Operating Systems

(*Principles of Operating Systems*)

Đỗ Quốc Huy
huydq@soict.hust.edu.vn
Department of Computer Science
School of Information and Communication Technology

Course Info

Text book:

- Operating System Concepts – Abraham Siblerschatz
- Modern Operating System – Andrew Tanenbaum

Group Projects

Chapter 1 Operating System Overview

① Notion of operating system
② History of operating Systems
③ Definition and Classifications
④ Basic Properties of Operating Systems
⑤ Notions in operating systems
⑥ Operating System structures
⑦ Principles of Operating Systems

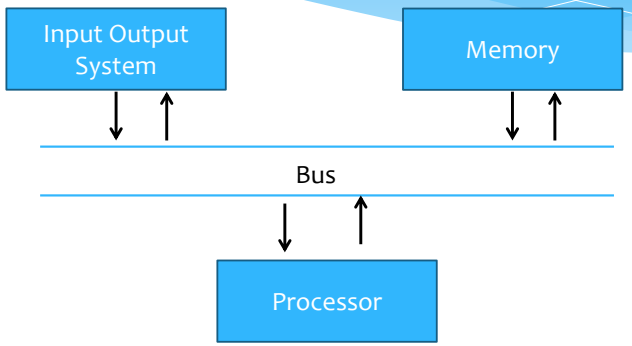
Chapter 1 Operating System Overview

① Operating system notion
② History of Operating Systems
③ Definition and Classifications
④ Basic Properties of Operating Systems
⑤ Notions in operating systems
⑥ Operating System structures
⑦ Principles of Operating Systems

① Notion of operating system

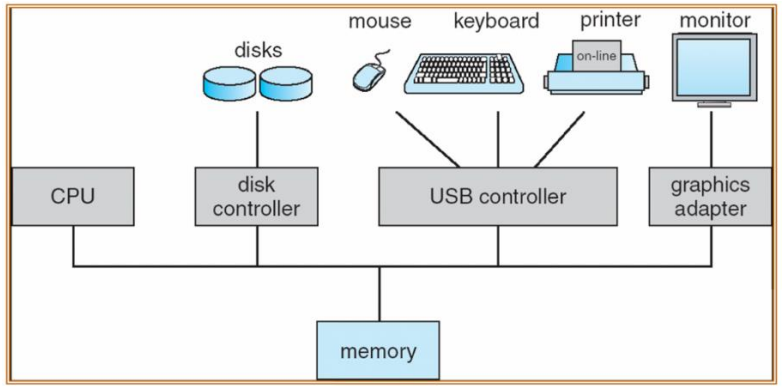
- Layered structure of a computing system
- Operating system's functions

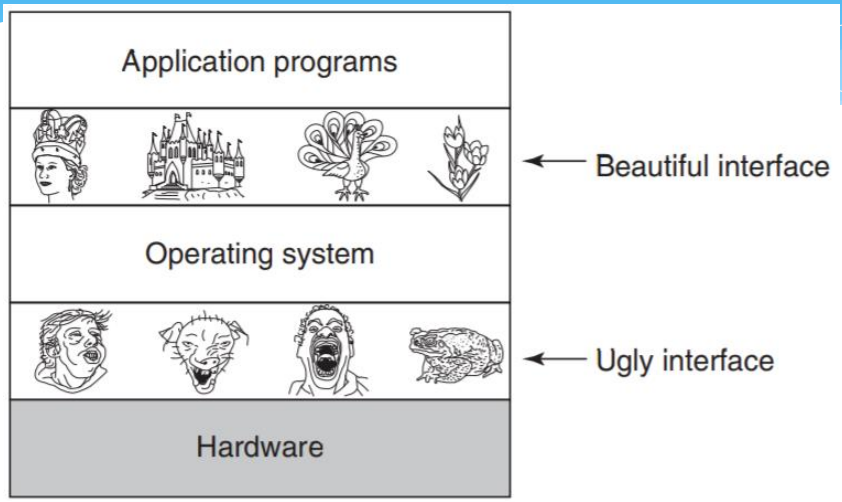
- A computer system's structure



- One/ many CPUs, controlling devices are linked by a common bus system to access a shared memory.
- Controlling devices and CPU operate simultaneously and compete with each other.

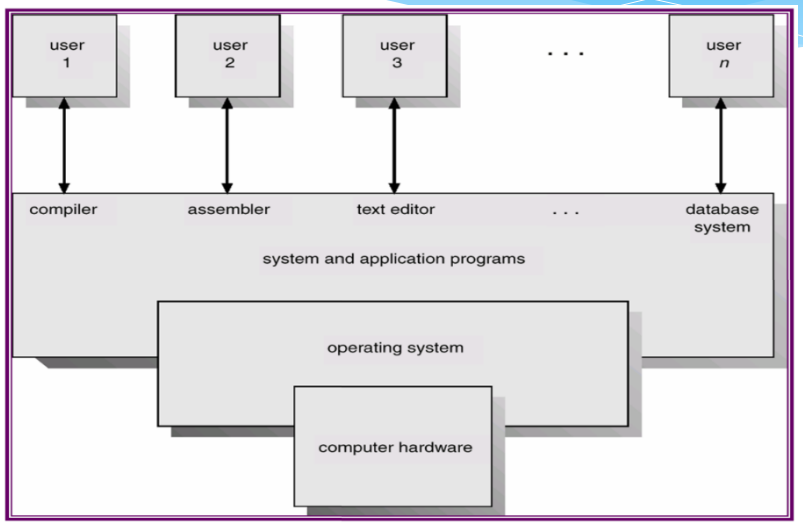
- A computer system's structure



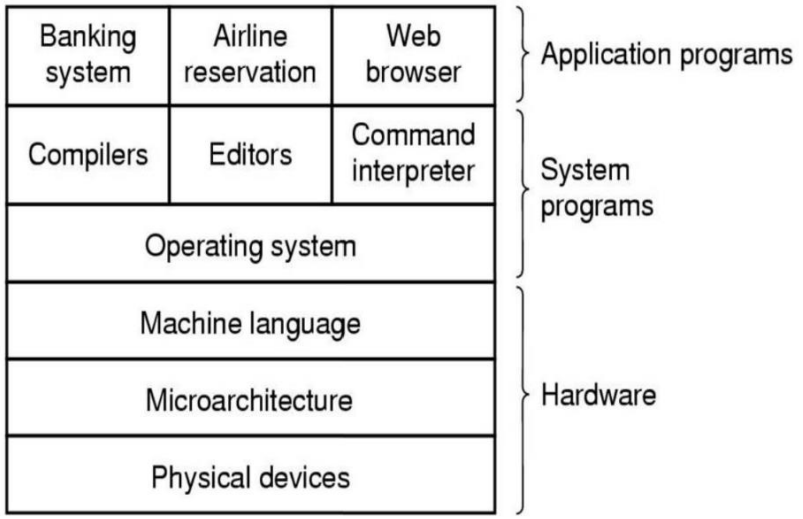


Operating systems turn ugly hardware into beautiful abstractions

- A computer system's components (Silberschatz 2002)

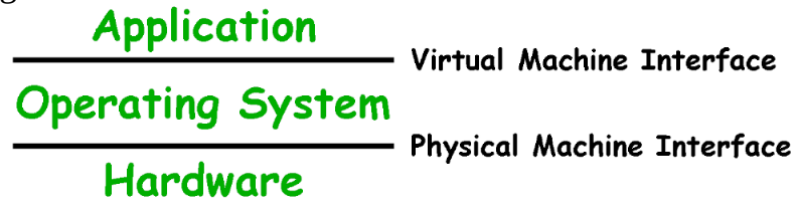


- A computer system's components (Tanenbaum 2001)



- A computing system's components
- Hardware: Provides basic computing resources (CPU, memory, input-output devices)
- Operating system: controls and cooperates hardware using works for application programs of different users.
- Application programs: (compiler, database system, game...) utilizes computer's resource to handle users' requests.
- Users: People who work/operate the machines or computers

- **Objectives**
- Operating system lies between the system's hardware and application program



- **Objectives**: To provide an environment which helps user run application program and use the computer system easier, more conveniently and effectively.
  - Standardize the user interface for different hardware systems
  - Utilize hardware resource effectively and exploit the hardware performance optimally.

① Notion of operating system

- Layered structure of a computing system
- Operating system's functions

① Simulate a virtual computer machine

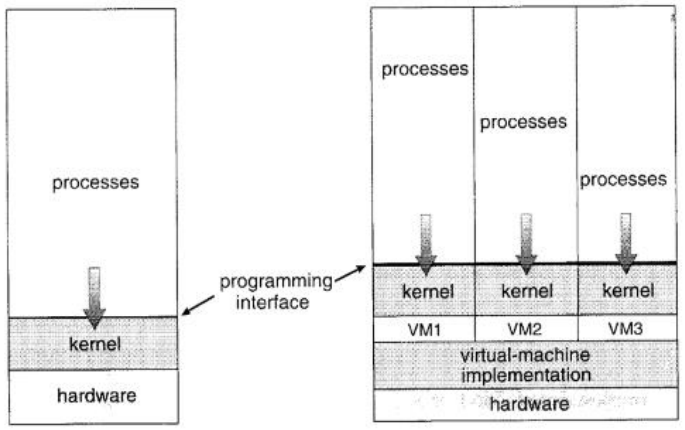
② Manage system resources

Simulate a virtual computer machine

Help hide detailed works and exploit computer hardware's functions easier and more effectively.

- Simplify programming problem
  - No need to work with binary sequences
  - Each program thinks that it own the whole computer's memory, CPU time, devices...
  - Help communicating with devices easier than with original device. Ex: Ethernet card: Reliable communication, ordered (TCP/IP)
- Extend the system's abilities: The system seem to have desired resources (virtual memory, virtual printer...)
- Help programs not violating each other → a program which does not work would not damage the whole system
- Useful for operating system's development
  - If the experimental operating system get errors, only limited in the virtual machine
  - Help verifying other programs in the operating system

Simulate a virtual computer machine



Non virtual machine

With virtual machine

System's resources management

- System's resources (CPU, memory, IO devices, files...) are utilized by program to perform a determined task.
- Programs require resources: time (CPU-usage) and space (memory)
- The operating system has to manage the resource so that the computer can work in the most effective way.
  - Provide resource for program when it's necessary.
  - Competition handling
  - Decide the order of resource providing for the programs' requirements
  - Example: memory resource management (limited)
    - Many program can operate at the same time.
    - Avoid illegal access
    - Data protection (memory sharing: file)

Chapter 1 Operating System Overview

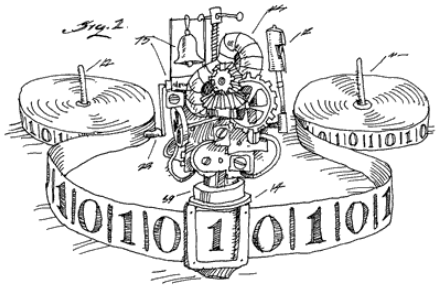
Operating systems development's history

- History of electronic computer
- Operating systems development's history

Development history of electronic computers

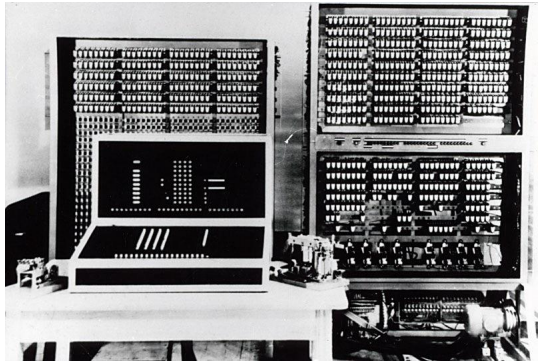
- 1936 - A. Turing & Church present logic computing model and prove the existence of a computer: Turing machine

●Model of a character processing device; Simple but able to perform all the computer's algorithm
●A Turing machine that is able to simulate any other Turing machine -> a universal Turing machine
● *Turing is considered as the father of computer science and artificial intelligence*



Development history of electronic computers

- 1941- Konrad Zuse (German) Constructed world's first programmable computer; the functional program-controlled Turing-complete Z3

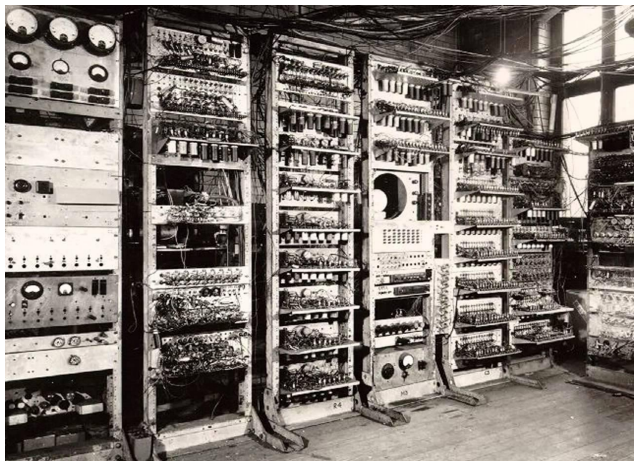


- Z3: use binary system
- *Has separated memory and controller*
- *Mechanical technique*

Chapter 1. Operating System Overview
2.History of Operating Systems
2.1. History of electronic computer

Development history of electronic computers

- 1946 ENIAC based on electric bulbs

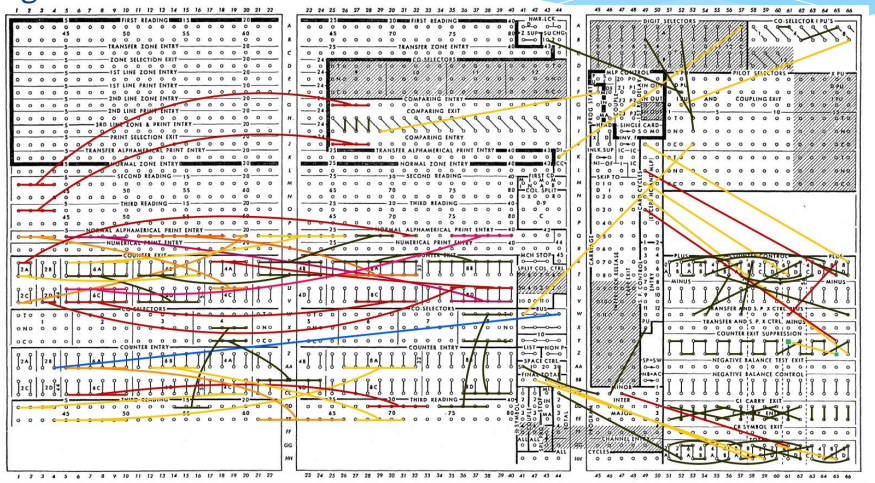


- 18000 vacuum tube
- 70000 resistance
- 5 million metal connector
- *Faster but least reliable*

Chapter 1. Operating System Overview
2.History of Operating Systems
2.1. History of electronic computer

Development history of electronic computers

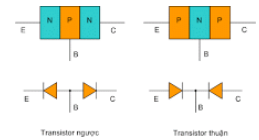
- plug board



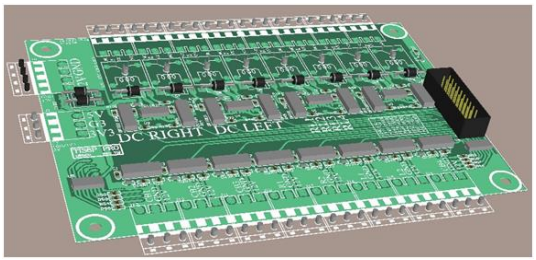
Chapter 1. Operating System Overview
2.History of Operating Systems
2.1. History of electronic computer

Development history of electronic computers

- 1950-1958 Transistor
- 1959-1963 Semiconductor



- 1964-1974 Integrated Circuit (IC)



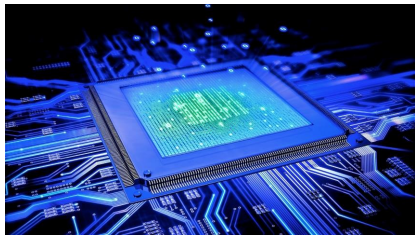
Development history of electronic computers

- 1974-1990 Large scale IC:

Allow CPU, main memory or similar device to be produced in a single integrated circuit

-> new class of smaller, cheaper computer and parallel processor with multi CPUs

- 1990-now Very large scale IC, smart IC



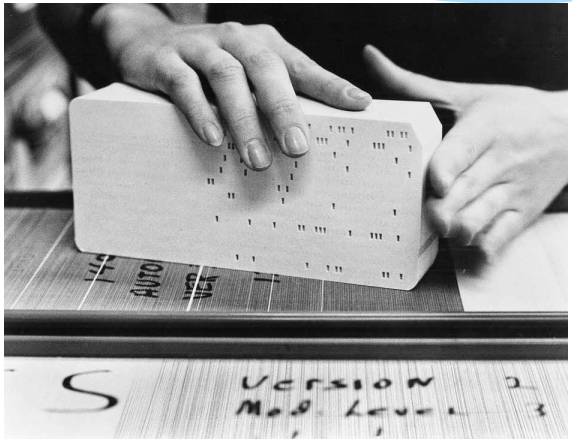
Operating systems development's history

- History of electronic computer
- Operating systems development's history

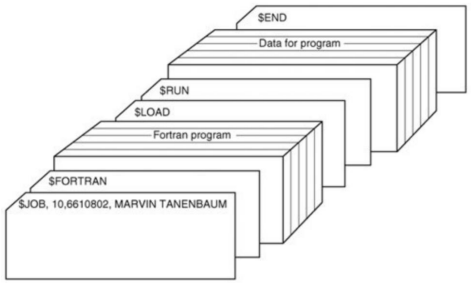
- 1948-1970 : Hardware expensive; human labor cheap
- 1970-1981 : Hardware cheap; human labor expensive
- 1981- : Hardware very cheap; human labor very expensive
- 1981- : Distributed system
- 1995- : Mobile devices

- 1948-1970 :
- Computer 1-5 M$: Nation 's property, mainly used for military's purposes ⇒ Require optimization for using hardware effectively
- Lack of human-machine interact
- User, programmer; operator are same group of people
- One user at a single time
  - User wrote program on punched cards
  - First card is bootstraps loader is loaded into memory and executed
  - Instructions in bootstraps loader fetch into memory and execute instructions on other later cards (application program)
  - Check light bulb for results, debug
- Debugging is difficult
- Waste processor time
- Solution: batch processing

* 1948-1970 :



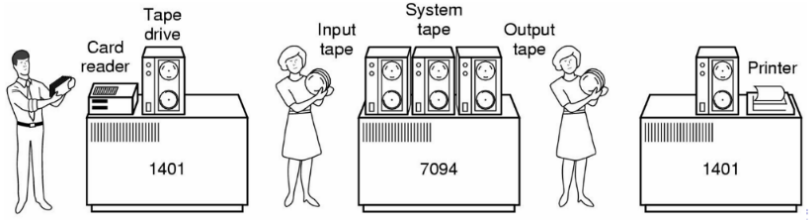
* Batch processing and professional operator
* Programmer give program to operator
* Operator group program into a single pack (batch)
* Computer read and run each program consequently
* Operator take the result, print out and give to programmer



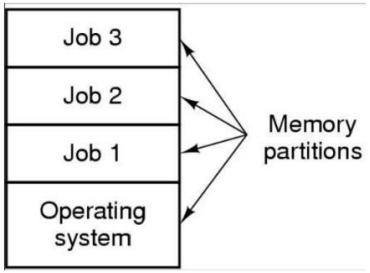
- Reduce waiting time between jobs
- Input/output problem
  - Computer getting faster
  - Card reader still slow

⇒CPU has to wait for card reading/writing

* Replace card reader by tape ⇒ Independent external computers for reading/writing data
* External devices are designed to be able to Direct Memory Access, using interrupt and i/o channel
  * OS request I/O device then continue its work
  * OS receive interrupt signal when I/O devices finishes
* ⇒Allow overlap between computing and I/O



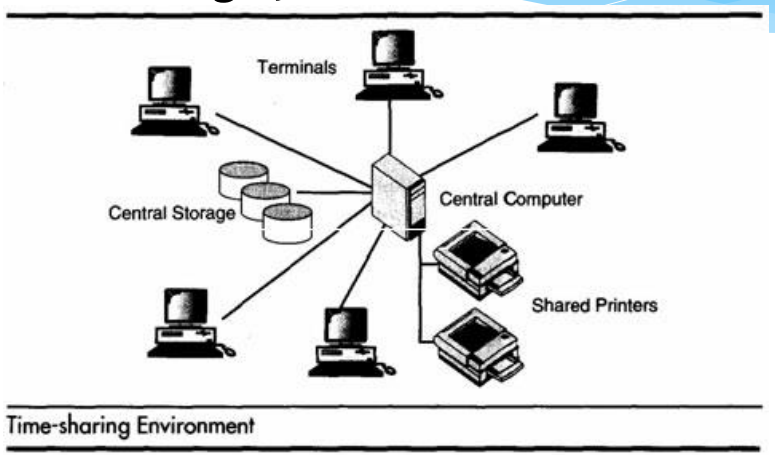
- CPU is reprogrammed to be switch easily between programs
- Hard ware: memory space larger and cheaper Some program can run simultaneously -> multi programming



  - More overlap between computing and I/O
  - Require memory protection between programs and keep one crashed program from damaging the system
- Problem: OS has to manage all interaction ⇒ out of control (OS360: 1000 error)

* 1970-1981 :
- Computers prices about 10.000$ ⇒ used widely for different jobs
- OS technology became stable.
- Using cheap terminal device (1000$) allow many user to interact with the system at the same time
  - User perform different works (text editor, chat, program debugging,..) ⇒ require system to be exploited effectively
    Example: a PC: 10M calculation/s; typing speed 0.2s/1 character => lost 2M calculation per one typing
  ⇒ Time sharing operating system
    - Problem: system's response time
- Computer network was born (ARPANet : 1968) Communication between computer; Protection against network attack

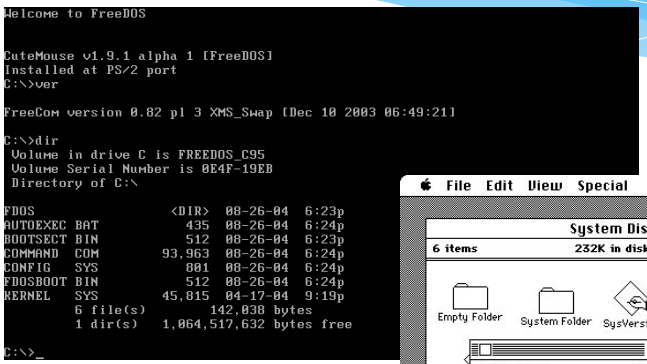
Time sharing system



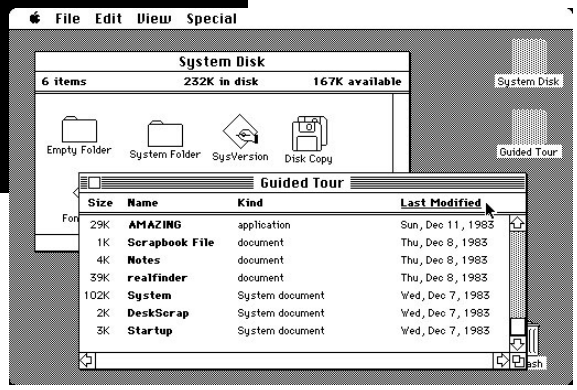
[FAIZAL MOHAMAD – Introduction to operating system]

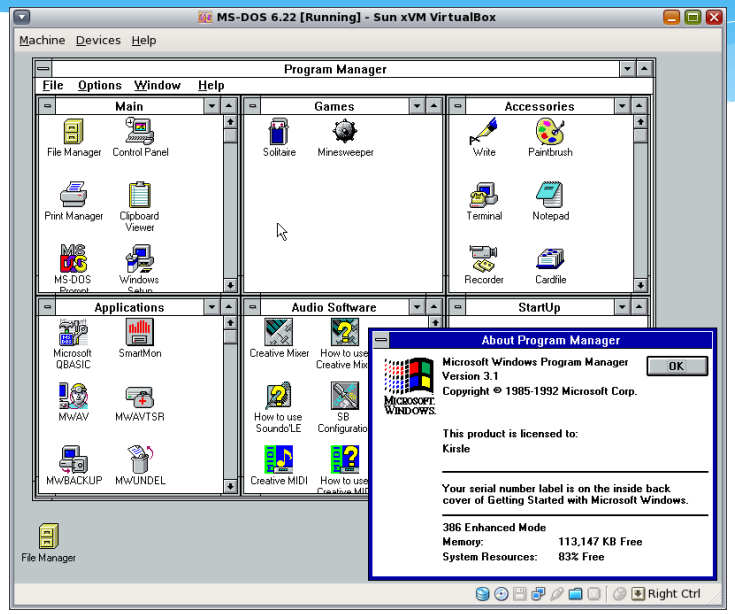
* 1981-1995 :
- Computers prices about 1000$; human labor 100K $/year ⇒ Computer are used more widely for working more effectively
- Personal computing
  - Cheap computer, single person can afford (PC).
  - OS on PC
- Hardware resources are limited (Early : 1980s)
  - OS become library of available procedures
  - Run One program at a time (DOS)
- PC become more powerful
  - OS meet complex problems: multi tasking, memory protection… (WINXP)
- Graphical user interface (MAC, WIN,..)

* DOS User interface

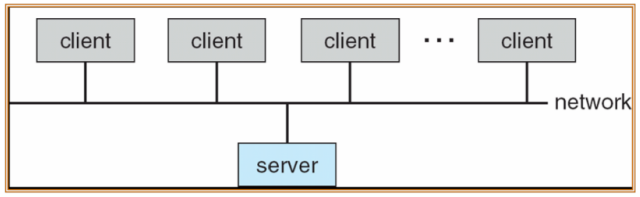


* Macintosh User interface





- Distributed systems
- Development time of networking and distributed operating systems



  - Local area network
    - Computers share resources: printer, File servers,..
    - Client / Server model
  - Services
    - Computing, storage
    - Services provided through Internet.
  - Problems
    - Transmission delay; bandwidth, reliable...
    - Virus (love letter virus 05/2000),..

- >45 millions computers were infected
- Stole information
- Auto send emails from contact list
- Download Trojan

- Mobile devices
- Mobile devices become more popular
  - Phone, Laptop, PDA . . .
  - Small, changeable and cheap → More computers/human
  - Limited ability: speed, memory,...
- Wide area network, wireless network
  - Traditional computer divided into many components (wireless keyboards, mouse, remoting storage)
- peer-to-peer system
  - Devices with the same role working together
  - “Operating system’s” components are spread globally
- Cloud computing
  - Cloud operating system

* Conclusion
- The development of the operating systems are strongly connected with the computers' development
- Operating system development pulled the development of computers

Chapter 1 Operating System Overview

① Operating system notion
② History of Operating Systems
③ Definition and Classifications
④ Basic Properties of Operating Systems
⑤ Notions in operating systems
⑥ Operating System structures
⑦ Principles of Operating Systems

Operating System definition and classification

- Definitions
- Classifications

Observer's perspective

- Different objects have different requirements for operating system
- Different observing perspectives ⇒ different definitions

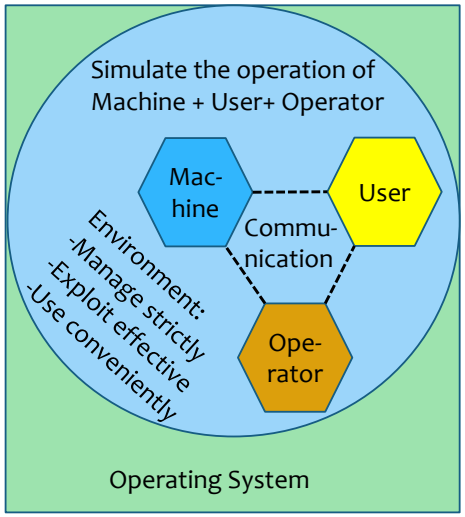
User: A system of programs that help exploiting the computing system conveniently

Manager: A system of programs that help managing computing system's resource effectively

Technical perspective: A system of programs equipped for a specific computer to make a new logic computer with new resource and new ability

System engineer perspective: A system of programs that modelize, simulate the operation of computer, user and operators. It work in a communicating mode in order to make a convenient environment for exploiting the computer system and maximum resource management.

System engineer's perspective



Simulate 3 roles ⇒ require 3 types of languages

- Machine language
  - The only working language of the system
  - All other languages have to be translated into machine language
- System operation's language
  - OS commands (DOS: Dir, Del..; Unix: ls, rm,..)
  - Translated by the Shell
- Algorithm language
  - Programming language
  - Compiler

Operating System definition and classification

- Definitions
- Classifications

- Batch processing single program system
- Batch processing multi program system
- Time sharing system
- Parallel system
- Distributed system
- Real-time processing system

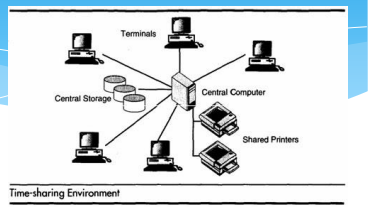
Batch processing single program system

- Programs are performed consequently follow predetermined instructions
- When a program finished, the system auto run the next program without any external intervention
- Require a supervisor process the sequence of jobs and the supervisor has to stay permanent in the memory
- Need to organize a job queue
- Problem: when a program access an I/O device, processor has to wait

Batch processing multi program system

- Allow many programs to run at the same time
  - Load one part of code and data of the programs into the memory (the remaining parts will be loaded at proper moments). The program are ready to run
  - Run the program similar to single program system
  - If the current program perform an IO, processor will be change to another program
- Save memory (no need to load all the program into the memory)
- Reduce processor spare-time
- High cost for processor scheduling. Which program can use processor next?
- Solve the memory sharing problem between programs

Time sharing system



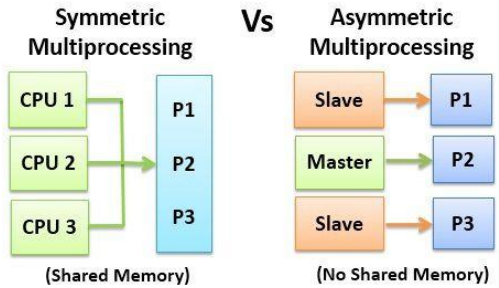
- Processor's usage allowance time is shared among ready-to-run programs
- Similar to batch processing multi program system (only load part of the programs)
- Processor is issued mainly by the operating system ⇒ how ? ⇒ Chapter 2
- Swapping times between program are small -> programs seem to run parallel
- Usually called: Multi tasking operating system (Windows)

Parallel system

- Constructed for system that has many processors
  - Many processors, works are done faster
  - More reliable: one processor breaks down will not affect the system
  - Advantage over single processor computer due to memory, peripheral devices sharing...
- Symmetric multi processing (SMP: symmetric)
  - Each processor run a single program
  - Processors communicate via a shared memory
  - Fault tolerance mechanism and optimal load balance
  - Problem: processor synchronization
  - Example: WinNT operating system

Parallel system

- Asymmetric multi processing (ASMP: asymmetric)
  - One processor controls the whole system
  - Other processors follow the main processor's commands or predetermined instructions
  - This model has the master-slave relation form: The main process will make schedule for other processors
  - Example: IBM System/360



Distributed system

- Each processor has a local memory and communicate via transmission lines
- Processors are different from sizes to functions (personal machine, workstation, mini computer,..)
- Distributed system is used for
  - Resource sharing：provide a mechanism for file sharing, remote printer…
  - Increase computing speed: One computing operation is divided into smaller parts and performed on different places at the same time.
  - Safety: One position get problem, others can continue working

Real-time processing system

- Used mainly in controlling field.
- Solve a problem no late than a specific time.
  - Each problem has a deadline
  - The system must generate correct result in a determined time period
- This OS requires highly cooperate between software and hardware.

Chapter 1 Operating System Overview

Basic Properties of Operating Systems

- High reliability
- Secure
- Effectiveness
- General overtime/ Inherit and adaption
- Convenience

High reliability

- Every actions, notations have to be accurate
  - Only provide information when it's surely correct
    - When error happens: notify and stop the proceed or let the user decide
  - Require support from device
- Example: C:/>COPY C:/F.TXT A:

High reliability

- Example: C:/>COPY C:/F.TXT A:
  - Check the syntax of command copy
  - Check I/O card (motor, drive accessibility)
  - Check for file F.TXT existence in C drive
  - Check A drive
  - Check if file F.TXT already existed in A drive
  - Check if there is enough space in A
  - Check if the disk is write protection
  - Check written information (if required)
  - ....

Basic Properties of Operating Systems

- High reliability
- Secure
- Effectiveness
- General overtime/ Inherit and adaption
- Convenience

Security

- Data and programs have to be protected
  - No unwanted modification in every working mode
  - Secure from illegal access
- Different resources have different protection requirements
- Many levels protections with various of tools
- Important for multi tasking system

Basic Properties of Operating Systems

- High reliability
- Secure
- Effectiveness
- General overtime/ Inherit and adaption
- Convenience

Effectiveness

- Resources are exploited thoroughly;
- Resource that is limited still able to handle complex requirement.
- The system need to maintain the synchronization;
  - Slow devices do not affect the whole system operation

Basic Properties of Operating Systems

- High reliability
- Secure
- Effectiveness
- Generalizable overtime/ Inherit and adaption
- Convenience

Generalizable overtime

- System must be Inheritable
  - Operations, notification can not change
  - If changed: notify and with detailed guide (chkdsk/scandisk)
  - Help keeping and increasing users
- System must have ability to adapt to changes that may happen
- Example: Y2K problem; FAT 12/16/32

Basic Properties of Operating Systems

- High reliability
- Secure
- Effectiveness
- General overtime/ Inherit and adaption
- Convenience

Convenience

- Easy to use
- Various effective levels
- Have many assisting system

Chapter 1 Operating System Overview

① Operating system notion
② History of Operating Systems
③ Definition and Classifications
④ Basic Properties of Operating Systems
⑤ Notions in operating systems
⑥ Operating System structures
⑦ Principles of Operating Systems

Notions in operating systems

- Process and Thread
- System's resources
- Shell
- System calls

Process

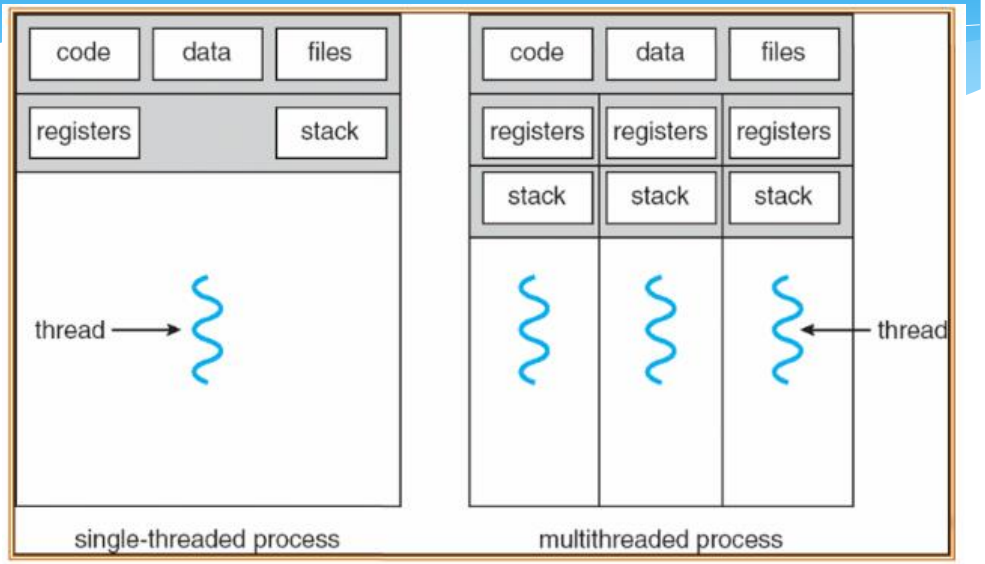
- A running program
  - Codes: Program's executable instruction
  - Program's data
  - Stack, stack pointer, registers
  - Information that is necessary for running program
- Process >< program
  - Program: a passive object, contains computer's instructions to perform a specific task
  - Process: program's active state.

Multi-process timesharing system:

- Periodically: OS pause one process and start another process
  - Need to store processes' information ⇒ process table
- One process can start other process
  - Ex: OS's Shell start a process to perform the command; when the command is done, terminate the started process
- Process can exchange information
- One process can include many threads

Thread

- A sequence/thread of instructions executed in the program
  - Executable code, data
  - Instruction pointer, stack, registers
- Heavyweight Process: process contains one thread
- Lightweight process: contains more than one thread
- Multi_Threading model:
  - Threads running parallel, sharing process's global variables



Notions in operating systems

- Process and Thread
- System's resources
- Shell
- System calls

Definition

- Everything that is necessary for a program to be performed
  - Space: System's storage space
  - Time: Instruction executing time, data accessing time
- System's resources
  - Memory
    - Distinguished by: Storage size, directly access, sequent access
    - Levelled by: main memory/internal; extend, external
    - Notions' distinguish: memory (physical area that contain data) and memory access (the process of searching for the data's location in memory)

- System's resources
  - Memory
  - Processor
    - System's most important component
    - Access level: instruction
    - Processing time
    - Multi-processor system: each processor's time is managed and scheduled independently
  - Peripheral devices
    - Retrieve, output information (I/O device)
    - Attached to the system via controller
    - Commonly considered peripheral devices-controller devices

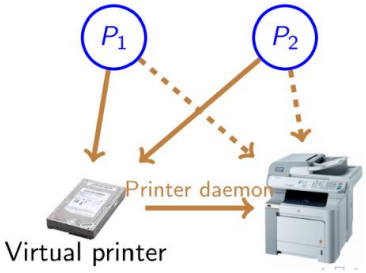
Resource's classification

- Resource's types
  - Physical resource: physical devices
  - Logic resource: variable; virtual devices
- Sharing ability
  - Sharable resource: at a specific time, it can be allocate for different processes. Example: Memory
  - Non-sharable but dividable resource: Processes use the resource follow an order; Example: processor
  - Non-sharable and non-dividable resource: at a specific time, only one process can use the resource. Example: Printer

Virtual printer

- Resource allocated for user's program in a changed form
- Only appear when the system requires or when the system creates it
- Automatically disappear then the system terminate or more precisely, when the process that works with this resource terminates

Example: Virtual printer



Notions in operating systems

- Process and Thread
- System's resources
- Shell
- System calls

- A special process: user and OS communication environment
- Task
  - Receive user's command
  - Analysis received command
  - Generate new process to perform command's requirement
- Receives command from command's line or graphical interface
- Single task environment (MS-DOS):
  - Shell will wait until one process finishes and then receive new command
- In multi-tasking system (UNIX, WINXP,. . .) After creating and running new process, Shell can receive new command

Notions in operating systems

- Process and Thread
- System's resources
- Shell
- System calls

- Provides environment for interacting between user's program and the operating system
  - Programs utilize system calls to request services from operating system
    - Create, delete, use other software objects operated by the operating system
  - Every single system call is corresponding to a library of sub-programs (functions)
- System calls are done in the form of
  - Instructions in low-levels programming languages
  - Interrupt requests (Int) in assembly language
  - API functions calls in Windows
- Input parameters for the services and returned results are located in special memory areas
  - For example: when making request for an interrupt, the function name is stored in the register AH
  - Int 05 : print to monitor ; Int 13/AH=03h : DISK – WRITE/ DISK SECTOR

Example

Func BOOL WINAPI ExitWindowsEx(int uFlags, int dwReason);	
uFlags	**Shutdown types**
EWX_LOGOFF EWX_POWEROFF EWX_ REBOOT	End process and exit Windows Shutdown system and turn off computer Shutdown and restart computer
dwReason	**Reason for shutdown**

File log_off.c

```
#include␣<windows.h>
int␣main(int␣argc,␣char␣*argv[]){
␣␣␣␣ExitWindowsEx(EWX_LOGOFF,␣0);
␣␣␣␣return␣0;
}
```

Chapter 1 Operating System Overview

① Operating system notion
② History of Operating Systems
③ Definition and Classifications
④ Basic Properties of Operating Systems
⑤ Notions in operating systems
⑥ Operating System structures
⑦ Principles of Operating Systems

- System's components
- Operating system's services
- System calls
- Operating system's structures

- Process management
- Main memory management
- Input Output system management
- Files management
- Storage memory management
- Data transmission system (network)
- Protection system
- User interface

Process management

- Process: A running program
- Process utilize system's resources to complete its task
  - Resources are allocated when process created or while it's running
  - Process terminates, resources are returned
- It is possible for many processes to exist in the system at the same time
  - System process
  - User process
- The tasks of OS in process management
  - Create and terminate user's process and system's process
  - Block or re-execute a process
  - Provide mechanism for process synchronization
  - Provide method for processes' communication
  - Provide mechanism for controlling deadlock among processes

Main memory management

- Main memory: an array of byte(word); Each element has an address; where data are accessed by CPU
- To be executed, a program must be given an absolute address and loaded into main memory. When the program is running, the system access instructions and data in main memory.
- To optimize CPU time and computer's speed, some processes are kept in memory
- The role of OS in main memory management
  - Store information about used areas in memory and who used them
  - Decide which process will be fetched into main memory when the memory is available.
  - Allocate and retrieve memory when it's necessary

Input-Output system management

- Objective: hide physical devices' details from users to help them operate easier.
- Input-Output system management includes
  - Memory management of buffering, caching, spooling
  - Communicate with device drivers.
  - Controller for special hardware devices. Only device driver understand its associated-device's specific structure

File management

- Computer can store information on many types of storage devices
- File: storage unit
- File management task
  - Creates/ deletes a file/directory
  - Provides operations over files and directory
  - Reflects file on secondary storage system
  - Backs up file system on storage devices

Storage memory management

- Program is stored in secondary memory (magnetic disk) until it's fetched into main memory and executed.
- Disk is utilized for storing data and processed result.
- Data and result can be stored temporarily on disk: virtual memory
- The role of operating system in disk management
  - Unused area management
  - Provide storage area as requested
  - Schedule disk accessing effectively

Data transmission system (Distributed system)

- Distributed system combined of set of processor (sym/asym) without common clock and memory. Each processor has a local memory.
- Processor connected via transmission network
- Transmission is performed via protocols (FTP, HTTP...)
- Distributed system allow user to access different resource
- Access to sharing resources will allow
  - Increase computing speed
  - Increase data availability
  - Increase the system reliable

System's protection

- Multi users operate with the system at the same time ⇒ Processes have to be protected from other processes' activities
- Protection is a controlling mechanism of program or user's access to system or resource
- Protection mechanism will require
  - Distinguish between legal or illegal usage
  - Set imposed controls
  - Provide tools for imposing

User interface

- Perform user's command. Commands are provide for operating system 's command controller to
  - Create and manage process
  - Manage main memory and storage memory
  - Access file system
  - Protect
  - Network system
  - . . .
- User interface can be command line (DOS, UNIX) or more friendly with graphical interface (Windows, MacOS)

User interface

Some forms of human-computer interact

- Command line
  - Simple but organized
  - Do not require complex system specification
  - Easy to add parameter
- Selection table
  - Menu
  - Popup
  - Menu_popup: 2 method: on and onselect
- Symbol

  window, icon, desktop

Main and basic services

- **Program execution**: the system is able to load the program into memory and execute it. Program must be finish execution in a normal or abnormal (has error) way.
- **Input-output operations**: To increase the performance, programs do not directly access IO devices. The OS has to provide means to perform I/O.
- **File system operations: Program is able to read, write, create or delete file.**
- Communication: Information exchange between running process on the same computer or different computer in the network.
- **Communication is performed via sharing memory or message transferring technique.**
- **Error detection: Confirm work correctly by showing an error is from** CPU, memory, in devices or in programs. Each type of error, OS has a corresponding way to handle.

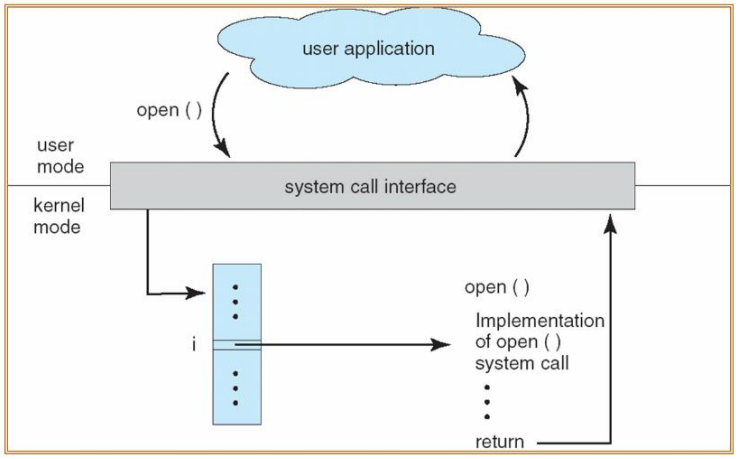
Support services

Not for user but for operate the system effectively

- **Provide resources** Allocate resource for many users or tasks to perform at the same time
- **Report statistics** Store information of types and amount of used resources for computing (usage cost), research (system improvement)
- **Protection** Ensure all access to system resources are controlled

System call

Provide an interface between process and operating system

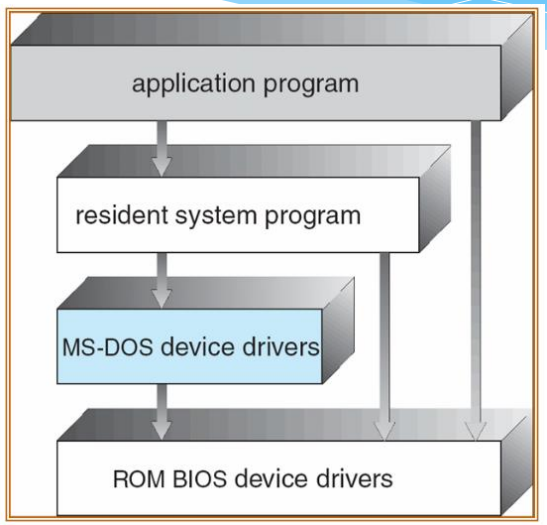


System call

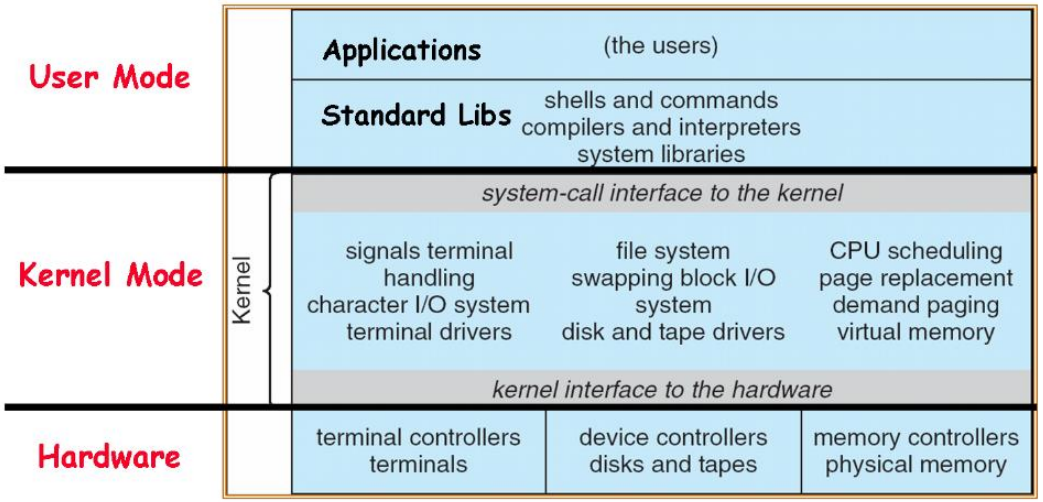
- Process management: initialize, terminate process..
- Memory management: allocate and free memory…
- File management: create, delete, read and write file…
- Input Output device management: perform input/output…
- Exchange information with the system. Example get/set time/date…
- Inter process communication

- System's components
- Operating system's services
- System calls
- Operating system's structures

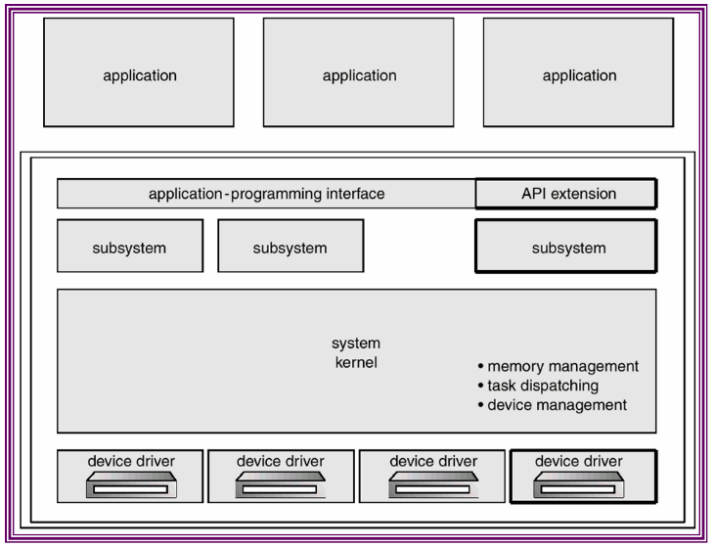
MS-DOS structure (Silberschatz 2002)



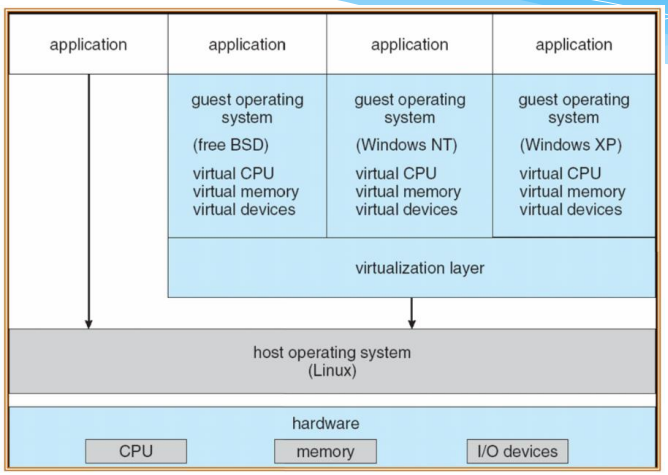
UNIX structure (Silberschatz 2002)



OS/2 structure (Silberschatz 2002)

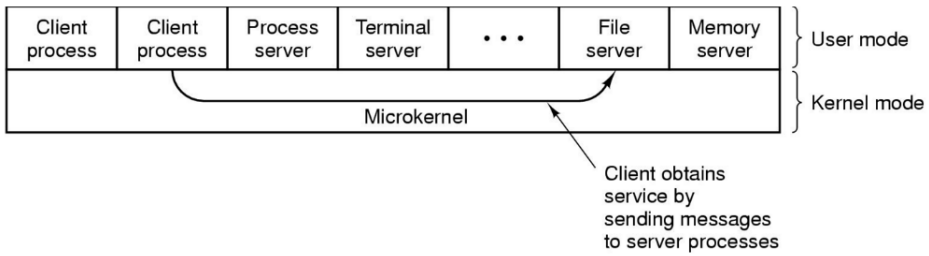


Virtual machine(Silberschatz 2002)

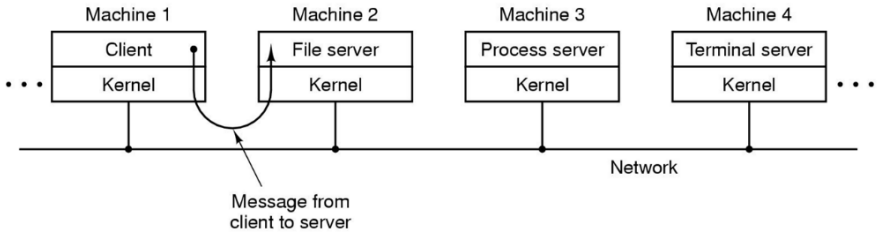


VMWare architecture

Client-Server Model (Tanenbaum 2001)



Client-Server model in distributed OS (Tanenbaum 2001)



Chapter 1 Operating System Overview

① Operating system notion
② History of Operating Systems
③ Definition and Classifications
④ Basic Properties of Operating Systems
⑤ Notions in operating systems
⑥ Operating System structures
⑦ Principles of Operating Systems

Principles of Operating Systems

- Modul principles
- Relatively localization principle
- Macroprocessor principle
- Initialization when start-up principle
- Function overlap principle
- Standard value principle
- Multi-level protections principle

Chapter 1 Operating System Overview

Summary

① Notion of Operating system
- Layering structure of OS
- OS's functions

② History of Operating system
- History of computers
- History of Operating system

③ Definition and classification of OS
- Definitions
- Classification

④ Basic properties of OS
- High reliability
- Security
- Effective
- Generalize overtime
- Convenience

⑤ Notions of Operating system
- Process and Thread
- System's resources
- Shell
- System calls

⑥ Operating system's structure
- OS's components
- OS's services
- System calls
- System's structures

⑦ Principles of Operating System

Hệ Điều Hành

(Nguyên lý các hệ điều hành)

Đỗ Quốc Huy
huydq@soict.hust.edu.vn
Bộ môn Khoa Học Máy Tính
Viện Công Nghệ Thông Tin và Truyền Thông

Chapter 2 Process Management

① Process's definition

Chapter 2 Process Management

Process (review)

- A running program
  - Provided resources (CPU, memory, I/O devices…) to complete its task
  - Resources are provided at:
    - The process creating time
    - While running
- The system includes many processes running at the same time
  - OS's process: Perform system instruction code
  - User's process: Perform user's code
- Process may contain one or more threads
- OS's roles:
  - Guarantee process and thread activities
  - Create/deltete process (user, system)
  - Process scheduling
  - Provide synchronization mechanism, communication and prevent deadlock among processes

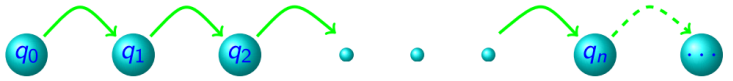
① Process
② Thread
③ CPU scheduling
④ Critical resource and process synchronization
⑤ Deadlock and solutions

- Notion of process
- Process Scheduling
- Operations on process
- Process cooperation
- Inter-process communication

System's state

- Processor: Registers' values
- Memory: Content of memory block
- Peripheral devices: Devices' status

- Program's executing ⇒ system's state changing
  - Change discretely, follow each executed instruction



- Process: a sequence of system's state changing
  - Begin with start state
  - Change from state to state is done according to requirement in user's program

Process is the execution of a program

Process >< program

- ***Program***: *passive object (content of a file on disk)*
  - *Program's code: machine instruction (CD2190EA...)*
  - *Data:*
    - *Variable stored and used in memory*
    - *Global variable*
    - *Dynamic allocation variable (malloc, new,..)*
    - *Stack variable (function's parameter, local variable)*
  - *Dynamic linked library (DLL)*
    - *Not compiled and lined with program*

When program is running, minimum resource requirement

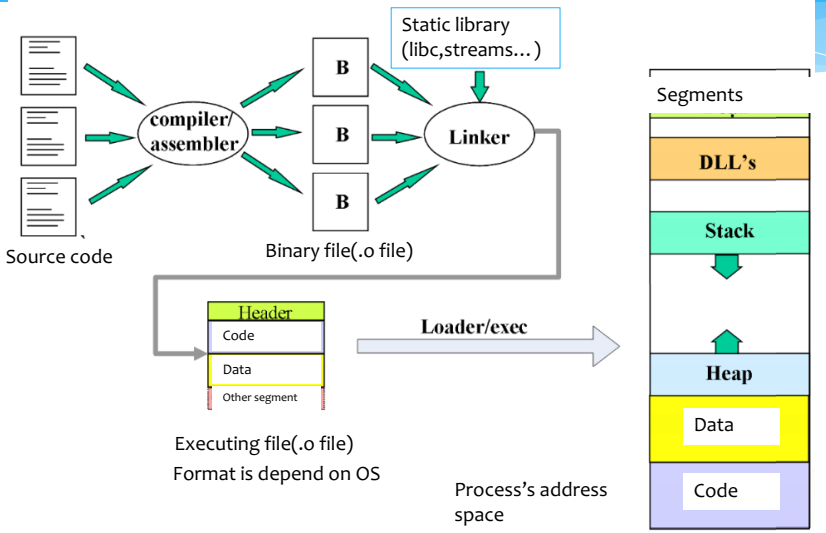
- *Memory for program's code and data*
- *Processor's registers used for program execution*

- ***Process***: *active object (instruction pointer, set of resources)*

A program can be

- *Part of process's state*
  - *One program, many process (different data set)*
    Example: gcc hello.c || gcc baitap.c
- *Call to many process*

Compile and run a program



Compile and run a program

- *The OS creates a process and allocate a memory area for it*
- *System program loader/exec*
  - *Read and interprets the executive file (file's header)*
  - *Set up address space for process to store code and data from executive file*
  - *Put command's paramenters, environment variable (argc, argv, envp) into stack*
  - *Set proper values for processor's registerand call "_start()" (OS's fucntion)*
- *Program begins to run at "_start()". This function call to main() (program's functions)*

⇒*"Process" is running, not mention "program" anymore*

- *When main() function end, OS call to "_exit()" to cancel the process and take back resources*

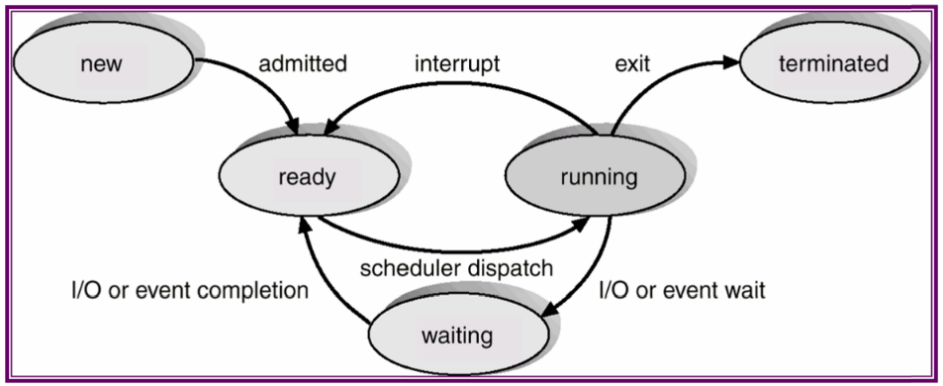
Process's state

When executing, process change its state

- New Process is being initialized
- Ready Process is waiting for its turn to use physical processor
- Running Process's instructions are being executed
- Waiting Process is waiting for an event to appear (I/O operation completion)
- Terminated Process finish its job

Process's state is part of process current's activity

Process's states changing flowchart (Silberschatz 2002)

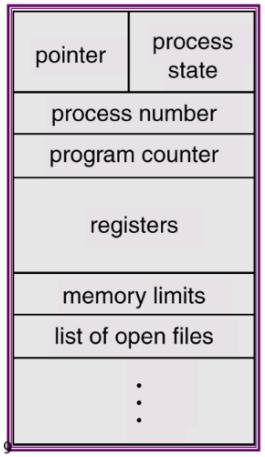


System that has only one processor

- Only one process in running state
- Many processes in ready or waiting state

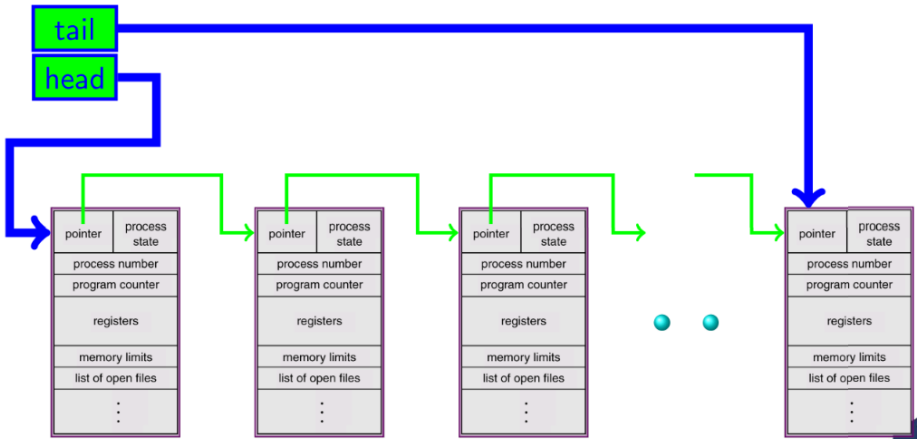
Process Control Block (PCB)

- Each process is presented in the system by a process control block
- PCB: an information structure allows identify only one process



- Process state
- Process counter
- CPU's registers
- Information for process scheduling
- Information for memory management
- Inforamtion of usable resources
- Statistic information
- Pointer to another PCB
- . . .

