Sergi Aliaga

Northeastern University

Homework 2

Question 1

In 1965, Edsger W. Dijkstra described the following problem. Five philosophers sit at a round table with bowls of noodles. Forks are placed between each pair of adjacent philosophers. Each philosopher must alternately think or eat. However, a philosopher can only eat noodles when she has both left and right forks. Each fork can be held by only one philosopher, and each fork is picked up sequentially. A philosopher can use the fork only if it is not being used by another philosopher. Eating takes a random amount of time for each philosopher. After she finishes eating, the philosopher needs to put down both forks so they become available to others. A philosopher can take the fork on her right or the one on her left as they become available, but cannot start eating before getting both of them. Eating is not limited by the remaining amounts of noodles or stomach space; an infinite supply and an infinite demand are assumed.

Implement a solution for an unbounded odd number of philosophers, where each philosopher is implemented as a thread, and the forks are the synchronizations needed between them. Develop this threaded program in pthreads. The program takes as an input parameter the number of philosophers. The program needs to print out the state of the table (philosophers and forks) – the format is up to you.

Answer the following questions: you are not required to implement a working solution to the 3 questions below, though adding each case to your C/C++ implementation will earn extra credit (on your quiz grade) for everyone who tries it.

Edsger W. Dijkstra was a Dutch scientist famous by his contributions to computing science, even considered one of the most influential founding figures of the discipline. In the 1960s he started working on concurrent computing, being credited as the first to identify and solve the mutual exclusion problem. He was also one of the early pioneers of the research on principles of distributed computing. The algorithm that carries his name, the Dijkstra's algorithm, is one solution to the shortest path problem in graphs, that is, finding the path between to nodes in a graph such that the sum of weights of its edges is minimized.

In 1965, Dijkstra formulated the dinning philosophers problem as a student exam exercise. The problem was presented in terms of computer competing for access to tape drive peripherals. The problem was designed to illustrate the challenges of avoiding deadlock, a system state in which no progress is possible. For a real world system, running into a deadlock might have terrible consequences. In the dinning philosophers problem, a deadlock might be incurred if a binary semaphore is used for each fork, that is, if each philosopher holds a fork as soon as it is available. Since each philosopher needs two forks for eating, in the situation in which each philosopher holds one fork there is no progress possible and a deadlock is incurred.

The solution implemented in this part of the assignment solves this problem by using a mutual exclusion variable, a state variable and a conditional variable for each philosopher. The possible states of a philosopher are: Thinking, Hungry or Eating. If one of the neighbor philosophers are using one of the forks, the philosopher remains waiting in the Hungry state. The Hungry state is necessary to make sure that, when signaling the neighbors after putting the forks down, the neighbors have already called "pthread_cond_wait()". If there is no hungry state, "pthread_cond_signal()" might be called before the neighbor stops thinking and calls "pthread_cond_wait()", incurring into a logical error.

Details of the implementation can be found in the file "Q1.c". Below we include a description of the input parameters and the metrics used to test the code. Table 1 includes the specific values used.

Input parameters

- Number of philosophers: Number of philosophers in the table. This parameter is swept in order to understand the behavior of the algorithm when increasing the number of philosophers in the table.
- Number of iterations: Number of cycles between the three possible states of the philosophers (Thinking, Hungry and Eating) before leaving the table. The dinner finishes when all philosophers have left the table. The output indicates when a philosopher leaves the table, its state is updated to 'Finished' and it is no longer displayed in the table.
- Lower and upper bounds: Bounds (in seconds) on the time taken to eat and think by each philosopher. Each philosopher will take a random time between this two bounds to eat or think. The two bounds are the same for the two states (eat and think) but they could be forced to be different in order to study the effect of eating slower than thinking (on average) or vice versa.

For the simulations results presented, the parameters employed have been:

Number of iterations	Lower Bound (s)	Upper bound
3	1	3

Table 1: Input parameters for the simulation

Metrics:

- Average wait time: For each iteration between the three states, the time spend in the Hungry state is captured. Once dinner is over, this time is averaged across iterations and across philosophers. The resulting value is used as a metric of the average wait time.

a. *Does the number of philosophers impact your solution in any way? How about if only 3 forks are placed in the center of the table, but each philosopher still needs to acquire 2 forks to eat?*
Figure 1 reports the results obtained when there are as many forks as philosophers, one between each. In Figure 2, the results with only three forks in the middle of the table are included. Code corresponding to this section can be found in the file *"Q1a.c"*.

From Fig. 1 we observe a relatively similar behavior when the number of philosophers is increased. A more statistically significant analysis could be carried out to decide but preliminary results seem to indicate the solution is no impacted by the number of philosophers. This makes sense, since for this solution, adding more philosophers means adding the same number of forks.

Moreover, when only three forks are placed in the middle of the table we see an increase in average wait time, due to the fact that the number of forks stays constant while more hungry philosophers are added to the table.

