Intro to Multithreading

Ryan Eberhardt and Armin Namavari
May 5, 2020

Class logistics

- Fill out weekly survey by Wednesday night
- First project (mini GDB) is out
    - You'll be free to work with a partner!
    - We'll have some way for you to find someone to work with if you'd like (suggestions welcome)
    - This is going to be a very systems-y project — you'll be dealing with registers, assembly, multiprocessing etc. so get ready for that!
    - It's going to be like trace but even more fun :^)
- By this Sunday, aim to complete Milestone 4

You rock! 🧗‍♀️

- Our journey so far
    - Learned a new model for managing memory safety
    - Met new, explicit types for the purposes of handling nulls and errors
    - Worked with a new alternative to inheritance in object-oriented programming
    - Implemented generic container types
    - Used with heap-allocated memory and reference-counted pointers
    - Battled a lot of compiler errors (and hopefully learned from them!)
    - Learned about ways to safely use constructs such as fork, pipes, and signals

You rock! 🧗‍♀️

- The journey ahead
  - Next two weeks: how to do safe multithreading
  - Week 7: asynchronous programming
  - Week 8: robustness in networked services
  - Week 9: looking back and looking around
  - Week 10: case studies and guest lectures

Plan for Today

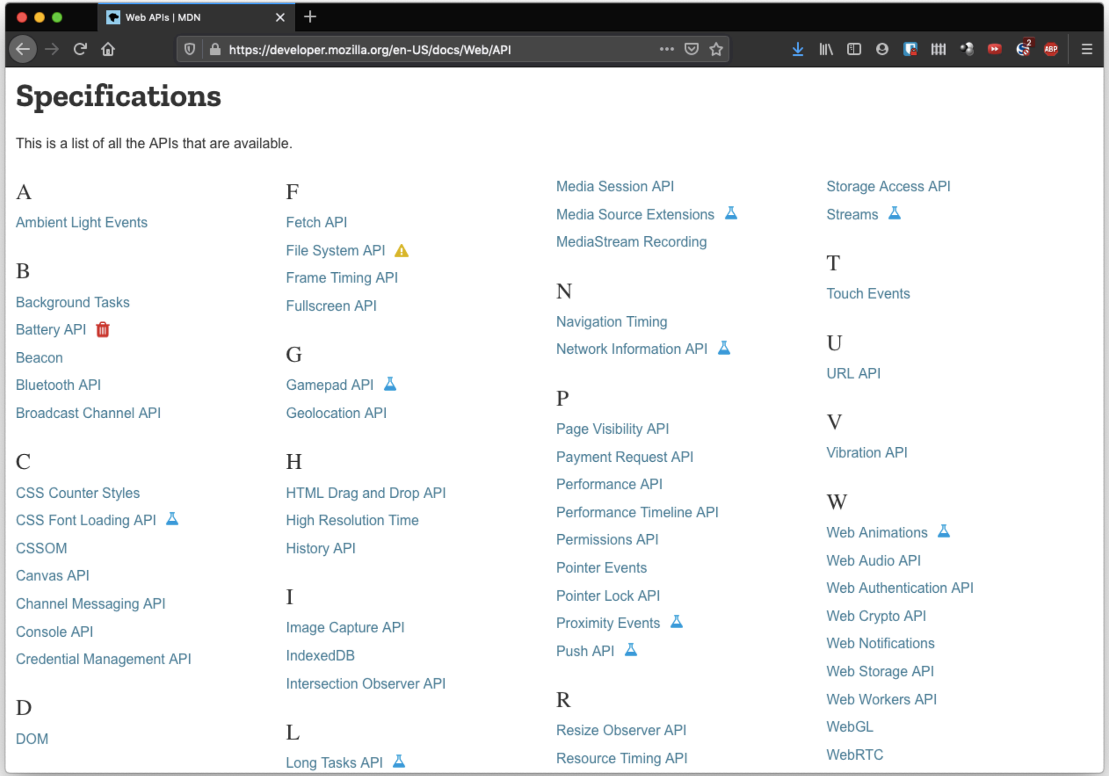
- Revisit discussion of Google Chrome
- Introduce multithreading in Rust
- If time permits: introduce locks/mutexes

Google Chrome

Considerations when designing a browser

- Speed
  - Typically faster to share memory and to use lightweight synchronization primitives
- Memory usage
  - Processes use more memory
- Battery/CPU usage
  - Threads incur less context switching overhead
- Ease of development
  - Communication is WAY easier using threads
  - (That being said, bugs caused by multithreading are extremely hard to track down)
- Security, stability
  - Multiprocessing provides isolation. Multithreading does not.

