

**Systemsoftware &
Distributionsystems**

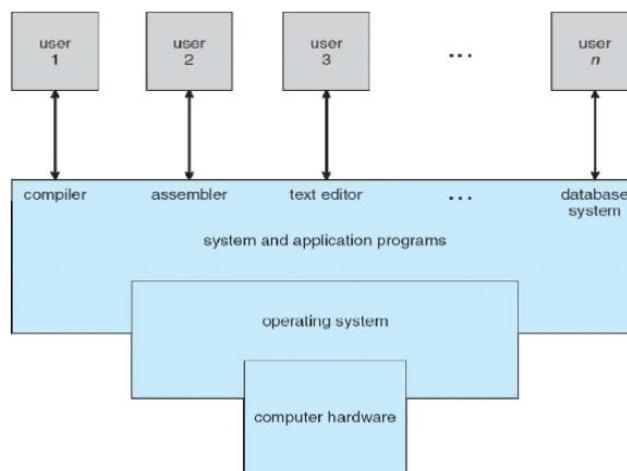
Summary

TOC

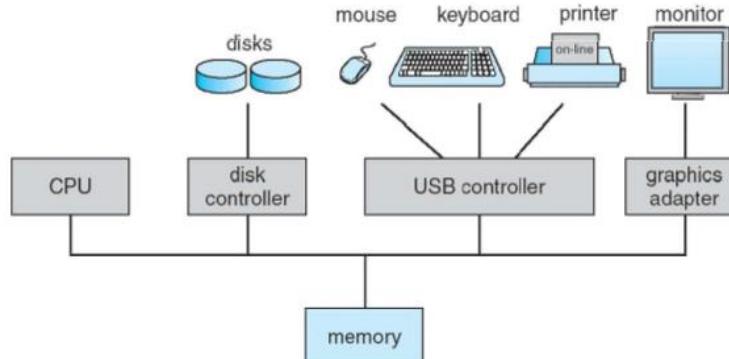
V1/2 Chapter 1: OS Introduction	4
V2 Chapter 2: Operating System Structures	12
V3/4 Chapter 3: Processes and Inter Process Process Communications Communications.....	20
V4 Chapter 4: Threads	30
V5 Chapter 5: Scheduling	33
V5 Chapter 6: Synchronization	39
V6 Chapter 7: Deadlocks	49
V7 Chapter 8: Memory Management	56
V8 Chapter 9: Virtual Memory	63
V9 Chapter 10: Distributed Systems	70
V11 Chapter 11: Naming	81
V11 Chapter 12: Distributed File Systems	86
V12 Chapter 13: Synchronization in Distributed Systems	93

V1/2 Chapter 1: OS Introduction

- Operating System Definition
 - A program that acts as an intermediary between a user, and its application programs, of a computer and the computer hardware
 - Operating System (OS) goals:
 - Execute user programs and make solving user problems easier
 - Make the computer system convenient to use
 - Use the computer hardware in an efficient manner
 - OS is a resource allocator
 - Manages all resources
 - Decides between conflicting requests for efficient and fair resource use
 - OS is a control program
 - Controls execution of programs to prevent errors and improper use of the computer
 - “The one program running at all times on the computer” is the kernel “Systemkern”
 - Everything else is either a system program (ships with the operating system) or an application program
 - “bootstrap program” (“Ladeprogramm”) is loaded at power-up or reboot
 - Typically stored in ROM “read only memory” or EPROM, generally known as firmware
 - Initializes all aspects of system
 - Loads operating system kernel and starts execution
- Computer System Structure
 - A computer system can be divided into four main components
 - **Hardware** - provides basic computing resources
 - CPU (Central Processing Unit), memory, I/O (in/out) devices
 - **Operating system**
 - Controls and coordinates use of hardware among various applications and users
 - **Application programs** - define the ways in which system resources are used to solve the computing problems of the users
 - Word processors, compilers, web browsers, database systems, video games
 - **Users**
 - People, machines, other computers
- Four Components of a Computer System

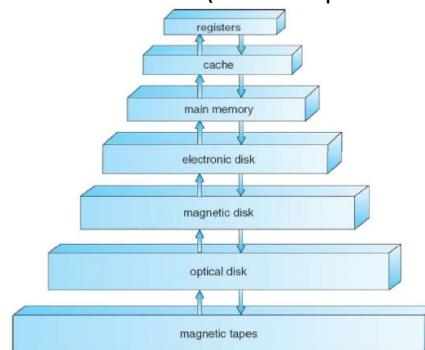


- Computer System Organization
 - Computer system **operation**
 - One or more CPUs, device controllers connect through common bus providing access to shared memory
 - Concurrent execution of CPUs and devices competing for memory cycles

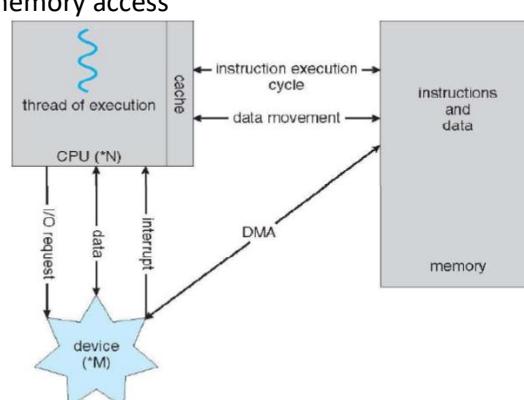


- Computer System Operation
 - I/O devices and the CPU can execute concurrently
 - Each device controller is in charge of a particular device type
 - E.g., g disks , network interface , GPUs (Graphical PU)
 - Each device controller has a local memory buffer
 - CPU moves data from/to main memory to/from local buffers
 - I/O is from the device to local buffer of controller
 - Device controller informs CPU that it has finished its operation by causing an **interrupt**
- I/O Structure
 - After I/O starts, control returns to user program **only upon I/O completion**
 - Wait instruction idles the CPU until the next interrupt
 - Wait loop (contention for memory access)
 - At most one I/O request is outstanding at a time, no simultaneous I/O processing
 - After I/O starts, control returns to user program **without waiting for I/O completion**
 - **System call** - request to the operating system to allow user to wait for I/O completion
 - **Device-status table** contains entry for each I/O device indicating its type, address, and state
 - Operating system indexes into I/O device table to determine status and to modify table entry to include interrupt
- Direct Memory Access Structure
 - Direct Memory Access (DMA)
 - Used for high-speed I/O devices able to transmit information at close to memory speeds
 - Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
 - Only one interrupt is generated per block, rather than the one interrupt per byte
- Storage Structure
 - Main memory - only large storage media that the CPU can access directly
 - Secondary storage - extension of main memory that provides large nonvolatile storage capacity
 - Magnetic disks - rigid metal or glass platters covered with magnetic recording material
 - Disk surface is logically divided into **tracks**, which are subdivided into **sectors**
 - The **disk controller** determines the logical interaction between the device and the computer

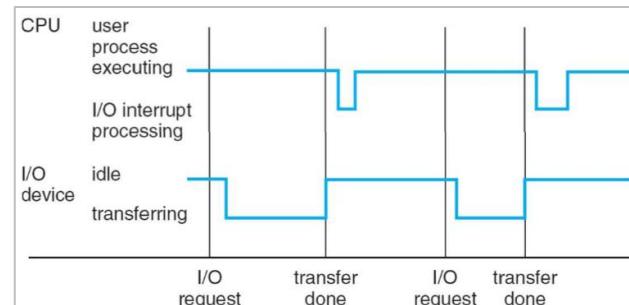
- Storage Device Hierarchy
 - Above electronic disk SSD → volatile (when no power cannot keep storage)



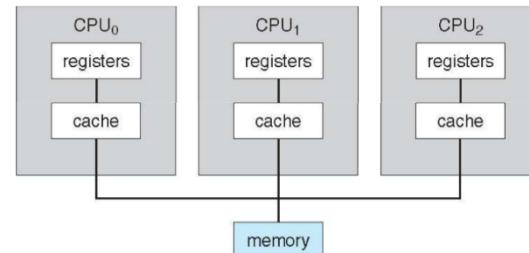
- Caching
 - Important principle, performed at many levels in a computer (in hardware, operating system, software)
 - Information in use **copied from slower to faster storage temporarily**
 - Faster storage (cache) checked first to determine
 - if information is there
 - If it is, information used directly from the cache (fast)
 - If not, data copied to cache and used there
 - Cache smaller than storage being cached
 - Cache management important design problem
 - Cache size and replacement policy
- Computer System Architecture
 - Most systems use one or more general-purpose processors
 - Most systems have special-purpose processors as well
 - **Multi-processor** systems growing in use and importance
 - Also known as parallel systems, tightly-coupled systems
 - Parallel systems are nice, but we buy synchronization problems
 - Advantages include
 - Increased throughput
 - economy of scale
 - Increased reliability - graceful degradation or fault tolerance
 - Two types
 - **Asymmetric Multiprocessing**
 - Only one or some CPUs may execute OS kernel code, I/O, use peripheral devices
 - **Symmetric Multiprocessing**
- Modern Computer Operation
 - DMA = Direct memory access



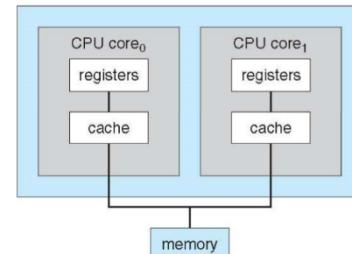
- Interrupt
 - An **interrupt** transfers control to the interrupt service routine generally, through the **interrupt vector**, which contains the addresses of all service routines
 - The interrupt architecture must save the address of the interrupted instruction
 - Incoming interrupts are disabled, while another interrupt is being processed to prevent a lost interrupt
 - A **trap** is a software-generated interrupt caused either by an error or a user request
 - Modern operating systems are **interrupt-driven**
- Interrupt Handling
 - The operating system preserves the state of the CPU by storing registers and the program counter
 - Determines which type of interrupt has occurred:
 - Polling
 - Vectored interrupt system
 - Separate segments of code determine what action should be taken for each type of interrupt
- Interrupt Timeline



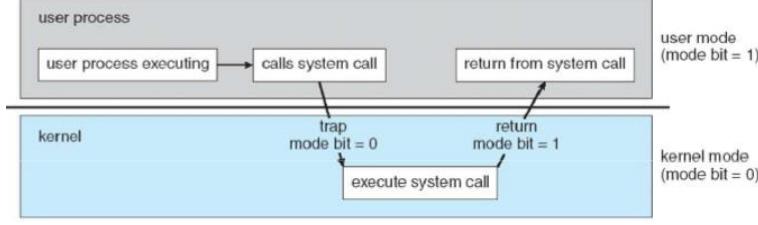
- Symmetric Multi-processing Architecture (SMP)



- A Dual Core Design
- Clustered Systems
 - Like multi-processor systems, but multiple systems working together
 - Usually sharing storage via a **Storage Area Network (SAN)**
 - Provides a **high availability** service which survives failures
 - **Asymmetric clustering** has one machine in hot-standby mode
 - **Symmetric clustering** has multiple nodes running applications, monitoring each other
 - Some clusters are for **High Performance Computing (HPC)**
 - Applications must be written to use **parallelization**
 - Other clusters of computers are used for **web and cloud services**



- Operating System Structure
 - **Multi-programming** needed for efficiency
 - Single user cannot keep CPUs, I/O devices both busy at all times
 - Multi-programming organizes jobs (code and data) so CPU always has one to execute
 - A subset of total jobs in system is kept in memory
 - One job selected and run via **job scheduling**
 - When job has to wait (e.g., for I/O), OS switches to another job

- **Timesharing (multi-tasking)** is the logical extension in which CPU switches jobs frequently that users can interact with each job while it is running → **interactive computing**
 - Response time should be below 1 second
 - User has at least one program executing in memory (**process**)
 - If several jobs ready to run at the same time → **CPU scheduling**
 - If processes do not fit in memory, **swapping** moves them in and out
 - **Virtual memory** allows execution of program not completely in memory
- Operating System Operations
 - Interrupt-driven by hardware
 - Software error or request creates exception or trap
 - Division by zero, request for operating system service
 - Other process problems include infinite loops, processes modifying each other or the operating system
 - Dual mode operation allows OS to protect itself and other system components
 - User mode and kernel mode
 - Mode bit provided by hardware
 - Provides ability to distinguish when system is running user code or kernel code
 - Some instructions designated as privileged, only executable in kernel mode
 - System call changes mode to kernel, return from call resets to user
- Transition from User to Kernel Mode
 - Timer to prevent infinite loop/process hogging resources
 - Set interrupt after specific period
 - Operating system decrements counter
 - When counter equals zero interrupt is generated
 - Set up before scheduling process to regain control or terminate program that exceeds allotted time
 

```

graph TD
    subgraph User [User Process]
        direction LR
        A[user process executing] --> B[calls system call]
        B --> C[return from system call]
    end
    subgraph Kernel [Kernel]
        direction LR
        D[trap mode bit = 0] --> E[execute system call]
        E --> F[return mode bit = 1]
    end
    B --> D
    C --> F
  
```
- Process Management
 - A **process is a program in execution**. It is a unit of work within the system. Program is a passive entity, process is an active entity.
 - Process needs resources to accomplish its task, executing the program instructions
 - CPU, memory, I/O, files
 - Initialization data
 - Process termination requires reclaim of any reusable resources
 - Single-threaded process has one program counter specifying location of next instruction to execute
 - Process executes instructions sequentially, one at a time, until completion
 - Multi-threaded process has one program counter per thread
 - Typically a system has many processes, some users, some operating systems running concurrently on one or more CPUs
 - Concurrency is reached by multiplexing CPUs among processes and threads
- Process Management Activities
 - The operating system is responsible for the following activities in connection with the process management:

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

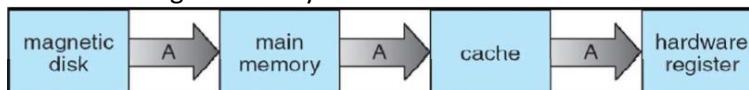
V1 until here

- Memory Management
 - Prerequisites for processes to run:
 - All data in memory before and after processing
 - All instructions in memory in order to execute
 - Memory management determines what is in memory and when
 - Optimizing CPU utilization and computer response to users
 - Memory management activities
 - Keeping track of which parts of memory are currently being used and by whom
 - Deciding which processes (or parts thereof) and data to move into and out of memory
 - Allocating and de-allocating memory space as needed
- Storage Management
 - OS provides uniform, logical view of information storage
 - Abstracts physical properties to logical storage unit - **file**
 - Each medium is controlled by device (i.e., disk drive, tape drive)
 - Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)
 - File System management
 - Files usually organized into directories
 - Access control on most systems to determine who can access
 - what
 - OS activities include
 - Creating and deleting files and directories
 - Primitives to manipulate files and directories
 - Mapping files onto secondary storage
 - Backup files onto stable (non-volatile) storage media
- Mass Storage Management
 - Usually disks used to store data that does not fit in main memory or data that must be kept for a “long” period of time
 - Proper management is of central importance
 - Entire speed of computer operation hinges on disk subsystem and its algorithms
 - OS activities
 - Free-space management
 - Storage allocation
 - Disk scheduling
 - Some storage need not be fast
 - Tertiary storage includes optical storage, magnetic tape
 - Still must be managed
 - Varies between WORM (write-once, read-many-times) and RW (read-write)
- Performance of Various Levels of Storage
 - Movement between levels of storage hierarchy can be explicit or implicit

Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	> 16 MB	> 16 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000.000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape

- Migration of Integer A from Disk to Register

- Multi-tasking environments must be careful to use most recent value, no matter where it is stored in the storage hierarchy



- Multi-processor environment must provide cache coherency in hardware such that all CPUs have the most recent value in their cache
- Distributed environment situation even more complex
 - Several copies of a datum can exist
 - Various solutions covered in Distributed File Systems

- I/O Subsystem

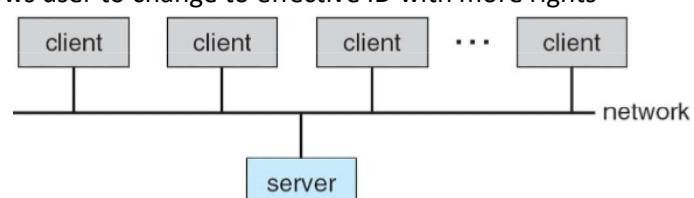
- One purpose of OS is to **hide peculiarities of hardware devices from the user**
- I/O subsystem responsible for
 - Memory management including buffering (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling (the overlapping of output of one job with input of other jobs)
 - General device-driver interface
 - Drivers for specific hardware devices

- Protection and Security

- Protection**
 - Any mechanism for controlling access of processes or users to resources defined by the OS
- Security**
 - Defense of the system against internal and external attacks
 - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- Systems generally first distinguish among users, to determine who can do what
 - User identities (user IDs, security IDs) include name and associated number, one per user
 - User ID then associated with all files, processes of that user to determine access control
 - Group identifier (group ID) allows set of users to be defined and controls managed, then also associated with each process, file
 - Privilege escalation allows user to change to effective ID with more rights

- Distributed Computing

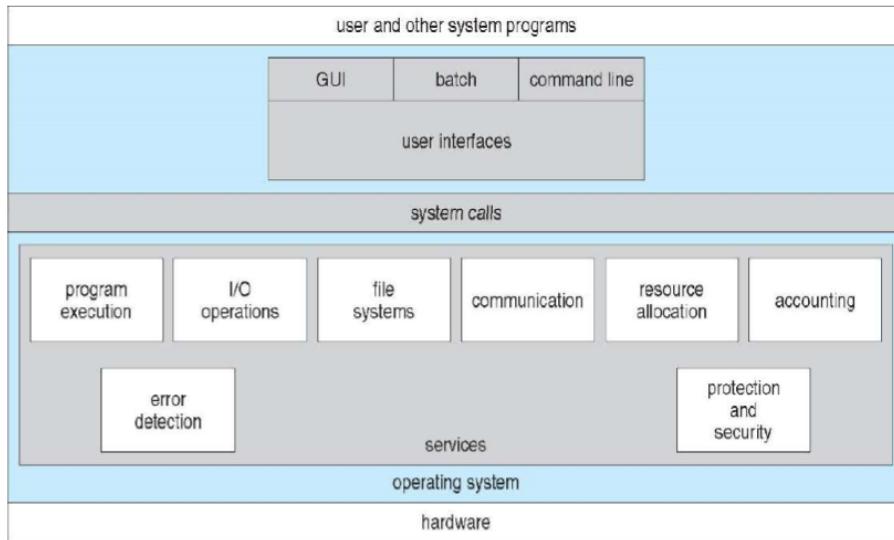
- Client/Server Computing
- Many systems are now servers, responding to requests generated by clients



- Compute server provides an interface to client to request services (i. e. database)
- File server provides interface for clients to store and retrieve files
- Web servers ...
- Peer-to-Peer Computing
 - **Peer-to-Peer (P2P)** another model of distributed system
 - P2P does not distinguish between clients and servers
 - Instead all nodes are considered peers
 - May each act as client, server or both
 - Node must join P2P network
 - Registers its service with central lookup service on network, or
 - Broadcast request for service and respond to requests for service via **discovery protocol**
 - Examples include Napster and Gnutella
- Web-based Computing
 - The Web has become ubiquitous
 - PCs are most prevalent devices in the past
 - Now many smart mobile devices exist
 - More (basically all) devices becoming networked to allow Web (or server) access
 - Use of operating systems like Windows 95, client-side only, have evolved into Linux, Windows 10, and Mac OS X, which can be used both for clients and servers
- Open-Source Operating Systems
 - Operating systems made available in source-code format rather than just binary closed-source
 - Counter move to the copy protection and Digital Rights Management Management (DRM) (DRM) movement movement
 - Started by Free Software Foundation (FSF), which has “copyleft” GNU Public License (GPL)
 - Examples include GNU/Linux, BSD UNIX (including core of Mac OS X), and Sun Solaris
- Ending Statement
 - There is no news to be expected, anywhere on earth ...
 - “Everything that can be invented has been invented.” Charles H. Duell, Commissioner, U.S. Office of Patents, 1899

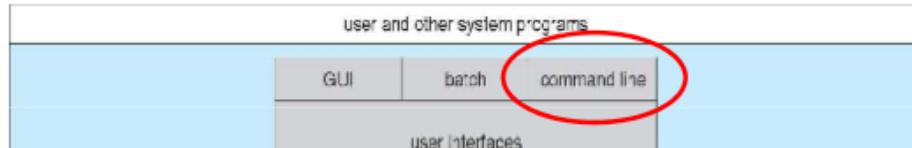
V2 Chapter 2: Operating System Structures

- A View of Operating System Services

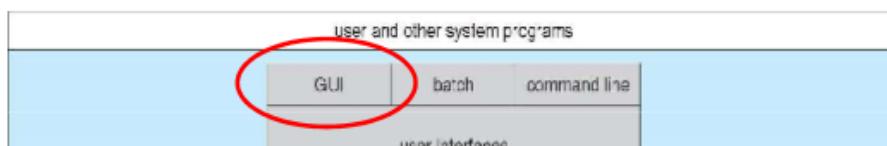
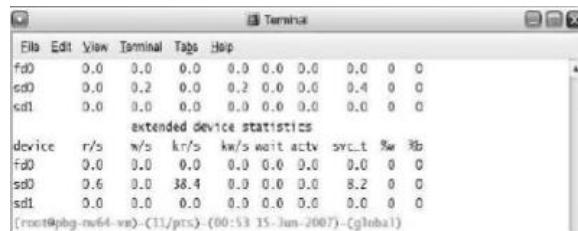


- Operating System Services
- User
 - One set of operating system services provides functions that are helpful to the user:
 - **User interface:** Almost all operating systems have a user interface (UI), which vary between Command-Line (CLI), Graphics User Interface (GUI), Batch
 - **Program execution:** execution: The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
 - **I/O operations:** A running program may require I/O, which may involve a file or an I/O device
 - **File system manipulation:** The file system is of particular interest. Obviously, programs need to read and write files and directories, create and delete them, search them, list file information, permission management.
 - **Communications:** Processes may exchange information, on the same computer or between computers over a network
 - Communications may be via shared memory or through message passing (packets moved by the OS)
 - **Error detection:** OS needs to be constantly aware of possible errors
 - May occur in the CPU and memory hardware, in I/O devices, in user program
 - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
 - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system
- Operations
 - Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing:
 - **Resource allocation:** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
 - Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code

- **Accounting:** To keep track of which users use how much and what kinds of computer resources
- **Protection and security:** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
 - Protection involves ensuring that all access to system resources is controlled
 - Security of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts
 - If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.



- User Operating System Interface
- CLI
 - Command Line Interface (CLI) or command interpreter allows direct command entry
 - Sometimes implemented in kernel, sometimes by systems program
 - Sometimes multiple implementation flavors available
 - **shells**
 - Primarily fetches a command from user and executes it
 - Sometimes commands built-in, sometimes just names of programs
 - If the latter, adding new features doesn't require shell modification

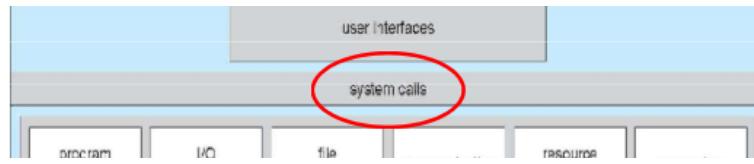


- User Operating System Interface
- GUI



- User friendly desktop metaphor interface
 - Usually mouse, keyboard, and monitor
 - Icons represent files, programs, actions, etc
 - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a folder))
- Most systems now include both CLI and GUI interfaces
 - Microsoft Windows is GUI with CLI "command" shell

- Apple Mac OS X has “Aqua” GUI interface with UNIX kernel underneath and shells available
- Solaris (and Linux) is CLI with optional GUI interfaces (Java Desktop, KDE)



- System Calls
 - Programming interface to the services provided by the OS
 - Typically written in a high level system language (C or C++)
 - Mostly accessed by programs via a high level Application Program Interface (API) rather than direct system call use
 - Three most common APIs are Windows API, POSIX API for POSIX based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
 - Example of System Calls
 - System call sequence to copy the content of one file to another file
 - Example of a Standard API
 - Consider the ReadFile() function in the Win32 API
 - A function for reading from a file
- return value

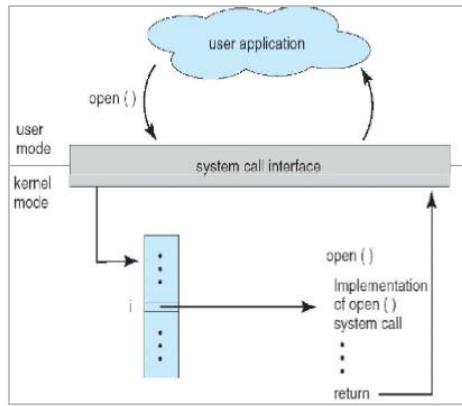
```
BOOL ReadFile c (HANDLE file,
                 LPVOID buffer,
                 DWORD bytes To Read,
                 LPDWORD bytes Read,
                 LPOVERLAPPED ov);
```

function name

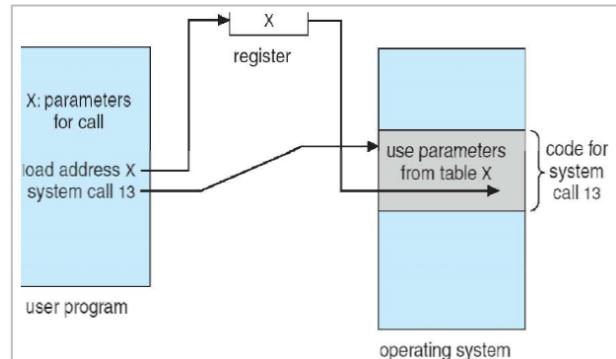
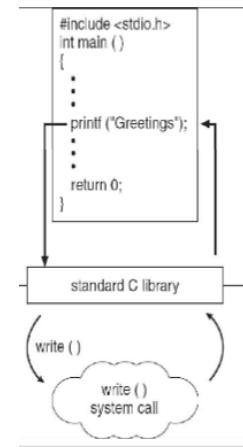
source file → destination file

Example System Call Sequence

```
Acquire input file name
Write prompt to screen
Accept input
Acquire output file name
Write prompt to screen
Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
  Until read fails
Close output file
Write completion message to screen
Terminate normally
```
- A description of the parameters passed to ReadFile()
 - HANDLE file — the file to be read
 - LPVOID buffer — a buffer where the data will be read into and written from
 -
 - DWORD bytesToRead — the number of bytes to be read into the buffer
 - LPDWORD bytesRead — the number of bytes read during the last read
 - LPOVERLAPPED ov — indicates if overlapped I/O is being used
 - System Call Implementation
 - Typically, a number is associated with each system call
 - System call interface maintains a table indexed according to these numbers
 - The system call interface invokes intended system call in OS kernel
 - Returns status of the system call and any return values
 - The caller needs know nothing about how the system call is implemented
 - Just needs to obey the API and understand what OS will do as a result call
 - Most details of OS interfaces hidden from programmer by API
 - Managed by run time support library (set of functions built into libraries included with compiler)
 - API
 - System Call: OS Relationship



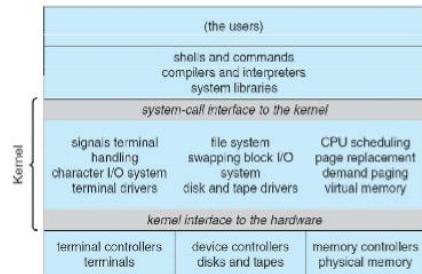
- Standard C Library Example
 - C program invoking `printf()` library call, which calls the `write()` system call
- System Call Parameter Passing
 - Often, more information is required than simply to identify a desired system call
 - Exact type and amount of information vary according to OS and call
 - Three general methods used to pass parameters to the OS
 - Simplest: pass the parameters in registers
 - In some cases, may be more parameters than registers
 - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
 - This approach taken by Linux and Solaris
 - Parameters placed, or pushed, onto the stack by the program and popped off the stack by the operating system
 - Block and stack methods do not limit the number or length of parameters being passed
- Parameter Passing via Table
- Types of System Calls
 - Process control
 - File management
 - Device management
 - Information maintenance
 - Communications
 - Protection
- Examples of Windows and Unix System Calls



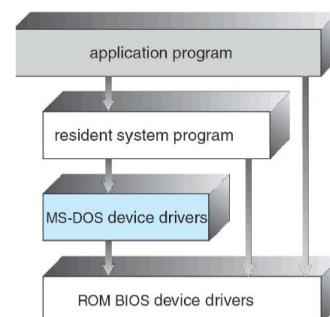
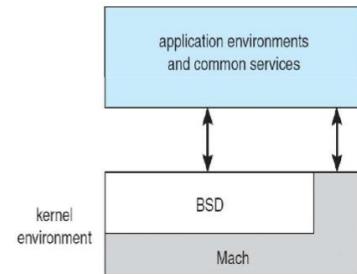
	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information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

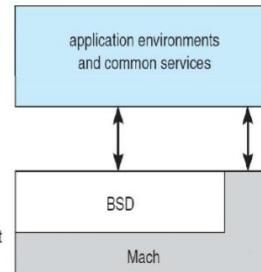
- System Programs
 - System programs provide a convenient environment for program development and execution
 - They can be divided into:
 - File manipulation: Create , delete , copy , print
 - Status information: Date, time, amount of available memory, disk space, number of users
 - File modification: Rename, edit
 - Programming language support: Compilers, assemblers, debuggers and interpreters
 - Program loading and execution
 - Communications: Send messages, browse web pages, send electronic mail, transfer files
 - Application programs: Word, Acrobat, Photoshop
 - Some of them are simply user interfaces to system calls; others are considerably more complex
- Operating System Design: Implementation
 - Best design and implementation of OS not “solvable”, but some approaches have proven successful
 - Internal structure of different Operating Systems can vary widely
 - Start by defining goals and specifications
 - Start by defining goals and specifications
 - Affected by choice of hardware, type of system
 - User goals and system goals
 - User goals
 - operating system should be convenient to use, easy to learn, reliable, safe, and fast
 - System goals
 - operating system should be easy to design,
 - implement, and maintain, as well as flexible, reliable, error-free, and efficient