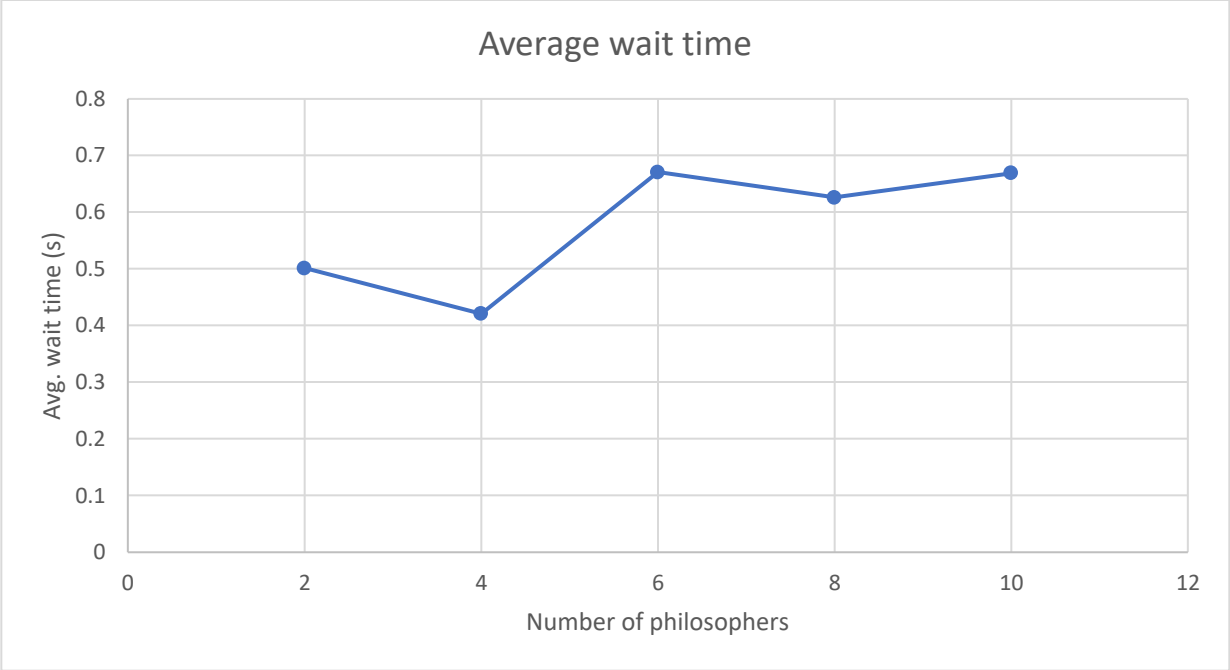

Figure 1: Average wait time

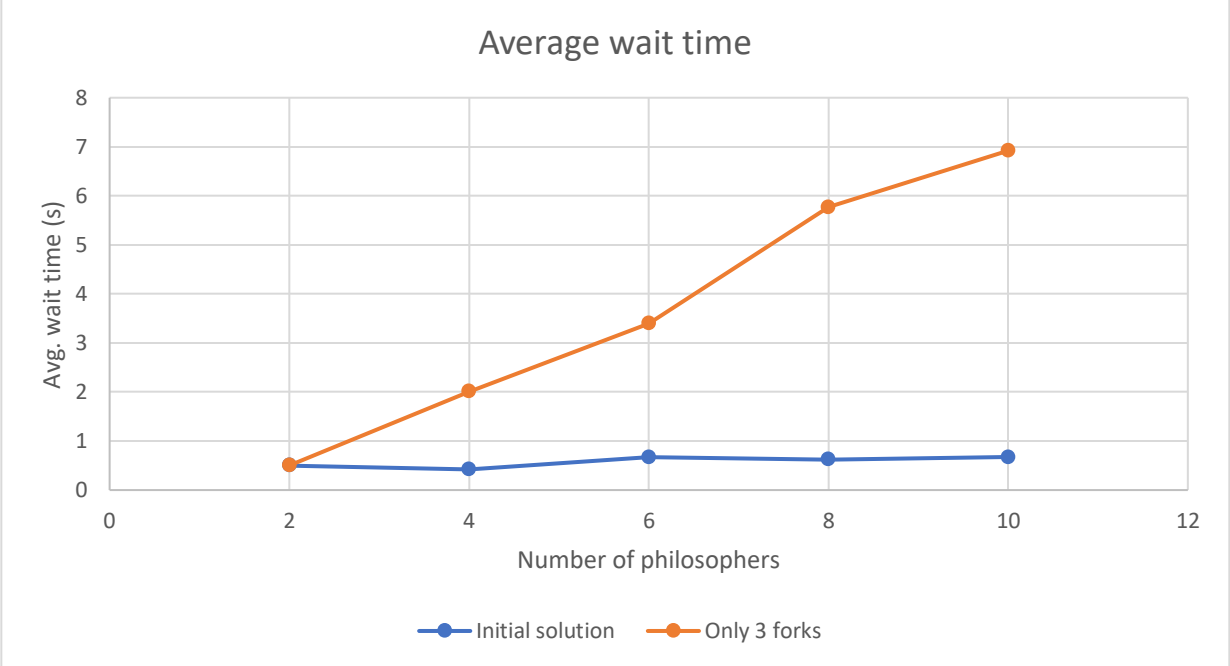

Figure 2: Performance when only three forks are considered

b. What happens to your solution if we give one philosopher higher priority over the other philosophers?

