C211 – Operating Systems

Tutorial: Synchronisation

– Answers –

Lecturer: Peter Pietzuch ⟨prp@doc.ic.ac.uk⟩

1. 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.

2. Discuss the pros and cons of busy waiting.

   Busy waiting is the 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).

3. 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, T_1 and T_2, and a semaphore, S. S currently has a value of 1. T_1 calls down(S). Since S is currently 1, T_1 is going to run the code to decrement S. At this point, a context switch allows T_2 to execute. T2 calls down(S) as well. The down(S) command that T_2 called completes, so S has a value of 0. Then T_2 is in the middle of its critical section when another context switch occurs, and T_1 begins to execute. T_1 decrements S to a value of -1, then begins to execute T_1's critical section code. At this point, both T_1 and T_2 are in their critical sections, which is a violation of the principle of mutual exclusion.

4. 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.

5. Does the strict alternation solution work the same way when process scheduling is preemptive?

   The strict alternation solution was designed with preemptive scheduling in mind. When scheduling is nonpreemptive, it might fail. Consider the case in which turn is initially 0 but process 1 runs first. It will just loop forever and never release the CPU.

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. 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

8. 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,

$$\texttt{waituntil}\quad x < 0 \quad \texttt{or} \quad y + z < n$$

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 Hansen monitors, processes can only be awakened on a `signal` *primitive.*