**Slip 6 .**

**Q.1 Write a program to find the execution time taken for execution of a given set of instructions**

**(use clock() function)**

**ANS**

#include <stdio.h>

#include <time.h>

void perform\_task() {

// Simulating a task by running a loop

long sum = 0;

for (long i = 0; i < 100000000; i++) {

sum += i;

}

printf("Sum: %ld\n", sum);

}

int main() {

// Start the clock

clock\_t start\_time = clock();

// Perform the task

perform\_task();

// Stop the clock

clock\_t end\_time = clock();

// Calculate the time taken in seconds

double time\_taken = (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC;

// Print the time taken

printf("Time taken: %f seconds\n", time\_taken);

return 0;

}

**Q.2 Write the simulation program to implement demand paging and show the page scheduling**

**and total number of page faults for the following given page reference string. Give input n as the**

**number of memory frames.**

**Reference String :3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6**

**Implement FIFO**

**ANS:**

#include <stdio.h>

#define MAX\_FRAMES 10

#define MAX\_REFERENCES 15

void print\_frames(int frames[], int frame\_count) {

for (int i = 0; i < frame\_count; i++) {

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

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

} else {

printf("- "); // For empty frames

}

}

printf("\n");

}

int main() {

int frames[MAX\_FRAMES];

int frame\_count, page\_faults = 0;

int reference\_string[MAX\_REFERENCES] = {3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6};

int n = sizeof(reference\_string) / sizeof(reference\_string[0]);

// Initialize frames

for (int i = 0; i < MAX\_FRAMES; i++) {

frames[i] = -1; // -1 indicates an empty frame

}

printf("Enter the number of memory frames (max %d): ", MAX\_FRAMES);

scanf("%d", &frame\_count);

if (frame\_count > MAX\_FRAMES || frame\_count <= 0) {

printf("Invalid number of frames. Please enter a number between 1 and %d.\n", MAX\_FRAMES);

return 1;

}

int next\_frame = 0; // Next frame to be replaced

printf("Reference String: ");

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

printf("%d ", reference\_string[i]);

}

printf("\n");

printf("Frame State after each reference:\n");

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

int page\_found = 0;

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

for (int j = 0; j < frame\_count; j++) {

if (frames[j] == reference\_string[i]) {

page\_found = 1; // Page is found in memory

break;

}

}

// If page not found, a page fault occurs

if (!page\_found) {

frames[next\_frame] = reference\_string[i]; // Replace the oldest page

next\_frame = (next\_frame + 1) % frame\_count; // Move to the next frame

page\_faults++;

}

print\_frames(frames, frame\_count); // Print current state of frames

}

printf("Total number of page faults: %d\n", page\_faults);

return 0;

}

**Slip 7 .**

**Q.1 Write a program to create a child process using fork().The parent should goto sleep state and**

**child process should begin its execution. In the child process, use execl() to execute the “ls”**

**command.**

**ANS:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

pid\_t pid = fork(); // Create a new process

if (pid < 0) {

// Fork failed

perror("Fork failed");

return 1;

} else if (pid == 0) {

// Child process

printf("Child process (PID: %d) executing 'ls' command...\n", getpid());

// Execute the 'ls' command

execl("/bin/ls", "ls", NULL);

// If execl returns, it must have failed

perror("execl failed");

exit(1);

} else {

// Parent process

printf("Parent process (PID: %d) going to sleep...\n", getpid());

sleep(5); // Sleep for 5 seconds

printf("Parent process (PID: %d) woke up!\n", getpid());

// Optionally, wait for the child process to finish

wait(NULL);

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

}

return 0;

}

**Q.2 Write the simulation program using FCFS. The arrival time and first CPU bursts of different jobs should be input to the system. Assume the fixed I/O waiting time (2 units). The next CPU burst should be generated using random function. The output should give the Gantt chart, Turnaround Time and Waiting time for each process and average times**

**ANS:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define MAX\_PROCESSES 10

typedef struct {

int id;

int arrival\_time;

int burst\_time;

int waiting\_time;

int turnaround\_time;

} Process;

void generate\_random\_burst(Process \*p) {

// Generate a random CPU burst between 1 and 10

p->burst\_time = rand() % 10 + 1;

}

void print\_gantt\_chart(Process processes[], int n) {

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

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

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

}

printf("\n");

}

int main() {

srand(time(NULL)); // Seed for random number generation

Process processes[MAX\_PROCESSES];

int n;

// Input number of processes

printf("Enter the number of processes (max %d): ", MAX\_PROCESSES);

scanf("%d", &n);

