

SRI SIVASUBRAMANIYA NADAR COLLEGE OF ENGINEERING

**(AN AUTONOMOUS INSTITUTION,
AFFILIATED TO ANNA UNIVERSITY)**

Rajiv Gandhi Salai (OMR), Kalavakkam - 603 110.

LABORATORY RECORD

NAME : P.T.JAYANNTHAN

Reg. No. : 205001049

Dept. : CSE Sem. : IV Sec. : A'

ssn

**SRI SIVASUBRAMANIYA NADAR
COLLEGE OF ENGINEERING, CHENNAI**

(AN AUTONOMOUS INSTITUTION, AFFILIATED TO ANNA UNIVERSITY)

BONAFIDE CERTIFICATE

Certified that this is the bonafide record of the practical work done in the

UCS1111 Operating Systems Laboratory by

Name P.T.JAYANNT.HAN

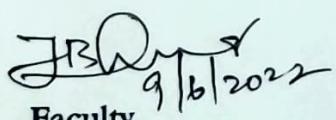
Register Number 205001049

Semester IV

Branch CSE

Sri Sivasubramaniya Nadar College of Engineering, Kalavakkam.

During the Academic year 2021 - 2022.


9/6/2022

Faculty



Head of the Department

Submitted for the Practical Examination held at SSNCE
on

Internal Examiner

External Examiner

INDEX

Name : P.T.JAYANNTHAN Reg. No. 205001049

Sem : IV, Sec : A

Ex. No.	Date of Expt.	Title of the Experiment	Page No.	Signature of the Faculty	Remarks
1	28/2/22	Study of System Calls and System Commands	7	7/3/22	
2	7/3/22	Simulation of System commands using calls	8	20/3/22	
3	20/3/22	CPU Scheduling : FCFS, SJF	9	21/3/22	
4	4/4/22	CPU Scheduling : Priority, RoundRobin	10	11/4/22	
5	11/4/22	Inter-Process Communication	11	12/4/22	
6	18/4/22	Producer and Consumer using Semaphores	12	23/4/22	
7	25/4/22	Deadlock : Avoidance and Detection	13	2/5/22	
8	2/5/22	Implementation of memory management system	14	3/5/22	
9	9/5/22	Implementation of Paging Technique	15	16/5/22	
10	16/5/22	Page Replacement Techniques	16	23/5/22	
11	23/5/22	Threading	17	30/5/22	
12	20/5/22	File Allocation Technique	18	30/5/22	
13	30/5/22	File organisation Technique	19	6/6/22	
				Completed	
				JBQ 9/6/22	*

LAB EXERCISE 1

Study of System calls and System Commands

Submission Date:10-03-2022

Name: Jayannthan P T

Dept: CSE 'A'

Roll No.: 205001049

1. Study the following system calls and system commands (using Linux manual pages)

a. System Commands

1) cp - -i

Name: Copy

Purpose: Copy SOURCE to DEST, or multiple SOURCE(s) to DIRECTORY.

Options: -i, --interactive

prompt before overwrite

Syntax: cp [OPTION]... SOURCE... DIRECTORY

Example: cp file1.txt file2.txt

2) mv - -i

Name: move

Purpose: Rename SOURCE to DEST, or move SOURCE(s) to DIRECTORY.

Options: -i, --interactive

prompt before overwrite

Syntax: mv [OPTION]... SOURCE... DIRECTORY

Example: mv file1.txt file2.txt

3) ls - -l

Name: list

Purpose: list directory contents

Options: -l use a long listing format

Syntax: ls [OPTION]... [FILE]...

Example: ls

4)grep - -c,-v

Name: Global Regular Expression Print

Purpose: print lines that match patterns

Options: -c uppress normal output; instead print a count of matching lines for each input file.

-v, --invert-match

option (see below), count non-matching lines

Syntax: grep [OPTION...] PATTERNS [FILE...]

Example: grep “^hello” file1

5)chmod

Name: Change mode

Purpose: changes the file mode bits of each given file according to mode, which can be either a symbolic representation of changes to make, or an octal number representing the bit pattern for the new mode bits.

Options: -R, --recursive

change files and directories recursively

Syntax: chmod [OPTION]... MODE[,MODE]... FILE...

chmod [OPTION]... OCTAL-MODE FILE...

Example: chmod 644 file

6)cat

Name: Concatenate

Purpose: concatenate files and print on the standard output

Options: -n number all output lines

Syntax: cat [OPTION]... [FILE]

Example: cat file1

7)mkdir

Name: Make directories

Purpose: print lines that match patterns

Options: - v print a message for each created directory

Syntax: mkdir [OPTION]... DIRECTORY

Example: mkdir files

8)rm

Name: Remove

Purpose: remove files or directories

Options: -i prompt before every removal

-R remove directories and their contents recursively

Syntax: rm [OPTION]... [FILE]

Example: rm files

9) rmdir

Name: Remove Directories

Purpose: remove empty directories

Options: -p remove DIRECTORY and its ancestors

 -v output a diagnostic for every directory processed

Syntax: rmdir [OPTION]... DIRECTORY

Example: rmdir files

10) wc

Name: Word Count

Purpose: print newline, word, and byte counts for each file

Options: -c, print the byte counts

 -m print the character counts

 -l print the newline counts

Syntax: wc [OPTION]... [FILE]

Example: wc file

11) who

Name: who

Purpose: show who is logged on

Options: -q, all login names and number of users logged on

 -t, last system clock change

Syntax: who [OPTION]

Example: who

12) head -n

Name: head

Purpose: output the first part of files

Options: -n, print the first n lines

Syntax: head [OPTION]... [FILE]

Example: head file

13) tail -n

Name: tail

Purpose: output the first part of files

Options: -n, print the first n lines

Syntax: head [OPTION]... [FILE]

Example: head file

14) nl

Name: number lines

Purpose: Write each FILE to standard output, with line numbers added.

Options: -i, line number increment at each line

-p, do not reset line numbers for each section

Syntax: nl [OPTION]... [FILE]

Example: nl file1

15) awk

Name: Aho, Weinberger, and Kernighan

Purpose: pattern scanning and processing language

Options: -F define the input field separator

-f Specify the pathname of the file progfile containing an awk program.

Syntax: awk [-F sepstring] [-v assignment]... program [argument...]

awk [-F sepstring] -f progfile [-f progfile]... [-v assignment]...[argument...]

Example: awk '{print}' file.txt

b. System Calls

1) fork()

Description: fork() creates a new process by duplicating the calling process. The new process is referred to as the *child* process. The calling process is referred to as the *parent* process.

Header File: unistd.h

Syntax: pid_t fork(void);

Arguments: none

Return type: *Negative Value:* creation of a child process was unsuccessful.

Zero: Returned to the newly created child process.

Positive value: Returned to parent or caller. The

value contains process ID of newly created child process.

2) execl()

Description: The `const char *arg` and subsequent ellipses can be thought of as `arg0, arg1, ..., argn`. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the filename associated with the file being executed. The list of arguments *must* be terminated by a null pointer, and, since these are variadic functions, this pointer must be cast (`char *`) `NULL`.

Header File: unistd.h

Syntax: `int execl(const char *pathname, const char *arg, /*, (char *) NULL */);`

Arguments: `char *pathname, char *arg`

Return type: return only if an error has occurred. The return value is -1,

3) getpid()

Description: `getpid()` returns the process ID (PID) of the calling process.

Header File: unistd.h

Syntax: `pid_t getpid(void);`

Arguments: Nil

Return type: returns the process ID of the parent of the current process. It never throws any error therefore is always successful.

4) getppid()

Description: `getppid()` returns the process ID of the parent of the calling process.

Header File: unistd.h

Syntax: `pid_t getppid(void);`

Arguments: Nil

Return type: returns the process ID of the parent of the current process. It never throws any error therefore is always successful.

5) exit()

Description: The exit() function causes normal process termination and the least significant byte of *status* is returned to the parent

Header File: stdlib.h

Syntax: void exit(int *status*);

Arguments: Status to return the parent process

Return type: No return value

6) wait()

Description: wait for process to change state

Header File: sys/wait.h

Syntax: pid_t wait(int *wstatus);

Arguments: wstatus - store status information in the int to which it points.

Return type: on success, returns the process ID of the terminated child; on failure, -1 is returned.

7) close()

Description: close() closes a file descriptor, so that it no longer refers to any file and may be reused.

Header File: unistd.h

Syntax: int close(int fd);

Arguments: fd is the last file descriptor referring to the underlying open file description

Return type: close() returns zero on success. On error, -1 is returned

8) opendir()

Description: The opendir() function opens a directory stream corresponding to the directory name, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

Header File: sys/types.h, dirent.h

Syntax: DIR *opendir(const char *name);

Arguments: directory name

Return type: functions return a pointer to the directory stream. On error, NULL is returned

9) readdir()

Description: The readdir() function returns a pointer to a *dirent* structure representing the next directory entry in the directory stream pointed to by *dirp*.

Header File: dirent.h

Syntax: struct dirent *readdir(DIR *dirp);

Arguments: directory pointer

Return type: returns a pointer to a dirent structure, If the end of the directory stream is reached, NULL is returned and errno is not changed. If an error occurs, NULL is returned and errno is set to indicate the error.

10) open()

Description: The open() system call opens the file specified by pathname. If the specified file does not exist, it may be created by open().

Header File: fcntl.h

Syntax: int open(const char *pathname, int flags, mode_t mode);

int open(const char *pathname, int flags);

Arguments: Pathname of the file, The argument flags must include one of the following access modes:

O_RDONLY, O_WRONLY, or O_RDWR

Return type: The return value of open() is a file descriptor. On error, -1 is returned

11) read()

Description: read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf and read from file descriptor

Header File: unistd.h

Syntax: ssize_t read(int fd, void *buf, size_t count);

Arguments: file descriptor fd, starting buffer size, read size

Return type: On success, the number of bytes read is returned, else -1 is returned

12) write()

Description: write() writes up to count bytes from the buffer starting at buf to the file referred to by the file descriptor fd.

Header File: unistd.h

Syntax: ssize_t write(int fd, void *buf, size_t count);

Arguments: file descriptor fd, starting buffer size, write size

Return type: On success, the number of bytes written is returned, else -1 is returned

13) creat()

Description: write() writes up to count bytes from the buffer starting at buf to the file referred to by the file descriptor fd.

Header File: sys/stat.h, fcntl.h

Syntax: int creat(const char *path, mode_t mode);

Arguments: path of file, open mode

Return type: On success, the number of bytes read is returned, else -1 is returned

14) sleep()

Description: sleep() causes the calling thread to sleep either until the number of real-time seconds specified in seconds have elapsed or until a signal arrives which is not ignored.

Header File: unistd.h>

Syntax: unsigned int sleep(unsigned int seconds);

Arguments: no. of seconds to sleep

Return type: Zero if the requested time has elapsed, or the number of seconds left to sleep, if the call was interrupted by a signal handler.

2. Develop a C program to understand the working of fork()

Algorithm:

- 1) Print a line before calling fork() to intimate it executes before calling fork().
- 2) Call fork() and store the return value in id.
- 3) If id is equal to zero then print it is a child process.
- 4) Else print it is a parent process.

Code:

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    printf("Before forking- this line is printed once\n\n");
    int id = fork();
    printf("After forking\n");
    if (id == 0)
        printf("Child process underway\n\n");
```

```

    else
        printf("Parent process underway\n\n");
    return 0;
}

```

Output:

```

Before forking...
After forking
Parent process underway

After forking
Child process underway

```

3. Develop a C program using system calls to open a file, read the contents of the same, display it and close the file. Use command line arguments to pass the file name to the program

Algorithm:

- 1) If argc greater than 2, then print error : too many arguments
- 2) Else if argc is lesser than 1, then print error : arguments required
- 3) Else
 - i. Open file using call open() using filename as argument provided in read-only mode and store the file pointer in file_descriptor
 - ii. If file_descriptor is equal to -1 then print error and exit
 - iii. Else then read the contents using call read() and store the return value in contents
 - iv. Print the contents
 - v. Close the file

Code:

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
int main(int argc, char *argv[])
{
    if (argc > 2)
        printf("Too many arguments\n");
    else if (argc < 1)
        printf("Atleast one argument required\n");
    else
    {
        int file_descriptor = open(argv[1], O_RDONLY);
        if (file_descriptor == -1)
            printf("File does not exist\n");
    }
}

```

```
else
{
    printf("File descriptor is: %d\n", file_descriptor);
    char contents[100];
    read(file_descriptor, contents, 100);
    printf("File contents : %s\n", contents);
    close(file_descriptor);
}
return 0;
}
```

Output:

```
File descriptor is: 3
File contents : #include <stdio.h>

int main(void) {
    printf("Hello World\n");
    return 0;
}
```

Learning Outcome:

- Learned system commands and system calls
- Implemented fork() in C program

LAB EXERCISE 2

Implementation of System calls

Submission Date:16-03-2022

Name: Jayannthan P T

Dept: CSE 'A'

Roll No.: 205001049

Implementing cp command in C using system calls

Algorithm:

- 1) If argc greater than 4, then print error : too many arguments
- 2) Else if argc is lesser than 2, then print error :more arguments required
- 3) Else
 - i. Open file using call open() using filename as argument provided in read-only mode and store the file pointer in file_descriptor1
 - ii. If file_descriptor1 is equal to -1 then print error and exit
 - iii. Else then read the contents using call read() and store the return value in contents
 - iv. Close the file
 - v. If "i" is present in argument then
 - a. Create a file using creat() and store the file pointer in file_descriptor2
 - b. If file_descriptor2 is equal to -1 then print error and exit
 - c. Write the "contents" into file_descriptor2 using write() call
 - d. Close the file
 - vi. Else then
 - a. Open a file using open() and store the file pointer in file_descriptor2
 - b. If file_descriptor2 is less than 0 then prompt for overwrite
 - If answer is yes overwrite in the same file
 - Else create new file and write the contents into it
 - c. Else then create new file and write the contents into it

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
int main(int argc, char *argv[])
{
```

```

if (argc > 4)
    printf("Too many arguements\n");
else if (argc < 2)
    printf("Atleast Two arguements required\n");
else
{
    printf("Opening file1:\n");
    int file_descriptor1 = open(argv[1], O_RDONLY);
    if (file_descriptor1 == -1)
        printf("Source File does not exist\n");
    else
    {
        char contents[100];
        printf("Reading file1:\n");
        int re = read(file_descriptor1, contents, 100);
        printf("Checking for i:\n");
        if (argc > 3 && strcmp(argv[3], "i") == 0)
        {
            printf("Found i:\n");
            printf("Creating file2:\n");

            int file_descriptor2 = creat(argv[2], S_IRUSR | S_IWUSR);
            if (file_descriptor2 < 0)
            {
                printf("!!!ERROR!!!\n");
            }
            else
            {

                printf("copying into file2:\n");
                int wr = write(file_descriptor2, contents, sizeof(contents));

                printf("Closing file2:\n");
                close(file_descriptor2);
                printf("\nSuccessfully Copied\n");
            }
            close(file_descriptor1);
        }
        else
        {

            printf("i not found:\n");

            printf("Checking file2:\n");
            int file_descriptor2 = open(argv[2], O_WRONLY);
            if (!(file_descriptor2 < 0))
            {
                char ch;
                printf("Overwrite %s file?(y/n) ", argv[2]);
                scanf(" %c", &ch);
                if (!(ch == 'y' || ch == 'Y'))
                {
                    close(file_descriptor2);
                    close(file_descriptor1);
                }
            }
        }
    }
}

```

```

    }
    else
    {

        printf("Creating file2:\n");
        int file_descriptor2 = creat(argv[2], S_IRUSR | S_IWUSR);
        if (file_descriptor2 < 0)
        {
            printf("!!!ERROR!!!\n");
        }
        else
        {

            printf("writing into file2:\n");
            int wr = write(file_descriptor2, contents, sizeof(contents));
            close(file_descriptor2);
            printf("\nSuccessfully Copied\n");
        }
        close(file_descriptor1);
    }
}
else
{
    printf("Creating file2:\n");

    int file_descriptor2 = creat(argv[2], S_IRUSR | S_IWUSR);
    if (file_descriptor2 < 0)
    {
        printf("!!!ERROR!!!\n");
    }
    else
    {

        printf("copying into file2:\n");
        int wr = write(file_descriptor2, contents, sizeof(contents));

        printf("Closing file2:\n");
        close(file_descriptor2);
        printf("\nSuccessfully Copied\n");
    }
    close(file_descriptor1);
}
}

return 0;
}
}

```

Output:

```
~/OS-Lab$ ./cp main.c temp.c
Opening file1:
Reading file1:
Checking for i:
i not found:
Checking file2:
Creating file2:
copying into file2:
Closing file2:
```

Successfully Copied

```
~/OS-Lab$ ./cp main.c temp.c i
Opening file1:
Reading file1:
Checking for i:
Found i:
Creating file2:
copying into file2:
Closing file2:
```

Successfully Copied

```
Successfully Copied
~/OS-Lab$ ./cp main.c temp.c
Opening file1:
Reading file1:
Checking for i:
i not found:
Checking file2:
Overwrite temp.c file?(y/n) y
Creating file2:
writing into file2:
```

Successfully Copied

Implementing ls command in C using system calls

Algorithm:

- 1) If argc greater than 4, then print error: too many arguments
- 2) Else if argc is lesser than 1, then print error: more arguments required
- 3) Else if argc is equal to 2
 - i. Open directory using call opendir() using directory-name as argument provided and store the file pointer in dir
 - ii. If dir is null then print error and exit
 - iii. Else display the names of all files and directories with name not starting with “.”
 - iv. Close the pointer dir
- 4) Else If “r” is present in argument then
 - i. Open directory using call opendir() using directory-name as argument provided and store the file pointer in dir
 - ii. If dir is null then print error and exit
 - iii. Else display the names of all files and directories with name recursively
 - iv. Close the pointer dir
- 5) Else If “a” is present in argument then
 - i. Open directory using call opendir() using directory-name as argument provided and store the file pointer in dir
 - ii. If dir is null then print error and exit

- iii. Else display the names of all files and directory name
- iv. Close the pointer dir

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <dirent.h>
#include <fcntl.h>

DIR *dir, *temp;
struct dirent *tmp;
void recursive(struct dirent *entry)
{
    if (entry == NULL)
    {
        return;
    }

    recursive(readdir(dir));
    printf(" %s\n", entry->d_name);
}

// void normal(struct dirent *entry, int n)
// {
//     if (entry == NULL)
//     {
//         return;
//     }
//     for (int i = 0; i < n; i++)
//     {
//         printf("\t");
//     }

//     printf(" %s\n", entry->d_name);

//     if (entry->d_type == DT_DIR && !(strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0 || (entry->d_name[0] == '.')))
//     {
//         temp = opendir(entry->d_name);
//         tmp = readdir(temp);
//         normal(&tmp, n + 1);
//         closedir(temp);
//     }
//     normal(readdir(dir), n);
// }

int main(int argc, char *argv[])
{
    if (argc > 4)
        printf("Too many arguments\n");
    else if (argc < 1)
        printf("Atleast one argument required\n");
```

```

else
{
    // printf("%d\n",argc);
    if (argc == 2)
    {
        struct dirent *entry;
        if ((dir = opendir(argv[1])) == NULL)
        {
            printf("CANNOT OPEN GIVEN DIRECTORY");
        }
        else
        {
            printf("Contents of the given:\n");
            while ((entry = readdir(dir)) != NULL)
            {
                if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") ==
0 || (entry->d_name[0] == '.'))
                    continue;
                printf(" %s\n", entry->d_name);
            }
            closedir(dir);
        }
    }
    else if (argc > 1 && strcmp(argv[2], "r") == 0)
    {
        printf("ls -R\n");
        struct dirent *entry;
        if ((dir = opendir(argv[1])) == NULL)
        {
            printf("CANNOT OPEN GIVEN DIRECTORY");
        }
        else
        {
            printf("Contents of the given:\n");
            recursive(&entry);
            closedir(dir);
        }
    }
    else if (argc > 1 && strcmp(argv[2], "a") == 0)
    {
        // DIR *dir;
        struct dirent *entry;
        if ((dir = opendir(argv[1])) == NULL)
        {
            printf("CANNOT OPEN GIVEN DIRECTORY");
        }
        else
        {
            printf("Contents of the given:\n");
            while ((entry = readdir(dir)) != NULL)
            {
                printf(" %s\n", entry->d_name);
            }
            closedir(dir);
        }
    }
}

```

```

        }
    }

    // else if (argc > 1 && strcmp(argv[2], "R") == 0)
    // {
    //     // DIR *dir;
    //     struct dirent *entry;
    //     if ((dir = opendir(argv[1])) == NULL)
    //     {
    //         printf("CANNOT OPEN GIVEN DIRECTORY");
    //     }
    //     else
    //     {
    //         printf("Contents of the given:\n");
    //         if ((entry = readdir(dir)) != NULL)
    //         {
    //             normal(&entry, 0);
    //         }

    //         closedir(dir);
    //     }
    // }
}

return 0;
}

```

Output:

```

~/OS-Lab$ ./ls .
Contents of the given:
replit.nix
Makefile
main.c
Assignment1
Assignment2
duplicate_main.c
full
ls
grep
cp
Assignment3
a.out

```

```

~/OS-Lab$ ./ls . a
Contents of the given:
.
..
.cache
.cccls-cache
replit.nix
.breakpoints
Makefile
.replit
main.c
Assignment1
Assignment2
duplicate_main.c
full
ls
grep
cp
Assignment3
a.out

```

```

~/OS-Lab$ ./ls . r
Contents of the given:
ls
a.out
Assignment3
cp
grep
full
duplicate_main.c
Assignment2
Assignment1
main.c
.replit
Makefile
.breakpoints
replit.nix
.cccls-cache
.cache
..

