

National University Ho Chi Minh City
HO CHI MINH UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING



Operating System - CO2018

Instructor: Assoc. Phạm Hoàng Anh

Class: CC04

Name: Nguyễn Quang Thiện

Student ID: 21522994

Completion Day: April 12th, 2023

Table of Contents

I.	Practice with CronTab (1.5)	2
	<i>Implementing</i>	2
II.	Exercise 2.2	4
	1. <i>Requirement</i>	4
	2. <i>Steps to build</i>	4
	3. <i>Explain the code paragraph</i>	4
	4. <i>Implementing</i>	8
	Source code of Exercise 2.2 (queue.c)	9
	Source code of Exercise 2.2 (sched.c)	10
III.	Homework (3.1, 3.2, 3.3)	14
	1. <i>Requirement</i>	14
	2. <i>Steps to build</i>	14
	3. <i>Explain each code paragraph</i>	15
	4. <i>Implementing</i>	18
	Source code of 3.1, 3.2, 3.3	19

I. Practice with CronTab (1.5)

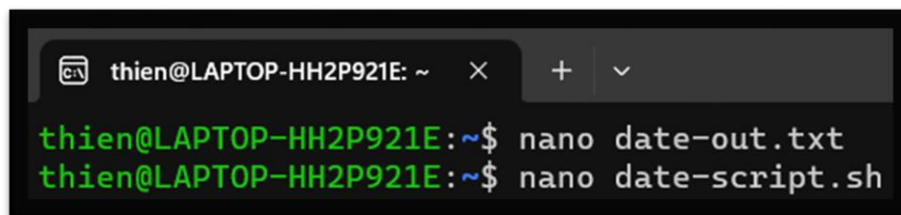
Implementing

A scheduling daemon called Cron runs tasks at predetermined intervals. Cron jobs are what are commonly used to automate system administration or maintenance.

To automate repetitive operations, for instance, we may set up a cron job to back up databases or data, update the system with the most recent security patches, monitor the amount of disk space used, send emails, etc.

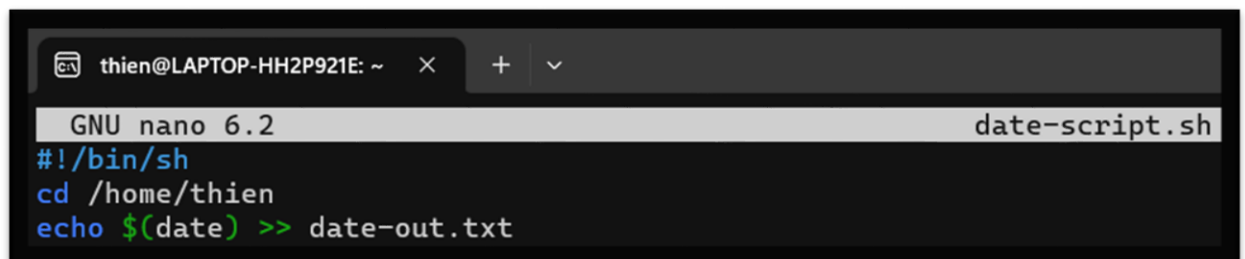
Cron tasks can be set to run every minute, every hour, every day of the week, every month, or any combination of these.

Initially, creating an empty file to save the output: **date-out.txt**



```
thien@LAPTOP-HH2P921E: ~  
thien@LAPTOP-HH2P921E:~$ nano date-out.txt  
thien@LAPTOP-HH2P921E:~$ nano date-script.sh
```

Then, make the following file with the contents displayed: **date-script.sh**



```
GNU nano 6.2 date-script.sh  
#!/bin/sh  
cd /home/thien  
echo $(date) >> date-out.txt
```

Next, opening the crontab configuration file (~\$ **crontab -e**) and adding the following command:

```
thien@LAPTOP-HH2P921E: ~ × + ▾
GNU nano 6.2 /tmp/crontab.ocL5Iy/crontab *
# Edit this file to introduce tasks to be run by cron.
#
# Each task to run has to be defined through a single line
# indicating with different fields when the task will be run
# and what command to run for the task
#
# To define the time you can provide concrete values for
# minute (m), hour (h), day of month (dom), month (mon),
# and day of week (dow) or use '*' in these fields (for 'any').
#
# Notice that tasks will be started based on the cron's system
# daemon's notion of time and timezones.
#
# Output of the crontab jobs (including errors) is sent through
# email to the user the crontab file belongs to (unless redirected).
#
# For example, you can run a backup of all your user accounts
# at 5 a.m every week with:
# 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
#
# For more information see the manual pages of crontab(5) and cron(8)
#
# m h dom mon dow  command
*/1 * * * * /home/thien/date-script.sh
```

