OS Project Report

Saba Madadi

21 Dey 1403

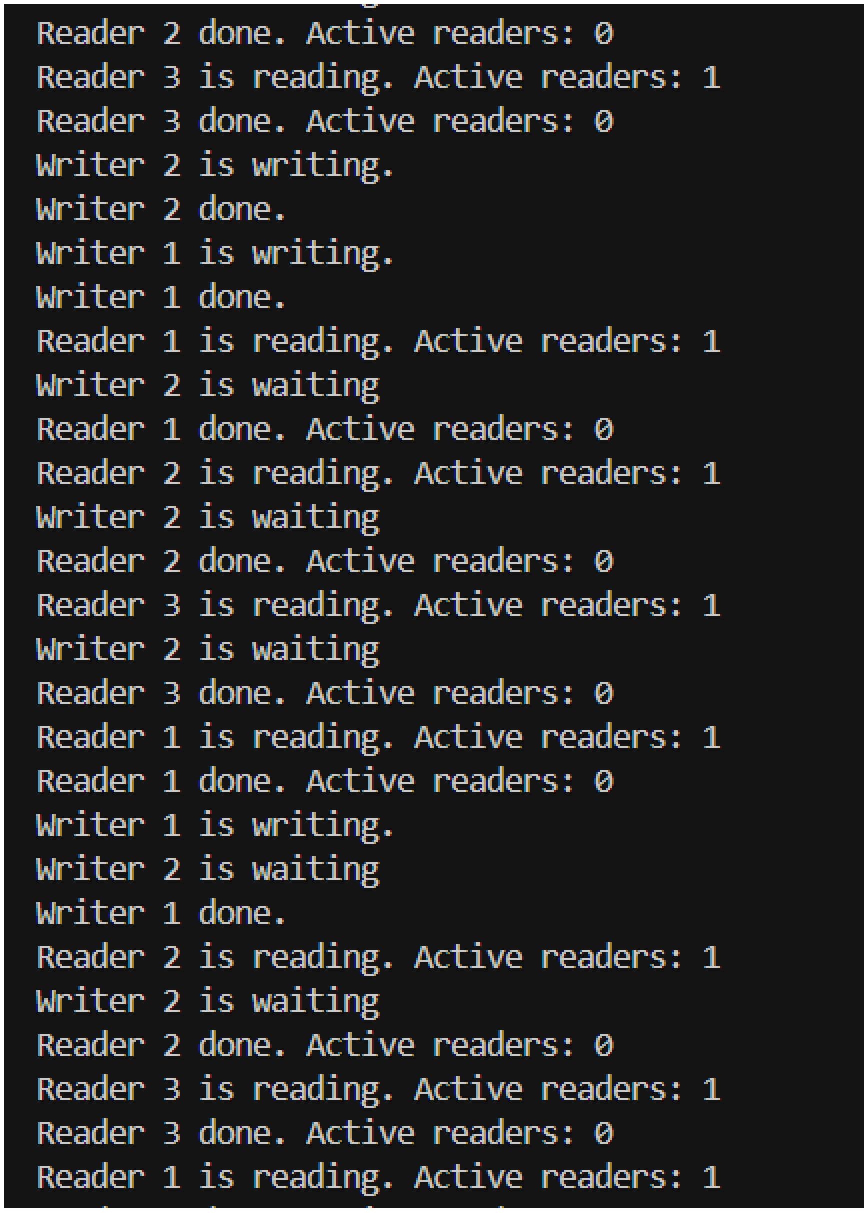
Overview

In this project, we have 2 problems. First is Readers-Writers problem, which involves managing access to a shared resource by multiple readers and writers. There are 2 scenarios: where readers have priority over writers and where writers have priority over readers. Second problem is Dining Philosophers problem, which involves 5 philosophers sitting around a table, needing to eat without causing a deadlock.

