



What's New:

Some of the questions are more conceptual than before, and may take some thinking through.

1. Concepts

- a. Why is it useful to have hardware primitives to support synchronization?
Synchronization may have timing constraints and hardware is generally faster. The more general issue though is that purely software-based solutions are difficult to get right, especially for general cases.
- b. Why is it not desirable to have race conditions?
Corruption of data may occur as a result of race conditions. It is also much harder to reason things out about a program correctly if race conditions can happen. In general, a race-free program is more likely to have predictable outcomes.
- c. Explain why high-level primitives like monitors are useful.
They make problem solving less specific to a particular OS or system, and higher-level features are easier in general to understand and program than low-level primitives.

2. Low-Level Primitives

- a. Try to find an example of use of the Bakery Algorithm (pp 196-197) which convinces you that it works (or doesn't).
Any example which takes you through variations on timing would do: feel free to ask me if your example is good.
- b. Why is a spinlock (Figures 7.7-8, pp 198-199) not suitable as a general synchronization mechanism for a single-processor CPU?
When would you use a spinlock?
On a uniprocessor, not only is no useful work being done, but allowing the process holding the lock to proceed (and hence release the lock) is also prevented until after at least 1 context switch. It can be an option to use a spinlock if the critical section is really short and the probability of contention for the lock is low. Even so, even on a multiprocessor, a spinlock can have unacceptable overheads (for reasons that should be covered in an architecture course).
- c. Work through an example using both the pseudocode semaphore definition (p 201) and the "real" algorithm (p 203) with the initial count of semaphore $S = -2$, and 3 processes defined as follows, assuming the processes run in order P_0, P_1, P_2, P_3 , i.e., P_3 reaches its signal after the others have already executed their wait:

```
P0:
    wait (S)
    signal (S)
P1:
    wait (S)
    signal (S)
P2:
    wait (S)
    signal (S)
P3:
    signal (S)
```

- i. Is there any difference between what the two versions of the semaphore algorithm do in this example?
 *P_3 will continue, since you don't block on a signal, but none of the others can continue, even if P_3 did manage to get in first (contrary to the problem spec).
If you don't get it, put in the numbers for each step.*
- ii. For both versions of the semaphore algorithm, explain what a negative initial value for S signifies with this specific use of semaphores.
In this specific usage of semaphores – for the "real algorithm" – a negative value (here, -2) represents the fact that we want 2 other processes to wait until another has reached a specific point, assuming that the process P_3 will get there last. If this last condition is met, both of the other processes will stall until P_3 gets to its signal operation. Could you generalize the example, so it would work even if P_3 didn't get to its synchronization point after the others? For the pseudocode version of the algorithm, initializing a semaphore to a negative value means that the negative of that negative number of signals must be executed by processes which don't have to wait before anyone waiting can continue.
- iii. Are the two versions of the semaphore algorithm the same in general? If not, list the differences. If so, justify your answer.
If S starts positive, they are the same in general. The "real" version allows use of a negative initial value to stall a process until another wakes it up, without requiring that it increase to 0, which allows a wider range of uses of semaphores than the pseudocode version does.

3. Classic Problems

- a. Bounded buffer (algorithm p 207).
 - i. Show that if the buffer is full, a producer will be blocked, but a consumer will not be.
When the buffer is full, the empty semaphore will be 0 so the producer will be blocked. The consumer only waits on the full semaphore, which will be n , so it will not be blocked. Also, the mutex semaphore will not be blocked unless either the buffer isn't full or empty.
 - ii. Show that if the buffer is empty, a consumer will be blocked, but a producer will not be.
When the buffer is empty, the empty semaphore will be n so the producer will not be blocked. The consumer waits for the full semaphore, which will be 0, so it will be blocked.
 - iii. Show that if the buffer is neither full nor empty, neither the producer nor the consumer will be blocked unless the other is in its critical section (between `wait(mutex)` and `signal(mutex)`).
If the buffer is neither full nor empty, neither the "full" nor the "empty" semaphores will be 0 so neither will be blocked. If either is in its critical section, the mutex semaphore will block the other until the first process exits its critical section by

iv. In view of (i)–(iii), is the given bounded buffer generally correct?

Yes. The previous answers add up to this one: if this were an exam question, you would need to summarize the key findings and explain why they add up to a general proof of correctness.

b. Dining Philosophers.

i. The simplest approach (p 220) can easily lead to deadlock as described in the book. Show a sequence of operations which will lead to deadlock.

If all of the philosophers pick up their left fork and are interrupted before getting further, then they will all be waiting for their right fork, i.e., all of the processes will pass through the first wait on the semaphores, but will deadlock on the second wait. Can you think of any other obvious cases?

ii. Show that the solution using a monitor (p 219) can lead to starvation.

Let's abstract the sequence of actions in the monitor as:

```
P
{
  pickup ();
  putdown ();
}
```

A philosopher wants to eat, but the person on the left is already eating, and the person on the right is not. If the person on the right starts eating (i.e. calls `pickup`) and the person on the left stops eating (i.e. calls `putdown`), then the philosopher in the middle cannot eat. If the person on the left starts eating again, and the person on the right stops eating, the philosopher in the middle still cannot eat. This can be repeated indefinitely, leading to starvation. It would be useful to work through the example in the book to be sure that it actually does this.

4. Work through any problems you may have arising out of week 5 lectures.