OS

Assignment 1A

Write a menu driven shell script to display disk space usage of the file system, including the total space, used space, available space, and usage percentage

Assignment-1A

Code :

```
#!/bin/bash

display_disk_usage() {

echo "Disk Space Usage:"

 df -h

}

display_total_space() { echo "Total Space:"

df -h --total | grep "total" | awk '{print $2}'

}

display_used_space() { echo "Used Space:"

df -h --total | grep "total" | awk '{print $3}'

}

display_available_space() { echo "Available

Space:" df -h --total | grep "total" | awk

'{print $4}'

}

display_usage_percentage() {

echo "Usage Percentage:"

 df -h --total | grep "total" | awk '{print $5}'

}

# Menu driven interface while true;

do echo "Menu:" echo "1.

Display Disk Space Usage" echo "2.

Display Total Space" echo "3.

Display Used Space" echo "4.

Display Available Space" echo "5.
```

```
Display Usage Percentage"
 echo "6. Exit" read -p "Enter your
choice [1-6]: " choice
 case $choice in
1) display_disk_usage ;;
2) display_total_space ;;
3) display_used_space ;;
4) display_available_space ;;
5) display_usage_percentage ;;
6) echo "Exiting..."; exit 0 ;;
 *) echo "Invalid choice. Please enter a number between 1 and 6." ;;
 esac
echo ""
done
```

Assignment-1B

Write a menu driven shell script that implements various file and directory related commands.

Code :

```
#!/bin/bash
while true; do clear
echo "File and Directory Operations Menu"
echo "=================================="
echo "1. List files and directories"
echo "2. Create a file" echo "3.
Create a directory" echo "4.
Delete a file" echo "5. Delete a
directory" echo "6. Exit" echo -n
"Enter your choice: " read choice
case $choice in
1)
echo
```

```
echo "Listing files and directories:"
echo "============================="
ls -lh echo
read -p "Press any key to return to the menu..."
;;
2)
echo -n "Enter the name of the file to create: "
read filename touch
"$filename" echo "File
'$filename' created."
echo
read -p "Press any key to return to the menu..."
;;
3)
echo -n "Enter the name of the directory to create: "
read dirname mkdir -p "$dirname"
echo "Directory '$dirname' created."
echo
read -p "Press any key to return to the menu..."
;;
4)
echo -n "Enter the name of the file to delete: "
read filename rm -i "$filename"
echo "File '$filename' deleted."
echo
read -p "Press any key to return to the menu..."
;;
5)
echo -n "Enter the name of the directory to delete: "
read dirname rmdir "$dirname" echo
"Directory '$dirname' deleted." echo
```

```
read -p "Press any key to return to the menu..."
;;
6) echo
"Exiting..."
exit 0
;;
*)
echo "Invalid option. Please try again."
read -p "Press any key to return to the menu..."
;;
esac done
```

Assignment 1C

Write a menu driven shell script that checks and sets file system permissions for files and directories based on specified criteria, such as ownership and read/write/execute permissions.

Code :

```
#!/bin/bash
# Function to display current permissions of a file or directory
check_permissions() { echo -n "Enter the file or directory to
check permissions: " read -r item if [ -e "$item" ]; then echo
"Current permissions for '$item':" ls -ld
"$item"
else echo "The file or directory '$item' does not
exist."
fi echo
}
# Function to set permissions for a file or directory set_permissions() {
echo -n "Enter the file or directory to set
permissions: "
read -r item
echo -n "Enter the new permissions (e.g., 755): " read -r
```

```
permissions
if [ -e "$item" ]; then if chmod "$permissions"
"$item"; then echo
"Permissions for '$item' set to
'$permissions'." else echo "Failed to set
permissions for '$item'." fi
else
echo "The file or directory '$item' does not exist."
fi echo
}
# Function to change the ownership of a file or directory
change_ownership() { echo -n "Enter the file or directory to
change ownership: " read -r item
echo -n "Enter the new owner (e.g., username): " read -r
owner echo -n "Enter the new group (e.g., groupname, leave
blank if
not changing): " read -r group if [ -e "$item" ]; then if [ -z
"$group" ]; then if chown "$owner" "$item"; then echo
"Ownership of '$item' changed to '$owner'."
else echo "Failed to change ownership of
'$item'." fi else
if chown "$owner:$group" "$item"; then echo
"Ownership of '$item' changed to
'$owner:$group'."
else echo "Failed to change ownership of
'$item'." fi
fi
else
echo "The file or directory '$item' does not exist."
fi echo
}
```

```
# Main menu function show_menu() {
clear echo "File and Directory Permissions Menu" echo "1.
Check file or directory permissions" echo "2. Set file or
directory permissions" echo "3. Change ownership of file or
directory" echo "4. Exit"
echo -n "Please choose an option [1-4]: "
}
# Main script loop while
true; do show_menu
read -r option
case $option in
1) check_permissions ;;
2) set_permissions ;;
3) change_ownership ;;
4) echo "Exiting..."; exit 0 ;;
*) echo "Invalid option, please try again." ;; esac
echo "Press [Enter] to return to the menu." read -r
done
```

