CS110 Lecture 09: Threads

Principles of Computer Systems

Winter 2021

Stanford University

Computer Science Department

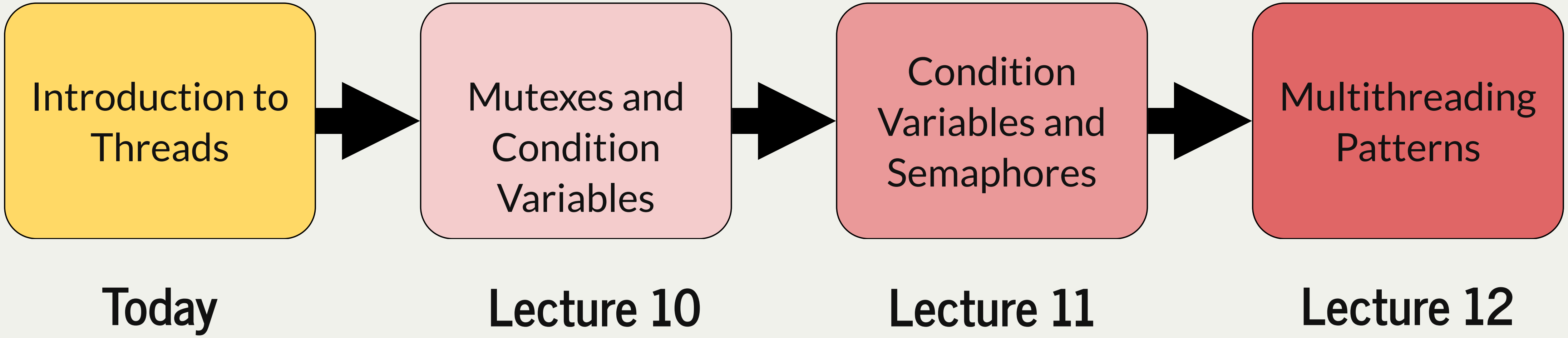
Instructors: Chris Gregg and

Nick Troccoli

PDF of this presentation

CS110 Topic 3: How can we have concurrency within a single process?

Learning About Processes



Today's Learning Goals

- Learn about how threads allow for concurrency within a single process
- Understand the differences between threads and processes
- Discover some of the pitfalls of threads sharing the same virtual address space

Plan For Today

- Threads
  - C threads
  - C++ threads

Threads

A **thread** is an independent execution sequence within a single process.

- Most common: assign each thread to execute a single function in parallel
- Each thread operates within the same process, so they *share global data* (!) (text, data, and heap segments)
- They each have their own stack (e.g. for calls within a single thread)
- Execution alternates between threads as it does for processes
- Many similarities between threads and processes; in fact, threads are often called **lightweight processes**.

Threads vs. Processes

Processes:

- isolate virtual address spaces (good: security and stability, bad: harder to share info)
- can run external programs easily (fork-exec) (good)
- harder to coordinate multiple tasks within the same program (bad)

Threads:

- share virtual address space (bad: security and stability, good: easier to share info)
- can't run external programs easily (bad)
- easier to coordinate multiple tasks within the same program (good)

Lecture 09: Introduction to Threads

- ANSI C doesn't provide native support for threads.
    - But **`pthreads`**, which comes with all standard UNIX and Linux installations of **`gcc`**, provides thread support, along with other related concurrency directives..
    - The primary **`pthreads`** data type is the **`pthread_t`**, which is an opaque type used to manage the execution of a function within its own thread of execution.
    - The only **`pthreads`** functions we'll need (before formally transitioning to C++ threads) are **`pthread_create`** and **`pthread_join`**.
    - Here's a very small program illustrating how **pthreads** work (see next slide for live demo).

```c
static void *recharge(void *args) {
    printf("I recharge by spending time alone.\n");
    return NULL;
}

static const size_t kNumIntroverts = 6;
int main(int argc, char *argv[]) {
    printf("Let's hear from %zu introverts.\n", kNumIntroverts);
    pthread_t introverts[kNumIntroverts];
    for (size_t i = 0; i < kNumIntroverts; i++)
        pthread_create(&introverts[i], NULL, recharge, NULL);
    for (size_t i = 0; i < kNumIntroverts; i++)
        pthread_join(introverts[i], NULL);
    printf("Everyone's recharged!\n");
    return 0;
}
```

