

# CS1 3131

## Operating Systems

### Study Guide



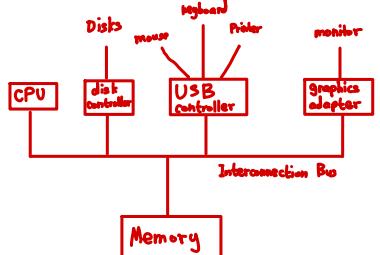
Summer 2024

# 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)

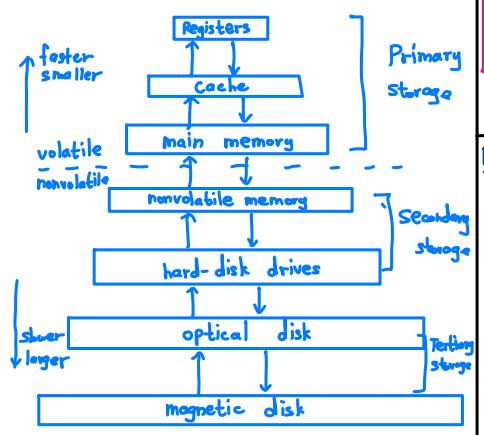
## Interconnection Bus



## 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

## 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.

## 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

## Computer System Architecture

- Single Processor
  - Some small systems still use a single processor
  - They can also have some other special-purpose processors as well

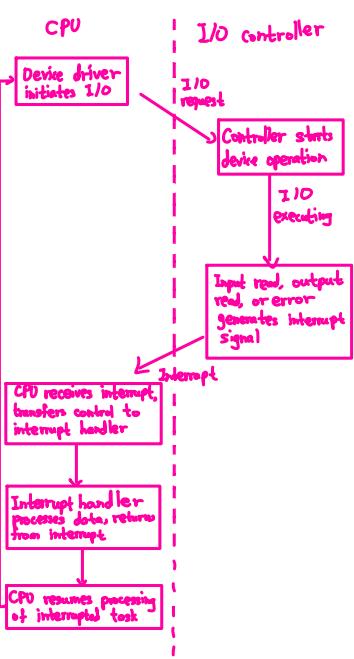
## Multiprocessors

- 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

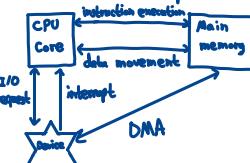
- Loosely coupled multiple CPUs
- Usually used to provide high-availability service

## 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 3.3 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



## 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 provide 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.

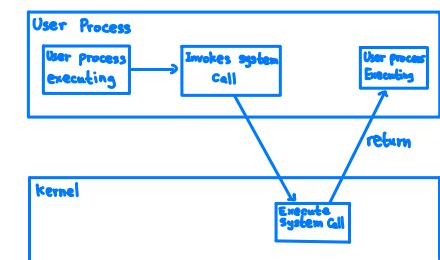
A timer can be set to interrupt the computer after a specified period.

- Keep a counter that is decremented by the physical clock. When it hits zero an interrupt is raised for an action.

If the timer interrupt, 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, 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 the 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 authentication 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: 'fingertip' 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 popular 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 tools 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 systems

- 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 programmers 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 block 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 these 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 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

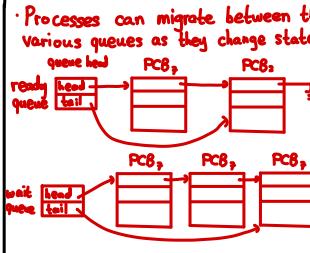
Process State
Process Number
Program Counter
Registers
Memory Limits
List of open files
...

## 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 reshuffle. The number of processes currently in memory is known as the degree of multiprogramming

## Process Scheduling Queues

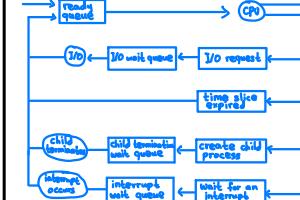
- As processes enter a system, they are put into a ready queue
  - where they are ready and waiting to execute on a CPU's core
  - This queue is generally stored as a linked list
  - Each PCB includes a pointer field that points to the next PCB in the ready queue
- The system includes other queues
  - 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 slower than processor, 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



- When a process migrates 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 queuing 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 used 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 state 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

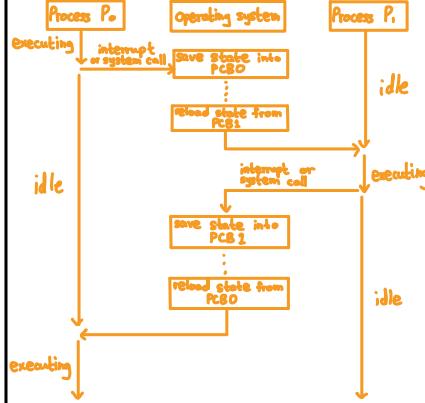
## Long-term Scheduling

- 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 resumed 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

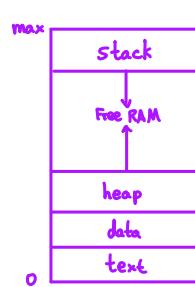
## Context Switch

- 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 fulfill 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() return: 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 child processes



## 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



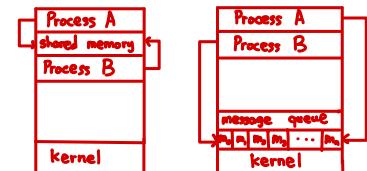
## Process Termination

- 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

## Shared Memory



## 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 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
- 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

## 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 pipe 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[2])

On Windows they are called anonymous pipes

The function Pipe(inet fd[2]) creates a pipe that is accessed through the int fd[2] 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
- Defined as FIFOs on UNIX systems
- Created with mknod() 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 16.25.19.8:1625 refers to port 1625 on host 16.25.19.8
- 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
- client
  - user calls bind to send RPC message to procedure X
  - kernel sends port F to matchmaker to find port number
  - matchmaker receives message, looks up owner
  - kernel places port F in user RPC message
  - kernel sends port F to client
  - client sends To: client, Port: kernel, Port: F, Port: P
  - kernel places port F in user RPC message
  - kernel sends port F to server
  - server sends From: client, Port: kernel, Port: F, Port: P
  - kernel receives reply, passes it to user
  - kernel places reply, port F to user
  - user receives reply, port F from kernel
  - kernel places reply, port F to user
  - user receives reply, port F from kernel

## 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 receives thousands of client service requests
- 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 server allows to create a separate thread rather than creating another process
  - (1) request
  - client → server
  - (2) create new thread to service the request
  - (3) resume listening for additional client requests

## Benefits of Threads

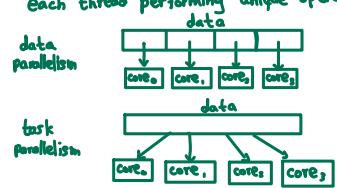
- Responsiveness:** may allow continued execution if part of a process is blocked, especially for user interface
- Resource Sharing:** threads share resources of process, easier than shared memory or message passing between processes
- Economy:** cheaper than process creation, thread switching lower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

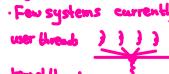
$$\text{speedup} \leq \frac{1}{S + \frac{(1-S)}{N}}$$

## 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

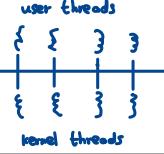
## Many-to-One Model

- 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



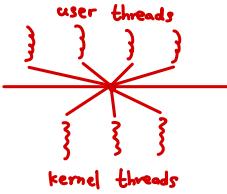
## One-to-One Model

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- A 1-1-1 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



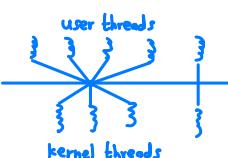
## Many-to-Many Model

- 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 pthread.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

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

## 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)
- Variables 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

## Implicit Thread

- Growing in popularity as number of threads increase program correctness more difficult with explicit thread
- Creation and management of threads done by compiler 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 space and this includes all threads
- Which of the two versions fork() to use?
  - It depends on the application:
    - if exec() is called immediately after forking, then duplicating all threads is unnecessary
    - 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 shared data in an inconsistent state

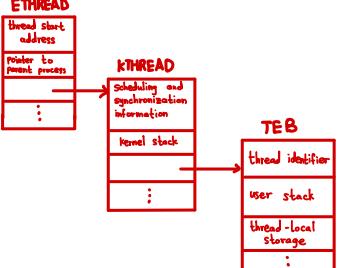
- Deferred cancellation allows the target thread to periodically check if it should be cancelled
- One thread indicates that a target thread is to be cancelled
- cancellation occurs only after the target thread has checked a flag to determine whether or not it should be cancelled
- The thread can only 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
  - ETHEAD (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 parent 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

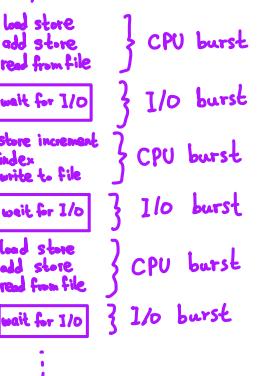
If no flags are set when clone() is invoked, no sharing takes place, resulting in functionality similar to fork()

## CPU Scheduling Motivation

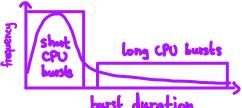
- The motivation behind multiprogramming is to have some process running at all times, to maximize CPU utilization.
- When for example if one process waits for an I/O, another process can execute on the core.
- This pattern continues - every time one process has to wait, another process can take over 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:
  - Switches from running to waiting state (I/O request)
  - Switches from running to ready state (interrupt occurs)
  - 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 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 particular 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

