Machine Problem 3 - Scheduling

CSIE3310 - Operating Systems
National Taiwan University

Total Points:	100
Release Date:	April 26
Due Date:	May 09, 23:59:00
TA e-mail:	`ntuos@googlegroups.com`
TA hours:	04/29 08:00-09:00, 05/04 10:00-11:00, https://meet.google.com/dgs-qwry-uvi

Contents

1 Algorithms Description

In this mp you will implement scheduling algorithms listed below.
Given processes in Table 1:

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

Table 1:

1.1 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

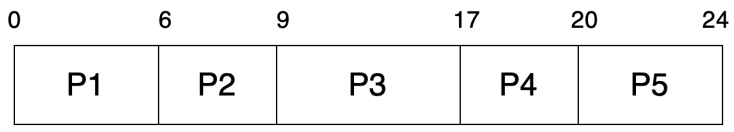

Figure 1: First Come First Served

1.2 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

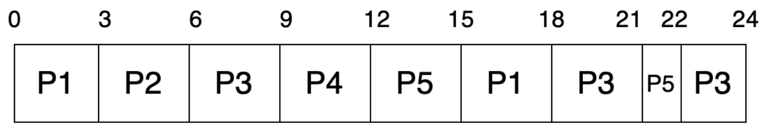

Figure 2: Round Robin

1.3 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

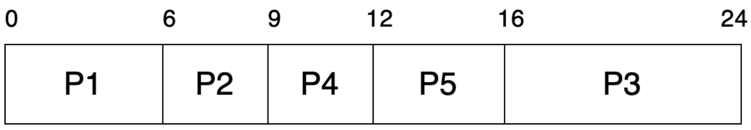

Figure 3: Non-Preemptive Shortest Job First

1.4 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

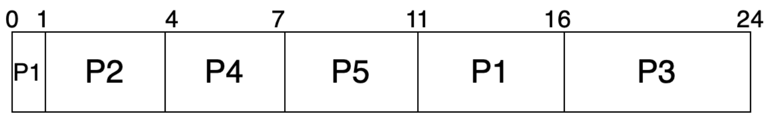

Figure 4: Preemptive SJF

2 MP3 Description

You have finished a user-level thread package in mp1. In this mp, you are going to implement the function **schedule()** with 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 CPU ticks and interrupt threads. Therefore, you are asked to implement 3 syscalls to achieve those goals. Their usage will be described below.

After you finish the syscalls, you should implement the scheduler. We will provide you **threads.c** and **threads.h**, and every function in **threads.c** will be implemented, and they use the syscalls you implement. The scheduling algorithm in our template is RR with time_quantum = 1 , so you should modify **schedule()** for other algorithms. You can also modify other functions if you want to.

2.1 Environment Setup

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

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

Run the docker:

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

3 Syscalls

3.1 int thrdstop()

int thrdstop(int delay, int thrdstop_context_id, void (*thrdstop_handler)(), void *handler_arg);

The primary usage of this syscall is:

- If a process calls **thrdstop()**, switch to execute the **thrdstop_handler** with an argument **handler_arg** after this process consumes **delay** ticks.
- Store the current program context according to **thrdstop_context_id**. It is similar to the **setjmp** in the MP1. **thrdstop_context_id** and the return value will be described below.

3.1.1 Invoke the handler

- You need to record how many ticks have been consumed by the process after the process calls **thrdstop()**. Therefore, you can add more attributes in **struct proc** in /**kernel**/**proc.h** for storing it and the **delay**. You can initialize your attributes in **allocproc()** in /**kernel**/**proc.c**.
- You also need some new attributes for storing **thrdstop_handler** and **handler_arg**.
- Every tick passes, the hardware clock forces an interruption, which is handled by **usertrap()** and **kerneltrap()** in **kernel/trap.c** . You can find something similar to:

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

  You may count down in those "if blocks".
- When the process is about to return to userspace, **usertrapret()** will be called. Find out what determines the value of the program counter when xv6 backs in userspace. (check xv6 book chapter4)
- As for the argument of the handler, you should know the calling convention of the **risc-v**.

