Machine Problem 3 - Scheduling

CSIE3310 - Operating Systems
National Taiwan University

Total Points:	100
Release Date:	April 25
Due Date:	May 08, 23:59:00
TA e-mail:	`ntuos@googlegroups.com`
TA hours:	04/28 11:00-12:00, 05/03 14:00-15:00

Contents

1 Overview **2**

2 Environment Setup **2**

3 Concepts **2**
- 3.1 Timer Interrupts and Ticks . . . 2
- 3.2 Traps . . . 2

4 Part I - System Calls **2**
- 4.1 thrdstop . . . 3
  - 4.1.1 Description . . . 3
  - 4.1.2 Hints for Implementation . . . 3
  - 4.1.3 Test Case Specifications . . . 3
- 4.2 thrdresume . . . 4
  - 4.2.1 Description . . . 4
  - 4.2.2 Hints for Implementation . . . 4
  - 4.2.3 Test Case Specifications . . . 4
  - 4.2.4 Example Usage . . . 4
- 4.3 cancelthrdstop . . . 4
  - 4.3.1 Description . . . 5
  - 4.3.2 Test Case Specifications . . . 5
  - 4.3.3 Example Usage . . . 5
- 4.4 Run the Public Tests . . . 5

5 Part II - Scheduler **6**
- 5.1 Real-Time Threading Library . . . 6
- 5.2 Scheduler Description . . . 7
  - 5.2.1 Functions to be Implemented . . . 7
  - 5.2.2 Specifications . . . 8
  - 5.2.3 Test Case Specifications . . . 9
- 5.3 Run the Public Tests . . . 9

6 Submission and Grading **10**
- 6.1 Folder Structure after Unzip . . . 10
- 6.2 Grading Policy . . . 10

1 Overview

The main tasks of this MP are as follows:

1. Exploit xv6's timer interrupts to enable preemptive user-level threading.
2. Implement several real-time scheduling algorithms with periodic tasks.

Recall that we have implemented an user-level threading library in MP1. One of its limitations is that the threads are *nonpreemptive*. That is, explicitly calling `thread_yield` is the only way to transfer control to the scheduler. To support *preemptive* scheduling, you need to exploit xv6's *timer interrupts*, which is a hardware-enabled mechanism for executing a specific routine at fixed timed intervals. Specifically, you need to modify the codes for handling timer interrupts to maintain your own timer. When your timer goes off, it preempts the current control in the user space and call the scheduler. To be able to manipulate this timer from the user space, you also need to implement several system calls. This is the first part of this MP.

For the second part, we provided a real-time threading library that depends on the aforementioned system calls. Your job is to implement two real-time scheduling algorithms taught in the class: *Earliest-Deadline-First Scheduling* and *Rate-Monotonic Scheduling*. Your scheduling algorithms should not produce unnecessary scheduling and dispatching overhead by allocating a proper amount of time to each thread.

2 Environment Setup

1. Unzip MP3.zip in your working directory.
2. Pull docker image from Docker Hub

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

3. Run the container:

```
$ docker run -it -v $(pwd)/xv6-riscv:/home/xv6-riscv/ -w /home/xv6-riscv/ \
           --name osmp3 ntuos/mp3 /bin/bash
```

3 Concepts

3.1 Timer Interrupts and Ticks

In xv6, timer interrupts are generated periodically by the clock hardware attached to each RISC-V CPU. Xv6 uses this mechanism to maintain its software clock and perform process scheduling. In this MP, we will refer to the time between two subsequent timer interrupts as **1 tick**. In function `timerinit` in `kernel/start.c`, we can see how xv6 configures the length of 1 tick, which is 1/10 seconds currently.

3.2 Traps

Trap is a mechanism that allows the system to transfer control to a specific routine when a particular event or condition occurs, such as an exception, a system call, or an interrupt. In xv6, all traps are handled in the kernel, and you can find the two trap handlers: `usertrap` and `kerneltrap` in `kernel/trap.c`. They handle traps from user space and kernel space respectively. When a trap occurs, the kernel needs to properly store and restore the values of the registers such that the trap is "transparent" to the current process. This also means we can change the values of the registers to intervene the execution. For example, we can change the program counter to redirect the control flow. It is recommended that you read Chapter 4 of the xv6 book before you start.

