2. Operating System Structures

ECE30021/ITP30002 Operating Systems

Types of System Calls

- Process control
- File management
- Device management
- Information maintenance
- Communication

Process Control: Load/Execution

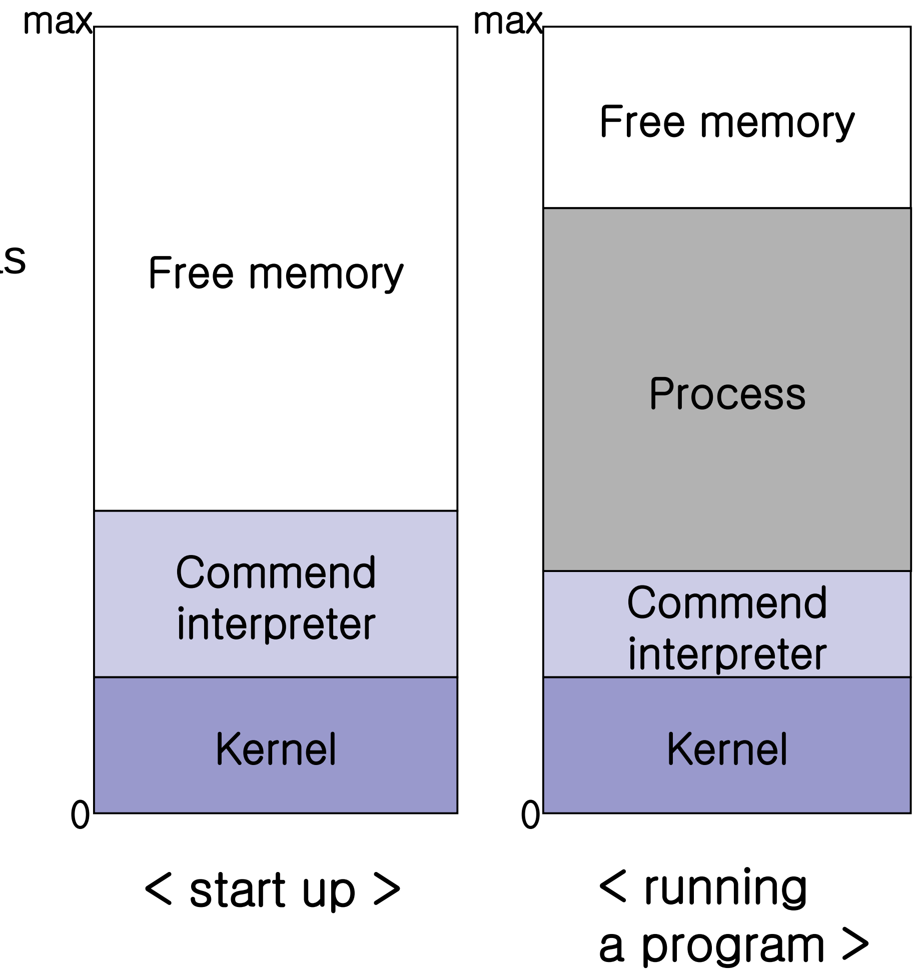
- A program can load/execute another program.

  Ex) Command interpreter

- Then, the parent program can
  - Be lost (replaced by the child program)
  - Be saved (paused)
  - Continue execution: multi-programming
    - Create process/submit job

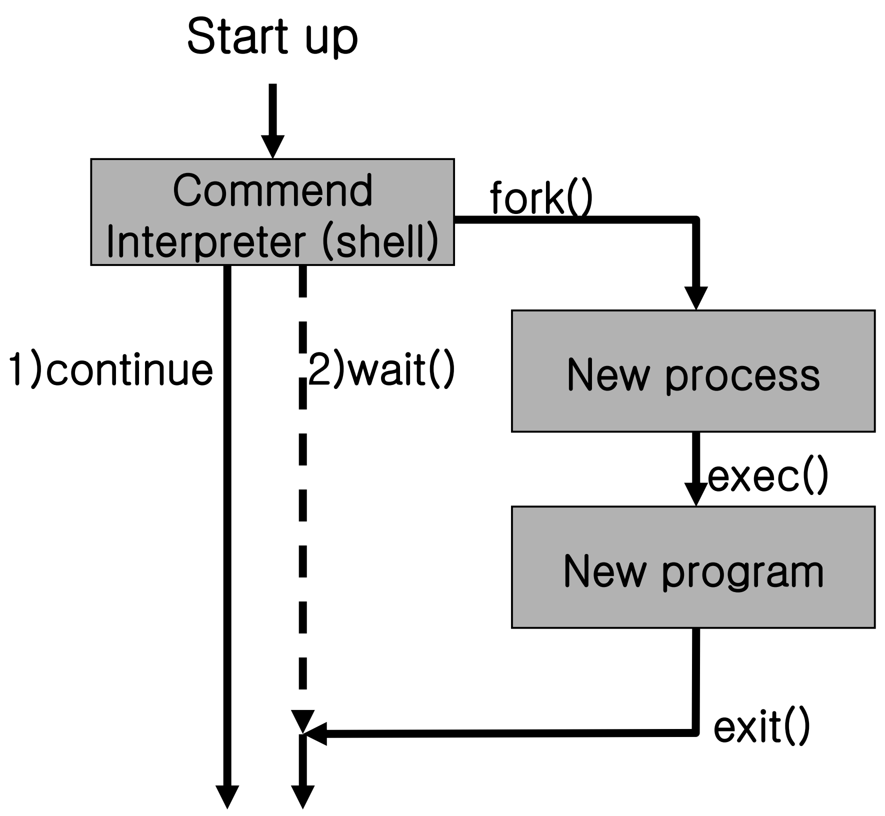
Example: MS-DOS

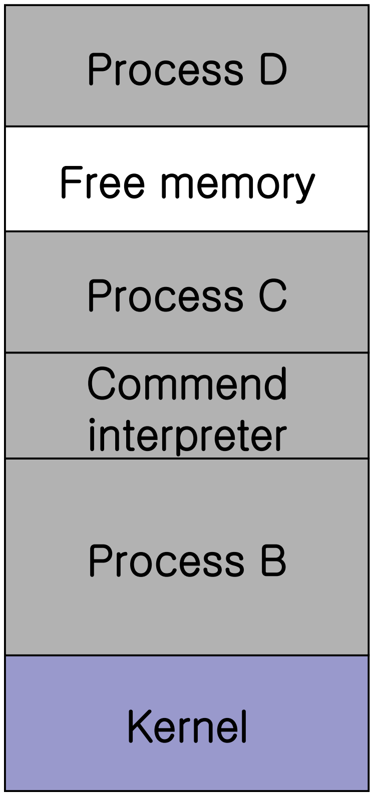
- Single-tasking system
  1. Command interpreter is invoked at system start
  2. Load a program to memory
     - Write over itself to provide as much memory as possible
  3. Run the program
  4. Terminates
     - When error occurs, error code is saved in memory
  5. Overwritten part of command interpreter is reloaded and resume execution
  6. Report error code and continues



