Hong Kong Metropolitan University
Department of Technology

Lecture Notes

COMPS267F
Operating Systems

2022 Presentation

Table of Content

Acknowledgement

- Icon made by Pixel perfect from www.flaticon.com

COMPS267F Operating Systems

Overview of Operating Systems 1

1.1 Why Studying Operating Systems?

This question should always be asked, but the timing is crucial.

- It is good to ask at the beginning of a course. It gives students some motivations.
- It is bad to ask at the end of the course. Students are clearly not happy after going through the course and they ask the instructor this question!

Before we start, I will provide a few short answers. Hopefully you will have you own positive answers at the end of the course.

- Operating systems are one of the most important pieces of software. A computer system (hardware) is nothing without software.
- Operating systems are one of the major achievements in computing. Computing resources can be used effectively.
- Many techniques invented for operating systems are clever, useful and general. You can apply them in other technical problems.
- The course involves a lot of analysis of different techniques and critical thinking. You will become a more able IT professional.

What are operating systems?

- Operating systems are essential components of computer systems.
    - They are computer programs running on computer hardware.
    - They are computer programs that manage computer hardware.
    - They are computer programs to facilitate the use of computer hardware.

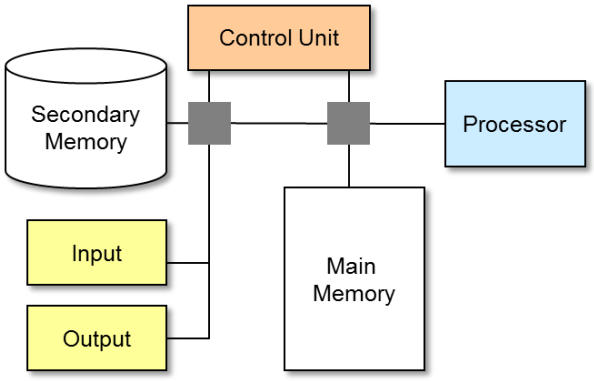
This course on operating systems starts from programmable computer systems.

1.2 Programmable Computer Systems

A programmable computer system means that a computer that can be repurposed.

- The system can become a word processor, a slide presenter, a computer game platform, an web browser.
- Running of different programs changes the purpose.

The following figure shows the structure of a programmable computer system.



It has a number of essential components:

- The Processor or the CPU
- Memory systems
    - Main or primary memory
    - Secondary memory
- Input and Output devices
- The Control Unit

You are assumed to have studied these components before, as well as how programs are executed, and so no explanation is given here.

You need to know that, for the execution of computer programs, all these components are involved in different capacities.

- Input: loading programs into the computer system
- Memory: storing programs long-term (i.e. secondary memory) and during-execution (i.e. main memory)
- Processor: executing programs
- Output: generate the results of program execution

What is a Computer Program?

A computer can do things very quickly. A programmable computer can do purposeful things.

- Modern computers have many useful purposes.
    - Examples include controlling your television at home to managing the transactions for a bank.
- These meaningful purposes require a computer to perform a designed sequence of operations.

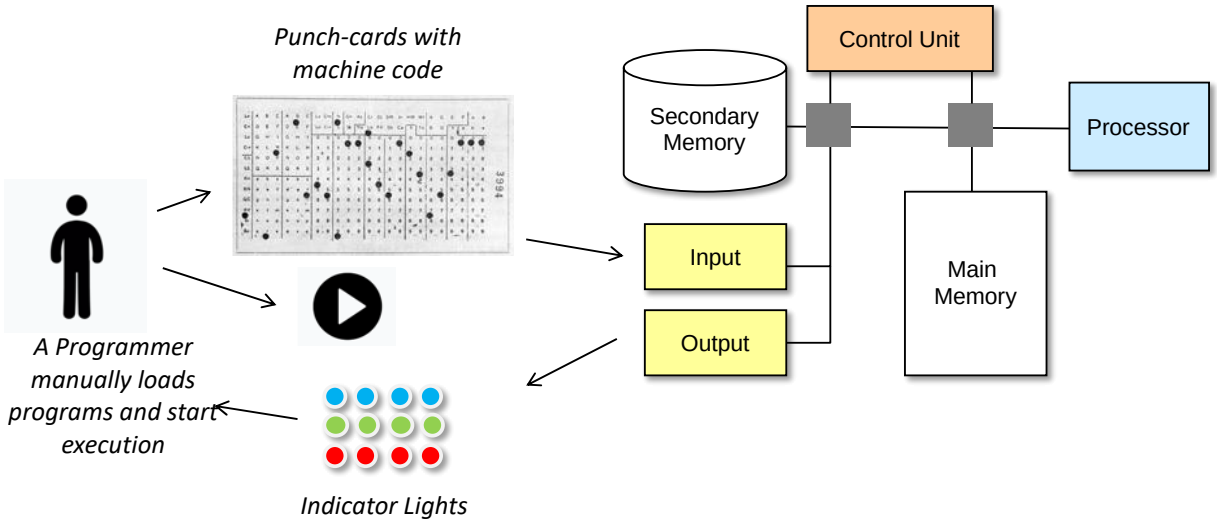
Computer programs are artefacts that make a computer to perform a purposefully designed sequence of operations.

- A computer program can take over the control of a computer.
- The computer follows the instructions defined in the program.
- A program is made up of instructions, and the execution of each instruction may cause one or more computer operations.

1.2.1 The Need for Operating Systems

We have taken operating systems for granted. An easy way to understand the importance of operating systems (OS) is to give you an OS-less computer system. How can you use it?

Serial Processing



The hardware has to provide some controls for programmers. In the old days, for example, the computer provides punch-card readers for loading machine code programs, and buttons to control the execution. No operating system was there.

Did the above way of computing work?

- Yes, to a good extent enough for making a lot of things such as atomic bombs and decryption.
- Computer hardware then was very expensive and so they must be used for the most important jobs.

There were two major problems.

- Only one programmer can run one program at a time.
    - It is not convenient when many people want to use the computer.
    - Usually many programmers want to run many jobs.
    - Someone has to manually manage the schedule, perhaps using a notebook to pencil in who should use the computer at each timeslot.
- The programmer can take some time to setup the program.
    - The setup time has wasted the expensive computing resource.
    - What happened if the punch-cards are dropped on the floor and messed up?

There is one inconvenience and one problem.

- The inconvenience is that the computer can only execute one program at a time.
- The problem is the time wasted in setup and scheduling, so that one-by-one each user can use the computer to run a program. This is often called serial processing.

A solution is needed for:

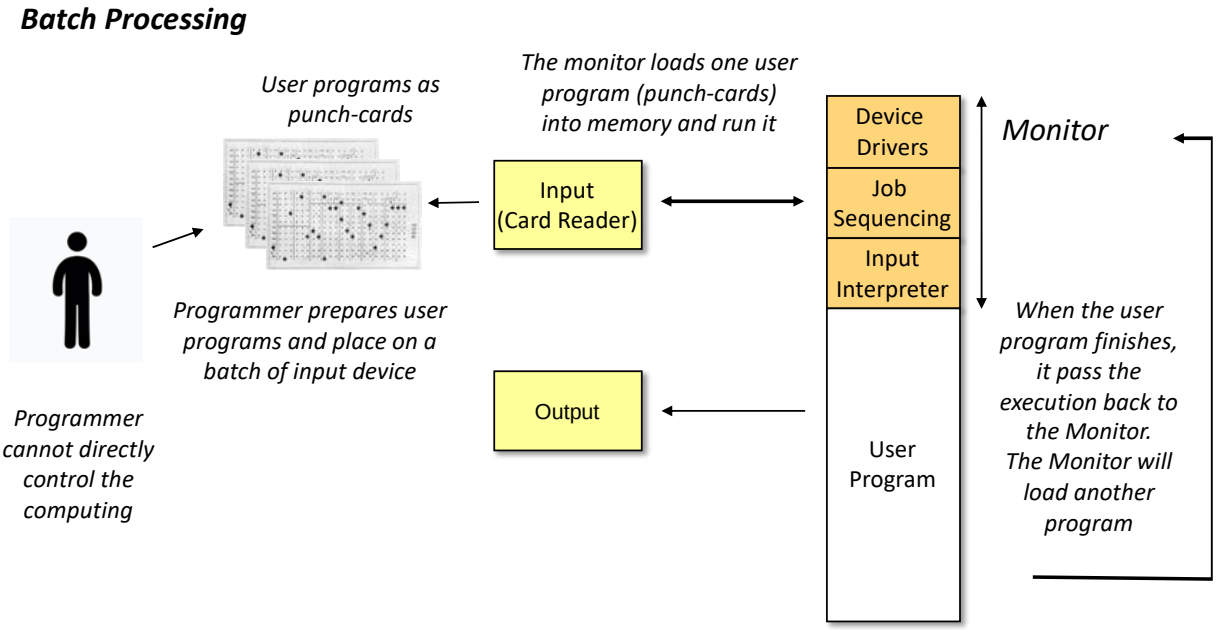
- Reducing the setup time for running a program.
- Scheduling the execution of programs.

1.2.2 Batch Systems

A simple solution is based on automation.

Simple Batch Systems

A program called monitor is designed to manage the setup of program and the execution of program.



The monitor is a program, that is residing in the memory of a computer.

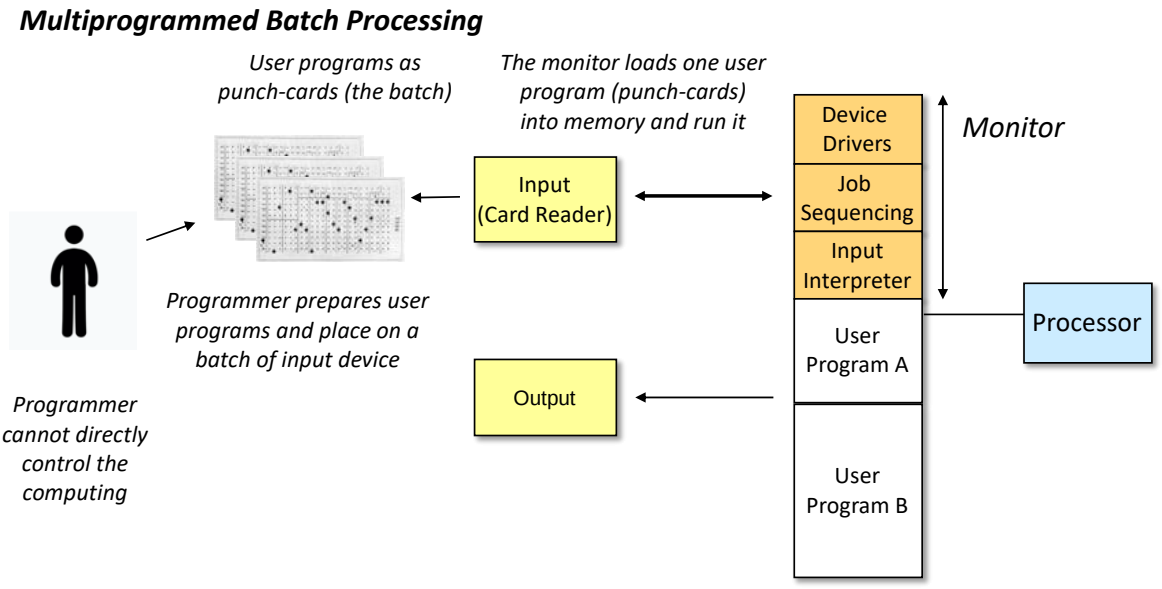
- The monitor loads a user program that has been placed in the input device (i.e. card reader) into a dedicated part of the memory.
- The end of the user program has included a branch back to the monitor when it finishes.
- The monitor passes the execution (i.e. program counter) to the user program.
- When the user program has finished, the execution is branched back to the monitor.
- The monitor then loads another user program from the batch.

This mode of operation is called batch processing. The monitor can be considered as an earlier form of operating systems called batch OS.

Multiprogrammed Batch Systems

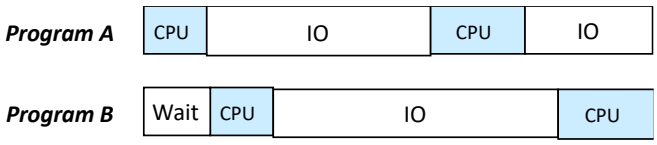
The most important drawback of the simple batch system is due to uni-programming (or single programming) nature. Single programming means only one program is running and another program cannot run until the first program has finished.

- Most programs do not use the processor all the time.
    - A program may be doing I/O, reading data or writing data.
- The processor is not fully utilized. Remind you that the processor is the most precious resource.
- The free processor should be made to do more work, such as running another program.

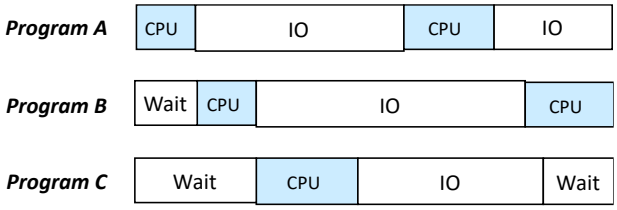


The figure illustrates a multiprogrammed batch system.

- Two programs (A and B) are loaded into the memory.
- The monitor will select a program (say Program A) to be executed by the processor.
- When the selected program is doing I/O, another program can be selected to occupy the processor.



With multiprogramming, the utilization of the processor can be kept at a high level if there are more programs that are executed together.



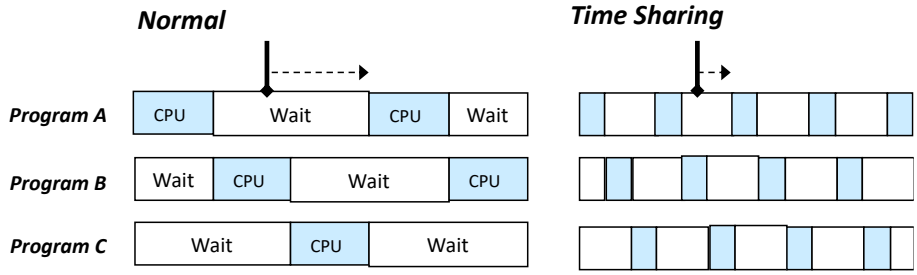
Despite the benefits, achieving multi-programming is not easy. There are quite a number of technical challenges to solve.

Benefits of Multiprogramming	**Challenges to Overcome**
Convenience to users. Able to do multiple tasks at the same time (multi-tasking)	Running multiple programs require a memory that is large enough to load multiple programs.
Increases the utilization of processors	Memory has to be managed so that each program can have its own space
Efficient in process execution	Memory of each program has to be secured to prevent other programs from attacking
	A large of memory is needed for effective multi-programming
	The monitor needs to manage the job scheduling effectively

1.2.3 Time Sharing Systems

Interactions between users and computers are desirable for technology-enhanced modern lifestyle. In some serious applications such as transactions, these interactions are necessary. Batch processing is efficient, but it does not allow user interaction.

An important system requirement for user interaction is response time. Users would not feel satisfied if a program does not respond quickly.



Consider a system with one processor, the above figure (left) shows how three programs share the processor in a multiprogramming system. Each program is executed by a user.

- A program can be either using the processor or waiting.
- If a user wishes to interact with Program A when A is waiting, the program does not respond until it is given the processor.
- The time lapse is known as the response time.

A short response time is desirable for user interactions. A solution is to adopt a time-sharing approach, which is shown on the above figure (left).

- The monitor or the operating system interleaves the execution of the three programs in a quick manner.
- Each program uses the processor for a very short time. The program then does not wait for too long. It is able to use the processor again.

Time-sharing systems are the standard approach for modern computing.

1.3 Fundamental Operations of Modern Computers

There is only one processor in a computer system, but there are many jobs to do. These jobs include user programs and operating systems. A mechanism is needed to direct the attention of the processor from one job to another job.

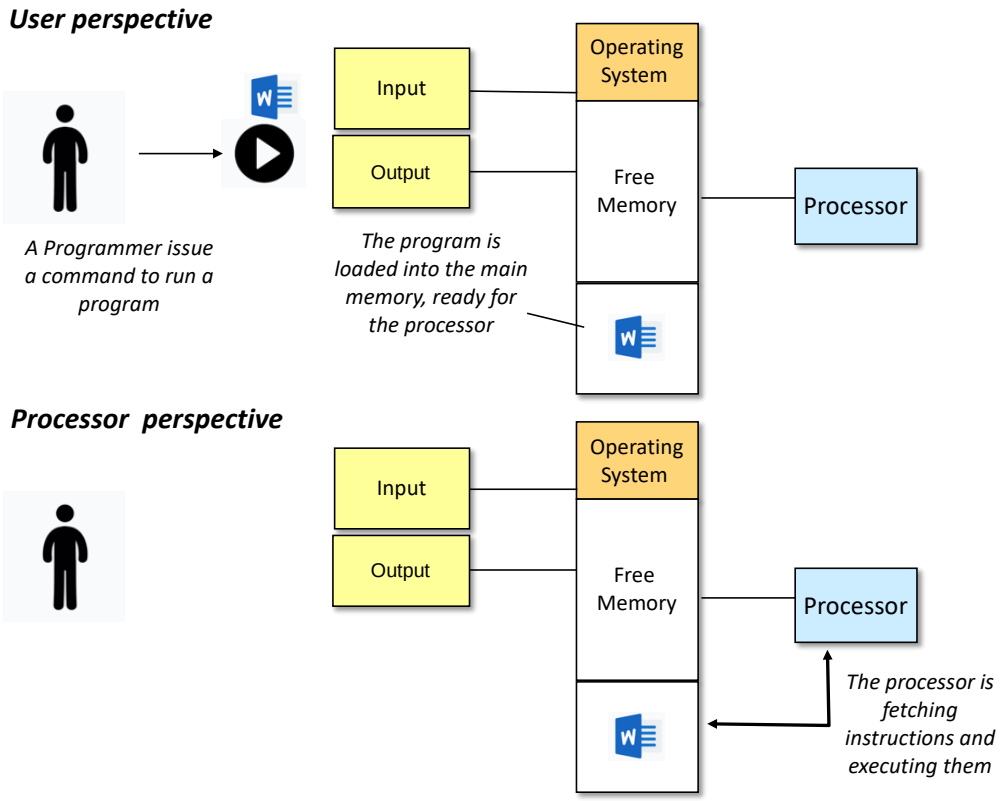
1.3.1 The Concept of Running a Program

What is Running a Program?

The term running (or executing) a program needs to be defined. There are two perspectives which are not the same.

- User perspective. User has issued a command to the computer to execute a program.

- Processor perspective. The processor is executing instructions of a program.

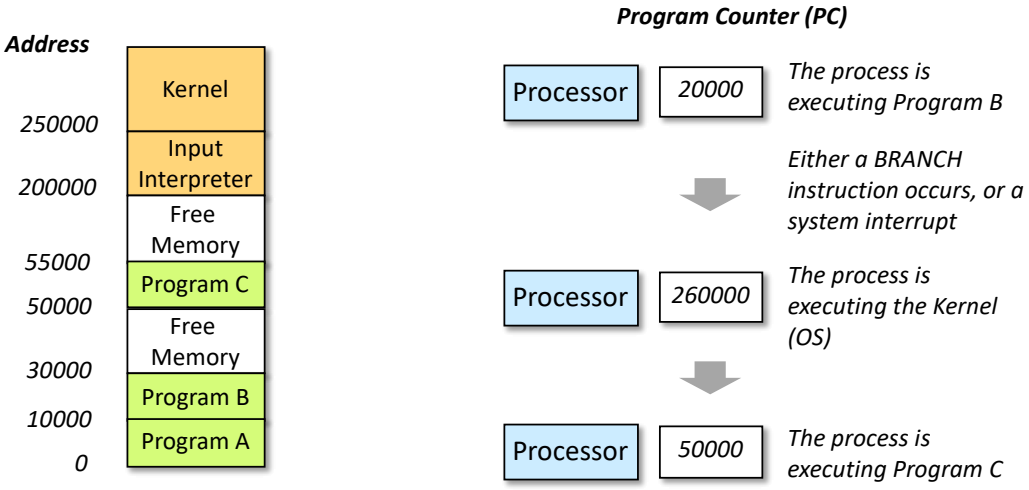


The above figure explains the difference between the two perspectives. Therefore, a program being executed by a computer system really means it has been loaded into the memory. The operating systems then schedule the use of process by the loaded programs.

What is Running a Program by the Processor?

A processor executes the instructions of different programs. At this moment, it is executing one program and the next moment it is executing another program. How can the processor made to change the program it is executing?

The program counter determines which program the processor is executing. Changing the program counter can change the program to be executed.



A BRANCH instruction can normally change the program counter. The change is however restricted due to security reasons.

- The OS can freely change the program counter when it is passing the execution to another program (i.e. a user program). Here are some examples.
  - After the OS has loaded a new program into the main memory, it then executes a BRANCH instruction to jump to the start of the new program
  - In a time-sharing system, the time slice of a program is used up, and the control goes back to the OS. The OS then makes another program to resume running.
- A user program can make a system call and jump to the address of the OS kernel.
- A user program cannot normally jump to the address area of another user program.

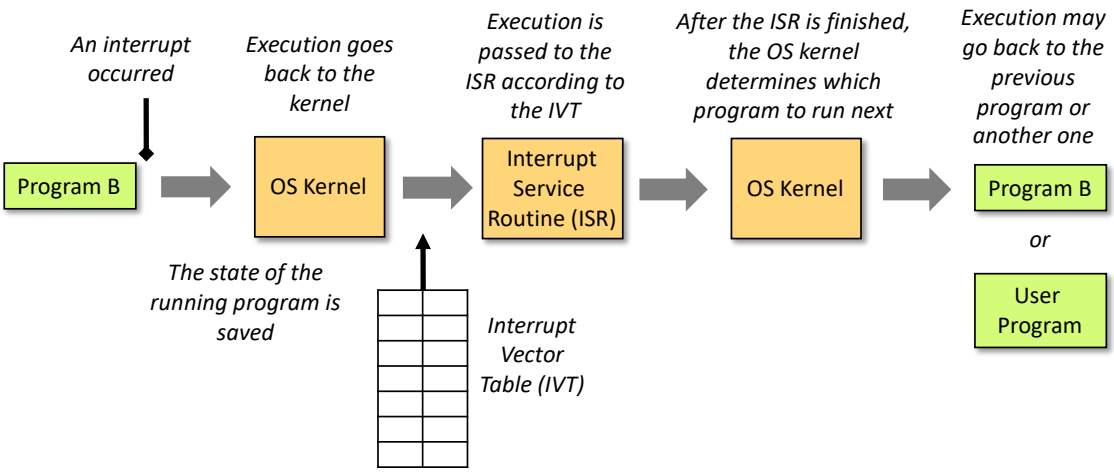
1.3.2 System Interrupts

System interrupt is one of the most important mechanisms in a running computer system. When an interrupt occurs, the system control always goes back to the OS. The program counter is moved back to the OS.

- A running program, when an interrupt occur, will be stopped. It is because the program counter has changed back to running the OS.
- The OS saves the essential information of the suspended program. The information, such as the program counter, will be useful later when the program resumes.
- According to the interrupt type, the OS looks at the Interrupt Vector Table (IVT) for the routine that is designed to handle the interrupt.
- The routines are called Interrupt Service Routines (ISR).
- The OS makes a call to the corresponding ISR.
- When the ISR finishes, the OS decides what to do next. The previously suspended program may be asked to resume, or a new program is executed.

The following figure shows how an interrupt is handled.

Handling of Interrupts

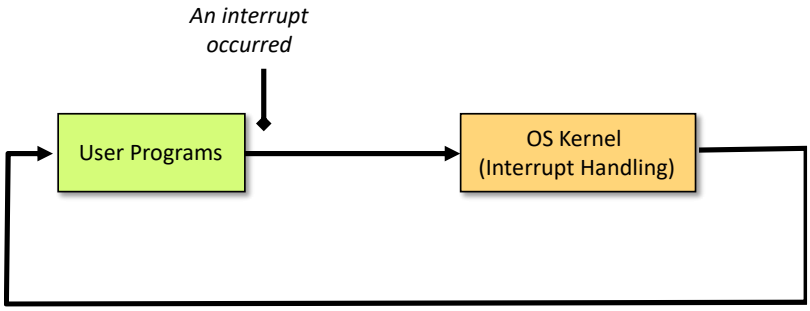


Importance of System Interrupts for Computer Systems

The importance of system interrupts lies in the mechanism of allowing the OS to regain control.

- It is not possible to know how long a user program will run.
    - The user program may contain an infinite loop, and it will run forever.
    - The computer system has therefore become out of control.
- Interrupts will force any user program to suspend and the execution can return to the OS.
- The OS has the chance to deal with any problem that has occurred.

Modern computer systems therefore run in a cycle similar to the following figure. The cycle involves running a user program and running the OS kernel.



Types of Interrupts

Interrupts are like signals that indicate something is happening. There are three typical types of interrupts.

- IO interrupts or external interrupts.
    - These interrupts are raised by IO devices such as keyboard (when the user press Enter) or mouse (when the user moves it).
    - IO devices may also raise it if there is an error or the job has finished, etc.
- Internal or hardware interrupts.
    - Caused by hardware such as system clock that raises an interrupt in regular time interval
    - Caused by hardware such as processor when there is a serious instruction error.
    - Caused by hardware such as memory system when there is a serious error such as invalid memory access.
- Software interrupts.
    - Caused by user programs. Programmers may use interrupts in their program to implement logic. Program triggers interrupts by a system call.
    - Software interrupts are often called a trap.

PC desktop computers running on Windows have specified Interrupt Request Number (IRQ) that maps an identifier to various types of interrupts.

- For example, IRQ 0 stands for system timer.
- IRQ 1 and IRQ 12 for keyboard and mouse, etc.

Interrupt Service Routine (ISR)

Interrupt Service Routine is a function that is designed to deal with a particular type of interrupts. The addresses of each ISR are stored in the Interrupt Vector Table (IVT) that maps the interrupt type to the address of ISR.

1.3.3 Security Issues in Multiprogramming Systems

User programs should not be trusted. They may be written in a way that is harmful to the computer system. They may take over the computer or they tamper with the data of other programs.

Dual Mode Operation

Hardware supports two modes of operations: user mode and monitor mode. Privileged instructions can only be executed in monitor mode. These should include any machine instruction that can cause harm.

- There exists a monitor bit in the hardware that indicates the state.
- When OS is in control of the computer system, it is in monitor mode. Before the OS pass the control to a user program, it switches to user mode.
- System call is the means that a user program can ask the OS to perform privileged instructions in monitor mode.
- Potentially a user program can capture the computer in monitor mode by modifying the interrupt vector, which resides in the memory.

Memory Protection

Memory access must be controlled to prevent several mishaps. For example, accessing another program's data or modifying the interrupt vector should be guarded.

- The operating system manages the range of legal memory address during the execution of a process.
- Allowed to access its own allocated memory range.
- Prevented from accessing other memory range.

A base register and a limit register are used to implement the protection. The computer system checks every address access by user program against this valid range of memory address. Only the OS has the permission to load the pair of registers.

CPU Protection

The operating system must ensure that it maintains control of the computer systems. The execution of a program must involve the operating systems relinquishing control to the program. The operating system therefore arranges to regain control in regular time intervals so that it has the opportunity to tackle security problems.

- This is achieved by having a timer so that when the time is up it generates an interrupt so that the OS regains control.
- When a user program is first executed, the OS assigns a timer to it with a certain time limit. The timer interrupts every second and the counter is decreased.

1.4 Case Study: Pseudo-Multitasking in DOS

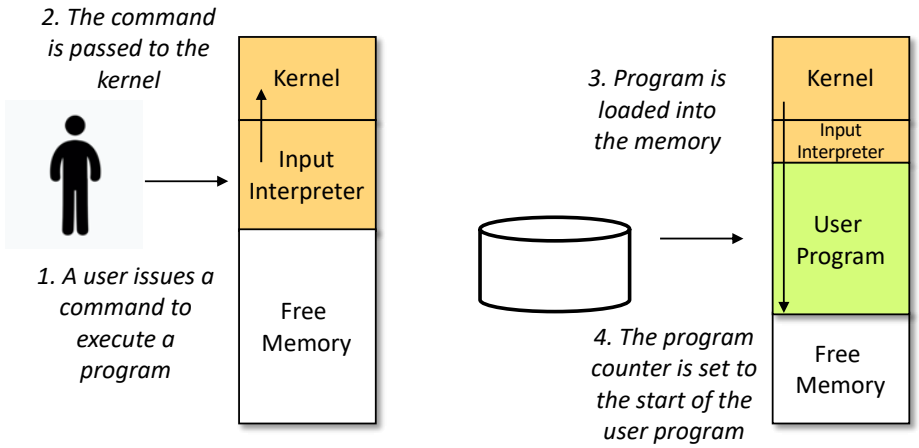
Disk Operating System (DOS) is a famous operating system for IBM PC compatibles. DOS consists of Microsoft MS-DOS as well as the IBM version called PC DOS, both released in 1981. It was also adopted

for the Apple II series of microcomputers. A version of DOS had remained part of the graphical Windows OS series (3.1 and 9x). It provides a command-line interface.

MS-DOS is turned into freeware status by Microsoft and it is available for download in Github: https://github.com/microsoft/ms-dos

Execution Model of MSDOS

Early versions of MSDOS did not support multi-tasking. One process is loaded in the memory and executed at one time. The following figure shows how MSDOS handles process execution.



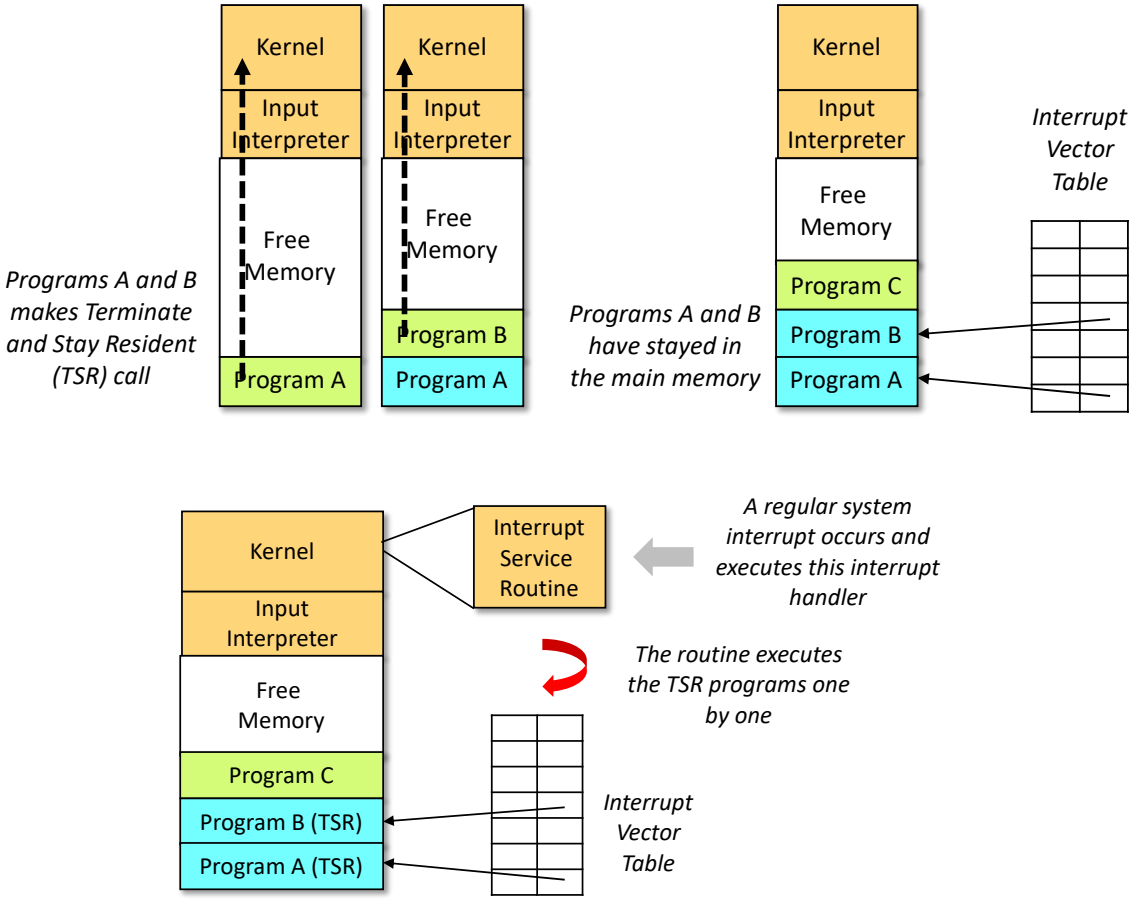
- DOS loads the program code into the memory
- DOS pass the execution to the initial location of the process.
- A part of the command interpreter may be overwritten if there is not enough space.
    - The command interpreter is not needed anyway when the user program is running
    - One part is kept for handling the execution when the process terminates.

Terminate and Stay Resident to simulate Multitasking

DOS supports a pseudo form of multi-tasking with Terminate and Stay Resident (TSR) mechanism.

Terminate and Stay Resident (TSR) is a system call offered in DOS.

- A program already in memory can make this system call so that DOS will keep it in memory.
- The program should have modified the interrupt vector table of system clock so that the vector points to the program.
- The system clock sends an interrupt several times a second.
    - Every time the program is executed because of the interrupt.
    - This creates a multi-tasking situation.
    - This is a good example of using interrupts to drive multiple programs.


Kernel
Input Interpreter
Free Memory
Program A
Kernel
Input Interpreter
Free Memory
Program B
Program A
Programs A and B makes Terminate and Stay Resident (TSR) call
Kernel
Input Interpreter
Free Memory
Program C
Program B
Program A
Programs A and B have stayed in the main memory
Interrupt Vector Table
Kernel
Input Interpreter
Free Memory
Program C
Program B (TSR)
Program A (TSR)
Interrupt Service Routine
A regular system interrupt occurs and executes this interrupt handler
The routine executes the TSR programs one by one
Interrupt Vector Table

COMPS267F Operating Systems

Computing Resources and Operating Systems 2

This chapter discusses the need of management of computer resources and the role of operating systems.

Programs executing on a computer system will make use of computing resources including processor, memory and I/O devices.

- These resources have limited instances and some of them have only one instance.
- If there is only one program running, then the program can have a free hand on all these resources.
- If there is more than one program running, then the programs must compete for these resources so that the program execution can complete.

A major purpose of an operating system is to manage these resources and allocate these resources to different resource utilisers. Operating systems allows computer systems to achieve good performance.

2.1 Computing Systems and Computing Environments

Computing systems have specialized architectures that fundamentally define the performance characteristics of the systems. Different architectures have different configurations of processors and main memory, and how program instructions and data are transferred to processors.

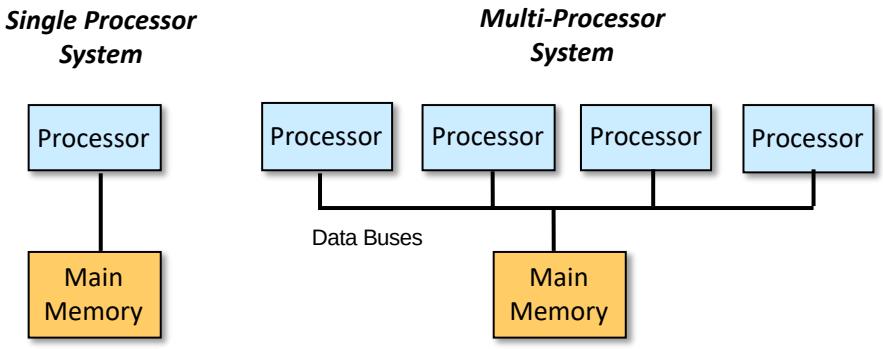
Computer systems can be applied to different environments. Each environment imposes different performance requirements on computer systems.

This section discusses the major types of computing systems and different computing environments.

2.1.1 Types of Computing Systems

The simplest computing system is perhaps consisted of one processor connected to a main memory system. This is known as a single processor system.

- The processor executes instructions of a program.
- The main memory stores programs and data.

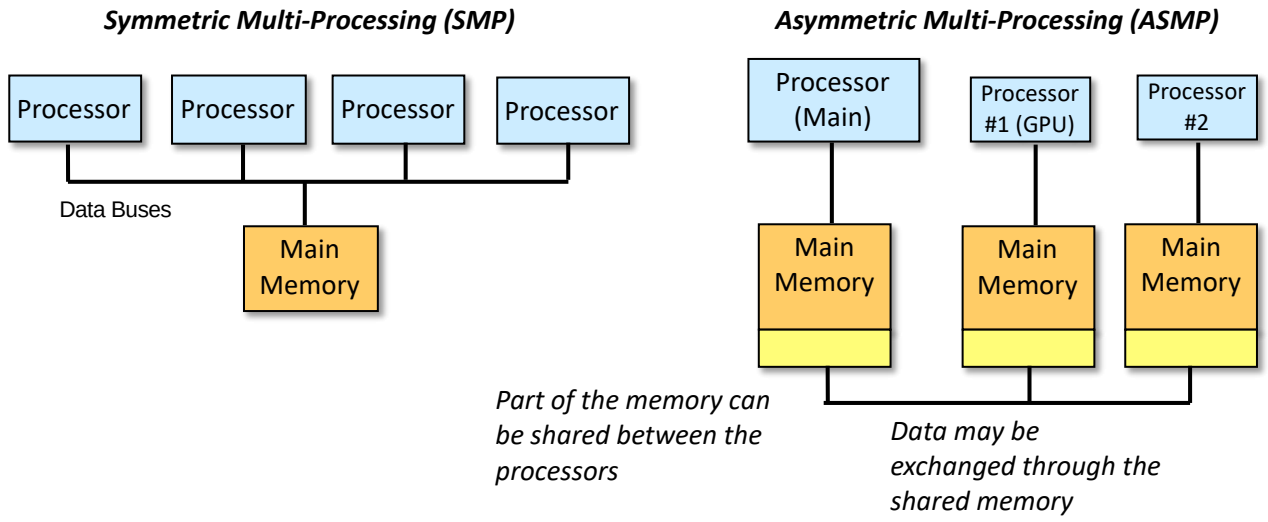


Multi-Processor Systems

A multi-processor system consists of two or more processors. The advantage is the ability to execute multiple instructions at the same time.

There are two types of multi-processor systems.

- Symmetric multiprocessing (SMP).
  - All processors are identical in symmetric multiprocessing and they share the same memory system.
  - There is one operating system managing all the processors.
  - The current multi-core processors are an example of symmetric multiprocessing.
- Asymmetric multiprocessing (ASMP).
  - Each processor has its dedicated role in an asymmetric multiprocessing system.
  - For example, one processor is responsible for integer calculation and another processor is dedicated to floating point calculation.
  - Modern computers often support a dedicated processor in Graphic Processing Unit (GPU).
  - In asymmetric multiprocessing system, there is usually one main processor can access the main memory system and other processors have their own memory space.

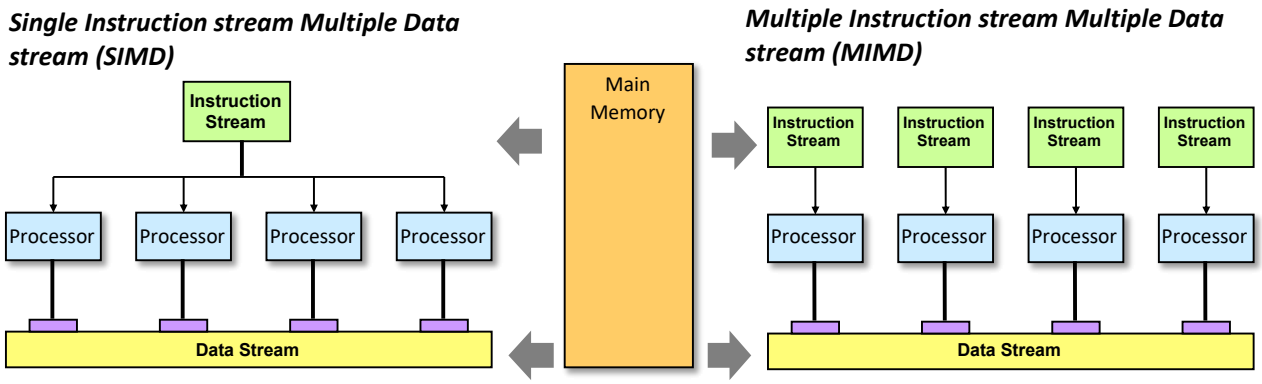


	SMP	***ASMP***
Processors	Should be of the same processor type	Can be different types of processors (i.e. CPU, GPU, etc)
Memory	One main memory system shared by all processors	Each process has its own memory. Data may be shared through specialized mechanisms such as shared memory area
Operating Systems	Single operating system	Single operating system with each processor has their self-contained management process

Program Execution Models in Multi-Processor Systems

There are also two types of multiprocessor systems in the aspect of program execution:

- Single Instruction stream Multiple Data stream (SIMD).
    - All processors execute the same instructions on different data.
    - SIMD computers are suitable for certain types of problems such as image processing and gaming support. This is an important class of supercomputers.
    - The Thinking Machine CM-2 is an example that provides up to 64,000 processors running in parallel.
    - A current example is the Intel extension of Streaming SIMD Extensions (SSE).
- Multiple Instruction stream Multiple Data stream (MIMD). In MIMD computers,
    - Each processor operates independently
    - Each processor executes its own instruction stream (e.g. programs).
    - In a shared memory model, all processors can access the same memory space.
    - The operating system has to maintain the coherency of the memory space.

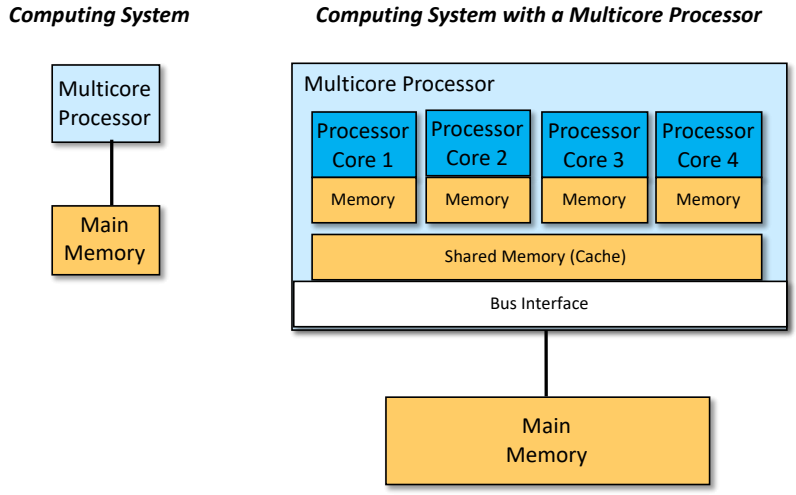


Multi-core Processors

The above has describes that a multi-processor system consists of two or more processors. These processors appear as physically separated units (i.e. integrated chips).

Multicore processors contain more than one processor within the same integrated chip. The integrated chip contains various components that are designed to work very efficient together, including:

- All processors in the integrated chip are normally of the same type.
- There are multiple processor cores with at least two levels of cache memory.
- Each core has its own memory as cache.
- There is a layer of memory shared between all cores.

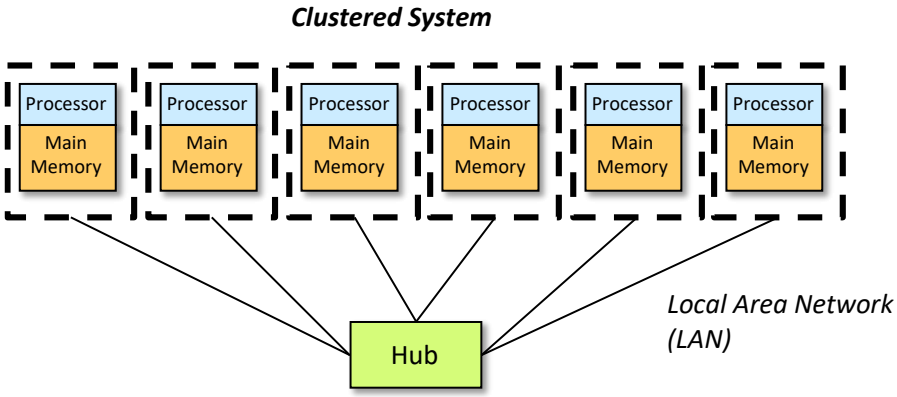


Clustered Systems

In traditional multiprocessor systems, processors and memory are connected by internal system bus through specialized dedicated hardware architecture. The architecture aims to make them work smoothly together. These internal system buses are often sophisticated and expensive to design and build. It is especially costly for high performance systems.

An alternative and more economical solution to obtain high computing capacity is to use a computer cluster.

- A computer cluster is a set of computers connected together by local area network (LAN).
- The many computers are made to appear to be a high performance or high availability single computer.
- Each single computer unit in a cluster has its own processor and memory space.
- It should be an inexpensive desktop computer. Therefore a high performance computer is built using cheap parts.



2.1.2 Computing Environments

Computers operate in different environments. Each environment, due to the usage, imposes different performance requirements. The following lists the major types of computing environments.

- Desktop computers
- Mainframe computers
- Mobile computing

- Client-server computing
- Peer-to-peer computing
- Cloud computing

Desktop Systems

Desktop systems are designed for personal use. They are often known as personal computers.

Desktop systems are not geared towards high throughput. CPU utilization is not a major concern because processors are no longer expensive. Rather desktop systems emphasis on user interactions. They react quickly to user's commands.

Mainframe Systems

Mainframe systems are by defining large-scale computer systems. The power and the size mean that these computers require special housing, cooling, and electrical power. In the old days mainframe systems occupied a dedicated room full of cables, conduits, and ducts.

Mainframe computers emphasize on large throughput.

- Throughput means the amount of work done.
- Large throughput refers to processing a large volume of data and making lots of calculation.

Because a large amount of data is involved in input and output, additional computer systems are connected to mainframe computers to handle input and output. The mainframe computers can concentrate on calculation and processing.

Mobile Systems

Mobile devices are usually significantly less powerful than other types of computer systems. They include mobile phones, PDAs, cameras, and even mobile computers. A key feature of mobile devices is network connectivity. Input and output capabilities are the major limitations.

Operating systems are designed specifically for mobile devices because of the above characteristics. Usually mobile operating systems are smaller and capable of handling special types of input.

Mobile operating system is a fast-growing area. The famous mobile OS include Symbian OS, Blackberry OS, iPhone OS, Windows CE, and Android.

Client-Server Systems

Client-server systems are a very common configuration of multiple computers working together. Clients and servers are roles assigned to the computers. Usually a very powerful server provides services to many clients.

The most renowned client-server system is the world-wide web in which web servers are providing HTTP based services to many web browsers acting as clients.

- The server should be a very powerful computer that is capable to doing a lot of transactions within a short time.
- The client can be relatively modest in computation power. These are often called thin clients.

Peer-to-peer (P2P) Systems

Peer to peer systems are also multiple computers working together. The difference from client-server systems is that there is no centralized service point. All computers are simultaneously servers and clients.

Cloud Systems

Cloud computing is about the provision of processing and computing facilities over the Internet. Users can requisite various computing resources from various places for a task. Computing resources may include computing power, memory, storage, and applications.

The cloud itself is often a sophisticated myriad of powerful processors, main memory, secondary memory, and many applications and platform. These computing resources are shared between many users and many programs. The operating environment requires effective management of these resources and support of users in what they want to do. Security is also an important requirement.

Summary

The following table summarizes the key performance requirements of each computing environment.

	Performance Requirements	***Hardware Configurations***
Desktop	Quick response to users Multi-programming	Flexible. Depending on user requirements
Mainframe	Large throughput	Large number of processors. Large I/O throughput for incoming and outgoing data.
Mobile	Network connectivity Quick response to users Good user interface	Flexible. Usable interface.
Client-Server	Internet connectivity Quick response to users (Server) Large throughput (Server) Multi-programming (Server)	Server should be powerful with large number of processors, large I/O throughput, and large amount of memory.
Peer-to-peer	Network connectivity	Flexible.
Cloud	Adaptable to different performance requirements Security	Similar to Servers. Flexible.

2.2 Responsibilities of Operating Systems

Modern operating systems have a wide range of responsibilities.

- Process management
- Main Memory Management
- File Management
- I/O Management
- Secondary Storage Management

- Networking Support
- System Protection
- Command-Interpreter User Interface

Process Management

A program being executed is known as a process. A process is a logical entity that includes the resources needed for the execution of a program on a computer system.

The OS is responsible for the following process management services:

- Process creation.
- Process termination.
- Process suspension and resumption.
- Inter-process coordination and communications.

These services cover the life cycle of the execution of programs on a computer system.

Main Memory Management

The main memory is an array of storage units. Usually the smallest unit is byte (8-bit), and each byte has an address. In other words, each piece of data found in the main memory is directly addressable.

Usually the smallest unit data read/write from memory is a word. This word size is shared by many things in a particular architecture including the registers in the processor and the basic unit of operations.

Certain addresses are read-only memory (ROM) or spaces for I/O operations (memory mapped I/O). The main memory itself is usually volatile and the data is loss at power off.

