Welcome back^2 to CS439!

No quiz everyone say AWW!

Stress

- 439H is **not an easy class**
  - Lots of new material
  - Unfamiliar programming environments
  - Fast, often relentless pace
- Struggling in this course is normal
  - There will be times you won’t know the answer or solution
  - This is expected - we want everyone to succeed, but the only way we can help is if you ask for it
- If you find yourself overwhelmed or spending more time on this class than you think you should be, **please reach out** to Dr. Gheith or the TAs
  - We can help out as far as the class goes
  - We can provide other resources if we are not able to help

Mental health resources available at UT

A reminder on your health

- If you are sick or have some personal emergencies, reach out to us on Ed privately
    - We can accommodate your situation as needed
- Please do not show up in-person if you are feeling sick!
    - This includes lectures, discussion, and office hours
    - If you have to miss a quiz because of illness, let us know on Ed!

Quiz

Question 1

- Why is it “all or nothing” with the Stopping prints?
- Does the program terminate?
- Deadlocks?
- Race conditions?

```
volatile int barrier1 = 4;
volatile int barrier2 = 4;

void kernelMain() {
    Debug::printf("Starting\n");
    barrier1 -= 1;
    while (barrier1 > 0) {}

    Debug::printf("Stopping\n");
    barrier2 -= 1;

    if (SMP::me() == 0) {
        while (barrier2 > 0) {}
        Debug::shutdown();
    }
}
```

Question 1

Core 0:

```
print Starting
load barrier1
sub barrier1, 1
store barrier1
```

(barrier)

```
print Stopping
load barrier2
sub barrier2, 1
store barrier2
```

(barrier)

```
shutdown
```

Core 1:

```
print Starting
load barrier1
sub barrier1, 1
store barrier1
```

(barrier)

```
print Stopping
load barrier2
sub barrier2, 1
store barrier2
```

Core 2:

```
print Starting
load barrier1
sub barrier1, 1
store barrier1
```

(barrier)

```
print Stopping
load barrier2
sub barrier2, 1
store barrier2
```

Core 3:

```
print Starting
load barrier1
sub barrier1, 1
store barrier1
```

(barrier)

```
print Stopping
load barrier2
sub barrier2, 1
store barrier2
```

Question 1

Can one core pass the first barrier without all the others also passing it?

