# CSI 3131

## Operating Systems

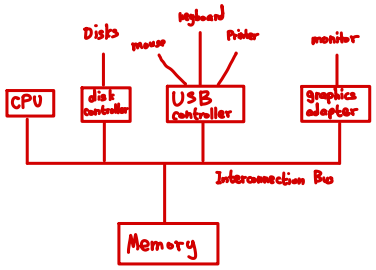
## Study Guide

Summer 2024

# Organization

Computer Hardware Consists of

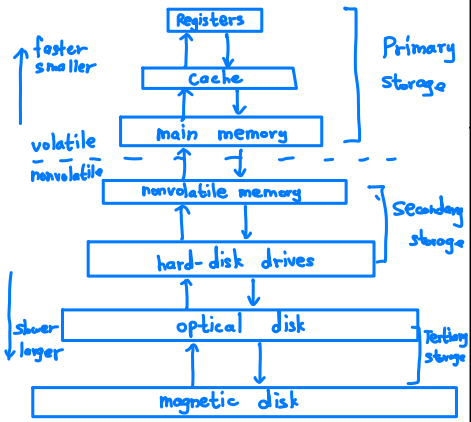
- one or more CPUs
- A main RAM memory for program
- I/O Device controllers
  - Local I/O ports
  - Special-purpose registers for moving data between peripheral devices
  - OS has device driver for each device controller that manages I/O, and provides uniform interface between controller and kernel (OS)
- Interconnection Bus




## Storage Structure

- Register and cache
  - Implemented in static random-access memory
  - Volatile
- Main memory
  - Implemented in dynamic random-access memory
  - Only large storage media that the CPU can access directly
  - Volatile
- Secondary Storage
  - Extension of main memory
  - Large, nonvolatile storage capacity
  - Example: Solid-state disks
    - faster than hard disks
    - nonvolatile
    - Popular for light-weight and no mechanical or moving components.

### The Storage Hierarchy




## Interrupts

- Alert CPU events that require attention
- Interrupts used by OS to handle asynchronous events
- Trap: software interrupt
  - Caused by error or system call
- To start an I/O operation, the device driver loads the appropriate registers in the device controller to control the data flow.
  - Controller starts transfer of data from device to its local buffer
  - Once done, device controller informs device driver it has finished its operation using interrupt
  - Interrupt signaled to CPU requesting to be serviced

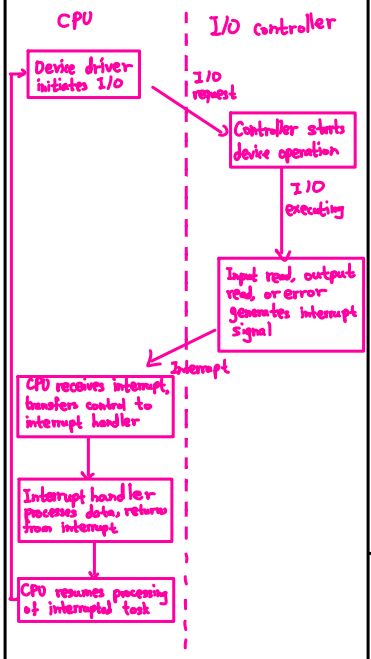
## Interrupt Handling

- CPU stops what it is doing
  - Push CPU register contents to a stack
  - Save address of interrupted instruction
- Then interrupt transfers control to interrupt service routine through interrupt vector
- After finishing execution of ISR, the system must return from interrupt
  - Pull CPU contents back from the stack and resume execution from next instruction of interrupted program
- An operating system is interrupt-driven

## I/O Interrupt Timeline

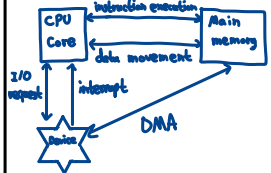
1. User program runs
2. I/O device receives I/O request
3. I/O device transfers data, signals interrupt
4. CPU receives interrupt, halts execution of user program.
5. Execution resumes
6. Repeat

## Interrupt Cycle




## Direct Memory Access (DMA)

- Used for high-speed I/O devices able to transmit information at close to memory speeds
- The form of interrupt-driven I/O described in Section Interrupts is fine for moving small amounts of data but can produce high overhead when used for bulk data movements
- To solve this problem, DMA is used
  - After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from the device and main memory, with no intervention by the CPU.
  - Only one interrupt is generated per block, to tell the device driver that the operation has completed, rather than the one interrupt per byte generated for low-speed devices.
  - While the device controller is performing these operations, the CPU is available to accomplish other work




## Interrupt-Handling Features

- Maskable interrupt can be temporarily disabled. Important when we want to defer interrupt handling during critical processing. Used by device controllers to request service.
- Nonmaskable interrupt cannot be disabled and must be served immediately. Reserved for events such as unrecoverable memory errors, or system reset.
- Interrupt mechanism also implements a system of interrupt priority levels
  - CPU defers handling of low-priority interrupts without masking all interrupts and makes it possible for a high-priority interrupt to preempt the execution of a low-priority interrupt.

## Computer System Architecture

- Single Processor
  - Some small systems still use a single processor
  - They can also have some other special-purpose processors as well
- Multiprocessors
  - Growing in use and importance
  - Also known as parallel systems, tightly-coupled systems
  - Advantages include:
    - Increased throughput
    - Economy of scale
    - Increased reliability
  - Two Types
    - Asymmetric Multiprocessing: each processor is assigned a specific task
    - Symmetric Multiprocessing: each processor performs all tasks
- Clustered Systems
  - Loosely coupled multiple CPUs
  - Usually used to provide high-availability service

## What Operating Systems Do?

- OS is a program that acts as an intermediary between a user of a computer and the hardware to
  - abstract and manage a computer's hardware resources
  - Use the computer hardware in an efficient manner
  - Provides a basis and control the execution of programs
- The OS is a resource allocator
  - Resources include CPU time, memory space, storage space, I/O devices, etc.
  - Hardware resources are shared between programs. The OS decides how to allocate them to specific programs and users in an efficient and fair way.
- The OS is a control program
  - Manages execution of user programs to prevent errors and improper use of computer.

## Computer System Structure

- Can be divided into 4 components: hardware, operating system, application programs, and user.
- Hardware: provides basic computing resources for the system.
- Operating System: controls hardware and coordinates its use among the various application programs (processes) for the various users.
- Application Programs: Define the ways in which these resources are used to solve users' computing problems
- User: People, machines, other computers

## Kernel

- The kernel is the fundamental part of an OS. It is a piece of software responsible for providing computer programs with secure access to the hardware.
- Many processes can run in parallel, and access to the limited hardware should be shared, the kernel decides when and how long a process should be able to make use of a piece of hardware.
- Kernels implement a set of abstractions over the hardware. These abstractions hide the hardware complexity and provide a clean and uniform interface to the underlying hardware, which makes application programs more portable and easier to develop.

## System Programs

- Some services are provided outside of the kernel by system programs.
- System programs are anything not in the kernel but shipped with the OS.
- System programs are loaded into memory at boot time to become system daemons.
  - System daemons run the entire time the kernel is running.
- System programs are also called utility programs that provides some services like:
  - File management
  - Status information
  - Editing files
  - Programming language support
  - Program loading and execution
  - Communications
  - Application programs

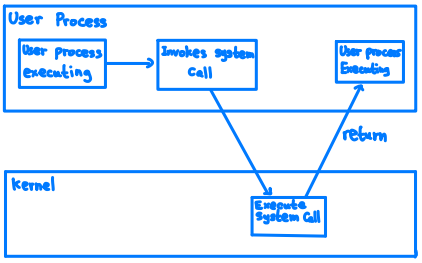
## Dual Mode

- OS is interrupt driven
  - Hardware interrupt by external devices
  - Software interrupt (trap): caused either by an error or by a system call
- Some operations will have to be done only by reliable programs
- A solution is an operation in dual mode
- Dual mode allows the OS to protect itself and other system components
  - User mode (unprotected)
  - Kernel mode (protected)
- Mode bit: a bit added to hardware to indicate the current mode
  - 0 for kernel
  - 1 for user
- Some instructions are designated as privileged (may cause harm if badly implemented), only executable in kernel mode.
  - If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the OS
- System call to OS changes mode bit to kernel mode, return from call sets it to user mode.

