Operating Systems – spring 2023
Tutorial-Assignment 3

Instructor: Hans P. Reiser

Submission Deadline: Monday, January 30, 2023 – 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.

Question 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?

Question 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 ListItem {
    struct ListItem *next;
    int value;
};

struct ListItem *listHead = NULL;

void appendItem(int valie) {
     // ... implement this
     // append at the end of the list
}

int removeFirstItem() {
    // implement this
    // removes the first item from the list and returns its value; returns -1 if list is empty
    return -1;
}

int containsItem(int value) {
    // implement this
    // return true (1) if list contains value, false (0) if not
    return 0;
}

int isEmpty() {
    // implement this
    // return true (1) if list is empty, false (0) otherweise
    return 0;
```

```
}

int main() {
    appendItem(42);
    appendItem(4711);
    removeFirstItem();
    appendItem(42);
    appendItem(4);
    for(int i=0; i<5; i++) printf("%d\n", removeFirstItem());
}
```

You should implement the empty functions in the template.

1 T-Questions (graded quiz on canvas)

T-Question 3.1: Scheduling

a. How would you explain the relationship between a scheduler and a dispatcher in the context of operating systems? Provide an illustration of how the concepts of "policy" and "mechanism" are applied in that relationship. **2 T-pt**

b. 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 queue (i.e., a process waiting in some waitqueue), is called a zombie process.
☐	☐	In cooperative scheduling, the operating system cooperates with the interrupt controller (APIC) in order to preempt processes.
☐	☐	One disadvantage of cooperative scheduling is that it can lead to process starvation when a long-running process does not yield control voluntarily.
☐	☐	Round Robin scheduling minimizes the average turn around time.

c. In the life cycle between process creation and process termination, a process can transition multiple times between which three states? **1 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	4	2
P_3	3	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: A Simple Buffer Overflow Exploit

Use the files in folder **p1** of the assignment template. You should only modify and upload the file `exploit_program.c` of P-Question 2.2.

Writing programs in C is risky, because simple mistakes can lead to security vulnerabilities. This programming assignment involves exploiting a security vulnerability in a program called `vulnerable.c`. This program is an example of a program that is vulnerable to buffer overflow attacks. The normal usage of the program is to invoke it with two parameters on the command line: a secret password and a message. The program checks if the supplied password is correct, and if it is, the message is written to a file called `success.txt`.

a. The template in `exploit_program.c` is a program that invokes the target program with a password and a message. Your task is to modify the template and make it use buffer overflow techniques to bypass the password check. **3 P-pt**

- Start by examining the `vulnerable.c` program code (included in the template). Study the variables and their location in memory (heap, stack).
- Find an approach that bypasses the password check.
- Implement an exploit in `exploit_program.c` that invokes the `vulnerable` program (the exact path to the executable file is passed as `program` parameter, as well as a value for `message`). The effect of invoking your new version of the function should basically be the same as that of the template version, but without the need for knowing the secret password.

```
int exploit_program(char *program, char *message);
```

- Your function should return the exit code of the vulnerable program that you started.
- Buffer overflow exploits are system dependent. We expect your solution to work on Linux / x86-64.

P-Question 3.2: Simple FIFO Scheduler

Download the template **p2** for this assignment from Canvas. You may only modify and upload the file `scheduler.c`. In this assignment, you will implement a simple FIFO process scheduler.

To implement the scheduler for this assignment, you will need to create data structures to store the ready queue of processes (no need to manage a waiting queue for now).

- The `threadReady()` function is invoked by the operating system to
  (a) add a new process to the ready queue, and
  (b) add a process currently contained in the waiting queue to the ready queue.
- The `threadPreempted()` function is invoked by the operating system to add a preempted process (that was running on the CPU) to the ready queue,
- The `threadWaiting()` function is invoked by the operating system if the currently running process makes a blocking I/O system call.

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. if malloc fails, _enqueue does not add anything to the queue)
- 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`.

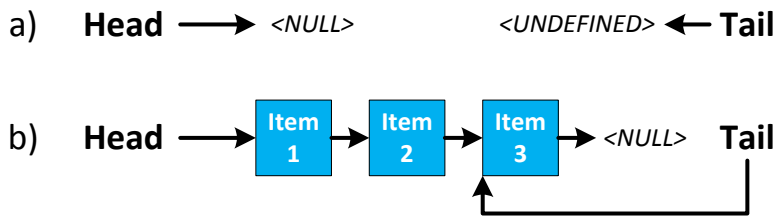

Figure 1: Example queue. a) Empty queue b) Queue with 3 items

b. Implement the event handler functions. These functions are supposed to be invoked by the operating system to inform the scheduler about state changes of a process, and the task of your implemention is to update the queues of your scheduler accordingly. **4 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 function is also called if a new thread is created. The thread needs to be placed in the ready queue.
- `void onThreadPreempted(int threadId)` is called if a thread that was running was preempted. It also needs to be placed on the ready queue, as it is ready to continue.
- `void onThreadWaiting(int threadId)` is called when a thread blocks (e.g., on an I/O operation). No action required besides updating the thread state (see below).
- `int scheduleNextThread();` is called to schedule the next thread. Your scheduler should return -1 if the ready queue is currently empty. Otherwise, it should remove the head of the queue and return the thread ID of the removed element as thread to be executed next.

All functions should update the thread state in the entry in the global tread array corresponding to the thread ID (READY/RUNNING/WAITING).

Total: 10 T-pt 10 P-pt