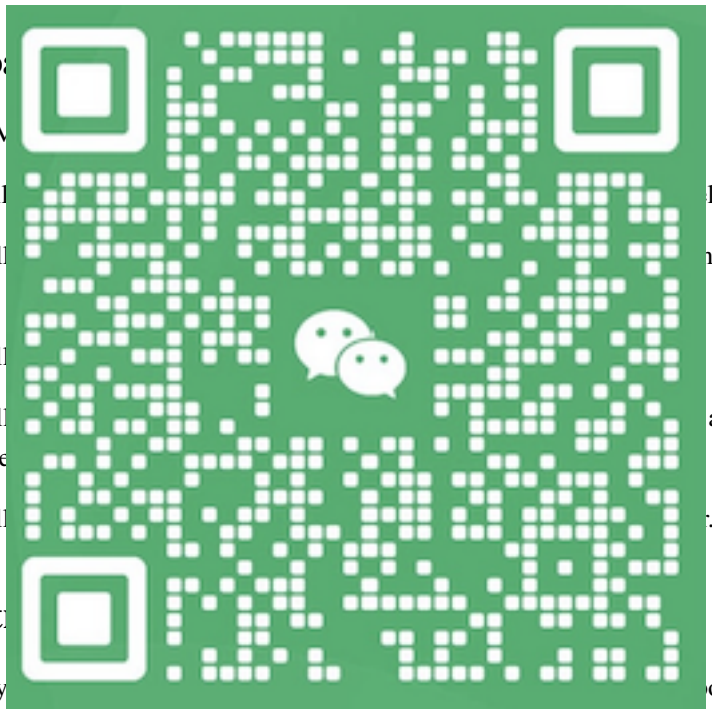
CS423 Fall 2022

MP2: Rate-Monotonic CPU Scheduling



1 Go

- In this M
- You will l Modules
- You will ic Scheduler
- You will
- You will allocation in the ke
- You will .

2 Int

Several sy onse time (e.g. delay and jitter) and *predictability* for the safety or enjoyment of their users. For example, a surveillance system needs to record video of a restricted area, the video camera must capture a video frame every 30 milliseconds. If the capture is not properly scheduled, the video quality will be severely degraded.

For this reason, the Real-Time systems area has developed several algorithms and models to provide this precise timing guarantees as close to mathematical certainty as needed. One of the most common models used is the *Periodic Task Model*.

A Periodic Task as defined by the Liu and Layland model [10] is a task in which a job is released after every period P, and must be completed before the beginning of the

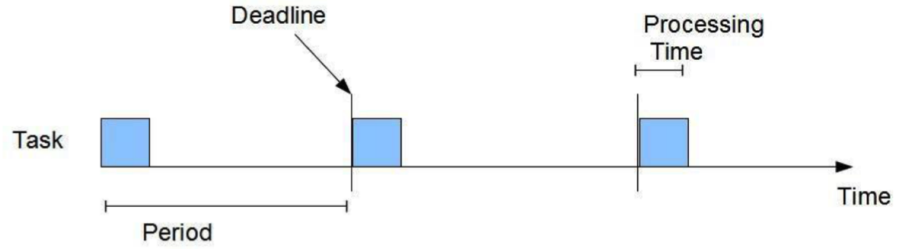


Figure 1: Liu and Layland Periodic Task Model

next perio res certain processing

In this d Periodic Task Mod *er (RMS)*. The RMS l based on the period heduler is preemptiv *mpt a task with a low*

3 Pro

In this MP -Monotonic Scheduler scheduler as a Linu tween the scheduler *try* for all the commu user. Our scheduler ystem:

- *Registration*: This allows the application to notify to Kernel module its intent to use the RMS scheduler. The application communicates its registration parameters to the kernel module in the following format:

  `R,<pid>,<period>,<processing time>`

  where `<pid>` is the integer PID of the process, `<period>` is the task period, and `<processing time>` is the task processing time. All times should be in **milliseconds** and **encoded as integers**. Notice that the "R" in the string is a literal 'R' that denotes this is a registration message.

[1]Please note that in this documentation we will use the term application and task interchangeably.

- *Yield*: This operation notifies the RMS scheduler that the application has finished its period. After a yield, the application will block until the next period. Yield messages are strings with the following format:

  `Y,<pid>`

  where `<pid>` is the integer PID of the process.

- *De-Registration*: This allows the application to notify the RMS scheduler that the application has finished using the RMS scheduler.

  `D,<pid>`

  whe

Your R ation if the
applicatio admission
control mu er already
admitted p registered
tasks in th on in your
kernel mo *cation will*
use more t

This s nes; there-
fore, you s low-level
functions.

