Operating System
Lab Report

Class – BCSE
Year – 3rd year 1st semester
Session – 2018-19

Sourav Dutta– Roll 001610501076

ASSIGNMENT - 1

1. Write a shell script that has 2 user created variables, userv1 and userv2. Ask for the values for the variables from the user and take in any type (real/integer/character) for the 2 variables. Print them as:
(i) value of userv2 followed by value of userv1 separated by a comma and
(ii) value of userv1 using single quotes followed by value of userv2 using double quotes separated from userv1 by the word "and".

CODE SNIPPET :

```
#! /bin/sh

read -p "Enter variable 1: " userv1
read -p "Enter variable 2: " userv2

echo
echo "$userv2,$userv1"
echo "'$userv1' and \"$userv2\""
```

EXPLANATION :

The first two lines is using to print the prompt message and then reading a variable at the same line (using –p). The values entered are stored in two variables names userv1 and userv2.

The prompt, $, which is called the command prompt, is issued by the shell. While the prompt is displayed, you can type a command.

In the third line "echo" is used to print a blank line to the screen.

In the next line, userv2 is displayed first. Then a comma is printed, and then the value stored in userv1. In the last line, userv1 is printed in single quotes, separated by "and", then userv2 is printed in double quotes. Since we need to print the double quotes as well, we need to open and close the string with backslash '\'.

OUTPUT :

```
$ bash q1.sh
Enter variable 1: 123.422
Enter variable 2: string

string,123.422
'123.422' and "string"
```

3. Write a shell script that takes 4 file names (containing C programs) as command line arguments and print the frequency of the occurrences of the following 3 strings "printf", "scanf", "int" in each file. The output (in tabular format) should clearly denote the frequency of each string in each file.

CODE SNIPPET :

```
#! /bin/sh

searchstrings=("printf" "scanf" "int")
filenames=($1 $2 $3 $4)
echo "--------------------------------------------------------------------------------"
echo -e "Name of the
file\t${searchstrings[0]}\t\t${searchstrings[1]}\t\t${searchstrings[2]}\t"
echo "--------------------------------------------------------------------------------"

for file in ${filenames[*]}
do
      echo -n -e "\t$file\t"
      for string in ${searchstrings[*]}
      do
            freq=$(grep -c "$string" $file)
            echo -n -e "\t$freq\t"
      done
      echo
done
echo "--------------------------------------------------------------------------------"
```

EXPLANATION :

Two lists is initialised, the first list named "searchstrings" is stored with the character strings that needs to be searched in each file (i.e. printf, scanf, int). The second list named filenames stores the names of the files that the user enters as arguments while executing the shell program.

For each file which is stored in the filenames list, the frequency of a particular string to be searched can be done using "grep –c"

grep searches the input files for lines containing a match to a given pattern list. When it finds a match in a line, it copies the line to standard output (by default), or whatever other sort of output you have requested with options.
" –e " is used to enable interpretation of backslash escapes (i.e. to make use of \t)

OUTPUT :

```
$ bash q3.sh a.c b.c c.c d.c
--------------------------------------------------------------------------------
Name of the file        printf          scanf           int
--------------------------------------------------------------------------------
        a.c             2               1               4
        b.c             1               2               1
        c.c             1               3               2
        d.c             0               0               4
--------------------------------------------------------------------------------
```

4. Write a shell script that accepts a file name as an input. The program then asks for a string of characters (that is, any word) to be provided by the user. The file will be searched to find whether it contains the given word.

(i) If file contains the given word, the program will display the total number of occurrences of the word.

(ii) The program is also required to display line number/s in which the word has occurred as well the frequency of the word in that line. (Note: the word may occur more than once in a given line).

(iii) If the file does not contain the word, an appropriate message will be displayed.

CODE SNIPPET :

```
#! /bin/sh

read -p "Input file name: " filename
read -p "Input string to be searched in file: " string

echo
freq=$(grep -o $string $filename | wc -l)
if [ $freq == 0 ]
then
      echo "No \"$string\" found in file \"$filename\""
      exit 1
fi

echo "Occurence of \"$string\" in file \"$filename\" is : $freq"

freqarr=(`grep -o -n $string $filename | cut -d : -f 1 `)

echo
echo "-----------------------------------"
echo -e "\tLine number \tFrequency"
echo "-----------------------------------"

for(( i=0; i<${#freqarr[@]}; i+=2 ))
do
      echo -e "\t${freqarr[$i+1]} \t\t${freqarr[$i]}"
done
echo "-----------------------------------"
echo
```

EXPLANATION :

`grep -o $string $filename | wc –l`

The above line first outputs the string in different lines. The numbers of lines is the frequency of the string in the designated file. So, to get the frequency of the string, " wc – l" is done.

`freqarr=(`grep -o -n $string $filename | cut -d : -f 1 `)`

The "`grep -o -n $string $filename`" outputs the lines having two numbers. The first number is the frequency of the string, and the second number is the line number. So, is done to separately store the two values as adjacent elements of the list.

Therefore, the odd indices of freqarr array have the frequencies and even indices have the line numbers. That is why the loop is incremented by 2.

