EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 5: September 8

Lecturer: Vijay Garg *Scribe: Nishanth Shanmugham*

5.1 Topics

The topics covered in this lecture are:

- Semaphores
  - Why / what / motivation
  - Usage
  - Implementation
  - Java code on GitHub
    - BinarySemaphore
    - CountingSemaphore
- Consumer-producer problem
- Reader-writer problem
- Dining philosopher's problem

5.2 Semaphores

Semaphores provide a means to achieve mutual exclusion.

5.2.1 Why / what / motivation

Think back to Peterson's algorithm. We performed a busy-wait:

```
while (wantCS[1] && (turn == 1)) { no_op(); }
```

This wait consumes the CPU while waiting for the condition to become false. So core execution time is wasted. This busy-wait can be avoided with help from the operating system.

So how does the OS schedule threads?

- The scheduler puts RUNNABLE threads to RUNNING queue.
- If a RUNNING thread issues a blocking call (for example, I/O) then it is put in BLOCKED queue.
- Once the I/O operation returns, the blocked thread can be scheduled for running again by putting in the RUNNABLE queue.

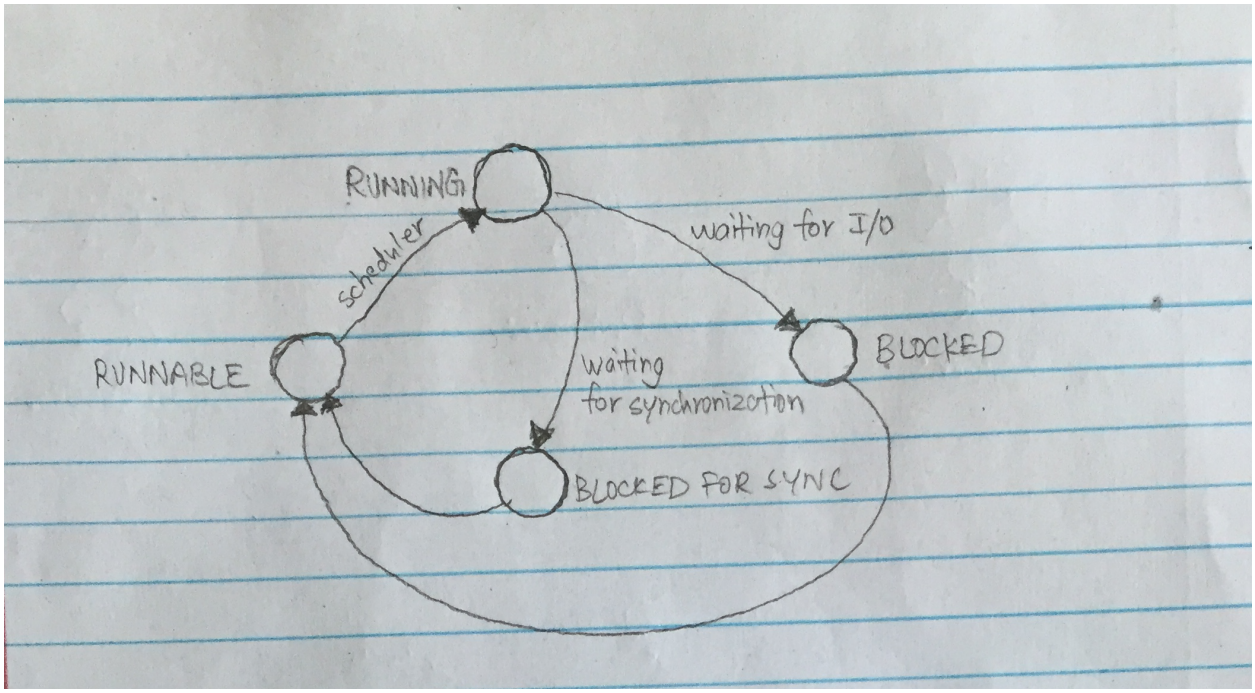

Figure 5.1: OS scheduling queues

- Similarly, instead of running no_op()'s, the OS should be able to take it out of the RUNNING queue and put into the BLOCKED queue waiting for synchronization.

5.2.2 Usage

Semaphores were introduced by Dijkstra in *The Operating System*. A semaphore has two operations defined on it:

- P(), or acquire()
- V(), or release()

That is, if `s` is a semaphore then `s.P()` and `s.V()` are the two possible calls.

