Operating System 2021 Machine Problem Site

The site provides the information for machine problems for Operating System Course offered in Spring 2021 at National Taiwan University

OS mp3 Scheduling (Due 5/24 23:59. No late submission)

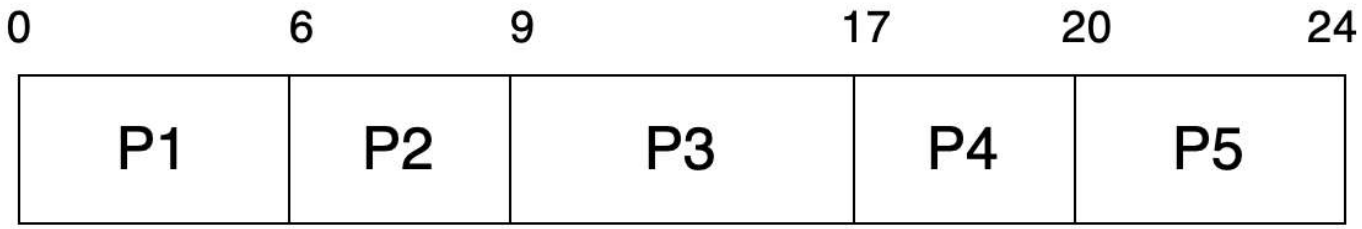
In this mp you will have to implement CPU scheduling with the algorithms below.

given processes

Process	CPU burst time	Arrival Time
P1	6	0
P2	3	1
P3	8	2
P4	3	3
P5	4	4

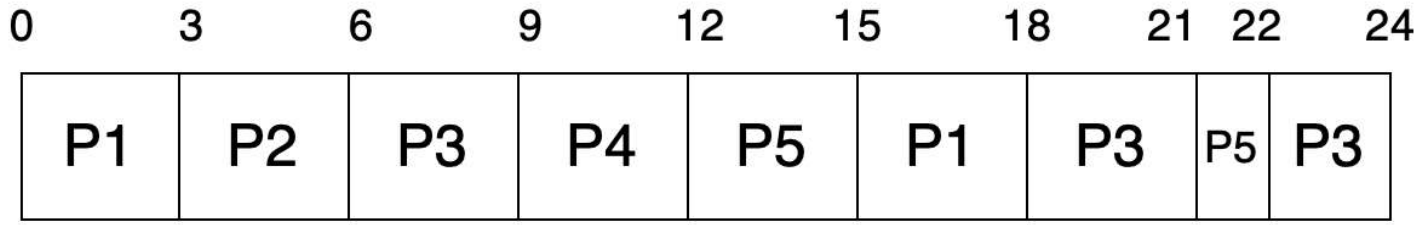
First Come First Served(FCFS):

FCFS executes queued requests and processes in order of their arrival. Thus the order of the execution of the above mentioned processes will be



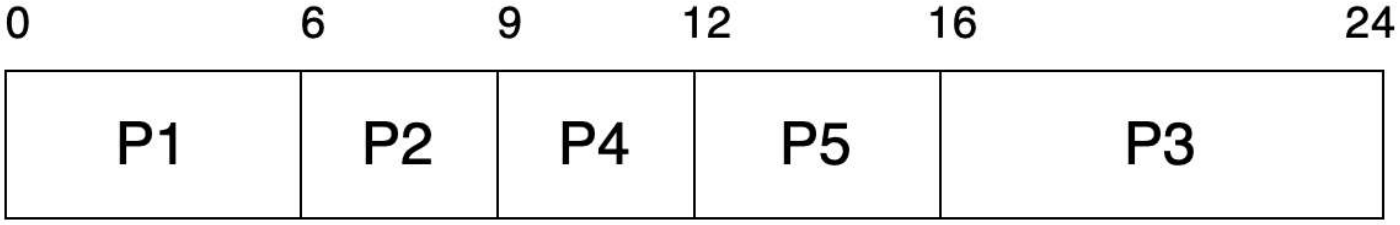
Round Robin(RR):

In Round-robin scheduling, each ready task runs turn by turn only in a cyclic queue for a limited time slice. This algorithm is preemptive also offers starvation free execution of processes. Thus assume time quantum for the example above is 3 the order of the execution of the above mentioned processes will be



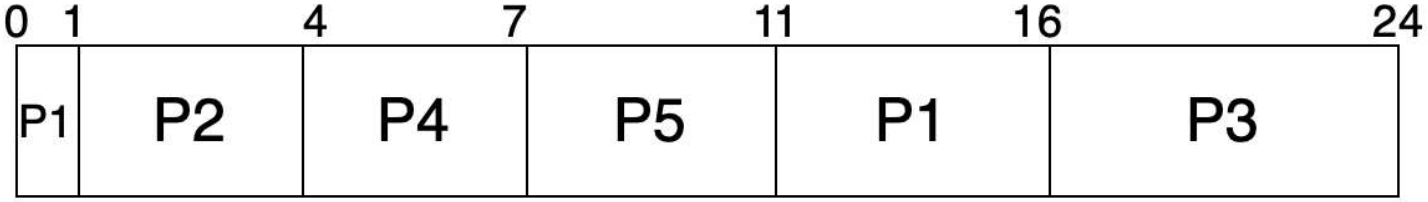
Non-Preemptive Shortest Job First(SJF):

Shortest Job First (SJF) is an algorithm in which the process having the smallest execution time is chosen for the next execution. In non-preemptive scheduling, once the CPU cycle is allocated to process, the process holds it till it reaches a waiting state or terminated.Thus the order of the execution of the above mentioned processes will be



Shortest Remaining Time First (Preemptive SJF)

In the Shortest Remaining Time First (SRTF) scheduling algorithm, the process with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progresses, processes will always run until they complete or a new process is added that requires a smaller amount of time. Thus the order of the execution of the above mentioned processes will be



Implementaion

How to run mp3

Download `xv6-riscv` directory. Pull docker image from Docker Hub

```
docker pull ntuos/mp3
```

Then, run the docker

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

You have finished a user-level thread package in mp1, but function `schedule` was implemented by TA. In this mp, you are going to implement the algorithms described above.

