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.

// Online C compiler to run C program online

#include <stdio.h>

#include<unistd.h>

int main() {

pid\_t pid , parent\_pid;

parent\_pid = getpid();

pid = fork();

if(pid<0)

{

perror("fork failed");

return 1;

}

if(pid == 0)

{

printf("child process:");

printf("pid of child : %d\n", getpid());

printf("pid of parent : %d\n", getppid());

}

else

{

printf("parent process:");

printf("pid of parent: %d\n", getpid());

printf("pid of parent of parent : %d\n", getppid());

}

}

Output:

Parent Process:

PID of parent: 6263

PID of parent of parent: 6256

Child Process:

PID of child: 6264

PID of parent: 6263

2.Identify the system calls to copy the content of one file to another and illustrate the same using a C program.

#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>

#include <errno.h>

#define BUFSIZE 1024

int main(int argc, char \*argv[]) {

int src\_fd, dest\_fd, n\_read;

char buffer[BUFSIZE];

// Check if source and destination files are provided

if (argc != 3) {

write(2, "Usage: <source\_file> <destination\_file>\n", 41);

return 1;

}

// Open source file in read-only mode

src\_fd = open(argv[1], O\_RDONLY);

if (src\_fd == -1) {

perror("Error opening source file");

return 1;

}

// Open destination file in write-only mode, create it if it doesn't exist

dest\_fd = open(argv[2], O\_WRONLY | O\_CREAT | O\_TRUNC, 0644);

if (dest\_fd == -1) {

perror("Error opening destination file");

close(src\_fd);

return 1;

}

// Read from source file and write to destination file

while ((n\_read = read(src\_fd, buffer, BUFSIZE)) > 0) {

if (write(dest\_fd, buffer, n\_read) != n\_read) {

perror("Error writing to destination file");

close(src\_fd);

close(dest\_fd);

return 1;

}

}

// Check for read error

if (n\_read == -1) {

perror("Error reading from source file");

}

// Close the files

close(src\_fd);

close(dest\_fd);

return 0;

}

Output:

Usage: <source\_file> <destination\_file>

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

#include <stdio.h>

typedef struct {

int processID; // Process ID

int burstTime; // Burst time of the process

int arrivalTime; // Arrival time of the process

int waitingTime; // Waiting time of the process

int turnaroundTime; // Turnaround time of the process

int completionTime; // Completion time of the process

} Process;

// Function to calculate waiting time, turnaround time, and completion time

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

processes[0].waitingTime = 0; // First process has 0 waiting time

processes[0].completionTime = processes[0].burstTime; // Completion time for first process is its burst time

processes[0].turnaroundTime = processes[0].burstTime; // Turnaround time for first process is its burst time

// Calculate times for remaining processes

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

// Waiting time for each process is the completion time of the previous process minus arrival time

processes[i].waitingTime = processes[i - 1].completionTime - processes[i].arrivalTime;

// Turnaround time is waiting time plus burst time

processes[i].turnaroundTime = processes[i].waitingTime + processes[i].burstTime;

// Completion time is waiting time plus burst time

processes[i].completionTime = processes[i].waitingTime + processes[i].burstTime + processes[i].arrivalTime;

}

}

// Function to calculate average waiting time and turnaround time

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

int totalWaitingTime = 0;

int totalTurnaroundTime = 0;

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

totalWaitingTime += processes[i].waitingTime;

totalTurnaroundTime += processes[i].turnaroundTime;

}

printf("\nAverage Waiting Time: %.2f\n", (float)totalWaitingTime / n);

printf("Average Turnaround Time: %.2f\n", (float)totalTurnaroundTime / n);

}

// Function to print the process schedule

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

printf("\nProcessID\tArrivalTime\tBurstTime\tWaitingTime\tTurnaroundTime\tCompletionTime\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\n",

processes[i].processID,

processes[i].arrivalTime,

processes[i].burstTime,

processes[i].waitingTime,

processes[i].turnaroundTime,

processes[i].completionTime);

}

}

// Function to sort the processes based on arrival time (ascending order)

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

Process temp;

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

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

if (processes[i].arrivalTime > processes[j].arrivalTime) {

// Swap the processes

temp = processes[i];

processes[i] = processes[j];

processes[j] = temp;

}

}

}

}

// Main function to execute the FCFS Scheduling

int main() {

int n;

// Get the number of processes

printf("Enter the number of processes: ");

scanf("%d", &n);

Process processes[n];

// Input arrival time and burst time for each process

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

processes[i].processID = i + 1;

printf("Enter arrival time for Process %d: ", i + 1);

scanf("%d", &processes[i].arrivalTime);

printf("Enter burst time for Process %d: ", i + 1);

scanf("%d", &processes[i].burstTime);

}

// Sort processes based on arrival time

sortByArrival(processes, n);

// Calculate waiting time, turnaround time, and completion time for each process

calculateTimes(processes, n);

// Print the process schedule

printSchedule(processes, n);

// Calculate and print the averages

calculateAverages(processes, n);

return 0;

}

3.Construct a scheduling program with C that selects the waiting process with the highest priority to execute next.

