Bedtime Stories on

Operating Systems

Bedtime Stories on

Operating Systems

Contents

UNIT 1
AN INTRODUCTION TO OPERATING SYSTEMS

A **computer** is a general purpose device that can execute sequences of instructions presented in a formal format to perform numerical calculations and other tasks.

Revise: block diagram of a computer, concept of memory hierarchy.

Computer science is the study of computer systems and computing processes.

Read: Newell, A., Perlis, A. J. and Simon, H. I. 1967. Is 'computer science' science or engineering? *Science*, **167**(3795): 1373-1374.

Computer hardware is the collection of all physical elements of the computer system.
Computer software is the collection of all programs stored in and executed by a computer system.

Application software performs specific task for the user.
System software operates and controls the computer system, and provides a platform to run application software.

An **operating system** is a piece of software that manages all the resources of a computer system, both hardware and software, and provides an environment in which the user can execute his/her programs in a convenient and efficient manner.

Operating systems exist because they offer a reasonable way to solve the problem of creating a usable computer system.

An operating system –

- manages the computer hardware
- facilitates execution of application programs
- acts as an intermediary between the user and the computer hardware
- designed to be convenient and efficient

User

Application programs
Operating system
Computer hardware

The operating system provides the means for proper use of the resources in the operation of the computer system.

Design goals –

- convenience (ease of use)
- efficiency (proper resource allocation)
- energy conservation
- minimal user interference

- personal computers
- high performance computers
- handheld devices
- embedded devices

An operating system acts as an –
- **Resource allocator**
- **Control program**

First operating system –
- ATLAS (Manchester Univ., late 1950s – early 1960s)
- evolved from control programs

Types of operating systems –
- Single process operating system [MS DOS, 1981]
- Batch-processing operating system [ATLAS, Manchester Univ., late 1950s – early 1960s]
- Multiprogramming operating system [THE, Dijkstra, early 1960s]
- Multitasking operating system [CTSS, MIT, early 1960s]

Multiprogramming increases CPU utilization by keeping multiple jobs (code and data) in the memory so that the CPU always has one to execute.

Job 1
Job 2
Job 3
Operating system

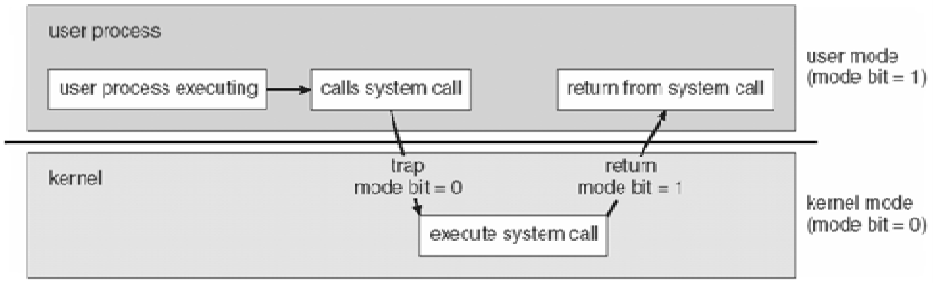
Multitasking is a logical extension of multiprogramming.
CPU executes multiple tasks by switching among them.
The switching is very fast.
Requires an interactive (hands-on) computer where the user can directly interact with the computer.
Response time should be minimal.

A **kernel** is that part of the operating system which interacts directly with the hardware and performs the most crucial tasks.
A **microkernel** is much smaller in size than a conventional kernel and supports only the core operating system functionalities.
A **shell**, also known as a command interpreter, is that part of the operating system that receives commands from the users and gets them executed.

A **system call** is a mechanism using which a user program can request a service from the kernel for which it does not have the permission to perform.
User programs typically do not have permission to perform operations like accessing I/O devices and communicating other programs.
A user program invokes system calls when it requires such services.
System calls provide an interface between a program and the operating system.
System calls are of different types.
E.g. – fork, exec, getpid, getppid, wait, exit.

Dual-mode operation –
- **User mode**
- **Kernel mode** / supervisor mode / system mode / privileged mode

Mode bit– 0: kernel, 1: user
Request using a system call



Duties of an operating system –

- Process management
  - creating and deleting user and system processes
  - suspending and resuming processes
  - interprocess communication
  - process synchronization
  - deadlock handling
- Memory management
  - Keeping track of which part of memory is being used by which job
  - Allocating and deallocating memory space
- Storage management
  - file system management
    - creating, deleting and manipulating files and directories
  - mass storage management
    - free space management
    - storage allocation
    - disk scheduling
- Caching
- Input-output management

Operating system services –

- Helpful to the user
  - user interface (CUI/shell and GUI)
  - program execution
  - I/O operation
  - file system manipulation
  - communication
  - error detection
- Helpful to the system
  - resource allocation
  - accounting
  - protection and security

Operating system structures –

- Monolithic [MS DOS, Unix, Linux]
- Layered [THE]
- Microkernel [Mach, MINIX]

A **real-time operating system** (**RTOS**) has well-defined and fixed time constraints which have to be met or the system will fail.
An RTOS is used when rigid time constraints have been placed on the operation of processes or flow of data.

An RTOS is often used in the control device in a dedicated application.
Hard- and soft- RTOS.
Applications: embedded systems, robotics, scientific utilities, etc.

Operating systems for smart phones –

- CPUs of smart phones are made to be much slower to conserve energy

Booting is the process of starting the computer and loading the kernel.
When a computer is turned on, the power-on self-tests (POST) are performed.
Then the bootstrap loader, which resides in the ROM, is executed.
The bootstrap loader loads the kernel or a more sophisticated loader.

Is a **device driver** a part of the operating system?

Case studies: CP/M, MS DOS, Unix, Linux, Windows, etc.

http://amturing.acm.org –

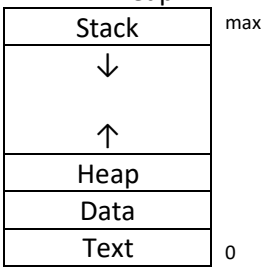
- Edsger Wybe Dijkstra
- Kenneth Lane Thompson
- Frederick Phillips Brooks, Jr.
- Barbara Liskov

> Chapters: 1 and 2.
> Sections: 1.1, 1.4 – 1.9, 1.11, 2.1 – 2.5, 2.7 and 2.10.

UNIT 2
PROCESS AND PROCESS SCHEDULING

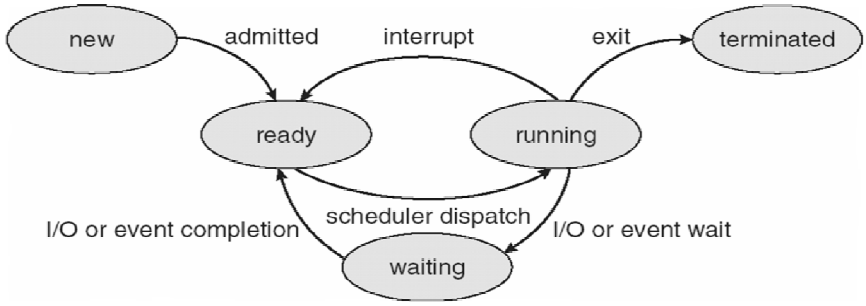
A **process** is a program in execution.
A process is a unit of work in a computer system.
The terms process and job are used interchangeably.

A process comprises of –
- text section containing the program code
- current activity represented by the values of the program counter and other registers
- program stack
- data section containing global variables
- heap



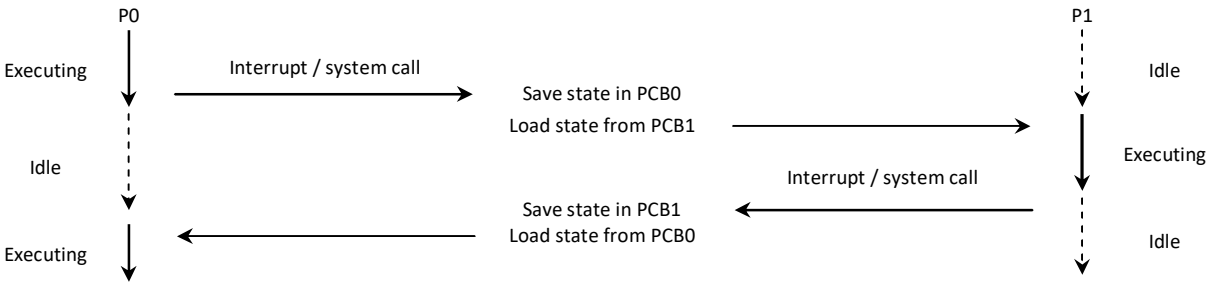
A program is a passive entity while a process is an active entity.

Process state is defined by the current activity of the process.
As a process executes, its state changes.



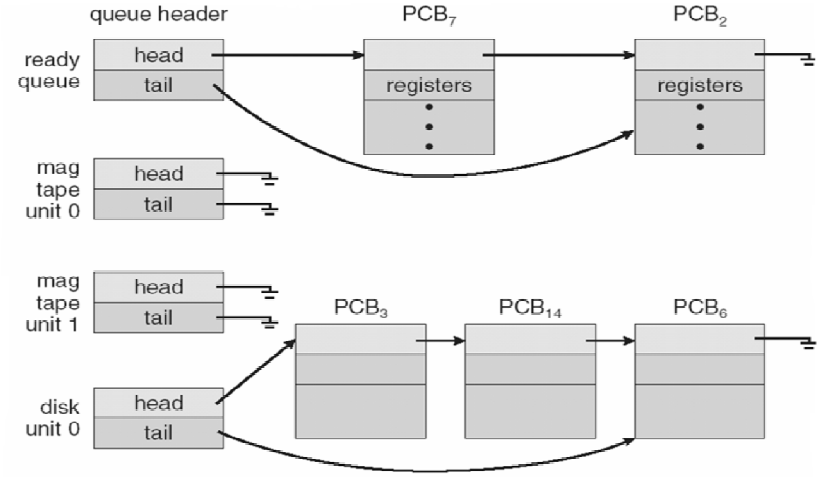
Only one process can be in the running state at any instant.
Many processes can be ready or waiting.

