ELL783 Assignment2_hard

Ananya Sharma 2023EET2186

February 2024

1 Introduction

We are given a set of **n periodic tasks**, and the model assumes that each **task i** has a **period Pi** and an **execution time Ti**. The period Pi is the time interval between successive releases of the task. We assume that the relative deadline of a task i is represented by Di . **Relative deadline (Di)** is the time interval between the task's release time and its deadline. Thes set of n preemptable tasks T, are such that the tasks do not share resources, and no precedence order exists among the tasks.

We have developed a CPU scheduler for the **Liu and Layland Periodic Task Model.** The scheduler will be a Rate-Monotonic Scheduler (RMS) and a Deadline-Monotonic Scheduler(DMS). RMS and DMS are static priority scheduler, in which the priorities are assigned based on the period of the job: the shorter the period, the higher the priority. This scheduler is preemptive, which means that a task with a higher priority will always preempt a task with a lower priority until its processing time has been used. Our scheduler relies only on the Linux Scheduler to perform context switches; therefore, we use only the Linux Scheduler API.

2 Implementation of deadline monotonic (DM) and rate monotonic (RM) scheduling algorithms using kernel API of the Linux CPU Scheduler

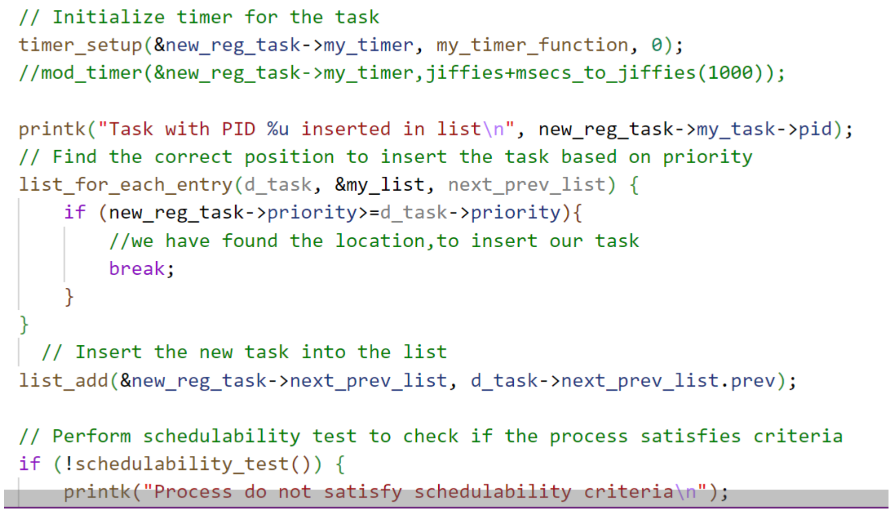
2.1 System Calls

- int **sys_register_dm(pid, period, deadline, exec_time) OR int sys_register_rm(pid, period, deadline, exec_time):**

  In RMS Scheduing period is equal to deadline. So we have created a single function for both these system calls. It is responsible for registering new tasks which satisfy the schedulability criteria.It allocates memory for the new task and associates it with the specified PID, initializing parameters such as period, deadline, and execution time. Additionally, it **initializes a timer** for the task and **inserts it into the list of tasks, ensuring correct positioning based on priority.**, as shown in figure **??**

