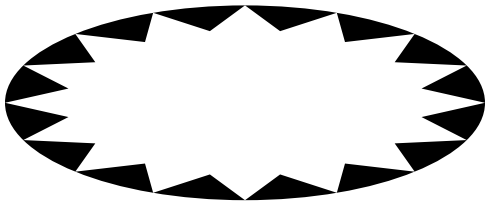
COMPUTER SCIENCE DEPARTMENT

COURSE TITLE:

INTRODUCTION TO OPERATING SYSTEMS CONCEPTS



Mob. 0724850225

benardosero@gmail.com

Flexible Learning

INSTRUCTIONAL MATERIAL FOR DISTANCE LEARNING STUDENTS

INTRODUCTION TO OPERATING SYSTEMS

INTRODUCTION TO OPERATING SYSTEMS CONCEPTS.

CONTENTS

COURSE TOPICS

UNIT 1	**INTRODUCTION TO OPERATING SYSTEMS.** **Topic 1: INTRODUCTION.** **Topic 2: EXAMPLES OF OPERATING SYSTEM.** **Topic 3: Abstract view of Computer System.** **Topic 4: Functions of Operating Systems.**
UNIT 2	**HISTORY OF OPERATING SYSTEM.** **Topic 1: Batch System.** **Topic 2: Multi-programming Systems.** **Topic 3: Time Sharing Systems.** **Topic 4: Real time, Multiprocessor, Distributed/Networked systems.**
UNIT 3	**MEMORY MANAGEMENT SCHEMES.** **Topic 1: Contiguous, Real Memory Management.** **Topic 2: Non- Contiguous Memory Management.** **Topic 3: Virtual Memory management Systems.**
UNIT 4	**PROCESSOR MANAGER** **Topic 1: Single User Systems.** **Topic 2: Multi-programming Systems.** **Topic 3: Process Scheduling.**
UNIT 5	**INTURRUPT MANAGEMENT** **Topic 1: Interrupts.** **Topic 2: Deadlocks.** **Topic 3: Resources.** **Topic 4: Deadlock Prevention.**
UNIT 6	**DEVICE MANAGEMENT.** **Topic 1 : System Devices** **Topic 2: Dedicated Devices.** **Topic 3: Virtual Devices.**

Unit 7	SYNCHRONISATION AND INTERPROCESS COMMUNICATION
Unit 8	FILE AND DISK MANAGEMENT

Course units

Unit 1 Introduction to operating Systems
Unit 2 History of Operating Systems.
Unit 3 Memory Management.
Unit 4 Process/Job Scheduling.
Unit 5 Device Management.
Unit 6 Priority Scheduling
Unit 7 SYNCHRONISATION AND INTERPROCESS COMMUNICATION
Unit 8 FILE AND DISK MANAGEMENT

INTRODUCTION TO OPERATING SYSTEMS.

COURSE OUTLINE

Course Objectives:

At the end of the course, the student should be able to:

- Understand the concept of an operating system through the historical developments.
- Understand and appreciate the role of operating systems in computer systems.
- Grasp the concepts of process, synchronization and communication and apply them to practical problems.
- Understand resource management in computer systems and the subsequent application in memory, file and disk management.
- Understand the underlying principles of operating system design.

Course Content:

1) Introduction to Operating System
2) History Of Operating Systems
3) Memory Management
4) Process/Job Scheduling

5) Device Management
6) Priority Scheduling
7) The file System Interface

Teaching Methods:

- Class lectures which consist of explanation of the functioning of an operating system.
- Regular CATs and assignments.
- Group discussion and active participation in class.

COURSE OBJECTIVES

The course objectives as noted before are to include understanding of the different types of Operating Systems and acquisition of skills to design an Operating System using familiar technology.

APPROACH

Each the above units are divided into a number of topics. At the beginning of each topic you will find a topic title followed by an introduction, objectives, and a content list within each topic. You will find a number of activities or learning points to stimulate your thinking. You should complete all such tasks.

At the end of each topic, you will find a series of self- assessment questions. These are meant to assist you evaluate your understanding of concepts presented in the topic. You should ensure that you understand and apply everything that is being discussed. It is suggested that you make sure that you:

- Complete each topic in each unit at a time before proceeding to the next one
- Consult relevant books and other information sources on each topic, especially where you find that a concept is difficult to understand.
- Make notes to simplify what you are studying
- Complete activities and self-assessment questions as you make progress

Though this module will be your main reading and study source, we however recommend and encourage you to read additional books as you will find that you need more information on a specific concept or topic to understand it fully.

Overview

Welcome to unit 1 of this core module. This is the first unit in a series of Six units that we shall be discussing. In unit one, we are going to discuss concepts and terms related to Operating System as an important aspect of information Technology. As noted above, it cannot be overemphasized that we are living in a post-industrial age which is also an age of information technology revolution. This course should enable you to acquire skills for acquiring, managing, structuring and designing Operating Systems, effectively and efficiently.

Dedication

This Academic paper is dedicated to my Mum and Dad.

Unit 1: Topic 1: INTRODUCTION

Operating System Definition

It is an interface between the user and hardware.

It is a system software which acts as an Interface between the user and the computer hardware. It enables the user to communicate and manage computer hardware and other computer software.

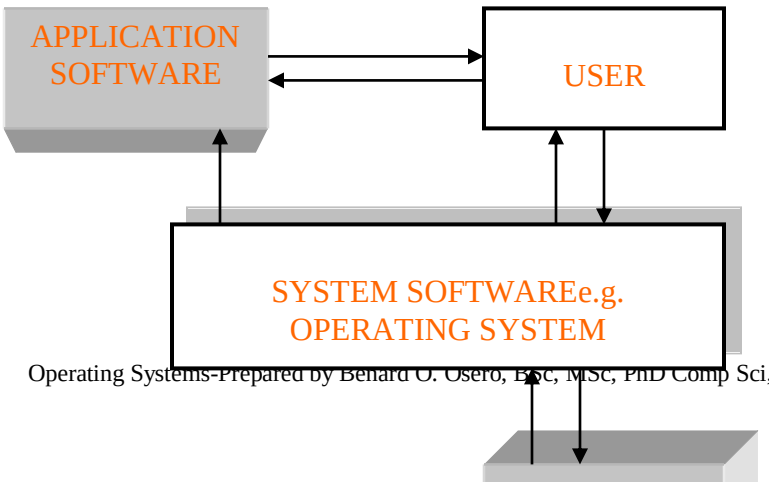
It controls:

i. Files (update, create, access rights).
ii. Devices (I/O devices, hard-disk, memory etc.).
iii. Processing-time in a multi user environment.
iv. Users-On how to use the system.

1.1 Examples of Operating System

i. Windows family (windows 95/98, 2000, XP professional etc).
ii. Linux, UNIX.
iii. DOS.
iv. Novell Netware.
v. Sun Solaris.
vi. Macintosh.

1.2 Abstract View of Computer System



Explanation of above diagram

- Hardware (CPU, Memory, I/O devices). They provide basic computing resources.
- Application Software (MS- Office, packaging Dbase)- Use hardware to resolve user problems.
- User (People, Machine, other computers).
- System Software (Operating System)-It co-ordinates use of H/W among various applications with different users

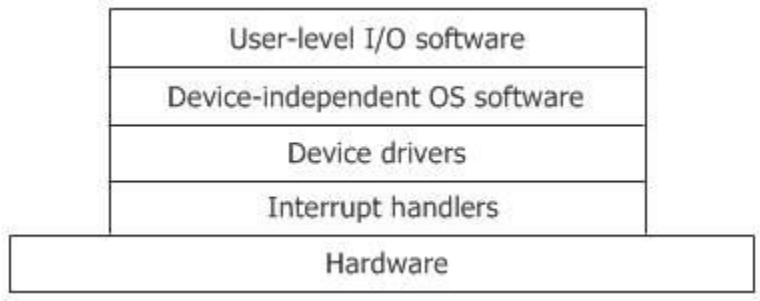
OS Input/output Software Layers

Basically, input/output software organized in the following four layers:

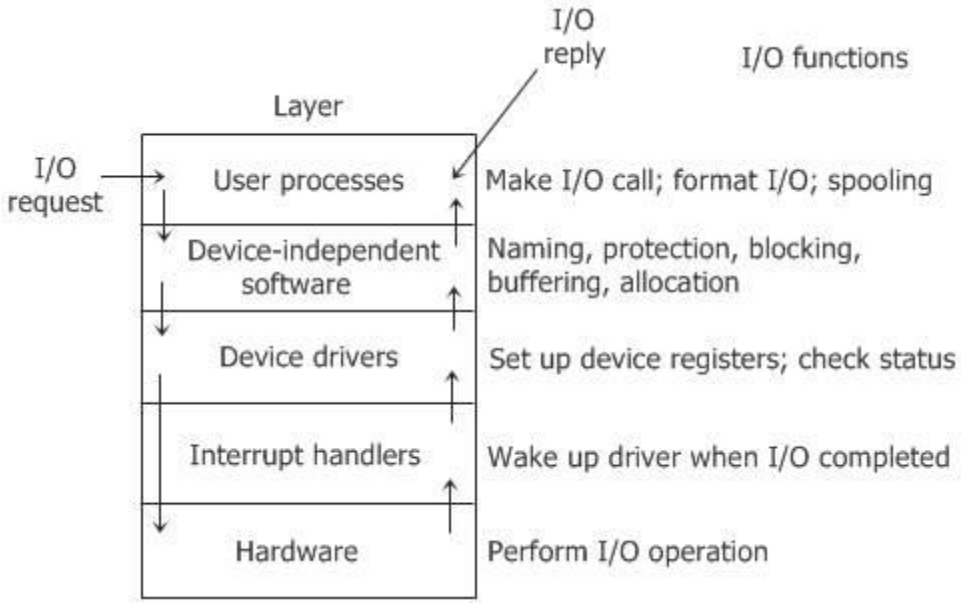
- Interrupt handlers
- Device drivers
- Device-independent input/output software
- User-space input/output software

In every input/output software, each of the above given four layer has a well-defined function to perform and a well-defined interface to the adjacent layers.

The figure given below shows all the layers along with hardware of the input/output software system.



Here is another figure shows all the layers of the input/output software system along with their principal functions.



Now let's describe briefly, all the four input/output software layers that are listed above.

Interrupt Handlers

Whenever the interrupt occurs, then the interrupt procedure does whatever it has to in order to handle the interrupt.

Device Drivers

Basically, device drivers is a device-specific code just for controlling the input/output device that are attached to the computer system.

Device-Independent Input/output Software

In some of the input/output software is device specific, and other parts of that input/output software are device-independent.

The exact boundary between the device-independent software and drivers is device dependent, just because of that some functions that could be done in a device-independent way sometime be done in the drivers, for efficiency or any other reasons.

functions that are done in the device-independent software:

- Uniform interfacing for device drivers
- Buffering
- Error reporting
- Allocating and releasing dedicated devices
- Providing a device-independent block size

User-Space Input/output Software

Most of the input/output software is within the operating system (OS), and some small part of that input/output software consists of libraries that are linked with the user programs and even whole programs running outside the kernel.

1.3 Functions of Operating System

i. It acts as a resource manager.
   - Hard-disk Space.
   - Memory Space.
   - CPU-time.
   - I/O devices

ii. It acts as a virtual Machine-Operating System hides the technicalities of processing from the user.

iii. Memory Management.
   - It allocates processes and data to memory space.
   - It keeps track of memory usage in a multi-user environment.
   - It preserves space in main memory to be used by itself.

iv. Process Management (Job Scheduler).
   - Schedules jobs as they enter into the system i.e. Job Queue.
   - Scheduler decides which process gets the CPU and for how long.

v. Device Management
   - It allocates devices to processes i.e. Keyboard, printer, monitor etc.

vi. File Management.
   - Keeps track of all files.
   - Enforces access restrictions to files.
   - Enables backup and recovery of files.

vii. It enforces protection policies, access rights/Access control onto the computer system.

viii. Maintains the internal clock.

Revision Questions.

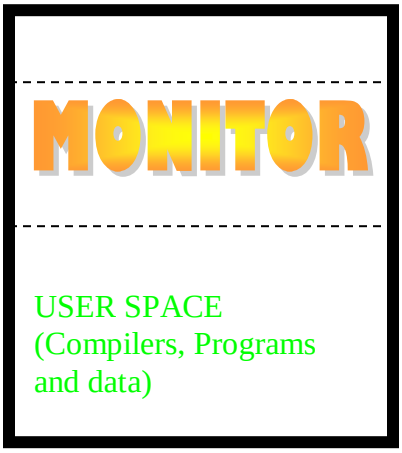
1. What is an Operating System?

2. Discuss the various parts of a computer system; explain the various functions of each of these parts.
3. Discuss the various advantages of an Operating System.

Unit 2: Topic 2: History of Operating System.

2.1 Batch System (Mid 1950-Mid 1960).

- Many jobs are bundled together with necessary instructions.
- All jobs are processed in sequence with no intervention.
- Jobs of similar nature are bundled together for easy processing (e.g. FORTRAN jobs, Cjobs, C++ jobs etc).



