UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity.
Dennis Ritchie

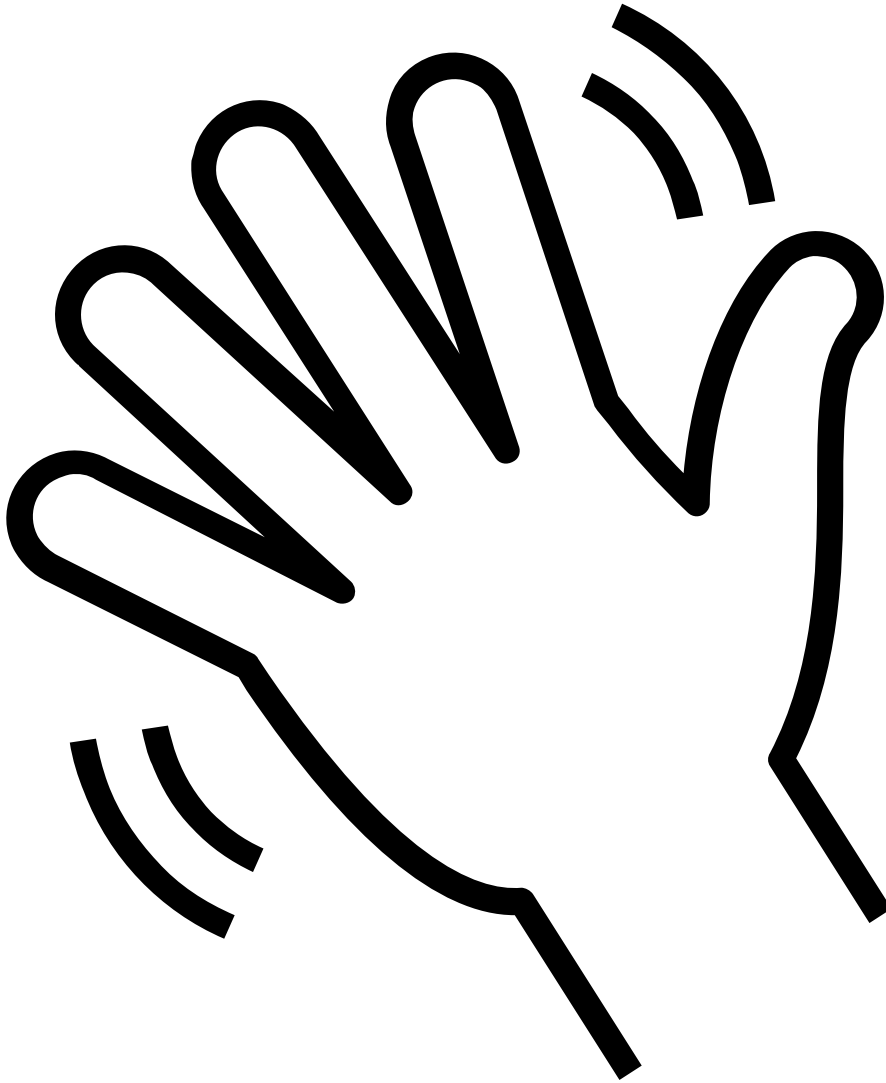
One of my big regrets is that Facebook hasn't had a major chance to shape the mobile operating system ecosystem.
Mark Zuckerberg

Welcome to COEN 346 Operating Systems

Instructor: Paula Lago
paula.lago@concordia.ca

Hello!

- Paula Lago
- PhD in Engineering, Software Engineering background
- Lived in Colombia, US, France, Japan, Canada.
- I work with wearables and smart home sensor data to understand human behaviors and their relations to health. For example, using behavior changes to predict disease.
- Operating Systems I use the most: MacOS and Android



What is this course about?

Operating systems **provide an interface to communicate with the computer without learning the computer's language i.e. machine language**.

At the end of the course, you will be able to understand the simplicity behind operating systems design and how to use those principles for programming real world applications

Why is this (one of) the most important course(s) of my program?

Simply because, as almost all code runs on top of an operating system, knowledge of how operating systems work is crucial to proper, efficient, effective, and secure programming.

Understanding the fundamentals of operating systems, how they drive computer hardware, and what they provide to applications is not only essential to those who program them but also highly useful to those who write programs on them and use them.

Let’s get to know each other

OK, but how are we going to be evaluated?

- Quizzes (5%)
  - In class or online
  - Readings and Basic concepts
- Assignments (10%)
  - Theory
  - Short exercises (in class or to give next class)
- Programming assignments and Laboratories (25%)
  - Work with TA during tutorial
- Mid-term (20%)
- Final Exam (40%)
  - Cumulative
  - Involves theory and programming

Late submission policy

- 4 late days to use at your disposition
- No partial days
  - One second late means you used one late day
- After you used all late days, you will not be graded on late submissions (that means you failed that assignment)
- If you have major circumstances that require more than 4 late days, please talk to me

Letter grades and passing criteria

Final Score	Grade
95 – 100	A+
90 – 94.99	A
86 - 89.99	A-
82 – 85.99	B+
79 – 81.99	B
76 – 78.99	B-
72 – 75.99	C+
69 – 71.99	C
66 – 68.99	C-
62 – 65.99	D+
59 – 61.99	D
55 – 58.99	D-
0 – 54.99	F

- If your total score before the final exam is less than 40% and you decide to defer the final exam, you will receive an ***R*** grade which prevents you to defer the final exam.
- To pass the class, your cumulative score, and the final examination must be above 50%. This will not affect your final grade unless either one is below the passing score.

Close is not enough

- No approximations!

94.99999 -> A
95 -> A+

54.9999 -> F

What are we going to learn?

- *By the end of this course, you will be able to:*

1. Describe the fundamental principles and concepts of operating systems, including processes, threads, concurrency, synchronization, and scheduling, as well as the evolution and architecture of modern operating systems (ECE-KB-3)
2. Explain the role of an operating system in managing hardware and software resources, including memory, storage, and I/O devices (ECE-KB-3)
3. Analyze and compare different operating system designs and their trade-offs, including traditional monolithic systems, microkernels, and virtualization. (ECE-DE-2)
4. Design and implement simple operating system components, such as a scheduler or a memory management system, using appropriate data structures and algorithms. (ECE-DE-1 and ECE-DE-3)
5. Evaluate the performance and efficiency of an operating system using analytical models and simulation techniques. (ECE-DE-4)
6. Apply their knowledge of operating systems to solve a small real-world problem (ECE-DE-1, ECE-DE-2, ECE-DE-3, ECE-DE-4)

What are we going to learn?

- The evolution, architecture, and use of modern operating systems (OS).
- Multi-tasking, concurrency and synchronization, Inter process Communication (IPC),
- deadlock, resource allocation, scheduling, multi-threaded programming,
- memory and storage managements, file systems, I/O techniques, buffering, protection and security,
- the client/server paradigm and communications.
- Introduction to real time operating systems.

Course Rules

Mainly two

- Respect

- Good Communication

Communication, communication, communication

- With your team members
  - Inform with time if you will not be able to meet your tasks
  - Do not disappear, answer messages, be clear about your available time (holidays, exams, etc.)
- With your instructor (aka me)
  - Inform if you will not come to lecture, if you observe any holiday
  - If you need any accomodation
  - Email if you want to come to office
- With your TA
  - They have been here before!
- With your peers
  - You can discuss topics, but do not share answers

Communication, communication, communication

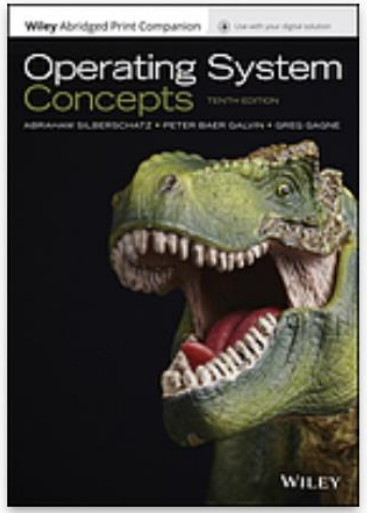
- Communication channels
  - Discord
  - Moodle
  - Email
- Moodle is the official channel
  - Lectures
  - Grades
  - Quizzes
  - etc

I expect respect

- Distractions
  - Digital communication is very distracting for you and those around you
  - If you need to, go outside
- Arriving to lecture
  - On time, or be quiet
- When communicating

Book

- Operating Systems Concepts



- Student companion site
- https://bcs.wiley.com/he-bcs/Books?action=index&itemId=1119320917&bcsId=11228
  - Practice exercise solutions
  - Source code
  - Linux Virtual Machine

Book

Concordia

Silberschatz

Sign in

arch Resources

Search history Saved Items (0)

le, Chapter (604)

Article (576)

Chapter (24)

Downloadable Article (4)

(35)

Print Book (23)

eBook (12)

ype

Text

Access

Fiction (639)

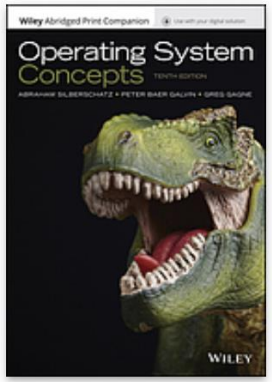
Reviewed (166)

n Year

Worldwide Editions and Formats View All (9)

2

Cite Share Save



Operating system concepts

Authors: Abraham Silberschatz (Author), Peter B Galvin (Author), Greg Gagne (Author)

Print Book 2018, Tenth edition.
Hoboken, NJ : Wiley, [2018]

Available

Concordia University Library, Webster Library - Course Reserve Room (3 hour loan)

QA 76.76 O63 S55825 2018 Locate

Other Editions and Formats at Concordia University Library View All (9)

View eBook

Older © 2008 eBook edition, 8th ed.
Details

Available

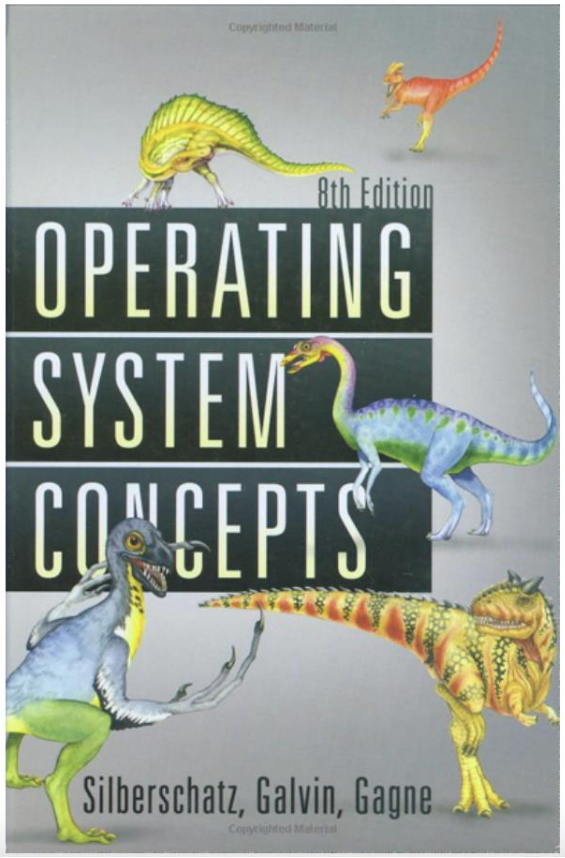
Report a Broken Link

Book

Operating System Concepts, 8th Edition

★★★★☆ 4 reviews

By ABRAHAM SILBERSCHATZ, PETER BAER GALVIN, GREG GAGNE



TIME TO COMPLETE:
33h 27m

TOPICS:
Operating Systems

PUBLISHED BY:
Wiley

PUBLICATION DATE:
July 2008

PRINT LENGTH:
992 pages

https://learning.oreilly.com/library/view/operating-system-concepts/9780470128725/

How to get an A+?

- Read the syllabus ☺ everything you need to know is there (including dates)
- Read before class
- Try it yourself at home
- Involve yourself in the programming assignments
- Do all quizzes and understand the answers
- Do the exercises consciously
- Find a good ***study*** group
- Ask if you don't understand!

What questions do you have?

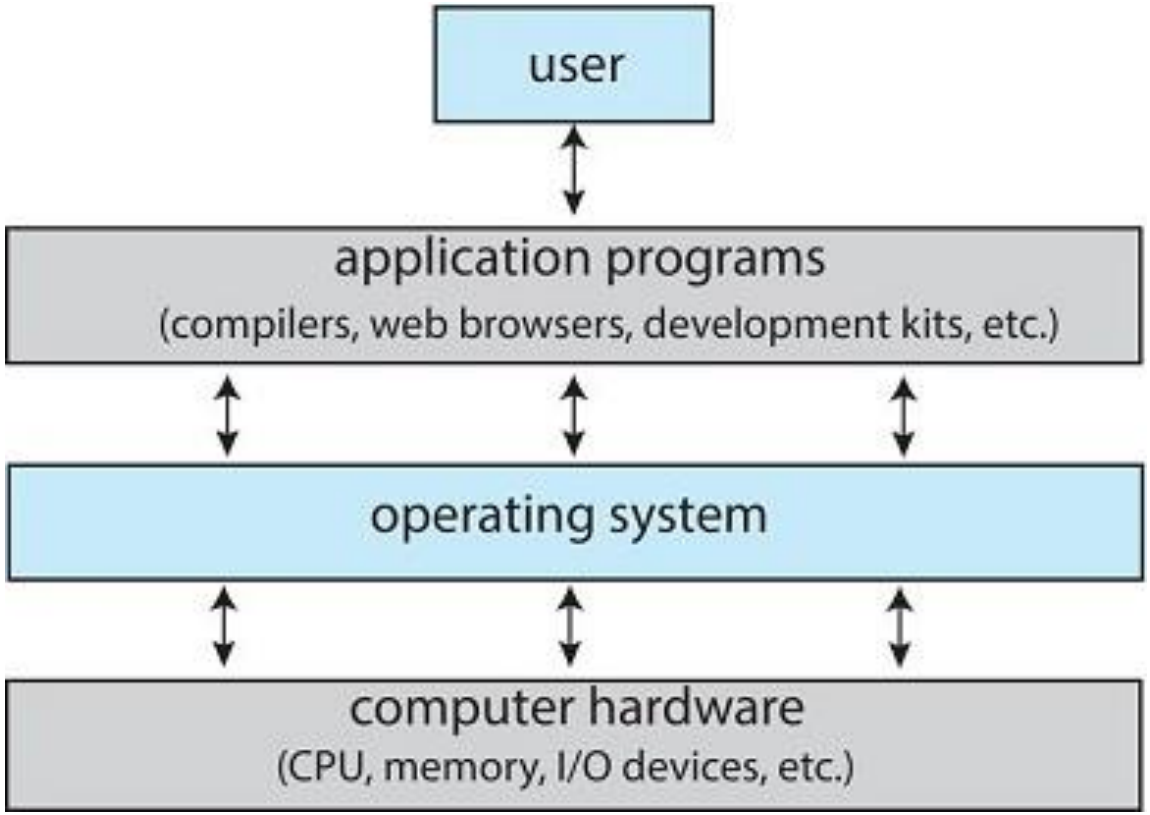
How many OS can you name?

Operating Systems

- Mainframe
  - OS/360
- Server OS
  - Windows Server, Linux
- Personal Computer OS
  - macOS, Windows 11
- Smartphone and handheld OS
  - Android, iOS
- IoT and Embedded Systems
  - TinyOS, embedOS
- Real Time OS
  - freeRTOS
- SmartCard OS
  - JVM

Let’s get started

What does an operating system do?



Abstract view of the components of a computer system.

Why use an Operating System?

- The operating system provides common operations that many user applications require to use the hardware
  - I/O managing
  - Memory management
  - Access to CPU

What is an operating system?

- The OS is the one program running at all times on the computer—usually called the ***kernel***.
- Along with the kernel, there are two other types of programs: ***system programs***, which are associated with the operating system but are not necessarily part of the kernel, and **application programs**, which include all programs not associated with the operation of the system.

- Mobile operating systems often include not only a core kernel but also **middleware**—a set of software frameworks that provide additional services to application developers. For example, each of the two most prominent mobile operating systems—Apple's IOS and Google's Android—features a core kernel along with middleware that supports databases, multimedia, and graphics (to name only a few).

In Summary

- The operating system is inclusive of the kernel that is loaded at boot time, any device drivers and kernel functions loaded at run time, and any system programs related to the operation of the system (as opposed to applications).
- The operating system is the piece of software that sits between the hardware and applications, providing a set of services to the computer and users, and is not part of the firmware or hardware logic.

Homework 0

- Syllabus Quiz on Moodle
- Make an appointment for office hours and come say hi!

COEN 346 Lecture 2: Introduction to Operating Systems

Paula Lago

paula.lago@concordia.ca

Agenda

1. What is an Operating System?
2. How does an operating system work?
3. How does the computer architecture impact the OS?

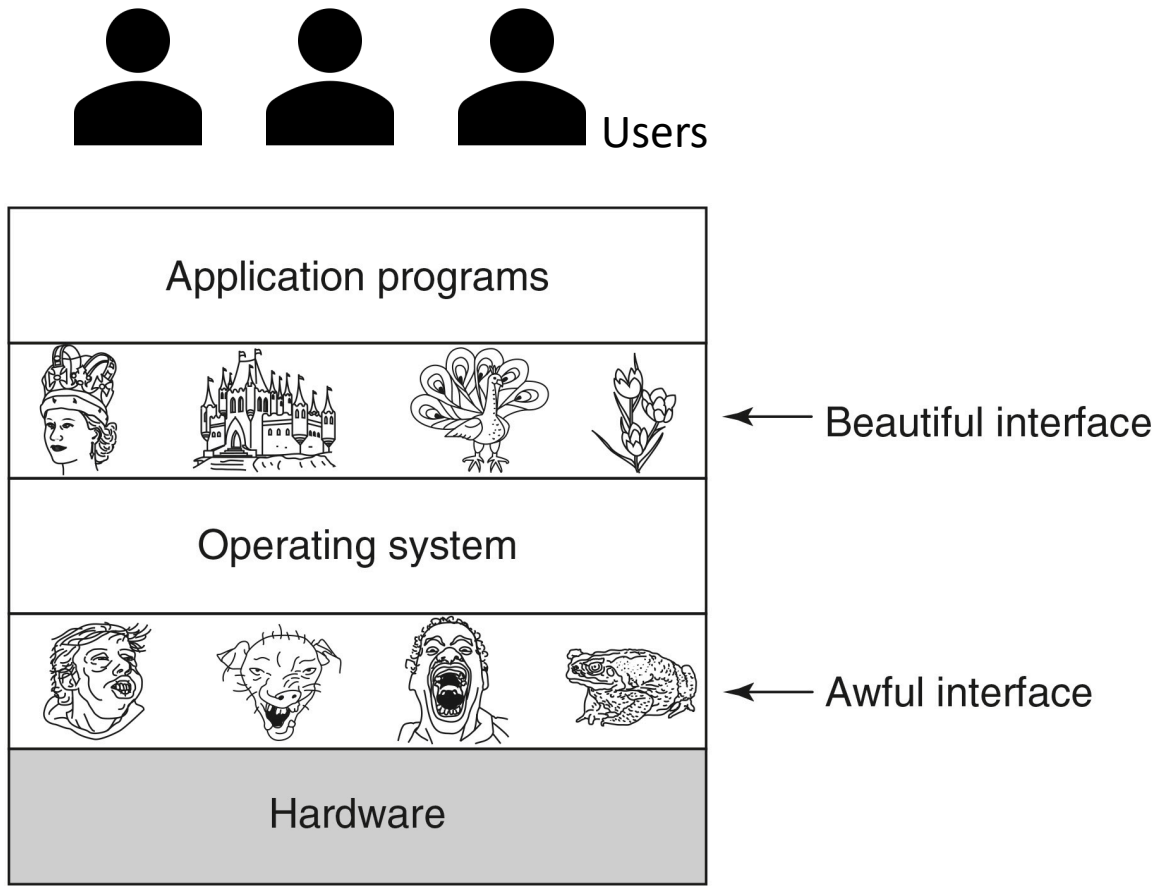
Copyright information

- Content and images from slides and by:

- Operating Systems Concepts by Silberchatz et al.
- Modern Operating Systems, by Tanenbaum et al

What is an Operating System?

- A program that acts as an intermediary between a user of a computer and the computer hardware
- 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

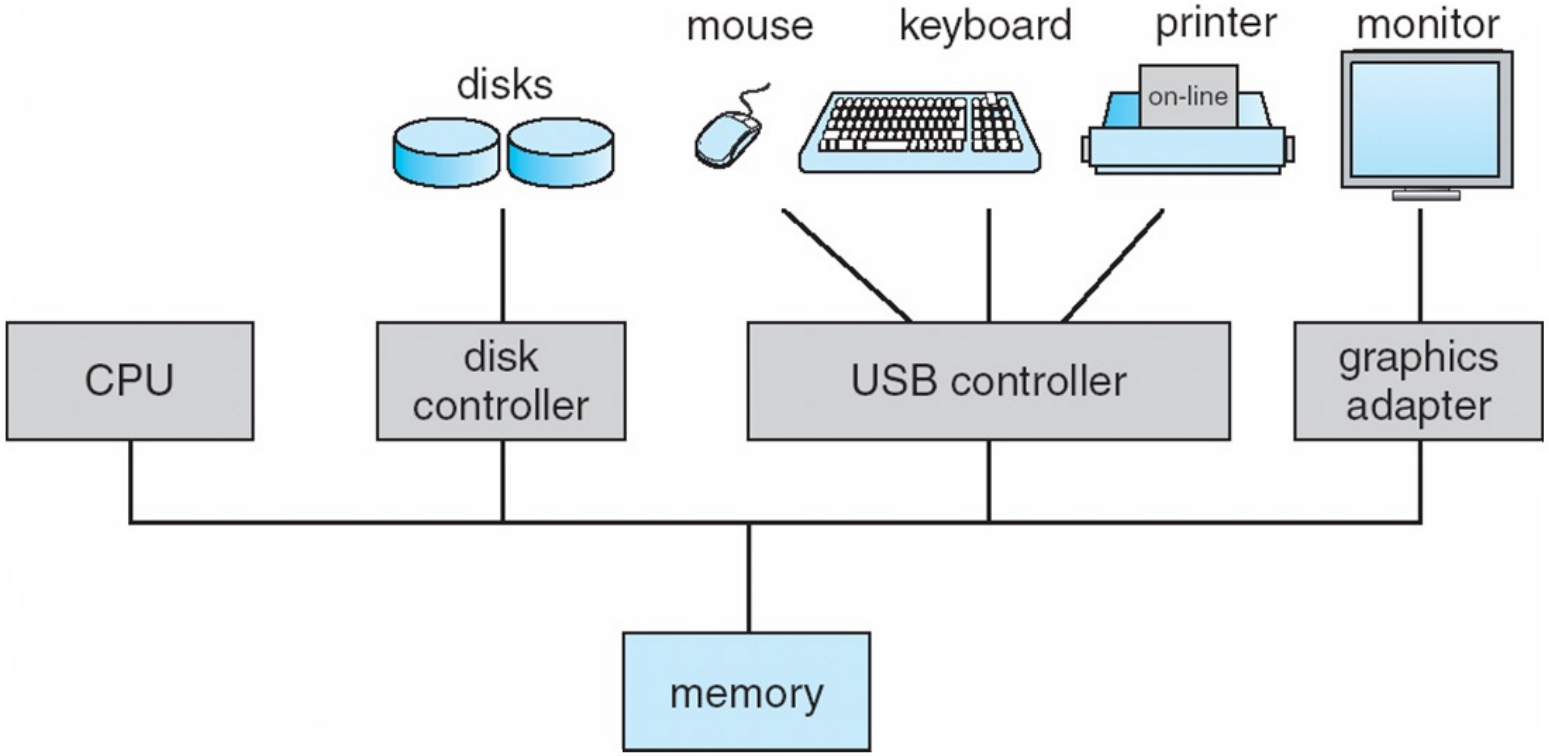


Defining operating systems is hard

- Kernel
  - Runs at all times
- System and application programs
- Middleware
  - Additional support for developers

An operating system manages the computer system

- CPU
- Storage (memory)
- I/O
  - Devices



What is an Operating System?

- OS is a **resource allocator**
  - Manages all resources
  - Decides between conflicting requests for efficient and fair resource use
- OS is a **control program**
  - Controls execution of programs to prevent errors and improper use of the computer

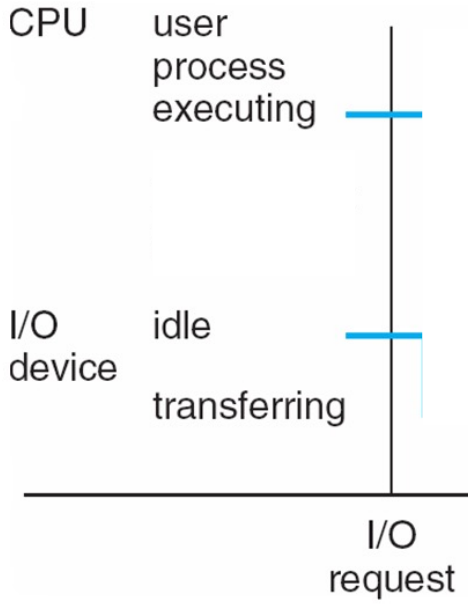
Resource management

- A program in execution is a process
- Processes need resources to complete their task
  - Memory
  - Files
  - I/O
- The operating system ensures fair use of the resources
  - Access control to files
  - Memory access protection
  - CPU time, etc

How does the operating system control resources?

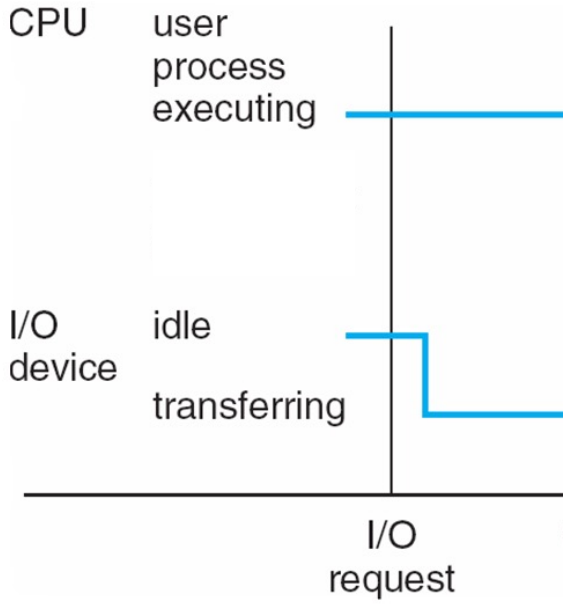
Interacting with I/O devices: the old way

- A user program requests a system call to use an i/o device



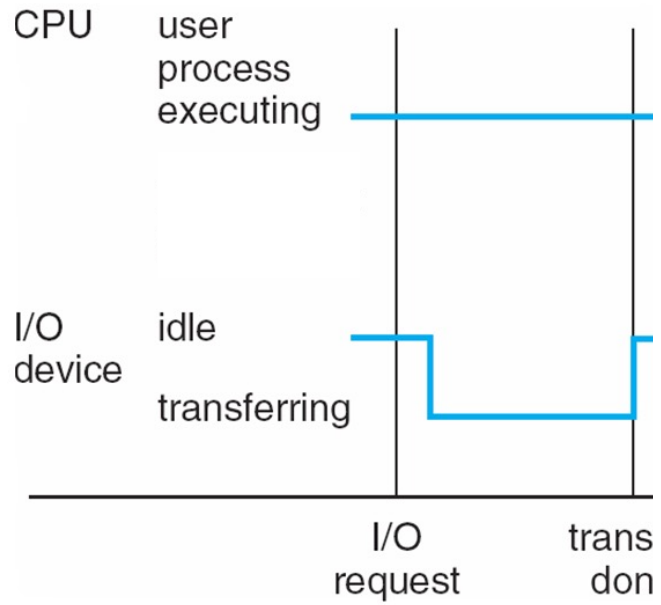
Interacting with I/O devices: the old way

- A user program requests a system call to use an i/o device
- The system calls the driver



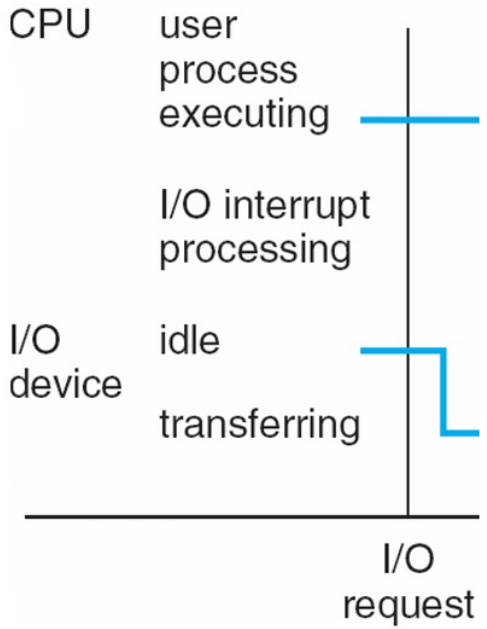
Interacting with I/O devices: the old way

- A user program requests a system call to use an i/o device
- The system calls the driver
- The driver starts the I/O device and continuously polls to see if it is done
- When the device tells the driver it is done, the driver puts the data where required and finishes



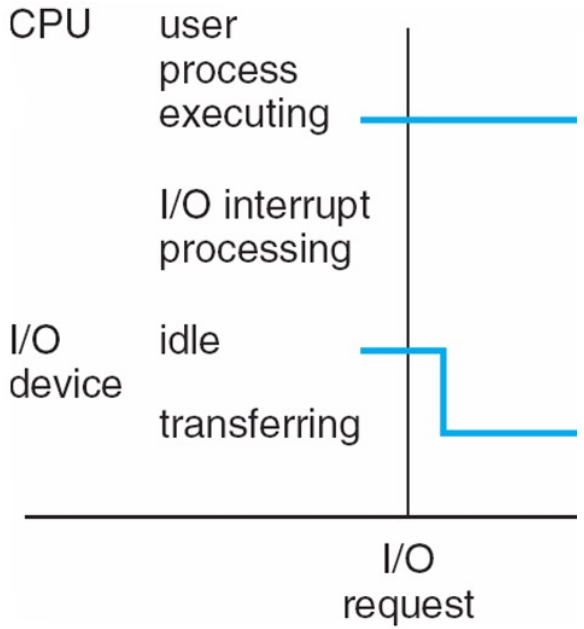
Interacting with I/O devices using INTERRUPTS

- A user program requests a system call to use an I/O device
- The system calls the driver
- The driver starts the I/O device and returns
- The CPU can look for another job to do



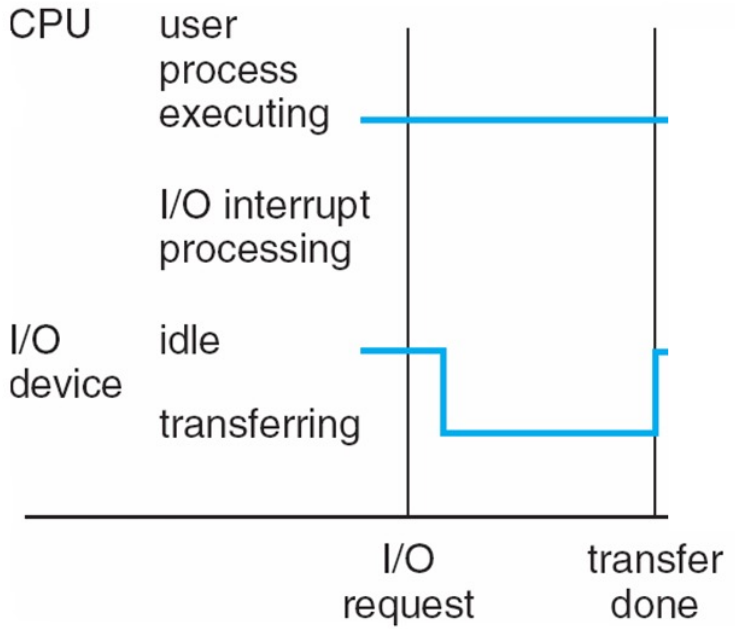
Interacting with I/O devices using INTERRUPTS

- A user program requests a system call to use an I/O device
- The system calls the driver
- The driver starts the I/O device and returns
- The CPU can look for another job to do
- When the device is done, it sends an INTERRUPT to signal it finished
- The CPU processes the data received



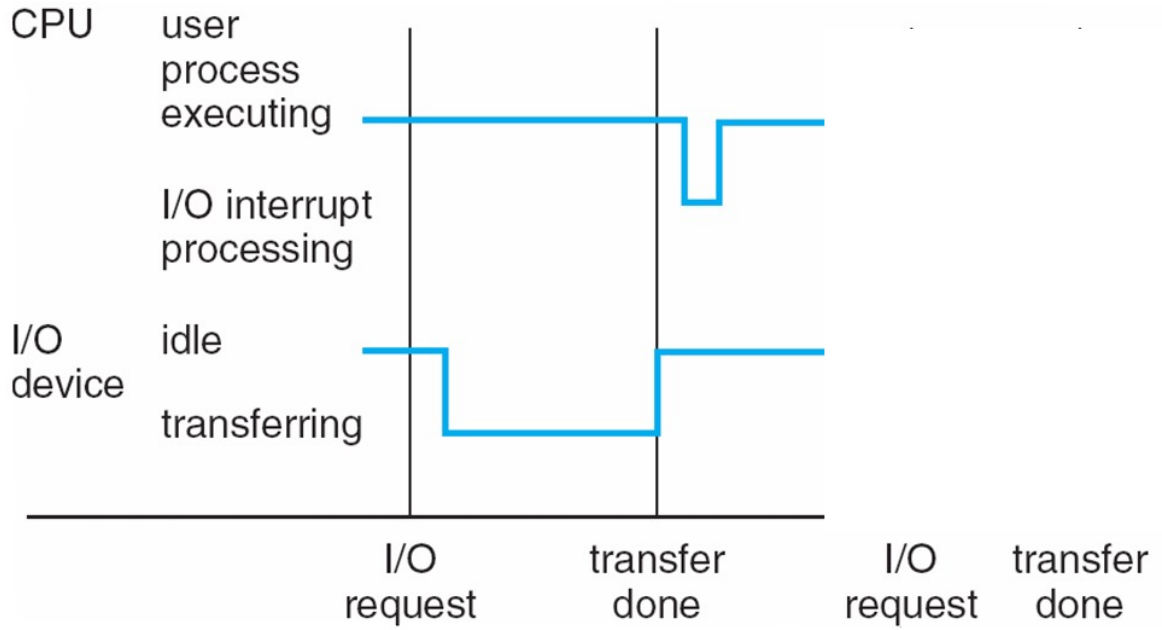
Interacting with I/O devices using INTERRUPTS

