Channels

Ryan Eberhardt and Armin Namavari
May 14, 2020
Logistics

- Congrats on making it through week 6!
- Week 5 exercises due Saturday
- Project 1 due Tuesday
- Let us know if you have questions! We have OH after class
Reconsidering multithreading
Characteristics of multithreading

- Why do we like multithreading?
  - It’s fast (lower context switching overhead than multiprocessing)
  - It’s easy (sharing data is straightforward when you share memory)
- Why do we not like multithreading?
  - It’s easy to mess up: data races
Radical proposition

- What if we didn’t share memory?
  - Could we come up with a way to do multithreading that is just as fast and just as easy?
- If threads don’t share memory, how are they supposed to work together when data is involved?
- Golang concurrency slogan: “Do not communicate by sharing memory; instead, share memory by communicating.” (Effective Go)
- Message passing: Independent threads/processes collaborate by exchanging messages with each other
  - Can’t have data races because there is no shared memory
Theoretical model introduced in 1978: sequential processes communicate via by sending messages over “channels”
- Sequential processes: easy peasy
- No shared state -> no data races!

Serves as the basis for newer systems languages such as Go and Erlang

Also served as an early model for Rust!
- Channels used to be the only communication/synchronization primitive

Channels are available in other languages as well (e.g. Boost includes an implementation for C++)
Channels: like semaphores
Semaphores

Mutex: Unlocked
Buffer:

SomeStruct {
    ...
}

thread1
Semaphores

Semaphore: semaphore.wait()

Thread 1

Mutex: Unlocked
Buffer: SomeStruct {
   ...
}

Diagram shows a semaphore with a bucket indicating the buffer and a thread waiting on the semaphore.
Semaphores

```
semaphore.wait()
```

![Diagram of thread 1 waiting on a semaphore with an unlocked mutex and a buffer containing a `SomeStruct`]

- Mutex: Unlocked
- Buffer: `SomeStruct { ... }`
Semaphores

```python
semaphore.wait()
```

Mutex: Unlocked  Buffer:

```c
SomeStruct {
    ...
}
```
Semaphores

```
thread1

SomeStruct {
    ...
}

Mutex: Unlocked
Buffer:

mutex.lock()
```
Semaphores

```java
thread1

mutex.lock()

Mutex:  Locked
Buffer: SomeStruct {
   ...
}
```
Semaphores

SomeStruct {
    ...
}

Mutex: Locked

Buffer:
Semaphores

SomeStruct {
  ...
}

mutex.unlock()

thread1

Mutex: Locked

Buffer:
Semaphores

SomeStruct {
    ...
}

thread1

mutex.unlock()

Mutex: Unlocked
Buffer:
Semaphore

thread1

SomeStruct {
    ...
}

Mutex: Unlocked

Buffer:
Semaphores

semaphore.wait() (again)

SomeStruct {
    ...
}

Mutex: Unlocked

Buffer:
Semaphores

Semaphore.wait() (again)

Mutex: Unlocked

Buffer:

SomeStruct {
    ...
}

SomeStruct {
    ...
}

thread1 (blocked)

thread2
Semaphores

```java
SomeStruct {
    ...
}

Buffer:

Mutex: Unlocked

semaphore.wait() (again)

mutex.lock()
```

Thread 1 (blocked)

Thread 2

Mutex: Unlocked

Buffer:
Semaphores

```plaintext
Semaphore: Semaphore

thread1 (blocked)

Mutex: Locked

buffer:

thread2

SomeStruct {
...
}

mutex.lock()

SomeStruct {
...
}
Semaphores

Semaphore.wait() (again)

SomeStruct {
  ...
}

Mutex: Locked
Buffer:

Semaphore.wait() (again)

SomeStruct {
  ...
}

Mutex: Locked
Buffer:

Semaphore.wait() (again)

SomeStruct {
  ...
}

Mutex: Locked
Buffer:

Semaphore.wait() (again)

SomeStruct {
  ...
}

Mutex: Locked
Buffer:
Semaphores

```
SomeStruct {
    ...
}
```

Semaphore: wait() (again)

Mutex: Locked

Buffer: SomeStruct {
    ...
}

Thread1: Blocked

Thread2

Mutex.unlock()
Semaphores

Semaphore.wait() (again)

SomeStruct {
  ...
}

Mutex: Unlocked
Buffer:
  SomeStruct {
    ...
  }

thread1 (blocked)

mutex.unlock()
SomeStruct { 
  ...
}

Mutex: Unlocked
Buffer:

.instances

Semaphore

thread1 (blocked)

Semaphore.wait()

Semaphore.signal()
Semaphores

