DT038G Project

Solutions to the dining philosophers problem

Computer Science
Introduction to Operating Systems (DT038G)

Joel Edström

Mittuniversitetet
MID SWEDEN UNIVERSITY

Abstract

A number of philosophers (typically five) sits at at table with an equal number of forks lying evenly spaced between them. To be able to eat they need to pickup both of their closest forks. If every philosopher picks up their right fork at the same time, they wont be able to continue, because everyone left fork is busy.

This problem is called the Dining Philosophers problem. It's a common computer science problem that we'll explore some different solutions to, using a benchmarking and testing framework uniquely for this problem.

Keywords: Deadlock, Mutex, Synchronization, Python.

Table of Contents

1 Introduction

A common example problem in the field of concurrent algorithms is the dining philosophers problem[1] which illustrates the various issues with having multiple independent entities (the philosophers) compete for a number of non-sharable resources (the forks). There are multiple solutions to this problem and some of those will be explored, tested and benchmarked against each other.

1.1 Background and problem formulation

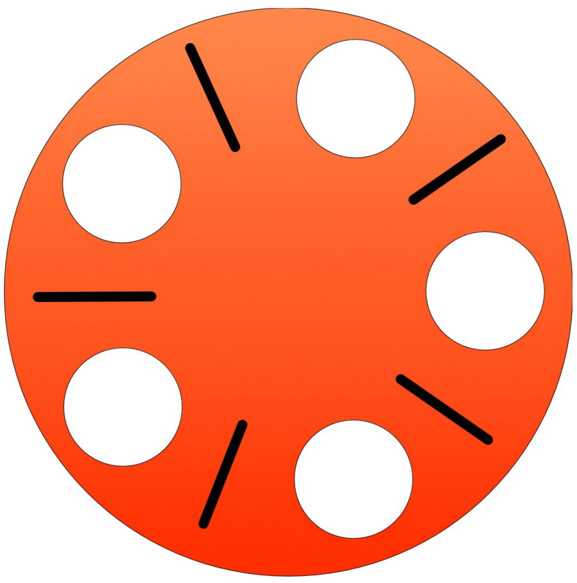

Figure 1: Dining philosophers problem illustration. A table for five philosophers.

In the dining philosophers problem there is an equal number of forks and philosophers. Each philosopher can access the closest forks to his/hers immediate left and right, as illustrated by Figure 1. To be able to eat the a philosopher requires both forks.

A philosopher basically has three states:

- The thinking state, where they will spend some time thinking.
- The hungry state, where they will attempt to pick up their forks.
- The eating state, where they will eat for a while.

They will attempt to cycle through those states in that order all the time independently of each other. But since they share forks with their neighbors, it wont be fully independent.

If they were truly independent they wouldn't spend any time at all in the hungry state because they would always be able to pick up their forks. And thus would be 100% "productive" with no unwanted downtime. We'll come back to this this metric in Chapter 2, and it will be used later as a benchmark to test different solutions to the problem.

1.2 Scope

This report is missing multiple good solutions that would have been interesting to explore, and the effect of different ratios than 1:1 between the philosophers being in a thinking and an eating state is ignored.

1.3 Outline

Chapter 2 starts with some theory then explores some solutions to the dining philosophers problem.

Chapter 3 comes back to those solutions and described implementations of them. As well as a general testing and benchmarking framework for comparing different solutions.

Chapter 4 shows the benchmark results.

2 Theory

There are two main issues that can occur with the dining philosophers problem:

- Deadlock: If for example all philosophers pick up their left for at the same time, the system reaches a state where it can't make anymore progress, and that is called a Deadlock[2].

- Starvation: Depending on the solution algorithm in conjugation with the Operating System (OS) scheduler[2] some of the philosophers may reach various levels starvation(both technically and literally)[2] due to not being able to grab both forks as often as the other philosophers.

Most Operating systems provides many tools that help when solving a problem involving these kinds of problems. The mutex(Mutual Exclusion) lock [2] being the most common.

A mutex lock is an OS construct that are available to user level programs. After a instance has been constructed it has two main operations:

- Acquire Lock: Attempts to acquire exclusive access to the lock. The calling thread will block if its not currently available.

- Release Lock: Released the lock which makes it available for others to acquire. If there are already other threads blocking on it, one of them will be awoken by the release.

A condition variable[2] combined with a mutex lock, gives the threads some further tools for communication. Consider the case of a producer/consumer pair (of some arbitrary data) with a boolean flag indicating if there is produced data ready to consume, protected by a mutex lock with a condition variable.

The consumer would acquire the mutex, check the boolean flag, and if data is available consume it, change flag and release the mutex. But what if the boolean flag indicated that no data was available: It would have to keep looping and consuming unnecessary computing resources.

That's where the condition variable comes it, instead of releasing the mutex the consumer would first use the wait operation on the condition variable. Which would implicitly release the lock and block on the condition variable in one atomic operation.

