# JERUSALEM COLLEGE OF ENGINEERING

**(An Autonomous Institution)**

**(Approved by AICTE, Affliated to Anna University**

**Accredited by NBA and NAAC with 'A' Grade) Velachery Main Road, Narayanapuram, Pallikaranai, Chennai – 600 100**


# OPERATING SYSTEM LABORATORY
# [JCS1412]

## RECORD NOTE BOOK

## ACADEMIC YEAR 2021 -2022

## II YEAR/ IV SEMESTER

## REGULATION 2019

## DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

## VISION OF THE DEPARTMENT

The Department of computer science and engineering is dedicated to be a center of excellence, in producing graduates as ethical engineers, innovative researchers, dynamic entrepreneurs and globally competitive technocrats.

## MISSION OF THE DEPARTMENT

- To craft students to be competent professionals with value based education, innovative teaching and practices.
- To enhance student‘s soft skill, personality and ethical responsibilities by augmenting in- plant training, value added courses and co curricular activities.
- To facilitate the student as researchers by widening their professional knowledge through continuous learning and innovative projects.
- To produce dynamic entrepreneur through interaction with network of alumni industry and academia and extracurricular activities.

## PROGRAM EDUCATIONAL OBJECTIVES (PEOS)

- **PEO1:**Graduates will apply engineering basics, laboratory and job oriented experiences to devise and unravel engineering problems in computer science engineering domain.
- **PEO2:**Graduates will be multi faceted researcher and experts in fields like computing, networking, artificial intelligence, software engineering and data science.
- **PEO3:**Graduates will be dynamic entrepreneur and service oriented professional with ethical and social responsibility.
- **PEO4:**Graduates will ingress and endure in core and other prominent organization across the globe and will foster innovation

## PROGRAM SPECIFIC OBJECTIVES (PSOS)

**PSO-I:** The ability to understand, analyze and to develop the design related to real-time system such as IOT, Secured automated systems, machine vision , computer vision and cognitive computing with various complexities , providing orientation towards green computing environment .

**PSO-II:** The ability to apply standard practices and strategies in software project development using open-ended programming environments to deliver a quality product.

**PSO-III:** The ability to innovate, introduce and produce socially relevant products to facilitate transformation of society into a digitally empowered knowledge economy, thereby to chart a successful career with a new dimension to entrepreneurship.

# JERUSALEM COLLEGE OF ENGINEERING

**(An Autonomous Institution)**
**(Approved by AICTE, Affiliated to Anna University,**
**Accredited by NBA and NAAC with 'A' Grade)**
Velachery Main Road, Narayanapuram, Pallikaranai, Chennai - 600100

**Name**……

**Year**……**Semester**……**Branch**…….

**Regulation:** ……

**Register No.**

Certified that this is a Bonafide Record work done by the above student in the ……..

…..…… Laboratory during the year 20 - 20 .

---------- ----------

**Signature of Lab. In charge** **Signature of Head of the Dept.**

**EXAMINERS**

**DATE:**_____

**INTERNAL EXAMINER** **EXTERNAL EXAMINER**

| JCS1412 | OPERATING SYSTEMS LABORATORY | L | T | P | C |
|---|---|---|---|---|---|
| | | 0 | 0 | 4 | 2 |

## SYLLABUS

### COURSE OBJECTIVES:

- To learn Unix commands and shell programming
- To implement various CPU Scheduling Algorithms
- To implement Process Creation and Inter Process Communication.
- To implement Deadlock Avoidance and Deadlock Detection Algorithms
- To implement Page Replacement Algorithms
- To implement File Organization and File Allocation Strategies

### LIST OF EXPERIMENTS

1. Basics of UNIX commands
2. Write programs using the following system calls of UNIX operating system fork, exec, getpid, exit, wait, close, stat, opendir, readdir
3. Write C programs to simulate UNIX commands like cp, ls, grep, etc.
4. Shell Programming
5. Write C programs to implement the various CPU Scheduling Algorithms
6. Implementation of Semaphores
7. Implementation of Shared memory and IPC
8. Bankers Algorithm for Deadlock Avoidance
9. Implementation of Deadlock Detection Algorithm
10. Write C program to implement Threading & Synchronization Applications
11. Implementation of the following Memory Allocation Methods for fixed partition
    a) First Fit b) Worst Fit c) Best Fit
12. Implementation of Paging Technique of Memory Management
13. Implementation of the following Page Replacement Algorithms
    a) FIFO b) LRU c) LFU
14. Implementation of the various File Organization Techniques
15. Implementation of the following File Allocation Strategies
    a) Sequential b) Indexed c) Linked

**TOTAL :60 PERIODS**

### COURSE OUTCOMES:

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

- Compare the performance of various CPU Scheduling Algorithms
- Implement Deadlock avoidance and Detection Algorithms
- Implement Semaphores
- Create processes and implement IPC
- Analyse the performance of the various Page Replacement Algorithms

Implement File Organization and File Allocation Strategies

# LIST OF EXPERIMENTS

## CYCLE 1

| S.No | NAME OF EXPERIMENTS |
|---|---|
| 1. | Basics of UNIX Commands |
| 2. | C Program for UNIX System Calls |
| 3. | C programs to simulate UNIX commands like cp, ls, grep |
| 4. | Simple Shell Programs |
| 5. | Implementation of CPU Scheduling Algorithms |
| 6. | Implementation of Semaphores |
| 7. | Implementation of Shared memory and IPC |

## CYCLE 2

| S.No | NAME OF EXPERIMENTS |
|---|---|
| 8 | Implementation of Deadlock Detection Algorithm |
| 9 | Implementation of Shared Memory and IPC |
| 10 | Implementation of Threading & Synchronization Applications |
| 11 | Implementation of Memory Allocation Methods For Fixed Partition |
| 12 | Implementation of Paging Technique Of Memory Management |
| 13 | Implementation of Page Replacement Algorithms |
| 14 | File Organization Technique |
| 15 | File Allocation Strategies |

# CONTENTS


| Ex.No | Date | Name of the Experiment | Page.No | Marks | Signature with Date |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |


**Average Marks :**

**Signature :**

# CONTENTS


| S.No | Date | Name of the Experiment | Page.No | Marks | Signature With Date |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |


**Average Marks :**

**Signature :**

| Exp. No. 1 | BASIC UNIX COMMANDS |
|---|---|
| Date : | |

**Aim**

To study and execute Unix commands.

**Login**

Type **telnet** *server_ipaddress* in **run** window.

User has to authenticate himself by providing *username* and *password*. Once verified, a greeting and **$** prompt appears. The shell is now ready to receive commands from the user. Options suffixed with a hyphen (–) and arguments are separated by space.

**General commands**

| Command | Function |
|---|---|
| Date | Used to display the current system date and time. |
| date +%D | Displays date only |
| date +%T | Displays time only |
| date +%Y | Displays the year part of date |
| date +%H | Displays the hour part of time |
| Cal | Calendar of the current month |
| cal*year* | Displays calendar for all months of the specified year |
| cal*month year* | Displays calendar for the specified month of the year |
| Who | Login details of all users such as their IP, Terminal No, User name, |
| who am i | Used to display the login details of the user |
| Uname | Displays the Operating System |
| uname –r | Shows version number of the OS (kernel). |
| uname –n | Displays domain name of the server |
| echo$HOME | Displays the user's home directory |
| Bc | Basic calculator. Press Ctrl+dto quit |
| lp *file* | Allows the user to spool a job along with others in a print queue. |
| man*cmdname* | Manual for the given command. Press qto exit |
| history | To display the commands used by the user since log on. |
| exit | Exit from a process. If shell is the only process then logs out |

**Directory commands**

| Command | Function |
|---|---|
| Pwd | Path of the present working directory |
| mkdir*dir* | A directory is created in the given name under the current directory |
| mkdir*dir1 dir2* | A number of sub-directories can be created under one stroke |
| cd*subdir* | Change Directory. If the *subdir* starts with / then path starts from **root** (absolute) otherwise from current working directory. |
| cd | To switch to the home directory. |
| cd / | To switch to the root directory. |
| cd .. | To move back to the parent directory |
| rmdir*subdir* | Removes an empty sub-directory. |

**File commands**

| Command | Function |
|---|---|
| cat >*filename* | To create a file with some contents. To end typing press Ctrl+d. The > symbol means redirecting output to a file. (< for input) |
| cat*filename* | Displays the file contents. |
| cat>>*filename* | Used to append contents to a file |
| cp*src des* | Copy files to given location. If already exists, it will be overwritten |
| cp–i *src des* | Warns the user prior to overwriting the destination file |
| cp –r *src des* | Copies the entire directory, all its sub-directories and files. |
| mv *old new* | To rename an existing file or directory. –ioption can also be used |
| mv *f1 f2 f3 dir* | To move a group of files to a directory. |
| mv –v *old new* | Display name of each file as it is moved. |
| rm *file* | Used to delete a file or group of files. –ioption can also be used |
| rm * | To delete all the files in the directory. |
| rm –r * | Deletes all files and sub-directories |
| rm –f * | To forcibly remove even write-protected files |
| Ls | Lists all files and subdirectories (blue colored) in sorted manner. |
| ls*name* | To check whether a file or directory exists. |
| ls*name***** | Short-hand notation to list out filenames of a specific pattern. |
| ls –a | Lists all files including hidden files (files beginning with **.**) |
| ls –x*dirname* | To have specific listing of a directory. |
| ls –R | Recursive listing of all files in the subdirectories |
| ls –l | Long listing showing file access rights (read/write/execute-**rwx** for user/group/others-**ugo**). |
| cmp *file1 file2* | Used to compare two files. Displays nothing if files are identical. |
| wc *file* | It produces a statistics of lines (**l**), words(**w**), and characters(**c**). |
| chmod *perm file* | Changes permission for the specified file. (r=4, w=2, x=1)<br>chmod 740 *file*sets all rights for user, read only for groupsand no rights for others |

The commands can be combined using the pipeline (|) operator. For example, number of users logged in can be obtained as.

who | wc -l

Finally to terminate the unix session execute the command **exit** or **logout**.

**Output**

**Result**

Thus the study and execution of Unix commands has been completed successfully.

| Exp. No. 2.A | FORK SYSTEM CALL |
|---|---|
| Date : | |

**Aim**

To create a new child process using fork system call.

**fork()**

- The fork system call is used to create a new process called *child* process.
  - The return value is 0 for a child process.
  - The return value is negative if process creation is unsuccessful.
  - For the parent process, return value is positive
- The child process is an exact copy of the parent process.
- Both the child and parent continue to execute the instructions following fork call.
- The child can start execution before the parent or vice-versa.

**getpid()and getppid()**

- The getpid system call returns process ID of the calling process
- The getppid system call returns parent process ID of the calling process

**Algorithm**

1. Declare a variable *x* to be shared by both child and parent.
2. Create a child process using fork system call.
3. If return value is -1 then
   Print "Process creation unsuccessfull"
   Terminate using exit system call.
4. If return value is 0 then
   Print "Child process"
   Print process id of the child using getpid system call
   Print value of *x*
   Print process id of the parent using getppid system call
5. Otherwise
   Print "Parent process"
   Print process id of the parent using getpid system call
   Print value of *x*
   Print process id of the shell using getppid system call.
6. Stop

**Program**

**/* Process creation - fork.c */#include**

**<stdio.h>**
**#include <stdlib.h>**

```
#include <unistd.h> #include
<sys/types.h>

main()
{
    pid_t    pid; int
    x = 5; pid =
    fork();x++;

    if (pid < 0)
    {
        printf("Process creation error");exit(-1);
    }
    else if (pid == 0)
    {
        printf("Child process:"); printf("\nProcess id is %d",
        getpid());printf("\nValue of x is %d", x);
        printf("\nProcess id of parent is %d\n", getppid());
    }
    else
    {
        printf("\nParent process:"); printf("\nProcess id is
        %d", getpid());printf("\nValue of x is %d", x);
        printf("\nProcess id of shell is %d\n", getppid());
    }
}
```

**Output**

**Result**

Thus a child process is created with copy of its parent's address space.

| Exp. No. 2.B | WAIT SYSTEM CALL |
|---|---|
| Date : | |

**Aim**

To block a parent process until child completes using wait system call.

