Write a shell program to **display Area and Circumference of a Circle** and calculate **the gross pay** of an employee.

#shell program to calculate the area and circumference of the circle

echo enter the radius

read r

area=`expr 22 / 7 \\* $r \\* $r`

circumference=`expr 2 \\* 22 / 7 \\* $r`

echo area= $area

echo circumference= $circumference

#shell program to find the gross salary

echo Enter the employee name

read name

echo enter the basic salary

read s

da=`expr $s \\* 47 / 100` hra=`expr $s \\* 12 / 100`

cca=`expr $s \\* 3 / 100`

gross=`expr $s + $hra + $cca + $da`

echo The gross salary of $name is $gross

Write a shell program to **Check the given number is positive, negative** and calculate the gross pay of an employee

echo "Enter a Number"

read num

if [ $num -lt 0 ]

then

echo "Negative"

elif [ $num -gt 0 ]

then

echo "Positive"

else

echo "Neither Positive Nor Negative"

fi

Implement basic **Unix Commands**

Syn:$cal 2 2009

Syn:$echo “text”

Syn:$lsls–s

Syn:$lp filename

Syn:$man unix command

$man cat

Syn:$who$whoami

Syn:$uptime

Syn:$uname–a

Syn:$ hostname

$bc

10/2\*3

C program to implement **Process Synchronization using Semaphores**

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#include <unistd.h>

sem\_t mutex;

int bal = 500;

void\* threada(void\* arg) {

// Wait

sem\_wait(&mutex);

printf("\nThread1 Entered\n");

bal = bal - 100;

printf("Thread1 - New Balance: %d\n", bal);

// Critical section

sleep(3); // Simulate work

// Signal

printf("Thread1 Exit\n");

sem\_post(&mutex);

return NULL;

}

void\* threadb(void\* arg) {

// Wait

sem\_wait(&mutex);

printf("\nThread2 Entered\n");

bal = bal - 50;

printf("Thread2 - New Balance: %d\n", bal);

// Critical section

sleep(3); // Simulate work

// Signal

printf("Thread2 Exit\n");

sem\_post(&mutex);

return NULL;

}

int main() {

sem\_init(&mutex, 0, 1); // Initialize semaphore with value 1 (mutex)

pthread\_t t1, t2;

// Create thread t1 (threada)

if (pthread\_create(&t1, NULL, threada, NULL) != 0) {

perror("pthread\_create");

return 1;

}

C program to **implement  IPC using Message Queue.**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

struct msg\_buffer {

long msg\_type;

int data;

};

int isPrime(int n) {

if (n <= 1) return 0;

for (int i = 2; i \* i <= n; i++) {

if (n % i == 0) return 0;

}

return 1;

}

void sender() {

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

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

int inputData;

printf("Enter an integer to send: ");

scanf("%d", &inputData);

struct msg\_buffer message = {1, inputData};

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

struct msg\_buffer response;

msgrcv(msgid, &response, sizeof(response), 2, 0);

printf("Received status from receiver: %s\n", response.data ? "Prime" : "Not Prime");

msgctl(msgid, IPC\_RMID, NULL); // Remove message queue

}

void receiver() {

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

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

struct msg\_buffer message;

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

int isPrimeResult = isPrime(message.data);

message.msg\_type = 2;

message.data = isPrimeResult;

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

}

int main() {

pid\_t pid = fork();

if (pid == -1) {

fprintf(stderr, "Fork failed\n");

exit(EXIT\_FAILURE);

}

if (pid > 0) {

// Parent process (sender)

sender();

} else {

// Child process (receiver)

receiver();

}

return 0;

}

C program to implement **Bankers Algorithm for Deadlock Avoidance.**

#include <stdio.h>

int main() {

int n, m, i, j, k;

n = 5; // Number of processes

m = 3; // Number of resources

// Allocation Matrix

int alloc[5][3] = {

{0, 1, 0}, // P0

{2, 0, 0}, // P1

{3, 0, 2}, // P2

{2, 1, 1}, // P3

{0, 0, 2} // P4

};

