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. The Relative deadline (D_i) is the time interval between the task's release time and its deadline.

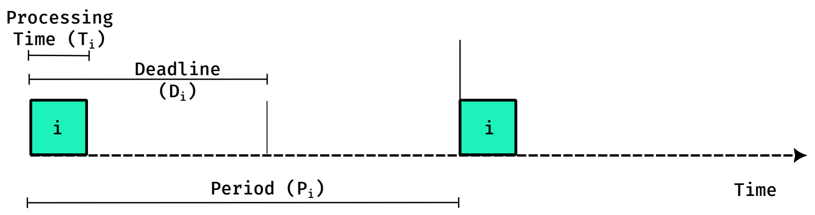

Figure 1: The Periodic Task Model

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

Implement the deadline monotonic (DM) and rate monotonic (RM) scheduling algorithms in 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.

Note that these are all uniprocessor algorithms, and thus the implementation should ensure that all the real-time tasks (that are a part of this assignment) run on the same core. They must use the same *rq*.

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(**type**: unsigned integer, **unit**: milliseconds):* period of the process (P_i).
   - *deadline (**type**: unsigned integer, **unit**: milliseconds):* The relative deadline of the task (after it has been released).
   - *exec_time (**type**: unsigned integer, **unit**: milliseconds):* The execution time of the task (T_i). You need to ensure that the work that you are doing will finish by *exec_time* milliseconds. This can be ensured by running the function of interest multiple times in isolation and taking the maximum. This assumption will hold since the demo will be on your machine.

   If the augmented (new) set of processes is schedulable, the corresponding system call should return 0; otherwise, they should return -22. You can read about schedulability checks from the following link RM scheduling.

3. To notify the kernel that an instance of a task 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. Note: You can also modify the implementation of the regular *sched_yield* system call and use that.

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

   ```
   void sys_list()
   ```

   - *pid:* pid of the process.

   This system call should print its result in the following format:

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

6. You must execute multiple applications to test the scheduler. The pseudo-code for one of the applications is shown 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);
    // you must ensure that this function finishes
       within the processing_time

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

    syscall(sys_yield(pid_t));
```

```
    //scheduling should be performed after the first
        invocation of sys_yield()
  } while (job);
  // perform_job() updates the status of the job flag

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

1.1 Priority Ceiling Protocol (PCP)

You need to implement Priority Ceiling Protocol(PCP) for the deadline monotonic scheduling (DMS). For the scope of this assignment, we will work with dummy resources. Each resource will have a unique resource id that must be specified when locking. The signature of the required system calls is shown below:

1. To notify the kernel about the future resource requests of a process, the *sys_resource_map()* system call must be invoked:

   ```
   void sys_resource_map(pid, RID)
   ```

   - *pid:* pid of the requesting process.
   - *RID*(**type**: unsigned integer): unique resource id.

2. The *sys_start_pcp()* system call must be invoked to notify the kernel that all the processes have arrived and their process-resource mapping is finalized:

   ```
   void sys_start_pcp(num_process)
   ```

   - *num_process (type: unsigned integer):* the number of processes registered for DMS.

   Note: Adjust the period and processing time of all the tasks registered for DM in such a way that the tasks are scheduled (once they have yielded) for the first time only after the *sys_start_pcp()* system call has been invoked.

3. To notify the kernel that an instance of a task is requesting a resource with a resource id (*RID*), the *sys_pcp_lock()* system call must be invoked:

   ```
   void sys_pcp_lock(pid, RID)
   ```

   - *pid:* pid of the requesting process.
   - *RID*(**type**: unsigned integer): unique resource id.

   If a process with a higher priority invokes the *sys_pcp_lock()* system call, the priority of the original process with the lock must be adjusted according to the priority ceiling protocol.

4. To unlock the resource, the *sys_pcp_unlock()* system call must be invoked with the following signature:

```
void sys_pcp_unlock(pid, RID)
```

- *pid:* pid of the process.
- *RID*(**type**: unsigned integer): unique resource id.

The priorities must be adjusted, and the scheduler must be invoked to schedule the process with the highest priority.

The *sys_pcp_lock()* and *sys_pcp_unlock()* system calls can be used inside the *perform_job()* function to lock a dummy resource. A separate application must register the processes scheduled using DMS for PCP using *sys_resource_map()* call. Once all the process-resource entries have been made, the separate application must invoke the *sys_pcp_start()* system call to signal the finalization of mappings for PCP.

2 Report

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

- Implementation of the DM and RM scheduling policies.
- Challenges faced and the novelties introduced (if any).
- Submit a PDF file containing all the relevant details.