3.1.2 Store the context

You need to store the context (register contents) before you switch to the handler. By doing so, you can restore the context when you want to switch back.(**thrdresume()** will handle the job of restoring the context)

Because we want to support multiple threads, you need to store multiple contexts. The **thrdstop()** should assign an ID number for each context of the caller. If we call **thrdstop(n, -1, handler)**, the **thrdstop()** should assign a new ID number for the caller; If we call **thrdstop(n, i, handler)**, it means that this caller has been assigned an ID number **i**. The ID number you assigned should ≥ 0. The return value of the **thrdstop** is the ID number.

When you are going to return to userspace (in **usertrapret()**), you need to find where the current user program context is. Then, you can store the context. Check xv6 book chapter4 for more information.

3.1.3 Test Case Specification

- We only call one **thrdstop** at the same time. For example:

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

  This will not happen. You only need to track one countdown at the same time.
- **int delay** > 0. (argument of **thrdstop()**)
- **thrdstop_handler** must be a function of the process.
- The number of ID numbers $\leq$ **MAX_THRD_NUM**. (defined in /**kernel**/**proc.h**)
- If we call **thrdstop(n, i, handler)**, then **i** must must be an ID number assigned by the **thrdstop()**.

3.2 int thrdresume()

int thrdresume(int thrdstop_context_id)

3.2.1 Description

Reload the context according to the **thrdstop_context_id**. It is similar to the **longjmp** in the MP1. Return any value you want, because we do not use it.

For example:

```
1  int kk = 0, main_id = -1212, bb = 0, tmp_time;
2  void handler1(void *addr)
3  {
4      kk = 123;
5      tmp_time = uptime() - tmp_time;
6      printf("after %d ticks\narg = %p\n", tmp_time, addr);
7      thrdresume(main_id); // should switch back to the while loop block at line 14.
8  }
9  
10 int main(){
11     // the handler1 will be called after 4 ticks.
12     main_id = thrdstop( 4, -1, handler1, (void *)0xbbbb87 ) ;
13     tmp_time = uptime();
14     while( kk != 123 ){
15         bb ++;
16     }
17     printf("1\n");
18     exit(0);
19 }
20 /*
21 The output is:
22 after 4 ticks
23 arg = 0x0000000000BBBB87
24 1
25 */
```

Similar to Section 3.1.2, you need to find where the current user program context locates when you are going to return to userspace (in **usertrapret()**). Then, you can replace the current user program context with the context specified by the **thrdstop_context_id**.

3.2.2 Test Case Specification

- **thrdstop_context_id** must be an ID number assigned by the **thrdstop()**.
- **thrdresume()** won't be called when count down timer is active.

3.3 int cancelthrdstop()

int cancelthrdstop(int thrdstop_context_id, int is_exit)

3.3.1 Description

This function cancels the **thrdstop()**. For example

```
thrdstop(10, 5, handler2);
cancelthrdstop(5, 0);
```

In this case, **handler2** won't be called.

The usage of **is_exit**:

- When **is_exit** is 0. It stores the current context according to the **thrdstop_context_id**. This indicates that the thread yielded. No need to store if **thrdstop_context_id** is −1. For example:

```
1  int a = 0, main_id;
2  void handler(void *arg){
3      printf("not be executed\n");
4  }
5  void go_to_2F(){
6      a+= 1;
7      printf("2he2he2he2he2he\n");
8      thrdresume(main_id); // switch back to line 12
9  }
10 int main(){
11     main_id = thrdstop(10, -1, handler, (void *)0xbbbb87);
12     cancelthrdstop(main_id, 0);
13     if( a == 0 ){
14         go_to_2F();
15     }
16     else{
17         printf("back from 2F\n");
18     }
19     exit(0);
20 }
21 /*
22 The output is:
23 2he2he2he2he2he
24 back from 2F
25 */
```

- When **is_exit** is not 0. No need to store the context. This indicates that the thread exited. Therefore, you can recycle the ID number.

The return value is the ticks consumed by the process from the time **thrdstop()** be called. If the count down timer is inactive now, the return value is ticks consumed by the previous count down timer. For example:

```
int main_id;
void handler2(void *arg){
    thrdresume(main_id);
}

int main(){
    main_id = thrdstop(10, -1, handler2, (void *)0);
    int start_time = uptime(), aa;
    while(uptime() - start_time < 15){
        aa ++;
    }
    int gg = cancelthrdstop(-1, 0); // gg == 10
    printf("%d\n", gg);
    main_id = thrdstop(10, -1, handler2, (void *)0);
    start_time = uptime();
    while(uptime() - start_time < 3){
        aa ++;
    }
    gg = cancelthrdstop(-1, 0); // gg == 3
    printf("%d\n", gg);
    exit(0);
}
/*
The output is:
10
```

```
3
*/
```

3.3.2 Test Case Specification

- **thrdstop_context_id** is an ID number assigned by the **thrdstop()** or **thrdstop_context_id** $== -1$
- If **is_exit** $!= 0$, then **thrdstop_context_id** must be an ID number assigned by the **thrdstop()**..