Lecture 09: Introduction to Threads

https://cplayground.com/embed?p=guanaco-seahorse-aardvark

Lecture 09: Introduction to Threads

- The program on the prior slide declares an array of six **`pthread_t`** handles.
- The program initializes each **`pthread_t`** (via **`pthread_create`**) by installing **`recharge`** as the thread routine each **`pthread_t`** should execute.
- All thread routines take a **`void *`** and return a **`void *`**. That's the best C can do to support generic programming.
- The second argument to **`pthread_create`** is used to set a thread priority and other attributes. We can just pass in **`NULL`** if all threads should have the same priority. That's what we do here.
- The fourth argument is passed verbatim to the thread routine as each thread is launched. In this case, there are no meaningful arguments, so we just pass in **`NULL`**.
- Each of the six **`recharge`** threads is eligible for processor time the instant the surrounding **`pthread_t`** has been initialized.
- The six threads compete for thread manager's attention, and we have very little control over what choices it makes when deciding what thread to run next.
- **`pthread_join`** is to threads what **`waitpid`** is to processes.
- The main thread of execution blocks until the child threads all exit.
- The second argument to **`pthread_join`** can be used to catch a thread routine's return value. If we don't care to receive it, we can pass in **`NULL`** to ignore it.

Lecture 09: Introduction to Threads: concurrency!

- When you introduce any form of concurrency, you need to be careful to avoid concurrency issues like race conditions and deadlock.
- Here's a slightly more involved program where friends meet up (see next slide for live demo):

```c
static const char *kFriends[] = {
    "Langston", "Manan", "Edward", "Jordan", "Isabel", "Anne",
    "Imaginary"
};

static const size_t kNumFriends = sizeof(kFriends)/sizeof(kFriends[0]) - 1; // count excludes imaginary friend!

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

int main() {
    printf("Let's hear from %zu friends.\n", kNumFriends);
    pthread_t friends[kNumFriends];
    for (size_t i = 0; i < kNumFriends; i++)
        pthread_create(&friends[i], NULL, meetup, &i);
    for (size_t j = 0; j < kNumFriends; j++)
        pthread_join(friends[j], NULL);
    printf("Is everyone accounted for?\n");
    return 0;
}
```

Lecture 09: Introduction to Threads

https://cplayground.com/embed?p=falcon-hedgehog-peafowl

Lecture 09: Introduction to Threads

- Here are a few sample runs that clearly illustrate that the program on the previous slide is severely broken.

```
cgregg@myth63$ ./friends
Let's hear from 6 friends.
Hey, I'm Jordan.  Empowered to meet you.
Hey, I'm Anne.  Empowered to meet you.
Hey, I'm Anne.  Empowered to meet you.
Hey, I'm Anne.  Empowered to meet you.
Hey, I'm Anne.  Empowered to meet you.
Hey, I'm Imaginary.  Empowered to meet you.
Is everyone accounted for?
cgregg@myth63$ ./friends
Let's hear from 6 friends.
Hey, I'm Jordan.  Empowered to meet you.
Hey, I'm Anne.  Empowered to meet you.
Hey, I'm Imaginary.  Empowered to meet you.
Hey, I'm Imaginary.  Empowered to meet you.
Hey, I'm Imaginary.  Empowered to meet you.
Hey, I'm Imaginary.  Empowered to meet you.
Is everyone accounted for?
cgregg@myth63$ ./friendsLet's hear from 6 friends.
Hey, I'm Jordan.  Empowered to meet you.
Hey, I'm Isabel.  Empowered to meet you.
Hey, I'm Isabel.  Empowered to meet you.
Hey, I'm Anne.  Empowered to meet you.
Hey, I'm Anne.  Empowered to meet you.
Hey, I'm Imaginary.  Empowered to meet you.
Is everyone accounted for?
```

Lecture 09: Introduction to Threads

- Clearly something is wrong, but why?
  - Note that `meetup` references its incoming parameter now, and that `pthread_create` accepts the address of the surrounding loop's index variable `i` via its fourth parameter. `pthread_create`'s fourth argument is always passed verbatim as the single argument to the thread routine.
  - The problem? The main thread advances `i` without regard for the fact that `i`'s address was shared with six child threads.
    - At first glance, it's easy to absentmindedly assume that pthread_create captures not just the address of `i`, but the value of `i` itself. That assumption of course, is incorrect, as it captures the address and nothing else.
    - The address of `i` (even after it goes out of scope) is constant, but its contents evolve in parallel with the execution of the six `meetup` threads. `*(size_t *)args` takes a snapshot of whatever `i` happens to contain at the time it's evaluated.
    - Often, the majority of the `meetup` threads only execute after the main thread has worked through all of its first for loop. The space at `&i` is left with a 6, and that's why `Imaginary` is printed so often.
  - This is another example of a race condition, and is typical of the types of problems that come up when multiple threads share access to the same data.