Example: FreeBSD UNIX

- Multitasking system





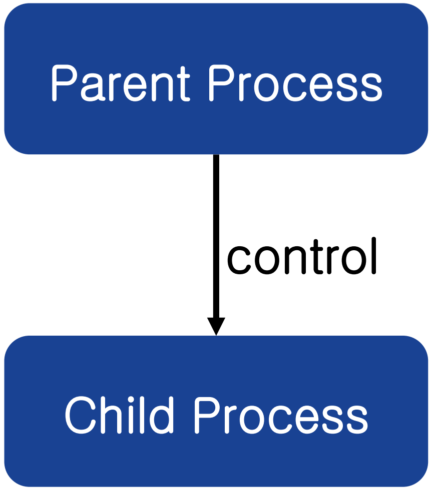
< FreeBSD running multiple program >

Example: FreeBSD UNIX

- Command interpreter may continue to execute
- Two cases of execution
  - Case 1, shell continues to execution
    - New program is executed in background
      - Console input is impossible for new program.
    - User is free to ask the shell to run other programs.
  - Case 2, shell waits new program
    - New program takes I/O access
    - When the program terminates (exit()), the control is returned to shell with a status code (0 or error code)

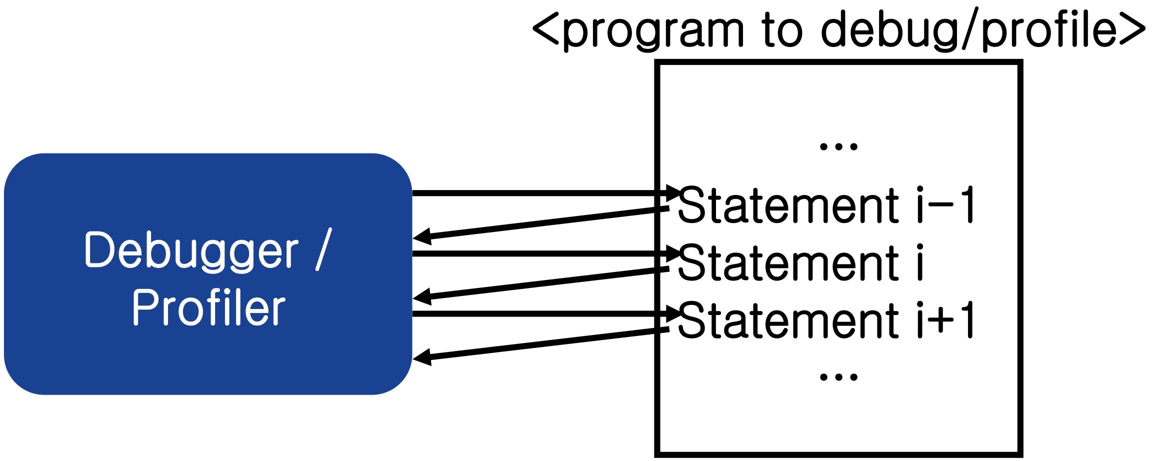
Process Control: Load/Execution

- Controlling new process
  - Get/set process attributes
    - Priority, maximum execution time, …
  - Terminate process
- Waiting for new job/process
  - Wait for a fixed period of time
  - Wait for event / signal event

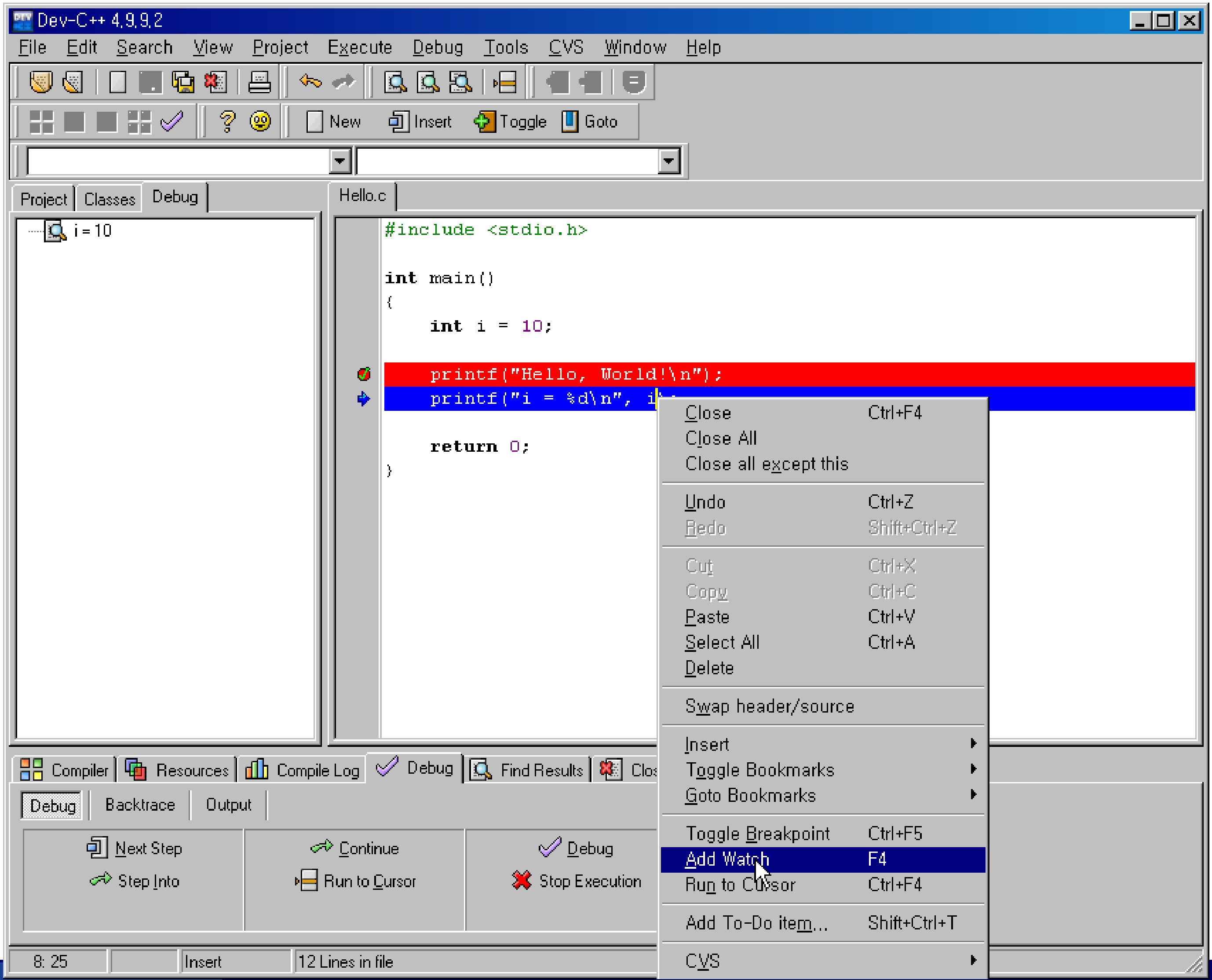


Process Control: Load/Execution

- Debugging
  - Dump
  - Trace: trap after every instruction



Debugger


Dev-C++ 4.9.9.2
File Edit Search View Project Execute Debug Tools CVS Window Help
New Insert Toggle Goto
Project Classes Debug
i = 10
Hello.c
#include <stdio.h>
int main()
{
int i = 10;
printf("Hello, World!\n");
printf("i = %d\n", i);
return 0;
}
Close Ctrl+F4
Close All
Close all except this
Undo Ctrl+Z
Redo Shift+Ctrl+Z
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Select All Ctrl+A
Delete
Swap header/source
Insert
Toggle Bookmarks
Goto Bookmarks
Toggle Breakpoint Ctrl+F5
Add Watch F4
Run to Cursor Ctrl+F4
Add To-Do item... Shift+Ctrl+T
CVS
Compiler Resources Compile Log Debug Find Results
Debug Backtrace Output
Next Step Continue Debug
Step Into Run to Cursor Stop Execution
8: 25 Insert 12 Lines in file