3.4 Test the syscalls

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

```
$ python3 grade-mp3-syscalls
```

You will see something like:

```
.
.
eac8a36c12c94b93e22928c1174372d15dca00a5724481788a7a0a901e065338 passTest 3
.
.
```

"passTest 3" means that you pass 3 test cases. You can also run **test1** and **test2** inside the xv6.

4 Scheduler

4.1 Explanation of the template code in thread.c

- **void my_thrdstop_handler(void *arg)**: This function updates and checks the **remain_execution_time** of thread. **my_thrdstop_handler()** also directly calls **schedule()** and **dispatch()** for switching to the next thread.
- **struct thread *thread_create(void (*f)(void *), void *arg, int execution_time_quantum)**: This function creates a thread object and initializes it. **execution_time_quantum** indicates that maximum execution time of this thread, which is **TIME_QUANTUM_SIZE * execution_time_quantum** ticks. When start threading, **thread->remain_execution_time** will be set to **TIME_QUANTUM_SIZE * execution_time_quantum**. Threads are allowed to exit early, so their execution time can be less than **remain_execution_time**.
- The runqueue is a linked list.
- **TIME_QUANTUM_SIZE**

  indicates that **my_thrdstop_handler** will be called every **TIME_QUANTUM_SIZE** ticks.

4.2 What you need to do for the scheduler

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

  - Execute the next thread when:
    - The current thread exits.
    - The current thread yields.
  - Choose the first thread to be executed based on **thrd->remain_execution_time** or **thrd->ID**.

- Preemptive SJF:
  - Choose the next thread based on **thrd->remain_execution_time** (the shortest one). If **thrd->remain_execution_time** are the same, choose the thread that arrived earlier. The arrival order is determined by **thrd->ID** (the smaller the earlier).
  - Execute the next thread when:
    - The current thread exits.
    - The current thread yields.
    - The current thread has consumed 1 time_quantum. (every time when my_thrdstop_handler call schedule()).
  - Choose the first thread to be executed based on **thrd->remain_execution_time** or **thrd->ID**.

Please implement those algorithms in the corresponding location.

```
void schedule(void){
    #ifdef THREAD_SCHEDULER_DEFAULT
    ...
    #endif

    #ifdef THREAD_SCHEDULER_RR
    ... implement RR here.
    #endif

    #ifdef THREAD_SCHEDULER_FCFS
    ... implement FCFS here.
    #endif


    #ifdef THREAD_SCHEDULER_SJF
    ... implement SJF here.
    #endif

    #ifdef THREAD_SCHEDULER_PSJF
    ... implement PSJF here.
    #endif

}
```

4.3 Test the scheduler

You can compile xv6 by running this command in docker:

```
$ make qemu SCHEDPOLICY=THREAD_SCHEDULER_FCFS
```

If you want to change the scheduler, just replace **THREAD_SCHEDULER_FCFS** with other flags. Remember to run **"make clean"** before you recompile. You can run **task1, task2 ,and task3** after you execute the **xv6**.

There are 5 python files.

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

You can also test your scheduler by those python files.

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

You can test all public test cases by running:

```
$ python3 grade-mp3
```

You can see the result:

```
************************************************************


syscall test: pass, FCFS: pass, SJF: pass, PSJF: pass, RR: pass
your score is 40


************************************************************
```

5 Submission and Grading

Please compress your xv6 source code as <whatever>.zip and upload to NTUCOOL. The filename does not matter since NTUCOOL will rename you submissions. You can run make clean in the container before you compress. Make sure your xv6 can be compiled by **make qemu** with the **Makefile** provided by TA.

