**Q NO 1**

Round Robin and First Come First Serve - In Paper (A & B)

Enter No of Process - Assign Burst Time

Calculate average waiting and turnaround time

**Code**

#include <stdio.h>

int n=5;

void fcfs(int n, int burst\_time[]) {

    int waiting\_time[n], turnaround\_time[n];

    int total\_wt = 0, total\_tat = 0;

    int completion\_time[n];

    // Completion time for first process = burst time

    completion\_time[0] = burst\_time[0];

    for (int i = 1; i < n; i++) {

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

    }

    for (int i = 0; i < n; i++) {

        turnaround\_time[i] = completion\_time[i];  // AT=0 for all processes

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

        total\_wt += waiting\_time[i];

        total\_tat += turnaround\_time[i];

    }

    printf("\nFCFS Scheduling:\n");

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

    for (int i = 0; i < n; i++) {

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

    }

    printf("Average Waiting Time = %.2f\n", (float)total\_wt / n);

    printf("Average Turnaround Time = %.2f\n", (float)total\_tat / n);

}

void round\_robin(int n, int burst\_time[], int time\_quantum) {

    int rem\_bt[n];

    int waiting\_time[n], turnaround\_time[n];

    int t = 0; // Current time

    int done = 0;

    for (int i = 0; i < n; i++) {

        rem\_bt[i] = burst\_time[i];

        waiting\_time[i] = 0;

        turnaround\_time[i] = 0;

    }

    while (done < n) {

        done = 0;

        for (int i = 0; i < n; i++) {

            if (rem\_bt[i] > 0) {

                if (rem\_bt[i] > time\_quantum) {

                    t += time\_quantum;

                    rem\_bt[i] -= time\_quantum;

                } else {

                    t += rem\_bt[i];

                    waiting\_time[i] = t - burst\_time[i];

                    rem\_bt[i] = 0;

                }

            }

            if (rem\_bt[i] == 0)

                done++;

        }

    }

    int total\_wt = 0, total\_tat = 0;

    for (int i = 0; i < n; i++) {

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

        total\_wt += waiting\_time[i];

        total\_tat += turnaround\_time[i];

    }

    printf("\nRound Robin Scheduling (Time Quantum = %d):\n", time\_quantum);

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

    for (int i = 0; i < n; i++) {

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

    }

    printf("Average Waiting Time = %.2f\n", (float)total\_wt / n);

    printf("Average Turnaround Time = %.2f\n", (float)total\_tat / n);

}

int main() {

    int  time\_quantum;

    printf("Enter number of processes: ");

    scanf("%d", &n);

    int burst\_time[n];

    printf("Enter burst time for each process:\n");

    for (int i = 0; i < n; i++) {

        printf("P%d: ", i + 1);

        scanf("%d", &burst\_time[i]);

    }

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

    scanf("%d", &time\_quantum);

    fcfs(n, burst\_time);

    round\_robin(n, burst\_time, time\_quantum);

    return 0;

}

A screenshot of a computer program

AI-generated content may be incorrect.

**Q NO 2**

Write a C program to implement the deadlock detection algorithm. Take the Allocation, Request, and

Available matrices as input and identify which processes are deadlocked. Extend the deadlock detection program to perform deadlock recovery by terminating one or more processes until the deadlock is resolved.

**Code**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

void print\_vector(int vec[], int size) {

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

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

}

printf("\n");

}

bool less\_than\_or\_equal(int a[], int b[], int size) {

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

if (a[i] > b[i]) return false;

}

return true;

}

void add\_vectors(int a[], int b[], int size) {

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

a[i] += b[i];

}

}

void subtract\_vectors(int a[], int b[], int size) {

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

a[i] -= b[i];

}

}

void deadlock\_detection(int n, int m, int Allocation[][MAX\_RESOURCES], int Request[][MAX\_RESOURCES], int Available[], bool deadlocked[], int \*deadlock\_count) {

int Work[MAX\_RESOURCES];

bool Finish[MAX\_PROCESSES] = { false };