In mp1, we didn't consider the time. If we want to implement those algorithms, we should have some functions that can count down the time and interrupt threads. Therefore, you have to implement 3 syscalls to achieve those goals. Their usage will be described below.

After you finish the syscalls, you can move on to implement the scheduler. we will provide you `threads.c` and `threads.h`, every functions in `threads.c` will be implemented, and they will use the syscalls you implement. Since the schedule algorithm in our template is RR with time_quantum=1 time slot, you may modify `schedule()` for other algorithms. You can also modify other functions.

Syscalls

int thrdstop(int ticks, int thrdstop_context_id, void (*thrdstop_handler)());

Primary usage of this syscall is:

1. If a program calls `thrdstop()`, after `ticks` ticks that this program consumes, switch to execute the `thrdstop_handler`. After you switch to `thrdstop_handler` for the first time, you shouldn't switch to it for the second time. The effect of this syscall just for one time.
2. Store the current program context according to `thrdstop_context_id`. `thrdstop_context_id` and return value will be described below.

a. Invoke the handler

You'll need to keep track of how many ticks have passed since the program calls `thrdstop()`; you may also need some attributes in `struct proc` to do this. We have defined some attributes for you. You can find them in `/kernel/proc.h`, we have marked them by `// for mp3`.

- **int thrdstop_ticks;** It should record how many ticks have passed since the program.

- **int thrdstop_interval;** It should store when we should switch to thrd_handler. You can just store `ticks` , argument of thrdstop.
- **uint64 thrdstop_handler_pointer;** It should store where we need to switch, You can just store `thrdstop_handler` , argument of thrdstop.

Every tick passes, the hardware clock forces an interruption, which is handled by `usertrap()` and `kerneltrap()` in `kernel/trap.c` . You can find something like:

```
if(which_dev == 2)
...
```

You may do your jobs in that "if block".

b. Store the context

Because this syscall will have program switch to `thrdstop_handler` , you should store the current program context. We have defined some attributes of `struct proc` for you to do this job. You can find them in `/kernel/proc.h` , we have marked them by `// for mp3` .

- **struct thrd_context_data thrdstop_context[MAX_THRD_NUM]** You store the current context in this array, `struct thrd_context_data` is a bunch of `uint64` to store registers. You can find definition of `struct thrd_context_data` and `MAX_THRD_NUM` in `/kernel/proc.h`
- **int thrdstop_context_used[MAX_THRD_NUM];** Indicated that `thrdstop_context[i]` is occupied or not. They will all be `0` at first.
- **int thrdstop_context_id;** When switch occurs, you should store the current context in `thrdstop_context[thrdstop_context_id]` .

You need to find where is the current program context when you are in `if(which_dev == 2)` block. You can find the answer in xv6 book chapter4. To simplify the problem, when you are in the `kerneltrap()` , the location of current program context is the same as the location when you are in the `usertrap()` . Moreover, what you do in `usertrap()` , just copy to corresponding location in `kerneltrap()` .

The value of `thrdstop_context_id` is determined by `thrdstop()`

1. If we call `thrdstop(n, -1, handler)` , you should find an empty slot in `thrdstop_context[MAX_THRD_NUM]` . and store the empty slot index in `thrdstop_context_id` , then `thrdstop()` will return the empty slot index. Remember to set `thrdstop_context_used[index]` to `1` . If you can't find an empty slot, return -1;
2. If we call `thrdstop(n, i, handler)` , it means that we want to store the current context in `thrdstop_context[i]` , set `thrdstop_context_id` as `i` , then `thrdstop()` will return `i` .

c. Some guarantees

1. We only call one `thrdstop` at the same time. For example

   ```
   thrdstop(10, -1, handler1);
   thrdstop(10, -1, handler2);
   ```

   This won't happen, you only need to keep track on one countdown at the same time.

2. `int ticks` > 0, argument of `thrdstop()`
3. `thrdstop_handler` must be some function of the program.
4. If you want to, you can add more attributes in `struct proc` or remove our attributes for mp3. You can initialize your attributes in `allocproc()` in `/kernel/proc.c` .

int thrdresume(int thrdstop_context_id, int is_exit);

1. If `is_exit` is zero, reload the context stored in `thrdstop_context[thrdstop_context_id]` , continue to execute that context.
2. If `is_exit` isn't zero, then `thrdstop_context[thrdstop_context_id]` will become empty , and previous `thrdstop()` will be cancelled. For example:

   ```
   thrdstop(10, -1, handler2);
   thrdresume(10, 1);
   ```

   In this case, `handler2` won't be called, and `thrdstop_context_used[10]` should be set `0` . It is used for `thread_exit()`

3. Return any value you want, we don't use it.

Some guarantees

1. `0` <= `thrdstop_context_id` < `MAX_THRD_NUM`

int cancelthrdstop(int thrdstop_context_id);

This function cancels the `thrdstop()` . It also save the current thread context into `thrdstop_context[thrdstop_context_id]` , no need to store if thrdstop_context_id is -1.

The return value is the time counted down by `thrdstop()` . That is, return `proc->thrdstop_ticks`

Some guarantees

1. `0` <= `thrdstop_context_id` < `MAX_THRD_NUM` or `thrdstop_context_id == -1`

Test the syscalls

You can run `test1` in xv6. Find the output, if "PASS 1" and "PASS 2" exist, then you pass the test. If you pass `test1`, you probably get the score of syscall part.

Explanation of some functions in template code in thread.c

- **void thread_start_threading(int time_slot_size);** This function starts threading and determine `time_slot_size`. `time_slot_size` indicates that `my_thrdstop_handler` will be called every `time_slot_size` ticks. `time_slot_size` is also basic time unit. Time quantum in RR must be integer times of `time_slot_size` in mp3.
- **void my_thrdstop_handler(void);** This function updates and checks the `remain_execution_time` of thread. `my_thrdstop_handler()` also directly calls `schedule()` for switching to next thread.
- **struct thread *thread_create(void (*f)(void *), void *arg, int execution_time_slot);** This function creates a thread object and does some initializations. `execution_time_slot` indicates that maximum execution time of this thread is `time_slot_size * execution_time_slot` ticks. When start threading, `thread->remain_execution_time` will be set to `time_slot_size * execution_time_slot`. Threads may exit early, so its execution time can be less than `remain_execution_time`.