5.1 Folder Structure after Unzip

```
<student_id>
|
+-- xv6-riscv
    |
    +-- kernel
    |   |
    |   +-- ...
...
    |
    +-- user
        |
        +-- ...
```

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

xv6-riscv should contain exactly 2 folders. (user and kernel)

5.2 Grading Policy

- There are public test cases and private test cases.
  - public test cases: (a) syscalls, 16%. (b) four scheduling algorithms, 6% for each.
  - private test cases: (a) syscalls, 24%. (b) four scheduling algorithms, 9% 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(raw_score $- 20 \cdot \lceil n \rceil,\ 0$) points. Note that you will not get any points if $\lceil n \rceil \geq 5$.

6 Explanation of public test cases

6.1 Explanation of task1

time_quantum_size is 5. The **t1** will yield when it is executed for the first time. Check the Table 2.

ID	maximum execution time quantum	arrival time
1	3	0
2	3	0
3	3	0

Table 2: Description of task1

6.1.1 Default (RR with tq=1)

The output is:

```
thrd2 exec 35 ticks
thrd3 exec 40 ticks
thrd1 exec 45 ticks

exited
```

See the explanation in the Figure 5.

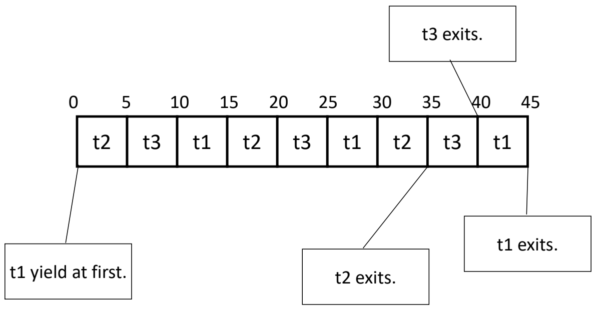

Figure 5: Default task1

6.1.2 RR with tq=3

The output is:

```
thrd2 exec 15 ticks
thrd3 exec 30 ticks
thrd1 exec 45 ticks

exited
```

See the explanation in the Figure 6.

6.1.3 FCFS

The output is:

```
thrd1 exec 15 ticks
thrd2 exec 30 ticks
thrd3 exec 45 ticks

exited
```

See the explanation in the Figure 7.

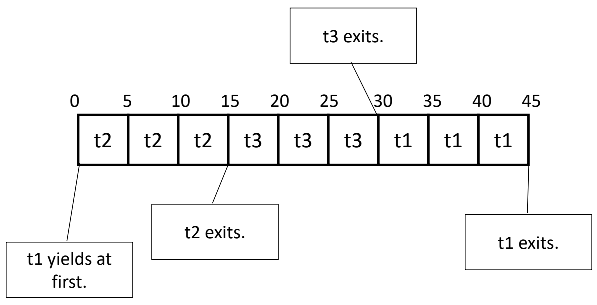

Figure 6: RR task1

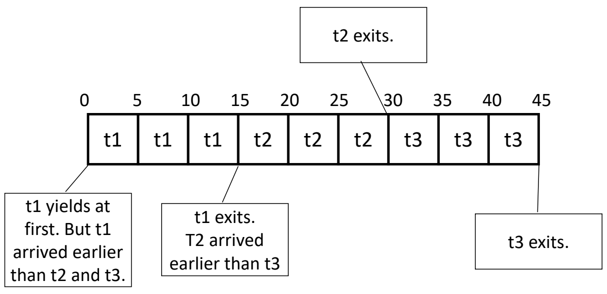
t2 exits.
0 5 10 15 20 25 30 35 40 45
t1 t1 t1 t2 t2 t2 t3 t3 t3
t1 yields at first. But t1 arrived earlier than t2 and t3.
t1 exits. T2 arrived earlier than t3
t3 exits.

Figure 7: FCFS task1

6.1.4 SJF

The output is:

```
thrd1 exec 15 ticks
thrd2 exec 30 ticks
thrd3 exec 45 ticks

exited
```

See the explanation in the Figure 8.

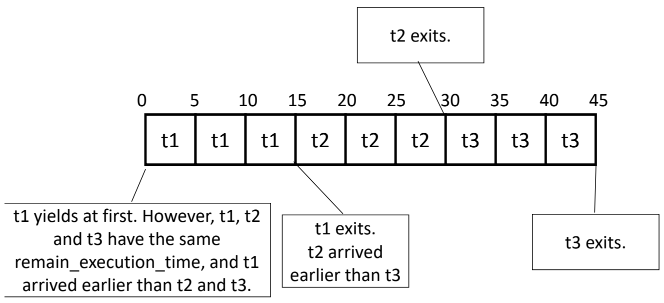
t2 exits.
0 5 10 15 20 25 30 35 40 45
t1 t1 t1 t2 t2 t2 t3 t3 t3
t1 yields at first. However, t1, t2 and t3 have the same remain_execution_time, and t1 arrived earlier than t2 and t3.
t1 exits. t2 arrived earlier than t3
t3 exits.