Core 0:

```
print Starting
load barrier1
sub barrier1, 1
store barrier1
```

(barrier)

```
print Stopping
load barrier2
sub barrier2, 1
store barrier2
```

(barrier)

```
shutdown
```

Core 1:

```
print Starting
load barrier1
sub barrier1, 1
store barrier1
```

(barrier)

```
print Stopping
load barrier2
sub barrier2, 1
store barrier2
```

Core 2:

```
print Starting
load barrier1
sub barrier1, 1
store barrier1
```

(barrier)

```
print Stopping
load barrier2
sub barrier2, 1
store barrier2
```

Core 3:

```
print Starting
load barrier1
sub barrier1, 1
store barrier1
```

(barrier)

```
print Stopping
load barrier2
sub barrier2, 1
store barrier2
```

Question 1

Core 0:

```
print Starting
load barrier1
sub barrier1, 1
store barrier1
```

(barrier)

```
print Stopping
load barrier2
sub barrier2, 1
store barrier2
```

(barrier)

```
shutdown
```

Core 1:

```
print Starting
load barrier1
sub barrier1, 1
store barrier1
```

(barrier)

```
print Stopping
load barrier2
sub barrier2, 1
store barrier2
```

Core 2:

```
print Starting
load barrier1
sub barrier1, 1
store barrier1
```

(barrier)

```
print Stopping
load barrier2
sub barrier2, 1
store barrier2
```

Core 3:

```
print Starting
load barrier1
sub barrier1, 1
store barrier1
```

(barrier)

```
print Stopping
load barrier2
sub barrier2, 1
store barrier2
```

Question 1

Core 0:

```
print Starting
load barrier1
sub barrier1, 1
store barrier1
```

(barrier)

```
print Stopping
load barrier2
sub barrier2, 1
store barrier2
```

(barrier)

```
shutdown
```

Core 1:

```
print Starting
load barrier1
sub barrier1, 1
store barrier1
```

(barrier)

```
print Stopping
load barrier2
sub barrier2, 1
store barrier2
```

Core 2:

```
print Starting
load barrier1
sub barrier1, 1
store barrier1
```

(barrier)

```
print Stopping
load barrier2
sub barrier2, 1
store barrier2
```

Core 3:

```
print Starting
load barrier1
sub barrier1, 1
store barrier1
```

(barrier)

```
print Stopping
load barrier2
sub barrier2, 1
store barrier2
```

Question 1

Core 0:

```
print Starting
load barrier1
sub barrier1, 1
store barrier1
```

(barrier)

```
print Stopping
load barrier2
sub barrier2, 1
store barrier2
```

(barrier)

```
shutdown
```

Core 1:

```
print Starting
load barrier1
sub barrier1, 1
store barrier1
```

(barrier)

```
print Stopping
load barrier2
sub barrier2, 1
store barrier2
```

Core 2:

```
print Starting
load barrier1
sub barrier1, 1
store barrier1
```

(barrier)

```
print Stopping
load barrier2
sub barrier2, 1
store barrier2
```

Core 3:

```
print Starting
load barrier1
sub barrier1, 1
store barrier1
```

(barrier)

```
print Stopping
load barrier2
sub barrier2, 1
store barrier2
```

Bonus question: why don't the prints get interleaved?

Question 2

- Why `compare_exchange`?

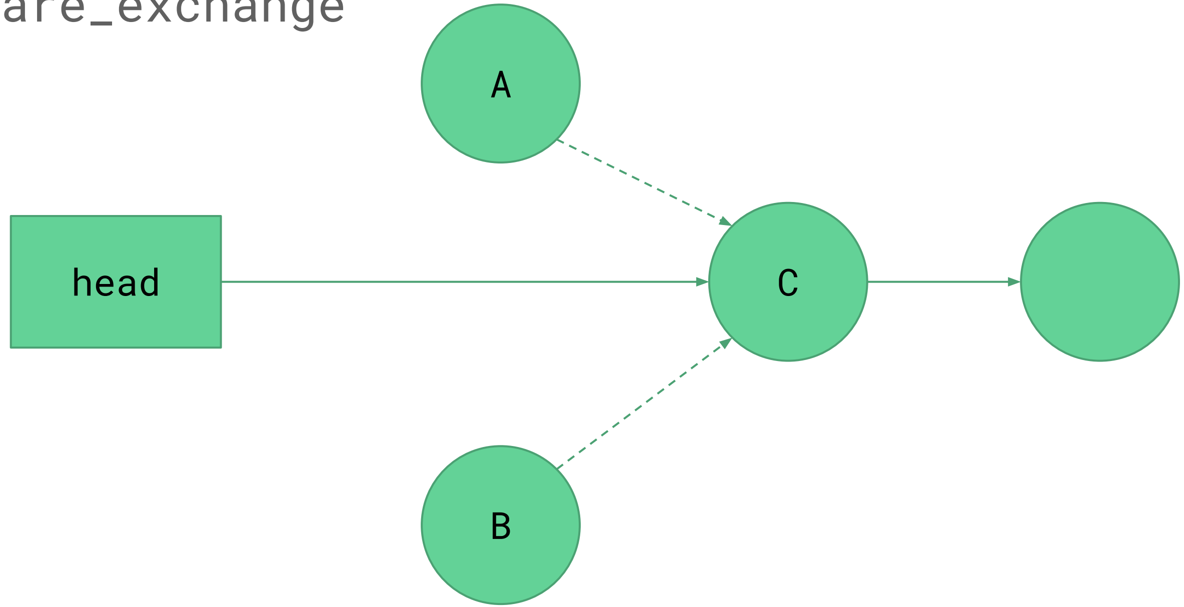
Question 2

- Why `compare_exchange`?
  - Atomically **and conditionally** set a value
    - Much stronger atomic primitive than simpler swaps
  - Don’t update state if it has been changed since the last read
  - Avoid corrupting a data structure/synchronization primitive in the presence of concurrent accesses/modifications

