#1 Introduction

System structure:
1) system startup
2) I / O
3) storage

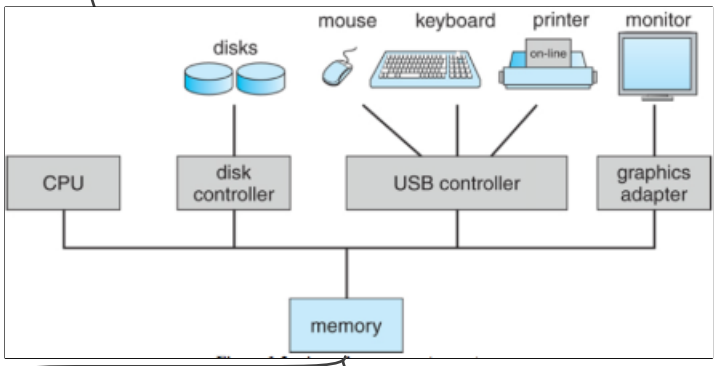
Operating System
Program manage computer hardware

> basis application programs
> intermediary between user and hardware
: convenient (mobile computers)
: efficient (mainframe computers)

Computer = Hardware + OS + Apps + Users

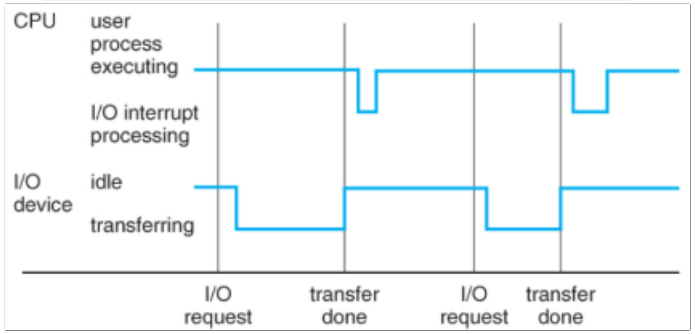
- OS serves as interface between HW and (Apps & Users)
- OS provides services for Apps & Users
- OS manages resources (Government model, it doesn't produce anything.)
- Debates about what is included in the OS Just the kernel, or everything the vendor ships? (Consider the distinction between system applications and 3rd party or user apps.)

Computer System Organization
What are all the parts, and how do they fit together?



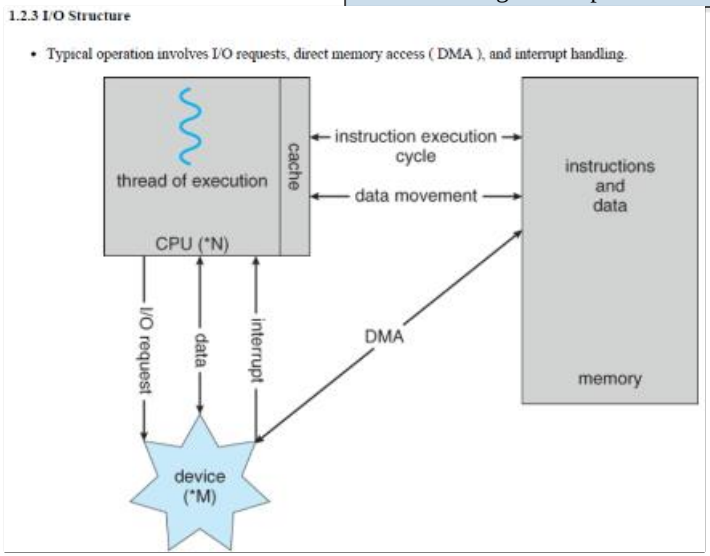
Computer System Operation

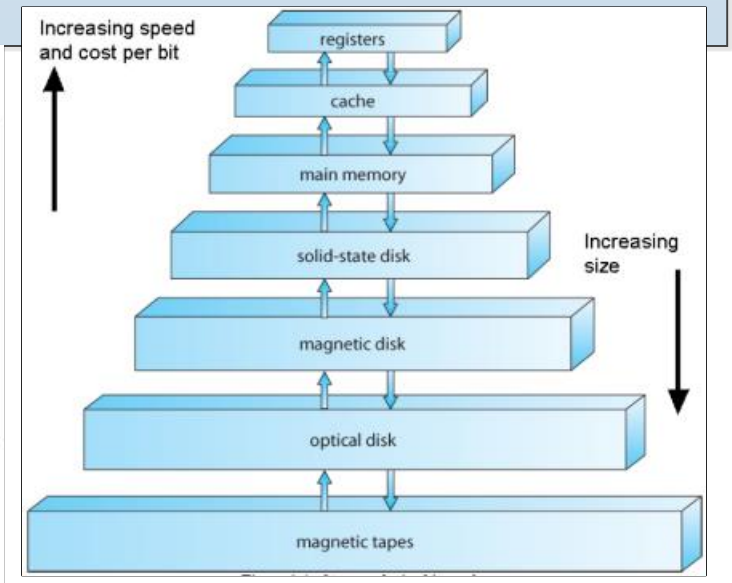
- Shared memory between CPU and I/O cards
- Time slicing for multi-process operation
- Interrupt handling clock, HW, SW
- Implementation of system calls



Storage Structure

- **Main memory (RAM)**
  > Programs must be loaded into RAM to run.
  > Instructions and data fetched from RAM into registers.
  > RAM is volatile
  > "Medium" size and speed
- **Other electronic (volatile) memory is faster, smaller, and more expensive per bit:**
  > Registers
  > CPU Cache
- **Nonvolatile memory ("permanent" storage) is slower, larger, and less expensive per bit:**
  > Electronic disks
  > Magnetic disks
  > Optical disks
  > Magnetic Tapes





Computer-System Architecture
Different Operating Systems for Different Kinds of Computer Environments

Single-Processor Systems

One main CPU which manages the computer and runs user apps.
Other specialized processors (disk controllers, GPUs, etc.) do not run user apps.

Multiprocessor Systems

Two or more processors close communication share computer bus + clock + memory + devices

1. Increased throughput Faster execution, but not 100% linear speedup.
2. Economy of scale Peripherals, disks, memory, shared among processors.
3. Increased reliability
   - Failure of a CPU slows system, doesn't crash it.
   - Redundant processing provides system of checks and balances. (e.g. NASA)

Blade Servers

Multiple processor boards, I/O boards networking boards placed same chassis
Each blade-processor board boots independent
Runs own operating system
Boards multiprocessor (multiple independent multiprocessor system)

Clustered Systems

Independent systems, with shared common storage and connected by a high-speed LAN, working together.
Special considerations for access to shared storage are required, (Distributed lock management), as are collaboration protocols.

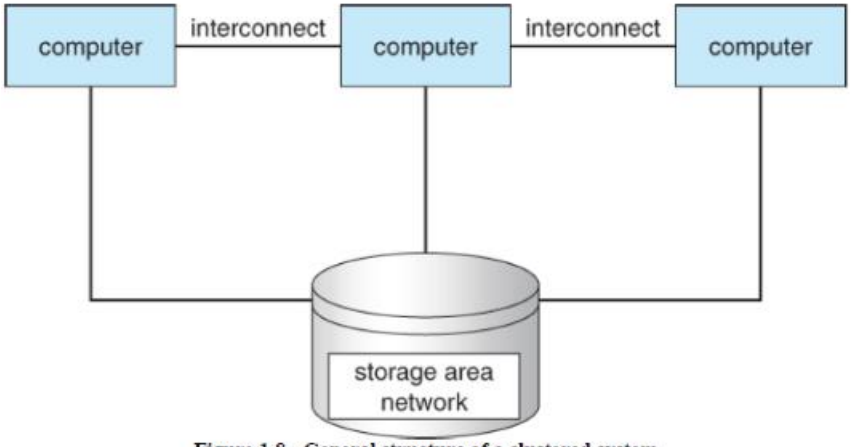


Figure 1.8 - General structure of a clustered system

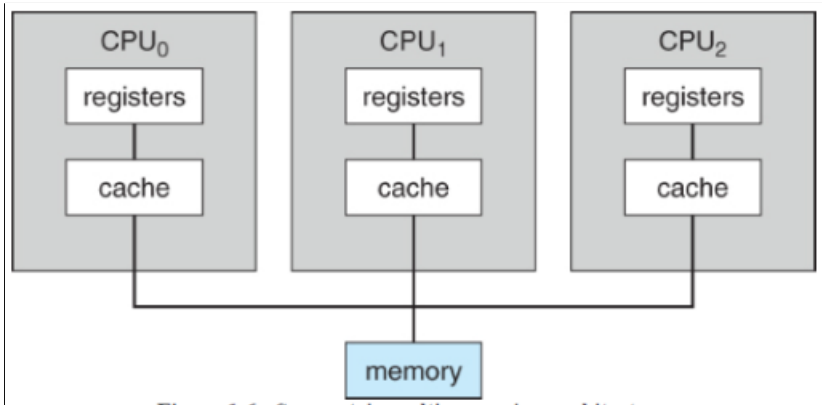


Figure 1.6 - Symmetric multiprocessing architecture

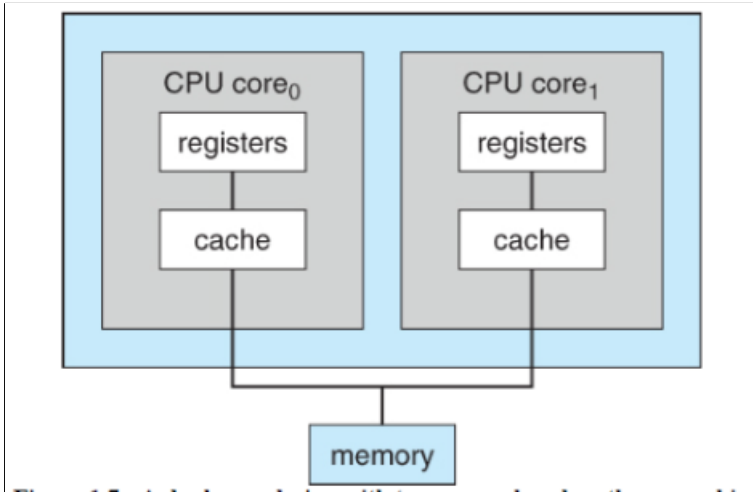


Figure 1.7 - A dual-core design with two cores placed on the same chip

Operating-System Structure

A timesharing (multiuser multitasking) OS requires:

- Memory management
- Process management
- Job scheduling
- Resource allocation strategies
- Swap space / virtual memory in physical memory
- Interrupt handling
- File system management
- Protection and security
- Inter-process communications

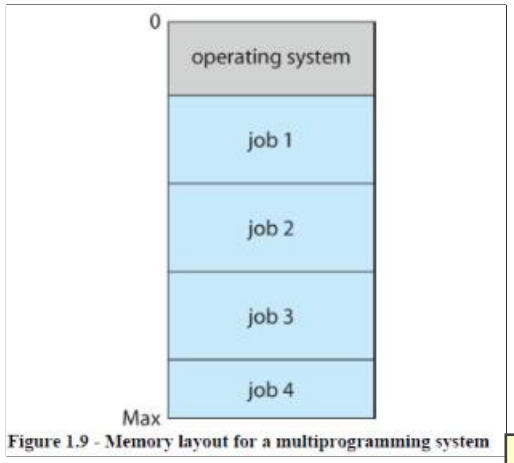


Figure 1.9 - Memory layout for a multiprogramming system

Operating-System Operations

Interrupt-driven nature of modern OSes requires that erroneous processes not be able to disturb anything else.

Timer

- Before kernel begins executing user code, a timer set to generate interrupt.
- Timer interrupt handler reverts control back to kernel.
- Assures no user process take over system.
- Timer control privileged instruction, (requiring kernel mode.)

Dual-Mode and Multimode Operation

- User mode when executing harmless code in user applications
- Kernel mode (a.k.a. system mode, supervisor mode, privileged mode) when executing potentially dangerous code in the system kernel.
- Certain machine instructions (privileged instructions) can only be executed in kernel mode.
- Kernel mode can only be entered by making system calls. User code cannot flip the mode switch.
- Modern computers support dual-mode operation in hardware, and therefore most modern OSes support dual-mode operation.

- Modes extended beyond two, requiring more than single mode bit
- CPUs support virtualization use one extra bits indicate when virtual machine manager, VMM, in control of system. VMM ore privileges than ordinary user programs, not many as full kernel.
- System calls implemented software interrupts, causes hardware's interrupt handler transfer control to appropriate interrupt handler, part of operating system, switching the mode bit to kernel mode in the process. The interrupt handler checks exactly which interrupt generated, checks additional parameters (generally passed through registers) if appropriate, calls appropriate kernel service routine to handle the service requested by the system call.
- User programs' attempts to execute illegal instructions (privileged or nonexistent instructions), or to access forbidden memory areas, also generate software interrupts, which are trapped by the interrupt handler and control is transferred to the OS, which issues an appropriate error message, possibly dumps data to a log (core) file for later analysis, and then terminates the offending program.

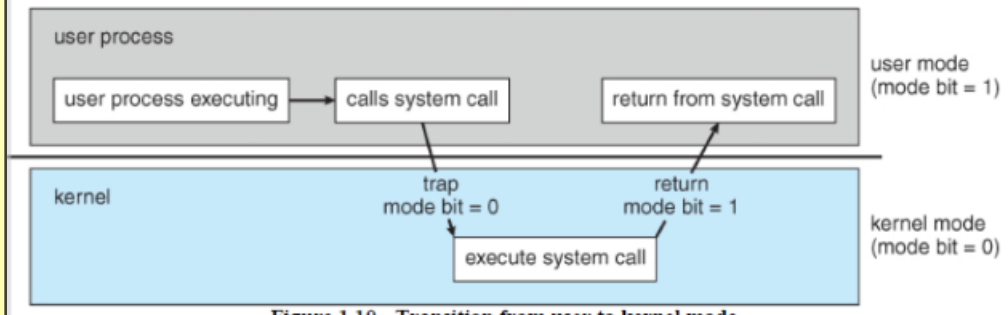


Figure 1.10 - Transition from user to kernel mode

Process Management
(program in execution)

OS responsible tasks with regards to process management:
1) Scheduling processes and threads on CPU's
2) Creating and deleting both user and system processes
3) Suspending and resuming processes
4) Mechanism process synchronization
5) Mechanism process communication

Memory Management

OS responsible tasks regards to memory management:
1) Keeping track parts of memory currently use, and which processes.
2) Determining which process (or part of) and data to move in and out of memory.
3) Allocating and deallocating memory space as needed.

Storage Management

Mass-Storage Management

OS responsible tasks regards mass-storage management:
Free space management
Storage allocation
Disk scheduling
Note tradeoffs regarding size, speed, longevity, security, and rewritability between different mass storage devices, including floppy disks, hard disks, tape drives, CDs, DVDs, etc.

FileSystem Management

OS responsible tasks regards to file-system management:
1) Creating and deleting files
2) Creating and deleting directories tp organize files
3) Supporting primitives for manipulating files and directories
4) Mapping files onto secondary storage.
5) Backing up files onto stable (nonvolatile) storage media.

Caching

- Many cases smaller higher-speed storage space serves as cache, temporary storage, for frequently needed portions of larger slower storage areas.
- Hierarchy of memory storage CPU registers to hard drives and external storage.
- OS responsible determining information store in what level of cache, and when transfer data one level to another.
- Proper choice cache management impact system performance.
- Data read from disk follows migration path from hard drive to main memory, to CPU cache, to registers before used, while data written follows reverse path. Each step (other than registers) fetch data than needed, cache excess for future requests faster.
- Writing, small amounts data frequently buffered until enough fill entire "block" on next output device in chain.
- Complicated when multiple processes access common data, important ensure every access reaches most up-to-date copy of cached data (amongst several copies in different cache levels.)

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Figure 1.11 - Performance of various levels of storage

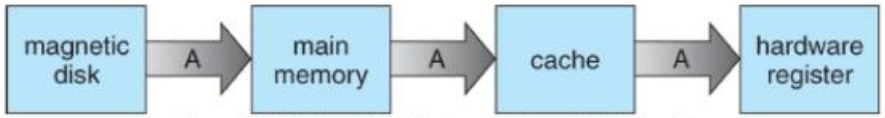


Figure 1.12 - Migration of integer A from disk to register

I/O Systems

I/O subsystem consists of several components:
1) Memory-management component includes buffering, caching, and spooling.
2) General device-driver interface.
3) Drivers for specific hardware devices.

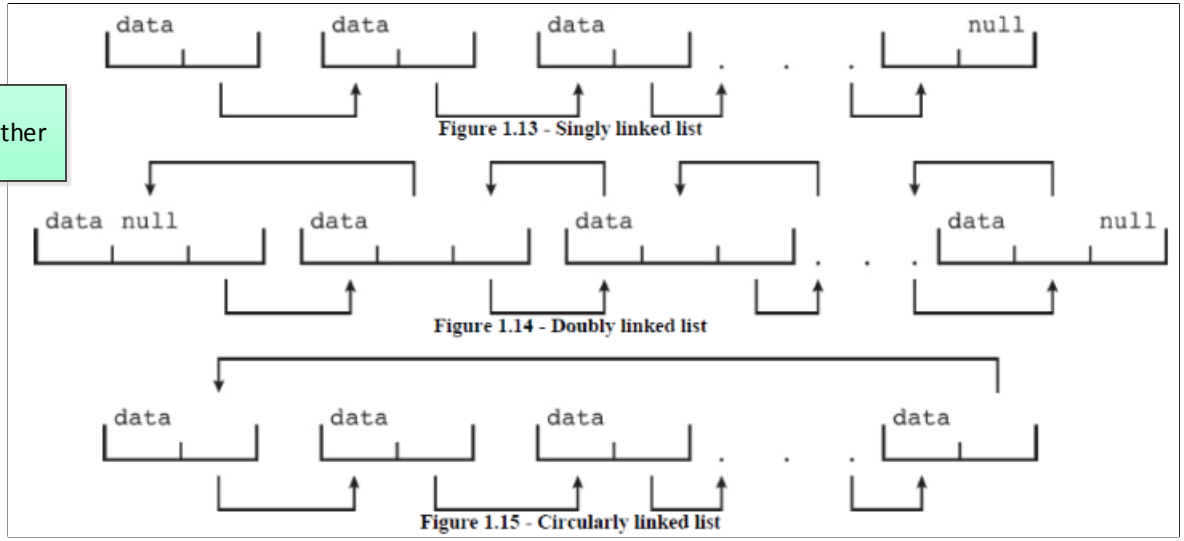
Protection and Security

Protection: ensuring no process access or interfere with resources to which not entitled, design or accident.
Security: protecting system deliberate attacks, legitimate users of system attempting to gain unauthorized access and privileges, or external attackers attempting to access or damage system.

Kernel Data Structures

Linked List

Data items linked to one another



Stack

Last in first out principle (LIFO)

Queue

Sequential order data structure
First in first out (FIFO)

Tree

Linked through parent-child relationship (binary tree have parent have two children, left and right child

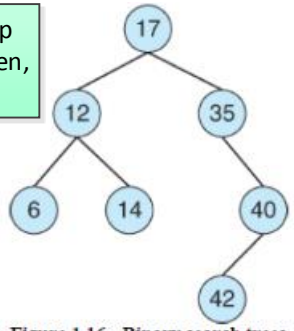
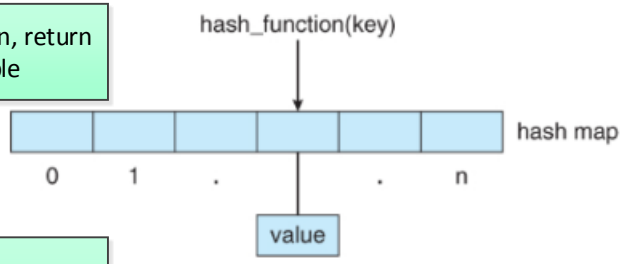


Figure 1.16 - Binary search trees

Hash Functions and Maps

Data as input, perform numeric operation, return numeric value used as index in table



Bitmaps

String binary digits represent status of items
0 0 1 0 1 1 1 0 1

Computing Environments

1) Traditional Computing
2) Mobile Computing
3) Distributed Systems
4) Client-Server Computing
5) Peer-to-Peer Computing
6) Virtualization
7) Cloud Computing
8) Real-Time Embedded Systems

1) Traditional Computing

PC connected to network, with servers providing file and print services

2) Mobile Computing

- Computing small handheld devices such as smart phones/tablets. (opposed to laptops traditional computing)
- May take advantage of additional built-in sensors, GPS, tilt, compass, inertial movement.
- Connect to Internet using wireless networking (IEEE 802.11) or cellular telephone technology.
- Limited storage capacity, memory capacity, and computing power relative to PC.
- Slower processors, consume less battery power and produce less heat.
- Two dominant OSes today are Google Android and Apple iOS.

3) Distributed Systems

- Multiple, possibly heterogeneous, computers connected together via network, cooperating some way, form, or fashion.
- Networks range small tight LANs to broad reaching WANs.
  WAN = Wide Area Network, such as an international corporation
  MAN =Metropolitan Area Network, covering a region the size of a city for example.
  LAN =Local Area Network, typical of a home, business, singlesite corporation, or university campus.
  PAN = Personal Area Network, such as the bluetooth connection between your PC, phone, headset, car, etc.
- Network access speeds, throughputs, reliabilities, important issues.
- OS view network range from just special form of file access to complex well-coordinated network operating systems.
- Shared resources include files, CPU cycles, RAM, printers, and other resources.

4) Client-Server Computing

- Server provides services (HW or SW) to other systems as clients. (clients and servers processes, not HW, and may coexist on same computer.
- Process may act as both client and server of either same or different resources.
- Served resources include disk space, CPU cycles, time of day, IP name information, graphical displays (X Servers), other

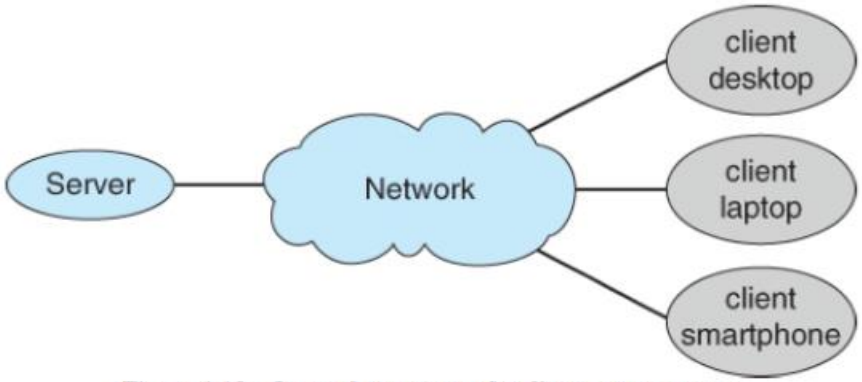


Figure 1.18 - General structure of a client-server system

5) Peer-to-Peer Computing

- Any computer or process on network provide services to any other which requests it. No clear "leader" or overall organization.
- May employ central "directory" server looking up location of resources, or may use peer-to-peer searching find resources.
- E.g. Skype uses central server locate desired peer, further communication is peer to peer.

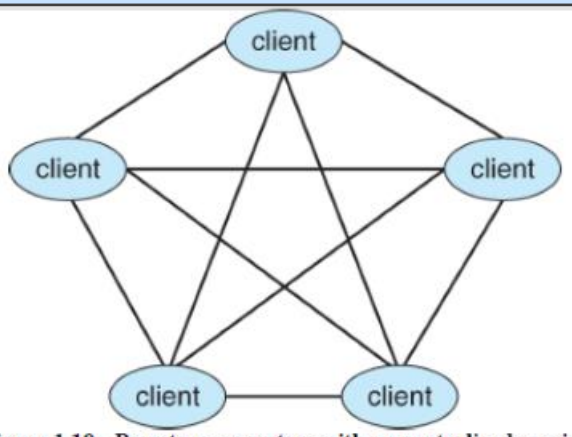


Figure 1.19 - Peer-to-peer system with no centralized service

6) Virtualization

- Allows one / more "guest" OS run on virtual machines hosted single physical machine and virtual machine manager.
- Useful for cross-platform development and support.
- Example, UNIX on virtual machine, hosted by virtual machine manager on Windows computer. Full root access to virtual machine, if crashed, underlying Windows machine unaffected.
- System calls caught by VMM and translated into (different) system calls made to underlying OS.
- Virtualization slow down program run through VMM, can also speed up things if virtual hardware accessed through cache instead of physical device.
- Programs can also run simultaneously on native OS, bypassing virtual machines.

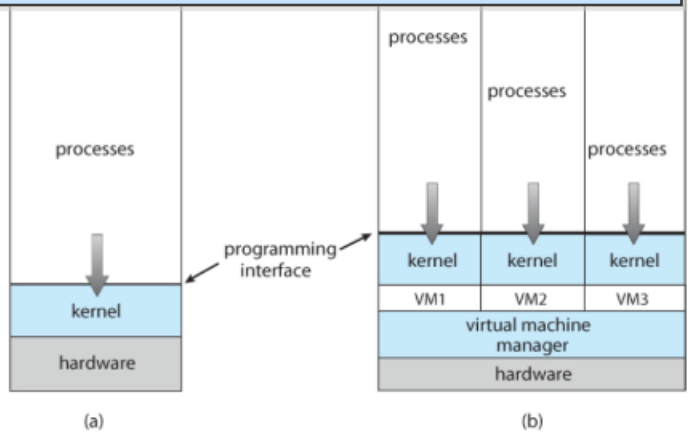
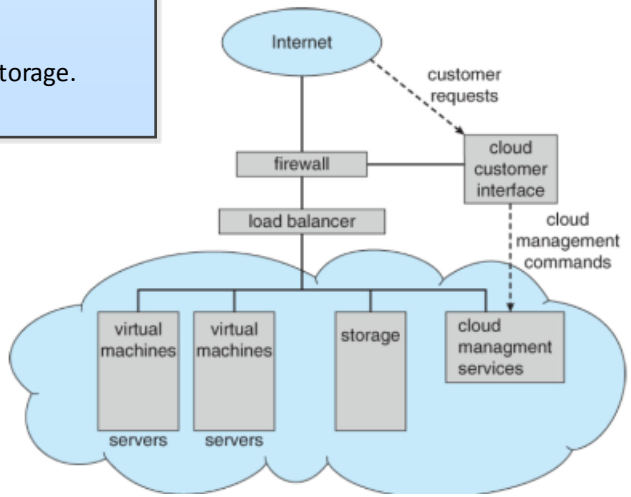


Figure 1.20 - VMWare

7) Cloud Computing

- Deliver storage, applications as service over a network.
- Types of cloud computing:
  - > Public cloud = anyone willing pay for service.
  - > Private cloud = Run company for internal use only.
  - > Hybrid cloud = Cloud with both public and private components.
  - > Software as a Service – SaaS = Applications (word processors) available via Internet
  - > Platform as a Service – PaaS = Software stack available for application use, (database server)
  - > Infrastructure as a Service – IaaS = Servers or storage available on Internet, backup servers, photo storage, file storage.
  - > Service providers may provide more than one type of service
- Clouds contain thousands physical computers, millions virtual ones, petabytes total storage.
- Web hosting services offer (one or more) virtual machine(s) to each client.



8) Real-Time Embedded Systems

- Embedded into devices (automobiles, climate control systems, process control)
- Involve specialized chips, generic CPUs applied to particular task.
- Process control devices require real-time (interrupt driven) OSes. Response time critical

OpenSource Operating Systems

- Open-Source software published (sold) with source code, can see and optionally modify the code.
- Open-source SW developed and maintained by small army of loosely connected often unpaid programmers, each working towards common good.
- Critics argue open-source SW is buggy, proponents counter bugs found and fixed quickly, many eyes inspecting code.
- Open-source operating systems good resource studying OS development, students examine source code and change it

Linux

- Developed by Linus Torvalds Finland 1991 as first full operating system developed GNU.
- Different distributions Linux evolved from Linus's original, including RedHat, SUSE, Fedora, Debian, Slackware, Ubuntu, each geared toward different group end-users and operating environments.
- Linux on Windows system using VMware, steps:
    1. Download free "VMware Player" tool from http://www.vmware.com/download/player
    2. Choose Linux version virtual machine images at http://www.vmware.com/appliances
    3. Boot the virtual machine within VMware Player.