Process List



Single-thread process and Multi-thread process

- Single-thread process : A running process with only one executed thread
  Have one thread of executive instruction
  ⇒ Allow executing only one task at a moment

- Multi-thread process : Process with more than one executed thread
  ⇒ Allow more than one task to be done at a moment

- Notion of process
- Process Scheduling
- Operations on process
- Process cooperation
- Inter-process communication

Introduction

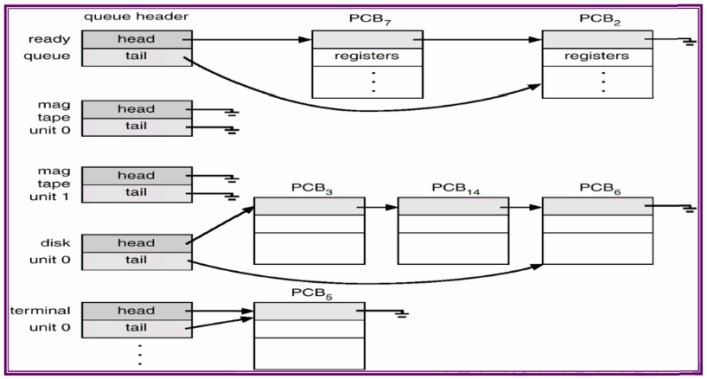
Objective Maximize CPU's usage time
⇒ Need many processes in the system
Problem CPU switching between processes
⇒ Need a queue for processes

Single processor system
⇒ One process running
⇒ Other processes have to wait until processor is free

Process queue I

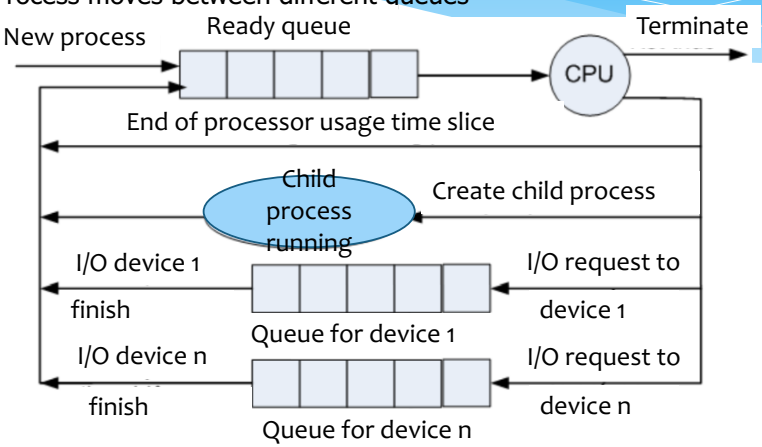
The system have many queues for processes

- **Job-queue Set of processes in the system**
- **Ready-Queue Set of processes exist in the memory, ready to run**
- **Device queues Set of processes waiting for an I/O device. Queues for each device are different**



Process queue II

- Process moves between different queues

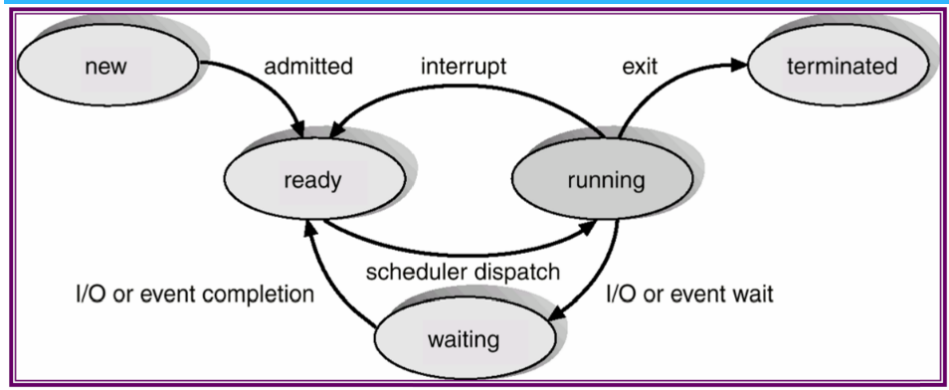


- Newly created process is putted in ready queue and wait until it's selected to execute

Process queue III

- Process selected and running
  - ①process issue an I / O request, and then be placed in an I / O queue
  - ②process create a new sub-process and wait for its termination
  - ③process removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue
- In case (1&2) after the waiting event is finished,
  - Process eventually switches from waiting state to ready state
  - Process is then put back to ready queue
- Process continues this cycle (ready, running, waiting) until it terminates.
  - it is removed from all queues
  - its PCB and resources deallocated

Scheduler



*Selection process is carried out by the appropriate scheduler

- Job scheduler; Long-term scheduler
- CPU scheduler; Short-term scheduler

Job Scheduler

- Select processes from process queue stored in disk and put into memory to execute
- Not frequently (seconds/minutes)
- Control the degree of the multi-programming (number of processin memory)
- When the degree of multi-programming is stable, the scheduler may need to be invoked when a process leaves the system

- Job selection problem
  - I/O bound process: use less CPU time
  - CPU bound process: use more CPU time
  - Should select good process mix of both

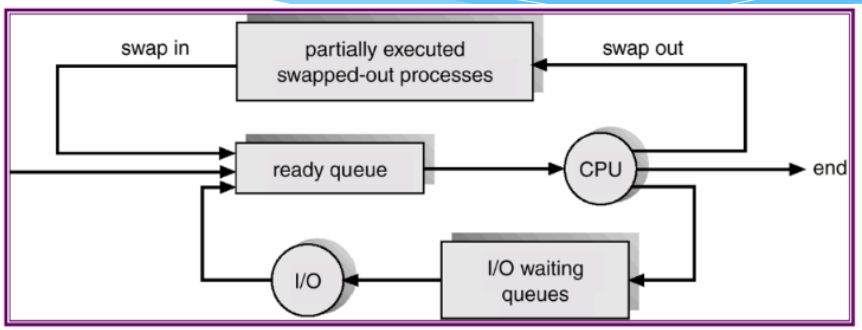
  ⇒ I/O bound process: ready queue is empty, waste CPU

  ⇒ CPU bound: device queue is empty, device is wasted

CPU Scheduler

* Selects from processes that are ready to execute, and allocates the CPU to one of them
- Frequently work (example: at least 100ms/once)
  - A process may execute for only a few milliseconds before waiting for an I/O request
  - Select new process that is ready
- the short-term scheduler must be fast
  - 10ms to decide ⇒10/(110)=9% CPU time is wasted
- Process selection problem (CPU scheduler)

Process swapping (Medium-term scheduler)



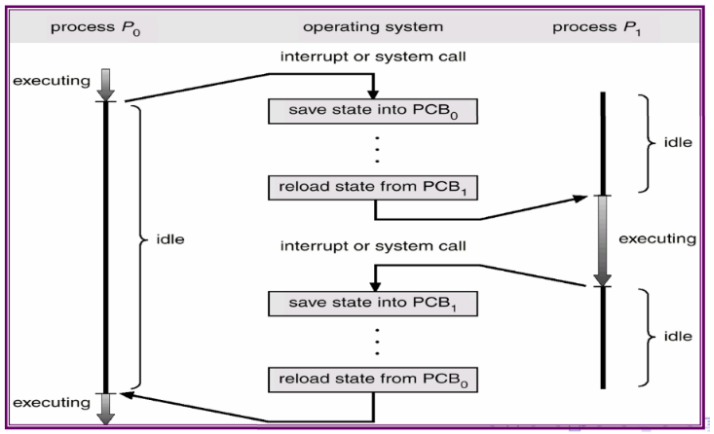
- Task
  - Removes processes from memory, reduces the degree of multiprogramming
  - Process can be reintroduced into memory and its execution can be continued where it left off
- Objective: Free memory, make an wider unused memory area

Context switch

- **Switch CPU from one process to another process**
  - Save the state of the old process and loading the saved state for the new process
    - includes the value of the CPU registers, the process state and memory-management information
  - Take time and CPU cannot do anything while switching
  - Depend on the support of the hardware
  - More complex the operating system, the more work must be done during a context switch

Context switch

- Occurs when an intterupt signal appear(timely interrupt) or when process issue a system call (to perform I/O)
- CPU switching diagram(Silberschatz 2002)



- CPU switch from this process to another process (switch the running process)

- Notion of process
- Process Scheduling
- Operations on process
- Process cooperation
- Inter-process communication

Operations on process

- Process Creation
- Process Termination

Process Creation

- Process may create several new processes (CreateProcess(), fork())
  - The creating process: parent-process
  - The new process: child-process
- Child-process may create new process ⇒ Tree of process
- Resource allocation
  - Child process receive resource from the OS
  - Child process receive resource from parent-process
    - All of the resource
    - a subset of the resources of the parent process (to prevent process from creating too many child process)
- When a process creates a new process, two possibilities exist in terms of execution:
  - The parent continues to execute concurrently with its children
  - The parent waits until some or all of its children have terminated

Process Termination

- Finishes executing its final statement and asks the OS to delete (exit)
  - Return data to parent process
  - Allocated resources are returned to the system
- Parent process may terminate the execution of child process
  - Parent must know child process's id ⇒ child process has to send its id to parent process when created
  - Use the system call (abort)
- Parent process terminate child process when
  - The child has exceeded its usage of some of the resources
  - The task assigned to the child is no longer required
  - The parent is exiting, and the operating system does not allow a child to continue
  - ⇒Cascading termination. Example: *turn off computer*

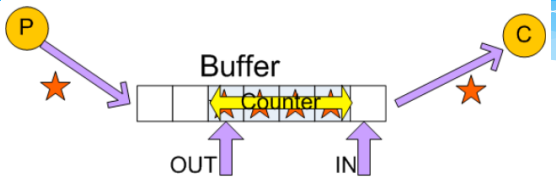
Some functions with process in WIN32 API

- CreateProcess(. . .)
  - LPCTSTR Name of the execution program
  - LPTSTR Command line paramenter
  - LPSECURITY_ATTRIBUTES A pointer to process **security attribute** structure
  - LPSECURITY_ATTRIBUTES A pointer to thread **security attribute** structure
  - BOOL allow process inherit devices (TRUE/FALSE)
  - DWORD process creation flag (Example: CREATE_NEW_CONSOLE)
  - LPVOID Pointer to environment block
  - LPCTSTR full path to program
  - LPSTARTUPINFO info structure for new process
  - LPPROCESS_INFORMATION Information of new process
- *TerminateProcess(HANDLE hProcess, UINT uExitCode)*
  - *hProcess* A handle to the process to be terminated
  - *uExitCode process termination code*
- *WaitForSingleObject(HANDLE hHandle, DWORD dwMs)*
  - *hHandle Object handle*
  - *dwMs Waiting time (INFINITE)*

Processes' relation classification

- Sequential processes
  - The start point of one process lie after the finish point of the other process
- Parallel processes
  - The start point of one process lie between the start and finish point of the other process
  - Independence: Not affect or affected by other running process in the system
  - Cooperation: affect or affected by other running process in the system
  - Cooperation in order to
    - Information sharing
    - Computation speedup
    - Modularity
    - Increase the convenience
  - Process collaborate require mechanism that allow process to
    - Communicate with one another
    - Synchronize their actions

Producer - Consumer Problem I



- The system includes 2 processes
  - Producer creates product
  - Consumer consumes created product
- Application
  - Print program (producer) creates characters that are consumed by printer driver (consumer)
  - Compiler (producer) creates assembly code, Assembler (consumer/producer) consumes assembly code and generate object module which is consumed by the exec/loader (consumer)

Producer - Consumer Problem II

- *Producer* and *Consumer* work simultaneously
  Use the sharing **Buffer** which store product filled in by producer and taken out by consumer
  - IN Next empty place in buffer
  - OUT First filled place in the buffer.
  - Counter Number of products in the buffer
- Producer and Consumer must be synchronized
  - Consumer does not try to consume a product that was not created
- Unlimited size buffer
  When buffer is empty, Consumer need to wait
  Producer can put product into buffer without waiting
- Limited size buffer
  - When Buffer is empty, *Consumer* need to wait
  - *Producer* need to wait if the Buffer is full

Producer - Consumer Problem II

Producer

```
while(1) {
    /*produce an item in nextProduced*/
    while (Counter == BUFFER_SIZE) ; /*do nothing*/
    Buffer[IN] = nextProduced;
    IN = (IN + 1) % BUFFER_SIZE;
    Counter++;
}
```

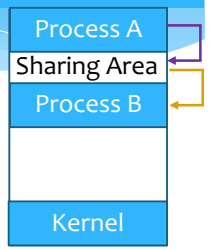
Consumer

```
while(1){
     while(Counter == 0) ; /*do nothing*/
      nextConsumed = Buffer[OUT];
      OUT =(OUT + 1) % BUFFER_SIZE;
     Counter--; /*consume the item in nextConsumed*/
}
```

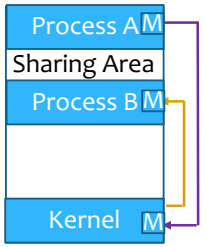

- Notion of process
- Process Scheduling
- Operations on process
- Process cooperation
- Inter-process communication

Communicate between process

- Memory sharing model
  - Process share a common memory area
  - Implementation code are written explicitly by application programmer
  - Example: Producer-Consumer problem



- Inter-process communication model
  - A mechanism allow processes to communicate and synchronize
  - Often used in distributed system where processes lie on different computers (chat)
  - Guaranteed by message passing system



Message passing system

- Allow processes to communicate without sharing variable
- Require two basic operations
  - Send (msg) msg has fixed or variable size
  - Receive (msg)
- If 2 processes P and Q want to communicate, they need to
  - Establish a communication link (physical/logical) between them
  - Exchange messages via operations: send/receive

Direct Communication

- Each process that wants to communicate must explicitly name the recipient or sender of the communication
  - send (P, message) – Send a message to process P
  - receive(Q, message) – Receive a message from process Q
- The properties of a communication link
  - A link is established automatically
  - A link is associated with exactly two processes
  - Exactly one link exists between each pair of processes
  - Link can be one direction but often two directions

Indirect Communication

- Messages are sent to and received from mailboxes, or ports
- Each mailbox has a unique identification
  - Two processes can communicate only if they share a mailbox
- Communication link's properties
  - Established only if both members of the pair have a shared mailbox
  - A link may be associated with more than two processes
  - Each pair of communicating processes may have many links
    - Each link corresponding to one mailbox
  - A link can be either one or two direction
- Operations:
  - Create a new mailbox
  - Send and receive messages through the mailbox
    - send(A, msg): send a msg to mailbox A
    - receive(A, msg): Receuve a msg from mailbox A
  - Delete a mailbox

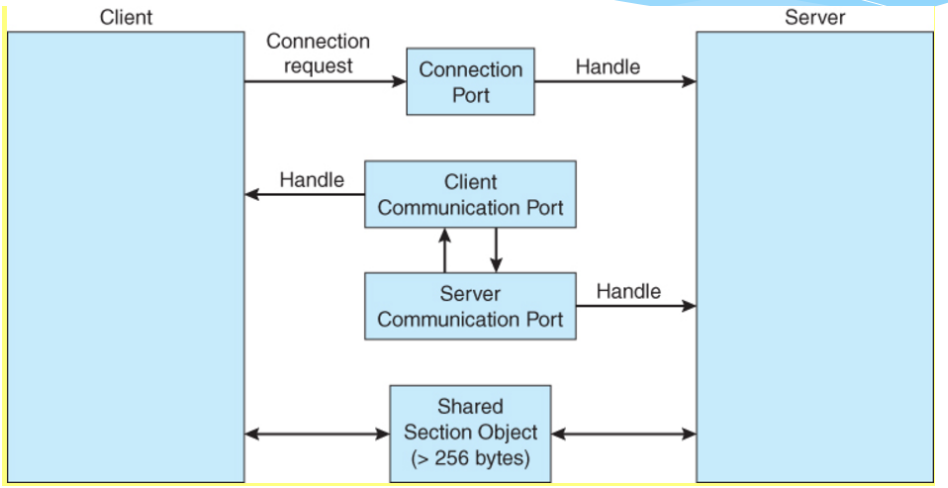
Synchronization

- Message passing may be either blocking or nonblocking-
  - Blocking synchronous communication
  - Non-blocking asynchronous communication
- send() and receive() can be blocking or non-blocking
  - Blocking send sending process is blocked until the message is received by the receiving process or by the mailbox
  - Non-blocking send The sending process sends the message and resumes operation
  - Blocking receive The receiver blocks until a message is available
  - Non-blocking receive The receiver retrieves either a valid message or a null

Buffering

- Messages exchanged by communicating processes reside in a temporary queue
- A queue can be implemented in three ways
  - Zero capacity: maximum length 0 => the link cannot have any messages waiting in it
    - sender must block until the recipient receives the message
  - Bounded capacity
    - Queue has finite length n ⇒ store at most n messages
    - If the queue is not full, message is placed in the queue and the sender can continue execution without waiting
    - If the link is full, the sender must block until space is available in the queue
  - Unbound capacity
    - The sender never blocks

Windows XP Message Passing



Communication in Client - Server Systems with Sockets

- Socket is defined as an endpoint for communication,
  - Each process has one socket
- Made up of an IP address and a port number. E.g.: 161.25.19.8:1625
  - IP Address: Computer address in the network
  - Port: identifies a specific process
- Types of sockets
  - Stream Socket: Based on TCP/IP protocol → Reliable data transfer
  - Datagram Socket: based on UDP/IP protocol → Unreliable data transfer
- Win32 API: Winsock
  - Windows Sockets Application Programming Interface

Winsock API 32 functions

socket()Tạo socket truyền dữ liệu
bind() Định danh cho socket vừa tạo (gán cho một cổng)
listen() Lắng nghe một kết nối
accept() Chấp nhận một kết nối
connect() kết nối với server.
send() Gửi dữ liệu với stream socket.
sendto() Gửi dữ liệu với datagram socket.
receive() Nhận dữ liệu với stream socket.
recvfrom() Nhận dữ liệu với datagram socket.
closesocket() Kết thúc một socket đã tồn tại.
.....

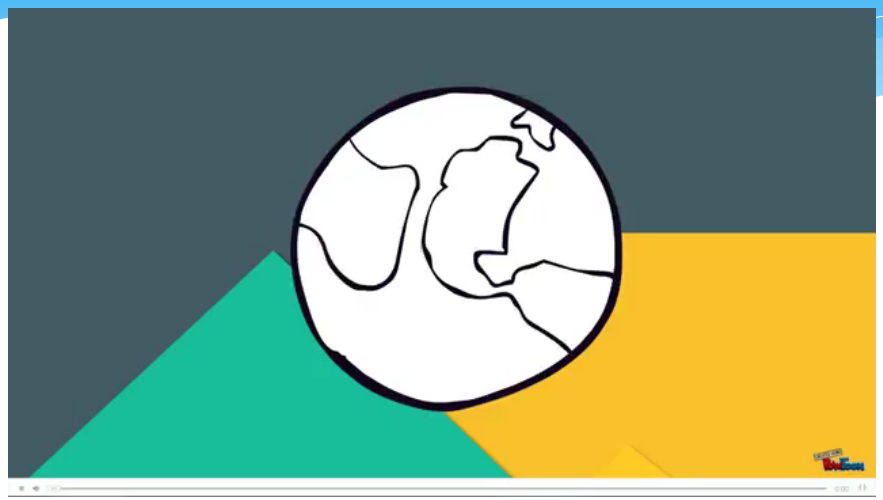
Homework

Use Winsock to make a Client-Server program

Chat program.
.....

① Process
② Thread
③ CPU scheduling
④ Critical resource and process synchronization
⑤ Deadlock and solutions

- Introduction
- Multithreading model
- Thread implementation with Windows
- Multithreading problem

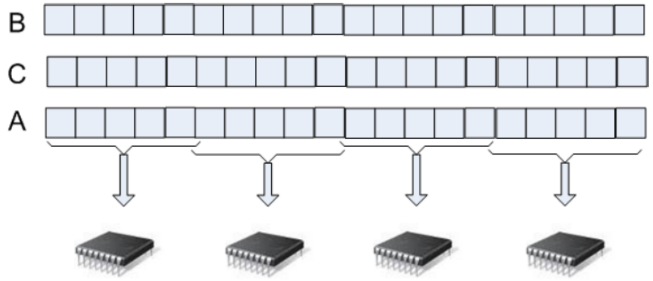


Example: Vector computation

- Large size vector computing

```
For (k = 0;k < n;k++) {
    a[k] = b[k]*c[k];
}
```

With multi processors system



Example: Chat

Process P

```
While (1) {
  ReadLine(Msg);
  Send(Q,Msg);
  Receive(Q,Msg);
  PrintLine(Msg);
}
```

Msg receiving problem

- Blocking Recieve
- Non-blocking Receive

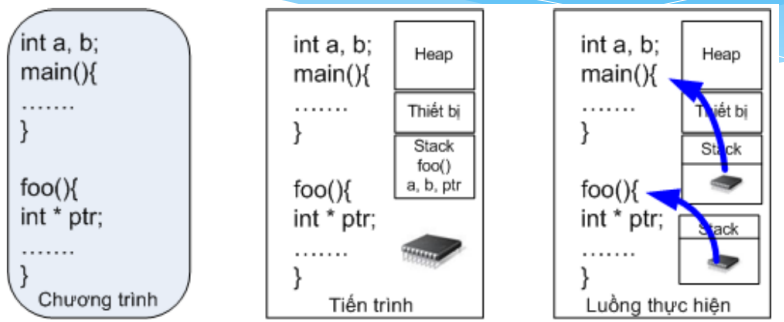
Solution

Concurrently perform Receive & Send

Process Q

```
While (1) {
  Receive(P,Msg);
  PrintLine(Msg);
  ReadLine(Msg);
  Send(P,Msg);
}
```

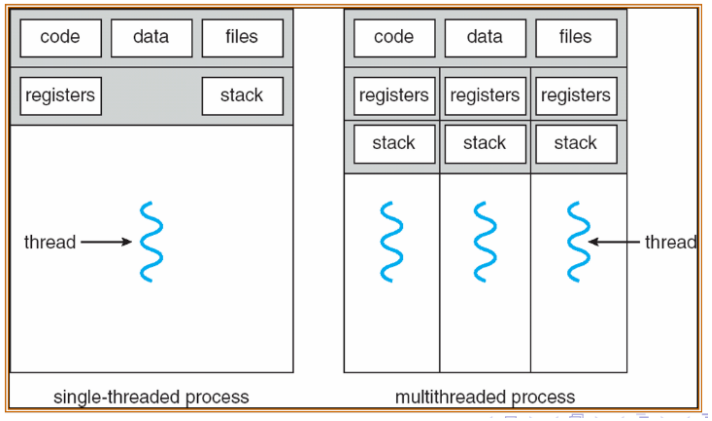

Program - Process - Thread



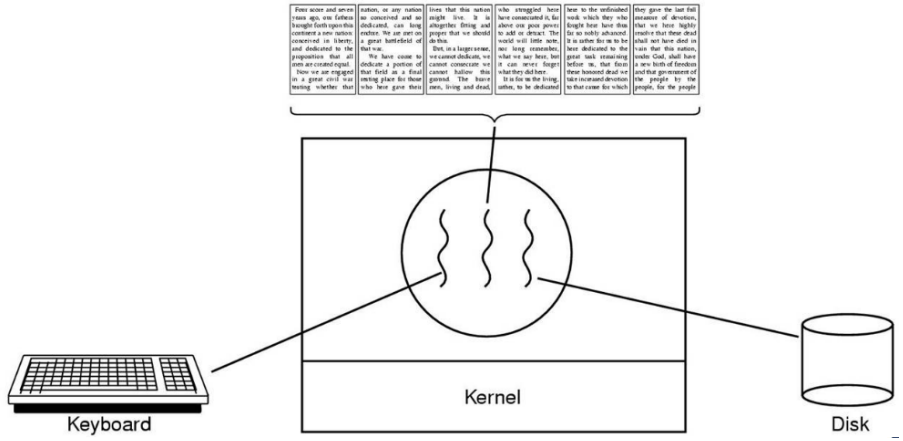
- Program: Sequence of instructions, variables,..
- Process: Running program: Stack, devices, processor,..
- Thread: A running program in process context
  - Multi-processor → Multi threads, each thread runs on one processor
    - Different in term of registers' values, stack's content

Single-threaded and multi-threaded process

- Traditional operating system(MS-DOS, UNIX)
  - Process has one controlling thread (heavyweight process)
- Modern operating system (Windows, Linux)
  - Process may have many threads
  - Perform many task at a single time



Example: Word processor (Tanenbaum 2001)



Notion of thread

- Basic CPU using unit, consists of
  - Thread ID
  - Program Counter
  - Registers
  - Stack space
- Sharing between threads in the same process
  - Code segment
  - Data segment (global objects)
  - Other OS's resources (opening file…)
- Thread can run same code segment with different context (*Register set, program counter, stack*)
- LWP: Lightweight Process
- A process has at least one thread

Distinguishing between process and thread

Process	Thread
Has code/data/heap and other segments	No separating code or heap segment
Has at least one thread	Not stand alone but must be inside a process
Threads in the same process share code/data/heap, devices but have separated stack and registers	Many threads can exist at the same time in each process. First thread is the main thread and own process's stack
Create and switch process operations are expansive	Create and switch thread operations are inexpensive
Good protection due to own address space	Common address space, need to protect
When process terminated, resources are returned and threads have to terminated	Thread terminated, its stack is returned

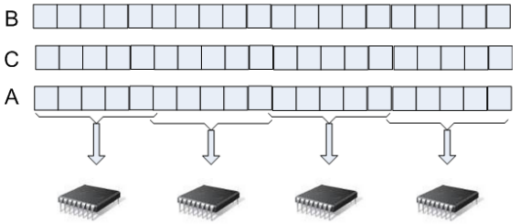
Benefits

- Responsiveness
  - allow a program to continue running even if part of it is blocked or is performing a long operation
  - Example: A multi-threaded Web browser
    - One thread interactives with user
    - One thread downloads data
- Resource sharing
  - threads share the memory and the resources of the process
    - Good for parallel algorithms (share data structures)
    - Threads communicate via sharing memory
  - Allow application to have many threads act in the same address space
- Economy
  - Create, switch, terminate threads is less expensive than process
- Utilization of multiprocessor architectures
  - each thread may be running in parallel on a different processor.

Benefit of multithreading-> example

Vector computing

```
for (k = 0;k < n;k++) {
    a[k] = b[k]*c[k];
}
```

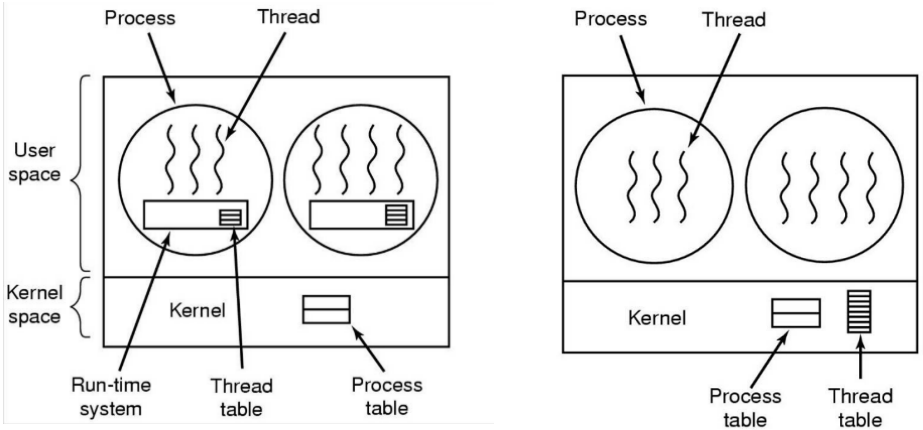


Multi-threading model

```
void fn(a,b)
for(k = a; k < b; k ++){
      a[k] = b[k] * c[k];
}
void main(){
   CreateThread(fn(0, n/4));
   CreateThread(fn(n/4, n/2));
   CreateThread(fn(n/2, 3n/4));
   CreateThread(fn(3n/4, n));
}
```


Thread implementation

User space | Kernel space



User -Level Threads

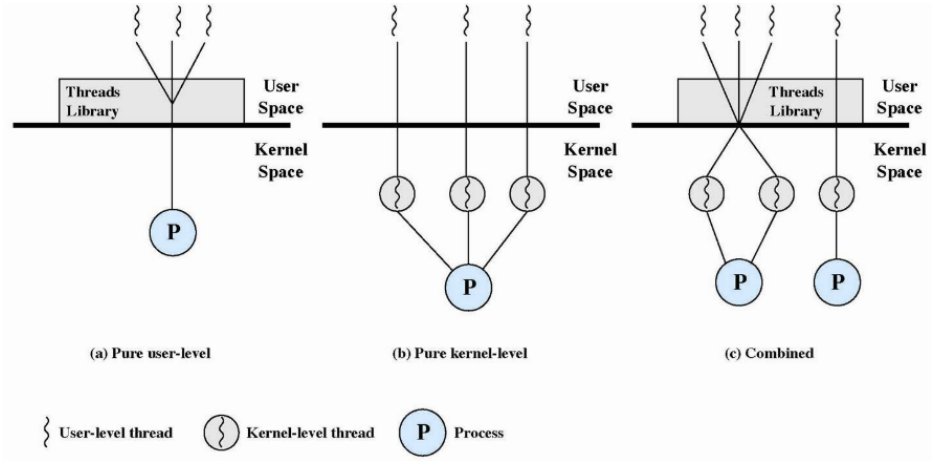
- Thread management is done by application
- Kernel does not know about thread existence
  - Process scheduling like a single unit
  - Each process is assigned with a single state
    - Ready, waiting, running,..
- User threads are supported above the kernel and are implemented by a thread library
  - Library support creation, scheduling and management..
- Advantage
  - Fast to create and manage
- Disadvantage
  - if a thread perform blocking system call , the entire process will be blocked ⇒ Cannot make use of multi-thread.

Kernel - Level threads

- Kernel keeps information of process and threads
- Threads management is performed by kernel
  - No thread management code inside application
  - Thread scheduling is done by kernel
- Disadvantage:
  - Slow in thread creation and management
- Advantage:
  - One thread perform system call (e.g. I/O request), other threads are not affected
  - In a multiprocessor environment, the kernel can schedule threads on different processors
- Operating system support kernel thread: Windows NT/2000/XP, Linux, OS/2,..

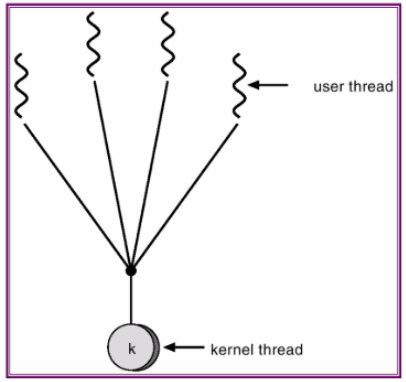
Introduction

- Many systems provide support for both user and kernel threads -> different multithreading models

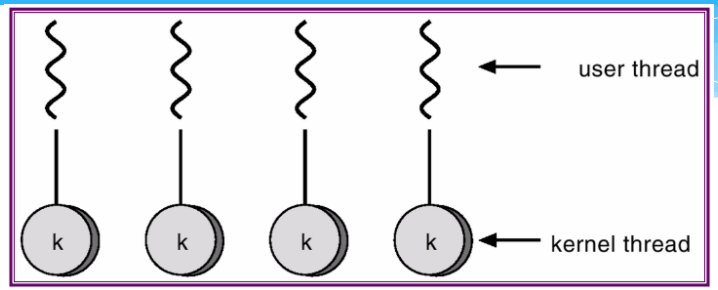


Many-to-One Model

- Maps many user-level threads to one kernel thread.
- Thread management is done in user space
  - Efficient
  - the entire process will block if a thread makes a blocking system call
  - multiple threads are unable to run in parallel on multi processors
- implemented on operating systems that do not support kernel threads use the many-to-one model



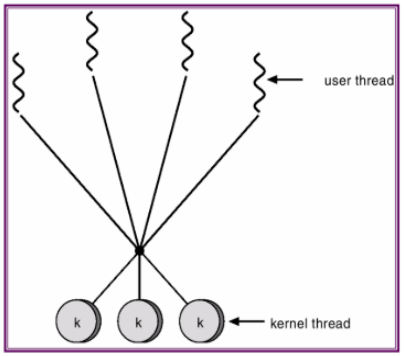
One-to-one Model



- Maps each user thread to a kernel thread
- Allow another thread to run when a thread makes a blocking system call
- Creating a user thread requires creating the corresponding kernel thread
  - the overhead of creating kernel threads can burden the performance of an application
  - ⇒ restrict the number of threads supported by the system
- Implemented in Window NT/2000/XP

Many-to-Many Model

- Multiplexes many user-level threads to a smaller or equal number of kernel threads
- Number of kernel threads: specific to either a particular application or a particular machine
  - E.g.: on multiprocessor -> more kernel thread than on uniprocessor
- Combine advantages of previous models
  - Developers can create as many user threads as necessary
  - kernel threads can run in parallel on a multiprocessor
  - when a thread performs a blocking system call, the kernel can schedule another thread for execution
- Supported in: UNIX



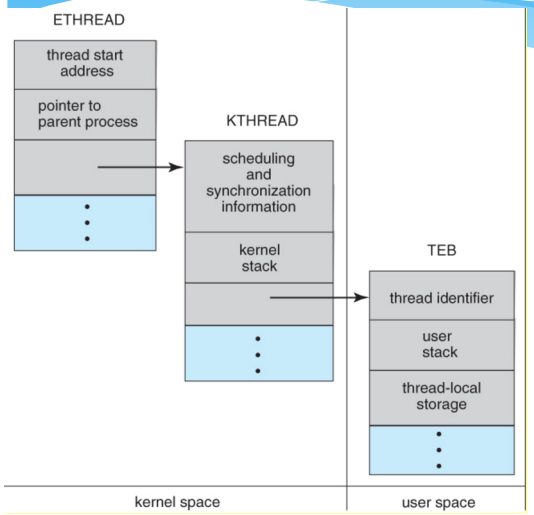
Functions with thread in WIN32 API

- HANDLE CreateThread(. . .);
  - LPSECURITY_ATTRIBUTESlpThreadAttributes,
    ⇒A pointer to a SECURITY_ATTRIBUTES structure that determines whether the returned handle can be inherited by child processes
  - DWORD dwStackSize,
    ⇒The initial size of the stack, in bytes
  - LPTHREAD_START_ROUTINE lpStartAddress,
    ⇒ pointer to the application-defined function to be executed by the thread
  - LPVOID lpParameter,
    ⇒A pointer to a variable to be passed to the thread
  - DWORD dwCreationFlags,
    ⇒The flags that control the creation of the thread
    - CREATE_SUSPENDED : thread is created in a suspended state
    - 0: thread runs immediately after creation
  - LPDWORD lpThreadId
    ⇒pointer to a variable that receives the thread identifier
- Return value: handle to the new thread or NULL if the function fails

Example

```
#include <windows.h>
#include <stdio.h>
void Routine(int *n){
printf("My argument is %d\n", &n);
}
int main(){
      int i, P[5]; DWORD Id;
       HANDLE hHandles[5];
       for (i=0;i < 5;i++) {
           P[i] = i;
           hHandles[i] = CreateThread(NULL,0,
                        (LPTHREAD_START_ROUTINE)Routine,&P[i],0,&Id);
           printf("Thread %d was created\n",Id);
       }
       for (i=0;i < 5;i++) WaitForSingleObject(hHandles[i],INFINITE);
    return 0;
}
```


Thread in Windows XP



Thread includes

- Thread ID
- Registers
- user stack used in user mode, kernel stack used in kernel mode.
- Separated memory area used by runtime and dynamic linked library

Executive thread block
Kernel thread block
Thread environment block

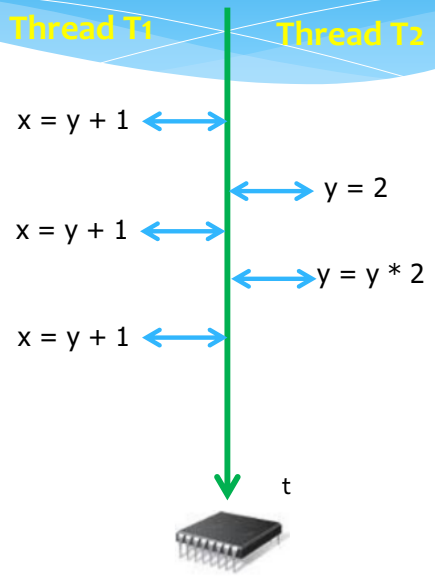
- Introduction
- Multithreading model
- Thread implementation with Windows
- Multithreading problem

Example

```
#include <windows.h>
#include <stdio.h>
int x = 0, y = 1;
void T1(){
      while(1){ x = y + 1; printf("%4d", x); }
}
void T2(){
      while(1){ y = 2; y = y * 2; }
}
int main(){
      HANDLE h1, h2; DWORD Id;
      h1=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)T1,NULL,0,&Id);
      h2=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)T2,NULL,0,&Id);
      WaitForSingleObject(h1,INFINITE);
      WaitForSingleObject(h2,INFINITE);
      return 0;
}
```

Explanation

Shared int y = 1	
Thread T_1	**Thread** T_2
x ←y+ 1	y ←2 y ← y * 2
x = ?	



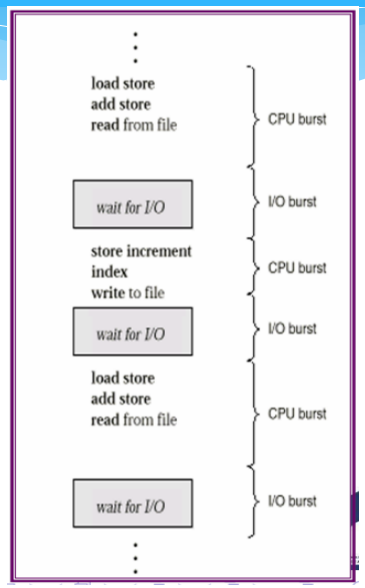
The result of parallel running threads depends on the order of accessing the sharing variable

Introduction

- System has one *processor* → Only one process is running at a time
- Process is executed until it must wait, typically for the completion of some I/O request
  - Simple system: CPU is not be used ⇒ Waste CPU time
  - Multiprogramming system: try to use this time productively, give CPU for another process
    - Several ready processes are kept inside memory at one time
    - When one process has to wait, OS takes the CPU away and gives the CPU to another process
- Scheduling is a fundamental operating-system function
  - Switch CPU between process → exploit the system more efficiently

CPU-I/0 Burst Cycle

- Process execution consists of a cycle of CPU execution and I/O wait
  - begins with a CPU burst
  - followed by an I/O burst
  - CPU burst → I/O burst → CPU burst → I/O burst → . . .
  - End: the last CPU burst will end with a system request to terminate execution
- Process classification
  - Based on distribution of CPU & I/O burst
    - CPU-bound program might have a few very long
    - I/O-bound program would typically have many very short CPU bursts
  - help us select an appropriate CPU-scheduling algorithm



CPU Scheduler

- CPU idle -> select one process from ready queue to be executed
  - Process in ready queue
    - FIFO queue, Priority queue, Simple linked list . . .
- CPU scheduling decisions may take place when
  1) Process switches from the running state to the waiting state (I/O request)
  2) Process switches from the running state to the ready state (out of CPU usage time→ time interrupt)
  3) Process switches from the waiting state to the ready state (e.g. completion of I/O)
  4) Process terminates
- Note
  - Case 1&4 ⇒ non-preemptive scheduling scheme
  - Other cases ⇒ preemptive scheduling scheme

Preemptive and non-preemptive Scheduling

- Non-preemptive scheduling
  - Process keeps the CPU until it releases the CPU either by
    - terminating
    - switching to the waiting state
    - does not require the special hardware (timer)
  - Example: DOS, Win 3.1, Macintosh
- Non-preemptive scheduling
  - Process only allowed to run in specified period
    - End of period, time interrupt appear, dispatcher is invoked to decide to resume process or select another process
  - Protect CPU from “CPU hungry" processes
  - Sharing data problem
    - Process 1 updating data and the CPU is taken
    - Process 2 executed and read data which is not completely update
  - Example: Multiprogramming OS WinNT, UNIX

- Basic Concepts
- Scheduling Criteria
- Scheduling algorithms
- Multi-processor scheduling

Scheduling Criteria I

- CPU utilization
  - keep the CPU as busy as possible
  - should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system)
- Throughput
  - number of processes completed per time unit
    - Long process: 1 process/hour
    - Short processes: 10 processes/second
- Turnaround time
  - Interval from the time of submission of a process to the time of completion
  - Can be the sum of
    - periods spent waiting to get into memory
    - waiting in the ready queue
    - executing on the CPU
    - doing I/O

Scheduling Criteria II

- Waiting time
  - sum of the periods spent waiting in the ready queue
  - CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue
- Response time
  - time from the submission of a request until the first response is produced
    - process can produce some output fairly early
    - continue computing new results while previous results are being output to the user

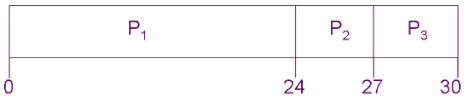
- Assumption: Consider one CPU burst (ms) per process
- Measure: Average waiting time

FCFS: First Come, First Served

- Rule: process that requests the CPU first is allocated the CPU first

  Process own CPU until it terminates or blocks for I/O request
- Example

Process	Burst Time
P_1	24
P_2	3
P_3	3



- Pros and cons
  - Simple, easy to implement
  - Short process has to wait like short process
    - If P1 executed last?