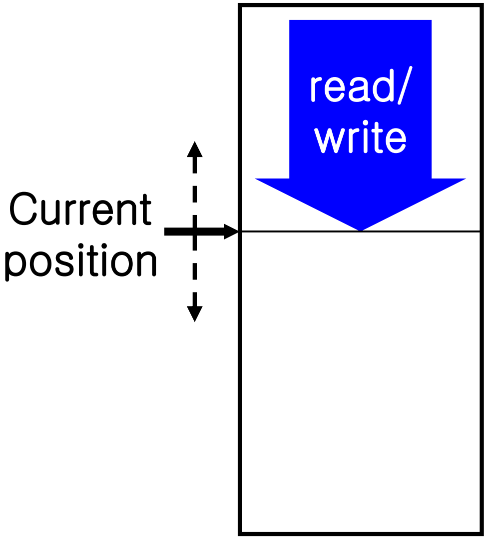
Process Control: Termination

- Normal termination (end)
  - Deallocate resources, information about current process

- Abnormal termination (abort)
  - Dump memory into a file for debugging and analysis
  - Ask user how to handle
    - Interactive system: command interpreter
    - GUI system: pop-up window
    - Batch system: terminates entire job and continue with next job
      - Control card: command to manage execution of process

File Management

- Create/delete files
- Read/write/reposition
- Get/set file attribute
- Directory operation
- More service
  - move, copy, …



➔ Functions can be provided by either system calls, APIs, or system programs

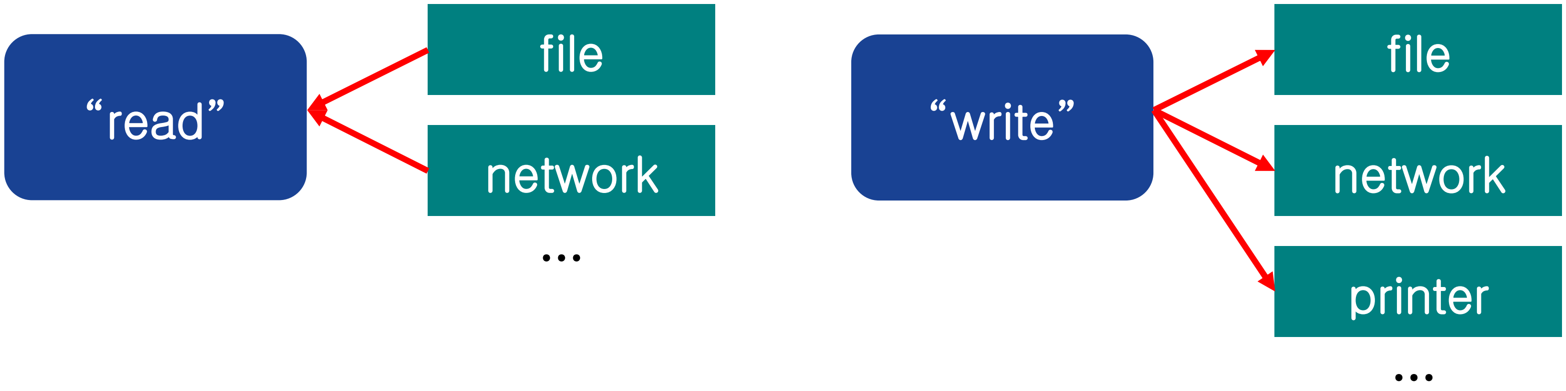
Device Management

- Resources
  - Physical device (disk, tape, …)
  - Abstract/virtual device (file, …)

- Operations
  - Request for exclusive use ≈ open()
  - Read, write, reposition ≈ read(), write(), ⋯
  - Release ≈ close()

Device Management

- Combined file-device structure
  - Mapping I/O into a special file
  - The same set of system calls on both files and devices



Information Maintenance

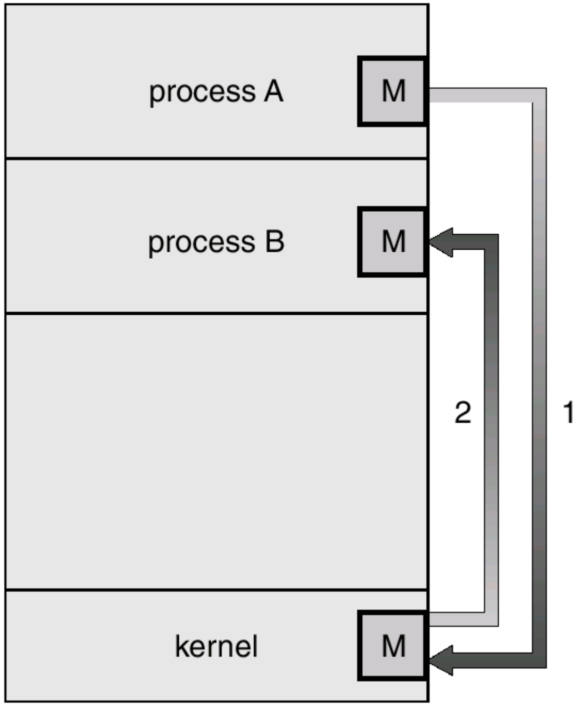
- Transfer information between OS and user program
  - Current time, date
  - Information about system
    - # of current user, OS version, amount of free memory/disk space