<p><b>Discussion of SJF</b></p> <ul style="list-style-type: none"> <li>Low average waiting time</li> <li>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)</li> <li>Starvation: occurs when a process waits for an indefinite time to get the resource it requires.</li> <li>Always overtaken by other processes</li> </ul>	<p><b>Multilevel Queue Scheduling</b></p> <ul style="list-style-type: none"> <li>Partition processes into several separate queues based on priority or process type           <ul style="list-style-type: none"> <li>foreground (interactive processes): h-priority</li> <li>background (batch processes): l-priority</li> </ul> </li> <li>Entering the system, a process permanently assigned to a given queue           <ul style="list-style-type: none"> <li>low scheduling overhead, but it is inflexible</li> </ul> </li> <li>Each queue has its own scheduling algorithm</li> <li>How must scheduling must be done from among the queues:           <ul style="list-style-type: none"> <li>Fixed priority preemptive scheduling               <ul style="list-style-type: none"> <li>i.e. the interactive queue may have absolute priority over the background queue</li> <li>serve all foreground processes first, then move to serve background processes</li> <li>Possibility of background processes starvation</li> </ul> </li> <li>Time slice               <ul style="list-style-type: none"> <li>Each queue gets a certain portion of the CPU time which it can then schedule among its various processes</li> <li>Not necessarily optimal</li> </ul> </li> </ul> </li> </ul>	<p><b>Symmetric Multiprocessing Complication</b></p> <ul style="list-style-type: none"> <li>Load Balancing - it is important to keep the workload balanced among all CPUs to fully benefit from having more than one CPU           <ul style="list-style-type: none"> <li>Required when each CPU has its own ready queue</li> <li>Push migration: a task runs periodically to evenly distribute load among CPUs (moving task from overloaded to an idle or a less-busy CPU)</li> <li>Pull migration: an idle CPU pulls a waiting task from a busy CPU</li> <li>Linux OS supports both Push and Pull techniques</li> <li>Can counteract the benefits of processor affinity, since it moves processes from one CPU to another</li> </ul> </li> <li>Processor affinity - process has affinity for processor on which it is currently running           <ul style="list-style-type: none"> <li>When a process runs on a physical CPU, cache memory is updated with content of the process</li> <li>If the process is moved to another CPU, benefits of caching is lost</li> <li>SMP systems try to keep processes running on the same physical CPU, known as processor affinity</li> </ul> </li> </ul>
<p><b>SJF-CPU Next Burst Prediction</b></p> <ul style="list-style-type: none"> <li>Can only estimate the length of a burst - should be like last ones</li> <li>Then pick process with shortest predicted next CPU burst</li> <li>Can be done by using the length of previous CPU bursts for this process, using exponential averaging           <ul style="list-style-type: none"> <li><math>t_n = \text{actual length of } n^{\text{th}} \text{ CPU burst}</math></li> <li><math>T_{n+1} = \text{predicted value for next CPU burst}</math></li> <li><math>\alpha, 0 \leq \alpha \leq 1</math></li> <li><math>T_{n+1} = \alpha t_n + (1-\alpha) T_n</math></li> <li>Commonly, <math>\alpha = \frac{1}{2}</math></li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>Each queue has its own scheduling algorithm</li> <li>How must scheduling must be done from among the queues:           <ul style="list-style-type: none"> <li>Fixed priority preemptive scheduling               <ul style="list-style-type: none"> <li>i.e. the interactive queue may have absolute priority over the background queue</li> <li>serve all foreground processes first, then move to serve background processes</li> <li>Possibility of background processes starvation</li> </ul> </li> <li>Time slice               <ul style="list-style-type: none"> <li>Each queue gets a certain portion of the CPU time which it can then schedule among its various processes</li> <li>Not necessarily optimal</li> </ul> </li> </ul> </li> </ul>	<p><b>2 ways</b></p> <ol style="list-style-type: none"> <li><b>Analytically:</b> 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.</li> <li><b>Empirically:</b> feed in data from a real system under study, and record the sequence to trace files, then use this sequence to drive the simulation</li> </ol> <pre> graph LR     A[actual process execution] --&gt; B[trace tape]     B --&gt; C[Simulation FCFS]     B --&gt; D[Simulation SJF]     B --&gt; E[Simulation RR(Quantum)]   </pre>
<p><b>Round Robin Scheduling (RR)</b></p>	<p><b>Multilevel Feedback Queue Algorithm</b></p>	<p><b>Race Condition</b></p>
<ul style="list-style-type: none"> <li>Each process gets a small unit of CPU time slice called time quantum <math>q</math>, usually 10-100 milliseconds</li> <li>After <math>q</math> time has elapsed (before any other interruption), the process is preempted (by the timer interrupt)           <ul style="list-style-type: none"> <li>The RR scheduling algorithm is thus preemptive</li> </ul> </li> <li>Interrupted process is added to the end of the ready queue</li> <li>The performance of the RR algorithm depends heavily on the size of the time quantum</li> <li>If there are <math>n</math> processes in the ready queue and the time quantum is <math>q</math>, then each process gets <math>\frac{1}{n}</math> of the CPU time in chunks of at most <math>q</math> time units at once</li> <li>No process waits more than <math>(n-1)q</math> time units</li> </ul> <p>Timer interrupts every quantum to schedule next process</p> <ul style="list-style-type: none"> <li><math>q</math> large <math>\Rightarrow</math> FIFO</li> <li><math>q</math> small <math>\Rightarrow</math> <math>q</math> must be large with respect to context switch, otherwise overhead is too high</li> <li>Typically, higher average turnaround than SJF, but better response time</li> <li>If the time quantum <math>q</math> is extremely large, RR degenerates to FCFS policy</li> <li>If the time quantum is extremely small, the RR approach can result in a large number of context switches</li> <li>Thus, we want the time quantum to be large with respect to the context-switch time, but not too large</li> </ul>	<ul style="list-style-type: none"> <li>A process can move between the various queues           <ul style="list-style-type: none"> <li>An aging process can be promoted to a higher priority queue to prevent starvation</li> <li>A "gourmand" process using too much CPU time can be moved down to a lower priority queue</li> <li>Short CPU bursts processes are left in the higher-priority queues</li> </ul> </li> <li>Multilevel-feedback queue scheduler defined by the following parameters:           <ul style="list-style-type: none"> <li>number of queues</li> <li>scheduling algorithms for each queue</li> <li>method used to determine when to promote a process</li> <li>method used to determine when to demote a process</li> <li>method used to determine which queue a process will enter when that process needs service</li> </ul> </li> <li>Adaptable to specific systems</li> <li>But most complex to implement</li> </ul>	<ul style="list-style-type: none"> <li>Concurrent threads often share user data (files or common memory) and resources (cooperative tasks)</li> <li>Race condition occurs where several tasks access and manipulate the same data concurrently</li> <li>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)</li> </ul>
<p><b>Priority Scheduling</b></p>	<p><b>Multiple - Processor Scheduling</b></p>	<p><b>Critical Section Problem</b></p>
<ul style="list-style-type: none"> <li>A priority number (integer) is associated with each process</li> <li>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</li> <li>Problem: Starvation - low priority processes may never execute</li> <li>Solution: Aging - as time progresses increase the priority of the process</li> <li>Some systems use lower numbers for lower priority; others use it for high priority. This difference can lead to confusion</li> <li>In this text, we assume that low numbers represent high priority</li> </ul>	<ul style="list-style-type: none"> <li>Scheduling multiple CPUs is more complex than a single CPU</li> <li>Therefore, OS must deal with efficient load sharing and time management among the available multiple CPUs</li> <li>Result: Parallel execution           <ul style="list-style-type: none"> <li>Load sharing - when multiple processes may run in parallel</li> <li>OS overheads shared among different cores</li> </ul> </li> <li>All processors are identical - homogeneous in terms of their functionality</li> <li>Two approaches to multiprocessor scheduling           <ul style="list-style-type: none"> <li>Asymmetric multiprocessing - only one core accesses the system data structures, reducing the need for data sharing</li> <li>Symmetric multiprocessing - each CPU is self-scheduling               <ul style="list-style-type: none"> <li>The standard approach in modern computing OS's to support multiprocessors</li> <li>Two possible strategies for organizing the scheduling                   <ul style="list-style-type: none"> <li>All processes in common ready queue</li> <li>Each has its own private queue of ready processes</li> </ul> </li> </ul> </li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>When multiple threads execute in parallel, we cannot make assumptions about the execution speed of the threads, nor their interleaving</li> <li>May be different each time the program is run</li> <li>Critical section is part of a task whose execution must not interfere with other tasks' critical sections           <ul style="list-style-type: none"> <li>Once a task enters, it must be allowed to complete this section without allowing other tasks to manipulate on the same data</li> </ul> </li> <li>Critical-section problem is to design a protocol that the threads can use to synchronize their activity so as to cooperatively share data           <ul style="list-style-type: none"> <li>The result of their actions does not depend on the interleaving order of execution</li> </ul> </li> <li>The execution of critical sections must be mutually exclusive: at any time, only one thread can execute a critical section for a given data           <ul style="list-style-type: none"> <li>This can be achieved by placing special instructions at the beginning and end of the critical section</li> </ul> </li> <li>Once a task enters the critical section, it must complete it as an atomic block (no other tasks are allowed to mess with the same data)</li> <li>The critical section must be locked to become indivisible (atomic)</li> </ul> <p><b>Program Structure</b></p> <pre> while (true) {     entry section     critical section     exit section     remainder section }   </pre> <p><b>Criteria for Valid Solutions to CSP</b></p> <ol style="list-style-type: none"> <li><b>Mutual Exclusion</b> <ul style="list-style-type: none"> <li>At any time, at most one thread can be in a critical section</li> </ul> </li> <li><b>Progress</b> <ul style="list-style-type: none"> <li>Ensures programs will cooperatively determine what task will next enter its critical section</li> <li>A critical section will only be given to a task that is waiting to enter it</li> <li>Whenever a CS becomes available, if there are threads/tasks waiting for it, one of them must be able to enter it</li> </ul> </li> <li><b>Bounded Waiting</b> <ul style="list-style-type: none"> <li>Limits the amount of time a program will wait before it can enter its critical section</li> </ul> </li> </ol>

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 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 Ti should wait if the other wants to enter and it's the other's turn

flag[j]=true and turn=j

Thread Ti can enter if the other doesn't want to enter or it's their turn

flag[j]=false or turn=i

Ti uses the boolean flag[i] to indicate its desire to enter its CS

It also uses turn to give Ti the priority to enter its CS

Ti can enter the critical section only when flag[i]=false or turn=i

To enter the critical section, a task Ti first sets flag[i] to true and then sets turn to the value j

thereby asserting that if Tj 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

### Software Based Solution Discussion

#### 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

P0 → turn=1 → flag[0]=true → CS

P1 → turn=0, flag[1]=true → CS

#### Software Based Solution Discussion

In order for threads with shared variables 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 change 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 Ti is

```
while (true){  
    while (test_and_set(&lock));
```

/\* critical section \*/

lock = false;

/\* remainder section \*/

#### Some disadvantages:

Still using busy waiting

When Ti leaves its critical section, the selection of Tj 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;
```

3

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 \*/

3

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 Ti 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, i+1, ..., n-1, 0, ..., i-1)
- It designates the first task in this ordering that is in the entry section (waiting[i]==true) as the next one to enter the critical section
- Any task waiting to enter its critical section will thus do so within n-1 turns

## Program Structure

```

while (true) {
    waiting[i] = true;
    key = i;
    while (waiting[i] && key == i)
        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
- Atomic variables ensure mutual exclusion on a shared variable
- Special atomic data types and functions for accessing and manipulating atomic variables exist only on supported operating systems
- Limitation: their use is often limited to single updates of shared data such as counters and sequence generators
- They do not entirely solve race conditions in all circumstances

## Mutex Locks (Spinlocks)

- Mutex: short for mutual exclusion
- Simplest of high-level software synchronization tools
- Mutex lock protects critical sections and thus prevents race conditions
- A task must acquire the lock before entering a critical section; it releases the lock when it exits the critical section
  - acquire() function acquires the lock
  - release() function releases the lock
- On multicore systems, spinlocks can be the preferable choice for locking, if a lock is to be held for a short duration
  - One thread can "spin" on one core while another thread performs its critical section on another core

Definition of acquire() and release():

```

acquire() {
    while(!available); available = true;
    available = false;
}

```

boolean variable available is true only if the lock is available, otherwise it is false

A task calling acquire() can enter critical section, only if the lock is available

- If not available, then the calling task is blocked until the lock is released
- Disadvantage:
  - requires busy waiting
  - Other tasks not in the critical section must loop continuously in the call to acquire()
- Not appropriate to implement on single-processor systems
- The condition that would break a task out of the spinlock can be obtained only by executing a different task
- If the task is not relinquishing the processor, other tasks do not get the opportunity to set the program condition required for the first task to make progress

- In a multiprocessor system, other tasks execute on other processors and therefore can modify the program state in order to release the first task from the spinlock

## Semaphores

- Similar to a mutex lock, but can also provide more sophisticated ways for tasks to synchronize their activities
- A semaphore S is an integer variable is accessed only through two standard atomic operations
  - wait() and signal() with definitions

```

wait(S) {
    while (S<=0); S++;
    S--;
}

```

  - The integer S in wait() and signal() operations must be executed atomically
    - no other task can simultaneously modify that same semaphore S value
  - Testing of ( $S \leq 0$ ) and its possible modification ( $S--$ ) must not be interrupted

## Binary Semaphore

- The value of a binary semaphore S can take only 0 or 1
- Thus, it behaves similarly to mutex locks
- In fact, a binary semaphore is a counting semaphore with S is restricted to 0 and 1
- Binary semaphores can be used for mutual exclusion instead of mutex locks on systems that do not provide mutex locks
- Binary semaphores can be used to implement mutual exclusion among n tasks

## Counting Semaphore

- Counting semaphores control access to a finite number of instances
  - S can range over an unrestricted domain
- S initialized to the number of resources available
- Each task that wishes to use a resource performs a wait() operation on S
- When a process releases a resource, it performs a signal() operation
- When the count for S goes to 0, all resources are being used
- After that processes that wish to use a resource will block until the count becomes greater than 0
- When  $S > 0$ :

- The number of threads that can run wait() without busy waiting is S
  - S threads can enter the critical section
  - In solutions where  $S > 1$  it will be necessary to have a 2nd semaphore to control threads entry to critical section one at a time (mutual exclusion)
- When S becomes greater than one, the first thread to enter the critical section is the first to test S. This is a random choice, which may cause starvation

## Semaphores with no busy waiting

Implementation without busy waiting

- The mutex and the previous semaphore all suffer from busy waiting
- To overcome the problem, we can modify the definition of wait() and signal() operations

- To overcome the busy waiting problem, the definition of wait is modified as follows:
  - Instead of busy waiting in wait(), the task can suspend itself
  - Suspend will place a task into a waiting queue associated with the semaphore and switches its state to waiting
  - Then control is transferred to the CPU scheduler, which selects another ready task to execute
- A suspended task, waiting on a semaphore S, should be restarted when some other task executes a signal() operation

- The task is restarted by a wakeup() operation, which changes the task from the waiting state to the ready state.
- The task is then placed in a ready queue

- To implement semaphores under this new definition, we define a semaphore as new type as follows:

```

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

```

- A semaphore typed variable has 2 members:
  - int value
  - list of processes

- When a task must wait on a Semaphore, it is added to the list of tasks

A signal() operation removes one task from the list of waiting bats and awakens that task

- Definition of wait():
  - S is a semaphore typed variable
  - \*S is a pointer to S
  - S->value is a pointer to S member value
  - sleep() replaces previous busy wait

```

wait(semaphore *S){
    S->value--;
    if (S->value < 0) {
        add process to S->list;
        sleep();
    }
}

```

- Definition of signal():
  - sleep() suspends the task that invokes it
  - wakeup(P) resumes the execution of a suspended task P
  - These two operations are provided by the OS as basic system calls

```

signal(semaphore *S){
    S->value++;
    if (S->value == 0) {
        remove a process from S->list
        wakeup();
    }
}

```

- Under this new definition, if a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore

- The operations wait() and signal() must be executed atomically

In a single core, it can be solved by inhibiting interrupts when a task is performing these operations

In multi-core, interrupts also must be disabled on every processing core

- It is important to admit that we have not completely eliminated busy waiting, we have limited busy waiting to the critical sections of the wait() and signal() operations, and these sections are short

- Thus, the critical section is almost never occupied, and busy waiting occurs rarely, and then for only a short time

## Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal(mutex) ... wait(mutex)
  - wait(mutex) ... wait(mutex)
  - Omitting of wait(mutex) or signal(mutex) (or both)
- Deadlock and starvation are possible
- Various types of errors can be generated easily when programmers use semaphores or mutex locks incorrectly to solve the critical-section Problem
- One strategy for dealing with such errors is to incorporate simple synchronization tools as high-level language constructs

## Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization

- Abstract data types, encapsulates:
  - internal variables and data with a set of functions (procedures) to operate on that data
  - initialization sequence

```

monitor monitor-name {
    /* shared variable declarations */
    function P1(...) {...}
    function P2(...) {...}
    ...
    function Pn(...) {...}
    initialization_code(...){...}
}

```

- Local (internal) variables are only accessible by code within encapsulated functions (procedures)

- A task enters the monitor by invoking one of its functions

- Ensures that only one task at a time can execute within the monitor at any moment; but more tasks can be waiting in the monitor.

