Machine Problem 3 - Scheduling

CSIE3310 - Operating Systems
National Taiwan University

Total Points:	100
Release Date:	April 22
Due Date:	May 05, 23:59:00
TA e-mail:	`ntuos@googlegroups.com`
TA hours:	Wed. & Thu. 10:00–11:00 before the due date, CSIE Building R604

Contents

1 Overview

This MP is divided into two parts. For each part, you are going to implement two scheduling algorithms.

1. None Real-time scheduling algorithms including Highest Response Ratio Next (HRRN) and Priority-based Round Robin.
2. Real-time scheduling algorithms with periodic tasks, including Earliest Deadline First (EDF) with Constant Bandwidth Server (CBS) scheduling and Deadline-Monotonic(DM) scheduling.

2 Environment Setup

1. Download the MP3.zip from NTU COOL, unzip it, and enter it.

```
$ unzip MP3.zip
$ cd mp3
```

2. Pull docker image from Docker Hub

```
$ docker pull ntuos/mp3
```

3. In the `mp3` directory, use `docker run` to enter the container.

```
$ docker run -it -v $(pwd)/xv6:/home/xv6/ -w /home/xv6/ ntuos/mp3
```

3 Threading Library

We provided you a basic threading library allows for the creation, scheduling, and execution of threads with both real-time and non-real-time characteristics. Here's a breakdown of the key components and functions within the threading library:

```
struct thread *thread_create(void (*f)(void *), void *arg,
                             int is_real_time, int processing_time,
                             int period, int n);
```

Note that the function f is NOT called every cycle. The thread is just preempted when it meets the deadline of the current cycle. The thread exits only if it completes all n cycles of processing.

- **Arrival Time**: Use `thread_add_at` to set the initial release time of a thread:

```
void thread_add_at(struct thread *t, int arrival_time);
```

- **CBS Setup**: To enable CBS, call:

```
void init_thread_cbs(struct thread *t, int budget, int is_hard_rt);
```

- **Priority Setting**: For fixed-priority schedulers, use:

```
void thread_set_priority(struct thread *t, int priority);
```

- **Switch Handler**: Handles time progression, CBS budget updates, and thread switching:

```
void switch_handler(void *arg);
```

- **Thread Releasing**: The following function checks the release queue and moves threads to the run queue when their arrival time is reached:

```
void __release(void);
```

- **Thread Dispatching**: Launches the current thread via context switch. Sets up the stack, saves/resumes the thread context, and invokes the function:

```
void __dispatch(void);
```

- **Thread Completion**: Handles post-execution behavior for normal and real-time threads:

```
void __finish_current(void);
void __rt_finish_current(void);
```

- **Thread Termination**: Called when a thread finishes all cycles or exits manually:

```
void thread_exit(void);
void __thread_exit(struct thread *t);
```

- **Scheduling Entry Point**: This function selects the next thread using the current scheduling policy:

```
void __schedule(void);
```

Data Structures:

```
struct thread {
    void (*fp)(void *arg);
    void *arg;
    void *stack;
    void *stack_p;
    int buf_set;
    struct list_head thread_list;

    int thrdstop_context_id;
    int ID;                  // Unique ID
    int is_real_time;        // 1 = RT, 0 = non-RT
    int processing_time;     // Burst time per cycle
    int deadline;            // = period for RT
    int period;              // Cycle interval
    int n;                   // RT cycles count
    int remaining_time;      // Remaining time in current cycle
    int current_deadline;    // Current deadline
    int priority;            // For priority-based schedulers
    int arrival_time;        // Release time of first cycle

    // CBS fields
    struct {
        int budget;
        int remaining_budget;
        int is_hard_rt;              // 1 if hard real-time, 0 if soft real-time
        int is_throttled;            // 1 if the thread is currently throttled
        int throttled_arrived_time;  // Time reset remaining budget
        int throttle_new_deadline;   // New deadline assigned after throttling
    } cbs;
};
```

4 Part I - None Real-time scheduling

Recall that we have implemented a user-level threading library in MP1. One of its limitations is that the threads are *non-preemptive.* In this assignment, we extend our work to support two scheduling strategies with different preemption behaviors. Specifically, you are required to implement:

- **Highest Response Ratio Next (HRRN)** — a *non-preemptive* scheduling algorithm that selects the next thread based on the highest response ratio. This favors shorter jobs that have waited longer without forcibly interrupting currently running threads.
- **Priority-based Round Robin (P-RR)** — a *preemptive* scheduling algorithm that groups threads by priority level. Within each priority group, threads are scheduled using round-robin time slicing. Higher-priority threads preempt lower-priority ones, enabling responsive scheduling across levels.