" –e " is used to enable interpretation of backslash escapes (i.e. to make use of \t)

OUTPUT :

```
$ bash q4.sh
Input file name: a.c
Input string to be searched in file: int

Occurence of "int" in file "a.c" is : 4

-------------------------------------
        Line number     Frequency
-------------------------------------
        1               2
        2               2
-------------------------------------
```

5. Extend the shell script written in question (4) to perform the following task : User is asked to enter another word.
(i) The first word (entered in question (4)) will have to be matched with the contebts of the file and replaced by the second word, if a match occurs.
(ii) If the first word does not occur in the file, an appropriate message will be displayed
(iii) Ignore replacing partial matches, but show that partial matches do exists.

CODE SNIPPET :

```
#! /bin/sh

read -p "Input file name: " filename
read -p "Input string to be searched in file: " string

echo
freq=$(grep -o $string $filename | wc -l)
if [ $freq == 0 ]
then
     echo "No \"$string\" found in file \"$filename\""
     exit 1
fi

read -p "Now Input string to be replaced with: " replacestring

sed -i 's/\<'"$string"'\>/'"$replacestring"'/g' $filename
```

EXPLANATION :

```
grep -o $string $filename | wc -l
```

The above line first outputs the string in different lines. The numbers of lines is the frequency of the string in the designated file. So, to get the frequency of the string, " wc – l" is done.

"- i" is used to edit files in place. Sed is a stream editor. A stream editor is used to perform basic text transformations on an input stream (a file or input from a pipeline).

OUTPUT :

```
SOURAV@DESKTOP-A8G1UT1 /cygdrive/f/shell_sourav/a1
$ cat a.c
printf printf
int scanf int

SOURAV@DESKTOP-A8G1UT1 /cygdrive/f/shell_sourav/a1
$ bash q5.sh
Input file name: a.c
Input string to be searched in file: int

Now Input string to be replaced with: double

SOURAV@DESKTOP-A8G1UT1 /cygdrive/f/shell_sourav/a1
$ cat a.c
printf printf
double scanf double

SOURAV@DESKTOP-A8G1UT1 /cygdrive/f/shell_sourav/a1
$ bash q5.sh
Input file name: a.c
Input string to be searched in file: ok

No "ok" found in file "a.c"
```

ASSIGNMENT - 2

1. a. With the help of a Linux command show the details of currently running processes with real-time update of CPU usage in the system. Keep this command executing in one window.

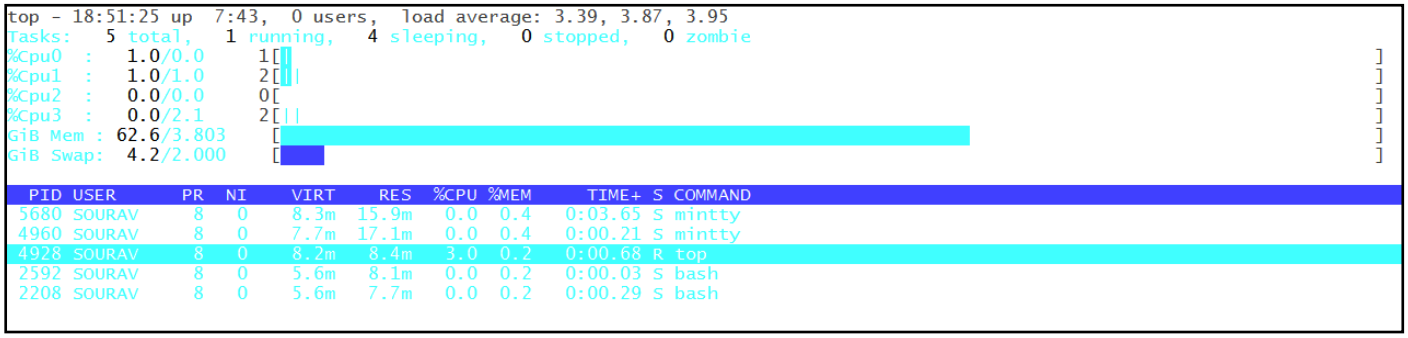
CODE SNIPPET :

```
top -o PID
```

EXPLANATION :

Top command shows all running processes in the server. It shows you the system information and the processes information just like up-time, average load, tasks running, no. of users logged in, no. of CPU processes, RAM utilisation and it lists all the processes running/utilised by the users in your server.

OUTPUT :



b. Write a program in C that runs for 30 seconds, but does not perform any I/O. Execute this in another window and see the output of the window of (a).
Now stop this process (before the process automatically stops) and see how the change is reflected in the other window.

CODE SNIPPET :

```
#include<stdio.h>
#include<time.h>
#include<sys/time.h>

void delay(int second) {
	int totaldelay = CLOCKS_PER_SEC * second;
	clock_t curtime = clock();
	while(clock() < curtime + totaldelay)
	;
}
int main() {

	int i;
	for(i=1;i<=30;i++) {
		delay(1);
		printf("%2d second have elapsed.\n", i);
	}

	return 0;
}
```

EXPLANATION :

When the above program is in execution, the state of the process goes to “running” state; which is noticed in the window running the top command. After complete successful execution of the program (i.e. after 30 seconds have been elapsed), the process state again shifts away from its running state.

When the above program is stopped (using CTRL + Z) before completion of its successful execution (i.e. the process is stopped before 30 seconds), the state of the process shifts from “running” state to “stopped” state; which is noticed in the window running the top command.