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

Work[i] = Available[i];

}

bool progress;

do {

progress = false;

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

if (!Finish[i] && less\_than\_or\_equal(Request[i], Work, m)) {

add\_vectors(Work, Allocation[i], m);

Finish[i] = true;

progress = true;

}

}

} while (progress);

\*deadlock\_count = 0;

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

if (!Finish[i]) {

deadlocked[i] = true;

(\*deadlock\_count)++;

} else {

deadlocked[i] = false;

}

}

}

void deadlock\_recovery(int n, int m, int Allocation[][MAX\_RESOURCES], int Request[][MAX\_RESOURCES], int Available[], bool deadlocked[], int \*deadlock\_count) {

while (\*deadlock\_count > 0) {

printf("\nDeadlocked processes detected: ");

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

if (deadlocked[i]) {

printf("P%d ", i);

}

}

printf("\n");

// Terminate one deadlocked process - here we pick the first one found

int victim = -1;

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

if (deadlocked[i]) {

victim = i;

break;

}

}

if (victim == -1) {

printf("No victim found for recovery.\n");

break;

}

printf("Terminating process P%d to recover from deadlock...\n", victim);

// Release resources allocated to victim

add\_vectors(Available, Allocation[victim], m);

// Set Allocation and Request of victim to zero to simulate termination

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

Allocation[victim][j] = 0;

Request[victim][j] = 0;

}

// Re-run deadlock detection after recovery

deadlock\_detection(n, m, Allocation, Request, Available, deadlocked, deadlock\_count);

}

if (\*deadlock\_count == 0) {

printf("\nDeadlock resolved after recovery.\n");

}

}

int main() {

int n, m;

printf("Enter number of processes: ");

scanf("%d", &n);

printf("Enter number of resource types: ");

scanf("%d", &m);

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

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

int Available[MAX\_RESOURCES];

bool deadlocked[MAX\_PROCESSES];

printf("Enter Allocation matrix (%d processes x %d resources):\n", n, m);

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

printf("Process P%d: ", i);

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

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

}

}

printf("Enter Request matrix (%d processes x %d resources):\n", n, m);

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

printf("Process P%d: ", i);

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

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

}

}

printf("Enter Available resources vector (%d resources):\n", m);

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

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

}

int deadlock\_count = 0;

deadlock\_detection(n, m, Allocation, Request, Available, deadlocked, &deadlock\_count);

if (deadlock\_count == 0) {

printf("\nNo deadlock detected.\n");

} else {

printf("\nDeadlock detected in processes: ");

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

if (deadlocked[i]) {

printf("P%d ", i);

}

}

printf("\n");

deadlock\_recovery(n, m, Allocation, Request, Available, deadlocked, &deadlock\_count);

}

return 0;

}

A screen shot of a computer program

AI-generated content may be incorrect.

**Q NO 3**

Write a C program to implement the Banker's Algorithm for deadlock avoidance. Take input for the

number of processes, number of resources, allocation matrix, maximum demand matrix, and available

resources. Display whether the system is in a safe state. Modify your Banker's Algorithm program to take an additional resource request from a particular process and check if the request can be safely granted.

**Code**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

bool less\_than\_or\_equal(int a[], int b[], int size) {

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

if (a[i] > b[i]) return false;

}

return true;

}

void add\_vectors(int a[], int b[], int size) {

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

a[i] += b[i];

}

}

void subtract\_vectors(int a[], int b[], int size) {

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

a[i] -= b[i];

}

}

bool is\_safe\_state(int n, int m, int Allocation[][MAX\_RESOURCES], int Need[][MAX\_RESOURCES], int Available[]) {

bool Finish[MAX\_PROCESSES] = {false};

int Work[MAX\_RESOURCES];

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

Work[i] = Available[i];

