Thread Safety

ICS432

Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Disclaimer

- The first 15 slides are a review of ICS 332 (Operating Systems) content
- So I'll go pretty fast, **but**, if any of this isn't clear or has been forgotten, it's important to stop me and ask for clarifications

The Trouble with Threads

- Threads share the same address space
- If no two threads update content at the same address in memory, we're fine
  - They would be just like processes, but for the fact that they share the code and can read the same memory
- But, it's very useful to have them "cooperate" by updating content at the same addresses
  - e.g., two threads that cooperate to compute the value of a single variable to compute it faster
  - e.g, two threads that update an index in an array to "the next image that should be analyzed"
- This is where problems can arise

Textbook Example

```
int x;  // global variable shared by
        // all threads

// thread #1's code
...
x++
...

// thread #2's code
...
x--
...
```

Textbook Example

- The previous code is compiled into lower level code
  - Assembly code in the case of C code
  - Byte code in the case of Java code
- Let's write the code in RISC-like x86 assembly, and assume a single core

Textbook Example

- The previous code is compiled into lower level code
  - Assembly code in the case of C code
  - Byte code in the case of Java code
- Let's write the code in RISC-like x86 assembly, and assume a single core

```
// Thread #1          // Thread #2

mov  eax,[@]          mov  eax,[@]
inc  eax              dec  eax
mov [@], eax          mov [@], eax
```

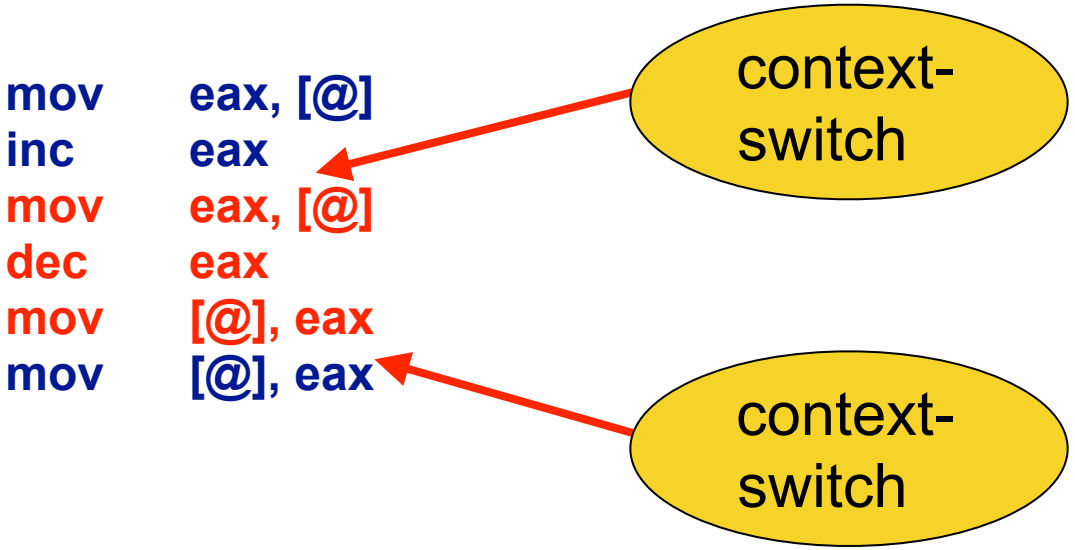
Load value from RAM into a register

Decrement the value in the register

Store the value from the register into RAM

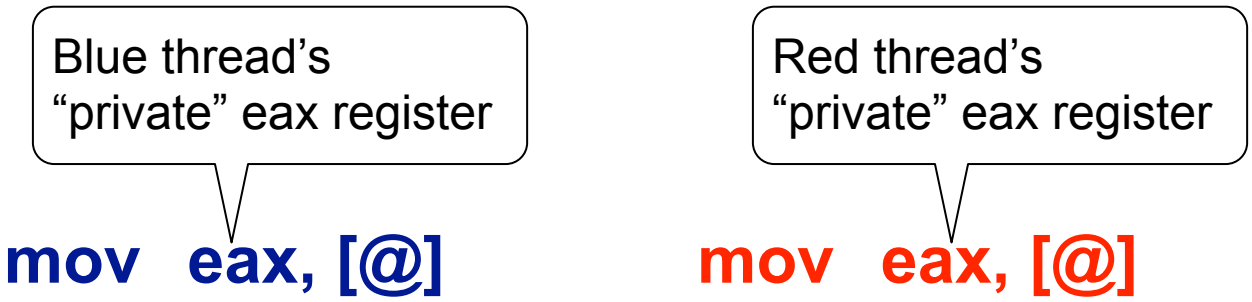
Textbook Example

- Illusion of concurrency: the OS context-switches threads rapidly
- We have many possible execution interleavings
- Here is one:



Same Register?

