



ANSWER->

//22BCE0682

CODE:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define T\_COUNT 4

#define SIZE 1000000

typedef struct {

    int\* a;

    int start\_idx;

    int end\_idx;

    long long total;

} *ThreadData*;

void\* calc\_partial\_sum(void\* *arg*) {

*ThreadData*\* td = (*ThreadData*\*)*arg*;

    long long s = 0;

    for (int i = td->start\_idx; i < td->end\_idx; i++) {

        s += td->a[i];

    }

    td->total = s;

    pthread\_exit(NULL);

}

int main() {

    int\* a = (int\*)malloc(SIZE \* sizeof(int));

*pthread\_t* t[T\_COUNT];

*ThreadData* td[T\_COUNT];

    long long total\_sum = 0;

    for (int i = 0; i < SIZE; i++) {

        a[i] = rand() % 100;

    }

    int chunk = SIZE / T\_COUNT;

    for (int i = 0; i < T\_COUNT; i++) {

        td[i].a = a;

        td[i].start\_idx = i \* chunk;

        td[i].end\_idx = (i == T\_COUNT - 1) ? SIZE : (i + 1) \* chunk;

        td[i].total = 0;

        pthread\_create(&t[i], NULL, calc\_partial\_sum, (void\*)&td[i]);

    }

    for (int i = 0; i < T\_COUNT; i++) {

        pthread\_join(t[i], NULL);

        total\_sum += td[i].total;

    }

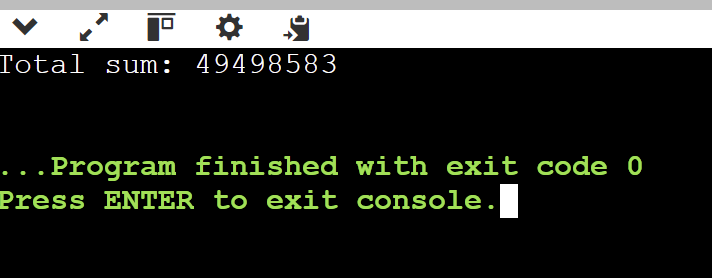
    printf("Total sum: %lld\n", total\_sum);

    free(a);

    return 0;

}

OUTPUT:





ANSWER->

//22BCE0682

CODE:

#include <stdio.h>

#include <stdlib.h>

#define MAX\_MEMORY\_BLOCKS 10

#define MAX\_TASKS 10

struct *Block* {

    int bidx;

    int bcap;

    int isass;

    int tidx;

};

struct *Task* {

    int tidx;

    int tsize;

};

void initializeBlocks(struct *Block* *blocks*[], int *capacities*[], int *block\_count*);

void firstFitAllocation(struct *Block* *blocks*[], int *block\_count*, struct *Task* *tasks*[], int *task\_count*);

void bestFitAllocation(struct *Block* *blocks*[], int *block\_count*, struct *Task* *tasks*[], int *task\_count*);

void worstFitAllocation(struct *Block* *blocks*[], int *block\_count*, struct *Task* *tasks*[], int *task\_count*);

void showMemoryStatus(struct *Block* *blocks*[], int *block\_count*, const char\* *algorithm\_name*);

void copyMemoryState(struct *Block* *target*[], struct *Block* *source*[], int *block\_count*);

int main() {

    struct *Block* memory[MAX\_MEMORY\_BLOCKS];

    struct *Block* temp\_memory[MAX\_MEMORY\_BLOCKS];

    struct *Task* tasks[MAX\_TASKS];

    int block\_count, task\_count, i;

    int block\_capacities[MAX\_MEMORY\_BLOCKS];

    printf("Enter the number of memory blocks: ");

    scanf("%d", &block\_count);

    printf("\nEnter the capacity of each memory block:\n");

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

        printf("Block %d: ", i + 1);

        scanf("%d", &block\_capacities[i]);

    }

    printf("\nEnter the number of tasks: ");

    scanf("%d", &task\_count);

    printf("\nEnter the size of each task:\n");

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

        printf("Task %d: ", i + 1);

        scanf("%d", &tasks[i].tsize);

        tasks[i].tidx = i + 1;

    }

    initializeBlocks(memory, block\_capacities, block\_count);

    printf("\n\n=== Memory Allocation Results ===\n");

    printf("\n1. First Fit Allocation\n");

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

    copyMemoryState(temp\_memory, memory, block\_count);

    firstFitAllocation(temp\_memory, block\_count, tasks, task\_count);

    showMemoryStatus(temp\_memory, block\_count, "First Fit");

    printf("\n2. Best Fit Allocation\n");