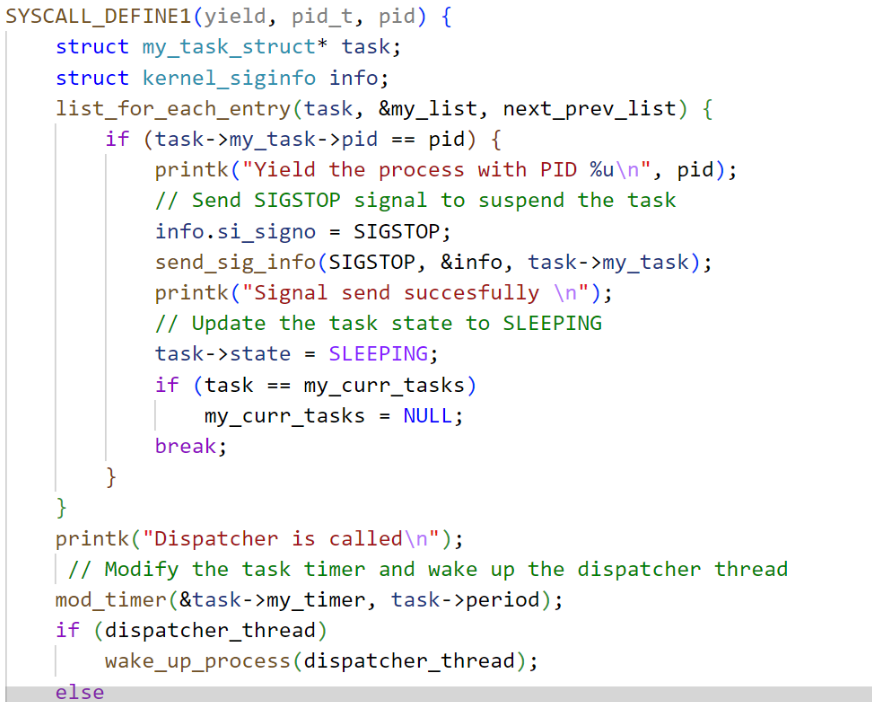
- **int sys_yield(pid)**

  sys_yield system call notifies the kernel that a process has finished executing its period, the process must send a yield message using the sys_yield system call, and the application will block it until the next period. It sends a **SIGSTOP signal to suspend the execution of the task** using the **send_sig_info** function, effectively pausing its execution. Upon successfully sending the signal, the **task's state is updated to SLEEPING** to reflect its suspended state. Following the suspension of the task, the system call proceeds to **modify the task's timer, adjusting it to the task's specified period.** This action **ensures that the task will be scheduled for execution again after the specified period elapses**. Additionally, the system call **invokes the dispatcher, signaling its execution by waking up the dispatcher thread** using the **wake_up_process** function. This step ensures that the dispatcher can reschedule tasks, as shown in figure **??**



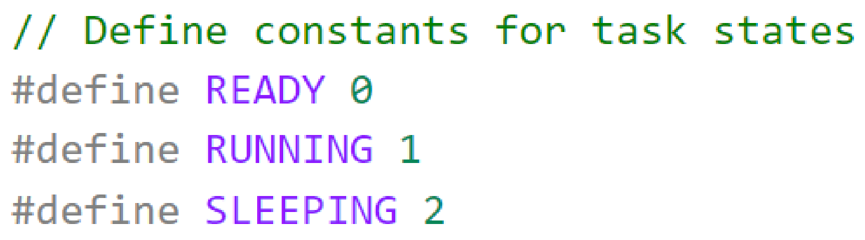
```c
// Initialize timer for the task
timer_setup(&new_reg_task->my_timer, my_timer_function, 0);
//mod_timer(&new_reg_task->my_timer,jiffies+msecs_to_jiffies(1000));

printk("Task with PID %u inserted in list\n", new_reg_task->my_task->pid);
// Find the correct position to insert the task based on priority
list_for_each_entry(d_task, &my_list, next_prev_list) {
    if (new_reg_task->priority>=d_task->priority){
        //we have found the location,to insert our task
        break;
    }
}
  // Insert the new task into the list
list_add(&new_reg_task->next_prev_list, d_task->next_prev_list.prev);

// Perform schedulability test to check if the process satisfies criteria
if (!schedulability_test()) {
    printk("Process do not satisfy schedulability criteria\n");
```

Figure 1: Register system call



```c
SYSCALL_DEFINE1(yield, pid_t, pid) {
    struct my_task_struct* task;
    struct kernel_siginfo info;
    list_for_each_entry(task, &my_list, next_prev_list) {
        if (task->my_task->pid == pid) {
            printk("Yield the process with PID %u\n", pid);
            // Send SIGSTOP signal to suspend the task
            info.si_signo = SIGSTOP;
            send_sig_info(SIGSTOP, &info, task->my_task);
            printk("Signal send succesfully \n");
            // Update the task state to SLEEPING
            task->state = SLEEPING;
            if (task == my_curr_tasks)
                my_curr_tasks = NULL;
            break;
        }
    }
    printk("Dispatcher is called\n");
     // Modify the task timer and wake up the dispatcher thread
    mod_timer(&task->my_timer, task->period);
    if (dispatcher_thread)
        wake_up_process(dispatcher_thread);
    else
```

Figure 2: yield system call



```
// Define constants for task states
#define READY 0
#define RUNNING 1
#define SLEEPING 2
```

Figure 3: Task States

- **int sys_remove(pid)**

  sys_remove system call is responsible for removing a task associated with a specific Process ID and free all data structures allocated during registration. This removal process involves several steps: first, the task is removed from the linked list of tasks using the **list_del function**, effectively unlinking it from the list. Next, the timer associated with the task is deleted using the **del_timer function**, ensuring that any pending timer events are discarded. Finally, the memory allocated for the task is freed using the **kfree function to prevent memory leaks.**

- **void sys_list()**

  sys_list system call is designed to list all registered tasks along with their relevant parameters such as **PID, period, deadline, and execution time**.

