



Operating Systems

Introduction

Seyyed Ahmad Javadi

sajavadi@aut.ac.ir

Fall 2021

My Background and Contact Details

- Seyyed Ahmad Javadi
 - PhD from New York State University at Stony Brook
 - Postdoc from University of Cambridge
 - Interest: Cloud computing, operating systems, performance analysis
-
- Office: CE department, 3rd floor
 - Email: sajavadi@aut.ac.ir
 - Include CE303 in your email subject
 - Home page: <https://ce.aut.ac.ir/~sajavadi/>

Course Introduction

- Saturday and Monday (13:30-15)
 - Attend class on time
- Course web page
 - Check the webpage on regular basis
 - Everything will be posted on CW
 - Post All your Questions on CW Forums
 - ▶ Check forum history before posting any question
- Office hours and TA classes
 - TBD



Textbook

- **Operating System Concepts**, 10th Edition, Wiley publishing
 - By A. Silberschatz, P. Galvin, & G. Gagne
- Other References:
 - Operating systems: design & implementation,
 - ▶ By A. Tanenbaum and A. Woodhull, 3rd edition, 2006.
 - Operating systems: internals and design principles,
 - ▶ By W. Stallings, 5th edition, 2005.



Grading

Section	Score	Considerations
assignments	2.5	five homework
midterm exam	4	1400/08/22
project	4 + 1	in three phases
final exam	8	1400/10/20
quiz	1	two quizzes
class participation	0.5	ask/answer questions be active in the course webpage

Harsh penalty for plagiarism and cheating

Project

- Adding new features to XV6 created in MIT's Operating System Engineering course; isn't this exciting 😊
 - XV6 is used in most of the well-known universities.
 - <https://pdos.csail.mit.edu/6.828/2012/xv6.html>
- Three Phases:
 - Phase 1: getting to know XV6 basics (solo work)
 - Phase 2: getting to know XV6 advanced features (solo work)
 - Phase 3: final project (teamwork)

Syllabus

- Introduction to operating systems
- Process management
 - Threads
 - Synchronization
 - Scheduling
- Memory management
- Storage management
- Protection and security



Copyright Notice

Slides are based on the slides of the main **textbook**.

Silberschatz

<https://www.os-book.com/OS10/slide-dir/index.html>



What is an Operating System?

- A program that acts as an **intermediary** between a user of a computer and the computer hardware.
 - User can execute programs **conveniently & efficiently**
- Operating system goals:
 - Execute user programs and make solving user problems easier.
 - Make the computer system convenient to use.
 - Use the computer hardware in an efficient manner.

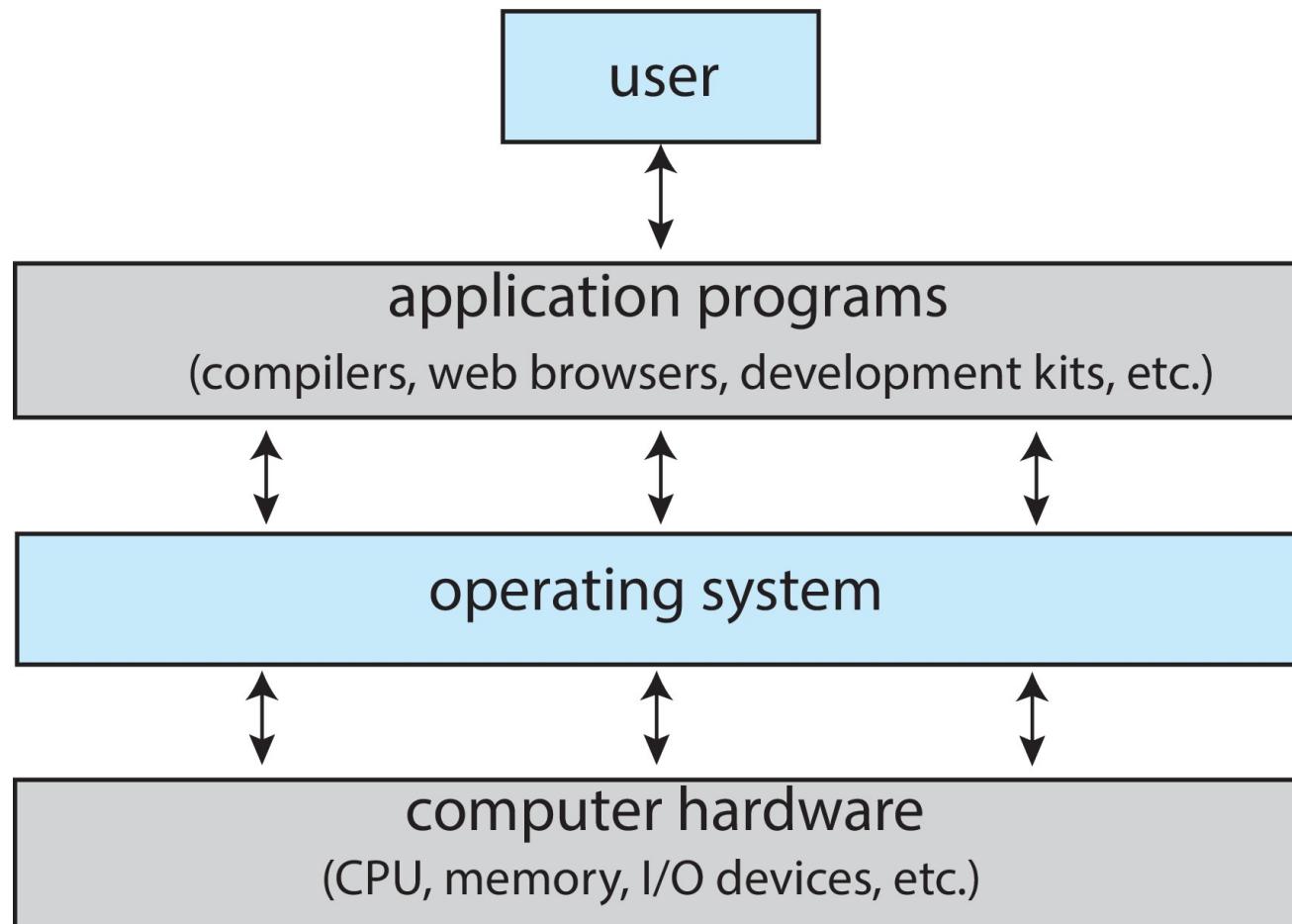


OS: Mandatory or Optional?

- **Can we run a computer without an operating system?**
 - Yes, earliest computers did not have OS.

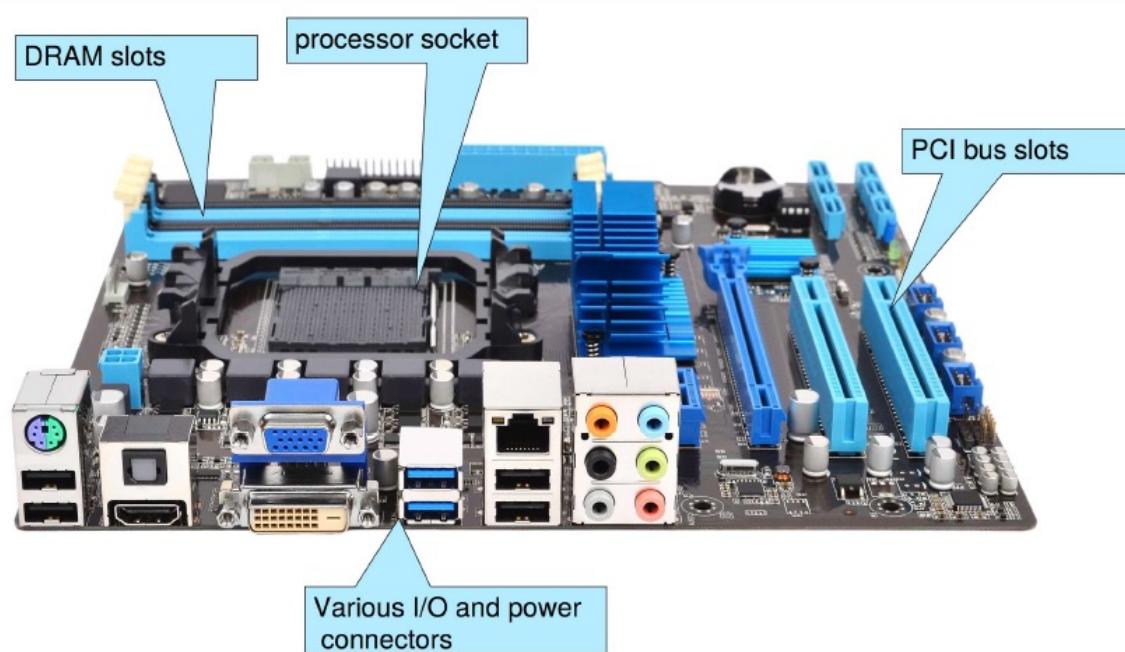
- **What does a compute without an OS look like?**
 - Machines tasked with one program at a time.
 - ▶ Cannot read a pdf while listening to a music.
 - Each program has a lot of work to do.
 - ▶ Where to load a program
 - ▶ IO access

Abstract View of Components of Computer



PC Motherboard

Consider the desktop PC motherboard with a processor socket shown below:



This board is a fully-functioning computer, once its slots are populated. It consists of a processor socket containing a CPU, DRAM sockets, PCIe bus slots, and I/O connectors of various types. Even the lowest-cost general-purpose CPU contains multiple cores. Some motherboards contain multiple processor sockets. More advanced computers allow more than one system board, creating NUMA systems.

Operating System Story

- **Vital goal of a computer system**
 - Execute user program and make solving user problem easier.

- **Shall user program use hardware directly?**
 - Hardware alone is ***not easy to use.***
 - Application programs require certain ***common operations.***
 - ▶ Example: I/O operations

Common functions of controlling and allocating resources brought together into one piece called **OS**

Operating System Definition (cont.)

- No universally accepted definition.
- “The one program running at all times on the computer” is the **kernel**, part of the operating system.
- Everything else is either
 - A **system program** (ships with the operating system, but not part of the kernel) , or
 - An **application program**, all programs not associated with the operating system.





Operating Systems

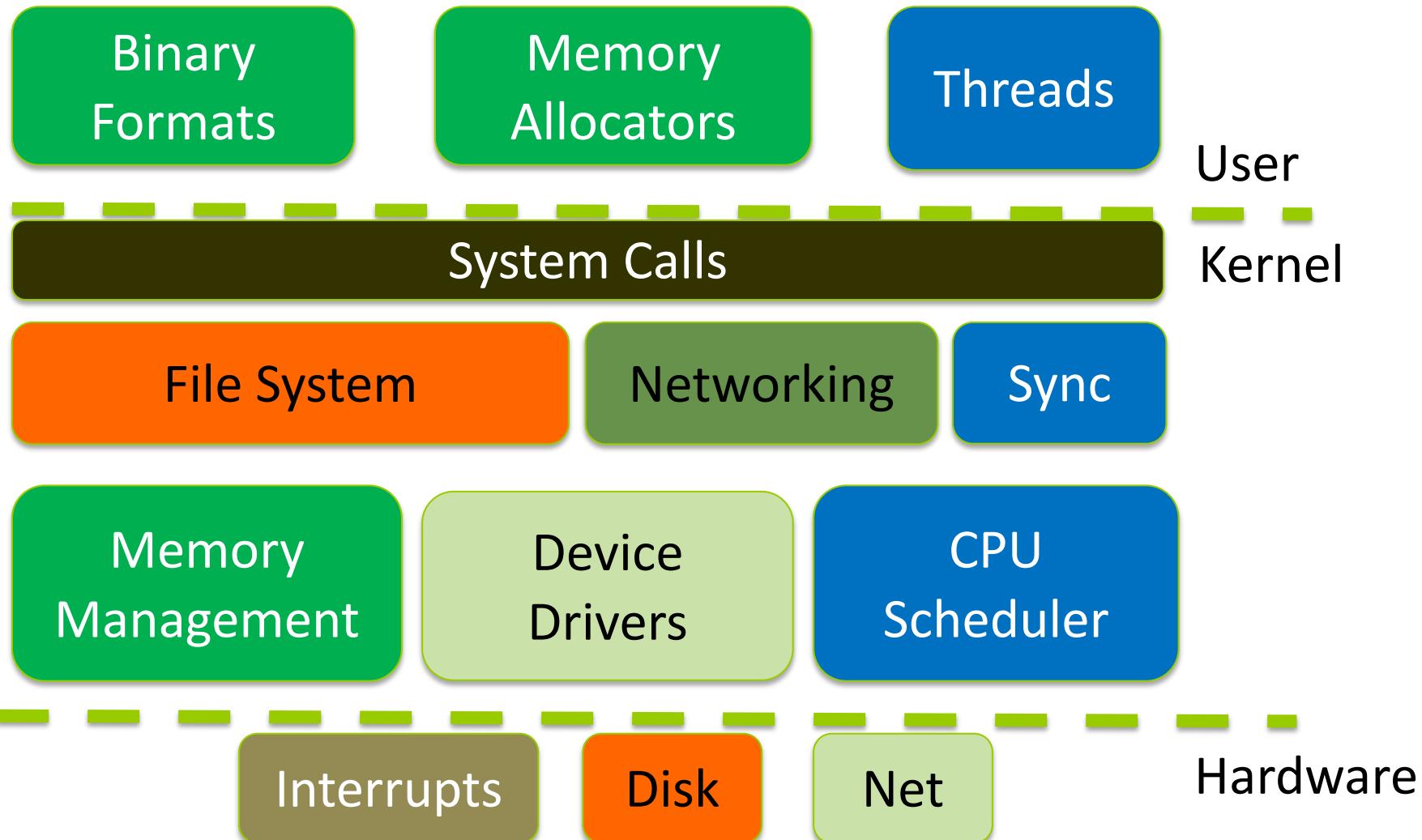
Computer System Organization

Seyyed Ahmad Javadi

sajavadi@aut.ac.ir

Fall 2021

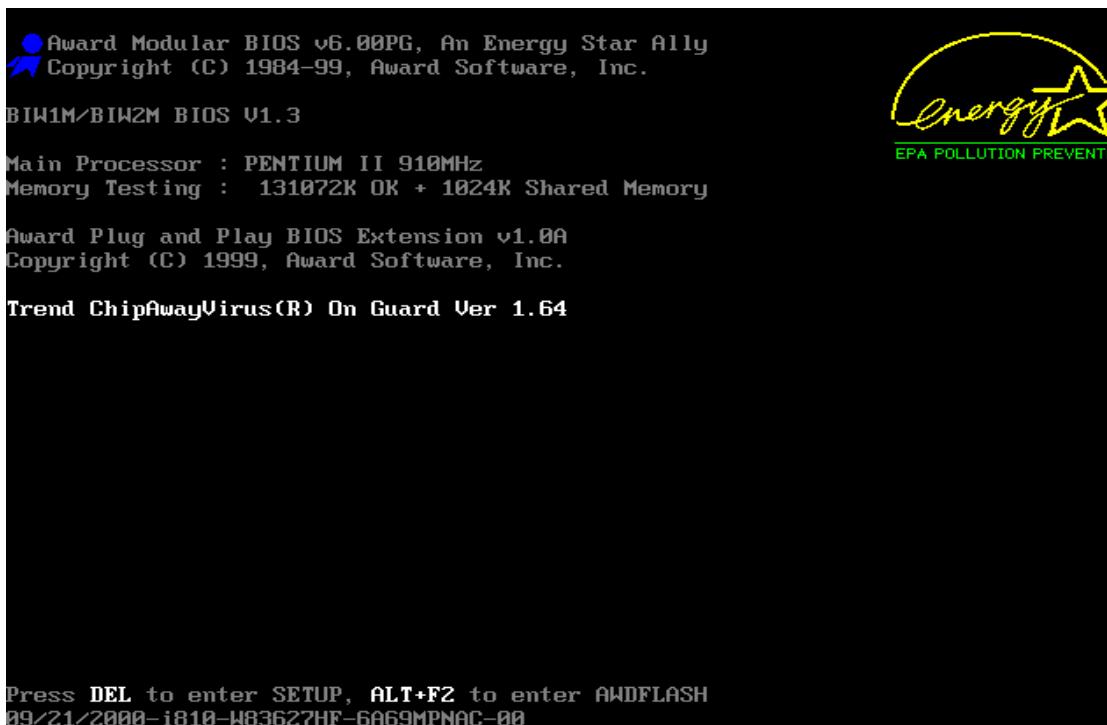
A logical view of the OS



Computer Startup

■ **Bootstrap program** is loaded at power-up or reboot.

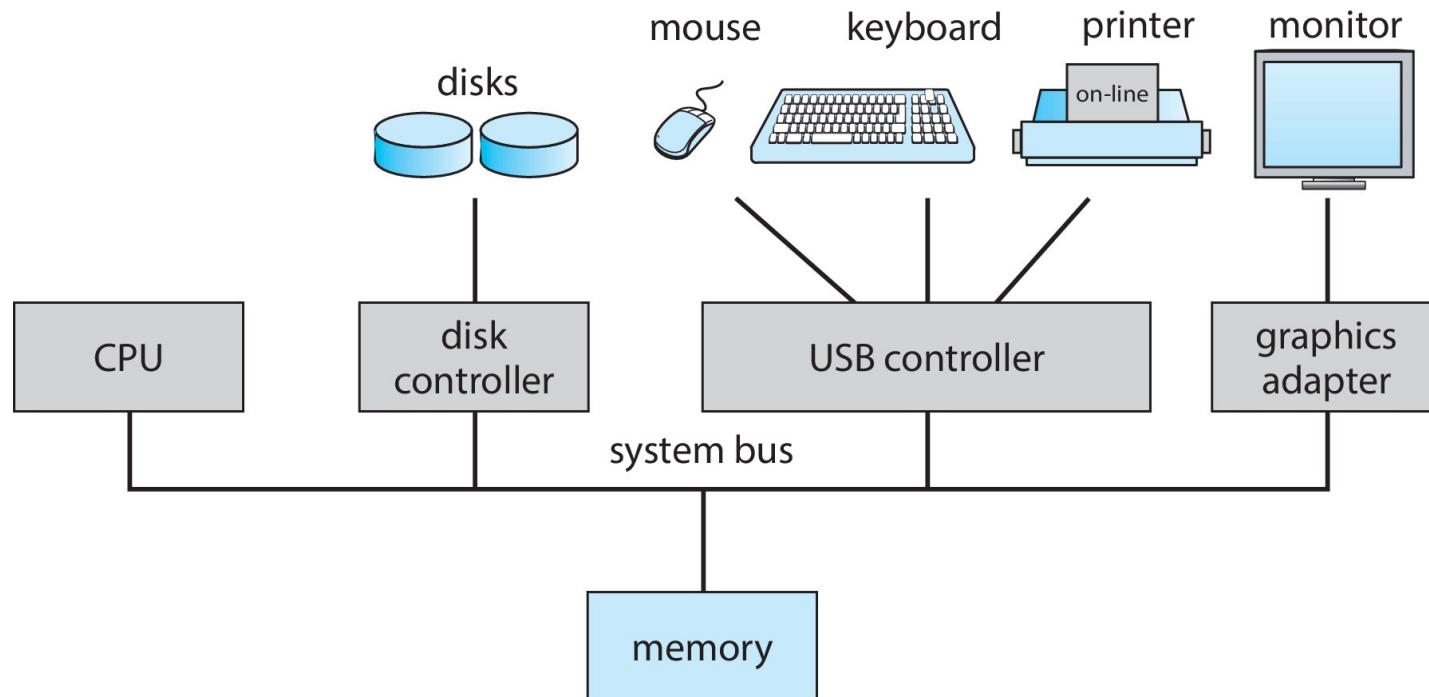
- Typically stored in ROM or EPROM, generally known as **firmware**.
- Initializes all aspects of system.
- Loads operating system kernel and starts execution.



Computer System Organization

■ Computer-system operation

- One or more CPUs, device controllers connect through common **bus** providing access to shared memory.
- Concurrent execution of CPUs and devices competing for memory cycles.

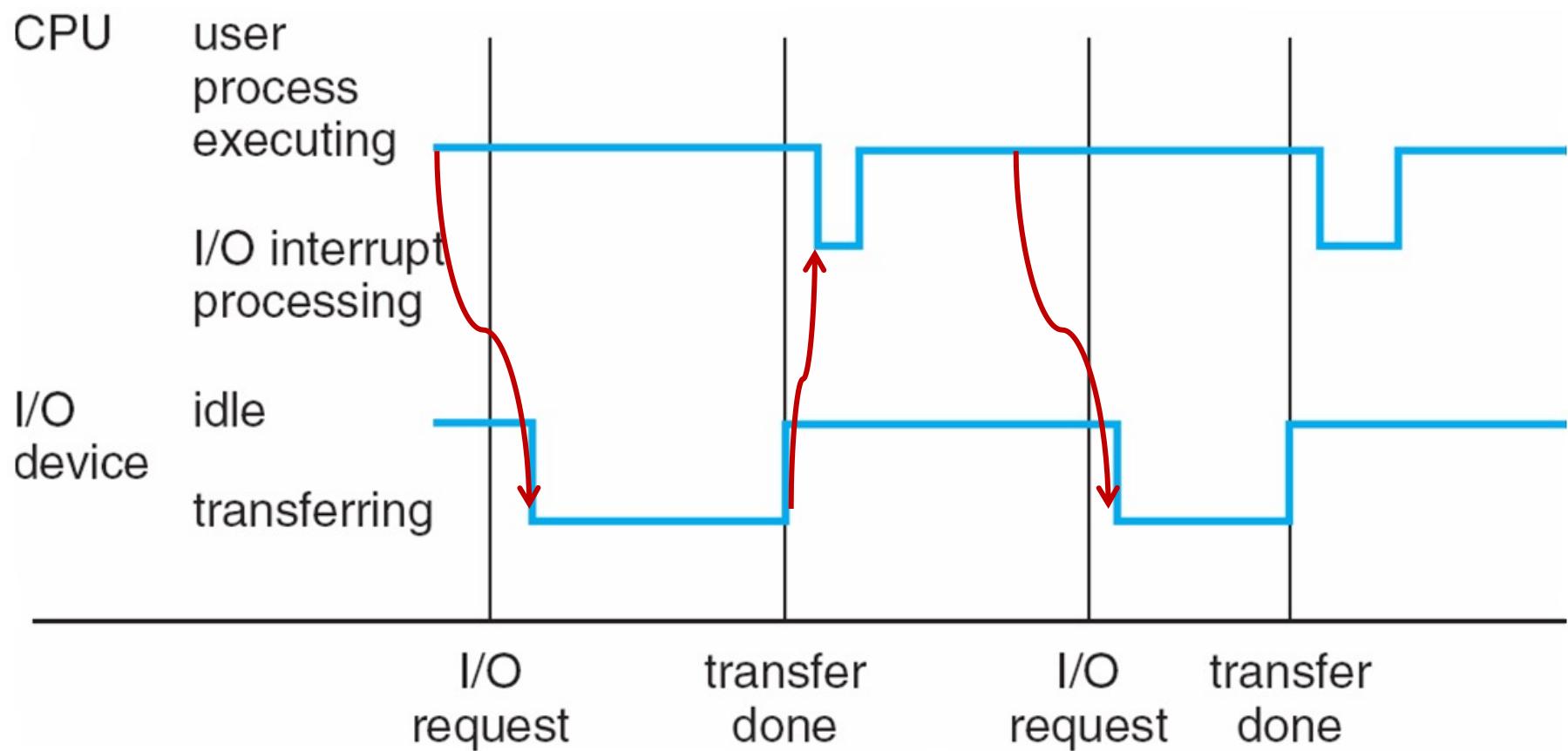


Computer-System Operation

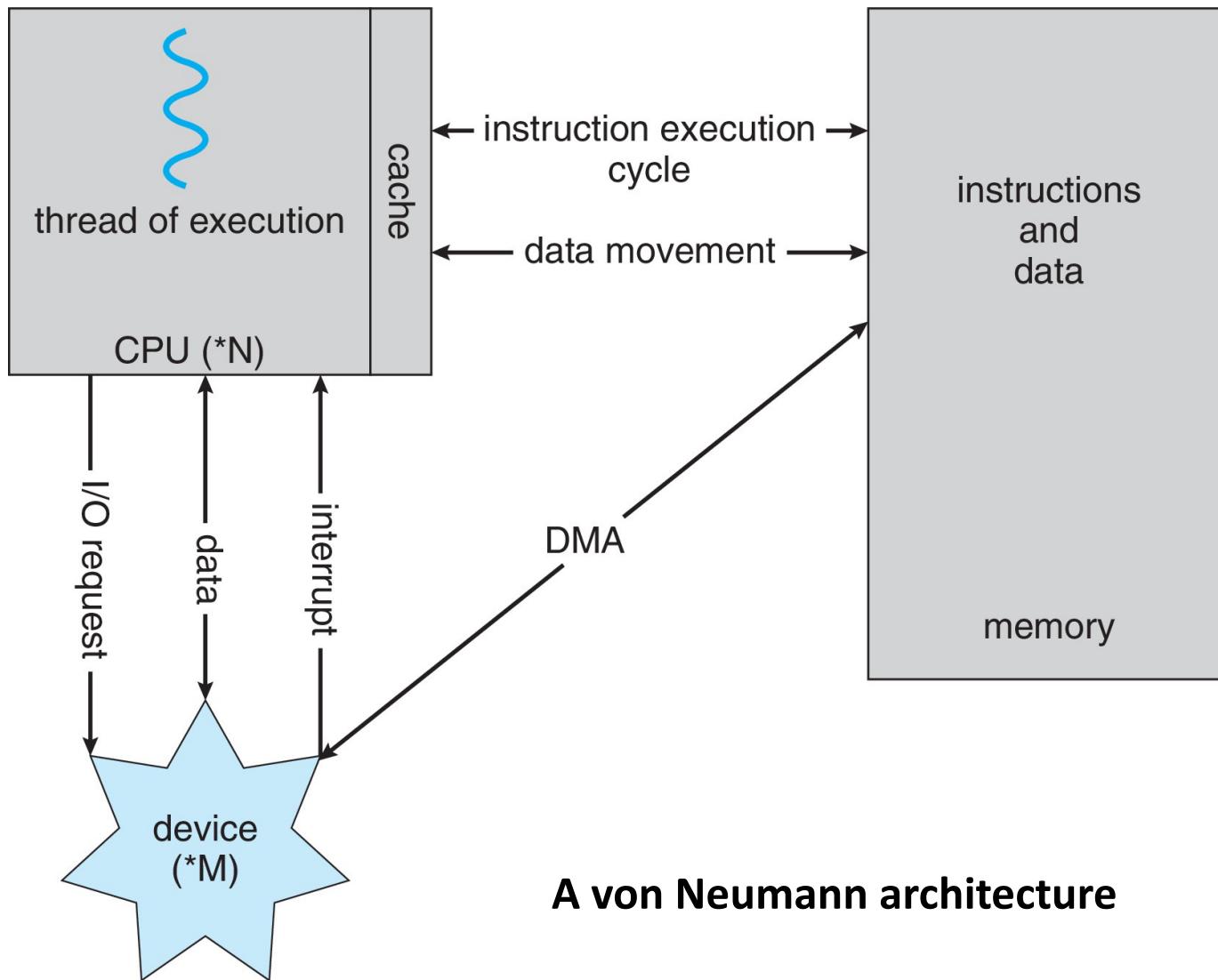
- Each device controller is in charge of a particular device type (e.g., disk drives, audio devices).
- Each device controller has a local buffer.
- I/O devices and the CPU can execute concurrently
- I/O: device \leftrightarrow local buffer of controller.
- Device controller informs CPU that it has finished its operation by causing an **interrupt**.
- **CPU moves data**
 - Main memory \leftrightarrow local buffers



Interrupt Timeline



How a Modern Computer Works



Direct Memory Access Structure

- Used for **high-speed I/O devices** able to transmit information at close to memory speeds.
- Device controller transfers blocks of data from buffer storage directly to main memory **without CPU intervention**.
- Only one **interrupt is generated per block**, rather than the one interrupt per byte.





Operating Systems

Multiprogramming and Dual-mode

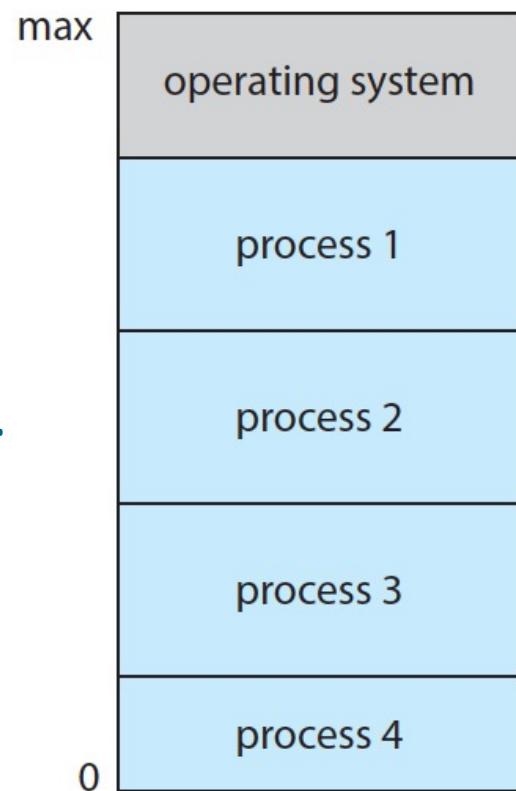
Seyyed Ahmad Javadi

sajavadi@aut.ac.ir

Fall 2021

Multiprogramming (Batch System) (cont.)

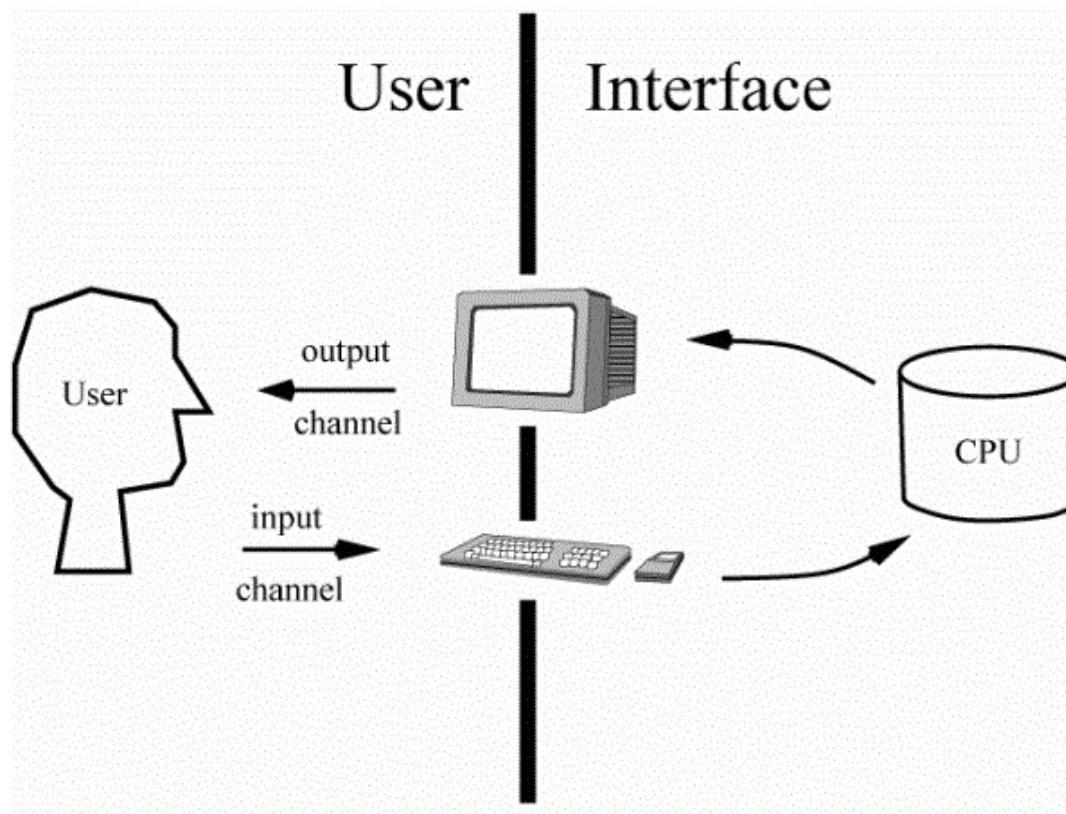
- Multiprogramming organizes multiple jobs (code and data) -->
 - CPU always has one to execute.
- A subset of total jobs in system is kept in memory.
- One job selected and run via **job scheduling**.
- When job has to wait (I/O for example), OS switches to another job.



Memory layout for a multiprogramming system

Multiprogramming (Batch System)

- Single user/program cannot always keep CPU and I/O devices busy.



Multiprogramming (Batch System) (cont.)

- Single user/program cannot always keep CPU and I/O devices busy.
- Examples

Program	CPU-intensive	Memory-intensive	I/O-intensive
Random Number Generator	?	?	?
Microsoft word	?	?	?
QuickTime Player (a long 4K video)	?	?	?

Multitasking (Timesharing)

- A logical extension of Batch systems
- The CPU ***switches jobs so frequently*** that users can interact with each job while it is running, creating **interactive** computing.
 - Response time should be < 1 second.
 - Each user has at least one program executing in memory \Rightarrow process.
 - If several jobs ready to run at the same time \Rightarrow CPU scheduling.
 - If processes don't fit in memory, **swapping** moves them in&out to run.
 - **Virtual memory** allows execution of processes not completely in memory.

<https://www.geeksforgeeks.org/difference-between-job-task-and-process/>



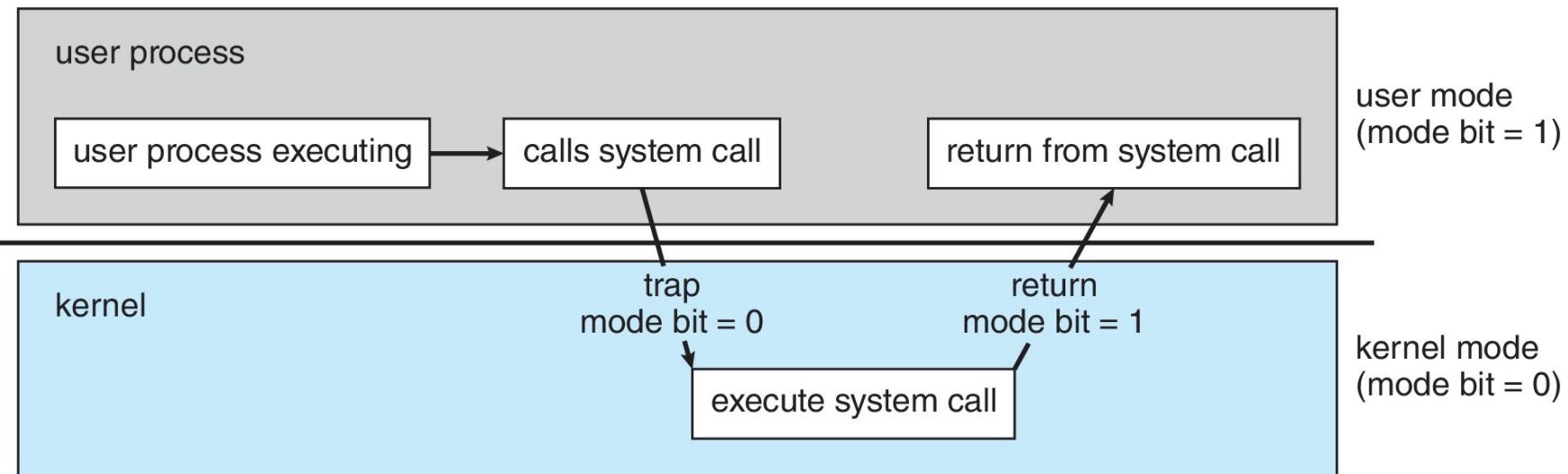
Dual-mode Operation

- **Dual-mode** operation allows OS to protect itself and other system components.
 - **User mode** and **kernel mode**
- **Mode bit** provided by hardware
 - Provides ability to distinguish when system is running user code or kernel code.
 - When a user is running \Rightarrow mode bit is “user”.
 - When kernel code is executing \Rightarrow mode bit is “kernel”.



Dual-mode Operation (Cont.)

- How do we guarantee that user does not explicitly set the mode bit to “kernel”?
 - System call changes mode to kernel, return from call resets it to user.



Types of Instructions

- Instructions are divided into two categories:
 - The ***non-privileged instruction*** instruction is an instruction that *any application or user can execute.*
 - The ***privileged instruction*** is an instruction that *can only be executed in kernel mode.*
- Instructions are divided in this manner because privileged instructions ***could harm the kernel.***

<http://web.cs.ucla.edu/classes/winter13/cs111/scribe/4a/>

Examples of instructions

Instruction	Type
Reading the status of Processor	?
Set the Timer	?
Sending the final printout of Printer	?
Remove a process from the memory	?



Examples of non-privileged instructions

- Reading the status of Processor
- Reading the System Time
- Sending the final printout of Printer

<https://www.geeksforgeeks.org/privileged-and-non-privileged-instructions-in-operating-system/>



Examples of privileged instructions

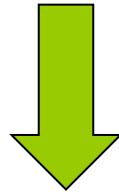
- I/O instructions and halt instructions
- Turn off all Interrupts
- Set the timer
- Context switching
- Clear the memory or remove a process from the memory
- Modify entries in the device-status table

<https://www.geeksforgeeks.org/privileged-and-non-privileged-instructions-in-operating-system/>



Privileged instructions

If an attempt is made to execute a privileged instruction in user mode



The hardware *does not execute the instruction* but rather treats it as *illegal* and *traps* it to the *operating system*.



Operating Systems

System Calls

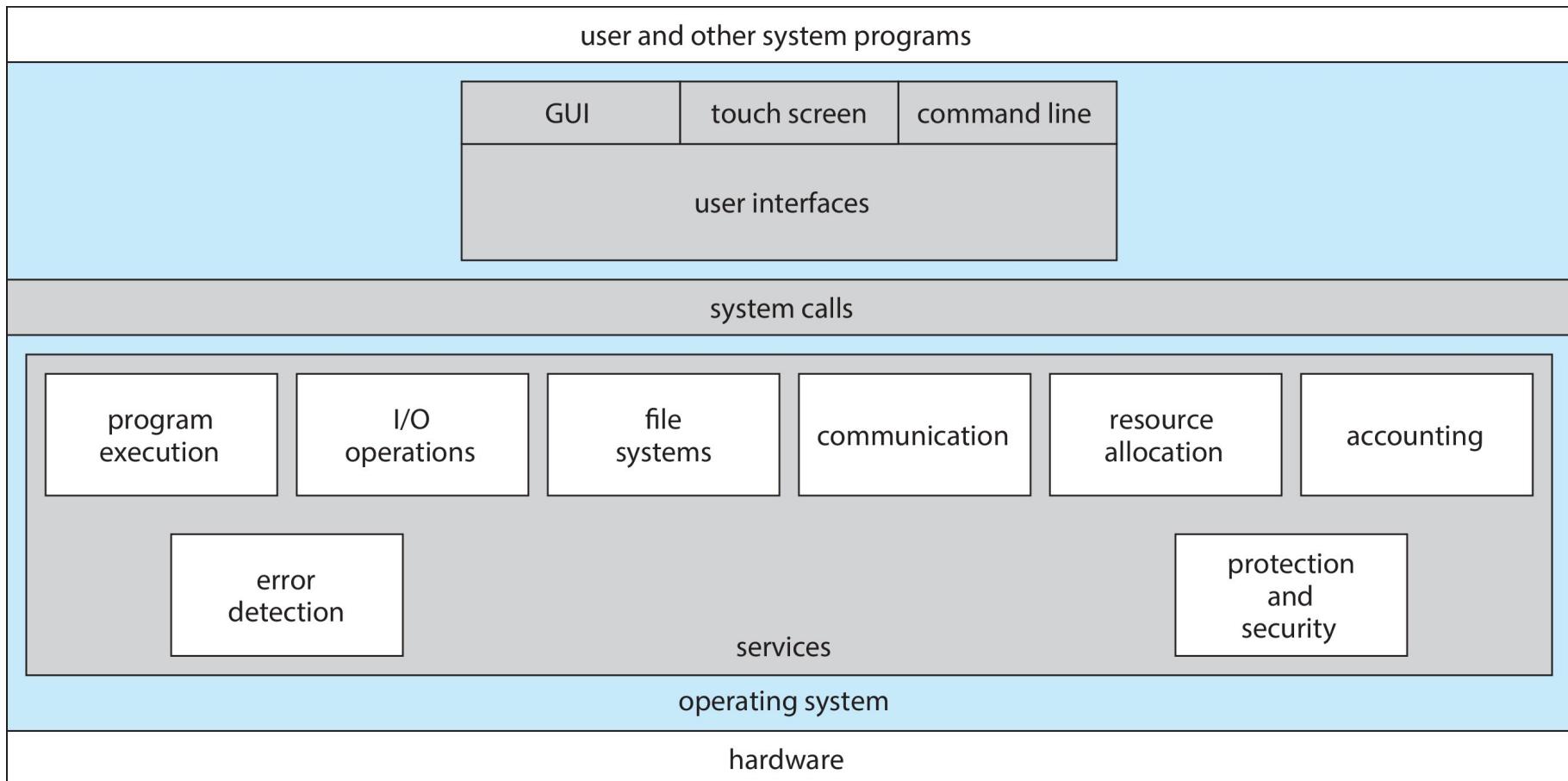
Seyyed Ahmad Javadi

sajavadi@aut.ac.ir

Fall 2021

System Calls

- Programming interface to the **services** provided by the OS.



System Calls (cont.)

- Typically written in a high-level language (C or C++ or Assembly).
- 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 (Win API)
 - [POSIX](#) API for POSIX-based systems (including virtually all versions of UNIX, Linux (*unistd.h*), and Mac OS X)
 - [Java](#) API for the Java virtual machine (JVM).

Note that the system-call names used throughout this text are generic.

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

Example of Standard API (Cont.)

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 into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

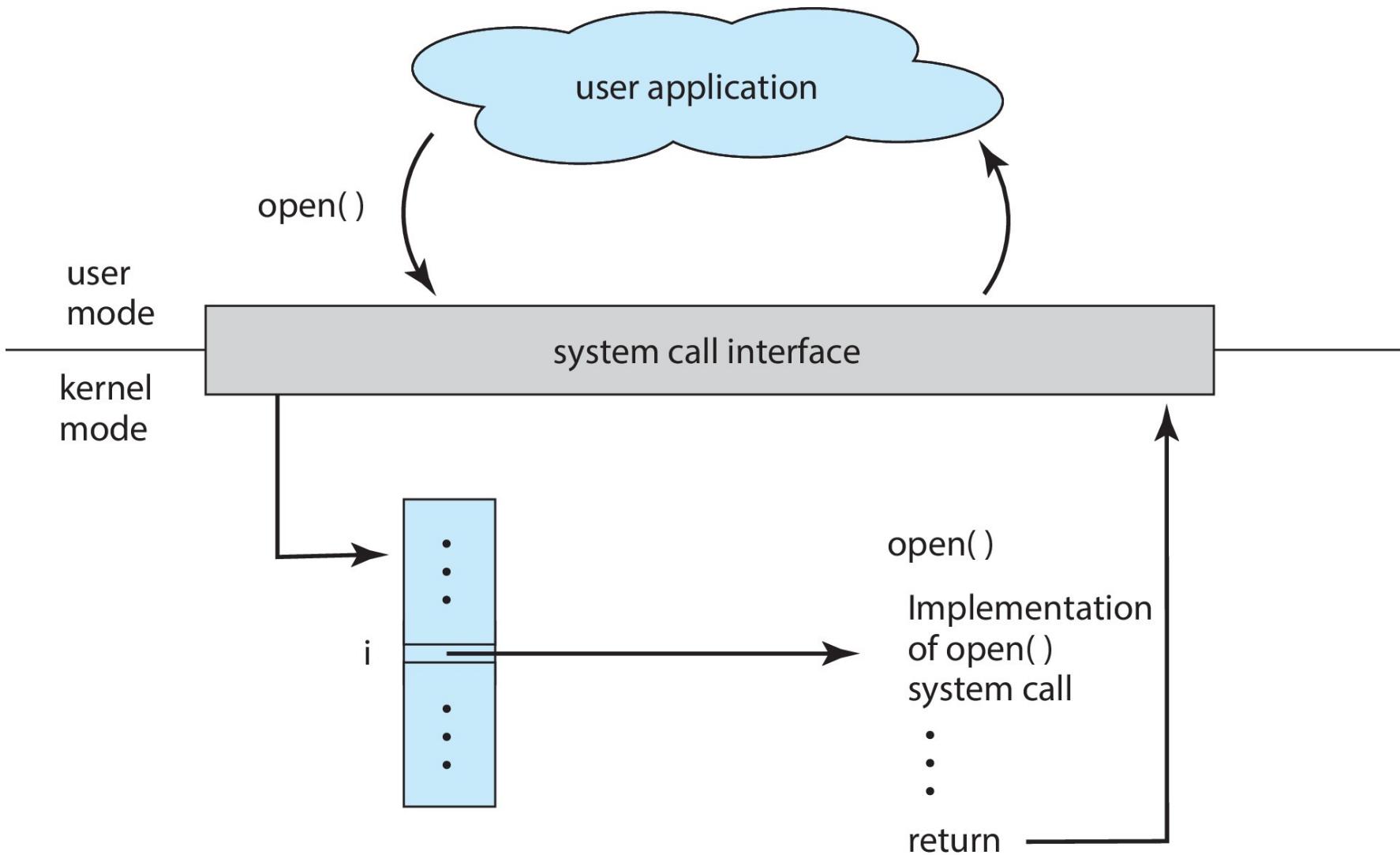
On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns -1.



System Call Implementation

- Typically, a number is 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



System calls in assembly programs (demo)

- Put the system call number in the EAX register.
- Store the arguments to the system call in the registers EBX, ECX,...
- Call the relevant interrupt (80h).
- The result is usually returned in the EAX register.

Let's see it in practice ☺

https://www.tutorialspoint.com/assembly_programming/assembly_system_calls.htm

System Call Parameter Passing

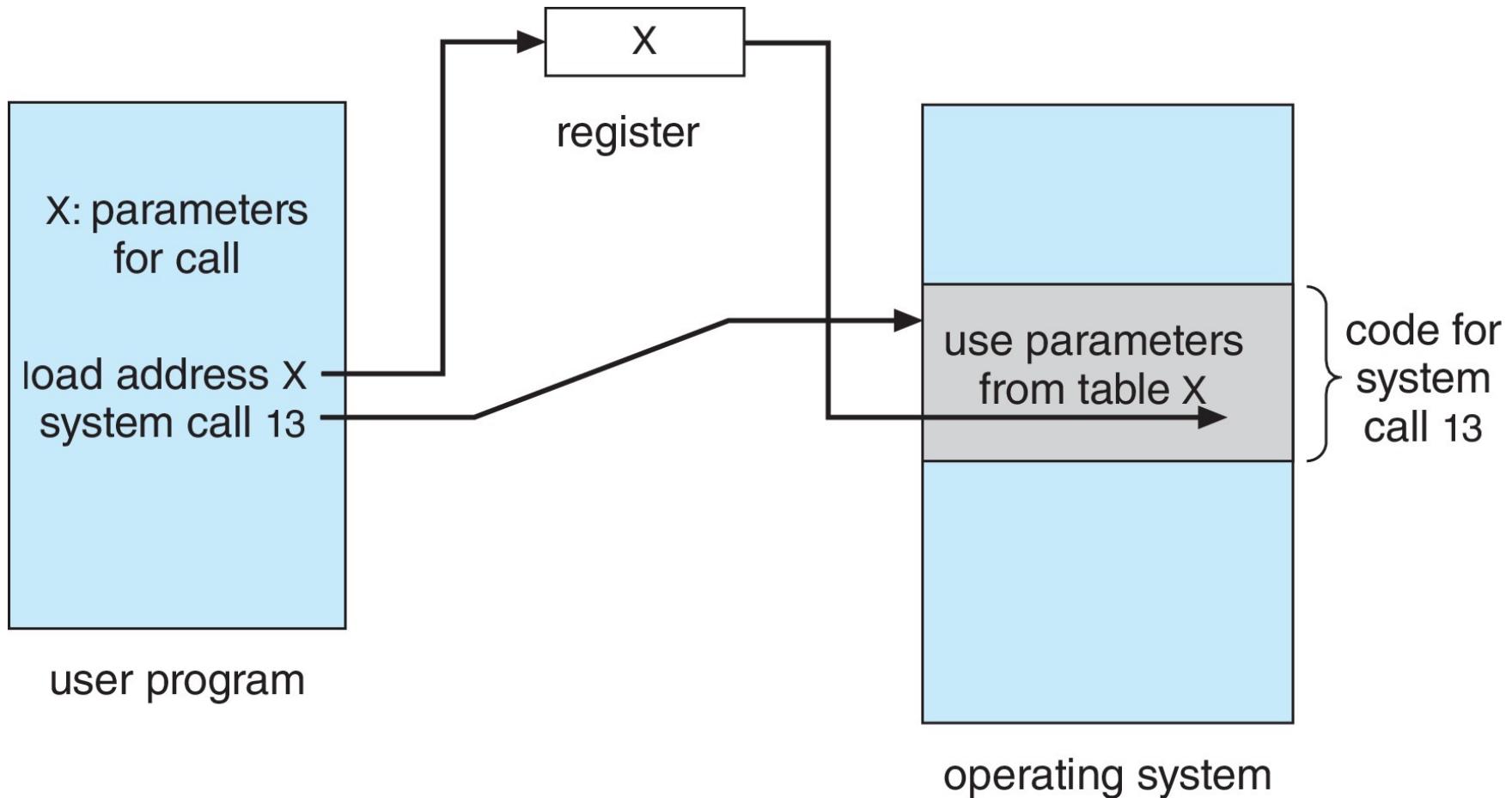
- **Parameter Passing**
 - Register
 - Register pointer to mem. Block
 - Stack (Push, Pop)
- Often, more information is required than simply identity of desired system call.
- Exact type and amount of information vary according to OS and call.



System Call Parameter Passing--Methods

- **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



Types of System Calls

- **Process control**
 - Create process, terminate process
 - ...
- **File management**
 - create file, delete file
 - ...
- **Device management**
 - request device, release device
 - ...
- **Please study the reference book for more details**



Types of System Calls (Cont.)

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



Why Applications are Operating System Specific

- Apps compiled on one system usually not executable on other OSs.
- Each OS provides its own unique system calls
 - Own file formats, etc.
- Apps can be multi-operating system
 - Written in interpreted language like Python, Ruby, and interpreter available on multiple OSs.
 - App written in language that includes a VM containing the running app (like Java).
 - Use standard language (like C), compile separately on each operating system to run on each.



Interrupts and System Calls

Seyyed Ahmad Javadi

sajavadi@aut.ac.ir

(Based on slides by Don Porter at UNC)

<https://www.cs.unc.edu/~porter/>



Questions from last session

```
section .data ;Data segment
    userMsg db 'Please enter a number: ' ;Ask the user to enter a number
    lenUserMsg equ $-userMsg ;The length of the message
    dispMsg db 'You have entered: '
    lenDispMsg equ $-dispMsg

section .bss ;Uninitialized data
    num resb 5 ;db: data byte

section .text ;Code Segment
    global _start ;equ: equivalent

_start: ;User prompt
    mov eax, 4 ;resb: reserve byte
    mov ebx, 1
    mov ecx, userMsg
    mov edx, lenUserMsg
    int 80h

https://www.tutorialspoint.com/assembly\_programming/assembly\_basic\_syntax.htm
```



Questions from last session

- Compiler vs interpreter
- int instruction
 - The instruction generates a software call to an interrupt handler.
 - Example
 - Trap to debugger: int \$3
 - Trap to interrupt 0xff: int \$0xff

<https://docs.oracle.com/cd/E19455-01/806-3773/instructionset-74/index.html>



Background: Control Flow

```
x = 2, y = true      void printf(va_args)
if (y) {
    2 /= x;          //
    printf(x);        }
} //...
```

Background: Control Flow

pc

```
x = 2, y = true      void printf(va_args)
if (y) {
    2 /= x;          // ...
    printf(x);        }
} // ...
```



Background: Control Flow

pc

```
x = 2, y = true      void printf(va_args)
if (y) {
    2 /= x;           {
                      //...
    printf(x);        }
} //...
```



Background: Control Flow

```
x = 2, y = true      void printf(va_args)
if (y) {                {
    2 /= x;          // ...
    printf(x);        }
} // ...
```

pc



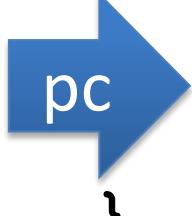
Background: Control Flow

```
x = 2, y = tr pc → void printf(va_args)
if (y) {
    2 /= x;           // ...
    printf(x);        }
} // ...
```



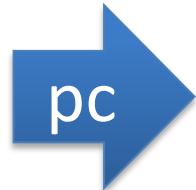
Background: Control Flow

```
x = 2, y = true      void printf(va_args)
if (y) {
    2 /= x;
    printf(x);
} //...
```



Background: Control Flow

```
x = 2, y = true      void printf(va_args)
if (y) {                {
    2 /= x;          //...
    printf(x);        }
} //...
```



Regular control flow: branches and calls
(logically follows source code)

Background: Control Flow

```
x = 0, y = true  
if (y) {  
    2 /= x;  
    printf(x);  
} //...
```

Background: Control Flow

pc → **x = 0, y = true**
if (y) {
 2 /= x;
 printf(x);
} //...

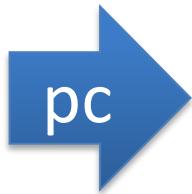
Background: Control Flow

```
x = 0, y = true
pc if (y) {
    2 /= x;
    printf(x);
} //...
```

Background: Control Flow

```
x = 0, y = true  
if (y) {  
    2 /= x;  
    printf(x);  
} //...
```

Divide by zero!
Program can't make
progress!



Background: Control Flow

```
x = 0, y = true  
if (y) {  
    2 /= x;  
    printf(x);  
} //...
```

pc

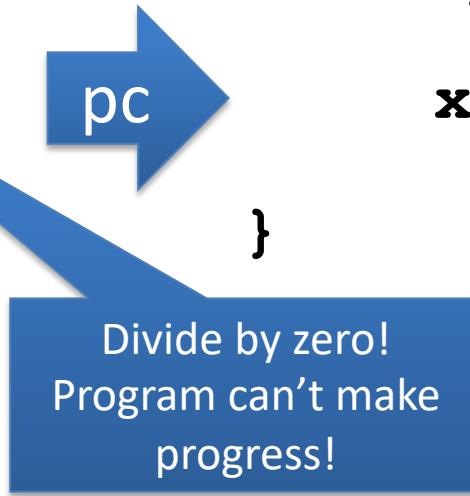
```
void  
handle_divzero() {  
    x = 2;  
}
```

Divide by zero!
Program can't make
progress!

Background: Control Flow

```
x = 0, y = true  
if (y) {  
    2 /= x;  
    printf(x);  
} //...
```

```
void  
handle_divzero() {  
    x = 2;  
}
```



Background: Control Flow

```
x = 0, y = true  
if (y) {  
    2 /= x;  
    printf(x);  
} // ...
```

pc

```
void  
handle_divzero() {  
    x = 2;  
}
```

Divide by zero!
Program can't make
progress!



Background: Control Flow

```
x = 0, y = true  
if (y) {  
    2 /= x;  
    printf(x);  
}  
pc //...
```

```
void  
handle_divzero() {  
    x = 2;  
}  
}
```

Divide by zero!
Program can't make
progress!

Irregular control flow: exceptions, system calls, etc.

Lecture goal

- Understand the hardware tools available for **irregular control flow**.
 - I.e., things other than a branch in a running program
- Building blocks for context switching, device management, etc.

