Operating Systems
Assignment 2 – *Hard*

Instructions:

1. The assignment has to be done individually.
2. You can use Piazza for any queries related to the assignment and avoid asking queries on the last day.

1 Real Time Scheduling

Real-time systems are developed to react to events within a strict time frame. These systems are deployed in a wide spectrum of applications ranging from medical equipment, financial trading, and industrial automation to military systems, air-traffic control, and spacecraft navigation. Soft real-time systems have relaxed timing constraints and do not incur severe consequences whereas missing a deadline in hard real-time systems can have serious repercussions.

Liu and Layland's periodic task model is a mathematical model for scheduling periodic tasks in real-time systems. For a set of **n** periodic tasks, the model assumes that each task i has a period P_i and an execution time T_i. The period P_i is the time interval between successive releases of the task. We will further assume that the relative deadline of a task i is represented by D_i. Relative deadline (D_i) is the time interval between the task's release time and its deadline.

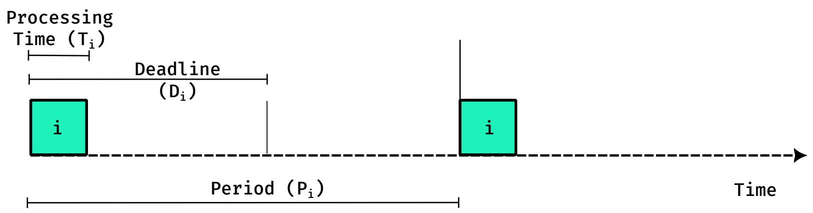

Figure 1: The Periodic Task Model

Deadline monotonic scheduling and rate monotonic scheduling are both static priority scheduling algorithms in which priorities are assigned based on the relative deadline(D_i) and the period(P_i) of the task respectively.

Implement the deadline monotonic (DM) and rate monotonic (RM) scheduling algorithms for the Linux kernel. The admission controller should ensure that the new processes meet its timing constraints. This includes the process's deadline, the worst-case execution time, and the period. The controller should ensure that the new process's timing requirements do not conflict with existing processes or exceed the system's capacity. Implement system calls to provide the details about the process's release time, period, worst-case execution time, and deadline.

1. We have a set of n preemptable tasks $\mathcal{T}$, such that the tasks do not share resources, and no precedence order exists among the tasks.

2. A process must register itself for RM or DM scheduling with the help of the following system calls:

   ```
   int sys_register_dm( pid,  period,  deadline, exec_time)
                   OR
   int sys_register_rm( pid,  period,  deadline, exec_time)
   ```

   - *pid:* pid of the process.
   - *period:* period of the process (P_i).
   - *exec_time:* the number of ticks for which the process should run (T_i).
   - *deadline:* the relative deadline of the process (D_i).

   If the process is schedulable, it should return 0; otherwise, it should return -22. You can read about schedulability checks from the following link RM scheduling.

3. To notify the kernel that a process has finished executing, the process must send a yield message using the *sys_yield* system call for which the signature is as follows:

   ```
    int sys_yield(pid)
   ```

   - *pid:* pid of the process.

   If the system call was successful, it should return 0; otherwise, it should return -22.

4. To remove the process *sys_remove* system call must be invoked with the following signature:

   ```
    int sys_remove(pid)
   ```

   - *pid:* pid of the process.

   If the system call was successful, it should return 0; otherwise, it should return -22.

5. To list all the registered processes *sys_list* system call must be invoked with the following signature:

```
void sys_list()
```

- *pid:* pid of the process.

This system call should print in the following format:

```
PID: period : deadline: execution time
PID: period : deadline: execution time
PID: period : deadline: execution time
```

6. You must implement an application to test the scheduler. The pseudo-code for the application is mentioned below:

```
int main(int argc, char *argv[]) {

  period = argv[1];
  exec_time = argv[2];
  deadline = argv[3];

  pid_t = getpid();
  syscall(sys_register_dm(pid_t, period, deadline,
     exec_time));

  init = gettime();

  do{
    exec_start = gettime();
    wakeup_time = (exec_start - init);
    print("Wakeup:", wakeup_time);

    perform_job(processing_time);

    finish = gettime();
    finish_time = (finish - exec_start);
    print("Time to finish:", finish_time);

    syscall(sys_yield(pid_t));
  } while (true);

  syscall(sys_remove(pid_t));
}
```

You will need to create a supplementary Process Control Block (PCB) that holds information about the current state (Ready, Sleeping, or Executing), process parameters (D_i, P_i, T_i), and timers (implementation dependent). When the kernel receives a yield message, it should put the associated process into a *sleeping* state and set up a per-process timer. Once the timer expires, the timer handler should change the state of the process to *ready* and wake up the kernel thread responsible for scheduling. This kernel thread must choose the process with the highest priority that is in the *ready* state and context switch to this process. The currently executing process might get preempted by the scheduler during the context switch. To start a process, the kernel thread should execute the *wake_up_process()* function. The kernel thread can use the *set_current_state ()* function to sleep while waiting for the next process.

Question 1: Explain the circumstances wherein the deadline monotonic algorithm might be preferred over the rate monotonic scheduling algorithm and vice versa with suitable examples.

1.1 Priority Ceiling Protocol (PCP)

Whenever a certain low-priority task holds a resource requested by a higher-priority task, it results in the higher-priority task getting blocked and becoming incapable to proceed. Since the scheduler constantly preempts the lower-priority task to execute the higher-priority one, it leads to a phenomenon called priority inversion. Priority Ceiling Protocol (PCP) thwarts priority inversion by allocating a priority ceiling to each shared resource. When a process acquires a resource, the priority of that process is temporarily raised to the priority ceiling (the highest priority of any process that acquires the resource) of the resource, assuring that a lower-priority task cannot preempt it. The resource for the scope of this assignment would be a read-only file that multiple processes try to access. Implement a priority ceiling protocol on top of RM.

Question 2: In case the Linux kernel utilizes the priority inversion protocol instead of the priority ceiling protocol. What will be the advantages and disadvantages of it?

2 Report

Page limit: 10
The report should clearly mention the implementation methodology for all the real-time scheduling policies. Showing small, representative code snippets in the report is alright, additionally, the pseudocode should also suffice.

- Implementation of the DM and RM scheduling policies.
- Answers to the in-text questions.
- Challenges faced and the novelties introduced (if any).
- Submit a pdf file containing all the relevant details.