5.2.3 Visualization

A simple way to visualize to semaphore is as a bottle with a marble inside it. The marble could either be present inside the bottle or not inside it the bottle.

If a thread wishes to enter a critical section, it should first obtain the marble before it proceeds to enter the CS. If the marble is present, it removes the marble and proceeds to enter the CS. If the marble is not present, it waits until the marble is present to try again.

The `value` variable in the implementation below corresponds to the marble.

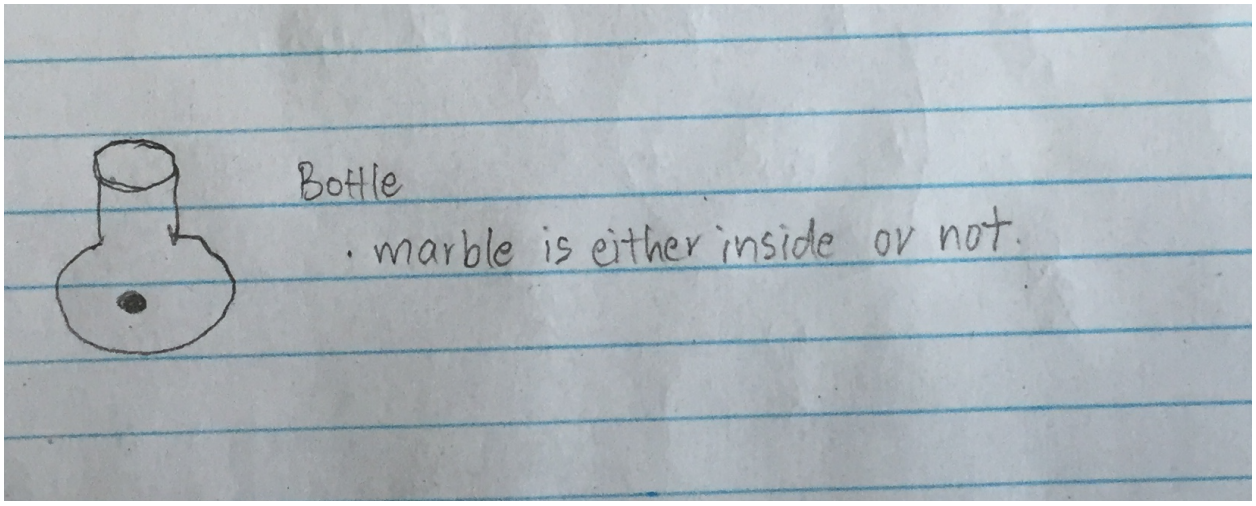

Figure 5.2: Semaphore

5.2.4 Implementation

A semaphore has one field:

```
value:  boolean, initially true
```

The implementation for `P()` is below. Note that both lines would need to execute atomically.

```
while (!value) { wait(); }
value = false;
```

The implementation for `V()` is:

```
value = false;
```

5.2.5 Discussion

But isn't it weird that we have to perform P() atomically? Essentially, it appears like we need to use mutual exclusion to build mutual exclusion.

We will have to use busy-wait to achieve atomicity of P().

- For programmer's CS, we will use semaphore.
- For semaphore's CS, we will use busy-wait.

This is an okay compromise. The reason to not use busy-wait for programmer's CS are that programmer's CS are generally longer and may contain bugs in the implementation. On the other hand, semaphore's CS is known to be small. Thus using a semaphore saves system resources.

5.2.5.1 notify

Sometimes, it might be necessary to wake up one of the sleeping threads once the current threads completes it work in the CS. For this, we use `notify()`. The `notify()` call does not guarantee the order in which threads are woken up—it simply wakes up any one thread.

To guarantee order, Java provides a fairness parameter. But generally, it is good practice to not expect order guarantees in concurrent programs that do not require ordering.

5.2.5.2 Monitors

Semaphores might hinder code readability since semaphores are used for both *mutual exclusion* and *conditional execution*. In addition:

- The order of `P()` calls is important to avoid deadlocks.
- `P()` calls should be paired with corresponding `V()` calls to avoid deadlocks.

Monitors provide a suitable mechanism to handle these, but are explored in a future lecture.

5.2.6 Java code

5.2.6.1 BinarySemaphore

The `BinarySemaphore` class on GitHub implements the kind of semaphore described above.

