Operating Systems – spring 2023
Tutorial-Assignment 4

Instructor: Hans P. Reiser

Submission Deadline: Monday, Februar 6th, 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.

Topics of this assignment are threads and segmentation, and you will implement extended the scheduler from the previous assignment with priority-based scheduling.

Question 4.1: Segmentation

a. How does segmentation work?

b. Assume a system with 16-bit virtual addresses that supports four different segments, which uses the following segment table:

Segment Number	Base	Limit
0	0xdead	0x00ef
1	0xf154	0x013a
2	0x0000	0x0000
3	0x0000	0x3fff

Complete the following table and explain briefly how you derived your solution for each row in the table.

Virtual Address	Segment Number	Offset	Valid?	Physical Address
	3	0x3999		
0x2020				
		0x0204	yes	
			yes	0xf15f

Question 4.2: Threads

a. Explain the terms process, address space, and thread. How do they relate to each other?

b. Compare the three thread models one-to-one (kernel-level threads), many-to-one (user-level threads), and many-to-many (hybrid threads). Point out advantages and limitations of each thread model.

c. Which types of events can trigger a thread switch in the one-to-one model?

d. Which types of events can trigger a thread switch in the many-to-one model?

e. The Unix system call `fork()` creates a new process (child), which is identical to its parent in most parts. Would it make sense for the new process to also contain copies of all the parent's (other) threads?

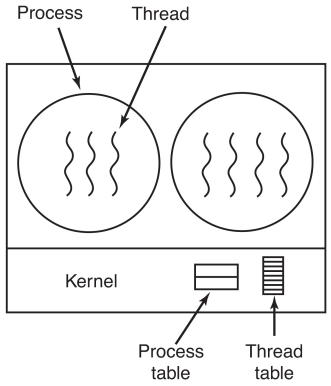
Question 4.3: Programming with pthreads

a. Write a small program that creates five threads using the pthread library. Each thread should print its number (e.g., `Hello, I am 4`) and the main program should wait for each thread to exit.

Note: This part is not (yet) relevant for this week's programming assignments, but we will come back to using pthreads later (in another programming assignment).

T-Question 4.1: Threads and Thread Models

a. The figure below illustrates one of the execution models for processes in an operating system (e.g., traditional single-threaded model, multi-threaded model with kernel-level threads, user-level threads, or hybrid threads). Explain which model is shown in the figure, what control data structures are used, and where (user space or kernel space) they are managed.



2 T-pt

b. Which of the following objects are generally all shared by the threads of a process (i.e., managed by/stored in the PCB, not the individual TCB)? **1 T-pt**

true	not true	
☐	☐	Code and stack
☐	☐	Instruction pointer, register contents, open files
☐	☐	Code, heap, open files
☐	☐	Code, stack, open network connections
☐	☐	Thread control block

c. Explain (briefly, 1-2 sentences) one main disadvantage of the one-to-one thread model. **1 T-pt**

d. Do you agree with the statement "'Switching to a different process is usually much faster than switching to a different user level thread, because switching processes can very efficiently be handled by the CPU scheduler of the operating systems"? Justify! **2 T-pt**

T-Question 4.2: Segmentation

a. Assume a system with 16-bit virtual addresses that supports four different segments (the two most significant bits of the virtual address encode the segment number, the remaining 14 bits the offset within the segment). Assume we use the following segment table: **4 T-pt**

Segment Number	Base	Limit
0	0x1000	0x15f3
1	0x8000	0x00FF
2	0x25f3	0x1000
3	0x4500	0x0300

Complete the following table.

Virtual Address	Segment Number	Offset	Valid?	Physical Address
0xC0DE				
	1	0x0200		
			yes	0x25f4

P-Question 4.1: 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 assignment, you will replace the simple FIFO scheduler from assignment 3 with a more complex 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.

You can reuse the queue implementation from the previous assignment. A sample solution for that queue will be published after the late submission deadline (Wednesday 23:59).

a. Implement/modify 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. **2 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.

b. 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

c. Add starvation prevention to the scheduler usingthe following additional rule: **4 P-pt**

- If a thread with priority P has been selected for four times without selecting a thread with a lower priority, then instead of selecting a thread 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 threads).
- Example: Assume that 0 is the lowest, 5 is the highest priority, and you have threads with priorities $1, 2$ and 4; these threads 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 2 4 4 4 4 2 4 4 4 4 2 4 4 4 4 2 4 4 4 4 1 4 ..."

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