Assignment 2

Develop a program to demonstrate process creation, termination, and communication using PIPE as an IPC (Inter process Communication) mechanism

CODE:-

```
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>
#include <cstring>
void processA(int); void
processB(int);
void processA(int writefd) {
std::string buff; std::cout <<
"Enter a string: ";
```

```
std::getline(std::cin, buff);


 if (!buff.empty() && buff[buff.size() - 1] == '\n') {
buff.pop_back();
 }
 write(writefd, buff.c_str(), buff.size());
}
void processB(int readfd) {
 int n, i, j;
 char str[80], temp;
 std::cout << "In child process" << std::endl;
 n = read(readfd, str, sizeof(str));
str[n] = '\0';
 // Reverse the string
 i = 0; j =
strlen(str) - 1;
while (i < j) {
temp = str[i];
str[i] = str[j];
str[j] = temp;
i++; j--;
 }
 std::cout << "Reversed string: " << str << std::endl;
}
int main() { int
pipe1[2];
pid_t childpid;
 pipe(pipe1);
childpid = fork();
 if (childpid == 0) {
```

```
 close(pipe1[1]);
processB(pipe1[0]);
 } else {

 close(pipe1[0]);
processA(pipe1[1]);
wait(NULL);
 }
 return 0;
}
```

OperatingSystem

Assignment:3

Implement a simple process scheduler using scheduling algorithms such as FCFS (First-Come, First Served), SJF(Shortest Job First), and RR(Round Robin), Priority Scheduling

Implement a simple process scheduler using non preemptive scheduling algorithms like First Come First Served

Implement a simple process scheduler using non preemptive and preemptive scheduling algorithms like SJF(Shortest Job First), SRTF (Shortest Remaining Time First-Preemptive SJF Scheduling Algorithm)

Implement a simple process scheduler using non preemptive and preemptive scheduling algorithms like priority scheduling

Implement a simple process scheduler using preemptive Round Robin scheduling algorithm.

1. FCFSScheduler(Non-Preemptive):

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include<semaphore.h>
structProcess{
```

```
int pid;
int burst_time;
intwaiting_time;
intturnaround_time;
int memory_size;
};
sem_tmemory_lock;
voidfindWaitingTime(structProcessproc[],intn){
proc[0].waiting_time = 0;
for(inti=1;i<n;i++){
proc[i].waiting_time=proc[i-1].waiting_time+proc[i-1].burst_time;
}
}
voidfindTurnaroundTime(structProcessproc[],intn){ for
(int i = 0; i < n; i++) {
proc[i].turnaround_time=proc[i].waiting_time+proc[i].burst_time;
}
}
voidallocateMemory(structProcess*proc){
sem_wait(&memory_lock);
printf("Allocating%dMBmemoryforprocess%d\n",proc->memory_size,proc->pid);
}
voidreleaseMemory(structProcess*proc){
printf("Releasing%dMBmemoryforprocess%d\n",proc->memory_size,proc->pid);
sem_post(&memory_lock);
}
void*processExecution(void*arg){
structProcess*proc=(structProcess*)arg;
allocateMemory(proc);
printf("Process%disexecuting...\n",proc->pid);
releaseMemory(proc);
```

```
returnNULL;
}
voidFCFS(structProcessproc[],intn){
pthread_t threads[n];
inttotal_wt=0,total_tat=0;
for(inti=0;i<n;i++){
pthread_create(&threads[i],NULL,processExecution,(void*)&proc[i]);
pthread_join(threads[i], NULL);
}
findWaitingTime(proc, n);
findTurnaroundTime(proc, n);
printf("\nProcessID\tBurstTime\tWaitingTime\tTurnaroundTime\tMemory\n");
for (int i = 0; i < n; i++) {
printf("%d\t\t%d\t\t%d\t\t%d\t\t%dMB\n",proc[i].pid,proc[i].burst_time,
proc[i].waiting_time, proc[i].turnaround_time, proc[i].memory_size);
total_wt += proc[i].waiting_time;
total_tat += proc[i].turnaround_time;
}
floatavg_wt=(float)total_wt/n;
floatavg_tat=(float)total_tat/n;
printf("\nAverage Waiting Time: %.2f\n", avg_wt);
printf("Average Turnaround Time: %.2f\n", avg_tat);
}
intmain(){
sem_init(&memory_lock,0,1);
intn;
printf("Enternumberofprocesses:");
scanf("%d", &n);
structProcessproc[n];
for(inti=0;i<n;i++){ proc[i].pid =
i + 1;
```