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

    copyMemoryState(temp\_memory, memory, block\_count);

    bestFitAllocation(temp\_memory, block\_count, tasks, task\_count);

    showMemoryStatus(temp\_memory, block\_count, "Best Fit");

    printf("\n3. Worst Fit Allocation\n");

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

    copyMemoryState(temp\_memory, memory, block\_count);

    worstFitAllocation(temp\_memory, block\_count, tasks, task\_count);

    showMemoryStatus(temp\_memory, block\_count, "Worst Fit");

    return 0;

}

void initializeBlocks(struct *Block* *blocks*[], int *capacities*[], int *block\_count*) {

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

*blocks*[i].bidx = i + 1;

*blocks*[i].bcap = *capacities*[i];

*blocks*[i].isass = 0;

*blocks*[i].tidx = 0;

    }

}

void copyMemoryState(struct *Block* *target*[], struct *Block* *source*[], int *block\_count*) {

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

*target*[i] = *source*[i];

    }

}

void firstFitAllocation(struct *Block* *blocks*[], int *block\_count*, struct *Task* *tasks*[], int *task\_count*) {

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

        int allocated = 0;

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

            if(!*blocks*[j].isass && *blocks*[j].bcap >= *tasks*[i].tsize) {

*blocks*[j].isass = 1;

*blocks*[j].tidx = *tasks*[i].tidx;

                printf("Task %d (%d KB) -> Allocated to Block %d (%d KB)\n",

*tasks*[i].tidx, *tasks*[i].tsize, *blocks*[j].bidx, *blocks*[j].bcap);

                allocated = 1;

                break;

            }

        }

        if(!allocated) {

            printf("Task %d (%d KB) -> Cannot be allocated\n",

*tasks*[i].tidx, *tasks*[i].tsize);

        }

    }

}

void bestFitAllocation(struct *Block* *blocks*[], int *block\_count*, struct *Task* *tasks*[], int *task\_count*) {

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

        int best\_block\_idx = -1;

        int smallest\_diff = 999999; // Use a large initial value to find the minimum difference

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

            if(!*blocks*[j].isass && *blocks*[j].bcap >= *tasks*[i].tsize) {

                int diff = *blocks*[j].bcap - *tasks*[i].tsize;

                if(diff < smallest\_diff) {

                    smallest\_diff = diff;

                    best\_block\_idx = j;

                }

            }

        }

        if(best\_block\_idx != -1) {

*blocks*[best\_block\_idx].isass = 1;

*blocks*[best\_block\_idx].tidx = *tasks*[i].tidx;

            printf("Task %d (%d KB) -> Allocated to Block %d (%d KB)\n",

*tasks*[i].tidx, *tasks*[i].tsize, *blocks*[best\_block\_idx].bidx, *blocks*[best\_block\_idx].bcap);

        } else {

            printf("Task %d (%d KB) -> Cannot be allocated\n",

*tasks*[i].tidx, *tasks*[i].tsize);

        }

    }

}

void worstFitAllocation(struct *Block* *blocks*[], int *block\_count*, struct *Task* *tasks*[], int *task\_count*) {

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

        int worst\_block\_idx = -1;

        int largest\_diff = -1; // Start with a negative value to find the largest available difference

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

            if(!*blocks*[j].isass && *blocks*[j].bcap >= *tasks*[i].tsize) {

                int diff = *blocks*[j].bcap - *tasks*[i].tsize;

                if(diff > largest\_diff) {

                    largest\_diff = diff;

                    worst\_block\_idx = j;

                }

            }

        }

        if(worst\_block\_idx != -1) {

*blocks*[worst\_block\_idx].isass = 1;

*blocks*[worst\_block\_idx].tidx = *tasks*[i].tidx;

            printf("Task %d (%d KB) -> Allocated to Block %d (%d KB)\n",

*tasks*[i].tidx, *tasks*[i].tsize, *blocks*[worst\_block\_idx].bidx, *blocks*[worst\_block\_idx].bcap);

        } else {

            printf("Task %d (%d KB) -> Cannot be allocated\n",

*tasks*[i].tidx, *tasks*[i].tsize);

        }

    }

}

void showMemoryStatus(struct *Block* *blocks*[], int *block\_count*, const char\* *algorithm\_name*) {

    printf("\nFinal Memory State (%s):\n", *algorithm\_name*);

    printf("Block\tCapacity\tStatus\t\tTask\n");

