Table of contents

SECTION I

Operating System -I

ASSIGNMENT 1: Operations on Processes

Process: A process is basically a program in execution. We write our computer programs in a text file, during execution it is loaded in computer's memory and then it becomes a process. A process performs all the tasks mentioned in the program. A process can be divided into four sections ─ stack, heap, text and data.

Stack: The process Stack contains the temporary data such as method/function parameters, return address and local variables.

Heap: This is dynamically allocated memory to a process during its run time.

Text: This includes all instructions specified in the program.

Data: This section contains the global and static variables.

In Linux operating system, new processes are created through fork () system call**.**

Fork() System Call:

System call fork () is used to create new processes. It takes no arguments and returns a process ID. The newly created process is called child process and the caller process is called as parent process. After a new child process is created, both parent and child processes will execute the next instruction following the fork () system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork().

Fork () returns a zero to the newly created child process and returns a positive value (process ID of the child process) to the parent.

If fork() returns a negative value, the creation of a child process was unsuccessful.

The returned process ID is of type pid_t defined in sys/types.h. Normally, the process ID is an integer. Moreover, a process can use function getpid() to retrieve the process ID assigned to this process.

Let us take the following example:

int main ()

```
{   printf("Before Forking"); fork();
    printf("After Forking");
   return 0;
}
```

If the call to fork () is executed successfully, Linux will

- Make two identical copies of address spaces, one for the parent and the other for the child.
- Both processes will start their execution at the next statement following the fork() call.

Output of above program:

Before Forking

After Forking

After Forking

Here printf() statement after fork() system call executed by parent as well as child process. Both processes start their execution right after the system call fork(). Since both processes have identical but separate address spaces, those variables initialized before the fork() call have the same values in both address spaces. Since every process has its own address space, any modifications will be independent of the others. In other words, if the parent changes the value of its variable, the modification will only affect the variable in the parent process's address space. Other address spaces created by fork() calls will not be affected even though they have identical variable names.

Consider one simpler example that distinguishes the parent from the child.

```
#include <stdio.h>
#include <sys/types.h>
void ChildProcess(); /* child process prototype */
void ParentProcess(); /* parent process prototype */
int main()
{ pid_t pid;
  pid = fork();
 if (pid == 0)
        ChildProcess();
```

```
    else
        ParentProcess();
    return 0;
}
void ChildProcess()
{ printf("I am child process..");
}
void ParentProcess()
{ printf("I am parent process..");
}
```

exec() system call

The exec() family of functions **replaces** the current process image with a new process image. It loads the program into the current process space and runs it from the entry point. The current process is just turned into a new process and hence the process id PID is not changed, this is because we are not creating a new process we are just replacing a process with another process in exec.

The exec() family consists of following functions,

execl(): l is for the command line arguments passed a list to the function.

*int execl(const char *path, const char *arg, ...);*

execlp(): p is the path environment variable which helps to find the file passed as an argument to be loaded into process.

*int execlp(const char *file, const char *arg, ...);*

execle(): It is an array of pointers that points to environment variables and is passed explicitly to the newly loaded process.

*int execle(const char *path, const char *arg, ..., char * const envp[]);*

execv(): v is for the command line arguments. These are passed as an array of pointers to the function.

*int execv(const char *path, char *const argv[]);*

execlp() System Call:

execlp() system call is used after a fork() call by one of the two processes to replace

the processes memory space with a new program. This call loads a binary file into memory and starts its execution. So two processes can be easily communicating with each other.

```
#include<sys/types.h>
 #include<stdio.h>
 #include<unistd.h>
int main()
{
        int pid;
        pid = fork(); /* fork a child
        process */ if (pid< 0) /* error
        occurred */
        { fprintf(stderr, "Fork Failed");
          return 1;
        }
        else if (pid == 0)   /* child process */
                execlp("/bin/wc","wc",NULL);
            else /* parent process */
                { wait(NULL);   /* parent will wait for the child to complete */
                  printf("Child Complete");
              }
   return 0;
}
```

fork() vs exec()

- Fork() starts a new process which is a copy of the one that calls it, while exec() replaces the current process image with another (different) one.
- Both parent and child processes are executed simultaneously in case of fork() while Control never returns to the original program unless there is an exec() error.