The OS is responsible for the following main memory management services:

- Keep track of which parts of memory are currently free or in use, and the ownership of the memory.
- Allocate and de-allocate memory space appropriately for loading and executing process, and data usage.
- Swapping in and out of the memory space associated with processes.

File Management

File is a collection of data under a file name. Other attributes of a file include: owner, permission, timestamp, and other indicators of its purposes. The data could in fact be program machine code.

The OS is responsible for the following file management services:

- File creation, deletion, and attribute change.
- Directory management.
- File backup.
- File mapping of I/O devices.

I/O System Management

I/O devices vary significantly in their purposes and properties speed, control and access, data transfer, and others. It is difficult to deal with such a great disparity of devices difficult to write programs and control them.

One approach adopted by the OS is to hide away the differences, and to present a uniform interface.

The OS is responsible for the following I/O system management services:

- Uniform device driver interface
- Specific drivers for I/O devices.
- Buffer for I/O devices.

Secondary Storage Management

Hard disk is the major secondary storage used in current computer systems. It is secondary because it is used to support the main memory by providing a backup.

The OS is responsible for the following disk management services:

- Free space management and storage allocation.
- Disk scheduling.
- Performance management.

Networking

A modern computer needs connection to the Internet so that data can be shared and obtained. Apart from sharing data, it is possible to share the execution of a task across a number of computers distributed systems.

A distributed system contains a number of heterogeneous computers connected together in a single logical system. A number of architectures exist: client-server, peer-to-peer.

Many protocols exist to allow communication to happen. More advanced protocol makes network connection transparent. For example, changing from File Transfer Protocol (FTP) to Network File System (NFS).

Protection System and Security

The OS controls access of processes and users to various users and system resources.

Authorization, authentication and auditing are the pillars of system security. Examples of areas that are supported by the OS are file access and execution.

Command Interpreter Systems

A user interface is provided for executing programs and managing resources on the computer systems. It is also provided to programmers to get easy access to system resources.

A program known as command-line interpreter or a shell can read commands and interprets control statements, and then executes them. Modern OS provides graphical user interface as the shell, making it more user friendly.

2.3 Services Provided by Operating Systems

Modern operating systems provide an environment for the execution of programs. To achieve this aim, OS provides a range of services to programs and the users of the programs. We can derive a set of desirable services that a program and its user want by analysis.

System Calls

System calls provide an interface between running process and the operating systems and its services. Systems calls may be made directly from a higher-level language programs - WIN32 API, .NET, ANSIC. It is usually an abstraction of a set of procedures. Making one system call performs a series of procedures.

For example, a system call for copying one file has abstracted away many details.

Types of system calls include:

- Process management.
- File management.
- Device management.
- Information maintenance.
- Communication between processes and across computers.

System Programs

System programs are another form of services provided by the operating systems. System programs are executable files that can be started through the command interpreter.

- Some system programs are simply user interface to underlying system calls. They hide away the complex details.
- Other system programs are sophisticated applications such as server listening for connection on the network.

The command interpreter is a most important system program. It allows the users to control a computer through giving commands.

System programs exist as files. The execution of a program begins with loading the file content, which is the program code, into the main memory. Then the operating systems pass the location of the program code to the CPU. Finally, the processor can begin executing the program code.

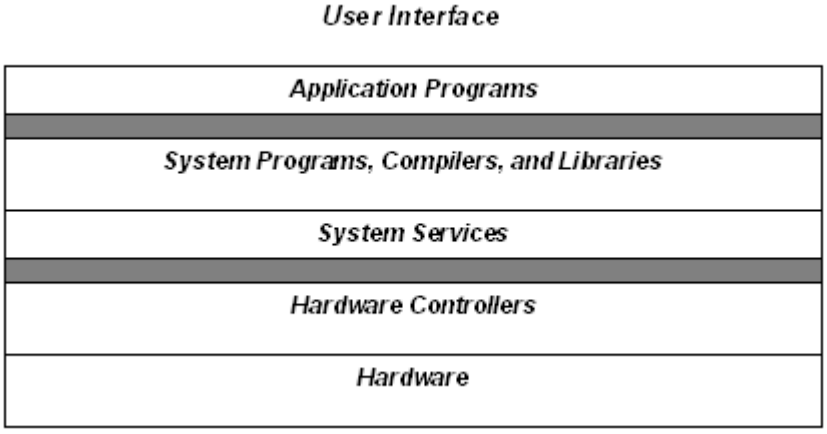
2.3.1 Structure of Operating Systems

There are many approaches in implementation of operating systems.

The OS is a large system. Design practices applicable to large system development are also suitable to OS development. The conventional and modern software engineering practices: modular approach, layered approach, component-based approach.

Layered approach is useful that allows stepwise development from the bottom up. The correctness of the lower layers can be assured independently and provides a platform for the development of upper layer.

The operating system also improves portability. Building an OS for another hardware involves the development of the lowest layer.



The Kernels

Kernel is the core of the operating system services. In UNIX, the kernel contains everything below the system call interface but above the hardware.

Some OS designs supports many functionalities and makes the kernel large and becoming difficult to manage. One approach is to reduce the function of kernel to minimal.

The micro-kernel approach minimizes the function suite to the essential minimal and moves other programs to the system level and application level.

System Generation

The OS must be configured for each specific computer because the configurations or parameters may differ from one computer to another.

System Generation (SYSGEN) is a process that configures an OS for the use on a computer, often interactively with an operator.

- OS may be distributed on a CDROM, USB or downloaded package.
- SYSGEN reads the files from the media and moves the software on the secondary storage on the computer.
- SYSGEN probes the configuration of the computer for information of what devices are used and their attributes: such as the CPU, memory, peripheral devices, and also user preferences.
- Several approaches of configuration of an OS.
    - Re-compilation after changes to the source code.
    - Module selection and composition at build-time.
    - Dynamic module selection at run-time.

COMPS267F Operating Systems

Process Management 3

This chapter discusses execution of multiple programs on a computer system. Operating systems are responsible for the effective use of computing resources for program execution and for the efficient execution of programs.

❖ ***Definitions***

Multi-processing

A computer system has more than one processor or CPU.

Single-programming (or uni-programming)

A computer system allows the execution or running only one program at a time. The next program must wait for the termination of the current program.

Multi-programming

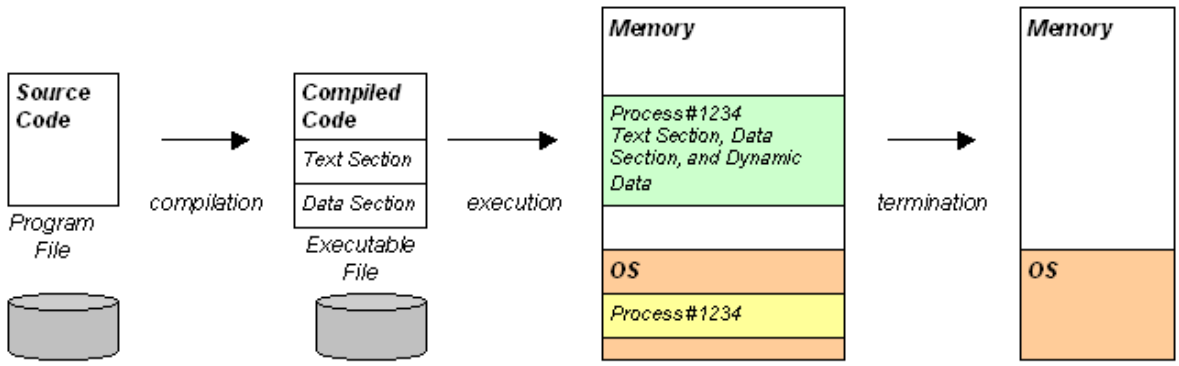
A computer system allows the execution of more than one program at a time. Multiple programs can be running together.

3.1 Execution of Multiple Programs on a Computer System

Process is a program when it is in execution.

- A process is an abstract entity created to model the state of a running program.
- A process is created if the operating system receives a command to execute a program.
- If a program is executed twice, two processes are created.

The following figure shows the birth and death of a process.



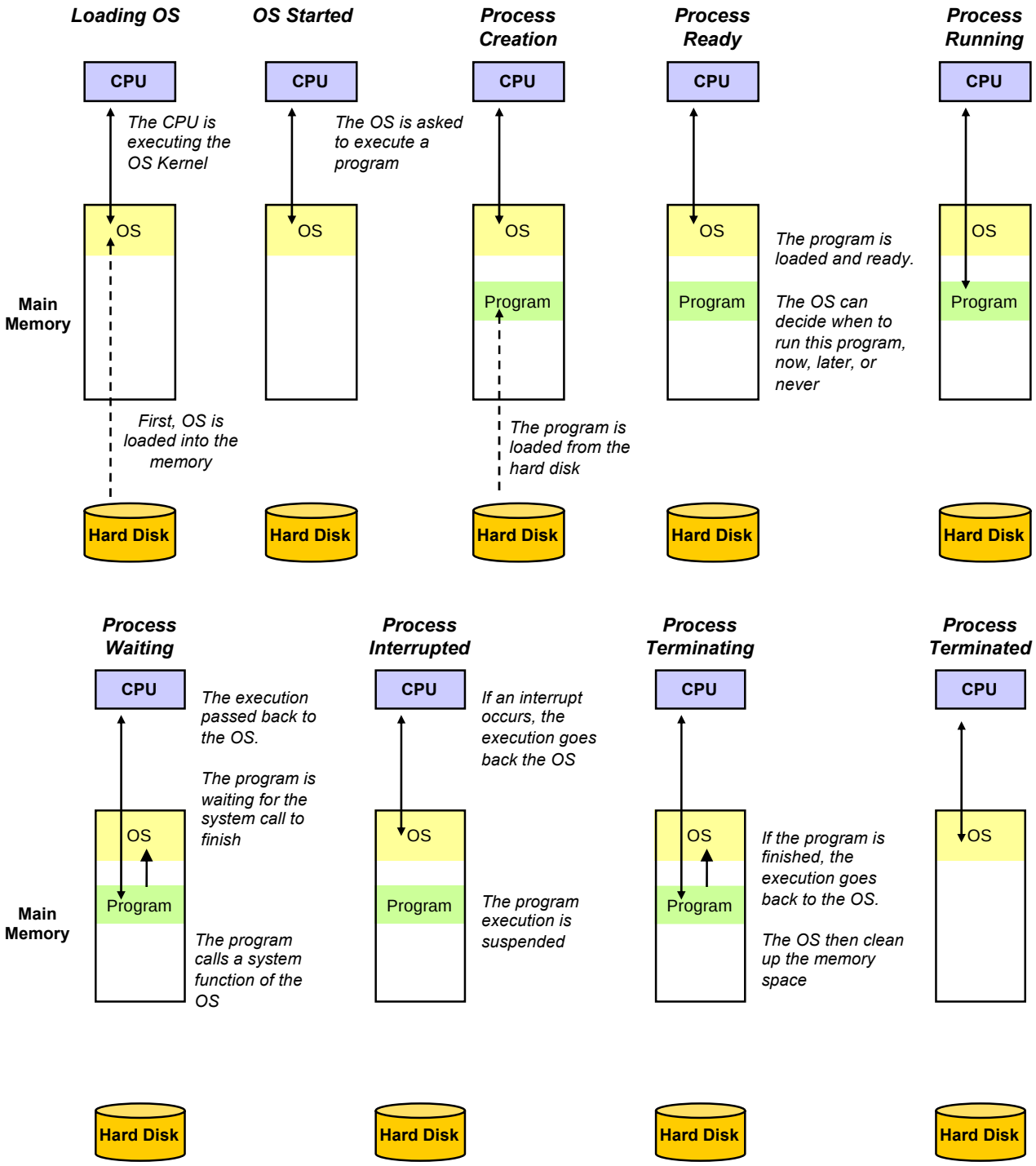
A program is an executable file, usually stored in the secondary memory.

- The file contains compiled code or machine code. The machine code is executable on the native processor or CPU.
- The file must be marked executable.
  - In Windows, the file should have .exe as the suffix in the filename.
  - In UNIX, the file should have the attribute x set.
- The file should also contain data associated with the programs (i.e. constants).
- The file can be executed interactively or through a command shell
  - Double-click on a file through a Graphical User Interface (GUI)
  - Select Applications on the Start Menu (Windows) or Application Launcher (MacOS). The Applications are actually executable files.
  - Typing the filename in a command shell.

3.1.1 Life Cycle of a Process

The life of a process involves the following stages.

- Program execution. The OS receives a command to execute a program.
- Loading. The OS looks for available space in the main memory. The OS loads the code and data from the program file in the main memory.
- Execution. The OS passes the control to the first instruction of the program code. From this point onwards, the processor (the CPU) will be executing instructions from the program.
- Interrupted. From time to time, the control is passed back to the OS because of interrupts. Interrupts may be caused by regular clock as a means to force the control to return back to the OS. Other interrupts may be due to the program performing input and output operations.
- Termination. The process is to be ended. The process ends if the execution reaches the end of the program, or the program calls exit system function. Termination may also be caused by external commands.
- Clean Up. The OS releases the part of the main memory that is previously occupied by the terminated process. The OS also releases other resources related to input and output operations.



3.1.2 A Secret of Multiprogramming

Multi-programming means that more than one process can exist at any one time on a computer system. The following questions naturally emerge.

- How can it happen?
- How can two programs running at the same time on one processor?

Multi-programming can happen on computers with a single processor with a single core.

- A running process is not always *running*.
- A running process is not always using the processor.

A program usually has the following three things to do:

- Its instructions are being executed by a processor
- It is doing input and output (I/O), or waiting for I/O to complete
- It is simply waiting

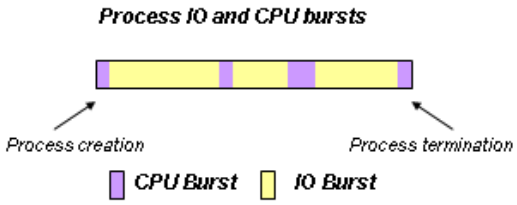
Many modern computer programs are spending the majority of time on I/O.

- Modern programs are mostly interactive. Users are slow. Programs have to wait for user responses.
- GUI operation is often idle.
- Internet operations are slow. Programs have to wait for data to arrive.

Only a few types of programs use a lot of processor (CPU).

- Training an artificial intelligence (AI) machine learner.
- Doing scientific calculation (i.e. simulation of weather).

The following figure shows a typical IO-bound program, referring to programs that spend much time on I/O.



- IO burst refers to the time when a program is doing or waiting for IO.
- CPU burst refers to the period when a program is controlling the CPU (that is the CPU executing the program).
- Generally speaking, programs execution involves alternative periods of CPU bursts and IO bursts.

IO-bound and CPU-bound describes the performance characteristics of a process.

- IO bound is labeled on a process if its performance is mainly influenced by IO performance because it spends most of the time doing or waiting for IO operations.
- CPU bound is labeled on a process if its performance is mainly influenced by CPU performance because it spends most of the time executed by the CPU.

Multiprogramming is possible on a single core CPU because other programs can be waiting or doing IO while one program is using the CPU core. Almost all programs running on desktop computers are IO-bound. Each program spends almost all the time doing IO. Therefore even

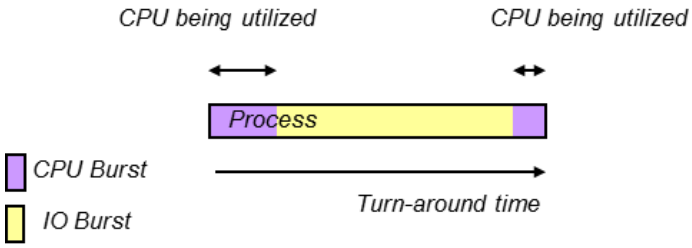
3.1.3 Benefits of Multiprogramming OS: CPU Utilization

A single-programming OS means wastage of money on a computer

- Only execution of one program is allowed at one time. It cannot fully utilize the CPU.

- An IO-bound program spending 90 percent of the time doing IO. The CPU is only in use 10 percent of the period.
- The CPU is said to be under-utilized. Much money spent on the CPU is wasted.

A multi-programming OS can improve CPU utilization drastically. CPU utilization is the proportion of time that the CPU is doing the work of executing programs.



Consider a set of processes that will spend 90% of the time on IO bursts are running on a single processor.

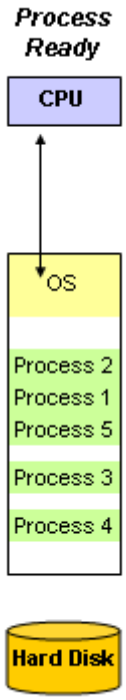
- We can run 10 such processes at the same time.
  - The chance of all 10 processes doing the IO is low, at least one of them should want to do CPU burst.
- If there is at least one process that is ready to use the processor, the processor will be utilized.
  - The more programs there are, the more likely that one of them is in CPU burst.

A high CPU utilization makes the money spent on the processors worth more. It requires keeping the processors active as much as possible.

- Increase the number of processes that are ready to be executed.
- Reduce the time needed for setup of the process to be executed.
  - There is a previous process that needs to be removed from executing on the processor.
  - Some work needs to be done when switching from one process to another

To achieve the above, a solution is to load the multiple programs in the main memory when they are executed.

- A program must be loaded in the main memory for execution by the processor.
- Loading of programs take time.
- If it is not the turn for a process to run on the processor, the process should be waiting in the main memory.
- Running the next process involves changing the program counter of the processor.
  - This step alone is very fast.
  - However, running the next process also involves other work known as context switching.
  - The OS is responsible for context switching, and needs effort and time to do it.



A multi-programming OS can execute many programs at a time.

- At most one of these programs is actually being executed by the processor (two programs if there are two CPUs) at any one time.
- Program execution in a multi-programming OS does not necessarily mean that the CPU is executing the program. It means that the program is loaded, the process is created, and it is ready to be executed by the CPU.
- A multi-programming OS can execute only at most a certain number of programs at a time due to memory size.
- Memory size is not unlimited and it can accommodate the loading of a fixed number of programs only.

3.1.4 Example 1: CPU Utilization and Turn-Around Time

A set of five processes is to be executed on a single processor computer system. Each process has the following characteristics: the turn-around time is 10 ms, and it spends 20% of the time in CPU bursts and 80% in IO bursts.

Calculate the CPU Utilization and the overall turnaround time of the five processes.

❖ ***Definition***

Turn-around time

Refers to the time between the creation of a process and the termination of a process.

It is the time taken to complete the execution of a process. For users, the shorter it is, the better. The work can be completed faster.

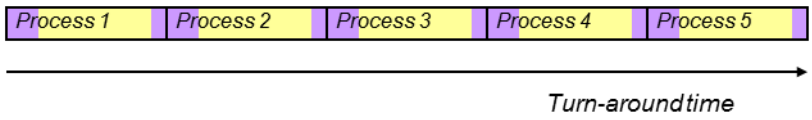
CPU utilization

The ratio of the time that the CPU is actively executing programs.

The higher the CPU utilization rate, the more useful computation work can be cajoled out of a computer system.

Solution

The processes must be executed in the following pattern.



Each process is executed one after another. Only after the first process has completed execution, the execution of the second process will begin.

Total turnaround time is 10 ms (each process) x 5 = 50 ms

Each process spends 10 ms x 20% = 2 ms using the CPU and 10 ms x 80 % = 8 ms doing IO operations.

Totally the time the CPU is being used is 2 ms (each process) x 5 = 10 ms.

CPU Utilization = 10 ms / 50 ms = 20%

This is a low utilization rate.

3.1.5 Example 2: CPU Utilization and Turn-Around Time

Consider the above example again.

Calculate the CPU Utilization and the overall turnaround time of the five processes if multi-programming is supported.

Solution

Assumption #1: Ignore the time taken by the OS to do work in executing the processes.

Assumption #2: Many IO operations cannot happen in parallel.

The OS will execute all five processes at the same time. The OS first loads the program of these five processes into the main memory.

There is only one CPU and so the OS will select one of the processes in CPU bursts to use the CPU. When the process selected has finished one CPU burst and doing IO operations, the OS will select another process in CPU burst to use the CPU.

The OS tries to arrange the five processes so that the CPU is not idling.

Total CPU burst time of 5 processes = 2 ms (each process) x 5 = 10 ms

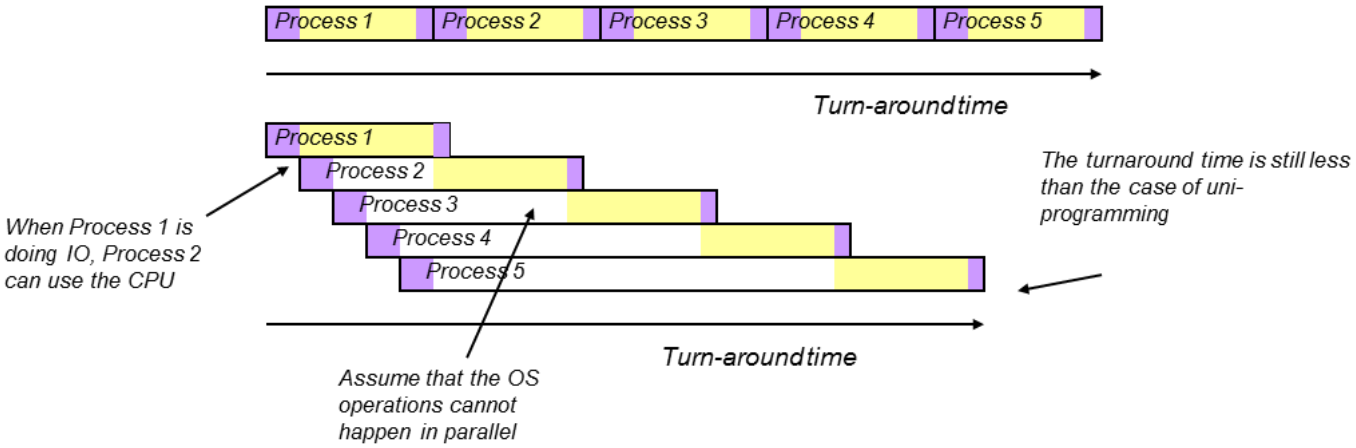
Total IO burst time of 5 processes = 8 ms (each process) x 5 = 40 ms

The CPU burst time can happen in parallel with the IO burst time, except for the first process.

Turnaround time = 40 ms (IO burst time) + 2 ms (CPU burst time of first process = 42 ms

CPU Utilization = 10 ms / 42 ms = 24%

The turnaround time is shortened and the CPU utilization improved.



3.1.6 Example 3: OS Process Management Overhead

Consider Example 2 above.

The above calculations have ignored the effort of the OS in context switching.

- The OS has to carry out a number of tasks such as loading the program code into the main memory, managing the processes, switching the CPU control from one process to another.
- These are overheads in process management.
- The CPU utilization is reduced and the turnaround time is increased as a result.

In a simplified model, the multi-programming OS spends on average 1 ms per process in overhead operations such as loading programs. Assume that these overhead operations of the processes cannot happen in parallel.

Calculate the turn-around time and CPU utilization for the situation of Example 2.

Solution

The total time for the OS to manage the five processes = 1 ms x 5 = 5 ms

Turnaround time becomes = 40 ms (IO burst time) + 2 ms (CPU burst time of first process) + 5 ms (overhead) = 47 ms

CPU Utilization = 10 ms / 47 ms = 21%

The turnaround time is longer and the CPU utilization reduced.

3.2 Process Management

A multi-programming OS needs to do a lot of work related to execution of multiple programs.

The OS needs to know the state of every process so that it can allocate CPU and memory resources to each of the process. A process can be in any of the following states.

- It is being loaded and created.
- It is ready for execution and waiting for the CPU.
- It is controlling and using the CPU (in actual execution).

- It is doing or waiting for IO.
- It is being terminated.

The OS also needs to know the state of every resource type, including the CPU, main memory, and IO devices in a number of situations,

- The amount of spare main memory for process creation.
- The availability of the CPU for process execution.
- The availability of IO devices for assigning processes in IO bursts to wait or to use the device.
- The resources assigned to each process, so that the resources can be released back to the system after the termination of processes.

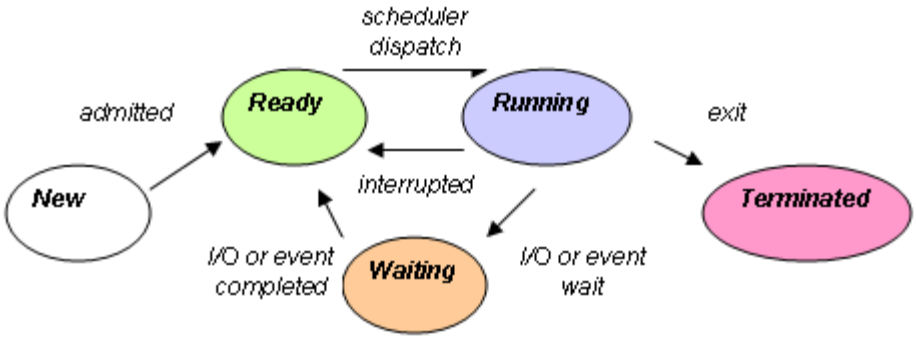
Process management is a most important responsibility of multi-programming OS. This is achieved by the following:

- Defining the process states.
- Defining the conditions of transitions from one state to another state.
- Defining the data structures to keep track of information related to processes and process management.
- Designing algorithms to manage the states of the processes to achieve high system performance.

3.2.1 The Five-State Model of Process State Transition

The five-state model of process state transition defines five states to represent the different stages of life of a process.

- New. The process is being created.
- Running. The instructions of the process are being executed by the CPU.
- Waiting. The process is waiting for IO operations.
- Ready. The process is ready to be executed. The loading of the program into the main memory has completed.
- Terminated. The process has finished execution.



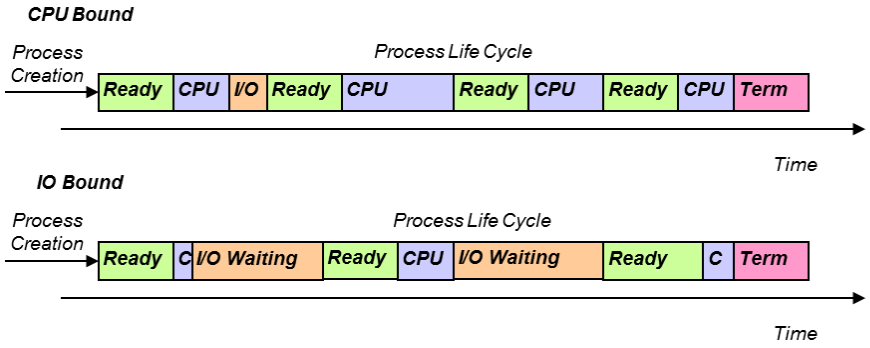
Process management of OS involves moving processes from one state to another.

- New to Ready. A process is ready after the program code is loaded into the main memory.
- Ready to Running. The OS passes the control of the CPU to this process, allowing the execution of the process' instructions.

- Running to Waiting. The process is executing an IO operation. It will take some time and the OS takes the CPU control from this process. The process will wait until the IO operation has completed.
- Waiting to Ready. The process' IO operation has completed. The OS receives the interrupt and moves the process into the Ready state.
- Ready to Terminate. The OS receives the signal that a process is terminated. The OS then carry out the resource release and clean up in the Terminated state.
- Running to Ready. The OS can choose to pre-empt a Running process, remove its CPU control, and put the process back to the Ready state.

This five-state process management model is one of the several process management models you will find in the literature. There are more basic and advanced models developed for computer systems, such as 3-state, 4-state, or 7-state models.

The following shows the life cycle of processes from the perspective of the five-state model.



- A process will go through different states in its life cycle.
- CPU bound processes may spend more time in the Running and Ready state.
- IO bound processes are likely to spend more time in the Waiting state.

3.2.2 Several Key Concepts in Process Management

This section goes through several key concepts:

- Process Control Block (PCB)
- Program Counter and the Context
- Context Switching
- Ready Queue

Process Control Block (PCB)

Process Control Block is the data structure that contains important and dynamic information about the state of a process.

- The OS needs to keep track of the state and information for all the processes. For example
  - Which state a process is in?
  - Where is the process in the main memory?

  - Whether the process is doing IO? What IO operation is undergoing?
- The OS needs to store data of the processor, such as the values of the registers (including the program counter), when a process is removed from the processor.
  - The stored values will be copied back to the processor when the process resumes execution.
- The OS needs to keep accounting information such as how much time has lapsed.

The content of each PCB is described in the following table.

Data	***Descriptions***
Process State	New, Ready, Running, etc.
Program Counter	The position of the program where the CPU is executing.
CPU Registers	The data involved in CPU calculations and data manipulations are saved here: accumulators, index registers, stack pointers, condition code (overflow, carry, etc).
CPU Scheduling Information	Process priority, queues, and other scheduling information.
Memory Management	Base and limit registers, and other information for memory management (to be covered later).
Accounting Information	Amount of CPU time used, quotas, and other book-keeping information.
I/O Status	Keep track of the I/O status: lists of open files.

- PCB is a table shape data structure.
- PCB is usually stored in the main memory.

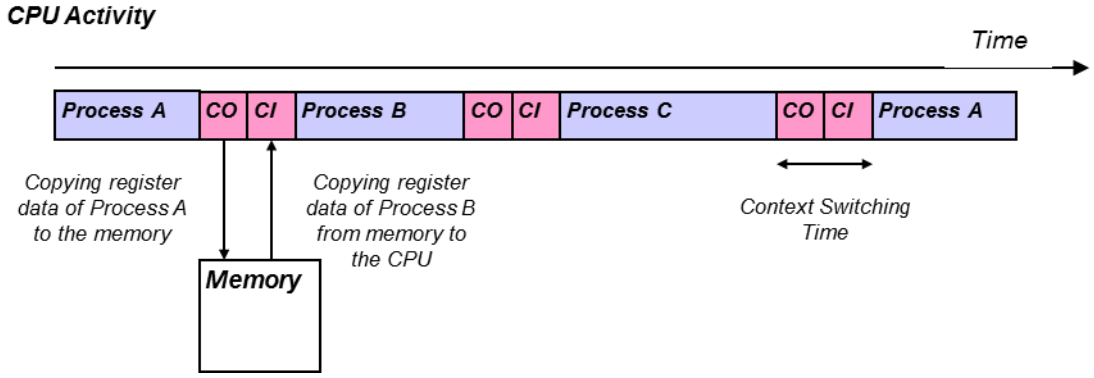
Program Counter and Context

The program counter is stored with the PCB because a process may be removed from using the processor.

- The process may be relieved of CPU control and moved to Ready or Waiting state.
- The program counter, indicating which address the CPU is executing, will have to be stored away.
- The process will in the future resume in the Running state and it should restart at the address previous stored in the program counter.

After the process is relieved of CPU control, another process will control the CPU and so the registers in the CPU will hold values relevant to the new process.

- The storage of the program counter in the PCB is essential because otherwise the PC in the CPU will be replaced.
- Other CPU general-purpose registers should also be stored away due to the same reason.
- The set of data in the CPU registers is known as the context.



Context Switching

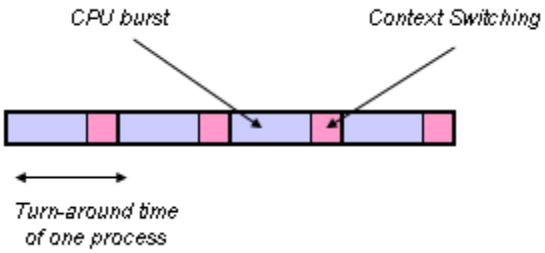
Context switching is a necessary procedure to allow multiple processes to share a processor (CPU). A process that is controlling the CPU would do some data processing and calculation. There are results and data stored in the CPU registers during processing.

For example, for the process to perform X = A + B, the data of A and B must first be loaded in the CPU first, and eventually the result X is available in the CPU as well.

The set of data in the CPU registers is known as the context.

Context switching refers to the operations that ensure a process can execute with the correct set of data in the CPU registers.

- For a process to be taken away from the CPU and then later to be given back CPU, then the CPU data relevant to the process must be stored away first.
- When the process returns in the future and to resume the operation, the data must be loaded back to the CPU registers.



The context data to store include mainly the program counters and various registers.

- They are commonly stored in the process control block or a switch frame.
- The time taken in context switching is an overhead factor in OS process management.
- The CPU is involved in executing the context switching procedure and unable to do the useful work of executing user processes.

Dispatcher

The dispatcher is the module that executes context switching between processes.

- Copy CPU registers to the Process Control Block (PCB) of the outgoing process.
- Copy the value from the PCB of the incoming process to the CPU registers
  - This step sets up the correct context for the execution resumption of the incoming process.
- Switch to user mode (the scheduler has been running in monitor mode).
- Set the program counter appropriately to start executing the incoming process.

The time is important for the dispatcher to be efficient so to reduce the overhead of process management and improve CPU utilization.

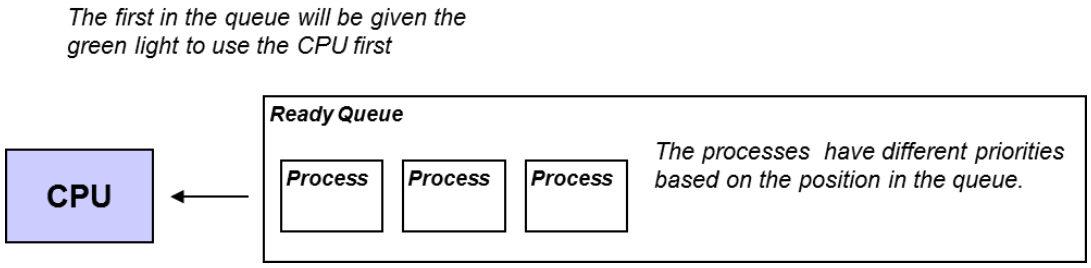
Ready Queue

The ready queue is a data structure that stores processes (i.e. their references) in a queue.

- In a multi-programming OS, many processes can be in the Ready state.
- The processes should be managed to use the CPU in a regulated manner.
- The ready queue imposes a priority on the waiting processes.

The OS has the power to control which process is given the CPU to use.

- A queue structure will allow the control process based on some orderly conditions such as first-come-first-serve.
- When the CPU is available, the OS will choose one process from the ready queue and give the CPU control to the process.



3.2.3 Example 4: CPU Utilization with Context Switching

A set of four CPU bound processes (named P1, P2, P3, and P4) share the CPU. Assume that the CPU burst time of each of these processes is 10 ms long. The context switching time is 1 ms. If these CPU bound processes take turn to control the CPU that the next process comes on when the last process has terminated. Estimate the CPU utilization.

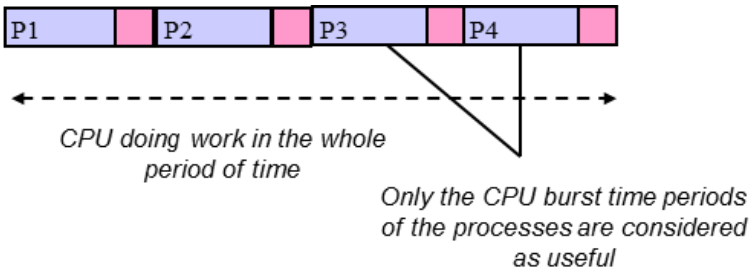
Solution

For each process, the CPU burst time is 10 ms and context switching is on average 1 ms.

The total CPU work time is 10 ms + 1 ms = 11 ms.

The CPU burst time is the time CPU doing useful work.

CPU utilization = 10 ms / 11 ms = 91%



3.2.4 Example 5: CPU Utilization with More Frequent Context Switching

Consider Example 4. If the OS decides that each of the four processes will not complete its execution in one go. Instead, each process will execute its CPU burst time in two steps. So the sequence of executing processes becomes P1 – P2 – P3 – P4 – P1 – P2 – P3 – P4. Estimate the CPU utilization.

Solution

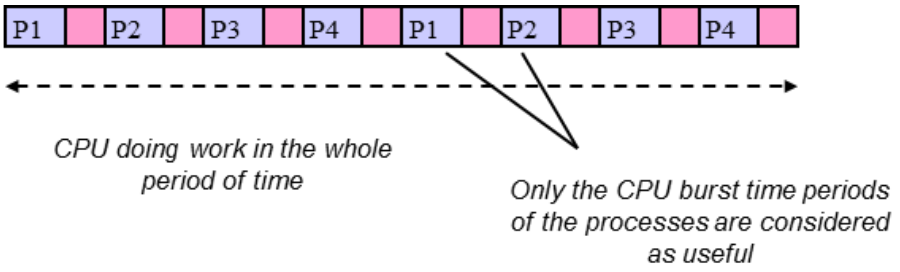
For each process, there are now two context-switching on average.

Sum of turnaround time of all four processes = 10 ms x 4 + 1 ms x 2 x 4 = 48 ms.

The CPU burst time is the time CPU doing useful work.

CPU utilization = 10 ms x 4 / 48 ms = 83%

The CPU utilization reduces if there is more switching between processes.



3.2.5 Process Creation, Process Hierarchy and Process Termination

The OS is a program, and it should exist as a process during operation.

In fact, modern OS is sophisticated, and it has a set of processes doing different tasks.

Process must be created from another process, except the very first one.

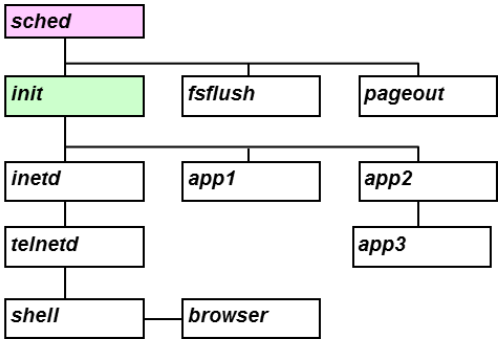
- New processes are created from the initiation of existing process.
- There is a relation between the new process and the existing process.
- The new process is called the child process, and the original process the parent process.

The creation of new processes is done through calling the system function `fork`.

System boot is a set of procedures when a computer system is started up. The last step in boot up sequences includes the creation of the first process.

In UNIX, the system creates the first process called sched that is responsible for the scheduling of other processes.

- The process sched creates a process called init.
- The process init then creates all other system processes such as the shell process.
- The shell process allows users to login and create more processes.



The following shows an example program that creates a child process from a parent process.

The program code for both the parent process and the child process seems to exist together. The program is written in a way that the parent executes one section and the child executes another section.

```
#include <stdio.h>


int main() {
  int pid;
  pid = fork();
  if (pid < 0) {
    fprintf(stderr, "Fork Failed");
    exit(1);
  } else if (pid == 0) {
    /* section executed by the child process */
    printf("Child is sleeping\n");
    sleep(10);
    printf("Child waiting for I/O\n");
    getchar();
  } else {
    /* section executed by the parent process */
    printf("Parent is waiting\n");
    wait(NULL);
    printf("Parent received that Child Complete\n");
    exit(0);
  }
}
```

The value of the variable `pid` in the parent process and the child process are different.

- In the parent process, the `pid` is non-zero, while the `pid` is zero in the child process.
- The child process is an exact duplicate of the parent process in the address space (memory) except the value of `pid`.

UNIX also offers a way for the child process to load a new executable file to the address space (memory) through the `execlp` system call.

The termination of a process is important for the resources to be released and cleaned up.

- A process can terminate normally (the program ends after the last statement)
- A process can signal its termination by making the system call exit. This may be due to an error caught in the program.
- A process may experience a serious error so it is exited involuntarily such as a pointer error that accesses a memory space out of the program. The OS terminates the process in this case.
- A process can cause the termination of another process (child process) using the system call `abort`.

3.3 Introduction to Process Scheduling

The execution of processes requires two critical resources:

- Main memory. Storing program code and data.
- Processor or CPU. Executing program code.

Both resource types are not unlimited. Processors are particularly scarce. The aim of process scheduling is to decide when and which process can use these resources.

- Loading into the main memory
- Using the CPU

Therefore are two types of process scheduling.

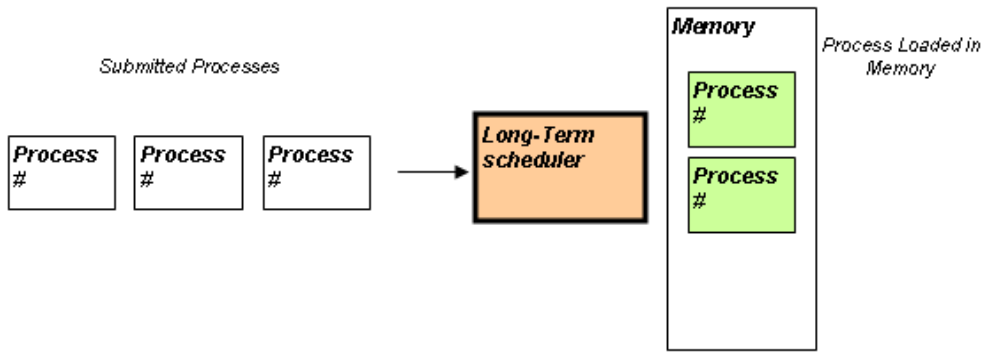
- Long-term scheduling. It controls the number of processes admitted into the ready queue. There are a few factors affecting a long-term scheduler: the size of the main memory that decides the maximum number of processes, the desired level of multi-programming, and the desired system resource utilization.
- Short-term scheduling. It controls which process is given the CPU control. It is also known as CPU scheduling.

3.3.1 Long-Term Schedulers and Level of Multi-Programming

Long-term scheduler selects processes from the submitted processes and loads them into the main memory. It manages the main memory as a resource for program execution.

It is working to satisfy the following:

- The available space of the main memory. If the main memory is fully occupied, then no new process can be admitted.
- The desired level of multi-programming. For interactive or multi-user systems, a high multi-programming level is desirable. It will reduce the CPU utilization because of more context switching. It can however improve use satisfaction with shorted response time.
- The desired system resource utilization. For system with inadequate resources or high performance systems, a high level of resource utilization is desirable. The long-term schedule may choose to reduce the number of admitted process to increase CPU utilization.



The number of processes loaded into the main memory indicates how many processes can be ready for execution.

- Too few processes (may be due to small memory size). The CPU utilization may suffer because there are few ready processes.
  - If there are more processes ready to execute then there is a higher chance that the CPU remains occupied.
- Too many processes. The CPU utilization may also suffer due to increased context switching. More time spent on overhead.
  - The OS may be tempted to switch frequently between the processes so that each process can use the CPU without waiting for too long.
  - The number of context switching increases and so the CPU utilization will decreases.

Having the *right* level of processes is important.

The memory size is also an important factor.

- The number of processes in the ready queue is dependent on the size of the main memory.
- A small main memory can allow fewer processes to be loaded into the main memory.
- A large main memory gives OS to flexibility to make decisions.

The level of multi-programming is the number of processes that are ready for execution in a computer system. It is supported by a more frequent context switching so that each process can have a share of the CPU.

3.3.2 Example 6: Max Number of Ready Processes

Given that the size of the main memory of a computer system is 1M. The OS occupied 350 K of the main memory. Now, there are a number of processes to be created and each process requires 100 K main memory. Estimate the maximum number of processes that can be admitted to the ready queue.

Solution

Remaining free memory space = 1M = 1000K – 350K = 650 K

Maximum number of process that can be loaded into the main memory = 650 K / 100K = 6

3.3.3 Short-Term Schedulers

Short-term scheduler selects processes from the ready queue for using the CPU. Hence, it is also called CPU scheduling.

There are two common strategies taken by short-term schedulers.

- A process may be removed from the CPU at any time. This is called preemptive scheduling.
- Allows a process to use the CPU as long as it likes. This is called non-preemptive scheduling.

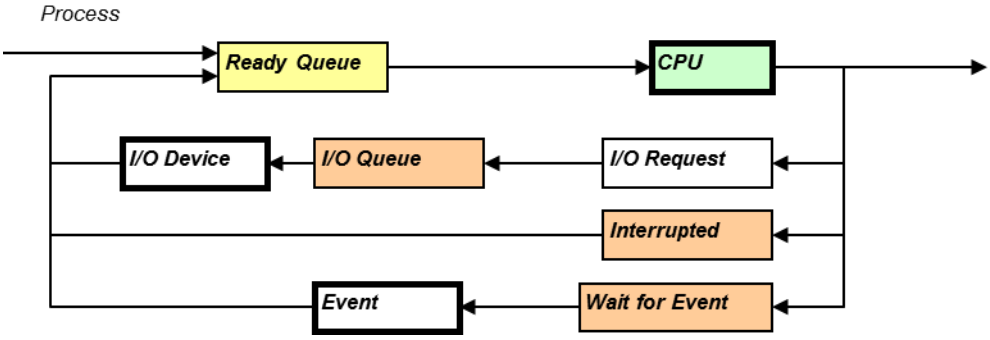
Each action taken to remove the current active process and admit a new process to the CPU involves context switching. A very frequent context switching gives the impression that all processes are using the CPU at the same time.

- The frequent context switching means each process needs not wait for too long for the next turn of using the CPU.

- It allows a process to response to user command quicker.
- It is an important measure of satisfaction in multi-users systems.
- Users always like to have the system responding quickly.

However, frequent context switching increases overhead and reduces CPU utilization.

Short-term scheduler adopts one or more scheduling algorithms to manage the admission and removal of processes from the CPU. This topic will be revisited in the next chapter.



The above figure illustrates how processes are moved around to different states by a typical short term scheduler.

- Select a process from the ready queue.
- Pre-emption of a Running process.
- A process starts an I/O operation and waits for its completion.
- A process gets interrupted and returns to the ready queue (i.e. pre-emption)

3.4 Multi-Threading

The thread of execution is an abstract concept that describes the progress of execution in different locations of a program.

- One principle of programming is that the execution thread runs from the first instruction of a program and finishes after the last instruction of a program.
- Programming constructs like if-else, while, function calls can change the direction of execution thread.

So far we have assumed one execution thread per process. Program execution with one execution thread can cause poor user experience.

- Each execution thread can be understood as a worker. This worker can do one thing at a time.
- If the worker is doing one thing, it cannot respond to other orders.
- For example, if Microsoft Word was single threading, and if you press the button to check grammar, then the only thread will do grammar checking for you. During that time, the Word cannot respond to your typing or other button pressed.
- Another example, Android has an UI thread that runs the Android programs. It can only respond to one event (i.e. pressing of the button) at a time.

Therefore, many programs are written in a multi-threading manner for better user experience.

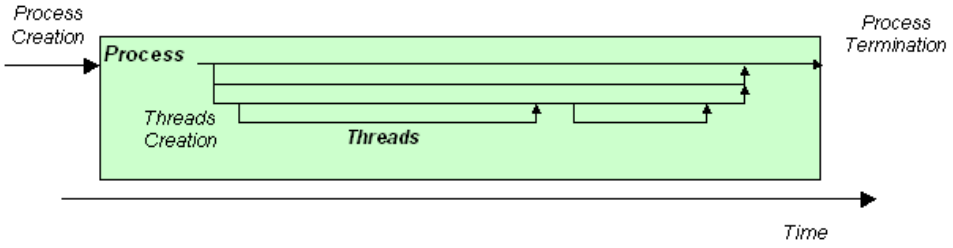
3.4.1 Brief Introduction of Threads

Threads are called lightweight process. The name lightweight succinctly describes the nature of a thread, which uses less resource. Usually, several threads share the resource of a single process.

A thread shares many similarities with a process.

- A thread has its own program counter.
- A thread has its own register set, stack and other state information.

A single process can contain multiple threads that share the same address space, global variables, and other system resources. There are some important implications that the threads of a same process share these resource types.



When a program is written in a multi-threading manner, the process created can contain many threads.

- These threads belong to the same process.
- These threads can access the data of another thread, or even destroy the data.
- The threads sharing the same address space are assumed to be friendly because they must be written in the same program.

If there are multiple threads and a single processor, which thread should run? This is the same question as in the multiple processes case. User-level threads and kernel-level threads are two major methods to implement threads.

User-level threads are managed by user-level libraries and scheduling is done in user-mode.

- It imposes lower overhead because it does not require the kernel to manage the scheduling.
- Each process can has its own scheduling algorithms for executing the threads.
- A severe downside is that all the threads may not have the chance to control the CPU
  - The process they belong is not given the CPU by the kernel.
  - The kernel knows only the processes and not the threads.

Kernel level threads are directly managed by the kernel.

- When a thread is blocked, the kernel can select one thread from the same process or a thread from another process.
- The kernel knows all the threads and may select any thread irrelevant of the process it belongs.

3.4.2 Resource Requirements of Processes and Threads

Threads are lightweight processes because they share resources that belong to their parent processes.

The following table summarises the differences between a thread and a process.