2.2 Implementation Challenges

- Waking up an application when it is ready to run. We have 3 states in the kernel for this, **READY**, in which an application has reached the beginning of the period and a new job is ready to be scheduled. **RUNNING**, in which an application is currently executing a job. This application is currently using the CPU. **SLEEPING**, in which an application has finished executing the job for the current period and is waiting for the next period,
- Preempting the current application when another application with a higher priority is ready to run. This involves triggering a context switch, using Linux Scheduler API.
- Preempting an application that has finished its current job.The process notifies the scheduler that it has finished its job for the current period. Upon receiving a yield message, RMS/DMS must put the process to sleep until the next period. This involves setting up a timer and preempting the CPU to the next ready application with the highest priority.

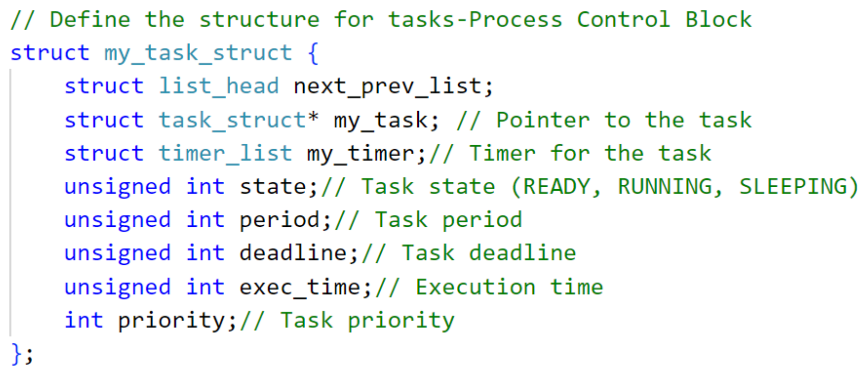
2.3 Process Control Block

We have augmented the Process Control Block of each task, which includes the application state (READY, SLEEPING, RUNNING), a wakeup timer for each task, and the scheduling parameters (Di, Pi, Ti), of the task, including the period of the application (which denotes the priority in RMS). Also, the scheduler will need to keep a list of all registered tasks to pick the correct task to schedule during any preemption point, as shown in figure 4. We are not going to directly modify the Linux PCB (struct task_struct), but instead declare a separate data structure that points to the corresponding PCB of each task.

2.4 How Scheduling Works?

we will have a **kernel thread (dispatching thread)** that is responsible for triggering context switches as needed. There will be two cases in which a context switch will occur:

- after receiving a yield message from the application, and



```
// Define the structure for tasks-Process Control Block
struct my_task_struct {
    struct list_head next_prev_list;
    struct task_struct* my_task; // Pointer to the task
    struct timer_list my_timer;// Timer for the task
    unsigned int state;// Task state (READY, RUNNING, SLEEPING)
    unsigned int period;// Task period
    unsigned int deadline;// Task deadline
    unsigned int exec_time;// Execution time
    int priority;// Task priority
};
```

Figure 4: Strcture for Process Control Block

- after the wakeup timer of the task expires.

On receives a yield message, RMS/DMS should put the associated process to sleep and setup the wakeup timer. It should also change the task state to SLEEPING. When the wakeup timer expires, the timer interrupt handler should change the state of the task to READY and should wake up the dispatching thread. The timer interrupt handler must not wake up the application.

