**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <string.h>

#define MAX\_BLOCKS 100

#define MAX\_PROCESSES 50

typedef struct {

int start;

int size;

bool free;

char pid[20];

} MemoryBlock;

typedef struct {

char pid[20];

int arrival;

int size;

bool allocated;

} Process;

MemoryBlock memory[MAX\_BLOCKS];

int blockCount = 1;

char fifoQueue[MAX\_PROCESSES][20]; // For FIFO replacement

int fifoFront = 0, fifoRear = 0;

void enqueue(const char \*pid) {

strcpy(fifoQueue[fifoRear++], pid);

}

char\* dequeue() {

return fifoQueue[fifoFront++];

}

void initializeMemory(int totalMemory) {

memory[0].start = 0;

memory[0].size = totalMemory;

memory[0].free = true;

strcpy(memory[0].pid, "");

}

void displayMemory(int totalMemory) {

int used = 0;

printf("\n🧠 Memory Layout:\n");

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

printf("[%d - %d] : %s (Size: %d)\n",

memory[i].start,

memory[i].start + memory[i].size - 1,

memory[i].free ? "Free" : memory[i].pid,

memory[i].size);

if (!memory[i].free) used += memory[i].size;

}

printf("Total: %d KB | Used: %d KB | Free: %d KB\n", totalMemory, used, totalMemory - used);

}

void mergeFreeBlocks() {

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

if (memory[i].free && memory[i + 1].free) {

memory[i].size += memory[i + 1].size;

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

memory[j] = memory[j + 1];

}

blockCount--;

i--; // recheck after merge

}

}

}

void deallocateOldest() {

if (fifoFront == fifoRear) {

printf("🚨 Error: FIFO queue empty! No processes to deallocate.\n");

return;

}

char \*oldestPid = dequeue();

printf("🔄 Replacing process %s due to lack of memory\n", oldestPid);

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

if (!memory[i].free && strcmp(memory[i].pid, oldestPid) == 0) {

memory[i].free = true;

strcpy(memory[i].pid, "");

mergeFreeBlocks();

return;

}

}

}

bool allocateMemory(Process \*p) {

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

if (memory[i].free && memory[i].size >= p->size) {

printf("✅ Allocating %s (%d KB) at address %d\n", p->pid, p->size, memory[i].start);

int remaining = memory[i].size - p->size;

memory[i].size = p->size;

memory[i].free = false;

strcpy(memory[i].pid, p->pid);

p->allocated = true;

enqueue(p->pid);

if (remaining > 0) {

for (int j = blockCount; j > i + 1; j--) {

memory[j] = memory[j - 1];

}

memory[i + 1].start = memory[i].start + p->size;

memory[i + 1].size = remaining;

memory[i + 1].free = true;

strcpy(memory[i + 1].pid, "");

blockCount++;

}

return true;

}

}

return false; // Not enough space

}

// Sorting function: prioritize smaller size if arrival time is same

void sortProcesses(Process \*processes, int n, int currentTime) {

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

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

if (!processes[j].allocated && !processes[j+1].allocated &&

processes[j].arrival <= currentTime && processes[j+1].arrival <= currentTime) {

if (processes[j].size > processes[j+1].size) {

Process temp = processes[j];

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

processes[j+1] = temp;

}

}

}

}

}

// Check for duplicate PIDs

bool isDuplicate(Process \*processes, int count, const char \*pid) {

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

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

return true;

}

}

return false;

}

int main() {

int totalMemory;

printf("Enter total memory size (KB): ");

scanf("%d", &totalMemory);

initializeMemory(totalMemory);

int n;

printf("Enter number of processes: ");

scanf("%d", &n);

Process processes[MAX\_PROCESSES];

printf("Enter process details (PID Arrival\_time Size\_in\_KB):\n");

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

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

// Check for duplicate PID

if (isDuplicate(processes, i, processes[i].pid)) {

printf("❌ Error: Duplicate Process ID '%s' entered. Process IDs must be unique!\n", processes[i].pid);

return 1; // Exit with error

}

processes[i].allocated = false;

}

int currentTime = 0;

printf("\n📦 Simulating Dynamic Partitioning with FIFO Replacement and Size-Based Selection...\n");

while (1) {

bool anyLeft = false;

sortProcesses(processes, n, currentTime); // Sort before each allocation step!

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

if (!processes[i].allocated && processes[i].arrival <= currentTime) {

// Try to allocate; if it fails, run replacement

while (!allocateMemory(&processes[i])) {

deallocateOldest();

}

displayMemory(totalMemory);

}

if (!processes[i].allocated)

anyLeft = true;

}

if (!anyLeft) break;

currentTime++;

}

printf("\n🎯 Simulation Complete!\n");

return 0;

}