Wait() system call

The **wait**() system call suspends execution of the calling process until one of its children terminates.

wait(int &status);

waitpid() system call : By using this system call it is possible for parent process to synchronize its execution with child process. Kernel will block the execution of a Process that calls waitpid system call if a some child of that process is in execution. It returns immediately when child has terminated by

return termination status of a child.

*int waitpid(int pid, int *status, int options);*

nice() System Call:

Using nice() system call, we can change the priority of the process in multi-tasking system. The new priority number is added to the already existing value.

int nice(int inc);

nice() adds *inc* to the nice value. A higher nice value means a lower priority. The range of the nice value is +19 (low priority) to -20 (high priority).

```
#include<stdio.h>
 main()
{
        int pid, retnice;
        printf("press DEL to stop process \n");
        pid=fork();
        for(;;)
        {
          if(pid == 0)
          {
            retnice = nice (−5);
            print("child gets higher CPU priority %d \n", retnice);
           sleep(1);
          }
        else
        {
           retnice=nice(4);
          print("Parent gets lower CPU priority %d \n", retnice);
         sleep(1);
        }
      }
}
```

Orphan process

The child processes whose parent process has completed execution or terminated are called orphan process. Usually, a parent process waits for its child to terminate or finish their

job and report to it after execution but if parent fails to do so its child results in the Orphan process. In most cases, the Orphan process is immediately adopted by the init process (a very first process of the system).

Practical Assignments:

Set A

(1) Implement the C Program to create a child process using fork (), display parent and child process id. Child process will display the message "I am Child Process" and the parent process should display "I am Parent Process".

(2) Write a program that demonstrates the use of nice () system call. After a child process is started using fork (), assign higher priority to the child using nice () system call.

Set B

(1) Implement the C program to accept n integers to be sorted. Main function creates child process using fork system call. Parent process sorts the integers using bubble sort and waits for child process using wait system call. Child process sorts the integers using insertion sort.

(2) Write a C program to illustrate the concept of orphan process. Parent process creates a child and terminates before child has finished its task. So child process becomes orphan process. (Use fork (), sleep (), getpid (), getppid ()).

Set C

(1) Implement the C program that accepts an integer array. Main function forks child process. Parent process sorts an integer array and passes the sorted array to child process through the command line arguments of execve() system call. The child process uses execve() system call to load new program that uses this sorted array for performing the binary search to search the particular item in the array.

(2) Implement the C Program to create a child process using fork (), Using exec () system call, child process will execute the program specified in Set A (1) and parent will continue by printing message "I am parent ".

Assignment Evaluation

0: Not Done [ ] 1: Incomplete [ ] 2: Late Complete [ ]

3: Needs Improvement [ ] 4: Complete [ ] 5: Well done [ ]

Signature of the instructor:_____Date: _____

ASSIGNMENT 2: Operations on Processes

What is Shell?

Shell is an interface between user and operating system. It is the **command interpreter**, which accept the command name (program name) from user and executes that command/program. Shell mostly accepts the commands given by user from keyboard. Shell gets started automatically when Operating system is successfully started.

When shell is started successfully it generally display some prompt (such as #, $ etc) to indicate the user that it is ready to accept and execute the command.

Shell executes the commands either **synchronously** or **asynchronously.**

When shell accepts the command then it locates the program file for that command, start its execution, wait for the program associated with the command to complete its execution and then display the prompt again to accept further command. This is called as Synchronous execution of shell.

In asynchronous execution shell accept the command from user, start the execution of program associated to the given command but does not wait for that program to finish its execution, display prompt to accept next command.

How Shell Execute the command?

1. Accept the command from user.
2. Tokenize the different parts of command.
3. First part given on the given command line is always a command name.
4. **Creates (Forks)** a child process for executing the program associated with given command.
5. Once child process is created successfully then it **loads (exec)** the binary (executable) image of given program in child process area.
6. Once the child process is loaded with given program it will start its execution while shell is **waiting (wait)** for it (child) to complete the execution. Shell will wait until child finish its execution.
7. Once child finish the execution then Shell wakeup, display the command prompt again and accept the command and continue.

