Operating system day-1 lab experiments

1.Create a new process by invoking the appropriate system call. Get the process

identifier of the currently running process and its respective parent using system

calls and display the same using a C program.

CODE:s

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

int main() {

pid\_t pid, ppid;

pid = fork(); // Create a new process

if (pid < 0) {

perror("Fork failed");

exit(EXIT\_FAILURE);

} else if (pid == 0) {

ppid = getppid();

printf("Child Process ID: %d\n", getpid());

printf("Parent Process ID: %d\n", ppid);

} else {

printf("Parent Process ID: %d\n", getpid());

printf("Created Child Process ID: %d\n", pid);

}

return 0;

}

OUTPUT:Parent Process ID: 987

Created Child Process ID: 988

Child Process ID: 988

Parent Process ID: 987

2.Identify the system calls to copy the content of one file to another and illustrate

the same using a C program.

CODE:

#include <stdio.h>

#include <stdlib.h>

int main()

{

FILE \*sourceFile;

FILE \*destFile;

char sourcePath[100];

char destPath[100];

char ch;

printf("Enter source file path: ");

scanf("%s", sourcePath);

printf("Enter destination file path: ");

scanf("%s", destPath);

sourceFile = fopen(sourcePath, "r");

destFile = fopen(destPath, "w");

if (sourceFile == NULL || destFile == NULL)

{

printf("\nUnable to open file.\n");

printf("Please check if file exists and you have read/write privilege.\n");

exit(EXIT\_FAILURE);

}

ch = fgetc(sourceFile);

while (ch != EOF)

{

fputc(ch, destFile);

ch = fgetc(sourceFile);

}

printf("\nFiles copied successfully.\n");

fclose(sourceFile);

fclose(destFile);

return 0;

}

OUTPUT:

Enter source file path: OPERATING SYSTEM

Enter destination file path:

Unable to open file.

Please check if file exists and you have read/write privilege.

3.Design a CPU scheduling program with C using First Come First Served

technique with the following considerations.

a. All processes are activated at time 0.

b. Assume that no process waits on I/O devices

CODE:

#include <stdio.h>

struct Process {

int id;

int burst\_time;

};

void findWaitingTime(struct Process proc[], int n, int wait\_time[]) {

wait\_time[0] = 0;

for (int i = 1; i < n; i++) {

wait\_time[i] = proc[i - 1].burst\_time + wait\_time[i - 1];

}

}

void findTurnAroundTime(struct Process proc[], int n, int wait\_time[], int turn\_around\_time[]) {

for (int i = 0; i < n; i++) {

turn\_around\_time[i] = proc[i].burst\_time + wait\_time[i];

}

}

void findavgTime(struct Process proc[], int n) {

int wait\_time[n], turn\_around\_time[n];

findWaitingTime(proc, n, wait\_time);

findTurnAroundTime(proc, n, wait\_time, turn\_around\_time);

float total\_wait\_time = 0, total\_turn\_around\_time = 0;

for (int i = 0; i < n; i++) {

total\_wait\_time += wait\_time[i];

total\_turn\_around\_time += turn\_around\_time[i];

}

printf("Average waiting time: %.2f\n", total\_wait\_time / n);

printf("Average turn around time: %.2f\n", total\_turn\_around\_time / n);

}

int main() {

struct Process proc[] = { {1, 5}, {2, 3}, {3, 8} };

int n = sizeof(proc) / sizeof(proc[0]);

findavgTime(proc, n);

return 0;

}

OUTPUT:

Average waiting time: 4.33

Average turn around time: 9.67

4.Construct a scheduling program with C that selects the waiting process with the

smallest execution time to execute next

CODE:

#include <stdio.h>

#define MAX 10

typedef struct {

int id;

int burst\_time;

} Process;

void schedule(Process p[], int n) {

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (p[j].burst\_time > p[j + 1].burst\_time) {

Process temp = p[j];

p[j] = p[j + 1];

p[j + 1] = temp;

}

}

}

}

int main() {

Process p[MAX];

int n;

printf("Enter number of processes: ");

scanf("%d", &n);

for (int i = 0; i < n; i++) {

p[i].id = i + 1;

printf("Enter burst time for process %d: ", p[i].id);

scanf("%d", &p[i].burst\_time);

}

schedule(p, n);

printf("Process execution order:\n");

for (int i = 0; i < n; i++) {

printf("Process %d (Burst Time: %d)\n", p[i].id, p[i].burst\_time);

}

return 0;

}

OUTPUT:

Enter number of processes: 3

Enter burst time for process 1: 4

Enter burst time for process 2: 5

Enter burst time for process 3: 6

Process execution order:

Process 1 (Burst Time: 4)

Process 2 (Burst Time: 5)

Process 3 (Burst Time: 6)

5.Construct a scheduling program with C that selects the waiting process with the

highest priority to execute next.

