CS111, Lecture 12
Multithreading Introduction

Optional reading:

Operating Systems: Principles and Practice (2nd Edition): Chapter 4 and Chapter 5 up through Section 5.1

Announcements

- Assign1 being graded, grades will be posted later this week
- Due to WiFi issues, lecture attendance policy being updated / re-evaluated, updates coming soon!

Recap: Pipes and I/O Redirection

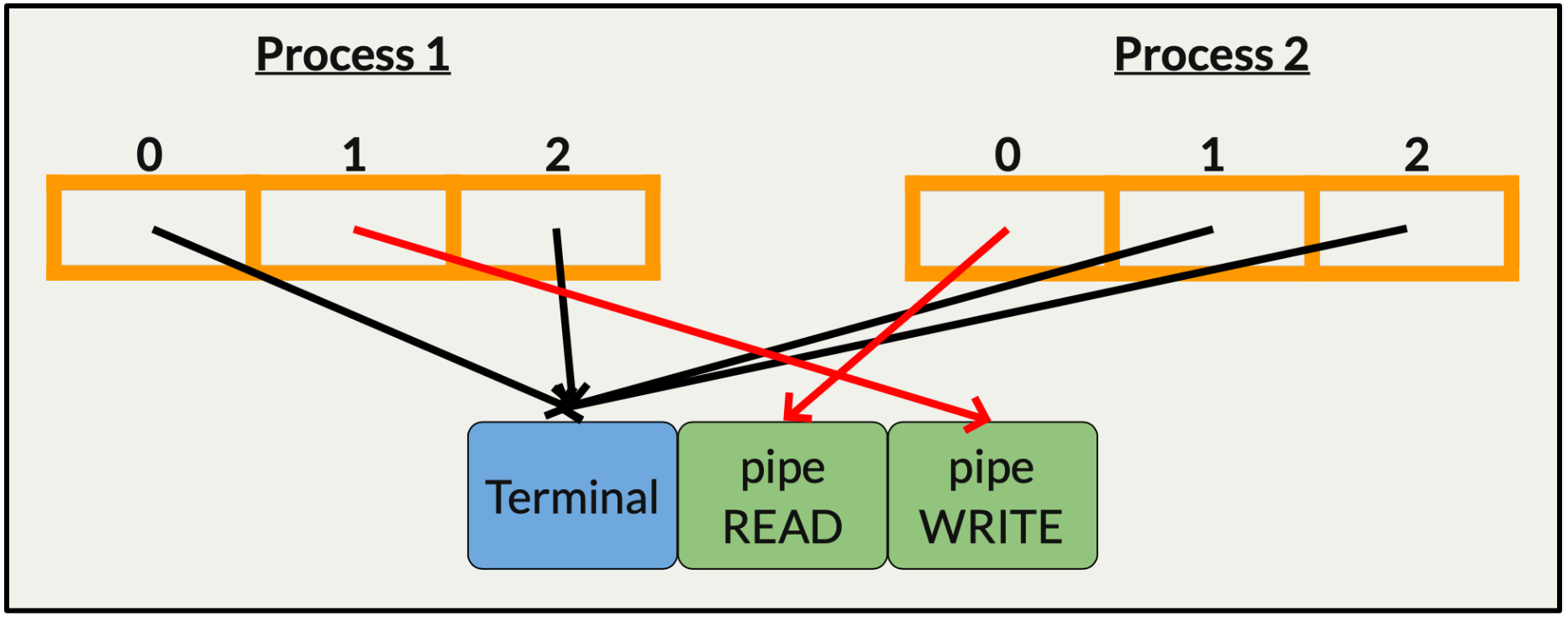
Pipes are sets of file descriptors that allow us to communicate across processes.

- Processes can share these file descriptors because they are copied on **fork()**
- File descriptors 0,1 and 2 are special and assumed to represent STDIN, STDOUT and STDERR
- If we change those file descriptors to point to other resources, we can redirect STDIN/STDOUT/STDERR to be something else without the program knowing!
- Pipes are how terminal support for piping and redirection (**command1 | command2** and **command1 > file.txt**) are implemented!

Redirecting Process I/O

Idea: what happens if we change a special FD to point somewhere else?

