Synchronization

Solution to Dining Philosophers Problem

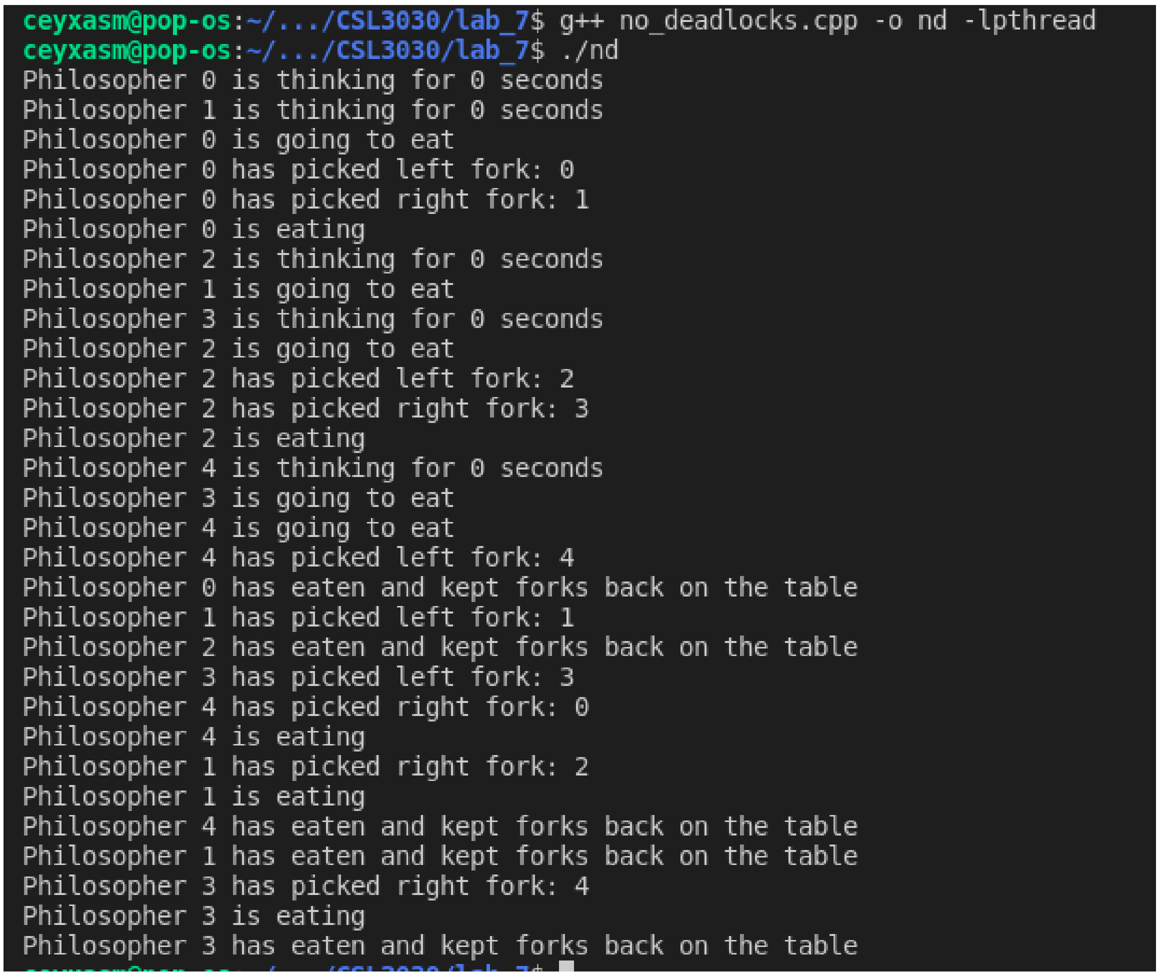
Execution

g++ no_deadlock.cpp -o nd -lpthread
g++ with_deadlock.cpp -o wd -lpthread
./nd
./wd