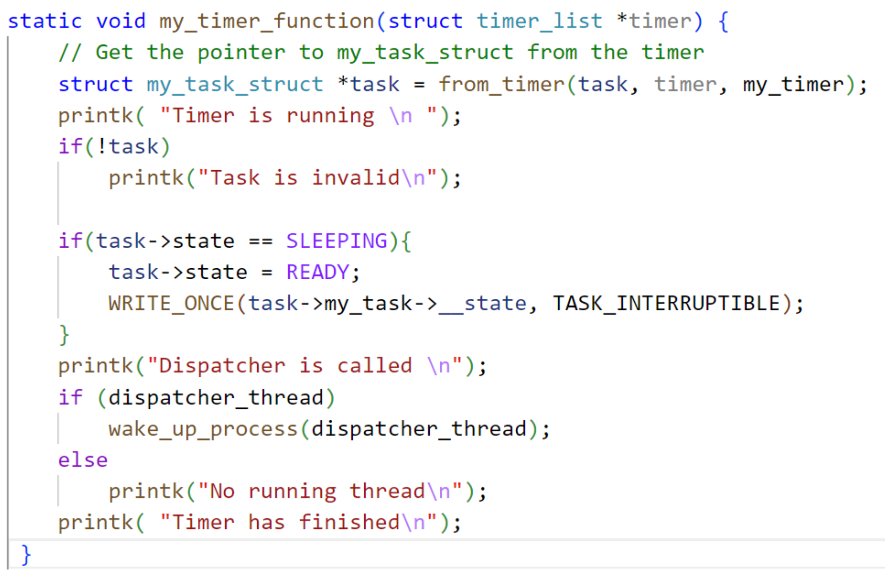
2.5 Scheduler API's

- schedule(): is a function provided by the Linux kernel scheduler. It's used to trigger the scheduler explicitly, allowing the scheduler to decide which process should be executed next on the CPU. - When a process calls schedule(), it voluntarily gives up the CPU and enters a sleep state, allowing another process to run. This function is used when a process wants to yield the CPU to other processes.
- wake_up_process(struct task_struct): is used to wake up a sleeping process. It takes a pointer to the task_struct of the process that needs to be woken up. This function is typically used when a kernel thread or a process needs to wake up another process, for example, after it has finished some work that another process is waiting for.
- sched_setscheduler(): is a function used to set scheduling parameters for a given process. It allows us to specify the scheduling policy and priority for a process. Common scheduling policies include FIFO (First-In-First-Out) for real-time scheduling, NORMAL for regular processes.
- set_current_state(): is a macro that sets the state of the currently running process. It's typically used by kernel code to change the state of the current process to indicate that it is about to sleep or wake up. The possible states are TASK_RUNNING, TASK_INTERRUPTIBLE, TASK_UNINTERRUPTIBLE, etc.
- set_task_state(): is similar to set_current_state() but it allows setting the state for a specific task identified by its task_struct. It's used to change the state of a specific task to indicate that it is about to sleep or wake up. This function is used by kernel threads when they want to enter a sleep state and wait for some condition to be satisfied before waking up again.

2.6 Timer function

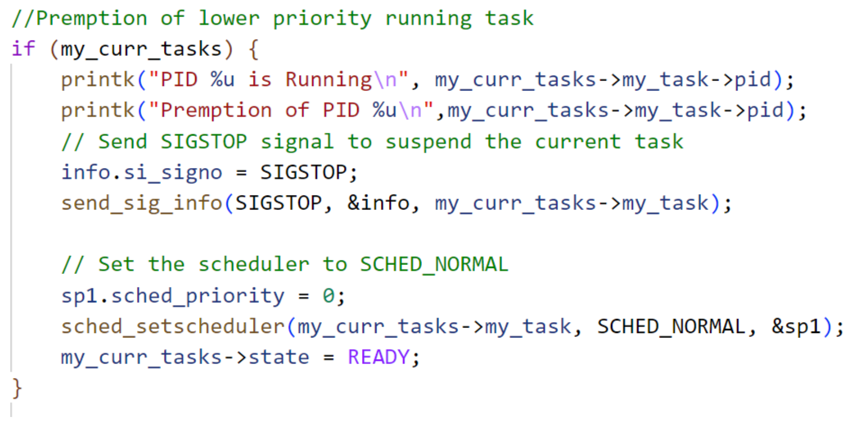
The my_timer_function, designed to serve as a callback for a timer event. The pointer to a my_task_struct structure is extracted from the timer_list structure. If the task is in a **SLEEPING state**, the function transitions **the task's state to READY** and alters its underlying kernel state to **TASK_INTERRUPTIBLE**, suggesting the task is now ready for scheduling.