- UNIX
- Monolithic Kernels
 - Limited by hardware functionality, the **original UNIX operating system** had limited structuring.
 - UNIX OS consists of:
 - Systems programs
 - Monolithic kernel
 - Consists of everything below the system call interface and above physical hardware
 - Single large process encompassing all services in a single kernel address space
 - Provides the file system, CPU scheduling, memory management, and other operating system functions; a large number of functions for one level
 - Examples: Unix/BSD, Linux, MS DOS/earlier

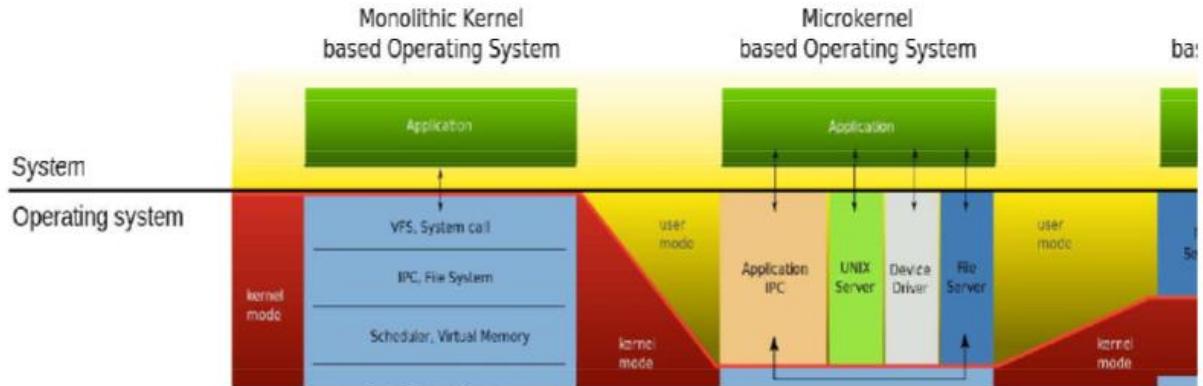


- Micro Kernel System Structure
 - Moves as much from the kernel into “user” space
 - Device drivers, protocol stacks and file systems, are removed from the micro kernel to run as separate processes, servers, often in user space
 - Communication takes place between micro kernels using message passing
 - Benefits
 - Easier to extend a microkernel
 - Easier to port the operating system to new architectures
 - More reliable (less code is running in kernel mode)
 - “More secure”
 - Detriments
 - Performance overhead of user space to kernel space communication
 - Mach and OS X use this approach
- Hybrid System Structure
 - Kernel structure similar to a micro kernel, but implemented in terms of a monolithic kernel
 - In contrast to a micro kernel, all (or nearly all) operating system services are in kernel space
 - No performance overhead for message passing and context switching it between kernel and user mode as in monolithic kernels
 - No reliability benefits of having services in user space as in micro kernels
 - Windows NT and Windows 8 used hybrid micro kernels
- Simple Structure
 - MS-DOS: written to provide the most functionality in the least space
 - Not divided into modules
 - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated
- Mac OS X Structure



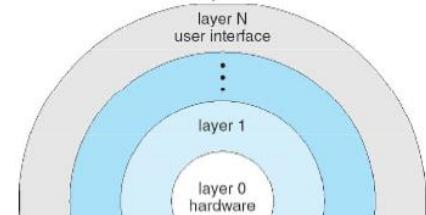


- System Structure Comparison



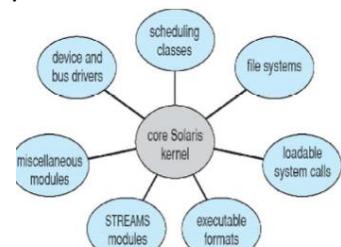
- Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers.
- The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower level layers



- Modules

- Modern operating systems implement kernel modules
 - Uses object oriented approach
 - Each core component is separate
 - Each talks to the others over known interfaces
 - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexibility
- OpenVMS, Linux, BSD, and UNIX variants such as SunOS
 - Can dynamically load executable modules at runtime, at the binary (image) level



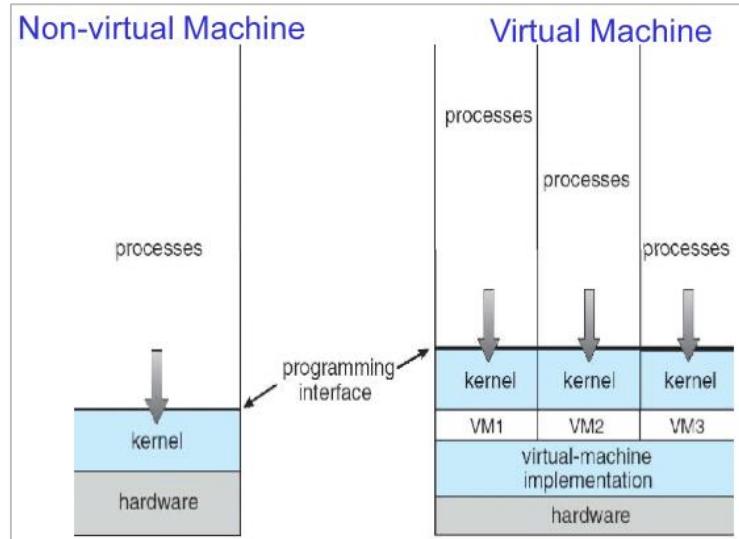
- Virtual Machines

- A virtual machine takes the layered approach to its logical conclusion. It treats hardware and the native operating system kernel as though they were all hardware
 - A virtual machine provides an interface identical to the underlying bare hardware
- The operating system host creates the illusion that a process has its own processor and (virtual memory)
- Each guest provided with a (virtual) copy of underlying computer

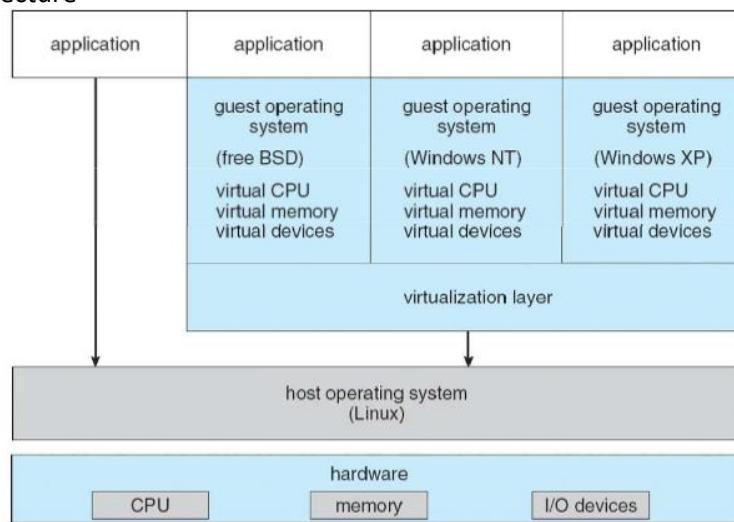
- Virtual Machines History and Benefits

- First appeared commercially in IBM mainframes 1972
- Fundamentally, multiple execution environments (different operating systems) can share the same hardware
- Protect from each other
- Some sharing of file can be permitted, controlled
- Commute with each other, other physical systems via networking
- Useful for development, testing

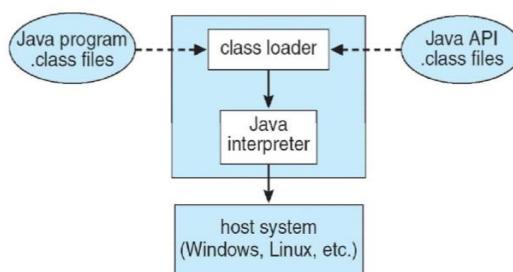
- Consolidation of many low resource use systems onto fewer busier systems
- “Open Virtual Machine Format”, standard format of virtual machines, allows a VM to run within many different virtual machine (host) platforms
- Virtual Machines
- Concept



- VMware Architecture



- The Java Virtual Machine



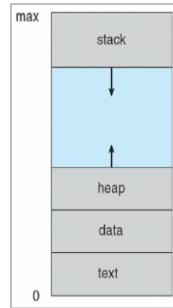
- .NET Framework

- Collection of technologies
 - Set of class libraries
 - Execution environment (and development platform)
- Target code written for .NET
 - Hides all specifics of underlying hardware (and OS software)

- Execution environment abstracts away
 - Virtual Machine
- Core characteristics
 - Common Language Runtime (CLR), the .NET VM
 - Just-in-time compiler compiles intermediate from C#, VB.NET
 - Assemblies contain MS-IL (Intermediate Language) instructions (.DLL)
 - Results of CLR executed on host system
- System Boot
 - Operating systems must be made available to hardware so hardware can start it
 - Small piece of code
 - **bootstrap loader** –, locates the kernel, loads it into memory, and starts it
 - Sometimes as a two step process where the boot block at fixed location loads the bootstrap loader
 - When power initialized on system, execution starts at a fixed memory location
 - Firmware used to hold initial boot code
- Ending Statement
 - There is expectations, which hold for a while ... I think there is a world market for maybe five computers. (Thomas Watson, Chairman IBM)

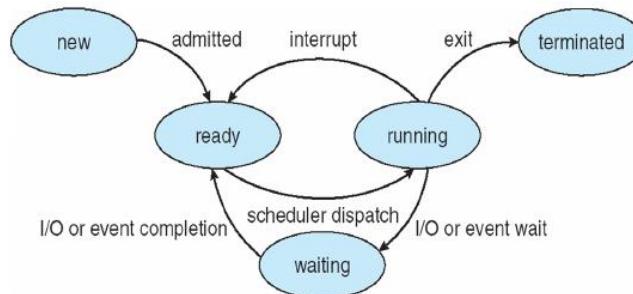
V3/4 Chapter 3: Processes and Inter Process Communications

- Processes
 - Objectives
 - To introduce the notion of a process - a program in execution, which forms the basis of all computation
 - To describe the various features of processes, including scheduling, creation and termination, and communication
 - Topics
 - Process Concept
 - Process Scheduling
 - Operations on Processes
 - Interprocess Communication
 - Examples of IPC Systems
 - Communication in Client-Server Systems
- Process Concept
 - An operating system executes a variety of programs:
 - Batch system
 - jobs
 - Time-shared systems
 - user programs or tasks
 - Textbook uses the terms job and process almost interchangeably
 - Process
 - a program in execution; process execution must progress in sequential fashion
 - A process includes:
 - Program counter
 - Stack
 - Data section
- Process in Memory



- Process States

- As a process executes, it changes state (Linux)
 - new: The process is being created
 - running: Instructions are being executed
 - waiting: The process is waiting for some event to occur
 - ready: The process is waiting to be assigned to a processor
 - terminated: The process has finished execution



- Process Control Block (PCB)

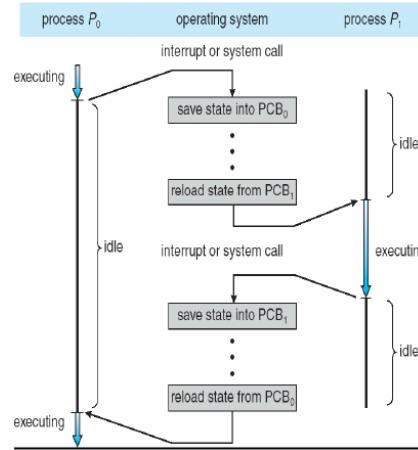
- Information associated with each process
 - Process state
 - Program counter
 - CPU registers
 - CPU scheduling information
 - Memory-management information
 - Accounting information
 - I/O status information

process state
process number
program counter
registers
memory limits
list of open files
...

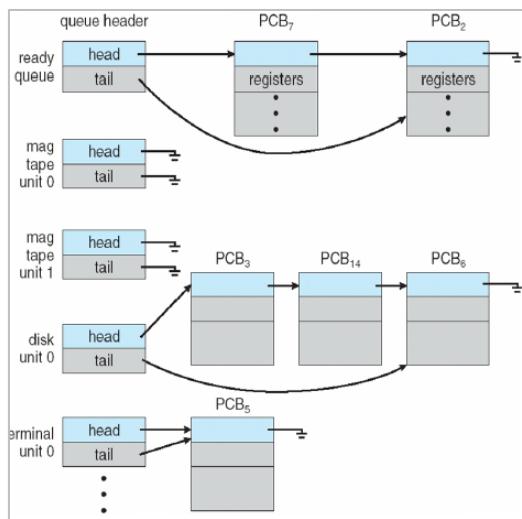
- Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
- Context of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support

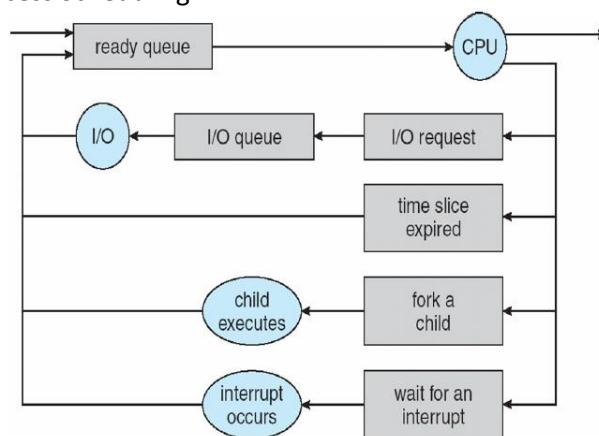
- CPU Switch From Process to Process



- Process Scheduling Queues
 - Job queue
 - set of all processes in the system
 - Ready queue
 - set of all processes residing in main memory, ready and waiting to execute
 - Device queues
 - set of processes waiting for an I/O device device
 - Processes migrate among the various queues
- Ready Queue And Various I/O Device Queues

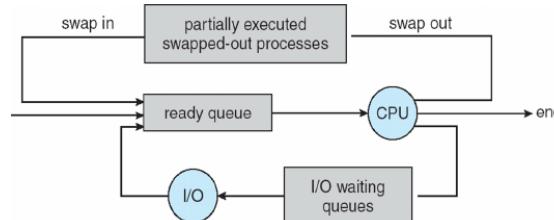


- Representation of Process Scheduling



- Schedulers (1)

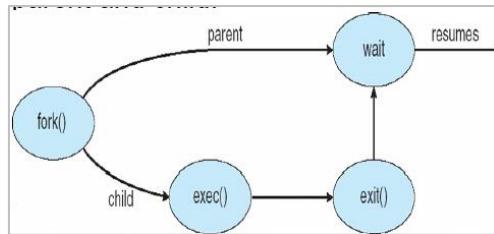
- Long-term scheduler (or job scheduler)
- selects which processes should be brought into the ready queue
- Short-term scheduler (or CPU scheduler)
- selects which process should be executed next and allocates CPU
- Addition of medium-term scheduling
- Swapping process in/out



- Short-term scheduler is invoked very frequently (milliseconds) (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) (may be slow)
 - May be absent, jobs simply added to ready queue
- Processes can be described as either:
 - I/O-bound process - spends more time doing I/O than computations, many short CPU bursts
 - CPU-bound process
 - spends more time doing computations; few very long CPU bursts
- Process Creation
 - Parent process creates children processes, creates other processes, which results in forming a tree of processes
 - Generally, process identified and managed via a process identifier (pid)
 - Resource sharing options
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
 - Execution strategies options
 - Parent and children execute concurrently
 - Parent waits until children terminate
 - Address space options
 - Child duplicate of parent
 - Child has a program loaded into it
- Explicit Process Creation
 - UNIX examples
 - fork system call creates new process
 - procid = fork() replicates calling process
 - exec system call used after a fork to replace the process' memory space p with a new program
 - Parent and child identical except for the value of procid
 - Use procid to diverge parent and child:

```

if (procid == 0)
    do_child_processing
else
    do_parent_processing
  
```



- C Program Forking Separate Process

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    pid_t pid;
    pid = fork(); /* fork a child process */
    if (pid < 0) { /* error occurred */
        printf("Fork Failed\n");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        printf("I am the child %d\n", pid);
        execvp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        printf("I am the parent %d\n", pid); /* parent will wait for the child to complete */
        wait(NULL);

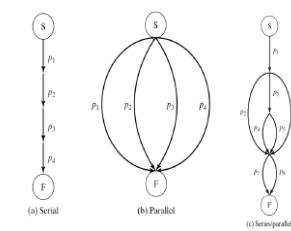
        printf("Child Complete\n");
        exit(0);
    }
}
  
```

- Process Termination

- Process executes last statement and asks the operating system to delete it (exit)
 - Output data from child to parent (via wait)
 - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes
 - Child has exceeded allocated resources
 - Task assigned to child is no longer required
 - Some operating system do not allow child to continue if its parent terminates
 - All children terminated - cascading termination
 - Linux: reparenting to pid 1 (init)

- Process Ordering and Precedence

- Process flow graph depicts process order, parallelism and precedence
 - Serial execution is expressed as: S(p₁, p₂, ...)
 - Parallel execution is expressed as: P(p₁, p₂, ...)
- Figure (c) represents the following → Serialparallel:
S(p₁, P(p₂, S(p₃, P(p₄, p₅)), p₆), P(p₇, p₈))



- Data Parallelism

- Same code is applied to different data
 - E.g., SIMD vector instructions
- The forall statement
 - Syntax: `y forall (parameters) statements`
 - Semantics (meaning):
 - Parameters specify set of data items
 - Statements are executed for each item parallel

- Data Parallelism

- Example: matrix multiply A = B × C

```

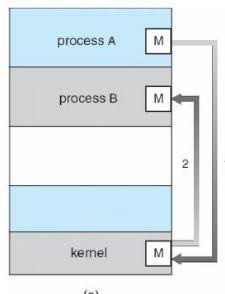
forall ( i:1..n, j:1..m )
  A[i][j] = 0;
  for ( k=1; k<=r; ++k )
    A[i][j] = A[i][j] +
      B[i][k]*C[k][j];
  
```

- Each inner product is computed sequentially
- All column-row products are computed in parallel
- Interprocess Communication (IPC)
 - Processes in a system: independent or cooperating
 - Independent process cannot affect or be affected by the execution of another process
 - Cooperating process can affect or be affected by the execution of another process
 - Reasons for cooperating processes:
 - Information sharing
 - Computation speedup
 - Modularity
 - Convenience
 - Cooperating processes need interprocess communication (IPC) mechanisms

- Communications Models

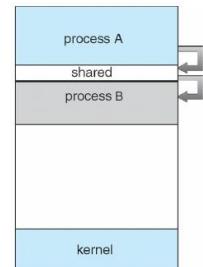
- Two models of IPC

Message passing



(a)

Shared memory



(b)

- Producer-Consumer Problem

- Paradigm for cooperating processes, producer process produces information that is consumed by a consumer process
 - Unbounded-buffer places no practical limit on the size of the buffer
 - Bounded-buffer assumes that there is a fixed buffer size
- Buffer data structure as circular array of items:
 - Using shared memory for IPC

```

#define BUFFER_SIZE 10
class item {
  . . .
};

item
buffer[BUFFER_SIZE];
// points to empty slot
int in = 0;
// next available item
int out = 0;
  
```

- Bounded-Buffer
- Producer

```

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

- Bounded Buffer
- Consumer

```

while (true) {
    while (in == out);
    // do nothing -- nothing to consume

    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    // do something with the item
}
  
```

- Message Passing
 - Mechanism for processes to communicate and to synchronize their actions
 - Message system
 - processes communicate with each other other without without resorting resorting to to shared shared variables variables
 - IPC facility provides two operations (setup
 - msgget():
 - send(message)
 - msgsnd ()
 - receive(message)
 - msgrcv()
 - If P and Q wish to communicate, they need to:
 - Establish a communication link between them
 - Exchange messages via send/receive
 - Implementation of communication link
 - Physical y (e.g., (g , shared memory, y, hardware bus, , network))
 - Logical (e.g., logical properties)
 - Logical properties of a link
 - Direct or indirect communication
 - Synchronous or asynchronous communication
 - Automatic or explicit buffering
- Direct Communication
 - Processes must name each other explicitly:
 - send (P, message)
 - send a message to process P
 - receive (Q, message)
 - receive a message from process Q
 - Properties of communication link
 - Links are established automatically
 - A link is associated with exactly one pair of communicating processes
 - Between each pair there exists exactly one link
 - receive(id, message) is possible too
 - receive message from any process (symmetric / asymmetric addressing)
- Indirect Communication
 - Messages are directed and received from mailboxes (also referred to as ports)
 - Each mailbox has a unique id
 - Processes can communicate only if they share a mailbox
 - Primitives Primitives are are defined defined as: as:
 - send(A, message)
 - send a message to mailbox A
 - receive(A, message)
 - receive a message from mailbox A
 - Properties of communication link
 - Link established only if processes share a common mailbox

- A link may be associated with many processes
 - Each pair of processes may share several communication links
- Mailbox sharing
 - P1, P2, and P3 share mailbox A
 - P1, sends; P2 and P3 receive
 - Who gets the message?
- Solutions
 - Allow a link to be associated with at most two processes
 - Allow only one process at a time to execute a receive operation
 - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.
- Synchronization
 - Message passing may be either blocking or non- blocking (IPC_NOWAIT)
 - Blocking is considered synchronous
 - Blocking send has the sender block until the message is received
 - Blocking receive has the receiver block until a message is available
 - Non-blocking is considered asynchronous
 - Non-blocking send has the sender send the message and continue
 - Non-blocking receive has the receiver receive a valid message or null
- Buffering
 - Queue of messages attached to the link; implemented in one of three ways
 1. Zero capacity - 0 messages
Sender must wait for receiver (rendezvous)
 2. Bounded capacity - finite length of n messages
Sender must wait if link full
 3. Unbounded capacity - infinite length
Sender never waits
- Examples of IPC Systems - POSIX
 - POSIX Shared Memory
 - Process first creates shared memory segment
 - segment id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
 - Process wanting access to that shared memory must attach to it
 - shared memory = (char *) shmat(id, NULL, 0);
 - Now the process could write to the shared memory
 - sprintf(shared memory, "Writing to shared memory");
 - When done a process can detach the shared memory from its address space
 - shmdt(shared memory);
- Examples of IPC Systems Mach
 - Mach communication is message based
 - Even system calls are messages
 - Each task gets two mailboxes at creation- Kernel and Notify
 - Only three system calls needed for message transfer
 - msg_send(), msg_receive(), msg_rpc()
 - Mailboxes needed for communication, created via
 - port_allocate()
- Shared Memory + Fork

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

int main()
{
    int pid;
    int segment_id; /* the identifier for the
shared memory segment */
    char *shared_memory; /* a pointer to the
shared memory segment */
    const int segment_size = 4096; /* the size
(in bytes) of the shared memory segment */

    /* allocate a shared memory segment */
    segment_id = shmget(IPC_PRIVATE,
segment_size, S_IRUSR | S_IWUSR);

    /* attach the shared memory segment */
    shared_memory = (char *) shmat(segment_id,
NULL, 0);
    //printf("shared memory segment %d attached
at address %p\n", segment_id, shared_memory);
}
pid = fork(); /* fork another process */

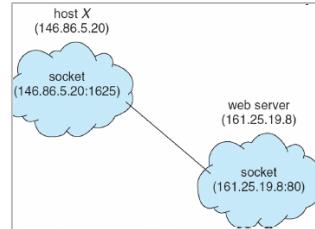
if (pid == 0) { /* child process */
    /* write a message to the shared memory
segment */
    printf("%s\n", shared_memory);
    sprintf(shared_memory, "Thomas was here");
} else { /* parent process */
    wait(NULL);

    /* now print out the string from shared
memory */
    printf("**Parent Got: %s\n",
shared_memory);

    /* now detach the shared memory segment
*/
    if (shmdt(shared_memory) == -1) {
        fprintf(stderr, "Unable to detach\n");
    }
    /* now remove the shared memory segment
*/
    shmctl(segment_id, IPC_RMID, NULL);
}

return 0;
}
```

- Communications in Client-Server Systems
 - Sockets
 - Remote Procedure Calls
 - Remote Method Invocation (Java)
 - gRPC
 - open source remote procedure call (RPC)
 - Uses HTTP/2, Protocol Buffers
 - Example NFS
 - Pipes
- Sockets
 - A socket is defined as an endpoint for communication
 - Concatenation of IP address and port
 - The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
 - Communication consists between a pair of sockets

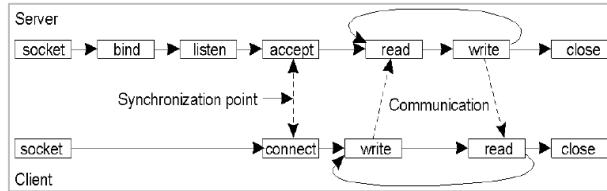


- System Calls for TCP/IP Sockets

<i>System call</i>	<i>Who calls it</i>	<i>Meaning</i>
Socket	Server / Client	Create a new communication endpoint
Bind	Server	Attach a local address and port to a socket
Listen	Server	Define how many clients can be queued
Accept	Server	Block until a connection request arrives
Connect	Client	Actively attempt to establish a connection
Write	Server / Client	Send some data over the connection
Read	Server / Client	Receive some data over the connection
Close	Server / Client	Release the connection

- TCP/IP Communication

- Connection-oriented



- Java Example

```

import java.net.*;
import java.io.*;

public class DateServer
{
    public static void main(String[] args)  {
        try {
            ServerSocket sock = new ServerSocket(10117);

            // now listen for connections
            while (true) {
                Socket client = sock.accept();
                // we have a connection

                PrintWriter pout = new
PrintWriter(client.getOutputStream(), true);
                // write the Date to the socket
                pout.println(new
java.util.Date().toString());

                // close the socket and resume listening
for more connections
                client.close();
            }
        } catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}

import java.net.*;
import java.io.*;

public class DateClient
{
    public static void main(String[] args)  {
        try {
            // this could be changed to an IP
name or address other than the localhost
            Socket sock = new
Socket("127.0.0.1",10117);
            InputStream in =
sock.getInputStream();
            BufferedReader bin = new
BufferedReader(new InputStreamReader(in));

