Assignment 6, Operating Systems

Jay R Bolton

November 15, 2011

Ch 6 problems: 6.1, 6.2, 6.4, 6.5, 6.6, 6.11, 6.14, 6.15, 6.18

1. **6.1** 1) **Mutual exclusion** Only car 1 can access a, car 2 b, car 3 c, and car 4 d. 2) **Hold and wait** Once car 1 consumes a and car 2 consumes b, car 1 will wait on the release of b. 3) **No preemption** Cars cannot bump each other out of the way (in this model). 4) **Circular wait** Car 1 waits on car 2 which waits on car 3 which waits on car 4 which waits on car 1.

2. **6.2** The OS (or traffic lights) could coordinate access based on columns and rows, just like a real intersection: Car 1 and 3 drive until completion before the other cars can go.

3. **6.4** The graph gives four non-deadlock paths. Here is one: Process Q can get B while process P can get A. Then process P releases A before process Q gets A. Then process Q releases B before process P gets B.

4. **6.5**

   (a) **a** The available array is calculated by summing the columns and subtracting that from total resource availability.

   (b) **b** The need matrix is calculated by subtracting the cells in "Current allocation" from the corresponding cells in "Maximum demand."

   (c) **c** I'll represent the Available array as a Haskell-style list of tuples of resource ID and int.

      i. P1 executes. "Available" becomes: `[(A,4),(B,2),(C,3)(D,2)]`
      ii. P1 exits. "Available" becomes `[A,6],[B,4],(C,6),(D,5)`
      iii. P2 executes. "Available" becomes: `[A,3],[B,0],(C,2),(D,3)`
      iv. P2 exits. "Available" becomes: `[A,10],[B,5],(C,6),(D,7)`
      v. P3 executes. "Available" becomes: `[A,8],[B,2],(C,3),(D,6)`
      vi. P3 terminates. "Available" becomes: `[A,11],[B,5],(C,6),(D,8)`
      vii. P4 executes. "Available" becomes: `[A,7],[B,4],(C,4),(D,7)`
      viii. P4 terminates. "Available" becomes: `[A,12],[B,6],(C,6),(D,8)`
      ix. P5 executes. "Available" becomes: `[A,9],[B,2],(C,3),(D,5)`

x. P5 terminates. "Available" becomes: `[A,13],[B,6],(C,7),(D,9)`

xi. P0 executes and terminates

(d) **d** If we're assuming the request is *instead of* current request, then it's fine. When P5 terminates, it frees 4, 4, 4, 4, which is more than enough for the others.

If we assume the request is *in addition to* the current request, then it won't work if it goes in the above order. If we do P0 before P4 then it'll work.

5. **6.6**

   a. P0 will try to get C when P2 already has it. P1 will try to get B when P0 already has it. P2 will try to get D when P1 already has it. Circular deadlock.

   b. Move `get(C)` to the end in P2. P2 will hold for P1. P1 will wait for P0. P0 will not wait for anybody and will terminating, releasing B for P1, P1, then P1 will complete and release D for P2.

6. **6.11** Yes. 1, 2, 3.

   a. Yes, 1,2,3

   b. No.

7. **6.14**

   a. Yes.

foo()	**bar()**
semWait(S);	
semWait(R);	semWait(R);
x++;	
semSignal(S);	
semSignal(R);	semWait(S);
semWait(S);	

   b. Yes.

8. **6.15** `(2 1 6 5)`

9. **6.18**

   a. If a leftie and a rightie are next to each other (which they invariably will be if it's mixed), then the leftie or rightie will be waiting upon two forks, which prevents deadlock on either side of him.

   b. Isn't starvation just a specific case of general deadlock?