Concurrency: Processes
CSE 333 Spring 2025

Instructor: Hal Perkins

Teaching Assistants:

Hannah Hempstead	Lainey Jeon	Hannah Jiang
Irene Lau	Nathan Li	Leanna Nguyen
Janani Raghavan	Deeksha Vatwani	Yiqing Wang
Jennifer Xu		

Administrivia (1)

❖ hw4 due Thursday night

  ▪ Usual late days (2 max) apply if you have any remaining

❖ Final exam Wed., June 11, 2:30-4:20

  ▪ In CSE2 G01 (A-L) and CSE G10 (M-Z), **not** regular classroom

  ▪ Review session Tue., June 10, 4:30-~5:30, CSE2 G20 (here)

  ▪ Topic list on the web now; exam will be somewhat weighted towards 2nd half of the quarter

  ▪ Closed book but you may have two 5x8 cards with handwritten notes

    • Free blank cards available after class

Administrivia (2)

❖ Please nominate great TAs for the Bandes award when nominations are available

❖ Please fill out course evals while they are available

❖ Do-not-miss (if you can possibly make it) talk tomorrow at 3:30 here (G20). David Patterson (UC Berkeley & Google), Turing Award winner for computer architecture and co-author of the most influential architecture textbooks for the last several decades, and Andy Konwinski (Laude Institute): "Shaping AI for the Public Good."

Outline

- ❖ `searchserver`
  - ▪ Sequential
  - ▪ Concurrent via forking threads – `pthread_create()`
  - ▪ **Concurrent via forking processes – `fork()`**
  - ▪ Concurrent via non-blocking, event-driven I/O – `select()`
    - We won't get to this ☹

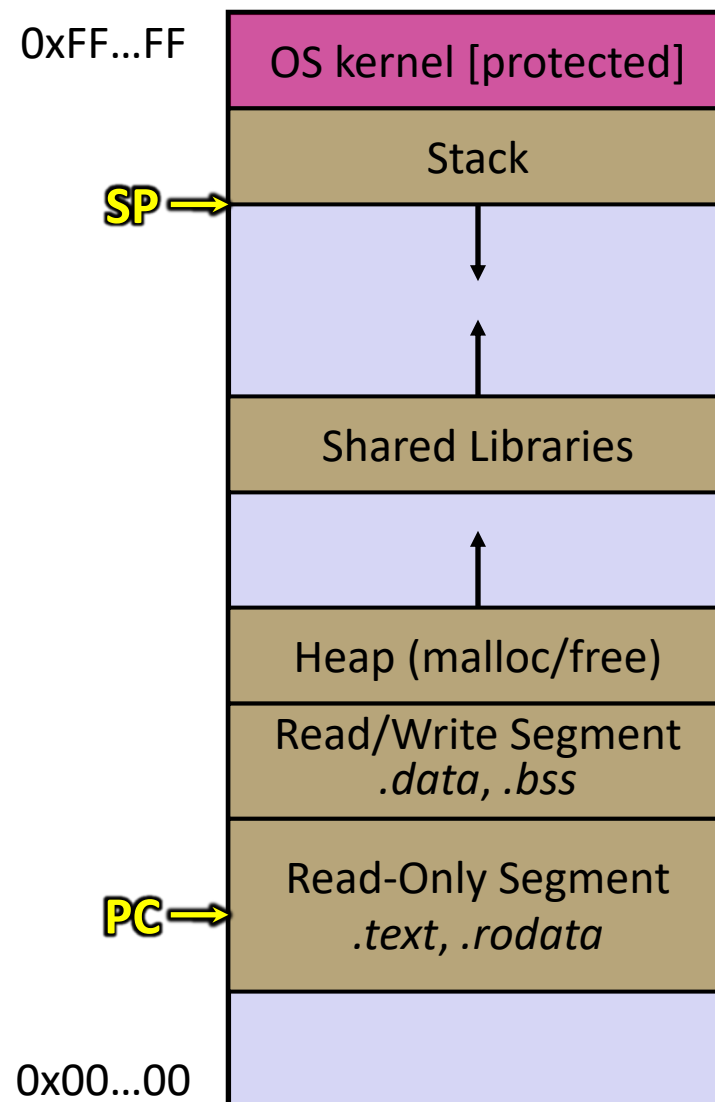- ❖ Reference: *Computer Systems: A Programmer's Perspective,* Chapter 12 (CSE 351 book)

Creating New Processes

❖ `pid_t fork(void);`

- Creates a new process (the "child") that is an *exact clone** of the current process (the "parent")

  - *Everything is cloned except threads: variables, file descriptors, open sockets, the virtual address space (code, globals, heap, stack), etc.