1. Readers-Writers Problem:

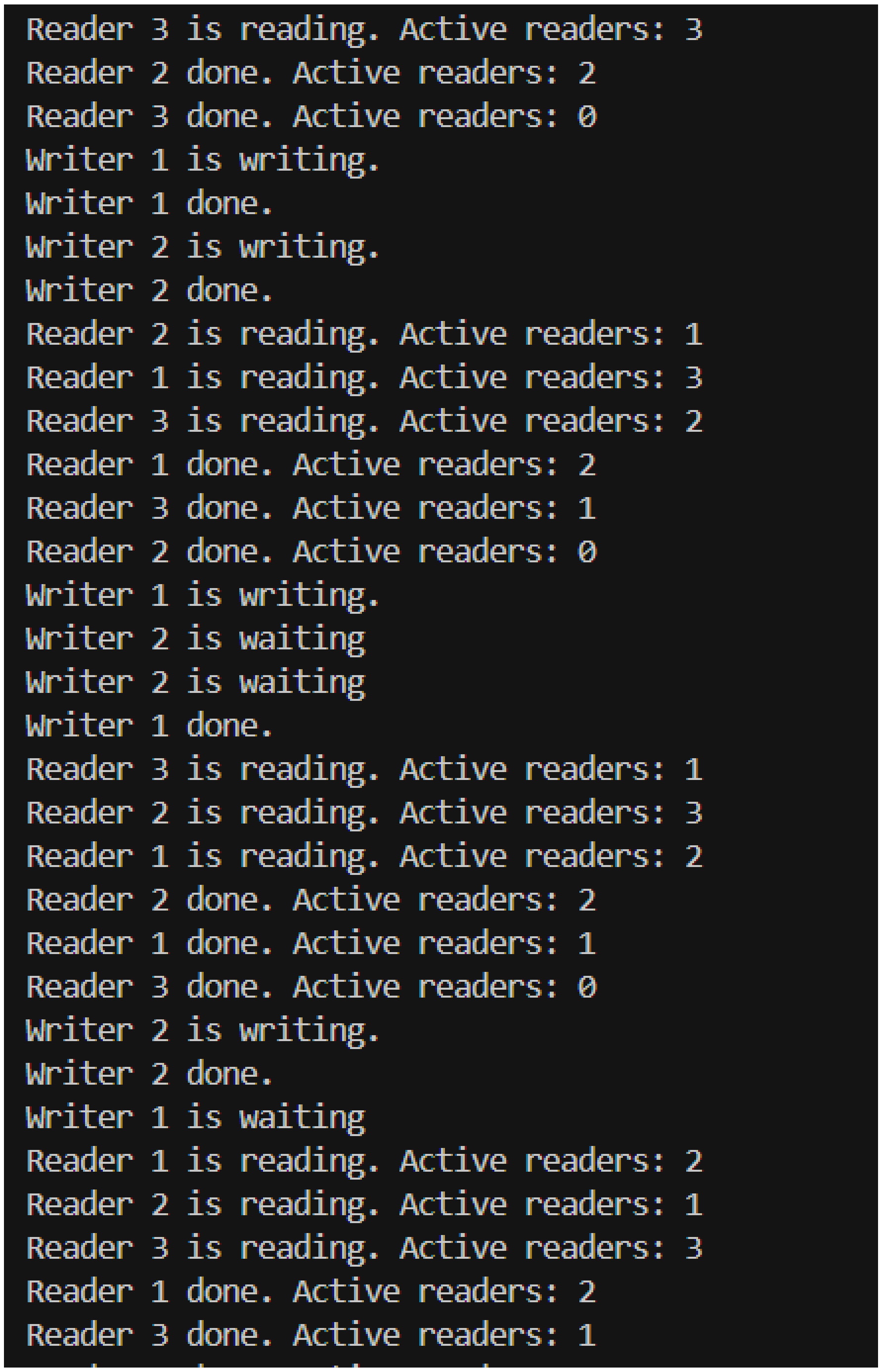
In this algorithm, two scenarios are considered. First scenario is when priority is given to readers. In second scenario, priority is given to writers, meaning that if a writer requests to enter its critical zone, it should not have to wait for all readers to exit and should be able to proceed with its work immediately.

