OPERATING SYSTEM LAB

CONTENTS

SYSTEM CONFIGURATION

CLIENT

HCL INFINITY 2000 BL

- Intel Pentium IV @3.0Ghz/HT
- Intel 915 Chipset
- 512 MB DDR II 400 RAM
- 80 GB SATA HDD
- 52 X CD ROM Drive
- 15” SVGA Digital Colour Monitor
- 10/100 PCI Ethernet Card with Boot ROM

SERVER

Wipro Systems

- Intel Pentium IV @2.6 GHz with HT
- 800Mhz FSB
- Intel 84G Chipset
- 256 MB DDR SDRAM
- 80GB UATA HDD
- 17” Digital Colour Monitor
- 52 X CD ROM Drive

Features of UNIX OS

Multitasking

Multitasking is the capability of the operating system to perform various tasks.ie., A single user can perform various tasks.

Mutiuser capability

This allows several users to use the same computer to perform their tasks.

Security

Every user have a login name and a password. So, accessing another user's data is impossible without permission

Portability

UNIX is portable because it is written in a high level language ©. So UXIX can be run on different computers.

Communication:

UNIX supports the following communications.

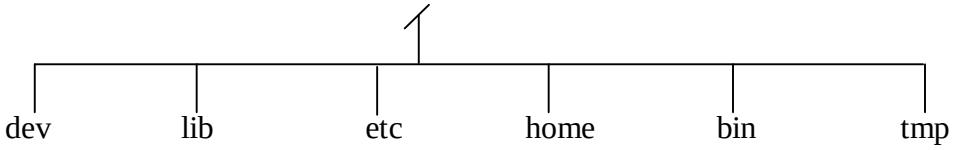
i) Between the different terminals connected to the UNIX server.

ii) Between the users of one computer to the users of another

Programming facility:

UNIX is highly programmable, the UNIX shell programming language has all the necessary ingredients like conditional and control structures (Loops) and variables.

Structure of a UNIX file system



/lib library files
/dev Contains file that link hardware devices
/tmp temporary storage of files
/home user home directories

Getting started with UNIX

Switching the system ON will provide the user with **login** prompt. Here we enter the login name. Then it prompts for the password. The password is not echoed on the screen to protect the privacy of the war. If both are correct, then we will get the **S** prompt.

UNIX Commands

Basic Commands

I File and Directory Related commands

1) pwd

This command prints the current working directory

2) ls

This command displays the list of files in the current working directory.

$ls –l Lists the files in the long format
$ls –t Lists in the order of last modification time
$ls –d Lists directory instead of contents
$ls -u Lists in order of last access time

3) cd

This command is used to change from the working directory to any other directory specified.

$cd directoryname

4) cd ..

This command is used to come out of the current working directory.

$cd ..

5) mkdir

This command helps us to make a directory.

$mkdir directoryname

6) rmdir

This command is used to remove a directory specified in the command line. It requires the specified directory to be empty before removing it.

$rmdir directoryname

7) cat

This command helps us to list the contents of a file we specify.

$cat [option][file]

cat > filename – This is used to create a new file.
cat >>filename – This is used to append the contents of the file

8) cp

This command helps us to create duplicate copies of ordinary files.

$cp source destination

9) mv

This command is used to move files.

$mv source destination

10) ln

This command is to establish an additional filename for the same ordinary file.

$ln firstname secondname

11) rm

This command is used to delete one or more files from the directory.

$rm [option] filename
$rm –i Asks the user if he wants to delete the file mentioned.
$rm –r Recursively delete the entire contents of the directory as well as the directory itself.

II) Process and status information commands

1) who

This command gives the details of who all have logged in to the UNIX system currently.

$ who

2) who am i

This command tells us as to when we had logged in and the system's name for the connection being used. $who am i

3) date

This command displays the current date in different formats.

+%D	mm/dd/yy	+%w	Day of the week
+%H	Hr-00 to 23	+%a	Abbr.Weekday
+%M	Min-00 to 59	+%h	Abbr.Month
+%S	Sec-00 to 59	+%r	Time in AM/PM
+%T	HH:MM:SS	+%y	Last two digits of the year

4) echo

This command will display the text typed from the keyboard.

$echo

Eg: $echo Have a nice day

O/p Have a nice day

II Text related commands

1. head

This command displays the initial part of the file. By default it displays first ten lines of the file.

$head [-count] [filename]

2. tail

This command displays the later part of the file. By default it displays last ten lines of the file.

$tail [-count] [filename]

3. wc

This command is used to count the number of lines, words or characters in a file.

$wc [-lwc] filename

4. find

The find command is used to locate files in a directory and in a subdirectory.

The –name option

This lists out the specific files in all directories beginning from the named directory. Wild cards can be used.

The –type option

This option is used to identify whether the name of files specified are ordinary files or directory files. If the name is a directory then use "-type d "and if it is a file then use "-type f".

The –mtime option

This option will allow us to find that file which has been modified before or after a specified time. The various options available are –mtime n(on a particular day),-mtime +n(before a particular day),-mtime –n(after a particular day)

The –exec option

This option is used to execute some commands on the files that are found by the find command.

IV File Permission commands

1) chmod

Changes the file/directory permission mode: $ chmod 777 file1

Gives full permission to owner, group and others

$ chmod o-w file1

Removes write permission for others.

V Useful Commands:

1) exit - Ends your work on the UNIX system.

2) Ctrl-l or clear

Clears the screen.

3) Ctrl-c

Stops the program currently running.

4) Ctrl-z

Pauses the currently running program.

5) man COMMAND

Looks up the UNIX command COMMAND in the online manual pages.

6) history

List all commands typed so far.

7) more FILE

Display the contents of FILE, pausing after each screenful.
There are several keys which control the output once a screenful has been printed.
<enter> Will advance the output one line at a time.
<space bar> Will advance the output by another full screenful.
"q" Will quit and return you to the UNIX prompt.

8) less FILE

"less" is a program similar to "more", but which allows backward movement in the file as well as forward movement.

9) lpr FILE

To print a UNIX text or PostScript file, type the following command at the system prompt:

Meta characters

Some special characters, called metacharacters may be used to specify multiple filenames. These characters substitute filenames or parts of filenames.

The "*"

This character is used to indicate any character(s)

$ cat ap*

This displays the contents of all files having a name starting with ap followed by any number of characters.

The "?" This character replaces any one character in the filename.

$ ls ?st

list all files starting with any character followed by st.

The [] These are used to specify range of characters.

$ ls [a-z]pple

Lists all files having names starting with any character from a to z.

Absolute path and relative path

Generally if a command is given it will affect only the current working directory. For example the following command will create a directory named curr in the current working directory.

$ mkdir curr

The directory can also be created else where in the file system using the absolute and relative path.If the path is given with respect to the root directory then it is called full path or absolute path $ mkdir /home/it2006/it2k601/curr

The full path always start with the /, which represents the root directory.