OUTPUT :

```
SOURAV@DESKTOP-A8G1UT1 /cygdrive/f/shell_sourav/a2
$ gcc q1b.c -o test

SOURAV@DESKTOP-A8G1UT1 /cygdrive/f/shell_sourav/a2
$ ./test
 1 second have elapsed.
 2 second have elapsed.
 3 second have elapsed.
 4 second have elapsed.
 5 second have elapsed.

[1]+  Stopped                 ./test
```

c. Find a Linux command that will bring the process to running state again.

CODE SNIPPET :

```
fg
```

EXPLANATION :

After we stop a process from completing its successful execution, we have seen that the state of the process shifts from “running” state to “stopped” state. To resume the process back to its “running” from its “stopped” state, fg command is used.

OUTPUT :

```
SOURAV@DESKTOP-A8G1UT1 /cygdrive/f/shell_sourav/a2
$ fg
./test
 6 second have elapsed.
 7 second have elapsed.
 8 second have elapsed.
 9 second have elapsed.
10 second have elapsed.
11 second have elapsed.
12 second have elapsed.
```

d. Modify the above program that will now perform some I/O. This program exits normally after completing its task. Execute this in another window and track the output in the window of (a).

CODE SNIPPET :

```
#include<stdio.h>

int main(){
      int num;
      printf("Enter a number : ");
      scanf("%d", &num);
      printf("You entered : %d \n", num);

      return 0;
}
```

EXPLANATION :

When the above program is in execution, we see that the process state shifts from “running” state to “sleeping” state instantaneously. This is because when the printf statement is executed, the process is in “running” state. But as soon as the scanf statement occurs, the system waits for the user to enter a value; and hence goes to “sleeping” state. After successful input from the user, the program execution again resumes, and process state shifts to “running” state. And then the process is terminated successfully after execution of the last printf statement.

2. Write a shell script to check if a file exists and if it does exist, check the type of the file. (Use test).

CODE SNIPPET :

```
#!/bin/sh

read -p "Enter file name : " filename
test -e $filename && echo $(file $filename) || echo "File not found!"
```

EXPLANATION :

The first line of the code is used to print the prompt and read the file name. The name of the file is saved in filename.
Using the test function, file is check whether the entered file exists or not (by using the specifier –e).

If the file exists, file command with the file name is used to print the full description about the type of the file. If the file does not exists, a prompt saying file could not be found is displayed.

OUTPUT :

```
SOURAV@DESKTOP-A8G1UT1 /cygdrive/f/shell_sourav/a2
$ bash q2.sh
Enter file name : q1b.c
q1b.c: C source, ASCII text

SOURAV@DESKTOP-A8G1UT1 /cygdrive/f/shell_sourav/a2
$ bash q2.sh
Enter file name : a.txt
File not found!
```

3. Create child processes: X and Y.

a. Each child process performs 10 iterations. The child process just displays its name/id and the current iteration number, and sleeps for some random amount of time. Adjust the sleeping duration of the processes to have a different output (i.e. another interleaving of processes' traces).

CODE SNIPPET :

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
int main() {
      pid_t child1, child2;
      int parent_pid = getpid();
         child1 = fork();
         int i, j;
         if(child1 < 0) {
                 perror("Fork failed.\n");
                 return 1;
         } else if(child1 == 0) {
                 for(i=1;i<=10;i++) {
                         printf("Parent PID : %d, Process PID : %d, Iteration
no: %d\n", parent_pid, getpid(), i);
                         sleep(2);
                 }
         } else {
                 child2 = fork();
                 if(child2 < 0) {
                         perror("Fork failed.\n");
                 } else if(child2 == 0 ) {
                         for(j=1;j<=10;j++) {
                                 printf("Parent PID : %d, Process PID : %d,
Iteration no: %d\n", parent_pid, getpid(), j);
                                 sleep(1);
                         }
                 }

         }
        wait(NULL);
         wait(NULL);
         return 0;
}
```

EXPLANATION :

```
int parent_pid = getpid();
```

In the above program, we create two processes, and we let it X and Y. The PID of parent is saved in an integer variable named "parent_pid" by invoking the getpid() function. Alternatively, we can use getppid() to get the parent PID, when the current child process is in execution.

```
child1 = fork();
```

fork() is used to create a new child process. The value returned by fork() is stored in child1, which is of type pid_t.

When fork() returns a negative value, it signifies that the the child process could not be created. And on successful creation of a child process, fork() returns the value 0. If the returned value is greater than 0, that means the process is back in parent process.