Each Process has	**Each Thread has**
Program counter	Program counter
Stack	Stack
Register set	Register set
Child thread	Child thread
State	State
Address space	
Global variables	
Open Files	
Timers	
Semaphores	
Accounting Information	

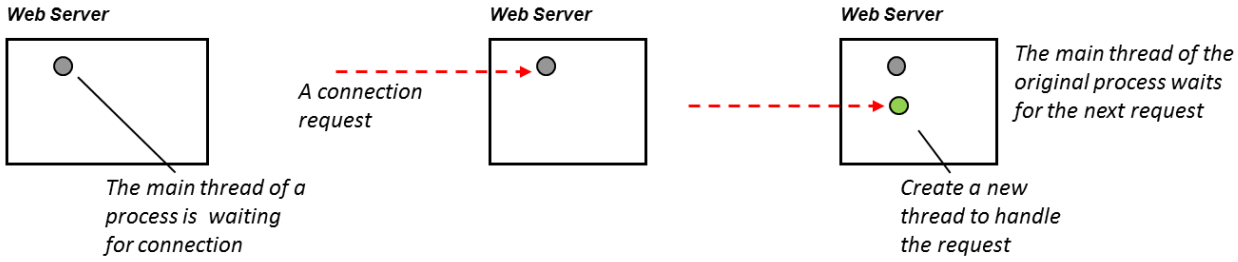
The following table describes the resource types that are owned by a process

Resources	***Descriptions***
Stack	For keeping local variables, parameters, and return values in function calls
Register set	The data involved in CPU calculations and data manipulations are saved here: accumulators, index registers, stack pointers, condition code (overflow, carry, etc).
Child threads	The set of threads belong to this process
State	
Address space	The memory space for the text section, and dynamic memory allocation.
Global variables	The data section of the process
Open Files	Including network connections.
Timers	
Semaphores	For synchronization between threads

3.4.3 Thread Pooling

Consider a web server that is designed to receive incoming requests (from a web browser) and to response with data (i.e. web page content).

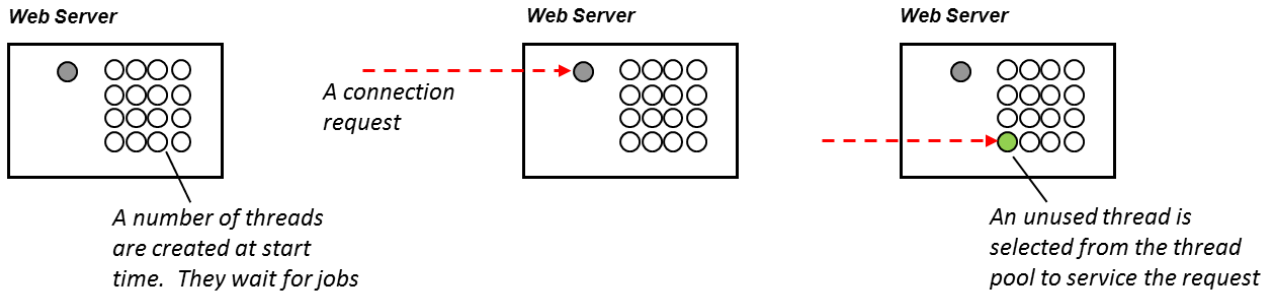
- A busy web server has to handle thousands of incoming requests per second (and more).
- Sometimes, a request may take a long time (e.g. involve image or video processing).
- It is essential that a web server is running in multi-threading manner.



An effective solution is to use a new thread for handling each incoming request. The new thread can handle the request, no matter how long it takes, while the original main thread can listen for the next request.

However, creation and destruction of threads incur overhead. A technique called thread pooling is often used.

- A number of threads are created at the start-up of the web server. It is commonly in the range of a few hundreds.
- The threads are gathered in a data structure known as a thread pool.
- An unused thread from the thread pool is assigned to every incoming request. No thread creation is needed.
- A thread finished its job is returned to the thread pool.

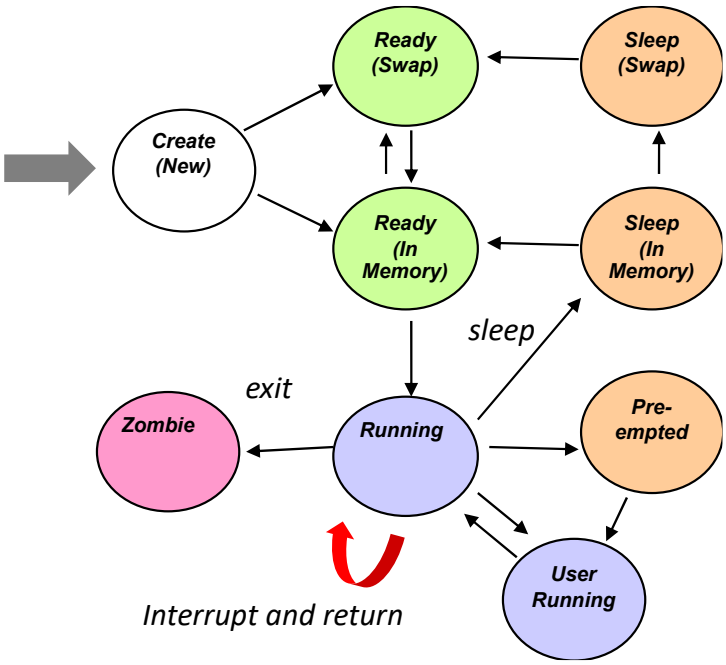


3.5 Case Study: UNIX Process Management

UNIX is arguably the most important OS family that has influenced modern computing. UNIX was designed to be multi-tasking and multi-user. UNIX is the parent of Linux, BSD and Apple MacOS.

UNIX SVR4 operating systems has a nine-state process model.

Create	Equivalent to the New state
Ready (In Memory)	Equivalent to the Ready state
Ready (Swap)	Equivalent to the Ready state but the program memory is swapped out to the secondary memory
User Running	Running state in user mode
Kernel Running	Running state in kernel (monitor) mode
Sleep	Equivalent to the Waiting state
Pre-empted	Equivalent to the Waiting state
Zombie	Equivalent to the Terminate state. The process does not exist, but the record remains for the parent process to clean up



This nine-state model has taken into two mechanisms.

- Program loaded into the main memory may be swapped out to the hard disks. A process may be considered ready but not immediately ready for the Running state.
- The two Running states distinguished between user level execution and kernel level execution for better system protection.

COMPS267F Operating Systems

CPU Scheduling 4

This chapter discusses how to manage the CPU, which is a precious resource on computer systems.

The operating systems are responsible of selecting one from the ready processes and then pass the CPU control to the selected process. The component responsible is called the CPU scheduler.

4.1 Overview of CPU Scheduling

For example, there is one CPU and ten processes. The CPU is now free to use. Which of the ten processes should be selected to use the CPU now?

The main issue about CPU scheduling is the method of selecting a process.

- There are different methods or algorithms.
- Each method will result in different CPU usage patterns.
- Each method will lead to different system performance characteristics.

4.1.1 Example 1: Customer Service in a Bank

We use how a bank selects customers as an example. Yes, nowadays most banks have placed a priority on their customers, whether you like it or not. The highest priority customer can use bank service first.

In a particular branch of Open Bank, there are ten customers queuing for teller service. The Open Bank is suffering from financial tsunami and can afford to make one teller counter available. Consider the following methods of selecting a customer from the queue and evaluate the impact on the business of the bank.

- The first customer arrived will be served first.
- The customer with the higher account balance will be served first.
- The customer wanting one simple transaction (such as deposit cheques, buying foreign currencies) will be served first.

Answer

Different methods will have different impact on the bank's business.

- The first-come-first-served method is most neutral to all customers. The time each customer arrived is apparently random. No other factors are in play here. The bank may be perceived as being neutral and welcome everybody.
- The rich-over-poor method gives priority to the rich (those with highest account balance). The rich will feel more welcome and may be encouraged to do more business with the bank. The bank may expect a higher turnover in the business. This would antagonize the poor.

- The short-over-long method gives priority to the short transactions. Some transactions may be exceedingly time consuming (such as arranging telegram money transfer), and it would cause many other customers to wait. If shorter transactions are allowed to go first, then the majority of customers will enjoy a shorter waiting time. The long transactions are disadvantaged and these customers may never get served if there are new customers keep coming in.

4.1.2 Performance Objectives of CPU Scheduling

CPU scheduler is responsible for selecting a process from the queue of ready processes. The CPU scheduler passes the CPU control to the process. The instructions of the process are then executed by the CPU.

The processes in the ready queue have the following characteristics:

- The processes have been loaded into the main memory.
- The processes do not have the control of the CPU right now.
- The processes are not waiting or doing any IO operation.

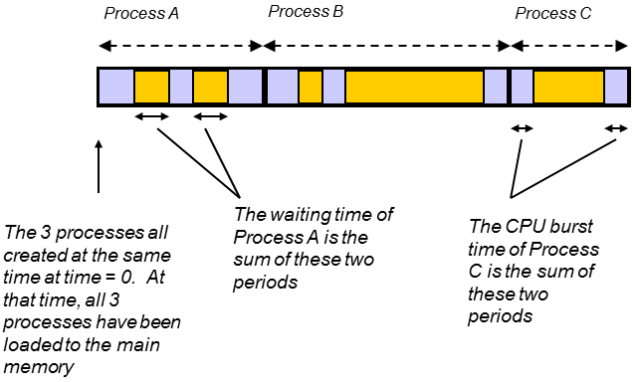
There can be many different methods in selecting the next process to use the CPU. There is no best method in CPU scheduling. Each method produces some specific performance characteristics. Therefore, the logical approach is to determine what performance characteristics are desirable, and then use them to select the method.

The following lists the major performance objectives that CPU scheduling is concerned with:

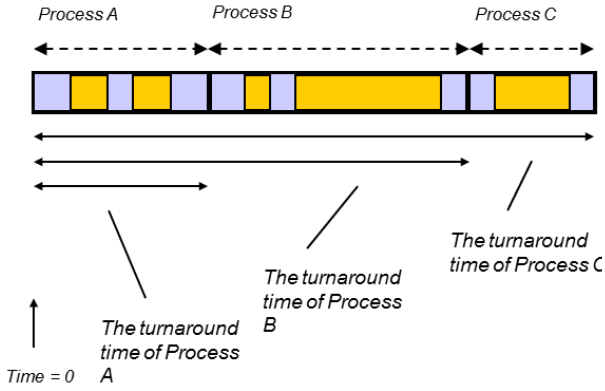
- CPU Utilization. The ratio of time CPU is doing useful work (i.e. executing user programs)
- Memory utilization. The amount of memory being used.
- Multi-user support. All users should receive a certain level of service from the OS.
    - Some users may deserve more of the system resources than the others and they should be given more.
- No Starvation. All processes will eventually have the chance to use the CPU and finish the execution.
- Throughput. The number of processes finished execution per time unit.
    - The amount of useful work carried out by the CPU.
    - Context switching and other overheads are not considered as useful work.
- Turn-around time. The time between the creation and the termination of a process.
    - It is the sum of the time waiting in the ready queue, executing on the CPU, doing I/O, context switching, and other overheads.
- Waiting time. The time not using the CPU.
    - The time in the ready queue and I/O queues.
- Response time. The time from the creation of the process and the first time using the CPU.
    - The time taken to have the process start responding.

The last three metrics are particularly important for performance measurement in CPU scheduling. They can change significantly if different scheduling algorithms are used. The following figures explain them in more details.

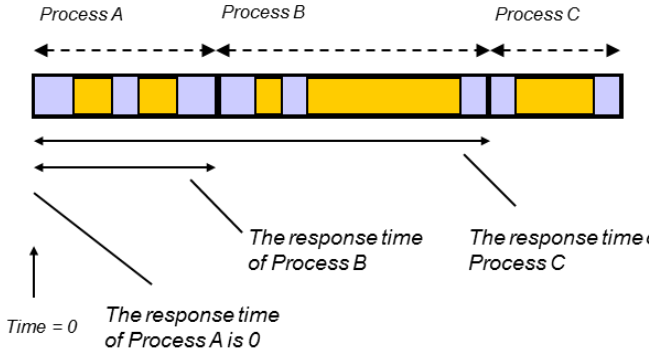
Waiting Time



Turnaround Time



Response Time



The following table gives illustration of these objectives with the bank example.

Objectives	***Bank Examples***
CPU Utilization	The proportion of time that the teller is busy
Turn-around time	The time between a customer joining the queue and the completion of transaction.
Waiting time	The time spending in the queue
Response time	The time between a customer joining the queue and start being served by the teller
No Starvation	All customers will eventually get the transaction finished
Throughput	The number of customers that can be served by the teller per time unit.

4.2 Concepts of CPU Scheduling

There are many different methods of CPU scheduling. It is useful to identify the key concepts first before going into the details of the methods.

4.2.1 Pre-emptive and Non-Preemptive Scheduling

Non pre-emptive scheduling allows a process to control the CPU unless one of the followings occurs:

- The process has finished execution and terminated. The process is switched to the Terminated state.
- The process cannot execute further because it is waiting for an IO operation. The process has to switch to the Waiting state.

Pre-emptive scheduling may remove the CPU control from a process at any time. This action is known as pre-emption. The conditions for removal may be based on time or priority.

- Each process is given a certain time period to use the CPU at a time. When the period is up, then the CPU control is removed from the process.
- Each process is assigned a priority based on the length of CPU bursts. Higher priority is given to processes with short CPU bursts. New processes created with a higher priority will cause the existing Running process to be pre-empted.

Concepts	***Bank Examples***
Non pre-emptive scheduling	Customers are allowed to stay at the teller counter until the transaction is completed or the customers cannot continue the transaction any further because of lack of documents.
Pre-emptive scheduling	Customers are asked to leave the teller if a time quota is up, or if another customer of a higher priority has arrived.

Advantages of pre-emptive scheduling include the following:

- It prevents processes with long CPU burst time to control the CPU exceedingly.
  - No one know how long a process will use the CPU
  - Any process can be pre-empted after a certain period of time.
  - Processes with short CPU burst time do not need to wait for the long process to complete first before having a chance to use the CPU.
- It facilitates scheduling methods to work more effectively by allowing the scheduler to arbitrarily remove the CPU from a Running process.
- Usually it gives better services to shorter processes while worst service to longer processes.

Disadvantages of pre-emptive scheduling include the following:

- Pre-emptive scheduling increases the overhead of process execution and reduces CPU utilization.

  - The increased frequency of switching processes means more context switching.
- Pre-emptive scheduling may cause problems of data inconsistency when a process is updating a set of data and is pre-empted.
  - The situation can be disastrous if system data is involved. So some OS do not allow a process to be pre-empted until a system call is completed.

4.2.2 Example 2: Pre-Emptive Customer Service

In the same branch of Open Bank, the manager decides to impose a rule that each customer can use the teller service for at most 5 minutes. If the transaction cannot complete at the end of the period, then the customer has to leave the teller and join the end of the queue again. Consider the impact of this new rule.

Answer

This is a form of pre-emptive scheduling. Customers wanting to do short transactions will be happier because they would not be delayed by another customer with an exceedingly long transaction.

4.2.3 Scheduling Priority

The concept of priority allows a systematic and unified means to explain process scheduling. The idea of priority based scheduling is simple. The next process to use the CPU is the one with the highest priority.

Each method of CPU scheduling is simply a way to assign priority to processes. The following lists some common ways to assign priority.

- The arrival or creation time. The older the arrival (or creation) time, the higher the priority. So the first created process has the highest priority.
- The length of CPU burst time. The longer the CPU burst time, the lower the priority. So long processes are assigned with lower priorities.
- The time waiting in the queue. The longer the time spending in the queue, the higher the priority.

Sometimes the priority of a process is allowed to change with time. For example, the last way mentioned above will keep changing if the process remains waiting in the queue.

4.2.4 Common Metrics of Short-Term Scheduling Performance

Metrics means measurements. When we talk about performance of a system, we must specific how to measure the performance.

For example, there are many different ways to judge the performance of a smartphone, such as screen size, display clarity and contrast, response time, number of applications, ergonomics, etc.

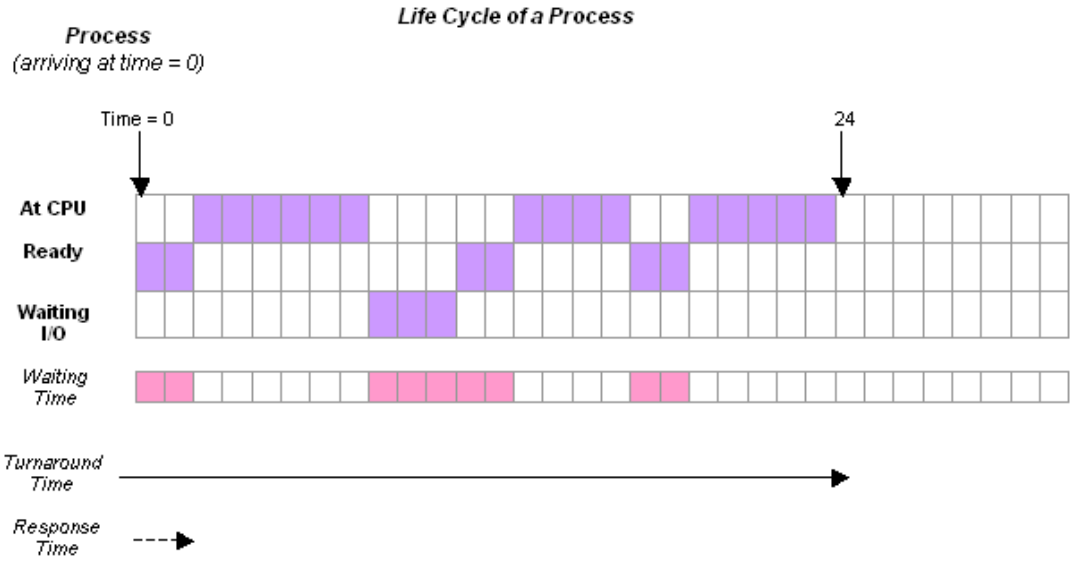
In evaluation of different short term scheduling algorithms, the following three metrics are commonly used.

- Turnaround time. It measures how long it takes for processes to complete the execution.
- Waiting time. It measures how long it takes for processes in waiting instead of doing work.

- Response time. It measures how quick do processes get to use the CPU the first time. This is important for interactive systems. A short response time means that a user will get a quicker response after the execution of a program.

Each algorithm will be assessed with the evaluation of these measurements. A more systematic evaluation will involve applying algorithms to different sets of processes. For example, several sets of processes are used including long processes, short processes, mixed long and short processes.

The following diagram shows how to measure the three factors.



CPU Burst Time (ms)	*15*
Waiting Time	*9*
Turnaround Time	*24*
Response Time	*2*

The above example shows process scheduling with pre-emption. The process is removed of the CPU control at two instances, time 8 and 17.

There are some relations between the three metrics and process pre-emption.

- Turnaround time is the sum of CPU burst time and Waiting time.
- It is easy to work out the Turnaround time by looking at the starting time and termination time. So the waiting time should be worked out from the turnaround time and CPU burst time.
- In non preemptive scheduling, the response time is the same as the waiting time.
    - For these processes, the only waiting time is the period before it first uses the CPU.

4.3 CPU Scheduling Algorithms

In this section the major CPU scheduling algorithms will be described one by one.

4.3.1 First-Come-First Served (FCFS)

First Come First Served selects a process based on the order of arrival or creation. In other words, it assigns higher priority to the older processes.

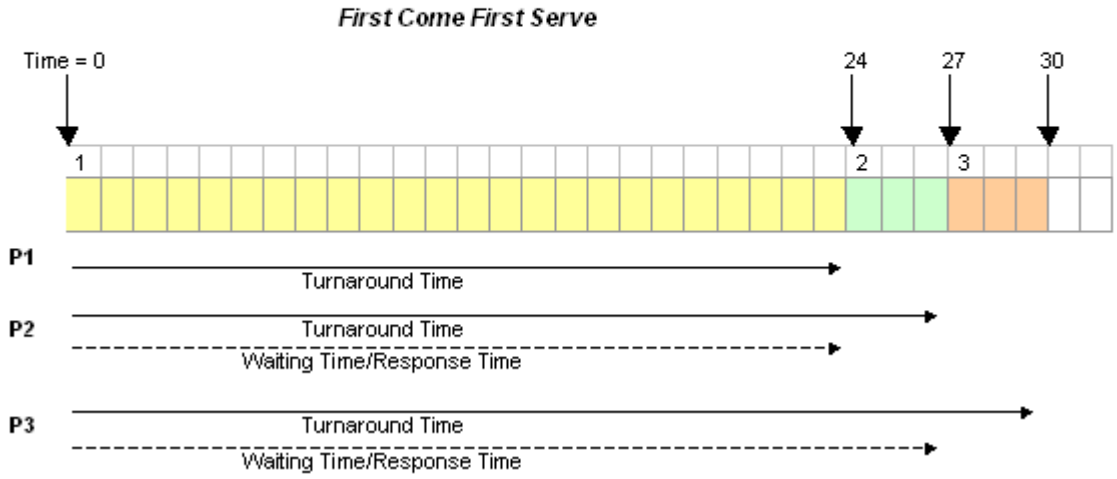
- FCFS is implemented in non preemptive manner. Processes are allowed to use the CPU until the end of the processes.
- The strength is simple implementation and low overhead. The number of context switching is minimized.
- The weakness is short processes may get starved of the CPU. There is no guarantee of response time. In modern computing short processes are overwhelmingly common and so FCFS is not suitable for everyday computing usage pattern.

The following shows an example of three processes managed by a FCFS scheduler. The chart that illustrates which process is controlling the CPU at each time quantum is known as the Gnatt chart.

All 3 processes arrived at time 0, they are considered to have an order P1 before P2 and P2 before P3. The three measurements of the 2 short processes are poor. The long P1 process has occupied the CPU for a long time and causing poor performance in other processes.

Process (arriving at this order)

Process	CPU Burst Time (ms)	Arrival Time
P1	24	0
P2	3	0
P3	3	0



4.3.2 Example 3: Evaluating the characteristics of FCFS

Given the following processes, their CPU burst time and arrival time, calculate the average turnaround time, waiting time, and response time.

Process	CPU Burst Time (ms)	Arrival Time
P1	24	0
P2	3	0
P3	3	0

Answer

The Gnatt chart is shown above. Turnaround time (TT) is clear from the chart and then we can work out the waiting time (WT) from the turnaround time. FCFS is non pre-emptive and so the waiting time is the same as the Response time (RT).

Process	TT	WT	RT
P1	24	0	0

P2	*27*	*24*	*24*
P3	*30*	*27*	*27*
Average	*27*	*17*	*17*

4.3.3 Shortest Job First (SJF)

Shortest Job First selects a process based on the CPU burst time. In other words, it assigns higher priority to processes with shorter CPU burst time. It favours short processes.

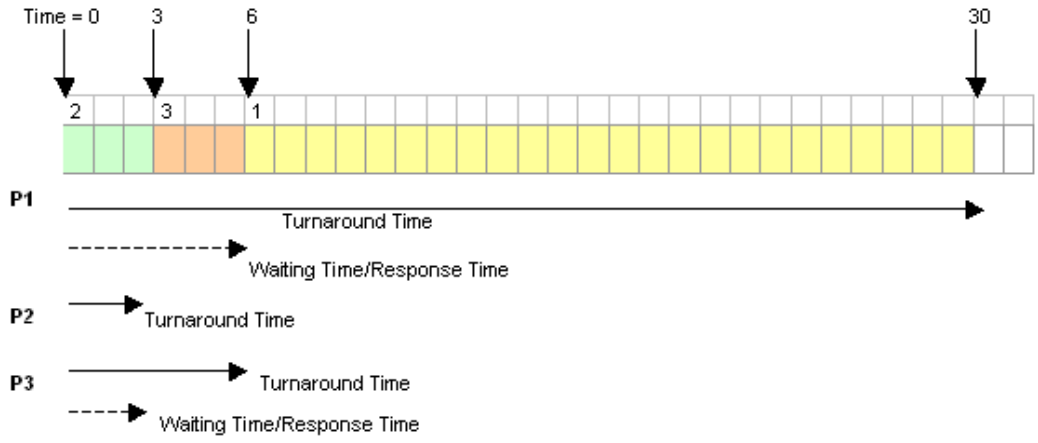
- SJF is the name for the non preemptive version. There is a pre-emptive version known as Shortest Remaining Time First (SRTF).
- The strength of SJF is minimizing the average waiting time.
- The weakness of SJF is that the actual turnaround time and response time for an individual process is difficult to predict. Long processes may be postponed forever.

The following shows an example of how SJF favours short processes. P2 and P3 are short processes and they are selected ahead of P1.

Process *(arriving at this order)*

Process	*CPU Burst Time (ms)*	*Arrival Time*
P1	*24*	*0*
P2	*3*	*0*
P3	*3*	*0*

Shortest Job First (SJF) (Non-Pre-emptive)



4.3.4 Example 4: Evaluating the characteristics of SJF

Given the following processes, their CPU burst time and arrival time, calculate the average turnaround time, waiting time, and response time.

Process	*CPU Burst Time (ms)*	*Arrival Time*
P1	*24*	*0*
P2	*3*	*0*
P3	*3*	*0*

Answer

The Gnatt chart is shown above. Turnaround time (TT) is clear from the chart and then we can work out the waiting time (WT) from the turnaround time. SJF is non pre-emptive and so the waiting time is the same as the Response time (RT).

Process	*TT*	*WT*	*RT*
P1	*30*	*6*	*6*
P2	*3*	*0*	*0*
P3	*6*	*3*	*3*
Average	*13*	*3*	*3*

Overall, the SJF performs significantly better than FCFS in all three measurements. The fact that two out of three processes are short ones helps to produce this performance for SJF.

4.3.5 Shortest Remaining Time First (SRTF)

Shortest Remaining Time First (SRTF) is the pre-emptive version of SJF. The current running process is pre-empted whenever there is a new process arrived.

- The priority is re-calculated at this instance and it includes the new process in the selection.
- The new process may be the one with the shortest CPU burst time and selected to control the CPU. It continues until the arrival of another process.
- The advantage is that new processes will not be waiting forever. Consider in SJF that there was only one long process remaining in the ready queue, SJF would select this long process. After a short while, a new process with short CPU burst time arrived, this new process will have to wait for the long process to finish.
- The disadvantage is greater overhead because of the more frequent context switching and priority evaluation. Long processes may be postponed forever.

4.3.6 Example 5: Evaluating the characteristics of SRTF

Given the following processes, their CPU burst time and arrival time, calculate the average turnaround time, waiting time, and response time.

Process	*CPU Burst Time (ms)*	*Arrival Time*
P1	*8*	*0*
P2	*4*	*1*
P3	*9*	*2*
P4	*5*	*3*

Answer

The Gnatt chart is shown above. SRTF is pre-emptive and so the response time is not always the same as the waiting time. Another important point is that in this scenario the arrival time of the processes varied.

First we work out the TT. Be careful about the arrival time is not always zero.

Then work out the WT = TT – CPU Burst

Finally the RT is worked out by checking the chart, finding when is the first time quantum of a process and then minus the arrival time.

Process	TT	WT	RT
P1	17	9	0
P2	4	0	0
P3	24	15	15
P4	7	2	2
Average	13	6.5	4.25

4.3.7 Highest Response Ratio Next (HRN)

The starvation of long processes is a problem with SJF and SRTF. This problem can be solved with a more sophisticated way to calculate priority.

The Highest Response Ratio Next uses the following formula to calculate the priority of a process.

priority = (waiting time + cpu-burst-time) / cpu-burst-time

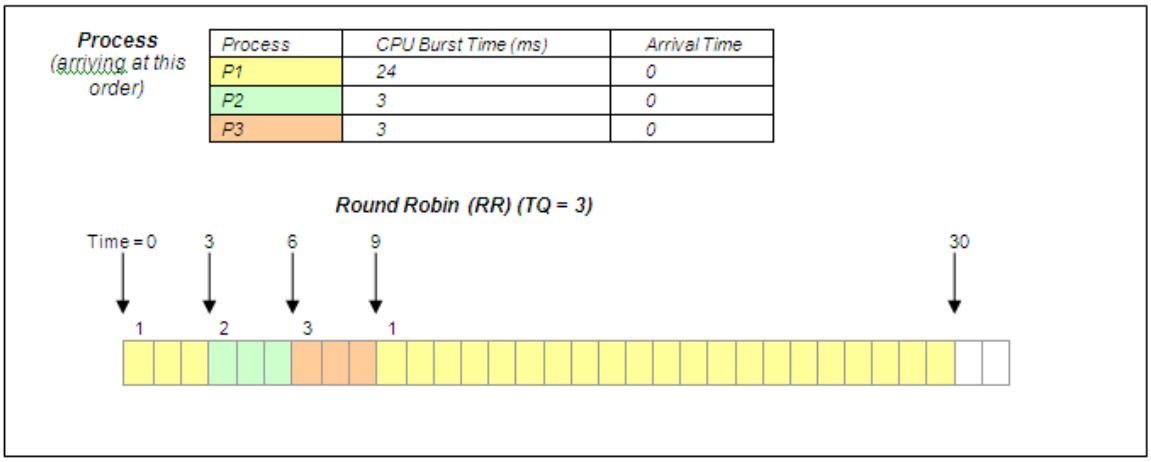
The existence of the waiting time in the formula means that the priority is affected by the time spent waiting. The longer is the waiting time, the higher is the priority.

- HRN is based on SJF and SRT but long processes are gaining more favour by increasing priority gradually. This is known as the aging effect.
- The strength of HRN is less biased towards long processes.
- A possible weakness is that it stills favour short jobs.

4.3.8 Round Robin (RR)

Round Robin ensures that each process has a chance to control the CPU. It achieves this objective with assigning a time quota to each Running process. The time quota is known as the time quantum.

- A process is pre-empted when the assigned time quantum is expired. Then the RR scheduler will select the next process from the ready queue.
- The pre-empted process is moved to the end of the ready queue, unless it happens to complete the execution.
- The strength of RR is less biased towards long or short processes. Each process gets a fair share of the CPU.
- The weakness is high overhead due to the very frequent context switching due to pre-emption.
- The choice of time quantum size is important. A small time quantum causes greater overhead.
- Typical time quantum ranges from 10 to 100 milliseconds.
- A very short time quantum in the range of 1 microsecond gives the impression of processor sharing.
- The implementation of RR scheduler is based on a circular queue as the ready queue. New processes and pre-empted processes are added to the end of the queue.
- The process that is given the CPU is allowed to do so until when it voluntarily gives up the CPU (due to I/O request or job completed) or the time quantum expires.
- The system timer that raises an interrupt causes the process to be put back to the ready queue.



4.3.9 Example 6: Evaluating the characteristics of RR

Given the following processes, their CPU burst time and arrival time, calculate the average turnaround time, waiting time, and response time.

Consider that the time quantum is 3 ms.

Process	CPU Burst Time (ms)	Arrival Time
P1	24	0
P2	3	0
P3	3	0

Answer

The Gnatt chart is shown above. RR is pre-emptive and the response time has to be worked out separately.

Process	TT	WT	RT
P1	30	6	0
P2	6	3	3
P3	9	6	6
Average	15	5	3

Comparing to other algorithms, the measurements are not as good as SJF but close enough. The key characteristic is a small RT for all processes and no process would perform especially poorly.

4.3.10 Example 7: Evaluating the characteristics of RR

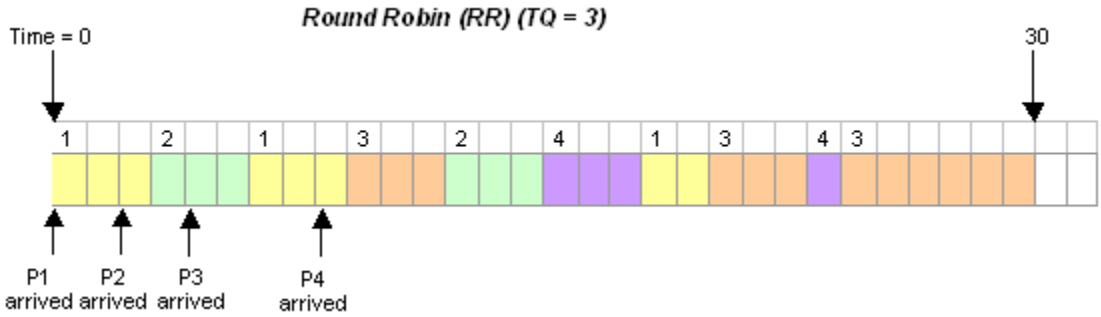
Given the following processes, their CPU burst time and arrival time, calculate the average turnaround time, waiting time, and response time.

Consider that the time quantum is 3 ms.

Process	CPU Burst Time (ms)	Arrival Time
P1	8	0
P2	6	2
P3	12	4
P4	4	8

Answer

Draw the Gnatt chart first. Except for the simplest scenarios, evaluating RR scheduling would involve keeping track of the state of the ready queue.



The Gnatt chart is shown above. The RT is the time when a process controls the CPU the first time, minus the arrival time.

Process	*TT*	*WT*	*RT*
P1	*20 – 0 = 20*	*20 – 8 = 12*	*0*
P2	*15 – 2 = 13*	*13 – 6 = 7*	*3 – 2 = 1*
P3	*30 – 4 = 26*	*26 – 12 = 14*	*9 – 4 = 5*
P4	*24 - 8 = 16*	*16 – 4 = 12*	*15 – 8 = 7*
Average	*18.75*	*11.25*	*3.25*

4.3.11 Example 8: Evaluating the characteristics of RR

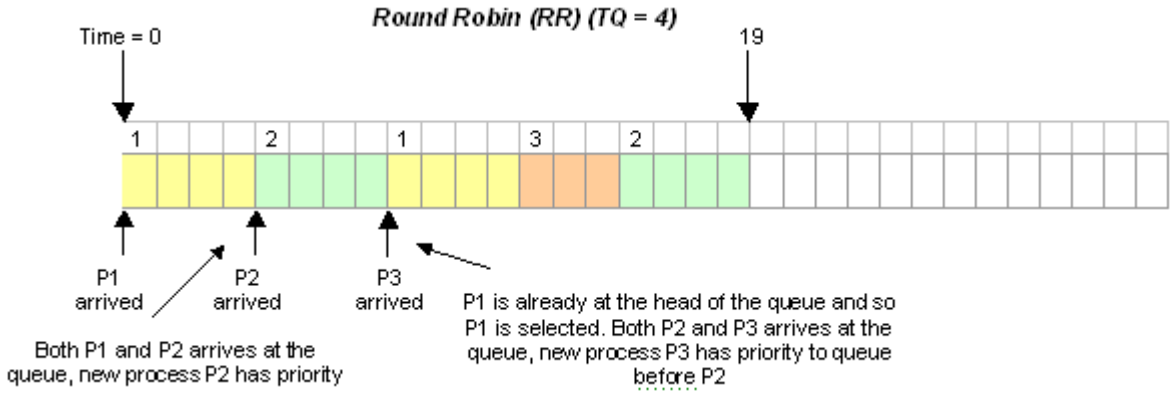
Given the following processes, their CPU burst time and arrival time, calculate the average turnaround time, waiting time, and response time. Consider that the time quantum is 4 ms.

Process	*CPU Burst Time (ms)*	*Arrival Time*
P1	*8*	*0*
P2	*8*	*4*
P3	*3*	*8*

(new processes take priority than old processes)

Answer

The Gnatt chart is shown below. This scenario illustrates the case when a pre-empted process and a newly arrived process coming to join the queue at the same time. For example at time = 4, P1 is pre-empted and re-join the queue and P2 arrived. New processes take priority over old processes, so P2 would join the queue before P1. P2 gets the CPU at time = 4.



The Gnatt chart is shown above. There are two instances when there is a clash of two processes joining the ready queue. At time = 4 and time = 8. In both cases, the new process joins the queue before the old process.

Process	*TT*	*WT*	*RT*
P1	*12 - 0 = 12*	*12 – 8 = 4*	*0*
P2	*19 – 4 = 15*	*15 – 8 = 7*	*4 – 4 = 0*
P3	*15 – 8 = 7*	*7 – 3 = 4*	*12 – 8 = 4*
Average	*11.33*	*5*	*1.33*

4.3.12 Multi -Level Queue Scheduling (MLQ)

Processes can be characterised into several groups of distinctive behaviours. Long processes have poor effect on the performance of short processes if the scheduling approach is non pre-emptive. Systems with a lot of short processes prefer more frequent switching of processes to improve the response time.

A composite algorithm can be designed to exploit knowledge about different groups of processes. It applies different algorithm to different groups of processes.

- Interactive processes are put in a group that is scheduled by RR scheduler with small time quantum. This will minimize the response time.
- Short processes are put in a group that is scheduled by RR scheduler with larger time quantum. It will give a reasonable performance for majority of processes.
- Long processes are put in a group that is not scheduled until there is no other process in the ready queue. It will make sure that the execution of long processes will not block the short processes.

MLQ partition the Ready queue into several queues. Each queue has its own scheduling algorithm suitable for the particular characteristics of its processes. One example of a set of queues includes the following queues. Each queue has absolute priority over the lower-priority queues.

- System processes.
- Interactive processes.
- Interactive editing processes
- Batch processes.
- Student processes.

A possibility is to have specific time proportion for each queue. For example, a high priority queue is given 60 percent, and a lower priority queue gets 10 percent.

The strength of MLQ is its ability to handle a mixed class of processes suitably. The performance is generally acceptable for many classes of processes.

The weakness is the complexity of implementation and the difficulty to know the actual characteristics of each process. It is therefore difficult to make sure each process is placed in the correct queue.

4.3.13 Multi-Level Feedback Queue Scheduling (MLFQ)

Multi-Level Feedback Queue Scheduling is based on MLQ and it allows processes to move from one queue to another queue dynamically.

MLFQ rectifies the problem of MLQ that the characteristics of a process could be difficult to know. It allows a process to execute first, collects data about the characteristic of execution, and then moves the process to the suitable scheduling queue.

- If a process is found to use a lot of CPU time, it is moved to a lower priority queue.
- I/O bound processes and interactive processes are those left in the higher priority queue, thus improving the performance of these processes.

In general, a multi-level feedback queue scheduler is specified by the following parameters:

- The number of queues
- The scheduling algorithm of the queues
- How a process is assigned a queue
- How a lower priority process is upgraded to a higher priority queue
- How a higher priority process is downgraded to a lower priority queue

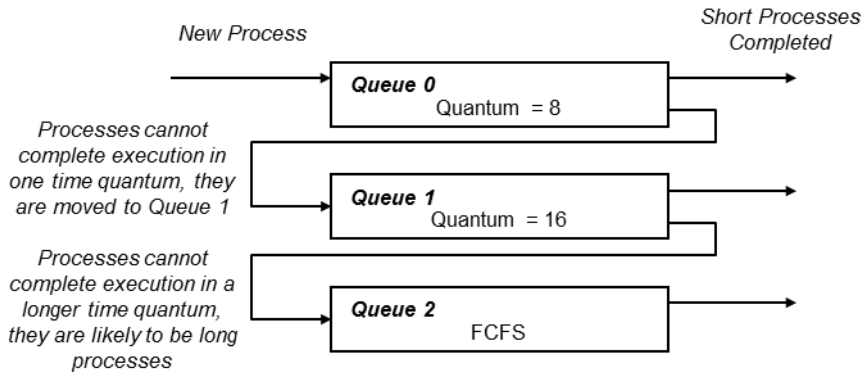
4.3.14 Example 9: Design a MLFQ

MLFQ has not specified a rigid scheduler. Rather it allows flexibility to have a particular of design of MLFQ to achieve a specific objective.

Design a MLFQ based scheduler that favours short process and makes sure long processes are not executed until there is no waiting short process.

Answer

The design of the MLFQ is shown below.



New processes will come in queue 0. Queue 0 has time quantum of 8. Once the process used up the time quantum, it is pre-empted and moved to queue 1. Queue 1 has time quantum of 16. A process used up the time quantum is moved to queue 2. Queue 2 uses FCFS. Processes that ended up in Queue 2 are likely to be long processes.

4.4 Evaluation of Scheduling Algorithms

This is a step-by-step approach of choosing a suitable CPU scheduler for a particular situation.

- The first step is to specify the situation and write down the criteria used in selecting an algorithm.

- The criteria may include CPU utilization, response time and other waiting time.
- If there is more than one desirable criterion, weights should be assigned to indicate which criterion is more important.
- An example is maximizing CPU utilization under the constraint that maximum response time is 1 second.

Deterministic Modelling

Deterministic modelling employs pre-determined workload to test the performance of each scheduling algorithm.

- The advantages include simple, fast, and quantitative.
- The downside is that it is too specific and the chosen pre-determined workload needs to be representative for any generalization.

Queuing Models

The processes in a real computing system can vary significantly. Each process may have its own characteristics of CPU burst and I/O burst.

- The performance of a CPU scheduler can be worked out analytically if there is an abstraction that represents the characteristics of a class of processes. For example, one such abstraction is the proportion of CPU burst and I/O burst.
- There exists a mathematical formula that describes the number of processes with different proportions of CPU/IO, and the distribution of the ratio is exponential.
- Other parameters in a process management scenario, such as arrival rates, service rates can also be modelled mathematically. These parameters can be used to evaluate utilization, average queue length, and waiting time.
- Little's formula, an useful estimation between average queue length, average waiting time, and average arrival rate, assuming that the number of processes leaving the queue equals the number of processes arriving.

$$n = \lambda \times W$$

Simulations

Simulation is the use of a computer program, called a simulator, to model the operation of a computer system. Characteristics of the computer system are included in the program design so that the performance characteristic of the simulator is similar to the performance characteristic of the real computer system.

The data used in driving the simulations is most important. Suitable data must reflect the characteristics of the real-world situations.

4.5 Case Study: Windows OS Scheduling

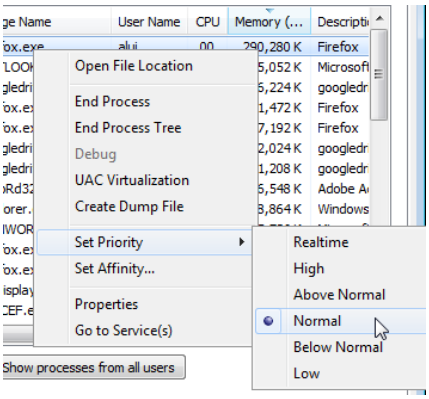
Windows family of OS has supported PC users since the 1990s. It is one of the most popular OS for desktop users.

The process dispatcher of Windows OS uses a priority-based round-robin scheduling algorithm.

- A selected running thread is allowed to run until the following conditions become true.

- The thread has terminated
- The time quantum is used up
- A higher priority has appeared (i.e. pre-empted)
- Making an I/O blocking system call (i.e. waiting)

- The priority scheme has 32 levels.
  - Variable class priority: Level 1 to 15
  - Real-time class priority: Level 16 to 31
  - A system thread for memory management is running at Level 0
- The priority can change
  - The dispatcher will adjust the priority (such as returning from waiting)
  - Users can change the priority of a process in the Task Manager (as shown below)



Windows are designed to boost the performance of interactive programs. It distinguishes foreground processes and background processes. Foreground processes are currently on the screen and visible to users. They are given longer time quantum for making the process more responsive to users.

COMPS267F Operating Systems

Process Synchronization 5

We now have a multi-programming computer system. Many programs can run at the same time, giving good user experience and high CPU utilization.

Programs use data, and sometimes they share data.

This chapter discusses a data integrity problem associated with a multi-programming system. The correctness of a data is threatened if there is more than one process reading/writing the data at the same time. The same is even more common with threads. Threads of the same program are usually designed to work together.

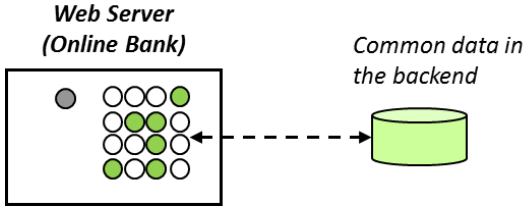
Concurrent access can cause errors in data processing. Process synchronization refers to the management and coordination of processes' activities so that data integrity is maintained.

5.1 Concurrent Access

Concurrent access is necessary in modern computing. The following lists some reasons of the need for processes to cooperate and to share data.

- Information sharing. Multiple processes are working on different tasks in a system. They must share data so that, for example, one process can pass the result to another process for further processing.
- Computation speedup. Multiprogramming can improve the CPU utilization and reduce the turnaround time for IO bound processes. A system designer can exploit this with developing a multi-threading program.
- Modularity. System designers using modular design approach can come up with a set of modules, each executing in a process.

The website for an online bank is one example that concurrent access is essential. As discussed before, website is running on thread pooling (or other multi-threading or multi-processing schemes). However, there should only be one logical database that keeps bank account information. Multiple threads are having concurrent access to the same data.



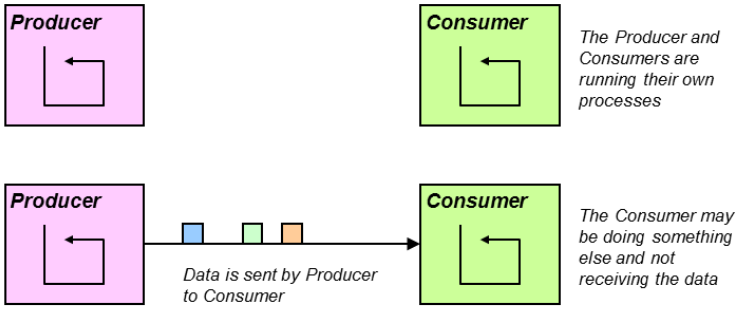
5.2 Data Integrity Problem: Race Conditions

This section will analyse the data integrity problem of multi-programming systems.

A case study will be used to illustrate that errors can occur in concurrent access. The case study involves one process (or thread) writing data to a buffer and another process (or thread) reading data from the buffer.

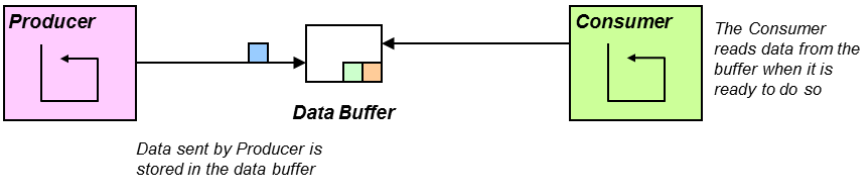
- Producer: the name given to the process that writes data
- Consumer: the name given to the process that reads data
- Buffer: the data structure that contains the data

Note that the buffer is essential for this operation. Data would lose if there is no buffer. The Consumer may be busy doing something else and miss the data coming from the Producer.



The correct design should include a buffer in the middle.

- The buffer collects data sent out by the Producer.
- The buffer waits for the Consumer to get the data. The Consumer can therefore do something else and check the buffer regularly.
- The buffer is bounded. The data may be full.



5.2.1 Bounded Buffer Implementation

Data integrity errors sometimes occur because of very small mistakes. In this case, we will study the implementation of the bounded buffer.

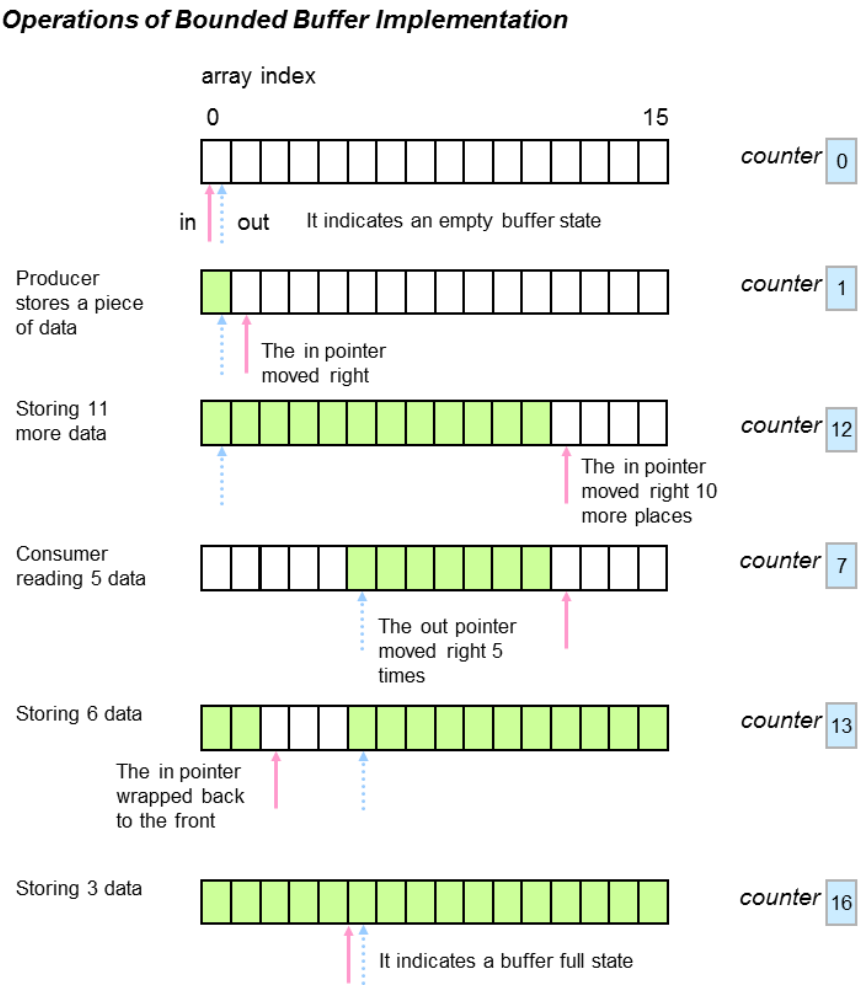
A Java implementation of the bounded buffer is shown.

```
static class Buffer {
    static final int BUFFERSIZE = 16;
    int data[] = new int[BUFFERSIZE];
    int in = 0;
    int out = 0;
    int counter = 0; /* how many data is stored in the array */
}
```

- The buffer is an array with size of 16.
- The array is used in a circular wrap around manner.

