**# Producer Consumer Problem**  
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define BUFFER\_SIZE 5 // Size of the buffer

int buffer[BUFFER\_SIZE]; // Shared buffer

int in = 0, out = 0; // Indices for producer and consumer

sem\_t empty; // Semaphore for empty slots

sem\_t full; // Semaphore for filled slots

pthread\_mutex\_t mutex; // Mutex for critical section

// Function for producer

void\* producer(void\* arg) {

int item;

for (int i = 0; i < 10; i++) {

item = rand() % 100; // Produce a random item

sem\_wait(&empty); // Wait for an empty slot

pthread\_mutex\_lock(&mutex); // Enter critical section

buffer[in] = item;

printf("Producer produced: %d\n", item);

in = (in + 1) % BUFFER\_SIZE;

pthread\_mutex\_unlock(&mutex); // Exit critical section

sem\_post(&full); // Signal that a slot is filled

sleep(1); // Simulate time to produce

}

return NULL;

}

// Function for consumer

void\* consumer(void\* arg) {

int item;

for (int i = 0; i < 10; i++) {

sem\_wait(&full); // Wait for a filled slot

pthread\_mutex\_lock(&mutex); // Enter critical section

item = buffer[out];

printf("Consumer consumed: %d\n", item);

out = (out + 1) % BUFFER\_SIZE;

pthread\_mutex\_unlock(&mutex); // Exit critical section

sem\_post(&empty); // Signal that a slot is empty

sleep(1); // Simulate time to consume

}

return NULL;

}

int main() {

pthread\_t prod, cons; // Threads for producer and consumer

// Initialize semaphores and mutex

sem\_init(&empty, 0, BUFFER\_SIZE);

sem\_init(&full, 0, 0);

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

// Create threads

pthread\_create(&prod, NULL, producer, NULL);

pthread\_create(&cons, NULL, consumer, NULL);

// Wait for threads to finish

pthread\_join(prod, NULL);

pthread\_join(cons, NULL);

// Destroy semaphores and mutex

sem\_destroy(&empty);

sem\_destroy(&full);

pthread\_mutex\_destroy(&mutex);

return 0;

}

**Simulate Bankers Algorithm for Dead Lock Avoidance**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 5

#define MAX\_RESOURCES 3

int main() {

int n, m; // Number of processes and resources

int alloc[MAX\_PROCESSES][MAX\_RESOURCES]; // Allocation matrix

int max[MAX\_PROCESSES][MAX\_RESOURCES]; // Maximum demand matrix

int avail[MAX\_RESOURCES]; // Available resources

int need[MAX\_PROCESSES][MAX\_RESOURCES]; // Need matrix

bool finish[MAX\_PROCESSES] = {false}; // Finish flag for each process

int safeSequence[MAX\_PROCESSES]; // Safe sequence

int work[MAX\_RESOURCES]; // Work array (temporary available resources)

// Input number of processes and resources

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

scanf("%d", &n);

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

scanf("%d", &m);

// Input Allocation Matrix

printf("\nEnter the Allocation Matrix:\n");

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

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

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

// Input Maximum Demand Matrix

printf("\nEnter the Maximum Matrix:\n");

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

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

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

// Input Available Resources

printf("\nEnter the Available Resources:\n");

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

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

// Calculate Need Matrix

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

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

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

// Print Need Matrix

printf("\nNeed Matrix:\n");

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

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

printf("%d ", need[i][j]);

printf("\n");

}

// Initialize work array

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

work[i] = avail[i];

// Safety Algorithm

int count = 0;

while (count < n) {

bool found = false;

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

if (!finish[i]) {

bool canAllocate = true;

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

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

canAllocate = false;

break;

}

}

if (canAllocate) {

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

work[j] += alloc[i][j];

safeSequence[count++] = i;

finish[i] = true;

found = true;

printf("Process %d executed.\n", i);

}

}

}

if (!found) {

printf("\nThe system is in an unsafe state.\n");

return 0;

}

}

// Print safe sequence

printf("\nThe system is in a safe state.\nSafe Sequence: ");

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

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

printf("\n");

return 0;

}

Output :   
Enter the number of processes: 5

Enter the number of resources: 3

Enter the Allocation Matrix:

0 1 0

2 0 0

3 0 2

2 1 1

0 0 2

Enter the Maximum Matrix:

7 5 3

3 2 2

9 0 2

4 2 2

5 3 3

Enter the Available Resources:

3 3 2

Ans :   
Need Matrix:

7 4 3

1 2 2

6 0 0

2 1 1

5 3 1

Process 1 executed.