What you need to do for scheduler

- FCFS: Execute next thread when current thread exits.
- Round Robin:
  1. You should implement RR with time_quantum=3.
  2. Execute the next thread when (1) current thread exits. (determined by `thrd->is_exited`) (2) current thread yields. (determined by `thrd->is_yield`) (3) current thread has used up its time_quantum.
- SJF:
  1. Choose the next thread based on `remain_execution_time` (the shortest one). If `remain_execution_time` is the same, choose the thread arived earlier. The arrival order is determined by `thread->ID` (the smaller the earlier).
  2. Execute the next thread when (1) current thread exits. (2) current thread yields.

UPDATE: Choose the first thread based on remain_execution_time.

- Preemptive SJF:

1. Choose the next thread based on `remain_execution_time` (the shortest one). If `remain_execution_time` is the same, choose the thread arived earlier. The arrival order is determined by `thread->ID` (the smaller the earlier).
2. Execute the next thread when (1) current thread exits. (2) current thread yields. (3) current thread has been executed `time_slot_size` ticks.

UPDATE: Choose the first thread based on remain_execution_time.

Test the scheduler

You can test your scheduler by those python files.

```
python3 grade-mp3-...
```

There're 5 python files.

```
grade-mp3-FCFS  grade-mp3-RR     grade-mp3-default
grade-mp3-PSJF  grade-mp3-SJF
```

The test tasks run by those python files are task1, task2 and task3. You can also run task1~3 in xv6.

UPDATE: grade-mp3-default is for the scheduler of sample code (RR with TQ==1) .

Explnation of task1

`time_slot_size` is 5 ticks. t1 will yield at first.

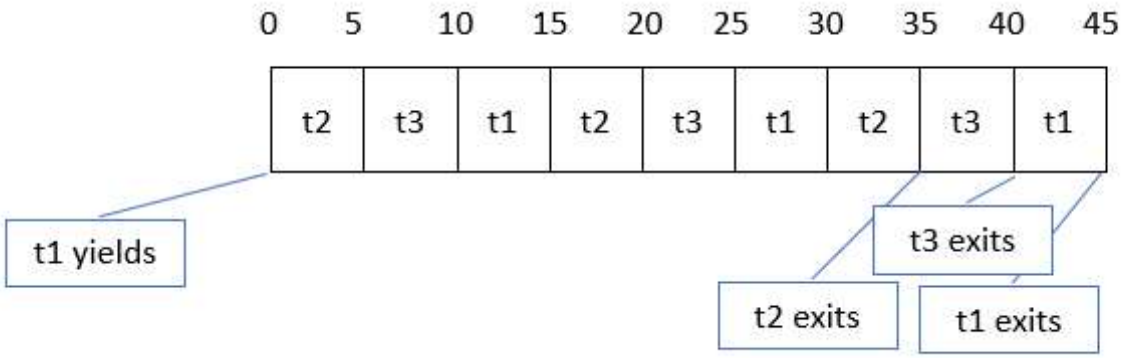
1. **RR with tq=1, the output is:**

```
$ task1
thread id 2 exec 35 ticks
thread id 3 exec 40 ticks
thread id 1 exec 45 ticks

exited
```

ID	maximum execution time	arrival time
1	3 time slot (15 ticks)	0
2	3 time slot (15 ticks)	0

ID	maximum execution time	arrival time
3	3 time slot (15 ticks)	0

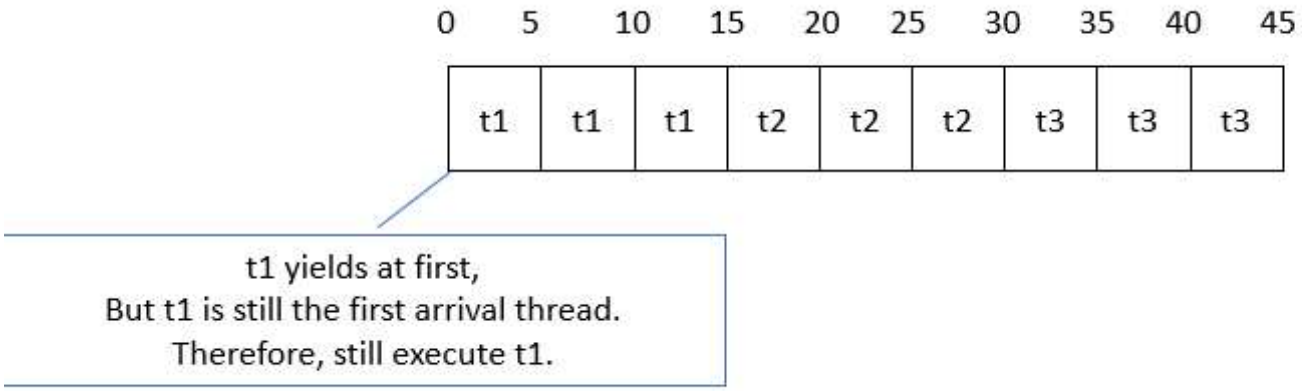


2. **FCFS, the output is:**

```
$ task1
thread id 1 exec 15 ticks
thread id 2 exec 30 ticks
thread id 3 exec 45 ticks

exited
```

ID	maximum execution time	arrival time
1	3 time slot (15 ticks)	0
2	3 time slot (15 ticks)	0
3	3 time slot (15 ticks)	0

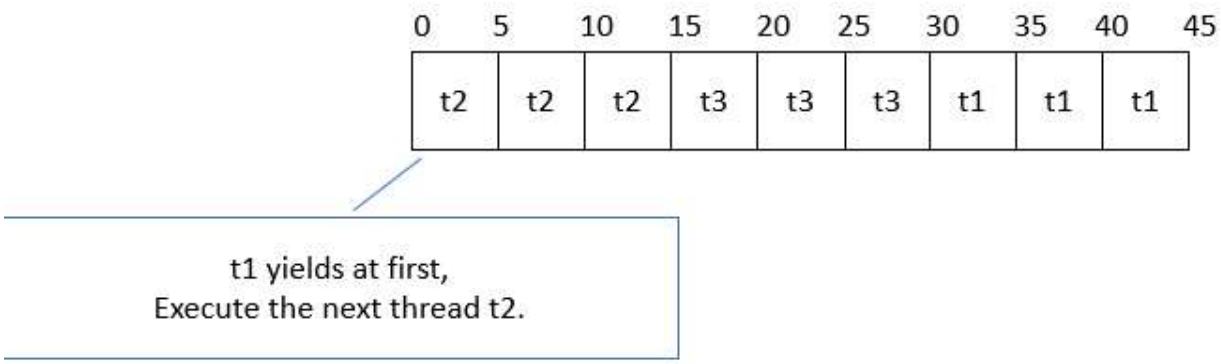


3. **RR with tq=3, the output is:**

```
$ task1
thread id 2 exec 15 ticks
thread id 3 exec 30 ticks
thread id 1 exec 45 ticks

exited
```