```
for(i=1;i<=10;i++) {
	printf("Parent PID : %d, Process PID : %d, Iteration no: %d\n",
parent_pid, getpid(), i);
	sleep(2);
}
```

When first child is created, we loop 10 times to print the iteration to show that the current process is scheduled to the child process.

Similarly, we create a second child and we loop 10 times to print iteration number, parent PID and its PID. Sleep(2) and sleep(3) has been used so that the two process does not take same time to execute.

OUTPUT :

```
$ ./test
Parent PID : 3004, Process PID : 4492, Iteration no: 1
Parent PID : 3004, Process PID : 3108, Iteration no: 1
Parent PID : 3004, Process PID : 3108, Iteration no: 2
Parent PID : 3004, Process PID : 4492, Iteration no: 2
Parent PID : 3004, Process PID : 3108, Iteration no: 3
Parent PID : 3004, Process PID : 3108, Iteration no: 4
Parent PID : 3004, Process PID : 4492, Iteration no: 3
Parent PID : 3004, Process PID : 3108, Iteration no: 5
Parent PID : 3004, Process PID : 3108, Iteration no: 6
Parent PID : 3004, Process PID : 4492, Iteration no: 4
Parent PID : 3004, Process PID : 3108, Iteration no: 7
Parent PID : 3004, Process PID : 3108, Iteration no: 8
Parent PID : 3004, Process PID : 4492, Iteration no: 5
Parent PID : 3004, Process PID : 3108, Iteration no: 9
Parent PID : 3004, Process PID : 3108, Iteration no: 10
Parent PID : 3004, Process PID : 4492, Iteration no: 6
Parent PID : 3004, Process PID : 4492, Iteration no: 7
Parent PID : 3004, Process PID : 4492, Iteration no: 8
Parent PID : 3004, Process PID : 4492, Iteration no: 9
Parent PID : 3004, Process PID : 4492, Iteration no: 10
```

b. Modify the program so that X is not allowed to start iteration i before process Y has terminated its own iteration i-1. Use semaphore to implement this synchronization.

CODE SNIPPET :

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<semaphore.h>
#include<fcntl.h>
#include<sys/types.h>
#include<sys/wait.h>
#define S1 "/sem1"
#define S2 "/sem2"
```

```
void display(unsigned int t,sem_t* p2_sw, sem_t* p2_sp ){
      for(int i = 1; i <= 10; i++){
            sem_wait(p2_sw);
            printf("Parent: %d, Process: %d, Itr: %d\n",getppid(),getpid(),i);
            sem_post(p2_sp);
            sleep(t);
      }
}

int main(){
      sem_t *p2_sem1, *p2_sem2;
      pid_t pid1 = -1, pid2 = -1;
      if((p2_sem1 = sem_open(S1,O_CREAT,0660,1)) == SEM_FAILED
            || (p2_sem2 = sem_open(S2,O_CREAT,0060,0)) == SEM_FAILED){
            perror("sem_open() Failed");
      }
      if((pid1 = fork()) < 0){
            perror("fork() Failed!");
            exit(1);
      }
      else if(pid1 == 0){                    // process X
            display(3,p2_sem1,p2_sem2);
      }
      else{
            if((pid2 = fork()) < 0){
                  perror("fork() Failed!");
                  exit(1);
            }
            else if(pid2 == 0){              // process Y
                  display(2,p2_sem2,p2_sem1);
            }
            else{                            // parent
                  wait(NULL);
                  wait(NULL);
                  if(sem_unlink(S1) == -1 || sem_unlink(S2) == -1){
                        perror("sem_unlink() Failed");
                  }
            }
      }
      return 0;
}
```

EXPLANATION :

In the above program, semaphore is used to provide the synchronization.

sem_open() function is used to create the semaphore which is initialized to 0. We know that the value of semaphore denotes the number of waiting process to access the resource. If the value is negative then the absolute value denotes the number of waiting processes. If the value is 0 or positive, it indicates that the resource can be allocated to the process arrived.
sem_wait() function is used before executing process X in each iteration. This makes the process X wait for negative value of semaphore.
sem_post() function is used to send a signal to process X after the completion of process Y in each iteration. The main purpose of this function is to increase the value of semaphore by 1 so that it becomes non-negative and process X can start its execution.

4. Implement the following programs using different IPC mechanisms. Your choice is restricted to Pipes, FIFOs, and Message Queues (use different mechanisms for each program)

a. Broadcasting weather information (one broadcasting process and more than one listeners)

b. Telephonic conversation (between a caller and a receiver)

c. Reader-Writer process (Writer writing in a buffer that is read by the Reader)

a. CODE SNIPPET :

Broadcaster :

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#define BUFFER_SIZE 1024
int main()
{
	int fd;
	char * myfifo = "/tmp/myfifo";
	mkfifo(myfifo, 0666);
	char buffer[BUFFER_SIZE], dummy;
	while (1){
		fd = open(myfifo, O_WRONLY);
		printf("Enter Message: ");
		scanf("%[^\n]s",buffer);
		scanf("%c",&dummy);
		write(fd, buffer, strlen(buffer)+1);
		close(fd);
	}
	return 0;
}
```

Listener :

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#define BUFFER_SIZE 1024
int main()
{
	int fd;
	char * myfifo = "/tmp/myfifo";
	mkfifo(myfifo, 0666);
	char buffer[BUFFER_SIZE], dummy;
	while (1){
		fd = open(myfifo, O_RDONLY);
		read(fd, str, strlen(buffer)+1);
		printf("%s\n", str);
		close(fd);
	}
	return 0;
}
```

EXPLANATION :

The problem states that there is one broadcaster which will broadcast a message, and on the other side, there are more than one broadcaster which will act as the receivers of broadcasted message. To implement this IPC mechanism, FIFO can be used.

In the broadcaster code, a new FIFO or named pipe is created or the existing FIFO is opened. We use system calls associated with it to store and retrieve information. The information is in character array format. The FIFO is first opened, and then a character array is entered by the user. The information, which is the message to be broadcasted, is stored in the FIFO, and then the FIFO is closed.

In the receiver code, the existing FIFO is opened for retrieving the message broadcasted by the broadcaster. The message is then printed. After printing the message, we close the FIFO by making the call close().

b. CODE SNIPPET (Caller - receiver):

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<string.h>
#include<sys/wait.h>

int main(){
      int fd1[2]; // Used to store two ends of first pipe
      int fd2[2]; // Used to store two ends of second pipe
      char input_str[100];
      pid_t p;

      if (pipe(fd1)==-1){
            fprintf(stderr, "Pipe Failed" );
            return 1;
      }
      if (pipe(fd2)==-1){
            fprintf(stderr, "Pipe Failed" );
            return 1;
      }
      scanf("%s", input_str);
      p = fork();
      if (p < 0){
            fprintf(stderr, "fork Failed" );
            return 1;
      }
      else if (p > 0){
            char received_string[100];
            close(fd1[0]);
            write(fd1[1], input_str, strlen(input_str)+1);
            close(fd1[1]);
            // Wait for child to send a string
            wait(NULL);
            close(fd2[1]);
            read(fd2[0], received_string, 100);
            printf("Recieved String : %s\n", received_string);
            close(fd2[0]);
      }
```

```
      else{
            close(fd1[1]);
            char received_string[100];
            read(fd1[0], received_string, 100);
            int k = strlen(received_string);
            int i;
            close(fd1[0]);
            close(fd2[0]);
            write(fd2[1], received_string, strlen(received_string)+1);
            close(fd2[1]);
            exit(0);
      }
}
```

EXPLANATION :

In the telephonic conversation one person will talk from one end and another person will listen. After that the second person will talk and the first person will listen. They both cannot talk simultaneously. To implement this inter process communication pipe is used. Pipe is a technique for passing information from one program process to another. Unlike other forms of inter-process communication (IPC), a pipe is oneway communication only. Basically, a pipe passes a parameter such as the output of one process to another process which accepts it as input. The system temporarily holds the piped information until it is read by the receiving process.
In the above file, a pipe is created in the main function. Basic operations on a pipe is used to read and write in the pipe.

c. CODE SNIPPET :

Writer :

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct mesg_buffer {
      long mesg_type;
      char mesg_text[100];
} message;
int main()
{
      key_t key;
      int msgid;
      key = ftok("progfile", 65); r
      msgid = msgget(key, 0666 | IPC_CREAT);
      message.mesg_type = 1;
      printf("Write Data : ");
      gets(message.mesg_text);
      msgsnd(msgid, &message, sizeof(message), 0);
      printf("Data send is : %s \n", message.mesg_text);
      return 0;
}
```

Reader :

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct mesg_buffer
{
      long mesg_type;
      char mesg_text[100];
} message;
int main()
{
       key_t key;
       int msgid;
       key = ftok("progfile", 65);
       msgid = msgget(key, 0666 | IPC_CREAT);
       msgrcv(msgid, &message, sizeof(message), 1, 0);
       printf("Data Received is : %s \n", message.mesg_text);
       msgctl(msgid, IPC_RMID, NULL);
       return 0;
}
```

EXPLANATION :

A Reader-Writer Problem is a situation where a file can be read and modified simultaneously by concurrent threads. Only one Writer is allowed to access the critical area at any moment in time. When no Writer is active any number of Readers can access the critical area but cannot edit the critical area. To implement this type of inter process communication message queue is used.

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created or an existing queue opened by msgget(). New messages are added to the end of a queue by msgsnd(). Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd() when the message is added to a queue. Messages are fetched from a queue by msgrcv(). There are two files – Writer and Reader. In writer file, the method to write in a message queue is implemented. The method to read from a message queue is implemented in the reader file.

ASSIGNMENT - 3

2. Considering the following scenario, accommodate the processes Pj (150 bytes), Pj (100 bytes) and Pk (150 bytes) respectively (arriving consecutively) using (i) first fit and (ii) best fit strategy.

Name	Size (bytes)	Starting memory address
P0	500	200
P1	400	1000
P2	250	1600
P3	800	2000

Assume that each memory location is capable of holding only 1 byte and the first location in memory is 0001 and the last address in memory is 9999.
Your output should show the following:
The entire memory map including the location and size of each process and hole:

i. After P0 to P3 are accommodated,
ii. After Pi is accommodated,
iii. After Pj is accommodated and
iv. After Pk is accommodated

CODE SNIPPET :

```
#include <bits/stdc++.h>
using namespace std;
vector<int> memory;
vector<vector<int> > curblocks;
void displaytable() {
    curblocks.clear();
    int start = 1;
    int curprocess = memory[1];
    for(int i=1;i<10000;i++) {
        if(curprocess != memory[i]) {
            curblocks.push_back({curprocess, start, i-1, i-start});
            curprocess = memory[i];
            start = i;
        }
    }
    curblocks.push_back({curprocess, start, 9999, 10000-start});
    cout<<"Current memory map : \n";
    cout<<"------------------------------------------------------------------------
--------------------\n";
    cout<<"Block no.\t Status\t\t Process no.\t Start address\t End address\t
Size\n";
    cout<<"------------------------------------------------------------------------
--------------------\n";
    int i = 1;
    char status[15];
    for(auto it : curblocks) {
        if(it[0] == -1) {
            strcpy(status, "Unallocated");
            printf("%d\t\t %s\t \t\t %d\t\t %d\t\t %d\n", i, status, it[1],
it[2], it[3]);
        }
```

```
        else {
            strcpy(status, "Allocated");
            printf("%d\t\t %s\t %d\t\t %d\t\t %d\t\t %d\n", i, status, it[0],
it[1], it[2], it[3]);
        }
        i++;
    }
    cout<<"-------------------------------------------------------------------
--------------------\n\n";
}
void allocate_process(int *p_start, int *p_size, int total) {
    for(int i=0;i<total;i++) {
        for(int j=p_start[i];j<p_start[i]+p_size[i];j++) {
            memory[j] = i;
        }
    }
}
void firstfit(int curprocess, int size) {
    for(auto it : curblocks) {
        if(it[0]==-1 && it[3] >= size) {
            for(int i=it[1];i<it[1]+size;i++) {
                memory[i] = curprocess;
            }
            return;
        }
    }
}
bool compare(vector<int> x, vector<int> y) {
    if(x[3] < y[3]) return 1;
    return 0;
}
void bestfit(int curprocess, int size) {
    sort(curblocks.begin(), curblocks.end(), compare);
    for(auto it : curblocks) {
        if(it[0]==-1 && it[3] >= size) {
            for(int i=it[1];i<it[1]+size;i++) {
                memory[i] = curprocess;
            }
            return;
        }
    }
}
int main() {
     int p_start[] = { 200 , 1000 , 1600 , 2000 };
     int p_size[] = { 500 , 400 , 250 , 800 };
     int process[] = {150, 100, 150};
     cout<<"_____________________________________________________________________
_____________________________\n";
     cout<<"\t\t\t\t FIRST FIT STRATEGY \n";
     cout<<"_____________________________________________________________________
_____________________________\n";
     memory.resize(10000, -1);
     displaytable();

     allocate_process(p_start, p_size, sizeof(p_size)/sizeof(int));
     printf("After allocating 4 given processes using first fit strategy, ");
     displaytable();

     for(int i=0;i<3;i++) {
         firstfit(i+10, process[i]);
```

```
        printf("After allocating process %d with size %d bytes using first
fit strategy, ", i+10, process[i]);
        displaytable();
    }

    cout<<"_____________________________________________________________________
_____________________\n";
    cout<<"\t\t\t\t BEST FIT STRATEGY \n";

    cout<<"_____________________________________________________________________
_____________________\n";

    memory.clear();
    memory.resize(10000, -1);
    displaytable();

    allocate_process(p_start, p_size, sizeof(p_size)/sizeof(int));
    printf("After allocating 4 given processes using best fit strategy, ");
    displaytable();

    for(int i=0;i<3;i++) {
        bestfit(i+10, process[i]);
        printf("After allocating process %d with size %d bytes using best fit
strategy, ", i+10, process[i]);
        displaytable();
    }
    return 0;
}
```

EXPLANATION :

```
vector<int> memory;
vector<vector<int> > curblocks;
memory.resize(10000, -1);
```

A dynamic array, vector, is used to store the process number. The size of the memory is set to 10000, and the initial value in them is set as -1 (denoting that the indexed memory is unallocated. In curblocks, the current state of memory map is stored and updated. Each block is denoted as the vector of size 4. 1st index of the vector stores the process number. Process number is -1 if the block is unallocated. 2nd and 3rd index of the vector stores the starting address and the ending address of the partition; and 4th index stores size of the partition.

```
void displaytable();
```

The above function is used to update the current partitions in the memory map. The current partitions is then displayed in tabular format, showing the block number status (allocated or unallocated), process number, starting address, end address and size of the partition.

```
void allocate_process(int *p_start, int *p_size, int total)
```

Firstly, the allocate_process() is called to process the given 4 process having the starting address and the size of process. We can simply do it by assigning the corresponding process number to the memory array of size 10000.

```
void firstfit(int curprocess, int size)
```

The first fit strategy is used to accommodate the process to a hole (unallocated partition). The process is allocated to the first hole having its size equal to or greater than the process size.

```
void bestfit(int curprocess, int size)
```

In best fit strategy, the process is accommodated to the “smallest” hole having size greater or equal to the process size.

Next, the 3 processes i,j,k (I have used their names as 10,11,12) are allocated to the holes initially with first fit strategy, and then using best fit strategy. After allocating each process, the current memory map is displayed.

OUTPUT :

```
_____________________________________________________________________________________________
                                FIRST FIT STRATEGY
_____________________________________________________________________________________________
Current memory map :
-----------------------------------------------------------------------------------------
Block no.       Status          Process no.     Start address   End address     Size
-----------------------------------------------------------------------------------------
1               Unallocated                     1               9999            9999
-----------------------------------------------------------------------------------------

After allocating 4 given processes using first fit strategy, Current memory map :
-----------------------------------------------------------------------------------------
Block no.       Status          Process no.     Start address   End address     Size
-----------------------------------------------------------------------------------------
1               Unallocated                     1               199             199
2               Allocated       0               200             699             500
3               Unallocated                     700             999             300
4               Allocated       1               1000            1399            400
5               Unallocated                     1400            1599            200
6               Allocated       2               1600            1849            250
7               Unallocated                     1850            1999            150
8               Allocated       3               2000            2799            800
9               Unallocated                     2800            9999            7200
-----------------------------------------------------------------------------------------