SJF: Shortest Job First

- Rule: associates with each process the length of the latter's next CPU burst
  - process that has the smallest next CPU burst
  - Two methods
    - Non-preemptive
    - preemptive (SRTF: Shortest Remaining Time First)
- Example

Process	Burst Time	Arrival Time
P_1	8	0.0
P_2	4	1.0
P_3	9	2.0
P_4	5	3.0

- Pros and Cons
  - SJF (SRTF) is optimal: Average waiting time is minimum
  - It's not possible to predict the length of next CPU burst
    - Predict based on previous one

Priority Scheduling

- Each process is attached with a priority value (a number)
  - CPU is allocated to the process with the highest priority
  - SJF: priority (p) is the inverse of the (predicted) next CPU burst
  - Two method
    - Non-preemptive
    - Preemptive
- Example

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

- "Starvation" problem: low-priority process has to wait indefinitely (even forever)
- Solution: increase priority by the time process wait in the system

Round Robin Scheduling

- Rule
  - Each process is given a time quantum (time slice) τ to be executed
  - When time's up processor is preemptive and process is placed in the last position of ready queue
  - If there are n process, longest waiting time is $(n-1)\tau$
- Example

Process	Burst Time
P_1	24
P_2	3
P_3	3

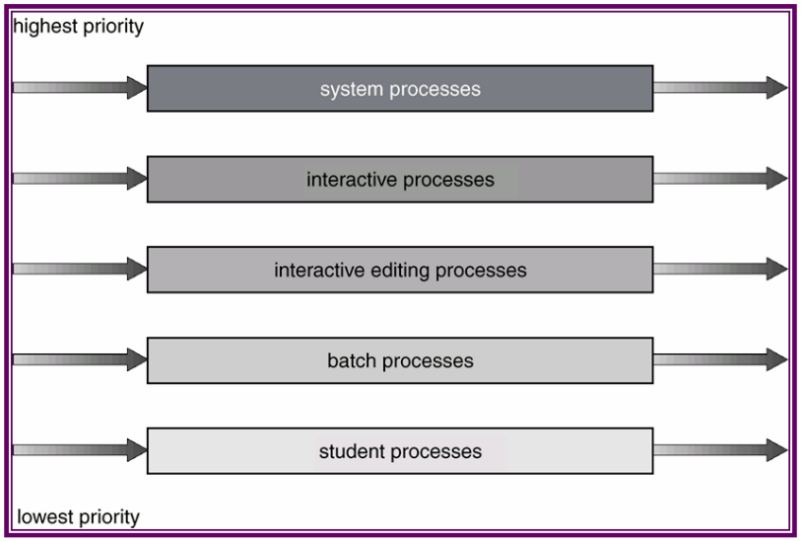
quantum $\tau = 4$

- Problem: select τ
  - τ large: FCFS
  - τ small: CPU is switched frequently
  - Commonly τ = 10-100ms

Multilevel Queue Scheduling

- Partitions the ready queue into several separate queue
- Processes are permanently assigned to one queue
  - Based on some property of the process, such as memory size, process priority, or process type..
- Each queue has its own scheduling algorithm
- There must be scheduling among the queues
  - Commonly implemented as fixed-priority preemptive scheduling
    - Processes in lower priority queue only executed if higher priority queues are empty
    - High priority process preemptive CPU from lower priority process
    - *Starvation* is possible
  - Time slice between the queues
    - foreground process, queue 80% CPU time for RR
    - background process queue, 20% CPU time for FCFS

Multilevel Queue Scheduling (Example)

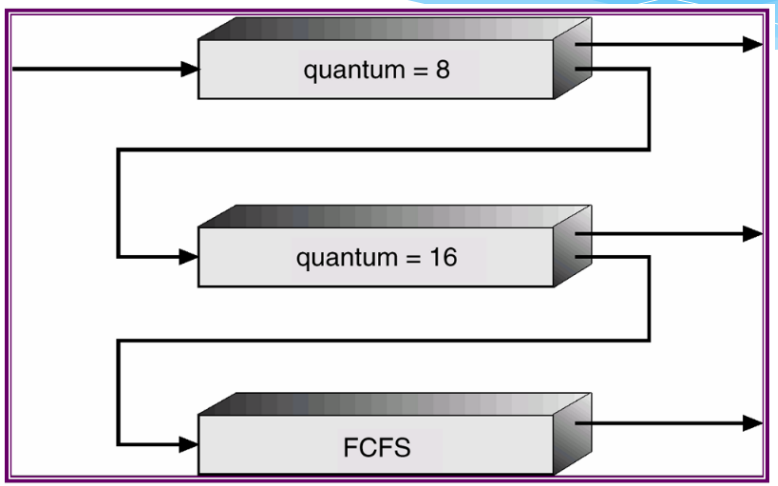


Multilevel Feedback Queue

- Allows a process to move between queues
- separate processes with different CPU-burst characteristics
  - If a process uses too much CPU time ->moved to a lower-priority queue
  - I/O-bound and interactive processes in the higher-priority queues
  - process that waits too long in a lower- priority queue may be moved to a higher-priority queue
    - Prevent "starvation"
- multilevel feedback queue scheduler is defined by the following parameters:
  - number of queues
  - scheduling algorithm for each queue
  - method used to determine when to upgrade/demote a process to a higher/lower priority queue
  - method used to determine which queue a process will enter when that process needs service

Multilevel Feedback Queue

- Example



- Basic Concepts
- Scheduling Criteria
- Scheduling algorithms
- Multi-processor scheduling

Problem

- the scheduling problem is correspondingly more complex
- Load sharing
  - separate queue for each processor
    - one processor could be idle, with an empty queue, while another processor was very busy
    - use a common ready queue
  - common ready queue's problems :

    →two processors do not choose the same process or

    →processes are lost from the queue
- asymmetric multiprocessing
- only one processor accesses the system's queue -> no sharing problem
  - I/O-bound processes may bottleneck on the one CPU that is performing all of the operations

Homework

- Write program to simulate multilevel feedback queue

① Process
② Thread
③ CPU scheduling
④ Critical resource and process synchronization
⑤ Deadlock and solutions

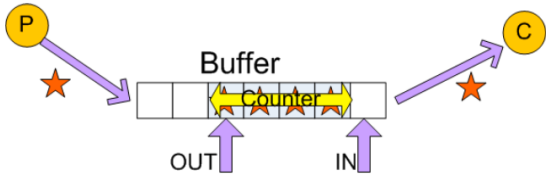
- Critical resource
- Internal lock method
- Test and Set method
- Semaphore mechanism
- Process synchronization example
- Monitor

Example

```
#include <windows.h>
#include <stdio.h>
int x = 0, y = 1;
void T1(){
      while(1){ x = y + 1; printf("%4d", x); }
}
void T2(){
      while(1){ y = 2; y = y * 2; }
}
int main(){
      HANDLE h1, h2; DWORD Id;
      h1=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)T1,NULL,0,&Id);
      h2=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)T2,NULL,0,&Id);
      WaitForSingleObject(h1,INFINITE);
      WaitForSingleObject(h2,INFINITE);
      return 0;
}
```


Results

producer-consumer I



- Two processes in the system
  - Producer creates products
  - Consumer consumes created product
- Producer and Consumer must be synchronized.
  - Do not create product when there is no space left
  - Do not consume product when there is none

Producer-Consumer Problem II

Producer

```
while(1) {
    /*produce an item in
nextProduced*/
    while (Counter == BUFFER_SIZE) ;
                /*do nothing*/
    Buffer[IN] = nextProduced;
    IN = (IN + 1) % BUFFER_SIZE;
    Counter++;
}
```

Consumer

```
while(1){
    while(Counter == 0) ;
          /*do nothing*/
    nextConsumed = Buffer[OUT];
    OUT =(OUT + 1) % BUFFER_SIZE;
    Counter--; /*consume the item in
                nextConsumed*/
}
```

Nhận xét

- Producer creates one product
- Consumer consumes one product

⇒Number of product inside Buffer does not change

Definition

Resource

Everything that is required for process's execution

Critical resource

- Resource that is limited of sharing capability
- Required concurrently by processes
- Can be either physical devices or sharing data

Problem

Sharing critical resource may not guarantee data completeness
⇒ Require process synchronization mechanism

Race condition

- Situation where the results of many processes access the sharing data depend of the order of these actions
  - Make program's result undefinable
- Prevent race condition by synchronize concurrent running processes
  - Only one process can access sharing data at a time
    - Variable counter in Producer-Consumer problem
  - The code segment that access sharing data in the process have to be executed in a defined order
    - E.g.: x←y+1 instruction in Thread T_1 only both two instruction of Thread T_2 are done

Critical section

- The part of the program where the shared memory is accessed is called the critical region or critical section
- When there are more than one process use critical resource then we have to synchronize them
  - Object: guarantee that no more than one process can stay inside critical section

Conditions to have a good solution

- Mutual Exclusion: Critical resource does not have to serve the number of process more than its capability at any time
  - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections
- Progress: If critical resource still able to serve and there are process want to be executed in critical section then this process can use critical resource
- Bounded Waiting: There exists a bound 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

Rule

- There are two processes *P1&P2* concurrently running
- Sharing the same critical resource
- Each process put the critical section at begin and the remainder section is next
  - Process must ask before enter the critical section *{entry section}*
  - Process perform *{exit section}* after exiting from critical section
- Program structure

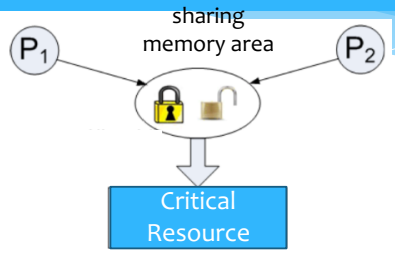
```
do {
    entry section
        critical section
    exit section
        remainder section
} while (1);
```


Methods' classification

- Low level method
  - Variable lock
  - Test and set
  - Semaphore
- High level method
  - Monitor

- Critical resource
- Variable lock method
- Test and Set method
- Semaphore mechanism
- Process synchronization example
- Monitor

Principle



- Each process use one byte in the sharing memory area as a lock
  - Process enter critical section, lock (byte lock = true)
  - Process exit from critical section, unlock (byte lock= false)
- Process want to enter critical section: check other process's lock byte 's status
  - Locking ⇒ Wait
  - Not lock ⇒ Have the right to enter critical section

Algorithm

- Share var C1,C2 Boolean // sharing variable used as lock
- Initialization C1 = C2 = false // Critical resource is free

Process P1

```
do{
    while(C2 == true);
    C1 ← true;
    { Process P1's critical section }
    C1 ← false;
    { Process P1's remaining section }
}while(1);
```

Process P2

```
do{
    while(C1 == true);
    C2 ← true;
    { Process P2's critical section }
    C2 ← false;
    { Process P2's remaining section }
}while(1);
```

Chapter 2 Process Management
4. Critical resource and process synchronization
4.2 Variable lock

Algorithm

- Share var C1,C2 Boolean // sharing variable used as lock
- Initialization C1 = C2 = false // Critical resource is free

Process P1

```
do{
    C1 ← true;
    while(C2 == true);
    { Process P1's critical section }
    C1 ← false;
    { Process P1's remaining section }
}while(1);
```

Process P2

```
do{
    C2 ← true;
    while(C1 == true);
    { Process P2's critical section }
    C2 ← false;
    { Process P2's remaining section }
}while(1);
```

Chapter 2 Process Management
4. Critical resource and process synchronization
4.2 Variable lock

Remark

- Not properly synchronize
  - Two processes request resource at the same time
    - Mutual exclusion problem (Case 1)
    - Progressive problem (Case 2)
- Reason: The following actions are done separately
  - Check the right to enter critical section
  - Set the right to enter critical section

Chapter 2 Process Management
4. Critical resource and process synchronization
4.2 Variable lock

Dekker's algorithm

- Utilize ***turn*** variable to show process with priority

Process P1

```
do{
    C1 ← true;
    while(C2==true){
        if(turn == 2){
            C1 ← false;
            while(turn ==2);
            C1 ← true;
        }
    }
    { Process P1's critical section }
    turn = 2;
    C1 ← false;
    { P1's remaining section }
}while(1);
```

Process P2

```
do{
    C2 ← true;
    while(C1==true){
        if(turn == 1){
            C2 ← false;
            while(turn ==1);
            C2 ← true;
        }
    }
    { P2's critical section }
    turn = 1;
    C2 ← false;
    { P2s remaining section }
}while(1);
```

Remark

- Synchronize properly for all cases
- No hardware support requirement -> implement in any languages
- Complex when the number of processes and resources increase
- "busy waiting" before enter critical section
  - When waiting, process has to check the right to enter the critical section => Waste processor's time

Principle

- Utilize hardware support
- Hardware provides uninterruptible instructions
- Test and change the content of aword

```
boolean TestAndSet(VAR boolean target) {
    boolean rv = target;
    target = true;
    return rv;
}
```

  - Swap the content of two different words

```
void Swap(VAR boolean , VAR boolean b) {
    boolean temp = a;
    a = b;
    b = temp;
}
```

- instruction is executed atomically
- Code block is uninterruptible when executing
  - When called at the same time, done in any order

Algorithm with TestAndSet instruction

- Sharing variable **Boolean: Lock**: resource's status:
  - Locked (*Lock=true*)
  - Free (*Lock=false*)
- Initialization: *Lock = false* ⇒ Resource is free
- Algorithm for process *Pi*

```
do{
    while(TestAndSet(Lock));
        { Process P's critical section }
    Lock = false;
        { P's remaining section }
}while(1);
```

Algorithm with Swap instruction

- Sharing variable **Lock** shows the resource's status
- Local variable for each process: **Key**: Boolean
- Initialization: *Lock = false* ⇒ Resource is free
- Algorithm for process *Pi*

```
do{
    key = true;
    while(key == true)
        swap(Lock, Key);
    { Process P's critical section }
    Lock = false;
    { P's remaining section }
}while(1);
```


Remark

- Simple, complexity is not increase when number of processes and critical resource increase
- "busy waiting" before enter critical section
- When waiting, process has to check if sharing resource is free or not => Waste processor's time
- No bounded waiting guarantee
  - The next process entering critical section is depend on the resource release time of currently resource-using process
    ⇒ Need to be solved

Semaphore

- An integer variable, initialize by resource sharing capability
  - Number of available resources (e.g. 3 printers)
  - Number of resource's unit (10 empty slots in buffer)
- Can only be changed by 2 operation P and V
- Operation P(S) (wait(S))

```
wait(S) {
    while(S ≤ 0) no-op;
    S − −;
}
```

- Operation V(S) (signal(S))

```
signal(S) {
    S + +;
}
```

- P and V are uninterruptible instructions

Semaphore usage I

- n-process critical-section problem
  - processes share a semaphore, mutex
  - initialized to 1.
  - Each process Pi is organized as

```
do {
    wait(mutex);
        critical section
    signal(mutex);
        remainder section
} while (1);
```


Semaphore usage II

- The order of execution inside processes:
  - PI with a statement S1 , P2 with a statement S2.
  - require that S2 be executed only after S1 has completed

P1 and P2 share a common semaphore synch, initialized to 0,
Code for each process

Process 1
S1; signal (synch) ; { Remainder code}

Process 2
wait (synch) ; S2; { Remainder code}

To overcome the need for busy waiting

Use 2 operations
- Block() Temporarily suspend running process
- Wakeup(P) Resume process P suspended by block() operation

- When a process executes the wait operation and semaphore value is not positive
  - it must wait. (block itself -> not busy waiting)
  - block operation places a process into a waiting queue associated with the semaphore,
  - Process's state is switched to the waiting
  - Control is transferred to the CPU scheduler,
  - process that is blocked, waiting on a semaphore S, restarted when some other process executes a signal operation.
- The process is restarted by a wakeup operation
  - changes the process from the waiting state to the ready state.
  - process is then placed in the ready queue.

Semaphore implementation

Semaphore S

```
typedef struct{
   int value;
   struct process * Ptr;
}Semaphore;
```

wait(S)/P(S)

```
void wait(Semaphore S) {
   S.value--;
   if(S.value < 0) {
        Insert process to    S.Ptr
       block();
    }
}
```

signal(S)/V(S)

```
void signal(Semaphore S) {
 S.value++;
 if(S.value ≤ 0) {
      Get process from    S.Ptr
   wakeup(P);
 }
}
```

Synchronization example

running

P_1

running

P_2

running

P_3

t

Semaphore S

S.Value = 1

S.Ptr → NULL

Chapter 2 Process Management
4. Critical resource and process synchronization
4.4. Semaphore mechanism

Synchronization example

running

P_1 $P_1 \rightarrow P(S)$

running

P_2

running

P_3

t

Semaphore S

S.Value = 0

S.Ptr → NULL

Chapter 2 Process Management
4. Critical resource and process synchronization
4.4. Semaphore mechanism

Synchronization example

running

P_1 $P_1 \rightarrow P(S)$

block

P_2 $P_2 \rightarrow P(S)$

running

P_3

t

Semaphore S

S.Value = -1

S.Ptr → PCB2

Chapter 2 Process Management
4. Critical resource and process synchronization
4.4. Semaphore mechanism

Synchronization example

running

P_1 $P_1 \rightarrow P(S)$

block

P_2 $P_2 \rightarrow P(S)$

block

P_3 $P_3 \rightarrow P(S)$

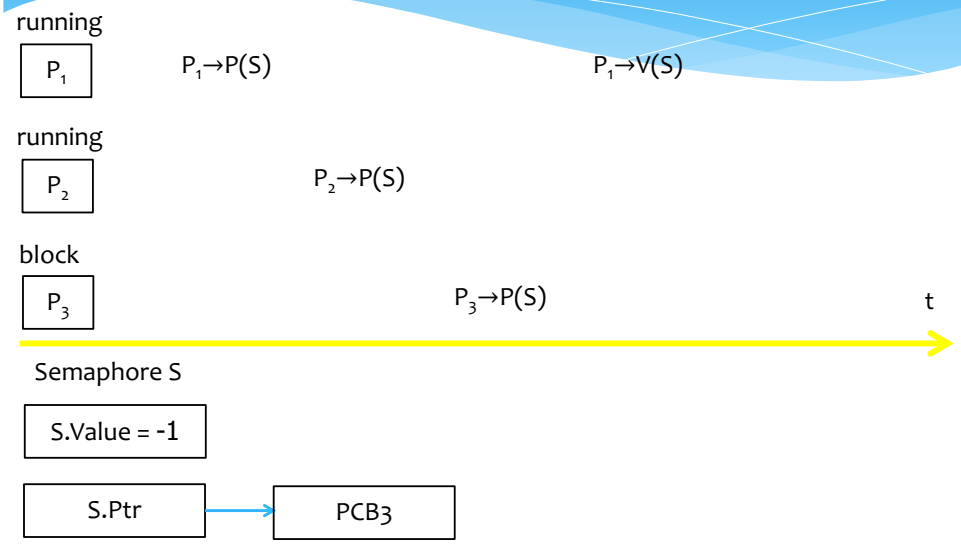
t

Semaphore S

S.Value = -2

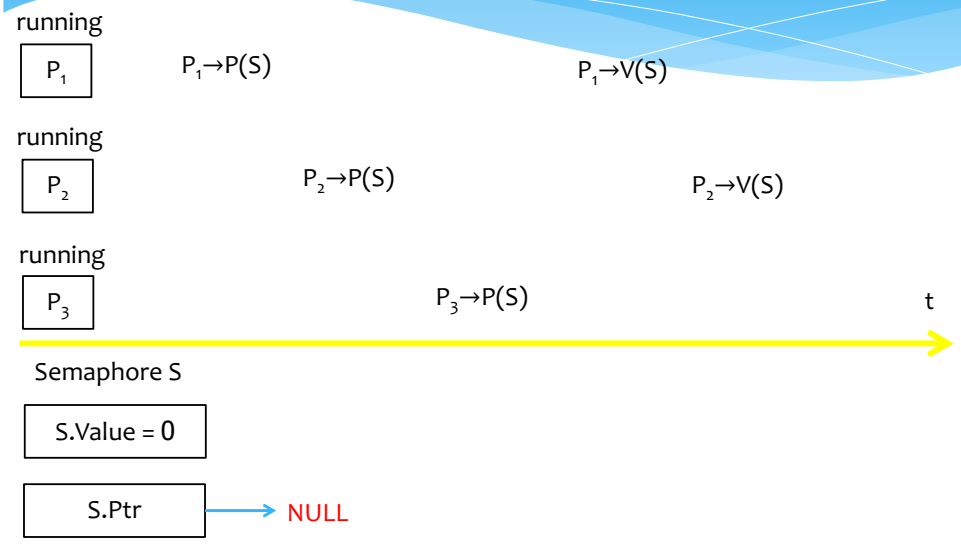
S.Ptr → PCB2 → PCB3

Synchronization example



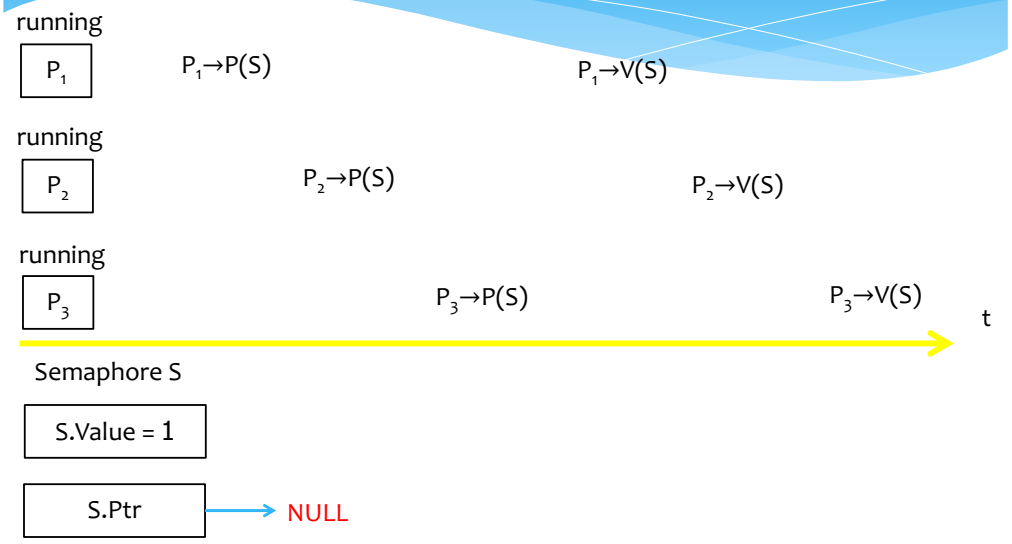
Chapter 2 Process Management
4. Critical resource and process synchronization
4.4. Semaphore mechanism

Synchronization example



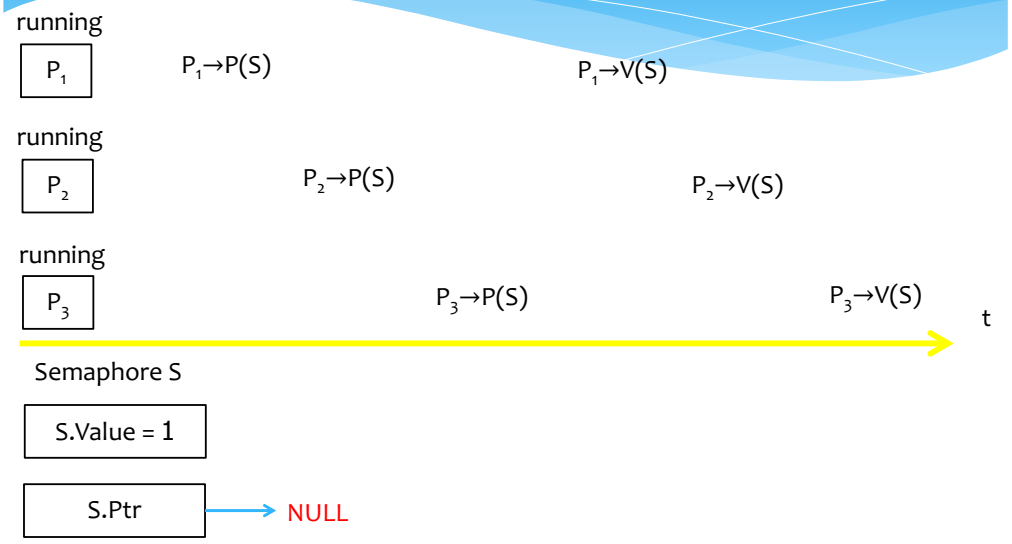
Chapter 2 Process Management
4. Critical resource and process synchronization
4.4. Semaphore mechanism

Synchronization example



Chapter 2 Process Management
4. Critical resource and process synchronization
4.4. Semaphore mechanism

Synchronization example



Remark

- Easy to apply for complex system
- No busy waiting
- The effectiveness is depend on user

P(S) {Critical section} V(S)	V(S) {Critical section} P(S)	P(S) {Critical section} P(S)
Correct synchronize	Wrong order	Wrong command

Remark

- P(S) and V(S) is nonshareable

⇒P(S) and V(S) are 2 critical resource

⇒Need synchronization

- Uniprocessor system: Forbid interrupt when perform wait(), signal()
- Multiprocessor system
  - Not possible to forbid interrupt on other processors
  - Use variable lock method ⇒ busy waiting, however waiting time is short (10 commands)

Semaphore object in WIN32 API

- CreateSemaphore(. . .) : Create a Semaphore
  - LPSECURITY_ATTRIBUTES lpSemaphoreAttributes
    ⇒ pointer to a SECURITY_ATTRIBUTES structure, handle can be inherited?
  - LONG InitialCount, ⇒ initial count for Semaphore object
  - LONG MaximumCount, ⇒ maximum count for Semaphore object
  - LPCTSTR lpName ⇒ Name of Semaphore object

  Example: CreateSemaphore(NULL,0,1,NULL);
  - Return HANDLE of Semaphore object or NULL
- WaitForSingleObject(HANDLE h, DWORD time)
- ReleaseSemaphore (. . .)
  - HANDLE hSemaphore, ⇐ handle for a Semaphore object
  - LONG lReleaseCount, ⇐ increase semaphore object's current count
  - LPLONG lpPreviousCount ⇐ pointer to a variable to receive the previous count
- Example: ReleaseSemaphore(S, 1, NULL);

Example

```
#include <windows.h>
#include <stdio.h>
int x = 0, y = 1;
HANDLE S1, S2;
void T1();
void T2();
int main(){
   HANDLE h1, h2;
   DWORD ThreadId;
   S1 = CreateSemaphore( NULL,0, 1,NULL);
   S2 = CreateSemaphore( NULL,0, 1,NULL);
   h1 = CreateThread(NULL,0,T1, NULL,0,&ThreadId);
   h2 = CreateThread(NULL,0,T2, NULL,0,&ThreadId);
   getch();
   return 0;
}
```

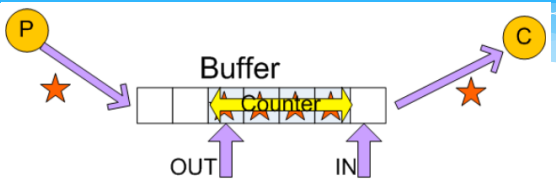
Example

```
void T1(){
    while(1){
        WaitForSingleObject(S1,INFINITE);
        x = y + 1;
        ReleaseSemaphore(S2,1,NULL);
        printf("%4d",x);
    }
}
void T2(){
    while(1){
        y = 2;
        ReleaseSemaphore(S1,1,NULL);
        WaitForSingleObject(S2,INFINITE);
        y = 2 * y;
    }
}
```


- Critical resource
- Variable lock method
- Test and Set method
- Semaphore mechanism
- Process synchronization examples
- Monitor

- Producer-Consumer problem
- Dining Philosophers problem
- Readers-Writers
- Sleeping Barber
- Bathroom Problem

Producer-consumer problem



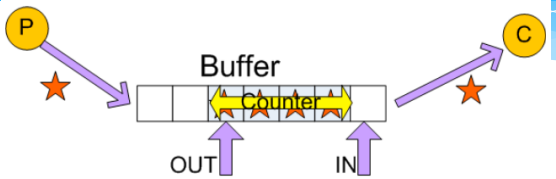

```
while(1) {
    /*produce an item in  Buffer*/
     while (Counter == BUFFER_SIZE) ;
                /*do nothing*/
    Buffer[IN] = nextProduced;
    IN = (IN + 1) % BUFFER_SIZE;
    Counter++;
}
```

Producer

```
while(1){
    while(Counter == 0) ;
          /*do nothing*/
     nextConsumed = Buffer[OUT];
     OUT =(OUT + 1) % BUFFER_SIZE;
    Counter--; /*consume the item in
                 nextConsumed*/
}
```

Consumer

Producer-Consumer problem




```
while(1) {
    /*produce an in  Buffer */
    if(Counter==SIZE) block();
                /*do nothing*/
     Buffer[IN] = nextProduced;
     IN = (IN + 1) % BUFFER_SIZE;
     Counter++;
if(Counter==1) wakeup(Consumer);
}
```

Producer

```
while(1){
      if(Counter == 0) block();
            /*do nothing*/
       nextConsumed = Buffer[OUT];
       OUT =(OUT + 1) % BUFFER_SIZE;
      Counter--;
if(Counter==SIZE-1) wakeup(Producer);
/*consume the item in   Buffer*/
}
```

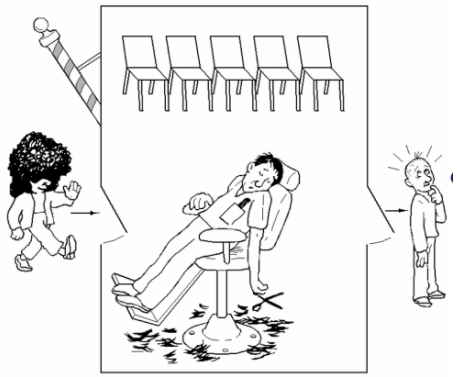
Consumer

Readers and Writers problem

- Many Readers processes access the database at the same time
- Several Writers processes update the database
- Allow unlimited Readers to access the database
  - One Reader process is accessing the database, new Reader process can access the database
  - (Writers processes has to stay in waiting queue)
- Allow only one Writer process to update the database at a time
- Non-preemptive problem. Process stays inside critical section without being interrupted

Sleeping barber

- N waiting chair for client
- Barber can cut for one client at a time
- No client, barber go to sleep
- When client comes
  - If barber is sleeping ⇒ wake him up
  - If barber is working
    - Empty chair exists ⇒ sit and wait
    - No empty chair left ⇒ Go away



Bathroom Problem

●A bathroom is to be used by both men and women, but not at the same time
●If the bathroom is empty, then anyone can enter
●If the bathroom is occupied, then only a person of the same sex as the occupant(s) may enter
●The number of people that may be in the bathroom at the same time is limited

- Problem implementation require to satisfy constraints
  - 2 types of process: male() và female()
  - Each process enter the Bathroom in a random period of time

Dining philosopher problem

Classical synchronization problem, show the situation where many processes share resources

- 5 philosophers having dinner at a round table
- In front each person is a disk of spaghetti
- Between two disk is a fork
- Philosopher do 2 things :Eat and Think
- Each person need two forks to eat
- Take only one fork at a time
- Take the left fork then the right fork
- Finish eating, return the fork to original place

Dining philosopher problem: Simple method

- Each fork is a critical resource, synchronized by a semaphore fork[i]
- Semaphore fork[5] = {1, 1, 1, 1, 1};
- Algorithm for philosopher Pi

```
do{
 wait(fork[i])
     wait(fork[(i+1)% 5]);
          {Eat}
     signal(fork[(i+1)% 5]);
 signal(fork[i]);
       {Thinks}
} while (1);
```

- If all the philosophers want to eat
  - Take the left fork (call to: wait(fork[i]))
  - Wait for the right fork (call to: wait(fork[(i+1)%5]))

⇒ deadlock

Dining philosopher problem – Solution 1

- Allow only one philosopher to take the fork at a time
- Semaphore mutex ← 1;
- Algorithm for philosopher Pi

```
do{
   wait(mutex)
       wait(fork[i])
       wait(fork[(i+1)% 5]);
   signal(mutex)
           {Eat}
       signal(fork[(i+1)% 5]);
       signal(i);
           {Thinks}
} while (1);
```

- It's possible to allow 2 non-close philosopher to eat at a time (P1: eats, P2: owns mutex⇒ P3 waits)

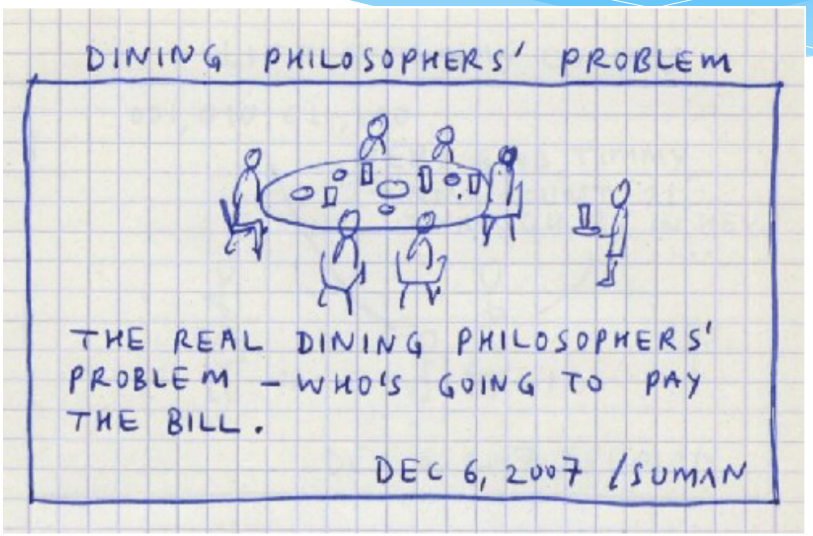
Dining philosopher problem – Solution 1

- Philosopher take the forks with different order
- Even id philosopher take the even id fork first
- Odd id philosopher take the odd id fork first

```
do{
   j = i%2
   wait(fork[(i + j)%5])
       wait(fork[(i+1 - j)% 5]);
           {Eat}
       signal(fork[(i+1 - j)% 5]);
   signal((i + i)%5);
           {Thinks}
} while (1);
```

- Solve the deadlock problem

True problem ?



- Critical resource
- Variable lock method
- Test and Set method
- Semaphore mechanism
- Process synchronization examples
- Monitor

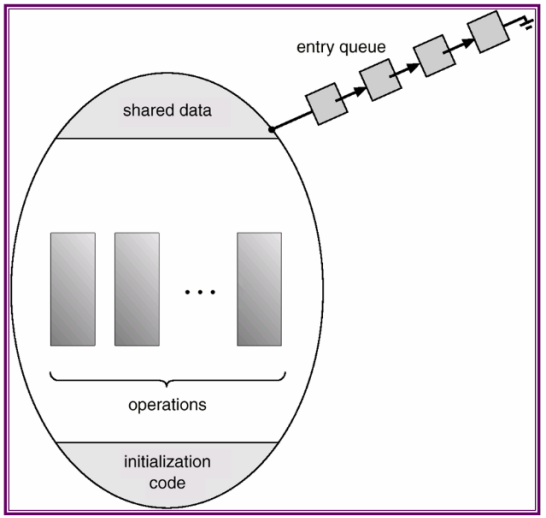
Introduction

- Special data type, proposed by HOARE 1974
- Combines of procedures, local data, initialization code
- Process can only access variables via procedures of Monitor
- Only one process can work with Monitor at a time
  - Other processes have to wait
- Allow process to wait inside Monitor
  - Utilize condition variable

```
monitor monitorName{
    Sharing variables declarations ;
    procedure P1(...){
        ...
    }
    ...
    procedure Pn(...){
        ...
    }

    {
            Initialization code
    }
} ;
```

Monitor's syntax

Model



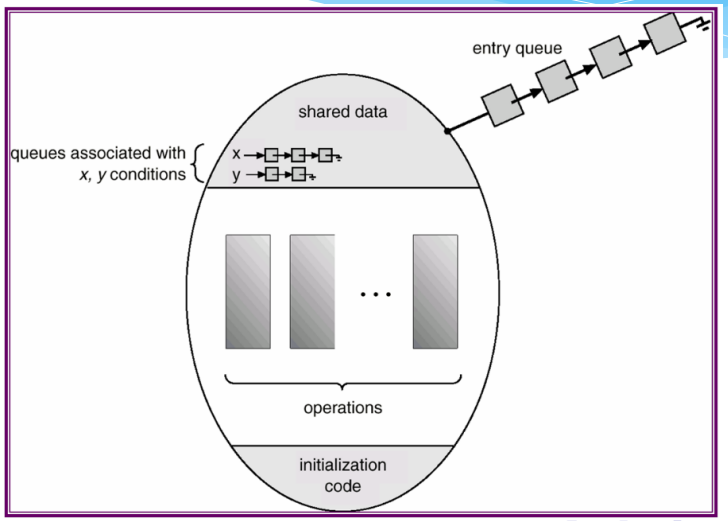
Condition Variable

Actually name of a queue
Declare: **condition x,y**;
Only used by 2 operations

wait() Called by Monitor's procedures ***(syntax x.wait() or wait(x))***. Allow process to be blocked until activated by other process via **signal()** procedure

signal() Called by Monitor's procedures **(syntax x.signal() or signal(x))**. Activate a process waiting at *x* variable queue. If no waiting process then the operation is skipped

Model



Monitor's usage: sharing a resource

```
Monitor Resource{
  Condition Nonbusy;
  Boolean Busy
//--        User's part        --
  void Acquire(){
    if(busy) Nonbusy.wait();
      busy=true;
  }
  void Release(){
    busy=false
    signal(Nonbusy)
  }
//---  Initialization part  ----
  busy= false;
  Nonbusy = Empty;
}
```

Process's structure

```
while(1){
    ...
    Resource.Acquire()
      { Using resource }
    Resource.Release()
    ...
}
```


Producer – Consumer problem

```
Monitor ProducerConsumer{
  Condition Full, Empty;
  int Counter ;
    void Put(Item){
      if(Counter=N) Full.wait();
      { Put Item into Buffer };
      Counter++;
      if(Counter=1)Empty.signal()
  }
  void Get(Item){
    if(Counter=0) Empty.wait()
        {Take Item from Buffer}
    Counter--;
    if(Counter=N-1)Full.signal()
  }
  Counter=0;
  Full, Empty = Empty;
}
```

```
ProducerConsumer M;
```

Producer

```
while(1){
    Item = New product
    M.Put(Item)
    ...
}
```

Consumer

```
while(1){
    M.Get(&Item)
     {Use Item}
    ...
}
```

① Process
② Thread
③ CPU scheduling
④ Critical resource and process synchronization
⑤ Deadlock and solutions

- Deadlock conception
- Conditions for resource deadlocks
- Solutions for deadlock
- Deadlock prevention
- Deadlock avoidance
- Deadlock detection and recovery

Deadlock conception

- System combines of concurrently running processes, sharing resources
  - Resources have different types (e.g.: CPU, memory,..).
  - Each type of resource may has many unit (e.g.: 2 CPUs, 5 printers..)
- Each process is combines of sequences of continuous operations
  - Require resource: if resource is not available (being used by other processes) ⇒ process has to wait
  - Utilize resource as required (printing, input data...)
  - Release allocated resources
- When processes share at least 2 resources, system may "in danger"

Deadlock conception

- Example: Two processes in the system P_1 & P_2
  - P_1 & P_2 share 2 resources R_1 & R_2
  - R_1 is synchronized by semaphore S_1 ($S_1 \leftarrow 1$)
  - R_2 is synchronized by semaphore S_2 ($S_2 \leftarrow 1$)
  - Code for P_1 and P_2

```
P(S1)
P(S2)
 { Use R1&R2 }
V(S1)
V(S2)
```

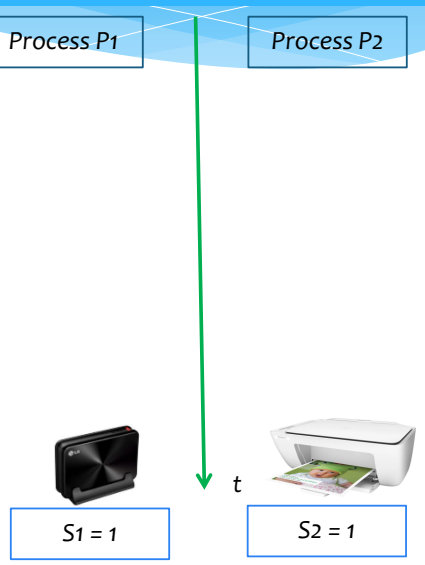
Example

Process P1

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$

Process P2

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$



Chapter 2 Process Management
5.Dead lock and solutions
5.1. Deadlock conception

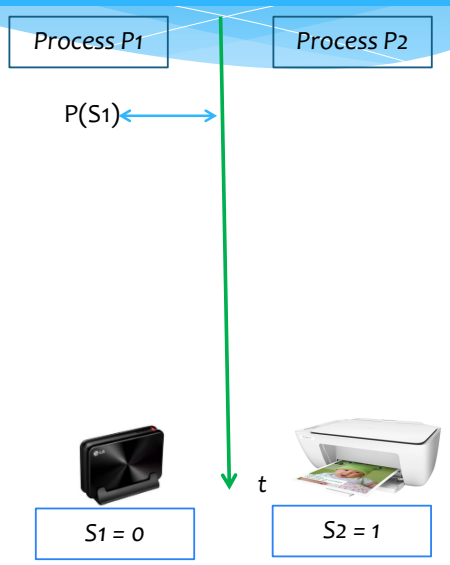
Example

Process P1

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$

Process P2

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$



Chapter 2 Process Management
5.Dead lock and solutions
5.1. Deadlock conception

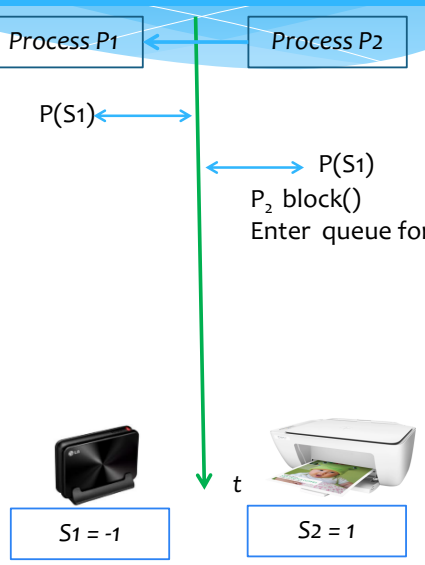
Example

Process P1

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$

Process P2

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$



Chapter 2 Process Management
5.Dead lock and solutions
5.1. Deadlock conception

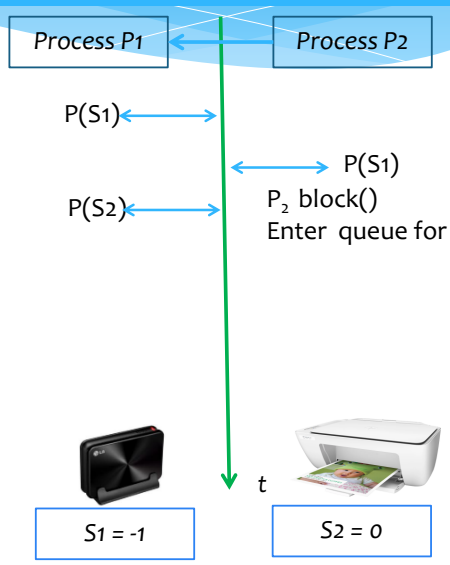
Example

Process P1

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$

Process P2

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$



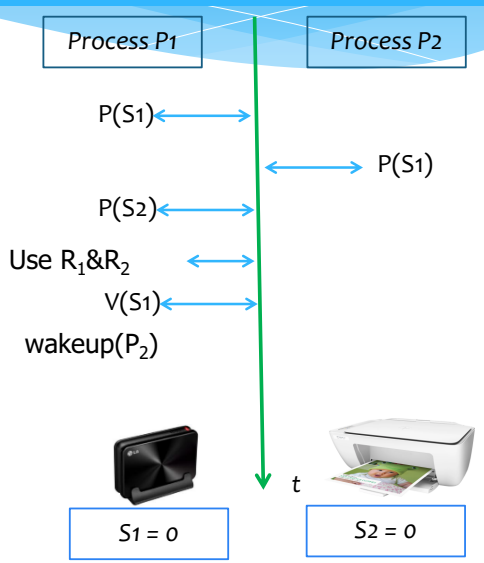
Example

Process P1

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$

Process P2

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$



Chapter 2 Process Management
5.Dead lock and solutions
5.1. Deadlock conception

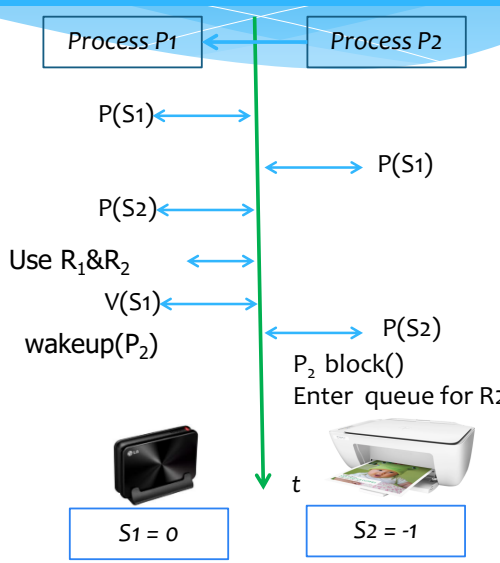
Example

Process P1

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$

Process P2

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$



Chapter 2 Process Management
5.Dead lock and solutions
5.1. Deadlock conception

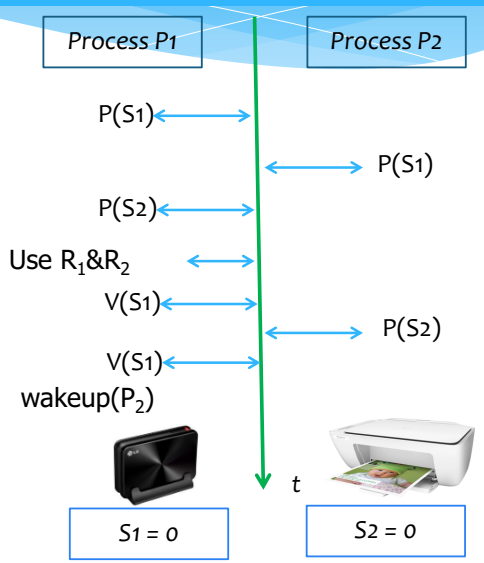
Example

Process P1

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$

Process P2

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$



Chapter 2 Process Management
5.Dead lock and solutions
5.1. Deadlock conception

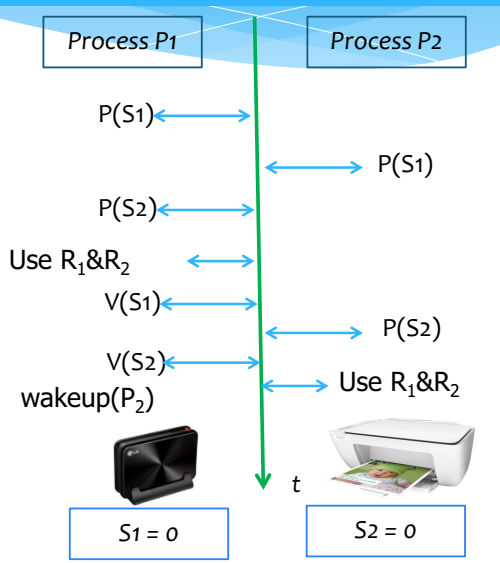
Example

Process P1

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$

Process P2

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$



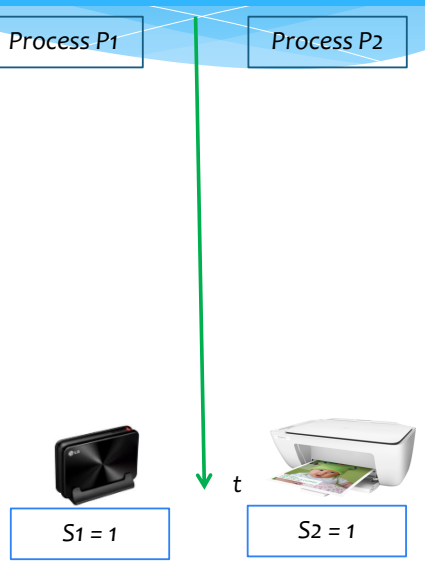
Example

Process P1

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$

Process P2

$P(S_2)$
$P(S_1)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$



Chapter 2 Process Management
5.Dead lock and solutions
5.1. Deadlock conception

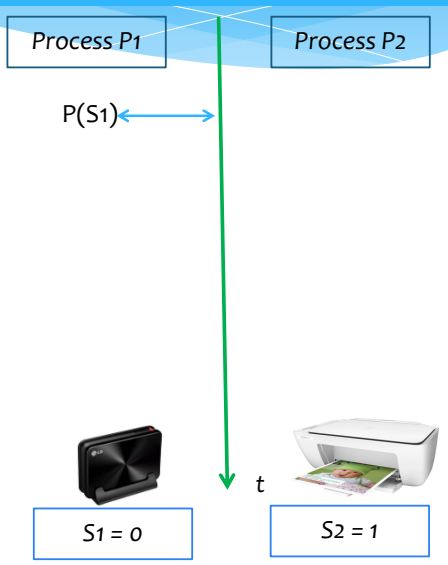
Example

Process P1

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$

Process P2

$P(S_2)$
$P(S_1)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$



Chapter 2 Process Management
5.Dead lock and solutions
5.1. Deadlock conception

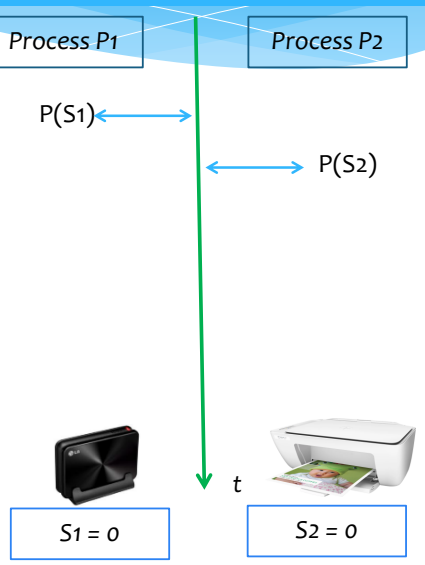
Example

Process P1

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$

Process P2

$P(S_2)$
$P(S_1)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$



Chapter 2 Process Management
5.Dead lock and solutions
5.1. Deadlock conception

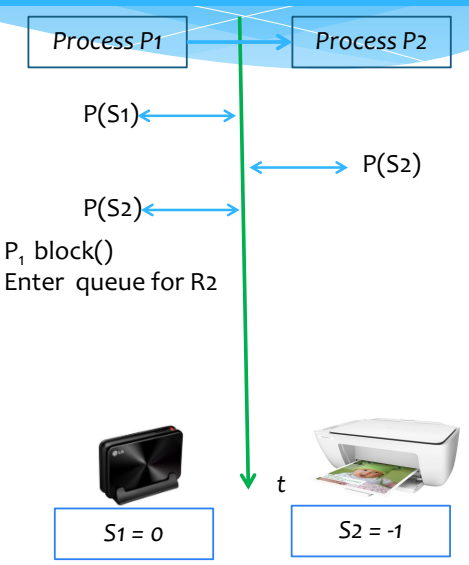
Example

Process P1

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$

Process P2

$P(S_2)$
$P(S_1)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$



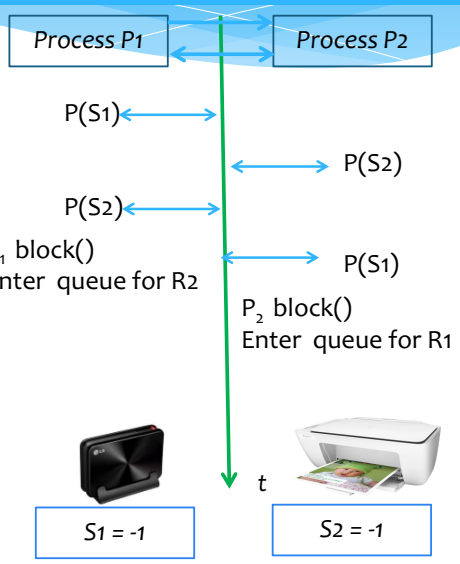
Example

Process P1

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$

Process P2

$P(S_2)$
$P(S_1)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$



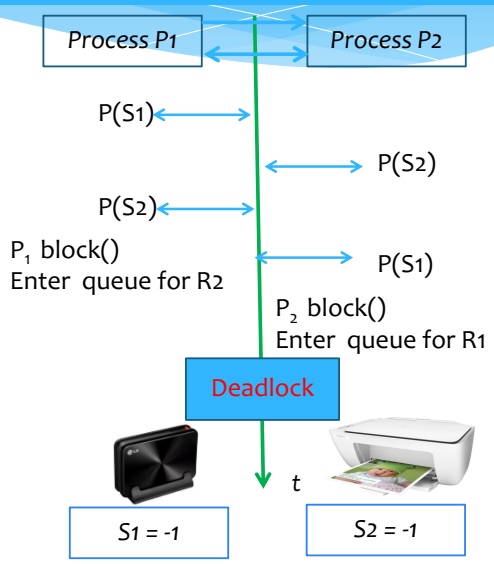
Example

Process P1

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$

Process P2

$P(S_2)$
$P(S_1)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$



Definition

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause

- Deadlock conception
- Conditions for resource deadlocks
- Solutions for deadlock
- Deadlock prevention
- Deadlock avoidance
- Deadlock detection and recovery

Conditions

4 conditions, must occur at the same time

- Critical resource
  - Resource is used in a non-shareable model
    - Only one process can use resource at a time
    - Other process request to use resource ⇒ request must be postponed until resource is released
- Wait before enter the critical section
  - Process can not enter critical section has to wait in queue
  - Still own resources while waiting
- No resource reallocation system
  - Resource is non-preemptive
  - Resource is released only by currently using process after this process finished its task
- Circular waiting
  - Set of processes $\{P_0, P_2, \ldots, P_n\}$ waiting in a order: $P_0 \rightarrow R_1 \rightarrow P_1$; $P_1 \rightarrow R_2 \rightarrow P_2$; $\ldots P_{n-1} \rightarrow R_n \rightarrow P_n$; $P_n \rightarrow R_0 \rightarrow P_0$
  - Circular waiting create nonstop loop

Example: Dining philosopher problem

- Critical resource
- Wait before enter critical section
- Non-preemptive resource
- Circular waiting

- Deadlock conception
- Conditions for resource deadlocks
- **Solutions for deadlock**
- Deadlock prevention
- Deadlock avoidance
- Deadlock detection and recovery

Methods

- Prevention
  - Apply methods to guarantee that the system never has deadlock
  - Expensive
  - Apply for system that deadlock happens frequently and once it happen the cost is high
- Avoidance
  - Check each process's resource request and reject request if this request may lead to deadlock
  - Require extra information
  - Apply for system that deadlock happens least frequently and once it happen the cost is high
- Deadlock detection and recovery
  - Allow the system work normally ⇒ deadlock may happen
  - Periodically check if deadlock is happening
  - If deadlock apply methods to remove deadlock
  - Apply for system that deadlock happens least frequently and once it happen the cost is low

- Deadlock conception
- Conditions for resource deadlocks
- Solutions for deadlock
- **Deadlock prevention**
- Deadlock avoidance
- Deadlock detection and recovery

Rule

Attack 1 of 4 required conditions for deadlock to appear

Critical resource

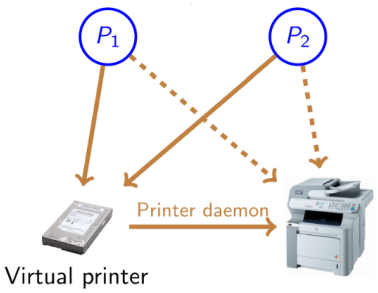
Wait before entering critical section

Non-preemptive resource

Circular wait

Critical resource condition

- Reduce the system's critical degree
  - Shareable resource(read-only ffile): accessed simultaneously
  - Non-shareable resource: Cannot be accessed simultaneously
- SPOOL mechanism(*Simultaneous peripheral operation on-line*)
  - Do not allocate resource when it's not necessary
  - A limited number of processes can request resourcen



- Only process *printer daemon* work with printer ⇒ Deadlock for resource printer is canceled
- Not any resource can be used with SPOOL

Wait before entering critical section condition

Rule: Make sure one process request resource only when it doesn't own any other resources

- Prior allocate
  - processes request all their resources before starting execution and only run when required resources are allocated
  - Effectiveness of resource utilization is low
    - Process only use resource at the last cycle?
    - Total requested resource higher than the system's capability?
- Resource release
  - Process releases all resource before apply(re-apply) new resource
    - Process execution's speed is low
    - Must guarantee that data kept in temporary release resource won't be lost

Wait before entering critical section condition - Example

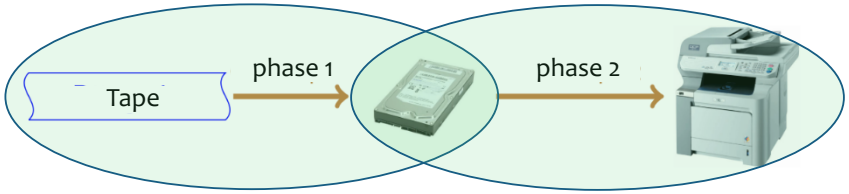
- Process combines of 2 phases
  - Copy data from tape to a file in the disk
  - Arrange the data in file and bring to printer

- Prior allocation method
  - Request both tape, file and printer
  - Waste printer in first phase, tape in the second phase

Chapter 2 Process Management
5.Dead lock and solutions
5.4. Deadlock prevention

Wait before entering critical section condition - Example

- Process combines of 2 phases
  - Copy data from tape to a file in the disk
  - Arrange the data in file and bring to printer



- Prior allocation method
  - Request both tape, file and printer
  - Waste printer in first phase, tape in the second phase
- Resource release method
  - Request tape and file for phase 1
  - Release tape and file
  - Request file and printer for phase 2

Chapter 2 Process Management
5.Dead lock and solutions
5.4. Deadlock prevention

No preemption resource condition

Rule: allow process to preempt resource when it's necessary

- Process P_i apply for resource R_j
  - R_j is available: allocate R_j to P_i
  - R_j not available: (R_j is being used by process P_k)
    - P_k is waiting for another resource
      - Preempt R_j from P_k and allocate to P_i as requested
      - Add R_j into the list of needing resource of P_k
      - P_k is execution again when
        - Receive the needing resource
        - Take back resource R_j
    - P_k is running
      - P_i has to wait (*no resource release*)
      - Allow resource preempt only when it's necessary

Chapter 2 Process Management
5.Dead lock and solutions
5.4. Deadlock prevention

No preemption resource condition

Rule: allow preemptive when it's necessary

- Only applied for resources that can be store and recover easily
  - (*CPU, memory space*)
  - Difficult to apply for resource like printer
- One process is preempted many time ?

Circular wait condition

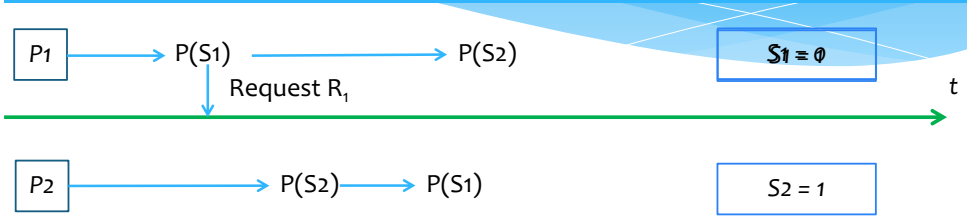
- provide a global numbering of all type of resources
  - R = {R_1, R_2, . . . R_n} Set of resources
  - Construct an ordering function f : R → N
    - Function f is constructed based on the order of resource utilization
      f(Tape) = 1
      f(Disk) = 5
      f(Printer) = 12
- Process can only request resource in an increasing order
  - Process holding resource type R_k can only request resource type R_j satisfy $f(R_j) > f(R_k)$
  - Process requests resource R_k has to release all resource R_i satisfy condition $f(R_i) \geq f(R_k)$

Circular wait condition

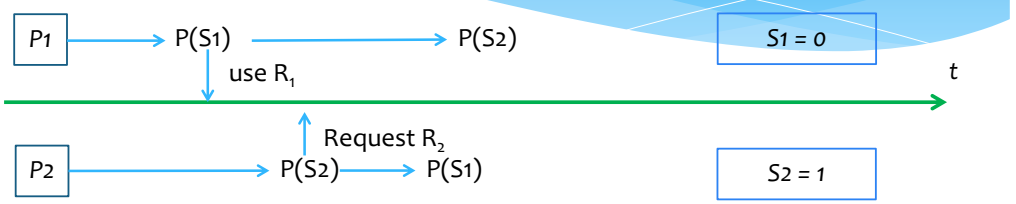
- Process can only request resource in an increasing order
  - Process holding resource type R_k can only request resource type R_j satisfy $f(R_j) > f(R_k)$
  - Process requests resource R_k has to release all resource R_i satisfy condition $f(R_i) \geq f(R_k)$
- Prove
  - Suppose deadlock happen between processes {P_1, P_2, . . . P_m}
    - $R_1 \rightarrow P_1 \rightarrow R_2 \rightarrow P_2 \Rightarrow f(R_1) < f(R_2)$
    - $R_2 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \Rightarrow f(R_2) < f(R_3)$ . . .
    - $R_m \rightarrow P_m \rightarrow R_1 \rightarrow P_1 \Rightarrow f(R_m) < f(R_1)$
  - $f(R_1) < f(R_2) < \ldots < f(R_m) < f(R_1) \Rightarrow$ invalid

- Deadlock conception
- Conditions for resource deadlocks
- Solutions for deadlock
- Deadlock prevention
- **Deadlock avoidance**
- Deadlock detection and recovery

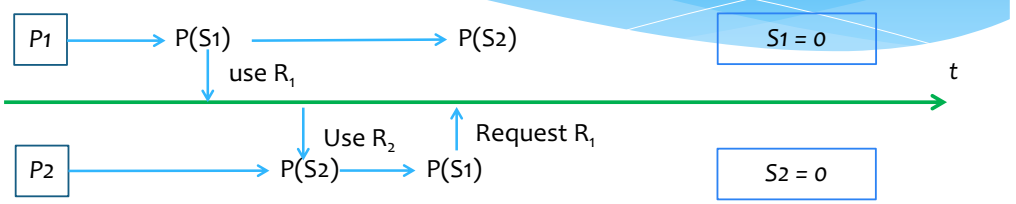
Example



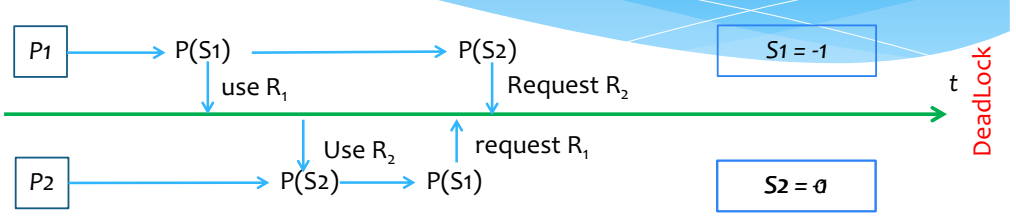
Example



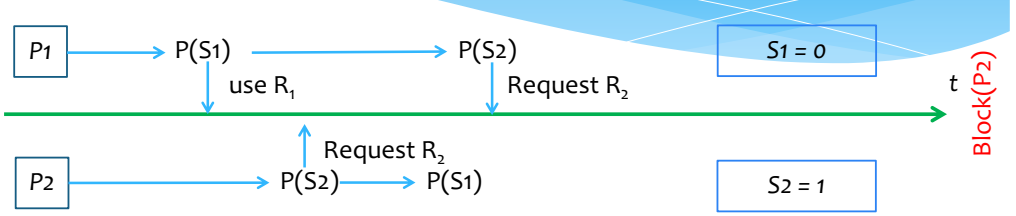
Example



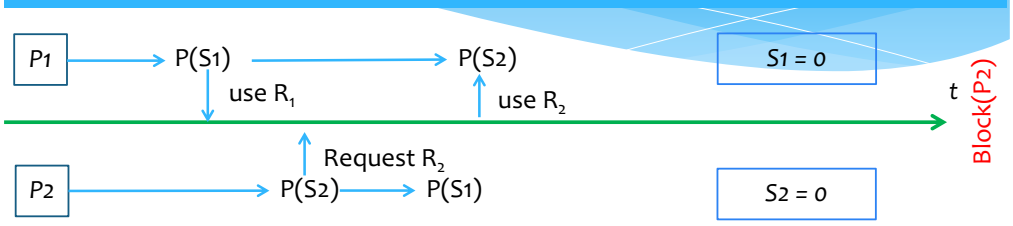
Example



Example



Example



Conclude:
If processes' orders of request/release resources are known in advance, the system could make a resource allocation decision (accept or block) for all request to let deadlock not occur.

