****

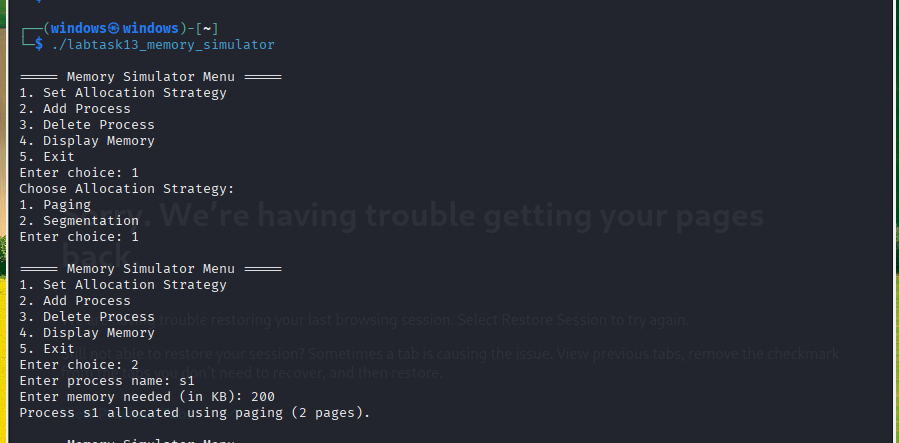
|  |
| --- |
| *BS Cyber Security Department AU* |

|  |  |
| --- | --- |
| **Registration ID** | **233026** |
| **Submitted By** | **Muhammad Sohaib Rafiq** |
| **Submitted To** | **Iram Fatima Hashmi** |
| **Date of Submission** | **05/21/2025** |
| **Lab NO.** | **13** |
| **Subject** | **OS LAB** |

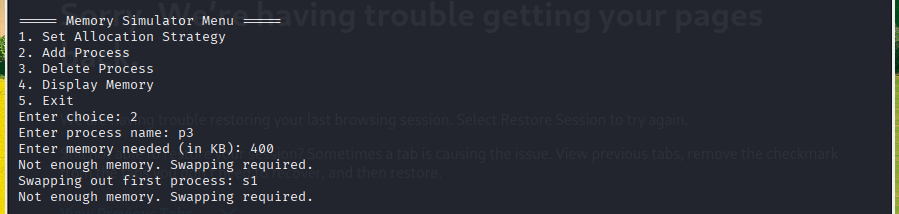
Create a menu-driven C program that:

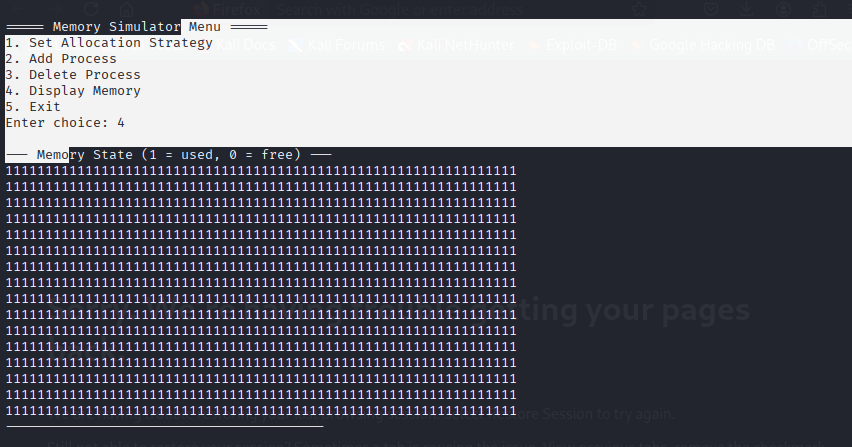
* Allows the user to define processes with memory requirements.
* Supports paging-based and segmentation-based allocation.
* Simulates memory allocation and deallocation, showing fragmentation.
* Handles swapping of processes when memory is full.