Each process is internally represented by the operating system by a **process control block** (**PCB**) also called task control block.
PCB contains all information associated with the process –
- Process state
- Values of program counter and other registers
- CPU scheduling information - priority, pointer to scheduling queue, etc.
- Accounting information - process id, CPU- and real- time used, time limits, etc.
- I/O status information - list of i/o devices allocated, list of open files, etc.

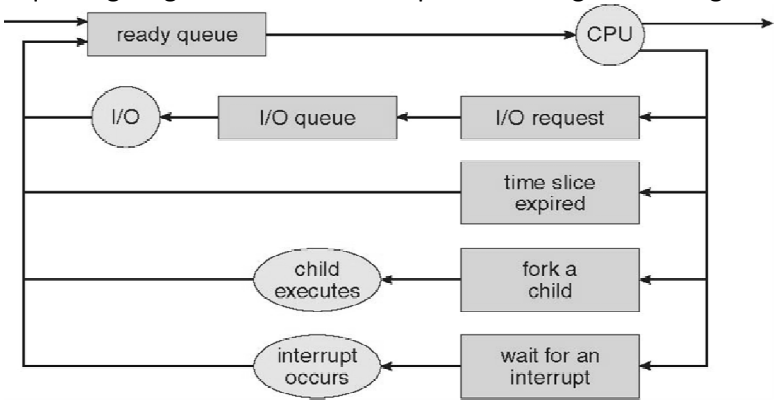


Process scheduling is selecting one process for execution out of all the ready processes.
The objective of multiprogramming is to have some process running at all times so as to maximize CPU utilization.
The objective of multitasking is to switch the CPU among the processes so frequently that the user can interact with each process while it is running.
To meet these objectives the process scheduler selects one of the available processes for execution.

Scheduling queues are used to perform process scheduling.
As a process enters the system, it is put in a **job queue** that contains all the processes in the system.
The processes that are residing in the memory and are ready for execution are kept in the **ready queue**.
The ready queue is implemented as a linked list of PCBs with a header containing pointers to the first and the last PCBs.
The list of the processes waiting for a particular i/o device is called a **device queue**.
Each device has its own device queue.



A queuing diagram shows how the processes migrate among the various scheduling queues.



There are three types of **schedulers**.
A process migrates among the various scheduling queues throughout its lifetime.
The operating system has to select the processes from the queues according to some criteria.
The selection is done by the appropriate scheduler.

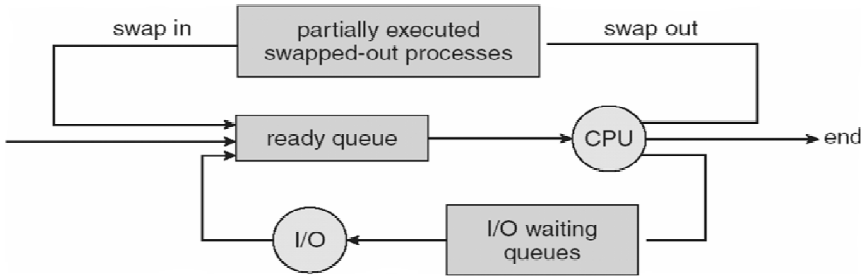
A **long-term scheduler** (**job scheduler**) selects processes from those submitted by the user and loads them into the memory.
The long-term scheduler controls the degree of multiprogramming which is represented by the number of processes in the memory.
It is invoked less frequently.

A **short-term scheduler** (**CPU scheduler**) selects one of the processes in the memory and allocates the CPU to it.
The short-term scheduler is invoked frequently and should be very fast.

The long-term scheduler should select a proper mix of CPU-bound processes and i/o-bound processes.
A CPU-bound process spends most of its time doing computations.
An i/o-bound process spends most of its time doing i/o.

Some multitasking operating systems, like Unix and Windows, do not use long-term schedulers.
All new processes are put in the memory for the perusal of the short-term scheduler.

A **medium-term scheduler** removes processes from the memory and from the competition for the CPU, thus reducing the degree of multiprogramming.
The processes can be later reintroduced in the memory and their executions can be resumed.
This scheme is called **swapping**.
Swapping may be used to improve process mix and to free up some memory in uncontrollable circumstances.



Context switching is done to switch between processes.
Switching the CPU to another process requires saving the state of the current process and reloading the state of another process.
States are saved into and reloaded from PCBs.
Context-switch time is a pure overhead as the system does not do any useful work during a control switch.
Context-switch time depends highly on the hardware.
Context switching is faster on RISC processors with overlapped register windows.

Process creation is one process creating another process.
The processes are called parent process and child process, respectively.
Each process has a unique id.
A process may obtain resources either from its parent or from the operating system directly.

A parent process may continue executing with its children processes or may wait for them to complete.
A process may be a duplicate of its parent process (same code and data) or may have a new program loaded into it.

Process termination marks the deletion of the PCB of the process.
A parent process may terminate a child process –

- if it has exceeded its resource usage
- if its result is no more needed
- if the parent process is terminating and the operating system does not allow an **orphan process** (this may lead to cascading process terminations)

Typically, the kernel is the first process to be created, is the ancestor of all other processes and is at the root of the process tree.

A **zombie process** is a process that has terminated but its PCB still exists because its parent has not yet accepted its return value.

Interprocess communication –

- Reasons –
    - information sharing
    - computational speedup
    - modularity
    - convenience
- Models –
    - shared memory
    - message passing [send(P,message) and receive(id,message)]

A **thread** is the smallest sequence of instructions that can be managed independently by a scheduler.
A thread is a component of a process.
Multiple threads can exist within the same process, executing concurrently and share resources such as memory.
The threads of a process share its instructions (executable code) and its context (the values of its variables at any given moment).

Difference between process and thread –

- processes are typically independent while threads exist as parts of a process
- processes carry considerably more state information than threads, whereas multiple threads within a process share process state as well as memory and other resources
- processes have separate address spaces, whereas threads share their address space
- processes interact only through system-provided inter-process communication mechanisms
- context switching between threads in the same process is typically faster than context switching between processes

Advantages of multi-threaded programming –

- responsiveness
- faster execution
- better resource utilization
- easy communication
- parallelization

The execution of a process consists of alternate **CPU bursts** and **i/o bursts**, starting and ending with CPU bursts.

The CPU scheduler is invoked when a process –
- switches from running state to waiting state (condition 1)
- switches from running state to ready state (condition 2)
- switches from waiting state to ready state (condition 3)
- terminates (condition 4)

In **non-preemptive scheduling** or cooperating scheduling, a process keeps the CPU until it terminates or switches to the waiting state.
Some machines support non-preemptive scheduling only.
E.g. – Window 3.1x.

In **preemptive scheduling**, a process can be forced to leave the CPU and switch to the ready queue.
E.g. – Unix, Linux, Windows 95 and higher.

CPU scheduling is optional for conditions 2 and 3, but necessary in the other two conditions.

MS DOS does not support multiprogramming, hence no CPU scheduling.

A **dispatcher** is the module of the operating system that gives control of the CPU to the process selected by the CPU scheduler.
Steps –
- switching context
- switching to user mode
- jumping to the proper location in the user program

Dispatch latency is the time taken to stop a process and start another.
Dispatch latency is a pure overhead.

Scheduling criteria –
- ↑ **CPU utilization** - percentage
- ↑ **Throughput** - number of processes completed per unit time
- ↓ **Turnaround time** - time from submission to completion (time spent in different queues + time spent in CPU + time spent in different i/o devices)
- ↓ **Waiting time** - time spent in ready queue (only)
- ↓ **Response time** - time from submission to first response

Variance in response time should be minimal.

Gantt chart (Henry Gantt)

Scheduling algorithms are used to select a process for execution.
There are several well known scheduling algorithms.

First-come first-served (FCFS) scheduling –
- non-preemptive
- high average waiting time
- **convoy effect** - several small processes may need to wait if a large process is given the CPU

Exercise 1.

Process	*Arrival time (ms)*	*CPU burst time (ms)*
P1	*0*	*24*
P2	*1*	*3*
P3	*2*	*3*

Calculate throughput, average turnaround time and average waiting time.

Shortest-job-first (SJF) scheduling –

- process with the smallest next CPU burst is selected
- FCFS to break ties
- more appropriate term: shortest-next-CPU-burst-first
- optimal, but cannot be implemented

We may predict the length of the next CPU burst.
We use exponential average.
$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$, where $0 \leq \alpha \leq 1$.
If $\alpha = 0$, predicted length is constant.
If $\alpha = 1$, predicted length is equal to that of the last (actual) CPU burst.
Typically, $\alpha = 0.5$.
τ_0 = constant or system average.

SJF can be either preemptive or non-preemptive.
Preemptive SJF = shortest-remaining-time-first.

Exercise 2.

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

Calculate throughput, average turnaround time and average waiting time for (a) non-preemptive and (b) preemptive scheduling.

Priority scheduling –

- CPU is allocated to the process with the highest priority
- priority range be 0 to 7 (say), with 0 representing the highest or the lowest priority
- priority may depend on internal factors (time limit, memory requirement, number of open files, etc.) and external factors (user, department, etc.)
- may be preemptive or non-preemptive
- SJF is a special case of priority scheduling, with priority inversely proportional to predicted next CPU burst length
- may cause starvation, *i.e.* indefinite blocking of processes
- **aging**: gradually increase the priority of a process waiting for a long time
- **priority inversion**: a low-priority process gets the priority of a high-priority process waiting for it

Exercise 3.

Process	*Arrival time (ms)*	*CPU burst time (ms)*	*Priority*
P1	*0*	*8*	*7*
P2	*2*	*4*	*5*
P3	*4*	*6*	*0 (highest)*
P4	*6*	*4*	*1*

Dispatch latency = 1 ms.
Calculate CPU utilization, throughput, average turnaround time and average waiting time for (a) non-preemptive and (b) preemptive scheduling.