Rule

- Must know in advance information of processes and resources
  - Process has to declare the maximum amount of each resources type that is required for execution
- Decision is made based on the result of Resource-Allocation State check
  - Resource allocation state is defined by following parameters
    - Number of resource unit available in the system
    - Number of resource unit allocated for each process
    - Number of maximum resource unit each process may request
  - If system is safe, request is accepted
- Perform checking every time a resource request is received
  - Objective: Guarantee the system's always in safe state
    - At the beginning (resource is not allocated), system is safe
    - Only allocate resource when safety is guaranteed

  ⇒ System changes from current safety state to another safety state

Safe state

The system's state is safe when

- It's possible to provide resource for each process (to maximum requirement) follow an order such that deadlock will not occur
- A safe sequence of all processes is existed

Safe sequence

Set of process P={P1, P2, . . . , Pn} is safe if

- For each process Pi, each resource request in future is accepted due to
  - The amount of available resource in the system
  - Resource is currently occupied by process Pj(j < i)

In safe sequence, when Pi request for resource

- If not accept immediately, Pi wait until Pj terminates (j < i)
- When Pj terminate and release resource, Pi receives required resource, executives, releases allocated resource and terminate
- When Pj terminate and release resource ⇒ Pi+1 will receive resource and able to finish. . .
- All the processes in the safe sequence is able to finish

Example

Consider a system includes
- 3 processes P_1, P_2, P_3 and one resource R has 12 units
- (P_1, P_2, P_3) may request maximum (10, 4, 9) unit of resource R
- At time t_0, (P_1, P_2, P_3) allocated (5, 2, 2) unit of resource R

- At current time (t_0), is the system safe?
  - System allocate (5 + 2 + 2) units => 3 units remain
  - (P_1, P_2, P_3) may request (5, 2, 7) units
  - With current 3 units, all request of P_2 is acceptable ⇒ P_2 guaranteed to finish and will release 2 unit of R after finished
  - With 3 + 2 units, P_1 guaranteed to finish and will release 5 units
  - With 3 + 2 + 5 unit P_3 guaranteed to finish
- At time t_0, P_1, P_2, P_3 are guaranteed to finish ⇒ system is safe with safe sequence (P_2, P_1, P_3)

Example

Consider a system includes
- 3 processes P_1, P_2, P_3 and one resource R has 12 units
- (P_1, P_2, P_3) may request maximum (10, 4, 9) unit of resource R
- At time t_0, (P_1, P_2, P_3) allocated (5, 2, 2) unit of resource R

- A time t_1, P_3 request and allocated 1 resource unit. Is the system safe?
  - With current 2 units, all request of P2 is acceptable ⇒ P2 guaranteed to finish and will release 2 unit of R after finished
  - When P_2 finished, the amount of available resource in the system is 4
  - With 4 resource unit, P_1 and P_3 both have to wait when apply for 5 more resource unit
  - Hence, system is not safe with (P_1, P_3)
- Remark: At time t_1, if P_3 has to wait when request for 1 more resource unit, deadlock will be removed

Example

- Conclude
  - System is safe ⇒ Processes are able to finish ⇒ no deadlock
  - System is not safe ⇒ Deadlock may occur

- Method
  - Verify all resource request
    - If the system is still safe after allocate resource ⇒ allocate
    - If not ⇒ process has to wait
  - The banker algorithm

The banker algorithm: Introduction

- Good for systems have resources with many units
- New appearing process has to declare the maximum unit of each resource type
  - Not larger than the total amount of the system
- When one process request for resource, system verify if it's safe to accept this requirement
  - If system still safe ⇒ Allocate resource for process
  - Not safe ⇒ wait

Algorithm's data I

System

n number of process in the system

m number of resource type in the system

Data structures

Available Vector with length m represents the number of available resource in the system. (Available[3] = 8 ⇒?)

Max Matrix n ∗m represents each process maximums request for each type of resource. (Max[2,3] = 5 ⇒?)

Allocation Matrix n ∗m represents amount of resource allocated for processes (Allocation[2,3] = 2 ⇒?)

Need Matrix n ∗m represents amount of resource is needed for each process Need[2,3] = 3 ⇒?)

Need[i][j] = Max[i][j] - Allocation[i][j]

Algorithm 's data I

Rule

- X, Y are vectors with length n
  - $X \le Y \Leftrightarrow X[i] \le Y[i]\ \forall i = 1, 2, \ldots, n$
- Each row of *Max,Need,Allocation* is processed like vector
- Algorithm calculated on vectors

- Local structures

Work vector with length m represents how much each resource still available

Finish vector with Boolean type, length m represents if a process is guaranteed to finish or not

Algorithm for safety checking

```
BOOL Safe(Current Resource-Allocation State){
        Work←Available
        for (i : 1 → n) Finish[i]←false
        flag← true
        While(flag){
           flag←false
           for (i : 1 → n)
                 if(Finish[i]=false AND Need[i] ≤Work){
                    Finish[i]← true
                    Work ← Work+Allocation[i]
                    flag← true
                 }//endif
        }//endwhile
        for (i : 1 → n) if (Finish[i]=false)return false
        return true;
}//End function
```


Example

- Consider system with 5 processes P_0, P_1, P_2, P_3, P_4 and 3 resources R_0, R_1, R_2
  - R_0 has 10 units, R_1 has 5 units, R_2 has 7 units
- Maximum resource requirement and allocated resource for each process

	R0	R1	R2
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

Max

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

Allocation

- Is the system safe?
- P_1 requests 1 unit of R_0 and 2 unit of R_2?
- P_4 requests 3 unit of R_0 and 3 unit of R_1?
- P_0 requests 2 unit of R_1. allocate?

Example: Check for safety

- Number of available resource in the system *Available*(R_0, R_1, R_2) =(3, 3, 2)
- Remaining request of each process (*Need = Max - Allocation*)

	R_0	R_1	R_2
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3
Max			

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1
Need			

Example: Check for safety

- Number of available resource in the system *Available*(R_0, R_1, R_2) =(3, 3, 2)

	R_0	R_1	R_2
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3
Max			

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1
Need			

Process	P_0	P_1	P_2	P_3	P_4
Finish	F	T	T	F	F
Work	([illegible])				

Example: Check for safety

- Number of available resource in the system Available(R0, R1, R2) =(3, 3, 2)

	R_0	R_1	R_2
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3
Max			

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1
Need			

Process	P_0	P_1	P_2	P_3	P_4
Finish	T	T	T	T	T
Work	([illegible])				

System is safe
(P1, P3, P4, P0, P2)

Algorithm for resource request

- Request[i] Resource requesting vector of process *Pi*
  - Request[3,2] = 2: P_3 requests 2 units of resource R_2
- When Pi make a resource request, the system checks

① if(Request[i]>Need[i])
  Error(Request higher than declared number)

② if(Request[i]>Available)
  Block(Not enough resource, process has to wait)

③ Set the new resource allocation for the system
  - Available = Available - Request[i]
  - Allocation[i] = Allocation[i] + Request[i]
  - Need[i] = Need[i] - Request[i]

④ Allocate resource based on the result of new system safety check
  if(Safe(New Resource Allocation State))
    Allocate resource for Pi as requested
  else
    Pi has to wait
    Recover former state (Available, Allocation,Need)

Example: P_1 request (1, 0, 2)

- Request[1]≤Available ((1, 0, 2) ≤ (3, 3, 2)) ⇒ It's possible to allocate
- If allocate resource: Available = (2 , 3, 0)

	R0	R1	R2
P0	0	1	0
P1	3	0	2
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	7	4	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1
Need			

Process	P_0	P_1	P_2	P_3	P_4
Finish	F	F	F	F	F
Work	(2,3,0)				

Chapter 2 Process Management
5.Dead lock and solutions
5.5. Deadlock avoidance

Example: P_1 request (1, 0, 2)

- Request[1]≤Available ((1, 0, 2) ≤ (3, 3, 2)) ⇒ It's possible to allocate
- If allocate resource: Available = (2 , 3, 0)

	R0	R1	R2
P0	0	1	0
P1	3	0	2
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	7	4	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1
Need			

Process	P_0	P_1	P_2	P_3	P_4
Finish	F	F	F	F	F
Work	(2,3,0)				

Chapter 2 Process Management
5.Dead lock and solutions
5.5. Deadlock avoidance

Example: P_1 request (1, 0, 2)

- Request[1]≤Available ((1, 0, 2) ≤ (3, 3, 2)) ⇒ It's possible to allocate
- If allocate resource: Available = (2 , 3, 0)

	R0	R1	R2
P0	0	1	0
P1	3	0	2
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	7	4	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1
Need			

Process	P_0	P_1	P_2	P_3	P_4
Finish	F	T	F	F	F
Work	(5,3,2)				

Chapter 2 Process Management
5.Dead lock and solutions
5.5. Deadlock avoidance

Example: P_1 request (1, 0, 2)

- Request[1]≤Available ((1, 0, 2) ≤ (3, 3, 2)) ⇒ It's possible to allocate
- If allocate resource: Available = (2 , 3, 0)

	R0	R1	R2
P0	0	1	0
P1	3	0	2
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	7	4	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1
Need			

Process	P_0	P_1	P_2	P_3	P_4
Finish	F	T	F	F	F
Work	(5,3,2)				

Example: P_1 request (1, 0, 2)

- Request[1]≤Available ((1, 0, 2) ≤ (3, 3, 2)) ⇒ It's possible to allocate
- If allocate resource: Available = (2 , 3, 0)

	R0	R1	R2
P0	0	1	0
P1	3	0	2
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	7	4	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1
Need			

Process	P_0	P_1	P_2	P_3	P_4
Finish	F	T	F	T	F
Work	(7,4,3)				

Chapter 2 Process Management
5.Dead lock and solutions
5.5. Deadlock avoidance

Example: P_1 request (1, 0, 2)

- Request[1]≤Available ((1, 0, 2) ≤ (3, 3, 2)) ⇒ It's possible to allocate
- If allocate resource: Available = (2 , 3, 0)

	R0	R1	R2
P0	0	1	0
P1	3	0	2
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	7	4	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1
Need			

Process	P_0	P_1	P_2	P_3	P_4
Finish	F	T	F	T	T
Work	(7,4,5)				

Chapter 2 Process Management
5.Dead lock and solutions
5.5. Deadlock avoidance

Example: P_1 request (1, 0, 2)

- Request[1]≤Available ((1, 0, 2) ≤ (3, 3, 2)) ⇒ It's possible to allocate
- If allocate resource: Available = (2 , 3, 0)

	R0	R1	R2
P0	0	1	0
P1	3	0	2
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	7	4	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1
Need			

Process	P_0	P_1	P_2	P_3	P_4
Finish	T	T	F	T	T
Work	(7,5,5)				

Chapter 2 Process Management
5.Dead lock and solutions
5.5. Deadlock avoidance

Example: P_1 request (1, 0, 2)

- Request[1]≤Available ((1, 0, 2) ≤ (3, 3, 2)) ⇒ It's possible to allocate
- If allocate resource: Available = (2 , 3, 0)

	R0	R1	R2
P0	0	1	0
P1	3	0	2
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	7	4	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1
Need			

Process	P_0	P_1	P_2	P_3	P_4
Finish	T	T	T	T	T
Work	(10,5,7)				

Request is accepted

Example: (continue)

- P_4 request 3 units of R_0 and 3 units of R_2
  - Request[4] = (3, 0, 3)
  - Available = (2, 3, 0)

  ⇒ Resource is not enough, P_4 has to wait

- P_0 request 2 units of R_1
  - Request[0]≤Available ((0, 2, 0) ≤ (2, 3, 0)) ⇒ It's possible to allocate
  - If allocate: Available = (2 , 1, 0)
  - Perform Safe algorithm

  ⇒ All processes may not finish

  ⇒ if accepted, system may be unsafe

⇒ Resource is sufficient but do not allocate, P_0 has to wait

- Deadlock conception
- Conditions for resource deadlocks
- Solutions for deadlock
- Deadlock prevention
- Deadlock avoidance
- **Deadlock detection and recovery**

Introduction

- Rule
  - Do not apply deadlock prevention or avoidance method, allow deadlock to occur
  - Timely check if the system has deadlock or not if yes then find a solution
  - To function properly, system has to provide
    - Algorithm to check if the system is deadlock or not
    - Algorithm to recorver from deadlock
- Deadlock detection
  - Algorithm for showing the deadlock
- Deadlock recovery
  - Terminate process
  - Resource preemptive

Algorithm to point out deadlock: Introduction

- Apply for system that has resource types with many units
- Similar to banker algorithms
- Data structures
  - Available Vector with length m: Available resource in the system
  - Allocation Matrix n ∗ m: Resources allocated to processes
  - Request Matrix n ∗ m: Resources requested by processes
- Local structures
  - Work Vector with length m: available resource
  - Finish Vector with length n: process is able to finish or not
- Rule
  - ≤ relations between Vectors
  - Rows in matrix n ∗ m are processed similar to vectors

Algorithm to point out deadlock

```
BOOL Deadlock(Current Resource-Allocation State){
    Work←Available
    for (i : 1 → n)
    if(Allocation[i]≠0) Finish[i]←false
    else Finish[i]←true;            //Allocation = 0 not in waiting circular
   flag← true
   While(flag){
      flag←false
      for (i : 1 → n)
           if (Finish[i] = false AND Request[i] ≤ Work){
              Finish[i]← true
             Work ← Work+Allocation[i]
            flag← true
           }//endif
    }//endwhile
   for (i : 1 → n) if (Finish[i] = false) return true;
   return false;          //Finish[i] = false, process Pi is in deadlock
}//End function
```


Example

- 5 processes P_0, P_1, P_2, P_3, P_4; 3 resources R_0, R_1, R_2
  - Resource R_0 has 7 units, R_1 has 2 units, R_2 has 6 units
- Resource allocation status at time t_0

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	3
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	0	0	0
P1	2	0	2
P2	0	0	0
P3	1	0	0
P4	6	0	2
Request			

- Available resource (R_0, R_1, R_2) =(0, 0, 0)

Example

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	3
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	0	0	0
P1	2	0	2
P2	0	0	0
P3	1	0	0
P4	6	0	2
Request			

- Available resource (R_0, R_1, R_2) =(0, 0, 0)

Process	P_0	P_1	P_2	P_3	P_4
Finish	F	F	F	F	F
Work	(0,0,0)				

Example

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	3
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	0	0	0
P1	2	0	2
P2	0	0	0
P3	1	0	0
P4	6	0	2
Request			

- Available resource (R_0, R_1, R_2) =(0, 0, 0)

Process	P_0	P_1	P_2	P_3	P_4
Finish	T	F	F	F	F
Work	(0,1,0)				

Example

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	3
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	0	0	0
P1	2	0	2
P2	0	0	0
P3	1	0	0
P4	6	0	2
Request			

- Available resource $(R_0, R_1, R_2) = (0, 0, 0)$

Process	P_0	P_1	P_2	P_3	P_4
Finish	T	F	T	F	F
Work	(3,1,3)				

Example

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	0	0	0
P1	2	0	2
P2	0	0	0
P3	1	0	0
P4	6	0	2
Request			

- Available resource $(R_0, R_1, R_2) = (0, 0, 0)$

Process	P_0	P_1	P_2	P_3	P_4
Finish	T	F	T	T	F
Work	(5,2,4)				

Example

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	0	0	0
P1	2	0	2
P2	0	0	0
P3	1	0	0
P4	6	0	2
Request			

- Available resource $(R_0, R_1, R_2) = (0, 0, 0)$

Process	P_0	P_1	P_2	P_3	P_4
Finish	T	F	T	T	F
Work	(5,2,4)				

Example

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	0	0	0
P1	2	0	2
P2	0	0	0
P3	1	0	0
P4	6	0	2
Request			

- Available resource $(R_0, R_1, R_2) = (0, 0, 0)$

Process	P_0	P_1	P_2	P_3	P_4
Finish	T	T	T	T	F
Work	(7,2,4)				

Example

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	0	0	0
P1	2	0	2
P2	0	0	0
P3	1	0	0
P4	6	0	2
Request			

- Available resource $(R_0, R_1, R_2) = (0, 0, 0)$

Process	P_0	P_1	P_2	P_3	P_4
Finish	T	T	T	T	T
Work	(7,2,6)				

System has no deadlock
$(P_0, P_2, P_3, P_1, P_4)$

Example

- At time t_1: P_2 request 1 more resource unit of R_2
- Resource allocation status

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	3
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	0	0	0
P1	2	0	2
P2	0	0	1
P3	1	0	0
P4	6	0	2
Request			

- Available resource $(R_0, R_1, R_2) = (0, 0, 0)$

Example

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	3
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	0	0	0
P1	2	0	2
P2	0	0	1
P3	1	0	0
P4	6	0	2
Request			

Process	P_0	P_1	P_2	P_3	P_4
Finish	F	F	F	F	F
Work	(0,0,0)				

Example

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	3
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	0	0	0
P1	2	0	2
P2	0	0	1
P3	1	0	0
P4	6	0	2
Request			

Process	P_0	P_1	P_2	P_3	P_4
Finish	T	F	F	F	F
Work	(0,1,0)				

Example

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	0	0	0
P1	2	0	2
P2	0	0	1
P3	1	0	0
P4	6	0	2
Request			

Process	P_0	P_1	P_2	P_3	P_4
Finish	T	F	F	F	F
Work	(0,1,0)				

P_0 may finish but the system is deadlock.
Processes are waiting for each other(P_1, P_2, P_3, P_4)

Deadlock recorvery: Process termination method

Rule: Terminate processes in deadlock and take back allocated resource

- Terminate all processes
  - Quick to eliminate deadlock
  - Too expensive
    - Killed processes may be almost finished
- Terminate processes consequently until deadlock is removed
  - After process is terminated, check if deadlock is still exist or not
    - Deadlock checking algorithm complexity is $m * n^2$
  - Need to point out the order of process to be terminated
    - Process's priority
    - Process's turn around time, how long until process finish
    - Resources that process is holding, need to finish
    - . . .
- Process termination's problem
  - Process is updating file ⇒ File is not complete
  - Process is using printer ⇒ Reset printer's status

Deadlock recorvery: Resource preemption method

Preempt continuously several resources from a deadlocked processes and give to other processes until deadlock is removed

Need to consider:

① Victim's selection
- Which resource and which process is selectedd?
- Preemptation 's order for smallest cost
- Amount of holding resource, usage time. . .

② Rollback
- Rollback to a safe state before and restart
- Require to store state of running process

③ Starvation
- One process is preempted many times ⇒ infinite waiting
- Solution: record the number of times that process is preempted

Another solution ?

Hệ Điều Hành

(*Nguyên lý các hệ điều hành*)

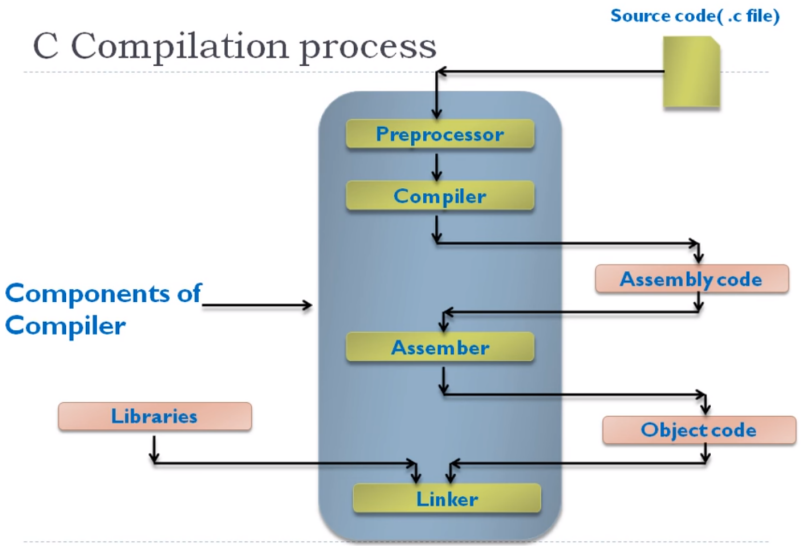
Đỗ Quốc Huy
huydq@soict.hust.edu.vn
Bộ môn Khoa Học Máy Tính
Viện Công Nghệ Thông Tin và Truyền Thông

Chap 3 Memory Management

1. Introduction
2. Memory management strategies
3. Virtual memory
4. Memory management in Intel's processors family

- Example
- Memory and program
- Address binding
- Program's structures

C compilation process



C compilation process

Preprocessor

```
/* this is demo */
#include<stdio.h>
void main()
{
printf ("hello");
}
```

Preprocessor

```
code of stdio.h file
void main()
{
printf ("hello");
}
```

Preprocessor remove comments and include header files in source code, replace macro name with code.

Chapter 3 Memory management
1. Introduction
1.1 Example

C compilation process

```
code of stdio.h file
void main()
{
printf ("hello");
}
```

Compiler

```
push    ebp
mov     ebp, esp
and     esp, -16
sub     esp, 16
mov     eax, OFFSET
FLAT:.LC0
mov     DWORD PTR
[esp], eax
call    printf
leave
ret
```

Chapter 3 Memory management
1. Introduction
1.1 Example

C compilation process

```
push    ebp
mov     ebp, esp
and     esp, -16
sub     esp, 16
mov     eax, OFFSET
FLAT:.LC0
mov     DWORD PTR
[esp], eax
call    printf
leave
ret
```

Assembler

```
00101001100101001
10101111010100101
01010010101011010
01010101010010101
01001100111111100
10001010010101001
01011111011111111
11111010000000011
10010010
```

Assebler convert assemble code into object code.

Chapter 3 Memory management
1. Introduction
1.1 Example

C compilation process

```
00101001
00101001
01011101
01001010
01001010
0110
```

```
00101001
00101001
01011101
01001010
01001010
0110
```

```
00101001
00101001
01011101
01001010
01001010
0110
```

Libraries

Linker

exe

Example: Generate programs from multi moduls

Toto project

file **main.c**

```
#include <stdio.h>
extern int x, y;
extern void toto();
int main(int argc, char *argv[]){
     toto();
     printf("KQ: %d \n",x * y);
     return 0;
}
```

file **M1.c**

```
int y = 10;
```

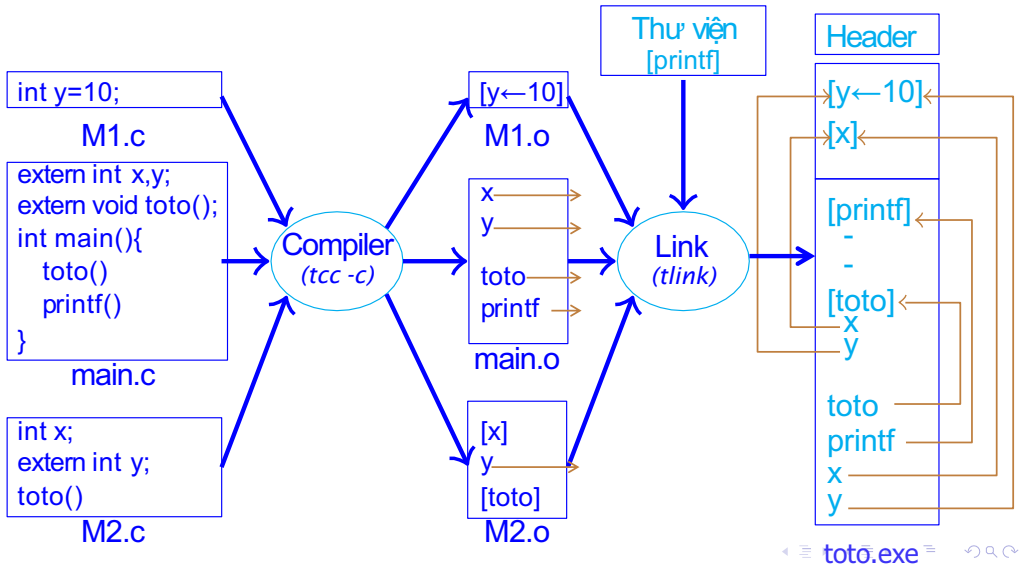
file **M2.c**

```
int x;
extern int y;
void toto(){
   x = 10 * y;
}
```

Result

```
KQ: 1000
```


toto project compilation process



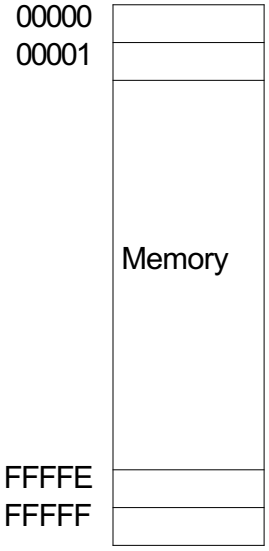
- Example
- Memory and program
- Address binding
- Program's structures

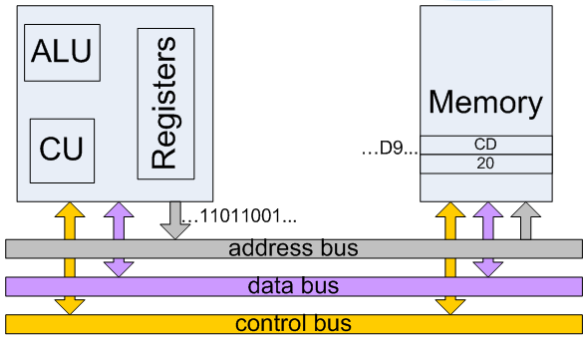
Memory leveling

- Memory is an important system's resource
  - Program must be kept inside memory to run
- Memory is characterized by size and access speed
- Memory leveling by access speed

Memory type	Size	Speed
Registers	bytes	CPU speed (ηs) 10
Cache on processor	Kilo Bytes	nano seconds
Cache level 2	KiloByte-MegaByte	100 nanoseconds e
Main memory	MegaByte-GigaByte	Micro-seconds
Secondary storage(Disk)	GigaByte-Terabytes	Mili-Seconds 10
Tape, optical disk	Unlimit	Seconds

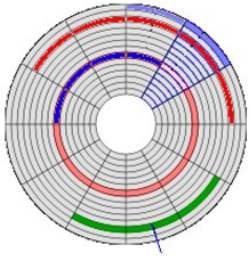
Main memory



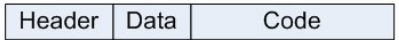


- Used for running program and data
- Array of memory block with size of bytes, words
- Each memory block has an address
  - **Physical address**

Program



File



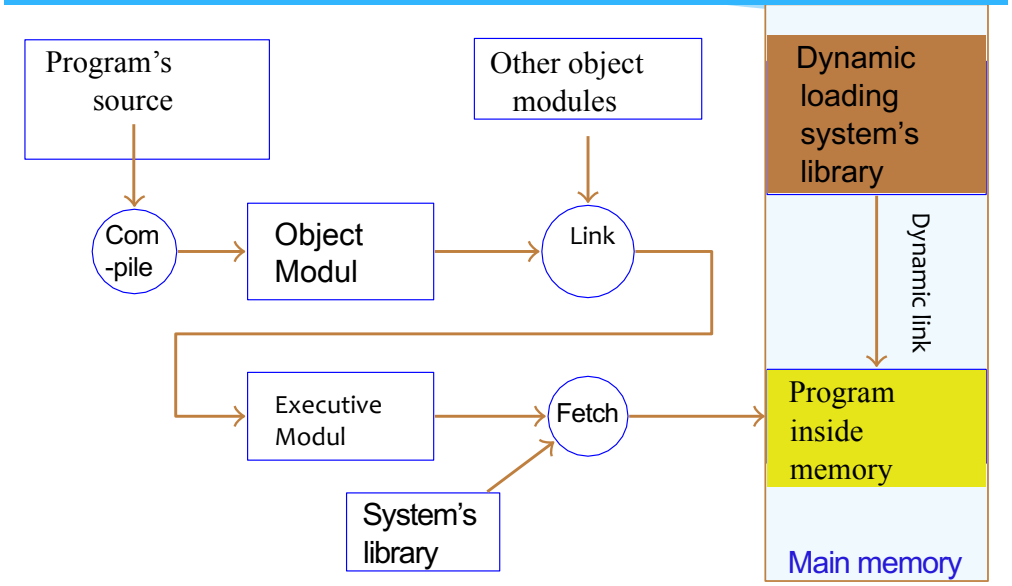
- Store on external storage devices
- Executable binary files
  - File's parameters
  - Machine instruction (binary code),
  - Data area (global variable), . .
- Must be brought into internal memory and put inside a process to be executed (process executes program)
- Input queue
  - Set of processes kept in external memory (*normally: disk*)
  - Wait to be brought into internal memory and execute

Execute a program

- Load the program into main memory
  - Read and analysis executable file (e.g. *.com, file *.exe)
  - Ask for a memory area to load program from disk
  - Set values for parameters, registers to a proper value
- Execute the program
  - CPU reads instructions in memory at location determined by program counter
    - 2 registers CS:IP for Intel's family processor (e.g. : 80x86)
  - CPU decode the instruction
    - May read more operand from memory
  - Execute the instruction with operand
  - If necessary, store the results into memory at a defined location
- Finish executing
  - Free the memory area that allocated to program
- Problem
  - Program may be loaded into any location in the memory
  - When program is executed, a sequence of addresses are generated
- How to access memory?

- Example
- Memory and program
- Address binding
- Program's structures

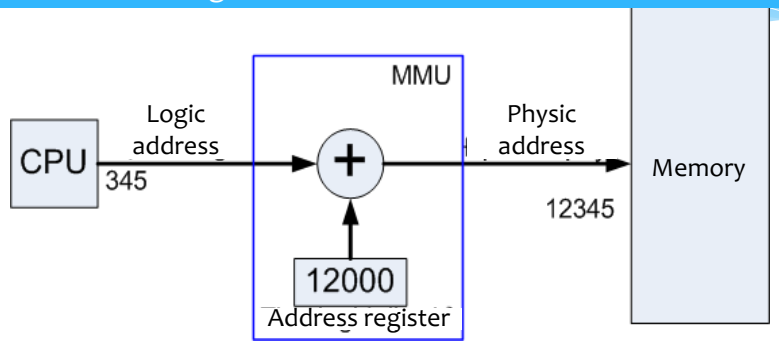
Application program processing steps



Types of address

- Symbolic
  - Name of object in the source program
  - Example: counter, x, y,...
- Relative address
  - Generated from symbolic address by compiler
  - Relative position of an object from the module's first position
    - Example: Byte number 10 from the begin of module
- Absolute address
  - Generate from relative address when program is loaded into memory
    - For IBM PC: relative address <Seg :Ofs>→ Seg * 16+Ofs
  - Object's address in physical memory – physical address
  - Example: JMP 010Ah ⇒ jump to the memory block at 010Ah at the same code segment (CS)
    - if CS=1555h, jump to location: 1555h*10h+010Ah =1560Ah

Physical address-logic address



- Logic address
  - Generate from process, (CPU brings out)
  - Converted to physical address when access to object in the program by the Memory management unit (MMU)
- Physical address
  - Address of an element (byte/word) in main memory
  - Correspond to the logic address bring out by CPU
- Program work with logical address

- Example
- Memory and program
- Address binding
- Program's structures

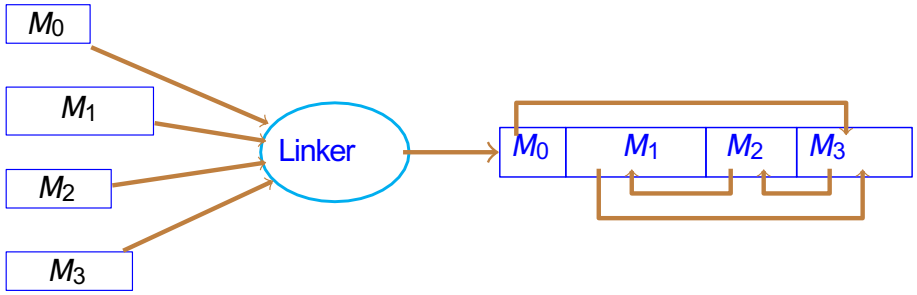
① Linear structure

② Dynamic loading structure

③ Dynamic link structure

④ Overlays structure

Linear structure

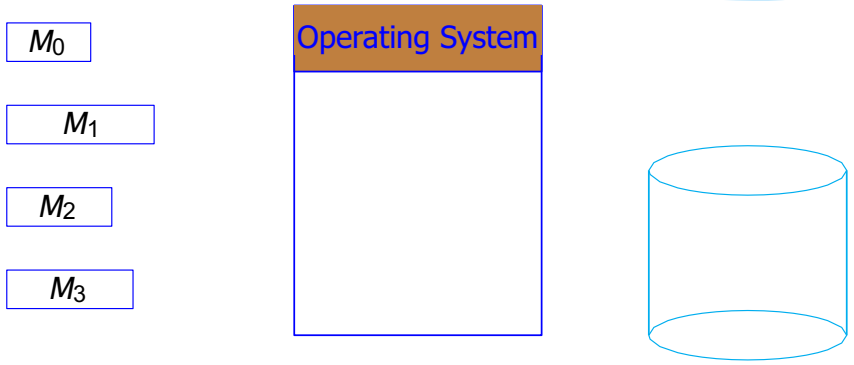


- After linking, modules are merged into a complete program
  - Contain sufficient information to be able to execute
  - External pointers are replaced by defined values
  - To execute, required only one time to fetch into the memory

Linear structure

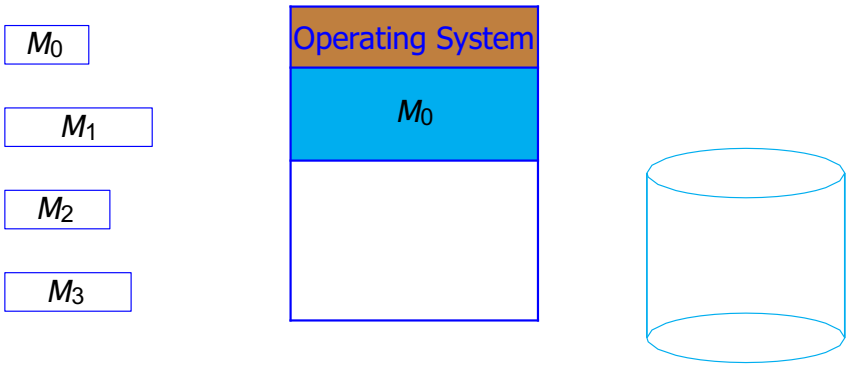
- Advantages
  - Simple, easy to link and localizing the program
  - Fast to execute
  - Highly movable
- Disadvantages
  - Waste of memory
    - Not all parts of the program are necessary for the program's execution
  - It's not possible to run the program that larger than physical memory's size

Dynamic loading structure



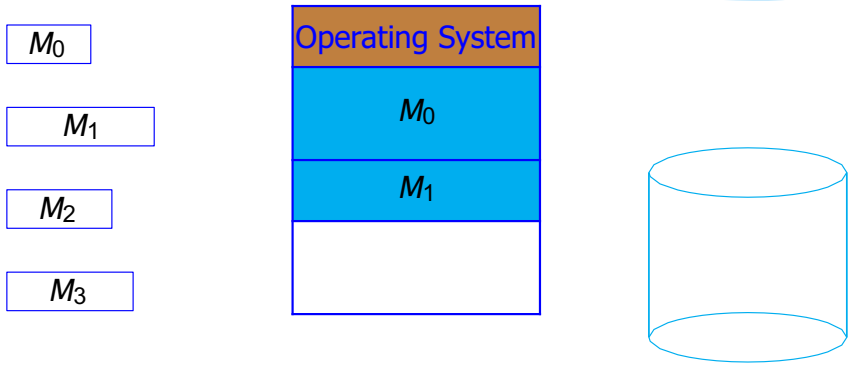
- Each module is edited separately

Dynamic loading structure



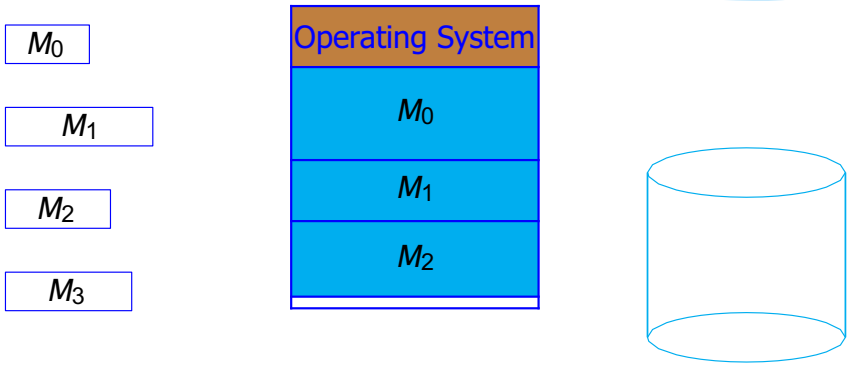
- Each module is edited separately
- When executing, system will load and localize the main module

Dynamic loading structure



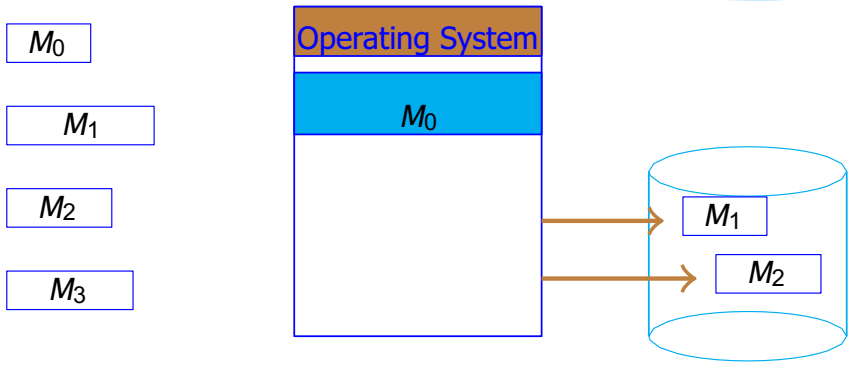
- Each module is edited separately
- When executing, system will load and localize the main module
- When module is needed, request for memory and load module into memory

Dynamic loading structure



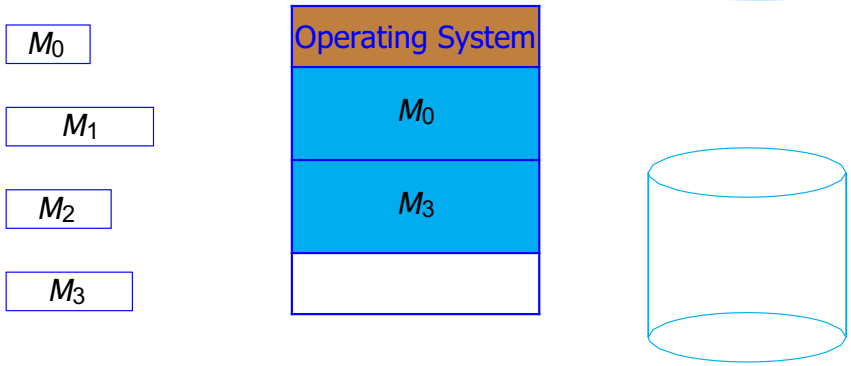
- Each module is edited separately
- When executing, system will load and localize the main module
- When module is needed, request for memory and load module into memory

Dynamic loading structure



- Each module is edited separately
- When executing, system will load and localize the main module
- When module is needed, request for memory and load module into memory
- When a module is finished using or not enough memory, bring unnecessary modules out

Dynamic loading structure



- Each module is edited separately
- When executing, system will load and localize the main module
- When module is needed, request for memory and load module into memory
- When a module is finished using or not enough memory, bring unnecessary modules out

Dynamic loading structure

Advantage

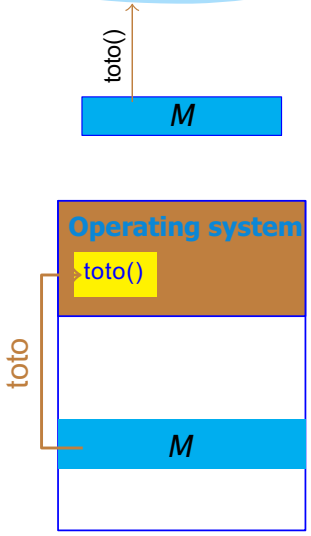
- Can use memory are smaller than the program's size
- High memory usage effectiveness if program is managed well

Disadvantage

- Slow when execution
- Mistake may cause waste of memory and increase execution time
- Require user to load and remove modules
  - User must understand clearly about the system
  - Reduce the program's flexible

Dynamic-link structure

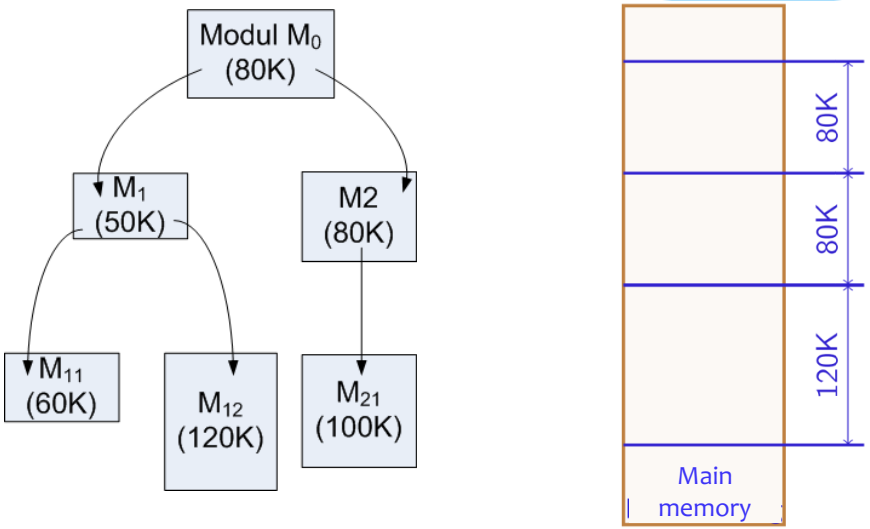
- Links will be postponed when program is executing
- Part of the code segment (stub) is utilized to search for corresponding function in the library in the memory
- When found, stub will be replace by the address of the function and function will be executed
- Useful for constructing library



Overlays structure

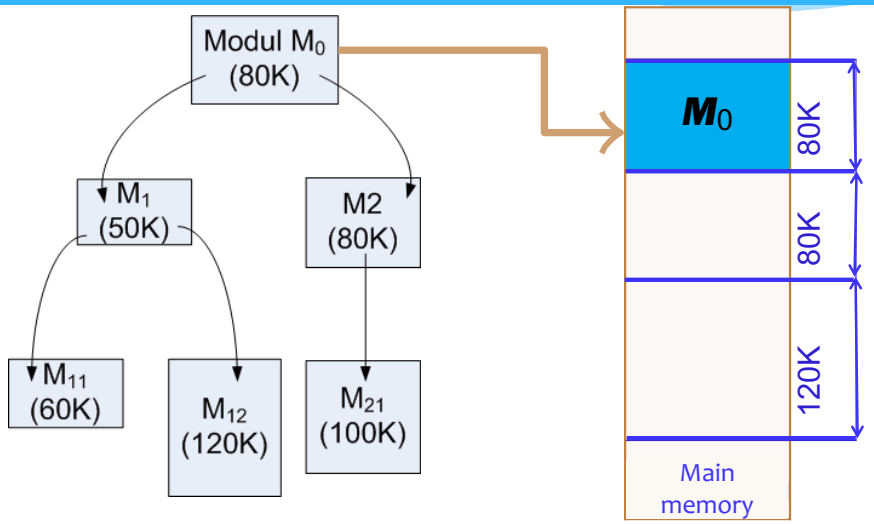
- Modules are divided into different levels
  - Level 0 contains main modul, load and localize the program
  - Level 1 contains modules called from level 0's module and these modules do not exist at the same time
  - . . .
- Memory is also divided into levels corresponding to program's levels
  - Size equal to the same level's largest module's size
- Overlay structure requires extra information
  - Program is divided into how many levels, which modules are in each levels
  - Information is stored in a file (overlay map)
- Module at level 0 is edited into an independent executable file
- When program is executed
  - Load level 0 module similar to a linear structure program
  - When another module is needed, load that module into corresponding memory's level
    - If there are module in the same level exist, bring that module out

Overlays structure



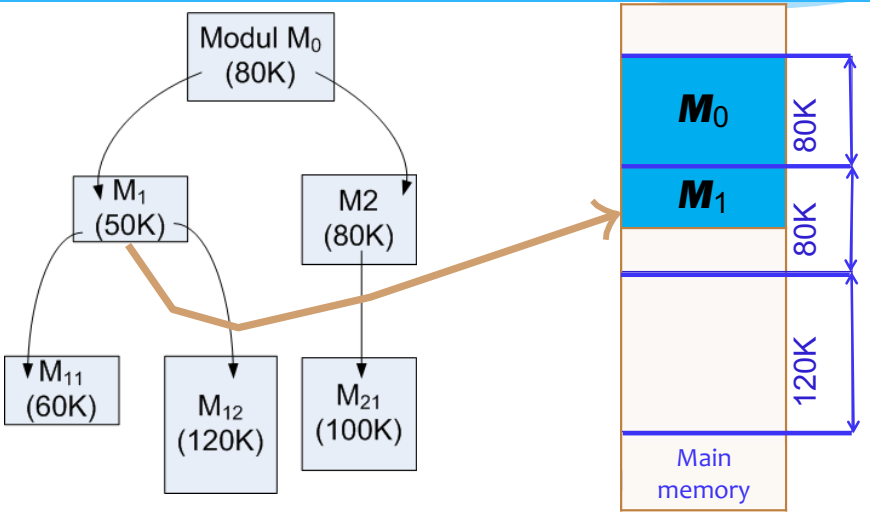
Chapter 3 Memory management
1. Introduction
1.4. Program's structures

Overlays structure



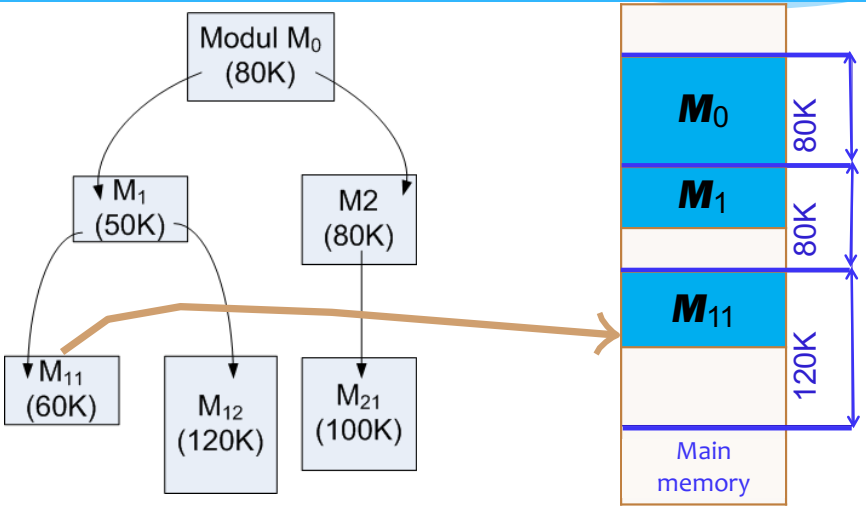
Chapter 3 Memory management
1. Introduction
1.4. Program's structures

Overlays structure

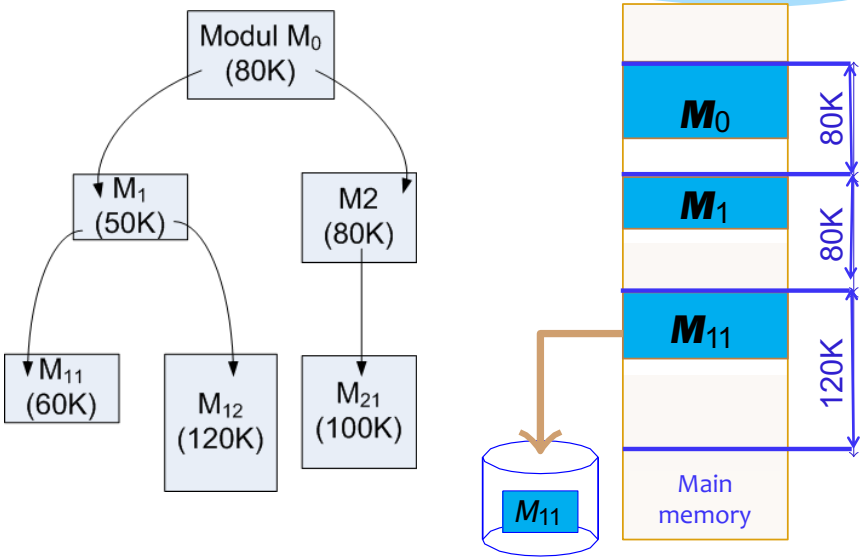


Chapter 3 Memory management
1. Introduction
1.4. Program's structures

Overlays structure

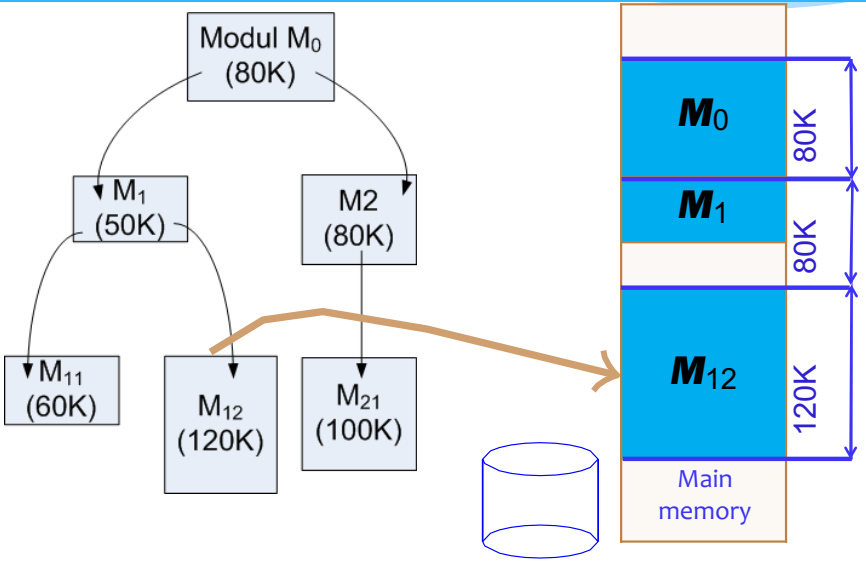


Overlays structure



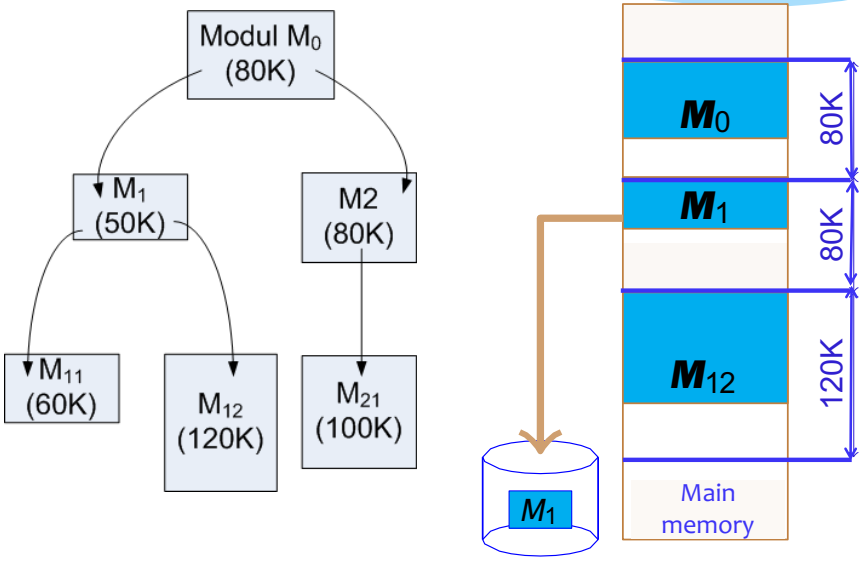
Chapter 3 Memory management
1. Introduction
1.4. Program's structures

Overlays structure



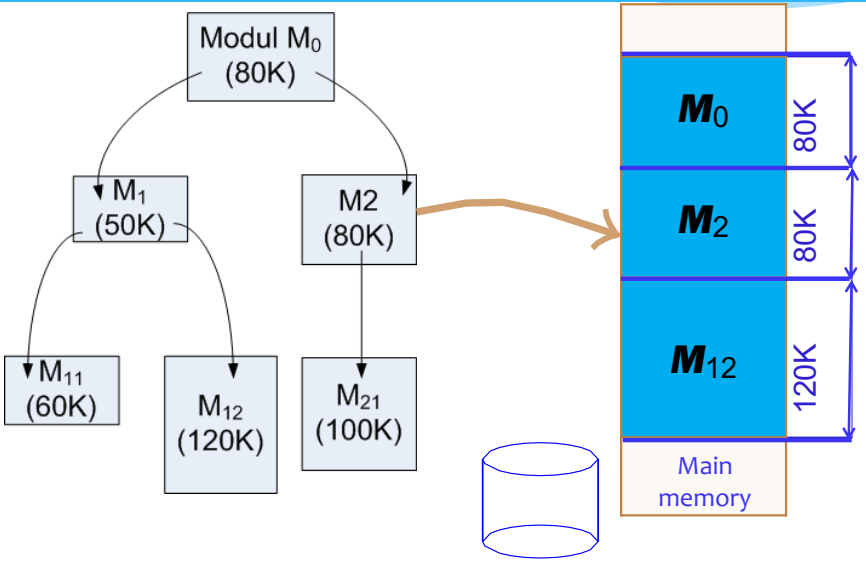
Chapter 3 Memory management
1. Introduction
1.4. Program's structures

Overlays structure



Chapter 3 Memory management
1. Introduction
1.4. Program's structures

Overlays structure



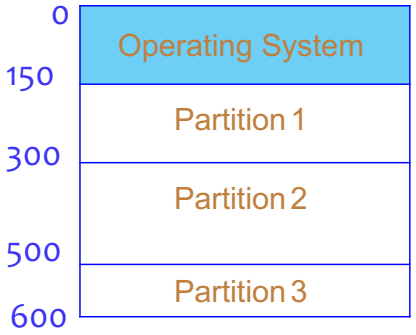
Overlays structure - Conclusion

- Allow program with larger size than memory area size allocated by the operating system
- Require extra information from user
  - Effectiveness is depend on provided information
- Memory usage effectiveness is depend on how program's modules are organized
  - If there are exist modules that larger than other modules in the same level ⇒ the effective is reduced
- Module loading process is dynamic but program's structure is static ⇒ not change at each time running
  - Provide more memory, the effectiveness does not increase

Chap 3 Memory Management

Rule

- Memory is divided into ***n*** parts
- Each part is called a *partition*
  - size: can be unequal
  - Utilized as and independent memory area
    - At a single time, only one program is allowed to exist
    - Programs lie inside memory until finish
- Example: Consider the following system



Process	Size	time
P_1	120	20
P_2	80	15
P_3	70	5
P_4	50	5
P_5	140	12
Queue		

Conclude

- Simple, easy for memory protection
  - Program and memory area have a protection lock
  - Compare 2 locks when program is loaded
- Reduce searching time
- Must copy controlling module into many versions and save at many places
- Parallel cannot be more than *n*
- Memory is segmented
  - Program's size is larger than the largest partition's size
  - Total free memory is large enough but can not load any program

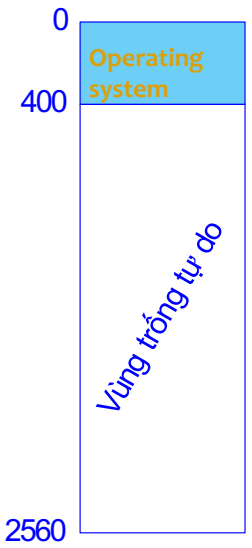
  ⇒Fix partition structure, merge neighboring partition
- Application
  - Large size disk management
  - IBM OS/360 operating system

Rule

Only one management list for free memory

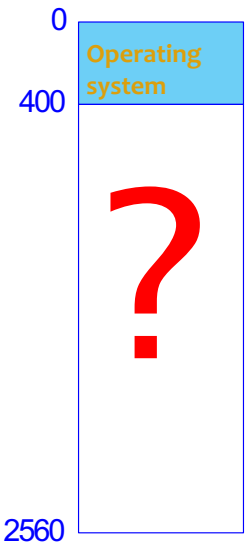
- At the start, the whole memory is free for processes ⇒ largest *hole*
- When a process requests for memory
  - Search in the list for a large enough hole for request
  - If found
    - Hole is divided into 2 parts
    - One part allocate to process as requested
    - One part return to the management list
  - If not found
    - Wait until there is a hole large enough
    - Allow another process in the queue to execution (if the priority is guaranteed)
- When the process finish
  - Allocated memory area is returned to the free memory management list
  - Combine with other neighboring holes if necessary

Main memory



Process	Size	time
P_1	600	10
P_2	1000	5
P_3	300	20
P_4	700	8
P_5	500	15

Waiting file queue



Free memory area selection strategy

Strategies to select free area for process's request

First Fit : First free area satisfy request

Best Fit : Most fitted area

Worst Fit : Largest area that satisfy request

Chapter 3 Memory management
2. Memory management strategies
2.2 Dynamic partition strategy

Free memory area selection strategy

- Suppose free memory area have the size 100K, 500K, 200K, 300K, and 600K (consequently),
- First-fit, Best-fit, and Worst-fit

How will process with size 212K, 417K, 112K, and 426K loaded?