Example

$ cat f1.dat f2.dat f3.dat

This command line has four tokens- cat, f1.dat, f2.dat and f3.dat. First token is the command name which is to be executed. In linux operating system, some commands are internally coded and implemented by shell such as mkdir, rmdir, cd, pwd, ls, cat, grep, who etc.

Objective of this practical assignment is to simulate the shell which interprets all internal or predefined Linux commands and additionally implement to interpret following extended commands.

(1) count: To count and display the number of lines, words and characters in a given file.

(2) typeline: It will display the all or number of lines in given file.

(3) List: It will list the files in current directory with some details of files

(4) Search: It will allow searching the file for the occurrence of given string/pattern

Program Logic

1) Main function will execute and display the command prompt as $

2) Accept the command at $ prompt from user.

3) Separate or tokenize the different parts of command line.

4) Check that first part is one of the extended commands or not (count, typeline, list, search).

5)If the command is extended command then call corresponding functions which is implementing that command

6) Otherwise fork a new process and then load (exec) a program in that newly created process and execute it. Make the shell to wait until command finish its execution.

7) Display the prompt and continue until given command is “q” to Quit.

Here's the example for tokenizing the given input string:

```
// Online C compiler to run C program online
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

// Function to split the input command into tokens
char** parsetoken(char *tok)
{    int i = 0;
    char *temp = strtok(tok, " \n"); // handles newline as well
    char **tokens = malloc(10 * sizeof(char *)); // up to 10 tokens for safety
    while (temp != NULL)
    {  tokens[i++] = temp;
        temp = strtok(NULL, " \n");
    }
    tokens[i] = NULL; // Null terminate for execvp
    return tokens;
}
// Function to count characters in a file
int count(char *filename, char *arg)
{    FILE *fp;
    char ch;
    int ccnt = 0;
```

```
    fp = fopen(filename, "r");
    if (fp == NULL) {
        perror("Error opening file");
        return 1;
    }
     while ((ch = fgetc(fp)) != EOF)
     {        ccnt++;
    }
    printf("Character count: %d\n", ccnt);
    fclose(fp);
    return 0;
}
int main()
{   char buff[80];
    char **tokens;
    int pid;
    printf("myshell$ ");
    fflush(stdout);
    fgets(buff, sizeof(buff), stdin);
    tokens = parsetoken(buff);
    if (tokens[0] == NULL)
   {   printf("No command entered.\n");
        return 1;
    }
     pid = fork();
```

```
   if (pid == 0)
{
// In child process
```

```
      if (strcmp(tokens[0], "count") == 0)
```

```
    {       if (tokens[1] == NULL || tokens[2] == NULL)
            {     printf("Usage: count <arg> <filename>\n");
                  exit(1);
            }
         count(tokens[2], tokens[1]);
     }
    else {
          // Execute external command
          execvp(tokens[0], tokens);
          perror("execvp failed");
          exit(1);
        }
   }
    else if (pid > 0) {
            // In parent process
              wait(NULL);
         } else {
              perror("fork failed");
              return 1;
  }
  free(tokens);
  return 0;
```

```
}
```

Above program tokenize the command line input using Myshell prompt and executes the command by creating a child process, and implement the "count" command.

Output: Myshell$ count c file.txt

Character count 128

Practical Assignments:

Set A

Write a C program that behaves like a shell which displays the command prompt '**myshell$**'. It accepts the command, tokenize the command line and execute it by creating the child process.
Also implement the additional command 'count' as:
myshell$ count c filename: It will display the number of characters in given file
myshell$ count w filename: It will display the number of words in given file
myshell$ count l filename: It will display the number of lines in given file

Set B

Write a C program that behaves like a shell which displays the command prompt '**myshell$**'. It accepts the command, tokenize the command line and execute it by creating the child process.
Also implement the additional command 'list' as:
myshell$ list f dirname: It will display filenames in a given directory.
myshell$ list n dirname: It will count the number of entries in a given directory.
myshell$ list i dirname: It will display filenames and their inode number for the files in a given directory.

Set C

(1) Write a C program that behaves like a shell which displays the command prompt '**myshell$**'. It accepts the command, tokenize the command line and execute it by creating the child process.
Also implement the additional command 'typeline' as:
myshell$ typeline n filename: It will display first n lines of the file.
myshell$ typeline -n filename: It will display last n lines of the file.
myshell$ typeline a filename: It will display all the lines of the file.