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

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

        printf("%d\t%d\t\t%s\t\t",

*blocks*[i].bidx,

*blocks*[i].bcap,

*blocks*[i].isass ? "Assigned" : "Free");

        if(*blocks*[i].isass)

            printf("T%d\n", *blocks*[i].tidx);

        else

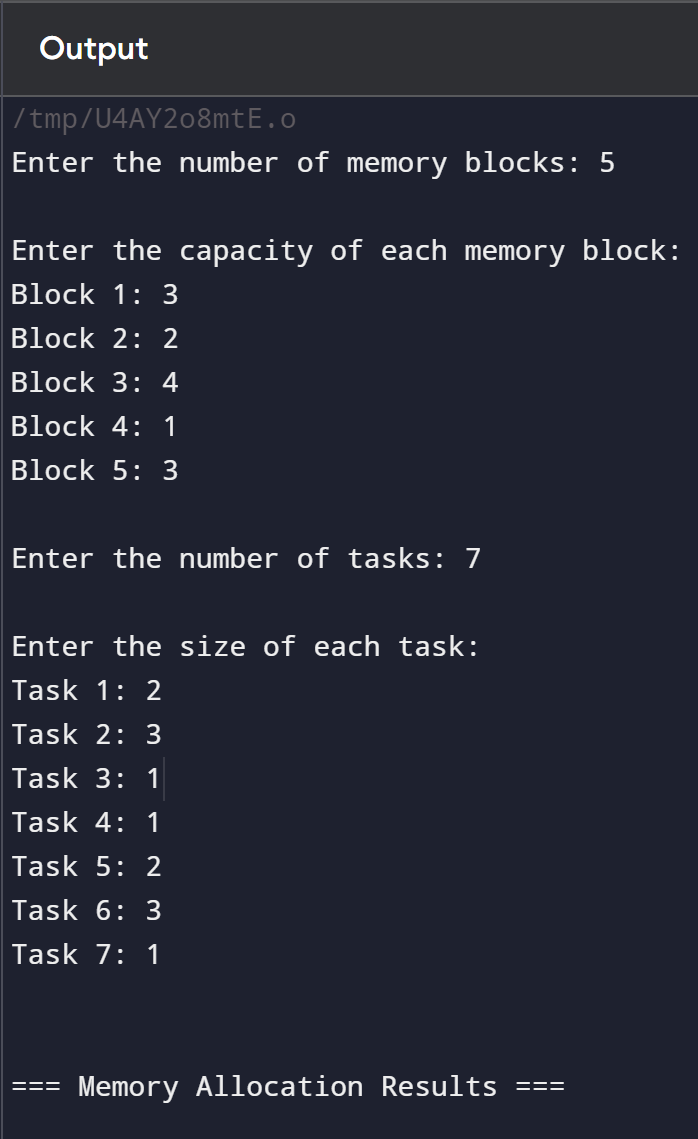
            printf("None\n");