Of note, the Util.myWait call on line 8 puts the current thread in the blocked queue. Also, the synchronized keyword in the method signatures guarantee that only one thread is executing the method at any given time. This provides the required atomicity.

5.2.6.2 CoutingSemaphore

`CountingSemaphore` allows multiple threads to be in the CS at a given time. The value field is now an int instead of a boolean.

5.3 Producer-consumer problem

5.3.1 Description

There is a producer thread and a consumer thread. The producer deposits to shared buffer, and the consumer reads from the shared buffer. The constraints are:

- Mutual exclusion (no concurrent read/write to shared buffer)
- Conditional synchronization. The consumer has to wait for buffer to be non-empty before read; the producer has to wait for buffer to be non-full before depositing.

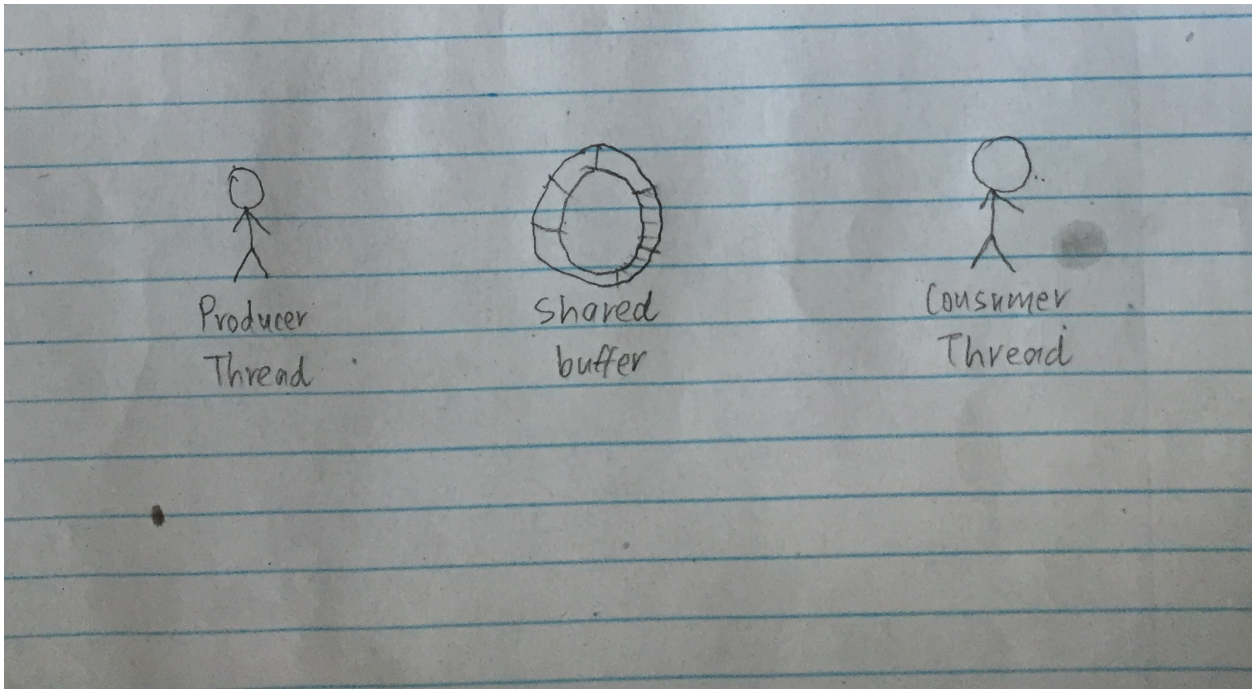

Figure 5.3: Producer-consumer

5.3.2 Implementation

`BoundedBuffer` class on GitHub contains a solution to this problem. The noteworthy observations from the code are:

- The mutex is a BinarySemaphore. It is a common convetion to use a BinarySemaphore for mutual exclusion.
- The conditional synchronization is provided by the CountingSemaphores, namely `isEmpty` and `isFull`.
- The other thread is woken up / notified by the `V()` calls on lines 15 and and 24.

Question: Is it okay to change the order of the calls on lines 10 and 11? No, here's an example execution order that results in deadlock if the order was changed.

- The buffer is full. The producer grabs the mutex. Then the producer waits for the buffer to be not full so that it can deposit.
- The condition that the producer is waiting for will never be fulfilled.
- This is because the consumer will not be able to consume from the buffer, because the mutex is held by the producer.