Note: Implementing a simple spinlock doesn’t need a compare exchange, only an atomic exchange. Can you see why? (Look at Gheith’s `SpinLock`)

Question 2

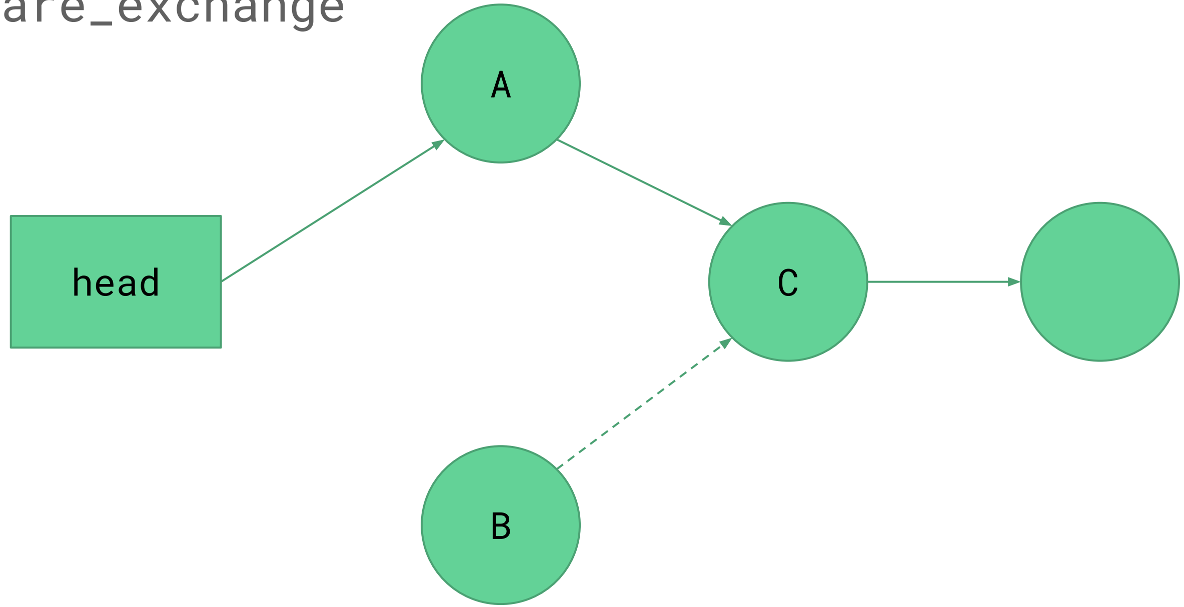
- `no compare_exchange`



head = C

Question 2