```

Implementing grep command in C using system calls

Algorithm:

- 1) If argc greater than 4, then print error: too many arguments
- 2) Else if argc is lesser than 2, then print error: more arguments required
- 3) Else if
 - i. Open file using call open() using filename as argument provided in read-only mode and store the file pointer in file_descriptor
 - ii. If file_descriptor is equal to -1 then print error and exit
 - iii. Else then read the contents using call read() and store the return value in buf
 - iv. Close the file
 - v. If argc is equal to 3 then
 - a. Iterate through the buf and store each line in line
 - b. Check for the input expression in line
 - c. If it is present then display
 - vi. Else If "c" is present in argument then
 - a. Iterate through the buf and store each line in line
 - b. Check for the input expression in line
 - c. If it is present then increment the count value
 - d. Display count

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <dirent.h>
#include <fcntl.h>
int main(int argc, char *argv[])
{
    if (argc > 4)
        printf("Too many arguments\n");
    else if (argc < 2)
        printf("Two arguments required\n");
    else
    {
        int file_descriptor = open(argv[2], O_RDONLY);
        if (file_descriptor == -1)
            printf("File does not exist\n");
        else
        {
            if (argc == 3)
            {
                // printf("Normal Grep\n");
                char line[100], buf[1024];
                int l = 0, i = 0, nr, count = 0;
                nr = read(file_descriptor, buf, 1024);
                close(file_descriptor);
                while (l < nr)
```

```

    {
        for (i = 0; buf[l] != '\n' && l < nr; i++, l++)
        {
            line[i] = buf[l];
        }
        line[i] = '\0';
        l++;
        if (strstr(line, argv[1]))
            printf("%s\n", line);
    }
}
else if (argc > 3 && strcmp(argv[3], "c") == 0)
{
    // printf("Grep -c\n");
    char line[100], buf[1024];
    int l = 0, i = 0, nr, count = 0;
    nr = read(file_descriptor, buf, 1024);
    close(file_descriptor);
    while (l < nr)
    {
        for (i = 0; buf[l] != '\n' && l < nr; i++, l++)
        {
            line[i] = buf[l]; // extracting lines
        }
        line[i] = '\0';
        l++;
        if (strstr(line, argv[1]))
        {
            count++;
        }
    }
    printf("%d\n", count);
}
}
return 0;
}

```

Output:

```

~/OS-Lab$ ./grep main main.c
int main(void) {
~/OS-Lab$ ./grep main main.c c
1

```

Learning Outcome:

- Implemented various system commands in C using system calls
- Learned to handle system calls in C program

LAB EXERCISE 3

Implementation of CPU Scheduling Policies

Submission Date:23-03-2022

Name: Jayannthan P T

Dept: CSE 'A'

Roll No.: 205001049

Develop a menu driven C program to implement the CPU Scheduling Algorithms FCFS and SJF

Algorithm for FCFS:

- 1) Get total no of process from the user.
- 2) Get process id, arrival time, burst time for all process.
- 3) Sort the process based on arrival time.
- 4) Loop until all process ends
 - a) Set waiting time of process as sum of previous process waiting time and burst time by subtracting arrival time of the process
 - b) Add current waiting time to total waiting time
 - c) Set turnaround time of process as sum of waiting time and burst time
 - d) Add current turnaround time to total turnaround time
- 5) Calculate average waiting time by dividing total waiting time by total no of process
- 6) Calculate average turnaround time by dividing total turnaround time by total no of process
- 7) Print process table
- 8) Print Gantt Chart

Algorithm for SJF - Non-Preemptive:

- 1) Get total no of process from the user.
- 2) Get process id, burst time for all process.
- 3) Sort the process based on burst time.
- 4) Loop until all process ends
 - a) Set waiting time of process as sum of previous process waiting time and burst time by subtracting arrival time of the process
 - b) Add current waiting time to total waiting time
 - c) Set turnaround time of process as sum of waiting time and burst time
 - d) Add current turnaround time to total turnaround time
- 5) Calculate average waiting time by dividing total waiting time by total no of process

- 6) Calculate average turnaround time by dividing total turnaround time by total no of process
- 7) Print process table
- 8) Print Gantt Chart

Code:

```
/*Algorithm: 1. Read the following a. Number of processes b. Process IDs c. Arrival time  
for each process d. Burst Time for each process 2. Design a menu with FCFSand SJFOptions  
3. Upon selection of menu option apply the corresponding algorithm. 4. Compute the  
Turnaround Time, Average waiting Time for each of the algorithm. 5. Tabularize the  
results. 6. Display the Gantt Chart.*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <dirent.h>
#include <cctype.h>

typedef struct process
{
    char pid[3];
    int arrival, burst, turnaround, waiting, completion;
} process;

void print_gantt_chart(process p[], int n)
{
    printf("\n\nGantt-Chart\n");
    int i, j;
    printf(" ");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < p[i].burst; j++)
            printf("--");
        printf(" ");
    }
    printf("\n|");

    for (i = 0; i < n; i++)
    {
        for (j = 0; j < p[i].burst - 1; j++)
            printf(" ");
        printf("%s", p[i].pid);
        for (j = 0; j < p[i].burst - 1; j++)
            printf(" ");
        printf("|");
    }
    printf("\n ");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < p[i].burst; j++)
            printf(" ");
```

```

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

    printf("0");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < p[i].burst; j++)
            printf(" ");
        if (p[i].turnaround > 9)
            printf("\b");
        printf("%d", p[i].turnaround);
    }
    printf("\n");
}

int main()
{
    int no_of_process;
    int totalwaitingtime = 0, totalturnaround = 0;
    int pos;
    char ch = 'y';
    process p[100];
    while (ch == 'y' || ch == 'Y')
    {
        int choice;
        printf("\nMenu\n\t1.FCFS\n\t2.SJF-Non Preemptive\n\t3.SJF-Preemptive\nEnter
Choice:");
        scanf(" %d", &choice);
        switch (choice)
        {
        case 1:
        {
            printf("\nFCFS\n");
            int no_of_process;
            printf("\nNumber of Processes :");
            scanf(" %d", &no_of_process);
            for (int i = 0; i < no_of_process; i++)
            {
                printf("\n\nProcess %d\n", i + 1);
                printf("Process ID: ");
                scanf(" %s", p[i].pid);
                printf("Arrival Time :");
                scanf(" %d", &p[i].arrival);
                printf("Burst Time :");
                scanf(" %d", &p[i].burst);
            }

            process temppro;
            for (int i = 0; i < no_of_process; i++)
            {
                pos = i;
                for (int j = i + 1; j < no_of_process; j++)

```

```

        {
            if (p[j].arrival < p[pos].arrival)
                pos = j;
        }
        temppro = p[i];
        p[i] = p[pos];
        p[pos] = temppro;
    }
    totalwaitingtime = 0, totalturnaround = 0;
    p[0].waiting = 0;
    p[0].turnaround = p[0].burst;
    totalturnaround += p[0].turnaround;
    for (int i = 1; i < no_of_process; i++)
    {
        if (p[i - 1].waiting + p[i - 1].burst - p[i].arrival > 0)
        {
            p[i].waiting = p[i - 1].waiting + p[i - 1].burst - p[i].arrival;
        }
        else
        {
            p[i].waiting = 0;
        }
        totalwaitingtime += p[i].waiting;
        p[i].turnaround = p[i].burst + p[i].waiting;
        totalturnaround += p[i].turnaround;
    }
    printf("\nP_ID\tArrival Time\tBurst Time\tTurnaround Time\tWaiting Time\n");
    for (int i = 0; i < no_of_process; i++)
    {
        printf("%s\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].arrival,
p[i].burst, p[i].turnaround, p[i].waiting);
    }

    float avgwaiting = (float)(totalwaitingtime / no_of_process);
    float avgt turnaround = (float)(totalturnaround / no_of_process);
    printf("\n\tAVERAGE AverageTurnaroundTime=%f\tAverageWaitingTime=%f\n",
avgt turnaround, avgwaiting);
    print_gantt_chart(p, no_of_process);
    break;
}

case 2:
{
    printf("\nSJF-Non Preemptive\n");
    int no_of_process;
    printf("\nNumber of Processes :");
    scanf(" %d", &no_of_process);
    int totalwaitingtime = 0, totalturnaround = 0;
    char pid[no_of_process][5];
    for (int i = 0; i < no_of_process; i++)
    {
        printf("\n\nProcess %d\n", i + 1);
        printf("Process ID: ");
        scanf(" %s", p[i].pid);

```

```

// printf("Arrival Time :");
p[i].arrival = 0;
printf("Burst Time :");
scanf(" %d", &p[i].burst);
}

process temppro;
for (int i = 0; i < no_of_process; i++)
{
    pos = i;
    for (int j = i + 1; j < no_of_process; j++)
    {
        if ((p[j].arrival < p[pos].arrival) || ((p[j].arrival <=
p[pos].arrival) && (p[j].burst < p[pos].burst)))
        {
            pos = j;
        }
    }

    temppro = p[i];
    p[i] = p[pos];
    p[pos] = temppro;
}
totalwaitingtime = 0, totalturnaround = 0;
p[0].waiting = 0;
p[0].turnaround = p[0].burst;
totalturnaround += p[0].turnaround;
for (int i = 1; i < no_of_process; i++)
{
    if (p[i - 1].waiting + p[i - 1].burst - p[i].arrival > 0)
    {
        p[i].waiting = p[i - 1].waiting + p[i - 1].burst - p[i].arrival;
    }
    else
    {
        p[i].waiting = 0;
    }
    totalwaitingtime += p[i].waiting;
    p[i].turnaround = p[i].burst + p[i].waiting;
    totalturnaround += p[i].turnaround;
}
printf("\nP_IDs\tBurst Time\tTurnaround Time\t\tWaiting Time\n");
for (int i = 0; i < no_of_process; i++)
{
    printf("%s\t%d\t%d\t\t%d\n", p[i].pid, p[i].burst, p[i].turnaround,
p[i].waiting);
}

float avgwaiting = (float)(totalwaitingtime / no_of_process);
float avgturnaround = (float)(totalturnaround / no_of_process);
printf("\n\t\tAVERAGE AverageTurnaroundTime=% .2f\tAverageWaitingTime=% .2f\n",
avgturnaround, avgwaiting);
print_gantt_chart(p, no_of_process);
break;

```

```

    }

case 3:
{
    printf("\nSJF - Preemptive\n");
    int no_of_process;
    printf("\nNumber of Processes :");
    scanf(" %d", &no_of_process);
    for (int i = 0; i < no_of_process; i++)
    {
        printf("\n\nProcess %d\n", i + 1);
        printf("Process ID: ");
        scanf(" %s", p[i].pid);
        printf("Arrival Time :");
        scanf(" %d", &p[i].arrival);
        printf("Burst Time :");
        scanf(" %d", &p[i].burst);
    }

    process temppro;
    for (int i = 0; i < no_of_process; i++)
    {
        pos = i;
        for (int j = i + 1; j < no_of_process; j++)
        {
            if (p[j].arrival < p[pos].arrival)
                pos = j;
        }
        temppro = p[i];
        p[i] = p[pos];
        p[pos] = temppro;
    }
    int rem_time[no_of_process];
    for (int i = 0; i < no_of_process; i++)
    {
        rem_time[i] = p[i].burst;
    }
    for (int cur_time = 0, completed = 0; completed < no_of_process; cur_time++)
    {
        int idx = -1;
        for (int i = 0; i < no_of_process; i++)
        {
            if (p[i].arrival <= cur_time && rem_time[i] > 0 && (idx == -1 || rem_time[i] < rem_time[idx]))
            {
                idx = i;
            }
        }
        if (idx != -1)
        {
            rem_time[idx]--;
            if (rem_time[idx] == 0)
            {
                completed++;
                p[idx].completion = cur_time;
            }
        }
    }
}

```

```

        p[idx].turnaround = p[idx].completion - p[idx].arrival + 1;
        p[idx].waiting = p[idx].turnaround - p[idx].burst;
    }
}

for (int i = 0; i < no_of_process; i++)
{
    // p[i].turnaround=p[i].burst+p[i].waiting;
    totalwaitingtime += p[i].waiting;
    totalturnaround += p[i].turnaround;
}
printf("\nP_ID\tArrival Time\tBurst Time\tTurnaround Time\t\tWaiting Time\n");
for (int i = 0; i < no_of_process; i++)
{
    printf("%s\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].arrival,
p[i].burst, p[i].turnaround, p[i].waiting);
}

float avgwaiting = (float)(totalwaitingtime / no_of_process);
float avgturnaround = (float)(totalturnaround / no_of_process);
printf("\n\t\tAVERAGE AverageTurnaroundTime=%.2f\tAverageWaitingTime=%.2f\n",
avgturnaround, avgwaiting);
print_gantt_chart(p, no_of_process);
break;
}
case 4:
    printf("Exiting...");
    return 0;
}
printf("\nWant to Continue (Y/N):");
scanf(" %c", &ch);
}
return 0;
}

```

Output:

```
Number of Processes :4

Process 1
Process ID: P1
Arrival Time :0
Burst Time :10

Process 2
Process ID: P2
Arrival Time :3
Burst Time :2

Process 3
Process ID: P3
Arrival Time :1
Burst Time :6

Process 4
Process ID: P
Arrival Time :5
Burst Time :4

Menu
1.FCFS
2.SJF-Non Preemptive
3.SJF-Preemptive
Enter Choice:1

FCFS
```

```
Number of Processes :4

Process 1
Process ID: P1
Arrival Time :0
Burst Time :10

Process 2
Process ID: P2
Arrival Time :3
Burst Time :2

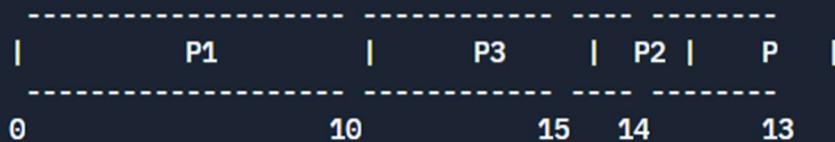
Process 3
Process ID: P3
Arrival Time :1
Burst Time :6

Process 4
Process ID: P
Arrival Time :5
Burst Time :4
```

P_ID	Arrival Time	Burst Time	Turnaround Time	Waiting Time
P1	0	10	10	0
P3	1	6	15	9
P2	3	2	14	12
P	5	4	13	9

AVERAGE AverageTurnaroundTime=13.00 AverageWaitingTime=7.00

Gantt-Chart



SJF-Non Preemptive

Number of Processes :4

Process 1

Process ID: 1

Burst Time :6

Process 2

Process ID: 2

Burst Time :8

Process 3

Process ID: 3

Burst Time :7

Process 4

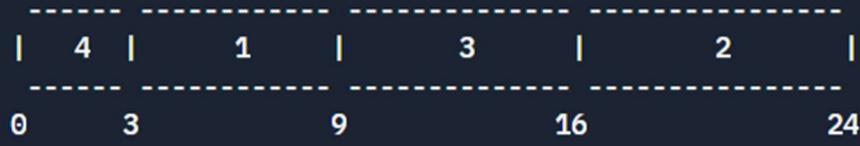
Process ID: 4

Burst Time :3

P_IDs	Burst Time	Turnaround Time	Waiting Time
4	3	3	0
1	6	9	3
3	7	16	9
2	8	24	16

AVERAGE AverageTurnaroundTime=13.00 AverageWaitingTime=7.00

Gantt-Chart



Process 1
Process ID: 1
Arrival Time :2
Burst Time :6

Process 2
Process ID: 2
Arrival Time :5
Burst Time :2

Process 3
Process ID: 3
Arrival Time :1
Burst Time :8

Process 4
Process ID: 4
Arrival Time :0
Burst Time :3

Process 5
Process ID: 5
Arrival Time :4
Burst Time :4

P_ID	Arrival Time	Burst Time	Turnaround Time	Waiting Time
4	0	3	3	0
3	1	8	22	14
1	2	6	13	7
5	4	4	6	2
2	5	2	2	0

AVERAGE AverageTurnaroundTime=28.00 AverageWaitingTime=15.00

Learning Outcome:

- Implemented FCFS Scheduling and SJF Scheduling in C program
- Displayed Gantt Chart for the above scheduling methods

LAB EXERCISE 4

Implementation of CPU Scheduling Policies

Submission Date:07-04-2022

Name: Jayannthan P T

Dept: CSE 'A'

Roll No.: 205001049

Develop a menu driven C program to implement the CPU Scheduling Algorithms

1. Priority (Non-Preemptive and Preemptive)
2. Round Robin

Algorithm for Priority Preemptive:

- 1) Get total no of process from the user.
- 2) Get process id, arrival time, burst time, priority for all process.
- 3) Take a temporary burst time(*rem_time*) to have a value of remining burst time of all the process
- 4) Have count of completed process, current time.
- 5) Loop until completed less than total no of process
 - a) Compare priority of current running job and new entering job at that current time
 - b) If priority is greater then stop the current processor and update *rem_time* and begin the new process
 - c) Else continue until the process ends
 - d) While the process ends set completion time as current time, waiting time as completion time minus sum of arrival time and burst time, turnaround time as sum of waiting time and burst time, update total turnaround time and total waiting time
- 6) Calculate average waiting time by dividing total waiting time by total no of process
- 7) Calculate average turnaround time by dividing total turnaround time by total no of process
- 8) Print process table
- 9) Print Gantt Chart

Algorithm for Round Robin:

- 1) Get total no of process, time quantum from the user.

- 2) Get process id, arrival time, burst time for all process.
- 3) Take a temporary burst time(rem_time) to have a value of remining burst time of all the process
- 4) Sort the process based on arrival time.
- 5) Have count of completed process, current time.
- 6) Loop until all process ends
 - a) If rem_time is less than or equal to quantum then current time is sum of current time and burst time of the process and set turnaround time as current time minus arrival time, waiting time as current time minus sum of arrival time and burst time, update total turnaround time and total waiting time
 - b) Else update rem_time as rem_time minus quantum and current time is sum of current time and quantum
- 7) Calculate average waiting time by dividing total waiting time by total no of process
- 8) Calculate average turnaround time by dividing total turnaround time by total no of process
- 9) Print process table
- 10) Print Gantt Chart

Code:

```

/*Develop a menu driven C program to implement the CPU Scheduling Algorithms - Priority
(Non-Preemptive and Preemptive) and Round Robin
Algorithm: 1. Read the following a. Number of p b. Process IDs c. Arrival time for each
process d. Burst Time for each process 2. Design a menu with FCFSand SJFOptions 3. Upon
selection of menu option apply the corresponding algorithm. 4. Compute the Turnaround
Time, Average waiting Time for each of the algorithm. 5. Tabularize the results. 6.
Display the Gantt Chart.*/
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>

typedef struct process
{
    char pid[3];
    int arrival, burst, teempburst, turnaround, waiting, completion, priority;
} process;

void print_gantt_chart(process p[], int n)
{
    printf("\n\nGantt-Chart\n");
    int i, j;
    printf(" ");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < p[i].burst; j++)
            printf("--");
        printf(" ");
    }
    printf("\n|");
}

```

```

        for (i = 0; i < n; i++)
    {
        for (j = 0; j < p[i].burst - 1; j++)
            printf(" ");
        printf("P%s", p[i].pid);
        for (j = 0; j < p[i].burst - 1; j++)
            printf(" ");
        printf("|");
    }
    printf("\n ");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < p[i].burst; j++)
            printf("--");
        printf(" ");
    }
    printf("\n");

    printf("0");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < p[i].burst; j++)
            printf(" ");
        if (p[i].turnaround > 9)
            printf("\b");
        printf("%d", p[i].turnaround);
    }
    printf("\n");
}

int main()
{
    int no_of_process;
    int totalwaitingtime = 0, totalturnaround = 0;
    int pos;
    char ch = 'y';
    process p[100];
    while (ch == 'y' || ch == 'Y')
    {
        int choice;
        printf("\nMenu\n\t1.Priority-Non Preemptive\n\t2.Priority-Preemptive\n\t3.Round
Robin\n\t4.Exit\n\t\nEnter Choice:");
        scanf(" %d", &choice);
        switch (choice)
        {
        case 1:
        {
            printf("\nPriority-Non Preemptive\n");
            int no_of_process;
            printf("\nNumber of p :");
            scanf(" %d", &no_of_process);
            for (int i = 0; i < no_of_process; i++)
            {
                printf("\n\nProcess %d\n", i + 1);

```

```

        printf("Process ID: ");
        scanf(" %s", p[i].pid);
        printf("Burst Time :");
        scanf(" %d", &p[i].burst);
        printf("Priority :");
        scanf(" %d", &p[i].priority);
    }

process temppro;
for (int i = 0; i < no_of_process; i++)
{
    pos = i;
    for (int j = i + 1; j < no_of_process; j++)
    {
        if (p[j].priority < p[pos].priority)
            pos = j;
    }
    temppro = p[i];
    p[i] = p[pos];
    p[pos] = temppro;
}
totalwaitingtime = 0, totalturnaround = 0;
p[0].waiting = 0;
p[0].turnaround = p[0].burst;
totalturnaround += p[0].turnaround;
for (int i = 1; i < no_of_process; i++)
{
    if (p[i - 1].waiting + p[i - 1].burst - p[i].arrival > 0)
    {
        p[i].waiting = p[i - 1].waiting + p[i - 1].burst - p[i].arrival;
    }
    else
    {
        p[i].waiting = 0;
    }
    totalwaitingtime += p[i].waiting;
    p[i].turnaround = p[i].burst + p[i].waiting;
    totalturnaround += p[i].turnaround;
}
printf("\nP_ID\tBurst Time\tPriority\tWaiting Time\t\tTurnaround Time\n");
for (int i = 0; i < no_of_process; i++)
{
    printf("%s\t%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].burst,
p[i].priority, p[i].waiting, p[i].turnaround);
}

float avgwaiting = (float)(totalwaitingtime / no_of_process);
float avgturnaround = (float)(totalturnaround / no_of_process);
printf("\n\t\tAVERAGE \tWaitingTime =%.2f\t TurnaroundTime =%.2f\n",
totalwaitingtime / no_of_process, totalturnaround / no_of_process);
print_gantt_chart(p, no_of_process);
break;
}

```