- The `in` and `out` are the pointers pointing to the used portion of the array.
- When the producer stores a piece of data, the data is placed in the array element pointed by in.
  - Then the in pointer then moves to the right.
  - If it reaches the right end, it will be wrapped back to the left most element.
- When the consumer retrieves a piece of data, the data to retrieve is placed in the array element pointed by out.
  - Then the out pointer then moves to the right.
  - If it reaches the right end, it will be wrapped back to the left most element.
- The `counter` variable keeps record of how many values are currently stored.
  - Essential for determining whether the buffer is empty or full.

The following figure shows how the bounded buffer implementation works.



A Java implementation of the Producer and Consumer are given below.

```
static class Producer implements Runnable {
...
        try {
          while (buffer.counter == Buffer.BUFFERSIZE) {
            Thread.currentThread().sleep(resttime);
          }
```

```
          buffer.data[buffer.in] = nextProduced;
          buffer.in = (buffer.in + 1) % Buffer.BUFFERSIZE;
          buffer.counter = buffer.counter + 1;
          Thread.currentThread().sleep(rate);
        } catch (InterruptedException ex) {
        }
...
```

```
static class Consumer implements Runnable {
...
        try {
          while (buffer.counter == 0) {
            Thread.currentThread().sleep(resttime);
          }
          lastConsumed = nextConsumed;
          nextConsumed = buffer.data[buffer.out];
          buffer.out = (buffer.out + 1) % Buffer.BUFFERSIZE;
          buffer.counter = buffer.counter - 1;

        } catch (InterruptedException ex) {
        }
...
```

5.2.2 Race Condition

There is a common data that can be accessed by both Producer and Consumer at the same time. The updating of the variable counter may be concurrent.

Producer: `buffer.counter = buffer.counter + 1;`

Consumer: `buffer.counter = buffer.counter - 1;`

Note that the focus is on the possibility of the above two statements happening at the same time.

- Producer and Consumer are running independently of each other. They are on two different threads or processes.
- There is a chance, even a very small chance, that the two statements are simultaneously executed by the Producer and Consumer correspondingly.
- It is not probable, but it is possible that is important.

The effect of running the two statements should *normally* make the counter remain the same.

- If the counter is initially 5.
- Add one by the Producer and subtract one by the Consumer will make it going back to 5.

It looks normal but in fact an error can occasionally happen.

The next step is to go deeper into the machine code for the two statements. They should be compiled into something like the following.

Producer	***Remarks***
1. LDA RA, counter	Loading counter from memory into CPU register RA
2. ADD RA, 1	RA = RA + 1
3. STO counter, RA	Storing the value in RA to the counter in memory

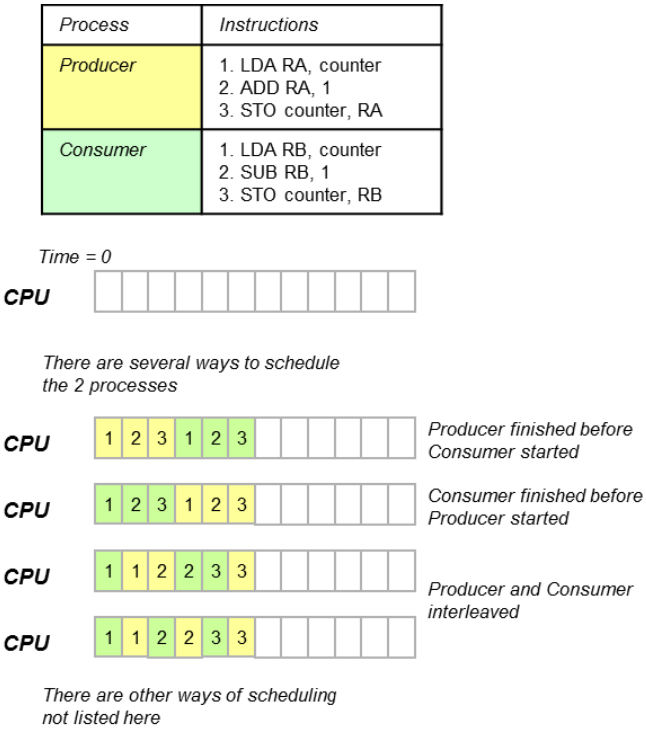
Consumer	***Remarks***
1. LDA RB, counter	Loading counter from memory into CPU register RB
2. SUB RB, 1	RB = RB - 1
3. STO counter, RB	Storing the value in RB to the counter in memory

Consider that there is only one processor, and there is a pre-emptive process scheduler (i.e. round-robin).

- It is up to the process scheduler to decide whether Producer or Consumer controls the CPU.
- It is also up to the process schedule whether to pre-empt one process and pass the CPU control to another process at any time.

The following example shows a scenario that the Producer is executing the code that adds one to the counter and the Consumer is executing the code that subtracts one from the counter.

- There are different ways to schedule the Producer process and the Consumer process.
- The instructions from each process may be executed in different orders.

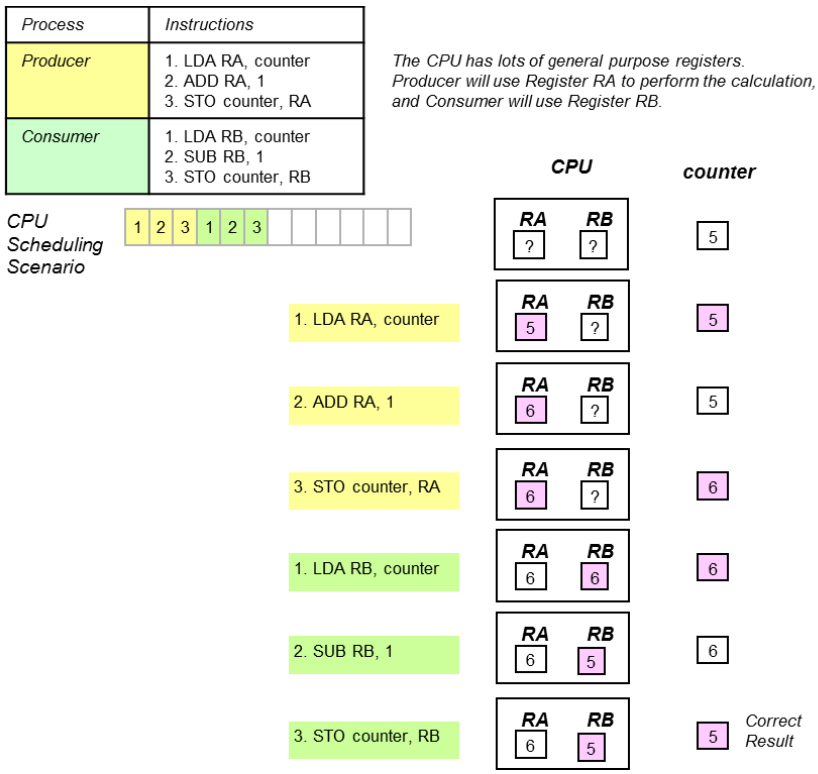


In fact, there are a total of 20 permutations of how the 6 instructions can be ordered with the original instruction order is preserved.

We have come to the error now. Be patient.

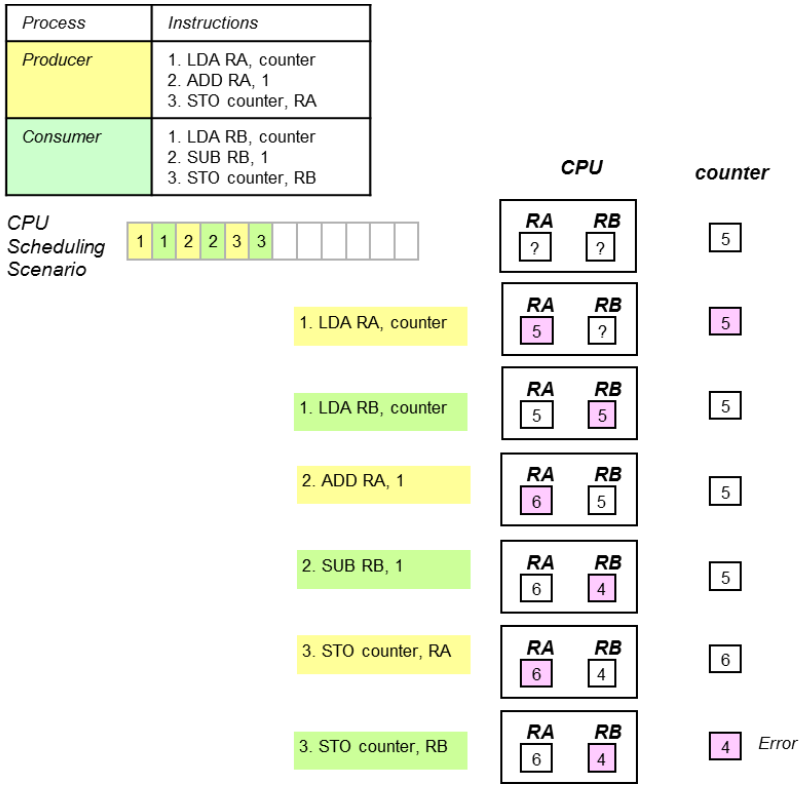
The problem is some patterns of scheduling will result in error.

If the original value of counter is 5, then after the execution the value of counter should remain 5. The following shows the scenario that the 3 instructions of the Producer are executed first and then the 3 instructions of the Consumer.



The following example shows the outcome of another order of execution.

- The Producer is selected to control the CPU first and after the first instruction it is pre-empted.
- The Consumer is then selected to control the CPU. It is pre-empted after the first instruction, and the Producer is selected again.
- Each process is pre-empted after executing one instruction.
- The outcome of this scheduling pattern is an error in the calculation.
- The final value of the counter variable is 4, which is wrong calculation.



Among the 20 permutations, only 2 will give the correct answer of 5. The other 18 permutations will either produce 4 or 6, which are wrong results.

This implementation of Producer and Consumer is said to suffer from race condition. Race condition is the situation that the outcome of processes depends on the execution order of the instructions.

- Multiple processes are reading/writing to a shared data
- The processes' execution order depends on the CPU scheduler.
- The outcome depends on the order of execution.

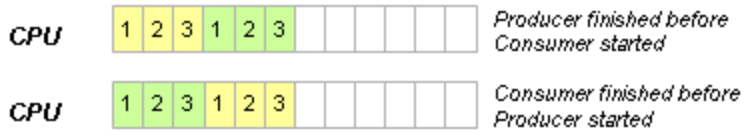
How to solve the race condition problem? The solution is to ensure that only the scheduling patterns leading to correct calculation are allowed.

5.2.3 Example 1: Scheduling Patterns that lead to Correct Calculation

The first example of scheduling pattern allowed correct calculation. Find out the only other scheduling pattern leading to correct calculation.

Answer

The first example is non-interleaved. The Producer executes all 3 instructions together and then the Consumer executes all 3 instructions. The other pattern is also non-interleaved. The Consumer executes all 3 instructions together and then the Producer executes all 3 instructions.



This leads to the observation that interleaving the instructions in a calculation procedure can cause errors. Allowing interleaved scheduling means that two processes are interfering each other with the reading/writing of the variable counter.

5.2.4 Critical Section and Solution for Race Condition

The race condition problem can be solved by disallowing the mutual interference of the reading/writing of common data or resources between processes.

Producer: `buffer.counter = buffer.counter + 1;`

Consumer: `buffer.counter = buffer.counter - 1;`

If the Producer is allowed to complete the execution of the whole operation, and similarly for the Consumer, the outcome is error free.

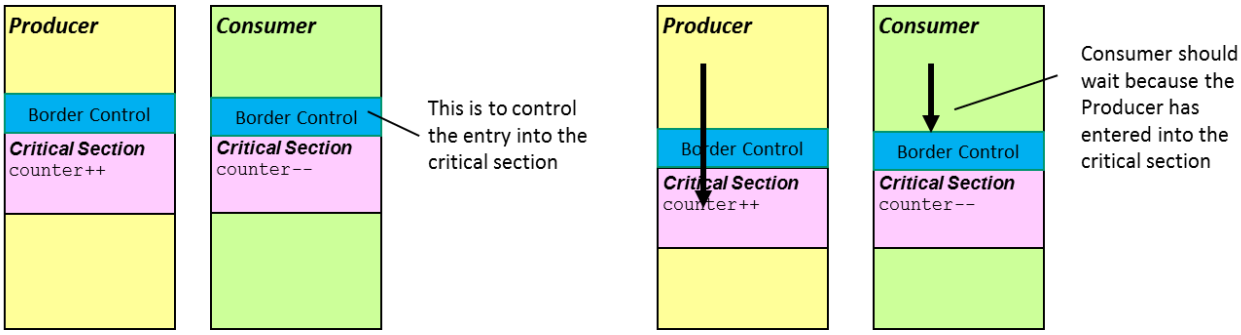
- No pre-emption is allowed while the Producer is executing its counter update operation (or the Consumer is executing its counter update operation).
- The update operations are considered as important. No messing around is allowed.
- The update operations are called critical section.

Critical section is a section of code that a process is updating a shared data or manipulating a common resource.

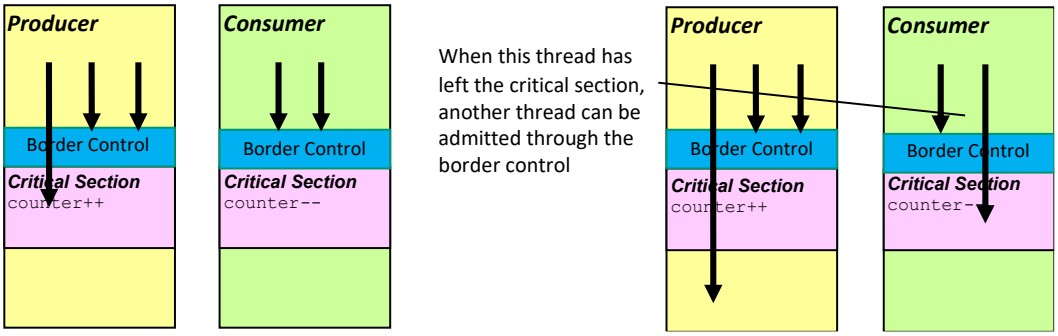
- Critical section is a place where race condition can occur.
- The identification of critical section is the first and essential step to find a solution to prevent race condition.

A proper solution should allow processes (or threads) to synchronize their cooperation so that the entry into critical section is under severe control.

- The control is similar to an immigration border control for processes or threads.
- The border control is placed just before critical section.
- Only one process should be allowed into the critical section.
- Any process entering into critical section may be disallowed and stopped.
- These processes are made to wait until the critical section is free of another process.



There can be multiple threads running Producer and Consumer, still only one of them can go into the critical section. Another thread may then go through if the first thread has left the critical section.

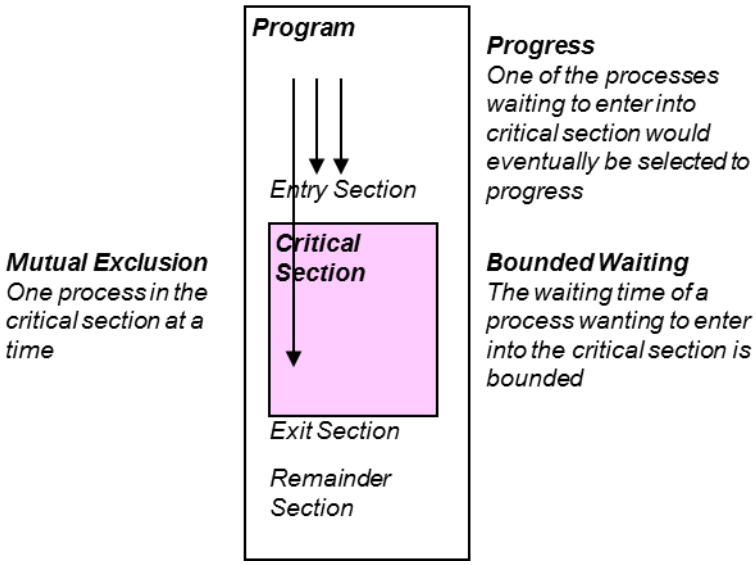


5.3 Implementations of Critical Section Solutions

A solution to the critical section problem must satisfy the following requirements:

- Mutual exclusion.
  - Only one process can enter into the critical section at a time
- Progress.
  - If there is no process in the critical section, then the selection of the next process to enter must take place.
- Bounded waiting.

- There should be an upper limit on the number of times that other processes are selected ahead of a certain process. In other words, a process cannot be omitted indefinitely.



The above figure shows the structure of a critical section, which include the following:

- Entry section. This is where a process is making a request to enter in the critical section. A process waiting for entry would stay in this section.
- Exit section. This is where a process has indicated it has left the critical section. This section is often followed by a remainder section.
- Critical section. The section contains concurrent access program code that causes race condition.
- Remainder section. Normal code considered irrelevant to the problem.

In the following sections, we will look at how to set up the border control as a solution to critical section. We will discuss whether each solution can satisfy the three requirements of critical section solution.

5.3.1 Solution #1 for the Critical Section Problem

Programming constructs should be designed to control the entry into the critical section. A loop is placed in the entry section that traps the process execution until the process is allowed to progress into the critical section.

The following shows a two-process solution that uses a variable turn as an exit condition of the loop.

- The variable turn holds either 1 or 2. It cannot equal both values at the same time.
- Even if the scheduling pattern causes Process 1 and Process 2 to arrive at the entry section (the bold part) at the same time, only one of the two Process will skip the while loop.
- If turn is initialized to 1, then Process 1 will enter into the critical section. If turn is initialized to 2, then Process 2 will enter into the critical section.
- At the exit section, the Process sets turn to allow another Process to progress.

```
/* PROCESS 1 */

while (true) {
...
  while (turn != 1)
    ; /* busy wait */

  **Critical Section;**

  turn = 2;
  Remainder Section;
}
```

```
/* PROCESS 2 */

while (true) {
...
  while (turn != 2)
    ; /* busy wait */

  **Critical Section;**

  turn = 1;
  Remainder Section;
}
```

This critical section solution fails the requirement of Progress.

- If turn is initialized to 1 and Process 2 arrives at the entry section, then Process 2 will wait in the loop.
- The wait is ended when Process 1 arrived at the scene, entered and left the critical section, and sets the turn to 2.
- The problem is Process 2 will wait forever if Process 1 never comes.

5.3.2 Solution #2 for the Critical Section Problem

The second solution uses two more variables to coordinate the action between the two processes. The two variables are declared in a boolean array flag. The two variables are `flag[1]` and `flag[2]`, and they are initialized to false.

- The two variables indicate if a process is in the entry section and wish to enter into the critical section.
- Process 1 sets `flag[1]` to true to indicates the desire to enter into the critical section.
- Process 2 checks this variable to see if Process 1 has any chance to have entered the CS. So `flag[1]` is true, then Process 2 will wait in the loop.
- Process 1 sets `flag[1]` to false after it has left the CS.

```
/* PROCESS 1 */
while (true) {
  flag[1] = true; /* indicating
the desire to go into cs */

  while (flag[2] == true)
    ; /* busy wait */

 **Critical Section;**

  flag[1] = false;
  Remainder Section;
}
```

```
/* PROCESS 2 */
while (true) {
  flag[2] = true; /* indicating
the desire to go into cs */

  while (flag[1] == true)
    ; /* busy wait */

  **Critical Section;**

  flag[2] = false;
  Remainder Section;
}
```

This critical section solution fails the requirement of Progress. The two processes have a chance of trapped in the while loops if the following scheduling pattern occurs.

- Process 1 executed: `flag[1] = true;`
- Process 2 executed: `flag[2] = true;`
- Process 1 will enter into the while loop because `flag[2]` is true.
- Process 2 will enter into the while loop because `flag[1]` is true.

5.3.3 Paterson's Solution for the Critical Section Problem

The third solution described in the following is a correct solution. This solution is often known as Peterson's solution. It is a combination of the above two solutions.

```
/* PROCESS 1 */
while (true) {

flag[1] = true;
turn = 2;

while (flag[2] == true && turn == 2)
 ; /* busy wait */

Critical Section;

flag[1] = false;

Remainder Section;
}
```

```
/* PROCESS 2 */
while (true) {

flag[2] = true;
turn = 1;

while (flag[1] == true && turn == 1)
 ; /* busy wait */

Critical Section;

flag[2] = false;

Remainder Section;
}
```

This solution fixed the problem of solution #2 with the `turn` variable. The variable prevents both processes trapped in the while loop together. This solution would satisfy all three requirements of a critical section solution.

In this solution, only one of the two processes can enter the while loop because the turn variable is part of the while loop condition.

It is important to stress that this solution assumes the operations on shared variables are atomic. An atomic operation means that it cannot be pre-empted by the scheduler before the operation finishes.

- If the operations on shared variables such as turn are not atomic, then the same problem of race condition would happen because these operations would become a critical section.
- Atomic operations can be achieved with computer hardware support.

Programmers will not like this solution. A lot of code has to be written and therefore not convenient to use. Some easy-to-use programming tools for synchronization would be most desired.

5.4 Semaphores

A semaphore is a tool for synchronization. It makes implementation of critical section solutions more convenient. It also allows other pattern of process synchronization.

A semaphore consists of the following three components:

- A semaphore variable. It is an integer that represents the number of available resource.
- The `wait` function. It is an atomic operation that accepts a semaphore variable as a parameter.
- The `signal` function. It is an atomic operation that accepts a semaphore variable as a parameter.

A critical section solution based on a semaphore is shown below.

```
while (true) {
  S = 1; /* initialization */

  wait(S);
  Critical Section;
  signal(S);

  Remainder Section;
}
```

The implementation details of wait(S) and signal(S) are shown below.

```
wait (S) {
  while (S <= 0)
        ; /* busy wait */
  S--;
}

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

Some discussions of the semaphore-based solution.

- The initial value of the semaphore variable is the maximum number of process that can enter into the critical section together. It is set to 1 to allow at most one process to enter.
- The semaphore value could be understood as the number of resources. So the value 1 means that there is only one instance of resource, and the entry right into the critical section could be seen a resource type.
- The wait function plays the role of a gate. It stops processes if the semaphore variable is not positive.
- The signal function plays the role of a notifier. It notifies if there is any process waiting for entry into the CS.

The above implementations of solutions of the critical section problem suffer from one major problem busy wait.

The following implementation uses a waiting process that does not occupy the CPU.

```
wait (S) {
  S--;
  if (S < 0) {
add this process to a queue
block();
  }
}

signal (S) {
 S++;
 if (S <= 0) {
   P = a process removed from the queue wakeup(P);
 }
}
```

5.4.1 Example 2: Using Semaphores for General Synchronization Tasks

Consider how to make process P1 executing a statement S1 before another process P2 executing another statement S2.

Answer

There are two statements S1 and S2. S1 belongs to P1 and S2 belongs to P2. To make P1 executing S1 before P2 executing S2, a way is needed to stop P2 from executing S2 first. We need a gate before S2 to stop P2. A semaphore is used for the gate.

- Wait(R) is called before S2. R is a semaphore variable. For the gate to stop P2, the semaphore variable R must be initialized to 0.
- Signal(R) is called after S1. P1 has already executed S1 and so we should open the gate that stops P2. Signal(R) notifies the gate to open.

```
Sem R = 0;
```

P1	***P2***
`S1;` `Signal (R);`	`Wait (R);` `S2;`

5.4.2 Semaphores and Deadlocks

Improper use of semaphores can cause deadlocks. In this context, **deadlock** means that a set of processes are unable to make any progress because they are made to wait for a signal that never comes.

The following shows an implementation of a critical section solution that will result in deadlock.

The semaphore variables S and Q are initialized to 1.

P1	***P2***
`wait(S);`	`wait(Q);`
`wait(Q);`	**`wait(S);`**
`...`	`...`
`signal(S);` `signal(Q);`	`signal(Q);` `signal(S);`

Process 1 will get trapped at Wait(Q) and P2 will get trapped at Wait (S). At that point both Q and S are zero.

How to fix the problem?

- The order of waiting semaphores is important.
- If the processes are calling wait on semaphores always in the same order, then no deadlock will occur.
- The deadlock problem is caused by P1 holding the resource S and waiting for Q, and P2 holding Q and waiting for S.
- If the resources S and Q are always acquired in the same order (i.e. S and then Q), then this hold-and-wait situation will not occur.

P1	***P2***
`wait(S);`	`wait(S);`
`wait(Q);`	`wait(Q);`
`...` `signal(Q);`	`...` `signal(Q);`
`signal(S);`	`signal(S);`

5.4.3 Binary and Counting Semaphores

A binary semaphore can support range of 0 and 1 only. Binary semaphores are also called mutex locks. Binary semaphores are the form of semaphores that are hardware supported. For example, the x86 CPU series support such locks in the instruction set.

Counting semaphores basically have no restriction on the range.

- Counting semaphores can be implemented with binary semaphores.
- Re-implementation of the wait and signal functions using two binary semaphores.
- For example, programmers can use the following implementation to simulate a counting semaphore.

```
Binary semaphores S1 and S2
Counting semaphore S
An integer variable C
```

```
wait (S) {
  wait(S1);
  C--;

  if (C < 0) {
    signal(S1);
    wait(S2);
  }
  else
    signal(S1);
}
```

```
signal (S) {
  wait(S1);
  C++;

  if (C <= 0)
    signal(S2);

  signal(S1);
}
```

5.4.4 An Alternative: Monitors

Semaphores are useful tools, but they can still be difficult to use correctly. It is a skillful undertaking for many programmers. The positions of the `wait` and `signal` functions and the initial value for semaphore variables have to be precise.

Monitors are an easier-to-use alternative to semaphores. The concept of monitor is based on monitor locks, which can be applied on any object in a program. The monitor itself is a module that contains one or more functions (or methods of a class). It is an object-oriented approach to synchronization.

The advantage of using monitors is its simplicity in programming. The following shows an example. The Source Code is a `Counter` class that the methods must be mutually exclusion. The `monitor class is` a syntactic sugar that signifies the requirement of mutex for each of its method. It is similar to putting the `synchronized` keyword in Java.

The right-hand side shows the implementation of the monitor class. It can be the result of an interpreter or converter. The `Lock` is a monitor lock similar to a binary semaphore. The `acquire` (similar to `wait`) and `release` (similar to `signal`) calls are always inserted at the beginning and the end of each procedure. However, the lock is not equivalent to a semaphore. There are often some subtle differences in the implementation of the synchronization methods.

Source Code	***Implemented Code***

monitor class Counter {
  int count = 0;
 	
  public method int add(int howmany) {
  count = count + howmany;
  return count;
  }
  public method int sub(int howmany){
  count = count - howmany;
  return count;
  }
}
	class Counter {
	  Lock thisLock = new Lock();
	  int count = 0;
	
	  public method int add(int howmany) {
	  thisLock.acquire();
	  try {
	  count = count + howmany;
	  return count;
	  } finally {
	  thisLock.release();
	  }
	  }
	  public method int sub(int howmany){
	  thisLock.acquire();
	  try {
	  count = count - howmany;
	  return count;
	  } finally {
	  thisLock.release();
	  }
	  }
	}

5.5 Classic Problems of Synchronization

There are a number of problems known as classics because they abstract the essence of a large range of general problems.

Abstraction is a very powerful tool in computing. It is used to simplify problems into their core so that the scale of analysis becomes feasible.

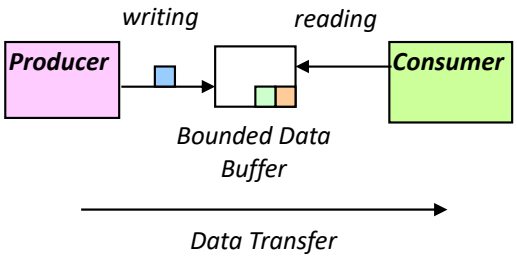
- Real world systems are often complicated and varied in the macro or micro levels.
- A proper analysis of a system is time-consuming and difficult.
- The value of an analysis result depends on whether it can be generalized.
  - Knowledge obtained from the analysis should be applicable to similar systems

The technique of abstraction is to discover and extract the common denominators (or factors) of a class of systems. Knowledge obtained from the analysis of the model of abstraction is applicable to all systems of the same class.

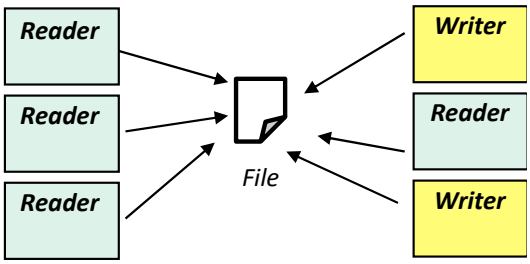
This section shows a semaphore solution to three classic problems.

- Bounded Buffer Problem (or Producer and Consumer Problems)
- Readers/Writers Problem
- Dining Philosophers Problem

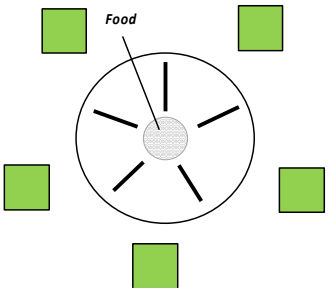
Producer Consumer Problem



Readers Writer Problem



Dining Philosophers Problem



5.5.1 Bounded Buffer Problem

The Bounded Buffer problem is also called the Producer Consumer problem. This problem has the following features:

- There are two processes in the system. One is called the Producer and the other the Consumer.
- The Producer generates data.
- The Producer sends the generated data to the Consumer through a bounded buffer
- The Consumer receives the data from the bounded buffer and process the data.

This model is powerful because it has covered a large set of systems.

- Web browser reading data from web server.
- Word processor reading data from the file system.
- Sending email from an email client to the SMTP server.

There are two conditions that could cause a process unable to proceed – empty buffer and full buffer.

Three semaphore variables are needed for a solution:

- The semaphore `mutex` to implement the critical section solution for updating the shared data buffer.
- The semaphore `full` to stop the Producer from sending data to the buffer if the buffer is full.
- The semaphore `empty` to stop the Consumer from retrieving data from the buffer if the buffer is empty.

Initial semaphore values:

```
Sem full = size of buffer;
Sem empty = 0;
Sem mutex = 1;
```

Producer	Consumer

`while (true) {`
`  wait(full);`
`  wait(mutex);`
 	
`  add a datam to buffer`
 	
`  signal(mutex);`
`  signal(empty);`
`}`
	`while (true) {`
	`  wait(empty);`
	`  wait(mutex);`
	
	`  remove a datum from buffer`
	
	`  signal(mutex);`
	`  signal(full);`
	`}`

5.5.2 Readers-Writers Problems

The readers-writers problem is best known as the file sharing problem.

- A file should allow many readers simultaneously because all readers are reading the same file and there is no data inconsistency.
- A file should allow only one writer. Two writers would cause data inconsistency because the file content becomes an arbitrary mix of two sources.
- When the file has a writer, no reader is allowed. The reader should not read from a partially composed file.

The solution should be based on the following:

- At most only one writer is allowed.
- If there is no writer, then any number of readers is allowed.

The following shows a solution to the readers-writers problem.

Initial semaphore values:

```
Sem mutex = 1;
Sem w = 1;
```

Reader	Writer

`wait(mutex);`
`readercount++;`
`if (readercount == 1)`
`  wait(w);`
`signal(mutex);`
 	
`/* critical section */`
 	
`wait(mutex);`
`readercount--;`
`if (readercount == 0)`
`  signal(w);`
`signal(mutex);`
	`wait(w);`
	
	`/* critical section */`
	
	`signal(w);`

The roles of the two semaphores are listed below:

- The semaphore `mutex` is for ensuring mutual exclusion over the shared variable `readercount`. This variable is shared between readers.

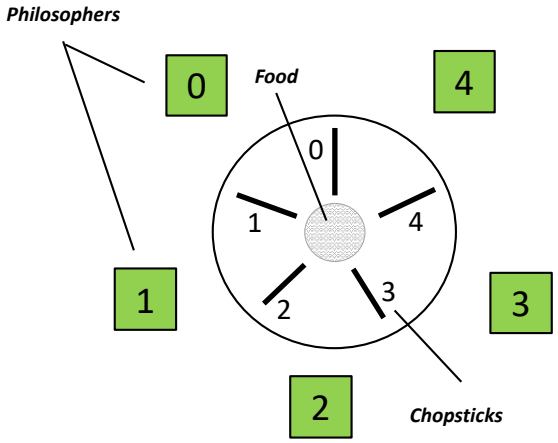
- The semaphore `w` is for stopping the writer if there is already at least one reader, and it is also for stopping a reader if there is already one writer in the critical section.

5.5.3 Dining Philosophers Problem

The dining philosopher problem is a famous because it abstracts a large class of problems that involve sharing a number of resources between multiple processes.

The following figure illustrates the problem with five philosophers at a dining table.

- Each philosopher has a plate.
- There are five chopsticks placed on the table, with one chopstick placed between each neighboring pair of philosophers.
- For any philosopher to eat, the philosopher must take food (spaghetti) at the table centre into the own plate.
  - The philosopher must first obtain the chopstick to the left and the chopstick to the right.
  - Then the philosopher can take the food using the pair of chopsticks.
- When finished, the philosophers replaced the chopsticks back to the original places.



The following shows the first solution. The chopsticks are shared resources. A semaphore is needed for each chopstick to prevent two philosophers fighting for it.

The philosophers are numbered 0 to 4. The chopsticks are also numbered 0 to 4. Chopstick 0 is on the left of philosopher 0 and on the right of philosopher 4.

Initial semaphore values:

```
Sem chopstick[0] ... chopstick[4] = 1;
```

Philosopher

```
/* PHILOSOPHER I */

while (true) {
  wait(chopstick [I]); /* get the left chopstick */
  wait(chopstick [(I+1)%5]);  /* get the right chopstick */
... eat
  signal(chopstick [I]);
  signal(chopstick [(I+1)%5]);
}
```

This solution is prone to deadlock. Consider the case that each philosopher happens to obtain the left chopstick only. All philosophers are holding one chopstick and waiting for another chopstick.

This is possible if the scheduling pattern causes each philosopher to execute only the bold statement.

5.5.4 Example 4: Fix the Deadlock Problem

Fix the deadlock problem of the above dining philosopher solution.

Answer

There are three possible solutions:

1. Allow at most 4 philosophers to share five chopsticks. One philosopher will have a pair of chopsticks to get food and will not have to wait.
2. A philosopher is not allowed to hold one chopstick and wait for another. The philosopher can only hold chopsticks if both are available.
3. Impose an order of picking up the chopsticks. Philosophers with odd ID always pick the left chopstick before the right. Philosophers with even ID always pick the right chopstick before the left.

Philosopher

```
/* PHILOSOPHER I */

while (true) {
if (I % 2 == 1) {
  wait(chopstick [I]); /* get the left chopstick */
  wait(chopstick [(I+1)%5]);  /* get the right chopstick */
} else {
  wait(chopstick [(I+1)%5]);  /* get the right chopstick */
  wait(chopstick [I]); /* get the left chopstick */
}

... eat

if (I % 2 == 1) {
  signal(chopstick [I]);
  signal(chopstick [(I+1)%5]);
} else {
  signal(chopstick [(I+1)%5]);
  signal(chopstick [I]);
}

}
```

5.6 Case Study: Synchronization in Java

Java offers both the traditional semaphore method and the easier monitor method for synchronization. Programmers can use these methods for simple mutex tasks or sophisticated thread coordination.

There are three major methods of synchronization in Java. One of the method is based on the `java.util.concurrent.Semaphore` class, which is basically a powerful implementation of semaphores. The other two methods are described in the following.

5.6.1 The synchronized Keyword

The `synchronized` keyword allows programmer to define a critical section that requires a mutex. There are three scopes that can be defined.

- Methods of an object as a critical section
- The class as a critical section
- Code block as a critical section

Methods of an Object as a Critical Section

The following code shows an example. The synchronized methods of an object of the class becomes a mutex section. The two synchronized methods of an object cannot be entered at the same time by two threads or processes. However, the synchronized methods of two different objects can be entered at the same time.

```
public class Counter {
  private int count = 0;

  public synchronized int add(int howmany) {
    count = count + howmany;
    return count;
  }
  public synchronized int sub(int howmany){
    count = count - howmany;
    return count;
  }
}
```

The Class as a Critical Section

A synchronized static method applies the mutex to all objects of the same class. The synchronized methods of all objects of the same class cannot be entered by two threads or processes at the same time.

```
public class Counter {
  private int count = 0;

  public static synchronized int add(int howmany) {
    count = count + howmany;
    return count;
  }
  public static synchronized int sub(int howmany){
    count = count - howmany;
    return count;
  }
}
```

Code Block as a Critical Section

A code block can be made synchronized with the `synchronized` keyword applied to an object.

```
...
  AnObject lock = new AnObject();  // AnObject is a class defined by user or use Object
...
  synchronized (lock) {
     // critical section under mutex
  }
```

5.6.2 Java Inter-Thread Signalling

Java threads can signal (i.e. communicate) each other with the `wait` and `notify` method call. These are not easy programming constructs to use. They are however similar to the Semaphore wait and signal functions but without the semaphore variable.

Methods	***Remarks***
`wait`	The thread calling wait will become inactive (or sleep) until it receives a notify signal.
`notify`	The thread calling notify will send a signal to one of the waiting threads (on the same synchronized object) and wake it up.
`notifyAll`	The thread calling notify will send a signal to all waiting threads (on the same synchronized object).

There should be multiple threads running an object of the following example class.

```
public class Counter {
  AnObject obj = new AnObject();
  private int count = 0;

