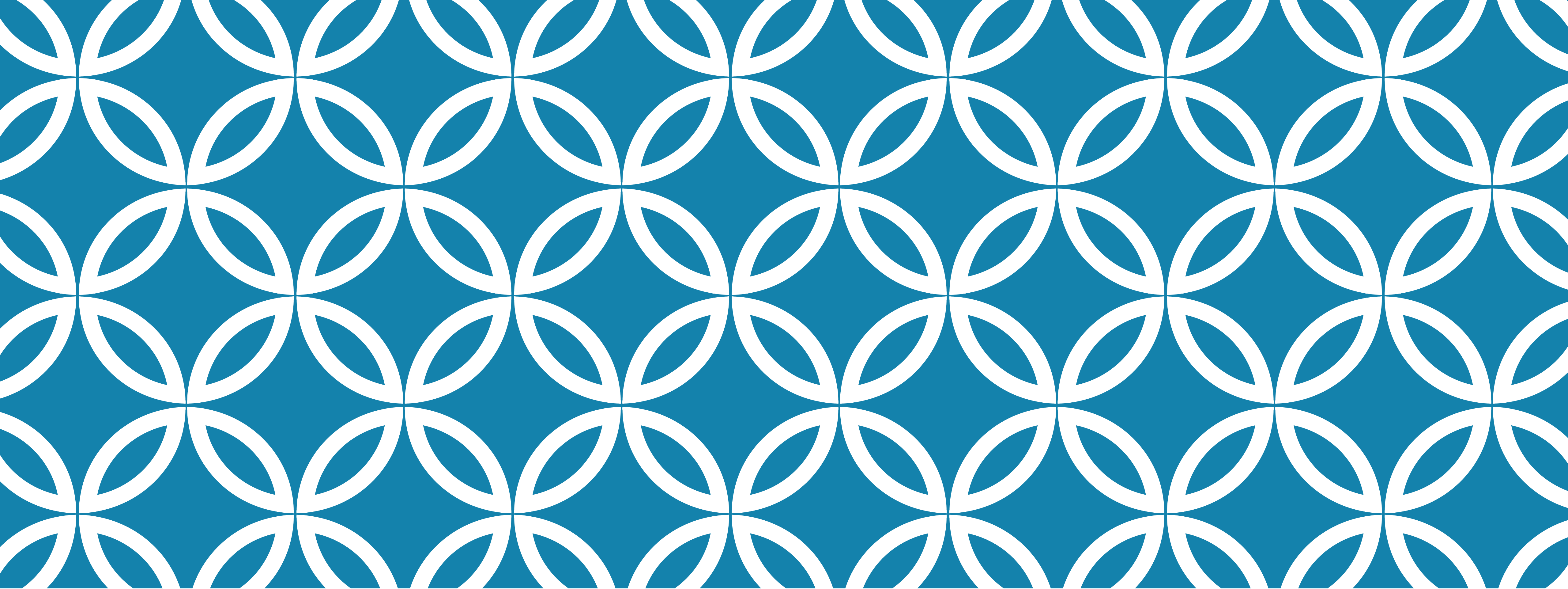


CS4414: RECITATION 8 — FORK(), EXEC(), AND THREADS

Ricky Takkar

Friday, March 17, 2023

INTRODUCING OUR NEXT MODULE

- Our next large module looks at threads and thread-level synchronization
  - More broadly, our focus is on **multiprocessing:** situations in which more than one element of a system cooperate to carry out some task, using a mixture of methods:
    - multiple **threads**, multiple **processes**, perhaps multiple **computers**, perhaps even attached **hardware accelerators** that are themselves programmable.

REMINDER: WHAT IS A THREAD? (FROM LECTURE 14)

But we can also write a single program that, at runtime, uses parallelism internally, via what we call a *thread*.

Use of threads requires a deep understanding of performance consequences, overheads, and how to program correctly with concurrency.

Many programs would slow down or crash if you just threw threads in.

THREADS IN THE LINUX KERNEL! (FROM LECTURE 14)

Linux itself is a multithreaded program. Each system call runs on a distinct thread, and the Linux scheduler and file system have additional threads of their own.

Linux evolved over decades to take full advantage of this power. It wasn't obvious or easy!

WORD COUNT (FROM LECTURE 14)

Recall our word count from Lectures 1-3. It had:

- One "main" thread to process the arguments, then launch threads
- One thread just to open files
- N threads to count words, in parallel, on distinct subsets of the files and implement parallel count-tree merge

Main thread resumed control at the end, sorted output, printed it.

HOW LINUX CREATES THREADS/PROCESSES (FROM LECTURE 14)

Any process can "clone itself" by calling pid = fork().

The parent process will receive the pid of its new child.

The child process is identical to the parent (even shares the same open files, like stdin, stdout, stderr), but gets pid 0. Typically, the child immediately "sets up" a runtime environment for itself.

WHY "FORK" (FROM LECTURE 14)

"Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood…"
-- Robert Frost

Because of poetry!

Recall that in Linux, every process has a parent process, and /etc/init (runs at boot time) is the parent of everything.

The inventors of Unix (first version of Linux) visualized this a bit like that famous road in the woods…

THE TERM "FORK" HAS LINGERED (FROM LECTURE 14)

If someone says "fork off a thread" or "fork off a process" it refers to creating a new concurrent task.

Later, we might wait for that thread or thread to finish. This is called a join (like when a stream joins a river)

FORK FOLLOWED BY EXEC (FROM LECTURE 14)

In Linux we normally call exec after calling fork.

Fork creates the process and leaves the parent process an opportunity to “set up” the runtime environment of the child.