Following this, we invoke the the dispatcher function, which is responsible for task scheduling.It is awakened to perform its scheduling duties;



```
static void my_timer_function(struct timer_list *timer) {
    // Get the pointer to my_task_struct from the timer
    struct my_task_struct *task = from_timer(task, timer, my_timer);
    printk( "Timer is running \n ");
    if(!task)
        printk("Task is invalid\n");

    if(task->state == SLEEPING){
        task->state = READY;
        WRITE_ONCE(task->my_task->__state, TASK_INTERRUPTIBLE);
    }
    printk("Dispatcher is called \n");
    if (dispatcher_thread)
        wake_up_process(dispatcher_thread);
    else
        printk("No running thread\n");
    printk( "Timer has finished\n");
}
```

Figure 5: Timer implementation



```
//Premption of lower priority running task
if (my_curr_tasks) {
    printk("PID %u is Running\n", my_curr_tasks->my_task->pid);
    printk("Premption of PID %u\n",my_curr_tasks->my_task->pid);
    // Send SIGSTOP signal to suspend the current task
    info.si_signo = SIGSTOP;
    send_sig_info(SIGSTOP, &info, my_curr_tasks->my_task);

    // Set the scheduler to SCHED_NORMAL
    sp1.sched_priority = 0;
    sched_setscheduler(my_curr_tasks->my_task, SCHED_NORMAL, &sp1);
    my_curr_tasks->state = READY;
}
```

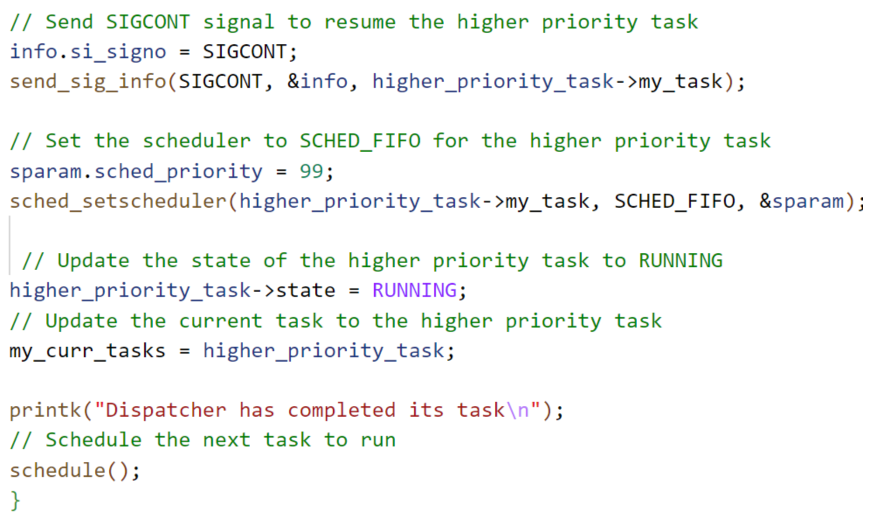
Figure 6: Premption of lower priority running task

2.7 Dispatcher thread Implementation

The dispatcher thread callback function is responsible for managing task scheduling. As soon as the dispatching thread wakes up, we need to find the highest priority (i.e., shortest period) READY task in the list. Then, we need to preempt the currently running task (if any) and context switch to the chosen task. If there are no tasks in READY state, simply preempt the currently running task. Set the new task's state to RUNNING. The state of the preempted task is set to READY only if the state was RUNNING. This is because the yield handler will set the preempted task's state to SLEEPING.

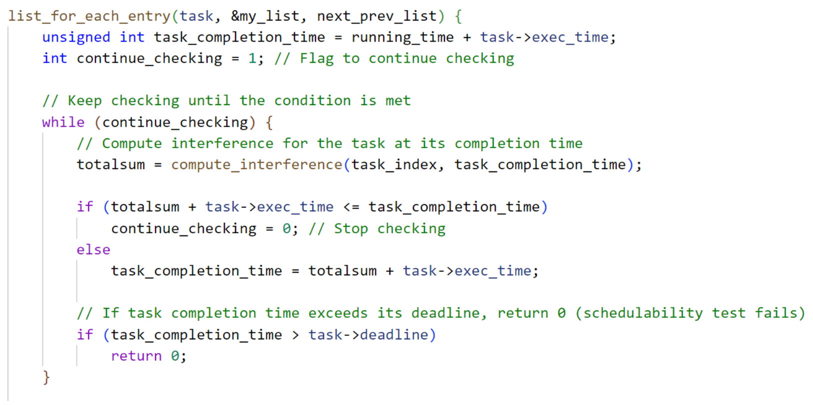
To handle the context switches and preemption, we will use the scheduler API based on some known behavior of the Linux scheduler. Any task running on **SCHED_FIFO will hold the CPU for as long as the application needs.** So we can **trigger a context switch by using the function sched_setscheduler().** we can use the functions **set_current_state() and schedule() to get the dispatching thread to sleep.** For the new running task, the dispatching thread should execute. Similarly, for the preempted task, the dispatching thread should execute.