Screenshots:

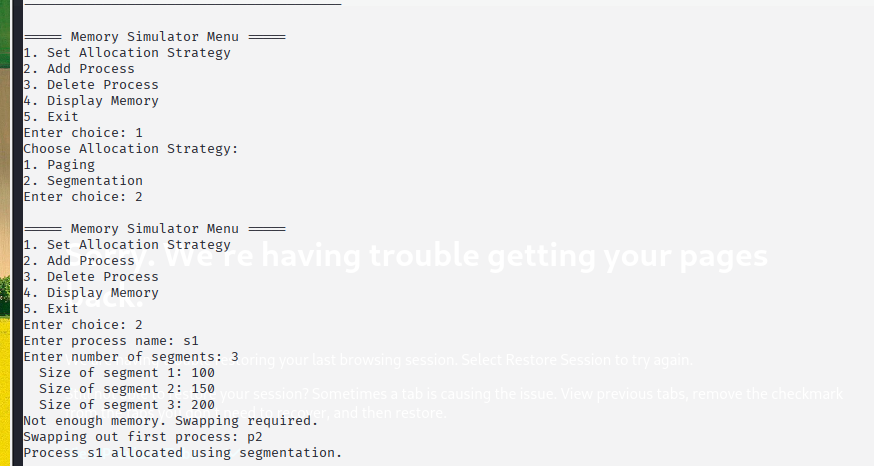


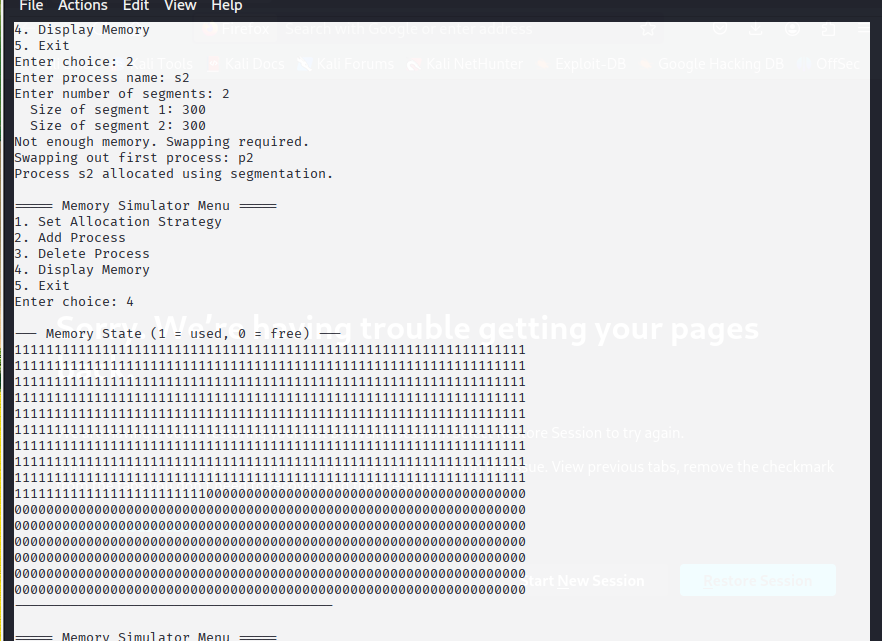




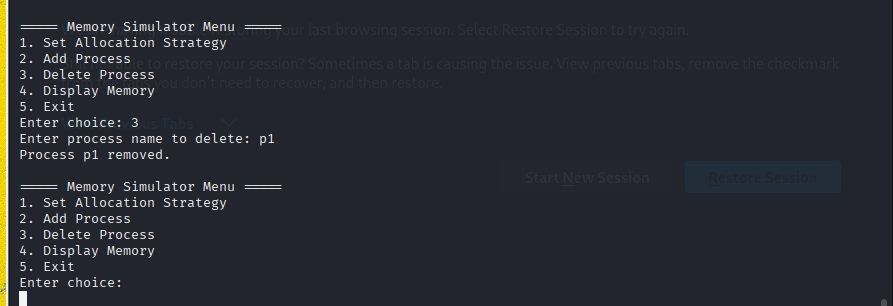


Segmentations:





De-Allocating the process:



Source code:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_PROCESSES 10

#define TOTAL\_MEMORY 1024

#define PAGE\_SIZE 128

#define MAX\_SEGMENTS 5

typedef enum { PAGING, SEGMENTATION } AllocationType;

typedef struct {

char name[20];

int memory;

int pages[10];

int segments[MAX\_SEGMENTS];

int num\_segments;

int allocated;

} Process;

Process processes[MAX\_PROCESSES];

int num\_processes = 0;

int memory[TOTAL\_MEMORY]; // Simulated memory

AllocationType allocation\_method = PAGING;

Void remove\_process(int index);

void display\_memory() {

printf("\n--- Memory State (1 = used, 0 = free) ---\n");

for (int i = 0; i < TOTAL\_MEMORY; i++) {

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

if ((i + 1) % 64 == 0) printf("\n");

}

printf("----------------------------------------\n");