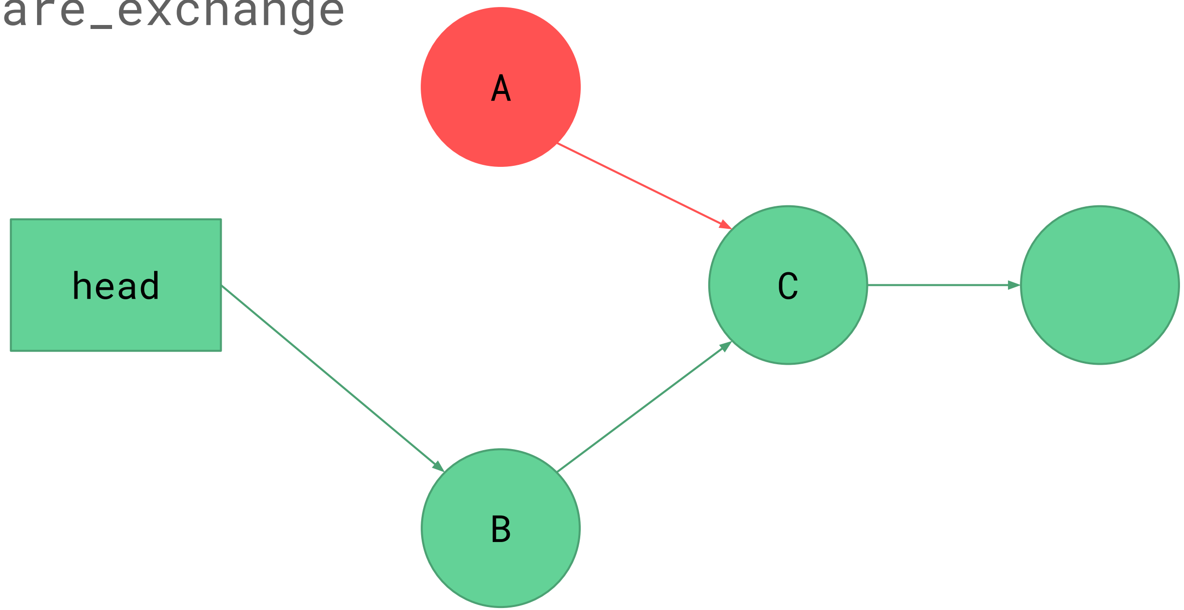
- `no compare_exchange`




```
store(head, A)
head = A
```

Question 2

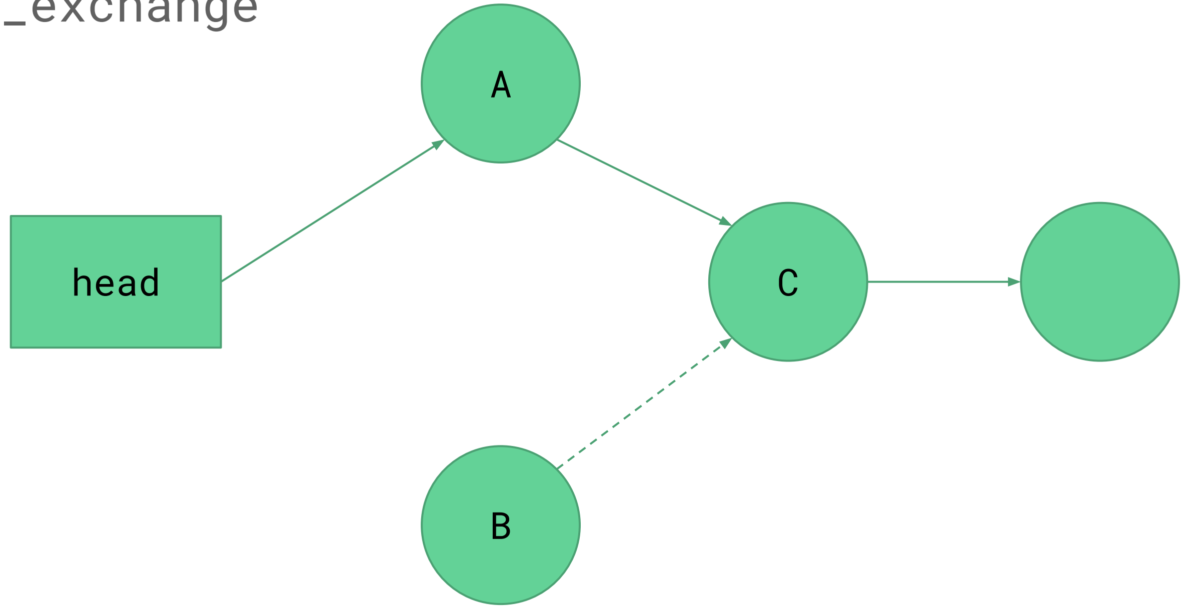
- `No compare_exchange`




```
store(head, B)
head = B
A is no longer in list!
```