Round robin (RR) scheduling –

- a small **time quantum** or time slice is defined
- the ready queue is treated as a circular queue
- each process is allocated the CPU for one time quantum
- preemptive
- if time quantum is too large, then RR behaves like FCFS
- if time quantum is too small, then RR behaves like processor sharing
- rule of thumb: 80% CPU bursts should be shorter than the time quantum

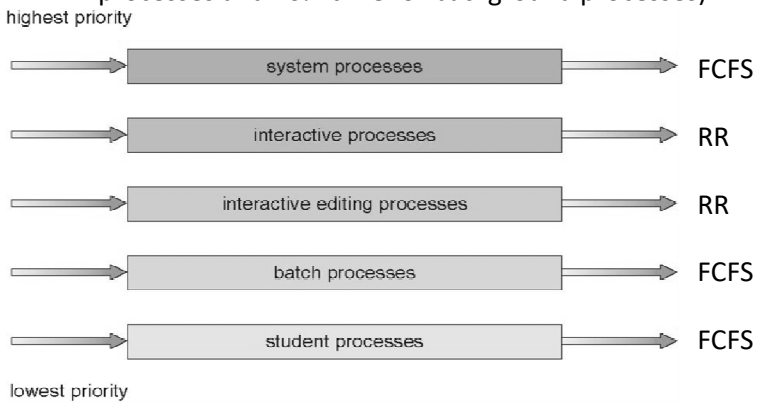
Exercise 4.

Process	*Arrival time (ms)*	*CPU burst time (ms)*
P1	*0*	*18*
P2	*2*	*4*
P3	*4*	*6*
P4	*6*	*12*

Calculate throughput, average turnaround time and average waiting time for (a) time quantum = 8 ms and (b) time quantum = 2 ms.

Multilevel queue scheduling –

- the ready queue is partitioned into several queues
- a process is permanently assigned to one queue
- each queue has its own scheduling algorithm
- inter-queue scheduling: preemptive priority scheduling or RR (80% time for foreground processes and 20% time for background processes)



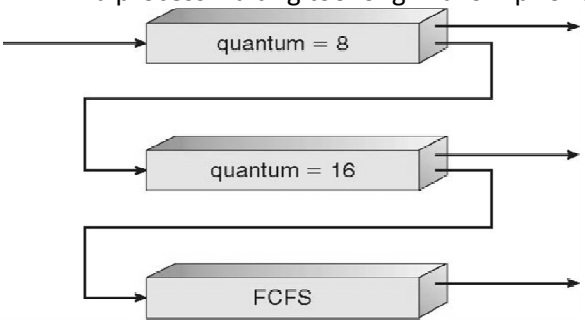
Exercise 5.

Process	*Arrival time (ms)*	*CPU burst time (ms)*	*Queue*
P1	*0*	*18*	*Q2*
P2	*2*	*4*	*Q2*
P3	*4*	*6*	*Q1 (highest-priority)*
P4	*6*	*12*	*Q2*

Q1: FCFS
Q2: RR, time quantum = 2 ms
Inter-queue: preemptive priority scheduling
Calculate throughput, average turnaround time and average waiting time.

Multilevel feedback queue scheduling –

- allows processes to move between queues
- inter-queue scheduling: preemptive priority scheduling
- a process waiting too long in a low-priority queue may be moved to a high-priority queue



Exercise 6.

Process	*Arrival time (ms)*	*CPU burst time (ms)*
P1	*0*	*32*
P2	*2*	*10*
P3	*4*	*16*
P4	*6*	*30*

Calculate throughput, average turnaround time and average waiting time.

Comparison –

	☺	☹
First come first served scheduling	Simple	Inefficient (high turnaround time, waiting time and response time)
Shortest job first scheduling	Most efficient	Impossible to implement
Priority scheduling	Low waiting time for high priority processes	Indefinite blocking of low priority processes
Round robin scheduling	Efficient and no indefinite blocking of processes	Too much context switching
Multilevel queue scheduling	Low waiting time for high priority processes	Complex
Multilevel feedback queue scheduling	Fast turnaround for short processes	Complex

Multiple processor scheduling –

- multiprocessor systems: homogeneous or heterogeneous
- CPU scheduling approaches
    - asymmetric multiprocessing - one processor performs scheduling for all
    - symmetric multiprocessing - each processor is self-scheduling
    - e.g. – Linux supports symmetric multiprocessing
- processor affinity –
    - because of high cost of invalidating and repopulating caches, migration of processes from one processor to another is avoided
    - attempt is made to keep a process at a given processor
    - hard- and soft- affinity
- load balancing –
    - attempt to keep all processors equally loaded
    - pull migration, push migration, combination
    - only for symmetric systems
    - counteracts the benefits of processor affinity

Chapters: 3 – 5.
Sections: 3.1 – 3.4, 4.1, 4.2 and 5.1 – 5.4.

UNIT 3
PROCESS SYNCHRONIZATION

A cooperating process is one that can affect or be affected by other processes executing in the system.

P0	P1
…	…
i++;	i--;
…	…
R1 ← i	R2 ← i
R1 ← R1 + 1	R2 ← R2 - 1
i ← R1	i ← R2

Initially, i = 10
P0: R1 = 10
P1: R2 = 10
P0: R1 = 11
P1: R2 = 9
P0: i = 11
P1: i = 9

Race condition is a situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the accesses take place.
To avoid such situations, it must be ensured that only one process can manipulate the data at a given time.
This can be done by **process synchronization**.

A **critical section problem** is defined as follows –

- there are n processes, *viz*. P_0, P_1, P_2 … P_{n-1}
- each process has a section of code, called the **critical section**, in which the process changes common variables and files
- the problem is to ensure that when one process is executing in its critical section then no other process can execute its own critical section
- the critical section is preceded by an **entry section** in which a process seeks permission from other processes
- the critical section is followed by an **exit section**
- the remaining code is called the **remainder section**

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

A solution to the critical section problem must satisfy the following properties –

- **Mutual exclusion**
- **Progress** - If no process is in its critical section and some processes want to enter their critical sections, then the processes which are in their entry sections or exit sections decide which process will enter its critical section next. This selection cannot be postponed indefinitely.
- **Bounded waiting** - There is a limit on the number of times other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Peterson's solution (2 processes only) –

Shared variables:

```
int turn;
Boolean flag[2];
```

Code for P_i:

```
do {  flag[i] = true;
      turn = j;
      while (flag[j] && turn == j)
        ;
      critical section
      flag[i] = false;
      remainder section
   } while (true);
```

Prove: Peterson's solution satisfy the three properties.

The critical section problem can be also solved by dedicated hardware.

A **semaphore** is an integer variable that, apart from initialization, is accessed only through two atomic operations called wait() and signal().

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

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

Only one process can access a semaphore at a time.

Binary semaphore –

- also called mutex lock
- used to implement solution of critical section problem with multiple processes
- initialized to 1

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

Counting semaphore –

- used to control access to a resource that has multiple instances
- initialized to n

Semaphores may lead to indefinite wait (starvation) and deadlocks.

```
P0              P1
wait (S);       wait (Q);
wait (Q);       wait (S);
…               …
signal (S);     signal (Q);
signal (Q);     signal (S);
```

Producers – consumers problem (bounded buffer problem)

Number of buffers = n

Semaphores –

```
mutex = 1; //access to buffers
empty = n;
full = 0;
```

Producers

```
do  {  //   produce an item
       wait (empty);
       wait (mutex);
       //  add the item to the  buffer
        signal (mutex);
       signal (full);
    } while (true);
```

Consumers

```
do  {  wait (full);
       wait (mutex);
       //  remove an item from  buffer
       signal (mutex);
       signal (empty);
       //  consume the item
    } while (true);
```

Readers – writers problem

Shared data –

```
int readcount = 0;
```

Semaphores –

```
mutex = 1;
wrt = 1;
```

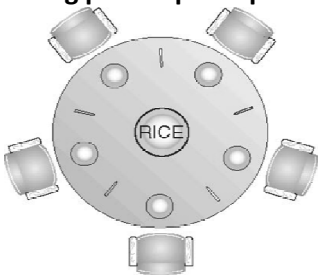
Writers

```
do  {  wait (wrt);
       //  write
       signal (wrt);
    } while (true);
```

Readers

```
do  {  wait (mutex);
       readcount++;
       if (readcount == 1)
          wait (wrt);
       signal (mutex);
       //  read
       wait (mutex);
       readcount--;
       if (readcount == 0)
          signal (wrt);
       signal (mutex);
    } while (true);
```

Dining philosophers problem

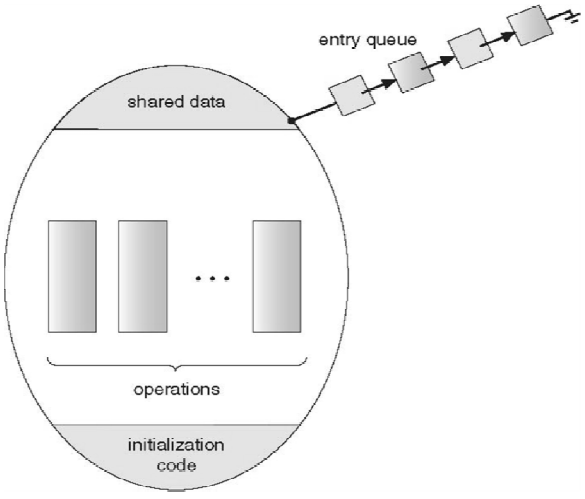


```
Semaphores –
        chopsticks[5];
        all initialized to 1
Pi
do  {  wait (chopstick[i]);
       wait (chopStick[(i+1)%5]);
       // eat
       signal (chopstick[i]);
       signal (chopstick[(i+1)%5]);
        // think
    } while (true);
```

If all philosophers get hungry simultaneously then there will be a deadlock.

A **monitor** is a high-level process synchronization construct.
Only one process can be active within the monitor at a time.
A monitor type presents a set of programmer defined operations that are provided mutual exclusion within the monitor.
A monitor can have variables of the condition type that can be accessed by wait() and signal() operations only.
The operation x.wait() means that the process making this operation is suspended until another process invokes x.signal().
The operation x.signal() resumes exactly one suspended process and has no effect if there is none.
Syntax:

```
monitor monitor_name
{       // shared variable declarations
        initialization_code (…) { … }
        procedure P1 (…) { … }
        procedure Pn (…) { … }
}
```



Solution of the dining philosophers problem using a monitor:

```
monitor dp
{       enum {THINKING, HUNGRY, EATING} state[5] ;
        condition self [5];
```

```
        initialization_code()
        {for (int i=0; i < 5; i++)
         state[i] = THINKING;
        }

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

        void pickup (int i)
        {state[i] = HUNGRY;
         test(i);
         if (state[i]!=EATING)
                self[i].wait;
        }

        void putdown (int i)
        {state[i] = THINKING;
         test((i+4)%5); // test left neighbor
         test((i+1)%5); // test right neighbor
        }
}

Pi
…
dp.pickup(i);
…
// eat
…
dp.putdown(i);
…
```

Chapter: 6.
Sections: 6.1 – 6.7.