(2) Write a C program that behaves like a shell which displays the command prompt '**myshell$**'. It accepts the command, tokenize the command line and execute it by creating the child process.Also implement the additional command 'search' as:

myshell$ search f filename pattern: It will search the first occurrence of pattern in the given file.

myshell$ search a filename pattern: It will search all the occurrence of pattern in the given file.

myshell$ search c filename pattern: It will count the number of occurrence of pattern in the given file.

Assignment Evaluation

0: Not Done [ ] 1: Incomplete [ ] 2: Late Complete [ ]

3: Needs Improvement [ ] 4: Complete [ ] 5: Well done [ ]

Signature of the instructor:_____Date: _____

Assignment 3- CPU Scheduling

CPU Scheduling is one of the important tasks performed by the operating system. In multiprogramming operating systems many processes are loaded into memory for execution and these processes are sharing the CPU and other resources of computer system.

Scheduler is a program that decides which program will execute next at the CPU or which program will be loaded into memory for execution.

CPU scheduler is a program module of an operating system that selects the process to execute next at CPU out of the processes that are in memory. This scheduler is also called as Short term scheduler or CPU Scheduler

The main objective is to increase system performance in accordance with the chosen set of criteria.

There are 3 types of schedulers as

1) Short Term Scheduler

2) Long Term Scheduler

3) Medium Term Scheduler

We have to implement various Short Term Scheduling algorithms as a part of this practical assignment.

Various CPU Scheduling algorithms are:

1) First Come First Serve (FCFS)

2) Shortest Job First (SJF)

3) Priority Scheduling

4) Round Robin Scheduling (RR)

These scheduling algorithms are further classified into 2 types as **Preemptive** and **Non-Preemptive**. FCFS scheduling is always Non-Preemptive while Round Robin is always Preemptive, while Shortest Job First and Priority Scheduling can be preemptive or non-preemptive. The performance of various scheduling algorithms is compared on the basis of following criteria called as **scheduling criteria's** as:

1) CPU Utilization 2) Throughput 3) Turnaround Time 4) Waiting Time

5) Response Time

Data Structures

To simulate the working of various CPU scheduling algorithms following data structures is required.

1)Ready Queue - It represents the queue of processes which ready to execute but waiting for CPU to become available. Scheduling algorithm will select appropriate process from the ready queue and dispatch it for execution. For FCFS and Round Robin this queue is strictly operated as **First In First out queue**. But for Shortest Job First and Priority Scheduling it will operate as **Priority Queue**.

2)Process Control Block (PCB) - It will maintain various details about the each process as processID, CPU-Burst, Arrival time, waiting time, completion time, execution time, turnaround time, etc. A structure can be used to define all these fields for processes and we can use array of structures of size n. (n is number of processes)

1) First Come First Serve Scheduling (FCFS):

In this algorithm the order in which process enters the ready queue, in same order they will execute on the CPU. This algorithm is simple to implement but it may be worst for several times.

DESCRIPTION:

To calculate the average waiting time using the FCFS algorithm first the waiting time of the first process is kept zero and the waiting time of the second process is the burst time of the first process and the waiting time of the third process is the sum of the burst times of the first and the second process and so on. After calculating all the waiting times the average waiting time is calculated as the average of all the waiting times. FCFS mainly said that first come first serve the algorithm which came first will be served first.

Note: Variables used in the Algorithm

wt - Waiting time

tat- Turnaround time

tatavg- Average Turnaround time

wtavg- Average waiting time

qt- time quantum

rembt – remaining burst time

t- Current time

ALGORITHM:

1)First Come First Serve (FCFS)

Step 1: Start the process

Step 2: Define an array of structure PCB processcontrolblock of size n to store the arrival time, first

CPU burst time, next CPU burst time, turnaround time, and waiting time for each process.

```
struct PCB
{
    // structure members
} p[n];
```

Step 3: Input the number of processes (n) with its arrival time and burst time. Using function void input (), accept the AT and BT for n processes.

Step 4: Display the inputted values required for processes from the Ready Queue.

Step 5: Sorting of processes will be done according to arrival time //Using any sorting technique.
// goto step 4(For displaying the sorted processes)

Step 6: After sorting the processes, to create gantt chart for FCFS algorithm following steps are Applied.