Modern browsers are essentially operating systems



https://developer.mozilla.org/en-US/docs/Web/API

Modern browsers are essentially operating systems

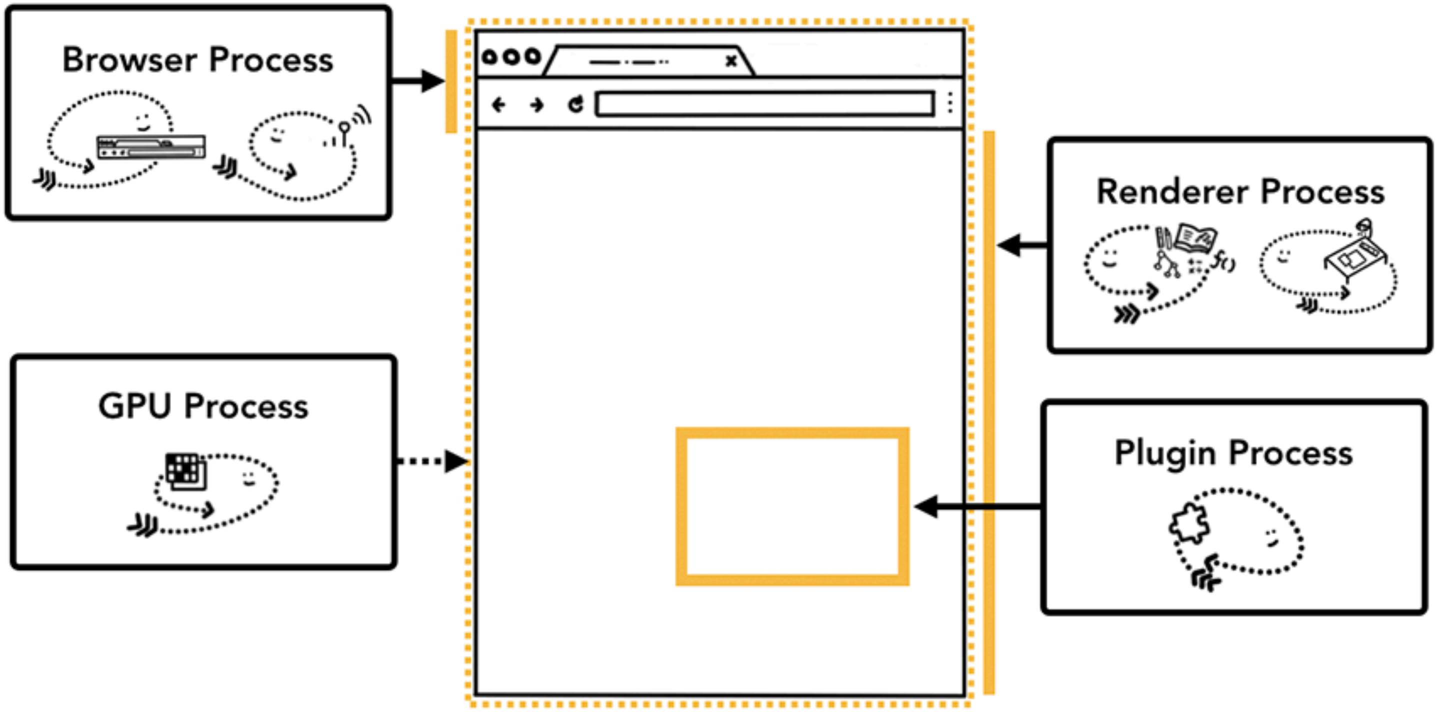
- Storage APIs
- Concurrency APIs
- Hardware APIs (e.g. communicate with MIDI devices, even GPU)
- Run assembly
- Run Windows 95: https://win95.ajf.me/