BSD UNIX

- Originally developed ATT Bell labs, source code available to computer science students.
- UCB students developed UNIX further, released product as BSD UNIX binary and source-code format.
- BSD UNIX not open-source, license needed from ATT.
- Several versions of BSD UNIX, including Free BSD, NetBSD, OpenBSD, and DragonflyBSD
- Source code located in /usr/src.
- Core of Mac operating system is Darwin, derived from BSD UNIX, available at http://developer.apple.com/opensource/index.html

Solaris

- UNIX operating system computers from Sun Microsystems.
- Based on BSD UNIX, migrated to ATT SystemV as basis.
- Parts of Solaris opensource, some are not because covered by ATT copyrights.
- Can change open-source components of Solaris, recompile, and link in with binary libraries of copyrighted portions of Solaris.
- Open Solaris from http://www.opensolaris.org/os/
- Solaris allows viewing of source code online, without download and unpack entire package.

Utility

- Free software movement gaining popularity, thousands ongoing projects involving untold numbers programmers.
- Sites such as http://freshmeat.net/ and http://distrowatch.com/ provide portals to many of these projects.

#2 System Structures

- OS provides environment execution of programs
- Provides certain services to programs and users
- Services differ between OS, can identify common classes
- Services make programming task easier

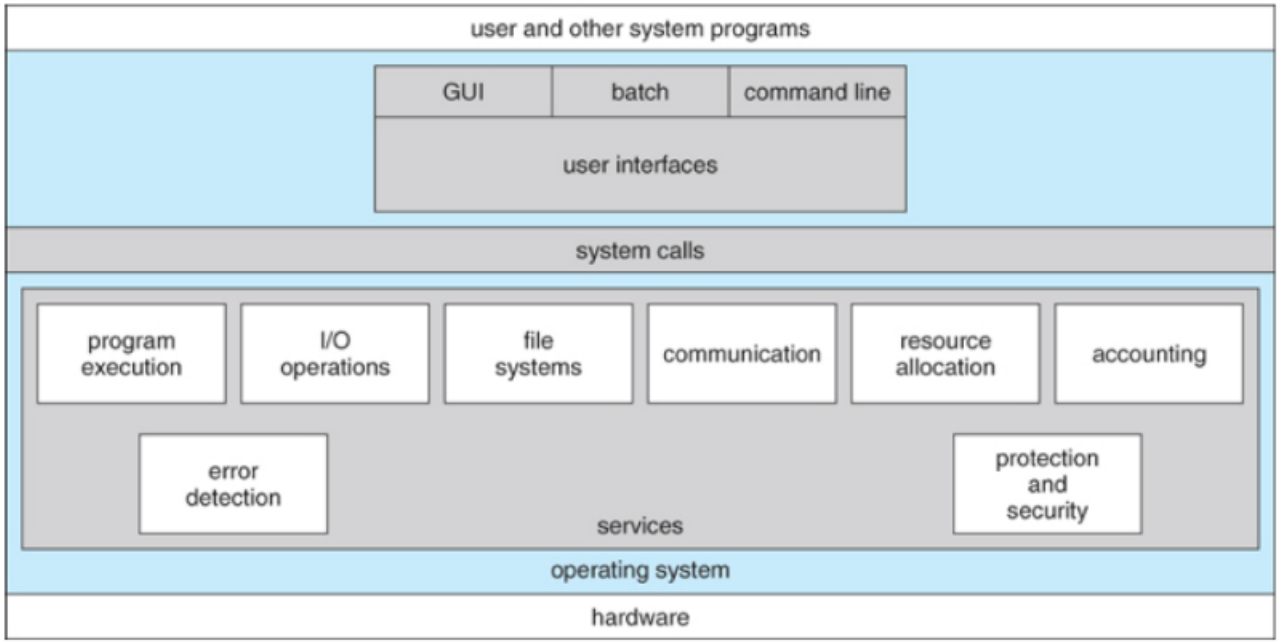
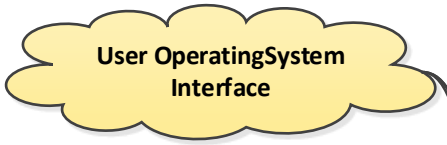


Figure 2.1 - A view of operating system services

- **User Interfaces:** users issue commands to system. May be command-line interface (e.g. sh, csh, ksh, tcsh, etc.), a GUI interface (e.g. Windows, XWindows, KDE, Gnome, etc.), or batch command systems.
- **Program Execution:** OS must be load program into RAM, run program, and terminate program, normally or abnormally.
- **I/O Operations:** OS responsible transferring data to and from I/O devices
- **File-System Manipulation:** Addition raw data storage, OS responsible maintaining directory and subdirectory structures, mapping file names to specific blocks of data storage, providing tools for navigating and utilizing file system.
- **Communications:** Inter-process communications, IPC, between processes running on same processor, or between processes running on separate processors or separate machines. May be implemented as shared memory or message passing, (or some systems may offer both.)
- **Error Detection:** Hardware and software errors detected and handled appropriately, minimum harmful repercussions. May include complex error avoidance or recovery systems, including backups, RAID drives, redundant systems. Debugging and diagnostic tools aid users and administrators in tracing down the cause of problems.

Systems aid in efficient operation of OS itself:

- **Resource Allocation:** E.g. CPU cycles, main memory, storage space, and peripheral devices. Resources managed generic systems and others carefully designed and specially tuned systems, customized for particular resource and operating environment.
- **Accounting Keeping:** track system activity and resource usage, billing purposes or statistical record keeping used optimize performance.
- **Protection and Security:** Preventing harm to system and resources, internal processes or malicious outsiders. Authentication, ownership, restricted access. Highly secure systems log process activity to excruciating detail, security regulation dictate storage of records on permanent medium for extended times in secure (offsite) facilities.



Command Interpreter

- Gets and processes user request, launches requested programs.
- CI may be incorporated directly into kernel.
- CI separate program launches when user logs in or accesses system.
- UNIX provides user with choice different shells, either configured launch automatically at login, or changed on the fly. (Each uses different configuration file initial settings and commands executed upon startup.)
- Different shells provide different functionality, certain commands implemented directly by shell without external programs. Least rudimentary command interpretation structure in shell script programming (loops, decision constructs, variables, etc.)
- Processing of wild card file naming and I/O redirection UNIX details handled by shell, program launched sees only list filenames generated by shell from wild cards. DOS system, wild cards passed to programs, can interpret wild cards as program sees fit.

Graphical User Interface, GUI

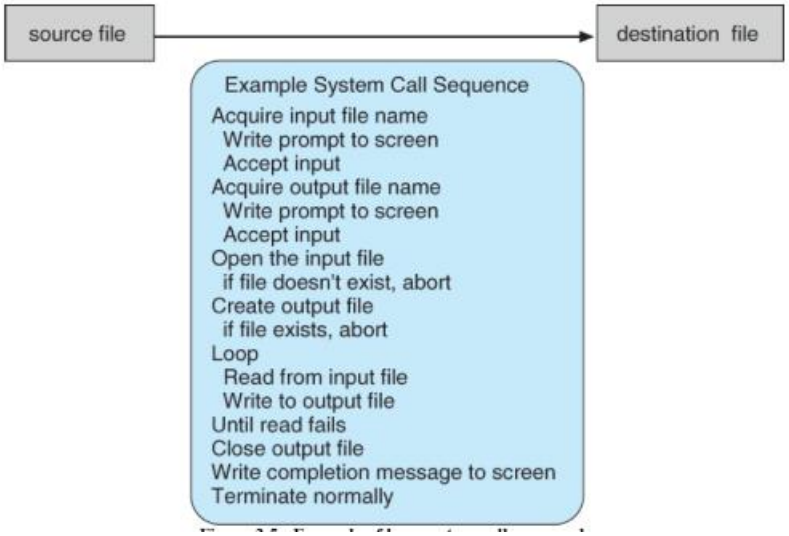
- implemented as desktop metaphor, file folders, trash cans, resource icons.
- Icons represent item on system, respond accordingly when icon activated.
- First developed 1970's Xerox PARC research facility.
- Some systems GUI is front end activating traditional command line interpreter running in background. Others GUI true graphical shell.
- Mac traditionally provided ONLY GUI interface. Advent of OSX (based partially UNIX), command line interface available.
- Mice and keyboards impractical for small mobile devices, use touchscreen interface, responds patterns swipes or "gestures".

Choice of interface

- Modern systems allow individual users select desired interface, customize operation, switch between different interfaces as needed. System administrators determine interface user starts with when they first log in.
- GUI provide option terminal emulator window for entering command-line commands.
- Command-line commands also entered shell scripts, run like programs.

System Calls

- System calls provide user or application programs call services of operating system.
- C or C++, some written assembly for optimal performance.
- Figure illustrates the sequence of system calls required to copy a file:



- Use "strace" to see examples system calls invoked by single simple command. (strace mkdir temp, strace cd temp, strace date > t.t, strace cp t.t t.2, etc.)
- Most programmers not use low-level system calls directly, use "Application Programming Interface", API

Use APIs instead of direct system calls provides greater program portability between different systems. API makes appropriate system calls through system call interface, using table lookup access specific numbered system calls

EXAMPLE OF STANDARD API

As an example of a standard API, consider the read( ) function that is available in UNIX and Linux systems. The API for this function is obtained from the man page by invoking the command

man read

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t    read(int fd, void *buf, size_t count)
```

return value — function name — parameters

A program that uses the read( ) function must include the unistd.h header file, as this file defines the ssize_t and size_t data types (among other things.) The parameters passed to read( ) are as follows:

- int fd - The file descriptor to be read
- void *buf - A buffer where the data will be read into
- size_t count - The maximum number of bytes to be read into the buffer.

On a successful read, the number ofbytes read is returned. A return value of 0 indicates end of file. If an error occurs, read( ) returns -1.

Parameters generally passed to system calls via registers, or by values pushed onto stack.
Large blocks data accessed indirectly, through memory address passed in register or on stack, as shown in Figure 2.7:

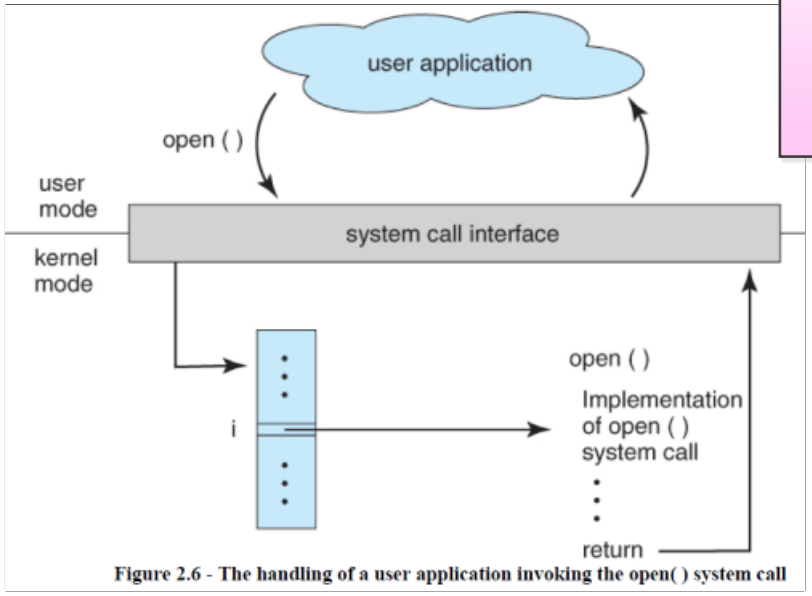


Figure 2.6 - The handling of a user application invoking the open( ) system call

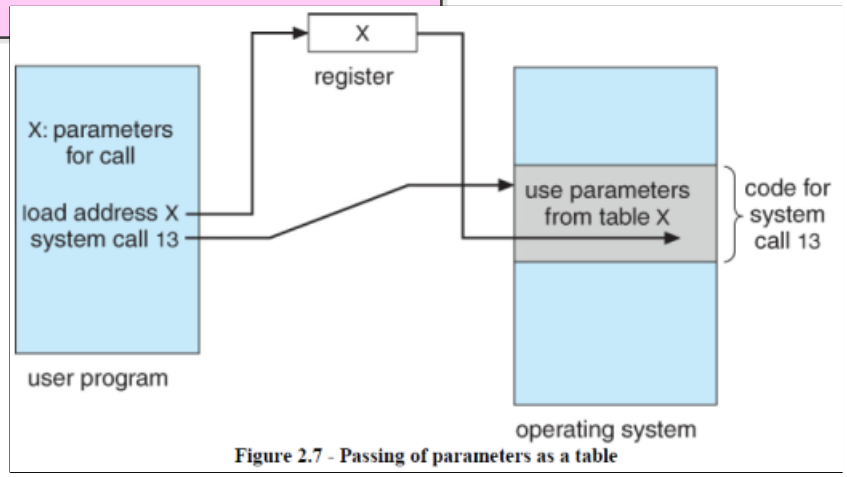


Figure 2.7 - Passing of parameters as a table

Types of System Calls

Six major categories:

1) Process Control
2) File manipulation
3) Device manipulation
4) Information maintenance
5) Communications
6) Protection

Process control:
> end, abort
> load, execute
> create process, terminate process
> get process attributes, set process attributes
> wait for time
> wait event, signal event
> allocate and free memory

File management:
> create file, delete file
> open, close
> read, write, reposition
> get file attributes, set file attributes

Device management:
> request device, release device
> read, write, reposition
> get device attributes, set device attributes
> logically attached or detach devices

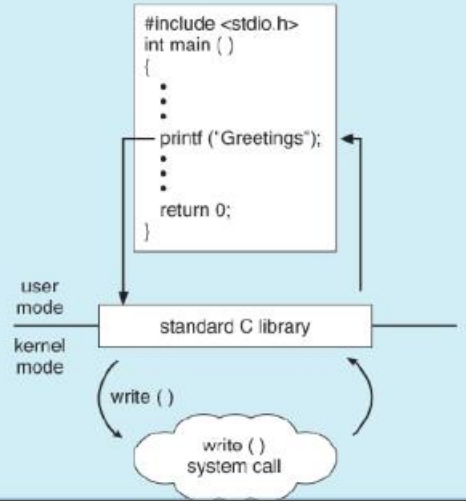
Information maintenance:
> get time or date, set time or date
> get system data, set system data
> get process, file, or device attributes
> set process, file, or device attributes

Communications:
> create, delete communication connection
> send, receive messages
> transfer status information
> attach or detach remote devices

	WINDOWS	UNIX
Process Control	`CreateProcess()`	`fork()`
	`ExitProcess()`	`exit()`
	`waitForSingleObject()`	`wait()`
File Manipulation	`CreateFile()`	`open()`
	`ReadFile()`	`read()`
	`WriteFile()`	`write()`
	`CloseHandle()`	`close()`
Device Manipulation	`SetConsoleMode()`	`ioctl()`
	`ReadConsole()`	`read()`
	`WriteConsole()`	`write()`
Information Maintenance	`GetCurrentProcessID()`	`getpid()`
	`SetTimer()`	`alarm()`
	`Sleep()`	`sleep()`
Communication	`CreatePipe()`	`pipe()`
	`CreateFileMapping()`	`shm.open()`
	`MapViewOfFile()`	`mmap()`
Protection	`SetFileSecurity()`	`chmod()`
	`InitializeSecurityDesciptor()`	`unmask()`
	`SetSecurityDesciptorGroup()`	`chown()`

EXAMPLE OF STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. For example, let's assume a C program invokes the printf( ) statement. The C library intercepts this call and invokes the necessary ssystem call(s) in the operating system - in this instance, the write( ) system call. The C library takes the value returned by write( ) and passes it back to the user program. This is shown below:



1) Process Control

- Process control system calls (end, abort, load, execute, create process, terminate process, get/set process attributes, wait for time or event, signal event, and allocate and free memory.
- Processes created, launched, monitored, paused, resumed, and eventually stopped.
- One process pauses or stops, another must be launched or resumed
- When processes stop abnormally it may be necessary to provide core dumps and/or other diagnostic or recovery tools.
- Compare DOS (single-tasking system) with UNIX (multi-tasking system).
  - > When process launched in DOS, command interpreter first unloads as much of itself as it can to free up memory, loads process and transfers control to it. Interpreter does not resume until process completed
  - > UNIX multitasking system, command interpreter resident executing process. User switch back command interpreter, place running process in background. Command interpreter first executes "fork" system call, creates second process exact duplicate (clone) of original command interpreter. Original process as parent, cloned process as child, own unique process ID and parent ID. Child process executes "exec" system call, replaces its code with desired process. Parent (command interpreter) waits child complete before issuing new command prompt, also issue new prompt right away, without waiting for child process to complete. (child running "in background")

2) File Management

- Include create file, delete file, open, close, read, write, reposition, get file attributes, an set file attributes.
- Operations supported directories as well as ordinary files.
- (Actual directory structure may be implemented using ordinary files on file system, or through other means.

3) Device Management

- I include request device, release device, read, write, reposition, get/set device attributes, and logically attach or detach devices.
- Devices may be physical (e.g. disk drives), or virtual / abstract (e.g. files, partitions, and RAM disks).
- Mays represent devices as special files in file system, accessing "file" calls upon appropriate device drivers in OS.

4) Information Maintenance

- Information maintenance system calls include calls to get/set time, date, system data, and process, file, or device attributes.
- May also provide ability dump memory at any time, single step programs pausing execution after each instruction, tracing operation of programs, all of which can help to debug programs.

5) Communication

- Calls create/delete communication connection, send/receive messages, transfer status information, attach/detach remote devices.
- Message passing model must support calls to:
  - > Identify a remote process and/or host with which to communicate.
  - > Establish a connection between the two processes.
  - > Open and close the connection as needed.
  - > Transmit messages along the connection.
  - > Wait for incoming messages, in either a blocking or non-blocking state.
  - > Delete the connection when no longer needed.
- The shared memory model must support calls to:
  - > Create and access memory that is shared amongst processes (and threads.)
  - > Provide locking mechanisms restricting simultaneous access.
  - > Free up shared memory and/or dynamically allocate it as needed.
- Message passing simpler and easier, (for inter-computer communications), for small amounts data.
- Shared memory is faster, better approach large amounts data shared, (when processes reading data rather than writing it, or only one or a small number of processes need to change any given data item.)

6) Protection

- Protection provides mechanisms controlling users / processes access to system resources.
- System calls allow access mechanisms adjusted, non-privileged users grant elevated access permissions controlled circumstances.
- Once only of concern multiuser systems, protection important on all systems, in the age of ubiquitous network connectivity.

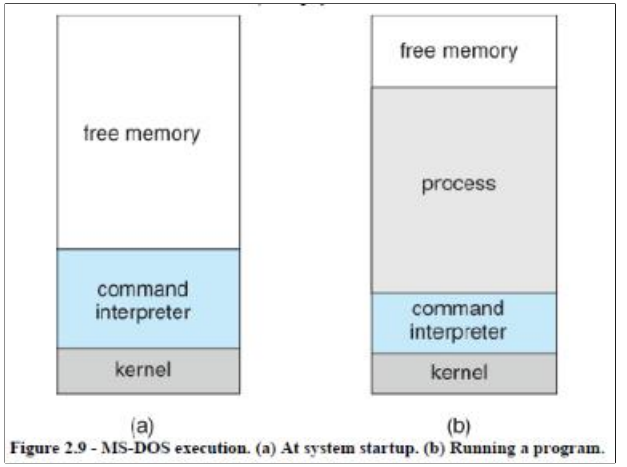


Figure 2.9 - MS-DOS execution. (a) At system startup. (b) Running a program.

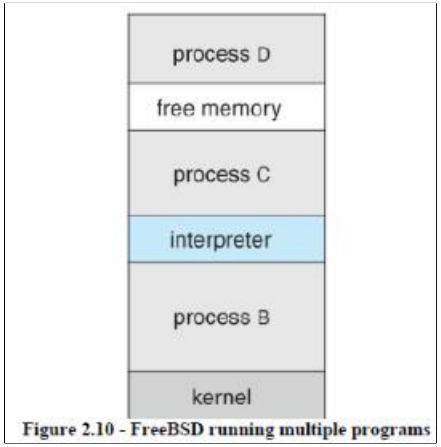


Figure 2.10 - FreeBSD running multiple programs

System Programs

- Provide OS functionality separate applications, not part of kernel or command interpreters. (system utilities or system applications.
- Also with useful applications such as calculators and editors, (e.g. Notepad). Debate border system and non-system applications.
- Categories:
  - > **File management programs:** create, delete, copy, rename, print, list, and generally manipulate files and directories.
  - > **Status information Utilities:** check date, time, number of users, processes running, data logging, etc. System registries used store and recall configuration information for particular applications.
  - > **File modification:** e. g. text editors and other tools change file contents.
  - > **Programming-language support:** E.g. Compilers, linkers, debuggers, profilers, assemblers, library archive management, interpreters for common languages, and support for make.
  - > **Program loading and execution:** loaders, dynamic loaders, overlay loaders, etc., as well as interactive debuggers.
  - > **Communications:** Programs providing connectivity between processes and users, mail, web browsers, remote logins, file transfers, and remote command execution.
  - > **Background services:** System daemons started when system booted, and run for as long as the system running, handling necessary services. Examples include network daemons, print servers, process schedulers, system error monitoring services.
- Also come complete with application programs provide additional services, copying files checking time and date.
- Views of system determined by command interpreter and application programs. Never make system calls, even through the API, (with exception of simple (file) I/O in user-written programs.)

OperatingSystem Design and Implementation

1) Design Goals

- Requirements define properties finished system must have, necessary first step in designing any large complex system.
  - > ***User requirements***: features users care and understand. Do not include implementation details.
  - > ***System requirements:*** written for developers, include details about implementation specifics, performance requirements, compatibility constraints, standards compliance, etc. Serve as "contract" between customer and developers
- Requirements operating systems can vary depending planned scope and usage of system. (Single user / multiuser, specialized system / general purpose, high/low security, performance needs, operating environment, etc.)

2) Mechanisms and Policies

- Policies determine what is to be done. Mechanisms determine how it is to be implemented.
- If properly separated and implemented, policy changes easily adjusted without rewriting code, adjusting parameters or possibly loading new data / configuration files. For example relative priority background versus foreground tasks.

3) Implementation

- Traditionally OSes written assembly language. Direct control hardware-related issues, inextricably tied OS to particular HW platform.
- Advances compiler efficiencies modern OSes written in C, or C++. Critical sections code still written assembly language
- Operating systems developed using emulators of target hardware, if real hardware unavailable, or not suitable platform development

Operating-System Structure

Efficient performance implementation OS partitioned into separate subsystems, each defined tasks, inputs, outputs, and performance characteristics. Subsystems arranged various architectural configurations:

1) Simple Structure

DOS written developers no idea how big and important become. Written by few programmers relatively short time, without benefit of modern software engineering techniques, gradually exceed original expectations. Does not break system into subsystems, has no distinction between user and kernel modes, allowing programs direct access to underlying hardware.

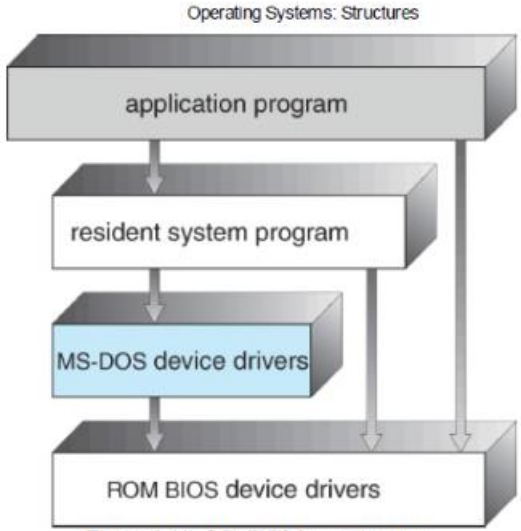


Figure 2.11 - MS-DOS layer structure

2) Layered Approach

- Break OS into number smaller layers, each rests on layer below it, relies solely on services provided by next lower layer.
- Approach allows each layer developed and debugged independently, assumption all lower layers debugged and trusted deliver proper services.
- Problem deciding order place layers, no layer can call services of higher layer
- Less efficient, request service higher layer filter all lower layers before reaches HW, significant processing each step.

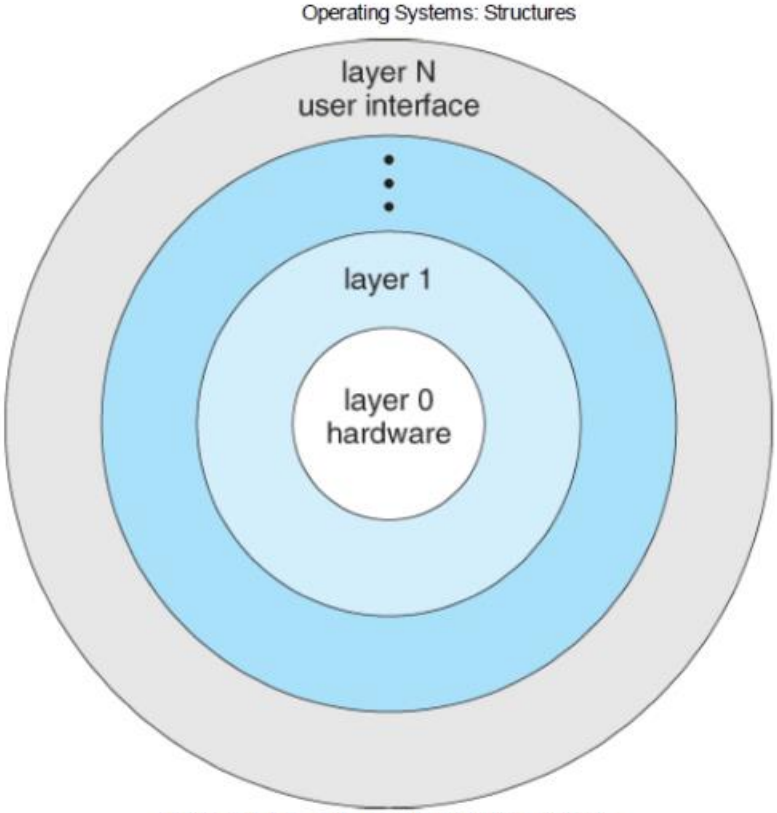


Figure 2.13 - A layered operating system

original UNIX OS used simple layered approach, almost all OS one big layer, not really breaking OS down into layered subsystems:

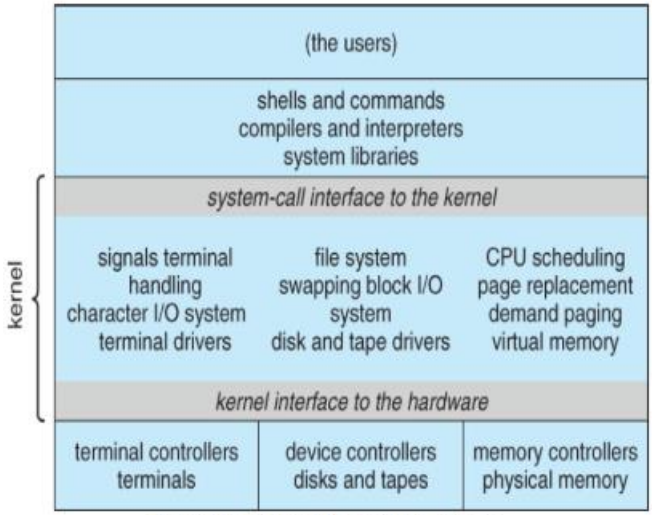


Figure 2.12 - Traditional UNIX system structure

3) Microkernels

- Remove nonessential services from kernel, implement as system applications, making kernel small and efficient as possible.
- Basic process and memory management, message passing between services.
- Security and protection enhanced, services performed in user mode, not kernel mode.
- System expansion easier, only adding more system applications, not rebuilding new kernel.
- Mach first and most widely known microkernel, major component of Mac OSX.
- Windows NT originally microkernel, performance problems relative to Windows 95. NT 4.0
- improved performance moving more services into kernel, XP more monolithic.
- QNX, a real-time OS for embedded systems.

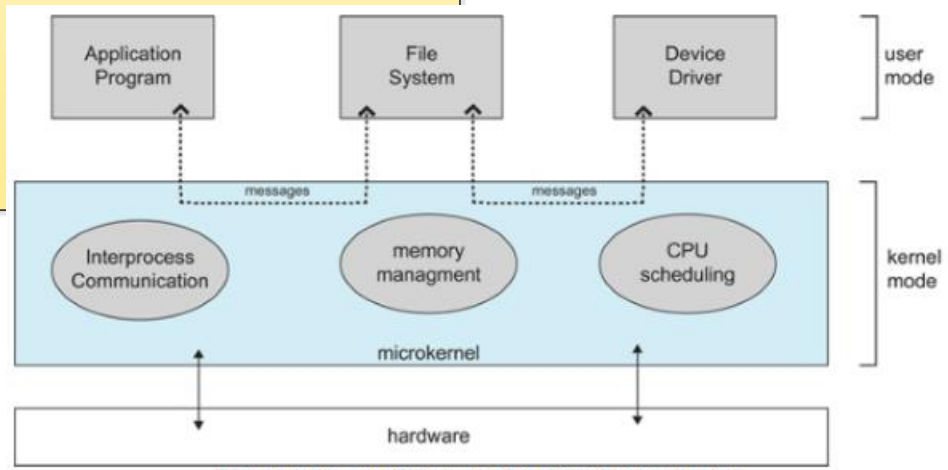


Figure 2.14 - Architecture of a typical microkernel

4) Modules

- Modern OS development is object-oriented, small core kernel and modules linked dynamically.
- Modules similar to layers, each subsystem clearly defined tasks and interfaces, module free contact any other module, eliminating the problems of going through multiple intermediary layers
- The kernel relatively small architecture, similar microkernels, kernel does not have to implement message passing since modules are free to contact each other directly.

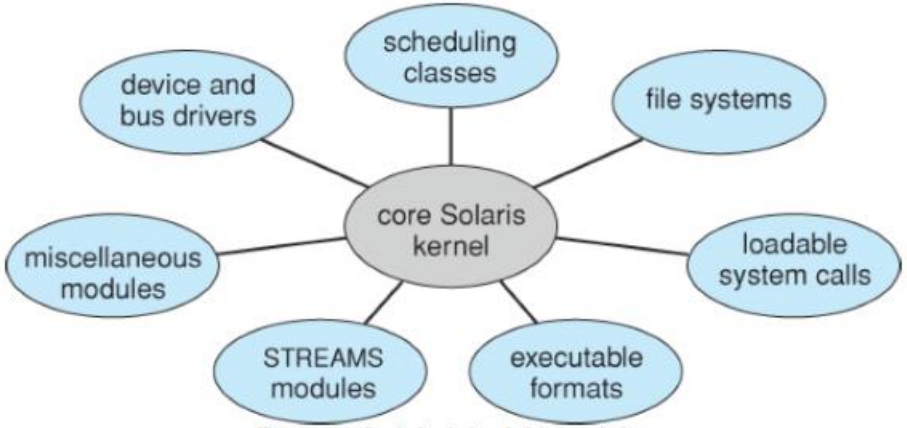


Figure 2.15 - Solaris loadable modules

5) Hybrid Systems
Most OSes today do not strictly adhere to one architecture, but are hybrids of several.

Mac OS X
Architecture relies on Mach microkernel for basic system management services, BSD kernel for additional services. Application services and dynamically loadable modules (kernel extensions) provide the rest of the OS functionality

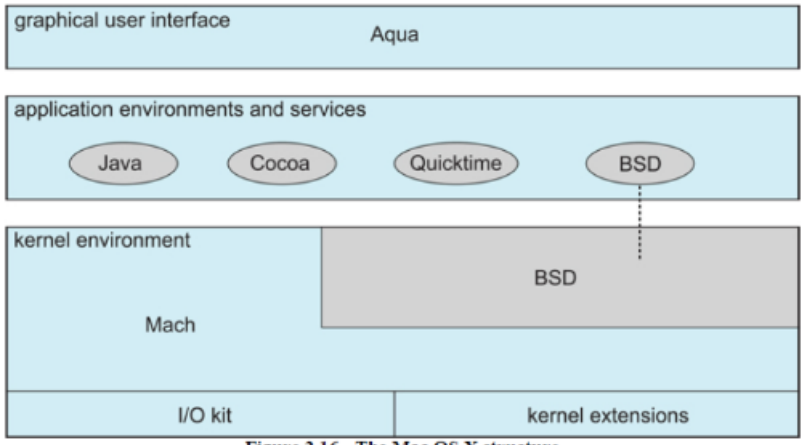


Figure 2.16 - The Mac OS X structure

iOS
Developed by Apple for iPhones and iPads. Less memory and computing power needs than Max OS X, supports touchscreen interface and graphics for small screens

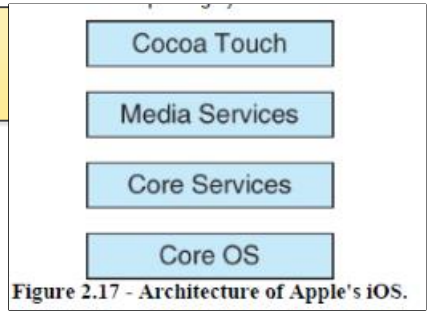


Figure 2.17 - Architecture of Apple's iOS.

Android
Developed for Android smartphones and tablets by Open Handset Alliance, primarily Google.
Open-source OS, lead to its popularity.
Includes versions of Linux and Java virtual machine both optimized for small platforms.
Developed using special Java-for-Android development environment.

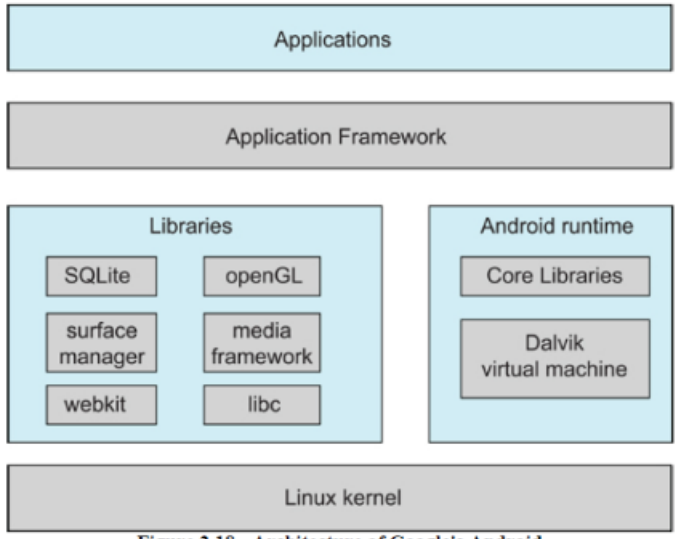


Figure 2.18 - Architecture of Google's Android

Debugging includes both error discovery and elimination and performance tuning.

Kernighan's Law
"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

Operating-System Debugging

1) Failure Analysis

- Debuggers allow processes executed stepwise, provide examination of variables and expressions execution progresses.
- Can document program execution, produce statistics how much time spent on sections or even lines of code.
- Ordinary process crashes, memory dump of state of process's memory at time of crash saved for later analysis.
  > Program specially compiled to include debugging information, slow down performance.
- Approaches don't work well for OS code:
  > Performance hit caused adding debugging code unacceptable.
  > Parts of OS run kernel mode, direct access to hardware.
  > Error during kernel's file-access or direct disk-access routines, not practical write crash dump into file.
  # Instead the kernel crash dump saved to unallocated portion of disk reserved for that purpose.

2) Performance Tuning

- Requires monitoring system performance.
- System record important events into log files, analyzed by tools. Traces evaluate proposed system perform under same workload.
- Provide utilities report system status upon demand, unix "top" command. (w, uptime, ps, etc.)
- System utilities may provide monitoring support.

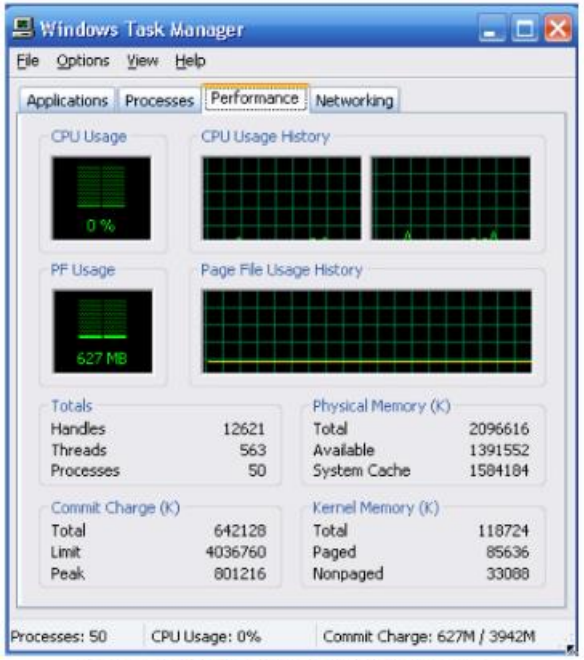


Figure 2.19 - The Windows task manager.

3) DTrace

- Special facility for tracing running OS, developed for Solaris 10.
- Adds "probes" directly into OS code, queried by "probe consumers".
- Probes removed when not use, DTrace facility zero impact on system when not used, proportional impact in use.
- Consider, for example, the trace of an ioctl system call
- Probe code restricted to be "safe", (no loops allowed), use minimum system resources.
- When probe fires, enabling control blocks, ECBs, performed, each having structure of if-then block
- When terminates, ECBs associated with consumer removed. When no more ECBs remain interested in particular probe, probe also removed
- Example, following D code monitors CPU time of each process running with user ID of 101.