```
printf("Enterbursttimeandmemorysizeforprocess%d:",proc[i].pid);
scanf("%d%d",&proc[i].burst_time,&proc[i].memory_size);
}
FCFS(proc,n);
sem_destroy(&memory_lock);
return 0;
}
```

2. SJFscheduler(Non-preemptive):

Code:

```
#include <stdio.h>
#include<limits.h>
structProcess{
int pid;
int burst_time;
int arrival_time;
intwaiting_time;
intturnaround_time;
int completed;
};
intfindNextProcess(structProcessproc[],intn,intcurrent_time){ int
shortest_idx = -1;
intmin_burst_time=INT_MAX;
for(inti=0;i<n;i++){
if(proc[i].arrival_time<=current_time&&proc[i].completed==0&&proc[i].burst_time<
min_burst_time) {
min_burst_time = proc[i].burst_time;
shortest_idx = i;
}
}
returnshortest_idx;
```

```
}
voidfindWaitingAndTurnaroundTime(structProcessproc[],intn,float*avg_wt,float*avg_tat){ int
current_time = 0;
intcompleted_processes=0;
inttotal_waiting_time=0,total_turnaround_time=0;
while(completed_processes<n){
intidx=findNextProcess(proc,n,current_time);
proc[idx].waiting_time=current_time-proc[idx].arrival_time; if
(proc[idx].waiting_time < 0) proc[idx].waiting_time = 0;
current_time+=proc[idx].burst_time;
proc[idx].turnaround_time=proc[idx].waiting_time+proc[idx].burst_time;
proc[idx].completed = 1;
completed_processes++;
total_waiting_time += proc[idx].waiting_time;
total_turnaround_time += proc[idx].turnaround_time;
}else{
current_time++;
}
}
*avg_wt=(float)total_waiting_time/n;
*avg_tat=(float)total_turnaround_time/n;
}
voidSJF(structProcessproc[],intn){
float avg_wt = 0.0, avg_tat = 0.0;
findWaitingAndTurnaroundTime(proc,n,&avg_wt,&avg_tat);
printf("ProcessID\tArrivalTime\tBurstTime\tWaitingTime\tTurnaroundTime\n"); for
(int i = 0; i < n; i++) {
printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n",proc[i].pid,proc[i].arrival_time,proc[i].burst_time,
proc[i].waiting_time, proc[i].turnaround_time);
}
printf("\nAverage Waiting Time: %.2f\n", avg_wt);
```

```
printf("Average Turnaround Time: %.2f\n", avg_tat);
}
intmain(){ int
n;
printf("Enternumberofprocesses:");
scanf("%d", &n);
structProcessproc[n];
for(inti=0;i<n;i++){ proc[i].pid =
i + 1;
printf("Enterarrivaltimeandbursttimeforprocess%d:",proc[i].pid);
scanf("%d %d", &proc[i].arrival_time, &proc[i].burst_time);
proc[i].completed=0;
}
SJF(proc,n);
return 0;
}
```

3. SJFscheduler(Preemptive):

Code:

```
#include <stdio.h>
#include<limits.h>
structProcess{
int pid;
int burst_time;
intarrival_time;
intremaining_time;
int waiting_time;
intturnaround_time;
int completed;
};
intfindNextProcess(structProcessproc[],intn,intcurrent_time){ int
```

```
shortest_idx = -1;
intmin_remaining_time=INT_MAX;
for(inti=0;i<n;i++){
if(proc[i].arrival_time<=current_time&&proc[i].remaining_time>0&&
proc[i].remaining_time < min_remaining_time) {
min_remaining_time=proc[i].remaining_time;
shortest_idx = i;
}
}
returnshortest_idx;
}
voidfindWaitingAndTurnaroundTime(structProcessproc[],intn,float*avg_wt,float*avg_tat){ int
current_time = 0;
intcompleted_processes=0;
inttotal_waiting_time=0,total_turnaround_time=0;
for(inti=0;i<n;i++){
proc[i].remaining_time=proc[i].burst_time;
}
while(completed_processes<n){
intidx=findNextProcess(proc,n,current_time);
if(idx!=-1){
proc[idx].remaining_time--;
current_time++;
if(proc[idx].remaining_time==0){
completed_processes++;
proc[idx].turnaround_time=current_time-proc[idx].arrival_time;
proc[idx].waiting_time=proc[idx].turnaround_time-proc[idx].burst_time;
if(proc[idx].waiting_time<0){
proc[idx].waiting_time = 0;
}
total_waiting_time += proc[idx].waiting_time;
```

```
total_turnaround_time += proc[idx].turnaround_time;
}
}else{
current_time++;
}
}
*avg_wt=(float)total_waiting_time/n;
*avg_tat=(float)total_turnaround_time/n;
}
voidSRTF(structProcessproc[],intn){ float
avg_wt = 0.0, avg_tat = 0.0;
findWaitingAndTurnaroundTime(proc,n,&avg_wt,&avg_tat);
printf("ProcessID\tArrivalTime\tBurstTime\tWaitingTime\tTurnaroundTime\n"); for
(int i = 0; i < n; i++) {
printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n",proc[i].pid,proc[i].arrival_time,proc[i].burst_time,
proc[i].waiting_time, proc[i].turnaround_time);
}
printf("\nAverage Waiting Time: %.2f\n", avg_wt);
printf("Average Turnaround Time: %.2f\n", avg_tat);
}
intmain(){ int
n;
printf("Enternumberofprocesses:");
scanf("%d", &n);
structProcessproc[n];
for(inti=0;i<n;i++){ proc[i].pid =
i + 1;
printf("Enterarrivaltimeandbursttimeforprocess%d:",proc[i].pid);
scanf("%d %d", &proc[i].arrival_time, &proc[i].burst_time);
proc[i].completed=0;
}
```

```
SRTF(proc,n);
return 0;
}
```

4. STRFscheduler:

Code:

```
#include <stdio.h>
#include<limits.h>
structProcess{
int pid;
int burst_time;
intarrival_time;
intremaining_time;
int waiting_time;
intturnaround_time;
int completed;
};
intfindNextProcess(structProcessproc[],intn,intcurrent_time){ int
shortest_idx = -1;
intmin_remaining_time=INT_MAX;
for(inti=0;i<n;i++){
if(proc[i].arrival_time<=current_time&&proc[i].remaining_time>0&&
proc[i].remaining_time < min_remaining_time) {
min_remaining_time=proc[i].remaining_time;
shortest_idx = i;
}
}
returnshortest_idx;
}
voidfindWaitingAndTurnaroundTime(structProcessproc[],intn,int*total_wt,int*total_tat){ int
current_time = 0;
```

```
intcompleted_processes=0;
for(inti=0;i<n;i++){
proc[i].remaining_time=proc[i].burst_time;
}
while(completed_processes<n){
intidx=findNextProcess(proc,n,current_time);
if(idx!=-1){
proc[idx].remaining_time--;
current_time++;
if(proc[idx].remaining_time==0){
completed_processes++;
proc[idx].turnaround_time=current_time-proc[idx].arrival_time;
proc[idx].waiting_time=proc[idx].turnaround_time-proc[idx].burst_time;
if(proc[idx].waiting_time<0){
proc[idx].waiting_time = 0;
}
*total_wt+=proc[idx].waiting_time;
*total_tat+=proc[idx].turnaround_time;
}
}else{
current_time++;
}
}
}
voidSRTF(structProcessproc[],intn){ int
total_wt = 0, total_tat = 0;
findWaitingAndTurnaroundTime(proc,n,&total_wt,&total_tat);
printf("ProcessID\tArrivalTime\tBurstTime\tWaitingTime\tTurnaroundTime\n"); for
(int i = 0; i < n; i++) {
printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n",proc[i].pid,proc[i].arrival_time,proc[i].burst_time,
proc[i].waiting_time, proc[i].turnaround_time);
```

```
}
floatavg_wt=(float)total_wt/n;
floatavg_tat=(float)total_tat/n;
printf("\nAverage Waiting Time: %.2f\n", avg_wt);
printf("Average Turnaround Time: %.2f\n", avg_tat);
}
intmain(){ int
n;
printf("Enternumberofprocesses:");
scanf("%d", &n);
structProcessproc[n];
for(inti=0;i<n;i++){ proc[i].pid =
i + 1;
printf("Enterarrivaltimeandbursttimeforprocess%d:",proc[i].pid);
scanf("%d %d", &proc[i].arrival_time, &proc[i].burst_time);
}
SRTF(proc,n);
return 0;
}
```

5. Priorityscheduling(Non-preemptive):

Code:

```
#include<stdio.h>
structProcess{
int pid;
int burst_time;
intarrival_time;
int priority;
intwaiting_time;
intturnaround_time;
```

```
intcompletion_time;
};
voidfindWaitingAndTurnaroundTime(structProcessproc[],intn){ int
current_time = 0, completed = 0, min_priority_idx;
inttotal_wt=0,total_tat=0; int
is_completed[n];
for(inti=0;i<n;i++)is_completed[i]=0;
while (completed < n) {
min_priority_idx = -1;
intmin_priority=9999;
for(inti=0;i<n;i++){
if(proc[i].arrival_time<=current_time&&!is_completed[i]){ if
(proc[i].priority < min_priority) {
min_priority=proc[i].priority;
min_priority_idx = i;
}elseif(proc[i].priority==min_priority){
if(proc[i].arrival_time<proc[min_priority_idx].arrival_time){
min_priority_idx = i;
}
}
}
}
if(min_priority_idx!=-1){
current_time += proc[min_priority_idx].burst_time;
proc[min_priority_idx].completion_time=current_time;
proc[min_priority_idx].turnaround_time=proc[min_priority_idx].completion_timeproc[min_priority_
idx].arrival_time;
proc[min_priority_idx].waiting_time=proc[min_priority_idx].turnaround_timeproc[min_priority_idx].
burst_time;
total_wt += proc[min_priority_idx].waiting_time;
total_tat += proc[min_priority_idx].turnaround_time;
is_completed[min_priority_idx] = 1;
```

