Operating Systems – spring 2025
Tutorial-Assignment 3

Instructor: Hans P. Reiser

Submission Deadline: Monday, February 3, 2025 – 23:59

A new assignment will be published every week. It must be completed before its submission deadline (late policy for programming assignments: up to two days, 10% penalty/day)

Lab Exercisess are theory and programming exercises discussed in the lab class. They are not graded, but should help you solve the graded questions and prepare for the final exam. Make sure to read and think about possible solutions before the lab class.

T-Questions are theory homework assignments and need to be answered directly on Canvas (quiz).

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

The topics of this assignment are processes and scheduling basics.

Plan for this lab class

- Lab 3.1 is revisiting theory from the lecture. This might be useful for some of the T3 questions, but more importantly, for exam preparation. You should be able to answer all these questions in the exam.
- Lab 3.2, the implementation of a linked list, and Lab 3.3, a simple round-robin scheduler, is the basis for assignment P3.
- In the remaining lab time, you should work on assignment P3, and the TA's will help you if you are stuck somewhere. Ideally you read the assignment and download the code templates before the class.

Lab 3.1: Scheduling Basics

a. What is the purpose of scheduling?

b. What is the difference between a long and short-term scheduler?

c. Consider an operating system that supports the five task states “new”, “running”, “ready”, “waiting”, and “terminated”. Depict the possible state transitions and the events that cause them.

d. What quantitative metrics/criteria can be used to estimate the quality of a scheduling policy?

e. What kind of hardware support is required for an operating system that implements a non-cooperative scheduling policy?

f. Discuss pros and cons of choosing a short timeslice length vs. choosing a longer timeslice length. What are common values for the length of a timeslice?

Lab 3.2: Link list implementation in C

a. Implement a linked list in C, using the following data structures:

```
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>

struct Task { int pid; };

struct QueueItem {
    struct QueueItem *next;
    void *task;
};

struct QueueItem *listHead = NULL;

void appendItem(void *task)
{
    //...implement this
    // append to the end of the list
}

void *removeFirstItem()
{
    //implement this
    // removes the first list item from the list and returns its value; returns NULL if list is empty
}

int containsItem(void *task)
{
    //implement this
    // return true(1) if list contains value, false(0)
}

int isEmpty() {
    // implement this
    // return true (1) if list is empty, false (0) otherwise
}

struct Task t1 = {42}, t2 = {4711}, t3 = {123};
int main() {
    appendItem(&t1);
    appendItem(&t2);
    struct Task *next = removeFirstItem();
    printf("Next tid: %d\n", next==NULL ? -1 : next->pid);
    appendItem(&t1);
    appendItem(&t3);
    for(int i=0; i<5; i++) {
        struct Task *next = removeFirstItem();
        printf("Next tid: %d\n", next==NULL ? -1 : next->pid);
    }
}
```

You should implement the empty functions in the template.

Lab 3.3: Scheduler implementation (simple round robin)

a. Using the list (queue) implementation from Part 3.2, implement a simple scheduler with the interface as defined in the file `scheduler.h`

- Remove the simple definition of `struct Task` from your code and add instead an `#include "scheduler.h"`.
- Implement the function `scheduleNextTask()`. This function should return the first element ofthe queue (or NULL if the queue is empty).
- Implement the function `onTaskReady(Task *task)`. This will set a task into the "ready" state (i.e., this means that you have to append the task to your ready queue.)

```
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>

#include "scheduler.h"

/*
 * Put your list implementation here (without the definition on struct Task,
 * this structure is now defined in scheduler.h
 */

/* .... */

/*
 * Skeleton of scheduler implementation, you have to complete these two functions
 * The functions should
 * (a) add or remove tasks from your queue, as appropriate
 * (b) change the task state ("state" element in struct Task), as appropriate
 */

Task *scheduleNextTask()
{
    // TODO: implement
}

void onTaskReady(Task *task) {
    // TODO: implement
}

/*
 * New main program for round robin scheduler
 */

// state, taskid, priority, name
static Task tasks[] = {
 { STATE_NEW, 0, 0, "A" },
 { STATE_NEW, 1, 0, "B" },
 { STATE_NEW, 2, 0, "C" },
 { STATE_NEW, 3, 0, "D" },
};

int main() {
    // Let's indicate that all four tasks are ready
    onTaskReady(&tasks[0]);
    onTaskReady(&tasks[1]);
    onTaskReady(&tasks[2]);
    onTaskReady(&tasks[3]);

    // Now let's look at 16 iterations....
    // let the schedular schedule a task
    // ... the tasks run
    // ... tell the scheduler that the task has been preempted (by calling onTaskPreempted)
    for(int i=0; i<16; i++) {
        struct Task *task = scheduleNextTask();
        printf("Next task id: %d\n", task==NULL ? -1 : task->pid);
        onTaskReady(task);
    }
}
```