- OS keeps information about all it processes

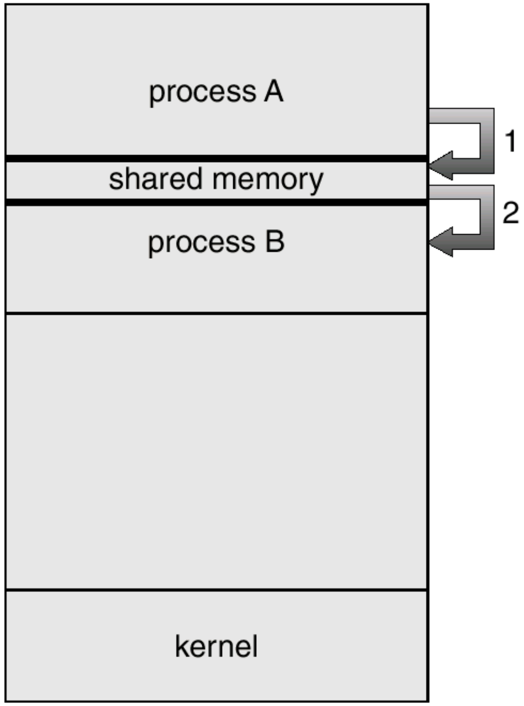
  Ex) /proc of Linux

Communication

- Inter-process communication
  - Message passing model
  - Shared-memory model



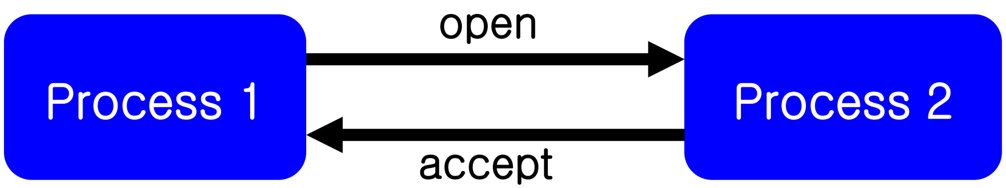
Message passing



Shared memory

Message-Passing Model

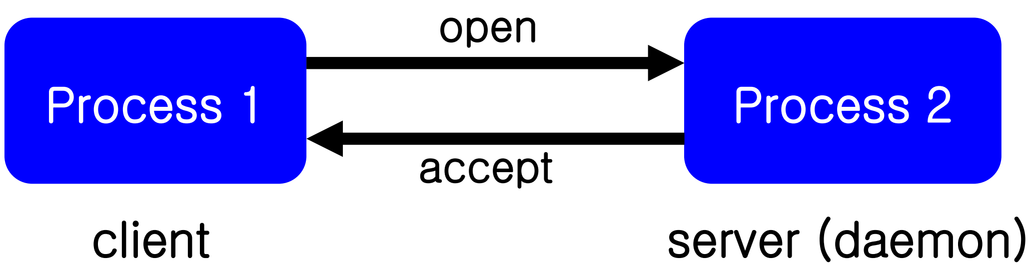
- Identifying communicator (counter part)
  - Host name/network identifier and process name
- Opening connection between processes
  - open / close (file system calls), or
  - open connection / close connection (connection system calls)
- Permission from recipient
  - Accept connection



Message-Passing Model

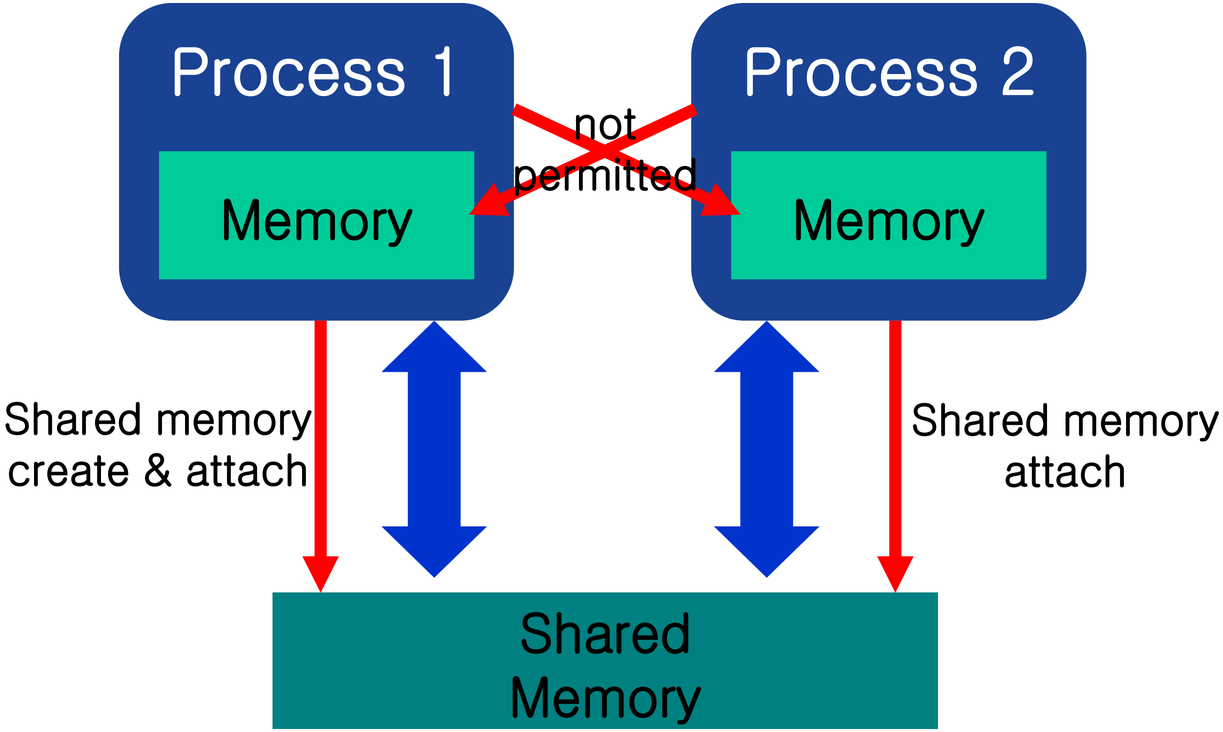
- Usually, *client* requests connection and *daemon* receives it.

- Daemon: a computer program that runs in the background to provide a service.
  - Ex) ftpd(ftp server), httpd(web server), syslogd, sshd (secure shell daemon), telnetd, …



Shared-Memory Model

- Process communication using shared memory
  - Shared memory create
  - Shared memory attach
    - Access to shared memory owned by other process