Two types of interrupts

- **Synchronous:** will happen every time an instruction executes (with a given program state)
 - Divide by zero
 - System call
 - Bad pointer dereference
- **Asynchronous:** caused by an external event
 - Usually device I/O
 - Timer ticks (well, clocks can be considered a device)

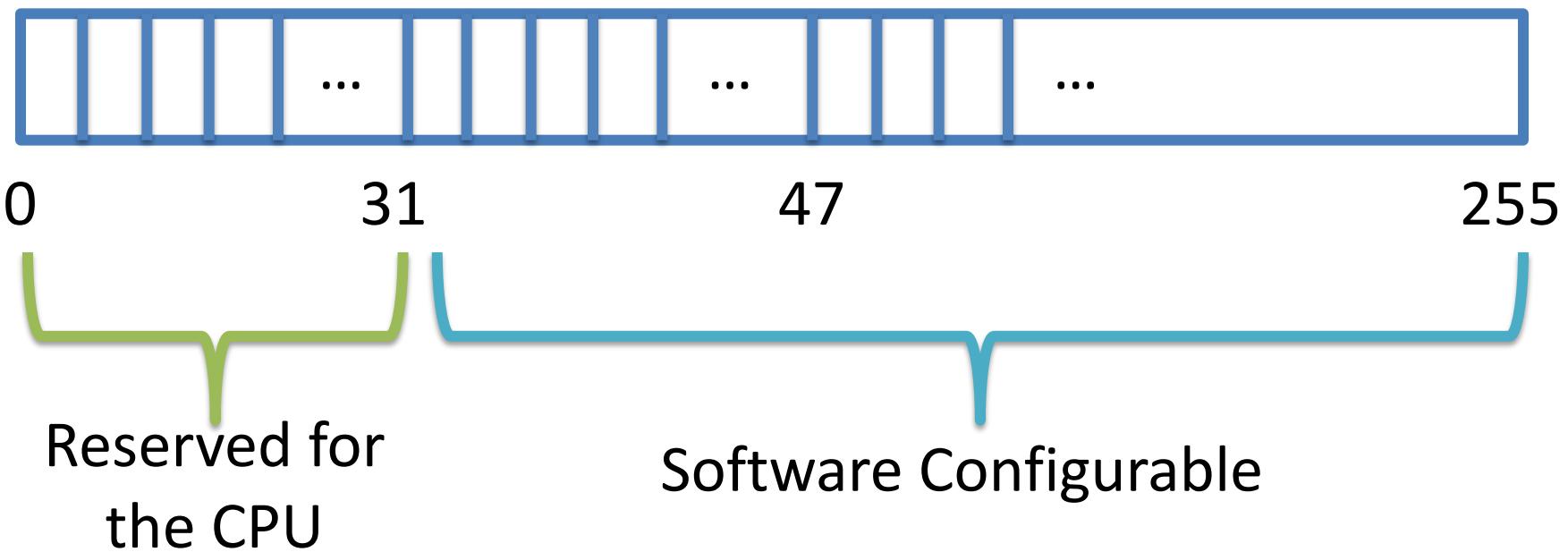
Intel nomenclature

- **Interrupt** – only refers to asynchronous interrupts
- **Exception** – synchronous control transfer
- Note: from the programmer's perspective, these are handled with the same abstractions

Interrupt overview

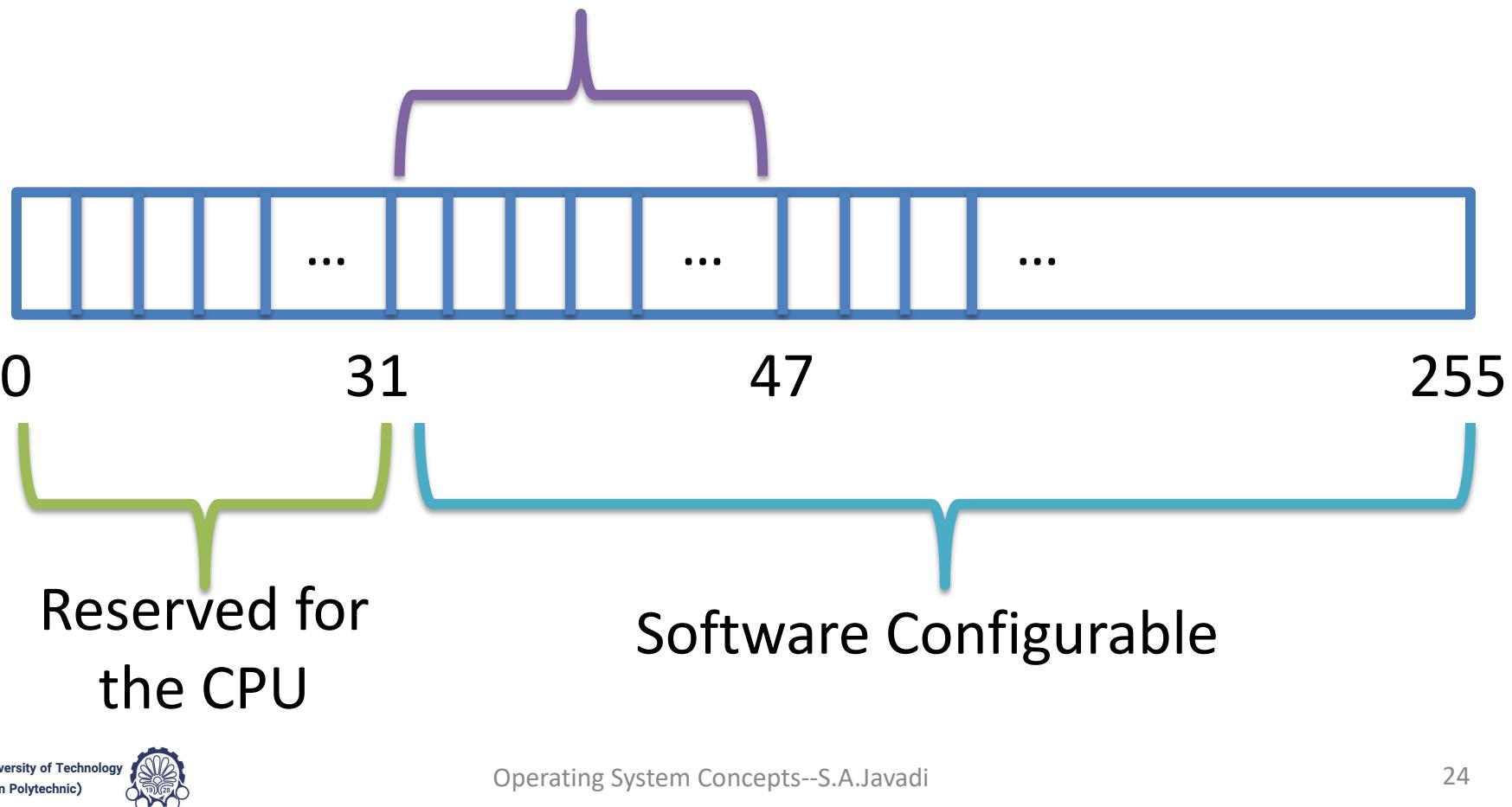
- Each interrupt or exception includes a number indicating its type.
- E.g., 14 is a page fault, 3 is a debug breakpoint.
- This number is the index into an interrupt table.

x86 interrupt table

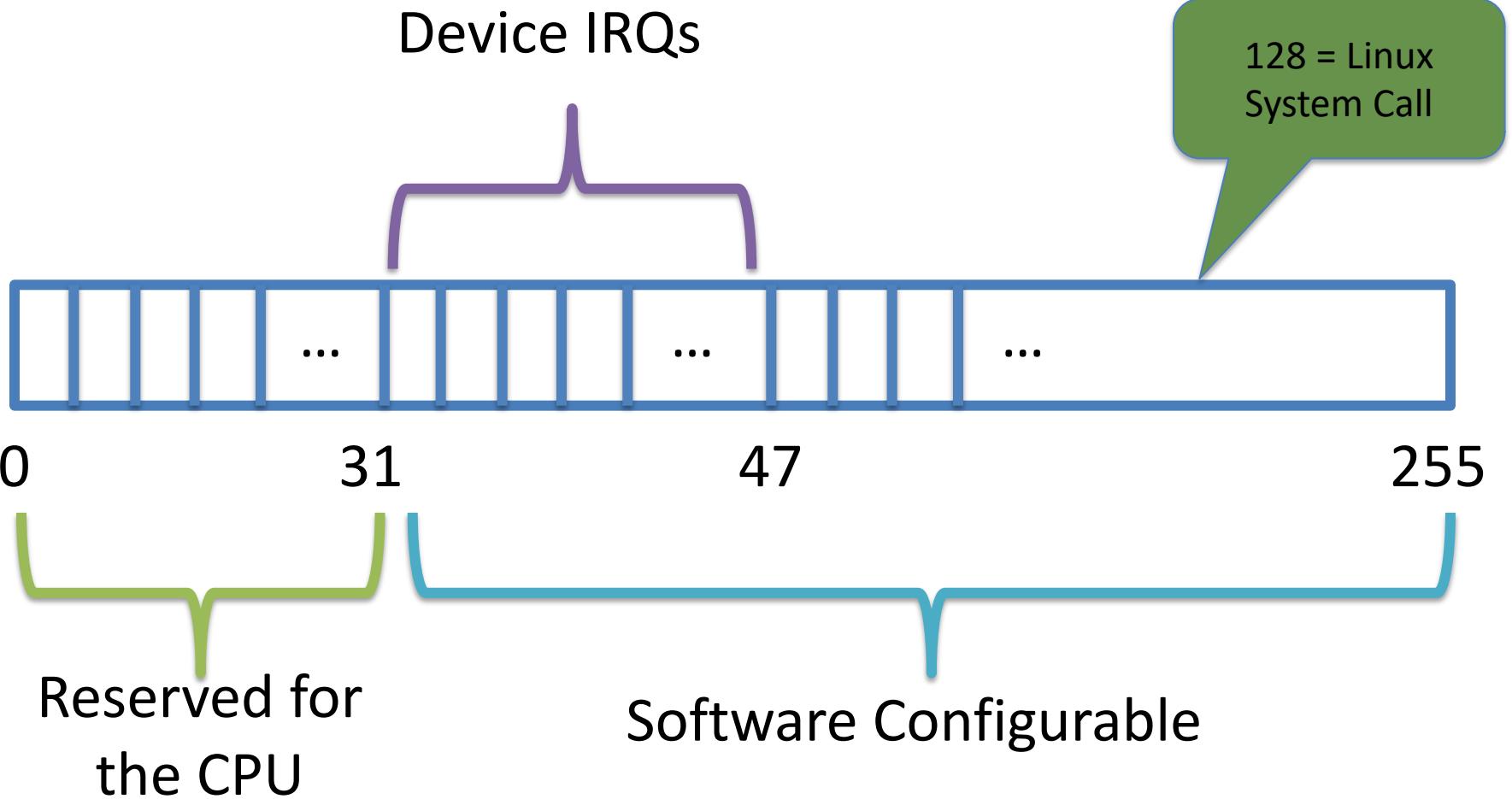


x86 interrupt table

Device IRQs (Interrupt ReQuests)



x86 interrupt table



x86 interrupt overview

- Each type of interrupt is assigned an index from 0—255.
- 0—31 are for processor interrupts; generally fixed by Intel
 - E.g., 14 is always for page faults
- 32—255 are software configured
 - 0x80 issues system call in Linux (more on this later)

Software interrupts

- The `int <num>` instruction allows software to raise an interrupt
 - 0x80 is just a Linux convention.
- There are a lot of spare indices
 - You could have multiple system call tables for different purposes or types of processes!
 - Windows does: one for the kernel and one for win32k

What happens (generally):

- Control jumps to the kernel
 - At a prescribed address (the interrupt handler)
- The register state of the program is dumped on the kernel's stack
- Kernel code runs and handles the interrupt
- When handler completes, resume program (see `iret` instr.)

System call “interrupt”

- Originally, system calls issued using `int` instruction
- Dispatch routine was just an interrupt handler
- Like interrupts, system calls are arranged in a table
 - See `arch/x86/kernel/syscall_table*.S` in Linux source
- Program selects the one it wants by placing index in `eax` register
 - Arguments go in the other registers by calling convention
 - Return value goes in `eax`



Operating Systems

Processes-Part1

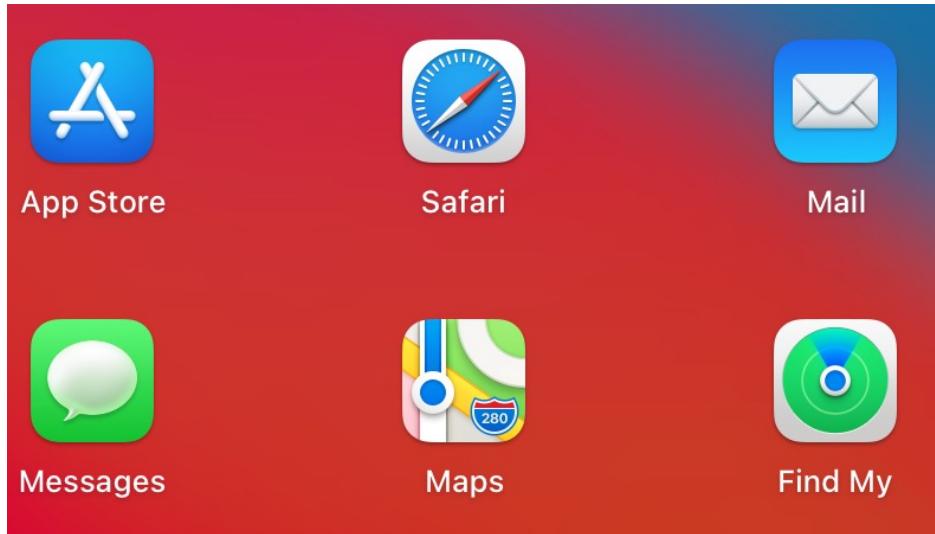
Seyyed Ahmad Javadi

sajavadi@aut.ac.ir

Fall 2021

Process Concept

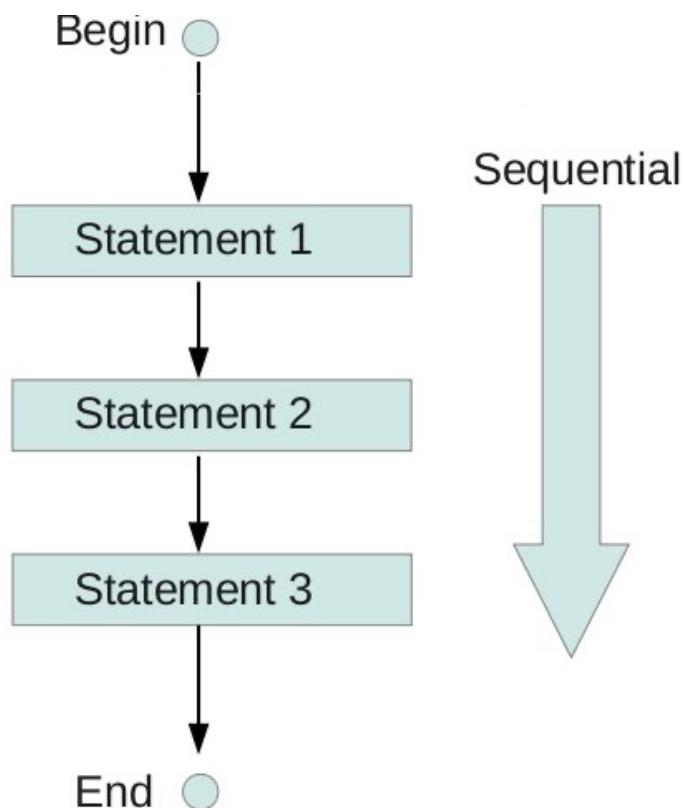
- An OS executes a variety of programs that run as a ***process***.



Process, a program in execution

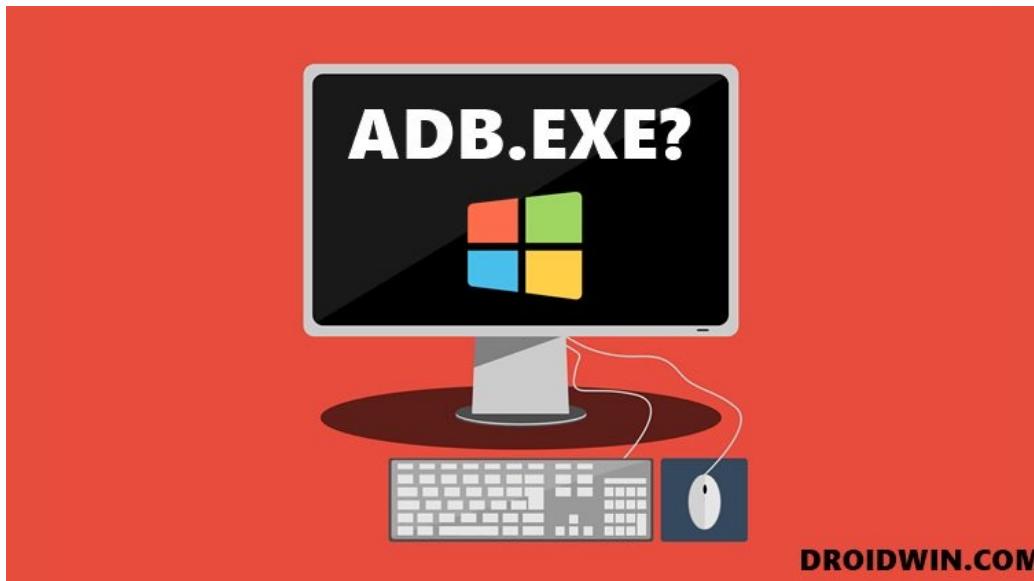
Process Concept (cont.)

- Process execution must progress in *sequential fashion*.
 - *No parallel execution* of instructions of a single process.



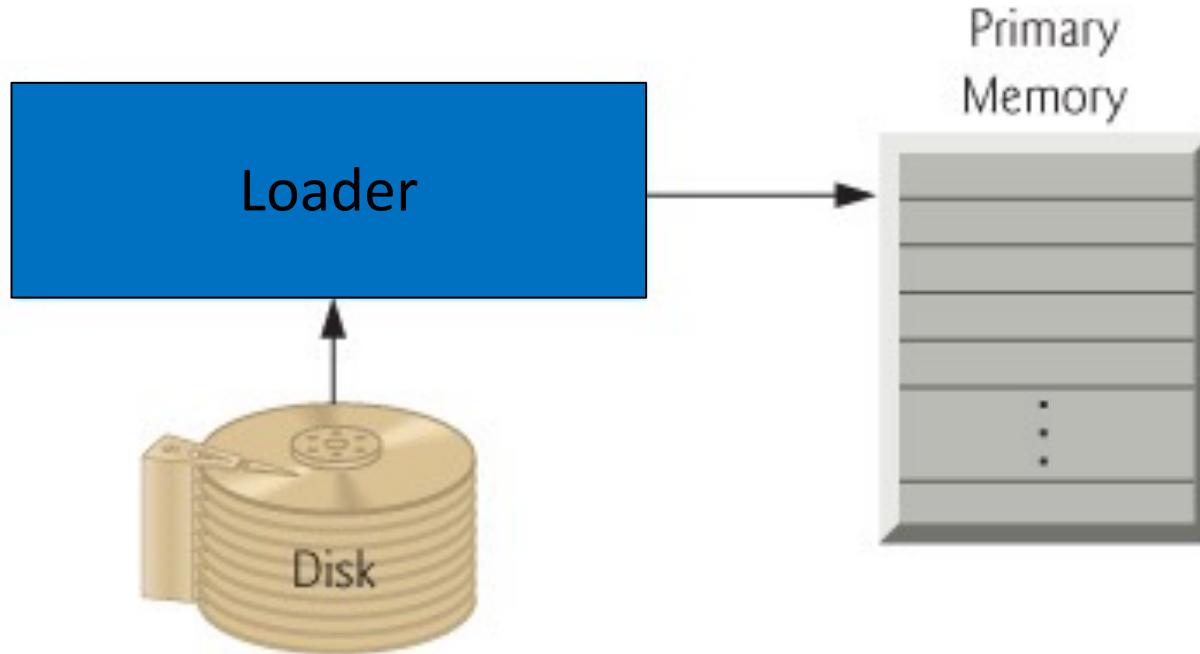
Process versus Program

- Program is **passive** entity stored on disk (*executable file*).



Process versus Program (cont.)

- **Process is active.**
 - **Program becomes process** when an executable file is loaded into memory.



Process versus Program (cont.)

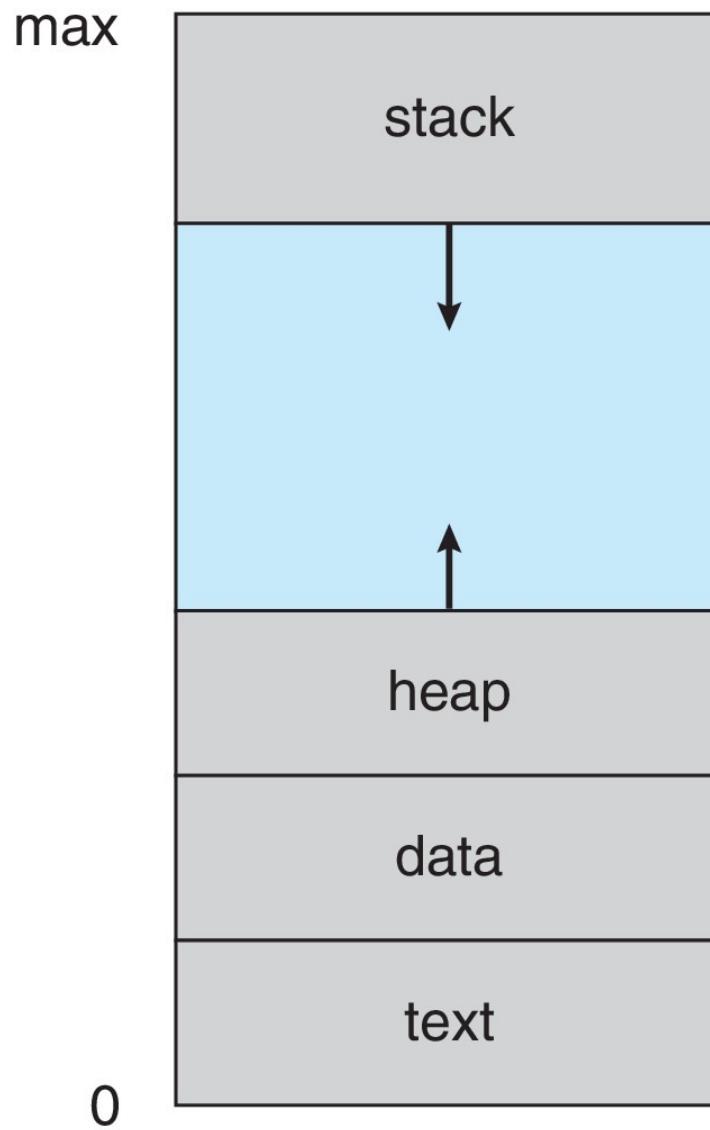
- Execution of program started via:
 - GUI mouse clicks
 - Command line entry of its name
 - Etc.
- One program can be several processes
 - Consider multiple users executing the same program.



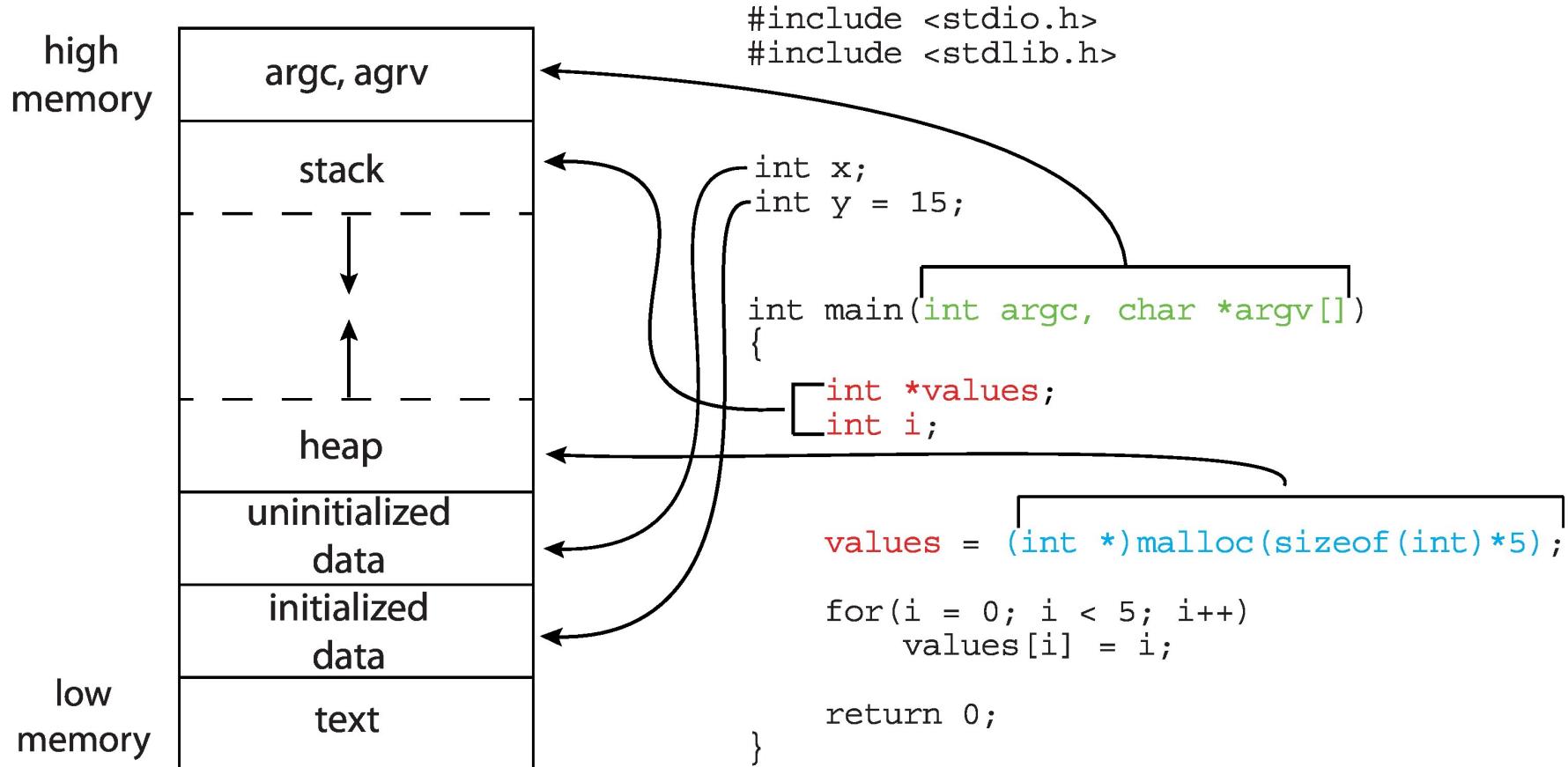
Multiple Parts of Process

- 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 in Memory



Memory Layout of a C Program

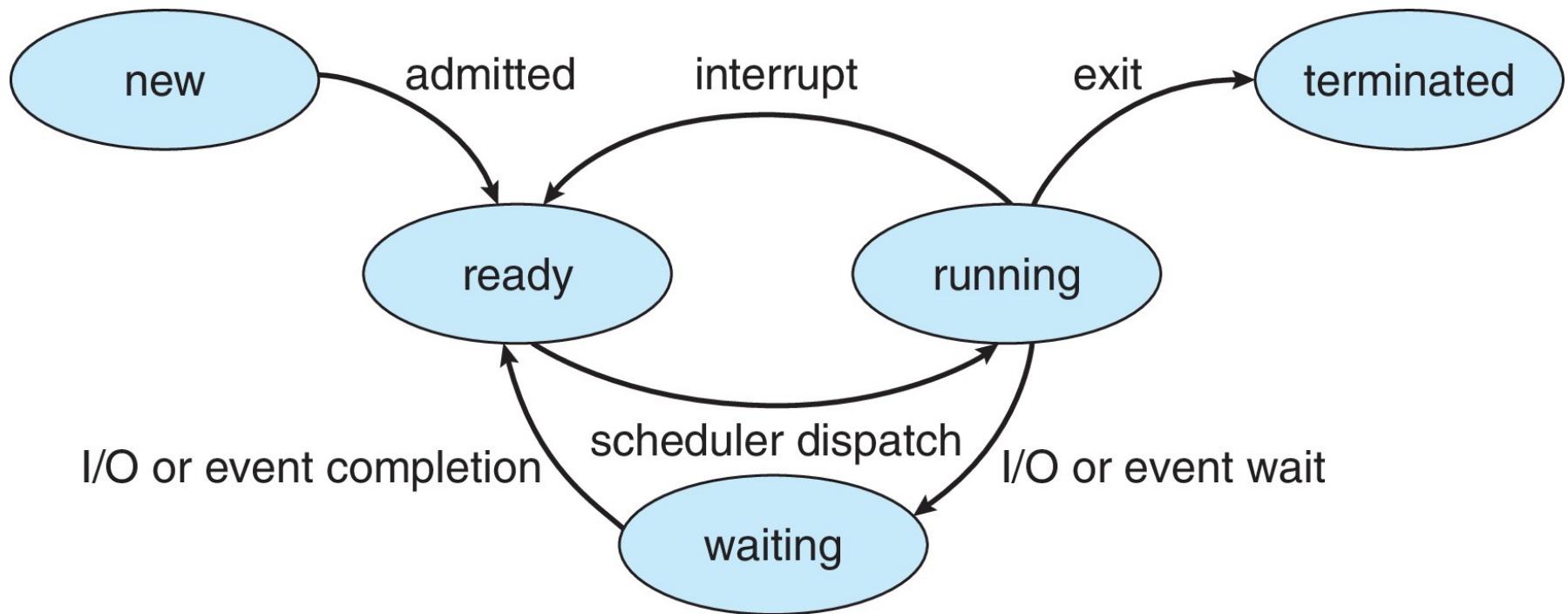


Process State

- 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

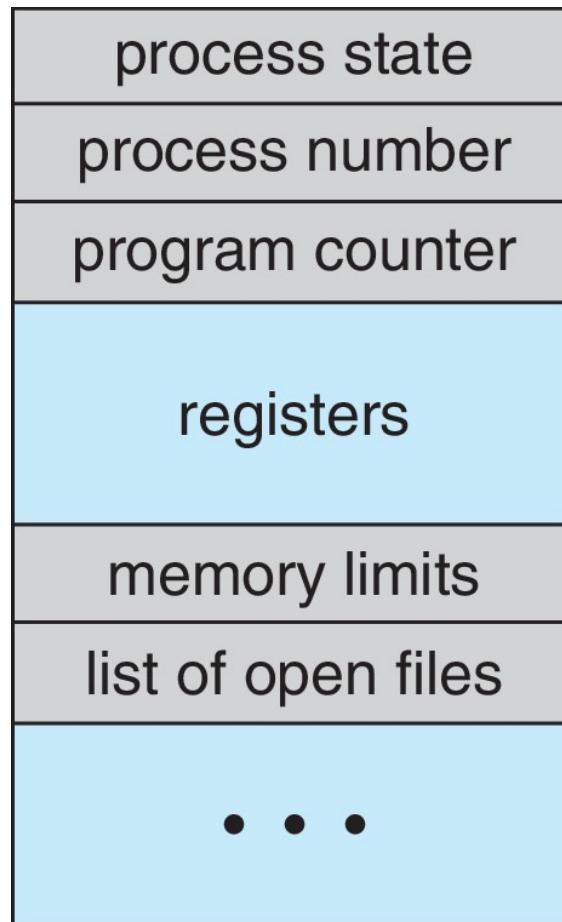


Process Control Block (PCB)

Information associated with each process

- **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:** allocated memory
- **Accounting information:** CPU used, clock time elapsed since start, etc
- **I/O status information:** allocated I/O devices, list of open files.

Process Control Block (PCB) (cont.)



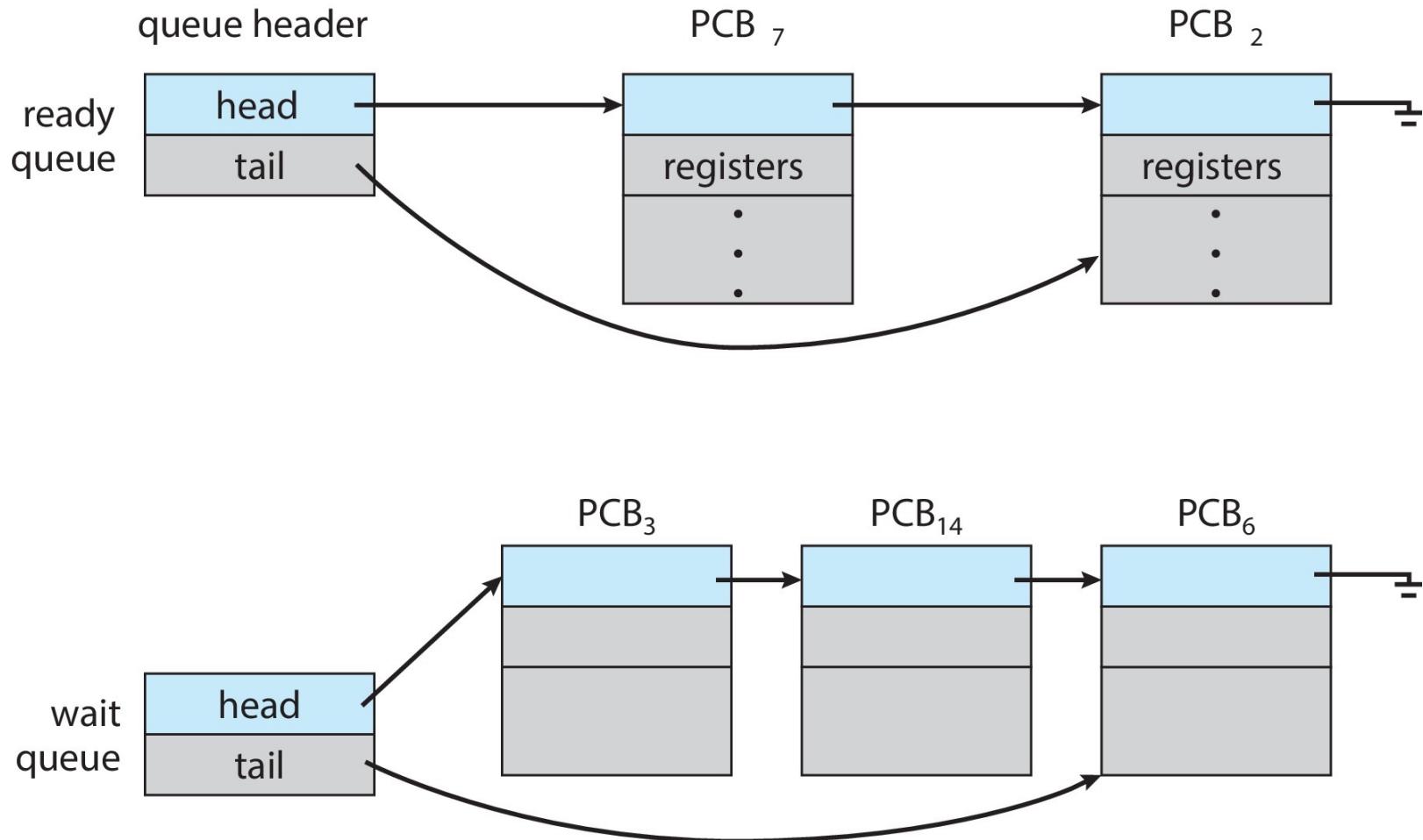
Threads

- So far, process has *a single thread of execution.*
- Consider having *multiple program counters* per process.
 - Multiple locations can execute at once
 - ▶ Multiple threads of control -> **threads**
- Must then have *storage for thread details*
 - Multiple program counters in PCB.
- Explore in detail in Chapter 4.

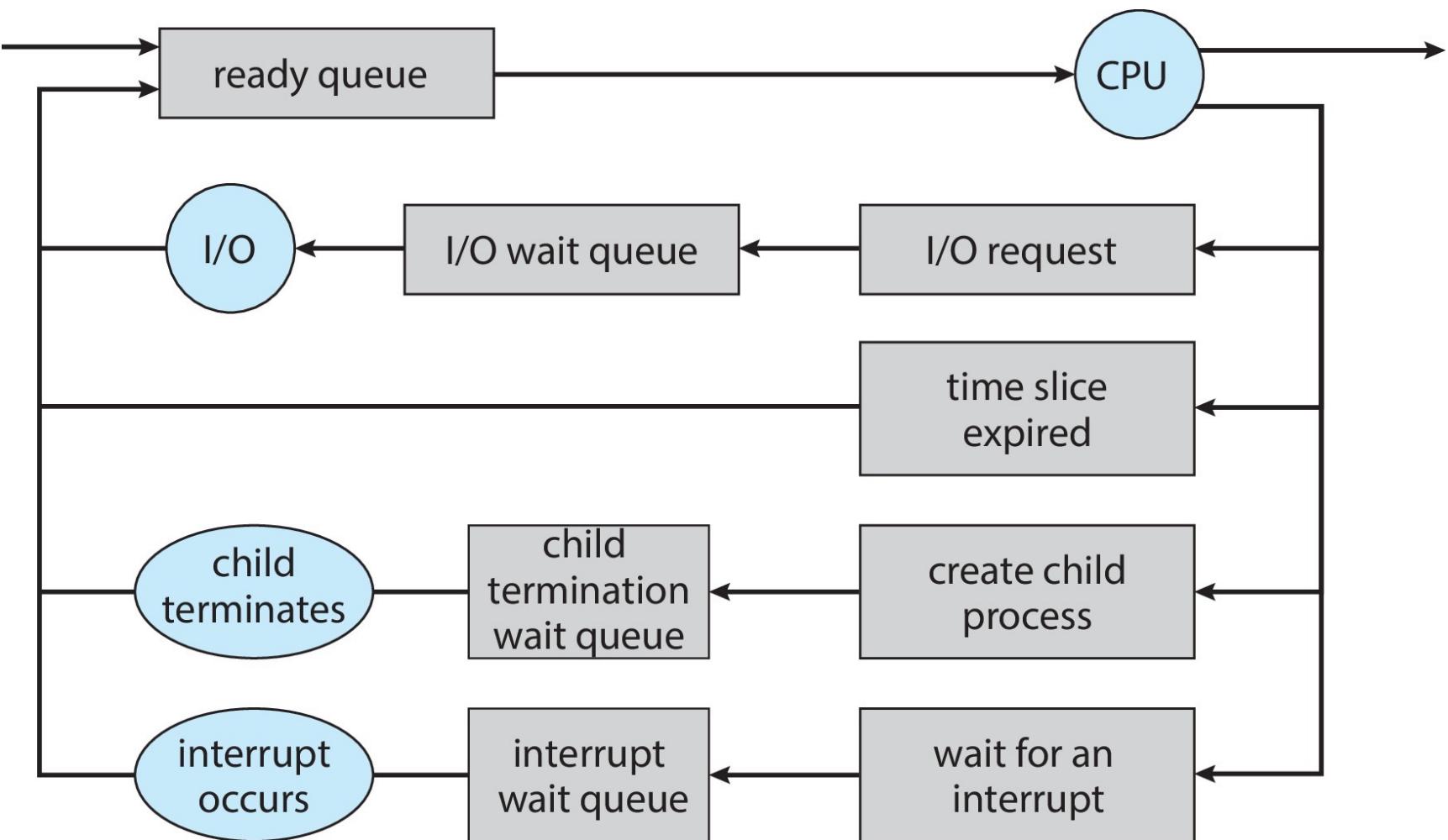
Process Scheduling

- **Process scheduler** selects among available processes for next execution on CPU core.
- Goal: Maximize CPU use, quickly switch processes onto CPU core.
- Maintains **scheduling queues** of processes
 - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute.
 - **Wait queues** – set of processes waiting for an event (i.e., I/O)
 - Processes migrate among the various queues.

Ready and Wait Queues



Representation of Process Scheduling





Operating Systems

Processes-Part2

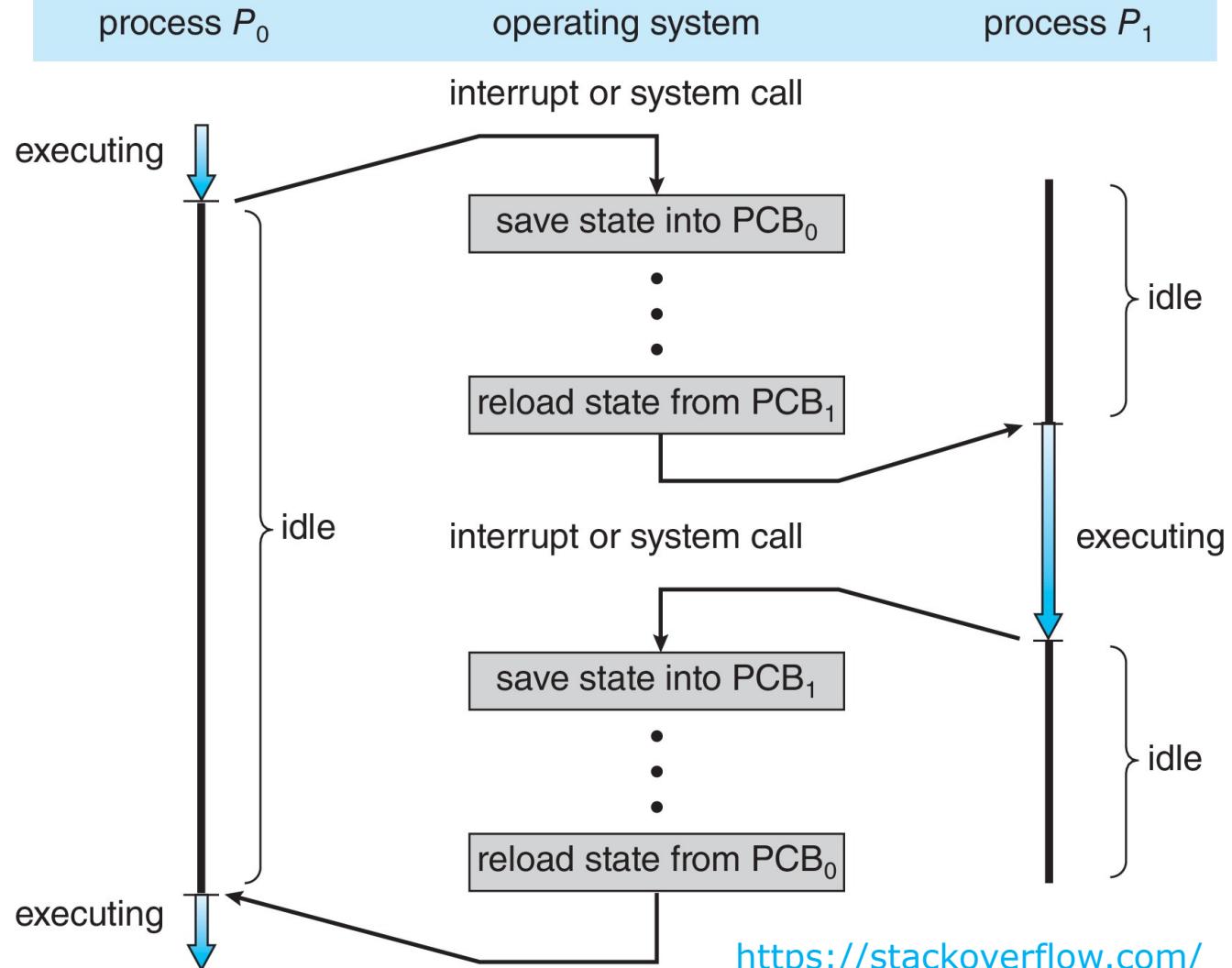
Seyyed Ahmad Javadi

sajavadi@aut.ac.ir

Fall 2021

CPU Switch From Process to Process

A context switch occurs when the CPU switches from one process to another.



<https://stackoverflow.com/questions/9238326/system-call-and-context-switch>



Context Switch

- 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 *pure overhead*
 - The system does no useful work while switching.

The more complex
the OS and the PCB



the longer
the context switch

Context Switch (cont.)

- 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.
- 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

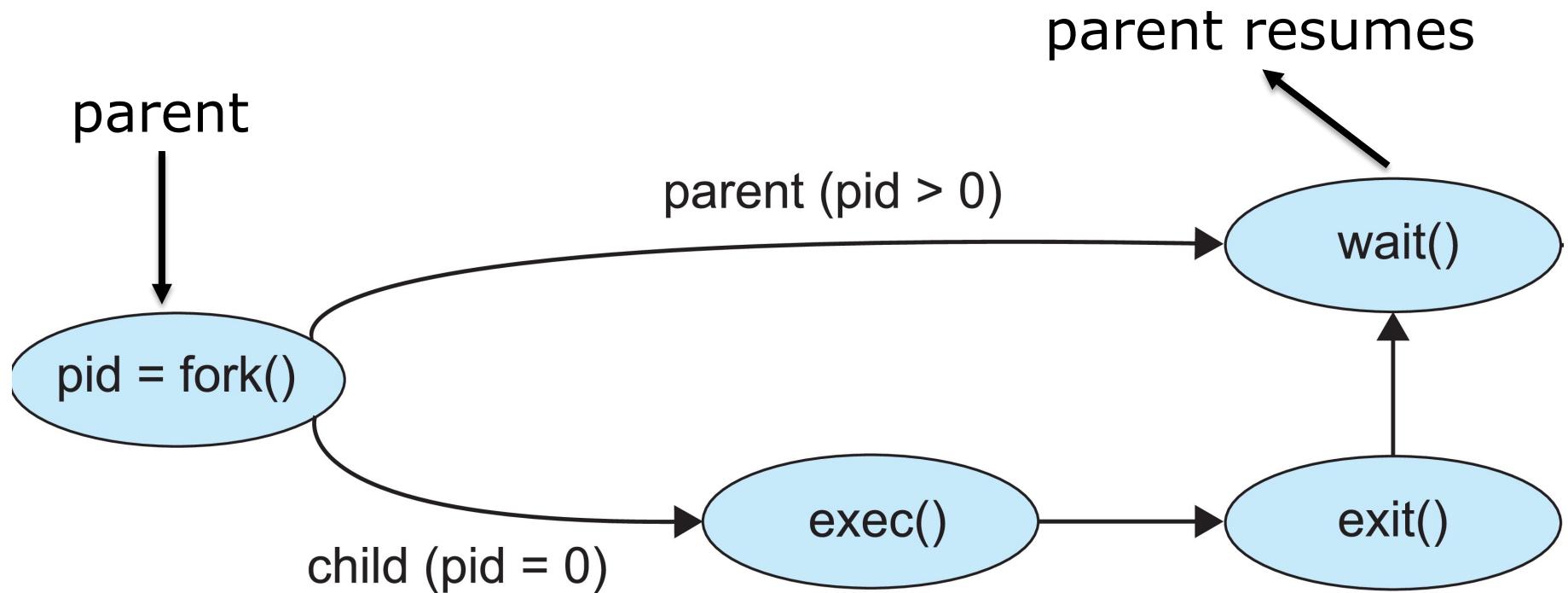
Process Creation (Cont.)

- **Address space**
 - Child duplicate of parent
 - Child has a program loaded into it

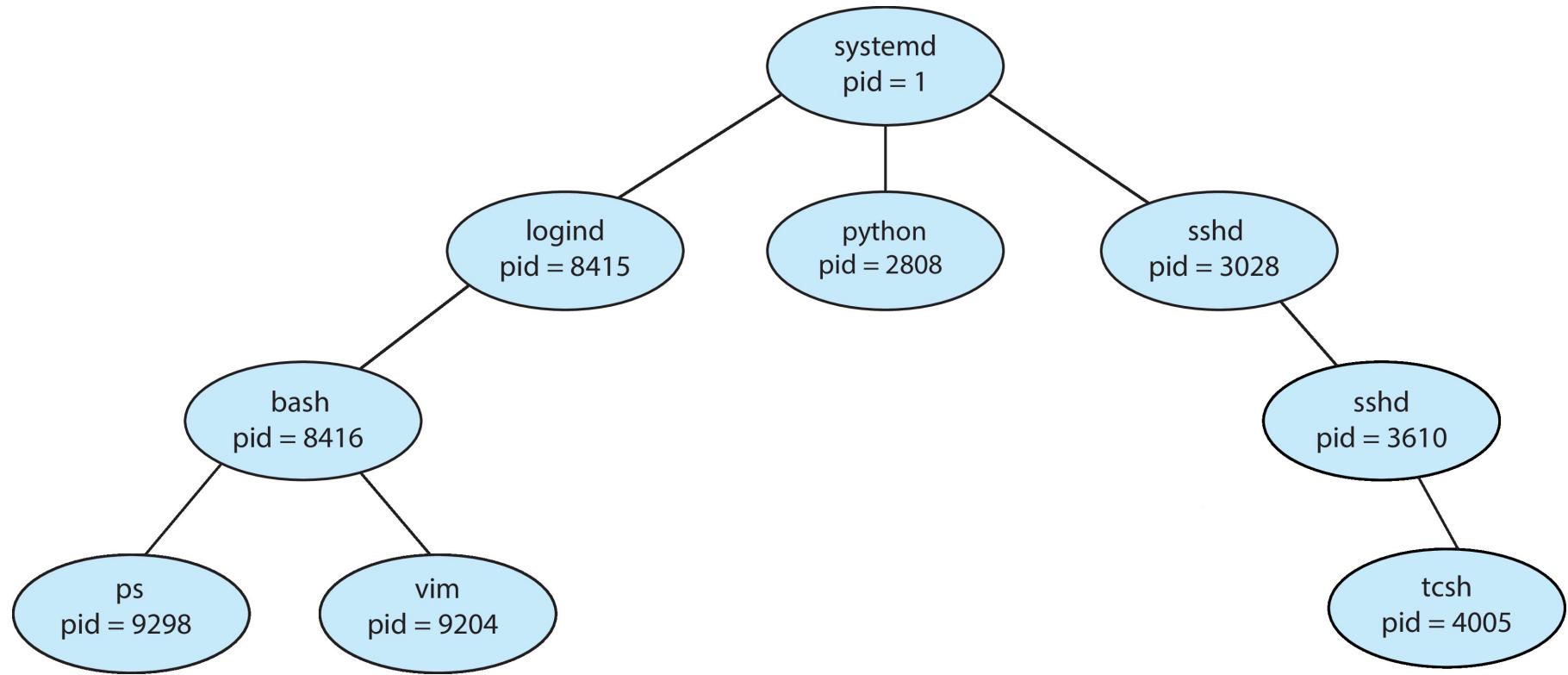
- **UNIX examples**
 - **fork()** system call creates new process.
 - **exec()** system call used after a **fork()** to replace the process' memory space with a new program.
 - Parent process calls **wait()** waiting for the child to terminate.



Process Creation (Cont.)



A Tree of Processes in Linux



C Program Forking Separate Process

```
#include <sys/types.h> <stdio.h> <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;
}
```



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 (Cont.)

- Some OSs 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.



Process Termination (Cont.)

- 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);

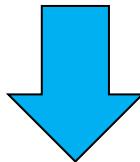
- If no parent waiting (did not invoke wait()) process is a **zombie**.
- If parent terminated without invoking wait(), process is an **orphan**.



Multiprocess Architecture – Browser (Cont.)

- Many web browsers ran as single process (some still do)

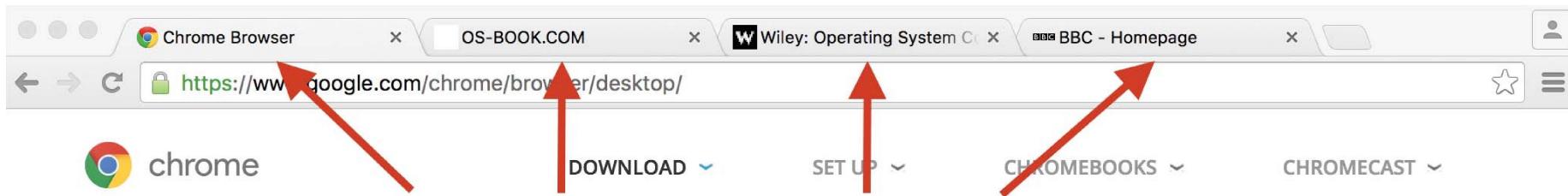
If one web site causes trouble



Entire browser can hang or crash

Multiprocess Architecture – Chrome Browser (Cont.)

- Google Chrome is multiprocess with 3 different types of processes:
 - **Browser** process manages user interface, disk and network I/O.
 - **Renderer** process renders web pages, deals with HTML, Javascript.
 - ▶ A new renderer created for each website opened
 - ▶ Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits.
 - **Plug-in** process for each type of plug-in.



Each tab represents a separate process.



Operating Systems

Processes-Part3

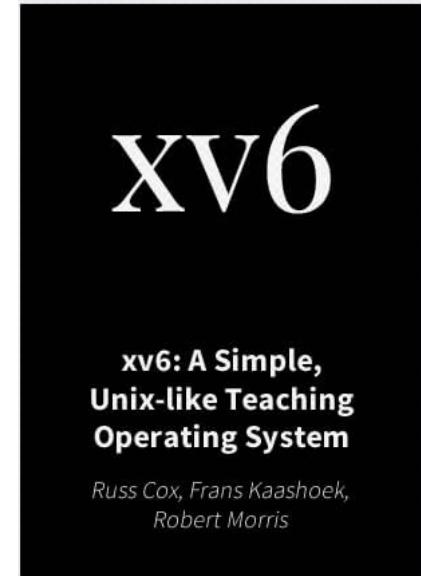
Seyyed Ahmad Javadi

sajavadi@aut.ac.ir

Fall 2021

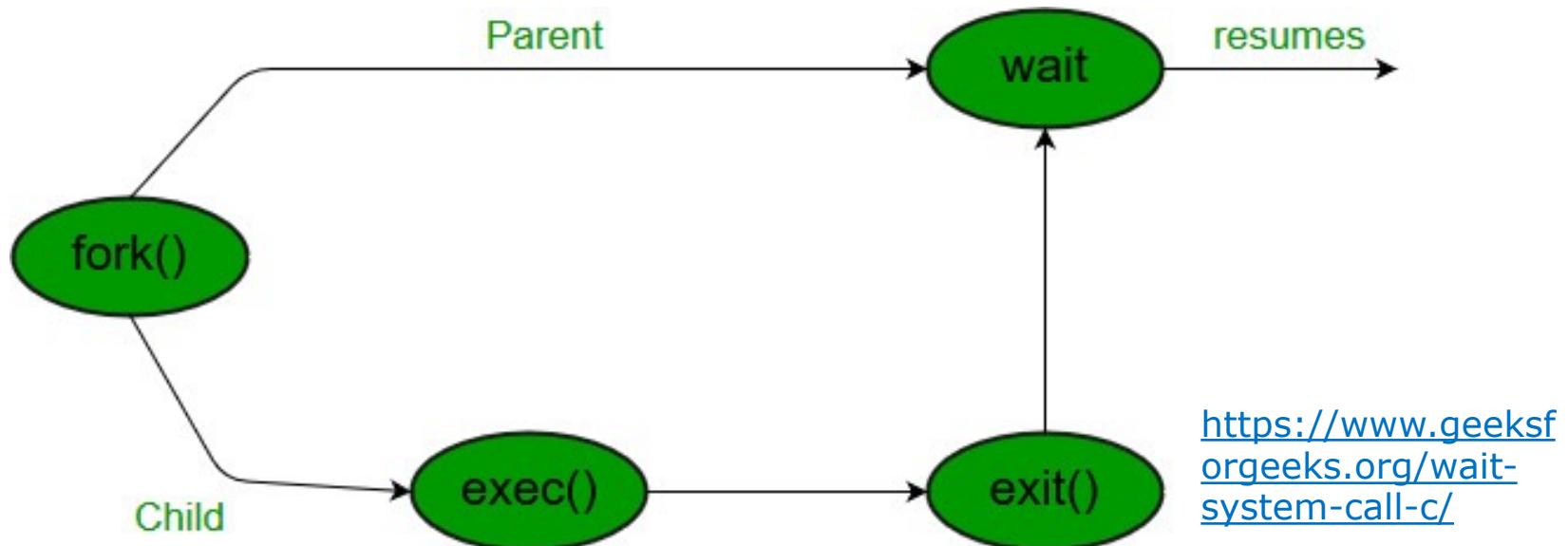
Course logistics

- You have **less than a week** to complete
 - Phase 1 of the project
 - Your first homework
- We will have an extra session on fork
 - The session is handled by TAs
 - It covers both theoretical aspects and how use fork in practice
 - Most probably at coming Thursday



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.



Process Termination (cont.)

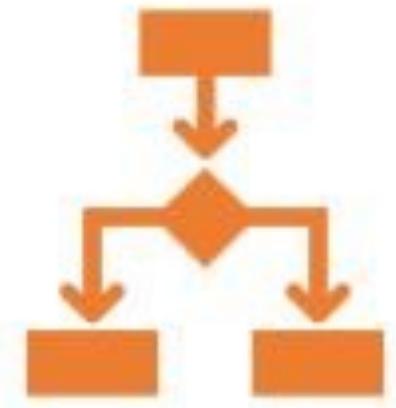
- Parent may terminate the execution of children processes using the **abort()** system call.

- 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 (cont.)

- Some OSs 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.



Process Termination (cont.)

- 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.

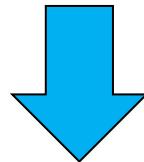
pid = wait(&status);

- If no parent waiting (did not invoke `wait()`), process is a **zombie**.
- If parent terminated without invoking `wait()`, process is an **orphan**.

Multiprocess Architecture – Browser

- Many web browsers ran as single process (some still do)

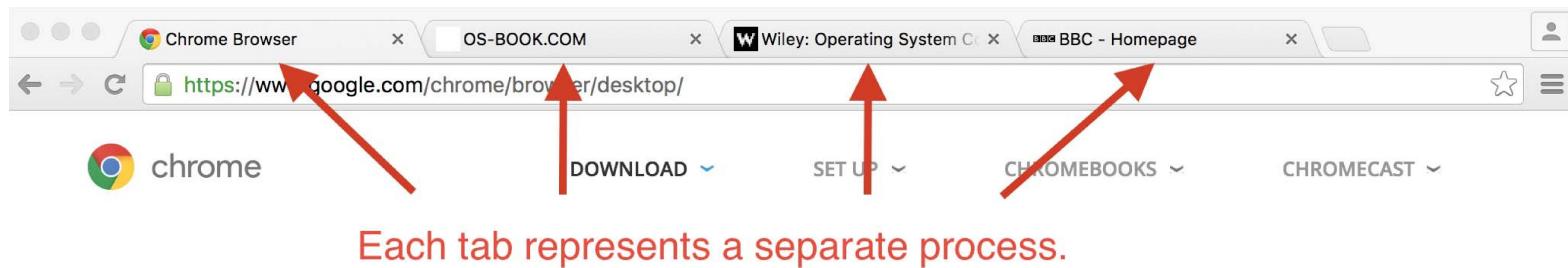
If one web site causes trouble



Entire browser can hang or crash

Multiprocess Architecture – Chrome Browser (cont.)

- Google Chrome is multiprocess with 3 different types of processes:
 - **Browser** process manages user interface, disk and network I/O.
 - **Renderer** process renders web pages, deals with HTML, Javascript.
 - ▶ A new renderer created for each website opened
 - ▶ Runs in **sandbox** restricting disk and network I/O (why?)
 - **Plug-in** process for each type of plug-in.



Inter-Process Communication

- Processes within a system may be *independent* or *cooperating*
- Cooperating process can affect or be affected by other processes, including *sharing data*.
- Reasons for cooperating processes:
 - Information sharing
 - Computation speedup
 - Modularity
 - Convenience

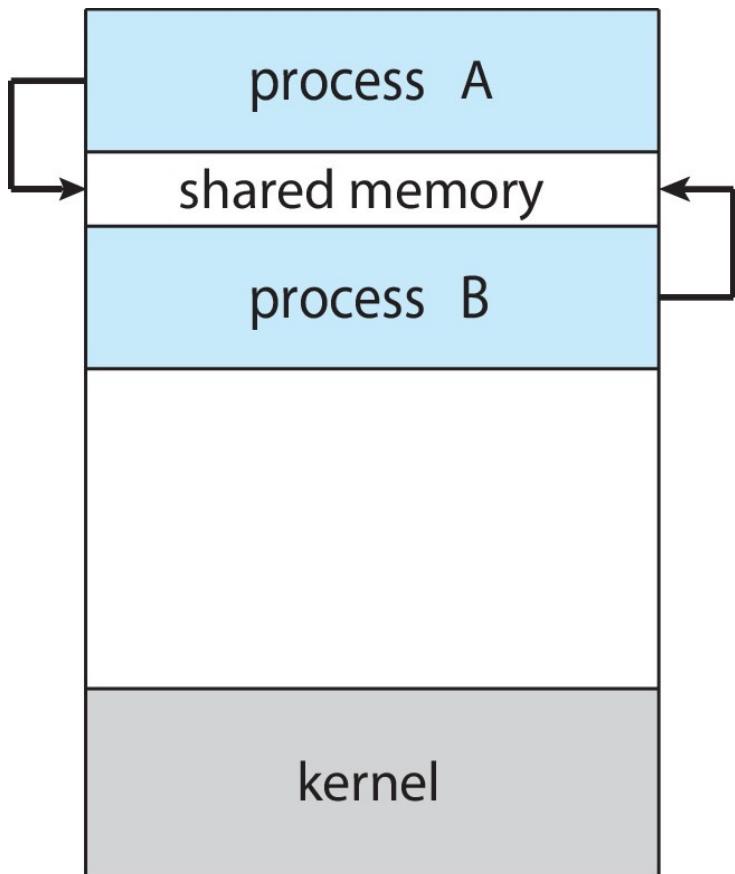


Inter-Process Communication (Cont.)

- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
 - **Shared memory**
 - **Message passing**
 - ▶ We do not cover this.

