Exp 1

#include <stdio.h>

#include <stdbool.h>

#define NUM\_PROCESSES 4

#define NUM\_RESOURCES 3

int claim[NUM\_PROCESSES][NUM\_RESOURCES] = {

{3, 2, 2},

{6, 1, 3},

{3, 1, 4},

{4, 2, 2}

};

int allocation[NUM\_PROCESSES][NUM\_RESOURCES] = {

{1, 0, 0},

{6, 1, 2},

{2, 1, 1},

{0, 0, 2}

};

int available[NUM\_RESOURCES] = {9, 3, 6};

bool finish[NUM\_PROCESSES] = {false};

int main() {

int work[NUM\_RESOURCES];

int need[NUM\_PROCESSES][NUM\_RESOURCES];

int i, j, k;

// Initialize the need matrix

for (i = 0; i < NUM\_PROCESSES; i++) {

for (j = 0; j < NUM\_RESOURCES; j++) {

need[i][j] = claim[i][j] - allocation[i][j];

}

}

// Initialize the work matrix

for (i = 0; i < NUM\_RESOURCES; i++) {

work[i] = available[i];

}

// Check if there is a safe sequence

int safe\_sequence[NUM\_PROCESSES];

int count = 0;

bool found;

while (count < NUM\_PROCESSES) {

found = false;

for (i = 0; i < NUM\_PROCESSES; i++) {

if (!finish[i]) {

for (j = 0; j < NUM\_RESOURCES; j++) {

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

break;

}

}

if (j == NUM\_RESOURCES) {

// Process i can finish

for (k = 0; k < NUM\_RESOURCES; k++) {

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

}

finish[i] = true;

safe\_sequence[count] = i;

count++;

found = true;

}

}

}

if (!found) {

break;

}

}

if (count == NUM\_PROCESSES) {

printf("Safe sequence: ");

for (i = 0; i < NUM\_PROCESSES; i++) {

printf("%d ", safe\_sequence[i]);

}

printf("\n");

return 0;

} else {

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

return 1;

}

}

Exp 2

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PAGE\_FRAMES 10

int main() {

int page\_frames[MAX\_PAGE\_FRAMES];

int page\_reference\_seq[] = {4, 1, 2, 4, 3, 2, 1, 5};

int num\_page\_frames = 3;

int num\_page\_reference = sizeof(page\_reference\_seq) / sizeof(page\_reference\_seq[0]);

int i, j, k, fault\_count = 0;

bool found;

// Initialize the page frames to -1

for (i = 0; i < num\_page\_frames; i++) {

page\_frames[i] = -1;

}

// Traverse the page reference sequence

for (i = 0; i < num\_page\_reference; i++) {

found = false;

// Check if the page is already in a frame

for (j = 0; j < num\_page\_frames; j++) {

if (page\_frames[j] == page\_reference\_seq[i]) {

found = true;

break;

}

}

// If the page is not in a frame, replace the oldest page in the queue

if (!found) {

page\_frames[fault\_count % num\_page\_frames] = page\_reference\_seq[i];

fault\_count++;

}

// Print the current state of the page frames

printf("Page frames: ");

for (k = 0; k < num\_page\_frames; k++) {

if (page\_frames[k] == -1) {

printf("- ");

} else {

printf("%d ", page\_frames[k]);

}

}

printf("\n");

}

printf("Number of page faults: %d\n", fault\_count);

return 0;

}

Exp 3

#include <stdio.h>

#define MAX\_PROCESSES 10

int main() {

int burst\_times[MAX\_PROCESSES] = {6, 8, 7, 3};

int num\_processes = 4;

int completion\_times[MAX\_PROCESSES];

int waiting\_times[MAX\_PROCESSES];

int turnaround\_times[MAX\_PROCESSES];

int i, j, shortest\_job\_index, total\_waiting\_time = 0, total\_turnaround\_time = 0;

// Find the completion times for each process

for (i = 0; i < num\_processes; i++) {

shortest\_job\_index = i;

for (j = i + 1; j < num\_processes; j++) {

if (burst\_times[j] < burst\_times[shortest\_job\_index]) {

shortest\_job\_index = j;

}

}

// Swap the current process with the one with the shortest burst time

int temp = burst\_times[i];

burst\_times[i] = burst\_times[shortest\_job\_index];

burst\_times[shortest\_job\_index] = temp;

// Calculate the completion time for the current process

if (i == 0) {

completion\_times[i] = burst\_times[i];

} else {

completion\_times[i] = completion\_times[i-1] + burst\_times[i];