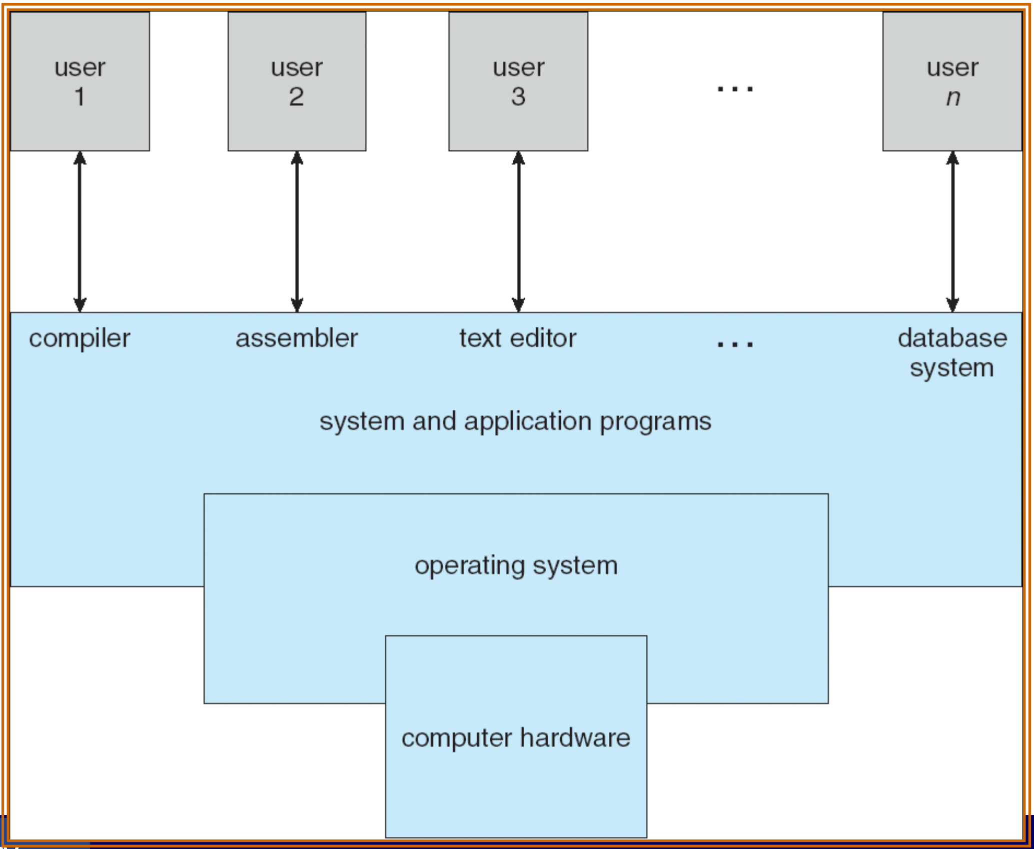
Comparison

- Message passing model
  - No conflicting on resource access
  - Size of the message is usually limited.
    - ➔ Suitable for a small amount of data
  - Communication with processes on remote machines.

- Shared-memory model:
  - Fast
    - ➔ Suitable for a large amount of data

System Programs

- System program: a program to provide a convenient environment for program development and execution.
- Some of them are simply user interfaces to system calls; others are considerably more complex



System Programs

- System programs can be divided into:
  - File manipulation
  - Status information sometimes stored in a File modification
  - Programming language support
  - Program loading and execution
  - Communications
  - Background services

System Programs

- **File management**
  - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information**
  - Some ask the system for info - date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information
  - Typically, these programs format and print the output to the terminal or other output devices
  - Some systems implement a **registry** - used to store and retrieve configuration information

System Programs (Cont.)

- **File modification**
  - Text editors to create and modify files
  - Special commands to search contents of files or perform transformations of the text
- **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes provided
- **Program loading and execution**- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems
  - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another

System Programs (Cont.)

- **Background Services**
  - Launch at boot time
    - Some for system startup, then terminate
    - Some from system boot to shutdown
  - Provide facilities like disk checking, process scheduling, error logging, printing
  - Run in user context not kernel context
  - Known as **services**, **subsystems**, **daemons**

Agenda

- Operating-system services
- Interfaces for users and programmers
- **Components and their interconnections**
- Virtual Machines
- Design, implementation, generation
- System boot

Operating System Design

- Design and Implementation of OS not "solvable", but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- Start by defining goals and specifications
- Affected by choice of hardware, type of system
- **User** goals and **System** goals
  - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

Operating System Design

- Important principle to separate

  Policy: ***What*** will be done?

  Mechanism: ***How*** to do it?

- Mechanisms determine how to do something, policies decide what will be done
  - The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later

- Specifying and designing OS is highly creative task of **software engineering**

Implementation

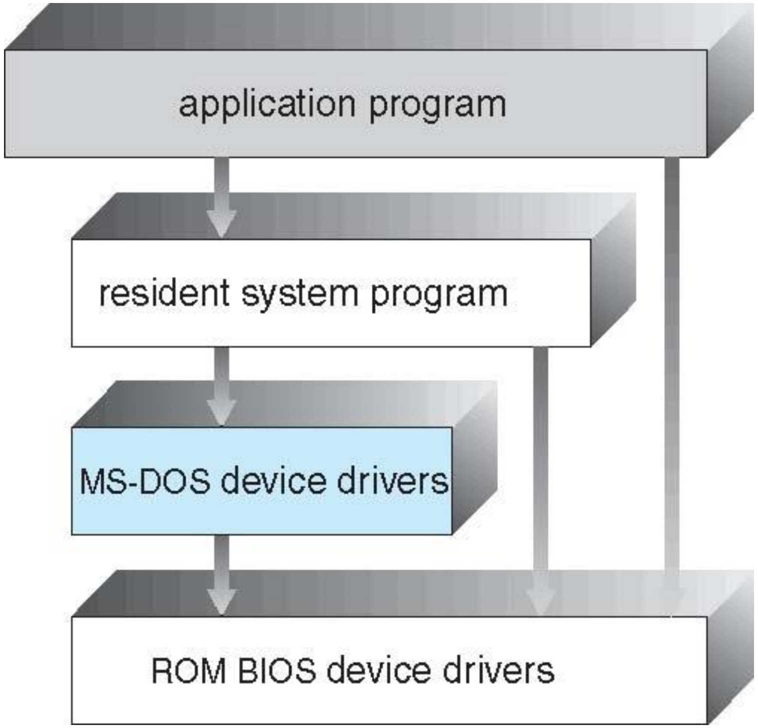
- Much variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- Actually usually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- More high-level language easier to **port** to other hardware
  - But slower
- **Emulation** can allow an OS to run on non-native hardware

Operating-System Structure

- General-purpose OS is very large program
- Various ways to structure ones
  - Simple structure – MS-DOS
  - More complex – UNIX
  - Layered – an abstrcation
  - Microkernel – Mach

Simple Structure

- MS-DOS (1981)
  - Started as small, simple limited system
    - Provide most functionality in least space
  - Interface / level of functionality are not well separated
    - No dual mode or H/W protection
    - Application program can access I/O directly
    - Vulnerable to errant program
      - An error in a program can crash all system
    - Limited on specific H/W



