VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

School of Computer Science and Engineering

Fall Semester 2024-25

Digital Assessment 3

SLOT: L25+L26 & L47+L48

Programme Name & Branch: B. Tech CSE

Course Name & Code: BCSE303P Operating Systems Lab

1) Write a 'C' program to implement Parallel Thread management using Pthreads library. Implement a data parallelism using multi-threading.

ANSWER->

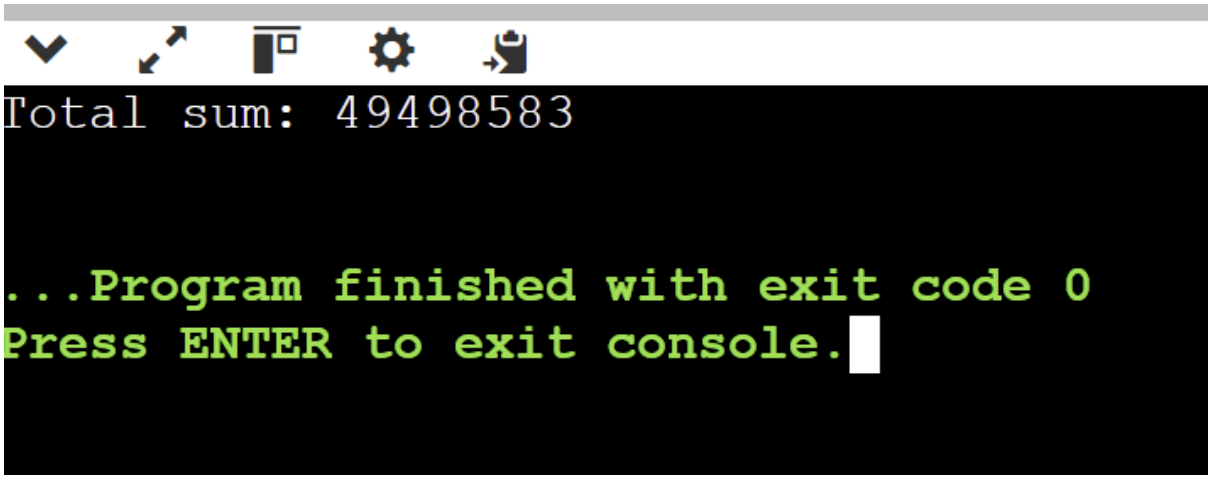
//22BCE0682

CODE:

```c
#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:



2) Dynamic memory allocation algorithms - First-fit, Best-fit, Worst-fit algorithms.

ANSWER->

//22BCE0682

CODE:

```c
#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);
```

```c
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:

Output

```
/tmp/U4AY2o8mtE.o
Enter the number of memory blocks: 5

Enter the capacity of each memory block:
Block 1: 3
Block 2: 2
Block 3: 4
Block 4: 1
Block 5: 3

Enter the number of tasks: 7

Enter the size of each task:
Task 1: 2
Task 2: 3
Task 3: 1
Task 4: 1
Task 5: 2
Task 6: 3
Task 7: 1


=== Memory Allocation Results ===
```

```
1. First Fit Allocation
----------------------------
Task 1 (2 KB) -> Allocated to Block 1 (3 KB)
Task 2 (3 KB) -> Allocated to Block 3 (4 KB)
Task 3 (1 KB) -> Allocated to Block 2 (2 KB)
Task 4 (1 KB) -> Allocated to Block 4 (1 KB)
Task 5 (2 KB) -> Allocated to Block 5 (3 KB)
Task 6 (3 KB) -> Cannot be allocated
Task 7 (1 KB) -> Cannot be allocated

Final Memory State (First Fit):
Block   Capacity    Status      Task
---------------------------------------
1   3       Assigned        T1
2   2       Assigned        T3
3   4       Assigned        T2
4   1       Assigned        T4
5   3       Assigned        T5


2. Best Fit Allocation
----------------------------
Task 1 (2 KB) -> Allocated to Block 2 (2 KB)
Task 2 (3 KB) -> Allocated to Block 1 (3 KB)
Task 3 (1 KB) -> Allocated to Block 4 (1 KB)
Task 4 (1 KB) -> Allocated to Block 5 (3 KB)
Task 5 (2 KB) -> Allocated to Block 3 (4 KB)
Task 6 (3 KB) -> Cannot be allocated
Task 7 (1 KB) -> Cannot be allocated
```

```
Final Memory State (Best Fit):
Block   Capacity    Status      Task
----------------------------------------
1   3       Assigned        T2
2   2       Assigned        T1
3   4       Assigned        T5
4   1       Assigned        T3
5   3       Assigned        T4


3. Worst Fit Allocation
---------------------------
Task 1 (2 KB) -> Allocated to Block 3 (4 KB)
Task 2 (3 KB) -> Allocated to Block 1 (3 KB)
Task 3 (1 KB) -> Allocated to Block 5 (3 KB)
Task 4 (1 KB) -> Allocated to Block 2 (2 KB)
Task 5 (2 KB) -> Cannot be allocated
Task 6 (3 KB) -> Cannot be allocated
Task 7 (1 KB) -> Allocated to Block 4 (1 KB)

Final Memory State (Worst Fit):
Block   Capacity    Status      Task
----------------------------------------
1   3       Assigned        T2
2   2       Assigned        T4
3   4       Assigned        T1
4   1       Assigned        T7
5   3       Assigned        T3
```

3) Write a 'C' program to implement Page Replacement Algorithms FIFO, LRU and Optimal.

ANSWER->

//22BCE0682

CODE:

```c
#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
```

```c
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:

Output

```
/tmp/e8VesCRyn1.o
Enter the number of frames: 5
Enter the number of pages: 12
Enter the page reference string: 5
2
1
2
4
2
1
3
3
4
2
1

FIFO Page Replacement Algorithm

For page 5: 5
For page 2: 5 2
For page 1: 5 2 1
For page 2: 5 2 1
For page 4: 5 2 1 4
For page 2: 5 2 1 4
For page 1: 5 2 1 4
For page 3: 5 2 1 4 3
For page 3: 5 2 1 4 3
For page 4: 5 2 1 4 3
For page 2: 5 2 1 4 3
For page 1: 5 2 1 4 3
Total Page Faults (FIFO): 5
```

```
LRU Page Replacement Algorithm

For page 5: 5
For page 2: 5 2
For page 1: 5 2 1
For page 2: 5 2 1
For page 4: 5 2 1 4
For page 2: 5 2 1 4
For page 1: 5 2 1 4
For page 3: 5 2 1 4 3
For page 3: 5 2 1 4 3
For page 4: 5 2 1 4 3
For page 2: 5 2 1 4 3
For page 1: 5 2 1 4 3
Total Page Faults (LRU): 5

OPT Page Replacement Algorithm

For page 5: 5
For page 2: 5 2
For page 1: 5 2 1
For page 2: 5 2 1
For page 4: 5 2 1 4
For page 2: 5 2 1 4
For page 1: 5 2 1 4
For page 3: 5 2 1 4 3
For page 3: 5 2 1 4 3
For page 4: 5 2 1 4 3
For page 2: 5 2 1 4 3
For page 1: 5 2 1 4 3
Total Page Faults (Optimal): 5
```