MONITOR- A system software used to interpret and carry out instructions of batch jobs.
Magnetic tapes and drums to store intermediate data and compiled programs.

Advantages of batch systems

i. Much of the work of the operator is moved to the computer
ii. Increased performance i.e. a job starts immediately after the other one is completed.

Disadvantages of batch systems

i. Large turn around time.
ii. Difficult to debug programs
iii. One batch job can affect pending jobs.
iv. A job could also corrupt the monitor and affect pending jobs-i.e. enter an infinite loop.

Major problem with batched Systems is that there is no protection Scheme.

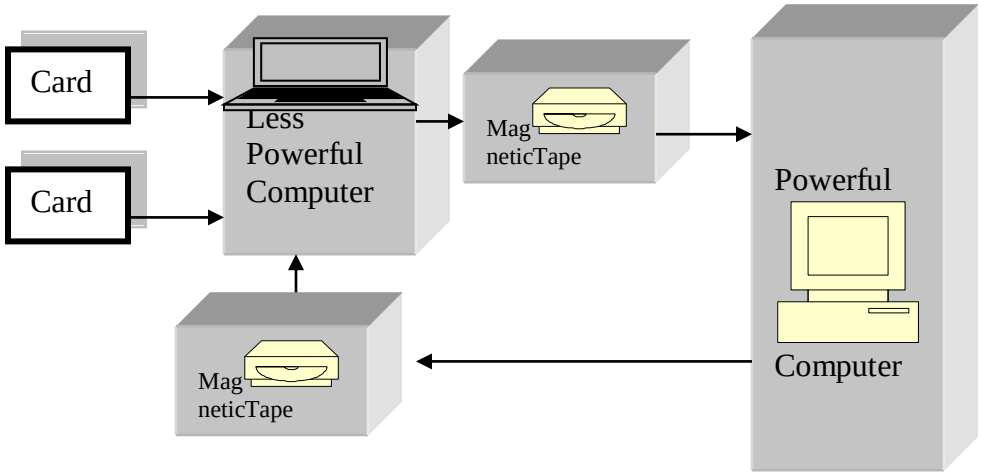
SOLUTION

i. Monitor was separated from user programs.
ii. Hard-ware was changed to allow the computer system operate in two modes: -
   - Monitor mode
   - User programs mode.
iii. Timer was added to the system to prevent-Infinite i.e. Maximum execution time was allocated to the job.

Spooling-Ability of I/O operations to take place while the CPU is processing jobs.
Problem with earlier Batch systems is that CPU is idle when the computer is reading a deck of cards.

Solution.

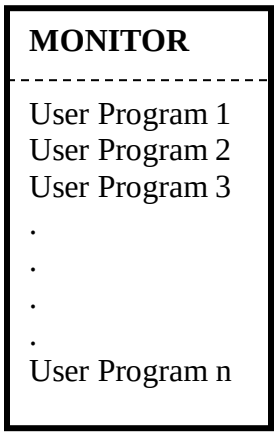
i. Less powerful computers were used to read decks of cards onto a magnetic tape. The tape would then contain many batched jobs.
ii. This tape was then loaded to the main computer to execute the jobs i.e the tape is much faster than cards.
iii. Output would then be stored on another tape, then removed and loaded to less powerful computers for printing.



SPOOL- Simultaneous peripheral operations online.

2.2 Multi-Programming Systems (1960s- present).

- Several jobs are loaded into memory at once.
- Each job is executed by the CPU for a specified period of time.
- The system ensures effective use of resources (CPU, Memory, Peripheral devices).



The Monitor was responsible for the following functions.

i. Spool operations
ii. Perform I/O operations.
iii. Switch between jobs.
iv. Ensure proper protection of job data.

2.3 Time Sharing Systems (1970s- present)

- Multiple terminals are connected to the computer system.
- Allows multiple users.
- Multiple jobs are executed by the CPU switching between them, but the time between switches is unnoticed.

2.4 Real time, multiprocessor, distributed/Networked Systems

Real time Systems are systems where there are rigid time requirements on the operation e.g. missile systems, office monitoring systems.

Multiprocessor Systems are computer systems with more than one CPU

Two types of Multiprocessor Systems

i. Shared Memory Multiprocessor-Many CPUs sharing a common memory.
ii. Distributed Multiprocessors systems-Each CPU has its own memory.

Distributed Systems-Multiple computers are networked not necessarily in the same location. Workloads are shared among different computers which might be running in a different operating systems.

Networked Systems-Multiple computers are connected usually sharing a common operating system.

Revision Questions.

i. Discuss in details the various historical developments surrounding the operating systems.
ii. What are batch systems? What are the advantages and disadvantages of batch systems?
iii. Explain in details the meaning of multi-programming systems.
iv. Discuss the various types of multiprocessor systems.

Unit 3: Topic 3: MEMORY MANAGEMENT SCHEMES.

3.1 CONTIGUOS, REAL MEMORY MANAGEMENT

3.1.1 Single Contiguous.

- All ready processes are held on disk.
- A process is then brought into main memory.
- When a process is complete it is swapped back to disk.
- Next high-priority process is then loaded into memory.
- There is only one process in memory at a time.
- Process is given contiguous memory locations.

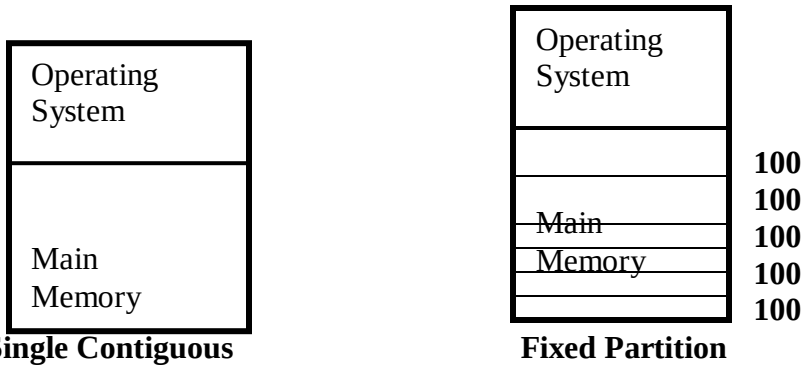
Disadvantages

i. The entire process must be loaded into memory, if the process size is larger than memory then the process can't be executed unless the memory size is increased or the process is modified.
ii. Wastes memory space especially for small processes.
iii. No support for multiprogramming.
iv. CPU is idle most of the time.

Advantages

i. Fast access time.

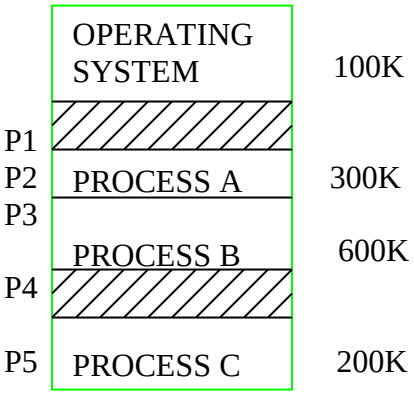
Difference between Single Contiguous and Fixed Memory Partitions.



Single Contiguous **Fixed Partition**

3.1.2 Fixed Partitioned Memory.

- Allows several jobs in memory
- Main memory had fixed partitions created i.e. one partition for each job.
- No job could enter another job's partition.
- Programs had to be stored in contiguous locations from beginning to end of execution.
- Operating System manager keeps a memory table to allocate space to jobs.



PARTITION			
ID	**Starting Address**	**Size**	**Status**
0	**0K**	**100K**	**BUSY**
1	**100K**	**200K**	**BUSY**
2	**300K**	**100K**	**FREE**
3	**400K**	**300K**	**BUSY**
4	**700K**	**100K**	**FREE**
5	**800K**	**200K**	**FREE**

Limitations

i. Internal fragmentation- Small process occupies a large memory space.
ii. External fragmentation- There are free partitions but they are not contiguous to accommodate a process.

3.1.3 Variable partitions memory Management

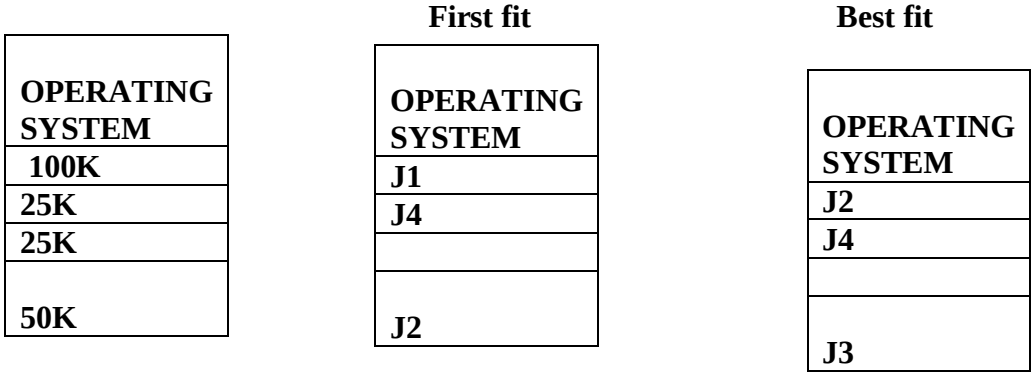
3.1.3.1Dynamic Partition

- Memory is kept in contiguous blocks.
- Jobs are given the size of memory they request i.e. contiguous blocks can be combined to form a larger hole or a larger hole can be partitioned into smaller holes.
- Memory is not wasted for the first jobs.

Strategies for selecting free blocks

i. **Best-fit-** Allocate the smallest block that is big enough. Operating system manager maintains a list of free blocks.

ii. **First-fit-** Allocate the first block that is big enough.



KEY
J1- 30K
J2 -50K
J3- 35K
J4- 25K

3.1.3.2 Relocatable Dynamic Partitions.

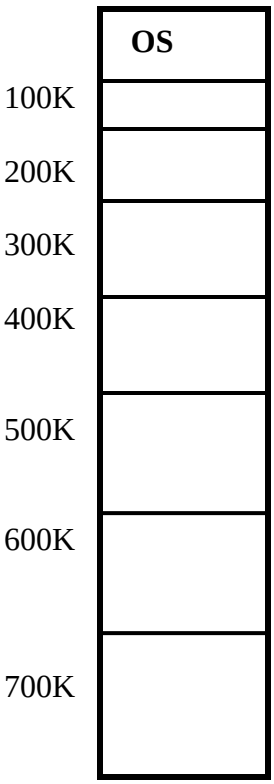
- Memory manager shuffles the memory contents to gather together all the empty blocks and compact them to make one block of memory large enough to accommodate some or all jobs waiting.
- Memory compaction (Garbage collector).

Limitations
While compaction is being done, all other tasks must wait

3.2 NON-CONTIGUOS MEMORY MANAGEMENT

3.2.1 Paged Memory Allocation.

- Memory is divided into fixed sized blocks (page frames).
- Jobs are also divided into pages of equal size i.e pages and frames are of same size.
- Memory manager determines the no of pages in the program.
- Then locates enough empty page frames in memory, then loads all program pages not necessarily contiguous.



N/B

- They are equal size partitions known as page frames.
- The page frames have address.

Disadvantage

i. Internal fragmentation in the last page frame.
ii. Requires complex algorithms to keep track of pages for each job.

Advantage

i. Allows non- Contiguous allocation i.e. it eliminates external fragmentation.
ii. It eliminates compaction of memory.

3.2.2 Demand Paging.

- Concept loads only those pages needed currently for processing into memory.
- Other pages initially reside on secondary storage.

Advantage

i. Requires less memory to run jobs.

Page Replacement Algorithms

Demand paging must have a policy to decide which page frames are to be replaced.

a. First In Fist Out (FIFO).

- FIFO queue holds all pages in memory.

- When a page must be replaced, the oldest page is chosen.
- When a page is brought into memory it is inserted at the tail of the queue.

b. Least Recently used (LRU)

- Swaps out pages that have not been used for the longest period of time.
- The algorithm associates time with each page for its last use.

3.2.3 Segmented Memory Allocation

- Each job is divided into several segments of different sizes i.e. one for each module (function, subroutines, stacks, queues).
- Memory is not divided into page frames because each segment size is different.
- Memory is allocated dynamically as per segment size.

3.2.4 Segmented- Demand paged Memory Allocation.

- Combination of segmentation and demand paging.
- First a program is divided into segments then each segment is sub-divided into pages of equal size- so that they can easily be manipulated than whole segments.
- Segmented pages are brought into main memory when required.

3.3 Virtual Memory Management System.

Problems with Contiguous and Non-Contiguous Memory Management.

i. The entire process (i.e. page, segment, and job) must be in memory at the time of execution.
ii. If a process is to be removed, the entire process must be swapped out.
iii. The degree of multiprogramming was limited; i.e. less processes can be contained in main memory.

Virtual memory is a technique which allows execution of processes or jobs that may not be completely stored in memory i.e. other parts of a process can be stored on disk (virtual space).

Advantages and Disadvantages of Virtual Memory

Advantages

i. Job size is no longer restricted to size of main memory.
ii. Allows unlimited multiprogramming.
iii. It eliminates external fragmentation and minimizes internal fragmentation.
iv. Facilitates dynamic linking of program.