```
completed++;
}else{
current_time++;
}
}
printf("\nNon-PreemptivePriorityScheduling\n");
printf("Process ID\tArrival Time\tBurst Time\tPriority\tCT\tTAT\tWT\n");
for (int i = 0; i < n; i++) {
printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t%d\n", proc[i].pid, proc[i].arrival_time,
proc[i].burst_time, proc[i].priority, proc[i].completion_time, proc[i].turnaround_time,
proc[i].waiting_time);
}
printf("\nAverage Waiting Time: %.2f\n", (float)total_wt / n);
printf("AverageTurnaroundTime:%.2f\n",(float)total_tat/n);
}
intmain(){ int
n;
printf("Enternumberofprocesses:");
scanf("%d", &n);
structProcessproc[n];
for(inti=0;i<n;i++){ proc[i].pid =
i + 1;
printf("Enter arrival time, burst time and priority for process %d: ", proc[i].pid);
scanf("%d%d%d",&proc[i].arrival_time,&proc[i].burst_time,&proc[i].priority);
}
findWaitingAndTurnaroundTime(proc, n);
return 0;
}
```

6. Priorityscheduling(Preemptive):

Code:

```
#include<stdio.h>
structProcess{
int pid;
intburst_time;
intarrival_time;
int priority;
intremaining_time;
int waiting_time;
intturnaround_time;
intcompletion_time;
};
voidfindWaitingAndTurnaroundTime(structProcessproc[],intn){ int
current_time = 0, completed = 0, min_priority_idx;
inttotal_wt=0,total_tat=0;
for(inti=0;i<n;i++){
proc[i].remaining_time=proc[i].burst_time;
}
while (completed < n) {
min_priority_idx=-1;
intmin_priority=INT_MAX;
for(inti=0;i<n;i++){
if(proc[i].arrival_time<=current_time&&proc[i].remaining_time>0){ if
(proc[i].priority < min_priority) {
min_priority=proc[i].priority;
min_priority_idx = i;
}
}
}
if(min_priority_idx!=-1){
proc[min_priority_idx].remaining_time--;
current_time++;
```

```
if(proc[min_priority_idx].remaining_time==0){
proc[min_priority_idx].completion_time=current_time;
proc[min_priority_idx].turnaround_time=proc[min_priority_idx].completion_time
-proc[min_priority_idx].arrival_time;
proc[min_priority_idx].waiting_time=proc[min_priority_idx].turnaround_timeproc[min_priority_idx].
burst_time;
total_wt+=proc[min_priority_idx].waiting_time;
total_tat += proc[min_priority_idx].turnaround_time;
completed++;
}
}else{
current_time++;
}
}
printf("\nPreemptivePriorityScheduling\n");
printf("Process ID\tArrival Time\tBurst Time\tPriority\tCT\tTAT\tWT\n");
for (int i = 0; i < n; i++) {
printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t%d\t%d\n",proc[i].pid,proc[i].arrival_time,
proc[i].burst_time, proc[i].priority, proc[i].completion_time, proc[i].turnaround_time,
proc[i].waiting_time);
}
printf("\nAverageWaitingTime:%.2f\n",(float)total_wt/n);
printf("AverageTurnaroundTime:%.2f\n",(float)total_tat/n);
}
intmain(){
int n;
printf("Enternumberofprocesses:"); scanf("%d",
&n);
structProcessproc[n];
for(inti=0;i<n;i++){ proc[i].pid = i
+ 1;
```

```
printf("Enter arrival time, burst time and priority for process %d: ", proc[i].pid);
scanf("%d%d%d",&proc[i].arrival_time,&proc[i].burst_time,&proc[i].priority);
}
findWaitingAndTurnaroundTime(proc, n);
return 0;
}
```

7. RoundRobin:

Code:

```
#include<stdio.h>
structProcess{
int pid;
intburst_time;
intremaining_time;
int waiting_time;
intturnaround_time;
};
voidRoundRobin(structProcessproc[],intn,intquantum){ int t
= 0;
int remaining_processes = n;
inttotal_wt=0,total_tat=0;
for(inti=0;i<n;i++){
proc[i].remaining_time=proc[i].burst_time;
}
while(remaining_processes>0){ for
(int i = 0; i < n; i++) {
if(proc[i].remaining_time>0){
if(proc[i].remaining_time>quantum){ t
+= quantum;
proc[i].remaining_time-=quantum;
}else{
```