- A) Readers have priority:
- 1 reader:



```
Reader 2 done. Active readers: 0
Reader 3 is reading. Active readers: 1
Reader 3 done. Active readers: 0
Writer 2 is writing.
Writer 2 done.
Writer 1 is writing.
Writer 1 done.
Reader 1 is reading. Active readers: 1
Writer 2 is waiting
Reader 1 done. Active readers: 0
Reader 2 is reading. Active readers: 1
Writer 2 is waiting
Reader 2 done. Active readers: 0
Reader 3 is reading. Active readers: 1
Writer 2 is waiting
Reader 3 done. Active readers: 0
Reader 1 is reading. Active readers: 1
Reader 1 done. Active readers: 0
Writer 1 is writing.
Writer 2 is waiting
Writer 1 done.
Reader 2 is reading. Active readers: 1
Writer 2 is waiting
Reader 2 done. Active readers: 0
Reader 3 is reading. Active readers: 1
Reader 3 done. Active readers: 0
Reader 1 is reading. Active readers: 1
```

- Many readers:



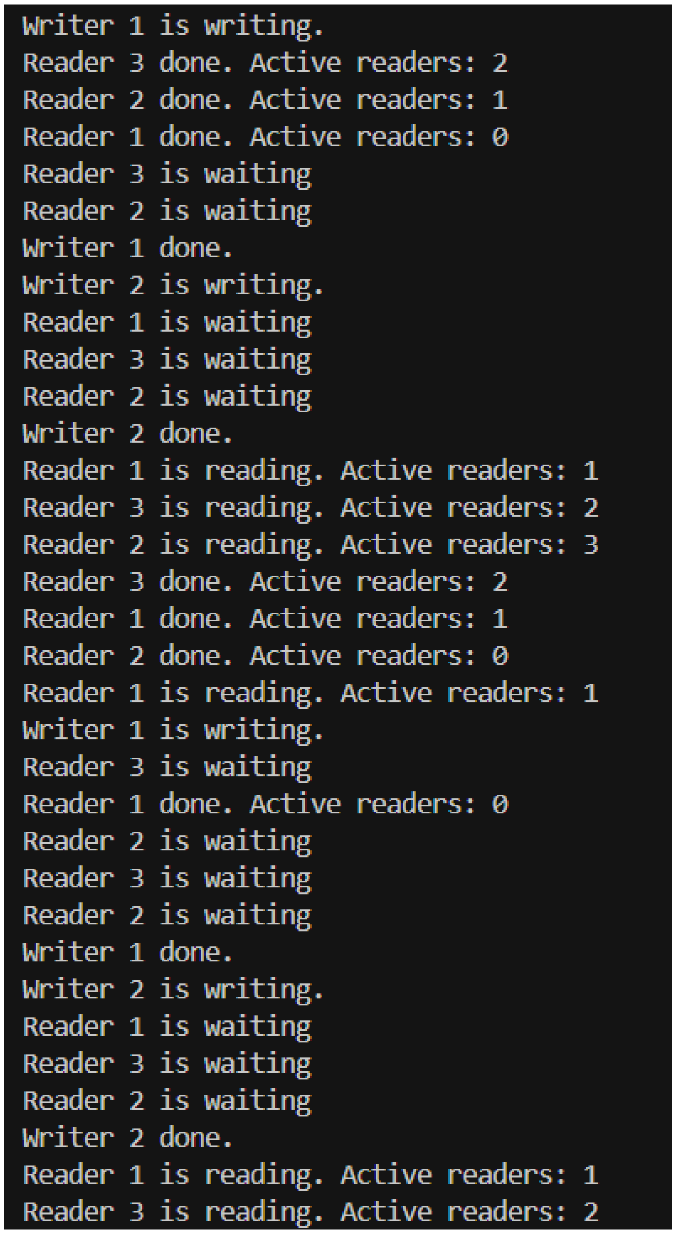
```
Reader 3 is reading. Active readers: 3
Reader 2 done. Active readers: 2
Reader 3 done. Active readers: 0
Writer 1 is writing.
Writer 1 done.
Writer 2 is writing.
Writer 2 done.
Reader 2 is reading. Active readers: 1
Reader 1 is reading. Active readers: 3
Reader 3 is reading. Active readers: 2
Reader 1 done. Active readers: 2
Reader 3 done. Active readers: 1
Reader 2 done. Active readers: 0
Writer 1 is writing.
Writer 2 is waiting
Writer 2 is waiting
Writer 1 done.
Reader 3 is reading. Active readers: 1
Reader 2 is reading. Active readers: 3
Reader 1 is reading. Active readers: 2
Reader 2 done. Active readers: 2
Reader 1 done. Active readers: 1
Reader 3 done. Active readers: 0
Writer 2 is writing.
Writer 2 done.
Writer 1 is waiting
Reader 1 is reading. Active readers: 2
Reader 2 is reading. Active readers: 1
Reader 3 is reading. Active readers: 3
Reader 1 done. Active readers: 2
Reader 3 done. Active readers: 1
```

First, we define locks. Read_count using multithreading which keeps track of how many readers are reading, and mutex is for readers. rw_mutex is for both readers and writers, and waiting_readers is a value that shows how many readers are waiting.

