Exercise 13

Handout: 23.05.2022 08:00
Due: 02.06.2022 23:59

Spring 2022

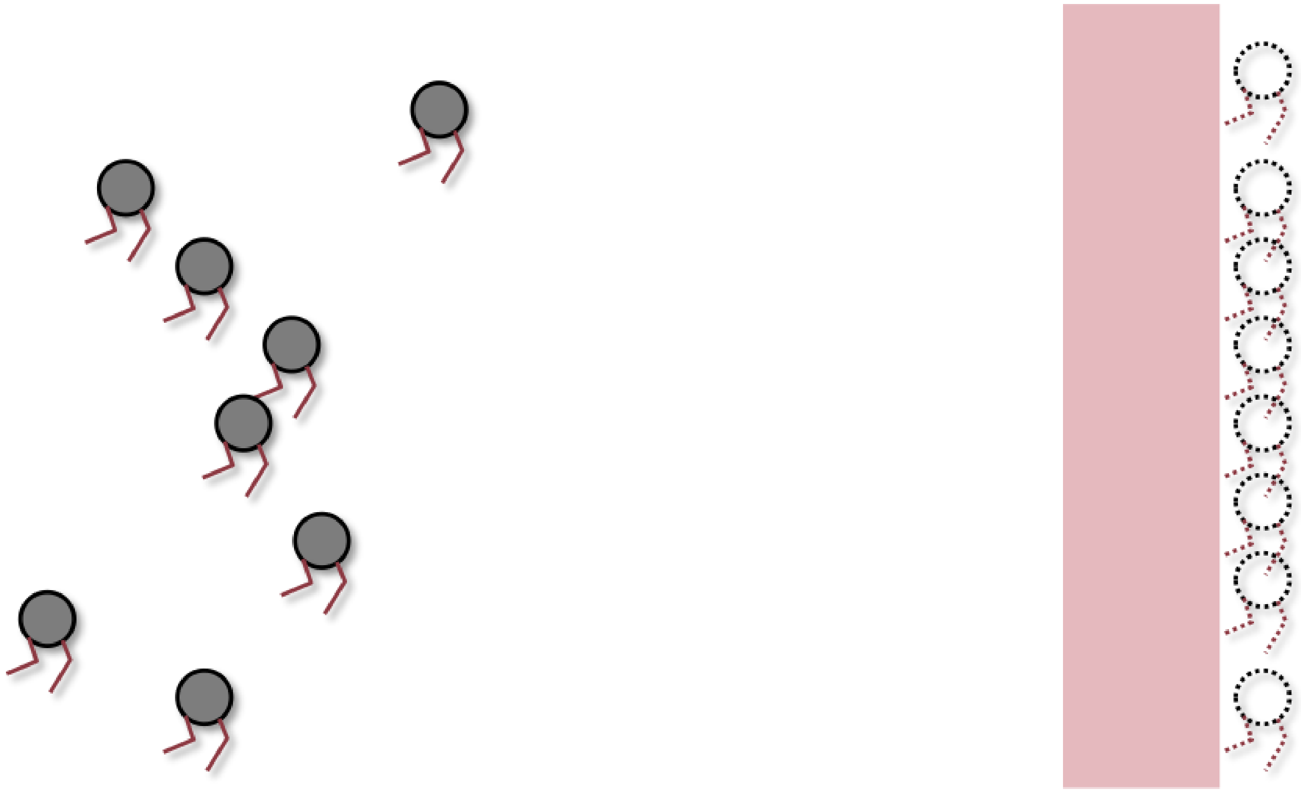
Datenstrukturen und Algorithmen CSE

</> Barrier

Barriers

A barrier (https://en.wikipedia.org/wiki/Barrier_%28computer_science%29) (for n threads) is a synchronization construct with the following semantics:

- Each thread that reaches a barrier in the code must wait until in total n threads have reached the barrier.
- Once n threads have reached the barrier, all waiting threads are allowed to continue their execution.