// Maximum Matrix

int max[5][3] = {

{7, 5, 3}, // P0

{3, 2, 2}, // P1

{9, 0, 2}, // P2

{2, 2, 2}, // P3

{4, 3, 3} // P4

};

// Available Resources

int avail[3] = {3, 3, 2};

int f[n], ans[n], ind = 0;

// Initialize the finish array

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

f[k] = 0;

}

// Calculate the Need matrix

int need[n][m];

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

for (j = 0; j < m; j++) {

need[i][j] = max[i][j] - alloc[i][j];

}

}

// Find a safe sequence using the Banker's algorithm

int y = 0;

for (k = 0; k < 5; k++) {

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

if (f[i] == 0) {

int flag = 0;

for (j = 0; j < m; j++) {

if (need[i][j] > avail[j]) {

flag = 1;

break;

}

}

if (flag == 0) {

// Process i can be executed

ans[ind++] = i;

for (y = 0; y < m; y++) {

avail[y] += alloc[i][y];

}

f[i] = 1; // Mark process i as finished

}

}

}

}

// Check if all processes are finished

int flag = 1;

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

if (f[i] == 0) {

flag = 0; // System is not safe

printf("The following system is not safe\n");

break;

}

}

// Print the safe sequence if the system is safe

if (flag == 1) {

printf("Following is the SAFE Sequence\n");

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

printf(" P%d ->", ans[i]);

}

printf(" P%d\n", ans[n - 1]);

}

return 0;

}

 C program to implement **LRU Page Replacement Algorithm.**

#include<stdio.h>

int main() {

int i, j, k, min, rs[25], m[10], count[10], flag[25], n, f, pf = 0, next = 1;

printf("Enter the length of reference string: ");

scanf("%d", &n);

printf("Enter the reference string: ");

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

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

flag[i] = 0;

}

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

scanf("%d", &f);

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

count[i] = 0;

m[i] = -1;

}

printf("\nThe Page Replacement process is: \n");

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

for (j = 0; j < f; j++) {

if (m[j] == rs[i]) {

flag[i] = 1;

count[j] = next;

next++;

}

}

if (flag[i] == 0) {

if (i < f) {

m[i] = rs[i];

count[i] = next;

next++;

} else {

min = 0;

for (j = 1; j < f; j++) {

if (count[min] > count[j])

min = j;

}

m[min] = rs[i];

count[min] = next;

next++;

}

pf++;

}

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

printf("%d\t", m[j]);

if (flag[i] == 0)

printf("PF No. -- %d", pf);

printf("\n");

}

printf("\nThe number of page faults using LRU is %d\n", pf);

return 0;

}

 C program to implement **FIFO Page Replacement Algorithm.**

#include<stdio.h>

int main() {

int i, j, k, f, pf = 0, count = 0, rs[25], m[10], n;

printf("\n Enter the length of reference string: ");

scanf("%d", &n);

printf("\n Enter the reference string: ");

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

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

printf("\n Enter the number of frames: ");

scanf("%d", &f);

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

m[i] = -1;

printf("\n The Page Replacement Process is: \n");

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

int pageFound = 0;

// Check if the page is already in memory

for (k = 0; k < f; k++) {

if (m[k] == rs[i]) {

pageFound = 1;

break;

}

}

// If page is not found in memory, perform page replacement

if (!pageFound) {

m[count++] = rs[i];

pf++;

}

// Print the current state of memory

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

printf("\t%d", m[j]);

// Print the page fault number if there's a page fault

if (!pageFound)

printf("\tPF No. %d", pf);

printf("\n");

// Reset the frame pointer if it reaches the frame limit

if (count == f)

count = 0;

}

printf("\n The number of Page Faults using FIFO is %d\n", pf);

return 0;

}

 Program to **implement the following action using pthreads**

#include <iostream>

#include <pthread.h>

// Structure to hold data passed to the child thread

struct ThreadData {

int start;

int end;

};

// Function to print odd or even numbers based on the passed argument

void\* printNumbers(void\* arg) {

ThreadData\* data = (ThreadData\*)arg;

int start = data->start;

int end = data->end;

