Homework 4 Commentary
COMP3151 T2 2021

Vincent Jackson

August 6, 2022

1 Smoker's Protocol

In the smoker's protocol, we asked you to complete a six agent resource distribution problem using semaphores.

The Smoker's protocol has two correctness conditions which you needed to meet. The first correctness condition was that the protocol doesn't deadlock.

The second correctness condition is induced by the description of the problem: each smoker requires resources they must procure from an agent, which provides these resources. Thus, the protocol must not produce resources out of thin air.

Another resource condition we might wish to have, but is less about correctness, is that we don't waste resources; e.g. that we don't pull three matches from the agents for every one match that actually gets to a smoker.

I have attached a promela file which models the protocol, which you can use to show deadlock freedom, and which also contains an LTL formula which ensures that all resources come from somewhere, and a formula which ensures resources aren't stockpiled without being used.

2 Semaphores

2.1 Busy-wait Semaphore

To begin, there was a good deal of confusion about `Thread.yield()`. In Java, `Thread.yield()` is a hint to the scheduler that this thread wants to yield. Importantly, Java *can just ignore the hint*. Thus, a good rule of thumb for using `Thread.yield()` is that your algorithm must continue to work even if you replaced the yield with a no-op.

A second rule of thumb is that you should not loop inside a synchronised block when the action that would end the loop is synchronised on the same lock. The code in Figure 1 deadlocks, because once `V` starts to loop, `P` can never run and end the loop.

```
void synchronized P() {
  while (permits <= 0) {
    // Doesn't release the lock. As I said earlier,
    // replace Thread.yield with a no-op,
    // and the protocol still needs to work.
    Thread.yield();
  }
  permits--;
}

void synchronized V() {
  permits++;
}
```

Figure 1: Deadlock due to Yield not Releasing the Lock

Here are two correct solutions. The first Figure 2 puts all the logic into `P` and `V`. Another good solution Figure 3 extracts the decrement logic into a synchronised method.

2.2 Weak Semaphore

This was well done by most students. As wait *does* release the lock when called, we can safely use a loop inside the synchronised block.

An important point to make is that Java monitors behave slightly differently to what you might expect. With a classic monitor [1], notifying a waiting process will immediately wake up that process. Thus the solution in Figure 4 works. However, this is not the case in Java.

In Java, processes waiting on a monitor are not prioritised over other threads. Thus after waking up, the process *must* check the guard again, as in Figure 5.

2.3 Strong Semaphore

Strong semaphores require processes which call `P` are released in the order in which they called `P`.

Because Java doesn't prioritise processes that have been signalled to end their `wait`, we can't assume the processes in the queue are prioritised over those which have just called `P`. Thus in this solution, every process must be entered into the queue, regardless of whether `permits > 0`.

References

[1] C. A. R. Hoare. 1974. Monitors: an operating system structuring concept. Commun. ACM 17, 10 (Oct. 1974), 549–557. DOI:`https://doi.org/10.1145/355620.361161`

```
void P() {
  while (true) {
    // for better performance, check before actually trying
    while (permits <= 0) {
      Thread.yield();
    }
    synchronized(this) {
      if (permits > 0) {
        permits--;
        break;
      }
    }
  }
}

void synchronized V() {
  // synchronized because ++ is not necessarily atomic.
  permits++;
}
```

Figure 2: Very-Weak Semaphore Solution #1

```
  bool synchronized testAndDecr() {
    if permits > 0 {
      permits--;
      return true;
    } else {
      return false;
    }
  }

  void P() {
    while (!testAndDecr()) {
      Thread.yield()
    }
  }

  void synchronized V() {
    permits++;
  }
```

Figure 3: Very-Weak Semaphore Solution #2

```
void synchronized P() {
  if (permits < 0) {
    wait();
  }
  permits--;
}

void synchronized V() {
  permits++;
  notify();
}
```

Figure 4: Classical Weak Semaphore Solution (Doesn't work in Java)

```
void synchronized P() {
  while (permits < 0) {
    wait();
  }
  permits--;
}

void synchronized V() {
  permits++;
  notify();
}
```

Figure 5: Weak Semaphore Solution

```
// Use whichever Java queue implementation you want
Queue<long> queue = ...;

void synchronized P() {
  queue.add(this.getId());
  while (permits < 0 || queue.peek() != this.getId()) {
    wait();
  }
  queue.poll();
  permits--;
}

void synchronized V() {
  permits++;
  notifyAll();
}
```

Figure 6: Strong Semaphore Solution