Process 3 executed.

Process 4 executed.

Process 0 executed.

Process 2 executed.

The system is in a safe state.

Safe Sequence: P1 P3 P4 P0 P2

**Implement multithreading for Matrix Operations using Pthreads without pointer simple program**

#include <stdio.h>

#include <pthread.h>

#define SIZE 3 // Define the size of the matrix

// Declare matrices

int matrixA[SIZE][SIZE];

int matrixB[SIZE][SIZE];

int resultSum[SIZE][SIZE];

int resultProduct[SIZE][SIZE];

// Function for calculating sum of matrices

void\* calculateSum(void\* arg) {

int row = (int)arg; // Thread ID corresponds to the row being processed

for (int col = 0; col < SIZE; col++) {

resultSum[row][col] = matrixA[row][col] + matrixB[row][col];

}

return NULL;

}

// Function for calculating product of matrices

void\* calculateProduct(void\* arg) {

int row = (int)arg; // Thread ID corresponds to the row being processed

for (int col = 0; col < SIZE; col++) {

resultProduct[row][col] = 0;

for (int k = 0; k < SIZE; k++) {

resultProduct[row][col] += matrixA[row][k] \* matrixB[k][col];

}

}

return NULL;

}

int main() {

pthread\_t threads[SIZE];

// Input matrices

printf("Enter elements of Matrix A (%d x %d):\n", SIZE, SIZE);

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

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

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

}

}

printf("Enter elements of Matrix B (%d x %d):\n", SIZE, SIZE);

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

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

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

}

}

// Calculate sum of matrices using threads

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

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

}

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

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

}

// Calculate product of matrices using threads

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

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

}

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

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

}

// Display results

printf("\nSum of Matrices:\n");

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

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

printf("%d ", resultSum[i][j]);

}

printf("\n");

}

printf("\nProduct of Matrices:\n");

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

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

printf("%d ", resultProduct[i][j]);

}

printf("\n");

}

return 0;

}

Enter elements of Matrix A (3 x 3):

1 2 3

4 5 6

7 8 9

Enter elements of Matrix B (3 x 3):

9 8 7

6 5 4

3 2 1

Sum of Matrices:

10 10 10

10 10 10

10 10 10

Product of Matrices:

30 24 18

84 69 54

138 114 90

**Write a C program to simulate disk scheduling algorithms. a) FCFS b) SCAN**

#include <stdio.h>

#include <stdlib.h>

void fcfs(int requests[], int n, int head) {

int totalSeekTime = 0;

printf("\nFCFS Disk Scheduling:\n");

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

totalSeekTime += abs(requests[i] - head);

head = requests[i];

}

printf("Total Seek Time: %d\n", totalSeekTime);

}

void scan(int requests[], int n, int head, int diskSize, int direction) {

int totalSeekTime = 0, i;

int sortedRequests[n + 2];

int idx = 0;

// Add requests to a temporary array and include boundaries

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

sortedRequests[idx++] = requests[i];

}

if (direction == 1) { // Moving upward

sortedRequests[idx++] = diskSize - 1;

} else { // Moving downward

sortedRequests[idx++] = 0;

}

// Sort the array

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

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

if (sortedRequests[i] > sortedRequests[j]) {

int temp = sortedRequests[i];

sortedRequests[i] = sortedRequests[j];

sortedRequests[j] = temp;

}

}

}

// Process requests in SCAN order

printf("\nSCAN Disk Scheduling:\n");

if (direction == 1) { // Moving upward

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

if (sortedRequests[i] >= head) break;

}

for (int j = i; j < idx; j++) {

totalSeekTime += abs(sortedRequests[j] - head);

head = sortedRequests[j];

}

for (int j = i - 1; j >= 0; j--) {

totalSeekTime += abs(sortedRequests[j] - head);

head = sortedRequests[j];

}

} else { // Moving downward

for (i = idx - 1; i >= 0; i--) {

if (sortedRequests[i] <= head) break;

}

for (int j = i; j >= 0; j--) {

totalSeekTime += abs(sortedRequests[j] - head);

head = sortedRequests[j];

}

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

totalSeekTime += abs(sortedRequests[j] - head);

head = sortedRequests[j];

}

}

printf("Total Seek Time: %d\n", totalSeekTime);

}

int main() {

int n, head, diskSize, direction;

