Solution
CO 502 – Operating Systems Tutorial: Synchronization

Morris Sloman

1. What happens when a signal is received by a process

   The default action for most signals is to terminate the process, unless the process has installed a handler for that signal. Note that SIGKILL and SIGSTOP cannot be ignored/handled.

2. Explain why the following statement is false: When several threads access shared information in main memory, mutual exclusion must be enforced to prevent the production of indeterminate results.

   If all of the threads read, but do not modify, the shared information, then the threads may access the information concurrently without mutual exclusion.

3. Discuss the pros and cons of busy waiting.

   Busy waiting is an efficient solution when the waits are certain to be short. The alternative is a blocked wait. The advantage of busy waiting is that the thread has the processor and can resume immediately after the wait is finished. The disadvantage is that busy waits can be wasteful when the waits are long. Also, busy waiting is not wasteful if there are fewer threads than processors (i.e., no other productive work could be accomplished).

4. One requirement in the implementation of the semaphore operations up and down is that each of these operations must be executed atomically; i.e., once started, each operation runs to completion without interruption. Give an example of a simple situation in which, if these operations are not executed atomically, mutual exclusion may not be properly enforced.

   Assume that there are two threads, T1 and T2, and a semaphore, S. S currently has a value of 1. T1 calls down(S). Since S is currently 1, T1 is going to run the code to decrement S. At this point, a context switch allows T2 to execute. T2 calls down(S) as well. The down(S) command that T2 called completes, so S has a value of 0. Then T2 is in the middle of its critical section when another context switch occurs, and T1 begins to execute. T1 decrements S to a value of -1, then begins to execute T1's critical section code. At this point, both T1 and T2 are in their critical sections, which is a violation of the principle of mutual exclusion.

5. Can two threads in the same process synchronize using a kernel semaphore if the threads are implemented by the kernel? What if they are implemented in user space? Assume no threads in other processes have access to the semaphore. Discuss your answers.

   With kernel threads, a thread can block on a semaphore and the kernel can run some other thread in the same process. Consequently, there is no problem using semaphores. With user-level threads, when one thread blocks on a semaphore, the kernel thinks the entire process is blocked and does not run it ever again. Consequently, the process fails.

6. Give a sketch of how a uniprocessor operating system that can disable interrupts could implement semaphores.

   To do a semaphore operation, the operating system first disables interrupts. Then it reads the value of the semaphore. If it is doing a down and the semaphore is equal to zero, it puts the calling process on a list of blocked processes associated with the semaphore. If it is doing an up, it must check to see if any processes are blocked on the semaphore. If one or more processes are blocked, one of them is removed from the list of blocked processes and made runnable. When all these operations have been completed, interrupts can be enabled again.

7. What is the difference between a binary semaphore and a general semaphore?

A binary semaphore can only take the values 0 or 1. An up operation on a binary semaphore which has the value of 1, leaves it as 1. A general semaphore can take the values 0 to n, i.e. can represent a count of items or number of processes allowed into the critical region.

8. Describe a suitable data structure for a general semaphore and give a pseudocode outline for the following operations in terms of the operations down(s) and up(s) on a binary semaphore s.
   init (value, gs) – initialises the general semaphore gs to value
   gen_down (var gs) – the down operation on a general semaphore gs
   gen_up (var gs) – the up operation on a general semaphore gs

```
struct gensem { binary_semaphore mutex, bs; integer count;  };

void init (integer n, gensem gs )
{    gs.mutex = 1;
     gs.count = n;
     if(n==0) gs.bs=0;
     else gs.bs = 1;
}

void gen_down (gensem gs)
{    down(gs.bs); /* wait here if count == 0) */
     down(gs.mutex);
     gs.count = gs.count-1;  /* count ≠ 0 */
     if (gs.count > 0) up(gs.bs); /* allow next process in */
     up(gs.mutex);
}

void gen_up (gensem gs )
{    down(gs.mutex);
     gs.count = gs.count + 1;
     if (gs.count == 1)up(gs.bs);
          /* i.e. gs.count was 0, so allow a process past semaphore.
          Could be just up(gs.bs) without test as up on binary
          semaphore with value equal to 1 leaves it 1 */
     up(mutex);
}
```

9. Consider the following three threads:

```
T1:          T2:        T3:
a = 1;       b = 1;     a = 2;
b = 2;
```

   (a) Show all possible thread interleavings.

   (1) a=1; b=2; b=1; a=2; ⇒ (a=2, b=1)
   (2) a=1; b=2; a=2; b=1; ⇒ (a=2, b=1)
   (3) a=1; b=1; b=2; a=2 ⇒ (a=2, b=2)
   (4) a=1; a=2; b=2; b=1; ⇒ (a=2, b=1)
   (5) a=1; b=1; a=2; b=2; ⇒ (a=2, b=2)
   (6) a=1; a=2; b=1; b=2; ⇒ (a=2, b=2)
   (7) b=1; a=1; b=2; a=2; ⇒ (a=2, b=2)
   (8) a=2; a=1; b=2; b=1; ⇒ (a=1, b=1)
   (9) b=1; a=1; a=2; b=2; ⇒ (a=2, b=2)
   (10) a=2; a=1; b=1; b=2; ⇒ (a=1, b=2)
   (11) b=1; a=2; a=1; b=2; ⇒ (a=1, b=2)
   (12) a=2; b=1; a=1; b=2; ⇒ (a=1, b=2)

   (b) If all thread interleavings are as likely to occur, what is the probability to have a=1 and b=1 after all threads complete execution?

   1/12

(c) What about a=2 and b=2?

5/12

10. Synchronization within monitors uses condition variables and two special operations, wait and signal. A more general form of synchronization would be to have a single primitive, waituntil, that had an arbitrary Boolean predicate as parameter. Thus, one could say, for example,

```
waituntil     x < 0  or   y + z < n
```

The signal primitive would be no longer needed. This scheme is clearly more general than that of Hoare, but it is not used. Why not? (Hint: think about the implementation.)

It is very expensive to implement. Each time any variable that appears in a predicate on which some process is waiting changes, the run-time system must re-evaluate the predicate to see if the process can be unblocked. With the Hoare and Brinch Hansen monitors, processes can only be awakened on a signal *primitive.*

Mentimeter Solutions

1) How many different thread interleavings are there?
2) Probability of a =1 and b =1 after all threads complete
3) Probability of a =2 and b =2 after all threads complete
   see 9 above

4) Can 2 *kernel* threads synchronize using a kernel semaphore?
5) Can 2 *user level* threads synchronize using kernel semaphore?
   See 5. Above

6) Semaphore Question How many times will P0 print ‘0’?

   At least twice.
   P0 completes loop prints 0, and waits on S0,
   P1 runs and wakes P0 so P0 prints 0
   p2 runs and wakes P0 so P0 prints 0 ie 0, 0, 0

   However if P2 runs immediately after P1, then S0 will still be 1 so P0 runs 2nd time and only prints 0, 0

7) If no process is suspended in a monitor, the signal operation on a condition variable:
   nothing happens