SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
COLLEGE OF ENGINEERING AND TECHNOLOGY
DEPARTMENT OF NETWORKING AND COMMUNICATIONS
21CSC202J-Operating Systems , Mini-Project Presentation

Timer Based Task Scheduler

Aman Anand (RA2211003010130)
Aadi Tiwari (RA2211003010127)
Krishna Mahajan (RA2211003010103)

Problem Statement

The problem is to develop a timer-based task scheduler for an operating system that efficiently manages resource allocation, real-time processing, task prioritization, preemption, synchronization, energy efficiency, and interrupt handling, to enhance system performance and responsiveness.

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

Table of contents

Resource Allocation:

This objective entails creating a task scheduler capable of efficiently distributing system resources, such as CPU time, memory, and I/O resources, among various concurrent tasks. The scheduler should aim to optimize resource usage, prevent resource contention, and allocate resources based on the priority and requirements of each task.

Real-Time Processing:

Real-time processing involves ensuring that critical tasks with strict timing constraints are executed promptly. The scheduler should be designed to provide guarantees that time-critical operations will be performed within specified deadlines, making it suitable for applications like autonomous vehicles, medical devices, and industrial control systems.

Task Prioritization

Task prioritization is about implementing a dynamic priority management system. This system adjusts the priorities of tasks based on their characteristics and resource needs. High-priority tasks should receive preferential treatment to ensure that important operations are not delayed, while maintaining fairness among tasks.

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

Task Preemption

Preemption refers to the ability of the scheduler to interrupt and temporarily suspend lower-priority tasks to allow higher-priority tasks to execute. This feature ensures that no task monopolizes the CPU, leading to more responsive and balanced task execution.

Synchronization:

Task synchronization is about providing mechanisms for tasks to coordinate their actions. This is crucial for tasks that need to work together, share resources, or communicate with each other without conflicts or data corruption.

Interrupt Handling:

The scheduler should efficiently manage hardware interrupts and timeouts. It should respond to external events, such as I/O operations or device requests, in a timely manner without compromising the performance of other tasks. Effective interrupt handling is essential for the proper functioning of a modern operating system.

Benefits of Timer-Based Task Scheduling

By adopting a Timer-Based Task Scheduler, operating systems can achieve several advantages. These include improved system responsiveness, enhanced task prioritization, reduced overhead, and better resource utilization. This approach enables efficient multitasking and ensures that critical tasks are executed in a timely manner.

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be **University** *u/s 3 of UGC Act, 1956*

Timer-Based Task Scheduling Algorithm

The Timer-Based Task Scheduling Algorithm utilizes timers to allocate CPU time to different tasks. It employs priority queues to manage the task execution order. This algorithm ensures that tasks with higher priorities are executed first, while lower priority tasks are scheduled accordingly. Additionally, it dynamically adjusts task priorities based on predefined criteria.

