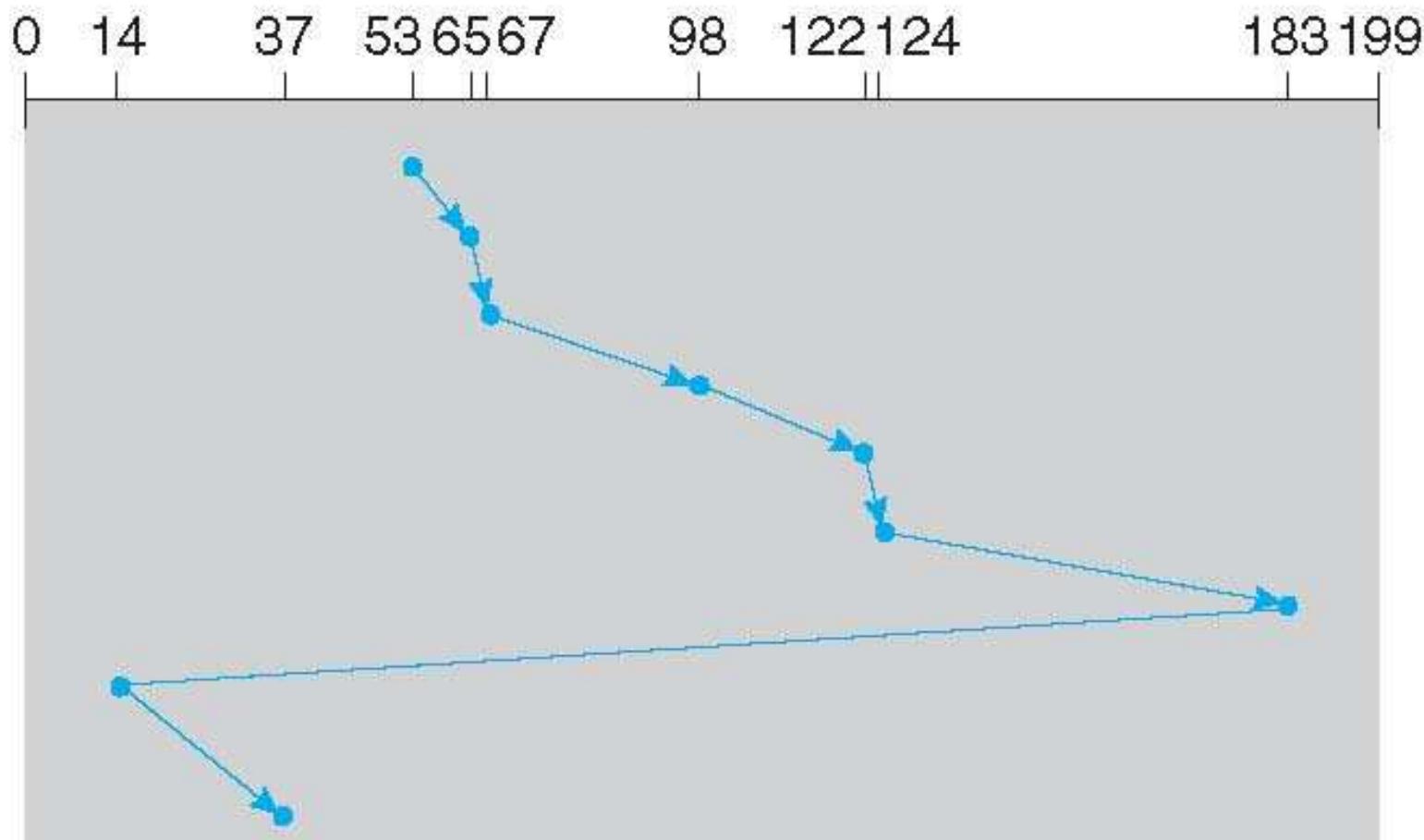
LOOK and C-LOOK

- ❑ LOOK - a version of SCAN
 - ❑ Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk
- ❑ C-LOOK a version of C-SCAN

C-LOOK (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



Storage Systems (2)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems

11/25/2019



Selecting a Disk-Scheduling Algorithm

- SSTF is common and has a natural appeal
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk
- Performance depends on the number and types of requests
- Requests for disk service can be influenced by the file-allocation method
- The disk-scheduling algorithm should be written as a separate module of the operating system, allowing it to be replaced with a different algorithm if necessary
- Either SSTF or LOOK is a reasonable choice for the default algorithm



In-class Work 9

Suppose that a disk drive has 200 cylinders, numbered 0 to 199. The drive is currently serving a request at cylinder 100. The queue of pending requests, in FIFO order, is:

55, 58, 18, 90, 160, 38

Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for each of the following disk-scheduling algorithms? For SCAN and LOOK, assume the head moves towards cylinder 199 first.

FCFS, SSTF, SCAN, LOOK, C-SCAN, C-LOOK

Answer

FCFS: 352

SSTF: 224

SCAN: 280

LOOK: 202

C-SCAN: 388

C-LOOK: 274

Disk Arrays

- ❑ Have many disk drives and therefore many disk arms (rather than a single disk arm):
- ❑ Increase potential throughput



Dependability

- How to decide when a system is operating properly?
- Infrastructure providers now offer Service Level Agreements (SLA) to guarantee that their networking or power service would be dependable
- Systems alternate between 2 states of service with respect to an SLA:
 1. **Service accomplishment**, where the service is delivered as specified in SLA
 2. **Service interruption**, where the delivered service is different from the SLA
 - **Failure** = transition from state 1 to state 2
 - **Restoration** = transition from state 2 to state 1

Dependability

- *Module reliability* = measure of continuous service accomplishment (or time to failure).
 - 2 metrics
 - 1. *Mean Time To Failure (MTTF)* measures Reliability
 - 2. *Failures In Time (FIT)* = $1/MTTF$, the rate of failures
 - Traditionally reported as failures per billion hours of operation
- If modules have *exponentially distributed lifetimes* (age of module does not affect probability of failure), overall failure rate is the sum of failure rates of the modules

