Lecture 09: Introduction to Threads

Principles of Computer Systems

Spring 2019

Stanford University

Computer Science Department

Instructors: Chris Gregg and Phil Levis

PDF of this presentation

Practice Midterm Problem 2 (we saw Problem 1 last lecture)

- Let's go through another example that is the kind of signals problem you may see on the midterm exam.
  - 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

Practice Midterm Problem 2

- Let's go through another example that is the kind of signals problem you may see on the midterm exam.
  - 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 Midterm Problem 2

- Let's go through another example that is the kind of signals problem you may see on the midterm exam.
  - 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 outputs now (6 x the first number)

Practice Midterm 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 (e.g. fork never fails, and **`waitpid`** only returns -1 because there aren't any child processes at the moment it decides on its return value).

```c
static pid_t pid; // necessarily global so handler1 has
                  // access to it
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?
- What are the two potential outputs of the above program if the **`while (true)`** loop is completely eliminated?
- Describe how the two processes would need to be scheduled in order for each of the two outputs to be presented.
- 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?

Practice Midterm 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 (e.g. fork never fails, and **`waitpid`** only returns -1 because there aren't any child processes at the moment it decides on its return value).

```c
static pid_t pid; // necessarily global so handler1 has
                  // access to it
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
```

- This is the only possible output based on the program's logic

Practice Midterm 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 (e.g. fork never fails, and **`waitpid`** only returns -1 because there aren't any child processes at the moment it decides on its return value).

```c
static pid_t pid; // necessarily global so handler1 has
                  // access to it
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?
  - The output from before (the **1** / **10** / **1001**) output is still possible, because the child process can be swapped out just after the **`kill(getppid(), SIGUSR1)`** call, and effectively emulate the stall that came with the **`while (true)`** loop when it was present.
  - 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, and we wouldn't expect to see **`counter = 10`**. (Note that the child process's call to **`waitpid`** returns **`-1`**, since it itself has no grandchild processes of its own).

- So, another possible output would be:

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

Practice Midterm 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 (e.g. fork never fails, and **`waitpid`** only returns -1 because there aren't any child processes at the moment it decides on its return value).

```c
static pid_t pid; // necessarily global so handler1 has
                  // access to it
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 (which may or may not have run to completion).
  - **`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`**.
  - **`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.

- Note that the child process either prints nothing at all, or it prints a **`10`**. The child process can never print **`1001`**, because its **`waitpid`** call would return **`-1`** and circumvent the code capable of printing the **`1001`**.

Lecture 09: Introduction to Threads

- A **thread** is an independent execution sequence within a single process.
    - Operating systems and programming languages generally allow processes to run two or more functions simultaneously via threading.
    - The stack segment is subdivided into multiple miniature stacks, one for each thread.
    - The thread manager time slices and switches between threads in much the same way that the OS scheduler switches between processes.
        - In fact, threads are often called **lightweight processes**.
        - Each thread maintains its own stack, but all threads share the same text, data, and heap segments.
            - Pro: it's easier to support communication between threads, because they run in the same virtual address space.
            - Con: there's no memory protection, since virtual address space is shared. Race conditions and deadlock threats need to be mitigated, and debugging can be difficult. Many bugs are hard to reproduce, since thread scheduling isn't predictable.
            - Pro and con: Multiple threads can access the same globals.
            - Pro and con: One thread can share its stack space (via pointers) with others.

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

- 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):

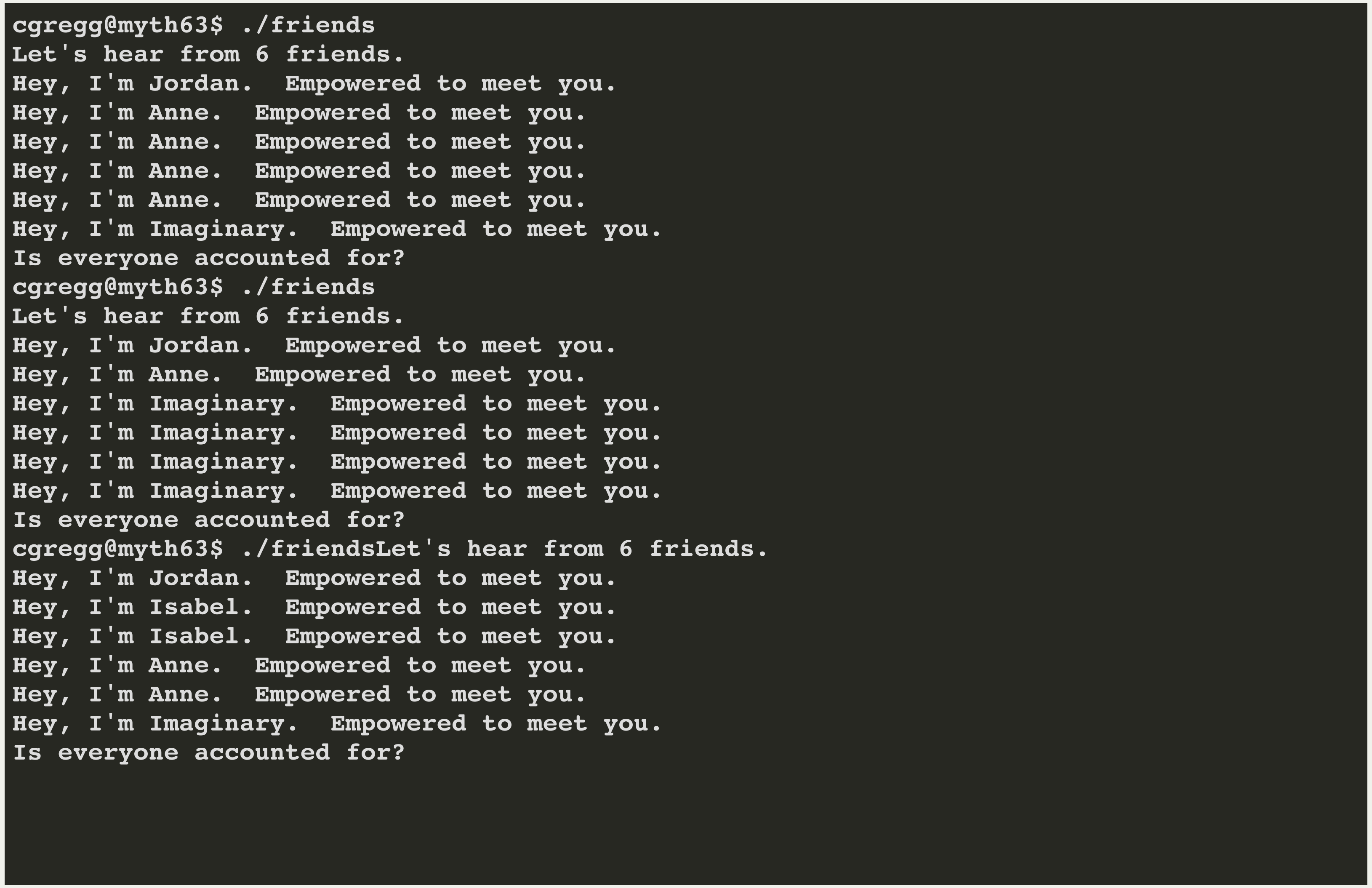
```c
static const char *kFriends[] = {
    "Langston", "Manan", "Edward", "Jordan", "Isabel", "Anne",
    "Imaginary"
};

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(&kFriends[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.

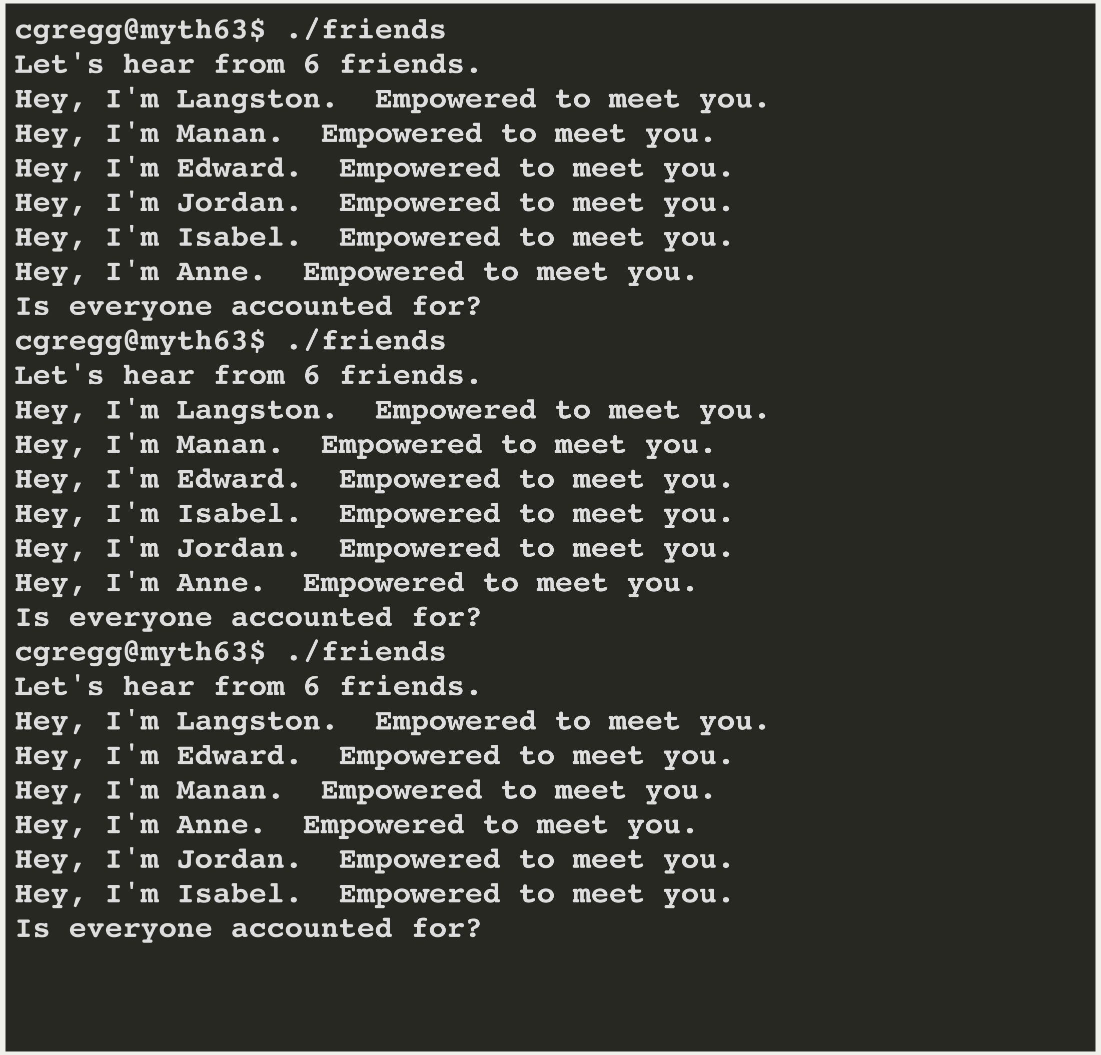
```c
static const char *kFriends[] = {
  "Langston", "Manan", "Edward", "Jordan", "Isabel", "Anne",
  "Imaginary"
};

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

static void *meetup(void *args) {
  const char *name = args;
  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 is different than before, too
  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?
```