Threads in Other Languages

ICS432

Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Goals

- Most languages provide ways to do threads, and there are many “thread” libraries, that looks very much like what we’ve seen in Java and C
- But there are some languages that do things a bit differently, in good and bad ways
- The objective here is to just go through a few interesting things for:
  - Python
  - Javascript
  - Rust

Python

- As you recall from your Operating Systems course, there are two kind of threads:
  - User-level threads (also called “green” threads): purely in user space, the kernel doesn’t know about them, and they thus cannot run on different cores
  - Kernel-level threads: known to the kernel, that can run on different cores
- With green threads, there is no way to achieve higher performance for compute-bound operations by using multiple cores
  - And it’s even iffy for interactivity if there is one CPU-bound thread
  - Basically, threads have to willingly give up the CPU
  - This is the case when all threads do I/O, waiting, sleeping

Python using `threading`

- The Python interpreter was designed a long time ago, and it is well known for having a Global Interpreter Lock (GIL): only one thread can do “python things” at a time
- A common mistake of developers is to use “threads” in Python, and expect to achieve a parallel speedup for compute-bound operations!
- Python provides a **`threading`** package that does green threads
  - There is also a **`thread`** package, which behaves the same
- I hear “I use threads in Python and I get no speedup, what’s going on??” Is super common
- Let’s look and run a Python program that uses the **`threading`** package, and thus looks parallel
  - Web site: python_green_threads_example.py

Python using `threading`

- Green threads can share memory
- Let's look at a program that demonstrates that memory is shared
  - Web site: python_green_threads_example_sharing.py
- Big question about this program: is it thread-safe??
  - At first glance it "shouldn't" since multiple threads update a data structure concurrently
- It is thread-safe because these are green threads! There is no context-switching unless a thread sleeps/blocks/ends
- So we don't get multi-core performance, but at least we don't get race conditions!

Python `multiprocessing`

- Turns out, Python has another package called **`multiprocessing`**, which avoids the "green threads"problem
- It uses processes and makes them look like threads!
- Let's look at a program very similar to our first multi-threaded program
  - Web site: python_processes_example.py

Python ThreadPool

- The `multiprocessing` package provides a convenient “pool” abstraction
  - Which really should be named “process pool”
  - But there is a (green) threadpool version
- It’s really convenient and allows to write very short programs
- Let’s just look at a simple example that applies a function in parallel to elements of an array:
  - Website: python_processes_pool_example.py

Python `multiprocessing`

- So, that's great, but these aren't threads, they are processes, so they don't share memory
- Let's confirm this by looking at that program:
  - Web site: python_processes_example_no_sharing.py
- What if we want both:
  - Multi-core speedups for compute-bound computations
  - Shared memory

Python `multiprocessing`

- The **`multiprocessing`** module makes it possible to use "shared memory segments" (mentioned earlier this semester, and likely in any Operating Systems course)
- Python makes it looks relatively nice (I guess):
  - Web site: python_processes_example_sharing.py
- Note that in this example processes write to different elements in a "sharable list"
- But if they need to update the same elements, then there could be race conditions, because a sharable list is not *process-safe!*
- We then have to use locks
  - That the **`multiprocessing`** module provides
- There are other data structures in the module, like a queue, that are process-safe
- So just like in any language, we have to know which provided data structures are safe, and which aren't…

Python and Concurrency

- The amount of confusion and wrong information regarding concurrency in Python is astounding
  - I found many, many online “tutorials” or “examples” that have plain wrong statements
- But if you know the basics it’s really simple:
  - Python does not support standard kernel threads, due to the GIL
  - If using green threads there is shared memory and mostly thread-safety (you can cause race conditions if you really want), but no multi-core speedup
  - If using non-green threads, then they are really processes, and you can have shared memory segments, and then you can watch out for race conditions

Wait! Python 3.13

- The very recent Python 3.13 (released October 7 2024) has something exciting
- https://docs.python.org/3/whatsnew/3.13.html
  - “CPython now has experimental support for running in a free-threaded mode, with the global interpreter lock (GIL) disabled”
- Let’s look at: https://peps.python.org/pep-0703/
  - Let’s search for “multiprocessing” in that page
- People have been complaining about the GIL for years, and turns out a big motivation for fixing it now is AI and GPUs!
  - After all, if Python want to remain a “language for Data Science and AI”, it needed to fix this

Rust

- Rust came out of Mozilla, and has been adopted by many big tech companies as a “safe and concurrent” option
  - Now an official language for Linux kernel development
- No Invalid pointers/references
  - Validity is checked at compile time
- No memory leaks
  - But, unlike Java, it doesn’t use a garbage collector, and unlike C++, it doesn’t use reference counting!!
  - All checked at compile time
- No data races
  - e.g., the “lost update” bug
  - Notion of mutability / immutability of data and of data owner
    - (basically, a mutable reference can have a single owner)

Message Passing

- One “safety first” philosophy is that threads should not communicate by sharing memory but instead via message passing
  - From the Golang documentation: “Do not communicate by sharing memory; instead, share memory by communicating”
- The rationale is that concurrency and shared memory is too difficult and leads to too many bugs
  - Especially when developers get “creative”
- Often the goal is just to communicate, so let’s just have send() and recv() operations on communication channels
  - Of course that’s what we do routinely for distributed-memory computing (see ICS632)
- Let’s see how Rust does message passing

Message Passing in Rust

- Rust channel:
  - An abstraction through which one or more threads can send a message to one receiver thread

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        println!("Sending: {}", val);
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

Message Passing in Rust

- Rust channel:
  - An abstraction through which one or more threads can send a message to one receiver thread

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        println!("Sending: {}", val);
        tx.send(val).unwrap();
        println!("Sent: {}", val);
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

Adding this line causes a compilation error, because after sending a value one is no longer its owner! Safety first!