Challenges and Limitations

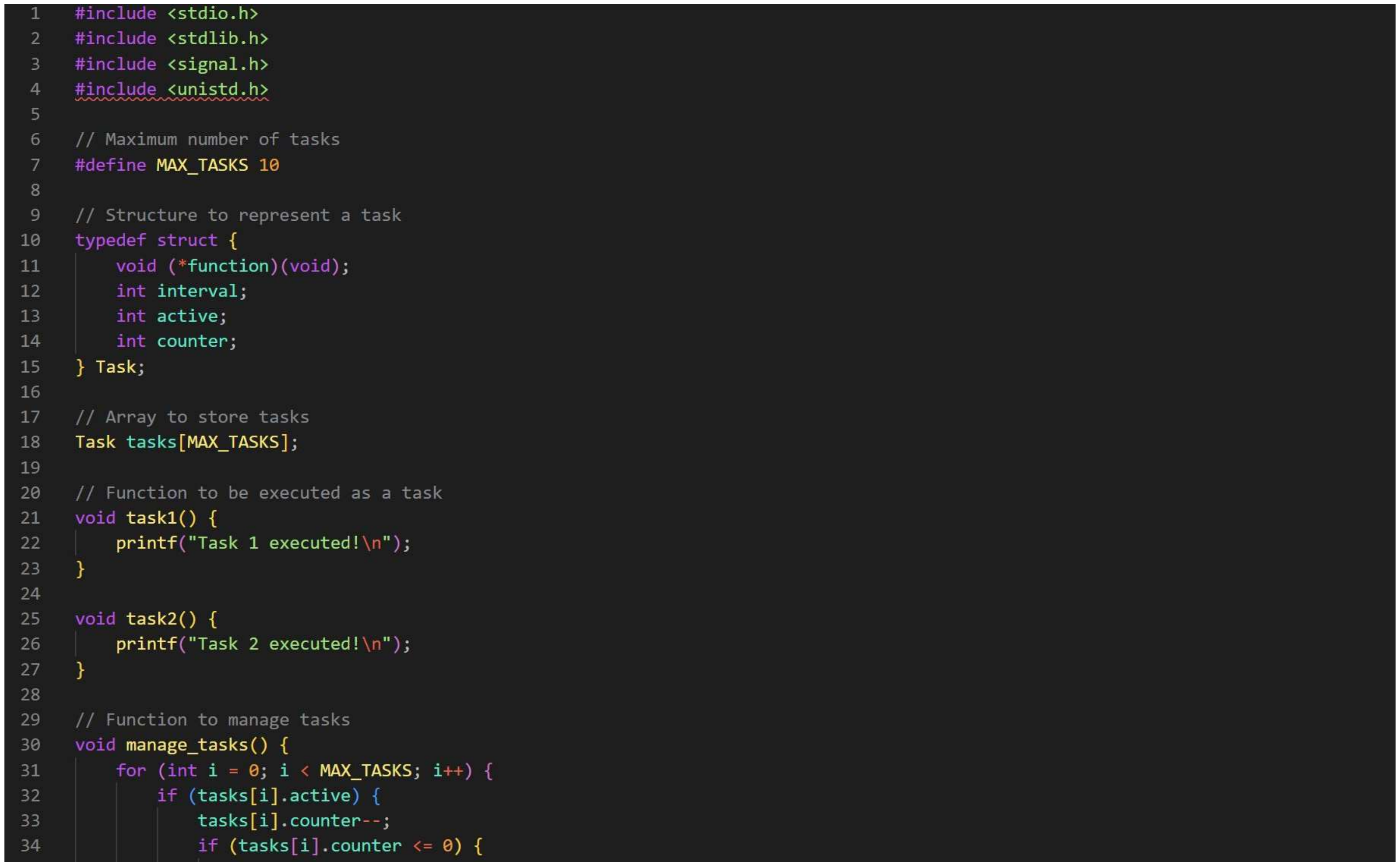
Although a Timer-Based Task Scheduler offers numerous benefits, it also presents some challenges and limitations. These include increased complexity in implementation, potential for starvation of low-priority tasks, and difficulties in handling dynamic task behavior. Understanding these limitations is crucial for successful adoption of this scheduling approach.

Use Cases and Applications

The Timer-Based Task Scheduler finds applications in various domains. It is particularly beneficial in real-time systems, multimedia processing, and embedded systems. Industries such as aerospace, automotive, and telecommunications can leverage this scheduling approach to ensure timely task execution and optimal system performance.