If the path is given with respect to the current working directory or parent directory then it is called relative path.

$ mkdir ../curr

The above command will create a directory named curr in the parent directory.

$ mkdir ./first/curr

The above command will create a directory named curr inside first directory , where the directory first is located in the current working directory.

Note "." Represents current directory and ".." represents parent directory.

PIPES AND FILTERS

In UNIX commands were created to perform single tasks only. If we want to perform multiple tasks we can go for pipes and filters.

PIPES

A pipe is a mechanism, which takes the output of a command as its input for the next command.

$who | wc –l

$cat text.c | head –3

FILTERS

Filters are used to extract the lines, which contain a specific pattern, to arrange the contents of a file in a sorted order, to replace existing characters with some other characters, etc.

1.Sort filter

The sort filter arranges the input taken from the standard input in alphabetical order. The sort command when used with "-r" option will display the input taken from the keyboard in the reverse alphabetical order. When used with "-n" option arranges the numbers, alphabets and special characters according to their ASCII value. If we want to sort on any one field, then sort provides us with an option called "+pos1 –pos2" option.

2.Grep filter

This command is used to search for a particular pattern from a file or from standard input and display those lines on the standard output. Grep stands for "Global search for regular expression".

There are various options available with grep command.

-v displays only those lines which do not match the pattern specified.

-c displays only the count of those lines which match the pattern specified

-n displays matched lines with line numbers

-i displays matched pattern ignoring case distinction

3.Uniq filter

The uniq filter compares adjacent lines in the sorted input file and when used with different options displays single and multiple occurrences.

-d displays only the lines which are duplicated in the input file.

-u displays only the lines with single occurrences.

4.Pg and more filter

These commands display the output of the command on the screen page by page. The difference between pg and more filter is that the viewing screen of the latter can be done by pressing space bar while that of the former is done by pressing enter.

5.Cut command

One particular field from any file or from output of any command can be extracted and displayed using this cut command. One particular character can also be extracted using the –c option of this command.

6.Tr command

This command is used to translate characters taken from the standard input. This command when used with "-s" option is used to squeeze multiple spaces into a single space.

1.UNIX SYSTEM CALLS

Objective

To write a programs using the following system calls of UNIX operating system: fork, exec, getpid, exit, wait, close, stat, opendir, readdir

Description

When a computer is turned on, the program that gets executed first is called the ``*operating system*." It controls pretty much all activity in the computer. This includes who logs in, how disks are used, how memory is used, how the CPU is used, and how you talk with other computers. The operating system we use is called "Unix".

The way that programs talk to the operating system is via ``*system calls*." A system call looks like a procedure call (see below), but it's different -- **it is a request to the operating system to perform some activity**.

Getpid

Each process is identified by a unique *process id* (called a "pid"). The init process (which is the supreme parent to all processes) posesses id 1. All other processes have some other (possibly arbitrary) process id. The getpid system call returns the current process' id as an integer.

```
// …
int pid = getpid();
printf("This process' id is %d\n",pid);// …
```

fork

The fork system call creates a new child process. Actually, it's more accurate to say that it *forks* a currently running process. That is, it creates a *copy* of the current process as a new child process, and then both processes resume execution from the fork() call. Since it creates two processes, fork also returns two values; one to teach process. To the parent process, fork returns the *process id of the newly created child process*. To the child process, fork returns 0. The reason it returns 0 is precisely because this is an invalid process id. You would have no way of differentiating between the parent and child processes if fork returned an arbitrary positive integer to each.

Therefore, a typical call to fork looks something like this:

```
int pid;
if ( (pid = fork()) == 0 ) {
   /* child process executes inside here */
}
else {
  /* parent process executes inside here */
}
```

execvp & execlp

The exec functions (there are more than one) are a family of functions that *execute* some program *within* the current process space. So if I write a program that calls one of the exec functions, as soon as the function call succeeds the original process gets *replaced* with whatever

program I asked exec to execute. This is usually used in conjunction with a fork call. You would typically fork a child process, and then call exec from within the child process, to execute some other program in the new process entry created by fork.

Directory Operations

The complex nature of Unix File system directory entries means that it is not realistic to read such a directory using normal *read()* system calls and, in fact, attempting to use *read()* to read a directory will fail. Instead the system call getdents() is used to read directory entries, however this has a rather complex interface and for all normal purposes a set of library routines is provided, these, of course, work via *getdents()*.

There are eight such routines and two data types.

Directory handling data objects

DIR	Used to hold a pointer to an open directory. Analagous to *FILE* *

struct dirent	A structure holding information about the directory entry.
	`struct dirent`
	`{`
	`ino_t d_ino; /* i-number */`
	`off_t d_off; /* offset into directory file */`
	`ushort d_reclen; /* length of record */`
	`char d_name[1]; /* file name */`
	`}`

Both the above are *#define*'d in the standard header **dirent.h**. The eight routines are

Prototype	Function
DIR *opendir(const char **path*)	Opens a directory
struct dirent *readdir(DIR **dirp*)	Gets the next entry
struct dirent *readdir_r(DIR **dirp*,struct dirent **res*)	Similar to *readdir*, takes address of buffer as parameter. Intended for MT applications.
long telldir(DIR **dirp*)	returns current location
void seekdir(DIR **dirp*,long *loc*)	alters current position
void rewinddir(DIR **dirp*)	set current position to start
int closedir(DIR **dirp*)	closes directory

2. I/O System Calls

Objective

To Write a programs using the I/O system calls of UNIX operating system (open, read, write, etc).

Description

System Calls for I/O

There are 5 basic system calls that Unix provides for file I/O. The system object that is used to manipulate files is file descriptor. This is an integer number that is used by the various I/O system calls to access a memory area containing data about the open file.

Open

Open makes a request to the operating system to use a file. The call takes two parameters. The first argument '**path**' specifies what file you would like to use, and the '**flags**' and 'mode' arguments specify how you would like to use it. This call returns a file descriptor.

The mode may be any of the following:

O_RDONLY
: Open the file in read-only mode.

O_WRONLY
: Open the file in write-only mode.

O_RDWR
: Open the file for both reading and writing.

In addition, any of the following flags may be OR-ed with the mode flag:

O_CREAT
: If the file does not exist already - create it.

O_EXCL
: If used together with O_CREAT, the call will fail if the file already exists.

O_TRUNC
: If the file already exists, truncate it (i.e. erase its contents).

O_APPEND
: Open the file in append mode. Any data written to the file is appended at the end of the file.

O_NONBLOCK (or O_NDELAY)
: If any operation on the file is supposed to cause the calling process block, the system call instead will fail, and errno be set to EAGAIN. This requires caution on the part of the programmer, to handle these situations properly.

O_SYNC
: Open the file in synchronous mode. Any write operation to the file will block until the data is written to disk. This is useful in critical files (such as database files) that must always remain in a consistent state, even if the system crashes in the middle of a file operation.