}

int count = 0;

while (count < n) {

bool found = false;

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

if (!Finish[i] && less\_than\_or\_equal(Need[i], Work, m)) {

add\_vectors(Work, Allocation[i], m);

Finish[i] = true;

found = true;

count++;

}

}

if (!found) {

break; // no process could proceed

}

}

// If all processes are finished, safe state

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

if (!Finish[i]) return false;

}

return true;

}

int main() {

int n, m;

printf("Enter number of processes: ");

scanf("%d", &n);

printf("Enter number of resource types: ");

scanf("%d", &m);

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

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

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

int Available[MAX\_RESOURCES];

printf("Enter Allocation matrix (%d processes x %d resources):\n", n, m);

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

printf("Process P%d: ", i);

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

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

}

}

printf("Enter Maximum Demand matrix (%d processes x %d resources):\n", n, m);

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

printf("Process P%d: ", i);

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

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

}

}

printf("Enter Available resources vector (%d resources):\n", m);

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

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

}

// Calculate Need matrix = Max - Allocation

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

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

Need[i][j] = Max[i][j] - Allocation[i][j];

}

}

// Check system safe state initially

if (is\_safe\_state(n, m, Allocation, Need, Available)) {

printf("\nSystem is in a safe state.\n");

} else {

printf("\nSystem is NOT in a safe state.\n");

return 0; // no need to proceed further if unsafe initially

}

// Additional resource request

int process\_id;

printf("\nEnter process number (0 to %d) making additional resource request: ", n - 1);

scanf("%d", &process\_id);

if (process\_id < 0 || process\_id >= n) {

printf("Invalid process number.\n");

return 1;

}

int Request[MAX\_RESOURCES];

printf("Enter resource request vector for process P%d:\n", process\_id);

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

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

}

// Check if Request <= Need

if (!less\_than\_or\_equal(Request, Need[process\_id], m)) {

printf("Error: Process has exceeded its maximum claim.\n");

return 1;

}

// Check if Request <= Available

if (!less\_than\_or\_equal(Request, Available, m)) {

printf("Resources are not available currently. Process must wait.\n");

return 1;

}

// Pretend to allocate requested resources

subtract\_vectors(Available, Request, m);

add\_vectors(Allocation[process\_id], Request, m);

subtract\_vectors(Need[process\_id], Request, m);

// Check if new state is safe

if (is\_safe\_state(n, m, Allocation, Need, Available)) {

printf("Request can be safely granted.\n");

} else {

// Rollback since unsafe

add\_vectors(Available, Request, m);

subtract\_vectors(Allocation[process\_id], Request, m);

add\_vectors(Need[process\_id], Request, m);

printf("Request cannot be granted as it leads to unsafe state.\n");

}

return 0;

}

A screenshot of a computer program

AI-generated content may be incorrect.

**Q NO 4**

A virtual memory system divides the logical address space into pages and the physical memory into frames. You are given a list of logical page references made by a running process. Write a program to simulate the basic paging technique of memory management.

Take the number of pages and number of frames as input.

Map each page to a frame.

If a frame is not available, show that the page cannot be loaded.

Display the final page-to-frame mapping.

**Code**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PAGES 100

#define MAX\_FRAMES 100

int main() {

    int num\_pages, num\_frames;

    int page\_references[MAX\_PAGES];

    int page\_frame\_map[MAX\_PAGES];  // Stores frame number assigned to each page (-1 if none)

    bool frame\_occupied[MAX\_FRAMES]; // Keeps track of which frames are occupied

    printf("Enter number of pages: ");

    scanf("%d", &num\_pages);

    printf("Enter number of frames: ");

    scanf("%d", &num\_frames);

    printf("Enter page references (%d pages):\n", num\_pages);

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

        scanf("%d", &page\_references[i]);

    }

    // Initialize mapping to -1 (no frame assigned)

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

        page\_frame\_map[i] = -1;