Description

1) No_deadlock.cpp

- Every philosopher randomly starts things for **t** seconds where t is between **1 and 10.** Whenever a philosopher is hungry a thread is generated. Since the threads are independent, it may so happen that 2 different philosopher may request for same forks.
- To avoid deadlock, even philosophers when hungry will pick their left fork first and then their right fork; whereas odd philosophers will pick just the opposite. This will insure that all the philosophers dont pick their left (or right) forks at the same time and wait for their right neighbour (or left neighbour); to release their fork.
- When the philosopher is eating, the forks are locked by **sem_wait()** and when the philosopher is done eating; they are released using **sem_post()**

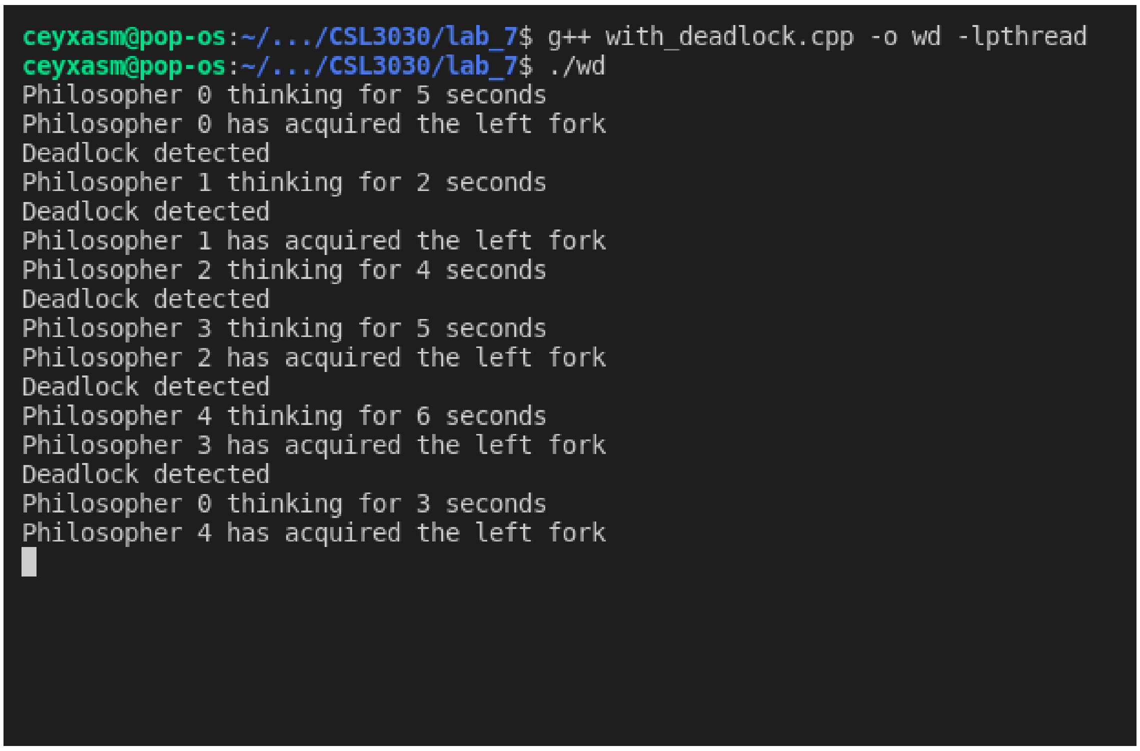


```
ceyxasm@pop-os:~/.../CSL3030/lab_7$ g++ no_deadlocks.cpp -o nd -lpthread
ceyxasm@pop-os:~/.../CSL3030/lab_7$ ./nd
Philosopher 0 is thinking for 0 seconds
Philosopher 1 is thinking for 0 seconds
Philosopher 0 is going to eat
Philosopher 0 has picked left fork: 0
Philosopher 0 has picked right fork: 1
Philosopher 0 is eating
Philosopher 2 is thinking for 0 seconds
Philosopher 1 is going to eat
Philosopher 3 is thinking for 0 seconds
Philosopher 2 is going to eat
Philosopher 2 has picked left fork: 2
Philosopher 2 has picked right fork: 3
Philosopher 2 is eating
Philosopher 4 is thinking for 0 seconds
Philosopher 3 is going to eat
Philosopher 4 is going to eat
Philosopher 4 has picked left fork: 4
Philosopher 0 has eaten and kept forks back on the table
Philosopher 1 has picked left fork: 1
Philosopher 2 has eaten and kept forks back on the table
Philosopher 3 has picked left fork: 3
Philosopher 4 has picked right fork: 0
Philosopher 4 is eating
Philosopher 1 has picked right fork: 2
Philosopher 1 is eating
Philosopher 4 has eaten and kept forks back on the table
Philosopher 1 has eaten and kept forks back on the table
Philosopher 3 has picked right fork: 4
Philosopher 3 is eating
Philosopher 3 has eaten and kept forks back on the table
```