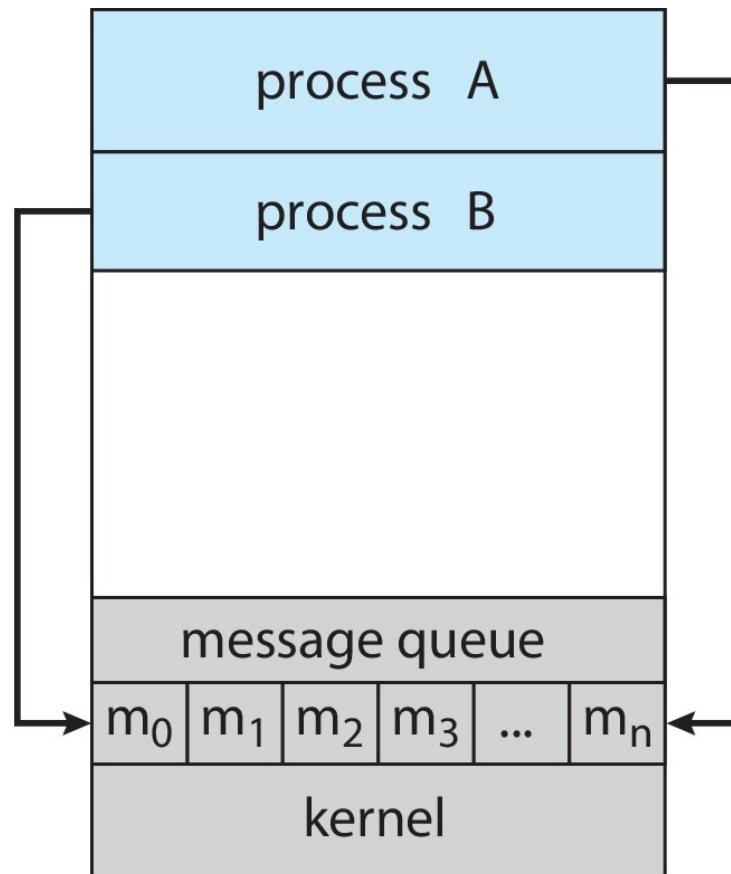
Communications Models

(a) Shared memory.



(a)

(b) Message passing.



(b)

Producer-Consumer Problem

- Paradigm for cooperating processes:
 - **Producer** process produces information that is consumed by a **consumer** process.

- Two variations:
 - **Unbounded-buffer** places no practical limit on the size of the buffer:
 - ▶ Producer never waits
 - ▶ Consumer waits if there is no buffer to consumer
 - **Bounded-buffer** assumes that there is a fixed buffer size
 - ▶ Producer must wait if all buffers are full
 - ▶ Consumer waits if there is no buffer to consume



IPC – 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.
- Synchronization is discussed in great details in Chapters 6 & 7.

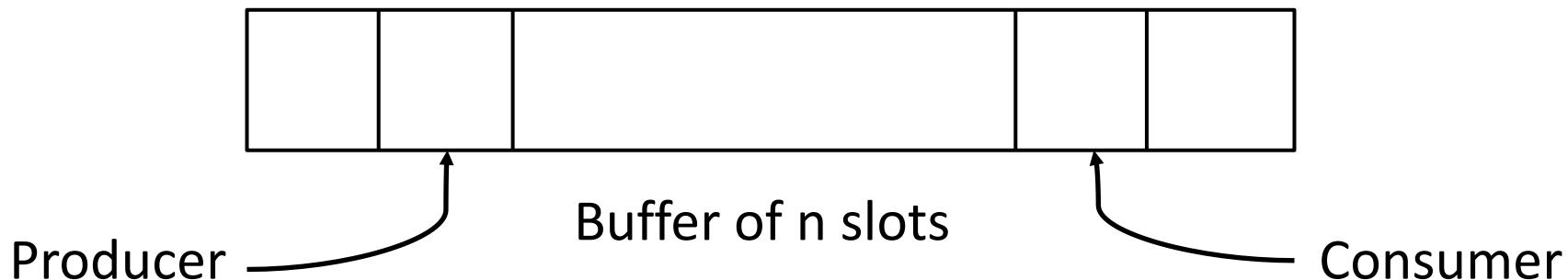


Bounded-Buffer – Shared-Memory Solution

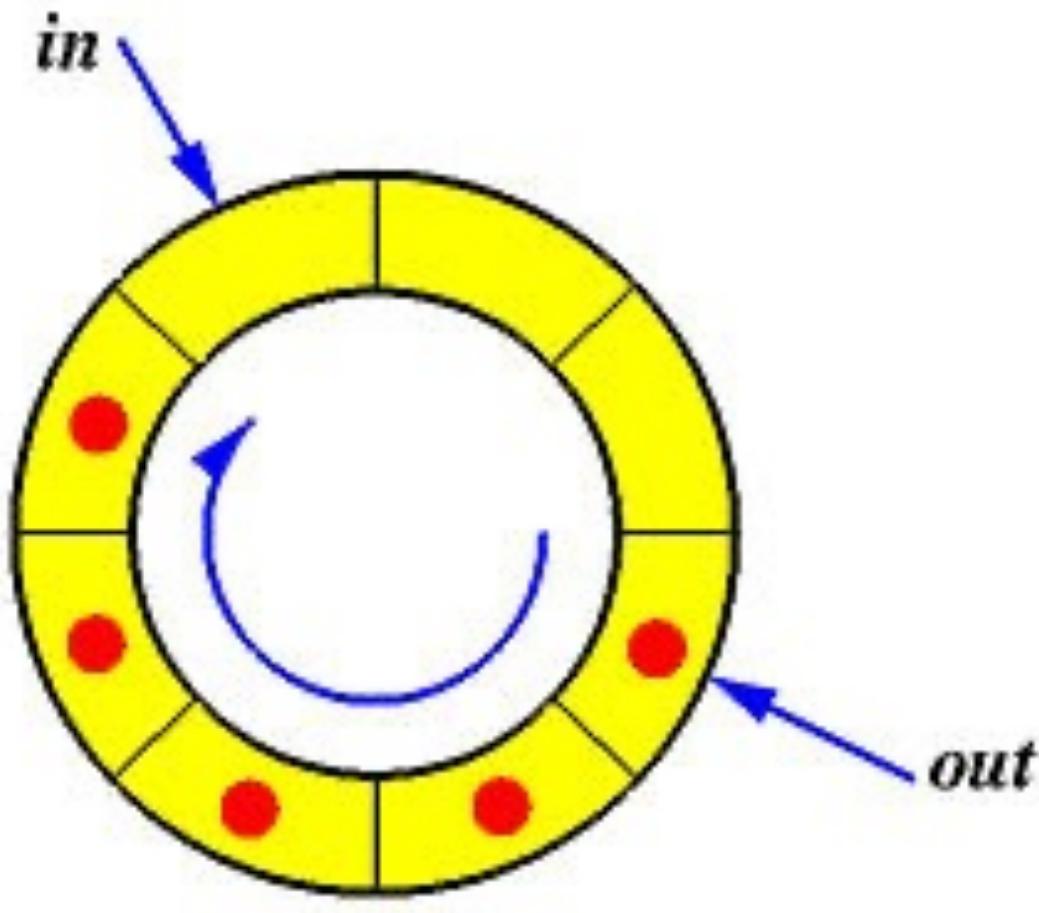
- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct but can only use **BUFFER_SIZE - 1** elements.



Circular Bounded-Buffer



Source: <https://pages.mtu.edu/~shene/NSF-3/e-Book/SEMA/TM-example-buffer.html>

Producer Process – Shared Memory

```
item next_produced;

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



Consumer Process – Shared Memory

```
item next_consumed;

while (true) {
    while (in == out)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next consumed */
}
```



What about Filling all the Buffers?

- Suppose that we wanted to provide a solution to the consumer-producer problem that **fills all the buffers**.
- How can we do it?



What about Filling all the Buffers? (ont.)

- We can do so by having *an integer counter* that keeps track of the number of full buffers.
- Initially, counter is set to 0.
- The integer counter is incremented by the producer after it produces a new buffer.
- The integer counter is decremented by the consumer after it consumes a buffer.



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 */  
}
```

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
```

Race Condition (cont.)

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

S0: 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 }



Race Condition (cont.)

Question – why was there no race condition in the first solution
(where at most $N - 1$) buffers can be filled?

More in Chapter 6.





Operating Systems

Processes-Part4

Seyyed Ahmad Javadi

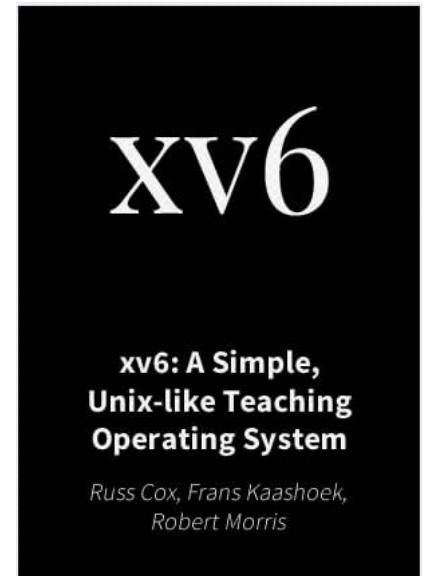
sajavadi@aut.ac.ir

Fall 2021

Course logistics

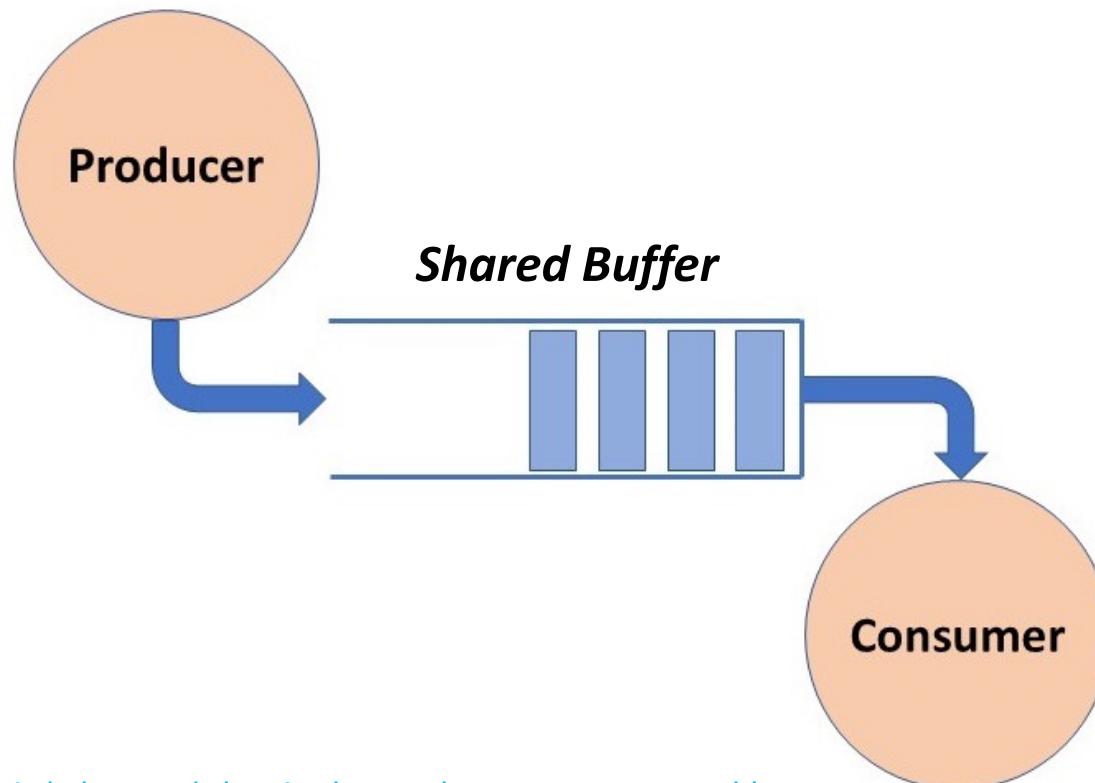
- You have **only few days** to complete
 - Phase 1 of the project
 - Your first homework

- We will have an extra session on fork
 - The session is handled by TAs
 - It covers both theoretical aspects and how use fork in practice
 - What time is best for you?



Producer-Consumer Problem

- Paradigm for cooperating processes:
 - Producer process produces information that is consumed by a consumer process.



<https://www.educative.io/edpresso/what-is-the-producer-consumer-problem>

Producer-Consumer Problem-Variations

- **Unbounded-buffer** places no practical limit on the size of the buffer:
 - Producer never waits
 - Consumer waits if there is no buffer to consumer
- **Bounded-buffer** assumes that there is a fixed buffer size
 - Producer must wait if all buffers are full
 - Consumer waits if there is no buffer to consume



IPC – 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.
- Synchronization is discussed in great details in Chapters 6 & 7.

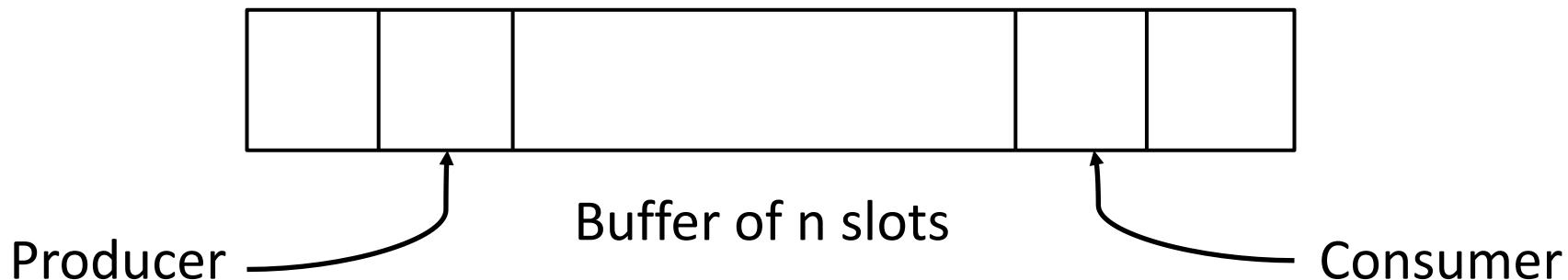


Bounded-Buffer – Shared-Memory Solution

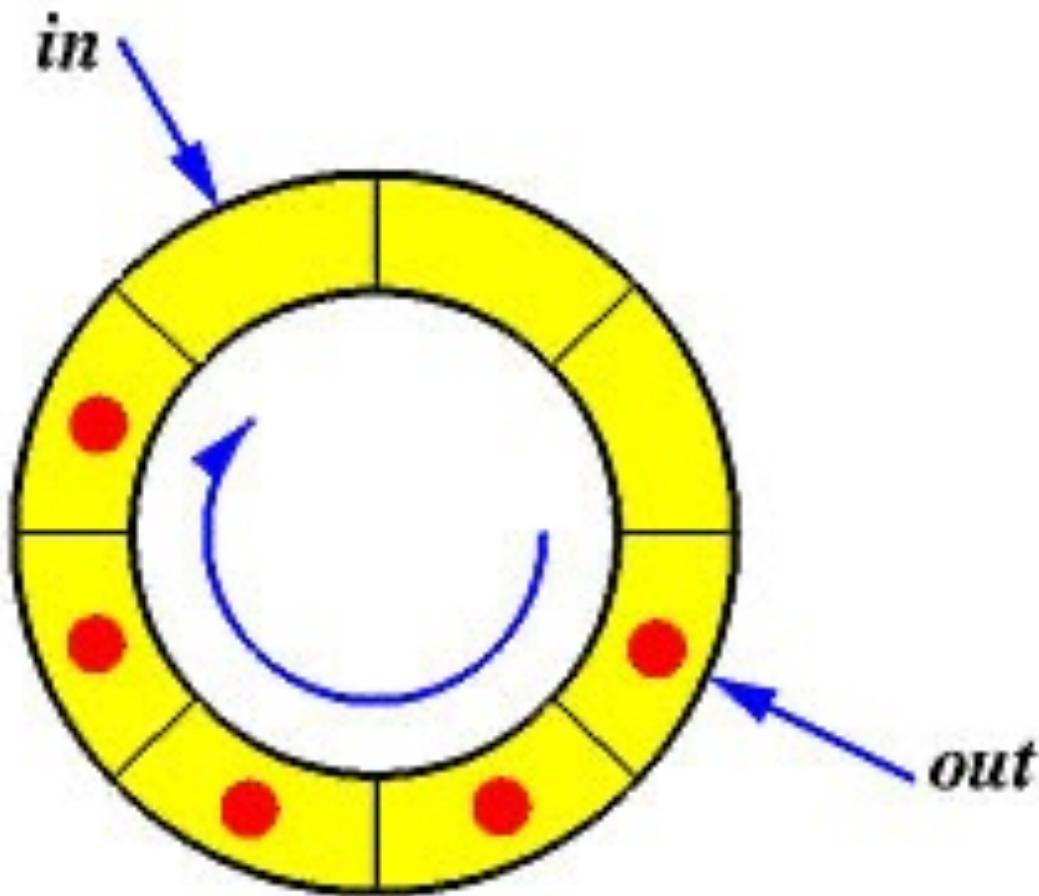
- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct but can only use **BUFFER_SIZE - 1** elements.



Circular Bounded-Buffer



Source: <https://pages.mtu.edu/~shene/NSF-3/e-Book/SEMA/TM-example-buffer.html>

Producer Process – Shared Memory

```
item next_produced;

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



Consumer Process – Shared Memory

```
item next_consumed;

while (true) {
    while (in == out)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next consumed */
}
```



What about Filling all the Buffers?

- Suppose that we wanted to provide a solution to the consumer-producer problem that **fills all the buffers**.
- How can we do it?



What about Filling all the Buffers? (ont.)

- We can do so by having *an integer counter* that keeps track of the number of full buffers.
- Initially, counter is set to 0.
- The integer counter is incremented by the producer after it produces a new buffer.
- The integer counter is decremented by the consumer after it consumes a buffer.



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 */  
}
```

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
```



Race Condition (cont.)

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

```
S0: 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 }
```



Race Condition (cont.)

Question – why was there no race condition in the first solution
(where at most $N - 1$) buffers can be filled?

More in Chapter 6.





Operating Systems

Threads and Concurrency

Seyyed Ahmad Javadi

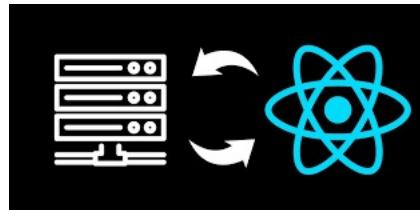
sajavadi@aut.ac.ir

Fall 2021

Motivation

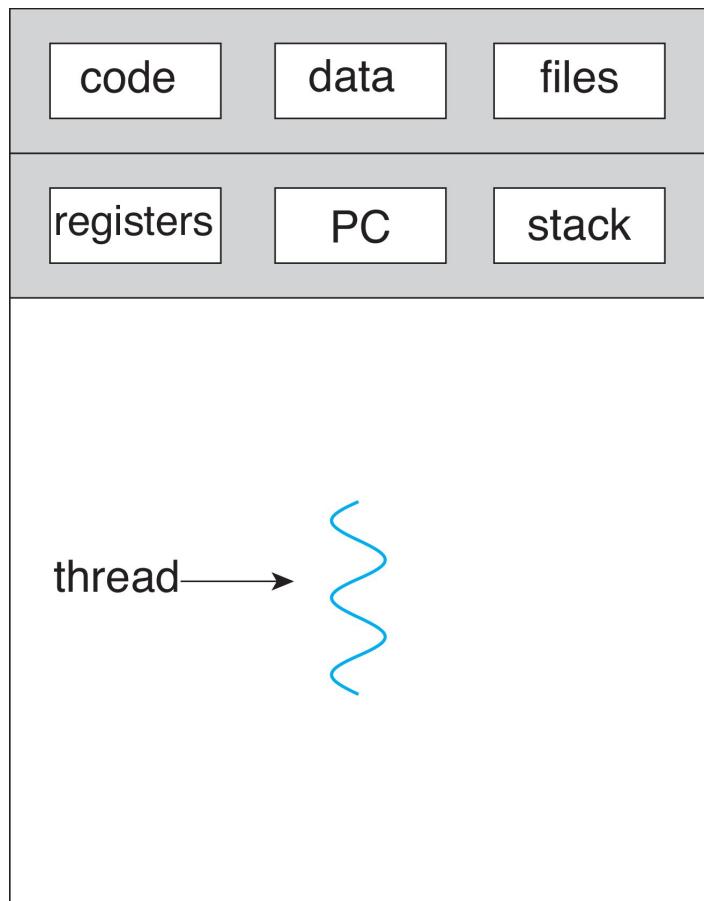
- Most modern applications are multithreaded
- Multiple tasks with the application can be implemented by threads

- Update display
- Fetch data
- Spell checking

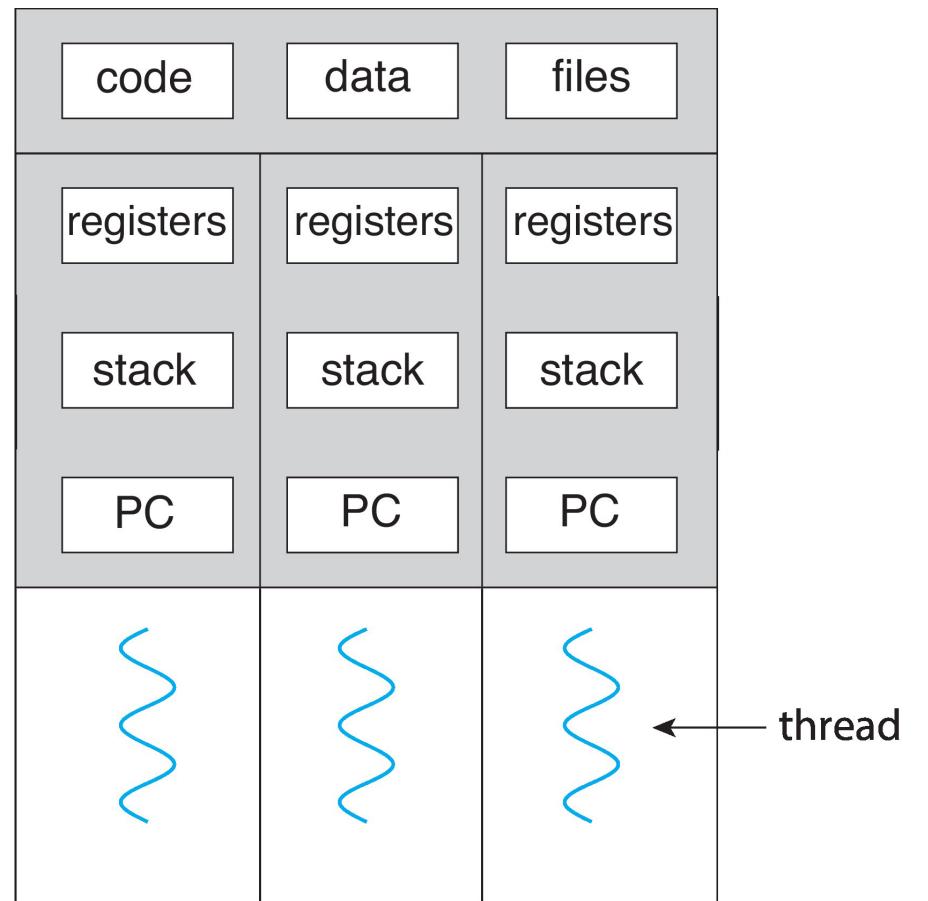


- Process creation is **heavy-weight** while thread creation is **light-weight**
- Kernels are generally multithreaded

Single and Multithreaded Processes

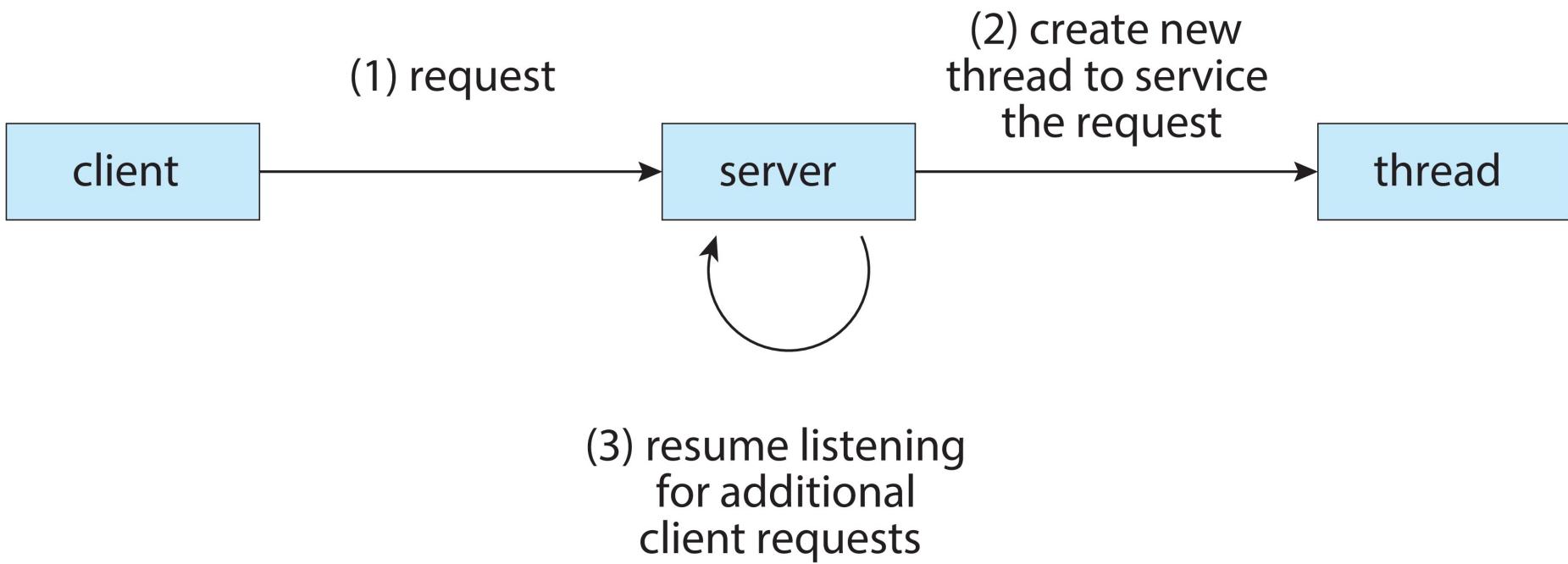


single-threaded process



multithreaded process

Multithreaded Server Architecture



Benefits

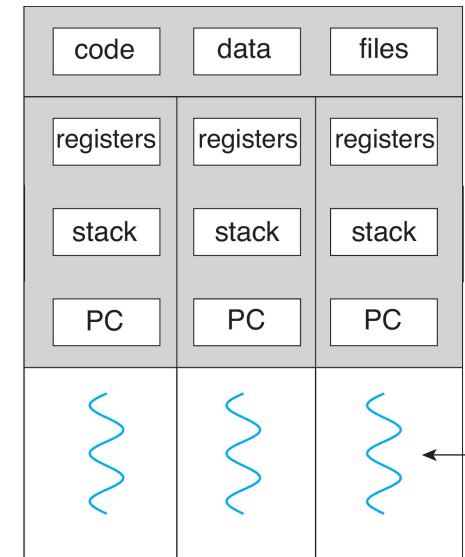
■ Responsiveness

- 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.



Benefits (cont.)

■ Economy

- Cheaper than process creation
- Thread switching lower overhead than context switching.



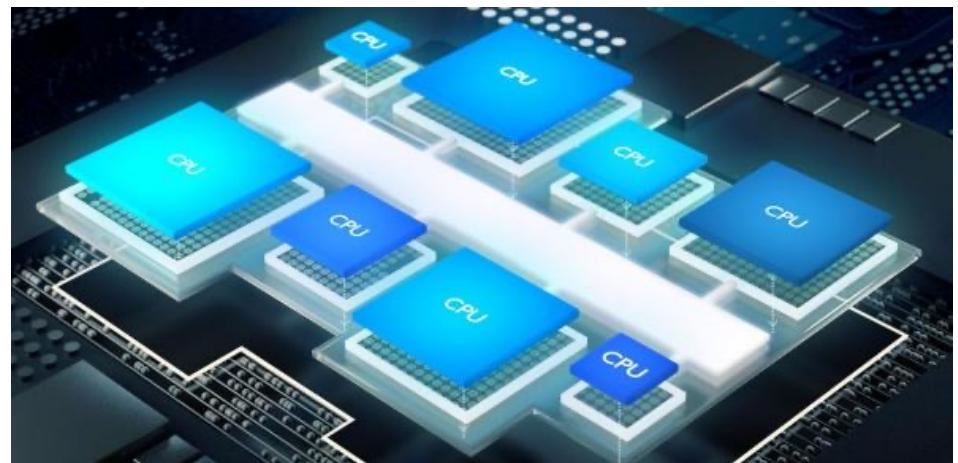
■ Scalability

- Process can take advantage of multicore architectures.



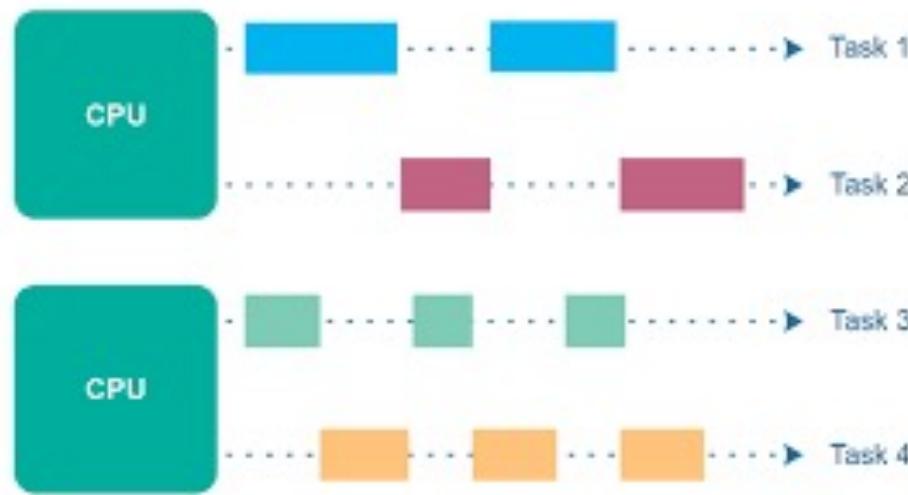
Multicore Programming

- Multicore or multiprocessor systems putting pressure on programmers, challenges include:
 - Dividing activities
 - Balance: ensuring that the tasks perform equal work of equal value.
 - Data splitting
 - Data dependency
 - Testing and debugging



Multicore Programming (cont.)

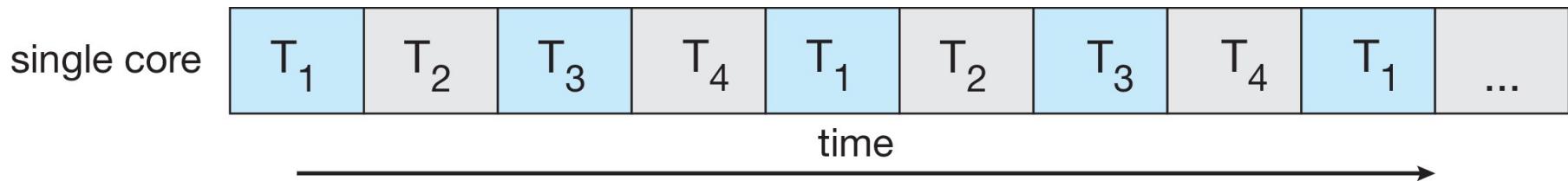
- **Parallelism** implies performing more than one task **simultaneously**



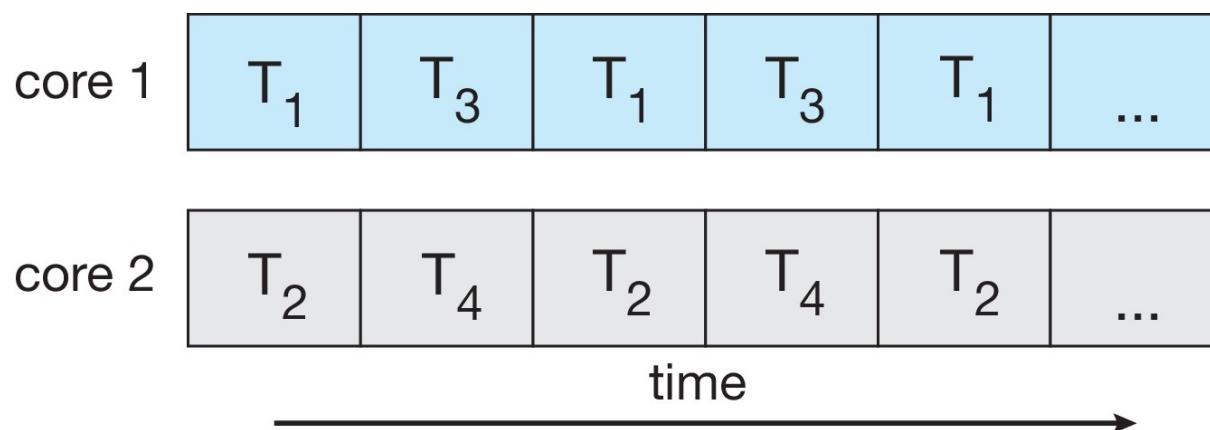
- **Concurrency** supports more than one task **making progress**
 - Single processor or core, scheduler providing concurrency

Concurrency vs. Parallelism

- Concurrent execution on single-core system:



- Parallelism on a multi-core system:



Multicore Programming-Types of Parallelism

■ Data parallelism

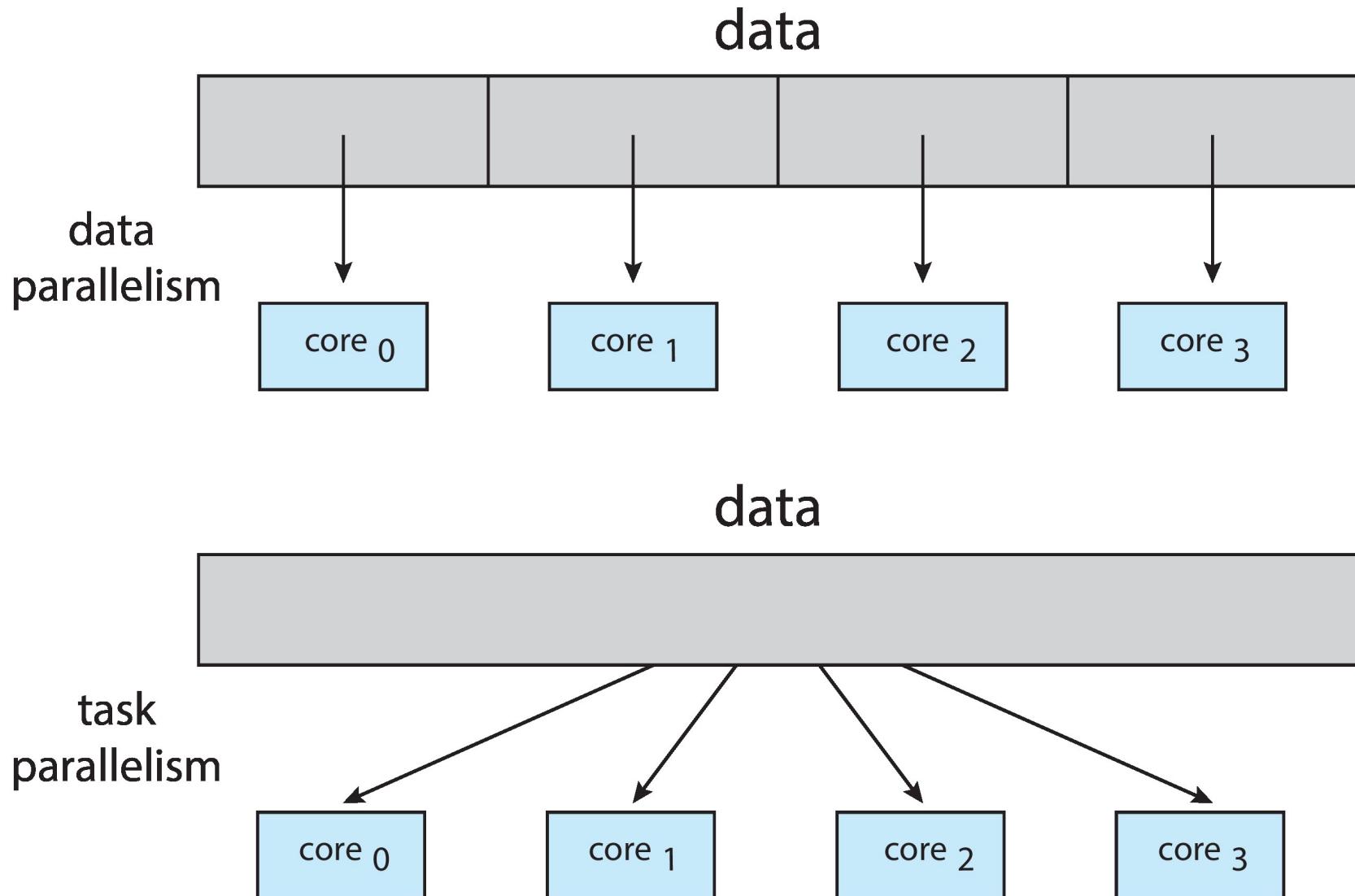
- Distributes subsets of the same data across multiple cores, same operation on each
- Example: summing the contents of an array of size N.

■ Task parallelism

- Distributing threads across cores, each thread performing unique operation
- Example: Unique statistical operation on the array of elements.



Data and Task Parallelism

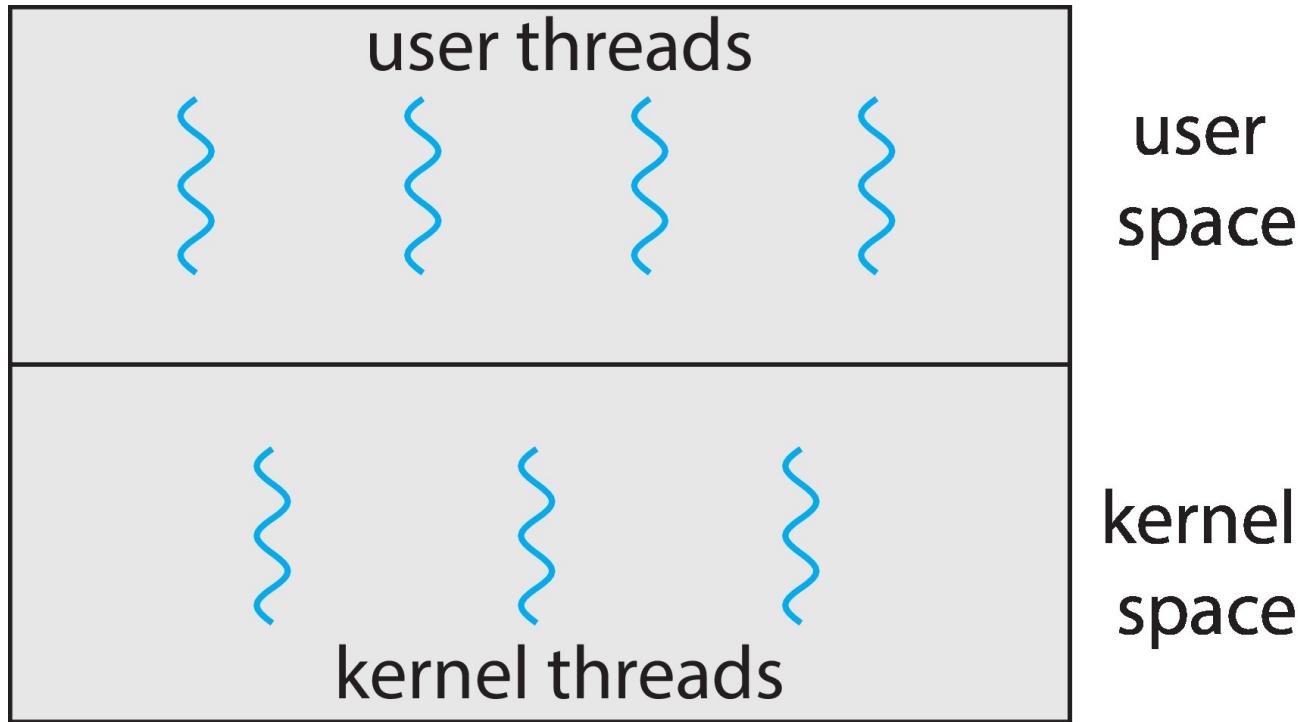


User Threads and Kernel Threads

- **User threads:** management done by user-level threads library.
- Three primary thread libraries:
 - POSIX Pthreads
 - Windows threads
 - Java threads
- **Kernel threads:** supported by the Kernel
 - Examples – virtually all general -purpose operating systems, including: Windows, Linux, Mac OS X, iOS, Android



User and Kernel Threads



Additional review: <https://www.geeksforgeeks.org/difference-between-user-level-thread-and-kernel-level-thread/>

Multithreading Models

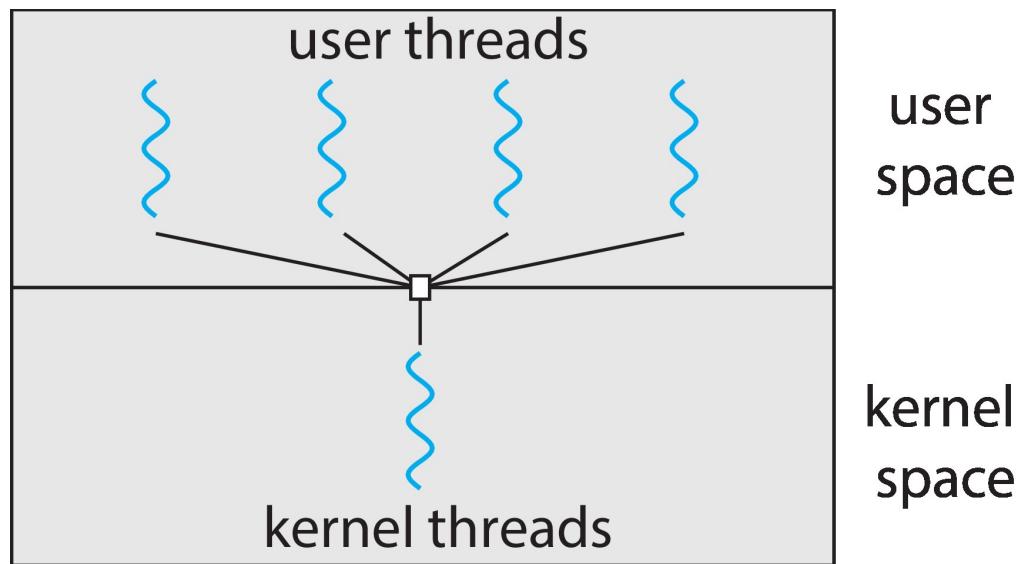
- How to map user threads to kernel threads?
- Many-to-One
- One-to-One
- Many-to-Many

Additional review:

<https://stackoverflow.com/questions/14791278/threads-why-must-all-user-threads-be-mapped-to-a-kernel-thread>

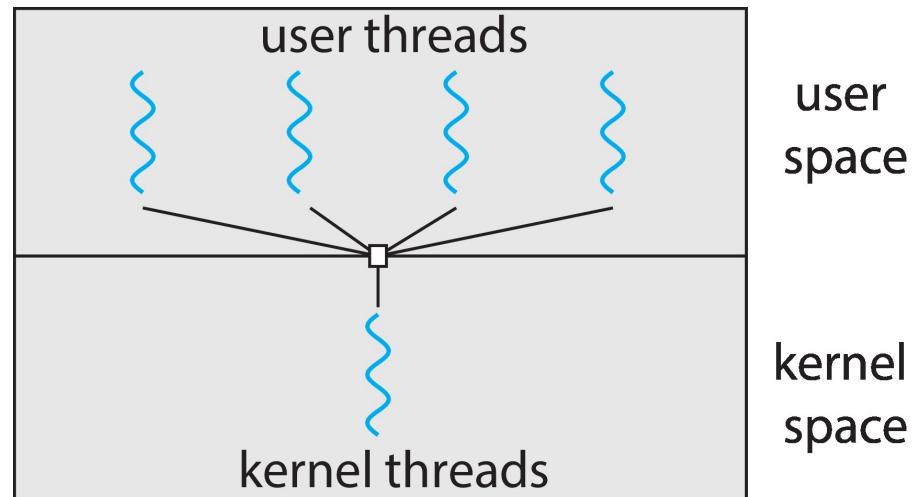
Many-to-One

- Many user-level threads mapped to single kernel thread
- Thread management is done by the thread library in user space
 - So it is efficient



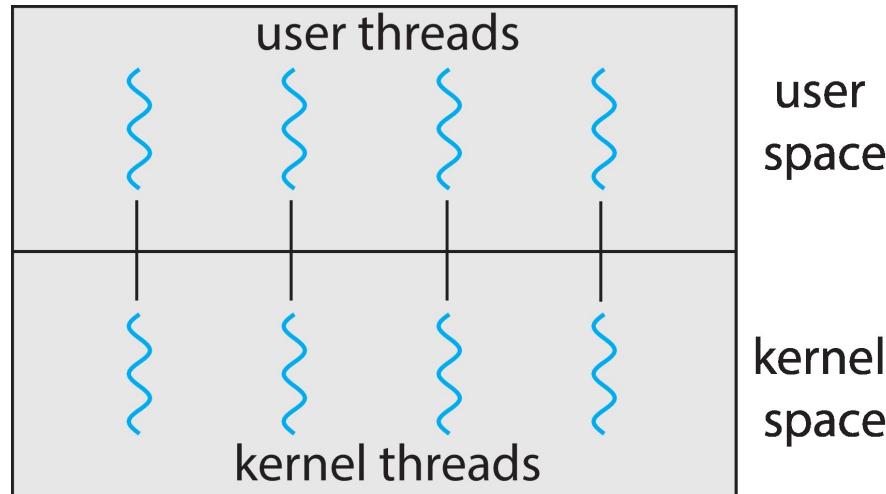
Many-to-One (cont.)

- 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



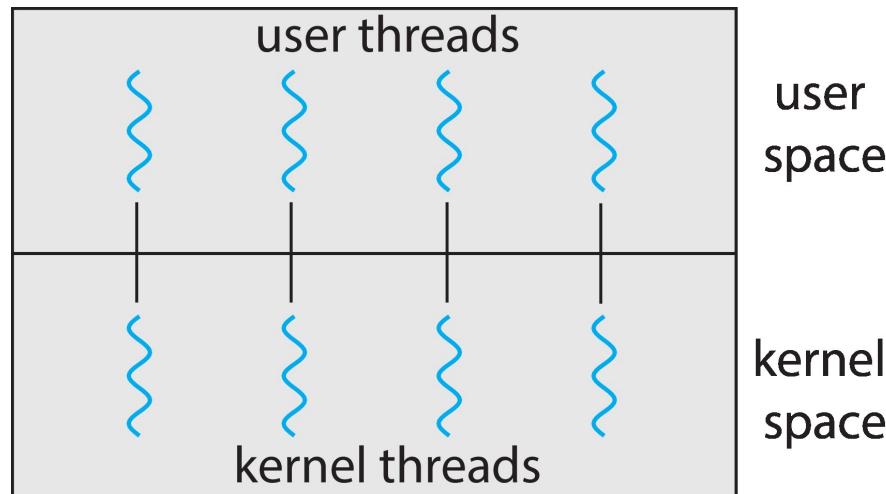
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread



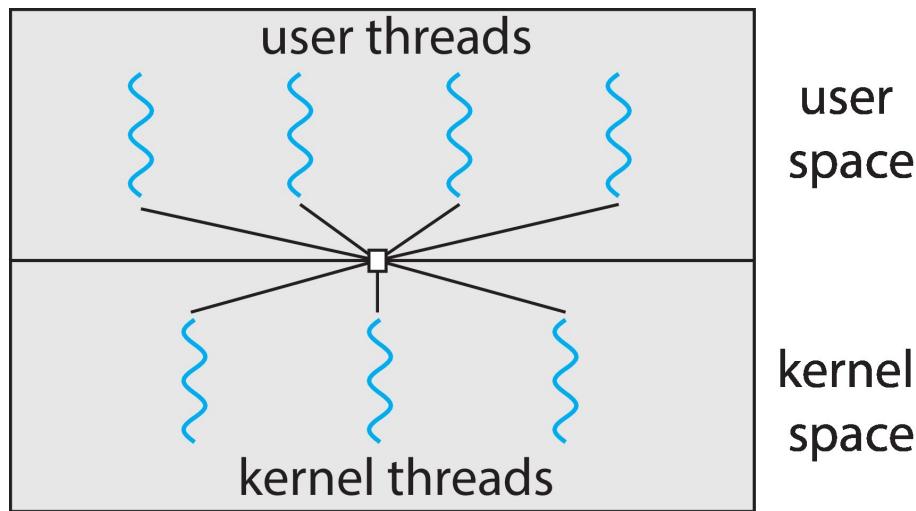
One-to-One (cont.)

- More concurrency than many-to-one
- Number of threads per process may be restricted due to overhead
- Examples
 - Windows
 - Linux



Many-to-Many Model

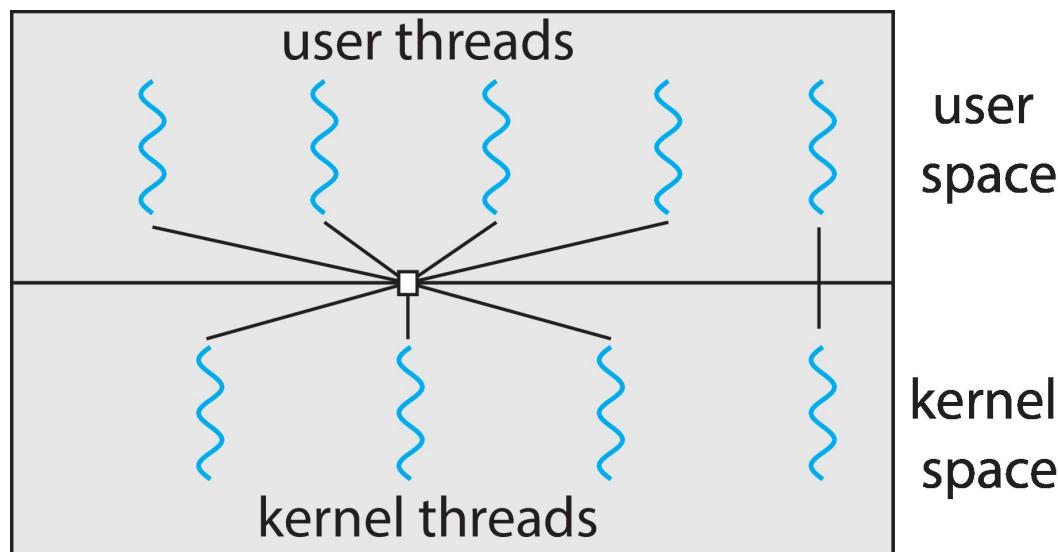
- Many user level threads to be mapped to many kernel threads
- Operating system can create a sufficient number of kernel threads



- Examples
 - Windows with the *ThreadFiber* package
 - Otherwise not very common

Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread





Operating Systems

Thread Libraries & Signal handling

Seyyed Ahmad Javadi

sajavadi@aut.ac.ir

Fall 2021

Thread Libraries

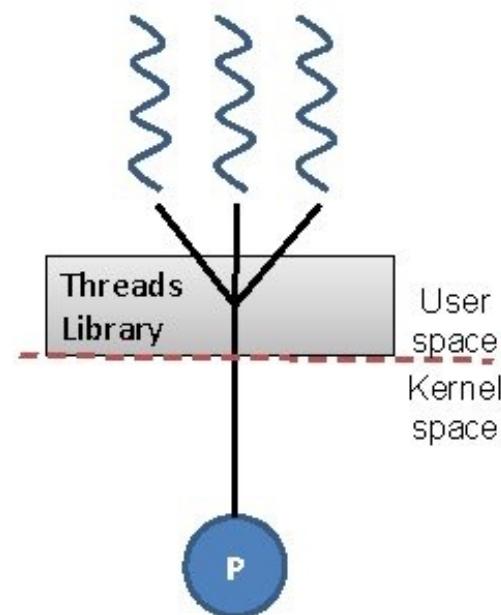
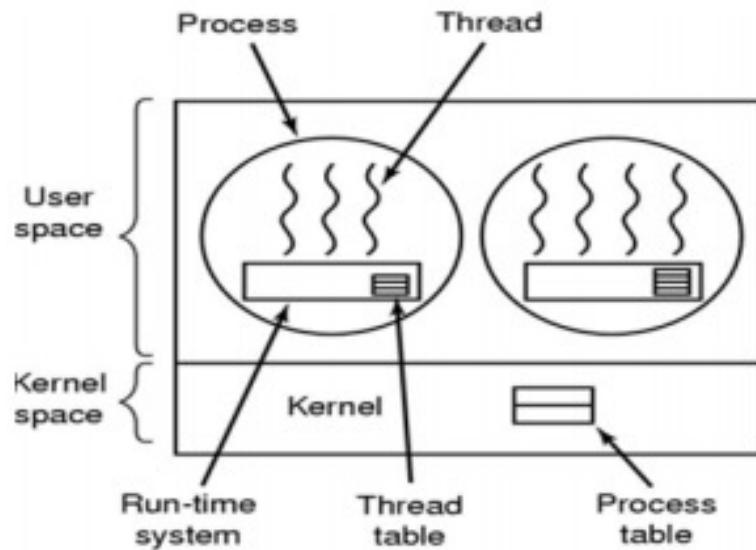
- Thread library provides programmer with API for creating and managing threads.
- Two primary ways of implementing
 - User-space library
 - Kernel-level library

https://www.tutorialspoint.com/operating_system/os_multi_threading.htm



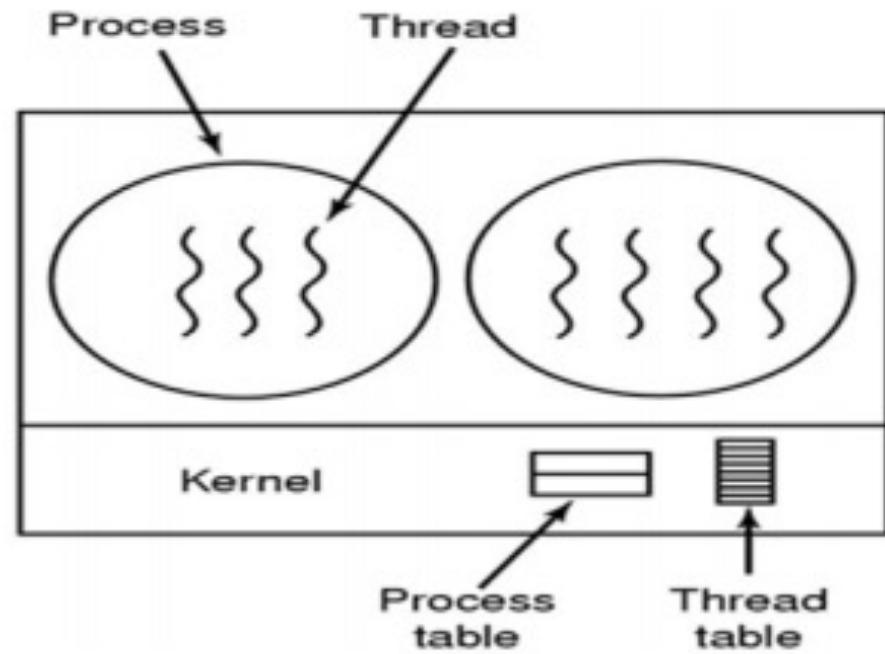
Library entirely in user space

- All code and data structures for the library *exist in user space*.
- Invoking a function in the library *results in a local function call* in user space and *not a system call*.



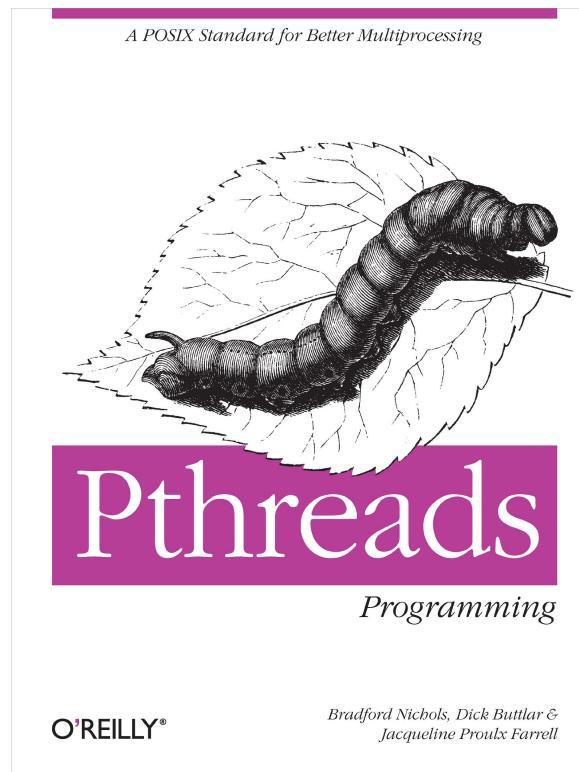
Kernel-level library supported directly by the OS

- Code and data structures for the library exist in kernel space.
- Invoking a function in the API for the library typically ***results in a system call*** to the kernel.



PThreads

- May be provided either as user-level or kernel-level.
- A POSIX standard API for thread creation and synchronization.



Pthreads (cont.)