```

case 2:
{
    printf("\nPriority-Preemptive\n");
    int no_of_process;
    printf("\nNumber of process :");
    scanf(" %d", &no_of_process);
    int tempburst[100];
    for (int i = 0; i < no_of_process; i++)
    {
        printf("\n\nProcess %d\n", i + 1);
        printf("Process ID: ");
        scanf(" %s", p[i].pid);
        printf("Arrival Time :");
        scanf(" %d", &p[i].arrival);
        printf("Burst Time :");
        scanf(" %d", &p[i].burst);
        printf("Priority :");
        scanf(" %d", &p[i].priority);
        tempburst[i] = p[i].burst;
        p[i].teempburst = p[i].burst;
    }

    int rem_time[no_of_process];
    for (int i = 0; i < no_of_process; i++)
        rem_time[i] = p[i].burst;

    process tempro[100];
    int tempcount = 0;
    int completed = 0;
    int cur_time = 0;
    while (completed < no_of_process)
    {
        int idx = -1;
        for (int i = 0; i < no_of_process; i++)
        {
            if (p[i].arrival <= cur_time && rem_time[i] > 0 && (idx == -1 ||
p[i].priority < p[idx].priority))
                idx = i;
        }
        cur_time++;
        if (tempcount != 0 && strcmp(tempro[tempcount-1].pid,p[idx].pid))
tempcount--;
        else
        {
            tempro[tempcount] = p[idx];
        }
        strcpy(tempro[tempcount].pid,p[idx].pid);
        tempro[tempcount].burst++;
        tempro[tempcount].turnaround = cur_time;
        tempcount++;
        rem_time[idx]--;
        if(rem_time[idx]==0)
        {
            completed++;
        }
    }
}

```

```

        p[idx].completion = cur_time;
        p[idx].waiting = p[idx].completion - p[idx].arrival - p[idx].burst;
        p[idx].turnaround = p[idx].burst + p[idx].waiting;
    }
}

for (int i = 0; i < no_of_process; i++)
{
    totalwaitingtime+=p[i].waiting;
    totalturnaround+=p[i].turnaround;
}
printf("\nP_ID\tArrival Time\tBurst Time\tPriority\tWaiting Time\t\tTurnaround
Time\n");
for (int i = 0; i < no_of_process; i++)
{
    printf("%s\t%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].arrival,
p[i].burst, p[i].priority, p[i].waiting, p[i].turnaround);
}
float avgwaiting = (float)(totalwaitingtime / no_of_process);
float avgturnaround = (float)(totalturnaround / no_of_process);
printf("\n\t\tAVERAGE \tWaitingTime =%.2f\t TurnaroundTime =%.2f\n",
avgwaiting, avgturnaround);
process temmptemppro[100];
int temptempcount=-1;
for(int i=0;i<tempcount;i++)
{
    if(strcmp(tempo[i+1].pid,tempo[i].pid)!=0)
    {
        temmptemppro[++temptempcount]=tempo[i];
    }
}

print_gantt_chart(temmptemppro, temptempcount+1);
print_gantt_chart(tempo, tempcount);
break;
}
case 3:
{
    printf("\nRound Robin\n");
    int no_of_process;
    int quantum;
    printf("\nNumber of p :");
    scanf(" %d", &no_of_process);
    int temp_nop = no_of_process;
    process tempopro[100];
    int tempburst[100];
    int count = 0;
    int tempcount = -1;
    for (int i = 0; i < no_of_process; i++)
    {
        printf("\n\nProcess %d\n", i + 1);
        printf("Process ID: ");
        scanf(" %s", p[i].pid);

```

```

        printf("Arrival Time :");
        scanf(" %d", &p[i].arrival);
        printf("Burst Time :");
        scanf(" %d", &p[i].burst);
        tempburst[i] = p[i].burst;
    }

process temppro;
for (int i = 0; i < no_of_process; i++)
{
    pos = i;
    for (int j = i + 1; j < no_of_process; j++)
    {
        if (p[j].arrival < p[pos].arrival)
            pos = j;
    }
    temppro = p[i];
    p[i] = p[pos];
    p[pos] = temppro;
}

totalwaitingtime = 0, totalturnaround = 0;
p[0].waiting = 0;
printf("\nTime Quantum :");
scanf(" %d", &quantum);
for (int sum = 0, i = 0; temp_nop != 0;)
{
    if (tempburst[i] <= quantum && tempburst[i] > 0)
    {
        int temptempburst = tempburst[i];
        sum = sum + tempburst[i];
        tempburst[i] = 0;
        count = 1;

        tempcount++;
        strcpy(temppro[tempcount].pid, p[i].pid);
        temppro[tempcount].burst = temptempburst;
        temppro[tempcount].arrival = p[i].arrival;
        temppro[tempcount].turnaround = sum;
    }
    else if (tempburst[i] > 0)
    {

        tempburst[i] = tempburst[i] - quantum;
        sum = sum + quantum;

        tempcount++;
        strcpy(temppro[tempcount].pid, p[i].pid);
        temppro[tempcount].burst = quantum;
        temppro[tempcount].arrival = p[i].arrival;
        temppro[tempcount].turnaround = sum;
        temppro[tempcount].waiting = sum - p[i].arrival - quantum;
    }
    if (tempburst[i] == 0 && count == 1)

```

```

{
    temp_nop--;

    p[i].turnaround = sum - p[i].arrival;
    p[i].waiting = sum - p[i].arrival - p[i].burst;
    totalwaitingtime = totalwaitingtime + sum - p[i].arrival - p[i].burst;
    totalturnaround = totalturnaround + sum - p[i].arrival;
    count = 0;
}

if (p[i + 1].arrival <= sum)
{
    i++;
}
else
{
    i = 0;
}
printf("\nP_ID\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");
for (int i = 0; i < no_of_process; i++)
{
    printf("%s\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].arrival,
p[i].burst, p[i].waiting, p[i].turnaround);
}
float avgwaiting = (float)(totalwaitingtime / no_of_process);
float avgturnaround = (float)(totalturnaround / no_of_process);
printf("\n\t\tAVERAGE \tWaitingTime =%.2f\t TurnaroundTime =%.2f\n",
avgwaiting, avgturnaround);
print_gantt_chart(tempopro, tempcount + 1);
break;
}
case 4:
    printf("Exiting...");
    return 0;
}
printf("\nWant to Continue (Y/N):");
scanf(" %c", &ch);
}
return 0;
}

```

Output:

Menu

- 1.Priority-Non Preemptive
- 2.Priority-Preemptive
- 3.Round Robin
- 4.Exit

Enter Choice:3

Round Robin

Number of p :5

Process 1

Process ID: 1

Arrival Time :0

Burst Time :10

Process 2

Process ID: 2

Arrival Time :0

Burst Time :1

Process 3

Process ID: 3

Arrival Time :0

Burst Time :2

Process 5

Process ID: 5

Arrival Time :0

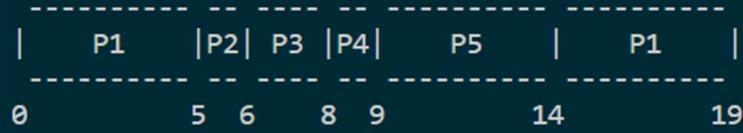
Burst Time :5

Time Quantum :5

P_ID	Arrival Time	Burst Time	Waiting Time	Turnaround Time
1	0	10	9	19
2	0	1	5	6
3	0	2	6	8
4	0	1	8	9
5	0	5	9	14

AVERAGE WaitingTime =7.00 TurnaroundTime =11.00

Gantt-Chart



Priority-Preemptive

Number of process :3

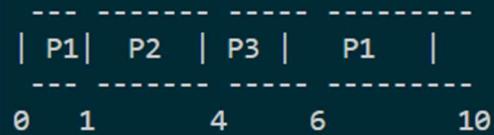
Process 1
Process ID: 1
Arrival Time :0
Burst Time :5
Priority :3

Process 2
Process ID: 2
Arrival Time :1
Burst Time :3
Priority :1

Process 3
Process ID: 3
Arrival Time :2
Burst Time :2
Priority :2

PID	Arrival_Time	Burst_Time	Waiting_Time	Completion_Time	Turnaround_Time
1	0	5	5	10	10
2	1	3	0	4	3
3	2	2	2	6	4

Average Waiting time : 2.33 Turn_around time : 5.67



Learning Outcome:

- Implemented Pre-emptive Priority Scheduling and Round Robin Scheduling in C program
- Displayed Gantt Chart for the above scheduling methods

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UCS1411 - OPERATING SYSTEMS LAB

LAB EXERCISE 5

Inter Process Communication

Submission Date:07-04-2022

Name: Jayannthan P T

Dept: CSE 'A'

Roll No.: 205001049

System Calls

1. Name:shmget()

- a. Description: returns the identifier for the shared memory segment associated with the value of the argument key
- b. Header file:sys/shm.h
- c. Syntax:int shmget(key_t key,size_t size,int shmflg);
- d. Arguments:
- e. Key - it identifies the shared memory segment
- f. Size - size of the shared segment
- g. Shmflg - specifies the required shared memory flag(s). Need to pass permissions as well.
- h. Return type:
- i. Success:returns valid shared memory identifier
- j. Failure:returns -1 and errno is set to indicate the error

2. Name:shmat()

- a. Description: attaches the shared memory segment identified by shmid to the address space of the calling process
- b. Header file:sys/shm.h
- c. Syntax:void *shmat(int shmid, const void *shmaddr, int shmflg);
- d. int shmdt(const void *shmaddr);
- e. Arguments:
- f. Shmid - shared memory identifier, which is the return value of shmget() system call.
- g. Shmaddr - specifies the address that attaches to the calling process.
- h. Shmflg - specifies the required shared memory flag/s.
- i. Return type:
- j. Success: returns address of the attached shared memory segment
- k. Failure:returns -1

3. Name:shmdt()

- a. Description:detaches the shared memory segment located at the address specified by shmaddr from the address space of the calling process
- b. Header file:sys/types.h
- c. Syntax:int shmdt(const void *shmaddr)

- d. Arguments:Shmaddr - the address of the shared memory segment to be detached. The to-be-detached segment must be the address returned by the shmat() system call.
 - e. Return type:
 - f. Success:returns 0
 - g. Failure:returns -1
4. Name:shmctl()
- a. Description: performs the control option specified by cmd on the system shared memory segment whose identifier is given by shmid
 - b. Header file:sys/shm.h
 - c. Syntax:int shmctl(int shmid,int cmd,struct shmid_ds *buf);
 - d. Arguments:
 - e. Shmid - shared memory identifier, which is the return value of shmget() system call.
 - f. Cmd - command to perform the required control operation on the shared memory segment.
 - g. Buf - pointer to the shared memory structure named struct shmid_ds.
 - h. Return type:
 - i. Success:returns 0
 - j. Failure:returns -1.

Develop the following applications that uses interprocess communication concepts using shared memory.

1. Develop an application for getting a name in parent and convert it into uppercase in child using shared memory.

Algorithm:

- 1) Fork() is called and the children id is stored in pid
- 2) If pid is equal to zero then
 - a) Created a unique key for a project using ftok() and stored in key
 - b) For the key, shared memory is allotted using shmget and returned id is stored in shmid
 - c) Get input from the user which to be stored in shared memory
 - d) Detach from the memory
- 3) Else then
 - a) Created a unique key for a project using ftok() and stored in key
 - b) For the key, shared memory is allotted using shmget and returned id is stored in shmid
 - c) Read the data in shared memory and convert it to uppercase then display it
 - d) Detach from the memory
 - e) Erase the shared memory

Code:

```
#include <sys/ipc.h>
#define NULL 0
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
```

```

#include <sys/wait.h>
#include <stdio_ext.h>

int main()
{
    int pid = fork();
    if (pid == 0)
    {
        int shmid = shmget(111, 1024, 0666 | IPC_CREAT);
        char *str = (char *)shmat(shmid, (void *)0, 0);
        printf("Data to be written in memory:");

        fgets(str, 100, stdin);
        // printf("Data written in memory: %s\n", str);
        shmdt(str);
    }
    else
    {
        wait(NULL);
        int shmid = shmget(111, 1024, 0666 | IPC_CREAT);
        char *str = shmat(shmid, (void *)0, 0);
        printf("\nActual Data read from memory: %s\n", str);

        for (int i = 0; str[i] != '\0'; i++)
        {
            if (str[i] >= 'a' && str[i] <= 'z')
            {
                str[i] = str[i] - 32;
            }
        }

        printf("Data to be displayed from memory: %s\n", str);
        shmdt(str);
        shmctl(shmid, IPC_RMID, NULL);
    }
}

```

Output:

```

jayannthan_hakr@jayannthan-Ubuntu:~/OS LAB/Assignment5$ ./1
Data to be written in memory:hi! how are you?

Actual Data read from memory: hi! how are you?

Data to be displayed from memory: HI! HOW ARE YOU?

```

2. Develop a client / server application for file transfer using shared memory.

Algorithm for server:

- 1) Created a unique key for a project using ftok() and stored in key

- 2) For the key, shared memory is allotted using `shmget` and returned id is stored in `shmid`
- 3) read input from the shared memory and store it in `str`
- 4) open file of name `str` and store file pointer in `fp`
- 5) read the file and write it into the shared memory using same `str`
- 6) close the file

Algorithm for client:

- 1) Created a unique key for a project using `ftok()` and stored in `key`
- 2) For the key, shared memory is allotted using `shmget` and returned id is stored in `shmid`
- 3) read input filename from the user and store it in the shared memory
- 4) After the server writes the file contents into the shared memory, display it and write it into a new file
- 5) Detach from the memory
- 6) Erase the memory

Code:

```
/*Server Code*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/shm.h>
#include <unistd.h>

int main()
{
    int shmid = shmget(1, 50, 666 | IPC_CREAT);
    char *str = (char *)shmat(shmid, (void *)0, 0);

    while (str[0] == '\0')
        ;

    printf("File name received\n");

    FILE *fp;
    fp = fopen(str, "r");
    if (fp == NULL)
    {
        strcpy(str, "File not found\n");
    }
    else
    {
        printf("Reading the file...\n");
        char c;
```

```

int i = 0;
while ((c = fgetc(fp)) != EOF)
{
    str[i] = c;
    i++;
}
str[i] = '\0';
printf("File content fetched successfully!\n");
fclose(fp);
}
}

```

```

/*Client Code*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/shm.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

int main()
{

int shmid = shmget(1, 50, 666 | IPC_CREAT);
char *str = (char *)shmat(shmid, (void *)0, 0);

printf("Enter file name : ");
scanf("%s", str);

printf("Waiting for file content from server... \n");
sleep(1);

if (strcmp(str, "File not found") == 0)
{
    printf("File not found\n");
}
else
{
    printf("Content received\n");
    printf("File Contents :\n%s", str);
}
shmdt(str);

shmctl(shmid, IPC_RMID, NULL);
}

```

Output:

```
Enter filename: main.c

File: main.c
File Request Sent...
Contents of File:
#include <stdio.h>

int main(int argc, char const *argv[])
{
    printf("Hello World!!!\n");
    return 0;
}
```

3. Develop an client/server chat application using shared memory.

Algorithm for server:

- 1) Get process id using getpid() and store it in pid
- 2) Create a shared memory using shmget and returned id is stored in shmid
- 3) Attach the pointer of message structure (memory) into the shared memory
- 4) Set pid2 in the memory as pid
- 5) Set status as -1
- 6) Call signal(). Send SIGUSR2, handler function as parameters
- 7) Loop until exit
 - a) If status is equal to 1 then wait for the other user to give input
 - b) Else then get input from the user to chat
 - c) Then set status as 1
 - d) Signal all the process using kill. Send pid1, SIGUSR1 as parameters
- 8) Detach from the memory
- 9) Destroy the memory

Algorithm for client:

- 1) Get process id using getpid() and store it in pid
- 2) Create a shared memory using shmget and returned id is stored in shmid
- 3) Attach the pointer of message structure (memory) into the shared memory
- 4) Set pid2 in the memory as pid
- 5) Set status as -1
- 6) Call signal(). Send SIGUSR1, handler function as parameters
- 7) Loop until exit
 - a) If status is equal to 1 then wait for the other user to give input
 - b) Else then get input from the user to chat
 - c) Then set status as 0
 - d) Signal all the process using kill. Send pid2, SIGUSR2 as parameters
- 8) Detach from the memory
- 9) Destroy the memory

Code:

```
/*Server Code*/
```

```

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

struct memory
{
    char buff[100];
    int status, pid1, pid2;
};

struct memory *shmptr;
void handler(int signum)
{
    if (signum == SIGUSR2)
    {
        printf("User1: ");
        puts(shmptr->buff);
    }
}

int main()
{
    int pid = getpid();
    int shmid = shmget(111, sizeof(struct memory), IPC_CREAT | 0666);
    shmptr = (struct memory *)shmat(shmid, NULL, 0);
    shmptr->pid2 = pid;
    shmptr->status = -1;
    signal(SIGUSR2, handler);
    while (1)
    {
        while (shmptr->status == 1)
            continue;
        sleep(1);
        printf("You: ");
        fgets(shmptr->buff, 100, stdin);
        shmptr->status = 1;
        kill(shmptr->pid1, SIGUSR1);
    }
    shmdt((void *)shmptr);
    shmctl(shmid, IPC_RMID, NULL);
    return 0;
}

```

```

/*Client Code*/

#include <signal.h>
#include <stdio.h>

```

```

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

struct memory
{
    char buff[100];
    int status, pid1, pid2;
};

struct memory *shmptr;
void handler(int signum)
{
    if (signum == SIGUSR1)
    {
        printf("User2: ");
        puts(shmptr->buff);
    }
}

int main()
{
    int pid = getpid();
    int shmid = shmget(111, sizeof(struct memory), IPC_CREAT | 0666);
    shmptr = (struct memory *)shmat(shmid, NULL, 0);
    shmptr->pid1 = pid;
    shmptr->status = -1;
    signal(SIGUSR1, handler);
    while (1)
    {
        while (shmptr->status != 1)
            continue;
        sleep(1);
        printf("You: ");
        fgets(shmptr->buff, 100, stdin);
        shmptr->status = 0;
        kill(shmptr->pid2, SIGUSR2);
    }
    shmdt((void *)shmptr);
    shmctl(shmid, IPC_RMID, NULL);
    return 0;
}

```

Output:

```
You: hi  
User1: hi  
  
You: hlo  
User1: hlo  
  
You: bye  
User1: bye
```

```
User2: hi  
  
You: hi  
User2: hlo  
  
You: hlo  
User2: bye  
  
You: bye
```

Learning Outcome:

- Executed shared memory functions and system calls
- Executed server-side and client-side program using shared memory

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UCS1411 - OPERATING SYSTEMS LAB

LAB EXERCISE 6

Implementation of Producer/Consumer Problem using Semaphores

Submission Date:21-04-2022

Name: Jayannthan P T

Dept: CSE 'A'

Roll No.: 205001049

1. To write a C program to create parent/child processes to implement the producer/consumer problem using semaphores in pthread library..

Algorithm:

- 1) For the segment, shared memory is allotted using shmget and returned id is stored in segid
- 2) For the empty, shared memory is allotted using shmget and returned id is stored in empty_id
- 3) For the full, shared memory is allotted using shmget and returned id is stored in full_id
- 4) For the mutex, shared memory is allotted using shmget and returned id is stored in mutex_id
- 5) Attach buffer to segid, empty to empty_id, full to full_id, mutex to mutex_id
- 6) Initialise semaphore to empty, full and mutex
- 7) Get the string from user and store it in str
- 8) Fork the process using call fork() and store it in m_pid
- 9) If m_pid greater than 0 then call producer function
- 10) Else call consumer function
- 11) Detach buffer, empty, full, mutex from memory
- 12) Destroy all shared memory
- 13) Destroy semaphores

Producer function:

- 1) Initialise i=0
- 2) If i greater than string length then exit from producer
- 3) Else
 - a) Empty semaphore acquired by producer
 - b) mutex semaphore acquired by producer
 - c) next character from string is written into buffer
 - d) Empty semaphore released by producer
 - e) mutex semaphore released by producer

Consumer function:

- 1) Initialise i=0
- 2) If i greater than string length then exit from consumer
- 3) Else
 - a) Empty semaphore acquired by consumer
 - b) mutex semaphore acquired by consumer
 - c) next character from buffer is read and pointer to buffer is increased
 - d) Empty semaphore released by consumer
 - e) mutex semaphore released by consumer

Code:

```
#include <stdio.h>
#include <semaphore.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <string.h>
#include <pthread.h>
#include <sys/ipc.h>
#define BUFSIZE 5

struct memory
{
    char buffer[BUFSIZE];
    int count;
    sem_t full;
    sem_t empty;
    sem_t mutex;
};

struct memory *shmptr;

char original[100], input_string[BUFSIZE];
int input_index = 0;
int c = 0;

void producer()
{
    do
    {
        if (shmptr->count >= BUFSIZE)
        {
            wait(NULL);
            continue;
        }

        sem_wait(&(shmptr->empty));
        sem_wait(&(shmptr->mutex));

        shmptr->buffer[shmptr->count++] = input_string[input_index++];
        shmptr->buffer[shmptr->count] = '\0';
        printf("Produced: %c\n", shmptr->buffer[shmptr->count - 1]);
    }
}
```