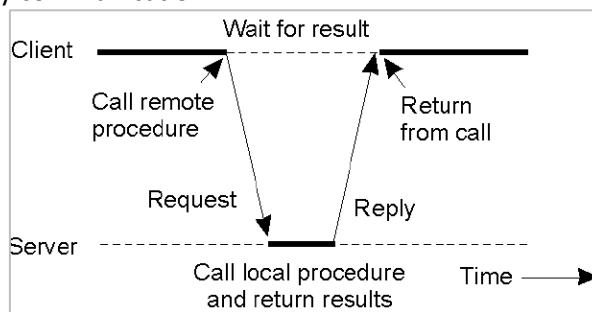
            String line;
            while( (line = bin.readLine()) !=
null)
                System.out.println(line);

            sock.close();
        } catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}

```

- Remote Procedure Call (RPC)

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
 - Stubs - client-side proxy for the actual procedure on the server
 - The client-side stub locates the server and marshalls the parameters
 - The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
 - RPC between client and server program
 - Request-reply communication



- Pipes in Practice

- Pipe between two processes
 - ps ax | less **or** ps ax | grep init **or** ps ax | grep init | awk '{print \$1}'
 - Named pipes
 - mkfifo tompipe
 - ls -l > tompipe

- ls -l > tompipe
- Pipes and network with netcat
 - cat web.sh | nc -l 8080
 - nc bocek.ch 8080 > web.sh (unencrypted!!)
- Ending Statement
 - Other predictions became true ... but continued to change.
 - "Where a calculator on the ENIAC is equipped with 18,000 vacuum tubes and weighs 30 tons, computers in the future may have only 1,000 vacuum tubes and weigh only 1.5 tons. (Popular Mechanics, 1949)

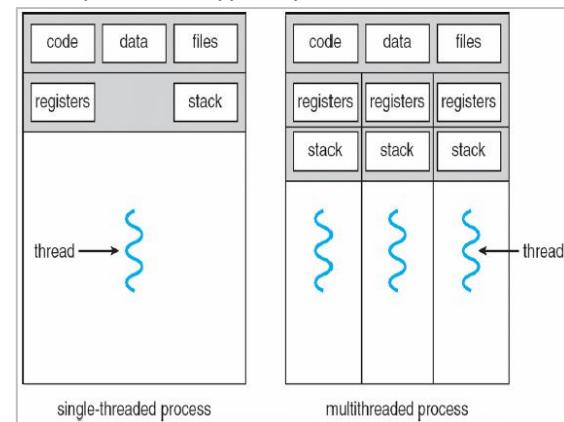
V4 Chapter 4: Threads

- Objectives
 - To introduce the notion of a thread — a fundamental unit of CPU utilization, the basis of multithreaded computer systems
 - To briefly discuss the APIs for the Pthreads, and Java thread libraries
 - To examine issues related to multi-threaded programming
- Topics
 - Multi-threading Models
 - Thread Libraries
 - Threading Issues
 - Operating System Examples
 - Linux Threads
- Multi-core Programming
 - All multi-core systems putting pressure on programmers, challenges include
 - Dividing activities
 - Define separate independent concurrent tasks
 - Balance
 - Identify tasks with equal work load
 - Data splitting
 - Divide data to be processed concurrently
 - Data dependency
 - Avoid serial dependencies of computations and synchronizations
 - Testing and debugging
 - Unknown parallel execution path and order
- Threads
 - Smallest unit of CPU utilization managed by the operating system scheduler
 - Threads are lightweight processes
 - Sharing code, data, and files
 - But have separate registers and stacks
 - Threads are contained inside a process
 - Multiple threads can exist within the same process and share resources, while different processes do not share their resources
- Comparison: Threads and Processes
 - Processes are typically independent, while threads exist as subsets of a process
 - Processes carry considerably more state information than threads, whereas multiple threads within a process share process state as well as memory and other resources
 - Processes have separate address spaces, whereas threads share their address space
 - Processes interact only through system-provided interprocess communication mechanisms

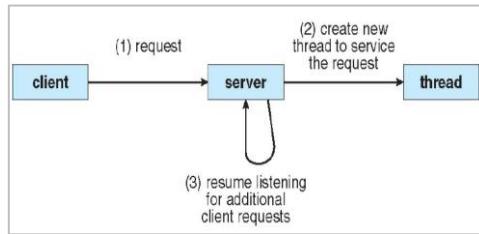
- Context switching between threads in the same process is typically faster than context switching between processes

- Single and Multi-threaded Processes

- Benefits:
 - Responsiveness
 - Resource Sharing
 - Economy
 - Scalability



- Multi-threaded Server Architecture

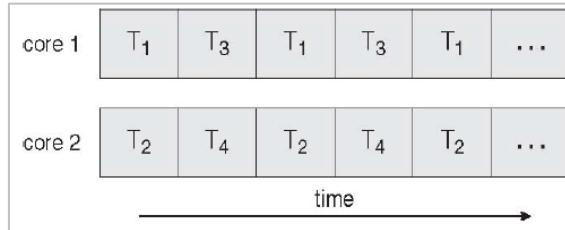


- Threaded Single/Multi-core Systems

- Concurrent execution on single core



- Parallel execution on multi-core



- Challenges:

- Dividing Activities
- Balance
- Data Splitting
- Data Dependency
- Testing and Debugging

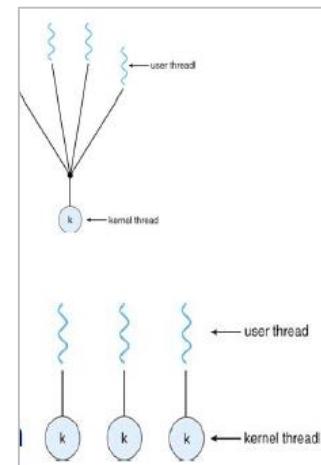
- Kernel Threads

- At least one kernel thread exists within each process
 - Created and scheduled by the OS kernel
- One kernel thread assigned to each (logical) CPU core
 - Unit of independent process, thread scheduling
- Support for kernel threads
 - Linux
 - Mac OS X
 - Solaris
 - Tru64 UNIX
 - Windows XP/2000

- User Threads

- User and application program thread management done above kernel by **user-level threads library**

- Ultimately a relationship must exist between user threads and kernel threads
 - **Many-to-one** Many to one prohibits multithreading within user process
 - No concurrent execution as only one user thread can run on a CPU core
 - All user threads get blocked if underlying kernel thread is blocked
 - **One-to-one** and many-to-many implementations enable multi-processor parallelism
- Threading
 - Semantics of **fork()** and **exec()** system calls
 - Does fork() duplicate only the calling thread or all threads?
 - **Thread cancellation** of target thread
 - Asynchronous cancellation terminates the target thread immediately
 - Deferred cancellation allows the target thread to periodically check if it should be cancelled
 - **Thread pools**
 - Usually slightly faster to service a request with an existing thread than create a new thread
 - Allows the number of threads in the application(s) to be bound to the size of the pool
 - **Thread-specific data**
 - Allows each thread to have its own copy of data
 - Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
 - **Signal handling**
 - ... next slide
- Signal Handling
 - Signals are used in UNIX systems to **notify a process** that a particular **event** has occurred
 - A **signal handler** is used to process signals
 - Signal is generated by particular event
 - Signal is delivered to a process
 - Signal is handled
 - **Handling options**
 - Deliver the signal to the thread to which the signal applies
 - Deliver the signal to every thread in the process
 - Deliver the signal to certain threads in the process
 - Assign a specific thread to receive all signals for the process
- Thread Libraries
 - Thread libraries provide programmers with API for creating and managing threads
 - Two primary ways of implementing
 - Library entirely in user space
 - Kernel-level library supported by the OS
 - Three very common user-level thread libraries
 - POSIX Pthreads
 - Win32 threads
 - Java threads
 - Operating thread systems:
 - Linux
- Pthreads



- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
 - [http://standards.ieee.org/findstds/interps/1003-1c-95_int/\(1996\)](http://standards.ieee.org/findstds/interps/1003-1c-95_int/(1996))
 - API specifies behavior of thread library
 - Implementation is up to the developer of the library
- Common in UNIX-like operating systems
 - Solaris, Linux, Mac OS X
 - Win32 threads quite similar to Pthreads

- Java Threads
 - Java threads are managed by the JVM. Threads determine the fundamental program execution model in Java, combined with a rich set of features for creation of threads and their management
 - Typically implemented using the threads model provided by underlying OS
 - Java threads may be created by two alternatives
 - Create a new class derived from the Thread class and override run() method
 - Implementing the Runnable interface

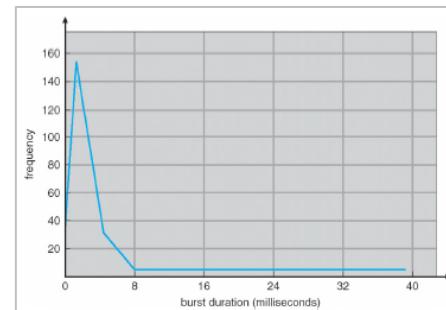
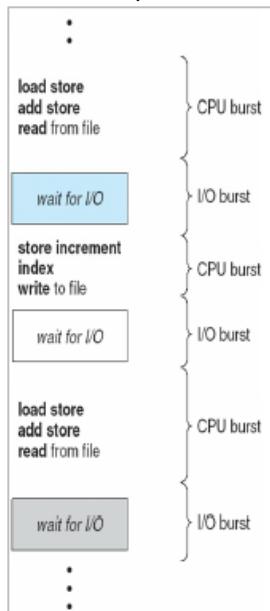
- Linux Threads
 - Linux refers to them as tasks rather than threads
 - Thread creation is done through clone() system call
 - Processes are created by fork() system call (Chapter 3)
 - clone() allows a child task to share the address space of the parent task (process)

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.

V5 Chapter 5: Scheduling

- Objectives
 - To introduce CPU scheduling, basis for multi-programmed operating systems
 - To describe selected CPU scheduling algorithms
 - To discuss evaluation criteria for selecting a CPU scheduling
- Topics
 - Basic Concepts
 - Scheduling Criteria
 - Scheduling Algorithms
 - Multiple-Processor Scheduling
 - Operating Systems Examples
- Basic Concepts
 - Maximum **CPU utilization** only obtained with concurrent multi-programming

- **CPU-I/O Burst Cycle**
 - Process execution
 - consists of a cycle of CPU execution and I/O wait
- **CPU burst distribution**
 - Measured extensively
 - Vary greatly from processes, computers, applications
- Alternating Sequence of CPU and I/O Bursts

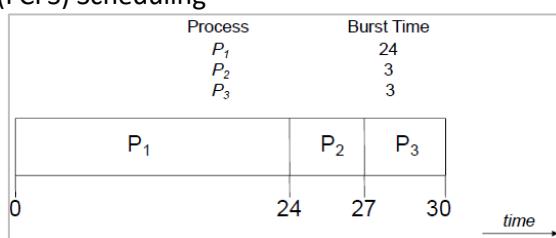


- Process States (Reminder)
 - As a process executes, it frequently changes states
 - new: the process is being created
 - running: instructions are being executed
 - waiting: the process is waiting for some event to occur
 - ready: the process is waiting to be assigned to a processor
 - terminated: the process has finished execution
- State transition diagram:

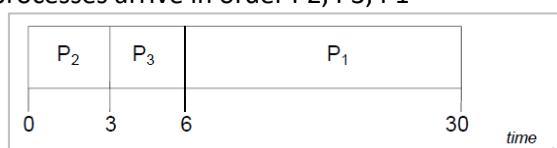

```

graph TD
    new((new)) -- admitted --> ready((ready))
    ready -- interrupt --> waiting((waiting))
    ready -- exit --> terminated((terminated))
    waiting -- "I/O or event completion" --> ready
    waiting -- "I/O or event wait" --> running((running))
    running -- "scheduler dispatch" --> ready
    
```
- CPU Scheduler
 - Selects from among processes in memory that are **ready to execute** and **allocates CPU** to one of them
 - Scheduling **decisions** may take place when a process:
 - 1. Switches from running to waiting state
 - 2. Switches from running to ready state
 - 3. Switches from waiting to ready
 - 4. Terminates
 - Scheduling under 1 and 4 is **non-preemptive**
 - Processes cooperate by giving up the CPU voluntarily
 - All other scheduling is **preemptive**
 - Running CPU execution interrupted, triggered by outside event
- Dispatcher Module
 - Dispatcher module gives control of the CPU to the process selected by the short-term scheduler

- This involves:
 - Switching context
 - Switching to user mode
 - Jumping to the proper location in the user program to restart that program
- Dispatch latency
 - Time it takes for the dispatcher to stop one process and start another running
 - Must be very low to avoid overhead
- Scheduling Criteria
 - **CPU utilization** - keep the CPU as busy as possible
 - **Throughput** - # of processes that complete their execution per time unit
 - **Turnaround time** - amount of time to execute a particular process
 - **Waiting time** - amount of time a process has been waiting in the ready queue
 - **Response time** - amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)
- First Come First Serve (FCFS) Scheduling

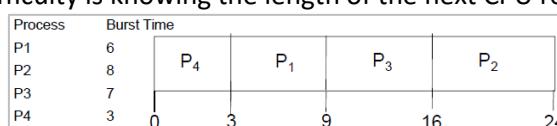


- Suppose that processes arrive in order P_1, P_2, P_3
- The **Gantt chart** (bars for start/end times) for the schedule is
 - Waiting time for $P_1 = 0; P_2 = 24; P_3 = 27$
 - Average waiting time: $(0 + 24 + 27)/3 = 17$
- Suppose that processes arrive in order P_2, P_3, P_1



- The Gantt chart for the schedule is
 - Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
 - Average waiting time: $(6 + 0 + 3)/3 = 3$
 - Much better than previous case
 - Convoy effect on short processes behind long processes

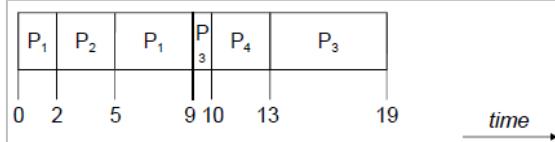
- Shortest Job First (SJF) Scheduling
 - Associate with each process the length of its next CPU burst and use these lengths to schedule the process with the shortest time to run and complete first
 - SJF is optimal
 - Gives minimum average waiting time for given set of processes
 - The difficulty is knowing the length of the next CPU request



- SJF scheduling chart
 - Average waiting time = $(3 + 11 + 16 + 9 + 0) / 4 = 7$
- Shortest Remaining Time (SRT) Scheduling
 - **Preemptive** SJF scheduling, allowing new processes to interrupt the currently running one

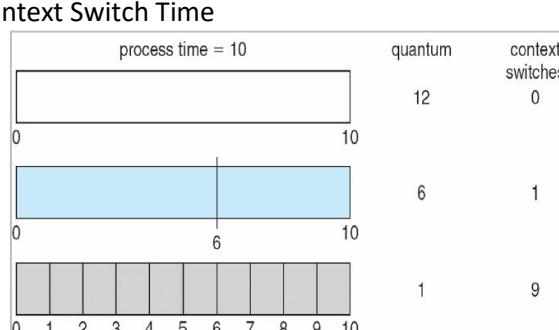
Process	Arrival Time	Burst Time
P_1	0.0	6
P_2	2.0	3
P_3	4.0	7
P_4	10.0	3

- SRT scheduling chart



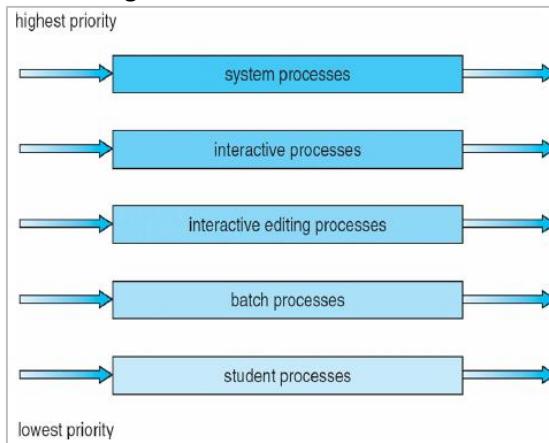
$$\text{○ Average waiting time} = (3 + 0 + 5+3 + 0) / 4 = 2.75$$

- Round Robin (RR) Scheduling
 - Process gets a small unit of CPU time (time quantum),
 - After time elapsed (usually 10-100 ms), process preempted and added to the end of the ready queue
 - If there are n processes in the ready queue and the time quantum is q, each process 1/n of the CPU time in chunks of at most q time units at once.
 - No process waits more than $(n-1)q$ time units
 - n processes run in “parallel” at 1/n of available CPU speed
 - Performance
 - q too large \Rightarrow FIFO
 - q too small \Rightarrow many context switches, overhead is too high
 - q must be large with respect to context switch cost
- Time Quantum and Context Switch Time

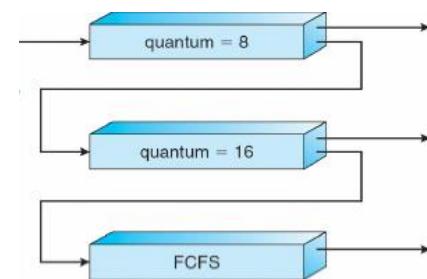


- Example of RR with Time Quantum = 4
 - The Gantt chart is
- Priority Scheduling
 - Priority number (integer) is associated with each process
 - CPU is allocated to the process with the highest priority (smallest integer \equiv highest priority)
 - FCFS is a priority scheduling, where priority is the process' arrival time: Non-preemptive
 - SJF is a priority scheduling, where priority is the (predicted) next CPU burst time: Preemptive
 - Problem \equiv Starvation
 - Low priority processes may never execute

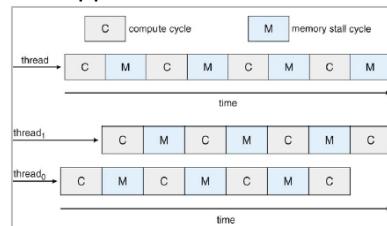
- Solution \equiv Aging
 - As time progresses increase the priority of the process
- Multi-level (ML) Queue Scheduling
 - Ready queue is partitioned into separate queues, e.g.:
 - Foreground (interactive)
 - Background (batch)
 - Each queue has its own scheduling algorithm
 - Foreground - RR
 - Background - FCFS
 - Scheduling must be done between the queues
 - Fixed priority scheduling, (i.e., serve all from foreground then from background): Possibility of starvation
 - Time slice: each queue gets a certain amount of CPU time, which it can schedule amongst its processes
 - I.e., 80% to foreground in RR and 20% to background in FCFS
- Example for ML Queue Scheduling



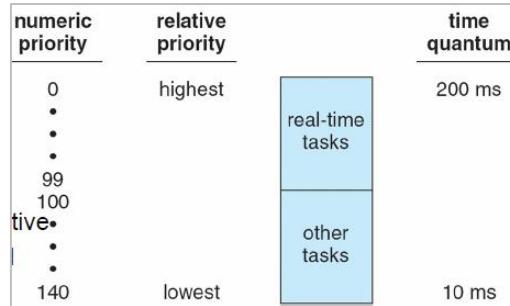
- Multi-level Feedback Queue (MLF)
 - A process can move between the various priority queues
 - Aging can be implemented this way
 - Multi-level feedback queue scheduler defined by the following parameters
 - Number of queues
 - Scheduling algorithms for each queue
 - Method used to determine when to upgrade a process
 - Method used to determine when to demote a process
 - Method used to determine which queue a process will enter when that process needs service
- Example of a 3-Level MLF
 - Three queues
 - Q_0 – RR with time quantum 8 ms
 - Q_1 – RR time quantum 16 ms
 - Q_2 – FCFS
 - Scheduling
 - New job enters queue Q_0 which is served RR. When it gains CPU, job receives 8 ms. If it does not finish in 8ms, job is preempted and moved to queue Q_1
 - At Q_1 job is again served RR and receives 16 additional ms. If it still does not complete, it is preempted and moved to queue Q_2
 - Final queue Q_2 is served FCFS



- Multiple Processor Scheduling
 - CPU scheduling is more complex when multiple CPUs are available
 - **Homogeneous processors** within a multi-processor assumed
 - **Asymmetric multiprocessing (AMP)**
 - Only one processor accesses system shared data structures
 - Alleviating the need for data sharing
 - **Symmetric multiprocessing (SMP)**
 - Each processor is self-scheduling
 - All processes in common ready queue or each has its own private queue of ready processes
 - Supported, e.g., in Windows XP, Mac OS X, Linux, Solaris
 - **Processor affinity**
 - Process has affinity for processor on which it is currently running
 - Hard affinity
 - Process runs only on one particular CPU
 - Soft affinity
 - Process can migrate to another CPU, but reluctantly done
 - **Process migration** involves cache invalidation on one CPU and repopulation on the other
- Multi-core Processors
 - Trend since 2010 to place multiple processor cores on same physical chip
 - Faster, consumes less power
 - Multiple kernel threads per core also growing
 - Takes advantage of memory stall to make progress on another thread, while memory retrieve happens



- Example 1: Linux Scheduling/Priorities (1)
 - Constant order O(1) scheduling time independent of number of processes
 - Suitable for large and SMP systems
 - Two priority ranges
 - Real-time
 - Real-time range from 0 to 99
 - Time sharing (nice values)
 - Nice value from 100 to 140
 - Longer time quantum for higher priority
 - Time sharing tasks have dynamic priorities, adjusted after expired time quantum
 - Level of interactivity determines ±5 adjustment
 - Long sleep times → more interactive
 - Short sleep times → less interactive
 - CPU bound



- Example 2: Java Thread Scheduling
 - JVM uses a **preemptive, priority-based scheduling algorithm**
 - FIFO queue is used if there are multiple threads with the same priority
 - JVM **schedules** a thread to run when
 - The currently running thread exits the runnable state
 - A higher priority thread enters the runnable state
 - Note: the JVM does not specify whether threads are time-sliced or not
- Ending Statement
 - There is no reason anyone would want a computer in their home.

V5 Chapter 6: Synchronization

- Communications and Synchronization
 - Objectives
 - Recalling communications between processes
 - Introduce synchronization and the critical section problem, whose solutions can be used to ensure the consistency of shared data
 - Present both software and hardware solutions of the critical-section problem
 - The Critical Section Problem
 - Peterson's Solution
 - Synchronization Hardware
 - Semaphores
 - Classical Problems of Synchronization
- Inter-process Communications (Recall)
 - Processes within a system may be
 - Independent: process cannot affect or be affected by the execution of another process
 - Cooperating: process can affect or be affected by the execution of another process
 - Reasons for cooperating processes:
 - Information sharing
 - Computation speed-up
 - Modularity
 - Convenience
 - Cooperating processes need Inter-process Communication (IPC) mechanisms (Chapter 3)
- Aside: Implementation Questions
 - How are links established?
 - Can a link be associated with more than two processes?
 - How many links can there be between every pair of communicating processes?
 - What is the capacity of a link?
 - Is the size of a message that the link can accommodate fixed or variable?
 - Is a link unidirectional or bi-directional?

- How are parallel/concurrent processes being handled?
- General Concurrent Process Structure

```

while (TRUE) {
    entry section
    critical section // prone to
                      // race conditions
    exit section
    remainder section
}
  
```

- A race condition is an undesirable situation that occurs, when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly.

- Race Conditions
 - Two simple execution examples
 - count++ could be executed on the CPU as


```

register1 = count
register1 = register1 + 1
count = register1
          
```
 - count-- could be executed on the CPU as


```

register2 = count
register2 = register2 - 1
count = register2
          
```

- Consider this execution interleaving with "count = 5" initially

```

S0: producer executes register1 = count {register1 = 5}
S1: producer executes register1 = register1 + 1
                 {register1 = 6}
S2: consumer executes register2 = count {register2 = 5}
S3: consumer executes register2 = register2 - 1
                 {register2 = 4}
S4: producer executes count = register1 {count = 6 }
S5: consumer executes count = register2 {count = 4}
  
```

- Major problem!
- Solution to Critical Section (CS) Problem
 - 1. Mutual Exclusion: If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
 - 2. Progress: If no process is executing in its critical section and there exist some processes that wish to enter their critical section, the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
 - 3. Bounded Waiting: A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
 - Under the assumption that each process executes at a non-zero speed, but w/o any assumption concerning relative speeds of N processes.
- Peterson's Solution
 - The two processes software solution
 - Assume that the LOAD and STORE instructions are atomic, they cannot be interrupted (not true in many of today's architectures!)
 - These two processes share two variables:
 - int turn;
 - Boolean flag[2]

- Variable turn indicates whose turn it is to enter the critical section.
- Flag array is used to indicate, if a process is ready to enter the critical section.
flag[i] = true implies that process PI is ready!
- Algorithm for Process Pi

```

while (TRUE) {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
                                // do nothing
    critical section

    flag[i] = FALSE;

    remainder section
};

```

- Synchronization Hardware
 - Many systems provide hardware support for critical section code
 - Uni-processors could disable interrupts
 - Currently running code would execute without preemption
 - Generally too inefficient on multi-processor systems
 - Operating systems using this are not broadly scalable
 - Modern machines provide special atomic hardware instructions
 - Atomic = non-interruptable
 - Either: test memory word and set value
 - Or: swap contents of two memory words
- Solution to Critical Section Problem Using Locks

```

while (TRUE) {

    acquire lock
    critical section
    release lock

    remainder section
};

```

- TestAndSet Instruction
 - TestAndSet function calls, if executed simultaneously on multiple CPUs, will be sequentialized in some arbitrary order, but are atomically executed


```

boolean TestAndSet (boolean *locked) // modifiable parameter
{
    boolean answer = *locked;
    *locked = TRUE;
    return answer;
}

```
 - Return value indicates the current lock status
 - If false a new lock on it was acquired
- Solution using TestAndSet
 - If computer architecture supports TestAndSet() instruction, define a shared Boolean variable locked
 - Initialized to FALSE.
 - Solution is called **busy waiting**

```

while (TRUE) {
    while (TestAndSet(&locked));
    // do nothing, busy wait
    // critical section ...
    locked = FALSE;

    // remainder section
};
  
```

- Swap Instruction

- Swap (in contrast to TestAndSet()) instruction operates on two parameters and is executed atomically
 - Arbitrarily sequentialized, data structure as follows:

```

void Swap (boolean *lock, boolean *key)
           // modifiable parameters
{
    boolean temp = *lock;
    *lock = *key;
    *key = temp
  
```

- The modified return argument key indicates the current lock status, mutual exclusion reached, not bounded wait

- Solution using Swap Instruction

- Shared Boolean variable
 - Lock initialized to FALSE
 - Each process has a local Boolean variable key
- Solution for process

```

while (TRUE) {
    key = TRUE;
    while (key == TRUE)
        Swap(&lock, &key); // busy wait
        // critical section ...
    lock = FALSE;
    // remainder section
};
  
```

- Bounded-waiting Mutual Exclusion with TestAndSet()

- Bounded-waiting more difficult than mutual exclusion
 - Needs extra data structures to identify waiting processes, Boolean array waiting[n]

```

while (TRUE) {
    waiting[i] = TRUE;           // process i waiting for access to CS
    key = TRUE;
    while (waiting[i] && key) // wait for lock from other process
        key = TestAndSet(&lock); // busy wait
    waiting[i] = FALSE;
    // critical section
    j = (i+1) % n;
    while ((j != i) && !waiting[j]) // check for waiting processes
        j = (j+1) % n;
    if (j == i)
        lock = FALSE;          // no one waiting, release lock
    else
        waiting[j] = FALSE;    // transfer lock
    // remainder section
};
  
```

- Semaphores

- Synchronization tool that does not require busy waiting
 - Semaphore S as an integer variable
- Two standard operations modify S only
 - wait() and signal()
 - Originally called P() and V() to lock or release semaphore
 - Less complicated
 - Can only be accessed via two indivisible (atomic) operations

```

wait(S) {
    while S <= 0 ; // no-op
    S--;
}
signal(S) {
    S++;
}

```

- Semaphore as a General Synchronization Tool
 - Binary semaphore (also known as mutex locks)
 - integer value can only be 0 or 1
 - Simpler to implement (simple mutual exclusion)
 - Counting semaphore
 - integer value can range over an unrestricted domain
 - Initial value indicates level of parallelism of passing this sem.
 - Counting semaphore S can be implemented using a binary semaphore
 - Control access to finite limited resources


```

semaphore mutex; // initialized to 1
while(TRUE) {
    wait (mutex);
    // critical section
    signal (mutex);
    // remainder section
  
```
- Semaphore Implementation
 - Must guarantee that no two processes can execute wait() and signal() on the same semaphore at the same time
 - Thus, the implementation becomes the CS problem where the wait and signal code are placed in the CS!
 - Software impl. now has busy waiting in the critical section
 - But implementation code is short
 - Little busy waiting, if critical section is rarely occupied
 - Note that applications may spend lots of time in critical sections and, therefore, this is not a good solution
 - Modified implementation, where processes are blocked, when waiting for a semaphore
- Semaphore Implementation without Busy Waiting
 - With each semaphore a waiting queue is associated. Each entry in a waiting queue has two data items:
 - Value (of type integer)
 - Pointer to next record in the list


```

typedef struct {
    int value;
    struct process *list;
} semaphore;
  
```
 - Two operations:
 - block: to place the process invoking the operation on the appropriate waiting queue
 - wakeup: to remove one of the processes in the waiting queue and place it in the ready queue
 - Block/wakeup provided by operating system as basic system calls
- Semaphore Implementation
 - Implementation of wait wait(semaphore *S) {

```

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

```

- Implementation of signal: signal(semaphore *S) {

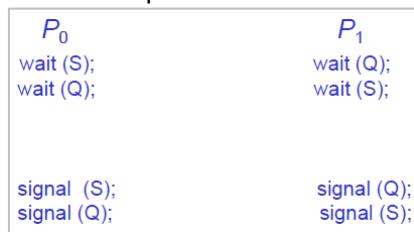
```

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

```

- Deadlocks

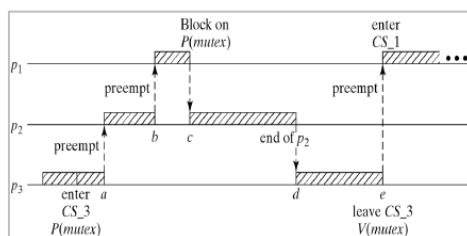
- Deadlock (cf. Chapter 7)
 - Two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
 - Let S and Q be two semaphores initialize to 1



- Starvation and Priority Inversion

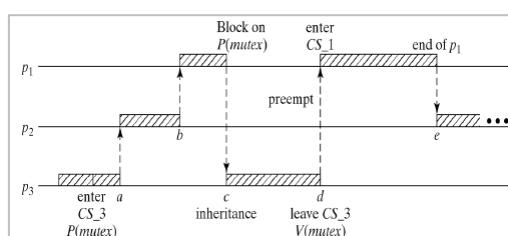
- Starvation
 - Indefinite blocking
A process may never be removed from the semaphore queue in which it is suspended
- Priority Inversion
 - Scheduling problem when lower-priority process holds a lock needed by higher-priority process
 - See priority inheritance protocol

- Priority inversion



- Assume priority order $p_1 > p_2 > p_3$
 - (unrelated) p_2 may delay p_1 indefinitely
- Naïve “solution”: always run CS at priority of highest process that shares the CS
- Problem: p_1 cannot interrupt lower-priority process inside CS – a different form of priority inversion

- Priority inheritance



- Solution: Dynamic Priority Inheritance
 - Inherit priority from higher blocked process
 - p3 is in its CS
 - p1 attempts to enter its CS
 - p3 inherits p1's (higher) priority for the duration of CS
- Classical Problems of Synchronization
 - Bounded Buffer Problem (see Chapter 3)
 - Readers and Writers Problem
 - Dining Philosophers Problem
- Readers and Writers Problem
 - Data set is shared among concurrent processes
 - Readers only read data set; they do not perform any updates
 - Writers can both read and write
 - Problem:
 - Allow multiple readers to read at the same time , but only one single writer can access the shared data at that same time
 - Shared Data
 - Data set
 - Semaphore mutex initialized to 1
 - Used to exclude others from modifying the buffer state concurrently
 - Semaphore wrt initialized to 1
 - Used to serialize exclusive write operations
 - Integer readcount initialized to 0
 - Used to indicate concurrent readers

- Readers and Writers Processes

```

while (TRUE) {
    wait(mutex);
    // operate safely on read count
    readcount++;
    if (readcount == 1)
        wait(wrt);
    // prevent any writers at this time
    signal(mutex);
    // reading is performed
    wait(mutex);
    readcount--;
    if (readcount == 0)
        signal(wrt);
    // make sure writers can update data
    signal(mutex);
};
  
```

- Dining Philosophers Problem

- Shared data
 - Bowl of rice (data set)
 - Semaphore chopstick[5] initialized to 1
 - Each one needs two sticks to actually eat
 - Can also use alternating forks and knives between odd number of hungry philosophers
- The structure of Philosopher I



```

while (TRUE) {
    wait(chopstick[i]); // get left chopstick
    wait(chopStick[ (i + 1) % 5 ]);
                                // get right chopstick
    // eat

    signal(chopstick[i]);
    signal(chopstick[ (i + 1) % 5 ] );

    // think
};

```

- Problems with Semaphores
 - Incorrect use of semaphore operations (timing errors)
 - signal (mutex) wait (mutex)
 - wait (mutex) ... wait (mutex)
 - Omitting of wait (mutex) or signal (mutex) (or both)
- Monitors
 - A high-level abstraction that provides a convenient and effective mechanism for process synchronization
 - Only one process may be active within the monitor at any time
 - CS mutex applied to each monitor function

```

monitor monitor-name
{
    // shared variable declarations
    procedure F1(...) { ... }
    ...
    procedure Fn(...) { ... }
    Initialization_code ( ... ) { ... }
    ...
}

```

- Monitor Mutex using Semaphores
 - Semaphore variables