- ***Specification, not implementation.***
- API specifies behavior of the thread library
 - Implementation is up to development of the library.
- Common in UNIX operating systems
 - Linux & Mac OS X

Optional reading:

<https://users.cs.cf.ac.uk/Dave.Marshall/C/node30.html>

<https://stackoverflow.com/questions/43219214/where-is-the-value-of-the-current-stack-pointer-register-stored-before-context-s>

Pthreads Example

```
#include <pthread.h>
#include <stdio.h>

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    /* set the default attributes of the thread */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid, &attr, runner, argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid,NULL);

    printf("sum = %d\n",sum);
}
```



Pthreads Example (Cont.)

```
/* The thread will execute in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```



Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```



Threading Issues

- Semantics of **fork()** and **exec()** system calls

- Signal handling
 - Synchronous and asynchronous

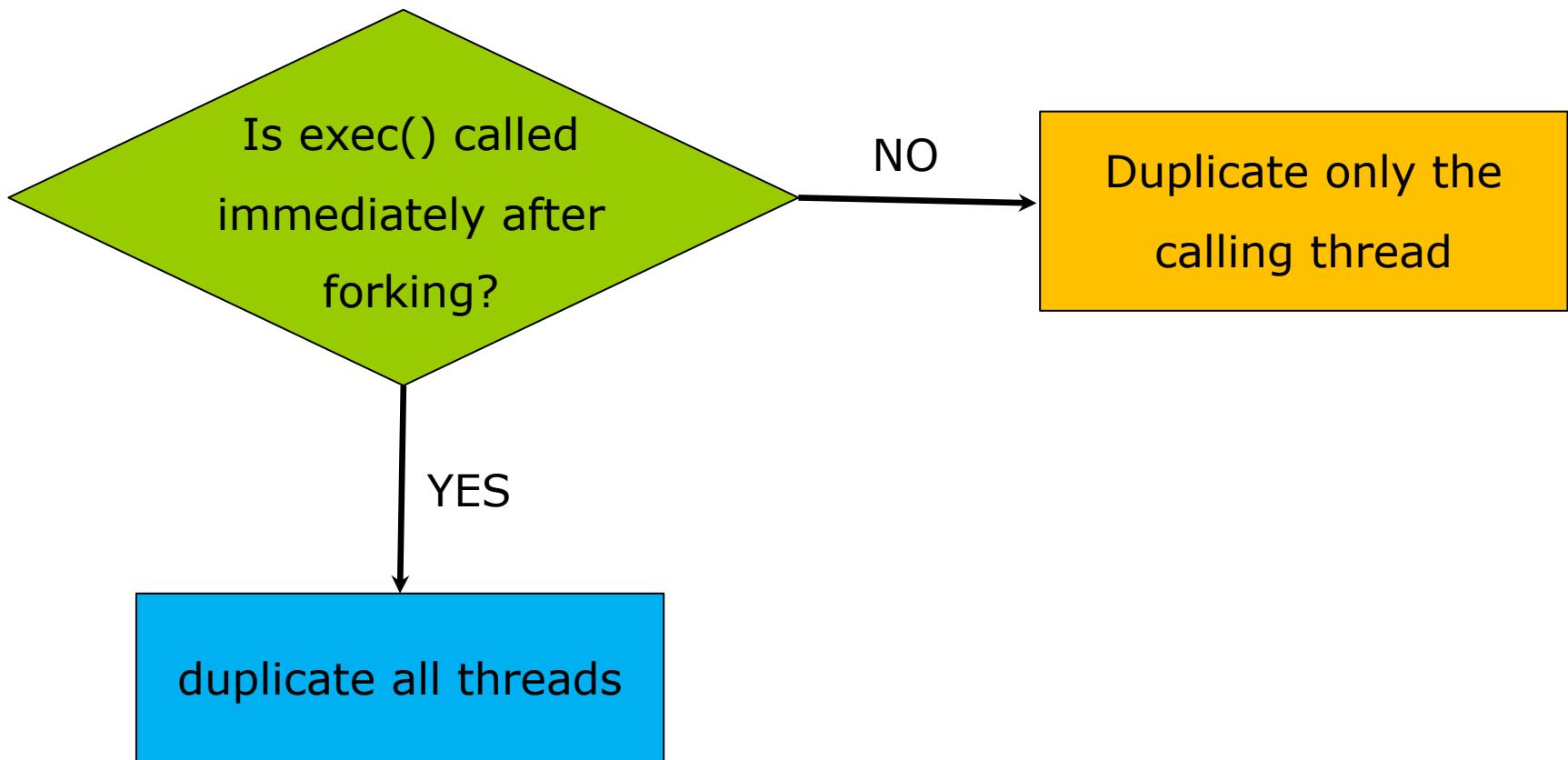
Semantics of fork() and exec()

- Does fork() duplicate only the calling thread or all threads?
 - Some UNIXes have **two versions of fork**
 - ▶ One that duplicates all threads
 - ▶ Another that duplicates only the thread that invoked the fork()
- exec() usually works as normal
 - Replace the running process including all threads.



Which Version of Fork() to use?

Depends on the application.



Which Version of Fork() to use?

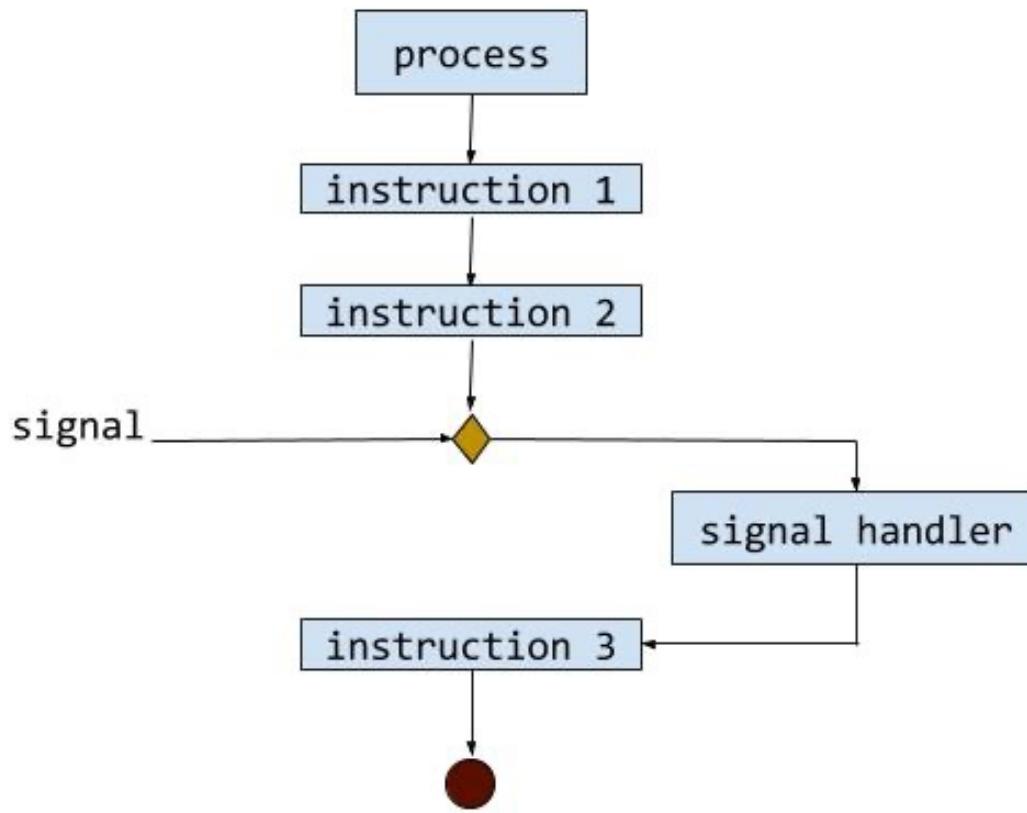
- **If exec() is called immediately after forking**
 - Duplicating only the calling thread is appropriate.
 - Since the program specified in the parameters to exec() will replace the process.

- **If the child process does not call exec() after forking**
 - **The child process should duplicate all threads.**



Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.



Signal Handling

- A **signal handler** is used to process signals
 - 1. Signal is generated by particular event
 - 2. Signal is delivered to a process
 - 3. Signal is handled by one of two signal handlers:
 - 1. **default**
 - 2. **user-defined**

Two Types of Signals

- **Synchronous**
- **Asynchronous**

Synchronous Signals

- When a signal is generated by an event internal to a running process.
- Example:
 - illegal memory access
 - divide by 0
- Synchronous signals are delivered to the same process that
 - performed the operation that caused the signal
 - that is the reason they are considered synchronous



Asynchronous Signals

- Signal is *generated by an event external* to a running process, that process receives the signal asynchronously.
- Example: terminating a process with specific keystrokes
 - <control><C>
- Typically, an asynchronous signal is sent to another process.



Signal Handling (Cont.)

- Every signal has **default-handler** that kernel runs when handling it
 - Some signals are simply ignored
 - ▶ such as changing the size of a window
 - Others are handled by terminating the program.
 - ▶ such as an illegal memory access
- User-defined **signal handler** can override default.
- For single-threaded, signal delivered to process.



Signal Handling (Cont.)

- Where should a signal be delivered for multi-threaded?
 - Deliver the signal to the thread to which the signal applies
 - Deliver the signal to every thread in the process
 - Deliver the signal to certain threads in the process
 - Assign a specific thread to receive all signals for the process
- Which one should be used?
 - Depends on the type of signal generated.



Signal Handling (cont.)

- **Synchronous signals** need to be delivered to the thread causing the signal and not to other threads in the process.
- However, the situation **with asynchronous signals is not as clear.**
 - Some asynchronous signals should be sent to all threads
 - ▶ Such as a signal that terminates a process (e.g., <control><C>)

Signal Handling (cont.)

- Most multithreaded versions of UNIX allow a thread to specify which signals it will accept and which it will block.
 - Therefore, in some cases, an asynchronous signal may be delivered only to those threads that are not blocking it.
- However, a signal is typically delivered only to the first thread found that is not blocking it.
 - Because signals need to **be handled only once**



Functions for Delivering Signals

- The standard UNIX function

```
kill(pid t pid, int signal)
```

- Specifies the process to which a particular signal is to be delivered.

- POSIX Pthreads function

```
pthread_kill(pthread t tid, int signal)
```

- Allows a signal to be delivered to a specified thread (tid)



Operating Systems

CPU Scheduling-Part1

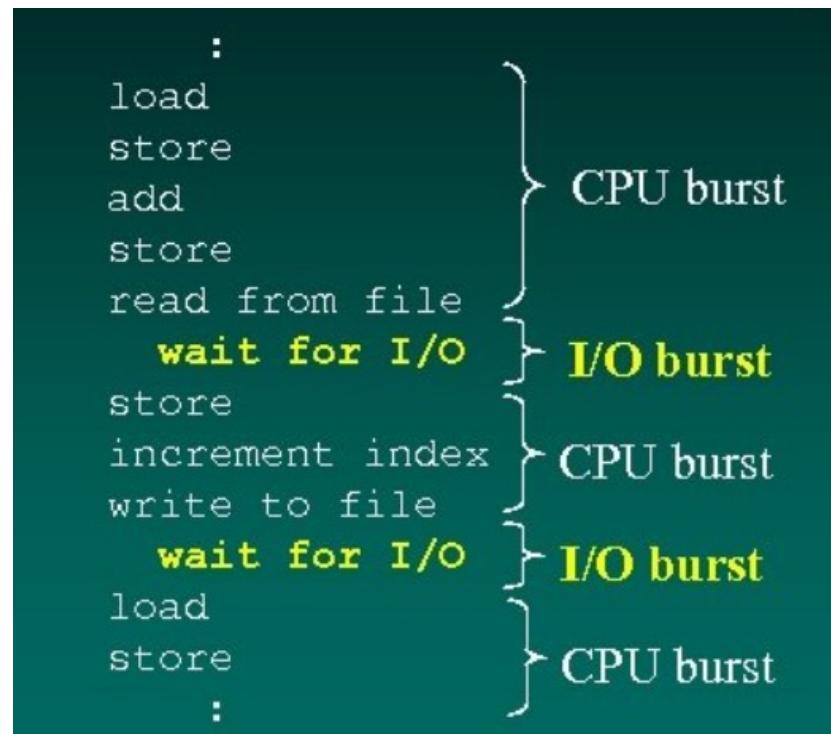
Seyyed Ahmad Javadi

sajavadi@aut.ac.ir

Fall 2021

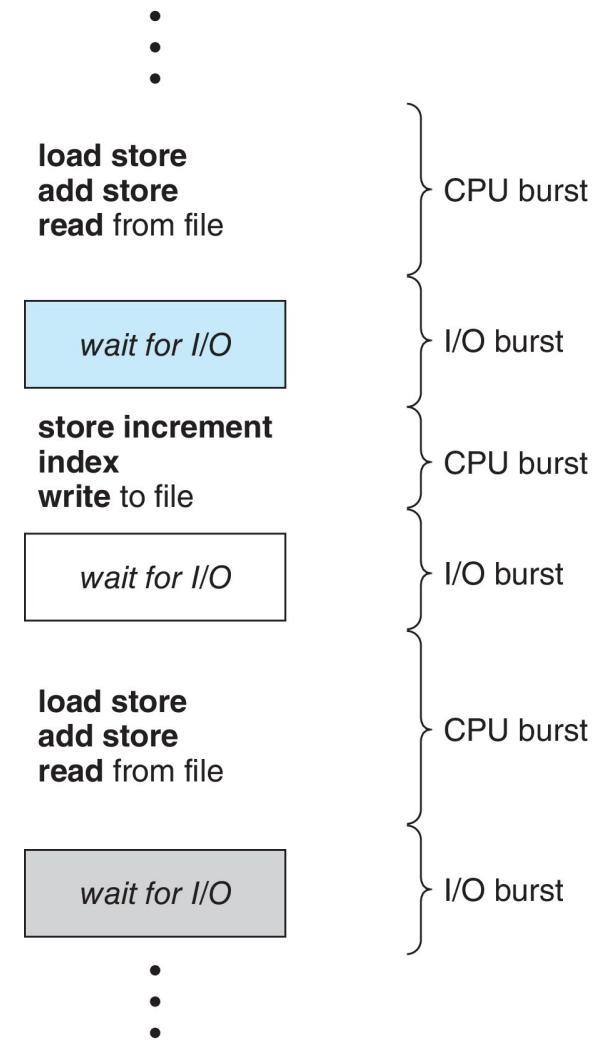
Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle
 - Process execution consists of a **cycle** of CPU execution and I/O wait



Basic Concepts

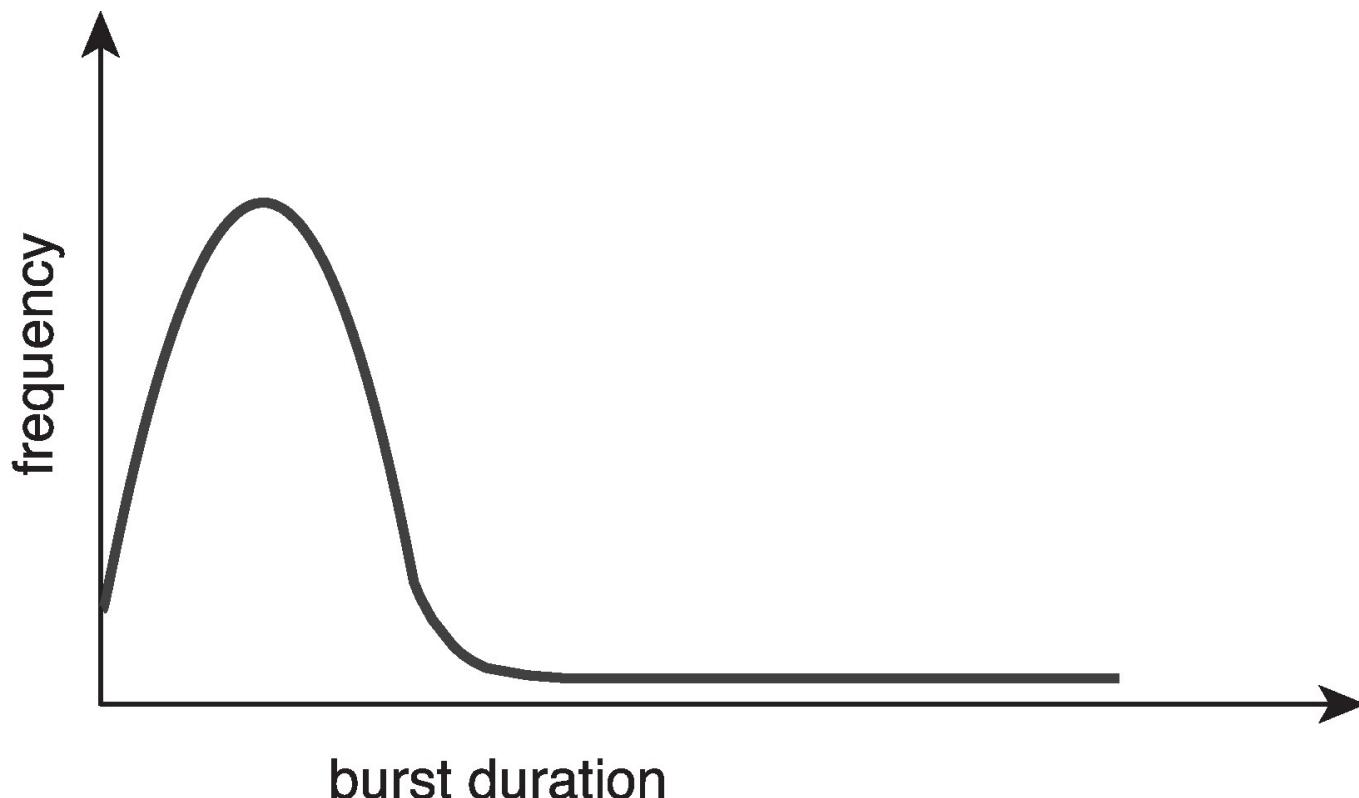
- CPU burst followed by I/O burst
- CPU burst distribution is of main concern



Histogram of CPU-burst Times

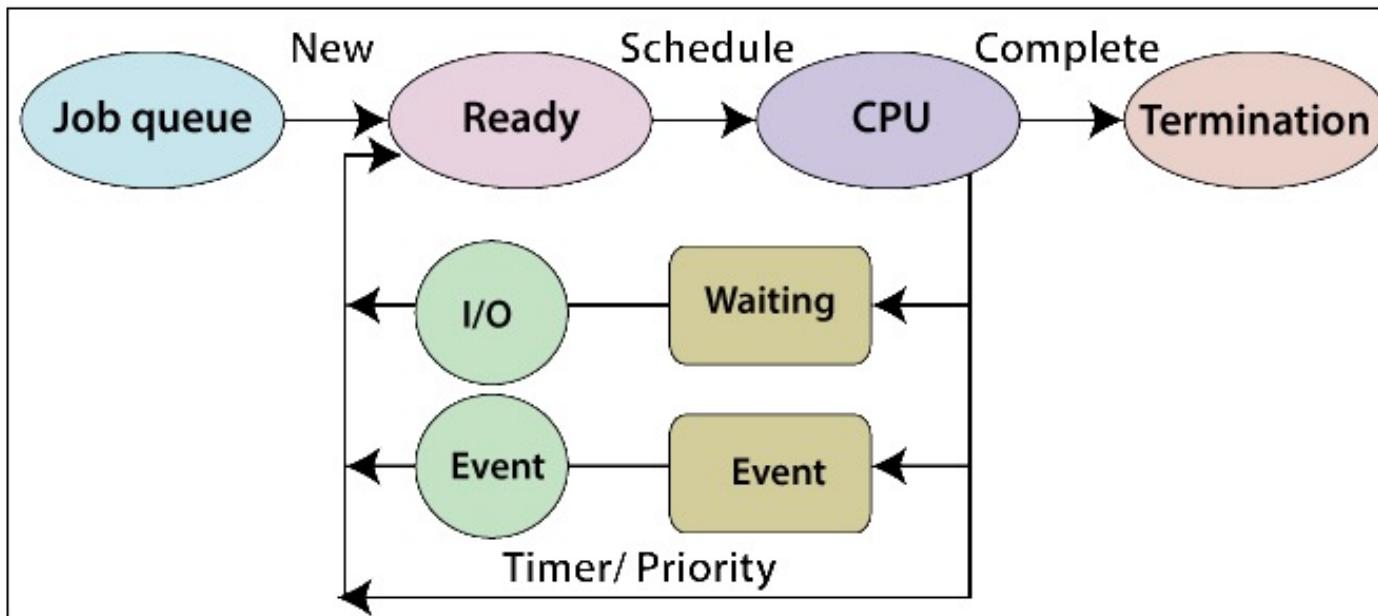
Large number of short bursts

Small number of longer bursts



CPU Scheduler

- The CPU scheduler selects from among the processes in ready queue and allocates a CPU core to one of them.
 - Queue may be ordered in various ways.

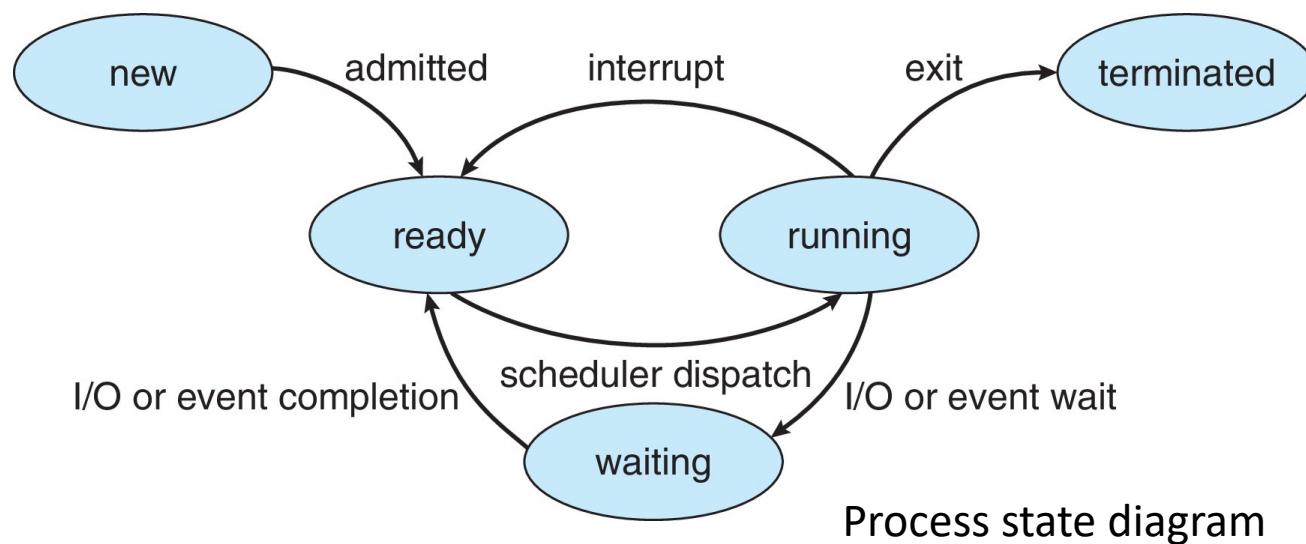


<https://www.tutorialandexample.com/process-schedulers-and-process-queue/>



CPU Scheduler (cont.)

- CPU scheduling decisions may take place when a process:
 1. Switches from **running** to **waiting** state
 2. Switches from **running** to **ready** state
 3. Switches from **waiting** to **ready**
 4. **Terminates**



CPU Scheduler (cont.)

- Four possible scheduling situations
 1. Switches from running to waiting state
 2. Switches from running to ready state
 3. Switches from waiting to ready
 4. Terminates
- For situations 1 and 4, there **is no choice in terms of scheduling.**
 - A new process must be selected for execution.
 - If at least one process exists in the ready queue
- For situations 2 and 3, however, there is a choice.



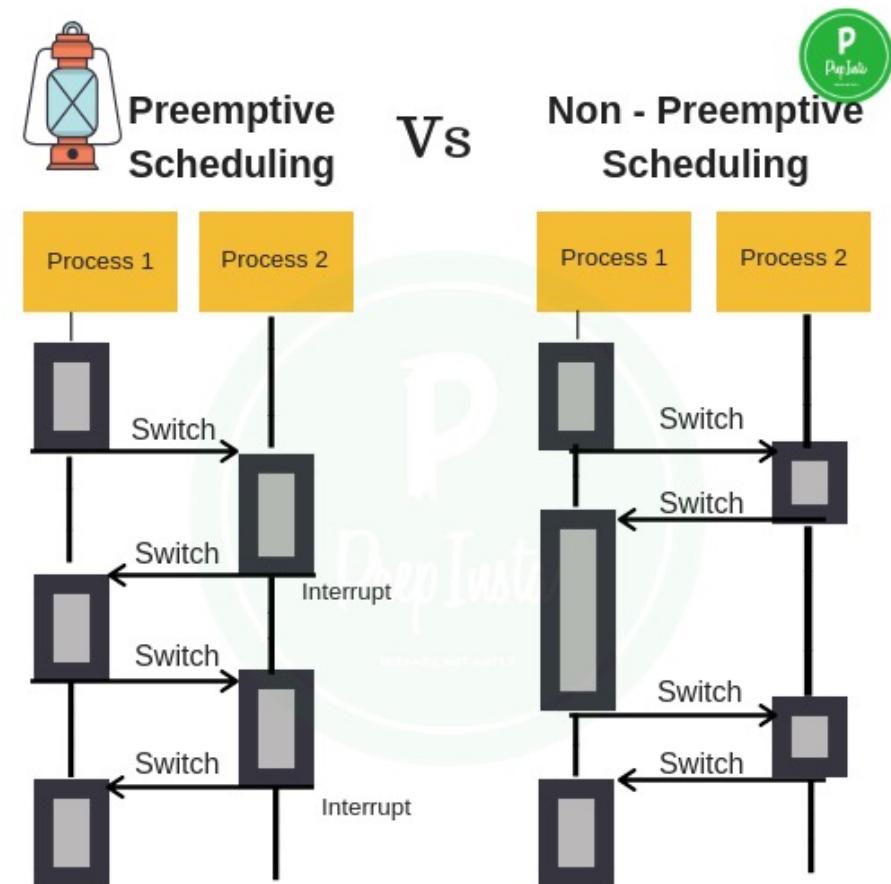
Preemptive and Nonpreemptive Scheduling

■ Non-preemptive (or cooperative)

- Circumstances 1 and 4

■ Preemptive

- Circumstances 2 and 3



Preemptive and Non-preemptive Scheduling (cont.)

■ Non-preemptive scheduling

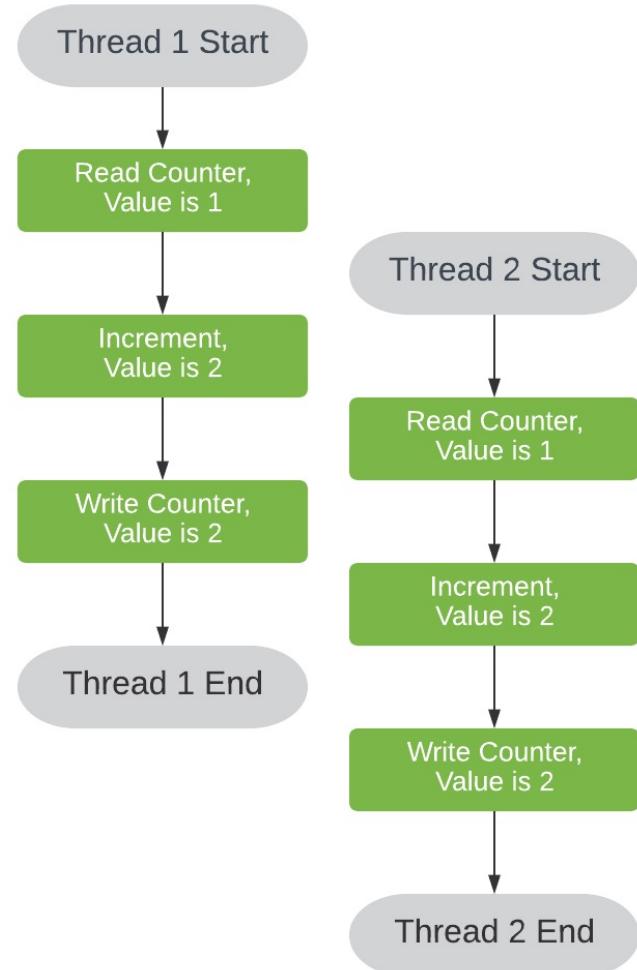
- Once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state.

■ Virtually all modern operating systems use preemptive scheduling algorithms.

- Including Windows, MacOS, Linux, and UNIX

Preemptive Scheduling and Race Conditions

- Preemptive scheduling can result in race conditions when *data are shared* among several processes.



Preemptive Scheduling and Race Conditions (cont.)

- Consider the case of two processes that share data.
 - While one process is **updating the data**, it is preempted so that the second process can run.
 - The second process then tries to read the data, which are in an **inconsistent state**.
- This issue will be explored in detail in Chapter 6.



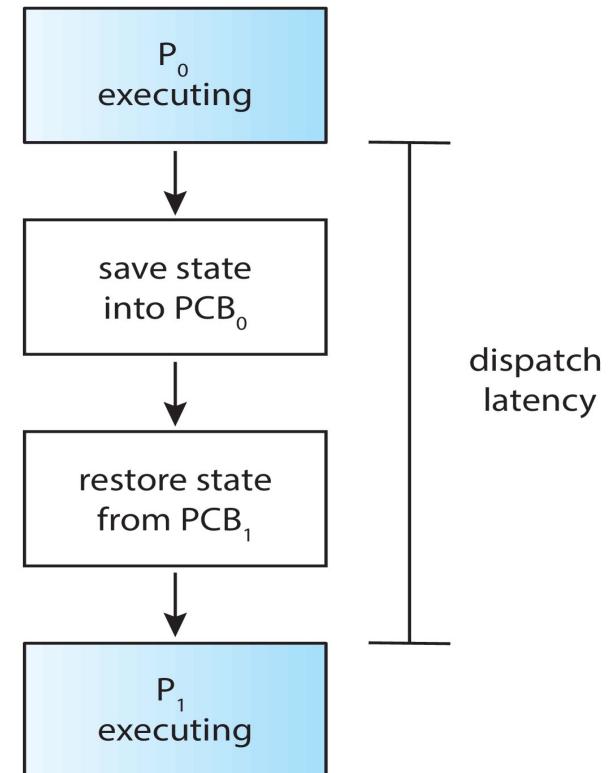
Dispatcher

- Gives control of the CPU to the process selected by the CPU 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

THROUGHPUT

TURNAROUND TIME

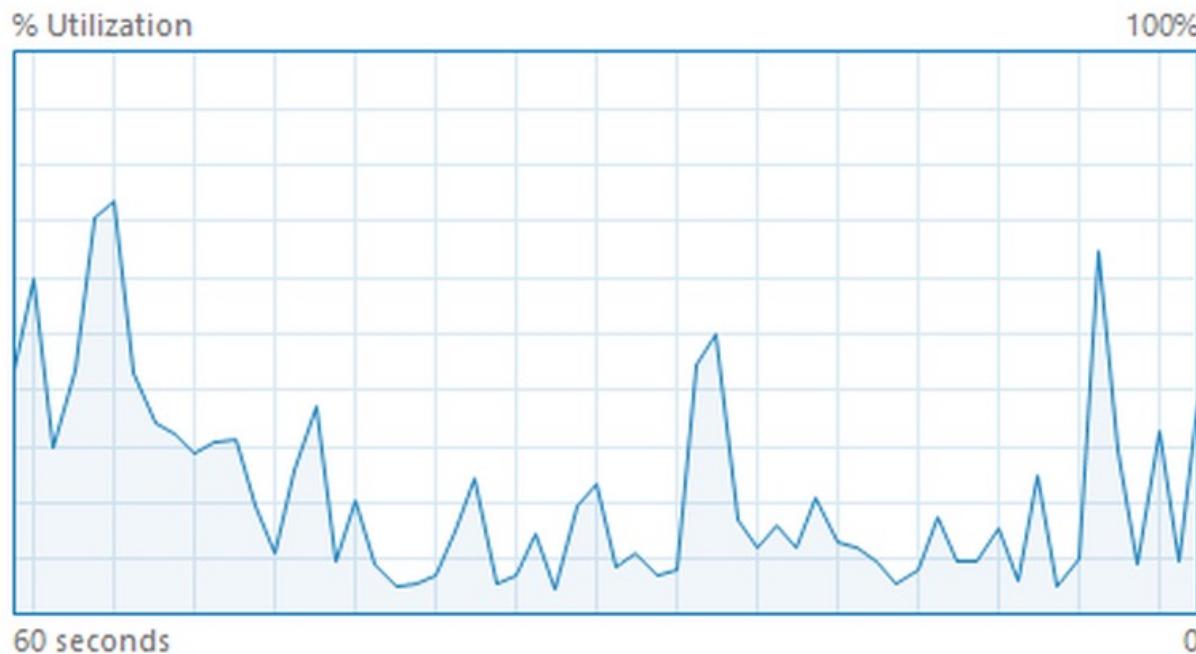
WAITING TIME

RESPONSE TIME



CPU utilization

- Keep the CPU as busy as possible.



Throughput

- Number of processes that complete their execution per time unit.



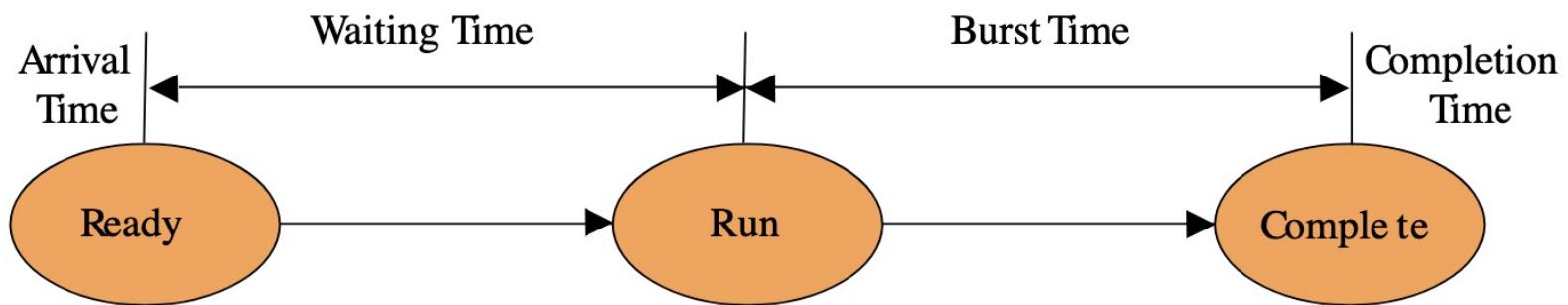
Turnaround time

- Amount of time to execute a particular process.
- Sum of the periods spent waiting, in the ready queue, executing on the CPU, and doing I/O.



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.



Scheduling Algorithm Optimization Criteria

Criteria	Min or Max?
CPU utilization	
Throughput	
Turnaround time	
Waiting time	
Response time	

Scheduling Algorithm Optimization Criteria

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

First-Come, First-Served

SCHEDULING ALGORITHM



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 $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case**



FCFS Scheduling and Convoy effect

- Short process behind long process.
 - Consider one CPU-bound and many I/O-bound processes.



- What is the important side-effect?

FCFS Scheduling and Convoy Effect (Cont.)

- Short process behind long process.
 - Consider one CPU-bound and many I/O-bound processes.
- What is the side-effect?
 - Results in **lower CPU and device utilization** than might be possible if the shorter processes were allowed to go first.





Operating Systems

CPU Scheduling-Part2

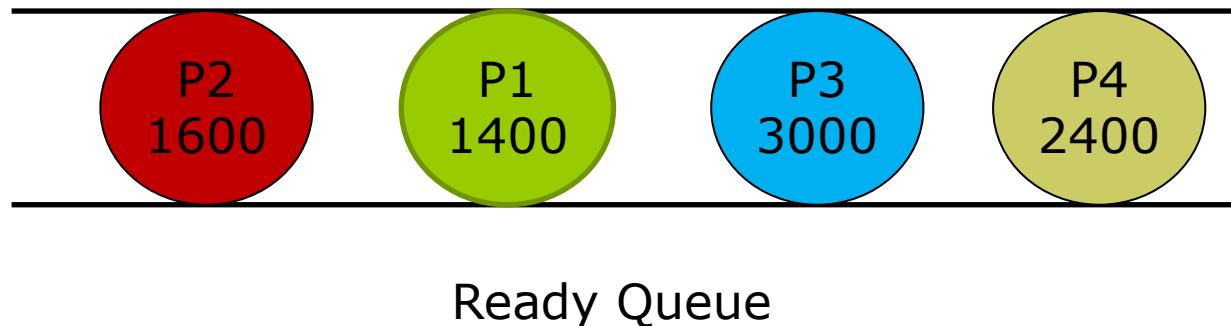
Seyyed Ahmad Javadi

sajavadi@aut.ac.ir

Fall 2021

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.



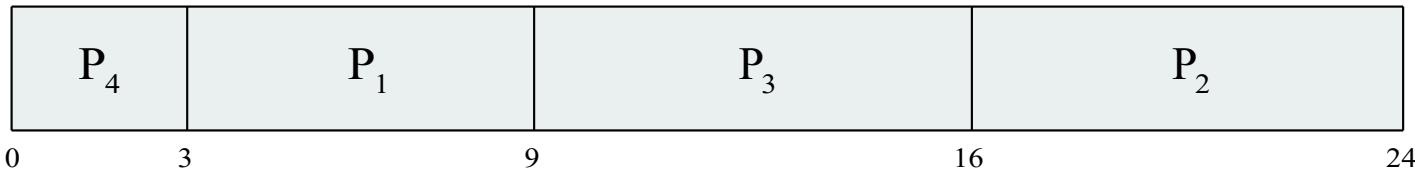
Shortest-Job-First (SJF) Scheduling (cont.)

- SJF is optimal
 - Gives minimum average waiting time for a given set of processes.
- Preemptive version called **shortest-remaining-time-first**
- The difficulty is knowing the length of the next CPU request
 - Could ask the user
 - Estimate (we do not cover this in the class and the exams)

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 = $(3 + 16 + 9 + 0) / 4 = 7$



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:**
 - Each process gets $1/n$ of the CPU time in chunks of at most q time units at once.
 - No process waits more than $(n-1)q$ time units.

Round Robin (RR) (cont.)

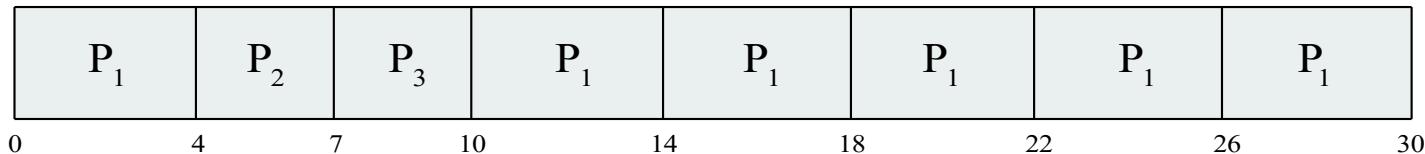
- Timer *interrupts* every quantum to schedule next process
- Performance
 - q large \Rightarrow FIFO
 - q small \Rightarrow q must be large with respect to context switch, otherwise overhead is too high



Example of RR with Time Quantum = 4

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

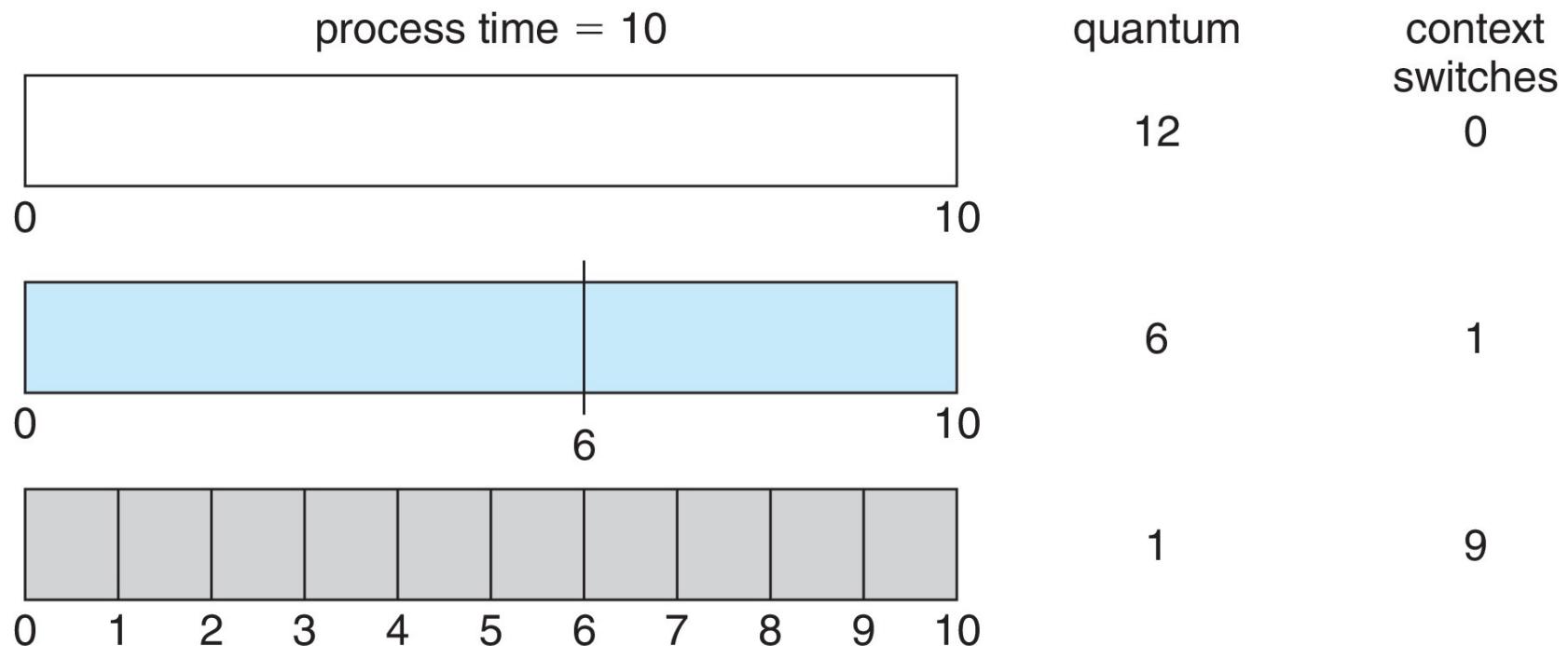
- The Gantt chart is:



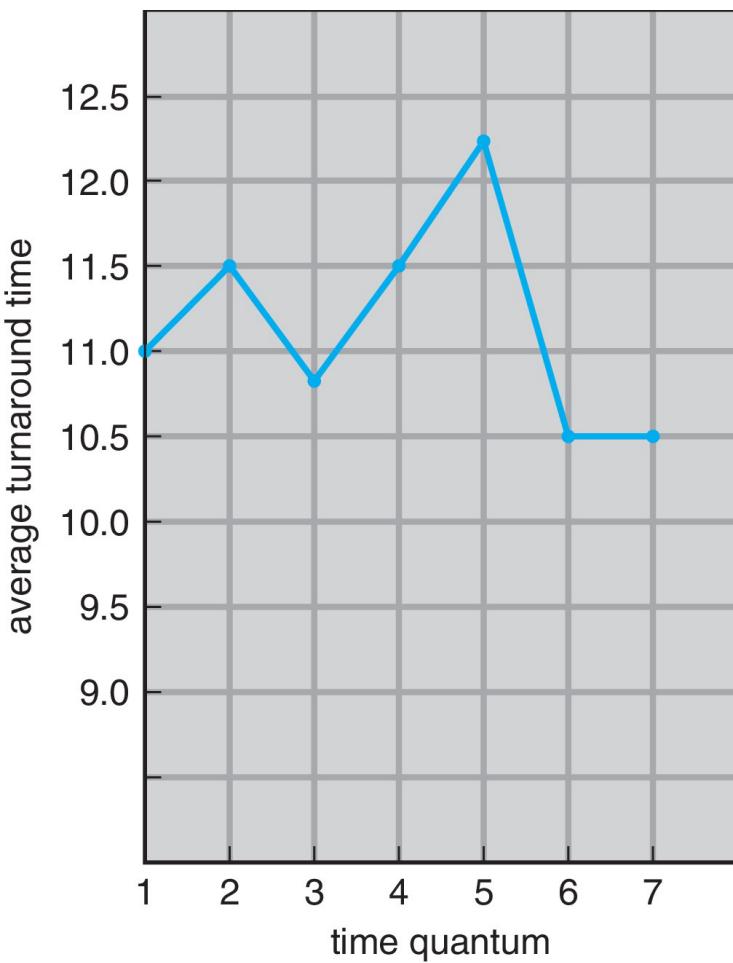
- Typically, higher average turnaround than SJF, but better **response**
- q should be large compared to context switch time
 - q usually 10 milliseconds to 100 milliseconds,
 - Context switch < 10 microseconds



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

A rule of thumb is that
80% of CPU bursts should
be shorter than q

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority
(smallest integer ≡ highest priority)
 - Preemptive
 - Nonpreemptive
- SJF is priority scheduling where priority is the ***inverse of predicted next CPU burst time***

Priority Scheduling

- **Problem ≡ Starvation**

- Low priority processes may never execute

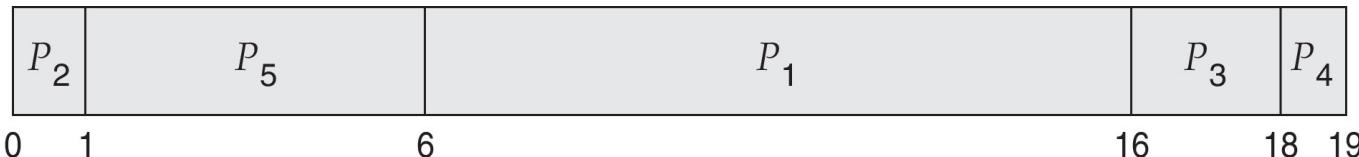
- **Solution ≡ Aging**

- 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

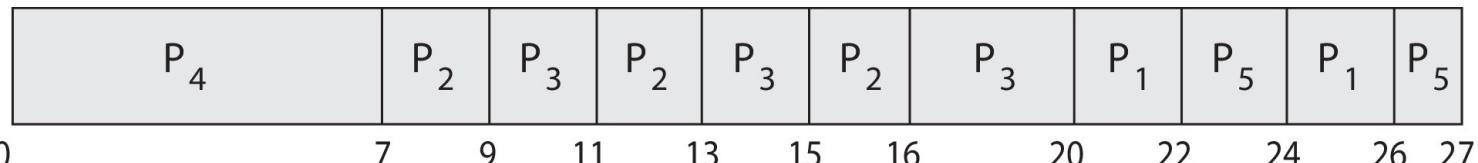


- Average waiting time = 8.2

Priority Scheduling w/ Round-Robin

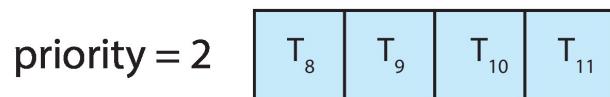
<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

- Run the process with the highest priority. Processes with the same priority run round-robin
- Gantt Chart with time quantum = 2



Multilevel Queue

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



•

•

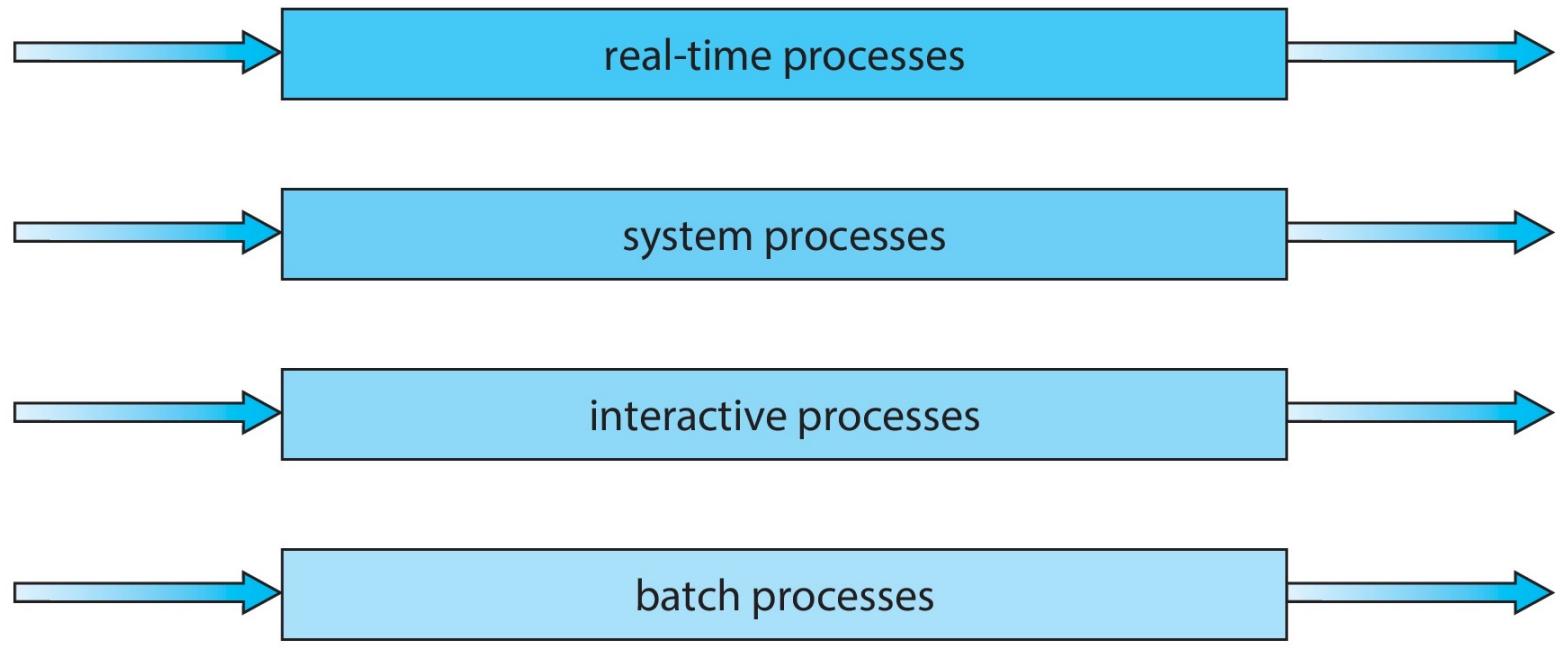
•



Multilevel Queue

- Prioritization based upon process type

highest priority



Multilevel Feedback Queue

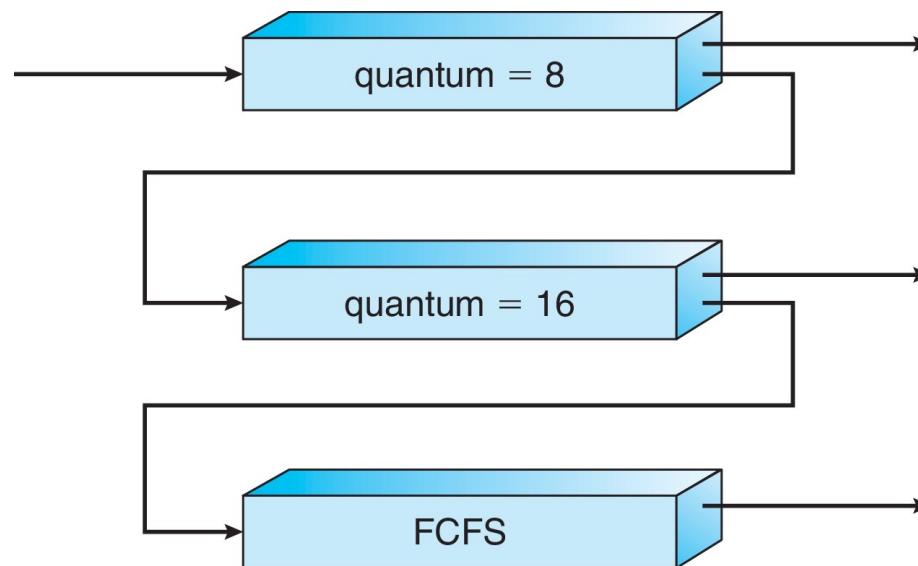
- A process can move between the various queues.
- Multilevel-feedback-queue defined by the following parameters:
 - Number of queues
 - Scheduling algorithms for each queue
 - Method used to determine when to upgrade a process
 - Method used to determine when to demote a process
 - Method used to determine which queue a process will enter when that process needs service
- **Aging** can be implemented using multilevel feedback queue



Example of Multilevel Feedback Queue

- Three queues:

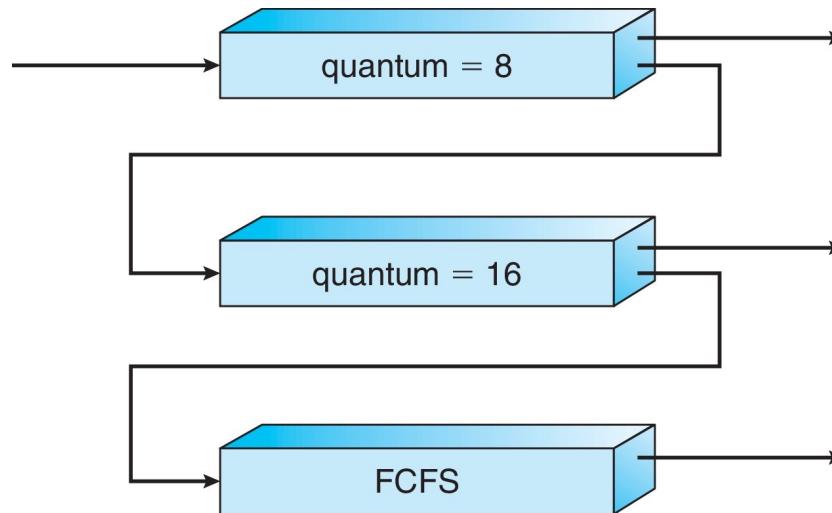
- Q_0 – RR with time quantum 8 milliseconds
- Q_1 – RR time quantum 16 milliseconds
- Q_2 – FCFS



Example of Multilevel Feedback Queue (cont.)

Scheduling

- A new process enters queue Q_0 which is served in RR
 - When it gains CPU, the process receives 8 milliseconds
 - If it does not finish in 8 milliseconds, the process is moved to queue Q_1
- At Q_1 job is again served in RR and receives 16 additional milliseconds
 - If it still does not complete, it is preempted and moved to queue Q_2





Operating Systems

Synchronization Tools-Part1

Seyyed Ahmad Javadi

sajavadi@aut.ac.ir

Fall 2021

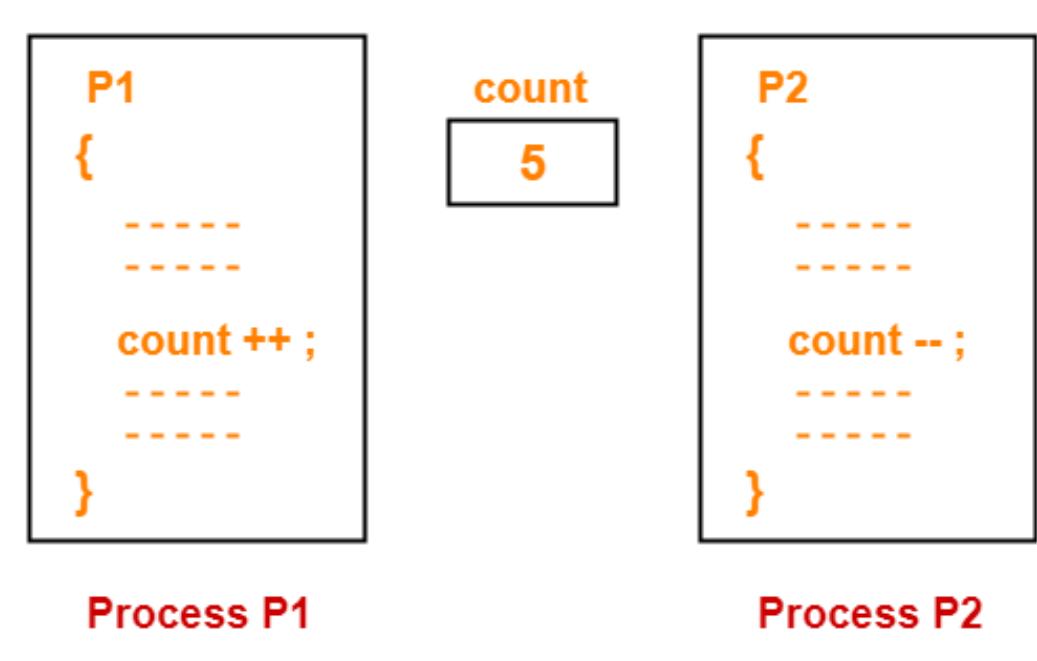
Background

- **Processes can execute concurrently**
 - May be interrupted at any time, partially completing execution.
- Concurrent access to shared data may result in **data inconsistency**.
- Maintaining data consistency requires mechanisms to ensure the **orderly execution of cooperating processes**.



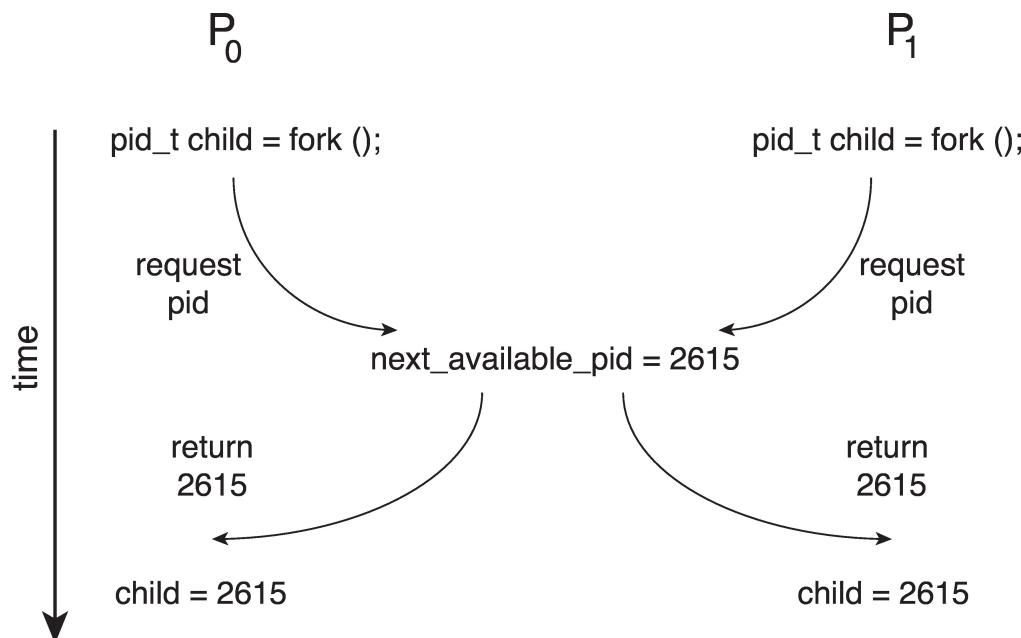
Background (cont.)

- In chapter 4, when we considered the Bounded Buffer problem with use of a counter that is updated concurrently by the producer and consumer, which **lead to race condition**.

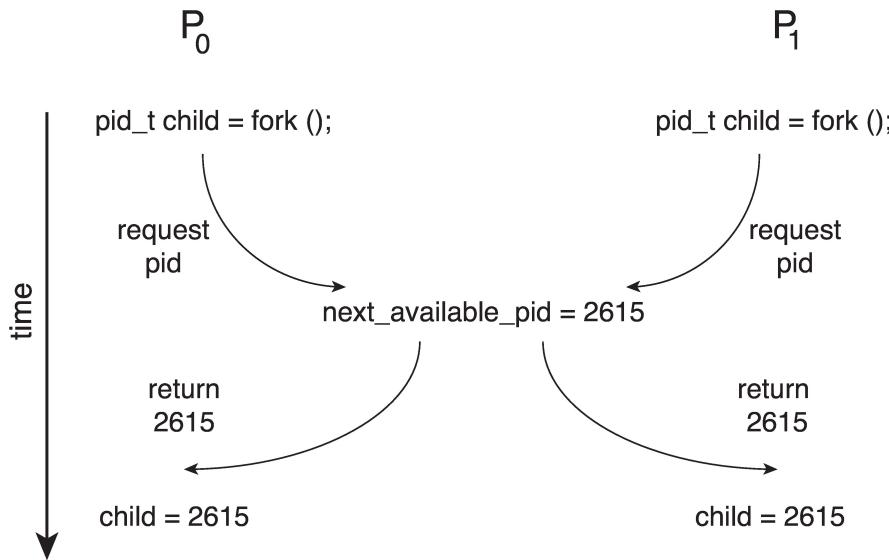


Race Condition

- Processes P_0 and P_1 are creating child processes using the **fork()** system call.
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)



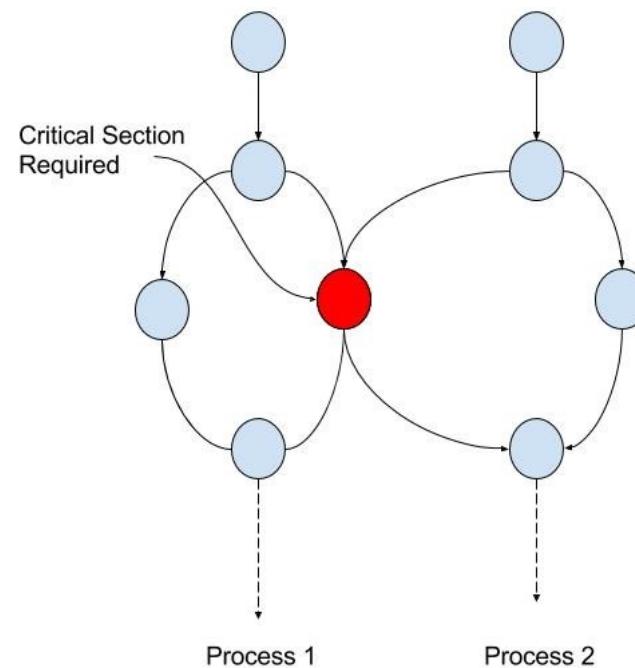
Race Condition (Cont.)



- Unless there is a mechanism to prevent P_0 and P_1 from accessing the variable `next_available_pid` **the same pid** could be assigned **to two different processes!**

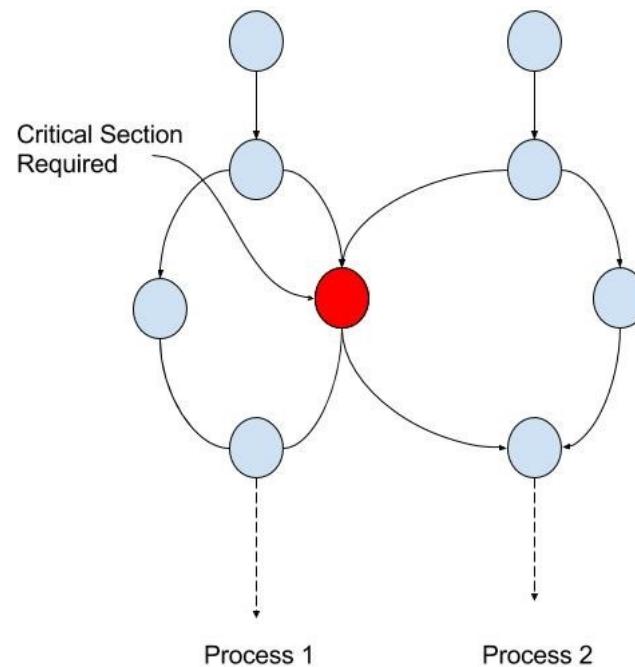
Critical Section Problem

- 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.



Critical Section Problem

- When one process in critical section, no other may be in its critical section.
- Critical section problem is to design protocol to solve this.



Critical Section

- Each process
 - must ask permission to enter critical section in **entry section**,
 - may follow critical section with **exit section**,
 - then **remainder section**.
 - General structure of process P_i
- ```
do {
 entry section
 critical section
 exit section
} while (true);
remainder section
```

# Requirements for solution to critical-section problem

---

**1. Mutual Exclusion**

**2. Progress**

**3. Bounded Waiting**

# 1- Mutual Exclusion

---

- If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections.
- **No two processes simultaneously in critical region.**



## 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 process that will enter the critical section next cannot be postponed indefinitely.
- **No process running outside its critical region may block another process.**



## 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
- **No process must wait forever to enter its critical region.**





# **Operating Systems**

## **Synchronization Tools-Part2**

Seyyed Ahmad Javadi

[sajavadi@aut.ac.ir](mailto:sajavadi@aut.ac.ir)

Fall 2021

# Interrupt-based Solution

---

- Entry section: disable interrupts
- Exit section: enable interrupts
- Will this solve the problem?
  - What if the critical section is code that runs for an hour?
    - ▶ Can some processes starve -- never enter their critical section.
  - What if there are two CPUs?

# Software Solution 1

---

- Two process solution.
- Assume that the load and store machine-language instructions are atomic; that is, cannot be interrupted.
- The two processes share one variable:
  - int *turn*;
  - *turn* indicates whose turn it is to enter the critical section.



# Algorithm for Process $P_i$

---

```
while (true) {
 while (turn == j);
 /* critical section */
 turn = j;
 /* remainder section */
}
```

# Algorithm for P<sub>0</sub> and P<sub>1</sub>

---

Initially turn = 0

```
while (TRUE) {
 while (turn != 0) /* loop */;
 critical_region();
 turn = 1;
 noncritical_region();
}
```

(a)

(a) Process 0.