- A user program requests a system call to use an I/O device
- The system calls the driver
- The driver starts the I/O device and returns
- The CPU can look for another job to do
- When the device is done, it sends an INTERRUPT to signal it finished
- The CPU processes the data received



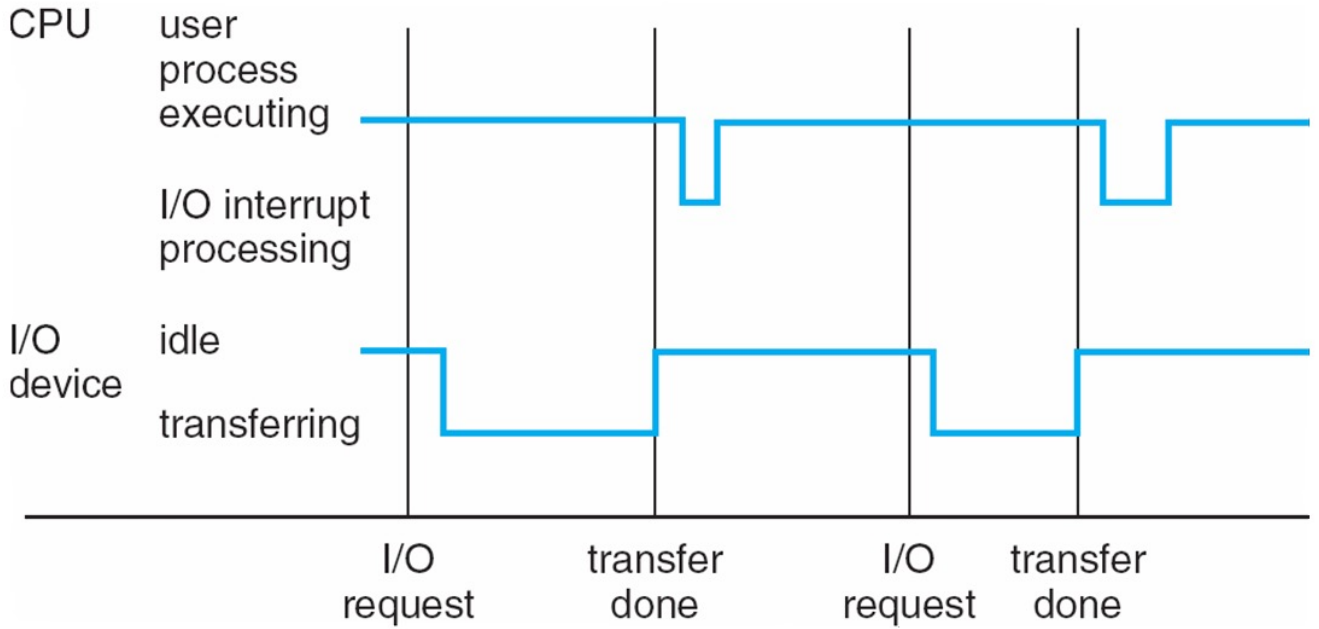
Interacting with I/O devices using INTERRUPTS

- A user program requests a system call to use an I/O device
- The system calls the driver
- The driver starts the I/O device and returns
- The CPU can look for another job to do
- When the device is done, it sends an INTERRUPT to signal it finished
- The CPU processes the data received



Interacting with I/O devices using INTERRUPTS

- A user program requests a system call to use an I/O device
- The system calls the driver
- The driver starts the I/O device and returns
- The CPU can look for another job to do
- When the device is done, it sends an INTERRUPT to signal it finished
- The CPU processes the data received

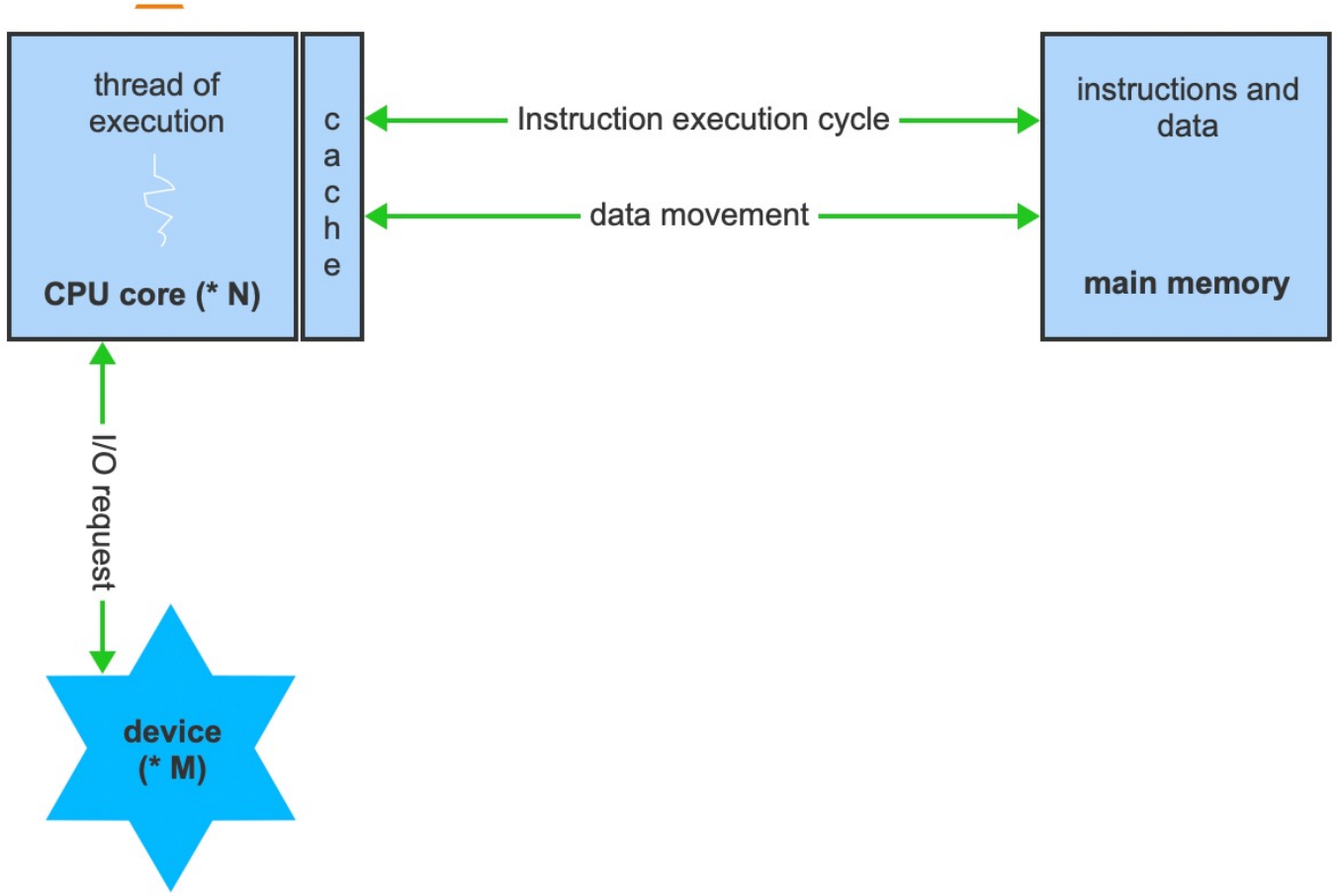


Not all interrupts are the same!

- Interrupt
- Trap
  - Error
  - System call
- Some interrupts require immediate action
- Some operations may not be interrupted
- Maskable vs non-maskable

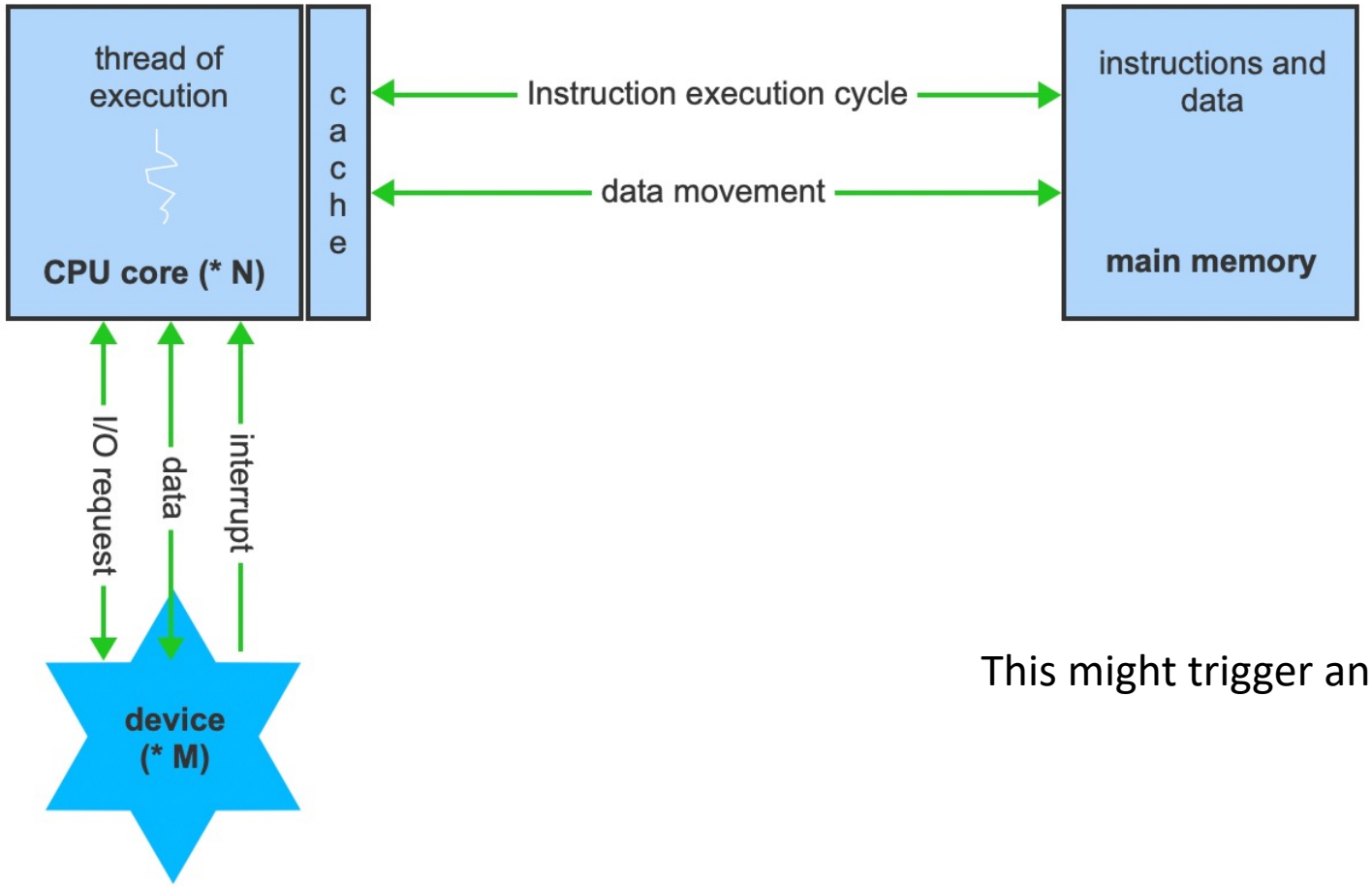
What questions do you have?

Managing I/O with larger data communication



An instruction might trigger an I/O request to a device.

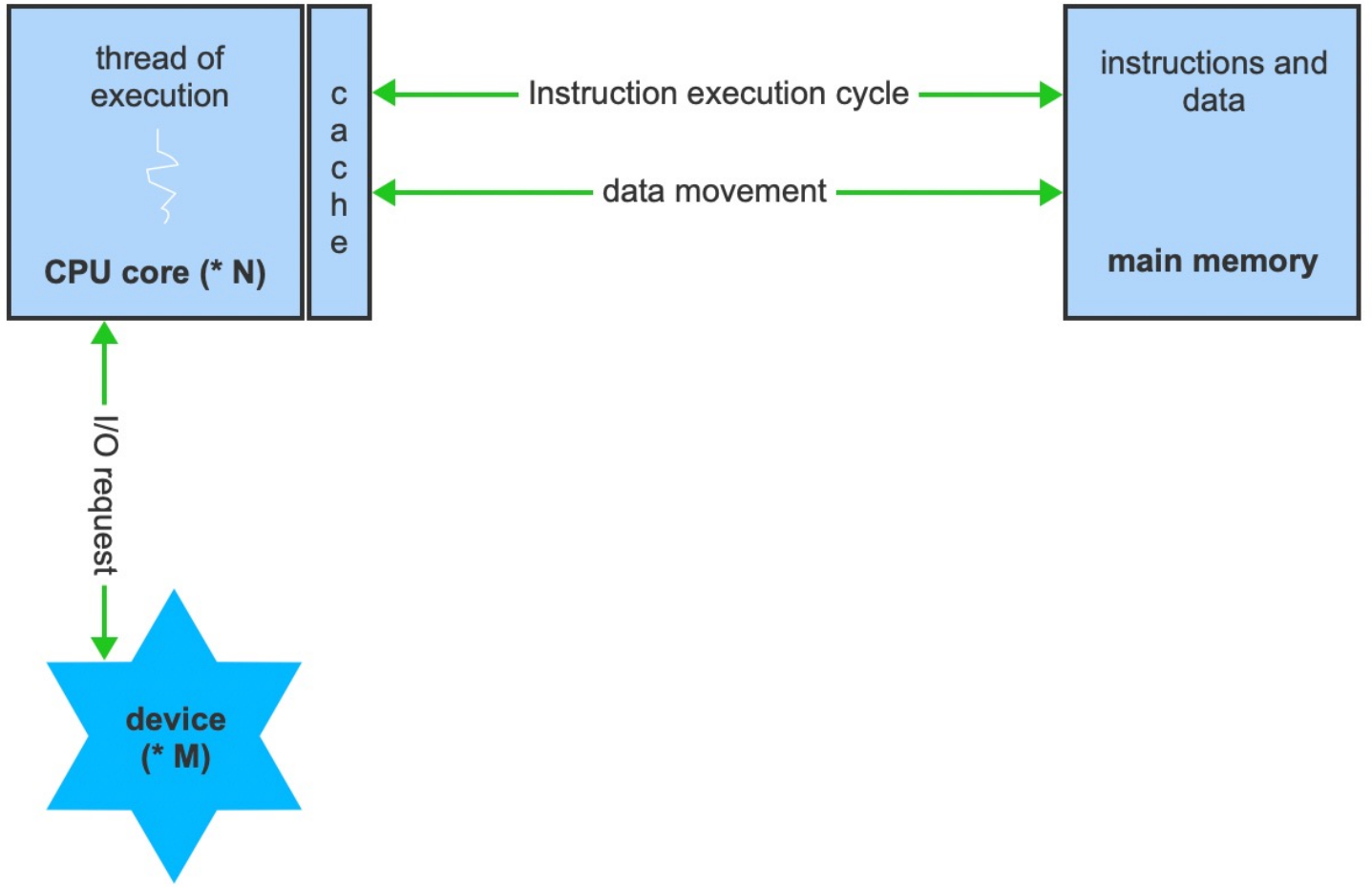
Managing I/O with larger data communication



This might trigger an interrupt per byte

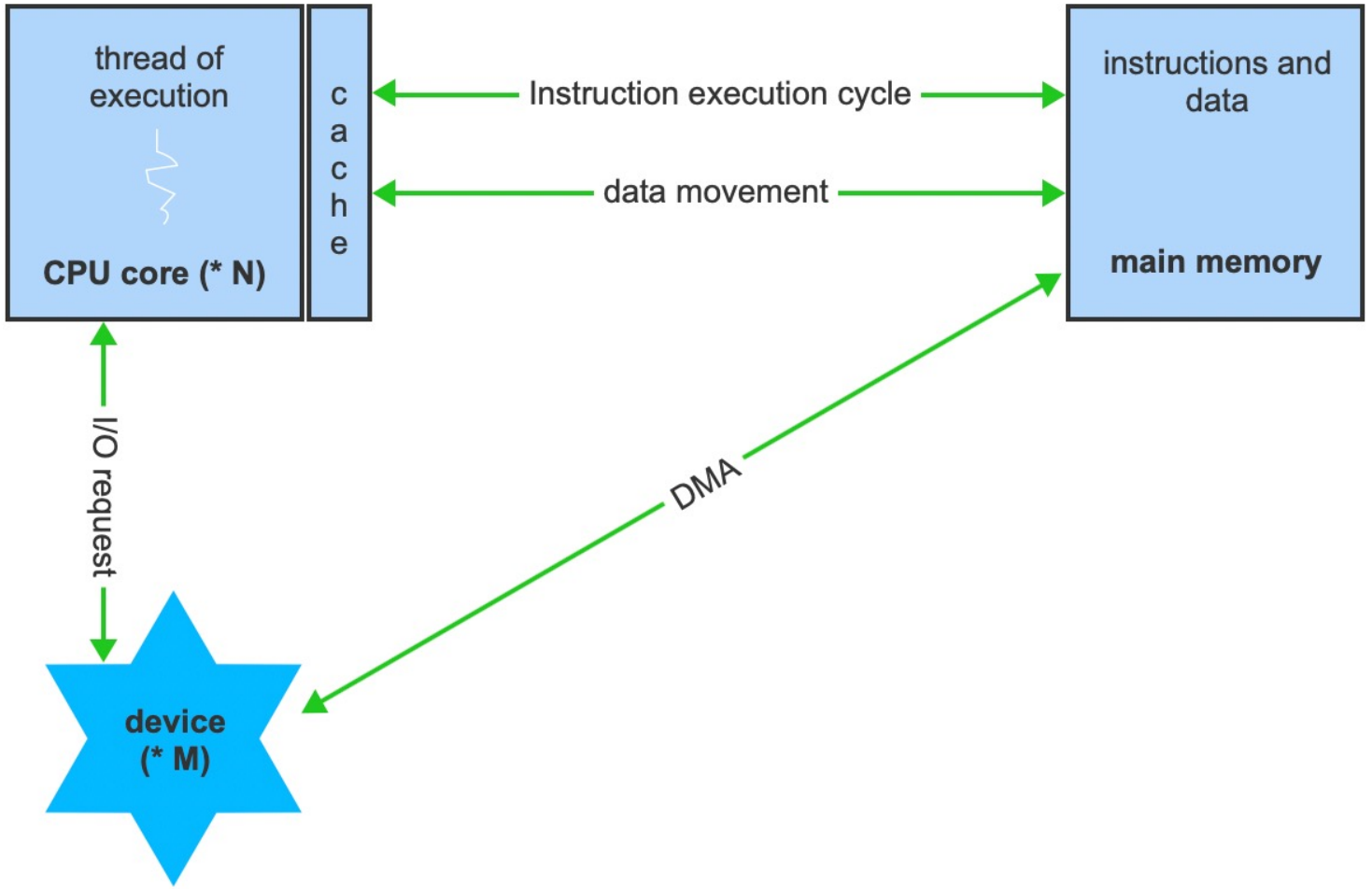
That request leads to data transfer with the device, and an interrupt to signal the CPU when the transfer is complete.

Managing I/O with larger data communication



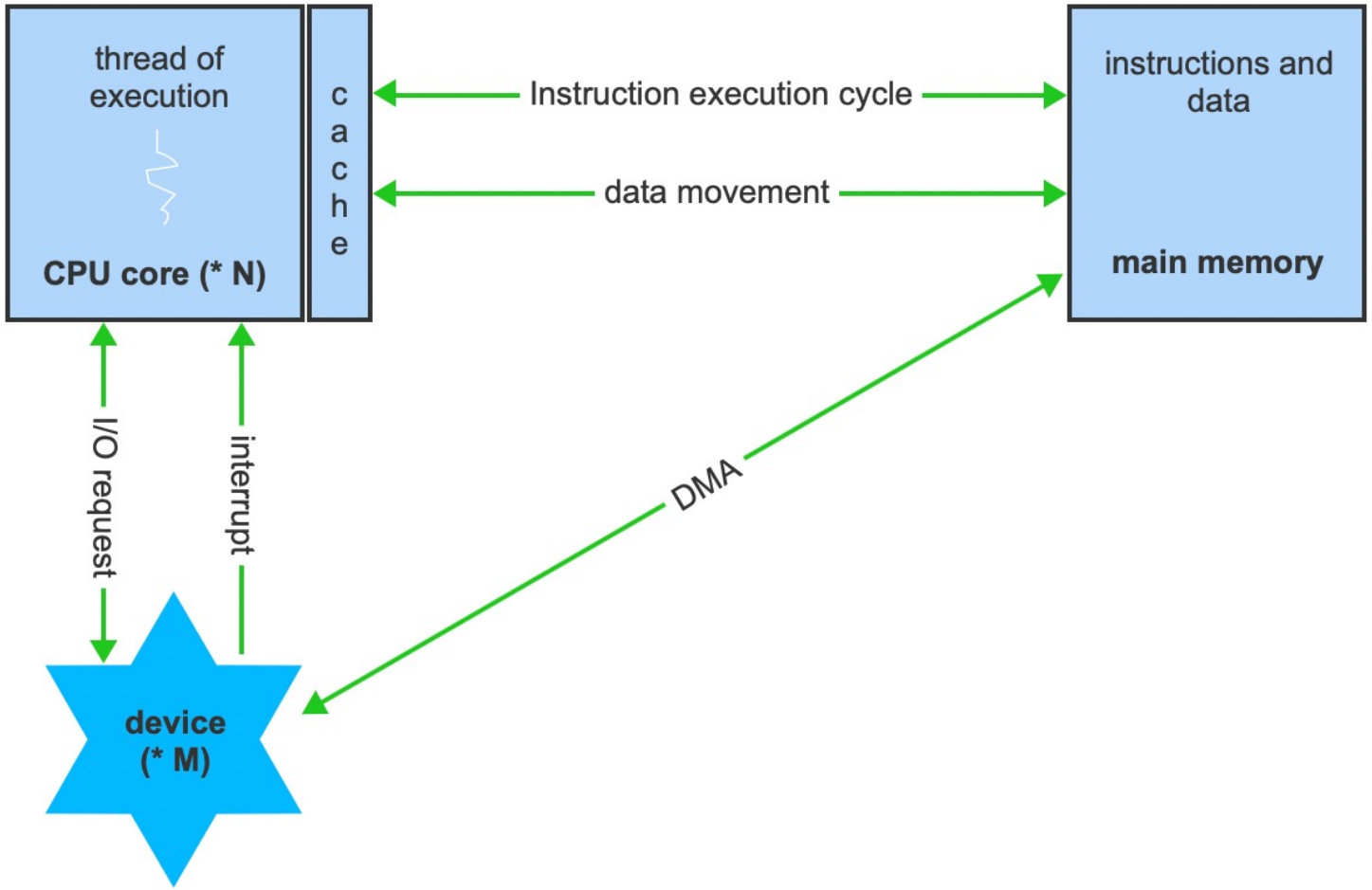
For larger mass media data movement, an instruction requests the I/O.

Managing I/O with larger data communication



The I/O request triggers a DMA transfer of data from the device to main memory.

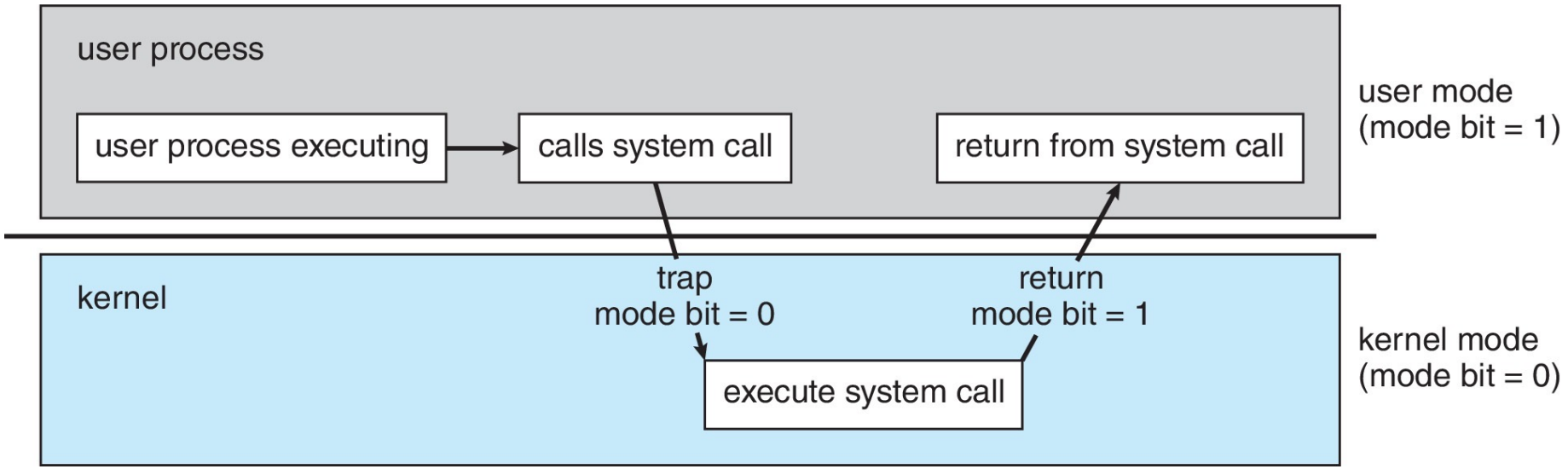
Managing I/O with larger data communication



The interrupt informs the CPU of the I/O completion.

How does the OS controls execution?

Multimode operation ensures safety

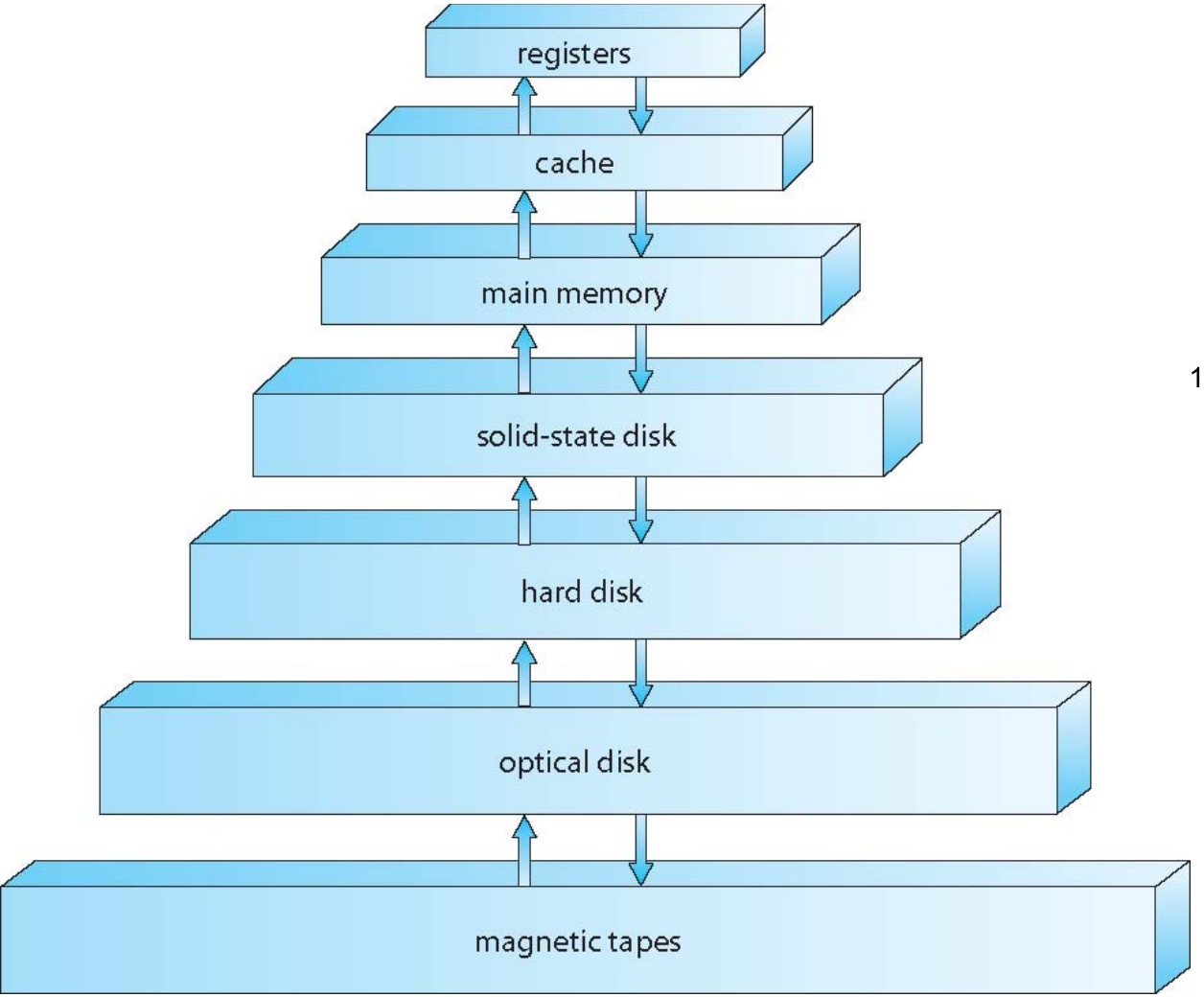


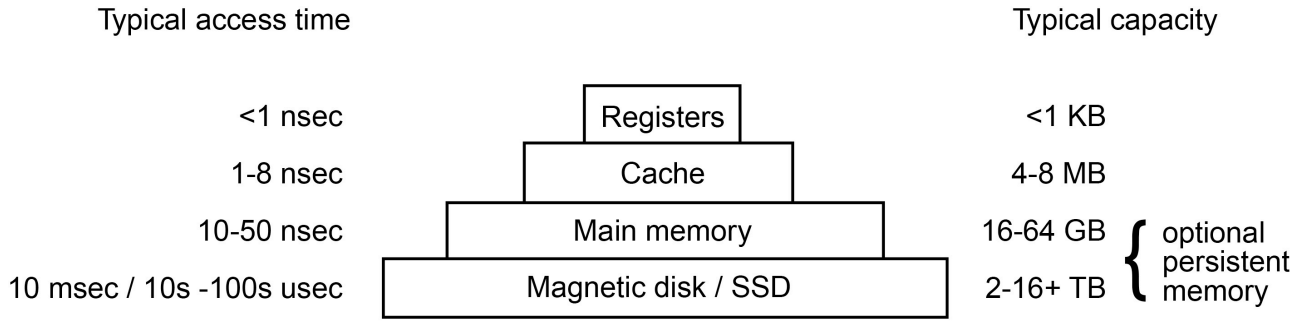
A system call occurs when the process requires a service from the operating system: file access, i/o device communication, etc

Services the OS provides

- I/O communication
- Storage management
- File management
- Process management

Storage Management





Memory management

- Keeping track of which parts of memory are currently being used and which process is using them
- Allocating and deallocating memory space as needed
- Deciding which processes (or parts of processes) and data to move into and out of memory

Mass storage management

- Mounting and unmounting
- Free-space management
- Storage allocation
- Disk scheduling
- Partitioning
- Protection

File management

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

Process management

- Creating and deleting both user and system processes
- Scheduling processes and threads on the CPUs
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication

How does the OS work?

- Bootstrap program
  - Init computer
  - Load operating system and start it
- Boot services
  - Services that are outside of kernel
- Wait for events
  - Interrupts (click, keyboard)
  - Traps
    - Errors
    - System call from user program

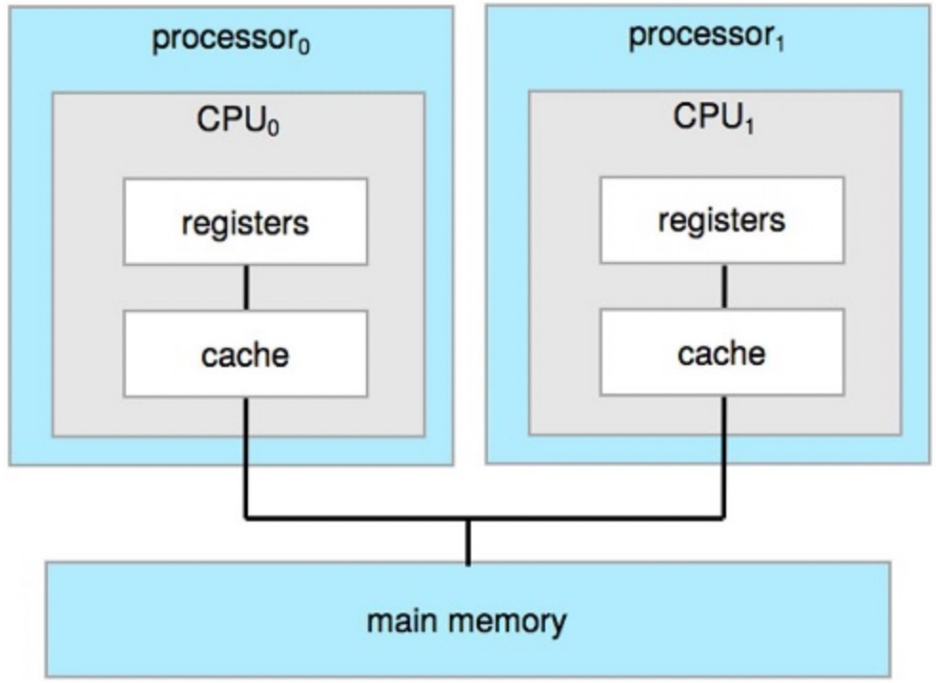
How does the computer architecture impact the OS?

Architectures

- Single processor
- Multi-processor
- Multi-core
- Clustered

Modern Computers are multiprocessor

Figure 1.3.1: Symmetric multiprocessing architecture.



The Operating System needs to ensure efficient use of all processors and CPU's

What is the difference between multiprocessor and multicore? Is it always more efficient to have more processors?

Figure 1.3.1: Symmetric multiprocessing architecture.

