Operating Systems – spring 2022
Assignment 3

Instructor: Hans P. Reiser

Submission Deadline: Monday, February 7, 2022 – 23:59

A new assignment will be published every week, right after (or a bit before) the last one was due. It must be completed before its submission deadline. (Hard deadlines, no extension.)

T-Questions are theory homework assignments. **The answers to the assignments must be uploaded to Canvas (in the quiz).**

P-Questions are programming assignments. Download the provided template from Canvas. Do not fiddle with the compiler flags. Submission instructions can also be found in first assignment.

The topic of this assignment are scheduling basics and policies.

T-Question 3.1: Scheduling

a. Use the example of a scheduler and a dispatcher to explain the distinction between policy and mechanism. **1 T-pt**

b. Briefly explain the difference between cooperative and preemptive scheduling! What is the main problem (and thus disadvantage) of cooperative scheduling, compared to preemptive scheduling? **2 T-pt**

c. With lottery scheduling, every process is assigned a certain number of tickets. To make a scheduling decision the lottery scheduler randomly chooses a ticket and selects the process that owns the ticket. Briefly explain how lottery scheduling can be implemented without allocating any dedicated objects per ticket such as structs or array elements. **2 T-pt**

d. What is the scheduling sequence (e.g., P_X, P_Y, P_Z,...) for the following processes with round robin scheduling and a timeslice length of 1 time unit? The scheduler first adds new processes (if any) to the tail of the ready queue and then inserts the previous process to the tail (if it is still runnable). **2 T-pt**

Process	Burst length	Arrival time
P_1	3	0
P_2	5	2
P_3	2	4

e. Calculate the average waiting time for the example in 3.1d. **1 T-pt**

f. Consider the same set of processes as in part 3.1d, but now with a FCFS scheduler. What is the scheduling sequence, and what is the average waiting time? **2 T-pt**

P-Question 3.1: Priority Scheduler

Download the template **p1** for this assignment from Canvas. You may only modify and upload the file `scheduler.c`.

Priority scheduling assigns each scheduling entity (i.e., a process or thread) a priority. For each priority the scheduler has a ready queue into which ready threads with the respective priority are enqueued. The scheduler always selects the first thread from the non-empty ready queue of the highest priority. If multiple threads have the same priority (i.e., a queue contains more than one thread), the scheduler employs round robin scheduling within the queue.

Refer to the lecture slides for more details. In this question you will write your own priority scheduler. We assume in the following that all thread control blocks are stored in a static array. The thread ID is the index of a thread in that array. The scheduler queues store only the thread ID.

a. The queue implementation in the template already contains the necessary structures to represent a queue (`Queue`) and its elements (`QueueItem`). The queue contains a `head` and a `tail` pointer to make it possible to access both the first and the last item in O(1). Implement the functions to add and remove elements. You can use the following guideline: **3 P-pt**

_enqueue Adds a new item to the queue's *tail*.

- Allocates a new `QueueItem` with `malloc` (silently ignores errors, i.e. does nothing)
- Assigns the supplied `data` (int, used for thread ID) to the new item
- Adds the new item to the tail of the queue by updating the `head` (if necessary) and `tail` pointers as well as the `next` pointer of the current tail element (if any)

_dequeue Removes an item from the queue's *head*.

- Returns -1 if the queue is empty
- Otherwise, removes the first item from the queue's head by updating all necessary pointers
- Frees the item with `free`
- Returns the `data` field of the removed item (Caution: Remember that you cannot (must not!) access the item anymore after freeing it!)

Hints: If the queue is empty `head` should be `NULL`. You may also set the `next` pointer of the last element to `NULL`.

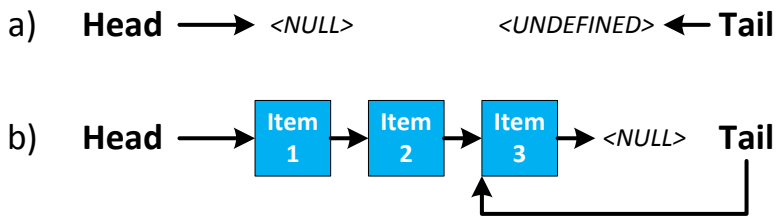

Figure 1: Example queue. a) Empty queue b) Queue with 3 items

```
void _enqueue(Queue *queue, int data);
int _dequeue(Queue *queue);
```

Note to a): It is important that you are able to solve this type of straight C programming questions without a big hassle. Following assignments will assume you can handle pointers in C. If you have a hard time getting the queue to work, consider investing some time reading one of the many good books on the C programming language (e.g., Brian W. Kernigham and Dennis M Ritchie *Programming in C*). You won't regret it! Also, the tutorials usually also contain useful programming examples.

b. Implement the event handler functions, which set the supplied thread's state and if necessary add the thread to the appropriate ready queue. Set the `QueueItem`'s data field to the thread's id.

3 P-pt

- `void onThreadReady(int threadId)` is called if a thread in waiting state becomes ready (e.g., thread was blocked on an I/O operation, and the I/O operation has finished). The thread needs to be placed in the appropriate runqueue.
- `void onThreadPreempted(int threadId)` is called if a thread that was running was preempted. It also needs to be placed in the right runqueue, as it is ready to continue.
- `void onThreadWaiting(int threadId)` is called when a thread blocks (e.g., on an I/O operation). Such a thread enters the waiting state and will not be part of any runqueue.

c. Implement the priority scheduling policy, with an additional rule for the prevention of starvation. Your scheduling function should perform the following basic operations whenever a new thread needs to be selected:

4 P-pt

- Find the ready queue with the highest priority that contains a ready thread
- Remove the first thread from the queue, updates its state and returns its thread id

Starvation prevention is done with the following additional rule:

- If the scheduler selects threads with a certain priority more then 5 times without selecting a thread with a lower priority, the scheduler resorts to the next lower priority queue that (1) is not empty and (2) does not break this starvation rule (i.e., exceeding the 5 times maximum without scheduling lower priority threads).

Hints: The simplest solution to the starvation prevention will use a recursive approach to determine the right queue for thread selection.

```
int scheduleNextThread();
```

Total:
10 T-pt
10 P-pt