**Slip 15 .**

**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());

// Wait for the child process to finish

wait(NULL);

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

}

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 :7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2**

**Implement LRU**

**Ans:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_FRAMES 10

#define MAX\_PAGES 12

int find\_page(int frames[], int n, int page) {

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

if (frames[i] == page) {

return i; // Page found

}

}

return -1; // Page not found

}

int find\_lru\_index(int time[], int n) {

int min\_index = 0;

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

if (time[i] < time[min\_index]) {

min\_index = i; // Find the least recently used page

}

}

return min\_index;

}

int main() {

int frames[MAX\_FRAMES];

int reference\_string[MAX\_PAGES] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};

int n, page\_faults = 0;

int time[MAX\_FRAMES]; // To track the last accessed time of pages

int current\_time = 0;

// Input number of frames

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

scanf("%d", &n);

// Initialize frames and time arrays

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

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

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

}

printf("Reference String: ");

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

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

}

printf("\n");

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

int page = reference\_string[i];

int page\_index = find\_page(frames, n, page);

if (page\_index == -1) {

// Page fault occurs

page\_faults++;

int lru\_index = find\_lru\_index(time, n);

frames[lru\_index] = page; // Replace LRU page

time[lru\_index] = current\_time; // Update access time

} else {

// Page is found in frames, update access time

time[page\_index] = current\_time;

}

current\_time++; // Increment current time

}

// Print the results

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

return 0;

}

**Slip 16 .**

**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\_operations() {

// Simulating some operations

for (long i = 0; i < 100000000; i++); // A simple loop to consume time

}

int main() {

clock\_t start\_time, end\_time;

double cpu\_time\_used;

// Start the timer

start\_time = clock();

// Perform the operations

perform\_operations();

// Stop the timer

end\_time = clock();

// Calculate the time taken

cpu\_time\_used = ((double)(end\_time - start\_time)) / CLOCKS\_PER\_SEC;

// Output the execution time

printf("Execution time: %f seconds\n", cpu\_time\_used);

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 =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**

**Ans:**

#include <stdio.h>

#define MAX\_FRAMES 3

#define MAX\_REFERENCES 15

// Function to find the page to replace using the OPT algorithm

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; // Track the page that will be used furthest in the future

}

break;

}

}

// If the page is not found in future references, 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");

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

int page = reference\_string[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] == page) {

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] = page; // Replace the page

page\_faults++;

printf("Page fault occurred! Replacing page with: %d\n", page);

}

// Print the current state of frames

printf("Current frames: ");

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

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

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

}

}

printf("\n");

}

// Print total page faults

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

return 0;

}

**Slip 17 .**

**Q.1 Write the program to calculate minimum number of resources needed to avoid**

**deadlock.**

**Ans:**

#include <stdio.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

void calculate\_minimum\_resources(int max[][MAX\_RESOURCES], int allocation[][MAX\_RESOURCES], int need[][MAX\_RESOURCES], int processes, int resources) {

int total\_resources[MAX\_RESOURCES] = {0};

int available[MAX\_RESOURCES] = {0};

// Calculate the total resources needed to avoid deadlock

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

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

total\_resources[j] += max[i][j]; // Total maximum resources required

available[j] += allocation[i][j]; // Total currently allocated resources

}

}

// Calculate the resources needed to ensure no deadlock

int min\_resources[MAX\_RESOURCES];

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

min\_resources[j] = total\_resources[j] - available[j];

if (min\_resources[j] < 0) {

min\_resources[j] = 0; // Minimum resources cannot be negative

}

}

// Output the results

printf("Minimum number of resources needed to avoid deadlock:\n");

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

printf("Resource %d: %d\n", j + 1, min\_resources[j]);

}

}

int main() {

int processes, resources;

int max[MAX\_PROCESSES][MAX\_RESOURCES], allocation[MAX\_PROCESSES][MAX\_RESOURCES], need[MAX\_PROCESSES][MAX\_RESOURCES];

// Input number of processes and resources

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

scanf("%d", &processes);

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

scanf("%d", &resources);

// Input maximum resource requirements for each process

printf("Enter the maximum resource matrix (max[i][j]):\n");

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

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

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

}

}

// Input allocation matrix for each process

printf("Enter the allocation matrix (allocation[i][j]):\n");

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

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

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

}

}

// Calculate the need matrix

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

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

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

}

}

// Calculate the minimum resources needed

calculate\_minimum\_resources(max, allocation, need, processes, resources);

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=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**