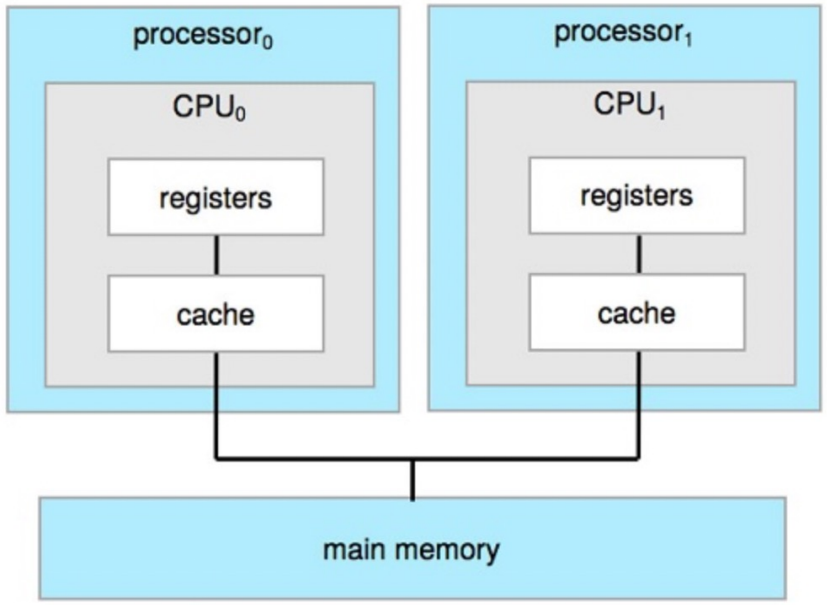
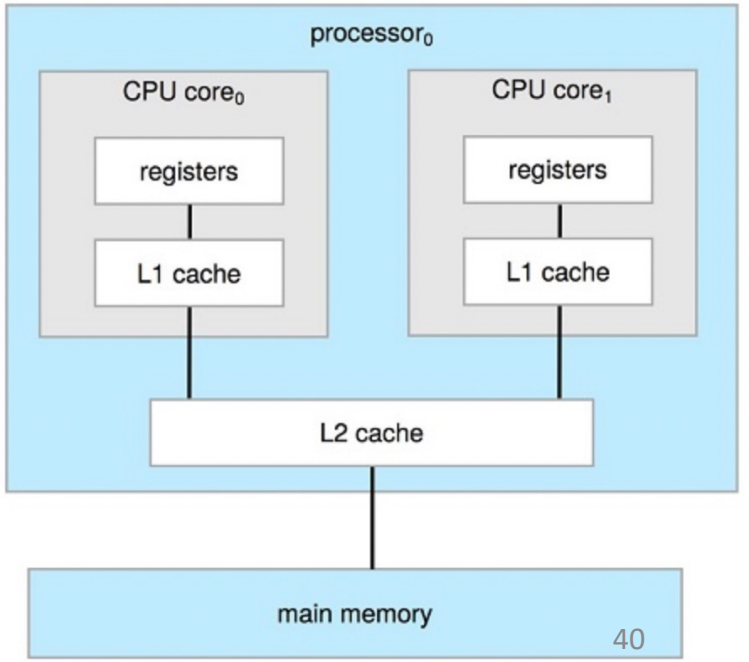
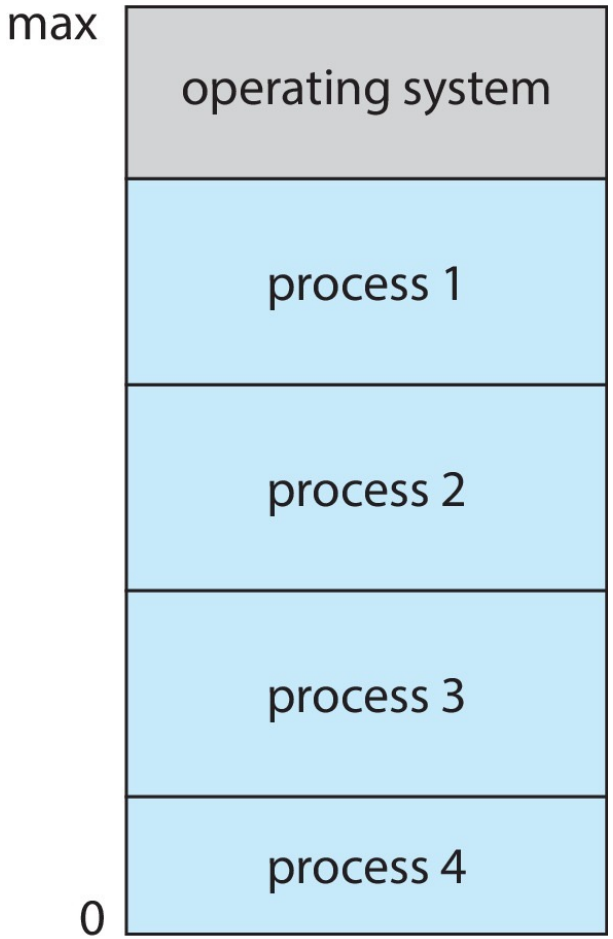


Figure 1.3.2: A dual-core design with two cores on the same chip.

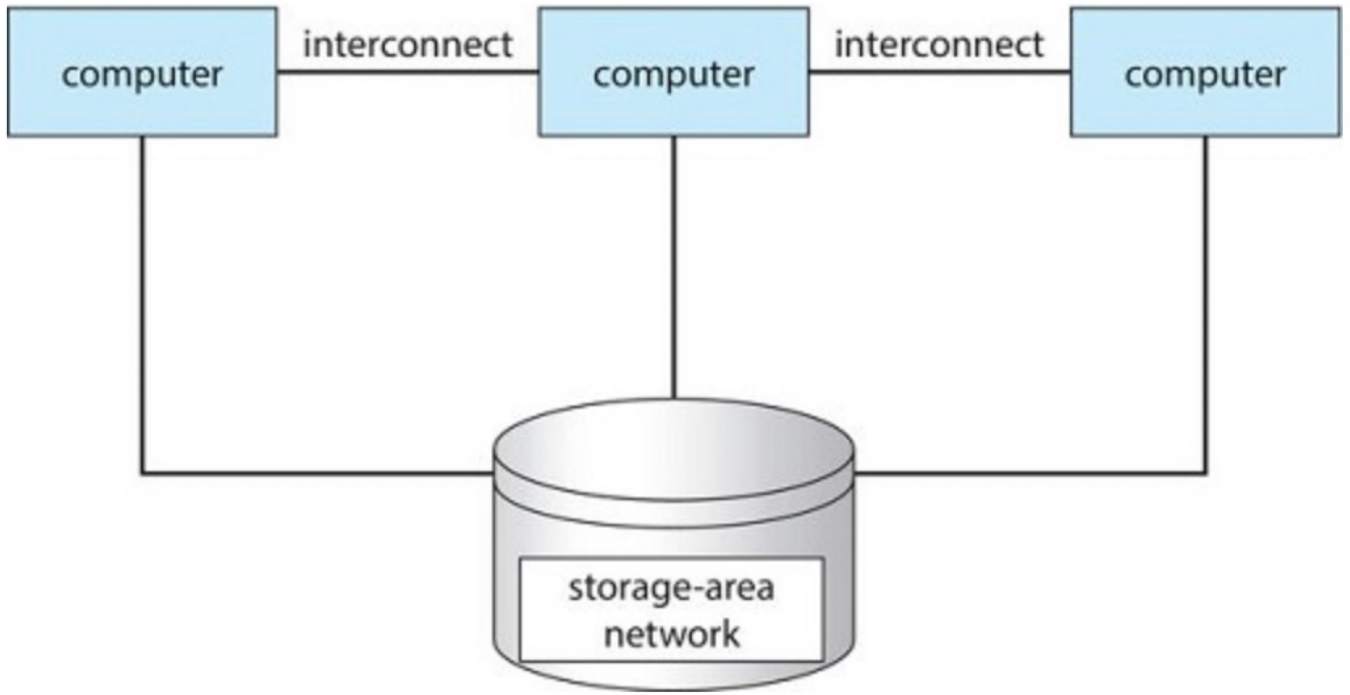


Multiprogramming and Multitasking



- Allowing multiple programs to run at the same time
- Multiprogramming: when one program is idle, the operating system chooses another to run
- Multitasking: switching between processes every X time (very short)

Figure 1.3.4: General structure of a clustered system.



Requires parallelization of operations

What questions do you have?

Homework

- Read Chapter 1.9 (not covered today)
- Reading Quiz on Moodle
- Identify boot services that run in your computer and show them in Discord (general channel)

Lecture 3: Operating System Structures

Paula Lago

Paula.lago@concordia.ca

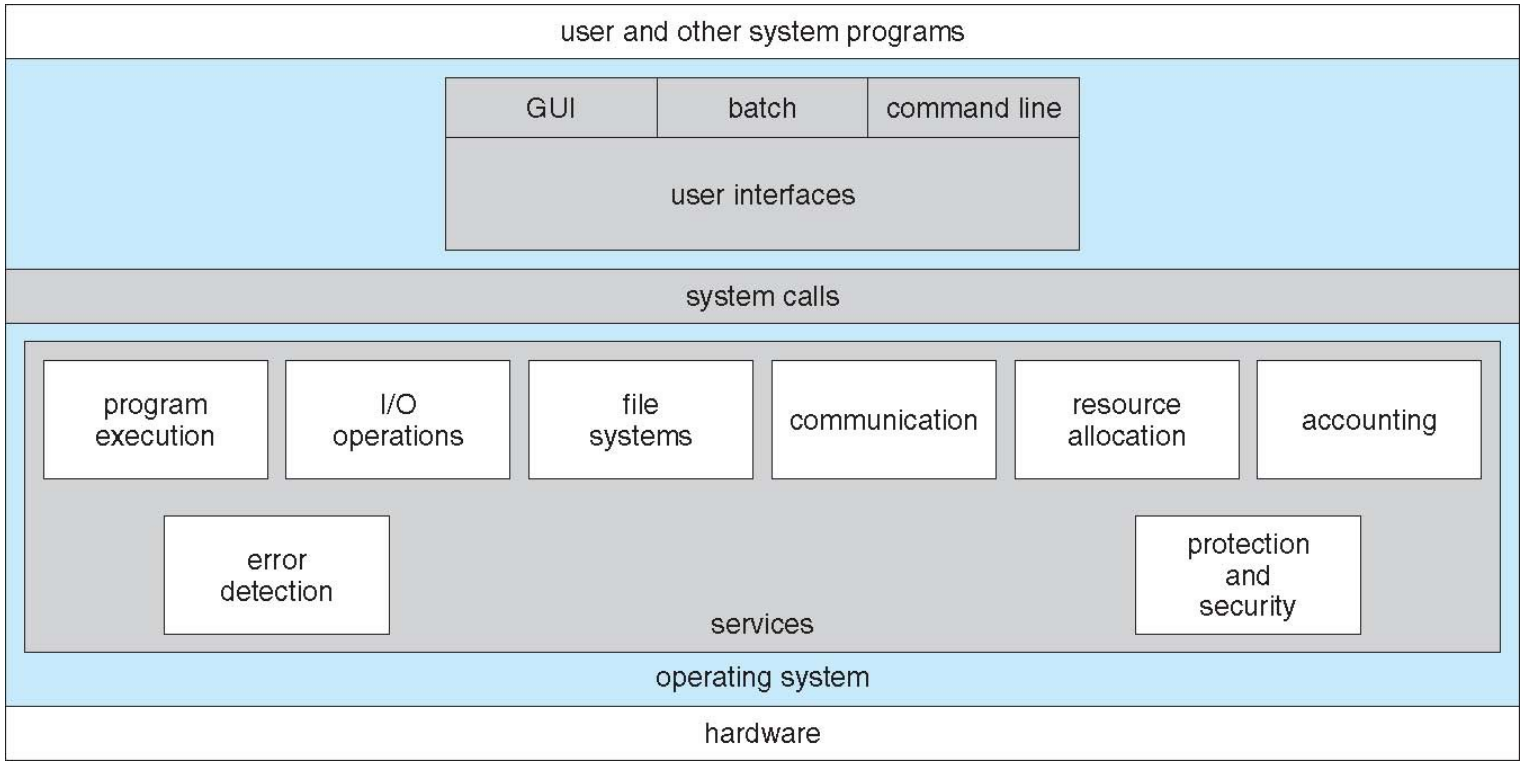
Pre-course survey

- Final exam rule is stressing
  - I agree!
  - Check individual understanding
  - Exam won't be a surprise, come to class
- "Take care of less intelligent students"
  - No one is less intelligent than others!

Today's activities

- Use a piece of paper with your name to answer
  - Write a pronunciation key to your name
  - Eg: Paula is pronounced as Paw-la (a as in father)
- Hand out at the end for participation points

Operating system components

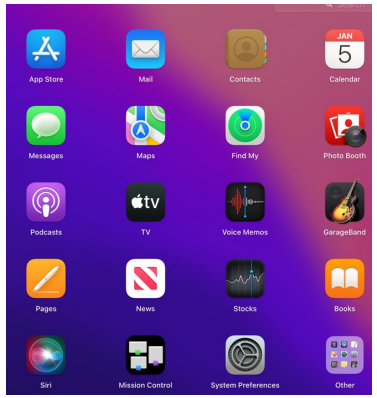


- Applications can't access operating system state or code directly
- OS code and state are stored in kernel space
  - Must be in kernel mode to access this data
- Application code and state is in user space

Interacting with the OS

- UI (user)
- System calls (software)
  - Traps
- Errors (exceptions)
  - Not intentional
- Interrupts (hardware)
  - Events
  - Maskable or non-maskable

Command interpreter



Graphical UI (GUI)

Touch based

Agenda

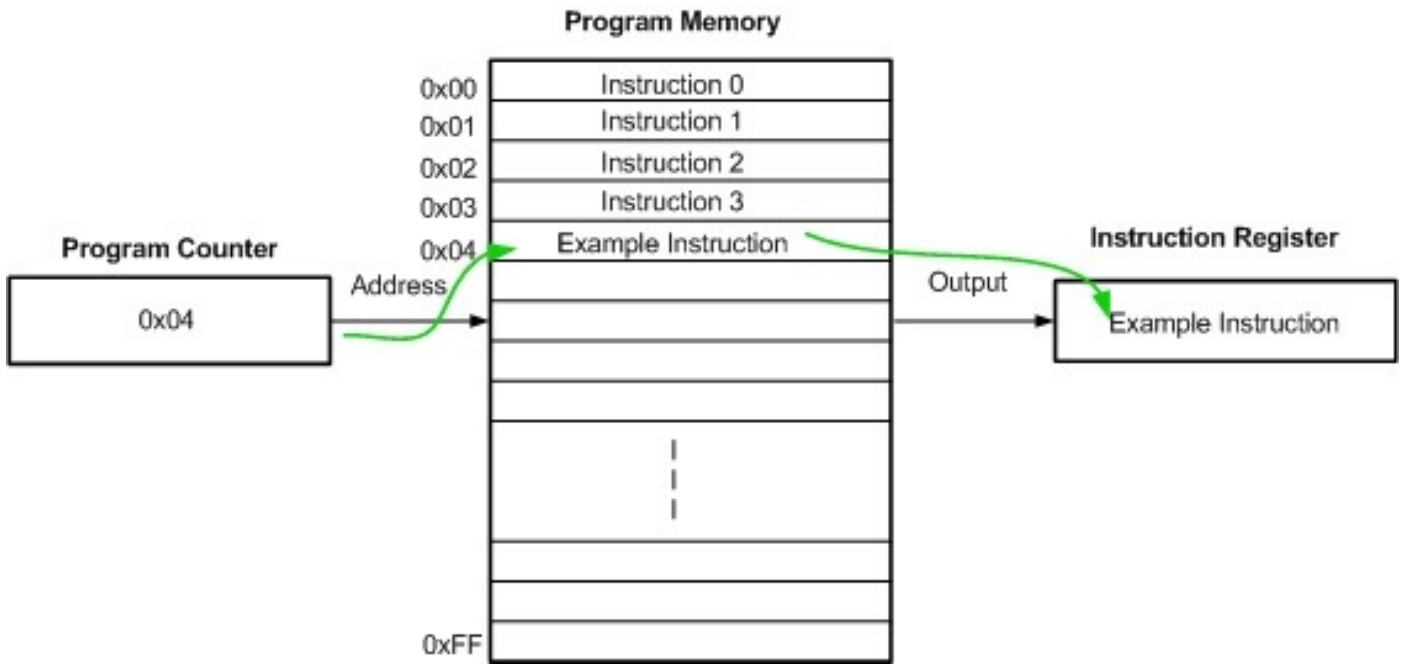
1. How are interrupts handled?
2. How are system calls handled?
3. How are O.S. organized?

How are interrupts handled?

Signaling an interrupt

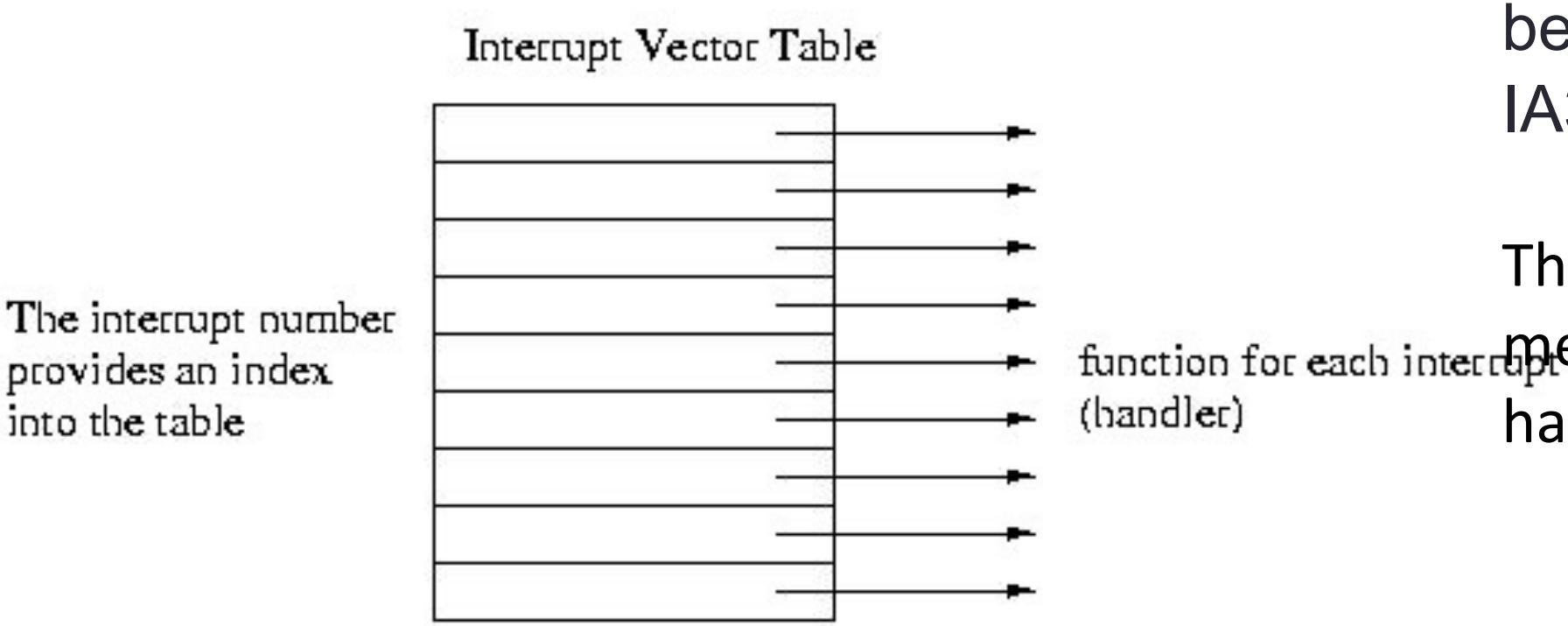
1. A PIN in the Processor is set
2. The Processor acknowledges the interrupt to the device
3. The device sends its interrupt number over the bus

The CPU is running a process, when the hardware device signals an interrupt



When the signal is received, the correct routine should be called

The interrupt has a number used by the processor to go to the handler



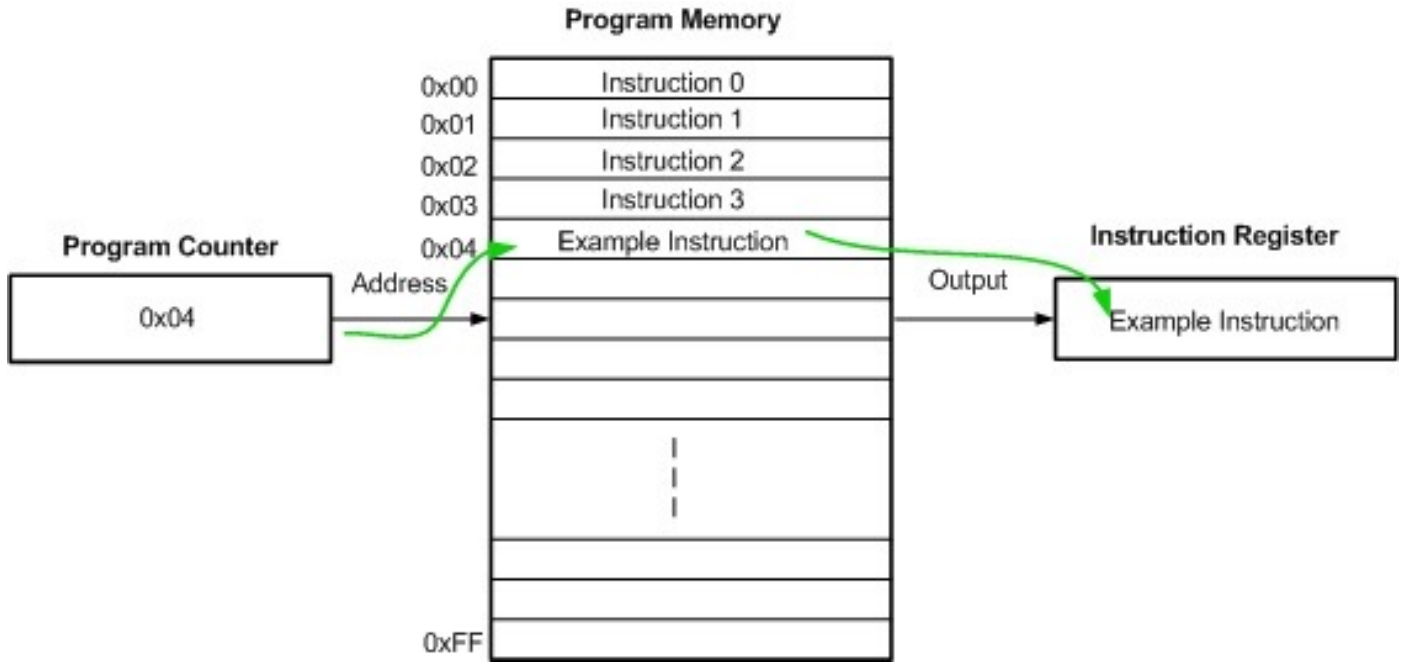
Location of Interrupt Descriptor Table can be set with the `lidt` IA32 instruction

The vector contains a memory address for the handler routine

What data structure would you use for the interrupt vector?

Before going to the routine, it needs to save what it was doing

- Save state of Program counter and Registers
- New Program Counter will be the handler address obtained from the Interrupt Table
- The mode is changed to the level required by the handler (i.e. kernel mode)



After the interrupt

- Restore the process
  - Program counter
  - Registers
- Reset mode
- Continue its execution

Seeing registered interrupts in linux (cat /proc/interrupts)

E.2.10. /proc/interrupts

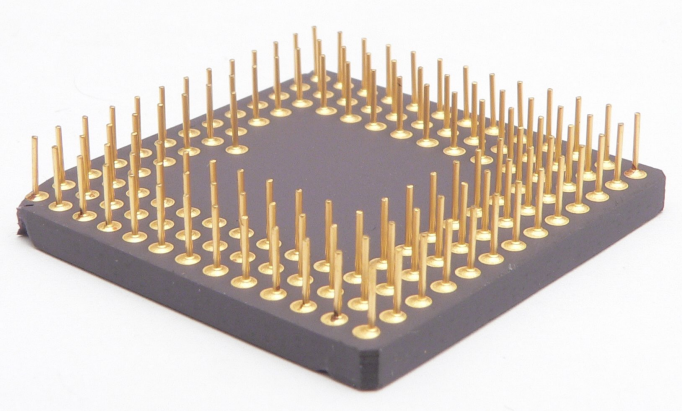
This file records the number of interrupts per IRQ on the x86 architecture. A standard `/proc/interrupts` looks similar to the following:

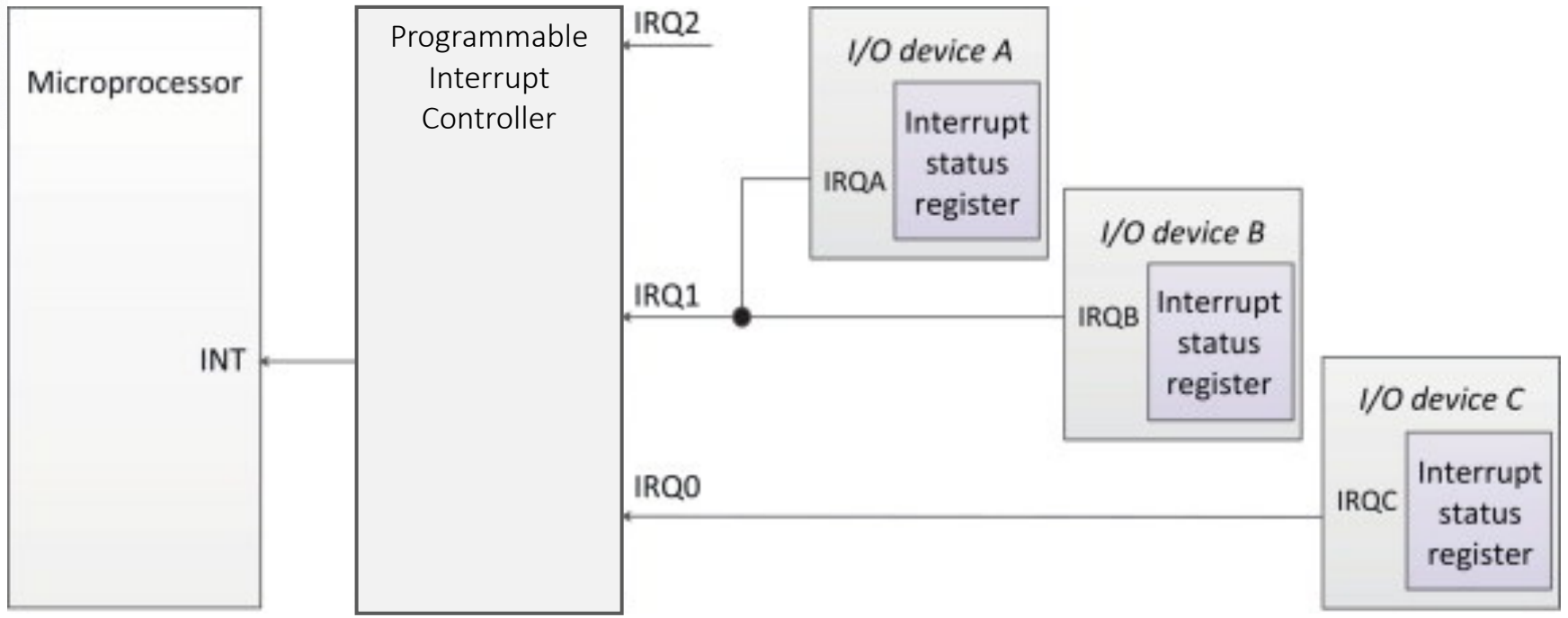
```
  CPU0
  0:   80448940          XT-PIC  timer
  1:     174412          XT-PIC  keyboard
  2:          0          XT-PIC  cascade
  8:          1          XT-PIC  rtc
 10:     410964          XT-PIC  eth0
 12:      60330          XT-PIC  PS/2 Mouse
 14:    1314121          XT-PIC  ide0
 15:    5195422          XT-PIC  ide1
NMI:          0
ERR:          0
```

How to manage a large number of interrupts?

- Chaining
  - The handler address contains multiple routines

Programmable Interrupt Controller

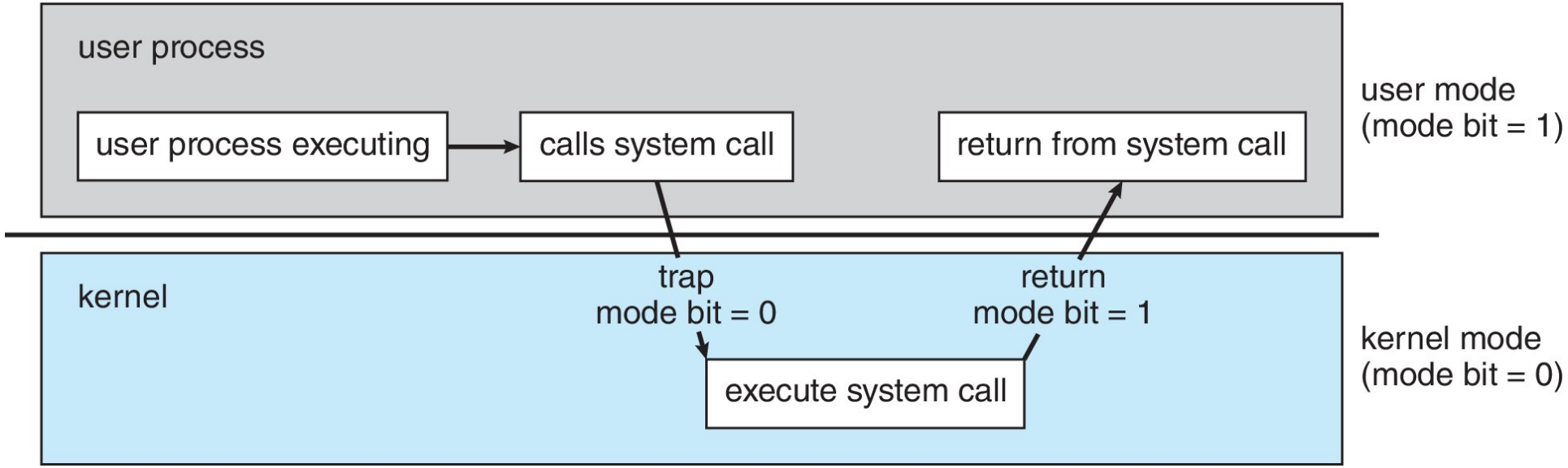




vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault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

How are system calls handled?

User programs interact with the OS via system calls



OS Services:

- Process management
- Memory management
- File system management
- I/O or mass storage management

System calls are traps (i.e. interrupts)

During OS initialization, the kernel provides a handler to be invoked by the processor when the trap occurs

- When the trap occurs, the processor can change the current protection level (e.g. switch from user mode to kernel mode)

Example:

```
movl $20, %eax        # Get PID of current process
int $0x80             # Invoke system call!
# Now %eax holds the PID of the current process
```

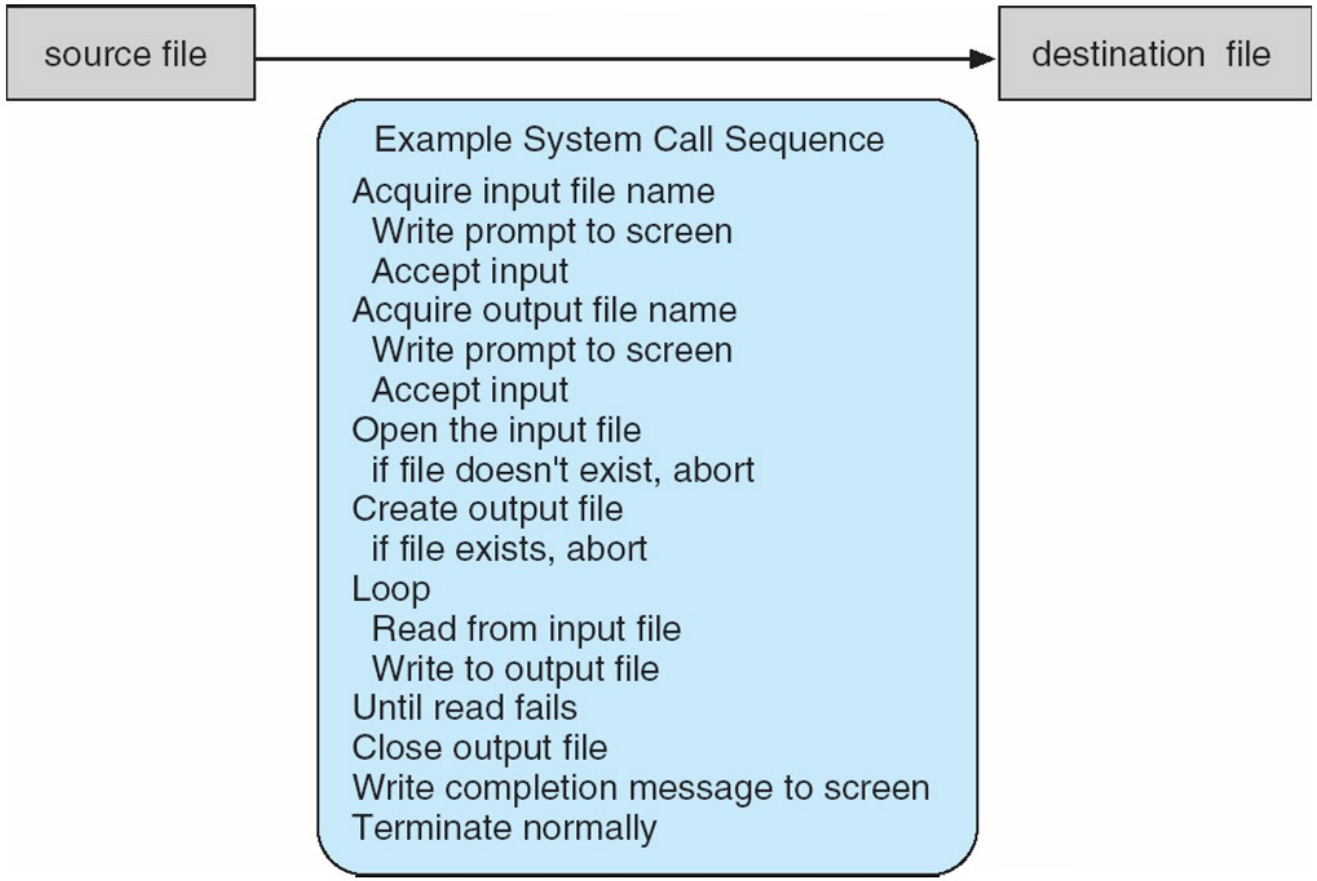
Of course, operating systems provide wrappers for this

```
int pid = getpid();  /* Does the syscall for us */
```

Applications can invoke operating system subroutines without having to know what address they live at

- OS can verify applications' requests and prevent misbehavior
  - e.g. check arguments to system calls, verify process permissions, etc.

