Machine Problem 3 - Scheduling

CSIE3310 - Operating Systems
National Taiwan University

Total Points:	100
Release Date:	April 23 Blue Correction: April 24 Red Correction: April 25
Due Date:	May 06, 23:59:00
TA e-mail:	`ntuos@googlegroups.com`
TA hours:	Mon., Thu., 15:30-16:30 at CSIE R439

Contents

1 Overview

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

1. Preemptive scheduling algorithms including Weighted-Round-Robin (WRR) scheduling and Shortest-Job-First(SJF) scheduling.
2. Real-time scheduling algorithms with periodic tasks, including Least-Slack-Time(LST) 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:

- **Threads**: The *thread_create* function takes 6 arguments, you can ignore the first two. *is_real_time* is set to 1 to indicate that the thread is real-time, otherwise it is non-real-time. For real-time thread, it should be executed for *processing_time* ticks in every *period* ticks for n cycles. The deadline is the same as the period. For non-real-time thread, *period* will always be set to -1 and n will always be set to 1. The API for creating a thread is as follows:

```
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**: A thread can be assigned an *arrival time* by the following function:

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

  which means its first cycle starts at `arrival_time` ticks.

- **Weight**: Weight is set to 1 by default. A weight can be assigned by the following function:

```
void thread_set_weight(struct thread *t, int weight);
```

  Note that only WRR will use the *weight* variable.

- **Data Structures**:
  - struct thread

```
struct thread {
    ...
    // a unique ID
    int ID;
    // 1 if real-time, 0 if non-real-time
    int is_real_time;
    // the processing time of the thread
    int processing_time;
    // weight for WRR
    int weight;
    // the deadline of the thread
    int deadline;
    // the period of the thread
    int period;
    // In real-time, the thread will be released n times
    int n;
    // number of ticks to be allocated in the current period
    int remaining_time;
    // the deadline of the current period
    int current_deadline;
};
```

  - struct release_queue_entry

```
struct release_queue_entry {
    struct thread *thrd;
    // for linked list
    struct list_head thread_list;
    // the time when `thrd` should be released to run queue, measured in ticks
    int release_time;
};
```

  - struct threads_sched_args

```
struct threads_sched_args {
    // the number of ticks since threading starts
    int current_time;
    // the linked list containing all the threads available to be run
    struct list_head *run_queue;
    // the linked list containing all the threads that will be available later
    struct list_head *release_queue;
};
```

  - struct threads_sched_result

```
struct threads_sched_result {
    // `scheduled_thread_list_member` should point to the `thread_list` member
    // of the scheduled `struct thread` entry
    struct list_head *scheduled_thread_list_member;
    // the number of ticks allocated for this thread to run
    int allocated_time;
};
```

- **Run Queue and Release Queue**: In our library, we put all the incomplete threads in two different queues: the *run queue* and the *release queue*. The run queue contains the threads that have not finished the current cycle (i.e. its `remaining_time` > 0). The release queue contains the thread which have finished the current cycle and are waiting for their next cycle to start (i.e. its `release_time` is greater than current time). Thus, the scheduler's job is to choose a thread in the run queue and allocate a proper number of ticks for it to run. In our implementation, both queues are implemented as *circular doubly linked list*.

4 Part I - Preemptive Scheduling

Recall that we have implemented a user-level threading library in MP1. One of its limitations is that the threads are *non−preemptive*. Preemptive scheduling involves the operating system interrupting a currently running thread to allocate CPU time to another thread, based on predefined criteria such as priority or time slices. Your task is to implement two preemptive scheduling algorithms, weighted round-robin scheduling and shortest-job-first scheduling.

4.1 Weighted Round-Robin Scheduling

Weighted Round-Robin (WRR) scheduling algorithm is an extension of the traditional Round-Robin scheduling algorithm. Each process is assigned a weight that determines the proportion of CPU time it receives relative to other processes. Processes with higher weights get more CPU time compared to those with lower weights. The scheduling algorithm cycles through the processes in the queue, allocating CPU time slices based on their weights. If more than one processes in the queue, ID is used to break the tie.

4.1.1 Specifications

Figure 1 is an example of weighted round-robin scheduling algorithm with a time quantum of 2. Two threads, Thread 1 and Thread 2. Thread 1 has a burst time of 5 and a weight of 2, while Thread 2 has a burst time of 3 and a weight of 1. Their arrival time is at tick 0.

- Tick 0: Both threads arrive. Thread 1 is assigned the CPU for the initial 2 ticks since ~~it has a higher weight~~ ID break the tie.
- Tick 2: Thread 1 gets assigned the CPU again since it has a weight of 2, it's allocated 2 ticks.
- Tick 4: Thread 2 gets assigned the CPU due to the round-robin. It gets 2 ticks.
- Tick 6: Thread 1's remaining burst time is 1. Therefore thread 1 is only assigned for 1 tick.
- Tick 7: Thread 2's remaining burst time is 1. Therefore thread 2 is only assigned for 1 tick.

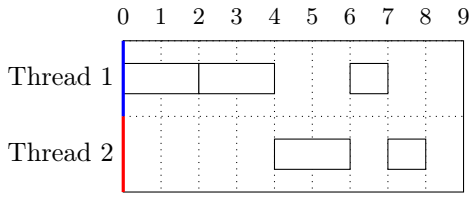

Figure 1: Weighted Round-Robin Scheduling with $time\ quantum = 2$, and thread 1 ($burst\ time = 5$, $weight = 2$) and thread 2 ($burst\ time = 3$, $weight = 1$). Both arrive at tick 0.

4.1.2 Functions to be Implemented

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