Storage Systems (3)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems

12/2/2019



Availability

- *Module availability* measures service as alternate between the 2 states of accomplishment and interruption (number between 0 and 1, e.g. 0.9)
- *Mean Time To Repair (MTTR)* measures Service Interruption
- *Mean Time Between Failures (MTBF) = MTTF+MTTR*
- *Module availability = $MTTF / (MTTF + MTTR)$*

Dependability of Disk Arrays

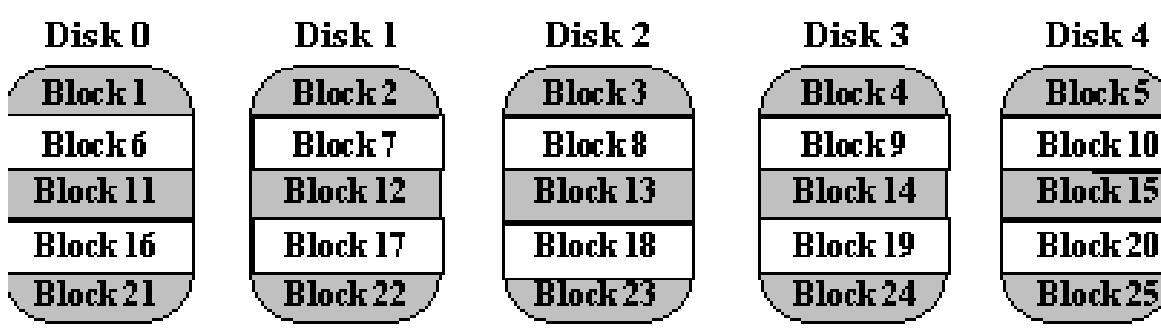
- with many more devices, dependability decreases: N devices generally have 1/Nth of the reliability of a single device.
 - Reliability metric: MTTF
 - $50,000 \text{ Hours} \div 70 \text{ disks} = 700 \text{ hours}$
 - Disk system MTTF: Drops from 6 years to 1 month!
- Result: disk array have many more faults than a small number of large disks

RAID

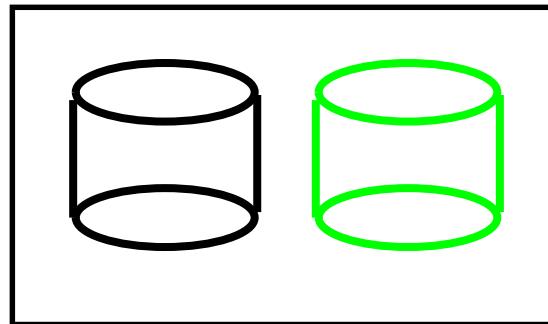
- Add redundant disks to tolerate faults:
 - Dependability increases
 - If a single disk fails: the lost information is reconstructed from the redundant information
- **RAID: redundant array of inexpensive disks**
 - Spread the data over multiple disks: striping
 - If second disk fail while the first one is being repaired, cannot recover
 - Not a problem: MTTF of a disk is tens of years, while MTTR is hours -> redundancy makes the measured reliability of 100 disks much higher than that of a single disk
- Different RAID levels: 0 - 6

RAID 0 – No Redundancy

- Data are striped but there is no redundancy to tolerate disk failure
 - Data is divided into blocks and is spread in a fixed order among all the disks in the array.
- Improves the performance for large access because many disks operate in parallel
- No space overhead, no fault tolerance