These algorithms reflect different trade-offs in scheduling policy and require appropriate control logic in your user-level threading library to support both non-preemptive and preemptive behaviors.

4.1 Highest Response Ratio Next (HRRN)

4.1.1 Specifications

Figure 1 shows an example of the Highest Response Ratio Next (HRRN) scheduling algorithm. Five threads (Thread1 to Thread5) arrive at different times and are scheduled non-preemptively based on the response ratio:

$$\text{Response Ratio} = \frac{\text{Waiting Time} + \text{Burst Time}}{\text{Burst Time}}$$

(Hint: use multiplication instead of division to avoid floating point operations)

This favors short jobs that have waited longer. **If multiple threads have the same response ratio, the one with the smaller ID is selected.**

Thread	Arrival Time	Burst Time
T1	0	10
T2	3	6
T3	3	4
T4	8	3
T5	13	5

- Tick 0: Only Thread1 has arrived. It starts execution immediately.
- Tick 10: Thread1 finishes. At this point, Thread2 and Thread3 have arrived (at tick 3). Response Ratios are:
  - Thread2: (10-3 + 6)/6 = 2.17
  - Thread3: (10-3 + 4)/4 = 2.75 ⇒ highest

  Thread3 is selected.
- Tick 14: Thread3 finishes. P4 has arrived at tick 8. Now ready queue has Thread2 and Thread4.
  - Thread2: (14-3 + 6)/6 = 2.83
  - Thread4: (14-8 + 3)/3 = 3.0 ⇒ highest

  Thread4 is selected.
- Tick 17: Thread4 finishes. Thread2 and Thread5 are ready.
  - Thread2: (17-3 + 6)/6 = 3.33 ⇒ Thread2 is selected
  - Thread5: (17-13 + 5)/5 = 1.8
- Tick 23: Thread2 finishes. Only Thread5 remains. It is selected.

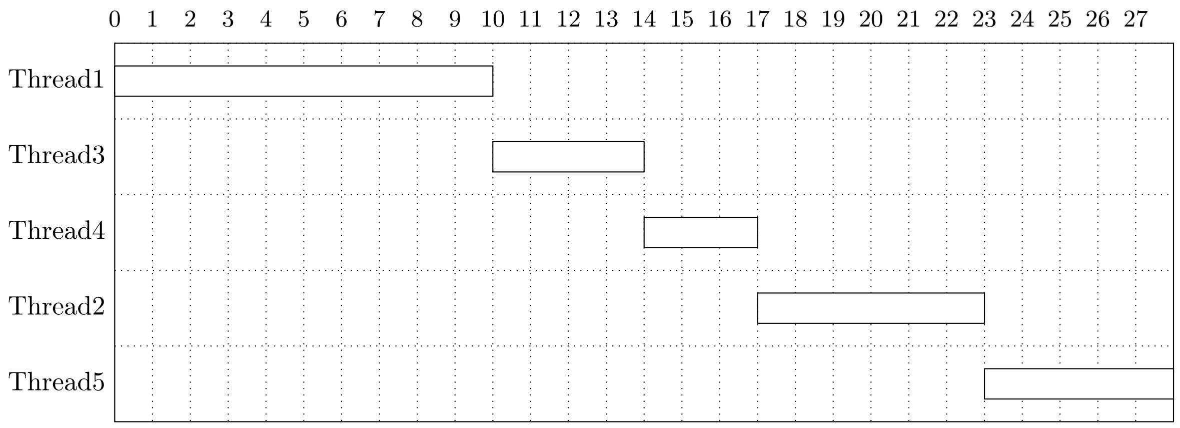

Figure 1: Highest Response Ratio Next (HRRN) Scheduling.

4.1.2 Functions to be Implemented

You have to implement the following function in `user/threads_sched.c`

```
struct threads_sched_result schedule_hrrn(struct threads_sched_args args)
```

4.1.3 Test Case Specifications

- $0 < burst\ time \leq 64$
- $period = -1$
- $n = 1$
- $0 \leq arrival\ time \leq 100$

4.2 Priority-based Round Robin (P-RR)

4.2.1 Specifications

Figure 2 shows an example of the Priority-based Round Robin (P-RR) scheduling algorithm with a time quantum of 2. Five threads (T1 to T5) have different priorities (lower number = higher priority), and within each priority group, threads are scheduled in round-robin order.

