**DM510 Operating Systems**
Spring 2026

---

# Exercise Sheet 4

Complete before tutorial on Thursday, March 5th

---

## Learning goals

Be able to

- identify race conditions in code
- use different synchronization mechanisms: critical section, mutex, semaphore
- understand different implementations for synchronization mechanisms and their advantages / disadvantages: disabling interrupts, atomic instructions, system calls to CPU scheduler
- program with atomic instructions

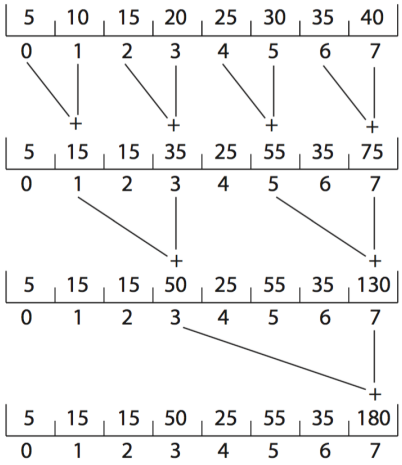
## Chapter 6+7

**Exercise 1.** Race conditions can occur in many computer systems. Consider a banking system that maintains an account balance with two functions: `deposit(amount)` and `withdraw(amount)`. These two functions are passed the amount that is to be deposited or withdrawn from the bank account balance. Assume that a husband and wife share a bank account. Concurrently, the husband calls the `withdraw()` function, and the wife calls `deposit()`. Assume that no precautions regarding race conditions were taken by the programmers. Describe how a race condition is possible and what might be done to prevent the race condition from occurring.

**Exercise 2.** The following program example can be used to sum the array values of size N elements in parallel on a system containing N computing cores (there is a separate processor for each array element):

```
for (int j = 1; j<log_2(N); ++j) {
        #pragma omp parallel for
        for (int k = 1; k < N; ++k) {
                if ((k + 1) % pow(2,j) == 0) {
                        values[k] += values[k - pow(2,(j-1))]
                }
        }
}
```

This has the effect of summing the elements in the array as a series of partial sums, as shown in the figure below.




After the code has executed, the sum of all elements in the array is stored in the last array location. Are there any race conditions in the above code example? If so, identify where they occur and explain how it could be fixed. If not, demonstrate why this algorithm is free from race conditions.

**Exercise 3.** Disabling interrupts frequently can affect the system's clock. Explain why this can happen and how such effects can be minimized.

**Exercise 4.** Explain why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems.

**Exercise 5.** The Linux kernel has a policy that a process cannot hold a spinlock while attempting to acquire a semaphore (implemented via scheduler). Explain why this policy is in place.

**Exercise 6.** The Python interpreter uses an extreme approach to synchronization. Read the following article: https://realpython.com/python-gil/. Then discuss:

a. What is the Global Interpreter Lock (GIL)?

b. What are advantages of the GIL?

c. What are disadvantages of the GIL?

d. How do Python threads relate to user and kernel threads?

**Exercise 7.** One approach for using `compare_and_swap()` for implementing a spinlock is as follows:

```
void lock spinlock(int *lock) {
    while (compare_and_swap(lock, 0, 1) != 0)
        ; /* spin */
}
```

A suggested alternative approach is to use the *compare and compare-and-swap* idiom, which checks the status of the lock before invoking the `compare_and_swap()` operation. The rationale behind this approach is to invoke `compare_and_swap()` only if the lock is currently available. This strategy is shown below:

```
void lock_spinlock(int *lock) {
    while (true) {
        if (*lock == 0) {
            /* lock appears to be available */

            if (!compare_and_swap(lock, 0, 1))
                break;
        }
    }
}
```

a. Discuss the possible motivation for *compare and compare-and-swap*.

b. Does this *compare and compare-and-swap* idiom work appropriately for implementing spinlocks? If so, explain. If not, illustrate how the integrity of the lock is compromised.

**Exercise 8.** The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P0 and P1, share the following variables:

```
boolean flag[2]; /* initially false */
int turn;
```

The structure of process Pi ($i = 0$ or $i = 1$) in Dekker's algorithm is shown below.

```
j = 1 - i; /* other process */
while (true) {
        flag[i] = true;

        while (flag[j]) {
                if (turn == j) {
                        flag[i] = false;
                        while (turn == j)
                                ; /* do nothing */
                        flag[i] = true;
                }
        }

        /* critical section */
```

```
        turn = j;
        flag[i] = false;

        /* remainder section */
}
```

a. Argue that the algorithm satisfies all three requirements for the critical-section problem.

b. Explain why memory barriers are necessary for correct functioning and where they need to be inserted.

**Exercise 9.** A multithreaded web server wishes to keep track of the number of requests it services (known as hits). Consider the two following strategies to prevent a race condition on the variable hits. The first strategy is to use a mutex lock when updating hits:

```
int hits;
mutex_lock hit_lock;
hit_lock.acquire();
hits++;
hit_lock.release();
```

A second strategy is to use an atomic integer:

```
atomic_t hits;
atomic_inc(&hits);
```

Explain which of these two strategies is more efficient.

**Exercise 10.** Discuss how a mutex could be implemented with a combination of atomic instructions and system calls, to achieve the best of both worlds.

# C Programming

**Exercise 11.** Implement the `wait()` and `signal()` operations of a semaphore using the `atomic_compare_exchange_strong()` instruction from `<stdatomic.h>`, see also https://pubs.opengroup.org/onlinepubs/9799919799/functions/atomic_compare_exchange_strong.html. Compare with the implementation of mutex operations. Your implementation does not need to guarantee bounded waiting. However, argue whether or not it does.

**Exercise 12.** In a typical implementation of a monitor, the object has a mutex, which is automatially acquired when calling any function on the object and released afterwards. Explain what issue would occur with a traditional mutex if one function on the object calls another function on the same object (or recursively itself)?
Using the `atomic_compare_exchange_strong()` instruction, implement the `acquire()` and `release()` operations of a *recursive mutex*. In a recursive mutex `acquire()` will not wait if the mutex is already held by the calling thread.
*Hint:* You can get a thread id of the current thread using `gettid()`.