**wait()**

- The wait system call causes the parent process to be blocked until a child terminates.
- When a process terminates, the kernel notifies the parent by sending the SIGCHLD signal to the parent.
- Without wait, the parent may finish first leaving a *zombie* child, to be adopted by init process

**Algorithm**

1. Create a child process using fork system call.
2. If return value is -1 then
   a. Print "Process creation unsuccessfull"
3. Terminate using exit system call.
4. If return value is > 0 then
   a. Suspend parent process until child completes using wait system call
   b. Print "Parent starts"
   c. Print even numbers from 0–10
   d. Print "Parent ends"
5. If return value is 0 then
   a. Print "Child starts"
   b. Print odd numbers from 0–10
   c. Print "Child ends"
6. Stop

**Program**

```
/* Wait for child termination - wait.c */#include <stdio.h>
#include <stdlib.h> #include
<unistd.h> #include
<sys/types.h>#include
<sys/wait.h>

main()
{
    int i, status;pid_t pid;

    pid = fork();
```

```
    if (pid < 0)
    {
        printf("\nProcess creation failure\n");exit(-1);
    }
    else if(pid > 0)
    {
        wait(NULL);
        printf ("\nParent starts\nEven Nos: ");for
        (i=2;i<=10;i+=2)
            printf ("%3d",i);
        printf ("\nParent ends\n");
    }
    else if (pid == 0)
    {
        printf ("Child starts\nOdd Nos: ");for
        (i=1;i<10;i+=2)
            printf  ("%3d",i); printf
        ("\nChild ends\n");
    }
}
```

**Output**

**Result**

Thus using wait system call zombie child processes were avoided.

| Exp. No. 2.B | EXEC SYSTEM CALL |
|---|---|
| Date : | |

**Date:**

**Aim**

To load an executable program in a child processes exec system call.

**execl()**

- The exec family of function (execl, execv, execle, execve, execlp, execvp) is used by the child process to load a program and execute.
- execl system call requires path, program name and null pointer

**Algorithm**

1. Create a child process using fork system call.
2. If return value is -1 then
   a. Print "Process creation unsuccessfull"
3. Terminate using exit system call.
4. If return value is > 0 then
   a. Suspend parent process until child completes using wait system call
   b. Print "Child Terminated".
   c. Terminate the parent process.
5. If return value is 0 then
   a. Print "Child starts"
   b. Load date program into child process using exec system call.
   c. Terminate the child process.
6. Stop

**Program**

```
/* Load a program in child process - exec.c */#include <stdio.h>
#include <stdlib.h> #include
<unistd.h> #include
<sys/types.h>

main()
{
    pid_t pid;

    switch(pid = fork())
    {
```

```
        case -1:
            perror("Fork  failed");exit(-1);

        case 0:
            printf("Child process\n");
            execl("/bin/date",  "date",  0);exit(0);

        default:
            wait(NULL);
            printf("Child  Terminated\n");exit(0);
    }
}
```

**Output**

**Result**

Thus the child process loads a binary executable file into its address space.

| Exp. No. 2.D | STAT SYSTEM CALL |
|---|---|
| Date : | |

**Aim**

To display file status using stat system call.

**exit()**

- The exit system call is used to terminate a process either normally or abnormally
- Closes all standard I/O streams.

**stat()**

- The stat system call is used to return information about a file as a structure.

**Algorithm**

1. Get *filename* as command line argument.
2. If *filename* does not exist then stop.
3. Call stat system call on the *filename* that returns a structure
4. Display members st_uid, st_gid, st_blksize, st_block, st_size, st_nlink, etc.,
5. Convert time members such as st_atime, st_mtime into time using ctime function
6. Compare st_mode with mode constants such as S_IRUSR, S_IWGRP, S_IXOTH anddisplay file permissions.
7. Stop

**Program**

```
/* File status - stat.c */#include
<stdio.h> #include <sys/stat.h>
#include <stdlib.h> #include
<time.h>

int main(int argc, char*argv[])
{
    struct stat file;int n;
    if (argc != 2)
    {
        printf("Usage: ./a.out <filename>\n");exit(-1);
    }
    if ((n = stat(argv[1], &file)) == -1)
    {
        perror(argv[1]);
        exit(-1);
    }
```

```
    printf("User id : %d\n", file.st_uid); printf("Group id : %d\n",
    file.st_gid); printf("Block size : %d\n", file.st_blksize); printf("Blocks
    allocated : %d\n", file.st_blocks);printf("Inode no. : %d\n",
    file.st_ino);
    printf("Last accessed : %s", ctime(&(file.st_atime)));printf("Last modified :
    %s", ctime(&(file.st_mtime)));printf("File size : %d bytes\n", file.st_size);
    printf("No. of links : %d\n", file.st_nlink);

    printf("Permissions : ");
    printf( (S_ISDIR(file.st_mode)) ? "d" : "-");
    printf( (file.st_mode & S_IRUSR) ? "r" : "-");
    printf( (file.st_mode & S_IWUSR) ? "w" : "-");
    printf( (file.st_mode & S_IXUSR) ? "x" : "-");
    printf( (file.st_mode & S_IRGRP) ? "r" : "-");
    printf( (file.st_mode & S_IWGRP) ? "w" : "-");
    printf( (file.st_mode & S_IXGRP) ? "x" : "-");
    printf( (file.st_mode & S_IROTH) ? "r" : "-");
    printf( (file.st_mode & S_IWOTH) ? "w" : "-");
    printf( (file.st_mode & S_IXOTH) ? "x" : "-");printf("\n");

    if(file.st_mode & S_IFREG) printf("File type :
        Regular\n");
    if(file.st_mode & S_IFDIR) printf("File type :
        Directory\n");
}
```

**Output**

**Result**

Thus attributes of a file is displayed using stat system call.

| Exp. No. 2.E | READ DIR SYSTEM CALL |
|---|---|
| Date : | |

**Aim**

To display directory contents using readdir system call.

**opendir(), readdir()and closedir()**

- The opendir system call is used to open a directory
  - It returns a pointer to the first entry
  - It returns NULL on error.
- The readdir system call is used to read a directory as a *dirent* structure
  - It returns a pointer pointing to the next entry in directory stream
  - It returns NULL if an error or end-of-file occurs.
- The closedir system call is used to close the directory stream
- Write to a directory is done only by the kernel**.**

**Algorithm**

1. Get directory *name* as command line argument.
2. If directory does not exist then stop.
3. Open the directory using opendir system call that returns a structure
4. Read the directory using readdir system call that returns a structure
5. Display d_name member for each entry.
6. Close the directory using closedir system call.
7. Stop

**Program**

```
/* Directory content listing - dirlist.c */

#include <stdio.h>
#include <dirent.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
    struct dirent *dptr;DIR
    *dname;

    if (argc != 2)
    {
        printf("Usage: ./a.out <dirname>\n");exit(-1);
    }

    if((dname = opendir(argv[1])) == NULL)
    {
        perror(argv[1]);
        exit(-1);
```

```
    }

    while(dptr=readdir(dname)) printf("%s\n",  dptr-
        >d_name);

    closedir(dname);
}
```

**Output**

**Result**

Thus files and subdirectories in the directory was listed that includes hidden files.

| Exp. No. 2.F | OPEN SYSTEM CALL |
|---|---|
| Date : | |

**Aim**

To create a file and to write contents.

**open()**

- Used to open an existing file for reading/writing or to create a new file.
- Returns a file descriptor whose value is negative on error.
- The mandatory flags are O_RDONLY, O_WRONLY and O_RDWR
- Optional flags include O_APPEND, O_CREAT, O_TRUNC, etc
- The flags are ORed.
- The mode specifies permissions for the file.

**creat()**

- Used to create a new file and open it for writing.
- It is replaced with open() with flags O_WRONLY|O_CREAT | O_TRUNC

**Algorithm**

1. Declare a character buffer *buf* to store 100 bytes.
2. Get the new filename as command line argument.
3. Create a file with the given name using open system call with O_CREAT and O_TRUNC options.
4. Check the file descriptor.
   a) If file creation is unsuccessful, then stop.
5. Get input from the console until user types Ctrl+D
   a) Read 100 bytes (max.) from console and store onto buf using read system call
   b) Write length of *buf* onto file using write system call.
6. Close the file using close system call.
7. Stop

**Program**

```
/* File creation - fcreate.c */

#include <stdio.h>
#include<stdlib.h>
#include<string.h>
#include <fcntl.h>

main(int argc, char *argv[])
{
    int fd, n, len;char
    buf[100];
```

```
    if (argc != 2)
    {
        printf("Usage: ./a.out <filename>\n");exit(-1);
    }

    fd = open(argv[1], O_WRONLY|O_CREAT|O_TRUNC, 0644);if(fd <
    0)
    {
        printf("File creation problem\n");exit(-1);
    }

    printf("Press Ctrl+D at end in a new line:\n");while((n = read(0, buf,
    sizeof(buf))) > 0)
    {
        len = strlen(buf); write(fd,
        buf, len);
    }
    close(fd);
}
```

**Output**

**Result**

Thus a file has been created with input from the user. The process can be verified by using cat command

| Exp. No. 2.G | READ SYSTEM CALL |
|---|---|
| Date : | |

**Aim**

To read the given file and to display file contents.

**read()**

- Reads no. of bytes from the file or from the terminal.
- If read is successful, it returns no. of bytes read.
- The file offset is incremented by no. of bytes read.
- If end-of-file is encountered, it returns 0.

**Algorithm**

1. Declare a character buffer *buf* to store 100 bytes.
2. Get existing filename as command line argument.
3. Open the file for reading using open system call with O_RDONLY option.
4. Check the file descriptor.
   a) If file does not exist, then stop.
5. Read until end-of-file using read system call.
   a) Read 100 bytes (max.) from file and print it
6. Close the file using close system call.
7. Stop

**Program**

```
/* File Read          - fread.c */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

main(int argc, char *argv[])
{
    int fd,i;
    char buf[100];if
    (argc < 2)
    {
        printf("Usage: ./a.out <filename>\n");exit(-1);
    }

    fd = open(argv[1], O_RDONLY);if(fd ==
    -1)
    {
        printf("%s file does not exist\n", argv[1]);exit(-1);
```

```
    }

    printf("Contents of the file %s is : \n", argv[1]);while(read(fd, buf,
    sizeof(buf)) > 0)
        printf("%s", buf);

    close(fd);
}
```

**Output**

**Result**

Thus the given file is read and displayed on the console. The process can be verified by using cat command.

| Exp. No. 2.H | WRITE SYSTEM CALL |
|---|---|
| Date : | |

**Aim**

To append content to an existing file.

**write()**

- Writes no. of bytes onto the file.
- After a successful write, file's offset is incremented by the no. of bytes written.
- If any error due to insufficient storage space, write fails.

**close()**

- Closes a opened file.
- When process terminates, files associated with the process are automatically closed.

**Algorithm**

1. Declare a character buffer *buf* to store 100 bytes.
2. Get exisiting filename as command line argument.
3. Create a file with the given name using open system call with O_APPEND option.
4. Check the file descriptor.
   a) If value is negative, then stop.
5. Get input from the console until user types Ctrl+D
   a) Read 100 bytes (max.) from console and store onto buf using read system call
   b) Write length of *buf* onto file using write system call.
6. Close the file using close system call.
7. Stop

**Program**

```
/* File append - fappend.c */ #include

<stdio.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>

main(int argc, char *argv[])
{
    int fd, n, len;char
    buf[100];

    if (argc != 2)
    {
        printf("Usage: ./a.out <filename>\n");exit(-1);
    }
```

```
    fd = open(argv[1], O_APPEND|O_WRONLY|O_CREAT, 0644);if (fd < 0)
    {
        perror(argv[1]);
        exit(-1);
    }

    while((n = read(0, buf, sizeof(buf))) > 0)
    {
        len = strlen(buf); write(fd,
        buf, len);
    }

    close(fd);
}
```

**Output**

**Result**

Thus contents have been written to end of the file. The process can be verified by using cat command.

| Exp. No. 3.A | WRITE C PROGRAMS TO SIMULATE UNIX COMMANDS<br>LS COMMAND |
|---|---|
| Date : | |

**Aim**

To simulate ls command using UNIX system calls.

**Algorithm**

1. Store path of current working directory using getcwd system call.
2. Scan directory of the stored path using scandir system call and sort the resultant array of structure.
3. Display dname member for all entries if it is not a hidden file.
4. Stop.

**Program**

```
/* ls command simulation - list.c */#include

<stdio.h>
#include <dirent.h>

main()
{
     struct dirent **namelist;int n,i;
     char pathname[100];

     getcwd(pathname);
     n = scandir(pathname, &namelist, 0, alphasort);

     if(n < 0)
          printf("Error\n");else
          for(i=0; i<n; i++)
               if(namelist[i]->d_name[0] != '.') printf("%-20s",
                    namelist[i]->d_name);
}
```

**Output**

**Result**

Thus the filenames/subdirectories are listed, similar to ls command

| Exp. No.3.B | GREPCOMMAND |
|---|---|
| Date : | |

**Aim**

To simulate grep command using UNIX system call.

**Algorithm**

1. Get filename and search string as command-line argument.
2. Open the file in read-only mode using open system call.
3. If file does not exist, then stop.
4. Let length of the search string be *n*.
5. Read line-by-line until end-of-file
   a. Check to find out the occurrence of the search string in a line by examining characters in the range 1–n, 2–n+1, etc.
   b. If search string exists, then print the line.
6. Close the file using close system call.
7. Stop.

