**Assignment Number 06**

**Name:** Mihir Unmesh Patil

**Roll NO**: TYCOC213

**Batch:** C/C-3

**CODE:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

typedef struct MemoryBlock {

    int start\_address;

    int size;

    bool is\_free;

    struct MemoryBlock\* next;

} MemoryBlock;

MemoryBlock\* createMemoryBlock(int start\_address, int size, bool is\_free) {

    MemoryBlock\* block = (MemoryBlock\*)malloc(sizeof(MemoryBlock));

    if (block == NULL) {

        perror("Failed to allocate memory for block");

        exit(EXIT\_FAILURE);

    }

    block->start\_address = start\_address;

    block->size = size;

    block->is\_free = is\_free;

    block->next = NULL;

    return block;

}

void displayMemory(MemoryBlock\* head) {

    MemoryBlock\* current = head;

    while (current != NULL) {

        printf("Block(start=%d, size=%d, free=%s)\n", current->start\_address, current->size, current->is\_free ? "true" : "false");

        current = current->next;

    }

}

int firstFit(MemoryBlock\* head, int process\_size) {

    MemoryBlock\* current = head;

    while (current != NULL) {

        if (current->is\_free && current->size >= process\_size) {

            int start\_address = current->start\_address;

            if (current->size == process\_size) {

                current->is\_free = false;

            } else {

                MemoryBlock\* new\_block = createMemoryBlock(current->start\_address + process\_size, current->size - process\_size, true);

                new\_block->next = current->next;

                current->next = new\_block;

                current->size = process\_size;

                current->is\_free = false;

            }

            return start\_address;

        }

        current = current->next;

    }

    return -1;

}

int bestFit(MemoryBlock\* head, int process\_size) {

    MemoryBlock\* current = head;

    MemoryBlock\* best\_block = NULL;

    int min\_size = \_\_INT\_MAX\_\_;

    while (current != NULL) {

        if (current->is\_free && current->size >= process\_size && current->size < min\_size) {

            best\_block = current;

            min\_size = current->size;

        }

        current = current->next;

    }

    if (best\_block != NULL) {

        int start\_address = best\_block->start\_address;

        if (best\_block->size == process\_size) {

            best\_block->is\_free = false;

        } else {

            MemoryBlock\* new\_block = createMemoryBlock(best\_block->start\_address + process\_size, best\_block->size - process\_size, true);

            new\_block->next = best\_block->next;

            best\_block->next = new\_block;

            best\_block->size = process\_size;

            best\_block->is\_free = false;

        }

        return start\_address;

    }

    return -1;

}

int worstFit(MemoryBlock\* head, int process\_size) {

    MemoryBlock\* current = head;

    MemoryBlock\* worst\_block = NULL;

    int max\_size = 0;

    while (current != NULL) {

        if (current->is\_free && current->size >= process\_size && current->size > max\_size) {

            worst\_block = current;

            max\_size = current->size;

        }

        current = current->next;

    }

    if (worst\_block != NULL) {

        int start\_address = worst\_block->start\_address;

        if (worst\_block->size == process\_size) {

            worst\_block->is\_free = false;

        } else {

            MemoryBlock\* new\_block = createMemoryBlock(worst\_block->start\_address + process\_size, worst\_block->size - process\_size, true);

            new\_block->next = worst\_block->next;

            worst\_block->next = new\_block;

            worst\_block->size = process\_size;

            worst\_block->is\_free = false;

        }

        return start\_address;

    }

    return -1;

}

bool freeBlock(MemoryBlock\* head, int start\_address) {

    MemoryBlock\* current = head;

    while (current != NULL) {

        if (current->start\_address == start\_address) {

            current->is\_free = true;

            if (current->next != NULL && current->next->is\_free) {

                current->size += current->next->size;

                MemoryBlock\* temp = current->next;

                current->next = current->next->next;

                free(temp);

            }

            MemoryBlock\* prev = head;

            if (head != current) {

                while (prev->next != current) {

                    prev = prev->next;

                }

                if (prev->is\_free) {

                    prev->size += current->size;

                    prev->next = current->next;

                    free(current);

                    current = prev;

                }

            }

            return true;

        }

        current = current->next;

    }

    return false;

}

void deallocateMemory(MemoryBlock\* head) {

    MemoryBlock\* current = head;

    MemoryBlock\* next;

    while (current != NULL) {

        next = current->next;

        free(current);

        current = next;

    }

}

int main() {

    int memory\_size = 200;

    MemoryBlock\* memory = createMemoryBlock(0, memory\_size, true);

    printf("Initial Memory:\n");

    displayMemory(memory);

    int p1\_address = firstFit(memory, 40);

    if (p1\_address != -1) {

        printf("Allocated 40MB to P1 at address %d\n", p1\_address);

    } else {

        printf("Failed to allocate 40MB to P1\n");

    }

    printf("Memory after P1 allocation:\n");

    displayMemory(memory);

    printf("\n");

    int p2\_address = bestFit(memory, 80);

    if (p2\_address != -1) {

        printf("Allocated 80MB to P2 at address %d\n", p2\_address);

    } else {

        printf("Failed to allocate 80MB to P2\n");

    }

    printf("Memory after P2 allocation:\n");

    displayMemory(memory);

    printf("\n");

    int p3\_address = worstFit(memory, 20);

    if (p3\_address != -1) {

        printf("Allocated 20MB to P3 at address %d\n", p3\_address);

    } else {

        printf("Failed to allocate 20MB to P3\n");

    }

    printf("Memory after P3 allocation:\n");

    displayMemory(memory);

    printf("\n");

    freeBlock(memory, 0);

    printf("After freeing P1:\n");

    displayMemory(memory);

    deallocateMemory(memory);

    return 0;

}

**OUTPUT:**