```
while (TRUE) {
 while (turn != 1) /* loop */;
 critical_region();
 turn = 0;
 noncritical_region();
}
```

(b)

(b) Process 1.

# Correctness of the Software Solution

---

- Mutual exclusion is preserved
  - $P_i$  enters critical section only if:  
**turn = i**
  - turn cannot be both 0 and 1 at the same time
- It **wastes** CPU time
  - So we should avoid busy waiting as much as we can.
- Can be used only when the waiting period is expected to be **short**.



# Correctness of the Software Solution (cont.)

---

- However there is a problem in the above approach!
  - What about the **Progress requirement?**
  - What about the **Bounded-waiting requirement?**

# Correctness of the Software Solution (cont.)

- $P_0$  leaves its critical region and sets turn to 1, enters its non-critical region.
- $P_1$  enters its critical region, sets turn to 0 and leaves its critical region.
- $P_1$  enters its non-critical region, quickly finishes its job and goes back to the while loop.
- Since turn is 0, process 1 **has to wait** for process 0 to finish its non-critical region so that it can enter its critical region.
- This violates the **second condition (progress)** of providing mutual exclusion.

$P_0$

Initially turn = 0

```
while (TRUE) {
 while (turn != 0)
 critical_region();
 turn = 1;
 noncritical_region();
}
```

```
while (TRUE) {
 while (turn != 1)
 critical_region();
 turn = 0;
 noncritical_region();
}
```

$P_1$

# How About this solution?

---

```
//Algorithm for Pi
while (true) {
```

if two processes arrive at the same time to this line, who runs this line

```
 turn = i;
 while (turn == j);
```

```
/* critical section */
```

this approach doesn't have bounded wa

```
 turn = j;
```

```
/* remainder section */
```

```
}
```



# Peterson's Solution

---

- The previous solution solves the problem of one process blocking another process while its outside its critical section.
- Peterson's Solution is a neat solution with busy waiting, that defines the procedures for entering and leaving the critical region.



# Peterson's Solution (cont.)

---

- 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.**
  - $\text{flag}[i] = \text{true}$  implies that process  $P_i$  is ready!



# Algorithm for Process $P_i$

---

```
while (true) {
```

```
 flag[i] = true;
 turn = j;
 while (flag[j] && turn == j);
```

```
 /* critical section */
```

```
 flag[i] = false;
```

```
 /* remainder section */
```

```
}
```



# Correctness of Peterson's Solution

---

- Provable that the three CS requirement are met:

1. Mutual exclusion is preserved

$P_i$  enters CS only if:

either  $\text{flag}[j] = \text{false}$  or  $\text{turn} = i$

2. Progress requirement **is satisfied**

3. Bounded-waiting requirement **is met**

# Peterson's Solution and Modern Architecture

---

- Although useful for demonstrating an algorithm, Peterson's Solution **is not guaranteed to work on modern architectures.**
  - To improve performance, **processors and/or compilers may reorder operations that have no dependencies.**
- Understanding why it **will not work** is useful for better understanding race conditions.
- **For single-threaded this is ok as the result will always be the same.**
- For multithreaded the **reordering may produce inconsistent or unexpected results!**



# **Operating Systems**

## **Synchronization Tools-Part3**

Seyyed Ahmad Javadi

[sajavadi@aut.ac.ir](mailto:sajavadi@aut.ac.ir)

Fall 2021

# Peterson's Solution

```
//P0
while (true) {
 flag[0] = true;
 turn = 1;
 while (flag[1] && turn == 1);
 /* critical section */
 flag[0] = false;
 /* remainder section */
}
```

```
//P1
while (true) {
 flag[1] = true;
 turn = 0;
 while (flag[0] && turn == 0);
 /* critical section */
 flag[1] = false;
 /* remainder section */
}
```



# Peterson's Solution and Modern Architecture

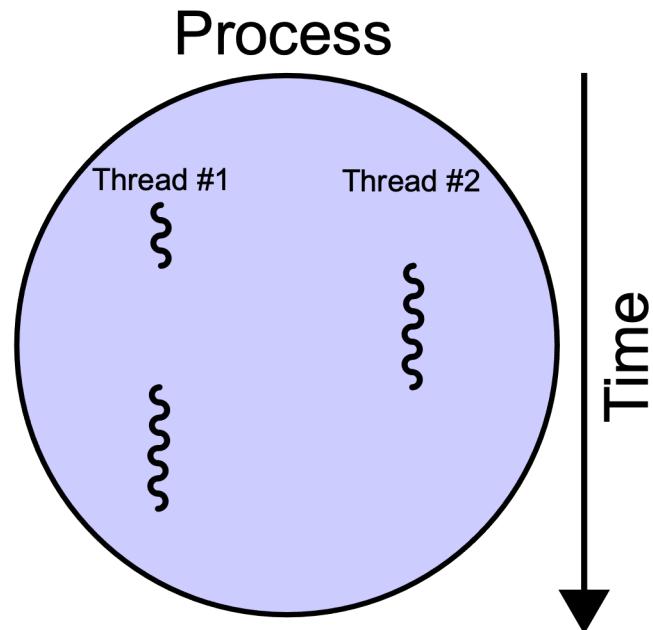
---

- Although useful for demonstrating an algorithm, Peterson's solution **is not guaranteed to work on modern architectures.**
- To improve performance, **processors and/or compilers may reorder operations that have no dependencies.**
- Understanding why it **will not work is useful for better understanding race conditions.**



# Peterson's Solution and Modern Architecture

- For **single-threaded** this is ok as the result will always be the same.
- For multithreaded the **reordering may produce inconsistent or unexpected results!**



# Modern Architecture Example

---

- Two threads share the data:

```
boolean flag = false;
int x = 0;
```

- Thread 1 performs

```
while (!flag);
 print x
```

- Thread 2 performs

```
x = 100;
flag = true
```

- What is the expected output?

100



# Modern Architecture Example (cont.)

---

- However, since the variables `flag` and `x` **are independent** of each other, the instructions:

```
flag = true;
```

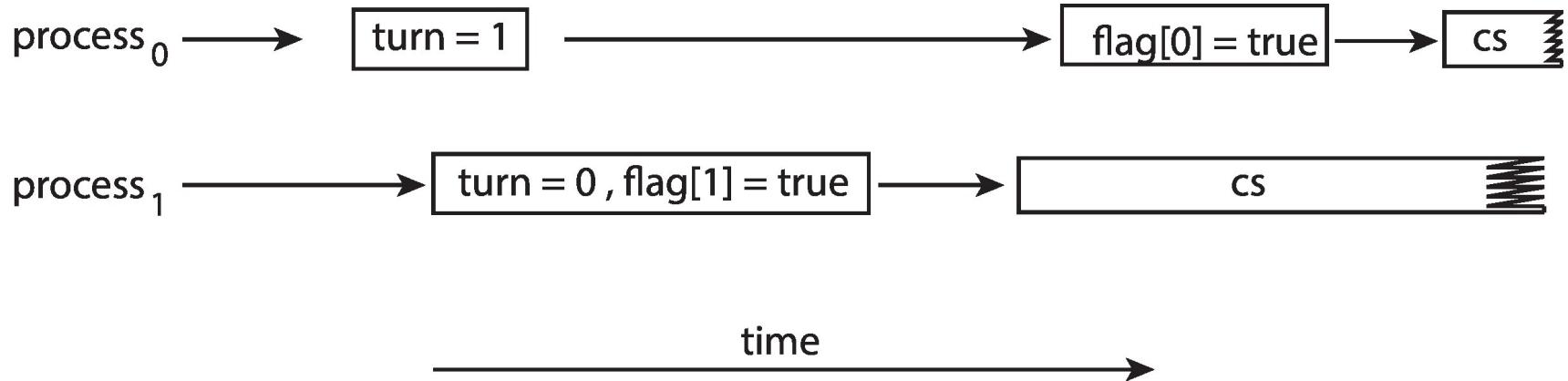
```
x = 100;
```

for Thread 2 may be reordered

- **If this occurs, the output may be 0!**

# Peterson's Solution Revisited

- The effects of instruction reordering in Peterson's Solution



- This allows both processes to be in their critical section at the same time!
- To ensure that Peterson's solution will work correctly on modern computer architecture we must use **Memory Barrier**.

# Memory Barrier

---

- **Memory model** are the memory guarantees a computer architecture makes to application programs.
- Memory models may be either:
  - **Strongly ordered** – where a memory modification of one processor is immediately visible to all other processors.
  - **Weakly ordered** – where a memory modification of one processor may not be immediately visible to all other processors.
- A **memory barrier** is an instruction that forces any change in memory to be propagated (made visible) to all other processors.



# Memory Barrier Instructions

---

- When a memory barrier instruction is performed, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed.
- Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed.



# Memory Barrier Example

---

- Returning to the example of slides 5-6
- We could add a memory barrier (as follows) to ensure Thread 1 outputs 100.
- Thread 1 now performs

```
while (!flag);
memory_barrier();
print x;
```
- Thread 2 now performs

```
x = 100;
memory_barrier();
flag = true;
```
- For Thread 1 → the value of flag is loaded before the value of x.
- For Thread 2 → the assignment to x occurs **before the assignment flag**.

# Memory Barrier for Peterson's solution

Where should we add memory barrier?

//P<sub>0</sub>

```
1. while (true) {

2. flag[0] = true;
 memory_barrier()
3. turn = 1;

4. while (flag[1] && turn == 1);

 /* critical section */

5. flag[0] = false;

 /* remainder section */

}
```

//P<sub>1</sub>

```
1. while (true) {

2. flag[1] = true;
 memory_barrier()
3. turn = 0;

4. while (flag[0] && turn == 0);

 /* critical section */

5. flag[1] = false;

 /* remainder section */

}
```

# Memory Barrier for Peterson's solution (Cont.)

---

We could place a memory barrier between the first two assignment statements in the entry section to avoid the reordering of operations shown in the previous slide.

Note that memory barriers are considered very low-level operations and are typically only used by kernel developers when writing specialized code that ensures mutual exclusion.



# Synchronization Hardware

---

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally, too inefficient on multiprocessor systems
    - ▶ Operating systems using this not broadly scalable.
- We will look at two forms of hardware support:
  1. **Hardware instructions**
  2. **Atomic variables**



# Hardware Instructions

---

- Special hardware instructions that allow us to either ***test-and-modify*** the content of a word, or two ***swap the contents of two words atomically*** (uninterruptedly.)
  - **Test-and-Set** instruction
  - **Compare-and-Swap** instruction



# The test\_and\_set Instruction

---

## ■ Definition

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

## ■ Properties

- Executed atomically
- Returns the original value of passed parameter
- Set the new value of passed parameter to **true**

# Solution Using `test_and_set()`

- Shared boolean variable `lock`, initialized to `false`

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

} while (true);
```

- Does it solve the critical-section problem?

| Requirement      | Yes/No |
|------------------|--------|
| Mutual Exclusion | yes    |
| Progress         | yes    |
| Bounded waiting  | no     |

# The compare\_and\_swap Instruction

---

## ■ Definition

```
int compare_and_swap(int *value, int expected, int new_value)
{
 int temp = *value;
 if (*value == expected)
 *value = new_value;
 return temp;
}
```

## ■ Properties

- Executed **atomically**
- Returns the original value of passed parameter value
- Set the variable value the value of the passed parameter new\_value but only if \*value == expected is true.

# Solution using compare\_and\_swap

- Shared integer **lock** initialized to 0;

```
while (true) {
 while(compare_and_swap(&lock, 0, 1) != 0); /*do nothing*/
 /* critical section */
 lock = 0;
 /* remainder section */
}
```

- Does it solve the critical-section problem?

| Requirement      | Yes/No                       |
|------------------|------------------------------|
| Mutual Exclusion | <input type="checkbox"/> yes |
| Progress         | <input type="checkbox"/> yes |
| Bounded waiting  | <input type="checkbox"/> no  |



# **Operating Systems**

## **Synchronization Tools-Part4**

Seyyed Ahmad Javadi

[sajavadi@aut.ac.ir](mailto:sajavadi@aut.ac.ir)

Fall 2021

# Second solution using compare-and-swap

---

- The common data structures are:

```
boolean waiting[n];
```

```
int lock;
```

- The elements in the ***waiting*** array are initialized to ***false***
- Variable ***lock*** is initialized to ***0***.

## Second solution using compare-and-swap

---

```
while (true) {
 waiting[i] = true;
 key = 1;
 while (waiting[i] && key == 1)
 key = compare_and_swap(&lock, 0, 1);
 waiting[i] = false;
 /* critical section */
 ...
}
```



## Second solution using compare-and-swap

```
while (true) {
 ...
 j = (i + 1) % n;
 while ((j != i) && !waiting[j])
 j = (j + 1) % n;
 if (j == i)
 lock = 0;
 else
 waiting[j] = false;
 /* remainder section */
}
```

| Requirement      | Yes/No? |
|------------------|---------|
| Mutual Exclusion | yes     |
| Progress         | yes     |
| Bounded waiting  | yes     |



# Synchronization Hardware Support

---

## ■ Hardware instructions

- `test_and_set()`
- `Compare_and_swap()`

## ■ Atomic variables

- We unfortunately do not have enough time to cover this
- Please read the related section in the reference book

# Mutex Locks

---

- *Previous solutions are complicated* and generally inaccessible to application programmers.
- OS designers build software tools to solve critical section problem
- Simplest is **mutex** lock
  - Boolean variable indicating if lock is available or not
  - In fact, the term mutex is short for mutual exclusion.



# Mutex Locks

---

- Protect a critical section by

- First **acquire()** a lock
  - Then **release()** the lock

```
acquire() {
 while (!available)
 ; /* busy wait */
 available = false;
}

release() {
 available = true;
}
```

- Calls to acquire() and release() must be **atomic**

- Usually implemented via hardware atomic instructions such as compare-and-swap.



# Solution to CS Problem Using Mutex Locks

```
while (true) {
 acquire lock
 critical section
 release lock
 remainder section
}
```

| Requirement      | Yes/No                                  |
|------------------|-----------------------------------------|
| Mutual Exclusion | <input checked="" type="checkbox"/> yes |
| Progress         | <input checked="" type="checkbox"/> yes |
| Bounded waiting  | <input type="checkbox"/> no             |

- But this solution requires **busy waiting**.
  - This lock therefore called a **spinlock**.
  - Is this a disadvantage all the time?

# Busy Waiting: Advantage or Disadvantage?

---

- Advantage of Spinlocks: no context switch is required
  - When a process must wait on a lock
  - A context switch may take considerable time

in multicore it is no problem to have spinlocks, and they can sometimes be advantageous (if they are short enough with compare to context switch)

- When we prefer **spinlocks (on multi core systems)?**
  - If a lock is to be held **for a short duration**
  - One thread can “spin” on one processing core while another thread performs its critical section on another core.

# Busy Waiting: Advantage or Disadvantage?

On modern multicore computing systems, spinlocks are  
*widely used* in many operating systems.

# Semaphore

---

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes 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()`

# Semaphore (Cont.)

---

- Definition of the wait() operation

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

- Definition of the signal() operation

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



# Semaphore (Cont.)

---

- 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 implement a counting semaphore S as a binary semaphore.
- **With semaphores we can solve various synchronization problems.**



# Semaphore Usage Example

- Solution to the CS Problem

- Create a semaphore “**mutex**” initialized to 1

```
wait(mutex);
```

CS

```
signal(mutex);
```

| Requirement      | Yes/No                                  |
|------------------|-----------------------------------------|
| Mutual Exclusion | <input type="checkbox"/> yes            |
| Progress         | <input checked="" type="checkbox"/> yes |
| Bounded waiting  | <input type="checkbox"/> no             |



# Semaphore Usage Example (Cont.)

---

- Consider  $P_1$  and  $P_2$  that with two statements  $S_1$  and  $S_2$  and the requirement that  $S_1$  to happen before  $S_2$ 
  - Create a semaphore “synch” initialized to 0

P1 :

```
S1 ;
 signal(synch) ;
```

P2 :

```
wait(synch) ;

S2 ;
```

# 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.



# Semaphore Implementation

---

- Could now have **busy waiting** in critical section implementation
  - But implementation code is short
  - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore **this is not a good solution.**

# Semaphore Implementation with no Busy waiting

---

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

# Semaphore Implementation with no Busy waiting

---

- Two operations:
  - **Block**
    - ▶ Place the process invoking the operation on the appropriate waiting queue
  - **Wakeup**
    - ▶ Remove one of processes in the waiting queue and place it in the ready queue

# Implementation with no Busy waiting (cont.)

---

- Waiting queue

```
typedef struct {
 int value;
 struct process *list;
} semaphore;
```

# Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
 S->value--;

 if (S->value < 0) {
 add this process to S->list;

 block();

 }

}

signal(semaphore *S) {
 S->value++;

 if (S->value <= 0) {
 remove a process P from S->list;

 wakeup(P);

 }

}
```

| S->value | => number of all processes that are blocked and are in semaphore list.



# Problems with Semaphores

---

- Incorrect use of semaphore operations:
  - **signal (mutex) ... wait (mutex)**
  - **wait (mutex) ... wait (mutex)**
  - Omitting of **wait (mutex)** and/or **signal (mutex)**
- These – and others – are examples of what can occur when semaphores and other synchronization tools are used incorrectly.

# Summary of What Are Not Covered

---

- **Monitors**
- **Condition Variables**
- **We also skip chapter 7 slides**
  - <https://www.os-book.com/OS10/slides-dir/index.html>



# **Operating Systems**

## **Deadlocks-Part1**

Seyyed Ahmad Javadi

[sajavadi@aut.ac.ir](mailto:sajavadi@aut.ac.ir)

Spring 2021

# Outline

---

- Liveness
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance

# Chapter Objectives

---

- Illustrate how deadlock can occur when mutex locks are used
- Define the four necessary conditions that characterize deadlock
- Identify a deadlock situation in a resource allocation graph
- Evaluate the four different approaches for preventing deadlocks
- Apply the banker's algorithm for deadlock avoidance

# Liveness

---

- Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore.
- Waiting indefinitely violates the progress and bounded-waiting criteria.

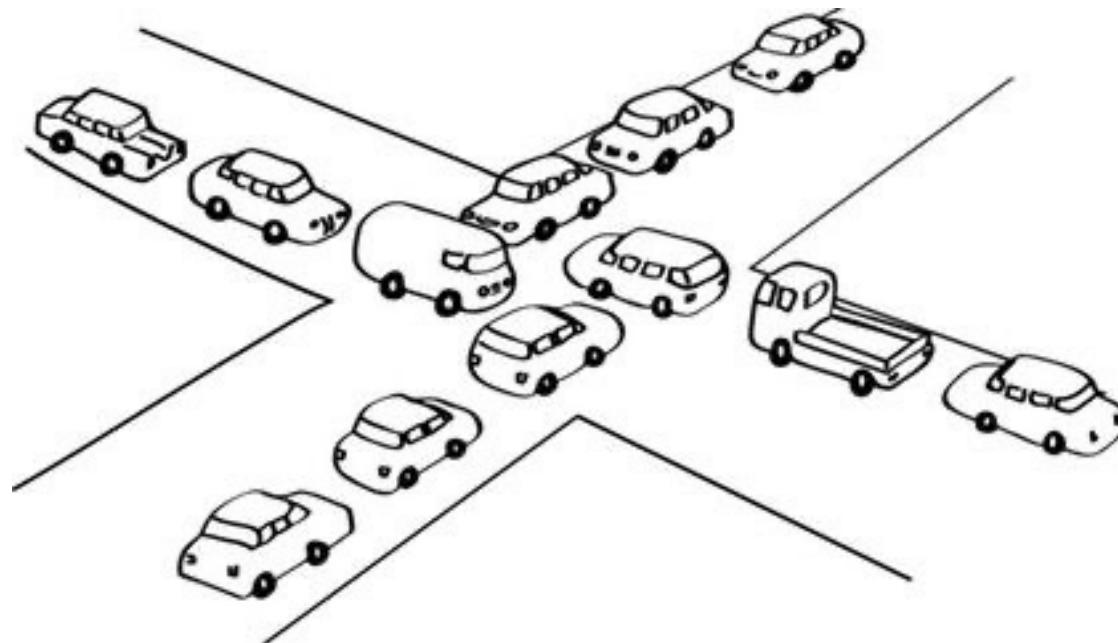
# Liveness

---

- **Liveness** refers to a set of properties that a system must satisfy to ensure processes make progress.
- **Indefinite waiting** is an example of a **liveness failure**.

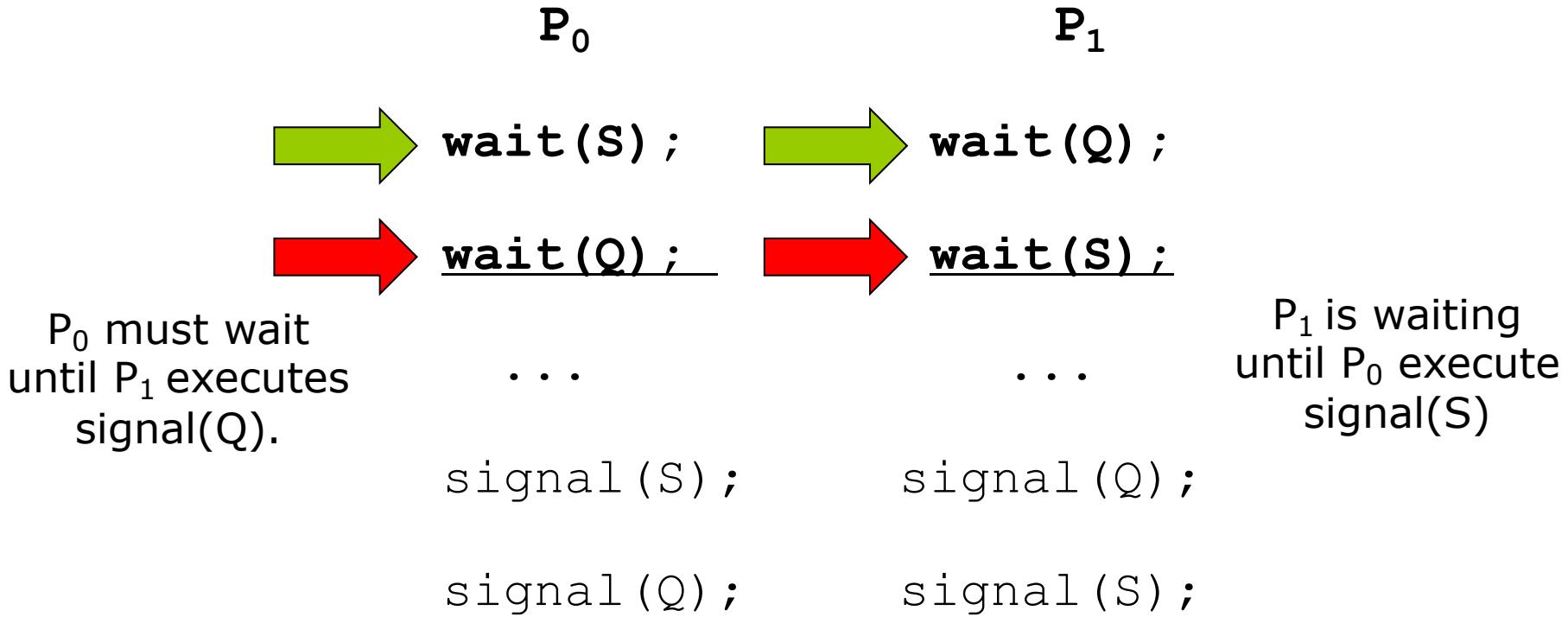
# Deadlock

two or more processes are **waiting indefinitely** for an event  
that can be caused by **only one of the waiting processes.**



# Liveness (cont.)

- Let S and Q be two semaphores initialized to 1



Since these `signal()` operations will never be executed,  $P_0$  and  $P_1$  are **deadlocked**.

# Other Forms of Deadlock

---

- **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.
  - We do not cover this.

# 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 with Semaphores

---

- Data:
  - A semaphore **S1** initialized to 1
  - A semaphore **S2** initialized to 1
- Two processes P1 and P2
- **P1:**

```
wait(s1)
wait(s2)
```
- **P2:**

```
wait(s2)
wait(s1)
```

# Deadlock Characterization

---

**Deadlock can arise if four conditions hold simultaneously.**

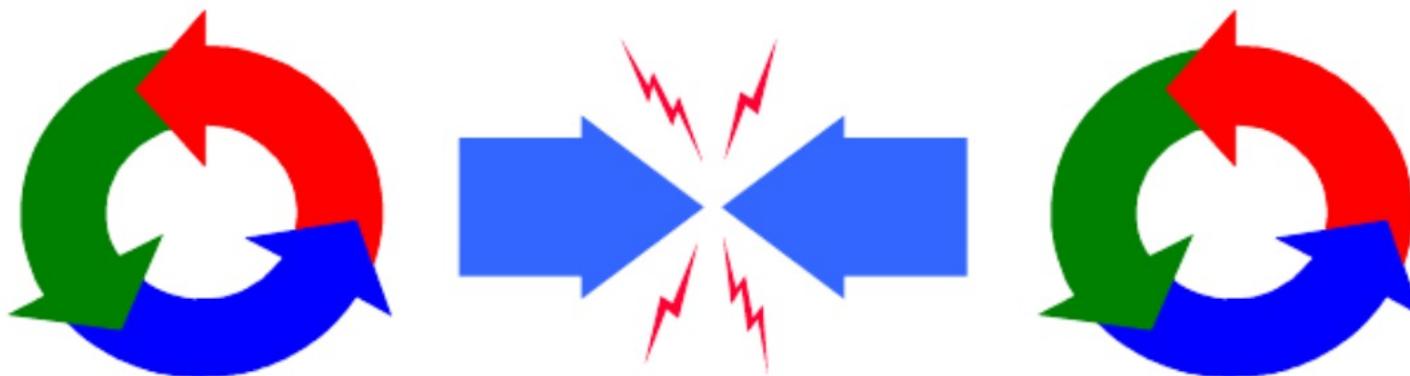
- 1. Mutual exclusion**
- 2. Hold and wait**
- 3. No preemption**
- 4. Circular wait**

all of them should be met together.

# 1-Mutual Exclusion

---

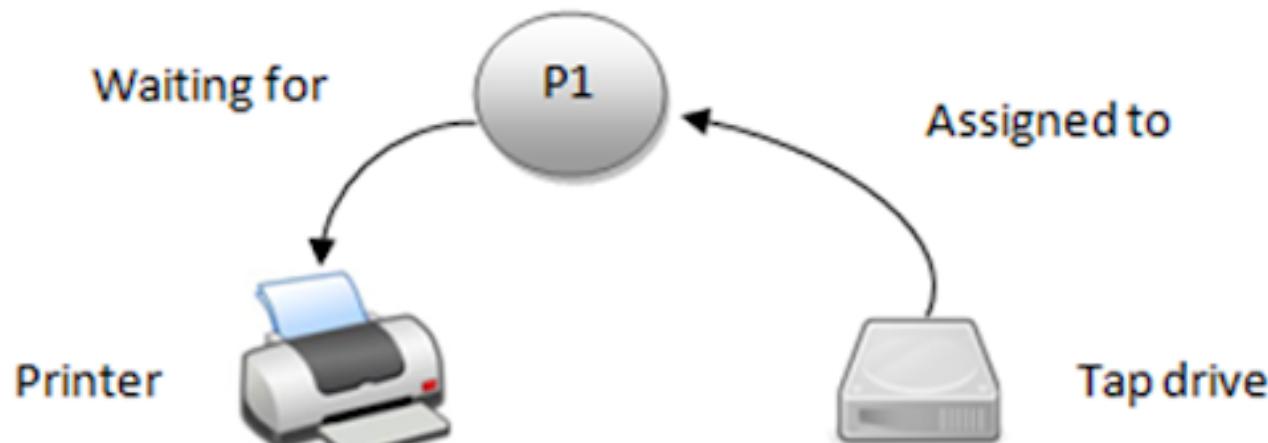
- Only one process at a time can use a resource.



# 2-Hold and Wait

---

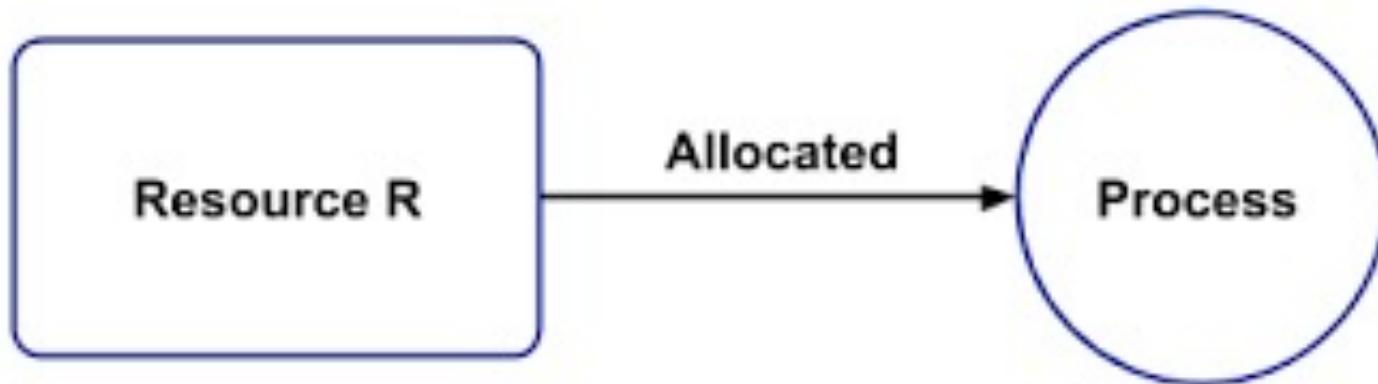
- A process holding at least one resource is waiting to acquire additional resources held by other processes.



# 3-No Preemption

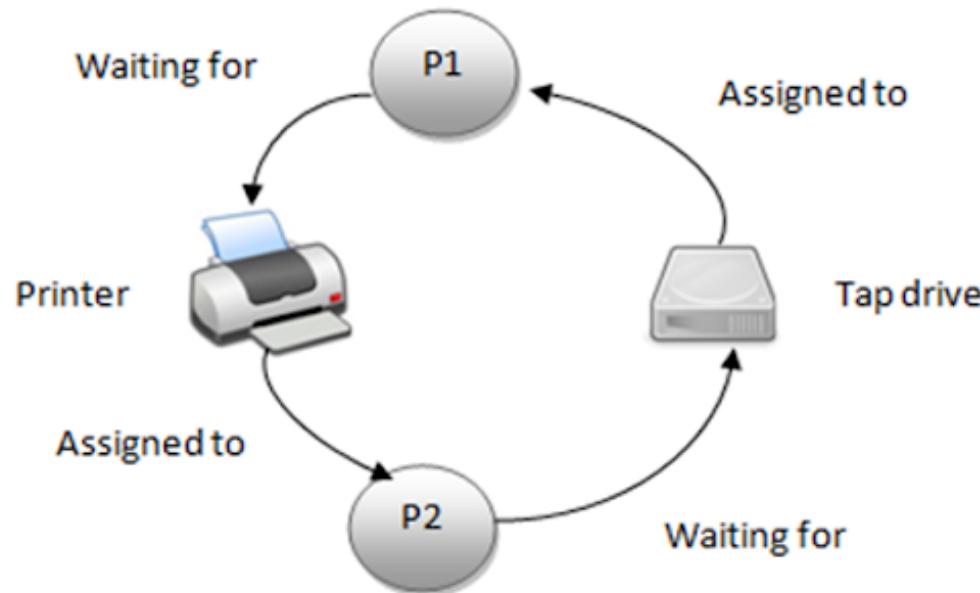
---

- A resource can be released only ***voluntarily*** by the process holding it, after that process has completed its task.



# 4-Circular Wait

- There exists a set  $\{P_0, P_1, \dots, P_n\}$  of waiting processes such that:
  - $P_0$  is waiting for a resource that is held by  $P_1$ ,
  - $P_1$  is waiting for a resource that is held by  $P_2$ , ...,
  - $P_{n-1}$  is waiting for a resource that is held by  $P_n$ ,
  - and  $P_n$  is waiting for a resource that is held by  $P_0$ .



# Resource-Allocation Graph

---

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

- $V$  is partitioned into two types:

- $P = \{P_1, P_2, \dots, P_n\}$ ,
    - ▶ The set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \dots, R_m\}$ ,
    - ▶ The set consisting of all resource types in the system.

# Resource-Allocation Graph

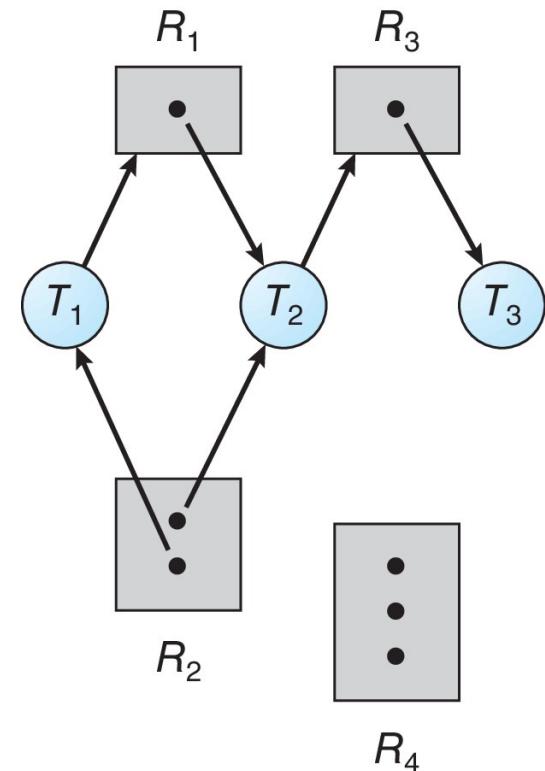
---

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

- **request edge** – directed edge  $P_i \rightarrow R_j$
- **assignment edge** – directed edge  $R_j \rightarrow P_i$

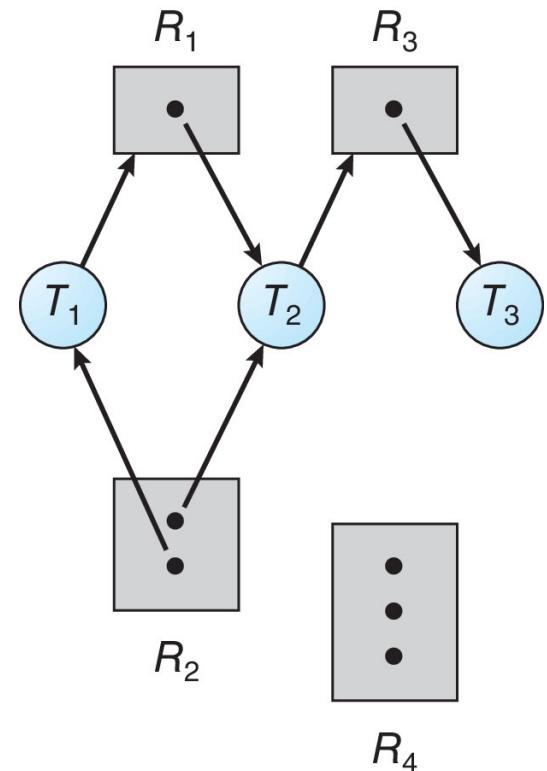
# Resource Allocation Graph Example

- One instance of R1
- Two instances of R2
- One instance of R3
- Three instances of R4

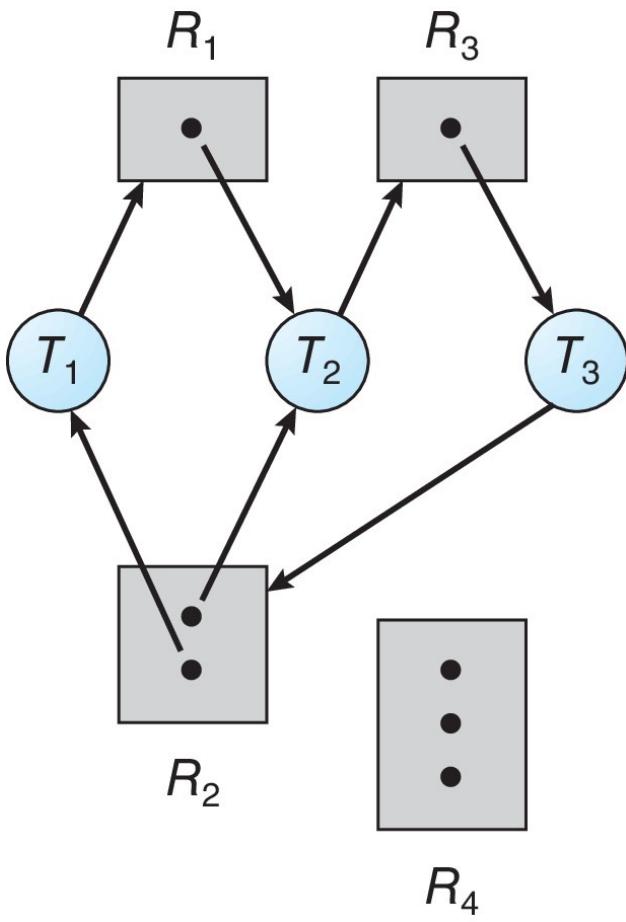


# Resource Allocation Graph Example

- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3.
- T3 is holding one instance of R3



# Resource Allocation Graph with a Deadlock



$T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$

$T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_2$

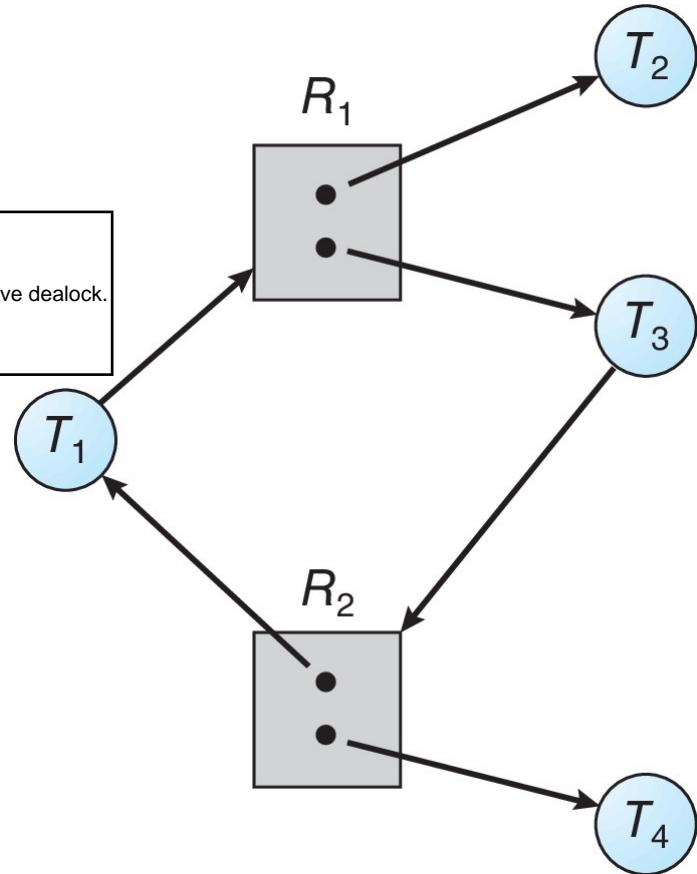
Threads  $T_1, T_2$ , and  $T_3$  are deadlocked.

- Thread  $T_2$  is waiting for the resource  $R_3$ , which is held by thread  $T_3$ .
- Thread  $T_3$  is waiting for either thread  $T_1$  or thread  $T_2$  to release resource  $R_2$ .
- In addition, thread  $T_1$  is waiting for thread  $T_2$  to release resource  $R_1$ .

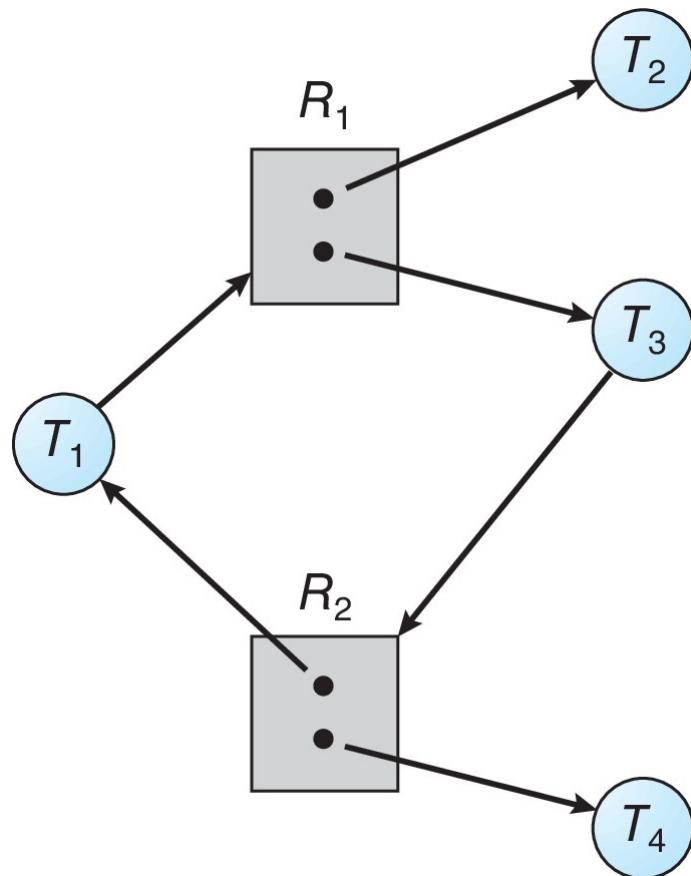
# Is there a Deadlock?

no

circular wait has been invalidated here, so we don't have deadlock.



# Graph with a Cycle But no Deadlock



$T_1 \rightarrow R_1 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$

**There is no deadlock.** Observe that thread  $T_4$  **may release** its instance of resource type  $R_2$ . That resource can then be allocated to  $T_3$ , breaking the cycle.

# Basic Facts

---

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

if the graph contains **no cycles**  $\rightarrow$  **no thread in the system is deadlocked**

If the graph **does contain a cycle**  $\rightarrow$  **a deadlock may or may not exist.**



# **Operating Systems**

## **Deadlocks-Part2**

Seyyed Ahmad Javadi

[sajavadi@aut.ac.ir](mailto:sajavadi@aut.ac.ir)

Fall 2021

# Outline

---

- Liveness
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance



# Methods for Handling Deadlocks

---

- Ensure that the system will **never** enter a deadlock state:
  - **Deadlock prevention**
  - **Deadlock avoidance**

we need processes information in this approach.
- Allow the system to enter a deadlock state and then recover.
- Ignore it and pretend that deadlocks never occur in the system.



# Deadlock Prevention

---

Invalidate ***one of the four*** necessary conditions for deadlock



# Deadlock Prevention-Mutual Exclusion

---

- Not required for sharable resources (e.g., read-only files)
- Must hold for non-sharable resources

impossible to invalidate mutual exclusion for non-sharable. we should invalidate other conditions.

# Deadlock Prevention- 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-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.



# Deadlock Prevention- Circular Wait

---

- Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.
- Invalidating the circular wait condition is most common.
- Simply assign each resource (i.e., mutex locks) a unique number.
- Resources must be acquired in order.



# Circular Wait

- If:

```
first_mutex = 1
second_mutex = 5
```

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

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

 pthread_exit(0);
}
```