**Program**

```
/* grep command simulation - mygrep.c */#include

<stdio.h>
#include <string.h>
#include <stdlib.h>

main(int argc,char *argv[])
{
    FILE *fd;
    char str[100];char c;
    int i, flag, j, m, k;char
    temp[30]; if(argc != 3)
    {
        printf("Usage: gcc mygrep.c –o mygrep\n"); printf("Usage: ./mygrep
        <search_text> <filename>\n");exit(-1);
    }

    fd = fopen(argv[2],"r");if(fd ==
    NULL)
    {
        printf("%s is not exist\n",argv[2]);
```

```
        exit(-1);
    }

    while(!feof(fd))
    {
        i = 0;
        while(1)
        {
            c = fgetc(fd);if(feof(fd))
            {
                str[i++] = '\0';break;
            }
            if(c == '\n')
            {
                str[i++] = '\0';break;
            }
            str[i++] = c;
        }
        if(strlen(str) >= strlen(argv[1]))
            for(k=0; k<=strlen(str)-strlen(argv[1]); k++)
            {
                for(m=0; m<strlen(argv[1]); m++)temp[m] =
                    str[k+m];
                temp[m] = '\0'; if(strcmp(temp,argv[1])
                == 0)
                {
                    printf("%s\n",str);break;
                }
            }
    }
}
```

**Output**

**Result**

Thus the program simulates grep command by listing lines containing the search text.

| Exp. No. 3.C | CP COMMAND |
|---|---|
| Date : | |

**Aim**

To simulate cp command using UNIX system call.

**Algorithm**

1. Get source and destination *filename* as command-line argument.
2. Declare a buffer of size 1KB
3. Open the source file in readonly mode using open system call.
4. If file does not exist, then stop.
5. Create the destination file using creat system call.
6. If file cannot be created, then stop.
7. File copy is achieved as follows:
   a. Read 1KB data from source file and store onto buffer using read system call.
   b. Write the buffer contents onto destination file using write system call.
   c. If end-of-file then step 8 else step 7a.
8. Close source and destination file using close system call.
9. Stop.

**Program**

```
/* cp command simulation - copy.c */#include

<stdio.h>
#include <stdlib.h> #include
<fcntl.h> #include
<sys/stat.h>

#define SIZE 1024

main(int argc, char *argv[])
{
    int src, dst, nread;char
    buf[SIZE];

    if (argc != 3)
    {
        printf("Usage: gcc copy.c -o copy\n"); printf("Usage: ./copy
        <filename> <newfile> \n");exit(-1);
    }

    if ((src = open(argv[1], O_RDONLY)) == -1)
```

```
    {
        perror(argv[1]);
        exit(-1);
    }

    if ((dst = creat(argv[2], 0644)) == -1)
    {
        perror(argv[1]);
        exit(-1);
    }

    while ((nread = read(src, buf, SIZE)) > 0)
    {
        if (write(dst, buf, nread) == -1)
        {
                printf("can't  write\n");exit(-1);
        }
    }

    close(src);
    close(dst);
}
```

**Output**

**Result**

Thus a file is copied using file I/O. The cmp command can be used to verify that contents of both file are same

| Exp. No. 3.D | RMCOMMAND |
|---|---|
| Date : | |

**Aim**

To simulate rm command using UNIX system call.

**Algorithm**

1. Get *filename* as command-line argument.
2. Open the file in read-only mode using read system call.
3. If file does not exist, then stop.
4. Close the file using close system call.
5. Delete the file using unlink system call.
6. Stop.

**Program**

```
/* rm command simulation - del.c */#include

<stdio.h>
#include <stdlib.h>
#include <fcntl.h>

main(int argc, char* argv[])
{
    int fd;

    if (argc != 2)
    {
        printf("Usage: gcc del.c -o del\n");printf("Usage:
        ./del <filename>\n");exit(-1);
    }

    fd = open(argv[1], O_RDONLY);if (fd
    != -1)
    {
        close(fd); unlink(argv[1]);
    }
    else
        perror(argv[1]);
}
```

**Output**

**Result**

Thus files can be deleted in a manner similar to rm command. The deletion of file can be verified by using ls command.

| Exp. No. 4 | SHELL PROGRAMMING |
|---|---|
| Date : | |

**Aim**

To write simple shell scripts using shell programming fundamentals.

The activities of a shell are not restricted to command interpretation alone. The shell also has rudimentary programming features. Shell programs are stored in a file (with extension .**sh**). Shell programs run in interpretive mode. The original UNIX came with the Bourne shell (**sh**) and it is universal even today. C shell (**csh**) and Korn shell (**ksh**) are also widely used. Linux offers Bash shell (**bash**) as a superior alternative to Bourne shell.

## Preliminaries

1. Comments in shell script start with **#**.
2. Shell variables are loosely typed i.e. not declared. Variables in an expression or output must be prefixed by **$**.
3. The **read**statement is shell's internal tool for making scripts interactive.
4. Output is displayed using **echo**statement.
5. Expressions are computed using the **expr**command. Arithmetic operators are + - * / %. Meta characters *** ( )** should be escaped with a \.
6. The shell scripts are executed

   **$** sh *filename*

**Decision-making**

Shell supports decision-making using **if** statement. The **if** statement like itscounterpart in programming languages has the following formats.

```
if [ condition ]then
    statements
fi
```

```
if [ condition ]then
    statements
else
    statements
fi
```

```
if [condition ]then
    statements
elif [ condition ]then
    statements
.. .
else
    statements
  fi
```

The set of relational operators are –eq –ne –gt –ge –lt –le and logical operators used in conditional expression are –a –o !

**Multi-way branching**

The case statement is used to compare a variables value against a set of constants. Ifit matches a constant, then the set of statements followed after ) is executed till a ;; is encountered. The optional *default* block is indicated by *****. Multiple constants can be specified in a single pattern separated by |.

```
case variable in
    constant1)
        statements ;;
    constant2)
        statements ;;
     . . .
        *)
        statements
esac
```

**Loops**

Shell supports a set of loops such as **for**, **while** and **until** to execute a set of statements repeatedly. The body of the loop is contained between **do** and **done** statement.

The **for** loop doesn't test a condition, but uses a list instead.

```
for variable in list
do
        statements
done
```

The **while** loop executes the *statements* as long as the condition remains true.

```
while [ condition ]do
        statements
done
```

The **until** loop complements the while construct in the sense that the *statements* are executed as long as the condition remains false.

```
until [ condition ]do
        statements
done
```

**A) Swapping values of two variables**

```
# Swapping values – swap.sh echo -n "Enter
value for A : "read a
echo -n         "Enter value for B : "read b
t=$a
a=$b
b=$t
echo "Values after Swapping"
echo "A Value is $a and B Value is $b"
```

**Output**

**B) Farenheit to Centigrade Conversion**

```
# Degree conversion – degconv.shecho -n
"Enter Fahrenheit : " read f
c=`expr \( $f - 32 \) \* 5 / 9`echo "Centigrade is :
$c"
```

**Output**

**C) Biggest of 3 numbers**

```
# Biggest – big3.sh
echo -n "Give value for A B and C: "read a b c
if [ $a -gt $b -a $a -gt $c ]then
      echo "A is the Biggest number"elif [ $b -gt
$c ]
then
      echo "B is the Biggest number"else
      echo "C is the Biggest number"
fi
```

**Output**

**D) Grade Determination**

```
# Grade – grade.sh
echo -n "Enter the mark : "read mark
if [ $mark -gt 90 ]then
      echo "S Grade" elif [
$mark -gt 80 ]then
      echo "A Grade" elif [
$mark -gt 70 ]then
      echo "B Grade" elif [
$mark -gt 60 ]then
      echo "C Grade" elif [
$mark -gt 55 ]then
      echo "D Grade" elif [
$mark -ge 50 ]then
      echo "E Grade"else
      echo "U Grade"
fi
```

**Output**

**E) Vowel or Consonant**

```
# Vowel       - vowel.sh
echo -n "Key in a lower case character : "read choice
case $choice in
         a|e|i|o|u) echo "It's a Vowel";;
                        *) echo "It's a Consonant"
esac
```

**Output**

**F) Simple Calculator**

```
# Arithmetic operations — calc.sh echo -n "Enter
the two numbers : "read a b
echo " 1. Addition" echo " 2.
Subtraction"
echo " 3. Multiplication"echo " 4.
Division"
echo -n "Enter the option : "read option
case $option in
    1) c=`expr $a + $b` echo "$a +
       $b = $c";;
    2) c=`expr $a - $b` echo "$a -
       $b = $c";;
    3) c=`expr $a \* $b` echo "$a * $b
       = $c";;
    4) c=`expr $a / $b` echo "$a / $b
       = $c";;
    *) echo "Invalid Option"esac
```

**Output**

**G) Multiplication Table**

```
# Multiplication table – multable.shclear
echo -n "Which multiplication table? : "read n
for x in 1 2 3 4 5 6 7 8 9 10do
    p=`expr $x \* $n`
    echo -n "$n X $x = $p"sleep 1
done
```

**Output**

**H) Number Reverse**

```
# To reverse a number – reverse.shecho -n "Enter
a number : "
read n
rd=0
while [ $n -gt 0 ]do
      rem=`expr $n % 10` rd=`expr $rd \*
      10 + $rem`n=`expr $n / 10`
done
echo "Reversed number is $rd"
```

**Output**

**I) Prime Number**

```
# Prime number – prime.sh echo -n "Enter
the number : "read n
i=2
m=`expr $n / 2` until [ $i -gt
$m ]do
      q=`expr $n % $i`if [ $q
      -eq 0 ] then
            echo "Not a Prime number"exit
      fi
      i=`expr $i + 1`done
echo "Prime number"
```

**Output**

$ **sh prime.sh**

**Result**

Thus shell scripts were executed using different programming constructs

| Exp. No. 5.A | FCFS SCHEDULING |
|---|---|
| Date : | |

**Aim**

To schedule snapshot of processes queued according to FCFS scheduling.

**Process Scheduling**

- CPU scheduling is used in multiprogrammed operating systems.
- By switching CPU among processes, efficiency of the system can be improved.
- Some scheduling algorithms are FCFS, SJF, Priority, Round-Robin, etc.
- Gantt chart provides a way of visualizing CPU scheduling and enables to understand better.

**First Come First Serve (FCFS)**

- Process that comes first is processed first
- FCFS scheduling is non-preemptive
- Not efficient as it results in long average waiting time.
- Can result in starvation, if processes at beginning of the queue have long bursts.

**Algorithm**

1. Define an array of structure *process* with members *pid*, *btime*, *wtime* & *ttime*.
2. Get length of the ready queue, i.e., number of process (say *n*)
3. Obtain *btime* for each process.
4. The *wtime* for first process is 0.
5. Compute *wtime* and *ttime* for each process as:
   a. $wtime_{i+1} = wtime_i + btime_i$
   b. $ttime_i = wtime_i + btime_i$
6. Compute average waiting time *awat* and average turnaround time *atur*
7. Display the *btime*, *ttime* and *wtime* for each process.
8. Display GANTT chart for the above scheduling
9. Display *awat* time and *atur*
10. Stop

**Program**

```
/* FCFS Scheduling           - fcfs.c */

#include <stdio.h>

struct process
{
     int pid; int
     btime;int
     wtime;int
     ttime;
} p[10];

main()
{
     int  i,j,k,n,ttur,twat;float
     awat,atur;

     printf("Enter no. of process : ");scanf("%d",
     &n);
     for(i=0; i<n; i++)
     {
          printf("Burst time for process P%d (in ms) : ",(i+1));scanf("%d", &p[i].btime);
          p[i].pid = i+1;
     }

     p[0].wtime = 0; for(i=0;
     i<n; i++)
     {
          p[i+1].wtime = p[i].wtime + p[i].btime;p[i].ttime =
          p[i].wtime + p[i].btime;
     }
     ttur = twat = 0; for(i=0;
     i<n; i++)
     {
          ttur += p[i].ttime;twat +=
          p[i].wtime;
     }
     awat = (float)twat / n;atur =
     (float)ttur / n;

     printf("\n                FCFS  Scheduling\n\n");
     for(i=0; i<28; i++)
          printf("-");
     printf("\nProcess B-Time T-Time W-Time\n");for(i=0; i<28;
     i++)
          printf("-");
```

```
    for(i=0; i<n; i++)
        printf("\n    P%d\t%4d\t%3d\t%2d",
                    p[i].pid,p[i].btime,p[i].ttime,p[i].wtime);
    printf("\n"); for(i=0; i<28;
    i++)
        printf("-");

    printf("\n\nAverage waiting time                    : %5.2fms", awat);
    printf("\nAverage turn around time : %5.2fms\n", atur);

    printf("\n\nGANTT  Chart\n");
    printf("-");
    for(i=0; i<(p[n-1].ttime + 2*n); i++)printf("-");
    printf("\n");
    printf("|"); for(i=0; i<n;
    i++)
    {
        k = p[i].btime/2; for(j=0;
        j<k; j++)
            printf(" "); printf("P%d",p[i].pid);
        for(j=k+1; j<p[i].btime; j++)
            printf(" ");
        printf("|");
    }
    printf("\n");
    printf("-");
    for(i=0; i<(p[n-1].ttime + 2*n); i++)printf("-");
    printf("\n");
    printf("0"); for(i=0; i<n;
    i++)
    {
        for(j=0; j<p[i].btime; j++)printf(" ");
        printf("%2d",p[i].ttime);
    }
}
```

**Output**

---

---

---

---

---

**Result**

Thus waiting time & turnaround time for processes based on FCFS scheduling was computed and the average waiting time was determined.

| Exp. No. 5.B | SJF SCHEDULING |
|---|---|
| Date : | |

**Aim**

To schedule snapshot of processes queued according to SJF scheduling.

**Shortest Job First (SJF)**

- Process that requires smallest burst time is processed first.
- SJF can be preemptive or non–preemptive
- When two processes require same amount of CPU utilization, FCFS is used to break the tie.
- Generally efficient as it results in minimal average waiting time.
- Can result in starvation, since long critical processes may not be processed.

**Algorithm**

1. Define an array of structure *process* with members *pid*, *btime*, *wtime* & *ttime*.
2. Get length of the ready queue, i.e., number of process (say *n*)
3. Obtain *btime* for each process.
4. *Sort* the processes according to their *btime* in ascending order.
   a. If two process have same *btime*, then FCFS is used to resolve the tie.
5. The *wtime* for first process is 0.
6. Compute *wtime* and *ttime* for each process as:
   *a.* $wtime_{i+1} = wtime_i + btime_i$
   *b.* $ttime_i = wtime_i + btime_i$
7. Compute average waiting time *awat* and average turn around time *atur.*
8. Display *btime*, *ttime* and *wtime* for each process.
9. Display GANTT chart for the above scheduling
10. Display *awat* and *atur*
11. Stop

**Program**

```
/* SJF Scheduling – sjf.c */#include

<stdio.h>

struct process
{
    int pid; int
    btime;int
    wtime;int
    ttime;
} p[10], temp;

main()
{
    int  i,j,k,n,ttur,twat;float
    awat,atur;

    printf("Enter no. of process : ");scanf("%d",
    &n);
    for(i=0; i<n; i++)
    {
        printf("Burst time for process P%d (in ms) : ",(i+1));scanf("%d", &p[i].btime);
        p[i].pid = i+1;
    }

    for(i=0; i<n-1; i++)
    {
        for(j=i+1; j<n; j++)
        {
            if((p[i].btime > p[j].btime) ||
                (p[i].btime == p[j].btime && p[i].pid > p[j].pid))
            {
                temp = p[i];p[i] =
                p[j];p[j] = temp;
            }

        }
    }
    p[0].wtime = 0; for(i=0;
    i<n; i++)
    {
        p[i+1].wtime = p[i].wtime + p[i].btime;p[i].ttime =
        p[i].wtime + p[i].btime;
    }
    ttur = twat = 0;
```

```
    for(i=0; i<n; i++)
    {
        ttur += p[i].ttime;twat +=
        p[i].wtime;
    }
    awat = (float)twat / n;atur =
    (float)ttur / n;

    printf("\n                SJF  Scheduling\n\n");
    for(i=0; i<28; i++)
        printf("-");
    printf("\nProcess B-Time T-Time W-Time\n");for(i=0; i<28;
    i++)
        printf("-"); for(i=0;
    i<n; i++)
        printf("\n    P%-4d\t%4d\t%3d\t%2d",
                    p[i].pid,p[i].btime,p[i].ttime,p[i].wtime);
    printf("\n"); for(i=0; i<28;
    i++)
        printf("-");
    printf("\n\nAverage waiting time                    : %5.2fms", awat);
    printf("\nAverage turn around time : %5.2fms\n", atur);

    printf("\n\nGANTT  Chart\n");
    printf("-");
    for(i=0; i<(p[n-1].ttime + 2*n); i++)printf("-");
    printf("\n|"); for(i=0; i<n;
    i++)
    {
        k = p[i].btime/2; for(j=0;
        j<k; j++)
            printf(" "); printf("P%d",p[i].pid);
        for(j=k+1; j<p[i].btime; j++)
            printf(" ");
        printf("|");
    }
    printf("\n-");
    for(i=0; i<(p[n-1].ttime + 2*n); i++)printf("-");
    printf("\n0"); for(i=0; i<n;
    i++)
    {
        for(j=0; j<p[i].btime; j++)printf(" ");
        printf("%2d",p[i].ttime);
    }
}
```

**Output**

**Result**

Thus waiting time & turnaround time for processes based on SJF scheduling was computed and the average waiting time was determined.

| Exp. No. 5.C | PRIORITY SCHEDULING |
|---|---|
| Date : | |

**Aim**

To schedule snapshot of processes queued according to Priority scheduling.

**Priority**

- Process that has higher priority is processed first.
- Prioirty can be preemptive or non–preemptive
- When two processes have same priority, FCFS is used to break the tie.
- Can result in starvation, since low priority processes may not be processed.

**Algorithm**

1. Define an array of structure *process* with members *pid*, *btime*, *pri*, *wtime* & *ttime*.
2. Get length of the ready queue, i.e., number of process (say *n*)
3. Obtain *btime* and *pri* for each process.
4. *Sort* the processes according to their *pri* in ascending order.
   a. If two process have same *pri*, then FCFS is used to resolve the tie.
5. The *wtime* for first process is 0.
6. Compute *wtime* and *ttime* for each process as:
   *a.* $wtime_{i+1} = wtime_i + btime_i$
   *b.* $ttime_i = wtime_i + btime_i$
7. Compute average waiting time *awat* and average turn around time *atur*
8. Display the *btime*, *pri*, *ttime* and *wtime* for each process.
9. Display GANTT chart for the above scheduling
10. Display *awat* and *atur*
11. Stop

**Program**

```
/* Priority Scheduling          - pri.c */

#include <stdio.h>

struct process
{
    int pid; int
    btime;int pri;
    int wtime;int
    ttime;
} p[10], temp;

main()
{
    int  i,j,k,n,ttur,twat;float
    awat,atur;

    printf("Enter no. of process : ");scanf("%d",
    &n);
    for(i=0; i<n; i++)
    {
        printf("Burst time for process P%d (in ms) : ", (i+1));scanf("%d", &p[i].btime);
        printf("Priority for process P%d : ", (i+1));scanf("%d", &p[i].pri);
        p[i].pid = i+1;
    }

    for(i=0; i<n-1; i++)
    {
        for(j=i+1; j<n; j++)
        {
            if((p[i].pri > p[j].pri) ||
                (p[i].pri == p[j].pri && p[i].pid > p[j].pid) )
            {
                temp = p[i];p[i] =
                p[j];p[j] = temp;
            }

        }
    }
    p[0].wtime = 0; for(i=0;
    i<n; i++)
    {
        p[i+1].wtime = p[i].wtime + p[i].btime;p[i].ttime =
        p[i].wtime + p[i].btime;
    }
```

```
    ttur = twat = 0; for(i=0;
    i<n; i++)
    {
        ttur += p[i].ttime;twat +=
        p[i].wtime;
    }
    awat = (float)twat / n;atur =
    (float)ttur / n;

    printf("\n\t Priority Scheduling\n\n");for(i=0; i<38; i++)
        printf("-");
    printf("\nProcess B-Time Priority T-Time                W-Time\n");
    for(i=0; i<38; i++)
        printf("-");  for (i=0;
    i<n; i++)
        printf("\n          P%-4d\t%4d\t%3d\t%4d\t%4d",
            p[i].pid,p[i].btime,p[i].pri,p[i].ttime,p[i].wtime);
    printf("\n"); for(i=0; i<38;
    i++)
        printf("-");

    printf("\n\nAverage waiting time                : %5.2fms", awat);
    printf("\nAverage turn around time : %5.2fms\n", atur);

    printf("\n\nGANTT Chart\n");
    printf("-");
    for(i=0; i<(p[n-1].ttime + 2*n); i++)printf("-");
    printf("\n|"); for(i=0; i<n;
    i++)
    {
        k = p[i].btime/2; for(j=0;
        j<k; j++)
            printf(" "); printf("P%d",p[i].pid);
        for(j=k+1; j<p[i].btime; j++)
            printf(" ");
        printf("|");
    }
    printf("\n-");
    for(i=0; i<(p[n-1].ttime + 2*n); i++)printf("-");
    printf("\n0"); for(i=0; i<n;
    i++)
    {
        for(j=0; j<p[i].btime; j++)printf(" ");
        printf("%2d",p[i].ttime);
    }
}
```

**Output**

---

---

---

---

---

**Result**

Thus waiting time & turnaround time for processes based on Priority scheduling was computed and the average waiting time was determined.

| Exp. No. 5.D | ROUND ROBIN SCHEDULING |
|---|---|
| Date : | |

**Aim**

To schedule snapshot of processes queued according to Round robin scheduling.

**Round Robin**

- All processes are processed one by one as they have arrived, but in rounds.
- Each process cannot take more than the time slice per round.
- Round robin is a fair preemptive scheduling algorithm.
- A process that is yet to complete in a round is preempted after the time slice and put at the end of the queue.
- When a process is completely processed, it is removed from the queue.

**Algorithm**

1. Get length of the ready queue, i.e., number of process (say *n*)
2. Obtain *Burst* time $B_i$ for each processes $P_i$.
3. Get the *time slice* per round, say *TS*
4. Determine the number of rounds for each process.
5. The wait time for first process is 0.
6. If $B_i > TS$ then process takes more than one round. Therefore turnaround and waiting time should include the time spent for other remaining processes in the same round.
7. Calculate *average* waiting time and turn around time
8. Display the GANTT chart that includes
   a. order in which the processes were processed in progression of rounds
   b. Turnaround time $T_i$ for each process in progression of rounds.
9. Display the *burst* time, *turnaround* time and *wait* time for each process (in order of rounds they were processed).
10. Display *average* wait time and turnaround time
11. Stop

**Program**

```
/* Round robin scheduling                - rr.c */

#include <stdio.h>

main()
{
     int  i,x=-1,k[10],m=0,n,t,s=0;
     int a[50],temp,b[50],p[10],bur[10],bur1[10];int
     wat[10],tur[10],ttur=0,twat=0,j=0; float awat,atur;

     printf("Enter no. of process : ");scanf("%d",
     &n);
     for(i=0; i<n; i++)
     {
          printf("Burst time for process P%d : ", (i+1));scanf("%d", &bur[i]);
          bur1[i] = bur[i];
     }
     printf("Enter the time slice (in ms) : ");scanf("%d", &t);

     for(i=0; i<n; i++)
     {
          b[i] = bur[i] / t;
          if((bur[i]%t) != 0)
               b[i] += 1;
          m += b[i];
     }

     printf("\n\t\tRound Robin Scheduling\n");

     printf("\nGANTT  Chart\n");for(i=0; i<m;
     i++)
          printf(" ----------");
     printf("\n");

     a[0] = 0;
     while(j < m)
     {
          if(x == n-1)x = 0;
          else
               x++;
          if(bur[x] >= t)
          {
               bur[x] -= t;
               a[j+1] = a[j] + t;
```

```
            if(b[x] == 1)
            {
                p[s] = x;
                k[s] = a[j+1];s++;
            }
            j++;
            b[x] -= 1;
            printf("        P%d      |", x+1);
        }
        else if(bur[x] != 0)
        {
            a[j+1] = a[j] + bur[x];bur[x] =
            0;
            if(b[x] == 1)
            {
                p[s] = x;
                k[s] = a[j+1];s++;
            }
            j++;
            b[x] -= 1;
            printf("        P%d   |",x+1);
        }
    }

    printf("\n"); for(i=0;i<m;i++)
        printf(" ----------");
    printf("\n");

    for(j=0; j<=m; j++)
        printf("%d\t", a[j]);

    for(i=0; i<n; i++)
    {
        for(j=i+1; j<n; j++)
        {
            if(p[i] > p[j])
            {
                temp = p[i];p[i] =
                p[j];p[j] = temp;

                temp = k[i];k[i] =
                k[j];k[j] = temp;
            }
        }
    }
```

```
    for(i=0; i<n; i++)
    {
        wat[i] = k[i] - bur1[i];tur[i] = k[i];
    }
    for(i=0; i<n; i++)
    {
        ttur += tur[i];twat +=
        wat[i];
    }

    printf("\n\n"); for(i=0; i<30;
    i++)
        printf("-"); printf("\nProcess\tBurst\tTrnd\tWait\n");
    for(i=0; i<30; i++)
        printf("-");  for (i=0;
    i<n; i++)
        printf("\nP%-4d\t%4d\t%4d\t%4d", p[i]+1, bur1[i],tur[i],wat[i]);
    printf("\n"); for(i=0; i<30;
    i++)
        printf("-");

    awat = (float)twat / n;atur =
    (float)ttur / n;
    printf("\n\nAverage waiting time                    : %.2f ms", awat);
    printf("\nAverage turn around time : %.2f ms\n", atur);
}
```

**Output**

---

---

---

**Result**

Thus waiting time and turnaround time for processes based on Round robin scheduling was computed and the average waiting time was determined.

| Exp. No. 6 | SEMAPHORE IMPLEMENTATION |
|---|---|
| Date : | |

**Aim**

To demonstrate the utility of semaphore in synchronization and multithreading.

