Lecture 11 — Concurrency: Synchronization/Atomicity

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

August 21, 2016

Synchronization

If a computer only ever did exactly one thing at a time we would have no concurrency and therefore no concurrency (co-ordination) problems.

A system with multiple processes or multiple threads will have concurrency.

If the system is multicore, we can have parallelism.

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.

The operating system does not usually give much thought to whether it is a convenient or inconvenient time to run a given thread.

Synchronization

The common English usage of the word “synchronization” refers to making two or more things happen at exactly the same time.

Synchronization in the computer sense is more general:

Some relationship between events: before, during, after.

Serialization

Serialization: some sort of order of events.

Certainty that Event *A* takes place before Event *B*.

Example: a thread or process that `waits` on another.

Merge sort: the merge thread 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*.

Mutual Exclusion

Mutual exclusion: events C and D must not happen at the same time.

IPC: 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
They work at the Springfield Nuclear Power Plant in Sector 7G.

They work in separate offices and cannot easily see one another.

Alice works the day shift and Bob works the night shift.

Due to safety regulations, Alice cannot go home until Bob has arrived.

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.

Serialization through Messages

How can we get such behaviour?

Simple solution: Alice won't leave until she gets a call from Bob.

Bob doesn't call until he's arrived at the office.

This is the sort of thing we do in real life often: "Text me when you get here".

Sequential and Concurrent Events

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.

Concurrent Events

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).

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.

Non-Deterministic Programs

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.

If “12” is the correct output but “21” occurs only very rarely, finding the cause and fixing it might be very painful.

Heisenbugs

This is, what is referred to as a “Heisenbug”.

This has nothing to do with the “Breaking Bad” TV show.

It comes from Werner Heisenberg and his scientific principle of uncertainty.

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:.

Simple example of a shared variable `x` is manipulated by two threads *A* and *B*:

Thread A

```
A1. x = 5
A2. print x
```

Thread B

```
B1. x = 7
```

Analyzing the *A* and *B* Example

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.

Interrupts

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.

We will store the count in a variable `count`.

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.

`count++;` is broken down into a series of smaller operations.

If `count` is 4 and we increment it:

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)

The variable `count` shows 5, but it should show 0 (or 1).

The user pressed the reset button but the count did not reset!

If the reset interrupt had occurred before reading the variable $\rightarrow$ 1.

If the reset interrupt had occurred after writing the variable $\rightarrow$ 0.

The reset action is “lost”.

Atomic Operation

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.

The word stems from Greek word *atomos*, meaning indivisible.

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!

Atomic Operation

Can we get atomic operations with serialization?

That is, can we make sure that the `count++` operation completes before the `count = 0` operation?

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 that there exists a correct order:
First this, then that.

Mutual Exclusion

Here, where both orders are valid, we need the other approach: mutual exclusion (also called mutex).

We do not know or enforce any particular order of event.

But: we don't have $>$ 1 threads trying to update a variable at the same time.

The action to reset `count` to zero would have to wait until the `count++` operation was completely finished (or vice versa).

Mutual Exclusions through Flags

So we have identified shared data as a potential source of error.

A section of code that should be accessed by a maximum of one thread at a time is referred to as a critical section.

The purpose of mutual exclusion is to ensure that at most one thread is in the critical section at a time.

If we ever have more than one thread in the critical section at a time, something has gone terribly wrong.

But on the other hand, the critical section is supposed to do something useful, so we cannot solve the problem by not allowing any thread to access it ever.

Critical Sections

It is the responsibility of the programmer to identify what critical sections, if any, exist in the program, and to protect them with mutexes.

Some analysis tools may exist to identify shared data, but ultimately the best analysis tool is taking a careful look.

Critical sections should be as short as possible (but must enclose all shared data accesses).

The critical section is something that cannot be run in parallel, so it increases the magnitude of the *S* term in Amdahl's Law.

Mutual Exclusion with Flags

Strategy: have a variable to indicate if the critical section is currently in use.