2) With deadlock

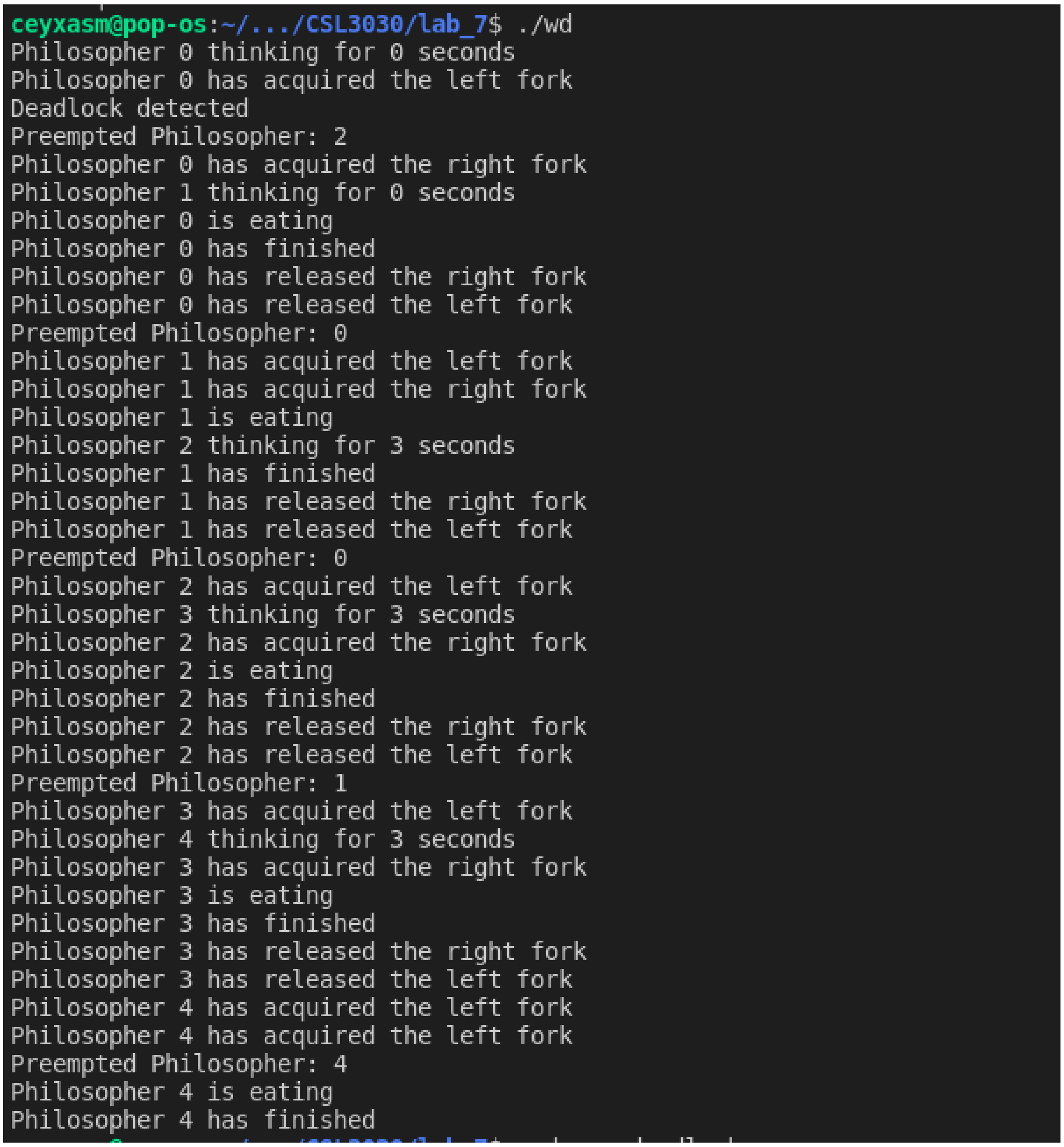
- In this scenario a deadlock is possible. To make deadlock occur more easily, (so as to better demonstrate the working of our program), the philosopher when hungry picks up the left fork, and then waits for 10 seconds before picking up the right fork. Such a condition makes it highly likely for a case to occur where all the philosophers have left fork and are waiting for the right fork.
- Such a condition demands for the detection of deadlock. This is done by **check_deadlock().** We maintain the record of forks as to which fork is picked by whom. In case of a circular dependency; a victim philosopher is randomly chosen and forced to keep his fork down. After the other 4 philosophers have dined, victim philosopher is allowed to eat.