Lecture 09: Introduction to Threads

- Fortunately, the fix is simple.
- We just pass the relevant **`const char *`** instead. Snapshots of the **`const char *`** pointers are passed verbatim to **`meetup`**. The strings themselves are constants.
- Full program illustrating the fix can be found right here.

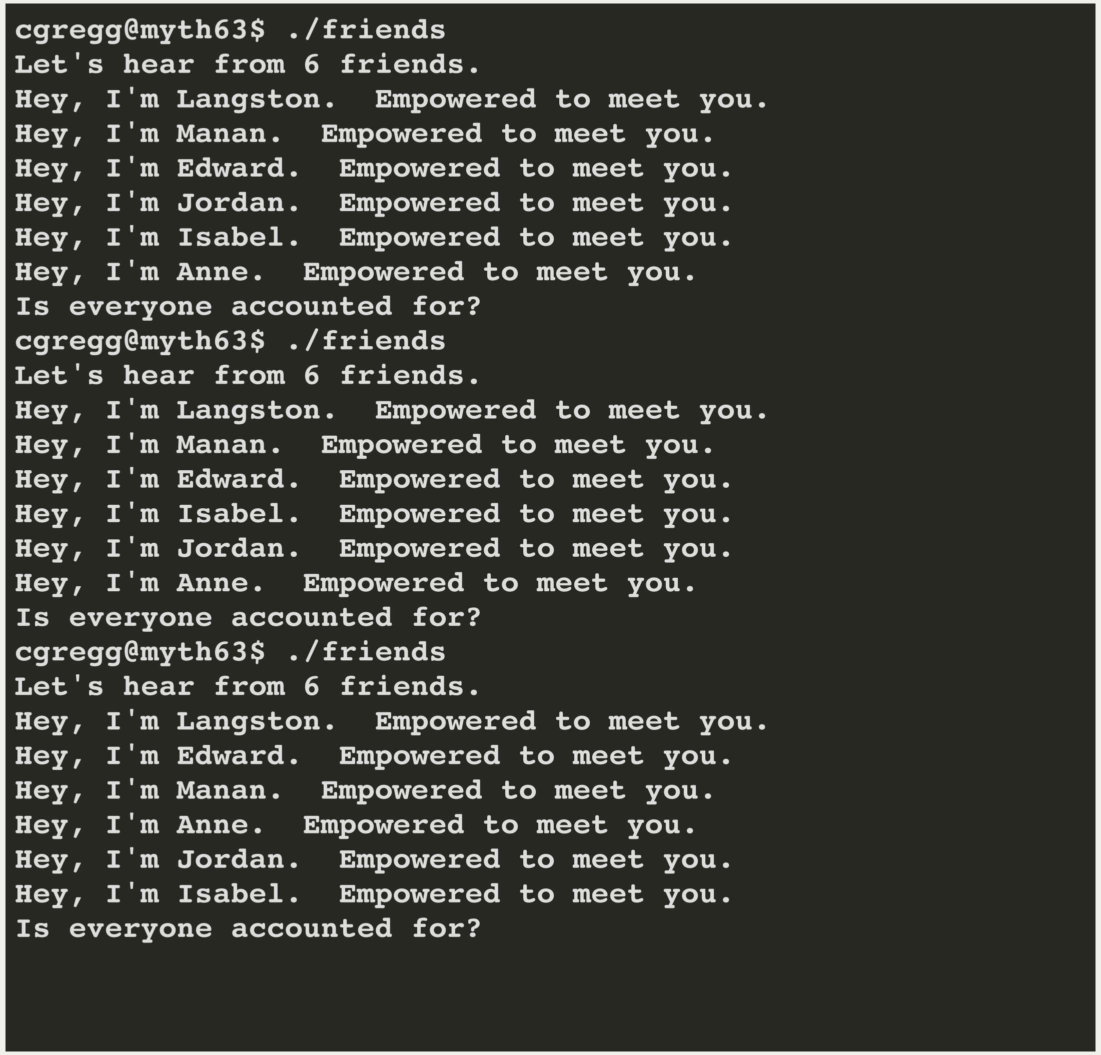
```c
static const char *kFriends[] = {
  "Langston", "Manan", "Edward", "Jordan", "Isabel", "Anne",
  "Imaginary"
};

static const size_t kNumFriends = sizeof(kFriends)/sizeof(kFriends[0]) - 1; // count excludes imaginary friend!

static void *meetup(void *args) {
  const char *name = args; // note that we get the name directly instead of through an index
  printf("Hey, I'm %s.  Empowered to meet you.\n", name);
  return NULL;
}

int main() {
  printf("%zu friends meet.\n", kNumFriends);
  pthread_t friends[kNumFriends];
  for (size_t i = 0; i < kNumFriends; i++)
    pthread_create(&friends[i], NULL, meetup, (void *) kFriends[i]); // this line now provides a variable that won't change
  for (size_t i = 0; i < kNumFriends; i++)
    pthread_join(friends[i], NULL);
  printf("All friends are real!\n");
  return 0;
}
```