4 Part I - System Calls

The function bodies of the following system calls should be implemented in `kernel/thrd.c`. However, to make them work, you also need to modify `kernel/trap.c`, `kernel/proc.h` and `kernel/proc.c`.

4.1 thrdstop

```c
int thrdstop(int delay, int *context_id_ptr, void (*handler)(void *),
             void *handler_arg);
```

4.1.1 Description

- `handler` should be called with an argument `handler_arg` after the process consumes `delay` ticks since the `thrdstop` call.
- Before executing `handler`, the current context (i.e. user registers) should be stored in the kernel space so that we can jump back later.
- Let the variable pointed by `context_id_ptr` be `context_id`. Depending on the value of `context_id`, it should take different actions:
  - If `context_id` is -1: Assign a new ID to `context_id` which is used to identify the stored context. If all the available IDs have been assigned, return -1 to indicate error.
  - If `context_id` is a value assigned by `thrdstop` previously: Directly use the memory identified by this ID to store the context. Do not modify `context_id`.
  - Otherwise: return -1 to indicate error.

  The maximum number of contexts that can be stored at the same time should be `MAX_THRD_NUM` defined in `kernel/proc.h`.
- Return value: 0 on success, -1 on error.
- It is undefined behavior to call another `thrdstop` before the handler registered by the previous `thrdstop` gets executed. For example, the following codes

  ```c
  thrdstop(10, &id1, handler1, (void *)0);
  thrdstop(10, &id2, handler2, (void *)0);
  ```

  will not occur. You only need to maintain one timer at a time.

Note that the process may switch to other contexts when executing `handler` and call `thrdstop` with another ID. The purpose of assigning an ID is to distinguish different contexts to resume.

4.1.2 Hints for Implementation

- You need to set up a timer for counting the number of ticks consumed by the process after a `thrdstop` call. Therefore, you may want to add some attributes to `struct proc` defined in `kernel/proc.h`. You may initialize your attributes in function `allocproc` defined in `kernel/proc.c`.
- You may want to maintain your timer in the codes for handling timer interrupts. You can find them in functions `usertrap` and `kerneltrap` defined in `kernel/trap.c`.
- Find out which member of `struct proc` is used to store the user registers. You can read Chapter 4 of xv6 book or trace `usertrapret` in `kernel/trap.c` and `userret` in `kernel/trampoline.S`.
- For supplying the argument to the handler, you need to know the calling convention of RISC-V.
- `context_id_ptr` is a pointer in the user address space. Thus, you cannot just dereference it using `*` operator in the kernel space. See `copyin` and `copyout` in `kernel/vm.c`.

4.1.3 Test Case Specifications

In the test cases, the following constraints on the arguments are always satisfied:

- `delay` > 0.
- `context_id_ptr` is a readable/writable address in the current process.
- `handler` is a function in the current process.

4.2 thrdresume

```
int thrdresume(int context_id);
```

4.2.1 Description

- When returning to user space, it should restore the context specified by `context_id`. It is similar to `longjmp` in MP1.
- If `context_id` is invalid (not registered or out of range), return −1 to indicate error.
- Return value: 0 on success, -1 on error.

After the handler registered by `thrdstop` is executed, the original context should be stored in the kernel space. Thus, it is necessary to make another system call to restore the context. Think about what will happen if you just `return` in the handler.

4.2.2 Hints for Implementation

- This is very similar to redirecting the execution to the `handler` in `thrdstop`, except that now we need to set up all of the user registers, not just the program counter.

4.2.3 Test Case Specifications

In the test cases, the following constraints on the argument are always satisfied:

- `context_id` is an ID assigned by `thrdstop` previously.