Disadvantages

i. Increased hardware costs.
ii. Software complexity to prevent thrashing

N/B

Thrashing is a process of spending more time paging than executing i.e. a process with less frames will want to replace a page whereby that actual page will be needed right, which may cause page faults.

REVISION QUESTIONS

a. Write notes on the following memory management jargons.
   i. Locality of Reference.
   ii. Working Set.
   iii. Page Replacement Policy.
   iv. Dirty Page/Dirty bit.

b. Differentiate between contiguous, real memory management and non-contiguous memory.
c. Differentiate Single contiguous and fixed partitioned memory.
d. What is demand paging and what are its advantages.

Unit 4: Topic 4: PROCESSOR MANAGER.

4.0 Single User System

- Only one job is executed at a time by the processor (CPU).
- Processor is busy when the job is being executed.
- Other times the processor is idle (I/O operations).

4.1 Multiprogramming Systems

- Many users, many job executing simultaneously.
- Processor is not idle most of the time.
- Processor is allocated to each job or to each process for a period of time and then deallocated after the time elapses.

Processor manager has two components.

1. **Job Scheduler**
   It selects jobs from disk and puts them into a process (READY) queue in main memory for execution.
2. **Processor Scheduler**
   It selects processes from the process queue and allocates them to the CPU. It also determines when and for how long a job should have the CPU. Some Processes are **I/O bound** and others are **CPU bound.**

4.2 Job /Process States.

a. HOLD
b. READY
c. RUNNING
d. FINISHED
e.

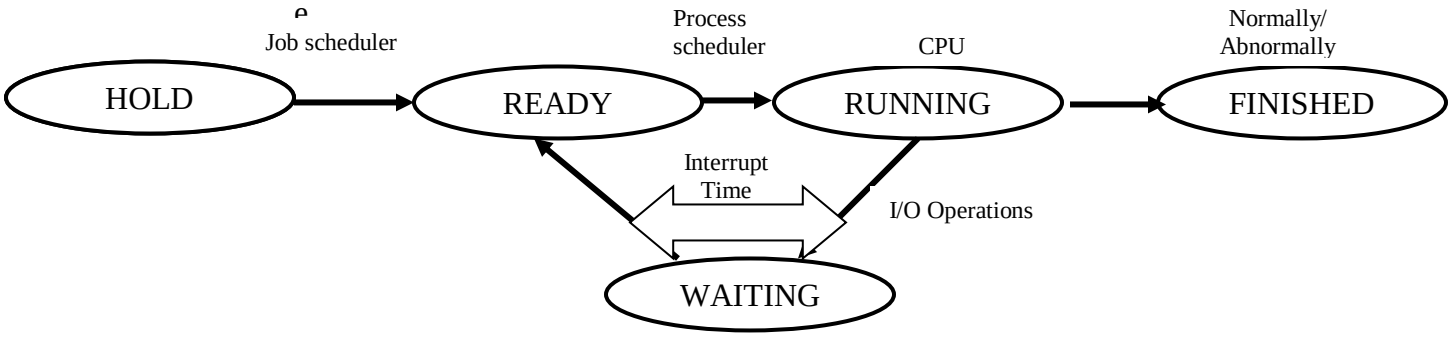


Diagram for Job /Process States.

4.3 PROCESS SCHEDULING

These are the ways which the operating system uses to allocate the CPU to processes in the READY queue.

4.3.1 Types of Process Scheduling

1. **Pre-emptive Scheduling.**
   This is a scheduling policy which interrupts the processing of a job and transfers the CPU to another job.
2. **Non-Preemptive Scheduling.**

The process is executed up to completion without intervention by any interrupts.

4.4 Scheduling Algorithms

a. First Come First Serve

This is a non-preemptive scheduling algorithm that uses a FIFO queue. The jobs are allocated to the CPU depending on their time of arrival in the ready queue (FIFO queue).

Example 1.

Consider the jobs below together with their CPU cycles in milliseconds.

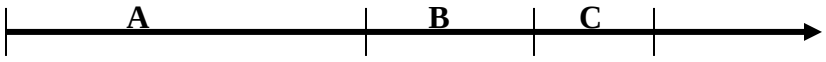
Job A-15

Job B- 2

Job C- 1

Assuming that the jobs arrived in the order A, B, C; calculate the average turn around time and average waiting time.

Solution.

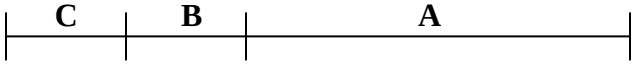


0 15 17 18

Average Turn around Time= $\frac{\mathbf{15+17+18}}{\mathbf{3}}$ **Milliseconds.**

Example 2.

Suppose the same jobs arrived in a different order i.e. C, B, A.



0 1 3 18

Average Turn around Time= $\frac{\mathbf{1+3+18}}{\mathbf{3}}$ **Milliseconds.**

Average Waiting Time (Using Example A) = $\frac{0+15+17}{3}$

Note: In Example above

A ⟶ 0

B ⟶ 15

C ⟶ 17

Using Example 2, Average turn around time.

C, B, A

=**0 +1 + 3 Milliseconds.**
3

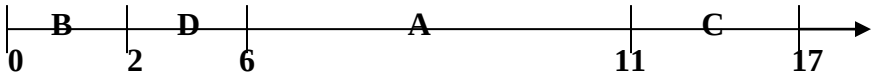
b. Shortest Job Next.

This is a non-preemptive scheduling algorithm. The CPU is allocated jobs that have the shortest time of execution.

Example 1.

Consider the jobs below in the ready queue.

JOB	**A**	**B**	**C**	**D**
CPU CYCLE	**5**	**2**	**6**	**4**



Average Turn Around Time.

= 2 + 6+ 11+ 17
4

Average waiting Time = 0 + 2+ 6+ 11 Milliseconds.
4

c. Shortest Remaining Time (SRT).

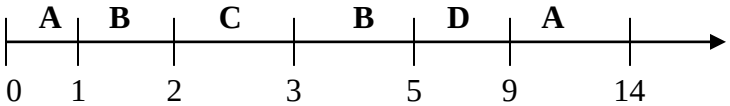
This is a preemptive version of SJN. The processor is allocated to the job closest to completion, but even this job can be pre-empted if a newer job comes to the ready queue and has the shortest time.

Example 1.

Consider the following jobs together with their arrival time and CPU cycle time. Calculate the average turn around and waiting times.

JOB	**A**	**B**	**C**	**D**
CPU CYCLE	**6**	**3**	**1**	**4**
Arriva Time	**0**	**1**	**2**	**3**

Solution



Average Turn Around Time = 14 + 4 +1+ 6 Milliseconds.
4

Average Waiting Time=8 + 1 +0+ 2 Milliseconds.
4

d. Round Robin Scheduling

This is a preemptive scheduling algorithm; each process is allocated a small amount of time slice (time quantum) to the CPU. Once the time slice expires, the process is preempted back to the queue.

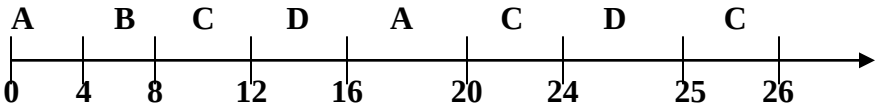
Example 1
Consider the following jobs along with their arrival time and a time slice of 4 milliseconds.

Arrival Time: 0 1 2 3

JOB : A B C D

CPU CYCLE: 8 4 9 5

Calculate the average turnaround time and waiting time.



A ⟶ 20

B ⟶ 7

C ⟶ 24

D ⟶ 22

Average turnaround time= $\frac{20 +7 +24 +22}{4}$ Milliseconds.

Average Waiting time= $\frac{12 +3 +15 + 17}{4}$ Milliseconds.

Highest Response Ratio Next (HRRN) Scheduling

HRRN (Highest Response Ratio Next) Scheduling is a non-preemptive scheduling algorithm in the operating system. It is one of the optimal algorithms used for scheduling.

As **HRRN is a non-preemptive scheduling algorithm** so in case if there is any process **that is currently in execution with the CPU** and during its execution, if any new process arrives in the memory with burst time smaller than the currently running process then at that time the currently running process will not be put in the ready queue & complete its execution without any interruption.

HRRN is a modification of **Shortest Job Next(SJN)** in order to reduce the problem of starvation.

In the HRRN scheduling algorithm, the CPU is assigned to the next process that has the **highest response ratio** and not to the process having less burst time.

Algorithm of HRRN

- In the **HRRN** scheduling algorithm, once a process is selected for execution it will run until its completion.
- The first step is to calculate the waiting time for all the processes. Waiting time simply means the sum of the time spent waiting in the ready queue by a processes.
- Processes get scheduled each time for execution in order to find the response ratio for each available process.
- Then after the process having the highest response ratio is executed first by the processor.
- In case, if two processes have the same response ratio then the tie is broken using the FCFS scheduling algorithm.

To calculate the Response ratio.

Response Ratio = (W+S)/S

Where,

W= Waiting Time.

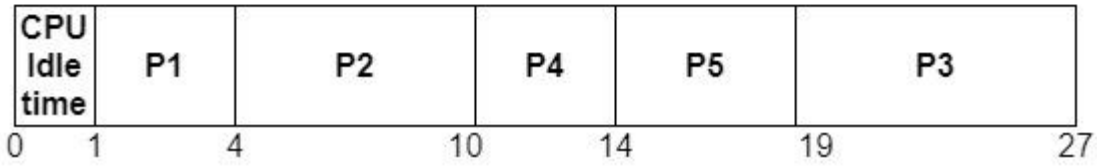
S= Service time / Burst Time.

HRRN Scheduling Example

Here, we have several processes with different burst and arrival times, and a Gantt chart to represent CPU allocation time.

Process	Burst Time	Arrival Time
P1	3	1
P2	6	3
P3	8	5
P4	4	7
P5	5	8

The Gantt Chart according to HRRN will be:



Explanation

Given below is the explanation of the above example

- At time=0 there is no process available in the ready queue, so from time 0 to 1 CPU is idle. Thus 0 to 1 is considered as CPU idle time.
- At time=1, only the process P1 is available in the ready queue. So, process P1 executes till its completion.
- After process P1, at time=4 only process P2 arrived, so the process P2 gets executed because the operating system did not have any other option.
- At time=10, the processes P3, P4, and P5 were in the ready queue. So in order to schedule the next process after P2, we need to calculate the response ratio.
- In this step, we are going to calculate the response ratio for P3, P4, and P5.

Response Ratio = W+S/S

RR(P3) = [(10-5) +8]/8

= 1.625

RR(P4) = [(10-7) +4]/4

= 1.75

RR(P5) = [(10-8) +5]/5

= 1.4

From the above results, it is clear that Process P4 has the Highest Response ratio, so the Process P4 is schedule after P2.

- At time t=10, execute process P4 due to its large value of Response ratio.
- Now in the ready queue, we have two processes P3 and P5, after the execution of P4 let us calculate the response ratio of P3 and P5

RR (P3) = [(14-5) +8]/8

=2.125

RR (P5) = [(14-8) +5]/5

=2.2

From the above results,it is clear that Process P5 has the Highest Response ratio, so the Process P5 is schedule after P4

- At t=14, process P5 is executed.
- After the complete execution of P5, P3 is in the ready queue so at time t=19 P3 gets executed.

Process	**Arrival Time**	**Burst Time**	**Completion time**	**Turnaround Time** **Turn Around Time = Completion Time – Arrival Time**	**Waiting Time** **Waiting Time = Turn Around Time – Burst Time**
P1	1	3	4	4-1=3	3-3=0
P2	3	6	10	10-3=7	7-6=1
P3	5	8	27	27-5=22	22-8=14
P4	7	4	14	14-7=7	7-4=3

P5	8	5	19	19-8=11	11-5=6

In the table given below, we will calculate turnaround time, waiting time, and completion time for all the Processes.