UNIT 4
DEADLOCKS

In a multiprogramming system, several processes may compete for a finite number of resources.
A process requests for resources, and if the resources are not available at the time then the process enters the waiting state.
Sometimes, a process will wait indefinitely because the resources it has requested for are being held by other similar waiting processes.
Deadlock is a situation in which two or more processes are waiting indefinitely because the resources they have requested for are being held by one another.

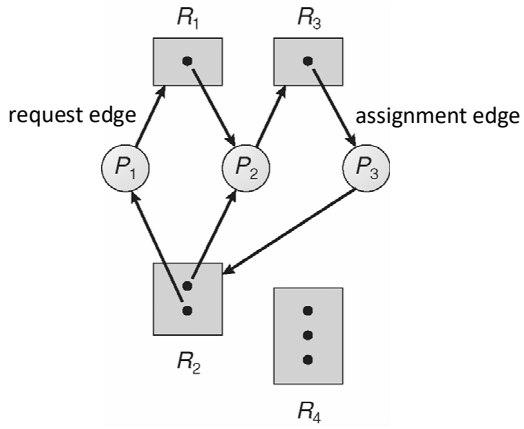
The resources are partitioned into several types, each of which has several identical instances.
E.g. – memory spaces, registers, i/o devices, etc.
If a process requests for an instance of the resource type, then the allocation of any one instance of the resource type will satisfy the process.
A process utilizes a resource in the following sequence: request → use → release.

Necessary conditions for deadlocks –

- **Mutual exclusion** - one or more non-sharable resources
- **Hold and wait** - a process is holding some resources and waiting for other resources
- **No preemption** - resources cannot be preempted
- **Circular wait** - a set {P0, P1, P2 … Pn} exist such that P0 is waiting for resources held by P1, P1 is waiting for resources held by P2, and so on, and Pn is waiting for resources held by P0

Circular wait implies hold and wait.

Resource allocation graph –



If there is a cycle, then there may be a deadlock (necessary but not sufficient condition).
If there is no cycle, then there is not deadlock.
If each resource type has one instance, then a cycle implies that there is a deadlock (necessary and sufficient condition).
If each resource type in a cycle has one instance, then there is a deadlock (necessary and sufficient condition).

Methods for handling deadlocks –

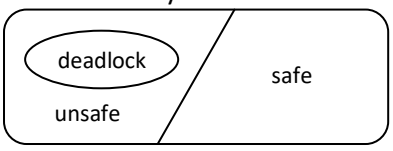
- **Deadlock prevention** - Ensure that at least one of the necessary conditions for occurrence of deadlocks cannot hold.

- **Deadlock avoidance** - Each process informs the operating system about the resources it will require in its lifetime. The operating system allocates the resources to the processes in a sequence that will not lead to a deadlock.
- **Deadlock detection and recovery**

No deadlock handling mechanism –
- deadlocks are infrequent, may be only once in a year
- deadlock handling is expensive
- e.g. – Windows, Linux, etc.

Deadlock prevention –
1. Mutual exclusion
   - Mutual exclusion is necessary for non-sharable resources, like printer and speaker.
   - Mutual exclusion can be prevented from sharable resources, like read-only files.
2. Hold and wait
   - Ensure that when a process requests for resources it is not holding some other resources.
   - Protocol 1: Request and get all the resources in the beginning.
   - Protocol 2: Release current resources before requesting other resources.
   - Disadvantages – low resource utilization, starvation of processes requiring several resources.
3. No preemption
   - If a process requests for some resources and they cannot be allocated right now, then the resources that the process is holding are preempted.
   - Resources that can be saved and later restored – registers and files.
   - Resources that cannot be preempted – printer.
4. Circular wait
   - Arrange the resource types as R1, R2, R3 … Rm.
   - Protocol 1: Request resources in increasing order.
   - Protocol 2: If a process requests for Ri, then it must release Rj for all $j \geq i$.
   - All instances of a resource type should be allocated together.
   - Prove: The protocols can prevent deadlock.
   - Disadvantages – low resource utilization, reduced throughput.

Deadlock avoidance –
- A **safe state** is one in which the system can allocate resources to each process up to the maximum in some order and still avoid deadlock.
- A system is in a **safe state** if there is a safe sequence.
- A sequence of processes <P1, P2, P3 … Pn> is safe if for each Pi the resources that Pi may still request can be satisfied by the currently available resources plus all resources held by all Pj with $j<i$.
- Pi may need to wait for one or more Pjs.



The **banker's algorithm** is used to implement deadlock avoidance.
This algorithm can be used in a bank to ensure that the bank never allocates the available money in a way that it could no longer satisfy the needs of all its clients.
A new process must declare the maximum number of instances of each resource type that it may need.
This number should not exceed the total number of instances of that resource type in the system.

When a process requests a set of resources, the system must determine whether allocating these resources will leave the system in a safe state.
If yes, then the resources may be allocated to the process.
If no, then the process must wait till other processes release enough resources.

n processes
m resource types
Data structures –

- Available[m] - currently available instances
- Max[nxm] - maximum demand
- Allocation[nxm] - currently allocated
- Need[nxm] = Max - Allocation

These data structures may vary in size with time.
For Pi: Max[i], Allocation[i], Need[i].

Safety algorithm checks if the system is in a safe state.

1. Let Work[m] and Finish[n] be vectors
   Work = Available
   Finish[i] = false, for i = 0, 1, 2 … n-1
2. Find an i such that (Finish[i] == false && Need[i] <= Work)
   If no such i exists then go to step 4
3. Work = Work + Allocation[i]
   Finish[i] = true
   Go to step 2
4. If (Finish[i] == true) for all i = 0, 1, 2 … n-1, then it is a safe state else it is an unsafe state

Time complexity = $O(mxn^2)$.

Resource-request algorithm checks if a request can be safely granted.
Pi is requesting for more resources and Request[m] be the request.

1. If Request > Need[i], then error
2. If Request > Available, then wait
3. Pretend to allocate the request
   Available = Available - Request
   Allocation[i] = Allocation[i] + Request
   Need[i] = Need[i] - Request

If the resultant state is safe then the resources are actually allocated, else values of Available, Allocation[i] and Need[i] are restored to their previous values.
Time complexity = O(m).

Exercise 1.
A system has four processes, viz. P1 to P4, and three resource types. The system is in a safe state with *Available* = $[1\ 1\ 1]$, *Max* = $\begin{bmatrix} 0 & 1 & 1 \\ 6 & 2 & 4 \\ 4 & 3 & 5 \\ 1 & 2 & 1 \end{bmatrix}$ *and Allocation* = $\begin{bmatrix} 0 & 0 & 1 \\ 4 & 2 & 3 \\ 3 & 3 & 3 \\ 0 & 2 & 1 \end{bmatrix}$. *Which of the following requests should be granted?*
(a) Request = $[1\ 0\ 1]$ *from P3.*
(b) Request = $[1\ 0\ 1]$ *from P2.*

Solution.
(a) Resource-request algorithm -

Request = [1 0 1], Available = [1 1 1] and Need = $\begin{bmatrix}0&1&0\\2&0&1\\1&0&2\\1&0&0\end{bmatrix}$

Since Request < Need[3] and Request < Available, pretend to allocate resources

Available = [0 1 0], Allocation = $\begin{bmatrix}0&0&1\\4&2&3\\4&3&4\\0&2&1\end{bmatrix}$*, and Need =* $\begin{bmatrix}0&1&0\\2&0&1\\0&0&1\\1&0&0\end{bmatrix}$

Safety algorithm -
Initially - Work: [0 1 0] Finish: [F F F F]
P1 selected - Work: [0 1 1] Finish: [T F F F]
P3 selected - Work: [4 4 5] Finish: [T F T F]
P4 selected - Work: [4 6 6] Finish: [T F T T]
P2 selected - Work: [8 8 9] Finish: [T T T T]
Safe sequence: <P1, P3, P4, P2>
Safe state: Request should be granted.

(b) Resource-request algorithm -

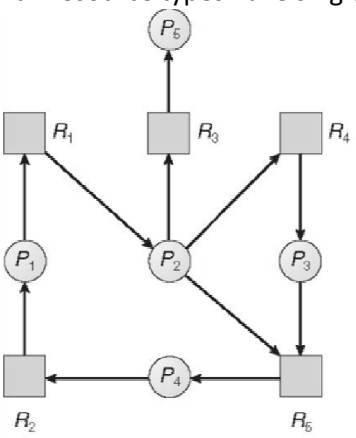
Request = [1 0 1], Available = [1 1 1] and Need = $\begin{bmatrix}0&1&0\\2&0&1\\1&0&2\\1&0&0\end{bmatrix}$

Since Request < Need[2] and Request < Available, pretend to allocate resources

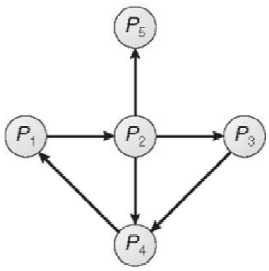
Available = [0 1 0], Allocation = $\begin{bmatrix}0&0&1\\5&2&4\\3&3&3\\0&2&1\end{bmatrix}$*, and Need =* $\begin{bmatrix}0&1&0\\1&0&0\\1&0&2\\1&0&0\end{bmatrix}$

Safety algorithm -
Initially - Work: [0 1 0] Finish: [F F F F]
P1 selected - Work: [0 1 1] Finish: [T F F F]
Unsafe state: Request should not be granted.

Deadlock detection –
If all resource types have single instances, then we can detect deadlocks using a wait-for graph.



Resource allocation graph



Wait-for graph

A cycle in the wait-for graph denotes that there is a deadlock and all the processes in the cycle are deadlocked.
Time complexity of cycle detection algorithm = $O(n^2)$.

A modification of the banker's algorithm can be used to detect deadlock in a system that has resource types with multiple instances.
Request[nxm] represents the requests.