```
struct threads_sched_result schedule_wrr(struct threads_sched_args args)
```

4.1.3 Test Case Specifications

- $Time\ quantum = 2$
- $0 < process\ time \leq 64$
- $1 \leq weight \leq 4$
- $period = -1$
- $n = 1$
- $0 \leq arrival\ time \leq 100$

4.2 Shortest-Job-First Scheduling

Shortest-Job-First Scheduling algorithm which will select the process with the shortest expected processing time to execute next. The CPU will be preempted the process which has the smallest next CPU burst. If the next CPU burst of two processes are the same, ~~FCFS scheduling~~ `ID` is used to break the tie.

4.2.1 Specifications

As an example of SJF scheduling , consider the following set of threads, with the length of the CPU burst given in milliseconds.

Thread	Arrival Time	Burst Time
T1	0	6
T2	1	2
T3	2	8
T4	3	4

If the threads arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the Figure 2

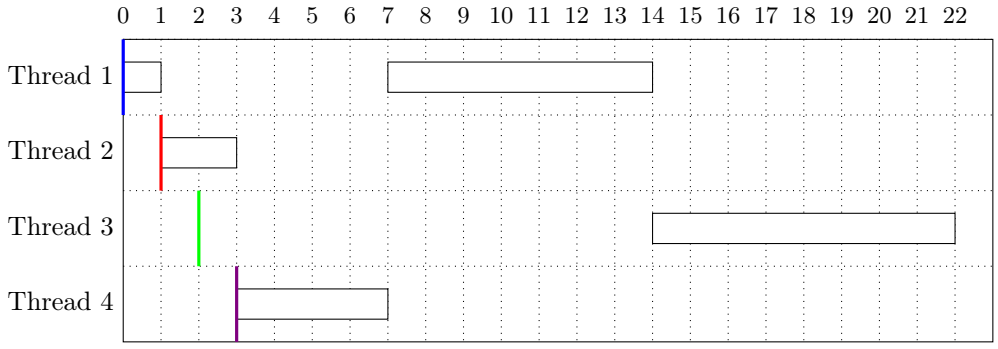

Figure 2: Shortest-Job-First Scheduling Gantt Chart

4.2.2 Functions to be Implemented

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

```
struct threads_sched_result schedule_sjf(struct threads_sched_args args)
```

4.2.3 Test Case Specifications

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

4.3 Run the Public Tests

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

```
root@fffffffffff:/home/xv6# python3 grade-mp3-WRR.py
...
== Test task1 == task1: OK (2.0s)
== Test task2 == task2: OK (1.8s)
== Test task3 == task3: OK (2.4s)
== Test task4 == task4: OK (1.7s)
Score: 8/8
...
```

You can also manually run `task${n}` in xv6. You can specify the scheduler by supplying a command line argument when running `make qemu`. Remember to run `make clean` before you recompile. For example,