Figure 8: SJF task1

6.1.5 PSJF

The output is:

```
thrd1 exec 15 ticks
thrd2 exec 30 ticks
thrd3 exec 45 ticks

exited
```

See the explanation in the Figure 9.

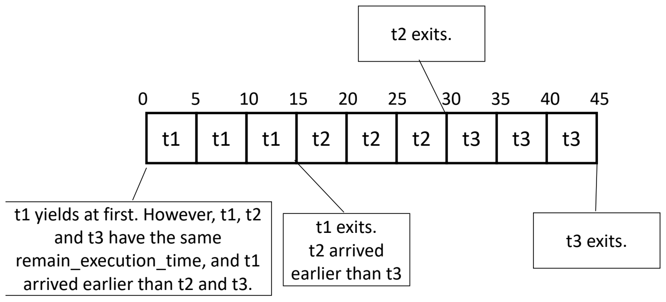
t2 exits.
0 5 10 15 20 25 30 35 40 45
t1 t1 t1 t2 t2 t2 t3 t3 t3
t1 yields at first. However, t1, t2 and t3 have the same remain_execution_time, and t1 arrived earlier than t2 and t3.
t1 exits. t2 arrived earlier than t3
t3 exits.

Figure 9: PSJF task1

6.2 Explanation of task2

time_quantum_size is 5. The **t1** will yield after it has executed for 2 ticks. Check the Table 3.

ID	maximum execution time quantum	arrival time
1	3	0
2	3	0
3	3	7

Table 3: Description of task2

6.2.1 Default (RR with tq=1)

The output is:

```
thrd2 exec 37 ticks
thrd1 exec 40 ticks
thrd3 exec 38 ticks

exited
```

See the explanation in the Figure 10.

6.2.2 RR with tq=3

The output is:

```
thrd2 exec 17 ticks
thrd1 exec 30 ticks
thrd3 exec 38 ticks

exited
```

See the explanation in the Figure 11.

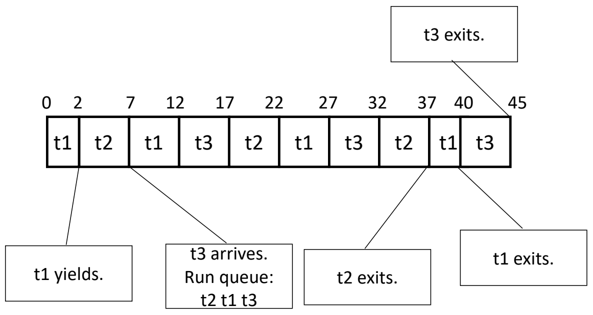

Figure 10: Default task2

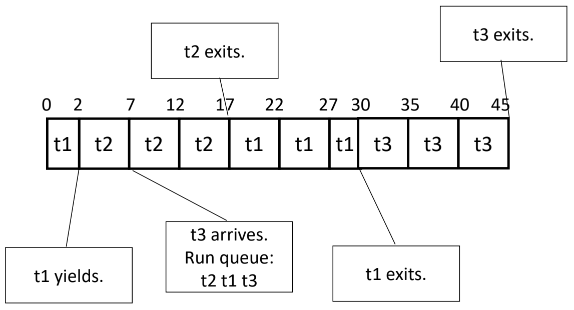

Figure 11: RR task2

6.2.3 FCFS

The output is:

```
thrd1 exec 15 ticks
thrd2 exec 30 ticks
thrd3 exec 38 ticks

exited
```

See the explanation in the Figure 12.

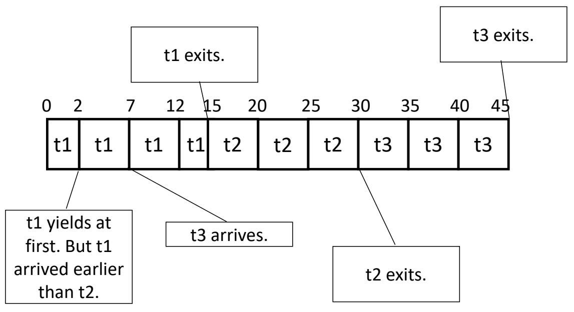
t1 exits.
t3 exits.
0 2 7 12 15 20 25 30 35 40 45
t1 t1 t1 t1 t2 t2 t2 t3 t3 t3
t1 yields at first. But t1 arrived earlier than t2.
t3 arrives.
t2 exits.