CODE:

#include <stdio.h>

#include <stdlib.h>

#define MAX\_PROCESSES 10

typedef struct {

int id;

int priority;

} Process;

void schedule(Process processes[], int n) {

int highestPriorityIndex = 0;

for (int i = 1; i < n; i++) {

if (processes[i].priority > processes[highestPriorityIndex].priority) {

highestPriorityIndex = i;

}

}

printf("Executing Process ID: %d with Priority: %d\n",

processes[highestPriorityIndex].id,

processes[highestPriorityIndex].priority);

}

int main() {

Process processes[MAX\_PROCESSES] = {

{1, 2}, {2, 5}, {3, 1}, {4, 3}, {5, 4}

};

schedule(processes, 5);

return 0;

}

OUTPUT:

Executing Process ID: 2 with Priority: 5

6.Construct a C program to implement preemptive priority scheduling algorithm

CODE:

#include <stdio.h>

#include <stdlib.h>

struct Process {

int id, bt, pt, wt, tat;

};

void findWaitingTime(struct Process proc[], int n) {

int complete = 0, t = 0, min\_pt = 9999, min\_index;

while (complete < n) {

for (int i = 0; i < n; i++) {

if (proc[i].bt > 0 && proc[i].pt < min\_pt) {

min\_pt = proc[i].pt;

min\_index = i;

}

}

t++;

if (min\_pt != 9999) {

proc[min\_index].bt--;

if (proc[min\_index].bt == 0) {

proc[min\_index].tat = t;

proc[min\_index].wt = t - proc[min\_index].pt;

complete++;

}

}

min\_pt = 9999;

}

}

void findTurnAroundTime(struct Process proc[], int n) {

for (int i = 0; i < n; i++)

proc[i].tat += proc[i].bt;

}

void priorityScheduling(struct Process proc[], int n) {

findWaitingTime(proc, n);

findTurnAroundTime(proc, n);

printf("Process\tBurst Time\tPriority\tWaiting Time\tTurnaround Time\n");

for (int i = 0; i < n; i++)

printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", proc[i].id, proc[i].bt, proc[i].pt, proc[i].wt, proc[i].tat);

}

int main() {

struct Process proc[] = {{1, 10, 2}, {2, 5, 1}, {3, 8, 3}};

int n = sizeof(proc) / sizeof(proc[0]);

priorityScheduling(proc, n);

return 0;

}

OUTPUT":

Process Burst Time Priority Waiting Time Turnaround Time

1 0 2 13 15

2 0 1 4 5

3 0 3 20 23

7.Construct a C program to implement a non-preemptive SJF algorithm.

CODE:

#include<stdio.h>

int main()

{

int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,totalT=0,pos,temp;

float avg\_wt,avg\_tat;

printf("Enter number of process:");

scanf("%d",&n);

printf("\nEnter Burst Time:\n");

for(i=0;i<n;i++)

{

printf("p%d:",i+1);

scanf("%d",&bt[i]);

p[i]=i+1;

}

//sorting of burst times

for(i=0;i<n;i++)

{

pos=i;

for(j=i+1;j<n;j++)

{

if(bt[j]<bt[pos])

pos=j;

}

temp=bt[i];

bt[i]=bt[pos];

bt[pos]=temp;

temp=p[i];

p[i]=p[pos];

p[pos]=temp;

}

wt[0]=0;

//finding the waiting time of all the processes

for(i=1;i<n;i++)

{

wt[i]=0;

for(j=0;j<i;j++)

//individual WT by adding BT of all previous completed processes

wt[i]+=bt[j];

//total waiting time

total+=wt[i];

}

//average waiting time

avg\_wt=(float)total/n;

printf("\nProcess\t Burst Time \tWaiting Time\tTurnaround Time");

for(i=0;i<n;i++)