- Ensures mutual exclusion without the need to explicitly program

- Shared data encapsulated and protected by the monitor

- Shared data is locked when accessed by a task

- Task synchronization is provided by condition typed variables

- They represent conditions after which a thread might wait before executing in the monitor

- Two versions

- Simple monitors

- Essentially Java's synchronized methods

- Monitors with condition variables

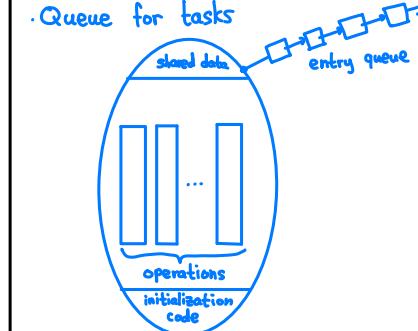
- Task synchronization is provided by condition-typed variables that are local to the monitor

## Simple Monitors

- Synchronized methods

- Only one task can enter at a time

- Queue for tasks

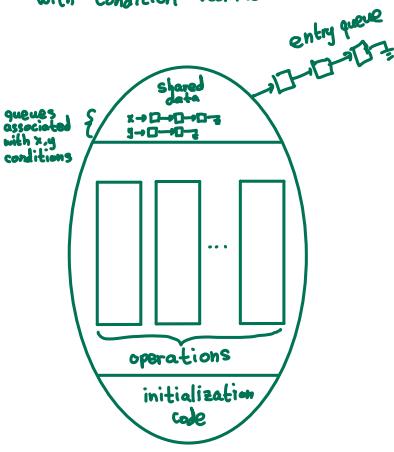


## Condition Variables

- The simple monitor is not sufficiently powerful for modeling some synchronization schemes. So, we need to define additional synchronization mechanisms: condition variables
- Condition variables are accessible only within the monitor
- `x.wait()` and `signal()` are the only two operations that can be invoked on a condition variable

- `x.wait()`: suspend the task invoking this operation on condition `x` until another process invokes `x.signal()`
- `x.signal()` resumes exactly one suspended task on condition `x`
  - If no task suspended, do nothing
  - If more than one task are suspended, then choose one

## Schematic view of monitors with condition variables



If a task executes `x.signal()` and a task Q is suspended in `x.wait()`, what should happen next?

Two tasks Q and P are not allowed to be active simultaneously within the monitor, then P must wait

Two possibilities exist

- Signal and wait: P either waits until Q leaves the monitor or waits for another condition
- Signal and continue: Q either waits until P leaves the monitor or waits for another condition

Language implementer can decide which of the 2 options to adopt

## Bounded Buffer Problem: Producer-Consumer

- A finite size array in memory is shared by a number of producer and consumer threads
- A classic problem in the study of communicating processes: a producer produces data for a consumer process
- A buffer is needed to store produced item
- If the buffer is of length 1, the producer and consumer must necessarily go at the same speed
- Longer length buffers allow some independence



## Bounded Buffer

- A fundamental data structure in OS
- Bounded buffer is in the memory shared between producer and consumer
- When writing information to the buffer, the producer updates the in pointer
- When reading information in the buffer, the consumer updates the out pointer
- If the buffer is full, the producer will have to go to sleep (later be woken up by the consumer)
- The role of the consumer is symmetrical
- Bounded buffers are everywhere in computing, and everywhere in operating systems
- The queues used in operating systems are bounded buffers:
  - Pipes in Unix
  - queues for resources
  - communication protocols use bounded buffers
  - a client communicates with a server through bounded buffers

## Difficulty Concerning Buffers

- A buffer is shared between two or more processes
- We must make sure to synchronize these processes, otherwise we will easily fall into troubles
- Suppose a consumer process seeks to read data that a producer has not yet finished writing
- Or that two processes seek to update the indices simultaneously
- Since the producer and the consumer are independent tasks, problems could occur allowing concurrent buffer access
- Semaphores can solve this problem

## Semaphore Solution to Bounded Buffer Problem

- Producer and consumer share the following data structures and initialization:

```
int N;
Semaphore mutex = 1;
Semaphore empty = N;
Semaphore full = 0;
```

- Assume that the pool consists of N buffers
  - each buffer capable of holding one item
  - The mutex binary semaphore provides mutual exclusion for accesses to the buffer pool
  - mutex is initialized to the value 1
- The empty and full semaphores count the number of empty and full buffers
  - empty is initialized to the value N
  - full is initialized to the value 0
- Note the symmetry between the producer and the consumer
- We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer

## Producer Process:

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

## Consumer Process:

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

## Readers-Writers Problem

- Multiple threads can access a database
- To read or write there
- Writers must be in sync with each other and with readers (only one writer at a time)
  - a thread should be prevented from reading during a write
  - two writers should be prevented from write simultaneously
  - if a writer and some other process access the database simultaneously, chaos may result
- Readers are allowed access to the database simultaneously
- Several variations are possible
  - The simplest one is the first readers-writers problem
    - No reader be kept waiting unless a writer has already obtained permission to use the shared object
  - Second readers-writers problem
    - A ready writer must write as soon as possible (a waiting writer must access the shared object before a new reader is ready)
  - A Solution to either problem may result in starvation

## First Readers-Writers Problem

- The reader processes share the following data structures

```
Semaphore rw-mutex = 1;
Semaphore mutex = 1;
int read_count = 0;
The binary semaphores mutex and rw-mutex are initialized to 1
read_count is a counter representing # of active readers - initialized to 0
The semaphore rw-mutex is common to both reader and writer processes
The mutex semaphore is used to ensure mutual exclusion when the variable read-count is updated
The read-count variable keeps track of how many processes are currently reading the object
The semaphore rw-mutex functions as a mutual exclusion semaphore for the writers
```

- It is also used by the first or last reader that enters or exits the critical sections
- It is not used by readers that enter or exit while other readers are in their critical sections
- If a writer is in the critical section and n readers are waiting
  - n-1 readers are queued on mutex
  - one reader is queued on rw-mutex
- When a writer executes signal(mutex)
  - we may resume the execution of either the waiting readers or a single waiting writer
- Selection decided by scheduler

- Reader process
 

```
while (true) {
        wait(mutex);
        read_count++;
        if (read_count == 1)
            wait(rw-mutex);
        Signal(mutex);
```

- /\* reading is performed \*/
 

```
wait(mutex);
      read_count--;
      if (read_count == 0)
          signal(rw-mutex);
      signal(mutex);
```

- Writer process:
 

```
while (true) {
        wait(rw-mutex);
        /* writing is performed */
        signal(rw-mutex);
```

## Dining Philosophers Problem

- Five philosophers think, eat, think, eat, think,...
- One philosopher on each chair
- In the center of the table is a bowl of rice
- Five chopsticks available (two needed to eat)
- Thinking philosophers don't interact with others
- Each philosopher needs to pick up left and right chopsticks
- Philosophers cannot pick up a chopstick that is already in use
- Philosophers with both chopsticks can eat
- When finished eating, philosopher puts down both chopsticks and starts thinking
- Illustrates the difficulty of syncing and allocating resources among process without deadlock and starvation

## Semaphore Solution to Dining Philosophers Problem

- Each philosopher is a thread
- One semaphore per chopstick
  - Two philosophers cannot grab the same chopstick simultaneously
  - semaphore chopstick[5]
  - chopstick[i] initialized to 1
- This solution guarantees that no two neighbors are eating simultaneously
- But suppose that all five philosophers become hungry at the same time and each philosopher grabs the chopstick to the left
  - All the elements of chopstick will now be equal to 0
  - Then, when they try to grab the right chopstick... never going to happen... forever
  - A deadlock
- Solution:
 

```
semaphore chopstick[5];
...
while (true) {
    wait(chopstick[i]);
    wait(chopstick[(i+1)% 5]);
    ...
    /* eat for a while */
    signal(chopstick[i]);
    signal(chopstick[(i+1)% 5]);
    ...
    /* think for a while */
}
```
- Several possible remedies to the deadlock problem are the following:
  - Allow at most four philosophers to be sitting simultaneously at the table
  - Allow a philosopher to pick up chopsticks only if both chopsticks are available
  - Use an asymmetric solution - that is, an odd-numbered philosopher picks up first left and then right chopstick, whereas an even-numbered philosopher picks up right and then left chopstick
  - Monitor Solution
  - However, a satisfactory solution must guard against the possibility that one of the philosophers will starve to death
- A deadlock-free solution does not necessarily eliminate the possibility of starvation
- One solution: admit only 4 philosophers at a time who can attempt to eat (0 to 3)

T=3

```
semaphore chopstick[5]
...
while (true) {
    wait(T)
    wait(chopstick[i]);
    wait(chopstick[(i+1)% 5]);
    ...
    /* eat for a while */
    ...
    signal(chopstick[i]);
    signal(chopstick[(i+1)% 5]);
    signal(T);
    ...
    /* think for a while */
}
```

3

- There will always be at least one philosopher who can eat
  - even if everyone takes 1 or 2 chopsticks
  - Added a semaphore T which limits the number of philosophers who can attempt to eat

## Monitor Solution to Dining Philosophers Problem

- Deadlock free solution
  - Imposes the restriction on a philosopher to pick up chopsticks only if both are available
  - To code the solution, we need to distinguish among three states in which we may find a philosopher by the following data structure:
- ```
enum{ THINKING, HUNGRY, EATING } state[5];
```
- A philosopher i can set the variable `state[i]=EATING` only if the two neighbours are not eating:
- $$(\text{state}[(i+4) \% 5] \neq \text{EATING}) \text{ and } (\text{state}[(i+1) \% 5] \neq \text{EATING})$$
- We also need to declare: condition self [5]
- This allows philosopher i to wait on self[i] when it is hungry but is unable to obtain the two chopsticks
- Each philosopher, before starting to eat, must invoke the operation pickup()
- This act may result in the suspension of the philosopher process
  - After the successful completion of the operation, the philosopher may eat
- Following this, the philosopher invokes the putdown() operation
- Thus, philosopher i must invoke the operations `Pickup()` and `putdown()` in the following sequence while (true){
- ```
DiningPhilosophers.pickup();
    eat
    DiningPhilosophers.putdown();
```

- 3
- The distribution of the chopsticks is controlled by the monitor `DiningPhilosophers`
  - Shared variables: each philosopher has its own state that can be: (thinking, hungry, eating)
    - Philosopher i can only have `state[i]=eating` if its neighbors are not eating
  - Conditional variables: each philosopher has a condition self (delay eat with no available chopsticks)
  - Four functions
    - `pickup(i)` - philosopher i wants to pick up chopsticks
    - `putdown(i)` - philosopher i puts down the chopsticks
    - `test(i)` - test the state of philosopher i
    - `initialization_code()` - initialization
  - Monitor definition

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

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

```
    void putdown (int i) {
        state[i] = THINKING;
        test ((i+4) \% 5);
        test ((i+1) \% 5);
    }
```

```
3
```

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

```
3
```

```
    initialization_code() {
        for (int i=0; i<5; i++)
            state[i] = THINKING;
    }
```

```
3
```

## Liveness Failure vs. Deadlock vs. Livelock

**Liveness Failure:** Failure of a task to progress or terminate as when it got stuck in an infinite loop or in a deadlock

**Deadlock:** The state in which at least two tasks are stuck waiting for an event that can only be caused by one of the two tasks

**Livelock:** A condition in which a thread continuously attempts an action that fails

## System Model

- A system has limited resources shared among some competing tasks
- A task may utilize a resource in the following sequence:
  - Request: if the request cannot be granted immediately, then wait until it can acquire the resource
  - Use: can operate on the resource
  - Release: the task releases the resource
- A set of threads is in a deadlocked state when every thread in the set is waiting for an event that can be caused only by another thread in the set
- Multithreaded application developers must pay careful attention to how locks are acquired and released. Otherwise, deadlocks can occur

## Necessary Conditions for Deadlock

- Mutual Exclusion: At least one resource must be held in a nonshareable mode; that is, only one thread at a time can use the resource. If another thread requests that resource, the requesting thread must be delayed until the thread has been released
- Hold and wait: A thread must be holding at least one resource and waiting to acquire additional resources that are currently being held by other threads
- No preemption: Resources cannot be preempted; that is, a resource can be released only voluntarily by the thread holding it, after that thread has completed its task
- Circular wait: A set  $\{T_1, T_2, \dots, T_n\}$  of waiting threads must exist such that  $T_1$  is waiting for a resource held by  $T_2$ ,  $T_2$  is waiting for a resource held by  $T_3, \dots, T_m$  is waiting for a resource held by  $T_1$ , and  $T_n$  is waiting for a resource held by  $T_1$ .