```c
SomeStruct {
  ...
}
```

Semaphore

```
semaphore.wait() (again)
```

Semaphore

```
semaphore.signal()
```

Thread 1: (Blocked)

Mutex: Unlocked

Buffer:

```
SomeStruct {
  ...
}
```
Semaphores

Semaphore.wait() (again)

SomeStruct {
    ...
}

Mutex: Unlocked

Buffer: SomeStruct {
    ...
}

thread1

thread2
Semaphore.wait() (again)
Semaphores

SomeStruct {
  ...
}

Buffer:

SomeStruct {
  ...
}

Mutex: Unlocked

thread1

thread2

mutex.lock()
Semaphores

```java
SomeStruct {
    ...
}

Buffer:
SomeStruct {
    ...
}

Mutex: Locked

mutex.lock()

thread1

thread2
```
Semaphores

SomeStruct {
    ...
}

SomeStruct {
    ...
}

Mutex: Locked
Buffer:
Semaphores

```cpp
SomeStruct {
    ...
}

SomeStruct {
    ...
}

Mutex: Locked
Buffer:
```
Semaphores

```cpp
SomeStruct {
    ...
}

Mutex: Unlocked

Buffer:

mutex.unlock()

thread1

thread2
```
Channels
Channels

let struct = receive_end.recv().unwrap()
let struct = receive_end.recv().unwrap()
let struct = receive_end.recv().unwrap()
let struct2 = receive_end.recv().unwrap() (again)
Channels

let struct2 = receive_end.recv().unwrap() (again)

SomeStruct {
  ...
}

thread1 (blocked)
let struct2 = receive_end.recv().unwrap() (again)
Channels

let struct2 = receive_end.recv().unwrap() (again)

send_end.send(struct).unwrap()
let struct2 = receive_end.recv().unwrap() (again)

send_end.send(struct).unwrap()
let struct2 = receive_end.recv().unwrap() (again)
let struct2 = receive_end.recv().unwrap() (again)
Channels: like strongly-typed pipes
Chrome architecture diagram

Inter-Process Communication channels:
  Pipes, but with an extra layer of abstraction to serialize/deserialize objects

https://www.chromium.org/developers/design-documents/multi-process-architecture (slightly out of date)
Using channels
Isn't message passing bad for performance?

- If you don't share memory, then you need to copy data into/out of messages. That seems expensive. What gives?
- Theory != practice
  - We share some memory (the heap) and only make shallow copies into channels
Partly-shared memory (shallow copies only)

Thread 1

Thread 2

Heap

Vec {
  len: 6,
  alloc_len: 16,
  data: Box<>,
}

[3, 4, 5, 6, 7, 8]
Partly-shared memory (shallow copies only)

Vec{
  len: 6,
  alloc_len: 16,
  data: Box<>
}

Heap

[3, 4, 5, 6, 7, 8]
Partly-shared memory (shallow copies only)

```
Vec {
    len: 6,
    alloc_len: 16,
    data: Box::<>
}
```

Heap

```
[3, 4, 5, 6, 7, 8]
```
Partly-shared memory (shallow copies only)