Close

Close() tells the operating system that you are done with a file descriptor. The OS can then reuse that file descriptor. The usage is

Close(file descriptor)

Read

Read() tells the operating system to read "size" bytes from the file opened in file descriptor "fd", and to put those bytes into the location pointed to by "buf". It returns how many bytes were actually read. The prototype is

int read(fd, buf, size)

Write

Write() is just like read(), only it writes the bytes instead of reading them. It returns the number of bytes actually written, which is almost invariably "size".

rename

The rename() system call may be used to change the name (and possibly the directory) of an existing file. It gets two parameters: the path to the old location of the file (including the file name), and a path to the new location of the file (including the new file name). If the new name points to an already existing file, that file is deleted first. We are allowed to name either a file or a directory.

```
/* rename the file 'logme' to 'logme.1' */
if (rename("logme", "logme1") == -1) {
   perror("rename (1):");
   exit(1);
}
```

delete

Deleting a file is done using the unlink() system call. This one is very simple:

```
/* remove the file "/tmp/data" */
if (unlink("/tmp/data") == -1) {
   perror("unlink");
   exit(1);
}
```

3.SIMULATION OF UNIX COMMANDS

Objective

To simulate the following unix commands
1)ls 2)grep

Description

ls

Use ls to see what files you have. Your files are kept in something called a directory.

ls---lists your files
ls -l --- lists your files in 'long format', which contains lots of useful information, e.g. the exact size of the file, who owns the file and who has the right to look at it, and when it was last modified.
ls -a --- lists all files, including the ones whose filenames begin in a dot, which you do not always want to see.

There are many more options, for example to list files by size, by date, recursively etc.

```
% ls
  foo       letter2
  foobar    letter3
  letter1   maple-assignment1
%
```

Note that you have six files. There are some useful variants of the **ls** command:

```
% ls l*
  letter1 letter2 letter3
%
```

Note what happened: all the files whose name begins with "l" are listed. The asterisk (*) is the " wildcard" character. It matches any string.

grep

grep ***string filename(s)*** --- looks for the string in the files. This can be useful a lot of purposes, e.g. finding the right file among many, figuring out which is the right version of something, and even doing serious corpus work. grep comes in several varieties (**grep**, **egrep**, and **fgrep**) and has a lot of very flexible options. Check out the man pages if this sounds good to you.

Use this command to search for information in a file or files. For example, suppose that we have a file *dict* whose contents are

```
red rojo
green verde
blue azul
white blanco
black negro
```

Then we can look up items in our file like this;

```
% grep red dict
  red rojo
```

```
% grep blanco dict
  white blanco
% grep brown dict
%
```

Notice that no output was returned by grep brown. This is because "brown" is not in our dictionary file.

Grep can also be combined with other commands. For example, if one had a file of phone numbers named "ph", one entry per line, then the following command would give an alphabetical list of all persons whose name contains the string "sona".

```
% grep sona ph | sort
  sona College of technology salem-5
```

The symbol "|" is called "pipe." It pipes the output of the grep command into the input of the sort command.

4. CPU SCHEDULING ALGORITHMS - I

Objective

To schedule the processes using FCFS(First Come First Served) and SJF(Shortest Job First) scheduling algorithms.

Description

When a computer is multi programmed , it has multiple processes competing for the CPU at the same time frequently. This situation occurs whenever two or more processes are simultaneously in the ready state. If only one CPU is available, a choice has to be made which process has to be in CPU. The part of the operating system that makes the choice is called the scheduler and the algorithm is called scheduling algorithm.

FCFS

In this scheduling policy the processes are assigned the CPU according to the order they arrive.

SJF

In this scheduling the process with shortest burst will be selected first. The processes are sorted in ascending order according to the CPU burst time.

Sample Input

Enter the number of processes:3
Process 1
Enter the CPU burst time: 5

Process 2
Enter the CPU burst time: 10

Process 3
Enter the CPU burst time:4

Sample Output

Process Name ArrivalTime BurstTime Wait time start End

The order in which the
processes are executed:

Waiting time for every
Proess Total waiting time is:

Average waiting time
for given FCFS :

Average turnaround time:

5. CPU SCHEDULING ALGORITHMS - II

Objective

To schedule the processes using Priority and Round Robin scheduling algorithms.

Description

Priority

In this scheduling policy the processes are given certain priorities usually specified as a number. They are sorted according to the priorities and the process with highest priority is scheduled first.

Round Robin

In this algorithm, a time quantum is fixed for the process to get executed in the CPU. After that time quantum, the process is pre-empted and CPU is scheduled to another process. This will continue until all processes in the system complete their turn.

Sample Input:

Enter the number of processes:3
Process 1
Enter the CPU burst time: 5

Process 2
Enter the CPU burst time: 10

Process 3
Enter the CPU burst time:4

Sample Output:

Process Name ArrivalTime BurstTime Wait time start End

The order in which the
processes are executed:

Waiting time for every
Proess Total waiting time is:

Average waiting time
for given FCFS :

Average turnaround time:

6. INTER PROCESS COMMUNICATION

Objective:

To implement Application using Inter Process communication (using shared memory, pipes or message queues).

Description:

Inter-process communication (**IPC**) is a set of techniques for the exchange of data among multiple threads in one or more processes. Processes may be running on one or more computers connected by a network. IPC techniques are divided into methods for message passing, synchronization, shared memory, and remote procedure calls (RPC). The method of IPC used may vary based on the bandwidth and latency of communication between the threads, and the type of data being communicated.

There are several reasons for providing an environment that allows process cooperation:

- Information sharing
- Computation speedup
- Modularity
- Convenience

IPC may also be referred to as *inter-thread communication* and *inter-application communication.*

1. **Pipes :** This allows the flow of data in one direction only. Data from the output is usually buffered until the input process receives it which must have a common origin.
2. **Named Pipes :** This is a pipe with a specific name. It can be used in processes that do not have a shared common process origin. Example is FIFO where the data is written to a pipe is first named.
3. **Message queuing:** This allows messages to be passed between messages using either a single queue or several message queues. This is managed by the system kernel. These messages are co-ordinated using an application program interface (API)
4. **Semaphores:** This is used in solving problems associated with synchronization and avoiding race conditions. They are integers values which are greater than or equal to zero
5. **Shared Memory:** This allows the interchange of data through a defined area of memory. Semaphore value has to be obtained before data can get access to shared memory.
6. **Sockets:** This method is mostly used to communicate over a network, between a client and a server. It allows for a standard connection which I computer and operating system independent.

7. PRODUCER – CONSUMER PROBLEM USING SEMAPHORES

Objective:

To Implement a the Producer – Consumer problem using semaphores (using UNIX system calls).