## Timer

- A timer is used to prevent processes from getting stuck and never return control to the OS.
  - Before turning over control to the user, the OS ensures that the timer is set to interrupt.
  - A timer can be set to interrupt the computer after a specified period.
    - Keep a counter that is decremented by the physical clock. When it hits zero an interrupt is raised for an action.
    - If the timer interrupts, then the control transfers automatically to the OS, which may treat the interrupt as a fatal error or may give the program some more time.
    - Clearly, instructions that modify the content of the timer are privileged

## Transition from User to Kernel mode




## Resource Management

- Process Management
  - A process is an instance of program in execution.
  - OS manage resources needed by a process to accomplish its task
  - OS monitor and manage process activities like creation, termination, communication between Processes, process scheduling
- Main Memory Management
  - OS determines which process and when it occupies the main memory to optimize CPU usage and the computer's response to users
- Storage and File-System Management
  - The OS provides a uniform, logical view of the information (files) stored in secondary memory
- I/O subsystem
  - One of the purposes of the OS is to hide the peculiarities of specific hardware devices from the user

## Virtualization

- Allows applications of different OS's to run on the same machine at the same time
- Allows different OS's to run on the same machine at the same time
- Allows OS's to run as applications with other OS's
- Emulation used when source CPU type is different from target type
- Virtual Machine Manager (VMM): runs the guest OS's, manages their resource use, and protects each guest from the others
- A user of a Virtual Machine (VM) can switch among the various OS's in the same way a user can switch among the various processes running concurrently in a single operating system
- Advantages of Virtualization
  - Each VM can use a different OS
  - In theory, we can build VMs on VMs
  - Complete protection, because VMs are isolated from each other
  - A new OS can be developed on a VM without disturbing others

## Operating System Services

- An OS provides an environment for the execution of programs
  - Makes certain services to programs and to the users of those programs
  - Differ from one OS to another
- Program Execution
  - System must be able to: load, execute, and end the program
- I/O Operations
  - A running program may require I/O, which may involve a file or an I/O device.
  - For efficiency and protection, this must be done in kernel mode
- File-System Manipulation
  - Programs may need to do some operations over files or directories:
    - read, write, delete, search, create
  - OS can manage ownership, permissions or access to files or directories
- Communications
  - One process may need to exchange information with another process
    - Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems
    - Can be implemented via shared memory or message passing
- Resource Allocation
  - Multiple processes running at the same time and to ensure the efficient operation, the limited hardware resources must be allocated to each of them by the OS
  - The OS manages many different types of resources
    - Some resources managed by a specific allocation code
    - Others may be managed by a general request and release code
- Accounting (Logging)
  - Statistics or record keeping for different purposes such as:
    - how much and what kinds of computer resources a program uses
    - accounting (so that users can be billed)
  - Usage statistics may be a valuable tool for system administrators who wish to reconfigure the system to improve computing services
- Error Detection
  - One of the OS tasks is to detect and correct errors constantly
  - OS should take appropriate action to ensure correct and consistent computing
- Protection and Security
  - To control the use of information stored in a multiuser or networked computer system
    - Manage access authentications of different users
    - Prevent unauthorized access to the system
    - Prevent unauthorized access to resources
- User Interface
  - Most commonly
    - Graphical user interface: mouse and a keyboard
    - Touch-screen interface: "fingers" as in mobile systems
    - Command-line interface: a keyboard for typing in commands for more professional use

## Command-Line Interface

- The command interpreter is a program that reads commands typed by the user
  - Command interpreters also known as shells
  - The main function of the command interpreter is to get and execute a user command
- The commands can be implemented in two general ways
  - Direct execution by the interpreter
  - By programming "custom-made" commands through system programs

## Graphical User Interface (GUI)

- Users employ a mouse-based window-and-menu system characterized by a desktop metaphor
- Icons on the screen represent programs, file, directories, and system functions.
- GUI first appeared in the early 1970s at Xerox PARC research facility
- GUI became more widespread with the advent of Apple MacIntosh computers in the 1980s

## User and OS Interface on Major OS's

- Major OS systems include CLI and GUI interfaces
  - Microsoft Windows is GUI with a CLI
  - MacOS provides both an Aqua GUI and a CLI
  - KDE and GNOME GUI desktops run on Linux, Solaris, and various UNIX systems and are available under open-source licenses

## Choice of Interface

- The choice of whether to use a command-line or GUI interface is mostly one of personal preference
  - GUI and Touch-Screen interface are most of the population choice... user friendly interfaces
  - System administrators and power users who have deep knowledge of a system frequently use the command-line interface.
  - In most cases, only a subset of system functions is available via the GUI, the most advanced tasks are command-line only!
  - CLI is more efficient, giving faster access to the activities need to perform.
  - CLI usually make repetitive tasks easier, in part because they have their own programmability
  - A set of command-line steps can be recorded into a file, and that file can be run like a program.

## System Calls

- System calls provide an interface to the services made available by an OS
- Generally available as C functions written in C and C++ (sometimes assembly)
- System calls are generated by user programs
- At the end of the function, the mode change back to user mode and values are returned to the calling program

## Application Programming Interface (API)

- Application developers design programs according to an API
- The API specifies a set of C functions that are available to an application programmer
- Three of the most common APIs
  - Windows API for Windows systems
  - POSIX API for POSIX-based system
  - Java API for Java Virtual Machine
- Behind the scenes, the functions that make up an API typically invoke the actual system calls on behalf of the application programmer.
  - Actual system calls abstracted from application programmer by the API function
- Application programmer have the choice to use system calls directly
- Advantages of API over system calls
  - Improve program portability: the program to compile and run on any system that supports the same API
  - API hides complicated details
    - Caller doesn't need to know about how system call is implemented or what it does during execution
    - Rather, caller need only obey the API and understand what the OS will do as a result of execution of system call.

## Handling a System Call

- User application invokes the system call through the system call interface
- System uses CPU-specific method to transition from user mode to kernel mode and pass the system call information to the kernel
- Each system call has a number that is used as an index in the system call table to invoke the appropriate code for the system call
- The system call executes and returns once it completes (or indicates an error).
- The system transitions from kernel mode back to user mode

## Types of System Calls

- System calls can be grouped into six major categories
  - Process Control
    - Create, terminate, load, execute, etc.
  - File Management
    - Create, delete, open, close, read, write, etc.
  - Device Management
    - Request, release, read, write, etc.
  - Information Management
    - Set/get time, date, and get/set, process, file, or device attributes
  - Communication
    - Create/delete communication connection, send/receive messages, etc.
  - Protection
    - Get/set file permissions

## Parameter Passing in System Calls

- Three general methods are used to pass parameters to the OS
  - Pass the parameters in registers: number of parameters is very limited
  - Store the parameters to pass in a block, or table, in memory, and the address of the block is passed as a parameter in a register or as a pointer to the bloc in C programming
  - Parameters can be placed, or pushed, onto a stack by the program and popped off the stack by the OS
- Some other operating systems prefer the block or stack method because those approaches do not limit the number or length of parameters being passed

## OS Design and Implementation

- At the highest level, the design of the system will be affected by the choice of hardware and the type of OS
- The requirements: user goals and system goals
  - A user wants a system to be convenient to use, easy to learn and to use, reliable, safe, and fast. These specifications are not particularly useful in the system design, since there is no general agreement on how to achieve them
  - The system should be easy to design, implement, and maintain; and it should be flexible, reliable, error free, and efficient. Again, these requirements are weak and may be interpreted in various ways
- Implementation
  - Today, especially in C and C++, with small sections in assembly. Java may be used for UIs
  - CPU registers and peripheral ports can only be accessed in assembly.

## OS Structure

- Internal structure of different OS's can vary widely
- Monolithic Structure
  - Hardware resources, simple functionality
  - Simplest structure, kernel is one program file
  - Advantage:
    - OS in a single address space provides very efficient performance
  - Disadvantage:
    - Tightly coupled kernel
    - Difficult to implement and extend
- Layered Structure
  - OS broken down into a number of layers
  - Bottom layer is hardware, highest layer is UI
  - Advantages
    - More resources and functionalities, simple to debug
    - Lower layer abstract operation implementation details from higher layer

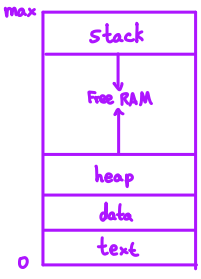
- The kernel is a loosely coupled system
- Disadvantages:
  - Appropriately defining the functionality of each layer
  - Poor overall performance
- Micro-kernel structure
  - Shrinking the kernel to minimum essential services
  - Moving all nonessential components from the kernel and implementing them as user-level programs
  - Advantages:
    - More resources and functionalities
    - Easily extendable and portable, security and reliability
  - Disadvantages:
    - Performance of microkernels can suffer due to increased system-function overhead
- Modular Structure
  - Flexibility and efficiency, services are implemented or removed dynamically
- Hybrid Systems
  - In practice very few operating systems adopt a single, strictly defined structure. Instead, they combine different structures, resulting in hybrid systems that address performance, security, and usability issues.