unlocking is in reverse order

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

 pthread_exit(0);
}
```



# Deadlock Avoidance

---

Requires that the system has some additional  
*a priori* information available.

# Deadlock Avoidance

---

- 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  $j < i$



# Safe State (cont.)

---

- 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

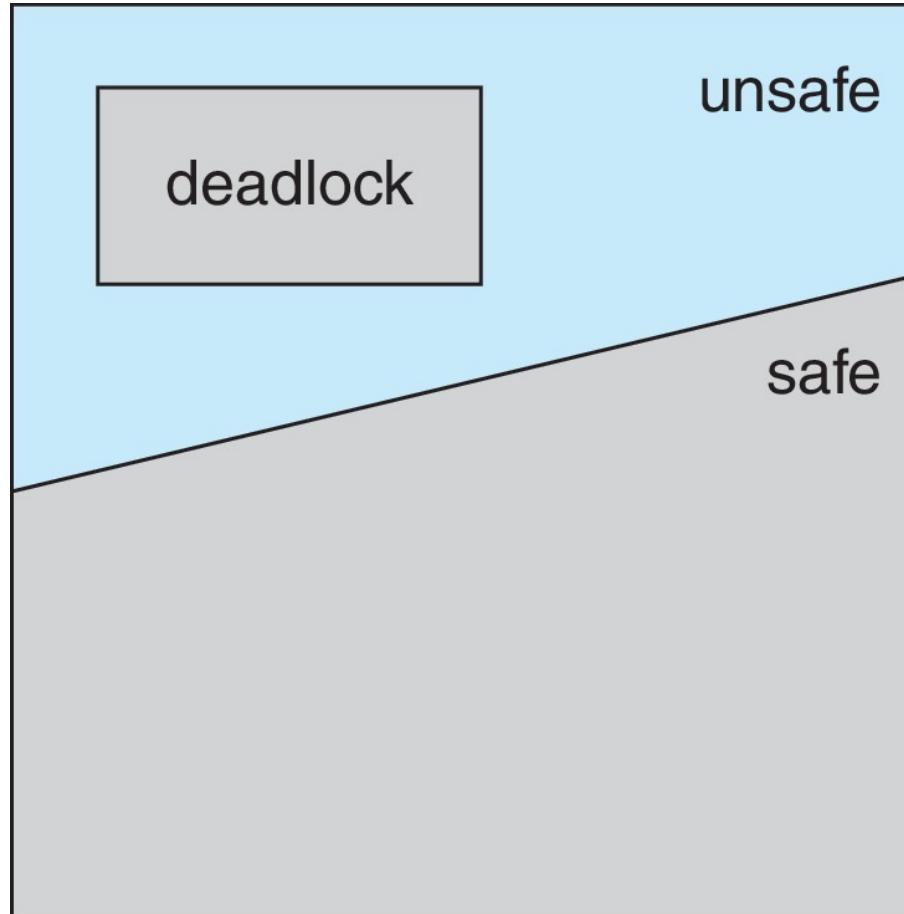
---

- 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



# 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.



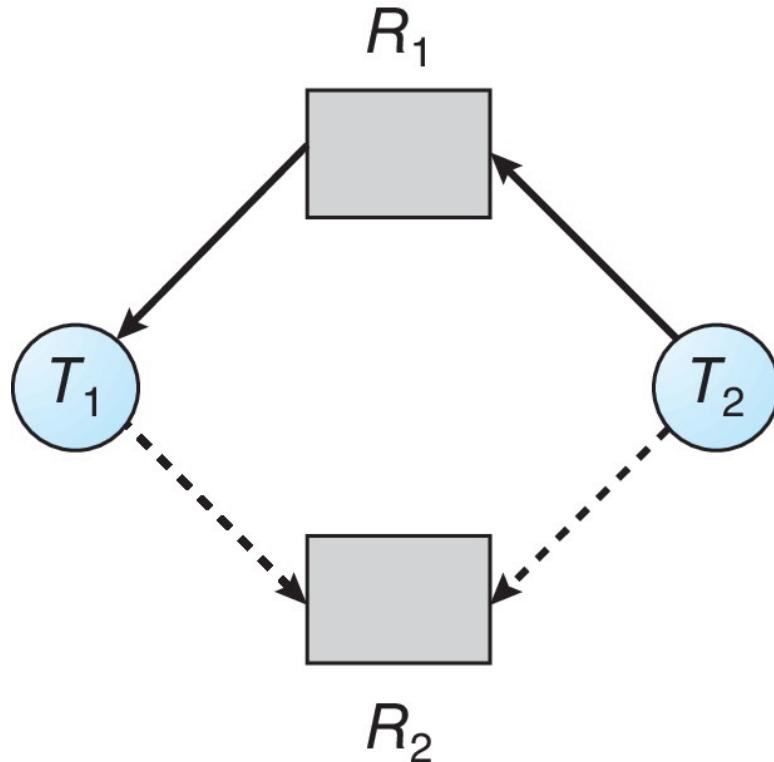
# Resource-Allocation Graph Scheme (cont.)

---

- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed a priori in the system.

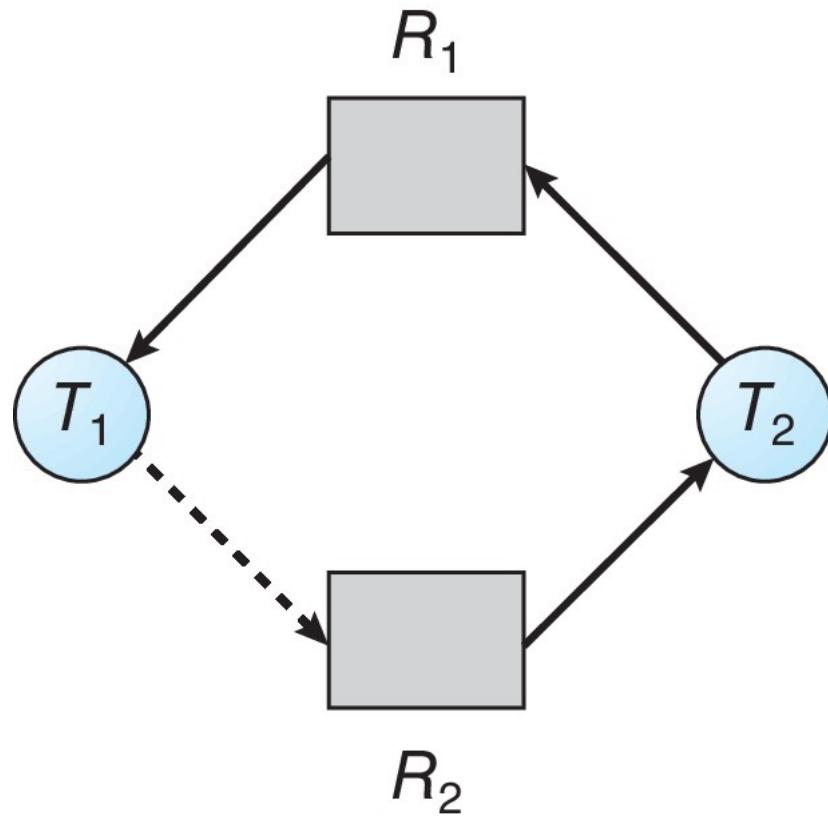
# Resource-Allocation Graph

---



# Unsafe State In Resource-Allocation Graph

---



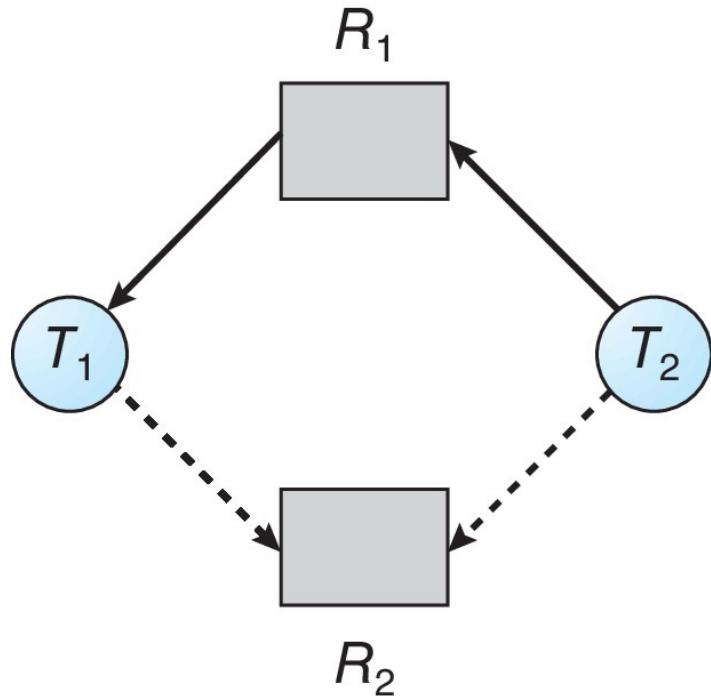
# Resource-Allocation Graph Algorithm

---

- 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.
- If a cycle is found, then the allocation will put the system in an unsafe state. In that case, thread  $T_i$  will have to wait for its requests to be satisfied.



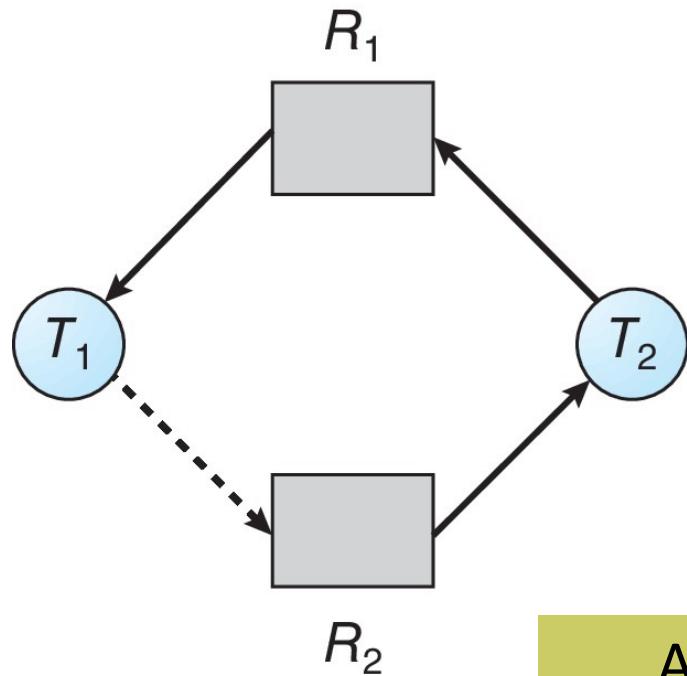
# Example of Using the Algorithm



Suppose that  $T_2$  requests  $R_2$ .

**Can the request be granted?**

# Example of Using the Algorithm (Cont.)



Suppose that  $T_2$  requests  $R_2$ .

Can the request be granted?

Although  $R_2$  is currently free, we cannot allocate it to  $T_2$ , since this action will create a cycle in the graph.

A cycle, indicates that the system is in an unsafe state. If  $T_1$  requests  $R_2$ , and  $T_2$  requests  $R_1$ , then a deadlock will occur.

# Banker's Algorithm

---

- Multiple instances of resources
- 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  $R_j$  max doesn't change during the algorithm.



# Data Structures for the Banker's Algorithm

---

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

- **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

time complexity:  $O(n \cdot n \cdot m)$  (n processes, and m resources)

1. Let  $Work$  and  $Finish$  be vectors of length  $m$  and  $n$ , respectively.

Initialize:

$Work = Available$

$Finish[i] = false$  for  $i = 0, 1, \dots, n-1$

2. Find an  $i$  such that both:

(a)  $Finish[i] = false$

(b)  $Need_i \leq Work$

If no such  $i$  exists, go to step 4

3.  $Work = Work + Allocation_i$

$Finish[i] = true$

go to step 2

4. If  $Finish[i] == true$  for all  $i$ , then the system is in a **safe state**



# Resource-Request Algorithm for Process $P_i$

---

- Algorithm determines whether requests can be safely granted.
- $\text{Request}_i = \text{request}$  vector for process  $P_i$
- If  $\text{Request}_i[j] = k$  then process  $P_i$  wants  $k$  instances of resource type  $R_j$



## Resource-Request Algorithm for Process $P_i$ (Cont.)

---

1. If  $Request_i \leq Need_i$ , go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If  $Request_i \leq Available$ , 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:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- 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> |          |          |
|-------|-------------------|----------|----------|------------|----------|----------|------------------|----------|----------|
|       | <i>A</i>          | <i>B</i> | <i>C</i> | <i>A</i>   | <i>B</i> | <i>C</i> | <i>A</i>         | <i>B</i> | <i>C</i> |
| $P_0$ | 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***

*Need*

*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</i>          | <i>B</i> | <i>C</i> | <i>A</i>    | <i>B</i> | <i>C</i> | <i>A</i>         | <i>B</i> | <i>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?  no
- Can request for (0,2,0) by  $P_0$  be granted?  yes



# **Operating Systems**

## **Main Memory-Part1**

Seyyed Ahmad Javadi

[sajavadi@aut.ac.ir](mailto:sajavadi@aut.ac.ir)

Fall 2021

# Chapter 9: Memory Management

---

- Background
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping

# Objectives

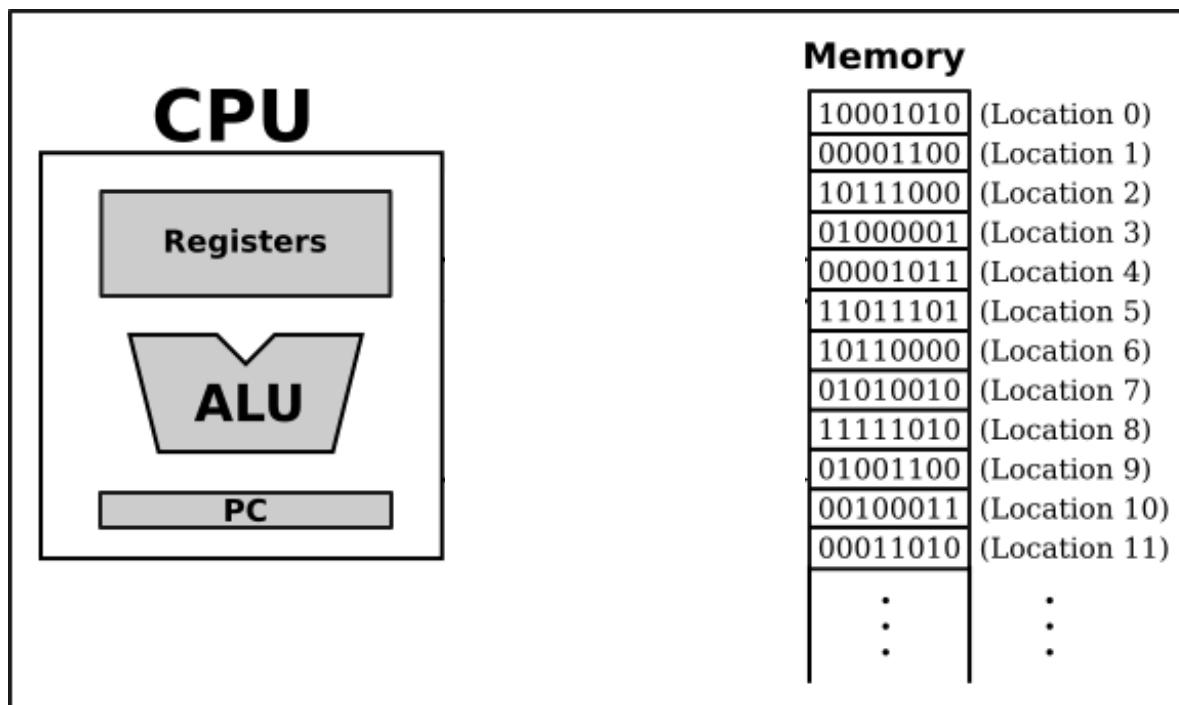
---

- To provide a detailed description of various ways of organizing memory hardware.
- To discuss various memory-management techniques.
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging.



# Background

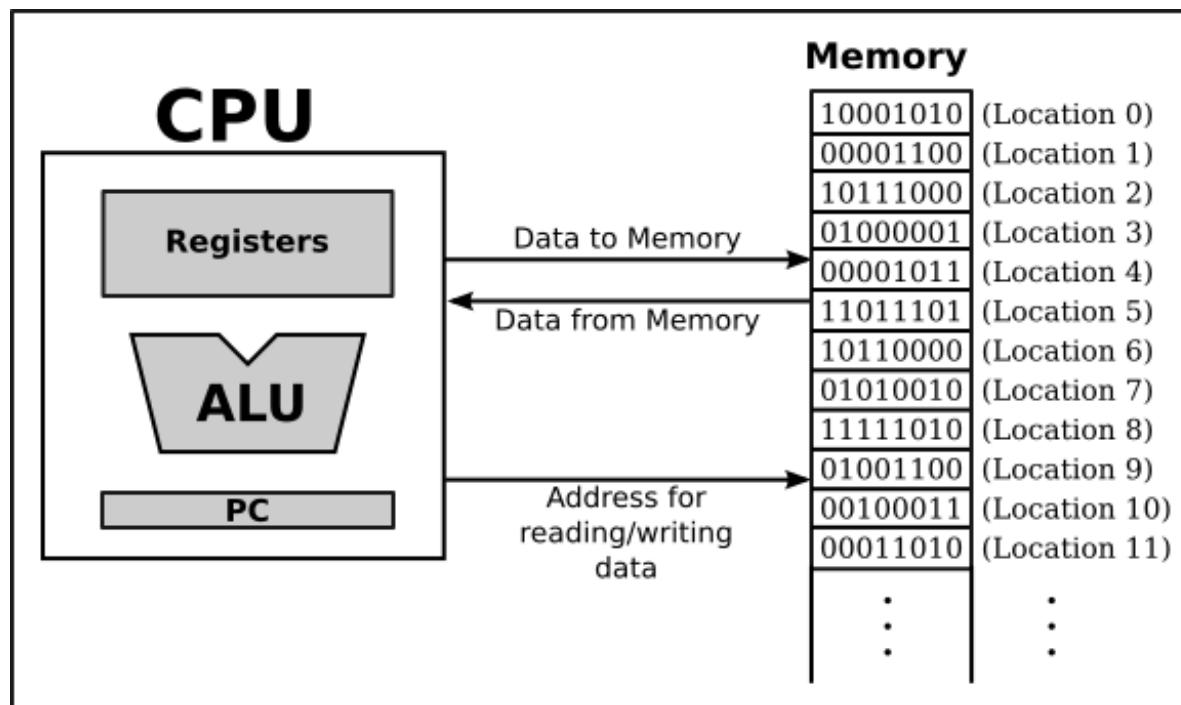
- Program must be brought (from disk) into memory and placed within a process for it to be run.
- Main memory and registers are only storage CPU can access directly.



<https://math.hws.edu/javanotes/c1/s1.html>

# Background (cont.)

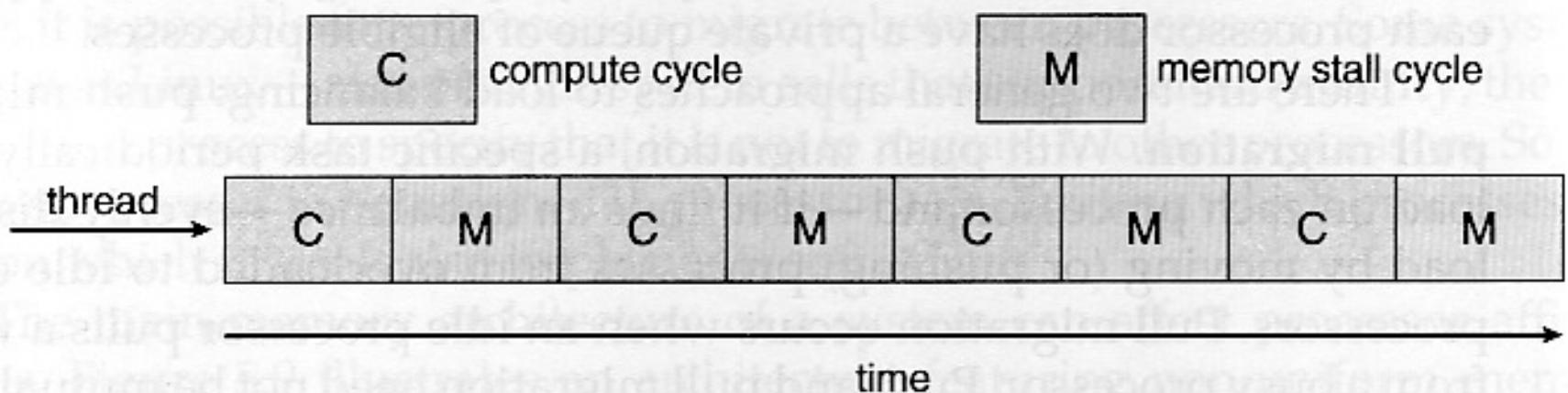
- Memory unit only sees a stream of:
  - addresses + read requests, or
  - address + data and write requests



<https://math.hws.edu/javanotes/c1/s1.html>

# Background (cont.)

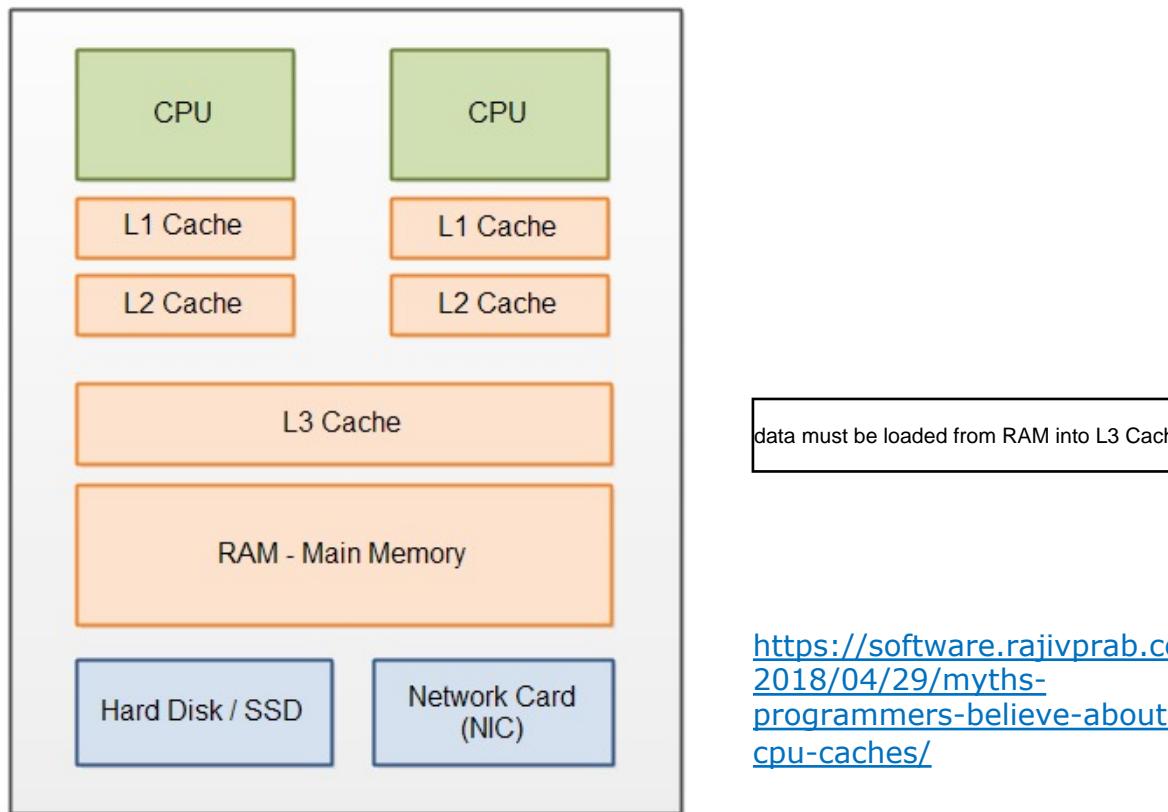
- Register access is done in one CPU clock (or less)
- Main memory can take many cycles, causing a **memory stall**



[https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/5\\_CPU\\_Scheduling.html](https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/5_CPU_Scheduling.html)

# Background (cont.)

- Cache sits between main memory and CPU registers
- Protection of memory required to ensure correct operation



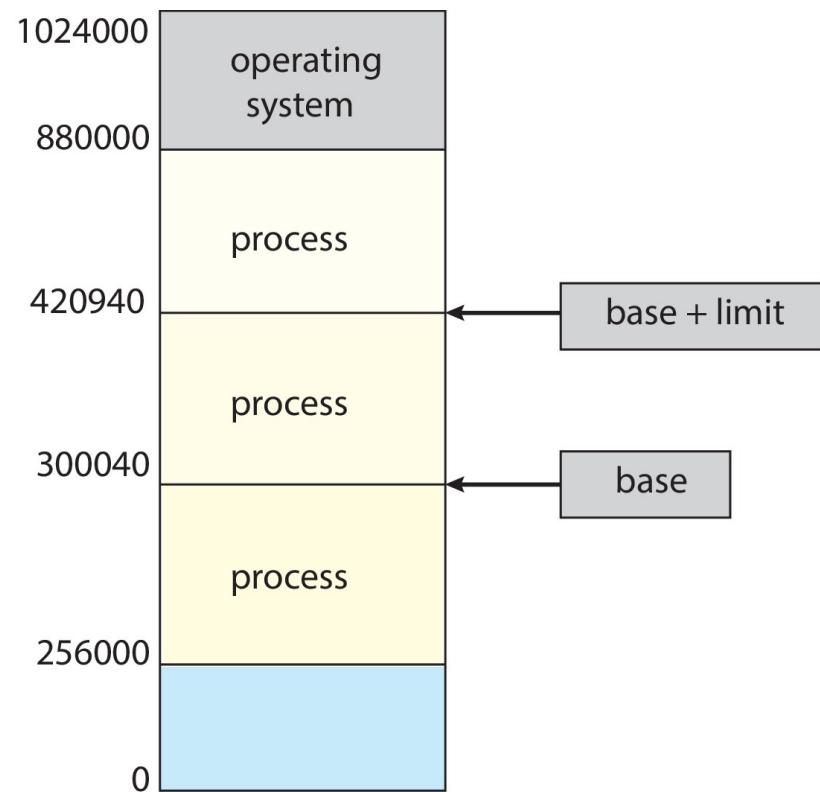
[https://software.rajivprab.com/  
2018/04/29/myths-  
programmers-believe-about-  
cpu-caches/](https://software.rajivprab.com/2018/04/29/myths-programmers-believe-about-cpu-caches/)

# Protection

- Need to ensure that a process can access only those addresses in its address space.

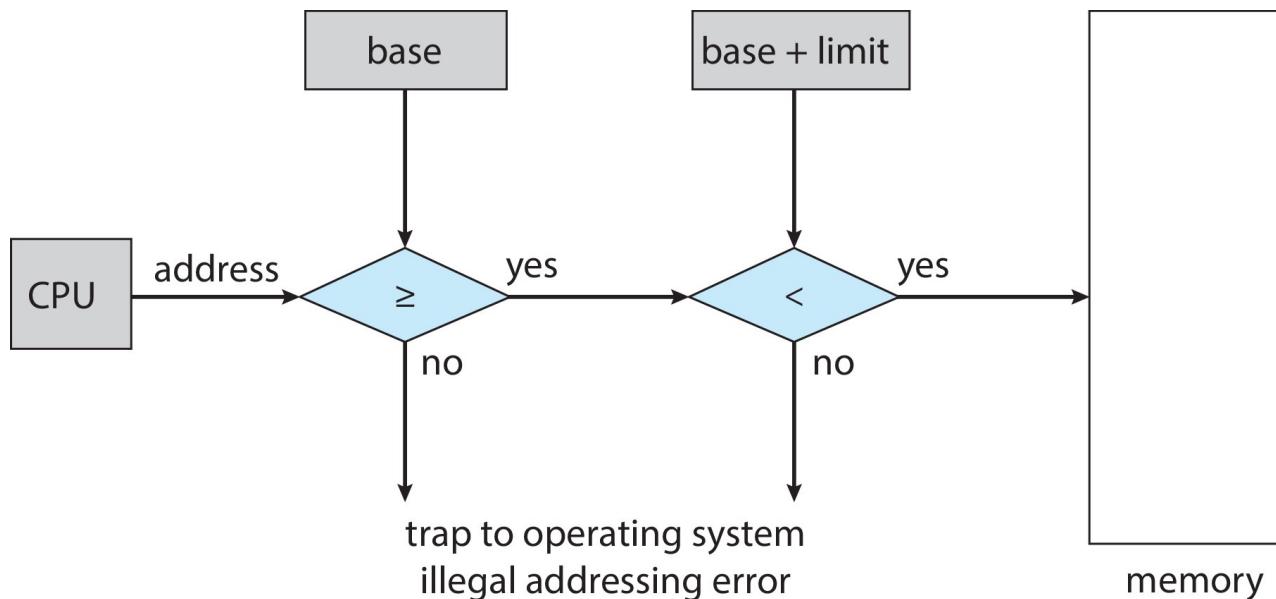
$\text{base} \leq \text{legal address} < \text{base} + \text{limit}$

- We can provide this protection by using a pair of **base** and **limit registers** to define the logical address space of a process.



# Hardware Address Protection

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



- The instructions to loading the base and limit registers are **privileged**.

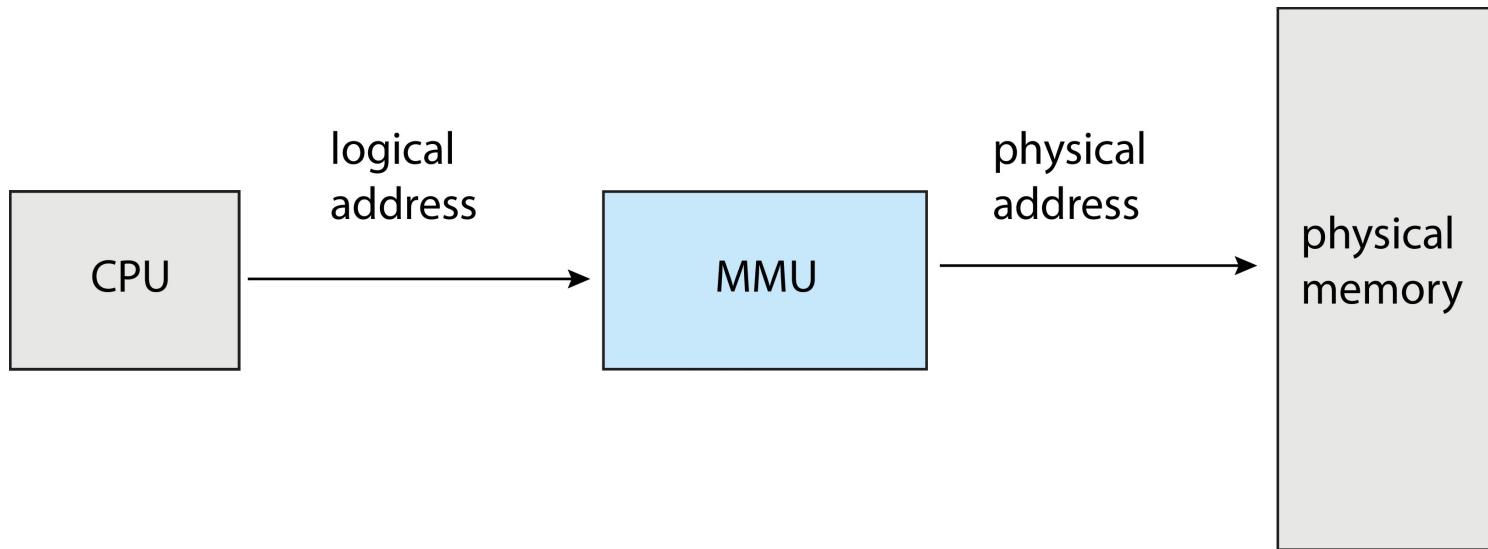
# Logical vs. Physical Address Space

---

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management.
- **Logical address**
  - Generated by the CPU
  - Also referred to as **virtual address**
- **Physical address**
  - Address seen by the memory unit

# Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address



- Many methods possible, covered in the rest of this chapter

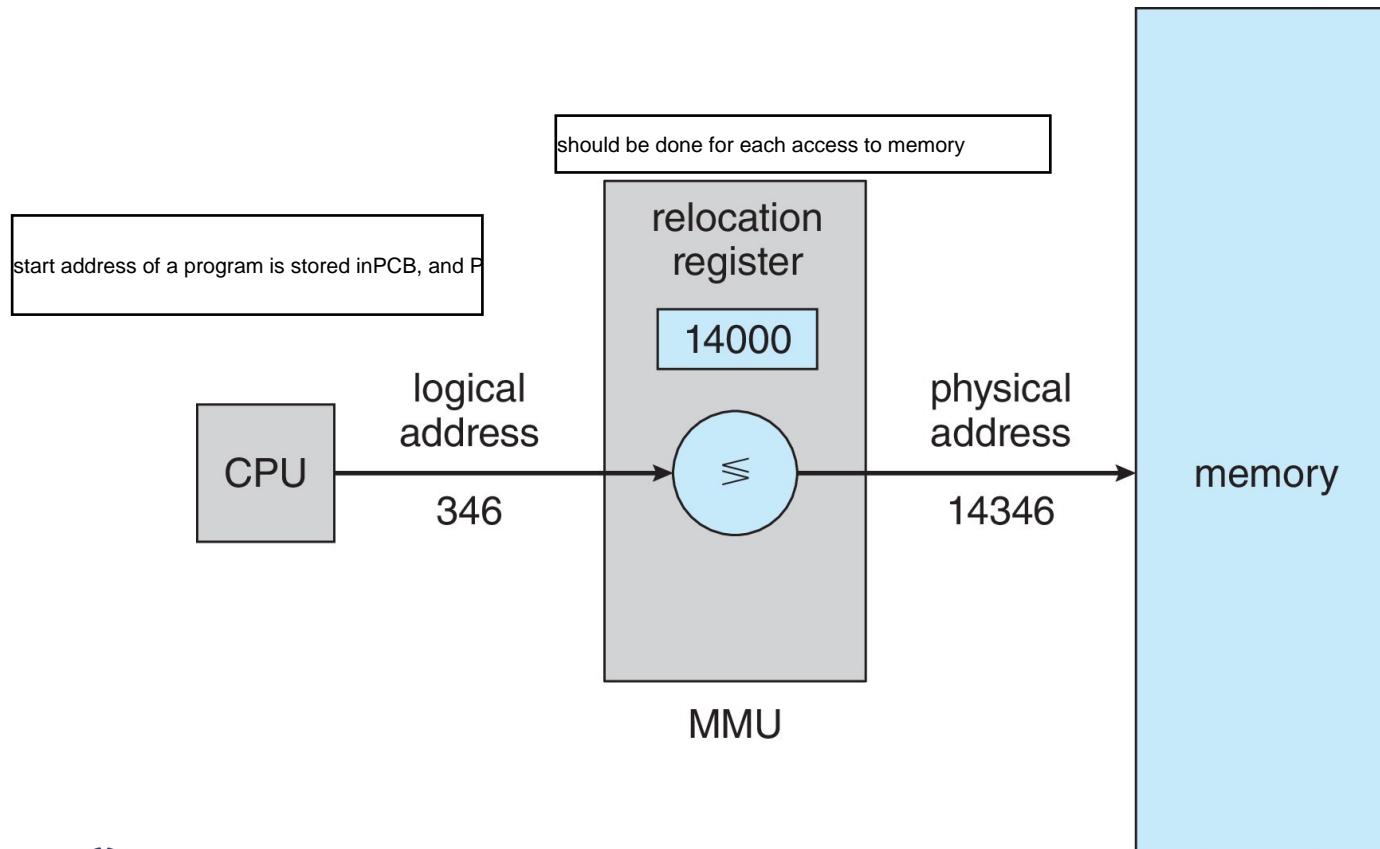
# Memory-Management Unit (Cont.)

---

- Consider simple scheme. which is a generalization of the base-register scheme.
- The base register now called **relocation register**.

# Memory-Management Unit (Cont.)

- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory.



# Memory-Management Unit (Cont.)

---

- The user program deals with logical addresses
  - It never sees the real physical addresses
- Execution-time binding occurs when reference is made to location in memory.
  - Logical address bound to physical addresses

# Contiguous 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

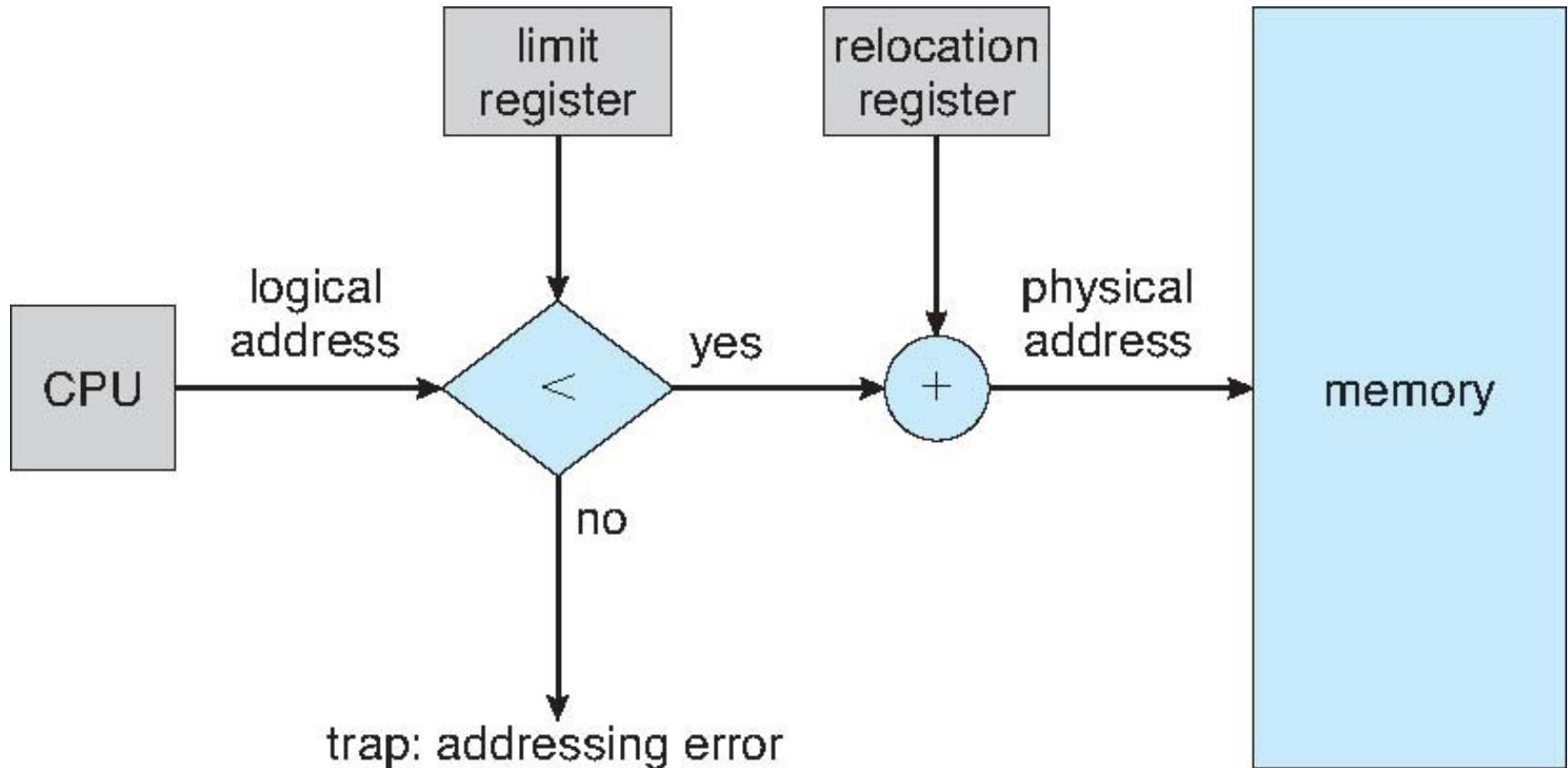


# Contiguous Allocation (cont.)

---

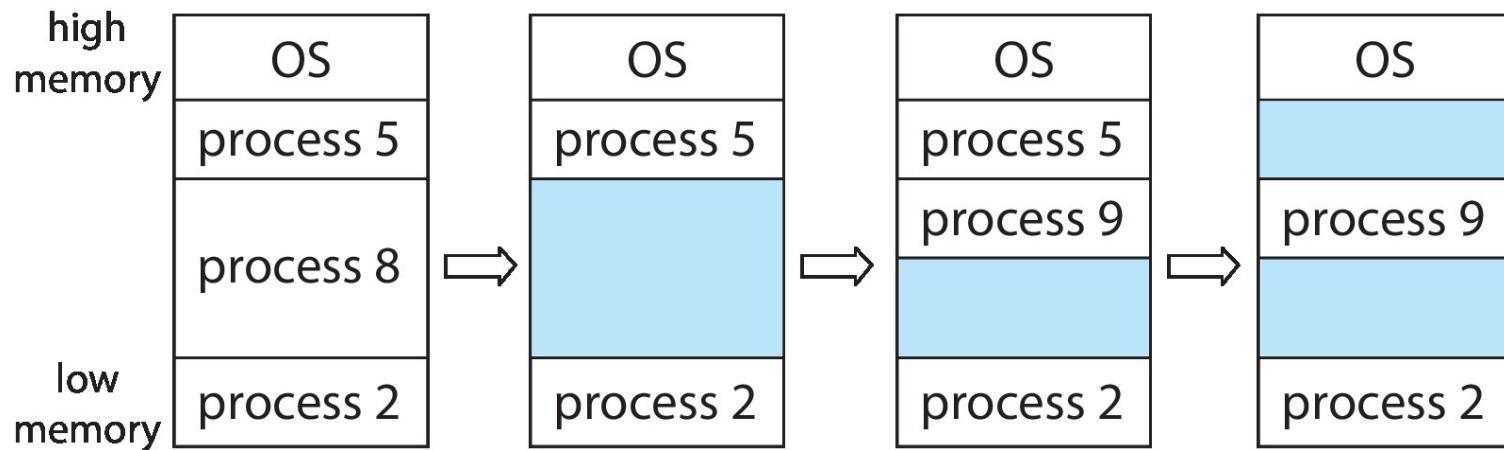
- **Relocation registers** used to protect user processes from each other, and from changing operating-system code and data
  - **Base register** contains value of smallest physical address
  - **Limit register** contains range of logical addresses – each logical address must be less than the limit register
  - **MMU** maps logical address *dynamically*

# Hardware Support for Relocation and Limit Registers



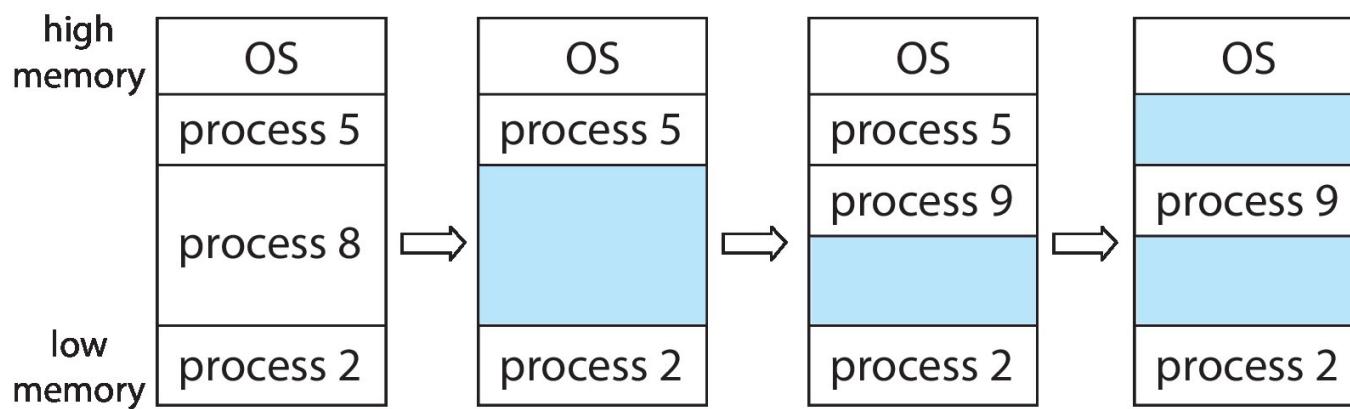
# Multiple-partition Allocation

- Degree of multiprogramming limited by number of 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



# Multiple-partition Allocation (cont.)

- 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)



# Dynamic Storage-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 lowest run time, doesn't want to search
- **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 reduces the chance of largest programs



# Dynamic Storage-Allocation Problem

---

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

memory (here we mean memory not disk)



# **Operating Systems**

## **Main Memory-Part2**

Seyyed Ahmad Javadi

[sajavadi@aut.ac.ir](mailto:sajavadi@aut.ac.ir)

Fall 2021

# Fragmentation

---

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous.

first fit, best fit and worst fit algorithms have external fragmentation but they don't have internal fragmentation

- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

paging has the problem of internal fragmentation

**Important** First fit analysis reveals that given N blocks allocated, another  $0.5*N$  blocks lost to fragmentation

- 1/3 may be unusable -> **50-percent rule**

# 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
- The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory. This scheme can be expensive.
- Now consider that backing store has same fragmentation problems.



# Paging

---

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



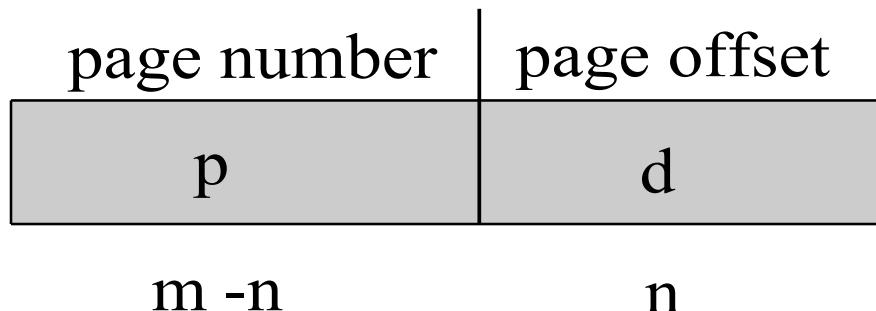
# Paging (Cont.)

---

- 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
- **Backing store** likewise split **into pages**
- **Still have Internal fragmentation**

# Address Translation Scheme

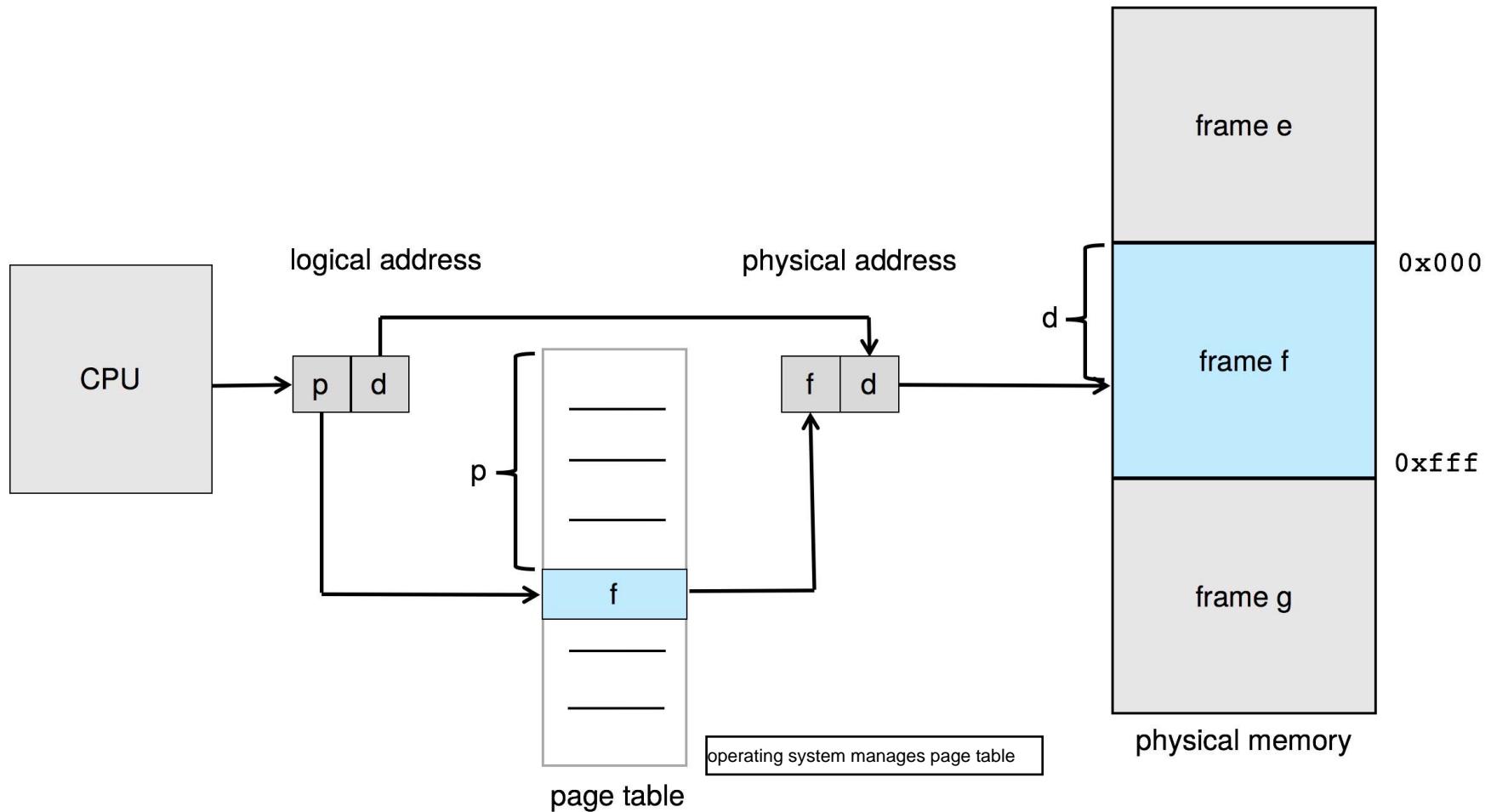
- Address generated by CPU is divided into:
  - **Page number (p)** – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit



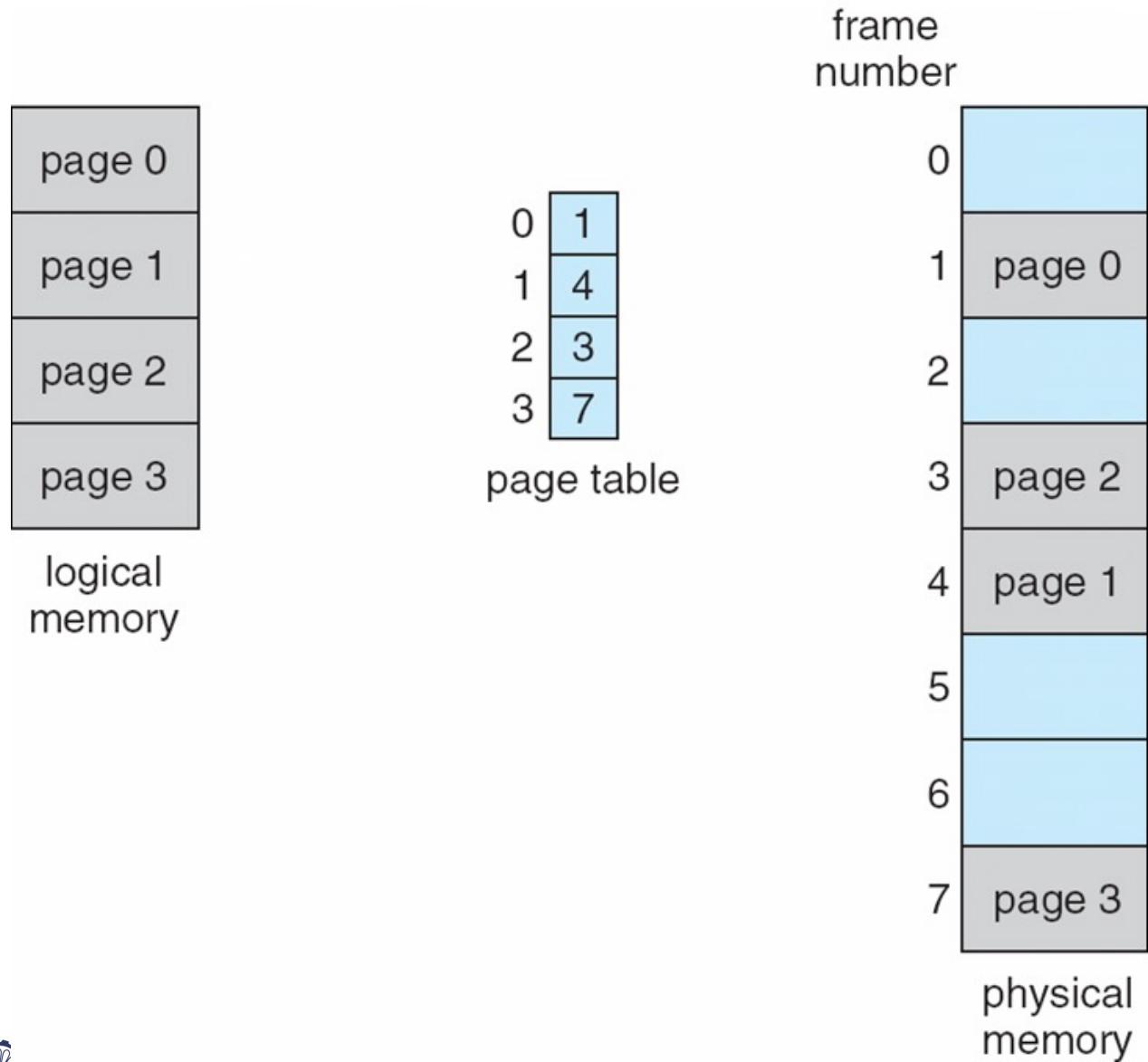
- For given logical address space  $2^m$  and page size  $2^n$

$$K = 1024 = 2^{10}M = 1024 * k$$

# Paging Hardware



# Paging Model of Logical and Physical Memory



# Paging Example

- Logical address:  $n = 2$  and  $m = 4$ . Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)

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

logical memory

|   |   |
|---|---|
| 0 | 5 |
| 1 | 6 |
| 2 | 1 |
| 3 | 2 |

page table

|    |   |
|----|---|
| 0  |   |
| 4  | i |
| 8  | j |
| 12 | k |
| 16 | l |
| 20 | m |
| 24 | n |
| 28 | o |

physical memory



## Paging -- 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

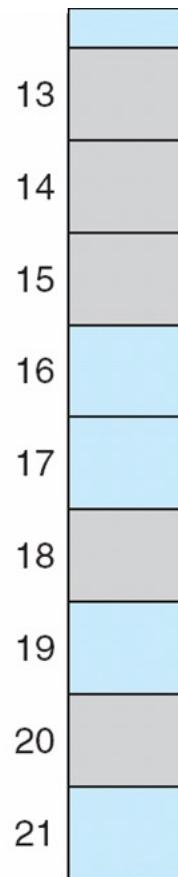
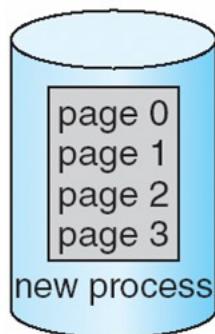
2047



# Free Frames

free-frame list

14  
13  
18  
20  
15

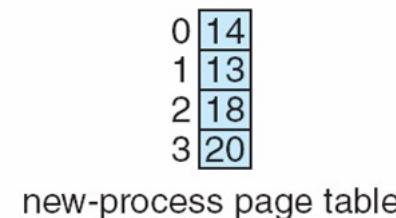
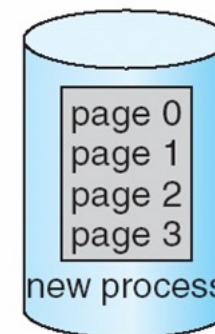


(a)

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
  - **Page-table length register (PTLR)** indicates size of the page table
- 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 **translation look-aside buffers (TLBs)** (also called **associative memory**).

# Translation Look-Aside Buffer

---

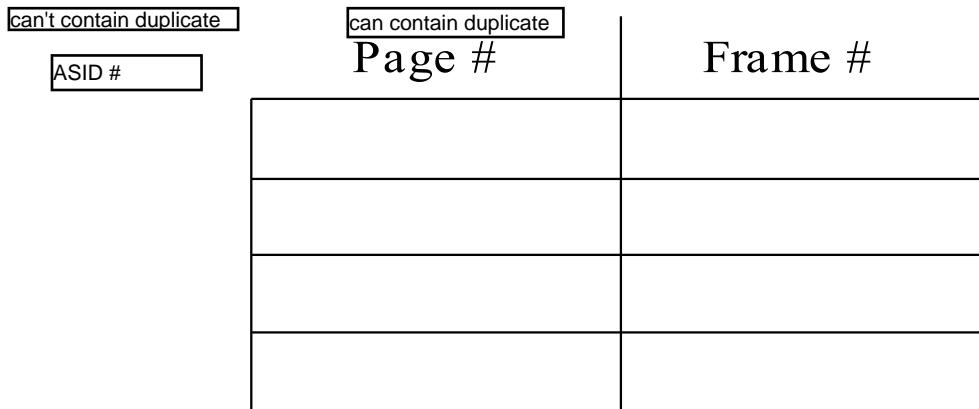
process id

- 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

# Hardware

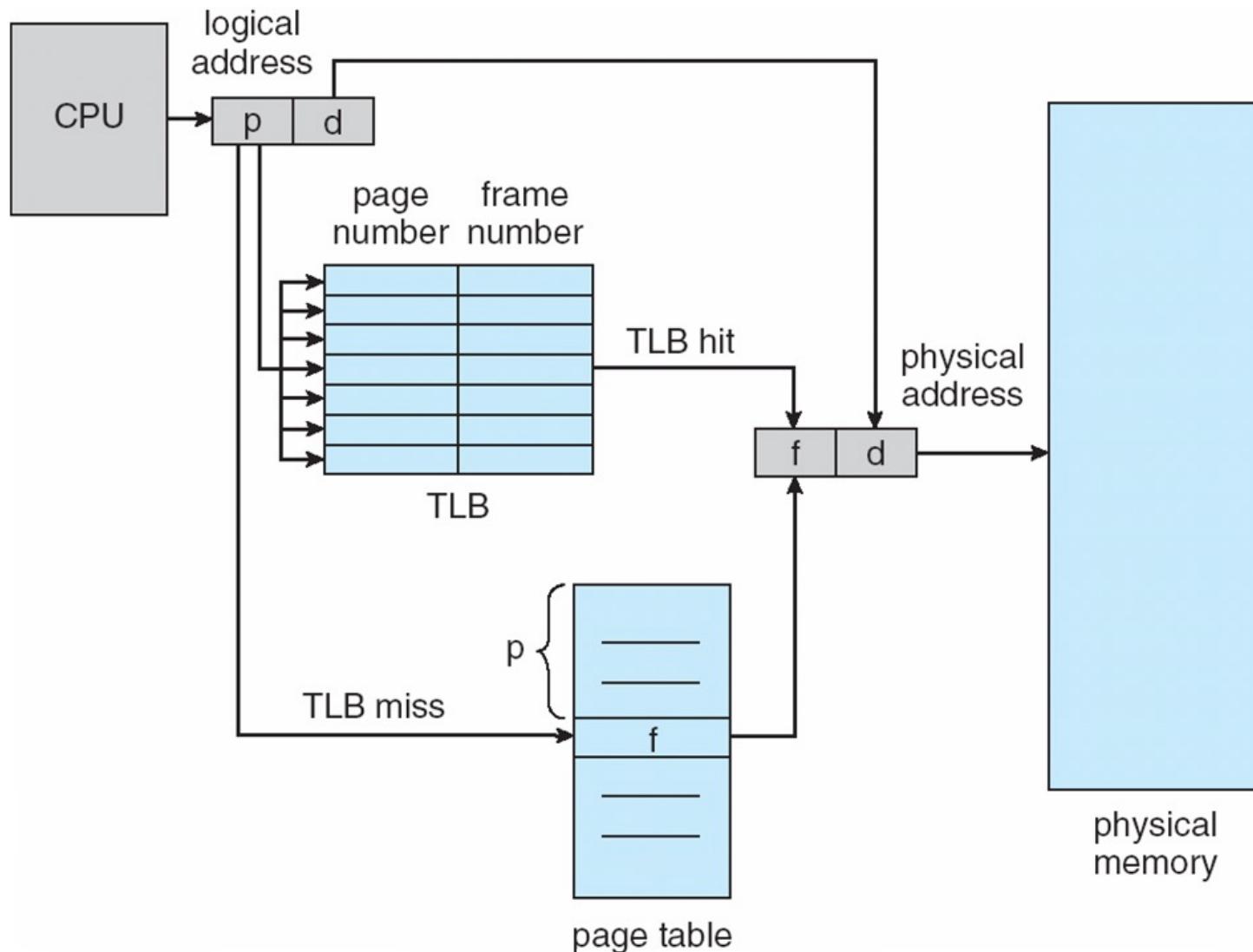
- Associative memory – parallel search

it is hardware supported



- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise get frame # from page table in memory

# Paging Hardware With TLB



# Effective Access Time

---

- Hit ratio: percentage of times that a page number is found in the TLB
- An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Suppose that 10 nanoseconds to access memory.
  - If we find the desired page in TLB then a mapped-memory access take 10 ns.
  - Otherwise, we need two memory access so it is 20 ns

here we suppose access time to TLB is zero, but it is not in reality



# Effective Access Time (Cont.)

---

- **Effective Access Time (EAT)**

$$EAT = 0.80 \times 10 + 0.20 \times 20 = 12 \text{ nanoseconds}$$

implying 20% slowdown in access time

- Consider a more realistic hit ratio of 99%,

$$EAT = 0.99 \times 10 + 0.01 \times 20 = 10.1\text{ns}$$

implying only 1% slowdown in access time.

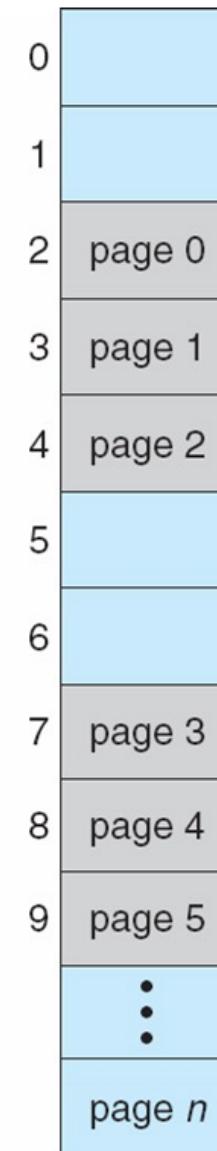
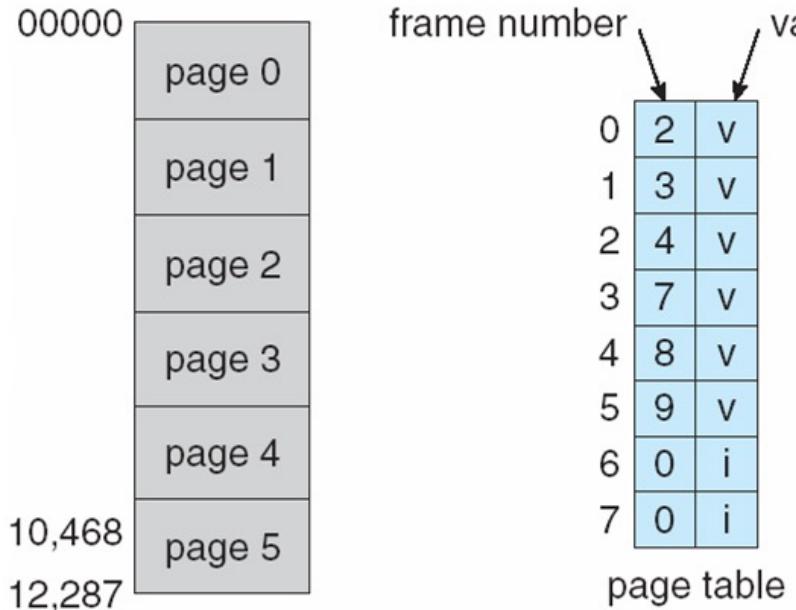
# Memory Protection

---

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed.
  - Can also add more bits to indicate page execute-only, and so on
- Valid-invalid bit attached to each entry in the page table:
  - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
  - “invalid” indicates that the page is not in the process’ logical address space
  - Or use [page-table length register \(PTLR\)](#)
- Any violations result in a trap to the kernel



# Valid (v) or Invalid (i) Bit In A Page Table



# Shared Pages

---

## ■ Shared code

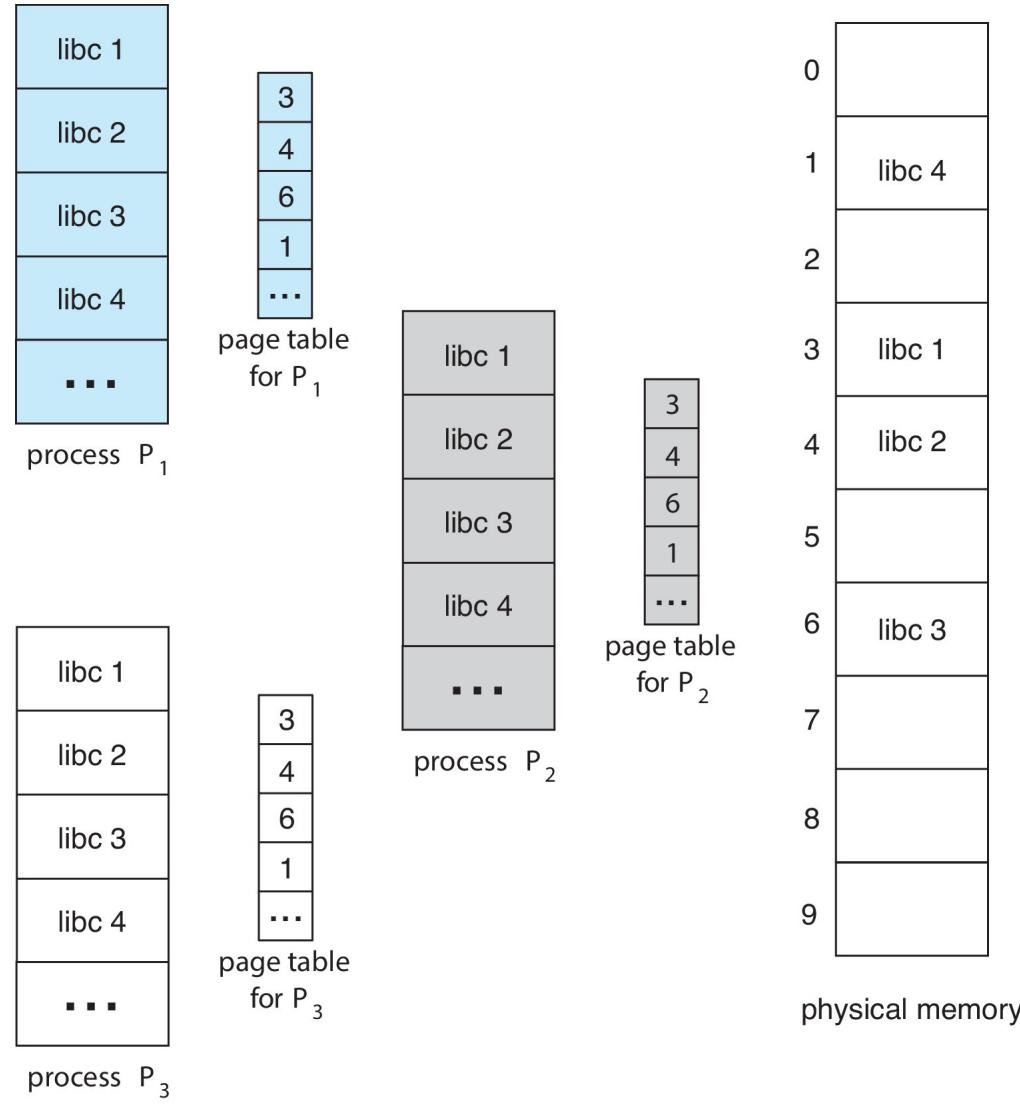
- One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
- Similar to multiple threads sharing the same process space
- Also useful for interprocess communication if sharing of read-write pages is allowed.

## ■ Private code and data

- Each process keeps a separate copy of the code and data
- The pages for the private code and data can appear anywhere in the logical address space.



# Shared Pages Example





# **Operating Systems**

## **Virtual Memory**

Seyyed Ahmad Javadi

[sajavadi@aut.ac.ir](mailto:sajavadi@aut.ac.ir)

Fall 2021

# Chapter 10: Virtual Memory

---

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement

# Objectives

---

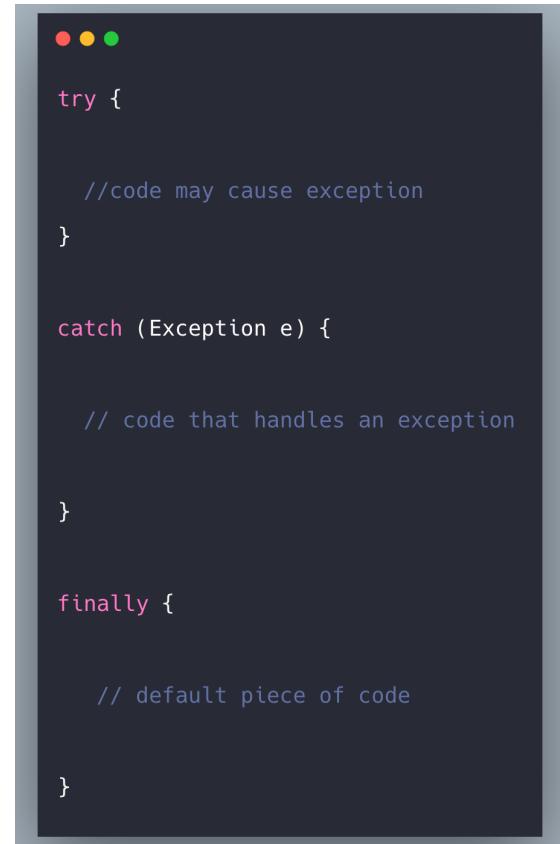
- Define virtual memory and describe its benefits.
- Illustrate how pages are loaded into memory using demand paging.
- Apply the FIFO, optimal, and LRU page-replacement algorithms.



# 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



```
try {
 //code may cause exception
}

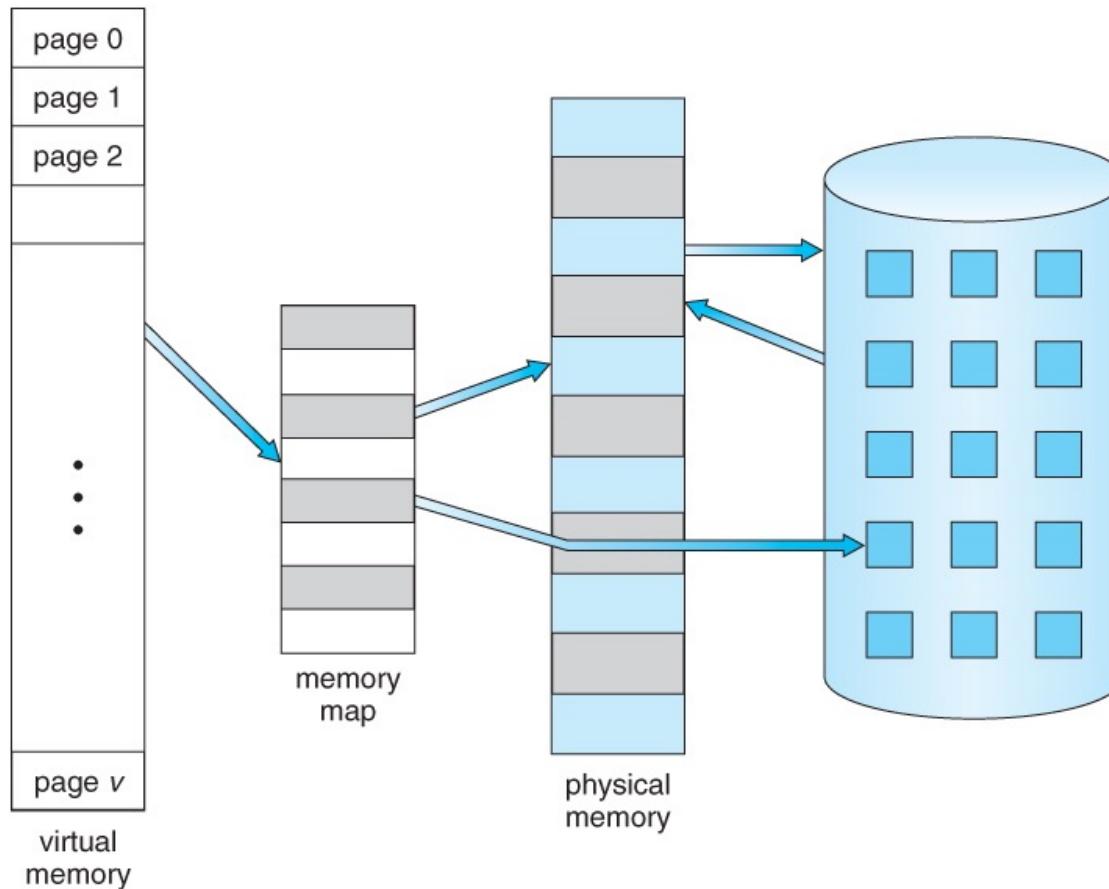
catch (Exception e) {
 // code that handles an exception
}

finally {
 // default piece of code
}
```

A screenshot of a terminal window on a Mac OS X system. The window has the characteristic red, yellow, and green control buttons in the top-left corner. The code inside the terminal is written in Java, demonstrating the use of try, catch, and finally blocks for exception handling.

# Benefits of executing partially-load programs

- Program no longer constrained by limits of physical memory

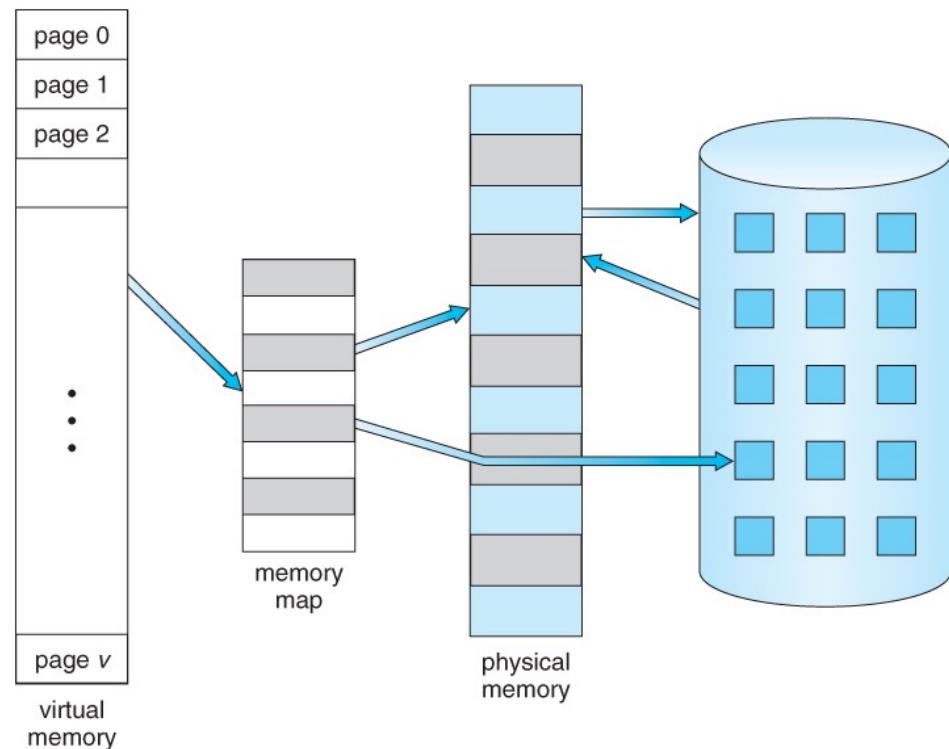


[https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/9\\_VirtualMemory.html](https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/9_VirtualMemory.html)



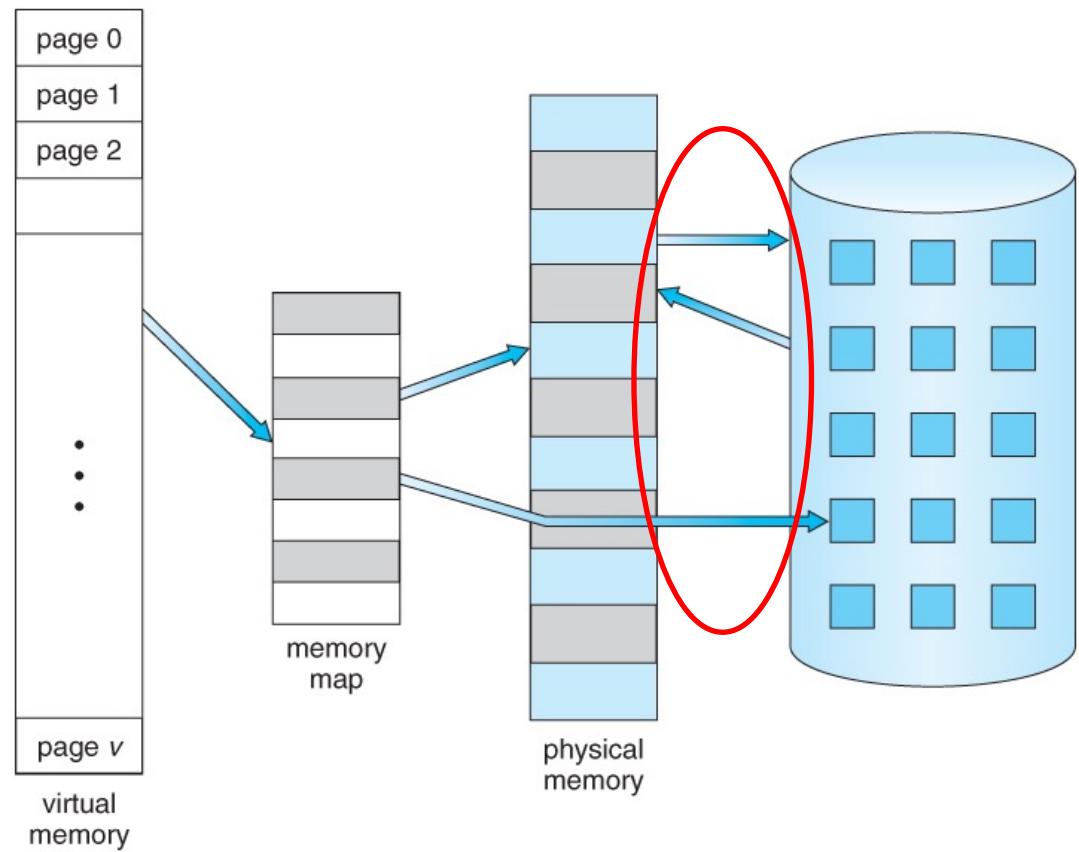
# Benefits of executing partially-load programs

- 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.



# Benefits of executing partially-load programs

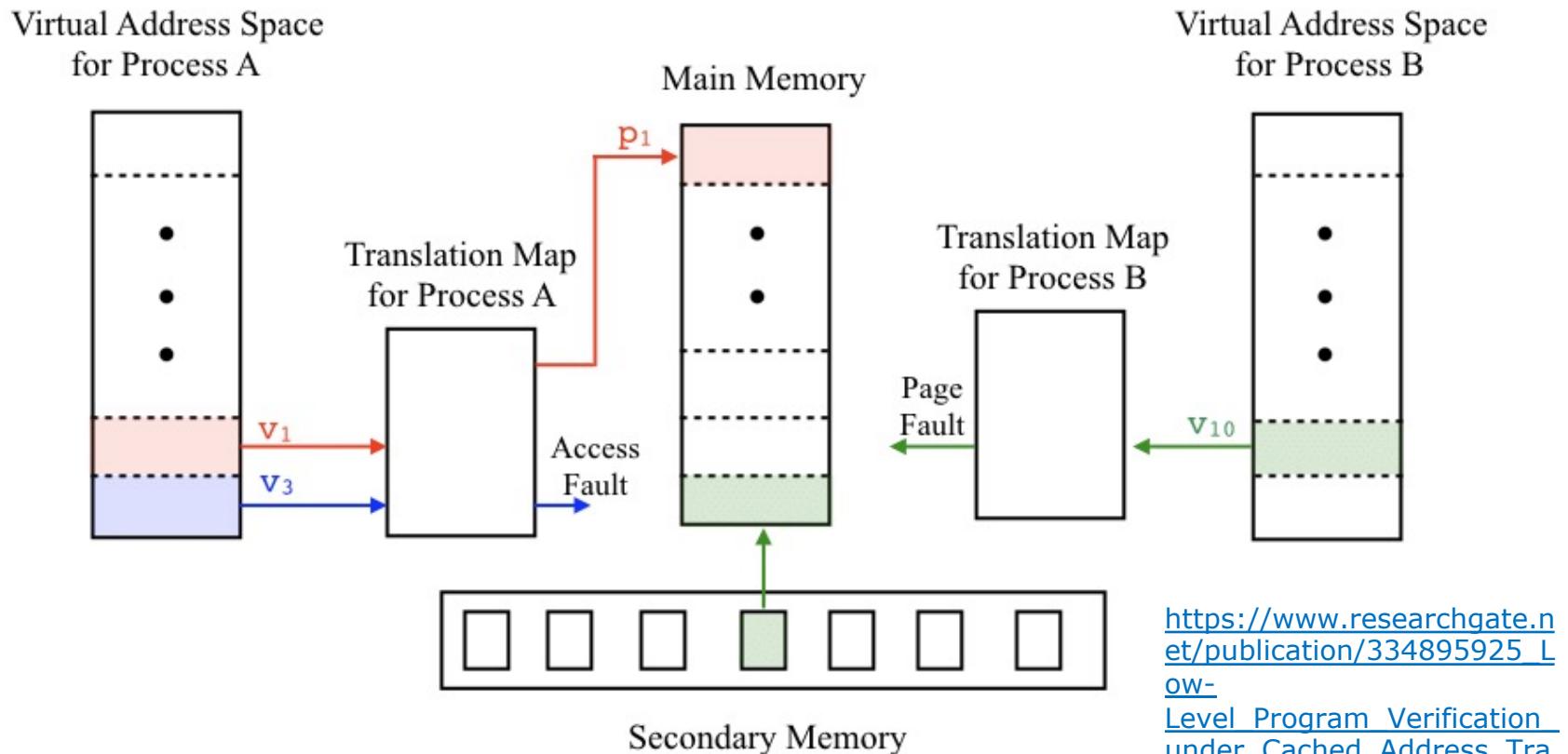
- Less I/O needed to load or swap programs into memory -> each user program runs faster.



# Virtual memory

separation of user logical memory from physical memory

Only part of the program needs to be in memory for execution

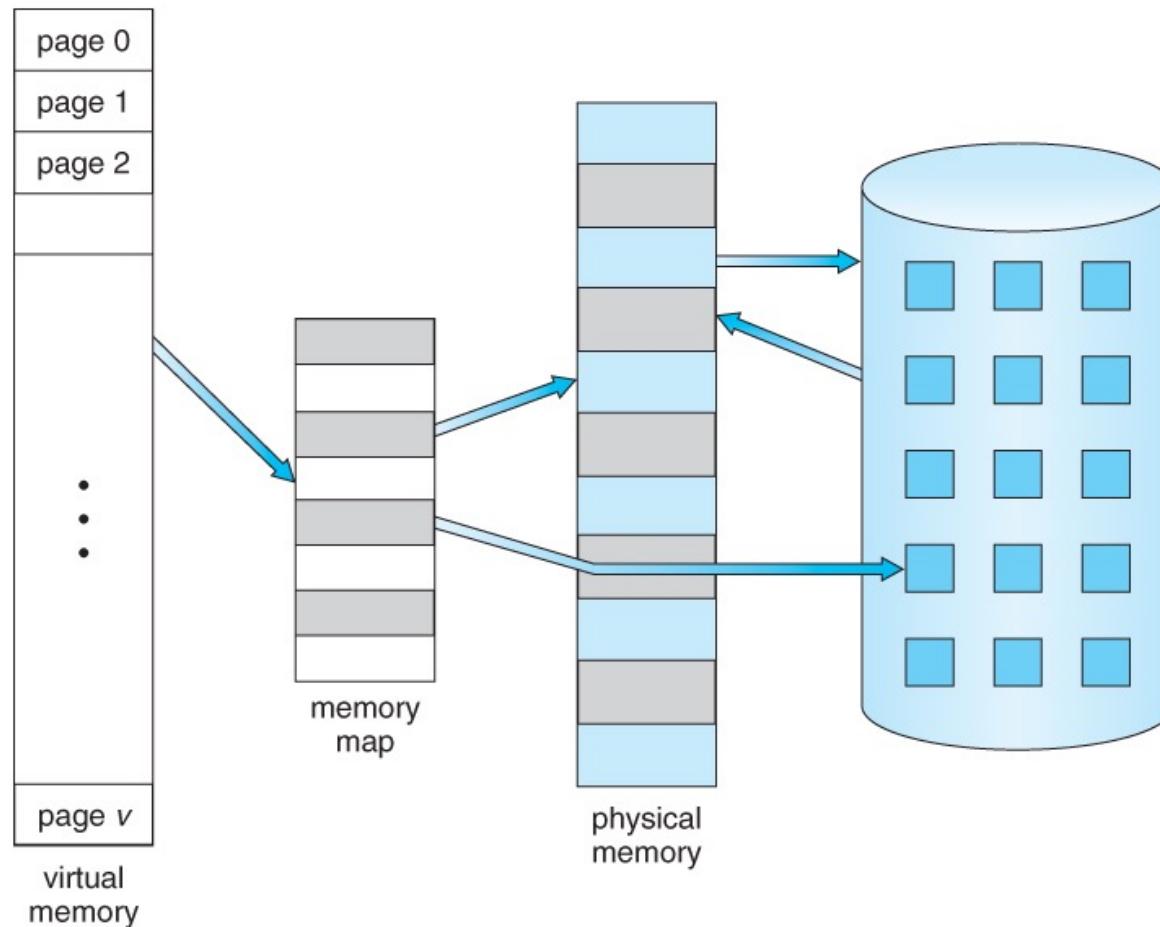


[https://www.researchgate.net/publication/334895925\\_Low\\_Level\\_Program\\_Verification\\_under\\_Cached\\_Address\\_Translation/figures?lo=1](https://www.researchgate.net/publication/334895925_Low_Level_Program_Verification_under_Cached_Address_Translation/figures?lo=1)



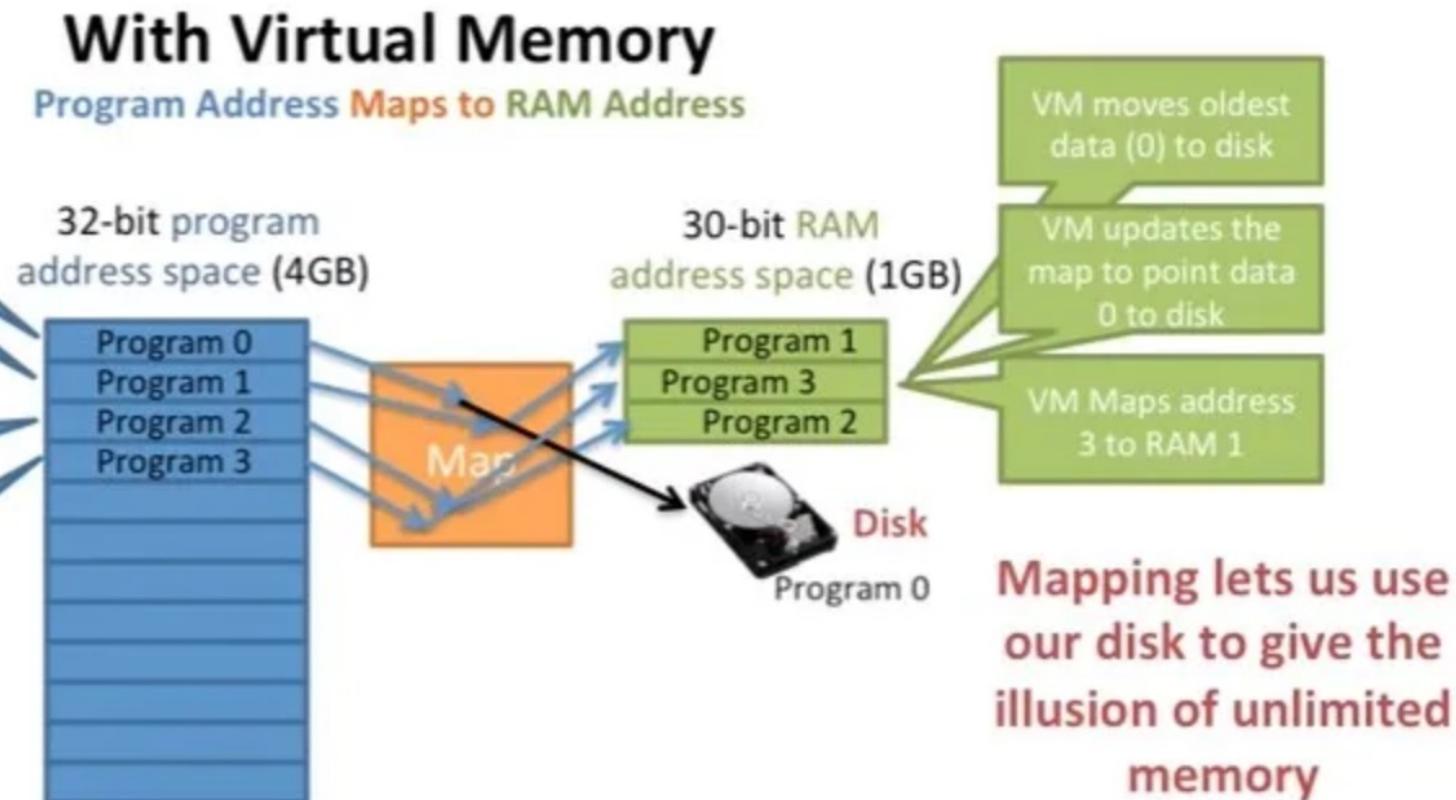
# Benefits of Virtual memory

- Logical address space can be much larger than physical address space



# Benefits of Virtual memory

- Allows address spaces to be shared by several processes



# Benefits of Virtual memory

---

- Allows for more efficient process creation
- More programs running concurrently
- Less I/O needed to load or swap processes

# Virtual memory (cont.)

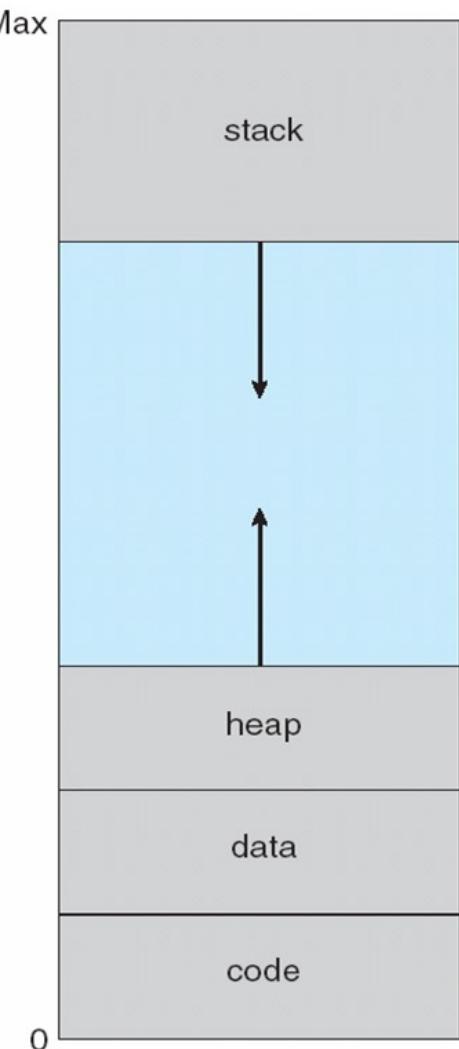
---

- **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**
  - Demand **segmentation**

we don't go over segmentation

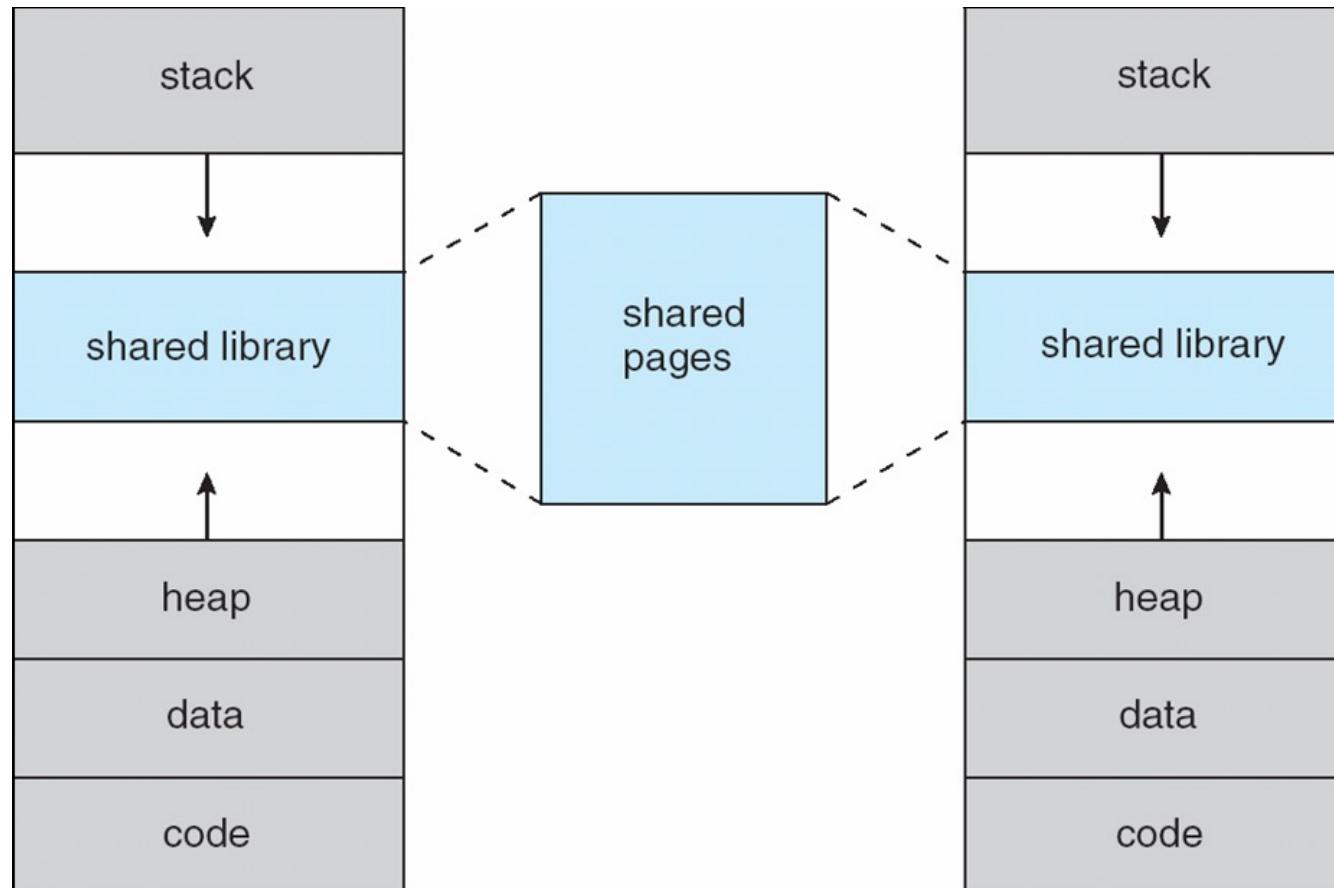
# Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between the two is hole
  - No physical memory needed until heap or stack grows to a given new page



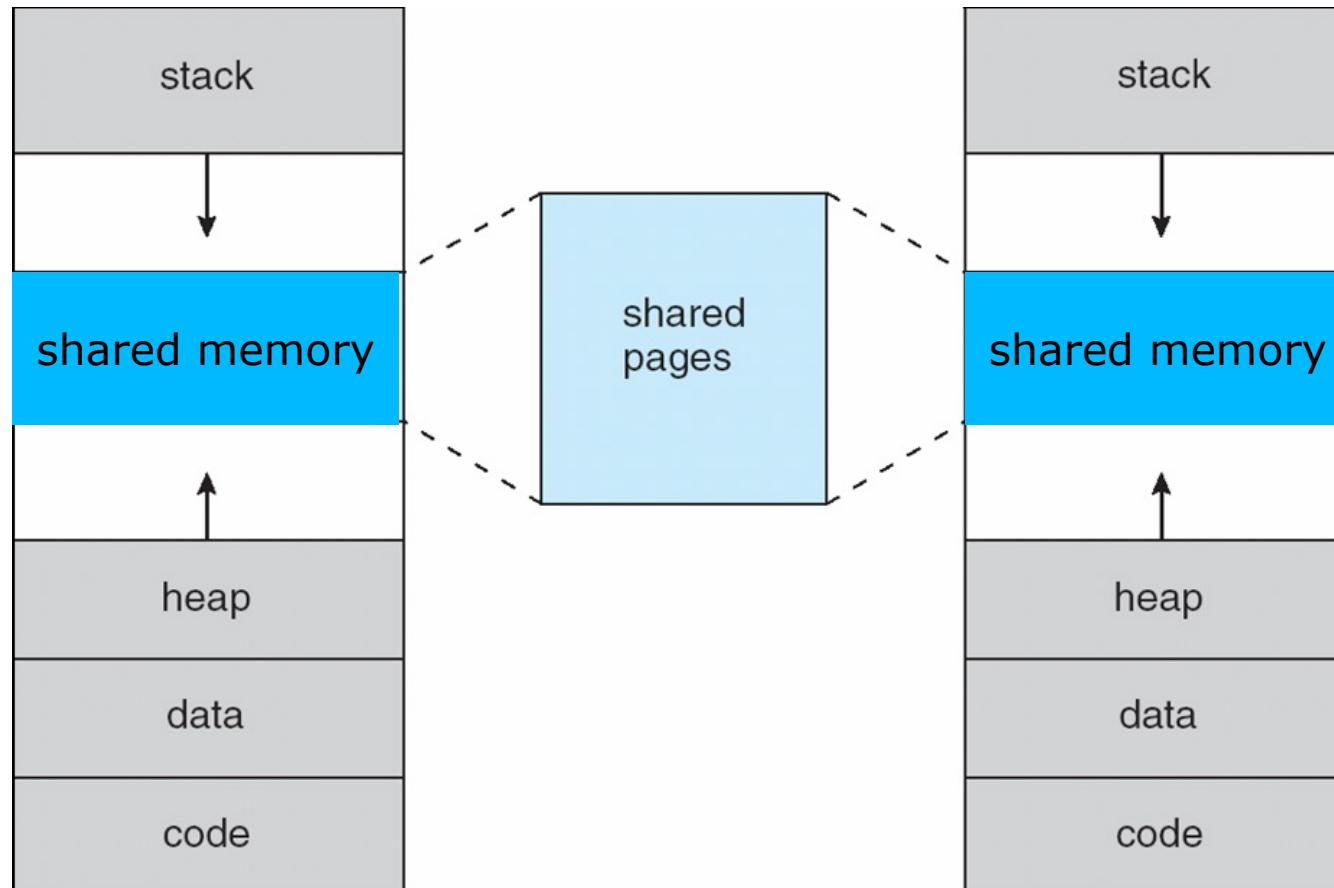
# Virtual-address Space

- System libraries shared via mapping into virtual address space



# Virtual-address Space

- Shared memory by mapping pages read-write into virtual address space



# Virtual-address Space

---

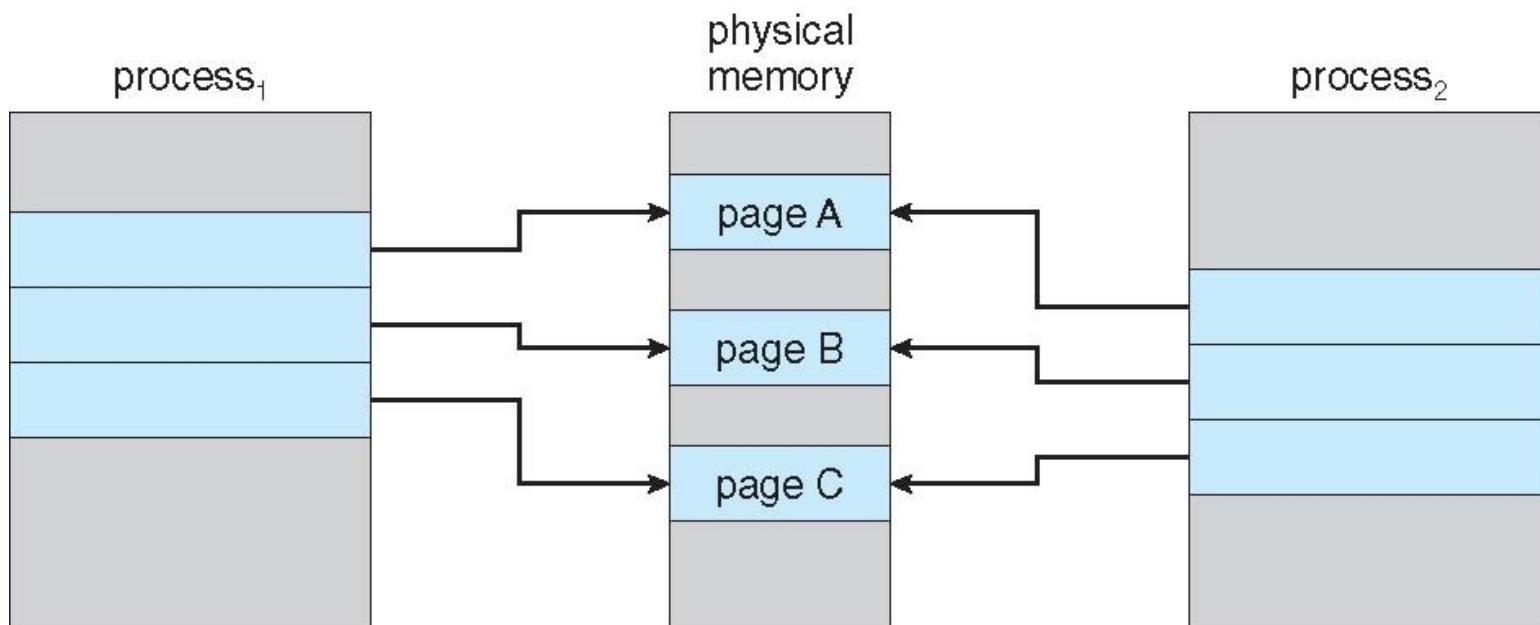
- Pages can be shared during `fork()`, speeding process creation
- How?

# Copy-on-Write

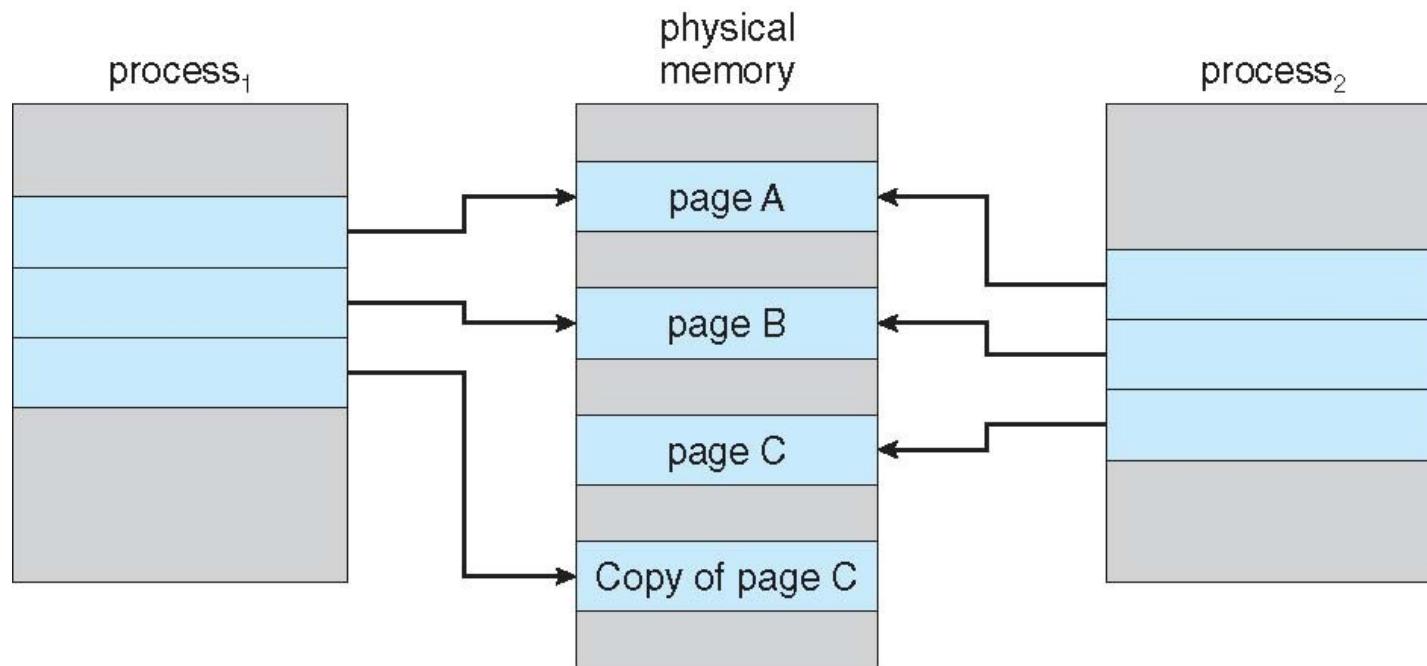
---

- **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.
- Furthermore, often exec is called immediately after fork.

# Before Process 1 Modifies Page C



# After Process 1 Modifies Page C



# Demand Paging

---

- Could bring entire process into memory at load time
- Or 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

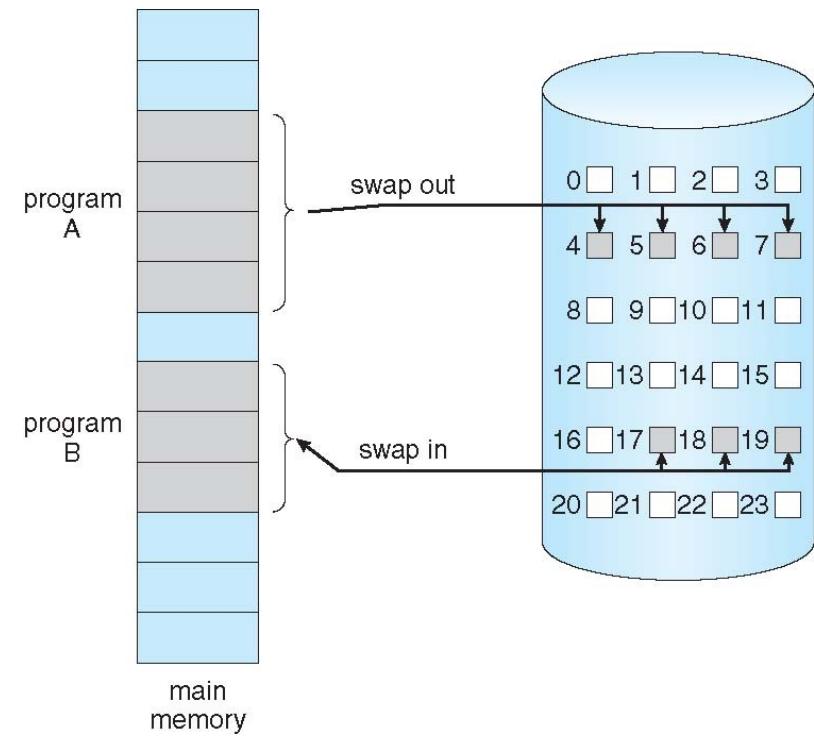


# Demand Paging (Cont.)

- Similar to paging system **with swapping** (diagram on right)

- Page is needed  $\Rightarrow$  reference to it
  - invalid reference  $\Rightarrow$  abort
  - not-in-memory  $\Rightarrow$  bring to memory

- **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 swapping out again
- Instead, pager brings in only those pages into memory
- How to determine that set of pages?
  - Need **new MMU functionality** to implement demand paging



# Basic Concepts (Cont.)

---

- 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 # | page # | valid-invalid bit |
|---------|--------|-------------------|
|         |        | invalid           |
|         |        | v                 |
|         |        | v                 |
|         |        | v                 |
|         |        | i                 |
| ...     |        |                   |
|         |        | i                 |
|         |        | i                 |

page table

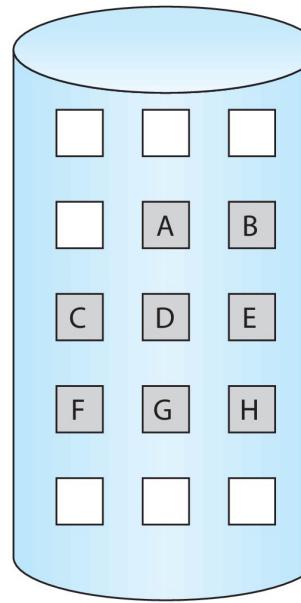
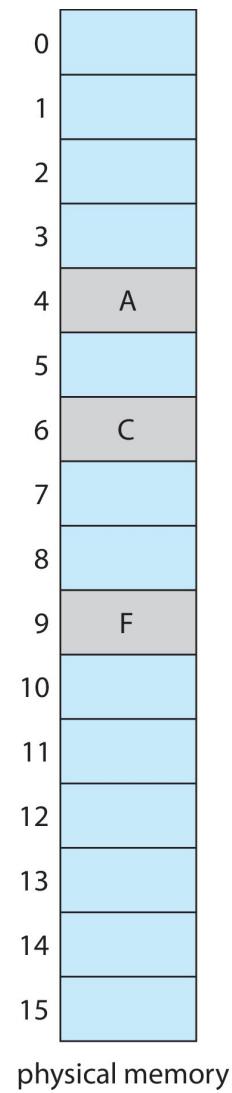
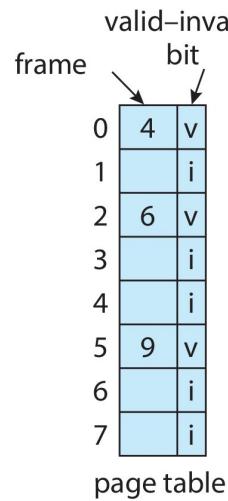
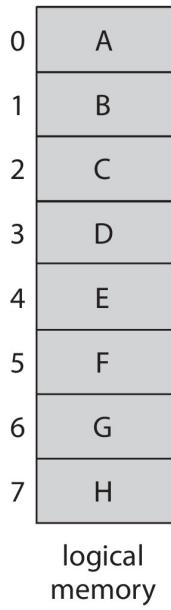
# Valid-Invalid Bit

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

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

page table

# Page Table When Some Pages Are Not in Main Memory



backing store

# Steps in Handling Page Fault

---

1. If there is a reference to a page, first reference to that page will trap to operating system

- Page fault

2. Operating system looks at another table to decide:

- Invalid reference  $\Rightarrow$  abort
- Just not in memory

...

# Steps in Handling Page Fault

---

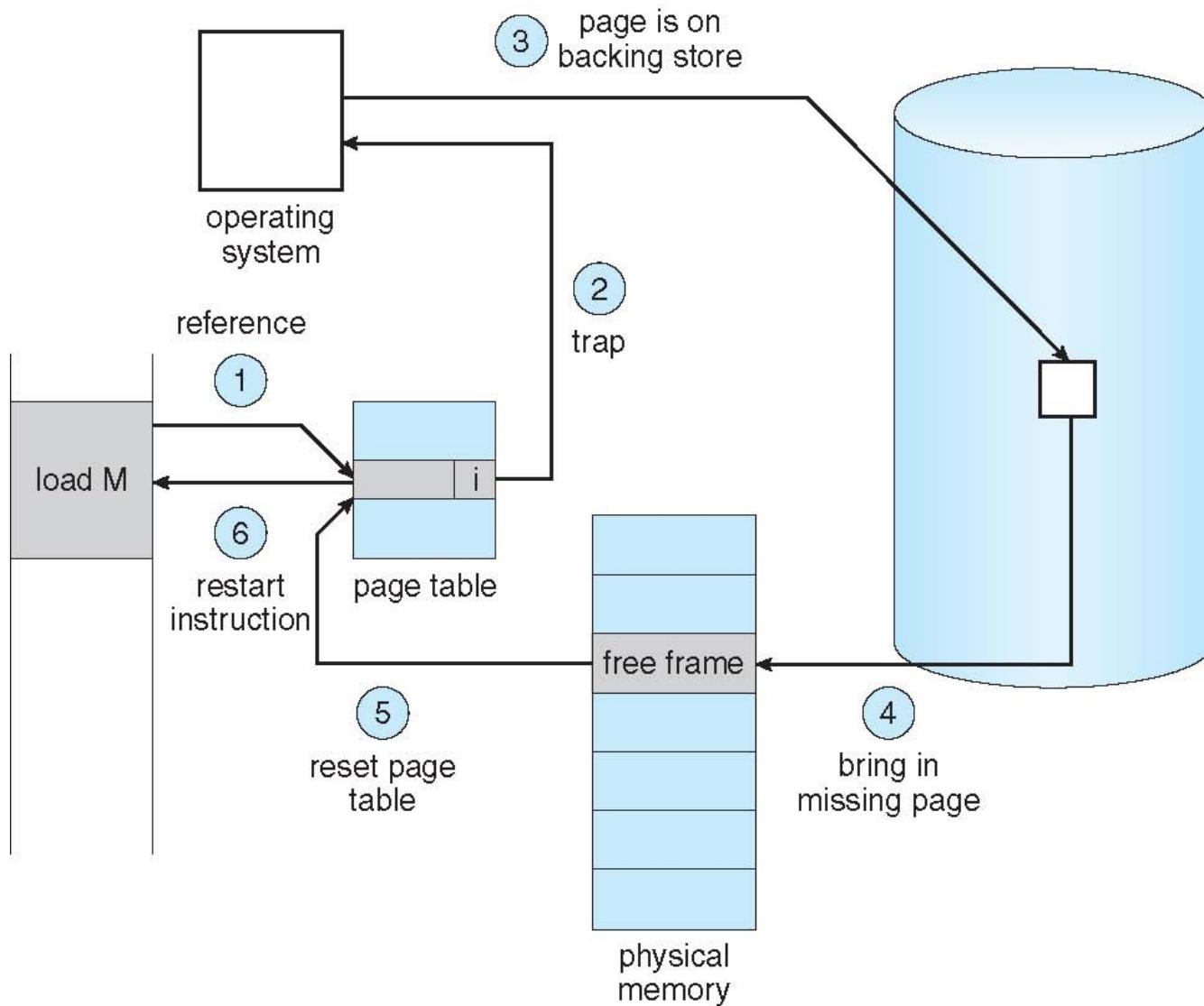
...

3. Find free frame
4. Swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory

**Set validation bit = v**

6. Restart the instruction that caused the page fault

# Steps in Handling a Page Fault (cont.)



# 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**

# Aspects of Demand Paging (cont.)

---

- ...
- 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**



# Locality of reference

- Page faults are expensive!
- **Thrashing:** Process spends most of the time paging in and out instead of executing code.
- Most programs display a pattern of behavior called the principle of locality of reference.

**Locality of Reference:** A program that references a location  $n$  at some point in time is likely to reference the same location  $n$  and locations in the immediate vicinity of  $n$  in the near future.

Source: <https://people.engr.tamu.edu/bettati/Courses/410/2017A/Slides/virtmemory.pdf>

# Aspects of Demand Paging

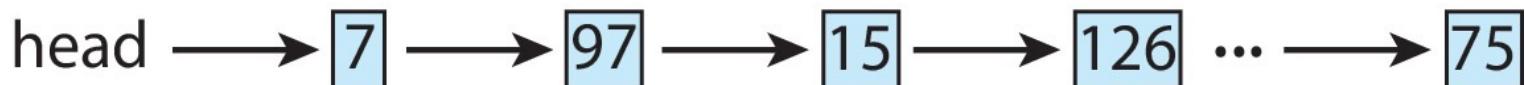
---

- ...
- ...
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)

# Free-Frame List

---

- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.
- Most operating systems maintain a **free-frame list** -- a pool of free frames for satisfying such requests.



# Free-Frame List (cont.)

---

- ..  
head → 
- Operating system typically allocate free frames using a technique known as **zero-fill-on-demand** -- the content of the frames zeroed-out before being allocated.
- When a system starts up, all available memory is placed on the free-frame list.



# Stages in Demand Paging – Worse Case

---

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk

....

# Stages in Demand Paging – Worse Case

---

....

5. Issue a read from the disk to a free frame:
  - a) Wait in a queue for this device until the read request is serviced
  - b) Wait for the device seek and/or latency time
  - c) Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)

...

# Stages in Demand Paging (cont.)

---

8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction



# Performance of Demand Paging

---

- Three major activities
  - **Service the interrupt** – careful coding means just several hundred instructions needed
  - **Read the page** – lots of time
  - **Restart the process** – again just a small amount of time

# Performance of Demand Paging (cont.)

---

- ...
- Page Fault Rate  $0 \leq p \leq 1$ 
  - if  $p = 0$  no page faults
  - if  $p = 1$ , every reference is a fault

- Effective Access Time (EAT)

$$EAT = (1 - p) \times \text{memory access}$$

+  $p$  (page fault overhead

+ swap page out

+ swap page in )

# Demand Paging Example

---

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- $EAT = (1 - p) \times 200 + p (8 \text{ milliseconds})$ 
$$= (1 - p) \times 200 + p \times 8,000,000$$
$$= 200 + p \times 7,999,800$$
- If one access out of 1,000 causes a page fault, then  
 $EAT = 8.2 \text{ microseconds.}$

This is a slowdown by a factor of 40!!

# Demand Paging Example (cont.)

---

- ....
- If want performance degradation < 10 percent
  - $220 > 200 + 7,999,800 \times p$   
 $20 > 7,999,800 \times p$
  - $p < .0000025$
  - < one page fault in every 400,000 memory accesses





# **Operating Systems**

## **Security-Part1**

Seyyed Ahmad Javadi

[sajavadi@aut.ac.ir](mailto:sajavadi@aut.ac.ir)

Fall 2021

# Chapter 16: Security

---

- The Security Problem
- Program Threats
- System and Network Threats
- Cryptography as a Security Tool
- User Authentication
- Implementing Security Defenses
- Firewalling to Protect Systems and Networks
- Computer-Security Classifications
- An Example: Windows 7

# Objectives

---

- Discuss security threats and attacks
- Explain the fundamentals of encryption, authentication, and hashing
- Examine the uses of cryptography in computing
- Describe the various countermeasures to security attacks



# The Security Problem

---

- System **secure** if resources used and accessed as intended under all circumstances
  - Unachievable
- **Intruders (crackers)** attempt to breach security
- **Threat** is potential security violation
- **Attack** is attempt to breach security
- Attack can be accidental or malicious
- Easier to protect against accidental than malicious misuse

# Security Violation Categories

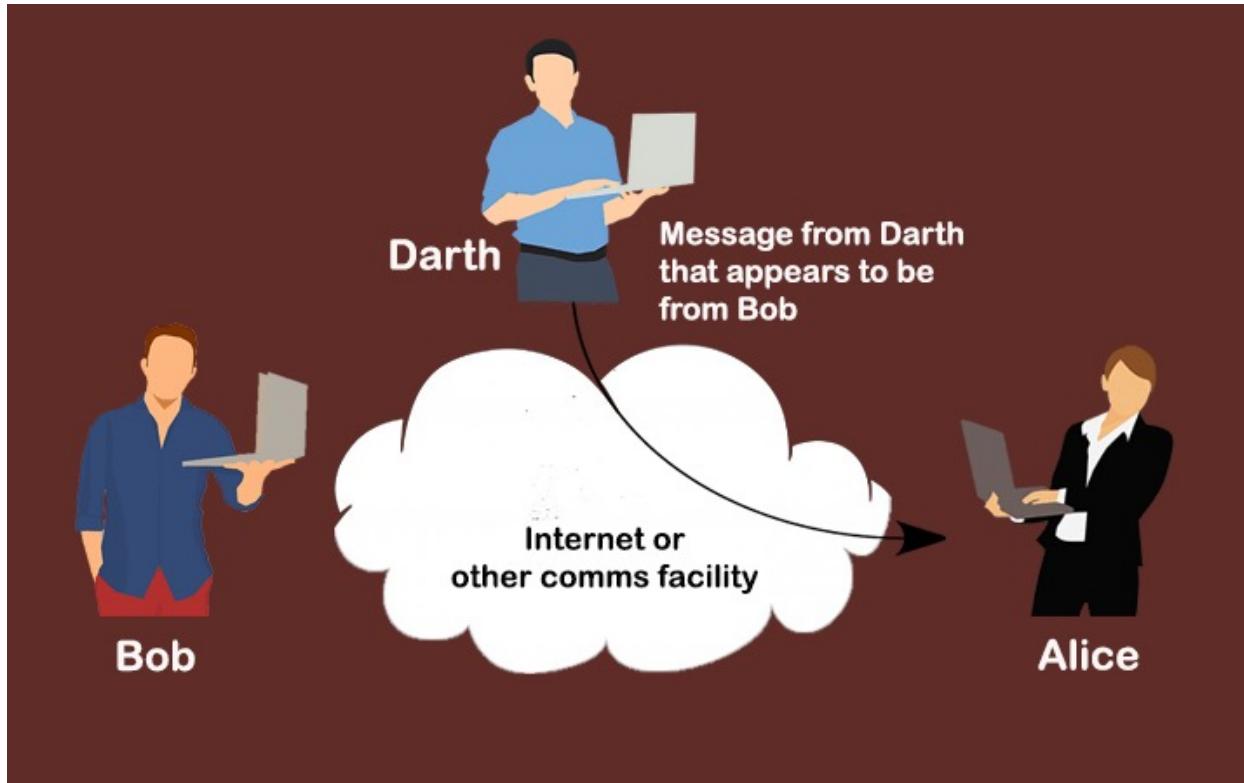
---

- **Breach of confidentiality**
  - Unauthorized reading of data
- **Breach of integrity**
  - Unauthorized modification of data
- **Breach of availability**
  - Unauthorized destruction of data
- **Theft of service**
  - Unauthorized use of resources
- **Denial of service (DOS)**
  - Prevention of legitimate use

# Security Violation Methods

## ■ Masquerading (breach authentication)

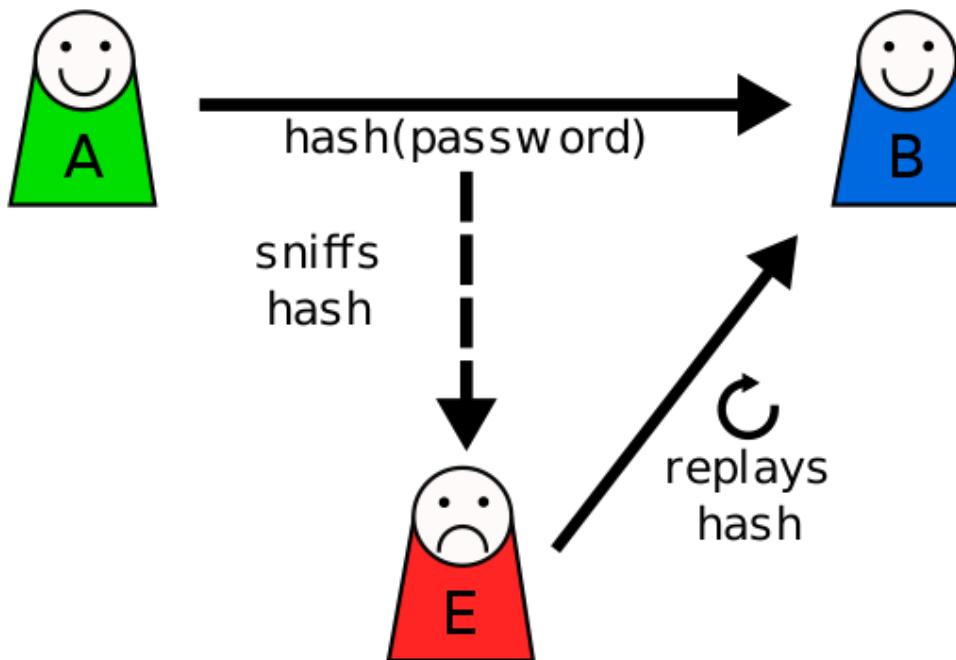
- Pretending to be an authorized user to escalate privileges



# Security Violation Methods

## ■ Replay attack

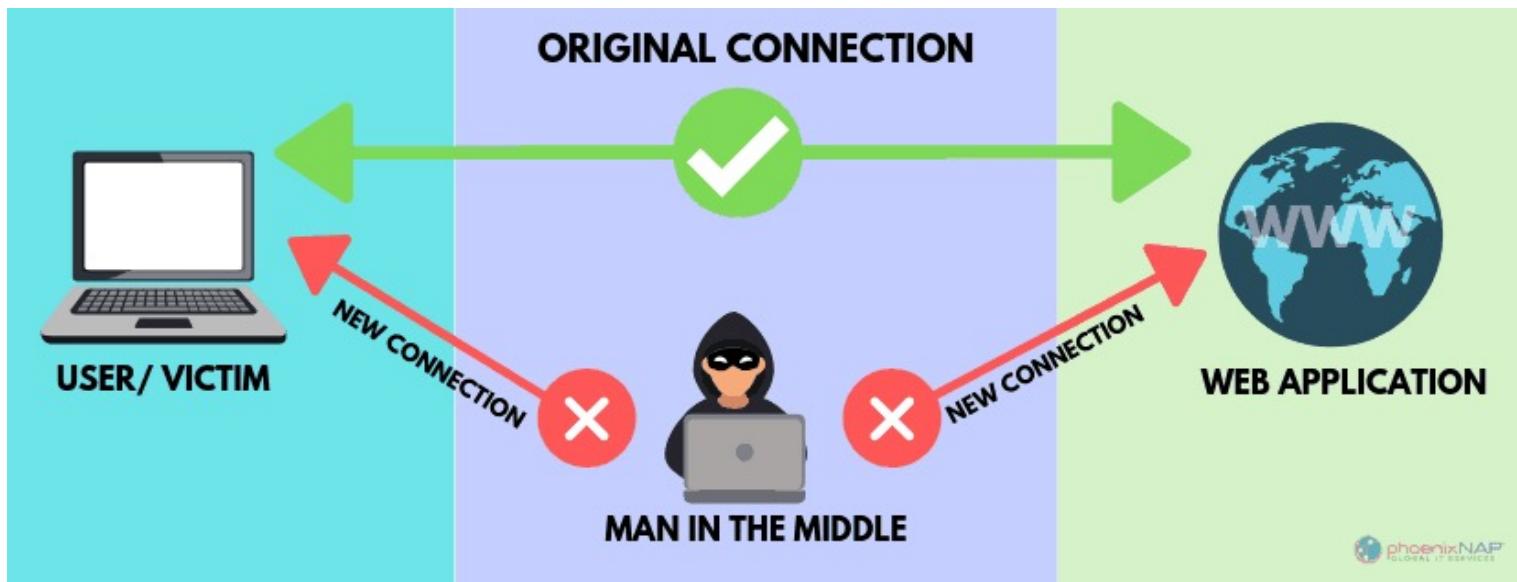
- As is or with **message modification**



# Security Violation Methods

## ■ Man-in-the-middle attack

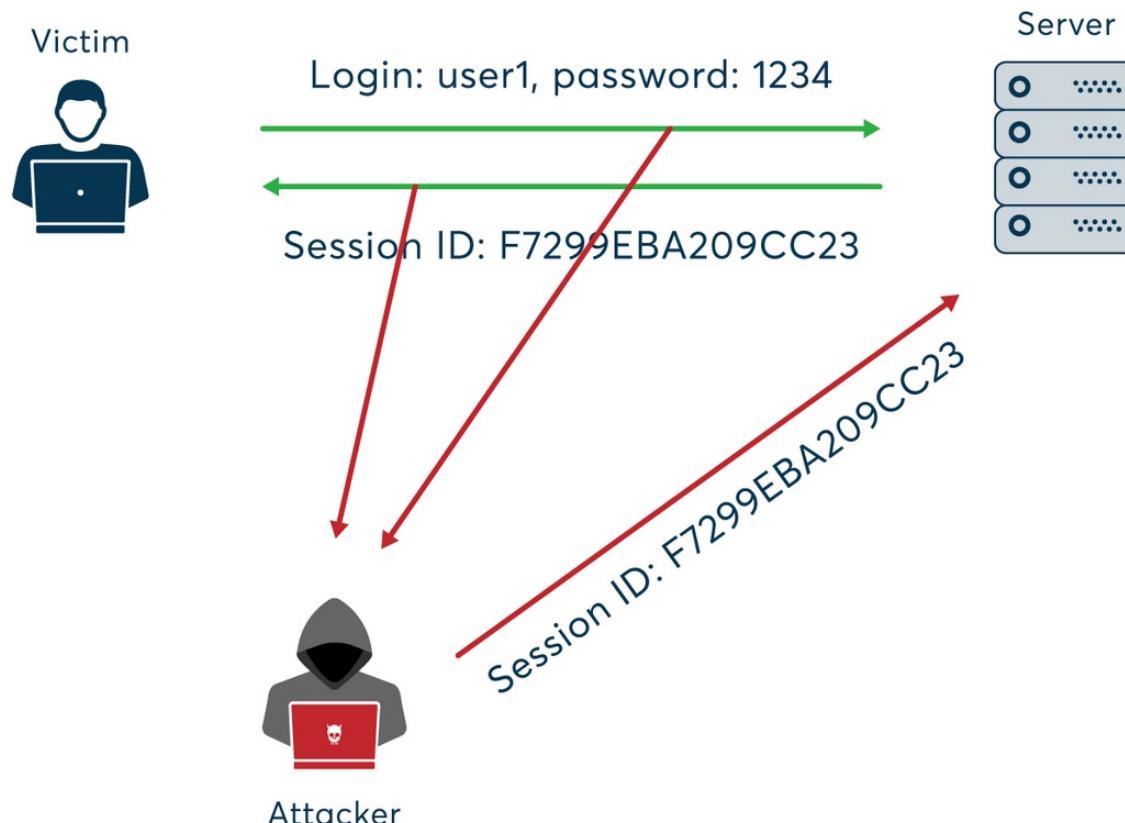
- Intruder sits in data flow, masquerading as sender to receiver and vice versa



# Security Violation Methods

## ■ Session hijacking

- Intercept an already-established session to bypass authentication

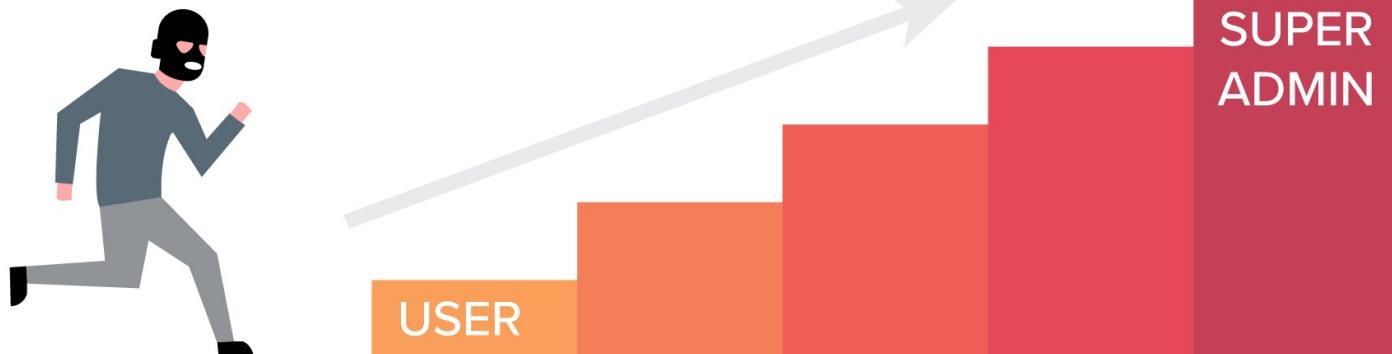


# Security Violation Methods

## ■ Privilege escalation

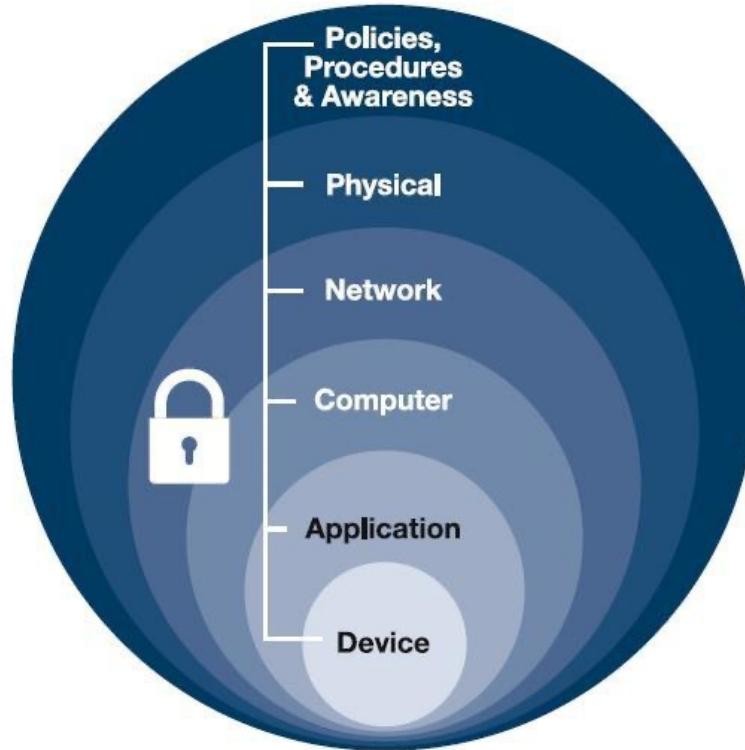
- Common attack type with access beyond what a user or resource is supposed to have

## PRIVILEGE ESCALATION



# Security Measure Levels

- Impossible to have absolute security, but make cost to perpetrator sufficiently high to deter most intruders.



# Security Measure Levels

---

## ■ Physical

- Data centers, servers, connected terminals

## ■ Application

- Benign or malicious apps can cause security problems

## ■ Operating System

- Protection mechanisms, debugging

## ■ Network

- Intercepted communications, interruption, DOS



# Security Measure Levels (cont.)

---

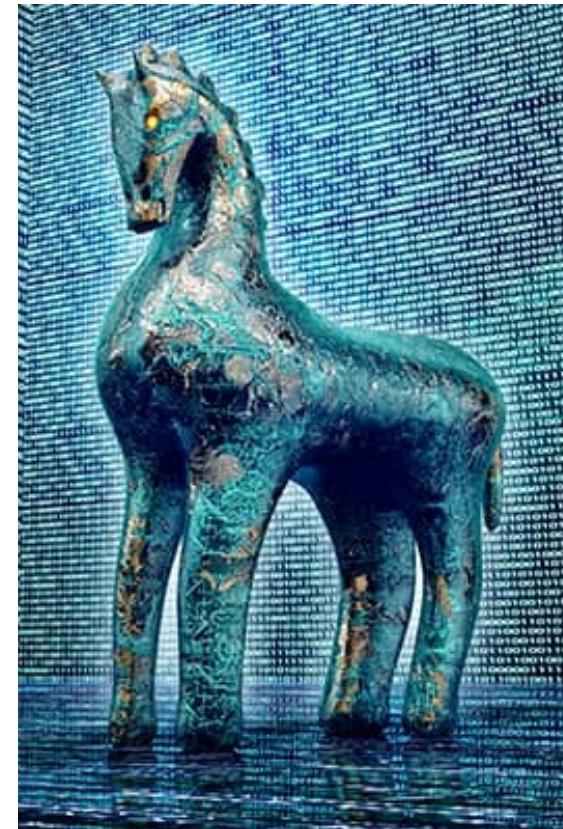
- Security is as weak as the weakest link in the chain
- Humans a risk too via **phishing** and **social-engineering** attacks
- But can too much security be a problem?

yes, can tire people

# Program Threats

---

- Many variations, many names
- **Trojan Horse**
  - Code segment that misuses its environment
  - Exploits mechanisms for allowing programs written by users to be executed by other users
  - **Spyware, pop-up browser windows, covert channels**
  - Up to 80% of spam delivered by spyware-infected systems



<https://www.kaspersky.com/resource-center/threats/trojans>

# Program Threats (cont.)

---

- Many variations, many names
- **Trap Door**
  - Specific user identifier or password that circumvents normal security procedures
  - Could be included in a compiler
  - How to detect them?



# Four-layered Model of Security

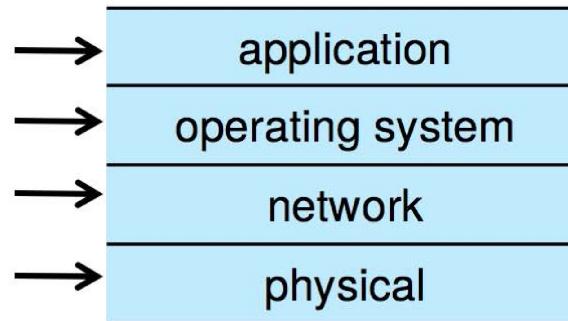
## types of attacks

logic bugs, design flaws, code injections

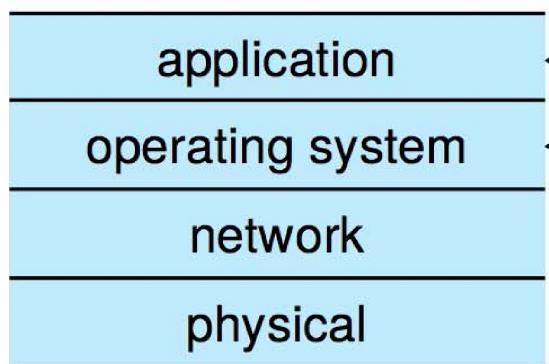
insecure defaults, platform vulnerabilities

sniffing, spoofing, masquerading

console access, hardware-based attacks



## attack prevention methods



# Program Threats (cont.)

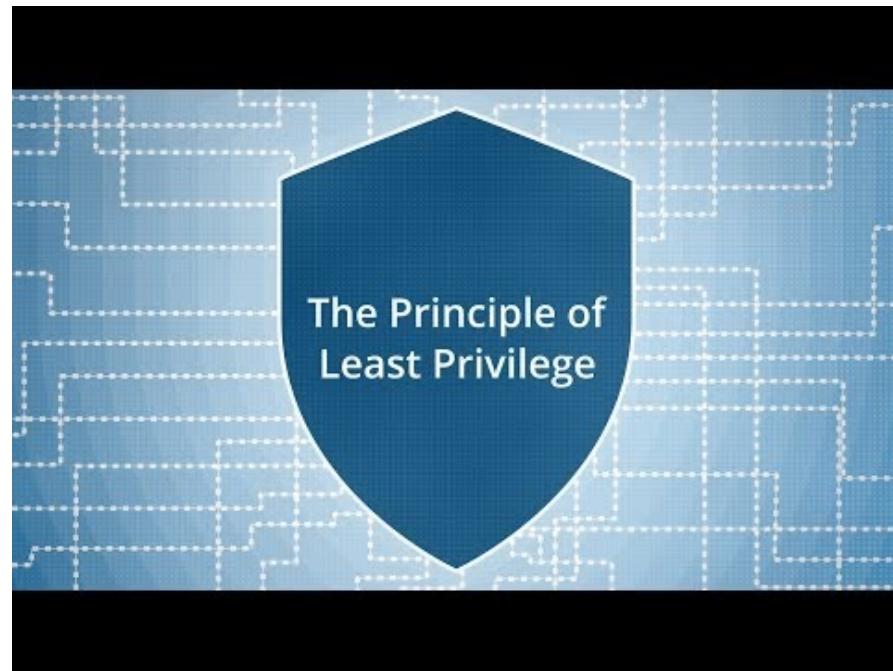
- **Malware**-Software designed to exploit, disable, or damage computer
- **Trojan Horse** – Program that acts in a clandestine manner
  - **Spyware** – Program frequently installed with legitimate software to display ads, capture user data
  - **Ransomware** – locks up data via encryption, demanding payment to unlock it



# Program Threats (cont.)

---

- Others include trap doors, logic bombs
- All try to violate the Principle of Least Privilege
- Goal frequently is to leave behind Remote Access Tool (RAT) for repeated access.



# Program Threats (cont.)

---

## *THE PRINCIPLE OF LEAST PRIVILEGE*

“The principle of least privilege. Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job. The purpose of this principle is to reduce the number of potential interactions among privileged programs to the minimum necessary to operate correctly, so that one may develop confidence that unintentional, unwanted, or improper uses of privilege do not occur.”—Jerome H. Saltzer, describing a design principle of the Multics operating system in 1974: <https://pdfs.semanticscholar.org/1c8d/06510ad449ad24fbdd164f8008cc730cab47.pdf>.



# C Program with Buffer-overflow Condition

---