Figure 12: FCFS task2

6.2.4 SJF

The output is:

```
thrd1 exec 15 ticks
thrd2 exec 30 ticks
thrd3 exec 38 ticks

exited
```

See the explanation in the Figure 13.

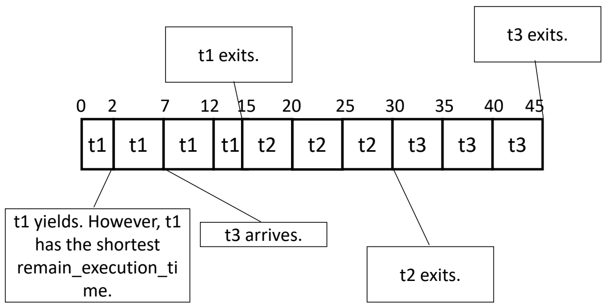
t1 exits.
t3 exits.
0 2 7 12 15 20 25 30 35 40 45
t1 t1 t1 t1 t2 t2 t2 t3 t3 t3
t1 yields. However, t1 has the shortest remain_execution_time.
t3 arrives.
t2 exits.

Figure 13: SJF task2

6.2.5 PSJF

The output is:

```
thrd1 exec 15 ticks
thrd2 exec 30 ticks
thrd3 exec 38 ticks

exited
```

See the explanation in the Figure 14.

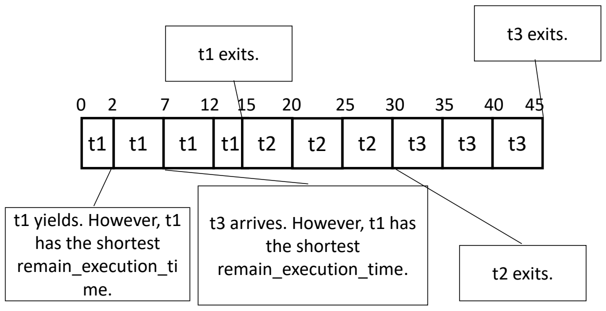
t1 exits.
t3 exits.
0 2 7 12 15 20 25 30 35 40 45
t1 t1 t1 t1 t2 t2 t2 t3 t3 t3
t1 yields. However, t1 has the shortest remain_execution_time.
t3 arrives. However, t1 has the shortest remain_execution_time.
t2 exits.

Figure 14: PSJF task2

6.3 Explanation of task3

time_quantum_size is 5. Check the Table 4.

ID	maximum execution time quantum	arrival time
1	4	0
2	3	0
3	3	0
4	1	5

Table 4: Description of task3

6.3.1 Default (RR with tq=1)

The output is:

```
thrd4 exec 15 ticks
thrd2 exec 45 ticks
```

```
thrd3 exec 50 ticks
thrd1 exec 55 ticks

exited
```

See the explanation in the Figure 15.

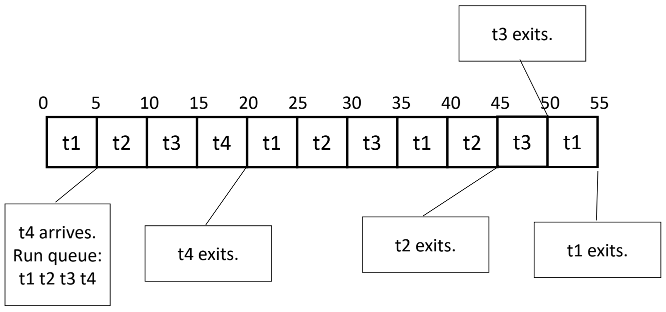

Figure 15: Default task3

6.3.2 RR with tq=3

The output is:

```
thrd2 exec 30 ticks
thrd3 exec 45 ticks
thrd4 exec 45 ticks
thrd1 exec 55 ticks

exited
```

See the explanation in the Figure 16.