Description:

Producer-Consumer problem **(also known as the** bounded-buffer problem) is a classical example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time the consumer is consuming the data (i.e. removing it from the buffer) one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

The solution for the producer is to go to sleep if the buffer is full. The next time the consumer removes an item from the buffer, it wakes up the producer who starts to fill the buffer again. In the same way, the consumer goes to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer. The solution can be reached by means of inter-process communication, typically using semaphores. An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

The problem can also be generalized to have multiple producers and consumers.

Using semaphores:

Semaphores solve the problem of lost wakeup calls. In the solution below we use two semaphores, fillCount and emptyCount, to solve the problem. fillCount is incremented and emptyCount decremented when a new item has been put into the buffer. If the producer tries to decrement emptyCount while its value is zero, the producer is put to sleep. The next time an item is consumed, emptyCount is incremented and the producer wakes up. The consumer works analogously.

```
semaphore fillCount = 0
semaphore emptyCount = BUFFER_SIZE

procedure producer() {
   while (true) {
      item = produceItem()
      down(emptyCount)
      putItemIntoBuffer(item)
      up(fillCount)
   }
}

procedure consumer() {
   while (true) {
      down(fillCount)
      item = removeItemFromBuffer()
      up(emptyCount)
      consumeItem(item)
   }}
```

The solution above works fine when there is only one producer and consumer. Unfortunately, with multiple producers or consumers this solution contains a serious race condition that could result in two or more processes reading or writing into the same slot at the same time. To understand how this is possible, imagine how the procedure putItemIntoBuffer() can be implemented. It could contain two actions, one determining the next available slot and the other writing into it. If the procedure can be executed concurrently by multiple producers, then the following scenario is possible:

1. Two producers decrement emptyCount
2. One of the producers determines the next empty slot in the buffer
3. Second producer determines the next empty slot and gets the same result as the first producer
4. Both producers write into the same slot

To overcome this problem, we need a way to make sure that only one producer is executing putItemIntoBuffer() at a time. In other words we need a way to execute a critical section with mutual exclusion. To accomplish this we use a binary semaphore called mutex. Since the value of a binary semaphore can be only either one or zero, only one process can be executing between down(mutex) and up(mutex). The solution for multiple producers and consumers is shown below.

```
semaphore mutex = 1
semaphore fillCount = 0
semaphore emptyCount = BUFFER_SIZE

procedure producer() {
    while (true) {
        item = produceItem()
        down(emptyCount)
        down(mutex)
        putItemIntoBuffer(item)
        up(mutex)
        up(fillCount)
    }
    up(fillCount) //the consumer may not finish before the producer.
 }

procedure consumer() {
    while (true) {
        down(fillCount)
        down(mutex)
        item = removeItemFromBuffer()
        up(mutex)
        up(emptyCount)
        consumeItem(item)
    }
}
```

Notice that the order in which different semaphores are incremented or decremented is essential: changing the order might result in a deadlock.

8. MEMORY MANAGEMENT SCHEMES - I

Objective

To implement first fit, best fit and worst fit storage allocation algorithms for memory management.

Description

A set of holes, of various sizes is scattered through the memory at any given time. When a process arrives and needs the memory, the system searches for a hole that is large enough for this process. The first-fit, best-fit and worst-fit are strategies used to select a free hole from the set of available holes.

Implementation details

Free space is maintained as a linked list of nodes with each node having the starting byte address and the ending byte address of a free block. Each memory request consists of the process-id and the amount of storage space required in bytes. Allocated memory space is again maintained as a linked list of nodes with each node having the process-id, starting byte address and the ending byte address of the allocated space.

When a process finishes (taken as input) the appropriate node from the allocated list should be deleted and this free disk space should be added to the free space list. [Care should be taken to merge contiguous free blocks into one single block. This result in deleting more than one node from the free space list and changing the start and end address in the appropriate node]. For allocation use first fit, worst fit and best fit.

First-Fit

Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

Best-Fit

Allocate the smallest hole that is big enough. We must search the entire list unless the list is kept ordered by size. The strategy produces the smallest leftover hole.

Worst fit

Allocate the biggest hole.

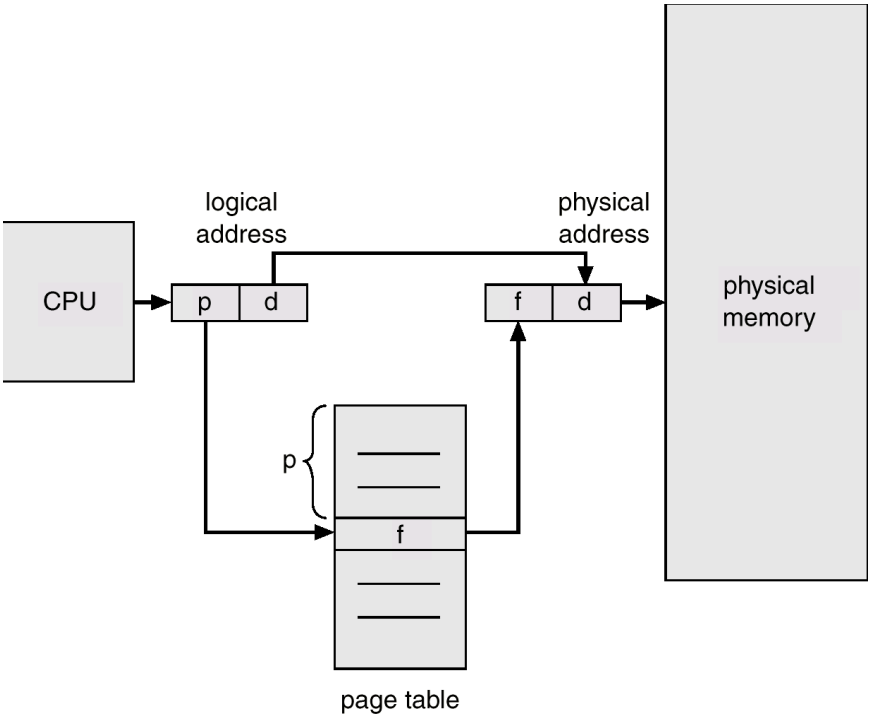
9. MEMORY MANAGEMENT SCHEMES - II

Objective

To implement memory allocation using paged memory management scheme.

Description

Paging is a memory allocation scheme that permits the physical address space of a process to be non-contiguous. Physical memory is broken into fixed-sized blocks called frames. Logical memory is also broken into blocks of same size called pages. When a process is to be executed, its pages are loaded into any available memory frames from the secondary memory. The secondary memory is divided into fixed size blocks of the same size as the memory frames.



Mapping of physical address to logical address is shown above. Every address generated by CPU is divided into two parts: a page number (p) and page offset (d). The page number is used as an index into a page table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical address that is sent to the memory unit.