1. Let Work[m] and Finish[n] be vectors
   Work = Available
   For i = 0, 1, 2 … n-1, if (Allocation[i] != 0) then Finish[i] = false else Finish[i] = true
2. Find an i such that (Finish[i] == false && Request[i] <= Work)
   If no such i exists, then go to step 4
3. Work = Work + Allocation[i]
   Finish[i] = true
   Go to step 2
4. If Finish[i] == false for some i, then there is a deadlock and Pi is deadlocked

Time complexity = $O(mxn^2)$.

How often to invoke deadlock detection algorithm?

- after fixed time interval
- when CPU utilization falls

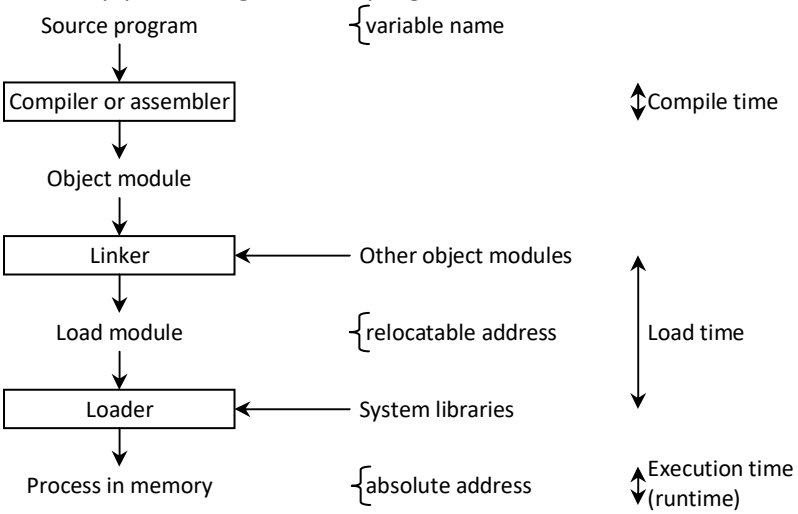
Recovery from a deadlock –

- Process termination
  - abort all deadlocked processes
  - abort one process at a time until the deadlock is broken
- Resources preemption
  - selecting a victim
    - rollback
    - starvation

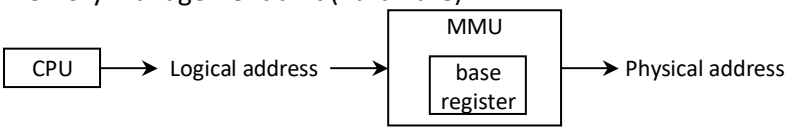
Chapter: 7.
Sections: 7.1 – 7.7.

UNIT 5
MEMORY MANAGEMENT

Multistep processing of a user program –



Memory management unit (hardware) –



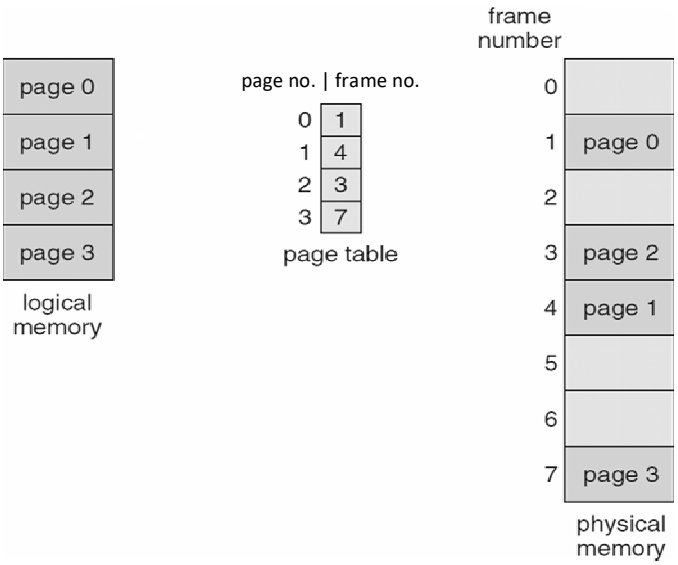
Swapping –

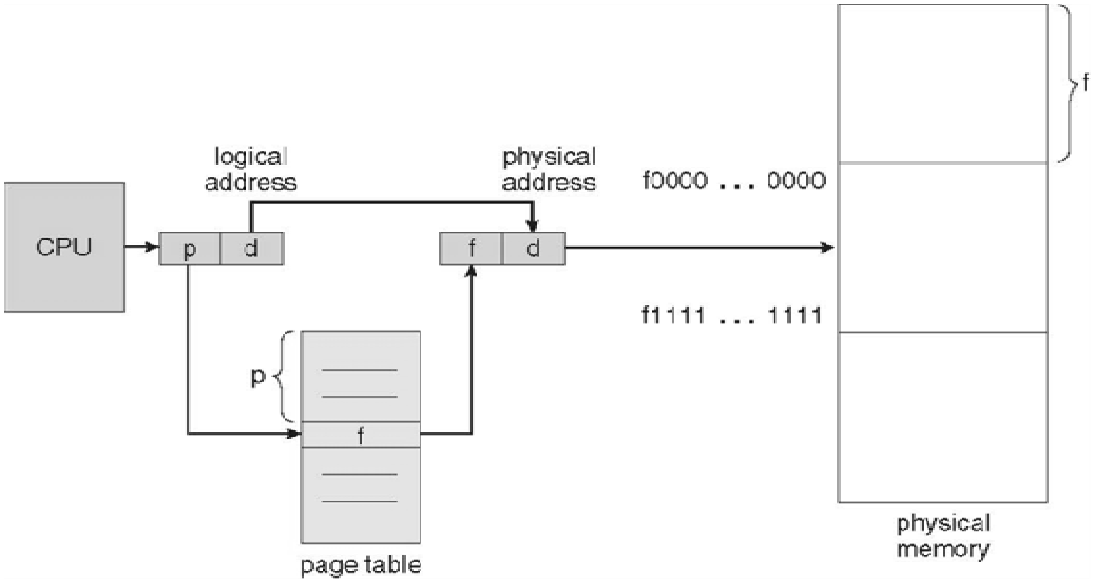
- swap-out, swap-in
- priority scheduling: roll-out, roll-in
- relocation is helpful

Contiguous memory allocation –

- Static: divide the memory into fixed-sized blocks
  - allocate one block to each process
  - number of partitions controls the degree of multiprogramming
- Dynamic: allocate available space to a process
  - Keep track of holes
  - strategies: **first-fit**, **best-fit**, **worst-fit**
  - the worst-fit strategy results in large leftover holes (reusable)
- Contiguous memory allocation leads to **external fragmentation**
  - there may be enough memory left to accommodate a process but it is scattered
  - solution: **compaction** (costly)

Paging is a non-contiguous memory allocation scheme
Page size = frame size





d = page offset or displacement
page size: 512 B to 16 MB

Advantages –
- size of process independent of page size
- no external fragmentation

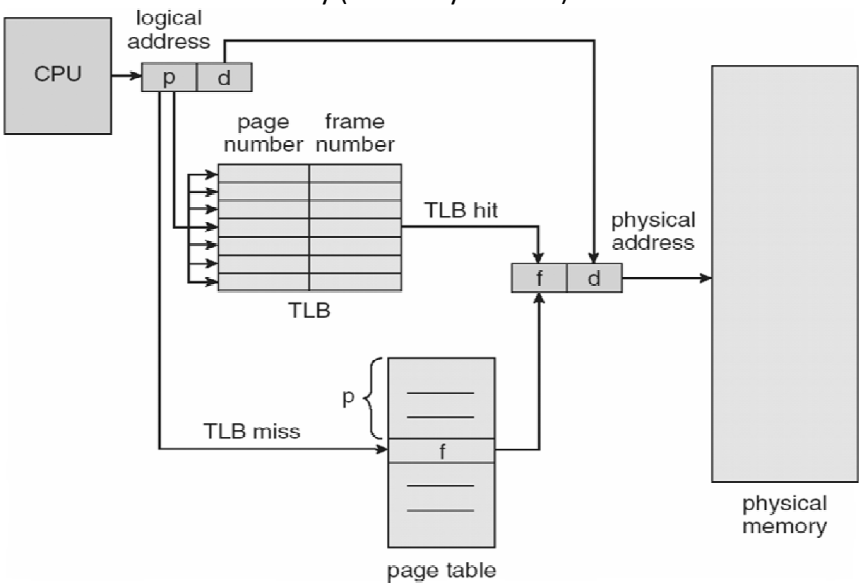
Disadvantage –
- **internal fragmentation**

Research topic –
- dynamic page size

Each process is allocated a page table.
The page table is stored in the memory.
A **page table base register** (PTBR) points to the page table.
Changing the page table requires just changing the value of the register, thus reducing the context switch time.
One read/write operation requires two memory accesses.

Transition look-aside buffer (**TLB**) –

- dedicated and fast (hardware)
- associative memory (search by content)



Effective access time = hit ratio x (TLB access time + memory access time)
+ (1 - hit ratio) x (TLB access time + 2 x memory access time)

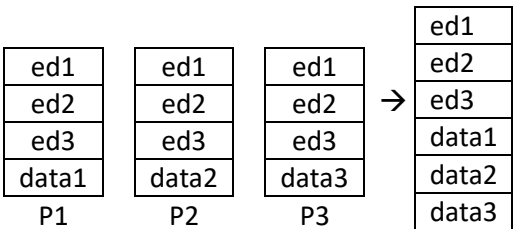
Each page table entry may have –

- a read-only bit
- a valid-invalid bit (invalid denotes that the page belongs to another process)

Shared pages

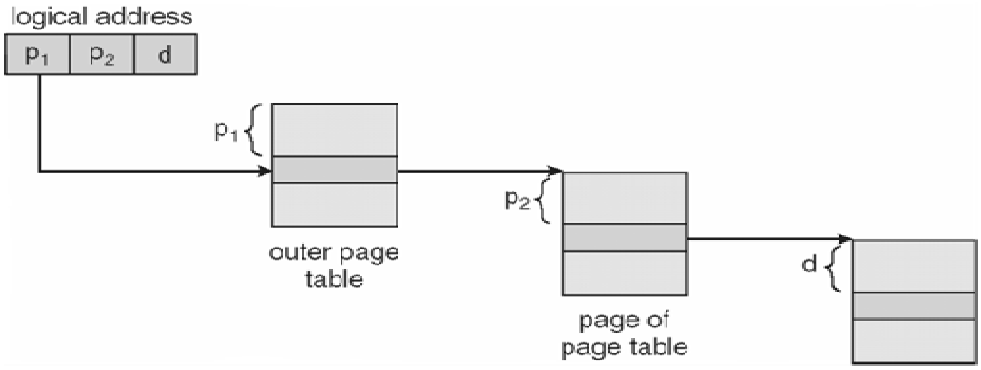
Reentrant code = pure code = non-self-modifying code = not changed during execution.

