Lecture 10: Introduction to Threads

Principles of Computer Systems
Stanford University, Dept. Of Computer Science
Lecturer: Roslyn Michelle Cyrus
Content adapted from material by Jerry Cain.
Announcements

● Feel free to schedule one-on-one office hours with me. I’m here to help!

● Gradebook has two sections: the score and the feedback.
  ○ Be sure to look at the notes to get feedback on why your code may not have been solid.

● Updated assignment and lecture schedule. See website

● Courseload
  ○ Less assignments, but we still cover the same amount of material as the regular 10-week quarter

● C++
  ○ As mentioned at the beginning of the quarter, C++ is a required language for this class so if you’re rusty on it, it is your responsibility to get up to speed. Also, you didn’t need to know Python at all for Assignment 2. We explained what it was doing in the spec.
  ○ You aren’t expected to know everything about C++, however. So there will be times when you’ll need to research how to use a certain feature. This is also by design; learning how to use a particular function and read the man pages are key parts of software design!

● Assignment walkthroughs: We do our best to do this for you because we want to make your lives easier, but we can’t promise that we will always have time to do it (and if we do, it may not always be as soon as you want it).

● Material recap: use the lecture slides to remember what we’ve learned.
Lecture Overview

● Virtual memory
  ○ We'll see how each process can operate as if it owns all of memory, even though it clearly doesn't.

● Threads
  ○ We'll discuss multithreading and the types of concurrency issues that sometimes present themselves because multiple threads are running within a process at the same time. We'll do this by working through a collection of examples in C and then transition over to C++, which in my opinion provides much better support for threading and concurrency than pure C does.
Midterm Information

- It will focus mostly on multiprocessing (I’ll aim for roughly 70%), with less on filesystems (~20%) and even less on this week’s material (~10% virtual memory and threading).
- It will have less freeform code writing than the practice midterms and more “tell me about this code” types of questions. But one freeform code problem is likely.
- It will have short answer questions
- It may use diagrams like the ones shown in class to test certain concepts
- I plan on writing it to be completed in 1.5 hours, but you will have 2 hours to finish it. This should help with time pressure!
- We’ll have a midterm review on Friday, taught by our wonderful lead CA Garrick.
Accessing Code Examples

- Today's lecture examples reside within:
  `/usr/class/cs110/lecture-examples/threads-c`.
  - First `ssh` into a myth machine (ssh `yourusername@myth.stanford.edu`). When prompted for your password, it is normal for the text not to appear as you enter your password. Once logged onto a myth machine, `cd` into the above directory.
  - To get started, type:
    `git clone /usr/class/cs110/lecture-examples cs110-lecture-examples` at the command prompt to create a local copy of the master.
  - Each time I mention there are new examples (or whenever you think to), descend into your local copy and type `git pull`. Doing so will update your local copy to match whatever the master has become.
Virtual Memory

- Main memory (RAM) is organized as an array of contiguous byte-size cells.
  - Each byte has a unique **physical address**
  - Addresses start at number 0
  - **Physical addressing** is the natural way to access memory.
  - Early PCs used physical addressing but modern processors use **virtual addressing**.
Virtual Memory (continued)

- With virtual addressing, the CPU accesses main memory by generating a virtual address, which is converted to the appropriate physical address before being sent to main memory.
  - To do this conversion, the CPU uses dedicated hardware on the CPU chip called the memory management unit (MMU), which uses a lookup table stored in main memory.
Virtual Memory: Address Spaces

- An **address space** is an ordered set of nonnegative integer addresses.
  - If the integers in the address space are consecutive, it is called a **linear address space**.
  - In a system with virtual memory, the CPU generates virtual addresses from an address space of $N = 2^n$ addresses called the **virtual address space**.
  - The size of an address space is characterized by the number of bits that are needed to represent the largest address.
  - Modern systems typically support a 32-bit or 64-bit virtual address space.
  - On a 64-bit architecture, every process thinks it has exclusive access to addresses 0x00000000 to 0xFFFFFFFF.
  - A system also has a **physical address space** that corresponds to the $M$ bytes of physical memory in the system.
  - Each byte of main memory has a virtual address chosen from the virtual address space and a physical address chosen from the physical address space.
Virtual Memory: Caching

- Virtual memory is organized as an array of N contiguous byte-size cells stored on disk.
  - Each byte has a unique virtual address that is an index into the array
  - The contents of the array on disk are cached in main memory
  - The data on disk is partitioned into blocks called physical pages (page frames)
  - Virtual memory is partitioned into fixed-size blocked called virtual pages
    - Virtual pages can be in one of three states:
      - **Unallocated**: pages that haven’t been allocated or created yet. These blocks don’t have any associated data and thus don’t take up space on disk.
      - **Cached**: allocated pages that are currently cached in physical memory
      - **Uncached**: allocated pages that are not cached in physical memory
  - DRAM is a cache that the VM system uses to cache virtual pages in main memory.
Virtual Memory: Caching (continued)

**Physical memory** is the RAM, which is the first memory used when the computer requires memory usage (e.g. opening a document).

