UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH

CSIS Department of Computer Science & Information Systems

CS4023 : Operating Systems

Sahir Sharma
University Teacher of Computer Science & Programming
CS2-035, Computer Science Building
sahir.sharma@ul.ie
Fall 2021-22

Lecture 1

Welcome!

Sahir Sharma

Module Leader for
CS4023 Operating Systems
CS4337 Big Data Management and Security

Contact Me
sahir.sharma@ul.ie
Room: CS2-035 , CSIS Building

Research Interest
Mixed Reality/AR/VR, Data Science, Machine Learning, Software Development

Topical Interest
3D Bioprinting, Digitization in Teaching, Game Economic designs, Monetary planning behind F2P games.

Previous Module Leader

Dr Patrick Healy (Paddy)

Room: CS108, CSIS Building

Tel: +353 61 202727

Email: patrick.healy@ul.ie

Administrative Details

*Meeting Times

Lecture Hours	:	Tue	16:00 - 17:00 (Live)
		Wed	12:00 - 13:00
Tutorial	:	Fri	11:00 – 12:00 (Weeks 2 onwards)
Lab	:	Exact schedule TBD (Weeks 2 onwards)	

All the lecture material will be made available on SULIS on Monday of every week

4 ⇒ 6 non-contact hours

* Subject to Change

Administrative Details

General Issues

All Lectures/Labs would take place on MS Teams

Lecture slides would be uploaded online under Module Materials > WeekXX > Resources

An updated class list would soon be uploaded on the Sulis site for CS4023

Attendance at all lectures is expected

Administrative Details

Assessment

Lab Assignments	:	6x5%=30% (Week03,04,06,08,09,10)
Mid Term Examination	:	20% (Week 06-7)
Final Examination	:	30% (Week 11-13)
Weekly Blog Participation	:	20% (10x 2%)
Bonus Assignment (Optional)	:	Upto 20% lost can be regained

Administrative Details

Assessment

Forum Participation : 16% (8 x 2%)

A maximum of 2 marks out of 100 for the whole module is allocated to each blog post.

Each post should provide your own inference towards the knowledge gathered throughout the week.

Questions towards the course content, arguments on a topic, inference after reading, bringing out additional information, debating on a topic, all of these are examples of a good blog narration.

Grading Rubrik would be made public on the course forum.

Administrative Details

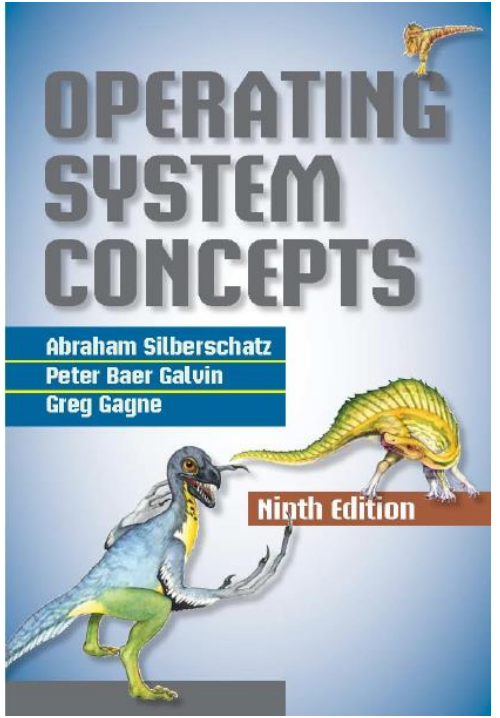
Assessment – Grade Bands

F	0 – 29
D2	30 – 34
D1	35 – 39
C3	40 – 47
C2	48 – 51
C1	52 – 55
B3	56 – 59
B2	60 – 63
B1	64 – 71
A2	72 – 79
A1	80 – 100

Administrative Details

Reading List

1. Silberschatz, Galvin & Gagne Operating System Concepts (Wiley) ISBN 0-470-12872-0 On SL in library (ed.s 6 & 8) http://www.os-book.com/
2. Silberschatz, et al. (earlier eds of above)
3. Loads of other OS books in library



To Do (for you)

Linux password resetting will be done by Liam O'Riordan (CS2-004)
Write to him on liam.oriordan@ul.ie

Don't forget to register online at http://www.si.ul.ie

Sign up for Skill Workshops (Weeks 1, 2, 3) via the CTL's Student Supports page

Maths / Erasmus / Exchange students: email your Name, ID, program of study to me, sahir.sharma@ul.ie in order to get a Linux account

Registration for NASA Space Apps Challenge Limerick is open:

https://2021.spaceappschallenge.org/locations/limerick/event

CS4023...

CS4023 (like school) (like life) is hard

"The only difference between success and failure is the ability to take action."
- Alexander Graham Bell

"Work hard, have fun, make history."
- Jeff Bezos

"Success isn't always about greatness. It's about consistency. Consistent hard work leads to success. Greatness will come."
- Dwayne "The Rock" Johnson

Syllabus – Rationale and Purpose

On successful completion of this module a student should have a clear understanding of the

- Logical structure of, and facilities provided by, a modern OS
- Concepts of processes, threads and multithreading and how they are implemented in a modern OS
- Problems that arise when processes collaborate and compete and well as being able to demonstrate practical experience of mechanisms for handling these situation
- Different ways of implementing virtual memory
- Use of system calls

Syllabus – Overview

- Overview of Operating-System Structure and Operations
- Process Management
- Memory Management
- Storage Management
- Protection and Security

Detailed Syllabus (1/3)

- Positioning the operating system (OS) between the user and the hardware; the need for the OS; different types of OSs; interfaces to an OS and the interface with the hardware
- The concept of a process and a thread; representation of processes and threads; process and thread state; process creation and termination; thread creation, scheduling and termination; multithreading
- Scheduling; context switching; concurrency, including interaction between threads
- Inter process communication (IPC); synchronization and mutual exclusion problems; software algorithms for IPC; 2 processes, n processes

Detailed Syllabus – continued... (2/3)

- Low- and high-level mechanisms for IPC and synchronization: signals; spinlocks; semaphores, message passing and monitors; deadlock; use of semaphores for synchronization, mutual exclusion, resource allocation; implementation of semaphores; use of event counts and sequencers for classical IPC problems; conditional critical regions; monitors and condition variables
- Physical and virtual memory; address translation; base and length registers; segmentation and paging; cache memory; system services for memory management
- I/O subsystem, directory name space; inodes; synchronous and asynchronous I/O; locking; buffering

Detailed Syllabus – continued… (3/3)

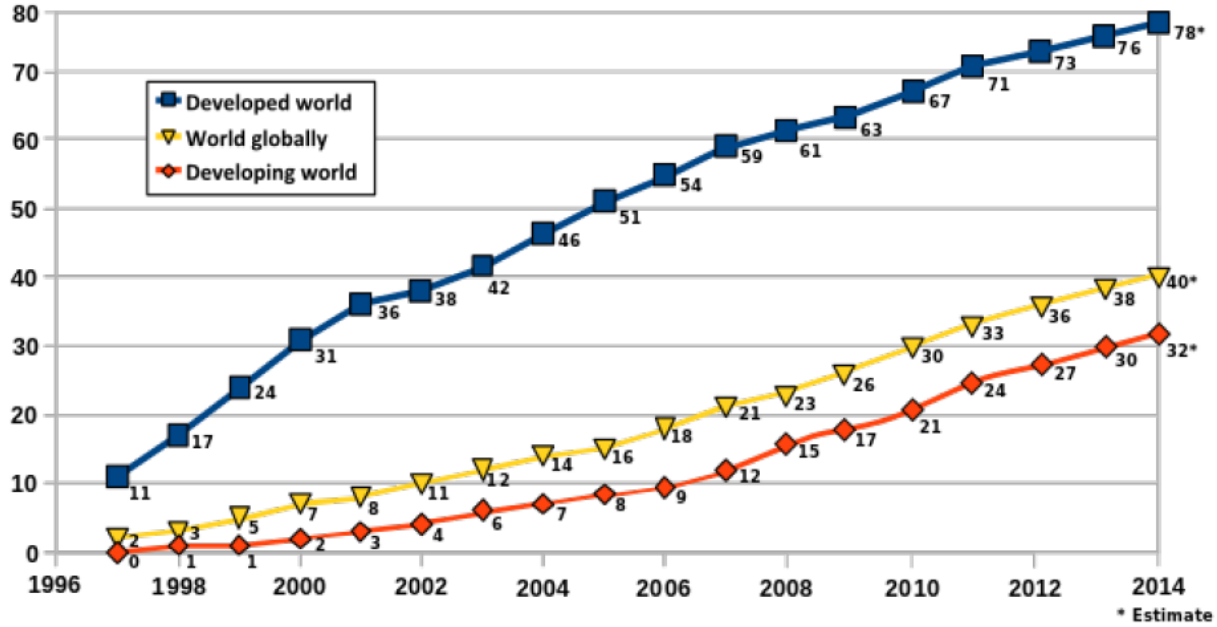
- File systems and file management; file system types; disk organization; mounting a file system; device drivers; file system-based IPC; pipes; the socket mechanism; IPC using sockets
- Fault tolerance and security

Learning Outcomes and Assessment Methods

Learning Outcomes	Assessment Method
Explain the objectives and functions of modern operating systems	Final/Mid Term
Describe the logical structure of, and facilities provided by, a modern operating system	Weekly Lab
Analyze the tradeoffs inherent in operating system design	Weekly Blogs
Differentiate between the concepts of processes, threads and multithreading	Weekly Blogs
Demonstrate practical experience of mechanisms for handling situations of process collaboration and competition	Weekly Blogs
Identify the problems that arise when processes collaborate and compete	Weekly Blogs
Categorize different ways of implementing virtual memory	Weekly Blogs
Discuss networked, client-server, distributed operating systems and how they differ from single user operating systems	Weekly Blogs
Summarize the use of system calls	Weekly Blogs

Why Operating Systems? – Internet Users

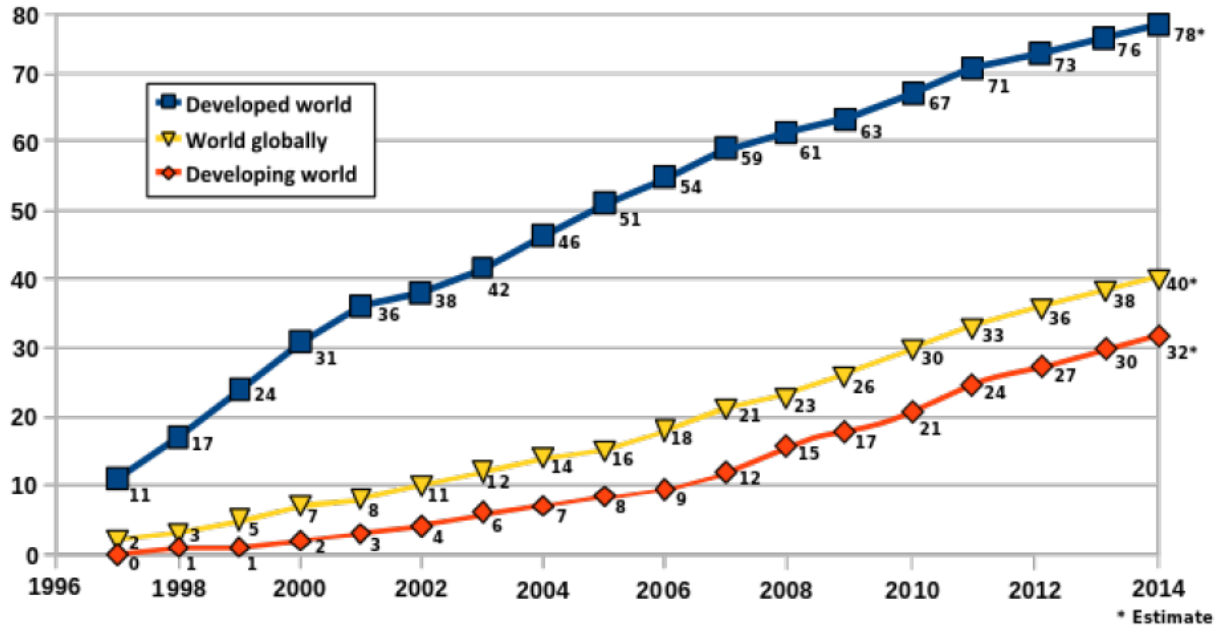
Number of internet users per 100 inhabitants :



More recent data here; somebody better be managing how this volume of people use the Internet

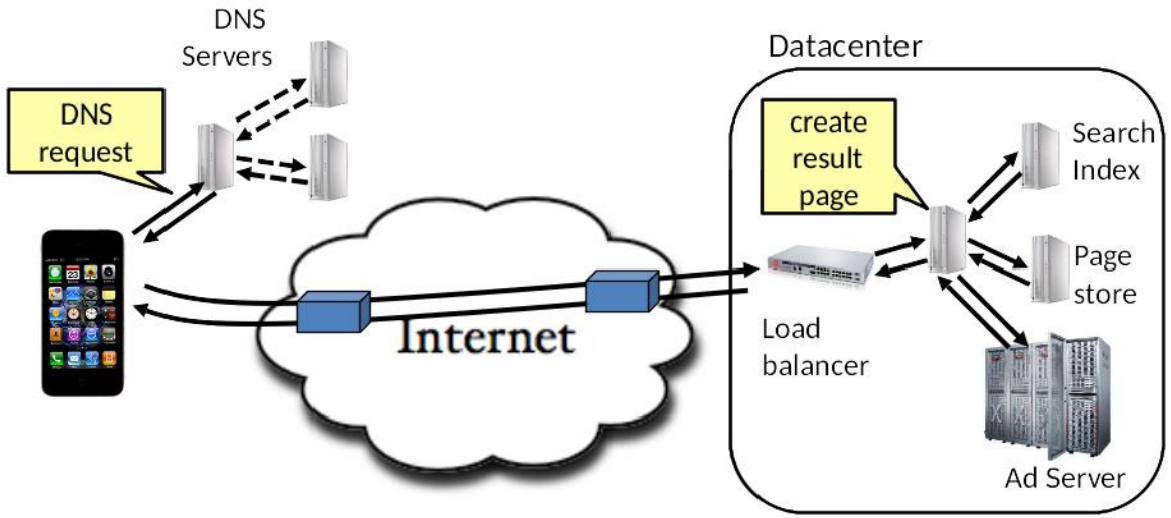
Why Operating Systems? – Speed of Computation

Moore's Law over the past 120 years:



Somebody better be managing how this computing power is harnessed

Why Operating Systems? – Internet Search Query



Somebody better be managing these communications steps

Why Operating Systems? – Software Complexity

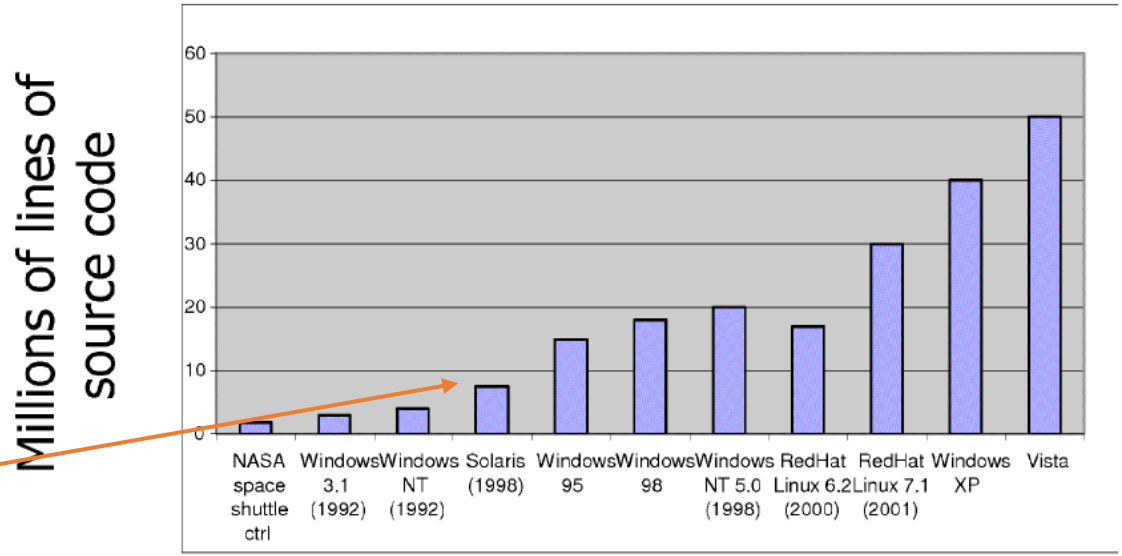
Lines of Code

- 1975 Unix kernel: 10,500 lines of code
- 2008 Linux 2.6.24 line counts:
  - 85,000 processes
  - 430,000 sound drivers
  - 490,000 network protocols
  - 710,000 file systems
  - 1,000,000 different CPU architectures
  - 4,000,000 drivers
  - 7,800,000 Total

Why Operating Systems? – Software Complexity

Lines of Code

- 1975 Unix kernel: 10,500 lines of code
- 2008 Linux 2.6.24 line counts:
  - 85,000 processes
  - 430,000 sound drivers
  - 490,000 network protocols
  - 710,000 file systems
  - 1,000,000 different CPU architectures
  - 4,000,000 drivers
  - 7,800,000 Total



Why Operating Systems? – Software Complexity

Taming this complexity

- Every piece of computer hardware different
  - Different CPU
    - Pentium, PowerPC, ColdFire, ARM, MIPS
  - Different amounts of memories.
    - HDD, SSD, RAM
  - Different types of devices
    - Mice, Keyboards, Sensors, Cameras, Fingerprint readers
  - Different networking environment
    - Cable, DSL, Wireless, VPN, Firewalls
- Issues:
  - Does the programmer need to write a single program that covers all operations of computer systems? Modularity?
  - Does every program have to be altered for every piece of hardware? Re-usability?
  - Does a faulty program crash everything? Fault isolation?
  - Does every program have access to all hardware? Security?


UNIVERSITY OF
LIMERICK
OLLSCOIL LUIMNIGH


CSIS
Department of
Computer Science
& Information Systems

Thank You

UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH

CSIS Department of Computer Science & Information Systems

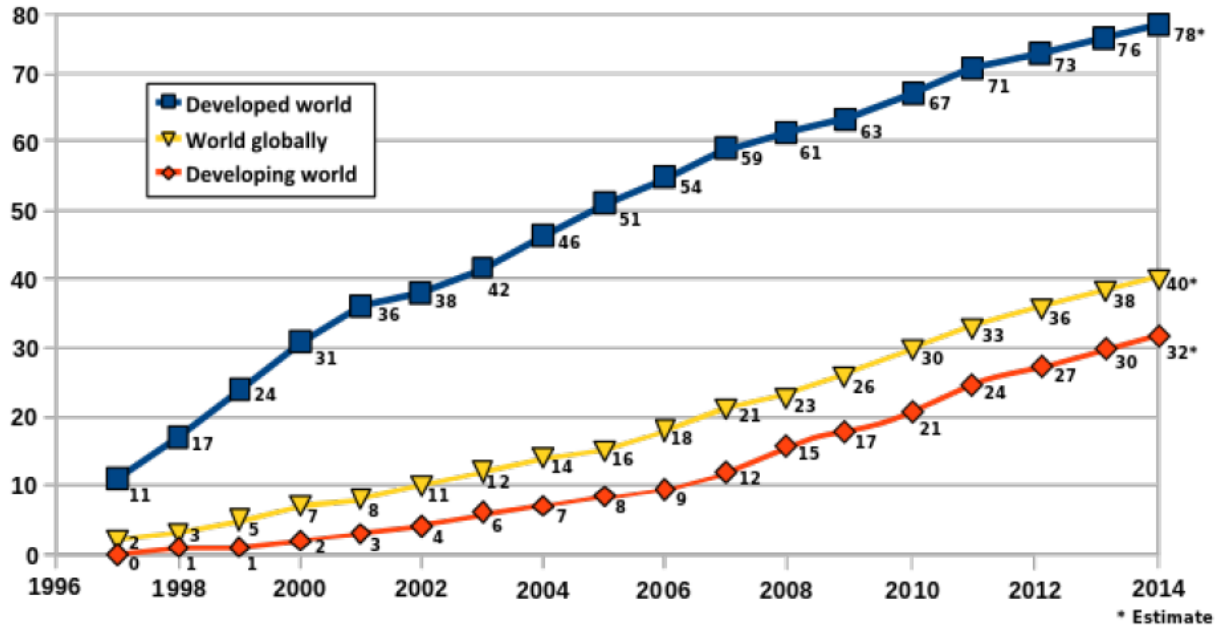
CS4023 : Operating Systems

Sahir Sharma
University Teacher of Computer Science & Programming
CS2-035, Computer Science Building, University of Limerick
sahir.sharma@ul.ie
Autumn 2021-22

Week2, Lecture 1

Why Operating Systems? – Internet Users

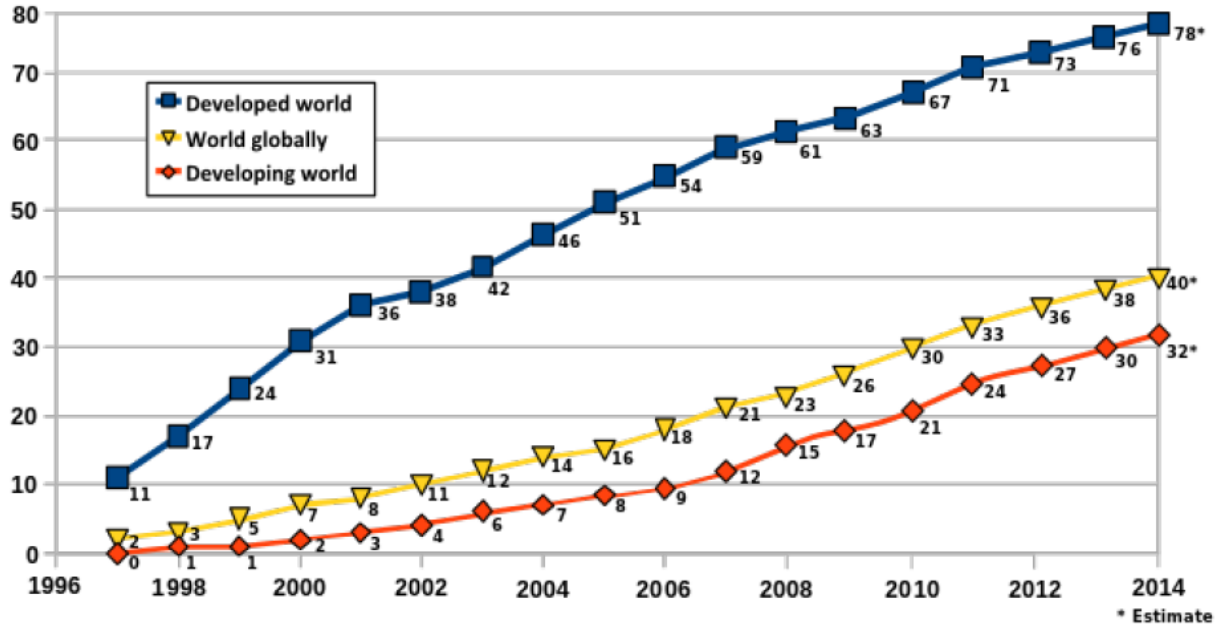
Number of internet users per 100 inhabitants :



More recent data here; somebody better be managing how this volume of people use the Internet

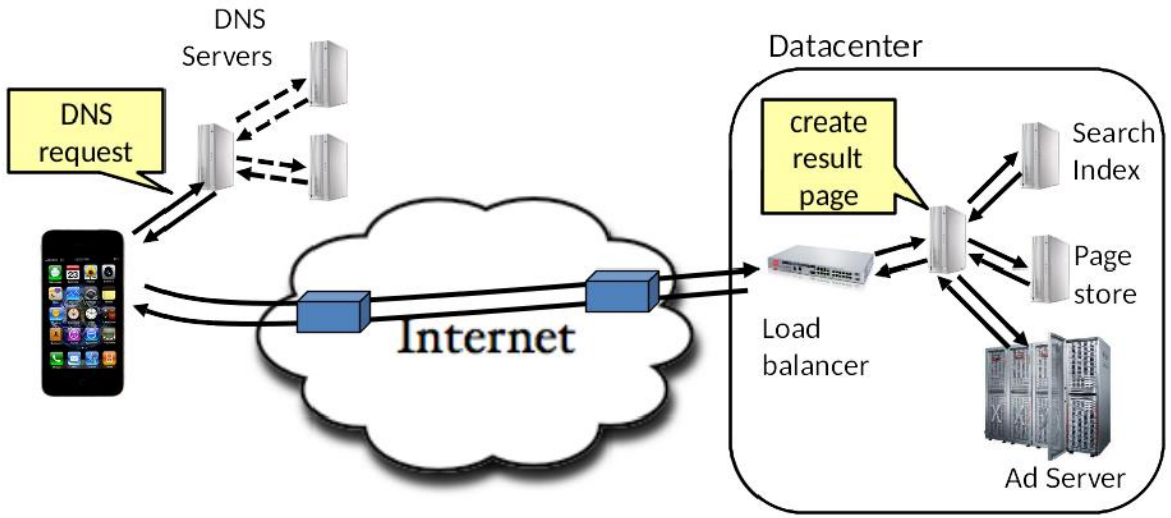
Why Operating Systems? – Speed of Computation

Moore's Law over the past 120 years:



Somebody better be managing how this computing power is harnessed

Why Operating Systems? – Internet Search Query

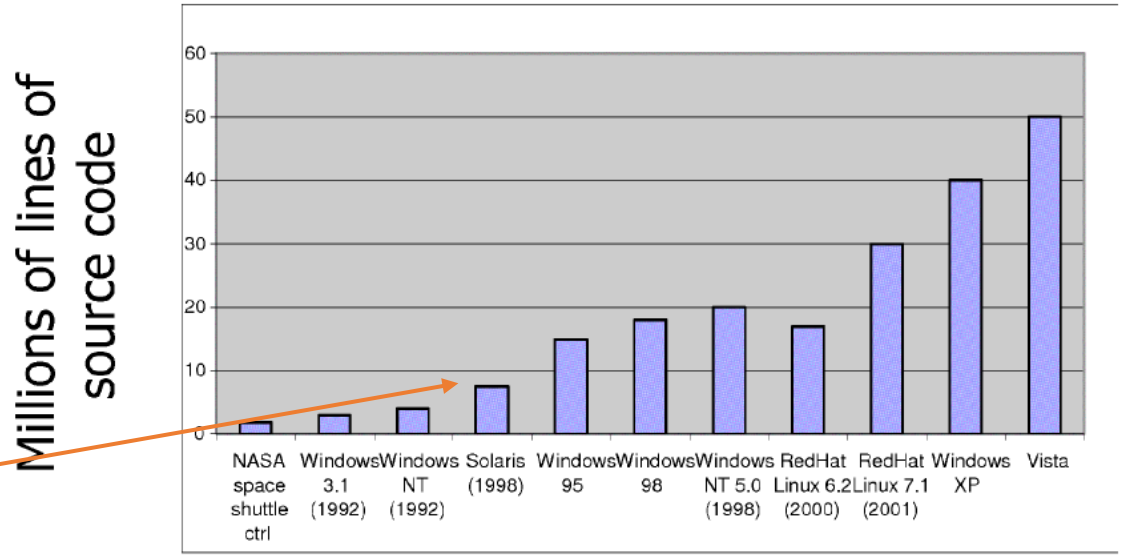


Somebody better be managing these communications steps

Why Operating Systems? – Software Complexity

Lines of Code

- 1975 Unix kernel: 10,500 lines of code
- 2008 Linux 2.6.24 line counts:
  - 85,000 processes
  - 430,000 sound drivers
  - 490,000 network protocols
  - 710,000 file systems
  - 1,000,000 different CPU architectures
  - 4,000,000 drivers
  - 7,800,000 Total



Why Operating Systems? – Software Complexity

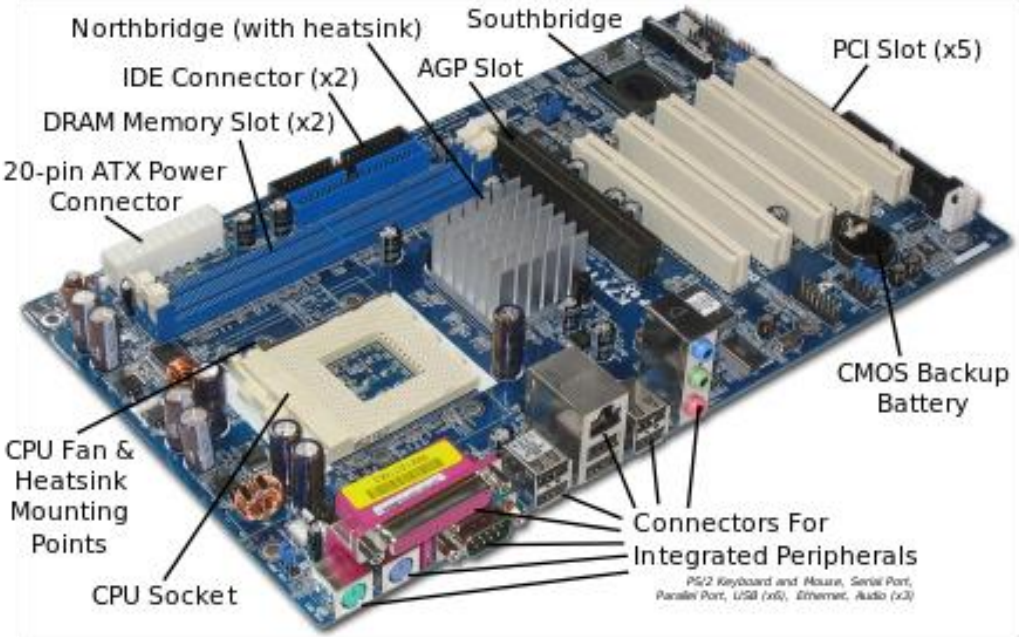
Taming this complexity

- Every piece of computer hardware different
  - Different CPU
    - Pentium, PowerPC, ColdFire, ARM, MIPS
  - Different amounts of memories.
    - HDD, SSD, RAM
  - Different types of devices
    - Mice, Keyboards, Sensors, Cameras, Fingerprint readers
  - Different networking environment
    - Cable, DSL, Wireless, VPN, Firewalls
- Issues:
  - Does the programmer need to write a single program that covers all operations of computer systems? Modularity?
  - Does every program have to be altered for every piece of hardware? Re-usability?
  - Does a faulty program crash everything? Fault isolation?
  - Does every program have access to all hardware? Security?

Before understanding Operating Systems

Let us first understand Computer Systems

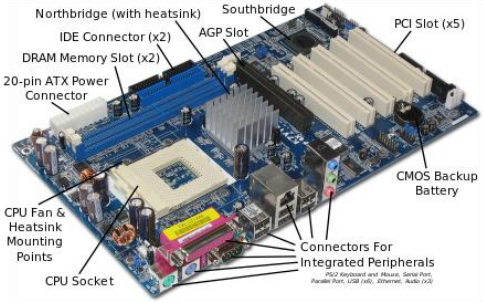
- Computer System Organization
- Computer System Architecture

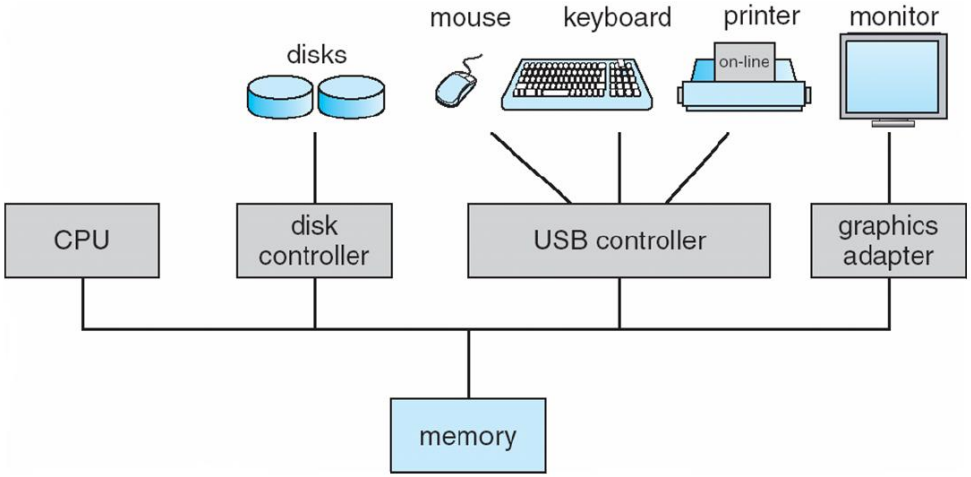


Before understanding Operating Systems

Computer System Operation

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





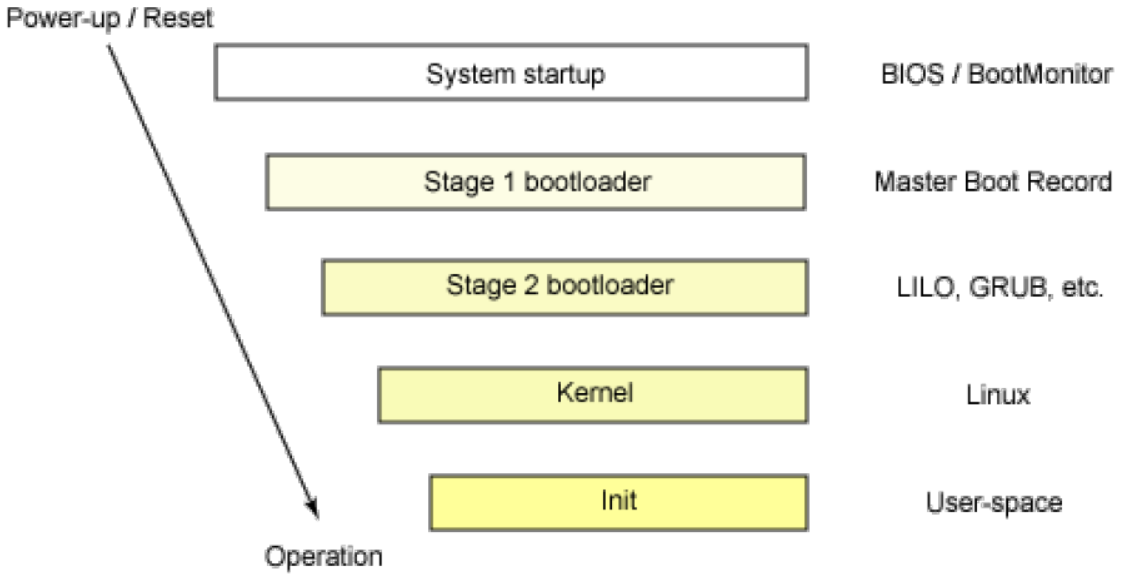
Computer System Organization

Computer Startup

- Bootstrap/Bootloader program is loaded at power-up or reboot
  - Typically stored in ROM or EEPROM, generally known as firmware.
  - Initializes all aspects of system.
  - Loads operating system kernel and starts execution.
- The OS then starts executing the first process, such as init
  - init is a system program or daemon; i.e., it provides services on behalf of OS but is outside kernel
    - init loads other daemons
    - example of a daemon is the "listener" program that reacts to external network requests e.g., ftp
- OS then waits for some event to occur

Computer System Organization

- Computer Startup : Linux



Computer System Organization

Computer System Operation

- I/O devices and the CPU can execute concurrently
- Each device controller (hardware) oversees a particular device type
- Each device controller has a local buffer (memory)
- CPU moves data from main memory to local buffers
- I/O is from local buffer of controller to the device
- Device controller informs CPU that it has finished its operation by causing an interrupt

Computer System Organization

Computer System Operation

- I/O devices and the CPU can execute concurrently
- Each device controller (hardware) oversees a particular device type
- Each device controller has a local buffer (memory)
- CPU moves data to main memory from local buffers
- I/O is to local buffer of controller from the device
- Device controller informs CPU that it has finished its operation by causing an interrupt

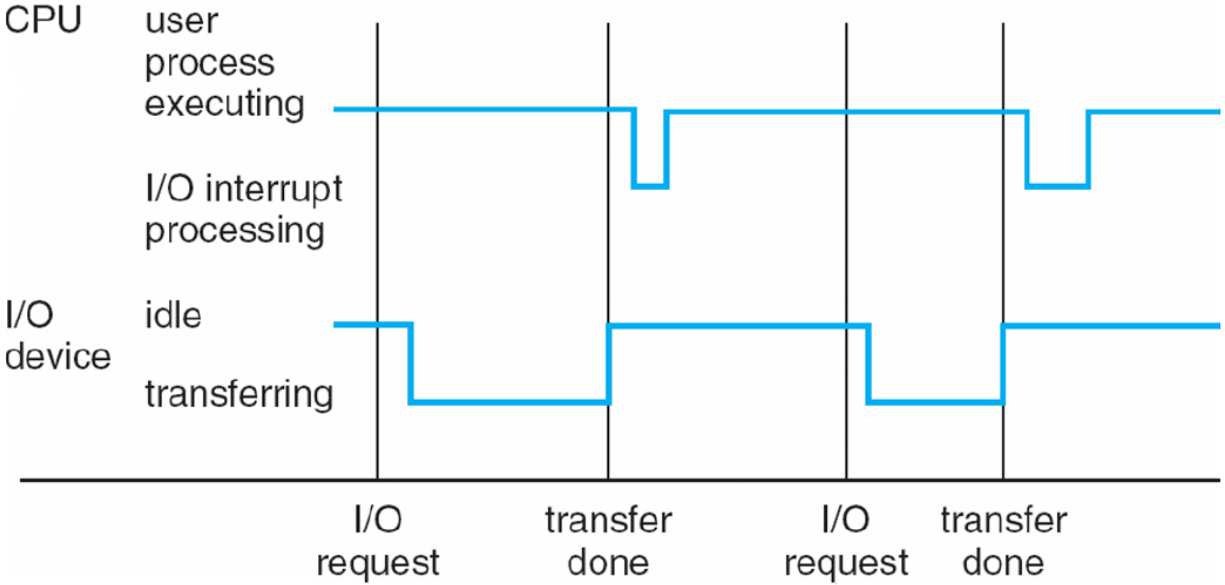
Computer System Organization

I/O Structure

- A **device controller** (hardware) maintains some local buffer storage and a set of special purpose registers
- OS has a **device driver** (software) for each device controller
- Start of an I/O operation:
  - The device driver loads the appropriate registers within the device controller
  - The device controller examines the contents of these registers to determine what action to take
  - The device controller starts the transfer of data from/to the local buffer to the to/from the device
  - Once the transfer is complete, the device controller informs the device driver via an interrupt
  - The device driver then returns control to the OS, possibly returning (if the operation was a 'read') the data or a pointer to the data

Computer System Organization

- Interrupt Timeline

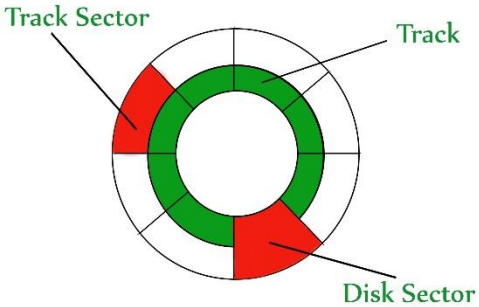


Computer System Organization

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 one per byte (as previously done)
- On current Linux systems blocksize is 4096 bytes

Computer System Organization



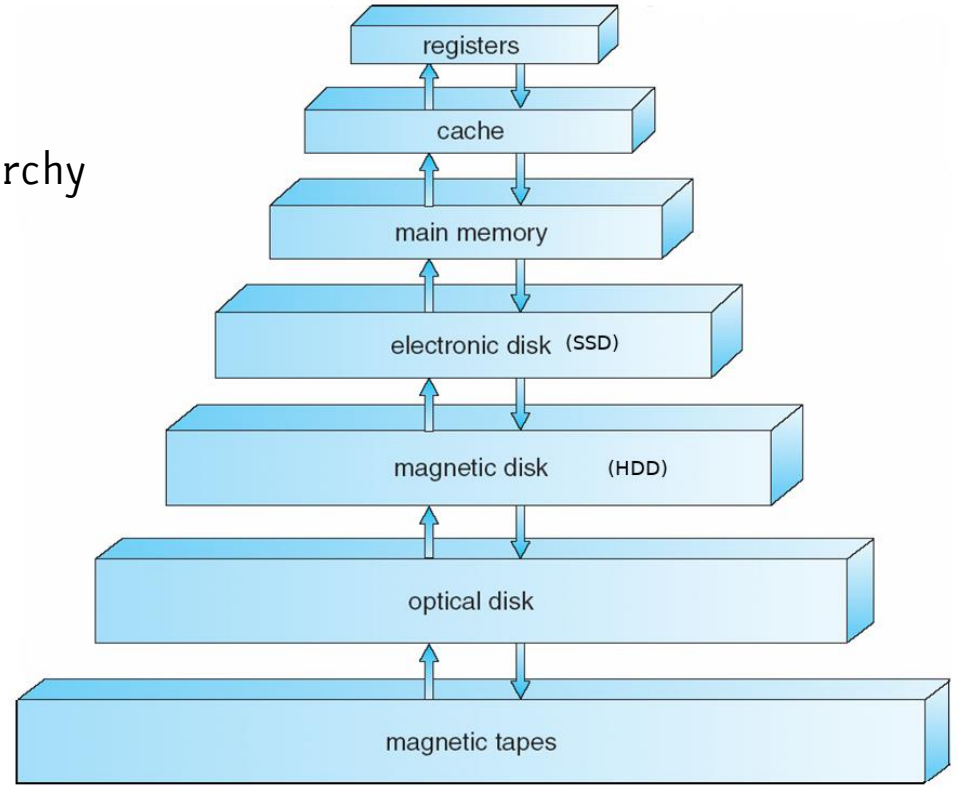
Storage Structure

- Main memory – only large storage media that the CPU can access directly (via data bus)
  - Cache doesn't count as it is small
- Secondary storage – extension of main memory that provides large nonvolatile storage capacity
- Magnetic disks – rigid metal or glass platters covered with magnetic recording material
  - Disk surface is logically divided into **tracks**, which are subdivided into sectors
  - The **disk controller** determines the logical interaction between the device and the computer
  - Beginning to become obsolete for laptops, lightweight devices; replaced by **solid-state disks.**

Computer System Organization

Storage Hierarchy

- Storage systems organized in hierarchy
  - Speed
  - Cost
  - Volatility



Computer System Organization

Large-Scale Storage Devices

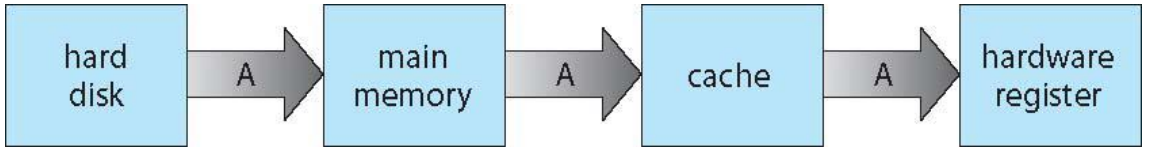
- Main memory: only large storage media that the CPU can access directly
  - Random access (RAM)
  - Typically, volatile
- Secondary storage: extension of main memory that provides large nonvolatile storage capacity. The main two types of these are:
  - HDD (Hard-disk Drive): rigid metal or glass platters covered with magnetic recording material
    - Disk surface is logically divided into tracks, which are subdivided into sectors
    - The disk controller determines the logical interaction between the device and the computer
  - SSD (Solid-state Drive): faster than hard disks, nonvolatile
    - Various technologies e.g., a mixture of RAM and magnetic disk
    - Popular in cameras, PDAs, USB etc.; now in larger devices

Read Up: https://uk.pcmag.com/ssd/8061/ssd-vs-hdd-whats-the-difference

Computer System Organization

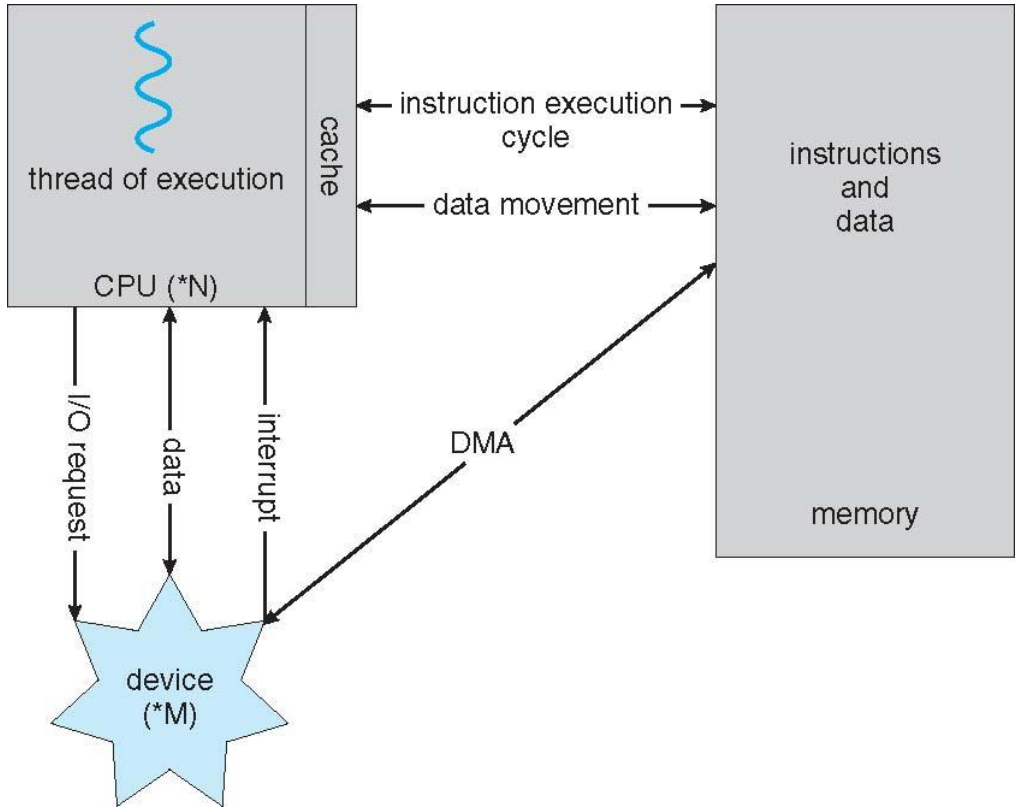
Caching

- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use (i.e., it will likely be used soon again) copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there
- Cache smaller than storage being cached
  - Cache management important design problem
  - Cache size and replacement policy



Computer System Organization

- How a modern computer works



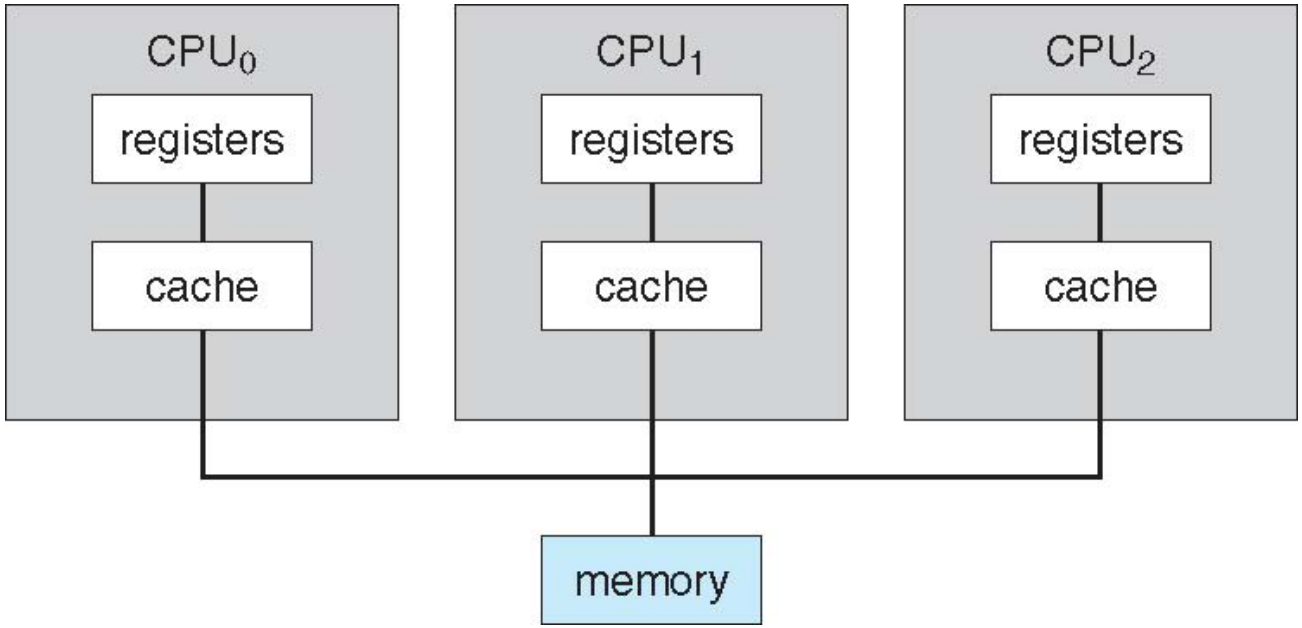
Computer System Architecture

System Types

- **Single Processor Systems**: Traditionally, systems use a single general-purpose processor (from PDAs all the way to mainframes) – though that is changing
  - Most systems have special-purpose processors as well
- **Multiprocessor systems** growing in use and importance
  - Also known as parallel systems, tightly-coupled systems
  - Advantages include
    - Increased throughput
    - Economy of scale
    - Increased reliability: graceful degradation or fault tolerance
  - Two types
    - Asymmetric Multiprocessing (master-slave relationship)
    - Symmetric Multiprocessing

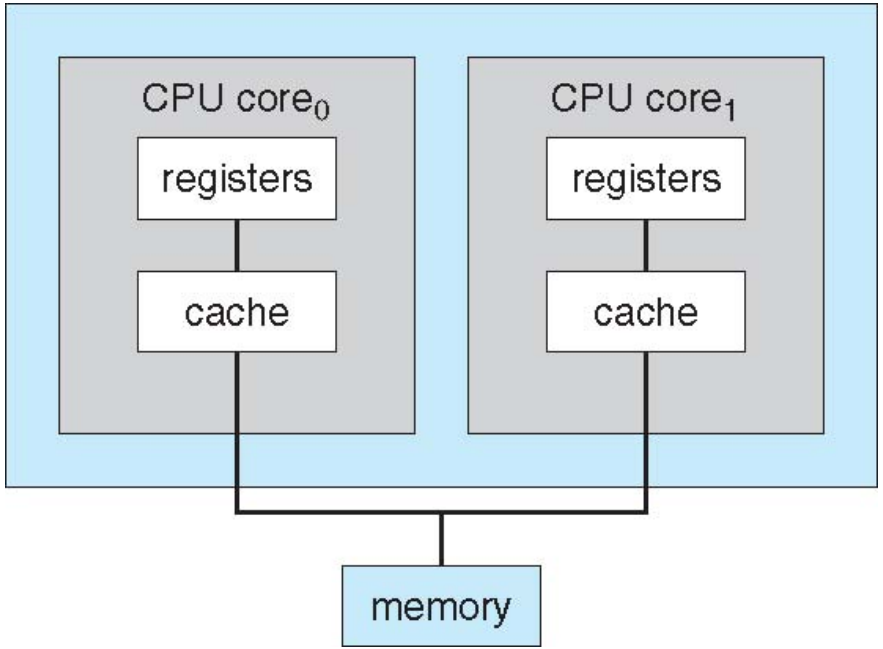
Computer System Architecture

- Symmetric Multiprocessing Architecture

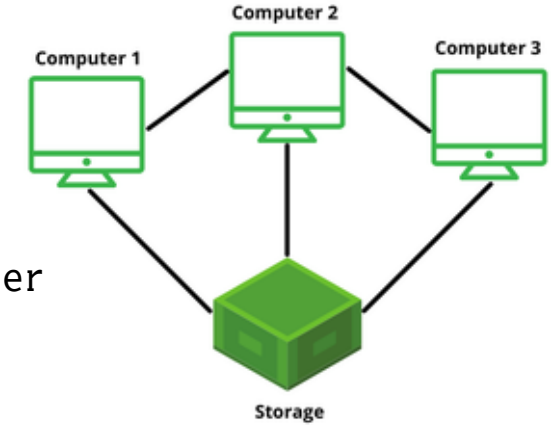


Computer System Architecture

- A Dual-Core Design



Computer System Architecture



Clustered Systems

Like multiprocessor systems, but multiple systems working together

- Usually sharing storage via a storage-area network (SAN)
- Provides a high-availability service which survives failures
  - Asymmetric clustering has one machine in hot-standby mode
  - Symmetric clustering has multiple nodes running applications, monitoring each other
- Some clusters are for high-performance computing (HPC)
  - Applications must be written to use parallelization


UNIVERSITY OF
LIMERICK
OLLSCOIL LUIMNIGH


CSIS
Department of
Computer Science
& Information Systems

Thank You

UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH

CSIS Department of Computer Science & Information Systems

CS4023 : Operating Systems

Sahir Sharma
University Teacher of Computer Science & Programming
CS2-035, Computer Science Building, University of Limerick
sahir.sharma@ul.ie
Autumn 2021-22

Week 03, Lecture 1

What are we going to do?

Outline

- Kernel Types
- Simple OS Examples
- System Calls
  - System Call Overview
  - Types of System Calls

Announcements

- I hope everyone has setup their Linux Machines
  - VPN Access + Linux Acess : Needed for Week03 lab
- strace: program to watch system calls
- LevUL up : Student Digital Skills Development
  https://www.ul.ie/ctl/students/levul-student-digital-skills-development

Pre-Req

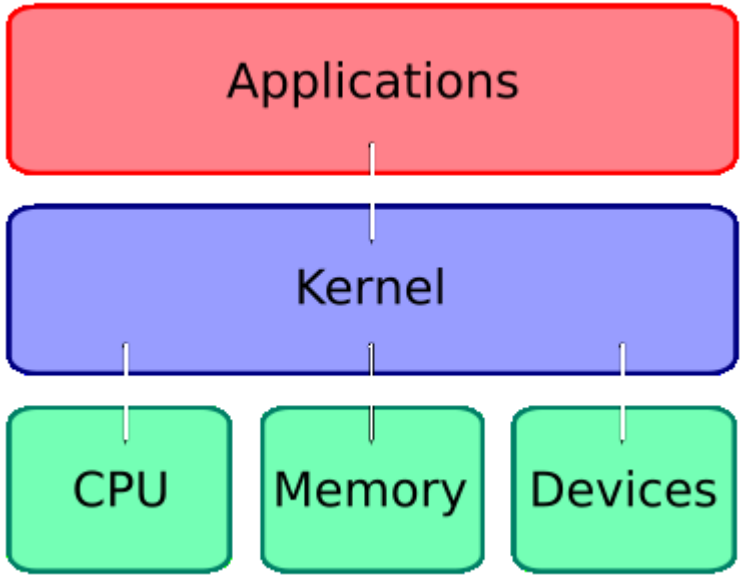
Basic C Programming

- Please carefully go through the concepts in C language here https://www.cprogramming.com/tutorial/c-tutorial.html

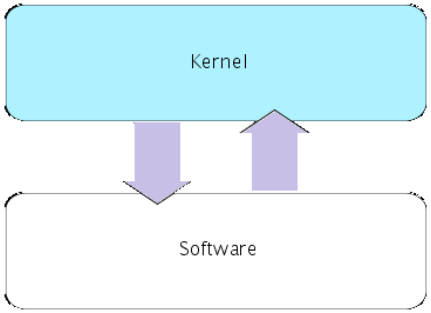
Kernel

The OS Kernel

- The kernel is the central component of an OS
- It has complete control over everything that occurs in the system
- Kernel overview and comparison



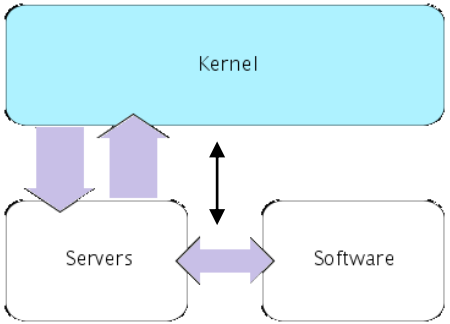
Kernel Types



Monolithic Kernel

- Older monolithic kernels were written as a mixture of everything the OS needed, without much of a structure. The monolithic kernel offers everything the OS needs: CPU scheduling , memory management, multiprogramming, interprocess communication (IPC), device access , file systems, network protocols, etc.
- Newer monolithic kernels have a modular design, which offers run-time adding and removal of services. The whole kernel runs in "kernel mode", a processor mode in which the software has full control over the machine. The processes running on top of the kernel run in "user mode", in which programs have only access to the kernel services
- Modern OS with monolithic kernels: Linux, FreeBSD, NetBSD, Solaris

Kernel Types



Microkernel

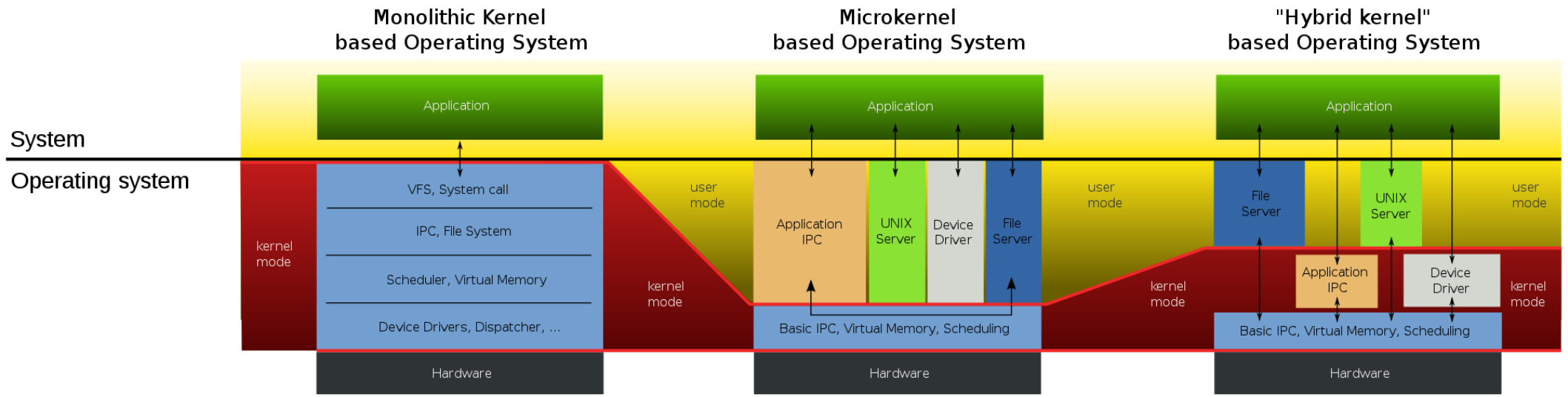
- Moves as much from the kernel into "user" space to run as processes
- Communication takes place between user modules using message passing
- Benefits
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Disadvantages
  - Performance overhead of user space to kernel space communication
- Example microkernels: L4, Minix, QNX

Kernel Types

Hybrid Kernels

- Like microkernels, except that they include additional code in kernel space so that such code can run more swiftly than it would were it in user space
- These kernels represent a compromise that was implemented by some developers before it was demonstrated that pure microkernels can provide high performance
- Hybrid kernels should not be confused with monolithic kernels that can load modules after booting (such as Linux)
- Example OS with hybrid kernels: Microsoft Windows NT, 2000 and XP, Windows 7, DragonFly BSD

Kernel Comparisons

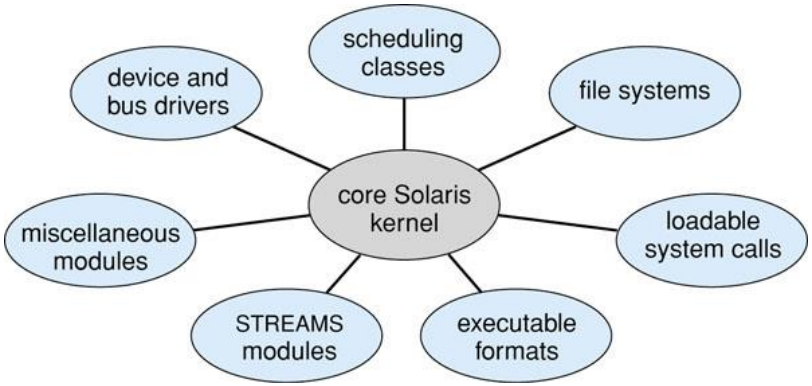


Kernel Types

Modules

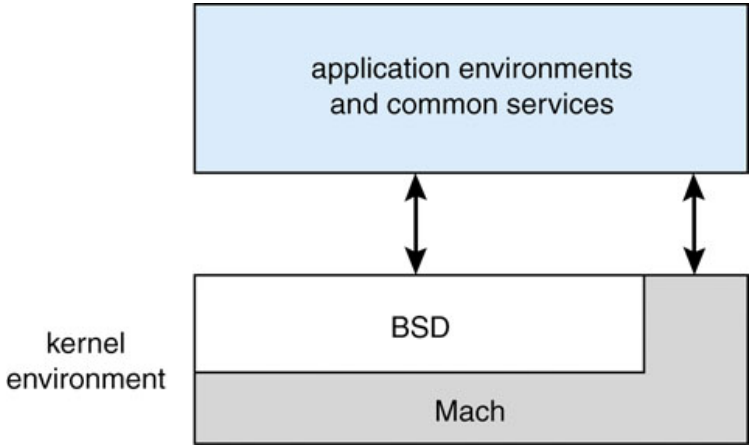
- Best current methodology for OS design involves using OOP techniques to create a modular kernel
- Kernel has set of core components and links in additional services as required – either at boot time or run time
- Uses dynamically loadable modules

Solaris and 7 types of loadable modules



Kernel Types

Mac OS X Structure

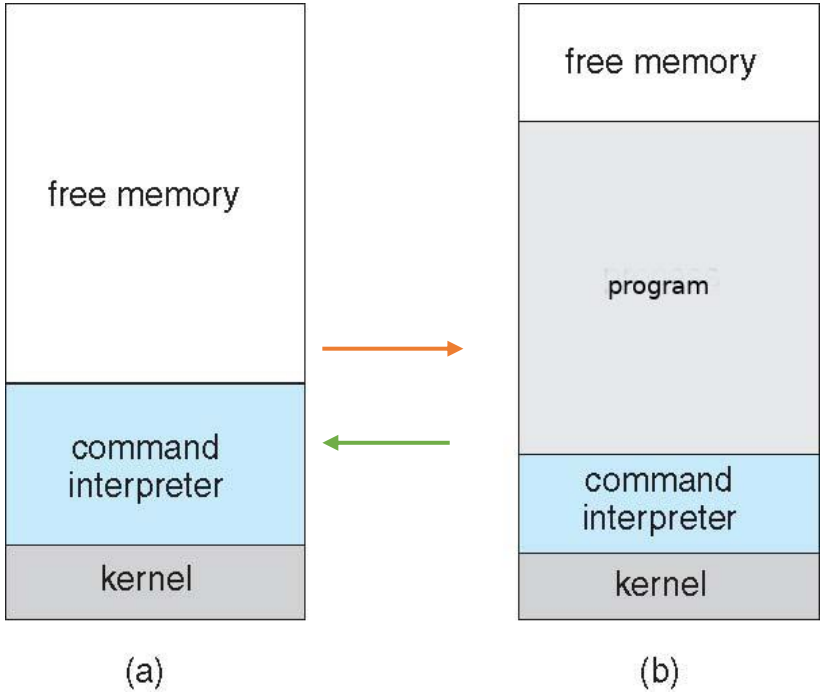


- Apple's Mac OS X uses a hybrid structure
- Top layers include application environments and services providing a graphical interface to applications
- Below these layers is kernel environment, which consists mainly of
  - Mach microkernel: provides memory management, thread scheduling and support for RPCs and other types of IPC
  - BSD kernel: file system support, networking

Simple OS Example

MS-DOS

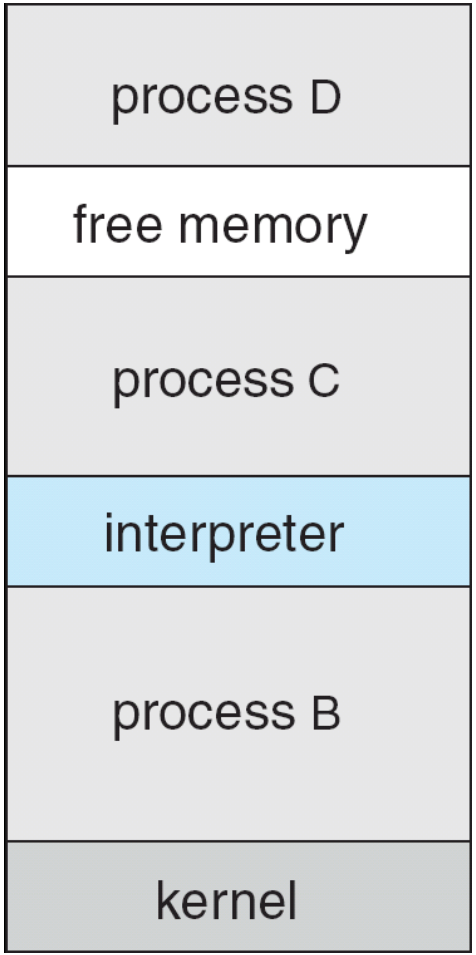
- Single-tasking
- Shell invoked when system booted
- Simple method to run program: No separate process created
- Single memory space; fig. (a)
- Loads program into memory, overwriting all but the kernel; fig. (b)
- On program exit shell is reloaded; back to (a)



Simple OS Example

FreeBSD

- Unix variant
- Multitasking
- On user login, invoke user's choice of shell
- Shell executes fork() system call to create process
  - Executes exec() to load program into process
  - Shell waits for process to terminate or continues with user commands
- Process exits with:
  - code = 0: no error
  - code > 0: error code



System Calls

Overview

- System call: a **request** made by any arbitrary program to the OS for performing tasks which the said program does not have required permissions to execute
- System calls are generally available as assembly language instructions, and they are usually listed in the various manuals used by the assembly-language programmers
  - Mostly accessed by programs via a **high-level Application Program Interface** (**API**) rather than direct system call use
  - Managed by **run-time support library** (set of functions built into libraries included with a compiler)

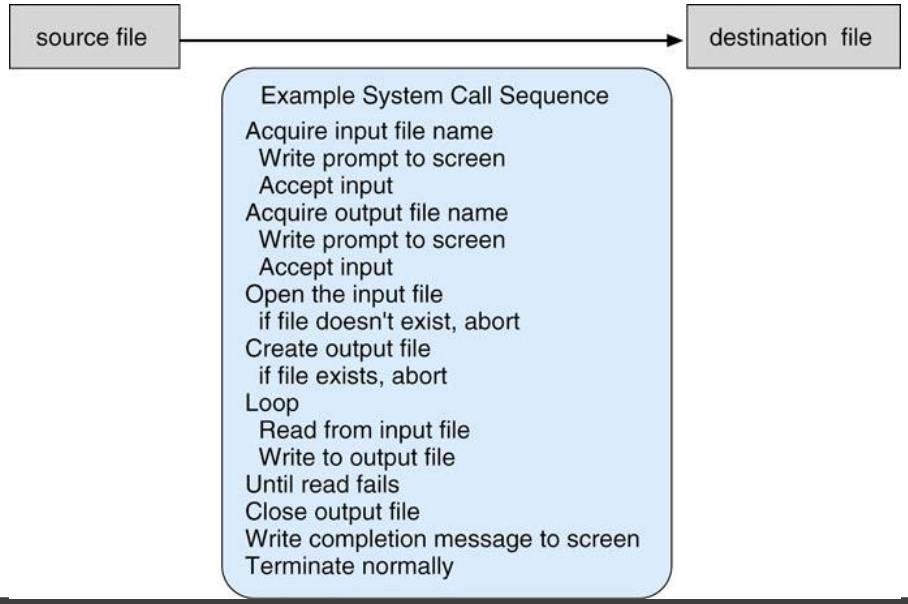
System Calls

- Three most common APIs are:
  - Windows API
  - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
  - Java API for the Java virtual machine (JVM)

System Calls

Example Use

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



System Calls

Linux open() Example

- The Linux open() function – a function (which uses a system call) for opening a file prior to reading / writing

```
OPEN(2)                    Linux Programmer's Manual                    OPEN(2)

NAME
       open, creat - open and possibly create a file or device

SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <fcntl.h>

       int open(const char *pathname, int flags);
       int open(const char *pathname, int flags, mode_t mode);
       int creat(const char *pathname, mode_t mode);

DESCRIPTION
       Given a pathname for a file, open() returns a file descriptor, a small,
       non-negative integer for use in subsequent system calls (read(2),
       write(2), lseek(2), fcntl(2), etc.). The file descriptor returned by a
       successful call will be the lowest-numbered file descriptor not cur-
       rently open for the process.
```

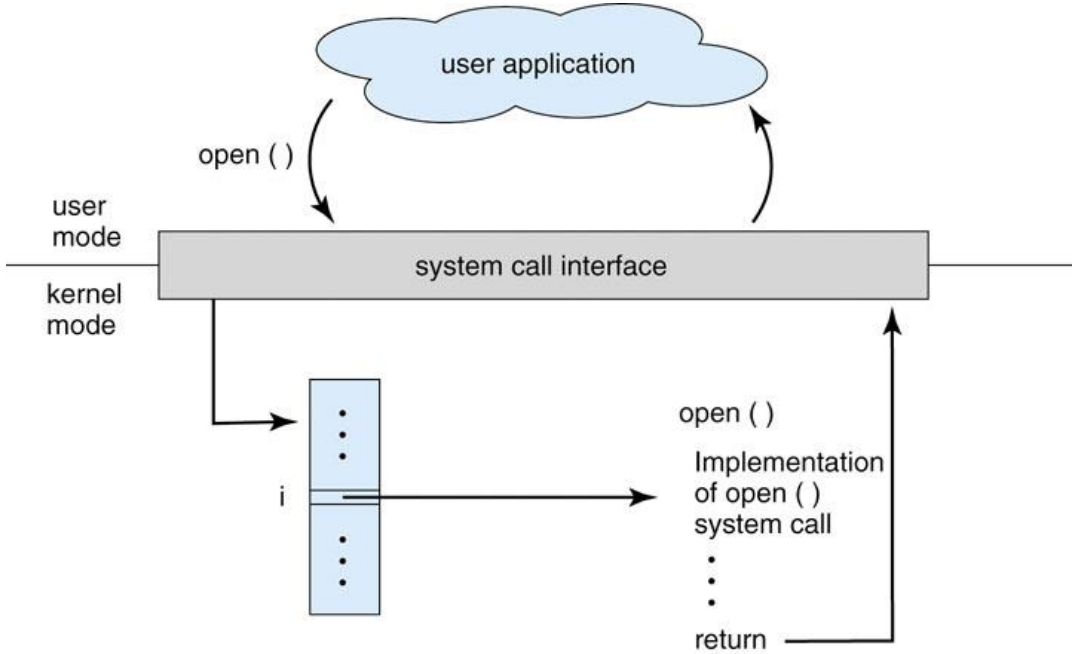
System Calls

Implementing System Calls

- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller (our program) 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)

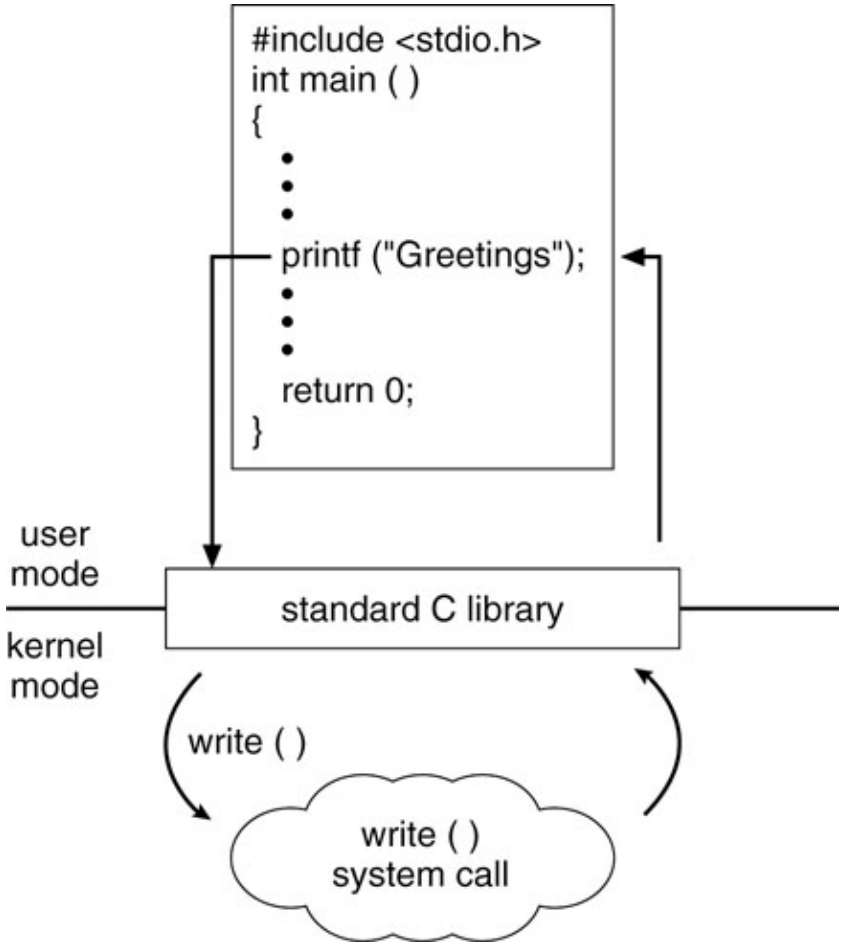
System Calls

API - System Call – OS Relationship



System Calls

Standard C Library Example



System Calls

System Call Parameter Passing

- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in registers
    - ...but what if there are more parameters than registers?
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - Each individual system call knows how to treat the table contents
    - 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

System Calls

Some relevant links to explore

- How to start developing your own OS kernel
  - https://tldp.org/LDP/LGNET/85/mahoney.html
- Java API
  - https://www.oracle.com/java/technologies/
- POSIX API
  - https://unix.org/apis.html
- Mac OS X API
  - https://en.wikipedia.org/wiki/Category:MacOS_APIs

System Calls

Types of System Calls

- Process Control
- File Management
- Device Management
- Information Maintenance
- Communications

System Calls

Process Control

- End, Abort
- Load, execute
- Create process, Terminate process
- Get/set process attributes
- Wait for time
- Wait/signal event
- Allocate and free memory

System Calls

File Management

- Create/Delete file
- Open, close
- Read, Write, reposition
- Get/Set file attributes

System Calls

Device Management

- Request device, release device
- Read, write, reposition
- Get/Set device attributes
- Logically attach or detach devices

System Calls

Information Maintenance

- Get/Set time or date
- Get/Set system data
  - Number of current users
  - Amount of free memory
  - (numerous other statistics)

System Calls

Communication

- Create, delete communication connection
- Send, receive messages
- Transfer status information
- Attach or detach remote devices


UNIVERSITY OF
LIMERICK
OLLSCOIL LUIMNIGH


CSIS
Department of
Computer Science
& Information Systems

Thank You

UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH

CSIS Department of Computer Science & Information Systems

CS4023 : Operating Systems

Sahir Sharma
University Teacher of Computer Science & Programming
CS2-035, Computer Science Building, University of Limerick
sahir.sharma@ul.ie
Autumn 2021-22

Week 3, Lecture 2

Previous Week/Lecture

Introduction

Computer System Organization

Computer System Architecture

Kernels

System Calls (Various Types)

What are we going to cover today?

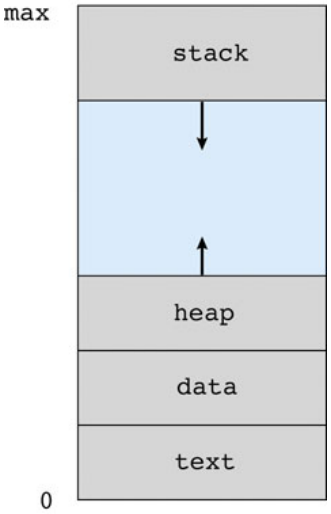
Processes

- What are they good for?
- What are they? Trivia
- Scheduling

Process (Definition)

- Process – a program in execution
- **Unit of work** in a modern time-sharing (multitasking) system
  - Terms job and process used almost interchangeably
  - Compare **process** and **thread** – not synonymous
- Process execution must progress in sequential fashion
- Needs certain resources (CPU time, memory, files, I/O devices) to accomplish its task
- Resources are allocated to a process either when it is created or while it is executing

Process (in Memory)



A process includes

- Text section
  - Program code, instructions
- Program counter
- Stack
  - Temporary data such as function parameters, return addresses, local variables
- Data section
  - Global variables
- Heap
  - Dynamically allocated memory during process run time

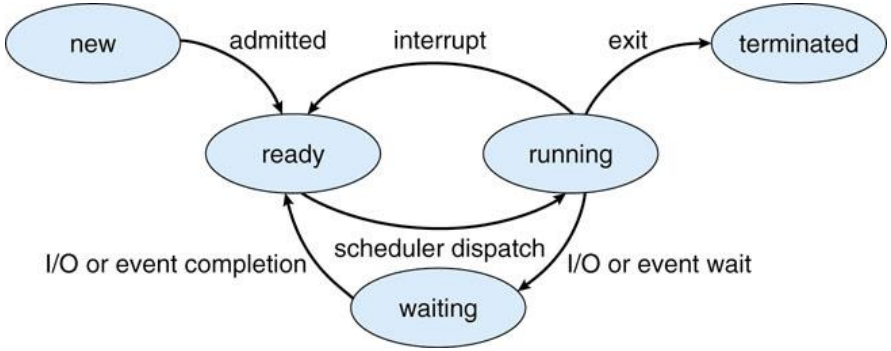
Read through for info on Memory layout of a process :

- https://en.wikipedia.org/wiki/Data_segment

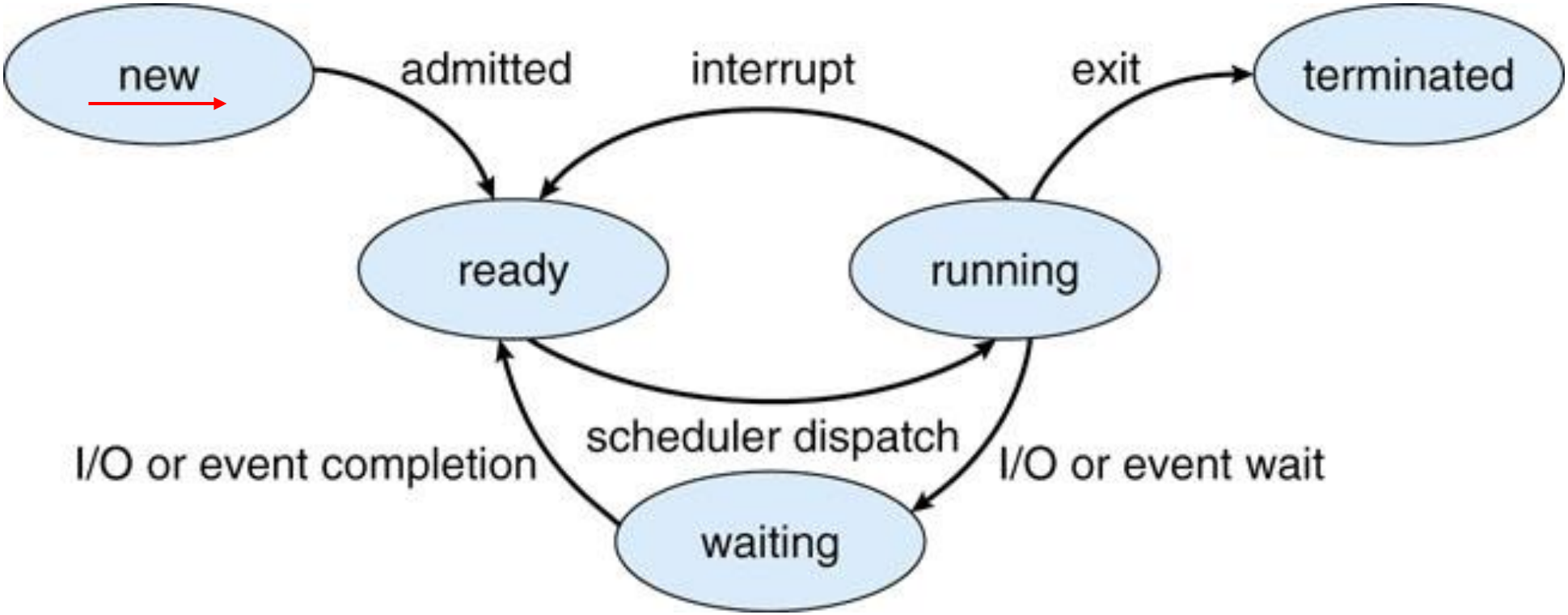
Process (State)

As a process executes, it changes state

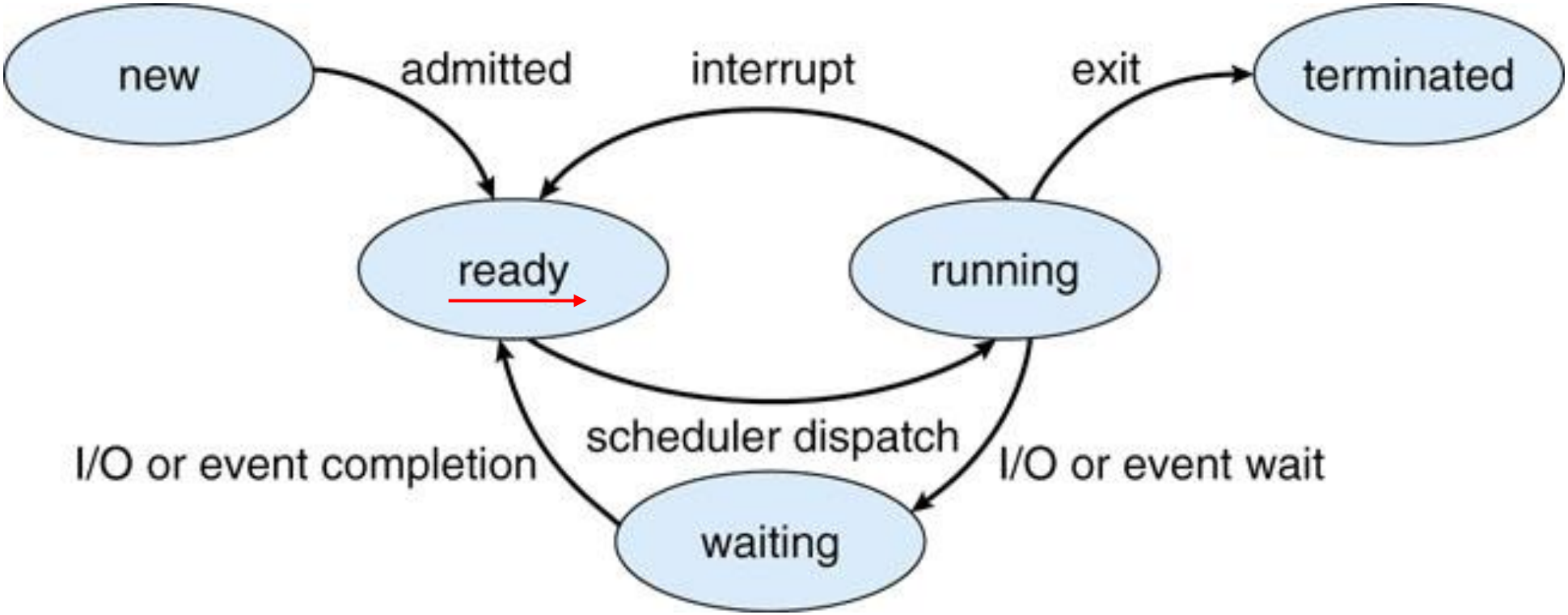
- New: The process is being created ; fork()
- Running: Instructions are being executed
- Waiting: The process is waiting for some event to occur
- Ready: The process is waiting to be assigned to a processor (CPU) to run
- Terminated: The process has finished execution



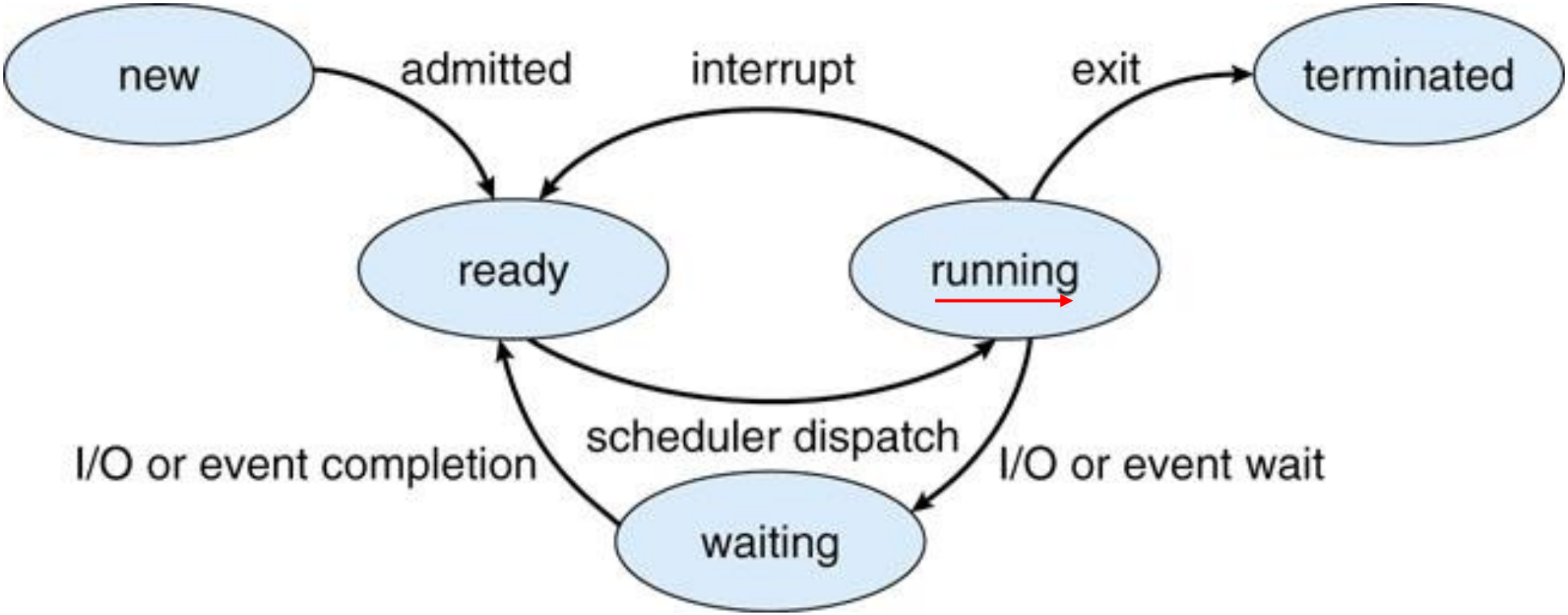
Process (State Transition Diagram)



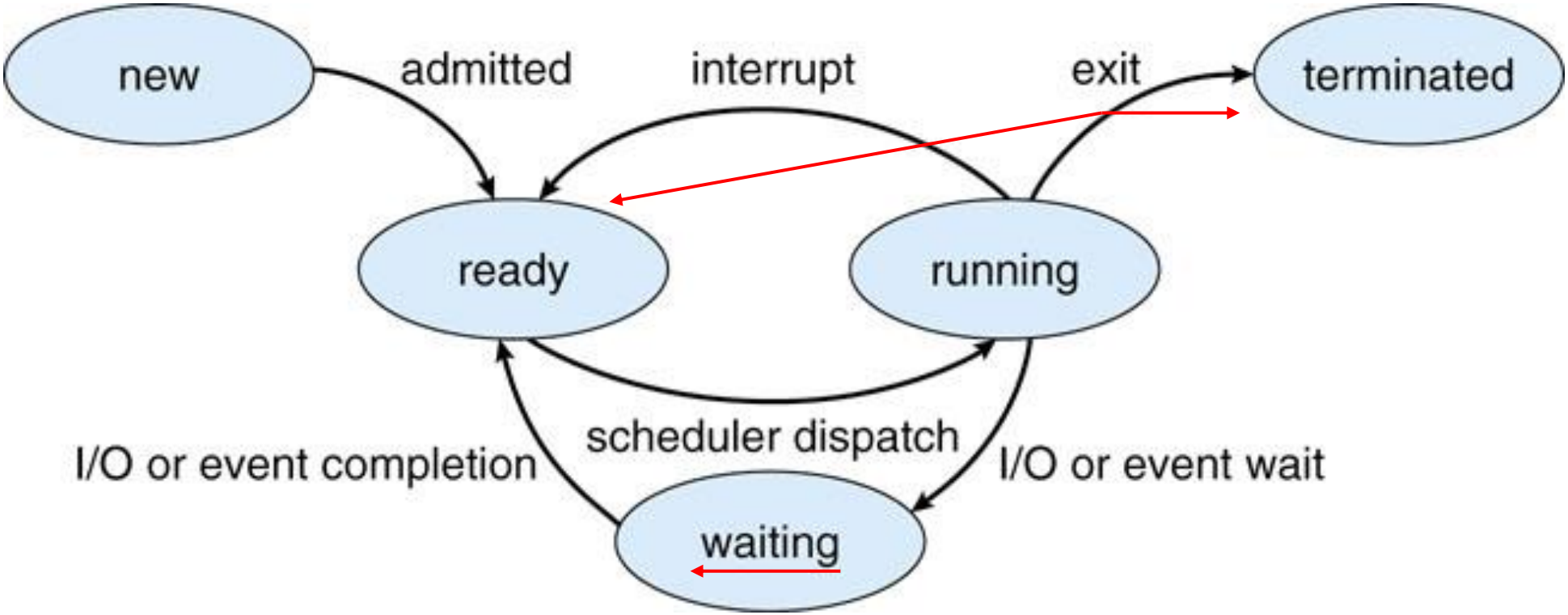
Process (State Transition Diagram)



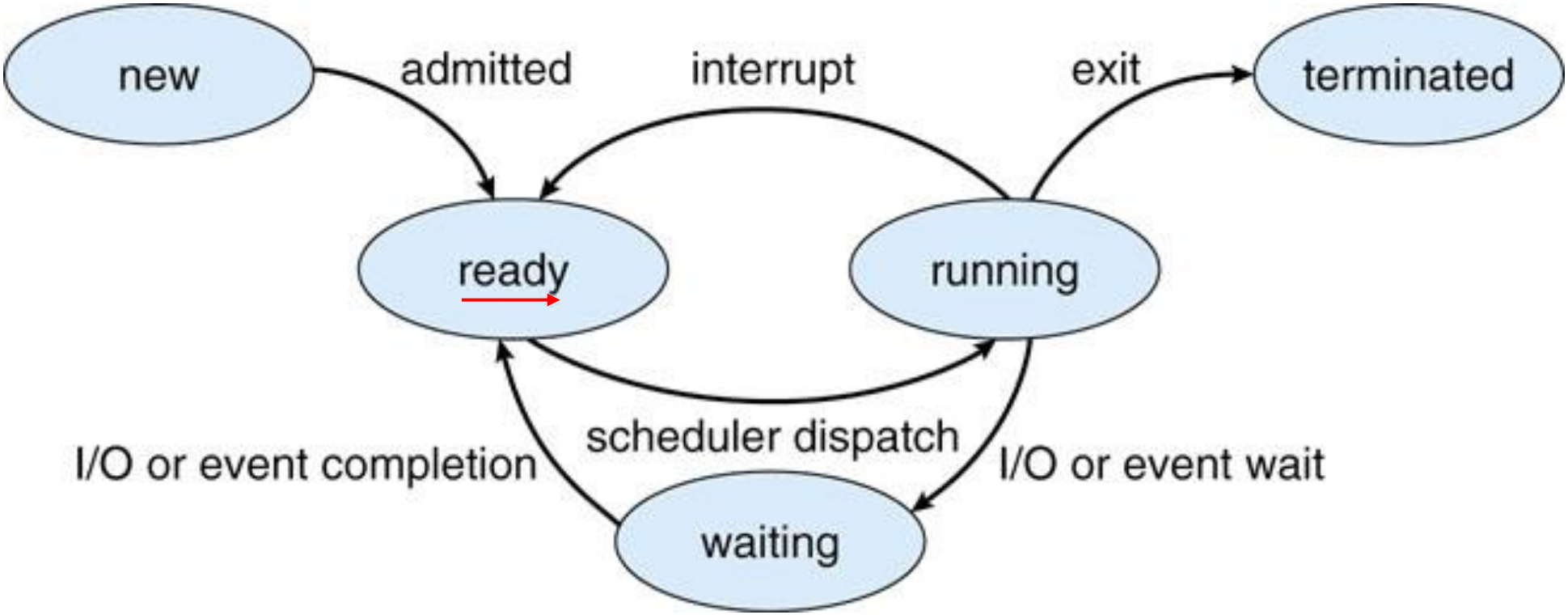
Process (State Transition Diagram)



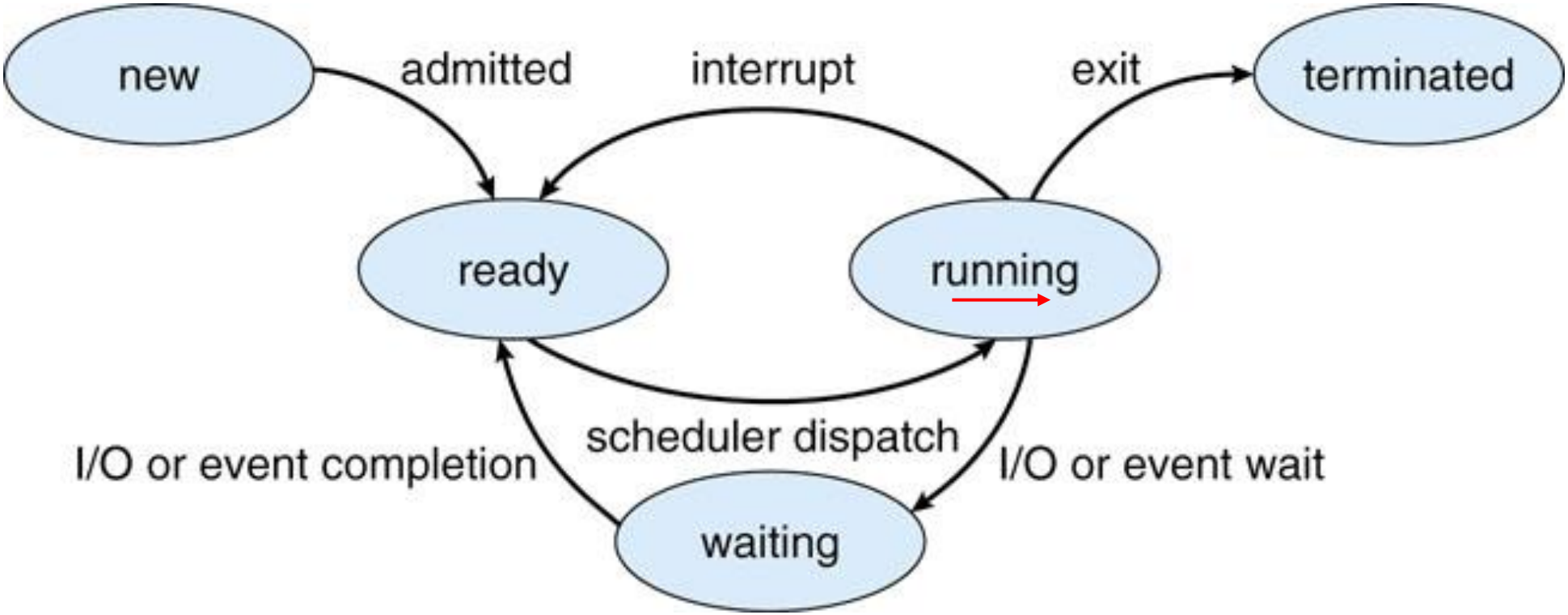
Process (State Transition Diagram)



Process (State Transition Diagram)



Process (State Transition Diagram)



Process

Process Control Block (PCB)

Information associated with each process

- Process state
- Program counter
- CPU registers
  - accumulators, index registers, stack pointers, etc.
- CPU scheduling information
  - process priority, pointers to scheduling queues, etc.
- Memory-management information
- Accounting information
  - Amount of CPU and real time used, time limits, account numbers, etc.
- I/O status information

Process

CPU Context Switch

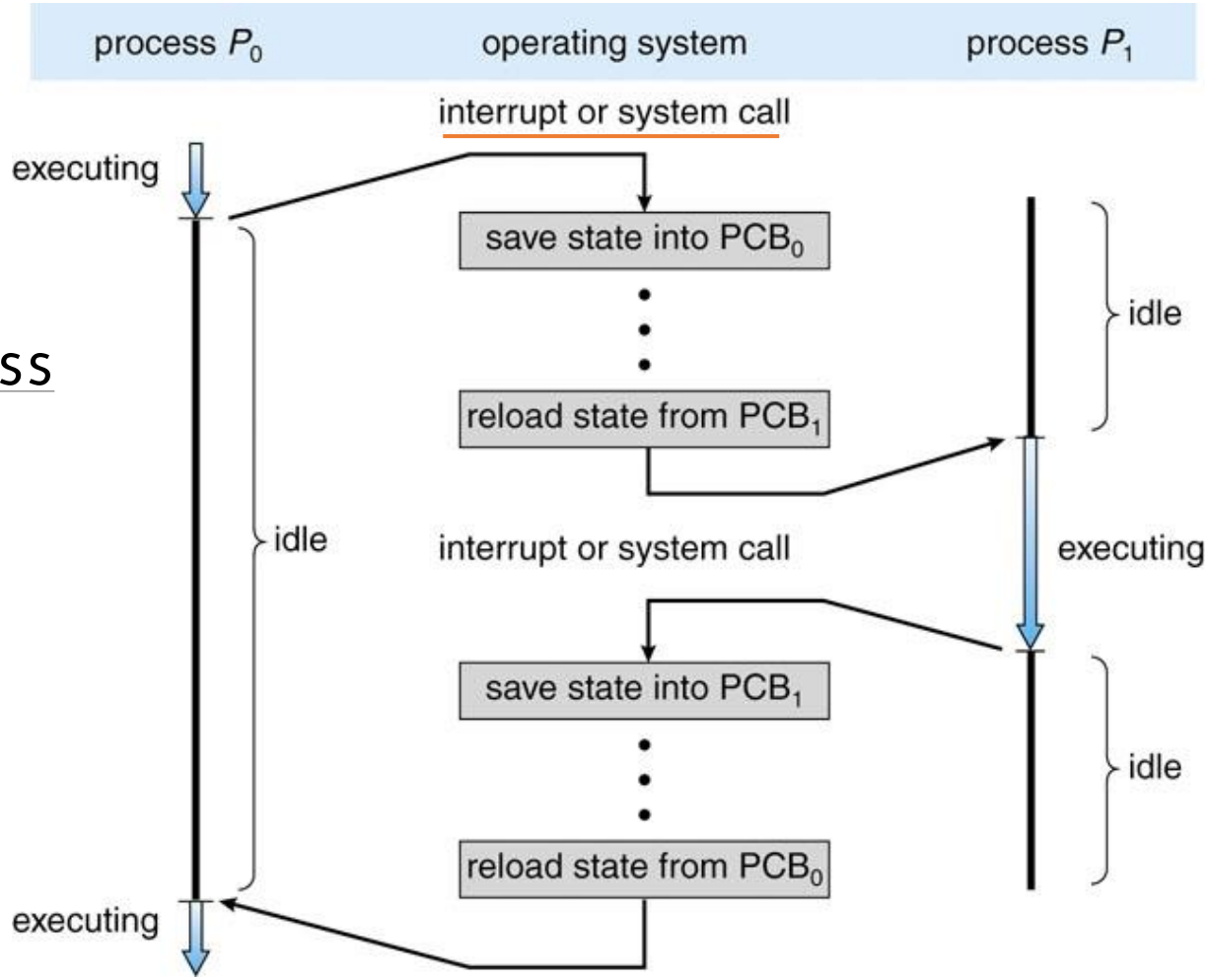
When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process

Context-switch time is overhead; the system does no useful work while switching

Time-dependent: needs hardware support

Process

- CPU (Context) Switch from Process to Process



Useful Trivia

Moore's Law Effect

Not a law; just a pattern that has held up so far

For almost the entire life of the computer transistor density has doubled every 18 months (Wikipedia)

Nothing like this in any other technology

Transportation in over 200 years:

- 2 orders of magnitude from horseback ~20km/h to Concorde 2,000km/h
- Computers have done this every decade (at least until 2011)!

Useful Trivia

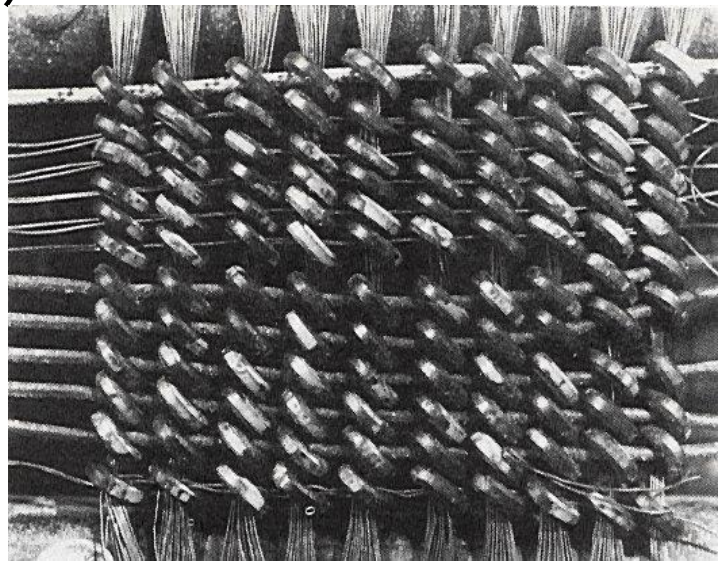
IBM 405 (First magnetic core memory)

The first magnetic core memory, from the IBM 405 Alphabetical Accounting Machine

Core Memory stored data as magnetization in iron rings

Iron cores woven into a 2-dimensional mesh of wires

Origin of the term “Core Dump”



Process

Process Scheduling Queues

Job queue: set of all processes in the system

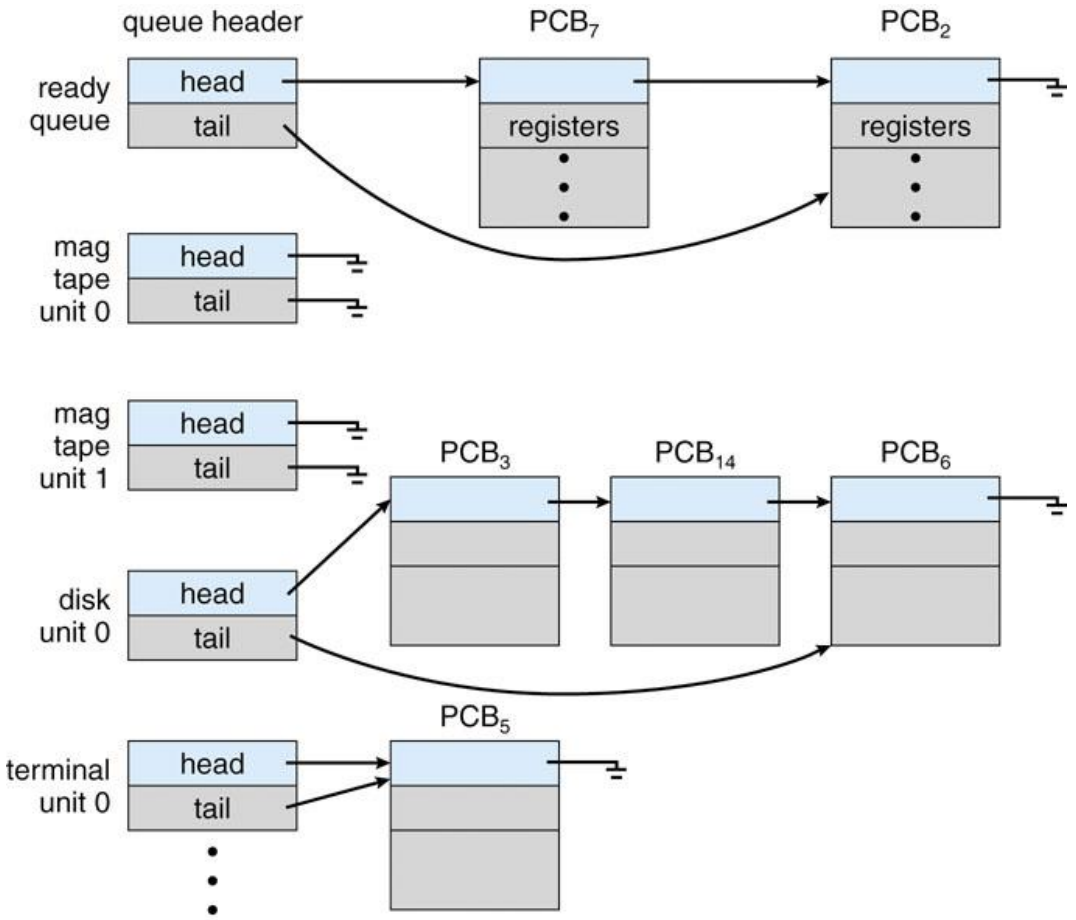
Ready queue: set of all processes residing in main memory, ready and waiting to execute

Device queues: set of processes waiting for an I/O device

Processes migrate among the various queues

Process

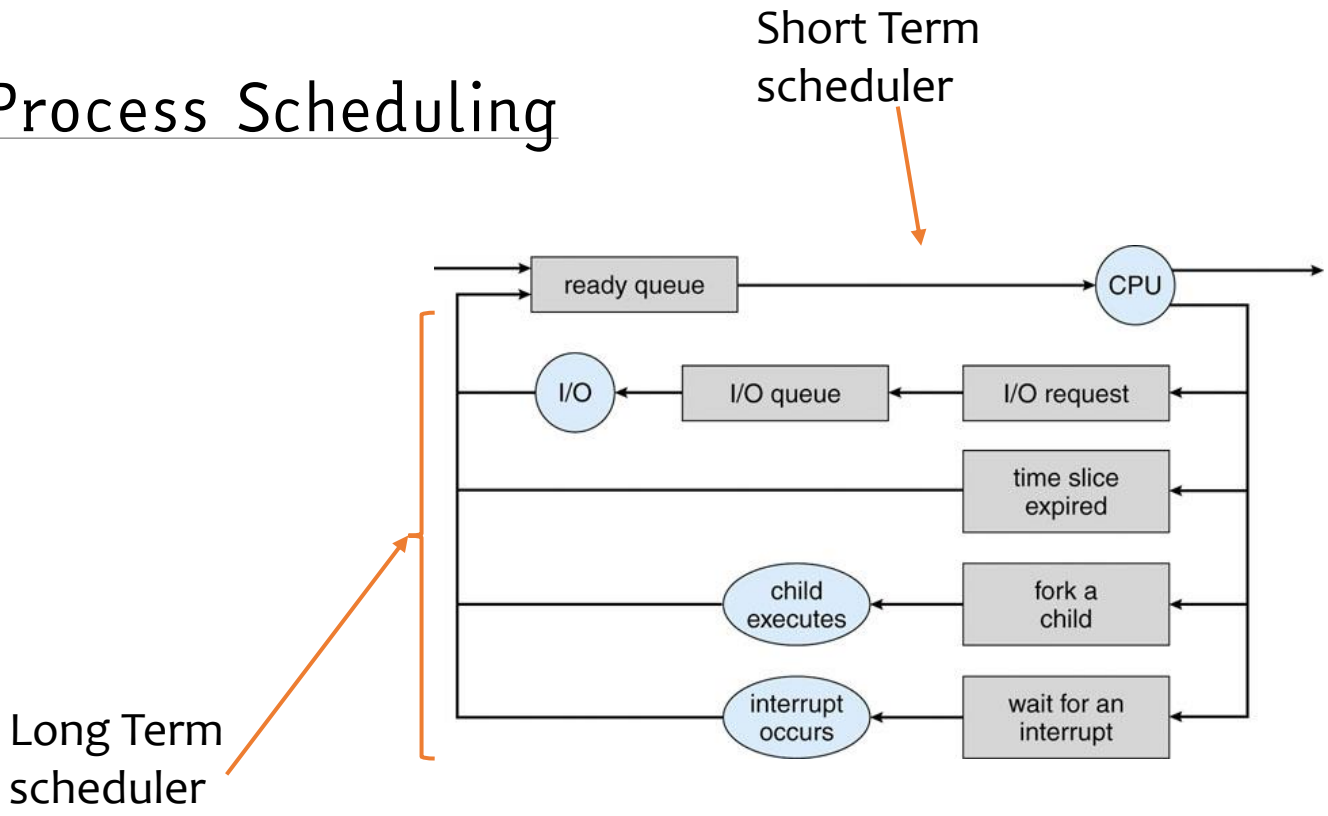
- Ready Queue and various I/O Device Queues



Process

Representation of Process Scheduling

- See also Wikipedia's Processor Scheduler page: https://en.wikipedia.org/wiki/Scheduling_(computing)



Process

Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU

Process

Schedulers (contd…)

- Short-term scheduler is invoked very frequently (milliseconds) so must be fast
- Long-term scheduler is invoked very infrequently (seconds, minutes) so can afford to be slow
  - The long-term scheduler controls the degree of multiprogramming (mix of active jobs)
- Processes may often be either:
  - I/O-bound process spends more time doing I/O than computations, many short CPU bursts
  - CPU-bound process spends more time doing computations; few very long CPU bursts
- Many OSs (UNIX, MS Windows) dispense with long-term scheduler

Process

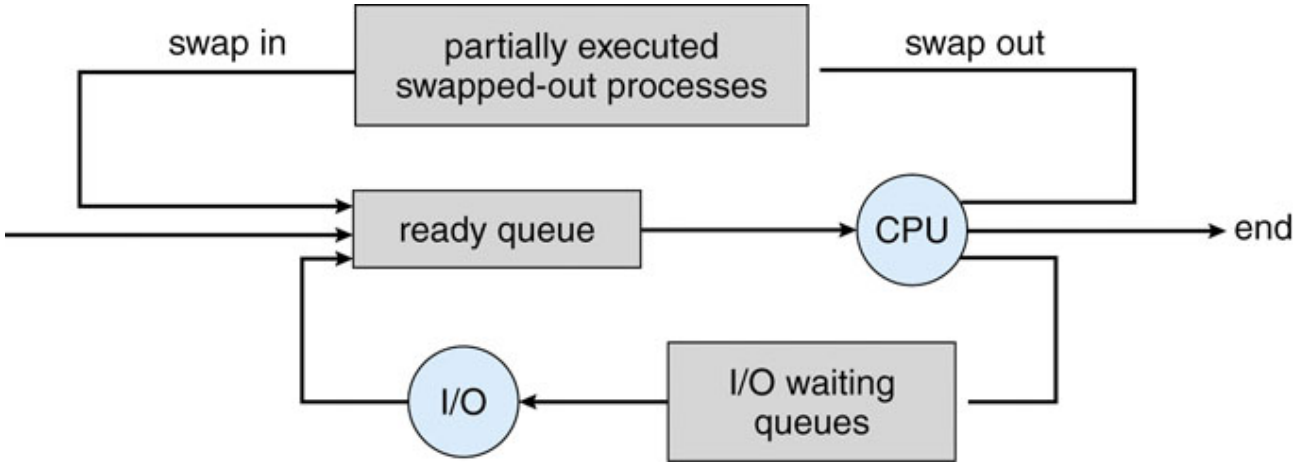
Who schedules the scheduler?

- We can think of the short-term scheduler as being the "overseer" deciding what job gets the CPU next
- But where does this "process" run?
  - On a different CPU?
  - Not if there's only one!
- See here for a discussion of how this trick is pulled off

Process

Addition of Medium-Term Scheduling

- Other OSs (such as time-sharing systems) introduce intermediate level of scheduling for jobs that are swapped out for reasons such as memory availability




UNIVERSITY OF
LIMERICK
OLLSCOIL LUIMNIGH


CSIS
Department of
Computer Science
& Information Systems

Thank You

UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH

CSIS Department of Computer Science & Information Systems

CS4023 : Operating Systems

Sahir Sharma
University Teacher of Computer Science & Programming
CS2-035, Computer Science Building, University of Limerick
sahir.sharma@ul.ie
Autumn 2021-22

Week 04 , Lecture 1

Announcements

- I hope everyone has setup their Linux Machines and can perform Remote Login
  - And are almost done with your Lab 03 Submissions
- Week04 lab: handin -m cs4023 -p w04

What are we going to do?

Processes: Part 2

- More on Context Switching
- Process Creation

Processes : Part 2

Context Switch in Detail

- A context switch is the switching of the CPU from one process or thread to another
- A **context** is the contents of a CPU's registers and program counter at any point in time
- A **register** is a small amount of very fast memory inside of a CPU that is used to speed the execution of computer programs by providing quick access to commonly used values
- A **program counter** is a specialized register that indicates the position of the CPU in its instruction sequence and which holds either the address of the instruction being executed or the address of the next instruction to be executed, depending on the specific system. Video(x1.5): https://www.youtube.com/watch?v=ccf9ngGIb8c
- Read through info site

Processes : Part 2

Context Switch in Detail

- Context switching can be described in more detail as the kernel performing the following activities with regard to processes (including threads) on the CPU:
  - Suspending the progression of one process and storing the CPU's state (i.e., the context) for that process somewhere in memory (as a PCB)
  - Retrieving the context of the next process from memory and restoring it in the CPU's registers and
  - Returning to the location indicated by the program counter (i.e., returning to the line of code at which the process was interrupted) in order to resume (execution of) the process

Processes : Part 2

Context Switch in Detail

- Context switches can occur **only** in kernel mode. Also known as system mode it is a privileged mode of the CPU in which only the kernel runs, and which provides access to all memory locations and all other system resources
- Other programs, including applications, initially operate in user mode, but they can run portions of the kernel code via system calls

Processes : Part 2

Context Switch in Detail

- The Intel x86 CPU provides a way of context switching completely in hardware, but for performance and portability reasons most modern OS's do context switches in software
- Software context switching can be used on all CPUs, and can be used to save and reload only the state that needs to be changed
- To use the hardware context switch you need to tell the CPU where to save the existing CPU state, and where to load the new CPU state from. The CPU state is always stored in a special data structure called a TSS (Task State Segment)

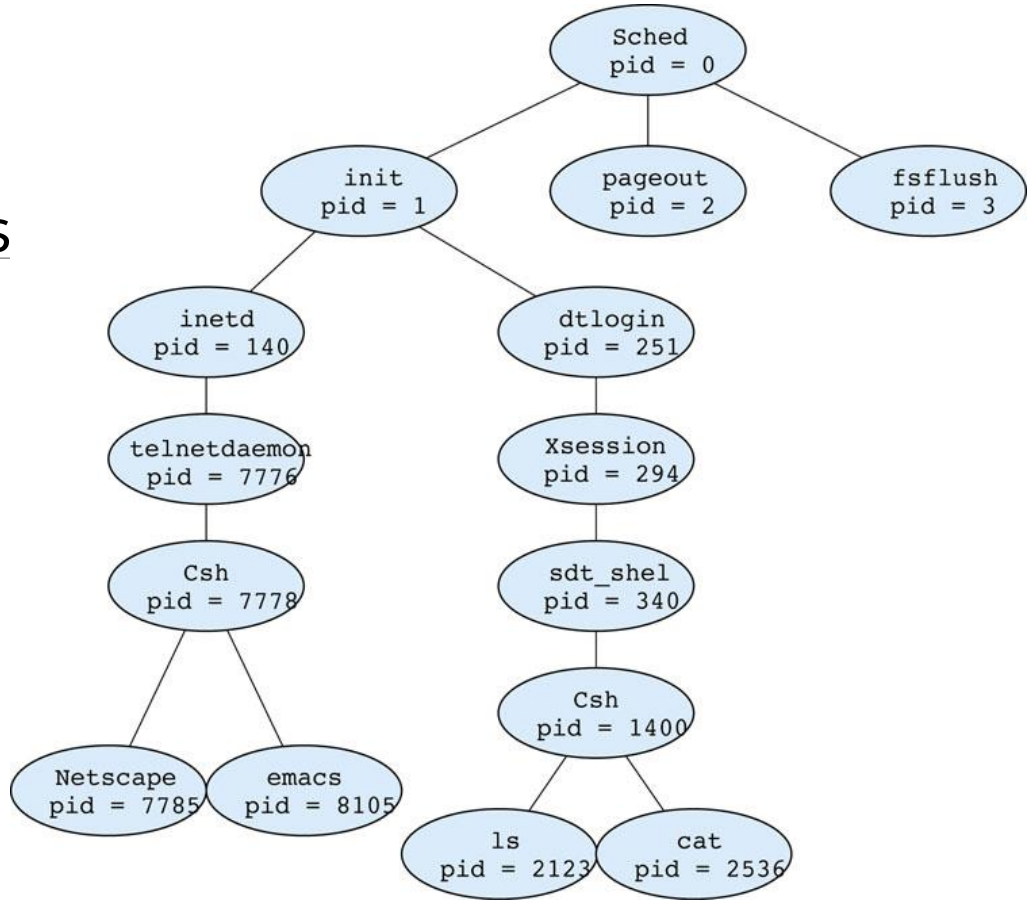
Process Creation

In the beginning there was one

- Parent processes create children processes, which, in turn create other processes, forming a tree of processes
- Most OSs identify processes by a **pid** (process ID)
- Resource sharing strategies / possibilities
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation

- Typical Tree of Processes (Solaris OS)



Process Creation (Contd.)

Concepts in Creation

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - fork() system call creates new (child) process
    - identical copy of parent's PCB with exception of process ID
  - Sometimes exec() system call may be used after a fork() to replace the process' memory space with a new program
- Another useful link: http://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html

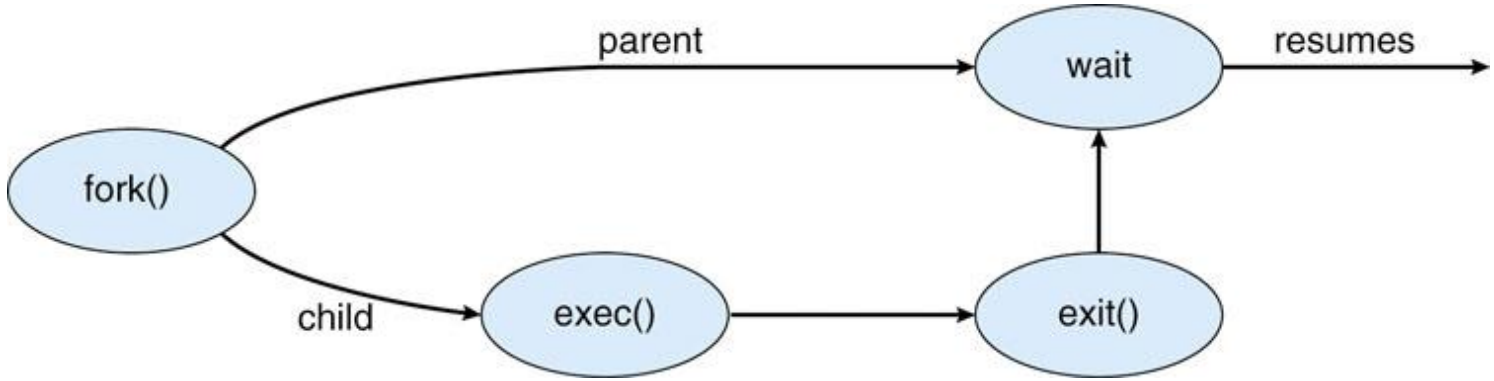
```
int main()
{
  pid_t  pidch;          /* pid of CHILD */

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

Process Creation (Contd.)

Concepts in Creation

- Flowchart for POSIX example of fork() on previous slide
- Parent waits until child terminates
- In other situations, may want both processes to live indefinitely so parent would not wait() on child



Process Creation (Contd.)

Concepts in Termination

- Process executes last statement and asks the operating system to delete it – exit()
  - Output data from child to parent via wait()
  - Child process' resources are then deallocated by operating system
- Parent may terminate execution of children processes (using kill() system call) if
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
- If parent is exiting
  - Some operating system do not allow child to continue if its parent terminates
  - All children terminated - cascading termination


UNIVERSITY OF
LIMERICK
OLLSCOIL LUIMNIGH


CSIS
Department of
Computer Science
& Information Systems

Thank You

UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH

CSIS Department of Computer Science & Information Systems

CS4023 : Operating Systems

Sahir Sharma
University Teacher of Computer Science & Programming
CS2-035, Computer Science Building, University of Limerick
sahir.sharma@ul.ie
Autumn 2021-22

Week 5, Lecture 1

Today (Last Day for any Lab Setup Queries)

If you are not able to setup the lab, it is because you have missed the live sessions in Week02/03 where some of the following instructions were mentioned:

- Usage of Virtual Machine to be a preference
- People Dual Booting, any issues by far?
- Problems with Credentials?
- If you don't know how to edit readFile, it is because you didn't pay attention to Lab01, and Lab02, or didn't readup online about linux.
  - You need to use a Text Editor like emacs or vim or gedit. There are many different kinds.

For people who are comfortable with this setup

Read more on sshfs and the storing a state of the VM on your hard disk, and then loading it back up to resume a session.

Life will be very easy, you might need not ssh again.

Previous Week/Lecture

Kernels

System Calls (Various Types)

Processes

Context Switching

Before we start

pid_t represents an integer in the system

Fork Command (Child Process Created)

fork-example.c

```
int main()
{
  pid_t  pidch;          /* pid of CHILD */

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

What are we going to cover today?

Threads

- What do they do?
- What are they?
- Multithreading
- Threading Issues
- Thread Libraries

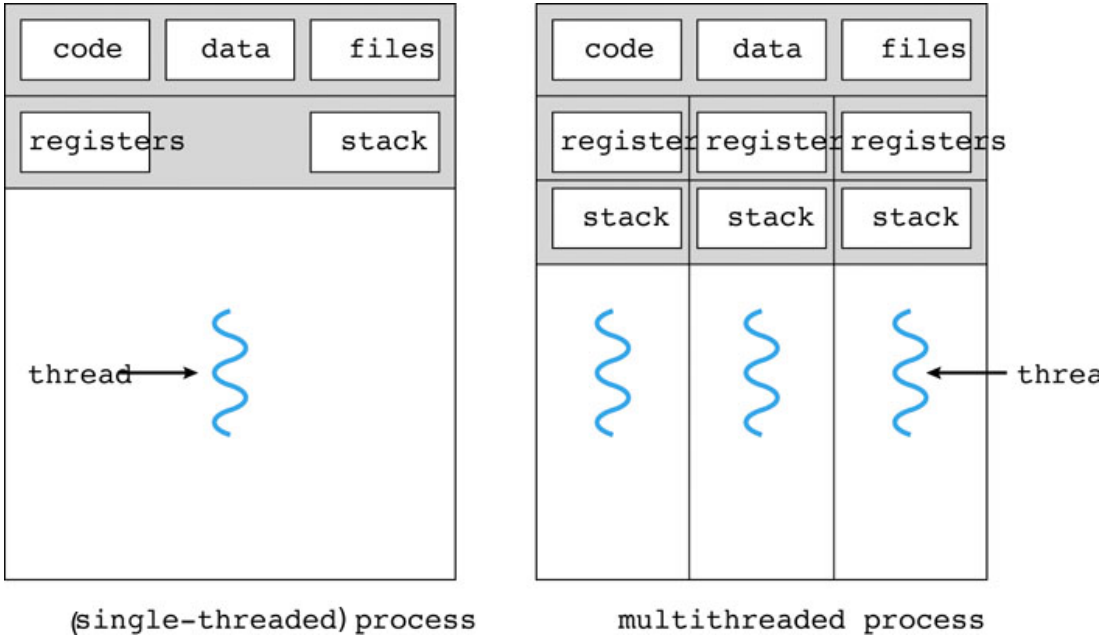
Thread (Definition)

- Thread – A thread is a path of execution within a process. A process can contain multiple threads.
- A thread is also known as lightweight process.
- The idea is to achieve parallelism by dividing a process into multiple threads.
- For example, in a browser, multiple tabs can be different threads. MS Word uses multiple threads: one thread to format the text, another thread to process inputs, etc.

Thread (in Memory)

Process: Unit of resources

Thread: Unit of scheduling and execution



Threads

Benefit

- Responsiveness
- Resource Sharing
- Economy
  - In Solaris, creating a process is about 30 times slower than is creating a thread, and context switching is about 5 times slower
- Utilization of multiprocessor architectures

Threads

Kernel –vs- User Threads

- **Kernel threads** are supported by almost every operating system currently; relatively expensive to create
- **User threads** Thread management done by user-level threads library
- Three primary thread libraries:
  1. POSIX Pthreads
  2. Win32 threads
  3. Java threads
- Ultimately, some relationship that maps from user- to kernel-thread must exist between the two: "shadowing"

Stack Overflow Explanation: https://stackoverflow.com/questions/15983872/difference-between-user-level-and-kernel-supported-threads

Threads

Thread Models

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

You can read

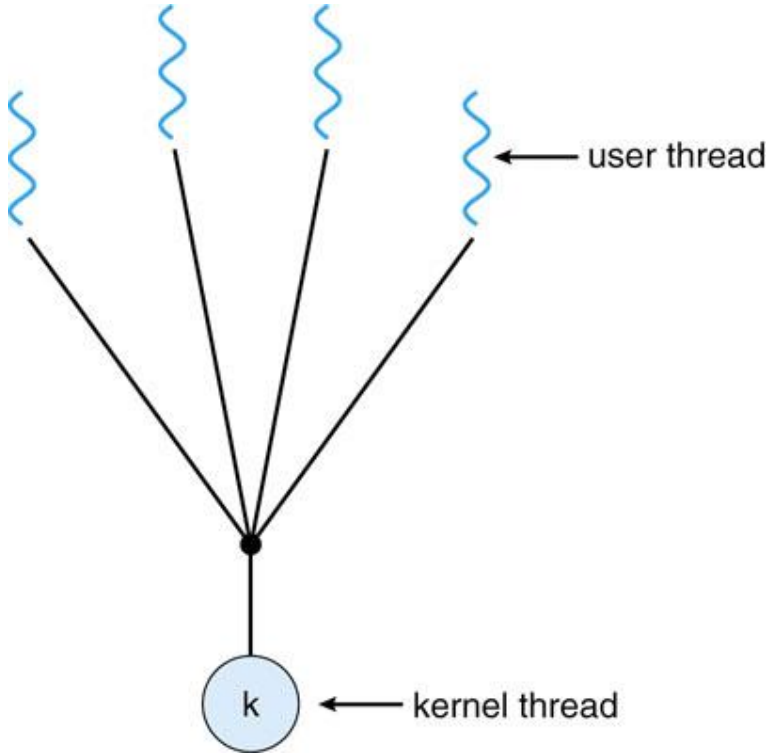
Processes, kernel threads

Kernel models

Thread Models

Many to One Thread Model

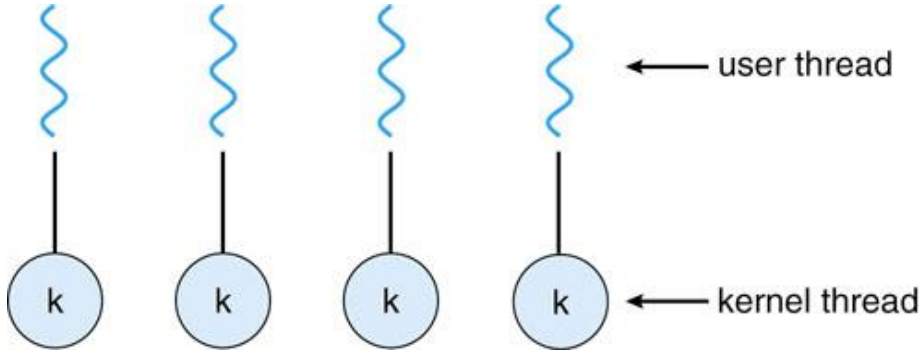
- Many user-level threads mapped to single **kernel thread**
- All threads blocked if kernel thread is blocked
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads
- One kernel thread) no parallelism (multi-cores useless)



Thread Models

One to One Thread Model

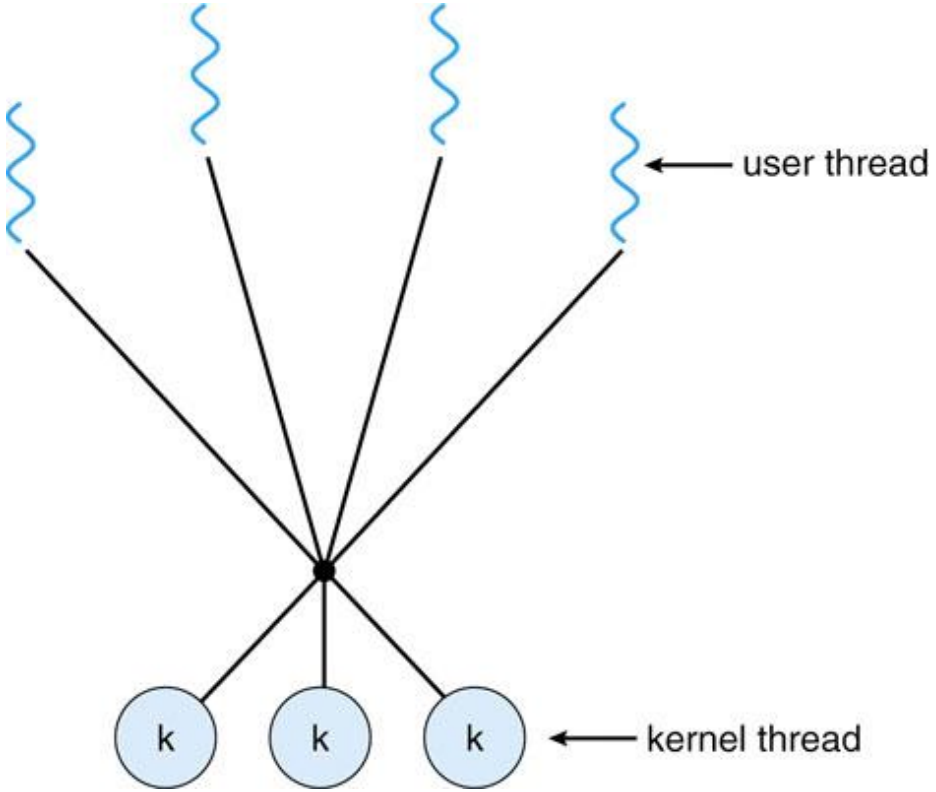
- Each user-level thread maps to kernel thread
- Blocking on a single kernel thread now not an issue but relatively high cost of kernel thread setup works against this approach
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later



Thread Models

Many to Many Thread Model

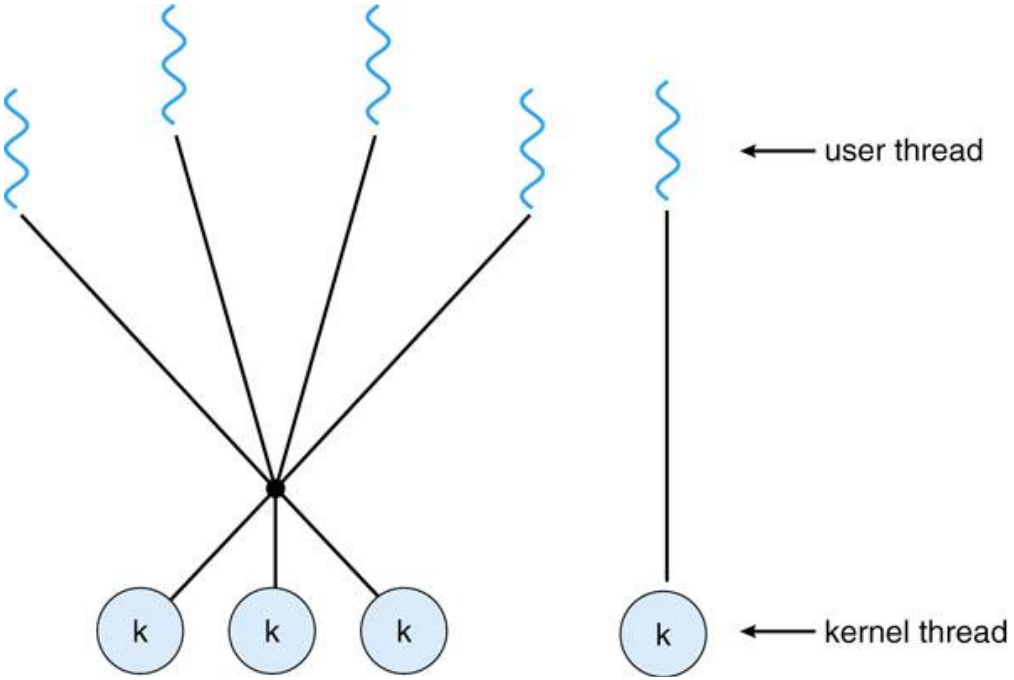
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
  - Solaris prior to v. 9
  - Windows NT/2000 with the ThreadFiber package



Thread Models

Two-level Model

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



Threading Issues

Some concerns

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

Threading Issues

Some concerns

- Signal handling
  - Signals are used in UNIX systems to notify a process that a particular event has occurred
  - A **signal handler** is used to process signals:
    - Signal is generated by particular event
    - Signal is delivered to a process
    - Signal is handled
  - Possibilities:
    - 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

Threading Issues

Some concerns

- Thread pools
  - Create a number of threads in a pool where they await work
  - Advantages
    - Usually slightly faster to service a request with an existing thread than create a new thread
    - Allows the number of threads in the application(s) to be bound to the size of the pool
- Thread-specific data
  - Allows each thread to have its own copy of data
  - Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

Threading Issues

Scheduler Activations

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

Thread Libraries

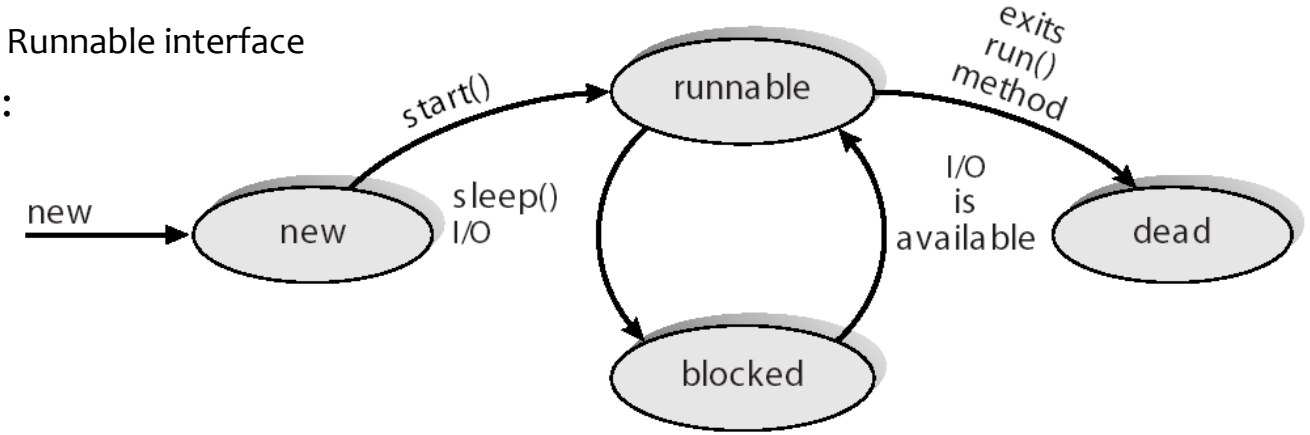
Pthreads

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

Thread Libraries

Java Threads

- Java threads are managed by the JVM which runs on some OS
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface
- Java thread states:



Thread Libraries

Linux Threads

- Linux refers to them as tasks rather than threads
- Thread creation is done through clone() system call
- clone() allows a child task to share the address space of the parent task (process)
- amount of "data sharing" with parent controlled by following flags

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

Thread Libraries

Windows XP Threads

- Implements the one-to-one mapping
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
- The register set, stacks, and private storage area are known as the context of the threads
- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

Thread Libraries

Windows Vista Enhancements

- Thread pool significantly enhanced to improve reliability, performance, and API simplicity
- Some specific examples of Windows Vista thread pool enhancements include:
  - Cleanup groups - introduce the ability to clean up pending thread pool requests on process shutdown
  - Multiple pools per process, each of which is scheduled independently. This significantly improves responsiveness as well as code isolation
  - Performance. The thread pool is optimized to provide the best possible processor usage and to reduce pressure on system memory. Thread recycling, maintaining a number of running threads that is proportional to the number of processors, smart load balancing, and work assignment are used to gain maximum performance from the thread pool.


UNIVERSITY OF
LIMERICK
OLLSCOIL LUIMNIGH


CSIS
Department of
Computer Science
& Information Systems

Thank You

UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH

CSIS Department of Computer Science & Information Systems

CS4023 : Operating Systems

Sahir Sharma
University Teacher of Computer Science & Programming
CS2-035, Computer Science Building, University of Limerick
sahir.sharma@ul.ie
Autumn 2021-22

Week 05 , Lecture 2

Announcements

- The following are the most common issues:
  - VPN Problems
  - Credential Problems
  - Virtual Machine Setup
  - Reading/Writing into the Terminal Problems
  - Anything else?

What are we going to do?

Interprocess Communication

- Introduction
- Shared-Memory IPC

Interprocess Communication

Introduction

- Interprocess communication is the mechanism provided by the operating system that allows processes to communicate with each other.
- This communication could involve a process letting another process know that some event has occurred or the transferring of data from one process to another

Credits:

https://www.tutorialspoint.com/what-is-interprocess-communication#:~:text=Interprocess%20communication%20is%20the%20mechanism,from%20one%20process%20to%20another.

Interprocess Communication

Cooperating Processes

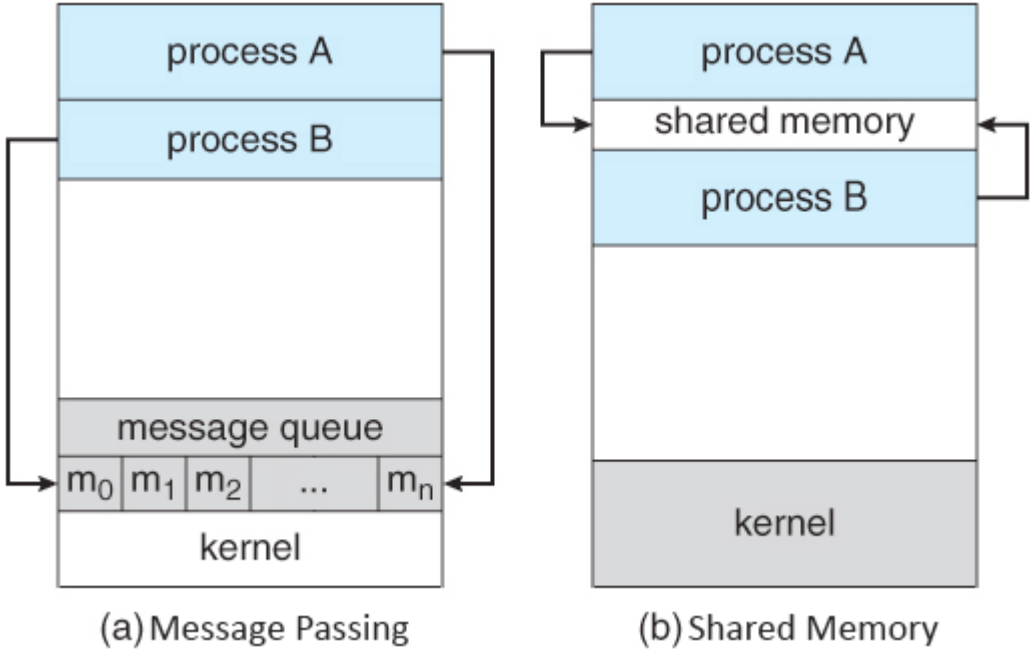
- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation (Several reason why we should promote it)
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Any process that shares data with other processes is a cooperating process

Interprocess Communication

Models

- Two fundamental models of IPC:
  - Shared memory
    - Faster, no assistance from the kernel required once the shared memory is established
    - Conflicts need be avoided: problem of overwriting
  - Message passing
    - Slow, requires assistance from the kernel
    - No conflicts for us to manage: kernel manages low-level details



Interprocess Communication

Shared Memory

- In computer science, shared memory is memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies.
- Shared memory is an efficient means of passing data between programs.
- Depending on context, programs may run on a single processor or on multiple separate processors.
- Using memory for communication inside a single program, e.g. among its multiple threads, is also referred to as **shared memory**.

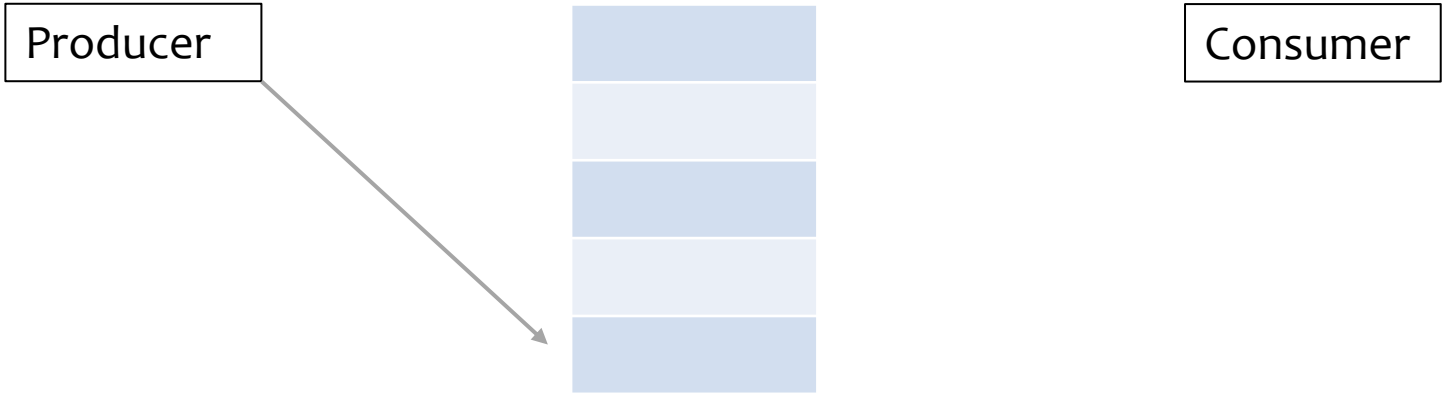
Shared Memory - IPC

Producer-Consumer Problem

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

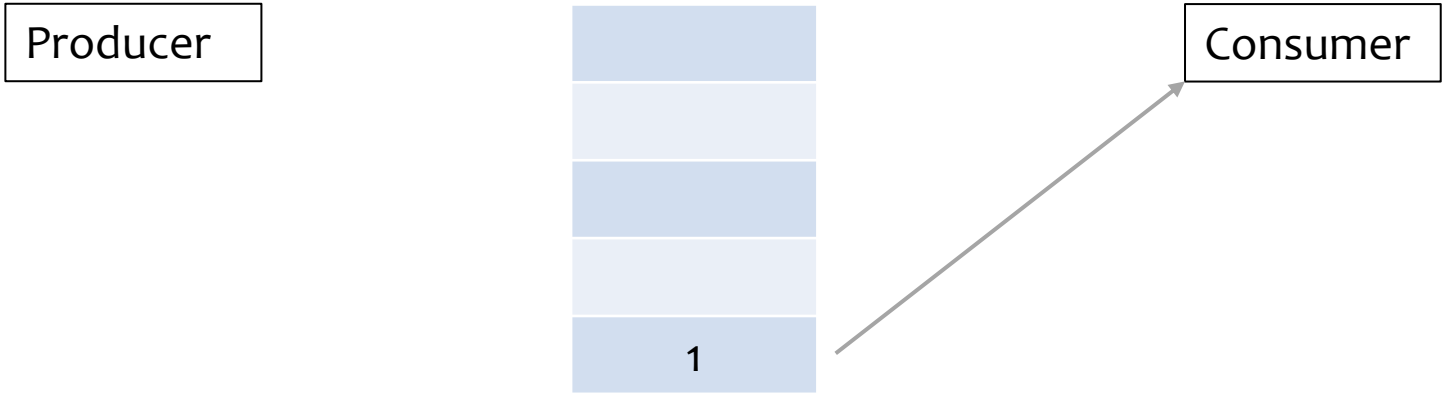
Shared Memory - IPC

Producer-Consumer Problem



Shared Memory - IPC

Producer-Consumer Problem



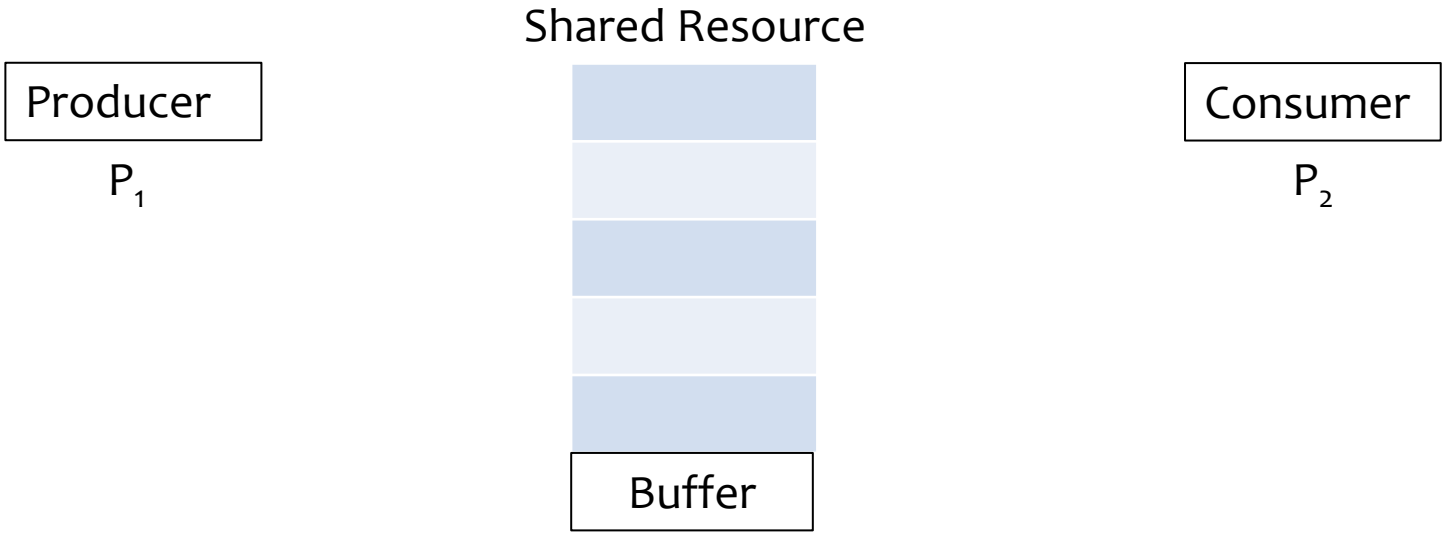
Shared Memory - IPC

- Producer-Consumer Problem

Problem?

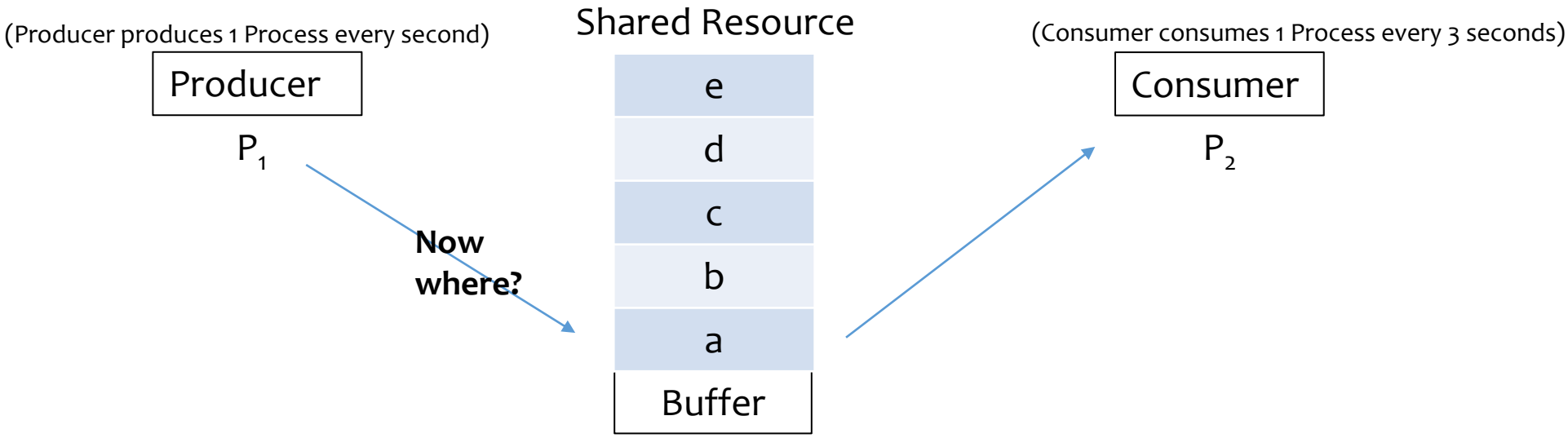
Shared Memory - IPC

Producer-Consumer Problem



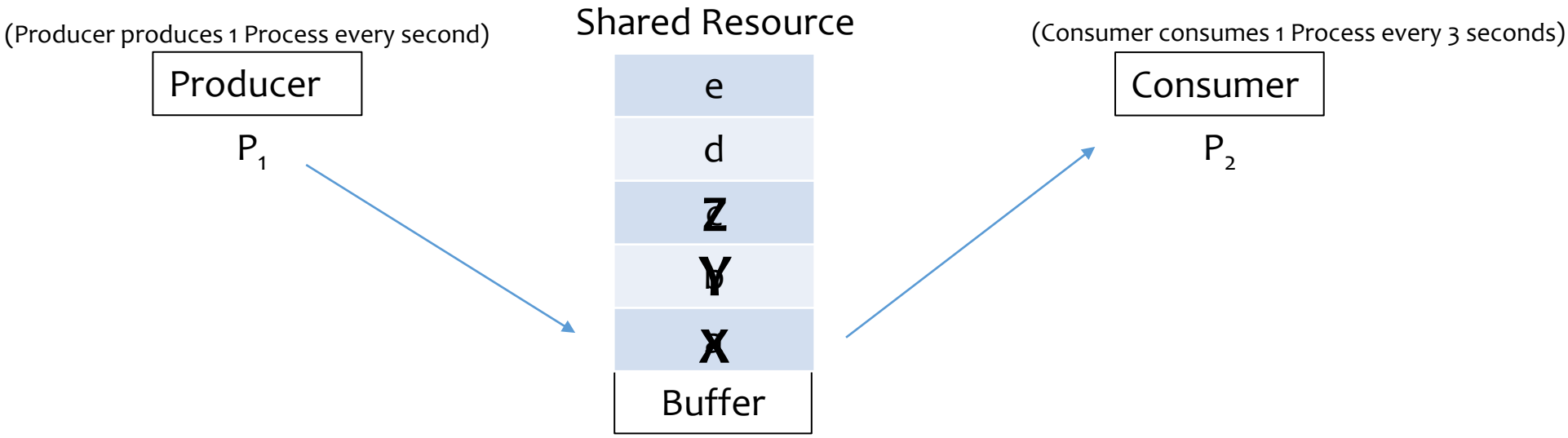
Shared Memory - IPC

Producer-Consumer Problem – Stack Overflow



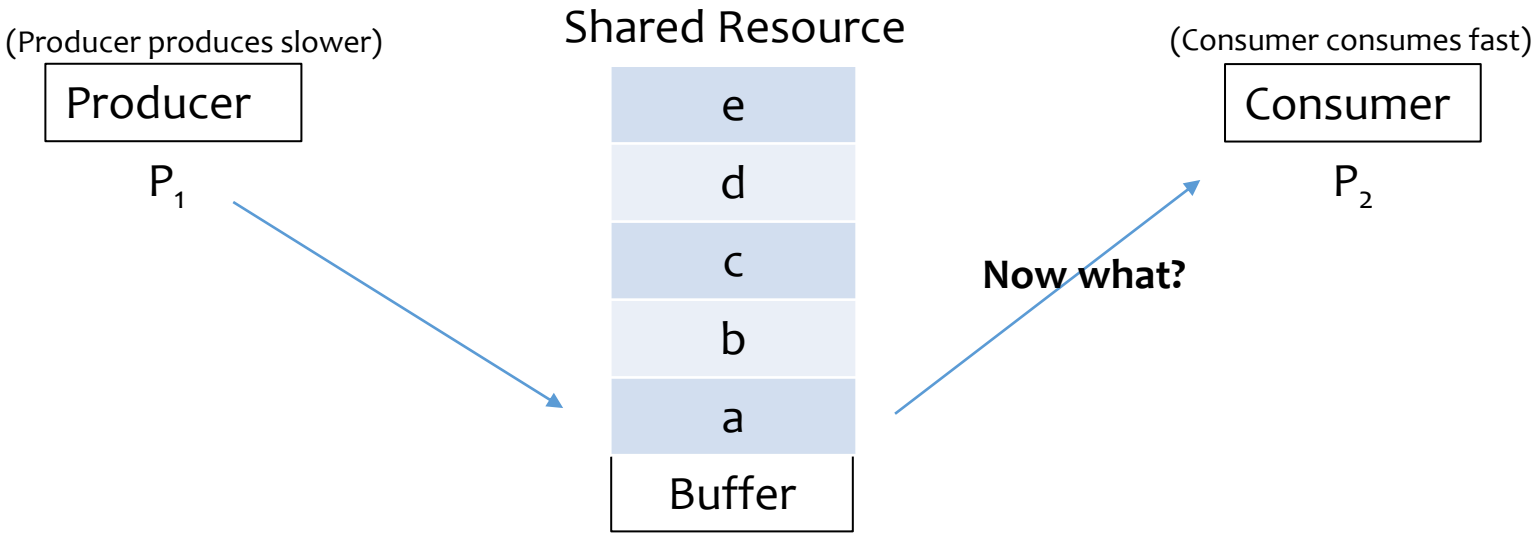
Shared Memory - IPC

Producer-Consumer Problem – Buffer Overwrite (Cyclic Queue)



Shared Memory - IPC

Producer-Consumer Problem – Empty Buffer



Shared Memory - IPC

Producer-Consumer Problem

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

Shared Memory - IPC

Bounded Buffer: Shared-Memory Solution

```
const int BUFFSZ = 10;                  // size of buffer
typedef struct {
. . .
} item;                                 // item is a new data type (typedef)
item buffer[BUFFSZ];                    // a store of items, think of it as a circular stack
                                        // at most BUFFSZ

int in = 0;
int out = 0;
```

- in - index of next free position
- out - index of first full position

Shared Memory - IPC

Bounded Buffer: The Consumer Process

```
item nextConsumed;
while (true) {
  while (out == in)
    ; // do nothing but wait: nothing to consume
  nextConsumed = buffer[out];
  : /* code here */
  : /* consumes item in nextConsumed */
  out = (out + 1) % BUFFSZ;
}
```

Wait for Producer

Shared Memory - IPC

Bounded Buffer: The Producer Process

```
item nextProduced;
while (true) {
  : /* code here should */
  : /* produce an item in nextProduced */
  while (((in + 1) % BUFFSZ) == out)
    ; /* do nothing -- no free buffers */
  buffer[in] = nextProduced;
  in = (in + 1) % BUFFSZ;
}
```

Testing for a condition, as long as the condition holds, we do nothing

Shared Memory - IPC

Bounded Buffer: Analysis

- Previous solution is correct but there must **always** be one unused "slot":
  - If all slots in use then in == out
  - But then consumer will never proceed as it thinks there is **nothing** available
  - Can use at most BUFFSZ - 1 slots
- Consider alternative solution which makes use of a shared integer count that keeps track of the number of full buffers
  - Initially, count is set to 0
  - It is incremented by the producer after it produces a buffer and is decremented by the consumer after it consumes a buffer
  - Safe???

Shared Memory - IPC

Bounded Buffer: Alternative Producer Process

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

```
item nextProduced;
while (true) {
  : /* code here should produce an item in nextProduced */
  while (((in + 1) % BUFFSZ) == out)
    ; /* do nothing -- no free buffers */
  buffer[in] = nextProduced;
  in = (in + 1) % BUFFSZ;
}
```

Shared Memory - IPC

Bounded Buffer: Alternative Consumer Process

```
item nextConsumed;
while (true) {
   while (count == 0)
      ; // do nothing but wait: nothing to consume
   nextConsumed = buffer[out];
   out = (out + 1) % BUFFSZ;
   count--;
   /* ready to consume item in nextConsumed */
}
```

Shared Memory - IPC

IRC Example (POSIX)

- Shared memory: shmget()
- make request for some shared memory with:
  segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR)
- now do something with it
  shared_mem = (char *) shmat(id, NULL, 0);
  sprintf(shared_mem, "Writing to shared memory");
- See also Shared Memory – w.r.t Client, Server
- Race condition at start


UNIVERSITY OF
LIMERICK
OLLSCOIL LUIMNIGH


CSIS
Department of
Computer Science
& Information Systems

Thank You

UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH

CSIS Department of Computer Science & Information Systems

CS4023 : Operating Systems

Sahir Sharma
University Teacher of Computer Science & Programming
CS2-035, Computer Science Building, University of Limerick
sahir.sharma@ul.ie
Autumn 2021-22

Week 6, Lecture 1

Administrative Details

You might have received a mail from ICT yesterday

It is a reminder to get in touch with them if you are still facing any issues with C and Linux (ICT Learning Center)

Check Week06 > Resources (Secondary Section) for Schedule on ICT Classes by tutors

Previous Week/Lecture

Threads

Introduction to IPC (Inter-Process Communication)

Shared Memory model

Producer-Consumer Problem

Alternate Solution to problem (Count tracking)

IPC Example (POSIX)

What are we going to cover in this lecture?

- Interprocess Communication (Message-Passing IPC)

Before we start – lets discuss some code

```
/*SERVER CODE*/
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<stdio.h>
#include<stdlib.h> /* for exit() */

#define SHMSZ 1025
```

Before we start – lets discuss some code

```
int main()
{
        int shmid; /*future id of the shared memory chunk*/
        key_t key;
        char *shm;

        key =5678
/* Name of our shared memory segment
```

Before we start – lets discuss some code

#define SHMSZ 1025

3 Subparts (Each **Octal**)
User, Group, Others
4+2+1 (Read, Write, Execute)

```
shmid = shmget (key, SHMSZ, IPC_CREAT|0666);
/*Create the segment and check if ok */
if(shmid<0)
{
        perror("shmget");
        exit(1);
}
```

System Call

Now Testing
If shmid is less than 0

Before we start – lets discuss some code

```
/*Now attach the segment to our data space*/
        if((shm = shmat(shmid, NULL, 0)) == (char *) -1)
        {
                perror("shmat");
                exit(1);
        }
```

Still Testing

This time if shm is -1

If we succeed in attaching or not (-1)

Let's hope everything went okay

Before we start – lets discuss some code

```
/*Now put something into the shared memory for the other process to read*/
        char *greeting = "Hello World";
        char *p, *s;
        for(p = greeting, s=shm ; *p ; p++, s++)
        {
                *s = *p;
        }
/*Now wait for the other process to let us know that it's consumed it*/
        while (*shm != '*')
                sleep(1);
        exit(0);
}
```

Max: 1024 characters
Save data

Point to first character element in greetings

Kind of Synchronizing
Write in shared memory, one by one

(also, refer to 0666)

Before we start – lets discuss some code

```
/*CLIENT CODE*/
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<stdio.h>
#include<stdlib.h> /* for exit() */

#define SHMSZ 1025
```

Same as Server

Before we start – lets discuss some code

```
int main()
{
        int shmid;
        key_t key;
        char *shm;

        key =5678 /* Same Name of our shared memory segment
```

Same as Server

Before we start – lets discuss some code

```
/*Locate the segment and check that all was ok*/
        if((shmid = shmget(key, SHMSZ, 0666)) < 0)
        {
                perror("shmget");
                exit(1);
        }
```

Compare with Server's Call

Let's hope everything went okay

Before we start – lets discuss some code

```
/*Now we attach this segment to our data space*/
        if((shm = shmat(shmid, NULL, 0)) == (char *) -1)
        {
                perror("shmat");
                exit(1);
        }
```

Same as Server

Before we start – lets discuss some code

```
/*Now read what the server put in the memory an put to screen*/
        char *p;
        for(p = shm ; *p ; p++)
                putchar(*p);
        putchar('\n');

/*Now the all-important notification*/
        *shm = '*';
        exit(0);
}
```

Put to screen the char that p point to

Puts the * in address pointed by shm, signaling the consumption

Before we start – lets discuss some code

Glitches/Bugs/Optimizations in the code? Bring me some – through blogs

How does the client know "Hello world." is already pointed to by shm

But is it possible the client will read "Hello world." before the server writes it?

Or try to read?

How would you prevent that?

Race Condition?

Message-Passing IPC

- Alternative mechanism for processes to communicate and to synchronize their actions
- Processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - send (message) – message size fixed or variable
  - receive (message)
- If P and Q wish to communicate, they need to:
  - Establish a communication link between them
  - Exchange messages via send/receive
- Implementation of communication link
  - Physical (e.g., shared memory, hardware bus)
  - Logical (e.g., logical properties)

Message-Passing IPC

Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

Message-Passing IPC

Direct Communication

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

Message-Passing IPC

Indirect Communication

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

Message-Passing IPC

Indirect Communication (Contd…)

- Operations
  - Create a new mailbox
  - Send and receive messages through mailbox
  - Destroy a mailbox
- Primitives are defined as:
  - Send(A, message) – send a message to mailbox A
  - Receive(A, message) – receive a message from mailbox A

Message-Passing IPC

Indirect Communication (Contd...)

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1 sends; P_2 and P_3 receive
  - Security Problem : Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute 1 receive operation
  - Allow the system to select the receiver, arbitrarily . Sender is notified who the receiver was.

Message-Passing IPC

Synchronization

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

Message-Passing IPC

Summary

- Message Passing –vs- Shared Memory : https://www.youtube.com/watch?v=SmJpGl_PYz8


UNIVERSITY OF
LIMERICK
OLLSCOIL LUIMNIGH


CSIS
Department of
Computer Science
& Information Systems

Thank You

UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH

CSIS Department of Computer Science & Information Systems

CS4023: Operating Systems

Sahir Sharma
University Teacher of Computer Science & Programming
CS2-035, Computer Science Building, University of Limerick
sahir.sharma@ul.ie
Autumn 2021-22

Week 06 , Lecture 2

CPU Scheduling: Part 1

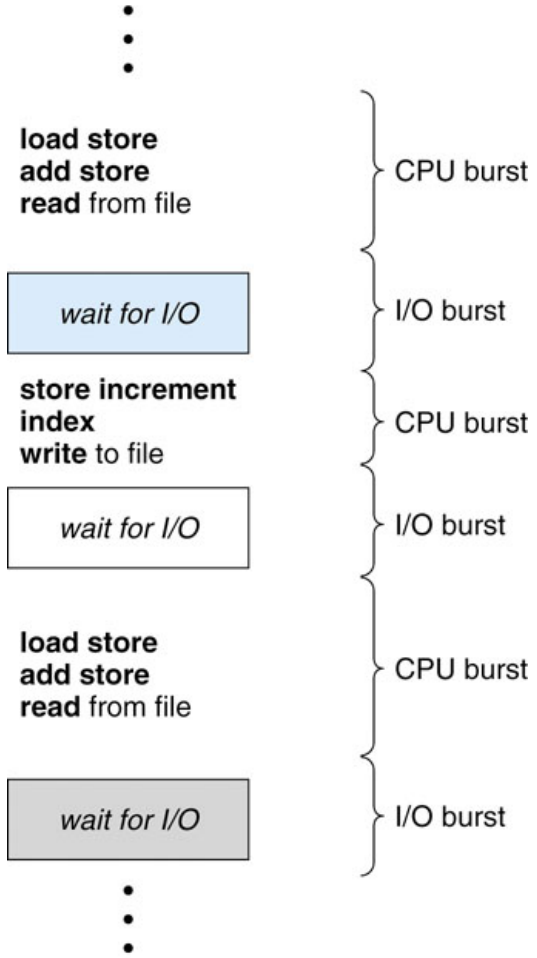
Introduction

- In computer, scheduling is the action of assigning resources to perform tasks.
- The resources may be processors, network links or expansion cards.
- The tasks may be threads, processes or data flows.
- The scheduling activity is carried out by a process called scheduler.

CPU Scheduling: Part 1

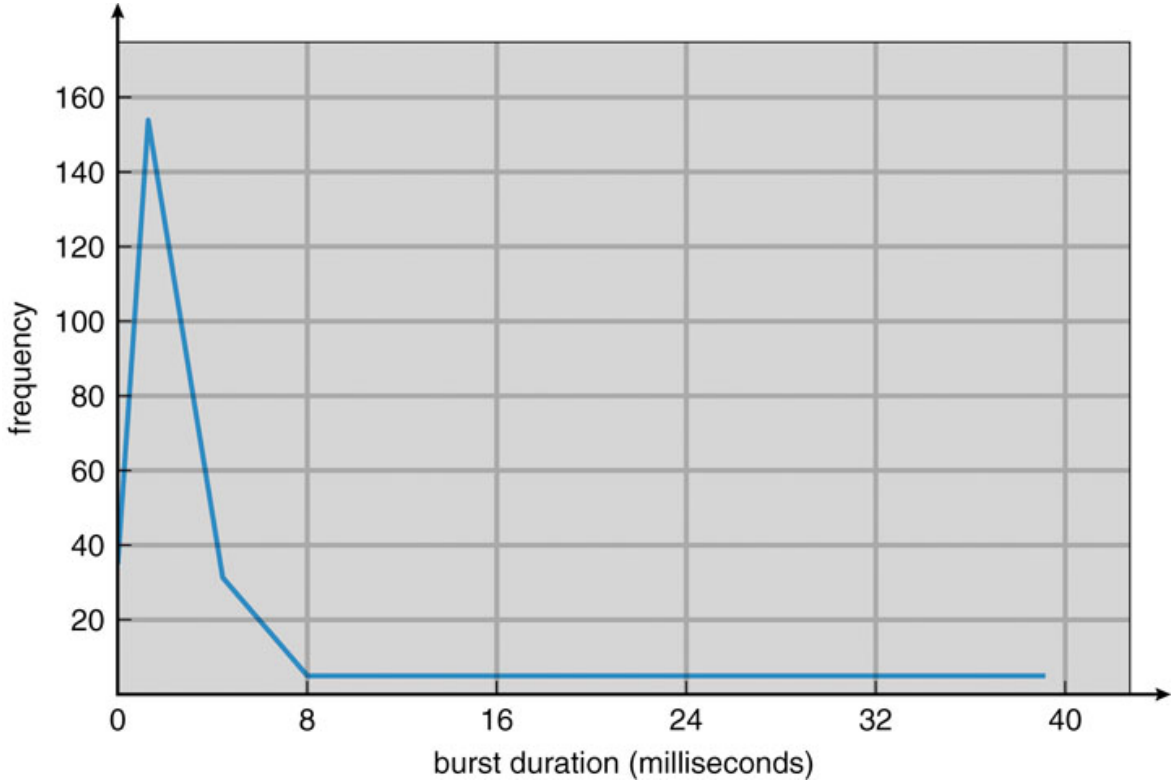
Basic Concepts – Typical Program Behavior

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



CPU Scheduling: Part 1

- Histogram of CPU-burst Times



CPU Scheduling: Part 1

CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- 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
- Scheduling according to conditions (1) and (4) is **nonpreemptive**, or **cooperative**
- All other scheduling is preemptive

CPU Scheduling: Part 1

CPU Scheduler (Contd…)

- **Non-preemptive** (cooperative) scheduling: once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state
  - MS Windows 3.x; Mac OS prior version X
  - The only method on some hardware platforms
- **Preemptive** scheduling incurs cost associated with:
  - Access to shared data
  - More complicated design of the kernel

CPU Scheduling: Part 1

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency**: time it takes for the dispatcher to stop one process and start another running

CPU Scheduling: Part 1

Scheduling Criteria

- **CPU utilization:** keep the CPU as busy as possible: 40% - 90%
- **Throughput:** no. of processes that complete their execution per time unit
- **Turnaround time:** amount of time to execute a particular process
- **Waiting time:** amount of time a process has been waiting in the ready queue = completion time - arrival time - execution time
- **Response time** (for time-sharing environment): amount of time it takes from when a job was submitted until the first response is produced, not output (excludes speed of output device)

These criteria may conflict with each other: it may not be possible to "optimize" all criteria simultaneously

CPU Scheduling: Part 1

Optimization Criteria

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

CPU Scheduling: Part 1

CPU Scheduling Algorithms (Six types of process scheduling algorithms are:)

- First Come First Serve (FCFS)
- Shortest-Job-First (SJF) Scheduling
- Shortest Remaining Time
- Priority Scheduling
- Round Robin Scheduling
- Multilevel Queue Scheduling

CPU Scheduling: Part 1

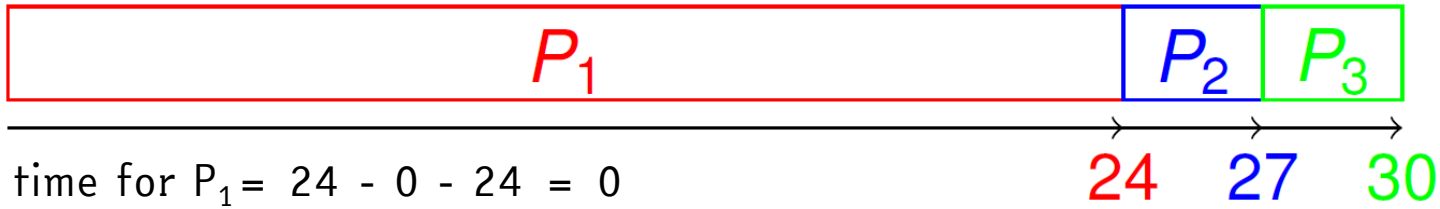
CPU Scheduling Algorithm (FCFC – First Come First Serve)

- First Come First Serve (FCFS) is an operating system scheduling algorithm that automatically executes queued requests and processes in order of their arrival.
- It is the easiest and simplest CPU scheduling algorithm. In this type of algorithm, processes which requests the CPU first get the CPU allocation first. This is managed with a FIFO queue.
- As the process enters the ready queue, its PCB (Process Control Block) is linked with the tail of the queue and, when the CPU becomes free, it should be assigned to the process at the beginning of the queue.

CPU Scheduling: Part 1

CPU Scheduling Algorithm (FCFC – First Come First Serve)

- Suppose we have three processes, P_1, P_2, P_3 in the ready state with predicted CPU burst times 24, 3, 3, respectively.
- Suppose that the processes are run in the order P_1, P_2, P_3
- The Gantt Chart for this schedule is:

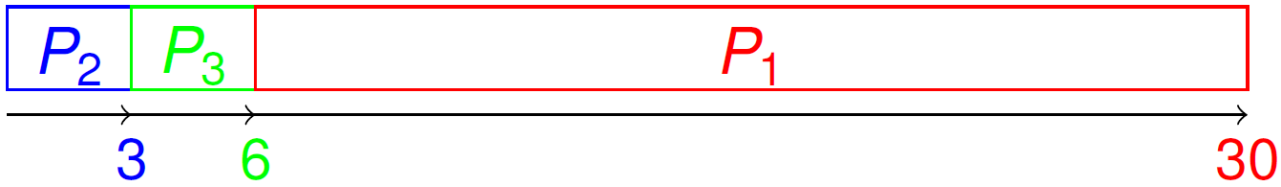


- Waiting time for P_1 = 24 - 0 - 24 = 0
- P_2 = 27 - 0 - 3 = 24 ; and for P_3 = 30 - 0 - 3 = 27
- Average waiting time is (0 + 24 + 27)/3 = 17

CPU Scheduling: Part 1

CPU Scheduling Algorithm (FCFC – First Come First Serve)

- Suppose that the processes are run in the order P_2, P_3, P_1 instead
- The Gantt Chart for this schedule is:



- Waiting time for P_1 = 30 - 0 - 24 = 6; P_2 = 3 - 0 - 3 = 0; and for P_3 = 6 - 0 - 3 = 3
- Average waiting time is (6 + 0 + 3)/3 = 3
- Much better than previous case
- FCFS is non-preemptive leading to…
- …Convoy effect: short processes build up behind long process

CPU Scheduling: Part 1

CPU Scheduling Algorithm (SJF/SRTF – Shortest Job/Remaining Time First)

- Associate with each process its estimated execution time
- Use these lengths to schedule the process with the shortest time
- Two schemes:
- Non-Preemptive: give CPU to job with smallest execution time
- Preemptive: if a new process arrives with estimated execution time less than remaining time of currently executing process, preempt. This scheme is known as the Shortest-Remaining-Time-First (SRTF)
- SJF is optimal: gives minimum average waiting time for a given set of processes

CPU Scheduling: Part 1

CPU Scheduling Algorithm (SJF – Shortest Job First)

Process	Arrival Time	Execution Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

Process	Start Time	End Time
P_1	0	7
P_3	7	8
P_2	8	12
P_4	12	16

- Waiting Time = Start Time – Arrival Time
- Average Waiting time: $(0+6+3+7)/4 = 4$

CPU Scheduling: Part 1

CPU Scheduling Algorithm (SRTF – Shortest Remaining Time First)

= Preemptive SJF

Process	Arrival Time	Execution Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

Process	Start Time	Preempt Time
P_1	0	2
P_2	2	4
P_3	4	5
P_2	5	7
P_4	7	11
P_1	11	16

- Average Waiting time: (9 + 1 + 0 +2)/4 = 3


UNIVERSITY OF
LIMERICK
OLLSCOIL LUIMNIGH


CSIS
Department of
Computer Science
& Information Systems

Thank You

UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH

CSIS Department of Computer Science & Information Systems

CS4023: Operating Systems

Sahir Sharma
University Teacher of Computer Science & Programming
CS2-035, Computer Science Building, University of Limerick
sahir.sharma@ul.ie
Autumn 2021-22

Week 07, Lecture 1

CPU Scheduling: Part 1

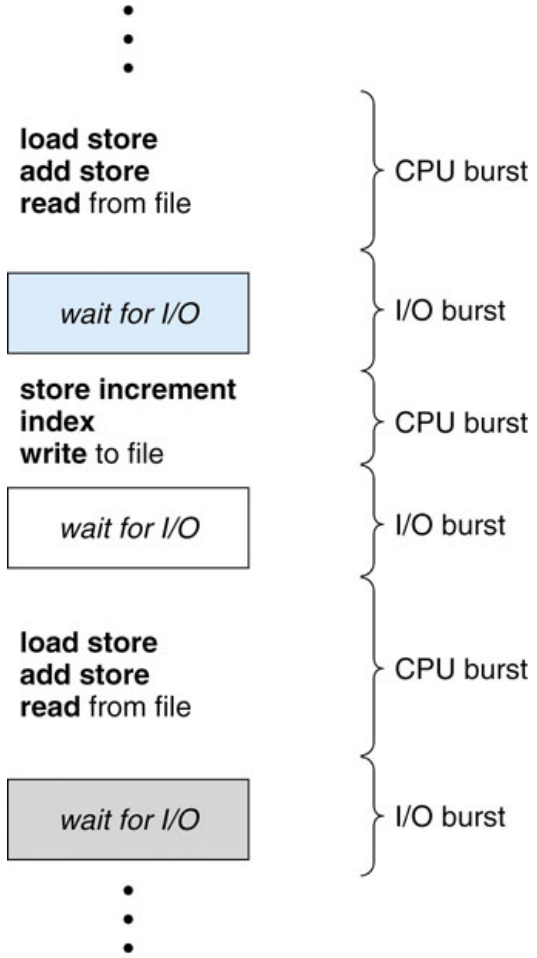
Introduction

- In computer, scheduling is the action of assigning resources to perform tasks.
- The resources may be processors, network links or expansion cards.
- The tasks may be threads, processes or data flows.
- The scheduling activity is carried out by a process called scheduler.

CPU Scheduling: Part 1

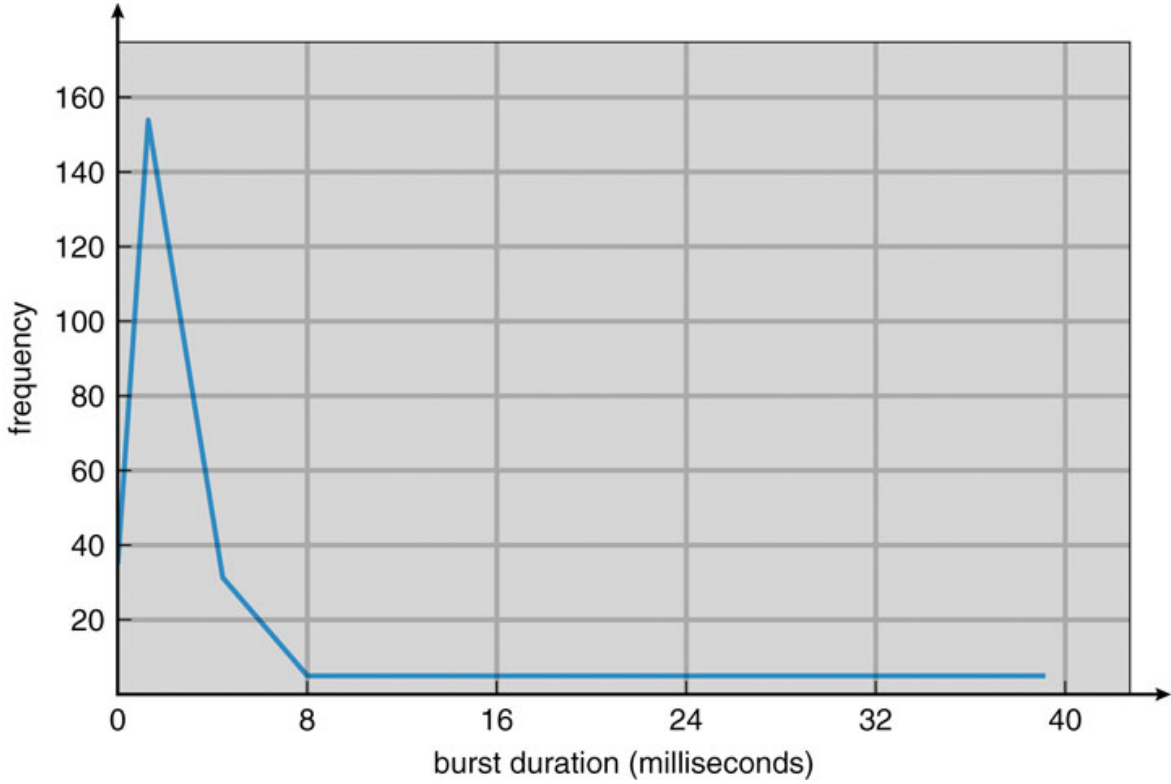
Basic Concepts – Typical Program Behavior

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



CPU Scheduling: Part 1

- Histogram of CPU-burst Times



CPU Scheduling: Part 1

CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- 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
- Scheduling according to conditions (1) and (4) is **nonpreemptive**, or **cooperative**
- All other scheduling is preemptive

CPU Scheduling: Part 1

CPU Scheduler (Contd…)

- **Non-preemptive** (cooperative) scheduling: once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state
  - MS Windows 3.x; Mac OS prior version X
  - The only method on some hardware platforms
- **Preemptive** scheduling incurs cost associated with:
  - Access to shared data
  - More complicated design of the kernel

CPU Scheduling: Part 1

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency**: time it takes for the dispatcher to stop one process and start another running

CPU Scheduling: Part 1

Scheduling Criteria

- **CPU utilization:** keep the CPU as busy as possible: 40% - 90%
- **Throughput:** no. of processes that complete their execution per time unit
- **Turnaround time:** amount of time to execute a particular process
- **Waiting time:** amount of time a process has been waiting in the ready queue = completion time - arrival time - execution time
- **Response time** (for time-sharing environment): amount of time it takes from when a job was submitted until the first response is produced, not output (excludes speed of output device)

These criteria may conflict with each other: it may not be possible to "optimize" all criteria simultaneously

CPU Scheduling: Part 1

Optimization Criteria

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

CPU Scheduling: Part 1

CPU Scheduling Algorithms (Six types of process scheduling algorithms are:)

- First Come First Serve (FCFS)
- Shortest-Job-First (SJF) Scheduling
- Shortest Remaining Time
- Priority Scheduling
- Round Robin Scheduling
- Multilevel Queue Scheduling

CPU Scheduling: Part 1

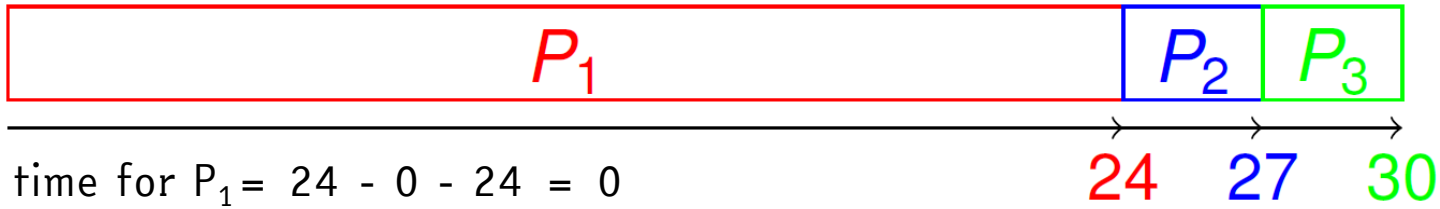
CPU Scheduling Algorithm (FCFC – First Come First Serve)

- First Come First Serve (FCFS) is an operating system scheduling algorithm that automatically executes queued requests and processes in order of their arrival.
- It is the easiest and simplest CPU scheduling algorithm. In this type of algorithm, processes which requests the CPU first get the CPU allocation first. This is managed with a FIFO queue.
- As the process enters the ready queue, its PCB (Process Control Block) is linked with the tail of the queue and, when the CPU becomes free, it should be assigned to the process at the beginning of the queue.

CPU Scheduling: Part 1

CPU Scheduling Algorithm (FCFC – First Come First Serve)

- Suppose we have three processes, P_1, P_2, P_3 in the ready state with predicted CPU burst times 24, 3, 3, respectively.
- Suppose that the processes are run in the order P_1, P_2, P_3
- The Gantt Chart for this schedule is:

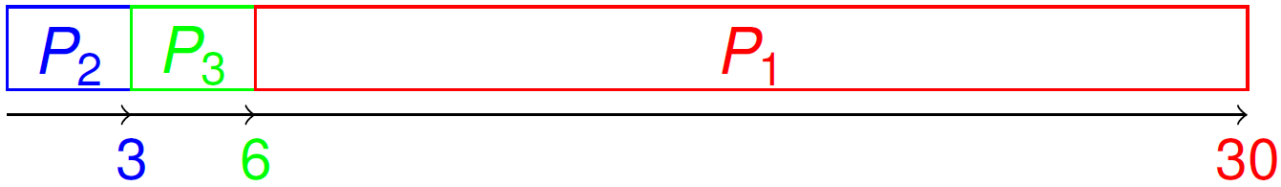


- Waiting time for P_1 = 24 - 0 - 24 = 0
- P_2 = 27 - 0 - 3 = 24 ; and for P_3 = 30 - 0 - 3 = 27
- Average waiting time is (0 + 24 + 27)/3 = 17

CPU Scheduling: Part 1

CPU Scheduling Algorithm (FCFC – First Come First Serve)

- Suppose that the processes are run in the order P_2, P_3, P_1 instead
- The Gantt Chart for this schedule is:



- Waiting time for P_1 = 30 - 0 - 24 = 6; P_2 = 3 - 0 - 3 = 0; and for P_3 = 6 - 0 - 3 = 3
- Average waiting time is (6 + 0 + 3)/3 = 3
- Much better than previous case
- FCFS is non-preemptive leading to…
- …Convoy effect: short processes build up behind long process

CPU Scheduling: Part 1

CPU Scheduling Algorithm (SJF/SRTF – Shortest Job/Remaining Time First)

- Associate with each process its estimated execution time
- Use these lengths to schedule the process with the shortest time
- Two schemes:
- Non-Preemptive: give CPU to job with smallest execution time
- Preemptive: if a new process arrives with estimated execution time less than remaining time of currently executing process, preempt. This scheme is known as the Shortest-Remaining-Time-First (SRTF)
- SJF is optimal: gives minimum average waiting time for a given set of processes

CPU Scheduling: Part 1

CPU Scheduling Algorithm (SJF – Shortest Job First)

Process	Arrival Time	Execution Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

Process	Start Time	End Time
P_1	0	7
P_3	7	8
P_2	8	12
P_4	12	16

- Waiting Time = Start Time – Arrival Time
- Average Waiting time: $(0+6+3+7)/4 = 4$

CPU Scheduling: Part 1

CPU Scheduling Algorithm (SRTF – Shortest Remaining Time First)

= Preemptive SJF

Process	Arrival Time	Execution Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

Process	Start Time	Preempt Time
P_1	0	2
P_2	2	4
P_3	4	5
P_2	5	7
P_4	7	11
P_1	11	16

- Average Waiting time: (9 + 1 + 0 +2)/4 = 3


UNIVERSITY OF
LIMERICK
OLLSCOIL LUIMNIGH


CSIS
Department of
Computer Science
& Information Systems

Thank You

UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH

CSIS Department of Computer Science & Information Systems

CS4023: Operating Systems

Sahir Sharma
University Teacher of Computer Science & Programming
CS2-035, Computer Science Building, University of Limerick
sahir.sharma@ul.ie
Autumn 2021-22

Week 07, Lecture 2

CPU Scheduling: Part 2

What we discussed

- What is CPU Scheduling? Or Process Scheduling
- Various Algorithms for Process scheduling in brief
- In Detail
  - FCFS
  - SJF
  - SRTF

CPU Scheduling: Part 2

Determining the Length of the Next CPU Burst

- We cannot accurately know the execution time of a job
- As a proxy for execution time, we **use length of next CPU burst**
- Can only **estimate** the length, however
- Can be done by using the length of previous CPU bursts, using **exponential averaging**
  - τ_n: predicted value for nth CPU burst
  - t_n: actual length of nth CPU burst
  - τ_{n+1}: predicted value for next CPU burst
  - $\alpha, 0 \leq \alpha \leq 1$, the **weighting** constant
  - Define

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

CPU Scheduling: Part 2

Exponential Averaging

- We cannot accurately know the execution time of a job
- As a proxy for execution time, we **use length of next CPU burst**
- Can only **estimate** the length, however
- Can be done by using the length of previous CPU bursts, using **exponential averaging**
  - τ_n: predicted value for nth CPU burst
  - t_n: actual length of nth CPU burst
  - τ_{n+1}: predicted value for next CPU burst
  - $\alpha, 0 \leq \alpha \leq 1$, the **weighting** constant
  - Define

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

CPU Scheduling: Part 2

Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - recent history doesn't count
- $\alpha = 1$
  - $\tau_{n+1} = t_n$
  - recent history doesn't count
- If we expanded this formula we get:
  - $\tau_{n+1} = \alpha t_n + (1- \alpha)\, \tau_n$
  - $\tau_{n+1} = \alpha t_n + (1- \alpha)\, \alpha t_{n-1} + (1- \alpha)^2\, \tau_{n-1}$
  - $\tau_{n+1} = \alpha t_n + (1- \alpha)\, \alpha t_{n-1} + (1- \alpha)^2\, \alpha t_{n-2} + (1- \alpha)^3\, t_{n-2} \ldots + (1- \alpha)^{n+1}\, \tau_0$
- Since both α and $1 - \alpha$ are less than or equal to 1, each successive term has less weight than its predecessor: Exponential Averaging

CPU Scheduling: Part 2

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (the smaller integer, the higher the priority)
  - Preemptive
  - Non-preemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- **Starvation**: A potential problem where low priority processes may never execute
- Solution: **Aging** – as time progresses increase the priority of the process

CPU Scheduling: Part 2

Round Robin (RR) Scheduling

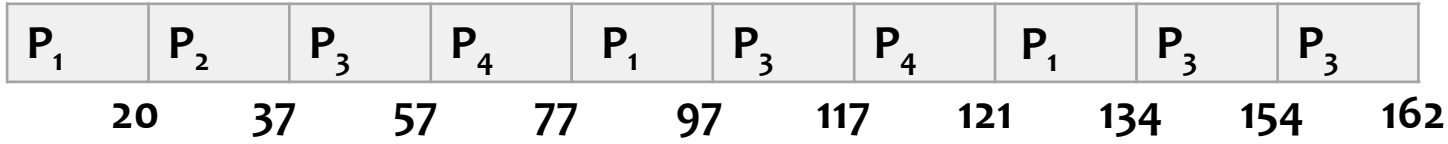
- Each process gets a small unit of CPU time (time quantum), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue
- If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units at once
- No process waits more than (n - 1)q time units
- Performance
  - q large: First-in-first-out (FIFO) queue
  - q small: need to keep q >> T, context switch time, otherwise overhead is too high

CPU Scheduling: Part 2

Process	Burst Time
P_1	53 33 13
P_2	17
P_3	68 48 8 28
P_4	24 4

Round Robin (RR) with Time Quantum of 20

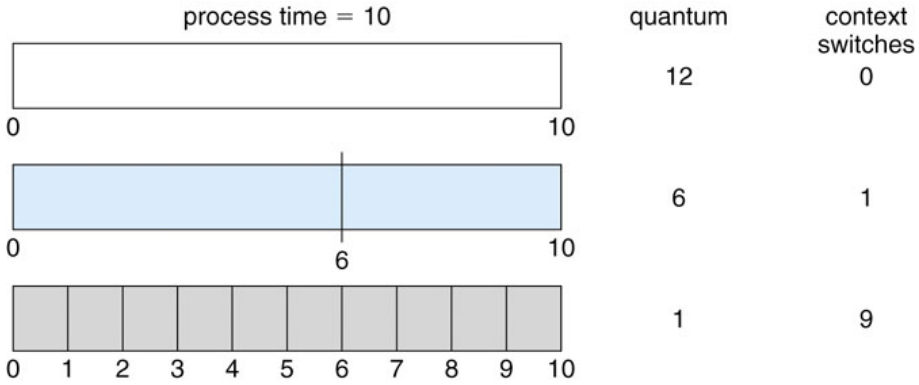
- Suppose the following jobs are on a queue, being processed in the order P_1, P_2, P_3, P_4. (Note that P2 and P4 could finish earlier if queue was reordered into SJF)
- The Gantt Chart is:



Typically, higher average turnaround than SJF, but better **response**.

CPU Scheduling: Part 2

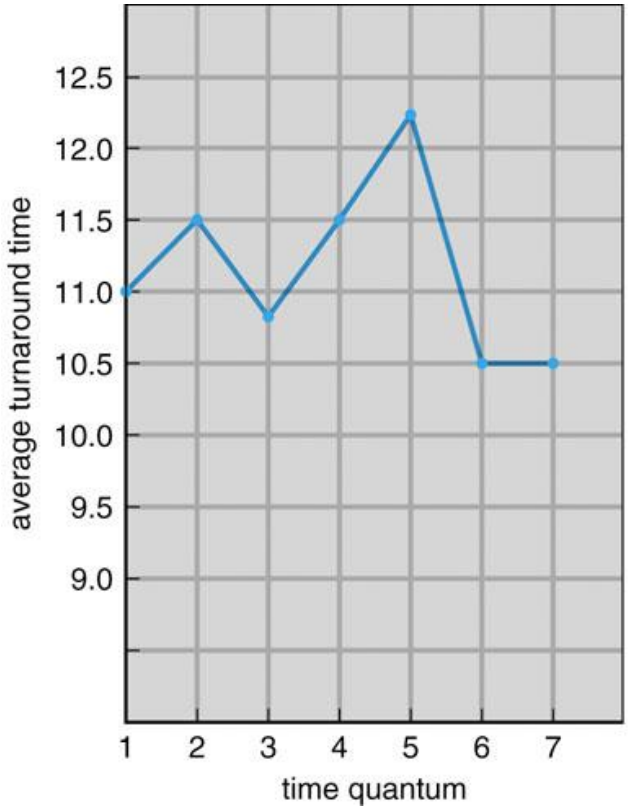
Time Quantum vs Context Switch Time Tradeoff



- The time quantum should be large with respect to the context switch time
- In modern systems time quanta ranging from 10 to 100 milliseconds
- Typically, context-switch requires less than 10 microseconds

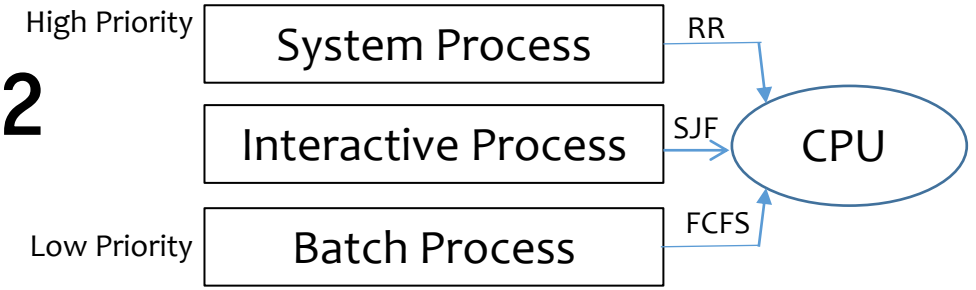
CPU Scheduling: Part 2

- Time Quantum vs Context Switch Time Tradeoff



process	time
P_1	6
P_2	3
P_3	1
P_4	7

CPU Scheduling: Part 2



Multilevel Queue

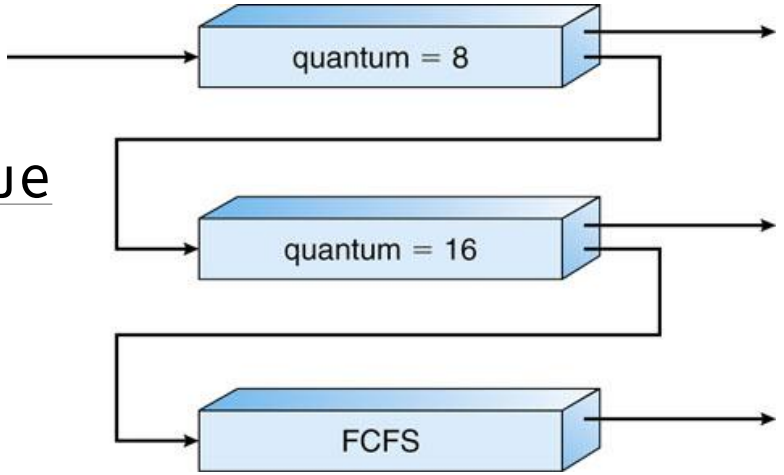
- Ready queue is partitioned into separate queues:
  - Foreground (interactive)
  - Background (batch)
- Each queue has its own scheduling algorithm
  - Foreground e.g.. SRTF, SJF, RR
  - Background e.g.. FCFS, non preemptive algorithms
- Scheduling must be done between the queues
  - Fixed priority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

CPU Scheduling: Part 2

Multilevel Feedback Queue

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

CPU Scheduling: Part 2



Example : Multilevel Feedback Queue

- Three queues:
  - Q0 - time quantum 8 milliseconds
  - Q1 - time quantum 16 milliseconds
  - Q2 - FCFS
- Scheduling
  - A new job enters queue Q0. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q1
  - At Q1 it receives 16 additional milliseconds but is run **only when** Q0 is **empty**. If it still does not complete, it is preempted and moved to queue Q2 where it is processed FCFS but **only when** Q0 and Q1 **are empty**.

CPU Scheduling: Part 2

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- Homogeneous processors within a multiprocessor
- Load sharing
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing

CPU Scheduling: Part 2

Real Time Sharing

- **Hard real-time** systems – required to complete a critical task within a guaranteed amount of time
- **Soft real-time** computing – requires that critical processes receive priority over less fortunate ones

CPU Scheduling: Part 2

Some Videos for Reference

- https://www.youtube.com/watch?v=aWlQYllBZDs


UNIVERSITY OF
LIMERICK
OLLSCOIL LUIMNIGH


CSIS
Department of
Computer Science
& Information Systems

Thank You

UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH

CSIS Department of Computer Science & Information Systems

Some Memes to lighten up...

(They might not be an exact representation of computer science but plays on the stereotypes around)

How to copy files between two different PCS?

1) RIght Click -> Copy
2) Unplug the mouse and connect it to the other PC
3) RIght Click -> Paste
4) Done!

son : dad, why is my sister's name rose

dad : because your mom loves roses

son : okay dad

dad : no problem, stack overflow

[redacted]
1 hr

Why do we need a backend, why not just connect front end to database???

7 6 comments

Like Comment

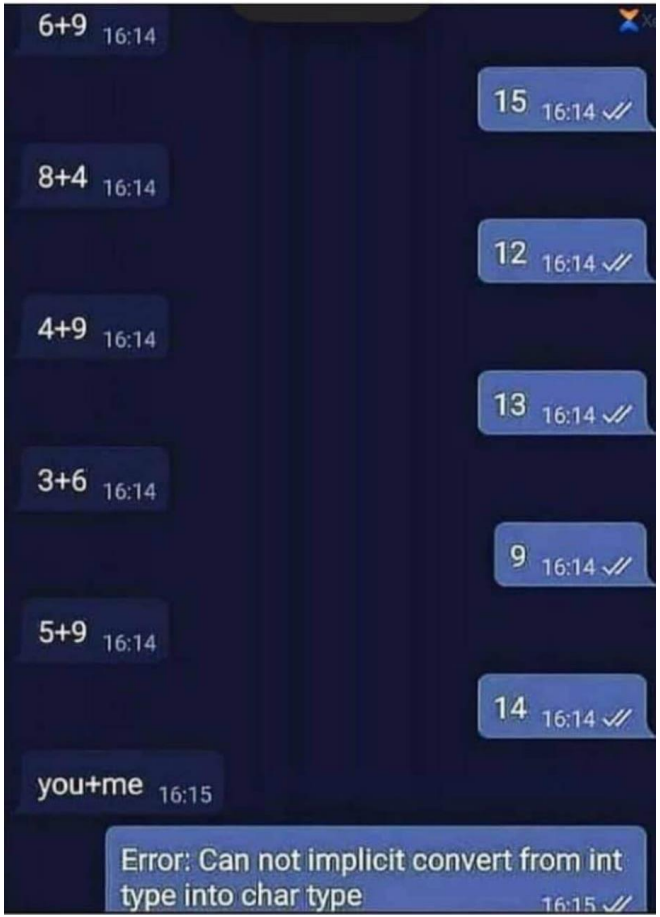
View 2 more comments

[redacted] Yeah! And why do we eat and go to the bathroom while we can throw the food directly in the toilet?

Because stuff needs to get processed

Like · Reply · 22m 6

I dont understand why she responded this way



Thank you doctor. You did your best