Motivation for Chrome

> *It's nearly impossible to build a rendering engine that never crashes or hangs. It's also nearly impossible to build a rendering engine that is perfectly secure.*
>
> *In some ways, the state of web browsers around 2006 was like that of the single-user, co-operatively multi-tasked operating systems of the past. As a misbehaving application in such an operating system could take down the entire system, so could a misbehaving web page in a web browser. All it took is one browser or plug-in bug to bring down the entire browser and all of the currently running tabs.*
>
> *Modern operating systems are more robust because they put applications into separate processes that are walled off from one another. A crash in one application generally does not impair other applications or the integrity of the operating system, and each user's access to other users' data is restricted.*

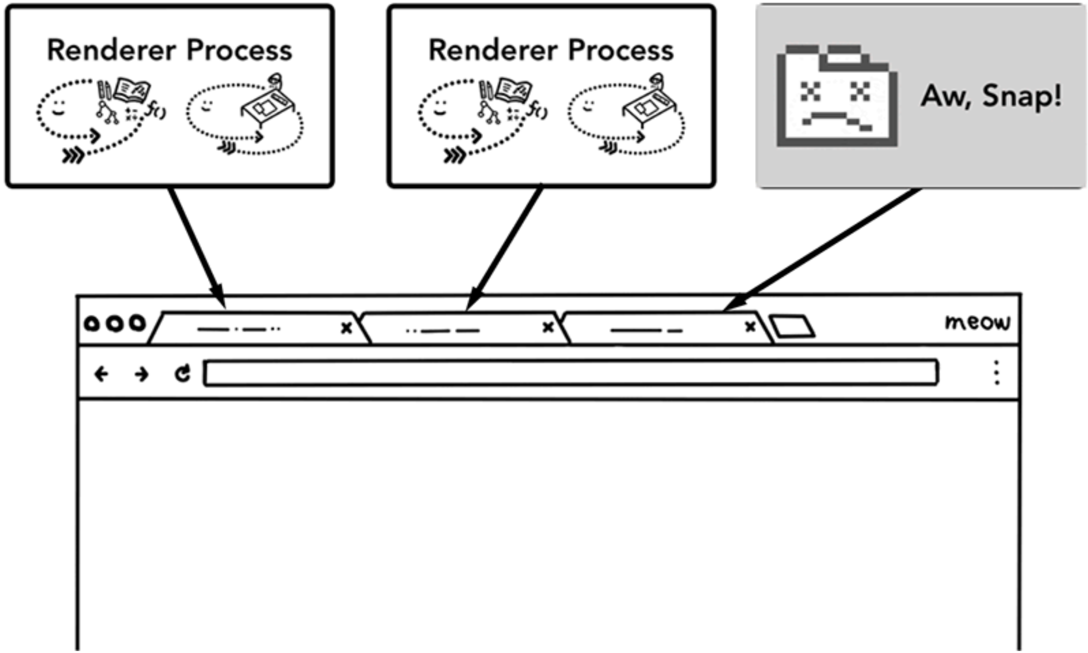
https://www.chromium.org/developers/design-documents/multi-process-architecture