Additi ery which
applicatio registered
applicatio cation, the
kernel mo e of each
applicatio

```
<pid 1>: <period 1>, <processing time 1>
<pid 2>: <period 2>, <processing time 2>
...
<pid n>: <period n>, <processing time n>
```

You will also develop a simple test application for our scheduler. This application will be a single-threaded periodic application with individual jobs doing some computations. It must do the following in order:

1. This periodic application must register itself with the scheduler (through admission control). During the registration process it must specify its scheduling pa-

rameters: The Period of the jobs expressed in milliseconds and Processing Time of each job also expressed in milliseconds.

2. After the registration the application must read the `/proc/mp2/status` entry to ensure that its PID is listed. This means the task is accepted.

3. After this, the application must signal the scheduler that it is ready to start by sending a YIELD message to `/proc/mp2/status`.

4. Then the application must initiate the Real-Time Loop, and begin the execution of the periodic jobs. One job is equivalent to one iteration of the Real-Time Loo tus.

5. At t itself after finis

Below

```
int m
{
   RE                                                    system
   li                                                    as

   if

   //
   t0
   YI

   //
   wh
   {
      wakeup_time = gettimeofday() - t0;
      do_job(); // factorial computation
      YIELD(PID); //ProcFS. JobProcessTime=gettimeofday()-
          wakeup_time
   }
   DEREGISTER(PID); //ProcFS
   return 0;
}
```

To determine the processing time of a job you can run the application using the Linux scheduler first and measuring the average execution time of one iteration of the Real-Time Loop.

Additionally, your application can perform a simple computation, we recommend calculating the factorial of a fixed number. This MP is about Real-Time scheduling, so keep the application simple.

4 Implementation Challenges

Scheduling might find yourself w y:

1) The first run. Rate Monoto any appli- cation t *sleep until the begi* luable re- sources arly states that you

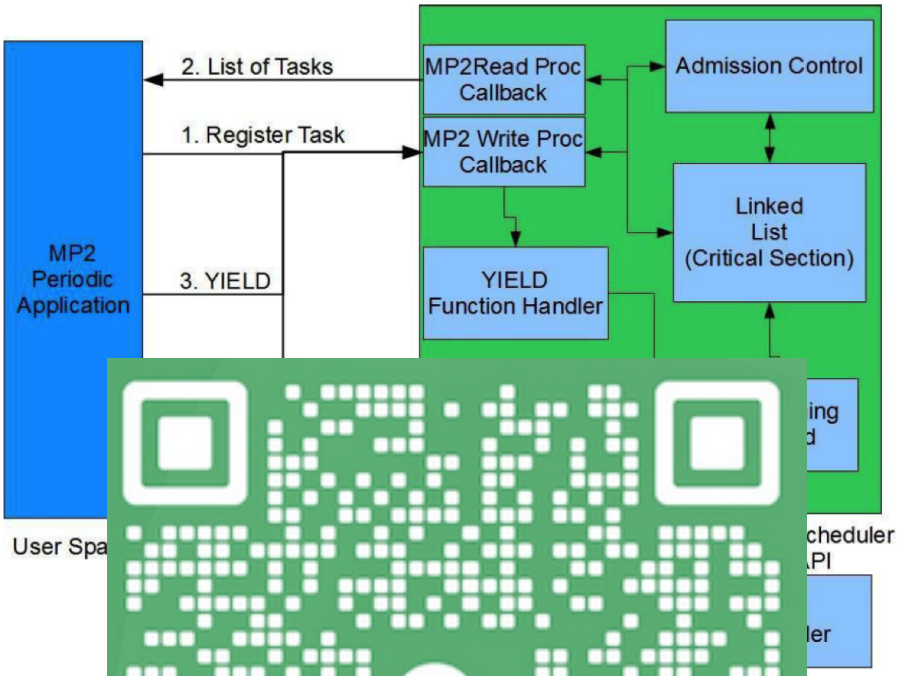
   – **RE** the period an

   – **RU** job. This ap

   – **SL** job for the cu

2) The sec *er priority than the current application as soon as this application becomes available to run.* This involves triggering a context switch. You will use the Linux Scheduler API for this.

3) The third challenge is to preempt an application that has finished its current job. To achieve this you will assume that the application always behaves correctly and notifies the scheduler that it has finished its job for the current period. Upon receiving a YIELD message from `/proc/mp2/status`, the RMS scheduler must put the application 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.



From t trol Block of each ta EEPING, RUNNIN(f the task, including . Also, the scheduler o you can pick the c *challenge* is perform arithmetic is very expensive and therefore it must be avoided.

5 Implementation Overview

In this section, we will guide you through the implementation process. Figure 2 shows the basic architecture of our scheduler.

1. The best way to start is by implementing an empty ('Hello World!') Linux Kernel Module. You should also be able to reuse some of the most generic functions you implemented on MP1, like linked list helper functions and so.