Chapter 3 Memory management
2. Memory management strategies
2.2 Dynamic partition strategy

Memory reallocation problem

After a long working time, the free holes are distributed and caused memory lacking phenomenon ⇒ Need to rearrange memory

- Move process
  - Problem: internal objects when move to new place will has new address
    - Use relocation register to store process's relocation value
  - Select method for lowest cost
    - Move all process to one side ⇒ largest free holes
    - Move processes to create a sufficient free hole immediately
- Swapping process
  - Select a right time to suspend process
  - Bring process and corresponding state to external memory
    - Free allocated memory area and combine with neighboring areas
  - Reallocation to former place and restore state
    - Use relocation register if process is moved to different places

Chapter 3 Memory management
2. Memory management strategies
2.2 Dynamic partition strategy

Concludes

- No need to copy the controlling modules to different places
- Increase/decrease parallel factor depend on the number and size of programs
- Cannot run program with size larger than the physical memory size
- Cause memory waste phenomenon
  - Memory area is not used and not in the memory management list
    - Cause by the operating system error
    - By malicious software
- Cause external memory defragment phenomenon
  - Free memory area is managed but distributed -> cannot used
- Cause internal memory defragment phenomenon
  - Memory allocated to process but not used by process

- Fixed partition strategy
- Dynamic partition strategy
- Segmentation strategy
- Paging strategy
- Segmentation and paging combination strategy

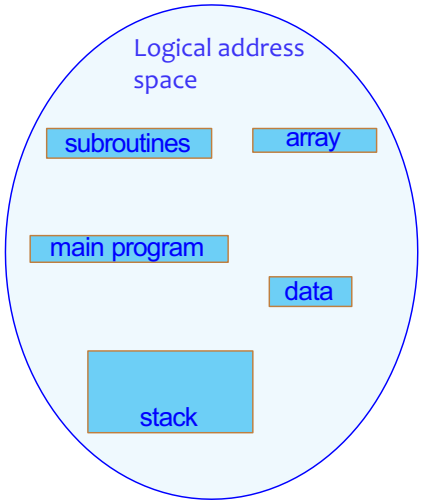
Program

- In general, a program includes following modules
  - main program
  - Set of sub-routine
  - Variables, data structures,. . .
- Modules, objects in program are defined by name
  - function sqrt(), procedure printf() . . .
  - x, y, counter, Buffer. . .
- Member inside a module is determined based on the distance from the head of the module
  - Instruction 10th of function sqrt(). . .
  - Second element of array Buffer. . .

How program is placed inside memory?
- Stack stay at the higher area or Data stay at the higher area?
- Object's physical address . . .?

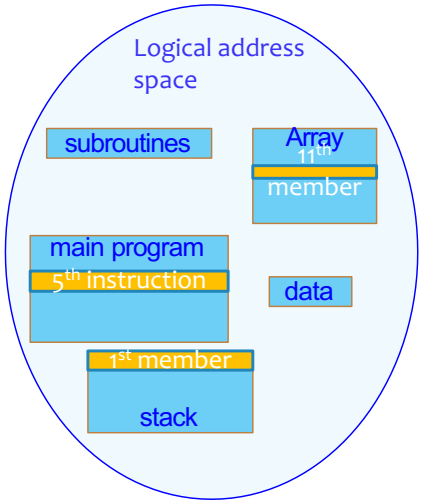
⇒ Users donot care

User's perspective



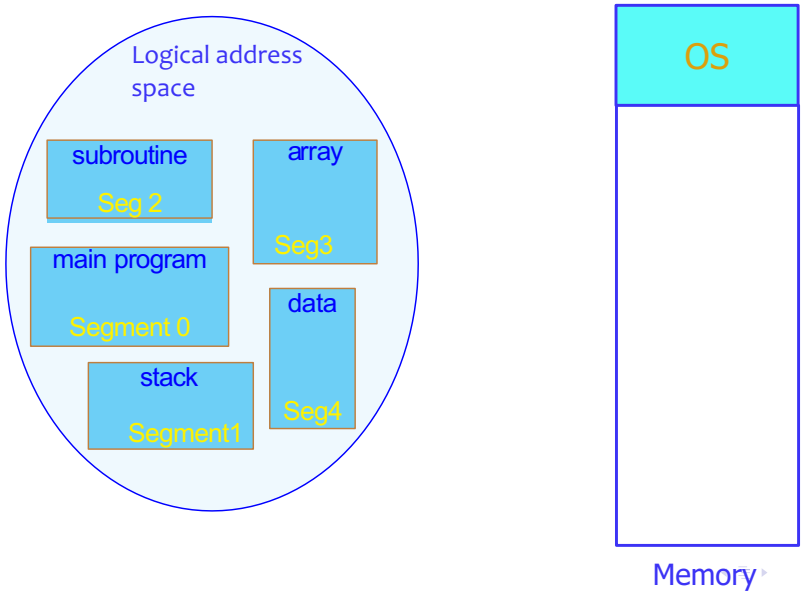
When program is loaded into memory to execute. Program is combined of several segments
- Each segement is a logic block, corresponding to a module
  - Code: main(), procedure, function. . .
  - Data: Global objects
  - Other segments: stack, array. . .
- Each segment occupy an contiguous memory area
  - Has a start position and size
  - Can be located at any place in the memory

User's perspective

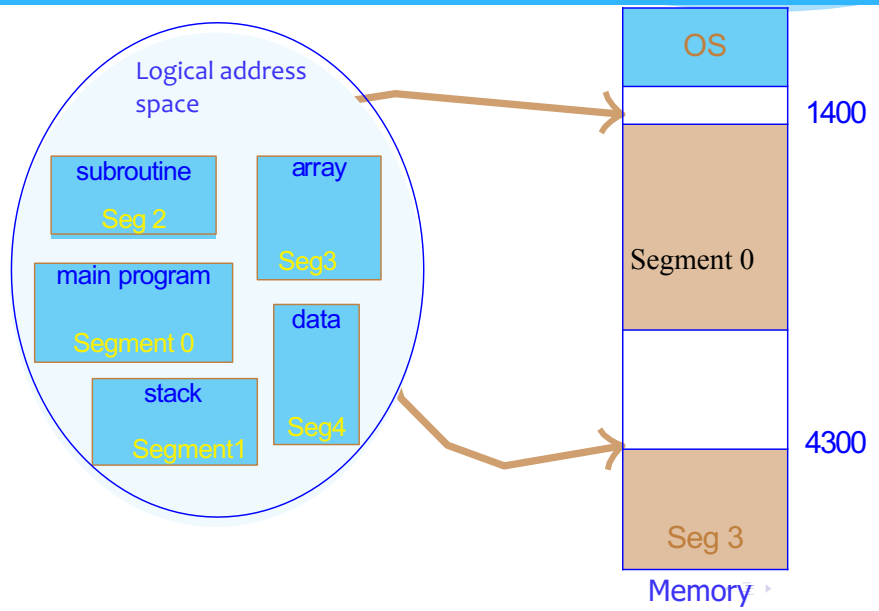


- Object in a segment is defined based on the relative distance from the start of the segment
  - 5th instruction of the main program
  - 1st member of the stack. . .
- Where is the location of these objects in the memory?

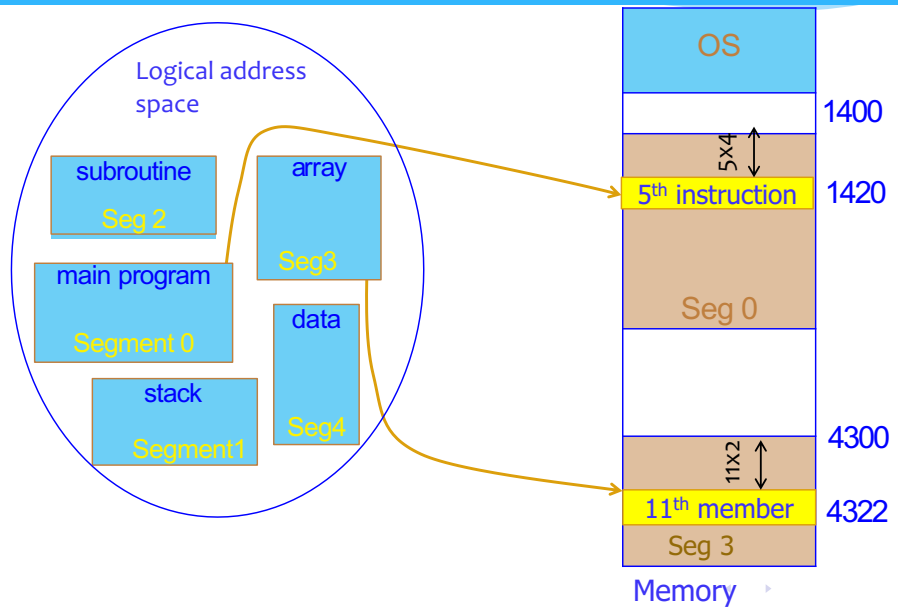
Example



Example



Example



Segmentation structure

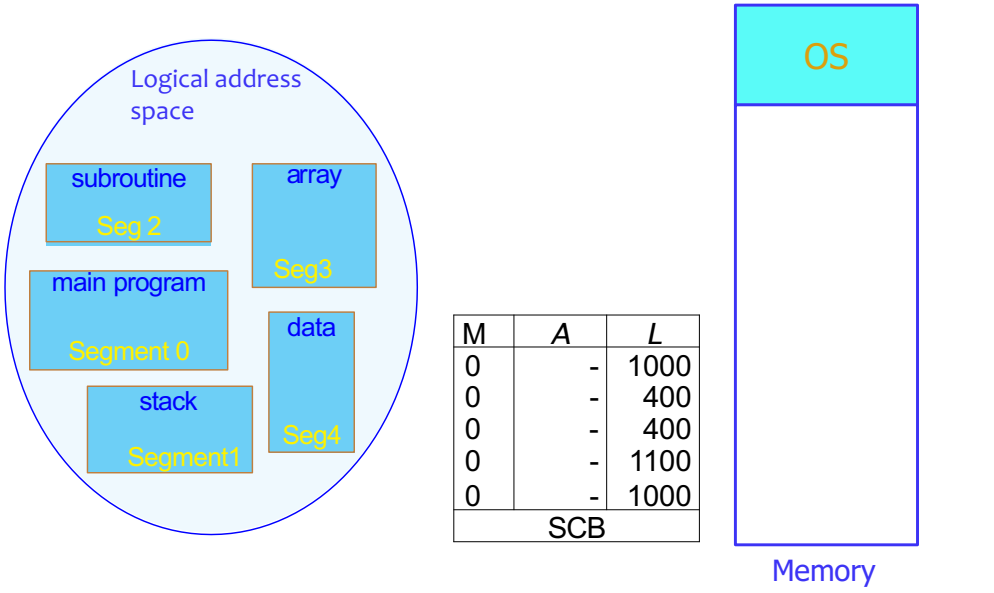
- Program is a combination of modul/segment
  - Segment number, segment's length
  - Each segment can be edited independently.
- Compile and edit program -> create SCB (Segement Control Block)
- Each member of SCB is corresponding to a program's segment

	Mark	Address	Length
0			
...			
n	...	...	...

  - Mark(0/1) : Corresponding segment is already inside memory
  - Address: Segment's base location in memory
  - Length: Segment's length
- Accessing address: segment's name (number) and offset

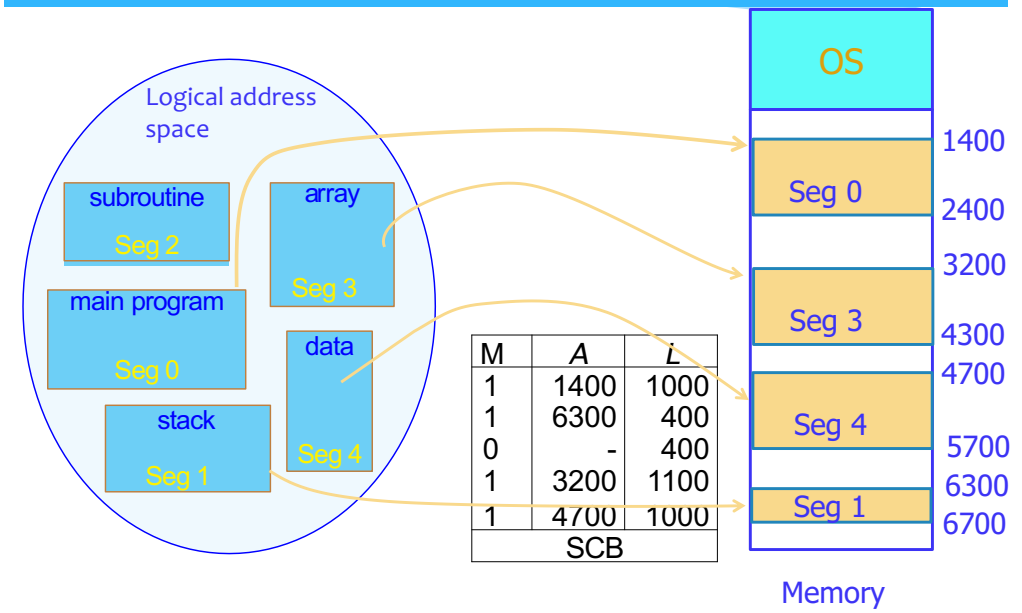
Problem: Convert from 2 dimension address to 1 dimension address

Example



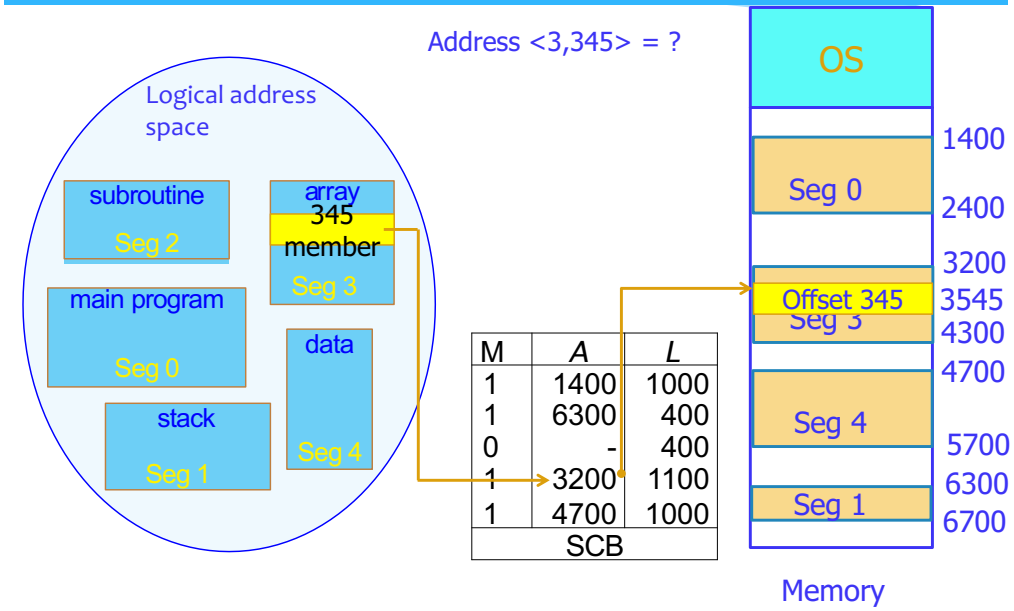
M	A	L
0	-	1000
0	-	400
0	-	400
0	-	1100
0	-	1000
SCB		

Example



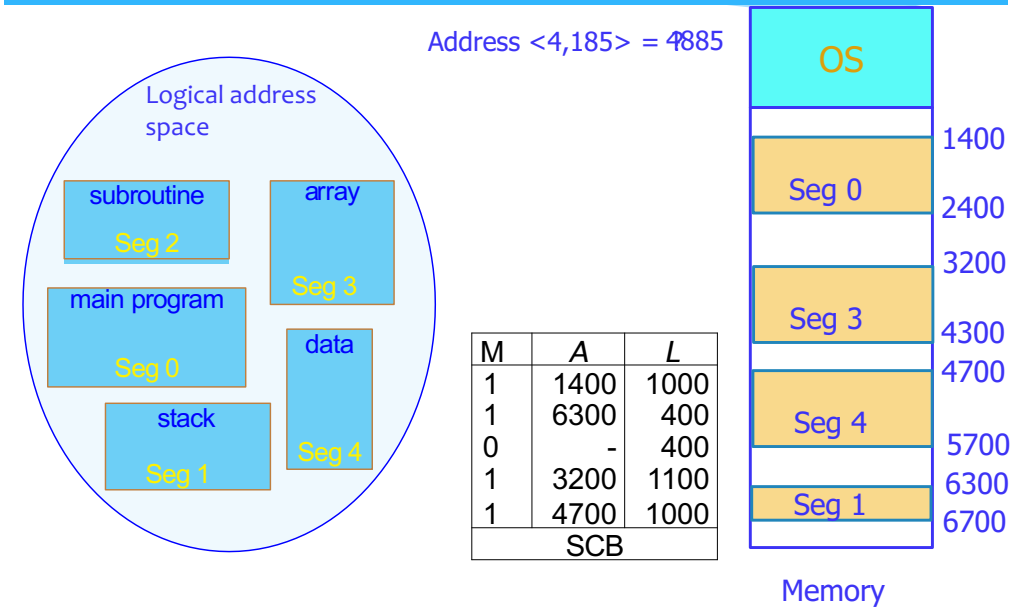
M	A	L
1	1400	1000
1	6300	400
0	-	400
1	3200	1100
1	4700	1000
SCB		

Example



M	A	L
1	1400	1000
1	6300	400
0	-	400
1	3200	1100
1	4700	1000
SCB		

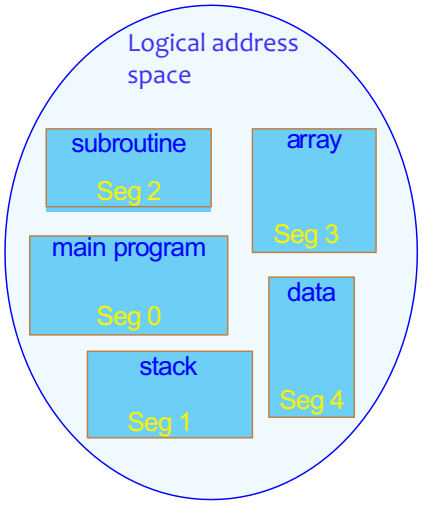
Example



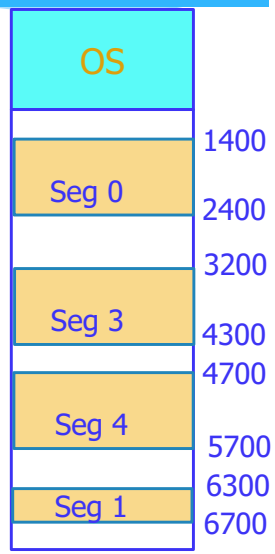
M	A	L
1	1400	1000
1	6300	400
0	-	400
1	3200	1100
1	4700	1000
SCB		

Example

Address <2,120> = ?



M	A	L
1	1400	1000
1	6300	400
0	-	400
1	3200	1100
1	4700	1000
SCB		

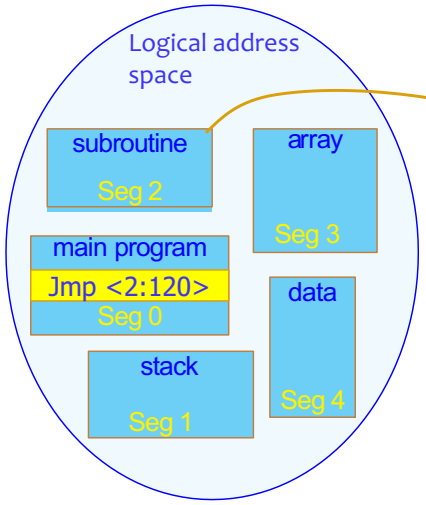


Memory

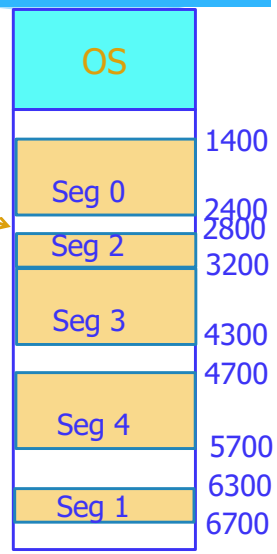
Chapter 3 Memory management
2. Memory management strategies
2.3 Segmentation strategy

Example

Address <2,120> = ?



M	A	L
1	1400	1000
1	6300	400
0	-	400
1	3200	1100
1	4700	1000
SCB		

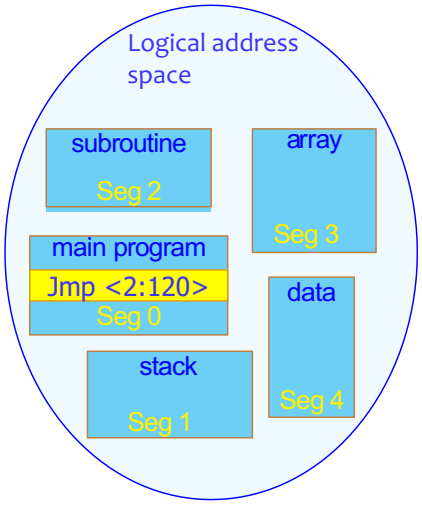


Memory

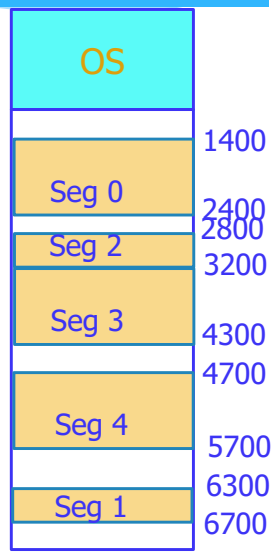
Chapter 3 Memory management
2. Memory management strategies
2.3 Segmentation strategy

Example

Address <2,120> = ?



M	A	L
1	1400	1000
1	6300	400
1	2800	400
1	3200	1100
1	4700	1000
SCB		

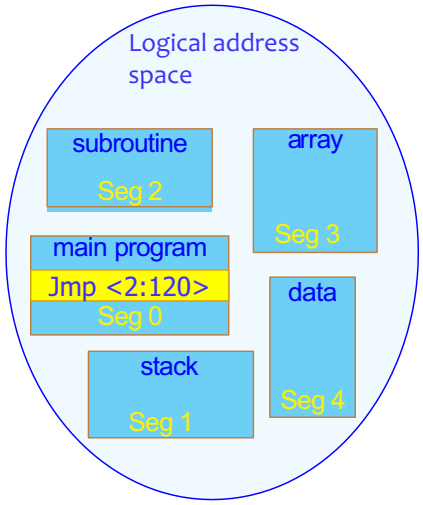


Memory

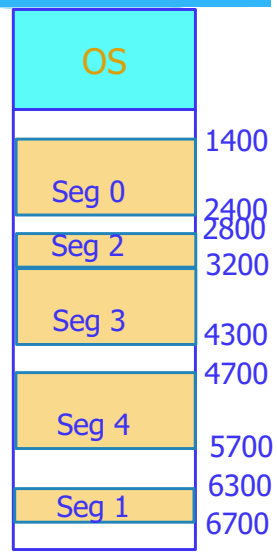
Chapter 3 Memory management
2. Memory management strategies
2.3 Segmentation strategy

Example

Address <2,120> = 2920

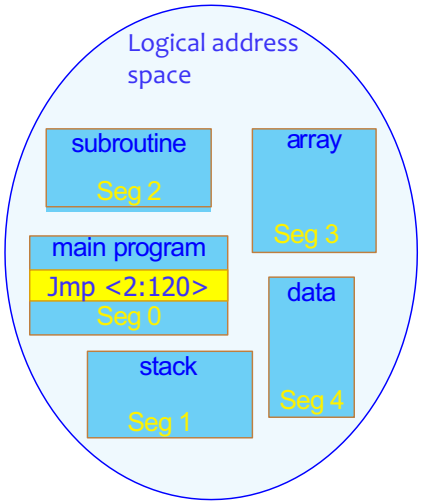


M	A	L
1	1400	1000
1	6300	400
1	2800	400
1	3200	1100
1	4700	1000
SCB		



Memory

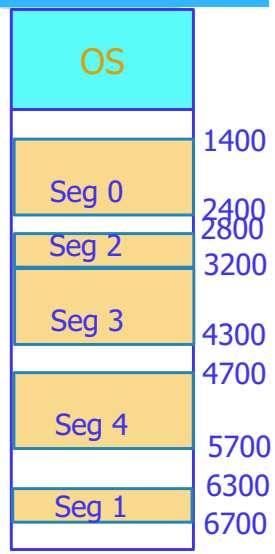
Example



Address <2,450> = ?

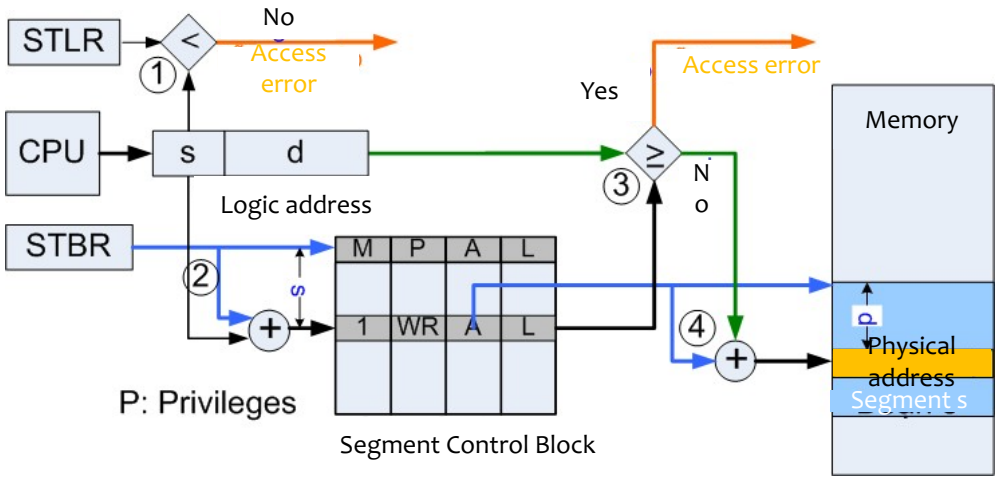
Access error!

M	A	L
1	1400	1000
1	6300	400
1	2800	400
1	3200	1100
1	4700	1000
SCB		



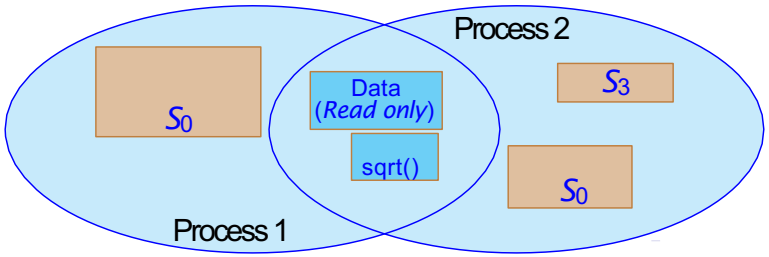
Memory

Address conversion: memory accessing diagram

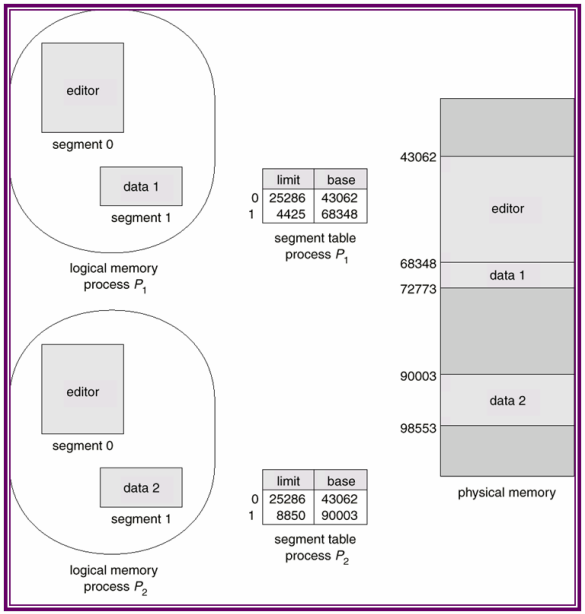


Conclusion: pros

- Module loading diagram does not require user 's participation
- Easy to protect segments
  - Check memory accessing error
    - Invalid address : more than segment's length
  - Check accessing's property
    - Code segment: read only -> Write into code segment: accessing error
  - Check the right to access module
    - Add accessing right (user/system) into SCB
- Allow segment sharing (Example: Text editor)



Segment sharing: Main problem



- Sharing segment
  - Call (0, 120) ?
  - Read (1, 245) ?

=>must has the same index number in the SCB

Conclude: cons

- Effective is depended on the program's structure
- Memory is fragmented
  - Memory allocated by methods first fit /best fit...
  - Require memory rearrangement (relocation, swapping)
    - Easier with the help of SCB
      - M ← 0 : Segment is not in memory
      - Memory's area defined by A and L is returned to the free memory management list
    - Select which module to bring out
      - Longest existed module
      - Last recently used module
      - Least frequently used module ⇒ Require media to record number and time that module is accessed
- Solution: allocate memory for equal size segment (page)?

- Fixed partition strategy
- Dynamic partition strategy
- Segmentation strategy
- Paging strategy
- Segmentation and paging combination strategy

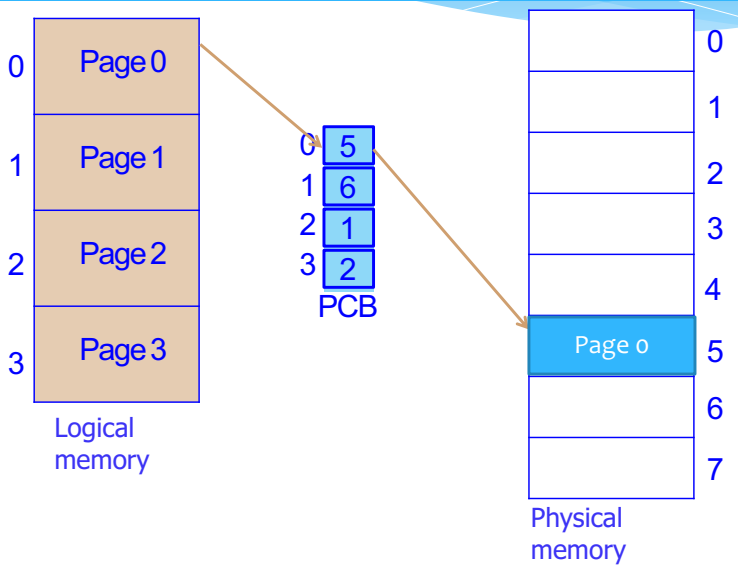
Rule

- Physical memory is divided into equal size blocks: page frames
  - Physical frame is addressed by number 0, 1, 2, . . . : frame's physical address
  - Frame is the unit for memory allocation
- Program is divided into blocks that have equal size with frame (pages)

Rule (cont)

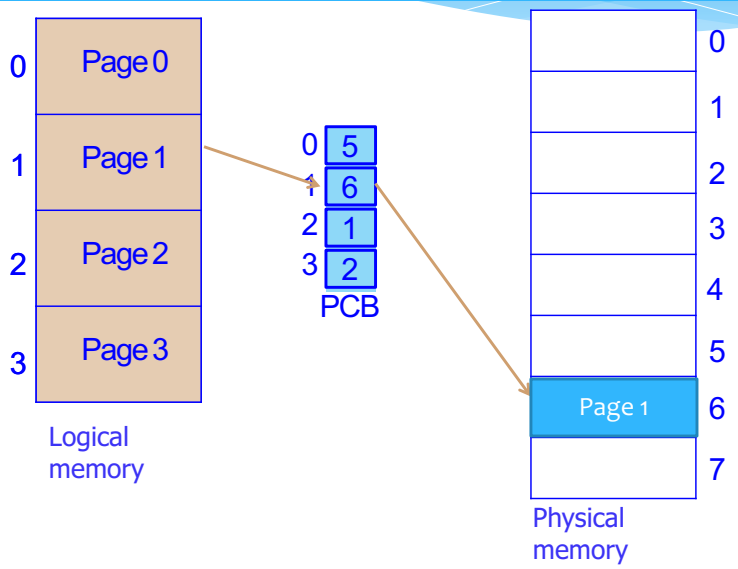
- When program is executed
  - Load logical page (from external memory) into page's frame
  - Construct a PCB(Page Control Block) to determine the relation between physical frame and logical page
  - Each element of PCB is corresponding to a program's page
    - Show which frame is holding corresponding page
    - Example PCB[8] = 4 ⇒ ?
  - Accessing Address is combined of
    - Page's number (p) : Index in PCB to find page's base address
    - Displacement in page (d): Combined with base address to find the physical address

Example



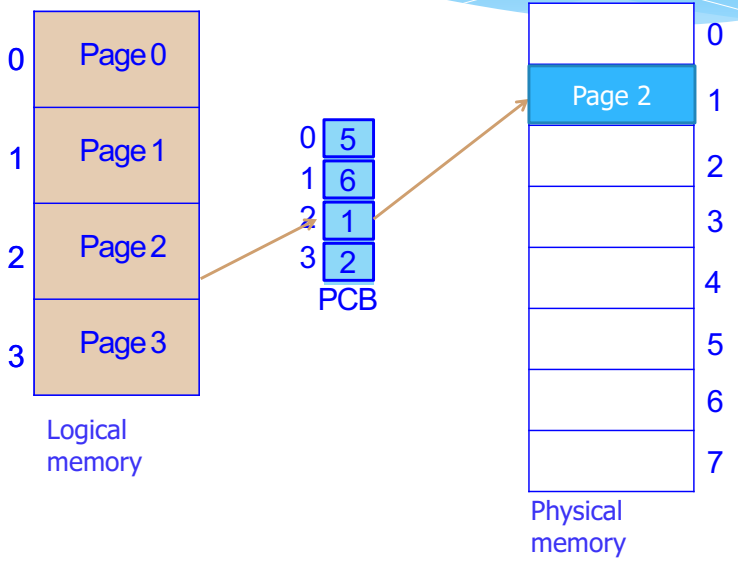
Chapter 3 Memory management
2. Memory management strategies
2.4 Paging strategy

Example



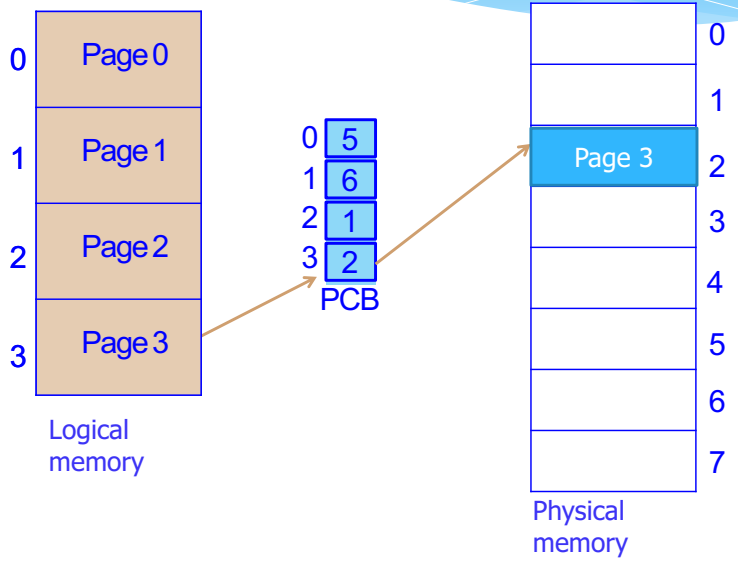
Chapter 3 Memory management
2. Memory management strategies
2.4 Paging strategy

Example



Chapter 3 Memory management
2. Memory management strategies
2.4 Paging strategy

Example



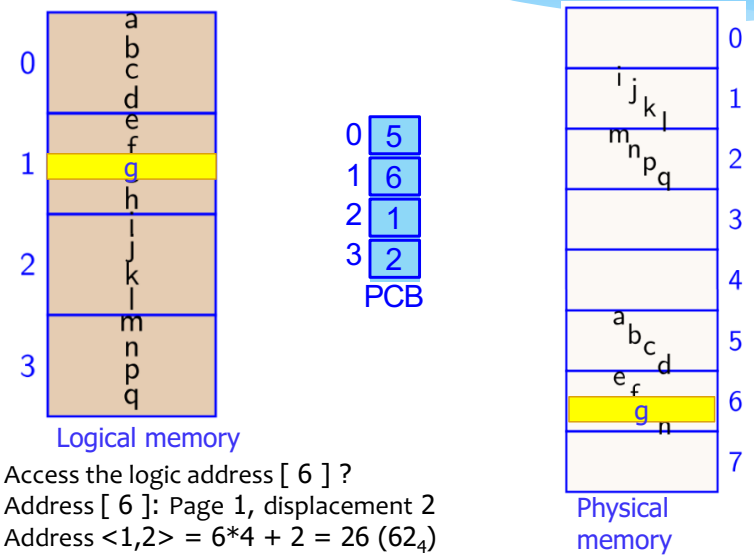
Note

- Frame size is always power of 2
  - Allow connection between frame number and displacement
  - Example: memory is addressed by n bit, frame's size 2^k

frame	displacement
n – k	k

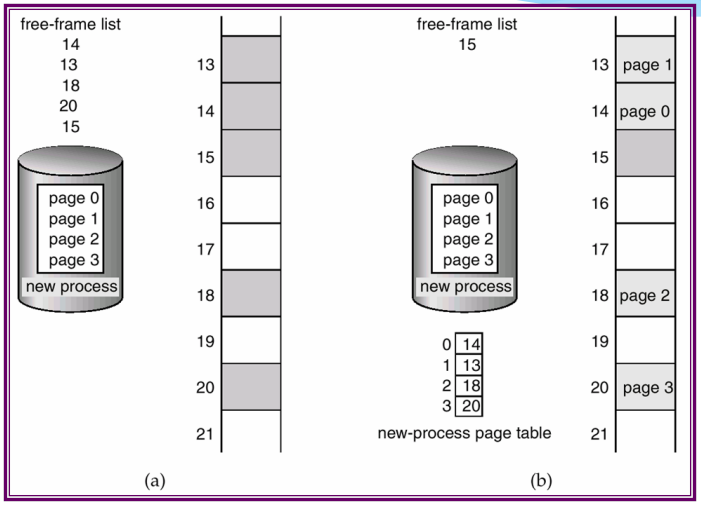
- Not necessary to load all page into memory
  - Number of frame is limited by memory's size, number of page can be unlimited
  - PCB need Mark filed to know if page is already loaded into memory
    - M = 0 Page is not loaded
    - M = 1 Page is loaded
- Distinguish between paging and segmentation
  - Segmentation
    - Module is depend on program's structure
  - Paging
    - Block's size is independent from program
    - Block size is depend on the hardware (e.g.: 2^9 → 2^{13} bytes)

Example



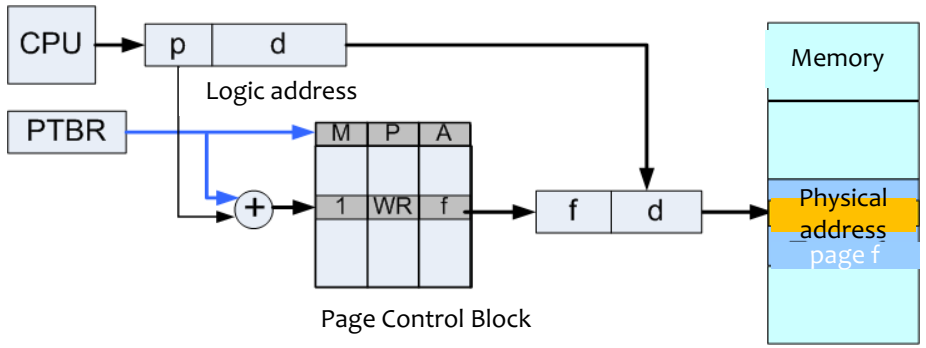
Access the logic address [6] ?
Address [6]: Page 1, displacement 2
Address <1,2> = 6*4 + 2 = 26 (62_4)

Program is running → Load program into memory



- If number of unused frame is enough ⇒ load all page
- If not enough ⇒ load parts of pages

Address conversion: Accessing diagram



Load and replace page

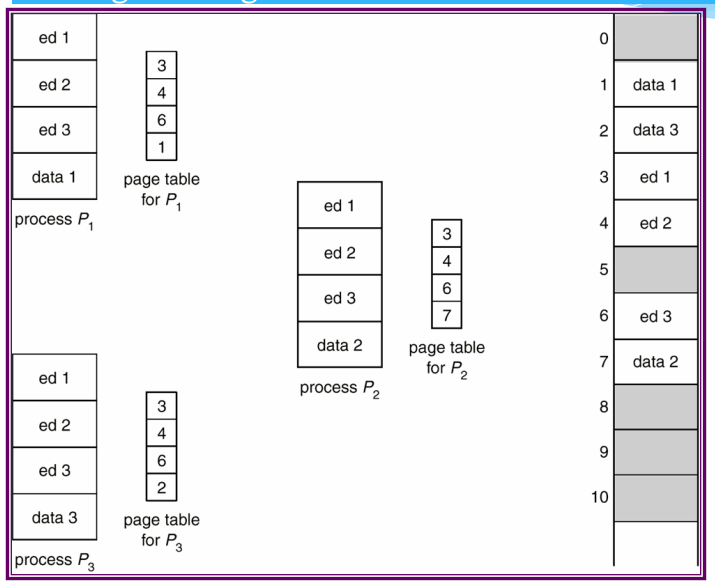
- Remark
  - Number of frame allocated to program
    - Large => Faster execution speed but parallel factor decrease
    - small => High parallel factor but execution speed slow because page is not inside memory

  ⇒ Effectiveness is depend on the page loading or page replacing strategy
- Page loading strategy
  - Load all page: Load all program
  - Prior loading: predict next page will be used
  - Load on demand: Only load page when it's necessary
- Page replacing strategy
  - FIFO First In First Out
  - LRU Least Recently Used
  - LFU Least Frequently Used
  - . . .

Advantage

- Increase memory access speed
  - Access memory 2 times (PCB and required address)
  - Perform connecting instead of adding operation
- No external fragmentation phenomenon
- High parallel factor
  - Only need several program's page inside memory
  - Program can have any size
- Easy to perform memory protection
  - Legally access address (not more than page size)
  - Access property (read/write)
  - Access right (user/system)
- Allow sharing page between processes

Page sharing: Text editor



- Each page size 50K
- 3 pages for code
- 1 page for data
- 40 user
- No sharing
  - Need 8000K
- Sharing
  - Require 2150K

Page sharing: rule

- Necessary while working in sharing environment
  - Reduce the size of memory area for all processes
- Sharing code
  - Only one copy of sharing page in the memory
    - Example: text editor, compiler....
  - Problem: Sharing code can not change
    - Sharing page must be in the same logic address of all address

      ⇒ Same page id in the PCB
- The code and data are separately
  - Separate for each process
  - Can be in any position in the logic memory of the process

Disadvantage

- Have external memory fragmentation
  - Always appear at the last page
  - Reduce memory fragmentation by reduce page size ?
    - Page fault more frequent
    - Large page control table
- Require support from hardware
  - Cost for paging is high
- When the program is large, page control block has many members
  - Program size 2^{30}, page size 2^{12} -> PCB has 2^{20} members
  - Spend more memory for store PCB
  - Solution: multi level page

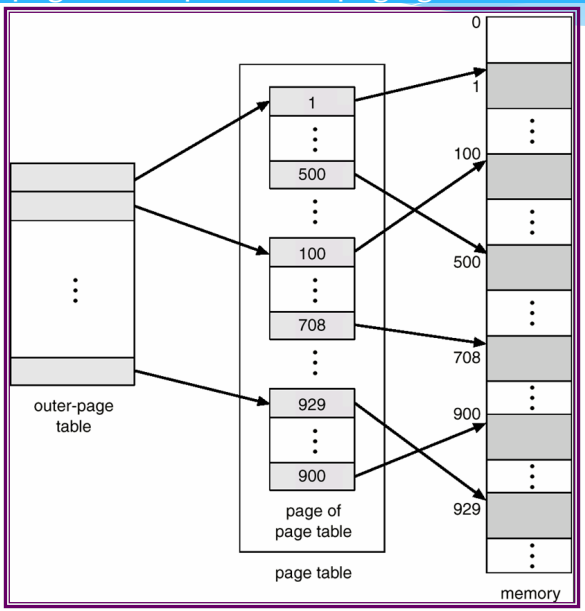
Multi-level paging

Rule: Divide PCB into pages

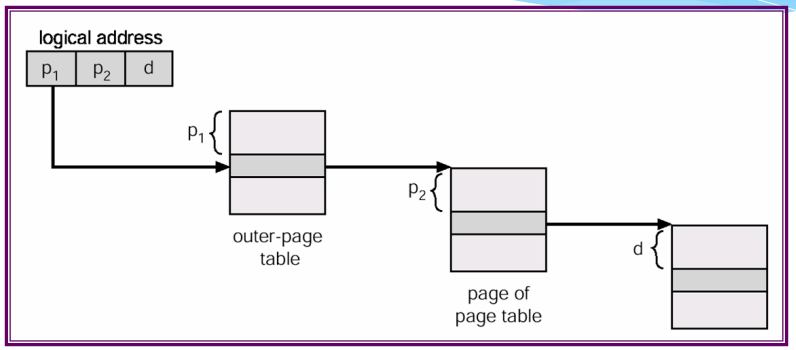
Example: 2 level paging

- Computer use 32 bit for addressing (2^{32}); Page size 4K (2^{12})
  - Page number - 20 bit
  - Offset in page -12 bit
- PCB is paged. Page number is divided into
  - Outer page table (page directory) - 10 bit
  - Offset in a page directory – 10 bit
- Access address has the form <p_1, p_2, d >

Multi-level page: Example 2 level paging



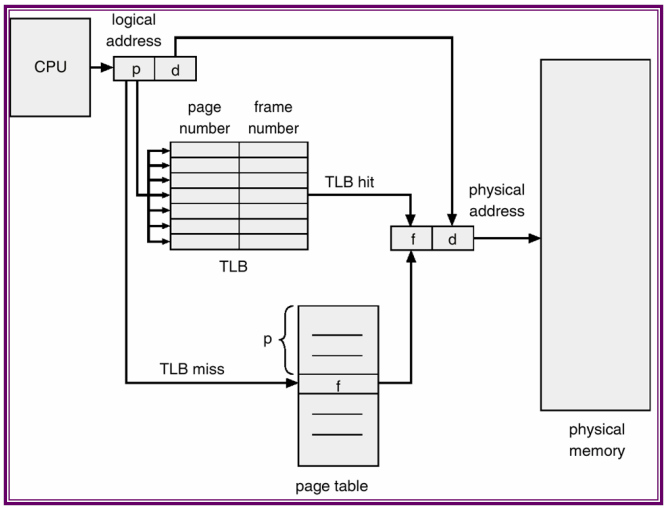
Multi-level page: Memory access



- When access: System load page directory into memory
- Unused page table and unused page are not necessary loaded into memory
- Require 3 times access memory
- Problem: For 64 bit system
  - 3, 4,... Level paging
  - Require access memory 4, 5,... times ⇒ slow
  - Solution: address translation buffer

Address translation buffer

TLB: translation look-aside buffers



- Associative registers
- Parallel access
- Each member contains
  - Key: Page number
  - Value: Frame number
- TLB contain recently accessing page
- When requested <p,d>
  - Search p in TLB
  - Missing p, find p in PCB then add < p, f > into TLB

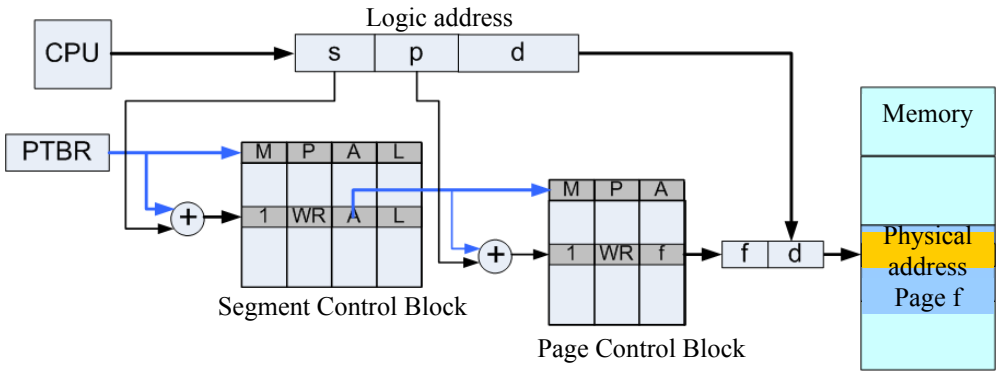
98% memory access is done via TLB

- Fixed partition strategy
- Dynamic partition strategy
- Segmentation strategy
- Paging strategy
- Segmentation and paging combination strategy

Rule

- Program is edited as in segmentation strategy
  - Create SCB
  - Each member of SCB correspond to one segment, has 3 fields M, A, L
- Each segment is edited separately as in paging strategy
  - Create PCB for each segment
- Memory accessing address: combination of 3 < s, p, d >
- Perform accessing address
  - STBR + s ⇒: address of s member
  - Check value of Ms, load PCBs if it's necessary
  - As + p ⇒ Load the address of member p in PCBs
  - Check value of Mp, load page p if it's necessary
  - Connect Ap with d => physical address if it's necessary
- Utilized in processor Intel 80386, MULTICS . . .