- In the previous slides the code shows that both threads use the same register (eax)
- As you recall from your OS course, at each context-switch register values for a thread/process are written back to RAM (to the PCB) and loaded back from RAM (from the PCB)
- So both threads can reference the same register in their code (in fact it's the SAME code), but through the magic of context-switching they have the illusion of having their own set of private registers
- We show this difference with the blue and red colors



Textbook Example

- Illusion of concurrency: the OS context-switches threads rapidly
- We have two 3-instructions sequences
- Discrete math: we have *20 possible instruction interleaving*
- Here are 3 possible execution paths:

```
mov   eax, [@]
inc   eax
mov   eax, [@]
dec   eax
mov   [@], eax
mov   [@], eax
```

```
mov   eax, [@]
inc   eax
mov   eax, [@]
dec   eax
mov   [@], eax
mov   [@], eax
```

```
mov   eax, [@]
mov   eax, [@]
dec   eax
inc   eax
mov   [@], eax
mov   [@], eax
```

Textbook Example

Let's assume that initially [@] = 5

```
load    eax, [@]    // eax = 5
inc     eax         // eax = 6
load    eax, [@]    // eax = 5
dec     eax         // eax = 4
store   [@], eax    // [@] = 4
store   [@], eax    // [@] = 6
```

```
load    eax, [@]    // eax = 5
load    eax, [@]    // Jax = 5
dec     eax         // eax = 4
inc     eax         // eax = 6
store   [@], eax    // [@] = 4
store   [@], eax    // [@] = 6
```

```
load    eax, [@]    // eax = 5
inc     eax         // eax = 6
load    eax, [@]    // eax = 5
dec     eax         // eax = 4
store   [@], eax    // [@] = 6
store   [@], eax    // [@] = 4
```

We would expect [@] to be 5 at the end
But in these executions we get 4 or 6!!

Race Condition

- The bug is called: **a race condition**
  - Its manifestation in this case is called: a lost update
- The race condition has a non-zero probability of manifesting
  - It may not manifest for 10,000 runs, and then do so for the 10,001st run
- It happens because the ++/-- operation is **not atomic**
  - It has to be done in multiple (hardware) steps, which can then be interleaved with other steps from other threads
  - Unlike, say, “set x to 2”, which is atomic because done in “one hardware instruction”
- Most of what we think of as an “operation” in a high-level language is not atomic when translated to assembly /bytecode
  - e.g., adding an element to a linked list
- There is a **`RaceCondition.java`** program on the Web site
  - Look at it to see race conditions in action

Program Correctness

- Race conditions mean that multiple executions of the same program may lead to different outcomes
- But in some cases this may be ok

```
int x = 0;
```

```
while(1) { x++;}
```

```
while(1) { print(x); }
```

- In the above we don't know what sequence of numbers will be printed, but perhaps that's ok
  - It all depends on what we mean by "correct"
  - Perhaps non-determinism is ok, or even a feature
  - After all, you can also generate (pseudo)random numbers!
- If the output is non-deterministic and non-desirable, then we say it's a race condition that we should fix because it's a bug

Race Condition: Be Picky

- There is a Race Condition if there is at least one execution path that leads to an incorrect outcome
- It doesn't matter how unlikely that execution path is
  - We'll see some pretty unlikely ones
- If its probability is >0, the program is incorrect and needs to be fixed
- In this course we'll often look at code and then wonder: is there a race condition?
- We do this by playing an adversary that makes the worst possible thread execution interleaving that will break the program
  - You pretend you're an evil OS that will insert the most inopportune context-switches to break concurrent code
- This is why we have programming assignments AND "pencil-and-paper" assignments

What about true concurrency?

- So far we've talked about context-switching in the context of race conditions, i.e., false concurrency
- Race conditions also happen with true concurrency when each thread is on its own core
- For instance, for the lost update example:
  - Each thread grabs the original value, apply its update on it, and then writes the result to RAM
  - Whichever thread writes the result last "wins"
- This is why it's called a race condition: the result depends on which threads gets to do the operation last, which is "randomly" determined by thread scheduling in the O/S, assignment of threads to cores, etc.
- Typically, I'll always assume a single core and talk about context-switching, but everything holds true with true concurrency
- A race condition may be more/less likely with true concurrency, but it remains a race condition nonetheless

Why we don't like Race Conditions

- We know that bugs can be difficult to identify
- Bugs that happen non-deterministically (perhaps very rarely) are close to impossible to identify
  - Often one needs to change the system to observe the bug's manifestation (e.g., the probability of manifestation could be higher with true concurrency than with false concurrency, the probability of manifestation is higher with one O/S than with another, could be minuscule if we add print statements, could depend on the compiler version)
- Therefore, one must learn how to write code without race conditions because debugging them after the fact is really difficult
- Hence the need for so-called "thread safety"