ID	maximum execution time	arrival time
1	3 time slot (15 ticks)	0
2	3 time slot (15 ticks)	0
3	3 time slot (15 ticks)	0



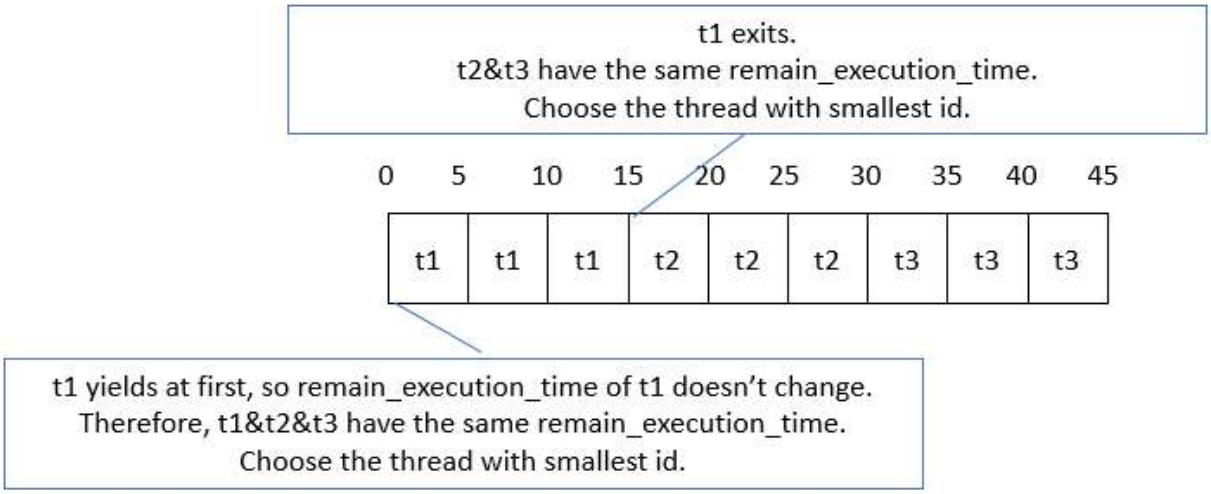
4. SJF, the output is:

```
$ task1
thread id 1 exec 15 ticks
thread id 2 exec 30 ticks
thread id 3 exec 45 ticks

exited
```

ID	maximum execution time	arrival time
1	3 time slot (15 ticks)	0
2	3 time slot (15 ticks)	0

ID	maximum execution time	arrival time
3	3 time slot (15 ticks)	0

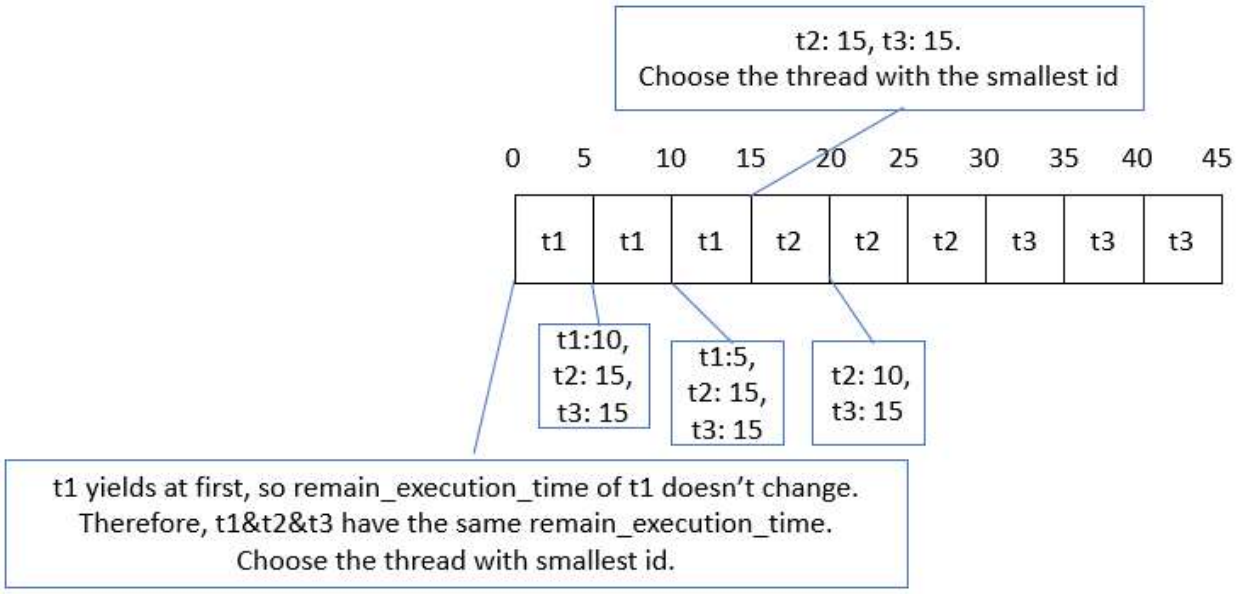


5. PSJF, the output is:

```
$ task1
thread id 1 exec 15 ticks
thread id 2 exec 30 ticks
thread id 3 exec 45 ticks

exited
```

ID	maximum execution time	arrival time
1	3 time slot (15 ticks)	0
2	3 time slot (15 ticks)	0
3	3 time slot (15 ticks)	0



Explnation of task2

`time_slot_size` is 5 ticks.