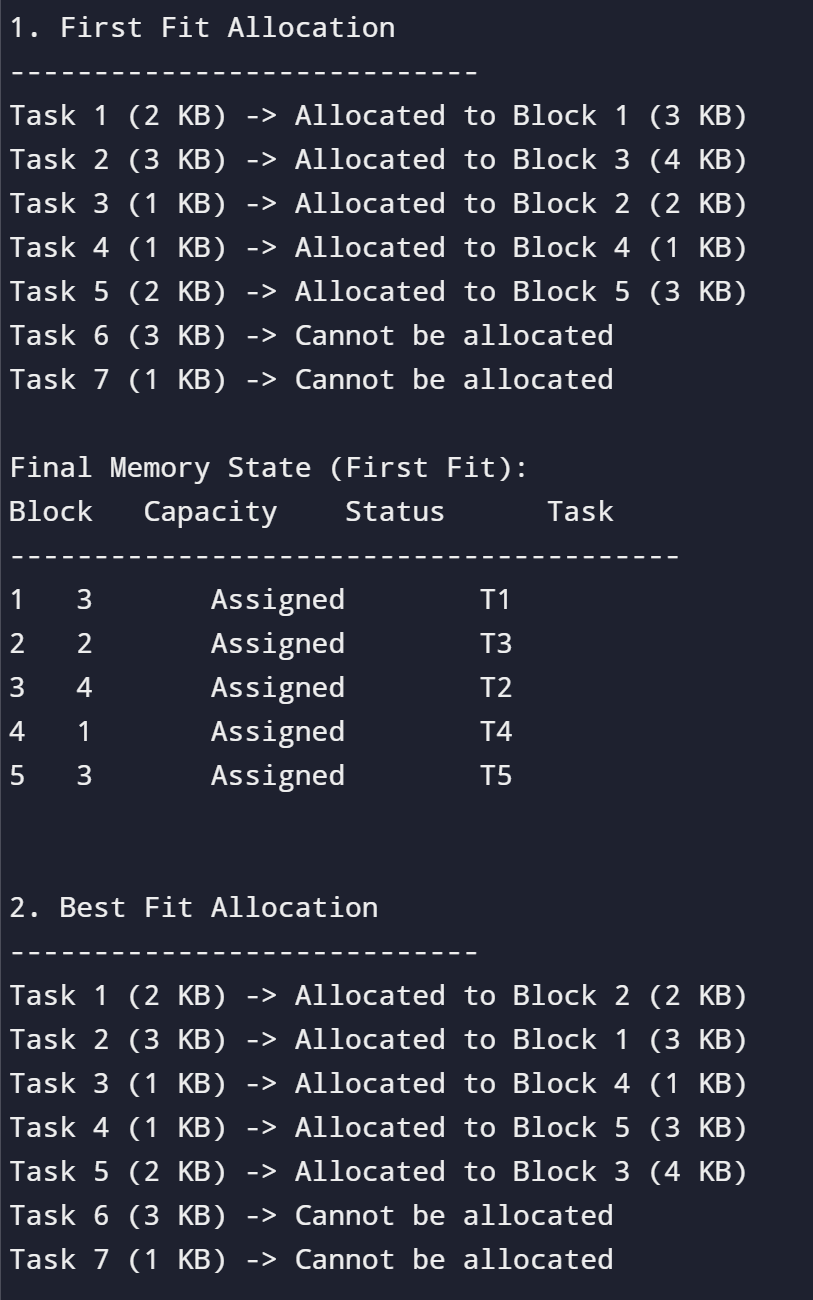
    }

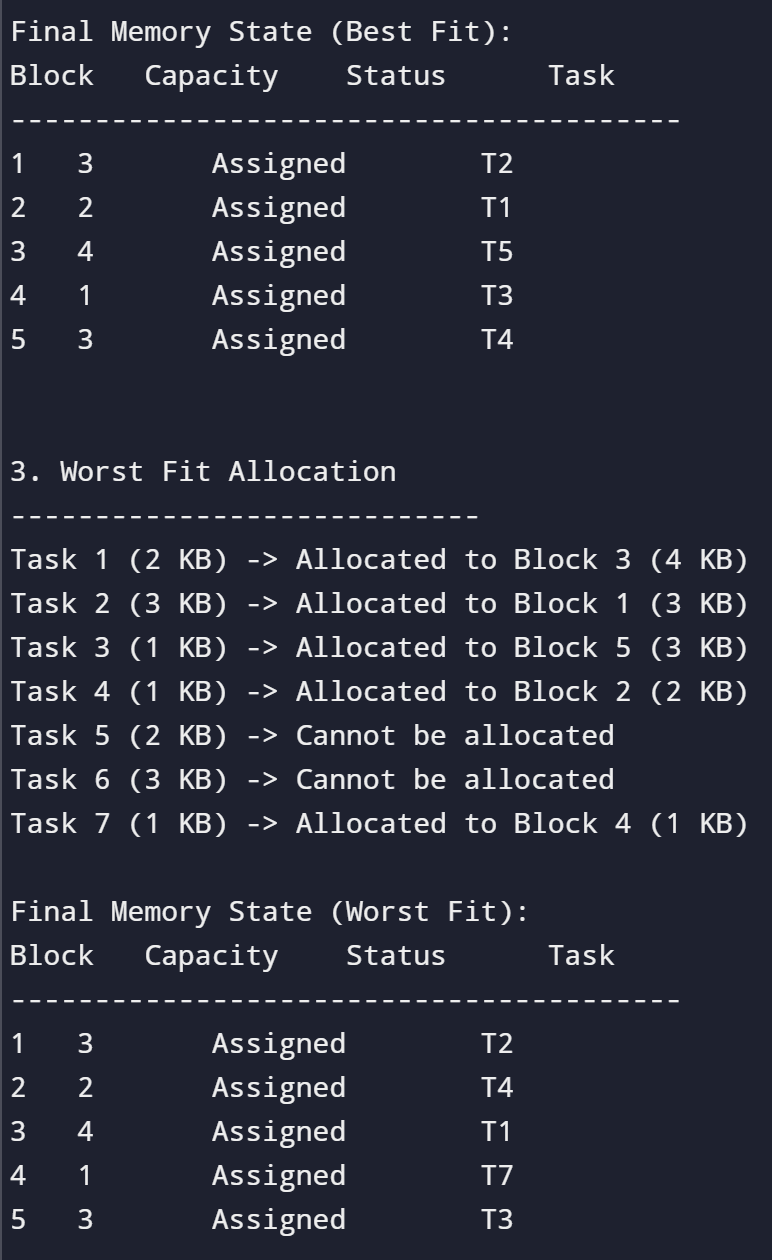
    printf("\n");

