ECE254: Operating Systems and Systems Programming Spring 2015

Lecture 11 — Concurrency: Synchronization and Atomicity

Jeff Zarnett

Synchronization

If a computer only ever did exactly one thing at a time (one process, one thread) we would have no concurrency and therefore no concurrency (co-ordination) problems. We have seen already, however, that in a system with multiple processes or multiple threads we have concurrency: more than one thread or process is making progress. And if the system is multicore, we can have parallelism: more than one thread or process actually executing on a CPU in a given instant. Either or both of these can lead to various problems.

The author of an application does not decide when a thread runs and when a thread switch will occur; these are things the operating system will decide. And the operating system does not usually give much thought to whether it is a convenient or inconvenient time to run a given thread.

The common English usage of the word "synchronization" refers to making two or more things happen at exactly the same time, but what we mean when we talk about synchronization in the computer sense is more general: it means we want some relationship between events, and the relationship can be before, during, after [Dow08]. There are two definitions that we need to consider at this point.

The first is *serialization*: we want there to be some sort of order of events. What we would like is to be certain that Event A takes place before Event B. We have already examined some examples of this, where we have a thread or process that `waits` on another. The merge sort algorithm, if implemented with two threads that sort the sublists and one thread that merges the results, necessarily implies that the merge thread has to wait for the sorting threads to finish. Thus the sorting (event A) must take place before the merging (event B). Or we could phrase it that B must happen after A.

The other thing we often want is *mutual exclusion*: events C and D must not happen at the same time. This is something we've referenced a few times before now, often phrased as the fact that we need some sort of co-ordination. In the discussion of inter-process communication, we said we might have an area of memory that is shared between two processes. If two processes can write to this area, there is some possibility that they both try to access the same place at the same time. If we have mutual exclusion, we are certain that P_1 writing to the shared area (event C) does not happen at the same time as P_2 tries to read it (event D).

Serialization through Messages

Suppose we have two people, Alice and Bob (these are the standard names in computer examples...) who work at the Springfield Nuclear Power Plant in Sector 7G, though they work in separate offices and cannot easily see one another. Alice works the day shift and Bob works the night shift. Suppose also that due to safety regulations, that Alice cannot go home until Bob has arrived and begun working. This is a situation where we would like serialization: the event of Bob's arrival must take place before the event of Alice's departure.

How can we get such behaviour? Well, the simple solution is that Alice stays at her desk and will not leave until she gets a call from Bob. Bob doesn't call until he's arrived at the office. This is a very simple scenario, but message passing is a valid solution for a lot of synchronization problems and it's the sort of thing we do in real life all the time: "Text me when you get here" or "Call me when you've finished".

If we are certain that Bob arrives before Alice departs, we can say that these events are *sequential*, because we know the order of events. At some point during the workday, Alice ate lunch, and at some point before work, Bob ate lunch. But we have no idea who ate lunch first, so we say they ate lunch *concurrently*. The formal, strict definition of concurrent is: two events are concurrent if we cannot tell by looking at the program which will

happen first [Dow08].

In common parlance, when we say something happened concurrently, it means they happened at the same time (Alice and Bob both had lunch at 12:00). In this context, concurrency is not the same as saying that they happened at the exact same time; it's saying that we do not know (and cannot guarantee) an order of events. It's possible that Alice had lunch at 12:00 and Bob had lunch at 13:00 or they both ate at 12:00 or Alice had lunch at 13:30 and Bob had lunch at 12:15. We do not know. The order of concurrent events may change on two runs of the program; this is what we call nondeterminism.

In a non-deterministic program, we cannot tell just by looking at the program what the execution order would be. If we have two concurrent events, one in which the program prints "1" to the console and one in which the program prints "2" to the console, these could happen in any order. So we might see the output as "12" or "21". Which one will happen? Without some form of co-ordination, it could be either. This non-determinism makes it difficult to analyze a program. If "12" is the correct output but "21" occurs only very rarely, finding the cause and fixing it might be very painful [Dow08].

This is, incidentally, what is referred to as a "Heisenbug" (a portmanteau of Heisenberg and Bug). This has nothing to do with the "Breaking Bad" TV show, but with Werner Heisenberg (the physicist) and his scientific principle of uncertainty that says: if we precisely know the position of a particle we know nothing about its momentum and vice versa. The Heisenbug is frustrating because the harder we try to track it down, the less likely it is to occur.