The scheduler always selects the highest-priority group first. Once all threads of the highest priority are finished, the next lower-priority group is scheduled.

Thread	Burst Time	Priority
T1	4	3
T2	5	2
T3	6	2
T4	7	1
T5	3	3

- Threads are grouped by priority, with lower values indicating higher priority.
- **If multiple threads share the same priority, they are scheduled in round-robin order, breaking ties by thread ID**.
- Tick 0–6: Only Thread4 (priority 1) is scheduled first. It runs for 7 ticks and finishes.
- Tick 7–17: Threads with priority 2 (Thread2, Thread3) are scheduled next in round-robin manner with time quantum 2:
  - T2 runs 2 ticks (7–8), T3 runs 2 ticks (9–10)
  - T2 runs 2 ticks (11–12), T3 runs 2 ticks (13–14)
  - T2 runs 1 tick (15), T3 runs 2 ticks (16–17)

- Tick 18–26: Threads with priority 3 (T1, T5) are scheduled in round-robin:
  - T1 runs 2 ticks (18–19), T5 runs 2 ticks (20–21)
  - T1 runs 2 ticks (22–23), T5 runs 1 tick (24), T1 runs 1 tick (25).

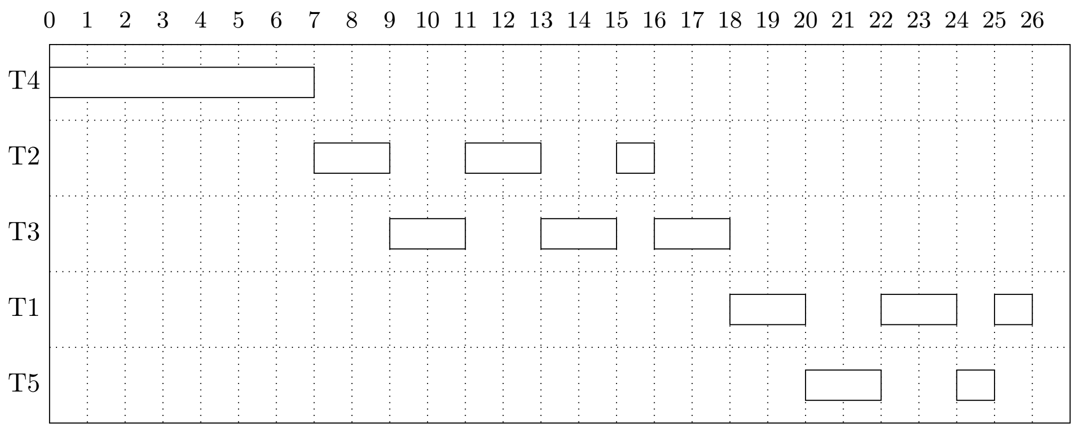

Figure 2: Priority-based Round Robin Scheduling with 5 threads.

4.2.2 Functions to be Implemented

You have to implement the following function in `user/threads_sched.c`

```
struct threads_sched_result schedule_priority_rr(struct threads_sched_args args)
```

4.2.3 Test Case Specifications

- $Time\ quantum = 2$
- $0 < burst\ time \leq 64$
- $0 \leq priority \leq 4$
- $period = -1$
- $n = 1$
- $arrival\ time = 0$

4.3 Run the Public Tests

You can run the following command inside the docker (not in xv6).

```
root@ffffffffffff:/home/xv6# python3 grade-mp3-HRRN.py
...
== Test task1 - HRRN == task1 - HRRN: OK (4.0s)
== Test task2 - HRRN == task2 - HRRN: OK (2.6s)
== Test task3 - HRRN == task3 - HRRN: OK (2.8s)
== Test task4 - HRRN == task4 - HRRN: OK (2.1s)
== Test task5 - HRRN == task5 - HRRN: OK (2.5s)
Score: 10/10
...
```

You can also manually run `task${n} HRRN` or `task${n} PRR` inside xv6. You must first specify the scheduler during compilation with a make flag. Remember to run `make clean` before switching scheduler policies. For example,

```
root@1234567890ab:/home/xv6# make clean
root@1234567890ab:/home/xv6# make qemu SCHEDPOLICY=THREAD_SCHEDULER_HRRN
...
$ task1 HRRN

root@1234567890ab:/home/xv6# make clean
root@1234567890ab:/home/xv6# make qemu SCHEDPOLICY=THREAD_SCHEDULER_PRIORITY_RR
...
$ task1 PRR
```

5 Part II - Real Time Scheduling

In this part, you are required to implement two different schedulers (Earliest Deadline First (EDF) with Constant Bandwidth Server (CBS) scheduling and Deadline-Monotonic(DM) scheduling) for the real-time threading library we provided.