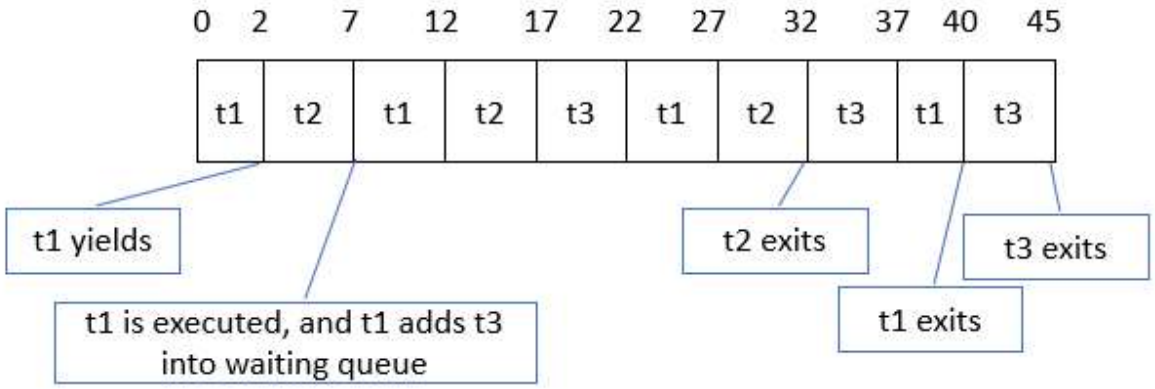
1. When thread1 is running and `uptime()-st1 >= 2`, thread1 will yield once. `st1` is the time when thread1 started.
2. When thread1 have yielded and `uptime()-st1 >= 7`,, thread1 will create thread3. `st1` is the time when thread1 started.

1. **RR with tq=1, the output is:**

```
$ task2
thread id 2 exec 32 ticks
thread id 1 exec 40 ticks
thread id 3 exec 38 ticks

exited
```

ID	maximum execution time	arrival time
1	3 time slot (15 ticks)	0
2	3 time slot (15 ticks)	0
3	3 time slot (15 ticks)	7

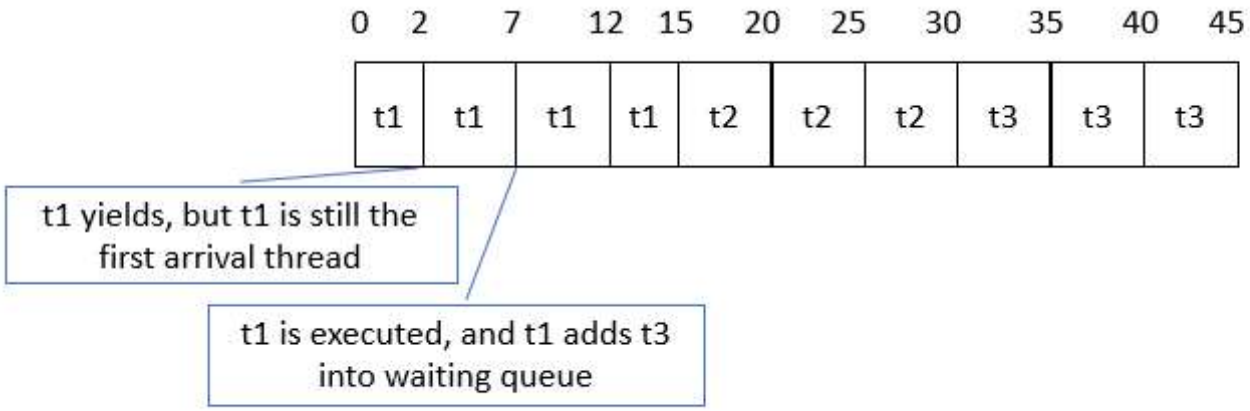


2. **FCFS, the output is:**

```
$ task2
thread id 1 exec 15 ticks
thread id 2 exec 30 ticks
thread id 3 exec 38 ticks

exited
```

ID	maximum execution time	arrival time
1	3 time slot (15 ticks)	0
2	3 time slot (15 ticks)	0
3	3 time slot (15 ticks)	7



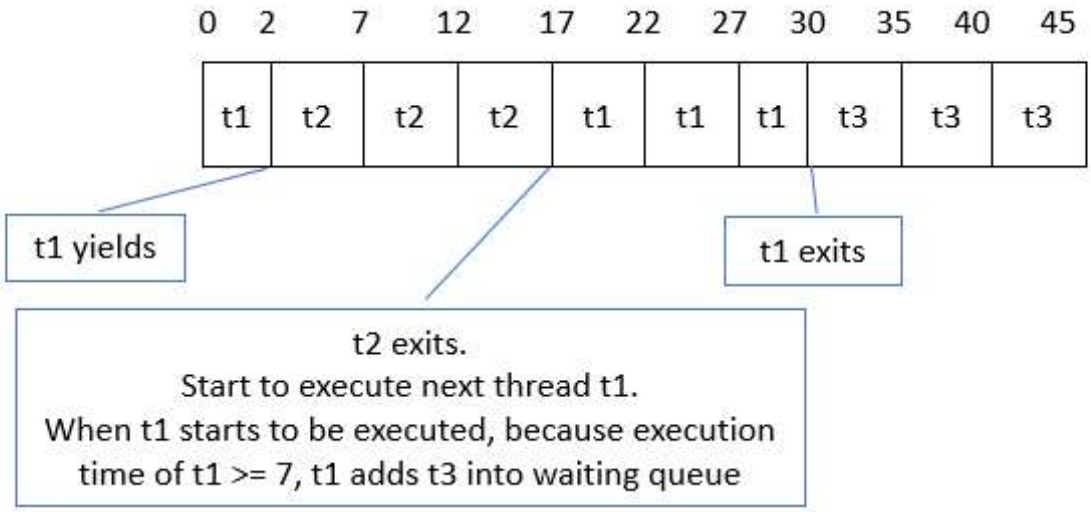
3. **RR with tq=3, the output is:**

```
$ task2
thread id 2 exec 17 ticks
```

```
thread id 1 exec 30 ticks
thread id 3 exec 28 ticks

exited
```

ID	maximum execution time	arrival time
1	3 time slot (15 ticks)	0
2	3 time slot (15 ticks)	0
3	3 time slot (15 ticks)	17