We have a while true loop that acquires mutex and increments count. If count is one, it means this is first reader. So, it blocks writers and resets waiting count, indicating that it is currently reading.

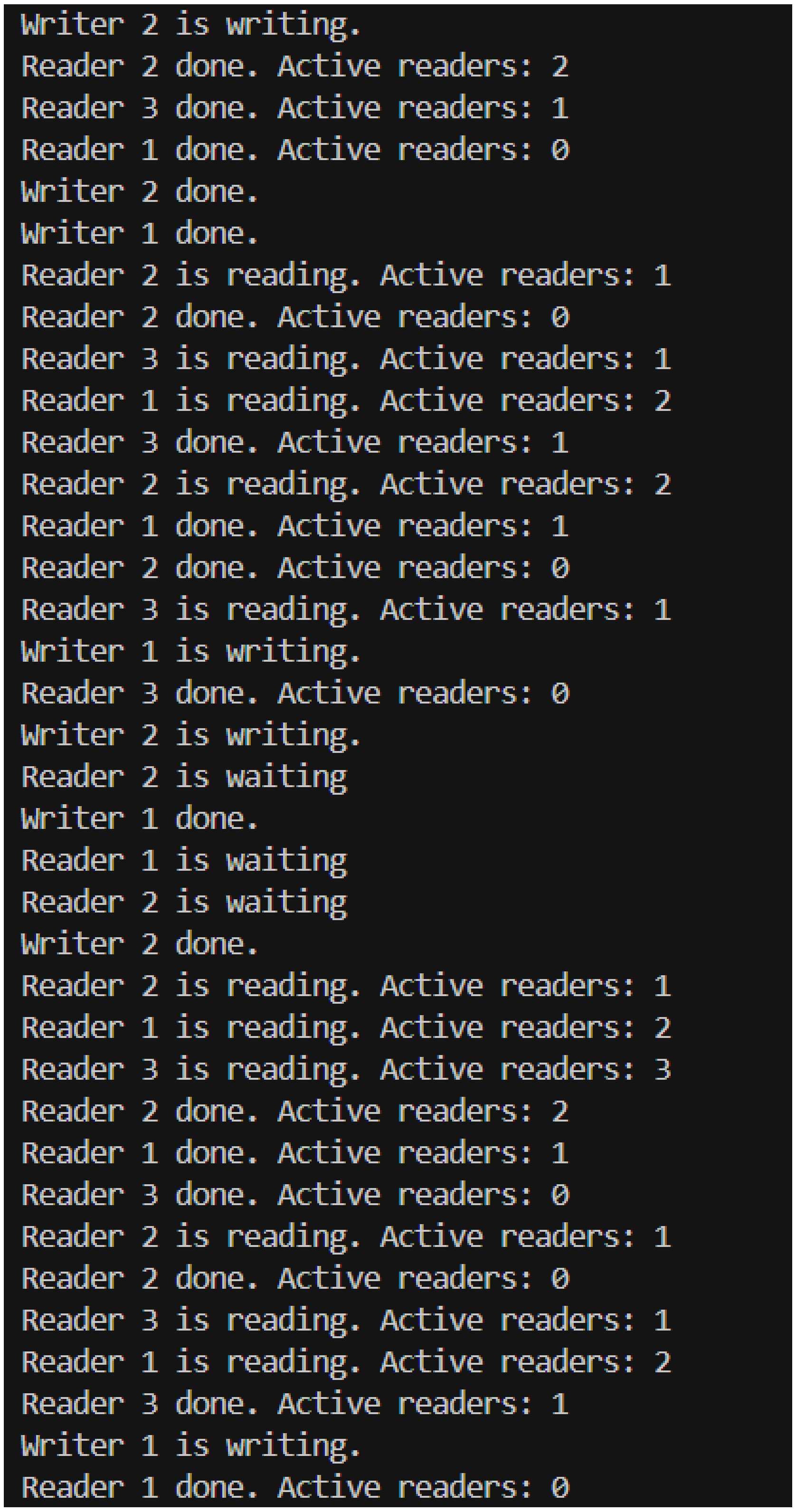
Difference btw many readers and one reader is that in many, we have mutex.release(), allowing other readers to enter. So, in many case, multiple readers can work together, while in one reader case, only one reader is allowed.
If count is zero, both reader and writer mutexes are released, allowing writers to proceed if they are waiting. Finally, mutex is released.
In writer function, if waiting_readers is zero, it can enter. If not, it waits for 0.005 seconds and then tries again.
Then, rw_mutex is acquired, writer performs its task, and finally, it is released.