  public synchronized void method1() {
    ...
    obj.wait();  // this thread waits here
    ...
  }
  public synchronized void method2(){
    ...
    obj.signal();  // this thread sends signal to a waiting thread
    ...
  }
}
```

COMPS267F Operating Systems

Deadlock 6

Textbook Reading

Silberschatz, Chapter 8, Section 8.1 to 8.8.

This chapter discusses the issue of deadlock in multi-programming systems.

Deadlock occurs when a set of processes are trapped and unable to make any progress. In many cases, a process cannot continue due to the lack of a required resource.

- The process has to wait for the resource.
- There are two possibilities
  - If the resource becomes available in the future, then the process will continue
  - If the resource is never available, the process will wait indefinitely.

A simple case of deadlock is two processes each holding some resources that the other needed.

Relation Between Resources and Processes

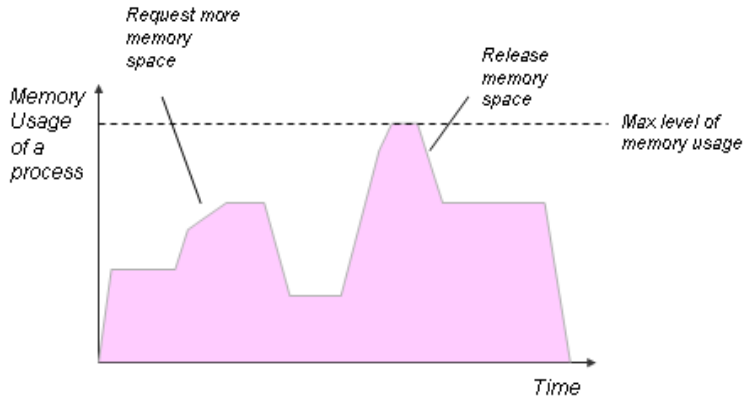
Deadlocks are related to how resources are requested, used, and released by processes in a multi-programming system. There are several resource types commonly associated with computer systems, including CPU time, memory, printers, tape drives, and hard disks.

A process may be associated with a resource in any of the following status.

- Request.
- Use.
- Release.

During its life cycle, a process will request and hold certain resource instances for use and will release the resource instances back to the OS. The number of resource instances held by a process can change quite drastically.

The following figure shows a simulated memory usage pattern of a process.



If a process is requesting instances of resources that are not available, then the process cannot progress any further.

- Hopefully these resource instances will become available later
  - The process could start working again.
- The resource instances will never become available.
  - This situation is deadlock.

6.1.1 Conditions of Deadlock

A deadlock situation can happen when all four following conditions are true at the same time.

- Mutual Exclusion.
  - One or more resources in the system are designated to be non-sharable. Only one process can use the resource at any instance.
- Hold and wait.
  - There exists a process that is holding on some system resource instances and waiting to acquire other resource instances that are currently unavailable.
- No pre-emption.
  - Resource instances cannot be pre-empted by the system.
  - The process can decide on its own whether to release the resource instances; and
- Circular wait.
  - One simple example suffices to illustrate.
  - A process P1 waits for a resource that is held by another process P2, which requires a resource held by P3, which in turn need a resource held by P1.

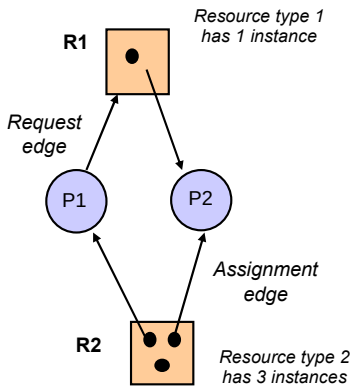
If one of the conditions is found to be false, then deadlock does not occur.

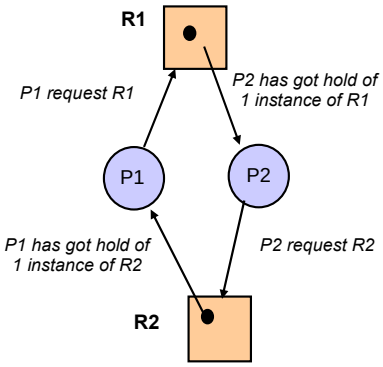
6.1.2 Resource Allocation Graph (RAG)

Resource allocation graph describes the relation between processes and resource instances in a system. The graph allows visual identification of some of the conditions of deadlock, including hold-and-wait and circular wait.

The following shows an example of a RAG.

- Processes are represented by circles and resource types are represented by squares.
- Each instance of a resource type is represented by a dot in the resource square.
- Arrows coming out from a process is a request edge. The arrows should be pointing to a resource type. It means the process is request for an instance of a resource type.
- Arrows coming out from an instance of resource type is an assignment edge. The arrows should be pointing to a process. It means the resource instance is allocated to the process.





Resource allocation graph (RAG) is useful to indicate a potential deadlock situation.

- If the graph has no cycle, then no process in the system is deadlocked.
- If the graph has a cycle, then a deadlock may exist.
- If all resource type has only one instance, then a cycle implies immediately a deadlock in all the processes involved in the cycle.
- If all resource types have more than one instance, then a cycle indicates a possibility that a deadlock has occurred.

The above (right) shows a simple case of deadlock of two processes.

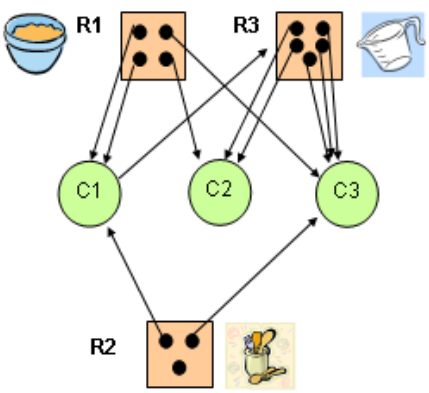
- P1 is hold-and-wait, holding R2 and waiting for R1
- P2 is hold-and-wait, holding R1 and waiting for R2
- P1 is waiting for R1 which is held by P2, and P2 is waiting for R2, which is held by P1

6.1.3 Example 1: RAG for a Cake Shop Kitchen

A cake shop kitchen has 4 plastic bowls (R1), 3 stirrers (R2), and 5 measuring cups (R3). There are three chefs in the kitchen.

- Chef 1 (C1) has got hold of 2 plastic bowls and 1 stirrer.
- Chef 2 (C2) has got hold of 1 plastic bowls and 2 measuring cups.
- Chef 3 (C3) has got hold of 1 plastic bowl and 1 stirrer and 3 measuring cups.
- Chef 1 (C1) is requesting 1 measuring cup.

Draw a resource allocation graph to describe this situation.



Has a deadlock has occurred in the kitchen?

- Only C1 is hold-and-wait
- There is no cycle found in the RAG involving C1
- This is no potential deadlock situation.

6.1.4 Example 2: Cake Shop Kitchen Deadlock

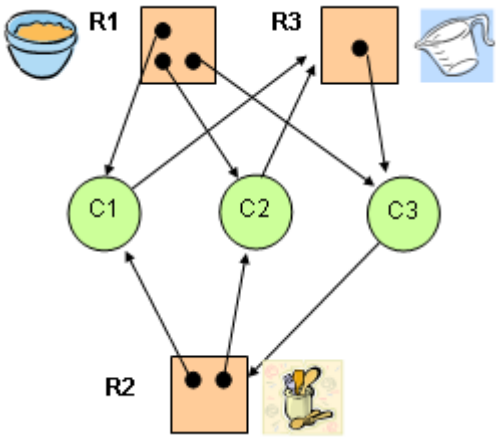
After a change of management, a cake shop kitchen has 3 plastic bowls (R1), 2 stirrers (R2), and 1 measuring cup (R3). There are three chefs in the kitchen.

- Chef 1 (C1) has got hold of 1 plastic bowl and 1 stirrer. C1 is requesting 1 measuring cup.
- Chef 2 (C2) has got hold of 1 plastic bowl and 1 stirred. C2 requesting 1 measuring cup.
- Chef 3 (C3) has got hold of 1 plastic bowl and 1 measuring cup. C3 requesting 1 stirrer.

Draw the RAG of the current resource allocation situation. Evaluate if a deadlock occurs.

Answer

The RAG is shown below.



- C1, C2 and C3 are hold-and-wait.
- A cycle seems to be found in C1/C2 – R3 – C3 – R2.
- There are no more instances available in R3.
- There is no alternative instance of R2, as one is held by C1 and one is held by C2
- Deadlock occurs between C1, C2, and C3

6.1.5 Example 2A: Cake Shop Kitchen Deadlock

Let's consider a very similar scenario as Example 2 above.

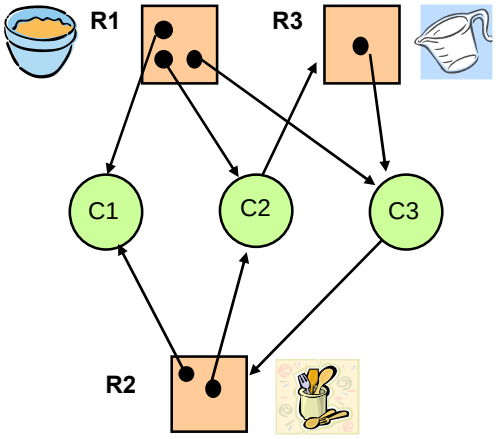
The cake shop kitchen now has 3 plastic bowls (R1), 2 stirrers (R2), and 1 measuring cup (R3). Their allocation situation is

- Chef 1 (C1) has got hold of 1 plastic bowl and 1 stirrer.
- Chef 2 (C2) has got hold of 1 plastic bowl and 1 stirred. C2 requesting 1 measuring cup.
- Chef 3 (C3) has got hold of 1 plastic bowl and 1 measuring cup. C3 requesting 1 stirrer.

Draw the RAG of the current resource allocation situation. Evaluate if a deadlock occurs.

Answer

The RAG is shown below.



- C1 is not hold-and-wait (so not considered anymore)
- C2 and C3 are hold-and-wait.
- A cycle seems to be found in C2 – R3 – C3 – R2.
- There are no more instances available in R3.
- But there is an alternative instance of R2.
- Chefs C3 can be waiting for R2 held by C1 rather than C2.
- The cycle is not truly there.
- A deadlock is possible but no deadlock now.

6.2 Deadlock Handling

Deadlock may be handled in one of the following three ways.

- Use a procedure to prevent or avoid deadlocks.
- Devise methods to detect a deadlock state and recover from it.
- Ignore the deadlock problem.

Deadlock prevention is a simple approach to prevent the possibility of a system entering into deadlock.

- A deadlock situation requires all four conditions of deadlock to be true. So if one of the conditions is prevented to become true, then deadlock will not occur.
- Deadlock prevention simply prevents any one of the four necessary conditions of deadlock to happen.
- The main drawback of deadlock prevention is the measures affect the utilization of system resources.

Deadlock avoidance is a more efficient approach to handle deadlock.

- Deadlock avoidance tries to maximum resource utilization. It exploits the fact the most processes' resource requirement is not always at the maximum. For example, a process needing 2G RAM may be using only 100M RAM most of the time. Only at certain computation intensive period would the actual usage reaches 2G RAM.
- It uses information about resource requirements of processes at different periods of time and tries to see if it is possible to determine a sequence of resource allocation so that deadlocks can be avoided.

A system can choose not to implement either deadlock prevention or deadlock avoidance.

- The system may enter into a deadlock situation.
- The system can get out of the problem with deadlock detection and deadlock recovery.
- Deadlock recovery allows a system to return to normal.

A system may choose to not implement any of the above.

- Deadlock may happen infrequently. Effort spent on deadlock management may not be worthwhile.
- Remember that a system may always be manually recovered from deadlock.

6.2.1 Deadlock Prevention

The method of deadlock prevention is simply to ensure that one of the four deadlock conditions never becomes true.

The following describes the feasibility of using each of the four conditions.

Mutual Exclusion

There is very little we can do here because resource instances are inherently non-shareable.

Hold and Wait

There are two possible solutions to make sure a process does not hold resources while waiting for more.

- A process must get all resource instances it needs before it starts to execute; if it cannot, then it has to release its held resources immediately.
- A process can incrementally request resource instances, but every time it makes those requests it must first release all the currently held resource instances. Then try to see if it can get them back plus the new resources.

These solutions are easy to implement.

- They will result in low utilization levels for resources.
- Some processes that require large amount of resources may starve because they can never get all the necessary resource instances.

No Pre-emption

Pre-emption is the action to force a process to give up all the currently held resources.

- Sometimes pre-emption of a resource type is difficult if not impossible.
- For example, some sequential processing devices such as tape drives are difficult to pre-empt.

Circular Wait

The solution is simply that avoid the circular dependency. A total ordering of the requesting of all resource types is imposed.

- Each resource type available in a system is assigned a unique rank an order. For example, hard disk is 6 and printer is 12.
- A process requesting resources only in increasing order.
- If several instances of a particular resource type are needed, a single request for all of them should be made.

Deadlock prevention solutions are too restrictive. Even when they work, they cause the processes to release held resources frequently and adding to system overhead.

6.2.2 Example 3: Cake Shop Kitchen Deadlock Prevention

Consider how to add deadlock prevention measures to prevent deadlock in the kitchen.

Answer

Each of the four deadlock conditions is discussed.

- Mutual Exclusion.
    - The resource instance like a bowl cannot be shared between two chefs at the same time. Nothing could be done with this condition.
- Hold and Wait.
    - Rules could be imposed on the chefs that if they cannot obtain all the needed resource instances at one go, they should release them.
    - They must not hold onto them.
    - A disadvantage of this solution is low utilization of the resources.
    - Now a chef cannot start working with just some of the required resource instance. Previously, for example, the chef could use a measuring cup and a bowl to start mixing up ingredients while waiting for the stirrer.
- No Pre-emption.
    - No chef would want to release the resource instance held while waiting for other needed ones. A kitchen manage could be installed to look after the allocation of resource. If there is a potential of a deadlock, the manager could order a chef to release all held resources.
- Circular Wait.
    - Rules could be imposed on the order of requesting resource types. Chefs must request in this order: bowl, measuring cups, and then stirrer.

6.2.3 Deadlock Avoidance

The usage level of resource instances of a process can change significantly during its lifetime.

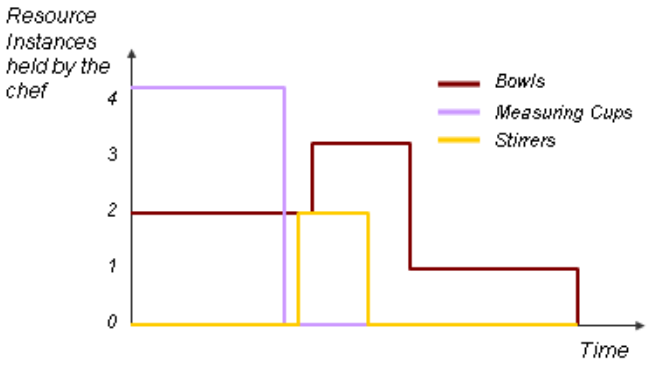
- The maximum level of resource instances specifies the amount of resource instances required to complete the execution of a process.
- The current level of resource instances is the amount of resource instances currently allocated to a process.
- Most processes should be able to perform work with just some of the required resource instances.

For example, a chef requires 3 bowls, 2 stirrers, and 4 measuring cups to complete baking a cake.

- If there are only 2 bowls and 4 measuring cups available, the chef could still start working on mixing ingredients.
- The maximum level of resource instances required is not necessary condition to start working.

- When the chef has put all the ingredients in the bowls, then the 2 stirrers are required, and the measuring cups are not needed.

The following diagram shows the resource utilization of the chef over the work period.



Deadlock avoidance attempts to improve resource utilization by exploiting the time-varying nature of resource requirement. The maximum level of resource instances is not needed for most period of time of a process lifetime. It is possible to have a sequence of resource allocation and de-allocation that can fulfil the resource requirement.

Safe State and Unsafe State

Deadlock avoidance avoids deadlock by defining safe, unsafe, and deadlock states.

- A system is in a safe state if there is at least a way to satisfy all pending requests by scheduling the processes in a sequence.
- A system is in an unsafe state if there is no sequence found that could fulfil the resource requirements of all processes. An unsafe state is not a deadlock state. An unsafe state can potentially become a deadlock state.

The key to deadlock avoidance is to ensure that after each resource allocation the system will remain in a safe state.

6.2.4 Example 4: Case Study: New Management Style for Cake Shop Kitchen

The existing utensils such as bowls and measuring cups have aged. New ones are to be bought. There is a new management installed in the cake shop kitchen in order to reduce the cost of infrastructure. The chefs are asked the maximum instances of bowls, stirrers and measuring cups needed for completing their jobs.

Chef	Max (Bowls, Stirrers, Measuring Cups)
C1	1 2 1
C2	3 1 0
C3	1 1 1

Calculate many bowls, stirrers, and measuring cups should be bought.

Answer

The expensive solution is simply a sum of the requirements of all chefs.

Bowls = 1 + 3 + 1 = 5

Stirrers = 2 + 1 + 1 = 4

Measuring Cups = 1 + 0 + 1 = 2

Cost Saving Measure of Cake Shop Kitchen

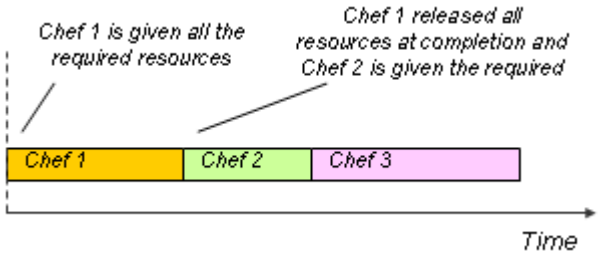
The new management considers buying 5 bowls, 4 stirrers, and 2 measuring cups cost too much. The management consulted you and checked if 3 bowls, 2 stirrers, and 1 cup would be sufficient to keep the kitchen running.

Chef	Max (Bowls, Stirrers, Measuring Cups)
C1	1 2 1
C2	3 1 0
C3	1 1 1

Answer

Yes. It can be done if the chefs use the resources in the sequence of C1 – C2 – C3. Chef 1 will use 1 bowl, 2 stirrers, and 1 measuring cup first. Then Chef 2 started working after Chef 1 has completed. The resources originally allocated to Chef 1 were released for Chef 2 to use. The Chef 3 could start working after Chef 2 completed the work.

The drawback of this approach is the chefs cannot work at the same time and reduced the output of the kitchen.



Cost Saving Yet More Efficient

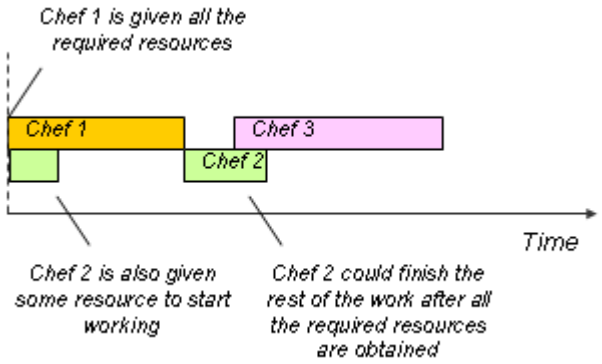
In Example 6, the solution of having the chefs working sequentially is not efficient. Is there a better method?

Chef	Max (Bowls, Stirrers, Measuring Cups)
C1	1 2 1
C2	3 1 0
C3	1 1 1

Answer

Yes. The kitchen has 3 bowls, 2 stirrers, and 1 cup.

At time = 0, C1 is given 1 bowl, 2 stirrer and 1 cup to start working, and C2 is given 2 bowls to start working. C2 is washing strawberries and the bowls can allow C2 to do some of the required work. The work of the three chefs could complete in a shorter time.



6.2.5 Example 5: A Safe State

Evaluate if the current state of resource allocation is in a safe state. There is only one resource type.

Process	Allocation	Max
P1	2	9
P2	2	4
P3	2	7

Available free resource instances = 3

Answer:

If it is in a safe state, there exists a sequence of resource allocation that can satisfy all pending requests. First we calculate how many resource instances are needed of each process.

Process	Allocation	Max	Need
P1	2	9	7
P2	2	4	2
P3	2	7	5

The following is a sequence that allows all processes to obtain the required resources.

- Allocate 2 to P2, P2 has 4, P2 completed execution, P2 releases 4 back to system.
  - Available resource instances = 1 + 4 = 5
- Allocate 5 to P3, P3 has 7, P3 completed execution, P3 releases 7 back to system.
  - Available resource instances = 0 + 7 = 7
- Allocate 7 to P1, P1 has 9, P1 completed execution, P1 releases 9 back to system.
  - Available resource instances = 0 + 9 = 9

The system is safe.

6.2.6 Banker's Algorithm

The name of this algorithm is very illustrative: it shows what bankers need to do when they allocate their resources. The problem faced by bankers is actually no different than the ones we have been dealing with.

Consider the scenario of a bank with $10,000 in deposit. Three customers each would like to apply $4,000 credit. The bank has approved their applications. The bank seems unable to satisfy all three customers and could be in trouble. Is it true?

- When and how much each customer needs the money is important.
- The bank is in trouble only if all three customers draw their approved credit line at the same time.
- The bank is actually safe as long as the total sum drawn by the clients is less than $10,000. Most clients do repay their loans on time and the available cash can increase.

The bankers' experience in handling this situation inspires the development of a deadlock avoidance algorithm known as banker's algorithm.

Generally, the banker's algorithm allocates M types of resources, each with a certain number of instances, to N number of processes.

The banker's algorithm requires the following data structures.

- Available vector. This stores the number of available resource instances of each type.
- Max matrix. This stores the maximum demand of each type of resource by each process.
- Allocation matrix. This stores the currently allocated instances of each type of resource to each process.
- Need matrix. This stores the remaining amount of instances of each type of resources required by each process.

If two of the three matrices, Max, Allocation, and Need, is known, the remaining one can be calculated

Need = Max - Allocation

Analysis with the banker's algorithm starts with a resource allocation table like the following.

- There are four types of resources in the system.
- The Allocation column indicates the number of instances of each type of resources allocated to a process. P1 is currently allocated with 3 Resource #1, 0 Resource #2, 1 Resource #3, and 1 Resource #4.
- The Need column indicates the resource instances required but not allocated.
- The Max column is the maximum level of resource instances required by each process so that the execution can complete.
- The Available vector indicates the free resource instances currently available.

Available vector is (1 0 2 0)

Process	***Allocation***	***Need***	***Max***
P1	3 0 1 1	1 1 0 0	4 1 1 1
P2	0 1 0 0	0 1 1 2	0 2 1 2
P3	1 1 1 0	3 1 0 0	4 2 1 0
P4	1 1 0 1	0 0 1 0	1 1 1 1
P5	0 0 0 0	2 1 1 0	2 1 1 0

The operation of the banker's algorithm is described below:

- Look for a row (i.e. Process) in Need whose unmet resources are all smaller than or equal to Available. If no such row exists, the system is in unsafe state and it will eventually deadlock.
- Allocate the resources to the process chosen above (and modify the Allocation matrix) and assume it can finish. Now mark the process as done and add all its resources to Available.
- Continue the above steps until all processes are marked finished, in which case the original state is safe.

6.2.7 Example 6: Banker's Algorithm

Evaluate if the above resource allocation status is in a safe state with banker's algorithm.

Answer

Actions	**Available**
	1 0 2 0
Allocate 0010 to P4. P4 has 1111. P4 completed and release resources back to the system	1 0 2 0 – 0 0 1 0 + 1 1 1 1 = 2 1 2 1
Allocate 1100 to P1. P1 has 4111. P1 completed and release resource back to the system	2 1 2 1 – 1 1 0 0 + 4 1 1 1 = 5 1 3 2
Allocate 3100 to P3. P3 has 4210. P3 completed and release resource back to the system	5 1 3 2 – 3 1 0 0 + 4 2 1 0 = 6 2 4 2
Allocate 0112 to P2. P2 has 0212. P2 completed and release resource back to the system	6 2 4 2 – 0 1 1 2 + 0 2 1 2 = 6 3 4 2
Allocate 2110 to P5. P5 has 2110. P5 completed and release resource back to the system	6 3 4 2 – 2 1 1 0 + 2 1 1 0 = 6 3 4 2

The system is in a safe state. There exists a sequence of resource allocation that can satisfy the resource requirements of all processes.

6.2.8 Example 7: Banker's Algorithm

Consider the following resource allocation state again, if P3 asks for the allocation of 1000 (1 instance of Resource #1). Should this request be granted?

Answer

Check if the system remains a safe state if the request is granted. If the request is granted, the resource allocation state is changed as below. P3 received one instance of Resource #1 and so the Allocation becomes 2110 and the Need becomes 2100.

Available vector is (0 0 2 0)

Process	***Allocation***	***Need***	***Max***
P1	3 0 1 1	1 1 0 0	4 1 1 1
P2	0 1 0 0	0 1 1 2	0 2 1 2
P3	**2** 1 1 0	**2** 1 0 0	4 2 1 0
P4	1 1 0 1	0 0 1 0	1 1 1 1
P5	0 0 0 0	2 1 1 0	2 1 1 0

The banker's algorithm is applied below.

Actions	**Available**
	0 0 2 0
Allocate 0010 to P4. P4 has 1111. P4 completed and release resources back to the system	0 0 2 0 – 0 0 1 0 + 1 1 1 1 = 1 1 2 1
Allocate 1100 to P1. P1 has 4111. P1 completed and release resource back to the system	1 1 2 1 – 1 1 0 0 + 4 1 1 1 = 4 1 3 2

Allocate 2100 to P3. P3 has 4210. P3 completed and release resource back to the system	4 1 3 2 – 2 1 0 0 + 4 2 1 0 = 6 2 4 2
Allocate 0112 to P2. P2 has 0212. P2 completed and release resource back to the system	6 2 4 2 – 0 1 1 2 + 0 2 1 2 = 6 3 4 2
Allocate 2110 to P5. P5 has 2110. P5 completed and release resource back to the system	6 3 4 2 – 2 1 1 0 + 2 1 1 0 = 6 3 4 2

It is in a safe state. The request from P3 can be granted.

6.2.9 Deadlock Detection

Deadlock detection is needed if a system is allowed to fall into a deadlock state. The reason is mostly due to the lack of deadlock prevention or avoidance service in the system.

If a system can fall into a deadlock state because it has no deadlock prevention or avoidance algorithms, it is necessary to have deadlock detection ability and a remedy of deadlock recovery.

A deadlock detection algorithm is very similar to the banker's algorithm.

- Look for a row (i.e. Process) in Need whose unmet resources are all smaller than or equal to Available. If no such row exists, the system is in deadlock, and all unfinished processes are deadlocked.
- If such a row exists, allocate the resources to the process chosen above (and modify the Allocation matrix) and assume it can finish. Now mark the process as done and add all its resources to Available.
- Continue the above steps until all processes are marked finished, in which case the original state is safe. If there are remaining unfinished processes, the processes are in deadlock.

Deadlock detection algorithm is costly to run and the frequency of running it should be carefully adjusted.

6.2.10 Deadlock Recovery

If neither deadlock avoidance nor prevention can be implemented, then we must resort to recovering from deadlocks. We assume that we are able to detect deadlocks using a resource allocation graph. There are several approaches for deadlock recovery:

- Process Termination.
  - This approach is simple and effective.
  - A deadlock is represented by a cycle in the resource allocation graph. One of the processes in the cycle is terminated. The allocated resource instances are released.
  - The above is repeated until the deadlock is resolved. In the worst case all processes involved in the deadlock cycle have been terminated.
  - The problem with this approach is that terminating a process is sometime difficult and usually expensive. Aborting a printing job after it has printed a few hundred pages is a good example.
  - Choosing the suitable process to be terminated is sometimes a matter of guessing.

- Process Pre-emption.
  - This approach will pre-empt a process.
  - The system will restart the pre-empted process from some safe state later.
  - It is not easy to select a suitable process, stop it, rollback and restart.

COMPS267F Operating Systems

Memory Management 7

The memory system plays an important role in program execution.

- The main memory is directly addressable by the CPU.
    - Program code can be stored there.
- The main memory stores multiple programs to support multi-programming.

This chapter discusses the techniques developed for memory management to support multi-programming.

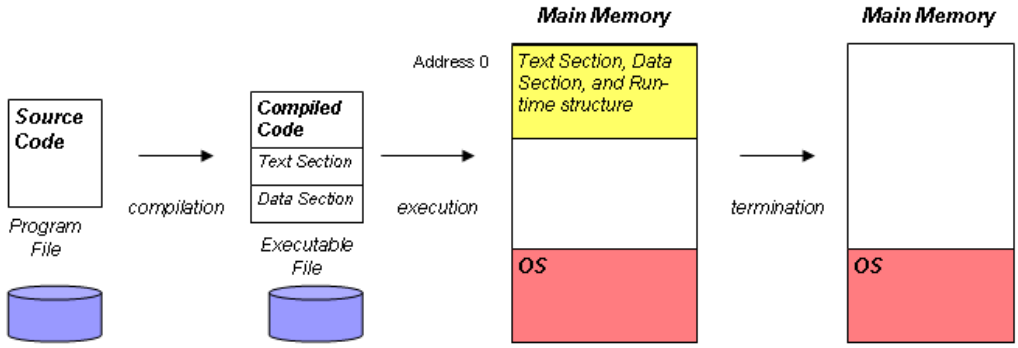
7.1 Program Execution and Main Memory

Program execution requires the loading of the program from secondary memory into the main memory.

- The program is in the form of an executable file.
    - The file contains two parts: the text section and the data section.
    - The text section contains program instructions.
    - The data section contains constant data declared in the program.
- The executable file is an outcome of compilation of a source file. It is stored in secondary memory so that it remains available for repeated used.
- The CPU can directly access the internal registers and the main memory.
    - The operating system needs to load the program into the main memory before program execution can begin.
    - The CPU can then read instructions through the memory management unit (the MAR and the MDR).

The task of the operating system is easy if only one program is in execution at a time.

- Memory usage becomes a simple cycle: loading program into main memory, executing program, waiting for program termination, releasing the memory.
- The program is always loaded to the bottom of the memory space. The first instruction is loaded at address 0.
- Many instructions refer to addresses in the memory. The compiler assumes that the first instruction would be loaded at address 0. This is the reference point of the addresses specified in all the instructions.



7.1.1 Example 1: Loading of LMC Programs

The following is a simple LMC program that reads and prints the sums two numbers.

```
00 901; IN
01 310; STO 10
02 901; IN
03 110; ADD 10
04 902; OUT
05 000; COB
10 000; DAT
```

The program contains both instructions and constant data (address 10). The program assumes that the first instruction is at address 0. The program will be loaded into the main memory at address 0. The program will be executed correctly.

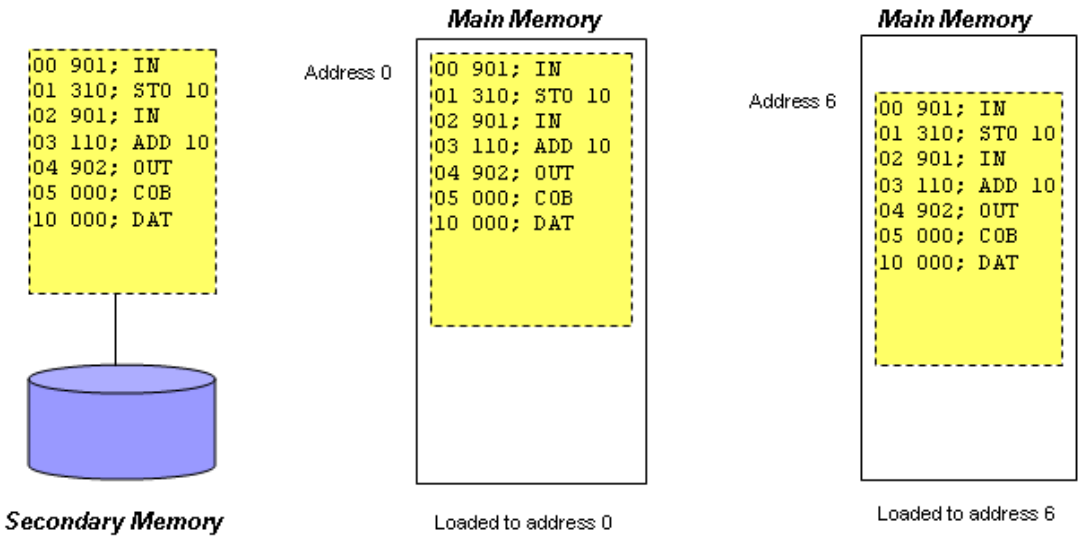
Question: Would the program run correctly if it were loaded to other addresses?

If the program were loaded to address 6, for example, then the program would not run correctly. The same instructions would now locate at different addresses.

```
06 901; IN
07 310; STO 10
08 901; IN
09 110; ADD 10
10 902; OUT
11 000; COB
```

The instruction at address 07 would store the input data to address 10. This would overwrite the instruction OUT now located at address 10.

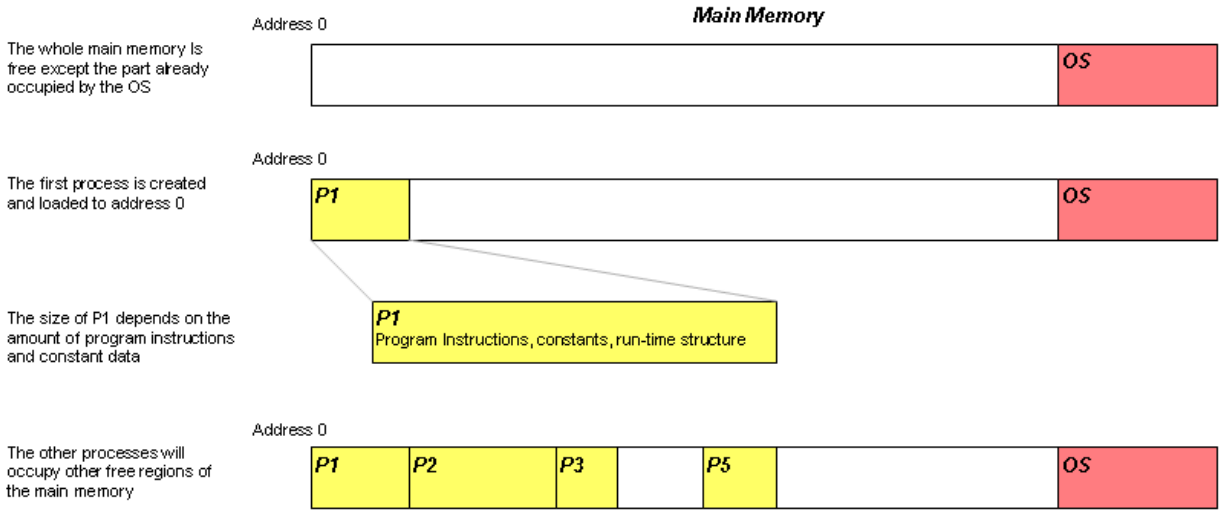
The program could still run correctly if loaded to some addresses, for example, address 20. However, the operating systems have no easy to know whether a particular starting address would render the program erroneous. To be safe from error, the program should always be loaded to address 0.



7.2 Logical and Physical Addressing Space

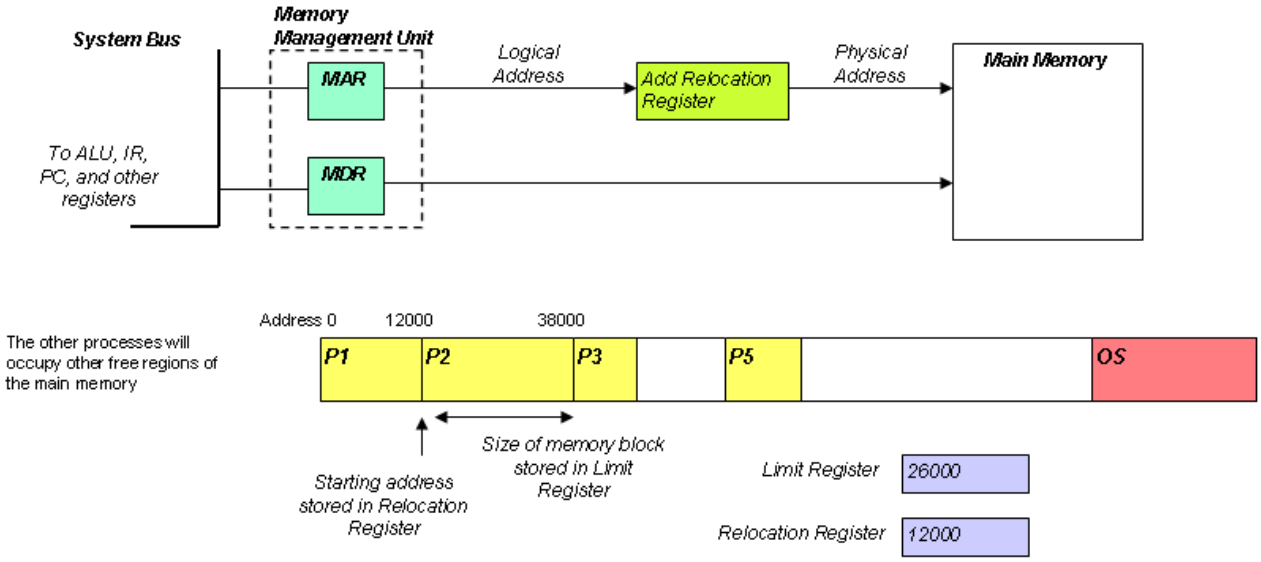
Program execution in multi-programming environments demands services provided by the operating systems.

- A technique for loading and executing programs correctly at any address is required.
  - There is more than one program executing concurrently.
  - Each of the many programs now occupies the main memory at the same time.
  - Only one process can occupy address 0. The other processes must use memory at different locations.
- The operating systems must keep track of which part of the main memory is occupied.
  - When new processes are created, the operating system knows which part of memory is still free.
  - The operating systems then allocate sufficient memory according to the size of the program and the associate runtime data structures.
  - The runtime structure includes a stack for local variables and heap space for other data.
- The operating systems must impose security measures and segregate the processes.
  - Each process is allowed to access addresses within its allocated memory space.
  - Intrusion into the memory space of another process should be strictly prohibited.



Address conversion is required so that a program loaded at address A can execute correctly.

- The memory management unit automatically adds the loading address A to the address specified in all memory operations.
- Before the process is being executed, the loading address A is stored in the relocation register.
- The memory management unit adds the relocation register and the memory address register (MAR) to obtain the address in the main memory with respect to the memory space occupied by the process.



7.2.1 Conversion from Logical Address to Physical Address

The resolution of the physical address from logical address is an important operation in supporting multi-programming.

- The logical address space assumes that the starting address is 0. All addresses specified in program instructions based on this assumption are logical addresses.
- The instructions generated by the compiler are referenced on logical address space.

- The physical address space is based on the perspective of the main memory. The starting address can be anywhere in the main memory.
- When a program is loaded into the main memory, it is in the physical address space. To make the program working correctly, a conversion from logical addresses to physical addresses is required.
- The conversion is performed by hardware built into the memory management unit as described above. This conversion of addresses is known as binding.

7.2.2 Example 2: Logical and Physical addresses

The following is the same LMC program. It is composed in the logical address space. The starting address is assumed to be 0. Instructions at 01 and 03 involve memory operations. The addressing mode is direct addressing. The addresses specified (ie. address 10) are also in logical address space. These are logical addresses.

```
00 901; IN
01 310; STO 10
02 901; IN
03 110; ADD 10
04 902; OUT
05 000; COB
10 000; DAT
```

Supposed that the program is loaded at address 6. A piece of memory space is allocated to the process from address 6 to address 16. The starting address 6 is stored in the relocation register.

```
06 901; IN
07 310; STO 10
08 901; IN
09 110; ADD 10
10 902; OUT
11 000; COB
```

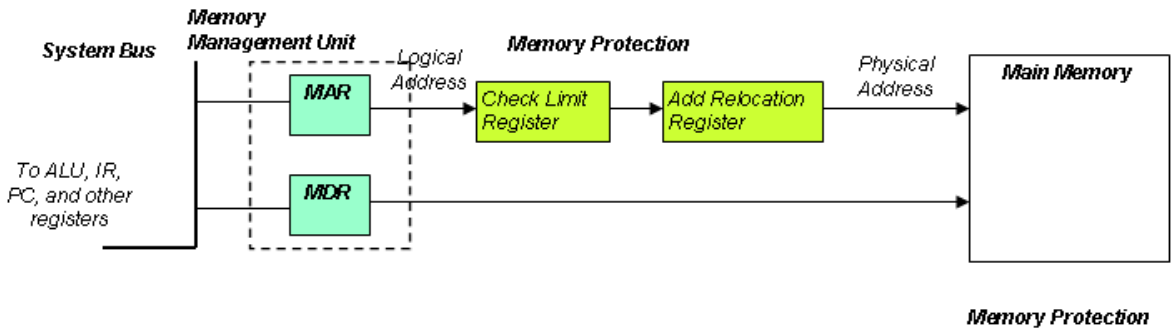
- The execution of the instruction at 06 is simple. It involves no memory operation.
- The execution of the instruction at 07 involves a memory operation.
    - The address 10 is in the logical address space but this address should be at 10 + 6 = 16 in the physical memory space.
    - At the execution phase, the address 10 is stored in the MAR.
    - The memory management unit then uses the sum of MAR and the relocation register as the actual address to store.
    - In the above program, the input value is stored at address 16.

The program should operate correctly.

7.2.3 Memory Protection

An effective technique to protect the memory space of each process is to control the access at the memory management unit.

- The memory management unit could use two registers: the relocation register and the limit register.
  - The relocation register specifies the starting address of the memory space of a process.
  - The limit register specifies the size of the memory space.
- Any memory operation in the execution phase is subject to a checking.
  - The logical address stored in the MAR is checked against the limit register.
  - If the MAR address is larger than the limit register, the corresponding physical address is outside the allocated memory space.
  - The physical address may be located in the memory space of another process.
  - This memory operation is considered hazardous and therefore aborted.



7.2.4 Example 3: Memory Protection

The program contains a STO instruction that stores data to an address way beyond its scope. This might be due to careless programming or a malicious intention.

Supposed that the following program is loaded at address 20. A piece of memory space is allocated to the process from address 20 to 23. The relocation register contains 20 and the limit register contains 4.