```
t+=proc[i].remaining_time;
proc[i].waiting_time=t-proc[i].burst_time;
proc[i].remaining_time = 0;
remaining_processes--;
}
}
}
}
for(inti=0;i<n;i++){
proc[i].turnaround_time=proc[i].waiting_time+proc[i].burst_time;
total_wt += proc[i].waiting_time;
total_tat+=proc[i].turnaround_time;
}
printf("ProcessID\tBurstTime\tWaitingTime\tTurnaroundTime\n");
for (int i = 0; i < n; i++) {
printf("%d\t\t%d\t\t%d\t\t%d\n", proc[i].pid, proc[i].burst_time, proc[i].waiting_time,
proc[i].turnaround_time);
}
floatavg_wt=(float)total_wt/n;
floatavg_tat=(float)total_tat/n;
printf("\nAverage Waiting Time: %.2f\n", avg_wt);
printf("Average Turnaround Time: %.2f\n", avg_tat);
}
intmain(){
intn,quantum;
printf("Enternumberofprocesses:"); scanf("%d",
&n);
structProcessproc[n];
for(inti=0;i<n;i++){ proc[i].pid = i
+ 1;
printf("Enterbursttimeforprocess%d:",proc[i].pid); scanf("%d",
```

```
&proc[i].burst_time);
}
printf("Entertimequantum:");
scanf("%d", &quantum);
RoundRobin(proc,n,quantum);
return 0;
}
```

Assignment 4

Implement synchronization mechanisms such as semaphores, mutexes, and condition variables to solve synchronization problems for any suitable multithreaded application.

Develop a thread pool manager that maintains a pool of worker threads for executing tasks concurrently. Use semaphores or mutexes to manage thread creation, task assignment, and synchronization between the main thread and worker threads.

Code :

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <unistd.h>
#define BUFFER_SIZE 5
int buffer[BUFFER_SIZE];
int count = 0;
pthread_mutex_t mutex;
sem_t emptySlots; sem_t
fullSlots;
void* producer(void* arg) {
int item; while (1) {
 item = rand() % 100;
sleep(1);
sem_wait(&emptySlots);
pthread_mutex_lock(&mutex);
```

```
 buffer[count] = item;
count++;
 printf("Producer produced: %d (Buffer count: %d)\n", item, count);
pthread_mutex_unlock(&mutex); sem_post(&fullSlots);
 }
 return NULL;
}
void* consumer(void* arg) {
 int item; while (1) {
sleep(2);
sem_wait(&fullSlots);
pthread_mutex_lock(&mutex);
 count--; item =
buffer[count];
 printf("Consumer consumed: %d (Buffer count: %d)\n", item, count);
pthread_mutex_unlock(&mutex); sem_post(&emptySlots);
 }
 return NULL;
}
int main() {
 pthread_t producerThread, consumerThread;
 pthread_mutex_init(&mutex, NULL);
 sem_init(&emptySlots, 0, BUFFER_SIZE);
 sem_init(&fullSlots, 0, 0);
 pthread_create(&producerThread, NULL, producer, NULL);
pthread_create(&consumerThread, NULL, consumer, NULL);
pthread_join(producerThread, NULL);
pthread_join(consumerThread, NULL);
pthread_mutex_destroy(&mutex);
sem_destroy(&emptySlots); sem_destroy(&fullSlots);
```

```
 return 0;
}
```

Assignment 5

In a real computer system, neither the resources available nor the demands of processes for resources are consistent over long periods(months).Resources break or are replaced, new processes come and go, and new resources are bought and added to the system. If deadlock is controlled by the banker's algorithm, which of the following changes can be made safely (without introducing the possibility of dead lock), and under what circumstances?

a. Increase Available(new resources added).