We distinguish barriers in terms of their reusability. It is simpler to implement a barrier that is used only once than to implement a barrier with reuse capability.

In this task, you get half the points for a correctly implemented barrier that can be used only once. You get all of the points if a barrier can be used multiple times (for example in a loop).

Usage example of a reusable barrier:

```
barrier barrier(number_threads); // usable by all threads


while(true){ // for each thread
  barrier.arrive_and_wait(); // wait for all threads to be ready
  action1(); // compute some result
  barrier.arrive_and_wait(); // e.g. result of all threads has been computed
  action2(); // use result from above in all threads
}
```

Task

Implement a barrier with the interface:

```
class Barrier {
public:
  Barrier(unsigned n); // constructor
  void arrive_and_wait();
}
```

only based on the given class `semaphore` in the file `semaphore.hpp` that features operations

```
semaphore::acquire()
semaphore::release()
```

Semaphores

Se|ma|phor, das od. der; -s, -e [zu griech. σεμα = Zeichen u. φορος = tragend]:
Signalmast mit beweglichen Flügeln.

Optische Telegrafievorrichtung mit Hilfe von schwenkbaren Signalarmen, Claude Chappe 1792

A semaphore (https://en.wikipedia.org/wiki/Semaphore_(programming)) (Edsger Dijkstra 1965) ensures that at most a given number of threads can enter a code area at the same time. A binary semaphore, for example, makes sure that only one thread can execute code at a time:

```
semaphore s(1); // shared by some threads
...

// some threads execute this:
s.acquire();
```

```
action(); // at most one thread can execute this code at a time
s.release(); // now the next thread that waited in front of this code can continue executing
```

Technically, a semaphore is implemented with a counter that is initially set to the number s of threads that may enter.

When a thread calls acquire, it

1. waits (sleeps) as long as $s = 0$
2. decreases the counter: $s \leftarrow s - 1$ and continues execution

When a thread calls release, it

1. increases the counter: $s \leftarrow s + 1$
2. informs a waiting thread (if any) about the counter change

A semaphore s that is initialized with `semaphore s(1)` is in a released state and can be acquired by (one) thread. A semaphore s that is initialized with `semaphore s(0)` starts in a locked state and needs to be released before a thread can pass.

Implementation Hints

The correct implementation of a reusable barrier is not simple. We suggest the following:

1. Implement a simple barrier for a single use first. This is easier and gives half the points.
   - Introduce a counter `entered` on the barrier with an initial value 0.
   - Each arriving thread increments the counter by one.
   - Threads wait until the counter is at n and then continue.

   It is usually not a good idea to wait for a variable to change its value using a loop (why?). You may want to employ two semaphores: one for protecting threads from executing code concurrently and one for threads waiting for the counter.

   An interesting and very helpful pattern is that of a turnstile: 

   All threads wait for a binary semaphore and once a thread gets access, it releases the semaphore for the others:

```
semaphore s(1); // binary shared semaphore
...

// turnstile
```

```
s.acquire(); // a thread that can pass here
s.release(); // opens the door for the next
```

In order to make the turnstile work, the semaphore needs to be set to 0 at the beginning (otherwise it does not block) and one thread needs to release it to open for all threads.

2. Now implement the multiple-use barrier. This must run in two phases so that the counter can be reset from the top before a thread re-enters the barrier (why?). There are several ways to implement this. We suggest the following:

   - Introduce an additional counter: `left`.

   Phase 1:

   - Count all incoming threads with the counter `entered` as above.
   - Each thread waits until this counter is at n and only then continues with phase 2. (Only) the last thread sets the counter `left` to 0. Only then any threads are allowed to continue with phase 2.

   Phase 2:

   - Count arriving threads with the counter `left`.
   - Each thread waits until this counter is at n and only then continues.
   - (Only) the last thread resets the counter `entered` to 0.

   Consider the state of your semaphore after n threads have entered and left both phases. The state should be the same as when the barrier was initialized.