Assignment – 2

Question-2

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, though cannot start eating before getting both forks. 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.

a) What happens if only 3 forks are placed in the center of the table, but each philosopher still needs to acquire 2 forks to eat?
b) What happens to your solution if we give one philosopher higher priority over the other philosophers?
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?

Provide clear directions on how you tested your pthreads code so that the TA can confirm that your implementation is working. Provide these directions in a README file which instructs how to run through at least 12 iterations of updating the state of the philosophers and forks around the table. In your writeup, also discuss who was Edgar Dijkstra, and what is so important about this dining problem, as it relates to the real world. Make sure to discuss the algorithm that bears his name, Dijkstra's Algorithm. Cite your sources carefully.
*Written answers to the questions should be included in your homework 2 write-up in pdf format. You should include your C/C++ program and the README file in the zip file submitted.

Answer:

```
[prayags@rho ~]$ nano dining_philosophers.cpp
[prayags@rho ~]$ g++ -o dining_philosophers dining_philosophers.cpp -pthread
[prayags@rho ~]$ ./dining_philosophers 5
[Philosophers] P0:THINKING P1:THINKING P2:THINKING P3:THINKING P4:THINKING
[Forks]        F0:AVAILABLE F1:AVAILABLE F2:AVAILABLE F3:AVAILABLE F4:AVAILABLE

[Philosophers] P0:HUNGRY P1:EATING P2:HUNGRY P3:EATING P4:HUNGRY
[Forks]        F0:AVAILABLE F1:IN USE F2:IN USE F3:IN USE F4:IN USE

[Philosophers] P0:THINKING P1:EATING P2:THINKING P3:HUNGRY P4:EATING
[Forks]        F0:IN USE F1:IN USE F2:IN USE F3:AVAILABLE F4:IN USE

[Philosophers] P0:EATING P1:THINKING P2:EATING P3:HUNGRY P4:HUNGRY
[Forks]        F0:IN USE F1:IN USE F2:IN USE F3:IN USE F4:AVAILABLE

[Philosophers] P0:HUNGRY P1:EATING P2:THINKING P3:HUNGRY P4:EATING
[Forks]        F0:IN USE F1:IN USE F2:IN USE F3:AVAILABLE F4:IN USE

[Philosophers] P0:EATING P1:HUNGRY P2:EATING P3:THINKING P4:HUNGRY
[Forks]        F0:IN USE F1:IN USE F2:IN USE F3:IN USE F4:AVAILABLE

[Philosophers] P0:HUNGRY P1:EATING P2:HUNGRY P3:EATING P4:THINKING
[Forks]        F0:AVAILABLE F1:IN USE F2:IN USE F3:IN USE F4:IN USE

[Philosophers] P0:THINKING P1:EATING P2:THINKING P3:HUNGRY P4:EATING
[Forks]        F0:IN USE F1:IN USE F2:IN USE F3:AVAILABLE F4:IN USE

[Philosophers] P0:EATING P1:HUNGRY P2:HUNGRY P3:THINKING P4:HUNGRY
[Forks]        F0:IN USE F1:IN USE F2:AVAILABLE F3:AVAILABLE F4:AVAILABLE

[Philosophers] P0:EATING P1:HUNGRY P2:THINKING P3:EATING P4:THINKING
[Forks]        F0:IN USE F1:IN USE F2:AVAILABLE F3:IN USE F4:IN USE


[Philosophers] P0:THINKING P1:EATING P2:HUNGRY P3:THINKING P4:EATING
[Forks]        F0:IN USE F1:IN USE F2:IN USE F3:AVAILABLE F4:IN USE

[Philosophers] P0:THINKING P1:THINKING P2:EATING P3:THINKING P4:EATING
[Forks]        F0:IN USE F1:AVAILABLE F2:IN USE F3:IN USE F4:IN USE
```

2 (a):

Since there are only three forks and each philosopher needs two forks to eat, only one philosopher can eat at a time. Since there are usually five philosophers in this setup, this drastically limits the amount of things that can happen at the same time. The chance of a deadlock goes up, especially if all the thinkers try to grab two forks at the same time. If each philosopher grabs one fork, they will all be waiting for a second fork that they won't let go of, causing a deadlock. Chances are that you can get stuck or at least seriously limit concurrency, in such a situation starvation is more possible, especially for the philosophers who may not be fast enough or "lucky" to get the first fork. Philosophers might have to wait forever if their neighbors keep taking the forks.