```
Vec {  
  len: 6,  
  alloc_len: 16,  
  data: Box<>,
}
```

Heap

```
[3, 4, 5, 6, 7, 8]
```
Partly-shared memory (shallow copies only)

Vec {
    len: 6,
    alloc_len: 16,
    data: Box<>
}

[3, 4, 5, 6, 7, 8]
Isn’t message passing bad for performance?

- If you don’t share memory, then you need to copy data into/out of messages. That seems expensive. What gives?
- Theory != practice
  - We share some memory (the heap) and only make shallow copies into channels
- In Go, passing pointers is potentially dangerous! Channels make data races less likely but don’t preclude races if you use them wrong
- In Rust, passing pointers (e.g. Box) is always safe despite sharing memory
  - When you send to a channel, ownership of value is transferred to the channel
  - The compiler will ensure you don’t use a pointer after it has been moved into the channel
The ideal channel is an MPMC (multi-producer, multi-consumer) channel
- We implemented one of these on Tuesday! A simple Mutex<VecDeque<>>, with a CondVar
- However, that approach is much slower than we’d like. (Why?)

It’s really, really hard to implement a fast and safe MPMC channel!
- Go’s channels are known for being slow
  - They essentially implement Mutex<VecDeque<>>, but using a “fast userspace mutex” (futex)
- A fast implementation needs to use lock-free programming techniques to avoid lock contention and reduce latency
The Rust standard library includes an MPSC (multi-producer, single-consumer) channel, but it’s not ideal (one of the oldest APIs in Rust stdlib)
- Great if you want multiple threads to send to one thread (e.g. aggregating results of an operation)
- Also great for thread-to-thread communication (superset of SPSC)
- Not so great if you want to distribute data/work (e.g. a work queue)
- Additionally, the API has some oddities (great article)
- There’s a good chance this channel implementation will be replaced within the next year or two (discussion)
Channel APIs and implementations

- The **crossbeam** crate recently (2018) added an excellent MPMC implementation
  - “If we were to redo Rust channels from scratch, how should they look?”
    - Much improved API
  - Mostly lock free
  - Even faster than the existing MPSC channels
  - Great read [here](#)
  - Likely to replace the stdlib channels in some capacity
fn main() {
    let (sender, receiver) = crossbeam::channel::unbounded();
}
fn main() {
    let (sender, receiver) = crossbeam::channel::unbounded();

    let mut threads = Vec::new();
    for _ in 0..num_cpus::get() {
                  
    channel {
        senders: 1,
        receivers: 1,
        ...
    }
}
fn main() {
    let (sender, receiver) = crossbeam::channel::unbounded();

    let mut threads = Vec::new();
    for _ in 0..num_cpus::get() {
        let receiver = receiver.clone();
    }
}
fn main() {
    let (sender, receiver) = crossbeam::channel::unbounded();
    let mut threads = Vec::new();
    for _ in 0..num_cpus::get() {
        let receiver = receiver.clone();
    }
}
fn main() {
    let (sender, receiver) = crossbeam::channel::unbounded();

    let mut threads = Vec::new();
    for _ in 0..num_cpus::get() {
        let receiver = receiver.clone();
        threads.push(thread::spawn(move || {

            Thread 1 stack
            Sender
            Receiver
            Receiver

            Heap
            channel {
                senders: 1,
                receivers: 1,
                ...
            }
        }));
    }
}
fn main() {
    let (sender, receiver) = crossbeam::channel::unbounded();

    let mut threads = Vec::new();
    for _ in 0..num_cpus::get() {
        let receiver = receiver.clone();
        threads.push(thread::spawn(move || {
            while let Ok(next_num) = receiver.recv() {
                factor_number(next_num);
            }
            ();
        }));
    }
}
fn main() {
    let (sender, receiver) = crossbeam::channel::unbounded();

    let mut threads = Vec::new();
    for _ in 0..num_cpus::get() {
        let receiver = receiver.clone();
        threads.push(thread::spawn(move || {
            while let Ok(next_num) = receiver.recv() {
                factor_number(next_num);
            }
        }));
    }
}
fn main() {
    let (sender, receiver) = crossbeam::channel::unbounded();

    let mut threads = Vec::new();
    for _ in 0..num_cpus::get() {
        let receiver = receiver.clone();
        threads.push(thread::spawn(move || {
            while let Ok(next_num) = receiver.recv() {
                factor_number(next_num);
            }
        }));
    }
}

Heap channel {
    senders: 1,
    receivers: 2,
    ... 
}

Thread 1 stack
Sender
Receiver

Thread 2 stack
Receiver
fn main() {
    let (sender, receiver) = crossbeam::channel::unbounded();

    let mut threads = Vec::new();
    for _ in 0..num_cpus::get() {
        let receiver = receiver.clone();
        threads.push(thread::spawn(move || {
            while let Ok(next_num) = receiver.recv() {