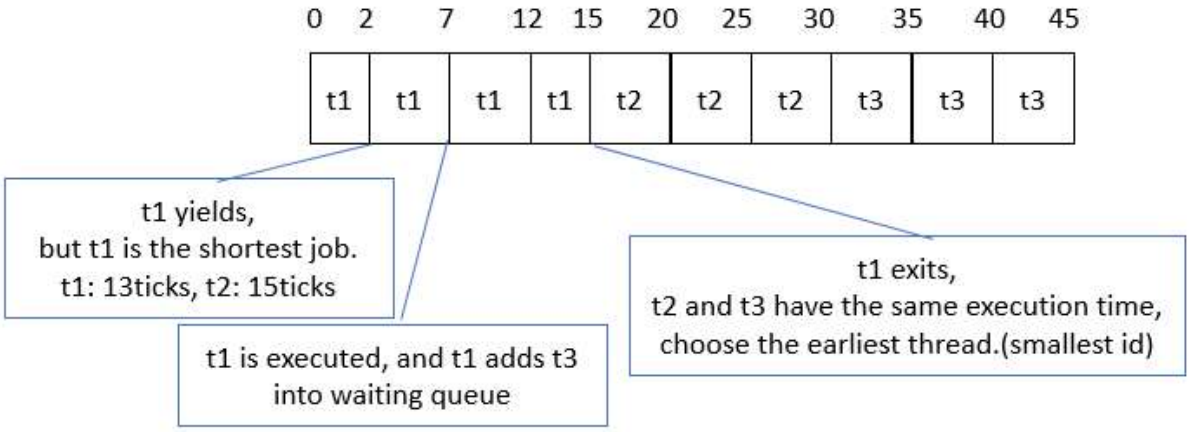
4. SJF, the output is:

```
$ task2
thread id 1 exec 15 ticks
thread id 2 exec 30 ticks
thread id 3 exec 38 ticks

exited
```

ID	maximum execution time	arrival time
1	3 time slot (15 ticks)	0
2	3 time slot (15 ticks)	0

ID	maximum execution time	arrival time
3	3 time slot (15 ticks)	7

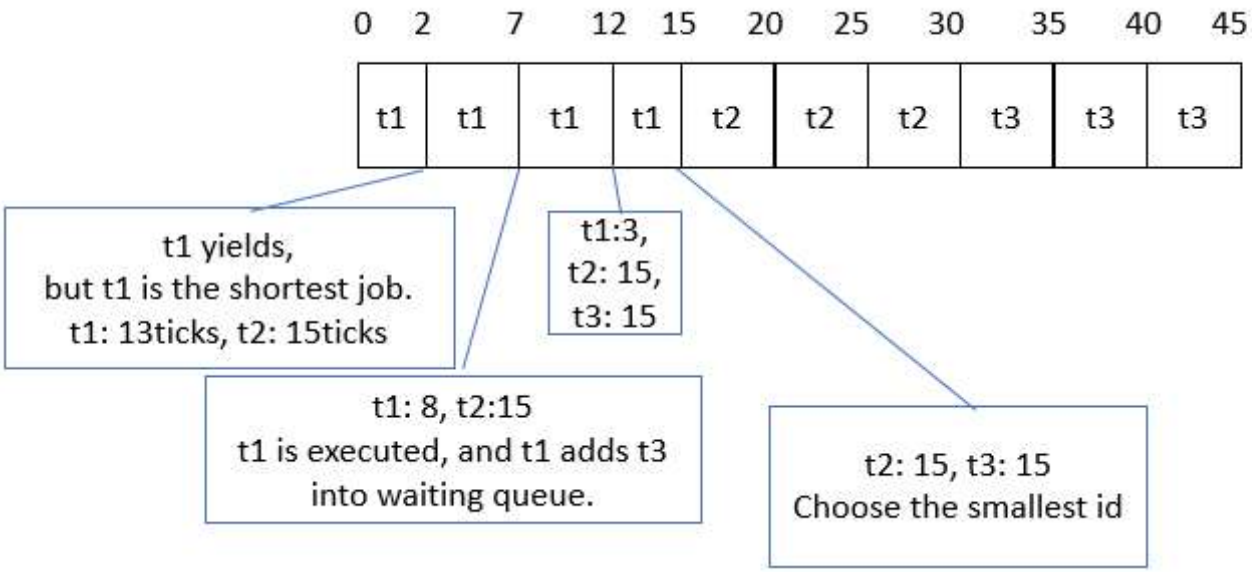


5. PSJF, the output is:

```
$ task2
thread id 1 exec 15 ticks
thread id 2 exec 30 ticks
thread id 3 exec 38 ticks

exited
```

ID	maximum execution time	arrival time
1	3 time slot (15 ticks)	0
2	3 time slot (15 ticks)	0
3	3 time slot (15 ticks)	7



Explnation of task3

`time_slot_size` is 3 ticks.

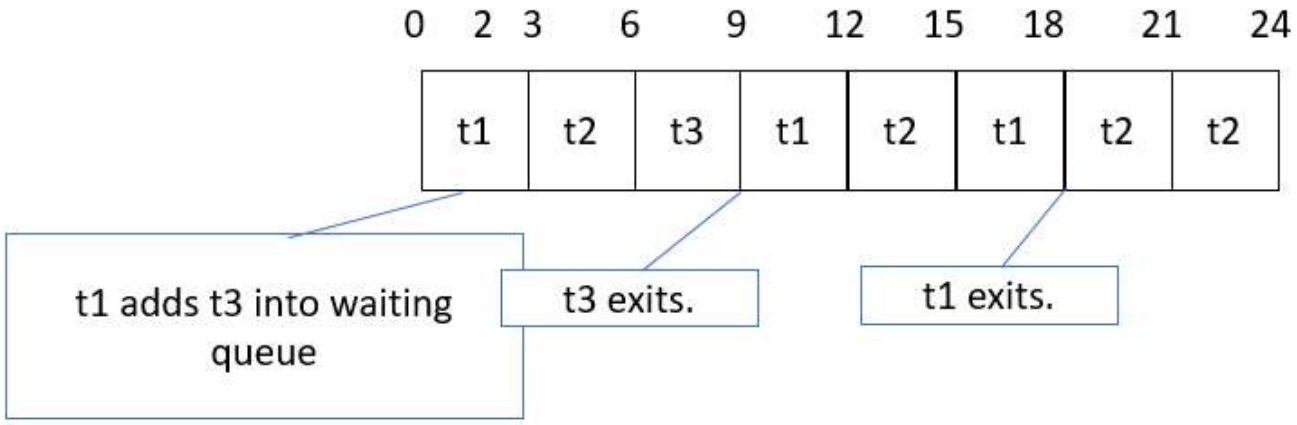
1. When thread1 is running and `uptime()-st1 >= 2`, thread1 will create thread3. `st1` is the time when thread1 started.