Average waiting time is calculated by adding the waiting time of all processes and then dividing them by no. of processes.

average waiting time = waiting time of all processes/ no.of processes

average waiting time=0+1+14+3+6/5=24/5 = **4.8ms**

HRRN Scheduling Example in C Language

Given below is the C program for HRRN Scheduling:

```
// C program for the Highest Response Ratio Next (HRRN) Scheduling

#include <stdio.h>



// This structure defines the details of the process

struct process {

        char name;

        int at, bt, ct, wt, tt;

        int complete;

        float ntt;

} p[10];



int m;
```

```
//The Sorting of Processes by Arrival Time

void sortByArrival()

{

        struct process temp;

        int i, j;



        // Selection Sort applied

        for (i = 0; i < m - 1; i++) {

                for (j = i + 1; j < m; j++) {



                        // This condition is used to Check for the lesser arrival time

                        if (p[i].at > p[j].at) {



                                // Swaping of earlier process to front

                                temp = p[i];

                                p[i] = p[j];

                                p[j] = temp;

                        }

                }

        }
```

```
}


void main()
{
        int i, j, t, sum_bt = 0;
        char c;
        float avgwt = 0, avgtt = 0;
        m = 5;

        // the predefined arrival times
        int arriv[] = { 1, 3, 5, 7, 8 };

        // the predefined burst times
        int burst[] = { 3, 7, 8, 4, 2};

        // Initialize the structure variables
        for (i = 0, c = 'A'; i < m; i++, c++) {
                p[i].name = c;
                p[i].at = arriv[i];
                p[i].bt = burst[i];
```

```
            // Variable for Completion status
            // for Pending = 0
            // for Completed = 1
            p[i].complete = 0;

            // the Variable for the sum of all Burst Times
            sum_bt += p[i].bt;
}

// Let us Sort the structure by the arrival times
sortByArrival();
printf("\nName\tArrival Time\tBurst Time\tWaiting Time");
printf("\tTurnAround Time\t Normalized TT");
for (t = p[0].at; t < sum_bt;) {

            // Now Set the lower limit to response ratio
            float hrr = -9999;

            //The Response Ratio Variable
```

```
float temp;


// Variable used to store the next processs selected
int loc;
for (i = 0; i < m; i++) {

        // Check if the process has arrived and is Incomplete
        if (p[i].at <= t && p[i].complete != 1) {

                // Calculating the Response Ratio
                temp = (p[i].bt + (t - p[i].at)) / p[i].bt;

                // Checking for the Highest Response Ratio
                if (hrr < temp) {

                        // Storing the Response Ratio
                        hrr = temp;

                        // Storing the  Location
                        loc = i;
```

```
                        }
                }
        }


        // Updating the time value
        t += p[loc].bt;


        // Calculation of the waiting time
        p[loc].wt = t - p[loc].at - p[loc].bt;


        // Calculation of the Turn Around Time
        p[loc].tt = t - p[loc].at;


        // Sum of Turn Around Time for the average
        avgtt += p[loc].tt;


        // Calculation of the Normalized Turn Around Time
        p[loc].ntt = ((float)p[loc].tt / p[loc].bt);


        // Updating  the Completion Status
```

```
                    p[loc].complete = 1;



                    // Sum of the Waiting Time to calculate the average

                    avgwt += p[loc].wt;

                    printf("\n%c\t\t%d\t\t", p[loc].name, p[loc].at);

                    printf("%d\t\t%d\t\t", p[loc].bt, p[loc].wt);

                    printf("%d\t\t%f", p[loc].tt, p[loc].ntt);

            }

            printf("\nThe Average waiting time:%f\n", avgwt / m);

            printf("The Average Turn Around time:%f\n", avgtt / m);

}
```

Output

```
Name    Arrival Time    Burst Time      Waiting Time    TurnAround Time  Normalized TT
A               1               3               0               3               1.000000
B               3               7               1               8               1.142857
D               7               4               4               8               2.000000
E               8               2               7               9               4.500000
C               5               8               12              20              2.500000
The Average waiting time:4.800000
The Average Turn Around time:9.600000
```

Advantages of HRNN Scheduling

The advantages of the HRRN scheduling algorithm are as follows:

1. HRRN Scheduling algorithm gives better performance than the shortest job first Scheduling.
2. With this algorithm, there is a reduction in waiting time for longer jobs and also it encourages shorter jobs.
3. With this algorithm, the throughput increases.

Disadvantages of HRNN Scheduling

The disadvantages of the HRNN scheduling algorithm are as follows:

1. The Practical implementation of HRRN scheduling is not possible because we cannot know the burst time of every process in advance.
2. In this scheduling, there may occur overhead on the processor.

Multilevel Queues

The CPU and the Operating System maintains two or more ready queues.

Examples.

i. The operating system may group processes into two categories i.e. CPU bound and I/O bound processes, and then places the processes in their corresponding ready queue.
ii. The processes can be grouped as foreground (interactive) or background (batch) processes and then placed in their corresponding ready queue respectively.
iii. The process can be grouped according to priority such that processes with same priorities are put in the same queue thus forming multi-level queues.

Each of the queues in the three examples has its own scheduling algorithm. e.g. the foreground queue may be handled with Round Robin Scheduling whereas the background queue is being handled with First come First Serve.

Revision Questions

i. Consider the following jobs along with their arrival time and a time slice of 4 milliseconds.

Arrival Time:	**0**	**1**	**2**	**3**
JOB	**: A**	**B**	**C**	**D**
CPU CYCLE:	**8**	**4**	**9**	**5**

Using Round Robin Scheduling Algorithm, Calculate the average turnaround time and waiting time.

ii. Consider the following jobs together with their arrival time and CPU cycle time. Calculate the average turn around and waiting times.

JOB	**A**	**B**	**C**	**D**
CPU CYCLE	**6**	**3**	**1**	**4**

iii. Discuss the various CPU scheduling Algorithms.
iv. Discuss the various types of process scheduling algorithms.
v. Consider the following jobs along with their arrival time in milliseconds.

Arrival Time: 0 1 2 3

JOB : A B C D

CPU CYCLE: 8 4 9 5

Using Highest Response Ratio Next (HRRN) Scheduling, Calculate the average turnaround time and waiting time

Unit 5: Topic 5: INTURRUPTS AND DEADLOCKS

5.1 Interrupts.

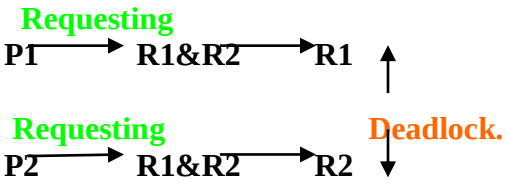
An interrupt is an event that disrupts the normal processing of the CPU.

Types of Interrupts.

i. Timer Interrupts-This is an event where the processor is deallocated from running a job when a quantum expires or approaches
ii. I/O Interrupts-These are used when Input (read) / Output (write) command is used.
iii. Internal Interrupts/Synchronous/Program Interrupts.
   - Divide by zero.
   - Arithmetic overflow.
iv. Illegal Job Instructions e.g.
   - Attempt to access protected data/location.
   - Attempt to access non-existent data/location.
   - Attempt to make system changes (BIOS).

5.2 Deadlocks.

This is a situation where a process is **P1** is holding a resource **R1** and waiting for a resource **R2** being held by process **P2** which in turn is waiting for a resource **R1** to complete execution. Both the processes **(P1 and P2)** cannot continue execution since they are waiting for each other's resource; hence a deadlock i.e.



5.3 Resources

Types of Resources

i. Hardware resources
They include;
   - Floppy disk.
   - Hard disk.
   - CD-Rom.
   - Printer.
   - Plotter.
   - Scanner.
   - Tape drive/Zip drive.

ii. Software Resources.

- Access to software application.
- Data in a database /file.

Resources can be categorized into two

1) Pre-emptive resources

This is a resource that can be taken away from the process owning it without causing errors e.g. Main Memory locations.

2) Non-preemptive resources

A resource that cannot be taken away from it's current owner without causing errors e.g. a printer.

Sequence of events required by a resource

i. Process requests a resource.
ii. Process uses the resource.
iii. Release the resource.

5.4 Deadlock prevention

Deadlock prevention falls into two classes.

1) Indirect Method.

This method prevents the occurrence of one of the following conditions.

a. Mutual exclusive.
b. Hold and wait.
c. No pre-emption.
d. Circular wait.

2) Direct Method.

This method prevents occurrence of circular wait.

a. Mutual Exclusive.

This is whereby once a process has been allocated a particular resource; it has an exclusive use of the resource.

To prevent mutual exclusion:

If access to a resource requires mutual exclusion, then this should be supported by the Operating System. The resource should be held exclusively only for the write privilege i.e.

A- Read memory location C.
B- Read memory location C.
D = C- 100.
E- Update Memory location C: → Mutual exclusive
C= 100+ 2

b. Hold and Wait.

A situation where a process is holding a resource while at the same time requesting for another resource.

Solution 1

Let the process request for the resources required in advance or else the process is blocked until all requested resources can be granted.

Problems with the above solution.

Resources will not be used optimally, this may lead to **starvation,** and also dynamic processes may not know how and which type of resources they will require.

Solution 2

Let a process request for resources temporarily and begin execution, then release those resources incase it can't have access to the remaining resources.

c. No Pre-emption.

This is a situation whereby a resource can be released only by the explicit action of a process –willingly- rather than the action of external entity.

Solutions

- **i.** If a process is holding certain resources; it is denied a further request, then the process must release all it's original resources and then request them afterwards with the new resource they still wanted.
- **ii.** Alternatively, if a process requests for a resource currently held by another process, the Operating system may pre-empt the second process and require its resources. This should depend on the priority of the process.

d. Circular Wait

This occurs when there exists a set {P0, P1, P2, P3……..Pn}of waiting processes such that P0 is waiting for a resource held by P1 and P3……Pn-1 is waiting for a resource held by Pn and Pn is also waiting for a resource held by P0.

P1 ⟶P2⟶ P3 ⟶ P4

Solution

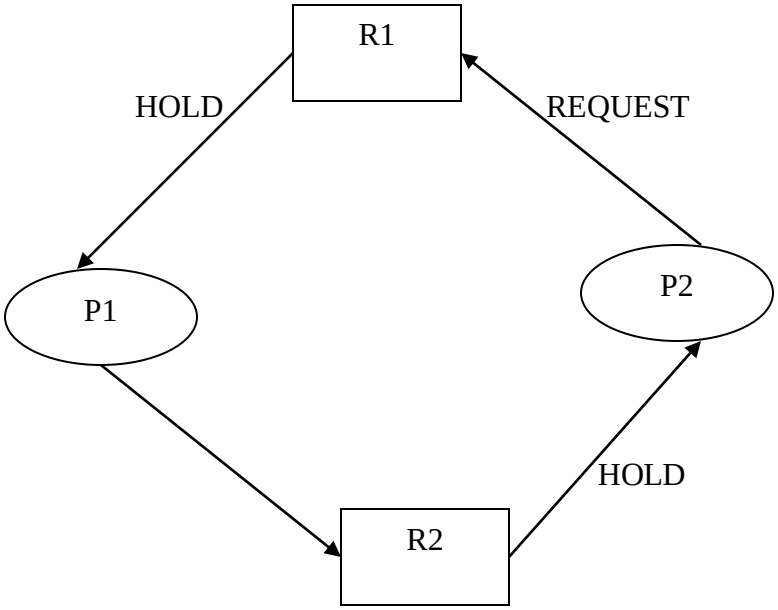
Resources are divided into groups called resource types. If a process is allocated a resource of type R then it can't subsequently request for resources of another type. The process must stick to its resource types otherwise it gets aborted.

5.5 Deadlock Avoidance.

Avoidance is a predictive approach which relies on information about the resources' activity that will be occurring for the process e.g. the process has to announce in advance the maximum number of resources it will request.

The operating system maintains a resource allocation graph. If there is a cycle in the graph then the resource causing a cycle will not be granted.

Example of resource allocation graph



Dining Philosophers Problem (DPP)

The dining philosopher's problem states that there are 5 philosophers sharing a circular table and they eat and think alternatively. There is a bowl of rice for each of the philosophers and 5 chopsticks. A philosopher needs both their right and left chopstick to eat. A hungry philosopher may only eat if there are both chopsticks available. Otherwise a philosopher puts down their chopstick and begin thinking again.

The dining philosopher is a classic synchronization problem as it demonstrates a large class of concurrency control problems.

Solution of Dining Philosophers Problem

A solution of the Dining Philosophers Problem is to use a semaphore to represent a chopstick. A chopstick can be picked up by executing a wait operation on the semaphore and released by executing a signal semaphore.

The structure of the chopstick is shown below −

semaphore chopstick [5];

Initially the elements of the chopstick are initialized to 1 as the chopsticks are on the table and not picked up by a philosopher.

The structure of a random philosopher i is given as follows −

```
do {
  wait(chopstick[i]);
  wait (chopstick [ (i+1) % 5]);
  . .
  . EATING THE RICE
  .
  signal(chopstick[i]);
  signal (chopstick [ (i+1) % 5]);
  .
  . THINKING
  .
} while (1);
```

In the above structure, first wait operation is performed on chopstick[i] and chopstick [(i+1) % 5]. This means that the philosopher i has picked up the chopsticks on his sides. Then the eating function is performed.

After that, signal operation is performed on chopstick[i] and chopstick [(i+1) % 5]. This means that the philosopher i has eaten and put down the chopsticks on his sides. Then the philosopher goes back to thinking.

Difficulty with the solution

The above solution makes sure that no two neighboring philosophers can eat at the same time. But this solution can lead to a deadlock. This may happen if all the philosophers pick their left chopstick simultaneously. Then none of them can eat and deadlock occurs.

Some of the ways to avoid deadlock are as follows −

- There should be at most four philosophers on the table.
- An even philosopher should pick the right chopstick and then the left chopstick while an odd philosopher should pick the left chopstick and then the right chopstick.
- A philosopher should only be allowed to pick their chopstick if both are available at the same time.

Revision Questions .

i. What do you understand by the term interrupt?
ii. Discuss the types of interrupts that are found in processors.
iii. What is a deadlock? What are the various measures undertaken to prevent deadlocks?
iv. Using an Example explain the dining philosopher's problem using Semaphores.

Unit 6: Topic 6: Device Management

The device manager, manages every peripheral device of the system, it uses the master-slave mechanism.