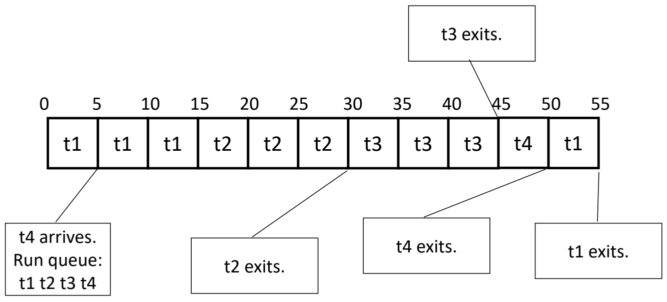

Figure 16: RR task3

6.3.3 FCFS

The output is:

```
thrd1 exec 20 ticks
thrd2 exec 35 ticks
thrd3 exec 50 ticks
thrd4 exec 50 ticks

exited
```

See the explanation in the Figure 17.

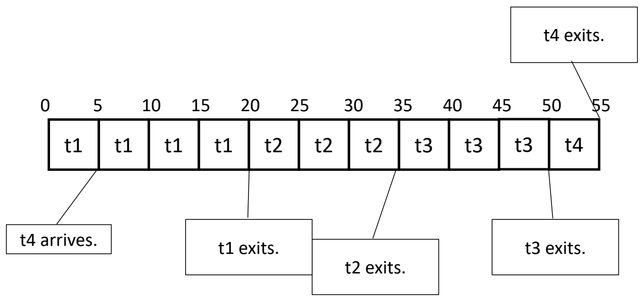

Figure 17: FCFS task3

6.3.4 SJF

The output is:

```
thrd2 exec 15 ticks
thrd4 exec 15 ticks
thrd3 exec 35 ticks
thrd1 exec 55 ticks

exited
```

See the explanation in the Figure 18.

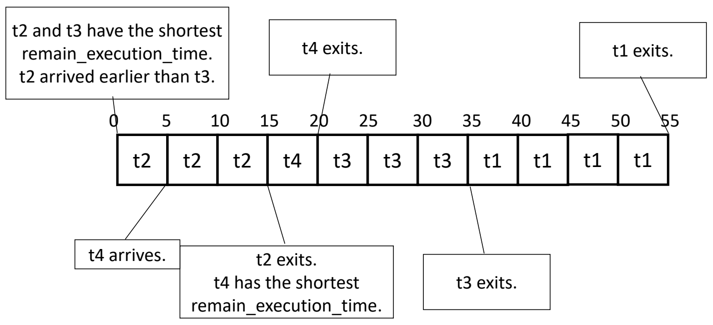
t2 and t3 have the shortest remain_execution_time. t2 arrived earlier than t3.
t4 exits.
t1 exits.
0 5 10 15 20 25 30 35 40 45 50 55
t2 t2 t2 t4 t3 t3 t3 t1 t1 t1 t1
t4 arrives.
t2 exits. t4 has the shortest remain_execution_time.
t3 exits.

Figure 18: SJF task3

6.3.5 PSJF

The output is:

```
thrd4 exec 5 ticks
thrd2 exec 20 ticks
thrd3 exec 35 ticks
thrd1 exec 55 ticks

exited
```

See the explanation in the Figure 19.

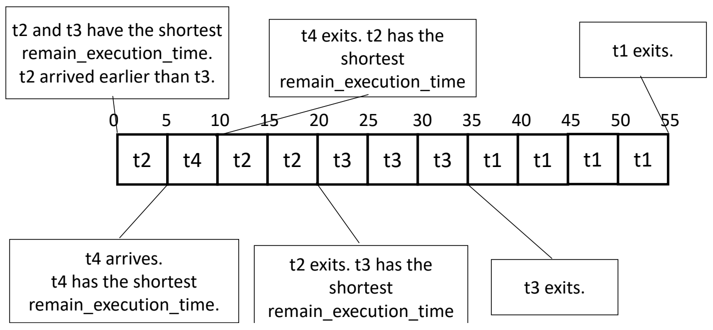
t2 and t3 have the shortest remain_execution_time. t2 arrived earlier than t3.
t4 exits. t2 has the shortest remain_execution_time
t1 exits.
0 5 10 15 20 25 30 35 40 45 50 55
t2 t4 t2 t2 t3 t3 t3 t1 t1 t1 t1
t4 arrives. t4 has the shortest remain_execution_time.
t2 exits. t3 has the shortest remain_execution_time
t3 exits.

Figure 19: PSJF task3