1. **RR with tq=1, the output is:**

```
$ task3
thread id 3 exec 7 ticks
thread id 1 exec 18 ticks
thread id 2 exec 24 ticks

exited
```

ID	maximum execution time	arrival time
1	3 time slot (9 ticks)	0
2	4 time slot (12 ticks)	0
3	1 time slot (3 ticks)	2

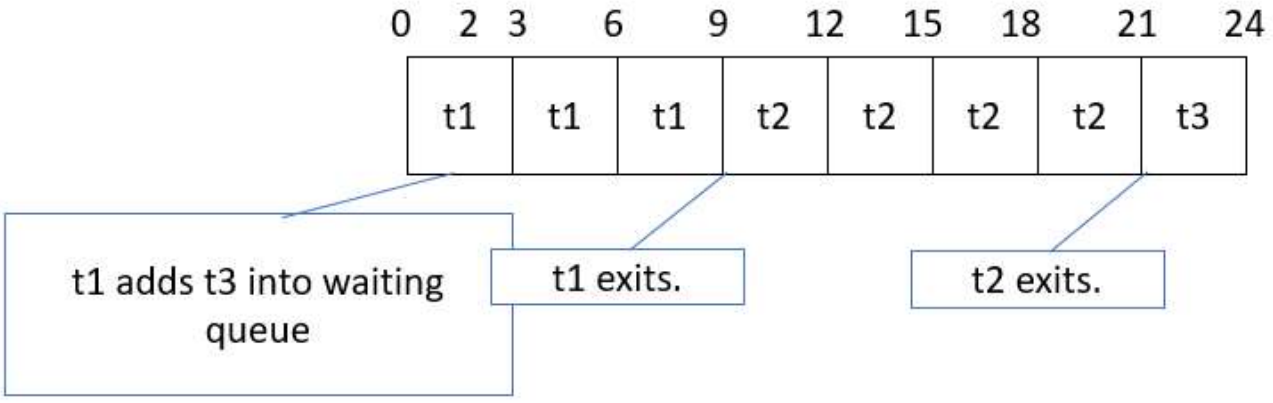


2. **FCFS, the output is:**

```
$ task3
thread id 1 exec 9 ticks
thread id 2 exec 21 ticks
thread id 3 exec 22 ticks

exited
```

ID	maximum execution time	arrival time
1	3 time slot (9 ticks)	0
2	4 time slot (12 ticks)	0
3	1 time slot (3 ticks)	2

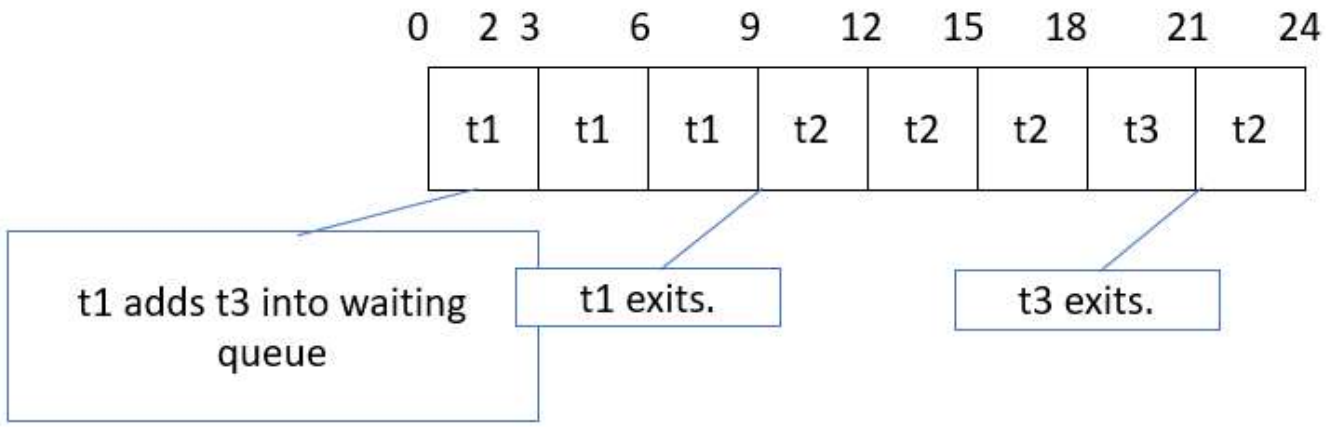


3. **RR with tq=3, the output is:**

```
$ task3
thread id 1 exec 9 ticks
thread id 3 exec 19 ticks
thread id 2 exec 24 ticks

exited
```

ID	maximum execution time	arrival time
1	3 time slot (9 ticks)	0
2	4 time slot (12 ticks)	0
3	1 time slot (3 ticks)	2