```
20 901; IN
21 380; STO 80
22 902; OUT
23 000; COB
```

In the execution phase of the execution of STO 80, the address 80 is stored in MAR. It is first checked against the limit register. Address 80 is outside the allocated memory space. The operating systems forbid this operation and a system exception would be raised.

7.3 Memory Management: Contiguous Allocation

To find free memory space for newly created processes, the operating systems must keep track of which part of the main memory is available.

Memory management provides services to allocate memory for processes, to release memory back to the operating systems, and other functions to maintain good utilization of memory.

A simple memory management technique is to consider that the memory space of a process is in one single block. The physical memory space of the process can be in one contiguous allocation.

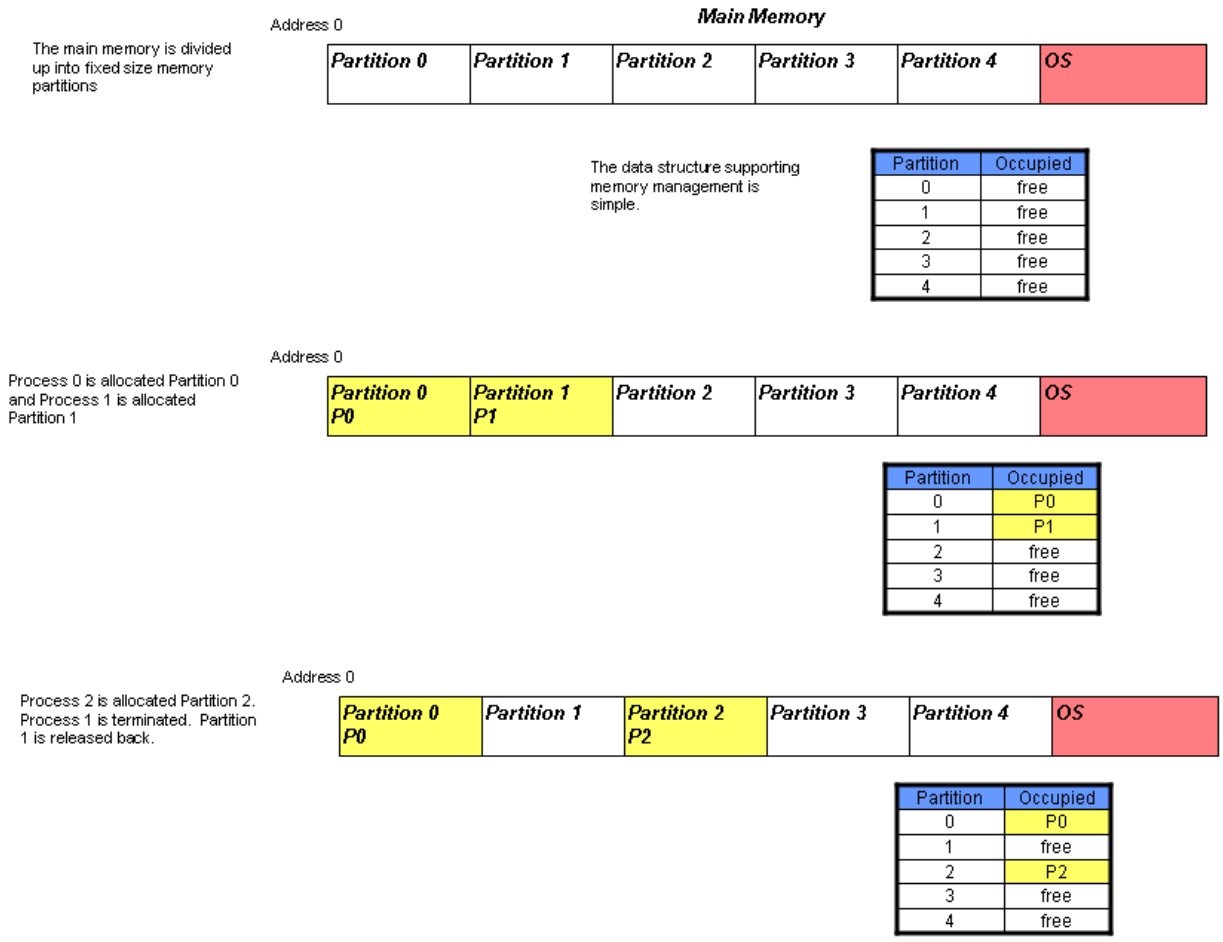
- Memory management requires a data structure to keep track of free and used locations in the main memory.
- The size and the type of the data structure are significant to the performance of memory management. The data structure itself occupies memory space. The type of the data structure affects the time needed to search and update records.

There are two major approaches in memory management: fixed size partition and variable size partition. They differ in how the main memory is allocated to processes.

7.3.1 Fixed Size Partitioning

The fixed size partition approach has the following characteristics:

- Only one memory block size is available to any process asking for memory. The block size should be large enough for the need of most processes.
- This approach is in trouble if a process needs a size larger than the fixed memory block size.
- The main memory is divided up according to the fixed memory block size. If there block size is not small, there is only a low number of memory blocks available. This number of available memory block decides how many processes can be loaded in the main memory and therefore the multi-programming level.
- The data structure to support memory management is relative small. A fixed length array (a table) containing whether each block is free or occupied is sufficient. Search for free partition and maintaining the data structure is efficient.
- This one-size-fits-all approach can result in low memory utilization. Some processes would need memory size much smaller than the fixed memory block. The unwanted extra memory is wasted.

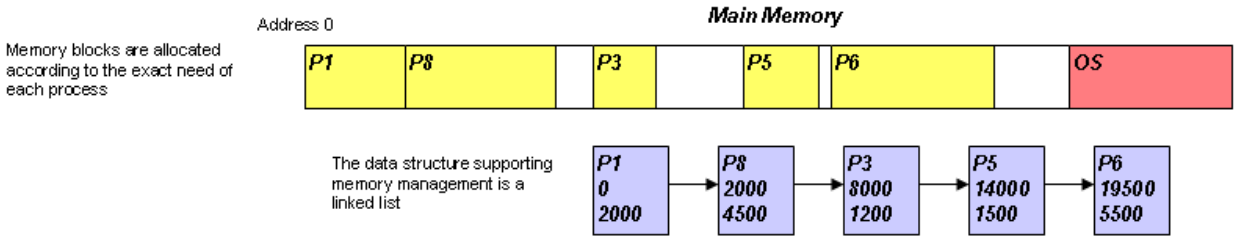


7.3.2 Variable Size Partitioning

The variable size partition approach has the following characteristics:

- Memory management allocates a memory block of size exactly equals to the need of any process asking for memory. Smaller processes are allocated smaller memory blocks. Larger processes are given large blocks if available.
- This is a very flexible approach that would satisfy the need of each process exactly.
- The data structure to support memory management is more complicated. The memory allocation (the starting address and the size of the memory block) of each process is recorded in a linked list. Searching involves traversing the possible many records in the linked list. Each record consists of the process ID, the starting address of the block, and the size of the block.
- Usually the nodes in the linked list are sorted according to the starting address.
- The problem of allocating memory to processes is known as the general dynamic storage allocation problem. Common strategies include the following.
    - First-fit. Start searching from one end to the other end. Allocate the first block (free space) that is large enough for the request. Once a suitable block is found, the search can stop. The searching may start from low to high or from high to low.
    - Best-fit. Search through the whole memory space and find the block (free space) that is just large enough for the request. The remaining free space in the free block is minimized.

- Worst-fit. Search through the whole memory space and find the largest block. The remaining free space in the free block is maximized.

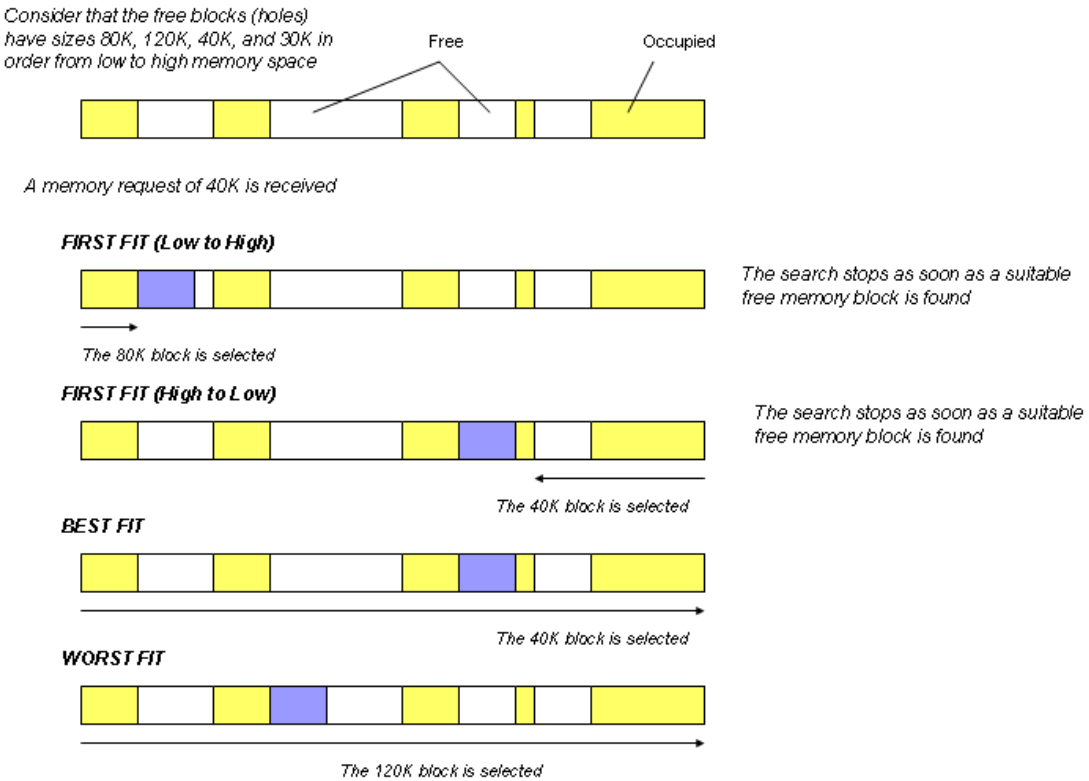


7.3.3 Example 4: Dynamic Storage Allocation Problem

Consider that the free blocks (holes) have sizes 80K, 120K, 40K, and 30K in order from low to high memory space. A memory request of 40K has arrived.

Work out how each of the allocation strategies of first-fit, best-fit, and worst-fit would handle the memory request.

- First-fit (from low to high): the 80K free block is selected, remaining 40K after allocation
- Best-fit: the 40K free block is selected, remaining 0K
- Worst-fit: the 120K free block is selected, remaining 80K



7.3.4 Fragmentation

The utilization rate of the main memory is related to the concept known as fragmentation. Fragments in the memory are memory space that is not usable or utilizable.

There are two types of fragmentation:

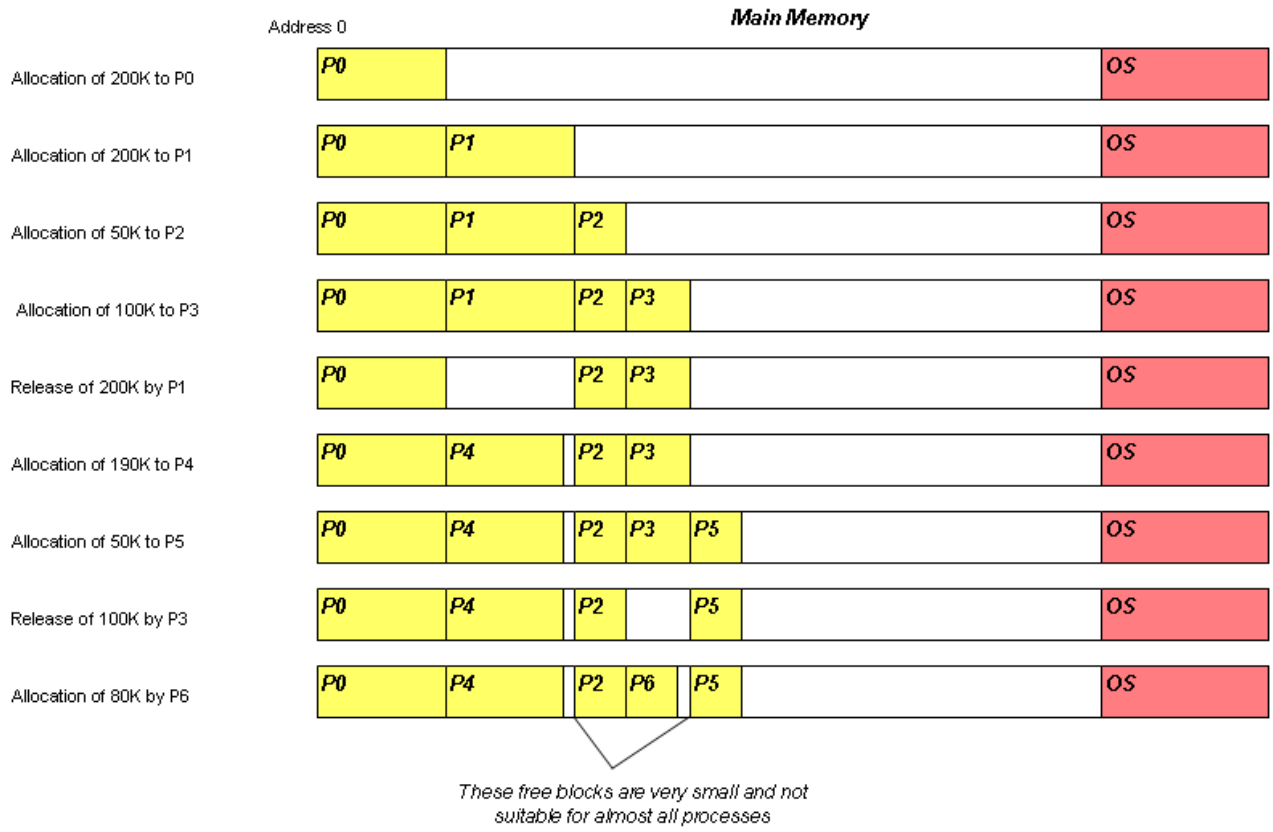
- In fixed-size partition scheme, the size of each partition is 200KB but a process requires only 150KB. There are 50KB of memory allocated to the process but it is not needed and used by the process.
  - The wastage is known as internal fragmentation.
- In variable-size partition scheme, arbitrary sizes of memory blocks are allocated and released. As time goes on, the available free memory is usually broken up into many small blocks or fragments. The small blocks are too little for nearly all processes and they are not utilizable.
  - The wastage is known as external fragmentation.

	Condition	***Memory Wastage***	***Remedial***
Internal Fragmentation	Happens in fixed-size partition scheme	On average half of partition size per process	Nothing can be done except reducing the partition size (which can cause other adverse effects)
External Fragmentation	Happens in variable-size partition scheme	Get worse as the system keeps running	Compaction

External fragmentation is usually caused by repeated allocation and release.

The following sequence of memory allocation and release would cause external fragmentation with variable-size partition scheme and first-fit allocation strategy.

- Allocation of 200K to P0
- Allocation of 200K to P1
- Allocation of 50K to P2
- Allocation of 100K to P3
- Release of 200K by P1
- Allocation of 190K to P4
- Allocation of 50K to P5
- Release of 100K by P3
- Allocation of 80K by P6



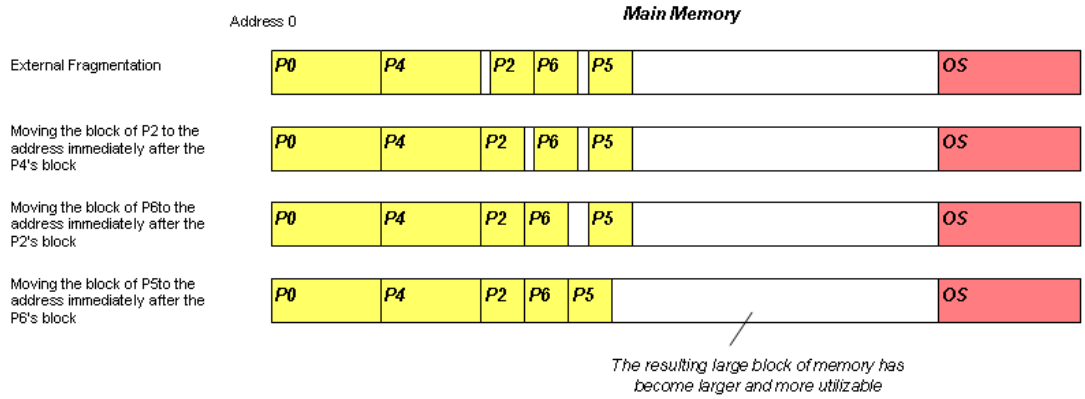
External fragmentation is a problem that can be fixed by compaction.

- Compaction is carried out with moving allocated blocks together by squeezing out the small and not-utilizable free blocks.
- Compaction is a costly operation. It involves a lot of data copying.

7.3.5 Example 5: External Fragmentation Solved by Compaction

The external fragmentation situation shown in the above example can be solved by compaction. The steps are:

- Checking there is no spare space before P0.
- Checking there is no spare space before P4.
- There is spare space before P2. Copy P2 block to the address immediately after P4 block.
- There is now spare space before P6. Copy P6 block to the address immediately after P2 block.
- There is now spare space before P5. Copy P5 block to the address immediately after P6 block.
-

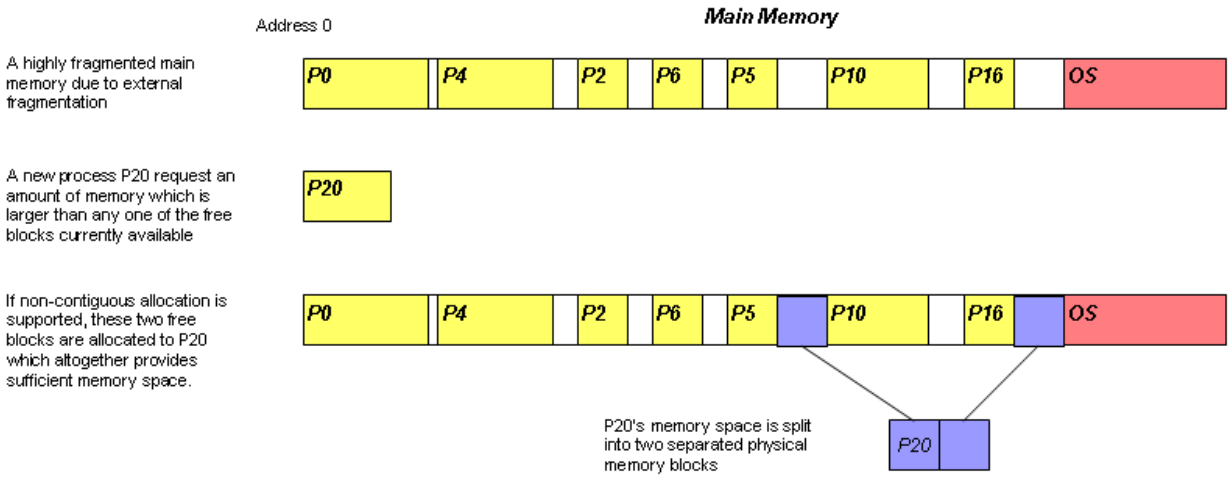


7.4 Memory Management: Non-Contiguous Allocation

There are advantages if the memory space of a process can be split into several memory blocks.

Each of the memory blocks is in a different address in the main memory. This way of allocating non-contiguous free memory block to a single process is known as non-contiguous allocation.

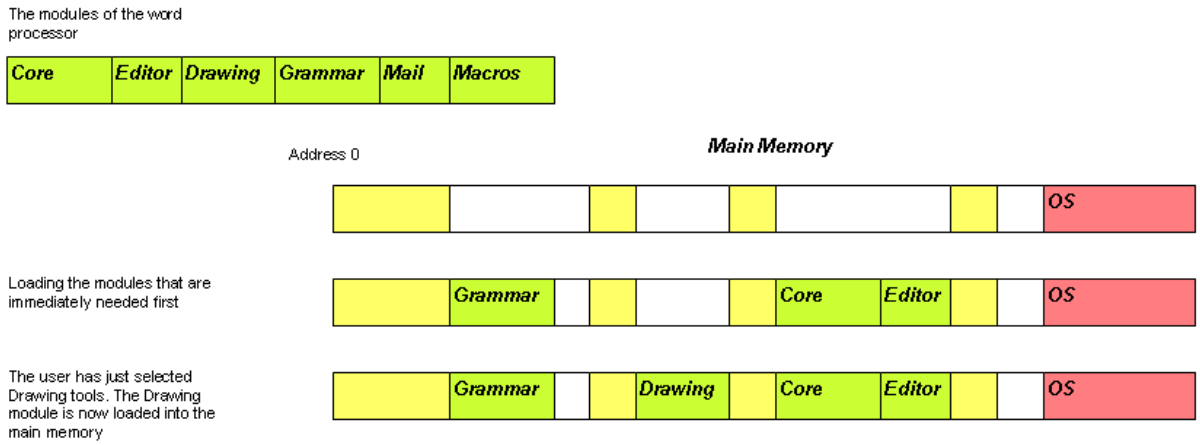
- It reduces the wastage caused by external fragmentation.
  - Small free memory blocks become more utilizable
  - A large memory block request could be satisfied by combining small free memory blocks.
- It reduces the demand on the main memory.
  - The execution of a process usually focus around a section of instructions at a time.
  - For example, given a long LMC program of 80 instructions, at a particular moment the execution would stay inside a loop made up of 12 to 15 instructions.
  - The instructions outside that active section are really not needed at the time.
  - The loading of the other sections can be delayed until they are needed.
  - This concept is known as dynamic loading.
- It facilitates sharing of memory between processes and further reduces the demand on the main memory.
  - Most programming languages provide common libraries that are often used by most programs.
  - For example, the printf function is used in almost every C programs. Instead of including the printf function instructions integrated in all C programs, it makes more sense if the printf function instructions are extracted out and made into a module sharable by all executing C programs. For example, even if there are 1000 thousand C programs running on a computer system, only one copy of the printf function module is loaded in the main memory.
  - This approach is usually supported by dynamic linking.



7.4.1 Better Utilization of Main Memory: Dynamic Loading

Dynamic loading delays the loading of a program module until it is actually required for program execution.

- This is only possible if non-contiguous memory allocation is supported. In non-contiguous memory allocation, the memory allocation for a process can be carried out multiple times. For example, a process may ask for one half of its memory need first and then another half later.
- Dynamic loading hopes to make better utilization of the main memory by loading in the modules that being executed (or expected to be executed). The not-needed parts are stored in secondary memory.
- Dynamic loading also speeds up the starting time of a large application because of the fewer amounts of data to load from secondary memory into main memory.
- The actual selection and loading of a module is the responsibility of the application itself.
- Dynamic loading is most useful with very large applications in which most modules are rarely used.
- The users would experience a delay when selecting a function of which the module is yet to be loaded into the main memory.



7.4.2 Example 6: Dynamic Loading of a Word Processor

Word processors are very large applications, providing a large number of functions. However, most of the functions are rarely used. It makes sense to dynamic load them in as and when needed.

A certain word processor is divided into the following modules. The size of each module is indicated in brackets:

- Core (10M)
- Editor (3M)
- Grammar checker (8M)
- Drawing tool (5M)
- Mail merge (6M)
- Macros (12M)

The Core, Editor, and Grammar checker of the word processor may be loaded first. The other modules are loaded when the user invokes the relevant functions. The loading time is significantly reduced. The amount of memory use is also reduced.

7.4.3 Sharing of Memory: Dynamic Linking

Dynamic linking allows different processes to share a common piece of code. If a piece of code, such as the `printf` function, is needed by a number of processes, it saves memory space if the processes share the same copy instead of having their own copy.

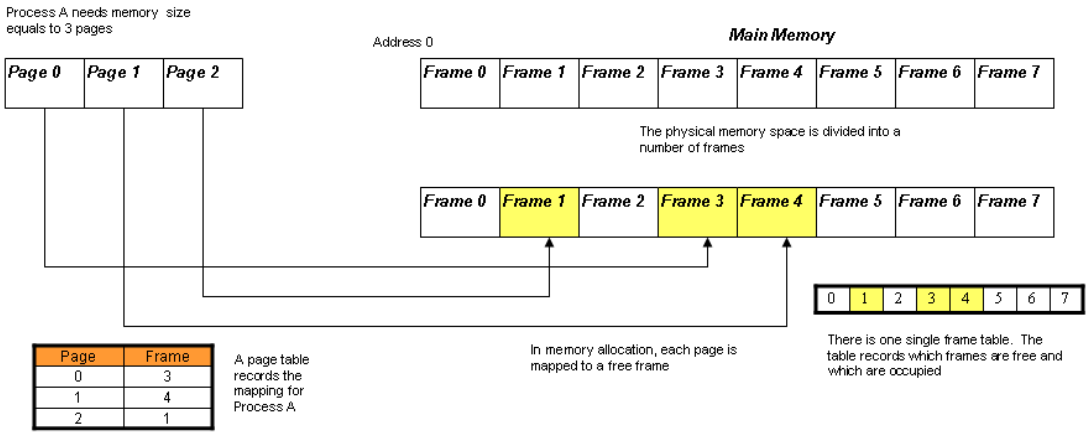
- Dynamic linking allows a module to be called and executed through a small piece of code known as a stub. Dynamic linking is used with dynamically linked libraries (DLL). The stub is programmed so that it checks whether the required routine is in memory, and if not, it loads the routine from the hard disk into the memory.
- The stub is inserted into the program at the code generation phase of the compilation process.
- Dynamic linking requires the support of the operating systems.
  - A required routine may already be loaded in the memory, being used by another process.
  - It is the job of the OS to check whether the routine is there.
- The sharing of a module between processes is a potential security hazard.
  - A malicious process would try to alter this common module so that other processes are harmed by the change.
  - A solution is to impose restriction such as read-only or execute-only in this module's memory block.

7.5 Paging

Paging is a memory management scheme based on:

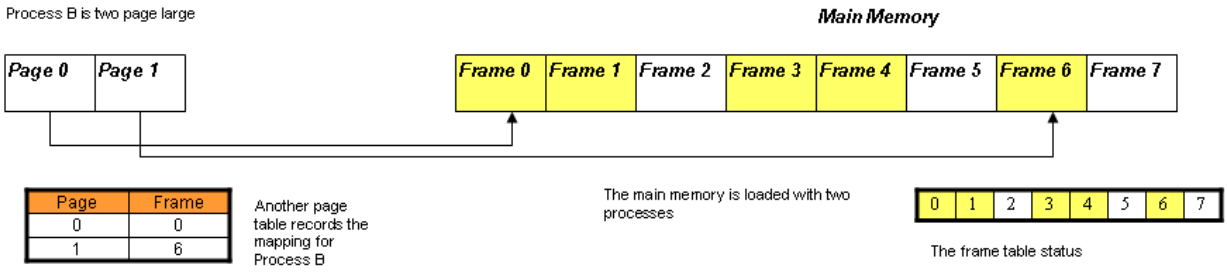
- Non-contiguous memory allocation, and
- Fixed-size partition scheme.

A process's memory space can be split up into several parts, each of which may be loaded into the main memory at different addresses. The parts are all of a standard size.



The following lists the core ideas of paging.

- The physical memory space is broken up into a number of memory blocks of the same size.
  - Each memory block is called a frame.
  - Each frame has a unique ID, starting from 0.
- Normally the size of each frame is not too large.
  - For convenience of management, the size is usually a power of 2.
  - Typical frame size is 4KiB (which is 4096 bytes).
- The logical memory space of a process is also broken up into a number of blocks of the same size.
  - Each memory block is called a page.
  - In a particular computer system, the page size is the same as the frame size. Each page has a unique ID.
- The loading of a program in a paging system happens as follows. For each page, find a free frame in the main memory and load the page into the free frame. Each page is loaded into a different frame in the main memory. The mapping of each page to the corresponding frame is stored in a data structure known as the page table.
- A page table per process.



7.5.1 Example 7: Number of Frames and Pages

Given a paging system with page size (frame size) of 4096 bytes. The size of the main memory is 1Mbytes. How many frames are there?

Size of main memory is 1Mbytes or 220 bytes.

Size of each frame is 4096 bytes or 212 bytes.

Dividing the main memory up with frames, there are 220 divided by 212 frames.

= 28 frames = 256 frames

The frame ID starts from 0 and the last frame ID is 255.

A process is created with memory requirement of 16000 bytes. How many pages are in its logical address space?

Frame size is the same as page size. Page size is also 4096 bytes or 212 bytes.

The number of pages required is 16000 / 4096 = 3.91 pages

Three pages are not enough.

The actual number of pages in the logical address space is 4.

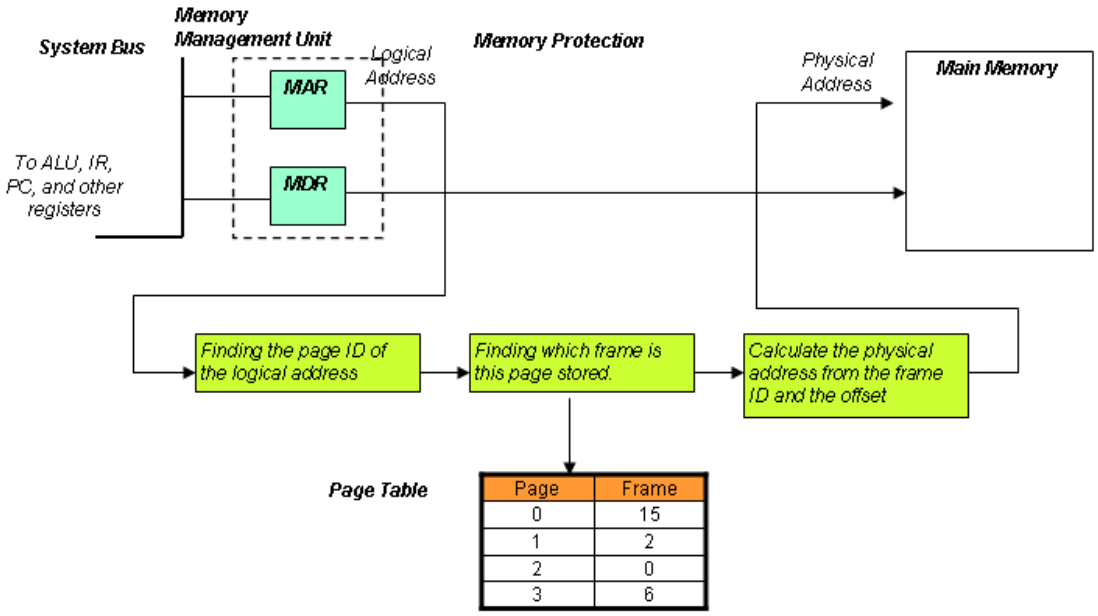
The system is multi-programming and on average each process's logical address space has 6 pages. Estimate how many processes can be loaded into the main memory at a time.

There are 256 frames and on average each process needs 6 frames. So the estimation is 256/6 = 42.67 processes = 42 processes

7.5.2 Paging: Conversion from Logical Address to Physical Address

Each conversion of a logical address into the corresponding physical address has to consult the page table. In other words, every memory operation in a paging system involves accessing the page table.

- If the page table is stored in the main memory, then each memory operation involves actually two memory read/write operations.
  - The first is to access the page table.
  - The second is the actual memory operation.



7.5.3 Example 8: Conversion of Logical Address into Physical Address 1

Given a paging system with page size (frame size) of 4096 bytes. The main memory has 1024 frames. So the frame ID ranges from 0 to 1023. The logical address space of a process has 6 pages.

What is the size of the logical address space of the process?

The size of logical address space = 4096 x 6 = 24576

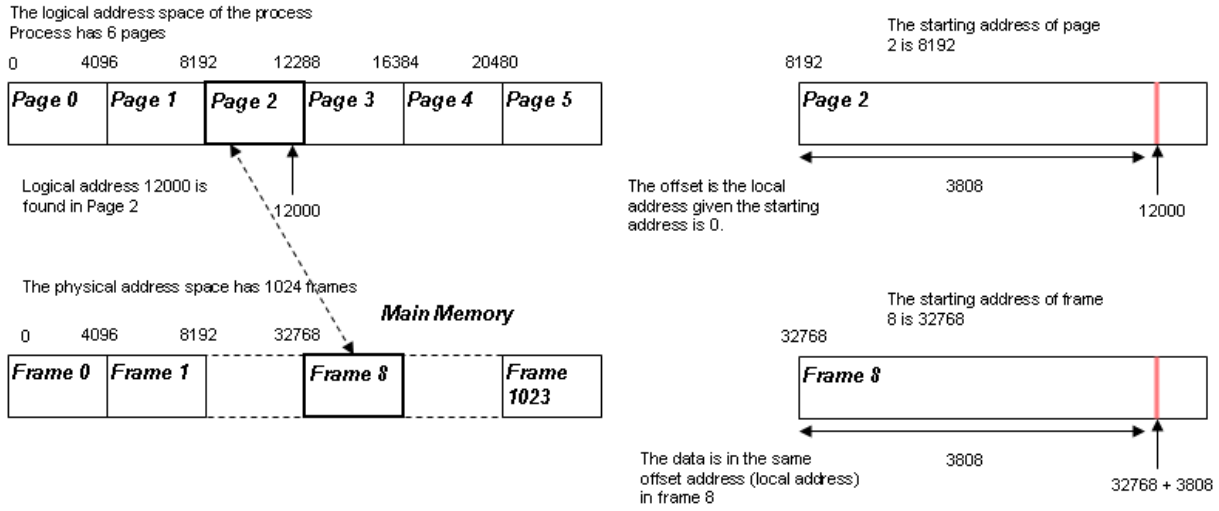
Given the logical address of 12000 and the following page table content, work out the corresponding physical address.

Page	Frame
0	600
1	23
2	8
3	1
4	245
5	246

The steps include the following:

- Work out which page contains logical address 12000. The page containing the address is 12000 / 4096 = 2.92 = Page ID 2.
- Work out the offset of logical address 12000 in page 2. The offset is the local address in page 2. Offset address = 12000 % 4096 = 3808.
- Work out where page 2 is stored in the main memory. The page table says page 2 is mapped to frame 8.
- Work out the starting address of frame 8. Starting address of frame 8 is 4096 x 8 = address 32768.
- Work out the physical address of logical address 12000. It is offset address 3808 in frame 8 = 32768 + 3808 = address 36576.

Logical address 12000 is found in physical address 36576.



7.5.4 Example 8: Conversion of Logical Address into Physical Address 2

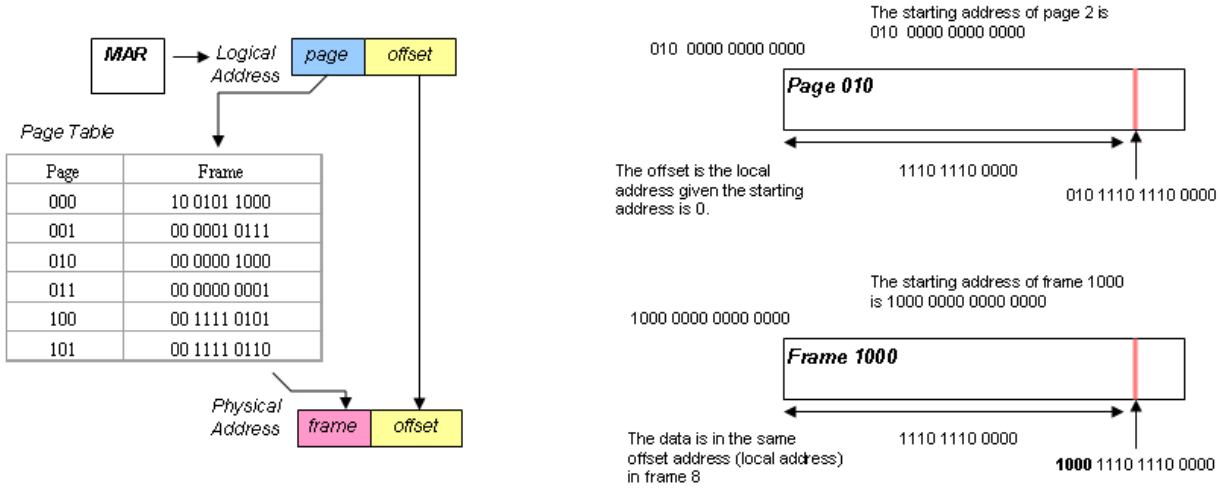
Given a paging system with parameters specified above, calculate the physical address of logical address 81200.

- Work out which page contains logical address 81200. The page containing the address is 81200 / 4096 = 19.82 = page ID 19.
- The logical address space of the process has only 6 pages. This address clearly lies beyond the legitimate memory block. This is either due to careless programming or malicious intention. A system exception would be raised.

7.5.5 Implementation of Page Table: Page Size

Because every memory operation involves accessing the page table, logical-address to physical-address conversion and page table must be implemented in a most efficient manner.

- Ensuring the page size a power of 2 can significantly speed up the conversion from logical address to physical address. No more addition operation is required in the conversion process. It becomes simply bit replacement.
- If the page size is a power of 2, then all logical addresses can be separated in two parts. Given that the page size is 2n, then the n number of least significant bits will form the offset address of the page, and the remaining more significant bits will form the page ID.
- For example, given a page size of 4096, n would be 12. The offset address of a page ranges from 0000 0000 0000 to 1111 1111 1111.
- If the logical address is 0011 1101 1111 1000, then the page ID is 0011 (assuming at most 16 pages) and the offset address is 1101 1111 1000.



7.5.6 Example 9: Conversion of Logical Address into Physical Address 3

This example re-attempts the first example but in binary values.

Given a paging system with page size (frame size) of 4096 bytes (which is 212). The main memory has 1024 frames (which is 210). Theframe ID ranges from 00 0000 0000 to 11 1111 1111. The logical address space of a process has 6 pages. The page ID ranges from 000 to 101.

Given the logical address of 12000 (binary 010 1110 1110 0000) and the following page table content, work out the corresponding physical address.

Page	Frame
000	10 0101 1000
001	00 0001 0111
010	00 0000 1000
011	00 0000 0001
100	00 1111 0101
101	00 1111 0110

The steps include the following:

The page ID is clear from the logical address. The 12 least significant bits are the offset address. The first 3 bits are the page ID.

Offset address = 1110 1110 0000 and Page ID = 010

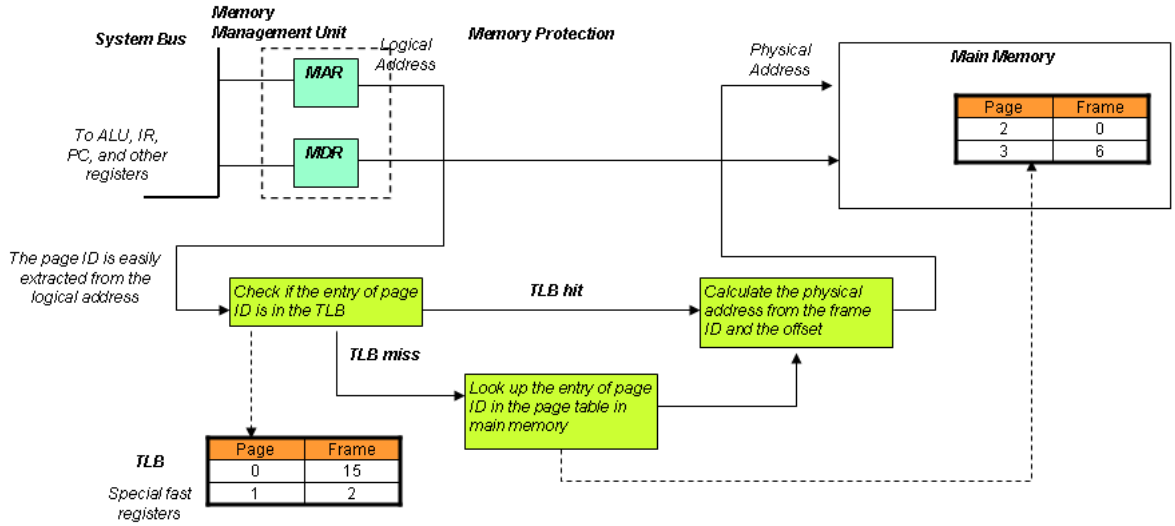
The physical address is simply obtained by replacing the page ID with the corresponding frame ID. The frame ID for the page ID 010 is **00 0000 1000.**

The physical address is **00 0000 1000** 1110 1110 0000. In decimal this value is 36576. We have got the same result without the need to add numbers.

7.5.7 Implementation of Page Table: Translation Look-aside Buffer

The conversion of addresses is still an expensive operation even without the need of addition operation. Each memory operation involves two memory read/write operations: reading the page table, and the actual memory operation.

- To maximize efficiency in address conversion, page tables are implemented in a set of high-speed registers, supported by dedicated hardware for address mapping.
- This solution becomes infeasible if the number of entries in the page table is large. It is too expensive to have a lot of registers. It is more economical to have the page table stored in the main memory.



A balanced solution is to have part of a page table stored in high-speed registers, and the other part in the main memory. The high-speed registers are known as the translation look-aside buffer (TLB).

- The operating systems guess which part of the page table would be accessed frequently. That part is cached in the TLB for quicker access.
- It is possible to guess correctly because it is common to have consecutive memory address operations localized (i.e. around same address range). Localized access is a common phenomenon in programming. For example, executing a program loop repeatedly would see a contiguous section of addresses of instructions being executed repeatedly. It is reasonable to assume that this section belongs to the same page (or the neighbouring pages).
- The TLB is established when a new process is scheduled to run, and it is flushed when the process is removed. Some entries are important, such as address of the kernel, are never removed and called wired down.

7.5.8 Example 10: Effective Memory Access Time with TLB

The effective memory access time (EMAT) is the average amount of time taken to perform a memory operation. Assume that each access to the main memory requires 100 ns. However in a paging system, the EMAT is not 100 ns. The conversion of logical to physical address requires an addition access to the page table, which is in the main memory. So the EMAT in a paging system is 100 ns (page table) + 100 ns (actual operation) = 200 ns.

The introduction of TLB in the conversion process can speed up the EMAT considerably. Access to TLB is usually much faster. Let's assume a small figure of 20 ns. If the whole page table can be stored in the TLB, then the EMAT becomes 20 ns (page table in TLB) + 100 ns.

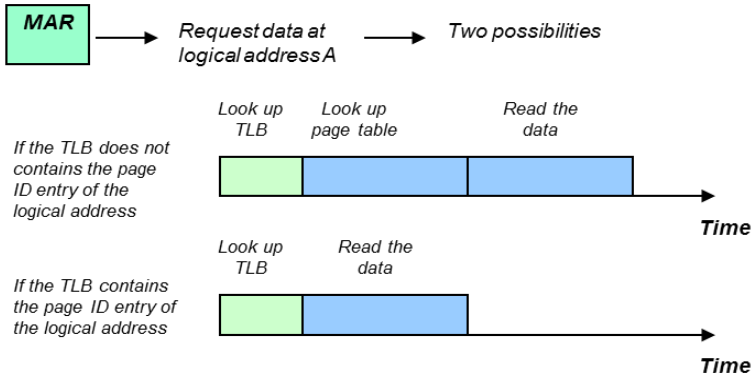
However, in real situations, the TLB is large enough to store part of the page table. We can still calculate the TLB if we know the hit rate. The hit-rate is the percentage that a page table entry is found in the TLB.

Calculate the EMAT if the hit-rate is 80%.

Time taken if page table entry is found in TLB = 20 ns + 100 ns = 120 ns

Time taken if page table entry is not found in TLB = 20 ns + 100 ns + 100 ns = 220 ns

EMAT = 0.8 x 120 ns + 0.2 x 220 ns = 140 ns



7.5.9 Memory Utilization in Paging System: the 50-percent Rule

Paging system still suffers from internal fragmentation. The amount of wastage caused by fragmentation can be estimated using the 50-percent rule.

- The 50-percent rule says that given N allocated memory blocks, another 0.5N blocks would become fragmented and not usable. So for every (N + 0.5N) allocation of blocks, 0.5N would be not usable. Statistically one-third of memory is not usable.
- It is useful estimating wastage in fixed-size partition schemes such as paging systems.
- Given that the page size is S bytes. A process P is asking for memory block size of B bytes. The logical memory space of the process will need B / S. The last page is usually not fully utilized because B is unlikely to be wholly divisible by S. This is internal fragmentation. The wastage is B % S, a number between 0 (zero wastage) to S – 1.
- If the size of memory request is random, each of the wastage amount from 0 to S-1 is equally likely, and the probability would be 1 / S.
- On average, the amount of wastage is the weighted average wastage:

  Average wastage = (1/S)(0) + (1/S)(1) + (1/S)(2) + … + (1/S)(S-1) = 0.5 S

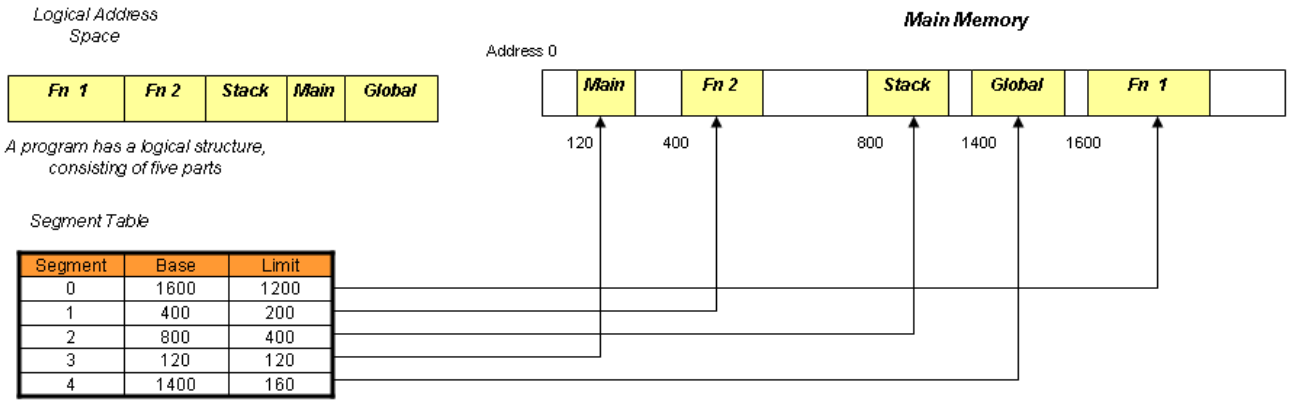
  The average wastage is half the page (block) size.

7.6 Segmentation

Segmentation is a memory management scheme that is also based on non-contiguous memory allocation. It is however a variable-size partition scheme. A key feature of segmentation is that the process's logical memory space is split up according to its logical structure.

The following lists the core ideas of segmentation.

- Segmentation takes the perspective of structure of programs.
- Programs often exist in individual segments (functions and data). Some segments come from different sources (e.g. runtime libraries). If a program is to be split up, there is no reason why separation is based on this logical structure.
- Each segment has three parameters: the segment ID, starting address in the main memory (the base address), and the segment size (the limit). The first logical address in every segment is zero and this is corresponding to the starting address in the main memory.
- Segmentation is a scheme that models the logical memory space with a segment number and an offset. The compiler should be able to work out the segment number and offset automatically, and requires no intervention from the users.
- Each process has a segmentation table containing a segment number, the base address of the segment in the physical memory, and the limit that indicates the size of the segment.



An important advantage of segmentation is the sharing of segments between processes. Certain segments may often be library routines that many processes would use. Once such segment loaded into the main memory, it is for all processes to use.

- To protected security between processes, segments must be protected. Each segment may be defined as read-only or execute-only. Such segments are often shared routines that they can be accessed safely from all processes.
- The hardware is responsible to prevent illegal access to physical memory addresses either outside the segment, or to a protected segment. The limit register is used to check against the offset address.
- A main drawback of segmentation is external fragmentation. Each segment is of variable length, and therefore it suffers the same consequence of a variable length memory allocation scheme. It would create holes in the memory too small for most requests.

7.6.1 Segmentation: Conversion from Logical Address to Physical Address

In segmentation, every memory operation involves consultation of the segmentation table and an addition operation.

- The logical address is in the form of (segment ID, offset address).
- The physical address is calculated by first looking up the base address and the limit of the segment. If the offset address is larger than the limit, then the address is one outside the segment boundary. This is a hazardous operation and a segmentation error would occur

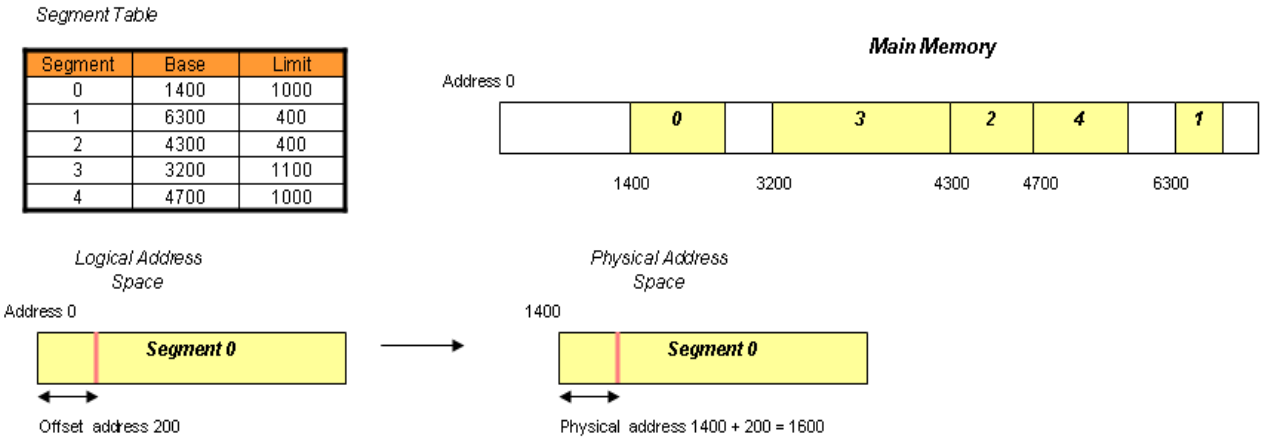
- The physical address is calculated by adding the offset address and the base address.

7.6.2 Example: Conversion of Logical Address into Physical Address in Segmentation

Given the following segment table, calculate the following logical addresses.

Segment	Base	Limit
0	1400	1000
1	6300	400
2	4300	400
3	3200	1100
4	4700	1000

- Address (0, 200)
- Address (1, 6)
- Address (2, 500)
- Address (3, 900)
  - Address (0, 200) means segment 0 and offset address 200. Offset address 200 is within the limit 1000. The physical address is 1400 + 200 = 1600
  - Address (1, 6) means segment 1 and offset address 6. Offset address 6 is within the limit 400. The physical address is 6300 + 6 = 6306
  - Address (2, 500) means segment 2 and offset address 500. Offset address 500 is outside the limit. Segmentation error would be raised.
  - Address (3, 900) means segment 3 and offset address 900. Offset address 900 is within the limit 1100. The physical address is 3200 + 900 = 4100



7.7 Swapping: Efficient Use of Physical Memory

In a single-processor system, only one process can be in the Running state (being executed by the CPU) at a time. Even if there are many processors loaded into the main memory, only one of them is actually running.

- The level of multi-programming depends on the size of the main memory.

- We can estimate the number of concurrent processes by a simple division.
- For example, if the main memory size is 1Mbytes, and on average each process needs 100Kbytes, then at most 10 processes may be loaded into the main memory at a time.

- We could actually increase the level of multi-programming with a simple idea.
  - A process that is not currently executing may be swapped out to the secondary memory (hard disk).
  - This creates free memory space for new processes to use.
- This scheme is feasible as long as the process scheduling is monitored.
  - If the process is near the front of the ready queue, then it will be executed soon.
  - The swapped out data is swapped back in the same memory space.

Swapping works best with the round-robin process scheduling algorithms.

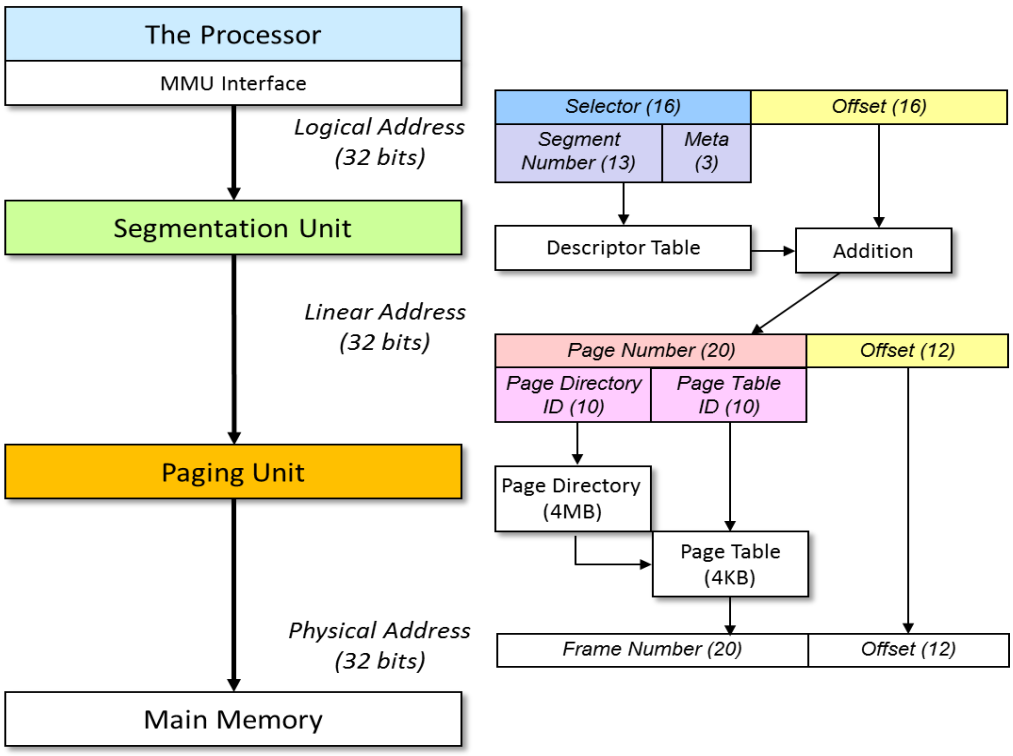
- Roll-in, roll-out refers to the process of swapping-in higher priority process and in doing so swapping-out lower priority process. When the higher priority process has completed the execution, the lower priority process is swapped in.
- Processes waiting for I/O should never be swapped because the I/O may perform DMA or other asynchronous access to the process memory space for data buffer.

Swapping allows that the total demand of memory of the processes can exceed the physical memory size.

7.8 Case Study: The Intel IA-32 Architecture

The Intel processors and chip-set have been the most popular architecture on PC computers for many years. The IA-32 (i.e. the series of 32-bit chips including the Pentium) uses a combination of segmentation and paging in the logical-to-physical address conversion.

Intel IA-32 Architecture



Logical Address

The 32-bit logical address has a selector part (a.k.a. segment ID) and the offset part. IA-32 supports two types of descriptor tables (i.e. segment table).

- Local Descriptor Table, which stores up to 8192 segments private to the process.
- Global Descriptor Table, which stores another 8192 segments maximum that are shared between processes.

The logical address is checked against the memory limit registers for system faults. Then the address is converted to the corresponding linear address based on the descriptor tables. The linear address is produced by adding the mapped output of the descriptor table and the offset address.

Linear Address

The next step involves the paging unit converting the linear address into physical address. IA-32 supports two page sizes, namely 4MB and 4KB, for efficiency, and adopts a two-level paging schemes for the mapping of addresses.

- The first 10-bit of the linear address points to a page directory. There is a flag in each entry that indicates whether the page is 4MB or 4KB.
- If the page size is 4MB, then only the first 10-bit is needed for the mapping to frame number.