```

printf("Available items : ");
for (int i = 0; i < strlen(shmptr->buffer); i++)
{
    printf("%c ", shmptr->buffer[i]);
}
printf("\n");
sem_post(&(shmptr->mutex));
sem_post(&(shmptr->full));

sleep(1);
} while (input_index < strlen(original));
printf("\nAll items produced\n");
}

void consumer()
{

do
{

sem_wait(&(shmptr->full));
sem_wait(&(shmptr->mutex));

printf("Consumed %c\n", shmptr->buffer[0]);
memmove(shmptr->buffer, shmptr->buffer + 1, strlen(shmptr->buffer));
shmptr->count--;
c++;

printf("Available items : ");
for (int i = 0; i < strlen(shmptr->buffer); i++)
    printf("%c ", shmptr->buffer[i]);
printf("\n");
sem_post(&(shmptr->mutex));
sem_post(&(shmptr->empty));

sleep(3);
} while (c < strlen(input_string));

printf("Consumed all the items\n");
// exit(1);
}

int main()
{
    int shmid = shmget(IPC_PRIVATE, sizeof(struct memory), IPC_CREAT | 0666);
    shmptr = (struct memory *)shmat(shmid, NULL, 0);
    sem_init(&(shmptr->full), 1, 0);
    sem_init(&(shmptr->empty), 1, BUFSIZE);
    sem_init(&(shmptr->mutex), 1, 1);
    shmptr->count = 0;

    printf("Enter the string : ");
    scanf("%s", original);
}

```

```
int pid = fork();

if (pid == -1)
{
    printf("Fork error\n");
}
else if (pid == 0)
{
    consumer();
}
else
{
    producer();
}

shmctl(shmid, IPC_RMID, NULL);
sem_destroy(&(shmptr->empty));
sem_destroy(&(shmptr->full));
sem_destroy(&(shmptr->mutex));
return 0;
}
```

Output:

```

jayannthan_hakr@jayannthan-Ubuntu:~/OS LAB/Assignment6$ ./1
Enter the string : abcdefgh;
producer starts
Produced: a
Available items : a
consumer starts
Consumed a
Available items :
Produced: b
Available items : b
Produced: c
Available items : b c
Consumed b
Available items : c
Produced: d
Available items : c d
Produced: e
Available items : c d e
Produced: f
Available items : c d e f
Consumed c
Available items : d e f
Produced: g
Available items : d e f g
Produced: h
Available items : d e f g h

!!!buffer full!!!cannot produce!!!
Consumed d
Available items : e f g h
Produced: ;
Available items : e f g h ;

All items produced
jayannthan_hakr@jayannthan-Ubuntu:~/OS LAB/Assignment6$ Consumed e
Available items : f g h ;
Consumed f
Available items : g h ;
Consumed g
Available items : h ;
Consumed h
Available items : ;
Consumed ;
Available items :
Consumed all the items

```

2. Modify the program as separate client / server process programs to generate ‘N’ random numbers in producer and write them into shared memory. Consumer process should read them from shared memory and display them in terminal.

Algorithm for server:

- 1) For the segment, shared memory is allotted using `shmget` and returned id is stored in `segid`

- 2) For the empty, shared memory is allotted using shmget and returned id is stored in empty_id
- 3) For the full, shared memory is allotted using shmget and returned id is stored in full_id
- 4) For the mutex, shared memory is allotted using shmget and returned id is stored in mutex_id
- 5) Attach buffer to segid, empty to empty_id, full to full_id, mutex to mutex_id
- 6) For the key, shared memory is allotted using shmget and returned id is stored in shmid
- 7) Loop until N becomes zero
 - f) Empty semaphore acquired by Server
 - g) mutex semaphore acquired by Server
 - h) a new random number written into buffer and N is decremented
 - i) Empty semaphore released by Server
 - j) mutex semaphore released by Server

Algorithm for client:

- 1) For the segment, shared memory is allotted using shmget and returned id is stored in segid
- 2) For the empty, shared memory is allotted using shmget and returned id is stored in empty_id
- 3) For the full, shared memory is allotted using shmget and returned id is stored in full_id
- 4) For the mutex, shared memory is allotted using shmget and returned id is stored in mutex_id
- 5) Attach buffer to segid, empty to empty_id, full to full_id, mutex to mutex_id
- 6) For the key, shared memory is allotted using shmget and returned id is stored in shmid
- 7) Loop until N becomes zero
 - a) Empty semaphore acquired by Client
 - b) mutex semaphore acquired by Client
 - c) a new random read from buffer and N is decremented
 - d) Empty semaphore released by Client
 - e) mutex semaphore released by Client

Code:

```
/*Server Code*/
#include <stdio.h>
#include <semaphore.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <string.h>
#include <pthread.h>
#include <sys/ipc.h>
#include <time.h>
#define BUFSIZE 5
```

```

struct memory
{
    int buffer[BUFSIZE];
    int count;
    sem_t full;
    sem_t empty;
    sem_t mutex;
    int n;
    int nstatus;

};

struct memory * shmptr;

int main()
{
    srand(time(0));
    int shmid = shmget(111, sizeof(struct memory), IPC_CREAT | 0666);
    shmptr = (struct memory *) shmat(shmid, NULL, 0);
    shmptr->nstatus = 0;
    shmptr->count = 0;
    printf("\nServer\n");
    while (1)
    {
        if (shmptr->nstatus != 0)
        {
            int i = shmptr->n;
            do {    sem_wait(&(shmptr->empty));
                      sem_wait(&(shmptr->mutex));
                      shmptr->buffer[shmptr->count++] = rand() % 100;
                      printf("Newly Produced: %d\n", shmptr->buffer[shmptr->count - 1]);
                      i--;
                      sem_post(&(shmptr->mutex));
                      sem_post(&(shmptr->full));
                      sleep(0);
            } while (i > 0);
            printf("\nProduction done\n");
            if (i == 0) break;
        }
    }
    shmdt(shmptr);
    shmctl(shmid, IPC_RMID, NULL);
    sem_destroy(&(shmptr->empty));
    sem_destroy(&(shmptr->full));
    sem_destroy(&(shmptr->mutex));
    exit(1);
}

```

*/*Client Code*/*

```
#include <stdio.h>
#include <semaphore.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <string.h>
#include <pthread.h>
#include <sys/ipc.h>
#include <time.h>
#define BUFSIZE 5

struct memory
{
    int buffer[BUFSIZE];
    int count;
    sem_t full;
    sem_t empty;
    sem_t mutex;
    int n;
    int nstatus;

};

struct memory * shmptr;

int main()
{
    srand(time(0));
    int shmid = shmget(111, sizeof(struct memory), IPC_CREAT | 0666);
    shmptr = (struct memory *) shmat(shmid, NULL, 0);
    sem_init(&(shmptr->full), 1, 0);
    sem_init(&(shmptr->empty), 1, BUFSIZE);
    sem_init(&(shmptr->mutex), 1, 1);
    if (shmptr->nstatus == 0)
    {
        printf("Number of items to generate: ");
        scanf("%d", &(shmptr->n));
        shmptr->nstatus = 1;
    }
    int c = 0;
    do {
        sem_wait(&(shmptr->full));
        sem_wait(&(shmptr->mutex));
        printf("Available items : ");
        for (int i = 0; i < shmptr->count; i++)
        {
            printf("%d ", shmptr->buffer[i]);
        }
        printf("\n");
        printf("Consumes %d\n", shmptr->buffer[0]);
        memmove(shmptr->buffer, shmptr->buffer + 1, sizeof(shmptr->buffer)));
        shmptr->count--;
        shmptr->nstatus = 1;
    } while (c++ < shmptr->n);
}
```

```

shmptr->count--;
c++;
sem_post(&(shmptr->mutex));
sem_post(&(shmptr->empty));
sleep(4);
} while (c < shmptr->n);

printf("\nFinished consuming all items\n");
shmctl(shmid, IPC_RMID, NULL);
sem_destroy(&(shmptr->empty));
sem_destroy(&(shmptr->full));
sem_destroy(&(shmptr->mutex));

exit(1);

}

```

Output:

```

Server
Newly Produced: 41
Newly Produced: 16
Newly Produced: 18
Newly Produced: 8
Newly Produced: 27
Newly Produced: 14
Newly Produced: 0
!!!buffer full!!!
Newly Produced: 48
!!!buffer full!!!
!!!buffer full!!!
!!!buffer full!!!
Newly Produced: 49
!!!buffer full!!!
!!!buffer full!!!
!!!buffer full!!!
Newly Produced: 26

Production done

```

```

Number of items to generate: 10
Available items : 41
Consumes 41
Available items : 16 18 8
Consumes 16
Available items : 18 8 27 14 0
Consumes 18
Available items : 8 27 14 0 48
Consumes 8
Available items : 27 14 0 48 49
Consumes 27
Available items : 14 0 48 49 26
Consumes 14
Available items : 0 48 49 26
Consumes 0
Available items : 48 49 26
Consumes 48
Available items : 49 26
Consumes 49
Available items : 26
Consumes 26

Finished consuming all items

```

Learning Outcome:

- Executed semaphore functions and system calls
- Executed server-side and client-side program using shared memory and semaphores

LAB EXERCISE 6

Implementation of Producer/Consumer Problem using Semaphores

Submission Date:21-04-2022

Name: Jayannthan P T

Dept: CSE 'A'

Roll No.: 205001049

1. Assignment 1: Develop a C program to implement Banker's algorithm for deadlock avoidance with multiple instances of resource types

Algorithm:

- 1) Get choice from user
- 2) If choice is equal to 1, then ask for data input
- 3) If choice is equal to 2, then print the data
- 4) If choice is equal to 3, then call bankers algorithm to execute
- 5) If choice is equal to 4 then call for resource request
- 6) Else exit

Algorithm for data input:

- 1) Get no of process, no of resources, available instances, maximum required matrix, allotted instances from user
- 2) Calculate need matrix by subtracting allocation matrix from maximum required matrix

Algorithm for bankers algorithm:

- 1) Set ind to 0
- 2) For i from 0 to no of process times
 - a) If $f[i]$ is equal to 0
 - i. Set flag = 0
 - b) For j no of resources times
 - i. If $need[i,j]$ greater than $available[j]$ then set flag = 1
 - c) If flag equal to 0 then
 - i. Set $safeseq[ind] = i$
 - ii. Increment ind
 - iii. Set $available[j]$ equal to sum of $available[j]$ and $allocstion[i,j]$
 - iv. Set $f[i]$ is equal to 1
- 3) Set flag equal to 1
- 4) For i from 0 to no of process
 - a) If $f[i]$ is equal to 0 then print not a safe sequence and exit
 - b) If flag is equal to 1 then print safe sequence

Algorithm for Resource Request:

- 1) Get process no from user to allocate
- 2) Get resource vector
- 3) Check if allocation[i] is greater than available[i] then exit
- 4) Else set available equal to available[i] minus allocationvector[i] and allocation[ind][i] equal to sum of allocation[ind][i] and allocationvector[i]
- 5) Now call bankers algorithm

Code:

```
// Assignment 1: Develop a C program to implement Banker's algorithm for deadlock avoidance with multiple instances of resource types

#include <stdio.h>
#include <string.h>
#define max 100

typedef struct bankersdata
{
    int no_of_process;
    int no_of_resources;
    char process_name[max][5];
    char resources_name[max][5];
    int available_instance[max]; // available_instance[no_of_resources]
    int max_req[max][max]; // max_req[no_of_process][no_of_resources];
    int allocation[max][max]; // allocation[no_of_process][no_of_resources];
    int f[max], safesequence[max]; // f[no_of_process], safesequence[no_of_process]
    int need[max][max];
} bankersdata;

void printdata(bankersdata *bk)
{
    printf("Pid\tAlloc\tMax \tNeed \tAvail\t\n--\t");
    for (int i = 0; i < bk->no_of_resources; i++)
    {
        printf("%s ", bk->resources_name[i]);
    }
    printf("\t");
    for (int i = 0; i < bk->no_of_resources; i++)
    {
        printf("%s ", bk->resources_name[i]);
    }
    printf("\t");
    for (int i = 0; i < bk->no_of_resources; i++)
    {
        printf("%s ", bk->resources_name[i]);
    }
    printf("\t");
    for (int i = 0; i < bk->no_of_resources; i++)
    {
        printf("%s ", bk->resources_name[i]);
    }
    // printf("\n");
    printf("\t");
}
```

```

printf("\n-----\n");
for (int i = 0; i < bk->no_of_process; i++)
{
    printf("%s ", bk->process_name[i]);
    for (int j = 0; j < bk->no_of_resources; j++)
    {
        printf(" %d", bk->allocation[i][j]);
    }
    printf("\t");
    for (int j = 0; j < bk->no_of_resources; j++)
    {
        printf(" %d", bk->max_req[i][j]);
    }
    printf("\t");
    for (int j = 0; j < bk->no_of_resources; j++)
    {
        printf(" %d", bk->need[i][j]);
    }
    printf("\t");
    if (i == 0)
    {
        for (int j = 0; j < bk->no_of_resources; j++)
        {
            printf(" %d", bk->available_instance[j]);
        }
    }
    printf("\t\n");
}
}

void bankersalgo(bankersdata *bk)
{
    printdata(bk);

    int ind = 0;
    int y = 0;
    for (int k = 0; k < 5; k++)
    {
        for (int i = 0; i < bk->no_of_process; i++)
        {
            if (bk->f[i] == 0)
            {
                int flag = 0;
                for (int j = 0; j < bk->no_of_resources; j++)
                {
                    if (bk->need[i][j] > bk->available_instance[j])
                    {
                        flag = 1;
                        break;
                    }
                }
                if (flag == 0)
                {
                    bk->safesequence[ind] = i;
                    ind++;
                }
            }
        }
    }
}

```

```

        for (int j = 0; j < bk->no_of_resources; j++)
        {
            bk->available_instance[j] += bk->allocation[i][j];
        }
        bk->f[i] = 1;
    }
}
}

int flag = 1;
for (int i = 0; i < bk->no_of_process; i++)
{
    if (bk->f[i] == 0)
    {
        flag = 0;
        printf("NOT A SAFE SYSTEM");
        break;
    }
}
if (flag == 1)
{
    printf("SAFE SEQUENCE\n");
    for (int i = 0; i < bk->no_of_process - 1; i++)
    {
        printf(" %s ->, bk->process_name[bk->safesequence[i]]));
    }
    printf(" %s", bk->process_name[bk->safesequence[bk->no_of_process - 1]]);
}
printdata(bk);
}

bankersdata getdata()
{
    bankersdata bk;
    printf("\nEnter no of process:");
    scanf("%d", &bk.no_of_process);
    printf("\nEnter process ids:\n");
    for (int i = 0; i < bk.no_of_process; i++)
    {
        printf("process name of process %d:", i + 1);
        scanf(" %s", &bk.process_name[i]);
    }

    printf("\nEnter no of resources:");
    scanf("%d", &bk.no_of_resources);
    printf("\nEnter resource ids:\n");
    for (int i = 0; i < bk.no_of_resources; i++)
    {
        printf("resource name of resource %d:", i + 1);
        scanf(" %s", &bk.resources_name[i]);
    }

    printf("\nEnter available instances:\n");
}

```

```

for (int i = 0; i < bk.no_of_resources; i++)
{
    printf("available instances of resource %s:", bk.resources_name[i]);
    scanf(" %d", &bk.available_instance[i]);
}

printf("\nEnter Maximum requirement:\n");
for (int i = 0; i < bk.no_of_process; i++)
{
    printf("Maximum requirement for process %s:", bk.process_name[i]);
    for (int j = 0; j < bk.no_of_resources; j++)
    {
        scanf(" %d", &bk.max_req[i][j]);
    }
}

printf("\nEnter Allocated instances:\n");
for (int i = 0; i < bk.no_of_process; i++)
{
    printf("Allocated instances for process %s:", bk.process_name[i]);
    for (int j = 0; j < bk.no_of_resources; j++)
    {
        scanf(" %d", &bk.allocation[i][j]);
    }
}

for (int i = 0; i < bk.no_of_process; i++)
{
    bk.f[i] = 0;
    for (int j = 0; j < bk.no_of_resources; j++)
    {
        bk.need[i][j] = bk.max_req[i][j] - bk.allocation[i][j];
    }
}
return bk;
}

int main(int argc, char const *argv[])
{
    bankersdata bk = getdata();
    int choice = 0;
    printf("\nMenu:\n\t1.Enter new data\n\t2.PrintData\n\t3.Bankers State\n\t4.Resource Request\n\t5.Exit\nEnter Choice:");
    scanf(" %d", &choice);
    while (choice)
    {
        switch (choice)
        {
            case 1:
            {
                bk = getdata();
                break;
            }
            case 2:
        }
    }
}

```

```

    {
        printdata(&bk);
        break;
    }

    case 3:
    {
        bankersalgo(&bk);
        break;
    }

    case 4:
    {
        char temp_process_name[5];
        printf("\nEnter process id for request:");
        scanf(" %s", &temp_process_name);
        int index_of_process = -1;
        for (int i = 0; i < bk.no_of_process; i++)
        {
            if (strcmp(temp_process_name, bk.process_name[i]) == 0)
            {
                index_of_process = i;
                break;
            }
        }
        if (index_of_process == -1)
        {
            printf("\nprocess name not correct!!!\n");
            break;
        }
        else
        {
            int allocation_vector[max];
            printf("\nEnter the request vector for %s:", bk.process_name[index_of_process]);
            for (int i = 0; i < bk.no_of_resources; i++)
            {
                scanf(" %d", &allocation_vector[i]);
            }
            int flag = 1;
            for (int i = 0; i < bk.no_of_resources; i++)
            {
                if (allocation_vector[i] > bk.available_instance[i])
                {
                    flag = 0;
                    break;
                }
            }
            if (flag == 0)
            {
                printf("\n!!!Resource cannot be allocated!!!");
                break;
            }
            for (int i = 0; i < bk.no_of_resources; i++)

```

```

    {
        bk.available_instance[i] -= allocation_vector[i];
        bk.allocation[index_of_process][i] += allocation_vector[i];
    }
    bankersalgo(&bk);
}
break;
}

case 5:
return 0;

default:
{
    printf("\n!!!Enter correct choice!!!\n");
    break;
}
}
printf("\nMenu:\n\t1.Enter new data\n\t2.PrintData\n\t3.Bankers
State\n\t4.Resource Request\n\t5.Exit\nEnter Choice:");
scanf(" %d", &choice);
}
return 0;
}

```

Output:

```

Menu:
1.Enter new data
2.PrintData
3.Bankers State
4.Resource Request
5.Exit
Enter Choice:2
Pid      Alloc   Max     Need    Avail
--      A B C   A B C   A B C   A B C
-----
P0  0 1 0      7 5 3   7 4 3   3 3 2
P1  2 0 0      3 2 2   1 2 2
P2  3 0 2      9 0 2   6 0 0
P3  2 1 1      2 2 2   0 1 1
P4  0 0 2      4 3 3   4 3 1

```

```

Menu:
1.Enter new data
2.PrintData
3.Bankers State
4.Resource Request
5.Exit
Enter Choice:4
Enter process id for request:P1
Enter the request vector for P1:1
0
2
Pid      Alloc   Max     Need    Avail
--      A B C   A B C   A B C   A B C
-----
P0  0 1 0      7 5 3   7 4 3   2 3 0
P1  3 0 2      3 2 2   0 2 0
P2  3 0 2      9 0 2   6 0 0
P3  2 1 1      2 2 2   0 1 1
P4  0 0 2      4 3 3   4 3 1
SAFE SEQUENCE
P1 -> P3 -> P4 -> P0 -> P2

```

2. Assignment 2: Develop a C program to implement algorithm for deadlock detection with multiple instances of resource types and display the processes involved in deadlock

Algorithms:

- 1) Get no of process, no of resources, available instances, maximum required matrix, allotted instances from user
- 2) Calculate need matrix by subtracting allocation matrix from maximum required matrix
- 3) Set ind to 0
- 4) For i from 0 to no of process times
 - c) If $f[i]$ is equal to 0
 - i. Set flag = 0
 - d) For j no of resources times
 - i. If $need[i,j]$ greater than $available[j]$ then set flag = 1
 - e) If flag equal to 0 then
 - i. Set safeseq[ind]=i
 - ii. Increment ind
 - iii. Set available [j] equal to sum of available[j] and allocstion[i,j]
 - iv. Set $f[i]$ is equal to 1
- 5) Set flag equal to 1
- 6) For i from 0 to no of process
 - f) If $f[i]$ is equal to 0 then print not a safe sequence and exit
 - g) If flag is equal to 1 then print safe sequence

Code:

```
#include <stdio.h>
//#include <conio.h>
#include <string.h>
#define max 100

typedef struct bankersdata
{
    int no_of_process;
    int no_of_resources;
    char process_name[max][5];
    char resources_name[max][5];
    int available_instance[max]; // available_instance[no_of_resources]
    int max_req[max][max]; // max_req[no_of_process][no_of_resources];
    int allocation[max][max]; // allocation[no_of_process][no_of_resources];
    int f[max], safesequence[max]; // f[no_of_process], safesequence[no_of_process]
    int need[max][max];
} bankersdata;

void printdata(bankersdata *bk)
{
    printf("Pid\tAlloc\tMax \tNeed \tAvail\t\n--\t");
    for (int i = 0; i < bk->no_of_resources; i++)
    {
        printf("%s ", bk->resources_name[i]);
    }
    printf("\t");
    for (int i = 0; i < bk->no_of_resources; i++)
    {
        printf("%s ", bk->resources_name[i]);
    }
}
```