**Semaphore**

- The POSIX system in Linux has its own built-in semaphore library.
- To use it, include semaphore.h.
- Compile the code by linking with -lpthread -lrt.
- To lock a semaphore or wait, use the **sem_wait** function.
- To release or signal a semaphore, use the **sem_post** function.
- A semaphore is initialised by using **sem_init**(for processes or threads)
- To declare a semaphore, the data type is sem_t.

**Algorithm**

1. 2 threads are being created, one 2 seconds after the first one.
2. But the first thread will sleep for 4 seconds after acquiring the lock.
3. Thus the second thread will not enter immediately after it is called, it will enter 4 – 2 = 2 secs after it is called.
4. *Stop*.

**Program**

```
/* C program to demonstrate working of Semaphores */#include <stdio.h>
#include <pthread.h> #include
<semaphore.h>#include
<unistd.h>

sem_t mutex;

void* thread(void* arg)
{
    //wait  sem_wait(&mutex);
    printf("\nEntered..\n");

    //critical sectionsleep(4);
```

```
    //signal
    printf("\nJust  Exiting...\n");
    sem_post(&mutex);
}

int main()
{
    sem_init(&mutex, 0, 1);
    pthread_t t1,t2;
    pthread_create(&t1,NULL,thread,NULL);sleep(2);
    pthread_create(&t2,NULL,thread,NULL);
    pthread_join(t1,NULL); pthread_join(t2,NULL);
    sem_destroy(&mutex);
    return 0;
}
```

**Output**

**Result**

Thus semaphore implementation has been demonstrated.

| Exp. No. 7.A | FIBONACCI & PRIME NUMBER |
|---|---|
| Date : | |

**Aim**

To generate 25 fibonacci numbers and determine prime amongst them using pipe.

**Interprocess Communication**

- Inter-Process communication (IPC), is the mechanism whereby one process can communicate with another process, i.e exchange data.
- IPC in linux can be implemented using pipe, shared memory, message queue, semaphore, signal or sockets.

**Pipe**

- Pipes are unidirectional byte streams which connect the standard output from one process into the standard input of another process.
- A pipe is created using the system call *pipe* that returns a pair of file descriptors.
- The descriptor pfd[0] is used for reading and pfd[1] is used for writing.
- Can be used only between parent and child processes.

**Algorithm**

1. Declare a array to store fibonacci numbers
2. Decalre a array *pfd* with two elements for pipe descriptors.
3. Create pipe on *pfd* using pipe function call.
   a. If return value is -1 then stop
4. Using fork system call, create a child process.
5. Let the child process generate 25 fibonacci numbers and store them in a array.
6. Write the array onto pipe using write system call.
7. Block the parent till child completes using wait system call.
8. Store fibonacci nos. written by child from the pipe in an array using read system call
9. Inspect each element of the fibonacci array and check whether they are prime
   a. If prime then print the fibonacci term.
10. Stop

**Program**

```
/* Fibonacci and Prime using pipe - fibprime.c */#include <stdio.h>
#include <stdlib.h> #include
<unistd.h> #include
<sys/types.h>

main()
{
    pid_t pid;
```

```
int pfd[2];
int i,j,flg,f1,f2,f3;
static unsigned int ar[25],br[25];

if(pipe(pfd) == -1)
{
    printf("Error in pipe");exit(-1);
}

pid=fork(); if (pid
== 0)
{
    printf("Child process generates Fibonacci series\n" );f1 = -1;
    f2 = 1;
    for(i = 0;i < 25; i++)
    {
        f3 = f1 + f2;
        printf("%d\t",f3);f1 = f2;
        f2 = f3; ar[i] = f3;
    }
    write(pfd[1],ar,25*sizeof(int));
}
else if (pid > 0)
{
    wait(NULL);
    read(pfd[0], br, 25*sizeof(int));
    printf("\nParent prints Fibonacci that are Prime\n");


    for(i = 0;i < 25; i++)
    {
        flg = 0;
        if (br[i] <= 1)flg = 1;
        for(j=2; j<=br[i]/2; j++)
        {
            if (br[i]%j == 0)
            {
                flg=1;
                break;
            }
        }
        if (flg == 0) printf("%d\t", br[i]);
    }
    printf("\n");
}
```

```
    else
    {
        printf("Process creation failed");exit(-1);
    }
}
```

**Output**

**Result**

Thus fibonacci numbers that are prime is determined using IPC pipe.

| Exp. No. 7.B | WHO \| WC -L |
|---|---|
| Date : | |

**Aim**

To determine number of users logged in using pipe.

**Algorithm**

1. Decalre a array *pfd* with two elements for pipe descriptors.
2. Create pipe on *pfd* using pipe function call.
   a. If return value is -1 then stop
3. Using fork system call, create a child process.
4. Free the standard output (1) using close system call to redirect the output to pipe.
5. Make a copy of write end of the pipe using dup system call.
6. Execute who command using execlp system call.
7. Free the standard input (0) using close system call in the other process.
8. Make a close of read end of the pipe using dup system call.
9. Execute wc –l command using execlp system call.
10. Stop

**Program**

```
/* No. of users logged - cmdpipe.c */#include

<stdio.h>
#include  <stdlib.h>
#include  <unistd.h>

int main()
{
     int pfds[2];
     pipe(pfds);

     if (!fork())
     {
          close(1); dup(pfds[1]);
          close(pfds[0]); execlp("who",  "who",
          NULL);
     }
```

```
    else
    {
        close(0); dup(pfds[0]);
        close(pfds[1]);
        execlp("wc", "wc", "-l", NULL);
    }
}
```

**Output**

**Result**

Thus standard output of who is connected to standard input of wc using pipe to compute number of users logged in.

| Exp. No. 7.C | CHAT MESSAGING |
|---|---|
| Date : | |

**Aim**

To exchange message between server and client using message queue.

**Message Queue**

- A message queue is a linked list of messages stored within the kernel
- A message queue is identified by a unique identifier
- Every message has a positive long integer type field, a non-negative length, and the actual data bytes.
- The messages need not be fetched on FCFS basis. It could be based on type field.

**Algorithm**

Server

1. Decalre a structure *mesgq* with *type* and *text* fields.
2. Initialize *key* to 2013 (some random value).
3. Create a message queue using msgget with *key* & IPC_CREAT as parameter.
   a. If message queue cannot be created then stop.
4. Initialize the message *type* member of *mesgq* to 1.
5. Do the following until user types Ctrl+D
   a. Get message from the user and store it in *text* member.
   b. Delete the newline character in *text* member.
   c. Place message on the queue using msgsend for the client to read.
   d. Retrieve the response message from the client using msgrcv function
   e. Display the *text* contents.
6. Remove message queue from the system using msgctl with IPC_RMID as parameter.
7. Stop

Client

1. Decalre a structure *mesgq* with *type* and *text* fields.
2. Initialize *key* to 2013 (same value as in server).
3. Open the message queue using msgget with *key* as parameter.
   a. If message queue cannot be opened then stop.
4. Do while the message queue exists
   a. Retrieve the response message from the server using msgrcv function
   b. Display the *text* contents.
   c. Get message from the user and store it in *text* member.
   d. Delete the newline character in *text* member.
   e. Place message on the queue using msgsend for the server to read.
5. Print "Server Disconnected".
6. Stop.

**Program**

Server

```
/* Server chat process - srvmsg.c */

#include <stdio.h> #include
<stdlib.h> #include <string.h>
#include  <sys/types.h>#include
<sys/ipc.h> #include
<sys/msg.h>

struct mesgq
{
     long type;
     char text[200];
} mq;

main()
{
     int msqid, len; key_t key =
     2013;

     if((msqid = msgget(key, 0644|IPC_CREAT)) == -1)
     {
          perror("msgget");exit(1);
     }

     printf("Enter text, ^D to quit:\n");mq.type = 1;

     while(fgets(mq.text, sizeof(mq.text), stdin) != NULL)
     {
          len = strlen(mq.text);
          if (mq.text[len-1] == '\n')
               mq.text[len-1] = '\0'; msgsnd(msqid,
          &mq, len+1, 0);

          msgrcv(msqid, &mq, sizeof(mq.text), 0, 0);printf("From Client:
          \"%s\"\n", mq.text);
     }
     msgctl(msqid, IPC_RMID, NULL);
}
```

Client

```
/* Client chat process - climsg.c */

#include <stdio.h> #include
<stdlib.h> #include <string.h>
#include  <sys/types.h>#include
<sys/ipc.h> #include
<sys/msg.h>

struct mesgq
{
     long type;
     char text[200];
} mq;

main()
{
     int msqid, len; key_t key =
     2013;

     if ((msqid = msgget(key, 0644)) == -1)
     {
          printf("Server not active\n");exit(1);
     }

     printf("Client ready :\n");
     while (msgrcv(msqid, &mq, sizeof(mq.text), 0, 0) != -1)
     {
          printf("From Server: \"%s\"\n", mq.text);

          fgets(mq.text, sizeof(mq.text), stdin);len =
          strlen(mq.text);
          if (mq.text[len-1] == '\n')
               mq.text[len-1] = '\0'; msgsnd(msqid,
          &mq, len+1, 0);
     }
     printf("Server  Disconnected\n");
}
```

**Output**

Server

Client

**Result**

Thus chat session between client and server was done using message queue.

| Exp. No. 7.D | SHARED MEMORY |
|---|---|
| Date : | |

**Aim**

To demonstrate communication between process using shared memory.

**Shared memory**

- Two or more processes share a single chunk of memory to communicate randomly.
- Semaphores are generally used to avoid race condition amongst processes.
- Fastest amongst all IPCs as it does not require any system call.
- It avoids copying data unnecessarily.

**Algorithm**

Server

1. Initialize size of shared memory *shmsize* to 27.
2. Initialize *key* to 2013 (some random value).
3. Create a shared memory segment using shmget with *key* & IPC_CREAT as parameter.
   a. If shared memory identifier *shmid* is -1, then stop.
4. Display *shmid*.
5. Attach server process to the shared memory using shmmat with *shmid* as parameter.
   a. If pointer to the shared memory is not obtained, then stop.
6. Clear contents of the shared region using memset function.
7. Write a–z onto the shared memory.
8. Wait till client reads the shared memory contents
9. Detatch process from the shared memory using shmdt system call.
10. Remove shared memory from the system using shmctl with IPC_RMID argument
11. Stop

Client

1. Initialize size of shared memory *shmsize* to 27.
2. Initialize *key* to 2013 (same value as in server).
3. Obtain access to the same shared memory segment using same *key*.
   a. If obtained then display the *shmid* else print "Server not started"
4. Attach client process to the shared memory using shmmat with *shmid* as parameter.
   a. If pointer to the shared memory is not obtained, then stop.
5. Read contents of shared memory and print it.
6. After reading, modify the first character of shared memory to '*'
7. Stop

**Program**

Server

```
/* Shared memory server - shms.c */

#include <stdio.h> #include
<stdlib.h> #include <sys/un.h>
#include  <sys/types.h>#include
<sys/ipc.h> #include
<sys/shm.h>

#define shmsize 27main()
{
     char c; int
     shmid;
     key_t key =          2013;
     char *shm, *s;

     if ((shmid = shmget(key, shmsize, IPC_CREAT|0666)) < 0)
     {
          perror("shmget");exit(1);
     }
     printf("Shared memory id : %d\n", shmid);

     if ((shm = shmat(shmid, NULL, 0)) == (char *) -1)
     {
          perror("shmat");
          exit(1);
     }

     memset(shm, 0, shmsize);s =
     shm;
     printf("Writing (a-z) onto shared memory\n");for (c = 'a'; c <=
     'z'; c++)
          *s++ = c;
     *s = '\0';

     while (*shm != '*');
     printf("Client finished reading\n");

     if(shmdt(shm) != 0)
          fprintf(stderr, "Could not close memory segment.\n");

     shmctl(shmid, IPC_RMID, 0);
}
```

Client

```
/* Shared memory client - shmc.c */

#include <stdio.h> #include
<stdlib.h> #include
<sys/types.h>#include
<sys/ipc.h> #include
<sys/shm.h>

#define shmsize 27main()
{
    int shmid;
    key_t key = 2013;char
    *shm, *s;

    if ((shmid = shmget(key, shmsize, 0666)) < 0)
    {
        printf("Server not started\n");exit(1);
    }
    else
        printf("Accessing shared memory id : %d\n",shmid);

    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1)
    {
        perror("shmat");
        exit(1);
    }

    printf("Shared memory contents:\n");for (s =
    shm; *s != '\0'; s++)
        putchar(*s);putchar('\n');

    *shm = '*';
}
```

**Output**

Server

Client

**Result**

Thus contents written onto shared memory by the server process is read by the client process.

| Exp. No. 7.E | PRODUCER-CONSUMER PROBLEM |
|---|---|
| Date : | |

**Aim**

To synchronize producer and consumer processes using semaphore.