```

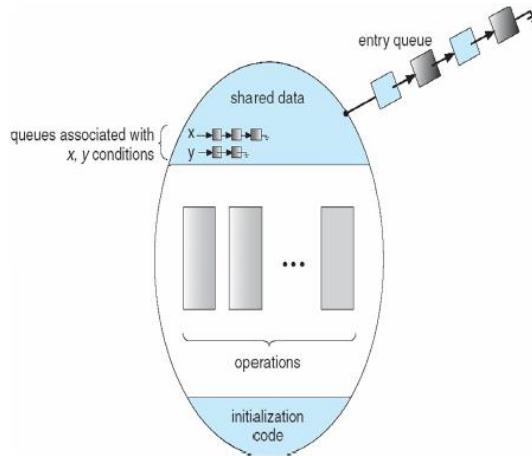
semaphore mutex; // (initially = 1)
semaphore next; // (initially = 0)
int next_count = 0;

```
 - Each procedure F will be replaced by


```

wait(mutex);
... // body of F;
if (next_count > 0)
    signal(next)
else
    signal(mutex);

```
 - Mutual exclusion is ensured within a monitor
 - Mutex lock is passed rather than released
 - Acquired from one to another waiting process
- Condition Variables
 - Condition x, y
 - Two operations on a condition variable
 - x.wait()
 - A process that invokes the operation is suspended.
 - x.signal() i l()
 - Resumes one of the processes (if any) that invoked x.wait ()
 - Note that actual re-scheduling after suspended and resumed processes is done only at the end of the operation on condition variables
- Monitor with Condition Variables



- Monitor Implementation

- For each condition variable x

```
semaphore x_sem; // (initially = 0)
int x_count = 0;
```

- The operation x.wait() can be implemented as

```
x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```

- The operation x.signal() can be implemented as

```
if (x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

- Monitor Approach to Dining Philosophers

- Each philosopher i invokes the operations pickup() and putdown() in the following sequence:

```
DiningPhilosophers.pickup(i);
// EAT
DiningPhilosophers.putdown(i);
```

- Monitor Solution to Dining Philosophers

```
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 left and right neighbors
        test((i+4) % 5);
        test((i+1) % 5);
    }
}
```

```
void test(int i) {
    if ((state[i] == HUNGRY) &&
        (state[(i+4)%5] != EATING) &&
        (state[(i+1)%5] != EATING)) {
        state[i] = EATING;
        self[i].signal();
    }
}
initialization_code() {
    for (int i = 0; i < 5; i++)
        state[i] = THINKING;
}
```

- Monitor to Allocate a Single Resource

```

monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = TRUE;
    }
    void release() {
        busy = FALSE;
        x.signal();
    }
    initialization code() {
        busy = FALSE;
    }
}

```

- Linux Synchronization (Example)
 - Linux Versions
 - Prior to kernel Version 2.6, Linux disables interrupts to implement short critical sections
 - Version 2.6 and later, Linux follows a fully preemptive approach approach
 - Linux provides
 - Semaphores
 - Spin locks
- Pthreads Synchronization (Example)
 - Pthreads API is fully OS-independent
 - The API provides
 - mutex locks
 - Condition variables
 - Non-portable extensions include
 - Read-write locks
 - Spin locks
- Mutual Exclusion
 - Access to critical sections in Pthreads can be controlled using mutex locks
 - A thread entering a critical section must request to acquire a lock on it first
 - Cannot go ahead , when the lock acquisition fails
 - Pthreads API provides following functions for handling mutex-locks

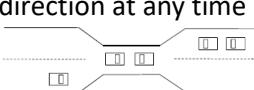