Chrome architecture



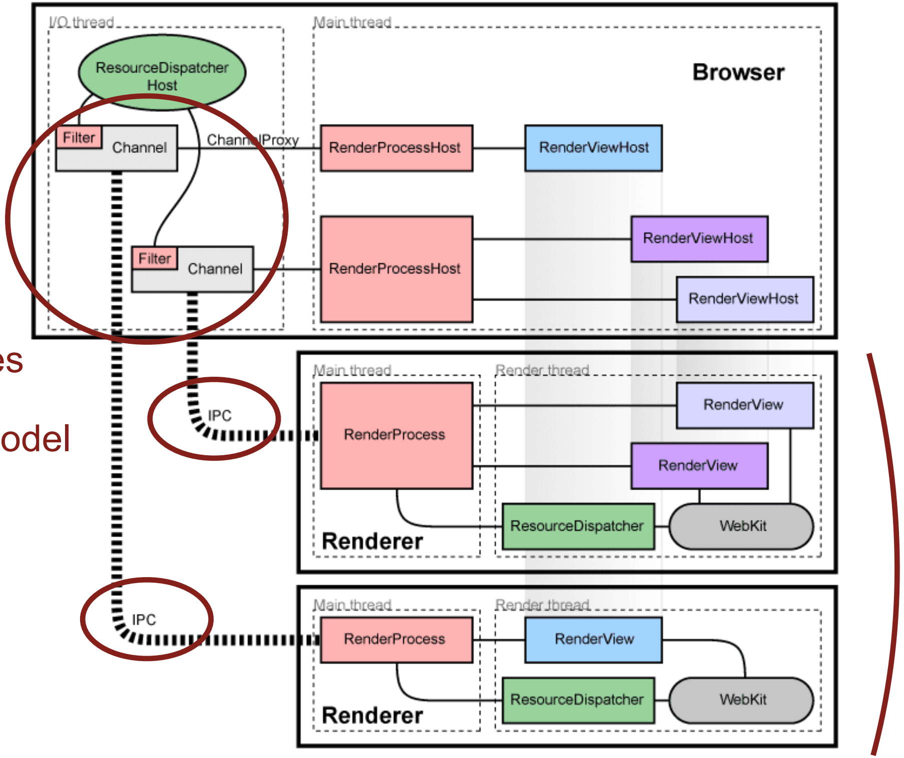
REALLY CUTE diagrams from https://developers.google.com/web/updates/2018/09/inside-browser-part1
(great read!)

Chrome architecture



REALLY CUTE diagrams from https://developers.google.com/web/updates/2018/09/inside-browser-part1
(great read!)

Chrome architecture



IPC channels = pipes

Message passing model

Events (e.g. click, keystroke, etc) are relayed through these pipes! No signals

Sandboxed processes: no access to network, filesystem, etc

If there is embedded content, may use multiple threads to render that content and manage communication between frames

https://www.chromium.org/developers/design-documents/multi-process-architecture (slightly out of date)

Not good enough

- What does all this work buy us?
  - Isolation between tabs
  - Isolation between (potentially malicious) websites and the host
- What does it *not* buy us?
  - Isolation between resources *within* a tab

Not good enough

Same-origin policy: www.evil.com can embed bank.com, but cannot interact with bank.com or see its data

Not good enough

- Site Isolation Project (2015-2019) aimed to put resources for different origins in different processes
- Extremely difficult undertaking. Cross-frame communication is common (JS postMessage API), and embedded frames need to share render buffers
  - Involved rearchitecting the most core parts of Chrome
- Became especially important in Jan 2018: Spectre and Meltdown
  - When the hardware fails to uphold its guarantees, JS can read arbitrary process memory (even kernel memory, and even if your software has no bugs)!
- Paper/video: https://www.usenix.org/conference/usenixsecurity19/presentation/reis

Anatomy of a sandbox escape

- https://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html (2012 but it's more accessible than some other writeups)
  - First exploit chains together *six bugs* to escape the sandbox
  - Second one uses *ten(!!)*
- https://googleprojectzero.blogspot.com/2019/04/virtually-unlimited-memory-escaping.html (2019)

More relevant reading

- How Chrome does fork():
  http://neugierig.org/software/chromium/notes/2011/08/zygote.html
  Fun related bug report: https://bugs.chromium.org/p/chromium/issues/detail?id=35793
  What steps will reproduce the problem?
  1. Develop a webapp, use chrome's devtools, minding your own business
  2. In the meantime, let chrome silently autoupdate in the background

  What is the expected result?
  Devtools continue working

  What happens instead?
  Devtools break after refreshing the page after the autoupdate happened.