```
sched:::oncpu
uid == 101
{
        self->ts = timestamp;
}
sched:::off-cpu
self->ts
{
        @time[execname] = sum( timestamp - self->ts );
        self->ts = 0;
}
```

- DTrace is restricted, due to direct access to (and ability to change) critical kernel data structures.
- DTrace is opensource, being adopted by several UNIX distributions.

```
# ./all.d 'pgrep xclock' XEventsQueued
dtrace: script './all.d' matched 52377 probe
CPU FUNCTION
  0 -> XEventsQueued                       U
  0   -> _XEventsQueued                    U
  0     -> _X11TransBytesReadable          U
  0     <- _X11TransBytesReadable          U
  0     -> _X11TransSocketBytesReadable    U
  0     <- _X11TransSocketBytesreadable    U
  0     -> ioctl                           U
  0       -> ioctl                         K
  0         -> getf                        K
  0           -> set_active_fd             K
  0           <- set_active_fd             K
  0         <- getf                        K
  0         -> get_udatamodel              K
  0         <- get_udatamodel              K
...
  0         -> releasef                    K
  0           -> clear_active_fd           K
  0           <- clear_active_fd           K
  0           -> cv_broadcast              K
  0           <- cv_broadcast              K
  0         <- releasef                    K
  0       <- ioctl                         K
  0     <- ioctl                           U
  0   <- _XEventsQueued                    U
  0 <- XEventsQueued                       U
```

OperatingSystem Generation

- Oses designed/built specific HW configuration specific site, designed with number variable parameters and components, configured particular operating environment.
- Systems need reconfigured after initial installation, add additional resources, capabilities, or tune performance, logging, or security.
- Information needed configure OS include:
  - > What CPU(s) installed on system, optional characteristics does each have?
  - > RAM installed? (determined automatically, either at install or boot time.)
  - > Devices present? OS determine which device drivers include, as well as some device-specific characteristics and parameters.
  - > OS options desired, values set for particular OS parameters. Include size of open file table, number buffers use, process scheduling (priority), disk scheduling algorithms, number of slots in process table, etc.
- OS source code can be edited, recompiled, and linked into new kernel.
- Configuration tables determine which modules link into new kernel, what values to set for some key important parameters. Require configuration of complicated make-files, done automatically or through interactive configuration programs; Make used to generate new kernel specified by new parameters.
- System configuration may defined by table data, "rebuilding" of system requires editing data tables.
- Once system regenerated, required to reboot system to activate new kernel. Possibilities errors, systems provide mechanism for booting to older or alternate kernels.

System Boot

- System powers up, interrupt generated loads memory address into program counter, system begins executing instructions found at address. Address points to "bootstrap" program located in ROM chips (or EPROM chips) on the motherboard.
- ROM bootstrap program first runs hardware checks, determining physical resources present and doing power-on self tests (POST) of all HW.
  - > Some devices, such as controller cards may have own onboard diagnostics, called by the ROM bootstrap program.
- User has option pressing special key during POST process, launch ROM BIOS configuration utility. Allows specify and configure certain hardware parameters as where to look for OS and whether or not restrict access to utility with password.
  - > Some hardware provide access to additional configuration setup programs, RAID disk controller or some special graphics or networking cards.
- Assuming utility invoked, bootstrap program looks for nonvolatile storage device containing OS. May look for floppy drive, CD ROM, primary or secondary hard drives.
- Find first sector on hard drive and load up ***fdisk*** table, contains information about how physical drive divided logical partitions, where each partition starts and ends, which partition "active" partition used for booting system.
- Also small amount system code in portion of first disk block not occupied by ***fdisk*** table. Bootstrap code is first step not built into hardware, i.e. first part which might be in any way OS-specific.
- Generally code knows just enough access hard drive, load and execute a (slightly) larger boot program.
- For a single-boot system, boot program loaded off of hard disk proceed locate kernel on hard drive, load into memory, transfer control over to kernel. Opportunity to specify particular kernel to be loaded, may be useful if new kernel generated and doesn't work, or multiple kernels available different configurations.
- For dual-boot or multiple-boot systems, boot program give user opportunity specify particular OS to load, default choice if user not pick within given time. Boot program finds boot loader for chosen single-boot OS, runs program as described in previous bullet point.
- Once kernel running, may give user opportunity enter into single-user mode (maintenance mode). Launches very few services, does not enable logins other than primary log in on console. This mode used primarily for system maintenance and diagnostics.
- When system enters full multiuser multitasking mode, examines configuration files determine system services to start and launches each. Spawns login programs (gettys) on each login devices configured to enable user logins.

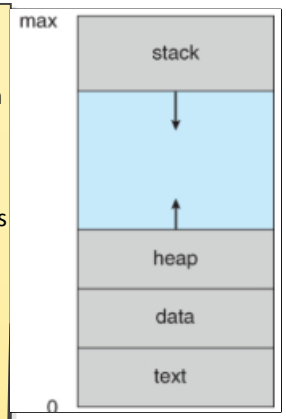
#3 Process Concept

- Process is instance of program in execution.
- Batch systems work in terms of "jobs". Modern process concepts still expressed in terms of jobs, (e.g. job scheduling), two terms often used interchangeably

The Process

- Process memory is divided four sections:
  - > Text section comprises compiled program code, read in from nonvolatile storage when the program is launched.
  - > Data section stores global and static variables, allocated and initialized prior to executing main.
  - > Heap used for dynamic memory allocation, managed via calls to new, delete, malloc, free, etc.
  - > Stack used for local variables. Space on stack reserved for local variables when declared (function entrance or elsewhere, depending language), space freed up when variables go out of scope. Stack also used for function return values, mechanisms of stack management language specific.

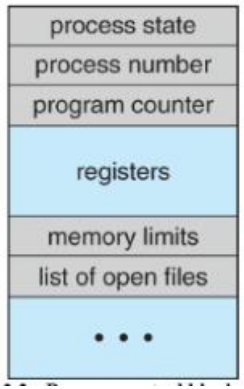
  Note stack and heap start opposite ends of process's free space and grow towards each other. If ever meet, stack overflow error occur, or call to new or malloc fail due to insufficient memory available.
- When processes swapped out of memory and later restored, additional information also stored and restored. Also program counter and value of all program registers.



Process Control Block

Each process has Process Control Block, PCB, stores following process-specific information:

- **Process State:** Running, waiting, etc., as discussed above.
- **Process ID:** and parent process ID.
- **CPU registers and Program Counter:** Need saved and restored when swapping processes in and out of CPU.
- **CPU-Scheduling information:** Priority information and pointers to scheduling queues.
- **Memory-Management information:** page tables or segment tables.
- **Accounting information:** user and kernel CPU time consumed, account numbers, limits, etc.
- **I/O Status information:** Devices allocated, open file tables, etc.



Process State

1) **New:** Process is in stage of being created
2) **Ready:** Process has all resources available to run, CPU not currently working on this process's instructions.
3) **Running:** CPU working on process's instructions.
4) **Waiting:** Process cannot run at moment, waiting for resource become available or some event to occur.
5) **Terminated:** Process completed.

- *Load average reported by the "w" command indicate average number of processes in "Ready" state over last 1, 5, and 15 minutes, i.e. processes who have everything they need to run but cannot because CPU is busy doing something else.*

Threads

- Modern systems allow single process have multiple threads of execution, which execute concurrently.

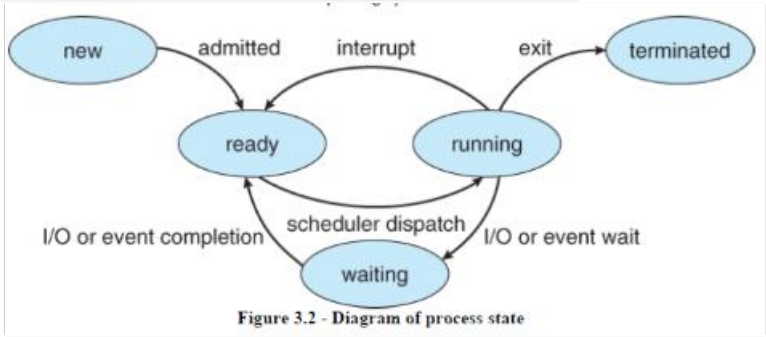


Figure 3.2 - Diagram of process state

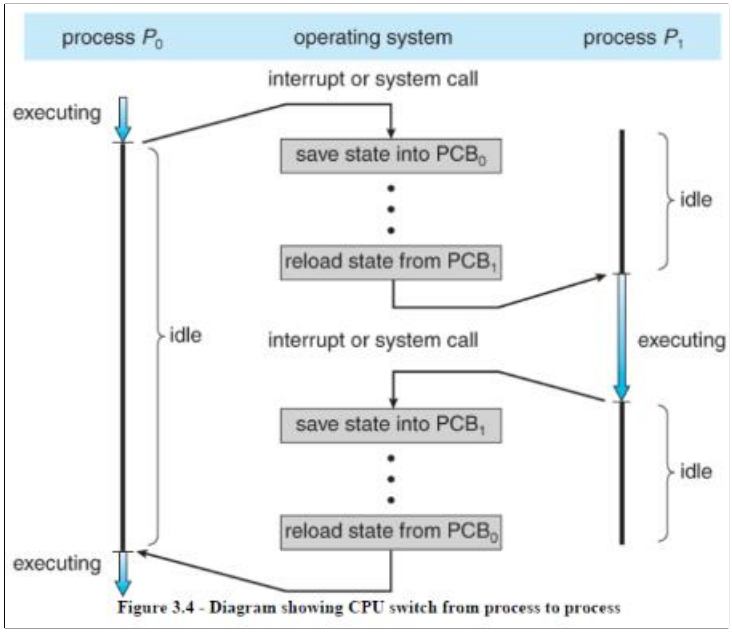


Figure 3.4 - Diagram showing CPU switch from process to process

Process Scheduling

- Two main objectives of process scheduling system are keep CPU busy at all times and deliver "acceptable" response times programs
- Process scheduler must meet objectives implementing suitable policies for swapping processes in and out of the CPU.
- (Note objectives can conflict. Every time system steps in to swap processes it takes up time on CPU)

(1) Scheduling Queues

- Processes stored in job queue.
- Processes in Ready state placed in ready queue.
- Processes waiting for device or deliver data, placed in device queues. Separate device queue for each device.
- Other queues may also be created and used as needed.

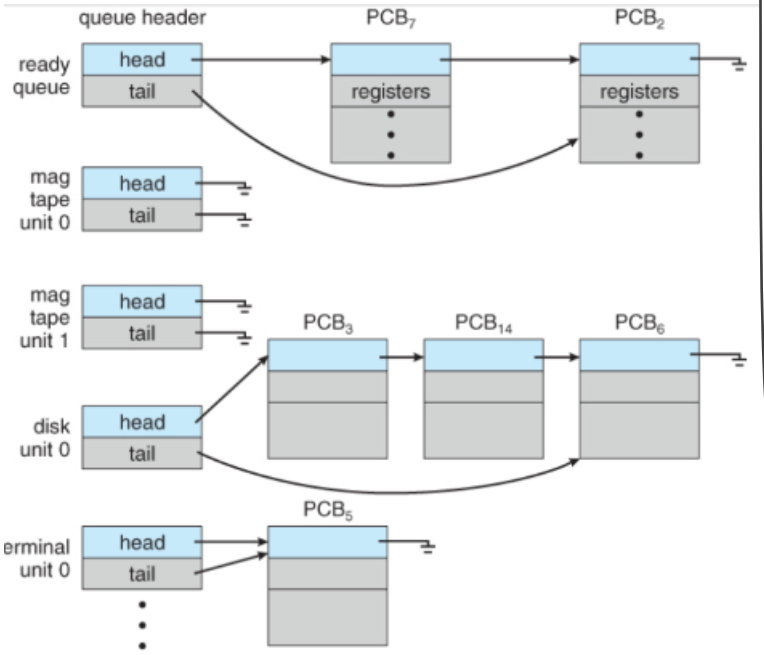


Figure 3.5 - The ready queue and various I/O device queues

(2) Schedulers

- Long-term scheduler is batch system (heavily loaded system). Runs infrequently, (such when process ends selecting one more to be loaded in from disk in its place), can afford take time implement intelligent and advanced scheduling algorithms.
- Short-term scheduler (CPU Scheduler) runs very frequently, 100 milliseconds, quickly swap one process out of CPU and swap in another one.
- Some systems employ medium-term scheduler. System loads get high, this scheduler swap one or more processes out of ready queue system for few seconds, to allow smaller faster jobs finish quickly and clear system.
- Efficient scheduling system select good process mix of CPU-bound processes and I/O bound processes.

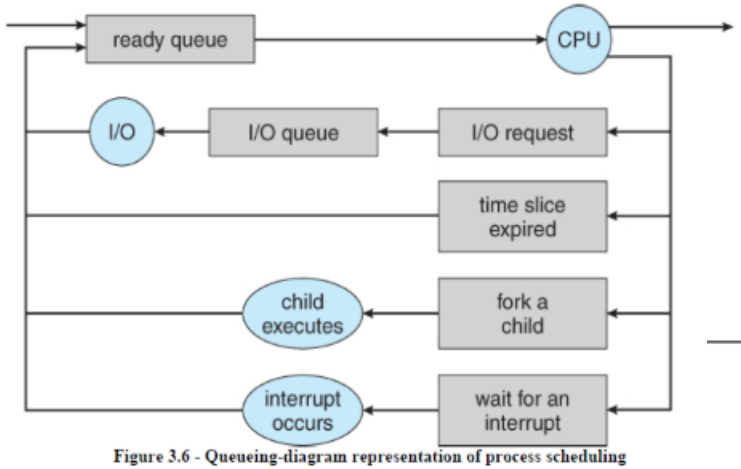


Figure 3.6 - Queueing-diagram representation of process scheduling

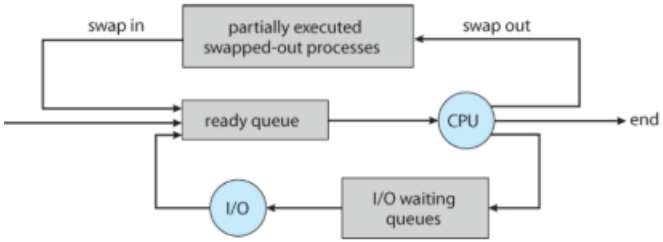


Figure 3.7 - Addition of a medium-term scheduling to the queueing diagram

(3) Context Switch

- Whenever interrupt arrives, CPU must do a state-save of currently running process, switch into kernel mode to handle interrupt, do state-restore of interrupted process.
- Similarly, context switch occurs when time slice for one process expired and new process to be loaded from ready queue. Instigated by timer interrupt, cause current process's state saved and new process's state restored.
- Saving and restoring states involves saving and restoring all registers and program counter(s), as well as process control blocks described above.
- Context switching happens VERY VERY frequently, overhead switching is lost CPU time, so context switches (state saves & restores) need to be fast as possible. Some hardware has special provisions for speeding this up, single machine instruction for saving or restoring all registers at once.
- Sun hardware has multiple sets registers, context switching speeded up switching which set registers currently in use. Limit processes switched between, attractive implement medium-term scheduler to swap some processes out

MULTITASKING IN MOBILE SYSTEMS

Because of the constraints imposed on mobile devices, early versions of iOS did not provide user-application multitasking; only one application runs in the foreground and all other user applications are suspended. Operating-system tasks were multitasked because they were written by Apple and well behaved. However, beginning with iOS 4, Apple now provides a limited form of multitasking for user applications, thus allowing a single foreground application to run concurrently with multiple background applications. (On a mobile device, the foreground application is the application currently open and appearing on the display. The background application remains in memory, but does not occupy the display screen.) The iOS 4 programming API provides support for multitasking, thus allowing a process to run in the background without being suspended. However, it is limited and only available for a limited number of application types, including applications

- running a single, finite-length task (such as completing a download of content from a network);
- receiving notifications of an event occurring (such as a new email message);
- with long-running background tasks (such as an audio player.)

Apple probably limits multitasking due to battery life and memory use concerns. The CPU certainly has the features to support multitasking, but Apple chooses to not take advantage of some of them in order to better manage resource use.

Android does not place such constraints on the types of applications that can run in the background. If an application requires processing while in the background, the application must use a service, a separate application component that runs on behalf of the background process. Consider a streaming audio application: if the application moves to the background, the service continues to send audio files to the audio device driver on behalf of the background application. In fact, the service will continue to run even if the background application is suspended. Services do not have a user interface and have a small memory footprint, thus providing an efficient technique for multitasking in a mobile environment.

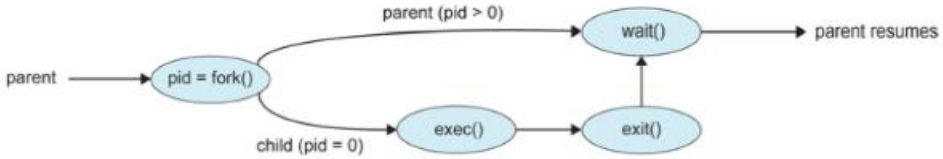


Figure 3.10 - Process creation using the fork( ) system call

Operations on Processes

(1) Process Creation

- Processes may create other processes appropriate system calls, fork or spawn. Process creating is termed parent of other process, termed its child.
- Each process given integer identifier, process identifier, (PID). PID stored for each process.
- UNIX systems process scheduler termed sched, PID 0. At system startup time it launch init, gives that process PID 1. Init launches all system daemons and user logins, becomes ultimate parent of all processes.

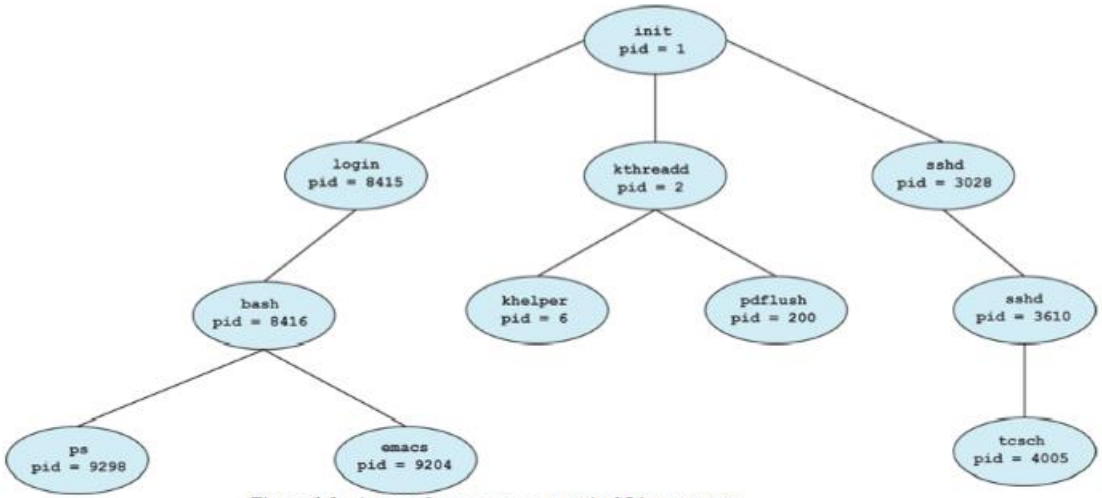


Figure 3.8 - A tree of processes on a typical Linux system

- Child process may receive shared resources with parent. May or may not limited subset resources originally allocated to parent, preventing runaway children consuming system resource.
- Options for parent process after creating child:
  1. Wait for child process terminate before proceeding. Parent makes wait( ) system call, either specific child or any child, causes parent process block until wait( ) returns. UNIX shells wait for children to complete before issuing new prompt.
  2. Run concurrently with child, continuing process without waiting. Operation seen when UNIX shell runs process background task. Possible for parent run, then wait for child later, might occur parallel processing operation. (parent fork off number children without waiting for any of them, then do a little work of its own, and then wait for the children.)
- Two possibilities address space of child relative to parent:
  1. The child may be exact duplicate of parent, sharing same program and data segments in memory. Each have their own PCB, program counter, registers, and PID. Behavior of fork system call UNIX.
  2. Child process new program loaded into its address space, all new code and data segments. Behavior spawn system calls in Windows. UNIX implement as second step, using exec system call.

Process Termination

- Process executes last statement and then asks the operating system to delete it using the exit() system call.
  - Returns status data from child to parent (via wait())
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using abort() system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - Parent exiting and operating systems does not allow child to continue if its parent terminates
- Some OSes don't allow child exists if parent has terminated
  > cascading termination - if process terminates, all its children, grandchildren, etc also terminated.
  > Termination initiated by operating system
- Parent process **may** wait for termination of child process by using **wait()** system call.
  > Call returns status information and pid of the terminated process
  pid = wait(&status);
- If no parent waiting (did not invoke **wait()**) process is a **zombie**
  > All its resources are deallocated, but exit status is kept
- If parent terminated without invoking wait , process is an **orphan**
  > UNIX: assigns init process as the parent
  > Init calls wait periodically

Cooperating Processes

Processes within a system may be independent or cooperating

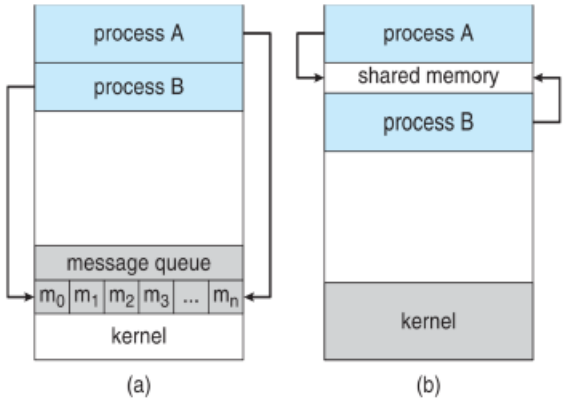
- When processes execute produce some computational results
- **Independent process** cannot affect (or affected) by such results of another process
- **Cooperating process** can affect (or affected) by such results of another process

Advantages of process cooperation

- Information sharing
- Computation speed-up
- Modularity
- Convenience

Interprocess Communication

For fast exchange of information, cooperating processes need some interprocess communication (IPC) mechanisms
> Two models of IPC
> Shared memory
>Message passing



Interprocess Communication – Shared Memory

> An area of memory is shared among the processes that wish to communicate
> The communication is under the control of the users processes, not the OS.
> Major issue is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.

Producer-Consumer Problem

Common paradigm for cooperating processes
> Used to exemplify one common generic way/scenario of cooperation among processes

Producer process
> produces some information
> incrementally

Consumer process
> consumes this information
> as it becomes available

Challenge:
- Producer and consumer should run concurrently and efficiently
- Producer and consumer must be synchronized
  > Consumer cannot consume item before it is produced

Shared-memory solution to producer-consumer
- Uses a buffer in shared memory to exchange information
  unbounded-buffer: assumes no practical limit on the buffer size
  bounded-buffer assumes a fixed buffer size

Interprocess Communication – Message Passing

Mechanism for processes to communicate and to synchronize their actions
> Message system – processes communicate with each other without resorting to shared variables
> IPC facility provides two operations:
send(message)
receive(message)
> Message size is either fixed or variable

If processes P and Q wish to communicate, they need to:
- Establish a **communication link** between them
- Exchange messages via send/receive

Implementation issues:
- 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 (buffer size) of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

Logical implementation of communication link
- Direct or indirect
- Synchronous or asynchronous
- 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 a direct communication link

> Links are established automatically
> A link is associated with exactly one pair of communicating processes
> Between each pair there exists exactly one link
> The link may be unidirectional, but is usually bi-directional

Indirect Communication

Messages directed and received from **mailboxes** (also referred to as **ports**)

> Each mailbox has a unique id
> Processes can communicate only if they share a mailbox

Properties of indirect communication link

> Link established only if processes share a common mailbox
> A link may be associated with many processes
> Each pair of processes may share several communication links
> Link may be unidirectional or bi-directional

Operations

create a new mailbox (port)
send and **receive** messages through mailbox
destroy a mailbox

Primitives are defined as:

send(A, message) – send a message to mailbox A
receive(A, message) – receive a message from mailbox A

Mailbox sharing issues

> 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**

Blocking is considered **synchronous**

> **Blocking send** -- the sender is blocked until the message is received
> **Blocking receive** -- the receiver is blocked until a message is available

Non-blocking is considered **asynchronous**

> **Non-blocking send** -- the sender sends the message and continues
> **Non-blocking receive** -- the receiver receives:
  - A valid message, or
  - Null message

Different combinations possible

> If both send and receive are blocking – called a **rendezvous**

Buffering in Message-Passing

Queue of messages is attached to the link.
Implemented in one of three ways

1. **Zero capacity** – no messages are queued on a link.
   - *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*

Communication in ClientServer Systems

Sockets

Socket is an endpoint for communication.
Two processes communicating over a network often use a pair of connected sockets as a communication channel. Software that is designed for clientserver operation may also use sockets for communication between two processes running on same computer - Example UI for database program may communicate with backend database manager using sockets.

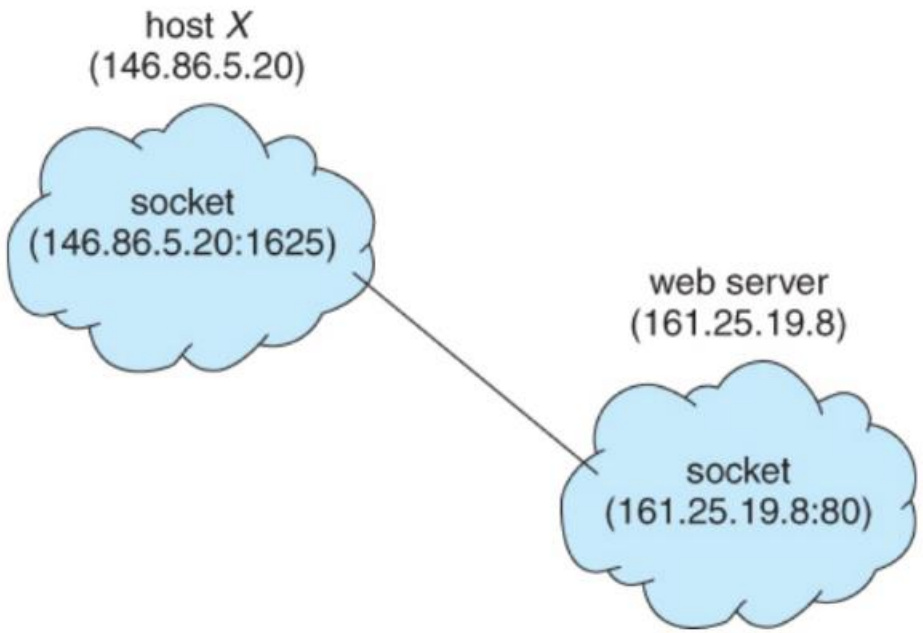
A socket is identified by an IP address concatenated with a port number, e.g. 200.100.50.5:80.

Communication channels via sockets may be of one of two major forms:

1) Connectionoriented (TCP, Transmission Control Protocol) connections emulate a telephone connection. All packets sent down the connection are guaranteed to arrive in good condition at the other end, and to be delivered to the receiving process in the order in which they were sent. The TCP layer of the network protocol takes steps to verify all packets sent, resend packets if necessary, and arrange the received packets in the proper order before delivering them to the receiving process. There is a certain amount of overhead involved in this procedure, and if one packet is missing or delayed, then any packets which follow will have to wait until the errant packet is delivered before they can continue their journey.

2) Connectionless (UDP, User Datagram Protocol) emulate individual telegrams. There is no guarantee that any particular packet will get through undamaged (or at all), and no guarantee that the packets will get delivered in any particular order. There may even be duplicate packets delivered, depending on how the intermediary connections are configured. UDP transmissions are much faster than TCP, but applications must implement their own error checking and recovery procedures.

Sockets are considered a lowlevel communications channel, and processes may often choose to use something at a higher level, such as those covered in the next two sections



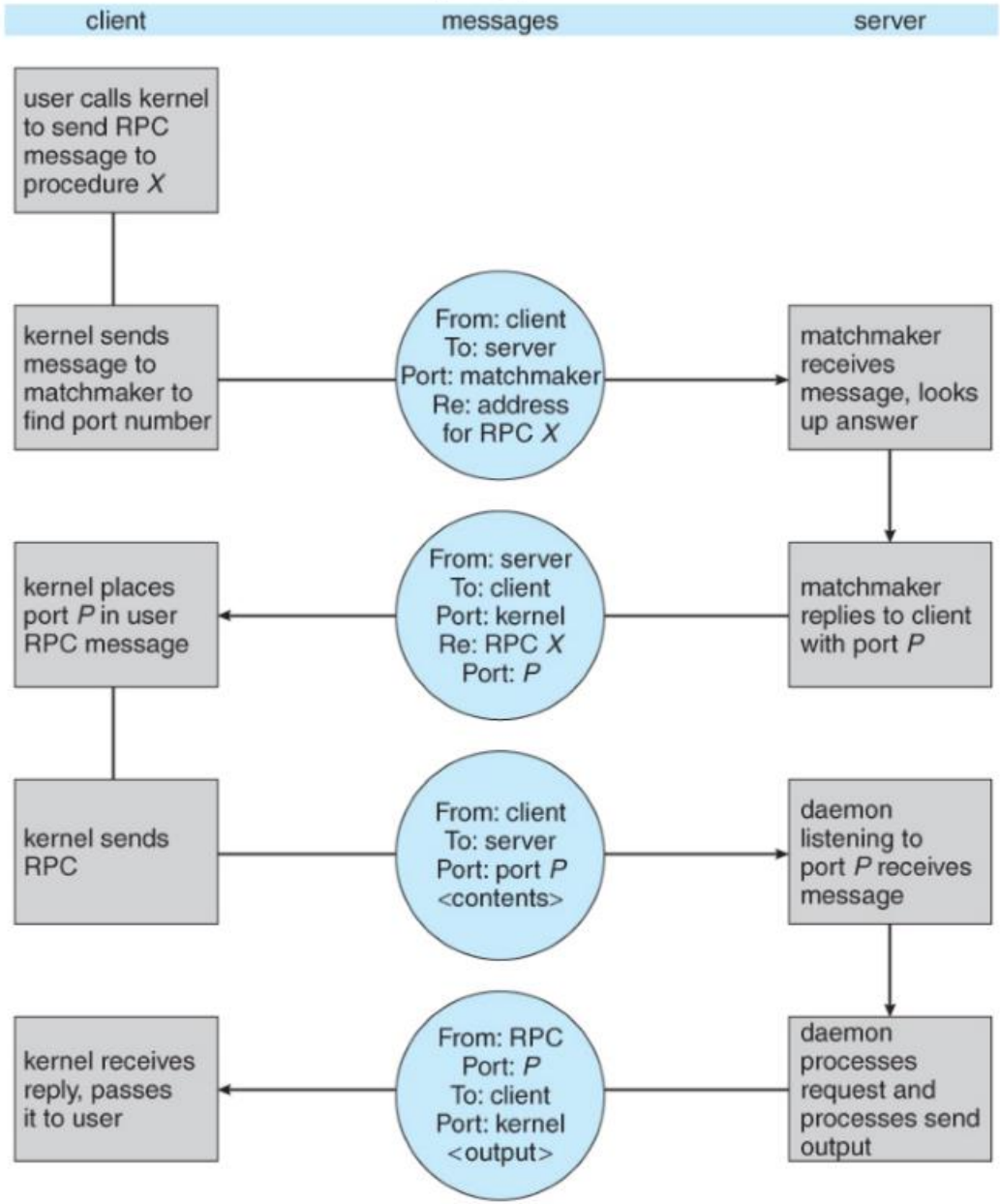
Remote Procedure Calls, RPC

> Concept of RPC is to make procedure calls similarly to calling ordinary local procedures, except procedure being called lies on a remote machine.
> Implementation involves stubs on either end of the connection.
- Local process calls on the stub, much as it would call upon a local procedure.
- RPC system packages up (marshals) the parameters to the procedure call, and transmits them to the remote system.
- On the remote side, the RPC daemon accepts the parameters and calls upon the appropriate remote procedure to perform the requested work.
- Any results to be returned are then packaged up and sent back by the RPC system to the local system, which then unpackages them and returns the results to the local calling procedure.

Difficulty formatting of data on local versus remote systems.
Resolution involves an agreedupon intermediary format, such as XDR (external data representation).

Another issue identifying which procedure on remote system particular RPC is destined for.
> Remote procedures identified by ports, not same ports as socket ports.
> One solution is for calling procedure to know port number to communicate with on remote system.
> Problematic, port number would be compiled into code, break down if remote system changes port numbers.