// Input the number of requests

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

scanf("%d", &n);

// Input the requests

int requests[n];

printf("Enter the disk requests: ");

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

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

}

// Input initial head position

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

scanf("%d", &head);

// Input disk size

printf("Enter the disk size: ");

scanf("%d", &diskSize);

// Input direction for SCAN

printf("Enter the direction for SCAN (0 for down, 1 for up): ");

scanf("%d", &direction);

fcfs(requests, n, head);

scan(requests, n, head, diskSize, direction);

return 0;

}

---------------------------------------------------------------------------------------

Enter the number of disk requests: 5

Enter the disk requests: 98 183 37 122 14

Enter the initial head position: 53

Enter the disk size: 200

Enter the direction for SCAN (0 for down, 1 for up): 1  
-------------------------------------------------------------------------------------------  
  
FCFS Disk Scheduling:

Total Seek Time: 640

SCAN Disk Scheduling:

Total Seek Time: 382

**Write a C program to simulate the following contiguous memory allocation Techniques a) Worst fit b) Best fit c) First fit.**

#include <stdio.h>

void firstFit(int blocks[], int m, int processes[], int n) {

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

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

if (blocks[j] >= processes[i]) {

printf("Process %d (Size %d) -> Block %d\n", i + 1, processes[i], j + 1);

blocks[j] -= processes[i];

break;

}

}

}

}

void bestFit(int blocks[], int m, int processes[], int n) {

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

int bestIdx = -1;

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

if (blocks[j] >= processes[i] && (bestIdx == -1 || blocks[j] < blocks[bestIdx])) {

bestIdx = j;

}

}

if (bestIdx != -1) {

printf("Process %d (Size %d) -> Block %d\n", i + 1, processes[i], bestIdx + 1);

blocks[bestIdx] -= processes[i];

}

}

}

void worstFit(int blocks[], int m, int processes[], int n) {

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

int worstIdx = -1;

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

if (blocks[j] >= processes[i] && (worstIdx == -1 || blocks[j] > blocks[worstIdx])) {

worstIdx = j;

}

}

if (worstIdx != -1) {

printf("Process %d (Size %d) -> Block %d\n", i + 1, processes[i], worstIdx + 1);

blocks[worstIdx] -= processes[i];

}

}

}

int main() {

int blocks[] = {100, 500, 200, 300, 600};

int processes[] = {212, 417, 112, 426};

int m = 5, n = 4;

int tempBlocks[m];

printf("First Fit Allocation:\n");

for (int i = 0; i < m; i++) tempBlocks[i] = blocks[i];

firstFit(tempBlocks, m, processes, n);

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

for (int i = 0; i < m; i++) tempBlocks[i] = blocks[i];

bestFit(tempBlocks, m, processes, n);

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

for (int i = 0; i < m; i++) tempBlocks[i] = blocks[i];

worstFit(tempBlocks, m, processes, n);

return 0;

}

Output:

Enter the number of memory blocks: 5

Enter the sizes of the memory blocks:

100 500 200 300 600

Enter the number of processes: 4

Enter the sizes of the processes:

212 417 112 426

First Fit Allocation:

Process Size Block

P1 212 B2

P2 417 B5

P3 112 B3

P4 426 Not Allocated

Best Fit Allocation:

Process Size Block

P1 212 B3

P2 417 B5

P3 112 B1

P4 426 Not Allocated

Worst Fit Allocation:

Process Size Block

P1 212 B5

P2 417 B2

P3 112 B5

P4 426 Not Allocated

Simulate the following CPU scheduling algorithms. a) FCFS b) SJF c) Round Robin

#include <stdio.h>

void fcfs(int n, int bt[]) {

int wt[n], tat[n];

wt[0] = 0;

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

wt[i] = wt[i - 1] + bt[i - 1]; // Waiting time for process i

}

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

tat[i] = wt[i] + bt[i]; // Turnaround time for process i

}

printf("\nFCFS Scheduling:\n");

printf("Process\tBT\tWT\tTAT\n");

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

printf("P%d\t%d\t%d\t%d\n", i + 1, bt[i], wt[i], tat[i]);

}

}

void sjf(int n, int bt[]) {

int wt[n], tat[n], completed[n], temp[n];

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

completed[i] = 0;

temp[i] = bt[i]; // Copy burst times for sorting

}

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

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

if (temp[i] > temp[j]) { // Sort burst times

int t = temp[i];

temp[i] = temp[j];

temp[j] = t;