## Resource Allocation Graph

- Consists of a set of vertices V and a set of edges E
- V is partitioned into two different types of nodes
  - $T = \{T_1, T_2, \dots, T_n\}$ , the set consisting of all the active tasks in the system
  - $R = \{R_1, R_2, \dots, R_m\}$ , the set consisting of all resource types in the system
- Request edge: A directed edge from task  $T_i$  to resource type  $R_j$  ( $T_i \rightarrow R_j$ )

- It signifies that task  $T_i$  has been requested an instance of resource type  $R_j$  and is currently waiting for that resource

- Assignment edge: A directed edge from resource  $R_j$  to task  $T_i$  ( $R_j \rightarrow T_i$ )

- It signifies that an instance of resource type  $R_j$  has been allocated to  $T_i$

- Pictorially, we represent each task  $T_i$  by a circle and each resource type  $R_j$  by a rectangle

Since resource type  $R_j$  may have more than one instance, we represent each such instance as a dot in the rectangle

## Deadlocks and Resource Allocation Graphs

- If a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state
- If there is a cycle, then the system may or may not be in a deadlocked state
  - If only one resource per type - deadlock
  - If several resources per type - possible deadlock

## Termination Hypothesis

- A task that has all the resources it needs, it uses them for a finite amount of time, then it releases them
- We say that the task terminates, but it could also continue, it doesn't matter, the important thing is that it releases the resource
- There is no deadlock if all tasks can terminate, one at a time
- In practice, this detection method oversimplifies the tasks

## Methods for Handling Deadlocks

- Deadlocks can be dealt with in one of the following ways:

- Ignore the problem altogether and pretend that deadlocks never occur in the system
- Use a protocol to prevent or avoid deadlocks, ensuring the system will never enter a deadlocked state
- Allow the system to enter a deadlocked state, detect it, and recover
- Deadlock Prevention: Provides a set of methods to ensure that at least one of the necessary conditions cannot hold
- Deadlock Avoidance: OS must get in advance some information concerning resources needed by a thread execution. Then the OS can decide whether or not the thread should wait - decision based on resources current availability
- Act after detection: Use an algorithm to detect a deadlock - If so, then run another algorithm to recover from the deadlock
- Don't care: OS do not deal with deadlock; system may stop working, in worst case scenario a manual intervention to restart the system is needed

## Deadlock Prevention

- Prevent at least one of the 4 necessary conditions
- Mutual exclusion: eliminate the use of shared resources and critical sections
  - Possible only in the case of completely independent processes
- Hold and wait: a process that requests new resources should not hold others
  - How to know?

- No preemption: if a process requests other resources and cannot have them, then it must release its resources then put to wait
  - Requires an intervention from OS
- Circular wait: impose an order on resources. Difficult

- Prevention is difficult in general

## Deadlock Avoidance

- Manage resources in a way that avoid getting into a deadlock
  - The simplest and most useful model requires that each task must declare the maximum number of anticipated resources
  - An algorithm examine all possible execution sequences to see if a circular wait is possible

The algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist

The resource-allocation state is defined by the number of available and allocated resources and the maximum demands of the tasks

## Safe State

- A state is safe if the system can allocate resources to each task (up to its anticipated maximum) in some order and still avoid a deadlock
- More formally, a system is in a safe state only if there exists a safe sequence for all tasks
- No deadlock is possible if a system is in safe state
- A deadlock is possible when a system is in unsafe state
  - In an unsafe state, the OS cannot prevent tasks from requesting resources in such a way that a deadlock occurs
  - The behaviour of the tasks controls unsafe states
- Avoidance ensure that a system will never enter an unsafe state
  - Do not allocate a resource to a task if the resulting state is not safe

A sequence of threads  $\langle T_1, T_2, \dots, T_n \rangle$  is a safe sequence for the current allocation state it, for each  $T_i$ , the resource requests that  $T_i$  can still make can be satisfied by the currently available resources include the resources held by all  $T_j$  which precede them ( $i < j$ )

- If the resources that  $T_i$  needs are not immediately available, then  $T_i$  can wait until  $T_j$  have finished
- When they have finished,  $T_i$  can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate
- When  $T_i$  terminates,  $T_{i+1}$  can obtain its needed resources, and so on. Therefore, a safe sequence is a sequence that allow to run all tasks to completion

If no such sequence exists, then the system state is said to be unsafe

## Resource Allocation Graph Algorithm

Consider a resource allocation graph with only one instance of each resource type

In addition to the actual request and assignment edges, we introduce a claim edge represented in the graph by a dashed line

A claim edge  $T_i \rightarrow R_j$  indicates that  $T_i$  may request  $R_j$  at some time in the future

Claim edge  $T_i \rightarrow R_j$  is converted to request edge when a process requests a resource, then when  $R_j$  is released the assignment edge  $R_j \rightarrow T_i$  is reconverted to a claim edge  $T_i \rightarrow R_j$

Resources must be claimed in advance in the system

If  $T_i$  requests  $R_j$ , this can be granted only if converting the request edge  $T_i \rightarrow R_j$  to an assignment edge  $R_j \rightarrow T_i$  does not result in the formation of a cycle in the resource allocation graph

A cycle detection algorithm is used to detect this

## Banker's Algorithm

A new thread entering the system must declare the maximum number of instances of each resource type that it may need, and it may not exceed the total number of resources in the system

When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state

If it will, the resources are allocated; otherwise, wait until available

Several data structures must be maintained to implement the banker's algorithm

$n$  is the number of threads in the system

$m$  is the number of resource types

Available: a vector of length  $m$  indicates the number of available resources of each type.

If  $\text{Available}[i]$  equals  $k$ , then  $k$  instances of resource type  $R_i$  are available

**Max:** An  $n \times n$  matrix defines the maximum demand of each thread. If  $\text{Max}[i][j]$  equals  $k$ , then thread  $T_i$  may request at most  $k$  instances of resource type  $R_j$ .

**Allocation:** An  $n \times n$  matrix defines the number of resources of each type currently allocated to each thread. If  $\text{Allocation}[i][j]$  equals  $k$ , then thread  $T_i$  is currently allocated  $k$  instances of resource type  $R_j$ .

**Need:** An  $n \times n$  matrix indicates the remaining resource need of each thread. If  $\text{Need}[i][j]$  equals  $k$ , then thread  $T_i$  may need  $k$  more instances of resource type  $R_j$  to complete its task.

Note:  $\text{Need}[i][j] = \text{Max}[i][j] - \text{Allocation}[i][j]$

**Safety Algorithm:** determines if a system is in a safe state

1. **Work**: vector of length  $m$

$\text{Finish}$ : vector of length  $n$

initialize  $\text{Work} = \text{Available}$ ,  $\text{Finish}[i] = \text{false}$  for  $i = 0, 1, \dots, n-1$

2. Find an index  $i$  such that both

- a.  $\text{Finish}[i] = \text{false}$
- b.  $\text{Need}[i] \leq \text{Work}$

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

3.  $\text{Work} = \text{Work} + \text{Allocation}[i]$

$\text{Finish}[i] = \text{true}$

go to step 2

4. If  $\text{Finish}[i] = \text{true}$  for all  $i$ , then system is in safe state

**Resource-Request Algorithm:** determines whether requests can be safely granted

•  $\text{Request}[i]$  be the request vector for thread  $T_i$ . If  $\text{Request}[i][j] = k$ , then thread  $T_i$  wants  $k$  instances of resource type  $R_j$

1. If  $\text{Request}[i] \leq \text{Need}[i]$ , go to step 2. Otherwise, raise error condition

2. If  $\text{Request}[i] \leq \text{Available}$ , go to step 3. Otherwise,  $T_i$  must wait

3. Have the system pretend to have allocated the requested resources to thread  $T_i$  as follows:

$\text{Available} = \text{Available} - \text{Request}[i]$

$\text{Allocation}[i] = \text{Allocation}[i] + \text{Request}[i]$

$\text{Need}[i] = \text{Need}[i] - \text{Request}[i]$

If the resulting resource-allocation state is safe, the transaction is completed, and thread  $T_i$  is allocated its resources. However, if the new state is unsafe, then  $T_i$  must wait for  $\text{Request}[i]$ , and the old resource-allocation state is restored

## Deadlock Detection

• It is difficult to detect if there is actually a deadlock in a system

• We might see that a number of processes are waiting for resources

• To know that there is a deadlock, we must know that no process in a group has a chance of reaching the resource

## Deadlock Detection: Single instance of Each Resource

• Construct a resource-allocation graph

• Then we can convert it into a wait-for graph. An edge from  $T_i$  to  $T_j$  in a wait-for graph implies that thread  $T_i$  is waiting for thread  $T_j$  to release a resource that  $T_i$  needs

•  $T_i \rightarrow T_j$  in a wait-for graph exists if and only if the corresponding resource-allocation graph contains  $T_i \rightarrow R_k$  and  $R_k \rightarrow T_j$  for some resource  $R_k$

• Use cycle detection algorithms to check if there are cycles in the wait-for graph. If there's a cycle, then a deadlock exists

## Deadlock Detection: Several instances of Resource type

• Uses similar method as banker's algorithm

```
1. Work = Available
   For i=0,...,n-1
     if Allocation[i]≠0
       Finish[i]=false
     else
       Finish[i]=true
```

2. Find an index  $i$  such that both

- a.  $\text{Finish}[i] = \text{false}$
- b.  $\text{Request}[i] \leq \text{Work}$

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

3.  $\text{Work} = \text{Work} + \text{Allocation}[i]$

$\text{Finish}[i] = \text{true}$

Go to step 2

4. If  $\text{Finish}[i] = \text{false}$  for some  $i$ ,  $0 \leq i < n$ , then system is in a deadlocked state. Moreover, if  $\text{Finish}[i] = \text{false}$ , then thread  $T_i$  is deadlocked

## Recovery From Deadlock

• When a detection algorithm determines that a deadlock exists, several alternatives are available

• Inform the OS to deal with deadlock

• Let system recover from deadlock

• Process and thread termination: simply to abort one or more threads to break the circular wait

• Resource preemption: Preempt some resources from one or more of the deadlocked threads

## Process and Thread Termination

• Abort all deadlocked processes

• It will break the deadlock cycle, but at great expense

• The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later

• Abort one process at a time until the deadlock cycle is eliminated

• Considerable overhead: after aborting a process a deadlock-detection algorithm must be invoked to determine if any processes are still deadlocked

• We need a policy:

• What the priority of the process is?

• How long the process has computed and how much longer the process will compute before completing its designated task?

• How many and what types of resources the process has used?

• How many more resources the process needs in order to complete?

• How many processes will need to be terminated?

## Resource Preemption

• It successfully preempts some resources from processes and give these resources to other processes until the deadlock cycle is broken

• Three issues needs to be addressed

• Selecting a victim: Determine the preemption order of to minimize cost

• factors may include number of resources per deadlocked process, time cost, etc

• Rollback: We must have a roll back mechanism for preempted process to some safe state and restart it from that state

• In most cases about process and restart it

• Starvation: Possible if same process is always picked as victim

• Solution: Include number of rollbacks in the cost factor

## Memory Management

• The main purpose of a computer system is to execute programs

• Modern computer systems maintain several processes in memory during system execution

• Many memory-management schemes exist, reflecting various approaches, and the effectiveness of each algorithm varies with the situation

• Most memory-management algorithms require some form of hardware support

• Memory can be managed in various ways

• The memory-management algorithms vary from a primitive bare-machine approach to a strategy that uses Paging

• Each approach has its own advantages and disadvantages

• Selection of a memory-management method for a specific system depends on many factors especially on the system hardware design

• Most algorithms require hardware support, leading many systems to have closely integrated hardware and OS memory management

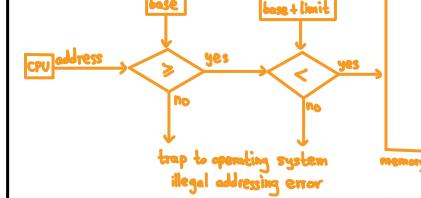
• Limit register: specifies the size of the range

• CPU compares every address generated in user mode with the registers

• Any attempt by a program executing in user mode to access OS memory or other users' memory results in a trap to the OS

• which treats the attempt as a fatal error

• This prevents a user program from modifying the code or data structures of either the OS or other users



• The base and limit registers can be loaded only by the OS by a special privileged instruction

• OS, executing in kernel mode, is given unrestricted access to both OS memory and users' memory

• This allows the OS to load users' programs

• To dump out those programs in case of errors

• To access and modify parameters of system calls

• To perform I/O and read from user memory

## Address Binding

• A user process may reside in any part of the physical memory

• A user program goes through several steps before being executed and addresses may be represented in different ways

• A source program code has symbolic names

• A compiler typically binds these symbolic addresses to relocatable addresses

• The linker or loader in turn binds the relocatable addresses to absolute addresses

• Each binding is a mapping from one address space to another

• Instructions/data binding to memory is done at any step along the way:

• At compile time: if the process space in memory is known, then absolute code can be generated from the address and up – must recompile if memory location change