}

OUTPUT:









ANSWER->

//22BCE0682

CODE:

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

int srch(int *key*, int *fi*[], int *fo*) {

    for (int i = 0; i < fo; i++)

        if (fi[i] == key)

            return 1;

    return 0;

}

int Flru(int *time*[], int *n*) {

    int minimum = time[0], pos = 0;

    for (int i = 1; i < n; i++) {

        if (time[i] < minimum) {

            minimum = time[i];

            pos = i;

        }

    }

    return pos;

}

void FIFO(int *pages*[], int *n*, int *frames*) {

    int fi[frames];

    for (int i = 0; i < frames; i++) fi[i] = -1;

    int fo = 0;

    int pgfaults = 0;

    int current = 0;

    printf("\nFIFO Page Replacement Algorithm\n");

    for (int i = 0; i < n; i++) {

        printf("\nFor page %d: ", pages[i]);

        if (!srch(pages[i], fi, fo)) {

            if (fo < frames) {

                fi[fo] = pages[i];

                fo++;

            } else {

                fi[current] = pages[i];

                current = (current + 1) % frames;

            }

            pgfaults++;

        }

        for (int j = 0; j < fo; j++)

            printf("%d ", fi[j]);

    }

    printf("\nTotal Page Faults (FIFO): %d\n", pgfaults);

}

void LRU(int *pages*[], int *n*, int *frames*) {

    int fi[frames], time[frames];

    for (int i = 0; i < frames; i++) fi[i] = -1;

    int fo = 0;

    int pgfaults = 0;

    printf("\nLRU Page Replacement Algorithm\n");

    for (int i = 0; i < n; i++) {

        printf("\nFor page %d: ", pages[i]);

        if (!srch(pages[i], fi, fo)) {

            if (fo < frames) {

                fi[fo] = pages[i];

                time[fo] = i;

                fo++;

            } else {

                int pos = Flru(time, frames);

                fi[pos] = pages[i];

                time[pos] = i;

            }

            pgfaults++;

        } else {

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

                if (fi[j] == pages[i]) {

                    time[j] = i;

                    break; // Fixed the typo here

                }

            }

        }

        for (int j = 0; j < fo; j++)

            printf("%d ", fi[j]);

    }

    printf("\nTotal Page Faults (LRU): %d\n", pgfaults);

}

// Updated OPTimal Algorithm

int fOPT(int *pages*[], int *fi*[], int *n*, int *index*) {

    int res = -1, far = index;

    for (int i = 0; i < n; i++) {

        int j;

        for (j = index; j < n; j++) {

            if (fi[i] == pages[j]) {

                if (j > far) {

                    far = j;

                    res = i;

                }

                break; // Fixed the typo here

            }

        }

        if (j == n)

            return i;

    }

    return res; // Simply return res

}

void OPT(int *pages*[], int *n*, int *frames*) {

    int fi[frames];

    for (int i = 0; i < frames; i++) fi[i] = -1;

    int fo = 0;

    int pgfaults = 0;

    printf("\nOPT Page Replacement Algorithm\n");

    for (int i = 0; i < n; i++) {

        printf("\nFor page %d: ", pages[i]);

        if (!srch(pages[i], fi, fo)) {

            if (fo < frames) {

                fi[fo] = pages[i];

                fo++;

            } else {

                int pos = fOPT(pages, fi, n, i + 1);

                fi[pos] = pages[i];

            }

            pgfaults++;

        }

        for (int j = 0; j < fo; j++)

            printf("%d ", fi[j]);

    }

    printf("\nTotal Page Faults (Optimal): %d\n", pgfaults);

}

int main() {

    int frames, n;

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

    scanf("%d", &frames);

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

    scanf("%d", &n);

    int pages[n];

    printf("Enter the page reference string: ");

    for (int i = 0; i < n; i++)

        scanf("%d", &pages[i]);

    FIFO(pages, n, frames);

    LRU(pages, n, frames);

    OPT(pages, n, frames);

    return 0;

}

OUTPUT:

