Operating Systems (521453A) Lab Work 2025

Aleksanteri Aska 2208409 & Mikael Tanskala TÄYTÄ

**TASK 1**

**TASK 1**

**1.(a)**

**The gaps occur because the producer and consumer tasks access the shared buffer without synchronization. The producer may overwrite the buffer before the consumer reads, and the consumer might read from an empty buffer. There are no checks to prevent the buffer from overflowing or underflowing, leading to missed or duplicated numbers.**

**1.(b)**

**Added the following two lines:**

**while ((in + 1) % NBUFFERS == out); // Wait if buffer is full**

**while (in == out); // Wait if buffer is empty**

**1.(c)**

**These lines fix the code by ensuring the producer waits when the buffer is full and the consumer waits when the buffer is empty. This prevents overwriting unread data and reading from an empty buffer, ensuring each number is produced and consumed exactly once in order.**

**A computer screen with text and numbers

AI-generated content may be incorrect.A computer screen shot of a program code

AI-generated content may be incorrect.**

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.

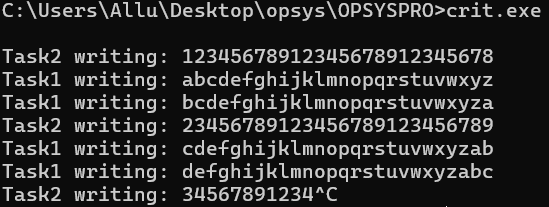
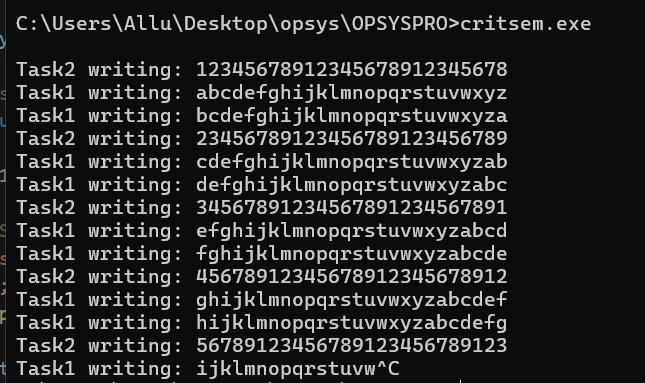
5. After writing, the task unlocks the mutex with pthread\_mutex\_unlock(&global\_data\_mutex);.

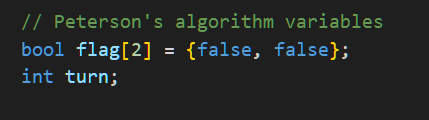
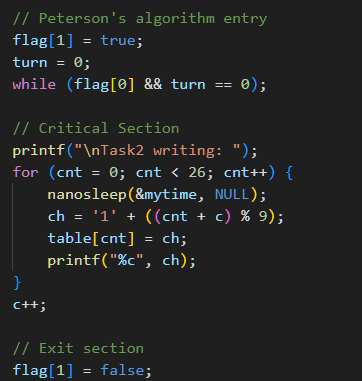
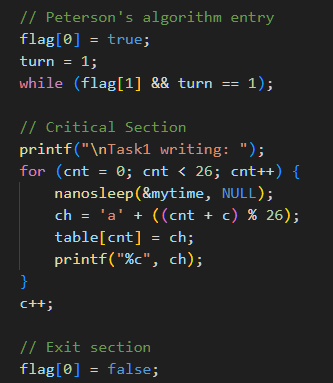
This ensures that only one task at a time can modify the shared variable, preventing race conditions.

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

**TASK 3**

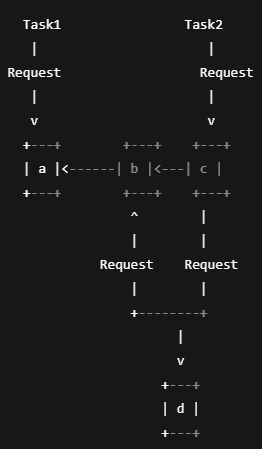
3.(b) & 3.(c)

****

****

**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());

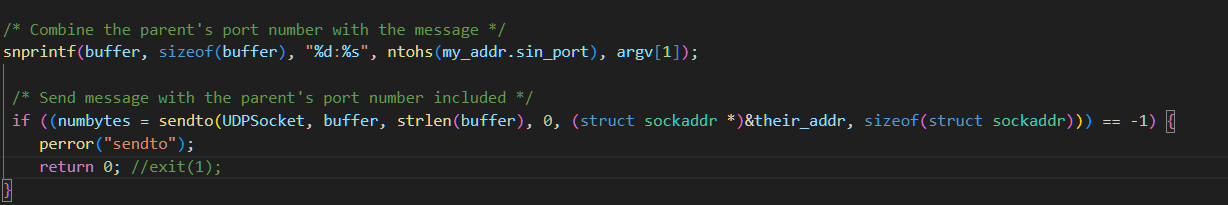
**6.(c)**

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  