// Print odd or even numbers based on the start and end range

for (int i = start; i <= end; i++) {

if ((i % 2) == (start % 2)) { // Check if i and start are both odd or even

std::cout << i << " ";

}

}

std::cout << std::endl;

// Thread exits

pthread\_exit(NULL);

}

int main() {

pthread\_t childThread;

ThreadData data = {1, 10}; // Range of numbers from 1 to 10

// Create child thread

int result = pthread\_create(&childThread, NULL, printNumbers, (void\*)&data);

if (result) {

std::cerr << "Error creating thread: " << result << std::endl;

return -1;

}

// Wait for the child thread to finish

pthread\_join(childThread, NULL);

std::cout << "Child thread has finished." << std::endl;

return 0;

}

C program to implement **Optimal Page Replacement Algorithm.**

#include <stdio.h>

// Function to find the index of the maximum value in an array

int findMaxIndex(int a[], int n) {

int max = a[0];

int maxIndex = 0;

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

if (a[i] > max) {

max = a[i];

maxIndex = i;

}

}

return maxIndex;

}

int main() {

int seq[30], fr[5], pos[5];

int max, n, pf = 0;

printf("Enter maximum limit of the sequence: ");

scanf("%d", &max);

printf("Enter the sequence: ");

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

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

printf("Enter number of frames: ");

scanf("%d", &n);

int count = 0; // Count of frames filled

printf("Page Frames:\n");

// Iterate over the sequence

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

int page = seq[i];

int found = 0;

// Check if the page is already in the frames

for (int j = 0; j < count; j++) {

if (fr[j] == page) {

found = 1;

break;

}

}

if (!found) {

if (count < n) {

fr[count] = page;

count++;

pf++;

} else {

// Find the page with the maximum distance in the future

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

pos[j] = 0;

for (int k = i + 1; k < max; k++) {

if (fr[j] == seq[k]) {

pos[j] = k;

break;

}

}

}

int replaceIndex = findMaxIndex(pos, n);

fr[replaceIndex] = page;

pf++;

}

}

// Print current state of page frames

for (int j = 0; j < count; j++) {

printf("%d\t", fr[j]);

}

printf("\n");

}

float pfr = (float)pf / (float)max;

printf("\nThe number of page faults: %d\n", pf);

printf("Page fault rate: %f\n", pfr);

return 0;

}

 Write a program to implement the following action using pthreads

Create a thread in the main program, this program passes the 'count' as an argument to that thread function and this created thread function has to print your name 'count' times.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <string.h>

// Function executed by the thread

void\* printName(void\* arg) {

char\* name = (char\*)arg;

int count = \*((int\*)arg + 1); // Extract count from the argument

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

printf("%s\n", name);

}

// Thread exits

pthread\_exit(NULL);

}

int main() {

pthread\_t thread;

int count = 5; // Number of times to print the name

char name[] = "John"; // Your name

// Create a thread and pass 'count' and 'name' as arguments

void\* args[] = { (void\*)name, (void\*)&count };

int result = pthread\_create(&thread, NULL, printName, (void\*)args);

if (result) {

fprintf(stderr, "Error creating thread: %d\n", result);

return 1;

}

// Wait for the thread to finish

pthread\_join(thread, NULL);

printf("Thread finished execution.\n");

return 0;

}

 C program to implement **FCFS  CPU scheduling** algorithm

#include <stdio.h>

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

wt[0] = 0; // Waiting time for the first process is always 0

// Calculate waiting time for remaining processes

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

wt[i] = wt[i - 1] + bt[i - 1];

}

}

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

// Calculate turnaround time by adding burst time and waiting time

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

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

}

}

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

printf("\nGantt Chart:\n");

printf("-------------------------------------------------------------\n");

printf("| Process |");

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

printf(" P%d |", processes[i]);

}

printf("\n-------------------------------------------------------------\n");

printf("| Time |");

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

printf(" %d |", wt[i]);

}

printf("\n-------------------------------------------------------------\n");