The function then switches execution to the higher priority task, resuming its execution, **setting its scheduler to SCHED_FIFO with maximum priority, and updating its state to RUNNING.**



```
// Send SIGCONT signal to resume the higher priority task
info.si_signo = SIGCONT;
send_sig_info(SIGCONT, &info, higher_priority_task->my_task);

// Set the scheduler to SCHED_FIFO for the higher priority task
sparam.sched_priority = 99;
sched_setscheduler(higher_priority_task->my_task, SCHED_FIFO, &sparam);

 // Update the state of the higher priority task to RUNNING
higher_priority_task->state = RUNNING;
// Update the current task to the higher priority task
my_curr_tasks = higher_priority_task;

printk("Dispatcher has completed its task\n");
// Schedule the next task to run
schedule();
}
```

Figure 7: Switch to the higher priority task



```
list_for_each_entry(task, &my_list, next_prev_list) {
    unsigned int task_completion_time = running_time + task->exec_time;
    int continue_checking = 1; // Flag to continue checking

    // Keep checking until the condition is met
    while (continue_checking) {
        // Compute interference for the task at its completion time
        totalsum = compute_interference(task_index, task_completion_time);

        if (totalsum + task->exec_time <= task_completion_time)
            continue_checking = 0; // Stop checking
        else
            task_completion_time = totalsum + task->exec_time;

        // If task completion time exceeds its deadline, return 0 (schedulability test fails)
        if (task_completion_time > task->deadline)
            return 0;
    }
```

Figure 8: Schedulability test check

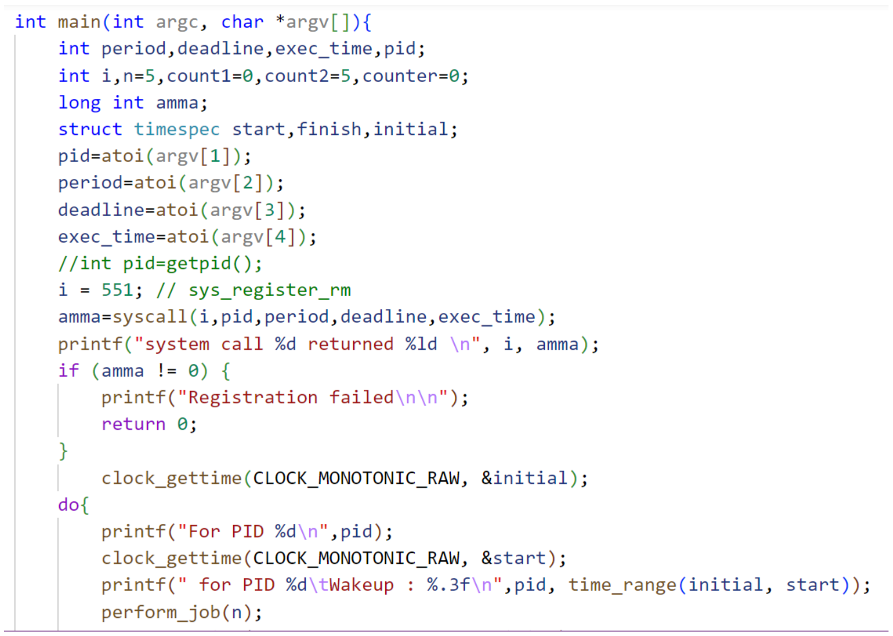
2.8 Schedulability Test

The ‘compute_interference‘ function calculates the interference caused by tasks with lower priority than the task currently under consideration. It iterates through the list of tasks until it reaches the task with the **given index current_task_index**. For each task encountered, it calculates the cumulative interference by considering the number of times the task can execute within its period up to the current time.

The schedulability_test function **evaluates whether the system remains schedulable, ensuring that all tasks meet their deadlines.** It iterates through the list of tasks and **computes the completion time of each task considering its execution time and interference from lower-priority tasks**. This computation continues iteratively until the completion time no longer increases, indicating that the task's schedule is feasible. If a task's completion time exceeds its deadline at any point during this process, the function immediately returns 0, indicating that the system is not schedulable. Otherwise, **if all tasks can complete within their deadlines, the function returns 1**, signifying that the system remains schedulable.