Question 2

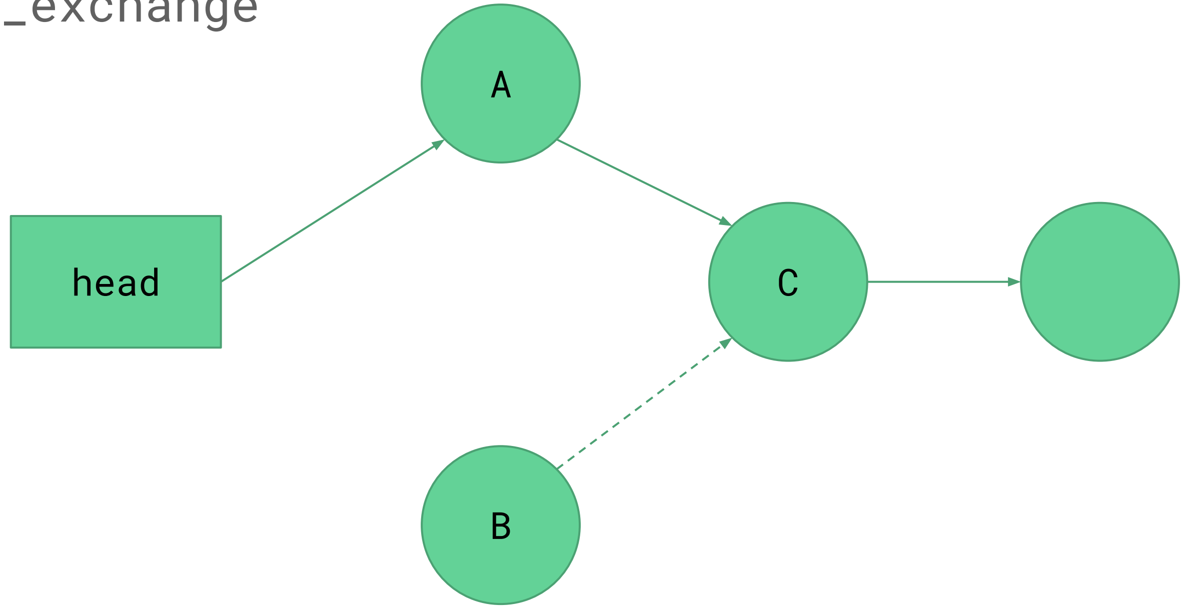
- `compare_exchange`




```
compare_exchange(head, C, A)
head = A
```

Question 2

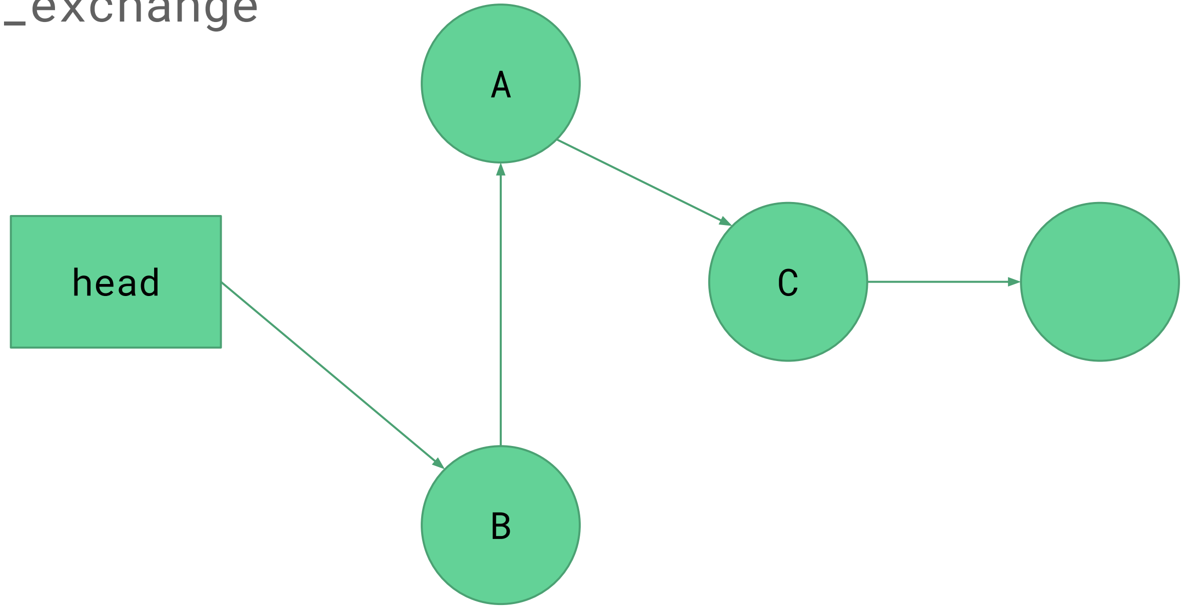
- `compare_exchange`




```
compare_exchange(head, C, B)
head = A
```