10. FILE ALLOCATION TECHNIQUE

Objective

To implement a any file allocation technique (Linked, Indexed or Contiguous)

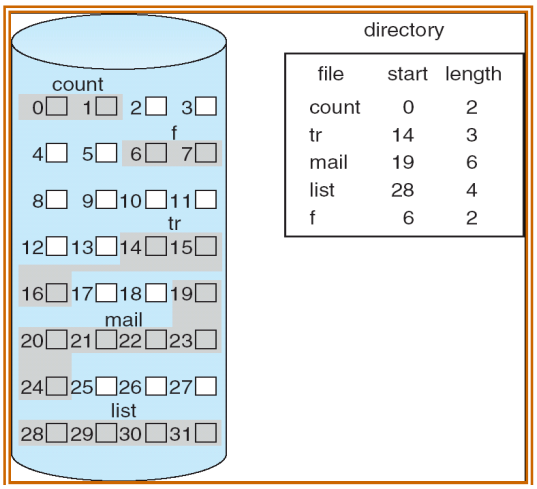
Description

An allocation method refers to how disk blocks are allocated for files. There are commonly used allocation methods as follows as,

1. **Contiguous allocation**
2. **Linked allocation**
3. **Indexed allocation**

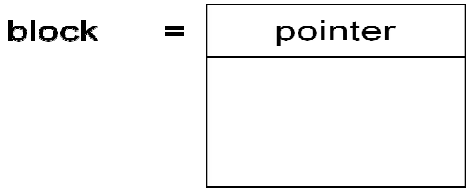
Contiguous Allocation

1. Each file occupies a set of contiguous blocks on the disk.
2. Simple – only starting location (block #) and length (number of blocks) are required.
3. Random access.
4. Wasteful of space (dynamic storage-allocation problem).
5. Files cannot grow.

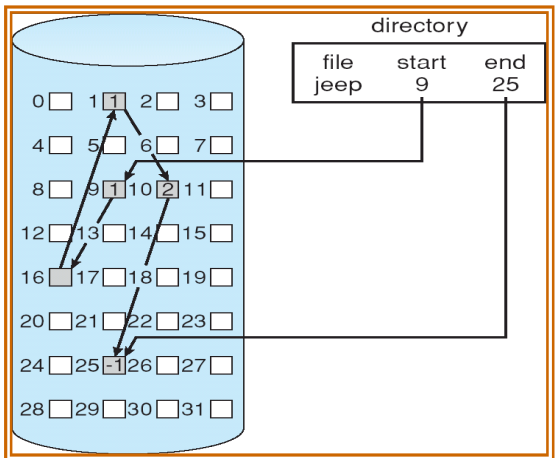


Linked allocation

Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.



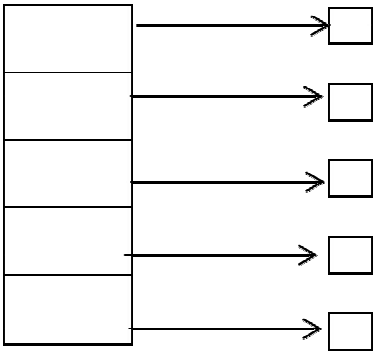
1. Simple – need only starting address
2. Free-space management system – no waste of space
3. No random access
4. Mapping



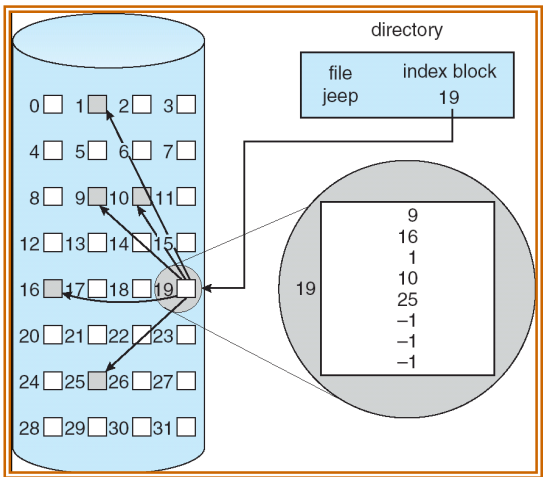
Indexed allocation

(i) Brings all pointers together into the *index block.*

(ii) Logical view.



Index table



1. Need index table
2. Random access
3. Dynamic access without external fragmentation, but have overhead of index block.
4. Mapping from logical to physical in a file of maximum size of 256K words and block size of 512 words. We need only 1 block for index table.

1.UNIX SYSTEM CALLS

1. Display parent id & process id

```
#include<stdio.h>
int main()
{
printf("\n Parent Process ID %d",getppid());
printf("\n Child Process ID %d\n",getpid());
}
```

2. Process creation using fork

```
#include<stdio.h>
main()
{
printf("Before FORK \n");
fork();
printf("After FORK \n\n");
}
```

3. Process with fork

```
#include<stdio.h>
main()
{
int pid;
pid=fork();
if(pid>0)
{
printf("From Parent \n");
printf("Parent process id %d\n",getpid());
}
else
{
printf("From Child \n");
printf("Child process id %d\n",getpid());
}
}
```

4. Making child as orphan*

```
#include<stdio.h>
main()
{
int pid,pid1;
pid=fork();
if(pid>0)
{
printf("From parent process\n");
printf("Parent process %d \n",getpid());
}
else
{
sleep(1);
printf("From child process\n");
printf("child process %d \n",getpid());
}}
```

5. Parent waits till completion of child

```
#include<stdio.h>
main()
{
int pid;
pid=fork();
printf("%d\n",pid);
if(pid==0)
{
printf("From child process \n");
}
else
{
wait(0);
printf("From parent process\n");
}
}
```

6. Use of exit system call

```
#include<stdio.h>
main()
{
int pid;
pid=fork();
printf("%d\n",pid);
if(pid<0)
{
perror("Child can't be executed\n");
exit(-1);
}
else
{
printf("Child created\n");
exit(0);
}
}
```

7. Using fork and exec system call

```
#include<stdio.h>
main()
{
int pid;
pid=fork();
printf("%d\n",pid);
if(pid==0)
{
execve("/bin/date\n",NULL,NULL);
exit(0); }
else
{
printf("Parent process %d\n",pid);   }}
```

8. To list files in specified directory

```
#include<stdio.h>
#include<dirent.h>
main()
{
char d[10];
DIR *p;
struct dirent *d1;
printf("ENTER A DIRECTORY NAME: ");
scanf("%s",d);
p=opendir(d);
if(p==NULL)
{
perror("Can't find directory");
exit(-1);
}
while(d1=readdir(p))
printf("%s\n",d1->d_name);
}
```

9. To list files in current directories & sub-directories

```
#include<stdio.h>
#include<dirent.h>
main()
{
DIR *p,*sp;
struct dirent *d,*dd;
p=opendir(".");
while(d=readdir(p))
{
printf("%s \n",d->d_name);
if(!strcmp(d->d_name,".")||!strcmp(d-
>d_name,".."))
continue;
sp=opendir(d->d_name);
if(sp)
while(dd=readdir(sp))
printf("--> %s\n",dd->d_name);
}
}
```