> Matchmaker process employed, acts like telephone directory service. Local process must first contact matchmaker on remote system, looks up desired port number and returns it. Local process can then use that information to contact desired remote procedure. Involves an extra step, but is more flexible.

One common example of a system based on RPC calls is a networked file system. Messages are passed to read, write, delete, rename, or check status, as might be made for ordinary local disk access requests.



Pipes

Pipes are one of the earliest and simplest channels of communications between (UNIX) processes.
Key considerations in implementing pipes:

1. Unidirectional or Bidirectional communication?
2. Is bidirectional communication halfduplex or fullduplex?
3. Must a relationship such as parentchild exist between the processes?
4. Can pipes communicate over a network, or only on the same machine?

Ordinary Pipes

Ordinary pipes are unidirectional, reading end and writing end. (If bidirectional communications needed, a second pipe is required.)

UNIX ordinary pipes are created with the system call "int pipe(int fd [ ])".

> Rreturn value is 0 on success, 1 if an error occurs.
> The int array must be allocated before call, values filled in by pipe system call:
- fd[0] is filled in with a file descriptor for the reading end of the pipe
- fd[1] is filled in with a file descriptor for the writing end of the pipe

> UNIX pipes accessible as files, using standard **read**( ) and **write**( ) system calls.
> Ordinary pipes are only accessible within the process that created them.
- Typically a parent creates the pipe before forking off a child.
- When the child inherits open files from its parent, including the pipe file(s), a channel of communication is established.
- Each process (parent and child) first close the ends of pipe they are not using.
example, if parent writing to pipe and child is reading, parent should close reading end of its pipe after fork and child close writing end.

Ordinary pipes in Windows are very similar

> Windows terms them anonymous pipes
> They are still limited to parentchild relationships.
> They are read from and written to as files.
> They are created with CreatePipe( ) function, which takes additional arguments.
> In Windows it is necessary to specify what resources a child inherits, such as pipes.

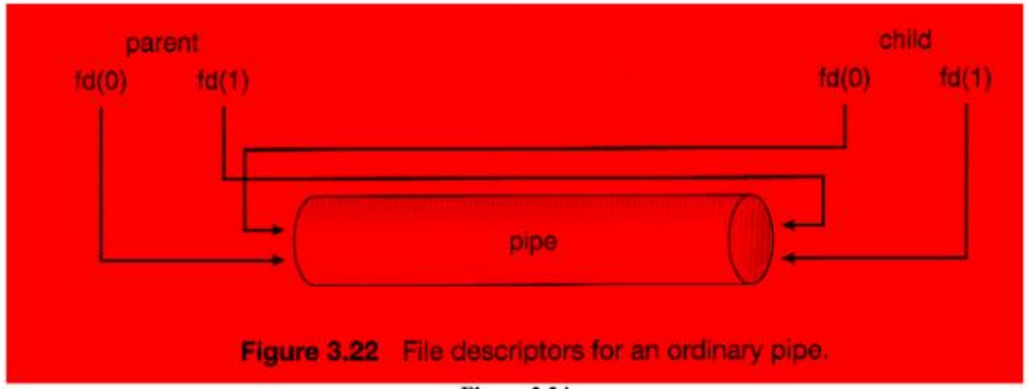


Figure 3.22 File descriptors for an ordinary pipe.

Named Pipes

Support bidirectional communication, communication between non parentchild related processes, and persistence after process which created them exits.
Multiple processes can also share a named pipe, typically one reader and multiple writers.

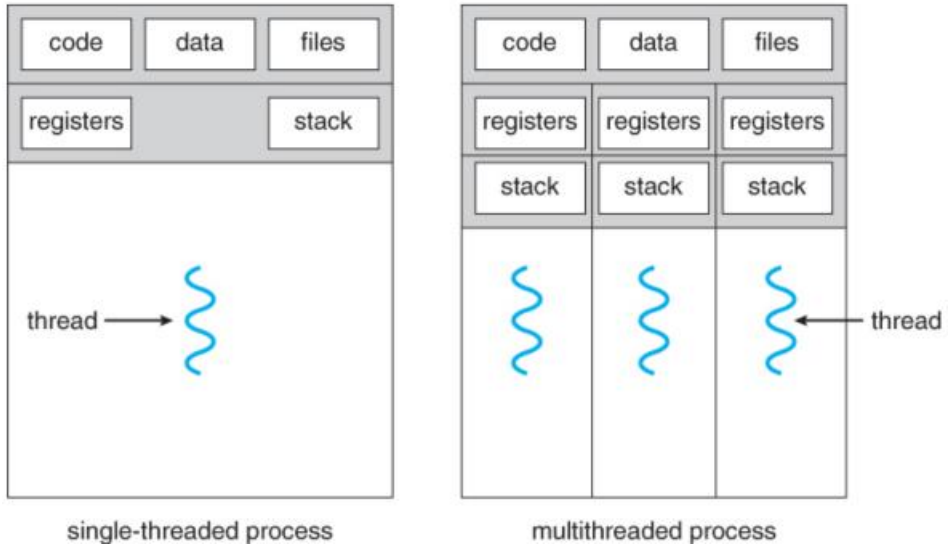
UNIX, named pipes are termed fifos, and appear as ordinary files in the file system.

> (Recognizable by a "p" as the first character of a long listing, e.g. /dev/initctl)
> Created with mkfifo( ) and manipulated with read( ), write( ), open( ), close( ), etc.
> UNIX named pipes are bidirectional, but halfduplex, so two pipes are still typically used for bidirectional communications.
> UNIX named pipes still require that all processes be running on the same machine. Otherwise sockets are used.

Windows named pipes provide richer communications. Fullduplex is supported.
Processes may reside on the same or different machines
Created and manipulated using CreateNamedPipe( ), ConnectNamedPipe( ), ReadFile( ),and WriteFile( ).

#4 Multithreaded Programming

- Thread is basic unit of CPU utilization, consisting of program counter, stack, and set of registers, a thread ID.
- Traditional (heavyweight) processes have single thread control There is one program counter, and one sequence of instructions that can be carried out at any given time.
- As shown in Figure, multithreaded applications have multiple threads within a single process, each having their own program counter, stack and set of registers, but sharing common code, data, and certain structures such as open files



Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

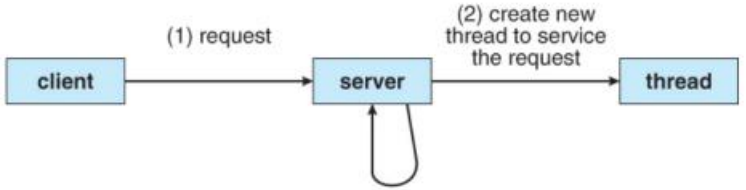


Figure 4.2 - Multithreaded server architecture

Benefits

1) Responsiveness

> may allow continued execution if part of process is blocked
> especially important for user interfaces

2) Resource Sharing

> threads share resources of process: easier than shared memory or message passing

3) Economy

> Thread creation is faster than process creation
- Less new resources needed vs a new process
- Solaris: 30x faster

> Thread switching lower overhead than context switching
- 5x faster

4) Scalability

> Threads can run in parallel on many cores

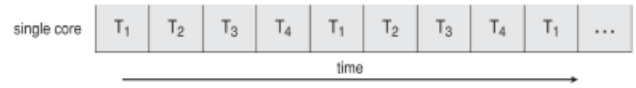
Multicore or multiprocessor systems

- Putting pressure on programmers
- How to load all of them for efficiency
- Challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging

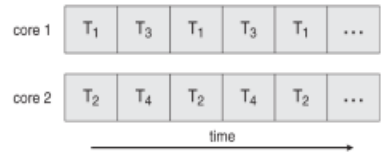
Concurrency vs. Parallelism

- Parallelism implies a system can perform more than one task simultaneously
- Concurrency supports more than one task making progress
  - True parallelism or an illusion of parallelism
  - Single processor / core, scheduler providing concurrency

Concurrent execution on single-core system



Parallelism on a multi-core system:



Types of Parallelism

1) Data parallelism – distributes subsets of the same data across multiple cores, same operation/task on each

Example: the task of incrementing elements by one of an array can be split into two: incrementing its elements in the 1st and 2nd halfs

2) Task parallelism – distributing threads across cores, each thread performing unique operation

In practice, people often follow a hybrid of the two

User Threads and Kernel Threads

User threads

- Support provided at the user-level
- Managed above the kernel
  - > without kernel support
- Management is done by thread library
  - Three primary thread libraries:
  - POSIX Pthreads, Windows threads, Java threads

Kernel threads

- Supported and managed by OS
- Virtually all modern general-purpose operating systems support them

Many-to-One

Many user-level threads mapped to single kernel thread

Advantage:

- Thread management in user space
- Hence, efficient

Disadvantages:

- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore sy
  - > Since only 1 may be in kernel at a time
  - > So, few systems currently use this model

Many-to-Many Model

> Allows many user-level threads to be mapped to (smaller or equal number of) kernel threads

> Allows the OS to create a sufficient number of kernel threads

- The number is dependent on specific machine or application
- It can be adjusted dynamically

> Many-to-one

- Any number of threads is allowed, but low concurrency

> One-to-one

- Great concurrency, but the number of threads is limited

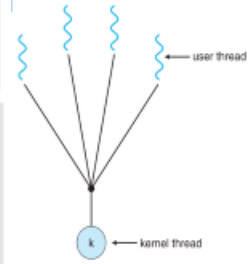
> Many-to-many

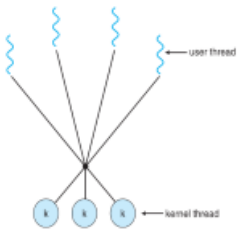
- Gets rid of the shortcomings of the precious two

Multithreading Models

Relationship between user threads and kernel threads

- Three common ways of establishing this relationships
  1) Many-to-One model
  2) One-to-One model
  3) Many-to-Many model





One-to-One

> Each user-level thread maps to kernel thread

> Creating a user-level thread creates a kernel thread

Advantages:

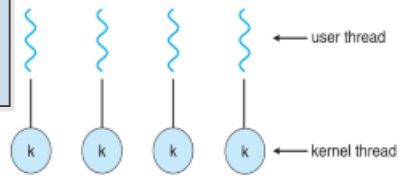
- More concurrency than many-to-one

Disadvantages:

- High overhead of creating kernel threads
- Hence, number of threads per process sometimes restricted

Examples

- Windows
- Linux
- Solaris 9 and later

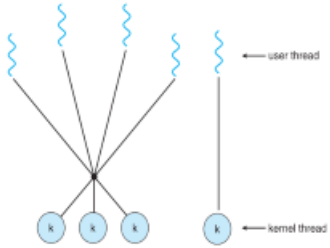


Two-level Model

> Similar to Many-to-Many

> Allows user thread to be bound to kernel thread

- IRIX
- HP-UX
- Tru64 UNIX
- Solaris 8 and earlier



Thread Libraries

Provides programmer with API for creating and managing threads
Primary ways of implementing
1) Library entirely in user space
2) Kernel-level library supported by the OS
Examples of thread libraries
Pthreads,
Java Threads,
Windows threads

Threading Issues

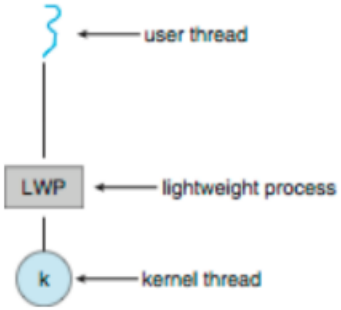
Semantics of **fork**() and **exec**() system calls
Many other issues
- Signal handling
  > Synchronous and asynchronous
- Thread cancellation of target thread
  > Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

Semantics of fork() and exec()

- Does **fork**()duplicate only the calling thread or all threads?
  > Some UNIXes have two versions of fork
- **exec**() usually works as normal – replace the running process including all threads

Lightweight process

Lightweight Process (LWP)
> Intermediate data structure between user and kernel threads
> To user-level thread library, it appears as a virtual processor on which process can schedule user thread to run
> Each LWP attached to a kernel thread
> LWP are used, for example, to implement Many-to-many and two-level models



Pthreads

The POSIX standard (IEEE 1003.1c) defines the specification for pThreads, not the implementation.
pThreads are available on Solaris, Linux, Mac OSX, Tru64, and via public domain shareware for Windows.
Global variables are shared amongst all threads.
One thread can wait for the others to rejoin before continuing.
pThreads begin execution in a specified function

Java Threads

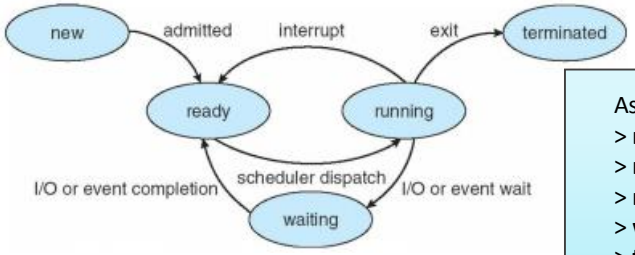
> ALL Java programs use Threads even "common" singlethreaded ones.
> Creation of new Threads requires Objects that implement the Runnable Interface, which means they contain a method "public void run( )" . Any descendant of the Thread class will naturally contain such a method. (In practice the run( ) method must be overridden / provided for the thread to have any practical functionality.)
> Creating a Thread Object does not start the thread running To do that the program must call the Thread's "start( )" method. Start( ) allocates and initializes memory for the Thread, and then calls the run( ) method. (Programmers do not call run( ) directly.)
> Because Java does not support global variables, Threads must be passed a reference to a shared Object in order to share data, in this example the "Sum" Object.
> Note that the JVM runs on top of a native OS, and that the JVM specification does not specify what model to use for mapping Java threads to kernel threads. This decision is JVM implementation dependant, and may be onetoone, manytomany, or many to one.. (On a UNIX system the JVM normally uses PThreads and on a Windows system it normally uses windows threads.)

Linux Threads

Linux refers to them as tasks rather than threads
Thread creation is done through clone() system call clone() allows a child task to share the address space of the parent task (process)
- Flags control behavior
- struct task_struct points to process data structures (shared or unique)

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.

#5 Process Scheduling



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

Maximum CPU utilization obtained with **multiprogramming**
- waiting for I/O is wasteful
- 1 thread will utilize only 1 core

CPU–I/O Burst Cycle
- Process execution consists of:
  - > a **cycle** of CPU execution
  - > and I/O wait

CPU burst followed by **I/O burst**
CPU burst distribution is of main concern

Levels of Scheduling

High-Level Scheduling
> See Long-term scheduler or Job Scheduling from Chapter 3
> Selects jobs allowed to compete for CPU and other system resources.

Intermediate-Level Scheduling
> See Medium-Term Scheduling from Chapter 3
> Selects which jobs to temporarily suspend/resume to smooth fluctuations in system load.

Low-Level (CPU) Scheduling or Dispatching
> Selects the ready process that will be assigned the CPU.
> Ready Queue contains PCBs of processes.

Dispatcher

a module that 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
- This time should be as small as possible

CPU Scheduler

- Short-term scheduler
  - > Selects 1 process from the ready queue
    - @ then allocates the CPU to it
  - > Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is called nonpreemptive (=cooperative)
- All other scheduling is called preemptive
  - > Process can be interrupted and must release the CPU
  - > Special care should be taken to prevent problems that can arise
    - @ Access to shared data –race condition can happen, if not handled
    - @ Etc.

Scheduling Criteria

How do we decide which scheduling algorithm is good?
Many criteria for judging this has been suggested
- Which characteristics considered can change significantly which algo is considered the best

> 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 to stat responding
- Used for interactive systems
- Time from when a request was submitted until the first response is produced

Scheduling Algorithm Optimization Criteria

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

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

Suppose that the processes arrive in the order: P1, P2, P3
The Gantt Chart for the schedule is:

- Waiting time for
  - P1 = 0
  - P2 = 24
  - P3 = 27
- **Average waiting time: (0 + 24 + 27)/3 = 17**

Suppose that the processes arrive in the order: P2, P3 ,P1
The Gantt chart for the schedule is:

- Waiting time for
  - P1 = 6
  - P2 = 0
  - P3 = 3
- **Average waiting time: (6 + 0 + 3) / 3 = 3**
- Much better than previous case
- Average waiting time of FCFS not minimal And it may vary substantially
- FCFS is **nonpreemptive**
- Not a good idea for timesharing systems
- FCFS suffers from the **convoy effect**

FCFS Scheduling: Convoy Effect

- when several short processes wait for long a process to get off the CPU
- Assume
  - 1 long CPU-bound process
  - Many short I/O-bound processes
- Execution:
  - The long one occupies CPU
    - The short ones wait for it: no I/O is done at this stage
    - No overlap of I/O with CPU utilizations
  - The long one does its first I/O
    - Releases CPU
    - Short ones are scheduled, but do I/O, release CPU quickly
  - The long one occupies CPU again, etc
- Hence **low CPU and device utilization**

Shortest-Job-First (SJF) Scheduling

> Associate with each process the length of its next CPU burst
  - SJF uses these lengths to schedule the process with the shortest time

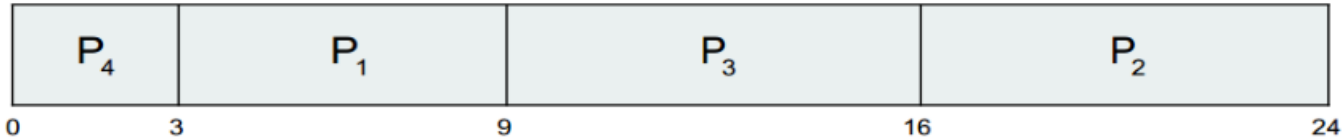
> Notice, the burstis used by SJF,
  - **Not** the process end-to-end running time
    implied by word "job" in SJF
  - Hence, it should be called ``Shorted-Next-CPU-Burst"
  - However, "job" is used for historic reasons

> Two versions of SJF: **preemptive** and **nonpreemptive**
  - Assume
    A new process Pnew arrives while the current one Pcur is still executing
    The burst of Pnew is less than what is left of Pcur
  - **Nonpreemptive** SJF –will let Pcur finish
  - **Preemptive SJF** –wil preempt Pcur and let Pnew execute
    This is also called shortest-remaining-time-first scheduling

Advantage:
  - SJF is optimal in terms of the average waiting time

Challenge of SJF:
  - Hinges on knowing the length of the next CPU burst
    But how can we know it?
    Solutions: ask useror estimateit
  - In a batch systemand long-term scheduler
    Could ask the user for the job time limit
    The user is motivated to accurately estimate it
      – Lower value means faster response
      – Too low a value will cause time-limit violation and job rescheduling
  - In a short-term scheduling
    Use estimation
    Will be explained shortly

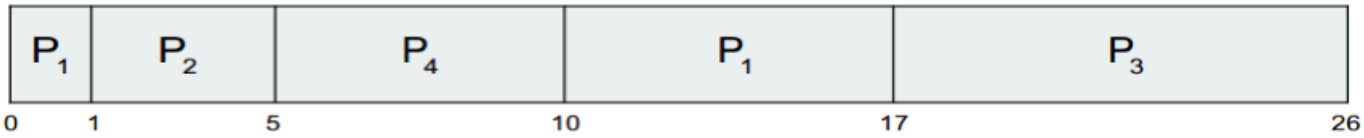
Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3



- SJF scheduling chart

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

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



- *Preemptive* SJF Gantt Chart

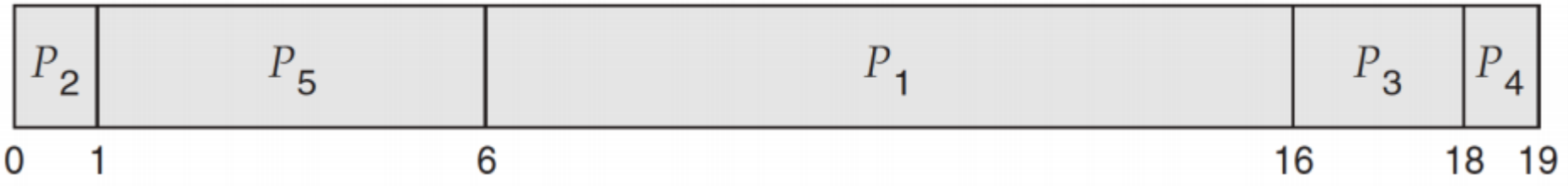
Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

Priority Scheduling

> A priority number (integer) is associated with each process
> The CPU is allocated to the process with the highest priority
(smallest integer ≡highest priority)
- **Preemptive**
- **Nonpreemptive**

> SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
> Problem ≡**Starvation**–low priority processes may never execute
> Solution ≡**Aging**–as time progresses increase the priority of the process

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



Average waiting time = 8.2 msec

Round Robin (RR)

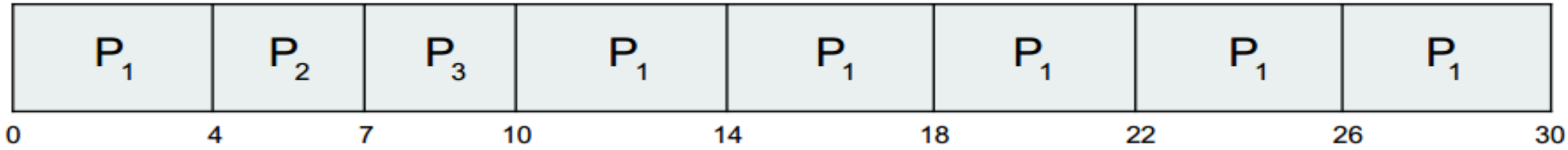
> Each process gets a small unit of CPU time
  - **Time quantum** q
  - Usually 10-100 milliseconds
> After this time has elapsed:
  - the process is **preempted** and
  - added to the end of the ready queue
> The process might run for ≤ q time
  - For example, when it does I/O
> If
  - *n* processes in the ready queue, and
  - the time quantum is q

then
  - "Each process gets 1/nof the CPU time"
    Incorrect statement from the textbook
  - in chunks of ≤ qtime units at once
  - Each process waits ≤ (n-1)q time units

> Timer interrupts every quantum to schedule next process
> Performance
  - q large ⇒FIFO
  - q small ⇒overhead of context switch time is too high
> Hence, q should be large compared to context switch time
  - q usually 10ms to 100ms,
  - context switch < 10 usec

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3



> Higher average turnaround (end-to-end running time) than SJF
> But better response than SJF

Multilevel Queue

> Another class of scheduling algorithms when processes are classified into groups, for example:
- **foreground**(interactive) processes
- **background**(batch) processes

> Ready queue is partitioned into separate queues, e.g.:
- **Foreground** and **background** queues

> Process is permanently assigned to one queue
> Each queue has its own scheduling algorithm, e.g.:
- foreground –RR
- background –FCFS

<Scheduling must be done between the queues:
- Fixed priority scheduling
  For example, foreground queue might have absolute priority over background queue
  – Serve all from foreground then from background
  – Possibility of starvation
- Time slice scheduling
  Each queue gets a certain amount of CPU time which it can schedule amongst its processes, e.g.:
  – 80% to foreground in RR
  – 20% to background in FCFS

Multiple-Processor Scheduling

> Multiple CPUs are available
- Load sharing becomes possible
- Scheduling becomes more complex

> Solutions: Have one ready queue accessed by each CPU
- **Self scheduled** -each CPU dispatches a job from ready Q
  - Called symmetric multiprocessing (SMP)
  - Virtually all modern OSes support SMP
- **Master-Slave** -one CPU schedules the other CPUs
  - The others run user code
  - Called asymmetric multiprocessing
  - One processor accesses the system data structures
    = Reduces the need for data sharing

Queueing Models

> Defines a probabilistic model for
- Arrival of processes
- CPU bursts
- I/O bursts

> Computes stats
- Such as: average throughput, utilization, waiting time, etc
- For different scheduling algorithms

Real-Time CPU Scheduling

> Special issues need to be considered for **real-time** CPU scheduling
- They are different for **soft** vs **hardreal**-time systems

> **Soft real-time systems**
- Gives preference to critical processed over over non-critical ones
- But no guarantee as to whencritical real-time process will be scheduled

> **Hard real-time systems**
- Task mustbe serviced by its deadline
- Otherwise, considered failure

> Real-time systems are often **event-driven**
- The system must detect the event has occurred
- Then respond to it as quickly as possible
- Event latency–amount of time from when event occurred to when it is services
- Different types of events will have different event latency requirements

> Two types of latencies affect performance

1. Interrupt latency
- time from arrival of interrupt to start of routine that services interrupt
- Minimize it for soft real-time system
- Bound it for hard real-time

2. Dispatch latency
- time for scheduler to take current process off CPU and switch to another
- Must also be minimized

#6 Synchronization

Processes can execute **concurrently**
> May be interrupted at any time, partially completing execution
 Concurrent access to shared data may result in data inconsistency
 Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Illustration of the problem:
Suppose that we wanted to provide a solution to the consumer producer problem that fills all the buffers. We can do so by having an integer counter that keeps track of the number of full buffers.
Initially, counter is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Race Condition

> counter++ could be implemented as
register1 = counter
register1 = register1 + 1
counter = register1
> counter-- could be implemented as
register2 = counter
register2 = register2 – 1
counter = register2

> execution interleaving with "count = 5" initially:

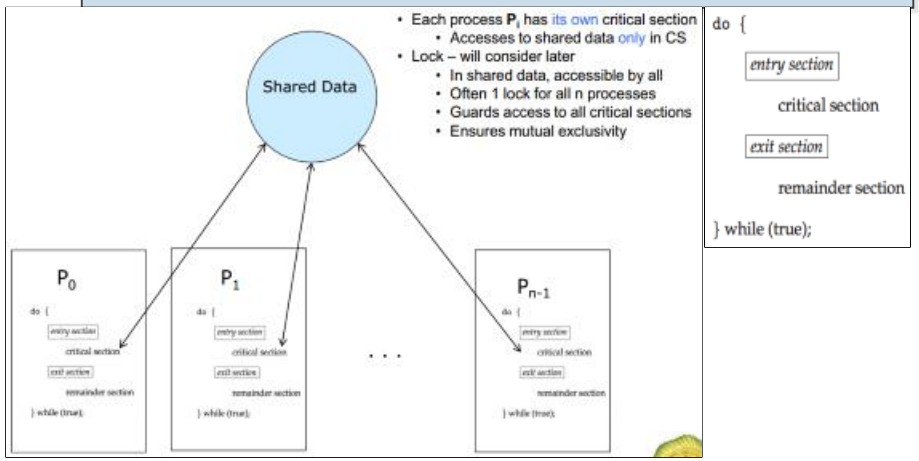
S0: producer execute **register1 = counter**	{register1 = 5}
S1: producer execute **register1 = register1 + 1**	{register1 = 6}
S2: consumer execute **register2 = counter**	{register2 = 5}
S3: consumer execute **register2 = register2 – 1**	{register2 = 4}
S4: producer execute **counter = register1**	{counter = **6**}
S5: consumer execute **counter = register2**	{counter = **4**}

Race condition – Situation when:
> several processes access the same data concurrently
> the outcome of the execution depends on the order of the accesses

Critical Section Problem

> Consider system of ***n*** processes **{P0, P1, … ,Pn-1}**
> Each process has **critical section** segment of code
- Process may be changing common variables, updating table, writing file, etc
- When one process in critical section, no other is allowed to be in its critical section

> ***Critical section problem*** is to design protocol to solve this
> Each process
- must ask permission to enter critical section in **entry section**,
- may follow critical section with **exit section**,
- then **remainder section**



Requirements to Critical-Section Problem

> Assume that each process executes at a nonzero speed
> No assumption concerning relative speed of the n processes

1. Mutual Exclusion
Only one process can be in the critical section at a time – otherwise what critical section?

2. Progress
Intuition:No process is forced to wait for an available resource – otherwise very wasteful.
Formal:If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

3. Bounded Waiting
Intuition: No process can wait forever for a resource – otherwise an easy solution: no one gets in
Formal: 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

Types of solutions to CS problem

1 Software-based
2 Hardware-based

Software-based solution to CS

Solutions assume that load and store machine-language instructions are atomic; that is, cannot be interrupted

> In general, this is not true for modern architectures

- Peterson's algorithm does not work in general
- Can work on some machines correctly, but can fail on others

> But good algorithmic description, allows to understand various issues

The two processes share two variables:

int turn;

Boolean flag[2]

The variable turn indicates whose turn it is to enter the CS

The flag array is used to indicate if a process is ready to enter the CS

flag[i] = true implies that process Pi is ready!

Synchronization Hardware

> Many systems provide hardware support for implementing the critical section code.

>All solutions below based on idea of locking

- Protecting critical regions via locks

>Uniprocessors – could disable interrupts

-Currently running code would execute without preemption

That is, without being interrupted

-Generally too inefficient on multiprocessor systems

OSes using this are not broadly scalable

>Modern machines provide special **atomic hardware instructions**

Atomic = non-interruptible

test_and_set instruction

test memory word and set value

compare_and_swapinstruction

swap contents of two memory words

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutexlock
- Protect a critical section by:
  - first acquire()a lock
  - then release()the lock
  - Boolean variable indicating if lock is available or not
- Calls to acquire()and release()must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires busy waiting
  - This lock therefore called a spinlock

acquire() and release()

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;
}
```

```
release() {
    available = true;
}
```

```
while (true) {
    acquire lock
        critical section
    release lock
        remainder section
}
```