In the correct order of code on GitHub, this will not happen because consumer and producer only grab the mutex if their entry conditions are already satisfied.

5.4 Reader-writer problem

5.4.1 Description

The problem aims to establish a protocol for multiple readers and writers to safely access a shared database. The constraints are:

- No read-write conflict.
- No write-write conflict.
- Read-read conflict is allowed.
- There is no need to guarantee starvation freedom for writers. It is okay for readers to read repeatedly.

5.4.2 Implementation

`ReaderWriter` class on GitHub contains a solution to this problem.

The `startWrite` and `startRead` methods are the entry protocol for writers and readers respectively. Similary, `endWrite` and `endRead` are the exit protocols for writers and readers respectively.

`startWrite` and `endWrite` are simple. Writers simply need to acquire `wlock` before they can write and release it when done writing. The acquiring of the BinarySemaphore wlock ensures that only one writer is writing at any given time.

Readers on the other hand need to ensure that no writers exist before they can read. The first reader to enter has to wait until all writers leave. When the first reader enters it grabs the wlock (line 8) ensuring no writers can enter until the last exiting readers release wlock (line 14). Readers after the first reader simply need to enter and exit. The last reader to exit has to release wlock as described before.

To track the number of readers, the numReaders variable is used. It is incremented on entry and decemented on exit. It is guarded by mutex to prevent concurrent modification of the variable by readers.

5.5 Dining philosophers problem

5.5.1 Description

n philosophers sit around a circular dining table. There are n forks placed in between the philosophers. To eat, a philosopher requires two forks—the one of her left and the one of her right.

Philosophers switch between these states in order: *thinking* to , *hungry* to *eating*, and *eating* back to *thinking*.

The goal is to design a protocol with the following constraint: Two neighboring philosophers cannot eat at the same time.

Philosophers should use an `acquire` call to acquire forks and a `release` call when done eating.

The Philosopher class on GitHub represents a philosopher. DiningPhilosopher provides a simple (but incorrect) implementation of Resource.