    }

    // Initialize all frames as free

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

        frame\_occupied[i] = false;

    }

    int next\_free\_frame = 0;

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

        int page = page\_references[i];

        if (page\_frame\_map[page] != -1) {

            // Page already mapped, just continue

            printf("Page %d is already mapped to Frame %d\n", page, page\_frame\_map[page]);

            continue;

        }

        if (next\_free\_frame < num\_frames) {

            // Assign next free frame to this page

            page\_frame\_map[page] = next\_free\_frame;

            frame\_occupied[next\_free\_frame] = true;

            printf("Mapping Page %d to Frame %d\n", page, next\_free\_frame);

            next\_free\_frame++;

        } else {

            // No free frame available

            printf("No free frame available for Page %d, cannot load this page\n", page);

        }

    }

    printf("\nFinal page to frame mapping:\n");

    printf("Page\tFrame\n");

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

        if (page\_frame\_map[i] != -1) {

            printf("%d\t%d\n", i, page\_frame\_map[i]);

        } else {

            printf("%d\tNot loaded\n", i);

        }

    }

    return 0;

}

A screen shot of a computer

AI-generated content may be incorrect.

**Q NO 5**

A web browser keeps track of the last time each tab (page) was used. If new tabs exceed available

memory, it closes the least recently used tab. Simulate the LRU page replacement algorithm in C:

Accept the reference string and number of frames as input.

Replace the page that hasn’t been used for the longest time.

Display page faults and the state of memory after each access.

**Code**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PAGES 100

#define MAX\_FRAMES 100

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

printf("Frames: ");

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

if (frames[i] == -1)

printf("- ");

else

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

}

printf("\n");

}

// Find index of page in frames, or -1 if not found

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

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

if (frames[i] == page) return i;

}

return -1;

}

// Find the index of the least recently used page

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

int min\_time = time[0], min\_index = 0;

for (int i = 1; i < frame\_count; i++) {

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

min\_time = time[i];

min\_index = i;

}

}

return min\_index;

}

int main() {

int n, frames\_count;

int pages[MAX\_PAGES];

int frames[MAX\_FRAMES];

int time[MAX\_FRAMES]; // To track last use time of each frame

int page\_faults = 0;

printf("Enter number of pages in reference string: ");

scanf("%d", &n);

printf("Enter reference string (pages):\n");

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

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

}

printf("Enter number of frames: ");

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

// Initialize frames and time arrays

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

frames[i] = -1; // empty

time[i] = 0;

}

int current\_time = 0;

printf("\nSimulating LRU Page Replacement:\n");

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

int page = pages[i];

current\_time++;

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

if (page\_index != -1) {

// Page hit - update last used time

time[page\_index] = current\_time;

printf("Page %d accessed - No page fault.\n", page);

} else {

// Page fault

page\_faults++;

// Check if there is an empty frame

int empty\_index = -1;

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

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

empty\_index = j;

break;

}

}

if (empty\_index != -1) {

// Place page in empty frame

frames[empty\_index] = page;

time[empty\_index] = current\_time;

printf("Page %d loaded into free frame %d - Page fault #%d\n", page, empty\_index, page\_faults);

} else {

// Replace least recently used page

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

printf("Page %d replaced page %d in frame %d - Page fault #%d\n", page, frames[lru\_index], lru\_index, page\_faults);

frames[lru\_index] = page;

time[lru\_index] = current\_time;