Deadlock and Starvation

> **Synchronization problems**

> **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

> Let Sand Q be two semaphores initialized to 1

P0	P1
wait(S);	wait(Q);
wait(Q);	wait(S);
...	...
signal(S);	signal(Q);
signal(Q);	signal(S);

> **Starvation (= indefinite blocking)**

- Related to deadlocks (but different)
- Occurs when a process waits indefinitely in the semaphore queue
- For example, assume a LIFO semaphore queue
  - Processed pushed first into it might not get a chance to execute

> **Priority Inversion**

A situation when a higher-priority process needs to wait for a lower-priority process that holds a lock

Classical Problems of Synchronization

Classical problems used to test newly-proposed synchronization schemes

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

#7 Deadlocks

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

System Model

> System consists of resources
> Resource types R1, R2, . . ., Rm
CPU cycles, files, memory space, I/O devices, etc
> Each resource type ***Ri*** has ***Wi*** instances.
- Type: CPU
2 instances - CPU1, CPU2
-Type: Printer
3 instances - printer1, printer2, printer3
> Each process utilizes a resource as follows:
- Request resource
A process can request a resource of a given type
E.g., "I request any printer"
System will then assign a instance of that resource to the process
E.g., some printer will be assigned to it
If cannot be granted immediately, the process must wait until it can get it
- Use resource
Operate on the resource, e.g. print on the printer
- Release resource
> Mutexes and Semaphores
Special case:
Each mutex or semaphor is treated as a separate resource type
Because a process would want to get not just "any" lock among a group of locks, but a specific lock that guards a specific shared data type – e.g., lock that

Deadlock Characterization

Deadlock can arise if **4 conditions hold simultaneously**:

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 {P0, P1, …, Pn} of waiting processes such that
- P0 is waiting for a resource that is held by P1, and so on:
- P0 → P1 → P2 → … → Pn–1→ Pn → P0

Notice: "Circular wait" implies "Hold and Wait"
- *Why then not test for only the "Circular wait"?*
- *Because, computationally, "Hold and wait" can be tested much more efficiently than "Circular wait"*
- *Some algorithms we consider only need to check H&W*

Methods for Handling Deadlocks

1. Ensure that the system will never enter a deadlock state:
- Deadlock prevention
- Deadlock avoidance
2. Allow the system to enter a deadlock state and then recover
3. Ignore the problem and pretend that deadlocks never occur in the system
- Used by most operating systems, including UNIX
- One reason: Handling deadlocks can be computationally expensive

Deadlock Prevention

If we ensure at least one condition is not met, we prevent a deadlock

Deadlock Prevention: Mutual Exclusion

#1 Mutual Exclusion: only one process at a time can use a resource

Solution:

> Mutual exclusion is not required for sharable resources

- Example: Accessing the same files, but only **for reading**
- So do not use mutual exclusion for such cases

> However, it must hold for non-sharable resources

:Use mutual exclusion only when you really have to

Deadlock Prevention: Hold and Wait

#2 Hold and Wait: a process holding at least one resource is waiting to acquire additional resources held by other processes

Idea: must guarantee that whenever a process requests a resource, it does not hold any other resources

Solutions include:

1. Require process to request and be allocated all its resources before it begins execution, or
2. Allow process to request resources only when the process has none allocated to it

- Cons: Low resource utilization; starvation possible

Deadlock Prevention: No Preemption

#3 No Preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task

One possible solution (is to implement preemption):

Assume

- Process P is holding some resources - (set) R
- P then requests another resource r
- But r cannot be immediately allocated to P

  That is, P must wait for r

Then

- All resources R are preempted

  That is, they are implicitly released
- Resources from R are added to the list of resources for which P is waiting
- P will be restarted only when it can regain R , as well as r

Deadlock Prevention: Circular Wait

#4 Circular Wait: there exists a set {P0, P1, …, Pn} of waiting processes such that

- P0 is waiting for a resource that is held by P1, and so on
- P0 → P1 → P2 → … → Pn–1→ Pn → P0

One possible solution:

- impose a total ordering of all resource types, and

  OrderOf(Rj) – gives order of Rj
- require that each process requests resources in an increasing order of enumeration

  Rewrite the code such that this holds

  If a process holds a lock for Rj it should not request a lock for any Rk such that

  OrderOf(Rk) < OrderOf(Rj)
- Example

  Order resources A,B,C,D,E as D < E < C < A < B

  Assume: Process holds a lock for, say, A and C

  Then, the process should not request locks for D or E

Deadlock Avoidance

Idea: Require system has some additional **priori** information about how resources will be used

- Simplest and most useful model requires that each process declare the **maximum number** of resources of each type that it may need
- The deadlock-avoidance algorithm **dynamically** examines the resource-allocation state to ensure that there can never be a **circular-wait condition**
- Resource-allocation **state** is defined by:
  1. number of **available** resources
  2. number of **allocated** resources
  3. **maximum resource demands** of the processes

Safe State

The deadlock avoidance algorithm relies on the notion of safe state

A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.

System is in safe state only if

- **There exists a safe sequence (of processes) -- explained shortly**
- **That is, it is possible to order all processes to form a safe sequence**

Safe Sequence

System is in safe state only if there exists a safe sequence

Safe sequence - is a sequence (ordering) <P1, P2, …, Pn> of all the processes in the systems, such that:

- for each Pi -- the resources that Pi can still request can be satisfied by:
  - > currently available resources, plus
  - > resources held by all the Pj, with j < i (that is, by P1, P2, …, Pi -1)

Intuition behind a safe sequence:

- ifPi resource needs are not immediately available
- then Pi can wait until P1, P2, …, Pi-1 have finished
  - at that stage Pi will be guaranteed to obtain the needed resources so Pi can execute ,return allocated resources ,and terminate

Basic Facts

> If a system is in **safe state ⇒ no deadlocks**

> If a system is in **unsafe state ⇒ possibility of deadlock**

> **Avoidance** ⇒ ensure that a system will never enter an unsafe state

> **Solution**:Whenever a process requests a resource that is available:

- Decide:
  - If the resource can be allocated immediately, or
  - If the process must wait
- Request is granted only if it leaves the system in the safe state

Deadlock Detection

If no deadlock-prevention or deadlock-avoidance algorithm is used

- Then system can enter a deadlock state

In this environment, the system may provide

- Deadlock detection algorithm
- Deadlock recovery algorithm

Example of Detection Algorithm

> Five processes P0 through P4;three resource types A (7 instances), B (2 instances), and C (6 instances)

> Snapshot at time T0:

	Allocation	**Request**	**Available**
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 0 0
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

Sequence <P0, P2, P3, P1, P4> will result in Finish[i] = true for all I

P2 requests an additional instance of type C

	Request
	A B C
P0	0 0 0
P1	2 0 2
P2	0 0 1
P3	1 0 0
P4	0 0 2

State of system?

- Can reclaim resources held by process P0, but insufficient resources to fulfill other processes; requests
- Deadlock exists, consisting of processes P1, P2, P3, and P4

Recovery from Deadlock: Process Termination

> Abort all deadlocked processes

Cons: Very costly – processes could have been running for long time.
They will need to be restarted.

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

Cons: High overhead since the deadlock detection algorithm is called each time one process is terminated

> In which order should we choose to abort?

1. Priority of the process
2. How long process has computed, and how much longer to completion
3. Resources the process has used
4. Resources process needs to complete
5. How many processes will need to be terminated
6. Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

> Resource preemption is another method to recover from deadlocks

> Selecting a victim – which process to choose to preempt its resources?
Minimize "cost".

> Rollback – return to some safe state, restart process for that state.

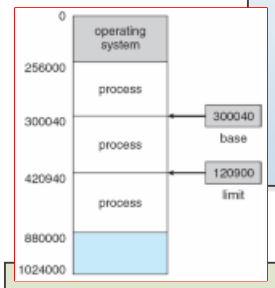
- Process cannot continue "as is", since its resources are preempted
- The rollback state should be such that the deadlock breaks

< **Starvation** – same process may always be picked as victim One solution: include number of rollback in cost factor

#8 Memory Management

Recap: Performance of Various Levels of Storage

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape



> Register access in 1 CPU clock (or less)
  - 0.25 – 0.50 ns (1 nanosec = 10-9 sec)
> Main memory can take many cycles, causing the CPU to stall
  - 80-250 ns (160-1000 times slower)
  - How to solve? – caching
> Cache sits between main memory and CPU registers

Base and Limit Registers

> Protection of memory is required to ensure correct operation
  - Protect OS from processes
  - Protect processes from other processes
> How to implement memory protection?
  - Example of one simple solution using basic hardware
  - A pair of base and limitregisters define the logical address space
> CPU must check every memory access generated in user mode to be sure it is between base and limit for that user

Address Binding

> In most cases, a user program goes through several steps before being executed
  - Compilation, linking, executable file, loader creates a process
  - Some of which may be optional
> Addresses are represented in different ways at different stages of a program's life
  - Each binding maps one address space to another
> Source code -- addresses are usually **symbolic**
  - E.g., variable `count`
> A **compiler** typically **binds** these symbolic addresses to **relocatable** addresses
  - E.g., "14 bytes from beginning of this module"
> Linker or loader will bind relocatable addresses to absolute addresses
  - E.g., 74014

Binding of Instructions and Data to Memory

Address binding of instructions and data to memory addresses can happen at 3 different stages:

1. Compile time:
- If you know at compile time where the process will reside in memory, then **absolute code** can be generated.
- Must **recompile** the code if starting location changes
- Example: MS DOS .com programs

2. Load time:
- Compiler must generate **relocatable code** if memory location is not known at compile time
- If the starting address changes, we need only reload the user code to incorporate this changed value.

3. Execution time:
- If the process can be moved **during its execution** from one memory segment to another, then binding must be delayed **until run time**
- **Special hardware** must be available for this scheme to work
- **Most general-purpose operating systems use this method**

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 (=virtual address)** – generated by the CPU
  - This is what a process sees and uses
- **Physical address** – address seen by the memory unit
  - Virtual addresses are mapped into physical addresses by the system

> **Logical address space**
- is the set of all logical addresses generated by a program

> **Physical address space**
- is the set of all physical addresses generated by a program

> Logical addresses and physical addresses
- Are the same for
  - **compile-time** and load-time address-binding schemes
- **Different** for
  - **execution-time** address-binding scheme

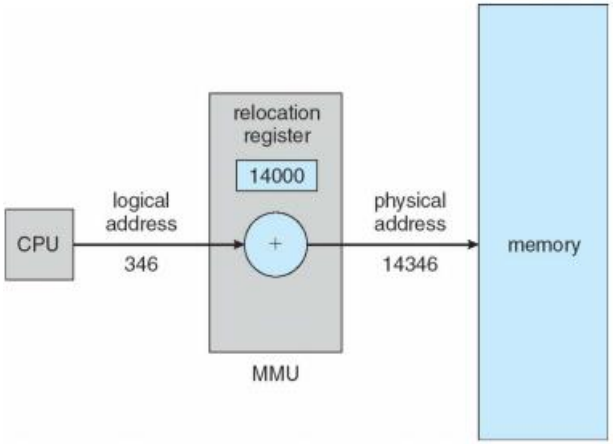
Memory-Management Unit (MMU)

> **Memory-Management Unit (MMU)**
- Hardware device that (at run time) maps virtual to physical address
- Many methods for such a mapping are possible

  Some are considered next

> To start, consider simple scheme:
- The value in the **relocation register** is added to every address generated by a user process at the time it is sent to memory
- **Base register is** now called **relocation register**
- MS-DOS on Intel 80x86 used 4 relocation registers

> The user program deals with **logical** addresses
- It never sees the **real physical** addresses
- Execution-time binding occurs when reference is made to location in memory
- Logical address bound to physical addresses

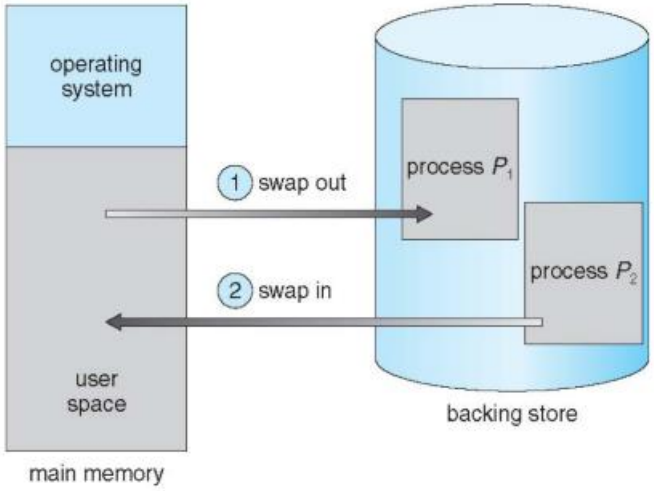


Dynamic Loading

> In our discussion so far, it has been necessary for the entire program and all data of a process to be in physical memory for the process to execute
> **Dynamic Loading** -- routine is not loaded (from disk) until it is called
> Better memory-space utilization; unused routine is never loaded
> All routines kept on disk in relocatable load format
> Useful when large amounts of code are needed to handle infrequently occurring cases
> No special support from the operating system is required
- Implemented through program design
- OS can help by providing libraries to implement dynamic loading

Swapping

> A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
: Total physical memory space of processes can exceed physical memory
: This increases the degree of multiprogramming in a system
> **Backing store** – fast disk
: large enough to accommodate copies of all memory images for all users
: must provide direct access to these memory images
> System maintains a **ready queue** of ready-to-run processes which have memory images on disk or in memory
> **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms;
: lower-priority process swapped out so higher-priority process be loaded and executed
> Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
> Modified versions of swapping are found on many systems
- For example, UNIX, Linux, and Windows
- Swapping normally disabled
- Started if more than threshold amount of memory allocated
- Disabled again once memory demand reduced below threshold