b. Decrease Available(resource permanently removed from system).

```
#include<bits/stdc++.h> using
namespace std ;
void Bankers_Algorithm() {
cout << "Enter no of Processes";
int no_of_processes ; cin >>
no_of_processes ;
cout << "\nEnter no of resources :" ; int
no_of_resources ;
cin >> no_of_resources ;
int initial[no_of_processes][no_of_resources] ;
cout << "\nEnter initial resources allocated :\n" ;
for (int i = 0 ; i < no_of_processes ; i++)
{
cout << "\nFor Process " << i << " : ";
for (int j = 0 ; j < no_of_resources ; j++)
{ cin >> initial[i][j] ;
} } int
max_need[no_of_processes][no_of_resources]
;
cout << "\nEnter max Process required : \n"
; for (int i = 0 ; i < no_of_processes ; i++) {
```

```
cout << "\nFor Process " << i
<< " : "; for (int j = 0 ; j < no_of_resources
; j++)
{ cin >> max_need[i][j] ;
} } int
need[no_of_processes][no_of_processes
] ;
for (int i = 0 ; i < no_of_processes ;
i++) { for (int j = 0 ; j < no_of_resources
;
j++) { need[i][j] = max_need[i][j] -
initial[i][j] ;
}
}
cout << "\nEnter Initial Available
resources"; int
resources[no_of_resources] ; for (int i = 0 ;
i < no_of_resources ; i++) { cout <<
"\nEnter resource " << i <<" : " ; cin >>
resources[i] ;
}
vector<int> safeSequence ; vector<bool>
process_done(no_of_processes, false) ; while
(true) { bool
deadLock = true ;
for (int i = 0 ; i < no_of_processes ; i++)
{ if (process_done[i]) continue ; bool
done = true ;
for (int j = 0 ; j < no_of_resources ; j++)
{ if (need[i][j] > resources[j] ) { done =
false
```

```
; break;
}
}
if (!done) continue ;
safeSequence.push_back(i + 1) ; process_done[i]
= true ; deadLock
= false ; for (int j = 0
; j <
no_of_resources ;
j++) { resources[j] +=
need[i][j];
}
}
if (deadLock) { cout << "\n\nDeadlock
will
Occur\n"; return ; }
if (safeSequence.size() ==
no_of_processes) { break; } } cout <<
"\n\nPr Allocation Max Need \t \n\n"; for
(int i = 0 ; i < no_of_processes ; i++) { cout
<< "P" << i + 1 << " " ; for (int j
= 0 ; j
< no_of_resources ; j++)
{ cout << initial[i][j] << " " ;
} cout << " ";
for (int j
= 0 ; j
<
no_
of_reso
urce
```

```
s
;
j++)
{ cout << max_need[i][j] << " " ;
} cout << " "; for (int j = 0 ; j <
no_of_resources ; j++) { cout <<
need[i][j] << " ";
} cout <<
endl; } cout << "\n\nSafe sequence
is : \n";
for (int i = 0 ; i < no_of_processes ; i++)
{ cout << safeSequence[i] <<
" : "; }
cout <<
endl;
}
int main() {
Bankers_Algorithm() ;
}
```

Assignment 6

Implement a virtual memory system with features like demand paging, page replacement algorithms (e.g., FIFO, LRU, Optimal), and handling page faults.

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>

using namespace std;
```

```
// Class to simulate the Virtual Memory System
class VirtualMemory { private: int
numFrames; vector<int> pageReference;
vector<int> memory;

public:
 VirtualMemory(int frames, vector<int>& reference) :
numFrames(frames), pageReference(reference) {
memory.resize(numFrames, -1); // Initialize memory with -1 (empty)
 }

 // FIFO Page Replacement
void FIFO() { queue<int>
fifoQueue; int
pageFaults = 0;

 for (int page : pageReference) { if (find(memory.begin(), memory.end(),
page) == memory.end()) { // page fault pageFaults++; if
(fifoQueue.size() == numFrames)
{ int oldPage = fifoQueue.front();
fifoQueue.pop(); replace(memory.begin(), memory.end(),
oldPage, page);
 }
 fifoQueue.push(page); auto it =
find(memory.begin(), memory.end(), -1); if (it !=
memory.end()) {
 *it = page;
 }
 }
 printMemoryState();
 }
```

```
 cout << "FIFO - Total Page Faults: " << pageFaults << endl;
 }

 // LRU Page Replacement
void LRU() { vector<int>
lruList; int pageFaults =
0;

 for (int page : pageReference) { auto it = find(lruList.begin(), lruList.end(),
page); if (it == lruList.end()) { // page fault pageFaults++; if
(lruList.size() == numFrames) { lruList.erase(lruList.begin()); // Remove
the least recently used
 }
 lruList.push_back(page);
 } else {
 // Move the recently accessed page to the end
 lruList.erase(it);
lruList.push_back(page);
 }
 updateMemory(lruList);
printMemoryState();
 }
 cout << "LRU - Total Page Faults: " << pageFaults << endl;
 }

 // Optimal Page Replacement
void Optimal() { int pageFaults
= 0;

 for (int i = 0; i < pageReference.size(); i++) { int page = pageReference[i]; if
(find(memory.begin(), memory.end(), page) == memory.end()) { // page fault
```

```
 pageFaults++; if (find(memory.begin(),
memory.end(), -1) != memory.end()) {
replace(memory.begin(), memory.end(), -1, page);
 } else {
 // Find the page to replace using Optimal algorithm
int farthestUse = -1, replaceIndex = -1; for (int j = 0; j
< numFrames; j++) {
 int nextUse = -1; for (int k = i + 1; k
< pageReference.size(); k++) { if
(pageReference[k] == memory[j]) { nextUse
= k; break;
 }
 }
 if (nextUse == -1) { replaceIndex
= j; break;
 } if (nextUse >
farthestUse) { farthestUse =
nextUse; replaceIndex
= j;
 }
 }
 memory[replaceIndex] = page;
 }
 }
 printMemoryState();
 }
 cout << "Optimal - Total Page Faults: " << pageFaults << endl;
 }