}

void findAvgTime(int processes[], int n, int bt[]) {

int wt[n], tat[n];

float total\_wt = 0, total\_tat = 0;

// Calculate waiting time of each process

findWaitingTime(processes, n, bt, wt);

// Calculate turnaround time of each process

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

// Display processes along with their respective waiting and turnaround times

printf("Process Burst Time Waiting Time Turnaround Time\n");

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

total\_wt += wt[i];

total\_tat += tat[i];

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

}

// Calculate average waiting and turnaround times

float avg\_wt = total\_wt / n;

float avg\_tat = total\_tat / n;

printf("\nAverage Waiting Time: %.2f\n", avg\_wt);

printf("Average Turnaround Time: %.2f\n", avg\_tat);

// Display Gantt Chart

findGanttChart(processes, n, bt, wt);

}

int main() {

int n;

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

scanf("%d", &n);

int processes[n], burst\_time[n];

printf("Enter Burst Time for each process:\n");

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

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

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

processes[i] = i + 1; // Assigning process IDs

}

findAvgTime(processes, n, burst\_time);

return 0;

}

 C program to implement  **SJF CPU scheduling** algorithm

#include <stdio.h>

int main() {

int p[20], bt[20], wt[20], tat[20], i, k, n, temp;

float wtavg = 0.0, tatavg = 0.0;

printf("\nEnter the number of processes: ");

scanf("%d", &n);

// Input burst time for each process

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

p[i] = i + 1; // Process ID (P1, P2, ...)

printf("Enter Burst Time for Process %d: ", i + 1);

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

}

// Sort processes based on burst time using bubble sort

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

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

if (bt[k] > bt[k + 1]) {

// Swap burst times

temp = bt[k];

bt[k] = bt[k + 1];

bt[k + 1] = temp;

// Swap process IDs accordingly

temp = p[k];

p[k] = p[k + 1];

p[k + 1] = temp;

}

}

}

// Calculate waiting time (wt) and turnaround time (tat)

wt[0] = 0;

tat[0] = bt[0];

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

wt[i] = wt[i - 1] + bt[i - 1];

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

wtavg += wt[i];

tatavg += tat[i];

}

// Calculate averages

wtavg = wtavg / n;

tatavg = tatavg / n;

// Display process details and calculated metrics

printf("\nPROCESS\tBURST TIME\tWAITING TIME\tTURNAROUND TIME\n");

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

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

}

printf("\nAverage Waiting Time: %.2f", wtavg);

printf("\nAverage Turnaround Time: %.2f\n", tatavg);

// Wait for user input before exiting

printf("\nPress Enter to exit...");

while (getchar() != '\n'); // Clear input buffer

getchar(); // Wait for Enter key press

return 0;

}

C program to implement **Priority CPU scheduling** algorithm

#include <stdio.h>

#include <stdlib.h> // Include stdlib.h for exit() function

int main() {

int p[20], bt[20], pri[20], wt[20], tat[20], i, k, n, temp;

float wtavg = 0.0, tatavg = 0.0; // Initialize floating-point variables

// Prompt for the number of processes

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

scanf("%d", &n);

// Input burst time and priority for each process

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

p[i] = i;

printf("Enter the Burst Time & Priority of Process %d: ", i);

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

}

// Sort processes based on priority using bubble sort

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

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

if (pri[i] > pri[k]) {

// Swap process details (priority, burst time)

temp = p[i];

p[i] = p[k];

p[k] = temp;

temp = bt[i];

bt[i] = bt[k];

bt[k] = temp;

temp = pri[i];

pri[i] = pri[k];

pri[k] = temp;

}

}

}

// Calculate waiting time (wt) and turnaround time (tat)

wt[0] = 0;

tat[0] = bt[0];

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

wt[i] = wt[i - 1] + bt[i - 1];

tat[i] = tat[i - 1] + bt[i];

wtavg += wt[i]; // Accumulate waiting times for average calculation

tatavg += tat[i]; // Accumulate turnaround times for average calculation

}

// Calculate averages

wtavg = wtavg / n;

tatavg = tatavg / n;