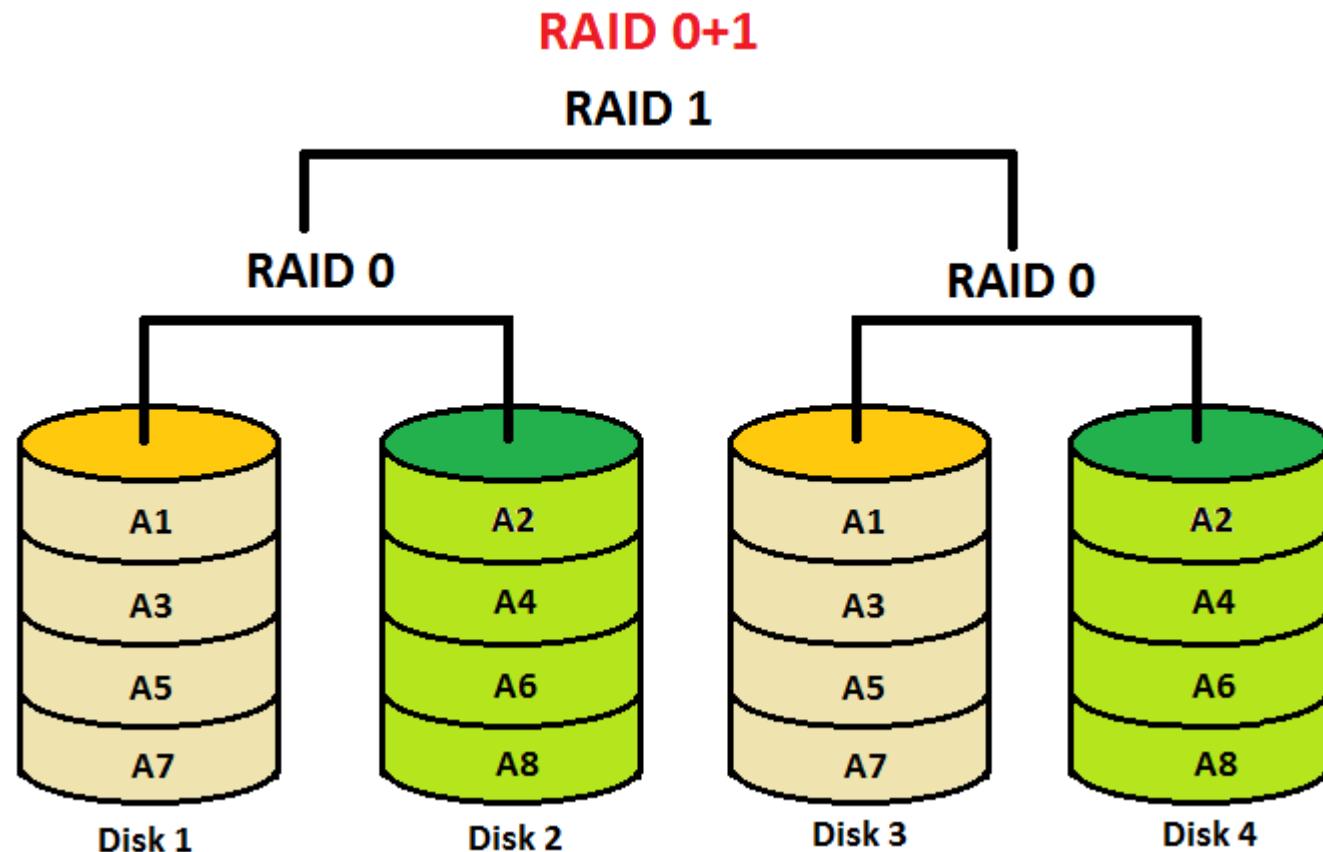


RAID 1: Disk Mirroring/Shadowing

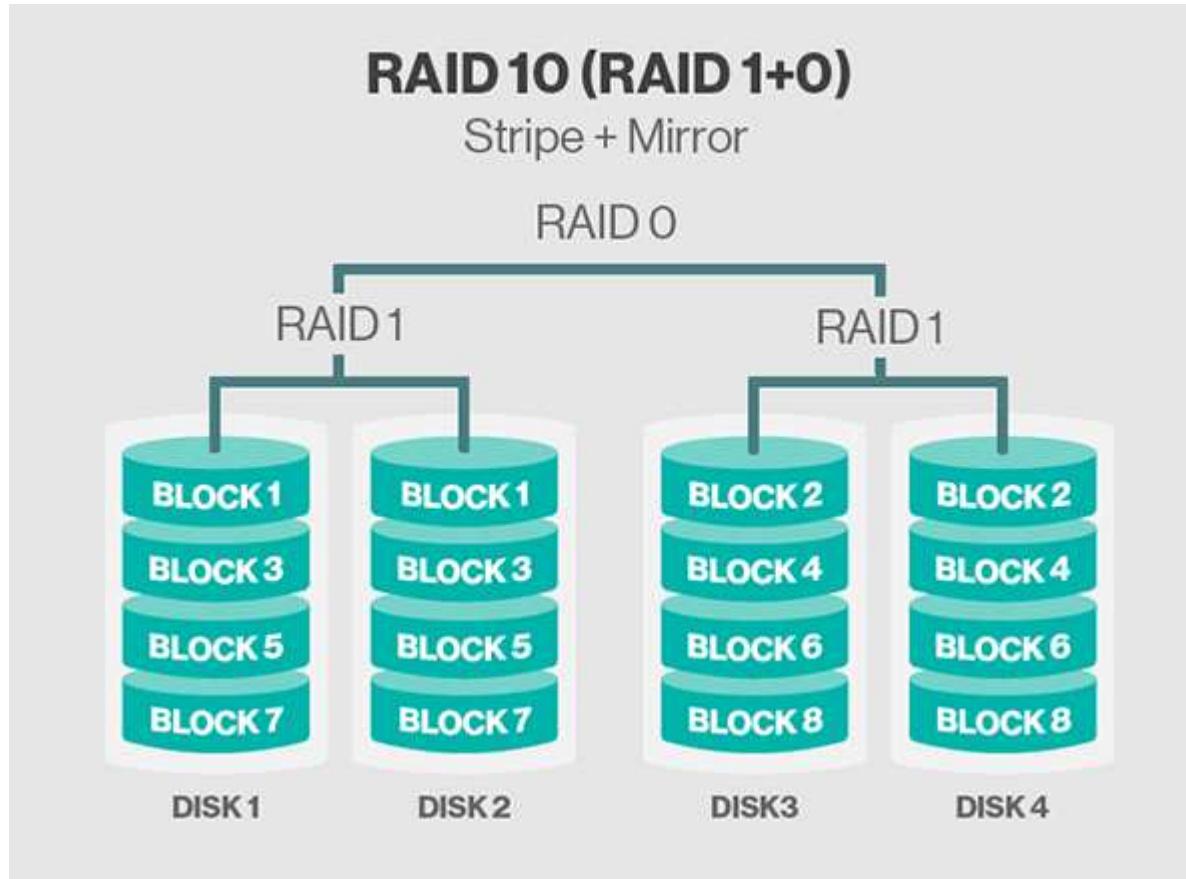


- The disk is fully duplicated onto its “mirror” (or more)
 - Very high availability can be achieved
- Bandwidth sacrifice on write:
 - Logical write = two physical writes
 - Reads may be optimized
- Most expensive solution: 100% capacity overhead
- (RAID 2 no longer used, so skip)

RAID 0+1



RAID 1+0



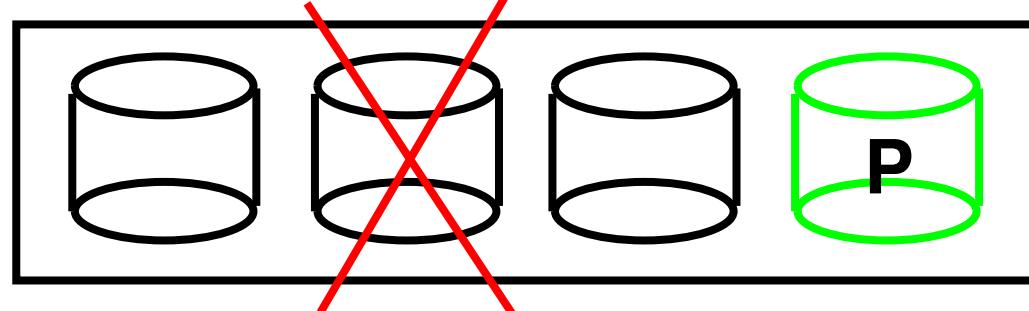
RAID 3 – Parity Disk (Bit-Interleaved)

10010011
11001101
10010011
...