Waiting a short while to check our job (using **chmod** command to make the file executable):

```
thien@LAPTOP-HH2P921E: ~ × + ▾
thien@LAPTOP-HH2P921E:~$ chmod +x date-script.sh
thien@LAPTOP-HH2P921E:~$ ./date-script.sh
thien@LAPTOP-HH2P921E:~$ sudo service cron start
[sudo] password for thien:
* Starting periodic command scheduler cron
thien@LAPTOP-HH2P921E:~$ cat date-out.txt
Fri Apr 14 19:30:28 +07 2023
Fri Apr 14 19:31:01 +07 2023
Fri Apr 14 19:32:01 +07 2023
Fri Apr 14 19:33:01 +07 2023
Fri Apr 14 19:34:01 +07 2023
Fri Apr 14 19:34:49 +07 2023
Fri Apr 14 19:35:01 +07 2023
thien@LAPTOP-HH2P921E:~$
```

II. Exercise 2.2

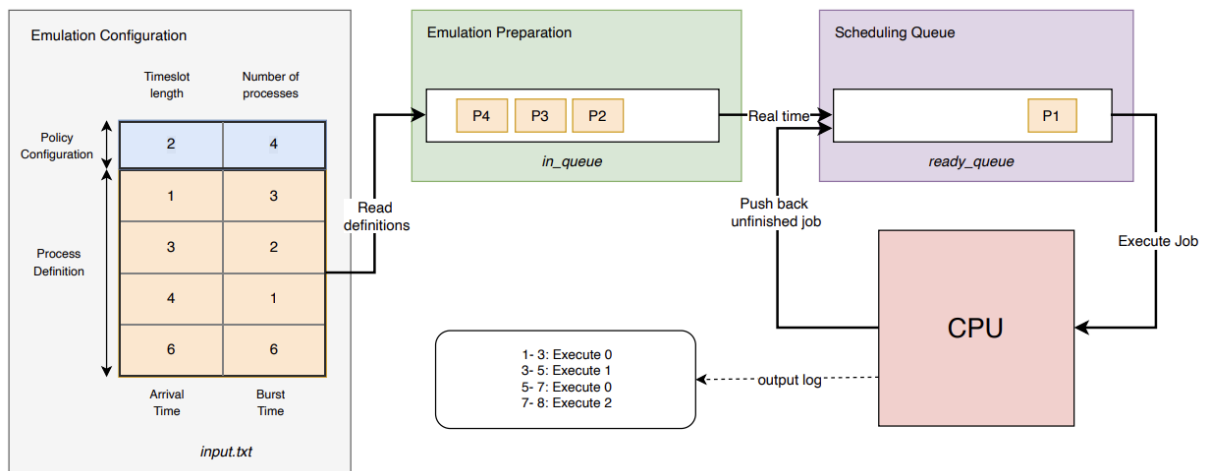
1. Requirement

Completing these methods which are marked by the `//TODO` directives:

1. The **en_queue** and the **de_queue** methods of the queue source file **queue.c**.
2. The behavior of the CPU() when executes a process (in **sched.c**):

- Calculating the time that the CPU() (**exec_time**) will spend to execute a process that is loaded into the CPU. The **exec_time** must not exceed the timeslot threshold. And updating the process **burst_time** to avoid infinite execution loops.
- Check whether the process has finished its execution. If the process is done, free the allocated memory of the process object (**pcb_t**). Otherwise, reinsert the process to the **ready_queue**.

Then, creating a Makefile with a sched target that prints the output. Considering the data structure which is declared in **structs.h**. This the file declares the attributes and the data structure of the process and queue.



OS scheduling algorithm simulation model

2. Steps to build

Applying knowledge about linked list and mutex clock to implementing the **en_queue** and **de_queue**.

Calculating the **exec_time** by the technique choosing the smaller time between **burst_time** and **time slot** then updating the **burst_time**.

After that, checking if the process has terminated. The **burst_time** is zero, so free its PCB. Otherwise, reinsert the process to the **ready_queue**.