// Display process details and calculated metrics

printf("\nPROCESS\tPRIORITY\tBURST TIME\tWAITING TIME\tTURNAROUND TIME\n");

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

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

}

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

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

// Wait for a key press (for old compilers, might use getch() for modern ones)

// getch();

return 0;

}

Write a program for Unix System Calls **fork() and getpid() and exit()**

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h> // Include stdlib.h for exit() function

int main() {

int pid, pid1, pid2;

int userPID;

// Prompt the user to enter a process ID (PID)

printf("Enter the process ID (PID) to fork from: ");

scanf("%d", &userPID);

// Attempt to create a child process using fork() with the specified PID

pid = fork();

if (pid == -1) {

// Error handling if fork() fails

perror("ERROR IN PROCESS CREATION");

exit(1);

}

if (pid != 0) {

// Parent process block

pid1 = getpid();

printf("\nThe parent process ID is %d\n", pid1);

} else {

// Child process block

pid2 = getpid();

printf("\nThe child process ID is %d\n", pid2);

}

return 0;

}

  Write a C program to implement **Round Robin CPU scheduling** algorithm

#include <stdio.h>

int main() {

int i, j, n, bu[10], wa[10], tat[10], ct[10], t, max;

float awt = 0, att = 0, temp = 0;

// Clear screen (for old compilers, might use system("cls") for modern ones)

// clrscr();

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

scanf("%d", &n);

// Input burst times for each process

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

printf("Enter Burst Time for process %d: ", i + 1);

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

ct[i] = bu[i]; // Store burst time in ct array for later calculations

}

printf("Enter the size of time slice (quantum): ");

scanf("%d", &t);

// Find the process with the maximum burst time

max = bu[0];

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

if (max < bu[i]) {

max = bu[i];

}

}

// Round Robin Scheduling

for (j = 0; j < (max / t) + 1; j++) {

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

if (bu[i] != 0) {

if (bu[i] <= t) {

tat[i] = temp + bu[i];

temp += bu[i];

bu[i] = 0;

} else {

bu[i] -= t;

temp += t;

}

}

}

}

// Calculate waiting time (wa) and turnaround time (tat)

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

wa[i] = tat[i] - ct[i];

awt += wa[i];

att += tat[i];

}

// Calculate averages

awt /= n;

att /= n;

// Output results

printf("\nThe Average Turnaround time is: %f", att);

printf("\nThe Average Waiting time is: %f", awt);

// Display detailed process information

printf("\nPROCESS\t BURST TIME \t WAITING TIME \t TURNAROUND TIME\n");

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

printf("%d \t %d \t\t %d \t\t %d \n", i + 1, ct[i], wa[i], tat[i]);

}

// Wait for a key press (for old compilers, might use getch() for modern ones)

// getch();

return 0;

}

C program to **implement IPC using Message Queue.**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

// Define message buffer structure

struct msg\_buffer {

long msg\_type;

int data;

};

// Function to check if a number is prime

int isPrime(int n) {

if (n <= 1) return 0;

for (int i = 2; i \* i <= n; i++) {

if (n % i == 0) return 0;

}

return 1;

}

// Sender function

void sender() {

// Create or get the message queue

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

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

// Prompt user to enter an integer

int inputData;

printf("Enter an integer to send: ");

scanf("%d", &inputData);

// Prepare message to send

struct msg\_buffer message = {1, inputData};

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

// Receive and display result from receiver

msgrcv(msgid, &message, sizeof(message), 2, 0);

printf("Received status from receiver: %s\n", message.data ? "Prime" : "Not Prime");

// Remove the message queue

msgctl(msgid, IPC\_RMID, NULL);

}

// Receiver function

void receiver() {

// Create or get the message queue

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

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

// Receive message from sender

struct msg\_buffer message;

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

// Check if received number is prime

int isPrimeResult = isPrime(message.data);

// Prepare and send result back to sender

message.msg\_type = 2;

message.data = isPrimeResult;

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

}

// Main function

int main() {

// Create child process

pid\_t pid = fork();