Lecture 09: Introduction to Threads

https://cplayground.com/embed?p=fox-dugong-gnu

Lecture 09: Introduction to Threads

- Here are a few test runs just so you see that it's fixed. Race conditions are often quite complicated, and avoiding them won't always be this trivial.



```
cgregg@myth63$ ./friends
Let's hear from 6 friends.
Hey, I'm Langston.  Empowered to meet you.
Hey, I'm Manan.  Empowered to meet you.
Hey, I'm Edward.  Empowered to meet you.
Hey, I'm Jordan.  Empowered to meet you.
Hey, I'm Isabel.  Empowered to meet you.
Hey, I'm Anne.  Empowered to meet you.
Is everyone accounted for?
cgregg@myth63$ ./friends
Let's hear from 6 friends.
Hey, I'm Langston.  Empowered to meet you.
Hey, I'm Manan.  Empowered to meet you.
Hey, I'm Edward.  Empowered to meet you.
Hey, I'm Isabel.  Empowered to meet you.
Hey, I'm Jordan.  Empowered to meet you.
Hey, I'm Anne.  Empowered to meet you.
Is everyone accounted for?
cgregg@myth63$ ./friends
Let's hear from 6 friends.
Hey, I'm Langston.  Empowered to meet you.
Hey, I'm Edward.  Empowered to meet you.
Hey, I'm Manan.  Empowered to meet you.
Hey, I'm Anne.  Empowered to meet you.
Hey, I'm Jordan.  Empowered to meet you.
Hey, I'm Isabel.  Empowered to meet you.
Is everyone accounted for?
```

C++ Threads

```
static void greeting() {
    cout << oslock << "Hello, world!" << endl << osunlock;
}

static const size_t kNumFriends = 6;
int main(int argc, char *argv[]) {
  cout << "Let's hear from " << kNumFriends << " threads." << endl;

  thread friends[kNumFriends]; // declare array of empty thread handles

  // Spawn threads
  for (size_t i = 0; i < kNumFriends; i++) {
     friends[i] = thread(greeting);
  }

  // Wait for threads
  for (thread &f : friends) {
     f.join();
  }

  cout << "Everyone's said hello!" << endl;
  return 0;
}


```

WARNING: Thread Safety and Standard I/O

- **operator<<**, unlike **printf**, isn't thread-safe.
  - Jerry Cain has constructed custom stream manipulators called **oslock** and **osunlock** that can be used to acquire and release exclusive access to an **ostream**.
  - These manipulators—which we can use by **#include**-ing **"ostreamlock.h"**—can be used to ensure at most one thread has permission to write into a stream at any one time.

Thread-Level Parallelism

- Threads allow a process to parallelize a problem across multiple cores
- Consider a scenario where we want to process 250 images and have 10 cores
- Completion time is determined by the slowest thread, so we want them to have equal work
  - Static partitioning: just give each thread 25 of the images to process. Problem: what if some images take much longer than others?
  - Work queue: have each thread fetch the next unprocessed image
- Here's our first stab at a **main** function.

```
int main(int argc, const char *argv[]) {
  thread processors[10];
  size_t remainingImages = 250;
  for (size_t i = 0; i < 10; i++)
    processors[i] = thread(process, 101 + i, ref(remainingImages));
  for (thread& proc: processors) proc.join();
  cout << "Images done!" << endl;
  return 0;
}
```

Thread Function

- The **processor** thread routine accepts an id number (used for logging purposes) and a reference to the **remainingImages**.
- It continually checks **remainingImages** to see if any images remain, and if so, processes the image and sends a message to **cout**
- **processImage** execution time depends on the image.
- Note how we can declare a function that takes a **size_t** and a **size_t&** as arguments

```
static void process(size_t id, size_t& remainingImages) {
  while (remainingImages > 0) {
    processImage(remainingImages);
    remainingImages--;
    cout << oslock << "Thread#" << id << " processed an image (" << remainingImages
     << " remain)." << endl << osunlock;
  }
  cout << oslock << "Thread#" << id << " sees no remaining images and exits."
       << endl << osunlock;
}
```

- Is there anything wrong with this code?