3. Explain the code paragraph

Queue.c:

Implementing **de_queue** function:

```
struct pcb_t * de_queue(struct pqueue_t * q)
{
    struct pcb_t * proc = NULL;
    // TODO: return q->head->data and remember to update the queue's head
    // and tail if necessary. Remember to use 'lock' to avoid race
    // condition
    // YOUR CODE HERE
    pthread_mutex_lock(&q->lock);

    // if empty (non zero) --> return
    if(empty(q))
    {
        pthread_mutex_unlock(&q->lock);
        return proc;
    }

    // store data to return
    proc = q->head->data;
    struct qitem_t *tmp = q->head;
    // update head
    q->head=q->head->next;
    free(tmp);
    // if head == NULL, then tail == NULL
    if(empty(q))
        q->tail=NULL;

    pthread_mutex_unlock(&q->lock);
    return proc;
}
```

Implementing **en_queue** function:

```

void enqueue(struct pqueue_t *q, struct pcb_t *proc)
{
    // TODO: Update q->tail.
    // Remember to use 'lock' to avoid race condition
    // YOUR CODE HERE
    pthread_mutex_lock(&q->lock);

    struct qitem_t *node = (struct qitem_t *)malloc(sizeof(struct qitem_t));
    node->data = proc;
    if (empty(q))
    {
        q->head = node;
        q->tail = node;
        node->next=NULL;
    }
    else
    {
        q->tail->next = node;
        q->tail = node;
        node->next = NULL;
    }
    pthread_mutex_unlock(&q->lock);
    return;
}

```

Sched.c:

Calculating the **exec_time**:

```

// TODO: Calculate exec_time from process's PCB

// YOUR CODE HERE

// choose smaller time between burst time & time slot
//exec_time = proc->burst_time > timeslot ? timeslot : proc->burst_time;
if(proc->burst_time >= timeslot)
{
    exec_time = timeslot;
    proc->burst_time -= exec_time;
}
else
{
    exec_time = proc->burst_time;
    proc->burst_time = 0;
}

// // update burst time
// proc->burst_time -= exec_time;

```

Checking whether the process has finished its execution:

```

// TODO: Check if the process has terminated (i.e. its
// burst time is zero. If so, free its PCB. Otherwise,
// put its PCB back to the queue.

// YOUR CODE HERE
if (proc->burst_time == 0)
{
    free(proc);
}
else
{
    en_queue(&ready_queue, proc);
}

/* Track runtime status */
printf("%2d-%2d: Execute %d\n", start, timestamp, id);

```

After that, generating a makefile with a sched target that prints the output:

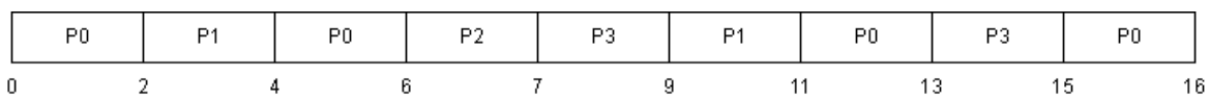

```
all:
    #!/bin/bash
    echo "Input file:"
    cat input.txt
    echo "Results"
    gcc sched.c queue.c -o sched -lpthread
    cat input.txt | ./sched
clean:
    rm -f all
```

4. *Implementing*

Results:

```
thien@LAPTOP-HH2P921E: ~$ make all
#!/bin/bash
echo "Input file:"
Input file:
cat input.txt
2 4
0 7
2 4
4 1
5 4
echo "Results"
Results
gcc sched.c queue.c -o sched -lpthread
cat input.txt | ./sched
finish loading input
0- 2: Execute 0
2- 4: Execute 1
4- 6: Execute 0
6- 7: Execute 2
7- 9: Execute 3
9-11: Execute 1
11-13: Execute 0
13-15: Execute 3
15-16: Execute 0
```

The outcome is following Gantt chart below:



Nevertheless, the Gantt chart for this issue may look something like this:

P0	P0	P1	P0	P2	P3	P1	P0	P3	
0	2	4	6	8	9	11	13	14	16

Because two processes, P0 and P1, entered the queue at the same moment in this instance, that is why there is a difference. As only one thread may alter the queue at a time due to the mutex lock we have in place, P0 or P1 will join the queue first, depending on when **CPU()** and **loader()** finish their tasks. My result and the first Gantt chart both indicate that P1 enters the queue first, whereas the second Gantt chart is inversed.

Source code of Exercise 2.2 (queue.c)

```
#include <stdlib.h>
#include "queue.h"
#include <pthread.h>

/* Remember to initilize the queue before using it */
void initialize_queue(struct pqueue_t *q)
{
    q->head = q->tail = NULL;
    pthread_mutex_init(&q->lock, NULL);
}

/* Return non-zero if the queue is empty */
int empty(struct pqueue_t *q)
{
    return (q->head == NULL);
}

/* Get PCB of a process from the queue (q).
 * Return NULL if the queue is empty */
struct pcb_t *de_queue(struct pqueue_t *q)
{
    struct pcb_t *proc = NULL;
    // TODO: return q->head->data and remember to update the queue's head
    // and tail if necessary. Remember to use 'lock' to avoid race
    // condition
    pthread_mutex_lock(&q->lock);

    // if empty (non zero) --> return
    if (empty(q))
    {
        pthread_mutex_unlock(&q->lock);
        return proc;
    }

    // store data to return
    proc = q->head->data;
```

```

    struct qitem_t *tmp = q->head;
    // update head
    q->head = q->head->next;
    free(tmp);
    // if head == NULL, then tail == NULL
    if (empty(q))
    {
        q->tail = NULL;
    }

    pthread_mutex_unlock(&q->lock);
    // YOUR CODE HERE

    return proc;
}

/* Put PCB of a process to the queue. */
void en_queue(struct pqueue_t *q, struct pcb_t *proc)
{
    // TODO: Update q->tail.
    // Remember to use 'lock' to avoid race condition
    // YOUR CODE HERE
    pthread_mutex_lock(&q->lock);

    struct qitem_t *node = (struct qitem_t *)malloc(sizeof(struct qitem_t));
    node->data = proc;
    if (empty(q))
    {
        q->head = node;
        q->tail = node;
        q->tail->next = NULL;
        pthread_mutex_unlock(&q->lock);
        return;
    }

    q->tail->next = node;
    q->tail = node;
    q->tail->next = NULL;
    pthread_mutex_unlock(&q->lock);
    return;
}

```

Source code of Exercise 2.2 (sched.c)

```

#include "queue.h"
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

```

```

#define TIME_UNIT 100 // In microsecond

static struct pqueue_t in_queue;    // Queue for incoming processes
static struct pqueue_t ready_queue; // Queue for ready processes

static int load_done = 0;

static int timeslot; // The maximum amount of time a process is allowed
                    // to be run on CPU before being swapped out

// Emulate the CPU
void *cpu(void *arg)
{
    int timestamp = 0;
    /* Keep running until we have loaded all process from the input file
     * and there is no process in ready queue */
    while (!load_done || !empty(&ready_queue))
    {
        /* Pickup the first process from the queue */
        struct pcb_t *proc = de_queue(&ready_queue);
        if (proc == NULL)
        {
            /* If there is no process in the queue then we
             * wait until the next time slice */
            timestamp++;
            usleep(TIME_UNIT);
        }
        else
        {
            /* Execute the process */
            int start = timestamp; // Save timestamp
            int id = proc->pid;     // and PID for tracking
            /* Decide the amount of time that CPU will spend
             * on the process and write it to 'exec_time'.
             * It should not exceed 'timeslot'.
             */
            int exec_time = 0;

            // TODO: Calculate exec_time from process's PCB

            // YOUR CODE HERE

            // choose smaller time between burst time & time slot
            exec_time = proc->burst_time > timeslot ? timeslot : proc-
>burst_time;

            // update burst time
            proc->burst_time -= exec_time;

            /* Emulate the execution of the process by using

```

```

        * 'usleep()' function */
        usleep(exec_time * TIME_UNIT);

        /* Update the timestamp */
        timestamp += exec_time;

        // TODO: Check if the process has terminated (i.e. its
        // burst time is zero. If so, free its PCB. Otherwise,
        // put its PCB back to the queue.

        // YOUR CODE HERE
        if (proc->burst_time == 0)
        {
            free(proc);
        }
        else
        {
            enqueue(&ready_queue, proc);
        }

        /* Track runtime status */
        printf("%2d-%2d: Execute %d\n", start, timestamp, id);
    }
}

// Emulate the loader
void *loader(void *arg)
{
    int timestamp = 0;
    /* Keep loading new process until the in_queue is empty*/
    while (!empty(&in_queue))
    {
        struct pcb_t *proc = dequeue(&in_queue);
        /* Loader sleeps until the next process available */
        int wastetime = proc->arrival_time - timestamp;
        usleep(wastetime * TIME_UNIT);
        /* Update timestamp and put the new process to ready queue */
        timestamp += wastetime;
        enqueue(&ready_queue, proc);
    }
    /* We have no process to load */
    load_done = 1;
}

/* Read the list of process to be executed from stdin */
void load_task()
{
    int num_proc = 0;
    scanf("%d %d\n", &timeslot, &num_proc);
    int i;

```

```

for (i = 0; i < num_proc; i++)
{
    struct pcb_t *proc =
        (struct pcb_t *)malloc(sizeof(struct pcb_t));
    scanf("%d %d\n", &proc->arrival_time, &proc->burst_time);
    proc->pid = i;
    en_queue(&in_queue, proc);
}
//printf("%s", "ENQUEUE\n");
}

int main()
{
    pthread_t cpu_id;    // CPU ID
    pthread_t loader_id; // LOADER ID

    /* Initialize queues */
    initialize_queue(&in_queue);
    initialize_queue(&ready_queue);

    /* Read a list of jobs to be run */
    load_task();
    printf("%s", "finish loading input\n");

    /* Start cpu */
    pthread_create(&cpu_id, NULL, cpu, NULL);
    /* Start loader */
    pthread_create(&loader_id, NULL, loader, NULL);

    /* Wait for cpu and loader */
    pthread_join(cpu_id, NULL);
    pthread_join(loader_id, NULL);

    pthread_exit(NULL);
}

```

III. Homework (3.1, 3.2, 3.3)

1. *Requirement*

- Updating the previous emulator to support preemptive Priority algorithms while the above Round Robin policy does not care about the process priority.

1. Including an additional priority field as the third column in the input.txt as the described.

2. Updating the process struct definition.

3. Changing the original Round Robin algorithm to the Preemptive Priority.

When a process is pushed back to the **ready_queue**, its priority will be decreased by 1 if the focus is more significant than 0.

2. *Steps to build*

First of all, let's talk about the Round Robin:

- RR (Round Robin) scheduling: The development of the RR scheduling algorithm focused on time-sharing systems, which give each work a time slot, time slice, or quantum (its allotment of CPU time), and interrupt the job if it is not finished by then. When a time window is allocated for that process again, the work is resumed. The scheduler chooses the first process in the ready queue to run if the process terminates or switches to waiting during the time quantum assigned to it. A process that produced huge jobs would be preferred over other processes in the absence of time-sharing or if the quanta were enormous in comparison to the sizes of the tasks.

- The Round Robin algorithm is a preemptive algorithm as the scheduler forces the process out of the CPU once the time quota expires. And a time quantum typically lasts between 10 and 100 milliseconds.

In this issue, we need adding **priority variable** and implementing the **en_queue_prior** function in order to the **ready_queue** is the priority queue (the higher priority process will be placed before the lower one in the sequence determined by each process's priority).

As in the previous situation, the **in_queue** is still a standard queue (FIFO). To avoid starvation (where a process ready for the CPU (resources) can wait to run indefinitely due to low priority).

The CPU() function differs in that if the process does not complete its task, its priority will be decreased by 1 if it is greater than 0, and then added to the priority queue by the **en_queue_prior** process.

3. *Explain each code paragraph*

Depend on the source code in Exercise 2.2 to do this task:

Adding **priority variable** type integer to struct **pcb_t** in file **structs.h**:

```
struct pcb_t {  
    /* Values initialized for each process */  
    int arrival_time; // The timestamp at which process arrives  
    // and wishes to start  
    int burst_time; // The amount of time that process requires  
    // to complete its job  
    int pid; // process id  
    int priority;  
};
```

Next, generating 1 more function **en_queue_prior** to alter the original Round Robin algorithm to the Preemptive Priority.


```

void en_queue_prior(struct pqueue_t * q, struct pcb_t * proc)
{
    // TODO: Update q->tail.
    // Remember to use 'lock' to avoid race condition
    // YOUR CODE HERE
    pthread_mutex_lock(&q->lock);
    struct qitem_t *node = (struct qitem_t *)malloc(sizeof(struct qitem_t));
    node->data = proc;
    if (empty(q))
    {
        q->head = node;
        q->tail = node;
        node->next=NULL;
    }
    else
    {
        struct qitem_t *prev = NULL, *curr = q->head;
        while (curr && curr->data->priority >= node->data->priority)
        {
            prev = curr;
            curr = curr->next;
        }
        if (!prev)
        {
            node->next = q->head;
            q->head = node;
        }
        else
        {
            node->next = prev->next;
            prev->next = node;
        }
        if (!node->next) q->tail = node;
    }
    pthread_mutex_unlock(&q->lock);
    return;
}

```

After updating the timestamp, to prevent starvation, add one additional condition that if priority is higher than 0, priority should be lowered by one.

```
// YOUR CODE HERE
if(proc->burst_time==0)
{
    free(proc);
}
else
{
    if(proc->priority>0)
    {
        proc->priority-=1;
    }
    en_queue_prior(&ready_queue,proc);
}
```

The `en_queue(&ready_queue, proc)` function has been changed to the `en_queue_prior(&ready_queue, proc)` mechanism in the loader function so that the `ready_queue` is now the priority queue.

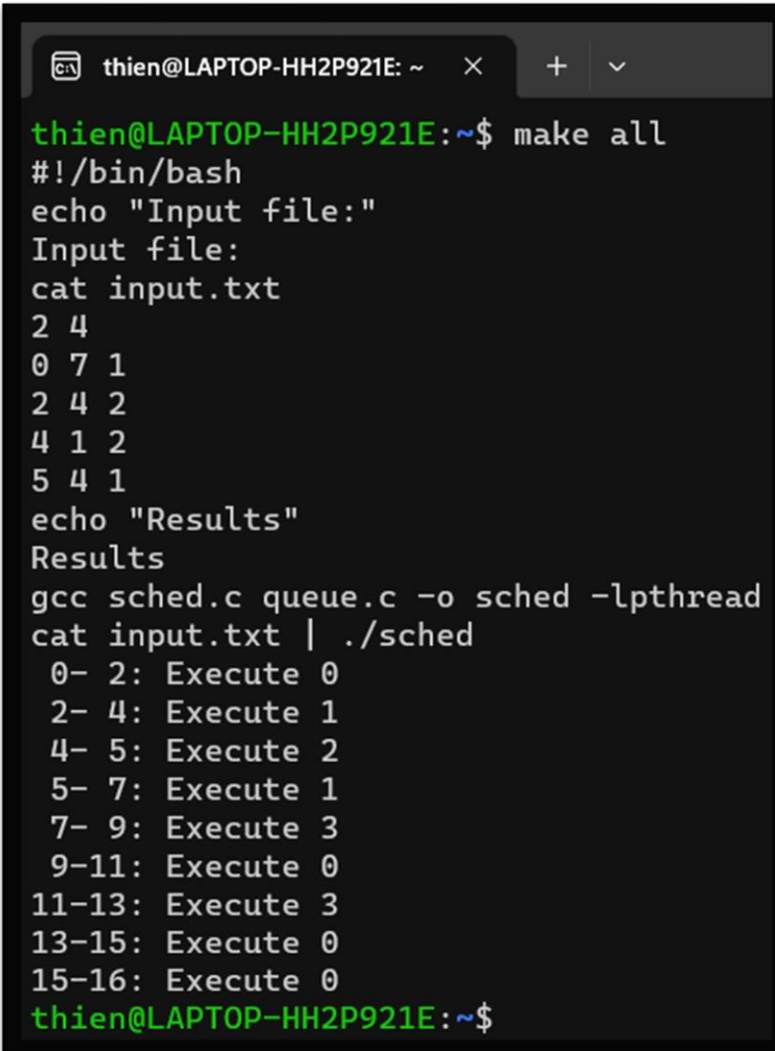
```
void * loader(void * arg) {
    int timestamp = 0;
    /* Keep loading new process until the in_queue is empty*/
    while (!empty(&in_queue)) {
        struct pcb_t * proc = de_queue(&in_queue);
        /* Loader sleeps until the next process available */
        int wastetime = proc->arrival_time - timestamp;
        usleep(wastetime * TIME_UNIT);
        /* Update timestamp and put the new process to ready queue */
        timestamp += wastetime;
        en_queue_prior(&ready_queue, proc);
    }
    /* We have no process to load */
    load_done = 1;
}
```

Then, using the same **makefile** in the previous exercise:

```
all:
    #!/bin/bash
    echo "Input file:"
    cat input.txt
    echo "Results"
    gcc sched.c queue.c -o sched -lpthread
    cat input.txt | ./sched
clean:
    rm -f all
```

4. *Implementing*

Results:



```
thien@LAPTOP-HH2P921E: ~$ make all
#!/bin/bash
echo "Input file:"
Input file:
cat input.txt
2 4
0 7 1
2 4 2
4 1 2
5 4 1
echo "Results"
Results
gcc sched.c queue.c -o sched -lpthread
cat input.txt | ./sched
0- 2: Execute 0
2- 4: Execute 1
4- 5: Execute 2
5- 7: Execute 1
7- 9: Execute 3
9-11: Execute 0
11-13: Execute 3
13-15: Execute 0
15-16: Execute 0
thien@LAPTOP-HH2P921E: ~$
```

The outcome is following Gantt chart below:

P0	P1	P2	P1	P3	P0	P3	P0	P0	
0	2	4	5	7	9	11	13	15	16

As can be seen, the outcome corresponds to the Gantt chart. We will schedule processes in the queue in a manner consistent with their priority if two processes have the same priority like FCFS (First come, First served).

Source code of 3.1, 3.2, 3.3

```
// queue.h
#ifndef QUEUE_H
#define QUEUE_H

#include "structs.h"

/* Initialize the process queue */
void initialize_queue(struct pqueue_t * q);

/* Get a process from a queue */
struct pcb_t * de_queue(struct pqueue_t * q);

/* Put a process into a queue */
void en_queue(struct pqueue_t * q, struct pcb_t * proc);
void en_queue_prior(struct pqueue_t * q, struct pcb_t * proc);

int empty(struct pqueue_t * q);

#endif

/* ##### */
//queue.c
#include <stdlib.h>
#include "queue.h"
#include <pthread.h>

/* Remember to initilize the queue before using it */
void initialize_queue(struct pqueue_t * q)
{
    q->head = q->tail = NULL;
    pthread_mutex_init(&q->lock, NULL);
}

/* Return non-zero if the queue is empty */
int empty(struct pqueue_t * q)
{
    return (q->head == NULL);
}

/* Get PCB of a process from the queue (q).
 * Return NULL if the queue is empty */
struct pcb_t * de_queue(struct pqueue_t * q)
```

```

{
    struct pcb_t * proc = NULL;
    // TODO: return q->head->data and remember to update the queue's head
    // and tail if necessary. Remember to use 'lock' to avoid race
    // condition
    // YOUR CODE HERE
    pthread_mutex_lock(&q->lock);

    // if empty (non zero) --> return
    if(empty(q))
    {
        pthread_mutex_unlock(&q->lock);
        return proc;
    }

    // store data to return
    proc = q->head->data;
    struct qitem_t *tmp = q->head;
    // update head
    q->head=q->head->next;
    free(tmp);
    // if head == NULL, then tail == NULL
    if(empty(q))
        q->tail=NULL;

    pthread_mutex_unlock(&q->lock);
    return proc;
}
/* Put PCB of a process to the queue. */
void en_queue(struct pqueue_t * q, struct pcb_t * proc)
{
    // TODO: Update q->tail.
    // Remember to use 'lock' to avoid race condition
    // YOUR CODE HERE
    pthread_mutex_lock(&q->lock);

    struct qitem_t *node = (struct qitem_t *)malloc(sizeof(struct qitem_t));
    node->data = proc;

    if (empty(q))
    {
        q->head = node;
        q->tail = node;
        node->next=NULL;
    }
    else
    {
        q->tail->next = node;
        q->tail = node;
        node->next = NULL;
    }
}

```

```

    pthread_mutex_unlock(&q->lock);
    return;
}
/* Put PCB of a process to the queue. */
void en_queue_prior(struct pqueue_t * q, struct pcb_t * proc)
{
    // TODO: Update q->tail.
    // Remember to use 'lock' to avoid race condition
    // YOUR CODE HERE
    pthread_mutex_lock(&q->lock);
    struct qitem_t *node = (struct qitem_t *)malloc(sizeof(struct qitem_t));
    node->data = proc;
    if (empty(q))
    {
        q->head = node;
        q->tail = node;
        node->next=NULL;
    }
    else
    {
        struct qitem_t *prev = NULL, *curr = q->head;
        while (curr && curr->data->priority >= node->data->priority)
        {
            prev = curr;
            curr = curr->next;
        }
        if (!prev)
        {
            node->next = q->head;
            q->head = node;
        }
        else
        {
            node->next = prev->next;
            prev->next = node;
        }
        if (!node->next) q->tail = node;
    }
    pthread_mutex_unlock(&q->lock);
    return;
}

/* ##### */
// sched.c
#include "queue.h"
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

```

```

#define TIME_UNIT    100 // In microsecond

static struct pqueue_t in_queue; // Queue for incoming processes
static struct pqueue_t ready_queue; // Queue for ready processes

static int load_done = 0;

static int timeslot; // The maximum amount of time a process is allowed
                    // to be run on CPU before being swapped out
// Emulate the CPU
void * cpu(void * arg) {
    int timestamp = 0;
    /* Keep running until we have loaded all process from the input file
     * and there is no process in ready queue */
    while (!load_done || !empty(&ready_queue)) {
        /* Pickup the first process from the queue */
        struct pcb_t * proc = de_queue(&ready_queue);
        if (proc == NULL) {
            /* If there is no process in the queue then we
             * wait until the next time slice */
            //timestamp++;
            usleep(TIME_UNIT);
        }else{
            /* Execute the process */
            int start = timestamp; // Save timestamp
            int id = proc->pid; // and PID for tracking
            /* Decide the amount of time that CPU will spend
             * on the process and write it to 'exec_time'.
             * It should not exceed 'timeslot'.
             */
            int exec_time = 0;

            // TODO: Calculate exec_time from process's PCB

            // YOUR CODE HERE

            // choose smaller time between burst time & time slot
            exec_time = proc->burst_time > timeslot ? timeslot : proc->burst_time;
            // update burst time
            proc->burst_time -= exec_time;

            /* Emulate the execution of the process by using
             * 'usleep()' function */
            usleep(exec_time * TIME_UNIT);

            /* Update the timestamp */
            timestamp += exec_time;

            // TODO: Check if the process has terminated (i.e. its
            // burst time is zero. If so, free its PCB. Otherwise,
            // put its PCB back to the queue.

```

```

        // YOUR CODE HERE
        if(proc->burst_time==0)
        {
            free(proc);
        }
        else
        {
            if(proc->priority>0)
            {
                proc->priority-=1;
            }
            enqueue_prior(&ready_queue,proc);
        }
        /* Track runtime status */
        printf("%2d-%2d: Execute %d\n", start, timestamp, id);
    }
}

// Emulate the loader
void * loader(void * arg) {
    int timestamp = 0;
    /* Keep loading new process until the in_queue is empty*/
    while (!empty(&in_queue)) {
        struct pcb_t * proc = dequeue(&in_queue);
        /* Loader sleeps until the next process available */
        int wastetime = proc->arrival_time - timestamp;
        usleep(wastetime * TIME_UNIT);
        /* Update timestamp and put the new process to ready queue */
        timestamp += wastetime;
        enqueue_prior(&ready_queue, proc);
    }
    /* We have no process to load */
    load_done = 1;
}

/* Read the list of process to be executed from stdin */
void load_task() {
    int num_proc = 0;
    scanf("%d %d\n", &timeslot, &num_proc);
    int i;
    for (i = 0; i < num_proc; i++) {
        struct pcb_t * proc =
            (struct pcb_t *)malloc(sizeof(struct pcb_t));
        scanf("%d %d %d\n", &proc->arrival_time, &proc->burst_time,&proc->priority);
        proc->pid = i;
        enqueue(&in_queue, proc);
    }
}

```



```

int main() {
    pthread_t cpu_id;    // CPU ID
    pthread_t loader_id; // LOADER ID

    /* Initialize queues */
    initialize_queue(&in_queue);
    initialize_queue(&ready_queue);

    /* Read a list of jobs to be run */
    load_task();

    /* Start cpu */
    pthread_create(&cpu_id, NULL, cpu, NULL);
    /* Start loader */
    pthread_create(&loader_id, NULL, loader, NULL);

    /* Wait for cpu and loader */
    pthread_join(cpu_id, NULL);
    pthread_join(loader_id, NULL);

    pthread_exit(NULL);
}

/* ##### */
// structs.h
#ifndef STRUCTS_H
#define STRUCTS_H

#include <pthread.h>

/* The PCB of a process */
struct pcb_t {
    /* Values initialized for each process */
    int arrival_time; // The timestamp at which process arrives
                      // and wishes to start
    int burst_time;   // The amount of time that process requires
                      // to complete its job
    int pid;          // process id
    int priority;
};

/* 'Wrapper' of PCB in a queue */
struct qitem_t {
    struct pcb_t * data;
    struct qitem_t * next;
};

/* The 'queue' used for both ready queue and in_queue (e.g. the list of
 * processes that will be loaded in the future) */
struct pqueue_t {

```

```
/* HEAD and TAIL for queue */
struct qitem_t * head;
struct qitem_t * tail;
/* MUTEX used to protect the queue from
 * being modified by multiple threads*/
pthread_mutex_t lock;
};

#endif
```