Multithreading

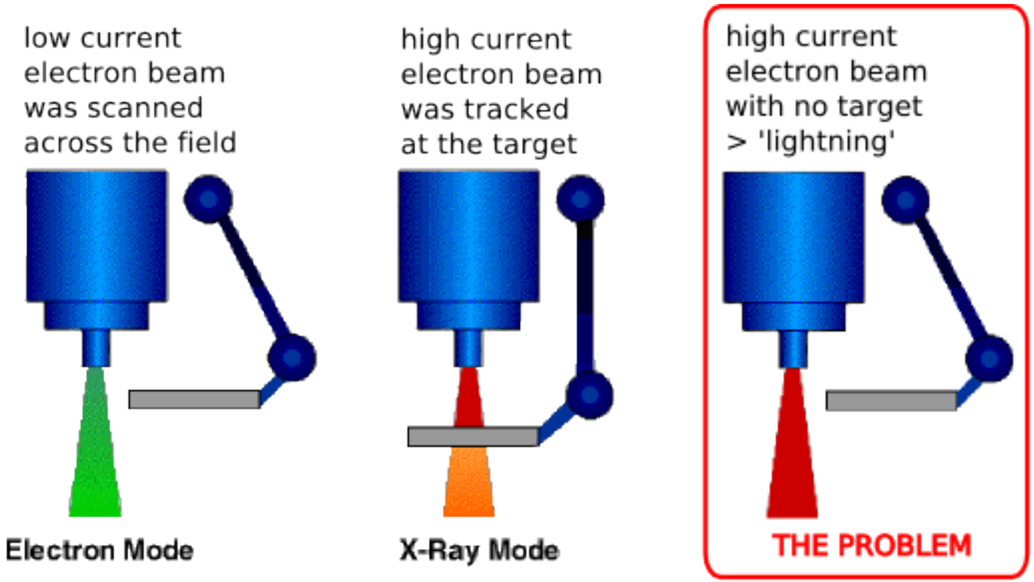
Perils of concurrency

- Why is multithreading nice?
- Why is multithreading dangerous?
    - Race conditions
    - Deadlock (more on Thursday and next week)

Perils of concurrency

https://hci.cs.siue.edu/NSF/Files/Semester/Week13-2/PPT-Text/Slide13.html
https://hackaday.com/2015/10/26/killed-by-a-machine-the-therac-25/

Perils of concurrency



http://radonc.wikidot.com/radiation-accident-therac25

Perils of concurrency

After each overdose the creators of Therac-25 were contacted. After the first incident the AECL responses was simple: "After careful consideration, we are of the opinion that this damage could not have been produced by any malfunction of the Therac-25 or by any operator error (Leveson, 1993)."

After the 2nd incident the AECL sent a service technician to the Therac-25 machine, he was unable to recreate the malfunction and therefore conclude nothing was wrong with the software. Some minor adjustments to the hardware were changed but the main problems still remained.

It was not until the fifth incident that any formal action was taken by the AECL. However it was a physicist at the hospital where the 4th and 5th incident took place in Tyler, Texas who actually was able to reproduce the mysterious "malfunction 54". The AECL finally took action and made a variety of changes in the software of the Therac-25 radiation treatment system.

http://radonc.wdfiles.com/local--files/radiation-accident-therac25/Therac_UGuelph_TGall.pdf

Investigation results:

- **The failure occurred only when a particular nonstandard sequence of keystrokes was entered** on the VT-100 terminal which controlled the PDP-11 computer: an "X" to (erroneously) select 25 MeV photon mode followed by "cursor up", "E" to (correctly) select 25 MeV Electron mode, then "Enter", all within eight seconds.
- The design did not have any hardware interlocks to prevent the electron-beam from operating in its high-energy mode without the target in place.
- The engineer had reused software from older models. These models had hardware interlocks that masked their software defects.
- The hardware provided no way for the software to verify that sensors were working correctly.
- **The equipment control task did not properly synchronize with the operator interface task, so that race conditions occurred** if the operator changed the setup too quickly. This was missed during testing, since it took some practice before operators were able to work quickly enough to trigger this failure mode.
- The software set a flag variable by incrementing it, rather than by setting it to a fixed non-zero value. Occasionally an arithmetic overflow occurred, causing the flag to return to zero and the software to bypass safety checks.

https://en.wikipedia.org/wiki/Therac-25 and http://sunnyday.mit.edu/papers/therac.pdf

What are race conditions?