Thread Safety

- You may have heard the term “thread-safe” before, applied to functions/methods/libraries
- A method is thread-safe if it can be active (has been called but hasn’t returned yet) for two or more threads at the same time and guarantees that no race condition will ever occur in that method
  - i.e., two or more threads can have an activation record for the method in their stack
  - i.e., two or more threads have called it and are still “in it”
- If a method is **not** thread-safe, then the programmer must be very careful when using threads
  - e.g., `public void increment() {this.value++;}`
- If the documentation doesn’t say what is thread-safe and what isn’t, then the documentation is poor
  - Sadly, very common (or not sufficiently clear)

Thread-Safe Methods/Functions

- In these lecture notes we assume that we can make methods/functions thread-safe
  - We will learn how to do that in the next modules
- But for now, imagine you have a programming language that has some **`threadsafe`** keyword when declaring methods
  - e.g., `public threadsafe void increment() {this.value++;}`
- We will see how it's actually done in various languages
  - Java is "close" to the above syntax actually

Thread-Safe Methods

- Say you use a library that provides a Counter class with two thread-safe methods: Counter.increment() and Counter.decrement()
- Thanks to thread safety we can have any number of threads call these methods and there will be no race conditions

```
Counter counter = new Counter(0);
```

```
// Thread #1
…
counter.increment()
…
counter.increment()
…
counter.increment()
…
```

```
// Thread #2
…
counter.decrement()
…
counter.increment()
…
counter.decrement()
```

```
// Thread #3
…
counter.decrement()
…
counter.decrement()
…
```

- We're guaranteed that the **final** counter value will be 0

Thread-Safe Methods?

- So, it would seem that one just needs to use thread-safe methods and we're good, right?
- Unfortunately, things are not so peachy
- **Problem #1:** Many methods out there are not thread-safe
  - Sometimes because developers haven't gotten around to making them thread-safe
  - Sometimes because they chose to not make them thread-safe
  - We will see why this is a reasonable choice in a few slides
- **Problem #2:** Even if you use only thread-safe methods, you can still have race conditions!
  - This seems counter-intuitive, but in fact it's pretty obvious
  - Let's see this on an example…

Thread-Safety Thwarted

- Say we have a DataBase ADT with 4 **thread-safe** API functions:
  - int threadsafe **read_record**(int r) // reads record at index r
    // reading a non-existing record is a bug
  - void threadsafe **write_record**(int val) // writes/appends a new record
  - void threadsafe **remove_last_record**() // removes the last record
  - int threadsafe **get_length**() // returns # of records
- We have three threads:

```
// Writer
…
a.write_record(stuff)
…
```

```
// Remover
…
if (a.get_length() > 0) {
   a.remove_last_record()
}
…
```

```
// Reader
…
length = a.get_length()
if (length > 0) {
   a.read_record(length-1)
}
…
```

- Can you see the race condition? (which causes a read of a non-existing record)

Thread-Safety Thwarted

```
// Writer
…
a.write_record(stuff)
…
```

```
// Remover
…
if (a.get_length() > 0) {
   a.remove_last_record()
}
…
```

```
// Reader
…
length = a.get_length();
if (length > 0) {
   a.read_record(length-1)
}
…
```

- Database is empty
- Writer puts in a record
- Reader calls a.get_length() and gets return value 1
- Reader gets into the if since length > 0
- Reader is about to call a.get_record(0), but is context-switched out!
- Remover removes the last record, making the database empty
- Reader is context-switched back in, and calls a.get_record(0) on an empty database, which is a bug!

Thread-Safety Thwarted

```
// Writer
…
a.write_re
```

```
// Remover
```

"checking length" followed by "reading" is not atomic

These are calls to thread-safe ("atomic") methods
But a sequence of two atomic operations is not atomic!

- _length() and gets return value 1
- Reader gets into the if since length > 0
- Reader is about to call a.get_record(0), but is context-switched out!
- Remover removes the last record, making the database empty
- Reader is context-switched back in, and calls a.get_record(0) on an empty database, which is a bug!