- If the page size is 4KB, then the next 10-bit is needed as the page number with a page table.

The IA-64 architecture (64-bits) has a maximum physical memory size of 2^{64} bytes. However, in practice only 48 bits are used in representing addresses. It supports page sizes of 4KB, 2MB, or 1GB (if supported) page sizes. The linear address consists of four levels of paging hierarchy instead of two.

An important change in IA-64 is that segmentation is not needed anymore. As of modern day computing requirements, 64 bit addressing seems to be sufficient. Paging can cover the whole addressing space.

COMPS267F Operating Systems

Virtual Memory 8

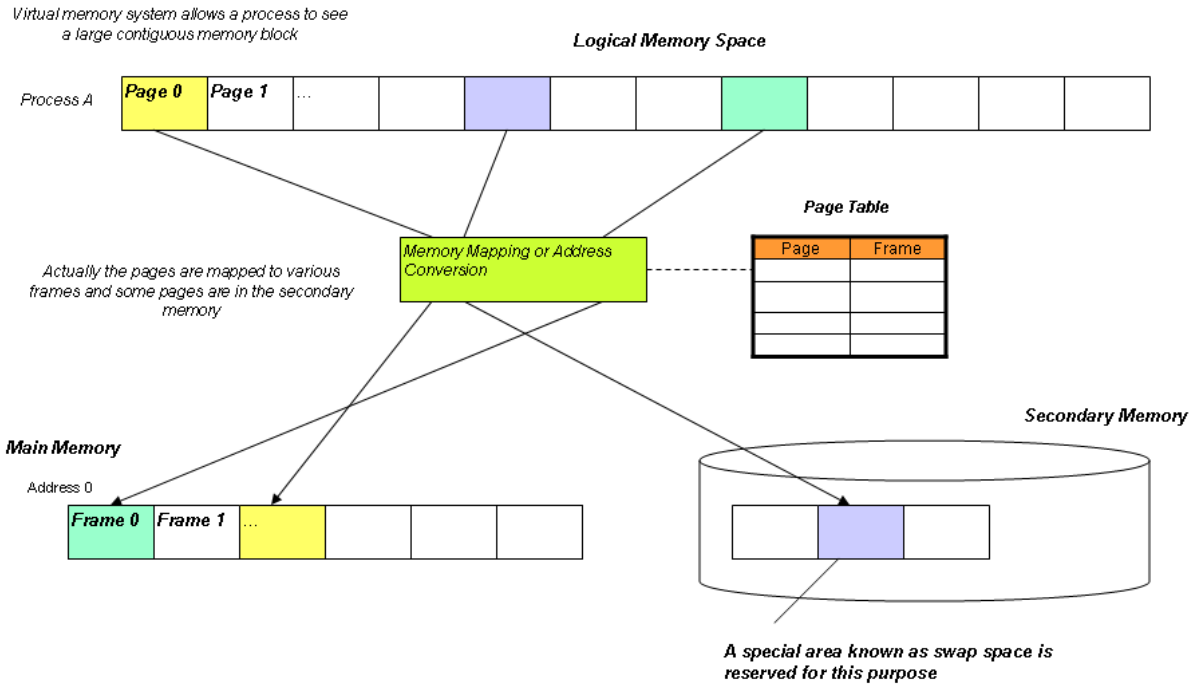
There is an increasing demand for larger main memory size. Application programs are increasingly larger in size. The average file size is also increasing, thanks to the generation of more multimedia data.

This chapter discusses techniques to make the main memory larger than its actual size. The techniques project an illusion that makes the main memory looks larger to processes. The relatively cheap secondary memory is exploited here.

8.1 Overview of Virtual Memory Systems

Virtual memory is a concept that is more or less a cumulation of features in memory management systems that deals with the separation of logical memory space and physical memory space.

- It is a layer of abstraction introduced to allow a very large program to be executed with relatively little physical memory.
- The virtual address space is the logical view of a program stored in a large contiguous block of memory. Of course in reality the program would be stored in multiple separated blocks in physical memory. Virtual memory system manages and provides the illusion.
- The memory management unit performs the conversion of logical addresses to physical addresses. It uses a page table to know the mapping between logical pages to physical frames.



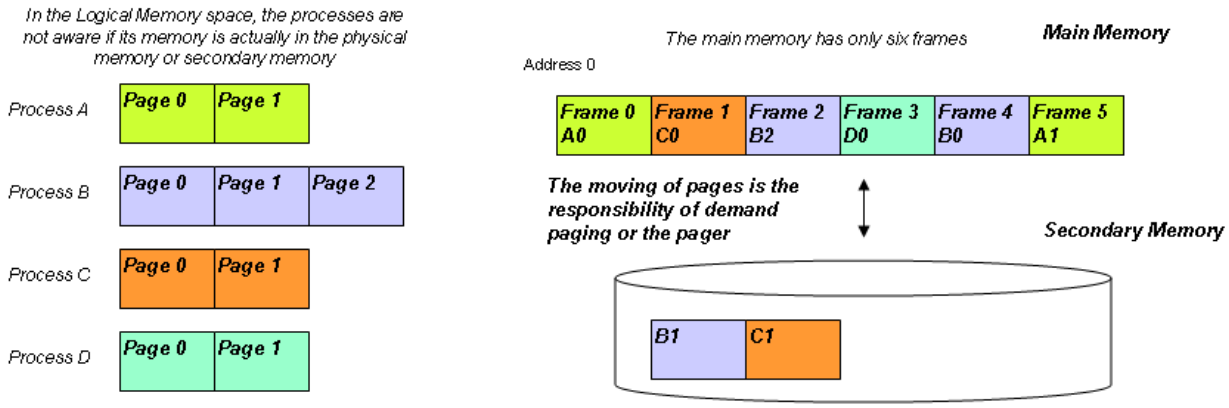
Another feature of virtual memory is to allow an executing program only partially loaded into the main memory. Loading program partially is acceptable because of the nature of programs.

- Only a fraction of the functions of a program is needed. Consider how many functions you would need when using Microsoft Word.
- Many branches in a program often deal with error handling. They are rarely executed.
- Data structures are often declared with a size larger than needed. For example, arrays for input buffer are often large so that input can be handled safely.

The capability to allow an executing program only partially loaded in the main memory is advantageous in a number of ways:

- The size of the logical memory space of a process can now be greater than the physical memory size. For example, even if a computer system has only 1GB memory, we can write a program that declares an array of 2GB size.
- More programs (or processes) can be executed at the same time. This increases the level of multi-programming.
- The program start-up time is faster because of the less loading time needed. Only the currently needed parts of the program are loaded to the main memory first.

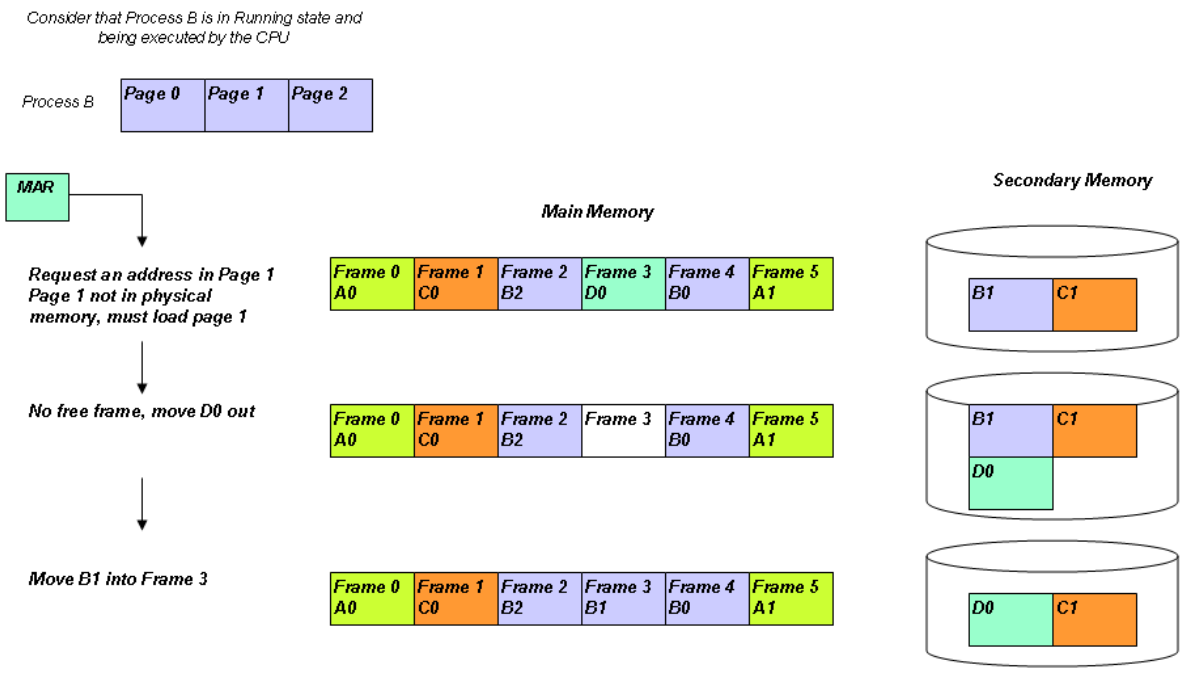
In summary, virtual memory system creates a memory illusion to a program that the memory space is large and contiguous. Virtual memory system hides away that fact that some pages are in different parts in the main memory and other pages are even saved in the backing store in secondary memory.



8.2 Demand Paging

Virtual memory can be implemented by a demand-paging system. Demand paging is a technique that allows the pages to load into the main memory when they are actually needed.

- Demand paging is another version of swapping. Demand paging deals with copying in and out individual pages between the main memory and the secondary memory.
- When a page is not needed at this moment, it is a candidate to be copied out to the secondary memory. The resulting free frame can then store another page copied in from the secondary memory. The process is the responsibility of a pager.
- In the ideal case, the pager will move in the pages that will be needed in the near future. If a page is moved in to the main memory, but soon the whole process is swapped out, then it would increase the demand on the main memory and IO unnecessarily.



The following lists some concepts relevant to demand paging.

- Pure demand paging is a scheme that never loads in a page until it is needed, including the very first page when the program begins its execution. This is a special case in which a process in the Ready state is not loaded into the main memory at all.
- Swap space refers to the space allocated in the hard disk to store swapped out pages. Disk access to the swap space is generally faster than access the file systems.

8.2.1 Page Table in Demand Paging

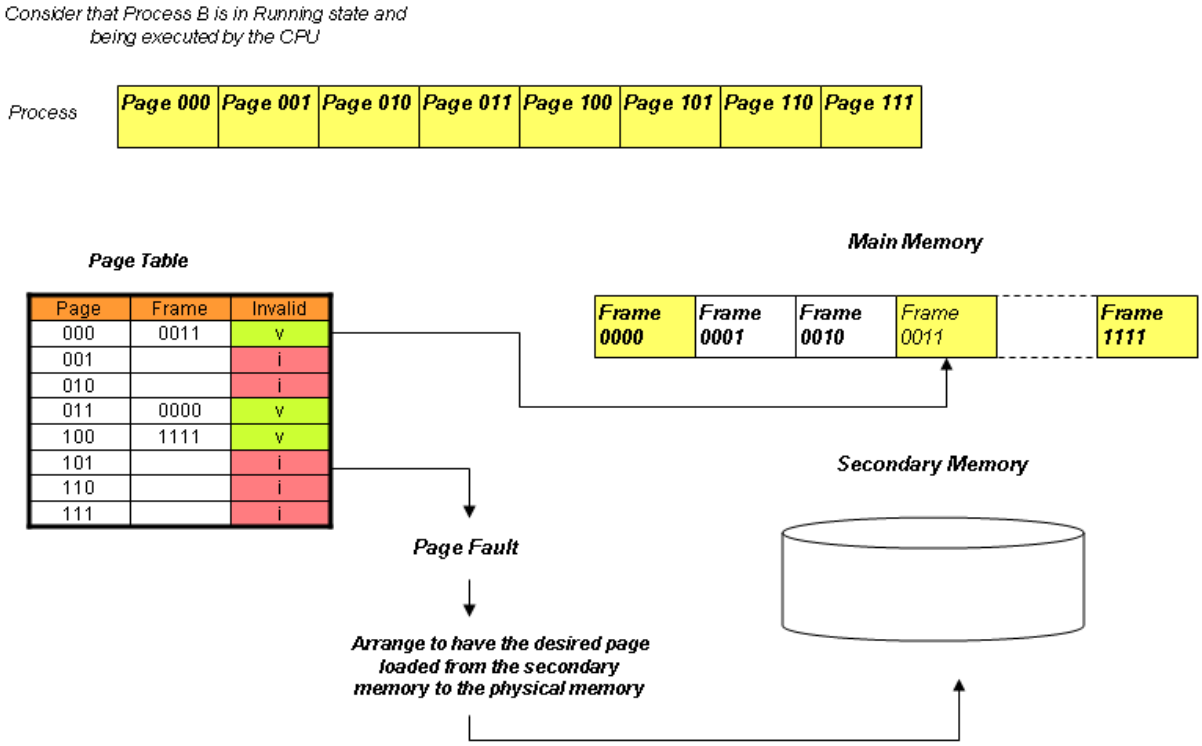
In a memory operation, a possibility is that the memory address may be in any of the frames in the main memory, or it is moved out to the backing store. The page table needs augmentation to support memory operation where the page is stored either in main memory or hard disk.

- A valid-invalid bit is used to indicate whether a certain page is currently in main memory. This is an additional column in a page table, which is also used to indicate whether an illegal memory access has occurred.
- Access to a valid page means that the memory address is in main memory and so the execution can continue.
- Access to an invalid page would cause a page fault. This is usually a trap (interrupt) that handles the execution back to the kernel.

The handling of a page fault involves a number of steps.

- First the kernel checks whether this is a true page fault or an illegal memory access. If it is the latter, then the process is terminated.
- A free frame in the physical memory is found (from frame table or free-frame list) and then an I/O request is scheduled to load in the page to the free frame.
- If there is no free frame, arrange to move out an existing page from the main memory to the backing-store. The process of selecting a page to move out is known as page replacement.

- When the I/O request is complete, the tables are updated to reflect the current status and then the instruction that caused the page fault is restarted.



In actual operation of demand paging system, the rate of page fault is fortunately not too high.

- Page fault handling involves a number of IO operations and so page fault is a costly situation. Memory access time is in the tens of nanosecond range, while the disk access time (including loading) is in the millisecond range. A difference in the scale of hundreds of thousands.
- The service time for the page fault mainly comes from the following three components page-fault interrupt handling, load-in the page, restart the process.
- Multiple page faults per instruction are possible if a page fault occurs to the access to the instruction and the data. This would cause a very poor system performance.
- Locality of references refers to the case that programs tend to refer to nearby memory addresses in batches (i.e. continuously). Consecutive memory access tends to refer to the address in the same page. This keeps the number of page faults low.

8.2.2 Example: Effective Memory Access Time in Demand Paging System

The effective memory access time (EMAT) is the average amount of time taken to perform a memory operation.

If p is the probability of page fault, the effective access time is given in the following.

EMAT = (1 – p) * time (memory access) + p * time (page fault handling)

In handling a memory operation, there are two possibilities:

1. Page fault not occurred: the time taken is time (memory access).

2. Page fault occurred: the time taken is time (page fault handling).

Normally the page fault handling time is several orders of magnitude higher than memory access time.

Assuming that the memory access time is 100 nanoseconds, and the page-fault service time is 25 milliseconds. The page fault rate is one in a thousand. Evaluate the effective memory access time.

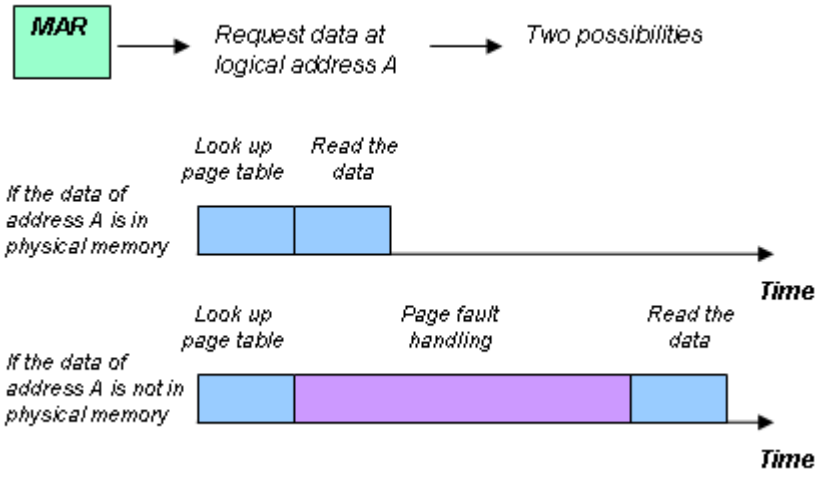
Assume that the page table is stored in the main memory.

EMAT = (1- 0.001) * (100 ns + 100 ns) + (0.001) * (25000000 + 100 + 100) ns

= 25200 ns

Is the resulting effective access time acceptable?

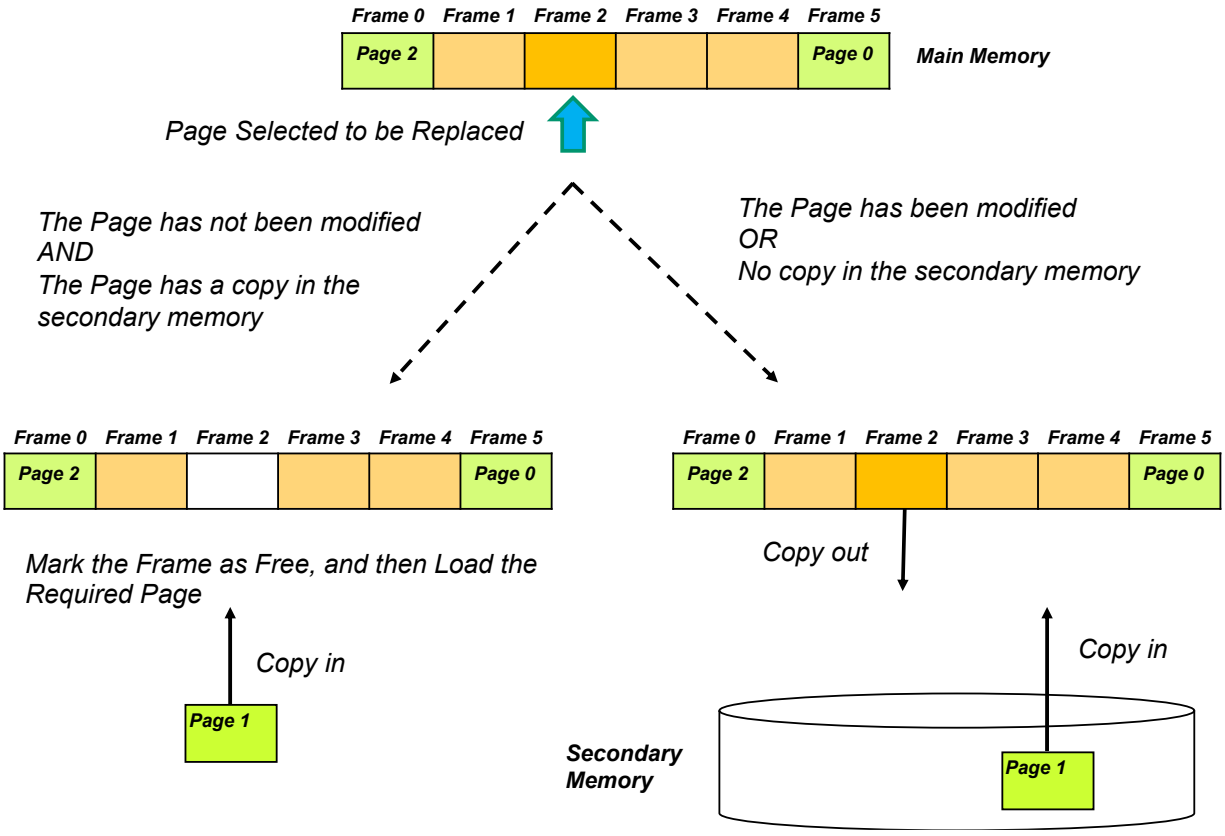
This is not acceptable. In the ideal case of no page fault, the EMAT would be 200 ns. So there is 12500% increase due to page fault handling.



8.3 Page Replacement

Page replacement refers to the situation where all frames in the main memory are occupied but a free frame is required to loading in a page. One of the existing pages in the main memory must be moved to the swap space. This happening is a consequence of over-allocation of memory.

- When a page fault occurs with a program, the kernel checks the page table and the address to confirm that this is due to a real page fault. The page is on the hard disk.
- Before an I/O request is scheduled to load the page in, and the kernel checks that no free frame is available. Page replacement is required.
- A page replacement will cause two page transfers (one in and one out). Page fault service time is doubled.
- It is possible to save one page transfer. The copying of the outgoing page may not be required if the page has not been changed. A modify bit (or a dirty bit) may be associated with each page table entry to record whether a page has been modified. The pager can examine the modify bit to decide whether the outgoing copying is required.



The steps carried out in a page-fault now involve page replacement.

1. Identify the desired page that is stored on hard disk.
2. Find a free frame from the frame table. If a free frame exists, then use it. Otherwise, use a page replacement algorithm to identify an existing frame to move out.
3. The modify bit of the selected frame is examined to see if copying out is really required. If required, I/O request is scheduled.
4. When the selected frame is made free, the desired page is loaded in.
5. The data structures (page tables, frame table) are updated, and the process is restarted.

There are many ways of selecting which page in the physical memory is to be replaced. The different strategies adopted are known as page replacement algorithms.

Some well-known algorithms are:

- First-in-first-out (FIFO)
- Optimal (OPT)
- Least-recently-used (LRU)
- Least-frequently-used (LFU)

8.3.1 Page Replacement Algorithm: First-in-first-out (FIFO)

FIFO selects the page that has been in memory for the longest time for removal based on the assumption that this page is less likely to be used.

- The main advantage of this algorithm is that it is simple to implement. A FIFO queue is set up to hold all pages in memory. When a page is brought into memory, it is inserted at the end of the queue.
- The performance of this algorithm is questionable since it basically ignores the usage pattern of the pages.
- Normally we expect that the performance improve with the amount of physical memory use. However, increasing the amount of physical memory use would worsen the performance of FIFO, which is known as Belady's anomaly.

Reference String `1 2 3 4 1 2 5 1 2 3 4 5`

The reference string describes the order of PAGEID requested by a process

FIFO Page Replacement

Physical Memory (3 Frames)

		1	2	3	4	1	2	5	1	2	3	4	5
Frame 0		1	1	1	4	4	4	5	5	5	5	5	5
Frame 1			2	2	2	1	1	1	1	1	3	3	3
Frame 2				3	3	3	2	2	2	2	2	4	4
		F	F	F	F	F	F	F			F	F	

FIFO Page Replacement

Physical Memory (4 Frames)

		1	2	3	4	1	2	5	1	2	3	4	5
Frame 0		1	1	1	1	1	1	5	5	5	5	4	4
Frame 1			2	2	2	2	2	2	1	1	1	1	5
Frame 2				3	3	3	3	3	3	2	2	2	2
Frame 3					4	4	4	4	4	4	3	3	3
		F	F	F	F			F	F	F	F	F	F

The number of page faults is 9 for 3 frames. Adding one more frame implies increasing resource for computation, but the number of page faults increased to 10 for 4 frames. It is not normal.

8.3.2 Page Replacement Algorithm: Optimal (OPT)

This algorithm theoretically has the lowest page fault rate.

- It simply replaces the page that will not be used for the longest period of time. Unfortunately OPT requires the system to know the exact timing of future paging demands, which is virtually impossible in practice.
- It is mainly used only for comparative studies.

Consider the same page access sequence (reference string) as in the previous example, evaluate how many page fault if OPT is used (i.e. if we can predict the future perfectly).

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

OPT Page Replacement

Physical Memory

		1	2	3	4	1	2	5	1	2	3	4	5
Frame 0		1	1	1	1	1	1	1	1	1	3	3	3
Frame 1			2	2	2	2	2	2	2	2	2	4	4
Frame 2				3	4	4	4	5	5	5	5	5	5
		F	F	F	F			F			F	F	

The number of page faults reduced to 7 (from 9 in the case of FIFO with 3 frames).

Consider another example reference string of 1 2 2 3 3 0 1 2 1 1 2 1. Compare FIFO and OPT with 3 frames.

FIFO Page Replacement

Physical Memory

		1	2	2	3	3	0	1	2	1	1	2	1
Frame 0		1	1	1	1	1	0	0	0	0	0	0	0
Frame 1			2	2	2	2	2	1	1	1	1	1	1
Frame 2					3	3	3	3	2	2	2	2	2
		F	F		F		F	F	F				

OPT Page Replacement

Physical Memory

		1	2	2	3	3	0	1	2	1	1	2	1
Frame 0		1	1	1	1	1	1	1	1	1	1	1	1
Frame 1			2	2	2	2	2	2	2	2	2	2	2
Frame 2					3	3	0	0	0	0	0	0	0
		F		F	F		F						

As the OPT can see the future perfectly, the page 0 request is entertained by replace the frame containing page 3. OPT knows that in the near future pages 1 and 2 are used often but page 0 is not used.

8.3.3 Page Replacement Algorithms: History Based Algorithms

There are two algorithms based on the history of page access. The aim is to guess which pages are least likely to be used in the near future, and select these pages for replacement.

- The least-recently-used (LRU) algorithm selects the page with the longest time since the last reference.
  - Chooses the page that has not been used for the longest period of time.
  - This strategy is considered to be a good one to follow and has been adopted in many systems.
  - The only disadvantage is that its implementation requires substantial hardware to keep track of the usage history of the pages.
  - Implementation methods include using a stack a time-of-use field, reference bit, additional-reference-bit, and second chance algorithm.
- The least-frequently-used (LFU) algorithm counts the number of times of reference for each page within a specific time in the past.
  - Chooses the page that has the least reference frequency.
  - A counter keeps count of the number of references that have been made to each page.

- The reasoning behind this algorithm is that we should keep the active pages in memory.

8.3.4 Example: Page Fault Rate of Different Page Replace Algorithms 1

Consider the following page access sequence (reference string), evaluate the number of page faults of OPT, LRU and LFU if there are three frames available.

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

LRU Page Replacement

Physical Memory

		1	2	3	4	1	2	5	1	2	3	4	5
Frame 0		1	1	1	4	4	4	5	5	5	3	3	3
Frame 1			2	2	2	1	1	1	1	1	1	4	4
Frame 2				3	3	3	2	2	2	2	2	2	5
		F	F	F	F	F	F	F			F	F	F

LFU Page Replacement

Physical Memory

		1	2	3	4	1	2	5	1	2	3	4	5
Frame 0		1	1	1	4	4	4	5	5	5	3	4	5
Frame 1			2	2	2	1	1	1	1	1	1	1	1
Frame 2				3	3	3	2	2	2	2	2	2	2
		F	F	F	F	F	F	F			F	F	F

All three pages have been accessed one time. Any one of them may be replaced

This sequence of page access does not demonstrate much locality of reference. Therefore LRU and LFU do not perform better than FIFO.

8.3.5 Example: Page Fault Rate of Different Page Replace Algorithms 2

Consider the following page access sequence (reference string), evaluate the number of page faults of LRU and LFU if there are three frames available. This reference string has more repeated page access (due to locality of references).

1 1 2 0 0 4 2 0 1 0 2 1

LRU Page Replacement

Physical Memory

		1	1	2	0	0	4	2	0	1	0	2	1
Frame 0		1	1	1	1	1	4	4	4	1	1	1	1
Frame 1				2	2	2	2	2	2	2	2	2	2
Frame 2					0	0	0	0	0	0	0	0	0
		F		F	F		F			F			

LFU Page Replacement

Physical Memory

		1	1	2	0	0	4	2	0	1	0	2	1
Frame 0		1	1	1	1	1	1	1	1	1	1	1	1
Frame 1				2	2	2	4	2	2	2	2	2	2
Frame 2					0	0	0	0	0	0	0	0	0
		F		F	F		F	F					

FIFO Page Replacement

Physical Memory

		1	1	2	0	0	4	2	0	1	0	2	1
Frame 0		1	1	1	1	1	4	4	4	4	4	4	4
Frame 1				2	2	2	2	2	2	1	1	1	1
Frame 2					0	0	0	0	0	0	0	2	2
		F		F	F		F			F		F	

Both LRU and LFU perform slightly better than FIFO.

8.3.6 Page Replacement Algorithms: Stack-based Implementation in LRU algorithm

The algorithms based on least-recently-used are often the better performers. They do not suffer from Belady's anomaly, and they belong to a class of algorithms known as stack algorithms.

Stack algorithms are algorithms that the set of pages in memory for N frames is always a subset (included the same set) of the page set that would be in memory with N+1 frames.

- Page identifiers are put onto a stack, and recently referred page is moved to the top of the stack. The bottom of the stack would be the LRU page.
- LRU requires more hardware support than FIFO, and not all architecture supports a true LRU algorithm.

Approximated algorithms are developed which are based on the following.

- Reference Bit. A reference bit is associated with each page. It is initially set to zero. When a page is referenced, it is set to one. A page with a set reference bit means that it has been referred, though the exact time is not known.
- Additional Reference Bit. More bits can be assigned so that the history of access can be recorded. For example, we can keep 8-bit for each page, and at regular intervals, the reference bit is shifted to the more significant bit direction.
- Second-Chance Algorithm.
  - Second-chance algorithm is based on FIFO, but when a page is selected further processing is required.
  - The selected page reference bit is inspected, and if the value is zero, then the page is replaced.
  - If the value is one, then the page is given a second chance and the algorithm moves on to select the next FIFO after reset the reference bit.
- Enhanced Second-Chance Algorithm.

- In enhanced second-chance algorithm, the modify bit is considered together with the reference bit.
- There are four difference possibilities with the two bits.
- All pages are put into one of the four classes.
- The page replaced is the first page encountered in the lowest non-empty classes.

Reference Bit	***Modify Bit***	***Notes***
0	0	Best page to replace
0	1	Modified page, the page needed to be copied out before replacement
1	0	Recently used and likely to be used again
1	1	Recently used and modified.

8.4 Performance of Virtual Memory Systems

This section discusses two issues concerning the performance of virtual memory systems.

- Thrashing
- Locality and working set

8.4.1 Thrashing

It is an undesirable situation in which the paging activity is high, and high enough to cause CPU utilization to approach zero.

- System designers have found that if the number of page frames allocated to a process falls below a certain threshold, page faults will quickly increase.
- When only a few frames are allocated to a process with many pages, then even if a page is in active use, it would still be selected for replacement.
- Unfortunately, these active pages will probably be needed again very soon, which leads to more page faults in order to bring these pages back.
- To make matters worse, as the faulting processes queue up for the paging disk, CPU utilization decreases. The CPU scheduler will try to increase the degree of multiprogramming to compensate for the low CPU utilization by starting more new processes. As each process requires some new pages to start execution, this leads to an even greater contention for physical memory and causes more page faults.
- The end result is that the system spends more time on paging than doing useful work, so system throughput drops. This is called thrashing.

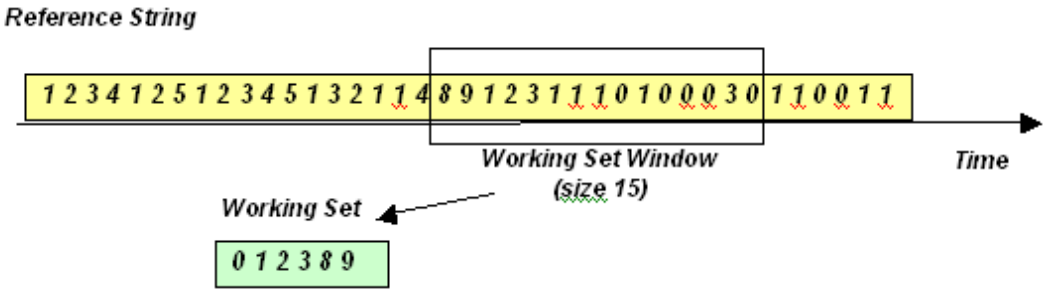
Another cause of thrashing is through increasing the degree of multiprogramming, up to a stage where there are too many active processes (and therefore active pages). Each new process is started with getting frames from active processes and causing more page faults.

8.4.2 Locality and Working Set

A simple method to prevent thrashing is to provide enough frames for a process. The challenge is of course to work out how many frames are needed.

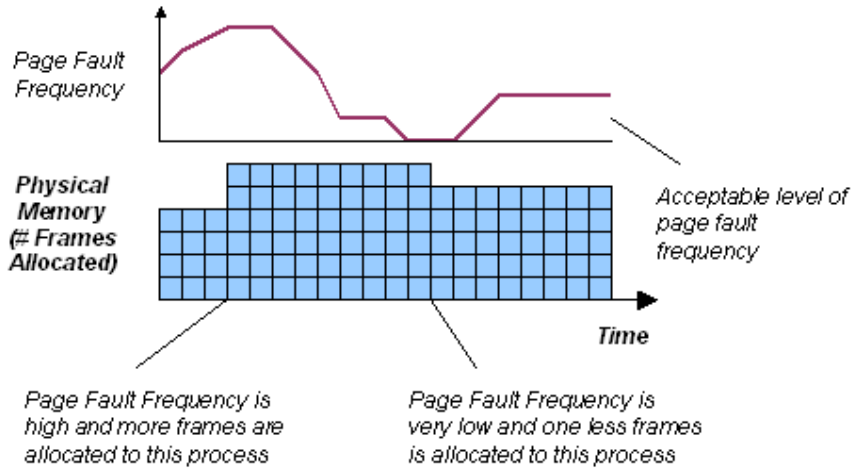
- The locality model of execution refers to the observation that when a process executes, it tends to access a small set of pages at one time. After a short while it would then move to another small set of pages. This actively access set of pages is known as a locality.
- A working set window is the total number of page references within a period of time by a process. The working set is then simply defined as the set of page references contained in the most recent working set window.
- Suppose a process has the following page references during a particular period of time. With a working set window is 10, the working set is (1, 6, 5, 3)

  11165533317621223212 35
- Usually the working set window is in the range of 10000.
- From the size of the working set of each process, we can calculate the total demand for frames. If the total demand is more than the number of available frames, then thrashing will occur.



An alternative method to prevent thrashing is to dynamically adjust frame allocation based on the current page-fault frequency.

- This is a more direct method by adjusting the number of frames allocated to a process based on the page-fault frequency.
- This technique begins with setting a standard desirable page-fault frequency.
- A process with a high page-fault frequency detected would be allocated more frames because it needs more frames.
- A process with a low page-fault frequency detected would have a frame removed.


Page Fault Frequency
Physical Memory (# Frames Allocated)
Acceptable level of page fault frequency
Time
Page Fault Frequency is high and more frames are allocated to this process
Page Fault Frequency is very low and one less frames is allocated to this process

COMPS267F Operating Systems

File Systems 9

Textbook Reading

Silberschatz, Chapter 11, Section 11.1 to 11.3; Section 11.6; Section 12.4; Section 12.6

This chapter describes how files may be organized into an effective IO system.

9.1 Overview of Computer Files

A computer file is a persistent entity containing a block of data.

- A computer file has a unique name and a set of attributes
- The block of data represented by a computer file is stored in a secondary memory system.
- Computer programs interact with the secondary memory system through reading and writing computer files.

The block of data in a computer file is in the form of a sequence. There is an order in the data and usually operations are carried out from the beginning to the end.

The data stored in a computer file can contain any type of information, such as a computer program or user data.

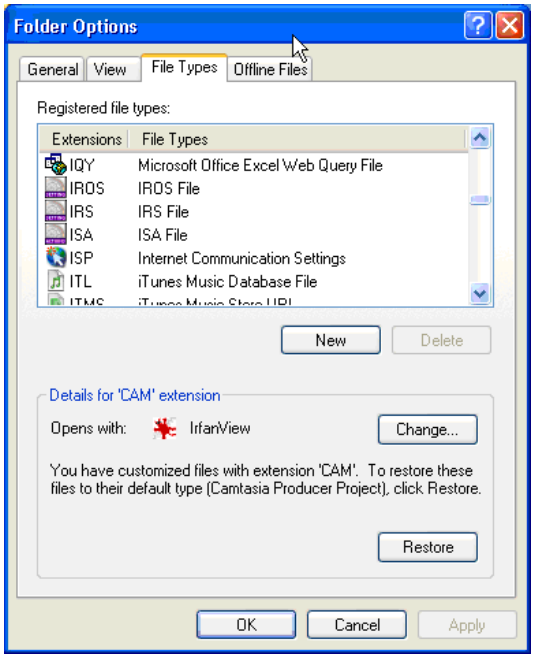
Attributes of a file indicate some information about the file to the operating systems.

- Examples of file attributes include the following: name, identifier (internal record), type, location, size, protection, time stamp, user identification.
- The set of file attributes can be different from one operating system to another.
- A program is distinguished with the execute bit in the file attribute in UNIX and with the file extension in Windows.
- Directories, network connection and device drivers are often regarded as a file. In such case, special attributes must exist to distinguish the type.

The file itself does not necessarily carry information about its purpose.

- The OS usually leaves it to the user to manage the purposes of the files and to apply appropriate operations to the files.
- The Windows operating systems, which emphasis on user friendliness, has tried to provide support to file purpose interpretation. Windows uses a registration scheme for file suffix that the operating systems know the relevant applications for a particular file suffix.
- In cases such as MIME based mail or HTTP web browsing, the meaning of a file is carried with the file externally as a MIME content type. This is the reason why a web browser generally knows how to handle the file it receives.

The following figure shows the file association in Windows OS.



9.1.1 Operations on Computer Files

The OS provides a number of file operations for programs to access the secondary storage.

- Creating a new file (allocating space, associate the space with the file).
- Writing a file.
- Reading a file.
- Repositioning within a file (or traversing in a file).
- Deleting a file (free the allocated space).
- Truncating a file (removing part of a file).

Most operating systems require an explicit open operation on a file before the above operations can take place.

- The advantage is improved efficiency through building up information about the file for subsequent operations. Many file operations requires information about the location of the data and attributes of the file. An open operation collects the information and store in an open-file table. Subsequent file operations can refer to the information to speed up the operation.
- Some OS open file implicitly when a file is used for the first time.
- Each process has its own open-file table, which points to entries in the system-wide open-file table.
- Synchronization rules apply to the reading and writing of files.

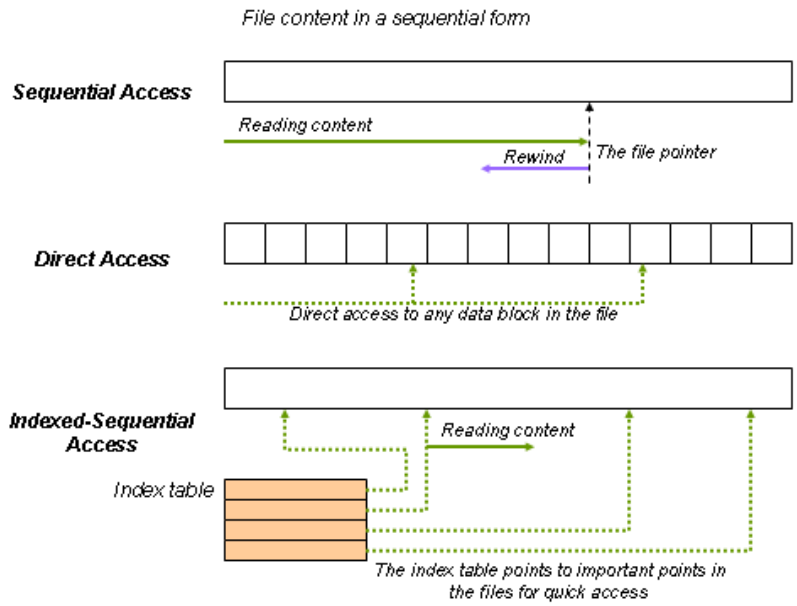
An open file generally has the following attributes.

- File pointer to keep track of the current read/write head (current read/write location).
- File open count to keep track of number of process that has opened a particular file.
- Disk location of the file.
- Access rights.

9.1.2 File Access Methods

The key file operation is concerned about reading and writing file content access. The most common methods for accessing the content of a file are in the following.

- Sequential.
  - The file is processed one piece of data after another from the beginning to the end. A rewind operation is usually available to move back to the beginning.
- Direct.
  - Arbitrary data block can be access directly. Each data block is identified with a block number.
- Indexed-sequential.
  - It is essentially sequential access with bookmarks at key places where the file pointer can be relocated.
  - This is useful for sequential access to a very long file.
  - The index itself can become too large, a master index would be built that kept in memory and secondary level indexes can be stored on hard disk.



9.1.3 Directory Systems

The OS provides directory as a means to manage named files.

- Directory allows files of the same name co-exist in a system if they are placed in a different directory. It is essential for multi-users system.
- Directory allows files to be grouped and organized logically.
- The directory itself can be viewed as a file that has entries listing all of the member files. Each directory entry for a file can contain information such as file name, file type, location, and size.

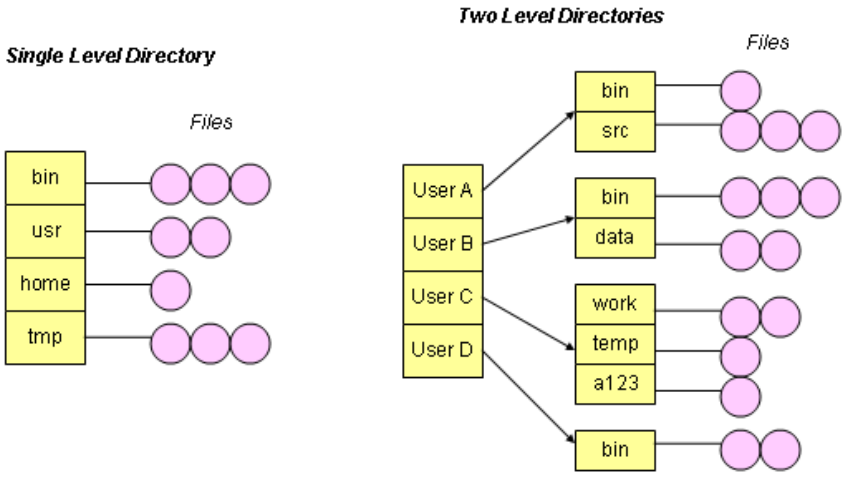
The objectives of directory system include the following.

- Efficient location and access to file.

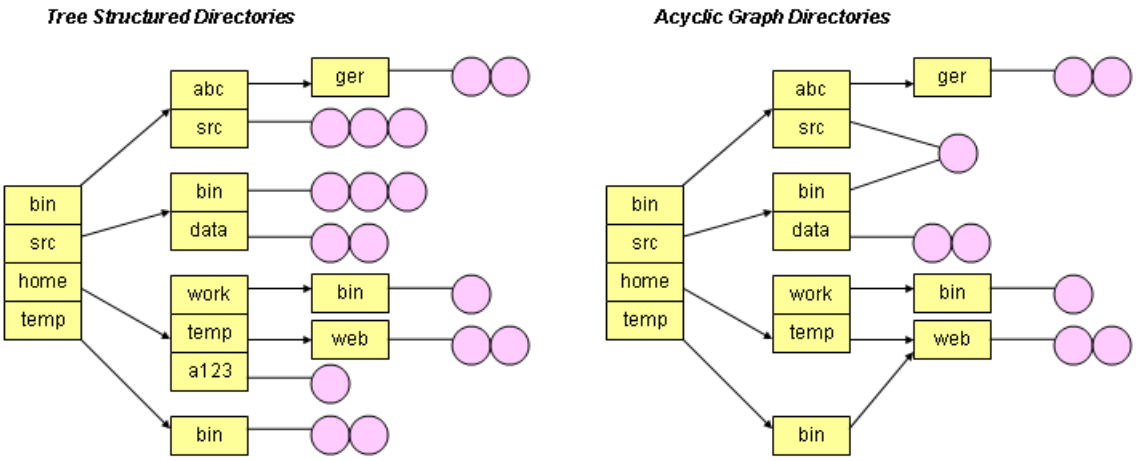
- File naming is convenient to users.
- Logical grouping of files.

Directory systems can be implemented in a number of ways.

1. Single level directory.
    - All the users share the same directory structure. It has the file-naming problem because users are sharing the same file space.
2. Two level directories.
    - Each user has own user file directory.
    - When a user logs on, the operating system searches the master file directory that is indexed by username or account number.
    - A problem is that users are isolated from each other and they cannot share files. This can be a major disadvantage when users cooperate on a task that requires accessing other user files.



3. Tree structured directories.
    - The two-level directory can be extended to a tree of multi-levels. Users can create their own sub-directories.
    - Directories in both UNIX and MSDOS are tree-structured.
    - The tree has a root directory and each file in the system has a unique path name that is the path from the root, through the sub-directories, to the specified file.
    - To uniquely specify a file, we have to use a path name such as root/progs/progB.
4. Acyclic-graph directories.
    - An acyclic graph directory, which can be considered an extension of the tree-structured directory system, allows directories to share sub-directories and files.
    - Implementing such a directory can be quite tricky.
    - BSD UNIX uses a directory entry called a link. A link is actually a pointer to another file or sub-directory.



9.1.4 File Protection

A file protection strategy is needed because there are many users working with a file system at a time. There are two aspects of file protection.

1. Protection from physical damage.
    - Accomplished by making duplicate copies of files.
2. Protection from improper or illegal access.
    - Implementing limiting different types of file access.
        - Some of the more common operations on a file are read, write, execute, append, delete, and list attribute.
        - A better implementation allows user level privilege differentiation.

The following discusses access protection schemes.

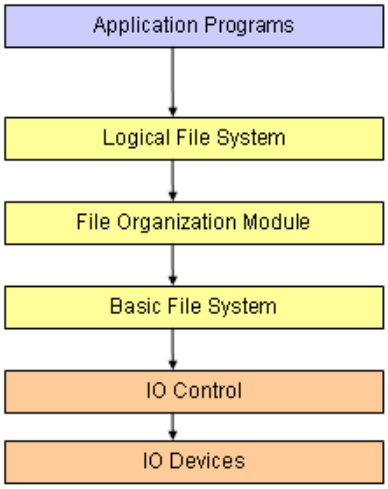
1. Passwords.
    - Passwords can protect access to a user account in a computer system to authorized users.
    - While a password can theoretically be assigned to each file there are significant disadvantages, however, to such a scheme the number of passwords can become very large and protection is on a yes/no basis rather than at a detailed level.
2. Access Lists.
    - An access list is associated with each file and directory.
        - The list specifies the username and the type of access allowed for each user.
    - Constructing such lists for all the files and sub-directories can be a tedious task.
3. Access Groups.
    - Access groups are a condensed version of the access.
        - Users are classified as the following: the owner (the creator of the file), the group (a set of users who share a file), and the world (all other users in the system).