• At load time: if at compile time the process space in memory is not known, then the compiler must generate relocatable code and final absolute binding is generated at load time

• If starting address changes, then only reload the code to incorporate this changed value

• At execution time: binding is done during execution time (dynamic binding)

• Load, relocate system library during run-time

• Special hardware is needed

• Most operating systems use this method

## Memory Space Protection

• Main memory and CPU registers are the only general-purpose storage that the CPU can access directly

• Instructions and data must be loaded from a secondary storage to main memory before the CPU can operate on them

• The hardware needs to provide memory access protection

• Protect the OS from access by user processes

• Protect user processes from one another

• Each process should have a separate memory space

• This is fundamental for concurrent execution

• To separate memory spaces, we need the ability to determine the range of legal addresses that a process may access

• Two registers needed

• Base register holds smallest legal physical memory address

## Logical vs. Physical Address Space

• An address generated by the CPU is referred to as a logical or virtual address to specify a location in the program

• A physical address or absolute is asserted on the address bus to decode a physical location in the system

• Binding addresses at either compile or load time generates identical logical and physical addresses

• Binding addresses at execution time results in differing logical and physical addresses

• The set of all logical addresses generated by a program is a logical address space

• The set of all physical addresses corresponding to these logical addresses is a physical address space

• The run-time mapping from virtual to physical addresses is done by a hardware device called the memory-management unit (MMU)

- Simple MMU:**
  - base-register value is added to every address generated by a user process at the time the address is sent to memory (Dynamic allocation)
- Real physical addresses are abstracted from the user program that deals with logical addresses only
- The memory-mapping hardware converts logical addresses into physical addresses
- We now have two different types of addresses: logical addresses (in the range 0 to max) and physical addresses (in the range R+0 to R+max for a base value R).
- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management

### Dynamic Loading

- By default, a routine is not loaded until it is called for the first time by a running process
- To load a routine, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables. Then, control is passed to the newly loaded routine.
- Implemented through program design, no special support from the OS is required. However, operating systems may provide library routines to implement dynamic loading

### Dynamically Linked Libraries (DLL)

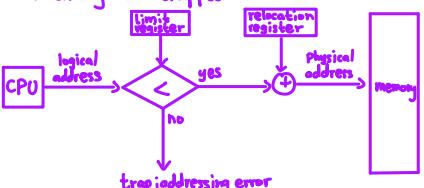
- System libraries linked to running user programs
- When a program references a routine that is a dynamic library, the loader locates the DLL and loads it into memory if necessary
- Dynamic linking generally require help from OS
  - OS needs to check and to allow sharing of the same routine by multiple processes

### Contiguous Memory Allocation

- Main memory must accommodate both the OS and the various user processes - in the most efficient way possible
- Main memory divided into partitions - one for the OS and one for the user processes
  - OS kernel can be placed at low memory together with the interrupt vector, or at high memory
  - Each process is contained in a single contiguous section of physical memory

### Contiguous Memory Allocation: Memory Protection

- Prevent a process from accessing memory that it does not own
- Use relocation register (base register) together with a limit register
  - The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses
  - Each logical address must fall within the range specified by the limit register
  - The MMU maps the logical address dynamically by adding the value in the relocation register
  - This mapped physical address is sent to memory or trapped if violation is found



### Variable Partition

- Variably sized partitions that may contain exactly one process
- OS keeps a table of available/occupied memory
  - Initially, all memory is available for user processes
  - But, eventually, memory contains a set of holes of various sizes

- The general dynamic storage-allocation problem concerns how to satisfy a request of size  $n$  from a list of free holes
- There are many solutions to this problem - most commonly used to select a free hole from the set of available holes:
  - first fit: Allocate the first hole that is big enough
    - searching either from the beginning of holes or from where the previous first-fit search ended
    - stop searching as soon as a free hole that is large enough is found
  - best fit: Allocate the smallest hole that is big enough
    - Search the entire list
    - This strategy produces the smallest leftover hole
  - worst fit: Allocate the largest hole
    - Search the entire list
    - This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach

### External Fragmentation

- Available memory contains holes that together have enough free space to satisfy a request, but the available spaces are not contiguous to fit the request
  - Storage is fragmented into a large number of small holes
  - If all these small pieces of memory were in one big free block instead, we might be able to run several more processes
- First-fit and Best-fit strategies both suffer from external fragmentation

### Compaction

- It is one solution to the external fragmentation problem
- Shuffle the memory contents so as to place all free memory holes together in one large block
  - It is possible only if relocation is dynamic and is done at execution time
  - Requires moving the process and data and then changing the base register to reflect the new base address
- When compaction is possible, we must determine its cost
- The simplest compaction algorithm is to move all processes toward one end of memory, producing one large hole of available memory
- Compaction can be expensive

### Internal Fragmentation

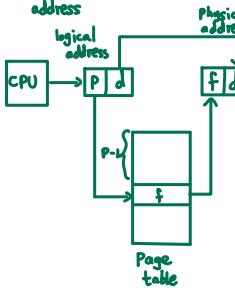
- Fragmentation that is internal to a partition
  - Consider a memory hole of 18,464 bytes and we want to fit a 18,562 bytes process
  - If we allocate exactly the requested block, we are left with a small external hole of 2 bytes only
  - The overhead to keep track of this hole will be substantially larger than the hole itself
  - The general approach to avoid this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size
  - Then the memory allocated to a process may be slightly larger than the requested memory
  - The difference between these two numbers is internal fragmentation - unused memory that is internal to a partition
  - Internal hole is reclaimed only when the process terminates

### Paging

- A memory management scheme that permits a process's physical address space to be noncontiguous
- To avoid external fragmentation and the associated need for compaction
- Used for most operating systems
- Implemented through cooperation between the OS and the computer hardware

### Basic Paging Method

- Break physical memory into fixed-sized blocks called frames
- Break logical memory into blocks of the same size called pages
- A process's logical pages are loaded into any available memory frames
- Every address space is now separate from the physical address space
- Every address generated by the CPU is divided into two parts: a page number ( $p$ ) and a page offset ( $d$ )
  - The page number is used as an index into a per-process page table
  - The page table contains the base address of each frame in physical memory, and the offset is the location in the frame being referenced
  - Thus, the base address of the frame is combined with the page offset to define the physical memory address



- The MMU takes the following steps to translate a logical address generated by the CPU to a physical address

- Extract the page number  $p$  and use it as an index into the page table
- Extract the corresponding frame number  $f$  from the page table
- Replace the page number  $p$  in the logical address with the frame number  $f$
- As the offset  $d$  does not change, it is not replaced, and the frame number and offset now comprise the physical address
- The page size is defined by the hardware

- The size of a page is a power of 2, typically varying between 4kB and 1GB per page
- If the size of the logical address space is  $2^m$ , and the page size is  $2^n$  bytes, then the high-order  $m-n$  bits of the logical address designate its page number and the  $n$  low-order bits designate the page offset

- The OS must maintain a page table for each process
- A list of free frames is also maintained
- When a process arrives in the system to be executed, its size is examined
- Each page of the process needs one frame

### Paging Observations

- No external fragmentation
- We may have some internal fragmentation
  - Frames are allocated as units
  - If the memory requirements of a process do not happen to coincide with page boundaries, the last frame may not be completely full (internal fragmentation)
- In the worst case, a process would need  $n$  pages plus 1 byte. It would be allocated  $n+1$  frames, resulting in internal fragmentation of almost an entire frame
- Internal fragmentation averages one-half page per process
- This consideration suggests that small page sizes are desirable
- However, overhead is involved in each page-table entry, and this overhead is reduced as the size of the pages increases
- Generally, page sizes have grown over time as processes, data sets, and main memory have become larger
- Today, pages are typically either 4kB or 8kB in size, and some systems support even larger page sizes
- Some CPUs and operating systems even support multiple page sizes
- Runtime address translation is easily accomplished by hardware:

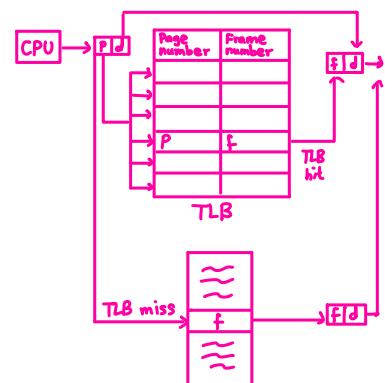
- A logical address ( $p, d$ ) is translated into a physical address ( $f, d$ ) by indexing the page table and appending the same offset  $d$  to the number of the frame  $f$
- A program can be run on different hardware using different page sizes
  - What changes is the interpretation of the bits by the addressing mechanism
- Paging provides a clear separation between the programmer's view of memory and the actual physical memory
- The programmer views memory as one single space, containing only this one program
- In fact, the user program is scattered throughout physical memory, which also holds other programs
- Mapping (translation of logical to physical address) is hidden from the programmer and is controlled by the OS
- User program cannot access memory address outside of its page table, and the table includes only those pages that the process owns
- The OS manages physical memory and is aware of the allocation details
  - Which frames are allocated
  - Which frames are available
  - How many total frames there are, and so on
- This information is generally kept in a single, system-wide data structure called a frame table
  - One entry for each physical page frame

### Hardware Support for Paging

- Page address translation is performed by hardware mechanisms
- However, if the page table is in the main memory, each logical address causes at least two references to memory
  - One for the page table entry
  - The other for the actual data access
- Thus, memory access is slowed by a factor of two - intolerable delay
- If the page table is moved to CPU registers, we gain in speed but lose in space
- We can have a mixed solution:
  - keep the page tables in main memory
  - Move most used addresses to CPU registers

## Translation Look-Aside Buffer

- TLB is a small, fast lookup hardware cache - associative, high-speed memory
- Each entry in the TLB consists of 2 parts:
  - A key (or tag) and a value
    - key is page number
    - value is frame
- When the associative memory is presented with an item, the item is compared with all keys simultaneously
  - However, the TLB must be kept small. It is typically between 32 and 1024 entries in size
  - If the item is found, the corresponding value field is returned
- The TLB contains only a few of the page-table entries
  - A logical address is generated by the CPU
  - The MMU first checks if its page number is present in the TLB
  - If it is found (TLB hit), its frame number is immediately available and is used to (quickly) access memory
  - If the page number is not in the TLB (TLB miss), proceed according to the basic method
    - When the frame number is obtained, we can use it to access memory
    - Then the page and frame numbers are added to the TLB to replace another entry



- Some TLBs support address-space identifiers (ASIDs) in each TLB entry
  - ASID is used to allow the TLB to contain entries for several different processes simultaneously and to identify each process for protection purpose
  - Each page number must be matched to the ASID of running process - if no match, then it is treated as TLB miss
- If the TLB does not support separate ASIDs, then every time a new page table is selected, the TLB must be flushed
  - This ensures that the next executing process does not use the wrong translation information

Otherwise, the TLB could include old entries that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process

## Effective Memory Access Time

- The percentage of times that the page number of interest is found in the TLB is called the hit ratio
  - To find the effective memory-access time, we weight the case by its probability
- $$EMAT = \text{Hit Ratio} \cdot \text{TLB hit memory access time} + (1 - \text{Hit Ratio}) \cdot \text{TLB miss memory access time}$$

## Paging Protection

- Protection bits can be associated with each frame in the page table
  - This approach can be easily expanded to provide a finer level of protection
  - A hardware can be implemented to provide read-only, read-write, or execute-only protection

We can use a valid-invalid bit to check for access to the process memory:

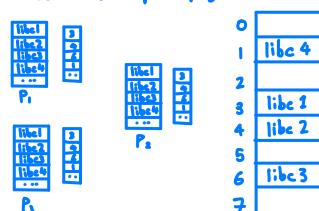
- Valid bit: the associated page is in the process's logical address space
- Invalid bit: the associated page is not in the process's logical address space
- OS can set this bit for each page to allow or disallow access to the page, then trap any illegal addresses
- This may result in internal fragmentation
- Rarely does a process use all its address range

Many processes use only a small fraction of the address space available to them

- It would be wasteful in these cases to create a page table with entries for every page in the address range
- Most of the table would be unused but would take up valuable memory space
- Solution: Some systems provide hardware-page-table length register (PTLR), to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process. Failure of this test causes an error trap to the OS

## Shared Pages

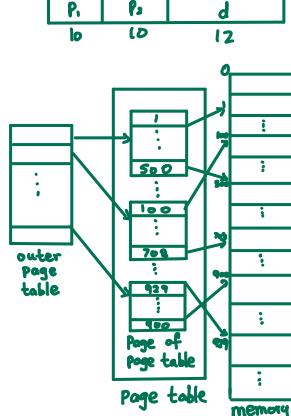
- An advantage of paging is the possibility of sharing common code
- Shared code
  - One copy of read-only code shared among processes
- Private code and data
  - Each process keeps a separate copy of private code and data
- The diagram below shows three processes sharing the pages for the standard C library libc
  - Reentrant code is non-self-modifying code; it never changes during execution. Thus, two or more processes can execute the same code at the same time
  - Only one copy of the standard C library need to be kept in physical memory



## Two-Level Paging

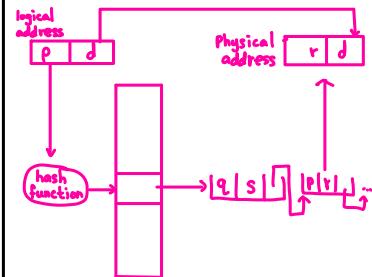
- Page itself is also paged
- Outer page table entries point to the pages of the real page table
- Consider a system with a 32-bit logical address and page size 4kB
  - A page number consisting of 20 bits and a page offset of 12 bits
  - Since we page the page table, the page number is further divided into two 10-bit each

Page number page offset



## Hashed Page Tables Algorithm

- The virtual page number in the virtual address is hashed into the hash table
- The virtual page number is compared with field 1 in the field element in the linked list
- If there is a match, the corresponding page frame (field 2) is used to form the desired physical address
- If there is no match, subsequent entries in the linked list are searched for a matching virtual page number



## Inverted Page Table

- Usually, each process has an associated page table that has one entry for each page that the process is using

A drawback is that each page table may consist of millions of entries that may consume large amounts of memory just to keep track of how memory is being used

- Inverted page tables have only one entry for each physical frame of memory

Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns the page

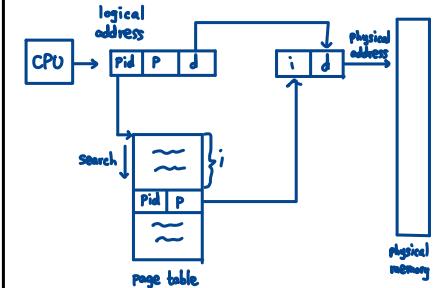
Only one page table is in the system, and it has only one entry for each page of physical memory

But how do we translate from page number to frame number?

