ECS 150 Homework 3

Hardy Jones
999397426
Professor Levitt
Winter 2015

3 §3.23 No, the system cannot be safe from deadlock at this point. If any process requests all of their resources, they will not get the full resources. So running any process will lead to deadlock.

4 §3.24

- The computers can be still negotiating if both programs want to separate an animal.

  Assume Cinderella wants Woofer's house first, and Prince wants Woofer first. Their programs send the requests to the lawyers, the lawyers realize the requests would separate Woofer from Woofer's house, so the whole process is restarted. This continues indefinitely.
- Deadlock is possible if both programs request the same property each time. If both programs only want Woofer, then they both send the request for Woofer. This request is then canceled and started again, repeating the process.
- Starvation is only possible if one of the programs does not request an item every day. Otherwise, deadlock will prevent starvation.

5 §3.15

- If D asked for one more unit, the state would be unsafe. There would be only one free unit, and no person could request their total units. So the next request would deadlock.
- If C asked for one more unit, the state would be safe. There would be one free unit, so C would be able to then request its last unit, and it would then release all four units. That would allow either B or D to request their maximum, which would free up either five or seven additional units Having at least five free units allows any person to request their maximum units.

6 §3.18 In theory, the system is deadlock free any n. The system would just have to ensure that if it gave out five of the tape drives, it gave the sixth tape drive to one of the processes that already had one tape drive. Then that process could run to completion, and eventually release the tape drives for future processes. This may lead to starvation if one process runs indefinitely, but it would not lead to deadlock.

7 The new monitor can be written as follows.

```
Monitor rw
     var nr := 0, nw := 0
     var OKR, OKW : condition
    procedure REQ_READ( )
        if nw > 0 then wait(OKR)
        end if
        nr := nr + 1
        signal(OKR)
    end procedure
    procedure REL_READ( )
        nr := nr - 1
        if nr = 0 then signal(OKW)
        end if
    end procedure
    procedure REQ_WRITE( )
        if nr > 0 or nw > 0 then wait(OKW)
        end if
        nw := nw + 1
    end procedure
    procedure REL_WRITE( )
        nw := nw - 1
        if not empty(OKW) then signal(OKW)
        else if not empty(OKR) then signal(OKR)
        end if
    end procedure
End
```

We ensure that a reader only has access when the number of writers is 0 on lines 8 and 24.

We still allow multiple readers to have access at the same time.

We still ensure that a writer only has access exclusively.

We prioritize writers over readers by checking for `OKW` first on line 22. This ensures that all writers are given access before readers.

8 (a)
- When process 1 "negotiate"s it sets `flag[1]` to `true`. Then it enters the outer while loop. At this point in time `turn` is equal to `0` still. so it `turn not = 1` is true. Process 1 then goes and enters inner while loop. Process 1 does a busy wait checking that `flag[0]` is `true`. So process 1 does not enter its critical section.
- After process 0 has exited its critical section and set `flag[0]` to `false`, process 1 will exit the inner loop and set `turn` to `1`. Then it checks the outer loop condition. This check fails, so process 1 exits the outer loop. The next statement is the critical section. So process 1 enters its critical section after process 0 exits its critical section.

(b)
- One interleaving would be allowing process 1 to run first until it gets through the inner while loop. Since process 0 has not run yet, it will work its way through the outer while loop, but not enter the inner while loop. Before it gets a chance to set `turn` to `1`, let process 0 run through the outer while loop. Process 0 will not enter the while loop as `turn` is still `0`. So process 0 enters its critical section. Next, let process 1 continue. Process 1 will set `turn` to `1`, check the outer condition of the while loop, and exit the loop. Process 1 then enters its critical section.
  Thus, this algorithm does not solve the mutual exclusion problem.
- In order to see if deadlock is possible, it is necessary to check three of the four Coffman Conditions. We view `turn` and `flag` as resources: when `turn` equals `i`, process `i` is requesting/holding-on to that resource; when `flag[i]` is `true`, process `i` is requesting/holding-on to that resource.
  i. **Mutual Exclusion**
     `flag[0]` and `flag[1]` are resources that can only be held by process `0` and process `1` respectively.
  ii. **Wait For**
     Both processes hold on to their respective `flag` resource, and also require the `turn` resource.
  iii. **No Preemption**
     Both processes hold on to their respective `flag` resource, and they do so until they have completed the algorithm. Neither process can forcibly lose their `flag` resource.

  Since these three conditions are true, deadlock is possible.
  N.B. We needn't check the last condition as we are only concerned with the possibility of deadlock, not the existence.