Later on the producer would produce the data, acquire the lock, change the flag to indicate data being available, then use the signal operation on the condition variable and lastly releasing the lock.

This would wake up the consumer with the mutex lock already being acquired automatically by the OS. Now the consumer has the data it requires to continue running. After using it it would release the lock again. And the process continuous like this.

2.1 Different solutions to the dining philosophers problem

To benchmark the different solutions we would like to explain the productivity metric introduced lightly in the introduction:

$$P=(T_{eating}+T_{thinking})/T_{total}$$

Where T_{eating} and $T_{thinking}$ is the time a philosopher has spent eating and thinking respectively. T_{total} is the total test time. Productivity P is the percentage of time spent doing anything productive like thinking or eating, with the rest of the time being wasted (hungry and blocking).

2.1.1 Deadlocking non-solution

The solution described in the introduction where each philosopher always picks his/her left fork first, with no other forms of protection, isn't a very good solution but can form a basis for other solutions, where they are variations based on it.

2.1.2 Simple Mutex lock protected solution

This solution is exactly the same as the previous solution, but protects the code when the philosophers are hungry and will start grabbing forks by a mutex lock. Thus only allowing one philosopher to eat concurrently, which will solve the deadlocking issue, but will lead to bad productivity, with philosophers spending lots of time waiting.

2.1.3 Odd/Even philosopher solution

What if instead of all philosophers always picking their left fork first, they pick either their left or right depending on their position around the table? Well that's a good solution that will protect against deadlock.

The state in 2.1.1 can never be reached because after everyone has attempted to pick their first forks, there will still be some free forks due to any philosopher pair sitting next to each other either attempting to pick the same fork between them first, or leaving it free and attempting to pick the forks on their far sides.

2.1.4 Condition variable solution

This solution uses a mutex to protect the state of the forks (whether they are free), and an associated condition variable to wake up philosophers when some forks become free, giving them a chance to check if both their required forks have become free. Only when both forks are available they will grab them and eat.

3 Methodology

3.1 Implementation

A simple testing and benchmarking framework that handles the setting up a common environment to test different solution implementations was built using the Python programming language.

The main part of the framework is the class PhilosopherEngine that accepts these parameters on construction:

- philosopherClass: Subclass of the abstract class Philosopher that implements a certain solution to the problem.
- count: Number of philosophers(and forks) to simulate
- minRuntime: It will run the experiment at least this long in seconds.
- maxEatThinkTime: When a philosopher is going to eat or think it will choose a random number between 0 and this parameter to decide how long to stay in that state.

There is also a class that represents a fork, implemented on top of a mutex lock, which in addition to having a grab(acquire) and release operation has an operation that lets you check if the fork is free.

This is a minimum implementation of the Philosopher class, that implements the deadlocking non-solution from 2.1.1:

```
class Deadlocking(Philosopher):

    def grabForks(self):
        self.leftFork.grab()
        self.rightFork.grab()

    def releaseForks(self):
        self.leftFork.release()
        self.rightFork.release()
```

By subclassing Philosopher you get access to your left and right fork, using the instance variables self.leftFork and self.rightFork. You will also get access to the philosopher index(position) at the table with self.index, which enables a clean implementation of the Odd/Even philosopher solution in this framework.

The solution using a condition variable was implemented like this:

```
class ConditionVariable(Philosopher):
    lock_and_cv = threading.Condition()

    def grabForks(self):
        with self.lock_and_cv:

            while not  (self.leftFork.free() and
                        self.rightFork.free()):
                self.lock_and_cv.wait()


            self.leftFork.grab()
            self.rightFork.grab()


    def releaseForks(self):
        with self.lock_and_cv:
            self.leftFork.release()
            self.rightFork.release()

            self.lock_and_cv.notifyAll()
```

The python class threading.Condition contains a mutex and a condition variable. Using the with statement we can safely acquire and later on automatically release the mutex.

After having acquired the mutex this solution algorithm will check if both forks are available before continuing, and if not call the wait operation on the condition variable. This frees up the mutex for other threads while its waiting for the forks to become available.

When a thread releases its forks, it will signal all waiting threads to wake up using the signalAll method. It's not really necessary to wake up all threads, but simplifies the algorithm significantly at the cost of some performance.