## Processes

- A process is a program loaded into memory and executing (program in execution)
- To accomplish its task, a process will need certain resources such as
  - CPU time
  - Memory
  - Files
  - I/O devices
- A process is the unit of work in most systems
- Systems consist of a collection of processes
  - OS processes execute system code
  - User processes execute user code
- Programs and processes are different
  - A program stored on a HD is a passive entity
  - A process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources

## Process in Memory

- Text section: executable code
- Data section: global variables
- Heap section: memory dynamically allocated during program run time
- Stack section: temporary data storage when invoking functions




## Process States

- New: The process is being created
- Running: Instructions of process being executed
- Waiting: Process waiting for some event to occur
- Ready: Process waiting to be assigned to CPU
- Terminated: Process finished execution or forced to terminate


## Process Control Block

- In multiprogramming, a process is running on the CPU intermittently
- Whenever a process takes over the CPU, it must resume from the state from where it was interrupted
- So, when a process execution is interrupted, PCB is used as the repository for all the data needed to start, or restart, a process, along with some accounting data.
- A PCB is a per-process data structure containing many pieces of information associated with the process.
- A PCB contains the following:
  - Process State: new, ready, running, waiting, terminated
  - Program Counter: indicates address of next instruction to be executed for this process. Must be saved to allow the process to be continued correctly when it runs again.
  - Other CPU registers: vary depending on architecture.
  - CPU scheduling information: process priority, scheduling queue pointers, other scheduling parameters
  - Memory-management information: value of base and limit registers and page tables, or segment tables, depending on memory system used by the OS
  - Accounting information: Amount of CPU and real time used, time limits, account numbers, job or process numbers
  - I/O status information: This information includes the list of I/O devices allocated to the process a list of open files, and so on.
  - Pointer to next PCB in a linked list

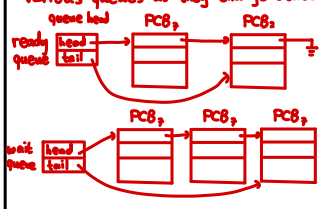
| Process State |
|---|
| Process Number |
| Program Counter |
| Registers |
| Memory limits |
| List of open files |
| . . |

## Process Scheduling

- The objective of multiprogramming is to have some process running at all times so as to maximize CPU utilization
- The objective of time sharing is to switch a CPU core among processes so frequently - users perceive as all programs are running in parallel
- Process scheduler selects a process from a set of available processes for execution on a core
- For a system with a single CPU core, there will never be more than one process running at a time
- Multicore system can run multiple processes at one time. But we still need scheduling as if there are more processes than cores, excess processes will have to reschedule. The number of processes currently in memory is known as the degree of multiprogramming

## Process Scheduling Queues

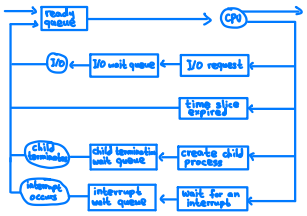
- As processes enter a system, they are put into a ready queue
  - where they are ready and waiting to execute on a CPU's core
  - This queue is generally stored as a linked list
  - Each PCB includes a pointer field that points to the next PCB in the ready queue
- The system includes other queues
- A running process on a CPU core will have to eventually terminate, interrupt, or wait
- Suppose the process makes an I/O request to a device such as a disk
  - A disk runs significantly than processes, the process will have to wait for the I/O to become available
  - Processes that are waiting for a certain event to occur are placed in the wait queue
- Processes can migrate between the various queues as they change states




- When a process migrate between the various queues, they do not physically move into memory
- The corresponding PCB's pointers are modified to point to a different queue

## CPU Scheduling (short-term)

- The role of the CPU scheduler is to select from among the processes that are in the ready queue and allocate a CPU core to one of them
  - CPU scheduling decides which of the ready processes should run next on the CPU
- The CPU scheduler must select a new process for the CPU frequently
  - An I/O-bound process may execute for only a few milliseconds before waiting for an I/O request
  - A CPU-bound process will require a CPU core for longer durations, the scheduler is likely to remove the CPU from it and schedule another process to run
  - Therefore, the CPU scheduler executes at least once every 100 milliseconds, or much more frequently
- There are 2 types of queues present in the scheduling process




- The diagram above is the queueing diagram for scheduling
  - Circles represent resources that serve the queues
  - Arrows indicate flow of processes
- When a process terminates, it is removed from all queues and has its PCB and resources deallocated

## Swapping (Intermediate Scheduling)

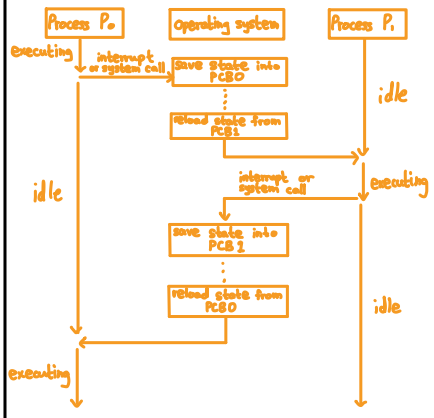
- Some operating systems have an intermediate form of scheduling, known as swapping
- Can be called if degree of multiprogramming needs to decrease by removing a process from memory to a disk storage
- Later, the process can be reintroduced into memory, to resume its execution (swapping)
- A swapping is
  - A process can be "swapped out" from memory to disk and its current status is saved
  - Later "swapped in" from disk back to memory, where its status is restored
- Swapping is typically only necessary when memory has been overcommitted and must be freed up

## Long-term Scheduling

- Long-term scheduling decides when a process should enter the ready state and start competing for the CPU
  - Processes are created at unpredictable times and enter the system with the process state new
  - Long-term scheduling decides which new processes are moved to the ready queue to compete for the CPU
  - Processes are also subject to long-term scheduling when suspended and later reactivated by the OS.
- Long-term scheduling controls the degree of multiprogramming to achieve optimal performance
- Long-term scheduling takes place less frequently than short-term scheduling
  - Occurs only at process creation and when the OS suspends the process

## Context Switch

- Context Switch is when the CPU moves from executing one process, say process 0, to another process, say process 1
- Context Switch occurs when the CPU is interrupted during the execution of process 0
- Save the running process (0) current context to PCB0 (state save)
  - Essentially suspending the process
  - Context: The state of its execution, including the contents of registers, its PC, and its memory content
- Finds and accesses the PCB1 of process 1, that was previously saved
- Perform context restore from PCB1
  - Essentially the opposite of context save to run the previously suspended process 1
- Then when process 1 is done, the above steps can be repeated to suspend process 1 and to run process 0 again




## Process Creation

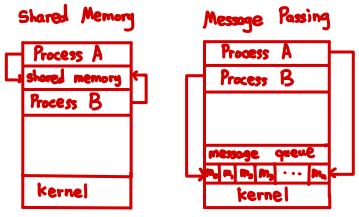
- During execution, a process (Parent) can create one child process
  - In turn, a child process can also have children processes - forming a tree of processes
- Generally, a process is identified and managed via a process identifier (pid), which is typically an integer number
- Usually, several properties can be specified at child creation time:
  - Resource sharing options
    - Share all resources
    - Share a subset of parent's resources
    - No sharing
  - When a parent creates a new process, two possibilities for execution exist:
    - Parent and child execute concurrently
    - Parent waits until children terminate
  - There are two address-space possibilities for the new process:
    - Child duplicate of parent
    - Child has a new program loaded into it
- We may need a child process because we need to fulfil a need for a program to perform more than one function simultaneously
- When a parent process creates a child process, it passes some data or instructions to the child process. The child process is an independent entity that can execute a different program or perform a different task from the parent process
- In UNIX, a new process is created by fork() system call
  - The new process consists of a copy of the address space of the parent process
    - The parent and the child continue execution at the instruction after the fork()
    - fork() returns: 0 if child executing, PID of child if parent executing
  - Parent may call wait() to wait until child terminates
  - exec...) system call, used after a fork() to replace the process' memory with a new program
  - When child process completes parent resumes from call to wait() until it completes
- n fork, $2^n - 1$ child processes