// Input arrival times and initial burst times

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

processes[i].id = i + 1;

printf("Enter arrival time and first CPU burst for Process P%d: ", processes[i].id);

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

}

// Sort processes by arrival time (FCFS)

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

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

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

Process temp = processes[j];

processes[j] = processes[j + 1];

processes[j + 1] = temp;

}

}

}

int current\_time = 0;

// Calculate waiting and turnaround times

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

if (current\_time < processes[i].arrival\_time) {

current\_time = processes[i].arrival\_time; // Idle CPU until the process arrives

}

processes[i].waiting\_time = current\_time - processes[i].arrival\_time;

current\_time += processes[i].burst\_time; // CPU time after process is finished

processes[i].turnaround\_time = processes[i].waiting\_time + processes[i].burst\_time;

// Generate next CPU burst (for demonstration)

generate\_random\_burst(&processes[i]);

}

// Print Gantt Chart

print\_gantt\_chart(processes, n);

// Print results

printf("\nProcess\tArrival\tBurst\tWaiting\tTurnaround\n");

int total\_waiting\_time = 0;

int total\_turnaround\_time = 0;

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

printf("P%d\t%d\t%d\t%d\t%d\n",

processes[i].id,

processes[i].arrival\_time,

processes[i].burst\_time,

processes[i].waiting\_time,

processes[i].turnaround\_time);

total\_waiting\_time += processes[i].waiting\_time;

total\_turnaround\_time += processes[i].turnaround\_time;

}

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

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

return 0;

}

**Slip 8 ,**

**Q.1 Write a C program to accept the number of process and resources and find the need matrix**

**content and display it.**

**Ans:**

#include <stdio.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

void calculate\_need(int processes[], int allocation[][MAX\_RESOURCES],

int max[][MAX\_RESOURCES], int need[][MAX\_RESOURCES], int n, int m) {

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

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

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

}

}

}

int main() {

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

int processes[MAX\_PROCESSES];

int allocation[MAX\_PROCESSES][MAX\_RESOURCES];

int max[MAX\_PROCESSES][MAX\_RESOURCES];

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

// Input number of processes and resources

printf("Enter number of processes: ");

scanf("%d", &n);

printf("Enter number of resources: ");

scanf("%d", &m);

// Input allocation matrix

printf("Enter allocation matrix:\n");

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

processes[i] = i;

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

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

}

}

// Input max matrix

printf("Enter maximum matrix:\n");

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

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

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

}

}

// Calculate need matrix

calculate\_need(processes, allocation, max, need, n, m);

// Display 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");

}

return 0;

}

**OR**

**Q.2. Write the simulation program to implement demand paging and show the page scheduling**

**and total number of page faults for the following given page reference string. Give input n =3 as**

**the number of memory frames.**

**Reference String : 12,15,12,18,6,8,11,12,19,12,6,8,12,15,19,8**

**Implement OPT [20 marks]**

**ANS:**

#include <stdio.h>

#define MAX\_FRAMES 3

#define MAX\_REFERENCES 16

void print\_frames(int frames[], int frame\_count) {

for (int i = 0; i < frame\_count; i++) {

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

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

} else {

printf("- "); // For empty frames

}

}

printf("\n");

}

int find\_optimal\_page(int frames[], int n, int reference[], int index, int frame\_count) {

int farthest = index;

int optimal\_page = -1;

for (int i = 0; i < frame\_count; i++) {

int j;

for (j = index; j < n; j++) {

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

if (j > farthest) {

farthest = j;

optimal\_page = i;

}

break;

}

}

// If the page is not found, it is the optimal candidate for replacement

if (j == n) {

return i;

}

}

return (optimal\_page == -1) ? 0 : optimal\_page; // If all pages are found, return the first page

}

int main() {

int frames[MAX\_FRAMES];

int reference\_string[MAX\_REFERENCES] = {12, 15, 12, 18, 6, 8, 11, 12, 19, 12, 6, 8, 12, 15, 19, 8};

int n = sizeof(reference\_string) / sizeof(reference\_string[0]);

int page\_faults = 0;

// Initialize frames

for (int i = 0; i < MAX\_FRAMES; i++) {

frames[i] = -1; // -1 indicates an empty frame

}

printf("Reference String: ");

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

printf("%d ", reference\_string[i]);

}

printf("\n");

printf("Frame State after each reference:\n");

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

int page\_found = 0;