Memory access diagram



Conclusion

M_0 2340B M_1 5730 B

M_2 4264 B M_3 1766 B

Segmentation

M	A	L
0	–	2340
1	2140	5730
0	–	4264
0	–	1766

SCB

Segmentation and paging combination

M	A	L
0	–	3
0	5	6
0	–	5
0	–	2

SCB

M	A
0	–
1	8
0	–
0	–
0	–
0	–

PCB_2

Chap 3 Memory Management

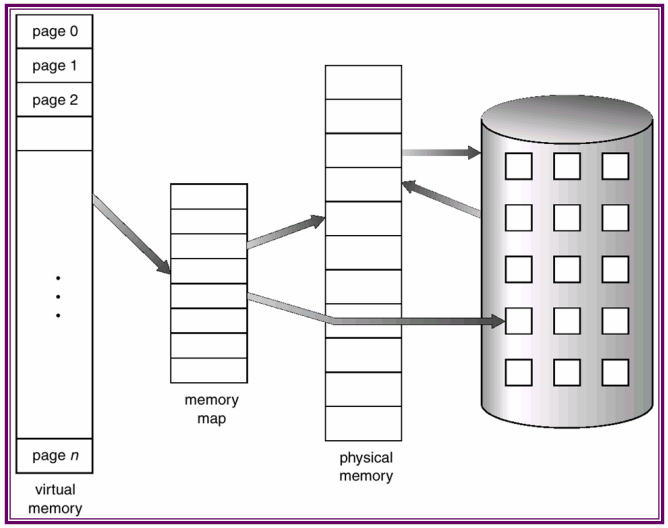
Program and memory issue

- Instruction must be placed in memory when executed !
- Whole program must stay inside memory ?
  - Dynamic loading, Overlays structures… : Partly loaded
    - Require special notice from programmer

  ⇒ Not necessary
  - Program's segment for handling errors
    - Errors occur least frequently, least frequently executed
  - Unused declared data
    - Declare a matrix 100x100, use 10x 10
- Run program with one part inside memory will allow
  - Write program in virtual address space
    - Unlimited size
  - Many program concurrently existed

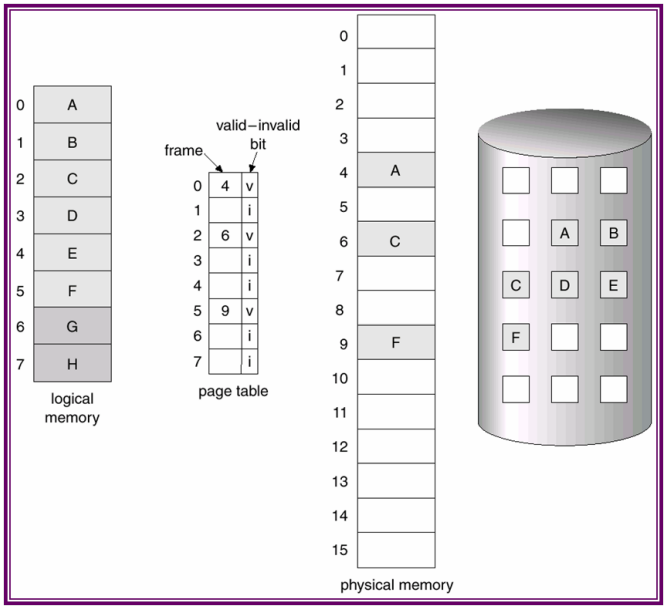
    ⇒ Increase CPU's productivity
  - Reduce I/O request for loading and swapping programs
    - The size of the swapped part is smaller

Concept of virtual memory



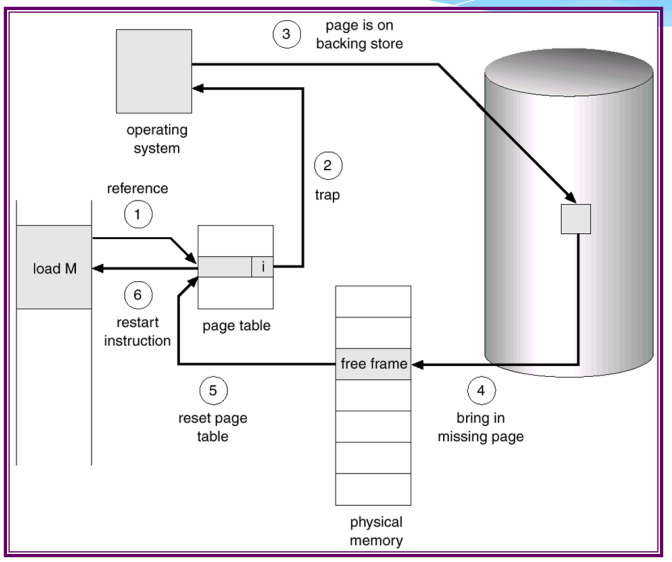
- Utilize the secondary storage (*HardDisk*) to store the unloaded program's part
- Separate logic memory (*user's space*) from physical memory
- Allow mapping large logical memory area to small physical memory area
- Implemented by
  - Segmentation
  - Paging

Load parts of program into memory



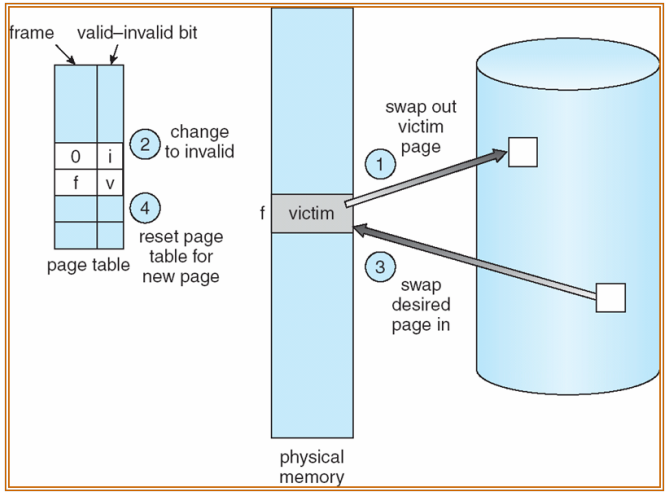
- Process's page:
  - Physical memory,
  - Some pages stay on disk (virtual memory)
- Represented by one bit in the PCB
- When a page is required, load page from secondary memory -> physical memory

Page fault handling



If there are no free frames, need to replace pages

Page replacement



① Determine location of logical page on disk
② Select physical frame
- Write to disk
- Modify bit valid-invalid

③ Load logic page into selected physical frame
④ Restart process

① Introduction

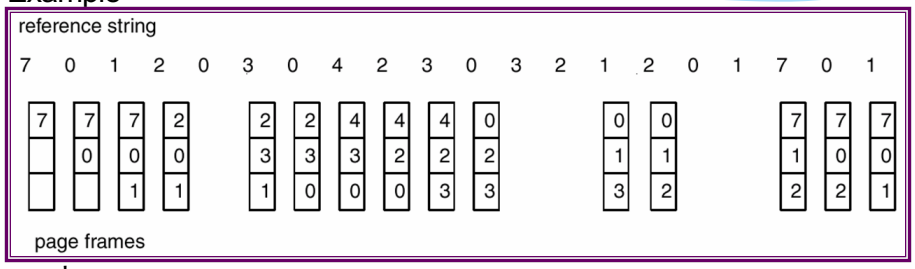
② Page replacement strategy

Strategies

- FIFO: First In First Out
- OPT/MIN: Optimal page replacing algorithm
- LRU: page that is Least Recently Used
- LFU: page that is Least Frequently Used
- MFU: page that is Most Frequently Used
- . . .

FIFO

Example

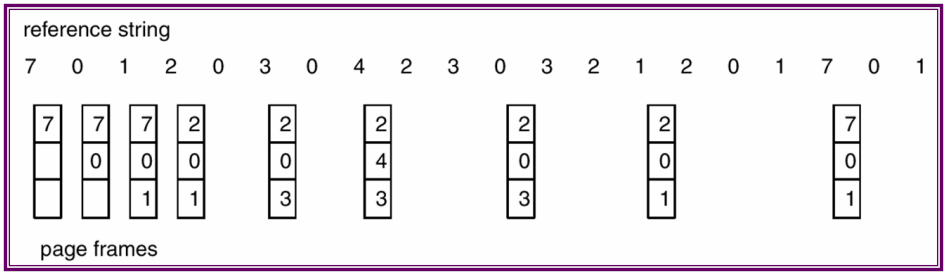


Remark

- Effective when the program has the linear structure
- Least effective when the program has different modules call
- Easy to implement
  - Utilize a queue to store program's pages inside memory
  - Insert into last position in the queue, replace page at the first position
- Increase physical page, no guarantee of reducing page fault
  - Accessing sequence: 1 2 3 4 1 2 5 1 2 3 4 5
  - 3 frames: 9 page faults; 4 frames: 10 page faults

OPT

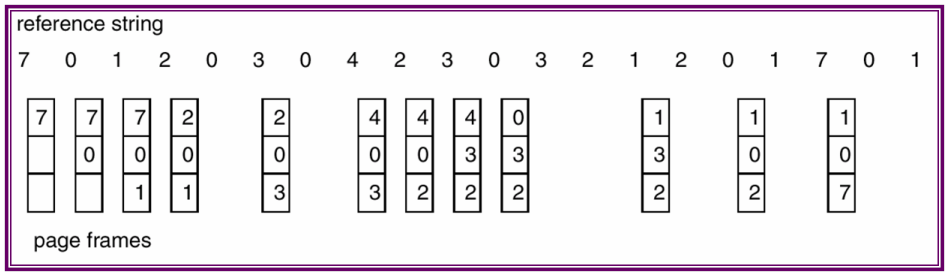
Rule: Replace page that has longest next used time



- Number of page fault is smallest
- Problem: it's hard to predict program sequence

LRU

Rule: Replace page that is least recently used



- Effective for page replacing strategy
- Guarantee that page fault is reduced when increase physical frame
  - Set of pages in memory with n frames is always a subset of pages in memory that have n + 1 frames
- Require support to know the last recently accessed time
- How to implement?

LRU: Implementation

- Counter
  - Add one more field to record the accessed time into each member of PCB
  - Add a clock/counter to the Control Unit
  - Where there is a page access request
    - Increase counter
    - Copy the counter content into the newly added field in PCB
  - Need a procedure to update PCB (write to the added field) and procedure to search for a smallest accessed time value
  - Number Overflow phenomenon !?
- List or Stack
  - Use a list or stack to record page number
    - Access to a page, put corresponding member to the top position
  - Replace: member in the last position
  - Usually implemented as a 2 dimension linked list
    - 4 pointer assigning operation ⇒ time consuming

Algorithm based on counter

Use counter (one field of PCB) to record the last time page is accessed

- LFU: Page that has smallest counter will be replaced
  - Page that is frequently used
    - Important page ⇒ reasonable
    - Initialization page, only used at start ⇒ unreasonable ⇒ Shift right the counter by 1 bit (time div 2)
- MFU: Replace page with largest counter
  - Page with smallest value, just recently loaded and not used much

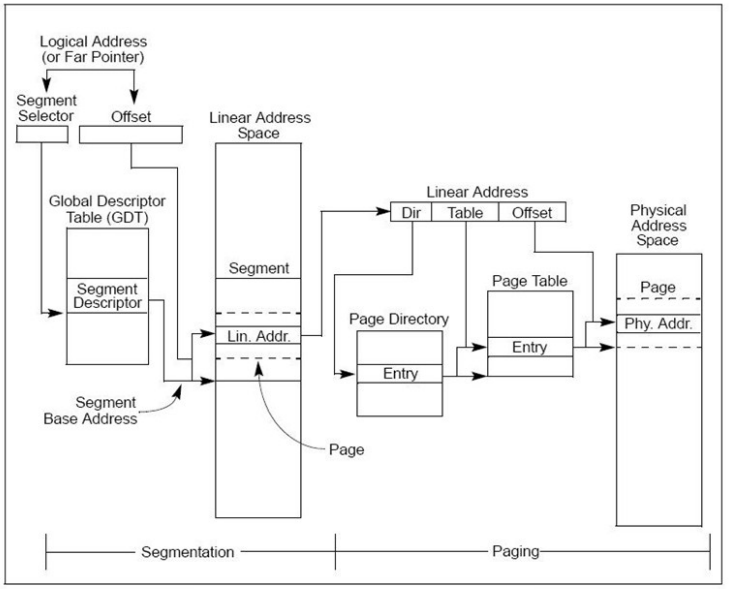
Chap 3 Memory Management

Memory management modes

- Intel 8086, 8088
  - Only one management mode: Real Mode
  - Managed memory area up to 1MB (20bit)
  - Xác định địa chỉ ô nhớ bằng 2 giá trị 16 bit: Segment, Offset
    - Segment register: CS, SS, DS, ES,
    - Offset register: IP, SP, BP...
    - Physical address: Seg SHL 4 + Ofs
- Intel 80286
  - Real mode, compatible with 8086
  - Protected mode
    - Utilize segmentation method
    - Exploit physical memory up to 16M (24bit)
- Intel 80386, Intel 80486, Pentium,..
  - Real mode, compatible with 8086
  - Protected mode : Combination of segmentation and paging
  - Virtual mode
    - Allow to run 8086 code in protected mode

Chapter 3: Memory management
4. Memory management in Intel's processors family

Protected mode in Intel 386, 486, Pentium,..



Hệ Điều Hành

(*Nguyên lý các hệ điều hành*)

Đỗ Quốc Huy
huydq@soict.hust.edu.vn
Bộ môn Khoa Học Máy Tính
Viện Công Nghệ Thông Tin và Truyền Thông

Chapter 4 file system management

- To keep information for long time and later use -> store on external memory (disk, magnetic tape, optical disk,..) => file
  - Data or program file
  - Many files ⇒ file system
  - Files system combined of 2 parts
    - files: Contain data/program
    - directory : provide information about file
- Files system is large ⇒ How to manage?
  - File's properties, required operation?
- How to store and access file on storage devices?
  - Storage space allocation, free memory management

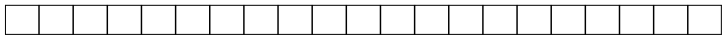
Chapter 4 File system management

1. File system
2. File system's implementation
3. Information organization on disk
4. FAT system

- File's notion
- Directory structure

Introduction

- Information is stored on different medias/devices
  - Example: Disk, tape, optical disk…
  - Storage device is modeled as an array of memory block



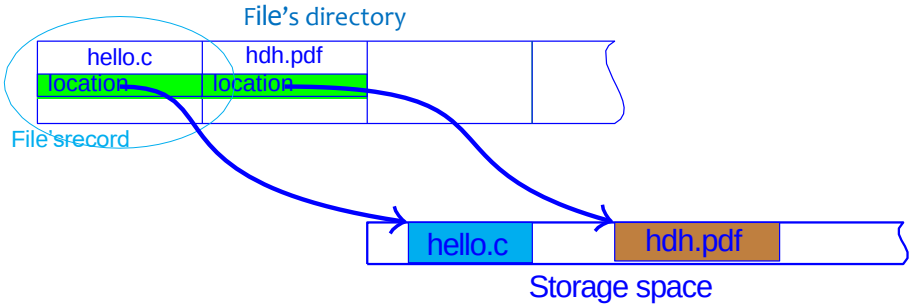
- File is a set of information stored on storage devices
  - File is a storage unit of OS on external devices
  - File contains sequence of bits, bytes, lines, records,… Contain meaning defined by creator
- Structure of file is defined by type of file
  - Text file: Sequence of character organized into lines
  - Object file: Bytes organized into block to be read and edited by the linker
  - Executive File: Sequence of codes can be run in memory
  - . . .

File's attributes

- File name: sequence of character (e.g. "hello.c")
  - Information store in the format that is readable by user
  - Can be distinguished by upper-case/lower-case
  - Guarantee the independence of file from process, user…
    - A created file hello.c by notepad on Windows
    - B uses emacs on Linux to modify a file specified by name hello.c
- Identifier: A tag to define an unique file
- Type: Used in system supports many types of file
  - Type of file is defined based on one part of file name
    - Example: .exe, .com/ .doc, .txt/ .c, .jav, .pas/ .pdf, .jpg,…
  - Based on type, OS decides corresponding operation
    - Run an execution file that source is modified ⇒ Recompile
    - Double click on a text file (*.doc) ⇒ Call word processor
- Position: Point to device and location of file on that device
- Size: Current/maximum size of file
- Protection: Control access: who can read/write…
- Time: Creation time, modified time, last used time …

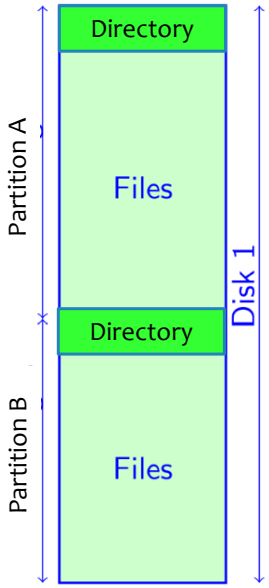
File's attributes (cont.)

- File's attributes are stored in data structure: File's record
  - May contain only file's name and file's identifier; file's identifier define other information
  - Size from several bytes to kilobytes
- File's record are stored in File's directory
  - Size may be up to Megabytes
  - Often stored on external memory
  - Directory parts are loaded into memory when necessary



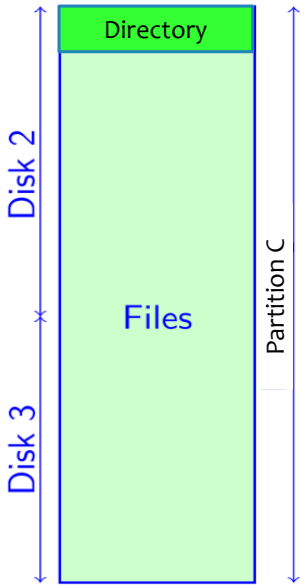
- File's notion
- Directory structure

Partition



- Disk is divided into Partitions, Minidisks, Volumes
- Each partition is processed as an independent storage area
- Can store an individual OS

Partition



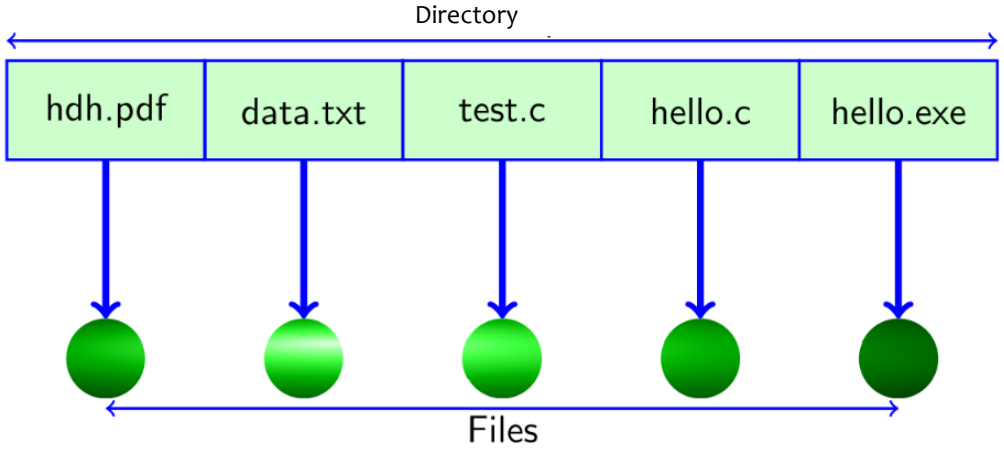
Merge several disks into a large logic structure

- User only care about file and directory 's structure
- Do not care about how physical memory space is allocated to files

Operations with directory

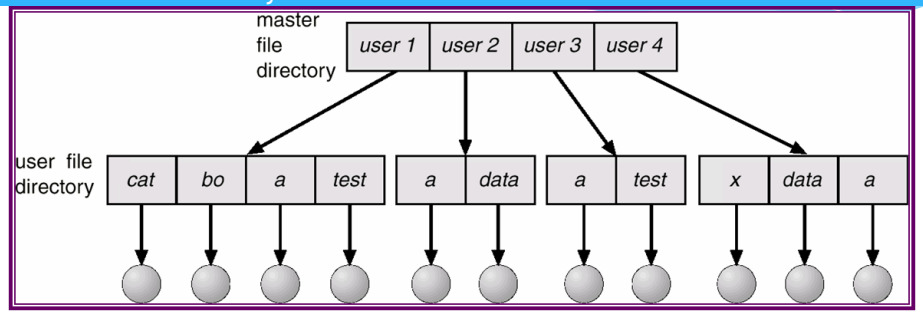
- Each partition contain information about contained files
  - File's information is stored in device's directory
- Directory is a translation table allow mapping from one name (file) to a member inside directories
  - Directory can be implemented by different methods
- Require operations for insert, create, delete or show the list
- Operations
  - Seek file: Search for an item corresponding to a file name
  - Create file: Create new file require to create new item in directory
  - Delete file: When delete a file, remove corresponding item in the directory
  - Listing file: Show the list of files and corresponding item in directory
  - Rename file: Rename file, position in the directory's structure
  - Traverse the file system: Access all directory and content of all files in the directory (backup data to tape)

One level directory



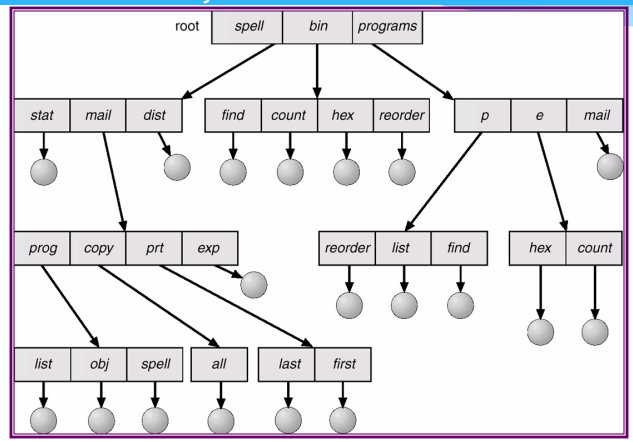
- Simplest structure, files are stored in the same directory
- If number of user and file is large, it's possible for the filename to be duplicated
  - Each user has his own directory riêng

One level directory



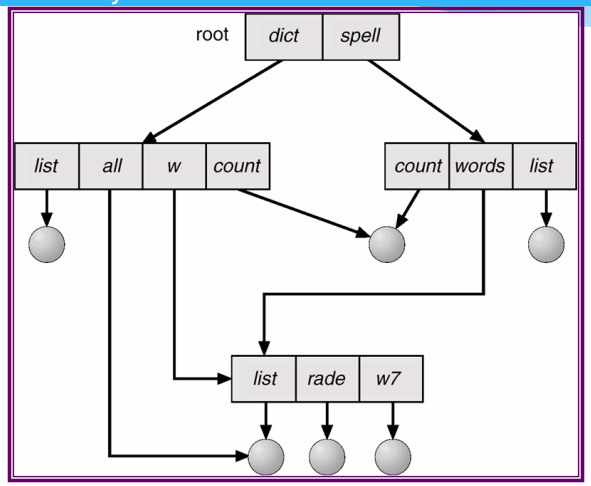
- Each user has his own private folder, when working, only work with private folder
- When log in, system check and allow user to work with private folder
- When a user is added
  - System create a new member in *Master file directory*
  - Create *User file directory*
- Solve the duplicate file name problem; Effect when user are independent
- Problem when user want to share a file

Tree structure directory



- A (relative/absolute) path to file exist
- Sub-directory is a file is treated specially (bit for marking)
- Operations create/delete/add... performed on current directory
  - Delete a sub-directory ⇒ delete its sub-tree

Sharing a directory



- User can link to a file from other user
- When traversing the directory (backup), file can be traversed many time
- Delete file: link/ content (file-created-user /last link)

Chapter 4 File system management

1. File system
2. File system's implementation
3. Information organization on disk
4. FAT system

- Directory implementation
- Storage area allocation methods
- Free storage area management

Method

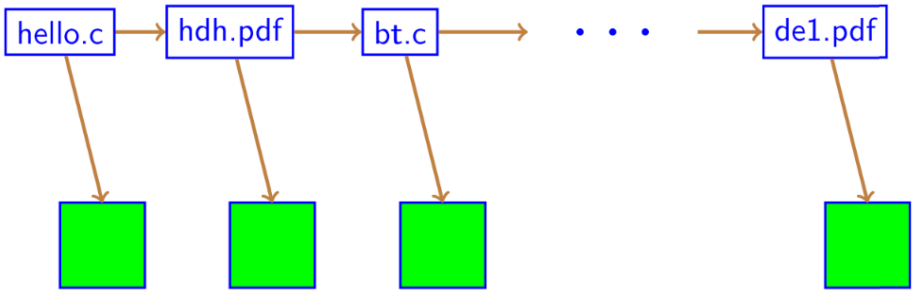
① Linear list with pointer to data blocks
- Simple for programming
- Time consuming when operate with directories
  - Must traverse all the list ⇐ Use binary tree?

② Hash table – Hash table with linear list
- Reduce directory traversing time
- Require an effective hash function

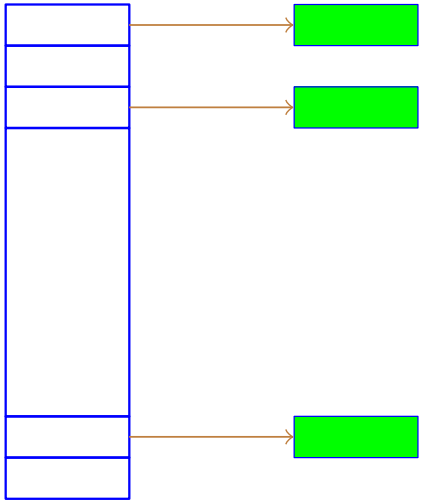
$$h(Name) = \frac{\sum_{i=1}^{Len(Name)} ASCII(Name[i])}{Table\_Size}$$

- Collision problem: hash function return same result with 2 different file name
- Fixed table size problem: Increase size has to recalculate existed numbers

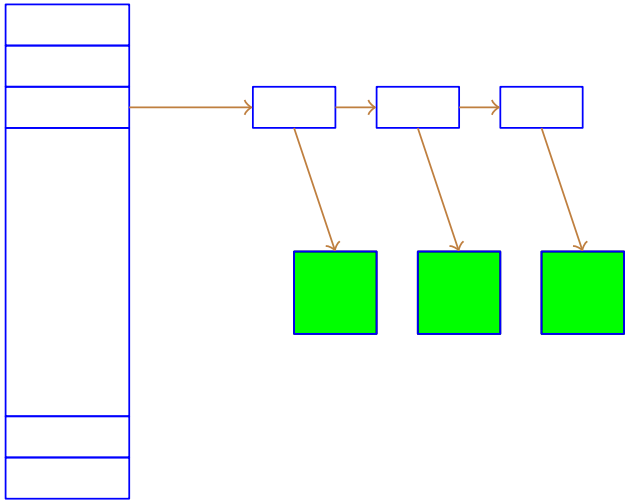
Linear list



Hash table



Hash table



- Directory implementation
- Storage area allocation methods
- Free storage area management

Methods

Objective

- Increase performance of sequence access
- Easy to randomly access file
- Easy to manage file

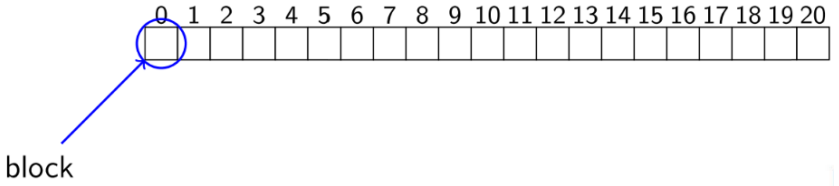
Methods

- Continuous Allocation
- Linked List Allocation
- Indexed Allocation

Continuous Allocation

Rule: File is allocated with continuous memory block

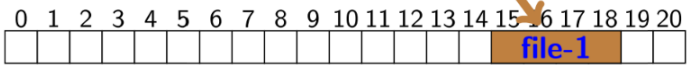
File	*Pos*	*Size*
file-1	15	4
file-2	4	5
file-3	11	3
Directory		



Continuous Allocation

Rule: File is allocated with continuous memory block

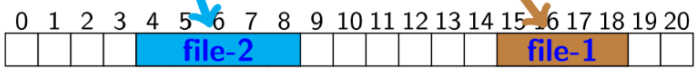
File	*Pos*	*Size*
file-1	15	4
file-2	4	5
file-3	11	3
Directory		



Continuous Allocation

Rule: File is allocated with continuous memory block

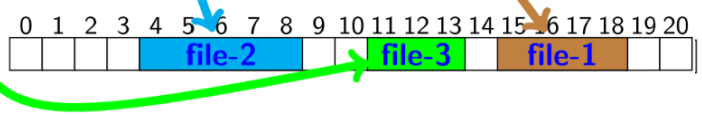
File	*Pos*	*Size*
file-1	15	4
file-2	4	5
file-3	11	3
Directory		



Continuous Allocation

Rule: File is allocated with continuous memory block

File	*Pos*	*Size*
file-1	15	4
file-2	4	5
file-3	11	3
Directory		



Continuous Allocation

- File has the length **n** and begin at block **b** will occupy blocks **b, b + 1, . . . , b + n − 1**
  - Two block **b** and **b + 1** are continuous
    ⇒ No need to move the header (except the last sector)
    ⇒ High speed access
  - Allow directly access block **i** of file
    ⇒ access block b + i − 1 on storage device
- Select empty space when there are request?
  - Strategies First-Fit /Worst Fit /Best Fit
  - External fragmentation phenomenon
- Difficult when the size of file increase

Linked List Allocation

Rule: File is allocated with non continuous memory blocks.
End of each block is a pointer, point to next block

File	*Pos*	*End*
abc	12	3
def	5	11
Directory		

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	3	-1	0	6	8	14	9	11	7	-1	10	0	15	2	0	0

Linked List Allocation

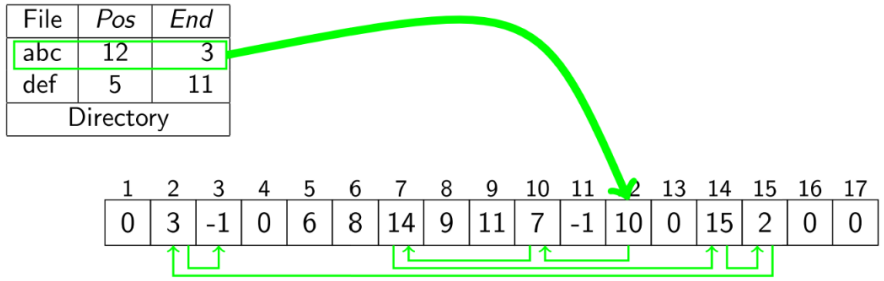
Rule: File is allocated with non continuous memory blocks.
End of each block is a pointer, point to next block

File	*Pos*	*End*
abc	12	3
def	5	11
Directory		

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	3	-1	0	6	8	14	9	11	7	-1	10	0	15	2	0	0

Linked List Allocation

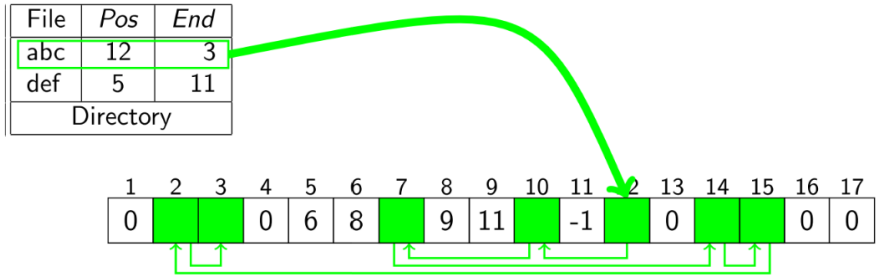
Rule: File is allocated with non continuous memory blocks.
End of each block is a pointer, point to next block



Chapter 4: File system management
2. File system's implementation
2.2 Storage area allocation methods

Linked List Allocation

Rule: File is allocated with non continuous memory blocks.
End of each block is a pointer, point to next block

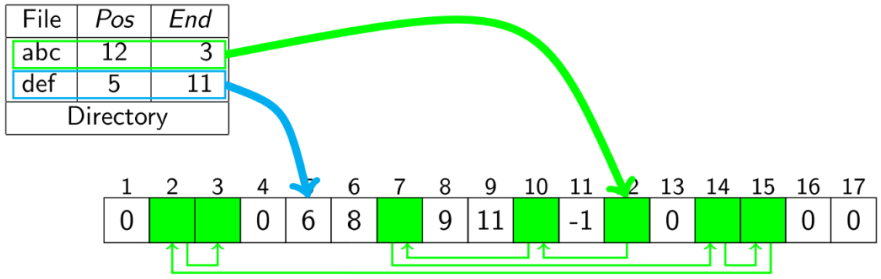


Files abc consists of 7 blocks: 12, 10, 7, 14, 15, 2, 3

Chapter 4: File system management
2. File system's implementation
2.2 Storage area allocation methods

Linked List Allocation

Rule: File is allocated with non continuous memory blocks.
End of each block is a pointer, point to next block

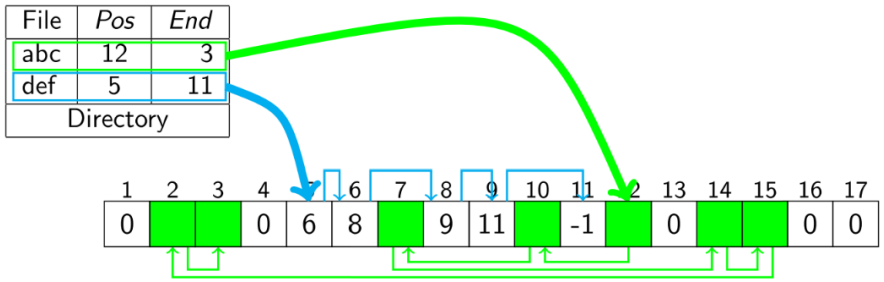


Files abc consists of 7 blocks: 12, 10, 7, 14, 15, 2, 3

Chapter 4: File system management
2. File system's implementation
2.2 Storage area allocation methods

Linked List Allocation

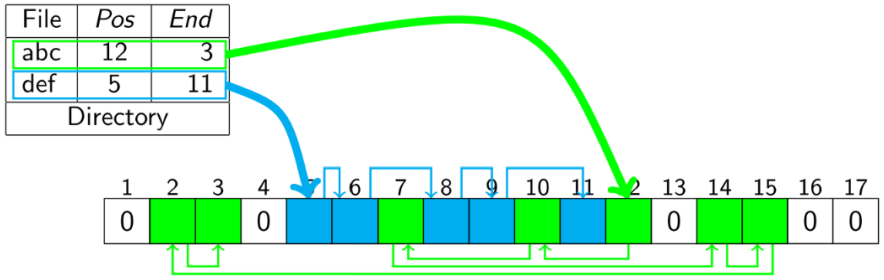
Rule: File is allocated with non continuous memory blocks.
End of each block is a pointer, point to next block



Files abc consists of 7 blocks: 12, 10, 7, 14, 15, 2, 3

Linked List Allocation

Rule: File is allocated with non continuous memory blocks.
End of each block is a pointer, point to next block



Files abc consists of 7 blocks: 12, 10, 7, 14, 15, 2, 3

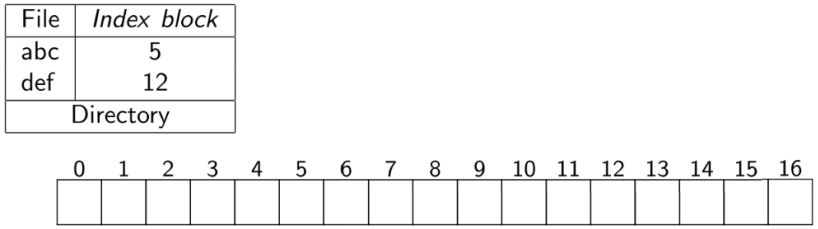
Files def consists of 5 blocks: : 5, 6, 8, 9, 11

Linked List Allocation

- Only effective for sequent access file
- To access block n, must traverse n − 1 blocks before it
  - Blocks are noncontiguous, has to relocation from start
  - Slowly access speed
- Blocks in file are linked by pointers -> if pointer broke?
  - Lost data due to link to block is lost
  - Link to block without data or block belongs to other file
- **Solution:** Apply many pointer in each block ⇒Consume memory
- Apply: FAT
  - Utilized as a linked list
  - Combined of many members, each member corresponding to a block
  - Each member in FAT, contain next block of file
  - Last block has special value (FFFF)
  - Error block has value (FFF7)
  - Unused block has value (0)
  - **Location** field in **file record** contains the first block of file

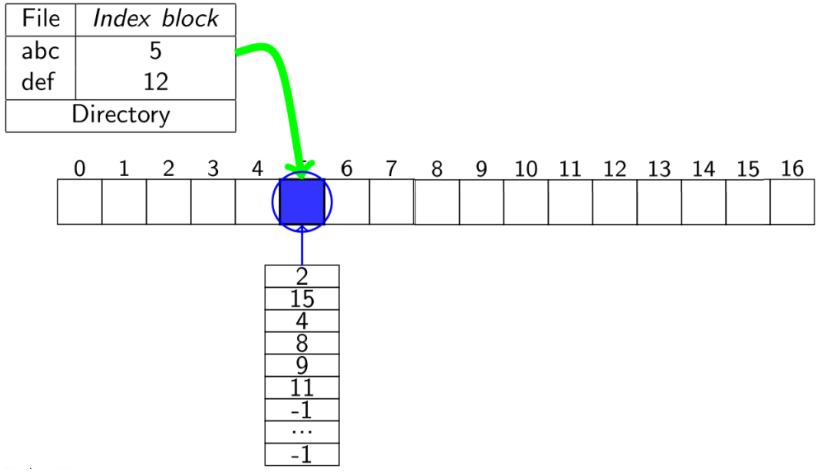
Index allocation

Rule: Each file has a main index block which contains list of file block



Index allocation

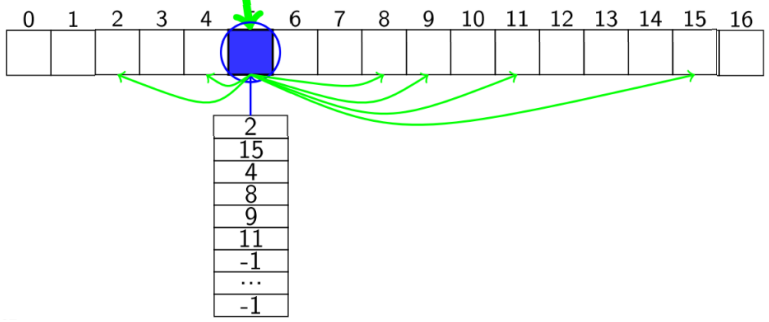
Rule: Each file has a main index block which contains list of file block



Index allocation

Rule: Each file has a main index block which contains list of file block

File	*Index block*
abc	5
def	12
Directory	

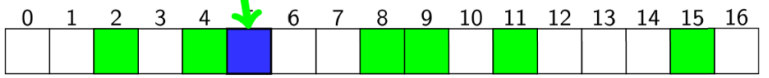


Chapter 4: File system management
2. File system's implementation
2.2 Storage area allocation methods

Index allocation

Rule: Each file has a main index block which contains list of file block

File	*Index block*
abc	5
def	12
Directory	



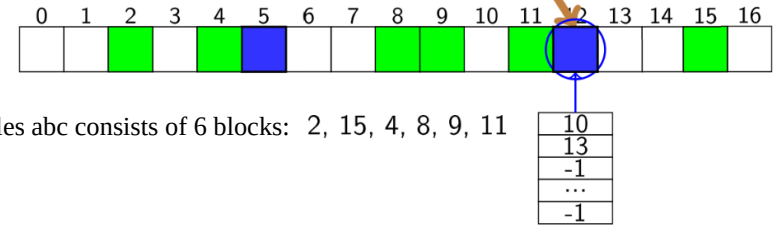
Files abc consists of 6 blocks: 2, 15, 4, 8, 9, 11

Chapter 4: File system management
2. File system's implementation
2.2 Storage area allocation methods

Index allocation

Rule: Each file has a main index block which contains list of file block

File	*Index block*
abc	5
def	12
Directory	



Files abc consists of 6 blocks: 2, 15, 4, 8, 9, 11

Chapter 4: File system management
2. File system's implementation
2.2 Storage area allocation methods

Index allocation

Rule: Each file has a main index block which contains list of file block

File	*Index block*
abc	5
def	12
Directory	

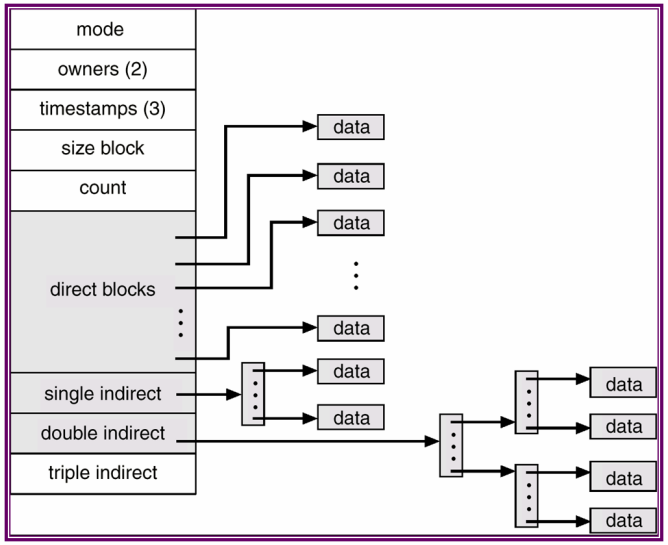
Files abc consists of 6 blocks: 2, 15, 4, 8, 9, 11
Files abc consists of 2 blocks: 10, 13

Index allocation

- Member **i** of index block point to block i of file
  - Read block **i** use pointer declared at member **i** of index block
- Create file, members of index block has null value (-1)
- Require block **i**, block address is allocated, putted into member **i**
- Remark
  - No external fragmentation
  - Allow direct access
  - Require index block: file has small size require 2 blocks
    - Block for data
    - Block for index (use 1 member)

    Solution: Reduce block size ⇒ Reduce cost of memory ⇒ file's size storing problem.
- Linked map
  - Connects index block
  - Last member of index block point to other index block if it's necessary
- Multi level Index
  - Use an index block point to other index block

Link map (UNIX)



- 12 direct block point to data block
- Single indirect block contains address of direct block
- Double indirect block contains address of Single indirect block
- Triple indirect block contain address of Double indirect

- Directory implementation
- Storage area allocation methods
- Free storage area management

Methods

- Bit vector
  - Each block represented by 1 bit (1: free; 0: allocated)
  - Easy to find n contiguous memory block
  - Need instruction to work with bit
- Link list
  - Hold pointer to first free disk block
  - This memory block contain pointer to next free disk block
  - Not effective when traversing the list
- Grouping
  - Hold address of n free block in the first free block
  - n − 1 first free block, block n contain address of next n free block
  - Pros: Fast to find free memory area
- Counting
  - Due to memory block continuously allocated and free concurrently
  - Rule: Store address of the first free block and size of contiguous memory area in free-memory-area management list
  - Effective when the counter larger than 1

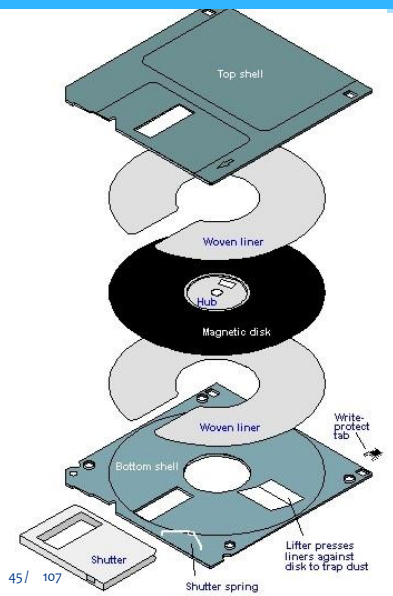
Chapter 4 File system management

1. File system
2. File system's implementation
3. Information organization on disk
4. FAT system

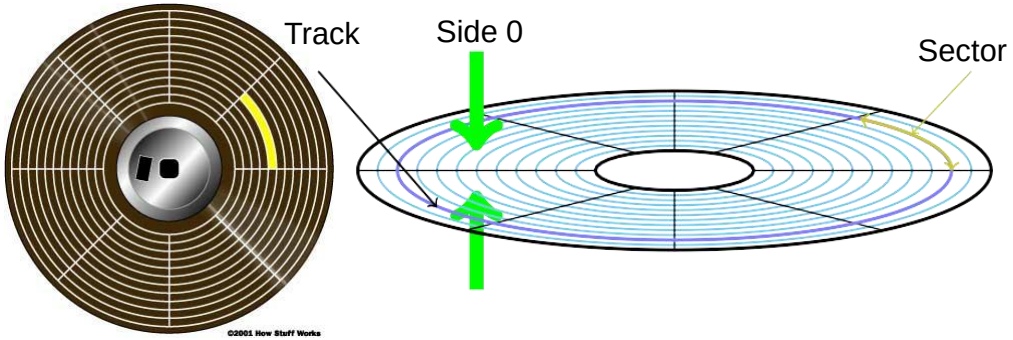
- Disk's physical structure
- Disk's logical structure

Floppy disk $5\frac{1}{4}$

Floppy disk $3\frac{1}{2}$



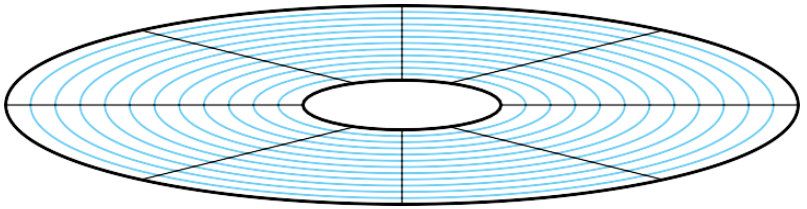
Floppy disk's physical structure



- Track: concentric circles around the disk
  - Numbered 0, 1,. . . From outer to inner
- Side. Each side is read by a Header
  - Headers are numbered 0, 1
- Sector
  - Numbered 1, 2,. . .

Information positioning on floppy disk

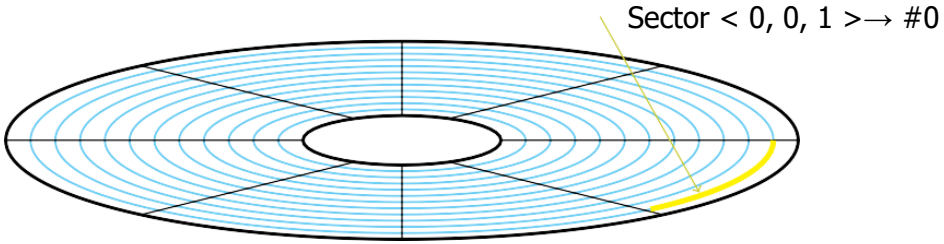
- Sector: information unit for working with disk
- Sector defined via 3 dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
- Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector



Chapter 4: File system management
3. Information organization on disk
3.1 Disk's physical structure

Information positioning on floppy disk

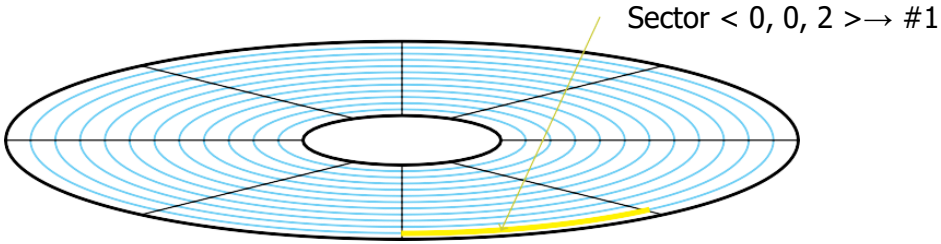
- Sector: information unit for working with disk
- Sector defined via 3 dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
- Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector



Chapter 4: File system management
3. Information organization on disk
3.1 Disk's physical structure

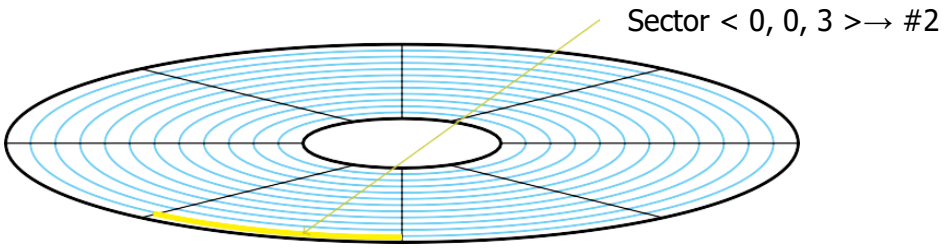
Information positioning on floppy disk

- Sector: information unit for working with disk
- Sector defined via 3 dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
- Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector



Information positioning on floppy disk

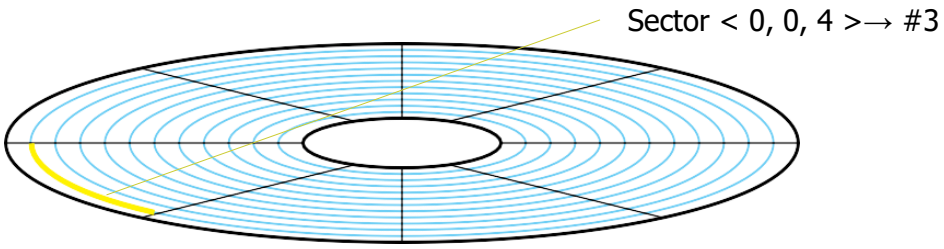
- Sector: information unit for working with disk
- Sector defined via 3 dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
- Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector



Chapter 4: File system management
3. Information organization on disk
3.1 Disk's physical structure

Information positioning on floppy disk

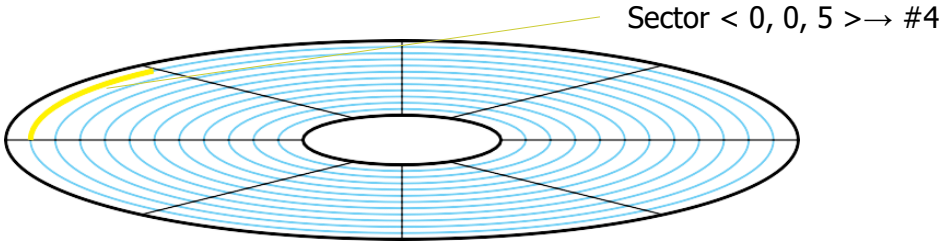
- Sector: information unit for working with disk
- Sector defined via 3 dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
- Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector



Chapter 4: File system management
3. Information organization on disk
3.1 Disk's physical structure

Information positioning on floppy disk

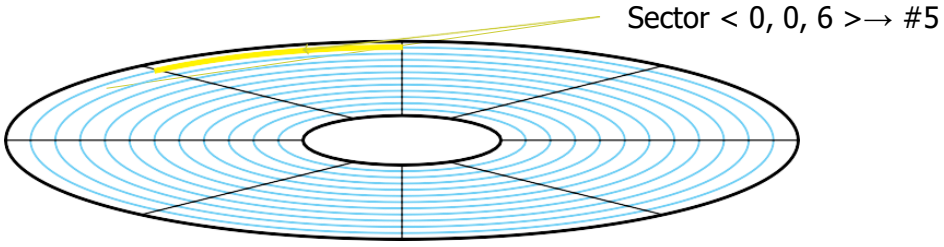
- Sector: information unit for working with disk
- Sector defined via 3 dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
- Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector



Chapter 4: File system management
3. Information organization on disk
3.1 Disk's physical structure

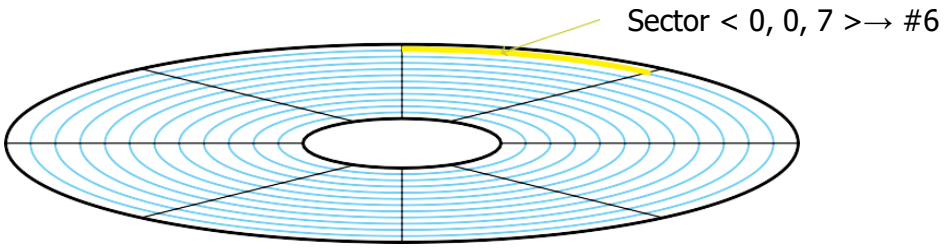
Information positioning on floppy disk

- Sector: information unit for working with disk
- Sector defined via 3 dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
- Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector



Information positioning on floppy disk

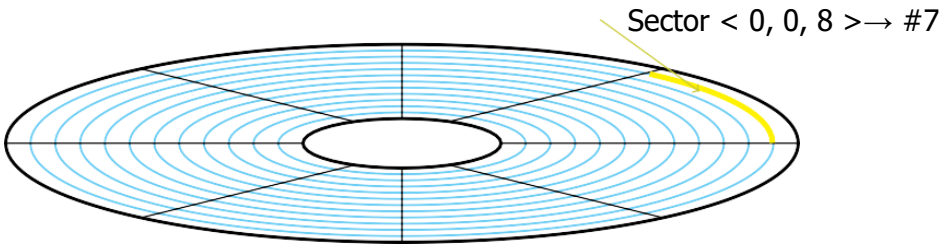
- Sector: information unit for working with disk
- Sector defined via 3 dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
- Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector



Chapter 4: File system management
3. Information organization on disk
3.1 Disk's physical structure

Information positioning on floppy disk

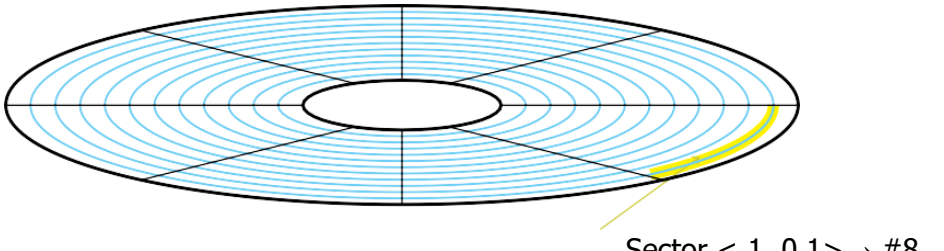
- Sector: information unit for working with disk
- Sector defined via 3 dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
- Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector



Chapter 4: File system management
3. Information organization on disk
3.1 Disk's physical structure

Information positioning on floppy disk

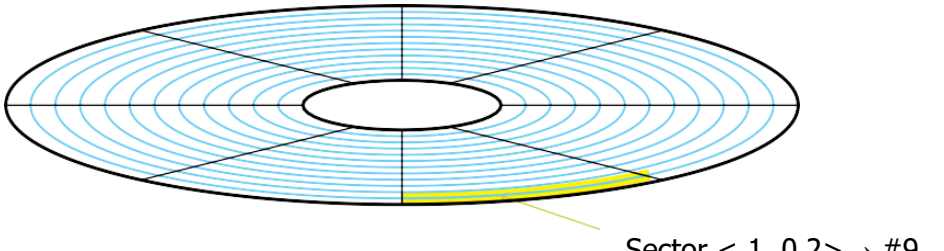
- Sector: information unit for working with disk
- Sector defined via 3 dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
- Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector



Chapter 4: File system management
3. Information organization on disk
3.1 Disk's physical structure

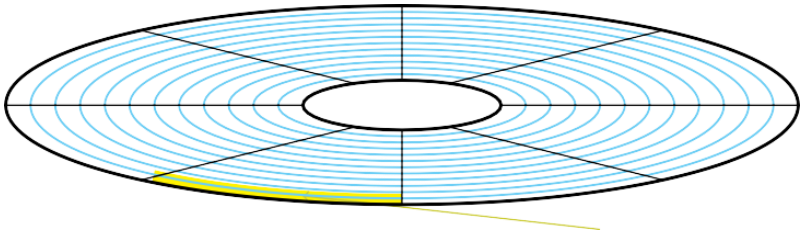
Information positioning on floppy disk

- Sector: information unit for working with disk
- Sector defined via 3 dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
- Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector



Information positioning on floppy disk

- Sector: information unit for working with disk
- Sector defined via 3 dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
- Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector

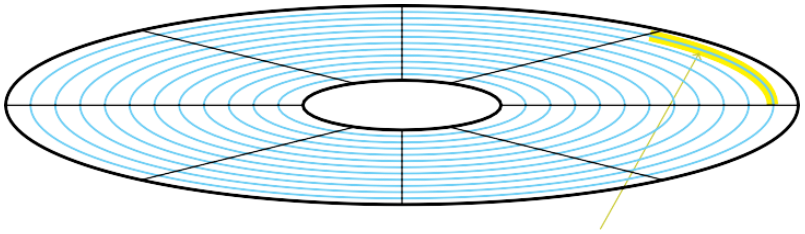


Sector < 1, 0,3>→ #10

Chapter 4: File system management
3. Information organization on disk
3.1 Disk's physical structure

Information positioning on floppy disk

- Sector: information unit for working with disk
- Sector defined via 3 dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
- Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector



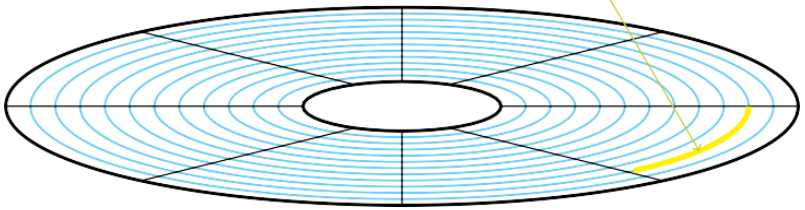
Sector < 1, 0,8>→ #15

Chapter 4: File system management
3. Information organization on disk
3.1 Disk's physical structure

Information positioning on floppy disk

- Sector: information unit for working with disk
- Sector defined via 3 dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
- Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector

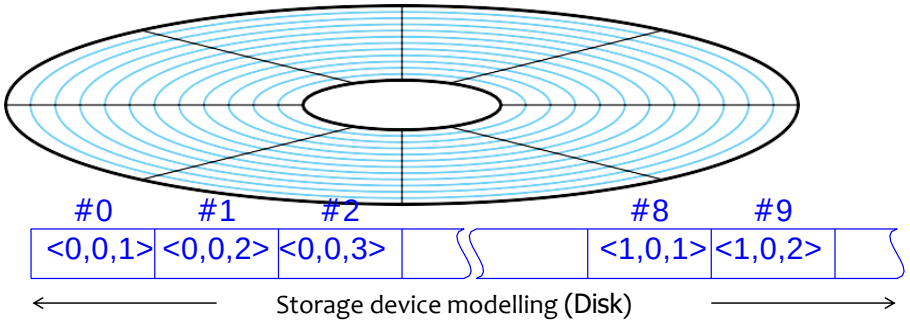
Sector < 0, 1,1>→ #16



Chapter 4: File system management
3. Information organization on disk
3.1 Disk's physical structure

Information positioning on floppy disk

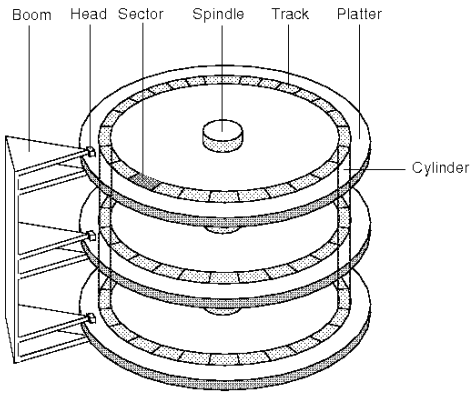
- Sector: information unit for working with disk
- Sector defined via 3 dimension position: Header, Track, Sector
  - Example: Floppy disk's Boot Sector: Sector <0, 0, 1>
- Sector defined via sector number (1 direction position)
  - Relative position from disk's first sector



Hard disk

Hard disk

Hard disk's physical structure



Structure

- Consist of many headers, numbered from 0,1
- Tracks with the same radius create a cylinder, numbered from 0, 1,..
- Sectors on each side of cylinder, numbered from 1,2,...