## Process Termination

- Process executes last statement and asks the OS to delete it by exit() system call
  - Return status data from child to parent via wait()
  - Process's resources deallocated by OS
- Abnormal termination can occur because of errors
- Parent may terminate the execution of children processes using the abort() system call
- Some operating systems do not allow a child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated
- The parent process may wait for termination of a child process by using the wait() system call. The call returns status information and the pid of the terminated process
- A child process that has terminated but whose parent has not invoked wait(), is known as a zombie process
  - Zombie processes exist only briefly until the parent calls wait(), then the zombie process identifier and its entry in the process table are released
- If the parent terminates without invoking wait(), the child process is an orphan

## Interprocess Communication (IPC)

- Processes executing concurrently in the OS may be independent or cooperating
  - A process is independent if it does not share data with any other processes executing in the system
  - A process is cooperating if it can affect or be affected by the other processes executing in the system
- Reasons for cooperating processes:
  - Information sharing
  - Communication speedup
  - Modularity
- Cooperating processes requires interprocess communication (IPC)
- Two models of IPC:
  - Shared memory: processes can read/write data by a shared memory region
  - Message passing: cooperating processes communicate by message exchange
- Cooperating processes need IPC mechanism to exchange data and to synchronize their actions
  - Shared memory: typically faster than message passing - initial system call only to establish the shared memory segment, afterward it can be accessed in user mode
  - Message passing: all communications use system calls
    - Easier to implement




## IPC - Shared Memory

- Normally the OS prevent one process from accessing another process's memory
- Shared memory requires that two or more processes agree to remove this restriction
  - One process makes a system call to create a shared memory region
  - Other processes make system call to attach a shared memory region to their address space
  - The data and the location are determined by these processes and are not under the operating system's control
- Major issue is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory
  - The processes are responsible for ensuring that they are not writing to the same location simultaneously
  - Producer-consumer problem, common paradigm for cooperating processes
  - Producer produces information and the consumer consumes it

## IPC - Message Passing

- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space
- Communication link is needed
  - Logical implementation: several methods, and notably the send()/receive() operations:
    - Direct or indirect communication
    - Synchronous or asynchronous communication
    - Automatic or explicit buffering
  - Hardware implementation: bus, or network

## Message Passing - Direct Communication

- Under direct communication, processes must name each other explicitly:
  - send (P, message) - send message to process P
  - receive (Q, message) - receive 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
- This scheme exhibits symmetry in addressing
  - Both sender and receiver processes must name the other to communicate
- A variant of Direct communication employs asymmetry in addressing
  - Only sender names recipient, recipient not required to name sender
- Primitives are defined as follows:
  - send (P, message) - send a message to process P
  - receive (id, message) - receive message from any process
  - variable id set to name of the process with which communication has taken place
- Limitations:
  - Process definitions limited modularity of the resulting process definitions
  - Changing the id of a process may require all other process definitions to update them with the new id

## Message Passing - Indirect Communication

- Messages are sent to and received from mailboxes or ports
  - A mailbox can be viewed abstractly as an object into which messages can be placed or removed by processes
  - Each mailbox has unique identification
  - Two processes can communicate only if they have a shared mailbox
- Primitives are defined as follows:
  - send (A, message) - send message to mailbox A
  - receive (A, message) - receive message from mailbox A
- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Mailbox sharing
  - Processes P1, P2, P3 all share mailbox A
  - P1 sends to A while P2 and P3 receive
  - Which process will receive?
- Depends on the chosen method:
  - Allow a link to be associated with two processes at most
  - Allow at most one process at a time to execute a receive() operation
  - Allow the system to select arbitrarily the receiver. The system may identify the receiver to the sender

## Message Passing - Synchronization

- Message passing may be either blocking or non-blocking
- Blocking - also known as synchronous message passing
  - Blocking send. The sending process is blocked until the message is received by the receiving process or by the mailbox
  - Blocking receive. The receiver blocks until a message is available
- Non-blocking is considered asynchronous
  - Non-blocking send - sender sends the message and return before the delivery of message
  - Non-blocking receive - receiver receives a valid message, or a null
- Different combinations possible
  - If both are blocking: rendezvous

## Message Passing - Buffering

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue.

- Zero capacity:
  - no queue exist
  - no messages waiting
  - sender must block until the recipient receives the message
- Bounded capacity:
  - queue has finite length n
  - at most n messages can reside in it
  - if the queue is not full when a new message is sent, the message is placed in the queue, and the sender can continue execution without waiting
  - if link is full, the sender must block until space is available in the queue
- Unbounded capacity:
  - The queue's length is potentially infinite, thus, any number of messages can wait in it. The sender never blocks

## Pipes

- Acts as a conduit allowing two processes to communicate
- Were one of the first IPC mechanisms in early UNIX systems
- They typically provide one of the simpler ways for processes to communicate with one another
- In implementing a pipe, four issues must be considered:
  - Does the pipe allow bidirectional communication, or is communication unidirectional?
  - If bidirectional communication is allowed, then is it half-duplex or full duplex?
  - Must a relationship exist between the communicating processes?
  - Can the pipes be used over a network, or must be on the same machine?
- Two common types:
  - Ordinary pipes - cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created
  - Named pipes - can be accessed without a parent-child relationship

## Ordinary Pipes

- Ordinary pipes allow communication in standard producer-consumer style
  - Producer writes to one end (write-end)
  - Consumer reads from other end (read-end)
  - As a result, ordinary pipes are unidirectional
  - For two-way, two pipes must be used
  - Require parent-child relationship between communicating processes
- Ordinary pipes on UNIX is constructed by the function: Pipe (int fd[])
- On Windows they are called anonymous pipes
- The function pipe(int fd[]) creates a pipe that is accessed through the int fd[] file descriptors:
  - fd[0] read end
  - fd[1] write end
- Pipes can be accessed using ordinary read() and write() system calls
- Typically, a parent process creates a pipe and uses it to communicate with a child process that it creates via fork()
- The child inherits the pipe from its parent process
- Relationship of the file descriptors in the fd array to the parent and child processes
- Limitations
  - Exist only while the processes are communicating with one another, once the processes terminate they cease to exist
  - Require a parent-child relationship between the communicating processes. This means they can be used only for communication between processes on the same machine
- Each pipe is implemented using a fixed-size buffer

## Named Pipes

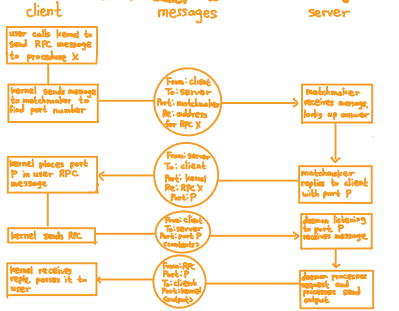
- Provides a much powerful communication tools than the ordinary pipes
- Communication can be bidirectional, and no parent-child relationship is needed
- Several processes can use the named pipe for communication
- Continue to exist after communicating processes have finished
- Provided on both UNIX and Windows systems
- Refened as FIFOs on UNIX systems
  - Created with mkfifo() system call
  - Once created, they appear as typical files in the file system
  - Can be manipulated with ordinary open(), read(), write(), and close() system calls
  - Continue to exist until it is explicitly deleted from file system
  - Bidirectional but only in half-duplex
  - Communicating processes must reside on the same machine
- Windows
  - Full-duplex communication is allowed
  - Communicating processes may reside on either the same or different machines
  - Only byte-oriented data may be transmitted across a UNIX FIFO, whereas Windows systems allow either byte- or message-oriented data
  - Named pipes created with CreateNamedPipe()
  - A client can connect to a named pipe using ConnectNamedPipe()
  - Communication over the named pipe can be accomplished using the ReadFile() and WriteFile() functions

## Sockets

- A socket is defined as an endpoint for communication
  - A pair of processes communicating over a network employs a pair of sockets
  - A socket is identified by an IP address concatenated with a port number
  - The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
  - Communication link corresponds to a pair of sockets
  - Port 80 reserved for HTTP server
- If the client process initiates a connection request, it is assigned a port by its host computer
  - This port has some arbitrary number greater than 1024
  - This connection will consist of a pair of sockets: on host X and on the web server
  - The packets travelling between the hosts are delivered to the appropriate process based on the destination port number
  - All connections must be unique