```
root@1234567890ab:/home/xv6# make clean
root@1234567890ab:/home/xv6# make qemu SCHEDPOLICY=THREAD_SCHEDULER_WRR
...
$ task1
dispatch thread#1 at 0: allocated_time=2
dispatch thread#1 at 2: allocated_time=1
thread#1 finish at 3
```

5 Part II - Real Time Scheduling

In this part, you are required to implement two different schedulers (*Least-slack-time-First Scheduler* and *Deadline-Monotonic Scheduler*) for the real-time threading library we provided.

5.1 Least-Slack-Time Scheduling

Least-Slack-Time (LST) scheduling is a method that prioritizes tasks based on how close they are to their deadlines, executing those with the least time remaining first. It's useful for real-time systems where meeting deadlines is crucial. A slack time is defined by *current deadline − currnet time − remaining time*. ~~FCFS scheduling~~ `ID` is used to break the tie.

5.1.1 Specifications

- Take Figure 3 for example.
  - At tick 2, the scheduler should allocate 3 ticks to thread 2. Although thread 1 will be released at tick 4, it is unnecessary to preempt the execution at that time since thread 2 still has higher priority at tick 4.
  - At tick 15, the scheduler dispatches thread 1. However, since it will miss its deadline at tick 16, the scheduler should allocate just 1 tick to it. After it runs for 1 tick, our threading library will detect the deadline miss and terminate the process.

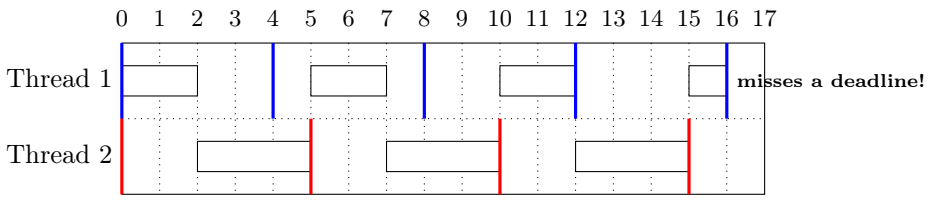

Figure 3: Least-Slack-Time Scheduling with thread 1 ($t = 2$, $p = 4$) and thread 2 ($t = 3$, $p = 5$). Both arrive at tick 0. Blue and red vertical lines represent the start of each cycle.

- Please regard different cycles of the same thread as separate scheduling units. Take Figure 4 for example. Although thread 1 is always running, you should allocate only 3 ticks to it at a time.

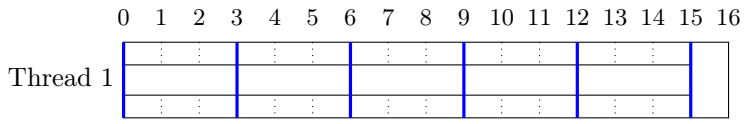

Figure 4: Least-Slack-Time Scheduling with only thread 1 ($t = 3$, $p = 3$) arriving at tick 0. Blue vertical lines represent the start of each cycle.

- If the run queue is empty, set `scheduled_thread_list_member` to `run_queue` and set `allocated_time` to the number of ticks the scheduler should wait until the next thread is released. Take Figure 5 for example. At tick 4, there is no available thread and the next thread (thread 2) is released at tick 5, so the allocated sleep time should be 1.

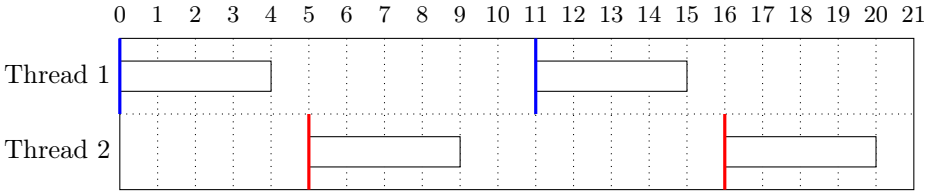

Figure 5: Least-Slack-Time Scheduling with thread 1 ($t = 4$, $p = 11$) and thread 2 ($t = 4$, $p = 11$). Thread 1 arrives at tick 0 while thread 2 arrives at tick 5. Blue and red vertical lines represent the start of each cycle.

5.1.2 Functions to be Implemented

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

```
struct threads_sched_result schedule_lst(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 Deadline-Monotonic Scheduling

5.2.1 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. ~~FCFS scheduling~~ `ID` is used to break the tie.