```

printf("\t");
for (int i = 0; i < bk->no_of_resources; i++)
{
    printf("%s ", bk->resources_name[i]);
}
printf("\t");
for (int i = 0; i < bk->no_of_resources; i++)
{
    printf("%s ", bk->resources_name[i]);
}
printf("\t");
printf("\n-----\n");
for (int i = 0; i < bk->no_of_process; i++)
{
    printf("%s ", bk->process_name[i]);
    for (int j = 0; j < bk->no_of_resources; j++)
    {
        printf(" %d", bk->allocation[i][j]);
    }
    printf("\t");
    for (int j = 0; j < bk->no_of_resources; j++)
    {
        printf(" %d", bk->max_req[i][j]);
    }
    printf("\t");
    for (int j = 0; j < bk->no_of_resources; j++)
    {
        printf(" %d", bk->need[i][j]);
    }
    printf("\t");
    if (i == 0)
    {
        for (int j = 0; j < bk->no_of_resources; j++)
        {
            printf(" %d", bk->available_instance[j]);
        }
    }
    printf("\t\n");
}
}
void bankersalgo(bankersdata *bk)
{
    int ind = 0;
    int y = 0;
    for (int k = 0; k < 5; k++)
    {
        for (int i = 0; i < bk->no_of_process; i++)
        {
            if (bk->f[i] == 0)
            {
                int flag = 0;
                for (int j = 0; j < bk->no_of_resources; j++)
                {
                    if (bk->need[i][j] > bk->available_instance[j])

```

```

        {
            flag = 1;
            break;
        }
    }
    if (flag == 0)
    {
        bk->safesequence[ind] = i;
        ind++;
        for (int j = 0; j < bk->no_of_resources; j++)
        {
            bk->available_instance[j] += bk->allocation[i][j];
        }
        bk->f[i] = 1;
    }
}
}

int flag = 1;
for (int i = 0; i < bk->no_of_process; i++)
{
    if (bk->f[i] == 0)
    {
        flag = 0;
        printf("\n\n!!!!NOT A SAFE SYSTEM!!!!\nDue to the following processes:");
        break;
    }
}
if (flag == 1)
{
    printf("SAFE SEQUENCE\n");
    for (int i = 0; i < bk->no_of_process - 1; i++)
    {
        printf(" %s ->, bk->process_name[bk->safesequence[i]]));
    }
    printf(" %s", bk->process_name[bk->safesequence[bk->no_of_process - 1]]);
}
else
{
    for (int i = 0; i < bk->no_of_process; i++)
    {
        if (bk->f[i] == 0)
        {
            printf(" %s ", bk->process_name[i]);
        }
    }
}
}

bankersdata getdata()
{
    bankersdata bk;
    printf("\nEnter no of process:");
}

```

```

scanf("%d", &bk.no_of_process);
printf("\nEnter process ids:\n");
for (int i = 0; i < bk.no_of_process; i++)
{
    printf("process name of process %d:", i + 1);
    scanf(" %s", &bk.process_name[i]);
}

printf("\nEnter no of resources:");
scanf("%d", &bk.no_of_resources);
printf("\nEnter resource ids:\n");
for (int i = 0; i < bk.no_of_resources; i++)
{
    printf("resource name of resource %d:", i + 1);
    scanf(" %s", &bk.resources_name[i]);
}

printf("\nEnter available instances:\n");
for (int i = 0; i < bk.no_of_resources; i++)
{
    printf("available instances of resource %s:", bk.resources_name[i]);
    scanf(" %d", &bk.available_instance[i]);
}

printf("\nEnter Maximum requirement:\n");
for (int i = 0; i < bk.no_of_process; i++)
{
    printf("Maximum requirement for process %s:", bk.process_name[i]);
    for (int j = 0; j < bk.no_of_resources; j++)
    {
        scanf(" %d", &bk.max_req[i][j]);
    }
}

printf("\nEnter Allocated instances:\n");
for (int i = 0; i < bk.no_of_process; i++)
{
    printf("Allocated instances for process %s:", bk.process_name[i]);
    for (int j = 0; j < bk.no_of_resources; j++)
    {
        scanf(" %d", &bk.allocation[i][j]);
    }
}

for (int i = 0; i < bk.no_of_process; i++)
{
    bk.f[i] = 0;
    for (int j = 0; j < bk.no_of_resources; j++)
    {
        bk.need[i][j] = bk.max_req[i][j] - bk.allocation[i][j];
    }
}
return bk;
}

```

```

int main(int argc, char const *argv[])
{
    bankersdata bk = getdata();
    int choice = 0;
    printf("\nMenu:\n\t1.Enter new data\n\t2.PrintData\n\t3.Bankers State\n\t4.Exit\nEnter Choice:");
    scanf(" %d", &choice);
    while (choice)
    {
        switch (choice)
        {
            case 1:
            {
                bk = getdata();
                break;
            }
            case 2:
            {
                printdata(&bk);
                break;
            }

            case 3:
            {
                bankersalgo(&bk);
                break;
            }

            case 4:
            {
                return 0;
                break;
            }

            default:
            {
                printf("\n!!!Enter correct choice!!!\n");
                break;
            }
        }
        printf("\nMenu:\n\t1.Enter new data\n\t2.PrintData\n\t3.Bankers State\n\t4.Exit\nEnter Choice:");
        scanf(" %d", &choice);
    }
    return 0;
}

```

Output:

```

Enter Choice:3
SAFE SEQUENCE
P1 -> P3 -> P4 -> P0 -> P2

```

Learning Outcome:

- Bankers algorithm implementation
- Importance of deadlock prevention
- Printing safe Sequence

LAB EXERCISE 8

Implementation of Memory Management Algorithms

Submission Date:23-05-2022

Name: Jayannthan P T

Dept: CSE 'A'

Roll No.: 205001049

1. Free space is maintained as a linked list of nodes with each node having the starting byte address and the ending byte address of a free block. Each memory request consists of the process-id and the amount of storage space required in bytes. Allocated memory space is again maintained as a linked list of nodes with each node having the process id, starting byte address and the ending byte address of the allocated space. When a process finishes (taken as input), the appropriate node from the allocated list should be deleted and this free disk space should be added to the free space list. [Care should be taken to merge contiguous free blocks into one single block. This results in deleting more than one node from the free space list and changing the start and end address in the appropriate node]. For allocation use first fit, worst fit and best fit algorithms..

Algorithm:

1. We create a header file which defines the structure with components as start, end, size, status, id, next pointer to the node.
2. It also has a functions insertlast(), insertmiddle(), create new node() and sorted merging()
3. We create an instance of the structure in main function
4. Read the number of partitions of the memory from the user
5. Read the starting and ending points of each partitions
6. For each entry create new node and insertlast
7. Display the status after memory partitioning
8. Each hole is allotted a unique id by itself and is not changed until the program is ended
9. Inside the do while loop:
 - Ask the user to choose the algorithm
 - Have a 2D array where we store the choice of algorithm
 - Inside it we have another do while loop where we ask for entry/allocate, exit/deallocate, display, coalesce and exit
 - ALLOCATION:

Ask the user for the size of process

- i. First fit:

- a) Check which is the first hole that satisfies the given process's size and return the position of that node
- ii. Best fit:
 - a) Assumes min as a constant
 - b) Iterates through the list to find the min difference between process size and node size
 - c) If min value has changed the node pointed by the function is allocated for the process
- iii. Worst fit:
 - a) Assumes 0 as max value
 - b) Iterates through the list to find the max difference
 - c) If max value is changed allocate the process to the node pointed by the function

`allocate()`

If `ptrsize` is equal to the node size we allocate the node as it is

If `ptrsize` is greater than the process size we insert 2 node one with size equal to process size and other with size as difference between the pointed node

- DEALLOCATION:
 - i. We search the pid in the list and find the node in which the process is allocated.
 - ii. Change the status of the node to 'H' and delete the process
 - iii. It also the combine function which combines the holes with same id to bring back the initial state
- DISPLAY:
 - i. Displays the list with free spaces and the other list with accommodation of processes
- COALESING:
 - i. It combines all the adjacent partitions with status 'H' irrespective of the hole id

Code:

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

struct node
{
    int start;
    int end;
    int size;
    char status[3];
    struct node *next;
};

struct node *newNode(int start, int end);
int insert(struct node *temp, struct node **head);
struct node *insertEnd(struct node *, struct node *);
struct node *clone(struct node *list);
struct node *merge(struct node *h1, struct node *h2);

struct node *newNode(int start, int end)
```

```

{
    struct node *temp = (struct node *)malloc(sizeof(struct node));
    strcpy(temp->status, "H");
    temp->end = end;
    temp->start = start;
    temp->size = temp->end - temp->start;
    temp->next = NULL;
    return temp;
}

struct node *insertEnd(struct node *p, struct node *temp)
{
    struct node *ptr = p;
    if (!ptr)
    {
        p = temp;
        p->next = NULL;
    }
    else
    {
        while (ptr->next)
            ptr = ptr->next;
        ptr->next = temp;
        temp->next = NULL;
    }
    return p;
}

int insert(struct node *temp, struct node **head)
{
    if (!temp)
        return 0;
    struct node *ptr = *head;
    if (!*head)
    {
        *head = insertEnd(*head, temp);
        return 1;
    }
    if (temp->start < ptr->start)
    {
        temp->next = *head;
        *head = temp;
        return 1;
    }
    while (ptr->next && temp->start > ptr->next->start)
        ptr = ptr->next;
    temp->next = ptr->next;
    ptr->next = temp;
    return 1;
}

struct node *clone(struct node *list)
{
    if (!list)

```

```

    return NULL;

struct node *result = (struct node *)malloc(sizeof(struct node));
result->start = list->start;
result->end = list->end;
result->size = list->size;
strcpy(result->status, list->status);
result->next = clone(list->next);
return result;
}

struct node *merge(struct node *h1, struct node *h2)
{
    if (!h1)
        return h2;
    if (!h2)
        return h1;
    if (h1->start < h2->start)
    {
        h1->next = merge(h1->next, h2);
        return h1;
    }
    else
    {
        h2->next = merge(h1, h2->next);
        return h2;
    }
}

void combine(struct node **p)
{
    struct node *ptr = *p, *temp;
    while (ptr)
    {
        temp = ptr;
        while (temp->next && temp->end == temp->next->start)
            temp = temp->next;
        ptr->next = temp->next;
        ptr->end = temp->end;
        ptr = ptr->next;
    }
}

struct node *deallocate(struct node **p, char *pid)
{
    struct node *ptr = *p, *prev;
    if (ptr && strcmp(ptr->status, pid) == 0)
    {
        *p = ptr->next;
        strcpy(ptr->status, "H");
        return ptr;
    }
    while (ptr && strcmp(ptr->status, pid))
    {

```

```

        prev = ptr;
        ptr = ptr->next;
    }
    if (!ptr)
        return NULL;
    strcpy(ptr->status, "H");
    prev->next = ptr->next;
    return ptr;
}

void table(struct node *p, char str[])
{
    struct node *ptr = p;
    if (!ptr)
    {
        printf("NULL\n\n");
        return;
    }
    for (int i = 0; i < strlen(str); i++)
        printf("%c", str[i] == '|' ? '+' : str[i] == '-' ? ' '
                                         : '-'));
    printf("\n%s\n", str);

    for (int i = 0; i < strlen(str); i++)
        printf("%c", str[i] == '|' ? '+' : str[i] == '-' ? ' '
                                         : '-'));
    printf("\n");

    int end, s;
    end = ptr->end;
    s = strlen(ptr->status);

    while (ptr)
    {
        if (!ptr->next || ptr->end == ptr->next->start)
            printf("%-*d", 9 + strlen(ptr->status), ptr->start);
        else
            printf("%-*d%-*d", 9 + strlen(ptr->status), ptr->start, 9, ptr->end);
        end = ptr->end;
        s = strlen(ptr->status);
        ptr = ptr->next;
    }
    printf("%d", end);
    printf("\n\n\n");
}

void disp(struct node *p)
{
    char buf[100], mem[1000];
    struct node *ptr = p;
    strcpy(mem, "|");
    while (ptr)
    {

```

```

        if (!ptr->next || ptr->end == ptr->next->start)
            sprintf(buf, "    %s    |", ptr->status);
        else if (ptr->end != ptr->next->start)
            sprintf(buf, "    %s    |-----|", ptr->status);
        strcat(mem, buf);
        ptr = ptr->next;
    }
    table(p, mem);
}

void display(struct node *p, struct node *q)
{
    printf("\nAllocated Memory Space\n\n");
    disp(p);
    printf("Free Memory Space\n\n");
    disp(q);
    struct node *r = merge(clone(p), clone(q));
    printf("Physical Memory Space\n\n");
    disp(r);
    free(r);
}

int first(struct node *f, int size)
{
    struct node *ptr = f;
    while (ptr && !(ptr->size >= size))
    {
        ptr = ptr->next;
    }
    if (!ptr)
        return -1;
    return ptr->size;
}

int best(struct node *f, int size)
{
    struct node *ptr = f;
    int min = INT_MAX;
    while (ptr)
    {
        if (ptr->size - size > 0 && min > ptr->size - size)
            min = ptr->size - size;
        ptr = ptr->next;
    }
    if (min == INT_MAX)
        return -1;
    return min + size;
}

int worst(struct node *f, int size)
{
    struct node *ptr = f;
    int max = INT_MIN;
    while (ptr)

```

```

{
    if (ptr->size - size > 0 && max < ptr->size - size)
        max = ptr->size - size;
    ptr = ptr->next;
}
if (max == INT_MIN)
    return -1;
return max + size;
}

int whichfit(struct node *f, int size, int ch)
{
    if (ch == 1)
        return first(f, size);
    if (ch == 2)
        return best(f, size);
    if (ch == 3)
        return worst(f, size);
}

struct node *allocate(struct node **f, char *pid, int size, int ptrsize)
{
    if (ptrsize - size < 0)
        return NULL;
    struct node *ptr = *f, *prev;

    if (ptr->size == ptrsize)
    {
        if (ptr && ptr->size >= size)
        {
            if (ptr->size == size)
            {
                *f = ptr->next;

                strcpy(ptr->status, pid);
                return ptr;
            }
            else
            {
                struct node *temp1 = newNode(ptr->start, ptr->start + size);
                struct node *temp2 = newNode(ptr->start + size, ptr->end);
                *f = temp2;
                temp2->next = ptr->next;
                strcpy(temp1->status, pid);
                free(ptr);
                return temp1;
            }
        }
    }

    while (ptr && !(ptr->size == ptrsize))
    {
        prev = ptr;
        ptr = ptr->next;
    }
}

```

```

    }
    if (!ptr)
        return NULL;
    if (ptr->size == size)
    {
        prev->next = ptr->next;
        strcpy(ptr->status, pid);
        return ptr;
    }
    else
    {
        struct node *temp1 = newNode(ptr->start, ptr->start + size);
        struct node *temp2 = newNode(ptr->start + size, ptr->end);
        prev->next = temp2;
        temp2->next = ptr->next;
        strcpy(temp1->status, pid);
        free(ptr);
        return temp1;
    }
}

int main()
{
    int ch, n, start, end;
    printf("\nEnter the Memory Representation:");
    printf("\nEnter the no.of partitions in memory: ");
    scanf("%d", &n);
    struct node *mempool = NULL, *alloc = NULL, *temp = NULL;
    for (int i = 0; i < n; i++)
    {
        printf("Enter Starting and ending address of partition %d: ", i + 1);
        scanf("%d%d", &start, &end);
        if (start >= end || i && temp->end != start)
        {
            i--;
            printf("Invalid entry,enter again\n");
        }
        else
        {
            temp = newNode(start, end);
            mempool = insertEnd(mempool, temp);
        }
    }

    display(alloc, mempool);

    do
    {
        printf("\n1. First Fit\n2. Best Fit \n3. Worst Fit \n4. Exit \nEnter your choice:");
    };
    scanf("%d", &ch);
    switch (ch)
    {
        case 1:

```

```

case 2:
case 3:
    break;
case 4:
    printf("Exiting...\n");
    return 0;
    break;
default:
    printf("\nInvalid Input!\n");
}
int ch1, size;
char pid[3];
char fits[3][15] = {"First Fit", "Best Fit", "Worst Fit"};
do
{
    printf("\n\t\t%s Memory Allocation Algorithm\n\n1. Entry / Allocate\n2. Exit / Deallocate\n3. Display\n4. Coalescing of Holes\n5. Back to Algorithm\n6. Exit\nEnter your choice: ", fits[ch - 1]);
    scanf("%d", &ch1);
    switch (ch1)
    {
        case 1:
            printf("\nEnter process id : ");
            scanf("%s", pid);
            printf("Enter size needed : ");
            scanf("%d", &size);
            if (size <= 0)
            {
                printf("\nInvalid size!\n");
                break;
            }
            if (!insert(allocate(&mempool, pid, size, whichfit(mempool, size, ch)),
&alloc))
            {
                printf("\nCouldn't allocate memory to %s!\n", pid);
                break;
            }
            else
                printf("\nMemory is allocted to %s\n", pid);
            display(alloc, mempool);
            break;
        case 2:
            printf("\nEnter process id : ");
            scanf("%s", pid);
            if (!insert(deallocate(&alloc, pid), &mempool))
            {
                printf("\nProcess %s is not there!\n", pid);
                break;
            }
            else
                printf("\n%s's memory is deallocted\n", pid);
            display(alloc, mempool);
            break;
        case 3:
    }
}

```

```
    display(alloc, mempool);
    break;
case 4:
    combine(&mempool);
    display(alloc, mempool);
    break;
case 5:
    break;
case 6:
    printf("Exiting...\n");
    return 0;
    break;
default:
    printf("\nInvalid Input!\n");
}
} while (!(ch1 == 5 || ch1 == 6));
} while (ch != 4);
return 0;
}
```

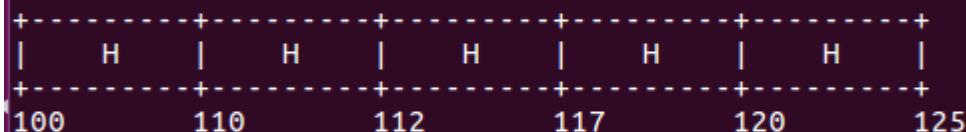
Output:

```
Enter the Memory Representation:  
Enter the no.of partitions in memory: 5  
Enter Starting and ending address of partition 1: 100 110  
Enter Starting and ending address of partition 2: 110 112  
Enter Starting and ending address of partition 3: 112 117  
Enter Starting and ending address of partition 4: 117 120  
Enter Starting and ending address of partition 5: 120  
125
```

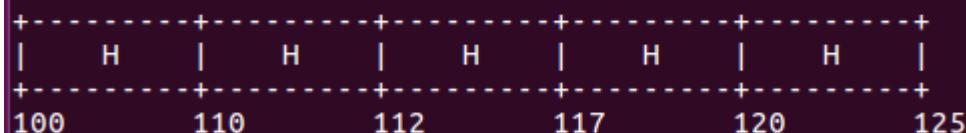
Allocated Memory Space

NULL

Free Memory Space



Physical Memory Space



1. First Fit
2. Best Fit
3. Worst Fit
4. Exit

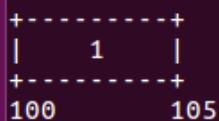
First Fit Memory Allocation Algorithm

1. Entry / Allocate
 2. Exit / Deallocate
 3. Display
 4. Coalescing of Holes
 5. Back to Algorithm
 6. Exit
- Enter your choice: 1

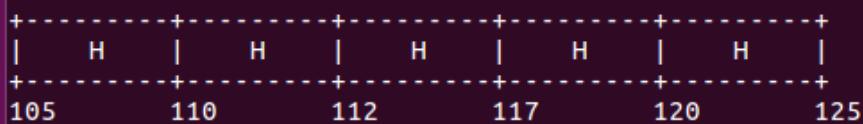
Enter process id : 1
Enter size needed : 5

Memory is allocated to 1

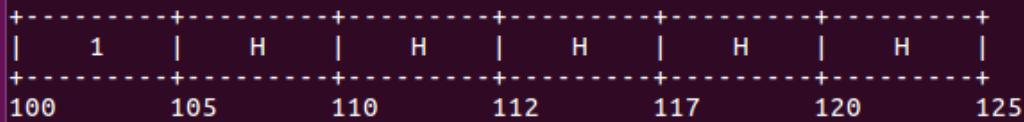
Allocated Memory Space



Free Memory Space



Physical Memory Space



First Fit Memory Allocation Algorithm

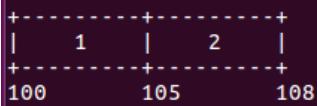
1. Entry / Allocate
2. Exit / Deallocate
3. Display
4. Coalescing of Holes
5. Back to Algorithm
6. Exit

Enter your choice: 1

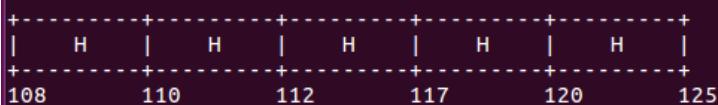
Enter process id : 2
Enter size needed : 3