**Virtual memory** is stored on the hard drive. Virtual memory is used when the RAM is filled. Virtual memory is slower than physical memory, so it can decrease the performance of applications.
A page table is a data structure stored in physical memory that maps virtual pages to physical pages.

The OS maintains the contents of the page table and transfers pages back and forth between disk and DRAM.

A page table is an array of page table entries, where each entry stores whether the virtual page is cached in DRAM.
Virtual Memory: Memory Mgmt.

- Virtual memory simplifies linking
  - A separate address space allows each process to use the same basic format for its memory image, regardless of where the code and data actually reside in physical memory.
  - Recall that every process on a given Linux system has a similar memory format. For 64-bit address spaces, the code segment always starts at virtual address 0x400000.
  - Such uniformity greatly simplifies the design and implementation of linkers, allowing them to produce fully linked executables that are independent of the ultimate location of the code and data in physical memory.

- Virtual memory simplifies loading
  - Easy to load executable and shared object files into memory, on demand. = memory mapping (mmap)
Virtual Memory: Memory Mgmt.

- Virtual memory simplifies sharing
  - In general, each process has its own private code, data, heap, and stack areas that are not shared with any other process.
  - Sometimes processes share code and data. For example, kernel code and standard C library functions like `printf`. The OS can have multiple processes share a single copy of this code by mapping the appropriate virtual pages in different processes to the same physical pages.

- Virtual memory simplifies memory allocation
  - When requesting additional heap space, the OS allocates an appropriate number of contiguous virtual memory pages and they are each mapped to arbitrary physical pages located anywhere in physical memory.

Reading: B&O’s Virtual Memory chapter
Virtual Memory: Memory Mgmt.

- Virtual memory provides memory protection
  - A user process should not be allowed to modify its read-only code section,
  - Shouldn’t be allowed to read or modify any code and data structures in the kernel.
  - Shouldn’t be able to read/write the private memory of other processes
  - Shouldn’t be able to modify any virtual pages that are shared with other processes (unless all parties explicitly allow it).

Reading: B&O’s Virtual Memory chapter
Virtual Memory: Memory Mgmt.

- What are some situations in which the same virtual address in multiple processes map to the same physical address in main memory?
Virtual Memory: Memory Mgmt.

- What are some situations in which the same virtual address in multiple processes map to the same physical address in main memory?
  - Sharing content in the memory mapped region
  - Forked processes explicitly want to share data among themselves (mmap)
  - Shared code segments

Reading: B&O's Virtual Memory chapter
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.
- Instead of being isolated, threads share most resources with other threads of the same process. Each thread maintains its own stack and registers, but all threads share the same text, data, and heap segments. Thus they share the same globals, file descriptor table, and more.
Threads (continued)

Reading: B&O’s Concurrent Programming with Threads chapter

- **Regular Process (main thread)**
  - kernel virtual memory
    - (Private) code, data, heap, stack
  - user stack
    - created at runtime; grows downward
  - memory-mapped region
    - for shared libraries (such as libc.so)
  - runtime heap
    - (via malloc) grows upward
  - read/write data segment
    - (.data, .bss)
  - read-only code segment
    - (.init, .text, .rodata)

- **Process with 3 threads**
  - kernel virtual memory
    - (Private) code, data, heap, stack
  - user stack (main thread)
    - created at runtime; grows downward
  - user stack (peer thread 1)
  - user stack (peer thread 2)
  - memory-mapped region
    - for shared libraries (such as libc.so)
  - runtime heap
    - (via malloc) grows upward
  - read/write data segment
    - (.data, .bss)
  - read-only code segment
    - (.init, .text, .rodata)
Threads (continued)