## Remote Procedure Calls (RPC)

- A message-based communication scheme to provide remote service
- It is similar in many respects to IPC mechanisms, but processes are executing on separate systems
- The parameters and return values need to be somehow transferred between the computers
- Support for locating the server/procedure needed
- Stubs: side proxies implementing the needed communication
  - The client-side stub locates the server and marshals the parameters
  - Then the stub transmits a message to the server using message passing
  - The server-side stub receives this message and invokes the procedure on the server
  - If necessary, return values are passed back to the client using the same technique
- Client can find the port numbers on the server by:
  - An RPC call has a fixed port number associated with it. Once a program is compiled, the server cannot change the port number of the requested service
  - Dynamically by OS provided matchmaker: Function that matches a caller to a service being called

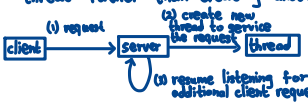


## Threads

- A thread is a basic unit of CPU utilization
  - It comprises a thread ID, a program counter, a register set, and a stack
  - Threads belonging to the same process share code section, data section, and other operating-system resources
- A process with multiple threads of control, can perform more than one task at a time

## Multithread Server Architecture

- A busy web server receive thousands of client service request
  - Each client request is concurrently serviced
  - A single-thread process, can service only one client at a time!
  - One solution is to create one process per received service request
  - It is generally more efficient to use one process that contains multiple threads
  - A multithreaded serve allows to create a separate thread rather than creating another process




## Benefits of Threads

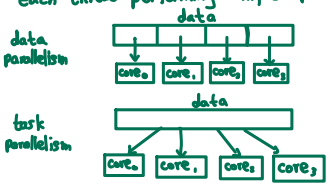
- Responsiveness: may allow continued execution if part of a process is blocked, especially for user interfaces
- Resource Sharing: threads share resources of process, easier than shared memory or message passing between processes
- Economy: cheaper than process creation, thread switching lower overhead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

## Concurrency vs. Parallelism

- Multicore or multiprocessor is to place multiple computing cores on single processing chip where each core appears as a separate CPU to the OS
  - Multithreaded programming improves concurrency in multicores
- Parallelism implies a system can perform more than one task simultaneously
- Concurrency supports more than one task making progress

## Types of parallelism

- Data parallelism: distributes subsets of the same data across multiple cores, Same operation on each
- Type parallelism: distributing threads across cores, each thread performing unique operation




## Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- S is serial portion
- N processing cores

$$\text{speedup} \leq \frac{1}{S + \frac{(1-S)}{N}}$$

## User Threads and Kernel Threads

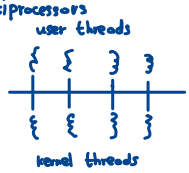
- User threads - management done by user-level thread library
  - Supported above the kernel and are managed without kernel support
- Kernel threads - Supported by the kernel
  - Supported and managed directly by the OS
  - Virtually all current OS's

## Many-to-One Model

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

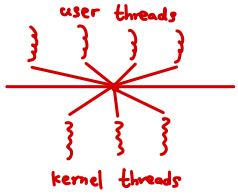

## One-to-One Model

- Each user-level thread maps to kernel thread
  - Creating a user-level thread creates a kernel thread
  - A 1-2-1 model drawback: each user thread requires a kernel thread. Large number of kernel threads can burden performance
  - So, number of threads per process sometimes restricted due to overhead
  - More concurrency than many-to-one by allowing another thread to run when a thread makes a blocking system call
  - Allows multiple threads to run in parallel on multiprocessors



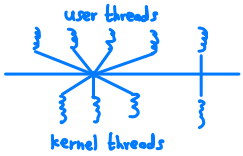

## Many-to-Many Model

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




## Two-Level Model

- Variation of many-to-many model
  - Still multiplexes many user-level threads to a smaller or equal number of kernel threads
  - but also allow a user-level thread to be bound to a kernel thread




## Thread Libraries

- Thread library provides the programmer with Application Programming Interface for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space -user controlled
    - Invoking a function in the library results in a local function call in user space and not a system call
    - Efficient as thread operations do not request system calls
    - Disadvantage: the kernel cannot schedule a thread of a process. Blocking a thread implies blocking a process
  - kernel-level library supported by OS
    - Code and data structures for the library exist in kernel space
    - The kernel is able to directly manage the states of the threads
    - Invoking a function in the API for the library results in a system call to the kernel

## POSIX Threads

- POSIX Pthreads, provided as either a user-level or a kernel-level library
- Portable Operating System Interface (POSIX) standard API for thread creation and synchronization
- Common in UNIX operating systems
- Must include the pthreads.h header file when using the Pthreads API
- Data sharing for Pthreads threading, is done by the mean of global variable (in C)
  - variables declared outside of any function are visible and can be accessed/shared by all threads belonging to the same process

## Windows Threads

- Windows thread library is a kernel-level library available on Windows systems
- The technique for creating threads is similar to the Pthreads technique in several ways
- Must include the windows.h header file when using the Windows API
- Data sharing for Windows threading is done by the mean of global variable (in C)
  - Variable declared outside of any function are visible and can be accessed/shared by all threads belonging to the same process

## Java Threads

- Java thread API allows threads to be created and managed directly in Java programs
  - JVM runs on top of a host OS
    - generally implemented using a thread library available on the host system
      - Windows: Windows API
      - UNIX: Pthreads
- Java has no equivalent notion of global data
  - access to shared data must be explicitly arranged between threads
- Two techniques for explicitly creating threads in a Java program
  - Method 1: Create a new class that is derived from the Thread class and override its run() method
  - Method 2: Define a class that implements the Runnable interface, that defines a single abstract method with the signature public void run()
- Thread creation involves:
  - creating a Thread object and passing it an instance of a class that implements Runnable
  - followed by invoking the start() method on the Thread object
- Invoking the start() method for the new Thread object does two things
  - It allocates memory and initializes a new Thread in the JVM
  - It calls the run() method, making the thread eligible to be run by the JVM
- join() method can be used to force the parent thread to wait for the children threads to finish before proceeding

## Java Executor Framework

- Beginning with version 1.5 and its API, Java introduced several new concurrency features
  - provide developers with much greater control over thread creation and communication
  - available in java.util.concurrent
  - Rather than explicitly creating Thread objects, thread creation is instead organized around the Executor interface:

```
public interface Executor{
    void execute(Runnable command);
}
```

  - Classes implementing this interface must define the execute() method, which is passed a Runnable object
  - This means using the Executor rather than creating a separate Thread object and invoking its start() method
- The Executor framework is based on the producer-consumer model
  - tasks implementing the Runnable interface are produced
  - and the threads that execute these tasks consume them
  - advantage of this approach, divides thread creation from execution and provides a mechanism for communication between concurrent tasks
- java.util.concurrent package defines the Callable interface
  - Callable behaves similarly to Runnable except that a result can be returned
  - Results returned from Callable tasks are known as Future objects that can be retrieved from the get() method defined in the Future interface

## Callable vs. Runnable in Java

- Callable and Runnable interfaces in Java are both used to represent tasks that can be executed by another thread
  - They encapsulate functions that are supposed to be completed by another thread
  - Are widely used in multithreaded programming to perform tasks concurrently
- The main advantage of using Callable over Runnable is that Callable tasks can return a result and throw exceptions, while Runnable tasks cannot

## Asynchronous and Synchronous Threading

- Asynchronous threading: once the parent creates a child thread, the parent resumes its execution
  - The parent and child execute concurrently and independently of one another
  - There is typically little data sharing between parent and child
  - Commonly used for designing responsive user interfaces
- Synchronous threading: parent thread must wait for all of its children to terminate before it resumes
  - Threads created by the parent perform work concurrently, but the parent cannot continue until this work has been completed
  - Once each thread has finished its work, it terminates and joins with its parent
  - Only after all of the children have joined can the parent resume execution
  - Typically, synchronous threading involves significant data sharing among threads

## Implict Thread

- Growing in popularity as number of threads increase program correctness more difficult with explicit thread
- Creation and management of threads done by compilers and run-time libraries rather than programmers

## Thread Pools

- A number of threads created at process startup and placed in a pool, where they sit and wait for work
- Server submit a request to the thread pool and resumes waiting for additional requests
- If there is an available thread in the pool, it is awakened, and the request is serviced immediately
- If the pool contains no available thread, the task is queued until one becomes free
- Once a thread completes its service, it returns to the pool and awaits more work (it is not terminated)
- Advantages:
  - Faster: Usually slightly faster to service a request with an existing thread than create a new thread
  - Bounded: Allows the number of threads in the application(s) to be bound to the size of the pool
  - Separating task to be performing from mechanics of creating task allows different strategies for running task

## Semantics of fork() and exec()

- The semantics of the fork() and exec() system calls change in a multithreaded program
- fork() sometimes have two versions
  - Duplicates all threads
  - Duplicates only the thread that invoked the fork()
- exec() replaces the running process address spaces and this includes all threads
- Which of the two versions fork() to use?
  - If depends on the application:
    - if exec() is called immediately after forking, then duplicating all threads is unecessary
    - if exec() is not called after forking, the separate process should duplicate all threads

## Thread Cancellation

- Terminating a thread before it has finished
- Two general cancellation approaches:
  - Asynchronous cancellation terminates the target thread immediately
  - Potential problems:
    - OS may not reclaim all the cancelled thread resources
    - May leave some share data in an inconsistent state

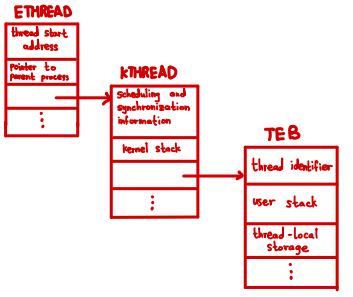
- Deferred cancellation allow the target thread to periodically check if it should be cancelled
  - One thread indicates that a target thread is to be cancelled
  - but cancellation occurs only after the target thread has checked a flag to determine wheter or not it should be cancelled
  - The thread can only perform this check at a point at which it can be cancelled safely

## Thread-Local Storage

- Threads belonging to a process share the data of the process
- But in some circumstances, each thread might need its own copy of certain data
- Thread-local storage (TLS) allows each thread to have its own copy of data
  - Useful when you do not have control over the thread creation process
  - Another example, in a transaction-processing system, transaction in a separate thread - each thread should have its unique transaction identifier in their TLS
- TLS vs. local variables
  - Local variables visible only during single function invocation (local scope)
  - TLS visible across function invocations
  - TLS unique to each thread

## Windows Threads

- One-to-one model
- General components of a thread:
  - Thread ID uniquely identifying the thread
  - Register set representing status of processor
  - Program counter
  - User stack, employed when thread is running in user mode
  - Kernel stack, employed when thread is running in kernel mode
  - Private data storage memory area
- Register set, stacks, and private storage area are known as the context of the thread
- The primary data structures of a thread include
  - ETHREAD (executive thread block): includes pointer to process to which thread belongs and to kTHREAD, in kernel space
  - KTHREAD (kernel thread block): scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
  - TEB (thread environment block): thread id, user-mode stack, thread-local storage, in user space




## 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 planned task
  - Flags control behaviour

| 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 |

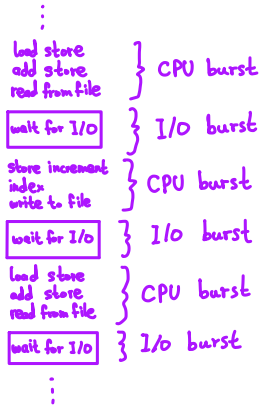
- If no flags is set when clone() is invoked, no sharing takes place, resulting in functionality similar to fork()

## CPU Scheduling Motivation

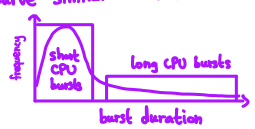
- The motivation behind multiprogramming is to have some process running at all times, to maximize CPU utilization.
  - When for example if one process wait for an I/O, another process can execute on the core.
  - This pattern continues - every time one process has to wait, another process can take over the CPU.
  - On a multicore system, this concept of keeping the CPU busy is extended to all processing cores on the system.
  - Scheduling of this kind is a fundamental OS function.
  - Almost all computer resources are scheduled before use
- The CPU is one of the primary computer resources. Thus, CPU scheduling is central to OS design.

## CPU-I/O Burst Cycle

- CPU scheduling success depends on an observed process property:
  - Process execution consists of a cycle of CPU execution and I/O wait- Phases alternate between these two states.
  - Process execution begins with a CPU burst, followed by an I/O burst, then followed by CPU burst, and so on.
  - Eventually, the final CPU burst ends with a system request to terminate execution.
  - CPU burst distribution of main concern.




- The durations of CPU bursts have been measured extensively:
  - They vary greatly from process to process and from computer to computer
  - They tend to have a frequency curve similar to this:




  - The curve is generally characterized as exponential or hyperexponential, with a large number of short CPU bursts and a small number of long CPU bursts.
  - An I/O-bound program typically has many short CPU bursts
  - A CPU-bound program might have a few long CPU bursts
  - This distribution can be important when implementing a CPU-scheduling algorithm.

## CPU Scheduler

- Short-term scheduler selects from among the processes in the ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state (I/O request)
  2. Switches from running to ready state (interrupt occurs)
  3. Switches from waiting to ready (I/O operation completed)
  4. Terminates
- Scheduling under 1 and 4 is nonpreemptive (cooperative)
  - Running process keeps the CPU until it releases it
- Scheduling under 2 and 3 is preemptive
  - Can result in race conditions when data are shared by several processes
    - When two processes that share data. While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state.

## Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - Context switching from one process to another
  - 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
  - Should be as fast as possible, since it is invoked during every context switch

## Scheduling Criteria

- Different algorithms may favor one class of processes over another
- A particular algorithm may favor a partiular situation
- We must consider the properties and criteria of the various algorithms:
  - Throughput: a measure of work - the number of processes that are completed per time unit
    - For long processes, this rate may be one process over several seconds
    - For short transactions, it may be tens of processes per second
  - Turnaround time: For a particular process, how long it takes to execute that process
    - Is the interval from the time of submission of a process to the time of completion
    - Is the sum of the periods spent waiting in the ready queue, executing on the CPU, and doing I/O
  - Waiting time: Measure only the amount of time that a process spends waiting in the ready queue
    - Is the sum of the periods spent waiting in the ready queue
  - Response time: time from the submission of a request until the first response is produced
    - Is the time it takes to start responding, not the time it takes to output the response
- Ideally, we want:
  - maximize CPU utilization, throughput
  - minimize turnaround, waiting, and response time

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

- Implementation is easily managed with a FIFO queue
  - When a process enters the ready queue, its PCB is linked onto the tail of the queue
  - The process at the head of the queue will run next on the CPU
  - Each process is associated with a burst time
  - When a process runs, it will run for the duration of its burst
  - FCFS scheduling algorithm is nonpreemptive
    - A running process releases the CPU, either
      - by terminating
      - by requesting I/O
    - The FCFS algorithm is thus particularly troublesome for interactive systems, where it is important that each process get a share of the CPU at regular intervals
    - It would be disastrous to allow one process to keep the CPU for an extended period
- Assume one CPU-bound process holds the CPU, and many I/O-bound processes
  - During this time, all the other I/O-bound processes finish their I/O and move into the ready queue, waiting for the CPU
  - While the processes wait in the ready queue, the I/O devices are idle
  - Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device
  - All the I/O-bound processes, which have short CPU bursts, execute quickly and move back to the I/O queues
  - At this point, the CPU sits idle
  - The CPU bound process will then move back to the ready queue and be allocated the CPU
  - Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done
  - There is a convoy effect as all the other processes wait for the one big process to get off the CPU
  - Convoy effect results in lower CPU and device utilization

### Discussion of FCFS

- Simple and easy to program
- Costs little time to make scheduling decision
- Waiting time can be quite long
- Convoy effect affects CPU utilization

## Shortest-Job-First (SJF) Scheduling

- Scheduling depends on the length of the next CPU burst of a process, rather than total length
  - Associate with each process the length of its next CPU burst
  - The shortest process burst will run first
- Optimal in principle from the point of view of average waiting time
- The difficulty is knowing in advance the length of the process's next CPU burst
- Can be either preemptive or nonpreemptive
- Choice arises when a new process arrives at the ready queue while a previous process is still executing
- The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process
  - nonpreemptive - will allow the current running process to finish its CPU burst
  - preemptive - will preempt the currently executing process
    - Sometimes called Shortest-Remaining-Time-First
    - Optimal: minimum average waiting time for a given set of processes