**Ans:**

#include <stdio.h>

#define MAX\_FRAMES 3

#define MAX\_REFERENCES 16

// Function to find the page to replace using the OPT algorithm

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; // Track the page that will be used furthest in the future

}

break;

}

}

// If the page is not found in future references, 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");

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

int page = reference\_string[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] == page) {

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] = page; // Replace the page

page\_faults++;

printf("Page fault occurred! Replacing page with: %d\n", page);

}

// Print the current state of frames

printf("Current frames: ");

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

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

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

}

}

printf("\n");

}

// Print total page faults

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

return 0;

}

**Slip 18 ,**

**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 resources, int max[][MAX\_RESOURCES], int allocation[][MAX\_RESOURCES], int need[][MAX\_RESOURCES]) {

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

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

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

}

}

}

void display\_matrix(int matrix[][MAX\_RESOURCES], int processes, int resources) {

printf("Matrix:\n");

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

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

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

}

printf("\n");

}

}

int main() {

int processes, resources;

int max[MAX\_PROCESSES][MAX\_RESOURCES], allocation[MAX\_PROCESSES][MAX\_RESOURCES], need[MAX\_PROCESSES][MAX\_RESOURCES];

// Input number of processes and resources

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

scanf("%d", &processes);

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

scanf("%d", &resources);

// Input maximum resource requirements for each process

printf("Enter the maximum resource matrix (max[i][j]):\n");

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

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

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

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

}

}

// Input allocation matrix for each process

printf("Enter the allocation matrix (allocation[i][j]):\n");

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

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

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

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

}

}

// Calculate the need matrix

calculate\_need(processes, resources, max, allocation, need);

// Display the need matrix

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

display\_matrix(need, processes, resources);

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

**Implement OPT**

**Ans:**

#include <stdio.h>

#define MAX\_FRAMES 10

#define MAX\_REFERENCES 16

// Function to find the page to replace using the OPT algorithm

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; // Track the page that will be used furthest in the future

}

break;

}

}

// If the page is not found in future references, 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;

int frame\_count;

// Input the number of memory frames

printf("Enter the number of memory frames (1 to %d): ", MAX\_FRAMES);

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

if (frame\_count < 1 || frame\_count > MAX\_FRAMES) {

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

return 1;

}

// Initialize frames

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

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

int page = reference\_string[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] == page) {

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, frame\_count);

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

page\_faults++;

printf("Page fault occurred! Replacing page with: %d\n", page);

}

// Print the current state of frames

printf("Current frames: ");

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

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

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

}

}

printf("\n");

}

// Print total page faults

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

return 0;

}

**Slip 19 .**

**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;

// Create a child process

pid = fork();

if (pid < 0) {

// Error occurred

perror("Fork failed");

exit(1);

} else if (pid == 0) {

// Child process

printf("Child Process: Executing 'ls' command...\n");

execl("/bin/ls", "ls", NULL); // Execute the 'ls' command

// If execl returns, it means there was an error

perror("execl failed");

exit(1);

} else {

// Parent process

printf("Parent Process: Going to sleep...\n");

sleep(5); // Sleep for 5 seconds

printf("Parent Process: Waking up after sleep.\n");

// Wait for the child process to complete

wait(NULL);

printf("Parent Process: Child has finished executing.\n");

}

return 0;

}

**Q.2 Write the program to simulate Non-preemptive Priority scheduling. The arrival time and first CPU burst and priority for different n number of processes should be input to the algorithm.**

**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 <time.h>

#define MAX\_PROCESSES 10

typedef struct {

int pid; // Process ID

int arrival; // Arrival time

int burst; // Initial CPU burst time

int priority; // Priority

int waiting; // Waiting time

int turnaround; // Turnaround time

} Process;

// Function to compare processes based on priority and arrival time

int compare(const void \*a, const void \*b) {

Process \*p1 = (Process \*)a;

Process \*p2 = (Process \*)b;

if (p1->arrival == p2->arrival) {

return p1->priority - p2->priority; // Higher priority first

}

return p1->arrival - p2->arrival; // Earlier arrival first

}

// Function to generate a random CPU burst

int generate\_random\_burst() {

return rand() % 5 + 1; // Random burst between 1 and 5

}

int main() {

int n;

Process processes[MAX\_PROCESSES];

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

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

// Input number of processes

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

scanf("%d", &n);

// Input process details

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

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

printf("Enter arrival time, burst time, and priority for Process %d:\n", processes[i].pid);

printf("Arrival Time: ");