**Semaphores**

- A semaphore is a counter used to synchronize access to a shared data amongst multiple processes.
- To obtain a shared resource, the process should:
  - Test the semaphore that controls the resource.
  - If value is positive, it gains access and decrements value of semaphore.
  - If value is zero, the process goes to sleep and awakes when value is > 0.
- When a process relinquishes resource, it increments the value of semaphore by 1.

**Producer-Consumer problem**

- A producer process produces information to be consumed by a consumer process
- A producer can produce one item while the consumer is consuming another one.
- With bounded-buffer size, consumer must wait if buffer is empty, whereas producer must wait if buffer is full.
- The buffer can be implemented using any IPC facility.

**Algorithm**

1. Create a shared memory segment *BUFSIZE* of size 1 and attach it.
2. Obtain semaphore id for variables *empty*, *mutex* and *full* using semget function.
3. Create semaphore for *empty*, *mutex* and *full* as follows:
   a. Declare *semun*, a union of specific commands.
   b. The initial values are: 1 for mutex, N for empty and 0 for full
   c. Use semctl function with SETVAL command
4. Create a child process using fork system call.
   *a.* Make the parent process to be the *producer*
   *b.* Make the child process to the *consumer*
5. The *producer* produces 5 items as follows:
   a. Call *wait* operation on semaphores *empty* and *mutex* using semop function.
   b. Gain access to buffer and produce data for consumption
   c. Call *signal* operation on semaphores *mutex* and *full* using semop function.
6. The *consumer* consumes 5 items as follows:
   a. Call *wait* operation on semaphores *full* and *mutex* using semop function.
   b. Gain access to buffer and consume the available data.
   c. Call *signal* operation on semaphores *mutex* and *empty* using semop function.
7. Remove shared memory from the system using shmctl with IPC_RMID argument
8. Stop

**Program**

```
/* Producer-Consumer problem using semaphore – pcsem.c */#include <stdio.h>
#include <stdlib.h>
#include  <sys/types.h>#include
<sys/ipc.h> #include
<sys/shm.h> #include
<sys/sem.h>

#define N 5
#define BUFSIZE 1
#define PERMS 0666

int *buffer;
int nextp = 0, nextc = 0;
int mutex, full, empty;                    /* semaphore variables */

void producer()
{
     int data; if(nextp ==
     N)
          nextp = 0;
     printf("Enter data for producer to produce : ");scanf("%d",(buffer + nextp));
     nextp++;
}

void consumer()
{
     int   g; if(nextc ==
     N)
          nextc = 0;
     g = *(buffer + nextc++); printf("\nConsumer consumes
     data %d", g);
}

void sem_op(int id, int value)
{
     struct sembuf op;int v;
     op.sem_num = 0; op.sem_op =
     value; op.sem_flg =
     SEM_UNDO;
     if((v = semop(id, &op, 1)) < 0)
          printf("\nError executing semop instruction");
}
```

```
void sem_create(int semid, int initval)
{
    int semval;union
    semun
    {
        int val;
        struct semid_ds *buf; unsigned
        short *array;
    } s;

    s.val = initval;
    if((semval = semctl(semid, 0, SETVAL, s)) < 0)printf("\nError in executing
        semctl");
}

void sem_wait(int id)
{
    int value = -1; sem_op(id,
    value);
}

void sem_signal(int id)
{
    int value = 1; sem_op(id,
    value);
}

main()
{
    int shmid, i;pid_t
    pid;

    if((shmid = shmget(1000, BUFSIZE, IPC_CREAT|PERMS)) < 0)
    {
        printf("\nUnable to create shared memory");return;
    }
    if((buffer = (int*)shmat(shmid, (char*)0, 0)) == (int*)-1)
    {
        printf("\nShared memory allocation error\n");exit(1);
    }

    if((mutex = semget(IPC_PRIVATE, 1, PERMS|IPC_CREAT)) == -1)
    {
        printf("\nCan't create mutex semaphore");exit(1);
    }
```

```
    if((empty = semget(IPC_PRIVATE, 1, PERMS|IPC_CREAT)) == -1)
    {
        printf("\nCan't create empty semaphore");exit(1);
    }
    if((full = semget(IPC_PRIVATE, 1, PERMS|IPC_CREAT)) == -1)
    {
        printf("\nCan't create full semaphore");exit(1);
    }

    sem_create(mutex,       1);
    sem_create(empty,       N);
    sem_create(full, 0);

    if((pid = fork()) < 0)
    {
        printf("\nError in process creation");exit(1);
    }
    else if(pid > 0)
    {
        for(i=0; i<N; i++)
        {
            sem_wait(empty);
            sem_wait(mutex); producer();
            sem_signal(mutex);
            sem_signal(full);
        }
    }
    else if(pid == 0)
    {
        for(i=0; i<N; i++)
        {
            sem_wait(full); sem_wait(mutex);
            consumer(); sem_signal(mutex);
            sem_signal(empty);
        }
        printf("\n");
    }
}
```

**Output**

**Result**

Thus synchronization between producer and consumer process for access to a shared memory segment is implemented.

| Exp. No. 8 | BANKERS ALGORITHM FOR DEAD LOCK AVOIDANCE |
|---|---|
| Date : | |

**AIM**

To implement deadlock avoidance by using Banker's Algorithm.

**ALGORITHM**

1. Start the program.
2. Get the values of resources and processes.
3. Get the avail value.
4. After allocation find the need value.
5. Check whether its possible to allocate.
6. If it is possible then the system is in safe state.
7. Else system is not in safety state.
8. If the new request comes then check that the system is in safety or not if we allow the request.
9. Stop.

**PROGRAM**

```
#include <stdio.h>#include
<stdio.h>

main()
{
    int r[1][10], av[1][10];
    int all[10][10], max[10][10], ne[10][10], w[10],safe[10];int i=0, j=0, k=0, l=0,
    np=0, nr=0, count=0, cnt=0;

    clrscr();
    printf("enter the number of processes in a system");scanf("%d", &np);
    printf("enter the number of resources in a system");scanf("%d",&nr);
    for(i=1; i<=nr; i++)
    {
        printf("Enter no. of instances of resource R%d " ,i);scanf("%d", &r[0][i]);
        av[0][i] = r[0][i];
    }

    for(i=1; i<=np; i++) for(j=1;
        j<=nr; j++)
            all[i][j] = ne[i][j] = max[i][j] = w[i]=0;
```

```
printf("Enter the allocation matrix");for(i=1; i<=np;
i++)
{
     for(j=1; j<=nr; j++)
     {
          scanf("%d", &all[i][j]);
          av[0][j] = av[0][j] - all[i][j];
     }
}

printf("Enter the maximum matrix");for(i=1; i<=np;
i++)
{
     for(j=1; j<=nr; j++)
     {
          scanf("%d",&max[i][j]);
     }
}

for(i=1; i<=np; i++)
{
     for(j=1; j<=nr; j++)
     {
          ne[i][j] = max[i][j] - all[i][j];
     }
}

for(i=1; i<=np; i++)
{
     printf("pocess P%d", i);for(j=1;
     j<=nr; j++)
     {
          printf("\n allocated %d\t",all[i][j]);printf("maximum %d\t",max[i][j]);
          printf("need %d\t",ne[i][j]);
     }
     printf("\n_____\n");
}

printf("\nAvailability ");for(i=1; i<=nr; i++)
     printf("R%d %d\t", i, av[0][i]);printf("\n
_____"); printf("\n safe
sequence");
```

```
    for(count=1; count<=np; count++)
    {
       for(i=1; i<=np; i++)
       {
            Cnt = 0;
            for(j=1; j<=nr; j++)
            {
                 if(ne[i][j] <= av[0][j] && w[i]==0)cnt++;
            }
            if(cnt == nr)
            {
                 k++;
                 safe[k] = i; for(l=1; l<=nr;
                 l++)
                      av[0][l] = av[0][l] + all[i][l];printf("\n P%d
                 ",safe[k]); printf("\t   Availability "); for(l=1;
                 l<=nr; l++)
                      printf("R%d %d\t", l, av[0][l]);w[i]=1;
            }
        }
    }
    getch();
}
```

**Output**

**Result**

Thus bankers algorithm for dead lock avoidance was executed successfully.

| Exp. No. 9 | DEAD LOCK PREVENTION |
|---|---|
| Date : | |

**Aim**

To determine whether the process and their request for resources are in a deadlocked state.

**Algorithm**

1. Mark each process that has a row in the Allocation matrix of all zeros.
2. Initialize a temporary vectorW to equal the Available vector.
3. Find an indexi such that processi is currently unmarked and thei th row ofQ
4. is less than or equal to W . That is,Q ik … Wk, for 1 … k … m . If no such row is
5. found, terminate the algorithm.
5. If such a row is found, mark processi and add the corresponding row of the
6. allocation matrix to W . That is, setWk = Wk + Aik, for 1 … k … m . Return
7. to step 3.

**Program**

```
#include<stdio.h>
#include<conio.h> int
max[100][100]; int
alloc[100][100];int
need[100][100]; int
avail[100];
int n, r; void
input();void
show(); void cal();

main()
{
    int i,j;
    printf("Deadlock Detection Algo\n");input();
    show();
    cal();
    getch();
}

void input()
{
int i,j;
    printf("Enter the no of Processes\t");
    scanf("%d",&n);
    printf("Enter the no of resource instances\t");scanf("%d", &r);

    printf("Enter the Max Matrix\n");
```

```
    for(i=0; i<n; i++) for(j=0; j<r;
        j++)
            scanf("%d", &max[i][j]);

    printf("Enter the Allocation Matrix\n");for(i=0; i<n; i++)
        for(j=0; j<r;  j++) scanf("%d",
            &alloc[i][j]);
    printf("Enter the available Resources\n");for(j=0;j<r;j++)
        scanf("%d",&avail[j]);
}

void show()
{
    int i, j;
    printf("Process\t Allocation\t Max\t Available\t");for(i=0; i<n; i++)
    {
        printf("\nP%d\t          ", i+1);
        for(j=0; j<r; j++)
        {
            printf("%d ", alloc[i][j]);
        }
        printf("\t"); for(j=0; j<r;
        j++)
        {
            printf("%d ", max[i][j]);
        }
        printf("\t");if(I == 0)
        {
            for(j=0; j<r; j++) printf("%d ",
                avail[j]);
        }
    }
}

void cal()
{
    int finish[100], temp, need[100][100], flag=1, k, c1=0;int dead[100];
    int safe[100];int i, j;
    for(i=0; i<n; i++)
    {
        finish[i] = 0;
    }
```

```
/*find need matrix */for(i=0; i<n; i++)
{
    for(j=0; j<r; j++)
    {
        need[i][j]= max[i][j] - alloc[i][j];
    }
}

while(flag)
{
    flag=0;
    for(i=0;i<n;i++)
    {
        int c=0; for(j=0;j<r;j++)
        {
            if((finish[i]==0) && (need[i][j] <= avail[j]))
            {
                c++;
                if(c == r)
                {
                        for(k=0; k<r; k++)
                        {
                                avail[k] += alloc[i][j];finish[i]=1;
                                flag=1;
                        }
                        if(finish[i] == 1)
                        {
                                i=n;
                        }
                }
            }
        }
    }
}
J = 0;
Flag = 0;
for(i=0; i<n; i++)
{
    if(finish[i] == 0)
    {
        dead[j] = i;j++;
        flag = 1;
    }
}
```

```
    if(flag == 1)
    {
        printf("\n\nSystem is in Deadlock and the Deadlock
process are\n");
        for(i=0;i<n;i++)
        {
            printf("P%d\t", dead[i]);
        }
  }
  else
  {
        printf("\nNo Deadlock Occur");
  }
}
```

**Output**

**********  Deadlock Detection Algo  **********

**Result**

Thus using given state of information deadlocked process were determined.

| Exp. No. 10 | THREADING AND SYNCHRONIZATION |
|---|---|
| Date : | |

**Aim**

To demonstrate threading and synchronization using mutex.