Search the inverted page table until we find a match for the page number and process ID

- Inverted page tables often require that an address space identifier be stored in each entry of the page table

Storing the address-space identifier ensures that a logical page for a particular process is mapped to the corresponding physical page frame



This scheme decreases the amount of memory needed to store each page table

But it increases the amount of time needed to search the table when a page reference occurs

Because the inverted page table is sorted by physical address, but lookups occur on virtual addresses, the whole table might need to be searched before a match is found - This would take far too long

To alleviate this problem, we use a hash table, to limit the search to one, or at most a few page table entries

Each access to the hash table adds a memory reference to the procedure - one virtual memory reference requires at least two real memory loads

TLB searched first

## Hashed Page Tables

- For handling address spaces larger than 32 bits

Hash value being virtual page number

Each entry in the hash table contains a linked list of elements that hash to the same location

Each element consists of 3 fields:

- virtual page number
- value of mapped page frame
- pointer to next element

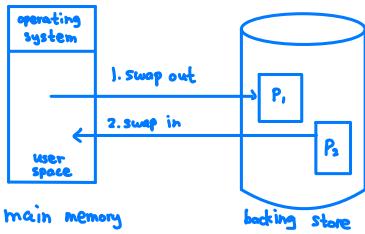
Virtual page numbers are compared in this chain searching for a match

## Swapping

- Used to increase the degree of multiprogramming in a system
- So, the total physical address space of all processes can exceed the real physical memory of the system
- A process, or a portion of a process, can be swapped temporarily out of memory to a backing store (secondary storage area for process swapping) and then brought back into memory for continued execution

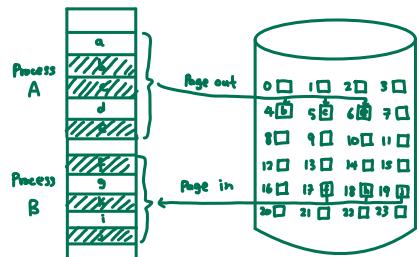
## Standard Swapping

- Standard swapping involves moving entire processes between main memory and a backing store
- For a multithreaded process, all per-thread data structures must be swapped as well
- The OS must also maintain metadata for processes that have been swapped out, so they can be restored when they are swapped back into memory
- The advantage of standard swapping is that it allows physical memory to be oversubscribed, so that the system can accommodate more processes than there is actual physical memory to store them
- Idle or mostly idle processes are good candidates for swapping; any memory that has been allocated to these inactive processes can then be dedicated to active processes



## Swapping with Paging

- Standard swapping is generally no longer used in contemporary operating systems
- The amount of time required to move entire processes between memory and the backing store is prohibitive
- Paging only swaps some pages of a process - rather than an entire process
- Still allows physical memory to be oversubscribed but does not incur the cost of swapping entire processes, as presumably only a small number of pages will be involved in swapping
- A page out operation moves a page from memory to the backing store; the reverse process is known as a page in



## Swapping on Mobile Systems

- They typically do not support swapping in any form
- Mobile devices generally use flash memory for nonvolatile storage - space constraint is one reason to avoid swapping
- Other reasons include the limited number of writes that flash memory can tolerate before it becomes unreliable and the poor throughput between main memory and flash memory in these devices
- Instead of using swapping, when free memory falls below a certain threshold:
  - Ask applications to optimize the use of memory
  - Applications that fail to free up sufficient memory may be terminated by the OS
  - However, before terminating a process, some OS writes its application state to flash memory so that it can be quickly restored

## Virtual Memory

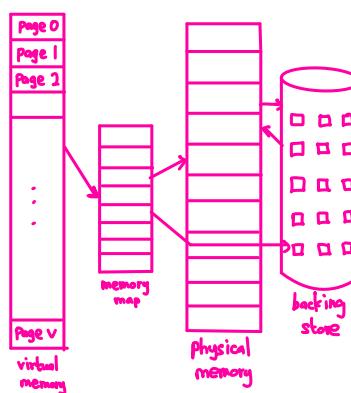
- A technique that allows the execution of processes that are not completely in memory
- Virtual memory abstracts main memory into an extremely large, uniform array of storage - it frees programmers from the concerns of memory-storage limitations
- Virtual memory allows processes to share files and libraries, and to implement shared memory
- Provides efficient mechanism for process creation
- Not easy to implement and may substantially decrease performance if used carelessly
- Instructions to be executed must be in physical memory

- One approach is to place the entire logical address space in physical memory
- Dynamic linking can help to ease this restriction but requires special precautions and extra work by the programmer
- In many cases, the entire program is not needed in main memory
  - Programs with code to handle error conditions - in practice almost never executed
  - Arrays, lists, and tables allocated more memory than needed
  - Certain options and features of a program may be used rarely

- Motivation: virtual memory has the ability to execute a program that is only partially in memory. It would confer many benefits:
- Possibility to run programs larger than the main memory
- Less physical memory to each program

## Logical / Physical Memory Separation

- Virtual memory involves the separation of logical memory as perceived by developers from physical memory - allows an extremely large virtual memory to programmers when physical memory is relatively small



## Virtual Address Space

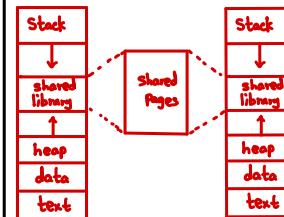
A process' virtual address space refers to the logical view of how a process is stored in memory - say address 0 - and exists in contiguous memory

- It is up to memory-management unit (MMU) to map logical pages to physical page frames in memory
- The heap grows upward in memory as it is used for dynamic memory allocation
- The stack grows downward in memory through successive function calls
- The hole in the middle is part of the VAS but will require actual physical pages only if the heap or stack grows. VAS that include holes are known as sparse address spaces; the holes can be filled by growing stack/heap or dynamically linked libraries



## Virtual Memory Sharing

- Virtual memory allows files and memory sharing between processes
- The actual frames in physical memory are shared by all the resources
- Virtual memory allows one process to create a region of memory that it can share with another process - each process considers it part of their VAS



## Demand Paging

- Demand paging: In memory management, bringing in pages from storage as needed
- Locality of reference: The tendency of processes to reference memory in patterns rather than randomly
  - Programs with code to handle error conditions - in practice almost never executed
  - Arrays, lists, and tables allocated more memory than needed
  - Certain options and features of a program may be used rarely
- Page fault: In virtual memory, a fault resulting from a reference to a non-memory-resident page
- Reference string: A string of memory references
- While a process executes, some pages are in main memory and the rest are in secondary storage - hardware support is needed to distinguish between the two:
  - Use valid-invalid bit scheme
  - When the bit = valid, the associated page is both legal and in main memory
  - If the bit = invalid, the page either is not valid or is valid but is currently in secondary storage
  - Marking a page invalid will have no effect if the process never attempts to access that page

## Handling Page Fault

- The valid-invalid bit is initialised to 0 for all pages
- Access to a page marked invalid is trapped by the operating system, causing a page fault
- The OS would then check if the reference is legal or not. If it is illegal, the process is terminated
- If the reference is legal, then the kernel finds a free frame in physical memory and schedules a secondary storage operation to read the desired page into the newly allocated frame. Then the frame number is loaded into the page table entry
- Then restart the instruction that was interrupted by the trap
- Associate a dirty bit to each page or frame
- A page dirty bit is set by the hardware whenever any byte in the page is written into, indicating that the page has been modified when selected for replacement
  - If dirty bit is set, it means that the page has been modified (updated) since it was read in from secondary storage - must write it to storage
  - If dirty bit is not set, the page has not been modified since it was read into memory - no need to write it to storage
- This technique also applies to read-only pages
- Can significantly reduce the time required to service a page fault - reduces I/O time by one-half

## Demand Paging Performance

- Pure demand paging: never bring a page into memory until it is required
  - In worst case, a process starts with zero pages in memory
  - It immediately faults from the page, then it is brought into main memory
  - The process continues to execute, faulting as necessary until every page that it needs is in memory. At that point, it can execute with no more faults
- Some processes could access several new pages causing multiple page faults for instruction - unacceptable performance
- Secondary memory should be a high-speed disk or NVRAM device. It is known as the swap device, and the section of storage used for this purpose is known as swap space

## Effective Access Time

- Let p be the probability of a page fault (ospi). We should expect p to be close to zero - that is, we would expect to have only a few page faults. The effective access time is then
 
$$\text{effective access time} = (1-p) \cdot \text{avg access time} + p \cdot \text{page fault time}$$
- The effective access time is directly proportional to the page-fault rate.
- It is important to keep the page-fault rate low in a demand-paging system. Otherwise, the effective access time increases, slowing process execution dramatically

## Page Replacement

- In a situation where all memory is in use, but an executing process needs a page from the secondary storage (swap space)
- The OS has several options at this point
  - Terminate the process (not the best choice)
  - Use standard swapping to reduce multiprogramming level - it is no longer used by most OS due to the overhead of swapping entire processes
  - Combine swapping pages with page replacement
- If there are no free frames, we find one that is not currently being used and free it by:
  - Moving it to swap space
  - We can now use the freed frame to hold the page for which the process faulted
- Modify the page-fault routine to include page replacement:
  - Find the location of the desired page on secondary storage
  - Find a free frame
    - If there is a free frame, use it
    - If there is no free frame, use a page-replacement algorithm to select a victim frame
    - Write the victim frame to secondary storage (if necessary); change the page and frame tables accordingly
  - Read the desired page into the newly freed frame; change the page and frame tables
  - Continue the process from where the page fault occurred

## Dirty Bit

- If no frames are free in the main memory, two page transfers (page-out and page-in) are required - page-fault service time doubled and effective access time increased accordingly
- To reduce this overload, use a modify bit (or dirty bit)
  - Associate a dirty bit to each page or frame
  - A page dirty bit is set by the hardware whenever any byte in the page is written into, indicating that the page has been modified when selected for replacement
    - If dirty bit is set, it means that the page has been modified (updated) since it was read in from secondary storage - must write it to storage
    - If dirty bit is not set, the page has not been modified since it was read into memory - no need to write it to storage
- This technique also applies to read-only pages
- Can significantly reduce the time required to service a page fault - reduces I/O time by one-half

## Algorithm Selection Criteria

- We want to select a page-replacement algorithm with the lowest page-fault rate
- As the number of available frames in main memory increases, the number of page faults drops to some minimal level
  - Adding physical memory increases the number of available frames



## FIFO Page-Replacement Algorithm

- Simplest algorithm - associates time-in to main memory to each page
- When a page must be replaced, the oldest page is chosen
- FIFO queue holds all pages in main memory - replace the page at the head - insert brought in page (to main memory) at the tail
- The algorithm is easy to understand and program - but its performance is not always good
- A replaced page could contain a heavily used variable that was initialized early and is in constant use - a fault occurs almost immediately to retrieve this page
  - Thus, a bad replacement choice increases the page-fault rate - slows execution

## Belady's Anomaly

- Belady's Anomaly is when increasing the number of available frames actually increases the number of page fault
- As a result, the assumption of improving a process performance by giving it more memory is not always true
- For some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases

## Optimal Page Replacement Algorithm

- Never suffer from Belady's anomaly
- Replaces the page that will not be used for the longest period of time
- Guarantees lowest possible page-fault rate
- Hard to implement

## LRU Page Replacement Algorithm

- An approximation of optimal replacement algorithm
  - use recent past as an approximation of the near future
- Associates last use of each page when a page must be replaced
- Replaces the page that has not been used for the longest period of time
  - Not necessarily the same as FIFO's "oldest page"
  - For LRU it is in terms of last page reference
- By the principle of locality, this should be the page least likely to be referenced in the near future
- Performs nearly as well as optimal algorithm
- Often used as a page-replacement algorithm and considered to be good
- Major problem: how to implement?
  - Require substantial hardware assistance
  - Problem is to determine an order for the frames defined by the time of last use
- Two implementations are feasible
  - Counters
  - Stack