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

for (int j = 0; j < MAX\_FRAMES; j++) {

if (frames[j] == reference\_string[i]) {

page\_found = 1; // Page is found in memory

break;

}

}

// If page not found, a page fault occurs

if (!page\_found) {

// Find the optimal page to replace

int replace\_index = find\_optimal\_page(frames, n, reference\_string, i + 1, MAX\_FRAMES);

frames[replace\_index] = reference\_string[i]; // Replace the page

page\_faults++;

}

print\_frames(frames, MAX\_FRAMES); // Print current state of frames

}

printf("Total number of page faults: %d\n", page\_faults);

return 0;

}

**Slip 9 ,**

**Q.1 Write a program to create a child process using fork().The parent should goto sleep state and**

**child process should begin its execution. In the child process, use execl() to execute the “ls”**

**command.**

**Ans:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

pid\_t pid = fork(); // Create a new process

if (pid < 0) {

// Fork failed

perror("Fork failed");

return 1;

} else if (pid == 0) {

// Child process

printf("Child process (PID: %d) executing 'ls' command...\n", getpid());

// Execute the 'ls' command

execl("/bin/ls", "ls", NULL);

// If execl returns, it must have failed

perror("execl failed");

exit(1);

} else {

// Parent process

printf("Parent process (PID: %d) going to sleep...\n", getpid());

sleep(5); // Sleep for 5 seconds

printf("Parent process (PID: %d) woke up!\n", getpid());

// Optionally, wait for the child process to finish

wait(NULL);

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

}

return 0;

}

**OR**

**Q.2 Write the program to simulate Round Robin (RR) scheduling. The arrival time and first CPU-**

**burst for different n number of processes should be input to the algorithm. Also give the time**

**quantum as input. Assume the fixed IO waiting time (2 units). The next CPU-burst should be**

**generated randomly. The output should give Gantt chart, turnaround time and waiting time for each**

**process. Also find the average waiting time and turnaround time.**

**Ans:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <time.h>

#define MAX\_PROCESSES 10

typedef struct {

int id;

int arrival\_time;

int burst\_time;

int remaining\_time;

int waiting\_time;

int turnaround\_time;

} Process;

void generate\_random\_burst(Process \*p) {

// Generate a random CPU burst between 1 and 10

p->burst\_time = rand() % 10 + 1;

p->remaining\_time = p->burst\_time; // Initialize remaining time

}

void print\_gantt\_chart(int order[], int size) {

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

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

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

}

printf("\n");

}

int main() {

srand(time(NULL)); // Seed for random number generation

Process processes[MAX\_PROCESSES];

int n; // Number of processes

int time\_quantum;

// Input number of processes

printf("Enter the number of processes (max %d): ", MAX\_PROCESSES);

scanf("%d", &n);

// Input arrival times and initial burst times

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

processes[i].id = i + 1; // Process ID

printf("Enter arrival time and first CPU burst for Process P%d: ", processes[i].id);

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

generate\_random\_burst(&processes[i]); // Generate a new random burst

}

// Input time quantum

printf("Enter the time quantum: ");

scanf("%d", &time\_quantum);

int time = 0; // Current time

bool all\_done = false;

int order[MAX\_PROCESSES \* 10]; // Store the order of execution for Gantt chart

int index = 0;

while (!all\_done) {

all\_done = true; // Assume all processes are done

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

if (processes[i].remaining\_time > 0) {

all\_done = false; // At least one process is not done

if (processes[i].arrival\_time <= time) {

int burst = (processes[i].remaining\_time < time\_quantum) ? processes[i].remaining\_time : time\_quantum;

processes[i].remaining\_time -= burst;

time += burst; // Increment current time

order[index++] = processes[i].id; // Record the order of execution

if (processes[i].remaining\_time == 0) {

processes[i].turnaround\_time = time - processes[i].arrival\_time;

processes[i].waiting\_time = processes[i].turnaround\_time - processes[i].burst\_time;

}

}

}

}

}

// Print Gantt Chart

print\_gantt\_chart(order, index);

// Print results

printf("\nProcess\tArrival\tBurst\tWaiting\tTurnaround\n");

int total\_waiting\_time = 0;

int total\_turnaround\_time = 0;

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

printf("P%d\t%d\t%d\t%d\t%d\n",

processes[i].id,

processes[i].arrival\_time,

processes[i].burst\_time,

processes[i].waiting\_time,

processes[i].turnaround\_time);

total\_waiting\_time += processes[i].waiting\_time;

total\_turnaround\_time += processes[i].turnaround\_time;

}

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

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

return 0;

}