It includes the following functions: -

i. It keeps track of the status of each device i.e. (in-use, idle, faulty).
ii. Allocates processes to devices using preset policies.
iii. De-allocates devices from processes.

6.1 System Devices.

Devices fall into three categories:

1. Dedicated devices.
2. Shared devices.
3. Virtual devices.

1. Dedicated devices.

These are devices assigned to only one job e.g. printer, Scanner, mouse, Joy-stick etc.

2. Shared devices.

These are devices that are assigned several jobs at a time e.g. Hard-disk, floppy, CD….).

3. Virtual devices.

These are devices which can be converted in a shared device by use of Operating System e.g. printer (Spooling) by use of printer buffer.

6.2 Storage Media.

Can be grouped into:-

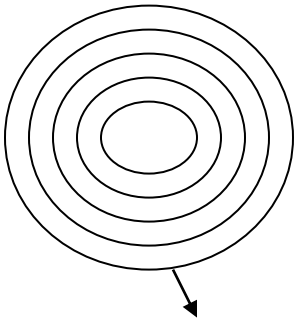
1) Sequential access e.g. magnetic tape.
2) Direct access e.g. CD, Hard-disk.

Direct access devices:

Can be grouped into:-

a. Fixed, read/ write heads
b. Movable read/write heads.

CD-Rom (Optical storage device)
Data/files are written sequentially in a spiral form whereas they are accessed directly i.e.



Search Time.

Access Time
The time required to access a file on a disk.
There are three factors that determine access time:

i. Seek time- It is the time required to place a read/write head on a proper track.
ii. Search time (rotation delay)-The time it takes to rotate the drum or disks until the required record is moved under read/write head.
iii. Transfer time-time it takes to transfer data from secondary storage to main memory.

Fixed head devices, Access time=Search time + transfer time.
Movable head devices, Access time=Seek time+ Search time+ transfer time.

Revision Questions.

i. What are the various functions carried out by any device manager?
ii. Discuss the various categories of system devices found in a processor.
iii. What do you understand by the term virtual devices
iv. Discuss the various groups of storage media.

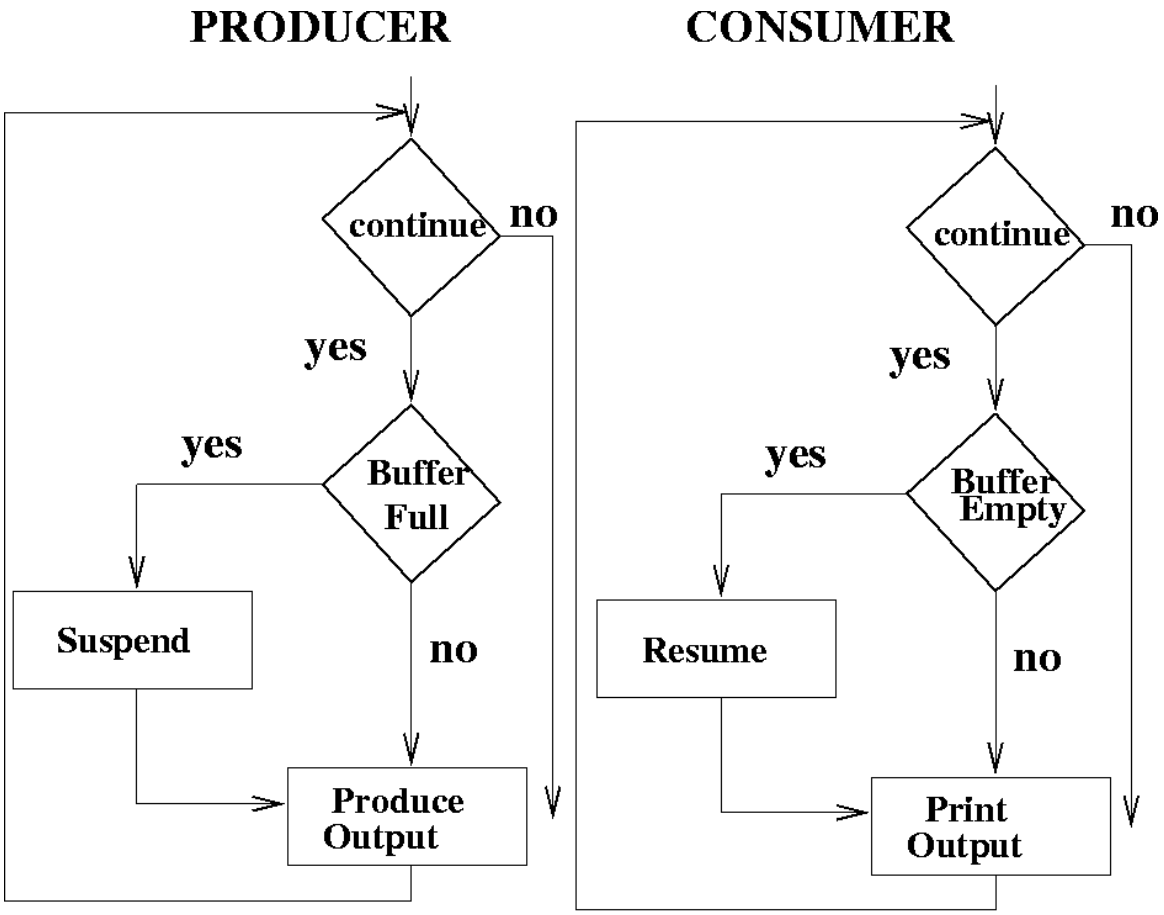
Unit 7: Topic 7: SYNCHRONISATION AND INTERPROCESS COMMUNICATION

7.1 Cooperating Processes

- Concurrent Processes can be
  - Independent processes
    - cannot affect or be affected by the execution of another process.
  - Cooperating processes
    - can affect or be affected by the execution of another process.
- Advantages of process cooperation:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience(e.g. editing, printing, compiling)
- Concurrent execution requires
  - process communication and process synchronization

7.2 Producer-Consumer Problem

- Paradigm for cooperating processes;
  - producer process produces information that is consumed by a consumer process.
- We need buffer of items that can be filled by producer and emptied by consumer.
  - Unbounded-buffer places no practical limit on the size of the buffer. Consumer may wait, producer never waits.
  - Bounded-buffer assumes that there is a fixed buffer size. Consumer waits for new item, producer waits if buffer is full.
  - **Producer and Consumer must synchronize.**



7.3 Bounded-buffer - Shared Memory Solution

- Shared data

var *n*;

type *item* = ….;

var *buffer*: array[0..n-1] of *item*;

in, *out*: 0..*n*-1;

in :=0; *out*:= 0; /* shared buffer = circular array */

/* Buffer empty if *in* == *out* */

/* Buffer full if (*in*+1) mod *n* == *out* */

/* *noop* means 'do nothing' */

Producer process - creates filled buffers

repeat

…

produce an item in *nextp*

…

while *in*+1 mod *n* = *out* do *noop*;

buffer[in] := *nextp*;

in := *in+1* mod *n*;

until *false*;

- Consumer process - Empties filled buffers

repeat

while *in* = *out* do *noop*;

nextc := *buffer[out]* ;

out:= *out*+1 mod *n*;

…

consume the next item in *nextc*

…

until *false*

7.4 Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Shared memory solution to the bounded-buffer problem allows at most (n-1) items in the buffer at the same time.

7.5 The Critical-Section Problem

- N processes all competing to use shared data.

- Structure of process P*i* ---- Each process has a code segment, called the critical section, in which the shared data is accessed.

repeat
entry section /* enter critical section */
critical section /* access shared variables */
exit section /* leave critical section */
remainder section /* do other work */
until false

- Problem
  - Ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Solution: Critical Section Problem - Requirements

- **Mutual Exclusion**
  - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
- **Progress**
  - If no process is executing in its critical section and there exists some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
- **Bounded Waiting**
  - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
- Assume that each process executes at a nonzero speed.
- No assumption concerning relative speed of the n processes.

Solution: Critical Section Problem -- Initial Attempt

- Only 2 processes, P0 and P1
- General structure of process Pi (Pj)

repeat
entry section
critical section
exit section
remainder section
until false

- Processes may share some common variables to synchronize their actions.

7.6 Semaphores

The refrigerator example solution is much too complicated. The problem is that the mutual exclusion mechanism was too simple-minded: it used only atomic reads and

writes. This is sufficient, but unpleasant. It would be unbearable to extend the fridge mechanism to many processes. Let's look for a more powerful, higher-level mechanism.

Requirements for a mutual exclusion mechanism:

- Must allow only one process into a critical section at a time.
- If several requests at once, must allow one process to proceed.
- Processes must be able to go on vacation outside critical section.

Desirable properties for a mutual exclusion mechanism:

- Fair: if several processes waiting, let each in eventually.
- Efficient: do not use up substantial amounts of resources when waiting. E.g. no busy waiting.
- Simple: should be easy to use (e.g. just bracket the critical sections).

Desirable properties of processes using the mechanism:

- Always lock before manipulating shared data.
- Always unlock after manipulating shared data.
- Do not lock again if already locked.
- Do not unlock if not locked by you.
- Do not spend large amounts of time in critical section. E.g. do not go on vacation.

Semaphore: A synchronization variable that takes on positive integer values. Invented by Dijkstra.

- P(semaphore): an atomic operation that waits for semaphore to become positive, then decrements it by 1.
- V(semaphore): an atomic operation that increments semaphore by 1.

  The names come from the Dutch, *proberen* (test) and *verhogen* (increment). Semaphores are simple and elegant and allow the solution of many interesting problems. They do a lot more than just mutual exclusion.

Too much milk problem with semaphores:

	Processes A & B
1 2 3	OKToBuyMilk.P(); if (NoMilk) BuyMilk(); OKToBuyMilk.V();

Note: OKToBuyMilk must initially be set to 1. What happens if it is not?

Show why there can never be more than one process buying milk at once.

Binary semaphores are those that have only two values, 0 and 1. They are implemented in the same way as regular semaphores except multiple V's will not increase the semaphore to anything greater than one.

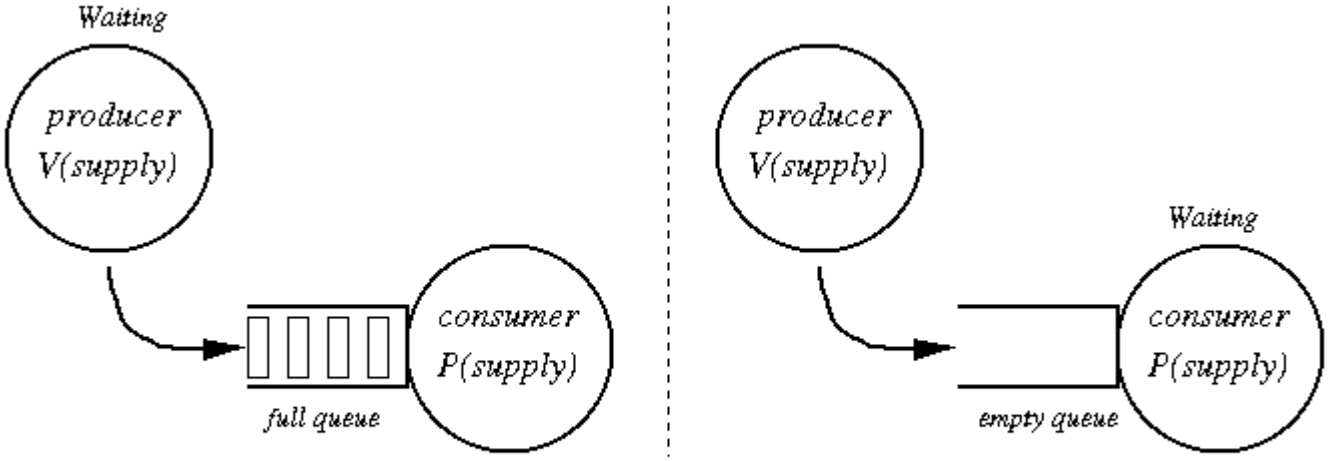
Semaphores are not provided by hardware (we will discuss implementation later). But they have several attractive properties:

- Machine independent.
- Simple.
- Powerful. Embody both exclusion and waiting.
- Correctness is easy to determine.
- Work with many processes.
- Can have many different critical sections with different semaphores.
- Can acquire many resources simultaneously (multiple P's).
- Can permit multiple processes into the critical section at once, if that is desirable.

Semaphores can be used for things other than simple mutual exclusion. For example, resource allocation: P = allocate resource, V = return resource. More on this later.

7.7 Producer & Consumer Problem

Consider the operation of an assembly line or pipeline. Processes do not have to operate in perfect lock-step, but a certain order must be maintained. For example, must put wheels on before the hub caps. It is OK for wheel mounter to get ahead, but hub-capper must wait if it gets ahead. Another example: compiler & disk reader.