Question 2

- `compare_exchange`




```
compare_exchange(head, A, B)
head = B
```

Question 2

- Combining `owner` and `flag`

Question 2

- Combining `owner` and `flag`

```
Atomic<int> owner{-1};

void critical(Work work) {

      if is_recursive_call {

            work();

      } else {

            while (!owner.compare_exchange(-1, SMP::me()));

            work();

            owner.store(-1);

      }

}
```

Question 2

```
uint32_t fetch_add(uint32_t* var, uint32_t increment) {
    compare_exchange(var, *var, *var + increment);
    return *var - increment;
}
```

Question 2

```
uint32_t fetch_add(uint32_t* var, uint32_t increment) {
    uint32_t previous = *var;
    while (!compare_exchange(var, previous, previous + increment)) {
        previous = *var;
    }
    return previous;
}
```

Question 3

```
SpinLock global_lock{};
unordered_map<int, SpinLock> locks;
unordered_map<int, int> owner;

template<typename Work>
void critical(Work work, uint32_t id) {
      global_lock.lock();
      if(locks.count(id)){
            locks[id] = new SpinLock{};
            owner[id] = -1;
      }
      global_lock.unlock();

      if(owner[id] == SMP::me()){
            work();
            return;
      }

      locks[id].lock();
      owner[id] = SMP::me();

      work();

      owner[id] = -1;
      locks[id].unlock();
}
```

```
//alternative implementation using atomics

global.lock();
if (!id in map) {
      taken[id] = false;
      owner[id] = -1;
}
id_owner = owner[id];
global.unlock()
if (id_owner == me) { work(); }
else {
      while (true) {
            global_lock()
            if !taken[id].exchange(true) {
                  owner[id] = me
                  global.unlock();
                  break;
            } else {
                  global_unlock();
            }
      }
      work();
      global.lock()'
      taken[id] = false;
      owner[id] = -1;
      global.unlock();
}
```

Question 3

```
SpinLock global_lock{};
unordered_map<int, SpinLock> locks;
unordered_map<int, int> owner;

template<typename Work>
void critical(Work work, uint32_t id) {
      global_lock.lock();
      if(locks.count(id)){
            locks[id] = new SpinLock{};
            owner[id] = -1;
      }
      global_lock.unlock();

      if(owner[id] == SMP::me()){
            work();
            return;
      }

      locks[id].lock();
      owner[id] = SMP::me();

      work();

      owner[id] = -1;
      locks[id].unlock();
}
```

```
//example situation 1

int counter1 = 0;
int counter2 = 0;

//core 1
critical([](){
      for(int i = 0; i < 10000; i++) counter1++;
}, 0);

//core 2
critical([](){
      for(int i = 0; i < 10000; i++) counter2++;
}, 1);
```

