Operating Systems (521453A) Lab Work 2025

Aleksanteri Aska 2208409 & Mikael Tanskala TÄYTÄ

**TASK 1**

**1.(a)**

The numbers printed on the screen have gaps because the Producer and Consumer tasks access shared memory (buffer, in, out) without synchronization. As a result, the Consumer might read a buffer slot before the Producer writes a new number into it, causing gaps zero or old values. (This is a race condition between Producer and Consumer.)

**1.(b)**

Added a global pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

Locked the mutex before Producer writes to the buffer and in.

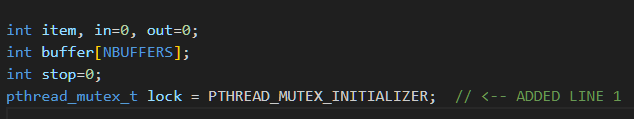
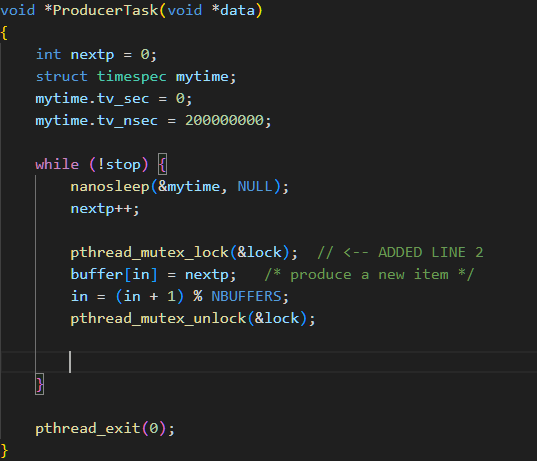
Locked the mutex before Consumer reads from the buffer and out.

Unlocked the mutex after each access.

**1.(c)**

Mutex locking ensures that only one thread at a time can access or modify the shared variables buffer, in, and out.

This prevents race conditions, so the Consumer always reads correct and fully written numbers, fixing the gaps.



TEEE 1d

**TASK 2**

1. Both Task1 and Task2 want to write into the shared variable table[].

2. Before writing, each task locks the global mutex using pthread\_mutex\_lock(&global\_data\_mutex);.

3. If the mutex is already locked by the other task, the task waits until it becomes available.

4. When a task has the lock, it can safely write to table[] without interference.

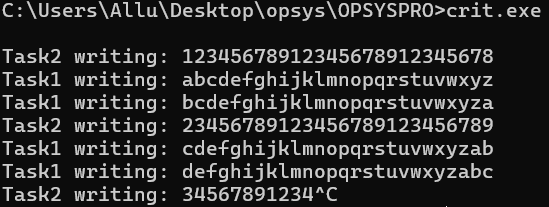
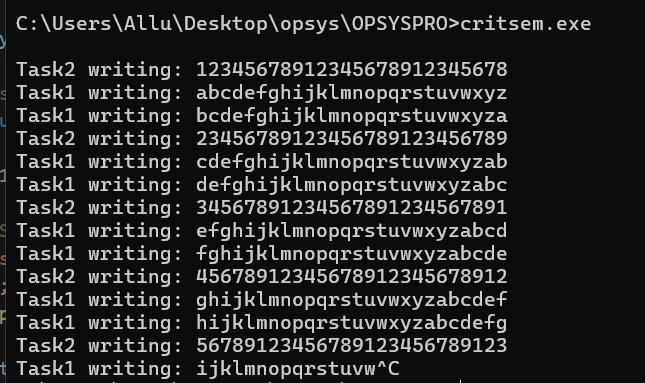
5. After writing, the task unlocks the mutex with pthread\_mutex\_unlock(&global\_data\_mutex);.