Information positioning

- 3 dimensions (H, C, S)
- 1 dimension: sector's number
  - Rule similar to floppy disk: Sector→Header→Cylinder

Accessing sector on disk

- Sector: information unit to work with disks
- Can access (read/write/format/...) to each sector
- Access via BIOS 13h interrupt (function 2, 3, 5,...)
  - Not depend on operating system
  - Sector is determined by address <H,C,S>
- Access via system call
  - Operating system's interrupt
    - Example: MSDOS provide 25h/26h interrupt allow read/write sectors at linear address
  - Use WIN32 API functions
    - CreateFile()/ReadFile()/WriteFile()...

Interrupt 13h

Register	Meaning
AH	2h:Read sector; 3h: write Sector
AL	Number of sector to read Sectors must be on the same side, track
DH	Header number
DL	Disk's number. 0h:A; 80h: First hard disk; 81h Second hard disk 2
CH	Track/Cylinder's number (Use 10 bits, borrow 2 high bits from CL)
CL	Sector's number (only use 6 lower bits)
ES:BX	Point to buffer area where data will be read (when AH=2h) or write to disk (when AH=3h)
CarryFlag	CF=0 no error; CL contains number of sector read CF=1 Had error, AH contains error's code

WinXP limit the use on interrupt 13h to direct access

Use interrupt 13h (Example)

```
#include <stdio.h>
#include <dos.h>
int main(int argc, char *argv[]){
      union REGS regs;
      struct SREGS sregs;
      int Buf[512];
      int i;
      regs.h.ah = 0x02;
      regs.h.al = 0x01;
      regs.h.dh = 0x00;
      regs.h.dl = 0x80;
      regs.h.ch = 0x00;
      regs.h.cl = 0x01;
      regs.x.bx = FP_OFF(Buf);
      sregs.es = FP_SEG(Buf);
      int86x(0x13,&regs,&regs,&sregs);
      for(i=0;i<512;i++) printf("%4X",Buf[i]);
      return 0;
}
```


Use WIN32 API

- HANDLE CreateFile(. . .): Open file/IO device
  - LPCTSTR lpFileName, ⇒ file/IO device's name
    - "\\.\C:" Partition / Drive C
    - "\\.\PhysicalDrive0" First hard drive
  - DWORD dwDesiredAccess,⇒ Operation with device
  - DWORD dwShareMode,⇒ Allow sharing
  - LPSECURITY_ATTRIBUTES lpSecurityAttributes (NULL),
  - DWORD dwCreationDisposition,⇒ Operation to perform
  - DWORD dwFlagsAndAttributes, ⇒ Attribute
  - HANDLE hTemplateFile (NULL)
- BOOL ReadFile(. . .)
  - HANDLE hFile,⇒ File to read
  - LPVOID lpBuffer, ⇒ Buffer to store data
  - DWORD nNumberOfBytesToRead,⇒, number of byte to read
  - LPDWORD lpNumberOfBytesRead,⇒ number of read bytes
  - LPOVERLAPPED lpOverlapped (NULL)
- BOOL WriteFile(. . .) ⇒Parameters similar to ReadFile()

Use WIN32 API (Example)

```
#include <windows.h>
#include <stdio.h>
int main(int argc, char *argv[]){
   HANDLE   hDisk;
   BYTE  Buf[512];
   int byteread,i;
   hDisk=CreateFile("\\\\.\\PhysicalDrive0",GENERIC_READ,
                    FILE_SHARE_READ | FILE_SHARE_WRITE,
                    NULL, OPEN_EXISTING,0,NULL);
   if (hDisk==INVALID_HANDLE_VALUE) printf("Device's error");
   else {
       ReadFile(hDisk,Buf,512,&byteread,NULL);
       for(i=0;i<512;i++)      printf("%4X",Buf[i]);
       CloseHandle(hDisk);
   }
   return 0;
}
```

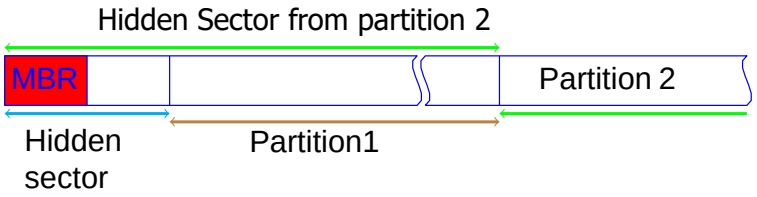
Result

```
33 C0 8E D0 BC 00 7C FB 50 07 50 1F FC BE 1B 7C BF 1B 06 50
57 B9 E5 01 F3 A4 CB BD BE 07 B1 04 38 6E 00 7C 09 75 13 83
C5 10 E2 F4 CD 18 8B F5 83 C6 10 49 74 19 38 2C 74 F6 A0 B5
07 B4 07 8B F0 AC 3C 00 74 FC BB 07 00 B4 0E CD 10 EB F2 88
4E 10 E8 46 00 73 2A FE 46 10 80 7E 04 0B 74 0B 80 7E 04 0C
74 05 A0 B6 07 75 D2 80 46 02 06 83 46 08 06 83 56 0A 00 E8
21 00 73 05 A0 B6 07 EB BC 81 3E FE 7D 55 AA 74 0B 80 7E 10
00 74 C8 A0 B7 07 EB A9 8B FC 1E 57 8B F5 CB BF 05 00 8A 56
00 B4 08 CD 13 72 23 8A C1 24 3F 98 8A DE 8A FC 43 F7 E3 8B
D1 86 D6 B1 06 D2 EE 42 F7 E2 39 56 0A 77 23 72 05 39 46 08
73 1C B8 01 02 BB 00 7C 8B 4E 02 8B 56 00 CD 13 73 51 4F 74
4E 32 E4 8A 56 00 CD 13 EB E4 8A 56 00 60 BB AA 55 B4 41 CD
13 72 36 81 FB 55 AA 75 30 F6 C1 01 74 2B 61 60 6A 00 6A 00
FF 76 0A FF 76 08 6A 00 68 00 7C 6A 01 6A 10 B4 42 8B F4 CD
13 61 61 73 0E 4F 74 0B 32 E4 8A 56 00 CD 13 EB D6 61 F9 C3
49 6E 76 61 6C 69 64 20 70 61 72 74 69 74 69 6F 6E 20 74 61
62 6C 65 00 45 72 72 6F 72 20 6C 6F 61 64 69 6E 67 20 6F 70
65 72 61 74 69 6E 67 20 73 79 73 74 65 6D 00 4D 69 73 73 69
6E 67 20 6F 70 65 72 61 74 69 6E 67 20 73 79 73 74 65 6D 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 2C 44 63
0A 08 0B 08 00 00 80 01 01 00 07 FE FF FF 3F 00 00 00 2C 92
00 02 00 00 C1 FF 0F FE FF FF 31 41 8A 03 0E D3 1D 01 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 55 AA
```


- Disk's physical structure
- Disk's logical structure

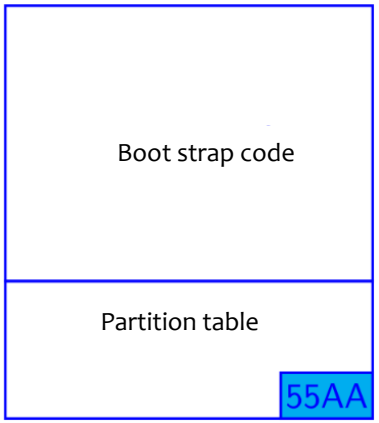
logical structure

- Floppy disk: Each OS has their own management strategy
- Hard disk (mass storage)
  - Divided into sections (Partitions, Volumes,..)
    - Each section is a set of contiguous Cylinders
    - User cab set the size (Example: use fdisk command)
  - Each partition can be manage by an independent OS
    - OS format partition in an usable format
    - Different systems exist: FAT, NTFS, EXT3,...
  - In front of partitions are masked sector
    - Master Boot Record (MBR): Disk's first Sector



Master Boor Record

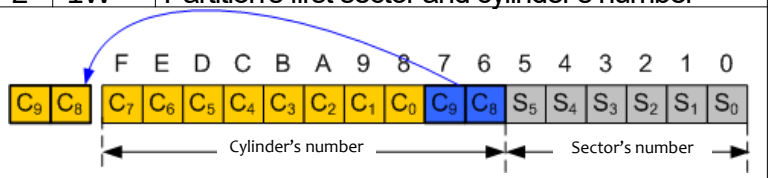
- Disk's most important Sector
- Disk's first sector (Number 0 or address <0, 0, 1>)
- Structure consist of 3 parts



1. Bootstrap code
   - Read partition table to know
     - Position of partitions
     - Active partition (contains OS)
   - Read and execute first sector of active partition
2. Partition table (64bytes)
   - Consist of 4 elements, each element 16 bytes
   - Element contains information of one partition: Position, size, owned system
3. OS's signature (always 55AA)

Structure of one partition table's element

	Stt	Ofs	Size	Meaning
	1	0	1B	Active partition? 80h if yes; 0: Data
Start address	2	1	1B	Partition's first header number
	3	2	1W	Partition's first sector and cylinder's number
	4	4	1B	Recognition code. 05/0F: extended Partition 06:Big Dos; 07:NTFS; 0B: FAT32,..
Last address	5	5	1B	Last header's number
	6	6	1W	Last sector and cylinder 's number *(sector's number only use 6 lower bits)*
	7	8	1DW	Start address, (in sector's numbering)
	8	12	1DW	Number of sectors in partition



Example 1

```
00 01 01  00   07 FE 3F  F8   3F 00  00 00   7A 09  3D 00
80 00 01  F9   0B FE  BF 30   B9 09  3D 00   38  7B 4C 00
00 00 81  EB   0F FE  FF FF   2B 1D  B7 00   72 13 7A  00
00 00 00  00   00 00 00  00   00 00 00  00   00 00 00  00
55 AA
```

Decode

Boot	Start position Hdr	Cyl	Sec	End position HdR	Cyl	Sec	#sector	Number of sector
No	1	0	1	254	248	63	63	4000122
Yes	0	249	1	254	560	63	4000185	5012280
No	0	747	1	254	1023	63	12000555	8000370
-	0	0	0	0	0	0	0	0

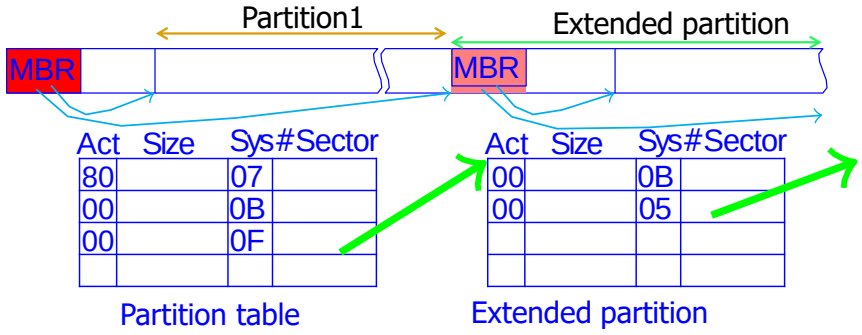
Example 2

```
 80   01   01   00   07   FE   FF   FF   3F   00   00   00   2C   92   00   02
 00   00   C1   FF   0F   FE   FF   FF   31   41   8A   03   0E   D3   1D   01
 00   00   00   00   00   00   00   00   00   00   00   00   00   00   00   00
 00   00   00   00   00   00   00   00   00   00   00   00   00   00   00   00
 55   AA
```

Active	Begin Hdr	Cyl	Sct	Sys	End Hdr	Cyl	Sct	Relative Sector	Number Of Sector
YES	1	0(0)	1	07	254	1023(2090)	63	63	33591852
NO	0	1023(3697)	1	0F	254	1023(4862)	63	59392305	18731790
NO	0	0(0)	0	00	0	0	0	0	0
NO	0	0(0)	0	00	0	0	0	0	0

Extended partition

- When recognition code is 05 or 0F -> extended partition
- Organized as a physical hard disk
  - First Sector is MBR, contain information of partitions inside this extended partition
    - Element in extended partition can be an extended partition
  - Allow to create more than 4 logic drive



Example of extended partition 1

```
 80   01   01   00   07   EF   FF   FF   3F   00   00   00   11   2F   F7   01
 00   00   C1   FF   0F   EF   FF   FF   50   2F   F7   01   B0   23   B1   02
 00   00   00   00   00   00   00   00   00   00   00   00   00   00   00   00
 00   00   00   00   00   00   00   00   00   00   00   00   00   00   00   00
 55   AA
```

Active	Begin Hdr	Begin Cyl	Begin Sct	Sys	End Hdr	End Cyl	End Sct	Relative Sector	Number Of Sector
YES	1	0	1	07	239	1023	63	63	32976657
NO	0	1023	1	0F	239	1023	63	32976720	45163440
NO	0	0	0	00	0	0	0	0	0
NO	0	0	0	00	0	0	0	0	0

Example of extended partition 2

```
Extended Partition (Sector number 32976720)...
 00   01   C1   FF   06   EF   FF   FF   3F   00   00   00   51   E8   76   01
 00   00   C1   FF   05   EF   FF   FF   90   E8   76   01   20   3B   3A   01
 00   00   00   00   00   00   00   00   00   00   00   00   00   00   00   00
 00   00   00   00   00   00   00   00   00   00   00   00   00   00   00   00
 55   AA
```

Active	Begin Hdr	Begin Cyl	Begin Sct	Sys	End Hdr	End Cyl	End Sct	Relative Sector	Number Of Sector
NO	1	1023	1	06	239	1023	63	63	24569937
NO	0	1023	1	05	239	1023	63	24570000	20593440
NO	0	0	0	00	0	0	0	0	0
NO	0	0	0	00	0	0	0	0	0

Example of extended partition 3

```
Extended Partition (Sector number 57546720)...
 00   01   C1   FF   0B   EF   FF   FF   3F   00   00   00   E1   3A   3A   01
 00   00   00   00   00   00   00   00   00   00   00   00   00   00   00   00
 00   00   00   00   00   00   00   00   00   00   00   00   00   00   00   00
 00   00   00   00   00   00   00   00   00   00   00   00   00   00   00   00
 55   AA
```

Active	Begin Hdr	Begin Cyl	Begin Sct	Sys	End Hdr	End Cyl	End Sct	Relative Sector	Number Of Sector
NO	1	1023	1	0B	239	1023	63	63	20593377
NO	0	0	0	00	0	0	0	0	0
NO	0	0	0	00	0	0	0	0	0
NO	0	0	0	00	0	0	0	0	0

Chapter 4 File system management

File systems

Many file systems existed

- FAT system
  - FAT 12/ FAT16 for MSDOS
  - FAT32 from WIN98
  - 12/16/32: Number of bit to identify a cluster
- NTFS
  - Used in WINNT, WIN2000
  - Utilize 64 bits to identify a cluster
  - Better than FAT in security, encoding, compress data,...
- EXT3
  - Used in Linux
- CDFS
  - File system used for CDROM
  - Limited depth in directory tree and size of names
- UDF
  - Developed from CDFS for DVD-ROM, support long file name

- Boot sector
- FAT (File Allocation Table)
- Root directory

Partitioning structure for FAT

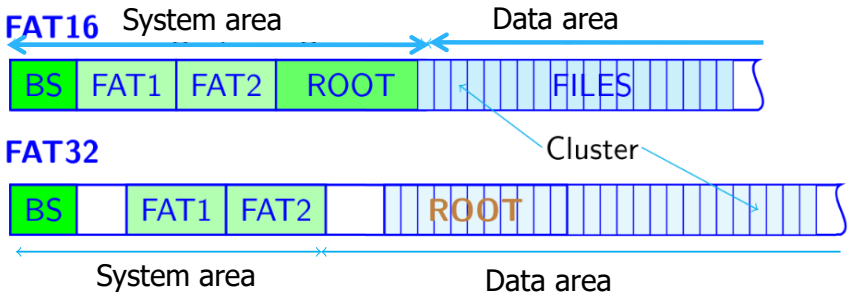
FAT12/16

- Maximum number of cluster FAT12: $2^{12} - 18$; FAT16 : $2^{16} - 18$
- Max size: FAT12: 32MB; FAT16: 2GB/4GB (32K/64K Cluster)

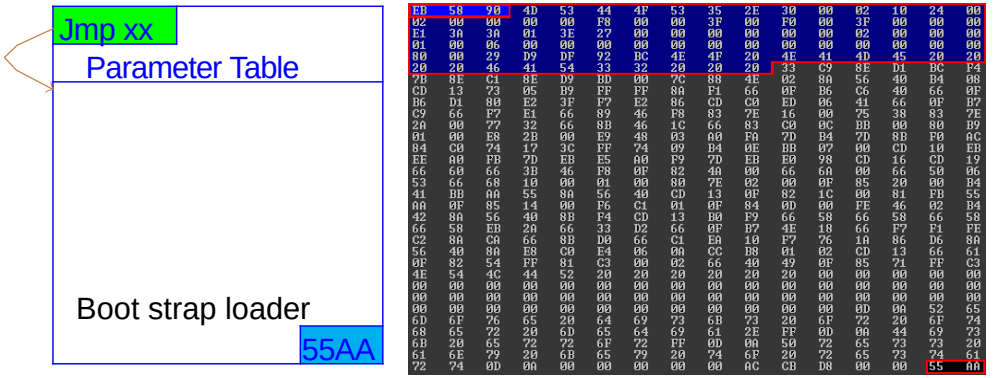
FAT32

- Only use 28 bits ⇒ Maximum Number of cluster: $2^{28} - 18$
- Max size: 2TB/8GB/16TB (8KB/32KB/64KB Cluster)

FAT's logical structure



Structure



- Partition's First Sector
- Structure consist of 3 parts
  - BPB: Bios Parameter Block
  - Boot strap loader
  - System's signature (always 55AA)

Bios Parameter Block's structure – Common part

Stt	Ofs	Size	Sample value	Meaning
1	0	3B	EB 3C 90	Jump to begin position of boot strap loader
2	3	8B	MSDOS5.0	Name of system used formatted the disk
3	11	1W	00 02	Size of 1 sector, normally 512
4	13	1B	40	Number of sectors for 1 cluster (32K-Cluster)
5	14	1W	01 00	Num of scts before FAT/Num of reserved scts
6	16	1B	02	Num of FAT
7	17	1W	00 02	Num of ROOT's element. FAT32: 00 00
8	19	1W	00 00	Total sector on disk (< 32M) or 0000
9	21	1B	F8	Disk format (F8:HD, F0: Disk 1.44M)
10	22	1W	D1 00	Num of sector for one FAT(209)
11	24	1W	3F 00	Num of sector for one track (63)
12	26	1W	40 00	Num of headers (64)
13	28	1DW	3F 00 00 00	Num of hidden sector – Sectors before volume (63)
14	32	1DW	41 0C 34 00	Total sector on disk (3411009)

Bios Parameter Block's structure – Part for FAT12/FAT16 only

Stt	Ofs	Kt	Sample value	Meaning
15	36	1B	80h	Physical disk num 0: A drive; 80h: C drive
16	37	1B	00	Reserved, high byte is for disk number
17	38	1B	29h	Boot sector extension 29h
18	39	1DW	D513 5B24	Volumn Serial number(245B-13D5)
19	43	11B	NO NAME	Volume's Label
20	54	8B	FAT16	Reserved, normally for FAT description's text
21	62	-		Bootstrap loader

Example:

```
EB 3C 90 4D 53 44 4F 53 35 2E 30 00 02 02 06 00
02 00 02 00 00 F8 F5 00 3F 00 FF 00 3F 00 00 00
C1 EB 01 00 00 00 29 A6 EA D4 70 4E 4F 20 4E 41
4D 45 20 20 20 20 46 41 54 31 36 20 20 20 33 C9
```


FAT16's disk parameter decoding example

```
EB 3C 90 4D 53 44 4F 53 35 2E 30 00 02 02 06 00
02 00 02 00 00 F8 F5 00 3F 00 FF 00 3F 00 00 00
C1 EB 01 00 00 00 29 A6 EA D4 70 4E 4F 20 4E 41
4D 45 20 20 20 20 46 41 54 31 36 20 20 20 33 C9
```

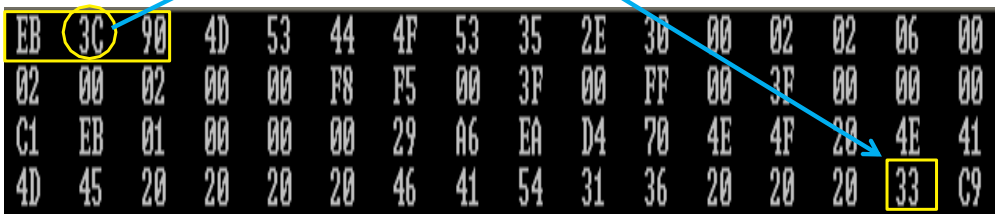

FAT16's disk parameter decoding example

First 3 Bytes đầu: Jump to the start of boot strap loader

```
EB 3C 90 4D 53 44 4F 53 35 2E 30 00 02 02 06 00
02 00 02 00 00 F8 F5 00 3F 00 FF 00 3F 00 00 00
C1 EB 01 00 00 00 29 A6 EA D4 70 4E 4F 20 4E 41
4D 45 20 20 20 20 46 41 54 31 36 20 20 20 33 C9
```

FAT16's disk parameter decoding example

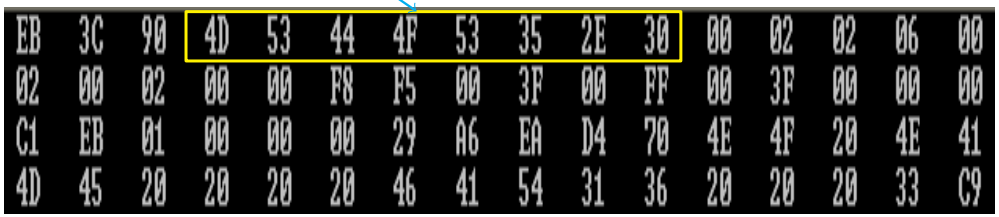
Jmp+3C



FAT16's disk parameter decoding example

8 Bytes
Name of the system utilized to format disk

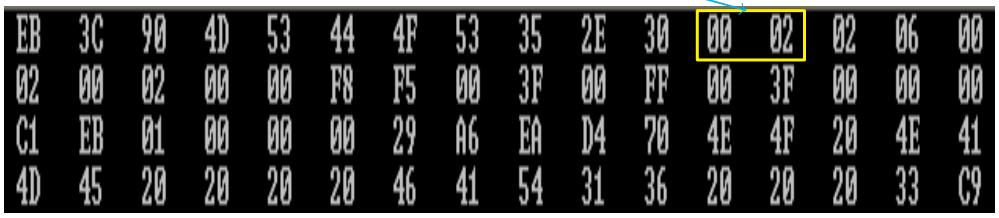
OEName: MSDOS5.0



FAT16's disk parameter decoding example

1 Word
Size of 1 sector

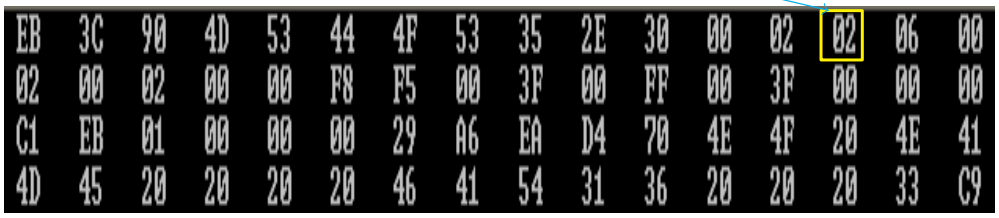
Sector's size: 512



FAT16's disk parameter decoding example

Number of sector for 1 cluster

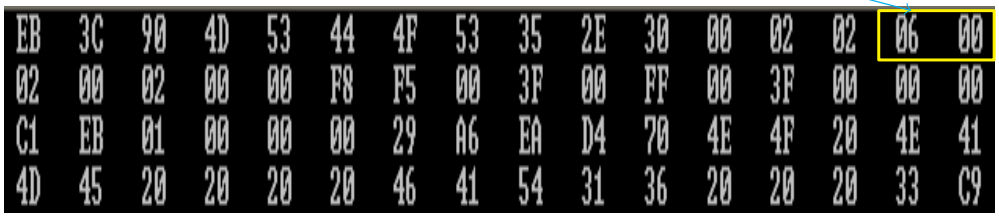
2 sector for 1 cluster



FAT16's disk parameter decoding example

Number of sector before FAT

6 sectors before the first FAT

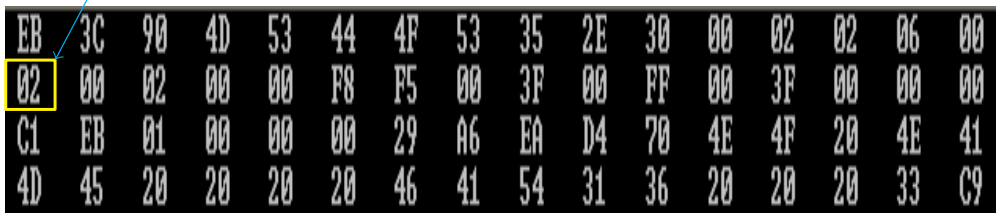


EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	02	06	00
02	00	02	00	00	F8	F5	00	3F	00	FF	00	3F	00	00	00
C1	EB	01	00	00	00	29	A6	EA	D4	70	4E	4F	20	4E	41
4D	45	20	20	20	20	46	41	54	31	36	20	20	20	33	C9

Chapter 4: File system management
4. FAT system
4.1 Boot sector

FAT16's disk parameter decoding example

There are 2 FAT



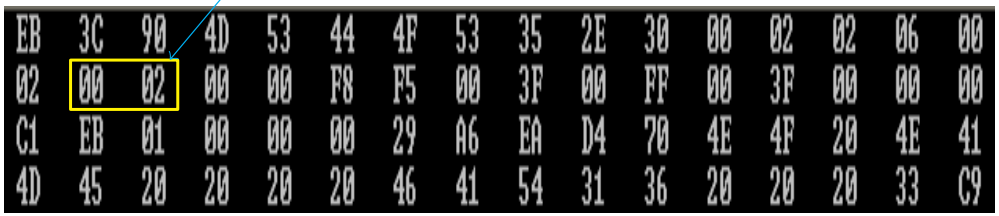
EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	02	06	00
02	00	02	00	00	F8	F5	00	3F	00	FF	00	3F	00	00	00
C1	EB	01	00	00	00	29	A6	EA	D4	70	4E	4F	20	4E	41
4D	45	20	20	20	20	46	41	54	31	36	20	20	20	33	C9

Chapter 4: File system management
4. FAT system
4.1 Boot sector

FAT16's disk parameter decoding example

Number of ROOT's elements

There are maximum 512 elements in Root's directory



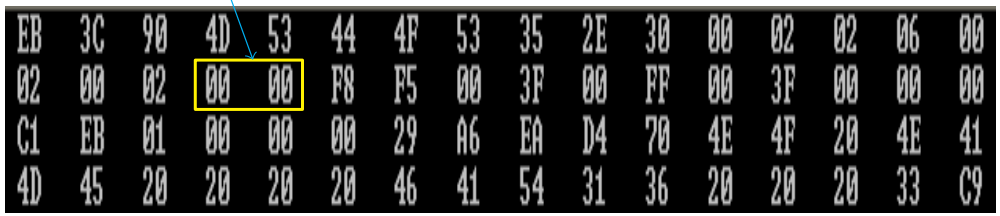
EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	02	06	00
02	00	02	00	00	F8	F5	00	3F	00	FF	00	3F	00	00	00
C1	EB	01	00	00	00	29	A6	EA	D4	70	4E	4F	20	4E	41
4D	45	20	20	20	20	46	41	54	31	36	20	20	20	33	C9

Chapter 4: File system management
4. FAT system
4.1 Boot sector

FAT16's disk parameter decoding example

Total sector on disk

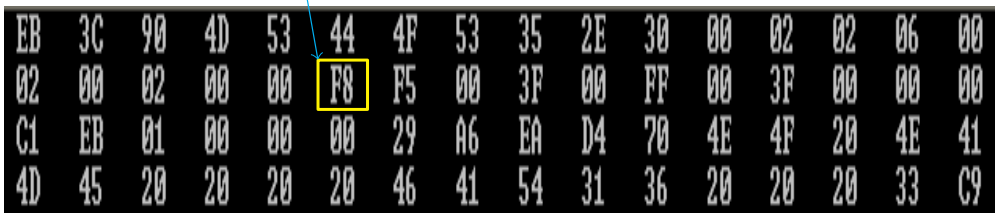
Disk larger than 32MB



EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	02	06	00
02	00	02	00	00	F8	F5	00	3F	00	FF	00	3F	00	00	00
C1	EB	01	00	00	00	29	A6	EA	D4	70	4E	4F	20	4E	41
4D	45	20	20	20	20	46	41	54	31	36	20	20	20	33	C9

FAT16's disk parameter decoding example

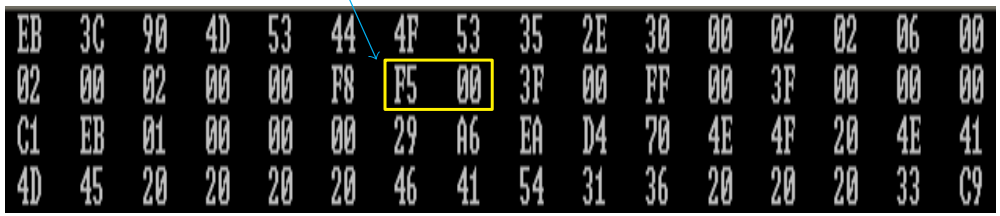
Disk format code: F8



EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	02	06	00
02	00	02	00	00	F8	F5	00	3F	00	FF	00	3F	00	00	00
C1	EB	01	00	00	00	29	A6	EA	D4	70	4E	4F	20	4E	41
4D	45	20	20	20	20	46	41	54	31	36	20	20	20	33	C9

Chapter 4: File system management
4. FAT system
4.1 Boot sector
FAT16's disk parameter decoding example

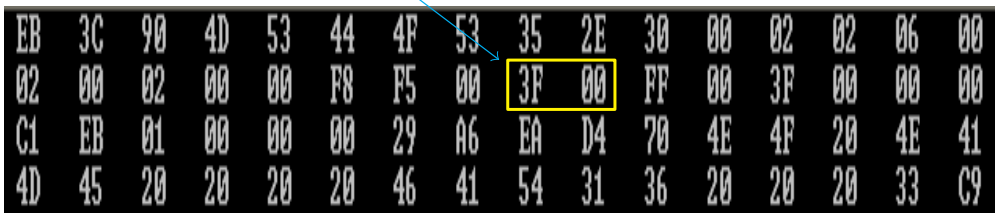
Number of sector for 1 FAT:245



EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	02	06	00
02	00	02	00	00	F8	F5	00	3F	00	FF	00	3F	00	00	00
C1	EB	01	00	00	00	29	A6	EA	D4	70	4E	4F	20	4E	41
4D	45	20	20	20	20	46	41	54	31	36	20	20	20	33	C9

Chapter 4: File system management
4. FAT system
4.1 Boot sector
FAT16's disk parameter decoding example

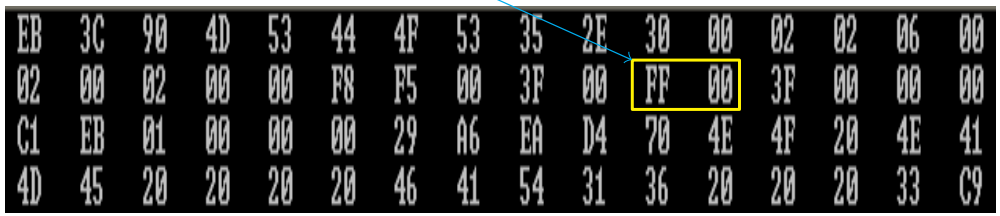
Number of sector for 1 track: 63



EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	02	06	00
02	00	02	00	00	F8	F5	00	3F	00	FF	00	3F	00	00	00
C1	EB	01	00	00	00	29	A6	EA	D4	70	4E	4F	20	4E	41
4D	45	20	20	20	20	46	41	54	31	36	20	20	20	33	C9

Chapter 4: File system management
4. FAT system
4.1 Boot sector
FAT16's disk parameter decoding example

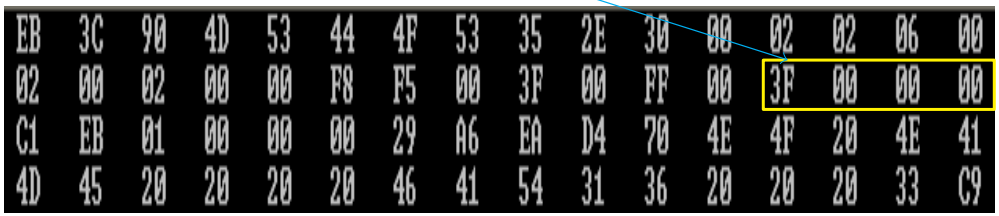
Number of headers: 255



EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	02	06	00
02	00	02	00	00	F8	F5	00	3F	00	FF	00	3F	00	00	00
C1	EB	01	00	00	00	29	A6	EA	D4	70	4E	4F	20	4E	41
4D	45	20	20	20	20	46	41	54	31	36	20	20	20	33	C9

FAT16's disk parameter decoding example

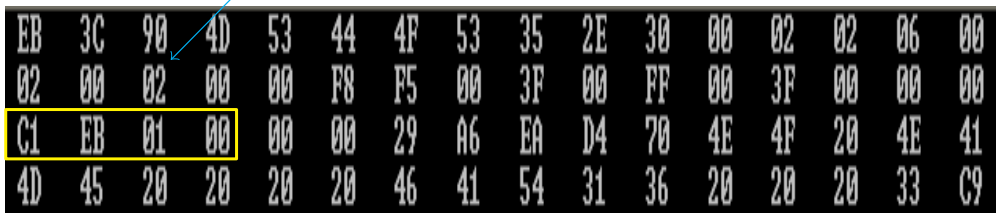
Number of hidden sectors: 63



EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	02	06	00
02	00	02	00	00	F8	F5	00	3F	00	FF	00	3F	00	00	00
C1	EB	01	00	00	00	29	A6	EA	D4	70	4E	4F	20	4E	41
4D	45	20	20	20	20	46	41	54	31	36	20	20	20	33	C9

Chapter 4: File system management
4. FAT system
4.1 Boot sector
FAT16's disk parameter decoding example

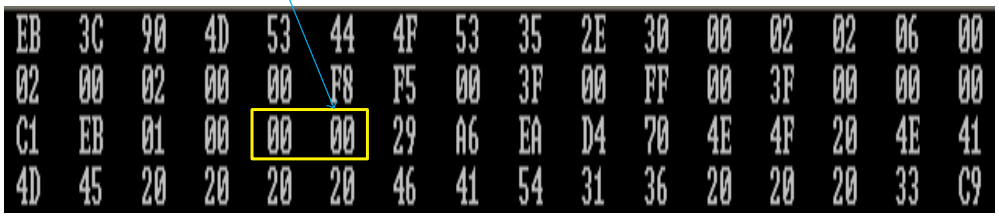
Total sector of Volume: 125889(≈64MB)



EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	02	06	00
02	00	02	00	00	F8	F5	00	3F	00	FF	00	3F	00	00	00
C1	EB	01	00	00	00	29	A6	EA	D4	70	4E	4F	20	4E	41
4D	45	20	20	20	20	46	41	54	31	36	20	20	20	33	C9

Chapter 4: File system management
4. FAT system
4.1 Boot sector
FAT16's disk parameter decoding example

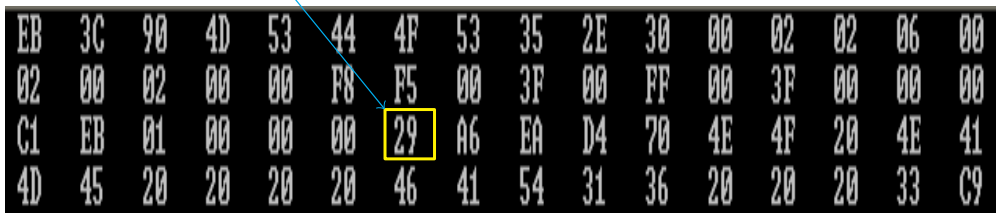
Physical disk number: 00 00



EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	02	06	00
02	00	02	00	00	F8	F5	00	3F	00	FF	00	3F	00	00	00
C1	EB	01	00	00	00	29	A6	EA	D4	70	4E	4F	20	4E	41
4D	45	20	20	20	20	46	41	54	31	36	20	20	20	33	C9

Chapter 4: File system management
4. FAT system
4.1 Boot sector
FAT16's disk parameter decoding example

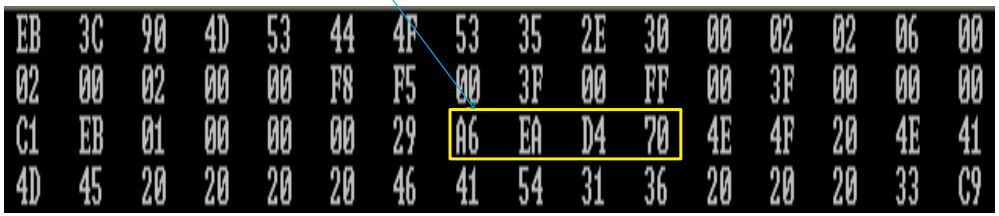
Bootsector extension: 29h



EB	3C	90	4D	53	44	4F	53	35	2E	30	00	02	02	06	00
02	00	02	00	00	F8	F5	00	3F	00	FF	00	3F	00	00	00
C1	EB	01	00	00	00	29	A6	EA	D4	70	4E	4F	20	4E	41
4D	45	20	20	20	20	46	41	54	31	36	20	20	20	33	C9

FAT16's disk parameter decoding example

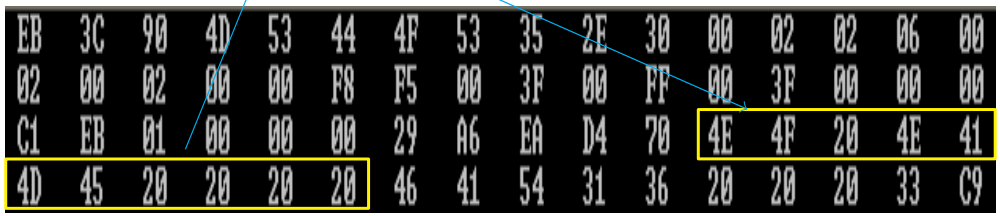
Volume serial number: 70D4-EAA6



Chapter 4: File system management
4. FAT system
4.1 Boot sector

FAT16's disk parameter decoding example

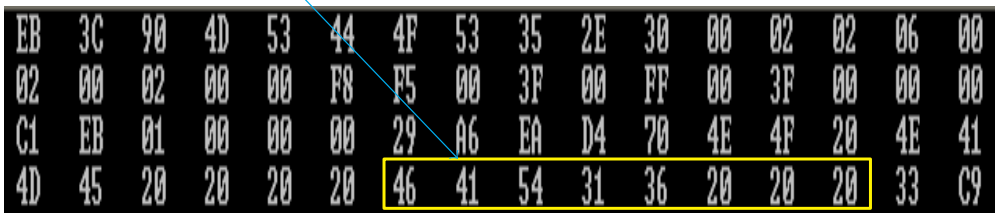
Disk label: NO NAME



Chapter 4: File system management
4. FAT system
4.1 Boot sector

FAT16's disk parameter decoding example

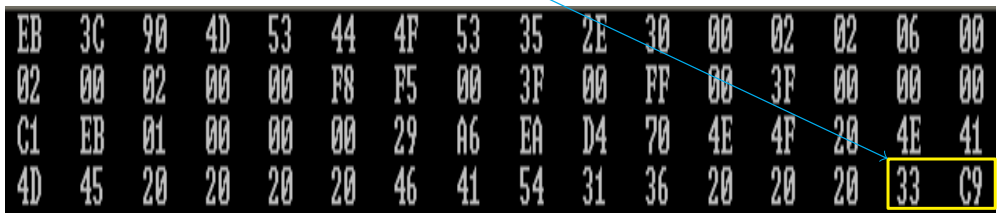
FAT Type: FAT16



Chapter 4: File system management
4. FAT system
4.1 Boot sector

FAT16's disk parameter decoding example

Start of boot strap loader



Bios Parameter Block's structure – Part for FAT32

Stt	Ofs	Size	Sample value	Meaning
15	36	1DW	C9 03 00 00	Total sector for FAT
16	40	1W	00 00	Flags: main #FAT (not used)
17	42	1W	00 00	Version: FAT32 (*not used*)
18	44	1DW	02 00 00 00	cluster starting number of ROOT
19	48	1W	01 00	#sector contain File System information
20	50	1W	06 00	Number of sector to backup Bootsector
21	52	12B	00 . . . 00	Reserved
22	64	1B	00	Physical disk number 0: drive A; 80h: drive C
23	65	1B	00	Reserved/High Byte for #Driver
24	66	1B	29	Boot sector extension. Always 29h
25	67	1DW	62 0E 18 66	Volumn Serial number
26	71	11B	NO NAME	Volumn Label: (*not used*)
27	82	8B	FAT32	Reserved, for FAT's description text

FAT32 Boot sector example

```
EB 58 90 4D 53 44 4F 53 35 2E 30 00 02 10 24 00
02 00 00 00 00 F8 00 00 3F 00 F0 00 3F 00 00 00
E1 3A 3A 01 3E 27 00 00 00 00 00 00 02 00 00 00
01 00 06 00 00 00 00 00 00 00 00 00 00 00 00 00
80 00 29 D9 DF 92 BC 4E 4F 20 4E 41 4D 45 20 20
20 20 46 41 54 33 32 20 20 20 33 C9 8E D1 BC F4
7B 8E C1 8E D9 BD 00 7C 88 4E 02 8A 56 40 B4 08
CD 13 73 05 B9 FF FF 8A F1 66 0F B6 C6 40 66 0F
B6 D1 80 E2 3F F7 E2 86 CD C0 ED 06 41 66 0F B7
C9 66 F7 E1 66 89 46 F8 83 7E 16 00 75 38 83 7E
2A 00 77 32 66 8B 46 1C 66 83 C0 0C BB 00 80 B9
01 00 E8 2B 00 E9 48 03 A0 FA 7D B4 7D 8B F0 AC
84 C0 74 17 3C FF 74 09 B4 0E BB 07 00 CD 10 EB
EE A0 FB 7D EB E5 A0 F9 7D EB E0 98 CD 16 CD 19
66 60 66 3B 46 F8 0F 82 4A 00 66 6A 00 66 50 06
53 66 68 10 00 01 00 80 7E 02 00 0F 85 20 00 B4
41 BB AA 55 8A 56 40 CD 13 0F 82 1C 00 81 FB 55
AA 0F 85 14 00 F6 C1 01 0F 84 0D 00 FE 46 02 B4
42 8A 56 40 8B F4 CD 13 B0 F9 66 58 66 58 66 58
66 58 EB 2A 66 33 D2 66 0F B7 4E 18 66 F7 F1 FE
C2 8A CA 66 8B D0 66 C1 EA 10 F7 76 1A 86 D6 8A
56 40 8A E8 C0 E4 06 0A CC B8 01 02 CD 13 66 61
0F 82 54 FF 81 C3 00 02 66 40 49 0F 85 71 FF C3
4E 54 4C 44 52 20 20 20 20 20 20 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 0D 0A 52 65
6D 6F 76 65 20 64 69 73 6B 73 20 6F 72 20 6F 74
68 65 72 20 6D 65 64 69 61 2E FF 0D 0A 44 69 73
6B 20 65 72 72 6F 72 FF 0D 0A 50 72 65 73 73 20
61 6E 79 20 6B 65 79 20 74 6F 20 72 65 73 74 61
72 74 0D 0A 00 00 00 00 00 AC CB D8 00 00 55 AA
```


FAT32 boot sector decoding result

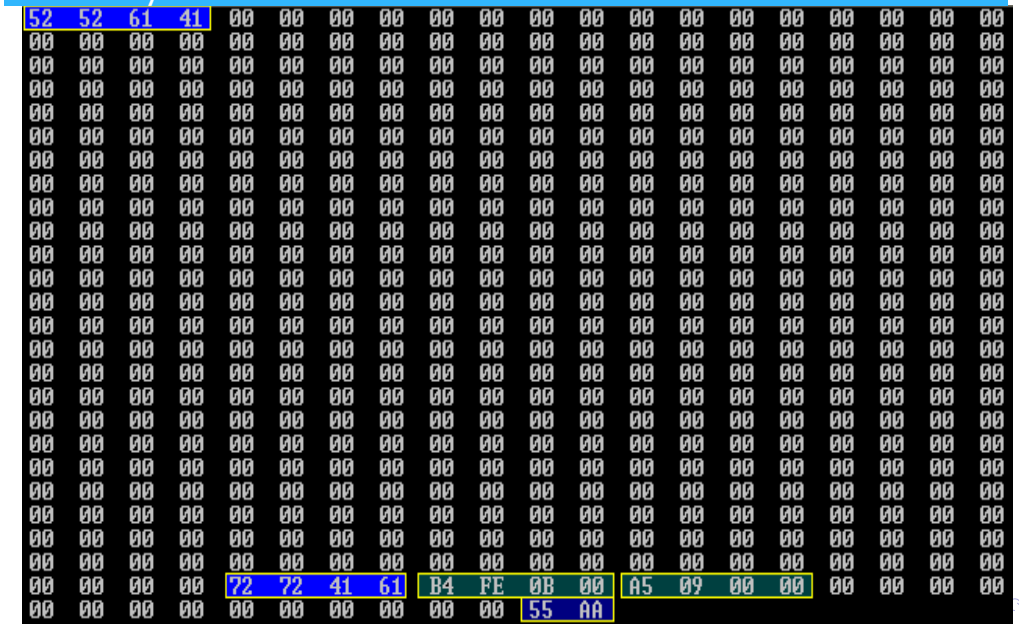
```
BIOS PARAMETER BLOCK (BPB)...
OEM Name                            :   MSDOS5.0
Bytes per sector                    :           512
Sectors per cluster                 :            16
Sectorss before the first FAT       :            36
Number of copies of FAT             :             2
Media Desctiptor                    :           F8h
Sectors per Tracks                  :            63
Number of Header                    :           240
Number of Hiden Scts in Volume      :            63
Number of Sectors in Volume         :      20593377
Number of Sectors per FAT           :         10046
Cluster num. of start of ROOT       :             2
Sct number of FileSystem Info       :             1
Sct number of Boot backup sct       :             6
Logical drive number of Volume      :           80h
Extend BPB Signature                :           29h
Serial Number of Volume             :     BC92-DFD9
Volumn lable                        :       NO NAME
FAT Type                            :         FAT32
Boot signature                      :         55 AA
```


File System Information Sector

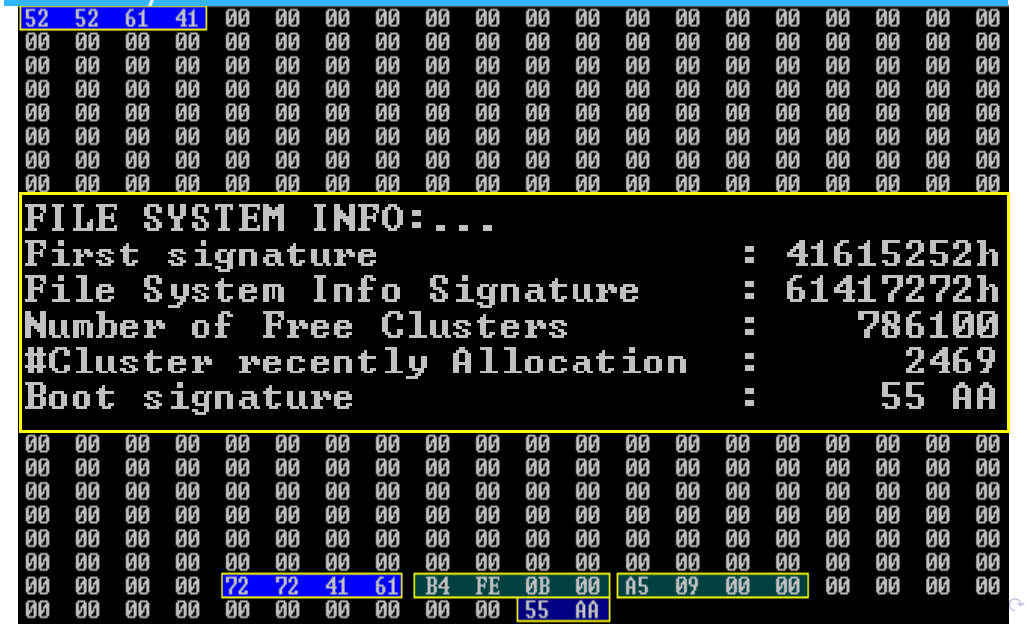
- Usually Sector # 2 of Volume
  - Right after Boot sector (Sector # 1)
- Structure

Stt	Ofs	Size	Meaning
1	0	1DW	First signature of FSInfo sector. Bytes' values in order: 52h 52h 61h 41h
2	4	480B	Not used, usually contain value 00
3	484	1DW	Signature of File System Information Sector. Bytes values in order: 72h 72h 41h 61h
4	488	1DW	Number of free cluster. *-1* if not defined
5	492	1DW	Number of cluster just provided
6	496	12B	Reserved
7	508	2B	Not defined, usually 0
8	510	2B	Bootsector's signature. Contain value 55 AA

File system information sector of 1volume use FAT32



File system information sector of 1 volume use FAT32



- Boot sector
- FAT (File Allocation Table)
- Root directory

Object

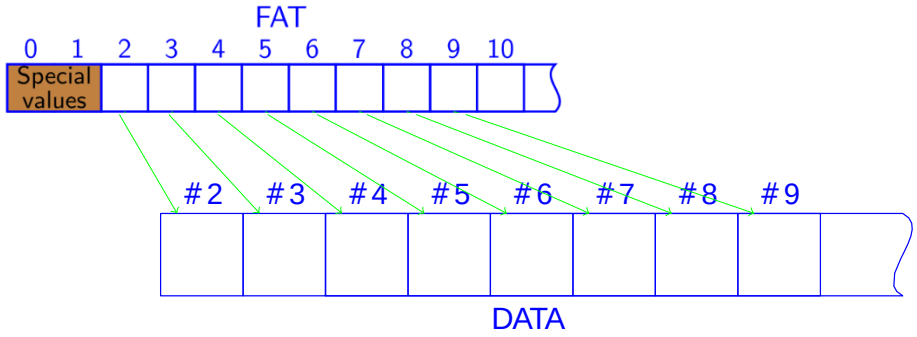
FAT utilized to manage memory blocks/clusters in the data area of storage memory