}

}

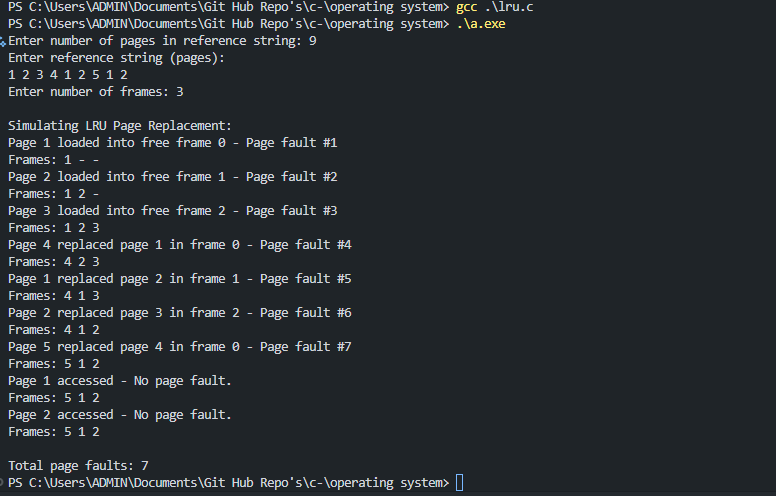
print\_frames(frames, frames\_count);

}

printf("\nTotal page faults: %d\n", page\_faults);

return 0;

}



**Q NO 6**

A system has limited cache size and wants to evict the page that is least frequently used to free up space. Write a program to:

Simulate LFU page replacement

Input a reference string and number of frames.

Keep track of frequency of each page.

Replace the page with the lowest frequency when a new page needs to be loaded.

Display total page faults and memory states.

**Code**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PAGES 100

#define MAX\_FRAMES 100

#define INF 999999

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

printf("Frames: ");

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

if (frames[i] == -1)

printf("- ");

else

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

}

printf("\n");

}

// Find index of page in frames, or -1 if not found

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

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

if (frames[i] == page) return i;

}

return -1;

}

// Find the index of the least frequently used page

// If multiple have the same frequency, choose the one with smallest index (FIFO tie breaker)

int find\_lfu\_index(int freq[], int frame\_count) {

int min\_freq = INF;

int min\_index = -1;

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

if (freq[i] < min\_freq) {

min\_freq = freq[i];

min\_index = i;

}

}

return min\_index;

}

int main() {

int n, frames\_count;

int pages[MAX\_PAGES];

int frames[MAX\_FRAMES];

int freq[MAX\_FRAMES]; // Frequency of pages in frames

int page\_faults = 0;

printf("Enter number of pages in reference string: ");

scanf("%d", &n);

printf("Enter reference string (pages):\n");

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

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

}

printf("Enter number of frames: ");

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

// Initialize frames and frequencies

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

frames[i] = -1;

freq[i] = 0;

}

printf("\nSimulating LFU Page Replacement:\n");

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

int page = pages[i];

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

if (page\_index != -1) {

// Page hit: increment frequency

freq[page\_index]++;

printf("Page %d accessed - Frequency updated to %d.\n", page, freq[page\_index]);

} else {

// Page fault

page\_faults++;

// Check for empty frame

int empty\_index = -1;

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

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

empty\_index = j;

break;

}

}

if (empty\_index != -1) {

// Load page in empty frame

frames[empty\_index] = page;

freq[empty\_index] = 1;

printf("Page %d loaded into free frame %d - Page fault #%d\n", page, empty\_index, page\_faults);

} else {

// Evict least frequently used page

int lfu\_index = find\_lfu\_index(freq, frames\_count);

printf("Page %d replaced page %d in frame %d (freq=%d) - Page fault #%d\n", page, frames[lfu\_index], lfu\_index, freq[lfu\_index], page\_faults);

frames[lfu\_index] = page;

freq[lfu\_index] = 1;