Shared Data and Atomic Operations

We noted earlier that the need for co-ordination in inter-process communication arises from the fact that some area of memory is shared. We also know that all the threads of a process share the same data: in the merge sort algorithm the sorting and merging routines both operate on the same data array. Let's examine a simpler example (from [Dow08]) where a shared variable `x` is manipulated by two threads A and B, in pseudocode:

Thread A	**Thread B**
`A1. x = 5`	`B1. x = 7`
`A2. print x`	

This is non-deterministic code: there is no co-ordination mechanism here so we cannot say what order these statements will occur. Some possible outcomes are: 5 is printed out and is the final value; 7 is printed out and is the final value, or 5 is printed out and 7 is the final value. Note that there is no way we can print out 7 and get a final value of 5, because we can be certain that statement A1 executes before A2; the problem is we do not know where in relation to the two A-statements that statement B1 will execute.

This example uses more than one thread, but we do not even need multiple threads to have a concurrency problem; just having interrupts in the system is sufficient.

Consider an application that is used to count occurrences of some event (whatever it is). We will store the count in a variable `count` and we will provide some facility for the user to reset the counter (the reset button). Each time we detect the event, we increment `count` with statement of `count++;` which seems like one single statement.

Even a seemingly simple operation like `count++;` is broken down into a series of smaller operations. You've just detected an event. Let's assume the current value of the variable is 4. Thus we want to increment the variable, which will require three steps.

1. Read the current value of `count` (read 4)
2. Add 1 to the value (now it's 5)
3. Write the changed value back to memory (write 5)

Now imagine an interrupt comes at the worst possible time. The interrupt is generated by the reset button: it's supposed to set the value of count to zero.

1. Read the current value of `count` (read 4)
2. Add 1 to the value (now it's 5)
3. INTERRUPT (control goes to the interrupt handler)
4. Write 0 to the variable (write 0)
5. END INTERRUPT (control returns to where it was before the interrupt)
6. Write the changed value back to memory (write 5)

At the end of this execution sequence, the variable `count` shows 5, but it should show 0 (or 1), but instead it shows 5, which is certainly wrong. The user pressed the reset button but the count did not reset! If the reset interrupt had occurred before reading the variable, the count would have been reset, and then an event detected and the count goes up to 1. If the interrupt had occurred after writing the value 5 to the variable we would see the count set to 0 as is expected. If it occurred after the read but before the write, there is an error and the changes the interrupt handler made were lost.

This problem arises because the instruction `count++` is really three things (read, add, write) and can be interrupted at any time. When we are performing an operation that cannot be interrupted, we say it is *atomic*: indivisible. Although since about 1945, we have been able to split the atom (proving that atoms themselves are divisible and that humans are very good at finding ways to make things explode) the use of the word atomic in this context is not about nuclear physics but references the other meaning that stems from Greek word *atomos*, meaning indivisible (or more literally, not-cuttable).

Though there are usually some atomic operations available to us in a given system, we cannot be certain that all operations are indivisible. In fact, the opposite is likely to be true: we can be certain there are some operations that are non-atomic. Therefore we need to make sure at the very least that one operation on the shared variable is finished before the next begins.

A thought: can we do this with serialization? That is, can we make sure that the `count++` operation completes before the `count = 0` operation? That would eliminate the problem of the reset being ignored. Unlike the scenario where Alice waits for Bob to get to work before she leaves, in this case there is no obvious order between the events: the user may press the reset button at any time, even if no event has just occurred. Similarly, the event may be detected even if the user is nowhere around and not going to press the reset button. Our concept of serialization requires there is a correct order: first this, then that. Here, where both orders are valid, we need the other approach: mutual exclusion (also called *mutex*).

With mutual exclusion, we do not know or enforce any particular order of events, but what we do care about is that we do not have multiple threads trying to update the variable at the same time. It would mean that the action to reset `count` to zero would have to wait until the `count++` operation was completely finished (or vice versa) before it gets to execute.

Mutual Exclusion through Messages

We can also solve the problem of mutual exclusion via messages. Suppose there is a third employee at the power plant, Charlie, who works on the day shift at the same time as Alice. Safety rules say that at least one of them has to monitor the safety of the reactor at all times and therefore they cannot both take lunch at the same time. If we cannot predict when lunch begins or how long it will last, how can Alice and Charlie co-ordinate to make sure they don't take lunch at the same time?

References

[Dow08] Allen B. Downey. *The Little Book of Semaphores (2nd Edition)*. Green Tea Press, 2008.