For each process i from 0 to n-1, do:
Calculate Start_time of each process depending on the Arrival_time and the Burst_time of each process.
Find waiting time (wt) for all processes.

// As first process that comes need not to wait so waiting time for process 1 will be 0 i.e. wt [0] = 0.
// For Other processes: Calculate wait time = start time – Arrival time
// For average WT calculate Avg_waitTime= (sum of wait time of all processes)/n
Calculate Finish time = start time+ burst time
Calculate Turnaround time for all processes
TAT=Waiting Time + Burst Time
Avg_TAT= (Sum of TAT of all process)/n
// go to step 4
// display table structure with TAT and WT calculated

Step 7: For each process i from 0 to n-1, do:
Generate the next CPU burst randomly for each process and waiting time by 2 units.
Display the newly created burst time and waiting time (go to step 4)
// go to step 5 (sorting according to newly created arrival time)
// go to step 4
//display table structure with TAT and WT calculated.

Step 8: Stop

2) Shortest Job First (SJF):

In this algorithm the jobs will execute at the CPU according their next CPU-Burst time. The job in a ready queue which has shortest next CPU burst will execute next at the CPU. This algorithm can be preemptive or non-preemptive.

DESCRIPTION: (non-preemptive SJF)

To calculate the average waiting time in the shortest job first algorithm the sorting of the process based on their burst time in ascending order then calculate the waiting time of each process as the sum of the bursting times of all the process previous or before to that process.

ALGORITHM:

Step 1: Start the process

Step 2: Define an array of structure PCB processcontrolblock of size n to store the arrival time, first CPU Burst time, next CPU Burst time, Turnaround time, and Waiting time for each process.
struct PCB

```
{
    // structure members
} p[n];
```

Step3: Input the number of processes (n) with its arrival time and burst time and accept the Arrival time and Burst time for n processes.

Step 4: Display the inputted values required for processes from the Ready Queue

Step 5: Sorting of processes will be done according to arrival time

a) We will check that the arrival time of all the processes are different or not.

b) Select the process which have less Arrival time (or 0) will arrive first.

If two or more processes have arrival time 0, then select the process having minimum burst time (using bubble sort)

If all the processes come at the same time, then we don't need to sort the array on the basis of arrival time, here you may consider min burst time.

c) Keep on selecting the processes having min burst time from the ready queue and add those to gantt chart.

d) In case two or more processes have the same burst time then the process having arrival time less is selected.

Step 6: Now display the content of ready queue after sorting and also display the value from gantt chart.

Step 7: Calculate the start time of each process depending on the arrival time and the burst time of each process.

Step 8: Calculate the Waiting time using formula,

Wait time=Start Time – Arrival Time

Average_wt= (Add wt of all processes)/n (no of processes)

Step 9: Calculate the Finish time using formula,

Finish time=Start Time + Burst Time

Step 10: Calculate the Turnaround time using formula,

TAT = Waiting Time + Burst Time

Average_wt= (Add TAT of all processes)/n (no of processes)

Step 11: Generate the next CPU burst randomly for each process using the `rand` function. Generate random burst time and waiting time by 2 units. And display the newly created burst time and waiting time.

Step 12: Stop

3) Priority Scheduling (PS):

In this algorithm the job will execute according to their priority order. The job which has highest priority will execute first and the job which has least priority will execute last at the CPU. Priority scheduling can be preemptive or non-preemptive.

DESCRIPTION

To calculate the average waiting time in the priority algorithm, sort the burst times according to their priorities and then calculate the average waiting time of the processes. The waiting time of each process is obtained by summing up the burst times of all the previous processes.

ALGORITHM:

Step 1: Start the process

Step 2: Define an array of structure PCB processcontrolblock of size n to store the arrival time, first CPU burst time, next CPU burst time, turnaround time, and waiting time for each process.

```
struct PCB
{
    // structure members
} p[n];
```

Step3: Input the number of processes (n) with its arrival time and burst time. Using function void input (), accept the AT and BT for n processes.