- Using block
  - Allocated to each file/directory
- Free block
- Error block

Method

FAT include many elements

- Each element can be 12bit, 16bit, 32bit
- Each element corresponding to 1 block (cluster) on the data area
  - First 2 elements (0,1) has special meaning
    - Disk format, Bit shutdown, Bit diskerrror
  - Element # 2 corresponding to first cluster of the Data



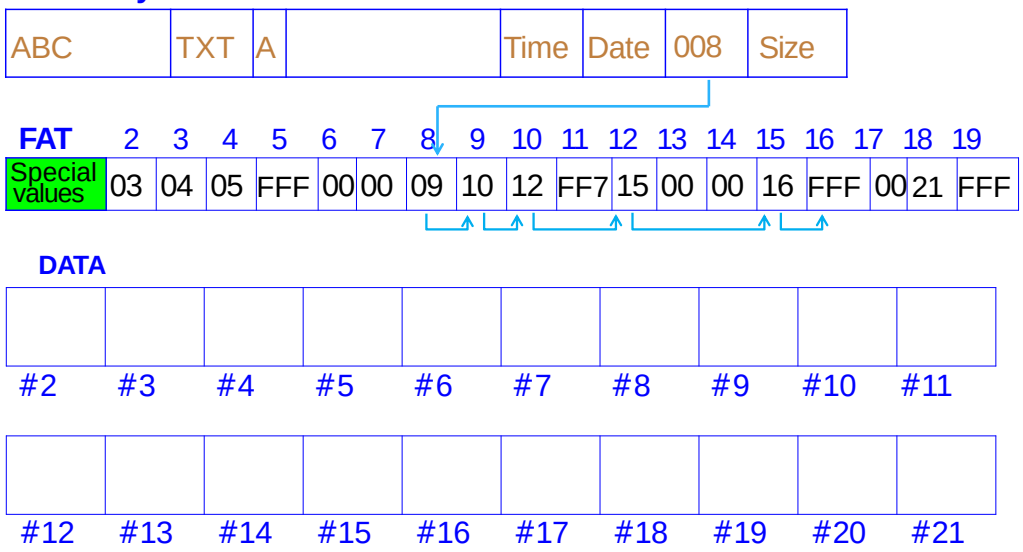
Implementation

Each element of FAT contain a value represents the property of corresponding cluster

FAT[(32)16]12	Meaning
[(0000)0]000h	Cluster is free
[(0000)0]001h	Unused value
[(0000)0]002h →[(0FFF)F]FEFh	Cluster being used. Value is a pointer, point to next cluster of file
[(0FFF)F]FF0h →[(0FFF)F]FF6h	Reserved value, unused
[(0FFF)F]FF7h	Mark corresponding cluster is broken
[(0FFF)F]FF8h→ →[(0FFF)F]FFFh	Cluster is being used and it's the last cluster of file (*EOC:End Of Cluster chain*). Usually value: [(0FFF)F]FFFh

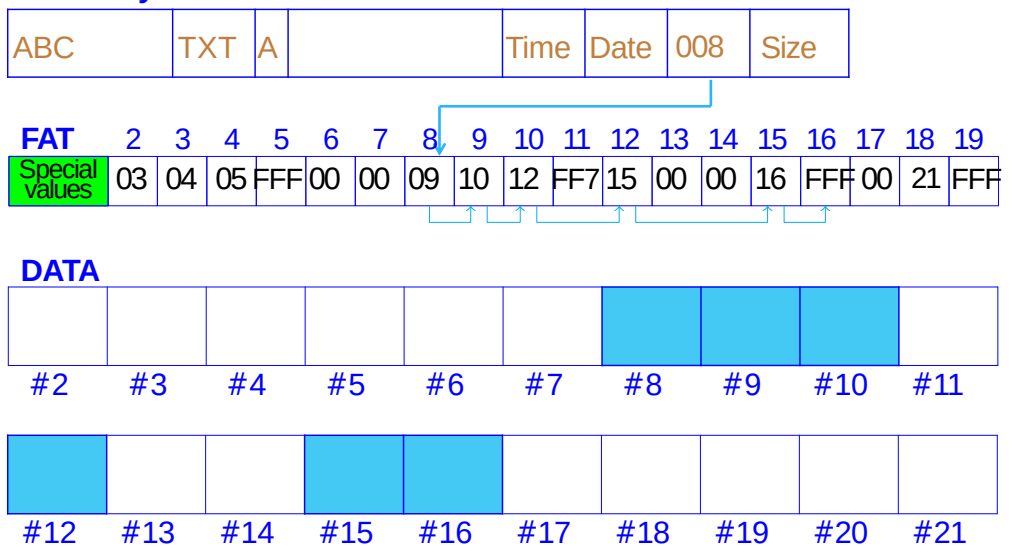
Cluster linking

Root entry

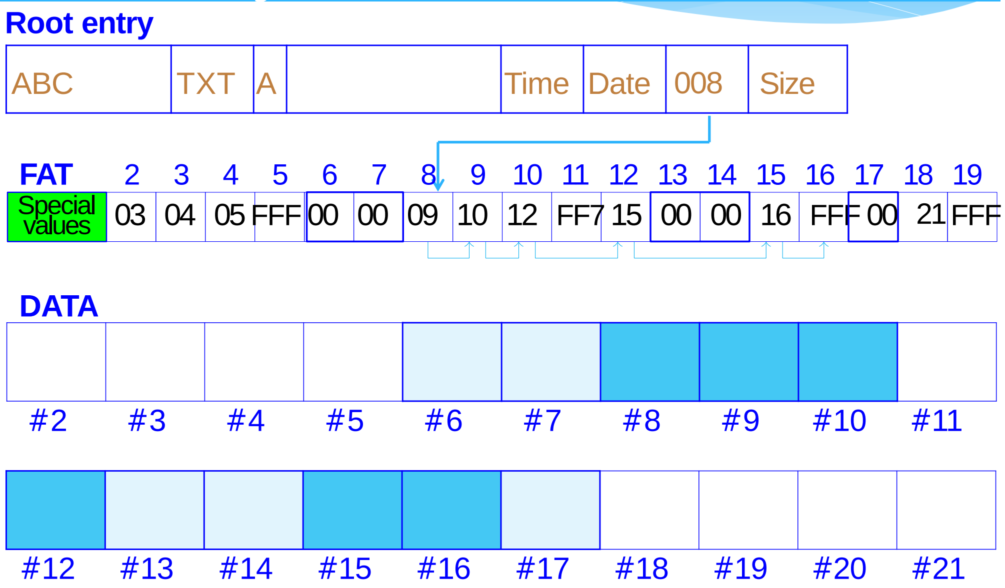


Cluster linking

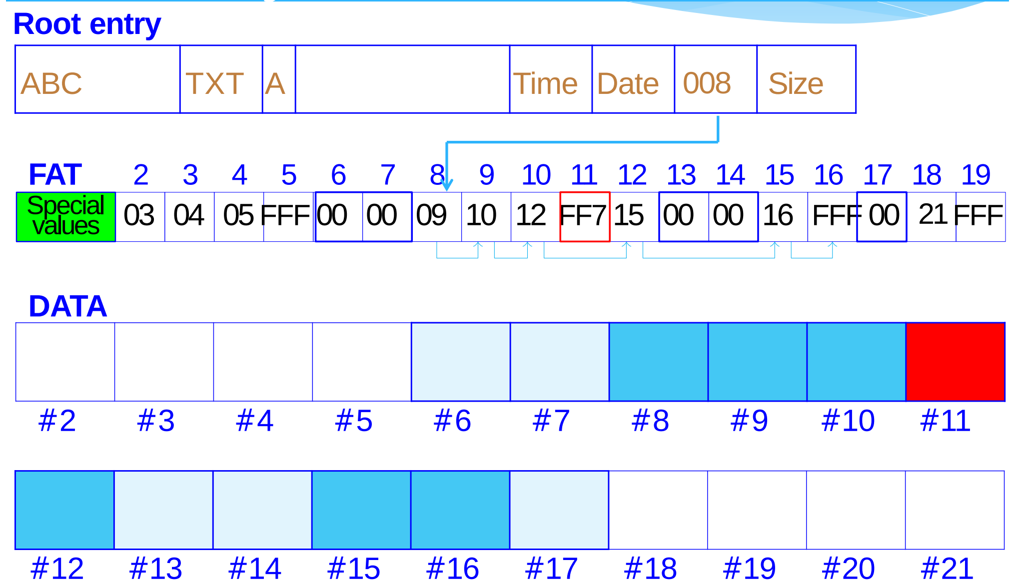
Root entry



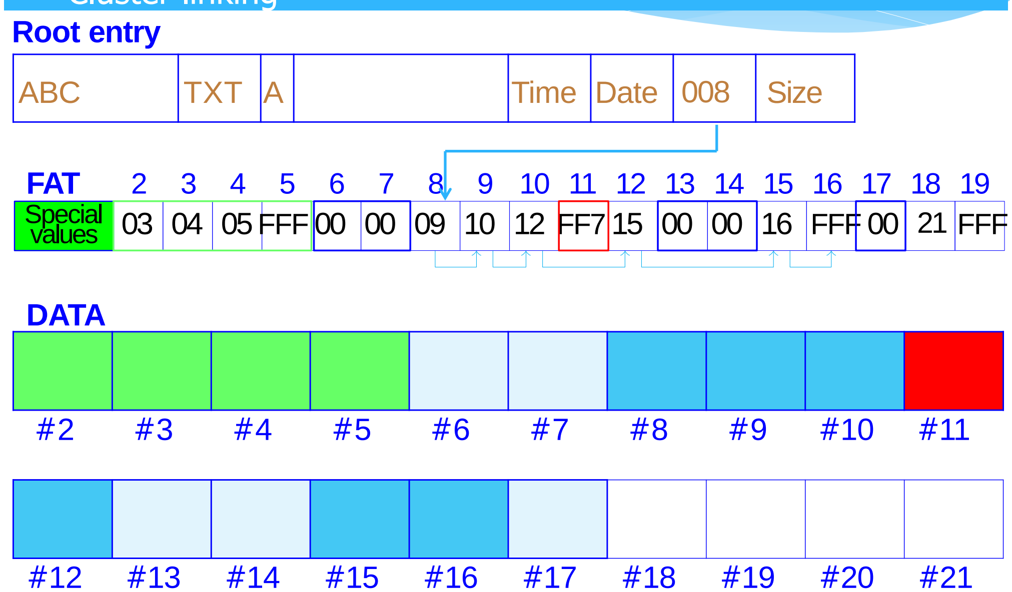
Cluster linking



Cluster linking



Cluster linking



Example: Read one sector of FAT32

```
#include <windows.h>
#include <stdio.h>
 int main(int argc, char *argv[]){
      HANDLE hDisk;
      BYTE Buf[512]; DWORD FAT[128];
      WORD FATAddr; DWORD byteread, i;
      hDisk = CreateFile("\\\\.\\F:", GENERIC_READ,
                    FILE_SHARE_READ|FILE_SHARE_WRITE, NULL,
                    OPEN_EXISTING,0,NULL);
      ReadFile(hDisk,Buf,512,&byteread,NULL);
      memcpy(&FATAddr,&Buf[14],2);//Offset 14 Sector truoc FAT
      SetFilePointer(hDisk,FATAddr * 512, NULL,FILE_BEGIN);
      ReadFile(hDisk,FAT,512,&byteread,NULL);
      for(i=0;i<128;i++) printf(" %08X ",FAT[i]);
      CloseHandle(hDisk);
      return 0;
}
```

Example: First Sector of a FAT32

A Root entry

52	45	41	44	4D	42	52	20	43	20	20	20	00	5E	B9	5A
A8	3E	A8	3E	00	00	CE	79	A4	3E	03	00	BD	0A	00	00

FAT

0FFFFFF8	FFFFFFFF	0FFFFFFF	00000004	00000005	00000006	00000007	00000008
0FFFFFFF	0000000A	0000000B	0000000C	0000000D	0FFFFFFF	0FFFFFFF	00000000
00000000	00000000	00000000	00000000	00000015	00000016	00000017	00000018
00000019	0000001A	0000001B	0000001C	0000001D	0000001E	0000001F	00000020
0FFFFFFF	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

- Boot sector
- FAT (File Allocation Table)
- Root directory

Structure of root directory

- Table consists of file record
  - Each record size is 32 bytes
    - Contain information involve one file/directory/ volume label
- FAT12/FAT16
  - Root directory lie right behind FAT
  - Size = Number of maximum elements in root directory * 32
- FAT32
  - Location is defined based on BPB
    - Filed #18: Number of ROOT's first cluster
  - Undefined size
  - Support long file name (LFN: Long File Name)
    - One file may use more than 1 element

Structure of one element

Stt	Ofs	Size	Meaning
1	0	8B	File name
2	8	3B	Extension
3	11	1B	File's attribute
4	12	10B	Not used in FAT12/FAT16. Used in FAT32
4.1	12	1B	Reserved
4.2	13	1B	File created time, unit 10ms
4.3	14	1W	File created time (*hour - minute - second*)
4.4	16	1W	File created date
4.5	18	1W	Last access date
4.6	20	1W	Number of starting cluster of file(*FAT32: High part*)
5	22	1W	Last update time
6	24	1W	Last update date
7	26	1W	Number of starting cluster of file (*FAT32: Low part*)
8	28	1DW	Size in byte

Structure of an element : file name

- Sequence of ASCII contain file name
- Not accept space in file name
  - Command copy, del,... do not recognize name contains space
- If smaller than 8 character, insert space character for sufficient 8 character
- First character may contain special meaning
  - 00h: First character of not used part
  - E5h (character "δ"): File corresponding to this element was deleted
  - 2Eh (character "."): Sub directory
    - cluster number field point to itself
    - Structure of sub directory is similar to root directory: contain 32bytes elements

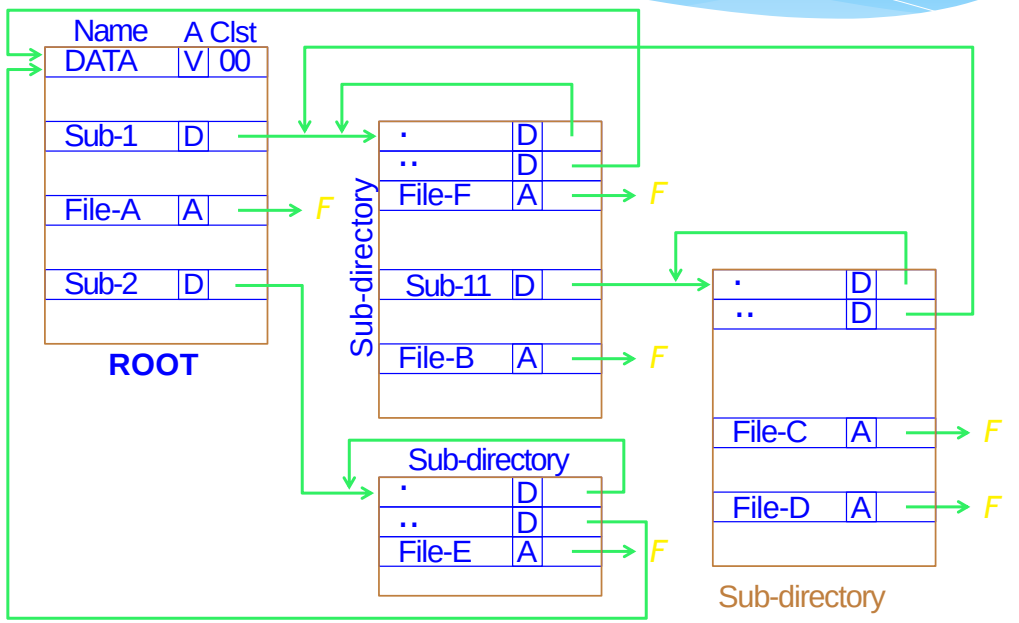
Chapter 4: File system management
4. FAT system
4.3 Root directory

Structure of an element : file name

- First character may contain special meaning (continues)
  - 2Eh2Eh (character ".."): Parent directory of current directory
    - Cluster number field point to parent directory
    - If parent is root, #cluster start by zero (FAT12/16)
    - Sub directory lie in Data area, manage similar to a file ⇒ File of file record
    - FAT12/16: Root directory is at a defined position; FAT32: Root directory lies in Data area

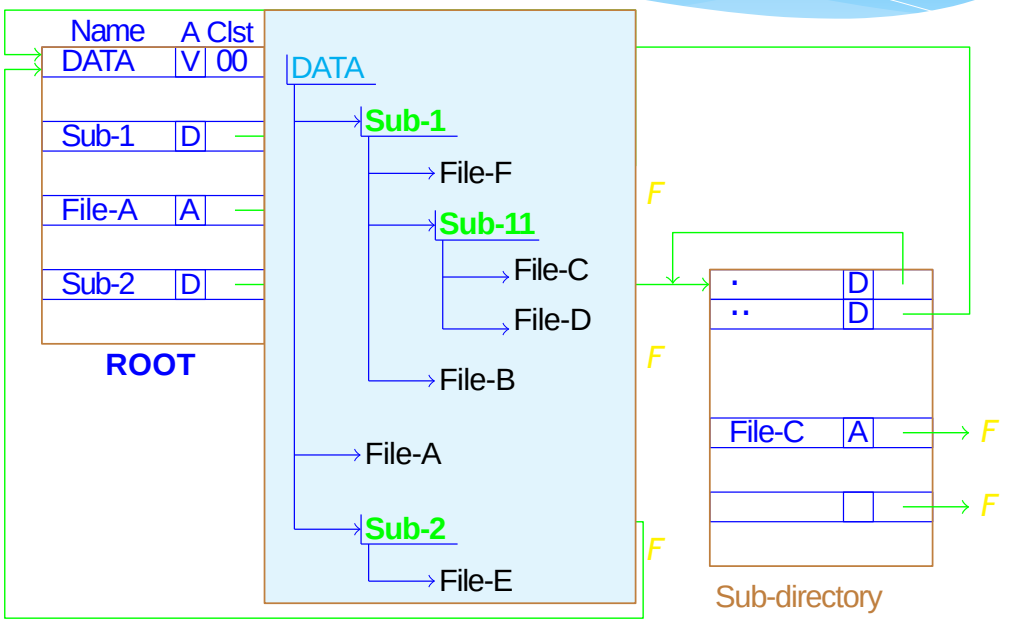
Chapter 4: File system management
4. FAT system
4.3 Root directory

Sub directory

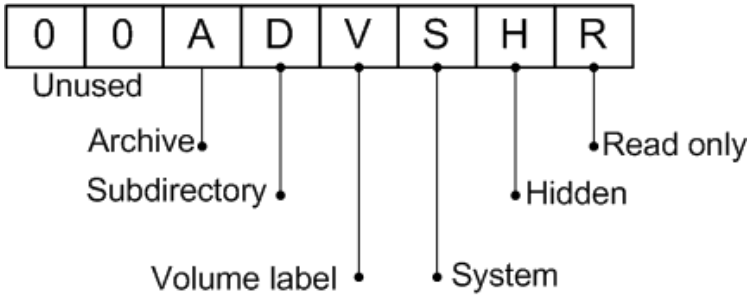


Chapter 4: File system management
4. FAT system
4.3 Root directory

Sub directory



Structure of an element : Attribute field



Example: Byte attribute **0Fh**:

0	0	0	0	1	1	1	1

⇒ File has attribute Volume label+System+Hidden+Read only

Note: Attribute byte 's value **0x0F** is not used in *MS-DOS* ⇒ For marking the *Long File Name element*

Structure of an element

Example: 15 hour 34 minute 45 second

0	1	1	1	1	1	0	0	0	1	0	1	0	1	1	0

Value: **7C56**

Structure of an element : Date

Example: Day 17 month 5 Year 2011

0	0	1	1	1	1	1	0	1	0	1	1	0	0	0	1

Value : **3EB1**

Long File Name (LFN) system

...
LFN element 3
LFN element 2
LFN element 1
element 8.3 (ttt~n.xxx)

Ofs	Size	Meaning
0	1B	Element Order
1	5W	First 5 unicode characters
11	1B	Attribute. Mark *LFN element* Always value 0Fh
12	1B	Reserved (00)
13	1B	Checksum: Allow check if long file name is corresponding to 8.3 name?
14	6W	unicode character 6,7,8,9,10,11
26	1W	Cluster number. Not used (0000)
28	1W	unicode character 12
30	1W	unicode character 13

Long File Name (LFN) system: Ordering field

- Show the order of LFN element
  - Each LFN element contain 13 Unicode characters
- First element has value of ordering field: 1
- Last element use bit number 6 to mark
  - Only use maximum 20 elements
  - After last character is 0x00 0x00.
  - Unused character has values 0xFF 0xFF
- Bit number 7 (0x80) show corresponding element is deleted
- Example: file "This is a very long file name.docx"

Entry	Ord	Attr	Data
LFN 3	0x43	0x0F	ame.docx
LFN 2	0x02	0x0F	y long file n
LFN 1	0x01	0x0F	This is a ver
8.3 Name	THISIS~1.DOC		

Example: A sector of ROOT

```
44 41 54 41 20 20 20 20 20 20 20 08 00 00 00 00
00 00 00 00 00 00 64 25 A5 3E 00 00 00 00 00 00
E5 44 48 20 20 20 20 20 50 44 46 20 18 0A 93 34
A5 3E A5 3E 00 00 6F 34 A5 3E 03 00 38 25 29 00
41 45 00 78 00 65 00 6D 00 70 00 0F 00 EF 6C 00
65 00 73 00 00 00 FF FF FF FF 00 00 FF FF FF FF
45 58 45 4D 50 4C 45 53 20 20 20 10 00 C4 9B 34
A5 3E A5 3E 00 00 9C 34 A5 3E 96 14 00 00 00 00
42 72 00 2E 00 63 00 00 00 FF FF 0F 00 43 FF FF
FF FF FF FF FF FF FF FF FF FF 00 00 FF FF FF FF
01 52 00 65 00 61 00 64 00 42 00 0F 00 43 69 00
6F 00 73 00 53 00 65 00 63 00 00 00 74 00 6F 00
52 45 41 44 42 49 7E 31 43 20 20 20 00 A6 B2 4B
A5 3E A5 3E 00 00 76 5C 9A 3E 3F 2E D1 01 00 00
52 45 41 44 4D 42 52 20 43 20 20 20 00 3C 86 5B
A5 3E A5 3E 00 00 CF 79 A4 3E 40 2E BD 0A 00 00
41 54 00 65 00 6D 00 70 00 73 00 0F 00 F0 00 00
FF FF FF FF FF FF FF FF FF FF 00 00 FF FF FF FF
54 45 4D 50 53 20 20 20 20 20 20 10 00 11 9A 96
A5 3E A5 3E 00 00 9B 96 A5 3E 46 2E 00 00 00 00
42 A1 01 6E 00 67 00 20 00 34 00 0F 00 12 2E 00
70 00 64 00 66 00 00 00 FF FF 00 00 FF FF FF FF
01 42 00 E0 00 69 00 20 00 67 00 0F 00 12 69 00
A3 1E 6E 00 67 00 20 00 63 00 00 00 68 00 B0 01
42 41 49 47 49 4E 7E 31 50 44 46 20 00 0A 93 34
A5 3E A5 3E 00 00 6F 34 A5 3E 03 00 38 25 29 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```


Example: Content of ROOT

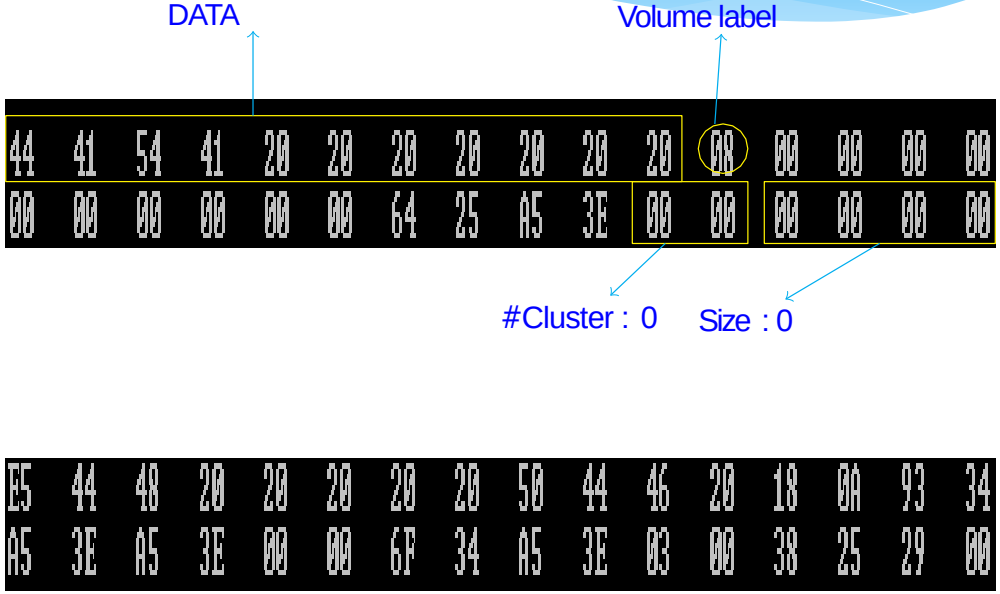
```
F:\>DIR
 Volume in drive F is DATA
 Volume Serial Number is DC27-F353

 Directory of F:\

05/05/2011  06:36 AM    <DIR>          Exemples
04/26/2011  11:35 AM               465 ReadBiosSector.c
05/04/2011  03:14 PM             2,749 READMBR.C
05/05/2011  06:52 PM    <DIR>          Temps
05/05/2011  06:35 AM         2,696,504 Bài gi?ng chuong 4.pdf
               3 File(s)      2,699,718 bytes
               2 Dir(s)      14,247,424 bytes free

F:\>_
```


Decode ROOT

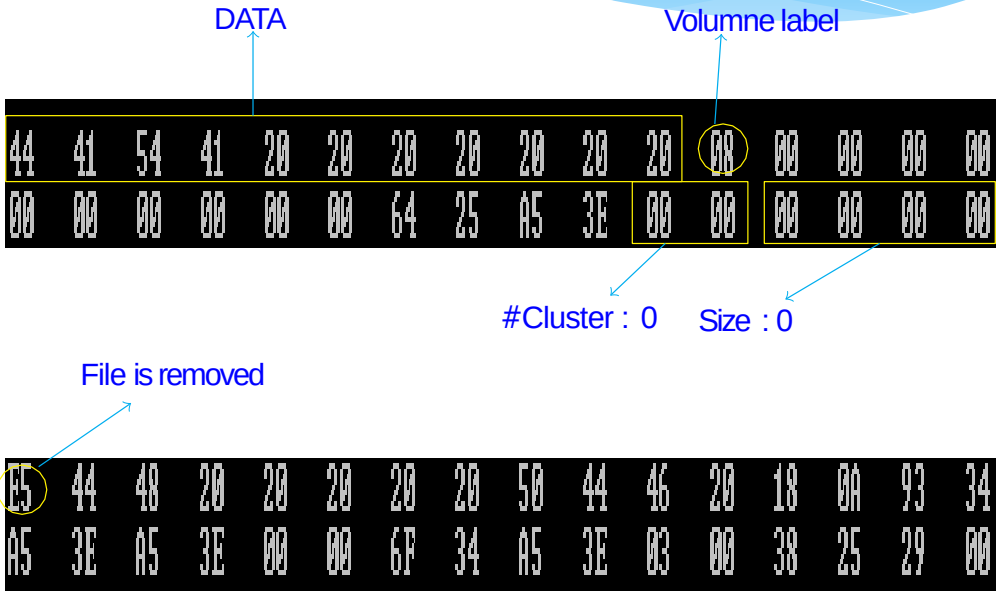
```
44 41 54 41 20 20 20 20 20 20 20 08 00 00 00 00
00 00 00 00 00 00 64 25 A5 3E 00 00 00 00 00 00
```

```
E5 44 48 20 20 20 20 20 50 44 46 20 18 0A 93 34
A5 3E A5 3E 00 00 6F 34 A5 3E 03 00 38 25 29 00
```

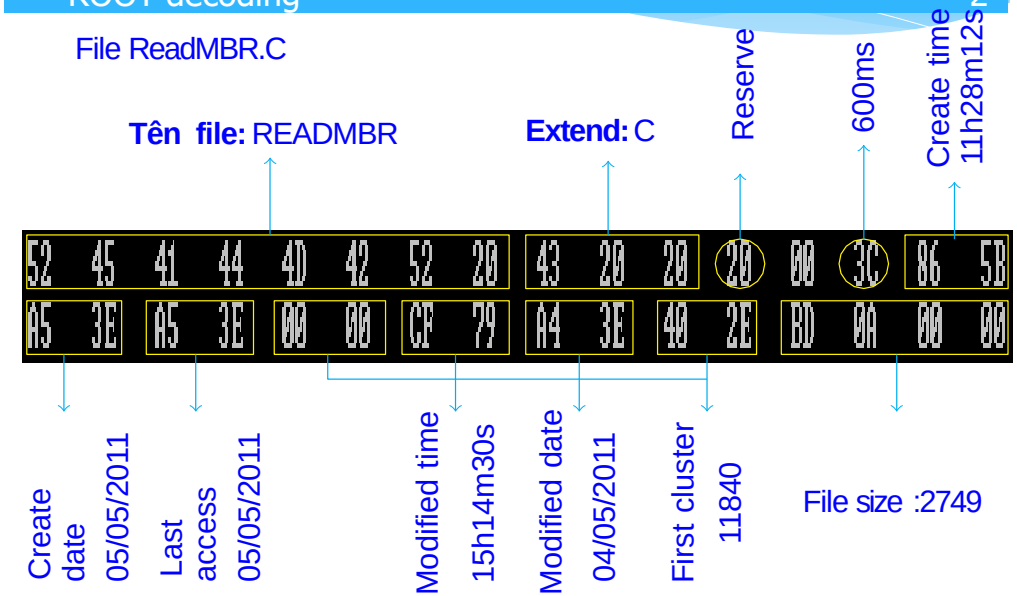
Decode ROOT 1



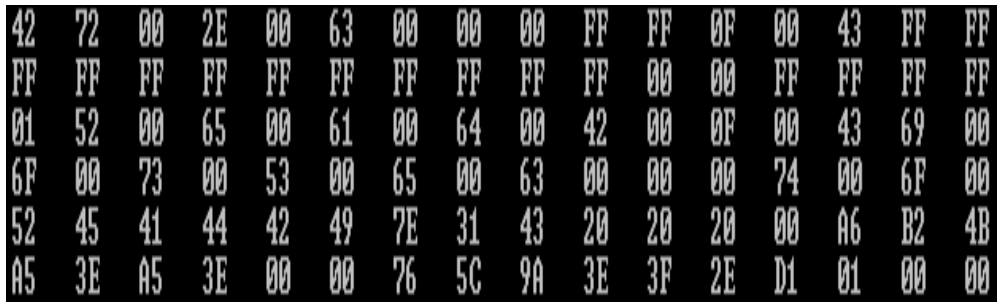
ROOT decoding 1



ROOT decoding 2

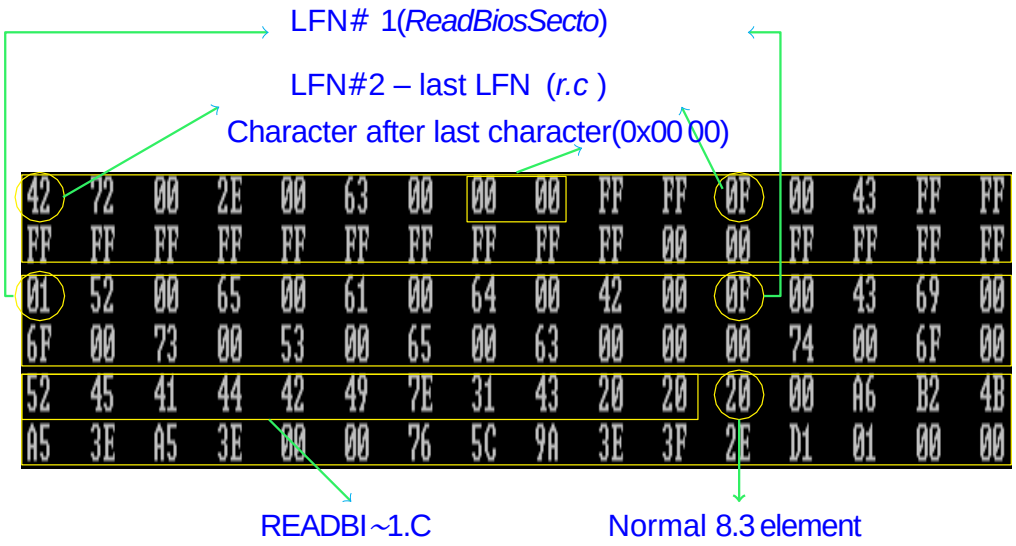


ROOT decoding 3



ROOT decoding 3

File: **ReadBiosSector.c**



Conclusion

4

Hệ Điều Hành

(*Nguyên lý các hệ điều hành*)

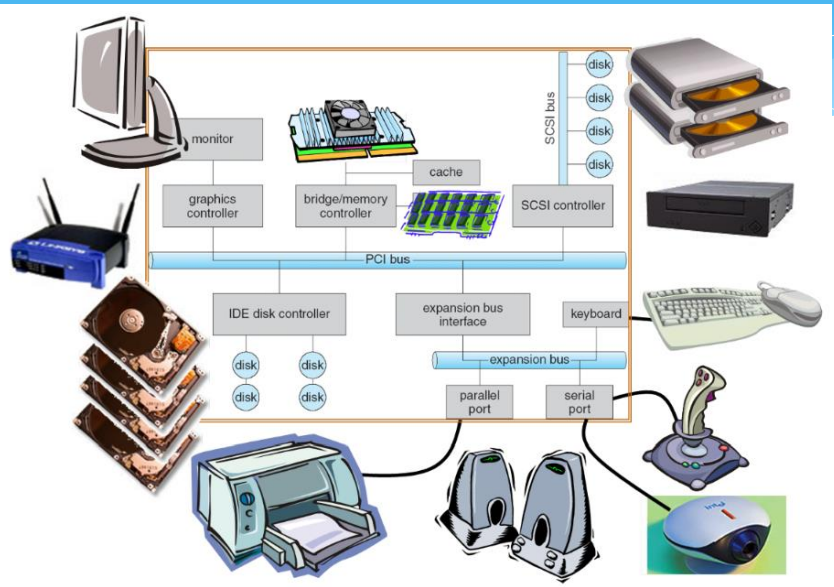
Đỗ Quốc Huy
huydq@soict.hust.edu.vn
Bộ môn Khoa Học Máy Tính
Viện Công Nghệ Thông Tin và Truyền Thông

Chapter 5 I/O Management

1. General management principle
2. System I/O service
3. Disk I/O system

- Introduction
- Interrupt and Interrupt handle

Chapter 5 I/O management



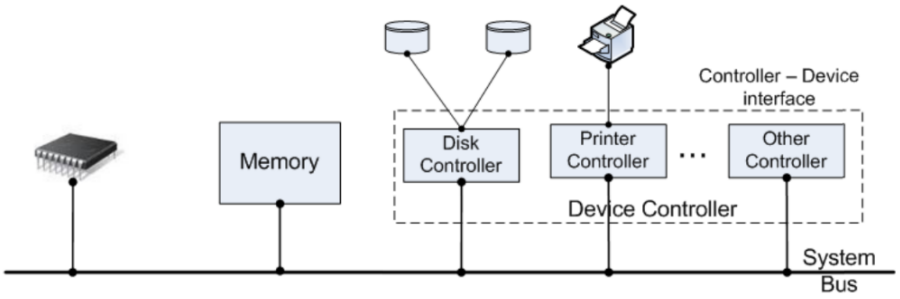
IO device

- Diversity, many kinds, different types
- Engineering perspective: device with processor, motor, and other parts
- Programming perspective: Interface like software to receive, executive command and return result
- Catergorize
  - Block device (disk, magenetic tape)
    - Info mation is stored with fixed size and private address
    - Possible to read/write a block independent from other
      - Operation to locate information exist (seek)
  - Character device (printer, keyboard, mouse,..)
    - Accept a stream of chracter, without block structure
    - No information localization operation
  - Other type: Clock

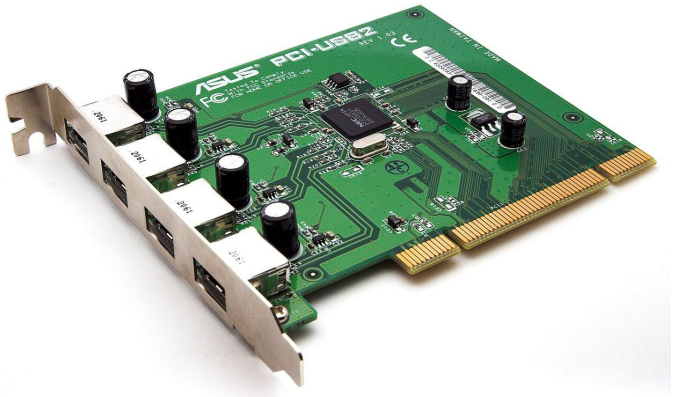
Controller device I

- Peripheral devices are diversity with many types
  - CPU do not know them all ⇒ No individual signal for each device
- Processor do not control device directly
  - Peripheral device is connected to the system via Device controller (DC)
    - Electrical circuit attached to the mainboard's slot
  - Each DC can control 1,2,4,.. peripheral devices
    - Depend on the number of connector on the DC
    - If the controller interface is standard (ANSI, IEEE, ISO,...) -> can connect to different devices
  - Each DC has its own register to work with CPU
    - Use special address space for registers: IO port

Controller device II



Controller device III



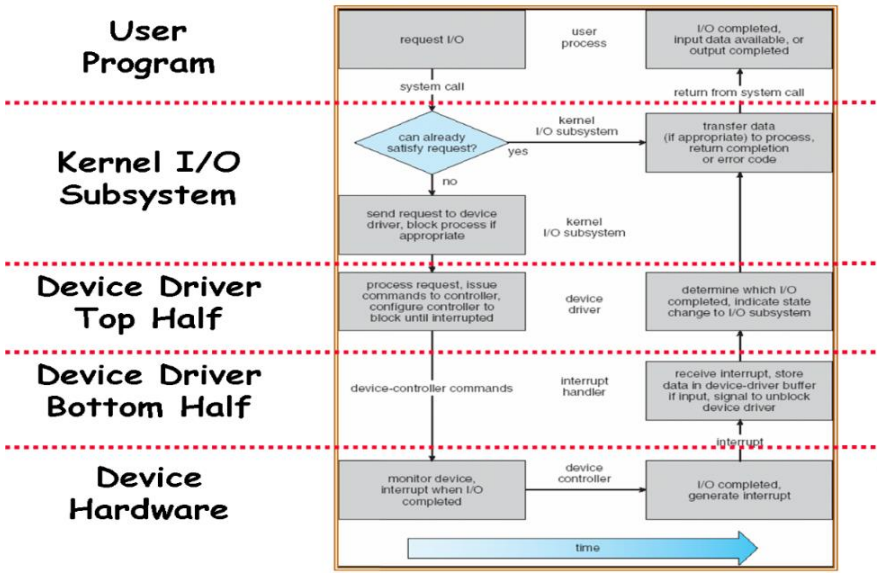
Controller device III

- Controller and device interface: Low level interface
  - Sector = 512bytes = 4096bits
  - Disk controller must read/write bits and group them into sectors
- OS only work with controller
  - Via device's registers
  - Commands and parameters are putted into controller's registers
  - When a command is accepted by the controller, CPU let the controller work itself and turn to other job
  - When command is finished, controller notify CPU via interrupt signal
  - CPU take result and device status via controlling device's register

Device driver

- Code segment in system's kernel allow interactive with hardware device
  - Provide standard interface for different I/O devices
- Categorized into 2 levels
  - High level: Access via system calls
    - Implement standard calls: open(), close(), read(), write()...
    - Interface between kernel and driver
    - High level thread wake up IO device then put control device thread into temporary sleep
  - Low level: Perform via interrupt procedure
    - Read input data or bring out next data block
    - Wake up the High level's temporary sleep thread when IO finish

IO request cycle



Peripheral device – Operating system interact

- After sending request to device, OS need to acknowledge
  - When device finish request
  - If device has error
- 2 methods to acknowledge
  - I/O interrupts
    - Device generate an interrupt signal to let CPU know
    - IRQ: physical path to interrupt manager
      - Map IRQ signal to interrupt vector
      - Call to interrupt handle routine
  - pooling
    - OS timely check device's status register
    - Waste checking period if the IO operation is not frequent
- Nowadays device can combine 2 methods (E.g. high bandwidth network device)
  - Send interrupt when first packet arrive
  - Pooling next coming packet until the buffer is empty

Interrupt definition

Mechanism to help device let the processor know its status

Phenomenon that a process is suddenly stopped and the system executive other process correspond to an event

Classification

- Based on Source
  - Internal interrupt
  - External interrupt
- Based on device
  - Hard
  - Soft
- Based on handling ability
  - maskable
  - unaskable
- Based on interrupt moment
  - Request
  - Report

Interrupt handle

① Write characteristic of event caused the interrupt into defined memory area

② Save interrupted process 'state

③ Change address of interrupt handle routine to instruction pointer register
- Utilize interrupt vector table (IBM-PC)

④ Run interrupt handle routine

⑤ Restore interrupted process
- Interrupt >< procedure !?

Chapter 5 I/O Management

1. General management principle
2. System I/O service
3. Disk I/O system

- Buffer
- SPOOL mechanism

General notion

- Peripheral device's characteristic: operate slow
  - Active the device
  - Wait for device to get to proper working status
  - Wait for IO operation to be perform
- To Guarantee the system's performance -> need to
  - Reduce number of IO operations, work with block of data
  - Perform IO operations parallelly with other operations
  - Perform accessing operation in advance

Buffer: Intermediate memory area, utilized for storing information during IO operation

Buffer classification I

- Input buffer
  - Can perform data access command
  - Example: read data from disk
- Output buffer
  - Information is putted into buffer, when buffer full, buffer content is then write to device

Buffer classification II

- Buffer attached to device
  - Constructed when open device/file
  - Serve device only, cleared when device is close
  - Good when devices have different physical record's structures
- Buffer attached to system
  - Constructed when the system start, not attached to a specific device
  - Exist during system working process
  - Open file/device ⇒ attach to already available buffer
  - Close device/file ⇒ buffer returned to system
  - Good for devices have same physical record's structure

Buffer organization

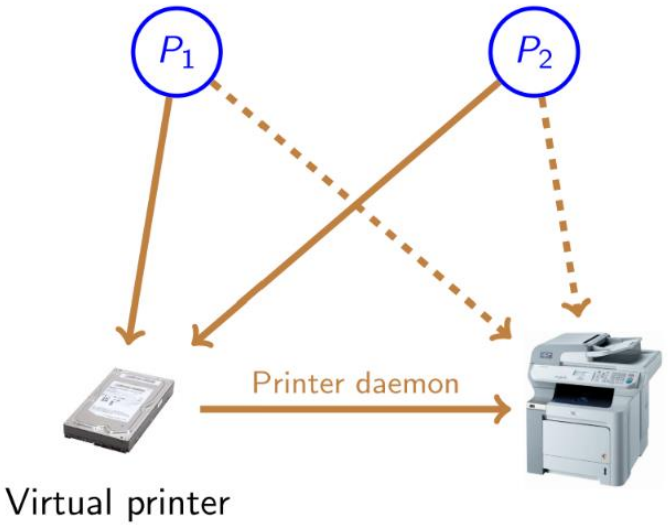
- Value buffer
  - Input buffer
  - Output buffer
- Process buffer
- Circular buffer
  - Input buffer
  - Output buffer
  - Processing buffer

- Buffer
- SPOOL mechanism

SPOOL (Simultaneous Peripheral Operation On-line)

- From programming perspective, IO device is
  - Station to receive request from program and perform
  - Return status code to be analyzed by the system
- -> use software to simulate IO device
  - IO device can be treated as process
    - Synchronized like in process management
- Objective
  - Simulate process of controlling and managing peripheral device
    - Check creating device working status
  - Create parallel effect for sequence device

SPOOL: Virtual printer



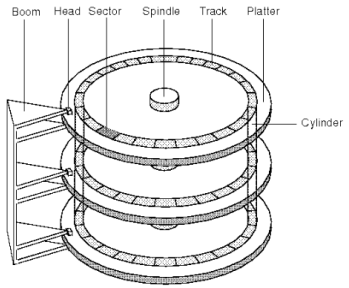
Virtual printer

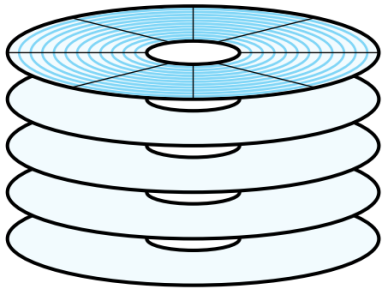
Chapter 5 I/O Management

1. General management principle
2. System I/O service
3. Disk I/O system

- Disk structure
- Disk accessing scheduling

Structure





- Modelled as array of logic block
  - logic block is the smallest exchange unit
- Map continuous logic block to disk's sector
  - Block 0 is first sector header 0 outer most track/Cylinder
    - Mapping follow an order: Sector → Header → Track/Cylinder
  - Reading header do not need to move much when read sector next to each other

Disk accessing problem

- OS is respond for effectively exploit the hardware
  - For disk: Fast access time and high bandwidth
- Bandwidth is calculated based on
  - Total bytes exchanged
  - Time from the first service request until the request is completed
- Access time consist of 2 part
  - seek time : Time to move header to cylinders contain required sector
  - Rotational latency: Time to wait until disk rotate to required sector

Disk accessing problem

- Hệ điều hành có trách nhiệm sử dụng hiệu quả phần cứng
  - Với đĩa: Thời gian truy nhập nhanh và băng thông cao
- Băng thông được tính dựa trên
  - Tổng số bytes đã trao đổi
  - Khoảng thời gian từ y/cầu dịch vụ đầu tiên cho tới khi hoàn thành
- Thời gian truy nhập gồm 2 phần
  - Thời gian định vị (seek time) : Thời gian dịch chuyển đầu từ tới cylinders chứa sector cần truy nhập
  - Độ trễ quay (Rotational latency) :Thời gian chờ đợi để đĩa quay tới sector cần truy nhập
- Mục đích: cực tiểu hóa thời gian định vị
  - Thời gian định vị ≈khoảng cách dịch chuyển
- Hàng đợi yêu cầu
  - Đĩa và bộ đ/khiển sẵn sàng, y/cầu truy nhập đc thực hiện ngay
  - Đĩa/bộ đ/khiển chưa sẵn sàng, yêu cầu đc đặt trong hàng đợi
  - Hoàn thành một yêu cầu truy nhập đĩa, lựa chọn y/cầu nào?

- Disk structure
- Disk accessing scheduling

Algorithm

- Objective: minimize seek time
  - Seek time ≈ moving distance
- Algorithm for disk IO request scheduling
  - FCFS:First Come First Served
  - SSTF: Shortest Seek Time First
  - SCAN
  - C-SCAN: Circular SCAN
  - LOOK/C-LOOK

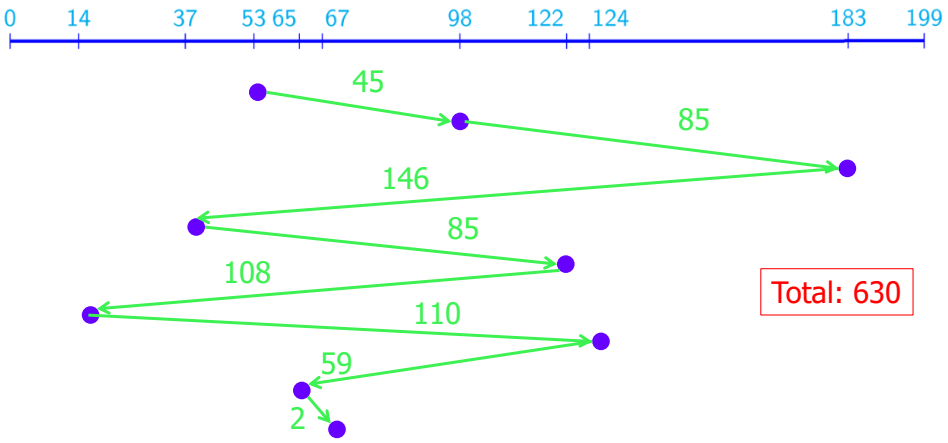
Assumption

- Accessing requests 98, 183, 37, 122, 14, 124, 65, 67
- Header current position at cylinder 53

FCFS

Access follow the request order ⇒Not effective

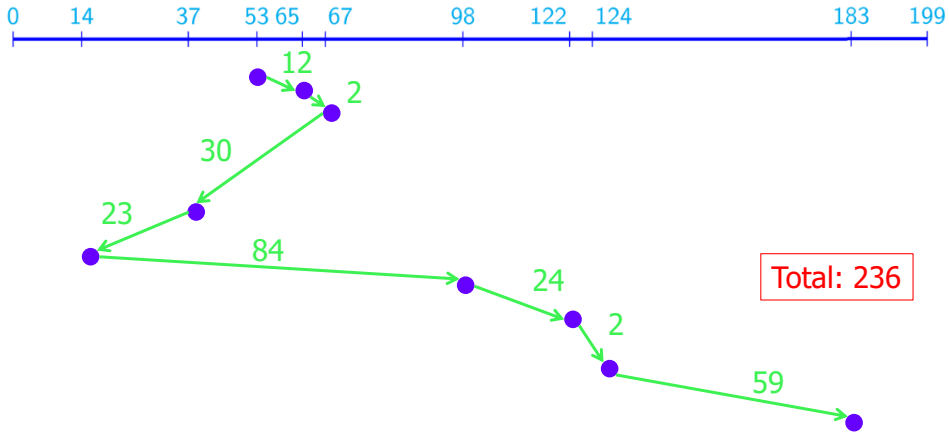
Accessing requests 98, 183, 37, 122, 14, 124, 65, 67



SSTF

Select access has smallest seek time from current position ⇒ A request may wait forever if new appearing requests closer to header (similar to SJF)

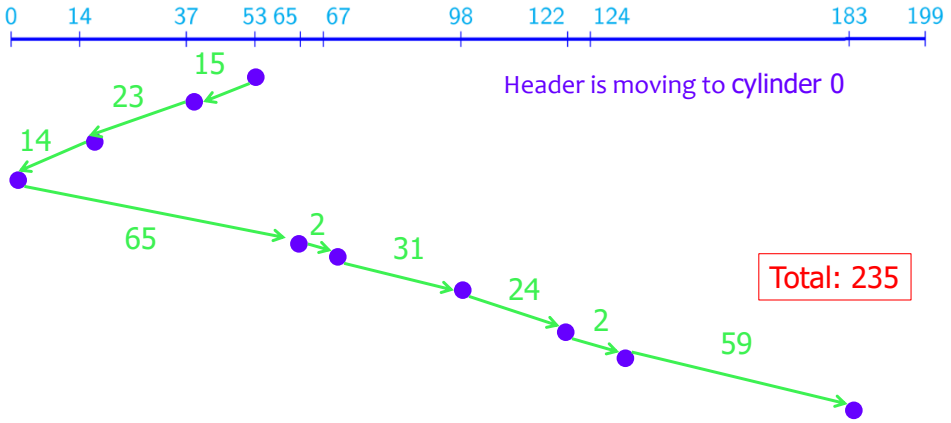
Accessing requests 98, 183, 37, 122, 14, 124, 65, 67



SCAN

Header move from outer most cylinder to innermost cylinder and return. Serve request met on the way

Accessing requests 98, 183, 37, 122, 14, 124, 65, 67



C-SCAN

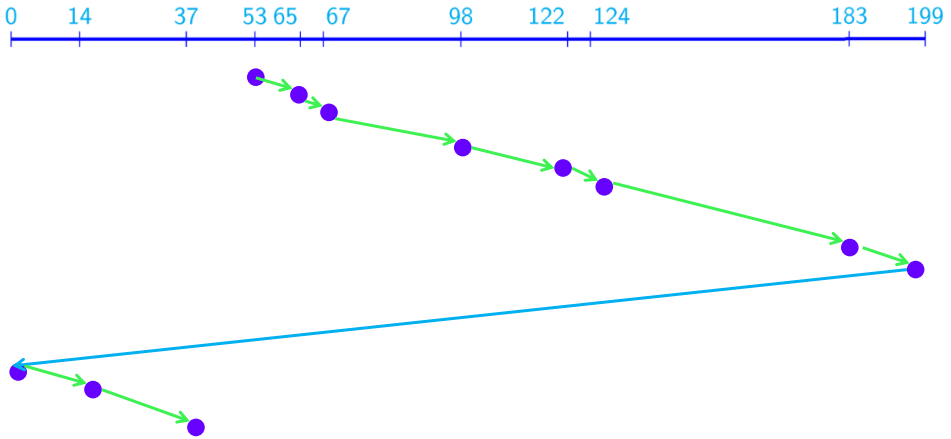
Principle: Treat cylinders like a circular linked list: Outer most Cylinder connect with innermost cylinder

- Header move from outermost cylinder to innermost cylinder
  - Serve request met on the way
- When inner most Cylinder is reached, return to outermost Cylinder
  - Do not serve request met on the way
- Remark: Retrieve more equal waiting time than SCAN
  - When header reach to one side of disk (innermost/outermost cylinders), density of requests appear at other side will be higher than current place (reason: header just passing by). This request need to wait longer ⇒ Return to other side immediately

C-SCAN

Header move from outermost cylinder to innermost cylinder and return. Serve request met on the way

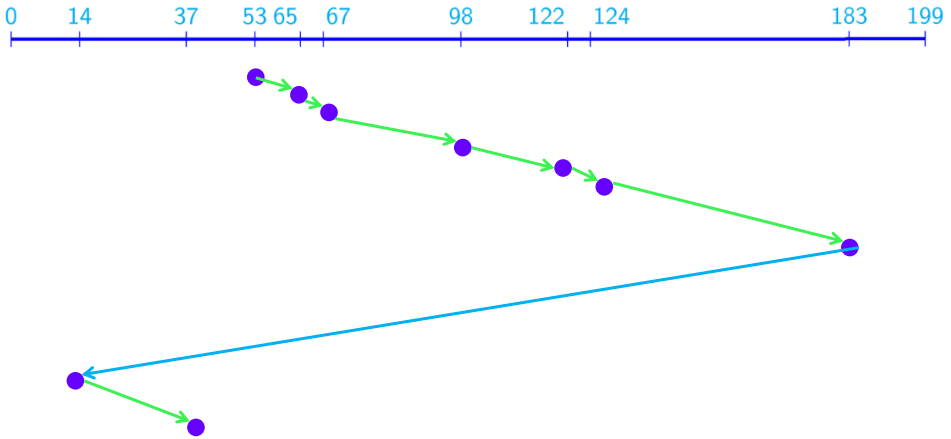
Accessing requests 98, 183, 37, 122, 14, 124, 65, 67



LOOK/ C-LOOK

SCAN/C-SCAN's version: Header does not move to outermost/innermost cylinders, only to farthest request at 2 sides and return

Accessing requests 98, 183, 37, 122, 14, 124, 65, 67



Conclusion