}

}

// Calculate the waiting and turnaround times for each process

for (i = 0; i < num\_processes; i++) {

waiting\_times[i] = completion\_times[i] - burst\_times[i];

turnaround\_times[i] = completion\_times[i];

total\_waiting\_time += waiting\_times[i];

total\_turnaround\_time += turnaround\_times[i];

}

// Print the results

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

for (i = 0; i < num\_processes; i++) {

printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\n", i+1, burst\_times[i], completion\_times[i], waiting\_times[i], turnaround\_times[i]);

}

printf("Average Waiting Time: %.2f\n", (float)total\_waiting\_time / num\_processes);

printf("Average Turnaround Time: %.2f\n", (float)total\_turnaround\_time / num\_processes);

return 0;

}

Exp 4

#include <stdio.h>

#define MAX\_PARTITIONS 10

int main() {

int partitions[MAX\_PARTITIONS] = {300, 600, 350, 200, 750, 125};

int num\_partitions = 6;

int processes[] = {115, 500, 358, 200, 375};

int num\_processes = 5;

int i, j, allocated\_partition\_index;

// Initialize all partitions to be unallocated

int allocation[num\_partitions];

for (i = 0; i < num\_partitions; i++) {

allocation[i] = -1;

}

// Allocate memory to each process using the First-Fit algorithm

for (i = 0; i < num\_processes; i++) {

allocated\_partition\_index = -1;

for (j = 0; j < num\_partitions; j++) {

if (allocation[j] == -1 && processes[i] <= partitions[j]) {

allocated\_partition\_index = j;

break;

}

}

if (allocated\_partition\_index != -1) {

allocation[allocated\_partition\_index] = i;

printf("Process %d of size %d KB allocated to partition %d of size %d KB\n", i+1, processes[i], allocated\_partition\_index+1, partitions[allocated\_partition\_index]);

} else {

printf("Process %d of size %d KB cannot be allocated\n", i+1, processes[i]);

}

}

return 0;

}

Exp 5

#include <stdio.h>

#define MAX\_PROCESSES 10

int main() {

int arrival\_times[MAX\_PROCESSES] = {0, 1, 2, 4};

int burst\_times[MAX\_PROCESSES] = {5, 3, 3, 1};

int num\_processes = 4;

int remaining\_burst\_times[MAX\_PROCESSES];

int completion\_times[MAX\_PROCESSES];

int waiting\_times[MAX\_PROCESSES];

int turnaround\_times[MAX\_PROCESSES];

int current\_time, shortest\_remaining\_time, shortest\_index, i, j;

// Initialize the remaining burst times to the initial burst times

for (i = 0; i < num\_processes; i++) {

remaining\_burst\_times[i] = burst\_times[i];

}

current\_time = 0;

while (1) {

shortest\_remaining\_time = -1;

shortest\_index = -1;

// Find the process with the shortest remaining burst time

for (i = 0; i < num\_processes; i++) {

if (arrival\_times[i] <= current\_time && remaining\_burst\_times[i] > 0) {

if (shortest\_remaining\_time == -1 || remaining\_burst\_times[i] < shortest\_remaining\_time) {

shortest\_remaining\_time = remaining\_burst\_times[i];

shortest\_index = i;

}

}

}

if (shortest\_index == -1) {

// All processes have completed

break;

}

// Execute the process with the shortest remaining burst time for 1 millisecond

remaining\_burst\_times[shortest\_index]--;

current\_time++;

// If the process has completed, record its completion time, waiting time, and turnaround time

if (remaining\_burst\_times[shortest\_index] == 0) {

completion\_times[shortest\_index] = current\_time;

waiting\_times[shortest\_index] = completion\_times[shortest\_index] - arrival\_times[shortest\_index] - burst\_times[shortest\_index];

turnaround\_times[shortest\_index] = completion\_times[shortest\_index] - arrival\_times[shortest\_index];

}

}

// Calculate the total waiting time and turnaround time

float total\_waiting\_time = 0, total\_turnaround\_time = 0;

for (i = 0; i < num\_processes; i++) {

total\_waiting\_time += waiting\_times[i];

total\_turnaround\_time += turnaround\_times[i];

}

// Calculate the average waiting time and turnaround time

float avg\_waiting\_time = total\_waiting\_time / num\_processes;

float avg\_turnaround\_time = total\_turnaround\_time / num\_processes;

// Print the results

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

for (i = 0; i < num\_processes; i++) {

printf("P%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", i+1, arrival\_times[i], burst\_times[i], completion\_times[i], waiting\_times[i], turnaround\_times[i]);

}

printf("Average waiting time: %.2f milliseconds\n", avg\_waiting\_time);

printf("Average turnaround time: %.2f milliseconds\n", avg\_turnaround\_time);

return 0;

}