```
int pthread_mutex_lock(pthread_mutex_t *mutex_lock);
int pthread_mutex_unlock(pthread_mutex_t *mutex_lock);
int pthread_mutex_init(pthread_mutex_t *mutex_lock, const
pthread_mutexattr_t *lock_attr);
```
- Pthreads' Types of Mutexes
 - Pthreads support three types of mutexes
 - A **normal mutex** deadlocks, if a thread that already has a lock tries a second lock on it
 - A **recursive mutex** allows a single thread to lock a mutex as many times as it wants, it simply increments a count on the number of locks, a lock is relinquished by a thread when the count becomes zero
 - An **error check mutex** reports an error, when a thread with a lock tries to lock it again (as opposed to deadlocking in the first case, or granting the lock, as in the second case)
 - The type of the mutex can be set in attributes object before it is passed at time of initialization.
- Overheads of Locking

- Locks represent serialization points, since critical sections must be executed by threads one after the other
 - Encapsulating large segments of the program within locks can lead to significant performance degradation
- It is often possible to reduce idling or wait-time overhead associated with locks using an alternate function, `pthread_mutex_trylock`
 - `int pthread_mutex_trylock(pthread_mutex_t *mutex_lock);`
 - `pthread_mutex_trylock` is typically much faster than `pthread_mutex_lock` on typical systems, since it does not have to deal with queues associated with locks for multiple threads waiting on the same lock
- Ending Statement
 - The value of technology or mechanisms becomes only visible to those, who believe in the future ...
 - "This 'telephone' has too many shortcomings to be seriously considered as a means of communication. The device is inherently of no value to us."

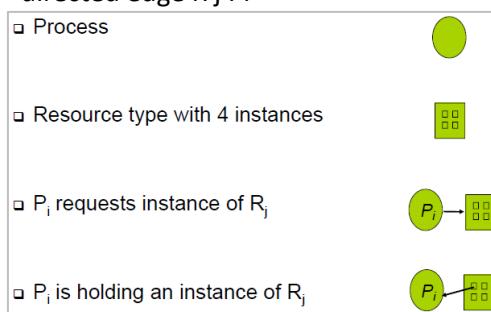
V6 Chapter 7: Deadlocks

- Deadlocks
 - Objectives
 - To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
 - To present a number of different methods for preventing or avoiding deadlocks in a computer system
 - Topics
 - The Deadlock Problem
 - System Model
 - Deadlock Characterization
 - Methods for Handling Deadlocks
 - Deadlock Avoidance
 - Deadlock Detection
 - Recovery from Deadlocks
- The Deadlock Problem
 - Assume a set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
 - Example
 - System has 2 disk drives
 - P1 and P2 each hold one disk drive and each needs another one
 - Example:
 - Semaphores A and B, initialized to 1

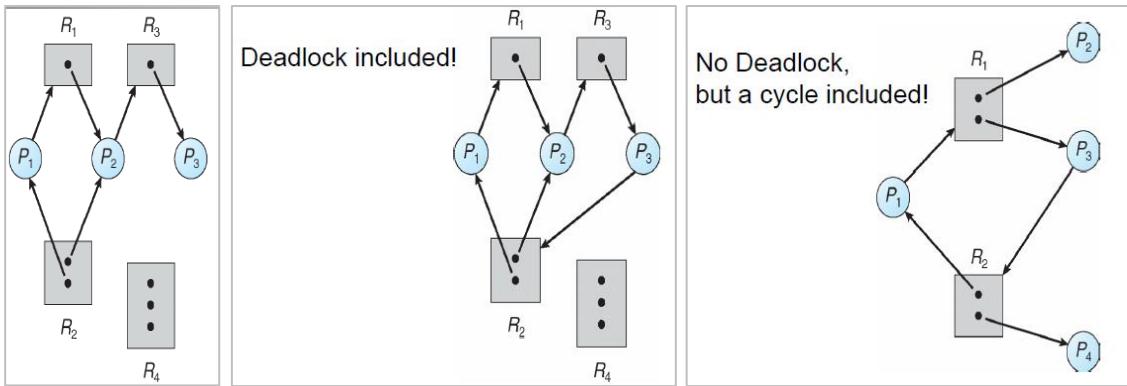


- Bridge Crossing Example
 - Traffic only possible in one direction at any time
 
 - Each entry section can be viewed as a resource
 - If a deadlock occurs, it can only be resolved, if one car backs up
 - Preempting the resource allocation and rollback
 - Several cars may have to back up, if a deadlock occurs

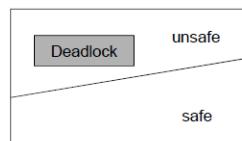
- Starvation is possible
 - Note: Most OSes do not prevent or deal with deadlocks
- System Model
 - Finite number of resource types R_1, R_2, \dots, R_m
 - CPU cycles, memory space, I/O devices
 - Each resource type R_i has W_i instances
 - Each process utilizes a resource as follows (normal operation)
 - Request
 - Use
 - Release
 - Request and release are system calls!
- Deadlock Characterization
 - Deadlocks can arise only, if four conditions hold simultaneously (necessary conditions)
 - 1. Mutual exclusion
 - Only one process at a time can use a resource
 - 2. Hold and wait
 - A process holding at least one resource is waiting to acquire additional resources held by other processes
 - 3. No preemption
 - A resource can be released only voluntarily by the process holding it, after that process has completed its task
 - 4. Circular wait
 - There exists a set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource that is held by P_2 , ..., P_{n-1} is waiting for a resource that is held by P_n , and P_n is waiting for a resource that is held by P_0
- Resource Allocation Graph (1)
 - Deadlocks can be described precisely in terms of a directed graph, the Resource Allocation Graph
 - A set of vertices V and a set of edges E
 - V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all processes in the system
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system
 - Request edge - directed edge P_i, R_j
 - Assignment edge - directed edge $R_j \rightarrow P_i$



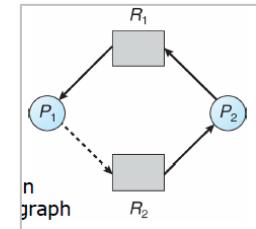
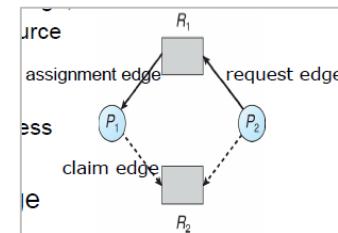
- Example: Resource Allocation Graph (1)



- Basic Observation and Facts
 - If a graph contains no cycles → no deadlock
 - If a graph contains a cycle
 - If only one instance per resource type → deadlock
 - If several instances per resource type → only the possibility of deadlocks exists
- Methods for Handling Deadlocks
 - Ensure that the system will never enter a deadlock state
 - Allow the system to enter a deadlock state and then recover
 - Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX
- Deadlock Avoidance
 - Restrain the ways resource requests can be made
 - Mutual Exclusion
 - Not required for sharable resources; must hold for non-sharable resources
 - Hold and Wait
 - Must guarantee that whenever a process requests a resource,
 - it does not hold any other (unrelated) resources
 - Requires process to request and see all its resources allocated before it begins with the execution, or allow the process to request resources only, when the process has none
 - Low resource utilization; starvation possible
 - No Preemption
 - If a process - holding some resources - requests another resource that cannot be allocated immediately, all resources currently being held are released
 - Released resources are added to the list of resources for which the process is waiting for
 - Process will be restarted only, when it can regain (a) its old resources as well as (b) the new ones that it is requesting for
 - Circular Wait
 - Impose a total ordering of all resource types
 - Require that each process requests resources in an increasing order of enumeration
- Safe State
 - Additional information on the “how” of resource requests, such as orders of resource needs or numbers of resources demanded.
 - When a process requests an available resource, the system must decide, if an immediate allocation leaves the system in a safe state!



- A system is in a safe state, if there exists a sequence $\langle P_1, P_2, \dots, P_n \rangle$ of ALL processes in the system such that for each P_i , resources that P_i can still request can be satisfied by currently available resources plus those resources held by all P_j , with $j < i$
 - Ordering of processes
- That is:
 - If P_i 's resource needs are not immediately available, P_i can wait until all $P_j < i$ have finished
 - When P_j can obtain resources needed, execute, return allocated resources, and terminate
 - When P_i is finished, P_i terminates, P_{i+1} can obtain its resources needed ...
- Avoidance Algorithms
 - Single instance of a resource type
 - Use a Resource Allocation Graph and cycle detection
 - Multiple instances of a resource type
 - Use Banker's Algorithm
- Resource Allocation Graph Scheme
 - Claim edge P_i indicated that process P_j may request resource R_j ; represented by a dashed line
 - Claim edge converts to request edge, when a process requests a resource
 - Request edge converted to an assignment edge, when the resource is allocated to the process
 - When a resource is released by a process, assignment edge converts back to a claim edge
 - Resources must be claimed a priori in the system R_j
- Resource Allocation Graph Algorithm
 - Suppose that process P_2 requests a resource R_2
 - The request can be granted only, if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph
 - Cycle detection algorithm on resource graph after potential assignment of resource to requesting process
 - Unsafe state, cycle in resource allocation graph
- Banker's Algorithm
 - Multiple instances of resources
 - Each process must claim maximum use a priori
 - When a process requests t a resource, it may have to wait
 - When a process gets all its resources, it must return them in a finite amount of time
- Data Structure for Banker's Algorithm
 - Let $n =$ number of processes and $m =$ number of resources types
 - Available
 - Vector of length m
 - $\text{Available}[j] = k$, there are k instances of resource type R_j available
 - Max
 - $n \times m$ matrix
 - $\text{Max}[i,j] = k$, process P_i may request at most k instances of resource type R_j
 - Allocation
 - $n \times m$ matrix.
 - $\text{Allocation}[i,j] = k$, process P_i is currently allocated k instances of R_j
 - Need
 - $n \times m$ matrix
 - $\text{Need}[i,j] = k$, process P_i may need k more instances of R_j to complete its task



- $\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$
- Safety Determination Algorithm
 - How to find out that a system is in a safe state?
Let Work and Finish be temporary vectors of length m and n, respectively
 - 1. Initialize
 - Work = Available
 - Finish[i] = false for $i = 0, 1, \dots, n-1$
 - 2. Find an index i such that both:
 - (a) Finish[i] = false
 - (b) Need \leq Work i
 - If no such i exists, go to step 4
 - 3. Work = Work + Allocation i
Finish[i] = true
go to step 2
 - 4. If Finish [i] == true for all i,
then the system is in a safe state, else unsafe
 - Requires $O(m \times n^2)$ operations
 - Tries to find possible sequence of processes finishing and releasing their resources
- Resource Request Algorithm for Process P
 - How to determine if a request can be safely granted?
 - Request = request vector for process P i 1.
 - Request $I[j] = k$, process P i wants k instances

1. If $\text{Request}_i \geq \text{Need}_i$, go to step 2

 - Otherwise, raise error condition, since process has exceeded its maximum claim
 2. If $\text{Request}_i \leq \text{Available}$, go to step 3

 - Otherwise P_i must wait, since resources are not yet available
 3. Pretend to allocate requested resources to P_i by modifying the state as follows:

 - Available = Available – Request $_i$
 - Allocation $_i$ = Allocation $_i$ + Request $_i$
 - Need $_i$ = Need $_i$ – Request $_i$
 - If safe \Rightarrow Resources are allocated and P_i can execute
 - If unsafe \Rightarrow P_i must wait, old resource allocation state restored

- Example of Banker's Algorithm

<p>5 processes P_0 through P_4</p> <ul style="list-style-type: none"> – 3 resource types – A (10 instances), B (5 instances), and C (7 instances) <p>Snapshot at time T_0:</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th></th><th><i>Allocation</i></th><th><i>Max</i></th><th><i>Available</i></th></tr> <tr> <th></th><th>A B C</th><th>A B C</th><th>A B C</th></tr> </thead> <tbody> <tr> <td>P_0</td><td>0 1 0</td><td>7 5 3</td><td>3 3 2</td></tr> <tr> <td>P_1</td><td>2 0 0</td><td>3 2 2</td><td></td></tr> <tr> <td>P_2</td><td>3 0 2</td><td>9 0 2</td><td></td></tr> <tr> <td>P_3</td><td>2 1 1</td><td>2 2 2</td><td></td></tr> <tr> <td>P_4</td><td>0 0 2</td><td>4 3 3</td><td></td></tr> </tbody> </table>		<i>Allocation</i>	<i>Max</i>	<i>Available</i>		A B C	A B C	A B C	P_0	0 1 0	7 5 3	3 3 2	P_1	2 0 0	3 2 2		P_2	3 0 2	9 0 2		P_3	2 1 1	2 2 2		P_4	0 0 2	4 3 3		<p>The content of the matrix <i>Need</i> is defined to be $\text{Max} - \text{Allocation}$</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th></th><th><i>Need</i></th></tr> <tr> <th></th><th>A B C</th></tr> </thead> <tbody> <tr> <td>P_0</td><td>7 4 3</td></tr> <tr> <td>P_1</td><td>1 2 2</td></tr> <tr> <td>P_2</td><td>6 0 0</td></tr> <tr> <td>P_3</td><td>0 1 1</td></tr> <tr> <td>P_4</td><td>4 3 1</td></tr> </tbody> </table>		<i>Need</i>		A B C	P_0	7 4 3	P_1	1 2 2	P_2	6 0 0	P_3	0 1 1	P_4	4 3 1
	<i>Allocation</i>	<i>Max</i>	<i>Available</i>																																								
	A B C	A B C	A B C																																								
P_0	0 1 0	7 5 3	3 3 2																																								
P_1	2 0 0	3 2 2																																									
P_2	3 0 2	9 0 2																																									
P_3	2 1 1	2 2 2																																									
P_4	0 0 2	4 3 3																																									
	<i>Need</i>																																										
	A B C																																										
P_0	7 4 3																																										
P_1	1 2 2																																										
P_2	6 0 0																																										
P_3	0 1 1																																										
P_4	4 3 1																																										

The system is in a safe state, since the sequence $< P_1, P_3, P_4, P_2, P_0 >$ satisfies the safety criteria

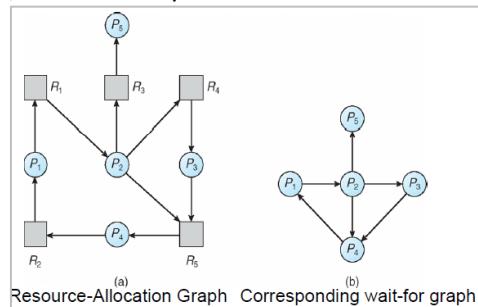
- Example: P1 Requests (1,0,2)

<p>Check that new Request \leq Available that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ true</p> <p>Check new possible state</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th></th><th><i>Allocation</i></th><th><i>Need</i></th><th><i>Available</i></th></tr> <tr> <th></th><th>A B C</th><th>A B C</th><th>A B C</th></tr> </thead> <tbody> <tr> <td>P_0</td><td>0 1 0</td><td>7 4 3</td><td>2 3 0</td></tr> <tr> <td>P_1</td><td>3 0 2</td><td>0 2 0</td><td></td></tr> <tr> <td>P_2</td><td>3 0 1</td><td>6 0 0</td><td></td></tr> <tr> <td>P_3</td><td>2 1 1</td><td>0 1 1</td><td></td></tr> <tr> <td>P_4</td><td>0 0 2</td><td>4 3 1</td><td></td></tr> </tbody> </table>		<i>Allocation</i>	<i>Need</i>	<i>Available</i>		A B C	A B C	A B C	P_0	0 1 0	7 4 3	2 3 0	P_1	3 0 2	0 2 0		P_2	3 0 1	6 0 0		P_3	2 1 1	0 1 1		P_4	0 0 2	4 3 1	
	<i>Allocation</i>	<i>Need</i>	<i>Available</i>																									
	A B C	A B C	A B C																									
P_0	0 1 0	7 4 3	2 3 0																									
P_1	3 0 2	0 2 0																										
P_2	3 0 1	6 0 0																										
P_3	2 1 1	0 1 1																										
P_4	0 0 2	4 3 1																										
<p>Executing safety algorithm shows that sequence $< P_1, P_3, P_4, P_0, P_2 >$ satisfies safety requirement</p> <ul style="list-style-type: none"> – Can next request for (3,3,0) by P_4 be granted? No, lack of R. – Can next request for (0,2,0) by P_0 be granted? No, unsafe! 																												

- Deadlock Detection

- If the system does NOT employ deadlock prevention or avoidance algorithms:
 - Allow the system to enter deadlock state
 - Apply detection algorithms
 - Apply pre-determined recovery scheme
- Single Instance of Each Resource Type
 - Maintain wait-for graph (variant of resource allocation graph), which removes all resource nodes and collapses appropriate edges
 - Nodes are processes
 - $P_i \rightarrow P_j$ if P_i is waiting for P_j
 - Periodically invoke an algorithm that searches for a cycle in the graph.
 - If there is a cycle, there exists a deadlock.
 - An algorithm to detect a cycle in a graph requires an order of $O(n^2)$ operations, where n is the number of vertices in the graph

- Resource Allocation Graph/Wait-for Graph



- Several Instances of a Resource Type
 - wait-for graphs do NOT work for multiple instances
 - Available
 - A vector of length m indicates the number of available resources of each type.
 - Allocation
 - An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
 - Request
 - An $n \times m$ matrix indicates the current request of each process.
 - If Request $[i][j] = k$, process P_i is requesting k more instances of resource type R_j
- Detection Algorithm
 - How to detect if a request can end up in a deadlocked state?
 - 1. Let Work and Finish be vectors of length m and n , respectively Initialize:
 - (a) Work = Available
 - (b) For $i = 1, 2, \dots, n$, if Allocation $[i] \neq 0$,
then $\text{Finish}[i] = \text{false}$;
otherwise, $\text{Finish}[i] = \text{true}$
 - 2. Find an index i such that both:
 - (a) $\text{Finish}[i] = \text{false}$
 - (b) $\text{Request}[i] \neq \text{Work}[i]$
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 $i, 1 \leq i \leq n$,
the system is in deadlock state
Moreover, if $\text{Finish}[i] = \text{false}$, P_i is deadlocked

- This algorithm requires an order of $O(m \times n^2)$ operations to detect, whether the system is in a deadlocked state
- Example of Detection Algorithm

5 processes P_0 through P_4

– 2 resource types

– A (7 instances), B (2 instances), and C (6 instances)

Snapshot at time T_0 :

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in

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

P_2 requests an additional instance of type C

Request

A B C

P_0	0 0 0
P_1	2 0 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

State of the system?

– Can reclaim resources held by process P_0 , but insufficient resources to fulfill other processes' requests

– Deadlock exists, consisting of processes P_1, P_2, P_3 , and P_4

- Detection Algorithm Usage

- When and how often to invoke depends on:

- How often a deadlock is likely to occur?
- How many processes will need to be rolled back?
 - One for each disjoint cycle

- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so the algorithm would not be able to tell which of the many deadlocked processes "caused" the deadlock!

- Recovery: Process Termination

- Abort all deadlocked processes

- Partial computation of many processes may be lost

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

- In which order should the abort start?

- Priority of the process
- How long process has computed and how much longer is has to completion?
- Resources the process has used
- Resources process needs to complete
- How many processes will be needed to terminate?
- Is process interactive or batch?

- Recovery: Resource Preemption

- Take away resources from other processes until the deadlock is resolved

- Issues to be addressed

- Selecting a victim and minimize cost
 - E.g., number of held resources, running time, estimated time to completion
- Rollback: Return to some safe state, restart process for that state
 - Preempted process must be restartable from earlier checkpoint
 - Total restart, if no prior "safe state" identifiable
- Starvation
 - Same process may always be picked as victim
 - E.g., include number of rollback in cost factor

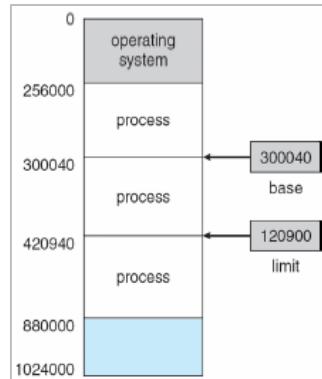
- Ending Statement

- Some predictions became true ... but already turned again.

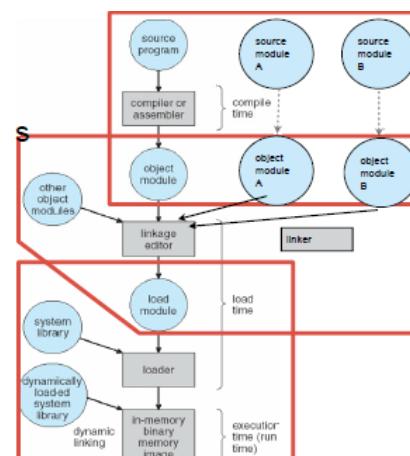
- "ISDN will be ubiquitous by 1985."

V7 Chapter 8: Memory Management

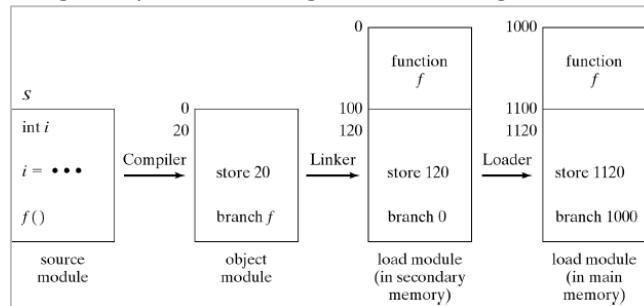
- Memory Management
 - Objectives
 - To provide a description of various ways of organizing memory hardware
 - To discuss memory management techniques, including paging and segmentation topics
 - Topics
 - Background
 - Contiguous Memory Allocation
 - Paging
 - Structure of the Page Table
 - Segmentation
 - Swapping
- Background
 - Program must be brought (from disk) into memory and placed within a process for it to be run
 - Main memory and registers are the only storage a CPU can access directly
 - Register access in one CPU clock (or less)
 - Main memory access can take many cycles
 - Cache sits between main memory and CPU registers
 - Simply to speed up memory-to-CPU-registers time
 - Protection of memory required to ensure correct operation
- Base and Limit Registers
 - A pair of base and limit registers defines the logical address space
 - Simple setup to map one process into memory



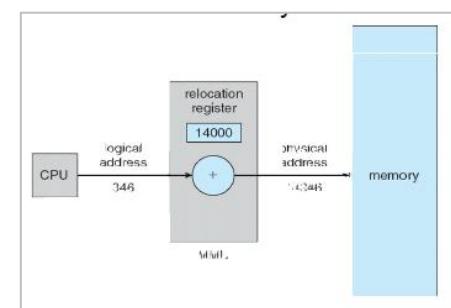
- Multistep Processing of a User Program
 - Translation
 - Source written in symbolic programming language (Assembler)
 - Conversion of symbolic instructions into machine specific form
 - Linking
 - Complex systems made up by multiple modules
 - Resolving of external references to other modules
 - Conversion from logical addresses to physical memory locations
 - Loading
 - Transfer of code from disk to main memory
 - Resolving of external references to system libraries



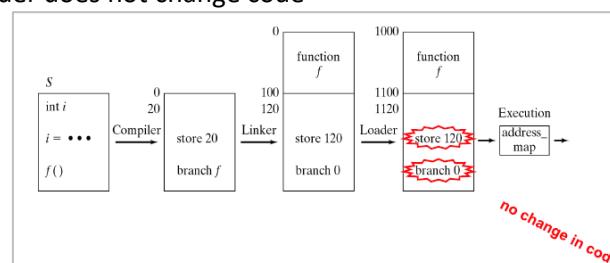
- Binding of Instructions & Data to Memory
 - Address binding of instructions and data to memory addresses can happen at three different stages
 - Compile time: If memory location known a priori, absolute code can be generated; must recompile code if starting loc. changes
 - Compiler and linker responsible for final static mapping of module into memory
 - Load time: Must generate relocatable code if memory location is not known at compile time
 - Linker and loader responsible to resolve relative addressing
 - Execution time: Binding delayed until run time if the process can be moved during its execution from one memory segment to another.
 - Needs hardware support for address maps (e.g., base/limit regs.)
- Logical vs. Physical Address Space
 - The concept of a logical address space that is bound to a separate physical address space is central to proper memory management
 - Logical address generated by the CPU; also referred to as virtual address
 - Physical address address seen by the memory unit
 - Logical and physical addresses are the same in compile-time, link-time, and load-time address-binding schemes
 - Logical (virtual) and physical addresses differ in execution-time address-binding scheme
- Static Address Binding
 - At programming, compilation, linking, and/or loading time



- Dynamic Relocation using a Relocation Register
 - Use of Memory Management Unit (MMU)
 - Hardware device that maps virtual to physical address
 - Relocation Register (RR) is added to every address generated by a user process at the time it is sent to memory
 - The user program deals with logical addresses; it never sees the real physical addresses

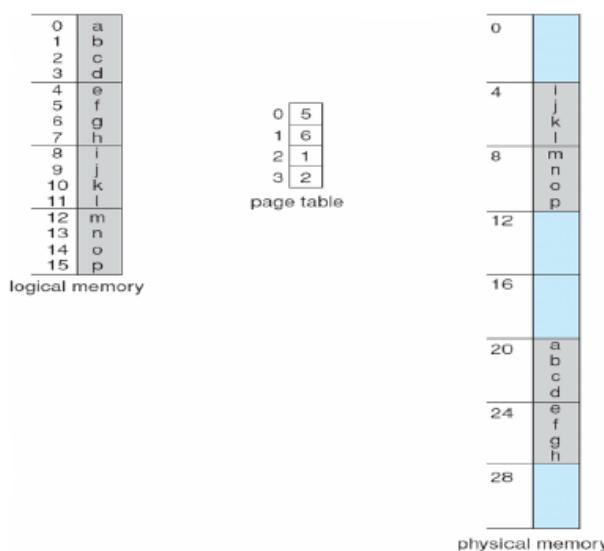


- Dynamic Address Binding
 - At execution time
 - Loader does not change code



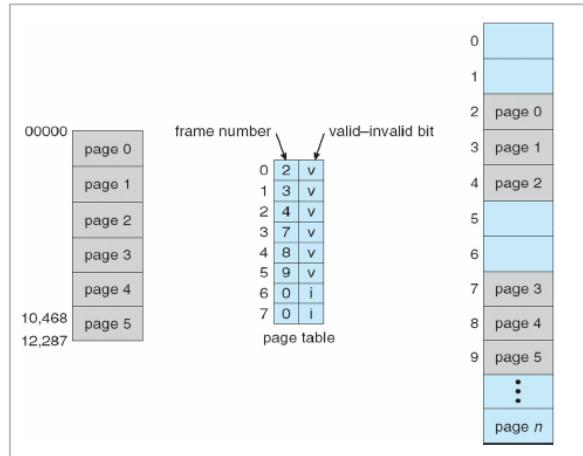
- Dynamic Loading
 - Routine is not loaded until it is called
 - Better memory space utilization; unused routine is never loaded
 - Useful when large amounts of code are needed to handle infrequently occurring cases
 - Dynamic relocatable code