Producer and Consumer

- Producer: creates copies of a resource.
- Consumer: uses up (destroys) copies of a resource.
- Synchronization: keeping producer and consumer in step.
- Define constraints (definition of what is "correct").
    - B must wait for A to fill buffers. (resource management)
    - A must wait for B to empty buffers. (resource management)
    - Only one process must fiddle with buffer list at once. (mutual exclusion)
- A separate semaphore is used for each constraint.
- Declare and initialize variables:

```
semaphore empty(N),
          full(0),
    list(1);

List empties (N),
   fulls (0);
```

Process A (Producer):

```
empty.P();
list.P();
b = empties.remove();
list.V();

Put data into B

list.P();
```

fulls.add(b);
list.V();
full.V();

Process B (Consumer):

full.P();
list.P();
b = fulls.remove();
list.V();

Remove data from B

consume data in buffer;
list.P();
empties.add(b);
list.V();

- empty.V();
- Important questions:
  - Why does A do a empty.P() but full.V()?
  - Why is order of P's important?
  - Is order of V's important?
  - Could we have separate semaphores for each list?
  - How would this be extended to have 2 consumers?

Producers and consumers are much like UNIX pipes.

Unit 8: Topic 8: FILE AND DISK MANAGEMENT

What Are Files?

Suppose we are developing an application program. A program, which we prepare, is a file. Later we may compile this program file and get an object code or an executable. The executable is also a file. In other words, the output from a compiler may be an object code file or an executable file. When we store images from a web page we get an image file. If we store some music in digital format it is an audio file. So, in almost every situation we are engaged in using a file. In addition, we saw in the previous module that files are central to our view of communication with IO devices. So let us now ask again:

What is a file?

File: a named collection of bits stored on disk. From the OS' standpoint, the file consists of a bunch of blocks stored on the device. Programmer may actually see a different interface (bytes or records), but this does not matter to the file system (just pack bytes into blocks, unpack them again on reading).

Irrespective of the content any organized information is a file.
So be it a telephone numbers list or a program or an executable code or a web image or a data logged from an instrument we think of it always as a file. This formlessness and disassociation from content was emphasized first in UNIX. The formlessness essentially means that files are arbitrary bit (or byte) streams. Formlessness in UNIX follows from the basic design principle: keep it simple. The main advantage to a user is flexibility in organizing files. In addition, it also makes it easy to design a file system. A file system is that software which allows users and applications to organize their files. The organization of information may involve access, updates and movement of information between devices. Later in this module we shall examine the user view of organizing files and the system view of managing the files of users and applications. We shall first look at the user view of files.
User's view of files: The very first need of a user is to be able to access some file he has stored in a non-volatile memory for an on-line access. Also, the file system should be able to locate the file sought by the user. This is achieved by associating an identification for a file i.e. a file must have a name. The name helps the user to identify the file. The file name also helps the file system to locate the file being sought by the user.
Let us consider the organization of my files for the Compilers course and the Operating Systems course on the web. Clearly, all files in compilers course have a set of pages that are related. Also, the pages of the OS system course are related. It is, therefore, natural to think of organizing the files of individual courses together. In other words, we would like to see that a file system supports grouping of related files. In addition, we would like that all such groups be put together under some general category (like COURSES).
This is essentially like making one file folder for the compilers course pages and other one for the OS course pages. Both these folders could be placed within another folder, say COURSES. This is precisely how MAC OS defines its folders. In UNIX, each such group, with related files in it, is called a directory. So the COURSES directory may have subdirectories OS and COMPILERS to get a hierarchical file organization. All modern OSs support such a hierarchical file organization. In Figure 2.1 we show a hierarchy of

files. It must be noted that within a directory each file must have a distinct name. For instance, I tend to have ReadMe file in directories to give me the information on what is in each directory. At most there can be only one file with the name “ReadMe" in a directory. However, every subdirectory under this directory may also have its own ReadMe file. UNIX emphasizes disassociation with content and form. So file names can be assigned any way.

Some systems, however, require specific name extensions to identify file type. MSDOS identifies executable files with a .COM or .EXE file name extension. Software systems like C or Pascal compilers expect file name extensions of .c or .p (or .pas) respectively. In Section 2.1.1 and others we see some common considerations in associating a file name extension to define a file type.

Common addressing patterns:

- Sequential: information is processed in order, one piece after the other. This is by far the most common mode: e.g. editor writes out new file, compiler compiles it, etc.
- Random Access: can address any record in the file directly without passing through its predecessors. E.g. the data set for demand paging, also databases.
- Keyed: search for records with particular values, e.g. hash table, associative database, dictionary. Usually not provided by operating system. *TLB is one example of a keyed search.*

Modern file systems must address four general problems:

- Disk Management: efficient use of disk space, fast access to files, sharing of space between several users.
- Naming: how do users select files?
- Protection: all users are not equal.
- Reliability: information must last safely for long periods of time.

Disk Management: how should the disk sectors be used to represent the blocks of a file? The structure used to describe which sectors represent a file is called the *file descriptor*.

Contiguous allocation: allocate files like segmented memory (give each disk sector a number from 0 up). Keep a free list of unused areas of the disk. When creating a file, make the user specify its length, allocate all the space at once. Descriptor contains location and size.

Directories

Users need a way of finding the files that they created on disk. One approach is just to have users remember descriptor indexes.

Of course, users want to use text names to refer to files. Special disk structures called *directories* are used to tell what descriptor indices correspond to what names.

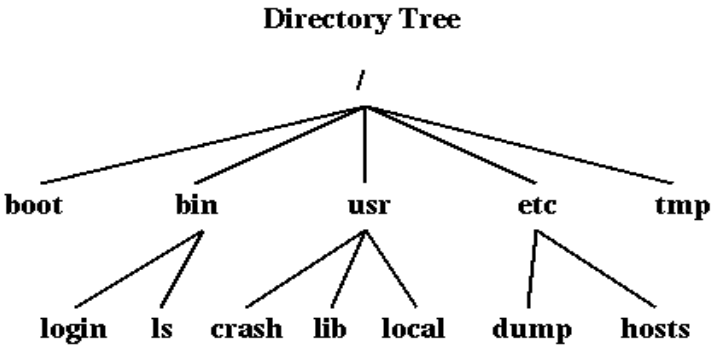
A hard concept to understand at the beginning: naming is one of the (if not *the*) most important issues in systems design.

Approach #1: have a single directory for the whole disk. Use a special area of disk to hold the directory.

- Directory contains pairs.
- If one user uses a name, no-one else can.

UNIX Directories

UNIX approach: generalize the directory structure to a tree.

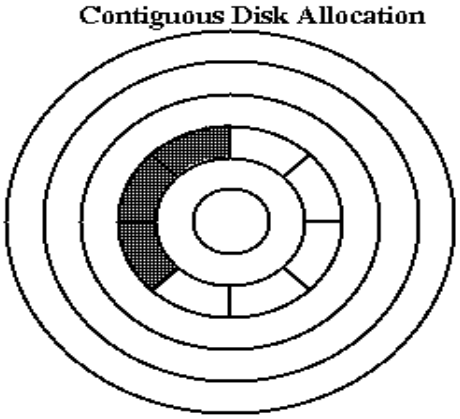


- Directories are stored on disk just like regular files (i.e. file descriptor with 13 pointers, etc.). User programs can read directories just like any other file (try it!). Only special system programs may write directories.
- Each directory contains pairs. The file pointed to by the index may be another directory. Hence, get hierarchical tree structure, name with /usr/local.
- There is one special directory, called the *root*. This directory has no name, and is the file pointed to by descriptor 2 (descriptors 0 and 1 have other special purposes).

It is very nice that directories and file descriptors are separate, and the directories are implemented just like files. This simplifies the implementation and management of the structure (can write "normal" programs to manipulate them as files).

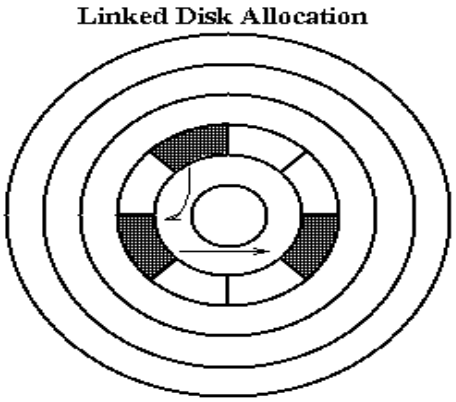
Working directory: it is cumbersome constantly to have to specify the full path name for all files.

- In UNIX, there is one directory per process, called the working directory that the system remembers.
- When it gets a file name, it assumes that the file is in the working directory. "/" is an escape to allow full path names.
- Many systems allow more than one current directory. For example, check first in A, then in B, then in C. This set of directories is called the *search path* or *search list*. This is very convenient when working on large systems with many different programmers in different areas.
- For example, in UNIX the shell will automatically check in several places for programs. However, this is built into the shell, not into UNIX, so if any other program wants to do the same, it has to rebuild the facilities from scratch. Should be in the OS.
- This is yet another example of locality.



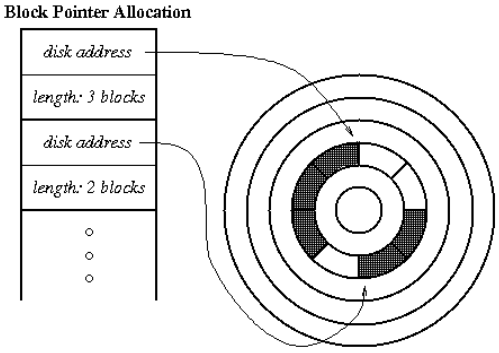
- Advantages: easy access, both sequential and random. Simple. Few seeks.
- Drawbacks: horrible fragmentation will preclude large files, hard to predict needs. With interleaved user requests, still cannot eliminate all seeks.

Linked files: In file descriptor, just keep pointer to first block. In each block of file keep pointer to next block. Can also keep a linked list of free blocks for the free list.



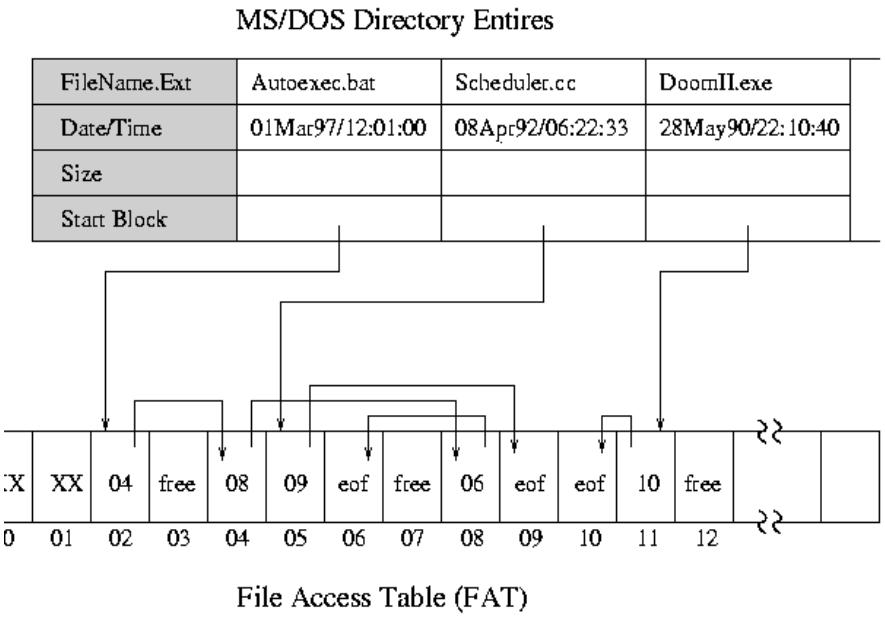
- Advantages: files can be extended, no fragmentation problems. Sequential access is easy: just chase links.
- Drawbacks: random access is virtually impossible. Lots of seeking, even in sequential access.
- Example: FAT (MSDOS) file system.

Array of block pointers: file maximum length must be declared when it is created. Allocate an array to hold pointers to all the blocks, but do not allocate the blocks. Then fill in the pointers dynamically using a free list.



- Advantages: not as much space wasted by over predicting, both sequential and random access are easy.
- Drawbacks: still have to set maximum file size, and there will be lots of seeks.

DOS FAT allocation table: A single File Allocation Table (FAT) that combines free list info and file allocation info. In file descriptor, keep pointer to first block. A FAT table entry contains either (1) the block number of the next block in the file, (2) a distinguished "end of file" (eof) value, or (3) a distinguished "free" value.



- Advantages/Disadvantages: similar to those mentioned above for linked file.

  None of these is a very good solution: what is the answer? First, and MOST IMPORTANT: understand the application. How are file systems used?

  - Most files are small.
  - Much of the disk is allocated to large files.
  - Many of the I/O's are made to large files.

  Thus, the per-file cost must be low, but the performance of large files must be good. Must allow reasonably fast random access, extensibility.

Windows (NT) File System

The Windows file system is called NTFS, and was introduced with Windows NT 4.0 and is the standard file system on Windows 2000 and later systems, such as Windows XP. Its goal was to solve the size, performance, reliability, and flexibility limitations in the DOS (aka "FAT" file system).

It has a general similarity to the FAT file system in that all files are described in a single table, called the *Master File Table* (MFT). However, it has more modern characteristics in that all components are files, including:

- Master File Table
- data files
- free list (bit map)
- boot images

- directories
- recovery logs

The file system also has features to support redundancy and transactions, which we will not discuss. A great reference for details is the book: **Inside the Windows NT File System** by Helen Custer, published by (not surprisingly) Microsoft Press.