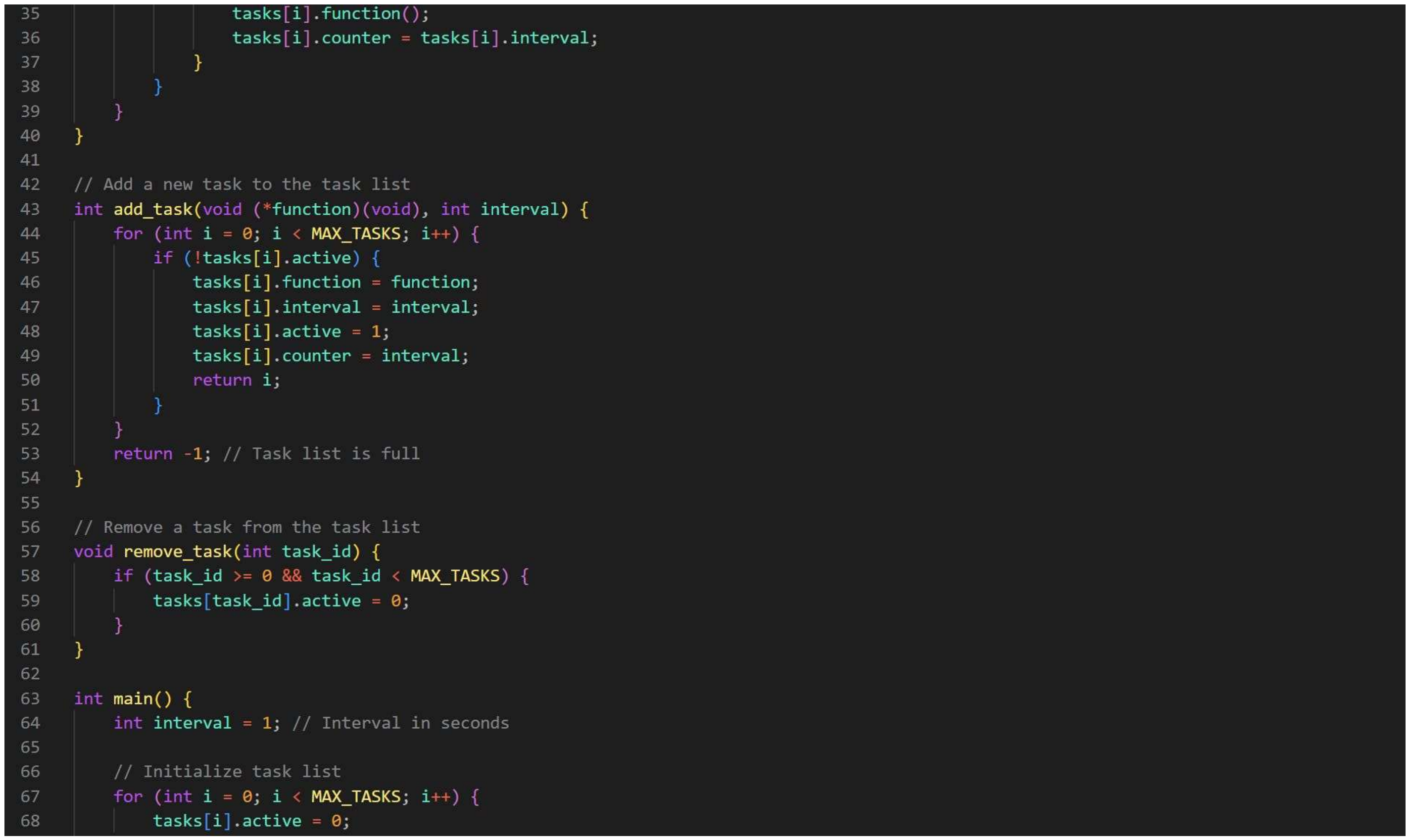
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

CODE SNIPPET



```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

// Maximum number of tasks
#define MAX_TASKS 10

// Structure to represent a task
typedef struct {
    void (*function)(void);
    int interval;
    int active;
    int counter;
} Task;

// Array to store tasks
Task tasks[MAX_TASKS];

// Function to be executed as a task
void task1() {
    printf("Task 1 executed!\n");
}

void task2() {
    printf("Task 2 executed!\n");
}

// Function to manage tasks
void manage_tasks() {
    for (int i = 0; i < MAX_TASKS; i++) {
        if (tasks[i].active) {
            tasks[i].counter--;
            if (tasks[i].counter <= 0) {
```