5.1 Deadline-Monotonic Scheduling

5.1.1 Specifications

- The `Deadline-Monotonic Scheduling` is a fixed priority based algorithm.Task with shortest deadline is assigned highest priority. It is a Preemptive Scheduling Algorithm which means if any task of higher priority comes, running task will be preempted and higher priority task is assigned to CPU. ID is used to break the tie.

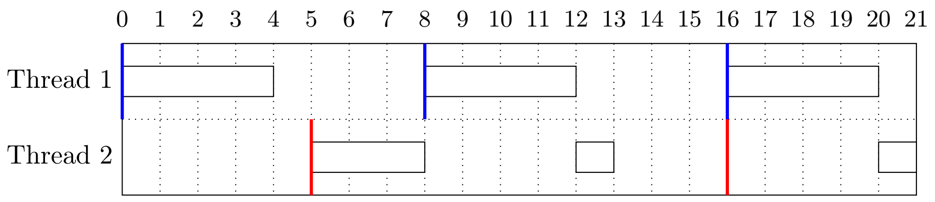

Figure 3: Deadline-Monotonic Scheduling with thread 1 ($t = 4$, $p = 8$) and thread 2 ($t = 4$, $p = 11$). Thread 1 arrives at tick 0 while thread 2 arrives at tick 5. Blue and red vertical lines represent the start of each cycle. Due to deadline (*Thread* 1) < deadline (*Thread* 2),So *Thread* 1 has higher priority. At tick 8, *thread* 2 still has 1 tick remaining , but *Thread* 1 has higher priority so *Thread* 1 gets CPU to execute.

- If there is a thread that has already missed its current deadline, set `scheduled_thread_list_member` to the `thread_list` member of that thread and set `allocated_time` to 0. When there are multiple threads missing their deadlines, choose the one with the smallest ID. Take Figure 4 for example. At tick 6 and 7, thread 4 and thread 3 miss their deadline. Since thread 3 has smaller ID, you should return thread 3. Note that you do not need to allocate fewer ticks to thread 2 at tick 5 just because thread 4 misses its deadline at tick 6. You only need to consider deadlines when (1) the thread you want to dispatch will miss its deadline when it is running (see Figure 5) (2) A thread in the run queue has missed its deadline.

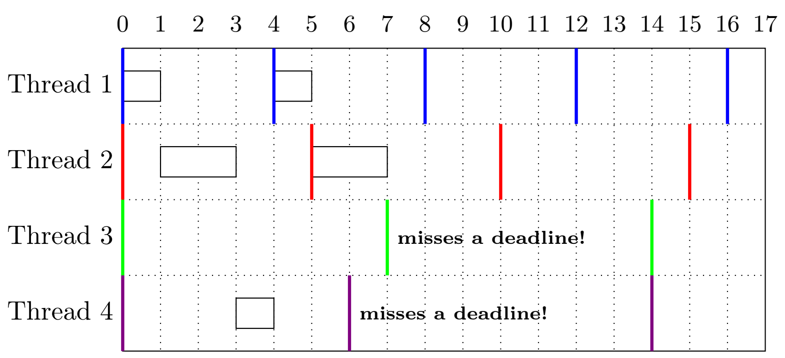

Figure 4: Deadline-Monotonic Scheduling with thread 1, ($t = 1$, $p = 4$) thread 2 ($t = 2$, $p = 5$), thread 3 ($t = 2$, $p = 7$) and thread 4 ($t = 2$, $p = 6$). All arrive at tick 0. Colored vertical lines represent the start of each cycle.

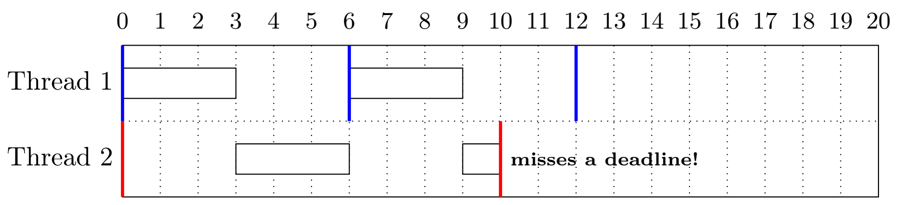

Figure 5: Deadline-Monotonic Scheduling with thread 1 ($t = 3$, $p = 6$) and thread 2 ($t = 5$, $p = 10$).