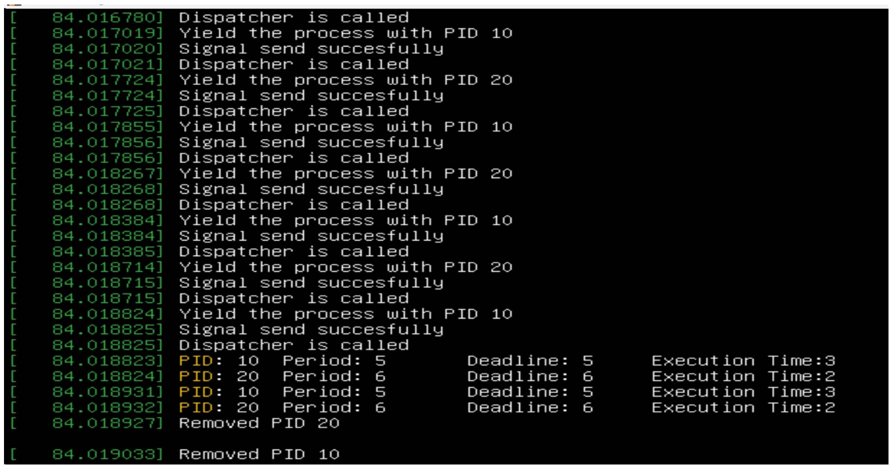
2.9 User Application

This application is a single-threaded periodic application with individual jobs doing some computation. In the main function, command-line arguments are parsed to obtain parameters such as process ID, period, deadline, and execution time. These parameters are then used to make a system call



```c
int main(int argc, char *argv[]){
    int period,deadline,exec_time,pid;
    int i,n=5,count1=0,count2=5,counter=0;
    long int amma;
    struct timespec start,finish,initial;
    pid=atoi(argv[1]);
    period=atoi(argv[2]);
    deadline=atoi(argv[3]);
    exec_time=atoi(argv[4]);
    //int pid=getpid();
    i = 551; // sys_register_rm
    amma=syscall(i,pid,period,deadline,exec_time);
    printf("system call %d returned %ld \n", i, amma);
    if (amma != 0) {
        printf("Registration failed\n\n");
        return 0;
    }
        clock_gettime(CLOCK_MONOTONIC_RAW, &initial);
    do{
        printf("For PID %d\n",pid);
        clock_gettime(CLOCK_MONOTONIC_RAW, &start);
        printf(" for PID %d\tWakeup : %.3f\n",pid, time_range(initial, start));
        perform_job(n);
```

Figure 9: User Application



```
[   84.016780] Dispatcher is called
[   84.017019] Yield the process with PID 10
[   84.017020] Signal send succesfully
[   84.017021] Dispatcher is called
[   84.017724] Yield the process with PID 20
[   84.017724] Signal send succesfully
[   84.017725] Dispatcher is called
[   84.017855] Yield the process with PID 10
[   84.017856] Signal send succesfully
[   84.017856] Dispatcher is called
[   84.018267] Yield the process with PID 20
[   84.018268] Signal send succesfully
[   84.018268] Dispatcher is called
[   84.018384] Yield the process with PID 10
[   84.018384] Signal send succesfully
[   84.018385] Dispatcher is called
[   84.018714] Yield the process with PID 20
[   84.018715] Signal send succesfully
[   84.018715] Dispatcher is called
[   84.018824] Yield the process with PID 10
[   84.018825] Signal send succesfully
[   84.018825] Dispatcher is called
[   84.018823] PID: 10  Period: 5       Deadline: 5      Execution Time:3
[   84.018824] PID: 20  Period: 6       Deadline: 6      Execution Time:2
[   84.018931] PID: 10  Period: 5       Deadline: 5      Execution Time:3
[   84.018932] PID: 20  Period: 6       Deadline: 6      Execution Time:2
[   84.018927] Removed PID 20

[   84.019033] Removed PID 10
```