Question 3

```
SpinLock global_lock{};
unordered_map<int, SpinLock> locks;
unordered_map<int, int> owner;

template<typename Work>
void critical(Work work, uint32_t id) {
      global_lock.lock();
      if(locks.count(id)){
            locks[id] = new SpinLock{};
            owner[id] = -1;
      }
      global_lock.unlock();

      if(owner[id] == SMP::me()){
            work();
            return;
      }

      locks[id].lock();
      owner[id] = SMP::me();

      work();

      owner[id] = -1;
      locks[id].unlock();
}
```

```
//example situation 2

int counter1 = 0;
int counter2 = 0;

//core 1
critical([](){
      for(int i = 0; i < 10000; i++) counter1++;
}, 0);

//core 2
critical([](){
      for(int i = 0; i < 10000; i++) counter2++;
}, 0);
```

Question 3

```
SpinLock global_lock{};
unordered_map<int, SpinLock> locks;
unordered_map<int, int> owner;

template<typename Work>
void critical(Work work, uint32_t id) {
	global_lock.lock();
	if(locks.count(id)){
		locks[id] = new SpinLock{};
		owner[id] = -1;
	}
	global_lock.unlock();

	if(owner[id] == SMP::me()){
		work();
		return;
	}

	locks[id].lock();
	owner[id] = SMP::me();

	work();

	owner[id] = -1;
	locks[id].unlock();
}
```

```
//example situation 3

int counter1 = 0;

//core 1
critical([](){
	for(int i = 0; i < 10000; i++) counter1++;
}, 0);

//core 2
critical([](){
	for(int i = 0; i < 10000; i++) counter1++;
}, 1);
```

Question 3

```
SpinLock global_lock{};
unordered_map<int, SpinLock> locks;
unordered_map<int, int> owner;

template<typename Work>
void critical(Work work, uint32_t id) {
	global_lock.lock();
	if(locks.count(id)){
		locks[id] = new SpinLock{};
		owner[id] = -1;
	}
	global_lock.unlock();

	if(owner[id] == SMP::me()){
		work();
		return;
	}

	locks[id].lock();
	owner[id] = SMP::me();

	work();

	owner[id] = -1;
	locks[id].unlock();
}
```

```
//example situation 4

//core 1
critical([](){
	//do some stuff
	critical([](){/*do some stuff*/}, 1);
	//do some stuff
}, 0);

//core 2
critical([](){
	//do some stuff
	critical([](){/*do some stuff*/}, 0);
	//do some stuff
}, 1);
```

P2

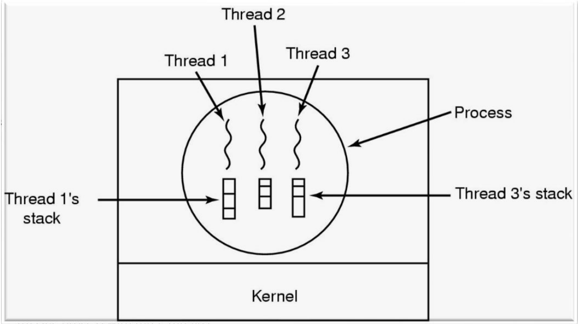
```
while (true) {
    check_feedback();
}
ASSERT(
    feedback.max() != 'A'
);
```

How is p2 going?

A. oops i forgot that we have a project
B. Cloned the project.
C. Looked through the starter code.
D. Started planning/writing code
E. Done with at least one part of the project
F. Done with the whole project but still failing a couple test cases
G. Fully done

Stacks

- Dedicated Stacks
  - Each task has its own stack to use
  - Can use the stack to save its own state, and simply swap stacks
  - p5 last semester (coroutines)
- Non-Dedicated Stacks (aka “Stackless”)
  - Still uses a stack when running!
  - Does not have a **personally allocated** stack
  - Doesn't keep the stack across "suspension points" (await)
  - Must save its state somewhere *besides* the stack
  - p2



Each thread has its own stack

“Stacks are like having a stable income”
- Unknown

Concurrency

- Cores
  - The actual hardware processors that we can use to achieve parallelism
- Threads
  - Allow for true parallelism
- Coroutines
  - Allow for non-parallel concurrency
- For p2, threads/coroutines are just logical groupings of callbacks
- This isn't standard terminology – this is just what we're using for this class

Channels

- What does it mean for a buffer to be size one?
- What is the point of the buffer?
- What does it mean to send on a channel which is already full?