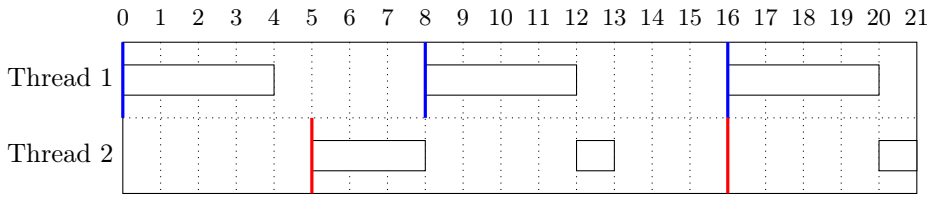

Figure 6: Deadline-Monotonic Scheduling with thread 1 ($t = 4$, $p = 8$) and thread 2 ($t = 4$, $p = 11$). Thread 1 arrives at tick 0 while thread 2 arrives at tick 5. Blue and red vertical lines represent the start of each cycle. Due to deadline (*Thread* 1) < deadline (*Thread* 2),So *Thread* 1 has 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 7 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 3) (2) A thread in the run queue has missed its deadline.

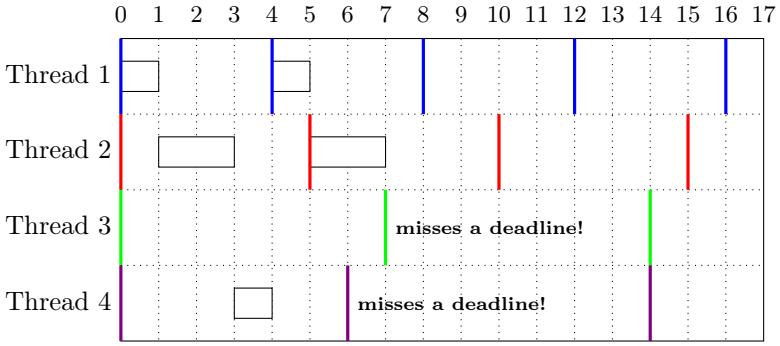

Figure 7: Deadline-Monotonic Scheduling with thread 1, ($t = 1$, $p = 4$) thread 2 ($t = 2$, $p = 5$), thread 3 ($t = 2$, $p = 7$) and thread 4 ($t = 2$, $p = 6$). All arrive at tick 0. Colored vertical lines represent the start of each cycle.

5.2.2 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)
```

5.2.3 Test Case Specifications

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

5.3 Run the Public Tests

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

```
root@fffffffffff:/home/xv6# python3 grade-mp3-LST.py
...
== Test rttask1 == rttask1: OK (4.0s)
== Test rttask2 == rttask2: OK (3.1s)
== Test rttask3 == rttask3: OK (2.9s)
== Test rttask4 == rttask4: OK (2.6s)
Score: 12/12
...
```

You can also manually run `rttask${n}` in xv6. You can specify the scheduler by supplying a command line argument when running `make qemu`. Remember to run `make clean` before you recompile. For example,

```
root@1234567890ab:/home/xv6# make clean
root@1234567890ab:/home/xv6# make qemu SCHEDPOLICY=THREAD_SCHEDULER_LST
...
$ rttask1
dispatch thread#1 at 0: allocated_time=3
thread#1 finish one cycle at 3: 2 cycles left
run_queue is empty, sleep for 2 ticks
dispatch thread#1 at 5: allocated_time=3
thread#1 finish one cycle at 8: 1 cycles left
run_queue is empty, sleep for 2 ticks
dispatch thread#1 at 10: allocated_time=3
thread#1 finish one cycle at 13: 0 cycles left
```

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 2 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: 16%, 8% per algorithm.
  - Part 1 private test cases: 24%, 12% per algorithm.
  - Part 2 public test cases: 24%, 12% per algorithm.
  - Part 2 private test cases: 36%, 18%, 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 $\max(\text{raw\_score} - 20 \cdot \lceil n \rceil,\ 0)$ 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.
- You can submit your work as many times as you want, but only the last submission will be graded. Previous submissions will be ignored.