- Other differences between processes and threads:
  - Threads, unlike processes, are not organized in a rigid parent-child hierarchy.
    - The threads associated with a process form a **pool** of peers, independent of which threads were created by which other threads.
    - The main thread is distinguished from other threads only in the sense that it is always the first thread to run in the process.
    - Thus a thread can kill any of its peers or wait for any of its peers to terminate!
    - Each peer can read and write the same shared data.
Thus far in this class, our programs have consisted of a single thread per process. Threads are scheduled automatically by the kernel. Each thread has its own thread context, including a unique integer thread ID, stack, stack pointer, program counter, general-purpose registers, and condition codes. Each process begins life as a single thread called the main thread. The main thread can create a peer thread, and from this point in time the two threads run concurrently. Eventually, control passes to the peer thread via a context switch, either because the main thread executes a slow system call (e.g. read or sleep) or because it is interrupted by the system’s interval timer. The peer thread executes for a while before control passes back to the main thread, and so on.
Pros and cons of threads

- **Pro**: it's easier to support communication between threads, because they run in the same virtual address space.
- **Pro**: thread context switches are faster than process context switches since a thread context is much smaller than a process context.
- **Con**: there's no memory protection, since virtual address space is shared. Race conditions and deadlock threats need to be mitigated.
- **Con**: 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.
Threads vs. Processes

- Real-world example: Chrome vs. Firefox
  - **Chrome uses multiprocessing: each tab is a process**
    - Chrome is less likely to crash since if one tab (process) crashes, it won't crash the entire browser (but it would crash in Firefox).
  - **Firefox used to use multithreading: each tab is a thread**
    - Since multithreading is faster, Firefox is faster (Chrome is slower)
    - A couple years ago: “Chrome and Firefox now both support multithreading, but they do it in different ways. In Chrome, each and every tab you open gets its own content process. Ten tabs, 10 processes. One hundred tabs, 100 processes. This approach maximizes performance, but you pay a hefty penalty in memory consumption and (when applicable) battery life. Firefox doesn’t take this approach to the problem, but instead spins up to four content process threads by default. Your first 4 tabs each use those 4 processes, and additional tabs run using threads within those processes. Multiple tabs within a process share the browser engine that already exists in memory, instead of each creating their own.” [source]
Threads vs. Processes

Considerations in browser design: Speed, memory usage, battery/CPU usage, security, stability
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 switching to C++ threads): **`pthread_create`** and **`pthread_join`**.

```c
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);
int pthread_join(pthread_t thread, void **retval);
```
Threads (continued)

Here’s a very small program illustrating how pthreads work.

The program 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 C 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 the routine’s argument, passed to the thread routine as each thread is launched. In this case, there are no meaningful arguments, so we just pass in NULL.

```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;
}
```

myth60$ ./introverts
Let's hear from 6 introverts.
I recharge by spending time alone.
I recharge by spending time alone.
I recharge by spending time alone.
I recharge by spending time alone.
I recharge by spending time alone.
Everyone's recharged!
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 first argument to pthread_join takes the thread to wait for.
- 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.

```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;
}
```

myth60$ ./introverts
Let's hear from 6 introverts.
I recharge by spending time alone.
I recharge by spending time alone.
I recharge by spending time alone.
I recharge by spending time alone.
I recharge by spending time alone.
I recharge by spending time alone.
Everyone's recharged!
When you introduce concurrency, you be careful to avoid concurrency issues like race conditions and deadlock. Here's a slightly more involved program where friends meet up (live demo here). See the broken output on right.

```c
// confused-friends.c
static const char *kFriends[] = {
    "Jack", "Michaela", "Luis", "Richard", "Jordan", "Lisa", "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;
}
```

myth60$ ./confused-friends
Let's hear from 6 friends.
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.
Hey, I'm Imaginary. Empowered to meet you.
Hey, I'm Imaginary. Empowered to meet you.
Is everyone accounted for?
myth60$ ./confused-friends
Let's hear from 6 friends.
Hey, I'm Blanca. Empowered to meet you.
Hey, I'm Garrick. Empowered to meet you.
Hey, I'm Garrick. Empowered to meet you.
Hey, I'm Garrick. Empowered to meet you.
Hey, I'm Ryan. Empowered to meet you.
Hey, I'm Imaginary. Empowered to meet you.
Is everyone accounted for?
myth60$
Threads (continued)

When you introduce concurrency, you be careful to avoid concurrency issues like race conditions and deadlock. Here's a slightly more involved program where friends meet up (live demo here). See the broken output on right.

```
// confused-friends.c
static const char *kFriends[] = {
    "Jack", "Michaela", "Luis", "Richard", "Jordan", "Lisa", "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;
}
```

- Clearly something is wrong, but why?
- Note that meetup now references its incoming parameter and 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.
Threads (continued)

When you introduce concurrency, you be careful to avoid concurrency issues like race conditions and deadlock. Here's a slightly more involved program where friends meet up (live demo here). See the broken output on right.

```
// confused-friends.c
static const char *kFriends[] = {
    "Jack", "Michaela", "Luis", "Richard", "Jordan", "Lisa","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;
}
```