< Structure of MS−DOS >

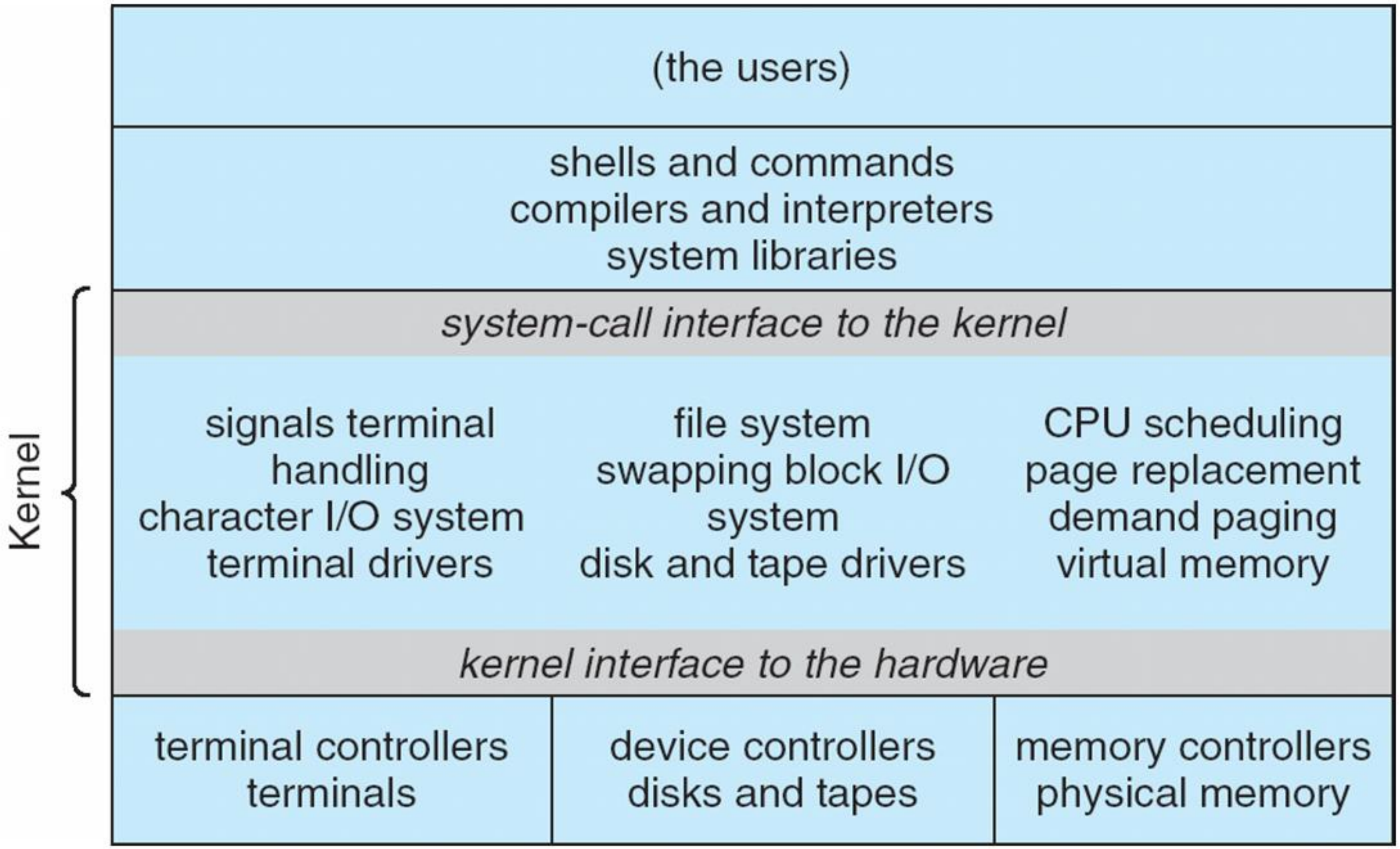
DOS: disk operating system

Non-Simple Structure

- Original UNIX(1973)
  - Also limited by H/W functionality
  - Systems programs
    - Shell, commands compiler, interpreter, system library, …
  - Monolithic kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides File system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level.

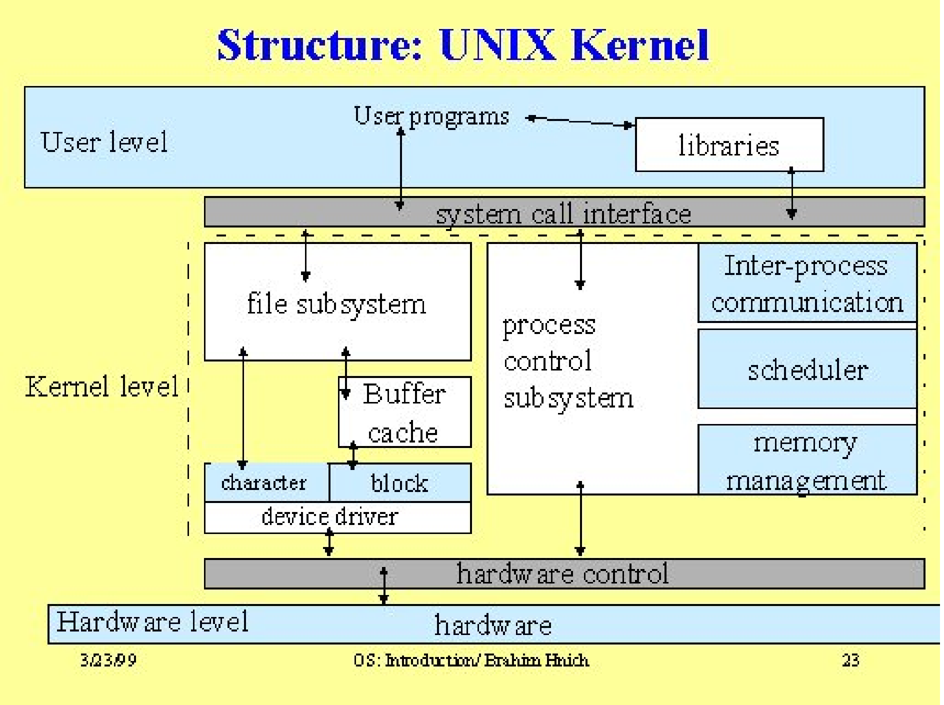
Simple Structure

- Original UNIX *(beyond simple but not fully layered)*



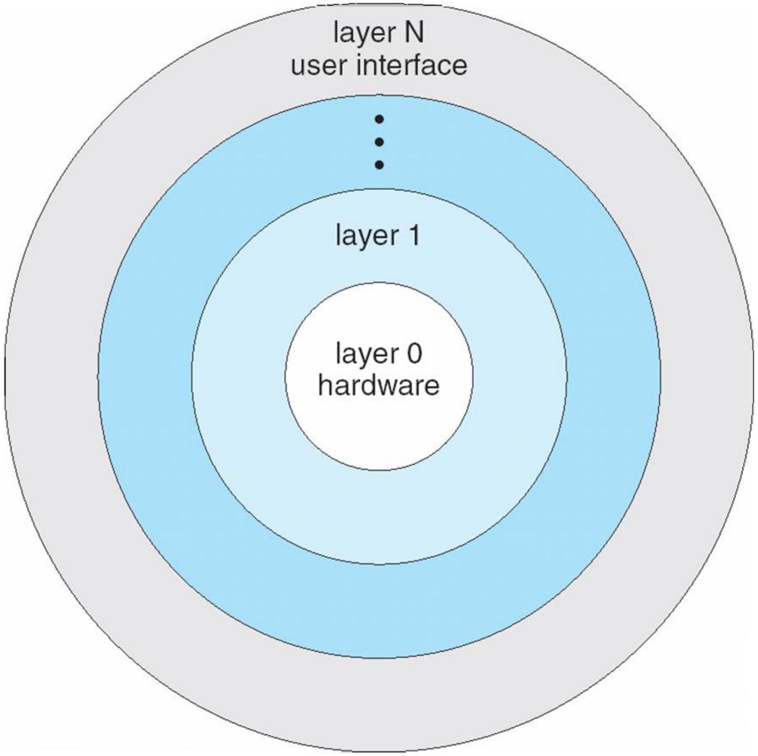
Modern Operating Systems

- Modern OS's can be broken into pieces appropriately
  - Easy to implement
  - Flexible
  - Information hiding