Channels - Backpressure

- Fast sender to a slow receiver?
- Unlimited buffer – run out of memory
- Backpressure – slows down sender

```
void kernelMain(void) {
    auto* channel = new Channel<uint32_t>();

    // Send an infinite stream of incrementing numbers
    auto* counter = new uint32_t(0);
    go([channel, counter] { send_loop(channel, counter); });

    // Process an infinite stream of numbers
    channel->receive([channel](uint32_t value) {
        receive_loop(channel, value);
    });

    // Run for two seconds
    go([] { Debug::shutdown(); }, 2000);
}
```

```
void send_loop(Channel<uint32_t>* channel, uint32_t* counter) {
    *counter += 1;
    Debug::printf("Sending value: %d\n", *counter);

    channel->send(*counter, [channel, counter] {
        send_loop(channel, counter);
    });
}
void receive_loop(Channel<uint32_t>* channel, uint32_t value) {
    Debug::printf("Received value: %d\n", value);

    for (int i = 0; i < 100'000; i++) {
        value ^= i; // Do some slow work
        pause();
    }

    channel->receive([channel](uint32_t value) {
        receive_loop(channel, value);
    });
}
```

(you can also do it with functors – explicit/manual closures)

```
void kernelMain(void) {
    auto channel = new Channel<uint32_t>();

    // Send an infinite stream of incrementing numbers
    go(Sender(channel));

    // Process an infinite stream of numbers
    channel->receive(Receiver(channel));

    // Run for two seconds
    go([] { Debug::shutdown(); }, 2000);
}
```

```
struct Sender {
    uint32_t* i;
    Channel<uint32_t>* ch;
    Sender(Channel<uint32_t>* ch) : i(new uint32_t(0)), ch(ch) {}

    void operator()() const {
        *i += 1;
        Debug::printf("Sending value: %d\n", *i);
        ch->send(*i, *this);
    }
};
struct Receiver {
    Channel<uint32_t>* ch;
    Receiver(Channel<uint32_t>* ch) : ch(ch) {}

    void operator()(uint32_t value) const {
        Debug::printf("Received value: %d\n", value);
        for (int i = 0; i < 100'000; i++) {
            value ^= i; // Do some slow work
            pause();
        }
        ch->receive(*this);
    }
};
```

(and with actual closures...)

```
void kernelMain(void) {
    auto channel = new Channel<uint32_t>();

    // Send an infinite stream of incrementing numbers
    go([channel]() {
        uint32_t* i = new uint32_t(0);
        auto send_inner = [channel, i](auto& self) -> void {
            auto cont = [self] { self(self); };

            *i += 1;
            Debug::printf("Sending value: %d\n", *i);
            channel->send(*i, cont);
        };
        send_inner(send_inner);
    });

    // Receive an infinite stream of incrementing numbers
    channel->receive([channel](auto value) {
        auto recv_inner = [channel](auto& self, auto value) -> void {
            auto cont = [self](auto value) { self(self, value); };

            Debug::printf("Received value: %d\n", value);
            for (int i = 0; i < 100'000; i++) {
                value ^= i;
                pause();
            }
            channel->receive(cont);
        };
        recv_inner(recv_inner, value);
    });

    // Run for two seconds
    go([] {
        Debug::shutdown();
    }, 2000);
}
```

Why so many synchronization primitives?

- Imagine that we want to implement the core synchronization part as few times as possible
  - i.e. scheduling callbacks properly, queueing callbacks for later, etc.
- What fundamental primitives could we use to achieve this?
- Given synchronization primitive x, could you use it to easily implement y?

Questions?