- B) Writers have priority:
- 1 writer:



```
Writer 1 is writing.
Reader 3 done. Active readers: 2
Reader 2 done. Active readers: 1
Reader 1 done. Active readers: 0
Reader 3 is waiting
Reader 2 is waiting
Writer 1 done.
Writer 2 is writing.
Reader 1 is waiting
Reader 3 is waiting
Reader 2 is waiting
Writer 2 done.
Reader 1 is reading. Active readers: 1
Reader 3 is reading. Active readers: 2
Reader 2 is reading. Active readers: 3
Reader 3 done. Active readers: 2
Reader 1 done. Active readers: 1
Reader 2 done. Active readers: 0
Reader 1 is reading. Active readers: 1
Writer 1 is writing.
Reader 3 is waiting
Reader 1 done. Active readers: 0
Reader 2 is waiting
Reader 3 is waiting
Reader 2 is waiting
Writer 1 done.
Writer 2 is writing.
Reader 1 is waiting
Reader 3 is waiting
Reader 2 is waiting
Writer 2 done.
Reader 1 is reading. Active readers: 1
Reader 3 is reading. Active readers: 2
```

- Many writers:



```
Writer 2 is writing.
Reader 2 done. Active readers: 2
Reader 3 done. Active readers: 1
Reader 1 done. Active readers: 0
Writer 2 done.
Writer 1 done.
Reader 2 is reading. Active readers: 1
Reader 2 done. Active readers: 0
Reader 3 is reading. Active readers: 1
Reader 1 is reading. Active readers: 2
Reader 3 done. Active readers: 1
Reader 2 is reading. Active readers: 2
Reader 1 done. Active readers: 1
Reader 2 done. Active readers: 0
Reader 3 is reading. Active readers: 1
Writer 1 is writing.
Reader 3 done. Active readers: 0
Writer 2 is writing.
Reader 2 is waiting
Writer 1 done.
Reader 1 is waiting
Reader 2 is waiting
Writer 2 done.
Reader 2 is reading. Active readers: 1
Reader 1 is reading. Active readers: 2
Reader 3 is reading. Active readers: 3
Reader 2 done. Active readers: 2
Reader 1 done. Active readers: 1
Reader 3 done. Active readers: 0
Reader 2 is reading. Active readers: 1
Reader 2 done. Active readers: 0
Reader 3 is reading. Active readers: 1
Reader 1 is reading. Active readers: 2
Reader 3 done. Active readers: 1
Writer 1 is writing.
Reader 1 done. Active readers: 0
```

If a writer requests to enter its critical section, it should not wait for all readers to exit and should be able to proceed with its work immediately.