Pages common to multiple processes can be shared if they contain reentrant code.



Hierarchical page tables

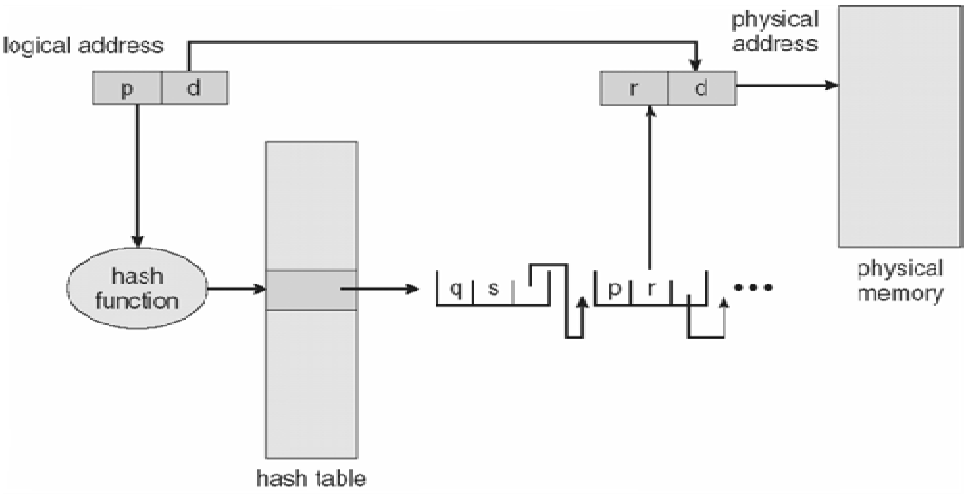
Two-level paging:



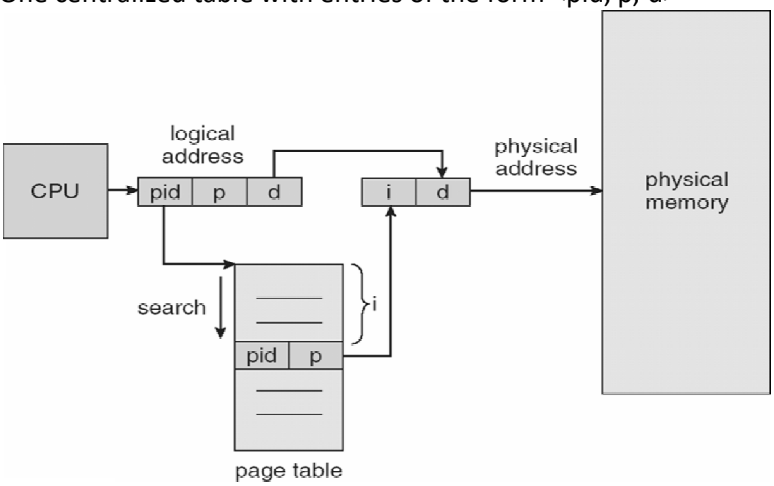
Three-level paging: <p1, p2, p3, d> - second outer page table, first outer page table, inner page table.

Too many memory accesses.

Hashed page table
A hashed value is used as a virtual page number.
Each entry in the hash table is a linked list of elements that hash to the same location (to avoid collisions).
Each element has three fields: virtual page number, frame number, pointer to the next element in the list.



Inverted page table
One centralized table with entries of the form <pid, p, d>



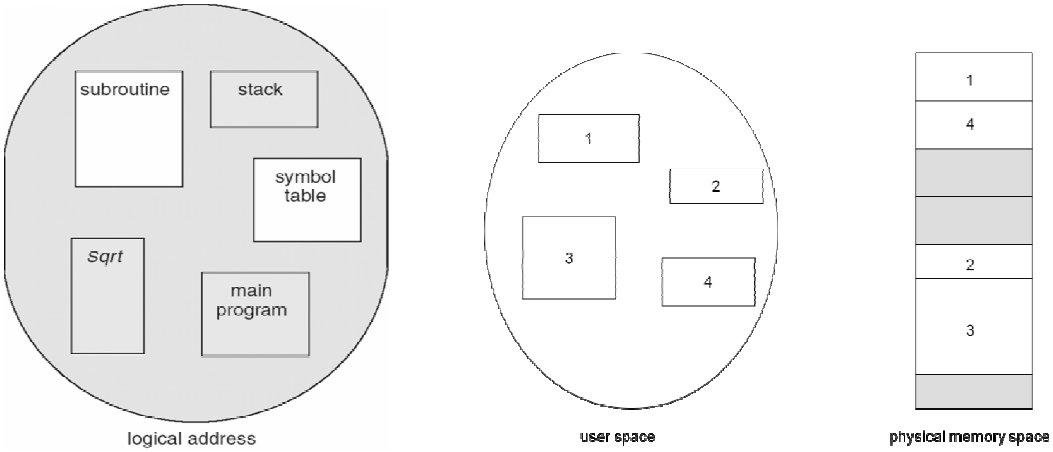
Difficult to implement shared pages.

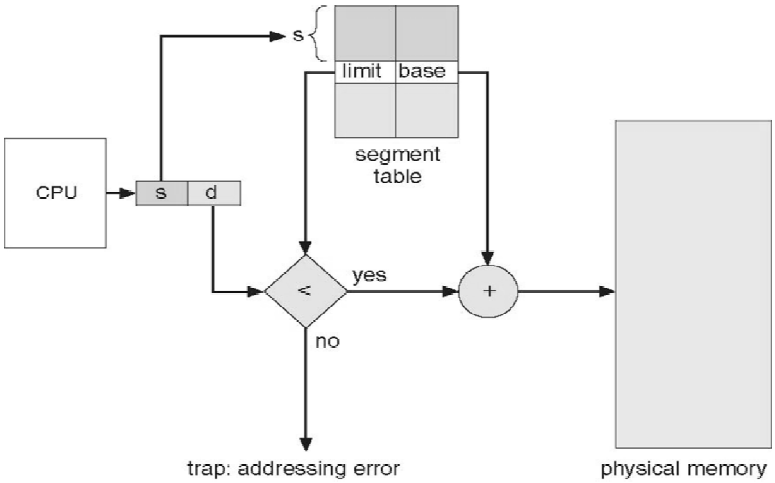
Segmentation is a memory management scheme where the linear memory is logically partitioned into segments each having unique purpose.
Address: <segment_number, offset>.

A process may have segments each storing a part of code or data.
A typical program has the following segments –

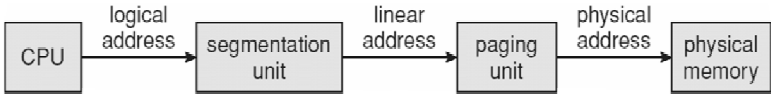
- code (CS)
- data (DS)
- stack (SS)
- extra (ES)

There is typically hardware support to use a limited number of segments at a given time.





Many processors use both segmentation and paging.



E.g. – Intel Pentium

- uses segmentation with paging
- up to 16 K segments per process
- a process can access up to 6 segments at a time
- a segment can be of up to 4 GB
- pages size can be between 4 KB and 4 MB

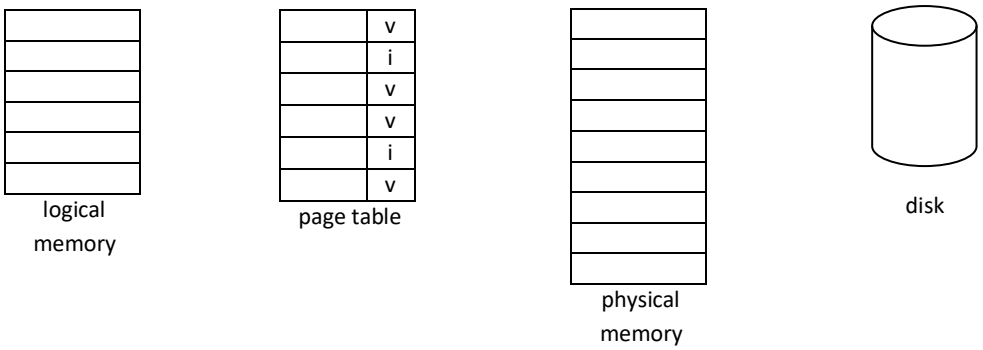
Chapter: 8.
Sections: 8.1 – 8.6.

UNIT 6
VIRTUAL MEMORY MANAGEMENT

Virtual memory is a technique that allows the execution of processes that are not completely in memory.

This allows execution of programs larger than the memory.
This allows higher degree of multiprogramming.

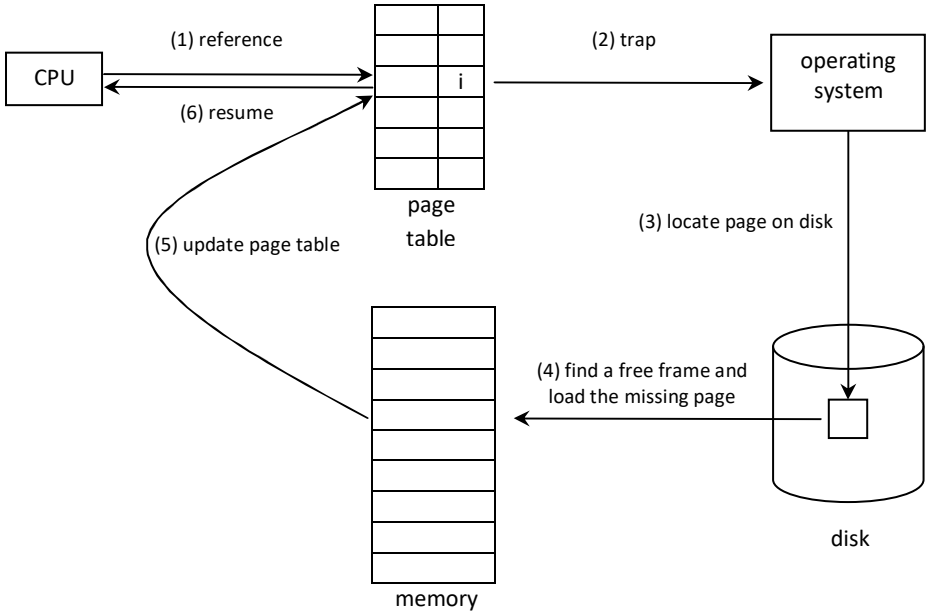
Demand paging is an approach to implement virtual memory.
Pages are loaded into the memory only when the CPU wants to access them.
Pages that the CPU does not want to access are not loaded.
We use a program called lazy swapper; also known as pager.



v (valid): page is in memory
i (invalid): page is not in memory

If the CPU wants to access a memory-resident page, then the execution continues normally.
If the CPU wants to access a page that is not in memory, then a **page fault** trap occurs.
(trap: highest priority non-maskable external-hardware interrupt)

Steps in handling a page fault –



Pure demand paging is an extreme case of demand paging where the execution of a process is started with none of its pages in memory.
Starting is fast and response time is low.
Takes advantage of locality of references for both code and data.

Hardware support for virtual memory (already present) –
- page table
- disk

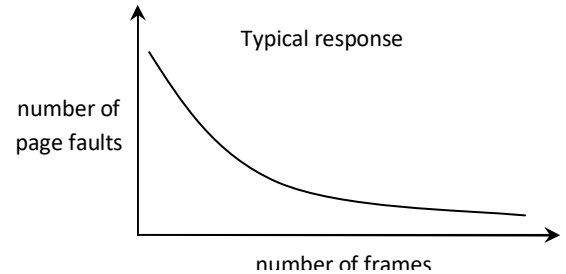
Let, page fault ratio = p.
Typically, $0 < p < 1$ with $p \rightarrow 0$ actually.
Effective access time = (1-p) x memory access time
+ p x page fault service time

Page replacement is the process of replacing one page by another in the memory when there is no free frame.
How to load a page in memory when there is no free frame?
1. Save the content of the page currently in memory on disk.
2. Load new page.
3. Update page table.

Avoid saving unmodified pages –
- use modify bit / dirty bit

We require **page replacement algorithms** and **frame allocation algorithms**.

We evaluate an algorithm by running it on a particular string of memory references and calculating the number of page faults.
This string of memory reference is called a **reference string** and is artificially generated by a random number generator.
E.g. – 0, 1, 0, 3, 5, 6, 1, 0, …



Page replacement algorithms –
1. **FIFO page replacement algorithm**
   - no need to note time, only a queue is required
   - simple, but performance not good
   - shows Belady's anomaly
   - e.g. –
     - reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1.
     - frames = 3.
     - page faults = 15.

Belady's anomaly
Number of page faults increases with number of frames allocated.

- e.g. –
    - reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.
    - frames = 3, page faults = 9.
    - frames = 4, page faults = 10.

2. **Optimal page replacement algorithm**

- replace the page that will not be used for the longest period of time
- lowest page fault rate, no Belady's anomaly
- needs future knowledge of reference string, hence, impossible to implement
- used for comparison
- e.g. (same) – 9 page faults.

3. **Least-recently used (LRU) page replacement algorithm**

- replace the least-recently used page
- no Belady's anomaly
- implemented using counters or a stack
- e.g. (same) – 12 page faults.

Implementation of LRU page replacement algorithm using counters –

- the page table has a 'time of use' field
- there is a logical clock which is incremented after every memory reference
- when a reference is made to a page, its 'time of use' is set to the current reading of the logical clock
- the LRU page has the minimum 'time of use' value, search and replace it
- logical clock must not overflow

Implementation of LRU page replacement algorithm using a stack –

- maintain a stack of page numbers referenced
- when a page is referenced, remove it from the stack and put it on the top
- the LRU page is at the bottom, no need to search
- suitable for software/microcode implementation

LRU page replacement algorithm requires substantial hardware support.

A **stack algorithm** does not suffer from Belady's anomaly.
A page replacement algorithm is a stack algorithm if we can show that the pages in memory for n frames is always a subset of pages in memory that will be there in memory for (n+1) frames.
For LRU page replacement algorithm –

- for n frames: n most recently used pages are in memory
- for (n+1) frames: (n+1) most recently used pages are in memory
- hence, it is a stack algorithm

4. **LRU approximation page replacement algorithms**

- most systems do not provide support necessary for LRU page replacement algorithm
- however, in most systems there is a **reference bit** for each page table entry
- initially, the reference bit is cleared to 0
- if the page is referenced, then the reference bit is set to 1

4a. **Additional-reference-bits page replacement algorithm**
- we have a 8-bit shift register for each page table entry
- after fixed time intervals, the shift registers are right-shifted with the reference bit filling the msb
- the LRU page will have the minimum shift register value
- shift register vale may not be unique – swap out all tied pages or use FIFO to break the tie

4b. **Second chance page replacement algorithm**
- it is like FIFO
- however, if reference bit == 1, do not replace page and clear the reference bit
- frequently used pages will never be replaced
- e.g. (same) – 11 page faults.

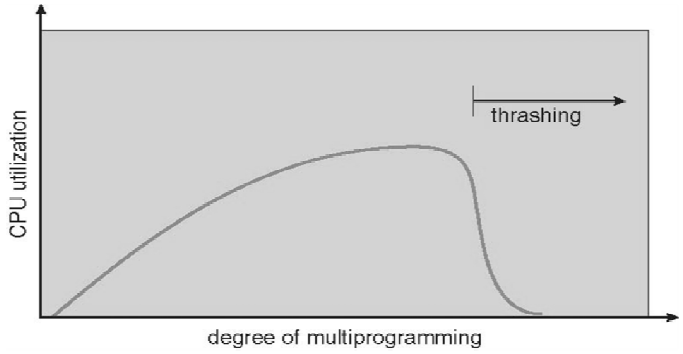
4c. **Enhanced second chance page replacement algorithm**
- principle: modified pages are not good candidate for replacement
- use (reference bit, modify bit) as an ordered pair
- classes: (0,0), (0,1), (1,0) and (1,1)
- replace a page of the lowest non-empty class by FIFO
- may require multiple iterations of the queue

5. **Counting based page replacement algorithms**
- a counter for each page
- **least frequently used (LFU) page replacement algorithm**
- **most frequently used (MFU) page replacement algorithm**
- expensive but performance not good

Allocation of frames –
Minimum number of frames – otherwise too many page faults
Frame allocation algorithms –
- **equal allocation**
- **proportional allocation** (proportional to the virtual memory of the process)

Local page replacement – a process can control its page fault rate
Global page replacement – better throughput, commonly used

Thrashing is a situation when a system is spending more time in servicing page faults than executing.
If a process does not have the number of frames it needs to store the pages in active use, then it will page fault frequently.
Since all the pages in memory are in active use, the page that will be replaced will be needed again.
This will bring down the CPU utilization and the system will load more processes in the memory in order to increase the degree of multiprogramming.
This will trigger a chain reaction of page faults.
This situation when the CPU utilization diminishes due to high paging activity is called thrashing.

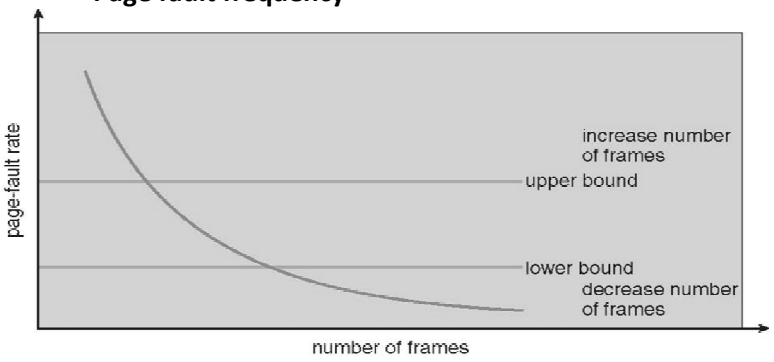


Avoiding thrashing –

- **Working-set model**
  - based on locality of references
  - working set of a process = pages referenced in Δ most recent references
  - e.g. –

    Δ = 10.

    reference string: 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 3 4 3 3 4 4 4 3 3.
  - WSS_i = working-set size of P_i
  - total demand of frames, $D = \Sigma WSS_i$
  - let, m = number of frames
  - if D > m, then there will be thrashing
  - we swap out one or more processes
  - this keeps degree of multiprogramming as high as possible and optimizes CPU utilization
  - keeping track of working-sets is difficult
- **Page fault frequency**



Chapter: 9.
Sections: 9.1, 9.2 and 9.4 – 9.6.

UNIT 7
STORAGE MANAGEMENT

Hard disks are the most common on-line storage medium.

The operating system abstracts from the physical properties of its storage devices to define a logical storage, the **file**.
A file is a named collection of related information that is recorded on secondary storage.
A file is the smallest allotment of logical secondary storage.
Data cannot be written to secondary storage unless they are within a file.
Files store both code and data.
File formats – magic number.

To provide efficient and convenient access to the disk, the operating system imposes a **file system** to allow data to be stored, located and retrieved easily.
Design issues –

- how the file system should look to the user?
- how to map logical files onto the physical disk?

The file system is typically implemented as a multi-leveled system.

Application programs	
↓	
Logical file system	[Manages metadata]
↓	logical block address
File organization module	[Allocation management, free space management]
↓	physical block address
Basic file system	
↓	high-level generic command
I/o control	[Device driver, interrupt]
↓	low-level device specific command
Disk	

Example of disk address: drive 1, track 2, sector 1.

A **directory** is a logical construct representing a set of files and subdirectories.