2 (b):

If a particular philosopher has a higher (thread) scheduling priority, the scheduler may run that philosopher more frequently or for longer lengths. That philosopher can get the forks they need more often and sooner. This can cause the other philosophers to starve if the high-priority philosopher keeps using the forks again and again. In many scheduling systems that focus on real-time or priority, there are methods like priority inheritance or priority limits that can reduce the problem of indefinite blocking without which the waiting time would be longer for less important tasks.

2 (c):

If a philosopher switches between starting with the left or right hand in an unpredictable way, it increases the chances of a deadlock or livelock. This happens because there's no consistent order to follow, which helps avoid situations where they wait on each other endlessly.
In traditional solutions, following a specific order (like always picking up the left fork before the right one) helps prevent deadlocks, which are situations where no progress can be made. Changing the order can lead to situations where all thinkers take one fork at the same time and then wait for the other fork, causing a deadlock.

Who was Edgar Dijkstra?

Edsger Wybe Dijkstra was a Dutch computer scientist who made important contributions to programming, software engineering, and the formal understanding of algorithms. Some of his most significant achievements:

- Dijkstra's Algorithm: It efficiently finds the shortest path from a single source vertex to all other vertices in a weighted graph. It is mainly used in google maps for finding the shortest route between the 2 locations. It is also for optimal network pathfinding.
- Structured programming: He advocate for structured programming, which emphasized breaking programs into well-organized, modular components instead of relying on unrestricted jumps (goto statements).
- Semaphores & Concurrency Control: In 1965, Dijkstra introduced semaphores, a synchronization primitive used in operating systems and concurrent programming. It helped in thread synchronization (prevent race conditions while multi-threading) and Data Base Management System (ensured multiple transactions don't modify the same data simultaneously)

The Dining Problem is important for the following reasons:

- It shows basic problems with multiple processes running at the same time: deadlock, hunger, livelock, and the difficulty of safely sharing limited resources.
- In a network, several devices might need to share bandwidth, use communication channels, or access cloud services. If devices are not properly synchronized or scheduled, some may use up all the resources while others don't get enough, leading to worse system performance. By studying this problem, engineers create fair resource allocation strategies in networking, cloud computing, and large-scale distributed systems such as Google Spanner and blockchain consensus mechanisms

Dijkstra's Algorithm:

Dijkstra's Algorithm is a method used to find the quickest path from one point to another in a graph where the connections (edges) do not have negative values.

- The algorithm begins at a starting point and looks at nearby points, adjusting the estimated lengths to each accessible point.
- It keeps checking the vertex that is closest to the starting point, adds it to the list of vertices that have been checked, and updates the distances for its neighbors that haven't been checked yet.
- The process stops when all points are visited or if the shortest distance to any unvisited point is infinite, meaning there is no path.
- It is essential in computer science for things like routing, map tracking (GPS, Google Maps), and managing network traffic.

Sources:

1) **Dijkstra's Dining Philosophers Problem**:
   - Dijkstra, Edsger W. *"Hierarchical ordering of sequential processes."*
   - Hoare, C. A. R. *"Communicating Sequential Processes."*
2) **Dijkstra's Algorithm**:
   - *"A note on two problems in connexion with graphs."* Numerische Mathematik 1 (1959): 269-271.
3) **Structured Programming and 'Go To'**:
   - *"Go To Statement Considered Harmful."* Communications of the ACM 11, no. 3 (1968): 147-148.
4) **Pthreads**:
   - Butenhof, David R. *Programming with POSIX Threads.* Addison-Wesley, 1997.