5.1.2 Functions to be Implemented

You have to implement the following function in `user/threads_sched.c`

```
struct threads_sched_result schedule_dm(struct threads_sched_args args)
```

5.1.3 Test Case Specifications

- The number of threads running concurrently < `MAX_THRD_NUM`.
- $0 <$ `processing_time` $\leq$ `period`
- $0 <$ `period` ≤ 100
- $0 <$ `n` ≤ 10
- $0 \leq$ `arrival time` ≤ 100

5.2 Earliest Deadline First (EDF) with Constant Bandwidth Server (CBS) scheduling

Figure 6 shows the behavior of an EDF with CBS (Constant Bandwidth Server) scheduler handling two periodic tasks. Task H1 is a hard real-time task, and task S1 is a soft real-time task with a CBS budget.

When a soft task is about to run, the CBS server checks if its remaining execution time would violate the bandwidth constraint. This is done using the following condition:

$$\frac{\text{Remaining CBS Budget Time}}{\text{Time Until CBS Deadline}} > \frac{\text{CBS Budget Time}}{\text{Period}}$$

(Hint: use multiplication instead of division to avoid floating point operations)

If the condition holds, the soft task exceeds its reserved bandwidth and is given a new CBS deadline and a replenished budget. This prevents it from overrunning its share and preserves fairness and temporal isolation.

Task	Type	Arrival Time	Burst Time	Period	Deadline	CBS Budget	Times
H1	Hard	0	5	10	10	5	2
S1	Soft	2	5	8	8	5	2

- Tick 0: H1 is released and starts execution immediately since it has the earliest deadline (10ms).
- Tick 2: S1 is released with its CBS deadline also set to 10ms. **Although both H1 and S1 have the same deadline, the EDF scheduler applies a tie-breaker rule based on thread ID. In this system, hard real-time tasks are always assigned smaller IDs than soft tasks.** Therefore, H1 continues to run.
- Tick 5: H1 completes execution. The scheduler now considers S1. CBS checks whether S1 can execute without exceeding its bandwidth:

$$\begin{aligned}\text{Remaining Runtime} &= 5\\ \text{Time until CBS Deadline} &= 10 - 5 = 5\\ \text{CBS Bandwidth} &= \frac{5}{8}\\ \Rightarrow \frac{5}{5} &> \frac{5}{8} \quad \text{(Condition True)}\end{aligned}$$

  ⇒ Since the condition is true, CBS postpones S1's deadline and replenishes its budget.

$$\text{New CBS Deadline} = \text{Current Time} + \text{Period} = 5 + 8 = 13$$

- Tick 5–10: S1 executes using its replenished CBS budget and completes its first instance at Tick 10.
- Tick 10: The second instance of H1 is released with a new deadline of 20ms.
- Tick 10–13: H1 begins executing its second instance. At Tick 13, its execution is paused to check for newly released soft tasks.
- Tick 13: CBS deadline of S1 expires, so the second instance of S1 is now released. CBS assigns it a new deadline and full budget:

$$\text{S1 Deadline} = 13 + 8 = 21$$

- Both H1 and S1 are now ready. The EDF scheduler compares deadlines:

$$\text{H1 Deadline} = 20, \quad \text{S1 Deadline} = 21$$

  Since H1 has the earlier deadline, it is selected.
- Tick 15: H1 completes its second instance. The scheduler now considers S1. CBS must check whether S1 can resume without violating the budget constraint:

$$\begin{aligned}\text{Remaining Runtime} &= 5\\ \text{Time until CBS Deadline} &= 21 - 15 = 6\\ \text{CBS Bandwidth} &= \frac{5}{8}\\ \Rightarrow \frac{5}{6} &> \frac{5}{8} \quad \text{(Condition True)}\end{aligned}$$

  ⇒ Since the condition is true, CBS postpones S1's deadline again and replenishes its budget:

$$\text{New CBS Deadline} = 15 + 8 = 23$$

- Tick 15–20: S1 runs using its replenished budget and completes its second instance at Tick 23.