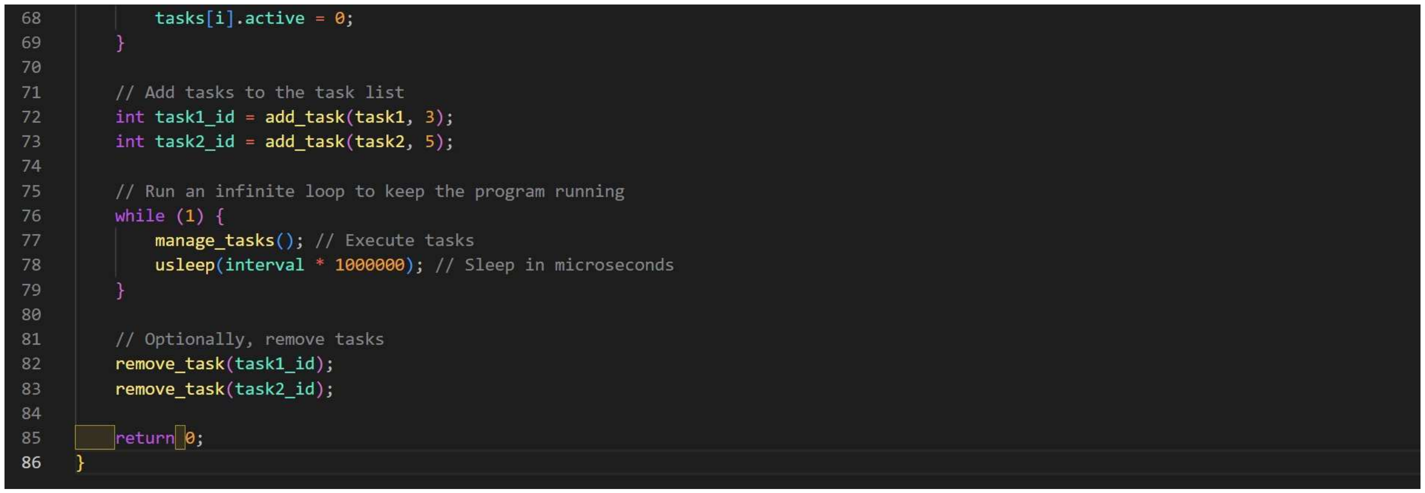
CODE SNIPPET



```c
                tasks[i].function();
                tasks[i].counter = tasks[i].interval;
            }
        }
    }
}

// Add a new task to the task list
int add_task(void (*function)(void), int interval) {
    for (int i = 0; i < MAX_TASKS; i++) {
        if (!tasks[i].active) {
            tasks[i].function = function;
            tasks[i].interval = interval;
            tasks[i].active = 1;
            tasks[i].counter = interval;
            return i;
        }
    }
    return -1; // Task list is full
}

// Remove a task from the task list
void remove_task(int task_id) {
    if (task_id >= 0 && task_id < MAX_TASKS) {
        tasks[task_id].active = 0;
    }
}

int main() {
    int interval = 1; // Interval in seconds

    // Initialize task list
    for (int i = 0; i < MAX_TASKS; i++) {
        tasks[i].active = 0;
```

INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

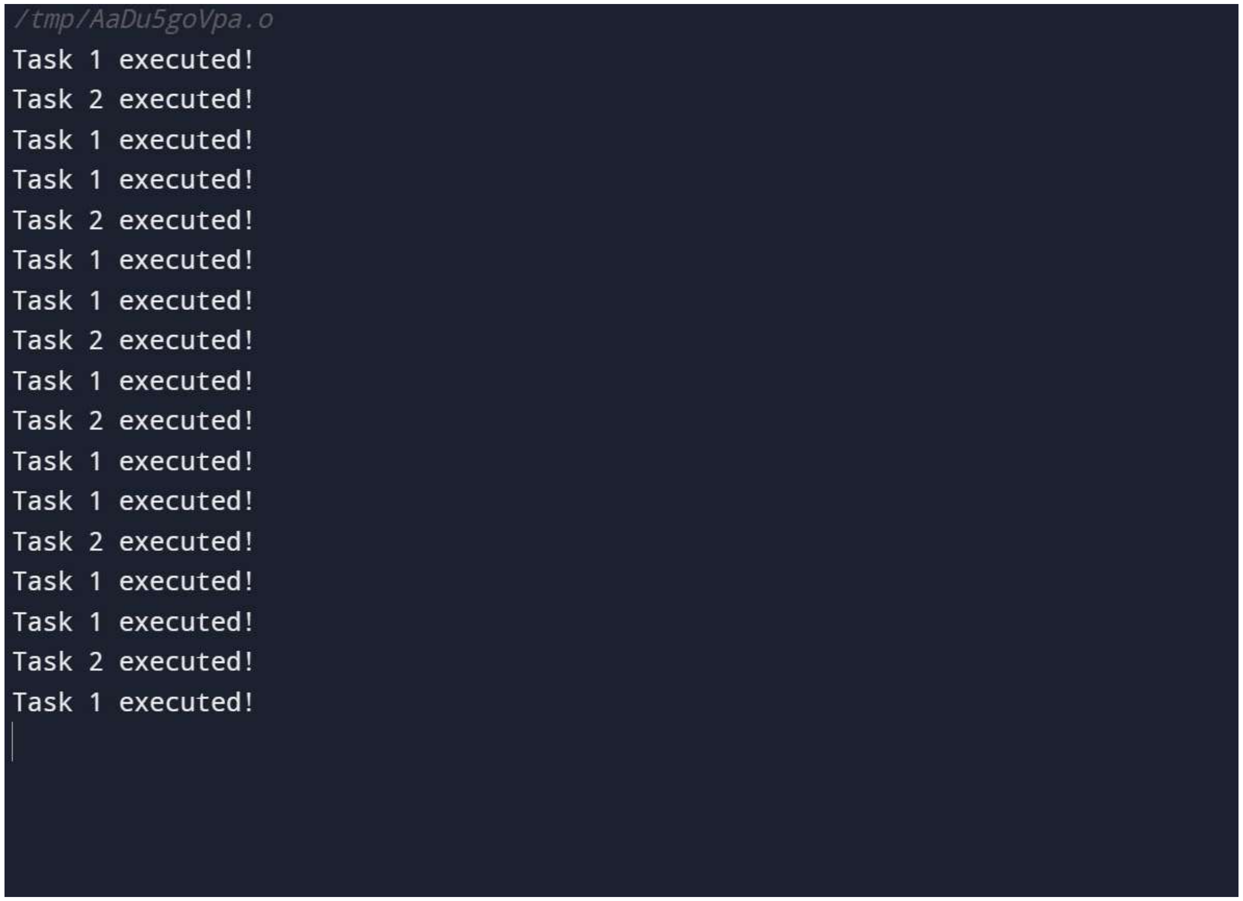
CODE SNIPPET



```c
            tasks[i].active = 0;
        }

        // Add tasks to the task list
        int task1_id = add_task(task1, 3);
        int task2_id = add_task(task2, 5);

        // Run an infinite loop to keep the program running
        while (1) {
            manage_tasks(); // Execute tasks
            usleep(interval * 1000000); // Sleep in microseconds
        }

        // Optionally, remove tasks
        remove_task(task1_id);
        remove_task(task2_id);

        return 0;
    }
```

INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

OUTPUT



```
/tmp/AaDu5goVpa.o
Task 1 executed!
Task 2 executed!
Task 1 executed!
Task 1 executed!
Task 2 executed!
Task 1 executed!
Task 1 executed!
Task 2 executed!
Task 1 executed!
Task 2 executed!
Task 1 executed!
Task 1 executed!
Task 2 executed!
Task 1 executed!
Task 1 executed!
Task 2 executed!
Task 1 executed!
```

CONCLUSION

In conclusion, a Timer-Based Task Scheduler offers significant advantages in optimizing task management for operating systems. By leveraging timers and dynamic task prioritization, this approach improves system responsiveness, task prioritization, and resource utilization. Despite its challenges and limitations, it proves valuable in time-sensitive environments and holds promise for future developments.