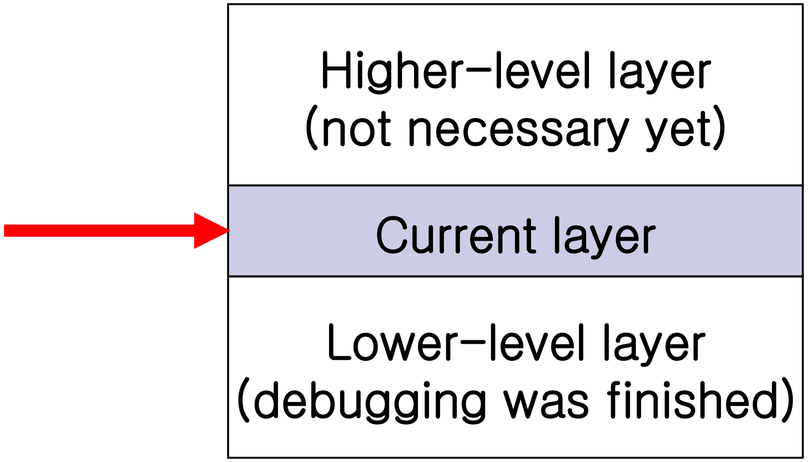
Layered Approach

- OS is composed of layers
- Layer
  - Implementation of abstract objects and operation
  - Each layer M can invoke lower-level layers
  - Each layer M can be invoked by higher-level layers
- Each layer uses functions/services of only lower-level layers



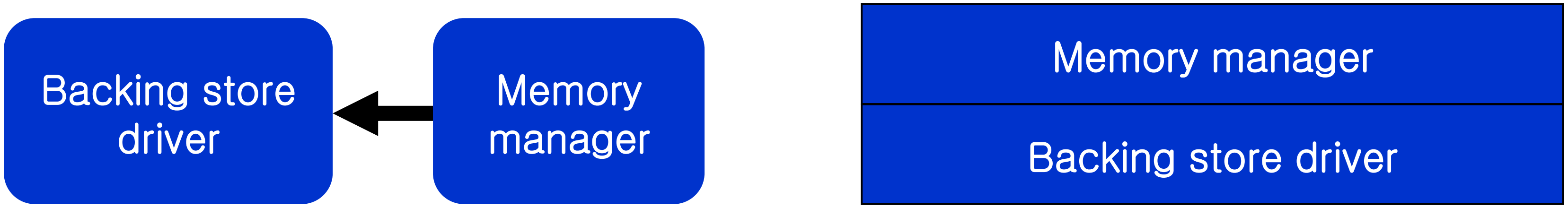
Layered Approach

- Advantages of layered approach: simple to construct and debug
  - If we develop from lower-level layer to higher-level layer, we can concentrate on current layer at each stage
  - A layer doesn't need to know detail of lower-level layer

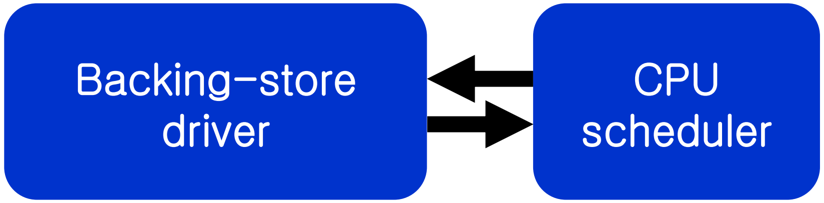


Layered Approach

- Difficulties of layered approach
  - Defining various layers needs careful planning

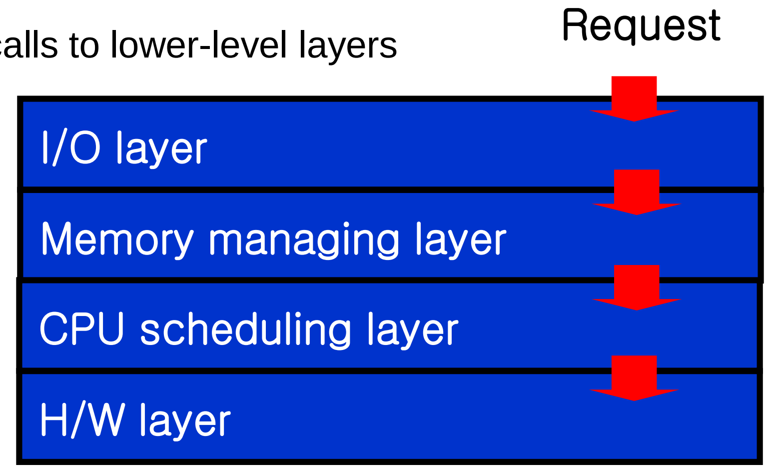


  - How to define hierarchy between the modules requires each other



Layered Approach

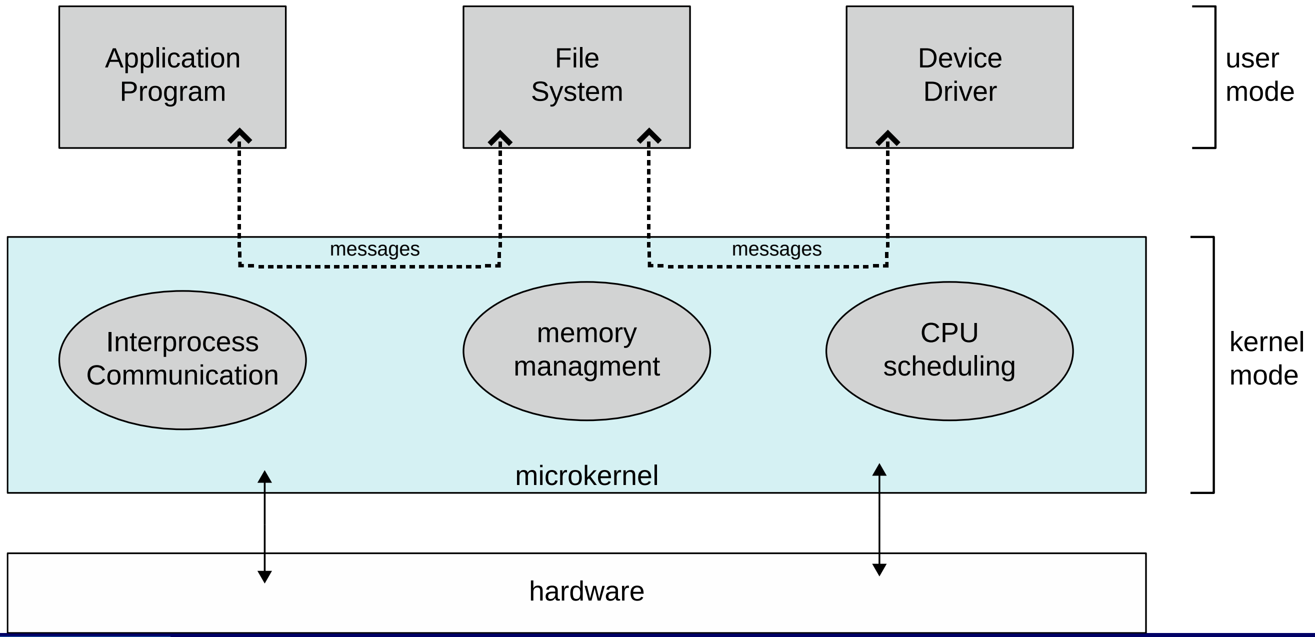
- Difficulties of layered approach
  - Inefficiency
    - Repeating calls to lower-level layers