Disk Layout

Disks are divide in fixed size regions:

- Each region is called a *volume*.
- Each volume can contain a different kind of file system, such as NTFS, FAT, or even UNIX.
- Since each volume is a separate file system, it has its own root directory.
- Multiple volumes allow for fixed limits on the growth of a particular file tree, such as limiting the size of temporary file space.
- Multiple volumes also allow a single disk to contain multiple, separating bootable operating system.

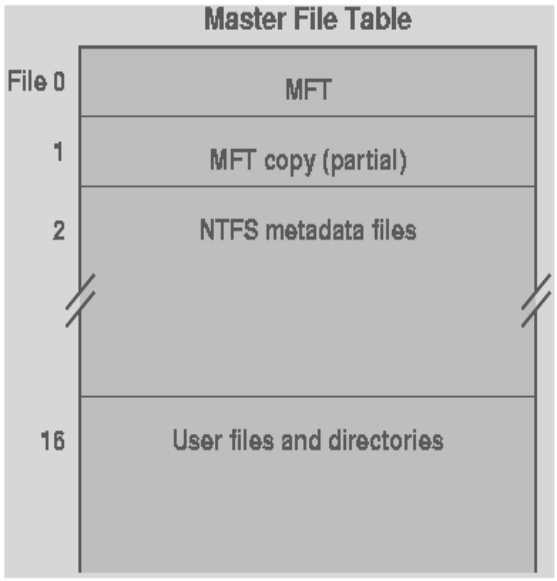
Master File Table (MFT)

Clusters are the key element to allocation:

- Logically, the disk consists of allocation units called ***clusters***.
- A cluster is a power-of-two multiple of the physical disk block size. The cluster size is set when the disk is formatted. A small cluster provides a finer granularity of allocation, but may require more space to describe the file and more separate operations to transfer data to or from memory.
- The free list is a **bitmap**, each of whose bits describe one cluster.
- Clusters on the disk are numbered starting from zero to the maximum number of clusters (minus one). These numbers are called *logical cluster numbers* (LCN) and are used to name blocks (clusters) on disk.

The MFT is the major, and in some ways, the only data structure on disk:

- All files, and therefore all objects stored on disk are described by the MFT.
- All files are logical stored *in* the MFT and, for small files are physically within the bounds of the MFT. In this sense, the MFT *is* the file system.
- The MFT logically can be described as a table with one row per file.
- The first rows in the table described important configuration files, including for the MFT itself.

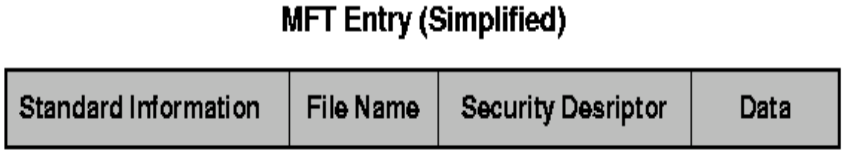


MFT Entries

As stated previously, each row or entry in the MFT (called a *record*) describes a file and logically contains the file. In the case of small files, the entry actually contains the contents of the file.

Each entry is consists of (attribute, value) pairs. While the conceptual design of NTFS is such that this set of pairs is extensible to include user-defined attributes, current version of NTFS have a fixed set. The main attributes are:

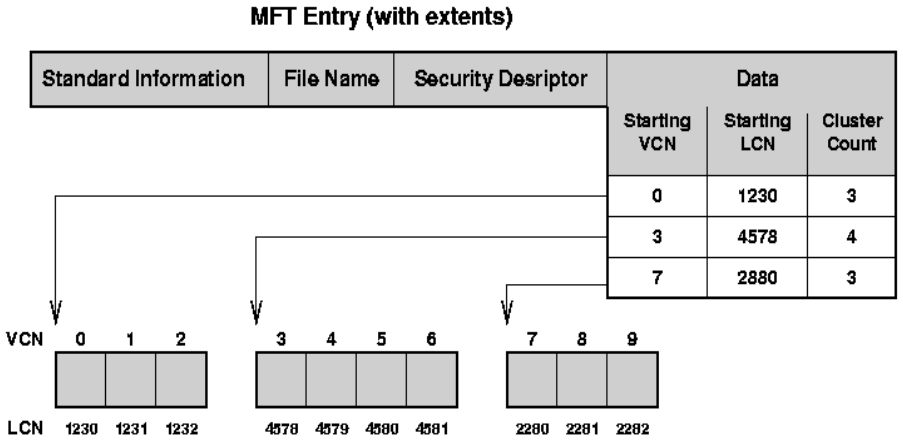
- *Standard information:* This attribute includes the information that was standard in the MS-DOS world:
  - read/write permissions,
  - creation time,
  - last modification time,
  - count of how many directories point to this this file (hard link count.
- *File Name:* This attribute describes the file's name in the Unicode character set. Multiple file names are possible, such as when:
  - the file has multiple links, or
  - the file has an MS-DOS short name.
- *Security Descriptor:* This attribute lists which user owns the file and which users can access it (and how they can access it).
- *Data:* This attribute either contains the actual file data in the case of a small file or points to the data (or points to the objects that point to the data) in the case of larger files.



For small files, this design is extremely efficient. By looking no further than the MFT entry, you have the complete contents of the file.

However, the Data field gets interesting when the data contained in the file is larger than an MFT entry. When dealing with large data, the Data attribute contains *pointers* to the data, rather than the data itself.

- The pointers to data are actually pointers to sequences of logical clusters on the disk.
- Each sequence is identified by three parts:
  - starting cluster in the file, called the *virtual cluster number* (VCN),
  - starting logical cluster (LCN) of the sequence on disk,
  - Length, counted as the number of clusters.
- The run of clusters is called an *extent*, following the terminology developed by IBM in the 1960's.
- NTFS allocates new extents as necessary. When there is no more space left in the MFT entry, then another MFT entry is allocated. This design is effectively a list of extents, rather than the UNIX or DEMOS tree of extents.



Directories

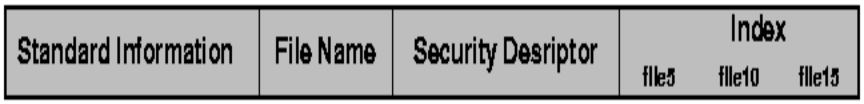
As with other modern file systems, a directory in NTFS is a file whose data contains a collection of name/file mappings.

- A directory entry contains the name of the file and *file reference*. The file references indentifies the file on this volume. In other words, it is an internal name for the file.

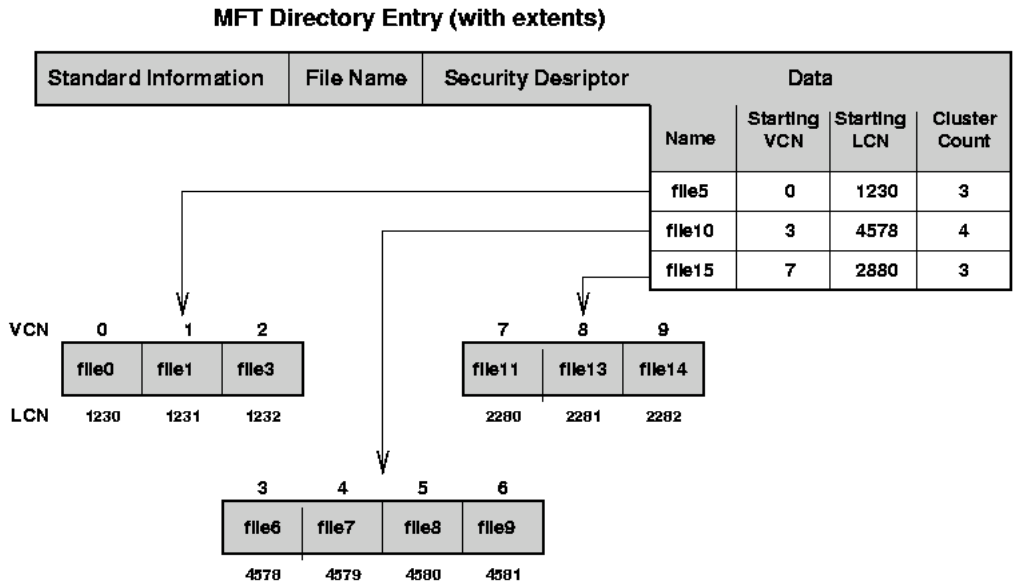
  A reference is a (*file number*, sequence number) pair. The file number is the offset of the file's entry in the MFT table. It is similar to the Unix inumber (Inode number).

- The list of file names in the directories is not stored in a simple list, but rather as a lexigraphically-sorted tree, called a B+ tree (this will be familiar to those with a database background). The data structure is called an *index* in NFTS (again, following the terminology from databases).
- The NTFS design specifies that an index can be constructed for any attribute, but currently only file name indices are supported.
- The name for a file appears both in its directory entry and in the MFT entry for the file itself.
- As with regular files, if the directory is small enough, it can fit entirely within the MFT entry.

MFT Directory Entry (Everything Fits)



If the directory is larger, then the top part of (the B+ tree of) the directory is in the MFT entry, which points to extents that contain the rest of the name/file mappings.



Storage Allocation

Information stored in memory is used in many different ways. Some possible classifications are:

- Role in Programming Language:
  - Instructions (specify the operations to be performed and the operands to use in the operations).
  - Variables (the information that changes as the program runs: locals, owns, globals, parameters, dynamic storage).
  - Constants (information that is used as operands, but that never changes: pi for example).
- Changeability:
  - Read-only: *(code, constants)*.
  - Read & write: *(variables)*.

Why is identifying non-changing memory useful or important?

- Initialized:
  - Code, constants, some variables: yes.
  - Most variables: no.
- Addresses vs. Data: Why is this distinction useful or important?
- Binding time:
  - Static: arrangement determined once and for all, before the program starts running. May happen at *compile-time*, *link-time*, or *load-time*.

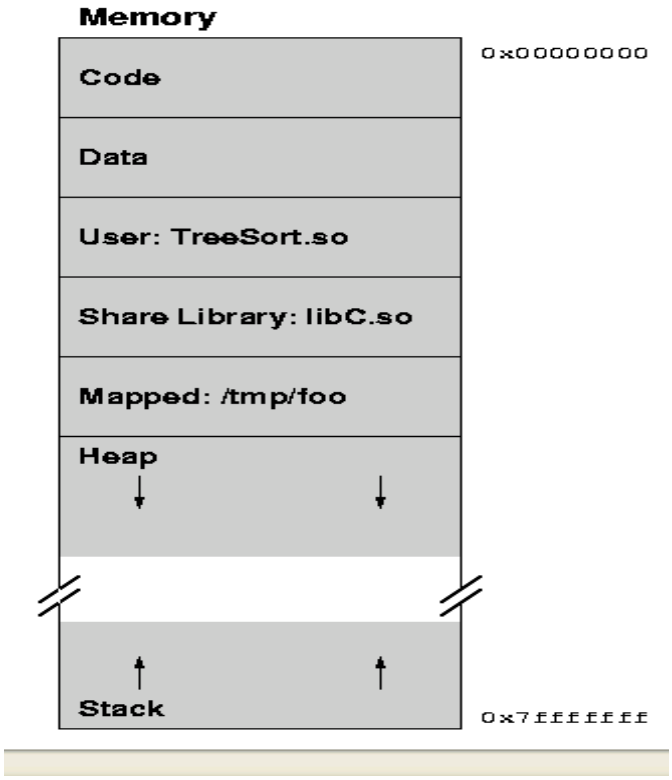
- Dynamic: arrangement cannot be determined until runtime, and may change.

Note that the classifications overlap: variables may be static or dynamic, code may be read-only or read&write, etc.

The compiler, linker, operating system, and run-time library all must cooperate to manage this information and perform allocation.

When a process is running, what does its memory look like? It is divided up into areas of stuff that the OS treats similarly, called *segments*. In Unix/Linux, each process has three segments:

- Code (called "text" in Unix terminology)
- Initialized data
- Uninitialized data
- User's dynamically linked libraries (*shared objects* (.so) or *dynamically linked libraries* (.dll))
- Shared libraries (system dynamically linked libraries)
- Mapped files
- Stack(s)



In some systems, can have many different kinds of segments.

One of the steps in creating a process is to load its information into main memory, creating the necessary segments. Information comes from a file that gives the size and contents of each segment (e.g. a.out in Unix/Linux and .exe in Windows). The file is called an *object file*.

Division of responsibility between various portions of system:

- Compiler: generates one object file for each source code file containing information for that file. Information is incomplete, since each source file generally uses some things defined in other source files.
- Linker: combines all of the object files for one program into a single object file, which is complete and self-sufficient.
- Operating system: loads object files into memory, allows several different processes to share memory at once, provides facilities for processes to get more memory after they have started running.
- Run-time library: provides dynamic allocation routines, such as *calloc* and *free* in C.

Dynamic Memory Allocation

Why is not static allocation sufficient for everything? Unpredictability: cannot predict ahead of time how much memory, or in what form, will be needed:

- Recursive procedures. Even regular procedures are hard to predict (data dependencies).
- OS does not know how many jobs there will be or which programs will be run.
- Complex data structures, e.g. linker symbol table. If all storage must be reserved in advance (statically), then it will be used inefficiently (enough will be reserved to handle the worst possible case).