Thread A

```
A1. while ( turn != 0 ) {
A2.     /* Wait for my turn */
A3. }
A4. /* critical section */
A5. turn = 1;
```

Thread B

```
B1. while ( turn != 1 ) {
B2.   /* Wait for my turn */
B3. }
B4. /* critical section */
B5. turn = 0;
```

Strict alternation!

Strict Alternation

First it is the turn of thread *A*, then the turn of thread *B*, then back to *A*, etc.

What if thread *A* is to run more often than *B*?

If thread *B* terminates then thread *A* will be stuck forever because the variable `turn` will always be saying that it is thread *B*'s turn.

This solution is obviously not satisfactory.

A Second Approach

Thread A

```
A1. while ( busy == true ) {
A2.     /* Wait for my turn */
A3. }
A4. busy = true;
A5. /* critical section */
A6. busy = false;
```

Thread B

```
B1. while ( busy == true ) {
B2.     /* Wait for my turn */
B3. }
B4. busy = true;
B5. /* critical section */
B6. busy = false;
```

A boolean flag.

Boolean Flag

The statement `while (busy == true)` is followed by `busy = true;`

These are two distinct steps: read of `busy` and write of `busy`.

A process switch could occur between the read and the write.

Then threads *A* and *B* will both be in the critical section at the same time!

This solution is also not satisfactory.

Third Time's the Charm

Thread A

```
A1. flag[0] = true;
A2. while ( flag[1] ) {
A3.     /* Wait for my turn */
A4. }
A5. /* critical section */
A6. flag[0] = false;
```

Thread B

```
B1. flag[1] = true;
B2. while ( flag[0] ) {
B3.     /* Wait for my turn */
B4. }
B5. /* critical section */
B6. flag[1] = false;
```

An array of flags, then.

Array of Flags

Once again, this strategy is defeated by an untimely process switch.

If statement A1 sets `flag[0]` to true and there is a switch to thread *B*, setting `flag[1]` to true, now both processes are stuck.

Neither can advance, because each is waiting for the other.

This is, perhaps, slightly better than two threads in the critical section, but we have two threads that are permanently stuck now.

This is also not satisfactory.

Disabling Interrupts

The attempts at solution we have attempted so far have all been foiled by an untimely process switch, which will be triggered by an interrupt.

This presents a possible solution: disabling interrupts.

Disabling interrupts is a crude solution.

While interrupts are disabled, the system will be unable to respond to user input or other events (e.g., a fire alarm or detection of an incoming missile!).

If an error is encountered in the critical section and the program is terminated, the system is effectively stuck because no other program will be able to run.

Disabling Interrupts

If we have multiple processors the disabling interrupts will not be sufficient.

But maybe we're on the right track by getting hardware involved.

The hardware designers were aware of the problem and have kindly provided a facility to help us out: the Test-and-Set instruction.

Test-and-Set

The Test-and-Set instruction is a special machine instruction that is performed in a single instruction cycle and is therefore not interruptible.

It is therefore an atomic read and write.

Its semantics in C are:

```
boolean testAndSet( int i ) {
  if ( i == 0 ) {
    i = 1;
    return true;
  } else {
    return false;
  }
}
```

Test-and-Set

Now let us put it into action.

```
while ( !testAndSet( busy ) ) {
   /* Wait for my turn */
}
/* critical section */
busy = 0;
```

(Both *A* and *B* would be the same, so only one copy is shown)

Mutual Exclusion Success!

Finally, we have something that will provide mutual exclusion without the risk that the threads will all get stuck because each thinks another is in the critical section.

This is good, but can be improved.

The `while` loop that is constantly checking the value with the Test-and-Set instruction is an example of busy-waiting.

A given thread is constantly checking and checking and checking the instruction, and this is a waste of time and effort.

Busy-Waiting

It is less wasteful of resources and effort if the `while` loop contains some instructions saying it should wait a little while before checking again.

Serialization was achievable through messages; and they can be used to get mutual exclusion. This is the topic to be examined next.