Lecture 08: 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 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 the 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 global variables.
    - Pro **and** con: Each thread can share its stack space (via pointers) with its peer threads.

Lecture 08: 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 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 cplayground program illustrating how `pthreads` work:

```
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 08: 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 thread 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. 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 pass through `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 the thread manager's attention, and we have very little control over what choices it makes when deciding which 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 08: 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 extroverts get their day.

```
static const char *kExtroverts[] = {
  "Frank", "Jon", "Lauren", "Marco", "Julie", "Patty",
  "Tagalong Introvert Jerry"
};
static const size_t kNumExtroverts = sizeof(kExtroverts)/sizeof(kExtroverts[0]) - 1; // count excludes impostor!

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

int main() {
  printf("Let's hear from %zu extroverts.\n", kNumExtroverts);
  pthread_t extroverts[kNumExtroverts];
  for (size_t i = 0; i < kNumExtroverts; i++)
    pthread_create(&extroverts[i], NULL, recharge, &i);
  for (size_t j = 0; j < kNumExtroverts; j++)
    pthread_join(extroverts[j], NULL);
  printf("Everyone's recharged!\n");
  return 0;
}
```

Lecture 08: Introduction to Threads

- Here are a few test runs to illustrate that our program on the previous slide is fairly broken.

```
poohbear@myth62:$ ./confused-extroverts
Let's hear from 6 extroverts.
Hey, I'm Jon.  Empowered to meet you.
Hey, I'm Lauren.  Empowered to meet you.
Hey, I'm Julie.  Empowered to meet you.
Hey, I'm Julie.  Empowered to meet you.
Hey, I'm Patty.  Empowered to meet you.
Hey, I'm Tagalong Introvert Jerry.  Empowered to meet you.
Everyone's recharged!
poohbear@myth62:$ ./confused-extroverts
Let's hear from 6 extroverts.
Hey, I'm Marco.  Empowered to meet you.
Hey, I'm Marco.  Empowered to meet you.
Hey, I'm Julie.  Empowered to meet you.
Hey, I'm Julie.  Empowered to meet you.
Hey, I'm Julie.  Empowered to meet you.
Hey, I'm Tagalong Introvert Jerry.  Empowered to meet you.
Everyone's recharged!
poohbear@myth62:$ ./confused-extroverts
Let's hear from 6 extroverts.
Hey, I'm Marco.  Empowered to meet you.
Hey, I'm Tagalong Introvert Jerry Cain.  Empowered to meet you.
Hey, I'm Tagalong Introvert Jerry Cain.  Empowered to meet you.
Hey, I'm Tagalong Introvert Jerry Cain.  Empowered to meet you.
Hey, I'm Tagalong Introvert Jerry Cain.  Empowered to meet you.
Hey, I'm Tagalong Introvert Jerry Cain.  Empowered to meet you.
Everyone's recharged!
poohbear@myth62:$
```

Lecture 08: Introduction to Threads

- Clearly something is wrong, but why?
  - Note that `recharge` references its incoming parameter in this version, and that `pthread_create` accepts the location 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 `recharge` threads. `*(size_t *)args` takes a snapshot of whatever `i` happens to contain at the time it's evaluated.
    - Often, the majority of the `recharge` threads only execute after `main` has worked through its first `for` loop. The space at `&i` is left with a 6, and that's why Tagalong Introvert is printed so often.
  - This is another example of a race condition, and it represents a common problem that comes up when multiple threads share access to the same data.

Lecture 08: 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 `recharge`. The strings themselves are constants.
  - Full program illustrating the fix can be found right here.

```
static const char *kExtroverts[] = {
  "Frank", "Jon", "Lauren", "Marco", "Julie", "Patty",
  "Tagalong Introvert Jerry"
};
static const size_t kNumExtroverts = sizeof(kExtroverts)/sizeof(kExtroverts[0]) - 1; // count excludes impostor!

static void *recharge(void *args) {
  const char *name = args; // this line is different than before
  printf("Hey, I'm %s.  Empowered to meet you.\n", name);
  return NULL;
}

int main() {
  printf("Let's hear from %zu extroverts.\n", kNumExtroverts);
  pthread_t extroverts[kNumExtroverts];
  for (size_t i = 0; i < kNumExtroverts; i++)
    pthread_create(&extroverts[i], NULL, recharge, (void *) kExtroverts[i]); // this line is different than before as well
  for (size_t j = 0; j < kNumExtroverts; j++)
    pthread_join(extroverts[j], NULL);
  printf("Everyone's recharged!\n");
  return 0;
}
```

Lecture 08: 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.)

```
poohbear@myth62:$ ./extroverts
Let's hear from 6 extroverts.
Hey, I'm Marco.  Empowered to meet you.
Hey, I'm Jon.  Empowered to meet you.
Hey, I'm Lauren.  Empowered to meet you.
Hey, I'm Julie.  Empowered to meet you.
Hey, I'm Frank.  Empowered to meet you.
Hey, I'm Patty.  Empowered to meet you.
Everyone's recharged!
poohbear@myth62:$ ./extroverts
Let's hear from 6 extroverts.
Hey, I'm Frank.  Empowered to meet you.
Hey, I'm Lauren.  Empowered to meet you.
Hey, I'm Jon.  Empowered to meet you.
Hey, I'm Julie.  Empowered to meet you.
Hey, I'm Marco.  Empowered to meet you.
Hey, I'm Patty.  Empowered to meet you.
Everyone's recharged!
poohbear@myth62:$ ./extroverts
Let's hear from 6 extroverts.
Hey, I'm Jon.  Empowered to meet you.
Hey, I'm Lauren.  Empowered to meet you.
Hey, I'm Frank.  Empowered to meet you.
Hey, I'm Julie.  Empowered to meet you.
Hey, I'm Marco.  Empowered to meet you.
Hey, I'm Patty.  Empowered to meet you.
Everyone's recharged!
poohbear@myth62:$
```