logical record

Striped physical records

P contains sum of other disks per stripe mod 2 (“parity”)
If disk fails, subtract P from sum of other disks to find missing information



1	1	1	1
0	1	0	1
1	0	1	0
0	0	0	0
0	1	0	1
0	1	0	1
1	0	1	0
1	1	1	1

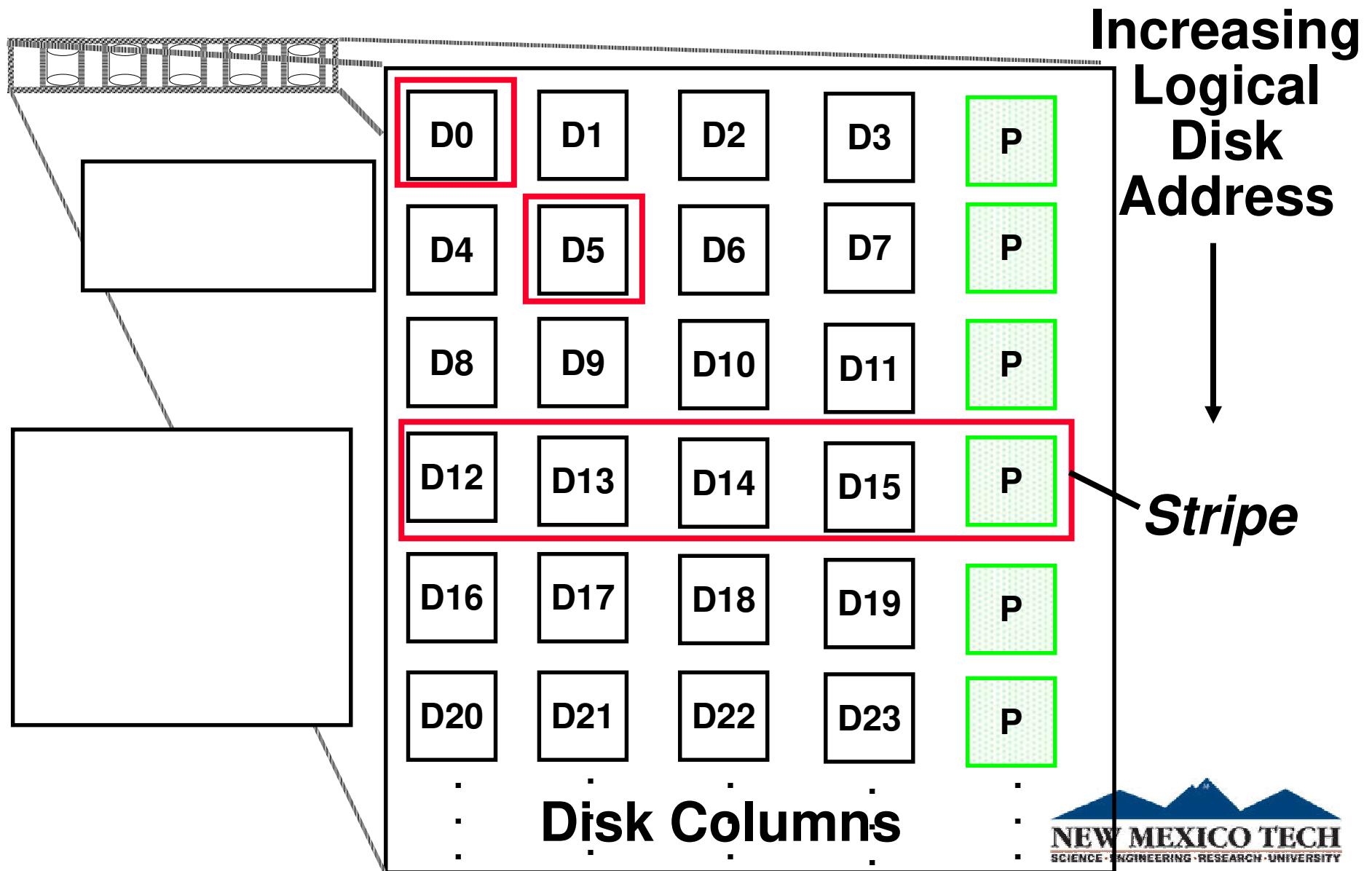
RAID 3 – Parity Disk

- Sum computed across recovery group to protect against hard disk failures, stored in P disk
- Logically, a single high capacity, high transfer rate disk: good for large transfers
- Wider arrays reduce capacity costs
 - 33% capacity cost for parity if 3 data disks and 1 parity disk

Inspiration for RAID 4 (Block-interleaved)

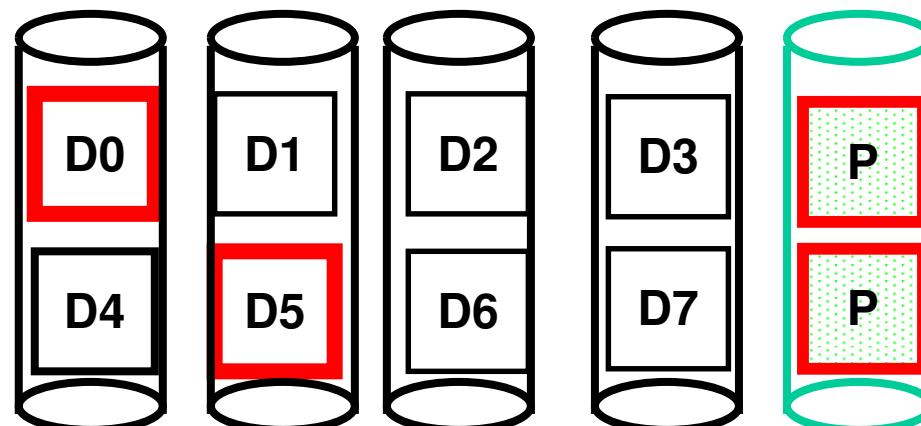
- RAID 3 relies on parity disk to discover errors on Read
- But every sector has an error detection field
- To catch errors on read, rely on error detection field vs. the parity disk
- Allows independent reads to different disks simultaneously