}

}

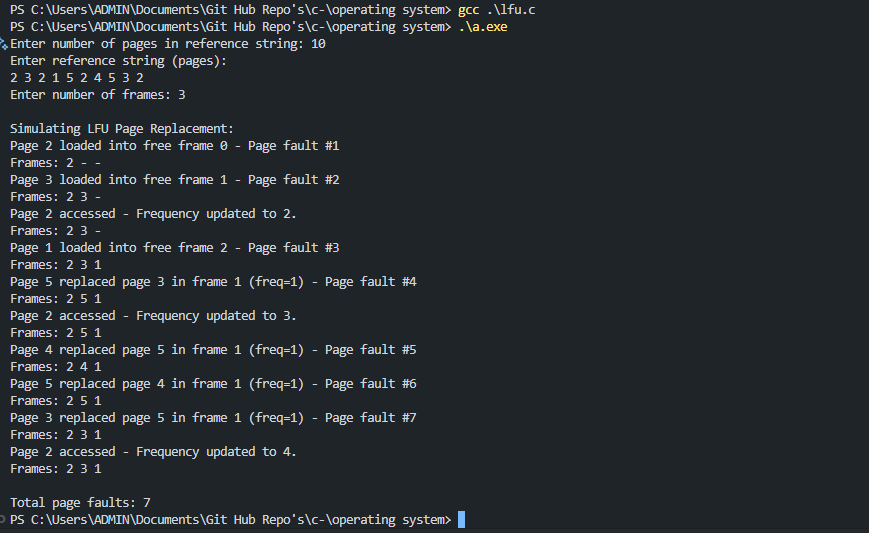
print\_frames(frames, frames\_count);

}

printf("\nTotal page faults: %d\n", page\_faults);

return 0;

}



**Q NO 7**

A virtual memory system divides the logical address space into pages and the physical memory into frames. You are given a list of logical page references made by a running process. Write a program to simulate the basic paging technique of memory management.

Take the number of pages and number of frames as input.

Map each page to a frame.

If a frame is not available, show that the page cannot be loaded.

Display the final page-to-frame mapping.

**Code**

#include <stdio.h>

#define MAX\_PAGES 100

#define MAX\_FRAMES 100

int main() {

int num\_pages, num\_frames;

int pages[MAX\_PAGES];

int frame\_for\_page[MAX\_PAGES]; // Stores which frame a page is loaded into; -1 if not loaded

int frames\_used[MAX\_FRAMES]; // 0 if free, 1 if occupied

printf("Enter number of pages in reference string: ");

scanf("%d", &num\_pages);

printf("Enter pages (logical addresses):\n");

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

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

frame\_for\_page[i] = -1;

}

printf("Enter number of frames: ");

scanf("%d", &num\_frames);

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

frames\_used[i] = 0;

}

printf("\nMapping pages to frames:\n");

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

// Find free frame

int frame\_index = -1;

for (int f = 0; f < num\_frames; f++) {

if (frames\_used[f] == 0) {

frame\_index = f;

break;

}

}

if (frame\_index != -1) {

// Assign this page to the free frame

frame\_for\_page[i] = frame\_index;

frames\_used[frame\_index] = 1;

printf("Page %d loaded into frame %d\n", pages[i], frame\_index);

} else {

printf("No free frame available for page %d. Page cannot be loaded.\n", pages[i]);

}

}

printf("\nFinal page-to-frame mapping:\n");

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

if (frame\_for\_page[i] != -1)

printf("Page %d -> Frame %d\n", pages[i], frame\_for\_page[i]);

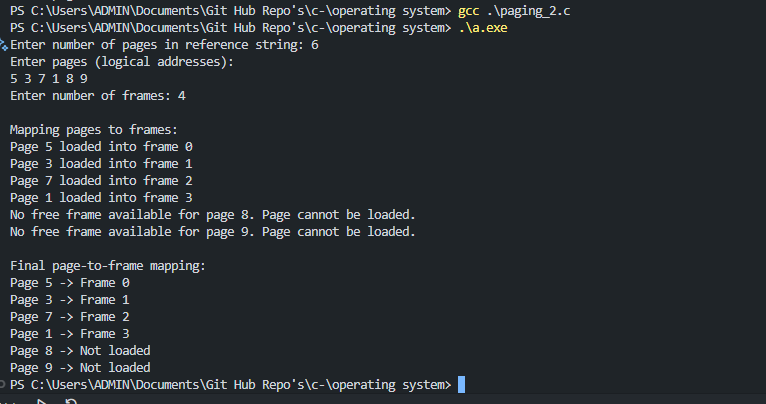
else

printf("Page %d -> Not loaded\n", pages[i]);

}

return 0;

}



**Q NO 8**

Imagine a file management system that starts with space for 100 file names (strings), but halfway through, the user decides they only need space for 30 files. Write a C program that:

Allocates memory for 100 file name entries using calloc().

Simulates name input for the first 30 entries.

Shrinks the memory to only hold 30 entries using realloc().

Displays the list and ensures memory is freed properly.

**Code**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_NAME\_LEN 100

int main() {

int initial\_size = 100;

int reduced\_size = 10;

// Allocate memory for 100 file name pointers, initialized to NULL

char \*\*file\_names = (char \*\*)calloc(initial\_size, sizeof(char \*));

if (file\_names == NULL) {

printf("Memory allocation failed.\n");

return 1;

}

// Simulate name input for first 30 entries

printf("Enter 30 file names:\n");

for (int i = 0; i < reduced\_size; i++) {

char buffer[MAX\_NAME\_LEN];

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

if (fgets(buffer, sizeof(buffer), stdin) == NULL) {

printf("Error reading input.\n");

break;

}

// Remove newline character if present

buffer[strcspn(buffer, "\n")] = 0;

// Allocate exact memory for this file name and copy it

file\_names[i] = (char \*)malloc(strlen(buffer) + 1);

if (file\_names[i] == NULL) {

printf("Memory allocation failed for file name.\n");

break;

}

strcpy(file\_names[i], buffer);

}

// Shrink memory to hold only 30 entries

char \*\*temp = (char \*\*)realloc(file\_names, reduced\_size \* sizeof(char \*));

if (temp == NULL) {

printf("Reallocation failed. Keeping original allocation.\n");

// We can continue with the original pointer, so just ignore realloc failure

} else {

file\_names = temp;

}

// Display the list of file names

printf("\nFile names stored:\n");

for (int i = 0; i < reduced\_size; i++) {

if (file\_names[i] != NULL)

printf("%d. %s\n", i + 1, file\_names[i]);

else

printf("%d. [Empty]\n", i + 1);

}

// Free allocated strings

for (int i = 0; i < reduced\_size; i++) {

free(file\_names[i]);

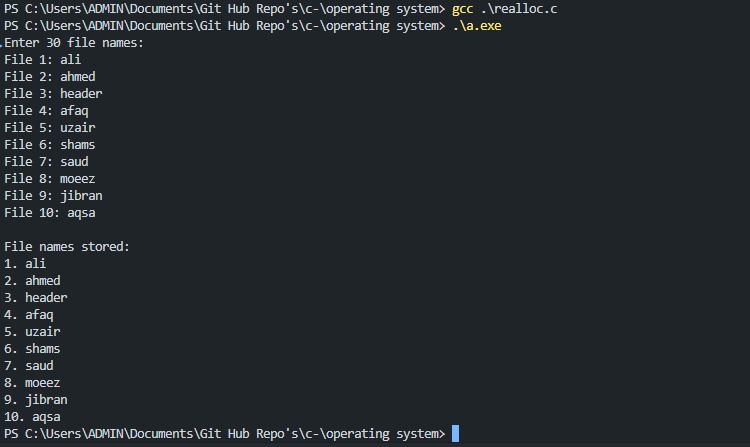
}

// Free the array of pointers

free(file\_names);

return 0;

}



**Q NO 9**

You're developing a budget tracker where users initially enter 7 daily expenses. The program uses calloc() to initialize and store the expenses. After entering all expenses, the user decides to update their data and reduce the number of tracked days to 4. Write a C program that:

Allocates memory for 7 float values using calloc().

Accepts user input for expenses.