Step 4: Display the inputted values required for processes from the Ready Queue

Step 5: Sort all processes on the basis of priority (Highest / lowest)

a) Select the process which have less Arrival time (or 0) will come first.

b) If two or more processes have arrival time 0, then select the process having highest priority from the ready queue.
c) If process with max priority is found, then for the first time CPU is getting to that process (start time= 0). And decrement the value of Burst time bt by 1 and continue till bt==0. And remove the process from ready queue.
d) Keep on selecting the processes having max priority and add those in gantt chart.
e) If priority is same then process with less arrival time is selected

Step 6: Now display the content of ready queue after sorting

Step 7: Calculate the Start time and Finish time using formula,
Start time = Current time
Finish time=start time + burst time

Step 8: Calculate the Waiting time for the pre-emptive process using formula,
Wait time= (Start time – Arrival Time) + (New Start Time – Old Finish Time)
Average_wt= (Add wt of all processes)/n (no of processes)

Step 9: Calculate the Turnaround time using formula,
TAT = Waiting Time + Burst Time
Average_wt= (Add TAT of all processes)/n (no of processes)

Step 10: Generate the next CPU burst randomly for each process using the `rand` function. Generate random burst time and waiting time by 2

Step 11: Stop

4) Round Robin Scheduling (RR):

This algorithm is mostly used in time-sharing operating systems like Unix/Linux. This algorithm gives the fair change of execution to each process in system in rotation. In this algorithm each process is allowed to execute for some fixed time quantum. If process has the CPU-burst more than time quantum, then it will execute for the given time quantum. But if it has CPU-burst less than the time quantum then it will execute for its CPU-burst time and then immediately release the CPU so that it can be assigned to other process. The advantage of this algorithm is that the average waiting time is less. This algorithm use/ ready queue and is strictly operated as First In First Out queue. This algorithm is intrinsically preemptive scheduling algorithm.

DESCRIPTION:

To calculate the average waiting time. There will be a time slice, each process should be executed within that time-slice and if not it will go to the waiting state so first check whether the burst time is less than the time-slice. If it is less than it assigns the waiting time to the sum of the total times. If it is greater than the burst-time, then subtract the time slot from the actual burst time and increment it by time-slot and the loop continues until all the processes are completed.

ALGORITHM:

Step 1: Start the process

Step 2: Define an array of structure PCB processcontrolblock of size n to store the arrival time, first CPU burst time, next CPU burst time, turnaround time, and waiting time for each process.

```
struct PCB
{
    // structure members
} p[n];
```

Step 3: Input the number of processes (n) and time quantum (tq) or time slice.

.

Step 4: Display the inputted values required for processes from the Ready Queue

Step 5: Calculate start time of each process depending on the arrival time and the burst time of each process

Step 6: Create a Gantt chart as follows:

a) Schedule the processes the way they arrived only for the time quantum tq given
b) Once the time quantum is over schedule the next process in order.
c) When the last process is scheduled, schedule the first again
d) Continue till processes terminate

Step 7: Calculate Waiting time:

Wait time = (Start time – Arrival time) + (New Start time – Old finish Time)
Average waiting time = Total waiting Time / Number of process

Step 8: Calculate Turnaround time:

Turnaround time = Finish time – Arrival time Where, Finish time = Start time + Burst time
Average Turnaround time = Total Turnaround Time / Number of process

Step 9: Stop

Practical Assignments:

Set A

i. Write the program to simulate FCFS CPU-scheduling. The arrival time and first CPU-burst for different n number of processes should be input to the algorithm. Assume that the fixed IO waiting time (2 units). The next CPU-burst should be generated randomly. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

ii. Write the program to simulate Non-preemptive Shortest Job First (SJF) -scheduling. The arrival time and first CPU-burst for different n number of processes should be input to the algorithm. Assume the fixed IO waiting time (2 units). The next CPU-burst should be generated randomly. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

Set B

i. Write the program to simulate Round Robin (RR) scheduling. The arrival time and first CPU-burst for different n number of processes should be input to the algorithm. Also give the time quantum as input. Assume the fixed IO waiting time (2 units). The next CPU-burst should be generated randomly. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

i. Write the program to simulate Non-preemptive Priority scheduling. The arrival time and first CPU-burst and priority for different n number of processes should be input to the algorithm. Assume the fixed IO waiting time (2 units). The next CPU-burst should be generated randomly. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