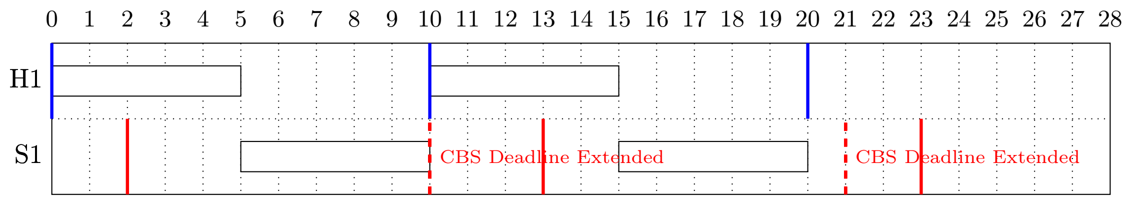

Figure 6: EDF with CBS Scheduling Timeline.

CBS Throttling and Replenishment Hint

- When a task's `remaining_budget` ≤ 0 and `remaining_time` > 0, the task is considered **throttled**.
- A throttled task cannot be scheduled again until its `current_deadline` is reached.
- At `current_time == current_deadline`, the CBS performs the replenishment:

$$\texttt{current\_deadline} \leftarrow \texttt{current\_deadline} + \texttt{period}$$
$$\texttt{remaining\_budget} \leftarrow \texttt{CBS\_budget}$$

- Be sure to check `is_throttled == 1` and `current_time == current_deadline` before performing the replenishment.

5.2.1 Functions to be Implemented

You have to implement the following function in `user/threads_sched.c`

```
struct threads_sched_result schedule_edf_cbs(struct threads_sched_args args)
```

5.2.2 Test Case Specifications

- $0 <$ `burst_time` $\leq$ `period`
- `burst_time` $< 2 \times$ `CBS_Budget`
- $0 <$ `period = deadline` ≤ 100
- $0 <$ `n` ≤ 5
- $0 \leq$ `arrival time` ≤ 100
- `Number of tasks` ≤ 5

5.3 Run the Public Tests

You can run the following command inside the docker (not in xv6).

```
root@ffffffffffff:/home/xv6# python3 grade-mp3-DM.py
== Test task1 == task1: OK (4.4s)
== Test task2 == task2: OK (2.3s)
== Test task3 == task3: OK (2.4s)
== Test task4 == task4: OK (2.8s)
== Test task5 == task5: OK (2.4s)
Score: 10/10
```

You can also manually run `rttask${n} DM` or `task${n} EDF_CBS` inside `xv6`. You must first specify the scheduler during compilation with a `make` flag. Remember to run `make clean` before switching scheduler policies. For example,

```
root@1234567890ab:/home/xv6# make clean
root@1234567890ab:/home/xv6# make qemu SCHEDPOLICY=THREAD_SCHEDULER_DM
...
$ rttask1 DM

root@1234567890ab:/home/xv6# make clean
root@1234567890ab:/home/xv6# make qemu SCHEDPOLICY=THREAD_SCHEDULER_EDF_CBS
...
$ rttask1 EDF_CBS
```

6 Submission and Grading

Run this command to pack your code into a zip named in your lowercase student ID, for example, d10922013.zip. Submit this file to Machine Problem 3 section on NTUCOOL.

```
make STUDENT_ID=d10922013 zip  # set your ID here
```

6.1 Grading Policy

- There are public test cases and private test cases.
  - Part 1 public test cases: 20%, 10% per algorithm.
  - Part 1 private test cases: 30%, 15% per algorithm.
  - Part 2 public test cases: 20%, 10% per algorithm.
  - Part 2 private test cases: 30%, 15%, per algorithm.
- You will get 0 point if we cannot compile your submission.
- You will get 0 point if we cannot unzip your file through the command line using the unzip command in Linux.
- You will get 0 point if the folder structure is wrong. Using uppercase in the <student id> is also a type of wrong folder structure.
- If your submission is late for n days, your score will be max(raw_score $- 20 \cdot \lceil n \rceil,\ 0$) points. Note that you will not get any points if $\lceil n \rceil \geq 5$.
- Our grading library has a timeout mechanism so that we can handle the submission that will run forever. Currently, the execution time limit is set to 600 seconds. We may extend the execution time limit if we find that such a time limit is not sufficient for programs written correctly.
- NOT plagiarize or get zero points, if you use any other resource, you should cite it in the reference.
- Do NOT share codes or prediction files with any living creatures
- You can submit your work as many times as you want, but only the last submission will be graded. Previous submissions will be ignored.