

Lecturers:	Prof. Dr. Florina M. Ciorba Dr. Osman Simsek	<code>florina.ciorba@unibas.ch</code> <code>osman.simsek@unibas.ch</code>
Assistants:	Thomas Jakobsche	<code>thomas.jakobsche@unibas.ch</code>
Tutors:	Reto Krummenacher Agni Ramadani Tom Rodewald	<code>reto.krummenacher@unibas.ch</code> <code>agni.ramadani@unibas.ch</code> <code>tom.rodewald@unibas.ch</code>

Exercise 3: Synchronization

(10 points)

Given: April 5, 2024

Deadline: April 30, 2024, 10:00

Objectives

- Understand the producer-consumer problem (semaphores and mutex locks).
- Investigate the dining-philosopher problem and report on the output.
- Correct the dining-philosopher problem and report the correct output.
- Investigate and correct example code that contains problems similar to deadlocks.

Tasks

- Task 1: Bounded-Buffer and Producer-Consumer (3 points)
- Task 2: Dining-Philosopher (3 points)
- Task 3: Problem Investigation (2 points)
- Task 4: Synchronization Problems (1 point)
- Task 5: Deadlock vs. Starvation (1 point)
- Task 6: Bonus (1 bonus point)

Instructions

- You can solve this exercises in teams of two.
- Submit the solution of each task with detailed comments that clarify your solution.
- Show your solution and upload it to <https://adam.unibas.ch> with all deliverables in a ZIP folder with the naming scheme: `[GroupID]_Ex[SheetNo]_LastName1_LastName2`.
- In total, at least 65% of exercise points have to be obtained (with a min of 30% of each exercise).

Task 1: Bounded-Buffer and Producer-Consumer**(3 points)**

In this task you will work on the bounded-buffer problem using the producer-consumer model. Producers and consumers (running as separate threads) move items to and from a buffer with a fixed size. T1.c contains the code without the necessary synchronization.

Hint: In this bounded-buffer example producers should stop producing when the buffer is full, and consumers should only consume items that are actually in the buffer.

To compile the code: `gcc -o T1 T1.c -lpthread`

To execute the code: `./T1 <duration> <producer threads> <consumer threads>`

i) Execute T1.c with the parameters below, report the output and explain the problems.

- `./T1 10 5 0`
- `./T1 10 0 5`

ii) Correct the code by inserting the necessary synchronization, execute your corrected code with the parameters below, report the output and explain the correct process of the producer-consumer model. **Hint:** You can use counting semaphores and mutex locks.

- `./T1 10 5 0`
- `./T1 10 0 5`
- `./T1 10 2 2`

You must use the given source file T1.c as your starting point. All you need is to implement the open TODOs in the code.

Task 2: Dining-Philosopher**(3 points)**

In this task you will work on the dining-philosophers problem using condition variables. Philosophers spend their lives alternating between thinking and eating, thinking and eating, etc. They occasionally try to pick up forks to eat from a bowl at the center of the table. They can only eat when their neighbors are not eating.

Hint: If you do not see the "DINNER IS OVER" message at the end of the program, then something is wrong and your code might encounter a deadlock. Deadlocks might not always occur, so try to run your code multiple times to be sure.

To compile the code: `make all`

To execute the code: `./diningphilosophers`

There are multiple files in this task. All you need is to implement the open TODOs in the code (main.c and dining.c).

Task 3: Problem Investigation**(2 points)**

Investigate the code example given below, in Listing 1. Adjust the code to make it executable and run it a few times. What is the name of the problem that can occur and how can you solve it?

Listing 1: Problem example.

```
1  // thread one runs in this function
2  void *do_work_one(void *param)
3  {
4      int done = 0;
5      while (!done)
6      {
7          pthread_mutex_lock(&first_mutex);
8          if (pthread_mutex_trylock(&second_mutex) == 0)
9          {
10             // do some work
11             pthread_mutex_unlock(&second_mutex);
12             done = 1;
13         }
14         pthread_mutex_unlock(&first_mutex);
15     }
16     pthread_exit(0);
17 }
18
19 // thread two runs in this function
20 void *do_work_two(void *param)
21 {
22     int done = 0;
23     while (!done)
24     {
25         pthread_mutex_lock(&second_mutex);
26         if (pthread_mutex_trylock(&first_mutex) == 0)
27         {
28             // do some work
29             pthread_mutex_unlock(&first_mutex);
30             done = 1;
31         }
32         pthread_mutex_unlock(&second_mutex);
33     }
34     pthread_exit(0);
35 }
```

Task 4: Synchronization Problems**(1 point)**

Describe the classical synchronization problems and tools to solve them for this exercise.

Task 5: Deadlock vs. Starvation**(1 point)**

Describe the difference between deadlocks and starvation for this exercise.

Task 6: Bonus**(1 bonus point)**

Investigate the code example given below, in Listing 2. Adjust the code to make it executable and run it a few times. What is the behavior of the code? How does the behavior differ from the code given in Task 3?

Listing 2: Problem example.

```
1  // thread one runs in this function
2  void *do_work_one(void *param)
3  {
4      int done = 0;
5      while (!done)
6      {
7          pthread_mutex_lock(&first_mutex);
8          if (!pthread_mutex_trylock(&second_mutex))
9          {
10             // do some work
11             pthread_mutex_unlock(&second_mutex);
12             done = 1;
13         }
14         pthread_mutex_unlock(&first_mutex);
15     }
16     pthread_exit(0);
17 }
18
19 // thread two runs in this function
20 void *do_work_two(void *param)
21 {
22     int done = 0;
23     while (!done)
24     {
25         pthread_mutex_lock(&second_mutex);
26         if (!pthread_mutex_trylock(&first_mutex))
27         {
28             // do some work
29             pthread_mutex_unlock(&first_mutex);
30             done = 1;
31         }
32         pthread_mutex_unlock(&second_mutex);
33     }
34     pthread_exit(0);
35 }
```