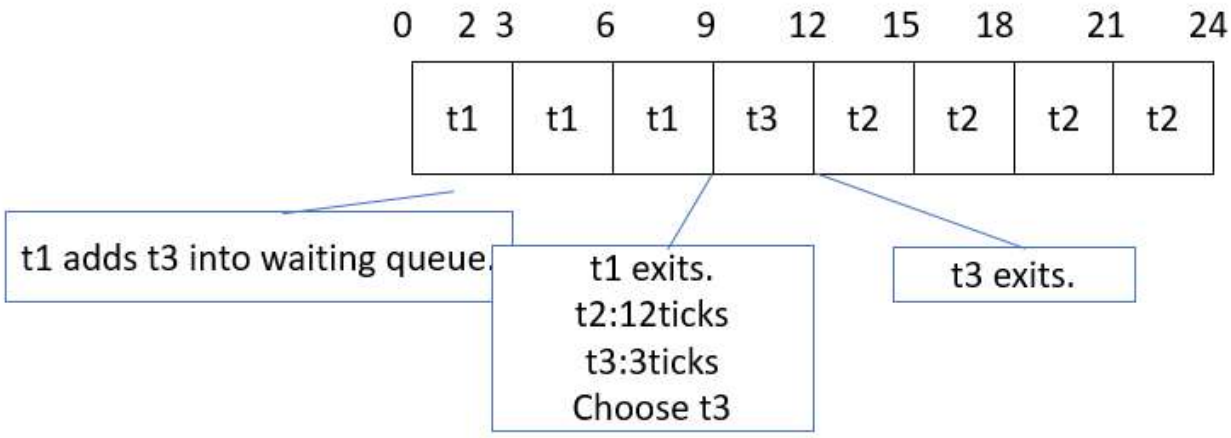


4. SJF, the output is:

```
$ task3
thread id 1 exec 9 ticks
thread id 3 exec 10 ticks
thread id 2 exec 24 ticks

exited
```

ID	maximum execution time	arrival time
1	3 time slot (9 ticks)	0
2	4 time slot (12 ticks)	0
3	1 time slot (3 ticks)	2

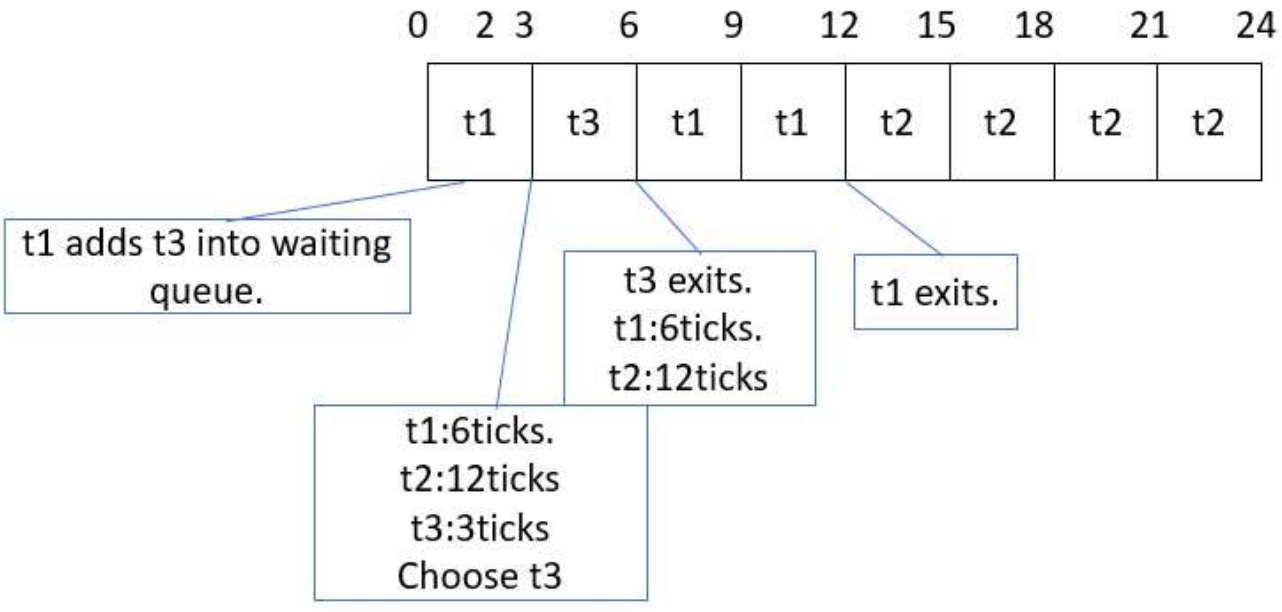


5. PSJF, the output is:

```
$ task3
thread id 3 exec 4 ticks
thread id 1 exec 12 ticks
thread id 2 exec 24 ticks

exited
```

ID	maximum execution time	arrival time
1	3 time slot (9 ticks)	0
2	4 time slot (12 ticks)	0
3	1 time slot (3 ticks)	2



Grading

- Syscalls: 10%
- FCFS: 15%
- RR: 15%
- SJF: 20%
- PSJF: 20%
- Report: 20%
    1. Explain how you implement 3 syscalls. (9%)
    2. When you switch to the `thrdstop_handler`, what context do you store? Is it redundant to store all callee and caller registers? **UPDATE: Explain your reason**(6%)
    3. Take a look at `struct context` in `/kernel/proc.h`. In context switching for processes, why does it only save callee registers and the ra register? (5%)

How to submit

Submit report in gradescope. Add 1 directory `myAns` in `xv6-riscv`. `myAns` should contain 3 directories, `kernel`, `user`, `SOL` Then push them to github class

```
xv6-riscv/
├── myAns
│   ├── kernel
│   │   ├── ...
...
│   └── user
│       ├── ...
...
│   ├── SOL
│   │   ├── FCFS
│   │   │   ├── threads.c
│   │   │   └── threads.h
│   │   ├── PSJF
│   │   │   ├── threads.c
│   │   │   └── threads.h
│   │   ├── RR
│   │   │   ├── threads.c
│   │   │   └── threads.h
│   │   └── SJF
│   │       ├── threads.c
│   │       └── threads.h
└── ...
```

```
|
....
```

When we test your implementation

1.We will replace the `kernel` and `user` in `xv6-riscv` with your `kernel` and `user`.

1. Copy the corresponding `thread.h`, `thread.c` to `xv6-riscv/user/thread.h`, `xv6-riscv/user/thread.c`
2. In order to add some test files, we will use the our Makefile. Therefore, you shouldn't remove or add files in `kernel` and `user`.
3. You should check that you can compile successfully with original Makefile.

Early Bird (5%)

1. Finish the **coding part** before 5/18 23:59, and it should be a correct solution. No early bird bonus for an incorrect answer.
2. We'll announce the grade in 1 day. You can submit again after we grade the early bird.
3. We will choose the higher one, if you submit for early bird and normal deadline.

Office Hour

5/13 17:00~18:00

5/18 17:00~18:00

5/20 17:00~18:00

資工540