#include <stdio.h>

#include <stdlib.h>

// Structure to represent a process

struct Process {

int id; // Process ID

int priority; // Process priority (lower number means higher priority)

int burst\_time; // Burst time (time required for execution)

};

// Function to compare two processes based on priority

int compare\_priority(const void \*a, const void \*b) {

return ((struct Process\*)a)->priority - ((struct Process\*)b)->priority;

}

// Function to simulate the execution of processes

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

// Sort the processes based on priority (highest priority first)

qsort(processes, n, sizeof(struct Process), compare\_priority);

printf("Scheduling processes based on priority:\n");

// Simulate the execution of processes

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

printf("Executing Process ID: %d with Priority: %d for %d units of time\n",

processes[i].id, processes[i].priority, processes[i].burst\_time);

}

}

int main() {

int n;

// Read the number of processes

printf("Enter the number of processes: ");

scanf("%d", &n);

struct Process processes[n];

// Read process details

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

processes[i].id = i + 1; // Assigning unique ID starting from 1

printf("Enter priority and burst time for Process %d: ", i + 1);

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

}

// Schedule the processes based on their priority

schedule\_processes(processes, n);

return 0;

}

3(question reference)

// Online C compiler to run C program online

#include <stdio.h>

#include<stdlib.h>

int comp(const void\* a, const void\* b)

{

return \*(int \*)a - \*(int \*)b;

}

int main() {

int a[]={1,5,3,4,2};

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

qsort(a, n, sizeof(int), comp);

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

{

printf("%d\t",a[i]);

}

return 0;

}

THREADS

CREATION

#include <stdio.h>

#include <pthread.h>

void \*thread\_function(void \*arg) {

printf("Thread is running\n");

return NULL;

}

int main() {

pthread\_t thread\_id;

// Create a thread

if (pthread\_create(&thread\_id, NULL, thread\_function, NULL) != 0) {

perror("Failed to create thread");

return 1;

}

printf("Main thread\n");

// Wait for the thread to finish

pthread\_join(thread\_id, NULL);

return 0;

}

JOIN

#include <stdio.h>

#include <pthread.h>

void \*thread\_function(void \*arg) {

printf("Thread started\n");

return NULL;

}

int main() {

pthread\_t thread\_id;

// Create the thread

pthread\_create(&thread\_id, NULL, thread\_function, NULL);

printf("Main thread waiting for thread to finish\n");

// Join the thread

pthread\_join(thread\_id, NULL);

printf("Main thread finished\n");

return 0;

}

EQUAL

#include <stdio.h>

#include <pthread.h>

void \*thread\_function(void \*arg) {

pthread\_t self\_id = pthread\_self(); // Get the current thread's ID

printf("Thread ID inside thread: %lu\n", self\_id);

return NULL;

}

int main() {

pthread\_t thread\_id;

pthread\_create(&thread\_id, NULL, thread\_function, NULL);

pthread\_t main\_thread\_id = pthread\_self(); // Get the main thread's ID

// Check if main thread and created thread are the same

if (pthread\_equal(main\_thread\_id, thread\_id)) {

printf("Main thread and created thread are the same!\n");

} else {

printf("Main thread and created thread are different.\n");

}

pthread\_join(thread\_id, NULL);

return 0;

}

EXIT

#include <stdio.h>

#include <pthread.h>

void \*thread\_function(void \*arg) {

printf("Thread is exiting\n");

pthread\_exit(NULL); // Exit the thread explicitly

}

int main() {

pthread\_t thread\_id;

// Create a thread

pthread\_create(&thread\_id, NULL, thread\_function, NULL);

printf("Main thread waiting for created thread\n");

// Wait for the thread to exit

pthread\_join(thread\_id, NULL);

printf("Main thread finished\n");

return 0;

}

MULTITHREADS

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

# define num 5

pthread\_mutex\_t mutex;

int counter=0;

void\* increment(void\* threadid)

{

long tid=(long)threadid;

pthread\_mutex\_lock(&mutex);

counter++;

printf("threat %d incremented to %d",tid,counter);

pthread\_mutex\_unlock(&mutex);

pthread\_exit(NULL);

}

int main()

{

pthread\_t threads[num];

pthread\_mutex\_init(&mutex,NULL);

for(long i=0;i<num;i++)

{

printf("create threats:%ld\n",i);

pthread\_create(&threads[i],NULL,increment,(void\*)i);

}

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

{

pthread\_join(threads[i],NULL);

}

pthread\_mutex\_destroy(&mutex);

printf("final counter value:%d\n",counter);

}

FILE MANAGEMENT

#include<stdio.h>

#include<stdlib.h>

int main()

{

FILE \*filepointer;

char filename[]="example.txt";

char data[100];

filepointer=fopen(filename,"w");

if(filepointer==NULL)

{

printf("error openiing file for writing");

return 1;

}

fprintf(filepointer,"hello");

fprintf(filepointer,"hello 2");

fclose(filepointer);

filepointer=fopen(filename,"r");

if(filepointer==NULL)

{

printf("error openiing file for writting");

return 1;

}

printf("content of the file");

while(fgets(data,sizeof(data),filepointer)!=NULL){

printf("%s",data);

}

fclose(filepointer);

}

FCFS

#include <stdio.h>

#include <stdlib.h>

int main() {

int n, i, total\_head\_movement = 0, current\_position;

// Input: Current head position

printf("Enter the current head position: ");

scanf("%d", &current\_position);

// Input: Number of requests

printf("Enter the number of requests: ");

scanf("%d", &n);

int requests[n];

// Input: Request order

printf("Enter the request order:\n");

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

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

}

// Start servicing requests

printf("%d", current\_position); // Print initial head position

total\_head\_movement += abs(current\_position - requests[0]); // Calculate movement to first request

printf(" -> %d", requests[0]); // Print first request

current\_position = requests[0]; // Update current position

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

total\_head\_movement += abs(current\_position - requests[i]); // Calculate movement to next request

printf(" -> %d", requests[i]); // Print next request

current\_position = requests[i]; // Update current position

}

printf("\nTotal head movement = %d\n", total\_head\_movement); // Output total movement

return 0;

}