 // Update memory with the current state of pages
void updateMemory(const vector<int>& lruList) { for
```

```
(int i = 0; i < numFrames; i++) { if (i < lruList.size())
{ memory[i] = lruList[i];
 } else {
 memory[i] = -1; // Mark as empty if fewer pages in memory
 }
 }
 }

 // Print current memory state void
printMemoryState() { for (int i = 0;
i < numFrames; i++) { cout
<< memory[i] << " ";
 }
 cout << endl;
 }
};

int main() { int numFrames;
int numPages;

 // Input number of frames cout << "Enter
the number of frames: "; cin >>
numFrames;

 // Input page reference string cout << "Enter the number
of pages in reference string: "; cin >> numPages;

 vector<int> pageReference(numPages); cout
<< "Enter the page reference string: "; for (int i
= 0; i < numPages; i++) { cin >>
pageReference[i];
```

```
}

// Create VirtualMemory object
VirtualMemory vm(numFrames, pageReference);

// FIFO Algorithm cout << "FIFO
Page Replacement:\n"; vm.FIFO();

// LRU Algorithm cout << "LRU
Page Replacement:\n"; vm.LRU();

// Optimal Algorithm cout << "Optimal Page
Replacement:\n"; vm.Optimal(); return 0;
}
```

Assignment 7

Design and implement a simple file system with basic operations like file creation, deletion, reading, and writing using data structures like inodes and file alllocation tables.

Operations

1. Create File: Create a file, assign an inode, allocate disk blocks, and add an entry in the directory.

2. Delete File: Remove the file's entry from the directory, free its disk blocks, and remove its inode.

3. Read File: Fetch data from blocks listed in the inode and reconstruct the file contents.

4. Write to File: Write data into blocks, update the inode and FAT, and manage block allocation.

Code:-

```
#include <iostream>
#include <unordered_map>
#include <vector>
#include <string>
```

```
#define MAX_BLOCKS 100 // Maximum number of storage blocks for simplicity using namespace std;
// Structure for an inode to store file metadata struct
Inode { string
name; int
size;
vector<int> blocks; // List of blocks allocated to the file
};
// Simple File Allocation Table (FAT) array to track block usage
int FAT[MAX_BLOCKS];
// FileSystem class to manage files and operations class
FileSystem { unordered_map<string,
Inode> files; public:
FileSystem() {
// Initialize FAT to -1 (indicating all blocks are free) for (int
i = 0; i < MAX_BLOCKS; i++) FAT[i] = -1;
}
// Create a file with a given name and size (in blocks) bool
createFile(const string& name, int size) { if
(files.find(name) != files.end()) { cout << "File already
exists.\n"; return false;
}
Inode inode = {name, size, {}}; int blocksAllocated =
0;
// Allocate blocks for the file for (int i = 0; i < MAX_BLOCKS &&
blocksAllocated < size; i++) { if
(FAT[i] == -1) { // Free block found
FAT[i] = 1; // Mark block as used
inode.blocks.push_back(i);
blocksAllocated++;
}
}
```

```
if (blocksAllocated < size) {
cout << "Not enough space.\n"; return
false;
}
files[name] = inode; cout << "File created: "
<< name << "\n"; return true;
}
// Delete a file by name bool
deleteFile(const string& name) { auto it =
files.find(name); if (it == files.end()) {
cout << "File not found.\n"; return false;
}
// Free blocks used by the file for
(int block : it->second.blocks) {
FAT[block] = -1; // Mark block as free
}
files.erase(it); cout << "File deleted: " <<
name << "\n"; return true;
}
// Read a file by name void readFile(const string&
name)
{ auto it = files.find(name); if (it ==
files.end()) { cout << "File not
found.\n"; return;
}
cout << "Reading file: " << name << ", Size: " << it->second.size << " blocks\n";
}
// Write additional data to a file by name (in blocks) bool
writeFile(const string& name, int additionalSize) { auto it
= files.find(name); if (it == files.end()) { cout
<< "File not found.\n"; return false;
```

```
}
int blocksAllocated = 0;
for (int i = 0; i < MAX_BLOCKS && blocksAllocated < additionalSize; i++)
{if (FAT[i] == -1) { FAT[i] = 1; it->second.blocks.push_back(i);
blocksAllocated++;
}
}
if (blocksAllocated < additionalSize) { cout << "Not enough
space to write additional data.\n"; return false;
}
it->second.size += additionalSize;
cout << "Data written to file: " << name << ", New Size: " << it->second.size << " blocks\n";
return true;
}
};
// Main function to demonstrate the file system operations int main() {
FileSystem fs;
// Perform basic file operations fs.createFile("file1", 10); // Create a file with 10
blocksfs.readFile("file1");
// Read the file fs.writeFile("file1", 5);
// Write additional 5 blocks fs.deleteFile("file1"); // Delete the file
return 0;
}
```

Output:-