                factor_number(next_num);
            }
        }));
    }
}

Read until recv() returns Err (i.e. until the channel is closed)
fn main() {
    let (sender, receiver) = crossbeam::channel::unbounded();

    let mut threads = Vec::new();
    for _ in 0..num_cpus::get() {
        let receiver = receiver.clone();
        threads.push(thread::spawn(move || {
            while let Ok(next_num) = receiver.recv() {
                factor_number(next_num);
            }
        }));
    }

    let stdin = std::io::stdin();
    for line in stdin.lock().lines() {
        let num = line.unwrap().parse::<u32>().unwrap();
    }
}
fn main() {
    let (sender, receiver) = crossbeam::channel::unbounded();

    let mut threads = Vec::new();
    for _ in 0..num_cpus::get() {
        let receiver = receiver.clone();
        threads.push(thread::spawn(move || {
            while let Ok(next_num) = receiver.recv() {
                factor_number(next_num);
            }
        }));
    }

    let stdin = std::io::stdin();
    for line in stdin.lock().lines() {
        let num = line.unwrap().parse::<u32>().unwrap();
        sender.send(num).expect("Tried writing to channel, but there are no receivers!");
    }
}
fn main() {
    let (sender, receiver) = crossbeam::channel::unbounded();

    let mut threads = Vec::new();
    for _ in 0..num_cpus::get() {
        let receiver = receiver.clone();
        threads.push(thread::spawn(move || {
            while let Ok(next_num) = receiver.recv() {
                factor_number(next_num);
            }
        }));
    }

    let stdin = std::io::stdin();
    for line in stdin.lock().lines() {
        let num = line.unwrap().parse::<u32>().unwrap();
        sender
            .send(num)
            .expect("Tried writing to channel, but there are no receivers!");
    }

    drop(sender);
}
fn main() {
    let (sender, receiver) = crossbeam::channel::unbounded();

    let mut threads = Vec::new();
    for _ in 0..num_cpus::get() {
        let receiver = receiver.clone();
        threads.push(thread::spawn(move || {
            while let Ok(next_num) = receiver.recv() {
                factor_number(next_num);
            }
        }));
    }

    let stdin = std::io::stdin();
    for line in stdin.lock().lines() {
        let num = line.unwrap().parse::<u32>().unwrap();
        sender.send(num).expect("Tried writing to channel, but there are no receivers!");
    }

drop(sender);
fn main() {
    let (sender, receiver) = crossbeam::channel::unbounded();

    let mut threads = Vec::new();
    for _ in 0..num_cpus::get() {
        let receiver = receiver.clone();
        threads.push(thread::spawn(move || {
            while let Ok(next_num) = receiver.recv() {
                factor_number(next_num);
            }
        })�)
    });

    let stdin = std::io::stdin();
    for line in stdin.lock().lines() {
        let num = line.unwrap().parse::<u32>().unwrap();
        sender
            .send(num)
            .expect("Tried writing to channel, but there are no receivers!");
    }

drop(sender);
fn main() {
    let (sender, receiver) = crossbeam::channel::unbounded();

    let mut threads = Vec::new();
    for _ in 0..num_cpus::get() {
        let receiver = receiver.clone();
        threads.push(thread::spawn(move || {
            while let Ok(next_num) = receiver.recv() {
                factor_number(next_num);
            }
        }));
    }

    let stdin = std::io::stdin();
    for line in stdin.lock().lines() {
        let num = line.unwrap().parse::<u32>().unwrap();
        sender.send(num).expect("Tried writing to channel, but there are no receivers!");
    }

drop(sender);
```rust
fn main() {
    let (sender, receiver) = crossbeam::channel::unbounded();

    let mut threads = Vec::new();
    for _ in 0..num_cpus::get() {
        let receiver = receiver.clone();
        threads.push(thread::spawn(move || {
            while let Ok(next_num) = receiver.recv() {
                factor_number(next_num);
            }
        }));
    }

    let stdin = std::io::stdin();
    for line in stdin.lock().lines() {
        let num = line.unwrap().parse::<u32>().unwrap();
        sender.send(num).expect("Tried writing to channel, but there are no receivers!");
    }

drop(sender);

    for thread in threads {
        thread.join().expect("Panic occurred in thread");
    }
}
```
Pick the right tool for the job

- Using channels is often much simpler and safer than using mutexes + CVs
  - Even in Rust, mutexes can still cause problems if you lock/unlock at the wrong times
  - E.g. semaphore will break if you unlock after cv.wait() and then re-lock before decrementing the counter. You hold the lock while touching the counter, so the compiler doesn't complain, but there is still a race condition

- However, channels aren't always the best choice
  - Not very well suited for global values (e.g. caches or global counters)