Dynamic Linking

> Some OS'es support only static linking
- **Static linking** – system libraries and program code combined by the loader into the binary program image
> **Dynamic linking**
- Linking is postponed until execution time
- Similar to dynamic loading, but linking, rather than loading, is postponed
- Usually used with **system libraries**, such as **language subroutine libraries**
- Without this, each program must include a copy of its language library (or at least the routines referenced by the program) in the executable image.
- This wastes both disk space and main memory
> **Dynamically linked libraries** are system libraries that are linked to user programs when the programs are run
> With dynamic linking, a **stub** is included in the image for each library routine reference.
> The **stub** is a small piece of code that indicates:
- how to locate the appropriate memory-resident library routine, or
- how to load the library if the routine is not already present
> Stub replaces itself with the address of the routine, and executes the routine
- Thus, the next time that particular code segment is reached, the library routine is executed **directly**, incurring **no cost** for dynamic linking.
- Under this scheme, all processes that use a language library execute only **1 copy** of the library code
> Dynamic linking is particularly useful for libraries
> System also known as **shared libraries**
- Extensions to handle library updates (such as bug fixes)
- A library may be replaced by a new version, and all programs that reference the library will automatically use the new version
  - No relinking is necessary
> **Versioning** may be needed
- In case the new library is incompatible with the old ones
- More than one version of a library may be loaded into memory
  - each program uses its version information to decide which copy of the library to use
- Versions with minor changes retain the same version number, whereas versions with major changes increment the number.
> Unlike dynamic loading, dynamic linking and shared libraries generally require help from the OS
- If the processes in memory are protected from one another, then the OS is the only entity that can check to see whether the needed routine is in another process's memory space
- or that can allow multiple processes to access the same memory addresses
- We elaborate on this concept when we discuss **paging**

Contiguous Allocation

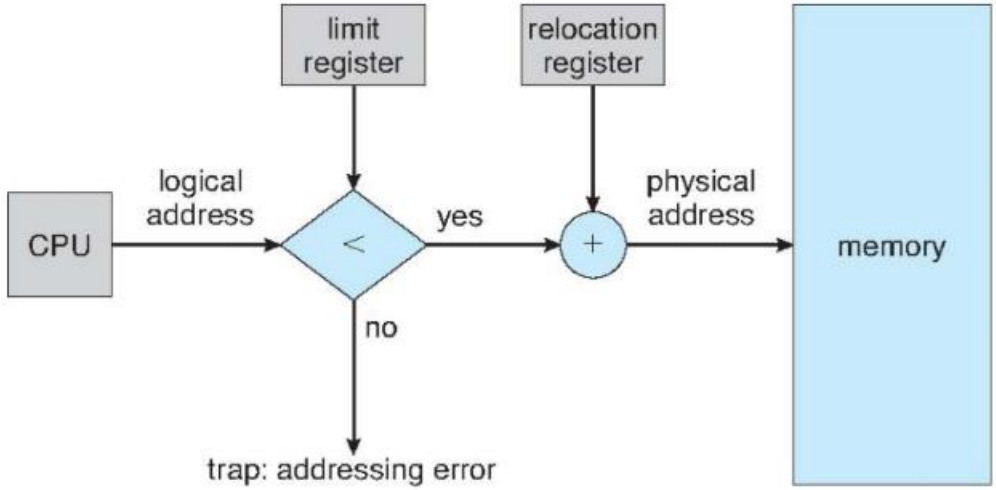
> Main memory must support both OS and user processes
  1. Limited resource, must allocate efficiently
  2. How? -- Many methods

> Contiguous memory allocation is one early method
  each process is contained in a single section of memory that is contiguous to the section containing the next process

> Main memory is usually divided into 2 partitions:
  1. Resident operating system
     - usually held in low memory
  2. User processes
     - usually held in high memory
     - each process contained in single contiguous section of memory

Contiguous Allocation: Memory Protection

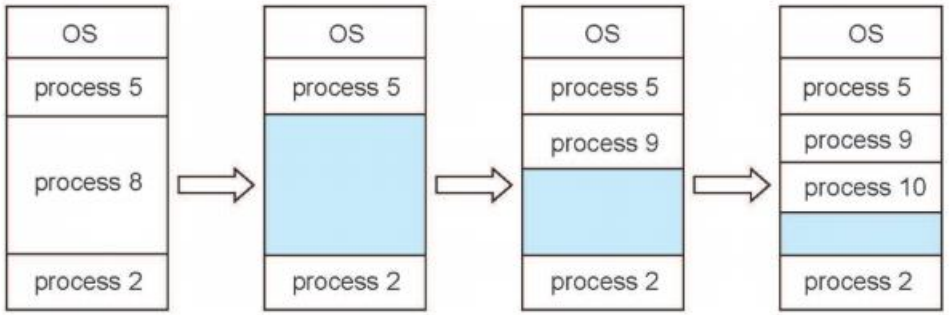
> Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Relocation register contains the value of the smallest physical address for the process
  - Limit register contains range of logical addresses for the process
    : each logical address must be less than the limit register
  - MMU maps logical address dynamically



Contiguous Allocation: Memory Allocation

> Multiple-partition allocation
  - One of the simplest methods
  - Originally used by the IBM OS/360 operating system (called MFT)
    : no longer in use
  - Divide memory into several fixed-sized partitions
  - Each partition may contain exactly 1 process
    : Thus, the degree of multiprogramming is limited by the number of partition
  - When partition is free, process selected from input queue and loaded into free partition
  - When the process terminates, the partition becomes available for another process

> Variable-partition scheme
  - Generalization of the previous method
  - Idea: Variable-partition sizes for efficiency
    : Sized to a given process' needs
  - Initially, all memory is available for user processes and is considered one large block of available memory (a hole).
  - Hole – block of available memory;
    : Holes of various size are scattered throughout memory
  - Operating system maintains information about:
    a) allocated partitions
    b) free partitions (holes)
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Process exiting frees its partition
    : adjacent free partitions combined



Dynamic Storage-Allocation Problem

How to satisfy a request of size n from a list of free holes?

1) First-fit: Allocate the first hole that is big enough
2) Best-fit: Allocate the smallest hole that is big enough
   - > Must search entire list, unless ordered by size
   - > Produces the smallest leftover hole
3) Worst-fit: Allocate the largest hole
   - > Must also search entire list
   - > Produces the largest leftover hole

First-fit and best-fit are better than worst-fit in terms of speed and storage utilization

Fragmentation

Memory allocation can cause fragmentation problems:

1. External Fragmentation
2. Internal Fragmentation

External Fragmentation

> Both the first-fit and best-fit strategies for memory allocation suffer from external fragmentation
> As processes are loaded and removed from memory, the free memory space is broken into little pieces
> External Fragmentation – total memory space exists to satisfy a request, but it is not contiguous
- If all these small pieces of memory were in one big free block instead, we might be able to run several more processes

> Analysis of the first-fit strategy reveals that, given N blocks allocated, another 0.5 N blocks will be lost to fragmentation
- That is, 1/3 of memory may be unusable!
- This is known as 50-percent rule

Some of the solutions to external fragmentation:

1. **Compaction**
2. **Segmentation**
3. **Paging**

Fragmentation: Compaction

> Compaction
- Shuffle memory contents to place all free memory together in 1 large block
- Compaction is possible only if relocation is dynamic, and is done at execution time
  - : If addresses are relocated dynamically, relocation requires only:
    1. moving the program and data, and then
    2. changing the base register to reflect the new base address

> I/O can cause problems
- Latch job in memory while it is involved in I/O
- Do I/O only into OS buffers

Internal Fragmentation

Example:

> Consider a multiple-partition allocation scheme with a hole of **10,002** bytes
> Suppose that the next process requests **10,000** bytes.
> If we allocate exactly the requested block, we are left with a hole of **2** bytes.
> Problem: The overhead to keep track of this hole will be substantially larger than the hole itself

Solution:

> Break the physical memory into **fixed-sized blocks**
> Allocate memory in units based on **block size**.

Issue: With this approach, the memory allocated to a process may be slightly larger than the requested memory.

> The difference between these two numbers is **internal fragmentation**— unused memory that is internal to a partition.
> Example:
- Block size is 4K
- Process request 16K + 2Bytes space
- 5 blocks will be allocated, (4K – 2) bytes are wasted in the last block

Segmentation: a Memory-Management Scheme

Motivation: Do programmers think of memory as a linear array of bytes, some containing instructions and others containing data

> Most programmers would say "no."

> Rather, they prefer to view memory as a collection of **variable-sized segments**

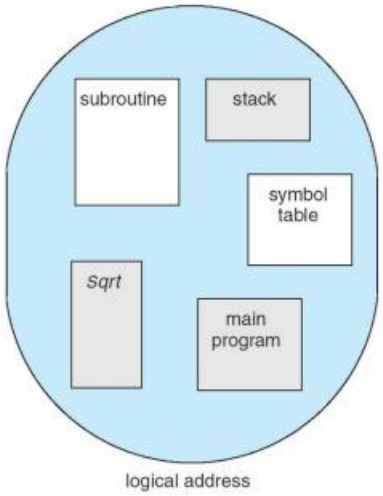
- with no necessary ordering among the segments

> The programmer talks about "the stack," "the math library," and "the main program" without caring what addresses in memory these elements occupy

> The programmer is not concerned with whether the stack is stored before or after the `sqrt()` function

> What if the hardware could provide a memory mechanism that mapped the programmer's view to the actual physical memory?

User's View of a Program



Segmentation

> **Solution**: Segmentation – a memory-management scheme that supports "programmer/user view" of memory

> A logical address space 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, array

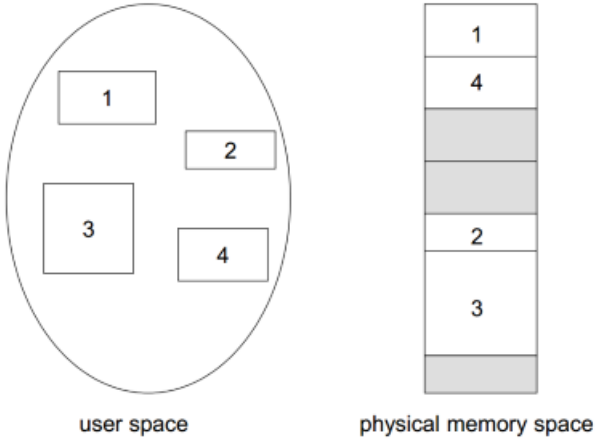
> Each segment has a **name** and a **length**

> The addresses specify both

- the segment name, and
- the offset within the segment

> For simplicity of implementation, segments are **numbered** and are referred to by a **segment number**, rather than by a segment name.

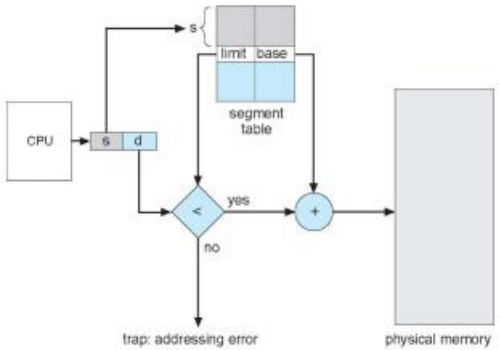
Logical View of Segmentation



Segmentation Architecture

> Logical address consists of a two tuple:
`<segment-number, offset>`
> How to map this 2D user-defined address into 1D physical address?
> **Segment table** – 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** < **STLR**

Segmentation Hardware



Paging

Segmentation
Pros: permits the physical address space of a process to be noncontiguous
Cons: can suffer from external fragmentation and needs compaction
- Any alternatives to it?
- **Paging** -- another memory-management scheme

Benefits of Paging
> Physical address space of a process can be noncontiguous
> Process is allocated physical memory whenever the latter is available
> Avoids external fragmentation
> Avoids problem of varying sized memory chunks

Main Idea of Paging

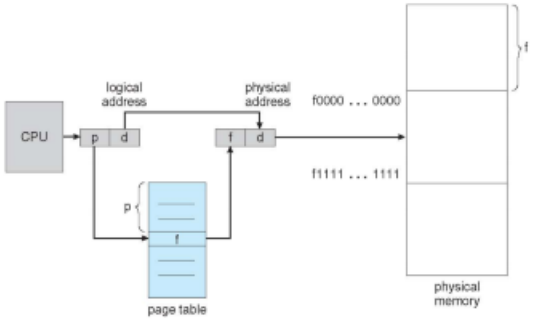
> Divide **physical memory** into **fixed-sized** blocks called **frames**
- Size is power of 2
- Between 512 bytes and 16 Mbytes
> Divide **logical memory** into blocks of same size called **pages**
> Keep track of all free frames
> To run a program of size N pages, need to find N free frames and load program
> Set up a **page table** to translate logical to physical addresses
- **One table per process**
> Still have internal fragmentation

Address Translation Scheme

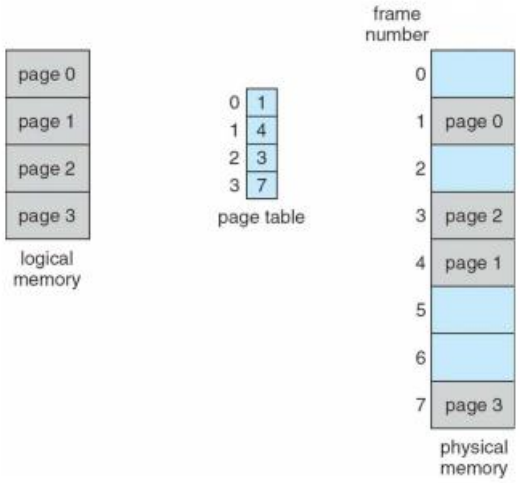
Address generated by CPU is divided into:
> Page number (p) – used as an index into a page table which contains base address of each page in physical memory
> Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit
> For given logical address space 2^m and page size 2^n

page number	page offset
p	d
m -n	n

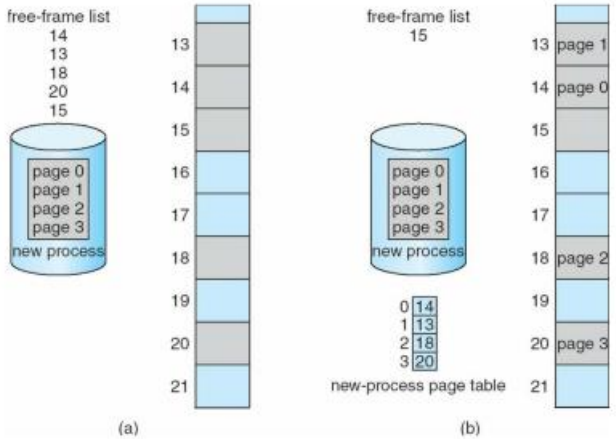
Paging Hardware



Paging Model of Logical and Physical Memory



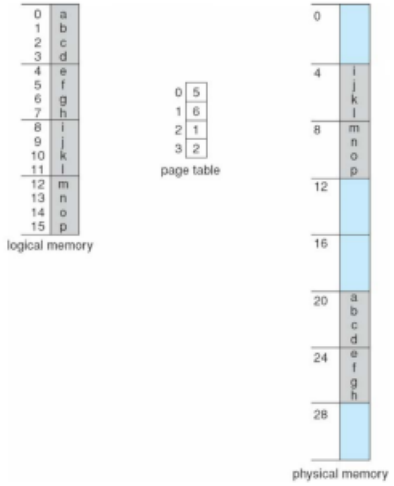
Free Frames



Before allocation

After allocation

Paging Example



n=2 and *m*=4 32-byte memory and 4-byte pages

Process view and physical memory now very different

By implementation a process can only access its own memory

Paging (Cont.)

> Calculating internal fragmentation
- Page size = 2,048 bytes
- Process size = 72,766 bytes
- 35 pages + 1,086 bytes
- Internal fragmentation of 2,048 - 1,086 = 962 bytes

> Worst case fragmentation = 1 frame – 1 byte

> On average fragmentation = 1 / 2 frame size

> So small frame sizes are desirable?
- Not as simple, as each page table entry takes memory to track
- Page sizes growing over time
  : Solaris supports two page sizes – 8 KB and 4 MB

Implementation of Page Table

> Page table is kept in main memory

> **Page-table base register (PTBR)**
- Stores the **physical address** of page table
- Changing page tables requires changing only this one regist
  : Substantially reduces context-switch time

> **Page-table length register (PTLR)**
- Indicates the size of the page table

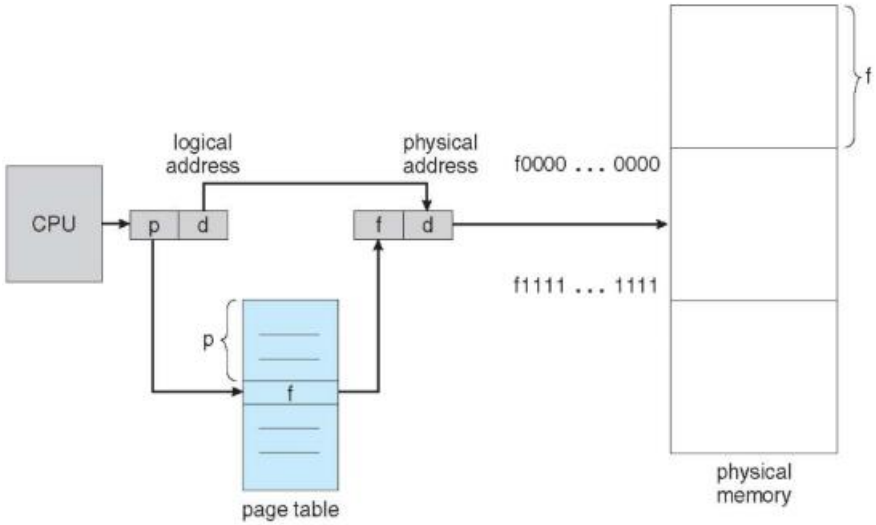
The 2 memory access problem

Problem:

In this scheme every data/instruction access requires 2 memory accesses:

1. One for the page table and (to get the frame number)
2. One for the data / instruction

Efficiency is lost



> Solution: (Use of TLB's)

- Use of a special fast-lookup hardware cache, called:
  - : associative memory, or
  - : translation look-aside buffers (TLBs)

> TLB

- Caches (p,f) tuples for frequently used pages
  - : That is, the mapping from p to the corresponding f
- Small
- Very fast

Associative Memory

> Associative memory – parallel search

Page #	Frame #

> Address translation (p, d)

- **If** p is in TLB **then** get the frame # out
  - : Hardware searches in parallel all entries at the same time
  - : Very fast
- **else** get the frame # from page table in memory

Implementation of Page Table (Cont.)

> Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry

- ASID uniquely identifies each process to provide address-space protection for that process
- Otherwise need to flush at every context switch

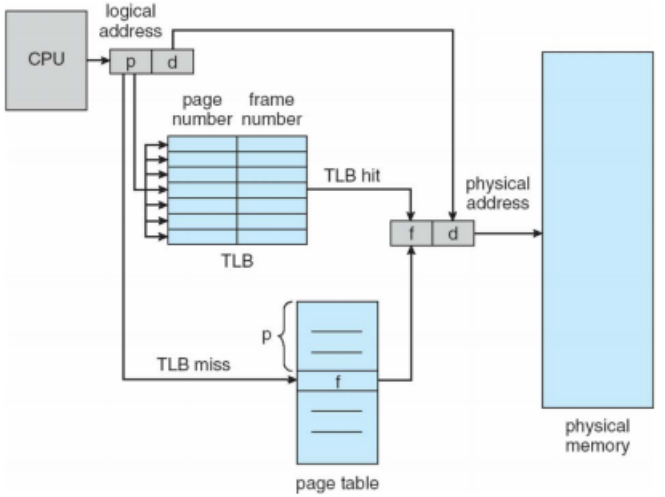
PID	Page#	Frame#

>TLBs typically small (64 to 1,024 entries)

> On a TLB **miss**, value is loaded into the TLB for faster access next time

- Replacement policies must be considered
- Some entries can be **wired down** for permanent fast access

Paging Hardware With TLB



Effective Access Time

> Associative Lookup = ε time unit
- Can be < 10% of memory access time

> Hit ratio = α
- Percentage of times that a page number is found in the TLB
- Hit ratio is related to number of associative registers in TLB

> Effective Access Time (EAT)

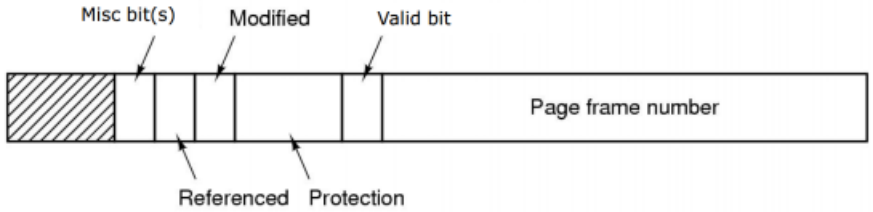
$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$

$= 2 + \varepsilon - \alpha$

// 1 memory access plus TLB access time, or 2 memory accesses +TLB

> Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
- EAT = 0.80 x 120 + 0.20 x 220 = 140 ns

> Consider more realistic hit ratio α = 99%, ε = 20ns for TLB search, 100ns for memory access
- EAT = 0.99 x 120 + 0.01 x 220 = 121 ns



Typical Page Table Entry

Memory Protection

> Memory protection in paged environment accomplished by **protection bits** associated with **each frame**

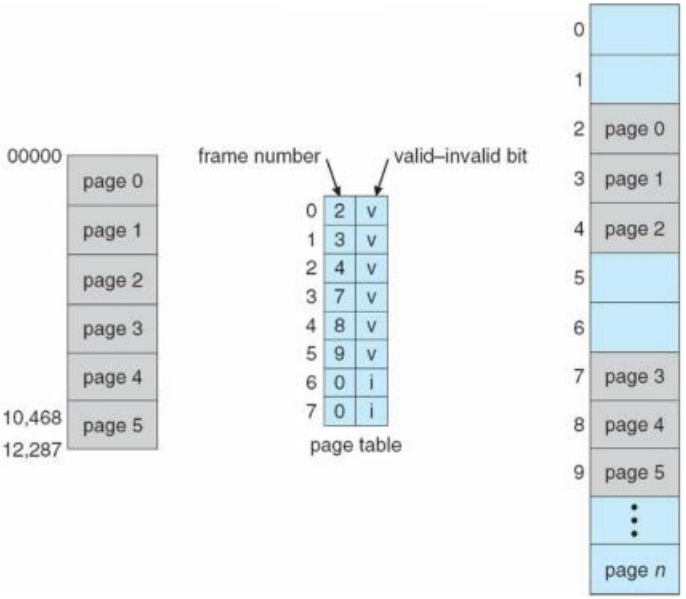
> For example, protection bit to indicate if
1) **read-only** or
2) **read-write** access is allowed
Can also add more bits to indicate page **execute-only**, and so on

> Normally, these bits are kept in the page table.

> **Valid-invalid** bit attached to each entry in the page table:
- "valid" indicates that associated page is in process' logical address space, and is thus a legal page
- "invalid" indicates that the page is not in the process' logical address space

> Some system have **page-table length register (PTLR)**
- Can also be used to check if address is valid

> Any violations result in a trap to the kernel



Valid (v) or Invalid (i) Bit In A Page Table

Shared Pages

> An advantage of paging is the possibility of **sharing** data and common code
> **Shared code**
- Particularly important in a time-sharing environment
  : Ex: A system that supports 40 users, each executes a text editor
- A single copy of read-only (**reentrant**) code shared among processes
  : For example, text editors, compilers, window systems
  : Reentrant code is non-self-modifying code: never changes during exec.
- This is similar to multiple threads sharing the same process space
- Each process has **its own copy** of registers and data storage to hold the data for the process's execution
- The data for two different processes will, of course, be different.

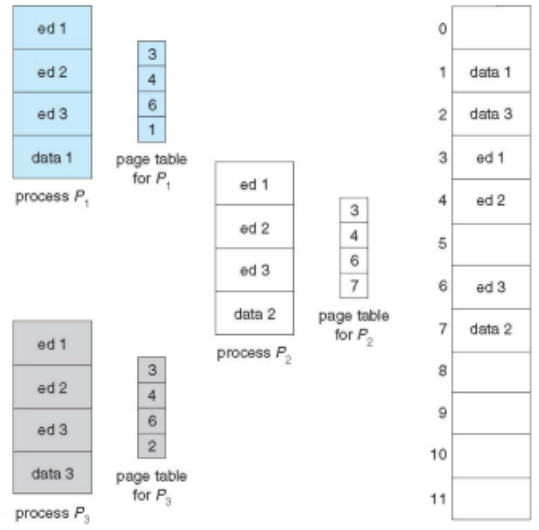
> **Shared data**
- Some OSes implement shared memory suing shared pages.

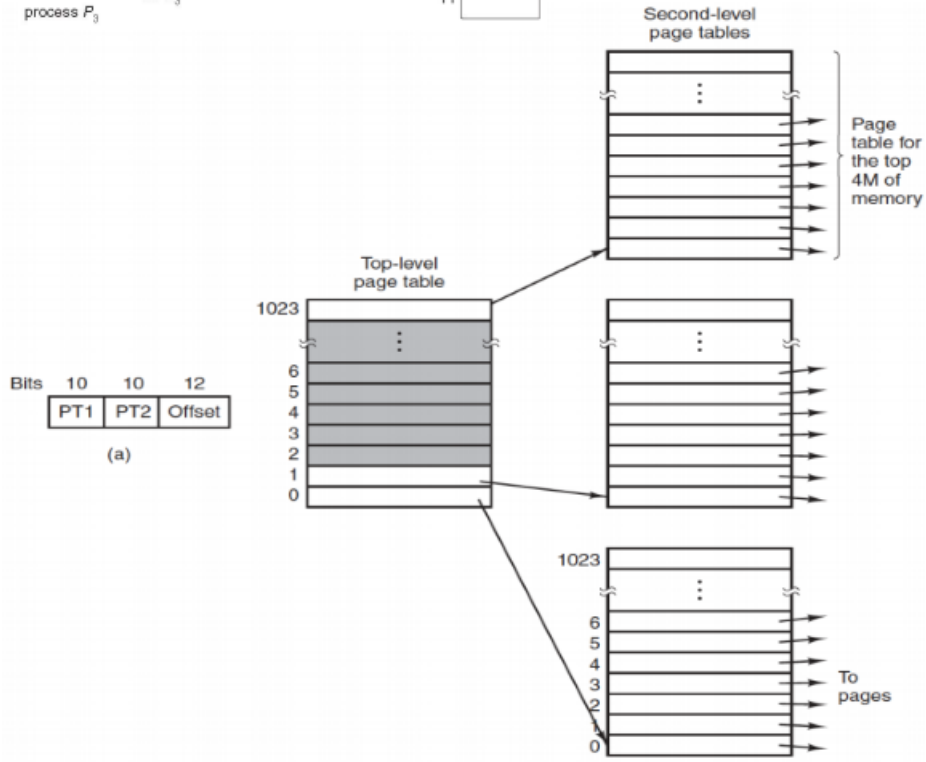
> **Private code and data**
- Each process keeps a separate copy of the code and data
- The pages for the private code and data can appear anywhere in the logical address space

Structure of the Page Table

Memory structures for paging can get huge using straightforward methods
> Consider a 32-bit logical address space as on modern computers
> Page size of 1 KB (2^{10})
> Page table would have 4 million entries (2^{32} / 2^{10}= 2^{22})
> **Problem**: If each entry is 4 bytes -> 16 MB of physical address space / memory for page table alone
  # **This is per process**
  # That amount of memory used to cost a lot
  # Do **not** want to allocate that contiguously in main memory
> **Solution**: One simple solution to this problem is to divide the page table into smaller pieces
> We can accomplish this division in several ways, e.g.:
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables

Shared Pages Example



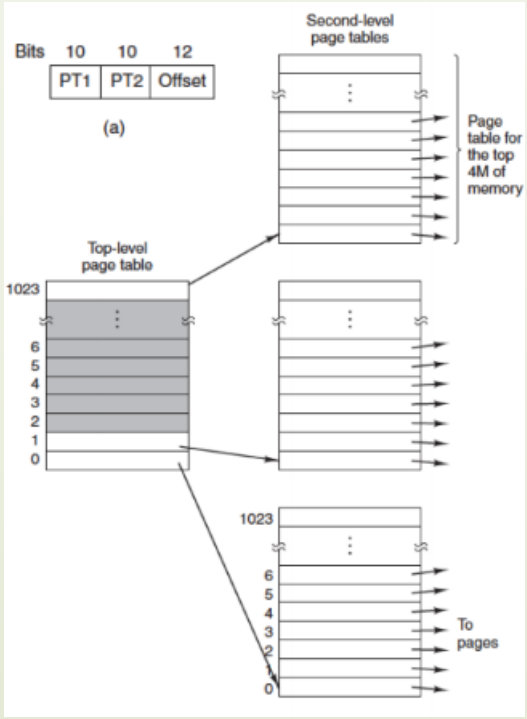


Hierarchical Page Tables

Break up the logical address space into multiple page tables
A simple technique is a two-level page table

Two-Level Paging Example

> 32-bit virtual address is partitioned
  # 10-bit PT1 field
  # 10-bit PT2 field
  # 12-bit offset
> 12-bit offset => pages are 4K(=2^{12}) and 2^{20} of them
> The secret to the multilevel page table method is to avoid keeping all the page tables in memory all the time.
  - In particular, those that are not needed should not be kept around
> Suppose, that a process needs 12 MB
  - the bottom 4 MB of memory for program text
  - the next 4 MB for data, and
  - the top 4 MB for the stack
  - Hence, a gigantic hole that is not used
    : in between the top of the data and the bottom of the stack



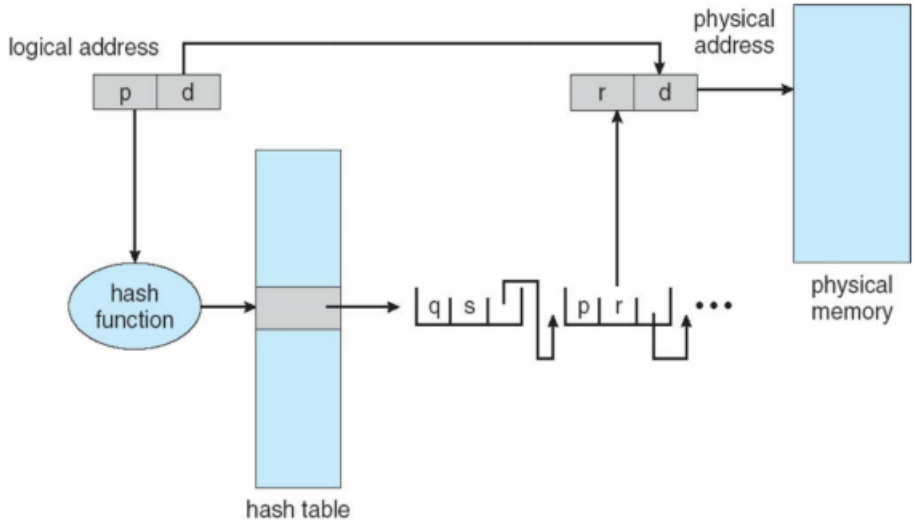
Two-Level Paging

> The top-level page table, with 1024 entries, corresponding to the 10-bit PT1 field.
> When a virtual address is presented to the MMU, it first extracts the PT1 field and uses this value as an index into the top-level page table
> Each of these 1024 entries in the top-level page table represents 4 MB
  # 4 GB (i.e., 32-bit) virtual address space has been chopped into 1024 chunks
  # 4 GB / 1024 = 4 MB
> The entry located by indexing into the toplevel page table yields the address or the page frame number of a **second-level page table**.
> Entry 0 of the top-level page table points to the page table for the program text
> Entry 1 points to the page table for the data
> Entry 1023 points to the page table for the stack
> The other (shaded) entries are not used
  @ No need to generate page tables for them
  @ **Saving lots of space!**
> The PT2 field is now used as an index into the selected second-level page table to find the page frame number for the page itself.