RAID 4 – High I/O Rate Parity



Inspiration for RAID 5

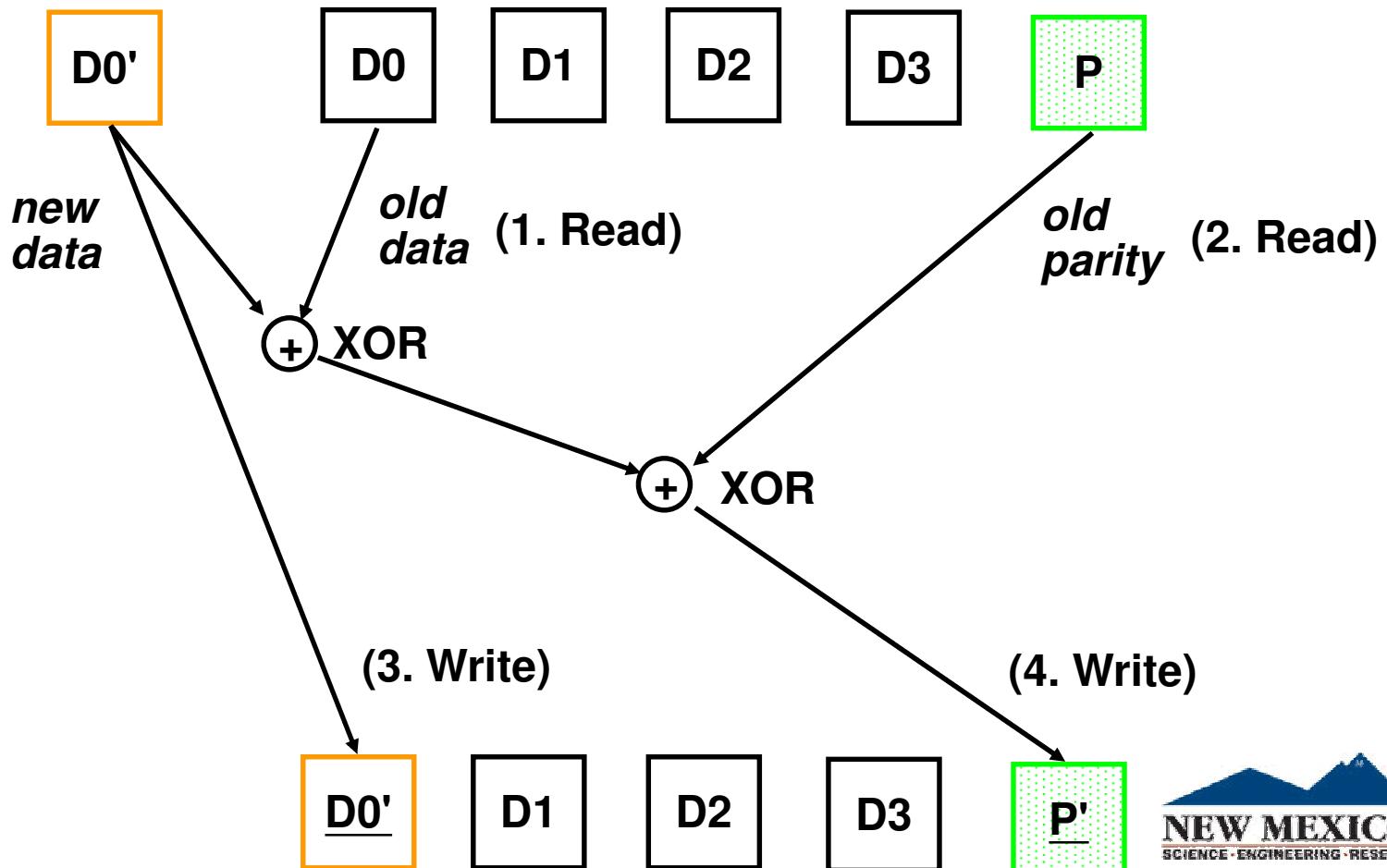
- RAID 4 works well for small reads
- Small writes (write to one disk):
 - Option 1: read other data disks, create new sum and write to Parity Disk
 - Option 2: since P has old sum, compare old data to new data, add the difference to P
- Small writes are limited by Parity Disk: Write to D0, D5 both also write to P disk