{

//turnaround time of individual processes

tat[i]=bt[i]+wt[i];

//total turnaround time

totalT+=tat[i];

printf("\np%d\t\t %d\t\t %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);

}

//average turnaround time

avg\_tat=(float)totalT/n;

printf("\n\nAverage Waiting Time=%f",avg\_wt);

printf("\nAverage Turnaround Time=%f",avg\_tat);

}

OUTPUT:

Enter number of process:5

Enter Burst Time:

p1:2

p2:3

p3:4

p4:5

p5:6

Process Burst Time Waiting Time Turnaround Time

p1 2 0 2

p2 3 2 5

p3 4 5 9

p4 5 9 14

p5 6 14 20

Average Waiting Time=6.000000

Average Turnaround Time=10.000000

8.Construct a C program to simulate Round Robin scheduling algorithm with C.

CODE:

#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[], int quantum) {

int rem\_bt[n];

for (int i = 0; i < n; i++) rem\_bt[i] = bt[i];

int t = 0;

while (1) {

int done = 1;

for (int i = 0; i < n; i++) {

if (rem\_bt[i] > 0) {

done = 0;

if (rem\_bt[i] > quantum) {

t += quantum;

rem\_bt[i] -= quantum;

} else {

t += rem\_bt[i];

wt[i] = t - bt[i];

rem\_bt[i] = 0;

}

}

}

if (done) break;

}

}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {

for (int i = 0; i < n; i++) tat[i] = bt[i] + wt[i];

}

void roundRobin(int processes[], int n, int bt[], int quantum) {

int wt[n], tat[n];

findWaitingTime(processes, n, bt, wt, quantum);

findTurnAroundTime(processes, n, bt, wt, tat);

printf("Process\tBurst Time\tWaiting Time\tTurnaround Time\n");

for (int i = 0; i < n; i++)

printf("%d\t%d\t\t%d\t\t%d\n", processes[i], bt[i], wt[i], tat[i]);

}

int main() {

int processes[] = {0, 1, 2};

int bt[] = {10, 5, 8};

int quantum = 3;

int n = sizeof(processes) / sizeof(processes[0]);

roundRobin(processes, n, bt, quantum);

return 0;

}

OUTPUT:

Process Burst Time Waiting Time Turnaround Time

0 10 13 23

1 5 9 14

2 8 14 22

9.Illustrate the concept of inter-process communication using shared memory with

a C program.

CODE:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <unistd.h>

#define SHM\_SIZE 1024

int main() {

key\_t key = ftok("shmfile", 65);

int shmid = shmget(key, SHM\_SIZE, 0666 | IPC\_CREAT);

if (shmid == -1) {

perror("shmget failed");

exit(1);

}

char \*shm\_addr = (char \*)shmat(shmid, NULL, 0);

if (shm\_addr == (char \*)(-1)) {

perror("shmat failed");

exit(1);

}

pid\_t pid = fork();

if (pid < 0) {

perror("fork failed");

exit(1);

}

if (pid == 0) {

sleep(1);

printf("Reader Process: Read from shared memory: %s\n", shm\_addr);

shmdt(shm\_addr);

shmctl(shmid, IPC\_RMID, NULL);

} else {

char message[] = "Hello from Shared Memory!";

strcpy(shm\_addr, message);

printf("Writer Process: Written to shared memory: %s\n", message);

shmdt(shm\_addr);

}

return 0;

}

OUTPUT:

Writer Process: Written to shared memory: Hello from Shared Memory!

10.Illustrate the concept of inter-process communication using message queue with

a C program.

CODE:

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <string.h>

#include <unistd.h>

#define MSG\_SIZE 100

struct msg\_buffer {

long msg\_type;

char msg\_text[MSG\_SIZE];

};

int main() {

key\_t key = ftok("progfile", 65);

int msgid = msgget(key, 0666 | IPC\_CREAT);

struct msg\_buffer message;

message.msg\_type = 1;

strcpy(message.msg\_text, "Hello, World!");

msgsnd(msgid, &message, sizeof(message), 0);

msgrcv(msgid, &message, sizeof(message), 1, 0);

printf("Received: %s\n", message.msg\_text);

msgctl(msgid, IPC\_RMID, NULL);

return 0;

}

OUTPUT:

Received: Hello, World!