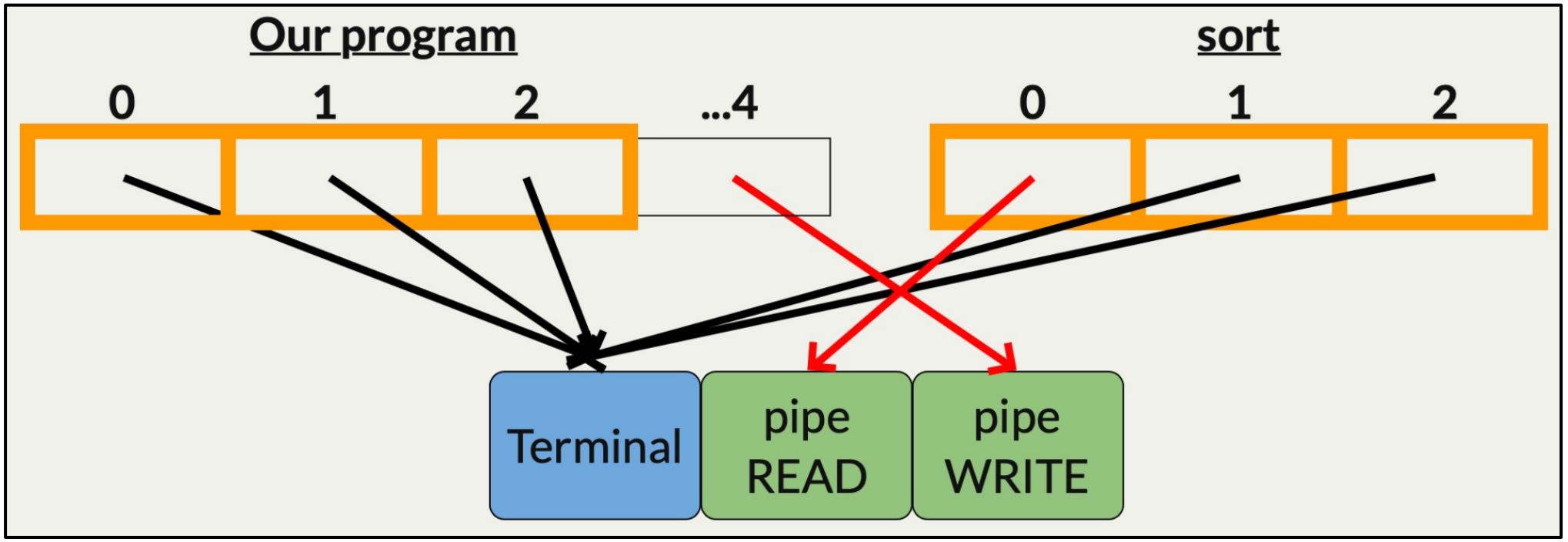
Could we do this with a pipe?



This is how piping works in the terminal! And the executables don't know they are using a pipe.

Redirecting Process I/O

Stepping stone: our first goal is to write code that spawns another program and sends data to its STDIN via a separate file descriptor.



This is useful because our program can now programmatically run another procedure and feed it input (even if we don't have the code for it).

subprocess

Implementing subprocess:

1. Create a pipe
2. Spawn a child process
3. That child process changes its STDIN to be the pipe read end (how?)
4. That child process calls **execvp** to run the specified command
5. We return the pipe write end to the caller along with the child's PID. That caller can write to the file descriptor, which appears to the child as its STDIN

execvp consumes the process but *leaves the file descriptor table in tact!*

subprocess

```
subprocess_t subprocess(const char *command) {
    // this line parses the command into a pipeline like is done for you on assign3
    pipeline p(command);

    // Make a pipe
    int fds[2];
    pipe(fds);

    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        // We are not writing to the pipe, only reading from it
        close(fds[1]);

        // Duplicate the read end of the pipe into STDIN
        dup2(fds[0], STDIN_FILENO);
        close(fds[0]);

        // Run the command
        execvp(p.commands[0].argv[0], p.commands[0].argv);
        exitIf(true, kExecFailed, stderr, "execvp failed to invoke this: %s.\n", command);
    }
    ...
```

subprocess

```
subprocess_t subprocess(const char *command) {
    // this line parses the command into a pipeline like is done for you on assign3
    pipeline p(command);

    // Make a pipe
    int fds[2];
    pipe(fds);

    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        ...
    }

    close(fds[0]);
    subprocess_t returnStruct;
    returnStruct.pid = pidOrZero;
    returnStruct.supplyfd = fds[1];
    return returnStruct;
}
```