Then exec launches some other program, but it runs in the same process “context” that the forked child set up.

FORK() AND EXEC() SYSTEM CALLS

- fork() system call creates a **separate, duplicate process**
  - Process that calls fork() is the **parent;** the spawned process is its **child** (parent and child have **different** pid)

- exec() system call **replaces** calling process (including all threads) with program specified in parameter (original and replacement processes **share** pid)

USING FORK() AND EXEC() IN C++ (PLUS SOME BONUS LINUX COMMANDS)

Let's code!

Outcome:

- Learned how and why fork() and exec() can be used in C++
- Utilized Linux commands such as ps and fg to monitor and control processes
- Realized the importance of perror()

BREAK: 5-10 MINUTES

WITH THREADS, YOU CAN “FOLLOW BOTH PATHS” IN THE WOODS… (FROM LECTURE 14)

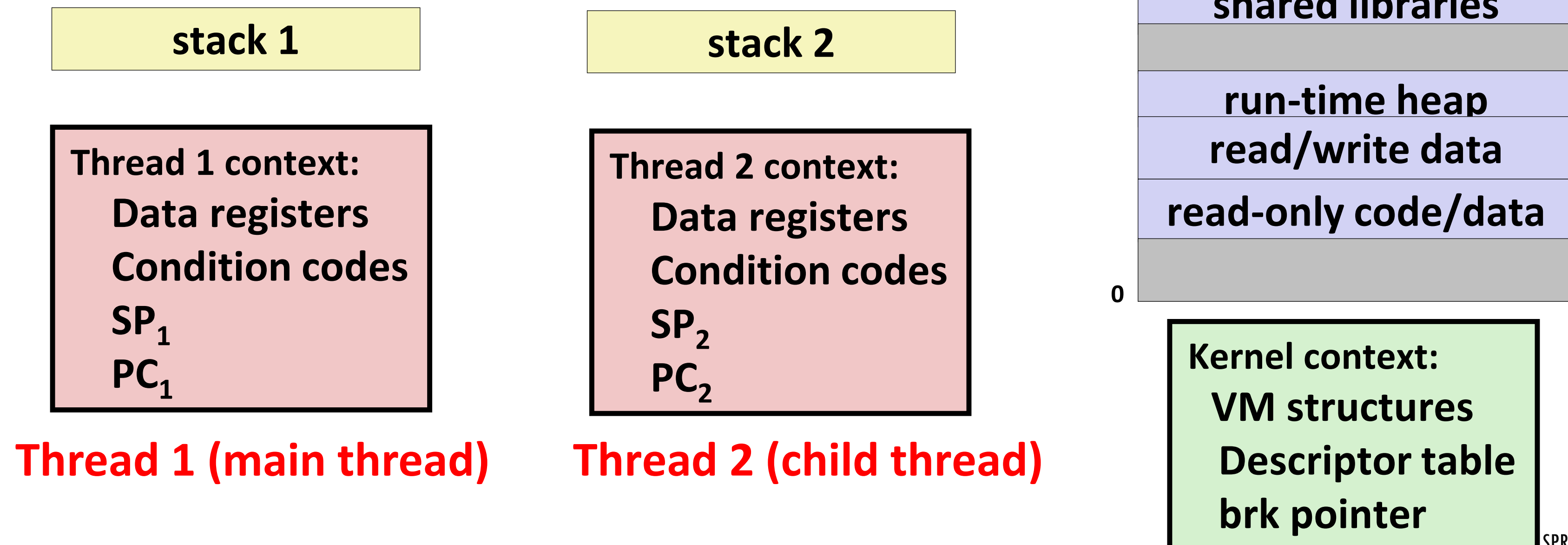
In computing, some ideas (like recursion) are really earth-shaking

Concurrency is one of them! In some ways very hard to do properly, because of mistakes that can easily arise, and hidden costs that can destroy the speedup benefits.

But in other ways, concurrency is revolutionary because we use the hardware so efficiently.

THREAD CONTEXTS (FROM LECTURE 14)

- **Multiple threads can be associated with a process**
  - Each thread has its own logical control flow
  - Each thread shares the same code, data, and kernel context
  - Each thread has its own stack for local variables
    - but not protected from other threads
  - Each thread has its own thread id (TID)



THREAD STACKS (FROM LECTURE 14)

Although the main thread has a stack that can grow without limit, this is *not* the situation for spawned child threads.

They have limited stack sizes (default: 2MB, but you can specify a larger size)

Overflow will cause the entire process to crash.

STACK ALLOCATION: SAFE, BUT BE CAUTIOUS (FROM LECTURE 14)

2MB is a large amount of space and won't easily be used up. C++ gives a stack overflow exception if you manage to do so.

But we can't put really big objects on the stack, or do really deep recursion with even medium-sized objects on the stack.

SO FAR IN THIS COURSE, WE'VE NOT EXPLOITED MULTITHREADING

- What that means is each program we've written so far has been run by a machine in a *line-by-line* way
  - What are some **multithreaded** programs you use everyday, and what makes you say they are multithreaded?
  - Would you say this notion of concurrent programming serves, *generally*, to make programs more **performant** and **functional**?

TIME FOR ANOTHER DEMO

Outcome:

- Created and accessed threads to boost functionality of program
- Realized thread lifecycle and consider design to optimize joint behavior

SUMMARY

- *fork() and exec() demo*
  - Learned how and why fork() and exec() are used in C++
  - Utilized Linux commands such as ps and fg to monitor and control processes through the command line
  - Realized the importance of perror() to locate errors in code
- *threads demo*
  - Created and accessed threads to boost functionality of program
  - Realized thread lifecycle and considered design to optimize joint behavior