Hashed Page Tables

> Common in address spaces > 32 bits
> The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
> Each element contains
  (1) the virtual page number
  (2) the value of the mapped page frame
  (3) a pointer to the next element
> Virtual page numbers are compared in this chain searching for a match
  If a match is found, the corresponding physical frame is extracted

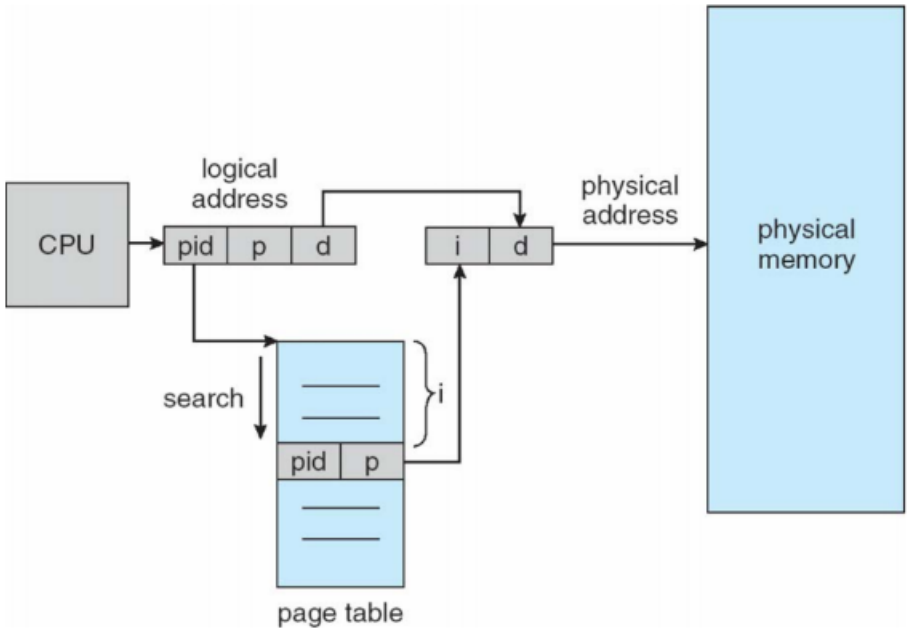
Hashed Page Table



Inverted Page Table

> **Motivation**: Rather than each process having a page table and keeping track of all possible **logical** pages, track all **physical** pages
> One entry for each real page of memory
  - The entry keeps track of which (process, virtual page) is located in the page frame
> **Pros**: tends to save lots of space
> **Cons**: virtual-to-physical translation becomes much harder
> When process n references virtual page p, the hardware can no longer find the physical page by using p as an index into the page table
> Instead, it must search the entire inverted page table for an entry (**n**, **p**).
> Furthermore, this search must be done on every memory reference, not just on page faults
> Searching a 256K table on every memory reference is slow
> Other considerations:
  - TLB and hash table (key: virtual address) is used to speed up accesses
  - Issues implementing shared memory when using inverted page table

Inverted Page Table Architecture



#9 Virtual-Memory Management

Code needs to be in memory to execute, but **entire program rarely used**
- > Example: Error code, unusual routines, large data structures
- > Entire program code is not needed at the same time

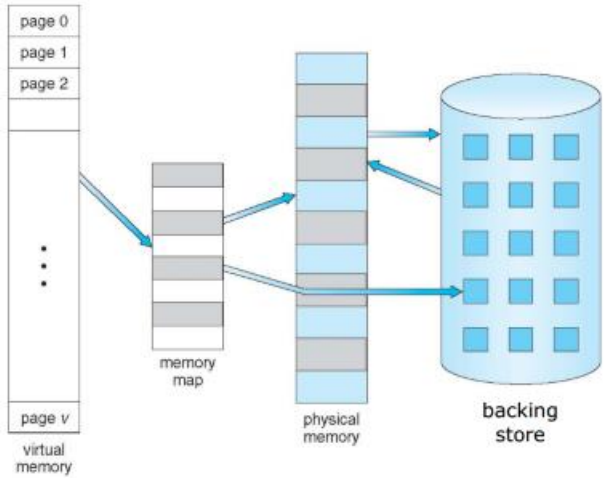
Consider advantages of the ability to execute partially-loaded program
- > Program no longer constrained by limits of physical memory
- > Each program takes less memory while running
  - * Thus, **more programs can run at the same time**
  - * **Increased CPU utilization** and throughput with no increase in response time or turnaround time
- > Less I/O needed to load or swap programs into memory
  - * Thus each user program runs faster

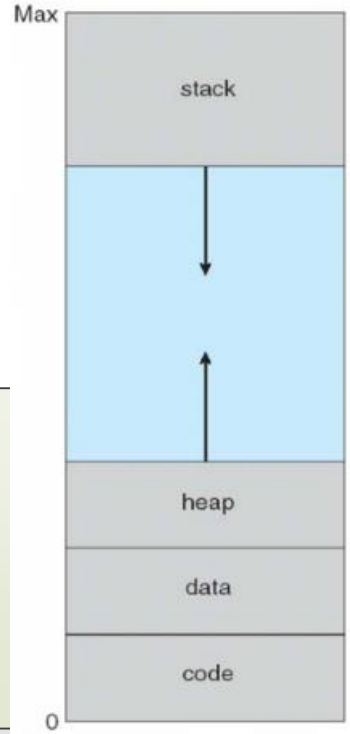
Virtual memory – separation of user logical memory from physical memory
- > Only part of the program needs to be in memory for execution
- > Logical address space can therefore be much larger than physical address space
- > Allows address spaces to be shared by several processes
- > Allows for more efficient process creation
- > More programs running concurrently
- > Less I/O needed to load or swap processes
- > Virtual memory makes the task of programming much easier
  - * the programmer no longer needs to worry about the amount of physical memory available;
  - * can concentrate instead on the problem to be programmed.

Virtual address space – **logical** view of how process is stored in memory
- > Usually start at address 0, contiguous addresses until end of space
- > Meanwhile, physical memory organized in page frames
- > MMU must map logical to physical

Virtual memory can be implemented via:
- > Demand paging
- > Demand segmentation

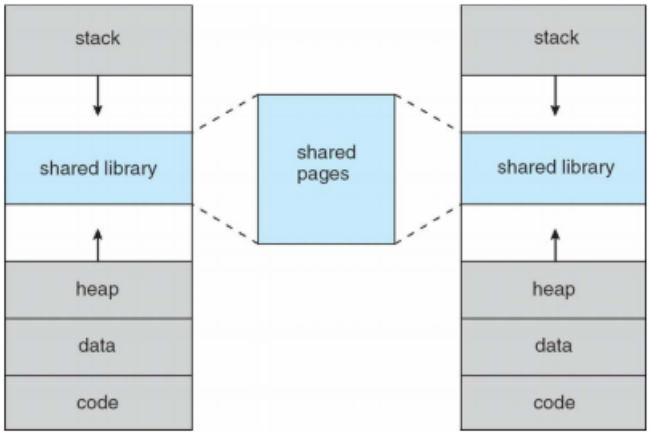
Virtual Memory That is Larger Than Physical Memory





Virtual-address Space
- > Usually design logical address space for stack to start at Max logical address and grow "down" while heap grows "up"
  - - Maximizes address space use
  - - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- > Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
- > System libraries shared via mapping into virtual address space
- > Shared memory by mapping pages into virtual address space

Shared Library Using Virtual Memory



Policies for Paging/ Segmentation

(1) Fetch Strategies

When should a page or segment be brought into primary memory from secondary (disk) storage?

> Demand Fetch

> Anticipatory Fetch

(2) Placement Strategies

When a page or segment is brought into memory, where is it to be put?

> Paging – trivial

> Segmentation - significant problem

(3) Replacement Strategies

Which page/segment should be replaced if there is not enough room for a required page/segment?

Basic Concepts

- When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again.
- Instead of swapping in a whole process, the pager brings only those pages into memory
- Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed
- How to determine that set of pages?
  > Need **new** MMU functionality to implement demand paging
- If pages needed are already **memory resident**
  > No difference from non demand-paging
- If page needed and not memory resident
  > Need to detect and load the page into memory from storage
  * Without changing program behavior
  * Without programmer needing to change code

Demand Paging

> Bring a page into memory only when it is needed.

1. Less I/O needed
2. Less Memory needed
3. Faster response
4. More users

> Demand paging is kind of a "lazy" swapping

- But deals with pages instead of programs
  swapping deals with entire programs, not pages
- **Lazy swapper**– never swaps a page into memory unless page will be needed
- **Pager** -- swapper that deals with pages

> The first reference to a page will trap to OS with a page fault

> OS looks at another table to decide

- Invalid reference – abort
- Just not in memory – bring it from memory

Valid-Invalid Bit

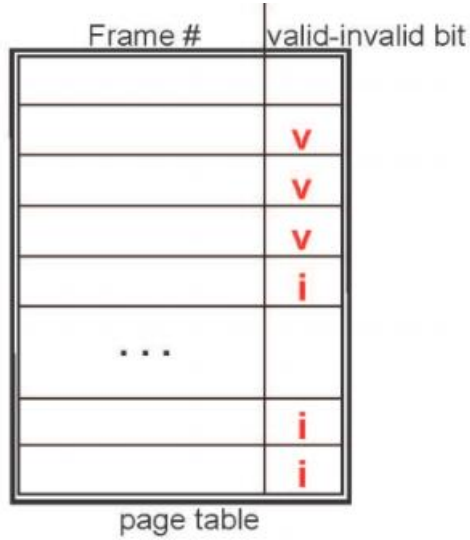
> With each page table entry a valid–invalid bit is associated:

- v ⇒ in-memory – memory resident
- i ⇒ not-in-memory

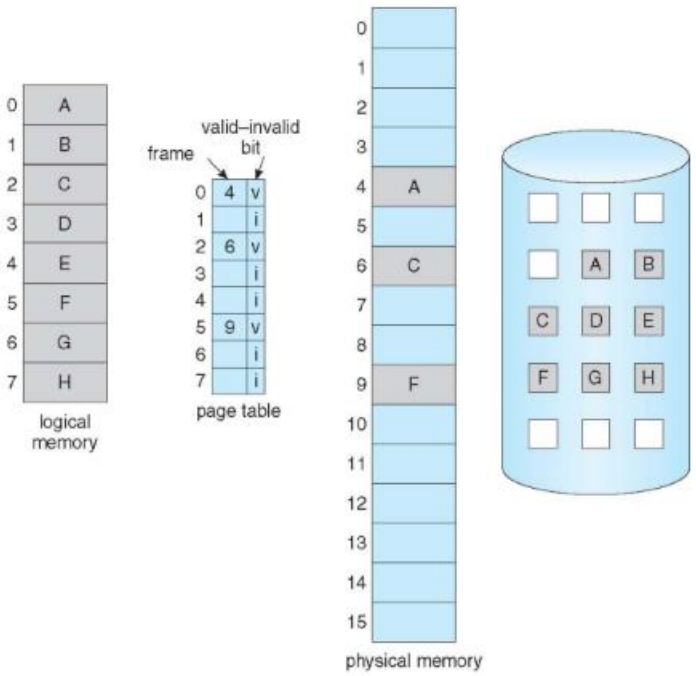
> Initially, valid–invalid bit is set to i on all entries

> Example of a page table snapshot:

> During MMU address translation, if valid_invalid_bit =i ⇒ page fault



Page Table When Some Pages Are Not in Main Memory



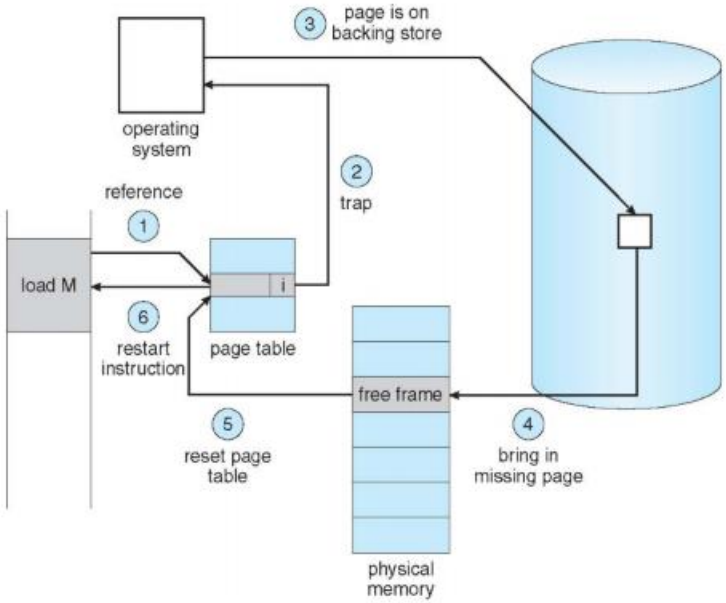
Page Fault

> If the process tries to access a page that was not brought into memory,
> Or tries to access any "invalid" page:
- That will trap to OS, causing a **page fault**
- Such as when the first reference to a page is made

Handling a page fault

1. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort the process
   - Just not in memory => continue
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now is in memory
   - Set validation bit = **v**
5. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



What happens if there is no free frame?

> **Page replacement** if not free frame
- Find some page in memory that is not really in use and swap it.
  - * Need a **page replacement algorithm**
  - * Performance Issue - need an algorithm which will result in minimum number of page faults.
- Same page may be brought into memory many times

Aspects of Demand Paging

> **Pure demand paging**
- Extreme case
- Start process with **no pages in memory**
- OS sets instruction pointer to first instruction of process,
  * non-memory-resident -> page fault
  * page faults for all other pages on their first access

> A given instruction could access **multiple pages**
- Causing **multiple** page faults
  * E.g., fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
- Pain decreased because of **locality of reference**

> Hardware support needed for demand paging
- Page table with valid / invalid bit
- Secondary memory (swap device with **swap space**)
- Instruction restart

Performance of Demand Paging

Stages in Demand Paging (worst case)

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
   i) Wait in a queue for this device until the read request is serviced
   ii) Wait for the device seek and/or latency time
   iii) Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of Demand Paging (Cont.)

> Three major activities during a Page Fault
1. Service the page fault interrupt
   Fast - just several hundred instructions needed
2. Read in the page
   Very slow - about 8 millisecs., including hardware and software time
   1 millisec = 10^{-3} sec, 1 microsec = 10^{-6} sec, 1 ns = 10^{-9} sec
3. Restart the process
   Fast – similar to (1)

> Page Fault Rate $0 \le p \le 1$
if p = 0 no page faults
if p = 1, every reference is a fault

> Effective Access Time (EAT)
EAT = (1 – p) * memory access +
+ p * (page fault overhead + swap page out + restart overhead)

Demand Paging Example

> Memory access time = 200 nanoseconds
> Average page-fault service time = 8 milliseconds
> EAT = (1 – p) * 200 + p (8 milliseconds)
= (1 – p) * 200 + p * 8,000,000
= 200 + p * 7,999,800
> If one access out of 1,000 causes a page fault, then
EAT = 8.2 microseconds (10-6 of a second)
This is a slowdown by a factor of 40!!
> If want performance degradation < 10 percent
220 > 200 + 7,999,800 * p
20 > 7,999,800 x p
p < .0000025
Less than 1 page fault in every 400,000 memory accesses

Page Replacement

> **Over-allocation**
  * While a user process is executing, a page fault occurs
  * OS determines where the desired page is residing on the disk
    - but then finds that there are no free frames on the free-frame list
    - all memory is in use

> We need to prevent **over-allocation** of memory
  * by modifying page-fault service routine to include **page replacement**

> **Page replacement**
  * If no frame is free, we find one that is not currently being used and free it.
    - as explained next
  * completes separation between logical memory and physical memory
  * large virtual memory can be provided on a smaller physical memory

> Use **modify (dirty) bit** to reduce overhead of page transfers
  * The dirty bit is set for a page when it is modified
  * Only modified pages are written to disk
    - No need to save unmodified pages, they are the same

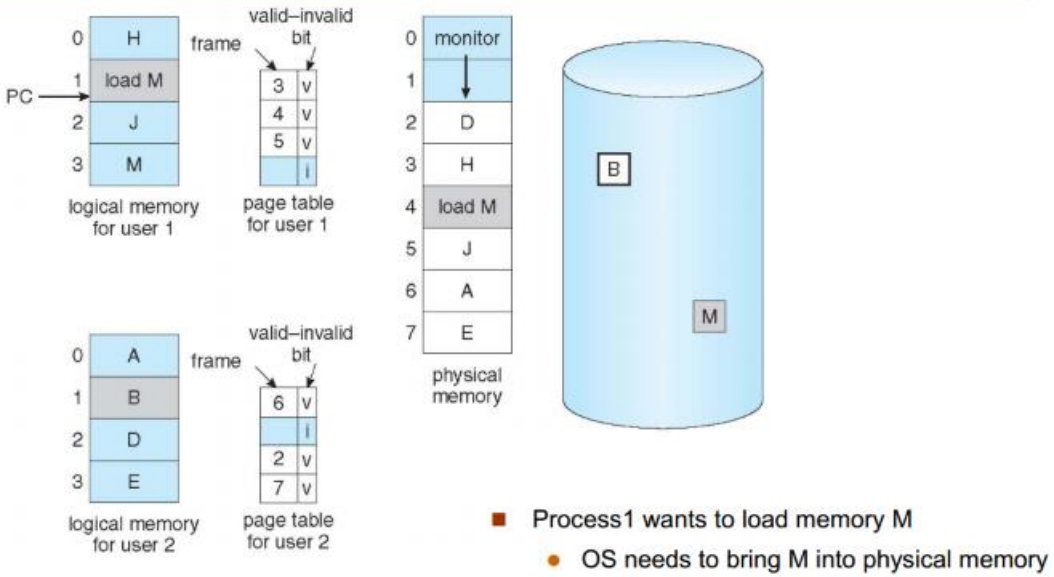
Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
   - **if** (a free frame exist) **then** use it
   - **else**
     use a **page replacement algorithm** to select a **victim frame**
     write victim frame to disk, if dirty
3. Bring the desired page into the (newly) free frame;
   - update the page and frame tables accordingly
4. Continue the process by restarting the instruction that caused the trap

Note: now potentially 2 page transfers for page fault

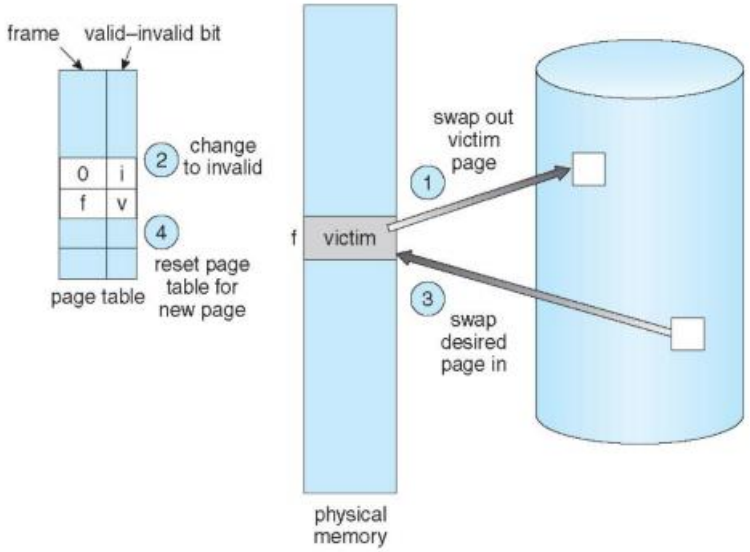
> Increasing EAT

Need For Page Replacement



- Process1 wants to load memory M
  - OS needs to bring M into physical memory
- **Problem:** no free frames
  - OS needs to do page replacement

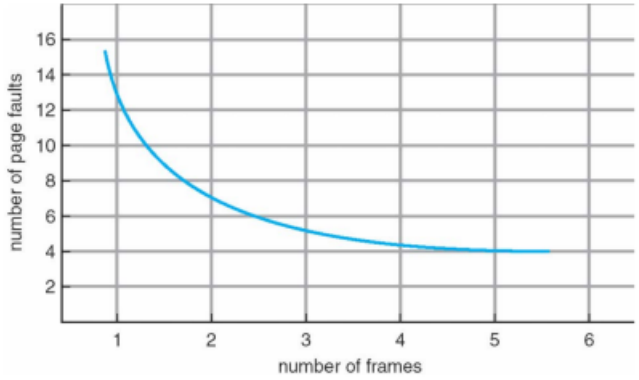
Page Replacement



Page and Frame Replacement Algorithms

> **Frame-allocation algorithm determines**
  * How many frames to give each process
  * Which frames to replace
> **Page-replacement algorithm**
  * Want the lowest page-fault rate on both first access and re-access
> Evaluating different page replacement algorithms
  * By running them on a particular string of memory references (reference string) and
  * Computing the number of page faults on that string
> String is just page numbers, not full addresses
> Repeated access to the same page does not cause a page fault
> Results depend on number of frames available
> In our examples, the **reference string** of referenced page numbers is
  7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Graph of Page Faults Versus The Number of Frames



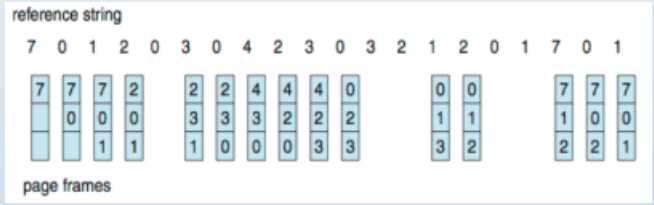
Page Replacement Strategies

> The Principle of Optimality
  Replace the page that will not be used again the farthest time into the future
> Random Page Replacement
  Choose a page randomly
> FIFO - First in First Out
  Replace the page that has been in memory the longest.
> LRU - Least Recently Used
  Replace the page that has not been used for the longest time.
> LFU - Least Frequently Used
  Replace the page that is used least often
> NUR - Not Used Recently
  An approximation to LRU
> Working Set
  Keep in memory those pages that the process is actively using

First-In-First-Out (FIFO) Algorithm

Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

3 frames (3 pages can be in memory at a time per process)



15 page faults

> How to track ages of pages?

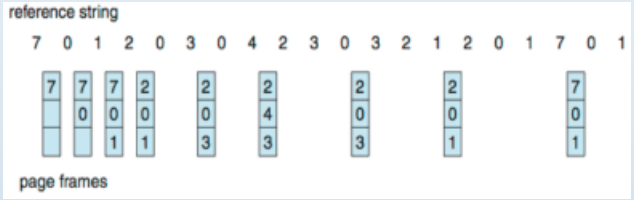
Just use a FIFO queue

> FIFO algorithm

Easy to implement

Often, not best performing

Optimal Algorithm



> Called **OPT** or **MIN**

> Replace page that will not be used for longest period of time

9 page faults is optimal for the example

> How do you know this?

You don't, can't read the future

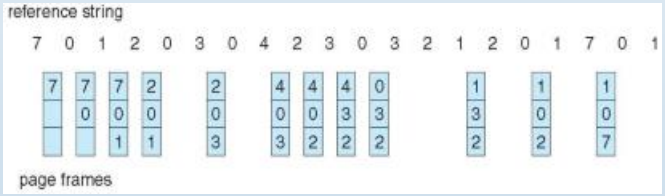
> Used for measuring how well other algorithms performs – against the theoretical optimal solution

Least Recently Used (LRU) Algorithm

> Use past knowledge rather than future

> Replace page that has not been used in the most amount of time

> Associate time of last use with each page



> 12 faults – better than FIFO but worse than OPT

> Generally a good algorithm and frequently used

> But how to implement?

