Operating Systems Deadlock lab report

Andrew Bergman

09-19-2023

1 Lab 5 modifications

I made minor changes to my lab five code for this lab. Since deadlocks were already possible in my code, I added a delay between picking up chopsticks which would guarantee a deadlock right away. This would ensure that all the philosophers would pick up a chopstick right away and wait enough time for all others to also pick up a chopstick. I got this idea from a classmate while we were discussing our different implementations. I also lowered the eating and thinking times to one second so that the philosophers access the chopsticks as much as possible. Notably I cannot make that time zero because of how I measure time in my simulation.

My code relies on aggregating the amount of time that a philosopher eats and thinks in order know when to end the simulation. For example, if a philosopher eats for 10 seconds, thinks for 20, then a total of 30 seconds have passed. If the simulation time is set to 30 seconds, instead of going to eat the simulation ends. Thus if I set the eat and think time to zero, the simulation will never end since the aggregate will never surpass the simulation time and the simulation will act as if time does not pass.

Other modifications to my code where minor and I shall list them here: I added two functions one that runs the simulation with a deadlock and the other that runs without deadlocking, removed magic constants, and made the code more readable.

2 Solving the Deadlock problem

The method I chose to implement was the circular wait prevention method that we discussed in class. Each chopstick is numbered 0 to 4, and each Philosopher must first pick up the chopstick with the lower value before the other one. This prevents a circular wait and thus a deadlock.

In [1], this method is discussed as an asymmetric solution where philosophers are characterized as a "lefty" or a "righty" depending on the side they prefer to pick up a chopstick first. Two theorems are given, the first proving that any arrangement of "lefties" and "righties" with at least one of each will avoid deadlock and the other stating that any arrangement of ”lefties” and ”righties” with at least one of each will prevent starvation.

In my implementation of the above method, four of the philosophers are ”lefties” and one is a ”righty”. Because of the theorems in [1], this arrangement will prevent deadlock and starvation because there is at least one of each type.

3 Analysis and Comparison

The circular wait method to deadlock avoidance has the downside that resources might go underused. At any given time, the maximum number of philosophers eating is two. However, for any arrangement of ”lefties” and ”righties” it is possible to find a configuration of philosophers with chopsticks where only one is eating. Thus this method is not maximally efficient in concurrency or resources.

A benefit to this method is a lack of overhead from the operating system. Since we are avoiding deadlock instead of detecting or preventing it there is no overhead for the OS to perform.

Another solution to this problem discussed in [1] is to use a modified queue. The main difference is that a philosopher may leave the queue even if they are not at the front as long as they are not blocked by a philosopher currently eating. The process works like this: when a philosopher wants to eat, they are put into the queue. A Monitor checks if the first philosopher in the queue can eat right now. If they can, they are allowed access to the chopsticks. Each subsequent philosopher in the queue is similarly handled. If a philosopher is blocked, then the next one in the queue is considered.

[1] Offers explanations that this queuing method prevents deadlock and starvation. It is also resource efficient because it allows two philosophers to eat at any given time. The downside to this solution is OS overhead, because it must maintain the queue and run checks.

This queue method is similar to the deadlock prevention method we discussed in class, with resource vectors and matrices. Only allowing a process to gain resources when it will not allow a deadlock.

References

[1] Armando R. Gingras. “Dining philosophers revisited”. In: *ACM SIGCSE Bulletin* 22.3 (1990), p. 21. DOI: https://doi.org/10.1145/101085.101091.