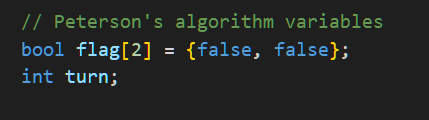
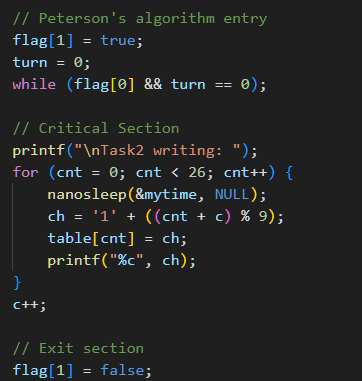
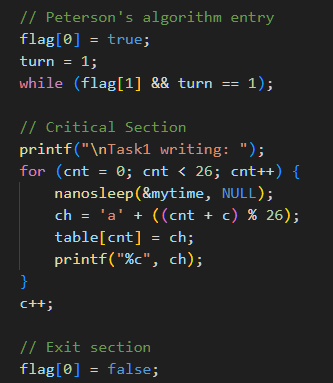
This ensures that only one task at a time can modify the shared variable, preventing race conditions.

6. The program continues this way until the user presses Enter, which stops the tasks and ends the program.

**TASK 3**

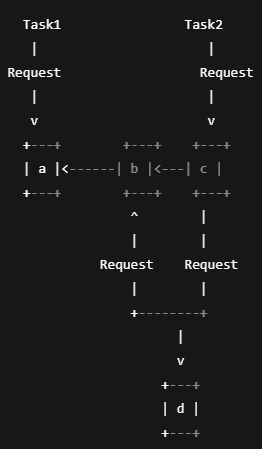
3.(b) & 3.(c)

****

****

**TASK 4**

**1.(a)**

 TARKISTA

The program stops printing because deadlock occurs:

Task1 locks c, then b, then tries to lock a.

Task2 locks b, then c, then tries to lock d.

At this point, both tasks are waiting for each other to release resources (b and c), and neither can proceed. Therefore, no further output is printed, and the program is stuck.

You can terminate the program only by pressing Ctrl+C because there is no progress, and the tasks are waiting forever.

**TASK 5**

**1.(a)**

The key difference between dead1 and dead2 is the order in which resources are acquired by the tasks.

In dead2, Task1 acquires resources in the order a -> b -> c, and Task2 acquires resources in

the order b -> c -> d.

By Task1 locking resources a -> b -> c, and Task2 locking resources b -> c -> d, the tasks no longer wait for each other in a circular manner.

In dead1, the circular wait situation arises because Task1 and Task2 are waiting for each other's resources in the opposite order, causing a deadlock.

In dead2, the tasks are more synchronized in their resource allocation, as they avoid locking the same resource in opposite orders, and thus the program can continue without deadlock.

5.(b) TEE

**TASK 6**

**6.(a)**1. **Parent (fork\_socket)**:

-Creates a socket and binds to MYPORT (4001).

- Calls fork() to create a child process.

-Sends a message to the child via UDP.

-Waits for the child to finish and prints a termination message.

2. **Child (child\_socket)**:

-Executes via execl("./child\_socket").

-Receives the message from the parent on MYPORT (6901).

-Prints received message and terminates.

**6.(b)**

Yes, the child uses getppid() to get the parent’s PID:

printf("Child of process[%d] in execution\n", getppid());

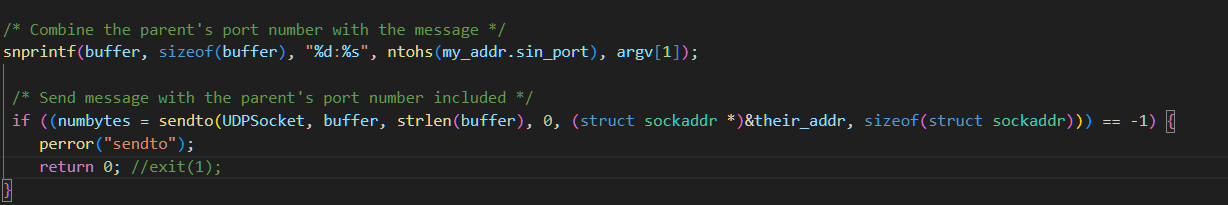
**6.(c)**

Yes, the parent gets the child’s PID from fork() and prints it:  
printf("I created a child (Pid = %d)\n", pid);

**6.(d)**

The child process knows because fork() returns 0 in the child process. This is used to distinguish the code:

if (pid == 0) { execl("./child\_socket", NULL, NULL); }

**6.(f)**for fork  
  
for child  