4.2.4 Example Usage

```c
int v = 0;
int main_id = -1;

void handler(void *arg){
    v = 99999;
    thrdresume(main_id);
}

int main()
{
    thrdstop(3, &main_id, handler, (void *)0);
    while (v == 0) {
        // zzz...
    }
    printf("v = %d\n", v);
    exit(0);
}
/*
The output is:
v = 99999
*/
```

4.3 cancelthrdstop

```
int cancelthrdstop(int context_id, int is_exit);
```

4.3.1 Description

- This function cancels the `thrdstop`. For example, in the following codes,

  ```
  thrdstop(1000, &id, handler, (void *)0);
  cancelthrdstop(id, 0);
  ```

  `handler` will not be called. Note that the `thrdstop` should be canceled no matter what the `context_id` is (e.g. the `context_id` can be different from the previous `thrdstop` call).

- The system call should takes different actions depending on the value of `is_exit`:
  - If `is_exit` is 0: It stores the current context according to the `context_id`.
  - If `is_exit` is 1: Do not store the current context. Instead, release the stored context specified by `context_id` and recycle this ID for other uses.

- The return value is the number of ticks consumed by the process since `thrdstop` is called. If the timer is inactive now (i.e. the handler has been executed), return the number of ticks consumed by the previous timer.

4.3.2 Test Case Specifications

In the test cases, the following constraints on the argument are always satisfied:

- `context_id` is an ID assigned by `thrdstop` previously.

4.3.3 Example Usage

```
int main_id = -1;
int is_handler_executed = 0;
void handler(void *arg){
    is_handler_executed = 1;
    thrdresume(main_id);
}
int main(){
    thrdstop(10, &main_id, handler, (void *)0);
    while(!is_handler_executed){
        /* wait for handler */
    }
    int t = cancelthrdstop(main_id, 0);
    printf("%d\n", t);
    thrdstop(10000, &main_id, handler, (void *)0);
    int start_time = uptime();
    while(uptime() - start_time < 3){
        /* wait for 3 ticks */
    }
    t = cancelthrdstop(main_id, 0);
    printf("%d\n", t);
    exit(0);
}
/*
The output is:
10
3
*/
```

4.4 Run the Public Tests

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

```
root@ffffffffffff:/home/xv6-riscv# python3 grade-mp3-syscalls.py
...
== Test thrdtest1 == thrdtest1: OK (43.4s)
== Test thrdtest2 == thrdtest2: OK (10.7s)
== Test thrdtest3 == thrdtest3: OK (9.7s)
== Test thrdtest4 == thrdtest4: OK (10.0s)
Score: 16/16
...
```

You can also manually run `thrdtest${n}` in xv6.

5 Part II - Scheduler

In this part, you are required to implement two different schedulers (*Earliest-Deadline-First Scheduler* and *Rate-Monotonic Scheduler*) for the real-time threading library we provided. In the following, we will explain our implementation and its interface to the schedulers.

5.1 Real-Time Threading Library

- **Threads**: Each thread is a *periodic task* specified by three values: t(*processing time*), p(*period*) and n(*number of cycles*), which means it should be executed for t ticks every p ticks for n cycles. The deadline is the same as the period p. The API for creating a thread is as follows:

  ```
  struct thread *thread_create(void (*f)(void *), void *arg,
                               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.

- **Data Types**:
  - The members unrelated to scheduling is omitted.

    ```
    struct thread {
        ...
        int ID;                         // a unique ID, mainly used for tie-breaking
        struct list_head thread_list; // for linked list
        ...
        int processing_time;  // see Section 5.1
        int period;           // see Section 5.1
        int n;                // see Section 5.1

        // number of ticks to be allocated in the current cycle
        int remaining_time;
        // the deadline of the current cycle, measured in ticks
        int current_deadline;
    };
    ```

  - This struct simply wraps a pointer to `struct thread` with a release time and a struct for linked list.