Set C

ii. Write the program to simulate Preemptive Priority scheduling. The arrival time and first CPU-burst and priority for different n number of processes should be input to the algorithm. Assume the fixed IO waiting time (2 units). The next CPU-burst should be generated randomly. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

ii. Write the program to simulate Preemptive Shortest Job First (SJF) -scheduling. The arrival time and first CPU-burst for different n number of processes should be input to the algorithm. Assume the fixed IO waiting time (2 units). The next CPU-burst should be generated randomly. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

Assignment Evaluation

0: Not Done	[ ]	1: Incomplete	[ ]	2: Late Complete	[ ]
3: Needs Improvement	[ ]	4: Complete	[ ]	5: Well Done	[ ]

Signature of the Instructor **Date of Completion____/____/_____**

Assignment 4- Demand Paging

Demand paging the one of the most commonly used method of implanting the **Virtual**

Memory Management scheme. Virtual memory management scheme has the following advantages:

1) System can execute the process of larger size than the size of available physical memory.
2) Reduced disk IO operations.
3) More numbers of processes can be loaded into memory.
4) Performance of multi-programming and multi-tasking can be improved.

In this memory management scheme instead of loading entire process into memory, only required portion of each process is loaded in memory for execution. Different parts of physical memory of each process are brought into memory as and when required hence the name is demand paging.

Each process is logically divided into number of pages. Each page is of same size.
Physical memory (RAM) is also divided into number of equal size frames. Generally size of frame and a page is same. When process is to be loaded into memory its pages are loaded into available free frames.
Memory management scheme maintains the list of free frames.
It also maintains the page table for each process. Page table keep track of various frames allocated to each process in physical memory. That is page table is used to map the logical pages of a process to frames in physical memory.
Memory Management Unit (MMU) of computer system is responsible to convert the logical address in process to physical memory in frame.
In demand paging memory management scheme no page of any process in brought into memory until it not really required. Whenever page is required (demanded) then only it is brought into memory.

Page Fault:
When process requests some page during execution and that page is not available in physical memory then it is called as Page Fault.
Whenever page fault occurs Operating system check really it is a page fault or it is an invalid memory reference. If page fault has occurred, then system try to load the corresponding page in available free frame.

Page Replacement:
When page fault occurs, system try to load the corresponding page into one of the available free frame. If no free frame is available them system try to replace one of the existing page in frame with new page. This is called as Page Replacement. Frame selected for page replacement is as victim frame. Page replacement algorithm of demand paging memory management scheme will decide which frame will be selected as victim frame. There are various page replacement algorithms as:

1. First In First Out Page Replacement (FIFO)
2. Optimal Page Replacement(OPT)
3. Least Recently Used Page Replacement (LRU)
4. Most Recently Used Page Replacement(MRU)
5. Most Frequently Used Page Replacement(MFU)
6. Least Frequently Used Page Replacement(LFU)
7. Second Change Page Replacement's

Input to Page Replacement Algorithm:

1. **Reference String:** It is the list of numbers which represent the various page numbers demanded by the process.
2. **Number of Frames:** It represents the number of frames in physical memory.

Ex. Reference String: 3, 6, 5, 7, 3, 5, 2, 5,7 ,3,2,4,2, 8, 3, 6

It means first process request the page number 3 then page number 6 then 5 and so on.

Data Structure:

1. Array of memory frames which is used to maintain which page is loaded in which frame. With each frame we may associate the counter or a time value whenever page is loaded in that frame or replaced.
2. Array of reference String.
3. Page Fault Count.

Output:

The output for each page replacement algorithm should display how each next page is loaded in which frame.
Finally it should display the total number of page faults.

FIFO:

- In this page replacement algorithm the order in which pages are loaded in memory, in same order they are replaced.
- We can associate a simple counter value with each frame when a page is loaded in memory.
- Whenever page is loaded in free frame a next counter value is also set to it.
- When page is to be replaced, select that page which has least counter value.
- It is most simple page replacement algorithm.

OPT:

- This algorithm looks into the future page demand of a process.
- Whenever page is to be replaced it will replace that page from the physical which will not be required longer time.
- To implement this algorithm whenever page is to be replaced, we compare the pages in physical memory with their future occurrence in reference string. The page in physical memory which will not be required for longest period of time will be selected as victim.