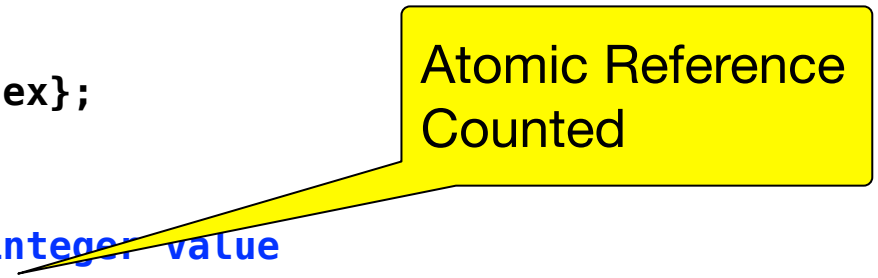
Message Passing in Java?

- Java does not support message passing between threads natively
- But of course it's very easy to emulate
  - Use a BlockingQueue of whatever Objects
  - Senders put "messages" into the queue
  - A receiver gets "messages" into the queue
- This is basically Producer-Consumer
  - And a Rust channel is basically a N-producers-1-consumer message buffer
- But Rusts adds all kinds of safety to this (e.g., the ownership feature in the previous slide)

Sharing State in Rust

- Doing everything with message passing is not always easy, so Rust makes it possible to share state (i.e., RAM) between threads
- It provides the notion of “value protected by a mutex”
  - i.e., you “lock memory” instead of “locking code”
- There are many details here, but let’s just look at a standard Rust example...

Sharing state in Rust




```
use std::sync::{Arc, Mutex};
use std::thread;
fn main() {
    // create a shared integer value
    let shared_state = Arc::new(Mutex::new(0));
    // create 16 threads that update the value
    let mut threads = vec![];
    for _ in 0..16 {
        // create an atomic copy of the shared state
        let shared_state = Arc::clone(&shared_state);
        let child_thread = thread::spawn(move || {
            let mut num = shared_state.lock().unwrap();
            *num += 1;
        });
        threads.push(child_thread);
    }
    // wait for all threads to complete
    for child_thread in threads {
        child_thread.join().unwrap();
    }
    println!("Result: {}", *shared_state.lock().unwrap());
}
```

Sharing State in Java?

- In Java, one could opt to never use locks/ synchronized but only use Atomics
  - e.g., java.util.concurrent.atomic.AtomicInteger
- But in Rust, that's the only option, which is safer

- What about condition variables in Rust?

Rust Condition Variables

```
use std::sync::{Arc, Mutex, Condvar};
use std::thread;

let pair = Arc::new((Mutex::new(false), Condvar::new()));
let pair2 = Arc::clone(&pair);

// Spawn a new thread
thread::spawn(move || {
    let (lock, cvar) = &*pair2;
    // Get the lock
    let mut started = lock.lock().unwrap();
    *started = true;
    // Notify the condvar
    cvar.notify_one();
});

// Wait for the thread to signal that it has started
let (lock, cvar) = &*pair;
let mut started = lock.lock().unwrap();
while !*started { // Typical while loop
    started = cvar.wait(started).unwrap();
}
```

Rust Takeaway

- **The underlying concepts/mechanisms are the same as what we've talked about all semester**
- But because of the pitfalls/difficulties of concurrency, Rust tries to constrain what users can do
  - Or at least they have to really make it clear they're doing something dangerous
  - There is an **`unsafe`** keyword in Rust!
- People vastly disagree on whether this is a good idea of course
- The good news: once you know all the concepts, the rest is just development details/constraints

Javascript?

- Javascript was never designed to support kernel threads
  - The well-known async/wait stuff in Javascript is often implemented using user-level “green” threads
- So there is no multi-core speedup for concurrency between compute-bound activities in Javascript in the browser
  - Typically, we don’t care as long compute-bound stuff is sent to the backend and the frontend just does async/wait
- But browsers run on multi-core machines, and so perhaps we want multi-core speedup in the browser
- That’s when you can use Web Workers
  - They come with all kinds of constraints/gotchas, but they work
  - I thought of doing our image processing app as a Web app and have you write Web Workers, but then decided against it because it was just too odd/difficult (but interesting!)… perhaps one day?
  - If time, we can look at some code...

Conclusion

- In my personal experience, if you don't know the basic concepts, it can be very difficult to understand how higher-level abstractions and/or language constructs work
- The amount of confusion and misinformation out there is pretty stunning