1 T-Questions (graded quiz on canvas)

T-Question 3.1: Scheduling

a. T3.1a: Use the example of a scheduler and a dispatcher to explain the distinction between a policy and a mechanism. **1 T-pt**

b. T3.1b: Which event can trigger a process switch in preemptive scheduling but not in cooperative scheduling? **1 T-pt**

c. Processes and scheduling: Are the following statements true or false? (correctly marked: 0.5P) **2 T-pt**

true	false	
☐	☐	A process that is currently not running on a CPU (i.e., a process waiting in some waitqueue), is called a zombie process.
☐	☐	In preemptive scheduling, the operating system may use timer interrupts to preempt running processes.
☐	☐	One advantage of cooperative scheduling is that it leads to process starvation when a long-running process does not yield control voluntarily.
☐	☐	Round Robin scheduling can be used for cooperative, but not for preemptive scheduling.

d. In the life cycle between process creation and process termination, a process can transition multiple times between which three states? **1 T-pt**

e. 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 2 time units? 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	4	2
P_3	3	5

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

g. 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/task/thread) a priority. For each priority the scheduler has a ready queue into which ready task with the respective priority are enqueued. The scheduler always selects the first task from the non-empty ready queue of the highest priority. If multiple tasks have the same priority (i.e., a queue contains more than one task), the scheduler employs round robin scheduling within the queue. Refer to the lecture slides for more details.

In this assignment, you will replace the simple scheduler from the lab tutorial (Lab 3.3) with a more complex priority scheduler. We assume in the following that the scheduler stores pointers to Task structures.

You can reuse (and extend) the queue implementation from the tutorial part.

a. The queue implementation of the tutorial already offers functions to append elements at the tail and remove elements from the head. For the assignment, you have to make two modifications:

   - Your code must be extended to handle multiple queues. This means that you need to add a parameter (pointer) to the queue functions that indicates the queue to use (This part is mandatory as basis for the second part of the assignment).
   - You should optimize the queue using an additional tail pointer, so that all operations (adding and removing items) can execute in constant time ($O(1)$). (You can use the queue without this optimization for implementing the second part).

   This is a complete specification of the expected functionality: **3 P-pt**

   appendItem 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` (a `void *` pointer, used to store a `Task *` by the scheduler) 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)

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

   - Returns NULL 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`.

```
void appendItem(Queue *queue, void *data);
void *removeFirstItem(Queue *queue);
```

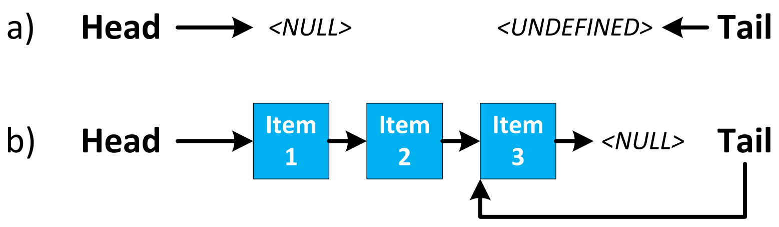

Figure 1: Example queue. a) Empty queue b) Queue with 3 items

b. Implement/modify the event handler functions, which set the supplied task's state and if necessary add the task to the appropriate ready queue. **1 P-pt**

- `void onTaskReady(Task *task)` is called if a thread in waiting state becomes ready (e.g., thread was blocked on an I/O operation before, and the I/O operation has finished). The thread needs to be placed in the appropriate queue.
- `void onTaskPreempted(Task *task)` is called if a running thread was preempted. It also needs to be placed in the right queue, as it is ready to continue.
- `void onTaskWaiting(Task *task)` is called when a task blocks (e.g., on an I/O operation). Such a task enters the waiting state and will not be part of any queue.

c. Implement the basic priority scheduling policy. Your scheduling function should perform the following basic operations whenever a new task needs to be selected: **4 P-pt**

- Find the ready queue with the highest priority that contains a ready task
- Remove the first task from the queue, updates its state and return a pointer to the task structure

d. Add starvation prevention to the scheduler using the following additional rule: **2 P-pt**

- If a t with priority P has been selected for four times without giving a lower-priority task the chance to run, then instead of selecting a task with priority P again, the scheduler will resort to the next lower priority queue that (1) is not empty and (2) does not break this starvation rule (i.e., exceeding the 4 times maximum without scheduling lower priority tasks).
- Example: Assume that 0 is the highest, 10 is the lowests priority, and you have tasks with priorities $7, 6$ and 4; these tasks do not block, i.e. after running they immediately return to the ready queue. In this case, the schedule will be:
"4 4 4 4 6 4 4 4 4 6 4 4 4 4 6 4 4 4 4 6 4 4 4 4 7 4 ..."

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

```
Task *scheduleNextTask();
```

Total:
10 T-pt
10 P-pt