After allocating process 10 with size 150 bytes using first fit strategy, Current memory map :
-----------------------------------------------------------------------------------------
Block no.       Status          Process no.     Start address   End address     Size
-----------------------------------------------------------------------------------------
1               Allocated       10              1               150             150
2               Unallocated                     151             199             49
3               Allocated       0               200             699             500
4               Unallocated                     700             999             300
5               Allocated       1               1000            1399            400
6               Unallocated                     1400            1599            200
7               Allocated       2               1600            1849            250
8               Unallocated                     1850            1999            150
9               Allocated       3               2000            2799            800
10              Unallocated                     2800            9999            7200
-----------------------------------------------------------------------------------------

After allocating process 11 with size 100 bytes using first fit strategy, Current memory map :
-----------------------------------------------------------------------------------------
Block no.       Status          Process no.     Start address   End address     Size
-----------------------------------------------------------------------------------------
1               Allocated       10              1               150             150
2               Unallocated                     151             199             49
3               Allocated       0               200             699             500
4               Allocated       11              700             799             100
5               Unallocated                     800             999             200
6               Allocated       1               1000            1399            400
7               Unallocated                     1400            1599            200
8               Allocated       2               1600            1849            250
9               Unallocated                     1850            1999            150
10              Allocated       3               2000            2799            800
11              Unallocated                     2800            9999            7200
-----------------------------------------------------------------------------------------

After allocating process 12 with size 150 bytes using first fit strategy, Current memory map :
-----------------------------------------------------------------------------------------
Block no.       Status          Process no.     Start address   End address     Size
-----------------------------------------------------------------------------------------
1               Allocated       10              1               150             150
2               Unallocated                     151             199             49
3               Allocated       0               200             699             500
4               Allocated       11              700             799             100
5               Allocated       12              800             949             150
6               Unallocated                     950             999             50
7               Allocated       1               1000            1399            400
```

```
8               Unallocated                     1400            1599            200
9               Allocated       2               1600            1849            250
10              Unallocated                     1850            1999            150
11              Allocated       3               2000            2799            800
12              Unallocated                     2800            9999            7200
-------------------------------------------------------------------------------------
```

BEST FIT STRATEGY

Current memory map :

Block no.	Status	Process no.	Start address	End address	Size
1	Unallocated		1	9999	9999

After allocating 4 given processes using best fit strategy, Current memory map :

Block no.	Status	Process no.	Start address	End address	Size
1	Unallocated		1	199	199
2	Allocated	0	200	699	500
3	Unallocated		700	999	300
4	Allocated	1	1000	1399	400
5	Unallocated		1400	1599	200
6	Allocated	2	1600	1849	250
7	Unallocated		1850	1999	150
8	Allocated	3	2000	2799	800
9	Unallocated		2800	9999	7200

After allocating process 10 with size 150 bytes using best fit strategy, Current memory map :

Block no.	Status	Process no.	Start address	End address	Size
1	Unallocated		1	199	199
2	Allocated	0	200	699	500
3	Unallocated		700	999	300
4	Allocated	1	1000	1399	400
5	Unallocated		1400	1599	200
6	Allocated	2	1600	1849	250
7	Allocated	10	1850	1999	150
8	Allocated	3	2000	2799	800
9	Unallocated		2800	9999	7200

After allocating process 11 with size 100 bytes using best fit strategy, Current memory map :

Block no.	Status	Process no.	Start address	End address	Size
1	Allocated	11	1	100	100
2	Unallocated		101	199	99
3	Allocated	0	200	699	500
4	Unallocated		700	999	300
5	Allocated	1	1000	1399	400
6	Unallocated		1400	1599	200
7	Allocated	2	1600	1849	250
8	Allocated	10	1850	1999	150
9	Allocated	3	2000	2799	800
10	Unallocated		2800	9999	7200

After allocating process 12 with size 150 bytes using best fit strategy, Current memory map :

Block no.	Status	Process no.	Start address	End address	Size
1	Allocated	11	1	100	100
2	Unallocated		101	199	99
3	Allocated	0	200	699	500
4	Unallocated		700	999	300
5	Allocated	1	1000	1399	400
6	Allocated	12	1400	1549	150
7	Unallocated		1550	1599	50
8	Allocated	2	1600	1849	250
9	Allocated	10	1850	1999	150
10	Allocated	3	2000	2799	800
11	Unallocated		2800	9999	7200

3. Write a program to simulate Least Recently Used (LRU) page replacement algorithm for the following page reference string: 9, 10, 11, 7, 12, 8, 7, 6, 12, 5, 4, 3, 10, 11, 12, 4, 5, 6, 9, 4, 5.
Consider (i) 4 frames and (ii) 5 frames. Compare the results.

CODE SNIPPET :

