```
void work() {
    Debug::printf("*** hello %d \n", counter);
    Debug::printf("*** goodbye %d \n", counter);
    counter++;
}

void kernelMain(void) {
    critical(work);
}
```

↳ callback: calls function when the conditions are right

Lambda Function

```
void kernelMain(void) {
    critical([] {
        Debug::printf("*** hello %d \n", counter);
        Debug::printf("*** goodbye %d \n", counter);
        counter++;
    });
}
```

work()

- function doesn't have a name
- embedded within another function
- can nest lambdas within lambdas
- Note: ONLY in C++ (not in C)

↓ Why?

```
void f() {
    :
    int x;
    :
    void g() {
        :
        x
        :
    }
    return g;
}
```

now if g is called externally it will use x which would've been de-allocated once f() ends

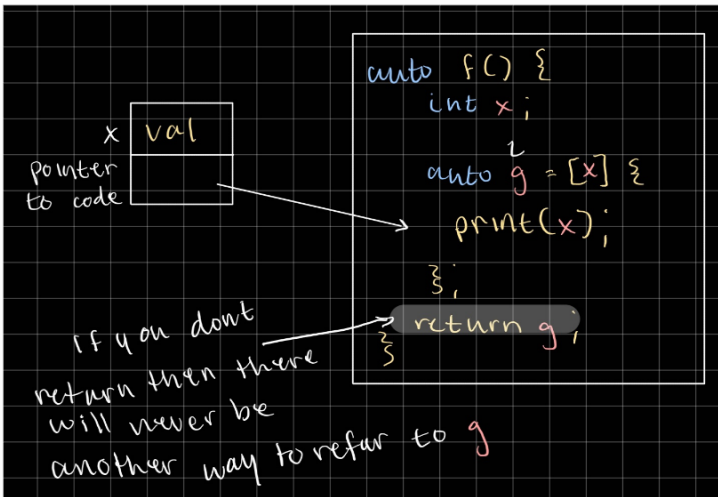
- you would need to capture the variable x
- smth about closures ??

```
void f() {
    int x;
    auto g = [x] {
        print(x);
    };
}
```

tells c++ to figure out the type (still strongly typed)

[] brackets are equivalent to fun key word from fun lang

[x] putting x into brackets tells language to capture x



if you dont return then there will never be another way to refer to g

Note: you cannot capture a pointer unless you know that the life time of whatever its pointing to will outlive the lambda function

↳ you could malloc the data to

↳ solve this

otherwise the pointer will become invalid once you leave the scope bc the data would be lost

Why can't the compiler automatically capture variables that are used in the lambda?

you can!

```
auto f() {
    int x;
    auto g = [=] {
        print(x);
    };
    return g;
}
```

putting = tells the compiler to figure things out

Note:

```
auto g = [](x){
    print(x);
}
```

can put parentheses to get parameters (just like normal function)

Note: C++ is memory UNSAFE!! because you can have a memory reference outlive the data at the memory location

- C++ creates a class for every lambda function
  - ↳ this class contains all the captured values as instance variables and the lambda function as a method
  - ↳ also creates an instance of the class to be returned

```
class ... {

  int x;

  public:

    T operator-() (...) {
        :
        :
    }
}
```

overwrites operators (this one overwrites ())
- this allows you to call the class like a method
- this is what the code looks like for a lambda

Structural Polymorphism

Nominal polymorphism: allows you to reference a group of types (ex. interfaces in java)
- ↳ there's nothing like that in c++ for lambdas

Note: After compilation the types are erased (erasure polymorphism)

```
inline void critical(Work work) {
    :
    :
    work();
    :
    :
}
```

- Note: in C++ it would compile and create multiple copies of critical function for each variation of work (every type of lambda)
  - ↳ redundant!!
- Note: There's no definition of what work is so you could technically pass anything to the function
  - BUT will fail at compile time if the structure doesn't match!!

Smth abt Reification ↑ making something that is abstract more concrete

↑ making some... that is abstract ... concrete

Project #2 Cooperative Multithreading

Fundamental Idea: Cores are an expensive resource!

You Don't want the cores to be spinning idly

2 options:
1) go into sleep (low power mode)
2) find something else to do * better than sleep

- If #2 fails then do #1
- give thread() a callable function

Its like cores are puppies and threads are treats
- ↳ you throw threads at the cores and someone will pick it up
- ↳ A thread can only be run by one core (no parallelism within the thread)
- ↳ The thread will appear to execute sequentially (even if passed to different cores)

- Shutdown when kernelMain() Returns, OR if shutdown is called directly
  - ↳ Both can easily create race conditions (even if other cores are still running (your problem))
- One core will call kernelMain() which will start handing out threads
  - ↳ there is a yeild() which allows kernelMain() gets handed to another thread

Need to implement 3 functions:

stop(): a way for thread to say it's done and to not run it again
- ↳ delete resources
  - ↳ There is an implicit stop when you reach end of thread

yeild(): cooperative part - allows another thread (if available) to run on the current core

thread(): creates new thread

Need some synchronization methods
↳ some way to communicate with one another

↳ There are a few abstractions:

Promise: a promise of a value (sometimes a future in different languages) saying "you can come to me to get a value"
- initially has no value
- ```
  auto answer = new Promise<int>();
  answer.set();   ← should only be called once
  answer.get();   ↳ undefined
  ```
  ↳ if get() is called and the promise has no value then the thread is blocked
  ↳ like a single use channel
  ↳ the value doesn't get consumed after get() is called

Barrier:
- auto b = new Barrier(x); ← how many threads it expects to arrive

b.sync() ← blocks threads until x threads arrive to the sync point

Note: in the git log there is a single core implementation

Project One

- Once cores are awaken, they call kernelMain()
- There is an original core that wakes up the other cores, on start up
  ↳ will be idle (asleep)

- In P1 kernelMain() will be called by all 16 cores
  ↳ (will be changed in P2 to have ONE core enter main and others wait for instructions)
  ↳ All cores will have their own stack (instance) of kernelMain()
- kernelMain() will be part of the test
  ↳ actually runs the test

* First thing [Gheith] would do: create test

Copy test:
```
% cp t0.cc [user].cc
% cp t0.ok [user].ok
```

how to run:
```
% ./run-qemu [test_name]
```

fix g++:
```
PATH=~gheith/public/cs439/bin:$PATH
```

Test Case t0

```
static uint32_t counter = 0;
void work() {
  Debug::printf("*** hello %d\n", counter);
  Debug::printf("*** goodbye %d\n", counter);
  counter++;
}

void kernelMain(void) {
  critical(work);
}
```

global variable: same for all cores
↳ creates issue if multiple cores run work() at same time

creates critical section which allows only one core to enter a critical section at a time
↳ having ONLY ONE critical section is BAD! makes cores wait for other cores

Implementation

```
static uint32_t counter = 0;
void work() {
  Debug::printf("*** hello %d\n", counter);
  Debug::printf("*** goodbye %d\n", counter);
  counter++;
}
bool someone_IS_in_critical_section = false;

void kernelMain(void) {
  while (someone_IS_in_critical_section); // do nothing
  someone_IS_in_critical_section = true;
  work(); // changed for simplicity
  someone_IS_in_critical_section = false;
}
```

↳ doesn't solve problem

ctrl was not created w/ parallelism in mind
↳ compiler optimizes the loop out b/c it doesn't think the loop will ever run

Problems:
① compiler optimizations can mess up the code
↳ Never expects a global variable to be changed externally

- Could do
  ```
  volatile bool someone_IS_in_critical_section = false;
  ```
  ↳ tells compiler to try using memory reference As Much As possible
  ↳ STILL not fullproof

you want stuff to be Atomic
↳ "indivisible"
↳ keeps a section of
  ```

*Look up: Peterson Algorithm

code together and prevents other things from running during it
↳ very hard to implement in software
↳ usually implemented in hardware

* USE lock keyword to protect an operation

Note: Standard C++ has an atomic function
- having this on compiler allows it to work on any architecture

- __atomic_exchange_n

```
static uint32_t counter = 0;
void work() {
  Debug::printf("*** hello %d\n", counter);
  Debug::printf("*** goodbye %d\n", counter);
  counter++;
}
bool someone_is_in_critical_section = false;

void kernelMain(void) {
  while (                                    );
  someone_is_in_critical_section = true;
  work(); // changed for simplicity
}
```

__atomic_exchange_n(&someone_is_in_critical_section, 1, __ATOMIC_SEQ_CST) == 1

↳ updates current value AND returns previous value

require sequential consistency
↳ Compiler might move things around

__atomic_store(&someone_is_in_critical_section, 0, __ATOMIC_SEQ_CST) == 1

↳ only updates current value
- DOESN'T return previous

* It's good practice to stay consistent w/ using atomic for a particular variable

* Should probably use atomic_clear
↳ creates critical section

Operating Systems

What is it?

* Files are a lie!!
- There are no files built into disks (they are sectors)
- Files are abstract
- when a file is made it finds the number of files it needs and remembers which sector #s have the data

* permissions also fake!!

↳ Resource Management
abstraction
identification

- OS gets uploaded first: It takes over the system (like a colonizer)
  - There is a priveledge system that tells what program has authority over the system
  - Priveledge State: allows program to run anything

- Hardware:
  memory, IO Bus, 1+ cores, etc
  SMP: Shared Memory Processor

↑ NONE of this can do anything!! you NEED SOFTWARE!!

OS kernel
- first layer of software
- first thing loaded to system
- finds all the resources

User Processor
- Second layer of software
- There are multiple independent processors
- Kernel assigns memory to user processors
- typically is unaware of stuff outside of it's allocated resources ("thinks it owns the world")
- there can be more user processes than cores
  ↳ kernel will figure out switching off or running everything.
- can throw exceptions

This class mainly focus on the kernel

The "Operating System" is the kernel + first few user processes

NONE OF THIS WORKS

what loads the kernel?
bootloader

what loads the bootloader
Firmware

why can't the firmware load the kernel?
Flexibility

where is the firmware?
Manufacturer puts a peice of non-volatile READ ~~ONLY~~ mostly memory

Firmware:
contains first instructions

We're using QEMU emulator
- Need to install QEMU