- Race condition:
  A race condition or race hazard is the condition of an electronics, software, or other system where the system's substantive behavior is dependent on the sequence or timing of other uncontrollable events. (Wikipedia)
- Data race:
  Multiple threads access a value, where at least one of them is writing
  - This should sound familiar!

Rust's design pays off

- Rust's design goals:
  - How do you do safe systems programming?
  - How do you make concurrency painless?
  - How do you make it fast?
- *"Initially these [first two] problems seemed orthogonal, but to our amazement, the solution turned out to be identical: the same tools that make Rust safe also help you tackle concurrency head-on."* (Rust blog)
- Compiler enforces rules for safe concurrency. *"Thread safety isn't just documentation; it's law."*
- There's very little in the core language specific to threading! (Only two traits!)

Hello world!

```
use std::{thread, time};
use rand::Rng;

const NUM_THREADS: u32 = 20;

fn main() {
    let mut threads = Vec::new();
    println!("Spawning {} threads...", NUM_THREADS);
    for _ in 0..NUM_THREADS {
        threads.push(thread::spawn(|| {
            let mut rng = rand::thread_rng();
            thread::sleep(time::Duration::from_millis(rng.gen_range(0, 5000)));
            println!("Thread finished running!");
        }));
    }
    // wait for all the threads to finish
    for handle in threads {
        handle.join().expect("Panic happened inside of a thread!");
    }
    println!("All threads finished!");
}
```

Parameters for closure function (none, in this case)

Closure/lambda function borrows any referenced variables

A panic in a thread will not crash the entire program
Need to check if the thread panicked

[Playground]

Extroverts demo (CS 110)

```
static const char *kExtroverts[] = {
  "Frank", "Jon", "Lauren", "Marco", "Julie", "Patty",
  "Tagalong Introvert Jerry"
};
static const size_t kNumExtroverts = sizeof(kExtroverts)/sizeof(kExtroverts[0]) - 1;

static void *recharge(void *args) {
  const char *name = kExtroverts[*(size_t *)args];
  printf("Hey, I'm %s.  Empowered to meet you.\n", name);
  return NULL;
}

int main() {
  printf("Let's hear from %zu extroverts.\n", kNumExtroverts);
  pthread_t extroverts[kNumExtroverts];
  for (size_t i = 0; i < kNumExtroverts; i++)
    pthread_create(&extroverts[i], NULL, recharge, &i);
  for (size_t j = 0; j < kNumExtroverts; j++)
    pthread_join(extroverts[j], NULL);
  printf("Everyone's recharged!\n");
  return 0;
}
```

Passes a pointer to i, but then the main thread changes i on the next iteration of the for loop

[Cplayground]

Can we do the same in Rust?

```
use std::thread;

const NAMES: [&str; 7] = ["Frank", "Jon", "Lauren", "Marco", "Julie", "Patty",
    "Tagalong Introvert Jerry"];

fn main() {
    let mut threads = Vec::new();
    for i in 0..6 {
        threads.push(thread::spawn(|| {
            println!("Hello from printer {}!", NAMES[i]);
        }));
    }
    // wait for all the threads to finish
    for handle in threads {
        handle.join().expect("Panic occurred in thread!");
    }
}
```

[Rust playground]

Can we do the same in Rust?

```
error[E0373]: closure may outlive the current function, but it borrows `i`, which is owned by the
current function
  --> src/main.rs:9:36
   |
9  |         threads.push(thread::spawn(|| {
   |                                    ^^ may outlive borrowed value `i`
10 |             println!("Hello from printer {}!", NAMES[i]);
   |                                                      - `i` is borrowed here
   |
note: function requires argument type to outlive `'static`
  --> src/main.rs:9:22
   |
9  |           threads.push(thread::spawn(|| {
   |  ______________________^
10 | |             println!("Hello from printer {}!", NAMES[i]);
11 | |         }));
   | |_________^
help: to force the closure to take ownership of `i` (and any other referenced variables), use the
`move` keyword
   |
9  |         threads.push(thread::spawn(move || {
   |                                    ^^^^^^^
```