Implementation of LRU Algorithm

> **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 replaced
  - LRU looks at the counters to find the smallest value
  - Search through the table is needed

> **Stack implementation**

* Keep a stack of page numbers
  - Most recently used – at the top
  - Least recently used – at the bottom
  - Implemented as a double link list
    - Easier to remove entries from the middle
    - Head pointer
    - Tail pointer
* Page referenced
  - move it to the top
  - requires 6 pointers to be changed
* No search for replacement
  - But each update is more expensive

> LRU and OPT don't have Belady's Anomaly

#10 File System

> Most visible aspect of operating system
> Mechanism storage and access data and programs of operating system and users
> Consist 2 parts:
    1) collection of files, each storing related data
    2) directory structure

File Attributes:

> **Name** symbolic file name, readable form
> **Identifier** unique tag, identifies file within file system (non human readable name)
> **Type** needed for system support different types files
> **Location** pointer to device and location of file on device
> **Size** current file size (bytes, words or blocks)
> **Protection** access-control information determines who can read, write, execute
> **Time, data, user identification** kept for creation, last modified, last use. Useful for protection, security and usage monitoring

File Operations

> **Create file** Two steps – find space in file system, entry for new file made in directory
> **Writing file** make system call specifying both name of file and information to be written
> **Reading file** system call specify name of file and where in memory put next block of file
> **Repositioning within file** directory searched appropriate entry, current-file-position pointer repositioned to given value
> **Deleting file** search directoryfor named file, release all file space to reuse by other files and erase directory entry
> **Truncating file** erase contents of file but keep attributes, file reset to length zero

Common file types

Executable	*exe, com, bin, none*	ready-to-run machine language program
Object	*obj, o*	compiled, machine language, not linked
Source code	*c, cc, jave, perl, asm*	source code in various languages
Batch	*bat, sh*	commands to command interpreter
Markup	*xml, html, tex*	textual data, documents
Word processor	*xml, rtf, docx*	variuos word-processor formats
Library	*lib, a, so, dll*	libraries of routines for programmers
Print or view	*gif, pdf, jpg*	ASCII or binary file in format for printing or viewing
Archive	*rar, zip, tar*	related files grouped into one file (compressed)
Multimedia	*mpeg, mov, mp3, mp4, avi*	binary file containing audio or A/V information

Information associated with open file

> File pointer
    - system keep track last read-write location as current-file-position pointer
> File-open count
    - as files closed, OS reuse open-file table entries
> Disk location of file
    - require system modify data within file, information needed to locate file on disk kept in memory so system not read from disk for each operation
> Access rights
    - process open file in access mode
    - information stored per-process table so operating system allow or deny subsequent I/O requests

#11 Implementing File System

Basics: File-System Structure

> File structure
- Logical storage unit
- Collection of related information

> **File system** resides on secondary storage (disks)
- Provided user interface to storage
  mapping logical to physical
- Provides efficient and convenient access to disk by allowing data to be stored, located, and retrieved easily

> Disk provides in-place rewrite and random access
- It is possible to read a block from the disk, modify the block, and write it back into the same place
- I/O transfers performed in **blocks** of 1 or more **sectors**
  A sector is usually 512 bytes

> **File control block (FCB)** – storage structure consisting of information about a file

Directory Implementation

> The choice of the directory implementation is crucial for the efficiency, performance, and reliability of the file system

> **Linear list** of file names with pointer to the data blocks
- **Pros**: Simple to program
- **Cons**: Time-consuming to execute -- Linear search time
- **Solutions**:
  - Keep sorted + binary search
  - Use indexes for search, such as B+ tree

> **Hash Table** – linear list with hash data structure
- Hash on file name
- Decreases directory search time
- **Collisions**– situations where two file names hash to the same location
- Each hash entry can be a linked list - resolve collisions by adding new entry to linked list

Allocation of Disk Space

> Low level access methods depend upon the disk allocation scheme used to store file data
- Contiguous Allocation
- Linked List Allocation
- Indexed Allocation

> allocation method refers to how disk blocks are allocated for files

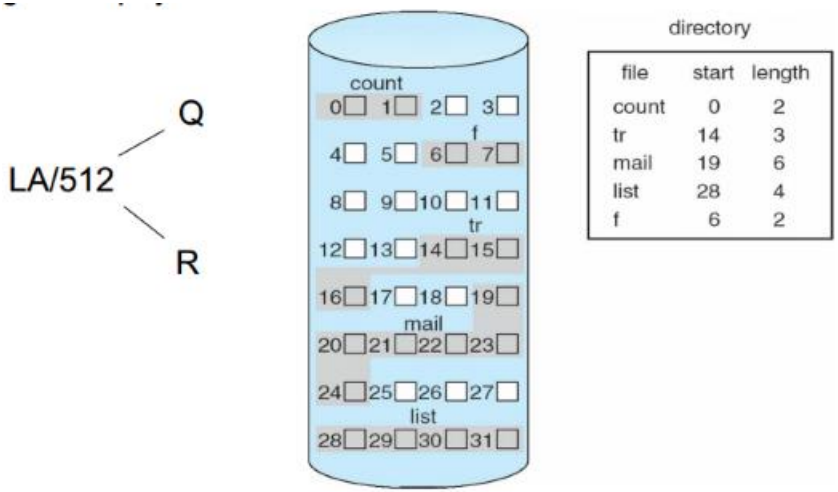
Contiguous allocation

each file occupies set of contiguous blocks

> Best performance in most cases
- Commonly, hardware is optimized for sequential access
- For a magnetic disk – reduces seek time, head movement

> Simple – only info required:
- starting location (block #) and
- length (number of blocks)

> Problems include
- finding space for file
- knowing file size
- external fragmentation
- need for **compaction off-line(downtime) or on-line**
  Can be costly



> For simplicity, assume 1 block = 1 sector
> Mapping from logical to physical - <Q, R>
> Block to be accessed = Q + starting address
> Displacement into block = R

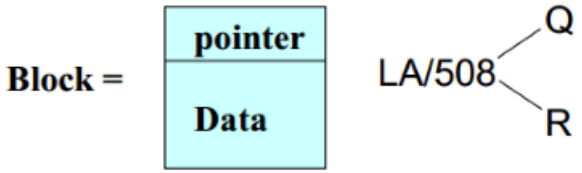
Allocation Methods -Linked

each file a **linked list** of blocks

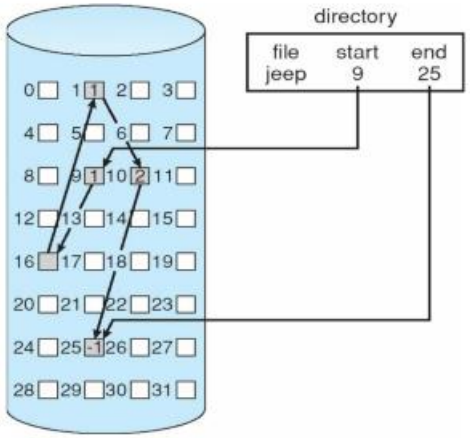
> Blocks may be scattered anywhere on the disk
> Each block contains **pointer** to the next block
- Disk space is used to store pointers,
  if disk block is 512 bytes, and pointer (disk address) requires 4 bytes, user sees 508 bytes of data.
- Pointers in list not accessible
- File ends at **nil** pointer

> **Pros**: No external fragmentation
- No compaction

> **Cons**: Locating a block can take many I/Os and disk seeks



> Mapping from logical address o physical
> Block to be accessed is the Q-th block in the linked chain of blocks representing the file
> Displacement into block = R + 4



> Slow - defies principle of locality
- Need to read through linked list nodes sequentially to find the record of interest

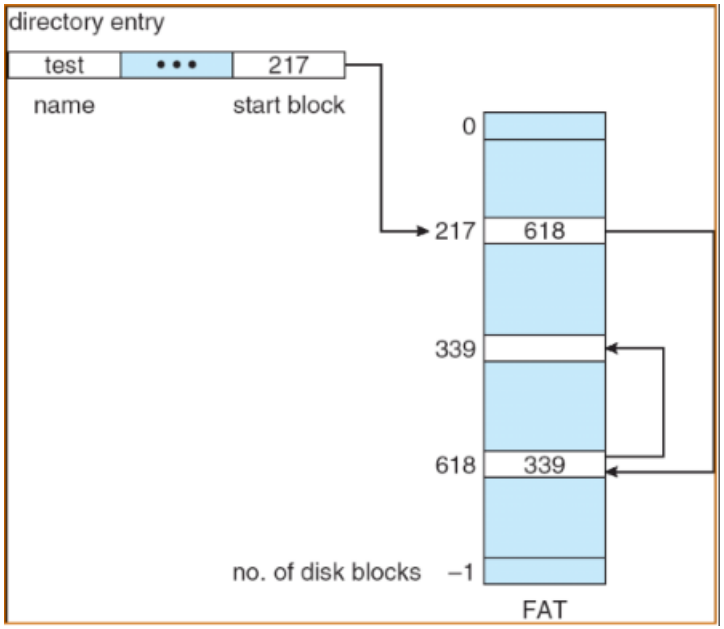
> Not very reliable
- System crashes can scramble files being updated

> Important variation on linked allocation method
- File-allocation table (FAT) - disk-space allocation used by MS-DOS and OS/2.

File Allocation Table (FAT)

> Instead of link in each block...
- put all links in one table
- the File Allocation Table (FAT)

> One entry per physical block in disk
- Directory points to first & last blocks of file
- Each block points to next block (or EOF)
- Unused block: value =0



> Advantages
- Advantages of Linked File System
- FAT can be cached in memory
- Searchable at CPU speeds, pseudo-random access

> Disadvantages
- Limited size, not suitable for very large disks
- FAT cache describes entire disk
  not just open files!
- Not fast enough for large databases

> Used in MS-DOS, early Windows systems

Disk Defragmentation

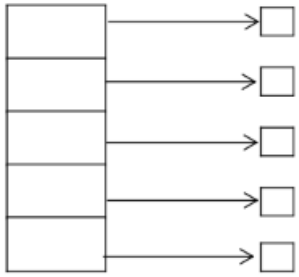
> Re-organize blocks in disk so that file is (mostly) contiguous
> Link or FAT organization preserved
> Purpose:
  To reduce disk arm movement during sequential accesses

Allocation Methods -Indexed

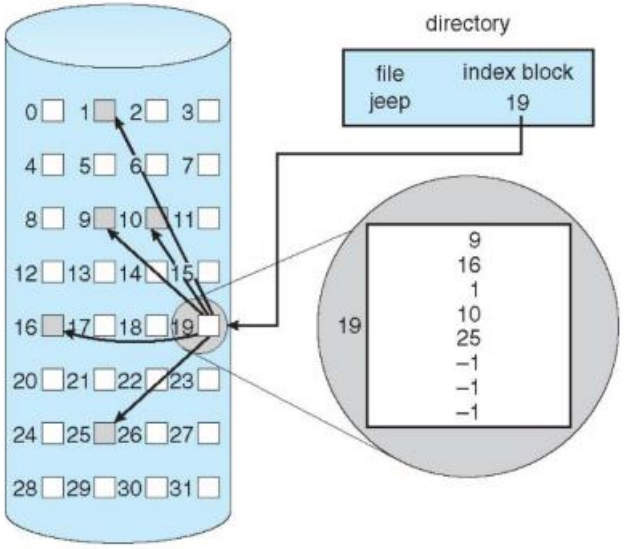
> If FAT is not used, linked allocation cannot support efficient direct access,
- since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order.
- How to solve this?

> **Indexed allocation**
- Each file has its own index block(s) of pointers to its data blocks

Logical view



Example of Indexed Allocation



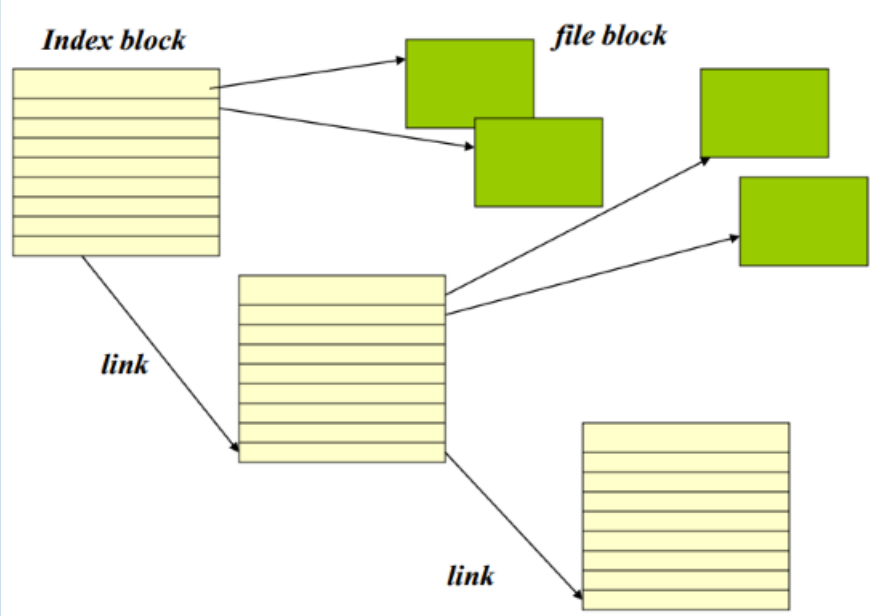
Indexed Allocation (Cont.)

> Dynamic access without external fragmentation, but have overhead of index block
> Mapping from logical to physical
- in a file of maximum size of 256K bytes and
- block size of 512 bytes
- We need only 1 block for index table

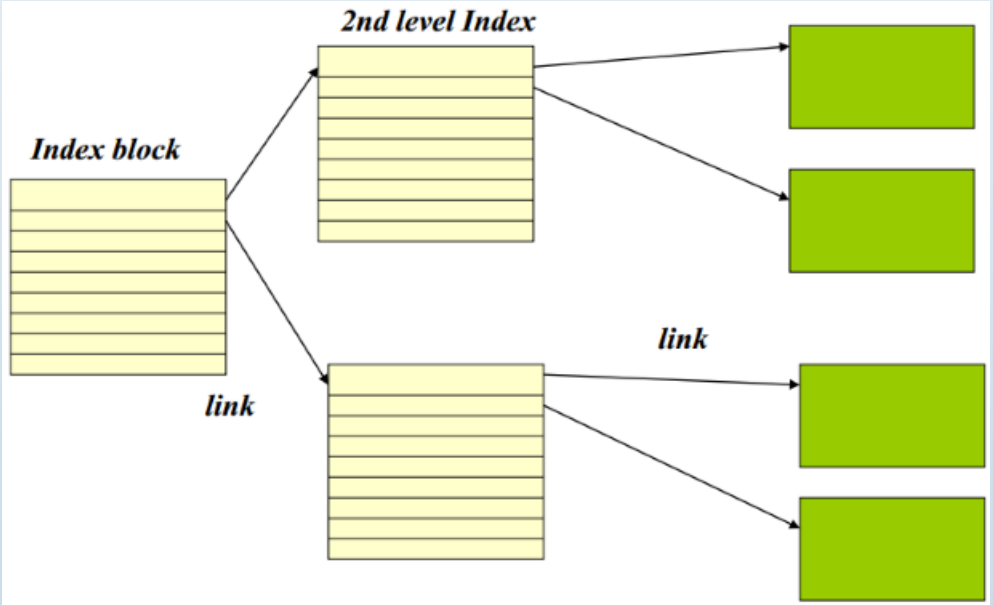
Q = displacement into index table
R = displacement into block

> A single index block might not be able to hold enough pointers for a large file
- Several schemes to deal with this issue (e.g., linked, multi-level, combined)

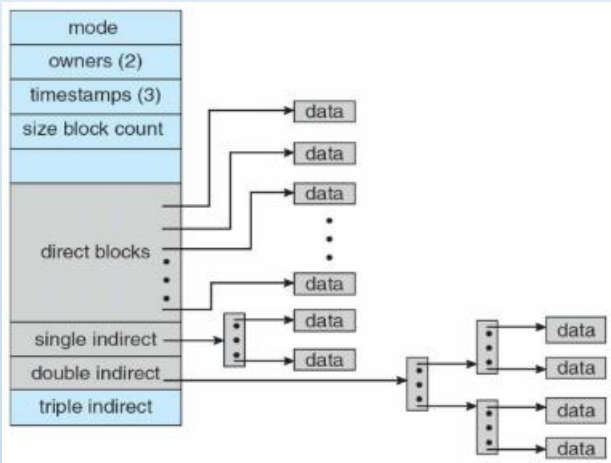
> **Linked scheme** – Link blocks of index table (no limit on size)



> **Two-level index** (generalizes to multi-level)
4K blocks could store 1,024 four-byte pointers in outer index – 1,048,567 data blocks and file size of up to 4GB



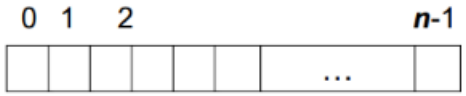
> **Combined Scheme**: used in UNIX UFS
> 4K bytes per block, 32-bit addresses



Free-Space Management

> File system maintains free-space list to track available blocks/clusters
- (Using term "block" for simplicity)
- Several ways to implement

> Bit vector or bit map (n blocks)



$$bit[i] = \begin{cases} 1 \Rightarrow block[i] \text{ free} \\ 0 \Rightarrow block[i] \text{ occupied} \end{cases}$$

Bit Vector (contd.)

> **Pros**
- Relative simplicity
- Efficiency in finding the first n consecutive free blocks on the disk
  Easy to get contiguous files

> **Example**: one technique for finding the first free block
- Sequentially check each word in the bit map if it is **not** 0
  0-valued word contains only 0 bits
  represents a set of allocated blocks
- First non-0 word is scanned for the first 1 bit
  which is the location of the first free block

> The calculation of this free block number is
- (number of bits per word) * (number of 0-value words) + offset of first 1 bit

Free-Space Management (Cont.)

> Bit map requires extra space

Example:
  block size = 4KB = 2^{12} bytes
  disk size = 2^{40} bytes (1 terabyte)
  $n = 2^{40}/2^{12} = 2^{28}$ bits (or 32MB)
  if clusters of 4 blocks -> 8MB of memory

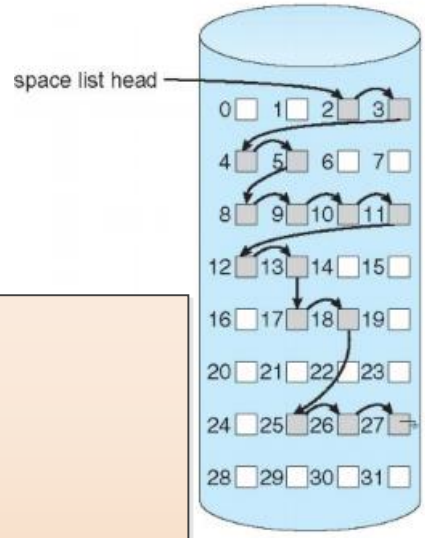
Example: BSD File syste

Linked Free Space List on Disk

> **Linked list (free list)** -- keep a linked list of free blocks

> **Pros**:
- No waste of space
- No need to traverse the entire list (if # free blocks recorded)

> **Cons**:
- Cannot get contiguous space easily
- not very efficient because linked list needs traversal



Free-Space Management (Cont)

> **Linked list of indices – Grouping**
- A modification of the free-list approach
- Keep a linked list of index blocks
- Each index block contains:
  1. addresses of free blocks and
  2. a pointer to the next index block
- **Pros**: A large number of free blocks can now be found quickly
  Compared to the standard linked-list approach

> **Counting**
- Linked list of contiguous blocks that are free
- Free list node contains pointer + number of free blocks starting from that address

Efficiency and Performance

> In general, the efficiency of a file system dependents on:
- Disk allocation and directory algorithms
- Types of data kept in file's directory entry
- Fixed-size or varying-size data structures used

> Even after basic file-system algorithms have been selected, we can still improve performance in several ways

> Performance improved by:
- Keeping data and metadata close together – generic principle
  Do not want to perform a lot of extra I/O just to get file information
- **Using buffer** cache – separate section of main memory for frequently used blocks

> Optimize caching - depending on the access type of the file
- E.g., a file being accessed **sequentially** then use **read-ahead**
- **Read-ahead** -- a requested page and **several subsequent pages** are read and cached
  These pages are likely to be requested after the current page is processed
  Retrieving these data from disk in one transfer and caching them saves considerable time

Recovery

> Files and directories are kept both in main memory and on disk
- Care must be taken to ensure that a system failure does not result in loss of data or in data inconsistency
- How to recover from such a failure

> **Consistency checking** – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
- Can be slow and sometimes fails

> Use system programs to **back up** data from disk to another storage device

> Recover lost file or disk by **restoring** data from backup

#14 System Protection

Goals of Protection

- To prevent malicious misuse of system by users or programs
- To ensure each shared resource used only in accordance with system policies, which may be set either by system designers or by system administrators.
- To ensure errant programs cause minimal amount damage possible.
- Note protection systems only provide mechanisms enforcing policies and ensuring reliable systems. It is up to administrators and users to implement those mechanisms effectively.

Principles of Protection

- Principle of least privilege dictates programs, users, and systems given just enough privileges to perform their tasks.
- Ensures failures do least amount of harm and allow least of harm to be done.
- example, if program needs special privileges to perform task, better make it a SGID program with group ownership of "network" or "backup" or some other pseudo group, rather than SUID with root ownership. This limits the amount of damage that can occur if something goes wrong.
- Typically each user given own account, has only enough privilege to modify their own files.
- Root account should not be used for normal day to day activities

  The System Administrator should also have an ordinary account, and reserve use of the root account for only those tasks which need the root privileges

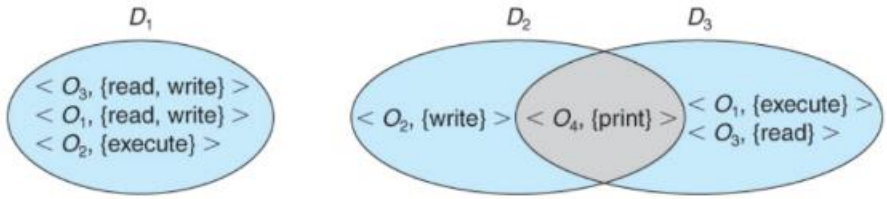
Domain of Protection

> Computer can be viewed as collection processes and objects (both HW & SW)
> Need to know principle states process should only have access objects it needs to accomplish its task, only in modes for which it needs access and only during time frame when it needs access.
> The modes available for a particular object may depend upon its type

Domain Structure

> Protection domain specifies resources that process may access.
> Each domain defines set objects and types operations may be invoked on each object.
> Access right is ability to execute operation on object.
> Domain is defined as set of < object, { access right set } > pairs.

Note some domains may be disjoint while others overlap.



> Association between process and domain may be static or dynamic
> If association static, need-to-now principle requires a way of changing contents of the domain dynamically.
> If association dynamic, then there needs to be a mechanism for domain switching

Domains may be realized in different fashions as users, or as processes, or as procedures. E.g. if each user corresponds to a domain, then that domain defines the access of that user, and changing domains involves changing user ID

#15 System Security

Most common types of violations include:

1) Breach of Confidentiality Theft of private or confidential information, such as creditcard numbers, trade secrets, patents, secret formulas, manufacturing procedures, medical information, financial information, etc.

2) Breach of Integrity Unauthorized modification of data, which may have serious indirect consequences. For example a popular game or other program's source code could be modified to open up security holes on users systems before being released to the public.

3) Breach of Availability Unauthorized destruction of data, often just for the "fun" of causing havoc and for bragging rites. Vandalism of web sites is a common form of this violation

4) Theft of Service Unauthorized use of resources, such as theft of CPU cycles, installation of daemons running an unauthorized file server, or tapping into the target's telephone or networking services

5) Denial of Service, DOS Preventing legitimate users from using the system, often by overloading and overwhelming the system with an excess of requests for service

One common attack is masquerading, in which the attacker pretends to be a trusted third party. A variation of this is the maninthe-middle, in which the attacker masquerades as both ends of the conversation to two targets.
A replay attack involves repeating a valid transmission. Sometimes this can be the entire attack, (such as repeating a request for a money transfer), or other times the content of the original message is replaced with malicious content.

Levels at which a system must be protected:

1) Physical Site contain systems physically secured entry by intruders. Both machine rooms and terminals

2) Human Authorization assure appropriate users access system.

Via social engineering:

i) Phishing – email or web site misleading user entering confidential information

ii) Dumpster diving – information gained in trash, phone books, notes containing passwords

iii) Password cracking – divining user passwords, watching typed, words from common dictionaries

3) Operating Systems System protect itself from accidental or purposed security breaches

4) Network Intercepting data from Internet, wireless connections

Program Threats

1) Trojan Horse

> Program secretly performs maliciousness in addition to its visible actions.
> Deliberately written, others are result of legitimate programs that have become infected with viruses
> Dangerous opening for Trojan horses is long search paths, paths which include current directory (".") as part of the path. If dangerous program having same name as legitimate program placed anywhere on path, then unsuspecting user may be fooled into running wrong program by mistake.
> Another classic Trojan Horse is a login emulator, records users account name and password, issues "password incorrect" message, logs off system. User then tries again (with a proper login prompt), logs in successfully, and doesn't realize information has been stolen.

Solutions to Trojan Horses:

> System print usage statistics on logouts,
> Require typing of non-trappable key sequences such as Control-Alt-Delete in order to log in.

Spyware is version of Trojan Horse included in "free" software downloaded off Internet. Generate popup browser windows, accumulate information about user and deliver it to some central site.

2) Trap Door

> Programmer deliberately inserts security hole use later to access system.
> Once system in untrustworthy state, system can never be trusted again. Backup tapes may contain copy of some cleverly hidden back door.
> Could be inserted into compiler, any programs compiled would contain security hole. Inspection of code being compiled would not reveal any problems

3) Logic Bomb

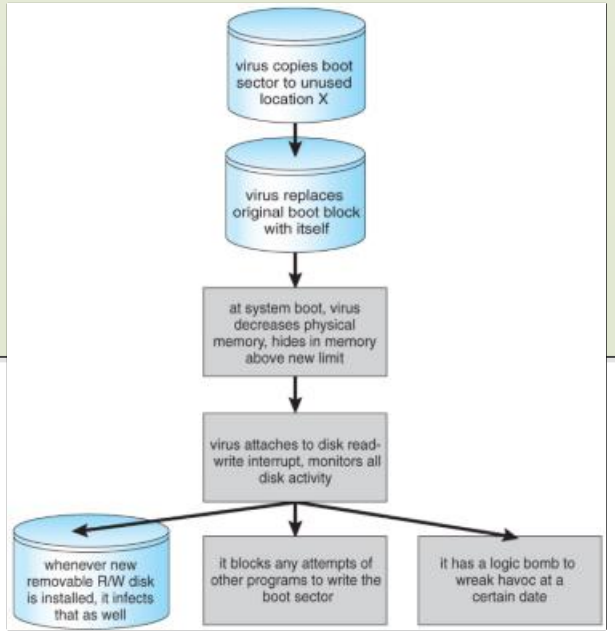
> Code not designed to cause havoc all the time, only when certain circumstances occurs, as particular date or time or noticeable event.
> Example DeadMan Switch, designed check certain person logging in every day, if don't log in for long time (presumably fired), then logic bomb goes off and opens security holes or causes problems.

4) Stack and Buffer Overflow

> Exploits bugs in system code that allows buffers to overflow.
    i) Overflow input field until it writes into stack
    ii) Overwrite current return address on stack with address of expoit code
    iii) Write simple code for next space in stack include commands attacker want executed

Viruses

> Fragment code embedded in legitimate program, designed to replicate itself, wreaking havoc.
> Infect PCs than UNIX, latter systems have limited authority to modify programs or access critical system structures
> Viruses delivered to systems in a virus dropper, some form of Trojan Horse, via email or unsafe downloads.
> Take many forms



Forms of viruses (program threats)

1) File attaches itself to executable file, run virus code first then jump to start of original program. Termed parasitic, do not leave new files on system, original program is still fully functional.
2) Boot occupies boot sector, runs before OS loaded. Known as memory viruses, operation reside in memory, do not appear in file system.
3) Macro exist as script run automatically by certain macro-capable programs (MS Word or Excel). Can exist in word processing documents or spreadsheet files.
4) Source code look for source code and infect it in order to spread.
5) Polymorphic change every time they spread - not underlying functionality, just their signature, by which virus checkers recognize them.
6) Encrypted travel in encrypted form to escape detection. Self-decrypting, allows to infect other files.
7) Stealth avoid detection modifying parts of system that could be used to detect it.
8) Tunneling attempt avoid detection inserting themselves into interrupt handler chain, or into device drivers.
9) Multipartite attack multiple parts of system, such as files, boot sector, and memory.
10) Armored are coded to make hard for antivirus researchers to decode and understand. Files associated with viruses are hidden, protected, given innocuous looking names such as "...".

System and Network Threats

1) Worms is process that uses the fork / spawn process to make copies of itself in order to wreak havoc on system.
Consume system resources, often blocking out other, legitimate processes.
Worms that propagate over networks especially problematic, can tie vast amounts network resources and bring down large-scale systems.
2) Port Scanning technically not an attack, search for vulnerabilities to attack.
Systematically attempt to connect to every known network port on some remote machine, attempt make contact.
Once determined computer listening to particular port, next determine what daemon is listening, whether or not it is version containing known security flaw be exploited.
Because port scanning easily detected and traced, it launched from zombie systems, i.e. previously hacked systems used without knowledge or permission of rightful owner.
Important protect "innocuous" systems and accounts as well as those contain sensitive information or special privileges.
Also port scanners available administrators use check own systems, report weaknesses but do not exploit weaknesses or cause problems.
3) Denial of Service
Denial of Service (DOS) attacks do not attempt access or damage systems, but clog up badly that cannot be used for any useful work.
Tight loops repeatedly request system services obvious form attack.
DOS attacks can also involve social engineering, chain letters that say "send this immediately to 10 of your friends, and then go to a certain URL", clogs up Internet mail system also the web server to which everyone is directed.
Security systems that lock accounts after number failed login attempts are subject to DOS repeatedly attempt logins to all accounts with invalid passwords to lock up all accounts.
DOS is not result of deliberate maliciousness.