## LRU - Counters Algorithm

- Associate a time-of-use field to each page-table entry - updated for every memory reference - always have the 'time' of the last reference to each page
- Replace the page with the smallest value
- Requires a search of the page table to find the LRU page and a write to memory for each memory access
- The times must also be maintained when page tables are changed
- Overflow of the clock must be considered

## LRU - Stack Algorithm

- keep a stack of page numbers - whenever a page is referenced, it is removed from the stack
- The most recently used page is always at the top of the stack
- The least recently used page is always at the bottom

- Requires removing elements from the middle of the stack - a doubly linked list with a head pointer and a tail pointer
- Requires changing six pointers at worst to maintain the stack
- Each update is a little more expensive, but there is no search for a replacement
- This approach is particularly appropriate for software or microcode implementations of LRU replacement

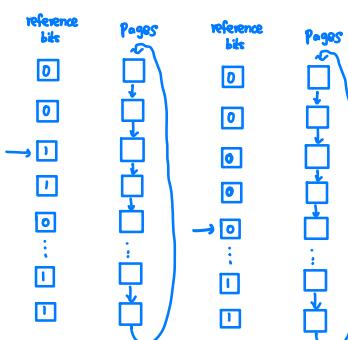
## LRU Approximation Page Replacement

- Most practical - implemented by most operating systems
- LRU approximation page replacement - associate a reference bit to each entry in the page table - initialized to 0 - then set to 1 with each page reference. After some time, we can determine which is not being used - order of use not maintained
- Additional-reference-bits algorithm - similar but adding more bits to help maintaining the order

## Second-Chance (Clock) Algorithm

- Is a FIFO with a reference bit, if the reference bit is 1 then give the page a second chance (reset bit = 0) and move on to select the next FIFO page

- Implementation: As a circular queue with pointer indicates which page is to be replaced next:
  - When a frame is needed, the pointer advances until it finds a page with a 0-reference bit, then is replaced with a new page in the circular queue in that position
  - As the pointer advances, it clears the reference bits - in the worst case, when all bits are found to be 1, the pointer cycles through the whole queue, giving each page a second chance
- Second-chance replacement degenerates to FIFO replacement if all bits are set



- Each block of memory has a reference bit
- When the page has been referenced to the bit is set to 1 by the hardware
- The OS can examine this bit
  - If 0, page can be replaced
  - If 1, change to 0

## Enhanced Second-Chance Algorithm

- Considering the reference bit and the modify (dirty) bit as an ordered pair - 4 possible classes:
  - (0,0) neither recently used nor modified - best victim page to replace
  - (0,1) not recently used but modified - not quite as good, because the page will need to be written out before replacement
  - (1,0) recently used but clean - probably will be used again soon
  - (1,1) recently used and modified - probably will be used again soon, and the page will need to be written out to secondary storage before it can be replaced

- At page fault - examine the class - replace first encountered page in the lowest nonempty class
- May have to scan the circular queue several times before picking a victim
- The major difference between this algorithm and the simpler check:
  - We give preference to pages that haven't been modified
  - reduce I/Os required

## Counting-Based Page Replacement

- keep a reference number counter - made to each page and develop the following two schemes

- Least Frequently Used (LFU) - an actively used page have a large reference count - replace the page with the smallest count

- Problem: A page heavily used for an initial phase has a large count - but then is never used again - no longer needed but persist in memory

- Solution: Shift counts right by 1 bit at regular intervals (binary division), forming an exponentially decaying average usage count

- Most frequently used (MFU) - the smallest count page was probably just brought in and has yet to be used

- Neither MFU nor LFU replacement is common

- The implementation of these algorithms is expensive, and they do not approximate OPT replacement well

## Page-Buffering Algorithms

- Not a page-replacement by itself - but a procedure that can be added to a specific page-replacement algorithm - pages to be replaced are kept in main memory for a while to guard against poorly performing algorithms such as FIFO

- keep a pool of free frames
- At page fault, a victim frame is chosen - not written out yet
- The requested page is read into a free frame from the pool
- The process can restart ASAP
- When later the victim is written out, its frame is added to the free-frame pool

- One variation is to maintain a list of modified pages (victims) - write a page to secondary storage whenever the paging device is idle and reset its modify bit to 0

- This scheme increases the probability that a frame will be free when it is selected for replacement

- Another variation is to keep a pool of free frames but to remember which page was in each frame

- When a frame is written to secondary storage - actually it is copied out - so the old page can be reused directly from the free-frame pool if it is needed before that frame is reused

- When a page fault occurs, we first check whether the desired page is in the free-frame pool

- Some versions of the UNIX system use this method in conjunction with the second-chance algorithm
- Can be a useful augmentation to any page-replacement algorithm, to reduce the penalty incurred if the wrong victim page is selected

## Allocation of Frames

- To execute, a process needs a minimal number of memory frames

- It is easy to see that when a process runs few frames, it will generate an excessive number of page faults and be slowed considerably

- How to ensure that a process is allocated its minimum

- Equal allocation: each process has an equal portion of physical memory

- Proportional allocation: each process is allocated according to its size

- The criteria should be more related to the pages needed

## Global vs. Local Allocation

- The way frames are allocated to the various processes is page replacement

- With multiple processes competing for frames, we can classify page-replacement algorithms into 2 broad categories:

- Global replacement: allows a process to select a replacement frame from all frames

- Local replacement: each process selects from only its own set of allocated frames

## Threshing

- Paging memory at a high rate - insufficient physical memory to meet virtual memory demand

- A process is threshing if it is spending more time paging (waiting in I/O queues) than executing - it no longer can complete any useful work

- A process needs a minimum number of pages to run efficiently

- In multiprogramming

- Number of pages needed by a running process hit a point where it exceeds the number of available physical memory frames
- It starts faulting and taking frames away from other processes
- So they also fault
- Heavy page swap in and out
- CPU utilization decreases as many processes are waiting in I/O queues
- CPU scheduler seeks to increase the degree of multiprogramming
  - introduce new processes in
  - taking frames away from already 'stuck' processes
  - causes more faulting
  - more new processes introduced
- Threshing occurs

## Limiting the Effects of Threshing

- Implement a local replacement algorithm - a process select from only its own set of allocated frames
  - If the process fault - select from only its own set of frames for replacement
  - Then it threshes, it cannot spread to others by stealing from their frames
  - Thus, threshing is localized to the threshing process
- However, the problem is "localized" but not entirely solved
- The affected process will stay longer in I/O queue - page fault average service time will increase for other processes that are not threshing

## Locality Model

- To prevent threshing, the OS must provide a process with as many frames as it needs - but how to know it "needs"?
- One strategy starts by looking at how many frames a process is actually using - the locality model of process execution
- States: as a process executes, it moves from locality to locality - may overlap
- A locality is a set of pages that are actively used together
- All programs will exhibit this basic patterned memory reference structure
- Allocate enough frames to a process to accommodate its current locality - it will fault until all locality pages are in main memory; then, it will not fault again until it changes localities
- Enough frames must accommodate the size of the current locality - if not - the process will thresh

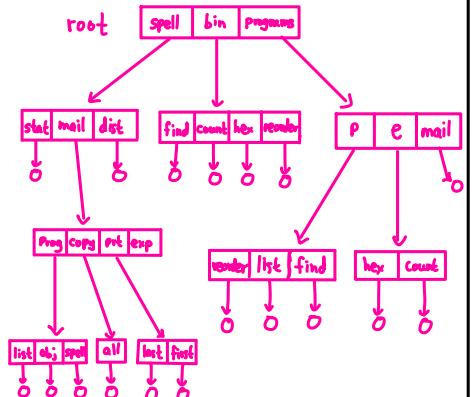
## Working-Set Model

- Based on the assumption of locality - an approximation of the program's locality
- Uses a parameter,  $\Delta$ , to define the working-set window (window of time)
  - Examine the most recent  $\Delta$  page references
  - The set of pages in the most recent  $\Delta$  page references is the working set
  - If a page is in active use, it will be in the working set
  - If it is no longer being used - it will drop from the working set  $\Delta$  time units after its last reference
- The accuracy of the working set depends on its size (the selection of  $\Delta$ )
  - If too small, it will not encompass the entire locality
  - If too large, it may overlap several localities
- If WSS is the working-set size for each process in the system, the total demand for frames is  $D = \sum WSS$



## Tree - Structured Directories

- Most common - allows sub-directories in a directory
- Has root directory
  - allows users to create subdirectories and organize their files accordingly
  - every file in the system has a unique path name
- Efficient searching
- Grouping capability

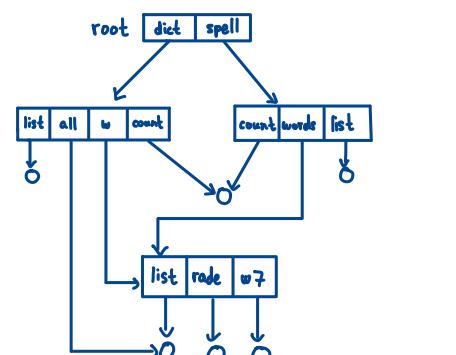


In normal use, each process has a current directory - it should contain most of the files that are of current interest to the process

- When reference is made to a file, the current directory is searched
- If a file is needed that is not in the current directory, then the user usually must either specify a path name or change the current directory to be the directory holding that file
- A path can be absolute or relative
  - An absolute path name begins at the root and follows a path down to the specified file
  - A relative path name defines a path from current directory

## Acyclic Graph Directories

- Allows sharing subdirectories and files - allows access to a file or a subdirectory from multiple directories
- Cycles are not allowed
- Symbolic links are used to point from multiple directories to a particular directory or file
- Similar to tree + links for sharing

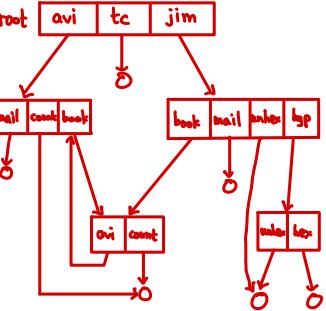


Several problems must be considered carefully

- A file may now have multiple absolute path names - distinct file names may refer to the same file
- Deletion problem - shared file by symbolic links space allocation - if a user deletes the shared file this will leave dangling pointers to the now-non-existent file and possibly to actual disk addresses - Possibly into other files in the reused space
- UNIX OS uses the following approach for hard links - keep the number of references count - adding a new link increments the count. If count > 0, then delete only the link and decrements the count - when the count is 0, then the file is deleted

## General Graph Directory

- Similar to acyclic-graph directory - cycles allowed inside a directory - offers more flexibility



As cycles are allowed - poorly designed algorithm might result in an infinite loop continually searching through the cycle and never terminating

- When crossing the graph, it is necessary to know if we land on a node already visited
- Algorithms exist to deal with these cases; however they are complicated and have significant execution times, which means that they are not always used
- A similar problem exists when we are trying to determine when a file can be deleted
- Acyclic-graph used reference count = 0 to delete the file - This does not work here due to cycles where count may not be 0 even when no more links refer to a file and it can be deleted
- Garbage collection solution is needed - requires two traversing passes to the entire file system - extremely time consuming

## Types of Access

- File owner/creator should be able to control:
  - what can be done to the file and by whom
- Types of access:
  - Read: read from file
  - Write: write or rewrite the file
  - Execute: load the file into memory and execute it
  - Append: write new information at the end of the file
  - Delete: delete the file and free its space for possible reuse
  - List: list the name and attributes of the file
  - Attribute change: changing the attributes of the file
  - Other operations, such as renaming, copying, and editing the files, may also be considered

## Access Control

- Different users may need different types of access to a file or directory
- Identity-dependent scheme access - associate an access-control list (ACL) with each file and directory
  - Each entry specifies: user names and the types of access allowed
    - Checked by OS at each access to allow or deny access accordingly
  - Advantage: enabling complex access methodologies
  - Problem: lengthy list
- Condensed version of the access list:
  - Recognize three classifications of users in connection with each file
  - Owner: the user who created the file is the owner
  - Group: a set of users who are sharing the file and need similar access is a group
  - Other: all other users in the system
- In UNIX, a three-field - owner, group, and mode - each consisting of the three bits rwx to control read access, write access, and execution
  - a user can list the content of a subdirectory only if the r bit is set in the appropriate file
  - a user can current directory to another current directory only if the x bit associated with the subdirectory is set

## File-System Structure

- To improve I/O efficiency

- I/O transfers between memory and mass storage are performed in units of blocks
- Each block on a hard disk drive has one or more sectors
- Sector size is usually 512 bytes or 4096 bytes
- NVM devices usually have blocks of 4096 bytes, and the transfer methods used are similar to those used by disk drives
- All disk blocks have fixed size, but tape blocks are variable

- Files are allocated in units of blocks. The last block is therefore rarely filled with data - Internal Fragmentation

- The file system resides in secondary storage - provides access to data on storage device as follows:

- Storing, locating, retrieving data

- It is generally composed of many different levels

- An example of a layered design is shown
- each level in the design uses the features of lower levels to create new features for use by higher levels

- I/O control level: consists of device drivers and interrupt handlers to transfer information between main memory and the disk system

- a translator

- Its input consists of high-level commands
- Its output of low-level, hardware specific instructions to interface the I/O device to the rest of the system

- Basic file system: issues some commands based on logical block addresses to the appropriate device driver to read and write blocks on the storage device

- I/O request scheduling

- Manages the memory buffers (mass data transfer) and caches that hold various file systems, directory, and data blocks

- File-organization module: knows about files and their logical blocks. Each file's logical blocks are numbered from 0 or 1 through N

- Includes the free-space manager, which tracks unallocated blocks and provides those blocks to the file-organization module when requested

- Logical file system: manages metadata information that includes all of the file-system structure except the actual data (or contents of the files)

- Manages the directory structure to provide the file-organization module with the information the latter needs, given a symbolic file name

- Maintains file structure via file-control blocks. A file-control block (FCB) (an inode in UNIX file systems)

- Inode contains information about the file, including ownership, permissions, and location of the file contents

- Responsible for protection

## A Typical File-Control Block

- To create a new file, a process calls the logical file system that allocates a new FCB (inode)

- Update the new file name and FCB ID into the appropriate directory

- A typical FCB as follows

file permissions
file dates
file owner/group, ACL
file size
file data blocks or pointers

## File Directory Implementation

- Linear list: of file names with pointers to data blocks

- Simple to program but time-consuming to execute

- Hash table: linear list with hashed data structure

- The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list

- It can greatly decrease the directory search time

- Insertion and deletion are also fairly straightforward

- Some provision must be made for collision situations in which two file names hash to the same location

- The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size

## Variations / Solutions

- Use a chained-overflow hash table - each hash entry can be a linked list

- Search may be somewhat slowed through a linked list of colliding table entries

- Still, this method is likely to be much faster than a linear search through the entire directory

## File Allocation Methods

- The main problem is how to allocate space to files on secondary storage so that storage space is utilized effectively, and files can be accessed quickly

- There are 3 major methods of allocating secondary storage space in wide use

- Contiguous

- Linked

- Indexed

- Each method has advantages and disadvantages

- Although some systems support all three, it is more common for a system to use one method for all files within a file-system type

## Contiguous Allocation

- Each file occupies a set of contiguous blocks on the device

- File is defined by the address of the first block and length (l to bth-l)

- The file directory entry is a pointer to the starting block and the length

- Support both sequential and random access

- Limitations - therefore not used in modern file systems

- Suffers from external fragmentation

- Difficulty in finding space for a new file

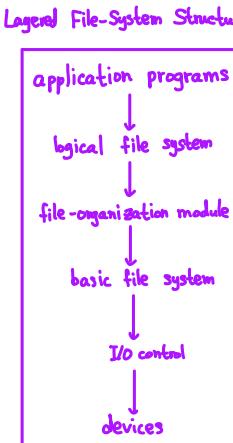
- Difficulty defining the space needed for a file - if too little allocated then the file cannot be extended - especially with a best-fit allocation strategy

- Relocating the file to a larger hole is costly

- Compaction is required periodically to ease external fragmentation accumulation

- Compaction is to copy an entire file system into another device
  - Format the original then restore it back to the original device in one large contiguous allocation

## Layered File-System Structure



## Linked Allocation

- Each file is a linked list of storage blocks - may be scattered anywhere on the device
- The directory contains a pointer to the first and last blocks of the file
- Each block contains a pointer to the next block
- Thus, if each block is 512 bytes in size, and a block address requires 4 bytes, then the user sees blocks of 508 bytes (block pointers are not accessible by the user)
- To create a new file - create a new entry in the directory
  - each directory entry has a pointer to the first block of the file - initialized to null (empty file) - The size field is also set to 0
  - a write to the file causes the free-space management system to find a free block, and this new block is written to and is linked to the end of the file
  - a read operation needs to follow the pointers from block to block

- Advantages:**
  - no external fragmentation
  - easy to grow files

## Drawbacks:

- sequential access only - always start at the beginning of the file and follow the pointers until to get to the needed block - inefficient to support a direct-access
- wasted space for pointers
- the file might be spread over all disc, so even reading it sequentially might be inefficient
- Reliability problem - what would happen if a pointer was lost or damaged? Or any bug that might result in picking up the wrong pointer?
- a redundancy can be implemented but will add more overhead

## Indexed Allocation

- Similar to paging - fixes the linked allocation's direct access efficiency by bringing all the pointers into one location called the index block
- Each file has its own index block - an array of storage-block addresses
- The  $i$ th entry in the index block points to the  $i$ th block of the file
- The directory contains the address of the index block
- To find and read the  $i$ th block, we use the pointer in the  $i$ th index-block entry
- Newly created files have all IB pointers set to null (-1)
- When data needs to be first written to the file, an  $i$ th block is obtained from the free-space manager and its address overwrites the -1 in the  $i$ th IB entry

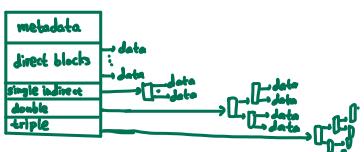
- Advantages:**
  - supports direct access, without suffering from external fragmentation

## Drawbacks:

- suffer from wasted space in index table blocks for relatively small files - consider a small file with only one or two non-null pointers

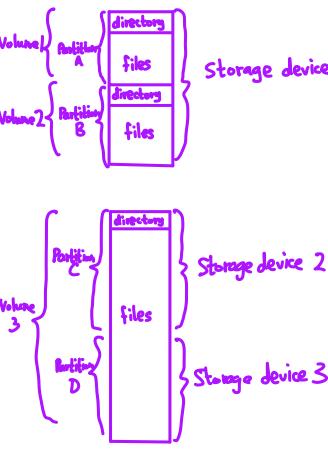
## A mechanism is needed to deal with the issue of file-size variation

- Linked scheme: to link together several index blocks - the next address (the last word in the first index block) is null (for a small file) or is a pointer to another index block (for a large file)
- Multilevel index: a variant of linked representation uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks - more levels can be added as needed
- Combined scheme: used in UNIX - inode points directly to small files (direct blocks) - other inode pointer points to indirect blocks that can be different level of addressing until we get to the actual data
- An indirect block contains the address of a block that contains the addresses of blocks... that contain pointers to the actual data blocks



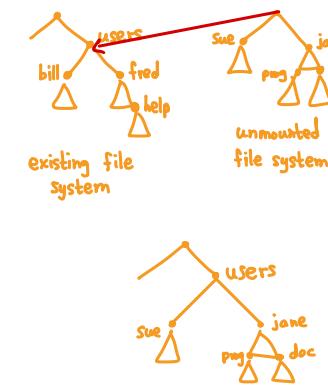
## Typical Storage Device Organization

A general-purpose computer system can have multiple storage devices, and those devices can be sliced up into partitions, which holds volumes, which in turn hold file systems - a volume may span multiple partitions as well



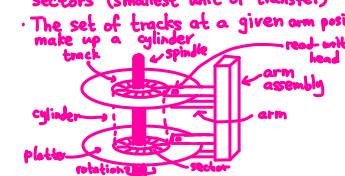
## Computer File Systems Mounting

- Typically, thousands, millions, even billions of files within a computer, stored on random-access storage devices
- Different file systems needed best adapted to each particular storage device
- A file system must be mounted before it can be available to processes
  - OS is given the name of the device and the mount point - the location within the file structure where the file system is to be attached - typically it is an empty directory
  - Next, OS verifies that the device contains a valid file system - asking the device driver to read the device directory and verifying that the directory has the expected format
  - Finally, the OS notes in its directory structure that a file system is mounted at the specified mount point
- Windows: automatically mounts into drive letters
- UNIX: explicit mount operation, can mount anywhere
- File mounting illustrated:



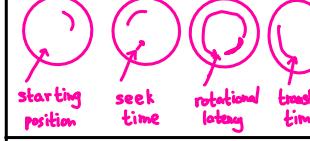
## Hard Disk Drives (HDDs)

- Each disk both platter flat surfaces are covered with a magnetic material
- Data stored/read magnetically on the platters - by detecting the magnetic pattern on the platters
- R/W head on each platter surface are all attached to the disk arm that moves as a single structure
- The surface of a platter is logically divided into circular tracks
- Tracks are subdivided into fixed-sized sectors (smallest unit of transfer)
- The set of tracks at a given arm position make up a cylinder



The positioning time or random-access time consists of 2 parts:

- Seek time:** the required time for the disk arm to move to the appropriate cylinder - the track at a given arm position of the HDD
- Rotational latency:** the time necessary for the desired sector to rotate to the disk head
- Rotation speed of platters relates to transfer rates between the HDD and the computer**



## Nonvolatile Memory Devices

- Flash-memory-based NVM is frequently used in a disk-drive-like container, in which case it is called a solid-state disk (SSD) - NAND semiconductors
- Other storage of NVM family like: USB drive, SD and micro SD cards

- Advantage: no moving mechanical parts, much faster data transfer rate than HDD also smaller and much lighter - more energy-efficient
- Drawback: NAND semiconductors deteriorate with every erase cycle, and after around 100,000 program-erase cycles - the cells no longer retain data - lifespan is measured in Drive Writes Per Day: Measure of times the drive capacity can be written per day
- Two major limitations with NVM:
  - Cannot be overwritten - must remain fixed
  - wear leveling: to select all NAND cells over time as write targets to avoid premature media failure due to wearing out a subset of cells

- The NVM device controller should manage the writing/erasing issue and at same time tries to level the wear across the entire disk by using various algorithms to place data on less-erased blocks so that subsequent erases will happen on those blocks rather than on the more erased blocks, leveling the wear across the entire device
- First, write data to an empty page - then later, if any update to the page data is needed then write it to another empty page (garbage collection and overwriting) and mark the first (old one) invalid - invalid data locations cannot be reused until after they get erased

- Similar to SCAN except that it fast returns to the start (cylinder 0) of the disk instead of reversing direction - as if the last sector would then be contiguous to the first

## HDDs vs. NVMs

- HDDs are still the most common secondary storage device
- They are the largest random-access storage device for the price, providing terabytes of storage at a low cost
- Many devices, from external chassis to storage arrays, are designed to use HDDs, providing a wide variety of ways to use HDDs
- They maintain the same read and write performance over their lifetime, unlike NVM storage devices, which lose write performance as they age

An NVM device near its end of life due to many erase cycles generally has much worse performance than a new device

- NVM devices are increasing in size and decreasing in price faster than HDDs
- High-speed NVM devices are much faster than HDDs. Secondary storage speed has a large impact on overall system performance
  - NVM devices use less power than HDDs, making them very useful for laptops and other portable, battery-powered devices. NVM devices can also be much smaller than HDDs and thus can, for example, be surface-mounted for mobile devices like smartphones

Because NVM devices have no moving parts, they tend to be much more reliable than HDDs

## HDD Scheduling

- OS responsibilities is to use the hardware efficiently
- For HDDs, minimizing access time and maximizing data transfer bandwidth
- HDD access time has two major components
  - The seek time: movement of the arm heads to the cylinder containing the desired sector
  - The rotational latency: time for the platter to rotate the desired sector to intercept the head
  - The device bandwidth: is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer
  - Both the access time and bandwidth can be improved by managing the order in which storage I/O requests are serviced

## HDD I/O Queue

- For a multiprogramming system there is a queue for the HDD - improve performance via queue ordering
- In what order should disk operation requests be chosen to minimize total head seeks?
- With an arbitrary queue, it contains digits that represent cylinder addresses
- We must also take into consideration the starting cylinder
- In what order to execute the queries in order to minimize the total cylinder positioning times  
Via simplistic hypothesis: a displacement of 1 cylinder costs 1 unit of time

## FCFS Scheduling

- Simplest scheduling algorithm
- Serve the first cylinder in the queue first
- Intrinsically fair, but does not provide fastest service

## SCAN Scheduling

- The head scans the disk in one direction, then the opposite direction, etc, excluding requests as it passes the desired cylinder - no starvation
- Take into consideration the direction of the reading head at the start of a sequence of requests - continue in the same direction until all the requests in its path are served
- If a request arrives in the queue just in front of the head, it will be serviced almost immediately; a request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back
- When the head hits one end and reverses direction, it has a little to do since these cylinders have recently been serviced

## C-SCAN Scheduling

- Similar to SCAN except that it fast returns to the start (cylinder 0) of the disk instead of reversing direction - as if the last sector would then be contiguous to the first

## Selection of Disk-Scheduling Algorithm

- There are many disk-scheduling algorithms not included in this coverage, because they are rarely used
- Performance depends heavily on the number and types of requests
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk, because they are less likely to cause a starvation problem

## NVM Scheduling

- NVM devices do not contain moving disk heads and commonly use a simple FCFS policy
- Sequential access is optimal for mechanical devices like HDD and tape because the data to be read or written is near the read/write head
- Random-access I/O (measured in input/output operations per second) causes HDD disk head movement. So, naturally, random access I/O is much faster on NVM
- An HDD can produce hundreds of IOPS, while an SSD can produce hundreds of thousands of IOPS