Need dynamic memory allocation both for main memory and for file space on disk.

Two basic operations in dynamic storage management:

- Allocate
- Free

Dynamic allocation can be handled in one of two general ways:

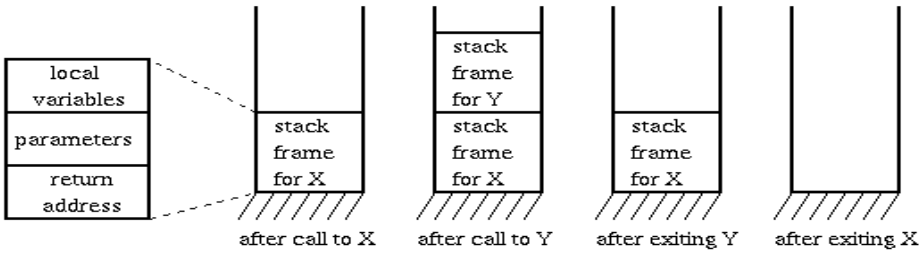
- Stack allocation (hierarchical): restricted, but simple and efficient.
- Heap allocation: more general, but less efficient, more difficult to implement.

Stack organization: memory allocation and freeing are partially predictable (as usual, we do better when we can predict the future). Allocation is hierarchical: memory is freed in

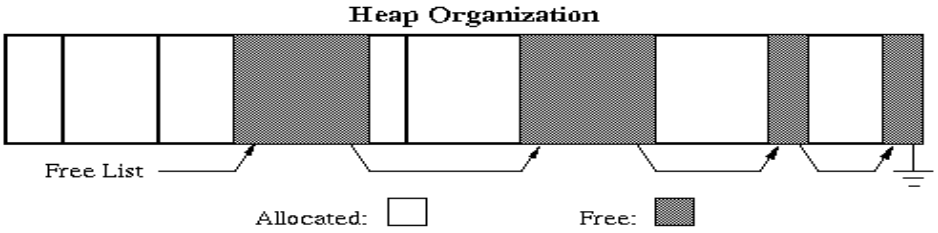
opposite order from allocation. If alloc(A) then alloc(B) then alloc(C), then it must be free(C) then free(B) then free(A).

- Example: procedure call. Program calls Y, which calls X. Each call pushes another stack frame on top of the stack. Each stack frame has space for variable, parameters, and return addresses.
- Stacks are also useful for lots of other things: tree traversal, expression evaluation, top-down recursive descent parsers, etc.

A stack-based organization keeps all the free space together in one place.



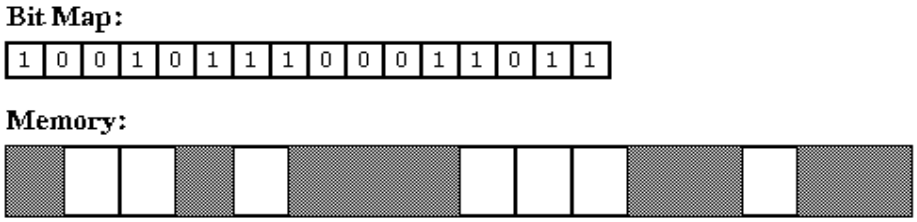
Heap organization: allocation and release are unpredictable. Heaps are used for arbitrary list structures, complex data organizations. Example: payroll system. Do not know when employees will join and leave the company, must be able to keep track of all them using the least possible amount of storage.



- Inevitably end up with lots of holes. Goal: reuse the space in holes to keep the number of holes small, their size large.
- Fragmentation: inefficient use of memory due to holes that are too small to be useful. In stack allocation, all the holes are together in one big chunk.
- Refer to Knuth volume 1 for detailed treatment of what follows.
- Typically, heap allocation schemes use a *free list* to keep track of the storage that is not in use. Algorithms differ in how they manage the free list.
    - Best fit: keep linked list of free blocks, search the whole list on each allocation, choose block that comes closest to matching the needs of the

allocation, save the excess for later. During release operations, merge adjacent free blocks.
    - First fit: just scan list for the first hole that is large enough. Free excess. Also merge on releases. Most first fit implementations are rotating first fit.

- Bit Map: used for allocation of storage that comes in fixed-size chunks (e.g. disk blocks, or 32-byte chunks). Keep a large array of bits, one for each chunk. If bit is 0 it means chunk is in use, if bit is 1 it means chunk is free. Will be discussed more when talking about file systems.



16 bits handles 16K of memory with the chunk (page) size of 1K

Pools: keep a separate allocation pool for each popular size. Allocation is fast, no fragmentation.

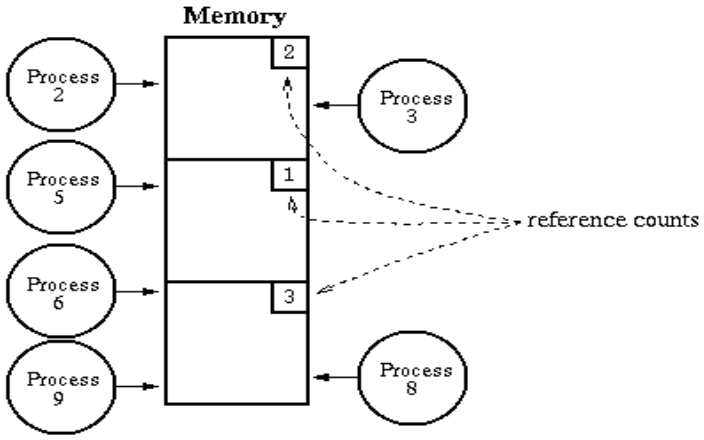
Reclamation Methods: how do we know when memory can be freed?

- It is easy when a chunk is only used in one place.
- Reclamation is hard when information is shared: it cannot be recycled until all of the sharers are finished. Sharing is indicated by the presence of *pointers* to the data (show example). Without a pointer, cannot access (cannot find it).

Two problems in reclamation:

- Dangling pointers: better not recycle storage while it is still being used.
- Core leaks: Better not "lose" storage by forgetting to free it even when it cannot ever be used again.

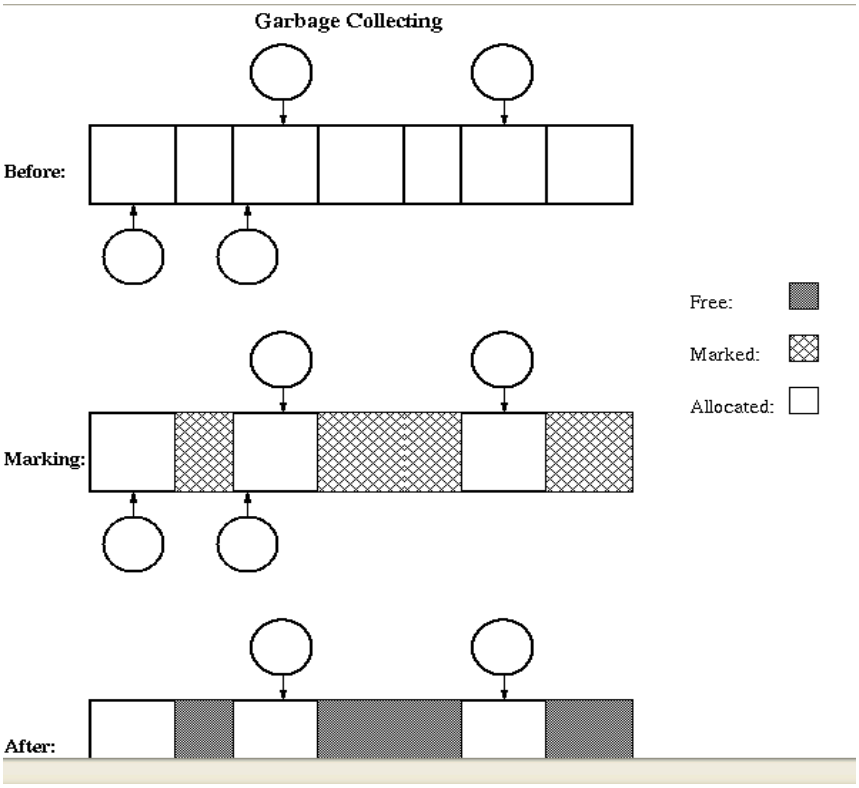
Reference Counts: keep track of the number of outstanding pointers to each chunk of memory. When this goes to zero, free the memory. Example: Smalltalk, file descriptors in Unix/Linux. Works fine for hierarchical structures. The reference counts must be managed automatically (by the system) so no mistakes are made in incrementing and decrementing them.



Garbage Collection: storage is not freed explicitly (using free operation), but rather implicitly: just delete pointers. When the system needs storage, it searches through all of the pointers (must be able to find them all!) and collects things that are not used. If structures are circular then this is the only way to reclaim space. Makes life easier on the application programmer, but garbage collectors are incredibly difficult to program and debug, especially if compaction is also done. Examples: Lisp, capability systems.

How does garbage collection work?

- Must be able to find all objects.
- Must be able to find all pointers to objects.
- Pass 1: mark. Go through all pointers that are known to be in use: local variables, global variables. Mark each object pointed to, and recursively mark all objects it points to.
- Pass 2: sweep. Go through all objects, free up those that are not marked.

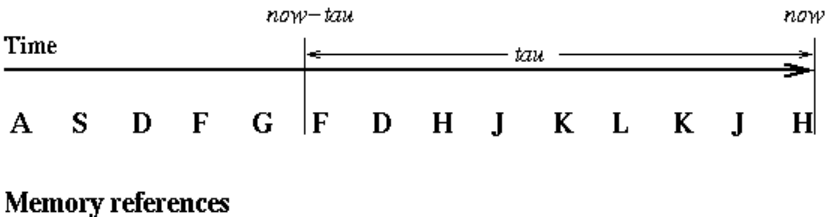


Garbage collection is often expensive: 20% or more of all CPU time in systems that use it.

Working Sets

Working Sets are a solution proposed by Peter Denning. An informal definition is "the collection of pages that a process is working with, and which must thus be resident if the process is to avoid thrashing." The idea is to use the recent needs of a process to predict its future needs.

- Choose tau, the working set parameter. At any given time, all pages referenced by a process in its last tau seconds of execution are considered to comprise its *working set*.



- A process will never be executed unless its working set is resident in main memory. Pages outside the working set may be discarded at any time.

Working sets are not enough by themselves to make sure memory does not get overcommitted. We must also introduce the idea of a *balance set*:

- If the sum of the working sets of all runnable processes is greater than the size of memory, then refuse to run some of the processes (for a while).
- Divide runnable processes up into two groups: active and inactive. When a process is made active its working set is loaded, when it is made inactive its working set is allowed to migrate back to disk. The collection of active processes is called the *balance set*.
- Some algorithm must be provided for moving processes into and out of the balance set. What happens if the balance set changes too frequently?

As working sets change, corresponding changes will have to be made in the balance set.

Problem with the working set: must constantly be updating working set information.

- One of the initial plans was to store some sort of a capacitor with each memory page. The capacitor would be charged on each reference, then would discharge slowly if the page was not referenced. Tau would be determined by the size of the capacitor. This was not actually implemented. One problem is that we want separate working sets for each process, so the capacitor should only be allowed to discharge when a particular process executes. What if a page is shared?
- Actual solution: take advantage of use bits
    - OS maintains *idle time* value for each page: amount of CPU time received by process since last access to page.
    - Every once in a while, scan all pages of a process. For each use bit on, clear page's idle time. For use bit off, add process' CPU time (since last scan) to idle time. Turn all use bits off during scan.
    - Scans happen on order of every few seconds (in Unix, tau is on the order of a minute or more).

This actual solution is an approximation algorithm known as *WSClock*.

Other questions about working sets and memory management in general:

- What should tau be?
    - What if it is too large?
    - What if it is too small?
- What algorithms should be used to determine which processes are in the balance set?
- How do we compute working sets if pages are shared?

- How much memory is needed in order to keep the CPU busy? Note than under working set methods the CPU may occasionally sit idle even though there are runnable processes.

References

1. William Stallings, *Operating Systems,* 5th edition, 2006.
2. Andrew S. Tanenbaum & Albert S. Woodhull, *Operating Systems Design and Implementation,* 3rd edition, 2006.
3. Milan Milenkovi'c, *Operating Systems concepts and Design,* 2nd edition.
4. Silberschatz, A. and Galvin B.: *Operating System Concepts* 7th Edition, Wiley Higher Education, 2007
5. Stallings, W.: *Operating Systems: Internals and Design Principles* 5th Edition, Addison-Wesley, 2004
6. Tanenbaum, A. and Woodhull, A. *Operating Systems* 2nd Edition, Prentice Hall, 1997.
7. Silberschatz, A. and Galvin B. *Operating System Concepts* 5th Edition, Addison Wesley, 1999.
8. Davis, William S.: *Operating Systems* 5th Edition, Addison Wesley, 2001.
9. Abrahams, Paul W.: *UNIX for the Impatient* Addison Wesley, 1992.