Can we do the same in Rust?

```
error[E0373]: closure may outlive the current function, but it borrows `i`, which is owned by the
current function
  --> src/main.rs:9:36
   |
9  |         threads.push(thread::spawn(|| {
   |                                    ^^ may outlive borrowed value `i`
10 |             println!("Hello from printer {}!", NAMES[i]);
   |                                                      - `i` is borrowed here
   |
note: function requires argument type to outlive `'static`
  --> src/main.rs:9:22
   |
9  |           threads.push(thread::spawn(|| {
   |  ______________________^
10 | |             println!("Hello from printer {}!", NAMES[i]);
11 | |         }));
   | |_________^
help: to force the closure to take ownership of `i` (and any other referenced variables), use the
`move` keyword
   |
9  |         threads.push(thread::spawn(move || {
   |                                    ^^^^^^^
   |
```

Can we do the same in Rust?

```
use std::thread;

const NAMES: [&str; 7] = ["Frank", "Jon", "Lauren", "Marco", "Julie", "Patty",
    "Tagalong Introvert Jerry"];

fn main() {
    let mut threads = Vec::new();
    for i in 0..6 {
        threads.push(thread::spawn(move || {
            println!("Hello from printer {}!", NAMES[i]);
        }));
    }
    // wait for all the threads to finish
    for handle in threads {
        handle.join().expect("Panic occurred in thread!");
    }
}
```

Closure function takes ownership of i (under the hood, value of i is copied into thread's stack)

[Rust playground]

Ticket agents demo (CS 110)

```
static void ticketAgent(size_t id, size_t& remainingTickets) {
    while (remainingTickets > 0) {
        handleCall(); // sleep for a small amount of time to emulate conversation time.
        remainingTickets--;
        cout << oslock << "Agent #" << id << " sold a ticket! (" << remainingTickets
             << " more to be sold)." << endl << osunlock;
        if (shouldTakeBreak()) // flip a biased coin
            takeBreak();        // if comes up heads, sleep for a random time to take a break
    }
    cout << oslock << "Agent #" << id << " notices all tickets are sold, and goes home!"
         << endl << osunlock;
}

int main(int argc, const char *argv[]) {
    thread agents[10];
    size_t remainingTickets = 250;
    for (size_t i = 0; i < 10; i++)
        agents[i] = thread(ticketAgent, 101 + i, ref(remainingTickets));
    for (thread& agent: agents) agent.join();
    cout << "End of Business Day!" << endl;
    return 0;
}
```

Multiple threads get mutable reference to remainingTickets

Value decremented simultaneously: ends up underflowing!

Cplayground

Can we do the same in Rust?

```rust
fn ticketAgent(id: usize, remainingTickets: &mut usize) {
    while *remainingTickets > 0 {
        handleCall();
        *remainingTickets -= 1;
        println!("Agent #{} sold a ticket! ({} more to be sold)",
            id, remainingTickets);
        if shouldTakeBreak() {
            takeBreak();
        }
    }
    println!("Agent #{} notices all tickets are sold, and goes home!", id);
}
```

Rust playground

Can we do the same in Rust?

```
fn main() {
    let mut remainingTickets = 250;

    let mut threads = Vec::new();
    for i in 0..10 {
        threads.push(thread::spawn(|| {
            ticketAgent(i, &mut remainingTickets)
        }));
    }
    // wait for all the threads to finish
    for handle in threads {
        handle.join().expect("Panic occurred in thread!");
    }
    println!("End of business day!");
}
```

Rust playground

Can we do the same in Rust?

```
error[E0499]: cannot borrow `remainingTickets` as mutable more than once at a time
  --> src/main.rs:38:36
   |
38 |             threads.push(thread::spawn(|| {
   |                                        -              ^^ mutable borrow starts here in previous iteration of
loop
   |  ______________________|
   | |
39 | |                 ticketAgent(i, &mut remainingTickets)
   | |                                   ---------------- borrows occur due to use of
`remainingTickets` in closure
40 | |             }));
   | |__________- argument requires that `remainingTickets` is borrowed for `'static`

error[E0373]: closure may outlive the current function, but it borrows `i`, which is owned by the
current function
...

error[E0373]: closure may outlive the current function, but it borrows `remainingTickets`, which
is owned by the current function
...
```