Race Condition

- Presented below right is the abbreviated output of a **imagethreads** run.
- In its current state, the program suffers from a serious race condition.
- Why? Because **remainingImages > 0** test and **remainingImages--** aren't atomic
- If a thread evaluates **remainingImages > 0** to be **true** and commits to processing an image, the image may have been claimed by another thread.
- This is a concurrency problem!
- Can we solve this by decrementing immediately in the **while** loop?
- We can try, and we will likely get a solution that seems to work just fine.

```
static void process(size_t id, size_t& rem
  while (remainingImages > 0) {
    remainingImages--;
    processImage(remainingImages);
    cout << oslock << "Thread#" << id << "
     << " remain)." << endl << osunlock;
  }
  cout << oslock << "Thread#" << id << " s
      << endl << osunlock;
}
```

```
myth60 ~../cs110/cthreads -> ./imagethreads
Thread# 109 processed an image, 249 remain
Thread# 102 processed an image, 248 remain
Thread# 101 processed an image, 247 remain
Thread# 104 processed an image, 246 remain
Thread# 108 processed an image, 245 remain
Thread# 106 processed an image, 244 remain
// 241 lines removed for brevity
Thread# 110 processed an image, 3 remain
Thread# 103 processed an image, 2 remain
Thread# 105 processed an image, 1 remain
Thread# 108 processed an image, 0 remain
Thread# 105 processed an image, 18446744073709551615 remain
Thread# 109 processed an image, 18446744073709551614 remain
```

Race Condition

- But, we still have a *subtle* concurrency problem!
  - Between the line that performs the **while** loop calculation, and the decrement, another thread could also get into the **while** loop, and we would have a similar problem.
- **But, *it goes even deeper!*** *Even the decrement instruction itself could have a race condition!*
- C++ statements aren't inherently atomic. Virtually all C++ statements—even ones as simple as **remainingImages--**—compile to multiple assembly code instructions.
- Assembly code instructions are atomic, but C++ statements are not.
- **g++** on the myths compiles **remainingImages--** to five assembly code instructions, as with:

```
0x0000000000401a9b <+36>:    mov    -0x20(%rbp),%rax
0x0000000000401a9f <+40>:    mov    (%rax),%eax
0x0000000000401aa1 <+42>:    lea    -0x1(%rax),%edx
0x0000000000401aa4 <+45>:    mov    -0x20(%rbp),%rax
0x0000000000401aa8 <+49>:    mov    %edx,(%rax)
```

- The first two lines drill through the **remainingImages** reference to load a copy of the **remainingImages** held on **main**'s stack. The third line decrements that copy, and the last two write the decremented copy back to the **remainingImages** variable held on **main**'s stack.
- The ALU operates on registers, but registers are private to a core, so the variable needs to be loaded from and stored to memory.
  - Each thread makes a local copy of the variable before operating on it
  - What if multiple threads all load the variable at the same time: they all think there's only 128 images remaining and process 128 at the same time

Race Condition

- Solution? Make the test and decrement *atomic* with a *critical section*
- Atomicity: externally, the code has either executed or not; external observers do not see any intermediate states mid-execution
  - We need a way to ensure that we can have a critical section where only one thread can check the value of **remainingImages** at a time, and perform the decrement as well.
  - The way we will do it is through the idea of *mutual exclusion*, as described on the next slide.

Mutual Exclusion

- A ***mutex*** is a type used to enforce *mutual exclusion*, i.e., a critical section
- Mutexes are often called locks
  - To be very precise, mutexes are one kind of lock, there are others (read/write locks, reentrant locks, etc.), but we can just call them locks in this course, usually "lock" means "mutex"
- When a thread locks a mutex
  - If the lock is unlocked the thread takes the lock and continues execution
  - If the lock is locked, the thread blocks and waits until the lock is unlocked
  - If multiple threads are waiting for a lock they all wait until lock is unlocked, one receives lock
- When a thread unlocks a mutex
  - It continues normally; one waiting thread (if any) takes the lock and is scheduled to run
- This is a subset of the C++ mutex abstraction: nicely simple!

```
class mutex {
public:
  mutex();          // constructs the mutex to be in an unlocked state
  void lock();      // acquires the lock on the mutex, blocking until it's unlocked
  void unlock();    // releases the lock and wakes up another threads trying to lock it
};
```