Memory is allocated to 2

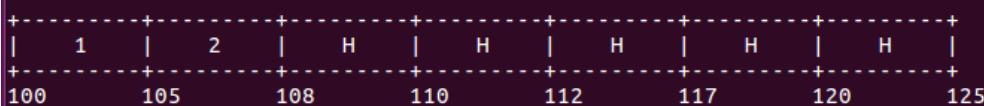
Allocated Memory Space



Free Memory Space



Physical Memory Space



First Fit Memory Allocation Algorithm

1. Entry / Allocate
2. Exit / Deallocate
3. Display
4. Coalescing of Holes
5. Back to Algorithm
6. Exit

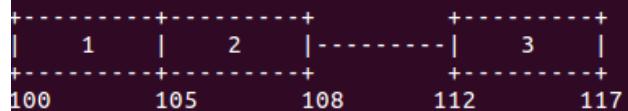
Enter your choice: 1

Enter process id : 3

Enter size needed : 5

Memory is allotted to 3

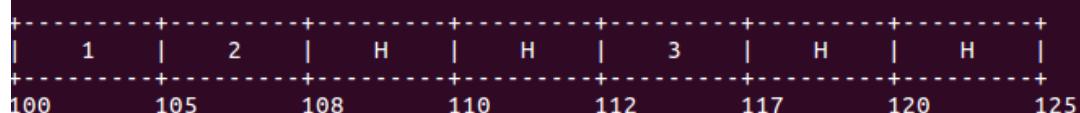
Allocated Memory Space



Free Memory Space



Physical Memory Space



First Fit Memory Allocation Algorithm

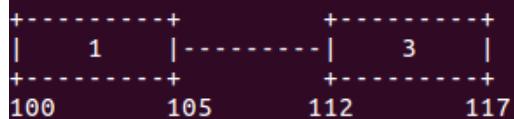
1. Entry / Allocate
2. Exit / Deallocate
3. Display
4. Coalescing of Holes
5. Back to Algorithm
6. Exit

Enter your choice: 2

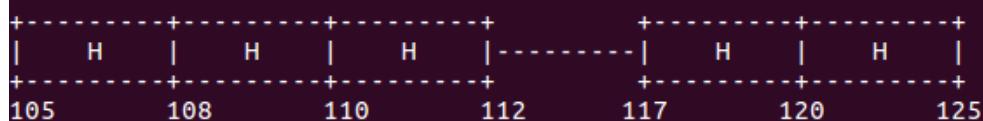
Enter process id : 2

2's memory is deallocated

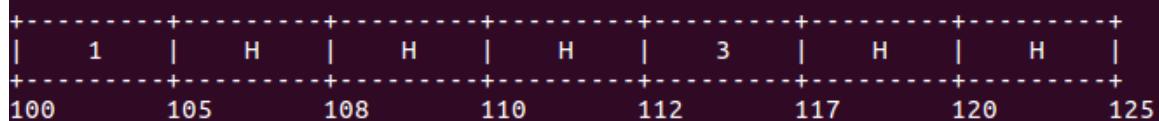
Allocated Memory Space



Free Memory Space



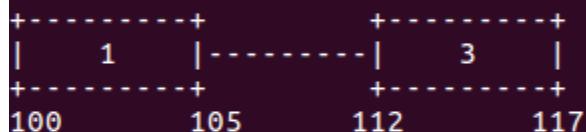
Physical Memory Space



First Fit Memory Allocation Algorithm

1. Entry / Allocate
 2. Exit / Deallocate
 3. Display
 4. Coalescing of Holes
 5. Back to Algorithm
 6. Exit
- Enter your choice: 4

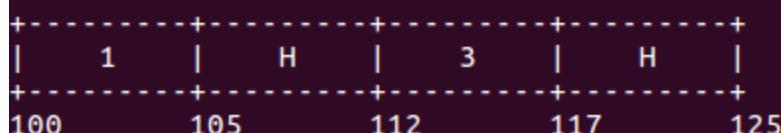
Allocated Memory Space



Free Memory Space



Physical Memory Space



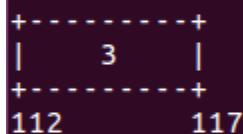
First Fit Memory Allocation Algorithm

1. Entry / Allocate
 2. Exit / Deallocate
 3. Display
 4. Coalescing of Holes
 5. Back to Algorithm
 6. Exit
- Enter your choice: 2

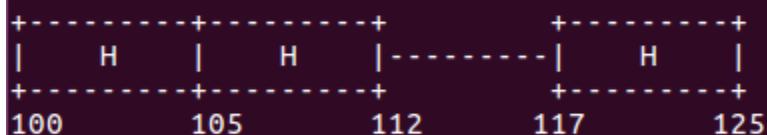
Enter process id : 1

1's memory is deallocated

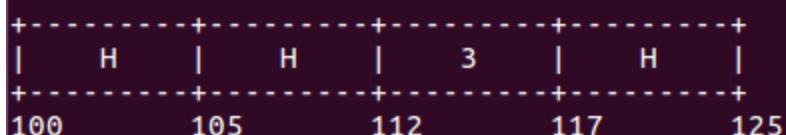
Allocated Memory Space



Free Memory Space



Physical Memory Space



First Fit Memory Allocation Algorithm

1. Entry / Allocate
2. Exit / Deallocate
3. Display
4. Coalescing of Holes
5. Back to Algorithm
6. Exit

Enter your choice: 2

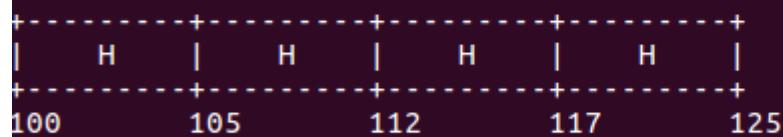
Enter process id : 3

3's memory is deallocated

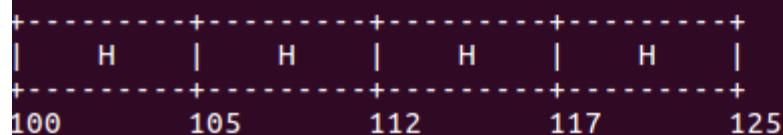
Allocated Memory Space

NULL

Free Memory Space



Physical Memory Space



Best Fit Memory Allocation Algorithm

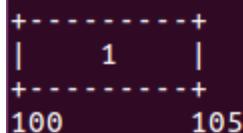
1. Entry / Allocate
 2. Exit / Deallocate
 3. Display
 4. Coalescing of Holes
 5. Back to Algorithm
 6. Exit
- Enter your choice: 1

Enter process id : 1

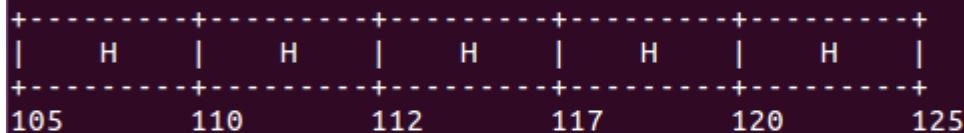
Enter size needed : 5

Memory is allocated to 1

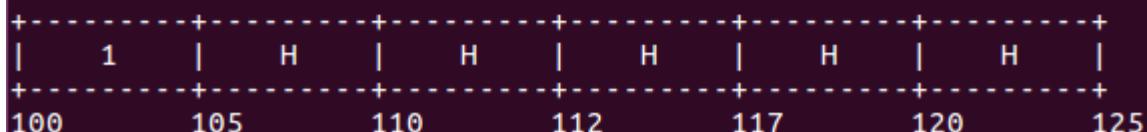
Allocated Memory Space



Free Memory Space



Physical Memory Space



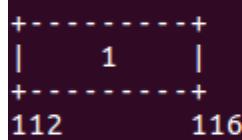
Best Fit Memory Allocation Algorithm

1. Entry / Allocate
 2. Exit / Deallocate
 3. Display
 4. Coalescing of Holes
 5. Back to Algorithm
 6. Exit
- Enter your choice: 1

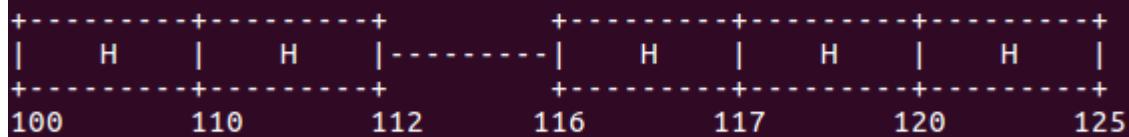
Enter process id : 1
Enter size needed : 4

Memory is allocated to 1

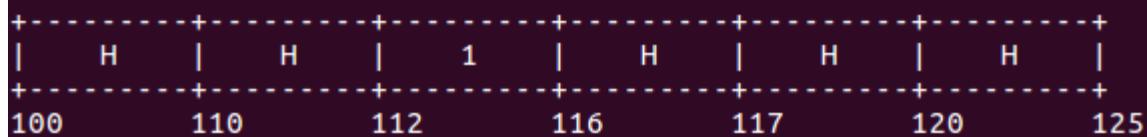
Allocated Memory Space



Free Memory Space



Physical Memory Space



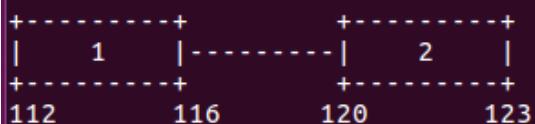
Best Fit Memory Allocation Algorithm

1. Entry / Allocate
 2. Exit / Deallocate
 3. Display
 4. Coalescing of Holes
 5. Back to Algorithm
 6. Exit
- Enter your choice: 1

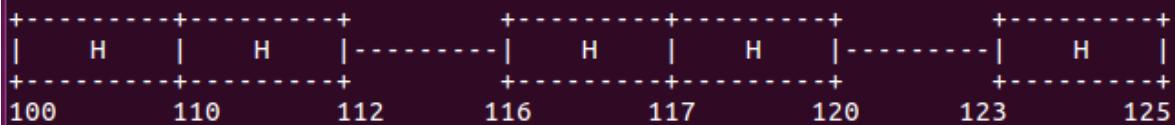
Enter process id : 2
Enter size needed : 3

Memory is allocted to 2

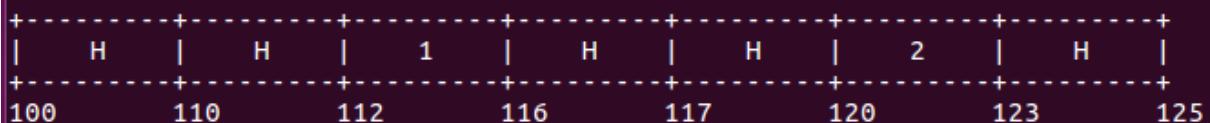
Allocated Memory Space



Free Memory Space



Physical Memory Space



Best Fit Memory Allocation Algorithm

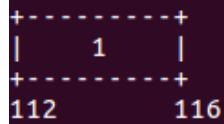
1. Entry / Allocate
2. Exit / Deallocate
3. Display
4. Coalescing of Holes
5. Back to Algorithm
6. Exit

Enter your choice: 2

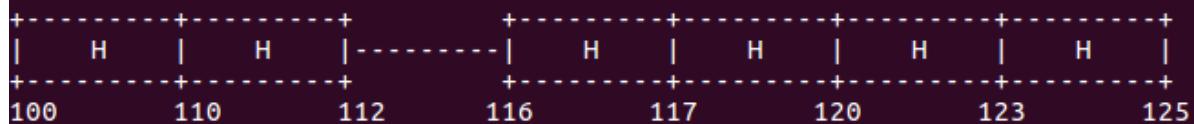
Enter process id : 2

2's memory is deallocated

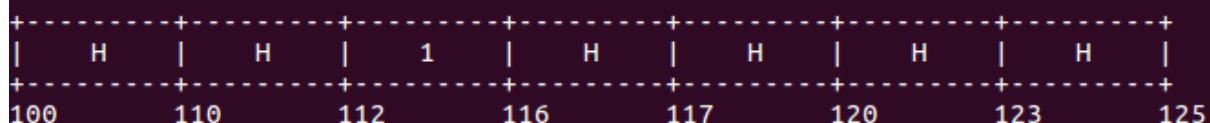
Allocated Memory Space



Free Memory Space



Physical Memory Space



Best Fit Memory Allocation Algorithm

1. Entry / Allocate
2. Exit / Deallocate
3. Display
4. Coalescing of Holes
5. Back to Algorithm
6. Exit

Enter your choice: 2

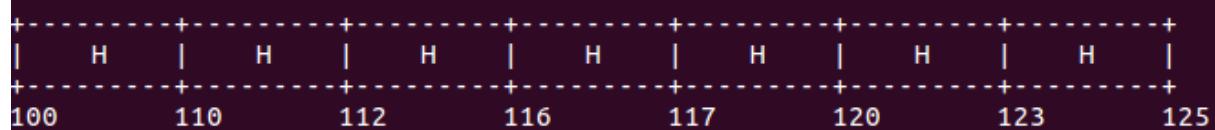
Enter process id : 1

1's memory is deallocated

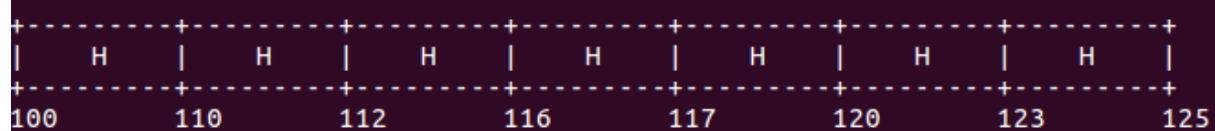
Allocated Memory Space

NULL

Free Memory Space



Physical Memory Space



Worst Fit Memory Allocation Algorithm

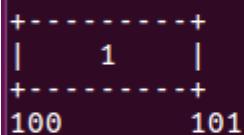
1. Entry / Allocate
 2. Exit / Deallocate
 3. Display
 4. Coalescing of Holes
 5. Back to Algorithm
 6. Exit
- Enter your choice: 1

Enter process id : 1

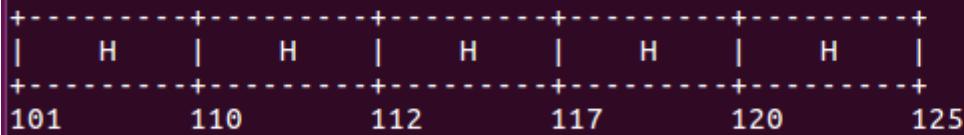
Enter size needed : 1

Memory is allocted to 1

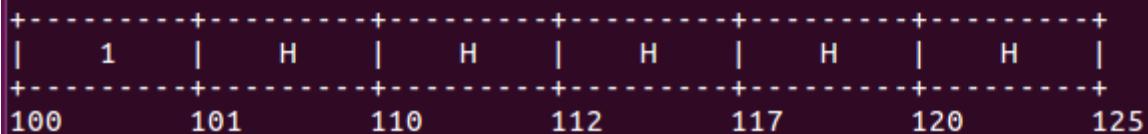
Allocated Memory Space



Free Memory Space



Physical Memory Space



Worst Fit Memory Allocation Algorithm

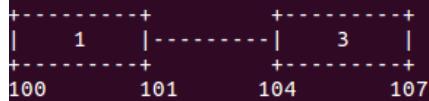
1. Entry / Allocate
2. Exit / Deallocate
3. Display
4. Coalescing of Holes
5. Back to Algorithm
6. Exit

Enter your choice: 1

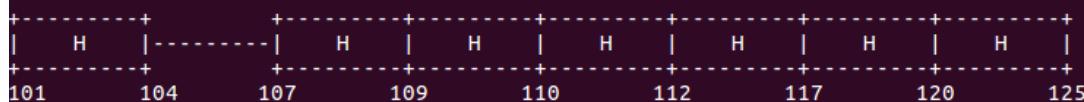
Enter process id : 3
Enter size needed : 3

Memory is allotted to 3

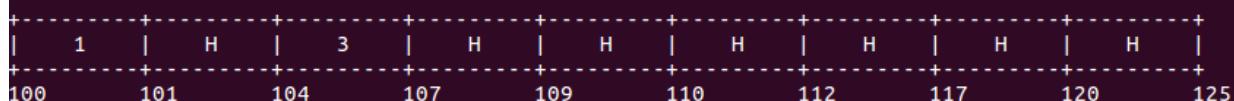
Allocated Memory Space



Free Memory Space



Physical Memory Space



Worst Fit Memory Allocation Algorithm

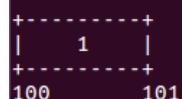
1. Entry / Allocate
2. Exit / Deallocate
3. Display
4. Coalescing of Holes
5. Back to Algorithm
6. Exit

Enter your choice: 2

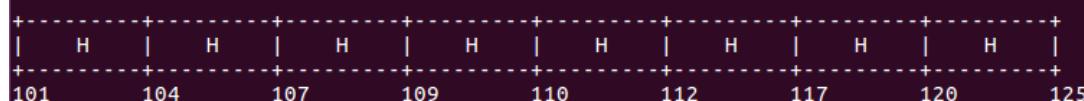
Enter process id : 3

3's memory is deallocated

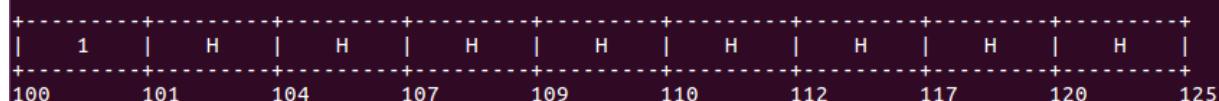
Allocated Memory Space



Free Memory Space



Physical Memory Space



Worst Fit Memory Allocation Algorithm

1. Entry / Allocate
2. Exit / Deallocate
3. Display
4. Coalescing of Holes
5. Back to Algorithm
6. Exit

Enter your choice: 2

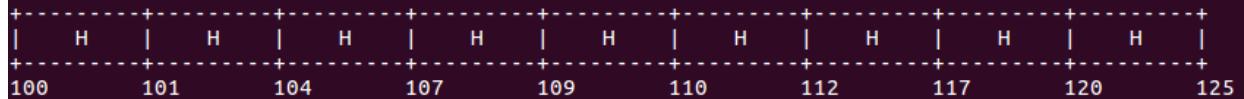
Enter process id : 1

1's memory is deallocated

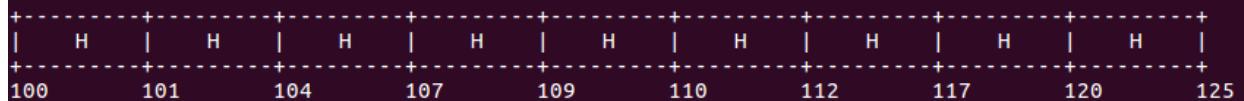
Allocated Memory Space

NULL

Free Memory Space



Physical Memory Space



Learning Outcome:

- Learnt how to allocate memory for processes
- Learnt to manipulate memory and linked lists

LAB EXERCISE 9

Paging Technique

Submission Date:23-05-2022

Name: Jayannthan P T

Dept: CSE 'A'

Roll No.: 205001049

1. To develop a C program to implement the paging technique in memory management

Algorithm:

1. Generate a random number using the srand() function.
2. Create an array of processes of type table.
3. Input the size of the physical memory and the size of each page.
4. Assign the ceil value of the quotient when the size of physical memory is divided by size of page as number of frames.
5. Initially, assign 0 to all elements of the frames array.
6. Assign -1 to random frames.
- 7.-1 indicate that the frames are free
8. Input an integer option.
9. If option is equal to 1, execute steps 9.1 to 9.6.
 - 9.1. Input the process ID and its size.
 - 9.2. Calculate the number of frames that will be occupied by the process.
 - 9.3. If the number of frames required by the process is less than the number of free frames allocate. Else, print that there aren't enough frames.
10. If option is equal to 2, execute steps 10.1 to 10.2.
 - 10.1. Input the ID of the process to be deallocated.
 - 10.2. Assign -1 to all the frames allocated to the process to indicate that the process frames have been deallocated. Remove the process from the process list as well.
11. If option is equal to 3, print the page table for all processes.
12. If option is equal to 4, print all the free frames, i.e., print all the frames where the (j+1)th element in the frames array is equal to -1.
13. If option is equal to 5, exit the program.
14. Else, declare that the user has entered an invalid input.