}

}

}

wt[0] = 0;

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

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

}

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

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

}

printf("\nSJF Scheduling:\n");

printf("Process\tBT\tWT\tTAT\n");

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

printf("P%d\t%d\t%d\t%d\n", i + 1, temp[i], wt[i], tat[i]);

}

}

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

int rem\_bt[n], wt[n], tat[n], t = 0;

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

rem\_bt[i] = bt[i]; // Remaining burst times

}

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;

}

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

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

}

printf("\nRound Robin Scheduling:\n");

printf("Process\tBT\tWT\tTAT\n");

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

printf("P%d\t%d\t%d\t%d\n", i + 1, bt[i], wt[i], tat[i]);

}

}

int main() {

int n, quantum;

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

scanf("%d", &n);

int bt[n];

printf("Enter the burst times of the processes:\n");

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

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

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

}

printf("Enter the time quantum for Round Robin: ");

scanf("%d", &quantum);

fcfs(n, bt); // First-Come, First-Served

sjf(n, bt); // Shortest Job First

roundRobin(n, bt, quantum); // Round Robin

return 0;

}  
  
  
Enter the number of processes: 3

Enter the burst times of the processes:

P1: 10

P2: 5

P3: 8

Enter the time quantum for Round Robin: 2  
  
FCFS Scheduling:

Process BT WT TAT

P1 10 0 10

P2 5 10 15

P3 8 15 23

SJF Scheduling:

Process BT WT TAT

P1 5 0 5

P2 8 5 13

P3 10 13 23

Round Robin Scheduling:

Process BT WT TAT

P1 10 13 23

P2 5 8 13

P3 8 12 20

**Process Related system Calls 1. To write C Programs using the following system calls of UNIX operating system fork, getpid, getppid, exit, wait. 2. To write C Programs using the execve system call**

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

#include <sys/wait.h>

#include <stdlib.h>

int main() {

pid\_t pid;

// Create a child process

pid = fork();

if (pid < 0) {

printf("Fork failed!\n");

return 1; // Exit if fork fails

}

if (pid == 0) {

// Child process

printf("Child Process:\n");

printf("PID: %d\n", getpid()); // Get the process ID of child

printf("Parent PID: %d\n", getppid()); // Get the parent process ID

exit(0); // Exit child process

} else {

// Parent process

printf("Parent Process:\n");

printf("PID: %d\n", getpid()); // Get the process ID of parent

printf("Waiting for child to finish...\n");

wait(NULL); // Wait for child process to finish

printf("Child process finished.\n");

}

return 0;

}  
  
  
  
-------------------------------------------------------------------------------

#include <stdio.h>

#include <unistd.h>

int main() {

char \*args[] = {"/bin/ls", "-l", NULL}; // Program to execute (ls -l)

printf("Before execve()\n");

// Execute the program using execve

if (execve(args[0], args, NULL) == -1) {

perror("Error executing execve");

}

// This line will not execute unless execve fails

printf("This line will not print if execve is successful.\n");

return 0;

}

**Assignment No 1 : Design a basic calculator**

#!/bin/bash

echo "Basic Calculator"

echo "Enter first number:"

read num1

echo "Enter second number:"

read num2

echo "Select operation:"

echo "1. Addition"

echo "2. Subtraction"

echo "3. Multiplication"

echo "4. Division"

read choice

case $choice in

1) result=$((num1 + num2))

echo "Result: $num1 + $num2 = $result";;

2) result=$((num1 - num2))

echo "Result: $num1 - $num2 = $result";;

3) result=$((num1 \* num2))

echo "Result: $num1 \* $num2 = $result";;

4) if [ $num2 -ne 0 ]; then

result=$((num1 / num2))

echo "Result: $num1 / $num2 = $result"

else

echo "Error: Division by zero is not allowed."

fi;;

\*) echo "Invalid choice";;

esac

**Assignment No 2 : Use of different loops**

#!/bin/bash

echo "Different Loop Examples"

# 1. FOR Loop

echo "For Loop: Numbers from 1 to 5"

for i in {1..5}

do

echo "$i"

done

# 2. WHILE Loop

echo "While Loop: Numbers from 1 to 5"

count=1

while [ $count -le 5 ]

do

echo "$count"

count=$((count + 1))

done

# 3. UNTIL Loop

echo "Until Loop: Numbers from 1 to 5"

count=1

until [ $count -gt 5 ]

do

echo "$count"

count=$((count + 1))

done