Figure 10: Kernel log 1

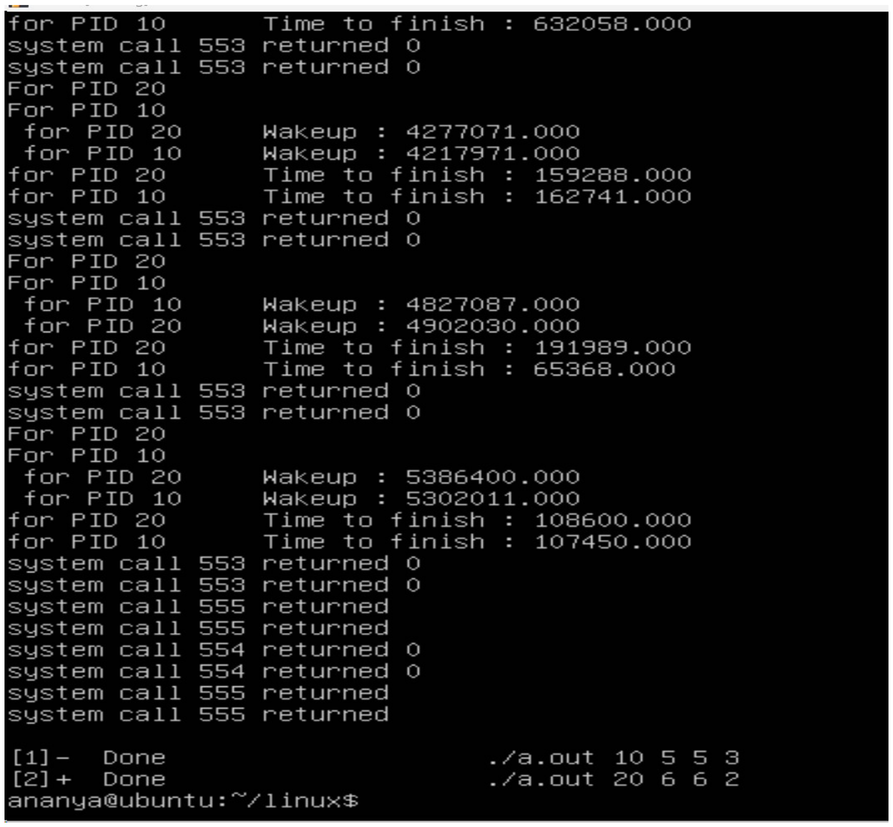
sys_register_rm. A loop is then executed, wherein the process wakes up periodically, performs some computational task using perform_job function which executes a factorial calculation, and then yields the CPU. After the loop, additional system calls are made to list and remove the registered real-time process.

2.10 Outputs with my User Application

- Output1:

```
$ gcc test.c $
$ ./a.out 10 5  5 3 \& ./a.out 20 6 6 2 &
```

- Kernel Log for Output1:



```
for PID 10       Time to finish : 632058.000
system call 553 returned 0
system call 553 returned 0
For PID 20
For PID 10
 for PID 20      Wakeup : 4277071.000
 for PID 10      Wakeup : 4217971.000
for PID 20       Time to finish : 159288.000
for PID 10       Time to finish : 162741.000
system call 553 returned 0
system call 553 returned 0
For PID 20
For PID 10
 for PID 10      Wakeup : 4827087.000
 for PID 20      Wakeup : 4902030.000
for PID 20       Time to finish : 191989.000
for PID 10       Time to finish : 65368.000
system call 553 returned 0
system call 553 returned 0
For PID 20
For PID 10
 for PID 20      Wakeup : 5386400.000
 for PID 10      Wakeup : 5302011.000
for PID 20       Time to finish : 108600.000
for PID 10       Time to finish : 107450.000
system call 553 returned 0
system call 553 returned 0
system call 555 returned
system call 555 returned
system call 554 returned 0
system call 554 returned 0
system call 555 returned
system call 555 returned

[1]-  Done                    ./a.out 10 5 5 3
[2]+  Done                    ./a.out 20 6 6 2
ananya@ubuntu:~/linux$
```

Figure 11: Output1

2.11 Deadline monotonic algorithm might be preferred over the rate monotonic scheduling algorithm and vice versa with suitable examples.

- Rate Monotonic Scheduling (RMS) where tasks are scheduled based on their fixed priority levels determined by their periods. **Tasks with shorter periods have higher priorities**.RMS is preferred in the following cases:

  1. **Tasks with shorter periods compared to their execution times.** This is because RM assigns priorities based on periods, and shorter periods result in higher priorities, allowing critical tasks to meet their deadlines.

  2. **Less frequent task releases** : If tasks are released less frequently, RM can efficiently utilize system resources because it assigns priorities based on fixed periods rather than deadlines.

- Deadline Monotonic Scheduling (DM): It assigns priorities based on task deadlines rather than periods. Tasks with earlier deadlines are assigned higher priorities. DM is less restrictive in terms of schedulability analysis. DMs is preferred in the following cases:

  1.**It allows tasks with tighter deadlines to be prioritized appropriately**. DMS is suitable for systems where task deadlines are closer to their periods or where deadlines are more critical than periods.

  2. In systems where tasks may arrive, depart, or change dynamically, DM can adapt more flexibly than RM. It allows for **more efficient utilization of system resources by prioritizing tasks based on their dynamic deadlines.**