Code: #include <stdio.h>

```
#include <stdlib.h>
#include <math.h>
#include <time.h>
```

```

#define MAX 50
struct table
{
    int page[MAX];
    int frame[MAX];
    int no_of_pages;
};
int main()
{
    srand(time(0));
    int no_of_frames, option, process, frames_req, no_of_free_frames = 0, no_of_process = 0, m = 1;
    double size, memory_size, page_size;
    int frames[MAX], process_[MAX];
    struct table processes_[MAX];
    printf("\nPAGING TECHNIQUE\n");
    printf("\nEnter the physical memory size: ");
    scanf(" %lf", &memory_size);
    printf("Enter the page size: ");
    scanf(" %lf", &page_size);
    no_of_frames = ceil(memory_size / page_size);
    printf("\nPhysical memory is divided into %d frames.", no_of_frames);
    for (int i = 1; i <= no_of_frames; i++)
        frames[i] = 0;
    int n = 10;
    for (int i = 1; i <= n; i++)
    {
        frames[rand() % no_of_frames + 1] = -1;
        no_of_free_frames++;
    }
    printf("\n\nAfter Initialisation: \n\nFree Frames: ");
    for (int i = 1; i <= no_of_frames; i++)
    {
        if (frames[i] == -1)
            printf("%d ", i);
    }
    int p = 0;
    printf("\n\nMENU:\n\t1. Process request\n\t2. Deallocation\n\t3. Page Table display for all input process\n\t4. Free Frame list display\n\t5. Exit\n");
    do
    {
        printf("\nEnter the option: ");
        scanf("%d", &option);
        switch (option)
        {
            case 1:
                printf("\nEnter the process requirement (ID,size): P");
                scanf(" %d %lf", &process, &size);
                frames_req = ceil(size / page_size);
                processes_[process].no_of_pages = frames_req;
                printf("\nProcess is divided into %d pages.\n\nPage Table for P%d:\n", frames_req, process);
                if (frames_req <= no_of_free_frames)
                {

```

```

p = 0;
no_of_process++;
process_[no_of_process] = process;
for (int i = 1; i <= frames_req;)
{
    for (int j = m, k = 1; k <= no_of_frames; k++)
    {
        if (frames[j] == -1)
        {
            printf("\n\tPage %d : Frame %d \n", i - 1, j);
            no_of_free_frames--;
            frames[j] = process;
            processes_[process].page[p] = i - 1;
            processes_[process].frame[p] = j;
            i++;
            p++;
            break;
        }
        j = j % no_of_frames + 1;
        m = j;
    }
}
else
    printf("\nThere is no enough free frames to allocate for this
process!\n");
break;
case 2:
printf("\nEnter the process to be deallocated: P");
scanf(" %d", &process);
for (int i = 1; i <= no_of_frames; i++)
{
    if (frames[i] == process)
    {
        frames[i] = -1;
        no_of_free_frames++;
    }
}
for (int i = 1; i <= no_of_process; i++)
{
    if (process_[i] == process)
    {
        process_[i] = -1;
        break;
    }
}
break;
case 3:
for (int k = 1; k <= no_of_process; k++)
{
    if (process_[k] != -1)
    {
        printf("\nPage table for P%d:\n", process_[k]);
        int i = 0;

```

```

        for ( ; i < processes_[process_[k]].no_of_pages; i++)
            printf("\n\tPage %d : Frame %d \n",
processes_[process_[k]].page[i], processes_[process_[k]].frame[i]);
    }
}
break;
case 4:
printf("\nFree Frames: ");
for (int i = 1, j = m; i <= no_of_frames; i++)
{
    if (frames[j] == -1)
        printf("%d ", j);
    j = j % no_of_frames + 1;
    m = j;
}
printf("\n");
break;
case 5:
printf("\nProgram terminated\n");
break;
default:
printf("\nInvalid option\n");
break;
}
} while (option != 5);
}
}

```

Output:

```

PAGING TECHNIQUE

Enter the physical memory size: 32
Enter the page size: 1

Physical memory is divided into 32 frames.

After Initialisation:

Free Frames: 1 2 11 14 18 21 27 28

MENU:
1. Process request
2. Deallocation
3. Page Table display for all input process
4. Free Frame list display
5. Exit

```

```
Enter the option: 1  
Enter the process requirement (ID,size): P1 5  
Process is divivded into 5 pages.  
Page Table for P1:  
    Page 0 : Frame 1  
    Page 1 : Frame 2  
    Page 2 : Frame 11  
    Page 3 : Frame 14  
    Page 4 : Frame 18
```

```
Enter the option: 1  
Enter the process requirement (ID,size): P1 5  
Process is divivded into 5 pages.  
Page Table for P1:  
    Page 0 : Frame 1  
    Page 1 : Frame 2  
    Page 2 : Frame 11  
    Page 3 : Frame 14  
    Page 4 : Frame 18
```

```
Enter the option: 1  
Enter the process requirement (ID,size): P2  
2  
Process is divivded into 2 pages.  
Page Table for P2:  
    Page 0 : Frame 21  
    Page 1 : Frame 27
```

```
Enter the option: 4  
Free Frames: 28
```

```
Enter the option: 2  
Enter the process to be deallocated: P1  
Enter the option: 4  
Free Frames: 28 1 2 11 14 18
```

```
Enter the option: 1  
Enter the process requirement (ID,size): P3 3  
Process is divided into 3 pages.  
Page Table for P3:  
    Page 0 : Frame 28  
    Page 1 : Frame 1  
    Page 2 : Frame 2
```

```
Enter the option: 2  
Enter the process to be deallocated: P2  
Enter the option: 4  
Free Frames: 11 14 18 21 27
```

```
Enter the option: 2  
Enter the process to be deallocated: P3  
Enter the option: 4  
Free Frames: 2 11 14 18 21 27 28 1
```

Learning Outcome:

- Learned about the use of paging technique.
- Learned how to allocate memory to processes using paging technique.

LAB EXERCISE 10

Implementation of Page Replacement Algorithms

Submission Date:23-05-2022

Name: Jayannthan P T

Dept: CSE 'A'

Roll No.: 205001049

1. Develop a C program to implement the page replacement algorithms (FIFO, Optimal, LRU and LFU) using linked list.

Algorithm:

1. Start.
2. Get user input and reference string.
3. Get user choice for the page replacement algorithm.

FIFO:

1. Create an empty linked list.
2. Get frame from reference string.
3. Search and check if the frame is already present in the list of frames.
 - a) If not found.
 - b) Insert into list.
 - c) Increase size.
 - d) Check for the oldest frame.
 - e) Replace the oldest frame with the current frame.
 - f) Increment the oldest frame.
4. Insert into table.
5. Increment number of faults.
6. Display table.

Optimal:

1. Create an empty linked list.
2. Search if the frame from the reference string is in the current list.
3. If not
 - If size is lesser than list size then insert and increment size.
 - Iterate through the list.
 - For each frame in the list check the next occurrence in the reference string in the future.
 - Assign and find the max distance.

- Replace the frame with greater future distance.
4. Increment the no of page faults.
 5. Display table.

LRU:

1. Create an empty list.
2. Search if the frame from the reference string is in the current list.
3. If not,
 - If size is less than no of frames then insert and increment size.
 - Iterate through the list.
 - Check the previous frames and assign distance.
 - Calculate max distance for each frame.
 - Replace the frame with max distance.
4. Increment the no of page faults.
5. Display table.

LFU:

1. Create an empty list.
2. Search if the frame from the reference string is in the current list.
3. If not
 - If size is lesser than list size then insert and increment size.
 - If not, iterate through the list and increment frequency.
 - Go backwards and check frequency.
 - Check the least frequency with the frame.
4. If found, increment the frequency.
5. Increment the number of faults.
6. Display table.

Code:

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

struct memory
{
    char page;
    struct memory * next;
};

void init_frame(struct memory *frame)
{
    struct memory *ptr = frame->next;
    while (ptr != NULL)
    {
        ptr->page = '-';
        ptr = ptr->next;
    }
}

void insert_frame(struct memory *frame, char page)
{
    struct memory *new_frame = (struct memory *) malloc(sizeof(struct memory));
    new_frame->page = page;
}
```

```

new_frame->next = NULL;
struct memory *ptr = frame;
while (ptr->next != NULL)
{
    ptr = ptr->next;
}
ptr->next = new_frame;
}

void delete_frame(struct memory *frame)
{
    while (frame->next != NULL)
    {
        struct memory *temp = frame->next;
        frame->next = frame->next->next;
        free(temp);
    }
}

void replace(struct memory *frame, int cur_fault, char page)
{
    int count = 0;
    struct memory *ptr = frame->next;
    while (count < cur_fault && ptr != NULL)
    {
        ptr = ptr->next;
        count++;
    }
    if (ptr != NULL)
    {
        ptr->page = page;
    }
}

int present(struct memory *frame, int start, int end, char page)
{
    int count = 0;
    struct memory *ptr = frame->next;
    while (count != start)
    {
        count++;
        ptr = ptr->next;
    }
    while (count < end && ptr != NULL)
    {
        if (page == ptr->page)
        {
            return count;
        }
        count++;
        ptr = ptr->next;
    }
    return 100;
}

int present_ref(char str_ref[], int start, int end, char page)
{
    for (int i = start; i < end; i++)

```

```

{
    if (page == str_ref[i])
    {
        return i;
    }
}
return 100;
}

int present_last(struct memory *frame, int start, int end, char page)
{
    int pos;
    int count = 0;
    struct memory *ptr = frame->next;
    while (count != start)
    {
        count++;
        ptr = ptr->next;
    }
    while (count <= end && ptr != NULL)
    {
        if (page == ptr->page)
        {
            pos = count;
        }
        count++;
        ptr = ptr->next;
    }
    return pos;
}
int present_last_ref(char ref_str[], int start, int end, char page)
{
    int pos;
    for (int i = start; i <= end; i++)
    {
        if (page == ref_str[i])
        {
            pos = i;
        }
    }
    return pos;
}
int max(int duration[], int page_follow[], int no_frames)
{
    int first = page_follow[0], first_pos = 0;
    for (int i = 0; i < no_frames; i++)
    {
        if (page_follow[i] > first)
        {
            first = page_follow[i];
            first_pos = i;
        }
    }
    if (page_follow[first_pos] == 100)
    {

```

```

        first = duration[first_pos];
        for (int i = 0; i < no_frames; i++)
        {
            if ((duration[i] > first) && (page_follow[i] == 100))
            {
                first = duration[i];
                first_pos = i;
            }
        }
    }
    return first_pos;
}
int min(int page_past[], int no_frames)
{
    int first = page_past[0], first_pos = 0;
    for (int i = 0; i < no_frames; i++)
    {
        if (page_past[i] < first)
        {
            first = page_past[i];
            first_pos = i;
        }
    }
    return first_pos;
}
int frequency(char ref_str[], int start, int end, char page)
{
    int freq = 0;
    for (int i = start; i <= end; i++)
    {
        if (page == ref_str[i])
        {
            freq++;
        }
    }
    return freq;
}
int min_freq(int duration[], int page_past[], int no_frames)
{
    int first = page_past[0], first_pos = 0, count = 0;
    for (int i = 0; i < no_frames; i++)
    {
        if (page_past[i] < first)
        {
            first = page_past[i];
            first_pos = i;
        }
    }
    for (int i = 0; i < no_frames; i++)
    {
        if (page_past[i] == page_past[first_pos])
        {
            count++;
        }
    }
}

```

```

    }
    if (count >= 2)
    {
        first = duration[first_pos];
        int pos = 0;
        for (int i = 0; i < no_frames; i++)
        {
            if ((duration[i] > first) && (page_past[i] == page_past[pos]))
            {
                first = duration[i];
                pos = i;
            }
        }
        return pos;
    }
    return first_pos;
}
void print_frame(char page, struct memory *frame, int page_fault)
{
    printf("\n%c\t--->\t", page);
    struct memory *ptr = frame->next;
    while (ptr != NULL)
    {
        printf("%c\t", ptr->page);
        ptr = ptr->next;
    }
    if (!page_fault) {}
    else
    {
        printf("\tPage fault : %d", page_fault);
    }
}
char frame_page(struct memory *frame, int index)
{
    struct memory *ptr = frame->next;
    int count = 0;
    while (ptr != NULL && count < index)
    {
        ptr = ptr->next;
        count++;
    }
    return ptr->page;
}
void fifo(struct memory *frame, int no_frames, char ref_str[])
{
    init_frame(frame);
    int page_fault = 0, cur_fault = 0;
    for (int i = 0; ref_str[i] != '\0'; i++)
    {
        if (present(frame, 0, no_frames, ref_str[i]) == 100)
        {
            replace(frame, cur_fault, ref_str[i]);
            page_fault++;
            cur_fault = (cur_fault + 1) % no_frames;
        }
    }
}

```

```

        print_frame(ref_str[i], frame, page_fault);
    }
    else
    {
        print_frame(ref_str[i], frame, 0);
    }
}
printf("\nTotal page faults : %d", page_fault);
}

void optimal(struct memory *frame, int no_frames, char ref_str[])
{
    init_frame(frame);
    int duration[100];
    for (int i = 0; i < no_frames; i++)
    {
        duration[i] = 0;
    }
    int page_fault = 0, cur_fault = 0;
    for (int i = 0; ref_str[i] != '\0'; i++)
    {
        if (present(frame, 0, no_frames, '-') != 100 && present(frame, 0,
            no_frames, ref_str[i]) == 100)
        {
            cur_fault = present(frame, 0, no_frames, '-');
            replace(frame, cur_fault, ref_str[i]);
            page_fault++;
            print_frame(ref_str[i], frame, page_fault);
        }
        else if (present(frame, 0, no_frames, ref_str[i]) == 100)
        {
            int page_follow[100];
            for (int j = 0; j < no_frames; j++)
            {
                char item = frame_page(frame, j);
                page_follow[j] = present_ref(ref_str, i + 1, strlen(ref_str),
                    item);
            }
            int cur_fault = max(duration, page_follow, no_frames);
            replace(frame, cur_fault, ref_str[i]);
            page_fault++;
            for (int j = 0; j < no_frames; j++)
            {
                if (j != cur_fault)
                {
                    duration[j]++;
                }
                else
                {
                    duration[j] = 0;
                }
            }
            print_frame(ref_str[i], frame, page_fault);
        }
    }
}

```

```

    {
        print_frame(ref_str[i], frame, 0);
    }
}
printf("\nTotal page faults = %d", page_fault);
}

void lru(struct memory *frame, int no_frames, char ref_str[])
{
    init_frame(frame);
    int page_fault = 0, cur_fault = 0;
    for (int i = 0; ref_str[i] != '\0'; i++)
    {
        if (present(frame, 0, no_frames, '-') != 100 && present(frame, 0,
            no_frames, ref_str[i]) == 100)
        {
            cur_fault = present(frame, 0, no_frames, '-');
            replace(frame, cur_fault, ref_str[i]);
            page_fault++;
            print_frame(ref_str[i], frame, page_fault);
        }
        else if (present(frame, 0, no_frames, ref_str[i]) == 100)
        {
            int page_follow[100];
            for (int j = 0; j < no_frames; j++)
            {
                char item = frame_page(frame, j);
                page_follow[j] = present_last_ref(ref_str, 0, i - 1, item);
            }
            int cur_fault = min(page_follow, no_frames);
            replace(frame, cur_fault, ref_str[i]);
            page_fault++;
            print_frame(ref_str[i], frame, page_fault);
        }
        else
        {
            print_frame(ref_str[i], frame, 0);
        }
    }
    printf("\nTotal page faults = %d", page_fault);
}

void lfu(struct memory *frame, int no_frames, char ref_str[])
{
    init_frame(frame);
    int duration[100];
    for (int i = 0; i < no_frames; i++)
    {
        duration[i] = 0;
    }
    int page_fault = 0, cur_fault = 0;
    for (int i = 0; ref_str[i] != '\0'; i++)
    {
        if (present(frame, 0, no_frames, '-') != 100 && present(frame, 0,
            no_frames, ref_str[i]) == 100)
        {

```

```

        cur_fault = present(frame, 0, no_frames, '-');
        replace(frame, cur_fault, ref_str[i]);
        page_fault++;
        print_frame(ref_str[i], frame, page_fault);
    }
    else if (present(frame, 0, no_frames, ref_str[i]) == 100)
    {
        int page_past[100]; //Finding frequency of page usage
        for (int j = 0; j < no_frames; j++)
        {
            char item = frame_page(frame, j);
            page_past[j] = frequency(ref_str, 0, i - 1, item);
        }
        int cur_fault = min_freq(duration, page_past, no_frames);
        replace(frame, cur_fault, ref_str[i]);
        page_fault++;
        for (int j = 0; j < no_frames; j++)
        {
            if (j != cur_fault)
            {
                duration[j]++;
            }
            else
            {
                duration[j] = 0;
            }
        }
        print_frame(ref_str[i], frame, page_fault);
    }
    else
    {
        print_frame(ref_str[i], frame, 0);
    }
}
printf("\nTotal page faults = %d", page_fault);
}

int main()
{
    struct memory *frame = (struct memory *) malloc(sizeof(struct memory));
    int no_frames, ref_len;
    char frames[100], ref_str[100];

    delete_frame(frame);
    printf("Number of frames: ");
    scanf(" %d", &no_frames);
    for (int i = 0; i < no_frames; i++)
    {
        insert_frame(frame, '-');
    }
    printf("\nReference string length: ");
    scanf(" %d", &ref_len);
    printf("\nReference string: ");
    for (int i = 0; i < ref_len; i++)
    {

```

```

    scanf(" %c", &ref_str[i]);
}

ref_str[ref_len] = '\0';

while (1)
{
    printf("\n1. FIFO\n2. Optimal\n3. LRU\n4. LFU\n5.Exit\nEnter your choice: ");
    int ch;
    scanf(" %d", &ch);
    switch (ch)
    {
        case 1:
        {
            fifo(frame, no_frames, ref_str);
            break;
        }
        case 2:
        {
            optimal(frame, no_frames, ref_str);
            break;
        }
        case 3:
        {
            lru(frame, no_frames, ref_str);
            break;
        }
        case 4:
        {
            lfu(frame, no_frames, ref_str);
            break;
        }
        case 5:
        {
            exit(1);
            break;
        }
        default:
        {
            printf("\nInvalid choice!");
            break;
        }
    }
}
printf("\n");
return 0;
}

```

Output:

1. FIFO
2. Optimal
3. LRU
4. LFU
- 5.Exit

Enter your choice: 1

7	--->	7	-	-	Page fault : 1
0	--->	7	0	-	Page fault : 2
1	--->	7	0	1	Page fault : 3
2	--->	2	0	1	Page fault : 4
0	--->	2	0	1	
3	--->	2	3	1	Page fault : 5
0	--->	2	3	0	Page fault : 6
4	--->	4	3	0	Page fault : 7
2	--->	4	2	0	Page fault : 8
3	--->	4	2	3	Page fault : 9
0	--->	0	2	3	Page fault : 10
3	--->	0	2	3	
2	--->	0	2	3	
1	--->	0	1	3	Page fault : 11
2	--->	0	1	2	Page fault : 12
0	--->	0	1	2	
1	--->	0	1	2	
7	--->	7	1	2	Page fault : 13
0	--->	7	0	2	Page fault : 14
1	--->	7	0	1	Page fault : 15

Total page faults : 15

Enter your choice: 2

7	--->	7	-	-	Page fault : 1
0	--->	7	0	-	Page fault : 2
1	--->	7	0	1	Page fault : 3
2	--->	2	0	1	Page fault : 4
0	--->	2	0	1	
3	--->	2	0	3	Page fault : 5
0	--->	2	0	3	
4	--->	2	4	3	Page fault : 6
2	--->	2	4	3	
3	--->	2	4	3	
0	--->	2	0	3	Page fault : 7
3	--->	2	0	3	
2	--->	2	0	3	
1	--->	2	0	1	Page fault : 8
2	--->	2	0	1	
0	--->	2	0	1	
1	--->	2	0	1	
7	--->	7	0	1	Page fault : 9
0	--->	7	0	1	
1	--->	7	0	1	

Total page faults = 9

```
3) EXTC  
Enter your choice: 3
```

7	--->	7	-	-	Page fault : 1
0	--->	7	0	-	Page fault : 2
1	--->	7	0	1	Page fault : 3
2	--->	2	0	1	Page fault : 4
0	--->	2	0	1	
3	--->	2	0	3	Page fault : 5
0	--->	2	0	3	
4	--->	4	0	3	Page fault : 6
2	--->	4	0	2	Page fault : 7
3	--->	4	3	2	Page fault : 8
0	--->	0	3	2	Page fault : 9
3	--->	0	3	2	
2	--->	0	3	2	
1	--->	1	3	2	Page fault : 10
2	--->	1	3	2	
0	--->	1	0	2	Page fault : 11
1	--->	1	0	2	
7	--->	1	0	7	Page fault : 12
0	--->	1	0	7	
1	--->	1	0	7	

```
Total page faults = 12
```

```
4) EXTC  
Enter your choice: 4
```

7	--->	7	-	-	Page fault : 1
0	--->	7	0	-	Page fault : 2
1	--->	7	0	1	Page fault : 3
2	--->	2	0	1	Page fault : 4
0	--->	2	0	1	
3	--->	2	0	3	Page fault : 5
0	--->	2	0	3	
4	--->	4	0	3	Page fault : 6
2	--->	4	0	2	Page fault : 7
3	--->	3	0	2	Page fault : 8
0	--->	3	0	2	
3	--->	3	0	2	
2	--->	3	0	2	
1	--->	3	0	1	Page fault : 9
2	--->	3	0	2	Page fault : 10
0	--->	3	0	2	
1	--->	1	0	2	Page fault : 11
7	--->	7	0	2	Page fault : 12
0	--->	7	0	2	
1	--->	1	0	2	Page fault : 13

Learning Outcome:

- Learnt to implement page replacement techniques

LAB EXERCISE 11

Threading Applications

Submission Date:23-05-2022

Name: Jayannthan P T

Dept: CSE 'A'

Roll No.: 205001049

1. Write a multithreaded program that calculates various statistical values for a list of numbers. This program will be passed a series of numbers on the command line and will then create three separate worker threads. One thread will determine the average of the numbers, the second will determine the maximum value, and the third will determine the minimum value. For example, suppose your program is passed the integers

90 81 78 95 79 72 85

The program will report

The average value is 82

The minimum value is 72

The maximum value is 95

Algorithm:

1. Initialize global variables sum, min, max, avg
2. Create functions to calculate avg, min, max
3. Inside the main function, create thread identifiers and set of thread attributes
4. Read the argument from the user as command line arguments and have it in the array
5. Get default attributes and create thread for each function
6. Wait for thread to exit before each creation
7. Using join function, close the existing executed thread to compile and link it
8. Print the data obtained from the thread

Code:

```
#include <stdio.h>
#include <pthread.h>
int arr[50], n, i;

void *calc_avg()
{
```

```

float sum = 0;
float average;
/*printf("enter size:");
scanf("%d", &n);
printf("\nenter array elements:\n");
for (i = 0; i < n; i++)
{
    scanf("%d", &arr[i]);
}*/
for (i = 0; i < n; i++)
{
    sum = sum + arr[i];
}
average = sum / n;
printf("The average value is: %.2f", average);
}

void *calc_min()
{

int temp = arr[0];
for (int i = 1; i < n; i++)
{
    if (temp > arr[i])
    {
        temp = arr[i];
    }
}
printf("\nThe Minimum value is:=%d", temp);
}

void *calc_max()
{

int temp = arr[0];
for (int i = 1; i < n; i++)
{
    if (temp < arr[i])
    {
        temp = arr[i];
    }
}
printf("\nThe Maximum value is:=%d\n", temp);
}

int main(int argc, char* argv[])
{
    int m;
    n=argc-1;
    for (i = 0; i < n; i++)
    {
        arr[i]=atoi(argv[i+1]);
    }
    pthread_t t;
    m = pthread_create(&t, NULL, &calc_avg, NULL);
    pthread_join(t, NULL);
    m = pthread_create(&t, NULL, &calc_min, NULL);
}

```