10. Create process & display pid of parent & child

```
#include<stdio.h>
#include<dirent.h>
main(int argc,char **argv)
{
int pid,i;
for(i=0;i<atoi(argv[1]);i++)
{
pid=fork();
if(pid==0)
{
printf("child process id %d    Parent process
id %d\n",getpid(),getppid());
}}}
```

11. Program to rename a directory.

```
#include<stdio.h>
main()
{
char s[10],d[10];
printf("Enter source Dir Name:\n");
scanf("%s",s);
printf("Enter New Dir Name:\n");
scanf("%s",d);
if(rename(s,d)==-1)
perror("Can't be changed\n");
else
printf("%s is changed to %s\n\n",s,d);
}
```

2.I/O SYSTEM CALLS

1. Write a program to open , read and write files using system calls.

```
#include<stdio.h>
#include<fcntl.h>
main()
{
char buff[100],fn[10];
int fd,n;
printf("Enter file name\n");
scanf("%s",fn);
fd=open(fn,O_RDONLY);
n=read(fd,buff,100);
n=write(1,buff,n);
close(fd);
}
```

2. Using system calls write line of texts in a file

```
#include<stdio.h>
#include<fcntl.h>
#include<string.h>
main()
{
char *buff,fn[10];
int fd,n,i;
printf("\nEnter file name ");
scanf("%s",fn);
printf("\nEnter text ");
scanf("%s",buff);
//fgets(buff,100,stdin);
fd=open(fn,O_CREAT|O_WRONLY);
n=write(fd,buff,strlen(buff));
close(fd);
}
```

3. Write a program to open, read and write files and perform file copy operation.

```
#include<stdio.h>
#include<fcntl.h>
main()
{
char buff[1000],fn1[10],fn2[10];
int fd1,fd2,n;
printf("Enter source filename\n");
scanf("%s",fn1);
printf("Enter destination file name\n");
scanf("%s",fn2);
fd1=open(fn1,O_RDONLY);
n=read(fd1,buff,1000);
fd2=open(fn2,O_CREAT|O_WRONLY);
n=write(fd2,buff,n);
close(fd1);
close(fd2);
}
```

4. Write a program to remove a directory.

```
#include<stdio.h>
#include<fcntl.h>
main()
{
char fn[10];
printf("Enter source filename\n");
scanf("%s",fn);
if(remove(fn)==0)
printf("File removed\n");
else
printf("File cannot be removed\n");
}
```

3.SIMULATION OF UNIX COMMANDS

1. Write a program for the simulation of ls command.

```
#include<stdio.h>
#include<dirent.h>
main()
{
char dirname[10];
DIR*p;
struct dirent *d;
printf("Enter directory name\n");
scanf("%s",dirname);
p=opendir(dirname);
if(p==NULL)
{
perror("Cannot find directory");
exit(-1);
}
while(d=readdir(p))
printf("%s\n",d->d_name);
}
```

2. Write a program for the simulation for grep command.

```
#include<stdio.h>
#include<string.h>
main()
{
char fn[10],pat[10],temp[200];
FILE *fp;
printf("Enter file name\n");
scanf("%s",fn);
printf("Enter pattern to be searched\n");
scanf("%s",pat);
fp=fopen(fn,"r");
while(!feof(fp))
{
fgets(temp,1000,fp);
if(strstr(temp,pat))
printf("%s",temp);
}
fclose(fp);
}
```

4. CPU SCHEDULING ALGORITHMS I

FIRST COME FIRST SERVED (FCFS)

Program:

```
#include<stdio.h>
struct process
{
int burst,wait;
}p[20]={0,0};
int main()
{
int n,i,totalwait=0,totalturn=0;
printf("\nEnter The No Of Process  :");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("Enter The Burst Time (in ms) For Process #%2d :",i+1);
scanf("%d",&p[i].burst);
}
printf("\nProcess \t Waiting Time  TurnAround Time ");
printf("\n        \t   (in ms)        (in ms)");
for(i=0;i<n;i++)
{
printf("\nProcess # %-12d%-15d%-15d",i+1,p[i].wait,p[i].wait+p[i].burst);
p[i+1].wait=p[i].wait+p[i].burst;
totalwait=totalwait+p[i].wait;
totalturn=totalturn+p[i].wait+p[i].burst;
}
printf("\n\nAVERAGE\n--------- ");
printf("\nWaiting    Time : %f ms",totalwait/(float)n);
printf("\nTurnAround Time : %f ms\n\n",totalturn/(float)n);
return 0;
}
```

Output:

```
Enter The No Of Process  :3
Enter The Burst Time (in ms) For Process # 1 :10
Enter The Burst Time (in ms) For Process # 2 :30
Enter The Burst Time (in ms) For Process # 3 :20

Process         Waiting Time  TurnAround Time
                  (in ms)        (in ms)
Process # 1          0             10
Process # 2          10             40
Process # 3          40             60

AVERAGE
---------
Waiting    Time : 16.666667 ms
TurnAround Time : 36.666667 ms
```

SHORTEST JOB FIRST(SJF)

Program:

```
#include<stdio.h>
struct process{
int burst,wait,no;
}p[20]={0,0};
int main(){
int n,i,j,totalwait=0,totalturn=0;
printf("\nEnter The No Of Process  :");
scanf("%d",&n);
for(i=0;i<n;i++){
printf("Enter The Burst Time (in ms) For Process #%2d :",i+1);
scanf("%d",&p[i].burst);
p[i].no=i+1;
}
for(i=0;i<n;i++)
for(j=0;j<n-i-1;j++)
if(p[j].burst>p[j+1].burst){
p[j].burst^=p[j+1].burst^=p[j].burst^=p[j+1].burst;
p[j].no^=p[j+1].no^=p[j].no^=p[j+1].no;
}
printf("\nProcess \t Waiting Time  TurnAround Time ");
for(i=0;i<n;i++){
printf("\nProcess # %-12d%-15d%-15d",p[i].no,p[i].wait,p[i].wait+p[i].burst);
p[i+1].wait=p[i].wait+p[i].burst;
totalwait=totalwait+p[i].wait;
totalturn=totalturn+p[i].wait+p[i].burst;
}
printf("\n\nAverage\n---------");
printf("\nWaiting    Time : %f ms",totalwait/(float)n);
printf("\nTurnAround Time : %f ms\n\n",totalturn/(float)n);
return 0;
}
```

Output:

```
Enter The No Of Process  :3
Enter The Burst Time (in ms) For Process # 1 :20
Enter The Burst Time (in ms) For Process # 2 :30
Enter The Burst Time (in ms) For Process # 3 :10

Process         Waiting Time  TurnAround Time
Process # 3         0             10
Process # 1         10            30
Process # 2         30            60

Average
---------
Waiting    Time : 13.333333 ms
TurnAround Time : 33.333333 ms
```

