1. **Draw the Gantt charts and compute the finish time, turnaround time and waiting time for the following algorithms:  a. First come First serve b. Shortest Job First**

#include <stdio.h>

// Structure to hold process information

typedef struct {

int processID;

int arrivalTime;

int burstTime;

int completionTime;

int turnaroundTime;

int waitingTime;

} Process;

// Function to sort processes by arrival time (for FCFS)

void sortByArrivalTime(Process proc[], int n) {

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

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

if (proc[j].arrivalTime > proc[j + 1].arrivalTime) {

Process temp = proc[j];

proc[j] = proc[j + 1];

proc[j + 1] = temp;

}

}

}

}

// Function to sort processes by burst time (for SJF)

void sortByBurstTime(Process proc[], int n) {

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

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

if (proc[j].burstTime > proc[j + 1].burstTime) {

Process temp = proc[j];

proc[j] = proc[j + 1];

proc[j + 1] = temp;

}

}

}

}

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

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

int currentTime = 0;

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

if (currentTime < proc[i].arrivalTime) {

currentTime = proc[i].arrivalTime;

}

proc[i].completionTime = currentTime + proc[i].burstTime;

currentTime = proc[i].completionTime;

proc[i].turnaroundTime = proc[i].completionTime - proc[i].arrivalTime;

proc[i].waitingTime = proc[i].turnaroundTime - proc[i].burstTime;

}

}

// Function to print Gantt chart and process details

void printResults(Process proc[], int n) {

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

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

printf("| P%d ", proc[i].processID);

}

printf("|\n");

int currentTime = 0;

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

if (currentTime < proc[i].arrivalTime) {

currentTime = proc[i].arrivalTime;

}

printf("%d ", currentTime);

currentTime += proc[i].burstTime;

}

printf("%d\n", currentTime);

// Print table of process details

printf("\nProcess Details:\n");

printf("ID\tArrival\tBurst\tCompletion\tTurnaround\tWaiting\n");

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

printf("P%d\t%d\t%d\t%d\t\t%d\t\t%d\n", proc[i].processID, proc[i].arrivalTime, proc[i].burstTime, proc[i].completionTime, proc[i].turnaroundTime, proc[i].waitingTime);

}

}

int main() {

int n;

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

scanf("%d", &n);

Process proc[n];

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

proc[i].processID = i + 1;

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

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

}

// FCFS

printf("\nFirst Come First Serve (FCFS):\n");

sortByArrivalTime(proc, n);

calculateTimes(proc, n);

printResults(proc, n);

// SJF (non-preemptive)

printf("\nShortest Job First (SJF):\n");

sortByBurstTime(proc, n);

calculateTimes(proc, n);

printResults(proc, n);

return 0;

}

**Draw the Gantt charts and compute the finish time, turnaround time and waiting time for the following algorithms:  a. First come First serve b. Round- Robin**

#include <stdio.h>

// Structure to store process details

struct Process {

int pid; // Process ID

int arrivalTime; // Arrival time

int burstTime; // Burst time

int waitingTime; // Waiting time

int turnaroundTime; // Turnaround time

int completionTime; // Completion time

};

// Function for First Come First Serve Scheduling

void FCFS(struct Process processes[], int n) {

int currentTime = 0;

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

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

if (currentTime < processes[i].arrivalTime) {

currentTime = processes[i].arrivalTime;

}

processes[i].completionTime = currentTime + processes[i].burstTime;

currentTime += processes[i].burstTime;

processes[i].turnaroundTime = processes[i].completionTime - processes[i].arrivalTime;

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

// Print Gantt chart

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

}

printf("|\n");

// Print times

printf("\nProcess Arrival Burst Completion Turnaround Waiting\n");

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

printf("P%d\t%d\t%d\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].arrivalTime, processes[i].burstTime,

processes[i].completionTime, processes[i].turnaroundTime, processes[i].waitingTime);

}

}

// Function for Round Robin Scheduling

void RoundRobin(struct Process processes[], int n, int timeQuantum) {

int remainingBurstTime[n];

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

remainingBurstTime[i] = processes[i].burstTime;

}

int currentTime = 0;

int completed = 0;

int totalTurnaroundTime = 0, totalWaitingTime = 0;