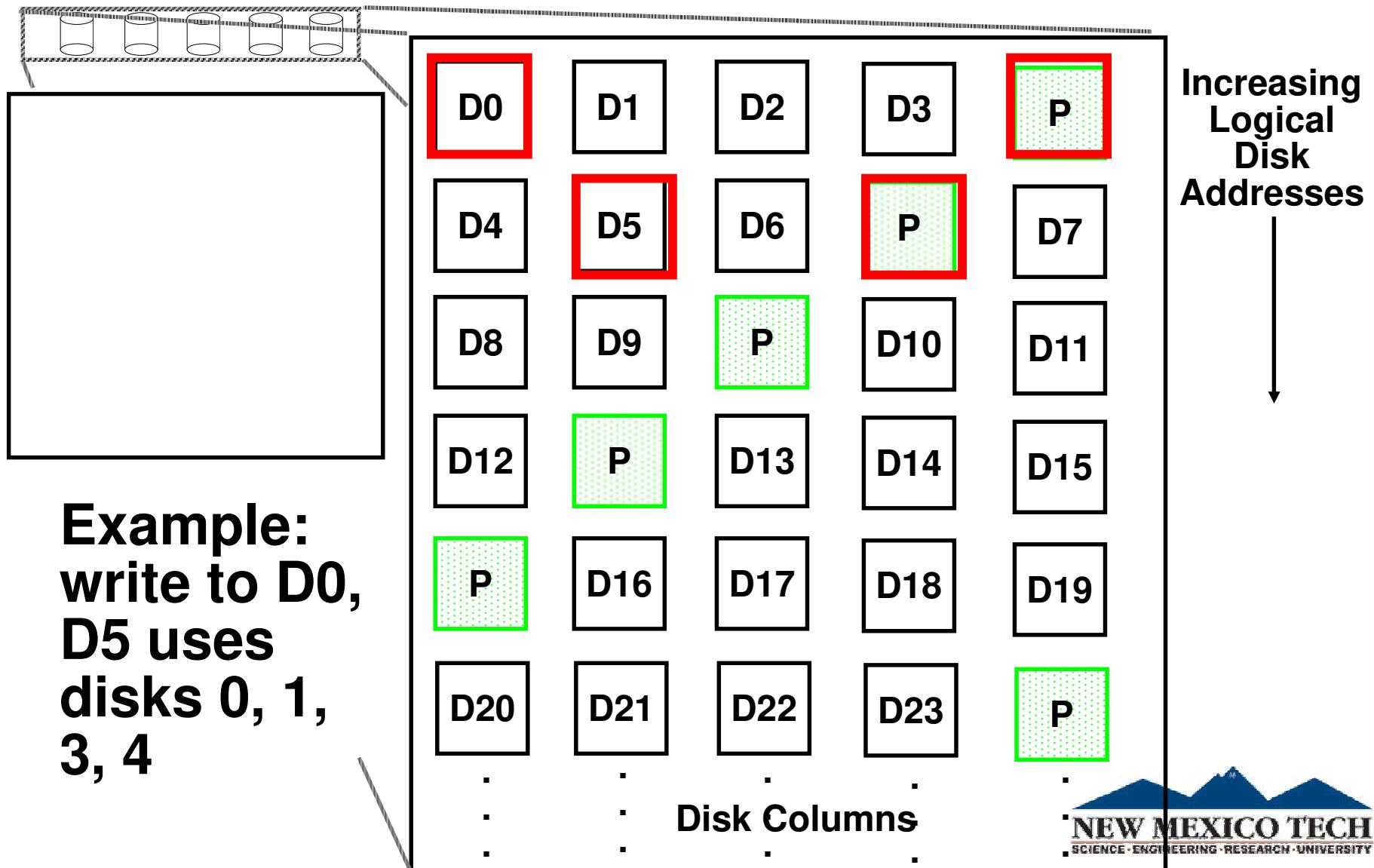
Problems of Disk Arrays: Small Writes

RAID-5: Small Write Algorithm

1 Logical Write = 2 Physical Reads + 2 Physical Writes



RAID 5 – High I/O Interleaved Parity



Storage Systems (4)

Dr. Jun Zheng
CSE325 Principles of Operating
Systems

12/4/2019



RAID 6: Recovering from 2 failures

- Why > 1 failure recovery?
 - operator accidentally replaces the wrong disk during a failure
 - since disk bandwidth is growing more slowly than disk capacity, the MTT Repair a disk in a RAID system is increasing
⇒ increases the chances of a 2nd failure during repair since takes longer
 - reading much more data during reconstruction meant increasing the chance of an uncorrectable media failure, which would result in data loss

RAID 6: Recovering from 2 failures

- Network Appliance's *row-diagonal parity* or *RAID-DP*
- Like the standard RAID schemes, it uses redundant space based on parity calculation per stripe
- Since it is protecting against a double failure, it adds two check blocks per stripe of data.
 - If $p+1$ disks total, $p-1$ disks have data; assume $p = 5$
 - Row parity disk is just like in RAID 4
 - Even parity across the other 4 data blocks in its stripe
- Each block of the diagonal parity disk contains the even parity of the blocks in the same diagonal

Example p = 5

- Row diagonal parity starts by recovering one of the 4 blocks on the failed disk using diagonal parity
 - Since each diagonal misses one disk, and all diagonals miss a different disk, 2 diagonals are only missing 1 block
- Once the data for those blocks is recovered, then the standard RAID recovery scheme can be used to recover two more blocks in the standard RAID 4 stripes
- Process continues until two failed disks are restored

Data Disk 0	Data Disk 1	Data Disk 2	Data Disk 3	Row Parity	Diagonal Parity
0	1	2	3	4	0
1	2	3	4	0	1
2	3	4	0	1	2
3	4	0	1	2	3

RAID Summary

RAID level	Disk failures tolerated, check space overhead for 8 data disks		Pros	Cons	Company products
0	Nonredundant striped	0 failures, 0 check disks	No space overhead	No protection	Widely used
1	Mirrored	1 failure, 8 check disks	No parity calculation; fast recovery; small writes faster than higher RAIDs; fast reads	Highest check storage overhead	EMC, HP (Tandem), IBM
2	Memory-style ECC	1 failure, 4 check disks	Doesn't rely on failed disk to self-diagnose	~ Log 2 check storage overhead	Not used
3	Bit-interleaved parity	1 failure, 1 check disk	Low check overhead; high bandwidth for large reads or writes	No support for small, random reads or writes	Storage Concepts
4	Block-interleaved parity	1 failure, 1 check disk	Low check overhead; more bandwidth for small reads	Parity disk is small write bottleneck	Network Appliance
5	Block-interleaved distributed parity	1 failure, 1 check disk	Low check overhead; more bandwidth for small reads and writes	Small writes → 4 disk accesses	Widely used
6	Row-diagonal parity, EVEN-ODD	2 failures, 2 check disks	Protects against 2 disk failures	Small writes → 6 disk accesses; 2X check overhead	Network Appliance

Question

Consider a RAID level 5 organization comprising five disks, with the parity for sets of four blocks on four disks stored on the fifth disk. How the blocks are accessed in order to perform the following?

- (1) A write of one block of data
- (2) A write of seven continuous blocks of data, assume the blocks begin at a four-block boundary

Answer

- (1) A write of one block of data requires the following: read of the parity block, read of the old data stored in the target block, computation of the new parity based on the differences between the new and old contents of the target block, and write of the parity block and the target block.
- (2) Assume that the seven contiguous blocks begin at a four-block boundary. A write of seven contiguous blocks of data could be performed by writing the seven contiguous blocks, writing the parity block of the first four blocks, reading the eight block, computing the parity for the next set of four blocks and writing the corresponding parity block onto disk.