```c
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;
};
```

- **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*.

5.2 Scheduler Description

The function signarue of a scheduler function is as follows:

```c
struct threads_sched_result schedule_edf(struct threads_sched_args args);
```

where the argument and the return value are defined as:

```c
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 {
    // `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;
};
```

The linked list type `struct list_head *` can be traversed using `list_for_each_entry` defined in `user/list.h`[1]. An example of usage can be found in function `scheduler_default` in `user/threads_sched.c`.

5.2.1 Functions to be Implemented

All of the following functions should be implemented in `user/threads_sched.c`.

- `struct threads_sched_result schedule_edf(struct threads_sched_args args);`
  The thread with the earliest deadline has the highest priority. If there is a tie, the one with a smaller ID wins.

- `struct threads_sched_result schedule_rm(struct threads_sched_args args);`
  The thread with the smallest period has the highest priority. If there is a tie, the one with a smaller ID wins.

[1] This library is ported from Linux kernel. It adopts an unique approach so that it can be embedded in any struct.

5.2.2 Specifications

- The returned `allocated_time` should be the **maximum** number of ticks that this thread can run until it **meets/misses the current deadline** or gets **preempted** by a higher priority thread. This is to minimize the scheduling and dispatching overhead. Take Figure 1 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.

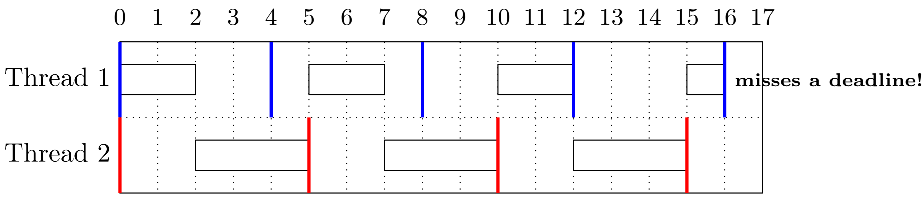

Figure 1: Earliest-Deadline-First Scheduling with thread 1 ($t = 2$, $p = 4$) and thread 2 ($t = 3$, $p = 5$). Both arrive at tick 0. Blue and red vertical lines represent the start of each cycle.

- Please regard different cycles of the same thread as separate scheduling units. Take Figure 2 for example. Although thread 1 is always running, you should allocate only 3 ticks to it at a time.

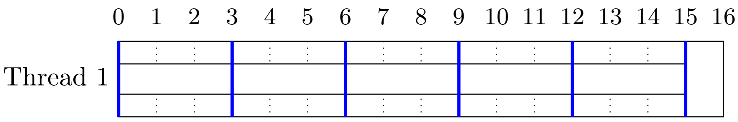

Figure 2: Earliest-Deadline-First Scheduling with only thread 1 ($t = 3$, $p = 3$) arriving at tick 0. Blue vertical lines represent the start of each cycle.

- 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 3 for example. At tick 4, there is no available thread and the next thread (thread 2) is released at tick 5, so the allocated time should be 1. While at tick 9, the allocated time should be 2.

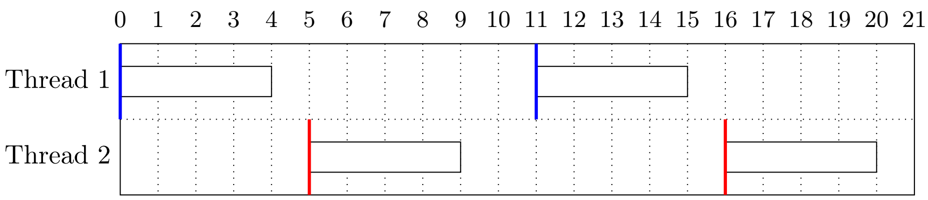

Figure 3: Earliest-Deadline-First Scheduling with thread 1 ($t = 4$, $p = 11$) and thread 2 ($t = 4$, $p = 11$). Thread 1 arrives at tick 0 while thread 2 arrives at tick 5. Blue and red vertical lines represent the start of each cycle.

- 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 7, both thread 3 and thread 4 have misses 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 1) (2) A thread in the run queue has missed its deadline.

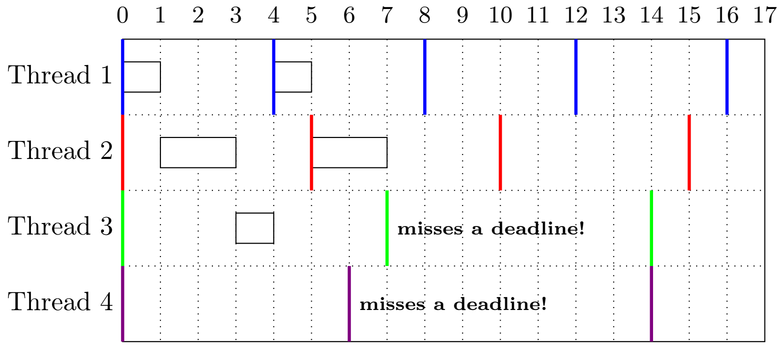

Figure 4: Rate-Monotonic Scheduling with thread 1, ($t = 1$, $p = 4$) thread 2 ($t = 2$, $p = 5$), thread 3 ($t = 2$, $p = 7$) and thread 4 ($t = 2$, $p = 6$). All arrive at tick 0. Colored vertical lines represent the start of each cycle.

5.2.3 Test Case Specifications

- The number of threads running concurrently < `MAX_THRD_NUM`.
- For every call of `thread_create`:
  - $0 <$ `processing_time` $\leq$ `period`
  - $0 <$ `period` ≤ 100
  - $0 <$ `n` ≤ 10
- For every call of `thread_add_at`, $0 \leq$ `arrival time` ≤ 100

5.3 Run the Public Tests

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-riscv# make clean
root@1234567890ab:/home/xv6-riscv# make qemu SCHEDPOLICY=THREAD_SCHEDULER_EDF
```