4 Results

The different algorithms compared by the previously introduced productivity metric, see Figure 2. Raw data available in Appendix A

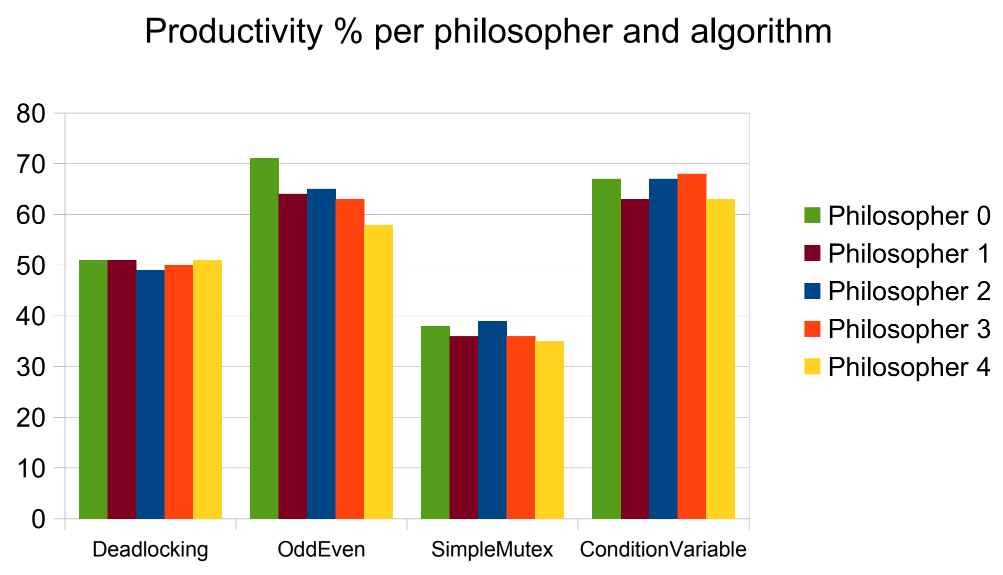

Figure 2: Productivity compared between the different algorithms

The deadlocking algorithm doesn't really deserve to participate in the performance picture because deadlocking algorithms aren't usually acceptable. But it was included just for comparison.

It was discovered however that the risk of a deadlock is extremely low, probably due to the granularity in the thread scheduling, and the philosophers always grabbing the left and right fork in quick succession. But by inserting a small pause(see code just below) between the grabbing the forks in the Deadlocking implementation, a deadlock could be achieved at every attempt.

```
def grabForks(self):
        self.leftFork.grab()
        time.sleep(0.01) # increase chance of deadlock
        self.rightFork.grab()
```

5 Conclusions

Two decently performing solution algorithms were found. After many runs it looks like the even/odd solution always gives philosopher 0 a bit more productivity. The other philosophers are slightly starved.

This could vary depending on Operating System, scheduler, and hardware. And should be explored further.

The solution using a condition variable doesn't appear to have this problem, and consistently appears to give slightly higher total productivity (66% vs 64%) which makes this algorithm the best of the algorithms tested here.

5.1 Possible Future Work

A better designed testing framework could be designed that would automatically change different parameters like: the delay between left and right fork as described at the end of Chapter 4, basically hammering the algorithm with different timings(delays), and different ratios than 1:1 between time spent eating and thinking. The goal would be to get better data, as well as expose deadlocks.

Such a framework could be used in the class and having the students make competing algorithms, that could be tested against each other.

6 References

[1] Wikipedia, ”Dining philosophers problem”,
http://en.wikipedia.org/wiki/Dining_philosophers_problem
Retrieved 2014-06-07.

[2] A. Silberschatz, P. Galvin, and G. Gagne: *Operating System Concepts*, 8th edition, 2008.

Appendix A: Output of testing framework

Deadlocking non-solution output

```
   Philosopher implementation: Deadlocking
     Times spent (s) during a total runtime of 5.03 s
     Blocking       Eating       Thinking     Productive %
0:   2.27           1.26         1.31         51%
1:   2.25           1.26         1.31         51%
2:   2.38           1.26         1.21         49%
3:   2.32           1.27         1.23         50%
4:   2.29           1.26         1.27         51%
sum: 11.51          6.32         6.34         50%
```

Odd/even solution output

```
     Philosopher implementation: OddEven
     Times spent (s) during a total runtime of 5.04 s
     Blocking       Eating       Thinking     Productive %
0:   1.16           1.79         1.79         71%
1:   1.53           1.60         1.63         64%
2:   1.53           1.61         1.66         65%
3:   1.65           1.59         1.56         63%
4:   1.85           1.41         1.53         58%
sum: 7.72           8.00         8.17         64%
```

Simple mutex solution

```
     Philosopher implementation: SimpleMutex
     Times spent (s) during a total runtime of 5.07 s
     Blocking       Eating       Thinking     Productive %
0:   2.96           0.95         0.95         38%
1:   3.07           0.86         0.94         36%
2:   2.91           0.97         1.02         39%
3:   3.07           0.95         0.88         36%
4:   3.11           0.91         0.87         35%
sum: 15.13          4.65         4.67         37%
```

Condition variable solution

```
     Philosopher implementation: ConditionVariable
     Times spent (s) during a total runtime of 5.03 s
     Blocking       Eating       Thinking     Productive %
0:   1.40           1.82         1.57         67%
1:   1.59           1.65         1.52         63%
2:   1.38           1.67         1.71         67%
3:   1.36           1.65         1.75         68%
4:   1.63           1.56         1.59         63%
sum: 7.37           8.35         8.14         66%
```