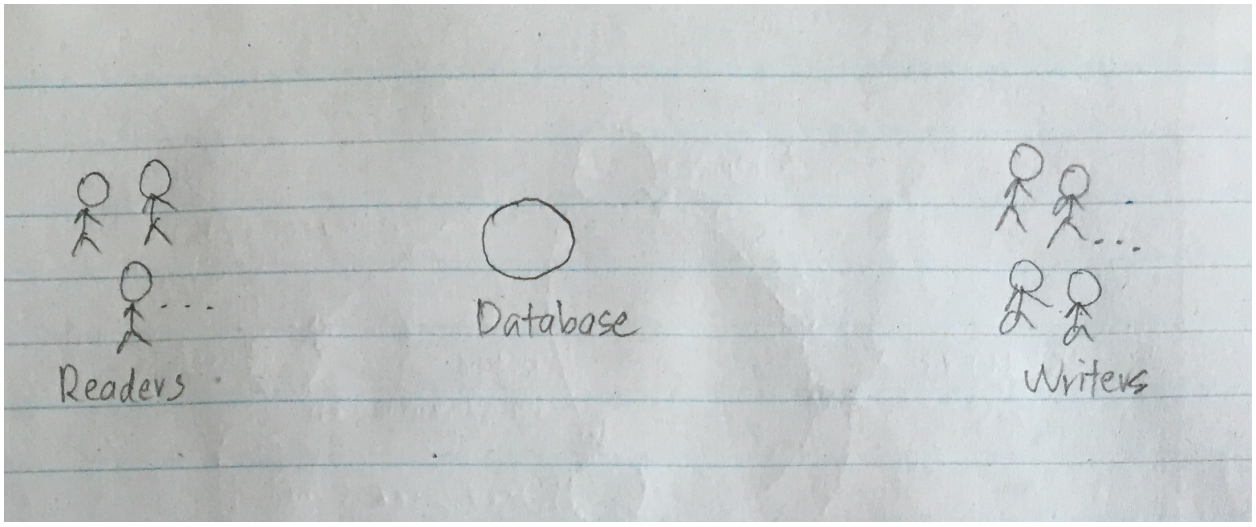

Figure 5.4: Reader-writer

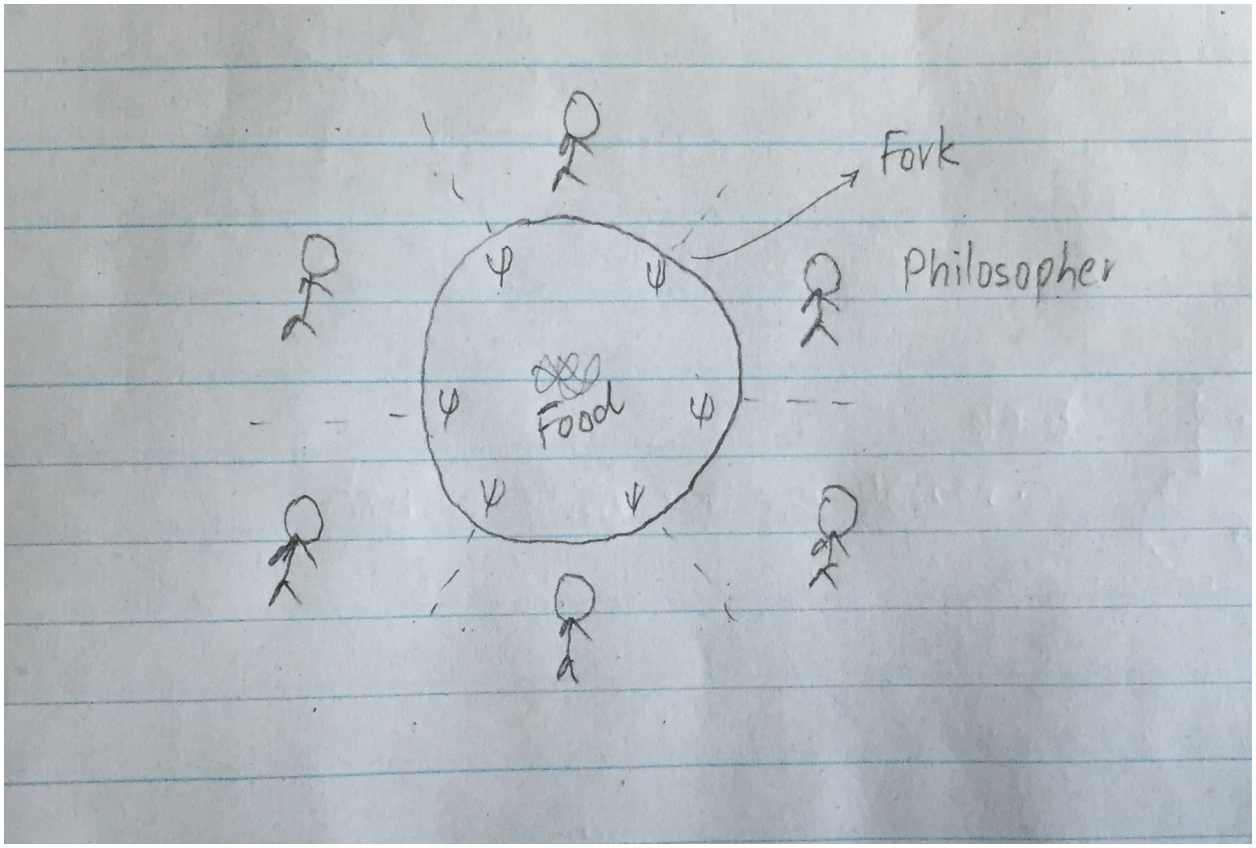

Figure 5.5: Dining philosophers

It is incorrect because it can result in a deadlock. In the acquire method, philosopher's first grab the left fork then the right fork. In a circular dining table, it could happen that the philosophers all grab their left fork first. When the philosophers now try to obtain the right fork, they will fail to do so because it has been obtained by their neighbor to the right (who grabbed it initially as her left fork).

This is problem arises because of symmetry.

5.5.1.1 Fixes

Possible fixes to the DiningPhilosopher class are listed below.

- Breaking symmetry: exactly one philosopher will be instructed to grab her right fork before the left. This will prevent the deadlock.
- Restrict number of philosophers that can be eating at any given time to less than n: For instance, there could be a precursor step of standing up before acquiring a fork. At that point, we can restrict that at most less than n philosophers can be standing at any instant.

References

[GitHub] Multicore source code, *https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore*