CS110 Lecture 09: Threads

Principles of Computer Systems

Winter 2020

Stanford University

Computer Science Department

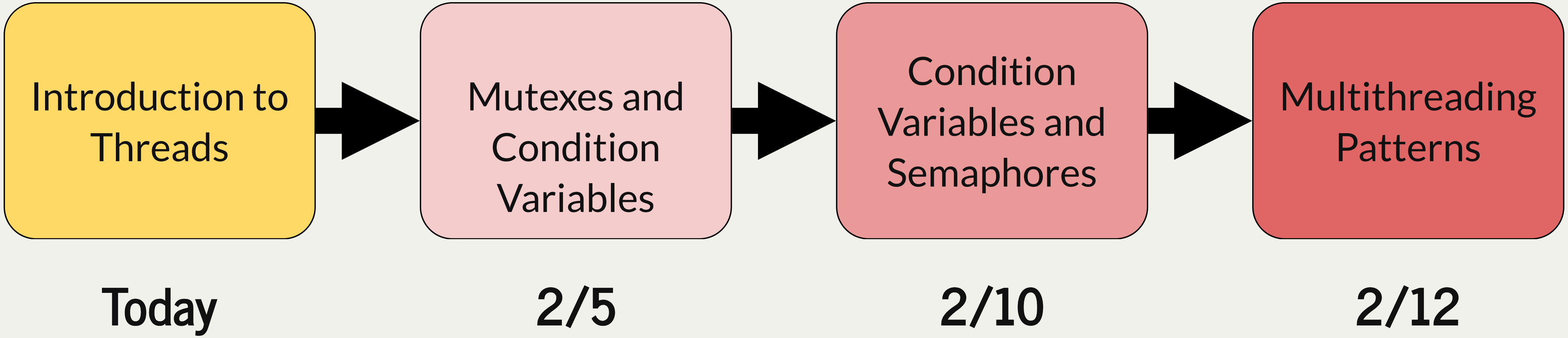
Instructors: Chris Gregg and

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

- More Practice: Race Conditions
- **Break:** Announcements
- Threads

Practice Problem 1

Consider this program and its execution. Assume that all processes run to completion, all system and **`printf`** calls succeed, and that all calls to **`printf`** are atomic. Assume nothing about scheduling or time slice durations.

```c
static void bat(int unused) {
  printf("pirate\n");
  exit(0);
}

int main(int argc, char *argv[]) {
  signal(SIGUSR1, bat);
  pid_t pid = fork();
  if (pid == 0) {
    printf("ghost\n");
    return 0;
  }
  kill(pid, SIGUSR1);
  printf("ninja\n"); return 0;
}
```

- For each of the five output orders, Place a **yes** if the text represents a possible output, and place a **no** otherwise.
  - ghost ninja pirate
  - pirate ninja
  - ninja ghost
  - ninja pirate ninja
  - ninja pirate ghost

Practice Problem 1

Consider this program and its execution. Assume that all processes run to completion, all system and **`printf`** calls succeed, and that all calls to **`printf`** are atomic. Assume nothing about scheduling or time slice durations.

```c
static void bat(int unused) {
  printf("pirate\n");
  exit(0);
}

int main(int argc, char *argv[]) {
  signal(SIGUSR1, bat);
  pid_t pid = fork();
  if (pid == 0) {
    printf("ghost\n");
    return 0;
  }
  kill(pid, SIGUSR1);
  printf("ninja\n"); return 0;
}
```

- For each of the five output orders, Place a **yes** if the text represents a possible output, and place a **no** otherwise.
  - ghost ninja pirate. **yes**
  - pirate ninja. **yes**
  - ninja ghost. **no**
  - ninja pirate ninja. **no**
  - ninja pirate ghost. **no**

Practice Problem 2

Consider this program and its execution. Assume that all processes run to completion, all system and **`printf`** calls succeed, and that all calls to **`printf`** are atomic. Assume nothing about scheduling or time slice durations.

```
int main(int argc, char *argv[]) {
    pid_t pid;
    int counter = 0;
    while (counter < 2) {
        pid = fork();
        if (pid > 0) break;
        counter++;
        printf("%d", counter);
    }
    if (counter > 0) printf("%d", counter);
    if (pid > 0) {
        waitpid(pid, NULL, 0);
        counter += 5;
        printf("%d", counter);
    }
    return 0;
}
```

- List all possible outputs
  - How many processes are created in total?
  - What number must be printed last?
  - What number must be printed first?
  - What number must be printed second-last?

Practice Problem 2

Consider this program and its execution. Assume that all processes run to completion, all system and **`printf`** calls succeed, and that all calls to **`printf`** are atomic. Assume nothing about scheduling or time slice durations.

```c
int main(int argc, char *argv[]) {
    pid_t pid;
    int counter = 0;
    while (counter < 2) {
        pid = fork();
        if (pid > 0) break;
        counter++;
        printf("%d", counter);
    }
    if (counter > 0) printf("%d", counter);
    if (pid > 0) {
        waitpid(pid, NULL, 0);
        counter += 5;
        printf("%d", counter);
    }
    return 0;
}
```

- List all possible outputs
  - How many processes are created in total? **3**
  - What number must be printed last?
  - What number must be printed first?
  - What number must be printed second-last?

Practice Problem 2

Consider this program and its execution. Assume that all processes run to completion, all system and **`printf`** calls succeed, and that all calls to **`printf`** are atomic. Assume nothing about scheduling or time slice durations.

```c
int main(int argc, char *argv[]) {
    pid_t pid;
    int counter = 0;
    while (counter < 2) {
        pid = fork();
        if (pid > 0) break;
        counter++;
        printf("%d", counter);
    }
    if (counter > 0) printf("%d", counter);
    if (pid > 0) {
        waitpid(pid, NULL, 0);
        counter += 5;
        printf("%d", counter);
    }
    return 0;
}
```

- List all possible outputs
  - What number must be printed last?
    - The last output is 5 (the grandparent that starts with counter 0 exits last, because it waits on its child)
  - What number must be printed first?
  - What number must be printed second-last?

Practice Problem 2

Consider this program and its execution. Assume that all processes run to completion, all system and **`printf`** calls succeed, and that all calls to **`printf`** are atomic. Assume nothing about scheduling or time slice durations.

```c
int main(int argc, char *argv[]) {
    pid_t pid;
    int counter = 0;
    while (counter < 2) {
        pid = fork();
        if (pid > 0) break;
        counter++;
        printf("%d", counter);
    }
    if (counter > 0) printf("%d", counter);
    if (pid > 0) {
        waitpid(pid, NULL, 0);
        counter += 5;
        printf("%d", counter);
    }
    return 0;
}
```

- List all possible outputs
  - What number must be printed last?
    - The last output is 5 (the grandparent that starts with counter 0 exits last, because it waits on its child)
  - What number must be printed first?
    - The parent that starts with counter 1 outputs first
  - What number must be printed second-last?

Practice Problem 2

Consider this program and its execution. Assume that all processes run to completion, all system and **`printf`** calls succeed, and that all calls to **`printf`** are atomic. Assume nothing about scheduling or time slice durations.

```c
int main(int argc, char *argv[]) {
    pid_t pid;
    int counter = 0;
    while (counter < 2) {
        pid = fork();
        if (pid > 0) break;
        counter++;
        printf("%d", counter);
    }
    if (counter > 0) printf("%d", counter);
    if (pid > 0) {
        waitpid(pid, NULL, 0);
        counter += 5;
        printf("%d", counter);
    }
    return 0;
}
```

- List all possible outputs
  - What number must be printed last?
    - The last output is 5 (the grandparent that starts with counter 0 exits last, because it waits on its child)
  - What number must be printed first?
    - The parent that starts with counter 1 outputs first
  - What number must be printed second-last?
    - The parent that starts with counter 1 exits second to last, after waiting for its child; the second-to-last output is 6

Practice Problem 2

Consider this program and its execution. Assume that all processes run to completion, all system and **`printf`** calls succeed, and that all calls to **`printf`** are atomic. Assume nothing about scheduling or time slice durations.

```c
int main(int argc, char *argv[]) {
    pid_t pid;
    int counter = 0;
    while (counter < 2) {
        pid = fork();
        if (pid > 0) break;
        counter++;
        printf("%d", counter);
    }
    if (counter > 0) printf("%d", counter);
    if (pid > 0) {
        waitpid(pid, NULL, 0);
        counter += 5;
        printf("%d", counter);
    }
    return 0;
}
```

- List all possible outputs
  - Possible Output 1: 112265
  - Possible Output 2: 121265
  - Possible Output 3: 122165

Practice Problem 2

Consider this program and its execution. Assume that all processes run to completion, all system and **`printf`** calls succeed, and that all calls to **`printf`** are atomic. Assume nothing about scheduling or time slice durations.

```c
int main(int argc, char *argv[]) {
    pid_t pid;
    int counter = 0;
    while (counter < 2) {
        pid = fork();
        if (pid > 0) break;
        counter++;
        printf("%d", counter);
    }
    if (counter > 0) printf("%d", counter);
    if (pid > 0) {
        waitpid(pid, NULL, 0);
        counter += 5;
        printf("%d", counter);
    }
    return 0;
}
```

- List all possible outputs
    - Possible Output 1: 112265
    - Possible Output 2: 121265
    - Possible Output 3: 122165
- The second parent (1) output and the two child (2) outputs are up to the scheduler

Practice Problem 2

Consider this program and its execution. Assume that all processes run to completion, all system and **`printf`** calls succeed, and that all calls to **`printf`** are atomic. Assume nothing about scheduling or time slice durations.

```c
int main(int argc, char *argv[]) {
    pid_t pid;
    int counter = 0;
    while (counter < 2) {
        pid = fork();
        if (pid > 0) break;
        counter++;
        printf("%d", counter);
    }
    if (counter > 0) printf("%d", counter);
    if (pid > 0) {
        waitpid(pid, NULL, 0);
        counter += 5;
        printf("%d", counter);
    }
    return 0;
}
```

- List all possible outputs
  - Possible Output 1: 112265
  - Possible Output 2: 121265
  - Possible Output 3: 122165
- If the **>** of the **`counter > 0`** test is changed to a **>=**, then **`counter`** values of zeroes would be included in each possible output. How many different outputs are now possible? (No need to list the outputs—just present the number.)

Practice Problem 2

Consider this program and its execution. Assume that all processes run to completion, all system and **`printf`** calls succeed, and that all calls to **`printf`** are atomic. Assume nothing about scheduling or time slice durations.

```c
int main(int argc, char *argv[]) {
    pid_t pid;
    int counter = 0;
    while (counter < 2) {
        pid = fork();
        if (pid > 0) break;
        counter++;
        printf("%d", counter);
    }
    if (counter > 0) printf("%d", counter);
    if (pid > 0) {
        waitpid(pid, NULL, 0);
        counter += 5;
        printf("%d", counter);
    }
    return 0;
}
```

- List all possible outputs
  - Possible Output 1: 112265
  - Possible Output 2: 121265
  - Possible Output 3: 122165
- If the **>** of the **`counter > 0`** test is changed to a **>=**, then **`counter`** values of zeroes would be included in each possible output. How many different outputs are now possible? (No need to list the outputs—just present the number.) **18 - 6x the original**

Practice Problem 3

Consider the following program. Assume that each call to **`printf`** flushes its output to the console in full, and further assume that none of the system calls fail in any unpredictable way.

```
static pid_t pid; // necessarily global so handler1 has access
static int counter = 0;
static void handler1(int unused) {
        counter++;
        printf("counter = %d\n", counter);
        kill(pid, SIGUSR1);
}
static void handler2(int unused) {
        counter += 10;
        printf("counter = %d\n", counter);
        exit(0);
}
int main(int argc, char *argv[]) {
        signal(SIGUSR1, handler1);
        if ((pid = fork()) == 0) {
                signal(SIGUSR1, handler2);
                kill(getppid(), SIGUSR1);
                while (true) {}
        }
        if (waitpid(-1, NULL, 0) > 0) {
                counter += 1000;
                printf("counter = %d\n", counter);
        }
        return 0;
}
```

- What is the output of the program?

Practice Problem 3

Consider the following program. Assume that each call to **`printf`** flushes its output to the console in full, and further assume that none of the system calls fail in any unpredictable way.

```
static pid_t pid; // necessarily global so handler1 has access
static int counter = 0;
static void handler1(int unused) {
        counter++;
        printf("counter = %d\n", counter);
        kill(pid, SIGUSR1);
}
static void handler2(int unused) {
        counter += 10;
        printf("counter = %d\n", counter);
        exit(0);
}
int main(int argc, char *argv[]) {
        signal(SIGUSR1, handler1);
        if ((pid = fork()) == 0) {
                signal(SIGUSR1, handler2);
                kill(getppid(), SIGUSR1);
                while (true) {}
        }
        if (waitpid(-1, NULL, 0) > 0) {
                counter += 1000;
                printf("counter = %d\n", counter);
        }
        return 0;
}
```

- What is the output of the program?

```
counter = 1
counter = 10
counter = 1001
```

Practice Problem 3

Consider the following program. Assume that each call to **`printf`** flushes its output to the console in full, and further assume that none of the system calls fail in any unpredictable way.

```
static pid_t pid; // necessarily global so handler1 has access
static int counter = 0;
static void handler1(int unused) {
        counter++;
        printf("counter = %d\n", counter);
        kill(pid, SIGUSR1);
}
static void handler2(int unused) {
        counter += 10;
        printf("counter = %d\n", counter);
        exit(0);
}
int main(int argc, char *argv[]) {
        signal(SIGUSR1, handler1);
        if ((pid = fork()) == 0) {
                signal(SIGUSR1, handler2);
                kill(getppid(), SIGUSR1);
                while (true) {}
        }
        if (waitpid(-1, NULL, 0) > 0) {
                counter += 1000;
                printf("counter = %d\n", counter);
        }
        return 0;
}
```

- What are the two potential outputs of the above program if the **`while (true)`** loop is completely eliminated?

Practice Problem 3

Consider the following program. Assume that each call to **`printf`** flushes its output to the console in full, and further assume that none of the system calls fail in any unpredictable way.

```
static pid_t pid; // necessarily global so handler1 has access
static int counter = 0;
static void handler1(int unused) {
        counter++;
        printf("counter = %d\n", counter);
        kill(pid, SIGUSR1);
}
static void handler2(int unused) {
        counter += 10;
        printf("counter = %d\n", counter);
        exit(0);
}
int main(int argc, char *argv[]) {
        signal(SIGUSR1, handler1);
        if ((pid = fork()) == 0) {
                signal(SIGUSR1, handler2);
                kill(getppid(), SIGUSR1);
                while (true) {}
        }
        if (waitpid(-1, NULL, 0) > 0) {
                counter += 1000;
                printf("counter = %d\n", counter);
        }
        return 0;
}
```

- What are the two potential outputs of the above program if the **`while (true)`** loop is completely eliminated?
  - Previous output still possible via scheduling
  - Now, though, the child process could complete and exit normally before the parent process—via its **`handler1`** function— has the opportunity to signal the child. That would mean **`handler2`** wouldn't even execute.
  - So, another possible output would be:

```
counter = 1
counter = 1001
```

Practice Problem 3

Consider the following program. Assume that each call to **`printf`** flushes its output to the console in full, and further assume that none of the system calls fail in any unpredictable way.

```
static pid_t pid; // necessarily global so handler1 has access
static int counter = 0;
static void handler1(int unused) {
        counter++;
        printf("counter = %d\n", counter);
        kill(pid, SIGUSR1);
}
static void handler2(int unused) {
        counter += 10;
        printf("counter = %d\n", counter);
        exit(0);
}
int main(int argc, char *argv[]) {
        signal(SIGUSR1, handler1);
        if ((pid = fork()) == 0) {
                signal(SIGUSR1, handler2);
                kill(getppid(), SIGUSR1);
                while (true) {}
        }
        if (waitpid(-1, NULL, 0) > 0) {
                counter += 1000;
                printf("counter = %d\n", counter);
        }
        return 0;
}
```

- Now further assume the call to **`exit(0)`** has also been removed from the **`handler2`** function . Are there any other potential program outputs? If not, explain why. If so, what are they?
  - No other potential outputs, because:
    - **`counter = 1`** is still printed exactly once, just in the parent, before the parent fires a **`SIGUSR1`** signal at the child
    - …

Practice Problem 3

Consider the following program. Assume that each call to **`printf`** flushes its output to the console in full, and further assume that none of the system calls fail in any unpredictable way.

```c
static pid_t pid; // necessarily global so handler1 has access
static int counter = 0;
static void handler1(int unused) {
        counter++;
        printf("counter = %d\n", counter);
        kill(pid, SIGUSR1);
}
static void handler2(int unused) {
        counter += 10;
        printf("counter = %d\n", counter);
        exit(0);
}
int main(int argc, char *argv[]) {
        signal(SIGUSR1, handler1);
        if ((pid = fork()) == 0) {
                signal(SIGUSR1, handler2);
                kill(getppid(), SIGUSR1);
                while (true) {}
        }
        if (waitpid(-1, NULL, 0) > 0) {
                counter += 1000;
                printf("counter = %d\n", counter);
        }
        return 0;
}
```

- Now further assume the call to **`exit(0)`** has also been removed from the **`handler2`** function . Are there any other potential program outputs? If not, explain why. If so, what are they?
    - No other potential outputs, because:
        - …
        - **`counter = 10`** is potentially printed if the child is still running at the time the parent fires that **`SIGUSR1`** signal at it. The **`10`** can only appear after the **`1`**, and if it appears, it must appear before the **`1001`**.
        - **. . .**

Practice Problem 3

Consider the following program. Assume that each call to **`printf`** flushes its output to the console in full, and further assume that none of the system calls fail in any unpredictable way.

```
static pid_t pid; // necessarily global so handler1 has access
static int counter = 0;
static void handler1(int unused) {
        counter++;
        printf("counter = %d\n", counter);
        kill(pid, SIGUSR1);
}
static void handler2(int unused) {
        counter += 10;
        printf("counter = %d\n", counter);
        exit(0);
}
int main(int argc, char *argv[]) {
        signal(SIGUSR1, handler1);
        if ((pid = fork()) == 0) {
                signal(SIGUSR1, handler2);
                kill(getppid(), SIGUSR1);
                while (true) {}
        }
        if (waitpid(-1, NULL, 0) > 0) {
                counter += 1000;
                printf("counter = %d\n", counter);
        }
        return 0;
}
```

- Now further assume the call to **`exit(0)`** has also been removed from the **`handler2`** function . Are there any other potential program outputs? If not, explain why. If so, what are they?
  - No other potential outputs, because:
    - …
    - **`counter = 1001`** is always printed last, after the child process exits. It's possible that the child existed at the time the parent signaled it to inspire **`handler2`** to print a **`10`**, but that would happen before the **`1001`** is printed.

Plan For Today

- More Practice: Race Conditions
- **Break:** Announcements
- Threads

Announcements

- Assignment 4 released later today - Stanford Shell
- Midterm Next Friday
  - Please notify us of any OAE accommodations ASAP
  - More midterm logistics on Wednesday

Assignment 4: Stanford Shell

- Assignment 4 is a comprehensive test of your abilities to **`fork`** / **`execvp`** child processes and manage them through the use of signal handlers. It also tests your ability to use pipes.
- You will be writing a shell (demo: **`assign4/samples/stsh_soln`**)
  - The shell will keep a list of all background processes, and it will have some standard shell abilities:
    - you can quit the shell (using **`quit`** or **`exit`**)
    - you can bring them to the front (using **`fg`**)
    - you can continue a background job (using **`bg`**)
    - you can kill a set of processes in a pipeline (using **`slay`**) (this will entail learning about process groups)
    - you can stop a process (using **`halt`**)
    - you can continue a process (using **`cont`**)
    - you can get a list of jobs (using **`jobs`**)
  - You are responsible for creating pipelines that enable you to send output between programs, e.g.,
    - **`ls | grep stsh | cut -d- -f2`**
    - **`sort < stsh.cc | wc > stsh-wc.txt`**
  - You will also be handing off terminal control to foreground processes, which is new

Assignment 4: Stanford Shell

- Assignment 4 contains a lot of moving parts!
- Read through all the header files!
- You will only need to modify **`stsh.cc`**
- You can test your shell programmatically with **`samples/stsh-driver`**
- One of the more difficult parts of the assignment is making sure you are keeping track of all the processes you've launched correctly. This involves careful use of a **`SIGCHLD`** handler.
  - You will also need to use a handler to capture **`SIGTSTP`** and **`SIGINT`** to capture ctrl-Z and ctrl-C, respectively (notice that these don't affect your regular shell -- they shouldn't affect your shell, either).
- Another tricky part of the assignment is with the piping between processes. It takes time to understand what we are requiring you to accomplish
- There is a very good list of milestones in the assignment -- try to accomplish regular milestones, and you should stay on track.

Plan For Today

- More Practice: Race Conditions
- **Break:** Announcements
- 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)

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 (size_t i = 0; i < kNumFriends; i++) {
     friends[i].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;
}
```

- Discuss with your neighbor -- what's 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!
- 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

```
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
```

Why Test and Decrement Is REALLY NOT Thread-Safe

- 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

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     agents[i] = thread(process, 101 + i, ref(remainingImages), ref(counterLock));
24   for (thread& agent: agents) agent.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

Questions about threads, mutexes, race conditions, or critical sections?