Example system calls for copying a file

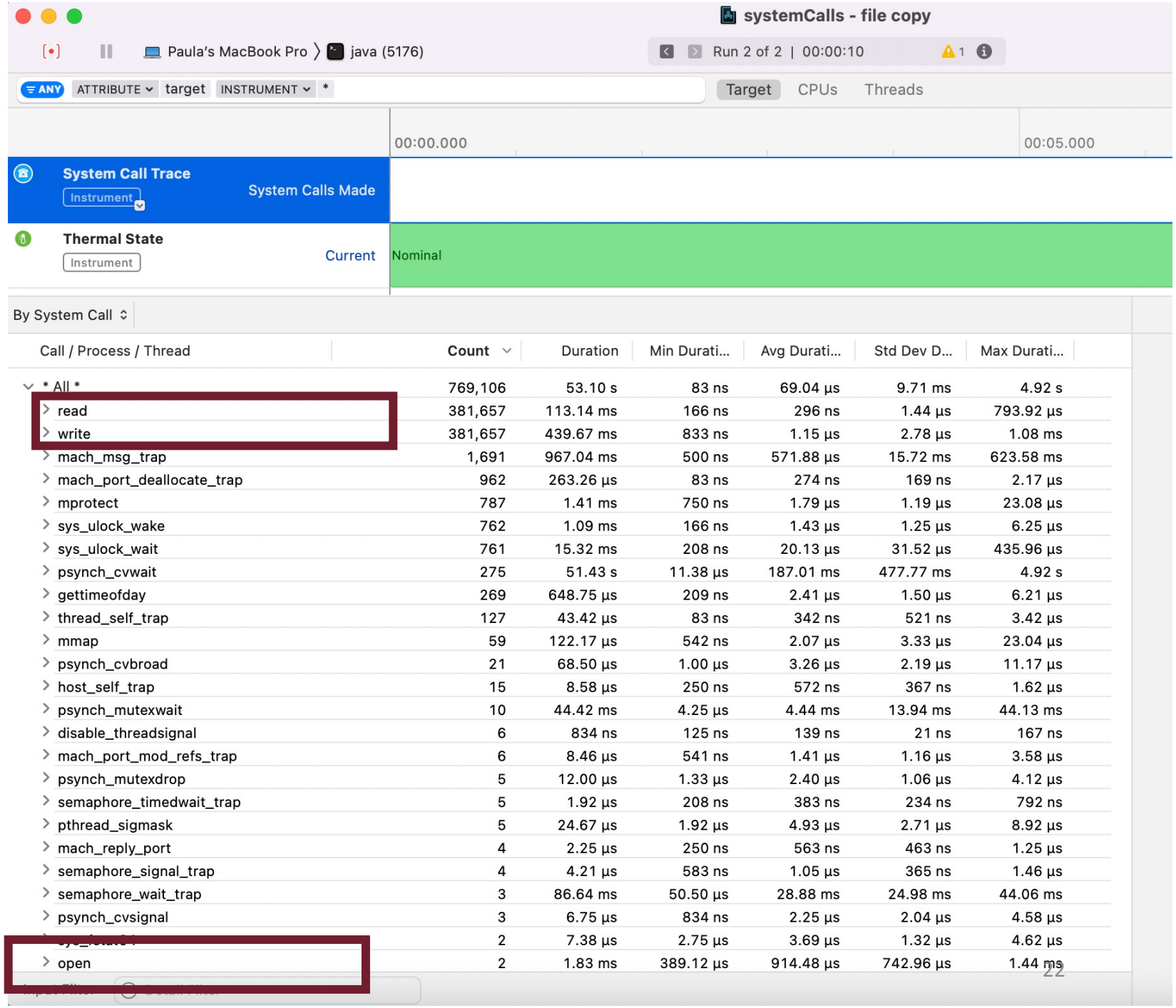


Example system calls for copying a file

Write a java program for copying a file to another
Use

- Instruments (mac)
- Strace (linux)

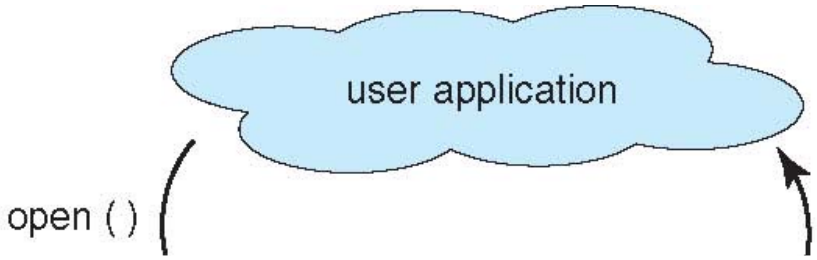
Execute the program



Call / Process / Thread	Count	Duration	Min Durati...	Avg Durati...	Std Dev D...	Max Durati...
* All *	769,106	53.10 s	83 ns	69.04 µs	9.71 ms	4.92 s
read	381,657	113.14 ms	166 ns	296 ns	1.44 µs	793.92 µs
write	381,657	439.67 ms	833 ns	1.15 µs	2.78 µs	1.08 ms
mach_msg_trap	1,691	967.04 ms	500 ns	571.88 µs	15.72 ms	623.58 ms
mach_port_deallocate_trap	962	263.26 µs	83 ns	274 ns	169 ns	2.17 µs
mprotect	787	1.41 ms	750 ns	1.79 µs	1.19 µs	23.08 µs
sys_ulock_wake	762	1.09 ms	166 ns	1.43 µs	1.25 µs	6.25 µs
sys_ulock_wait	761	15.32 ms	208 ns	20.13 µs	31.52 µs	435.96 µs
psynch_cvwait	275	51.43 s	11.38 µs	187.01 ms	477.77 ms	4.92 s
gettimeofday	269	648.75 µs	209 ns	2.41 µs	1.50 µs	6.21 µs
thread_self_trap	127	43.42 µs	83 ns	342 ns	521 ns	3.42 µs
mmap	59	122.17 µs	542 ns	2.07 µs	3.33 µs	23.04 µs
psynch_cvbroad	21	68.50 µs	1.00 µs	3.26 µs	2.19 µs	11.17 µs
host_self_trap	15	8.58 µs	250 ns	572 ns	367 ns	1.62 µs
psynch_mutexwait	10	44.42 ms	4.25 µs	4.44 ms	13.94 ms	44.13 ms
disable_threadsignal	6	834 ns	125 ns	139 ns	21 ns	167 ns
mach_port_mod_refs_trap	6	8.46 µs	541 ns	1.41 µs	1.16 µs	3.58 µs
psynch_mutexdrop	5	12.00 µs	1.33 µs	2.40 µs	1.06 µs	4.12 µs
semaphore_timedwait_trap	5	1.92 µs	208 ns	383 ns	234 ns	792 ns
pthread_sigmask	5	24.67 µs	1.92 µs	4.93 µs	2.71 µs	8.92 µs
mach_reply_port	4	2.25 µs	250 ns	563 ns	463 ns	1.25 µs
semaphore_signal_trap	4	4.21 µs	583 ns	1.05 µs	365 ns	1.46 µs
semaphore_wait_trap	3	86.64 ms	50.50 µs	28.88 ms	24.98 ms	44.06 ms
psynch_cvsignal	3	6.75 µs	834 ns	2.25 µs	2.04 µs	4.58 µs
[illegible]	2	7.38 µs	2.75 µs	3.69 µs	1.32 µs	4.62 µs
open	2	1.83 ms	389.12 µs	914.48 µs	742.96 µs	1.44 ms

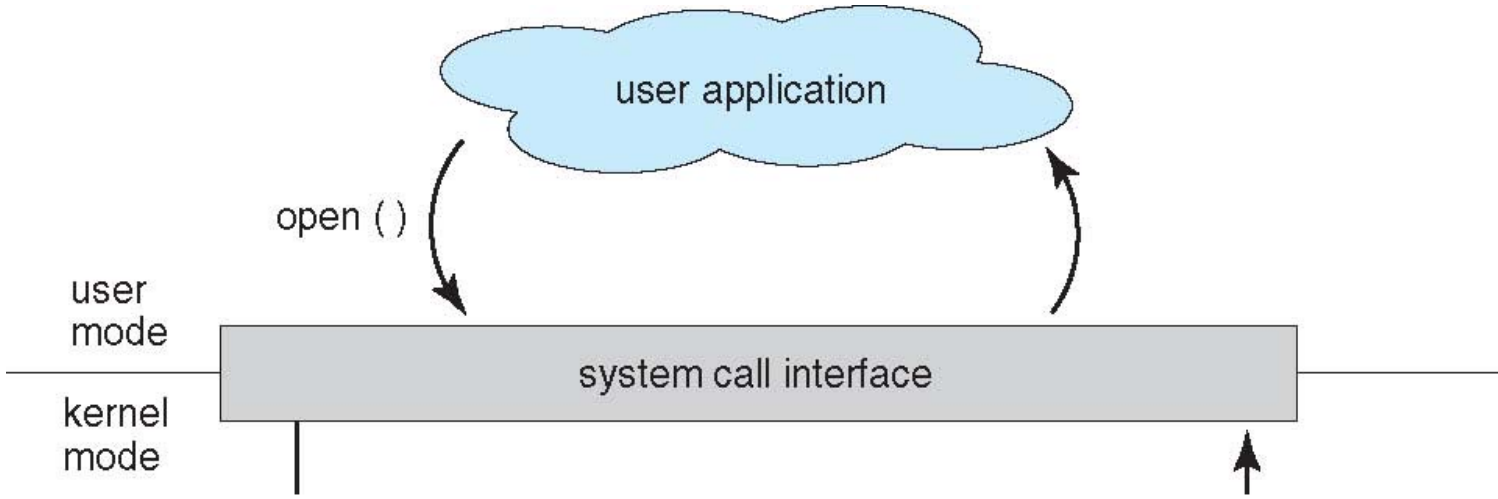
Implementation of system calls

- Application invokes the system call.



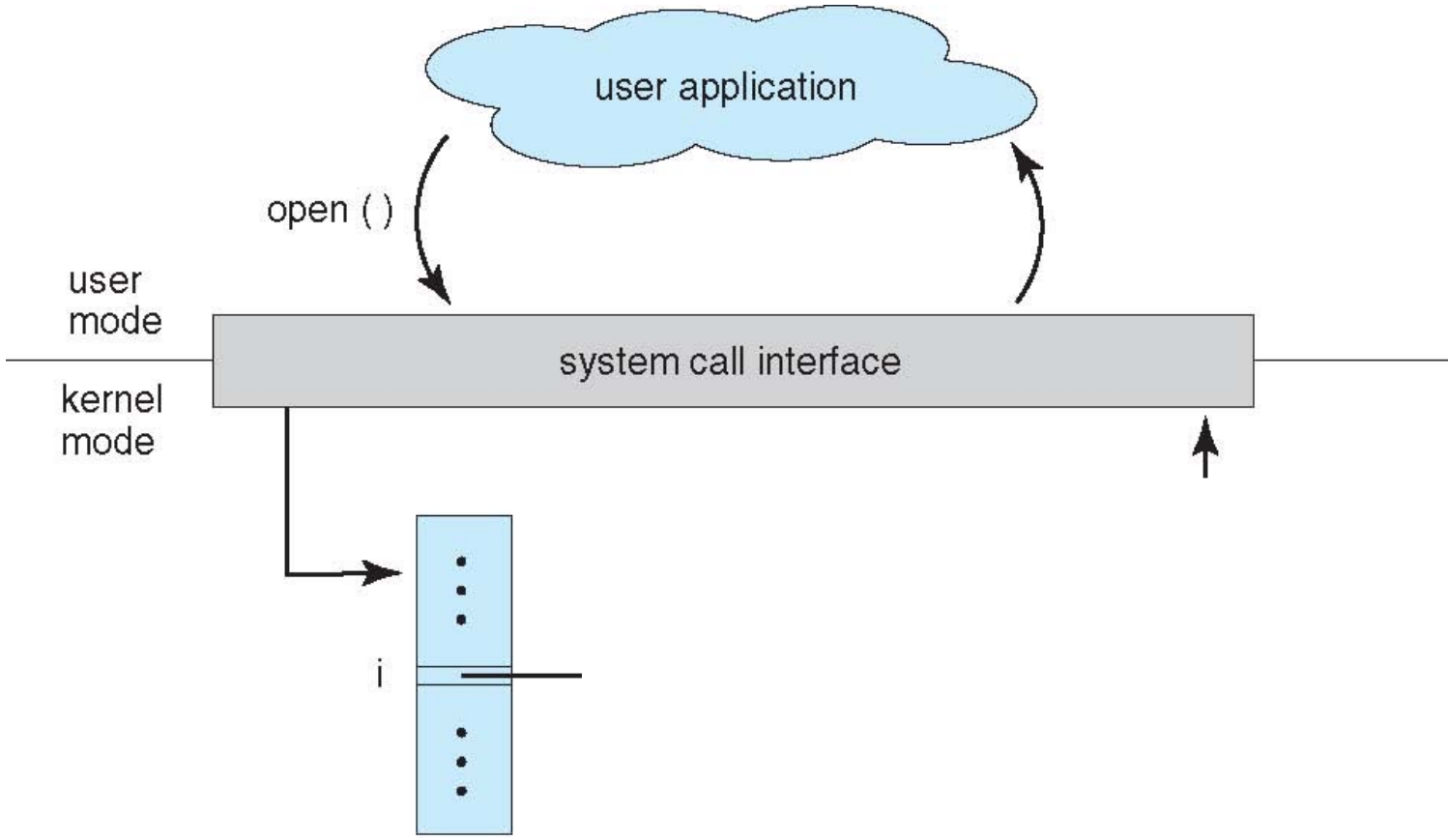
Implementation of system calls

- Application invokes the system call.
- The system uses a CPU-specific method to transition from user to kernel mode and pass the system-call information to the kernel.



Implementation of system calls

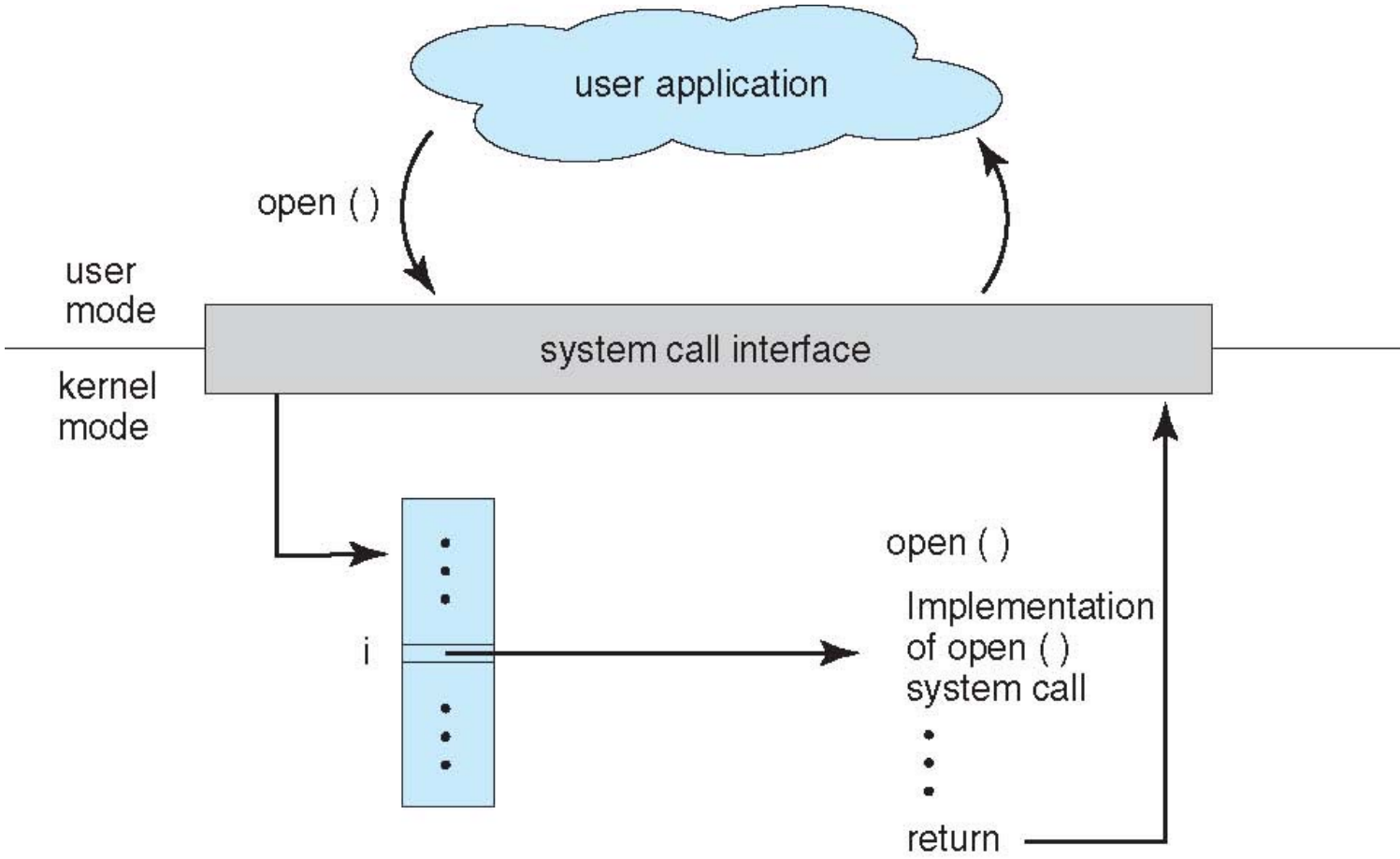
Each system call has a number, and that number is used as an index in the system call table, to invoke the appropriate routine.



Implementation of system calls

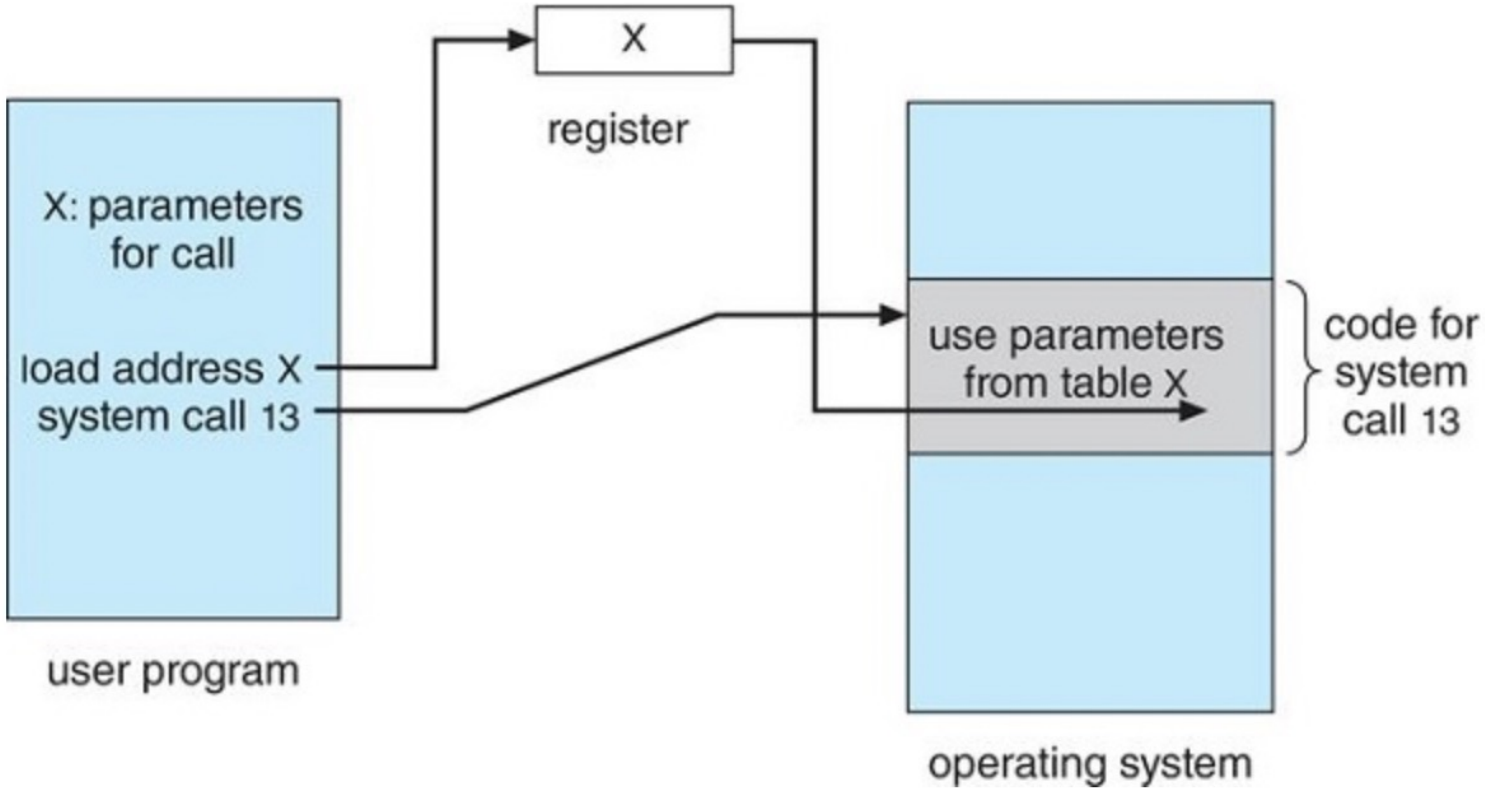
The routine executes and returns when complete or indicating an error.

The return transitions the thread from kernel to user mode.



System calls require parameters sometimes

Figure 2.3.2: Passing of parameters as a table.



Examples of Windows and Unix System Calls

	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() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

What types of system calls would be called by this program?

```
public class PortReader {
    public static void main(String[] args) {
        // Set the port number to listen on
        int port = 1234;

        try (ServerSocket serverSocket = new ServerSocket(port);
             Socket socket = serverSocket.accept();
             InputStream inputStream = socket.getInputStream();
             BufferedReader reader = new BufferedReader(new
InputStreamReader(inputStream));
             FileWriter fileWriter = new FileWriter("data.txt")) {

            // Create a thread to read data from the port for 5 minutes
            Thread readThread = new Thread(new Runnable() {
                @Override
                public void run() {
                    try {
                        // Set a 5 minute timeout
                        socket.setSoTimeout(300000);

                        String line;
                        while ((line = reader.readLine()) != null) {
                            fileWriter.write(line +
System.lineSeparator());
                        }
                    } catch (SocketTimeoutException e) {
                        System.out.println("Timed out after 5 minutes");
```

Types of System Calls

- Process control
  - create process, terminate process
  - end, abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - Dump memory if error
  - **Debugger** for determining **bugs, single step** execution
  - **Locks** for managing access to shared data between processes
- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices

Types of System Calls (Cont.)

- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages if **message passing model** to **host name** or **process name**
    - From **client** to **server**
  - **Shared-memory model** create and gain access to memory regions
  - transfer status information
  - attach and detach remote devices
- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

Hardware Requirements

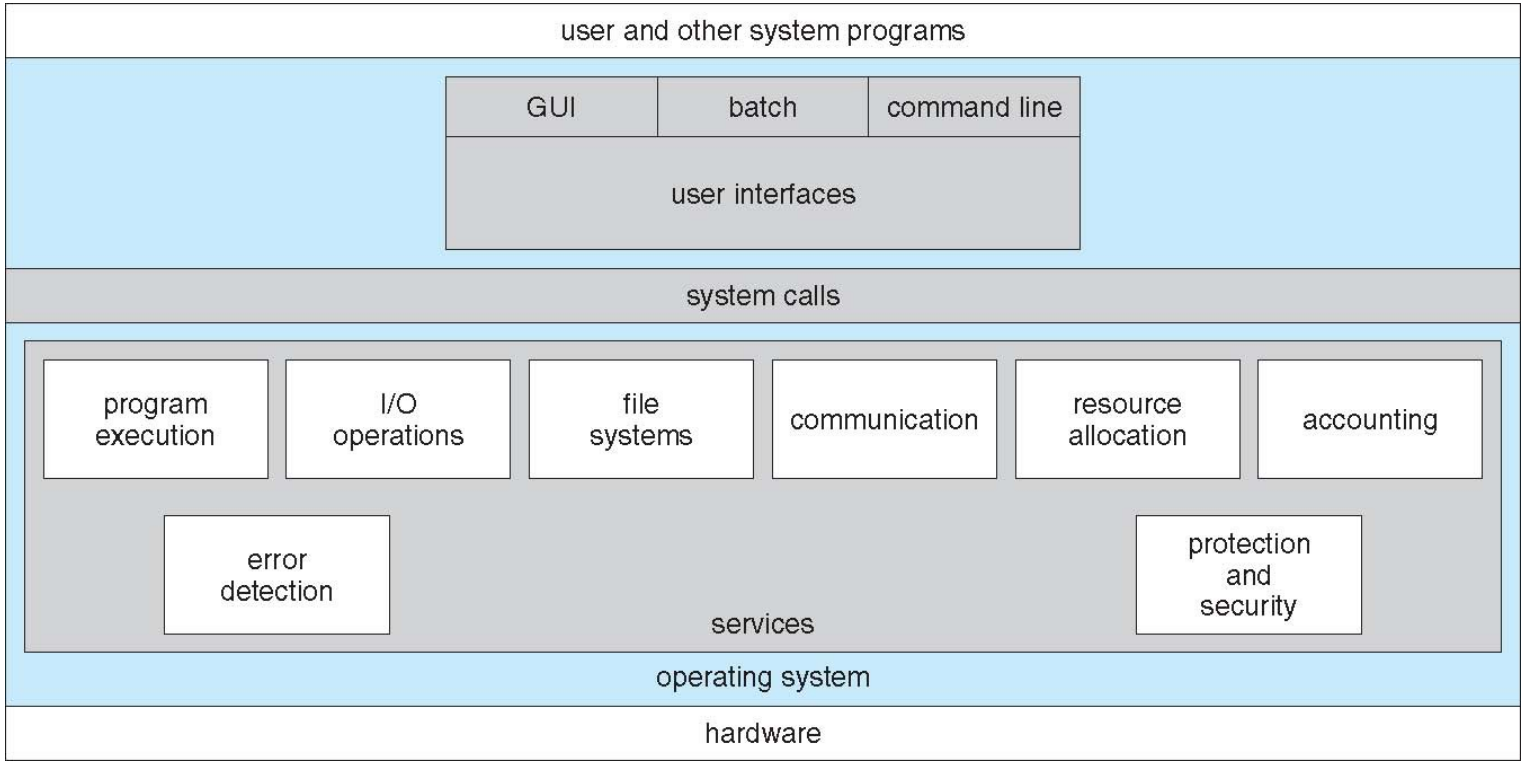
- At least dual-mode execution: kernel mode and user mode
- Virtual memory management, including memory protection to
- enforce barrier between kernel space and user space
- Ability to trap (or otherwise transfer control) into kernel-mode code
- Support for both software interrupts (e.g. traps, faults) and hardware interrupts (e.g. I/O devices)
- A hardware timer facility to allow OS to regain control of the CPU

Operating system structures

Design Patterns

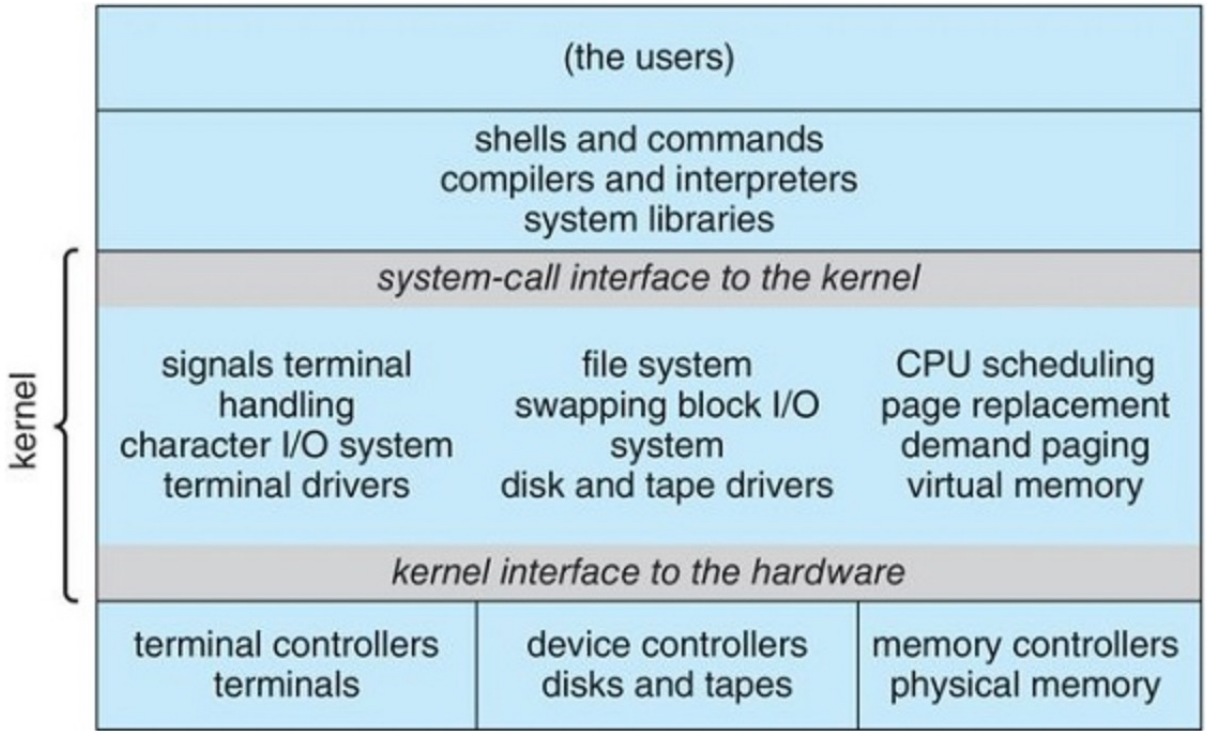
- A guiding principle in operating system design: **separation of policy and mechanism**
- **Policy** specifies what needs to be done
  - e.g. Which virtual memory page should the OS evict? • e.g. What process should the OS run next?
- **Mechanism** specifies how to do it
  - e.g. how to save dirty pages, how to update the CPU's page table to
  - reflect the page-out/page-in operations, other bookkeeping
  - e.g. how to save the current process' context, how to restore the new process' context
- Mechanisms are unlikely to change substantially over time • (In the context of a given OS and set of hardware)
- Policies are very likely to change
  - May even be part of an operating system's configuration options!

The operating system components are interconnected into a kernel. Different structures can be used to interconnect them



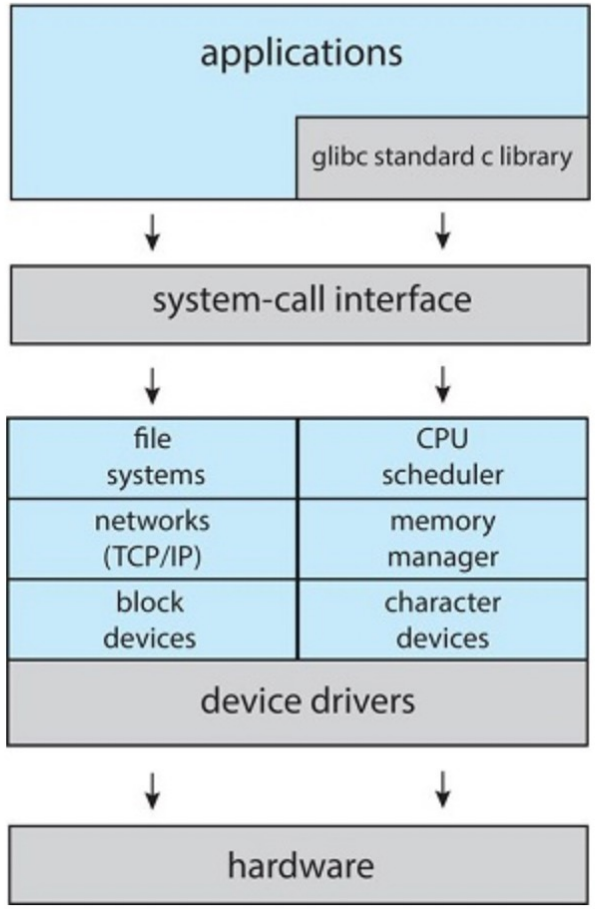
Monolithic structure (single binary file and single address space)

Figure 2.8.1: Traditional UNIX system structure.



Linux structure

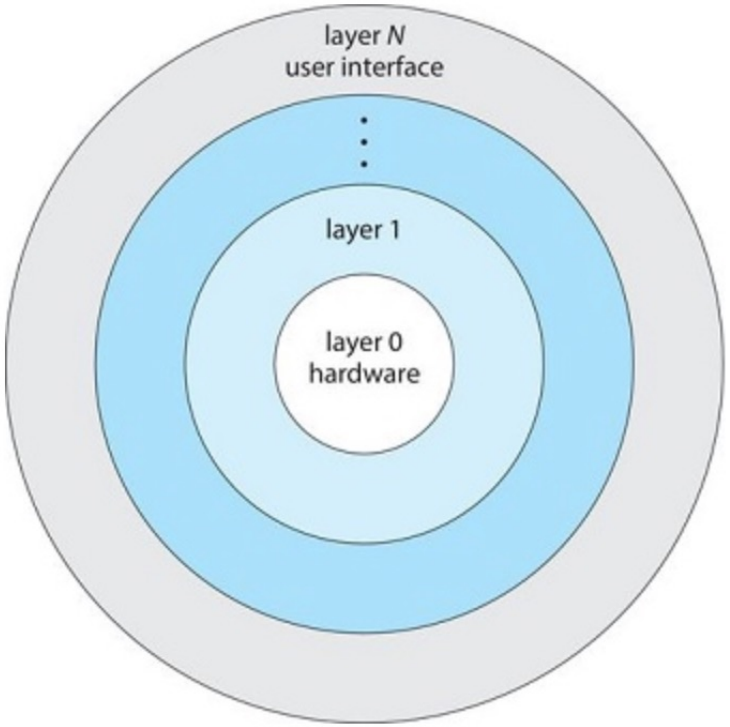
Figure 2.8.2: Linux system structure.



Layered Structure

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

Figure 2.8.3: A layered operating system.



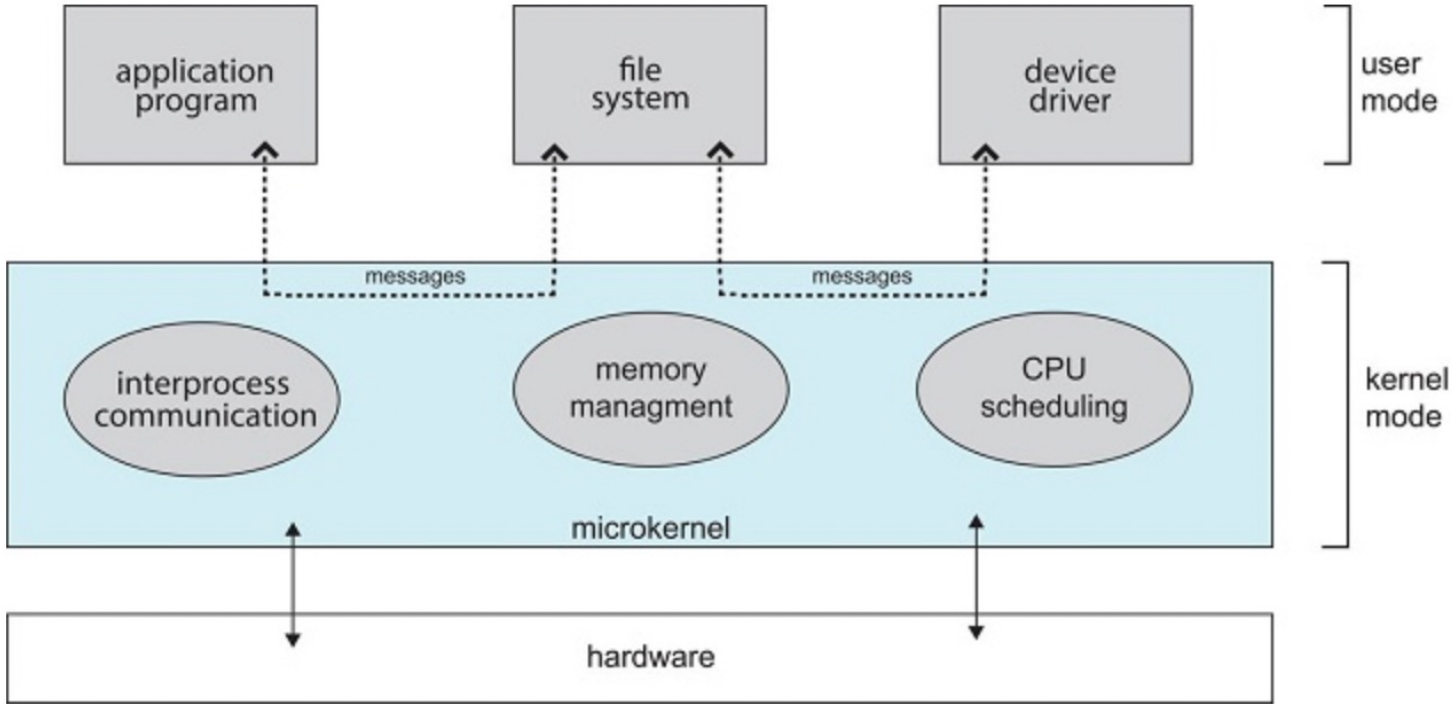
Easy to debug ☑

More latency for system calls ✗

1. Tradeoff: make fewer layers with greater functionality
   - Still realizes many of the benefits of layering

Microkernels: the kernel keeps only essential functions (process and memory mgt, comm)

Message passing involves opening a connection between two processes



Easy to extend

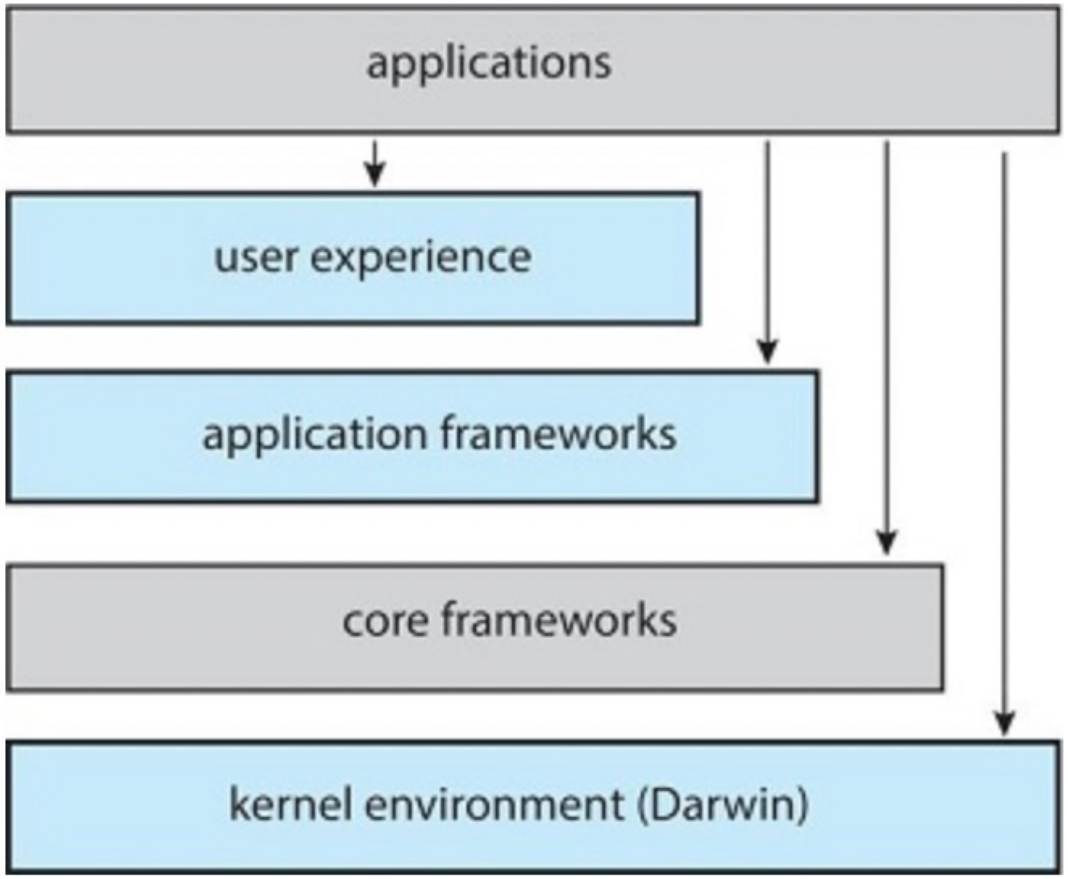
Communication overhead

Modules

- Modern monolithic kernels use loadable kernel modules to provide extensibility during normal operation
  - Such kernels are called modular kernels
- Kernel defines interfaces for parts of kernel that require extensibility
  - e.g. what set of operations must all filesystems support?

- Kernel includes a module loader that is able to load and statically link a kernel module directly into the kernel
  - At load time, module-references to kernel symbols are updated with actual addresses of kernel variables and functions
  - (Modules can also expose symbols for other modules to access)
  - Module code is loaded into kernel space; code runs in kernel mode
  - Modules can be unloaded, if no other module is using its symbols

Hybrid systems



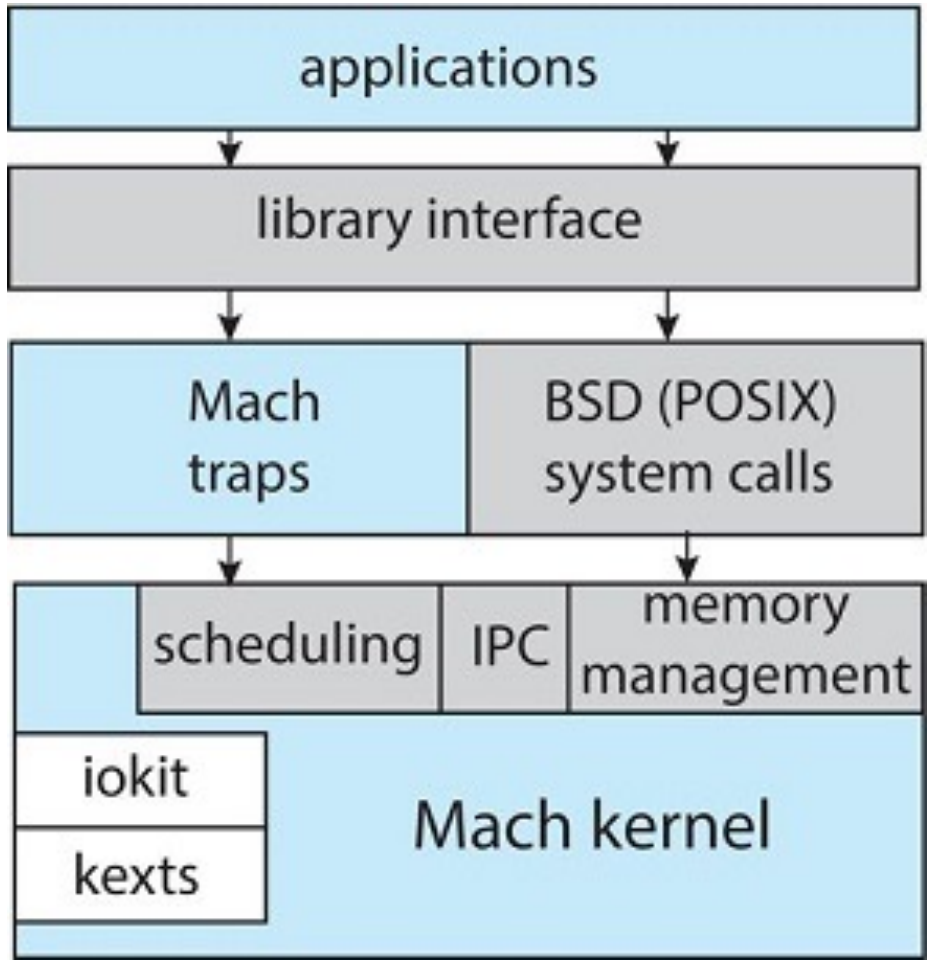
Aqua: keyboard and mouse
Springboard: touch based

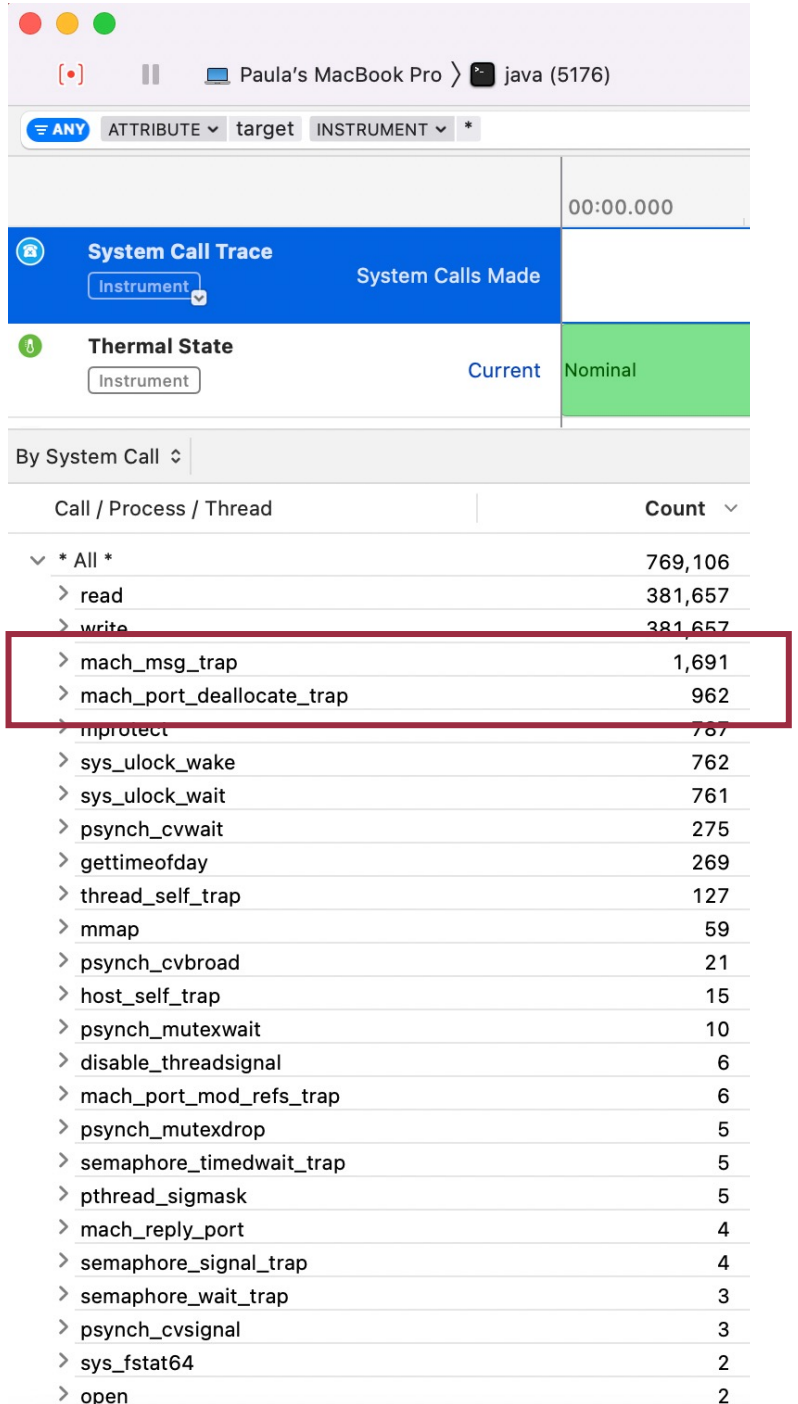
Cocoa: interface for Objective-C and Swift

Support graphics and media: Quicktime and Open GL

Two kernels: Mach and BSD UNIX

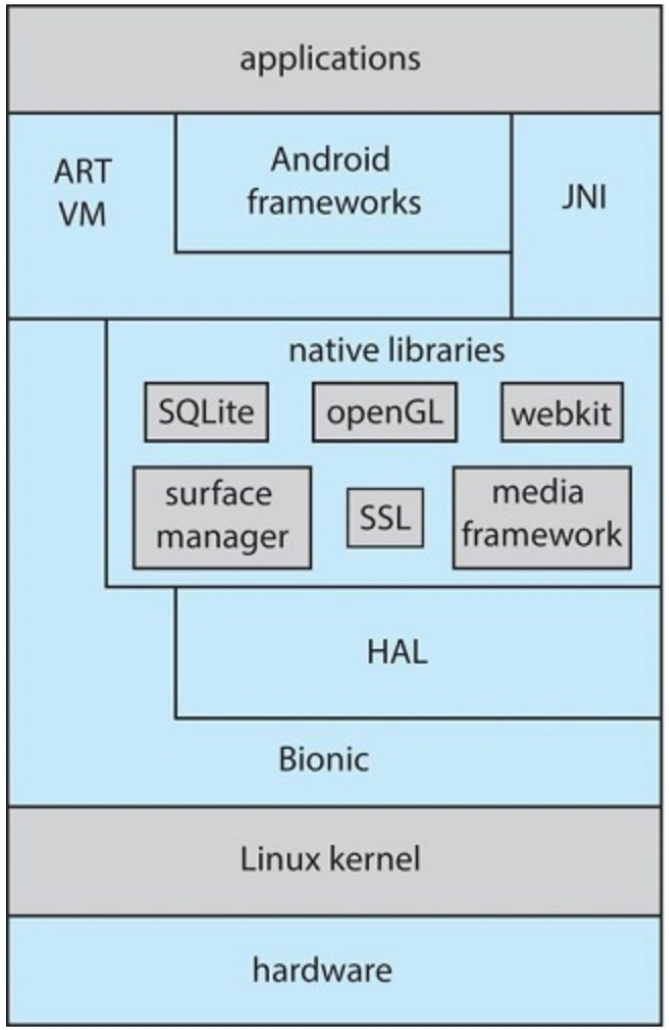
Darwin Kernel





Android

Figure 2.8.7: Architecture of Google's Android.



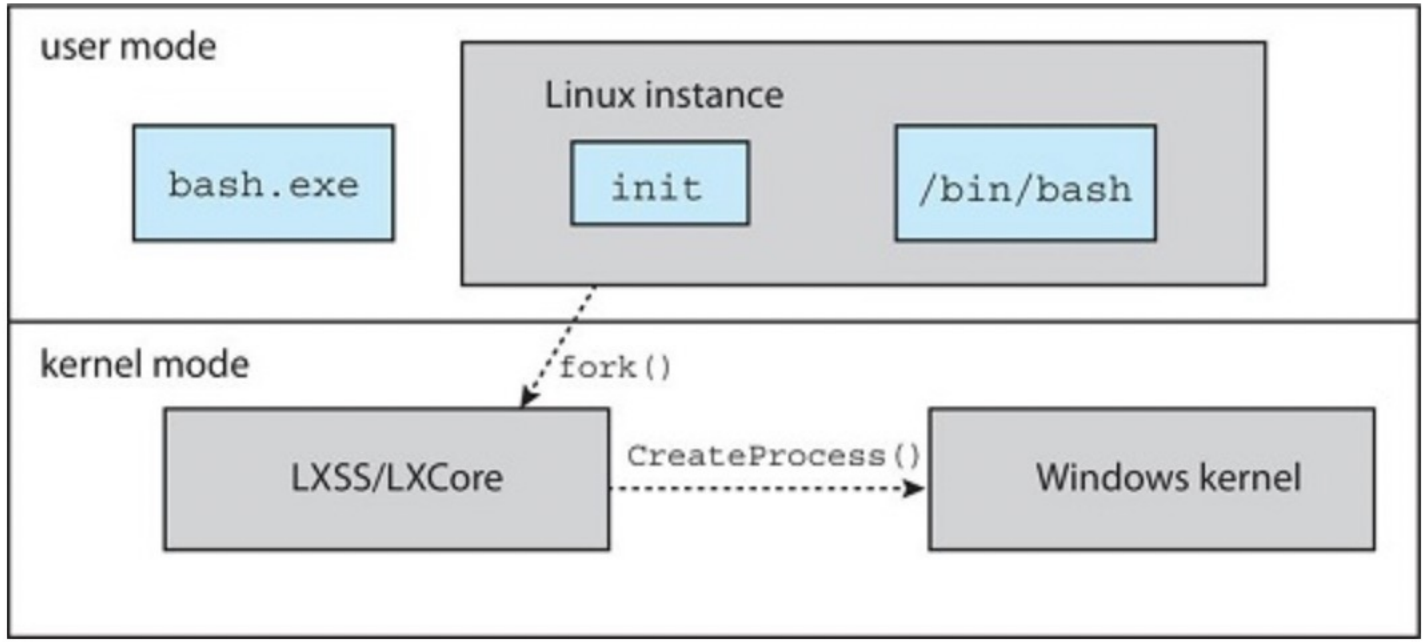
Ahead of time compilation

Abstract hardware

C library

Modified for mobile: power

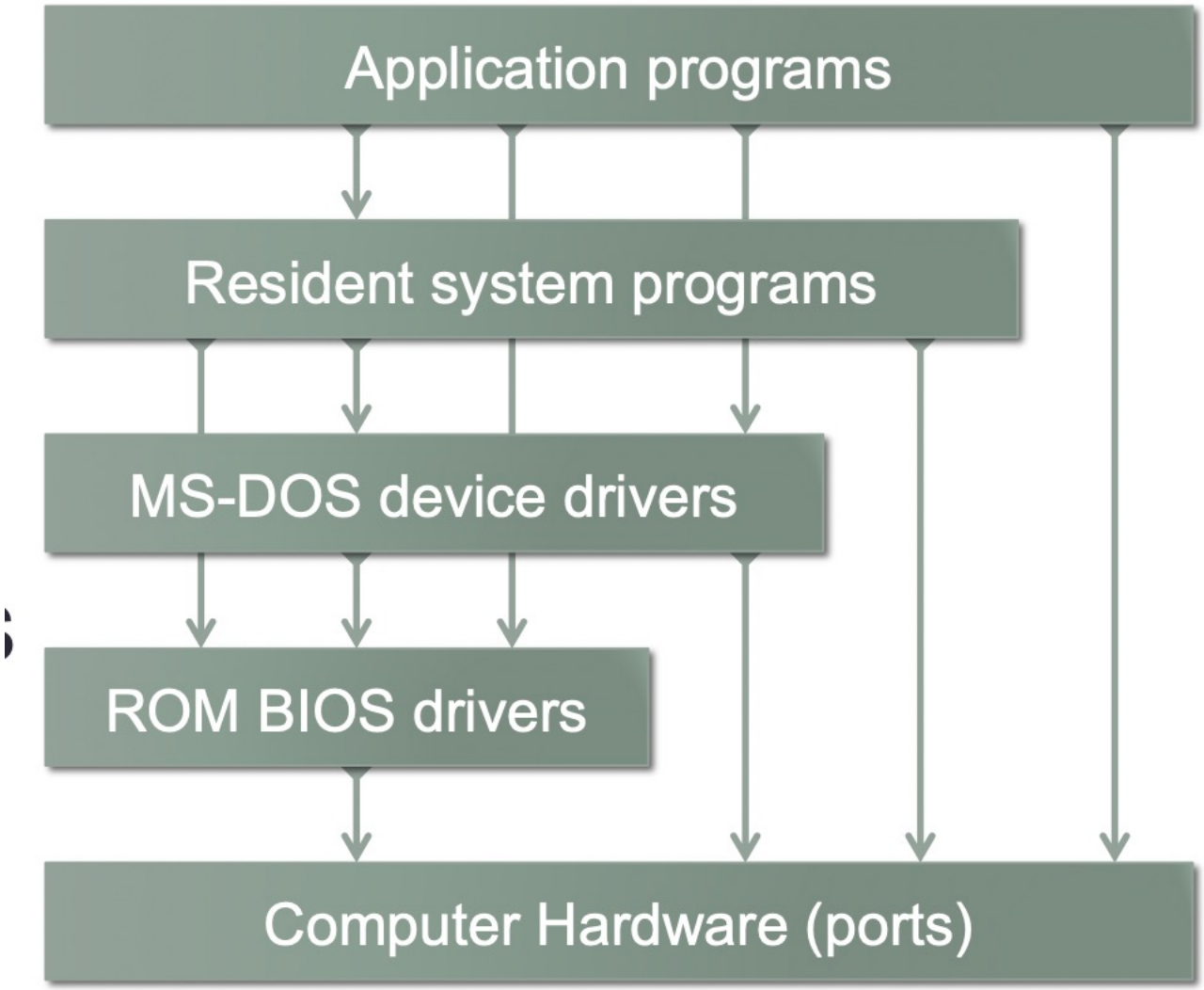
Windows Subsytem for Linux (WSL)



This is MS-DOS structure

What type of structure?

What benefits and problems can you imagine?



Read more

- How PCI works https://computer.howstuffworks.com/pci.htm
- How interrupts work in intel (chapter 6 of https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html)
- Plug and play
- https://www.pcmag.com/encyclopedia/term/plug-and-play

Homework

- Quiz!
- What interrupts are mapped in your OS?

Quiz

Definitions of computer system components

- ***CPU***—The hardware that executes instructions.
- ***Processor***—A physical chip that contains one or more CPUs.
- ***Core***—The basic computation unit of the CPU.
- ***Multicore***—Including multiple computing cores on the same CPU.
- ***Multiprocessor***—Including multiple processors.

Although virtually all systems are now multicore, we use the general term ***CPU*** when referring to a single computational unit of a computer system and ***core*** as well as ***multicore*** when specifically referring to one or more cores on a CPU.

Two important design issues for cache memory are ___.

Although cache memory is volatile (requires the computer to be powered-on to store data) , speed is not an issue as cache memory is the fastest form of memory after hardware registers.

The size of cache memory is small and because of this limitation, the replacement policy for managing data is crucial to gaining the most benefit from cache memory.

Another definition of OS

An operating system is the collection of data objects and procedures that are invoked by CPU on interrupts and exceptions (such as traps, division by 0, etc.).

The processor is the physical chip containing CPUs. Each CPU has a processing core.

Lecture 4: Process Management Part I

Paula Lago

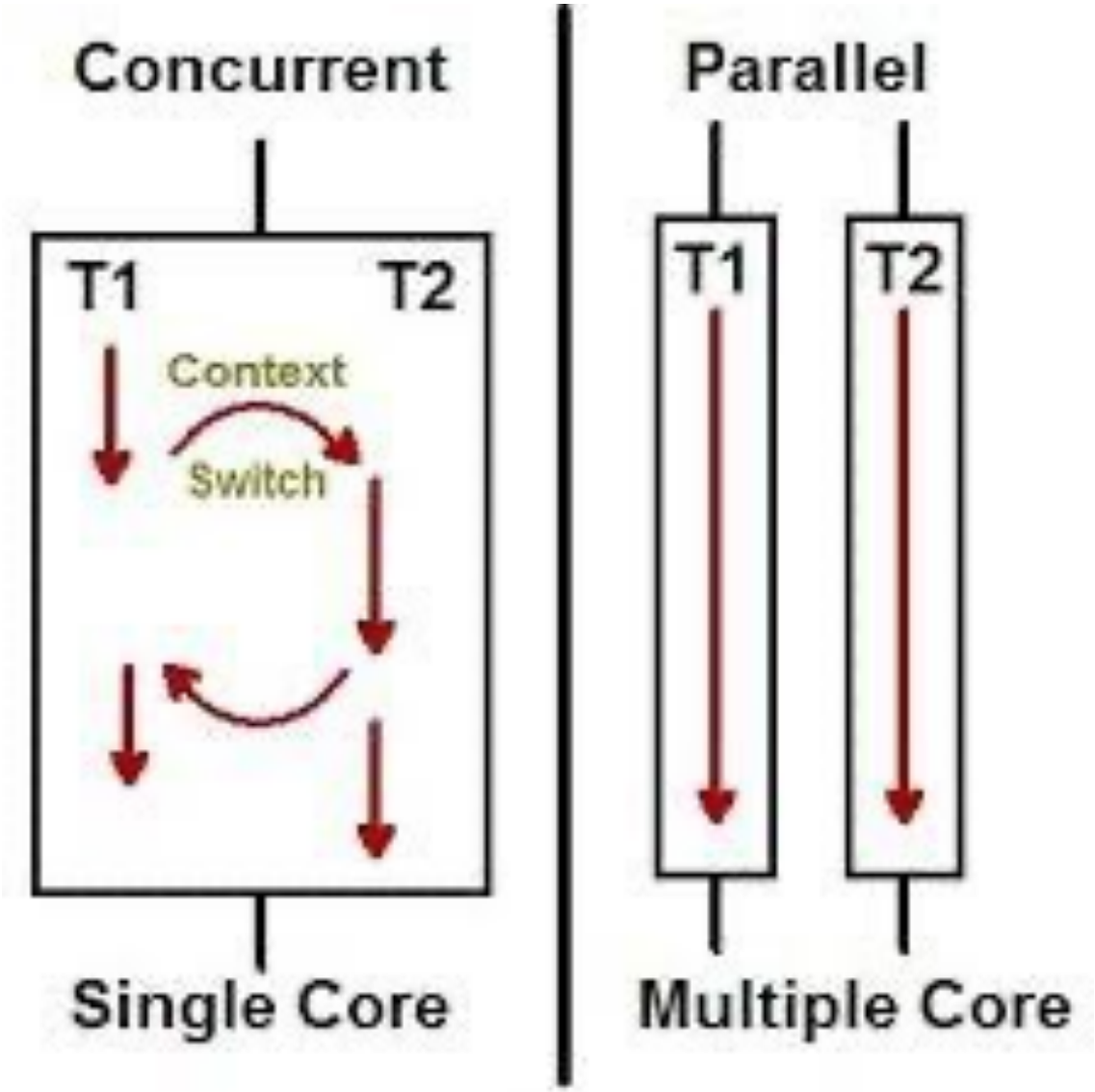
Paula.lago@concordia.ca

Agenda

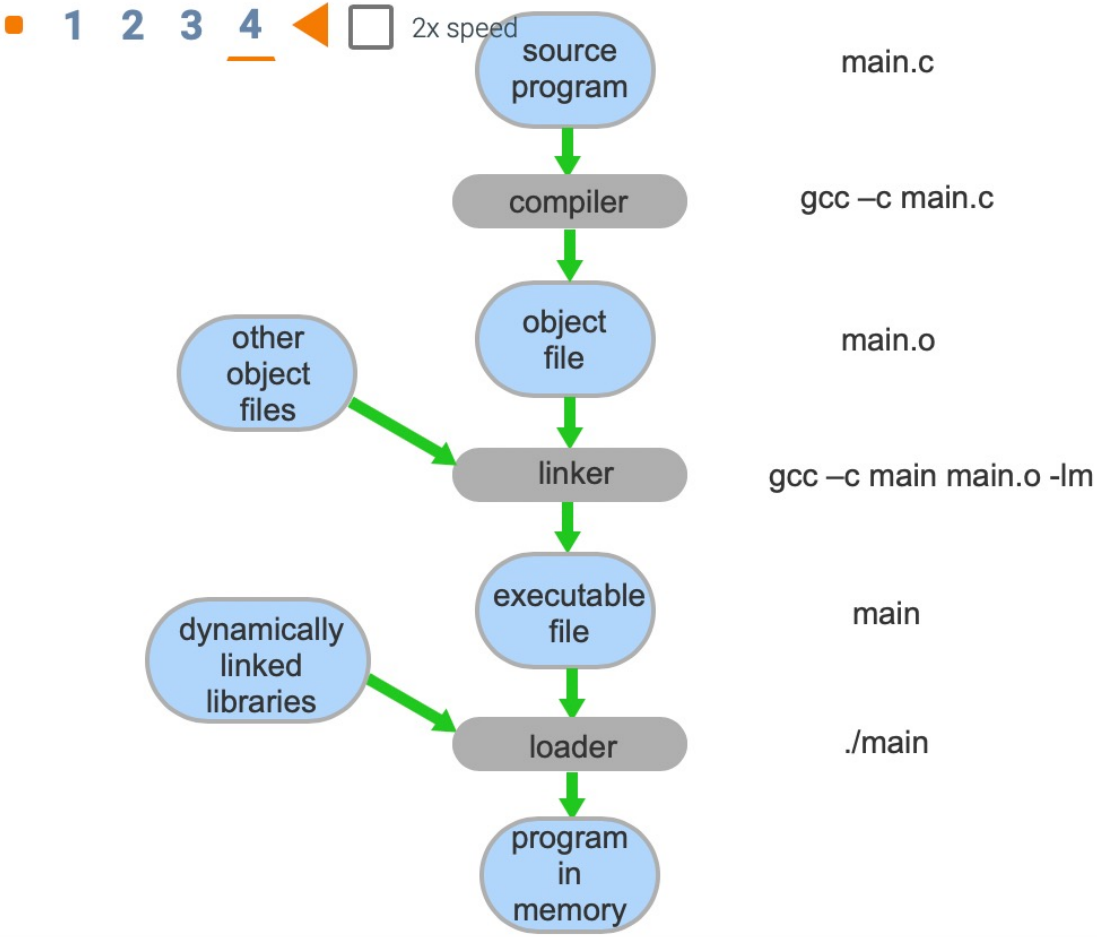
1. What is a process and how is it created?
2. What happens when a process is created?
3. How do processes terminate?

Processes

- ***process* is a program in execution.**
- A process is the unit of work in most systems.
- Processes may execute concurrently.

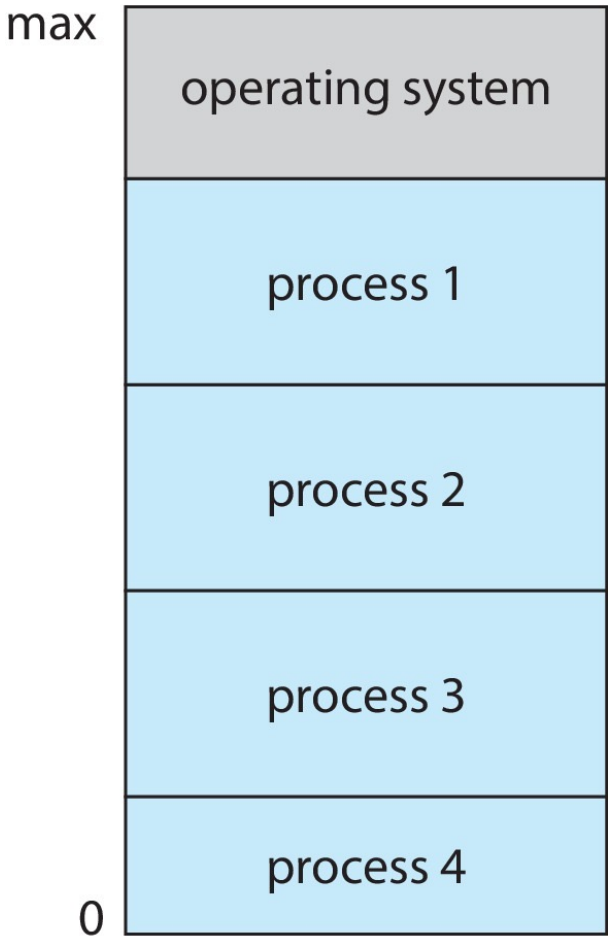


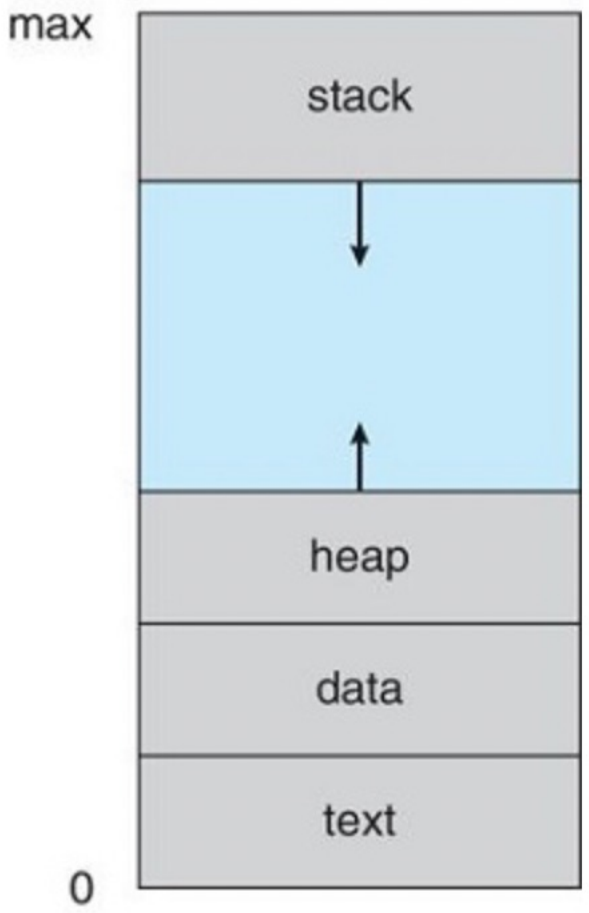
From programs to processes: linkers and loaders



1.Source program main.c is read by the C compiler gcc (in general, source program is read by the appropriate compiler).
2.Output is object (compiled) code, which is processed by the linker which finds and merges in other object files needed by the program.
3.Output is an executable program, ready to run.
4.At run time, when program is invoked, it is loaded into memory by the loader, which also loads other code (dynamically linked libraries). The addresses in the ./main process are adjusted to call the now-loaded library functions. Process can now execute.

Each process is given an address space





When the program is called,

1. A process is created (fork())

2. The loader is called with the name of the executable file.

3.It loads the program into the memory space given to the process (text section)

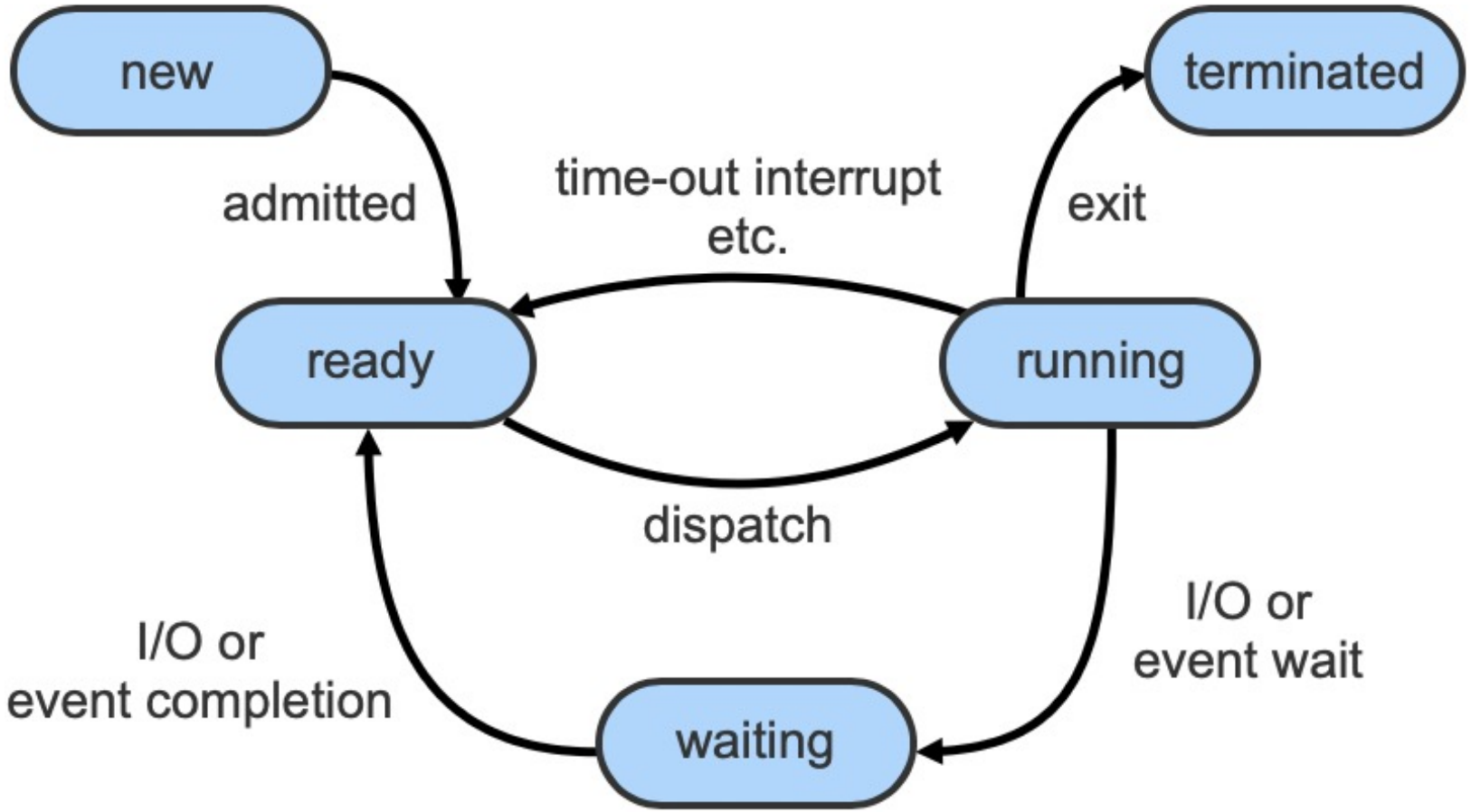
Process

- Active entity -> not the program
- Two process of the same program are separate entities
  - With equivalent text sections
- a process can itself be an execution environment for other code
  - Java vm

When the process is created, the OS needs to keep track of it

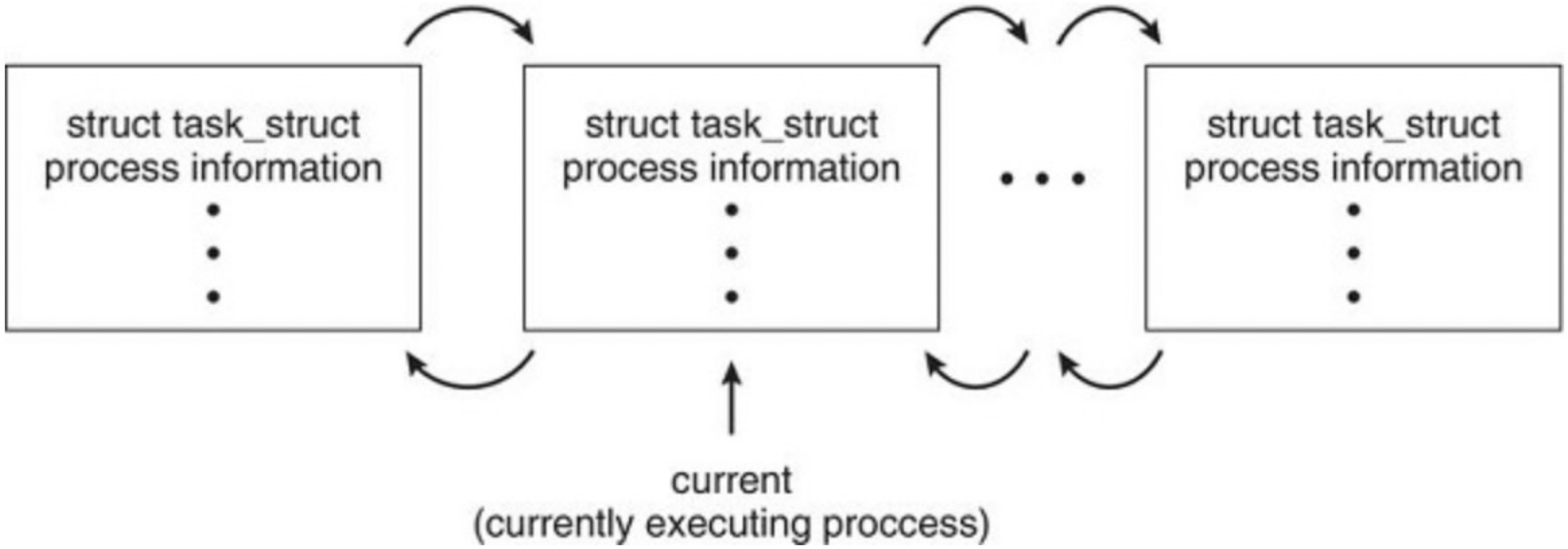
- A process is represented with a Process Control Block (PCB)

Process states



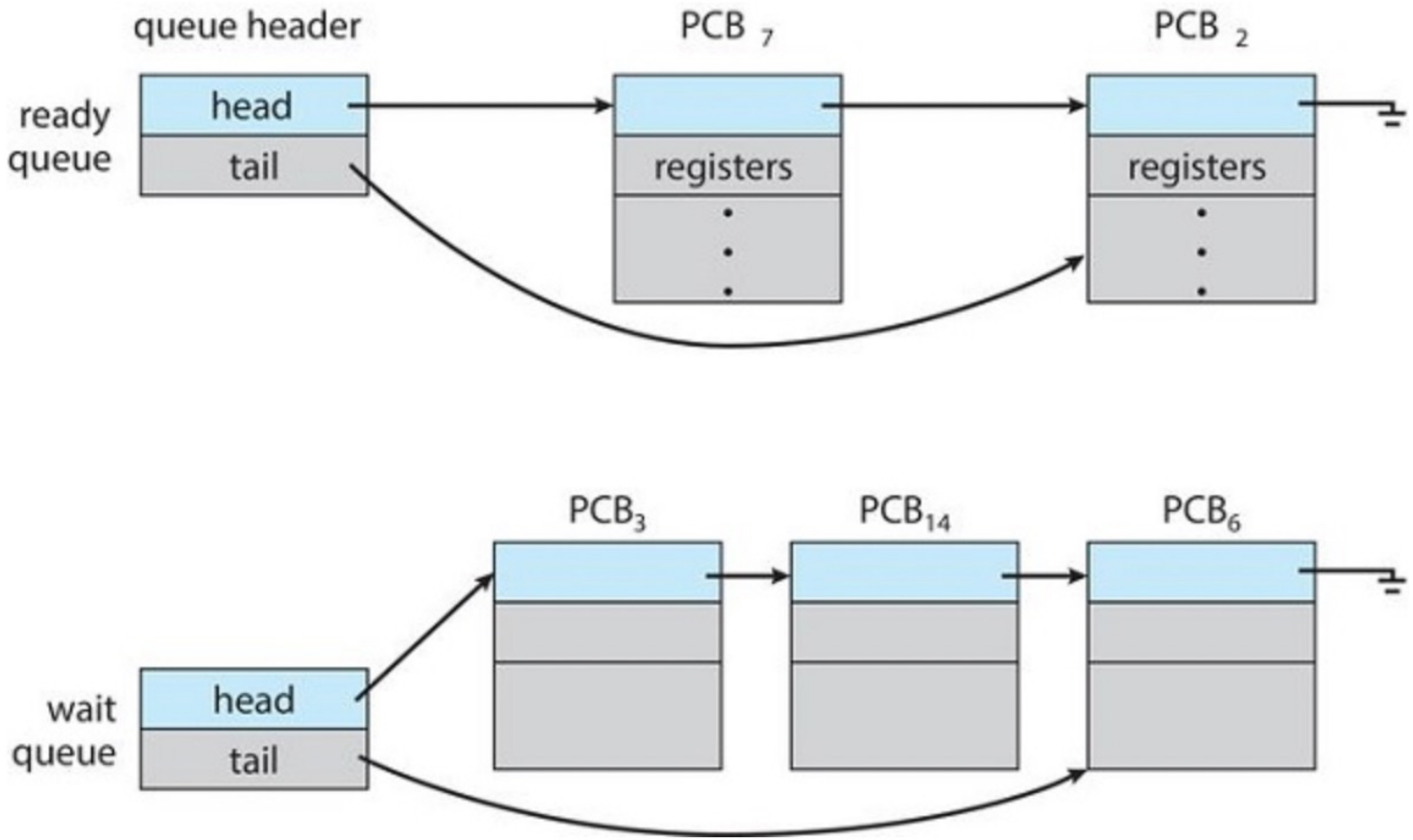
Process scheduling

Processes in the O.S. are represented with PCB. There is a list of PCB's

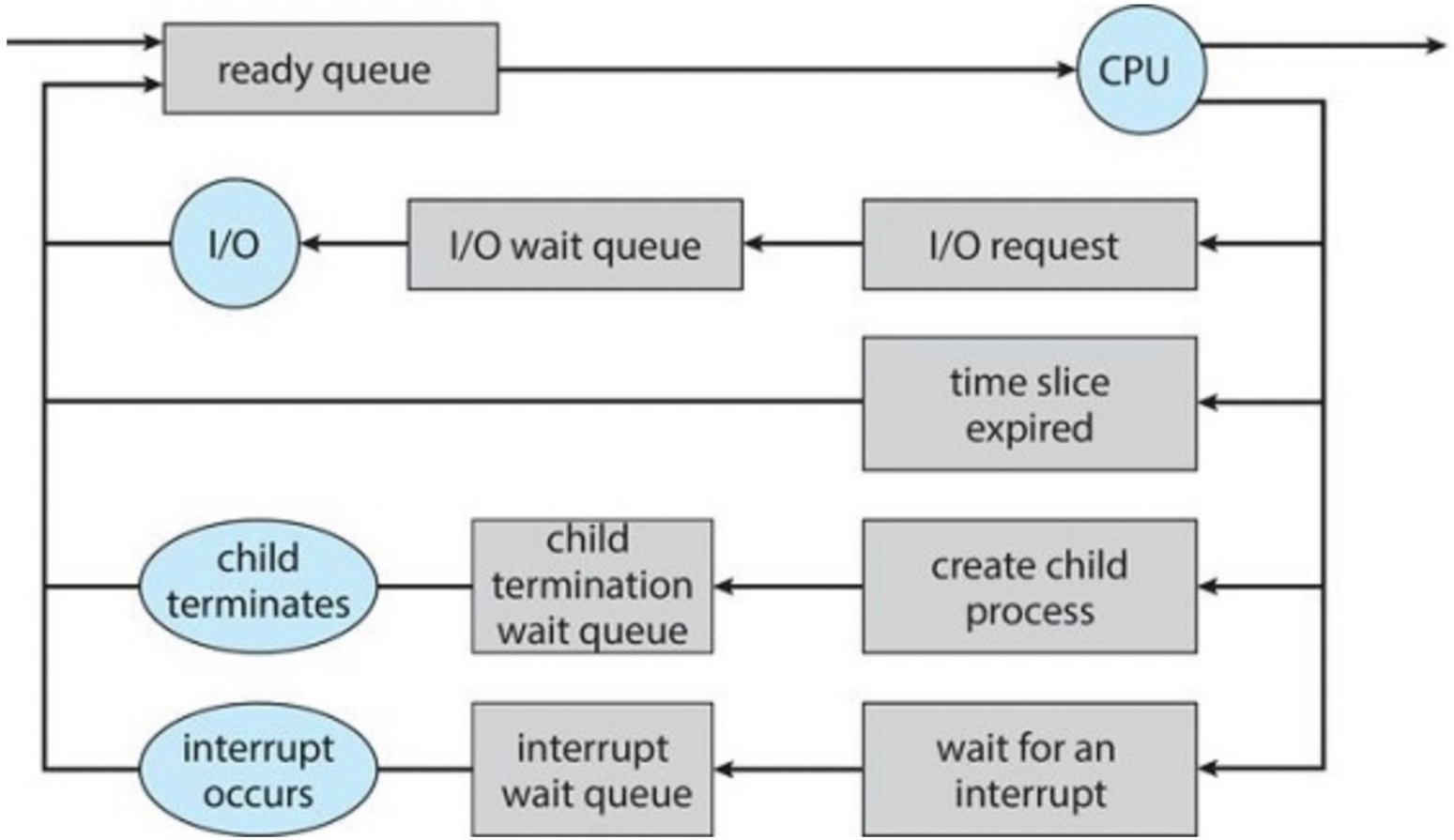


What other data structure would you use to find a process quickly based on its process id?

As processes enter, they are in the ready queue



A process transitions through the queues as events happen



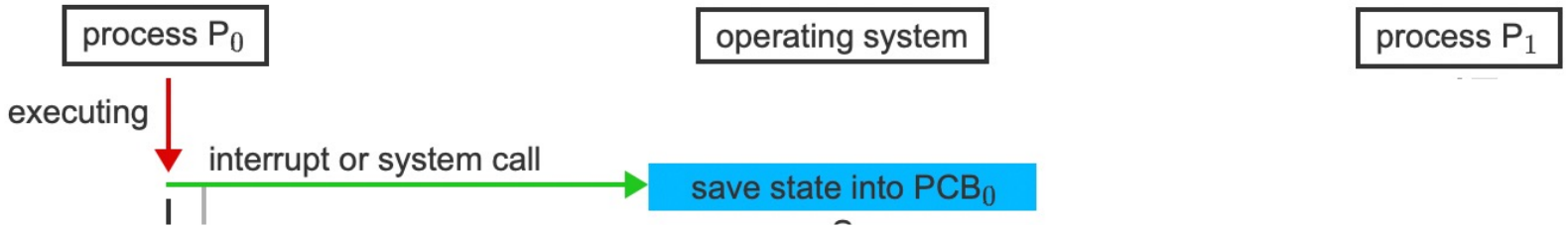
The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

CPU Scheduler

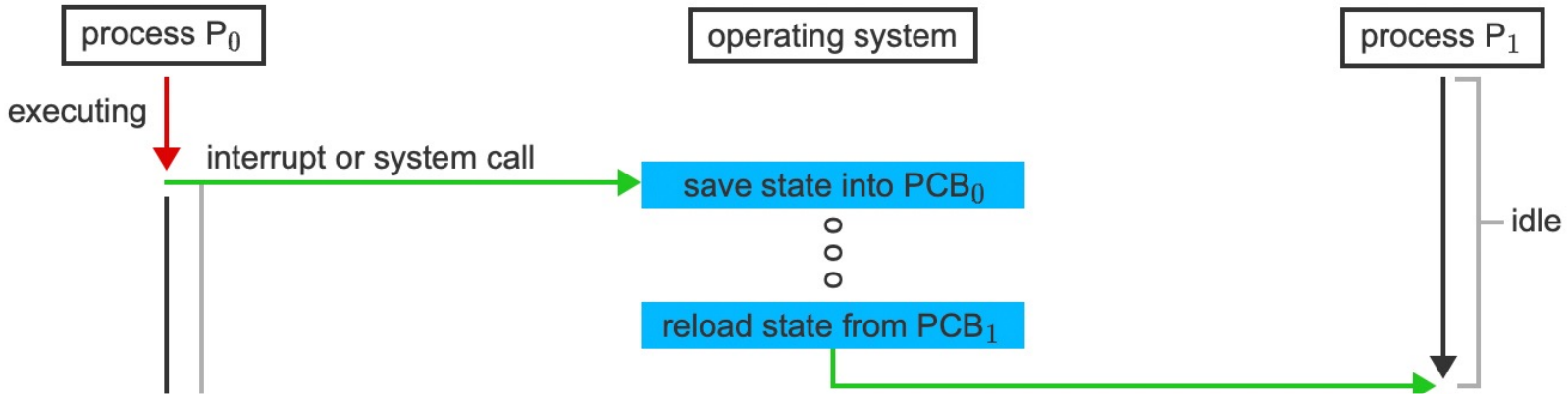
- A process that runs at least once every 100 ms and selects a new process from the ready queue and allocates a CPU core to it.

- Swapping
  - Move a process to disk to free memory, discussed in Memory Management
- Whenever the process being executed changes, there is a context switch

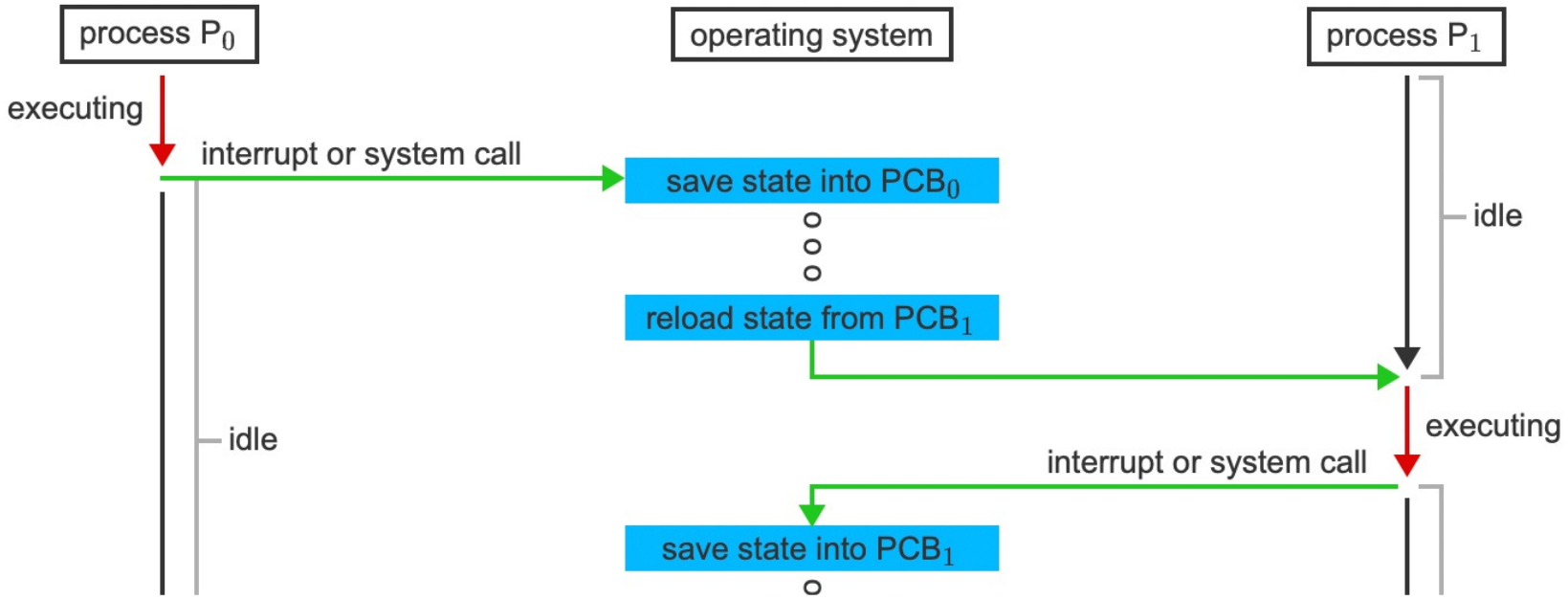
Context switch



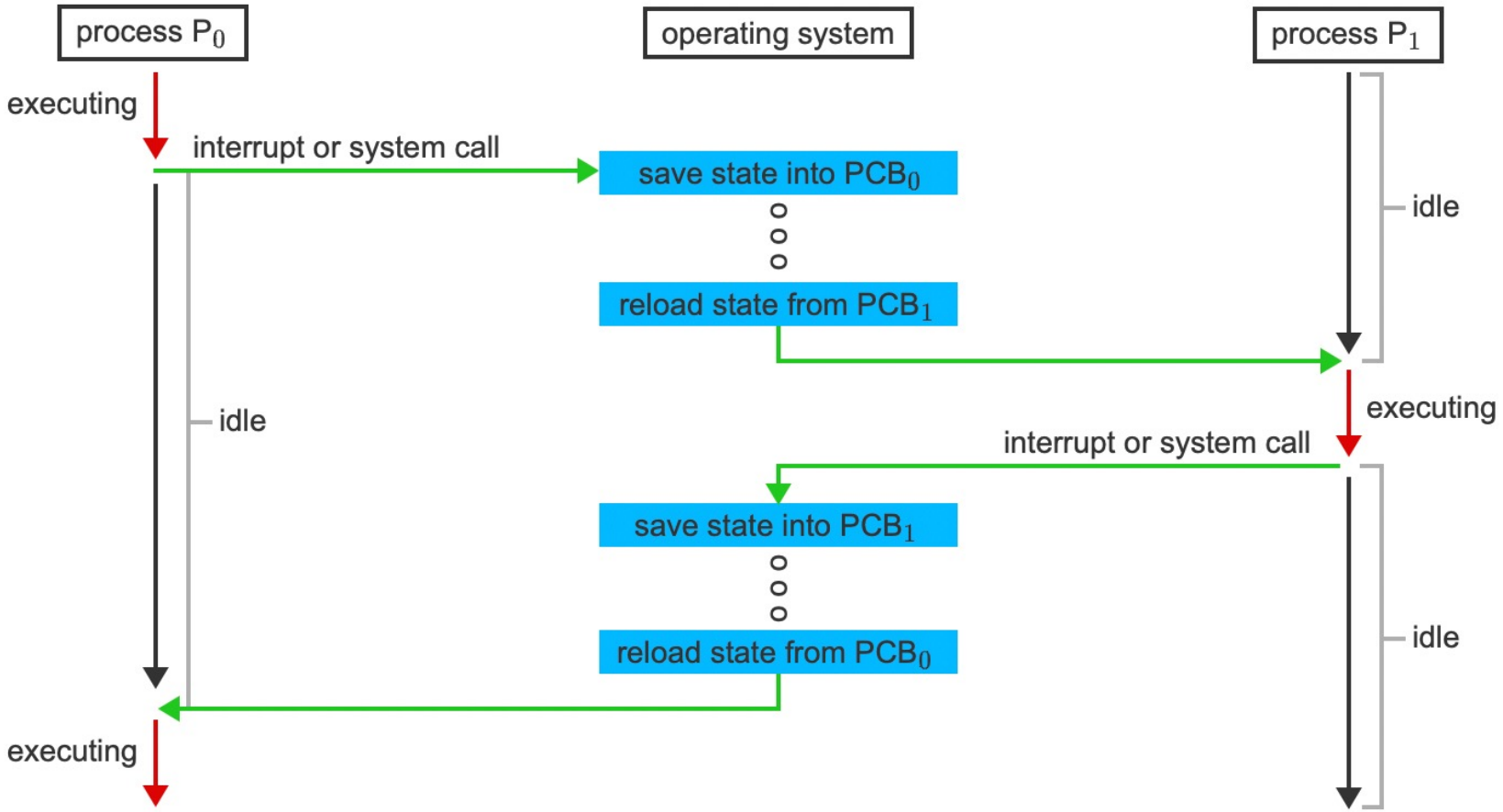
Context switch



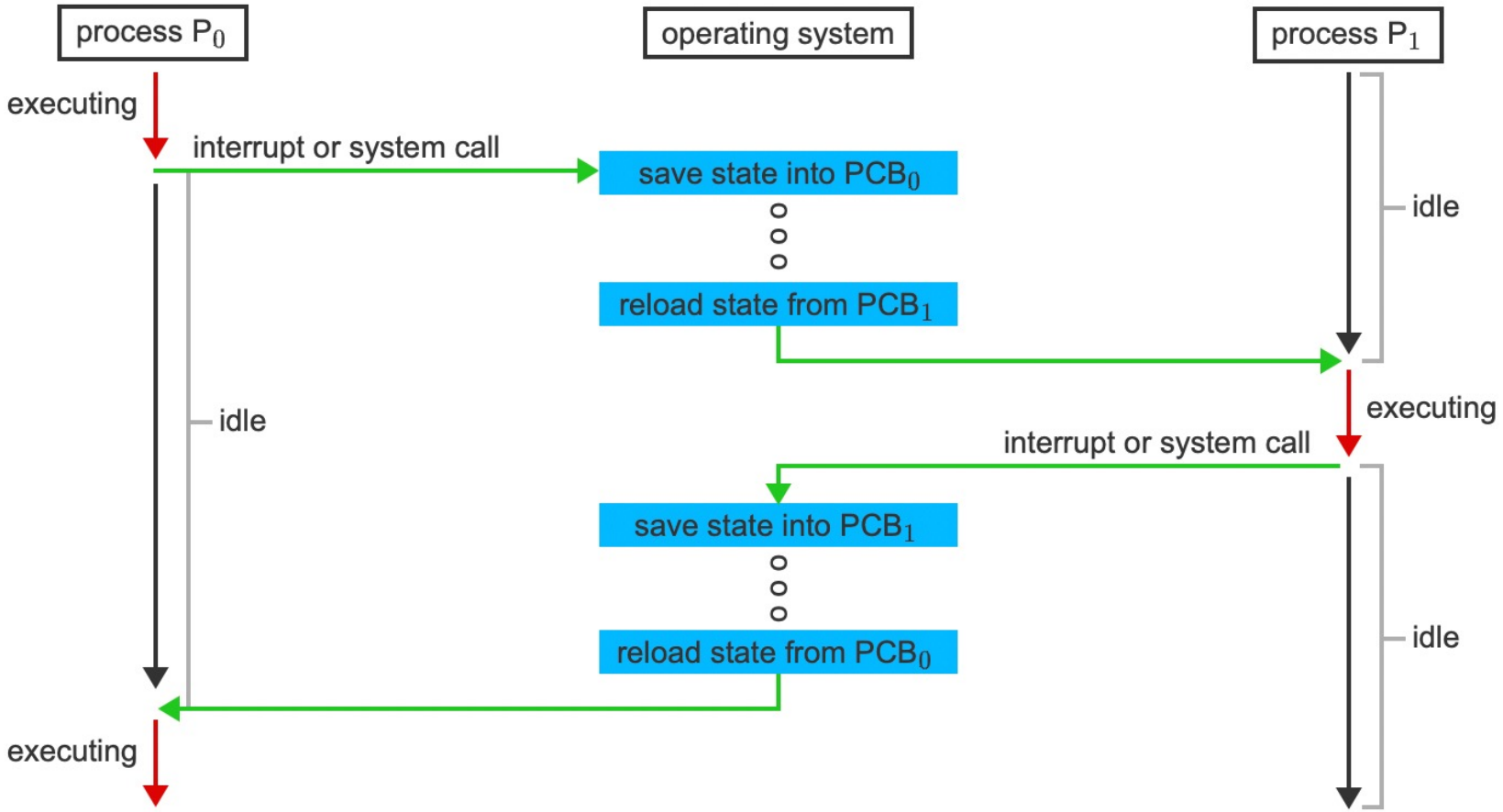
Context switch



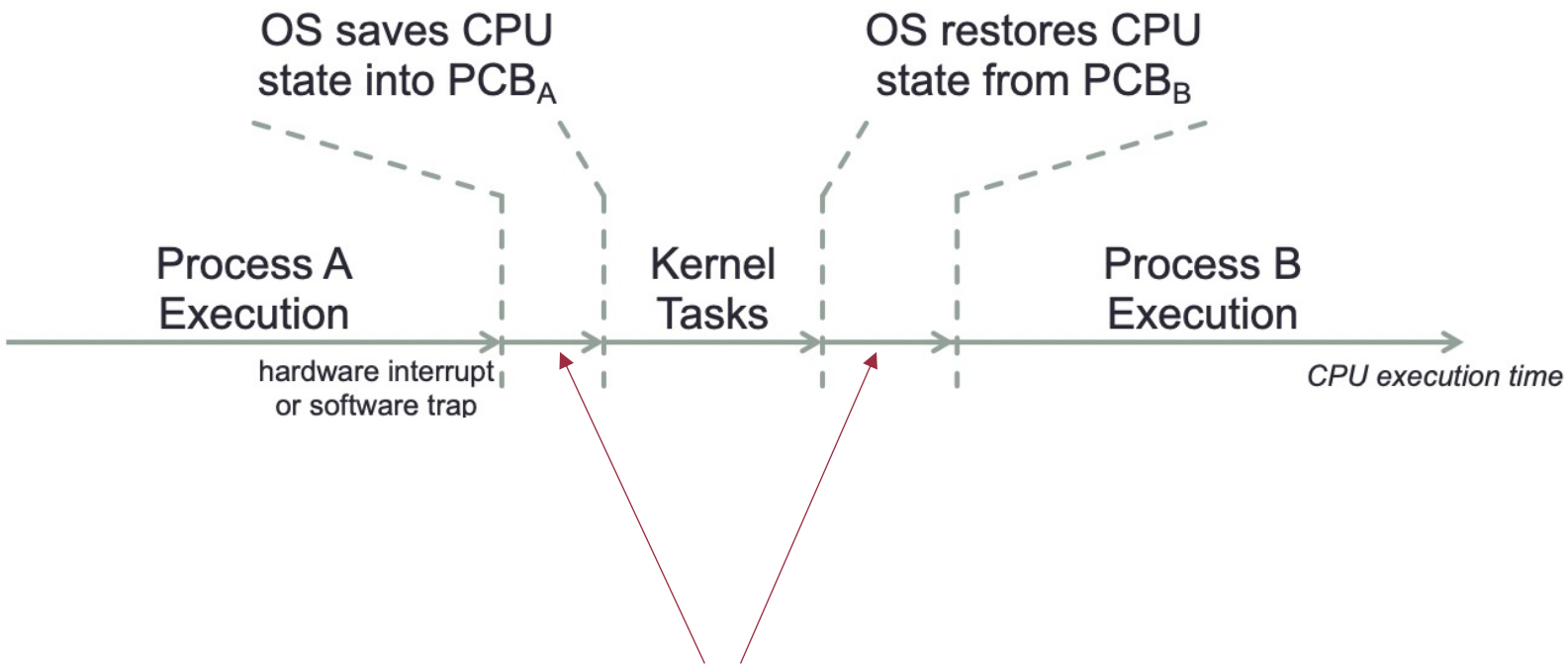
Context switch



Context switch



Context switching takes time!

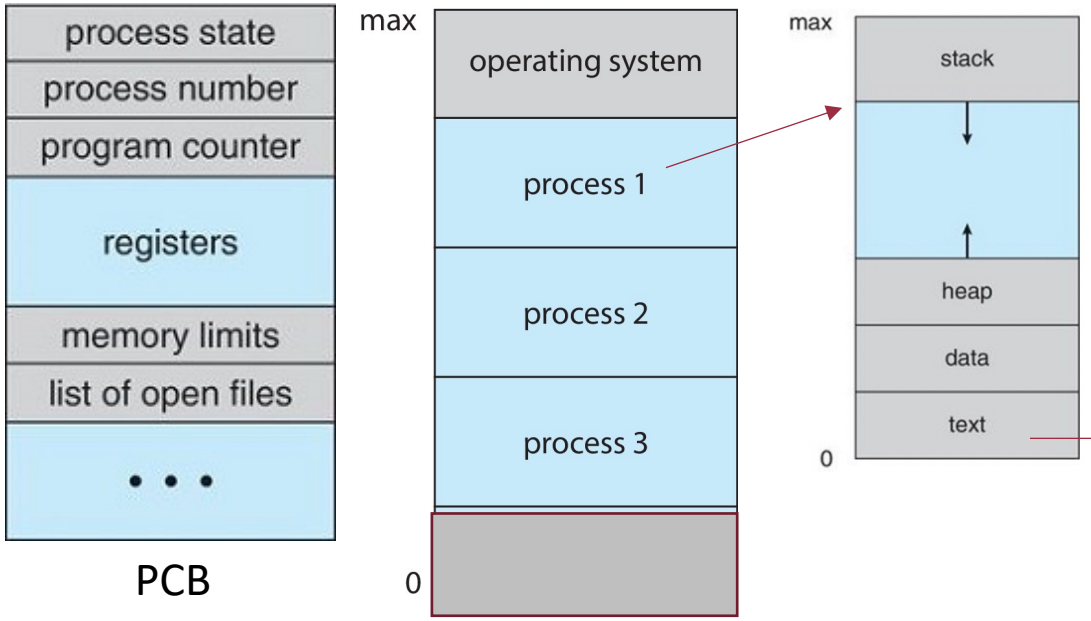


There is some time during which no user process is running

Operations on Processes

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination

A process is represented by the PCB and has a memory address space

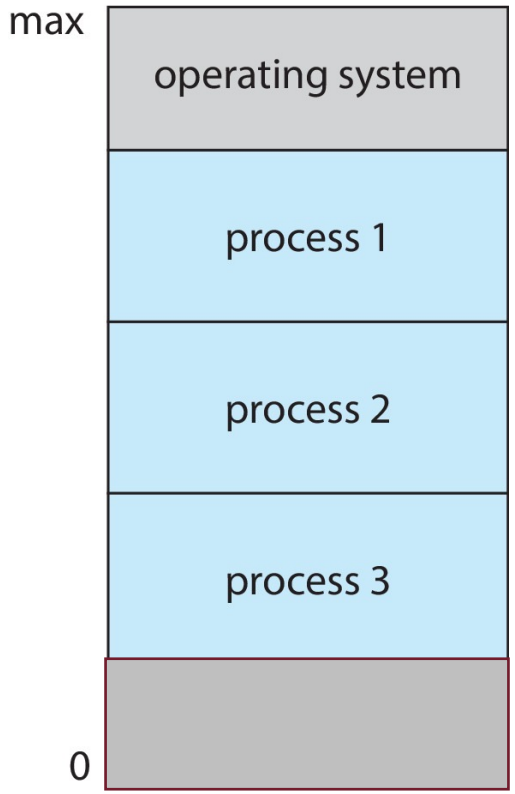



```
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

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

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

  return 0;
}
```

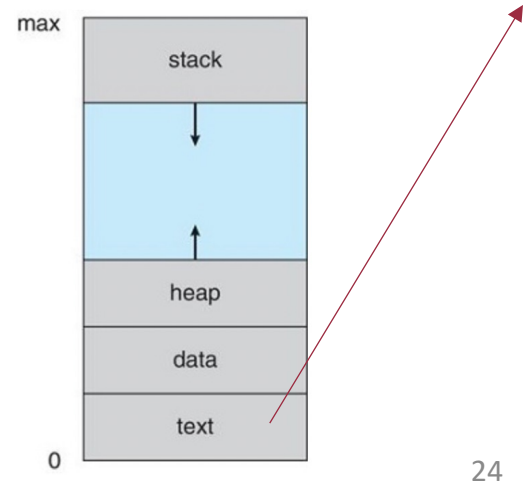
When fork system call is called, Parent process is duplicated (PCB and address space)

Ready

PCB of P1




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

int main()
{
pid_t pid;

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

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

  return 0;
}
```



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

int main()
{
pid_t pid;

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

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

  return 0;
}
```

P1

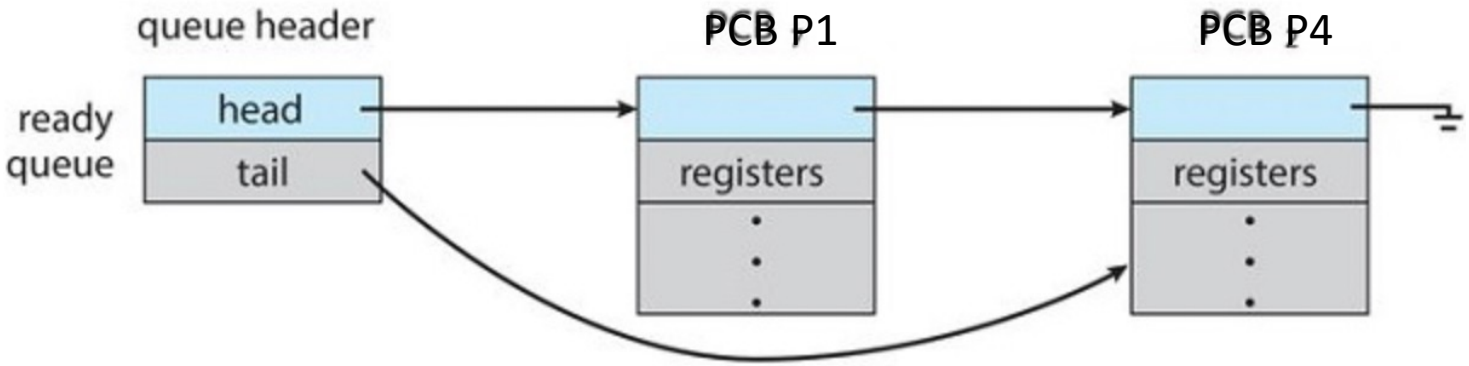
```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

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

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

  return 0;
}
```

P4

Fork returns

- < 0 if error during process creation
- = 0 for child process
- pid of the new process for parent process

Both P1 and P4 are in the ready queue!



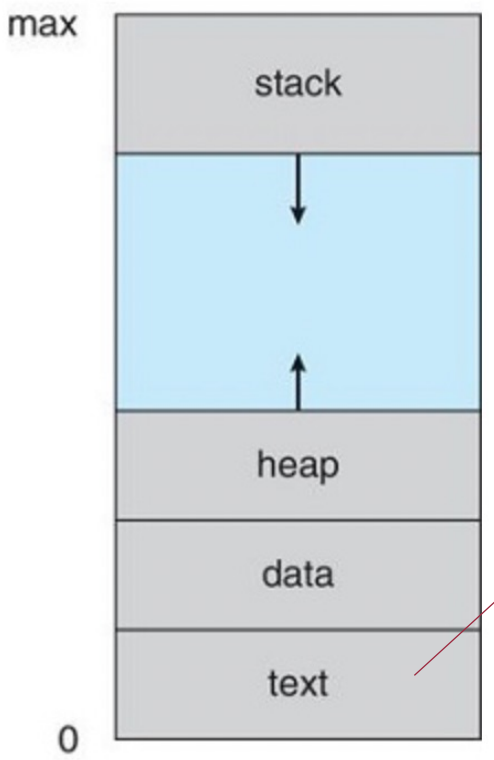
Any one of them can be chosen to continue

The child

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

int main()
{
pid_t pid;

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

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

  return 0;
}
```

pid = 0
(pid variable is the return value of fork, not its actual pid)

Exec system call loads a new program into the text space




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

int main()
{
pid_t pid;

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

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

  return 0;
}
```

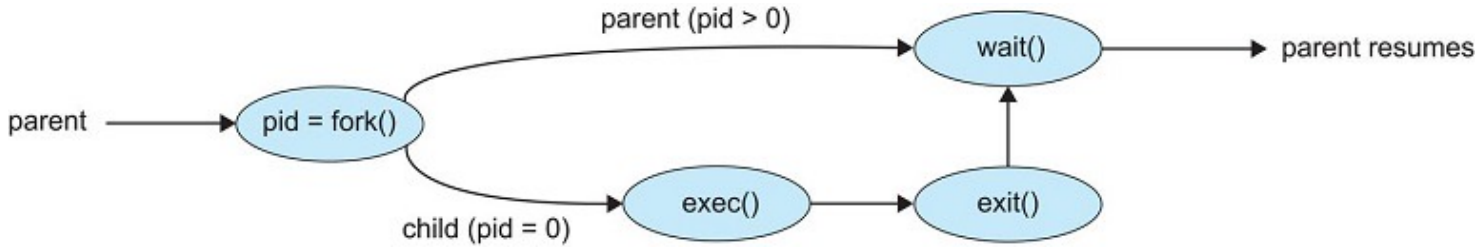
After exec, the text will be that of /bin/ls

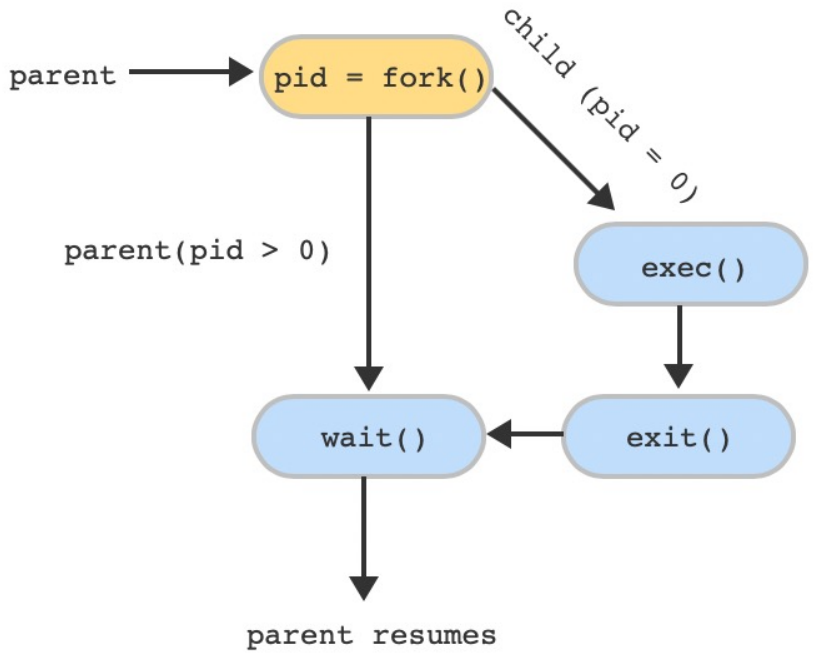
The parent

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

int main()
{
pid_t pid;

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

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

  return 0;
}
```

pid = 4 (child process id)

Wait system call means wait for child to finish
-> process is moved to waiting queue






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

int main()
{
pid_t pid;

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

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

  return 0;
}
```

Exercise

6. Predict the output of the following program

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

void forkexample()
{
    int x = 1;

    if (fork() == 0)
        printf("Child has x = %d\n", ++x);
    else
        printf("Parent has x = %d\n", --x);
}
int main()
{
    forkexample();
    return 0;
}
```

Exercise

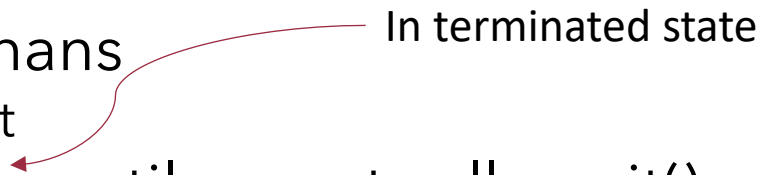
17. How many times is hello printed?

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

L1
L2
L3

Process Termination

- A process may terminate because
  - It has finished its execution
  - Unrecoverable fault
  - Another process kills it
- If a process terminates
  - Children terminate or become orphans
    - Another process becomes their parent
  - IF child process -> becomes zombie until parent calls wait() and reads its status
- Daemon processes : continue to run in the background even if parent terminates



Cooperating processes

Cooperating processes

- Cooperating processes are those that share state. (May or may not actually be "cooperating")
- Behavior is *nondeterministic*

Reasons to cooperate

- **Information sharing**: Several processes may need to access the same data (such as stored in a file.)
- **Computation speedup**: A task can ofter be run faster if it is broken into subtasks and distributed among different processes.
- **Modularity**: It may be easier to organize a complex task into separate subtasks, then have different processes or threads running each subtask.
- **Convenience**: An individual user can run several programs at the same time, to perform some task.

Processes can cooperate if they share the same file

- Only if they share the same file system
- Slow access
- Difficult to synchronize (files are easy to access)

Inter process communication mechanisms

- Two methods
  - Shared memory: shared memory space for two or more process
  - Message passing
- (Third) signals

Signals

- Very short messages
  - No arguments only signal number
- Signals serve two main purposes:
  - To make a process aware that a specific event has occurred
  - To cause a process to execute a *signal handler* function included in its code
-

Two phases of signals

- *Signal generation:* The kernel updates a data structure of the destination process to represent that a new signal has been sent.
- *Signal delivery:* The kernel forces the destination process to react to the signal by changing its execution state, by starting the execution of a specified signal handler, or both

- A signal is pending if it has not been delivered.

Signal handling

- When switching from Kernel Mode to User Mode, the kernel checks whether a signal for a process has arrived.
- Determine whether the signal can be ignored.
- Handle the signal, which may require switching the process to a handler function at any point during its execution (i.e. catch code) and restoring the original execution context after the function returns. Usual actions:
  - Ignore
  - Dump
  - Continue
  - Terminate

Exceptions

- Code
  - On exception raises interrupt (error)
  - Hardware looks up in IDT and goes to handler routine
- Interrupt handler
  - Signal process
- Receive signal and go to handler code

Implementations vary across OS

Consumer-Producer Model

unbounded-buffer places no practical limit on the size of the buffer:

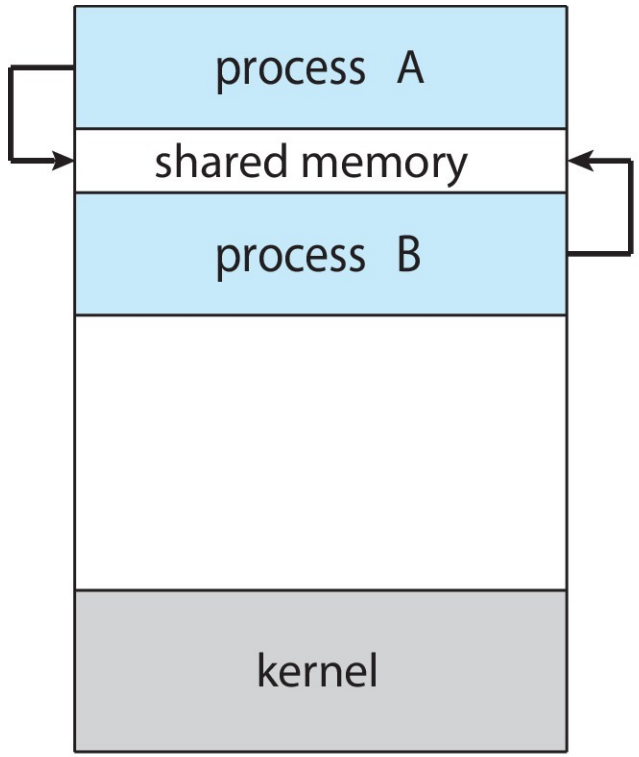
- Producer never waits
- Consumer waits if there is no buffer to consume

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

Shared memory

- A space in memory that both processes can access



(a)

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.

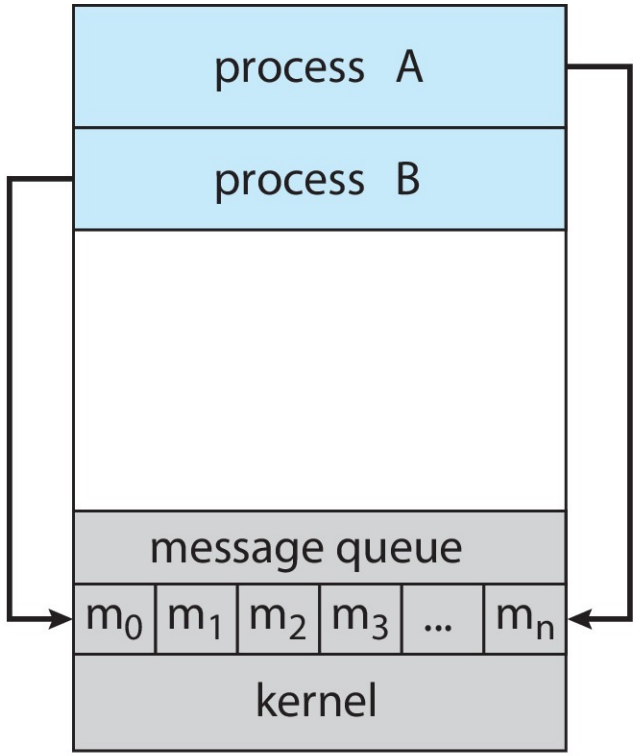
Shared memory synchronization

Each process:

- Reads counter
- Updates value
  - Producer c=c+1
  - Consumer c=c-1
- Writes new value

Will learn how to prevent this when we talk about synchronization!

Message passing



(b)

Processes communicate with each other without resorting to shared variables

IPC facility provides two operations:

`send`(*message*)

`receive`(*message*)

The *message* size is either fixed or variable

Message Passing

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Direct communication
  - Processes must name each other explicitly (only two processes)
    - **`send`** (*P, message*) – send a message to process P
    - **`receive`**(*Q, message*) – receive a message from process Q
- Indirect communication
  - Messages are directed and received from mailboxes (also referred to as ports)
    - Each mailbox has a unique id (many processes can share it)
    - Processes can communicate only if they share a mailbox
  - **`send`**(*A, message*) – send a message to mailbox A
  - **`receive`**(*A, message*) – receive a message from mailbox A

Synchronization in message passing

Message passing may be either blocking or non-blocking

- Blocking is considered synchronous
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- Non-blocking is considered asynchronous
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a rendezvous

Homework

- Quiz!
- Optional for participation points: Solve the guide for lecture 6 (and bring it next lecture to hand in)

COEN 346 OPERATING SYSTEMS – WINTER 2023

Introduction

These activities explore various aspects of supporting processes in an operating system and how a user can create processes programmatically.

Model I. Process States (5 minutes)

The following is a picture of the states a process can be in, and how a process transitions between states.

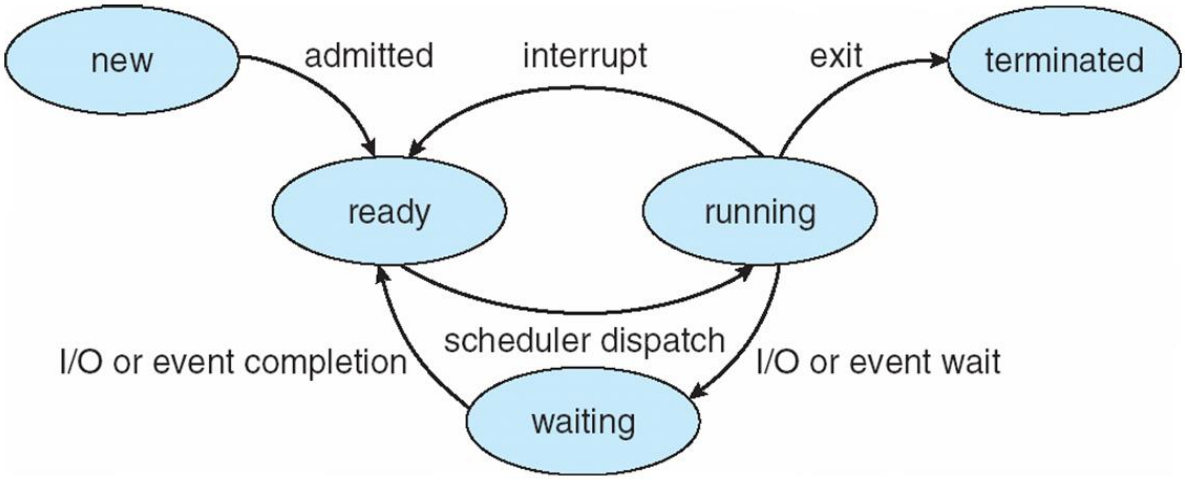


Figure 3.2 from Operating Systems Essentials, 2nd ed. by Silberschatz, Galvin, Gagne; John Wiley, Inc.

Critical Thinking Questions

1. What causes a process to move from **running** to **ready**?

2. After a process is admitted to the system, what state is it in?

3. What two things might a process do that would move it into **waiting** state?

4. Speculate: can multiple processes be in **running** state at the same time? Can multiple processes be in the **ready** state at the same time?

5. Fill in the following summary of process states

Match each state transition to the corresponding event.

If unable to drag and drop, refresh the page.

Running -> Ready | **Running -> Waiting** | **Running -> Terminated** | **Ready -> Running** | **Waiting -> Ready**

	Process exits.
	Process is selected to run on a processing core.
	A timeout has occurred.
	Process needs to wait for an event (such as completion of an I/O operation).
	The event for which the process was waiting has occured.

Model II: CPU Switch from Process to Process (5 minutes)

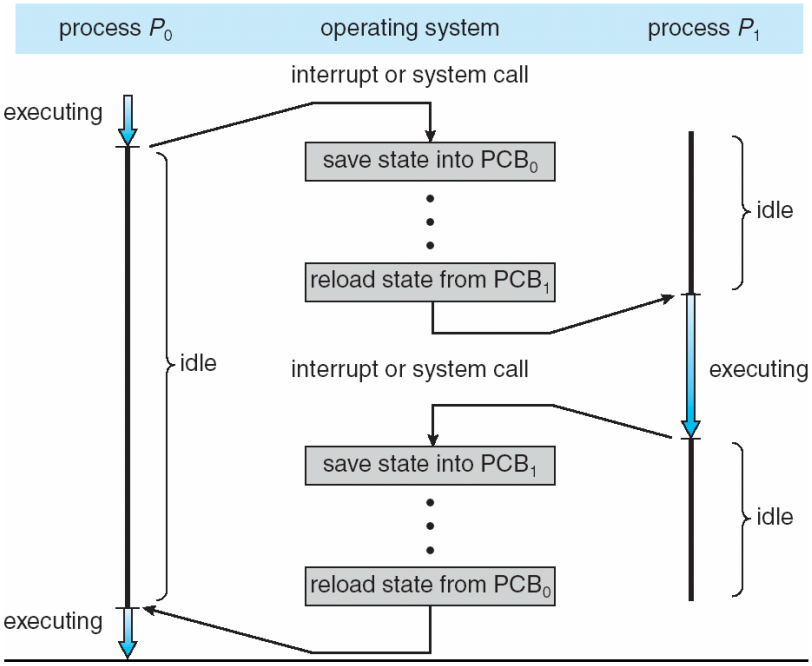


Figure 3.4 from Operating Systems Essentials, 2nd ed. by Silberschatz, Galvin, Gagne; John Wiley, Inc.

A PCB is a **Process Control Block** -- a data structure created by the operating system to hold information about each process. The illustration shows a **context switch**, from P_0 to P_1 and back to P_0.

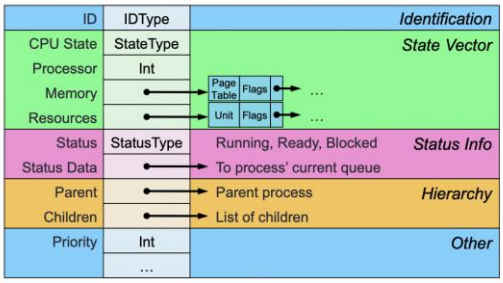
6. How many processes are represented in the model of the figure?

7. What do the solid black lines (like the ones labeled “interrupt or system call”) mean?

8. Why would making a system call cause the operating system to save the state of P_0, load the state of P_1, and start executing P_1?

9. Speculate: When the OS "saves the state into PCB_0", what information might it be saving?

10. Given the following PCB structure, can you write the class attributes to define it?



Model III: Creating Processes on Linux/Unix (5 minutes)

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

int main()
{
pid_t pid;

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

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

    return 0;
}
```

Consider the code above:

11. What does a call to `fork()` do?

12. If `fork()` returns the value 0, what process is executing?

13. If `fork()` returns a value > 0, what process is executing?

14. Suppose the value of pid is >= 0. How many processes are running the line
    `if (pid < 0) {`

15. What does `execlp()` seem to do?

16. Predict the output of the following program

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

void forkexample()
{
      int x = 1;

      if (fork() == 0)
            printf("Child has x = %d\n", ++x);
      else
            printf("Parent has x = %d\n", --x);
}
int main()
{
      forkexample();
      return 0;
}
```

17. How many times is hello printed?

#include <stdio.h>

```
#include <sys/types.h>
int main()
{
      fork();
      fork();
      fork();
      printf("hello\n");
      return 0;
```

}

COEN 346 OPERATING SYSTEMS – WINTER 2023

Introduction

These activities explore various aspects of how processes can communicate with shared memory or message passing.

Model IV: Producer-Consumer Problem, Shared Memory, Bounded Buffer

The following code illustrates a classic and common problem where one process produces data and another process consumes it. This is a common problem in Operating Systems: e.g., an ethernet device driver's interrupt handler reads a packet off of the ethernet chip, and then needs to send that packet to another process to handle it (i.e., consume it). This code uses a shared-memory, fixed-sized "circular" buffer.

```
final int BUFFER_SIZE = 10;

class Item {
      // class body goes here
}

Item[] buffer = new Item[BUFFER_SIZE];
int in = 0;  /* next location to write to */
int out = 0; /* next location to read from */
```

1. What are the initial values for `in` and `out` and how many items are in the buffer, initially?

2. Write the if statement that a process would use to check if the buffer were empty.

3. Now, suppose that a producer process creates an item to be inserted in the buffer. After inserting it at `buffer[in]`, what value should `in` be changed to?

4. Suppose that `out` has the value 1 and there are 3 items in the buffer ready to be consumed. What value does `in` have?

4

5. Write the if statement that a process would use to check if the buffer is full.

6. How many items can a buffer of size BUFFER_SIZE hold?

Model V: Produce-Consumer with Message Passing (20 minutes)

Suppose that our operating system provides a nice message passing mechanism with this API:

```
void send(String message);
void receive(StringBuffer message);
```

Suppose also that the message passing mechanism has a limit of messages that can be sent before some must be received. That limit is 10. When that limit is reached, a send() call will block until the message can be sent. Similarly, the receive() call will block only if there are no messages available to be received.

7. Write the producer code using send() to send the data to the consumer.

8. Write the corresponding consumer code, using receive().

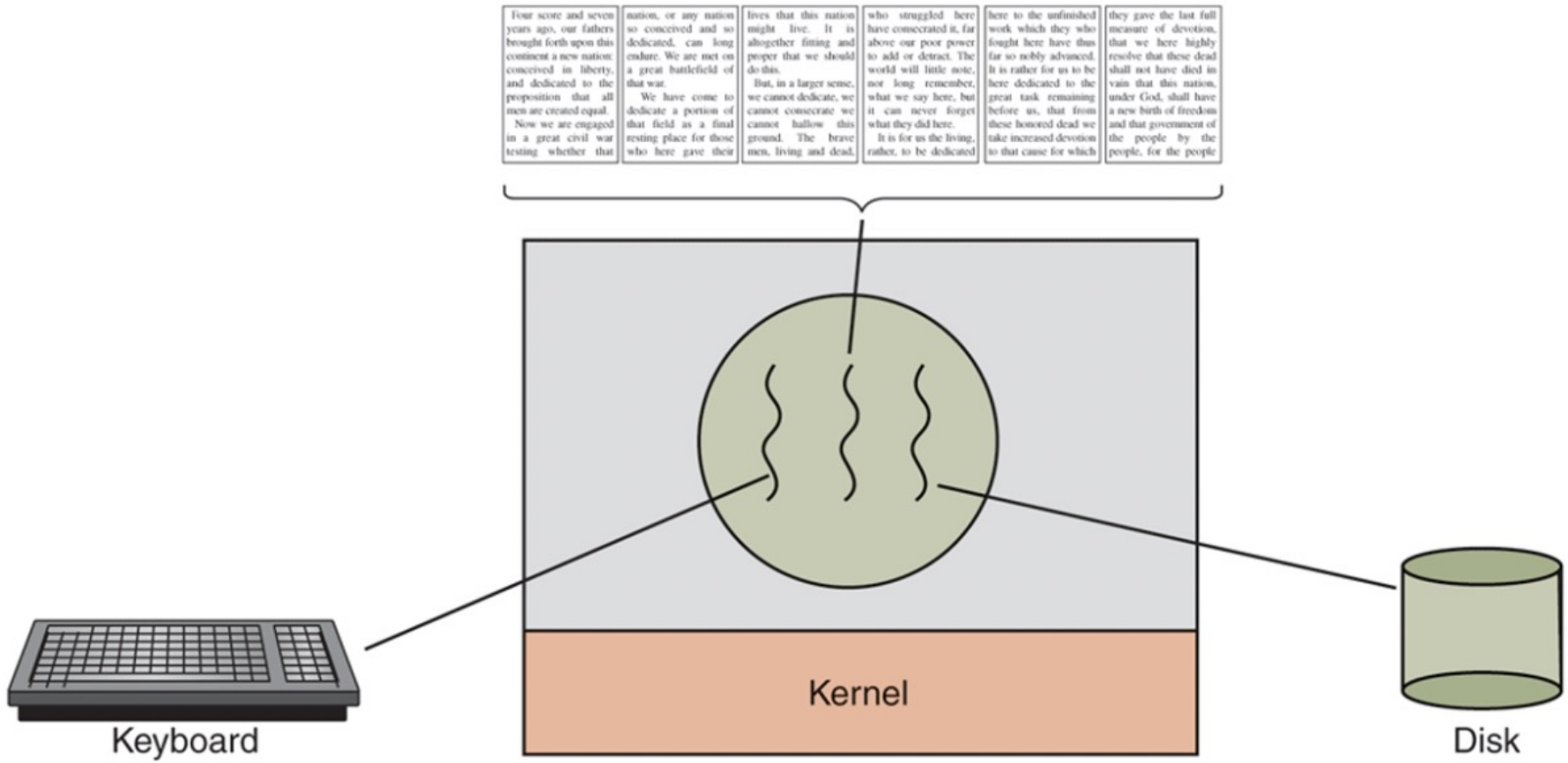
9. Suppose that neither send() nor receive() block, but instead immediately return the value 0 when a message cannot be sent for some reason (for send()), or a message cannot be received because no message is available (for receive()). Rewrite the producer and consumer code to handle this situation.

This guide was modified by the guide provided by Victor Norman at Calvin College, 2019
vtn2@calvin.edu

COEN 346 OPERATING SYSTEMS – WINTER 2023

Threads

Figure 2-7



A word processor with three threads.

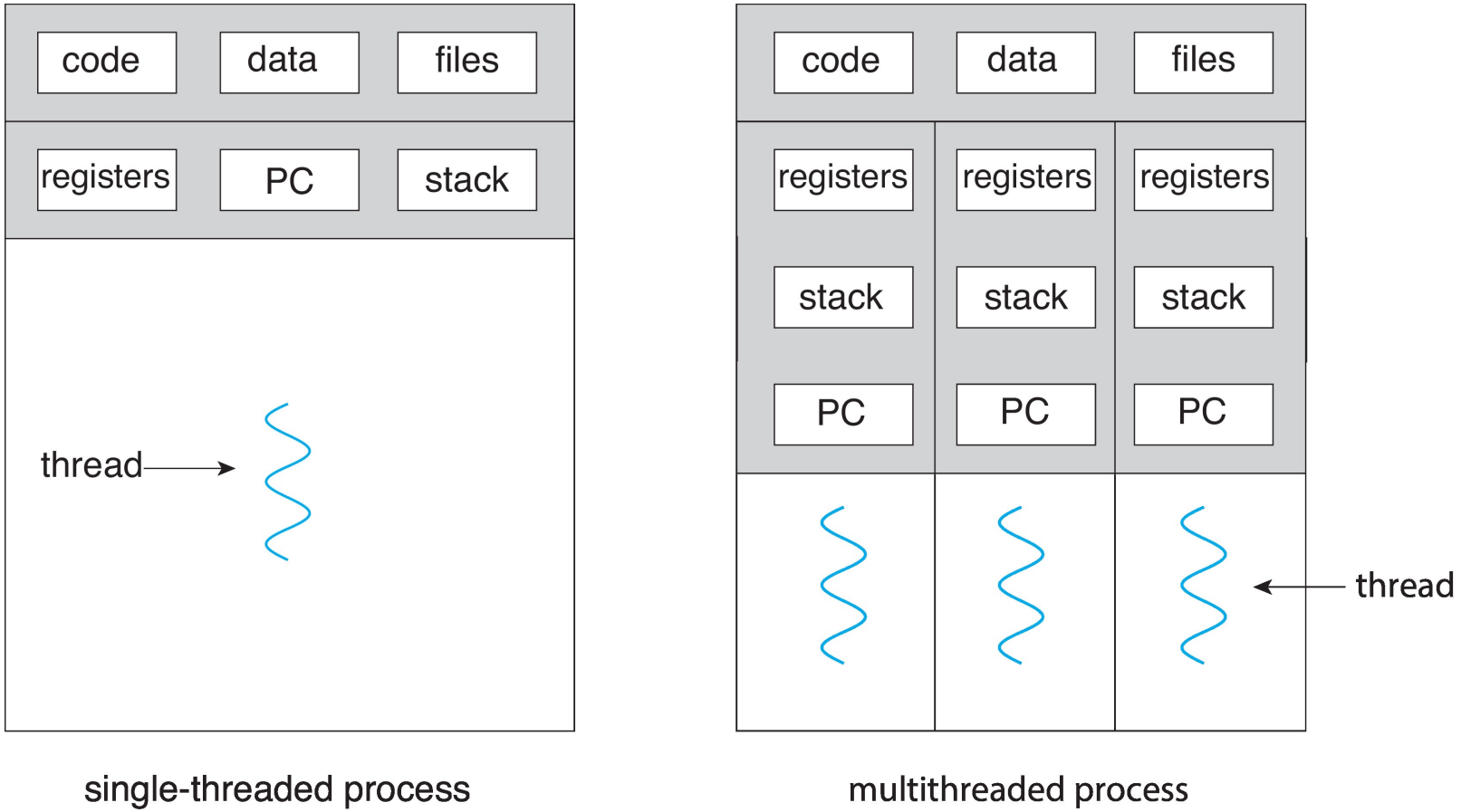
02-07.png

Creating processes is an expensive operation

- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request

The tasks share code and data, why duplicate it?

A process is a group of related resources



single-threaded process

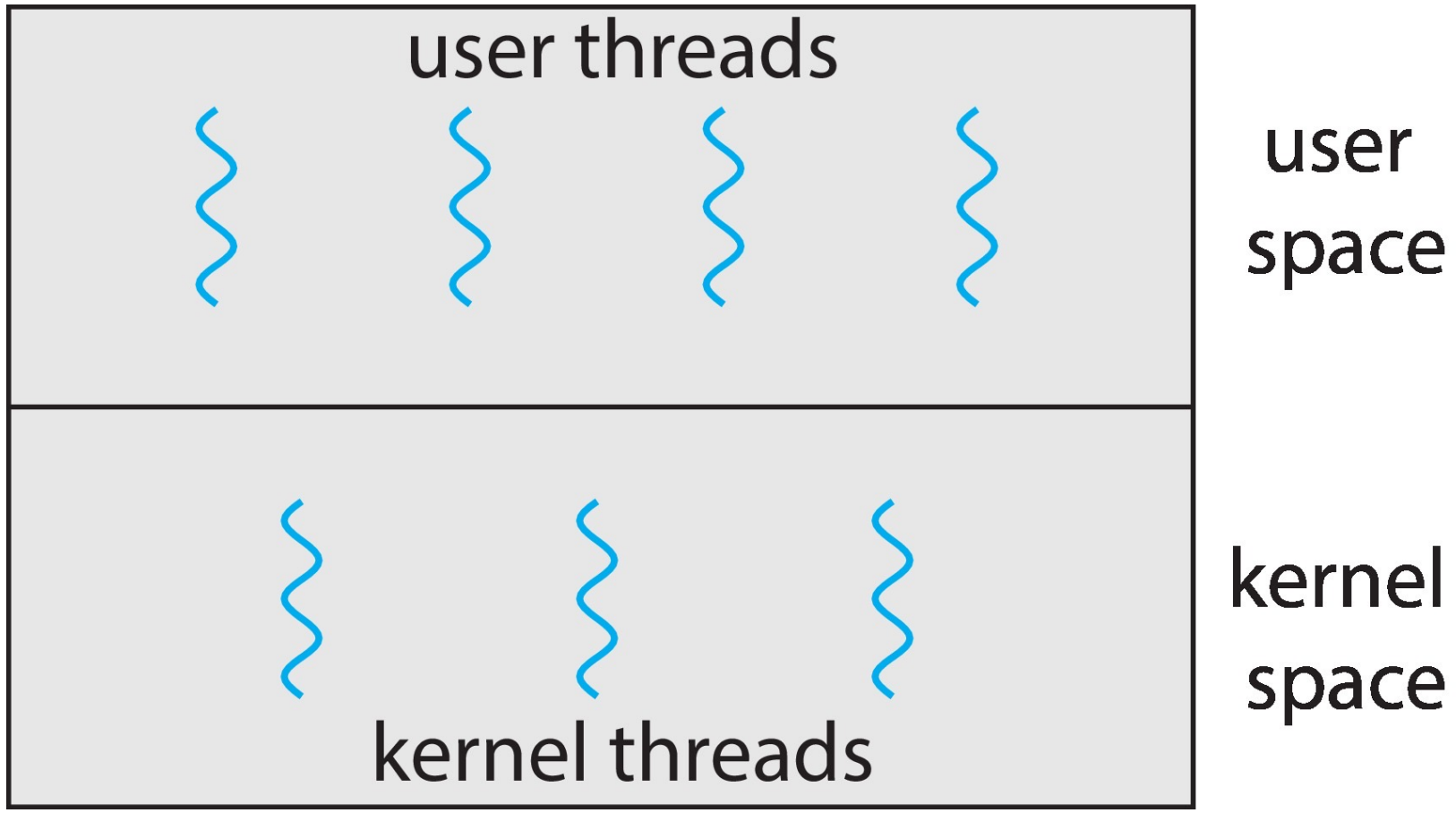
multithreaded process

A thread is

- A unit of execution
- Execution within a process
- Faster to create and destroy than a process

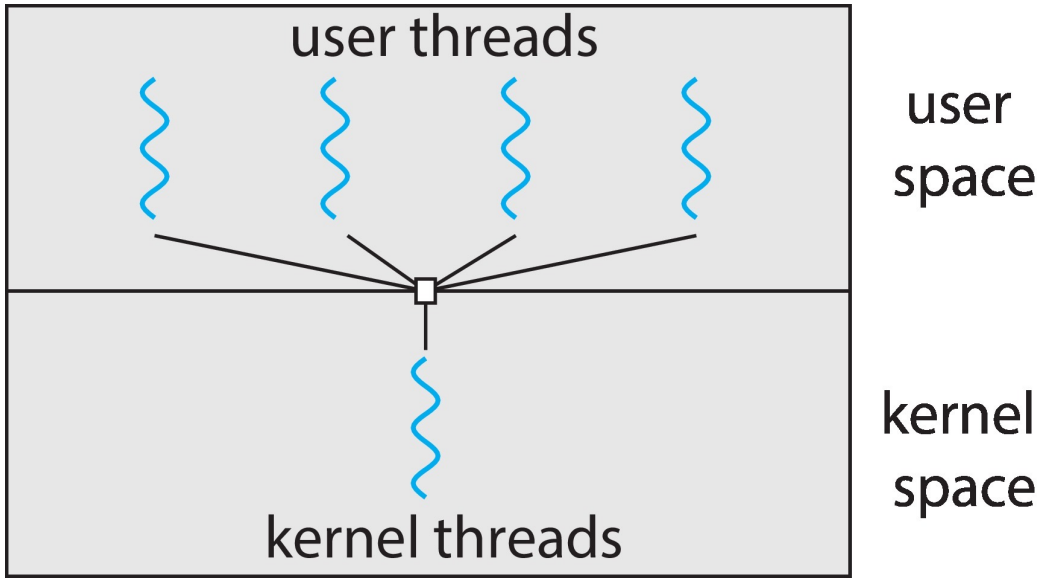
- Kernel schedules threads

User process may have user threads, managed by the process (i.e. the JVM)



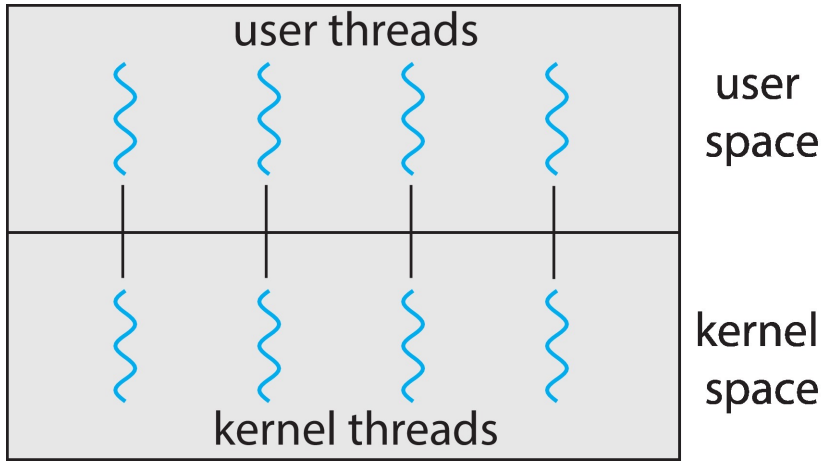
Many-to-One

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



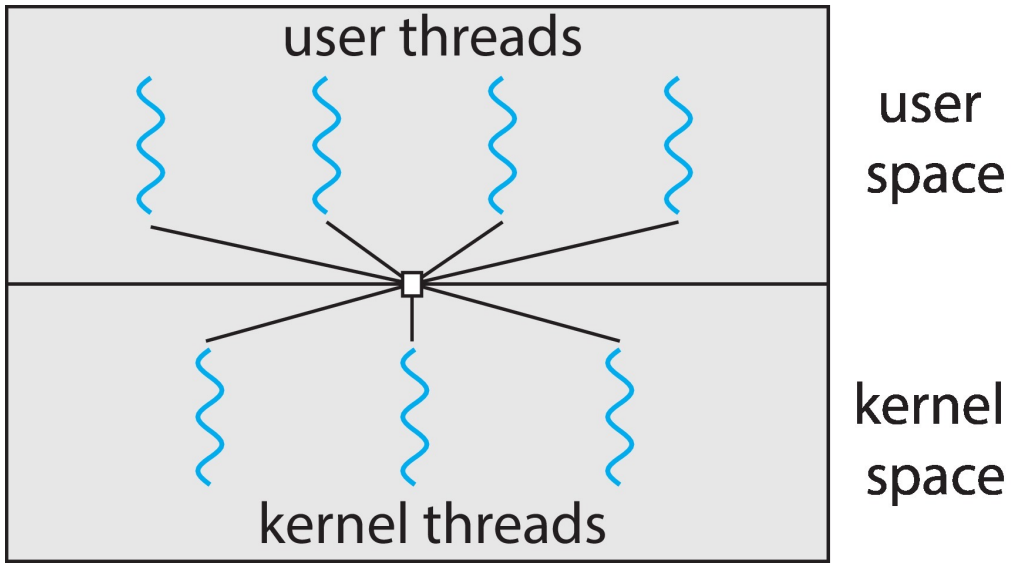
One-to-One

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



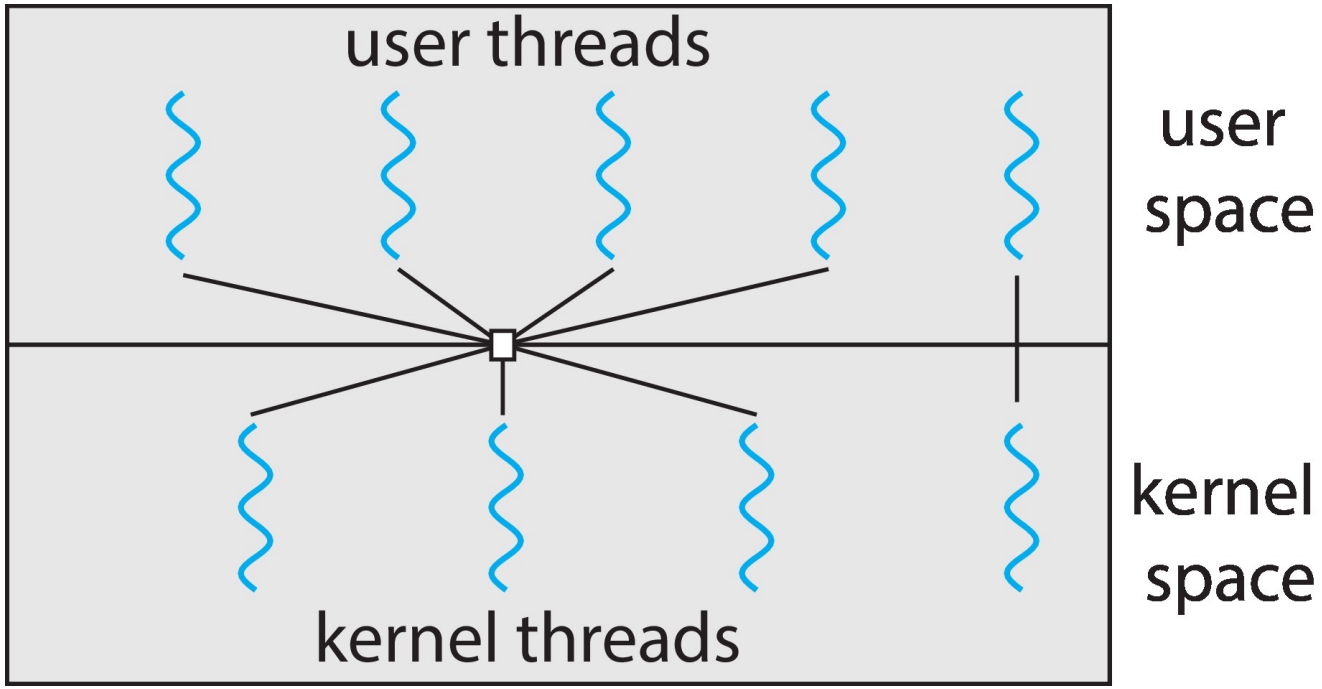
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- 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



Threads on processes

Per-process items	**Per-thread items**
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

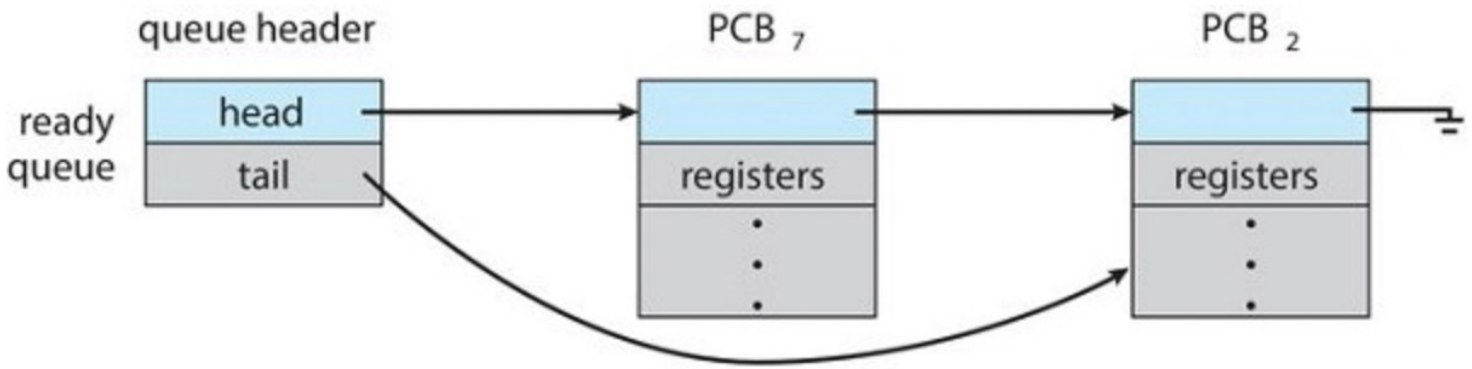
Thread programming

- In Tutorial
- In Programming assignment # 3

Scheduling

Remember: Multiprogramming *and* multitasking mean multiple *concurrent* processes

Concurrent: *def* **existing**, happening, or done at the same time.



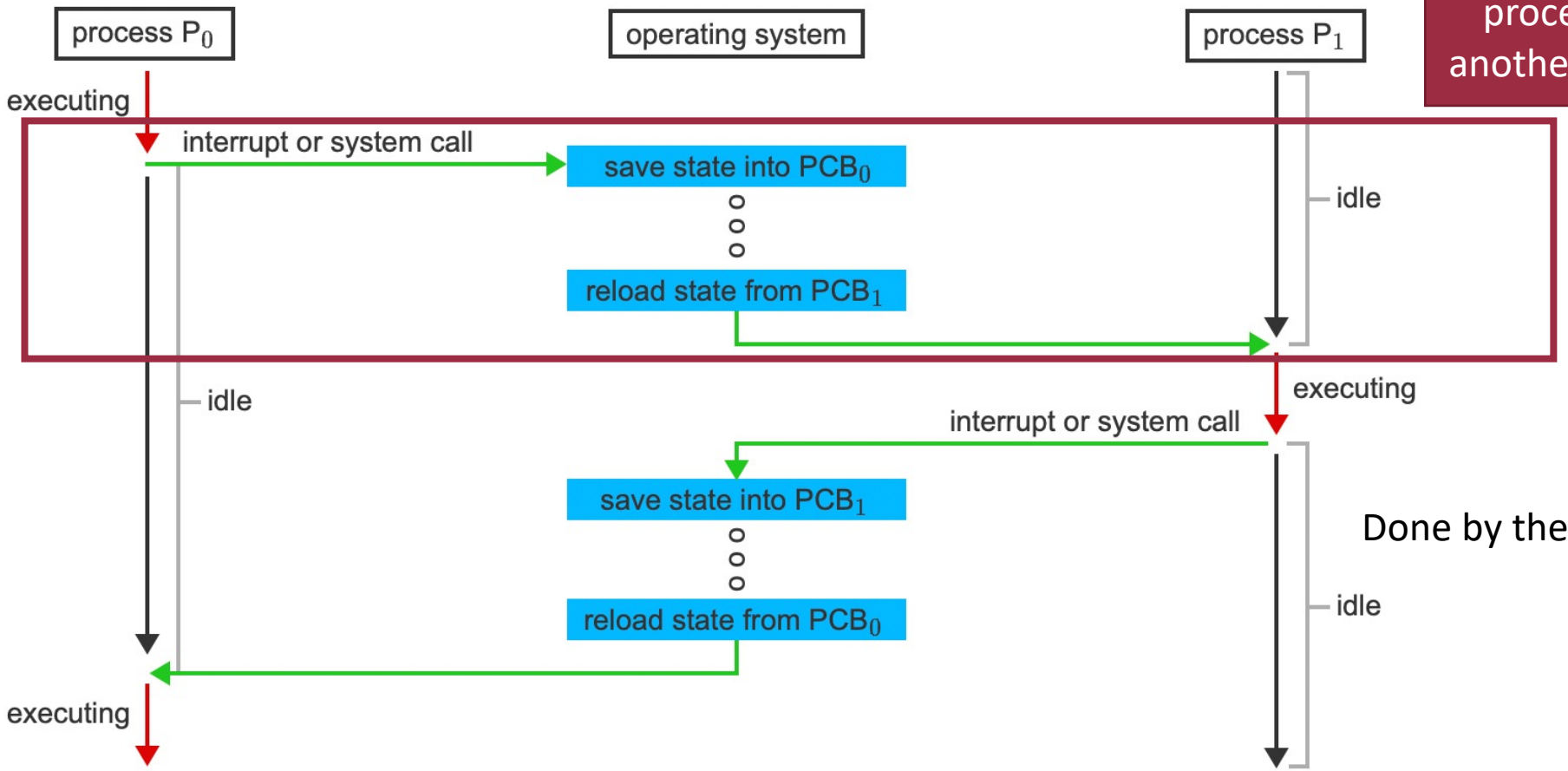
In single core systems, but you already know that

Multiple processes in the ready queue, only one of them can be selected to run.

Which process gets to run next is a decision of the scheduler, following a *policy*

Remember: When a process calls a waiting system call, another process is selected to run. This causes a context switch

How the OS switches from executing one process to executing another is a mechanism



Dispatch latency

Done by the dispatcher.

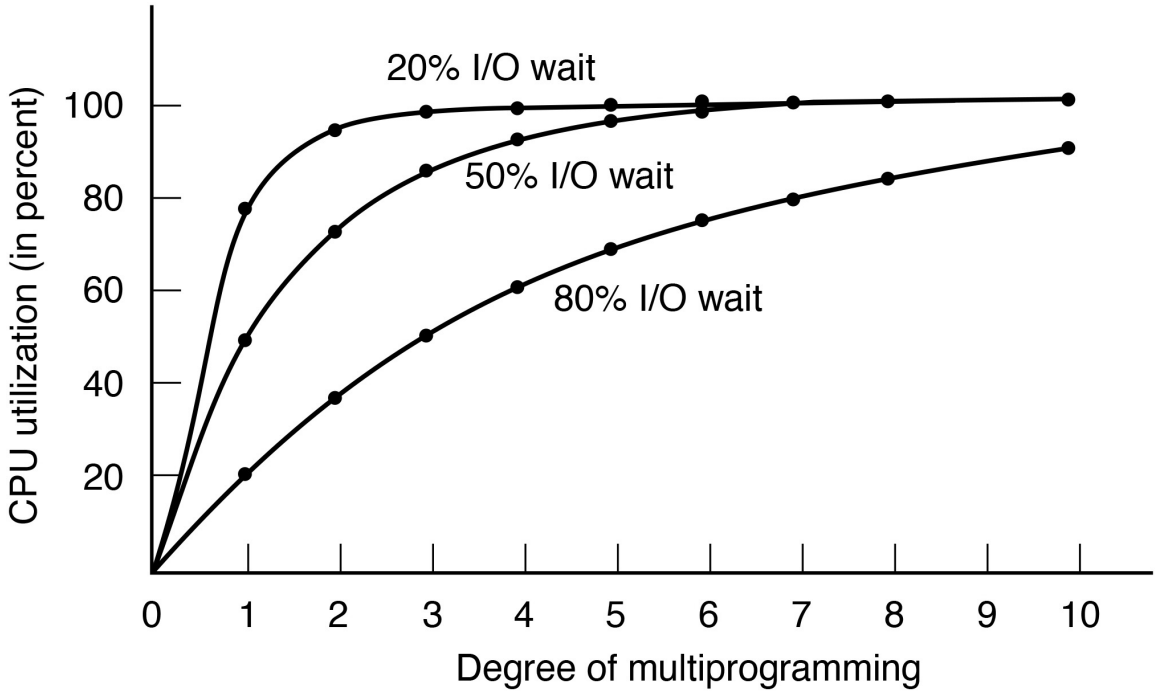
How to select the next process to run?

- The OS could choose to optimize
  - **CPU utilization:** the percentage of time the CPU is being used
  - **Throughput:** Number of processes completed / unit of time
  - **Turnaround time:** Time from process submission (or creation) to completion
  - **Waiting time:** Total time a process spends in the ready queue
  - **Response time:** Time from submission of request to response to be ready.

How to select the next process to run?

- The OS could choose to optimize
  - **CPU utilization:** the percentage of time the CPU is being used
  - **Throughput:** Number of processes completed / unit of time
  - **Turnaround time:** Time from process submission (or creation) to completion
  - **Waiting time:** Total time a process spends in the ready queue
  - **Response time:** Time from submission of request to response to be ready.

CPU utilization



Number of processes in memory at the same time

With only one process, and 20% i/o wait, the cpu is used 80% of the time

With only one process, and 50% i/o wait, the cpu is used 50% of the time

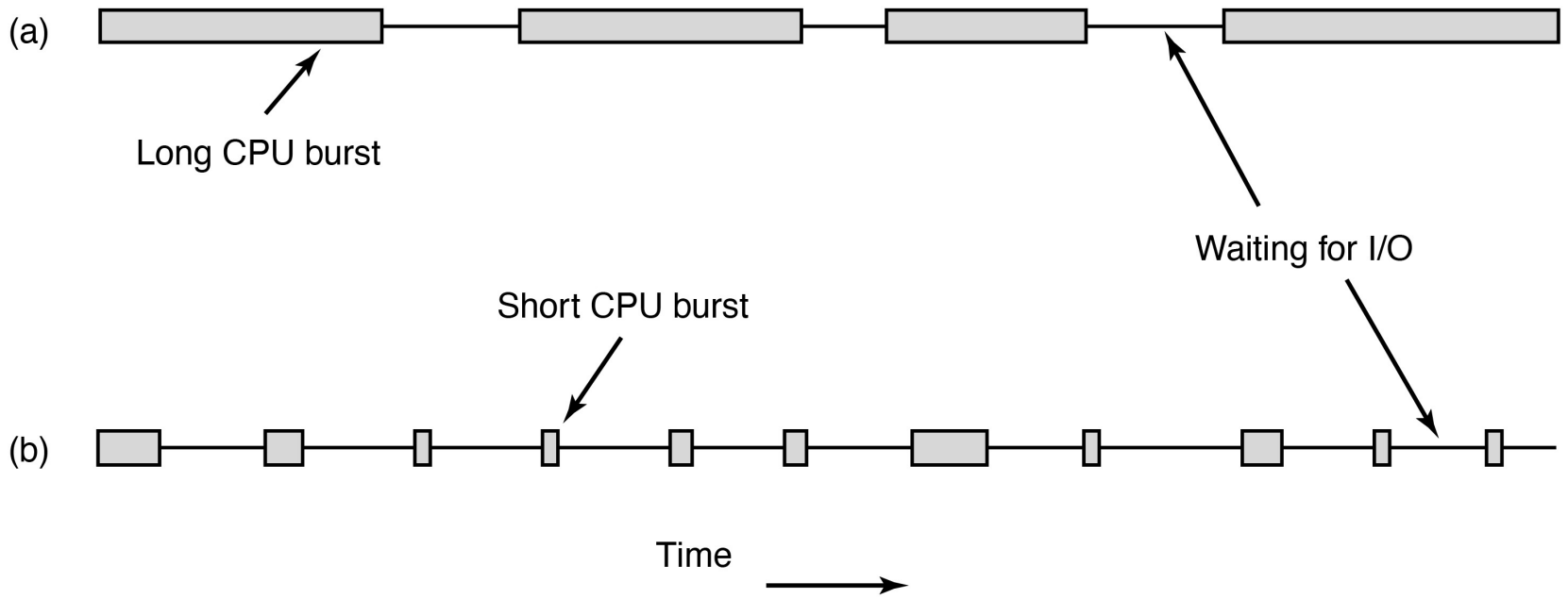
With only one process, and 80% i/o wait, the cpu is used 20% of the time

With two processes, and 50% i/o time,

Scheduling algorithms

Based on the policy followed by the OS to select next process to run

We will consider the CPU bursts, IO bursts to analyze scheduling algorithms



CPU burst is the time the process requires the CPU before requiring to wait (intentional wait)

Another consideration is if scheduling is preemptive or non-preemptive

- Non-preemptive means, the running process is changed only if it calls a wait or terminates.
  - It means the programmer knows when the process execution may be stopped
  - It also means the CPU burst is finished before context switch
- Preemptive means the current running process can be changed also if a new process becomes ready, or if the current process switches from running to ready (non-waiting system call)
  - The programmer can’t know when the process execution is stopped or resumed

First-in First out

Processes are scheduled in order of arrival. Processes that become ready go to the end of the queue

It is a non-preemptive algorithm.

Let’s analyze the waiting time and turn around time

Avg Waiting time? Avg Turn around time?

Process	Burst Time
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:

Order of arrival: waiting time, turnaround time?

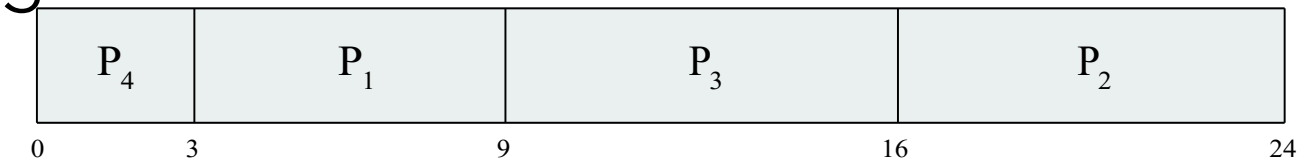
Shortest job first

- What is it optimizing?
- Non-preemptive: shortest job is scheduled, let it run its whole cpu burst
- Preemptive algorithm: if a shorter process arrives, switch to that one.

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



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

How to know the burst time of the process?

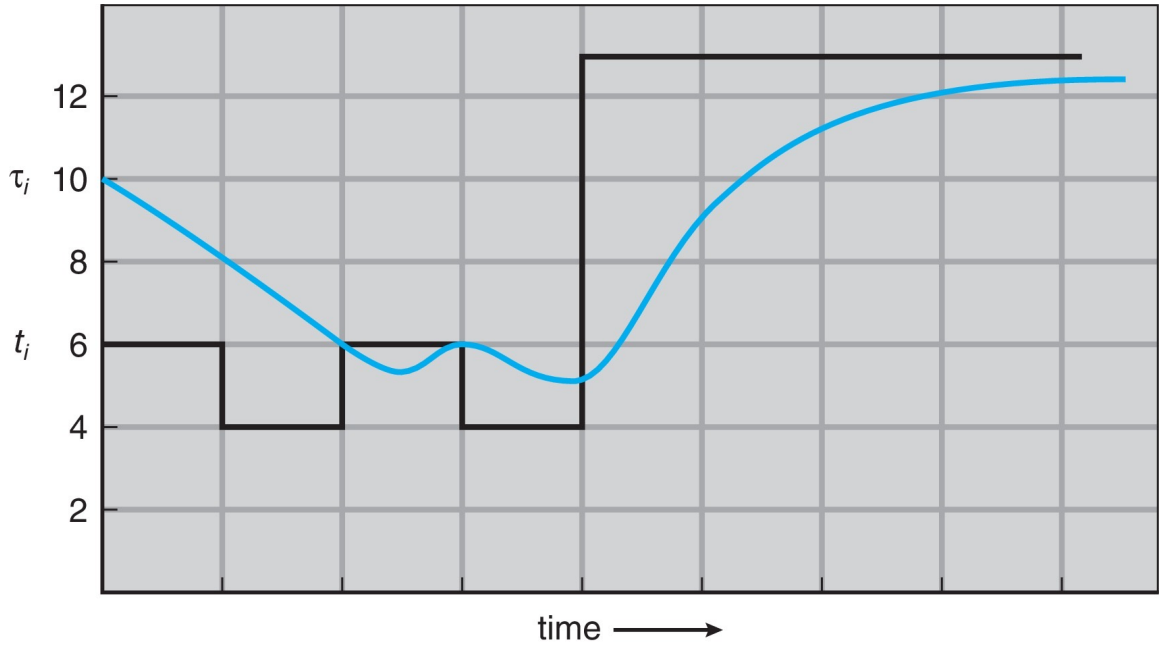
- Can't, but estimates based on previous bursts time using an exponential average
  - Arithmetic average would give the same importance to all previous bursts. Exponential average considers the next bursts will be most similar to the immediate previous

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. α, $0 \leq \alpha \leq 1$
4. Define:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$$

Commonly, α set to ½

Prediction of the Length of the Next CPU Burst



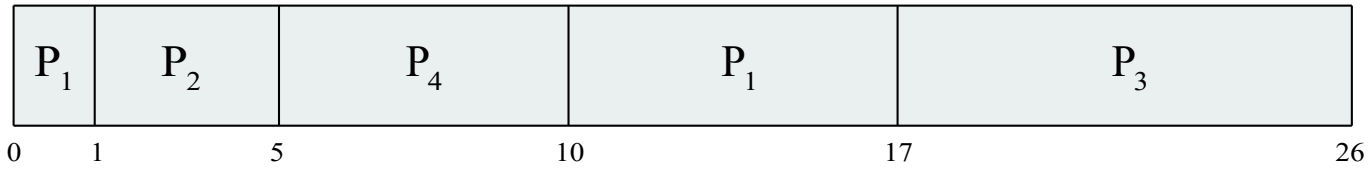
CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Example of Shortest-remaining-time-first

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

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



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

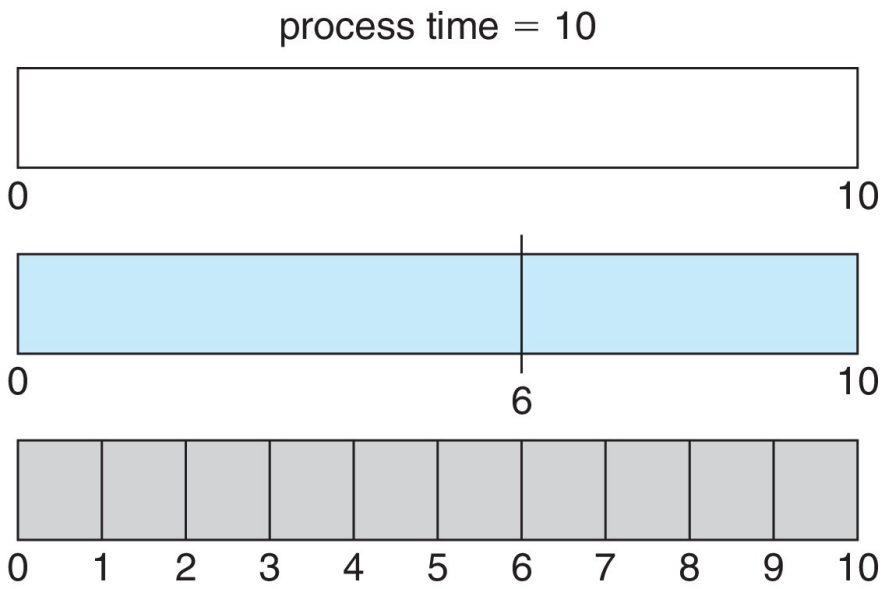
Round Robin

- First come first served with time slices (preemptive) i.e. taking turns
- Each process is given a quantum to execute, then switches to another process

Example of RR with Time Quantum = 4

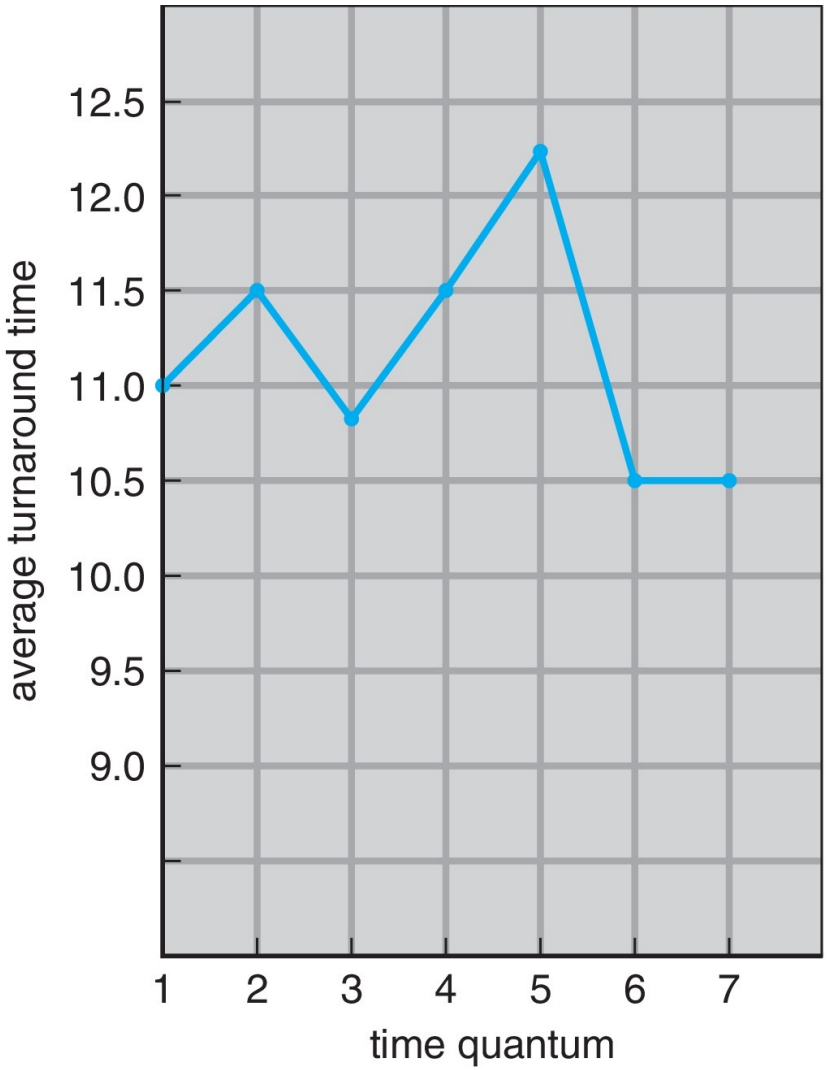
Process	Burst Time
P_1	24
P_2	3
P_3	3

Time Quantum and Context Switch Time



quantum	context switches
12	0
6	1
1	9

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than q

Priority Scheduling

- **Non-preemptive version**: choose the highest priority process in the ready queue and run its whole CPU burst.
- **Preemptive**: choose the highest priority process but only for a quantum

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Priority Scheduling w/ Round-Robin

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

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

- Gantt Chart with time quantum = 2

Processes may starve under priority scheduling

- That is a process is never run because higher priority processes are always arriving at the queue
- Aging can prevent starvation
  - Process priority is increased every X time (i.e. every second) so that it moves up in the queue.

2nd Programming assignment

- Implement scheduling algorithms and simulate running processes to compare their turnaround time and waiting time.