// Check fork result

if (pid == -1) {

fprintf(stderr, "Fork failed\n");

exit(EXIT\_FAILURE);

}

// Execute sender or receiver based on process ID

if (pid > 0) {

sender(); // Parent process (sender)

} else {

receiver(); // Child process (receiver)

}

return 0;

}

Write a program for Memory Management Techniques- **First Fit, Best fit and worst  fit**

#include <stdio.h>

#define MAX\_PROCESS 10

#define MAX\_MEMORY\_BLOCK 10

// Function prototypes

void firstFit(int process[], int m, int block[], int n);

void bestFit(int process[], int m, int block[], int n);

void worstFit(int process[], int m, int block[], int n);

int main() {

int process[MAX\_PROCESS], block[MAX\_MEMORY\_BLOCK];

int m, n;

printf("Enter number of processes: ");

scanf("%d", &m);

printf("Enter sizes of processes:\n");

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

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

}

printf("Enter number of memory blocks: ");

scanf("%d", &n);

printf("Enter sizes of memory blocks:\n");

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

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

}

printf("\nFirst Fit:\n");

firstFit(process, m, block, n);

printf("\nBest Fit:\n");

bestFit(process, m, block, n);

printf("\nWorst Fit:\n");

worstFit(process, m, block, n);

return 0;

}

// First Fit Allocation

void firstFit(int process[], int m, int block[], int n) {

int allocation[m];

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

allocation[i] = -1;

}

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

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

if (block[j] >= process[i]) {

allocation[i] = j;

block[j] -= process[i];

break;

}

}

}

printf("Process No.\tProcess Size\tBlock No.\n");

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

printf("%d\t\t%d\t\t", i + 1, process[i]);

if (allocation[i] != -1) {

printf("%d\n", allocation[i] + 1);

} else {

printf("Not Allocated\n");

}

}

}

// Best Fit Allocation

void bestFit(int process[], int m, int block[], int n) {

int allocation[m];

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

allocation[i] = -1;

}

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

int bestIdx = -1;

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

if (block[j] >= process[i]) {

if (bestIdx == -1 || block[j] < block[bestIdx]) {

bestIdx = j;

}

}

}

if (bestIdx != -1) {

allocation[i] = bestIdx;

block[bestIdx] -= process[i];

}

}

printf("Process No.\tProcess Size\tBlock No.\n");

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

printf("%d\t\t%d\t\t", i + 1, process[i]);

if (allocation[i] != -1) {

printf("%d\n", allocation[i] + 1);

} else {

printf("Not Allocated\n");

}

}

}

// Worst Fit Allocation

void worstFit(int process[], int m, int block[], int n) {

int allocation[m];

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

allocation[i] = -1;

}

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

int worstIdx = -1;

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

if (block[j] >= process[i]) {

if (worstIdx == -1 || block[j] > block[worstIdx]) {

worstIdx = j;

}

}

}

if (worstIdx != -1) {

allocation[i] = worstIdx;

block[worstIdx] -= process[i];

}

}

printf("Process No.\tProcess Size\tBlock No.\n");

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

printf("%d\t\t%d\t\t", i + 1, process[i]);

if (allocation[i] != -1) {

printf("%d\n", allocation[i] + 1);

} else {

printf("Not Allocated\n");

}

}

}

C program to implement **indexed file allocation strategy**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <string.h>

// Function executed by the thread

void\* printName(void\* arg) {

char\* name = (char\*)arg;

int count = \*((int\*)arg + 1); // Extract count from the argument

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

printf("%s\n", name);

}

// Thread exits

pthread\_exit(NULL);

}

int main() {

pthread\_t thread;

int count = 5; // Number of times to print the name

char name[] = "John"; // Your name

// Create a thread and pass 'count' and 'name' as arguments

void\* args[] = { (void\*)name, (void\*)&count };

int result = pthread\_create(&thread, NULL, printName, (void\*)args);

if (result) {

fprintf(stderr, "Error creating thread: %d\n", result);

return 1;

}

// Wait for the thread to finish

pthread\_join(thread, NULL);

printf("Thread finished execution.\n");

return 0;

}