For this part of the assignment the implementation of the solution should be fairly similar to the one before. We should modify the function *"try_forks"* so that when called by the philosopher with higher priority, instead of entering the while loop in case the neighbors are eating, it forces the neighbors to stop eating or keep them in hungry state and let the philosopher start eating.

One challenge would be how to force the eating process to stop, since the way it is implemented, a sleep process with a random time is used. Therefore, the sleep method should be interrupted somehow by the philosopher with higher priority. One possibility would be to use "*pthread_cond_timedwait*" instead of "*sleep*" while eating and, if not signaled by the high priority philosopher, implement a "try/catch" statement with "*setjmp*" and "*longjmp*", as indicated in here, that catches the timeout error.

In terms of performance there would be two contributing effects to a possible difference in behavior:

- Philosopher with priority would have an average waiting time of 0s (or close, only accounting for the awake of neighbors eating).
- Neighbor philosophers could have a larger wait time in case they get interrupted by the philosopher with priority and need to get back to the hungry state.

This two effects might make a difference in the total average wait time in case one is larger than the other, but they could also cancel each other out and not affect the average wait time at all.

c. *What happens to your solution if the philosophers change which fork is acquired first (i.e., the fork on the left or the right) on each pair of requests?*

We implemented this section of the assignment by modifying the function "*try_forks()*" with the following pseudo-code:

```
While(left or right fork not free){
        While(fork right is not free){
                Wait();
        }
        Grab right fork;
        If (Left fork free){
                Grab left fork;
        }else{
                Leave right fork;
                Wait();
        }
}
```

Code is included in the file *"Q1c.c",* where the pseudo code has been adapted to the implementation without any actual fork object (only state variables). Results are reported in Fig. 3, along with the previous sections. We observe a similar behavior to the first section, which is expected since we add more forks as we add philosophers and the fork preference condition only adds an small extra wait time in some random

situations. Since a philosopher needs both forks to eat, it makes little difference to wait for one or the other, in any specific order.

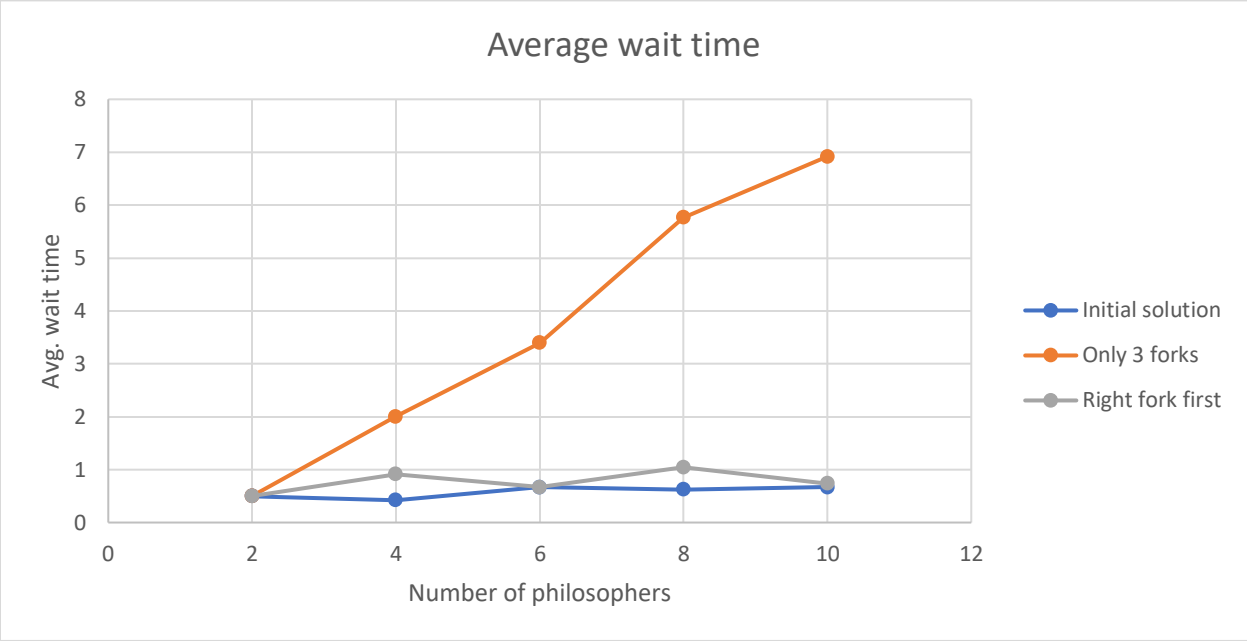

Figure 3: Performance when fork preference is considered