```
drwxr-xr-x   2 comps260f   comps260f  8192 May 31  2009 guest
-rw-r--r--   1 comps260f   comps260f   460 Oct 17  2010 index.htm
-rw-------   1 comps260f   comps260f   460 Oct 16  2010 index.htm.local
-rw-------   1 comps260f   comps260f   488 Oct 16  2010 index.htm.plbpc005
drwxr-xr-x   6 comps260f   comps260f  8192 Nov 16  2010 mt258
drwxr-xr-x   7 comps260f   comps260f  8192 Sep 22 11:41 mt258-2004
drwxr-xr-x   9 comps260f   comps260f  8192 Sep 22 11:41 mt258-2005
```

9.2 File Systems

The file system is the module in an OS that supports the above file operations.

- The file system works with the secondary storage and to use the features such as in-place rewritten of data, direct access to any data block on hard disk, and even data redundancy.
- A file system generally consists of the following layers.
- Logical File System manages metadata information, which means anything but the file content itself. It also manages the directory structure. File structure is managed using file control blocks.
- File Organization Module knows about files, and their logical and physical structure. It is responsible to translate logical location (0, 1, 2, ...) into physical location.
- Basic File System needs to distinguish appropriate device driver for a particular instruction. Each physical block across multiple devices is uniquely identified by a numeric disk address (drive, cylinder, track, and sector).
- Device Drivers are responsible for transferring information between the main memory and the disk system. It translates high-level instructions such as retrieve block A into lower level instruction that involves hard disk controls.



9.2.1 Management of File Space

The file space is the place where the file content and metadata are actually stored. The file space is usually built on the hard disks, but it can also be based on tape drives, remote servers, and even the main memory (called the RAM Disk).

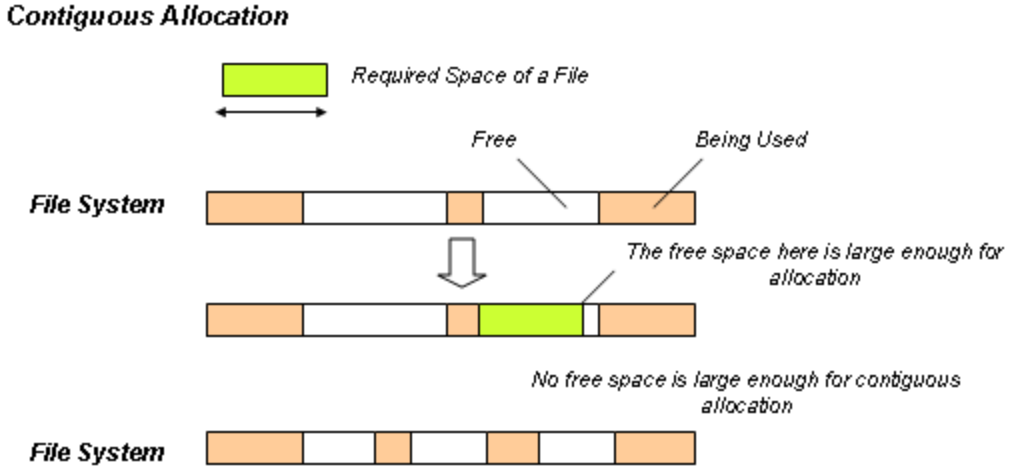
The capacity of storage devices such as hard disks has grown rapidly due to technological advances, but the average size of files has grown even more dramatically, particularly as multimedia applications come into the mainstream of computing.

File allocation schemes should satisfy the objective that efficient file access is supported while minimizing overhead.

The following discusses some common approaches used in allocating disk space to files.

Contiguous Allocation

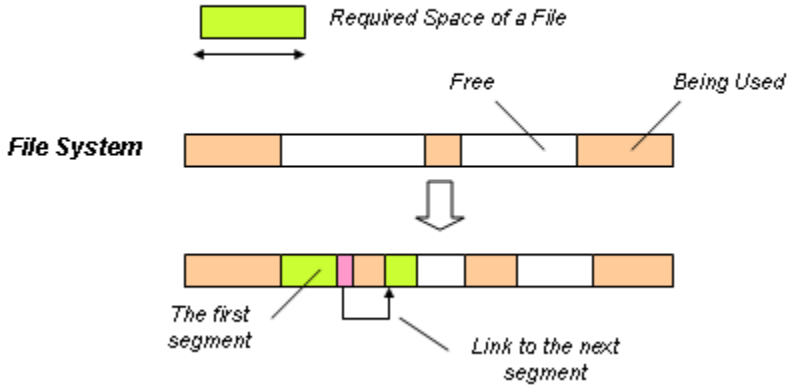
- The contiguous allocation method requires each file to be a set of contiguous blocks on the disk.
- The advantage of this method is that the number of disk seeks for accessing contiguous allocated files is minimal (only the search for the first block is required).
- A major drawback of this method is finding space for new files due to severe fragmentation.
- Another problem is determining how much space is to be allocated to a file. If some kind of best-fit strategy is used then, when the file is to be extended, the space around the file might well have been allocated to other files.



Linked Allocation

- A file consists of a linked list of disk blocks and the disk blocks may be located anywhere on the disk. Since any free space on the disk can be used, external fragmentation is no longer an issue.
- File extension is also handled easily. An additional free disk block is simply allocated and linked to the list. The major drawback of linked allocation is that it can only be used for sequential-access files by following the pointers.
- Pointers also take space.
- Linked allocation also has a potential reliability problem. When a single pointer was corrupted, many blocks would be lost.

Linked Allocation

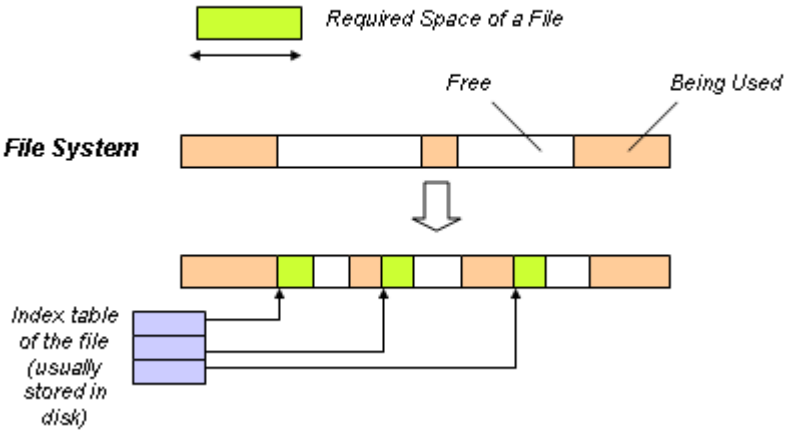


Indexed Allocation

Indexed allocation corrects the problem of linked allocation by collecting all the links together in an index block.

- Each file has its own index block, which is a list of disk block addresses. The directory contains the address of the index block. Now there is no need to traverse the linked disk blocks sequentially.
- An index block always needs to be allocated, even if the file is a small one and only a few pointers are actually needed in the index block. This may seem to be a minor problem until you realize that most of the files in a typical file system are small.
- A solution is to build multiple levels of index blocks and access to records through indirection.
- MS-DOS is an interesting variant of linked and indexed allocation.
    - A file-allocation table (FAT) is used for each file.
    - The FAT contains an entry for each disk block in the system and is indexed by the block number.
    - Similar linked allocation in that the entries in the FAT are linked together in a list but note that it can also be viewed as indexed allocation because the FAT can be considered an index block.

Indexed Allocation



9.3 Efficiency and Performance

The success of any file system from the user perspective has to be measured against how efficiently it makes use of the available disk space and the effect on the performance of the given memory allocation scheme and adopted directory structure.

There are two key issues.

- Minimizing the overhead such as pointer size and fixed structures (e.g. UNIX file nodes).
- Minimizing disk accesses because in modern computer systems where disks tend to be the bottleneck in system performance.

There are some general solutions.

- A very common technique is to include a local cache in the disk controllers that is large enough to store an entire track at a time. Once a seek is made, a whole track is read into the cache thus almost completely eliminating latency time for subsequent sector requests.
- Caching can also be maintained in memory, which is known as a disk cache.
- UNIX treats all unused memory as a buffer pool to be used for paging and disk caching.
- RAM disk is a section of memory is mapped to a virtual disk, and all disk operations will actually take place in memory. This needs user intervention to set up.
- Cache replacement strategies must be carefully designed.
- Techniques for sequential access include free-behind and read-ahead. Free-behind simply means that a block is removed in the buffer as soon as the next buffer is requested, so as to free up buffer space. Read-ahead will retrieve and cache several subsequent blocks when a block is read.

COMPS267F Operating Systems

IO Structure and Disk Management 10

Textbook Reading

Silberschatz, Chapter 13, Section 13.3 to 13.4; Chapter 14, Section 14.1 to 14.4

This chapter is to describe how the OS manages the large variety of IO devices effectively. This chapter also discusses in more details the management of hard disks.

10.1 Overview of IO Management

There are a large variety of IO devices that may be connected to the computer systems. These IO devices can be differentiated in the following aspects.

1. Unit of data transfer.
   - Character stream transfer moves one character after another between the device and computer system. The keyboard is an example.
   - Block transfer moves a block of data in one operation. The size of the block depends on the actual IO device. The hard disk is an example of block transfer.
2. Interaction of programs and IO operations
   - Synchronous data transfer requires the calling program to monitor the progress of IO operations.
   - Asynchronous data transfer can happen independently while the calling program working on other tasks.
3. Nature of data access
   - Sequential access restricts the reading and writing operations to occur in a sequential manner (from the beginning to the end).
   - Random access allows the reading and writing operation to take place at an arbitrary location in the IO device.
4. IO device mutual exclusion
   - Sharable devices allow multiple processes to access them at the same time. The display monitor is an example.
   - Dedicated devices allow only one process to control it at a time. The printer is an example.
5. Speed of IO device
   - IO devices differ in the initial setup time and data transfer rate.
6. Read and write allowances
   - Read only devices supports only read operations.

The aim of I/O management is to hide the great disparity of hardware configurations. A uniform interface is presented to programs and users. It carries out the necessary mapping of instructions and data structure to the underlying hardware.

10.1.1 Concept of Device Independence

Device independence refers to the provision of a uniform method or interface exists for the manipulation of very different underlying IO devices.

- Device independence is not really a novel concept. A program can perform file read operations to the files stored on CDROM, hard disks, and network drives.
- Plug-and-play is an up-to-date example of this concept. As far as user is concerned, a new hardware could be simply added to a computer without the need to re-configure. The operating systems are designed to carry out the necessary management and mapping.
- Another important condition for device independence is uniform naming. In UNIX, a particular pathname may refer to a actual data file or an IO device.

10.1.2 Plug-and-Play

The concept of plug-and-play is often associated with Microsoft range of operating systems (from Windows 95 onwards). However, the concepts were actually implemented by a number of other operating systems before 1990.

- Connecting a piece of hardware to a computing can be hazardous
  - Linking two electrical circuitries together could cause problem to each other.
  - Physically and electrically safe is the first condition of a plug-and-play system.
- Hot-swapping plug-and-play allows plugging and unplugging of new hardware without risk of damage.
- Self-configuration is the ability to configure a piece of hardware without user intervention or any need for software configuration programs.
- Each piece of hardware must provide an ID so that it can be recognized.
- The operating systems must be designed to handle the detection and handling of new hardware.
  - When a new hardware is plugged in, an interrupt would happen that alert the operating systems.
  - OS reads from the bus information about the new hardware.
  - It can then determine the appropriate device driver and load it in.
- Modern operating systems have the advantage of network connectivity, that device driver can be downloaded from the Internet.
- Older operating systems would need to spend longer time to scan the whole system for changes in hardware.

10.1.3 Example: Universal Serial Bus (USB)

The universal serial bus allows the connection of a multitude of USB devices (up to 127 devices) through branching. The branching of a USB connection happens in a USB hub.

- USB is hot-swappable. New device can be plug-in without the need to restart the computer. When a new USB device is plugged-in, the OS checks it and loads the appropriate device driver.
- USB it supports multiple transfer speeds, including 1.5Mbs for human interface devices, 12Mbs for full speed transfer, and 480Mbs for high-speed transfer in USB 2.0. USB 3.2 supports up to 20Gbs.

A number of device classes are defined for USB devices. An OS should implement all the device classes.

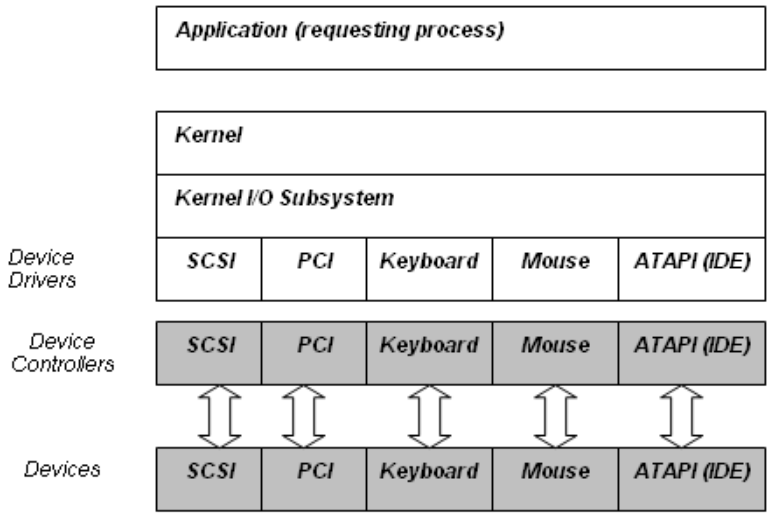
Some USB device classes are given in the following table.

Device ID	***Device Class***
0	Reserved
1	Audio devices such as sound card
3	Human interface such as keyboard, mouse, etc
6	Camera uses Picture Transfer Protocol
7	Printer devices
8	Mass storage device. Device is presented as a file system
9	USB Hubs
10	Communications devices such as wireless network interface
14	Video devices, webcam
224	Wireless controllers such as Bluetooth
255	Custom

10.2 I/O System Structure

The I/O system structure is a layered structure. The objective is to offer the application (user) a uniform I/O interface to different devices.

The following figure illustrates this layered structure.



The layered structure aims to tackle the diversity of device characteristics

- The lower layers deal with all the complexities of real devices.
  - Carefully hidden from the user.
- A device driver is a software module that provides a device-independent interface for the kernel I/O subsystem (not the user process).
- There is a device driver for every type of device
  - It would be very difficult to write the kernel I/O subsystem unless the devices could be characterized using some standard interface.

10.2.1 Kernel I/O Subsystem

I/O kernel subsystem is a system manager that translates user requests into low-level I/O requests and allocates system resources (like buffers) to these requests.

- I/O kernel subsystem sends requests to the appropriate device driver.
- The device driver maps these generic I/O requests into device-specific instructions for the device controllers.
- The hardware executes the commands, and the interrupt handler handles the returned status.

Kernel mainly provides four services related to the I/O.

1. I/O Scheduling

- The aim is to determine a good order to execute a set of I/O requests.
  - The hard disk IO scheduling is a good example of the importance of scheduling.
- Scheduling aims to improve the overall system performance.
  - A queue of requests is kept for each device.
  - The scheduler to examine the requests and make the necessary rearrangement.

2. Buffering

- A buffer is a piece of memory that can store data when two devices are transferring data.
- The objectives of buffer include

    - Rectifying speed mismatch between devices
    - Rectifying data size mismatch.
    - Supports copy semantics that ensures the data sent to the I/O device is the copy finally written.
- Spooling is a special type of buffer that disallows interleaving of data streams.

3. Caching
    - Caching is the storage of commonly used information in memory such as metadata.

4. Error Handling
    - Devices and I/O transfer can fail in many different ways, and the kernel usually returns the status of I/O transfer call.

10.3 Disk Management

Hard disks provide the secondary storage for modern computers. Managing a disk subsystem is important so that it can provide good file system performance.

The following discussion refers to mechanical hard disks, which are still very common despite the increasingly popular solid-state disks.

The disk space in modern disk drives is addressed as 1D array of blocks, and a logical block is the smallest unit of transfer. Previous disks are addressed as a set of cylinder, track, and section number.

10.3.1 Disk Scheduling

Disk access delay can be divided into several components

- Seek time. The time taken from the rest state to the ready state in which the disk has started rotating steadily and the read-write arm positioning correctly. The largest delay is usually the seek time.
- Rotational latency. The time taken for the read-write head to position over the correct sector through rotating the disk.
- Data transfer time.

Disk scheduling is the selection of a request from the pending disk I/O requests. The requests that minimizes the overall seek time is favored.

The following sections discuss a number of commonly used disk scheduling algorithms.

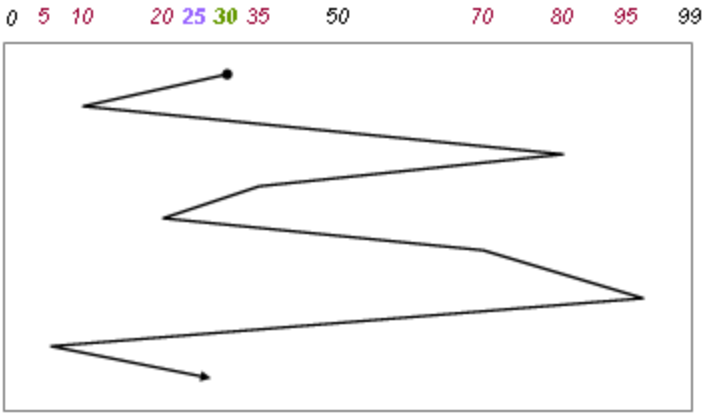
First-Come First-Served Scheduling (FCFS)

FCFS schedules the disk requests based on their order of arrival.

A potential drawback is that it may result in lots of disk head movement. For example, if one request is for a track near the centre and the next request is for a track near the edge of the disk, then seek time will be very high.

First-Come-First-Served (FCFS)

Pending Requests:	10 80 35 20 70 95 5 25
Disk Head Position:	30



Tracks Travelled = (30 – 10) + (80 – 10) + (80 – 35) + (35 – 20)
+ (70 – 20) + (95 – 70) + (95 – 5) + (25 – 5)

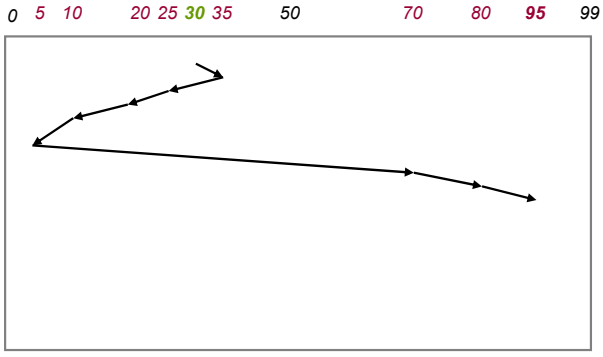
Shortest-Seek-Time-First (SSTF) scheduling

SSTF attempts to service all requests close to the current head position first.

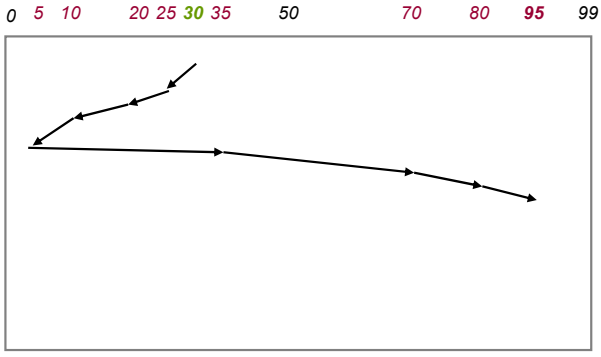
More pending requests can be satisfied with a short travelling distance before moving the disk head far away to service other requests. The disadvantage of SSTF scheduling is that it may result in starvation for some requests that are far away from the current position.

Shortest-Seek-Time-First (SSTF)

Pending Requests:	10 80 35 20 70 95 5 25
Disk Head Position:	30



Tracks Travelled = (35 – 30) + (35 – 5) + (95 – 5)



Tracks Travelled = (30 - 5) + (95 – 5)

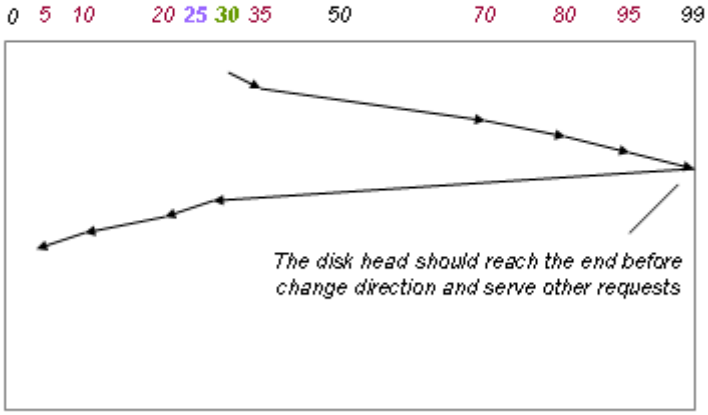
- Both are the results of SSTF.
- At the initial position of track 30, travelling to 25 and travelling to 35 takes equal distance of 5.
- There are two possible paths.

SCAN Scheduling

SCAN scheduling is also called the elevator algorithm.

SCAN

Pending Requests:	10 80 35 20 70 95 5 25
Disk Head Position:	30 (Travelling from Low to High)



Tracks Travelled = (99 – 30) + (99 – 5)

Operation of SCAN:

The disk head sweeps from one end of the disk to the other, and it services all requests on the way.

Then it handles any requests on its way back.

Limitation of SCAN:

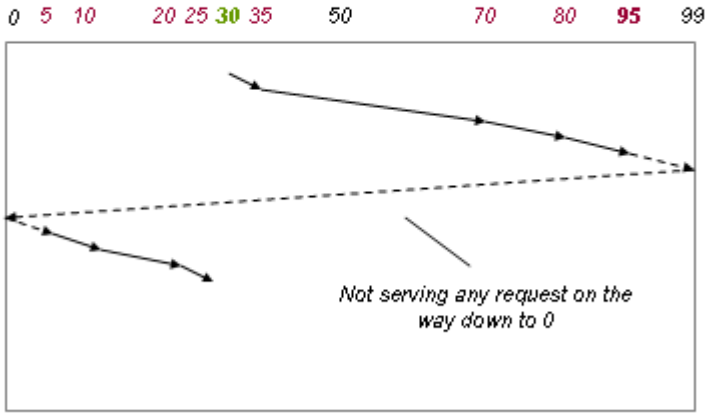
- Requests that have just missed the disk head will have to wait a long time.
- It is particularly true for those requests near one side of the disk.

C-SCAN scheduling

C-SCAN is SCAN scheduling with a minor variation.

C-SCAN (Service from Low to High)

Pending Requests:	10 80 35 20 70 95 5 25
Disk Head Position:	30 (Travelling from Low to High)



Tracks Travelled = (99 – 30) + (99 – 0) + (25 – 0)

Operation of C-SCAN:

The disk head sweeps from one end of the disk to the others and it services all requests on the way.

Then it goes back to the starting point and start again.

There is a service direction.

It's like an elevator that services from ground floor to the top floor, and then heads directly down to ground floor without taking any more passengers on its way down.

This in general will result in a more uniform waiting time.

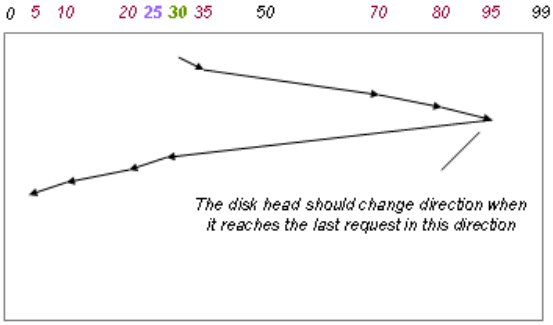
LOOK scheduling/C-LOOK scheduling

This is another minor variation of SCAN/C-SCAN scheduling. Instead of going from one end of the disk to the other end, the disk head will stop at the track of the last request in each direction and then change direction.

C-LOOK is LOOK scheduling that serve only in one direction.

LOOK

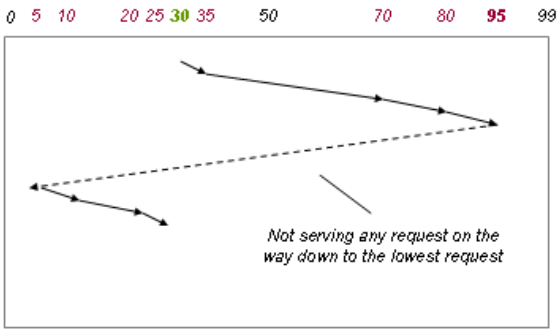
Pending Requests:	10 80 35 20 70 95 5 25
Disk Head Position:	30 (Travelling from Low to High)



Tracks Travelled = (95 – 30) + (95 – 5)

C-LOOK (Service from Low to High)

Pending Requests:	10 80 35 20 70 95 5 25
Disk Head Position:	30 (Travelling from Low to High)



Tracks Travelled = (95 – 30) + (95 – 5) + (25 – 5)

Exercise 1: Performance of Disk Scheduling Algorithms

A disk has track number 0 to 199. Given a disk queue with the following request for an I/O block in the order: 98, 183, 37, 122, 14, 124, 65, 68, and the disk head is currently at cylinder 53 and moving towards 0. The read-write arm travels at 5 ms per track. Evaluate the traversal of the disk head and the seek time of each of the algorithm FCFS.

Answer:

Tracks Travelled = (98 – 53) + (183 – 98) + (183 – 37) + (122 – 37) + (122 – 14)
+ (124 – 14) + (124 – 65) + (68 – 65)

Time = [(98 – 53) + (183 – 98) + (183 – 37) + (122 – 37) + (122 – 14) + (124 – 14)
+ (124 – 65) + (68 – 65)] x 5ms

Exercise 2: Performance of Disk Scheduling Algorithms

Evaluate the traversal of the disk head and the seek time for the pending requests in the previous exercise if the scheduling algorithm of SSTF

Answer:

Visited sequence = 53 > 65 > 68 > 98 > 122 > 124 > 183 > 37 > 14

Tracks Travelled = (183 – 53) + (183 – 14)

Exercise 3: Performance of Disk Scheduling Algorithms

Evaluate the traversal of the disk head and the seek time for the pending requests in the previous exercise if the scheduling algorithm of SCAN and C-SCAN are used. Assume that the C-SCAN algorithm serves when travelling in the downward direction (towards 0)

Answer:

SCAN Visited sequence = 53 > 37 > 14 > 0 > 65 > 68 > 98 > 122 > 124 > 183

Tracks Travelled = (53 – 0) + (183 - 0)

C-SCAN Visited sequence = 53 > 37 > 14 > 0 > 199 > 183 > 124 > 122 > 98 > 68 > 65

Tracks Travelled = (53 – 0) + (199 – 0) + (199 – 65)

Exercise 4: Performance of Disk Scheduling Algorithms

Evaluate the traversal of the disk head and the seek time for the pending requests in the previous exercise if the scheduling algorithm of LOOK and C-LOOK are used. Assume that the C- LOOK algorithm serves when travelling in the downward direction (towards 0)

Answer:

LOOK Visited sequence = 53 > 37 > 14 > 65 > 68 > 98 > 122 > 124 > 183

Tracks Travelled = (53 – 14) + (183 - 14)

C-LOOK Visited sequence = 53 > 37 > 14 > 183 > 124 > 122 > 98 > 68 > 65

Tracks Travelled = (53 – 14) + (183 - 14) + (183 – 65)

10.3.2 Swap Space Management

Virtual memory systems use the disk as an extension of main memory to support large logical address space. Frames that cannot be fitted into main memory are moved out to swap space on disks.

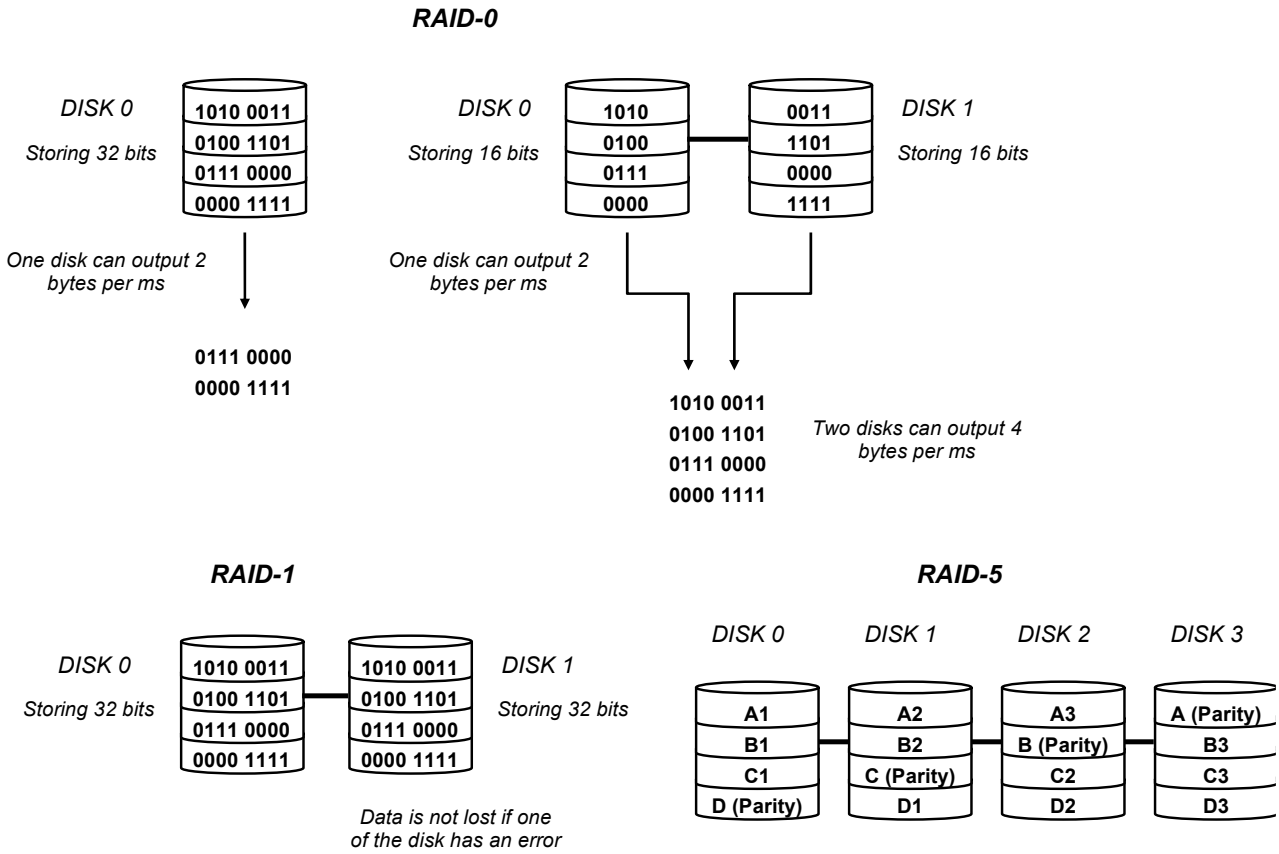
- A simple approach is to treat it as a big file within the file system.
  - The performance is ordinary.
- Another approach is to locate the swap-space in a separate partition and use special algorithms to allocate and de-allocate space.
  - This is commonly adopted in commercial operating systems,
  - This approach is used because page swapping must be done very fast since the disk is used to emulate main memory.
  - The drawback of this approach is that this is a static allocation that is typically done when the system is first configured. If the swap space partition needs to be enlarged later, then the system will typically have to be reconfigured.

10.3.3 RAID Structure

Redundant Arrays of Independent Disks (RAID) is a technology of applying redundancy to achieve a higher reliability, and also applying parallelism to achieve a higher performance. This is applied by connecting a large number of disks to a computer system in several specific manners.

The simplest method to have greater reliability through redundancy is to mirror or duplicate every disk. This is very expansive in terms of time and hardware cost.

- Data striping is the approach to split the bits of each byte between several disks.
  - The transfer rate is improved because a read can be carried out simultaneously on the disks.
  - A recombination process will recover the original bytes.
- RAID levels are specified ways to connect multiple hard disks together. Each has its own characteristics in redundancy and performance. Common RAID levels include RAID 0, RAID 1, and RAID 5.
- RAID level 0 applies striping at block level and no redundancy.
- RAID level 1 mirrors the content of one disk on another disk.
- RAID level 5 is called block-interleaved distributed parity. It calculates the parity of data bytes and distributes the parity across all disks (except the disk where the data locates). The parity bit allows error correction.



COMPS267F Operating Systems

Distributed Systems 11

This chapter discusses distributed systems, a configuration for running applications that involves multiple computers connected to a network. Distributed systems are almost everywhere now because computer networks and mobile devices are ubiquitous.

- A distributed system is a multi-computer system in which the computers are networked and work together to provide a service to its user.
- A huge number of different computers located in multiple sites to a few similar computers in a single location.
- Despite the variety in scales and forms, distributed systems exhibit a number of common characteristics which are explained in this chapter.

11.1 Introduction to Distributed Systems

A distributed system is a collection of independent computers that are linked by a network and provide coherent services to its users. A distributed system makes use of software such as operating systems to produce the image of an integrated computing facility.

Differences between a computer network and a distributed system:

- A computer network is simply a collection of independent computers that are interconnected.
- Distributed systems are built on top of computer networks
  - The network is just one of the resources of a distributed system.

The existence of multiple autonomous computers in a distributed system is transparent to its users.

- It appears as a single coherent system.
- Users do not need to know exactly how and where the remote services are located in the system.

11.1.1 Characteristics of Distributed Systems

A distributed system aims to make its users think that the collection of independent computers is simply a single coherent system.

The following are the related features:

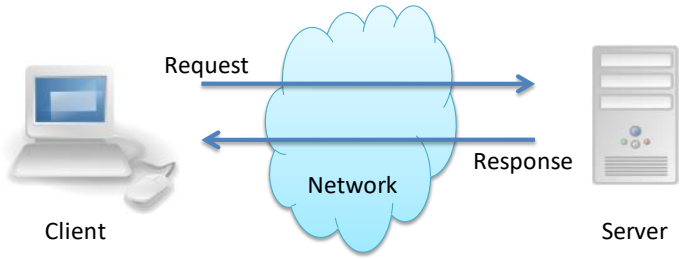
Features	***Remarks***
Heterogeneity	Computers of the distributed system may vary with respect to computer hardware (e.g. CPU type, memory capacity, and I/O facility), operating systems, and interconnection networks.
Location transparency	Resources can be accessed without knowing their physical locations. Users might need to know the names of resources but not their locations.
Access transparency	Local resources and remote resources are accessed in the same way. Whether the resources are from local or remote machines is irrelevant, as the way to access them does not differ.
Migration (mobility) transparency	Movement of resources and users' submitted jobs within the distributed system without affecting how the users operate the system
Replication transparency	Multiple instances of resources to be present throughout the system without the user or application programmers knowing anything about the replicas. Increases the reliability and performance of the services provided by the system.
Concurrency transparency	Multiple users can use the same resource at the same time without external mechanisms, to avoid interference. The users may not even be aware that there are other users who access the same resource. An example of this transparency is multiple processes, or users accessing a shared database simultaneously.
Failure transparency	Service is as usual when some failures occurred within the system Failure types include hardware, software, or communication link failures Users are not aware of the failures, to a certain extent, within the system.

11.2 Models of Distributed Systems

This section discusses the major models of distributed systems.

11.2.1 Models of Distributed Systems: Client-Server Model

Resource sharing in distributed systems is commonly achieved using the client-server model



The client and server are two systems connected by a network:

- A server machine manages certain resources such as printers, files or database records.
- A client machine accesses these resources by sending a request to the server.
- The server, upon receiving the request, provides the required service by sending a response to the client.

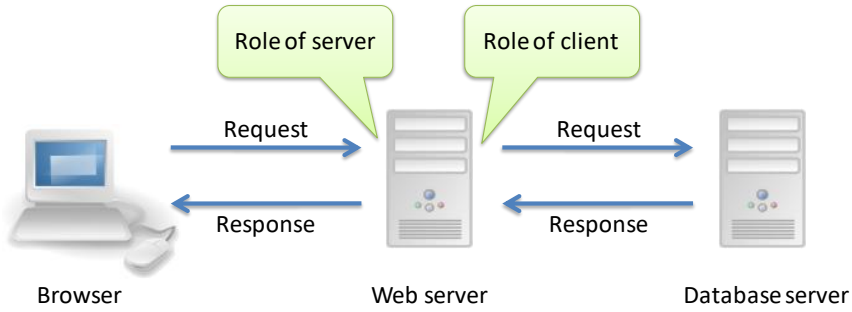
The terms client and server are better understood through their roles in a distributed system:

- A program action has the role of a client when it requests a service from another program.
- A program has the role of a server when it provides a service to another program.

One advantage of client-server model is improved resource utilization.

- For example, a printer is more utilized when it is shared by multiple client programs.

In some cases, a program may assume both the roles of a client and a server.



11.2.2 Computer Clusters

A computer cluster is a group of dedicated computers connected and configured to work together as a unit.

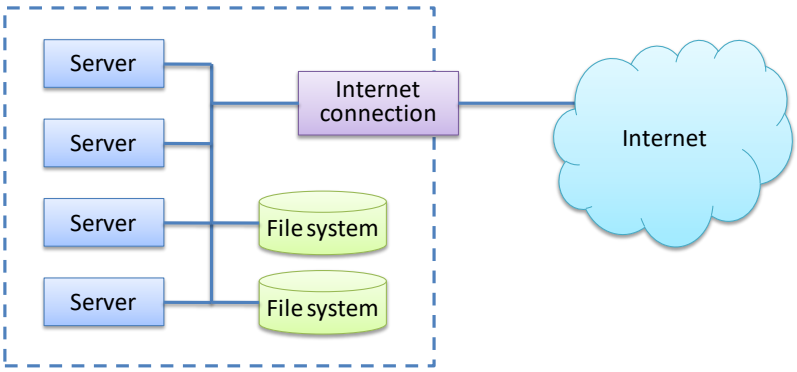
Users of the cluster are not aware of multiple computers present in the system, since they are configured to appear to users as a single processing machine.

Example: High Performance Web Server

A web server for large enterprises must satisfy a number of non-functional requirements:

- Scalability.
  - The capability to handle millions of accesses a day is not the same as the ordinary web servers.
- Availability.
  - The server should always work and respond to clients' requests all the time without any service interruptions.

A single computer for a Web server would make it very hard to satisfy these requirements. Clusters are applied to build Web servers for large enterprises. A cluster for a Web server contains a group of server machines and other resources, such as file systems, and acts like a single server system.



A cluster for web server

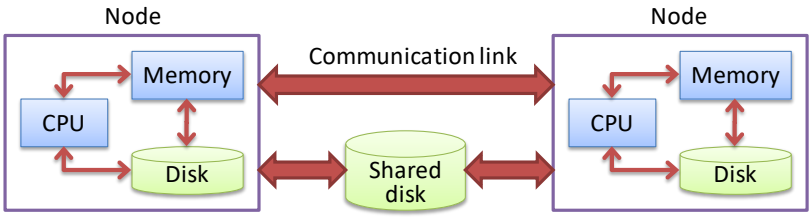
Each server within the cluster can process client connections independently.

A failure in any server will not affect the overall Web services, only causes a gradual degradation in response time, since one less server is used to provide the Web services.

Major Configurations of Clusters

Shared-nothing configuration and shared-disk configuration are two major configurations of clusters.

(a) Shared-nothing configuration



(b) Shared-disk configuration

Shared nothing configuration	***Shared-disk configuration***
Each node has its own local disk and no access of shared resources. Communication of all control and data messages is achieved via the links that connect the nodes.	The nodes share a disk on top of their individual local disks.
May involve a lot of data transfer on the communication link. Inter-node communication should be minimized	No need to transfer data using the communication links that connect the nodes. Controlling mechanisms to ensure the disk's data integrity during data update.

Characteristics and Benefits of Clusters

Computers in a cluster are designed to work for each other. They exhibit the following characteristics:

- Computers in a cluster are typically owned by a single company or organization.
- Computers in a cluster can be situated in a single location or scattered in multiple locations.
- Computers in a cluster are usually homogeneous
  - Identical or very similar in configuration.
  - Facilitates both purchase and control.

Clusters has a number of strengths:

Strengths	***Remarks***
Scalable	A cluster is scalable by nature because it is made up of multiple computers. Clustering achieves scalability in both the absolute sense and the incremental sense. For absolute scalability, a cluster with hundreds or even thousands of computers can be more powerful than standalone machines. For incremental scalability, a properly designed and configured cluster allows the addition of new computers to increase its processing power as the need arises.
Fault tolerance	Failure of a computer in a cluster will not break down the whole system. A cluster is able to switch the tasks to other functioning computers through failover.
High availability	Clusters are able to function 24 hours a day and 365/366 days a year. In many clusters, additions and removals of nodes (computing units) can be performed without shutting down the systems.
Load-balancing	Tasks can be distributed across the computers evenly to balance their workloads.
Superior performance-to-price ratio	Inexpensive personal computers can be used to build a cluster of very high performance.

Example: PC Cluster

A PC cluster implementation uses the cost-effective technology of personal computers. PC clusters are used in a number of applications, including scientific simulations, bio-technology research, financial modeling, data mining, high-volume Internet servers.

An early and well-known example of a PC cluster was the Beowulf cluster (http://www.beowulf.org/).

- Originated in 1994 in a NASA project that investigated the use of personal computers to build high-performance clusters.
- Technology developed in the project continues to evolve and is widely used in different systems.

Key characteristics of a Beowulf cluster include:

- Based on commodity hardware such as personal computers.
- Connecting by a private system network.

- Using open source software such as Linux.

There are two classes of PC clusters:

Class I Clusters	***Class II Clusters***
Built using commodity-off-the-shelf (COTS) components such as personal computers and the Linux operating system.	use specialized hardware to achieve higher performance.
This class of clusters is inexpensive to construct.	Board-based computers called blades can be installed in a standard-sized rack to build a cluster.

11.2.3 Grid Computing

Grid computing is a form of distributed systems that consume certain resources of individual computers in a network to achieve a goal. The individual computers are serving the users, and also playing a role in a distributed system.

- A grid computing system may send tasks to its associated computers for executing in their spare CPU processing time.
- Improve computing resource utilization and perform useful tasks.

Utilizing Spare Computing Power

For most computer systems, their computing power is rarely utilized.

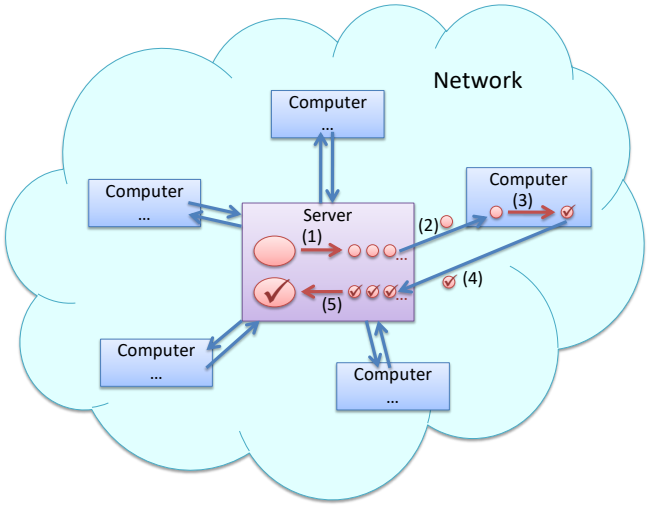
- Rarely are operations demanding a lot of processing (causing the feeling of slow computing).
- Most of the time, operations demand little processing power.

There is a pattern of stop-start in terms of processing power utilization. Examples:

- Using a word processor. User thinks and types.
- Using a web browser. User loads page and reads.
- Writing programs. User writes a program and compiles.

Mechanism of Grid Computing

A grid computer system works with a server controlling a number of participating computers.



A management server is connected to a number of computers in a network. As indicated in the figure, the steps of performing an operation are:

1. The server breaks down the task of a large operation into many small subtasks.
2. It sends each subtask to an associated computer in the network for execution.
3. A computer receives a subtask and executes it.
4. The computer sends the result of its completed subtask to the server.
5. The server receives all results and combines them to form a result of the original task.

Computers in grid systems are often heterogeneous, scattered, and owned by different parties.

Grid computing makes use of partial computing capacity of individual computers in a network to act as a computing system. Characteristics of computers used in grid computing include:

- Computers in a grid system belong to different companies, organizations or individuals, though it is possible that they belong to a single owner.
- Computers in a grid system are often scattered over a wide geographical area.
- Computers in a grid system are usually heterogeneous, or different in configuration.

Grid computing shares the same benefits of computer clusters, namely scalability, fault tolerance, high availability, load-balancing and good performance-to-price ratio.

Grid computing however has security concerns, because some computers are owned by others.

Examples of Grid Computing

The following lists two major grid computing projects.

- The SETI@home project.
  - SETI means Search for Extraterrestrial Intelligence at Home, http://setiathome.ssl.berkeley.edu/) project.
  - It involves the analysis of a huge amount of radio signals from space.
  - The aim is to detect life outside the earth.
  - Uses the spare processing power of individuals' home computers over the Internet to analyze those radio signals.
  - Each participating computer runs a special screen saver application that performs some analysis tasks when the computer is idle.

- The Folding@home project.
  - This project uses home computers over the Internet to perform simulations of protein folding.
  - Protein folding is about studying the structure of protein.

11.3 Cloud Computing

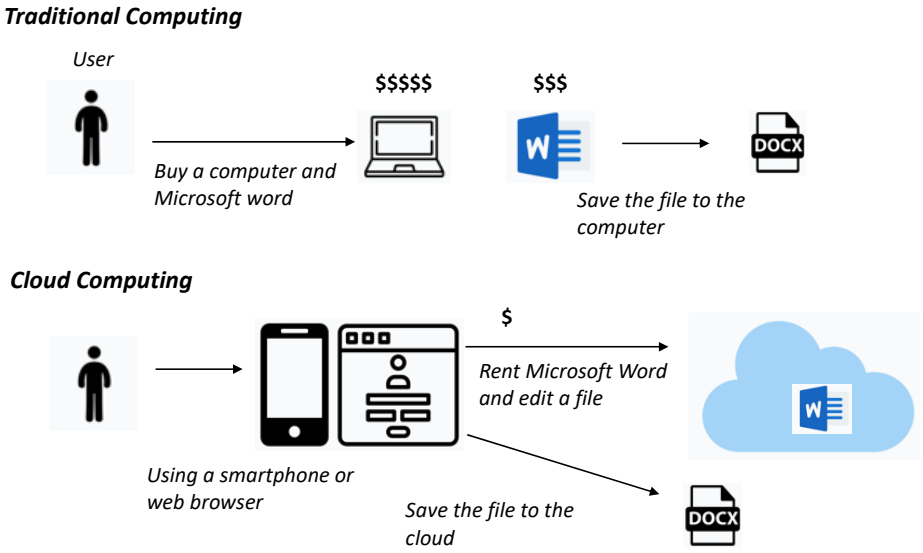
Cloud computing concerns about the provision of processing and computing facilities over the Internet. It turns computing services into utilities like electricity and water. Users pay for what they have used and how long they have used. It is a type of utility computing.

There is a potential for achieving the economy of scale and leading to overall financial benefits.

Computing resources may include computing power, memory, storage, and applications. They may be free or charged at a particular rate. Users can requisite various computing resources from anywhere.

Cloud computing releases the traditional view that supporting hardware and software must be provisioned directly. Now, these supporting resources can be rented or acquired in an ad hoc manner.

An example of cloud computing can be illustrated with a computing task such as editing a document.



Servers for cloud computing should be highly powerful and reliable. They are nearly always implemented by high-performance systems such as computer clusters.

11.3.1 Types of Cloud Computing: Application Level

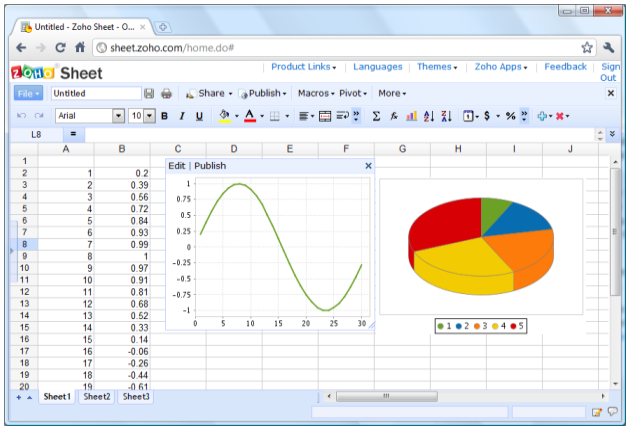
For the general users, the most visible form of cloud computing services is the provision of application software functionalities. Application level cloud computing is also known as Software as a Service (SaaS).

- One example is a web-based office productivity suite for preparing documents, spreadsheets and presentations.
- Files are edited on a web page and store them to the server of the website.

The functionalities of these cloud-based software are usually large subsets of their traditional counterparts (software installed in a computer).
Examples of available cloud-based office productivity suites include:

- Google Docs — http://docs.google.com/
- Zoho — http://www.zoho.com/
- Microsoft Office Web Apps — http://office.microsoft.com/en-us/web-apps/
- Oracle Cloud Office — http://www.oracle.com/us/products/applications/open-office/cloud-office-170206.html

There are many other kinds of cloud-based applications, ranging from personal finance (e.g., iMint at http://www.imint.in/) to project management (e.g., Basecamp at http://basecamphq.com/) to customer relationship management (e.g., Salesforce at http://www.salesforce.com/).



Characteristics of application level cloud computing include:

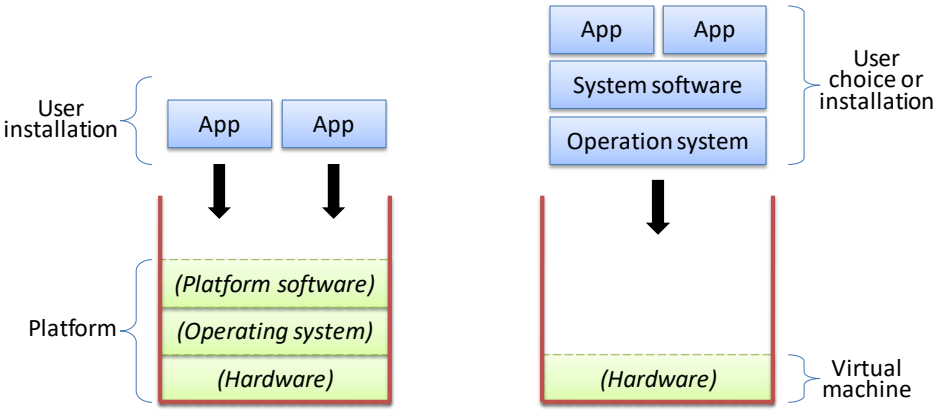
- Designed to be intuitive for users with experience in using other word processing applications.
- The speed of their operations depends on a number of conditions, including the server load, network traffic, and the complexity of your document.
- The functionalities of these applications should satisfy most general usage, but some very advanced functions may be missing.

11.3.2 Types of Cloud Computing: Platform Level

Cloud-based platforms are software environments for running applications of companies or individuals.

An online application using technologies is built compatible with a cloud-based platform service. The application is deployed to the platform instead of running it on a server.

This form of cloud computing is also known as Platform as a Service (PaaS).



(a) Cloud-based platform (b) Cloud-based infrastructure

Cloud platform service providers include:

- Google App Engine (GAE, http://code.google.com/appengine/)
- Windows Azure (http://www.microsoft.com/windowsazure/)
- Force.com (http://www.salesforce.com/platform/)
- Engine Yard (http://www.engineyard.com/)

11.3.3 Types of Cloud Computing: Infrastructure Level

Cloud-based infrastructure services provide virtual computers, or virtual machines, to the users.

- Virtual computers are accessed via the Internet.
- They appear as complete systems to users.
- Users may install and execute software of their choice to these virtual computers, including the operating system, system software and applications.
- Cloud-based infrastructures are more flexible than cloud-based platforms in term of the software that can be deployed.

This form of cloud computing is also known as Infrastructure as a Service (IaaS).

Cloud infrastructure provides include:

- Amazon Elastic Compute Cloud (Amazon EC2, http://aws.amazon.com/ec2/).
- Flexiant (http://www.flexiant.com/)
- GoGrid (http://www.gogrid.com/)

11.3.4 Benefits and Concern of Cloud Computing

Benefits of cloud computing include:

Benefits	***Remarks***
Cost saving	• Cloud-based services are charged on usage or even free. • Reduces the cost of hardware and software investment. • Especially for small companies and start-ups.

Maintenance	• Maintenance is carried out by the cloud service providers • Users still need to take care of the devices for accessing the services and the deployed specific software.
Scalability	• Hosted applications enjoy scalability provided by cloud-based platforms and infrastructures. • When demands increase, these applications automatically utilize more processing, bandwidth and storage resources without service degradation (possibly to a prearranged extent with the service providers).
Universal access	• Cloud-based services can be accessed universally over the Internet. • Data are primarily stored in the servers of the providers.
Automatic software update	• Applications are hosted in the servers of the providers. • It is the most updated version as hosted in the servers.
Sharing and collaboration	• Cloud-based applications store user data online, which facilitates sharing of data and collaboration of teamwork. • For example, a document may be viewed or edited by multiple users concurrently.
Reliability for hosted applications	• Cloud-based platforms and infrastructures are more stable and reliable than server systems of an average company. • Applications hosted there generally have higher availability than being hosted in-house.

There are also some concerns:

- Online requirement. Cloud-based applications require users to be online.
- Reliability of cloud-based applications. Reliability has two aspects in cloud computing. Cloud-based platforms and infrastructures are more reliable than in-house systems, but cloud-based applications are considered less reliable than traditional applications in a computer.
- Security. It concerns both user data of cloud-based applications and hosted applications in cloud-based platforms and infrastructures. Storing confidential information in remote servers of cloud service providers can be a potential risk. This is true even if certain security measures are implemented by the service providers.

11.4 Virtualization

Virtualization is a general concept in computing. It is an illusion of a computer system created for users. The underlying real and physical system is hidden away.

In virtual memory systems, programs are given the illusion that the memory space is large enough. In fact, some part of the memory space is not in the physical memory.

Users are given a virtual server and they have the impression that they own the whole server. In fact, there are multiple users sharing the same physical server. Virtualization is in the middle that performs the work to create the illusion.

Virtualization technologies are not new.

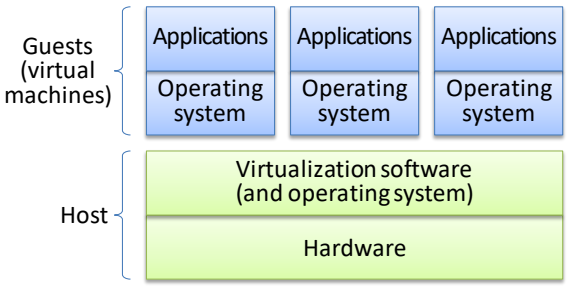
In the late 1960s, an IBM VM/370 mainframe machine could divide its resources into multiple partitions for running separate operating systems.

Each partition was called a pseudo machine, which is equivalent to virtual machine in today's terminology.

11.4.1 Types of Virtualization: Platform Virtualization

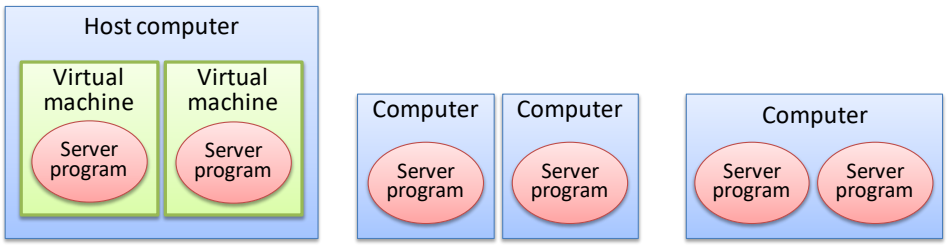
Platform virtualization is also known as machine virtualization.

- It creates a number of virtual machines from a physical system (the host)
- The virtual machines (the guests) operate independently of each other.
- Users see the virtual machines as individual computers.



Platform virtualization is used to mainly achieve the following two purposes:

Server Virtualization	**Desktop Virtualization**
Virtual machines are set up as servers so that each application (such as email or web servers) can be set up in each virtual machine.	Virtual machines are set up as desktops so that multiple operating systems or platforms can be installed on a physical machine.
Save hardware and operating costs	Running multiple operating systems concurrently.
Improve reliability by avoiding interference.	Useful for developing and testing systems
Widely used in data centers and web hosting.	



(a) Server virtualization (b) Using two computers (c) Using one computer

Examples of server virtualization software:

- VMware ESX/ESXi — http://www.vmware.com/products/vsphere/esxi-and-esx/index.html
- Microsoft Hyper-V — http://www.microsoft.com/hyper-v-server/en/us/default.aspx
- Citrix XenServer — http://www.citrix.com/xenserver

Examples of desktop virtualization software:

- VMware Workstation — http://www.vmware.com/products/workstation/
- VirtualBox — http://www.virtualbox.org/
- Microsoft Virtual PC — http://www.microsoft.com/windows/virtual-pc/
- QEMU — http://www.qemu.org/
- Bochs — http://bochs.sourceforge.net/

11.4.2 Types of Virtualization: Software Virtualization

Software virtualization provides virtual software environments for executing applications.

The virtual environments may be at the level of operating systems or applications. Four major types of software virtualization are operating system virtualization, kernel-level virtualization, application virtualization, and library emulation.

Operating system virtualization is also called shared kernel virtualization and system level virtualization.

Providing operating system guest environments that are identical to the host operating system by sharing the kernel. In Linux/UNIX systems, this can be done by the 'chroot' command.

Kernel-level virtualization allows a host system to execute modified guest kernels at the application level.

Guest kernels are user-mode applications running on the host. Each of the host and guest systems uses own kernel. Examples include User-Mode Linux (UML, http://user-mode-linux.sourceforge.net/) and Kernel Based Virtual Machine (KVM, http://www.linux-kvm.org/).

Application virtualization is often called a sandbox. It provides a standalone execution environment that is isolated from the underlying operating system.

Useful in software distribution and testing. An instance of application virtualization software is VMware ThinApp (http://www.vmware.com/products/thinapp/).

Library emulation provides the operating system level system calls of a guest system in a different host system. For example, Wine (http://www.winehq.org/) implements Windows Application Programming Interfaces (APIs) on Linux/UNIX and thus enables you to run Windows programs in Linux/UNIX systems.

11.4.3 Other Types of Virtualization

Presentation Virtualization	***Network Virtualization***	***Storage Virtualization***
Provides virtual software environments for executing applications.	Implements logical networks to the users.	Provides a logical storage device from one or more physical storage devices.

Users can view and control remote systems through terminal services.	A group of computers behave and are used as if they are connected in a network.	The coherent view of the logical storage device simplifies the access, control and management of the physical devices.
	Regardless of whether or how they are actually connected in a real network.	The physical devices may be heterogeneous or distributed over a network.
Examples: Remote Desktop Services and Virtual Network Computing	Examples: Virtual LAN	Examples: Storage Area Network (SAN).

Virtual memory enables applications to use a logical addressing space that is much larger than the physical memory in a system. Virtual memory is available in almost all modern computer systems.

11.4.4 Benefits of Platform Virtualization

Platform virtualization is arguably an important form of virtualization.

There are a number of benefits:

Benefits	***Remarks***
Cost savings	Server virtualization reduces hardware cost and operating cost.
Reliability	Virtual machines operate independently of and do not interfere with each other.
Improved utilization	Employing multiple virtual machines to run multiple server programs or applications. Improves computer utilization effectively without the fear of the programs interfering with each other.
Disaster recovery	Backups of guest systems can be done easily by copying one or a few files that implement the virtual machine and its virtual hard disks. This is often a much faster and more readily available solution than setting up and putting another physical machine to use.
Ease of management	Guest systems in virtual machines of a virtualization host can be managed by an administrative console of the virtualization system.
Ease of development	Virtual environments can be set up easily for testing and developing applications.
Ease of deployment	Simple and fast to deploy several servers of similar configurations. Set up the server in one virtual machine, clone a few copies, and adjust them.

11.4.5 Virtual Machine Technologies

There are different technologies in building virtual machines in a host system: emulation, hypervisors and paravirtualization.

Emulation

The concept of emulation is simple.

- An emulator is a software application that pretends to be a processor. and interprets each CPU instruction given to it. Common peripherals are often provided by an emulator for use in the guest system.

The processor of a guest system can be different from that of the host system.

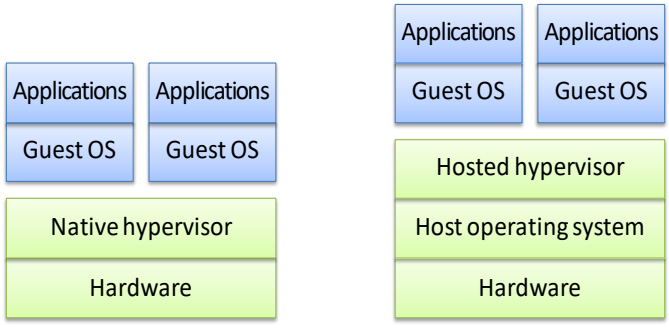
- Convenient in software development.
- An ARM android phone or device can be emulated when developing using an Intel x86 personal computer.

Native and hosted hypervisors

A hypervisor is a virtualization software layer that interfaces with guest systems on one side, and the computer hardware or host operating system on the other side.

- Called a virtual machine monitor in an IBM mainframe of the old days.
- Most instructions of guest systems can be performed natively and directly by underlying hardware

There are two main configurations: native hypervisor and hosted hypervisor

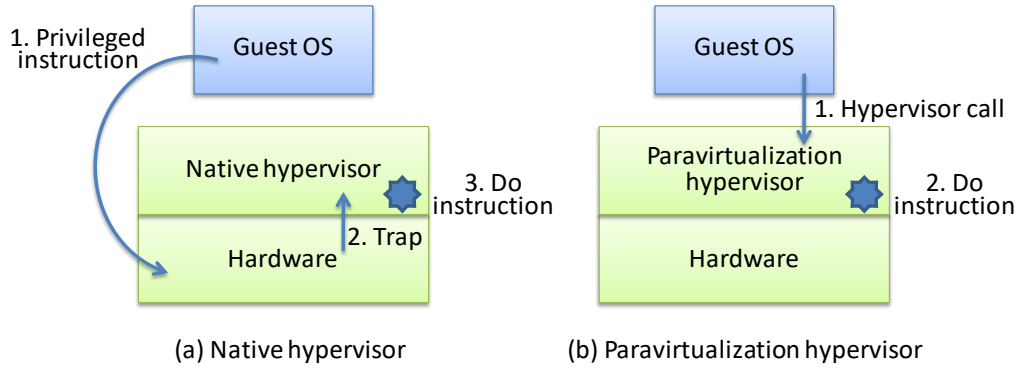


(a) Native hypervisor (b) Hosted hypervisor

Paravirtualization

Paravirtualization is another virtualization approach that uses a hypervisor.

- Guest systems in paravirtualization are running modified operating systems.
- The modifications change privileged (kernel-mode) instructions to special calls to the hypervisor. These privileged instructions are said to be para-virtualized.
- Paravirtualization may be faster than native and hosted hypervisor systems in some situations.
- Using modified guest operating systems can cause problems in software maintenance.



Citrix XenServer is an example of a paravirtualization hypervisor.