You can't Escape Concurrency

- **Because the world has become multi-threaded, even when your program doesn't use threads explicitly, you can have race conditions!**
- A great example of this is with Java GUIs
  - And in fact other GUI systems as well
- JavaFX grew out of Java Swing, which itself grew out of java.awt
- The nice thing about java.awt was that it was thread-safe!
  - The awt developers did a bunch of work to avoid race conditions, so that the awt users don't have to
- But thread-safety reduces performance
  - We'll understand why that is in future lectures
  - Essentially: even if **you** know that there is no risk of race condition, the library doesn't and has to assume the worst

JavaFX is not Thread-Safe

- Most methods in JavaFX are not thread-safe!!
- At first glance this seems terrible:
  - Your JavaFX code is now susceptible to race conditions!
  - The JavaFX developer needs to know about concurrency!
    - But then, all developers should know about concurrency nowadays…
- But, because JavaFX is not thread-safe, it is more efficient than awt
- Let's see how this all works…

Threads in the JVM

- The JVM has many *daemon threads* (e.g., the Garbage Collector)
- We've talked about the JavaFX Application Thread, which:
  - Catches and dispatches GUI events
    - e.g., detects a mouse click and figures out that it's on a particular swing component
  - Executes "paint" operations of GUI components
    - e.g., to redraw something
- JavaFX was designed so that it is not thread-safe
- Therefore we have a problem:
  - The JavaFX Application Thread manipulates the states of JavaFX components
  - Any user thread (including your main thread) can manipulate the states of JavaFX components, and in fact you need that for most useful GUIs
  - But then, you're open to race conditions!
  - Let's check if this happens…

Bad JavaFX Program

- Let's look at BadJavaFXProgram.java on the course Web site….
  - It simply creates a displayable list of strings, and then starts a thread that adds/removes from that list of strings
- Let's run the program…
  - From my IDE
  - Due to a known JVM issues, the program will not stop due to exceptions (but we should be able to see them in the terminal output!)

What is Going On?

- We get several exceptions, and in particular this one
- **`java.lang.IllegalStateException: Not on FX application thread`**
- This is sort of informative: one of your threads (we only have one!) is doing things that some other thread should be doing
  - That other thread is the JavaFX application thread

How do we fix it?

- What we need, is a mechanism to say “please JavaFX Application Thread, do this thing for me”
  - We don’t have access to the code of the JavaFX Application Thread, but we need it to run our code
- Platform.runLater(Runnable)
  - We’ve talked about this for unfreezing an application
- But we can also use it to force some code to be executed by the JavaFX Application Thread
  - Not right now, but as soon as it gets to it!
  - Recall that the JavaFX Application Thread essentially maintains a list of Runnable objects on which it will call the run() method in sequence
- Some of the Platform.runLater() calls in the starter code of ics432imgapp are about thread safety
  - Sometimes just to avoid the “this should run on the JavaFX Application Thread” exception

First Fix Attempt

- Let's put each call to add()/remove() inside a Runnable and run the program again…
- What do we see?

First Fix Attempt

- Let's put each call to add()/remove() inside a Runnable and run the program again…
- What's do we see???
- `java.lang.IndexOutOfBoundsException: Index 1 out of bounds for length 0`
- We are getting race conditions!
  - By the time a Runnable runs, an item could already have been removed by another Runnable
    - Since removing an element is not instant, we can't just look at what's in the list right now to pick an element in it!
- So, this doesn't work because our code could issue, for instance: remove(0) followed by remove(0)

Second Fix Attempt

- Let's put the whole loop body inside a thread and run the program again…
- What do we see?

Second Fix Attempt

- Let's put the whole loop body inside a thread and run the program again…
- What do we see?

- Nothing moves!
- This is because our code HAMMERS the JavaFX Application thread with Runnable objects
  - This is a common "bad practice"
- Let's do another bad practice: add a sleep!
- We have a pretty terrible program, and there are still possibilities for race conditions!
  - The probability is 0.000…01, but still

On using Platform.runLater()

- Using `Platform.runLater()` in this program doesn't fix it
- One typically limits its use to a one-shot activity
  - e.g., click on the "Remove this file" button will spawn off a "Remove this file" Runnable
  - e.g., a click on this button disables that other button
  - See Homework #2 :)
- In the end, our example is just too brutal (and useless)
- A way to fix it would be to somehow "remember" previous deletions and additions in a separate data structure
  - We just cannot "look" at the item list to know what's in the list
  - This is because the JavaFX Application Thread is about to modify it based on our previous requests
- But in a reasonable program, hopefully we can do the right thing without too much trouble

JavaFX Philosophy Summary

- The JavaFX philosophy in twofold:
  - Methods that update GUI elements are not thread-safe
  - But we have mechanisms to ensure they're always called by the same thread
- This philosophy works for JavaFX but is not universal: often we need to create thread-safe methods

Conclusion

- A thread-safe method is one that can be called by multiple threads simultaneously without race conditions
- When using third-party software, you must find out which methods are thread-safe and which are not
  - When writing libraries to be used by others, you must document which methods are thread-safe
- Even if you only call thread-safe methods, you can still have thread-safety issues
- Even if you don't use threads, you use libraries/ runtimes that use them
  - e.g., when building GUIs with JavaFX

- Next up: How we make methods thread-safe