Answers Exercise 3

Task 1

i)

The output with 5 producers and 0 consumers is:

```
Producer produced: 1534507321, buffer position: 1
Producer produced: 1171885191, buffer position: 0
Producer produced: 93906833, buffer position: 2
Producer produced: 1880913188, buffer position: 3
Producer produced: 1626918669, buffer position: 4
Producer produced: 813268848, buffer position: 0
Producer produced: 1111910226, buffer position: 1
Producer produced: 1174213401, buffer position: 2
Producer produced: 1898119435, buffer position: 3
Producer produced: 1445350030, buffer position: 4
Producer produced: 1080500579, buffer position: 0
Producer produced: 1205686382, buffer position: 1
Producer produced: 867974659, buffer position: 2
Producer produced: 1033013534, buffer position: 3
Producer produced: 481134014, buffer position: 4
Producer produced: 1920510410, buffer position: 0
Producer produced: 728059674, buffer position: 1
Producer produced: 1476770413, buffer position: 2
Producer produced: 2135844037, buffer position: 3
Producer produced: 199523957, buffer position: 4
```

The problem is that the buffer will be overwritten; the producers don't stop producing when the buffer is full!

The output with 0 producers and 5 consumers is:

```
Consumer consumed: 0, buffer position: 0
Consumer consumed: 0, buffer position: 1
Consumer consumed: 0, buffer position: 2
Consumer consumed: 0, buffer position: 3
Consumer consumed: 0, buffer position: 4
Consumer consumed: -1, buffer position: 0
Consumer consumed: -1, buffer position: 1
Consumer consumed: -1, buffer position: 2
Consumer consumed: -1, buffer position: 3
Consumer consumed: -1, buffer position: 4
Consumer consumed: -1, buffer position: 0
Consumer consumed: -1, buffer position: 1
Consumer consumed: -1, buffer position: 2
Consumer consumed: -1, buffer position: 3
Consumer consumed: -1, buffer position: 4
Consumer consumed: -1, buffer position: 0
Consumer consumed: -1, buffer position: 2
Consumer consumed: -1, buffer position: 1
Consumer consumed: -1, buffer position: 3
Consumer consumed: -1, buffer position: 4
Consumer consumed: -1, buffer position: 0
Consumer consumed: -1, buffer position: 1
Consumer consumed: -1, buffer position: 2
Consumer consumed: -1, buffer position: 3
Consumer consumed: -1, buffer position: 4
Consumer consumed: -1, buffer position: 0
Consumer consumed: -1, buffer position: 1
Consumer consumed: -1, buffer position: 2
```

The problem is that the consumers consumes even if the buffer is empty!

ii)

After editing the code, the output with 5 producers and 0 consumers is:

```
Producer produced: 1277546584, buffer position: 0
Producer produced: 804715142, buffer position: 1
Producer produced: 2113838101, buffer position: 2
Producer produced: 950719586, buffer position: 3
Producer produced: 568877856, buffer position: 4
```

The 5 producers stop producing when the buffer is full, but nothing will ever be consumed…

The output with 0 producers and 5 consumers is "empty" because nobody produces something…

Output with 2 producers and 2 consumers:

```
Producer produced: 1415501505, buffer position: 0
Producer produced: 1722149391, buffer position: 1
Consumer consumed: 1415501505, buffer position: 0
Consumer consumed: 1722149391, buffer position: 1
Producer produced: 1490085254, buffer position: 2
Consumer consumed: 1490085254, buffer position: 2
Producer produced: 1450319603, buffer position: 3
Producer produced: 186830753, buffer position: 4
Consumer consumed: 1450319603, buffer position: 3
Producer produced: 1983687688, buffer position: 0
Producer produced: 1800176350, buffer position: 1
Consumer consumed: 186830753, buffer position: 4
Consumer consumed: 1983687688, buffer position: 0
Consumer consumed: 1800176350, buffer position: 1
Producer produced: 1886612574, buffer position: 2
```

The producer consumer model is working as it should! Produces until max limit and consumes maximum to finish the buffer elements.

Task 3

The error lies in the interpretation of integers as booleans in C. The function try_lock returns 0 if the requested mutex could be locked and 1 if the mutex is already locked by anoter thread. For this code to work we want to enter the if statements at lines 8 and 28 if the mutex could be locked. But because the inteher 0 is interpreted as false the if statements are never entered. If the locking failed an even worse situation arrises where one thread tries to access resources currently used by the other thread and later tries to unlock the mutex owned by the other thread.

Task 4

Bounded Buffer Problem

The bounded buffer problem is about a buffer of size n capable of storing n entities of data. Further there are two processes. The first is called the producer, the second one the Consumer. Both try to operate on the buffer. The producer creates new data entities and tries to append them to the buffer. The consumer tries to remove data entities from the buffer.

The challenge lies in synchronizing the read and write operations to the buffer and preventing an over- or inderflow of the buffer. This can be accomplished using three semaphores. A semaphore mutex initialized to 1, a semaphore full initialized to 0 and a semaphore called empty initialized to n. In the producer, wait is called for empty and mutex before an item is added to the buffer. After adding an item to the buffer signal is called for mutex and full. On the consumer side wait is called for full and mutex before removing an item. Signal is called afterwards for mutex and empty.

Readers-Writers Problem

The readers-writers problem is about a common data set that is shared among a number of concurrent processes. THe processes are split into two groups. The readers that only read the data set and the writers that can both read and write. The challenge lies in allowing multiple processes (reader/writer) to read at the same time but only allow one single writer to access the shared data at the same time. After each writer is finished we have the choice of either scheduling another waiting writer on all the readers.

The problem can be solved using two seamphores called rw_mutex and mutex initialized to 1 together with an integer variable called read_count that is initialized to 0.

The writers call wait on rw_mutex before and signal on rw_,utex after writing is performed. The readers first call wait on mutex and then increment the read_count. Then if the read count ws previously zero, they call wait on rw_mutex. Then they signal mutex and execute the CS. Afterwards they call wait on mutex and decrement the read_count. If the read:count is now zero, they signal rw_mutex. Afterwards signal is called for mutex.

When a writer executes signal(rw_mutex), we may resume the execution of either the waiting reader(read_count) or of a single waiting writer (waiting on rw_mutex).

Dining Philosophers Problem

The dining philosophers problem is used as an example of concurrency-control when needed to allocate several resources among several (in this case five) processes in a deadlock-free and starvation-free manner. The processes representing philosophers spend their runtime thinking (waiting) or eating (accessing the resources) in an alternating manner. Each process doesn't interact with their neighbours. They occasionally try to access up to two chopsticks (one at a time) to eat from bowl at the center of table. They nee dboth to eat and then release both when done. A siple solution with 5 semaphores leads to a deadlock when all philosophers hungry and all grab left chopstick at the same time.

There are different approaches on how to prevent the deadlock. Those following ones are a few of them.

- Allow at most four philosophers to be simultaneously at the table.
- Use an asymmetric solution such that an oddnumbered philosopher picks up first the left chopstick and then the right chopstick, whereas an even-numbered philosopher picks the right chopstick and then the left one.
- Allow a philosopher to pick up the chopsticks only if both of them are available. This is done by putting the taking of the chopsticks in a CS.

Task 5

A Deadlock is caused when two proceses hold a resource each and wait for the other to release it's resource. BOth threads wait for each other indefinitely. No progress is done. Starvation occurs if processes have different priorities. If high priority tasks are constantly executed therefore blockig resources of lower priority ones the lower priority processes are blocked indefinitely or at least for a problematic amount of time.