NOTE *The weekly exercises are individual (unless marked otherwise).*

READING **10TH EDITION** *chapter 6 up to and incl. 6.6 + 6.8 + 6.9 (including the blue box **Priority inversion and the Mars Pathfinder**).*

HAND IN *Please hand in the exercises in PDF format in Brightspace.*

Exercise 1

In this exercise, you look at two implementations of multi-threaded algorithms, and the goal is to check whether mutual exclusion is guaranteed. To do so, answer the following questions for the two code fragments below:

- In which lines are the critical sections of all shared variables?
- Is mutual exclusion guaranteed? Why?

```
1  int x = 0;
2  mutex m;
3
4  void T1() {
5    while(true) {
6      m.lock();
7      x = x+1;
8      m.unlock();
9    }
10 }
11
12 void T2() {
13   while(true) {
14     if(x > 0) {
15       m.lock();
16       x = x+1;
17       m.unlock();
18     }
19   }
20 }
```

```
1  int el[5];
2  int filled = 0;
3  int next = 0;
4
5  void T1() {
6    while(true) {
7      m.lock();
8      if (filled > 0) {
9        el[filled] = 0;
10       filled = filled - 1;
11     }
12     m.unlock();
13   }
14 }
15
16 void T2() {
17   while(true) {
18     m.lock();
19     if (filled < 4) {
20       el[filled] = next;
21       filled = filled + 1;
22       next = next + 1;
23       m.unlock();
24     }
25     else {
26       m.unlock();
27     }
28   }
29 }
```

Exercise 2

Below you can find a function that applies `f` on every element of an array. Since executing `f` might take a long time, we would like to perform this task in parallel. The goal of this exercise is to give a multi-threaded solution. Below an implementation is given, but mutual exclusion is not guaranteed in it. There are multiple threads and each of them, executes the procedure `T`.

- Where are the critical sections of the variables `next` and `output` in the piece of code below?
- Give a version of the algorithm using mutexes such that mutual exclusion is guaranteed.

```
1  const int total = 10000000;
2  int next = 0;
3  int output [total];
4
5  int f (int x) {
6    // f does not make use of the global variables
7    ...
8  }
9
10 void T() {
11   int index = next;
12   next = next+1;
13
14   while (index < total) {
15     output[index] = f(index);
16     index = next;
17     next = next + 1;
18   }
19 }
```

Exercise 3

In this exercise, we look at a very simple application of binary semaphores. We would like to make the threads `T1` and `T2` execute the tasks `task1` and `task2` in turn.

- Add semaphores so that this behavior is guaranteed.
- Analyze the behavior of your solution if the initial values of `s` and `t` are changed.

```
semaphore s = 1;
semaphore t = 0;

void T1() {
  while (true) {
    task1();
  }
}

void T2() {
  while (true) {
    task2();
  }
}
```