In-class Work 10

- Consider a 4-drive, 200 GB-per-drive RAID array. What is the available data storage capacity for each of the RAID levels, 0, 1+0, 3, 4, 5, and 6?

I/O Systems

Dr. Jun Zheng
CSE325 Principles of Operating
Systems
12/4/2019

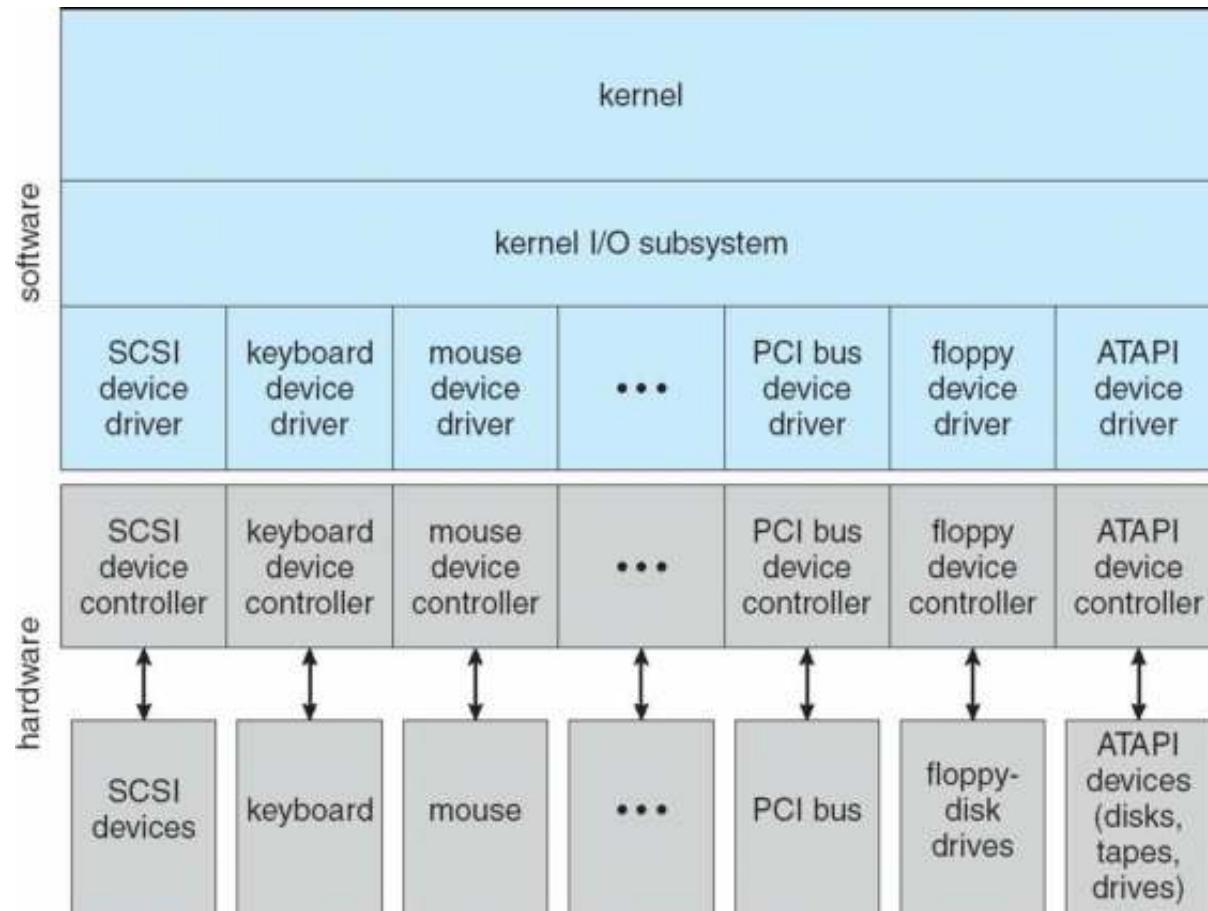


Architecture of I/O Systems

□ Key components

- **System bus:** allows the device to communicate with the CPU, typically shared by multiple devices.
- A device **port** typically consisting of 4 registers:
 - **Status** indicates a device busy, data ready, or error condition
 - **Control:** command to perform
 - **Data-in:** data being sent from the device to the CPU
 - **Data-out:** data being sent from the CPU to the device
- **Controller:** receives commands from the system bus, translates them into device actions, and reads/writes data onto the system bus.
- The device itself
- Traditional devices: disk drive, printer, keyboard, modem, mouse, display
- Non-traditional devices: joystick, robot actuators, flying surfaces of an airplane, fuel injection system of a car, ...

Kernel I/O Subsystem



Memory Mapped I/O

- Memory mapped I/O is a popular way for the device controller to do the I/O transfer.
- The CPU executes I/O requests using the standard data transfer instructions to read and write the device-control registers at their mapped locations in physical memory.

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Device I/O port locations on PCs
(partial)