5.CPU SCHEDLING ALGORITHMS II

PRIORITY SCHEDULING

Program:

```
#include<stdio.h>
struct process{
int burst,wait,no,priority;
}p[20]={0,0};
int main(){
int n,i,j,totalwait=0,totalturn=0;
printf("\nEnter The No Of Process  :");
scanf("%d",&n);
for(i=0;i<n;i++){
printf("Enter The Burst Time (in ms) For
Process #%2d :",i+1);
scanf("%d",&p[i].burst);
printf("Enter The Priority          For Process
#%2d :",i+1);
scanf("%d",&p[i].priority);
p[i].no=i+1;
}
for(i=0;i<n;i++)
for(j=0;j<n-i-1;j++)
if(p[j].priority>p[j+1].priority){
p[j].burst^=p[j+1].burst^=p[j].burst^=p[j+1]
.burst;
p[j].no^=p[j+1].no^=p[j].no^=p[j+1].no;
//Simple way to swap 2 var's
p[j].priority^=p[j+1].priority^=p[j].priority^
=p[j+1].priority;
//printf("j");
}
printf("\nProcess \t Starting   Ending
Waiting   TurnAround  ");
printf("\n       \t  Time       Time    Time
Time  ");
for(i=0;i<n;i++){
printf("\nProcess # %-11d%-10d%-10d%-
10d%10d",p[i].no,p[i].wait,p[i].wait+p[i].bu
rst,p[i].wait,p[i].wait+p[i].burst);
p[i+1].wait=p[i].wait+p[i].burst;
totalwait=totalwait+p[i].wait;
totalturn=totalturn+p[i].wait+p[i].burst;
}
printf("\n\nAverage\n---------");
printf("\nWaiting    Time : %f
ms",totalwait/(float)n);
printf("\nTurnAround Time : %f
ms\n\n",totalturn/(float)n);
return 0;
}
```

Output:

```
Enter The No Of Process  :3
Enter The Burst Time (in ms) For Process #
1 :30
Enter The Priority          For Process # 1 :2
Enter The Burst Time (in ms) For Process #
2 :20
Enter The Priority          For Process # 2 :1
Enter The Burst Time (in ms) For Process #
3 :40
Enter The Priority          For Process # 3 :3

Process         Starting   Ending  Waiting
TurnAround
                Time       Time    Time       Time
Process # 2         0        20       0        20
Process # 1         20       50       20       50
Process # 3         50       90       50       90

Average
---------
Waiting    Time : 23.333333 ms
TurnAround Time : 53.333333 ms
```

ROUND ROBIN SCHEDULING

Program:

```
#include<stdio.h>
struct process{
int burst,wait,comp,f;
}p[20]={0,0};
int main(){
int
n,i,j,totalwait=0,totalturn=0,quantum,flag=1,
time=0;
printf("\nEnter The No Of Process :");
scanf("%d",&n);
printf("\nEnter The Quantum time (in ms)
:");
scanf("%d",&quantum);
for(i=0;i<n;i++){
printf("Enter The Burst Time (in ms) For
Process #%2d :",i+1);
scanf("%d",&p[i].burst);
p[i].f=1;
}
printf("\nOrder Of Execution \n");
printf("\nProcess Starting Ending
Remaining");
printf("\n Time Time Time");
while(flag==1){
flag=0;
for(i=0;i<n;i++){
if(p[i].f==1){
flag=1;
j=quantum;
if((p[i].burst-p[i].comp)>quantum){
p[i].comp+=quantum;
}
else{
p[i].wait=time-p[i].comp;
j=p[i].burst-p[i].comp;
p[i].comp=p[i].burst;
p[i].f=0;
}
printf("\nprocess # %-3d %-10d %-10d
%-10d",i+1,time,time+j,p[i].burst-
p[i].comp);
time+=j;
}}}
printf("\n\n----------");
printf("\nProcess \t Waiting Time
TurnAround Time ");
for(i=0;i<n;i++){
printf("\nProcess # %-12d%-15d%-
15d",i+1,p[i].wait,p[i].wait+p[i].burst);
totalwait=totalwait+p[i].wait;
totalturn=totalturn+p[i].wait+p[i].burst;
}
printf("\n\nAverage\n----------");
printf("\nWaiting Time : %f
ms",totalwait/(float)n);
printf("\nTurnAround Time : %f
ms\n\n",totalturn/(float)n);
return 0;
}
```

Output:

```
Enter The No Of Process :3
Enter The Quantum time (in ms) :5
Enter The Burst Time (in ms) For Process #
1 :25
Enter The Burst Time (in ms) For Process #
2 :30
Enter The Burst Time (in ms) For Process #
3 :54
Order Of Execution
Process      Starting     Ending     Remaining
               Time        Time        Time
process # 1    0           5           20
process # 2    5           10          25
process # 3    10          15          49
process # 1    15          20          15
process # 2    20          25          20
process # 3    25          30          44
process # 1    30          35          10
process # 2    35          40          15
process # 3    40          45          39
process # 1    45          50          5
process # 2    50          55          10
process # 3    55          60          34
process # 1    60          65          0
process # 2    65          70          5
process # 3    70          75          29
process # 2    75          80          0
process # 3    80          85          24
process # 3    85          90          19
process # 3    90          95          14
process # 3    95          100         9
process # 3    100         105         4
process # 3    105         109         0
Process      Waiting Time  TurnAround Time
Process # 1          40            65
Process # 2          50            80
Process # 3          55            109
Average
Waiting Time : 48.333333 ms
TurnAround Time : 84.666667 ms
```

7.PRODUCER CONSUMER PROBLEM USING SEMAPHORE

```
#include<stdio.h>    /*standard I/O
routines*/
#include<stdlib.h>   /*rand() and strand()
function*/
#include<unistd.h>  /*fork(),etc*/

#include<time.h>   /*nanosleep,etc*/

#include<sys/types.h>/*various type
definitions*/
#include<sys/ipc.h>/* general SysV IPC
structures */
#include<sys/sem.h>/* semephore fuctions
and structs*/
#define NUM_LOOPS 10 /* number of
loops to perform*/
int main (int argc,char * argv[])
{
int sem_set_id; /*ID of the semephore set*/
int child_pid;/*PID of our chiold process*/
int i;/*counter for loop operation*/
struct sembuf sem_op;/*structure for
semaphoreops.*/
int rc;/*return value of system calls*/
struct timespec delay;/*used for wasting
time*/
/* create a privete semaphore set with one
semaphore in it */
/* with access only to the owner */
sem_set_id=semget(IPC_PRIVATE,1,0600)
;
printf("semaphore set created,semaphore set
id'%d'\n",sem_set_id);
rc=semctl(sem_set_id,0,SETVAL,0);
/* fork-off a child process and start a
producer/consumer job*/
child_pid=fork();
switch(child_pid)
{
case 0: /*child process here */
for(i=0;i<NUM_LOOPS;i++)
 /* block on the semaphore,unless its value
is non negative*/
{
sem_op.sem_num=0;
sem_op.sem_op=-1;
sem_op.sem_flg=0;
semop(sem_set_id,&sem_op,1);
printf("consumer:'%d'\n",i);
fflush(stdout);
}
break;
default: /* parent process here*/
for(i=0;i<NUM_LOOPS;i++)
{
printf("producer:'%d'\n",i);
fflush(stdout);
/* increase the value of the semaphore by
1*/
sem_op.sem_num=0;
sem_op.sem_op=1;
sem_op.sem_flg=0;
semop(sem_set_id,&sem_op,1);
/*pause execution for a little bit,to allow the
child process to run  and handle some
request */
/* this is done about 25% of the time*/
if(rand()>3*(RAND_MAX/4))
{
delay.tv_sec=0;
delay.tv_nsec=10;
nanosleep(&delay,NULL);
}
}
break;
}
return 0;
}
```