Implementing a file system –

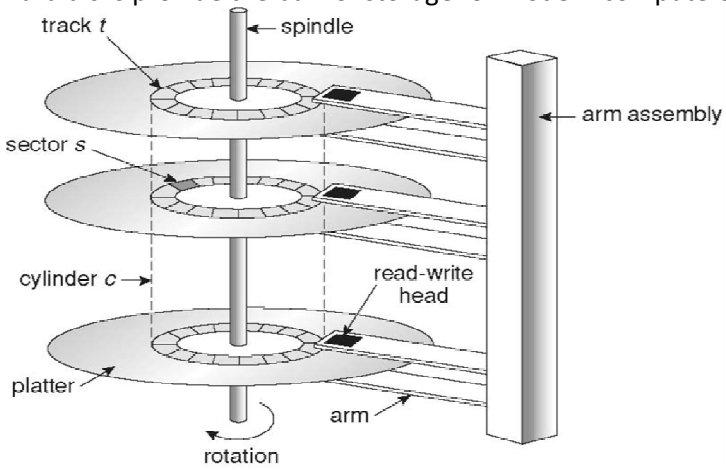
- **boot control block** (volume) - information needed to boot an operating system from that volume, if there is one
- **volume control block** (volume) - information about the volume
- **master file table** (volume) – information about the directory structure
- **file control block** (file) - information about the file

Allocation methods –

- contiguous
- linked
- indexed

Free space management – maintain a free-space-list

Hard disks provide the bulk of storage for modern computers.



Each disk platter has a flat circular shape.
Bits are stored magnetically on the platters.
A read-write head flies over a platter.
The arm moves the heads together.
The surface of the platter is divided into circular **tracks**.
Tracks are subdivided into **sectors**.
The set of tracks at one arm position consist a **cylinder**.
There are thousands of cylinders in a disk and hundreds of sectors in a track.
The outermost cylinder is denoted cylinder 0.
A motor rotates the disk at a high speed.

Transfer rate: rate at which data is transferred between the bus and the disk.
Seek time: time taken to move the arm to the cylinder.
Rotational latency: time taken to rotate the disk so that the head comes over the sector.
Positioning time (random access time) = seek time + rotational latency.
Performance of a disk depends on transfer rate and positioning time.

Disks can be addressed as a large one-dimensional array of **logical blocks**.
A block is the smallest unit of transfer.
The one-dimensional array of logical blocks has to be mapped onto the physical sectors.
Sequence –

- cylinder (outermost → innermost)
- track (top → bottom)
- sector

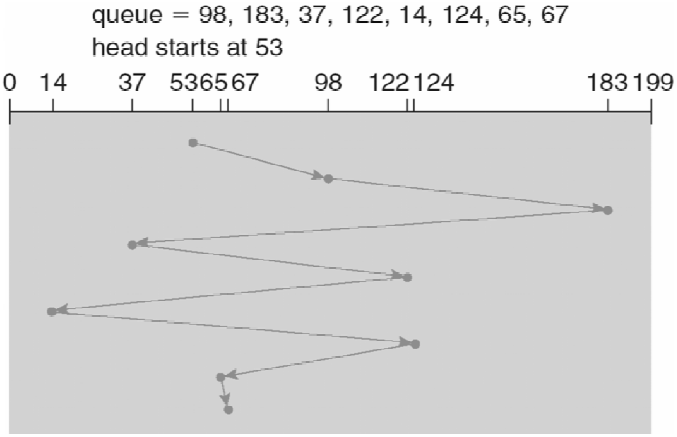
Storage –

- uniform bit density - more sectors on outer tracks
- non-uniform bit density - same number of sectors on all tracks

Bandwidth of a disk = number of bytes transferred / time elapsed between first request and last service
Disk scheduling – try to improve access time and bandwidth

Disk scheduling algorithms –

- **First come first served (FCFS) scheduling**
- **Shortest-seek-time-first (SSTF) scheduling**
- **SCAN scheduling (elevator algorithm)**
- **C-SCAN scheduling**
- **LOOK scheduling**
- **C-LOOK scheduling**



Disk formatting –

- physical formatting or low-level formatting
    - fills the disk with a special data structure for each sector
    - the data structure has header, data area and trailer
    - contains sector number, error correcting code, etc.
- logical formatting
    - creates a file system

To increase efficiency, some file systems groups blocks to form chunks.
Some operating systems allows some programs to perform raw i/o, e.g. – large databases.

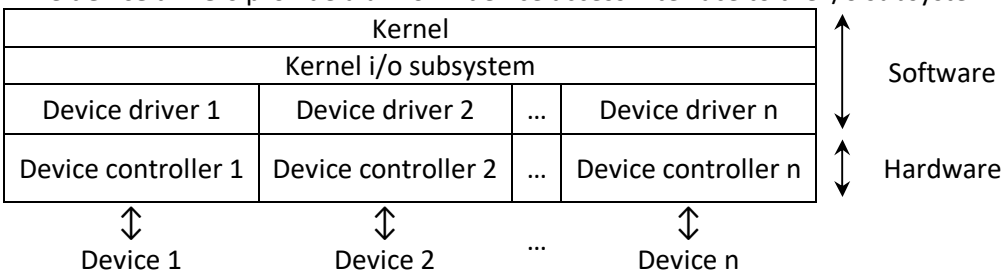
Bad blocks are formed when some sectors become defective.
Logical formatting informs the file system to ignore them.

> Chapters: 10 to 12.
> Sections: 10.1, 10.3, 11.1, 11.2, 11.4, 11.5, 12.1, 12.2, 12.4 and 12.5.

UNIT 8
(A) I/O MANAGEMENT AND (B) SYSTEM PROTECTION AND SECURITY

Two main responsibilities of a computer: processing and i/o.
Some processes perform more i/o and minimal processing.
I/o devices vary widely in function and speed.

The **i/o subsystem** of the kernel separates the rest of the kernel from the complexities of managing the i/o devices.
The **device drivers** provide a uniform device access interface to the i/o subsystem.



Types of devices –
- storage devices (disk, USB drives)
- transmission devices (modem)
- human-interface devices (keyboard, mouse, display)
- specialized devices (microscope, telescope, case of flight control computer)

Hardware –
- **port** - a connection point where a device is connected
- **bus** - set of wires to transfer data and control information
- **controller** - controls a port or a device

An i/o port consists of –
- status register
- control register
- data-in register
- data-out register

Modes of performing i/o –
- polling (busy-waiting)
- interrupts
- DMA
    - burst mode
    - cycle stealing mode
    - direct virtual memory access (DVMA)

Complementary devices –
- teletype = keyboard + display
- photocopier = scanner + printer

Virtual devices –
- PDF printer

Properties of devices –
- character-stream versus block
- sequential access versus random access
- synchronous versus asynchronous
- sharable versus dedicated
- speed
- read-only, write-only, read-write

Services provided by the i/o subsystem –
- i/o scheduling
- buffering
- caching
- spooling (a spool is a buffer that holds the output of a device, like a printer, that cannot accept interleaved data streams)
- error handling
- i/o protection

Protection - from internal and genuine errors
Security - from external and intentional threats
Operating systems implement protection and provide support for security.

Object - hardware or software objects
Domain - specification of resources a process can access

Access matrix

Object / Domain	O1	O2	O3	O4
D1	R			R
D2		RW	R	RW
D3	R		E	

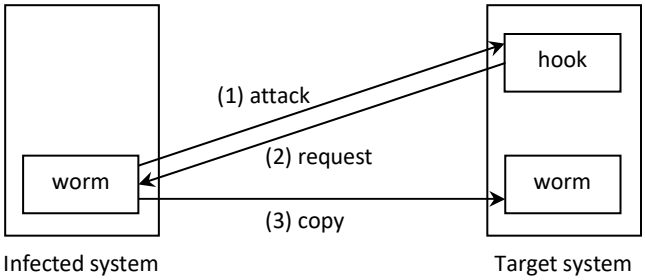
Switch - switch domain
Copy (*)
- transfer
- copy limited
- copy unlimited

Object / Domain	O1	O2	O3	O4	D1	D2	D3
D1	R			R			
D2		RW*	R	RW	S		S
D3	R		E*		S	S	

Implementation of access matrix –
(Access matrix is sparse)
- Global table
  - ordered triples <domain, object, rights>
- **Access lists of objects**
  - a column of access matrix
  - blank entities may be discarded

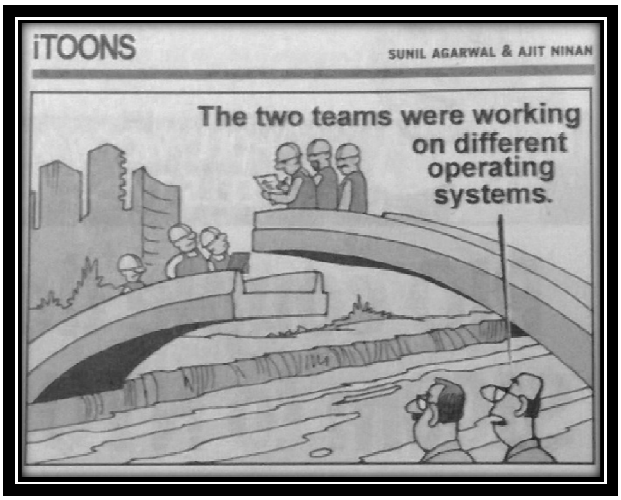
- **Capability lists of domains**
  - a row of access matrix
  - a process should not be able to modify its capability list

Threats –
- Trojan horse
  - attractive and harmless cover program
  - harmful hidden program
  - used as virus carrier
- Trap door
  - hidden door
- Logic bomb
  - initiate attack under specific situation
- Virus
  - self-replicating and infectious
  - modify and destroy files
  - Linux is not susceptible to virus because it does not allow modifying executable programs
- Worm
  - infection programs spread through networks



Types of computer viruses –
- file/parasitic - appends itself to a file
- boot/memory - infects the boot sector
- macro - written in high-level language like VB and affects MS Office files
- source code - searches and modifies source codes
- polymorphic - changes on copying each time
- encrypted - encrypted virus + decrypting code
- stealth - avoids detection by modifying parts of system that can be used to detect it, like the read system call
- tunneling - installs itself in the interrupt service routines and device drivers
- multipartite - infects multiple parts of the system

Chapters: 13, 17 and 18.
Sections: 13.1 – 13.4, 17.4, 17.5, 18.2 and 18.3.


iTOONS
SUNIL AGARWAL & AJIT NINAN
The two teams were working on different operating systems.

Last revised on 16 February 2017.