}

int find\_free\_block(int size) {

for (int i = 0; i <= TOTAL\_MEMORY - size; i++) {

int free = 1;

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

if (memory[j] == 1) {

free = 0;

break;

}

}

if (free) return i;

}

return -1;

}

void allocate\_paging(Process \*p) {

int pages\_needed = (p->memory + PAGE\_SIZE - 1) / PAGE\_SIZE;

int allocated\_pages = 0;

for (int i = 0; i < TOTAL\_MEMORY && allocated\_pages < pages\_needed; i += PAGE\_SIZE) {

int used = 0;

for (int j = i; j < i + PAGE\_SIZE; j++) {

if (memory[j] == 1) {

used = 1;

break;

}

}

if (!used) {

for (int j = i; j < i + PAGE\_SIZE; j++)

memory[j] = 1;

p->pages[allocated\_pages++] = i;

}

}

if (allocated\_pages == pages\_needed) {

p->allocated = 1;

printf("Process %s allocated using paging (%d pages).\n", p->name, pages\_needed);

} else {

printf("Not enough memory. Swapping required.\n");

p->allocated = 0;

}

}

void allocate\_segmentation(Process \*p) {

int success = 1;

for (int i = 0; i < p->num\_segments; i++) {

int block = find\_free\_block(p->segments[i]);

if (block == -1) {

success = 0;

break;

} else {

for (int j = block; j < block + p->segments[i]; j++)

memory[j] = 1;

}

}

if (success) {

p->allocated = 1;

printf("Process %s allocated using segmentation.\n", p->name);

} else {

printf("Not enough memory. Swapping required.\n");

p->allocated = 0;

}

}

void add\_process() {

if (num\_processes >= MAX\_PROCESSES) {

printf("Process limit reached.\n");

return;

}

Process \*p = &processes[num\_processes];

printf("Enter process name: ");

scanf("%s", p->name);

if (allocation\_method == PAGING) {

printf("Enter memory needed (in KB): ");

scanf("%d", &p->memory);

allocate\_paging(p);

} else {

printf("Enter number of segments: ");

scanf("%d", &p->num\_segments);

p->memory = 0;

for (int i = 0; i < p->num\_segments; i++) {

printf(" Size of segment %d: ", i + 1);

scanf("%d", &p->segments[i]);

p->memory += p->segments[i];

}

allocate\_segmentation(p);

}

if (!p->allocated) {

printf("Swapping out first process: %s\n", processes[0].name);

remove\_process(0);

// Retry allocation

if (allocation\_method == PAGING)

allocate\_paging(p);

else

allocate\_segmentation(p);

}

if (p->allocated) {

num\_processes++;

}

}

void remove\_process(int index) {

if (index < 0 || index >= num\_processes) return;

Process \*p = &processes[index];

if (allocation\_method == PAGING) {

for (int i = 0; i < 10; i++) {

int start = p->pages[i];

if (start >= 0 && start < TOTAL\_MEMORY) {

for (int j = start; j < start + PAGE\_SIZE; j++)

memory[j] = 0;

}

}

} else {

// Very basic deallocation assumption: clears all memory

for (int i = 0; i < TOTAL\_MEMORY; i++)

memory[i] = 0;

}

// Shift processes

for (int i = index; i < num\_processes - 1; i++) {

processes[i] = processes[i + 1];

}

num\_processes--;

}

void delete\_process() {

char name[20];

printf("Enter process name to delete: ");

scanf("%s", name);

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

if (strcmp(processes[i].name, name) == 0) {

remove\_process(i);

printf("Process %s removed.\n", name);

return;

}

}

printf("Process not found.\n");

}

void set\_strategy() {

int choice;

printf("Choose Allocation Strategy:\n");

printf("1. Paging\n");

printf("2. Segmentation\n");

printf("Enter choice: ");

scanf("%d", &choice);

allocation\_method = (choice == 1) ? PAGING : SEGMENTATION;

}

void menu() {

int choice;

while (1) {

printf("\n===== Memory Simulator Menu =====\n");

printf("1. Set Allocation Strategy\n");

printf("2. Add Process\n");

printf("3. Delete Process\n");

printf("4. Display Memory\n");

printf("5. Exit\n");

printf("Enter choice: ");

scanf("%d", &choice);

switch (choice) {

case 1: set\_strategy(); break;

case 2: add\_process(); break;

case 3: delete\_process(); break;

case 4: display\_memory(); break;

case 5: exit(0);

default: printf("Invalid choice.\n");

}

}

}

int main() {

menu();

return 0;

}