LRU:

- This algorithm replaces that page from physical memory which is used least recently.
- To implement this algorithm, we associate a next counter value or timer value with each frame/page in physical memory wherever it is loaded or referenced from physical memory. When page replacement is to be performed, it will replace that frame in physical memory which has smallest counter value.

MRU:

- This algorithm replaces that page from physical memory which is used most recently.
- To implement this algorithm, we associate a next counter value or timer value with each frame/page in physical memory wherever it is loaded or referenced from physical memory. When page replacement is to be performed, it will replace that frame in physical memory which has greatest counter value.

MRU:

- This algorithm replaces that page from physical memory which is used most recently.
- To implement this algorithm, we associate a next counter value or timer value with each frame/page in physical memory wherever it is loaded or referenced from physical memory. When page replacement is to be performed, it will replace that frame in physical memory which has greatest counter value.

MFU:

- This algorithm replaces that page from physical memory which is used most frequently.
- This algorithm is bit complex to implement.
- To implement this algorithm, we have to maintain the history of each page that how many times it is used (frequency count) so far whenever it is loaded in physical memory or referenced from physical memory. When page replacement is to be performed, it will replace that frame in physical memory of which frequency count is greatest.
- If frequency count is same for two or more pages then it will apply FCFS.

LFU:

- This algorithm replaces that page from physical memory which is used most frequently.
- This algorithm is bit complex to implement.
- To implement this algorithm, we have to maintain the history of each page that how many times it is used (frequency count) so far whenever it is loaded in physical memory or referenced from physical memory. When page replacement is to be performed, it will replace that frame in physical memory of which frequency count is smallest.
- If frequency count is same for two or more pages then it will apply FCFS.

Second Chance Page Replacement:

- This algorithm is also called as Clock replacement policy.
- In this algorithm frames from physical memory are consider for the replacement in round robin manner. A page that has been accessed in two consecutive considerations will not be replaced.
- This algorithm is an extension of FIFO page replacement.
- To implement this algorithm, we have to associate second chance bit to each frame in physical memory. Initially when a page is loaded in memory its second chance bit is set to 0. Each time a memory frame is referenced set the second chance bit to 1. When page replacement it to be performed access the memory frames in round robin manner. If second chance bit is 1 then reset, it to 0. If second chance bit is zero, then replace the page with that frame.
- This algorithm gives far better performance than FCFS page replacement.

Algorithm:

1) First In First Out (FIFO):

Step 1: Start

Step 2: Define a structures of memory frame and variables as M reference string length and F number of memory frames, time, faults, and next_replace.

```
Struct MemoryFrame
{   // Structure Member
 } m[n];
```

Step 3: Input the length of reference string M and number of memory frames F
Initialize each frame in PT page_number = -1 (to indicate empty)
Read M page numbers into RS [] an array of reference string.

Step 4: For each page RS[i] in the reference string (from i = 0 to M - 1)

a) **Search** for the page in memory frames:
b) If found (Search () returns index) then it is a **hit**
c) If not found, then it is a **fault**
   - Replace the frame at next_replace index with the current page.
   - Increment next_replace
   - Increment faults.

Step 5: Print the current state of memory frames.

Step 6: After all references are processed, print the total number of page faults.

Step 7: Stop

Practical Assignment:

Set A

i. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n as the number of memory frames.
Reference String : 12,15,12,18,6,8,11,12,19,12,6,8,12,15,19,8

   i. Implement FIFO
   ii. Implement LRU

Set B

i. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n as the number of memory frames.
Reference String : 12,15,12,18,6,8,11,12,19,12,6,8,12,15,19,8

   i. Implement OPT
   ii. Implement MFU

Set C

i. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n as the number of memory frames.
Reference String: 2,5,2,8,5,4,1,2,3,2,6,1,2,5,9,8

   i. Implement MRU
   ii. Implement Second Chance Page Replacement.
   iii. Least Frequently Used

Assignment Evaluation

0: Not Done	[ ]	1: Incomplete	[ ]	2: Late Complete	[ ]
3: Needs Improvement	[ ]	4: Complete	[ ]	5: Well Done	[ ]

Signature of the Instructor_____ **Date of Completion**____/____/_____