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

while (completed < n) {

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

if (remainingBurstTime[i] > 0 && processes[i].arrivalTime <= currentTime) {

if (remainingBurstTime[i] > timeQuantum) {

printf("| P%d (%d) ", processes[i].pid, currentTime + timeQuantum);

currentTime += timeQuantum;

remainingBurstTime[i] -= timeQuantum;

} else {

printf("| P%d (%d) ", processes[i].pid, currentTime + remainingBurstTime[i]);

currentTime += remainingBurstTime[i];

processes[i].completionTime = currentTime;

processes[i].turnaroundTime = currentTime - processes[i].arrivalTime;

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

totalTurnaroundTime += processes[i].turnaroundTime;

totalWaitingTime += processes[i].waitingTime;

remainingBurstTime[i] = 0;

completed++;

}

}

}

}

printf("|\n");

// Print times

printf("\nProcess Arrival Burst Completion Turnaround Waiting\n");

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

printf("P%d\t%d\t%d\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].arrivalTime, processes[i].burstTime,

processes[i].completionTime, processes[i].turnaroundTime, processes[i].waitingTime);

}

}

int main() {

int n, timeQuantum;

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

scanf("%d", &n);

struct Process processes[n];

// Input process details

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

processes[i].pid = i + 1;

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

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

}

// Sort processes by arrival time for FCFS

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

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

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

struct Process temp = processes[i];

processes[i] = processes[j];

processes[j] = temp;

}

}

}

// Run FCFS scheduling

FCFS(processes, n);

// Input time quantum for Round Robin

printf("\nEnter time quantum for Round Robin: ");

scanf("%d", &timeQuantum);

// Run Round Robin scheduling

RoundRobin(processes, n, timeQuantum);

return 0;

}

1. **Draw the Gantt charts and compute the finish time, turnaround time and waiting time for the following algorithms: a. Priority scheduling b. Shortest Job First (Non Pre-emptive)**

#include <stdio.h>

// Structure to hold process details

struct Process {

int pid; // Process ID

int arrival\_time; // Arrival time

int burst\_time; // Burst time

int priority; // Priority (for Priority Scheduling)

int completion\_time;

int turnaround\_time;

int waiting\_time;

};

// Function to sort processes by arrival time

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

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

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

if (proc[j].arrival\_time > proc[j + 1].arrival\_time) {

struct Process temp = proc[j];

proc[j] = proc[j + 1];

proc[j + 1] = temp;

}

}

}

}

// Function to implement Priority Scheduling (Non-Preemptive)

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

sortByArrival(proc, n);

int completed = 0, time = 0;

while (completed < n) {

int idx = -1;

int highest\_priority = 1e9; // Assume lower number means higher priority

// Select the process with the highest priority that has arrived

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

if (proc[i].arrival\_time <= time && proc[i].completion\_time == 0 && proc[i].priority < highest\_priority) {

highest\_priority = proc[i].priority;

idx = i;

}

}

if (idx != -1) {

time += proc[idx].burst\_time;

proc[idx].completion\_time = time;

proc[idx].turnaround\_time = proc[idx].completion\_time - proc[idx].arrival\_time;

proc[idx].waiting\_time = proc[idx].turnaround\_time - proc[idx].burst\_time;

completed++;

} else {

time++;

}

}

// Print Gantt Chart and times

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

printf("PID\tAT\tBT\tPriority\tCT\tTAT\tWT\n");

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

printf("P%d\t%d\t%d\t%d\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);

}

}

// Function to implement Shortest Job First (Non-Preemptive)

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

sortByArrival(proc, n);

int completed = 0, time = 0;

while (completed < n) {

int idx = -1;

int min\_burst = 1e9;

// Select the process with the shortest burst time that has arrived

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

if (proc[i].arrival\_time <= time && proc[i].completion\_time == 0 && proc[i].burst\_time < min\_burst) {

min\_burst = proc[i].burst\_time;

idx = i;

}

}

if (idx != -1) {

time += proc[idx].burst\_time;

proc[idx].completion\_time = time;

proc[idx].turnaround\_time = proc[idx].completion\_time - proc[idx].arrival\_time;

proc[idx].waiting\_time = proc[idx].turnaround\_time - proc[idx].burst\_time;

completed++;

} else {

time++;

}

}

// Print Gantt Chart and times

printf("\nShortest Job First (Non-Preemptive):\n");

printf("PID\tAT\tBT\tCT\tTAT\tWT\n");