- Remedy
  - Apply fewer layers - Take advantage and avoid difficulties

Microkernels

- Smaller kernel
  - All unessential components are not implemented in kernel but as system/user-level programs.
    - Only essential components are included in kernel
    - Other components are provided by system/user programs



Microkernels

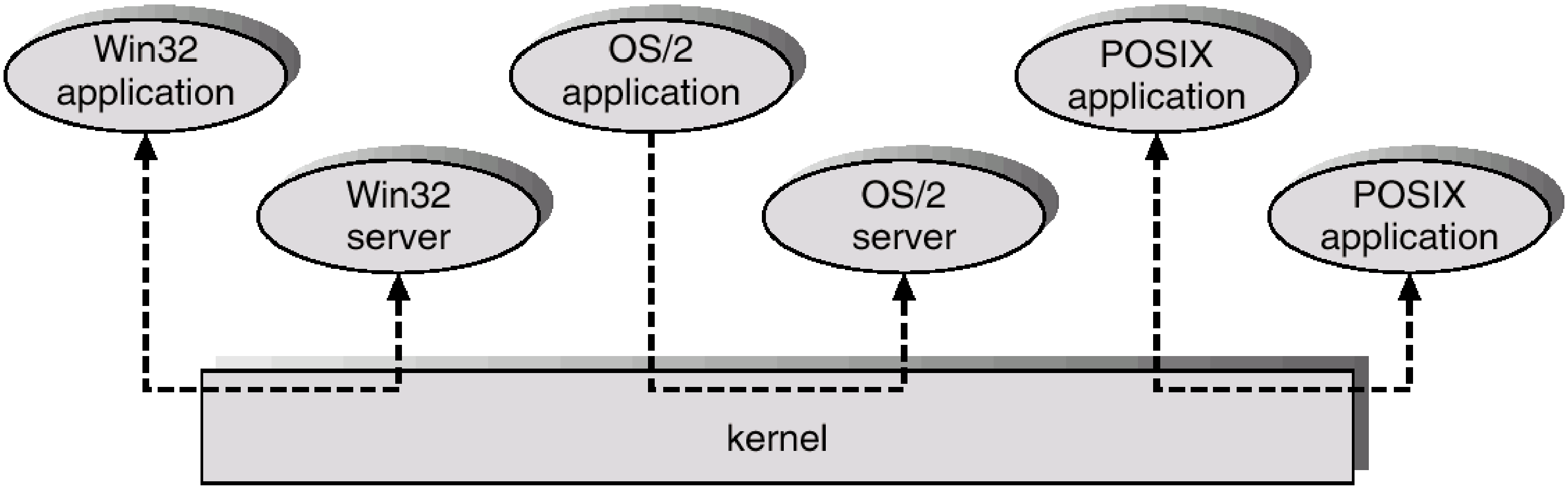
- Generally, process/memory management, communication facility are in the kernel.

- System calls are provided through message passing.
  - Clients and services are running in user space
  - Kernel provides only a message passing facility between client and server

Microkernels

- Examples
  - Tru64 UNIX, QNX (real-time OS)
  - Windows NT: hybrid structure (layered microkernel)

    cf. Windows XP: more monolithic than Windows NT



< Windows NT client-server structure >

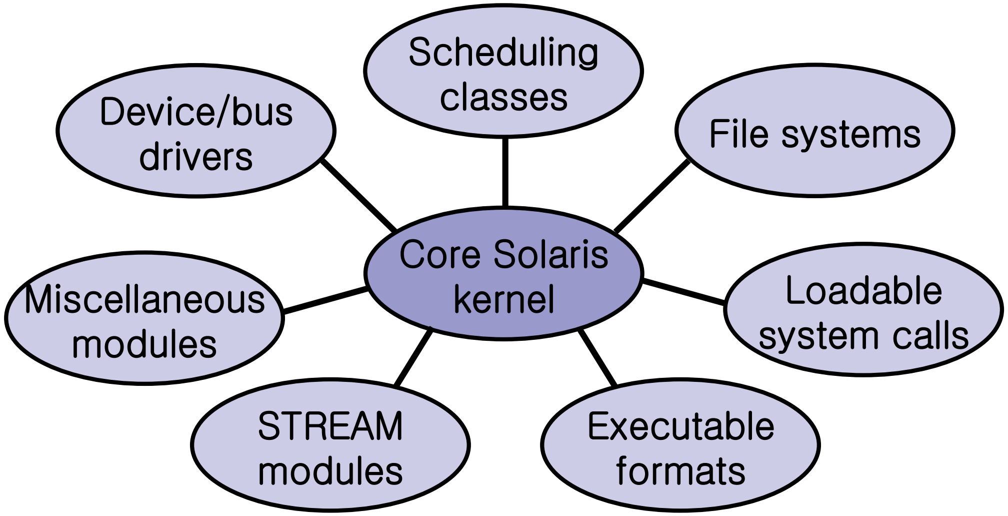
Microkernels

- Advantages of microkernel
  - Ease of extending
  - Ease to port
  - Security and reliability
    - Most services are on user space

- Disadvantages
  - Performance decrease due to increased system function overhead.
  - Performance overhead of user space to kernel space communication

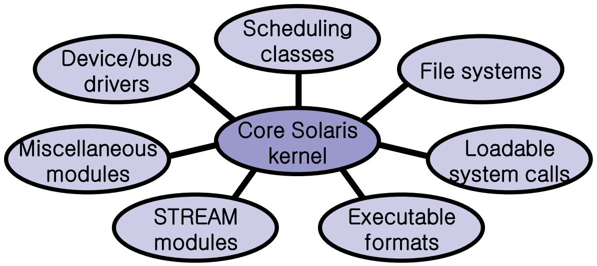
Modules

- Kernels with loadable modules (Linux, Solaris, etc)
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel

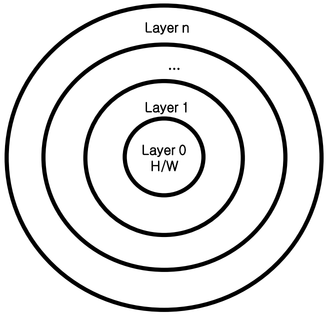


Modules

- Advantage
  - Provides core services
  - Allows certain features to be implemented dynamically



- Comparison with layered structure
  - More flexible (any module can call any other modules)



- Comparison with microkernel
  - Each module can run in kernel mode
  - Modules don't need to invoke message passing