```
#include <stdio.h>

#define BUFFER SIZE 256

int main(int argc, char *argv[])
{
 char buffer[BUFFER SIZE];
 if (argc < 2)
 return -1;
 else {
 strcpy(buffer, argv[1]);
 return 0;
 }
}
```



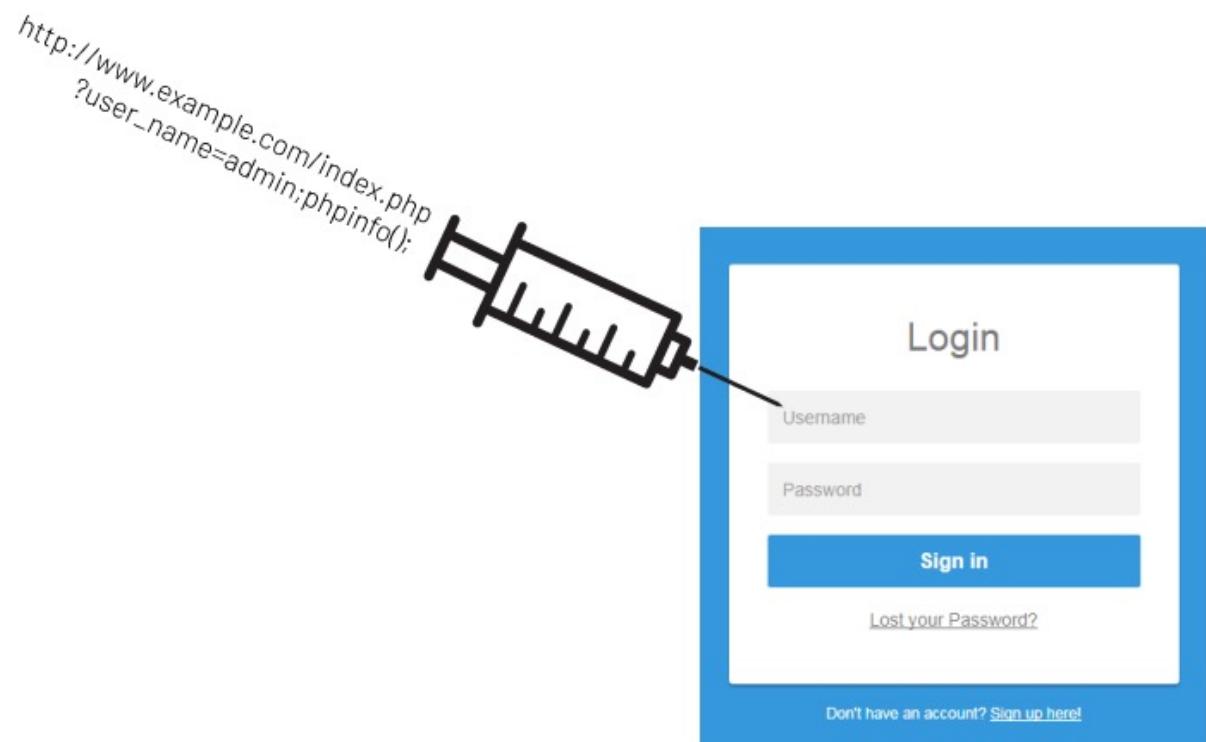
# C Program with Buffer-overflow Condition

- **Code review** can help – programmers review each other's code, looking for logic flows, programming flaws.



# Code Injection

- **Code-injection attack** occurs when system code is not malicious but has bugs allowing executable code to be added or modified.



# Code Injection (cont.)

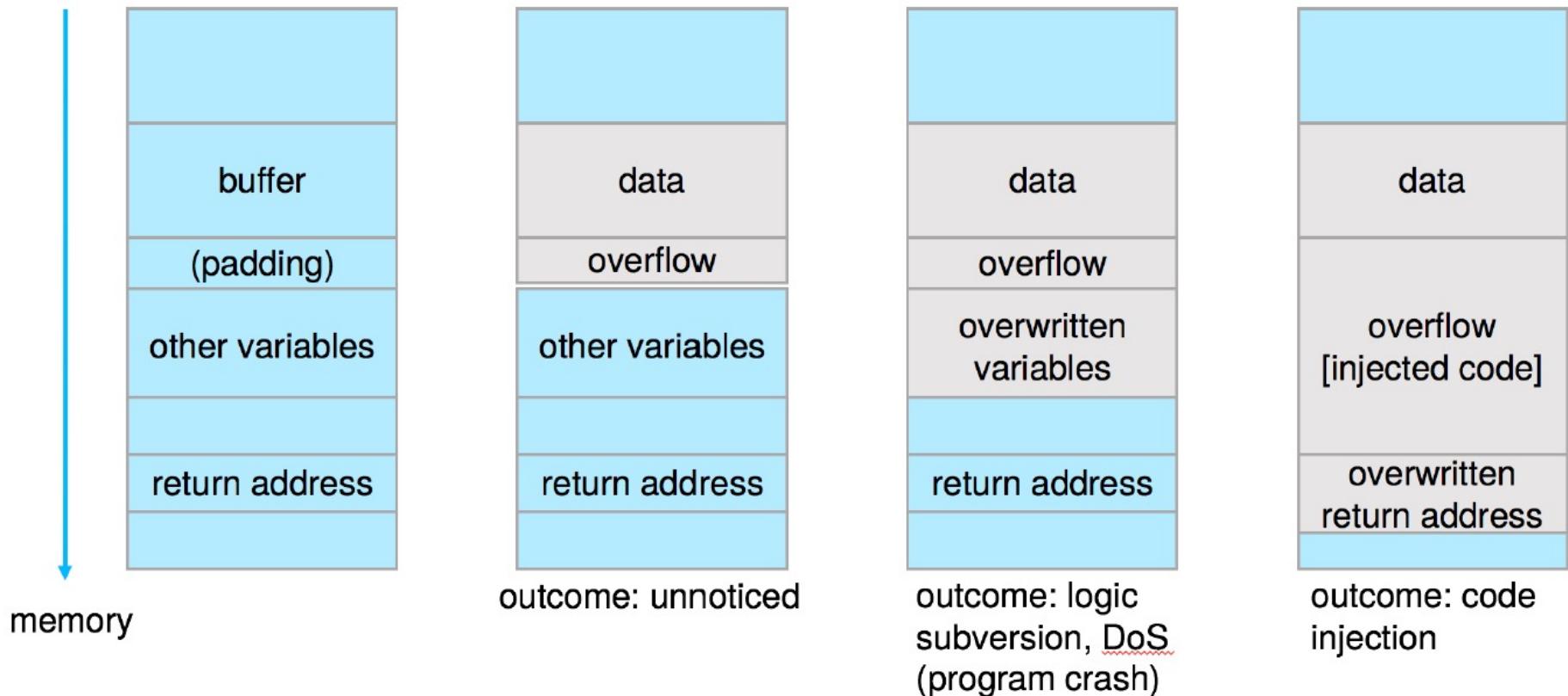
---

- Results from poor or insecure programming paradigms, commonly in low level languages like C or C++ which allow for direct memory access through pointers.
- Goal is a buffer overflow in which code is placed in a buffer and execution caused by the attack.
- Can be run by script kiddies – use tools written but exploit identifiers.



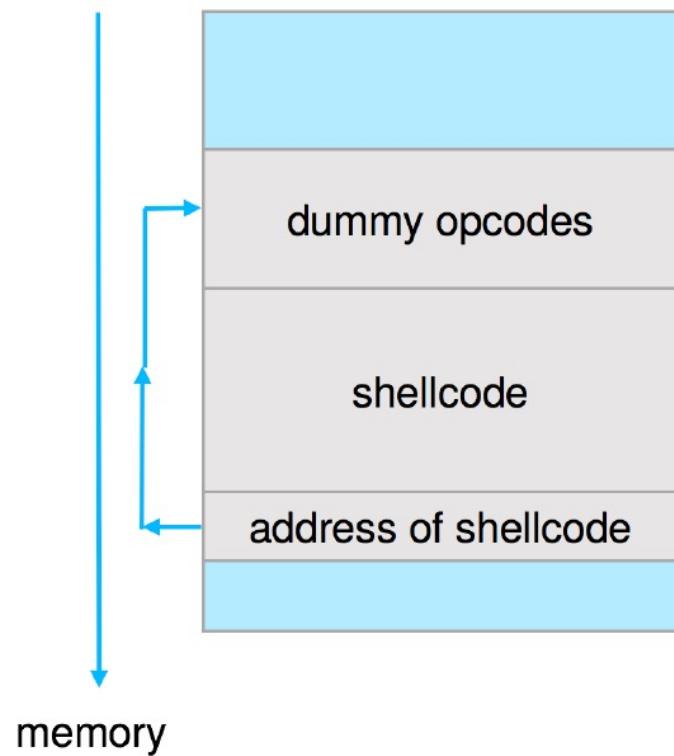
# Code Injection (cont.)

- Outcomes from code injection include:



# Code Injection (cont.)

- Frequently use trampoline to code execution to exploit buffer overflow:



# Great Programming Required?

---

- For the first step of determining the bug, and second step of writing exploit code, yes.
- **Script kiddies** can run pre-written exploit code to attack a given system.
- Attack code can get a shell with the processes' owner's permissions.
  - Or open a network port, delete files, download a program, etc.



# Great Programming Required?

---

- Depending on bug, attack can be executed across a network using allowed connections, bypassing firewalls.
  
- Buffer overflow can be disabled by disabling stack execution or adding bit to page table to indicate “non-executable” state
  - Available in SPARC and x86
  - But still have security exploits



# Program Threats (cont.)

---

## ■ Viruses

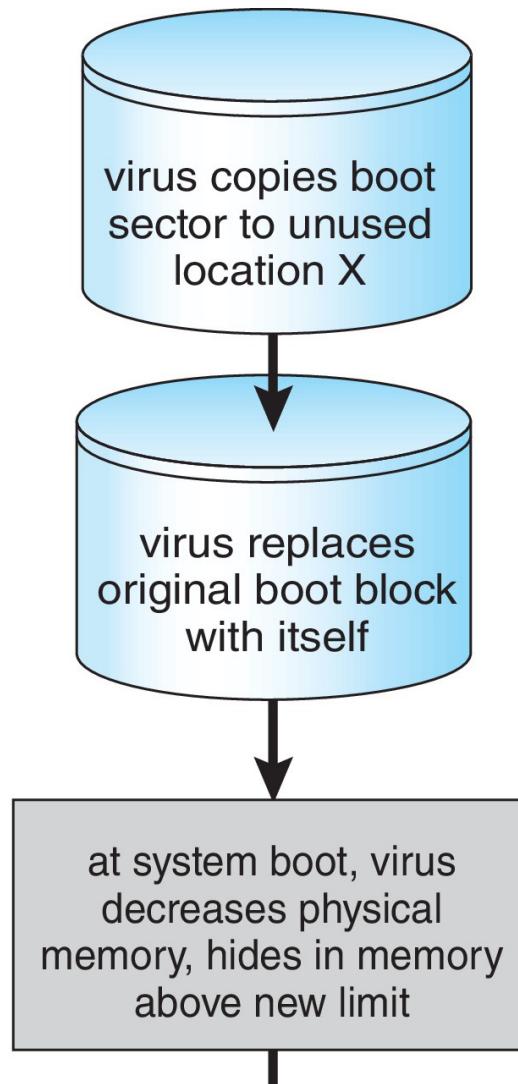
- Code fragment embedded in legitimate program
- Self-replicating, designed to infect other computers
- Very specific to CPU architecture, operating system, applications
- Usually borne via email or as a macro
- Visual Basic Macro to reformat hard drive

## ■ Virus dropper inserts virus onto the system

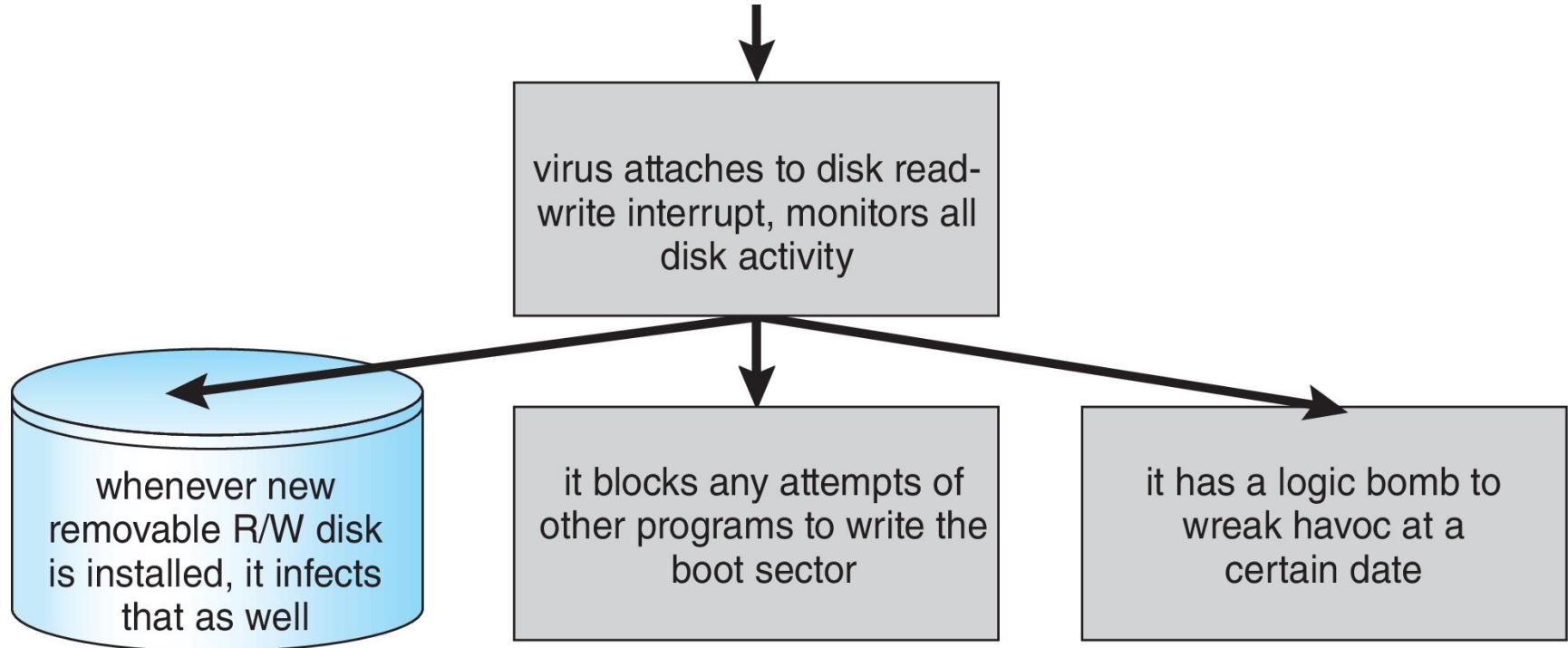


# A Boot-sector Computer Virus

---



# A Boot-sector Computer Virus



# The Threat Continues

---

- Attacks still common, still occurring
- Attacks moved over time from science experiments to tools of organized crime
  - Targeting specific companies
  - Creating botnets to use as tool for spam and DDOS delivery
  - **Keystroke logger** to grab passwords, credit card numbers



# The Threat Continues

---

- Why is Windows the target for most attacks?
  - Most common
  - Everyone is an administrator
  - **Monoculture** considered harmful
    - ▶ Many systems run the same hardware, operating system, and application software.



# System and Network Threats

---

- Network threats harder to detect, prevent
  - Protection systems weaker
  - More difficult to have a shared secret on which to base access
  - No physical limits once system attached to internet
    - ▶ Or on network with system attached to internet
  - Even determining location of connecting system difficult
    - ▶ IP address is only knowledge



# System and Network Threats (cont.)

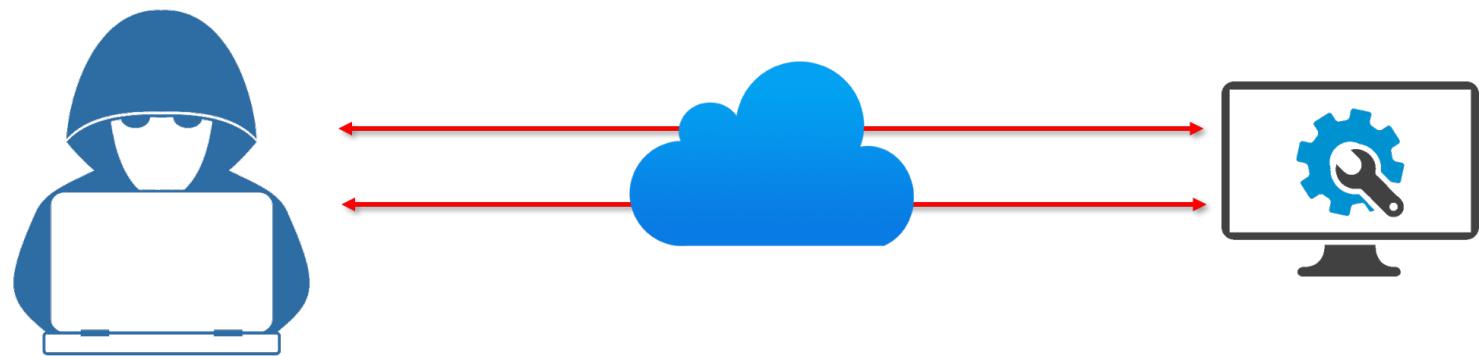
---

- Worms – use **spawn** mechanism; standalone program
- Internet worm
  - Exploited UNIX networking features (remote access) and bugs in *finger* and *sendmail* programs
  - Exploited trust-relationship mechanism used by *rsh* to access friendly systems without use of password

# System and Network Threats (cont.)

## ■ Port scanning

- Automated attempt to connect to a range of ports on one or a range of IP addresses
- Detection of answering service protocol
- Detection of OS and version running on system
- ...



# System and Network Threats (cont.)

---

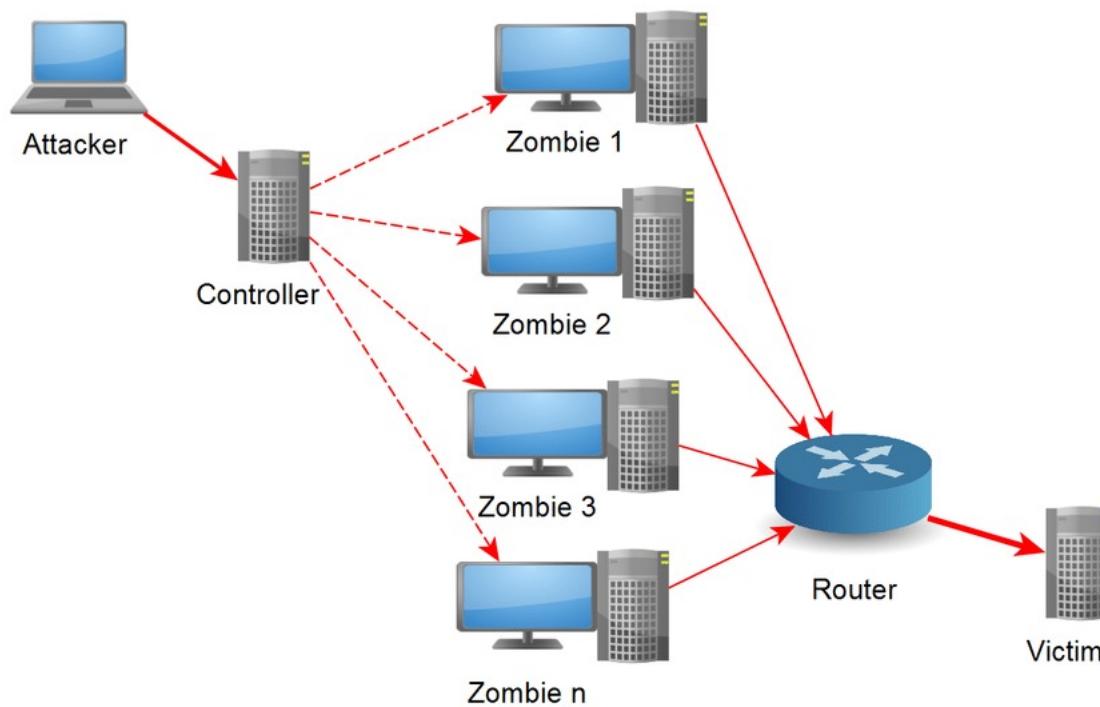
## ■ Port scanning

- ....
- nmap scans all ports in a given IP range for a response
- nessus has a database of protocols and bugs (and exploits) to apply against a system
- Frequently launched from **zombie systems**
  - ▶ To decrease trace-ability

# System and Network Threats (cont.)

## ■ Denial of Service

- Overload the targeted computer preventing it from doing any useful work
- **Distributed Denial-of-Service (DDoS)** come from multiple sites at once
- ...



# System and Network Threats (cont.)

---

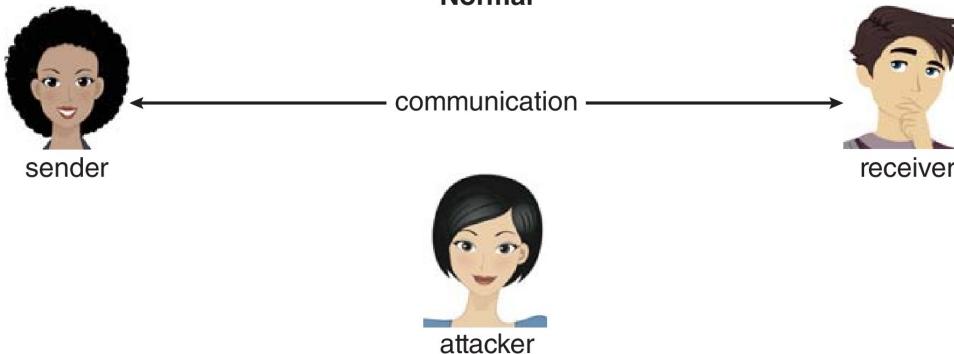
## ■ Denial of Service

- Consider the start of the IP-connection handshake (SYN)
  - ▶ How many started-connections can the OS handle?
- Consider traffic to a web site
  - ▶ How can you tell the difference between being a target and being really popular?
- Accidental – CS students writing bad `fork()` code
- Purposeful – extortion, punishment

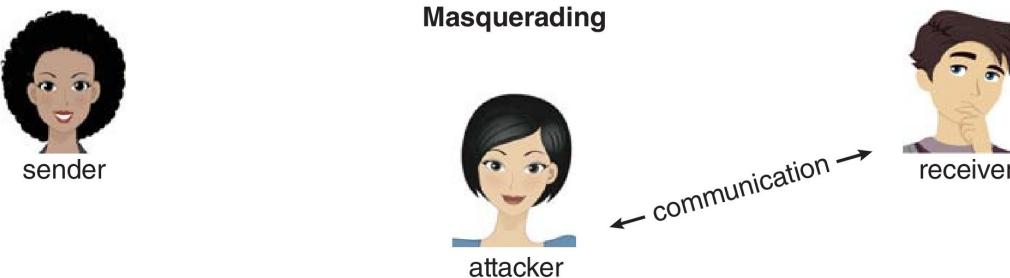


# Standard Security Attacks

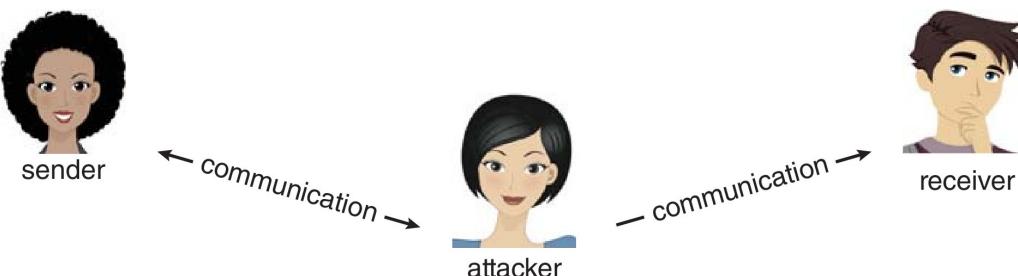
Normal



Masquerading



Man-in-the-middle





# **Operating Systems**

## **Security-Part2**

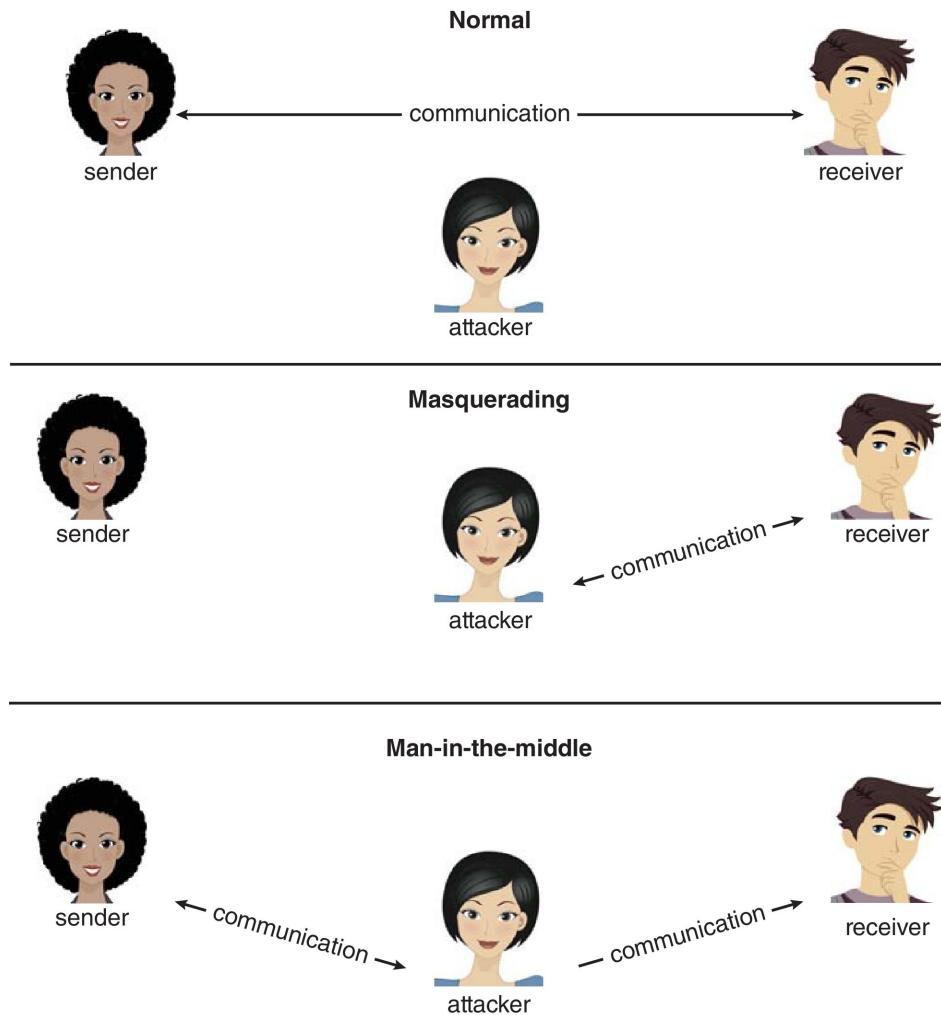
Seyyed Ahmad Javadi

[sajavadi@aut.ac.ir](mailto:sajavadi@aut.ac.ir)

Fall 2021

# Standard Security Attacks

---



# Cryptography

---

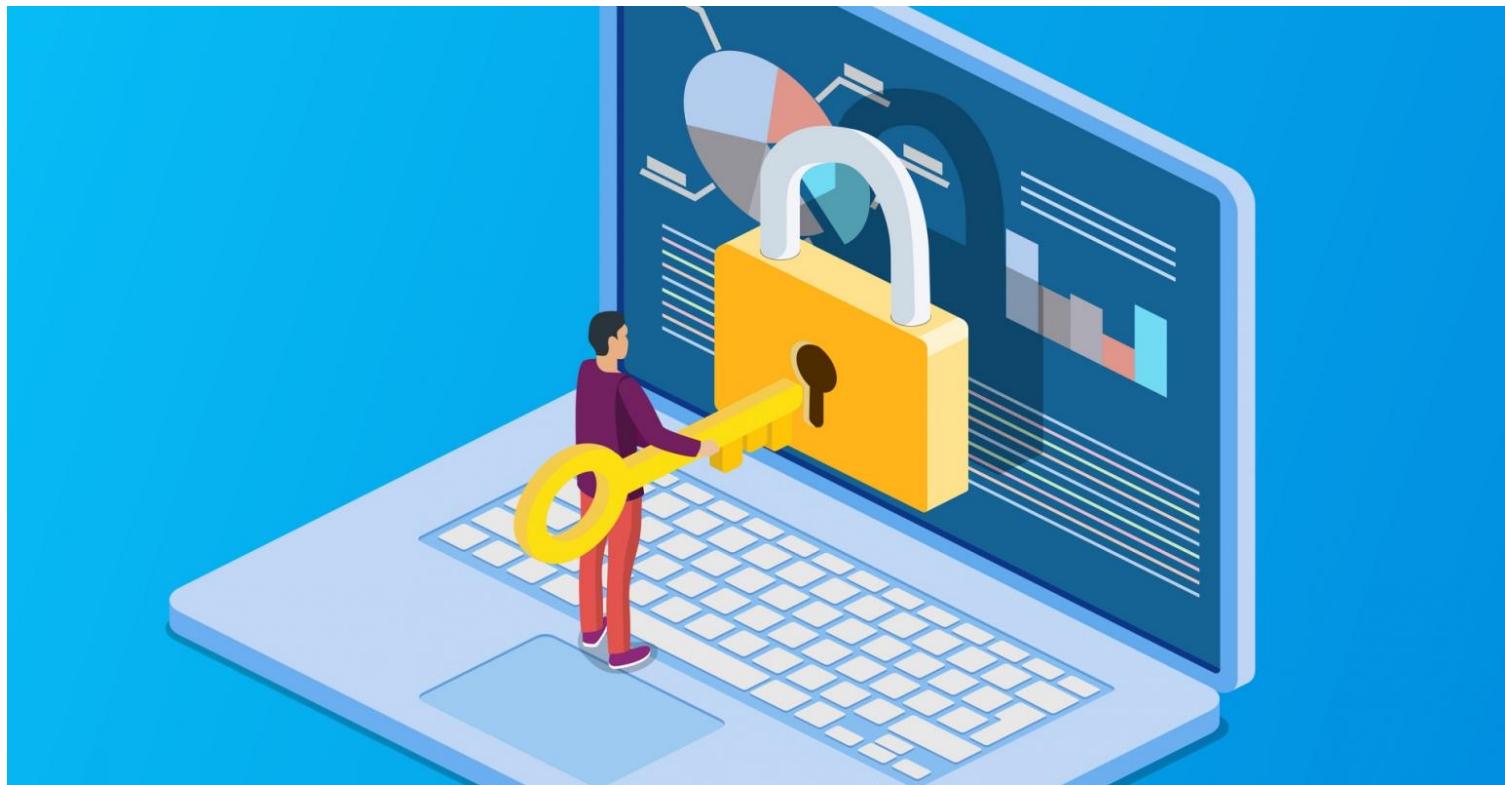
- Means to constrain potential senders (*sources*) and / or receivers (*destinations*) of messages
  - Based on secrets (**keys**)
  - Enables
    - ▶ Confirmation of source
    - ▶ Receipt only by certain destination
    - ▶ Trust relationship between sender and receiver



# Encryption

---

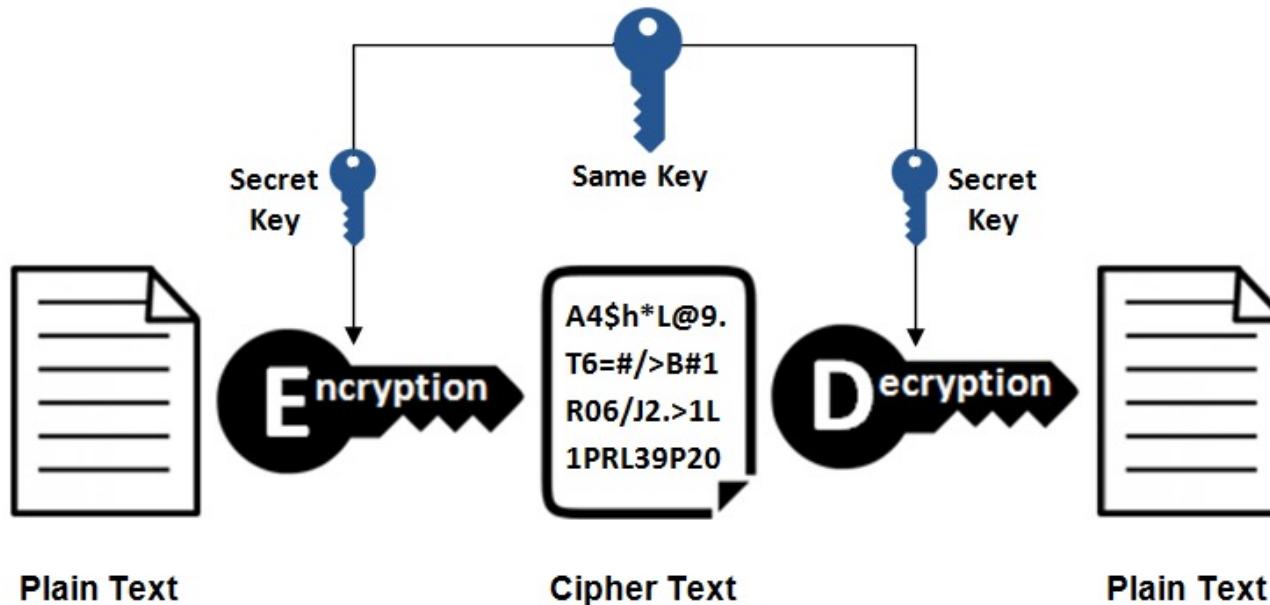
- Constrains the set of possible receivers of a message



# Symmetric Encryption

- Same key used to encrypt and decrypt
  - Therefore  $k$  must be kept secret

## Symmetric Encryption



# Symmetric Encryption (cont.)

---

- DES was most commonly used symmetric block-encryption algorithm (created by US Govt)
  - Encrypts a block of data at a time
  - Keys too short so now considered insecure
- 2001 NIST adopted new block cipher
- Advanced Encryption Standard (**AES**)
  - Keys of 128, 192, or 256 bits, works on 128 bit blocks

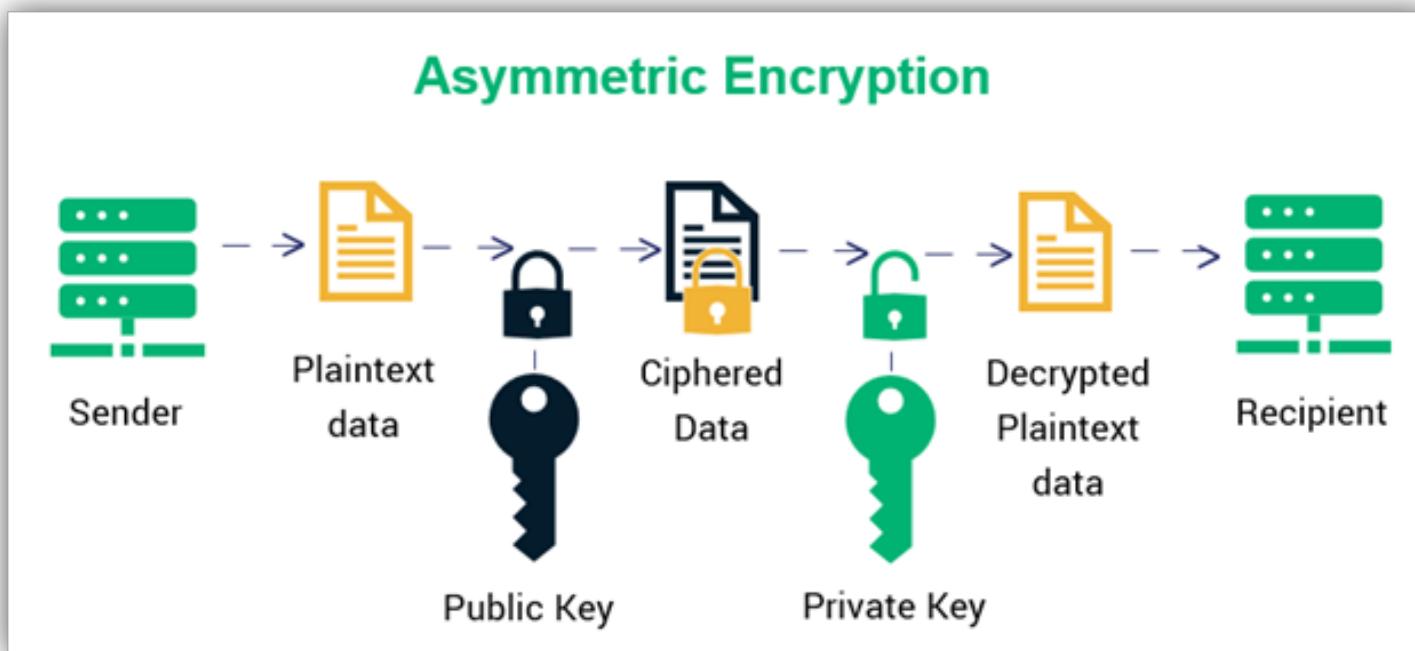
des is not secure



Symmetric --> one key asymmetric

# Asymmetric Encryption

- **Public-key encryption** based on each user having two keys:
  - **public key** – published key used to encrypt data
  - **private key** – key known only to individual user used to decrypt data



# Cryptography (cont.)

---

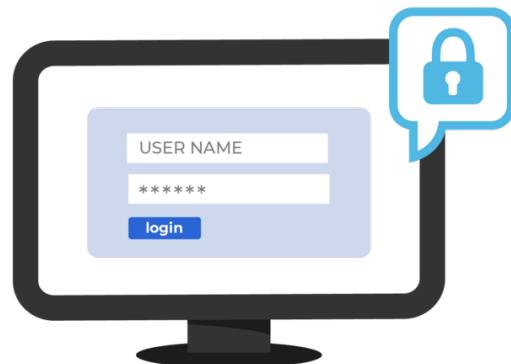
- Asymmetric much more compute intensive
  - Typically not used for bulk data encryption
  - Typically used for secure key exchange
- Symmetric encryption is used for bulk data encryption



# User Authentication

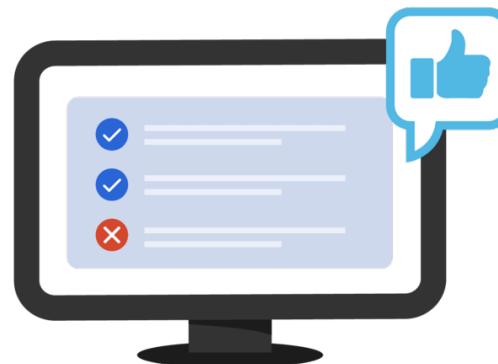
- Crucial to identify user correctly, as protection systems depend on user ID
- User identity most often established through **passwords**, can be considered a special case of either keys or capabilities

## Authentication



Confirms users are who they say they are.

## Authorization



Gives users permission to access a resource.

# User Authentication

---

- Passwords must be kept secret
  - Frequent change of passwords
  - History to avoid repeats
  - Use of “non-guessable” passwords
  - Log all invalid access attempts (but not the passwords themselves)
  - Unauthorized transfer

# User Authentication

---

- Passwords may also either be encrypted or allowed to be used only once
  - Does encrypting passwords solve the exposure problem?
    - ▶ Might solve **sniffing**
    - ▶ Consider **shoulder surfing**
    - ▶ Consider Trojan horse keystroke logger
    - ▶ How are passwords stored at authenticating site?



# Passwords

---

- Encrypt to avoid having to keep secret
  - But keep secret anyway (i.e. Unix uses superuser-only readable file /etc/shadow)
  - Use algorithm easy to compute but difficult to invert
  - Only encrypted password stored, never decrypted
  - Add “salt” to avoid the same password being encrypted to the same value

# Passwords

---

- One-time passwords
  - Use a function based on a seed to compute a password, both user and computer
  - Hardware device / calculator / key fob to generate the password
    - ▶ Changes very frequently



ART: MYKYTA/ADORE STOCK  
©2019 TECHTARGET. ALL RIGHTS RESERVED

# Passwords

---

- Biometrics
  - Some physical attribute (fingerprint, hand scan)
- Multi-factor authentication
  - Need two or more factors for authentication
    - ▶ i.e., USB “dongle”, biometric measure, and password

# Passwords (cont.)

---

## *STRONG AND EASY TO REMEMBER PASSWORDS*

It is extremely important to use strong (hard to guess and hard to shoulder surf) passwords on critical systems like bank accounts. It is also important to not use the same password on lots of systems, as one less important, easily hacked system could reveal the password you use on more important systems. A good technique is to generate your password by using the first letter of each word of an easily remembered phrase using both upper and lower characters with a number or punctuation mark thrown in for good measure. For example, the phrase “My girlfriend’s name is Katherine” might yield the password “Mgn.isK!”. The password is hard to crack but easy for the user to remember. A more secure system would allow more characters in its passwords. Indeed, a system might also allow passwords to include the space character, so that a user could create a **passphrase** which is easy to remember but difficult to break.



# Implementing Security Defenses

---

- **Defense in depth** is most common security theory – multiple layers of security
- **Security policy** describes what is being secured
- Vulnerability assessment compares real state of system / network compared to security policy



# Implementing Security Defenses

---

- Intrusion detection endeavors to detect attempted or successful intrusions
  - **Signature-based** detection spots known bad patterns
  - **Anomaly detection** spots differences from normal behavior
- ▶ Can detect **zero-day** attacks
- **False-positives** and **false-negatives** a problem

signature-based detection is not secure under zero-day attack

# Implementing Security Defenses

---

- Virus protection
  - Searching all programs or programs at execution for known virus patterns
  - Or run in **sandbox** so can't damage system
- Auditing, accounting, and logging of all or specific system or network activities
- Practice **safe computing** – avoid sources of infection, download from only “good” sites, etc

does not for zero-day

# Firewalling to Protect Systems and Networks

---

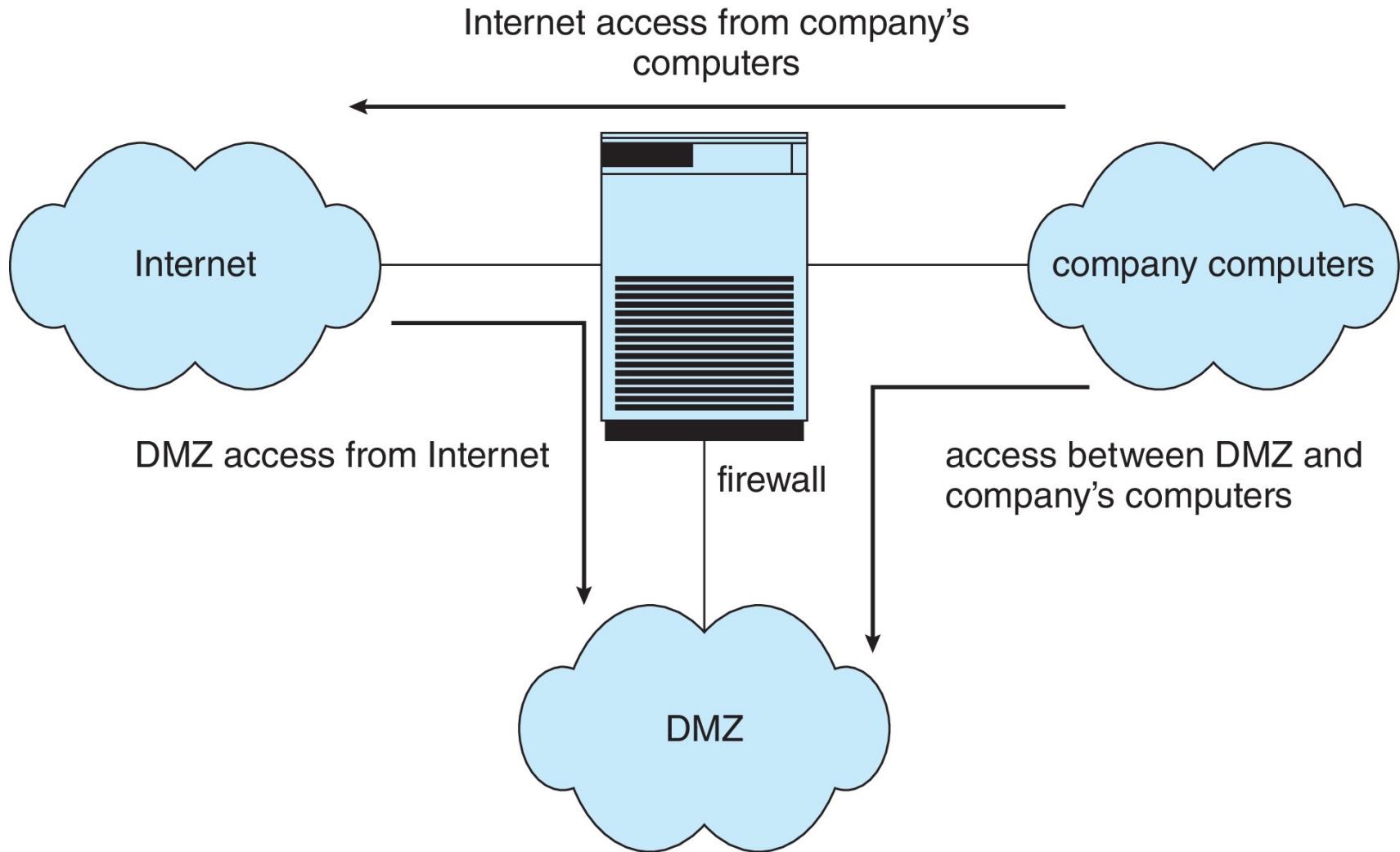
- A network **firewall** is placed between trusted and untrusted hosts
  - The firewall limits network access between these two **security domains**
- Can be tunneled or spoofed
  - Tunneling allows disallowed protocol to travel within allowed protocol (i.e., telnet inside of HTTP)
  - Firewall rules typically based on host name or IP address which can be spoofed

# Firewalling to Protect Systems and Networks

---

- **Personal firewall** is software layer on given host
  - Can monitor / limit traffic to and from the host
- **Application proxy firewall** understands application protocol and can control them (i.e., SMTP)
- **System-call firewall** monitors all important system calls and apply rules to them (i.e., this program can execute that system call)

# Network Security Through Domain Separation Via Firewall



# Security Defenses Summarized

---

- By applying appropriate layers of defense, we can keep systems safe from all but the most persistent attackers.
- In summary, these layers may include the following:
  - Educate users about safe computing
    - ▶ Don't attach devices of unknown origin to the computer
    - ▶ Don't share passwords
    - ▶ Use strong passwords
    - ▶ Avoid falling for social engineering appeals
    - ▶ Realize that an e-mail is not necessarily a private communication, and so on

# Security Defenses Summarized

---

- In summary, these layers may include the following:
  - Educate users about how to prevent phishing attacks
    - ▶ Don't click on email attachments or links from unknown (or even known) senders
    - ▶ Authenticate (for example, via a phone call) that a request is legitimate



# Security Defenses Summarized

---

- In summary, these layers may include the following:
  - Use secure communication when possible
  - Physically protect computer hardware
  - Configure the operating system to minimize the attack surface; disable all unused services
  - Configure system daemons, privileges applications, and services to be as secure as possible



# Security Defenses Summarized (cont.)

---

- Use modern hardware and software, as they are likely to have up-to-date security features
- Keep systems and applications up to date and patched
- Only run applications from trusted sources (such as those that are code signed)
- Enable logging and auditing; review the logs periodically, or automate alerts

# Security Defenses Summarized (cont.)

---

- Install and use antivirus software on systems susceptible to viruses, and keep the software up to date
- Use strong passwords and passphrases, and don't record them where they could be found
- Use intrusion detection, firewalling, and other network-based protection systems as appropriate
- For important facilities, use periodic vulnerability assessments and other testing methods to test security and response to incidents



# Security Defenses Summarized (cont.)

---

- Encrypt mass-storage devices, and consider encrypting important individual files as well
- Have a security policy for important systems and facilities, and keep it up to date