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

printf("P%d\t%d\t%d\t%d\t%d\t%d\n", proc[i].pid, proc[i].arrival\_time, proc[i].burst\_time,

proc[i].completion\_time, proc[i].turnaround\_time, proc[i].waiting\_time);

}

}

int main() {

int n;

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

scanf("%d", &n);

struct Process proc[n];

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

proc[i].pid = i + 1;

printf("Enter arrival time, burst time, and priority (for Priority Scheduling) for process P%d: ", i + 1);

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

proc[i].completion\_time = 0; // Initialize completion time

}

// Perform Priority Scheduling

priorityScheduling(proc, n);

// Perform Shortest Job First Scheduling

shortestJobFirst(proc, n);

return 0;

}

**Deadlock -Bankers Algorithm**

#include <stdio.h>

#include <stdbool.h>

// Number of processes and resources

#define P 5

#define R 3

// Function to check if the system is in a safe state

bool isSafe(int processes[], int available[], int max[P][R], int allocation[P][R]) {

int work[R];

bool finish[P] = {false};

int safeSequence[P];

int count = 0;

// Initialize work array as available

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

work[i] = available[i];

}

while (count < P) {

bool found = false;

for (int p = 0; p < P; p++) {

if (!finish[p]) {

int j;

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

if (max[p][j] - allocation[p][j] > work[j]) {

break;

}

}

if (j == R) {

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

work[k] += allocation[p][k];

}

safeSequence[count++] = p;

finish[p] = true;

found = true;

}

}

}

if (!found) {

printf("The system is not in a safe state.\n");

return false;

}

}

printf("The system is in a safe state. Safe sequence is:\n");

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

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

}

printf("\n");

return true;

}

int main() {

int processes[P] = {0, 1, 2, 3, 4};

// Available resources

int available[R] = {3, 3, 2};

// Maximum resources for each process

int max[P][R] = {

{7, 5, 3},

{3, 2, 2},

{9, 0, 2},

{2, 2, 2},

{4, 3, 3}

};

// Allocated resources for each process

int allocation[P][R] = {

{0, 1, 0},

{2, 0, 0},

{3, 0, 2},

{2, 1, 1},

{0, 0, 2}

};

// Check system safety

isSafe(processes, available, max, allocation);

return 0;

}

1. **Write a program to implement Banker’s algorithm.**

#include <stdio.h>

#include <stdbool.h>

// Number of processes and resources

#define P 5

#define R 3

// Function to check if the resources can be allocated

bool isSafe(int processes[], int avail[], int max[][R], int allot[][R]) {

int work[R];

bool finish[P] = {false};

// Copy the available resources to work[]

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

work[i] = avail[i];

}

int safeSeq[P];

int count = 0;

while (count < P) {

bool found = false;

for (int p = 0; p < P; p++) {

if (!finish[p]) {

bool canAllocate = true;

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

if (max[p][j] - allot[p][j] > work[j]) {

canAllocate = false;

break;

}

}

if (canAllocate) {

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

work[k] += allot[p][k];

}

safeSeq[count++] = p;

finish[p] = true;

found = true;

}

}

}

if (!found) {

printf("The system is not in a safe state.\n");

return false;

}

}

printf("The system is in a safe state.\nSafe sequence is: ");

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

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

}

printf("\n");

return true;

}

int main() {

int processes[P] = {0, 1, 2, 3, 4};

// Available resources

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

// Maximum resources that each process may need

int max[P][R] = {

{7, 5, 3},

{3, 2, 2},

{9, 0, 2},

{2, 2, 2},

{4, 3, 3}

};

// Resources currently allocated to each process

int allot[P][R] = {

{0, 1, 0},

{2, 0, 0},

{3, 0, 2},

{2, 1, 1},

{0, 0, 2}

};

isSafe(processes, avail, max, allot);

return 0;

}

1. **Write a program to implement FIFO Page Replacement Algorithm.**

#include <stdio.h>

#include <stdlib.h>

void fifo(int ref[], int size, int n) {

int frames[n];

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

frames[i] = -1;

}

int pageMiss = 0;

int pageHits = 0;

int currentIndex = 0;

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

int flag = 0;

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

if (frames[j] == ref[i]) {

flag = 1;

break;

}

}

if (flag == 1) {

pageHits++;

} else {

frames[currentIndex] = ref[i];

currentIndex = (currentIndex + 1) % n; // current index value to next index in FIFO % no of frames

pageMiss++;

}

}

printf("FIFO Page Faults: %d, Page Hits: %d\n", pageMiss, pageHits);