What if we wanted to read the output of the child instead of feeding input to the child?

subprocess

```
subprocess_t subprocess(const char *command) {
    // this line parses the command into a pipeline like is done for you on assign3
    pipeline p(command);

    // Make a pipe
    int fds[2];
    pipe(fds);

    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        // We are not reading from the pipe, only writing to it
        close(fds[0]);

        // Duplicate the read end of the pipe into STDIN
        dup2(fds[1], STDOUT_FILENO);
        close(fds[1]);

        // Run the command
        execvp(p.commands[0].argv[0], p.commands[0].argv);
        exitIf(true, kExecFailed, stderr, "execvp failed to invoke this: %s.\n", command);
    }
    ...
```

subprocess

```
subprocess_t subprocess(const char *command) {
    // this line parses the command into a pipeline like is done for you on assign3
    pipeline p(command);

    // Make a pipe
    int fds[2];
    pipe(fds);

    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        ...
    }

    close(fds[1]);
    subprocess_t returnStruct;
    returnStruct.pid = pidOrZero;
    returnStruct.ingestfd = fds[0];
    return returnStruct;
}
```

Check out subprocess-dual.cc in the lecture 11 code for an example subprocess implementation that does both input and output rewiring.

Topic 3: Multithreading - How can we have concurrency within a single process? How does the operating system support this?

CS111 Topic 3: Multithreading

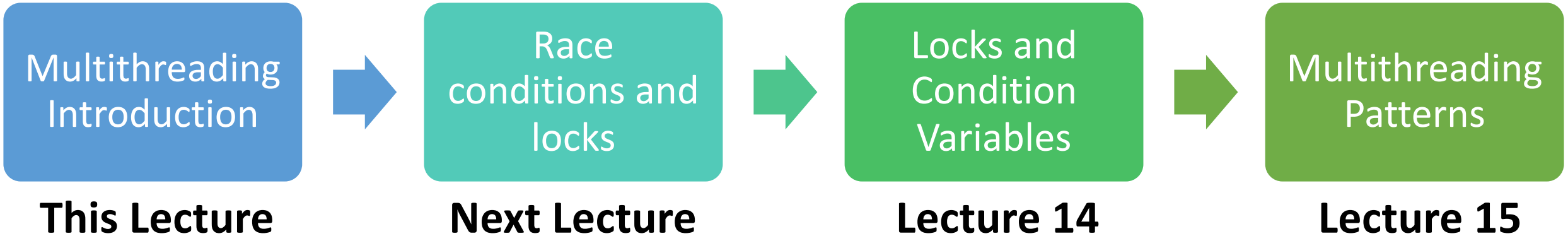
Multithreading - *How can we have concurrency within a single process? How does the operating system support this?*

Why is answering this question important?

- Helps us understand how a single process can do multiple things at the same time, a technique used in various software (today)
- Provides insight into *race conditions*, unpredictable orderings that can cause undesirable behavior, and how to fix them (next few lectures)
- Allows us to see how the OS schedules and switches between tasks (after midterm)

assign4: implement several multithreaded programs while eliminating race conditions

CS111 Topic 3: Multithreading, Part 1



assign4: implement several multithreaded programs while eliminating race conditions!

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

- Introducing multithreading
- **Example:** greeting friends
- Race conditions
- Threads share memory
- **Example:** selling tickets

```
cp -r /afs/ir/class/cs111/lecture-code/lect12 .
```

Plan For Today

- **Introducing multithreading**
- **Example:** greeting friends
- Race conditions
- Threads share memory
- **Example:** selling tickets

```
cp -r /afs/ir/class/cs111/lecture-code/lect12 .
```

From Processes to Threads

Multiprocessing has allowed us to spawn other processes to do tasks or run programs

- Powerful; can execute/wait on other programs, secure (separate memory space), communicate with pipes and signals
- But limited; interprocess communication is cumbersome, hard to share data/coordinate
- Is there another way we can have concurrency beyond multiprocessing that handles these tradeoffs differently?