**Description**

- Thread synchronization is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as critical section.
- Processes' access to critical section is controlled by using synchronization techniques.
- When one thread starts executing the critical section (serialized segment of the program) the other thread should wait until the first thread finishes.
- If proper synchronization techniques are not applied, it may cause a race condition where the values of variables may be unpredictable
- A Mutex is a lock that we set before using a shared resource and release after using it.
- When the lock is set, no other thread can access the locked region of code. So this ensures a synchronized access of shared resources in the code.

**Algorithm**

1. Create two threads
2. Let the threads share a common resource, say counter
3. Even if thread2 si scheduled to start while thread was not done, access to shared resource is not done as it is locked by mutex
4. Once thread1 completes, thread2 starts execution
5. Stop

**Program**

```
#include <stdio.h> #include
<string.h> #include
<pthread.h>#include
<stdlib.h> #include
<unistd.h>

pthread_t tid[2];int
counter;
pthread_mutex_t lock;

void* trythis(void *arg)
{
    pthread_mutex_lock(&lock);

    unsigned long i = 0;counter +=
    1;
    printf("\n Job %d has started\n", counter);for(i=0;

    i<(0xFFFFFFFF);i++);
```

```
    printf("\n Job %d has finished\n", counter);pthread_mutex_unlock(&lock);

    return NULL;
}

main()
{
    int i = 0;int
    error;

    if (pthread_mutex_init(&lock, NULL) != 0)
    {
        printf("\n mutex init has failed\n");return 1;
    }

    while(i < 2)
    {
        err = pthread_create(&(tid[i]), NULL, &trythis, NULL);if (error != 0)
            printf("\nThread can't be created :[%s]",
strerror(error));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);

    return 0;
}
```

**Output**

**Result**

**Thus concurrent threads were synchronized using mutex lock.**

| Exp. No.11.A | FIRST FIT ALLOCATION |
|---|---|
| Date : | |

**Aim**

To allocate memory requirements for processes using first fit allocation.

**Memory Management**

- The first-fit, best-fit, or worst-fit strategy is used to select a free hole from the set of available holes.

**First fit**

- Allocate the first hole that is big enough.
- Searching starts from the beginning of set of holes.

**Algorithm**

1. Declare structures *hole* and *process* to hold information about set of holes and processes respectively.
2. Get number of holes, say *nh*.
3. Get the size of each hole
4. Get number of processes, say *np*.
5. Get the memory requirements for each process.
6. Allocate processes to holes, by examining each hole as follows:
    a. If hole size > process size then
        i. Mark process as allocated to that hole.
        ii. Decrement hole size by process size.
    b. Otherwise check the next from the set of hole
7. Print the list of process and their allocated holes or unallocated status.
8. Print the list of holes, their actual and current availability.
9. Stop

**Program**

```
/* First fit allocation - ffit.c */#include <stdio.h>

struct process
{
    int size; int flag;
    int holeid;
} p[10];
struct hole
{
    int size;
```

```
    int actual;
} h[10];

main()
{
    int i, np, nh, j;

    printf("Enter the number of Holes : ");scanf("%d",
    &nh);
    for(i=0; i<nh; i++)
    {
        printf("Enter size for hole H%d : ",i);scanf("%d",
        &h[i].size);
        h[i].actual =           h[i].size;
    }

    printf("\nEnter number of process : " );scanf("%d",&np);
    for(i=0;i<np;i++)
    {
        printf("enter the size of process P%d : ",i);scanf("%d", &p[i].size);
        p[i].flag = 0;
    }

    for(i=0; i<np; i++)
    {
        for(j=0; j<nh; j++)
        {
            if(p[i].flag != 1)
            {
                if(p[i].size <= h[j].size)
                {
                    p[i].flag = 1; p[i].holeid = j;
                    h[j].size -= p[i].size;
                }
            }
        }
    }

    printf("\n\tFirst fit\n");
    printf("\nProcess\tPSize\tHole");for(i=0; i<np; i++)
    {
        if(p[i].flag != 1)
            printf("\nP%d\t%d\tNot allocated", i, p[i].size);else
            printf("\nP%d\t%d\tH%d", i, p[i].size, p[i].holeid);
    }
```

```
    printf("\n\nHole\tActual\tAvailable");for(i=0; i<nh
    ;i++)
        printf("\nH%d\t%d\t%d", i, h[i].actual, h[i].size);printf("\n");
}
```

**Output**

**Result**

Thus processes were allocated memory using first fit method.

| Exp. No. 11.B | BEST FIT ALLOCATION |
|---|---|
| Date : | |

**Aim**

To allocate memory requirements for processes using best fit allocation.

**Best fit**

- Allocate the smallest hole that is big enough.
- The list of free holes is kept sorted according to size in ascending order.
- This strategy produces smallest leftover holes

**Algorithm**

1. Declare structures *hole* and *process* to hold information about set of holes and processes respectively.
2. Get number of holes, say *nh*.
3. Get the size of each hole
4. Get number of processes, say *np*.
5. Get the memory requirements for each process.
6. Allocate processes to holes, by examining each hole as follows:
    a. Sort the holes according to their sizes in ascending order
    b. If hole size > process size then
        i. Mark process as allocated to that hole.
        ii. Decrement hole size by process size.
    c. Otherwise check the next from the set of sorted hole
7. Print the list of process and their allocated holes or unallocated status.
8. Print the list of holes, their actual and current availability.
9. Stop

**Program**

```
#include <stdio.h>struct

process
{
    int size; int flag;
    int holeid;
} p[10];

struct hole
{
    int  hid; int
    size; int actual;
} h[10];
```

```
main()
{
    int i, np, nh, j;
    void bsort(struct hole[], int); printf("Enter the number
    of Holes : ");scanf("%d", &nh);
    for(i=0; i<nh; i++)
    {
        printf("Enter size for hole H%d : ",i);scanf("%d",
        &h[i].size);
        h[i].actual =           h[i].size;
        h[i].hid = i;
    }
    printf("\nEnter number of process : " );scanf("%d",&np);
    for(i=0;i<np;i++)
    {
        printf("enter the size of process P%d : ",i);scanf("%d", &p[i].size);
        p[i].flag = 0;
    }
    for(i=0; i<np; i++)
    {
        bsort(h, nh); for(j=0; j<nh;
        j++)
        {
            if(p[i].flag != 1)
            {
                if(p[i].size <= h[j].size)
                {
                    p[i].flag = 1; p[i].holeid =
                    h[j].hid;
                    h[j].size -= p[i].size;
                }
            }
        }
    }
    printf("\n\tBest fit\n");
    printf("\nProcess\tPSize\tHole");for(i=0; i<np; i++)
    {
        if(p[i].flag != 1)
            printf("\nP%d\t%d\tNot allocated", i, p[i].size);else
            printf("\nP%d\t%d\tH%d", i, p[i].size, p[i].holeid);
    }
    printf("\n\nHole\tActual\tAvailable");for(i=0; i<nh
    ;i++)
        printf("\nH%d\t%d\t%d", h[i].hid, h[i].actual,h[i].size);
```

```
    printf("\n");
}

void bsort(struct hole bh[], int n)
{
    struct hole temp;int i,j;
    for(i=0; i<n-1; i++)
    {
        for(j=i+1; j<n; j++)
        {
            if(bh[i].size > bh[j].size)
            {
                temp = bh[i]; bh[i]
                = bh[j];bh[j] =
                temp;
            }
        }
    }
}
```

**Output**

**Result**

Thus processes were allocated memory using best fit method.

| Exp. No. 12 | PAGING TECHNIQUE |
|---|---|
| Date : | |

**Aim**

To determine physical address of a given page using page table.

**Algorithm**

1. Get process size
2. Compte no. of pages available and display it
3. Get relative address
4. Determine the corresponding page
5. Display page table
6. Display the physical address

**Program**

```
#include <stdio.h>#include
<math.h>

main()
{
    int size, m, n, pgno, pagetable[3]={5,6,7}, i, j, frameno;double m1;
    int ra=0, ofs;

    printf("Enter process size (in KB of max 12KB):");scanf("%d", &size);
    m1 = size / 4;n =
    ceil(m1);
    printf("Total No. of pages: %d", n); printf("\nEnter relative
    address (in hexa) \n");scanf("%d", &ra);

    pgno = ra / 1000;ofs = ra
    % 1000;
    printf("page no=%d\n", pgno);
    printf("page table"); for(i=0;i<n;i++)
        printf("\n %d [%d]", i, pagetable[i]);frameno =
    pagetable[pgno];
    printf("\nPhysical address: %d%d", frameno, ofs);

}
```

**Output**

**Result**

Thus physical address for the given logical address is determing using Paging technique.

| Exp. No. 13.A | FIFO PAGE REPLACEMENT |
|---|---|
| Date : | |

**Aim**

To implement demand paging for a reference string using FIFO method.

**FIFO**

- Page replacement is based on when the page was brought into memory.
- When a page should be replaced, the oldest one is chosen.
- Generally, implemented using a FIFO queue.
- Simple to implement, but not efficient.
- Results in more page faults.
- The page-fault may increase, even if frame size is increased (Belady's anomaly)

**Algorithm**

1. Get length of the reference string, say *l*.
2. Get reference string and store it in an array, say *rs*.
3. Get number of frames, say *nf*.
4. Initalize *frame* array upto length *nf* to -1.
5. Initialize position of the oldest page, say *j* to 0.
6. Initialize no. of page faults, say *count* to 0.
7. For each page in reference string in the given order, examine:
    a. Check whether page exist in the *frame* array
    b. If it does not exist then
        i. Replace page in position *j*.
        ii. Compute page replacement position as (*j*+1) modulus *nf*.
        iii. Increment *count* by 1.
        iv. Display pages in *frame* array.
8. Print *count*.
9. Stop

**Program**

```
#include <stdio.h>main()
{
    int  i,j,l,rs[50],frame[10],nf,k,avail,count=0;

    printf("Enter length of ref. string : ");scanf("%d", &l);
    printf("Enter reference string :\n");for(i=1; i<=l;
    i++)
        scanf("%d",   &rs[i]); printf("Enter number
    of frames : ");scanf("%d", &nf);
```

```
    for(i=0; i<nf; i++)frame[i]
        = -1;
    j = 0;
    printf("\nRef. str              Page frames");
    for(i=1; i<=l; i++)
    {
        printf("\n%4d\t",  rs[i]);avail = 0;
        for(k=0; k<nf; k++) if(frame[k]
            == rs[i])
                avail = 1;
        if(avail == 0)
        {
            frame[j] = rs[i];j = (j+1)
            % nf; count++;
            for(k=0; k<nf; k++) printf("%4d",
                frame[k]);
        }
    }
    printf("\n\nTotal no. of page faults : %d\n",count);
}
```

**Output**

**Result**

Thus page replacement was implemented using FIFO algorithm.

| Exp. No. 13.B | LRU PAGE REPLACEMENT |
|---|---|
| Date : | |

**Aim**

To implement demand paging for a reference string using LRU method.

**LRU**

- **Pages used in the recent past are used as an approximation of future usage.**
- **The page that has not been used for a longer period of time is replaced.**
- **LRU is efficient but not optimal.**
- **Implementation of LRU requires hardware support, such as counters/stack.**

**Algorithm**

1. Get length of the reference string, say *len*.
2. Get reference string and store it in an array, say *rs*.
3. Get number of frames, say *nf*.
4. Create *access* array to store counter that indicates a measure of recent usage.
5. Create a function *arrmin* that returns position of minimum of the given array.
6. Initalize *frame* array upto length *nf* to -1.
7. Initialize position of the page replacement, say *j* to 0.
8. Initialize *freq* to 0 to track page frequency
9. Initialize no. of page faults, say *count* to 0.
10. For each page in reference string in the given order, examine:
    a. Check whether page exist in the *frame* array.
    b. If page exist in memory then
        i. Store incremented *freq* for that page position in *access* array.

c. If page does not exist in memory then
    i. Check for any empty frames.
    ii. If there is an empty frame,
        - Assign that frame to the page
        - Store incremented *freq* for that page position in *access* array.
        - Increment *count*.
    iii. If there is no free frame then
        - Determine page to be replaced using *arrmin* function.
        - Store incremented *freq* for that page position in *access* array.
        - Increment *count*.
    iv. Display pages in *frame* array.
11. Print *count*.
12. Stop

**Program**

**/* LRU page replacement - lrupr.c */#include**

**<stdio.h>**

```
int arrmin(int[], int);

main()
{
    int i,j,len,rs[50],frame[10],nf,k,avail,count=0;int access[10], freq=0,
    dm;

    printf("Length of Reference string : ");scanf("%d", &len);
    printf("Enter reference string :\n");for(i=1; i<=len;
    i++)
        scanf("%d", &rs[i]); printf("Enter no. of
    frames : ");scanf("%d", &nf);

    for(i=0; i<nf; i++)frame[i]
        = -1;
    j = 0;

    printf("\nRef. str              Page frames");
    for(i=1; i<=len; i++)
    {
        printf("\n%4d\t",  rs[i]);avail = 0;
        for(k=0; k<nf; k++)
        {
            if(frame[k] == rs[i])
            {
                avail = 1; access[k] =
                ++freq;break;
            }
        }
        if(avail == 0)
        {
            dm = 0;
            for(k=0; k<nf; k++)
            {
                if(frame[k] == -1)
                    dm = 1;
                    break;
            }
            if(dm == 1)
            {
                frame[k] = rs[i]; access[k] =
                ++freq;count++;
            }
```

```
                else
                {
                    j = arrmin(access, nf);frame[j] =
                    rs[i]; access[j] = ++freq;
                    count++;
                }
                for(k=0; k<nf; k++) printf("%4d",
                    frame[k]);
            }
        }
        printf("\n\nTotal no. of page faults : %d\n", count);
}

int arrmin(int a[], int n)
{
       int i, min = a[0];for(i=1;
          i<n; i++)if (min > a[i])
                min = a[i];
        for(i=0; i<n; i++)
            if (min == a[i])return
                i;
}
```

**Output**

**Result**

Thus page replacement was implemented using LRU algorithm.

| Exp. No. 14.A | SINGLE-LEVEL DIRECTORY |
|---|---|
| Date : | |

**Aim**

To organize files in a single level directory structure, I,e., without sub-directories.