}

int main(void) {

int frames;

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

scanf("%d", &frames);

int size;

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

scanf("%d", &size);

int ref[size];

printf("Enter the reference string: ");

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

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

}

fifo(ref, size, frames);

return 0;

}

**Write a program to implement LRU Page Replacement Algorithm.**

#include <stdio.h>

#include <stdlib.h>

void lru(int ref[], int size, int n) {

int frames[n]; // Array to store frames

int time[n]; // Array to store the time of last usage of each frame

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

frames[i] = -1; // Initialize frames as empty

time[i] = 0; // Initialize time to 0

}

int pageMiss = 0; // To count the page misses

int pageHits = 0; // To count the page hits

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

int flag = 0; // Flag to check if the page is found in frames

// Check if the current page is already in one of the frames

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

if (frames[j] == ref[i]) {

flag = 1; // Page hit

time[j] = i; // Update the time of last usage

break;

}

}

if (flag == 1) {

pageHits++; // Increment page hits if the page was found

} else {

int oldest = 0;

// Find the least recently used page (oldest)

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

if (time[j] < time[oldest]) {

oldest = j; // Update to the frame with the least recent usage

}

}

frames[oldest] = ref[i]; // Replace the oldest frame with the new page

time[oldest] = i; // Update the time of last usage

pageMiss++; // Increment page misses

}

}

// Print the result

printf("LRU Page Faults: %d, Page Hits: %d\n", pageMiss, pageHits);

}

int main(void) {

int frames; // Number of frames

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

scanf("%d", &frames);

int size; // Size of the reference string

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

scanf("%d", &size);

int ref[size]; // Reference string

printf("Enter the reference string: ");

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

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

}

lru(ref, size, frames); // Call the LRU function to perform the algorithm

return 0;

}

**PREDEFINED FIFO**

#include <stdio.h>

#include <stdlib.h>

void fifo(int ref[], int size, int n) {

int frames[n];

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

frames[i] = -1;

}

int pageMiss = 0;

int pageHits = 0;

int currentIndex = 0;

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

int flag = 0;

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

if (frames[j] == ref[i]) {

flag = 1;

break;

}

}

if (flag == 1) {

pageHits++;

} else {

frames[currentIndex] = ref[i];

currentIndex = (currentIndex + 1) % n; // current index value to next index in FIFO % no of frames

pageMiss++;

}

}

printf("FIFO Page Faults: %d, Page Hits: %d\n", pageMiss, pageHits);

}

int main(void) {

int frames = 3; // Predefined number of frames

int size = 12; // Predefined size of the reference string

int ref[] = {1, 3, 0, 3, 5, 6, 3, 2, 4, 3, 6, 2}; // Predefined reference string

printf("Number of frames: %d\n", frames);

printf("Reference string: ");

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

printf("%d ", ref[i]);

}

printf("\n");

fifo(ref, size, frames);

return 0;

}

**PREDEFINED LRU**

#include <stdio.h>

#include <stdlib.h>

void lru(int ref[], int size, int n) {

int frames[n]; // Array to store frames

int time[n]; // Array to store the time of last usage of each frame

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

frames[i] = -1; // Initialize frames as empty

time[i] = 0; // Initialize time to 0

}

int pageMiss = 0; // To count the page misses

int pageHits = 0; // To count the page hits

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

int flag = 0; // Flag to check if the page is found in frames

// Check if the current page is already in one of the frames

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

if (frames[j] == ref[i]) {

flag = 1; // Page hit

time[j] = i; // Update the time of last usage

break;

}

}

if (flag == 1) {

pageHits++; // Increment page hits if the page was found

} else {

int oldest = 0;

// Find the least recently used page (oldest)

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

if (time[j] < time[oldest]) {

oldest = j; // Update to the frame with the least recent usage

}

}

frames[oldest] = ref[i]; // Replace the oldest frame with the new page

time[oldest] = i; // Update the time of last usage

pageMiss++; // Increment page misses

}

}

// Print the result

printf("LRU Page Faults: %d, Page Hits: %d\n", pageMiss, pageHits);

}

int main(void) {

int frames = 3; // Predefined number of frames

int ref[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2}; // Predefined reference string

int size = sizeof(ref) / sizeof(ref[0]); // Calculate the size of the reference string

lru(ref, size, frames); // Call the LRU function to perform the algorithm

return 0;

}