```
    pthread_join(t, NULL);
    m = pthread_create(&t, NULL, &calc_max, NULL);
    pthread_join(t, NULL);
}
```

Output:

```
root@spl25:~/Jayannthan/Threads# ./threads 1 2 3 4 5
The average value is:3.00
The Minimum value is:=1
The Maximum value is:=5
```

Learning Outcome:

- Implemented threading application
- Learnt the importance of threading

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UCS1411 - OPERATING SYSTEMS LAB

LAB EXERCISE 12

File Allocation Techniques

Submission Date:30-05-2022

Name: Jayannthan P T

Dept: CSE 'A'

Roll No.: 205001049

1. To develop a C program to implement the various file allocation techniques.

Algorithm:

1. Get Main memory size and block size as input.
2. Create a Main memory with 'n' number of blocks of equal size.
3. Main memory is maintained as Linked List with structure containing block id, Free / Filename, Link to next Memory block , Link to Next File block (only for Linked Allocation), File block table(integer array to hold block numbers only for Indexed Allocation)
4. Get the number of files and their size as input.
5. Calculate the no. of blocks needed for each file.
6. Select the Allocation Algorithm. – For every algorithm display Directory information and File information.
7. For Contiguous Allocation - For each file do the following
 - i. Generate a random number between 1 to 'n'
 - ii. Check for continuous number of needed file free blocks starting from that random block no.
 - iii. If free then allot that file in those continuous blocks and update the directory structure.
 - iv. else repeat step 1
 - v. If no continuous blocks are free then 'no enough memory error'
 - vi. The Directory Structure should contain Filename, Starting Block, length (no. of blocks)
8. For Linked Allocation- For each file do the following
 - i. Generate a random number between 1 to 'n' blocks.
 - ii. Check that block is free or not.
 - iii. If free then allot it for file. Repeat step 1 to 3 for the needed number of blocks for file and create linked list in Main memory using the field "Link to Next File block".
 - iv. Update the Directory entry which contains Filename, Start block number, Ending Block Number.
 - v. Display the file blocks starting from start block number in Directory upto ending block number by traversing the Main memory Linked list using the field "Link to Next File block.

9.For Indexed Allocation - For each file do the following

- i. Generate a random number between 1 to 'n' blocks for index block.
- ii. Check if it is free else repeat index block selection
- iii. Generate needed number of free blocks in random order for the file and store those block numbers in index block as array in File block table array.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <math.h>
typedef struct
{
    char fname[20];
    int start;
    int length;
    int end;
    struct dir *next;
} dir;
typedef struct
{
    char fname[20];
    int blockid;
    int fileblocktable[100];
    struct node *next;
    struct node *link;
} node;
typedef struct
{
    char fname[20];
    int fsize;
    int fileblocks;
} fileDetails;

void insert(node *head, node data)
{
    node *t;
    node *newnode;
    newnode = (node *)malloc(sizeof(node));
    newnode->blockid = data.blockid;
    strcpy(newnode->fname, data.fname);
    newnode->next = NULL;
    t = head;
    while (t->next != NULL)
    {
        t = t->next;
    }
    t->next = newnode;
}
void contiguous_alloc(node *mainmem, fileDetails filedatal[])
{
    dir d[noOfFiles];
    int i, j;
```

```
int alloc = 0;
int randno;
int occur[noOfBlocks + 1];
node *t;
node *start;
int found;
int count_rand;
for (i = 0; i < noOfFiles; i++)
{
    found = 0;
    count_rand = 0;
    for (j = 1; j <= noOfBlocks; j++)
        occur[j] = 0;
    while (count_rand != noOfBlocks)
    {
        randno = (rand() % noOfBlocks) + 1;
        while (occur[randno] != 0)
            randno = (rand() % noOfBlocks) + 1;
        count_rand++;
        occur[randno] = 1;
        t = mainmem;
        for (j = 0; j < randno; j++)
            t = t->next;
        start = t;
        found = 1;
        for (j = 0; j < filedata[i].fileblocks; j++)
        {
            if (t == NULL)
                break;
            if (strcmp(t->fname, "free") == 0)
            {
                t = t->next;
                continue;
            }
            else
            {
                found = 0;
                break;
            }
        }
        if (found == 1)
        {
            d[alloc].start = start->blockid;
            d[alloc].length = filedata[i].fileblocks;
            strcpy(d[alloc].fname, filedata[i].fname);
            for (j = 0; j < filedata[i].fileblocks; j++)
            {
                strcpy(start->fname, filedata[i].fname);
                start = start->next;
            }
            break;
        }
    }
    if (found == 0)
```

```

        printf("\nMEMORY UNAVAILABLE\n");
    else
        alloc++;
}
printf("No. of files allocated:%d\n", alloc);
printf("\nDirectory\n");
printf("\tFile Name\tStart\tLength\n");
for (i = 0; i < alloc; i++)
    printf("\t%s\t%d\t%d\n", d[i].fname, d[i].start, d[i].length);
}

void linked_alloc(fileDetails filedatal[], node *mainmem)
{
    node *temp, *start, *new;
    int i, j, k;
    int randno;
    int found;
    dir d[noOfFiles];
    int startpos;
    for (i = 0; i < noOfFiles; i++)
    {
        for (j = 0; j < filedatal[i].fileblocks; j++)
        {
            found = 0;
            while (found == 0)
            {
                randno = (rand() % noOfBlocks) + 1;
                temp = mainmem;
                for (k = 0; k < randno; k++)
                    temp = temp->next;
                if (strcmp(temp->fname, "free") == 0)
                {
                    strcpy(temp->fname, filedatal[i].fname);
                    found = 1;
                    if (j == 0)
                    {
                        new = temp;
                        strcpy(d[i].fname, filedatal[i].fname);
                        d[i].start = temp->blockid;
                    }
                    else if (j == filedatal[i].fileblocks - 1)
                    {
                        new->link = temp;
                        temp->link = NULL;
                        d[i].end = temp->blockid;
                    }
                    else
                    {
                        new->link = temp;
                        new = new->link;
                    }
                }
            }
        }
    }
}

```

```

printf("\nDirectory\n");
printf("\tFile Name\tStart\tEnd\n");
for (i = 0; i < noOfFiles; i++)
    printf("\t%s\t\t%d\t%d\n", d[i].fname, d[i].start, d[i].end);
printf("\nIndividual File listing\n");
for (i = 0; i < noOfFiles; i++)
{
    printf("File Name: %s\n", d[i].fname);
    startpos = d[i].start;
    temp = mainmem;
    for (j = 0; j < startpos; j++)
        temp = temp->next;
    printf("\tData-block %d\n", temp->blockid);
    temp = temp->link;
    while (temp != NULL)
    {
        printf("\tData-block %d\n", temp->blockid);
        temp = temp->link;
    }
}
}

void indexed_alloc(fileDetails filedatal[], node *mainmem)
{
    node *temp, *start, *indexblock;
    int i, j, k;
    int indexblockid;
    int randno;
    int found;
    dir d[noOfFiles];
    for (i = 0; i < noOfFiles; i++)
    {
        found = 0;
        while (found != 1)
        {
            randno = (rand() % noOfBlocks) + 1;
            temp = mainmem;
            for (k = 0; k < randno; k++)
                temp = temp->next;
            if (strcmp(temp->fname, "free") == 0)
            {
                found = 1;
                strcpy(temp->fname, filedatal[i].fname);
            }
        }
        indexblock = temp;
        strcpy(d[i].fname, filedatal[i].fname);
        d[i].start = indexblock->blockid;
        for (j = 0; j < filedatal[i].fileblocks; j++)
        {
            found = 0;
            while (found != 1)
            {
                randno = (rand() % noOfBlocks) + 1;
                temp = mainmem;

```

```

        for (k = 0; k < randno; k++)
            temp = temp->next;
        if (strcmp(temp->fname, "free") == 0)
        {
            found = 1;
            strcpy(temp->fname, filedata[i].fname);
            indexblock->fileblocktable[j] = temp->blockid;
        }
    }
}
printf("\nDirectory\n");
printf("\tFile Name\tIndexed Block\n");
for (i = 0; i < noOfFiles; i++)
    printf("\t%s\t%d\n", d[i].fname, d[i].start);
printf("\n\nIndex Table\n");
printf("File Name\tBlock Indexed\n");
for (i = 0; i < noOfFiles; i++)
{
    indexblockid = d[i].start;
    temp = mainmem;
    for (j = 0; j < indexblockid; j++)
        temp = temp->next;
    printf("\n%s", temp->fname);
    for (j = 0; j < filedata[i].fileblocks; j++)
        printf("\t\t\tData-block %d\n", temp->fileblocktable[j]);
}
}

void main()
{
    int mem_size, choice, i;
    node data;
    node *mainmem;
    node *temp;
    fileDetails filedata[100];
    char c;
    mainmem = malloc(sizeof(node));
    mainmem->next = NULL;
    printf("Enter the main memory size:");
    scanf("%d", &mem_size);
    printf("Enter the block size:");
    scanf("%d", &block_size);
    noOfBlocks = (int)mem_size / block_size;
    printf("Total no. of blocks available:%d\n", noOfBlocks);
    for (i = 0; i < noOfBlocks; i++)
    {
        data.blockid = i + 1;
        strcpy(data.fname, "free");
        insert(mainmem, data);
    }
    printf("Number of files to be allocated:");
    scanf("%d", &noOfFiles);
    for (i = 0; i < noOfFiles; i++)

```

```

{
    printf("\nName of file %d:", i + 1);
    scanf("%s", filedatal[i].fname);
    printf("Size of file %d(in KB):", i + 1);
    scanf("%d", &filedata[i].fsize);
    filedata[i].fileblocks = ceil((float)filedata[i].fsize / (float)block_size);
}
do
{
    printf("\n\nFILE ALLOCATION TECHNIQUES\n");
    printf("1.Contiguous\n");
    printf("2.Linked\n");
    printf("3.Indexed\n");
    printf("Enter choice:");
    scanf("%d", &choice);
    temp = mainmem->next;
    while (temp != NULL)
    {
        strcpy(temp->fname, "free");
        temp = temp->next;
    }
    srand(time(NULL));
    switch (choice)
    {
        case 1:
            contiguous_alloc(mainmem, filedatal);
            break;
        case 2:
            linked_alloc(filedatal, mainmem);
            break;
        case 3:
            indexed_alloc(filedatal, mainmem);
            break;
    }
    printf("\nDo you want to continue?:");
    scanf("%s", &c);
} while (c == 'y');
}

```

Output:

```
Enter the main memory size:500
Enter the block size:10
Total no. of blocks available:50
Number of files to be allocated:5
```

```
Name of file 1:f1
Size of file 1(in KB):10
```

```
Name of file 2:f2
Size of file 2(in KB):20
```

```
Name of file 3:f3
Size of file 3(in KB):10
```

```
Name of file 4:f4
Size of file 4(in KB):5
```

```
Name of file 5:f5
Size of file 5(in KB):5
```

FILE ALLOCATION TECHNIQUES

- 1.Contiguous
 - 2.Linked
 - 3.Indexed
- Enter choice:1
No. of files allocated:5

Directory

File Name	Start	Length
f1	32	1
f2	17	2
f3	36	1
f4	14	1
f5	33	1

FILE ALLOCATION TECHNIQUES

- 1.Contiguous
 - 2.Linked
 - 3.Indexed
- Enter choice:2

Directory

File Name	Start	End
	7	0
f2	36	28
f3	14	32764
f4	8	0
f5	43	0

Individual File listing

```
File Name:
    Data-block 7
File Name: f2
    Data-block 36
    Data-block 28
File Name: f3
    Data-block 14
File Name: f4
    Data-block 8
File Name: f5
    Data-block 43
```

FILE ALLOCATION TECHNIQUES

- 1.Contiguous
 - 2.Linked
 - 3.Indexed
- Enter choice:3

Directory

File Name	Indexed Block
	9
f2	11
f3	14
f4	26
f5	12

Index Table

File Name	Block Indexed
	Data-block 21
f2	Data-block 1 Data-block 35
f3	Data-block 47
f4	Data-block 17
f5	Data-block 33

Learning Outcome:

- Learnt about the various file allocation techniques
- Learnt to implement the c program for the various file allocation techniques

LAB EXERCISE 13

File Organization Techniques

Submission Date:30-05-2022

Name: Jayannthan P T

Dept: CSE 'A'

Roll No.: 205001049

1. To develop a C program to implement the following file organization techniques
 - a) Single level Directory
 - b) Hierarchical Structure.

Algorithm:

1. Single Level Directory
 - a. Maintain a table containing the filename and the starting address location of that file.
 - b. Give options for creating a new file.
 - c. When creating the file, check for name collision.
 - d. Update the table accordingly.
2. Tree Structured Directory
 - a. Maintain tables for each directory starting from root.
 - b. Create a structure for a node in tree which contains an array to hold directories and an array to hold files.
 - c. Limit each directory to have a maximum of three sub-directories and files.
 - d. For each sub-directory follow the same table structure as described above

Code:

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

char files[100][20];
int address[100];
int cur = 1;
typedef struct dir
{
    char dirname[20];
    char filename[100][10];
    int address[100];
```

```

    int curfile;
    struct dir *ptr1, *ptr2, *ptr3;
}
dir;
dir * root;
dir* create(char name[20])
{
    dir *p = (dir*) malloc(sizeof(dir));
    p->ptr1 = NULL;
    p->ptr2 = NULL;
    p->ptr3 = NULL;
    p->curfile = 0;
    strcpy(p->dirname, name);
    return p;
}
void singleLevel()
{
    cur = 1;
    printf("\n\t\tSINGLE LEVEL FILE SYSTEM\n");
    int choice = -1;
    printf("\n\tMenu : \n\t1.Create file\n\t2.Print files \n\t3.Exit\n\tEnter Choice:");
    scanf("%d", &choice);
    do
    {   // if (choice == 3) return;
        if (choice == 2)
        {
            printf("\nFiles.....\n");
            for (int i = 1; i <= cur-1; i++) printf("%d.%s\t \n", i, files[i]);
            printf("\nFiles printed...\n");
        }
        else if (choice == 1)
        {
            char name[20];
            int found = 0;
            printf("\n\tName of the file : ");
            scanf("%s", name);
            for (int i = 1; i <= cur-1; i++)
            {
                if (strcmp(name, files[i]) == 0)
                {
                    found = 1;
                    break;
                }
            }
            if (found)
            {
                printf("\n\tFile name already exists!\n");
            }
            else
            {
                strcpy(files[cur++], name);
                address[cur - 1] = rand() % 10000;
                printf("\n\tFile created successfully!\n");
            }
        }
    }
}

```

```

    }

    printf("\n\tMenu : \n\t1.Create file\n\t2.Print files \n\t3.Exit\n\t\tEnter
Choice: ");
    scanf("%d", &choice);
} while (choice != 3);
}

dir* find(dir *p, char name[20])
{
    if (p == NULL) return NULL;
    if (strcmp(p->dirname, name) == 0) return p;
    dir *p1 = find(p->ptr1, name);
    if (p1 != NULL) return p1;
    dir *p2 = find(p->ptr2, name);
    if (p2) return p2;
    dir *p3 = find(p->ptr3, name);
    if (p3) return p3;
    return NULL;
}
void print(dir *p)
{
    printf("\nDirectory Name : %s\n", p->dirname);
    if (p->curfile == 0)
    {
        printf("\n\tNO FILES IN DIRECTORY!\n");
    }
    else
    {
        printf("Files in Directory :");
        for (int i = 1; i <= p->curfile; i++)
        {
            printf("\n\t%s", p->filename[i]);
        }
    }
    if (p->ptr1 == NULL) return;
    printf("\n\tSubdirectories :");
    if (p->ptr1 != NULL)
    {
        printf("\n\t\t%s", p->ptr1->dirname);
    }
    if (p->ptr2 != NULL)
    {
        printf("\n\t\t%s", p->ptr2->dirname);
    }
    if (p->ptr3 != NULL)
    {
        printf("\n\t\t%s", p->ptr3->dirname);
    }
    printf("\n\n");
}

void display(dir *p)
{
    if (p == NULL)

```

```

    return;
    print(p);
    display(p->ptr1);
    display(p->ptr2);
    display(p->ptr3);
}
void hierarchial()
{
    cur = 1;
    root = create("root");
    printf("\n\t\tTREE LEVEL FILE SYSTEM\n");
    int choice = -1;
    printf("\n\tMenu : \n\t1.Create directory\n\t2.Create file\n\t3.Print files
\n\t4.Exit\n\tEnter Choice: ");
    scanf("%d", &choice);
    do
    {
        if (choice == 1)
        {
            char name[20];
            printf("\n\tName of directory to be created :");
            scanf("%s", name);
            char parent[20];
            printf("\n\tParent name : ");
            scanf("%s", parent);
            dir *p = find(root, parent);
            if (p == NULL) printf("\n\tDirectory not found!");
            else
            {
                if (p->ptr1 == NULL)
                {
                    dir *temp = create(name);
                    p->ptr1 = temp;
                    printf("\n\tDirectory successfully created!\n");
                }
                else if (p->ptr2 == NULL)
                {
                    if (strcmp(p->ptr1->dirname, name) == 0) printf("\n\t Name already
exists!\n");
                    else
                    {
                        dir *temp = create(name);
                        p->ptr2 = temp;
                        printf("\n\tDirectory successfully created!\n");
                    }
                }
                else if (p->ptr3 == NULL)
                {
                    if (strcmp(p->ptr1->dirname, name) == 0 || strcmp(p->ptr2->dirname,
name) == 0) printf("\n\tDirectory Name already exists!\n");
                    else
                    {
                        dir *temp = create(name);
                        p->ptr3 = temp;
                    }
                }
            }
        }
    }
}

```

```

        printf("\n\tDirectory successfully created!\n");
    }
}
else printf("\nCannot create directory. Space exceeded!\n");
}
else if (choice == 2)
{
    char file[20];
    printf("\n\tEnter file name : ");
    scanf("%s", file);
    printf("\n\tEnter directory under which you want to create the file : ");
    char direc[20];
    scanf("%s", direc);
    dir *p = find(root, direc);
    if (p == NULL)
    {
        printf("\n\tDirectory does not exist!");
    }
    else
    {
        int found = 0;
        for (int i = 1; i <= p->curfile; i++)
        {
            if (strcmp(p->filename[i], file) == 0)
            {
                found = 1;
                break;
            }
        }
        if (found) printf("\n\tFilename already exists!");
        else
        {
            strcpy(p->filename[+p->curfile], file);
            p->address[p->curfile] = rand() % 10000;
            printf("\n\tFile successfully created!");
        }
    }
}
else if (choice == 3)
{
    printf("\nDisplaying directory structure.....\n");
    display(root);
}
else
{
    printf("\n Enter Valid Choice\n");
}
printf("\n\tMenu : \n\t1.Create directory\n\t2.Create file\n\t3.Print files
\n\t4.Exit\n\tEnter Choice: ");
scanf("%d", &choice);
} while (choice != 4);
}

int main()

```

```
{  
    int option = -1;  
    printf("\nMenu : \n1.Single level\n2.Hierarchial \n3.Exit\n\tEnter Choice: ");  
    scanf("%d", &option);  
    do {  
        switch (option)  
        {  
            case 1:  
                singleLevel();  
                break;  
            case 2:  
                hierachial();  
                break;  
            default:  
                printf("\n Enter Valid Choice\n");  
                break;  
        }  
        printf("\nMenu : \n1.Single level\n2.Hierarchial \n3.Exit\n\tEnter Choice: ");  
        scanf("%d", &option);  
    } while (option != 3);  
    return 0;  
}
```

Output:

Enter Choice: 1

SINGLE LEVEL FILE SYSTEM

Menu :

- 1.Single level
- 2.Hierachial
- 3.Exit

Enter Choice: 1

SINGLE LEVEL FILE SYSTEM

Menu :

- 2.Print files
- 3.Exit

Enter Choice: 1

Name of the file : f1

File created successfully!

Menu :

- 1.Create file
- 2.Print files
- 3.Exit

Enter Choice: 1

Name of the file : f1

File name already exists!

Menu :

- 1.Create file
- 2.Print files
- 3.Exit

Enter Choice: 1

Name of the file : f2

File created successfully!

Menu :

- 1.Create file
- 2.Print files
- 3.Exit

Enter Choice: 2

Files.....

- 1.f1
- 2.f2

TREE LEVEL FILE SYSTEM

Menu :

- 1.Create directory
- 2.Create file
- 3.Print files
- 4.Exit

TREE LEVEL FILE SYSTEM

Menu :

- 1.Create directory
- 2.Create file
- 3.Print files
- 4.Exit

Enter Choice: 1

Name of directory to be created :dir1

Parent name : root

Directory successfully created!

Menu :

- 1.Create directory
- 2.Create file
- 3.Print files
- 4.Exit

Enter Choice: 1

Name of directory to be created :dir1

Parent name : root

Name already exists!

```
Menu :  
1.Create directory  
2.Create file  
3.Print files  
4.Exit  
Enter Choice: 2  
  
Enter file name : f1  
  
Enter directory under which you want to create the file : root  
3.Print files  
4.Exit  
Enter Choice: 2  
  
Enter file name : f1  
  
Enter directory under which you want to create the file : dir1  
  
File successfully created!  
Menu :  
1.Create directory  
2.Create file  
3.Print files  
4.Exit  
Enter Choice: 1  
  
Name of directory to be created :dir2  
  
Parent name : dir1  
  
Directory successfully created!  
  
Menu :  
1.Create directory  
2.Create file  
3.Print files  
4.Exit  
Enter Choice: 2  
  
Enter file name : f1  
  
Enter directory under which you want to create the file : dir2  
  
File successfully created!
```

Learning Outcome:

- Learnt about the different directory structures
- Implemented single level directory and tree structured directory using c program