## Discussion of SJF

- Low average waiting time
- Can have long response time because if there is a steady stream of short CPU bursts, the longer bursts will not be scheduled (suffer from starvation)
  - Starvation: occurs when a process waits for an indefinite time to get the resource it requires. Always overtaken by other processes

## SJF - CPU Next Burst Prediction

- Can only estimate the length of a burst - should be like last ones
  - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts for this process, using exponential averaging
  - $t_n$ = actual length of $n^{th}$ CPU burst
  - $\tau_{n+1}$ = predicted value for next CPU burst
  - $\alpha, 0 \leq \alpha \leq 1$
  - $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$
- Commonly, $\alpha = \frac{1}{2}$

## Round Robin Scheduling (RR)

- Each process gets a small unit of CPU time slice called time quantum q, usually 10-100 milliseconds
- After q time has elapsed (before any other interruption), the process is preempted (by the q timer interrupt)
  - The RR scheduling algorithm is thus preemptive
- Interrupted process is added to the end of the ready queue
- The performance of the RR algorithm depends heavily on the size of the time quantum
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once
- No process waits more than $(n-1)q$ time units

Timer interrupts every quantum to schedule next process

- q large $\Rightarrow$ FIFO
- q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high
- Typically, higher average turnaround than SJF, but better response time

- If the time quantum q is extremely large, RR degenerates to FCFS policy
- If the time quantum is extremely small, the RR approach can result in a large number of context switches
- Thus, we want the time quantum to be large with respect to the context-switch time, but not too large

## Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the ready process with the highest priority (with preemption or with no preemption)

Equal-priority processes are scheduled in FCFS order

- Problem: Starvation - low priority processes may never execute
- Solution: Aging - as time progress increase the priority of the process
- Some systems use lower numbers for lower priority; others use it for high priority. This difference can lead to confusion
- In this text, we assume that low numbers represent high priority

## Multilevel Queue Scheduling

- Partition processes into several separate queues based on priority or process type
  - foreground (interactive processes): high-priority
  - background (batch processes): low-priority
- Entering the system, a process permanently assigned to a given queue
  - low scheduling overhead, but it is inflexible
- Each queue has its own scheduling algorithm
- How must scheduling must be done from among the queues:
  - Fixed priority preemptive scheduling
    - i.e. the interactive queue may have absolute priority over the background queue
    - serve all foreground processes first then move to serve background processes
    - Possibility of background processes starvation
  - Time slice
    - Each queue gets a certain portion of the CPU time which it can then schedule among its various processes
    - Not necessarily optimal

## Multilevel Feedback Queue Algorithm

- A process can move between the various queues
  - An aging process can be promoted to a higher priority queue to prevent starvation
  - A "gourmand" process using too much CPU time can be moved down to a lower priority queue
  - Short CPU bursts processes are left in the higher-priority queues
- Multilevel-feedback queue scheduler defined by the following parameters:
  - number of queues
  - Scheduling algorithms for each queue
  - method used to determine when to promote 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
- Adaptable to specific systems
- But most complex to implement

## Multiple-Processor Scheduling

- Scheduling multiple CPUs is more complex than a single CPU
- Therefore, OS must deal with efficient load sharing and time management among the available multiple CPUs
- Result: Parallel execution
  - Load sharing - when multiple processes may run in parallel
  - OS overheads shared among different cores
- All processors are identical - Homogeneous - in terms of their functionality
- Two approaches to multiprocessor scheduling

Asymmetric multiprocessing - only one core accesses the system data structures, reducing the need for data sharing

- Symmetric multiprocessing - each CPU is self-scheduling
  - The standard approach in modern computing OS's to support multiprocessors
  - Two possible strategies for organizing the scheduling
    - (a) All processes in common ready queue
    - (b) Each has its own private queue of ready processes

## Symmetric Multiprocessing Complication

- Load Balancing - it is important to keep the workload balanced among all CPUs to fully benefits of having more than one CPU
  - Required when each CPU has its own ready queue
  - Push migration: a task runs periodically to evenly distribute load among CPUs (moving task from overloaded to an idle or a less-busy CPU)
  - Pull migration: an idle CPU pulls a waiting task from a busy CPU
  - Linux OS supports both Push and Pull techniques

Can counteract the benefits of processor affinity, since it moves processes from one CPU to another

- Processor affinity - process has affinity for processor on which it is currently running
  - When a process runs on a physical CPU, cache memory is updated with content of the process
  - If the process is moved to another CPU, benefits of caching is lost
  - SMP systems try to keep processes running on the same physical CPU, known as processor affinity

## Scheduling Algorithm Evaluation

- First, identify the selection criteria
- Next, evaluate the algorithms under consideration using various methods:
  - Deterministic modelling
  - Queuing models
  - Simulations

## Deterministic Modeling

- A type of analytic evaluation
- Takes a particular predetermined workload and defines the performance of each algorithm for that workload

## Queuing Models

- Analytical method based on probability theory
- Describes the arrival of CPU and I/O bursts with probability theory
  - Commonly exponential, described by mean
  - Computes average throughput, utilization, waiting time, etc
- Computer system described as network of servers, each with queue of waiting processes (queuing-network analysis)
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc.
- Simplified model: OS times ignored

## Little's Formula

- $n$ = average queue length
- $W$ = average waiting time in queue
- $\lambda$ = average arrival rate into queue

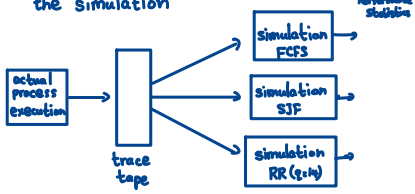
Little's Law: in steady state, processes leaving queue must equal processes arriving, thus:

$$n = \lambda \times W$$

- Valid for any scheduling algorithm and arrival distribution

## Evaluation by Simulations

- Build a simulator and run different scheduling algorithms and evaluate results
- Assign a duration of time to each event
- Run the model for this sequence to get algorithm performance statistics
- How to feed data about CPU and I/O bursts?
  - 2 ways
    1. Analytically: a random number generator that is programmed to generate processes, CPU burst times, arrivals, departures, and so on, according to probability distributions. The distributions can be defined mathematically.
    2. Empirically: feed in data from a real system under study, and record the sequence to trace files, then use this sequence to drive the simulation




## Race Condition

- Concurrent threads often share user data (files or common memory) and resources (cooperative tasks)
- Race condition occurs where several tasks access and manipulate the same data concurrently
- A race condition exists when access to shared data is not controlled, possibly resulting in corrupt data values (outcome of execution depends on the particular order in which the access takes place)

## Critical Section Problem

- When multiple threads execute in parallel, we cannot make assumptions about the execution speed of the threads, nor their interlacing
- May be different each time the program is run
- Critical section is part of a task whose execution must not interlace with other tasks' critical sections
  - Once a task enters, it must be allowed to complete this section without allowing other tasks to manipulate on the same data
- Critical-section problem is to design a protocol that the threads can use to synchronize their activity so as to cooperatively share data
  - The result of their actions does not depend on the interlacing order of execution
- The execution of critical sections must be mutually exclusive: at any time, only one thread can execute a critical section for a given data
- This can be achieved by placing special instructions at the beginning and end of the critical section
- Once a task enters the critical section, it must complete it as an atomic bloc (no other tasks are allowed to mess with the same data)
- The critical section must be locked to become indivisible (atomic)