Without pre-emption, the philosophers starve:



```
ceyxasm@pop-os:~/.../CSL3030/lab_7$ g++ with_deadlock.cpp -o wd -lpthread
ceyxasm@pop-os:~/.../CSL3030/lab_7$ ./wd
Philosopher 0 thinking for 5 seconds
Philosopher 0 has acquired the left fork
Deadlock detected
Philosopher 1 thinking for 2 seconds
Deadlock detected
Philosopher 1 has acquired the left fork
Philosopher 2 thinking for 4 seconds
Deadlock detected
Philosopher 3 thinking for 5 seconds
Philosopher 2 has acquired the left fork
Deadlock detected
Philosopher 4 thinking for 6 seconds
Philosopher 3 has acquired the left fork
Deadlock detected
Philosopher 0 thinking for 3 seconds
Philosopher 4 has acquired the left fork
```

With pre-emption:

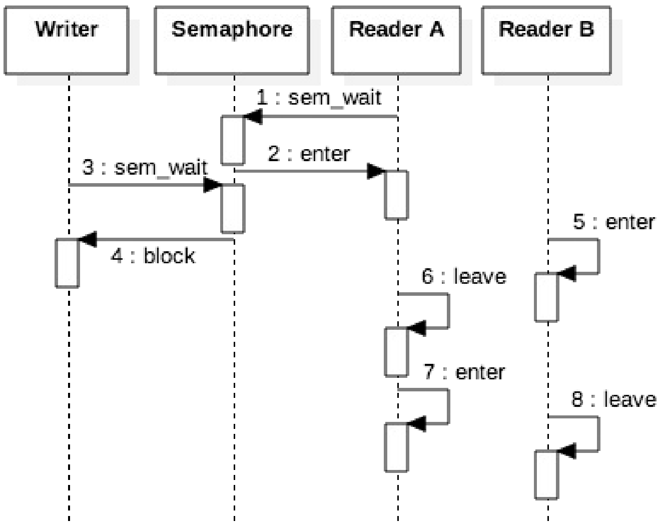


```
ceyxasm@pop-os:~/.../CSL3030/lab_7$ ./wd
Philosopher 0 thinking for 0 seconds
Philosopher 0 has acquired the left fork
Deadlock detected
Preempted Philosopher: 2
Philosopher 0 has acquired the right fork
Philosopher 1 thinking for 0 seconds
Philosopher 0 is eating
Philosopher 0 has finished
Philosopher 0 has released the right fork
Philosopher 0 has released the left fork
Preempted Philosopher: 0
Philosopher 1 has acquired the left fork
Philosopher 1 has acquired the right fork
Philosopher 1 is eating
Philosopher 2 thinking for 3 seconds
Philosopher 1 has finished
Philosopher 1 has released the right fork
Philosopher 1 has released the left fork
Preempted Philosopher: 0
Philosopher 2 has acquired the left fork
Philosopher 3 thinking for 3 seconds
Philosopher 2 has acquired the right fork
Philosopher 2 is eating
Philosopher 2 has finished
Philosopher 2 has released the right fork
Philosopher 2 has released the left fork
Preempted Philosopher: 1
Philosopher 3 has acquired the left fork
Philosopher 4 thinking for 3 seconds
Philosopher 3 has acquired the right fork
Philosopher 3 is eating
Philosopher 3 has finished
Philosopher 3 has released the right fork
Philosopher 3 has released the left fork
Philosopher 4 has acquired the left fork
Philosopher 4 has acquired the left fork
Preempted Philosopher: 4
Philosopher 4 is eating
Philosopher 4 has finished
```