 - Code with fixed starting address is not dynamically relocatable even if bound at load-time
 - Dynamically relocatable if:
 - Relocatable operands and variables are marked
 - Relative memory address given by relocation constant (offset) and relocation register (base address of module)
- Dynamic Linking
 - Linking postponed until execution time
 - Small piece of code, stub, used to locate the appropriate memory-resident library routine
 - Stub replaces itself with the address of the routine and executes the routine
 - Operating system needed to check, if routine is in processes' memory address
 - Dynamic linking is particularly useful for libraries
 - Also known as shared libraries
 - All processes execute the same library code
- Main Memory Management
 - All available free as well as occupied main memory must be managed dynamically at run-time
 - Processes' memory requirements change over time
 - Divide into free (holes) and allocated memory (regions)
 - Partitioning strategies
 - Fixed partitions with different fixed sized memory regions are very restricted
 - Single-program system with 2 partitions for OS & user prog.
 - Prog. size limited to largest partition, internal fragmentation
 - Variable partitioning allows allocation of free space on request
 - Manage variable sized memory regions
 - External or internal fragmentation depending on allocation strategy and implementation
- Contiguous Allocation
 - Main memory usually partitioned into two partitions:
 - Resident operating system, usually held in low memory address range, together with the interrupt vector
 - User processes held in higher memory address ranges
 - Contiguous physical address range for each process
 - Relocation registers used to protect user processes from each other, and from changing operating-system code and data
 - Base register: value of smallest physical address
 - Limit register: range of logical addresses each logical address must be less than the limit register
 - MMU maps logical address dynamically
- Dynamic Storage-Allocation Problem
 - Problem: Given a request for n bytes, find free hole of size $\geq n$
 - Generally more than one available
 - Constraints: Constraints:
 - Maximize memory utilization (minimize external fragmentation)
 - Limit the creation of too small holes
 - Minimize search time

- Allocation Strategies
 - How to satisfy a request of size n from a list of free holes?
 - First-fit: Allocate the first hole that is big enough
 - Best-fit: Allocate the smallest hole that is big enough; must search entire list, unless ordered by size
 - Produces the smallest leftover hole
 - Worst-fit: Allocate the largest hole; must also search entire list
 - Produces the largest leftover hole
 - First-fit and best-fit better than worst-fit in terms of speed and storage utilization
 - Memory will be divided into alternating sections, allocated partitions and holes
- Fragmentation
 - External Fragmentation total memory space exists to satisfy a request, but it is not contiguous
 - Caused by any of the first-fit, best-fit or worst-fit strategies
 - Concentration of holes in one area of memory
 - Excessive number of small holes
 - Reduced availability of large partitions
 - Internal Fragmentation allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
 - Reduce external fragmentation by compaction
 - Shuffle memory contents to place all free memory in one large block
 - Compaction is possible only if code relocation is dynamic (execution time)
 - I/O problems: Latch job in memory while it is involved in I/O, do I/O only into OS buffers
- Paging
 - Logical address space of a process can be noncontiguous in memory; process is allocated physical memory whenever the latter is available
 - Divide physical memory into fixed-sized blocks called frames (size is power of 2, between 512 bytes and 8,192 bytes)
 - Divide logical memory into blocks of same size called pages
 - Keep track of all free frames
 - To run a program of size n pages, need to find n free frames and load program
 - Set up a page table to translate logical addresses (pages) to physical addresses (frames)
 - Causes some internal fragmentation, avoids external fragmentation
- Paging Example

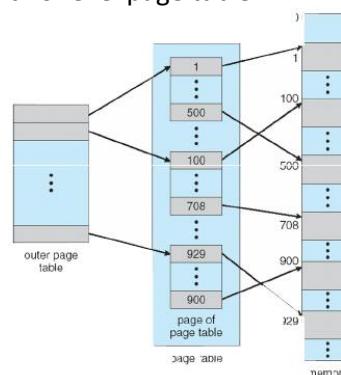


32 byte memory and 4 byte pages

- Implementation of Page Table
 - Page table is kept in main memory
 - One page table for each process, which can be very large
 - Page-table base register (PTBR) points to the page table
 - Page-table length register (PRLR) indicates size of the page table
 - In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
 - The two memory access problems can be solved by the use of a special fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs)
 - Some TLBs store address-space identifiers (ASIDs) in each TLB entry uniquely identifies each process
 - Provide address-space protection
- Memory Protection
 - Memory protection implemented by associating protection bit with each frame
 - Valid-invalid bit attached to each entry in the page table:
 - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
 - “invalid” indicates that the page is not in the process’ logical address space
- Valid (v) or Invalid (i) Bit in a Page Table



- Structure of the Page Table
 - Hierarchical Paging
(Detailed within the next slides)
 - Hashed Hashed Page Page Tables Tables
 - Inverted Page Tables
- Hierarchical Two-Level Page Tables
 - Break up the logical address space into multiple page tables
 - A simple technique is a two-level page table



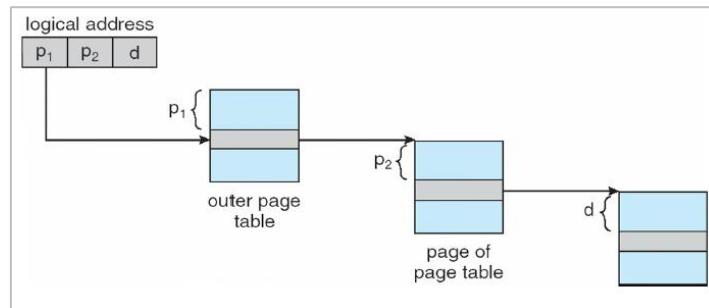
- Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
 - A page number consisting of 22 bits
 - A page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
 - A 12-bit page number
 - A 10-bit page offset
- Thus, the logical address page number page offset is as follows:

page number	page offset	
p_1	p_2	d
12	10	10

- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the outer page table

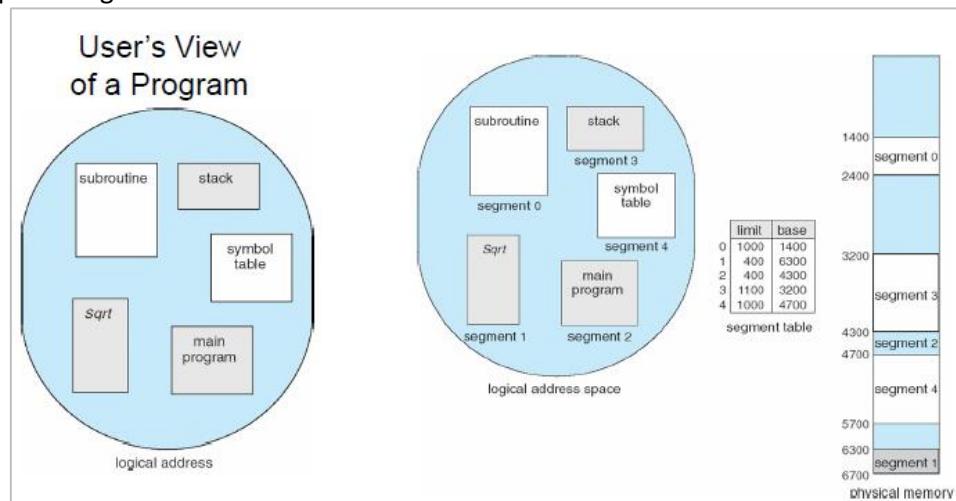
- Address-Translation Scheme



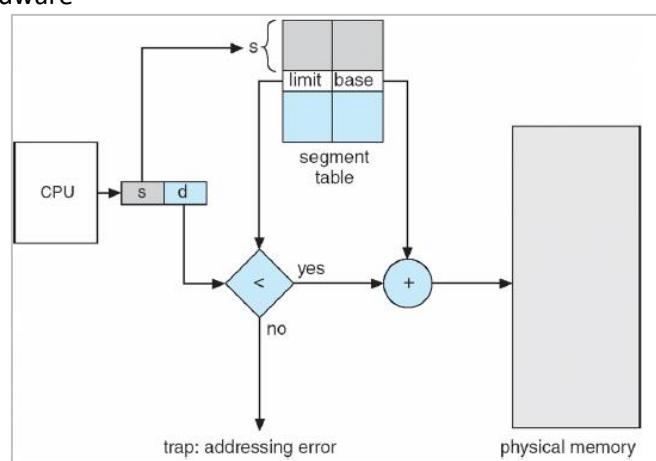
- Segmentation

- Memory management scheme that supports logical structured view of memory
- A program is a collection of segments
 - A segment is a logical unit such as
 - Main program
 - Procedure
 - Function
 - Method
 - Object
 - Local variables, global variables
 - Common block
 - Stack
 - Symbol table
 - Arrays

- Example of Segmentation



- Segmentation Architecture (1)
 - Logical address consists of a two-tuple: [segment-number, offset]
 - Segment table maps segments to physical addresses; each table entry has:
 - Base contains the starting physical address where the segments reside in memory
 - Limit specifies the length of the segment
 - Segment-table base register (STBR) points to the segment table's location in memory
 - Segment-table length register (STLR) indicates number of segments used by a program
 - Segment number s is legal if $s < \text{STLR}$
 - Protection
 - With each entry in segment table associate:
 - validation bit = 0 illegal segment
 - read/write/execute privileges
 - Protection bits associated with segments; code sharing occurs at segment level
 - Since segments vary in length, memory allocation is again a dynamic storage-allocation problem
 - Segmentation can be combined with paging
 - And use of TLBs and hash tables for performance optimization
- Segmentation Hardware

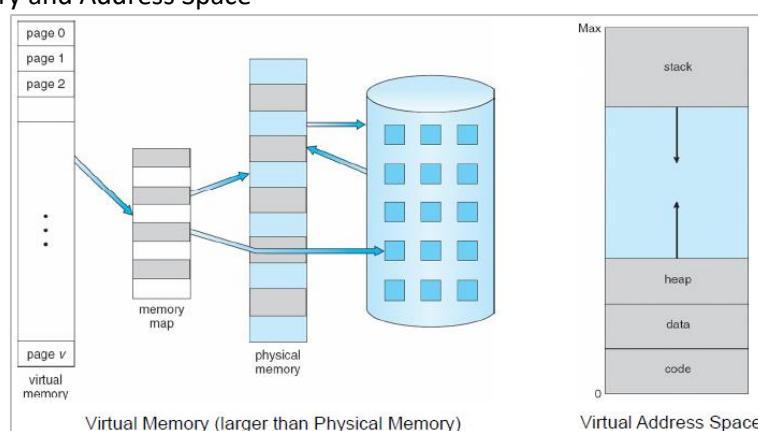


- Insufficient Memory
 - Swapping
 - Temporarily move process and its memory to disk
 - Requires dynamic relocation
 - Memory compaction
 - Reduce external fragmentation by moving holes
 - How much and what to move?
 - Overlays
 - Allow programs larger than physical memory
 - Programs loaded as needed according to calling structure
- Swapping
 - A process can be swapped temporarily out of memory to external storage and brought back into main memory for cont'd execution
 - External storage (Backing Store) fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
 - Roll out, roll in swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed

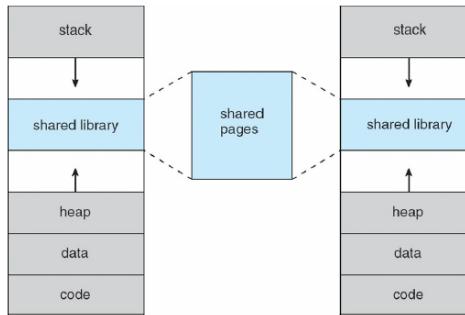
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
 - 100 MB user process may need a second or more to swap-in before being ready to execute on the CPU
- Variants of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
 - System maintains a ready queue of ready-to-run processes, which have memory images on disk
- Ending Statement
 - Some predictions became true ...
 - "I'll bet we'll be manufacturing cores in 1980." H. Lamire, VP Manufacturing of Digital, 1975

V8 Chapter 9: Virtual Memory

- Objectives
 - To describe the **benefits** of a virtual memory system
 - To explain the **concepts** of demand paging, page-replacement algorithms, and allocation of page frames
 - To discuss the **principles** of the working-set model
- Topics
 - Background
 - Demand Paging
 - Process Creation: Copy-on-Write and Memory-mapped Files
 - Page Replacement
 - Allocation of Frames
 - Thrashing
 - Memory-Mapped Files
 - Other Considerations
- Background
 - Virtual memory
 - Separation of user logical memory from physical memory
 - Only part of the program needs to be in memory for execution
 - Logical address space can therefore be much larger than physical address space
 - Allows address spaces to be shared by several processes
 - Allows for more efficient process creation
 - Virtual memory can be implemented via
 - Demand paging
 - Demand segmentation
- Virtual Memory and Address Space



- Shared Library Using Virtual Memory



- Demand Paging

- Bring a page into memory only when it is needed
 - Less I/O needed
 - Less memory needed
 - Faster response
 - More users
- Page is needed \Rightarrow reference to it
 - Invalid reference \Rightarrow abort
 - Not-in-memory \Rightarrow bring to memory
- **Lazy swapper:** never swaps a page into memory unless page will be needed
 - Swapper that deals with pages is a pager

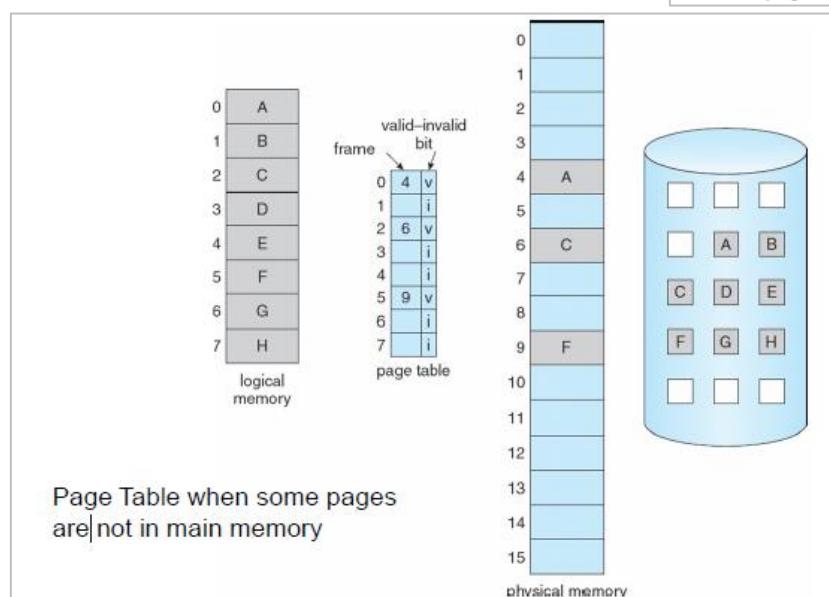
- Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated
 - v \Rightarrow in-memory
 - i \Rightarrow not-in-memory
- Initially valid–invalid valid–invalid bit is set to i on all entries
- Example of a page table snapshot: During address translation, if valid–invalid bit in page table entry is i \Rightarrow page fault

- Pages Missing in Main Memory

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

page table

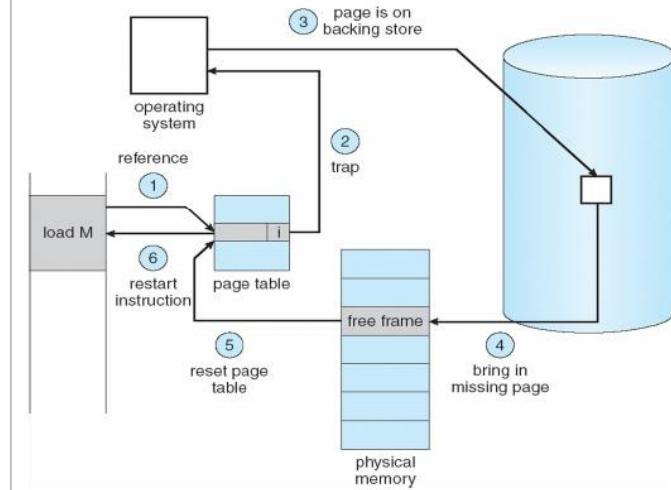


- Page Fault

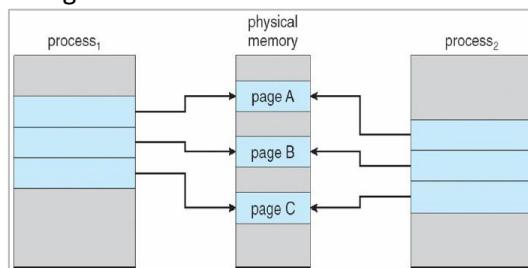
- If there is a reference to a page, first reference to that page will trap to operating system:

- Page fault
- 1. Operating system looks at another table to decide:
 - Invalid reference ⇒ abort
 - Just not in memory
- 2. Get empty frame
- 3. Swap page into frame
- 4. Reset tables
- 5. Set validation bit = v
- 6. Restart the instruction that caused the page fault

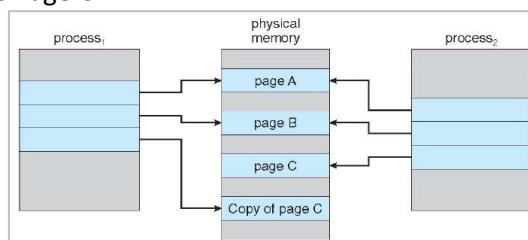
- Steps in Handling a Page Fault



- Process Creation
 - Virtual memory allows for other benefits during the process creation
 - Copy-on-Write
 - Memory-Mapped Files
- Copy-on-Write
 - Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory
 - If either process modifies a shared page, only then is the page copied
 - COW allows more efficient process creation as only modified pages are copied
 - Free pages are allocated from a **pool** of zeroed-out pages
- Before Process 1 Modifies Page C

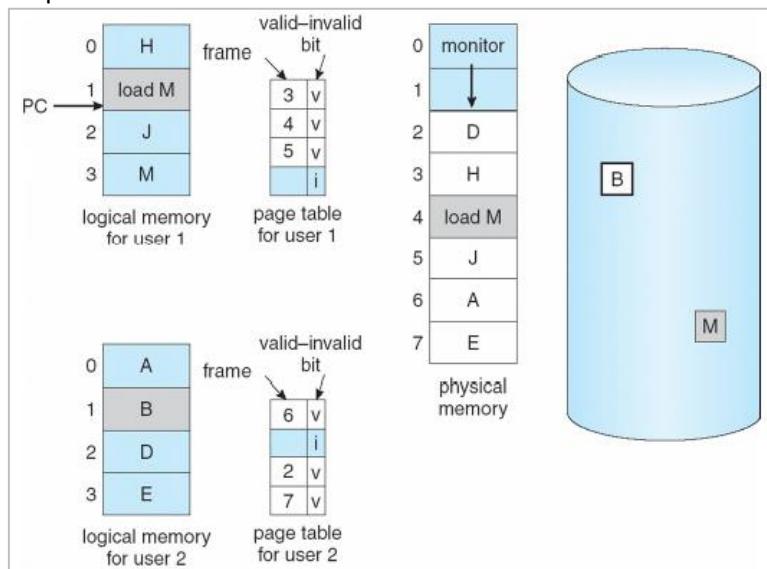


- After Process 1 Modifies Page C

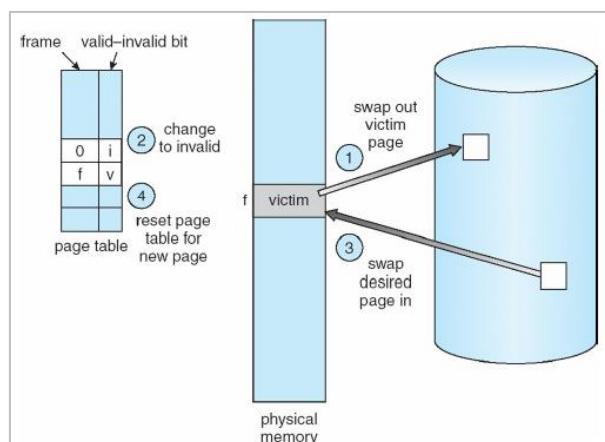


- Lack of Free Frames
 - Page replacement

- Find some page in memory, but not really in use, swap it out
 - Algorithm
 - Performance demands
 - In search of an algorithm which will result in a minimum number of page faults
 - Same page may be brought into memory several times
- Page Replacement
 - Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
 - Use modify (dirty) bit to reduce overhead of page transfers only modified pages are written to disk
 - Page replacement completes separation between logical memory and physical memory large virtual memory can be provided on a smaller physical memory
- Need for Page Replacement

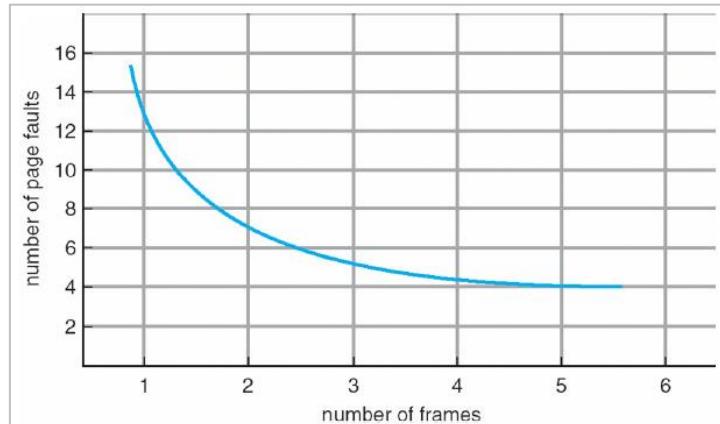


- Basic Page Replacement
- 1. Find the location of the desired page on disk
- 2. Find a free frame
 - If there is a free frame, use it
 - If there is no free frame, use a page replacement algorithm to select a victim frame
- 3. Bring the desired page into the (newly) free frame; update the page and frame tables
- 4. Restart the process
- Page Replacement

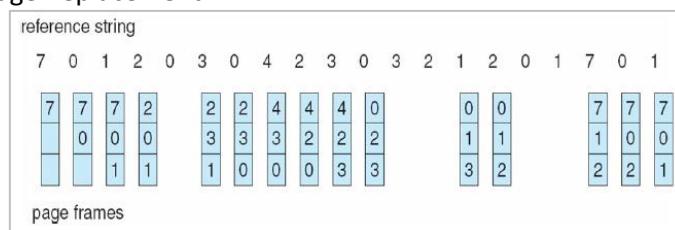


- Page Replacement Algorithms
 - Want lowest page-fault rate

- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
- In all of the examples the reference string is:
- 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- Page Faults vs. Number of Frames



- First-in-First-Out (FIFO) Algorithm
 - Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
 - 3 frames (3 pages can be in memory at a time per process):
 - 4 frames:
 - Belady's Anomaly: more frames \Rightarrow more page faults
- FIFO Page Replacement

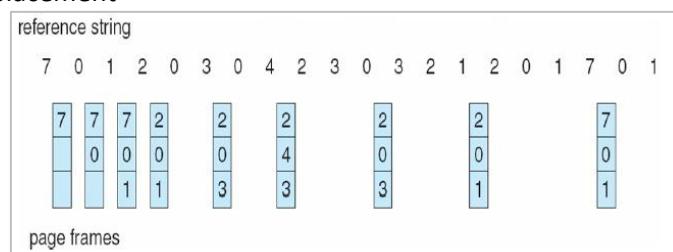


1	1	4	5
2	2	1	3
3	3	2	4
1	1	5	4
2	2	1	5
3	3	2	
4	4	3	

9 page faults

10 page faults

- Optimal Algorithm
 - Replace page that will not be used for longest period of time
 - 4 frames example:
 - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
 - How do you know this?
 - Used for measuring how well your algorithm performs
- Optimal Page Replacement

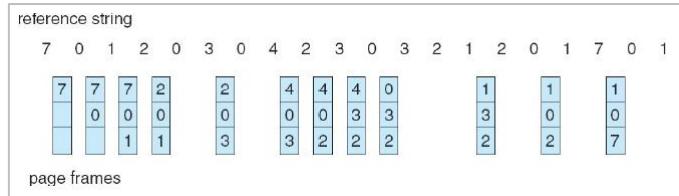


1	4
2	
3	
4	5

6 page faults

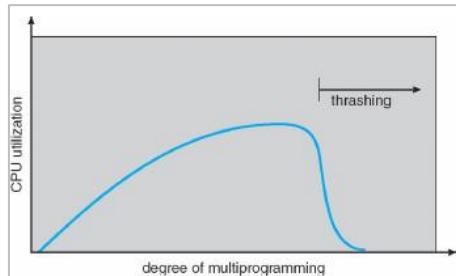
- Least Recently Used (LRU) Algorithm
 - Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
 - Counter implementation
 - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
 - When a page needs to be changed, look at the counters to determine which are to change
- LRU Page Replacement

1	1	1	1	5
2	2	2	2	2
3	5	5	4	4
4	4	3	3	3



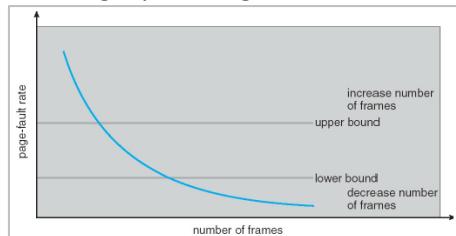
- Counting Algorithms
 - Keep a counter of the number of references that have been made to each page
 - LFU Algorithm
 - Replaces page with smallest count
 - MFU Algorithm
 - Based on the argument that the page with the smallest count was probably just brought in and has yet to be used
 - Allocation of Frames
 - Each process needs minimum number of pages
 - Example: IBM 370
 - 6 pages to handle SS MOVE instruction:
 - Instruction is 6 byte, might span 2 pages
 - 2 pages to handle from
 - 2 pages to handle to
 - Two major allocation schemes
 - Fixed allocation
 - Priority allocation
 - Fixed Allocation
 - Equal allocation
 - E.g., if there are 100 frames and 5 processes, give each process 20 frames
 - Proportional allocation
 - Allocate according to the size of process
- $s_i = \text{size of process } p_i$
 $S = \sum s_i$
 $m = \text{total number of frames}$
 $a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$
- $m = 64$
 $s_1 = 10$
 $s_2 = 127$
 $a_1 = \frac{10}{137} \times 64 \approx 5$
 $a_2 = \frac{127}{137} \times 64 \approx 59$
- Priority Allocation
 - Use a proportional allocation scheme using priorities rather than size
 - If process P I generates a page fault
 - Select for replacement one of its frames
 - Select for replacement a frame from a process with lower priority number
 - Global vs. Local Allocation
 - Global replacement
 - Process selects a replacement frame from the set of all frames; one process can take a frame from another
 - Local replacement
 - Each process selects from only its own set of allocated frames
 - Thrashing
 - If a process does not have “enough” pages, the page-fault rate is very high
 - This leads to

- Low CPU utilization
- Operating system thinks that it needs to increase the degree of multiprogramming
- Another process added to the system
- Thrashing \equiv a process is busy swapping pages in and out



- Page-Fault Frequency Scheme

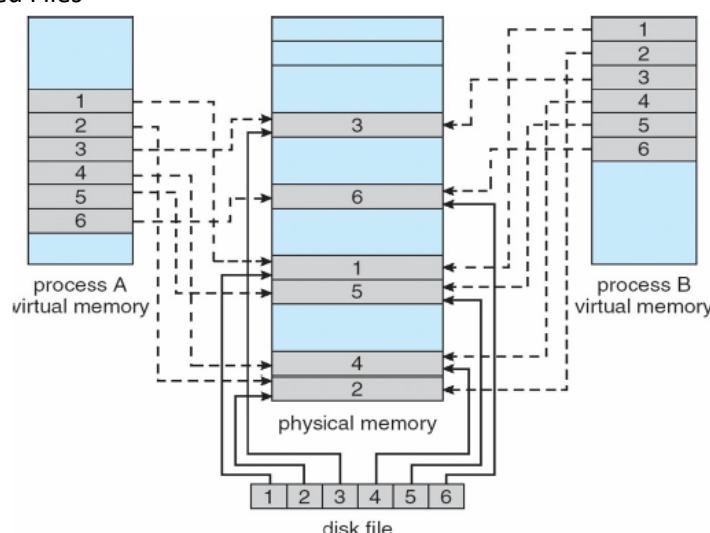
- Establish "acceptable" page-fault rate
 - If actual rate too low, process loses frame
 - If actual rate too high, process gains frame



- Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory
- A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
- Simplifies file access by treating file I/O through memory rather than read()/write() system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared

- Memory Mapped Files



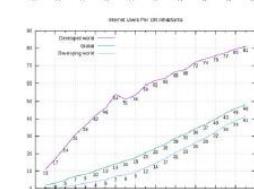
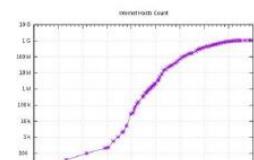
- Other Issues

- Prepaging
 - To reduce the large number of page faults that occurs at

- process startup Prepage all or some of the pages a process will need, before they are referenced But if prepaged pages are unused, I/O and memory was wasted Assume s pages are prepaged and α of the pages is used
 - Is cost of $s * \alpha$ save pages faults > or < than the cost of prepaging $s * (1 - \alpha)$ unnecessary pages?
 - α near zero \Rightarrow prepaging loses
- Page size selection must take into consideration:
 - fragmentation
 - table size
 - I/O overhead
 - Locality
- I/O Interlock
 - Pages must sometimes be locked into memory
 - Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm

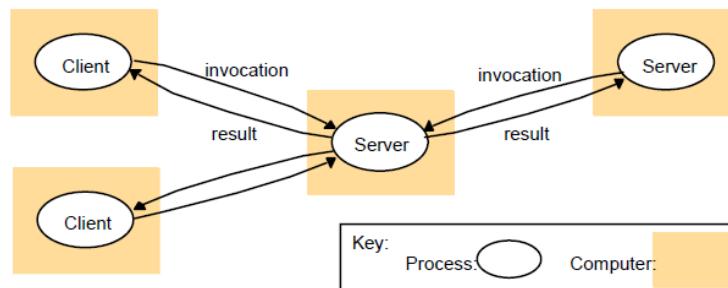
V9 Chapter 10: Distributed Systems

- Objectives
 - To develop an understanding of what a distributed system is
 - To present the key challenges in distributed systems
 - To overview main concepts and approaches
- Topics
 - Evolution of communication networks and scale
 - Definition of distributed systems
 - Examples of distributed systems
 - Challenges and examples for those
 - Hardware architectures
 - Software concepts
 - Middleware
 - Network interaction types
- Evolution of Networks
 - In early times, computers were standalone devices
 - In the late 80's/early 90's, computer networks started
 - Internet brought a revolution to the way of life on the planet!
 - Today networking is essential to everyday life
 - Even low budget laptops (One Laptop Per Child project, OLPC, & Lehrplan 21) are designed to be networked
 - Shift from standalone computers, to the paradigm of computers communicating, interacting, and collaborating with each other
 - Challenging to manage Distributed Systems at a large and complex scale
- The Scale
 - Increasing number of computers
 - Increasing number of Internet users
 - Increasing number of distributed applications
 - All needs are increasingly
 - Complex
 - Larger scale
 - Application-specific
 - The real scale is larger

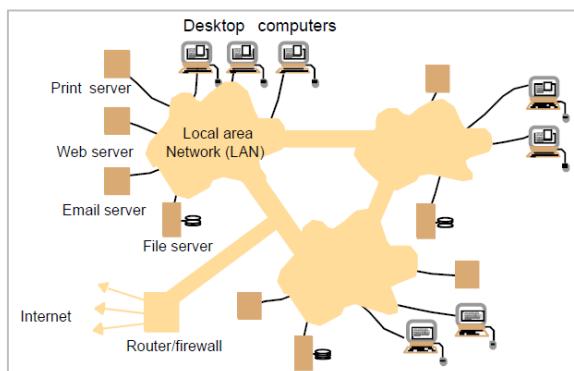


- “The” or “a” Definition
 - Distributed Systems come in many flavors!
 - Computers connected by a network and
 - Spatially separated by any distance
 - Def.: A collection of independent computers that appears to its users as a single coherent system
 - Hardware: All machines are fully autonomous
 - Software: Users think they deal with a single system
 - Consequences
 - Concurrency
 - No global clock
 - Independent failures
- Distributed System Examples
 - Client Server Communications
 - Intranet
 - Internet
 - Automation Network
 - Personal Network
 - Peer-to-Peer Systems
 - Cloud Computing
 - leading to
 - Distributed Systems’ Middleware

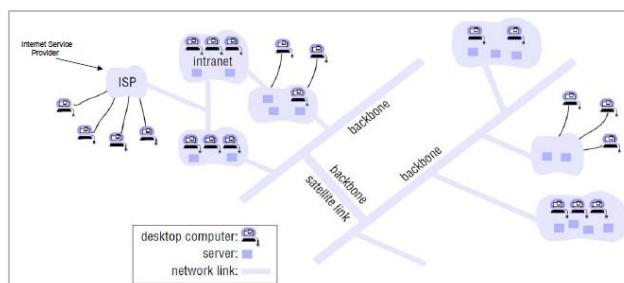
- Client Server Communications



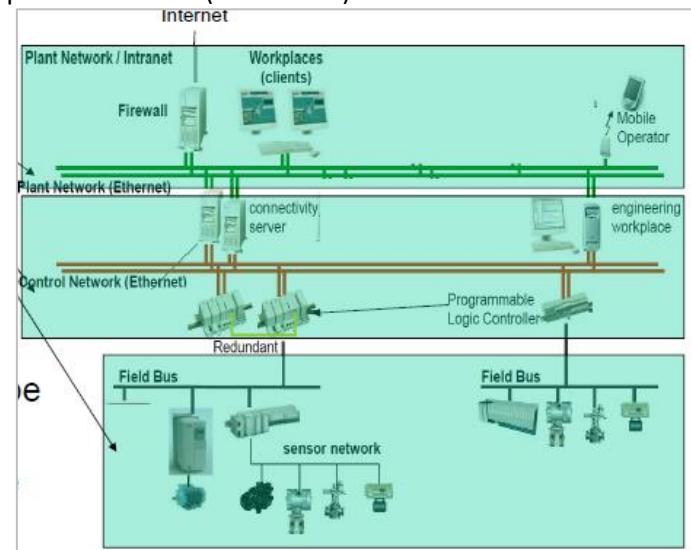
- Intranet



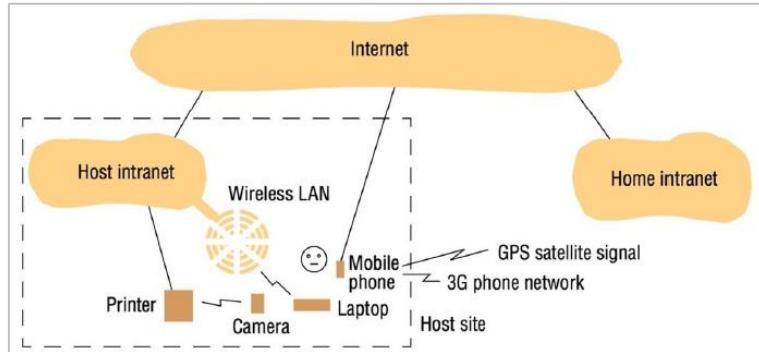
- Internet



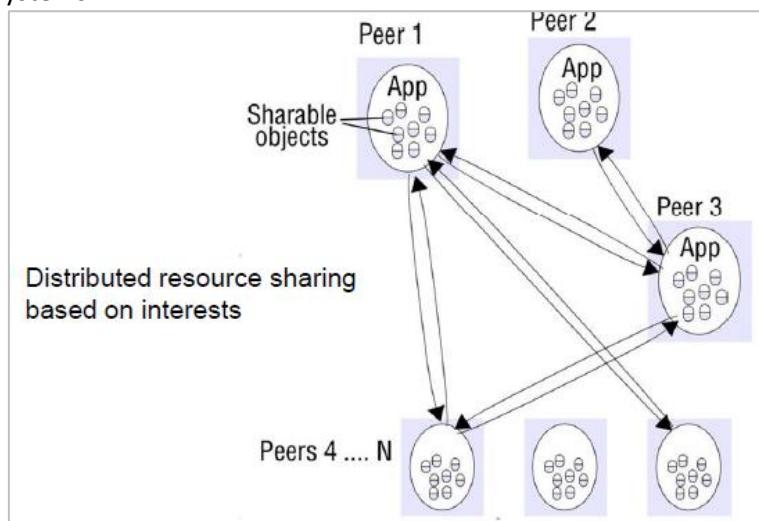
- Automation Network
 - 3 layers
 - Focus on reliability
 - Redundancy
 - Real-time (control) network
 - Messages must be delivered in time
 - Special hardware (controllers)



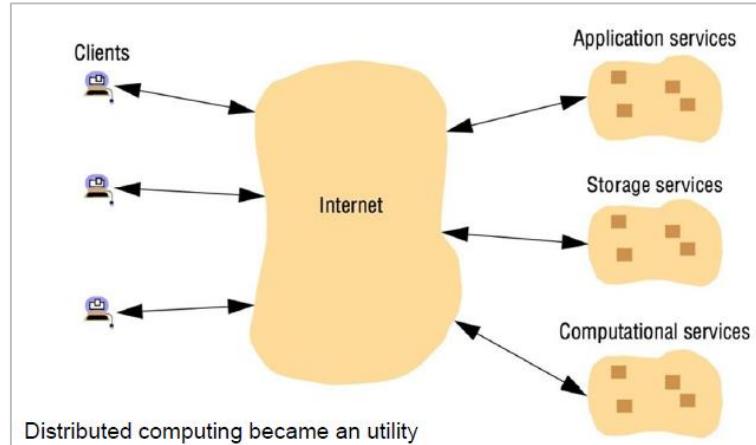
- Personal Network
 - Portable and handheld devices in a distributed system



- Peer-to-Peer Systems

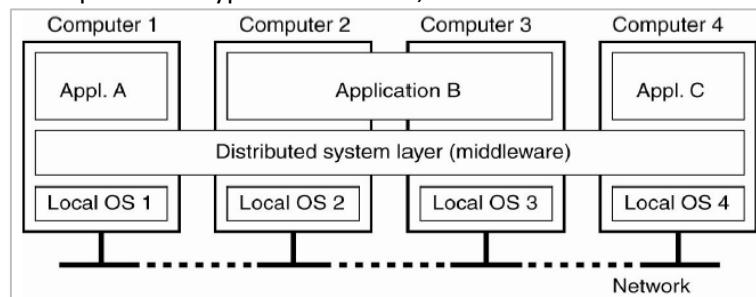


- Cloud Computing



- Distributed System as a Middleware

- Middleware provides similar interfaces to all participants
 - Irrespective of types of OS across multiple machines
 - Irrespective of types of networks, communications used



- Challenges in Distributed Systems

1. Transparency	4. Openness
<ul style="list-style-type: none"> Single view of the system Hide numerous details 	<ul style="list-style-type: none"> Extensibility Publication of interfaces
2. Heterogeneity	5. Scalability
<ul style="list-style-type: none"> Networks Computers (HW) Operating systems Programming languages Developers 	<ul style="list-style-type: none"> Controlling the cost of resources Controlling the performance Preventing resources from running out Avoiding performance bottlenecks
3. Failure Handling	6. Security
<ul style="list-style-type: none"> Detecting Masking Tolerating Recovery Redundancy 	<ul style="list-style-type: none"> Secrecy Authentication Authorization Non-repudiation

- 1. Transparency**

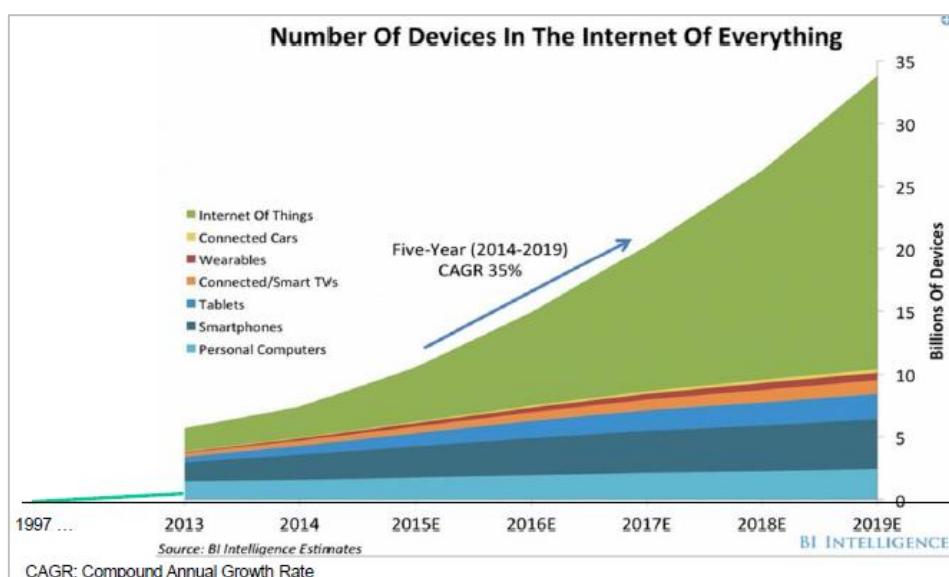
- Recall of distributed system definition: A collection of independent computers that appears to its users as a single coherent system
- Thus, transparency means
 - A set of computers appears as a single computer to all applications and users
 - Abstractions needed to facilitate application development
 - All details are hidden and dealt with transparently

- Transparency comes in eight different flavors
 - Access transparency
 - Enables local and remote resources to be accessed using identical operations
 - Location transparency
 - Enables resources to be accessed without knowledge of their physical or network location (for example, which building or IP address)
 - Concurrency transparency
 - Enables several processes to operate concurrently using shared resources without interference between them
 - Replication transparency
 - Enables multiple instances of resources to be used to increase reliability and performance without knowledge of the replicas by users or application programmers
 - Failure transparency
 - Enables the concealment of faults, allowing users and application programs to complete their tasks despite the failure of hardware or software components