Program Structure

```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

## Criteria for Valid Solutions to CSP

1. Mutual Exclusion
   - At any time, at most one thread can be in a critical section
2. Progress
   - Ensures programs will cooperatively determine what task will next enter its critical section
   - A critical section will only be given to a task that is waiting to enter it
   - Whenever a CS becomes available, if there are threads/tasks waiting for it, one of them must be able to enter it
3. Bounded Waiting
   - Limits the amount of time a program will wait before it can enter its critical section

- A thread waiting to enter a critical section will finally be able to enter (alternation and no starvation)
  - No thread can be forever excluded from the critical section because of other threads monopolizing it

## Some Critical Section Problem Solutions

- Software based solutions
  - algorithms that do not use special instructions
  - Peterson's algorithm
- Hardware solutions
  - rely on the existence of certain special instructions
  - test_and_set(), compare_and_swap()
- Higher-level software tools solution
  - Provides certain system calls to the programmer
  - Mutex, Semaphores, monitor
- All solutions are based on the atomic access to central memory: a memory address can only be assigned by one instruction at a time, therefore by one thread at a time
- In general, all solutions are based on the existence of atomic instructions, which function as basic critical sections

## Software Solution 1

- Have $T_i$ tasks and a shared variable turn indicating whose turn it is now
  - Turn is initialized to a valid task number
- $T_i$ may enter the critical section if and only if turn = i
- After exiting the critical section, turn is assigned next i task number, so it gains access to its critical section next
- $T_i$ can be in a busy wait if $T_j$ is in its CS
- Mutual exclusion and bounded waiting satisfied
  - Only one thread can enter the critical section at any moment (Mutual exclusion)
  - Tasks alternate very well and no starvation (bounded waiting)
- But assuming we have 1000 threads and only a handful are active:
  - Each time, before a thread can enter the critical section, it must wait until all the others have had this chance
  - This is a contradiction with the Progress requirement

## Software Solution 2

- Gives the critical section only to threads that want it (active threads)
- However, we cannot allow a task to systematically give itself the critical section
  - This will cause starvation for others
  - Each process that wants to enter must give others a chance before entering
- Use a boolean variable per thread: flag[0] and flag[1]
- $T_i$ signals that it wants to execute its CS by setting flag[i] = true
- But $T_i$ does not enter if another thread also set its flag to enter
- Mutual Exclusion is satisfied
- Progress is not satisfied
  - $T_0$: flag[0] = true
  - $T_1$: flag[1] = true
  - Each thread will wait indefinitely to execute, resulting in a deadlock
- This algorithm allows only active threads to enter the critical section, but it may result in a deadlock when multiple threads want to enter at the same time

## Software Solution 3: Peterson's Solution

- Provides a good algorithmic description of solving the critical-section problem
- Illustrates some of the complexities involved in designing software that addresses the requirements of:
  - mutual exclusion, progress, bounded waiting
- No guarantees that it will work correctly on modern computer architectures
- Restricted to 2 processes/tasks that alternate
- Combines ideas from previous solutions
  - flag[i] = intention to enter
  - turn = whose turn
- Initialization:
  - flag[0] = flag[1] = false
  - turn = i or j
- If we want to run the critical section, it indicated by flag[i] = true
- Put flag[i] = false on exit
- Thread $T_i$ should wait if the other wants to enter and it's the other's turn
  - flag[j] == true and turn == j
- Thread $T_i$ can enter if the other doesn't want to enter or it's their turn
  - flag[j] == false or turn == i
- $T_i$ uses the boolean flag[i] to indicate its desire to enter its CS
- It also uses turn to give $T_j$ the priority to enter its CS
- $T_i$ can enter the critical section only when flag[j] == false or turn == i
- To enter the critical section, a task $T_i$ first sets flag[i] to be true and then sets turn to the value j
  - thereby asserting that if $T_j$ wishes to enter its critical section, it can do so
- If both tasks try to enter at the same time, turn will be set by both tasks roughly the same time:
  - But turn is shared by both, so only the latest of these assignments will hold
  - The eventual value of turn determines which of the two tasks is allowed to enter the critical section first
- Mutual Exclusion is preserved
  - A task enters its critical section only if it is its turn when both want in
- Progress requirement is satisfied
  - A task will enter its critical section if and only if it is its turn and the other task does not want to enter its critical section
  - If both wants to enter, it is not possible as turn can hold only 1 value that allow only 1 task to enter CS

Bounded-waiting requirement is met

- An executing task will set its flag to false before reaching its remainder section, this will unlock the other task from its busy wait and allows it to execute its critical section
- If a solution satisfies the requirements of mutual exclusion and progress, it provides robustness against the thread failure in its non critical section
- If a thread fails in its critical section, it does not send a signal to the other threads, which results in them being blocked
  - Timeout is a solution: a thread that has CS after a certain time is interrupted by the OS
- Peterson's algorithm can be generalized to more than 2 tasks

## Peterson's Solution Possible Failure

- May not work on modern computer
- Processors and/or compilers may reorder read and write operations that have no dependencies
- For a multithread application with shared data, the reordering of instructions may render inconsistent or unexpected results
- This can cause two threads to be active in their critical sections at the same time

$P_0 \rightarrow$ turn = 1 $\longrightarrow$ flag[0] = true $\rightarrow$ CS

$P_1 \longrightarrow$ turn = 0, flag[1] = true $\rightarrow$ CS

## Software Based Solution Discussion

In order for threads with shared variable to succeed, it is necessary that all involved threads use the same coordination algorithm

- Software solutions are difficult to program or understand
  - Threads that desire entry into their CS are busy waiting, which consumes CPU time
  - For long critical sections, it is preferable to completely block threads that must wait, then wake them up when they can enter critical section

## Hardware Support: Memory Barriers

- A multicore system may reorder instructions, leading to unreliable data states
- Memory barriers
  - Force propagate any memory changes to all threads running on different cores
  - Ensures that all load and store operations are completed before additional load and store operations are performed

## Hardware Support: Instructions

- Modern computer systems provide special hardware instructions that:
  - Test and modify the content of a word
  - Automatically swap the contents of two words atomically - as one uninterruptible unit
- We can use these special instructions to solve the critical-selection problem in a relatively simple manner

## test_and_set() instruction

- Executed atomically
- If two test_and_set instructions are executed simultaneously, they will be executed sequentially in some arbitrary order
- Mutual exclusion can be implemented by declaring a boolean variable lock, initialized to false

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

The structure of process $T_i$ is

```
while (true){
  while (test_and_set(&lock));
    /* critical section */
  lock = false;
    /* remainder section */
}
```

Some disadvantages:

- Still using busy waiting
- When $T_i$ leaves its critical section, the selection of $T_j$ to enter its CS is arbitrary: no bounded waiting → starvation is possible

## compare_and_swap() instruction

- Very similar to test_and_set() instruction atomically
- Mechanism is based on the swapping of two words
- Operates on three operands: lock(value), expected, and new_value
- Definition of instruction:

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

Some disadvantages:

- Still using busy waiting
- It does not satisfy the bounded-waiting requirement

- Mutual exclusion can be implemented by declaring a boolean variable lock, initialized to false or 0.
- The first process that invokes compare and swap will set lock to 1
- It will then enter its critical section, because the original value of lock was equal to the expected value of 0
- Subsequent calls to compare and swap will not succeed, because lock now is not equal to the expected value of 0
- When a program exits its critical section, it sets lock back to 0, which allows another process to enter its critical section

```
while (true){
  while(compare_and_swap(&lock,0,1)!=0);
    /* critical section */
  lock = 0;
    /* remainder section */
}
```

- On intel x86 architecture, the assembly language statement on intel x86 is used to implement the compare_and_swap() instruction
- The general form of this instruction appears as

  lock cmpxchg <destination operand>, <source operand>

- To enforce atomic execution, the lock prefix is used to lock the bus while the destination operand is being updated
- Interchanges the contents of source operand and destination operand atomically

## compare_and_swap() with waiting

- An algorithm using compare and swap that satisfies all the critical-section requirements
- The common data structures are:

```
boolean waiting[n]; //initialized to false
int lock; //initialized to 0
```

- Task $T_i$ can enter its critical section only if either:
  - waiting[i] == false
  - key == 0
- The value of key can become 0 only if the compare_and_swap() is executed; all other tasks must wait
- The variable waiting[i] can become false only if another process leaves its critical section
- Only one waiting[i] is set to false, maintaining the mutual-exclusion requirement
- Mutual exclusion maintained:
  - A task exiting the CS either sets the lock to 0 or waiting[i] to false - both allow a waiting task to enter CS

- Bounded-waiting requirement fulfilled
  - when a task leaves its critical section, it scans the waiting[i] in the cyclic ordering (i+1, i+2, ..., n-1, 0, ..., i-1)
  - It designates the first task in this ordering that is in the entry section (waiting[j] == true) as the next one to enter the critical section
  - Any task waiting to enter its critical section will thes do so within n-1 turns

Program Structure

```
while (true) {
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key == 1)
      key = compare_and_swap(&lock,0,1);
      waiting[i] = false;

    /* critical section */
   j = (i+1) % n;
   while ((j!=i) && !waiting[j])
      j = (j+1) % n;
   if (j == i)
     lock = 0;
   else
      waiting[j] = false;

    /* remainder section */
}
```

## Atomic Variables

- Atomic variables are a tool for solving the critical-section problem
- Provides atomic operations on basic data types such as Int and bool