8.MEMORY MANAGEMENT SCHEMES I

FIRST FIT

Program:

```
#include<stdio.h>
struct process{
int size;
char name[20];
int id;
}p[20]={0,0};
struct block{
int size;
int id;
}b[20]={0,0};
int main(){
int
nb,np,i,j,totalwait=0,totalturn=0,quantum=4,
flag=1,time=0;
printf("\nEnter The No Of Blocks :");
scanf("%d",&nb);
for(i=0;i<nb;i++){
printf("Enter The Size of Block $ %-3d
:",i+1);
scanf("%d",&b[i].size);
}
printf("\nEnter The No Of Processes
:");
scanf("%d",&np);
for(i=0;i<np;i++){
printf("Enter The Name of process # %-3d
:",i+1);
scanf("%s",p[i].name);
printf("Enter The Size of process # %-3d
:",i+1);
scanf("%d",&p[i].size);
}
for(i=0;i<np;i++){
for(j=0;j<nb;j++){
if(b[j].id==0&&p[i].size<=b[j].size){
b[j].id=i+1;
p[i].id=j+1;
flag=1;
break;
}}
}
printf("Block \n\n-----------");
printf("\nBlock ID   Block_Size
Process_Name Process_Size");
for(i=0;i<nb;i++){
if(b[i].id)
printf(" \nBlock #%-7d%-10d%-10s%-10d
",i+1,b[i].size,p[b[i].id-1].name, p[b[i].id-
1].size);
else
printf(" \nBlock #%-7d%-10dEmpty
Empty   ",i+1,b[i].size);
}
printf("\n\nProcess \n-----------");
printf("\nProcess_Name  Process_Size
Block ID    Block_Size");
for(i=0;i<np;i++){
if(p[i].id)
printf(" \nProcess $ %-10s%-10d%-10d%-
10d ",p[i].name,p[i].size,p[i].id, b[p[i].id-
1].size);
else
printf(" \nProcess $ %-10s%-10dWaiting
Waiting       ",p[i].name,p[i].size);
}
printf("\n");
}
```

Output:

```
Enter The No Of Blocks        :3
Enter The Size of Block $ 1   :30
Enter The Size of Block $ 2   :20
Enter The Size of Block $ 3   :10
Enter The No Of Processes       :3
Enter The Name of process # 1  :P
Enter The Size of process # 1     :10
Enter The Name of process # 2  :Q
Enter The Size of process # 2     :30
Enter The Name of process # 3  :R
Enter The Size of process # 3     :20
Block
-----------
Block ID   Block_Size  Process_Name
Process_Size
Block #1     30       P        10
Block #2     20       R        20
Block #3     10       Empty    Empty

Process
-----------
Process_Name  Process_Size  Block ID
Block_Size
Process $ P       10       1        30
Process $ Q       30       Waiting  Waiting
Process $ R       20       2        20
```

BEST FIT

Program:

```
#include<stdio.h>
#include<conio.h>
struct process{
int size;
int id;
}p[20]={0,0};
struct block{
int no;
int size;
int id;
}b[20]={0,0};
int main(){
int
nb,np,i,j,totalwait=0,totalturn=0,quantum=4,
flag=1,time=0;
clrscr();
printf("\nEnter The No Of Blocks :");
scanf("%d",&nb);
for(i=0;i<nb;i++){
printf("Enter The Size of Block $ %-3d
:",i+1);
scanf("%d",&b[i].size);
b[i].no=i+1;
}
printf("\nEnter The No Of Processes
:");
scanf("%d",&np);
for(i=0;i<np;i++){
printf("Enter The Size of process # %-3d
:",i+1);
scanf("%d",&p[i].size);
}
for(i=0;i<nb;i++)
for(j=0;j<nb-i-1;j++)
if(b[j].size>b[j+1].size){
b[j].size^=b[j+1].size^=b[j].size^=b[j+1].siz
e;
b[j].no^=b[j+1].no^=b[j].no^=b[j+1].no;
}

for(i=0;i<np;i++){
for(j=0;j<nb;j++){
if(b[j].id==0&&p[i].size<=b[j].size){
b[j].id=i+1;
p[i].id=b[j].no;
flag=1;
break;
}}}
printf("Block \n\n-----------");
printf("\nBlock_ID  Block_Size
Process_ID Process_Size");
for(i=0;i<nb;i++)
//for(j=0;j<nb;j++)
{
//if(j+1==b[j].no){
if(b[i].id)
printf(" \nBlock #%-7d%-10d%-10d%-10d
",b[i].no,b[i].size,b[i].id, p[b[i].id-1].size);
else
printf(" \nBlock #%-7d%-10dEmpty
Empty   ",b[i].no,b[i].size);
}
//}
printf("\n\nProcess \n-----------");
printf("\nProcess_ID  Process_Size
Block_ID    Block_Size");
for(i=0;i<np;i++){
if(p[i].id)
printf(" \nProcess $ %-3d%-14d%-10d%-
10d ",i+1,p[i].size,p[i].id, b[p[i].id-1].size);
else
printf(" \nProcess $ %-3d%-14dWaiting
Waiting       ",i+1,p[i].size);
}
printf("\n");
getch(); }
```

Output:

```
Enter The No Of Blocks        :3
Enter The Size of Block $ 1   :30
Enter The Size of Block $ 2   :20
Enter The Size of Block $ 3   :10
Enter The No Of Processes       :3
Enter The Size of process # 1     :10
Enter The Size of process # 2     :20
Enter The Size of process # 3     :30
Block
-----------
Block_ID  Block_Size  Process_ID
Process_Size
Block #3    10        1         10
Block #2    20        2         20
Block #1    30        3         30
Process
-----------
Process_ID  Process_Size  Block_ID
Block_Size
Process $ 1  10           3         30
Process $ 2  20           2         20
Process $ 3  30           1         10
```