 - Mobility transparency
 - Allows the movement of resources and clients within a system without affecting the operation of users or programs
 - Performance transparency
 - Allows the system to be reconfigured to improve performance as loads vary
 - Scaling transparency
 - Allows the system and applications to expand in scale without change to the system structure or the application algorithms
- 1. Sample Cases and Problems
 - Access transparency
 - Different data representations, e.g., “100” binary
 - 4 decimals in Big Endian by reading from left to right, e.g., Sun Sparc
 - 1 decimal in Little Endian by reading from right to left, e.g., Intel
 - Different conventions
 - E.g., the Linux file system is case-sensitive, Windows’ not
 - Location/Migration/Relocation transparency
 - Access to Web page without knowing its IP, physical location
 - Transparent handover when changing wireless access points
 - Replication transparency
 - GMail, Facebook, banks maintain their data transparently replicated for increased availability and improved access time
- 2. Heterogeneity
 - Access Transparency is dealing with one part of heterogeneity
 - Hiding away details of technology differences
 - Additionally, transparently addressing the differences in
 - Performance
 - Capabilities
 - Network connectivity
 - For instance, in a Personal Area Network (PAN), transparently connecting desktop, laptop, watch, and mobile phone
 - Avoiding the assignment of intensive tasks to the mobile phone or a watch

- 3. Failure Handling
 - Another definition of a distributed system (L. Lamport)
 - You know you have one, when the crash of a computer, you've never heard of, stops you from getting any work done." 😊
 - Fundamental points in distributed systems
 - Reliability
 - High Availability
 - Distributed systems should be failure-transparent
 - A failure on "some" components should not be fatal (or, ideally even detectable) to the applications.
 - E.g., a failure in a bank server should not prevent you from withdrawing money
- 3. Types of Failures

Class of Failure	Affects	Description
Fail-stop	Process	Process halts and remains halted. Other processes may detect this state.
Crash	Process	Process halts and remains halted. Other processes may not be able to detect this state.
Omission	Channel	A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
Send-omission	Process	A process completes a <i>send</i> , but the message is not put in its outgoing message buffer.
Receive-omission	Process	A message is put in a process's incoming message buffer, but that process does not receive it.
Arbitrary (Byzantine)	Process or channel	Process/channel exhibits arbitrary behavior: it may send/transmit arbitrary messages at arbitrary times, commit omissions; a process may stop or take an incorrect step.

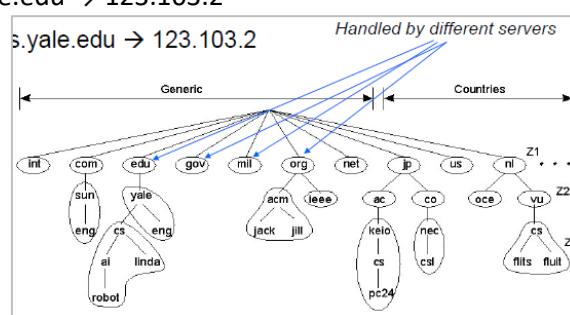
- 4. Openness
 - Well-defined interfaces
 - E.g., Application Programming Interfaces (API)
 - Well designed
 - Clearly described (publicly)
 - Standardized
 - Use of Interface Definition Language (IDL)
 - Boosting interoperability, portability, and extensibility
 - Boosting modularity
 - Upgrading one module should not affect the rest
- 5. Scalability



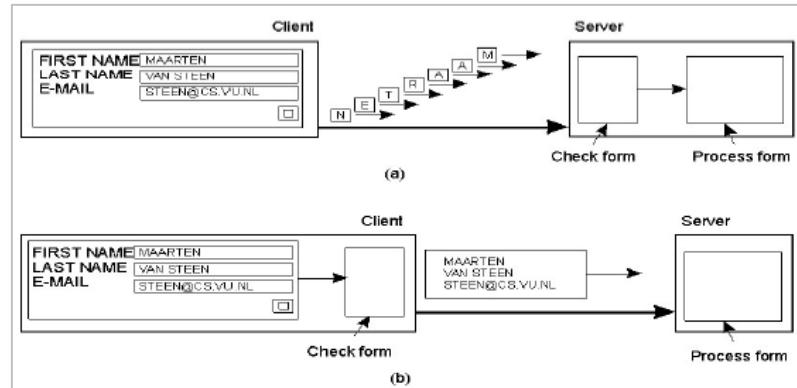
- Scalability comes in different flavors
 - Scalability with respect to size
 - E.g., more computers, more users, or more devices
 - Scalability with respect to geography
 - E.g., applicable at different languages, regions, communities
 - Scalability with respect to administration
 - E.g., more active users, more failures, more connections
 - Scalability with respect to load
 - Add/remove resources depending on load
 - Effects and impacts
 - Most systems suffer from performance loss when scaling
 - 5. Scalability Limitation
 - Problematic dimensions
 - Server performance
 - Network bandwidth
 - Solutions
 - Storing data in a distributed manner might reduce overall communicated data volume.

Centralized services	A single server for all users
Centralized data	A single on-line telephone book
Centralized algorithms	Doing routing based on complete information

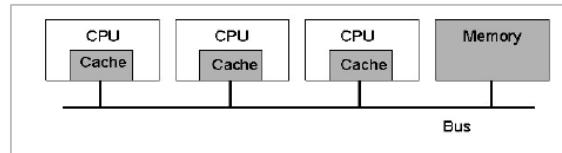
- 5. Techniques to Address Scalability
 - Alternative approaches
 - Distribution of responsibilities
 - E.g., partition computation, split data on multiple computers
 - Hide communication latencies
 - Avoid waiting for responses, do something else
 - Apply replication techniques
 - E.g., copy data onto multiple servers
 - 5.1 Distribution of Responsibilities
 - Example: Domain Name System (DNS)
 - Server name → IP address
 - DNS divided into zones
 - cs.yale.edu → 123.103.2



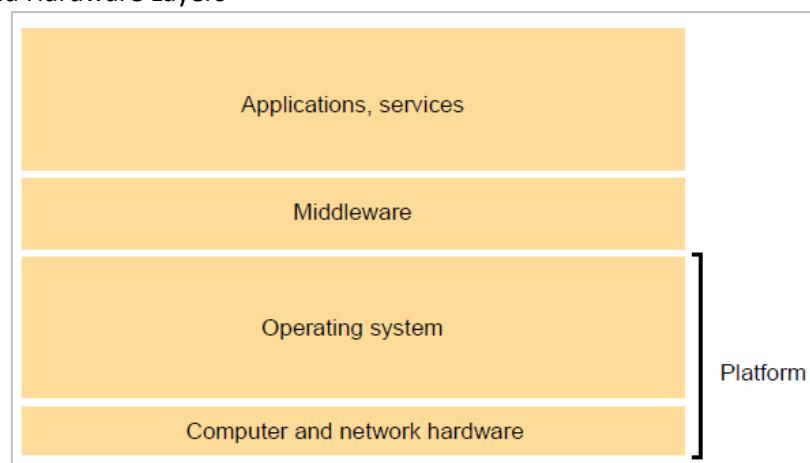
- 5.2 Hiding Communication Latencies
 - The difference between letting
 - a server or
 - a client check forms as they are being filled



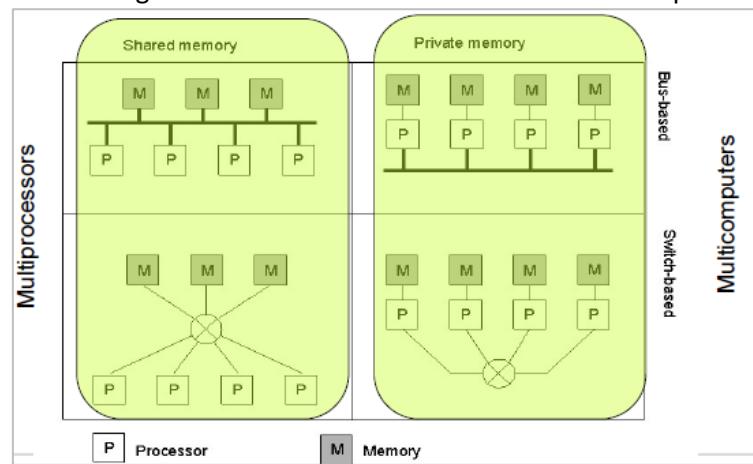
- 5.3 Replication Techniques
 - Caching is a typical example
 - Challenge due to replication
 - Maintaining consistency for updates



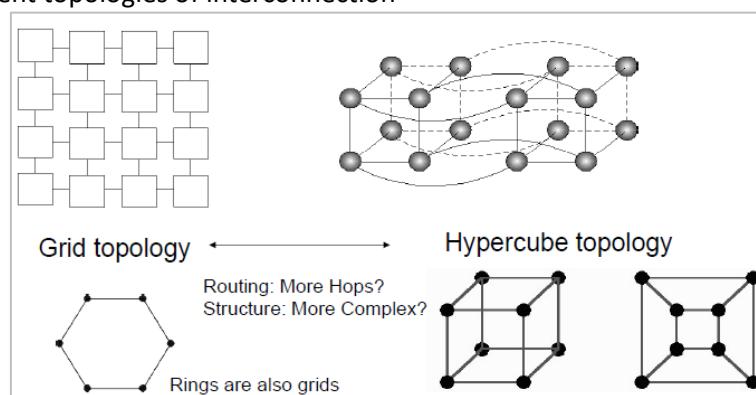
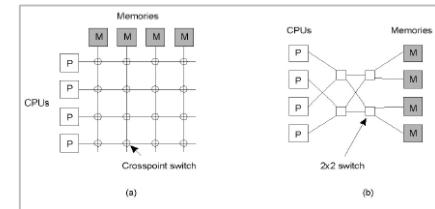
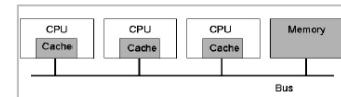
- 6. Security
 - Sensitive information passing through a number of systems, is complex to secure
 - E.g., passwords, credit card data, medical records
 - To secure systems against attacks (incomplete)
 - Viruses, worms, (Distributed) Denial-of-Service (DDoS)
 - In general “cybercrime” cases
 - Countermeasures (in super short form)
 - Authentication, authorization, encryption, non-repudiation
 - Security follows the “weakest link” principle
 - Security measures are defined by a risk assessment
- Software and Hardware Layers



- Hardware Concepts
 - Different basic organizations and memories in distributed computer systems



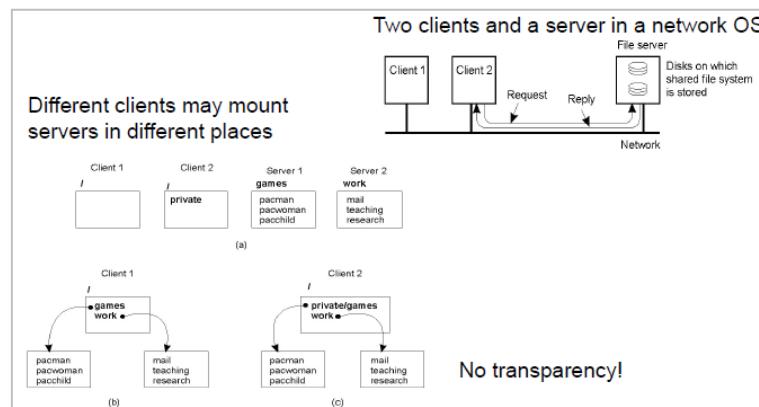
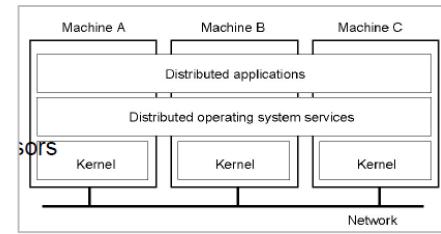
- Bus-based Multiprocessors
 - All processors share the same bus to access memory
 - Advantages
 - Simpler and cheaper construction
 - Drawback
 - Not scalable: with more than a few processors, the bus is saturated
 - Caches introduced to prevent bus saturation
 - Consistency problems
 - Need to invalidate other caches after every write
- Switch-based Multiprocessors
 - Concurrent memory access by multiple processors
 - Although not all combinations useful
 - Advantage
 - Increased concurrency → gives speed
 - Drawback
 - Delay due to many switches, expensive linkage, and fast crosspoint switches
- Switch-based Multicomputers
 - Different topologies of interconnection



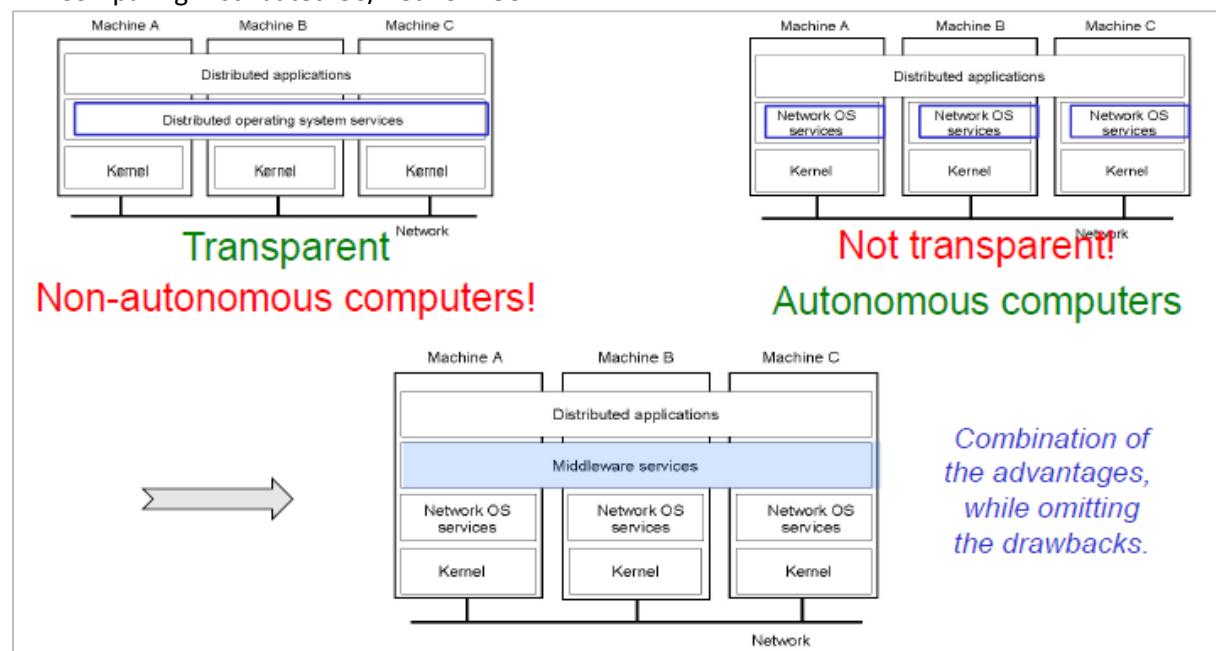
- Software Concepts

System	Description	Main Goal
Distributed Operating System (DOS)	Tightly-coupled operating system for multiprocessors and homogeneous multicomputers	Hide and manage hardware resources
Network OS (NOS)	Loosely-coupled operating system for heterogeneous multicomputers (LAN and WAN)	Offer local services to remote clients
Middleware	Additional layer atop of NOS implementing general-purpose services	Provide distribution transparency

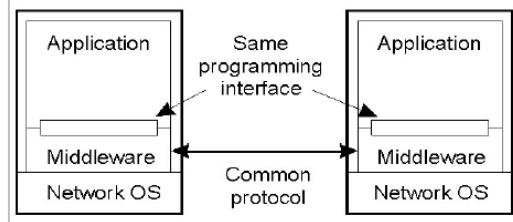
- Distributed Operating System (OS)
 - Tightly coupled and detailed control
 - Transparent task allocation to a processor
 - Transparent memory access (Distributed Shared Memory)
 - Transparent storage
 - Provides complete transparency and a single view of the system
 - Requires multiprocessors or homogeneous multicomputers
- Network OS
 - Loosely coupled and less control
 - Provides services such as
 - rlogin (remote login)
 - ftp (file transfer protocol)
 - scp (secure copy)
 - NFS (network file systems)
 - Not transparent
 - No single view of the system
 - Very flexible with respect to heterogeneity and participation
- Network OS Instances
- Client/Server architecture



- Comparing Distributed OS/Network OS



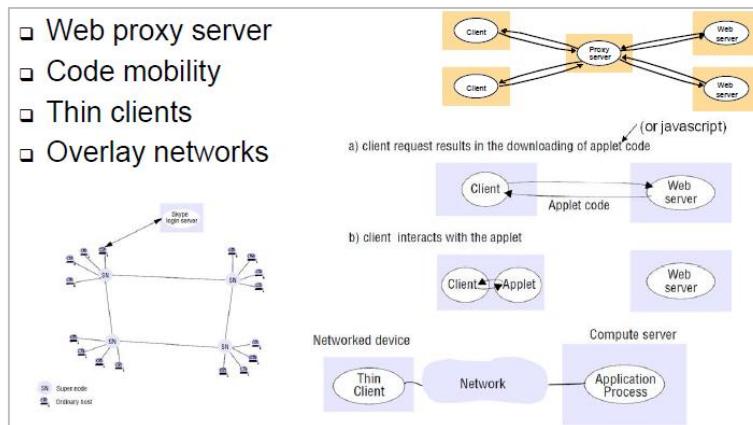
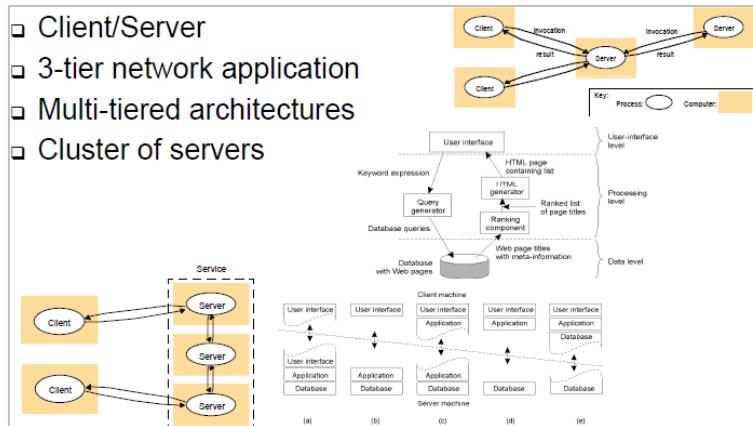
- Middleware
 - In an open system the same protocols are used by each middleware layer and they offer the same interfaces to applications
 - Openness, transparency, (scalability)



- Comparing All Three Alternatives

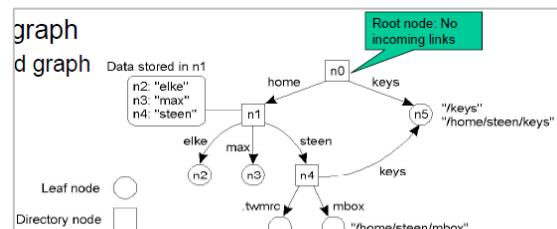
Item	Distributed OS		Network OS	Middleware-based OS
	Multiproc.	Multicomp.		
Degree of transparency	Very High	High	Low	High
Same OS on all nodes	Yes	Yes	No	No
Number of copies of OS	1	N	N	N
Basis for communication	Shared memory	Messages	Files	Model specific
Resource management	Global, central	Global, distributed	Per node	Per node
Scalability	No	Moderately	Yes	Varies
Openness	Closed	Closed	Open	Open

- Network Interaction Types

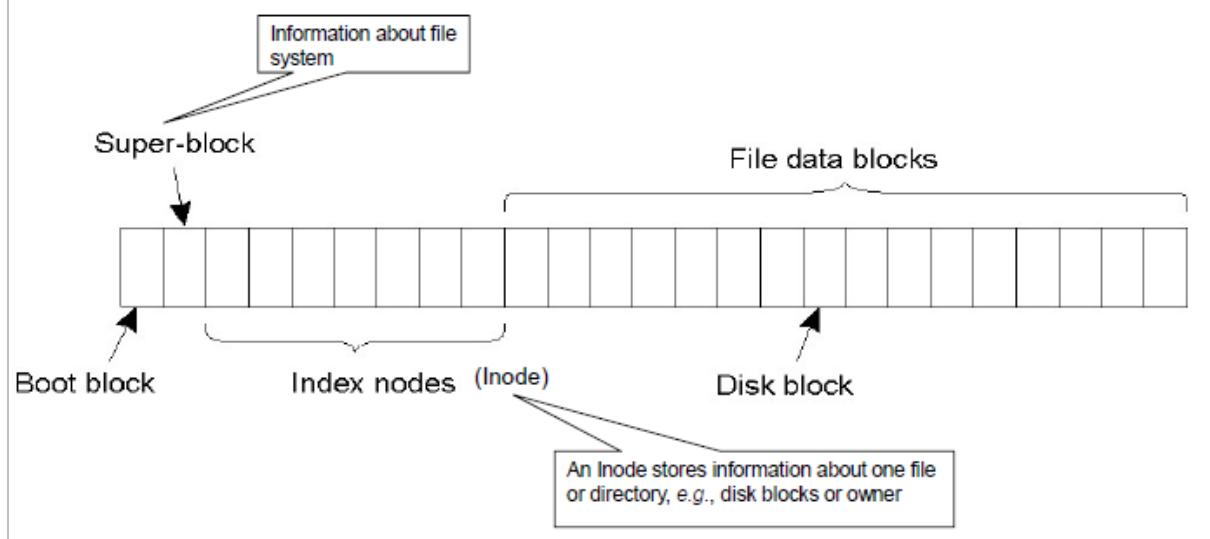


V11 Chapter 11: Naming

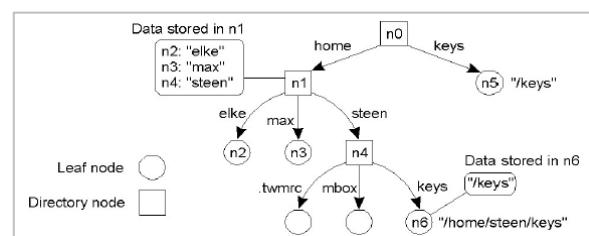
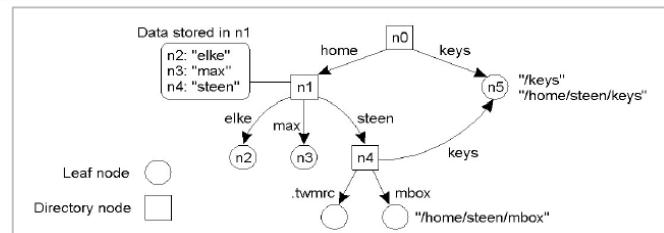
- Objectives
 - To motivate the differences of address and name
 - To develop an understanding of naming types backed by example systems
- Topics
 - Address
 - Names
 - Naming types
 - Hierarchical Naming
 - Flat Naming
 - Attribute-based Naming
- Accessing Entities: Address
 - Need to operate on entities exists for
 - Hosts and users
 - Web pages and services
 - Files (e.g., on an FTP server) ...
 - To operate on an entity, access to it is essential
 - Each entity should have an access point, usually called address
 - Various types of addresses exist, e.g.:
 - Phone number, IP address, port, URL
 - User ID, postal address
- Names
 - If an address is enough to access an entity, what are names needed?
 - Various reasons:
 - Address change of an entity
 - E.g., E.g., due to relocation (with a new IP address)
 - E.g., due to reorganization (a Web server may be moved to a new host)
 - An entity may have more than one access point (replication)
 - Naming provides an abstraction useful for
 - Location independence
 - Relocation of entities
 - Allowing a single reference to a set of alternative access points
 - Offering human-friendly names
- The Three Naming Types
 - Hierarchical Naming
 - Unix File System
 - DNS (Domain Name System)
 - Flat Naming
 - Distributed Hash Tables (DHT)
 - Attribute-based Naming
 - X.500
- Hierarchical Naming: Name Spaces
 - Naming graph
 - Directed graph
 - A leaf node represents a named entity
 - No outgoing links
 - A directory node stores links to other nodes (leafs or other directory nodes)



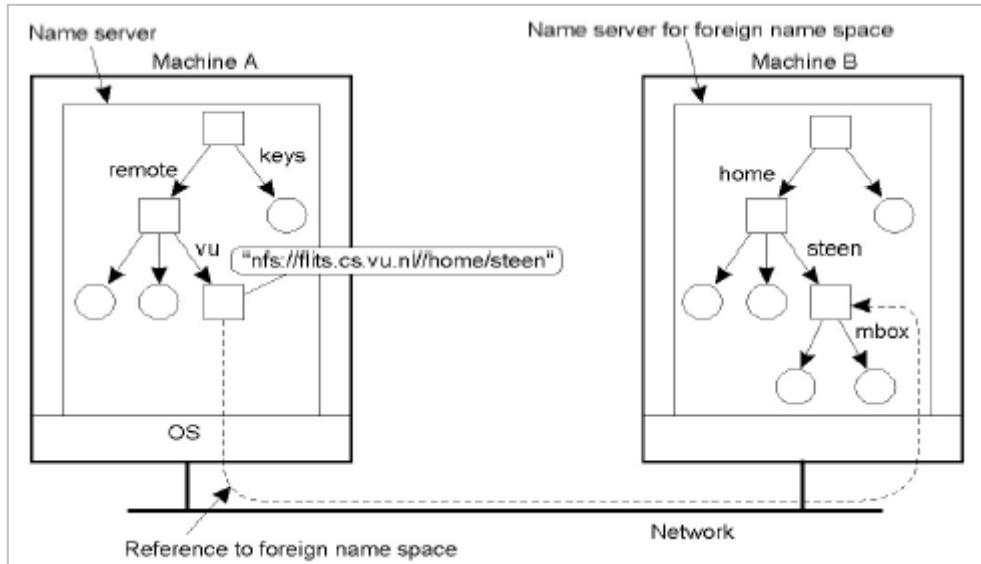
- Hierarchical Naming Concept
 - Based on the concept of “Name Spaces”
 - A directed graph of names
 - Ordered and defined by a fixed alphabet of characters
 - Each name is a path in the naming graph
 - If starting from the root, we call it an absolute name
 - Otherwise, relative name
 - Examples (of absolute names):
 - File system: /home/stiller/teaching/fs18/SSDDS08/exam.pdf
 - DNS: www.csg.uzh.ch
 - URL: http://www.csg.uzh.ch/csg/en/teaching/fs18/ssds.html
- Unix File System
 - Here on a logical disk of contiguous disk blocks



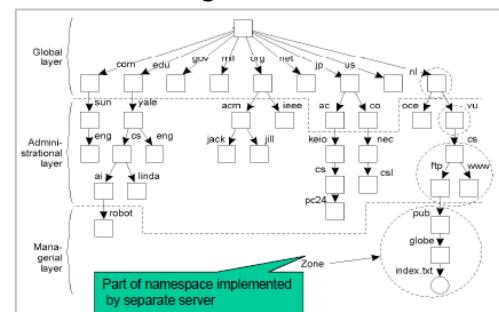
- Multiple Names: Hard Links
 - An entity may have multiple names within a name space
 - Multiple paths that lead to the same leaf node
 - In Unix they are called hard links
 - Pointing to the “Inode”
- Multiple Names: Symbolic Links
 - A special node contains the absolute (or relative) name of another node
 - In Unix this is called “soft link” or “symbolic link”



- Mounting: Remote Linking
 - Mounting
 - Setting a symbolic link referring to a remote name space
 - Performed through a specific process protocol



- Merging Name Spaces
 - Adding a new root and mounting two or more namespaces below it (e.g., DEC Global Name Service)
 - Problem
 - Absolute names of all name-spaces spaces are are changed changed
 - Solution
 - At root node, cache original top-level names
 - E.g., root remembers that home, keys map to /vu, mbox maps to /oxford
 - Only file systems with different top-level directories can be merged
 - Name Space Distribution
 - Ok, so far, for small scale naming
 - E.g., a company-wide file system
 - But what happens when the scale grows to ... global?!
 - E.g., DNS
 - Partitioning of the DNS name space, including Internet-accessible files, into three layers
 - Domain Name Systems (DNS)
 - How to map structured hostnames to IP addresses?
 - Old days: HOSTS.TXT file FTPed among hosts
 - DNS operated as a distributed directory service
 - Hierarchical name space
 - One global root
 - Replicated across 13 root servers
 - All Denial-of-Service (DoS) attack on these root servers unsuccessful
 - Due to caching: queries to root servers are relatively rare
 - DNS is most longstanding and stable global directory service, if not the only one!



- DNS Layers
 - A comparison between name servers for implementing nodes from a large-scale name space partitioned into 3 layers
 - Arrows coming from “large-scale”:

Item	Global	Administrational	Managerial
Geographical scale of network	Worldwide	Organization	Department
Total number of nodes	Few	Many	Vast numbers
Responsiveness to lookups	Seconds	Milliseconds	Immediate
Update propagation	Lazy	Immediate	Immediate
Number of replicas	Many	None or few	None
Is client-side caching applied?	Yes	Yes	Sometimes

- DNS Data Base Entries

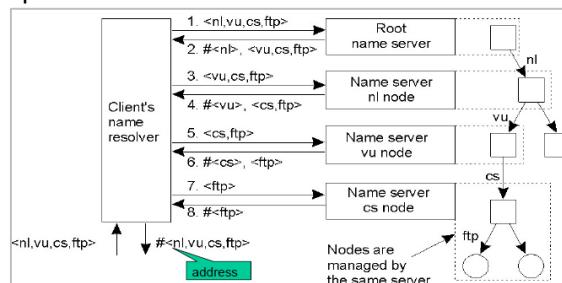
□ For the zone cs.vu.nl

Name	Record type	Record value
cs.vu.nl	SOA	star (1999121502,7200,3600,2419200,86400)
cs.vu.nl	NS	star.cs.vu.nl
cs.vu.nl	NS	top.cs.vu.nl
cs.vu.nl	NS	solo.cs.vu.nl
cs.vu.nl	TXT	"Vrije Universiteit - Math. & Comp. Sc."
cs.vu.nl	MX	1 zephyr.cs.vu.nl
cs.vu.nl	MX	2 tornado.cs.vu.nl
cs.vu.nl	MX	3 star.cs.vu.nl
star.cs.vu.nl	HINFO	Sun Unix
star.cs.vu.nl	MX	1 star.cs.vu.nl
star.cs.vu.nl	MX	10 zephyr.cs.vu.nl
star.cs.vu.nl	A	130.37.24.6
star.cs.vu.nl	A	192.31.231.42
zephyr.cs.vu.nl	HINFO	Sun Unix

RR type	Entity	Description
SOA	Zone	"Start of Authority": Holds information on the represented zone
A	Host	Contains an IP address (4 bytes) of the host this node represents
MX	Domain	Refers to a mail server to handle mail addressed to this node
NS	Zone	Refers to a name server that implements the represented zone
HINFO	Host	Holds information on the host this node represents
TXT	Any kind	Contains any human-readable information for an entity
...

- Iterative Name Resolution

- IP address for ftp.cs.vu.nl?

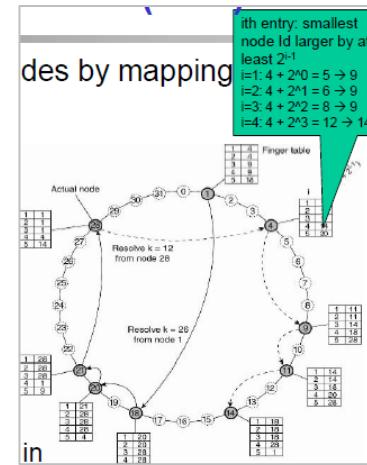


- Flat Naming

- Useful for addressing space in a homogeneous way
 - E.g., memory addressing
- Very common in centralized systems
 - E.g., memory, low-level disk access
 - In decentralized systems very complicated
- Naïve approaches
 - Every node knows all names and addresses
 - Every node knows only some
 - Flood the network asking who has the name in question
 - The node that has that name replies
 - Reminds of peer-to-peer systems

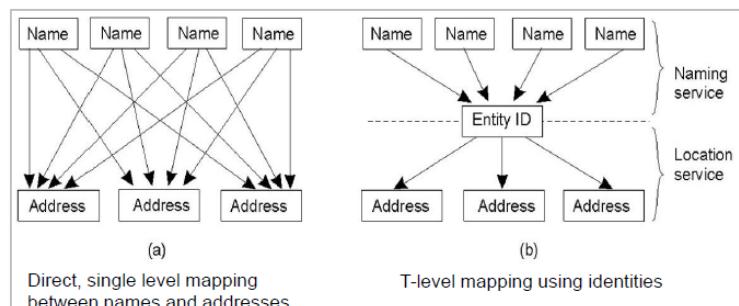
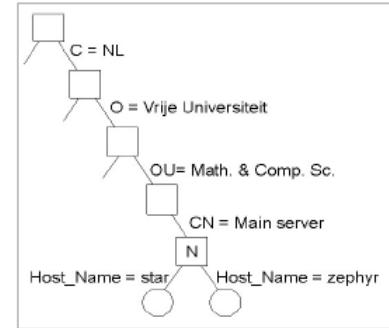
- Distributed Hash Tables (DHT)
 - Chord DHT, supports many nodes by mapping
 - Key → value
 - Also useful for name → address
 - Each node has a unique ID and maintains small routing table (pointers to other nodes)
 - How to find node that holds the value for a key?
 - Find largest node ID ≤ key in routing table
 - If none exists: Values are stored in the node with the next higher ID in routing table
- Attribute-based Names
 - Example: X.500 Name Space
 - Structured and attribute-based naming
 - Used for Lightweight Directory Access Protocol (LDAP) and for Microsoft's Active Directory
 - Each attribute called a Relative Distinguished Name (RDN)
 - Sequence of RDNs gives globally unique name
 - Better searching possible than with DNS with queries
 - E.g., Find “C = NL and CN = Main server”
 - Example
 - X.500 directory entry using X.500 naming conventions

Attribute	Abbr.	Value
Country	C	NL
Locality	L	Amsterdam
Organization	O	Vrije Universiteit
OrganizationalUnit	OU	Math. & Comp. Sc.
CommonName	CN	Main server
Mail_Servers	--	130.37.24.6, 192.31.231, 192.31.231.66
WWW_Server	--	130.37.21.11

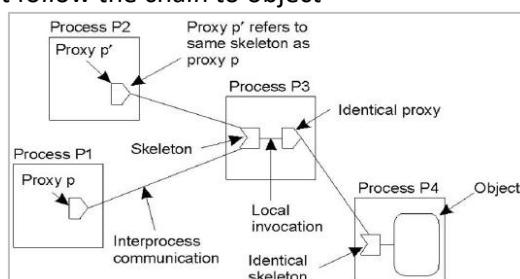


- X.500 naming graph

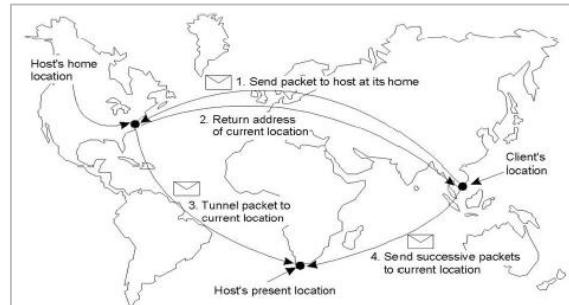
- Naming versus Locating Entities
 - Names do not generally indicate, where an object is located



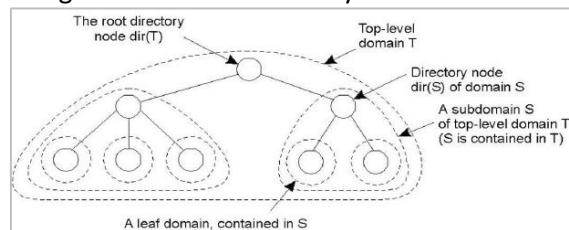
- Forwarding Pointers (Movements)
 - Forwarding pointer use (proxy, skeleton) pairs
 - When objects move, they leave a (proxy, skeleton) pair
 - Calls just follow the chain to object



- Home-based Approaches (Comparison)
 - Each host has a fixed home location, e.g., an IP address
 - When a host moves to a new network, it receives a new (temporary) IP address (Mobile IP)
 - The temporary IP address is stored at the home location



- Hierarchical Location Services
 - Locations are split into domains
 - Each having an associated directory node



V11 Chapter 12: Distributed File Systems

- Objectives
 - To be reminded on storage systems and their properties
 - To understand the basics of file system modules and attributes
 - To compare differences of file systems and their distributed sisters as well as understand major characteristics of example file systems
- Topics
 - Storage, file systems, and file attribute records
 - Reference counting
 - Distributed file systems: Operations and requirements
 - File service architecture
 - Network File System (NFS)
 - Andrew File System (AFS)
- Storage Systems and Their Properties

System	Sharing across network	Persistence	Distributed cache/replicas	Consistency	Example
Main memory	✗	✗	✗	1	RAM
File system	✗	✓	✗	1	UNIX file system
Distributed file system	✓	✓	✓	2 ✓	Sun NFS
Web	✓	✓	✓	✗	Web server
Distributed shared memory	✓	✗	✓	2	Ivy (DSM, Ch. 18)
Remote objects (RMI)	✓	✗	✗	1	CORBA
Persistent object store	✓	✓	✗	1	CORBA Persistent Object Service
Peer-to-peer storage system	✓	✓	✓	3	OceanStore (Ch. 10)

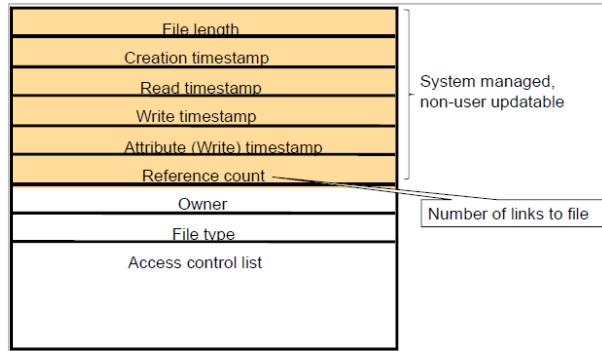
Consistency types:

- 1: Strict one-copy
- 2: Slightly weaker guarantees
- 3: Considerably weaker guarantees

- File System Modules

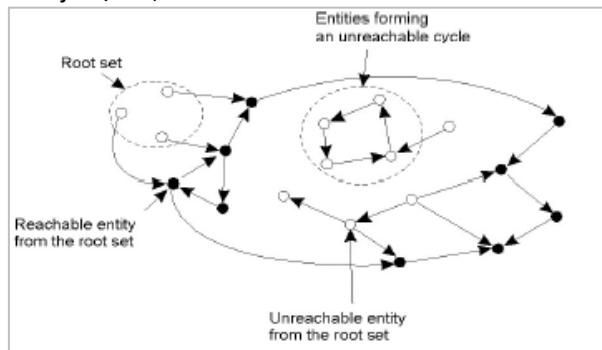
Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	disk I/O and buffering

- File Attribute Record Structure



- Reference Counting

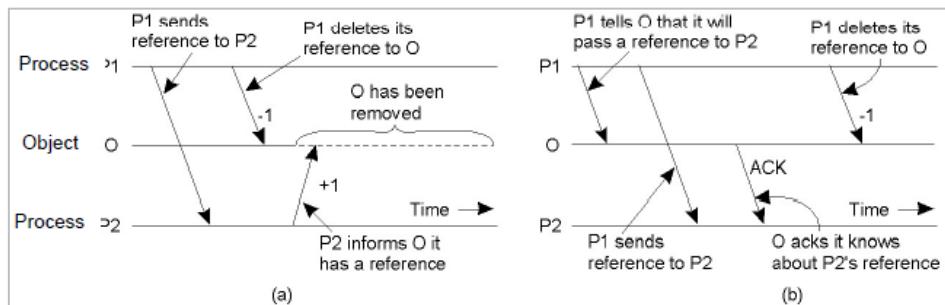
- Maintenance task for distributed systems, OSes
 - Applied to files and objects
- E.g., garbage collection
 - Delete object/file, if not reachable from root



- How to run such a scheme in a distributed manner?

- Distributed Reference Counting

- Copying a reference to another process and incrementing the counter can be too late
- A solution



- UNIX File System Operations

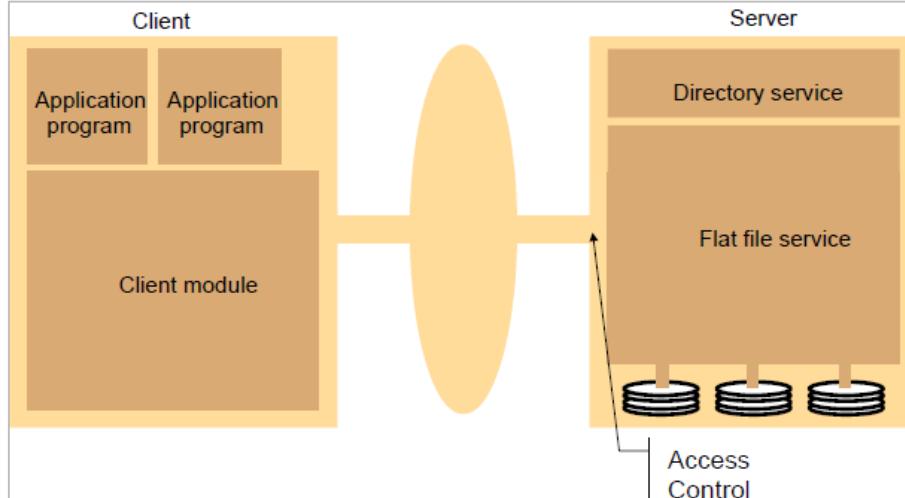
<code>filedes = open(name, mode)</code>	Opens an existing file with the given <i>name</i> .
<code>filedes = creat(name, mode)</code>	Creates a new file with the given <i>name</i> . Both operations deliver a file descriptor referencing the open file. The <i>mode</i> is <i>read</i> , <i>write</i> or both.
<code>status = close(filedes)</code>	Closes the open file <i>filedes</i> .
<code>count = read(filedes, buffer, n)</code>	Transfers <i>n</i> bytes from the file referenced by <i>filedes</i> to <i>buffer</i> .
<code>count = write(filedes, buffer, n)</code>	Transfers <i>n</i> bytes to the file referenced by <i>filedes</i> from <i>buffer</i> . Both operations deliver the number of bytes actually transferred and advance the read-write pointer.
<code>pos = lseek(filedes, offset, whence)</code>	Moves the read-write pointer to offset (relative or absolute, depending on <i>whence</i>).
<code>status = unlink(name)</code>	Removes the file <i>name</i> from the directory structure. If the file has no other names, it is deleted.
<code>status = link(name1, name2)</code>	Adds a new name (<i>name2</i>) for a file (<i>name1</i>).
<code>status = stat(name, buffer)</code>	Gets the file attributes for file <i>name</i> into <i>buffer</i> .

- Distributed File System Requirements

- Transparency
 - Access, Location, Scaling, and all the others
- Concurrent file updates
- File replication
- Hardware and software heterogeneity
- Fault tolerance
- Consistency (one-copy update semantics)
- Security
- Efficiency

- File Service Architecture

- Services: read, write, create, delete, ...



- Flat File Service Operations

<code>Read(FileId, i, n) -> Data</code>	If $1 \leq i \leq \text{Length}(\text{File})$: Reads a sequence of up to <i>n</i> items from a file starting at item <i>i</i> and returns it in <i>Data</i> .
<code>Write(FileId, i, Data)</code>	If $1 \leq i \leq \text{Length}(\text{File})+1$: Writes a sequence of <i>Data</i> to a file, starting at item <i>i</i> , extending the file if necessary.
<code>Create() -> FileId</code>	Creates a new file of length 0 and delivers a UFID for it.
<code>Delete(FileId)</code>	Removes the file from the file store.
<code>GetAttributes(FileId) -> Attr</code>	Returns the file attributes for the file.
<code>SetAttributes(FileId, Attr)</code>	Sets the file attributes (only those attributes that are not shaded in Figure 8.3).

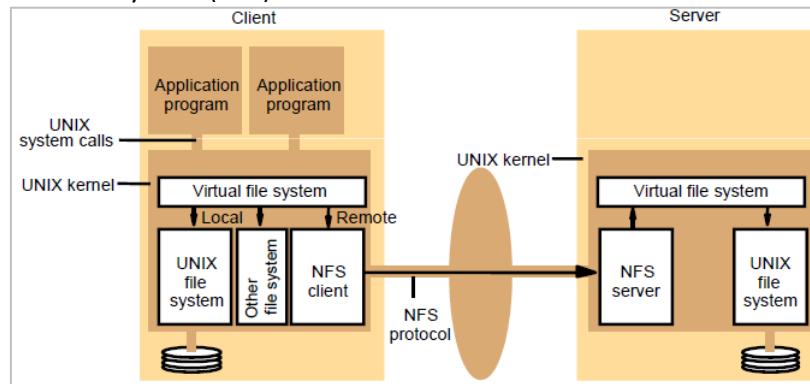
- Note: This interface is repeatable (i.e., idempotent) and stateless!

- Directory Service Operations

<i>Lookup(Dir, Name) -> FileId</i> — throws <i>NotFound</i>	Locates the text name in the directory and returns the relevant UFID. If <i>Name</i> is not in the directory, throws an exception.
<i>AddName(Dir, Name, FileId)</i> — throws <i>NameDuplicate</i>	If <i>Name</i> is not in the directory, adds <i>(Name, FileId)</i> to the directory and updates the file's attribute record. If <i>Name</i> is already in the directory: throws an exception.
<i>UnName(Dir, Name)</i> — throws <i>NotFound</i>	If <i>Name</i> is in the directory: the entry containing <i>Name</i> is removed from the directory. If <i>Name</i> is not in the directory: throws an exception.
<i>GetNames(Dir, Pattern) -> NameSeq</i>	Returns all the text names in the directory that match the regular expression <i>Pattern</i> .

- Case Study 1: NFS Architecture

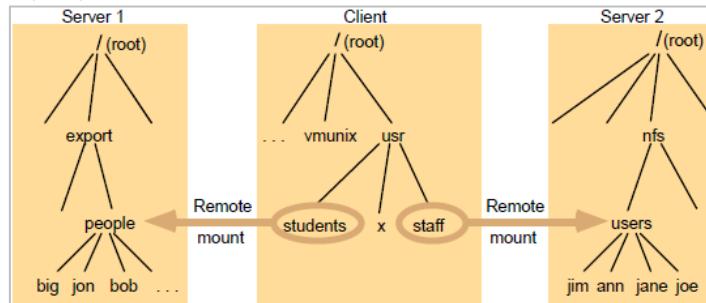
- Network File System (NFS)



- NFS Server Operations Simplified