Question 2: A starvation free solution of Reader-Writers Problem

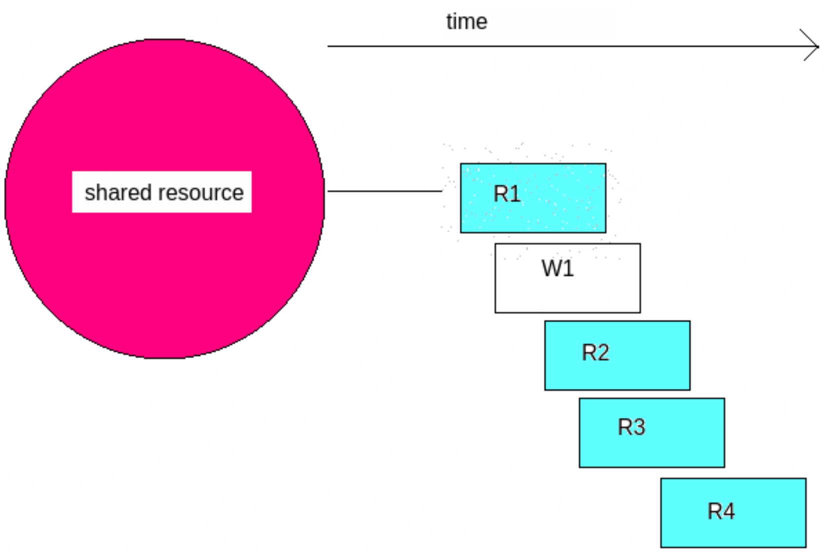
Problem parameters:

- Common set of data is shared among numerous processes
- Only one writer may perform the write at on time
- If a process is writing no ther process may read it
- Even if a single process is reading; no other process shall write
- Inherent flaw with the naive solution is that: writers require mutual exclusion where as readers do not. There in a given setting; a writer may starve as it waits indefinitely for reader processes to terminate as depicted in the image below:



1) This situation of writer's starvation can be fixed by placing an Oracle before a reader process starts its execution. The purpose of this Oracle is to block any new reader process; if there is already a writer process in the process queue. This simplifies our problem to a FIFO solution.

If there is no writer process, the reader processes can pass through the oracle and enter the Critical section. However when a writer arrives, it will call **sem_wait()** on the oracle semaphore, blocking new readers. When last of the readers call **leave(),** the process thread will **sem_post()** to the oracle semaphore; allowing new reader/writer processes to enter CS.



For example: for above scenario: Classical Solution would have processed the processes in the order: **R1 R2 R3 R4 W1**
However our solution will execute the processes in the order: **R1 W1 R2 R3 R4**

2) Yet another non-intutive approach can be to assign priorities to the reader-writer process. Default priority of reader process can be **X** where as that of writer process cab be **Y,** where X>Y. For a given scenario: Y(t+1)= f(Y(t)). such that Y(t+1) > Y(t). That is, the more time a writer process is kept waiting; the more its priority inceases. Eventually when Y(t+k) becomes greater than X, it pre-empts reader processes and forces its execution. Here the priority update function f(x) can grow linearly, polynomialy, exponentially, etc depending upon the problem to be addressed.