Lecture 10: February 23

Lecturer: Prashant Shenoy *TA: Vimal Mathew & Tim Wood*

10.1 Readers/Writers Problem

In computer science, the readers-writers problems are examples of a common computing problem in concurrency. The problem deals with situations in which many threads must access the same shared memory at one time, some reading and some writing, with the natural constraint that no process may access the data for reading or writing while another process is in the act of writing to it. In particular, it is allowed for two readers to access the share at the same time.

This is useful in many systems in order to optimize performance–if many threads want to read a data structure there is no need to limit them to accessing the data one at a time. However, synchroniztion still must be insured in the case of a writer–only one writer should be able to write at a time, and no reader should run concurrently with the writer. Clearly, using a simple set of locks like we have described previously will be inefficient.

10.1.1 Three types of solutions

There are several different solutions which can be used to build a Readers/Writers system.

1. Reader preferred: waiting readers go before waiting writers. A constraint is added so that no reader shall be kept waiting if the share is currently opened for reading.
2. Writer preferred: waiting writers go before waiting readers. A constraint is added so that no writer, once added to the queue, shall be kept waiting longer than absolutely necessary.
3. Neither preferred: try to treat readers and writers fairly (a simple queue is not good enough; we want parallel readers whenever possible).

In fact, the solutions implied by the first two approaches - *reader preferred* and *writer preferred* - result in starvation; the *reader preferred problem* may starve writers in the queue, and the *writer preferred problem* may starve readers. Therefore, *neither preferred* is sometimes proposed, which adds the constraint that no thread shall be allowed to starve; that is, the operation of obtaining a lock on the shared data will always terminate in a bounded amount of time. Solutions to this will necessarily sometimes require readers to wait even though the share is opened for reading, and sometimes require writers to wait longer than absolutely necessary.

Look at the lecture slides for pseudocodes for the three different solutions

10.1.2 Readers/Writers in Java

The Java programming language can be used to solve the Readers/Writers problem in several different ways. Monitors can be used in order to synchronize the methods that control access for readers and writers.

Alternatively, Java supports the idea of a read/write lock using the *ReadWriteLock* class. This is a special lock class that has different lock calls used by either readers or writers. The lock automatically enforces the rule that multiple readers can obtain the lock at once, but only one writer can be allowed.

10.2 Dining Philosophers Problem

The Dining Philosophers Problem is an illustrative example of a common computing problem in concurrency. The dining philosophers problem describes a group of philosophers sitting at a table doing one of two things - eating or thinking. While eating, they are not thinking, and while thinking, they are not eating. The philosophers sit at a circular table with a large bowl of spaghetti in the center. A chopstick is placed in between each philosopher, thus each philosopher has one chopstick to his or her left and one chopstick to his or her right. As spaghetti is difficult to serve and eat with a single chopstick, it is assumed that a philosopher must eat with two chopsticks. The philosopher can only use the chopstick on his or her immediate left or right.

The philosophers never speak to each other, which creates a dangerous possibility of deadlock. Deadlock could occur if every philosopher holds a left chopstick and waits perpetually for a right chopstick (or vice versa). Originally used as a means of illustrating the problem of **deadlock**, this system reaches deadlock when there is a 'cycle of unwarranted requests'. In this case philosopher P_1 waits for the chopstick grabbed by philosopher P_2 who is waiting for the chopstick of philosopher P_3 and so forth, making a circular chain.

Starvation (quite literally) might also occur independently of deadlock if a philosopher is unable to acquire both chopsticks due to a timing issue. For example there might be a rule that the philosophers put down a chopstick after waiting five minutes for the other chopstick to become available and wait a further five minutes before making their next attempt. This scheme eliminates the possibility of deadlock (the system can always advance to a different state) but still suffers from the problem of **livelock**. A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, but none make any useful progress. If all the philosophers appear in the dining room at exactly the same time and each picks up their left chopstick at the same time the philosophers will wait five minutes until they all put their chopsticks down and then wait a further five minutes before they all pick them up again.

The lack of available chopsticks is an analogy to the locking of shared resources in real computer programming. Locking a resource is a common technique to ensure the resource is accessed by only one program or chunk of code at a time. The challenge occurs when there are multiple resources which must be acquired individually. When several programs are involved in locking multiple resources, deadlock can occur. For example, one program needs two files to process. When two such programs lock one file each, both programs wait for the other one to unlock the other file, which will never happen.

10.2.1 Solution to the Dining Philosophers problem

A naive solution is to first wait for the left chopstick using a semaphore. After successfully acquiring it, wait for the right chopstick. After both chopsticks have been acquired, eat. When done with eating, release the two chopsticks in the same order, one by one, by calls to *signal*. Though simplistic, this solution may still lead to a deadlock when every philosopher around the table is holding his/her left chopstick and waiting for the right one.

A variation of this solution, makes a philosopher release the left chopstick, if it can't acquire the right one. But even this can lead to livelock because of timing issues (as explained before). The correct solution is to let a philosospher acquire both the chopsticks or none. This can be done within a *monitor* wherein only

one philosopher is allowed to check for availability of chopsticks at a time (within a *test* function). See the lecture slides for the complete pseudocode.