**Algorithm**

1. Get name of directory for the user to store all the files
2. Display menu
3. Accept choice
4. If choice =1 then
   - Accept filename without any collision
   - Store it in the directory
5. If choice =2 then
   - Accept filename
   - Remove filename from the directory array
6. If choice =3 then
   - Accept filename
   - Check for existence of file in the directory array
7. If choice =4 then
   - List all files in the directory array
8. If choice =5 then
   - Stop

**Program**

```
#include <stdio.h>
@include <stdlib.h>
#include <conio.h>

struct
{
    char   dname[10]; char
    fname[25][10];int fcnt;
}dir;

main()
{
    int i, ch; char
    f[30]; clrscr();
    dir.fcnt = 0;
    printf("\nEnter name of directory -- ");scanf("%s",
    dir.dname);

    while(1)
    {
        printf("\n\n 1. Create File\t2. Delete File\t3. Search File \n4. Display Files\t5.
        Exit\nEnter your choice--");scanf("%d",&ch);
```

```
switch(ch)
{
    case 1:
        printf("\n Enter the name of the file -- ");scanf("%s",
        dir.fname[dir.fcnt]); dir.fcnt++;
        break;

    case 2:
        printf("\n Enter the name of the file -- ");scanf("%s", f);
        for(i=0; i<dir.fcnt; i++)
        {
            if(strcmp(f, dir.fname[i]) == 0)
            {
                printf("File %s is deleted ",f); strcpy(dir.fname[i],
                dir.fname[dir.fcnt-1]);break;
            }
        }
        if(I == dir.fcnt)
            printf("File %s not found", f);else
            dir.fcnt--;break;

    case 3:
        printf("\n Enter the name of the file -- ");scanf("%s", f);
        for(i=0; i<dir.fcnt; i++)
        {
            if(strcmp(f, dir.fname[i]) == 0)
            {
                    printf("File %s is found ", f);break;
            }
        }
        if(I == dir.fcnt)
            printf("File %s not found", f);break;

    case 4:
        if(dir.fcnt == 0)
            printf("\n Directory Empty");else
        {
            printf("\n The Files are -- ");for(i=0;
            i<dir.fcnt; i++)
                    printf("\t%s", dir.fname[i]);
        }
        break;
```

```
                default:
                    exit(0);
            }
        }
        getch();
}
```

**Output**

**Result**

Thus files were organized into a single level directory.

| Exp. No. 14.B | TWO-LEVEL DIRECTORY |
|---|---|
| Date : | |

**Aim**

To organize files as two-level directory with each user having his own user file directory (UFD).

**Algorithm**

1. Display menu
2. Accept choice
3. If choice =1 then
   - Accept directory name
   - Create an entry for that directory
4. If choice =2 then
   - Get directory name
   - If directory exist then accept filename without collision else report error
5. If choice =3 then
   - Get directory name
   - If directory exist then Get filename
     - If file exist in that directory then delete entry else report error
6. If choice =4 then
   - Get directory name
   - If directory exist then Get filename
     - If file exist in that directory then Display filename else report error
7. If choice =5 then Display files directory-wise
8. If choice =6 then Stop

**Program**

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

struct
{
    char dname[10], fname[10][10];int fcnt;
}dir[10];

main()
{
    int i, ch, dcnt, k;char f[30],
    d[30]; clrscr();
    dcnt=0; while(1)
    {
        printf("\n\n 1. Create Directory\t 2. Create File\t 3.
Delete File");
        printf("\n 4. Search File \t \t 5. Display \t 6. Exit \nEnter your choice -- ");
```

```
scanf("%d", &ch);switch(ch)
{
    case 1:
        printf("\n Enter name of directory -- ");scanf("%s",
        dir[dcnt].dname); dir[dcnt].fcnt = 0;
        dcnt++;
        printf("Directory  created");break;

    case 2:
        printf("\n Enter name of the directory -- ");scanf("%s", d);
        for(i=0; i<dcnt; i++) if(strcmp(d,dir[i].dname) ==
        0)
        {
            printf("Enter name of the file -- "); scanf("%s",
            dir[i].fname[dir[i].fcnt]);dir[i].fcnt++;
            printf("File  created");break;
        }
        if(i == dcnt)
            printf("Directory %s not found",d);break;

    case 3:
        printf("\nEnter name of the directory -- ");scanf("%s", d);
        for(i=0; i<dcnt; i++)
        {
            if(strcmp(d,dir[i].dname) == 0)
            {
                printf("Enter name of the file -- ");scanf("%s", f);
                for(k=0; k<dir[i].fcnt; k++)
                {
                    if(strcmp(f, dir[i].fname[k]) == 0)
                    {
                        printf("File %s is deleted ", f);dir[i].fcnt--;
                        strcpy(dir[i].fname[k],
                            dir[i].fname[dir[i].fcnt]);
                        goto jmp;
                    }
                }
                printf("File %s not found",f);goto jmp;
            }
        }
```

```
                printf("Directory %s not found",d);jmp : break;

            case 4:
                printf("\nEnter name of the directory -- ");scanf("%s", d);
                for(i=0; i<dcnt; i++)
                {
                    if(strcmp(d,dir[i].dname) == 0)
                    {
                        printf("Enter the name of the file -- ");scanf("%s", f);
                        for(k=0; k<dir[i].fcnt; k++)
                        {
                            if(strcmp(f, dir[i].fname[k]) == 0)
                            {
                                printf("File %s is found ", f);goto jmp1;
                            }
                        }
                        printf("File %s not found", f);goto jmp1;
                    }
                }
                printf("Directory %s not found", d);jmp1: break;

            case 5:
                if(dcnt == 0)
                    printf("\nNo Directory's ");else
                {
                    printf("\nDirectory\tFiles");
                    for(i=0;i<dcnt;i++)
                    {
                        printf("\n%s\t\t",dir[i].dname);
                        for(k=0;k<dir[i].fcnt;k++)
                            printf("\t%s",dir[i].fname[k]);
                    }
                }
                break;

            default:
                exit(0);
            }
        }
        getch();
}
```

**Output**

**Result**

Thus user files have been stored in their respective directories and retrieved easily.

| Exp. No. 15.A | CONTIGUOUS ALLOCATION |
|---|---|
| Date : | |

**Aim**

To implement file allocation on free disk space in a contiguous manner.

**File Allocation**

The three methods of allocating disk space are:

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

**Contiguous**

- Each file occupies a set of contiguous block on the disk.
- The number of disk seeks required is minimal.
- The directory contains address of starting block and number of contiguous block (length) occupied.
- Supports both sequential and direct access.
- First / best fit is commonly used for selecting a hole.

**Algorithm**

1. Assume no. of blocks in the disk as 20 and all are free.
2. Display the status of disk blocks before allocation.
3. For each file to be allocated:
   a. Get the *filename*, *start* address and file *length*
   b. If *start* + *length* > 20, then goto step 2.
   c. Check to see whether any block in the range (start, start + length-1) is allocated. If so, then go to step 2.
   d. Allocate blocks to the file contiguously from start block to start + length – 1.
4. Display directory entries.
5. Display status of disk blocks after allocation
6. Stop

**Program**

```
/* Contiguous Allocation - cntalloc.c */#include <stdio.h>
#include <string.h>

int num=0, length[10], start[10];char fid[20][4],
a[20][4];

void directory()
{
    int i;
    printf("\nFile Start Length\n");
```

```
    for(i=0; i<num; i++)
        printf("%-4s  %3d  %6d\n",fid[i],start[i],length[i]);
}

void display()
{
    int i;
    for(i=0; i<20; i++)printf("%4d",i);
    printf("\n"); for(i=0; i<20;
    i++)
        printf("%4s", a[i]);
}

main()
{
    int  i,n,k,temp,st,nb,ch,flag;char id[4];

    for(i=0; i<20; i++) strcpy(a[i], "");
    printf("Disk space before allocation:\n");display();
    do
    {
        printf("\nEnter File name (max 3 char) : ");scanf("%s", id);
        printf("Enter start block : ");scanf("%d",
        &st);
        printf("Enter no. of blocks : ");scanf("%d",
        &nb); strcpy(fid[num], id);
        length[num] = nb;flag = 0;

        if((st+nb) > 20)
        {
            printf("Requirement exceeds range\n");continue;
        }

        for(i=st; i<(st+nb); i++) if(strcmp(a[i],
            "") != 0)
                flag = 1;
        if(flag == 1)
        {
            printf("Contiguous allocation not possible.\n");continue;
        }
        start[num] =   st; for(i=st;
        i<(st+nb);  i++)
```

```
            strcpy(a[i], id);; printf("Allocation
        done\n");num++;

        printf("\nAny more allocation (1. yes / 2. no)? : ");scanf("%d", &ch);
    } while (ch == 1); printf("\n\t\t\tContiguous Allocation\n");
    printf("Directory:");
    directory();
    printf("\nDisk space after allocation:\n");display();
}
```

**Output**

**Result**

Thus contiguous allocation is done for files with the available free blocks.

| Exp. No. 15.B | LINKED FILE ALLOCATION |
|---|---|
| Date : | |

**Aim**

To st

**Linked**

- Each file is a linked list of disk blocks.
- The directory contains a pointer to first and last blocks of the file.
- The first block contains a pointer to the second one, second to third and so on.
- File size need not be known in advance, as in contiguous allocation.
- No external fragmentation.
- Supports sequential access only.

**Indexed**

- In indexed allocation, all pointers are put in a single block known as index block.
- The directory contains address of the index block.
- The $i^{th}$ entry in the index block points to $i^{th}$ block of the file.
- Indexed allocation supports direct access.
- It suffers from pointer overhead, i.e wastage of space in storing pointers.

**Algorithm**

1. Get no. of files
2. Accept filenames and no. of blocks fo each file
3. Obtrain start block for each file
4. Obtain other blocks for each file
5. Check block availability before allocation
6. If block is unavailable then report error
7. Accept file name
8. Display linked file allocation blocks for that file
9. Stop

**Program**

```
#include <stdio.h>
#include <conio.h>
#include  <string.h>


main()
{
     static int b[20], i, j, blocks[20][20];char F[20][20], S[20],
     ch;
     int sb[20], eb[20], x, n;clrscr();
     printf("\n Enter no. of Files ::");
     scanf("%d",&n);
```

```
    for(i=0;i<n;i++)
    {
        printf("\n Enter file %d name ::", i+1);scanf("%s",
        &F[i]);
        printf("\n Enter No. of blocks::", i+1);scanf("%d",&b[i]);
    }

    for(i=0;i<n;i++)
    {
        printf("\n Enter Starting block of file%d::",i+1);scanf("%d", &sb[i]);
        printf("\nEnter blocks for file%d::\n", i+1);for(j=0; j<b[i]-1;)
        {
            printf("\n Enter the %dblock ::", j+2);scanf("%d", &x);
            if(b[i] != 0)
            {
                blocks[i][j] = x;j++;
            }
            else
                printf("\n Invalid block::");
        }
    }

    printf("\nEnter the Filename :");scanf("%s",
    &S);
    for(i=0; i<n; i++)
    {
        if(strcmp(F[i],S) == 0)
        {
            printf("\nFname\tBsize\tStart\tBlocks\n");   printf("\n_____\n");
            printf("\n%s\t%d\t%d\t", F[i], b[i], sb[i]); printf("%d->",sb[i]);
            for(j=0; j<b[i]; j++)
            {
                if(b[i] != 0)
                    printf("%d->", blocks[i][j]);
            }
        }
    }
    printf("\n_____\n");
    getch();
}
```

**Output**

**Result**

Thus blocks for file were allocation using linked allocation method.