Shrinks the memory size using realloc() to hold only the first 4 expenses.

Prints the updated expenses list.

Frees the allocated memory.

#include <stdio.h>

#include <stdlib.h>

int main() {

int initial\_days = 7;

int reduced\_days = 4;

// Allocate memory for 7 float expenses initialized to 0.0

float \*expenses = (float \*)calloc(initial\_days, sizeof(float));

if (expenses == NULL) {

printf("Memory allocation failed.\n");

return 1;

}

// Input expenses for 7 days

printf("Enter expenses for %d days:\n", initial\_days);

for (int i = 0; i < initial\_days; i++) {

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

if (scanf("%f", &expenses[i]) != 1) {

printf("Invalid input.\n");

free(expenses);

return 1;

}

}

// Shrink memory to hold only 4 expenses

float \*temp = (float \*)realloc(expenses, reduced\_days \* sizeof(float));

if (temp == NULL) {

printf("Reallocation failed. Keeping original data.\n");

// Continue with original expenses pointer if realloc fails

} else {

expenses = temp;

}

// Print updated expenses

printf("\nUpdated expenses for %d days:\n", reduced\_days);

for (int i = 0; i < reduced\_days; i++) {

printf("Day %d: %.2f\n", i + 1, expenses[i]);

}

// Free allocated memory

free(expenses);

return 0;

}

A screen shot of a computer

AI-generated content may be incorrect.

**Q NO 10**

A developer wrote a program to allocate memory using malloc for a list of 6 product prices. However, they forgot to free the memory after use, causing a memory leak. Write a C program that:

Dynamically allocates memory for 6 float product prices.

Accepts user input and calculates the average price.

Corrects the bug by freeing the memory at the end and explains what could go wrong if it's not freed.

**Code**

#include <stdio.h>

#include <stdlib.h>

int main() {

int n = 6;

float \*prices = (float \*)malloc(n \* sizeof(float));

if (prices == NULL) {

printf("Memory allocation failed.\n");

return 1;

}

// Input product prices

printf("Enter prices for %d products:\n", n);

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

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

if (scanf("%f", &prices[i]) != 1) {

printf("Invalid input.\n");

free(prices);

return 1;

}

}

// Calculate average price

float sum = 0.0f;

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

sum += prices[i];

}

float average = sum / n;

printf("\nAverage price: %.2f\n", average);

// Free allocated memory to prevent memory leak

free(prices);

return 0;

}

A screen shot of a computer

AI-generated content may be incorrect.

**Q NO 11**

A sensor records temperature values throughout the day. However, the memory is allocated without knowing how many readings will come in. Your task is to write a C program that:

Continuously accepts temperature readings one by one.

For each new reading, use realloc to increase the storage.

Stop when the user enters -1.

Display all stored readings and release the memory properly using free.

**Code**

#include <stdio.h>

#include <stdlib.h>

int main() {

float \*temperatures = NULL; // Pointer to store temperatures

int count = 0; // Number of readings stored

float input;

printf("Enter temperature readings (enter -1 to stop):\n");

while (1) {

printf("Reading %d: ", count + 1);

if (scanf("%f", &input) != 1) {

printf("Invalid input. Exiting.\n");

break;

}

if (input == -1.0f) {

// Stop input

break;

}

// Reallocate memory for one more reading

float \*temp\_ptr = realloc(temperatures, (count + 1) \* sizeof(float));

if (temp\_ptr == NULL) {

printf("Memory allocation failed.\n");

free(temperatures);

return 1;

}

temperatures = temp\_ptr;

// Store the new reading

temperatures[count] = input;

count++;

}

// Display all stored readings

printf("\nStored temperature readings:\n");

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

printf("Reading %d: %.2f\n", i + 1, temperatures[i]);

}

// Free allocated memory

free(temperatures);

return 0;

}

A screen shot of a computer

AI-generated content may be incorrect.