```
#include<bits/stdc++.h>
using namespace std;

int LRU_pagefaults(int pages[], int n, int frames) {
      set<int> st;
      list<int> l;
      bool flag;
      int faults = 0;
      for(int i=0;i<n;i++) {
            if(st.find(pages[i]) == st.end()) {
                  if(l.size() >= frames) {
                        st.erase(st.find(l.front()));
                        l.pop_front();
                  }
                  l.push_back(pages[i]);
                  st.insert(pages[i]);
                  faults++;
                  flag = 1;
            }
            else {
                  l.remove(pages[i]);
                  l.push_back(pages[i]);
                  flag = 0;
            }
            printf("After inserting %2d, current list is: ", pages[i]);
            for(auto it : l) cout<<it<<" ";
            if(flag) cout<<" -- Page fault\n";
            else cout<<" -- Page hit\n";
      }
      return faults;
}

int main() {
      int pages[] = { 9,10,11,7,12,8,7,6,12,5,4,3,10,11,12,4,5,6,9,4,5 };
      int n = sizeof(pages) / sizeof(pages[0]);

      cout<<"With 4 frames : \n";
      int fault1 = LRU_pagefaults(pages, n, 4);
      int hit1 = n - fault1;
      printf("Total page faults: %2d\t Total page hits: %2d\n\n", fault1,
hit1);

      cout<<"With 5 frames : \n";
      int fault2 = LRU_pagefaults(pages, n, 5);
      int hit2 = n - fault2;
      printf("Total page faults: %2d\t Total page hits: %2d\n\n", fault2,
hit2);

      return 0;
}
```

EXPLANATION:

In the above program, LRU_pagefaults() function is used to compute the number of page faults after processing all the pages stored in the reference string.

In LRU_pagefaults() function, the reference string, which is named as pages, is an array of integers stored the page numbers. N is the total length of the reference string. Frames is the number of frames that can contain the pages, at the moment of time.

Set is used to check the existence of the page in the frame list, as using set we can directly check the existence using the find(value) function.

List is used as the frame list to store the pages in it. Until the size of the list is less than the number of frames, the unique element is added to the end of the list. When the size is equal or exceeds the number of frames, the front element is removed from the list and set when there is page fault. When there is page hit, the element is removed from the list, and appended at end of the list.

The number of page faults with 4 frames is 17, whereas with 5 frames, the number of page faults is 16. On comparing the results, we can clearly find that on increasing number of frames, number of page faults decreases.

OUTPUT :

```
With 4 frames :
After inserting  9, current list is: 9  -- Page fault
After inserting 10, current list is: 9 10  -- Page fault
After inserting 11, current list is: 9 10 11  -- Page fault
After inserting  7, current list is: 9 10 11 7  -- Page fault
After inserting 12, current list is: 10 11 7 12  -- Page fault
After inserting  8, current list is: 11 7 12 8  -- Page fault
After inserting  7, current list is: 11 12 8 7  -- Page hit
After inserting  6, current list is: 12 8 7 6  -- Page fault
After inserting 12, current list is: 8 7 6 12  -- Page hit
After inserting  5, current list is: 7 6 12 5  -- Page fault
After inserting  4, current list is: 6 12 5 4  -- Page fault
After inserting  3, current list is: 12 5 4 3  -- Page fault
After inserting 10, current list is: 5 4 3 10  -- Page fault
After inserting 11, current list is: 4 3 10 11  -- Page fault
After inserting 12, current list is: 3 10 11 12  -- Page fault
After inserting  4, current list is: 10 11 12 4  -- Page fault
After inserting  5, current list is: 11 12 4 5  -- Page fault
After inserting  6, current list is: 12 4 5 6  -- Page fault
After inserting  9, current list is: 4 5 6 9  -- Page fault
After inserting  4, current list is: 5 6 9 4  -- Page hit
After inserting  5, current list is: 6 9 4 5  -- Page hit
Total page faults: 17    Total page hits:  4

With 5 frames :
After inserting  9, current list is: 9  -- Page fault
After inserting 10, current list is: 9 10  -- Page fault
After inserting 11, current list is: 9 10 11  -- Page fault
After inserting  7, current list is: 9 10 11 7  -- Page fault
After inserting 12, current list is: 9 10 11 7 12  -- Page fault
After inserting  8, current list is: 10 11 7 12 8  -- Page fault
After inserting  7, current list is: 10 11 12 8 7  -- Page hit
After inserting  6, current list is: 11 12 8 7 6  -- Page fault
After inserting 12, current list is: 11 8 7 6 12  -- Page hit
After inserting  5, current list is: 8 7 6 12 5  -- Page fault
After inserting  4, current list is: 7 6 12 5 4  -- Page fault
After inserting  3, current list is: 6 12 5 4 3  -- Page fault
After inserting 10, current list is: 12 5 4 3 10  -- Page fault
After inserting 11, current list is: 5 4 3 10 11  -- Page fault
After inserting 12, current list is: 4 3 10 11 12  -- Page fault
After inserting  4, current list is: 3 10 11 12 4  -- Page hit
After inserting  5, current list is: 10 11 12 4 5  -- Page fault
After inserting  6, current list is: 11 12 4 5 6  -- Page fault
After inserting  9, current list is: 12 4 5 6 9  -- Page fault
After inserting  4, current list is: 12 5 6 9 4  -- Page hit
After inserting  5, current list is: 12 6 9 4 5  -- Page hit
Total page faults: 16    Total page hits:  5
```