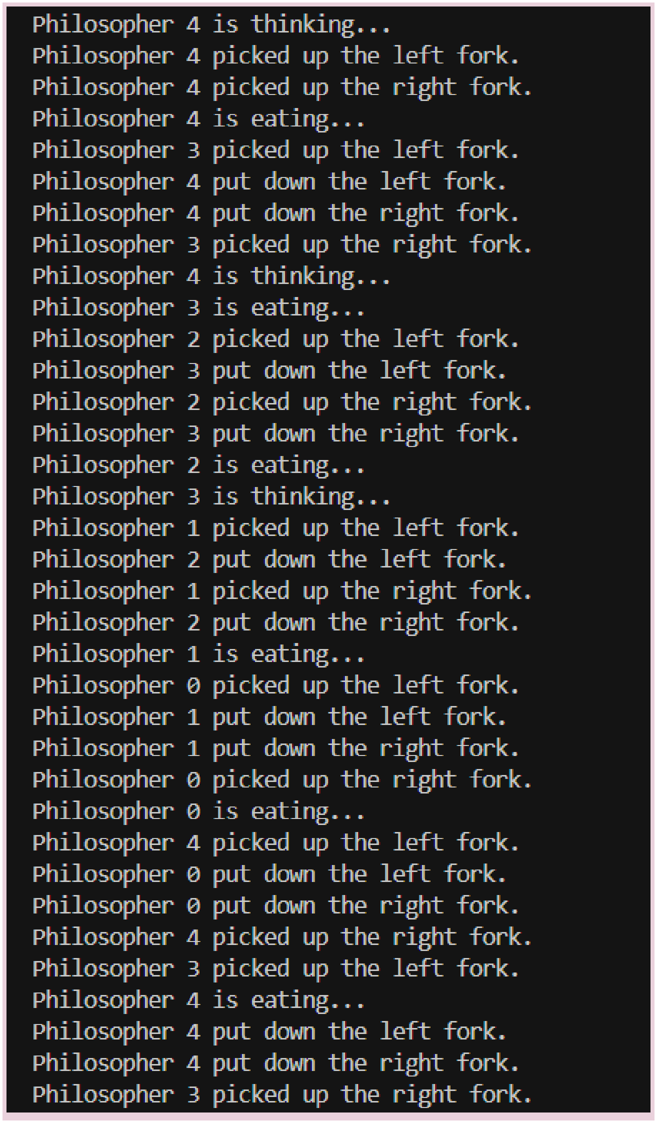
We implemented Readers-Writers problem with writers priority. In this code we defined 2 separated mutexes using multithreading, 1 is only for readers and the other one only for writers. We also defined a value for writers count and the other for readers count. In readers and writers functions we didn't use shared mutexes and one mutex is only locking readers and the other one only lock writers.
The only limitation is on readers and when reader are trying to enter to their critical sections no writer must be inside. The writer count value checks this condition. Only if writers count is zero reader are allowed to enter the critical section. We also handled 1 writer and many writers being in their critical sections using the writer mutex and for the many writers situation after increasing the writer count, writer mutex in released so other writers can enter their critical section. But in the 1 writer situation the mutex is not released until writer is done.

2. Dining-Philosophers Problem:

In this scenario, there are 5 philosophers who must be able to complete their work without deadlock.
First, there are 5 philosophers. We use a boolean *TERMINATE* to check if ends everywhere. For example, after 5 seconds in main thread, it becomes false, and everyone knows to finish their work. *room* shows how many philosophers can grab forks. Logically, max num of philosophers is total divided by 2 (floor) .For example, 2 philosophers can use 4 forks.

To prevent starvation, we have a wait list. We place forks on table equal to number of philosophers. To avoid starvation, we defined a wait list. When a philosopher finishes thinking, they add their name to end of list. When it’s their turn, they acquire room.
Then, they check their ID in wait list. If they are not at [0] (they must be first), they release room for first person in list to take it and wait for 0.5 seconds. They set waiting to true so we can print a message and know they are waiting.
We handled starvation, but in threading library, semaphore handles it also before us.

In else part, they remove themselves from wait list. They acquire left and right forks, eat their food, put down forks, and release room.



```
Philosopher 4 is thinking...
Philosopher 4 picked up the left fork.
Philosopher 4 picked up the right fork.
Philosopher 4 is eating...
Philosopher 3 picked up the left fork.
Philosopher 4 put down the left fork.
Philosopher 4 put down the right fork.
Philosopher 3 picked up the right fork.
Philosopher 4 is thinking...
Philosopher 3 is eating...
Philosopher 2 picked up the left fork.
Philosopher 3 put down the left fork.
Philosopher 2 picked up the right fork.
Philosopher 3 put down the right fork.
Philosopher 2 is eating...
Philosopher 3 is thinking...
Philosopher 1 picked up the left fork.
Philosopher 2 put down the left fork.
Philosopher 1 picked up the right fork.
Philosopher 2 put down the right fork.
Philosopher 1 is eating...
Philosopher 0 picked up the left fork.
Philosopher 1 put down the left fork.
Philosopher 1 put down the right fork.
Philosopher 0 picked up the right fork.
Philosopher 0 is eating...
Philosopher 4 picked up the left fork.
Philosopher 0 put down the left fork.
Philosopher 0 put down the right fork.
Philosopher 4 picked up the right fork.
Philosopher 3 picked up the left fork.
Philosopher 4 is eating...
Philosopher 4 put down the left fork.
Philosopher 4 put down the right fork.
Philosopher 3 picked up the right fork.
```