From Processes to Threads

We can have concurrency *within a single process* using **threads:** independent execution sequences within a single process.

- Threads let us run multiple functions in our program concurrently
- Multithreading is common to parallelize tasks, especially on multiple cores
- In C++: spawn a thread using **thread()** and the **thread** variable type and specify what function you want the thread to execute (optionally passing parameters!)
- Each thread operates within the same process, so they *share a virtual address space* (!) (globals, heap, pass by reference, etc.)
- The processes's stack segment is divided into a "ministack" for each thread.
- In the OS, threads are actually the unit of concurrency, not processes (more on this later)
- Many similarities between threads and processes, but some key differences

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

Threads are a much more common approach to doing tasks in parallel on a system because of easier data sharing and lighter weight creation. Many modern software programs use multithreading:

- Mobile apps may spawn a background thread to download a web resource while allowing another thread to handle user input (e.g. taps, swipes)
- A web server may spawn threads to handle incoming requests in parallel
- Your computer's task manager can show you how many threads a process is using

C++ Thread

A thread object can be spawned to run the specified function with the given arguments.

```
thread myThread(myFunc, arg1, arg2, ...);
```

- **myFunc:** the function the thread should execute asynchronously
- **args:** a list of arguments (any length, or none) to pass to the function upon execution
- **myFunc**'s function's return value is ignored (use pass by reference instead)
- Once initialized with this constructor, the thread may execute at any time!

C++ Thread

To wait on a thread to finish, use the **.join()** method:

```
thread myThread(myFunc, arg1, arg2);
...
// Wait for thread to finish (blocks)
myThread.join();
```

For multiple threads, we must wait on a specific thread one at a time:

```
thread friends[5];
...
for (int i = 0; i < 5; i++) {
     friends[i].join();
}
```

Plan For Today

- Introducing multithreading
- **Example: greeting friends**
- Race conditions
- Threads share memory
- **Example:** selling tickets

```
cp -r /afs/ir/class/cs111/lecture-code/lect12 .
```

Our First Threads Program

```
static void greeting(size_t i) {
    cout << "Hello, world! I am thread " << i << endl;
}

...
```

friends.cc

Our First Threads Program

```
static const size_t kNumFriends = 6;

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

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

    // Wait for threads
    for (size_t i = 0; i < kNumFriends; i++) {
        friends[i].join();
    }

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

C++ Thread

We can make an array of threads as follows:

```
// declare array of empty thread handles
thread friends[5];

// Spawn threads
for (size_t i = 0; i < 5; i++) {
    friends[i] = thread(myFunc, arg1, arg2);
}
```

We can also initialize an array of threads as follows (note the loop by reference):

```
thread friends[5];
for (thread& currFriend : friends) {
    currFriend = thread(myFunc, arg1, arg2);
}
```

Plan For Today

- Introducing multithreading
- **Example:** greeting friends
- **Race conditions**
- Threads share memory
- **Example:** selling tickets

```
cp -r /afs/ir/class/cs111/lecture-code/lect12 .
```

Race Conditions

- Like with processes, threads can execute in unpredictable orderings.
- A **race condition** is an unpredictable ordering of events where some orderings may cause undesired behavior.
- A *thread-safe* function is one that will always execute correctly, even when called concurrently from multiple threads.
- **printf** is thread-safe, but **operator<<** is *not*. This means e.g. **cout** statements could get interleaved!
- To avoid this, use **oslock** and **osunlock** (custom CS111 functions - **#include "ostreamlock.h"**) around streams. They ensure at most one thread has permission to write into a stream at any one time.

```
cout << oslock << "Hello, world!" << endl << osunlock;
```

Our First Threads Program

```
static void greeting(size_t i) {
    cout << oslock << "Hello, world! I am thread " << i << endl <<
osunlock;
}

...
```

friends.cc

Recap

- Introducing multithreading
- **Example:** greeting friends
- Race conditions
- Threads share memory
- **Example:** selling tickets

Next time: introducing mutexes

Lecture 12 takeaway: A process can have multiple threads executing tasks simultaneously. Threads share the same virtual address space, and race conditions can cause unintended problems!

```
cp -r /afs/ir/class/cs111/lecture-code/lect12 .
```