<i>lookup(dirfh, name) -> fh, attr</i>	Returns file handle and attributes for the file <i>name</i> in the directory <i>dirfh</i> .
<i>create(dirfh, name, attr) -> newfh, attr</i>	Creates a new file name in directory <i>dirfh</i> with attributes <i>attr</i> and returns the new file handle and attributes.
<i>remove(dirfh, name) status</i>	Removes file name from directory <i>dirfh</i> .
<i>getattr(fh) -> attr</i>	Returns file attributes of file <i>fh</i> . (Similar to the UNIX <i>stat</i> system call.)
<i>setattr(fh, attr) -> attr</i>	Sets the attributes (mode, user id, group id, size, access time and modify time of a file). Setting the size to 0 truncates the file.
<i>read(fh, offset, count) -> attr, data</i>	Returns up to <i>count</i> bytes of data from a file starting at <i>offset</i> . Also returns the latest attributes of the file.
<i>write(fh, offset, count, data) -> attr</i>	Writes <i>count</i> bytes of data to a file starting at <i>offset</i> . Returns the attributes of the file after the write has taken place.
<i>rename(dirfh, name, todirfh, toname) -> status</i>	Changes the name of file <i>name</i> in directory <i>dirfh</i> to <i>toname</i> in directory to <i>todirfh</i>
<i>link(newdirfh, newname, dirfh, name) -> status</i>	Creates an entry <i>newname</i> in the directory <i>newdirfh</i> which refers to file <i>name</i> in the directory <i>dirfh</i> .
<i>symlink(newdirfh, newname, string) -> status</i>	Creates an entry <i>newname</i> in the directory <i>newdirfh</i> of type symbolic link with the value <i>string</i> . The server does not interpret the <i>string</i> but makes a symbolic link file to hold it.
<i>readlink(fh) -> string</i>	Returns the string that is associated with the symbolic link file identified by <i>fh</i> .
<i>mkdir(dirfh, name, attr) -> newfh, attr</i>	Creates a new directory <i>name</i> with attributes <i>attr</i> and returns the new file handle and attributes.
<i>rmdir(dirfh, name) -> status</i>	Removes the empty directory <i>name</i> from the parent directory <i>dirfh</i> . Fails if the directory is not empty.
<i>readdir(dirfh, cookie, count) -> entries</i>	Returns up to <i>count</i> bytes of directory entries from the directory <i>dirfh</i> . Each entry contains a file name, a file handle, and an opaque pointer to the next directory entry, called a <i>cookie</i> . The <i>cookie</i> is used in subsequent <i>readdir</i> calls to start reading from the following entry. If the value of <i>cookie</i> is 0, reads from the first entry in the directory.
<i>statfs(fh) -> fsstats</i>	Returns file system information (such as block size, number of free blocks and so on) for the file system containing a file <i>fh</i> .

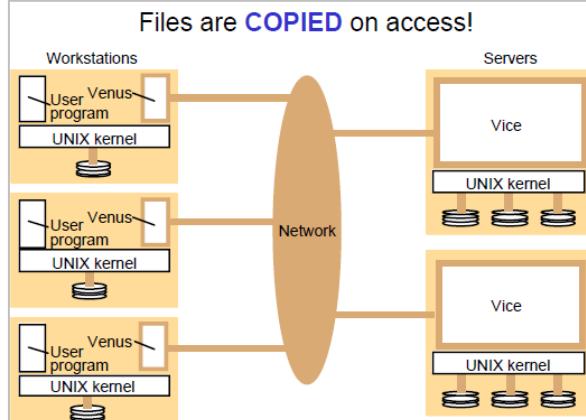
- Local and Remote File Systems Access
 - Accessible from an NFS client
 - The file system mounted at /usr/students in the client is actually the sub-tree located at /export/people in Server 1
 - The file system mounted at /usr/staff in the client is actually the sub-tree located at /nfs/users in Server 2



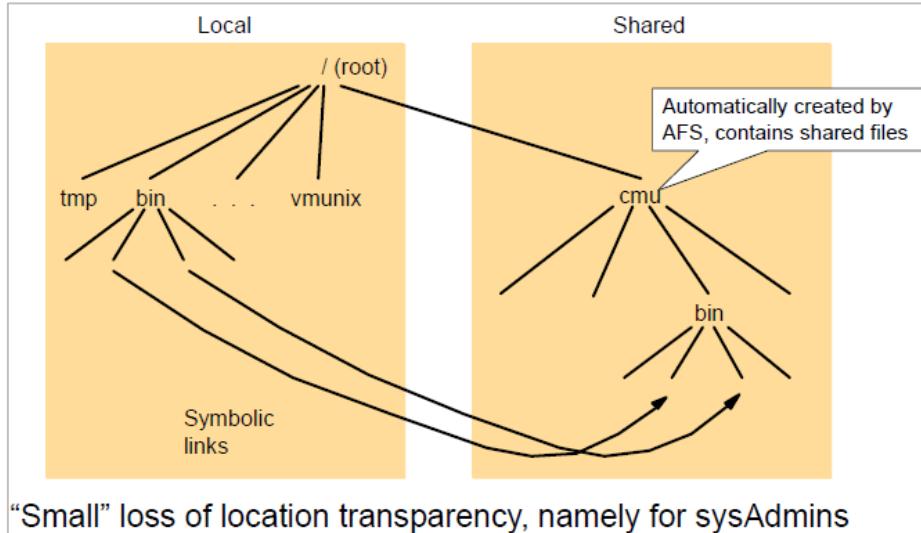
- NFS Concerns
 - Auto-mounter
 - Mount on access of any non-mounted file
 - Valid for a set of registered directories
 - Server caching
 - Conventional UNIX: read-ahead and delayed write
 - NFS: write-through
 - Client caching
 - Timeouts
 - Cache valid iff $(T-T_c < c)$ or $(\text{FileAttr}_{\text{client}} = \text{FileAttr}_{\text{server}})$
 - Security
 - Quite flaky → Better with access control systems (Kerberos)

- Case Study 1: Summary
 - Transparency
 - Access
 - Location
 - Mobility
 - Scaling
 - Concurrent file updates
 - File replication
 - Hardware and software heterogeneity
 - Fault tolerance
 - Consistency
 - Security
 - Efficiency
- | |
|--|
| NFS client process
File name space identical
Remounts possible
Only if no "hot spot files" |
| Not supported
Read-only files
Given
Stateless and idempotent
Close to one-copy semantics
Medium
Seems ok (still in larger use) |

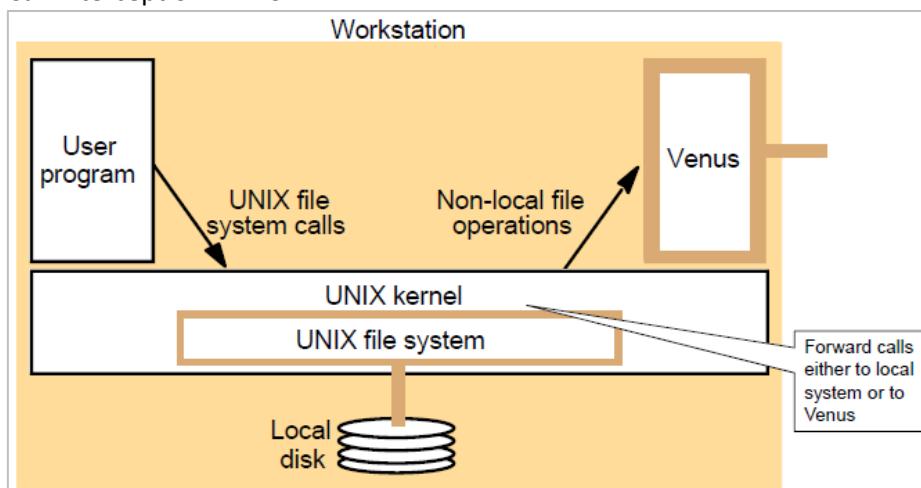
- Case Study 2: Andrew File System (AFS)



- File Name Space as Seen by AFS Clients



- System Call Interception in AFS

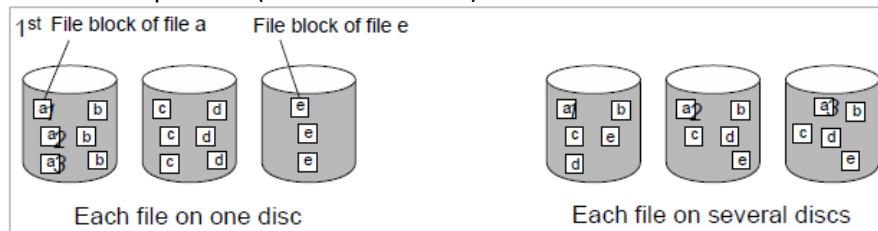


- Main Components of Vice Service Interface

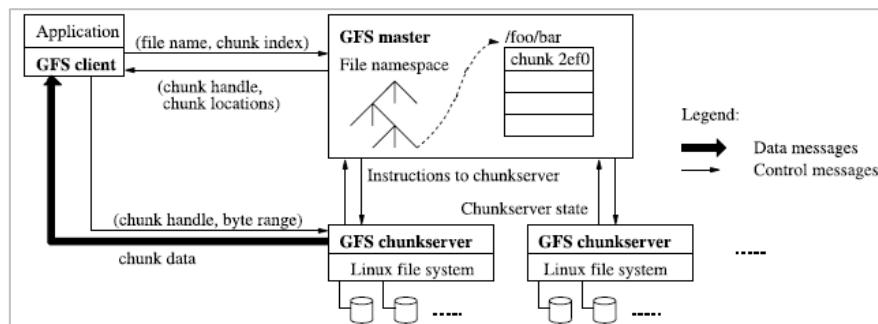
<i>Fetch(fid) -> attr; data</i>	Returns the attributes (status) and, optionally, the contents of file identified by the <i>fid</i> and records a callback promise on it.
<i>Store(fid, attr; data)</i>	Updates the attributes and (optionally) the contents of a specified file.
<i>Create() -> fid</i>	Creates a new file and records a callback promise on it.
<i>Remove(fid)</i>	Deletes the specified file.
<i>SetLock(fid, mode)</i>	Sets a lock on the specified file or directory. The mode of the lock may be shared or exclusive. Locks that are not removed expire after 30 minutes.
<i>ReleaseLock(fid)</i>	Unlocks the specified file or directory.
<i>RemoveCallback(fid)</i>	Informs server that a Venus process has flushed a file from its cache.
<i>BreakCallback(fid)</i>	This call is made by a Vice server to a Venus process. It cancels the callback promise on the relevant file.

Note: Callbacks ensure file-level update semantics (consistency)

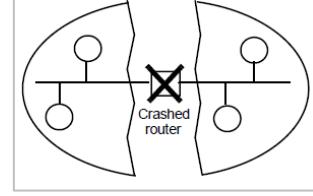
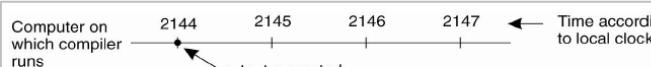
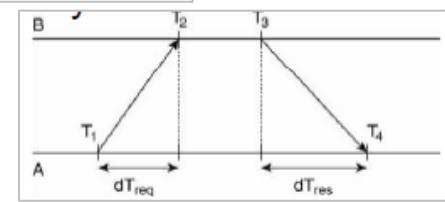
- AFS Other Aspects
 - Explicitly excludes data bases
 - Large files, many updates → This does not fit into AFS
 - UNIX Kernel modifications necessary
 - File system on user level
 - Location data base replicated throughout servers
 - Read-only replicas
 - Bulk transfers of 64 KB large chunks
 - Newer NFS: Partial file-caching
 - Performance: Less server load than NFS
 - Wide area support (1000 servers @ 150 sites)
- Case Study 3: Google File System
 - Designed for large files and appends to files
 - Files are distributed over multiple servers in chunks of 64 MB (like in a Redundant Array of Independent Disks (RAID)) → File striping
 - Chunks are replicated (not shown below)



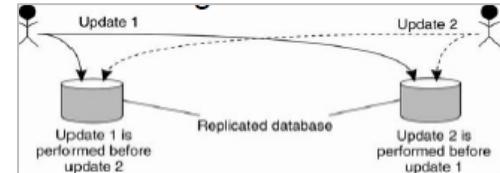
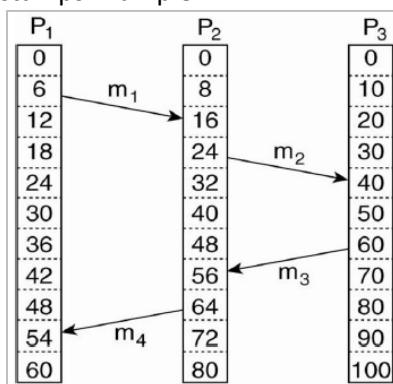
- Google File System Architecture
 - Master serves as directory only
 - All chunks are distributed over chunk server → better scalability



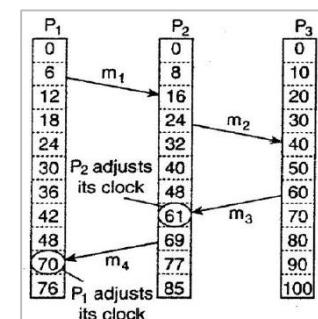
V12 Chapter 13: Synchronization in Distributed Systems

- Objectives
 - To develop an understanding of distributed view on time
 - To learn about alternative clock designs and implementation
 - To control sequentialized, distributed processes operations
 - To select a unique “master” in distributed settings
 - To combine multicast and group communications
 - Topics
 - Time Synchronization
 - Clock Synchronization
 - Logical and Vector Clocks
 - Mutual Exclusion
 - Leader Election
 - The Multicast Problem
 - Need for Synchronization
 - Being able to communicate as such is not sufficient
 - Communicating nodes also need to coordinate and synchronize for various tasks
 - Synchronize with respect to time
 - Not access a resource (e.g., a printer, or some memory location) simultaneously
 - Agree on an ordering of (distributed) events
 - Appoint a coordinator
 - Assumptions and Algorithms
 - Assumptions
 - Communication is reliable (but may incur delays)
 - Network partitioning might occur
 - Detecting failure is difficult
 - time-out is not reliable
 - Algorithms
 - Distributed Mutual exclusion
 - Elections
 - Multicast Communication
 - Clock Synchronization
 - Time consistency is not an issue for a single computer
 - Time never runs backward (a later reading of the clock returns a later time)
 - In distributed environments it can be a real challenge
 - Think of how make works
- 
- | | |
|---|--------------------------------------|
| <p>Computer on which compiler runs</p>  | <p>Time according to local clock</p> |
| <p>Computer on which editor runs</p>  | <p>Time according to local clock</p> |
- 
- Synchronization with a Time Server
 - A time server has very accurate time
 - E.g., atomic clock or GPS receiver
 - How can a client synchronize with a time server?
 - Problem: messages do not travel instantly
 - Cristian’s algorithm: algorithm:
 - Estimate the transmission delay to the server: $((T4-T1) (T3-T2)) / 2$
 - Used in the Network Time Protocol (NTP)
 - Cristian’s algorithm is run multiple times
 - Outlier values are ignored to rule out packets delayed due to congestion or longer paths

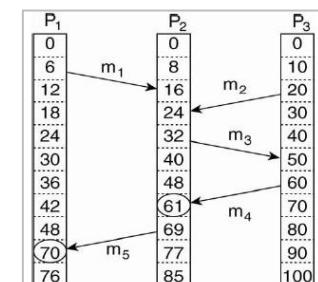
- Logical Clocks
 - Absolute time synchronization is not always needed
 - Need to ensure that the order in which events happen is preserved across all computers
 - More specifically: All computers should agree on a total ordering of events
 - Example
 - A person's account holds 1,000 CHF and he adds 100 CHF
 - At the same time, an accountant invokes a command that gives 1% interest to each account
 - Does that person's account end up with 1,110 CHF or 1,111 CHF instead?
- Lamport's Timestamps
 - In the classic paper as of 1978 Leslie Lamport defined the fundamental rules to reach consistent timestamps on events:
 1. If a and b are events on the same process, then if a occurs before b, $\text{CLOCK}(a) < \text{CLOCK}(b)$
 2. If a and b correspond to the events of a message being sent from the source process, and received by the destination process, respectively, then $\text{CLOCK}(a) < \text{CLOCK}(b)$, because a message cannot be received before it is sent
- Lamport's Timestamps Example



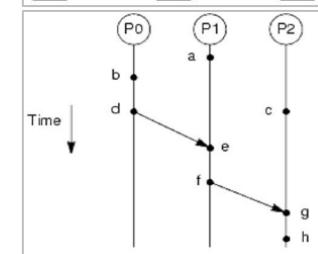
1. If a and b are events on the same process, then if a occurs before b, $\text{CLOCK}(a) < \text{CLOCK}(b)$
2. If a and b correspond to the events of a message being sent from the source process, and received by the destination process, respectively, then $\text{CLOCK}(a) < \text{CLOCK}(b)$, because a message cannot be received before it is sent



- Vector (Logical) Clocks
 - Lamport's Timestamps give total ordering of events
 - Notion of causality (dependencies between events) is lost
 - Total ordering is often too strict

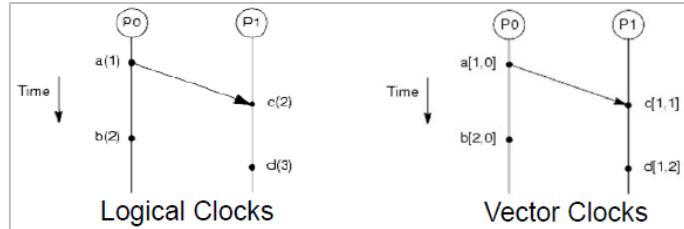
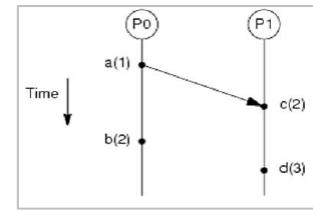


- Vector Clock Example
 - The reception of m3 (50) could depend on the reception of m2 (24) and m1 (16)
 - That's correct
 - The sending of m2 (20) seems to depend on the reception of m1 (16) But is it?
 - No!



- Causality
 - Generally, two events can:
 - Be linked by a dependency
 $a \rightarrow b$, which means a happens before b
 - E.g., $b \rightarrow d, d \rightarrow e, b \rightarrow e, b \rightarrow h$
 - Independent (concurrent)
 - E.g., a and c, or a and b

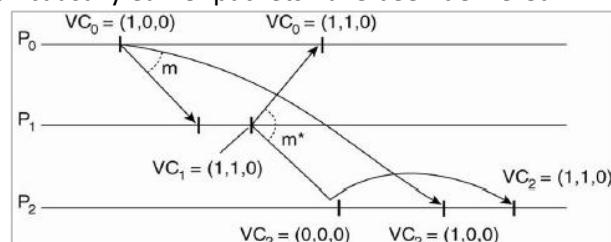
- Inefficiency of Logical Clocks
 - To observe causality with Logical Clocks (a.k.a. Lamport Timestamps), the approach may fail:
 - d consistently has a later timestamp than b, so one would (wrongly) assume that $b \rightarrow d$
- Logical and Vector Clocks
 - With Vector Clocks
 - $a \rightarrow c$, because $[1,0] < [1,1]$
 - $a \rightarrow d$, because $[1,0] < [1,2]$
 - Same for $a \rightarrow b$, $c \rightarrow d$
 - But b and d are independent (concurrent), because there is no clear order between $[2,0]$ and $[1,2]$

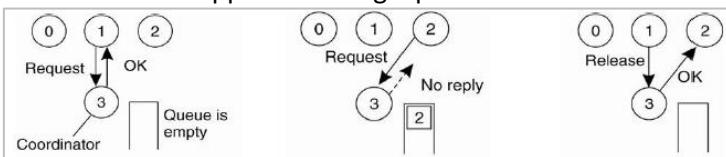


- Vector Clocks
 - Assuming N nodes
 - Each node maintains a vector of N logical clocks
 - One logical clock is its own
 - The rest N-1 logical clocks are estimations for the other nodes
 - Logical clocks are managed as follows:
 - They are all initialized with zero
 - When an event happens in a node, it increases its own logical clock in the vector by one
 - When a node sends a message, it includes its whole vector
 - When a node receives a message, it updates each element in its vector by taking the maximum of the value in its own vector clock and the value in the vector in the received message (for every element)
 - An event a is considered to happen before event b, only if all elements of the VC of a are less than or equal than the respective elements of the VC of b
 - In fact, at least one element has to be lower

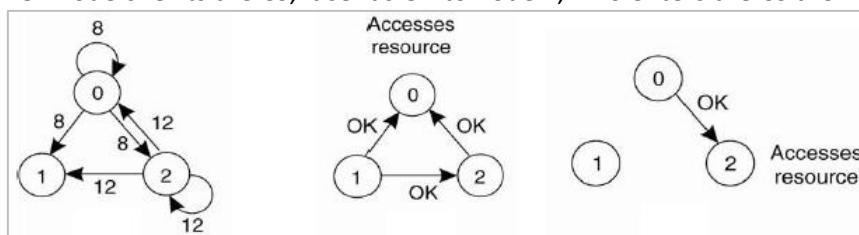
- Causal Communication
 - Vector Clocks can be used to enforce causal communication
 - Do not deliver a packet until all causally earlier packets have been delivered

- Assumptions
 - No packets are lost
 - Clocks are increased only when sending a new message
 - A packet is delivered when only the sender's logical clock is increased



- Causal Communication's "Location" (1)
 - Included in middleware layer
 - Advantage: Generic approach
 - Drawback
 - Potential (but not definite) causality is captured
 - Even messages that are not related, but happen to occur in a given order, are assumed dependent: this makes it "heavier" than necessary
 - Some causality may not be captured
 - Alice posts a message, and then calls Bob and informs him about that message. Bob may take some other action that depends on the information he got from Alice, even before receiving the official message of Alice. This causality is not captured by the middleware.
 - External communication can mess up the assumptions of the middleware
 - Application-specific
 - Advantages
 - Can be tuned to be more lightweight
 - Can be tuned to be more accurate
 - Drawback
 - Puts the burden of causality checking on the application developer
- Mutual Exclusion Problem
 - Application level protocol
 - Enter Critical Section
 - Use resource exclusively
 - Exit Critical Section
 - Requirements
 - Safety: At most one process may execute in Critical Section at once
 - Liveness: Requests to enter and exit the critical section should eventually succeed (no deadlocks or livelocks should occur, and fairness should be enforced)
 - Ordering: Requests are handled in order of appearance
 - Evaluation criteria
 - Bandwidth (number of messages)
 - Client waiting time to enter Critical Section
 - Vulnerabilities
 - Alternative approaches
 - Centralized Approach
 - Distributed Approach
 - Token-Ring Approach
- Centralized Approach
 - Simplest algorithm to achieve Mutual Exclusion
 - Simulate what happens in a single processor
 - 
 - Easy to implement, few messages (3 per CS: Request, OK, Release), fair (First-In-First-Out), no starvation
 - Single point of failure, processes cannot distinguish between dead coordinator or busy resource

- Distributed Approach
 - Ricart and Agrawala's algorithm
 - Nodes use logical clocks: all events are in total order
 - When a node wants to enter a CS (CriticalSection) it sends a message with its (logical) time and CS name to all other nodes
 - When a node receives such a request
 - If it is not interested in this CS, it replies OK immediately
 - If it is interested in this CS:
 - If its message's timestamp was older, then replies OK,
 - Else, it puts the sender in a queue and doesn't reply anything (yet)
 - If it is already in the CS, it puts the sender in a queue and doesn't reply anything (yet)
 - A node enters the CS when it received OK but all other nodes
 - A node that exits the CS, sends immediately OK to all nodes that it may have placed in the queue
 - Example
 - Nodes 0 and 2 express interest in the CS almost simultaneously
 - Node 0's message has an earlier timestamp, so it wins
 - Node 1 (not interested) and node 2 (interested, but higher timestamp) send OK to node 0, so node 0 enters the CS
 - When node 0 exits the CS, it sends OK to node 2, who enters the CS then



- Ricart and Agrawala's Algorithm

On initialization

state := RELEASED;

To enter the section

state := WANTED;

Multicast *request* to all processes;

T := *request*'s timestamp;

Wait until (number of replies received = (*N* − 1));

state := HELD;

} request processing deferred here

On receipt of a request $\langle T_i, p_i \rangle$ at p_j ($i \neq j$)

if (*state* = HELD or (*state* = WANTED and $(T, p_j) < (T_i, p_i)$))

then

queue *request* from p_i without replying;

else

reply immediately to p_i ;

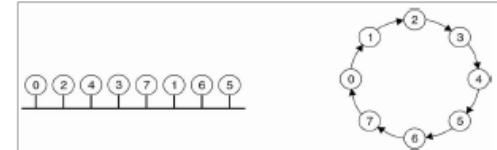
end if

To exit the critical section

state := RELEASED;

reply to any queued requests;

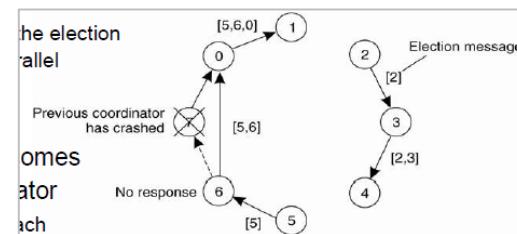
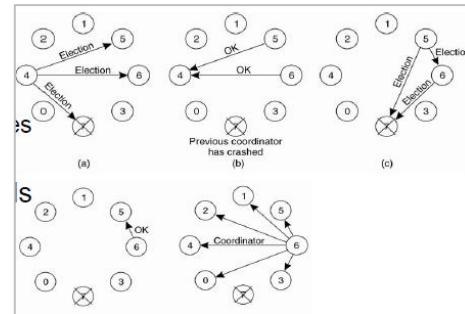
- Distributed Approach Evaluation
 - Problems:
 - More messages: $2*(n-1)$
 - No single point of failure ... but n points of failure
 - A failure on any one of n processes brings the system down
 - Some improvements have been proposed
 - Maekawa's algorithm: Don't wait for approval from all, but from the majority
 - Moral conclusion: Distributed Algorithms are not always more robust to failures!
- Token-Ring Approach
 - Nodes are organized in a ring
 - A token goes around
 - Each one passes it to its successor
 - If a node wants to enter a CS, it can do so when it gets the token
 - It is guaranteed it is the only one holding the token
 - When it exits the CS, it passes the token to the next node
 - Very simple, fair, no starvation
 - Messages per entry/exit: 1 to infinite
 - Problem upon a token lost
 - Long delay might mean that the token is lost or someone is using it
- Comparison



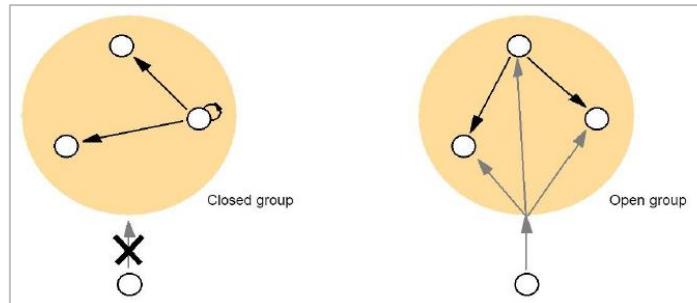
Algorithm	Messages per entry/exit	Waiting time to enter CS (best case)	Problems
Centralized	3	2	Crash of coordinator
Distributed	$2*(n-1)$	$2*(n-1)$	Crash of any node
Token ring	1 to infinite	0 to $n-1$	Lost token, crash of any node

- Leader Election Problem
 - Choice of one node among a selection of participants
 - Each process gets a number (no two have the same)
 - For each process p_i : There is a variable elected i
 - Initialize: Set all elected $i = \text{NONE}$
 - Requirements
 - Safety: Participant p_i has elected $i = \text{NONE}$ or p_i , where p is the number of the elected process
 - Liveness: All participating processes p eventually have elected $i = p$ or crash
- Bully Algorithm
 - Assumptions
 - Synchronous messages
 - Timeouts
 - Message types
 - Election (announcement)
 - Ok (response)
 - Coordinator (result)

- Bully Algorithm Election Procedure
 1. When a node notices that the coordinator is not responding, it starts the election process
 2. Sends election message to all processes with a higher ID; if no response, then it is elected
 3. If one gets an election message and has higher ID, he replies OK and starts election
 4. Process that knows it has the highest ID elects itself by sending a coordinator message to all others
- Bully Algorithm Example
 - Remarks:
 - 7 was the coordinator, but it failed
 - 4 notices it first and starts election
 - Notifies higher nodes
 - Eventually 6 prevails and becomes the new coordinator
- Ring Algorithm
 - Assumptions
 - Synchronous messages
 - Timeouts
 - Nodes are organized in a ring
 - Message type
 - Election: <list of IDs>
- Ring Algorithm Election Procedure
 1. When a node notices that the coordinator is not responding, it starts the election process
 2. Sends election message to its successor, with a list containing only its own ID
 3. When one gets an election message that originated at a different node, it appends its ID to the list, and forwards the message to its successor
 4. When one gets back its own election message, it picks the highest ID as the leader and announces it to everyone
- Ring Algorithm Example - Remarks
 - 7 was the coordinator, but it failed
 - Nodes 2 and 5 notice it has crashed
 - They both start the election procedure in parallel
 - Eventually 6 prevails and becomes the new coordinator
 - Both 2 and 5 reach the same conclusion
- Multicast
 - Multicast messages are useful in distributed systems
 - Multiple receivers can receive the same message
 - Characteristics
 - Fault tolerance based on replicated servers
 - Requests sent to all servers, all service may not fail at the same time
 - Discovering services in spontaneous networking
 - Locating available services, e.g., in order to register interfaces
 - Better performance through replicated data
 - Data are replicated, new values are multicast to all processes
 - Propagation of event notifications
 - Groups may be notified on happenings



- Group Communications
 - Group communications are an important building block in distributed systems, particularly for reliable ones
 - Characteristics and application areas
 - Reliable dissemination of information to many clients
 - Financial industry
 - Events distributed to multiple users to preserve a common user view
 - Collaborative applications, multi-user games
 - Fault tolerance strategies
 - Consistent updates of replicated data, e.g., backups for machines
 - System monitoring and management
 - Data and configuration collection and distribution
- Multicast Problem
 - Process sends a single send operation
 - Efficiency
 - Delivery guarantees
 - System model
 - $\text{Multicast}(m, g) \rightarrow$ sends message m to all members of group g
 - $\text{deliver}(m) \rightarrow$ delivers the message to the receiving process
 - Properties
 - Integrity: Each message is delivered at most once
 - Validity: If $\text{multicast}(m, g)$ and $p \in g \rightarrow$ eventually $p.\text{deliver}(m)$
 - Agreement: If a message of $\text{multicast}(m, g)$ is delivered to p, it should be delivered also to all other processes in g
- Open and Closed Groups
 - A group is closed iff only members can send messages
 - A group is open iff any outside process can send to it



- Basic Multicast
 - Basic multicast:
 - $B\text{-multicast}(m, g)$
 - For each $p \in g$, do $\text{send}(p, m)$
 - On $\text{receive}(m)$ at p: $B\text{-deliver}(m)$ at p
 - Problems
 - Implosion of acknowledgements
 - Not reliable
 - Reliable Multicast Algorithm
 - Problems
 - Inefficient: $O(|g|^2)$ messages
 - Implosion of acknowledgements
- ```

On initialization
Received := {};

For process p to R-multicast message m to group g
 B-multicast(g, m); // p ∈ g is included as a destination

On B-deliver(m) at process q with g = group(m)
 if (m ∉ Received)
 then
 Received := Received ∪ {m};
 if (q ≠ p) then B-multicast(g, m); end if
 R-deliver m;
 end if

```