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

printf("Burst Time: ");

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

printf("Priority: ");

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

}

// Sort processes by arrival time and priority

qsort(processes, n, sizeof(Process), compare);

int time = 0;

int completed = 0;

int gantt[MAX\_PROCESSES \* 2]; // Gantt chart array

int gantt\_index = 0;

while (completed < n) {

int idx = -1;

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

// Find the next process to execute

if (processes[i].arrival <= time && processes[i].burst > 0) {

if (idx == -1 || processes[i].priority < processes[idx].priority) {

idx = i;

}

}

}

if (idx != -1) {

// Process found, execute it

gantt[gantt\_index++] = processes[idx].pid; // Add process to Gantt chart

time += processes[idx].burst; // Increment time by burst time

// Calculate waiting and turnaround times

processes[idx].waiting = time - processes[idx].arrival - processes[idx].burst;

processes[idx].turnaround = time - processes[idx].arrival;

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

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

// Mark the process as completed

processes[idx].burst = 0;

completed++;

} else {

// No process is ready, increment time

time++;

}

// Generate next CPU burst for each process after completion

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

if (processes[i].burst == 0 && completed < n) {

processes[i].burst = generate\_random\_burst(); // Assign a new random burst time

}

}

}

// Output results

printf("\nGantt Chart: ");

for (int i = 0; i < gantt\_index; i++) {

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

}

printf("\n");

// Display turnaround and waiting times

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

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

printf("P%d\t%d\t%d\t%d\t\t%d\t%d\n", processes[i].pid, processes[i].arrival, processes[i].burst, processes[i].priority, processes[i].waiting, processes[i].turnaround);

}

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 20 ,**

**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;

// Create a child process

pid = fork();

if (pid < 0) {

// Error occurred

perror("Fork failed");

exit(1);

} else if (pid == 0) {

// Child process

printf("Child Process: Executing 'ls' command...\n");

execl("/bin/ls", "ls", NULL); // Execute the 'ls' command

// If execl returns, it means there was an error

perror("execl failed");

exit(1);

} else {

// Parent process

printf("Parent Process: Going to sleep...\n");

sleep(5); // Sleep for 5 seconds

printf("Parent Process: Waking up after sleep.\n");

// Wait for the child process to complete

wait(NULL);

printf("Parent Process: Child has finished executing.\n");

}

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=3 as the number of memory frames.**

**Reference String : 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2**

**i. Implement LRU**

**Ans:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_FRAMES 10

#define MAX\_PAGES 20

// Function to check if a page is in memory

int is\_page\_in\_memory(int page, int frames[], int frame\_count) {

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

if (frames[i] == page) {

return 1; // Page found

}

}

return 0; // Page not found

}

// Function to find the index of the least recently used page

int find\_lru\_index(int frames[], int frame\_count, int time[], int current\_time) {

int lru\_index = 0;

int min\_time = current\_time; // Set initial min\_time to current\_time

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

if (time[i] < min\_time) {

min\_time = time[i];

lru\_index = i; // Update index of the least recently used page

}

}

return lru\_index;

}

int main() {

int frames[MAX\_FRAMES];

int reference\_string[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};

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

int frame\_count = 3; // Number of memory frames

int page\_faults = 0;

int time[MAX\_FRAMES]; // To store the last used time of pages

// Initialize frames and time arrays

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

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

time[i] = -1; // -1 indicates the page has not been used

}

printf("Reference String: ");

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

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

}

printf("\n");

for (int current\_time = 0; current\_time < n; current\_time++) {

int page = reference\_string[current\_time];

// Check if the page is already in memory

if (!is\_page\_in\_memory(page, frames, frame\_count)) {

// Page fault occurs

page\_faults++;

int replace\_index = -1;

// Check for an empty frame

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

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

replace\_index = i; // Replace empty frame

break;

}

}

// If no empty frame is found, use LRU algorithm to replace

if (replace\_index == -1) {

replace\_index = find\_lru\_index(frames, frame\_count, time, current\_time);

}

// Replace the page in memory

frames[replace\_index] = page;

time[replace\_index] = current\_time; // Update last used time

printf("Page fault occurred! Replacing frame with: %d\n", page);

} else {

// Update last used time for this page

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

if (frames[i] == page) {

time[i] = current\_time; // Update the time of the current page

break;

}

}

}

// Print the current state of frames

printf("Current frames: ");

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

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

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

}

}

printf("\n");

}

// Print total page faults

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

return 0;

}