Building a Critical Section with a Mutex

- **main** instantiates a mutex, which it passes (by reference!) to invocations of **process.**
- The **process** code uses this lock to protect **remainingImages**.
- Note we need to unlock on line 5 -- in complex code forgetting this is an easy bug

```
 1 static void process(size_t id, size_t& remainingImages, mutex& counterLock) {
 2   while (true) {
 3     counterLock.lock();
 4     if (remainingImages == 0) {
 5       counterLock.unlock();
 6       break;
 7     }
 8     processImage(remainingImages);
 9     remainingImages--;
10     cout << oslock << "Thread#" << id << " processed an image (" << remainingImages
11      << " remain)." << endl << osunlock;
12     counterLock.unlock();
13   }
14   cout << oslock << "Thread#" << id << " sees no remaining images and exits."
15   << endl << osunlock;
16 }
17 
18 int main(int argc, const char *argv[]) {
19   size_t remainingImages = 250;
20   mutex  counterLock;
21   thread processors[10];
22   for (size_t i = 0; i < 10; i++)
23     processors[i] = thread(process, 101 + i, ref(remainingImages), ref(counterLock));
24   for (thread& process: processes) process.join();
25   cout << "Done processing images!" << endl;
26   return 0;
27 }
```

Critical Sections Can Be a Bottleneck

- The way we've set it up, only one thread agent can process an image at a time!
  - Image processing is actually serialized
- We can do better: serialize deciding which image to process and parallelize the actual processing
- Keep your critical sections as small as possible!

```
static void process(size_t id, size_t& remainingImages, mutex& counterLock) {
  while (true) {
    size_t myImage;

    counterLock.lock();     // Start of critical section
    if (remainingImages == 0) {
      counterLock.unlock(); // Rather keep it here, easier to check
      break;
    } else {
      myImage = remainingImages;
      remainingImages--;
      counterLock.unlock(); // end of critical section

      processImage(myImage);
      cout << oslock << "Thread#" << id << " processed an image (" << remainingImages
      << " remain)." << endl << osunlock;
    }
  }
  cout << oslock << "Thread#" << id << " sees no remaining images and exits."
  << endl << osunlock;
}
```

Problems That Might Arise

- What if **processImage** can return an error?
  - E.g., what if we need to distinguish allocating an image and processing it
  - A thread can grab the image by decrementing **remainingImages** but if it fails there's no way for another thread to retry
  - Because these are threads, if one thread has a SEGV the whole process will fail
  - A more complex approach might be to maintain an actual queue of images and allow threads (in a critical section) to push things back into the queue
- What if image processing times are *highly* variable (e.g, one image takes 100x as long as the others)?
  - Might scan images to estimate execution time and try more intelligent scheduling
- What if there's a bug in your code, such that sometimes processImage randomly enters an infinite loop?
  - Need a way to reissue an image to an idle thread
  - An infinite loop of course shouldn't occur, but when we get to networks sometimes execution time can vary by 100x for reasons outside our control

Some Types of Mutexes

- Standard **mutex**: what we've seen
  - If a thread holding the lock tries to re-lock it, deadlock
- **recursive_mutex**
  - A thread can lock the mutex multiple times, and needs to unlock it the same number of times to release it to other threads
- **timed_mutex**
  - A thread can **try_lock_for** / **try_lock_until**: if time elapses, don't take lock
  - Deadlocks if same thread tries to lock multiple times, like standard mutex
- In this class, we'll focus on just regular **mutex**

How Do Mutexes Work?

- Something we've seen a few times is that you can't read and write a variable atomically
  - But a mutex does so! If the lock is unlocked, lock it
- How does this work with caches?
  - Each core has its own cache
  - Writes are typically write-back (write to higher cache level when line is evicted), not write-through (always write to main memory) for performance
  - Caches are *coherent* -- if one core writes to a cache line that is also in another core's cache, the other core's cache line is invalidated: this can become a performance problem
- Hardware provides atomic memory operations, such as compare and swap
  - cas old, new, addr
    - If addr == old, set addr to new
  - Use this as a single bit to see if the lock is held and if not, take it
  - If the lock is held already, then enqueue yourself (in a thread safe way) and tell kernel to sleep you
  - When a node unlocks, it clears the bit and wakes up a thread