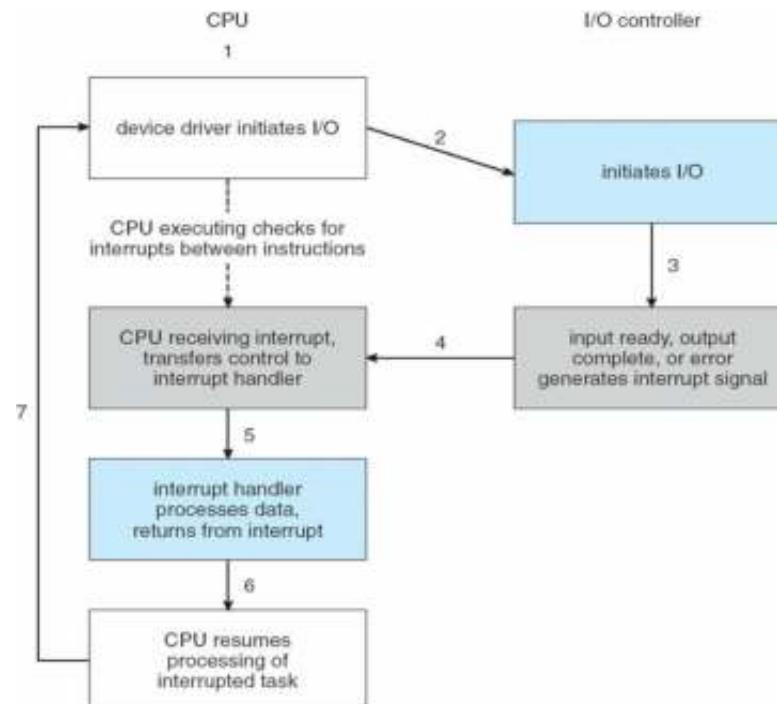
Polling

- CPU busy-waits until the status of the controller is idle.
- CPU sets the command register and data-out if it is an output operation.
- CPU sets status to command-ready => controller sets status to busy.
- Controller reads the command register and performs the command, placing a value in data-in if it is an input command.
- If the operation succeeds, the controller changes the status to idle.
- CPU observes the change to idle and reads the data if it was an input operation.
- Good choice if data must be handled promptly, like for a modem or keyboard.
- What happens if the device is slow compared to the CPU?



Interrupts

- Rather than using busy waiting, the device can interrupt the CPU when it completes an I/O operation.
- On an I/O interrupt:
 - Determine which device caused the interrupt.
 - If the last command was an input operation, retrieve the data from the device register.
 - Start the next operation for that device.



Intel X86 Event-Vectors

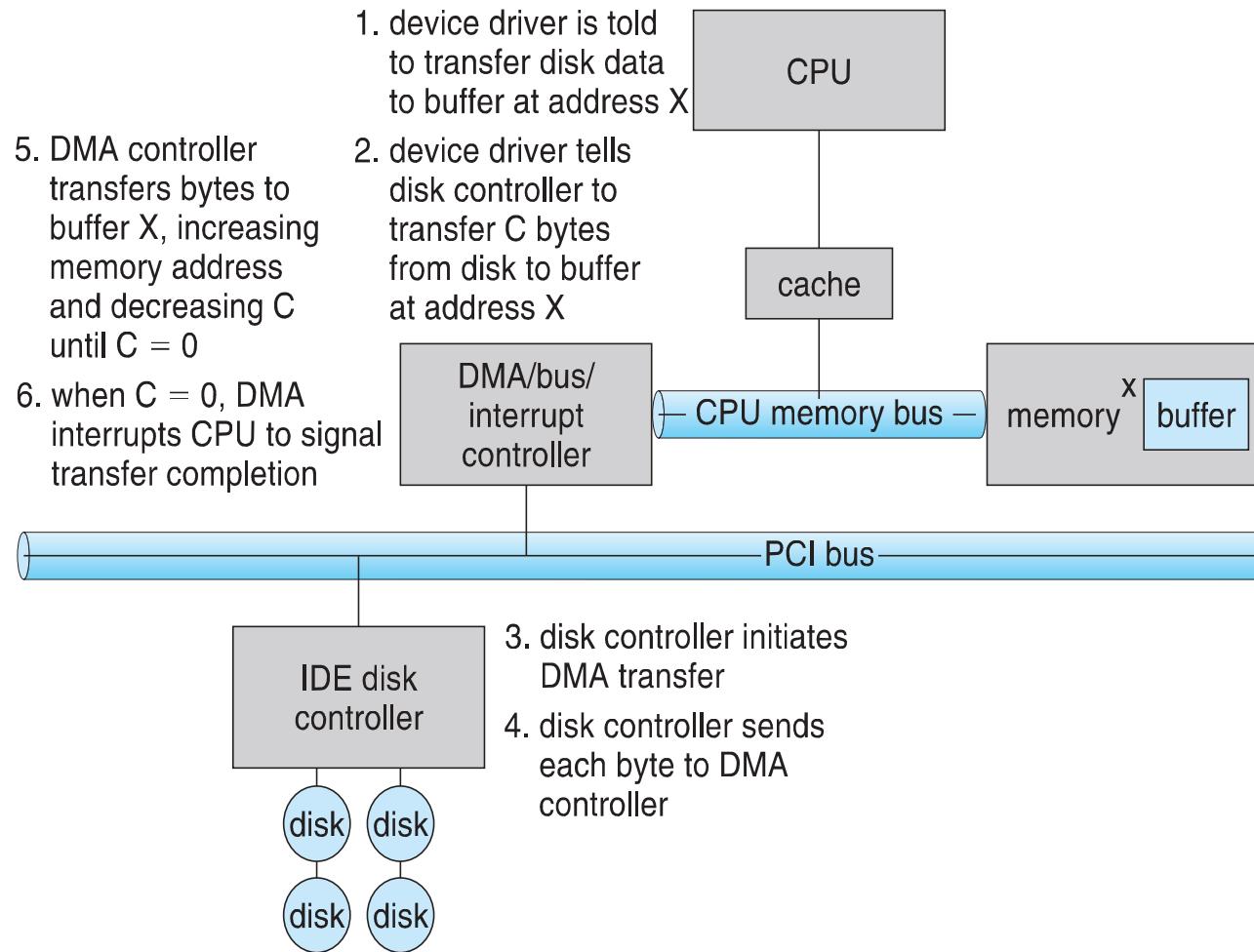
- Unmaskable interrupt: reserved for events such as unrecoverable memory errors
- Maskable: it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted.

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Direct Memory Access

- For devices that transfer large volumes of data at a time (like a disk block), it is expensive to have the CPU retrieve these one byte at a time.
- **Solution:** Direct memory access (DMA)
 - Use a sophisticated DMA controller that can write directly to memory. Instead of data-in/data-out registers, it has an address register.
 - The CPU tells the DMA the locations of the source and destination of the transfer.
 - The DMA controller operates the bus and interrupts the CPU when the entire transfer is complete, instead of when each byte is ready.
 - The DMA controller and the CPU compete for the memory bus, slowing down the CPU somewhat, but still providing better performance than if the CPU had to do the transfer itself.

Six Step Process to Perform DMA Transfer



Examples of I/O Device Types

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read-write	CD-ROM graphics controller disk