- 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.
Threads (continued)

When you introduce concurrency, you be careful to avoid concurrency issues like race conditions and deadlock. Here's a slightly more involved program where friends meet up (live demo here). See the broken output on right.

```c
// confused-friends.c
static const char *kFriends[] = {
    "Jack", "Michaela", "Luis", "Richard", "Jordan", "Lisa","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;
}
```

- 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.
Fortunately, the fix is simple: pass the relevant `const char *` instead. Snapshots of the `const char *` pointers are passed verbatim to `meetup`. The strings themselves are constants. Fixed program [right here](https://example.com).
Threads (continued)

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.

```
myth60$ ./friends
Let's hear from 6 friends.
Hey, I'm Roz. Empowered to meet you.
Hey, I'm Jerry. Empowered to meet you.
Hey, I'm Blanca. Empowered to meet you.
Hey, I'm Ruchir. Empowered to meet you.
Hey, I'm Garrick. Empowered to meet you.
Hey, I'm Ryan. Empowered to meet you.
All friends are real!
```

```
myth60$ ./friends
Let's hear from 6 friends.
Hey, I'm Roz. Empowered to meet you.
Hey, I'm Blanca. Empowered to meet you.
Hey, I'm Jerry. Empowered to meet you.
Hey, I'm Ruchir. Empowered to meet you.
Hey, I'm Ryan. Empowered to meet you.
Hey, I'm Garrick. Empowered to meet you.
All friends are real!
```
C++ Threads

- Introverts Revisited, in C++: Rather than deal with pthreads as a platform-specific extension of C, I'd rather use a thread package that's officially integrated into the language itself.
  - As of 2011, C++ provides support for threading and many synchronization directives.
  - Because C++ provides better alternatives for generic programming than C does, we avoid the void * tomfoolery required when using pthreads.
- Presented below is the object-oriented C++ equivalent of the introverts example we've already seen once before. The full program is online right here.

```cpp
static void recharge() {
    cout << oslock << "I recharge by spending time alone." << endl << osunlock;
}

static const size_t kNumIntroverts = 6;
int main(int argc, char *argv[]) {
    cout << "Let's hear from " << kNumIntroverts << " introverts." << endl
    thread introverts[kNumIntroverts]; // declare array of empty thread handles
    for (thread& introvert: introverts) { // move anonymous threads into empty handles
        introvert = thread(recharge);
    }
    for (thread& introvert: introverts)
        introvert.join();
    cout << "Everyone's recharged!" << endl;
    return 0;
}
```

Reading: B&O’s Concurrent Programming with Threads chapter
C++ Threads (continued)

- We declare an array of empty thread handles as we did in the equivalent C version.
- We install the recharge function into temporary threads that are then moved (via the thread's operator = (thread&& other)) into a previously empty thread handle.
  - This is a relatively new form of operator= that fully transplants the contents of the thread on the right into the thread on the left, leaving the thread on the right fully gutted, as if it were zero-arg constructed. Restated, the left and right thread objects are effectively swapped.
  - This is an important distinction because a traditional operator= would produce a second working copy of the same thread, and we don't want that.

```cpp
static void recharge() {
    cout << oslock << "I recharge by spending time alone." << endl << osunlock;
}

static const size_t kNumIntroverts = 6;
int main(int argc, char *argv[]) {
    cout << "Let's hear from " << kNumIntroverts << " introverts." << endl
    thread introverts[kNumIntroverts]; // declare array of empty thread handles
    for (thread& introvert: introverts) // move anonymous threads into empty handles
        introvert = thread(recharge);
    for (thread& introvert: introverts)
        introvert.join();
    cout << "Everyone's recharged!" << endl;
    return 0;
}
```
C++ Threads (continued)

- The `join` method is equivalent to the `pthread_join` function we've already discussed.
- The prototype of the thread routine—in this case, `recharge`—can be anything (although the return type is always ignored, so it should generally be `void`).
- `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.

```cpp
static void recharge() {
    cout << oslock << "I recharge by spending time alone." << endl << osunlock;
}

static const size_t kNumIntroverts = 6;
int main(int argc, char *argv[]) {
    cout << "Let's hear from " << kNumIntroverts << " introverts."
    for (thread& introvert: introverts) // declare array of empty thread handles
        introvert = thread(recharge); // move anonymous threads into empty handles
    for (thread& introvert: introverts)
        introvert.join();
    cout << "Everyone's recharged!" << endl;
    return 0;
}
```
End of Lecture 10