You can run the public tests by the python scripts `grade-mp3-EDF.py` and `grade-mp3-RM.py`. For example,

```
root@1234567890ab:/home/xv6-riscv# python3 grade-mp3-EDF.py
...
== Test task1 == task1: OK (39.9s)
== Test task2 == task2: OK (12.8s)
== Test task3 == task3: OK (10.0s)
== Test task4 == task4: OK (9.5s)
Score: 12/12
...
```

You can also run the user programs `task1`, `task2`, and `task3` in xv6 to see the actual output of the tests. For example,

```
$ task2  # SCHEDPOLICY=THREAD_SCHEDULER_EDF
dispatch thread#1 at 0: allocated_time=5
thread#1 finish one cycle at 5: 2 cycles left
dispatch thread#2 at 5: allocated_time=7
thread#2 finish one cycle at 12: 2 cycles left
dispatch thread#1 at 12: allocated_time=5
thread#1 finish one cycle at 17: 1 cycles left
dispatch thread#2 at 17: allocated_time=3
dispatch thread#1 at 20: allocated_time=5
thread#1 finish one cycle at 25: 0 cycles left
dispatch thread#2 at 25: allocated_time=4
thread#2 finish one cycle at 29: 1 cycles left
run_queue is empty, sleep for 3 ticks
dispatch thread#2 at 32: allocated_time=7
thread#2 finish one cycle at 39: 0 cycles left
```

6 Submission and Grading

Please organize `kernel/proc.h`, `kernel/proc.c`, `kernel/thrd.c`, `kernel/trap.c` and `user/threads_sched.c` into the following folder structure. Then compress them as <whatever>.zip and upload the zipped file to NTU COOL. The file name does not matter since NTU COOL will rename your submissions. Please do not compress any files we do not request (e.g. `*.o`, `*.d`) and make sure your codes can be compiled by `make qemu` with the Makefile provided by TA.

6.1 Folder Structure after Unzip

```
<student ID>
|
+-- kernel
|   |
|   +-- proc.h
|   |
|   +-- proc.c
|   |
|   +-- thrd.c
|   |
|   +-- trap.c
|
+-- user
    |
    +-- threads_sched.c
```

Note that all the English letters in the <student id> must be lowercase. E.g., it should be b12123a23 instead of B12123A23.

6.2 Grading Policy

- There are public test cases and private test cases.
  - public test cases: (a) syscalls, 16%. (b) two scheduling algorithms, 12% for each.
  - private test cases: (a) syscalls, 24%. (b) two scheduling algorithms, 18% for each.
- You will get 0 point if we cannot compile your submission.
- You will be deducted 10 points if we cannot unzip your file through the command line using the unzip command in Linux.

- You will be deducted 10 points 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.