- Primarily used in two patterns:

  - Servers: fork a child to handle a connection
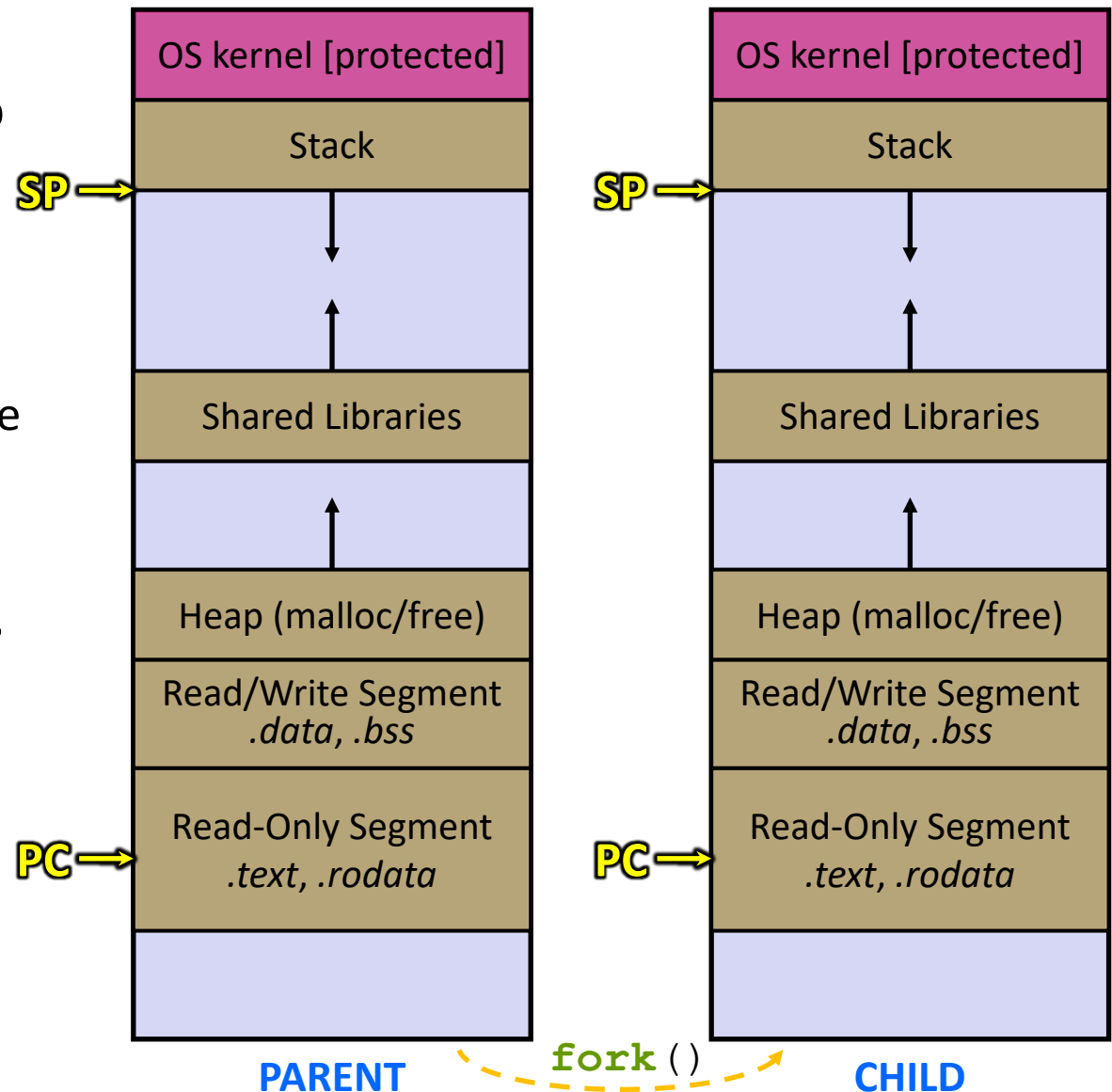
  - Shells: fork a child that then exec's a new program

`fork()` and Address Spaces

❖ A process executes within an *address space*

- Includes segments for different parts of memory

- Process tracks its current state using the stack pointer (SP) and program counter (PC)

0xFF...FF	OS kernel [protected]
	Stack
SP →	
	Shared Libraries
	Heap (malloc/free)
	Read/Write Segment *.data, .bss*
PC →	Read-Only Segment *.text, .rodata*
0x00...00	

`fork()` and Address Spaces

❖ Fork cause the OS to clone the

address space

▪ The *copies* of the memory segments are (nearly) identical

▪ The new process has *copies* of the parent's data, stack-allocated variables, open file descriptors, etc.
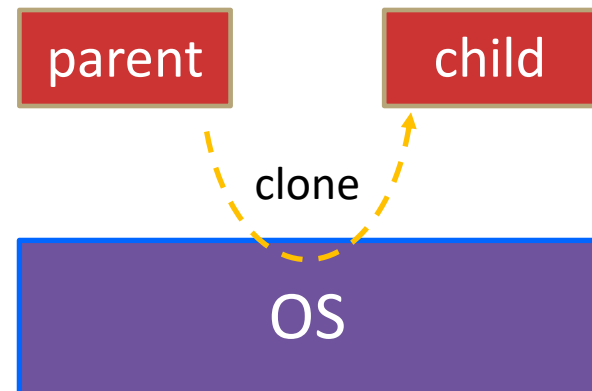
PARENT		CHILD
OS kernel [protected]		OS kernel [protected]
Stack		Stack
SP →		SP →
↓ ↑		↓ ↑
Shared Libraries		Shared Libraries
↑		↑
Heap (malloc/free)		Heap (malloc/free)
Read/Write Segment .data, .bss		Read/Write Segment .data, .bss
PC → Read-Only Segment .text, .rodata		PC → Read-Only Segment .text, .rodata

`fork()`

7

`fork()`

❖ **`fork`**`()` has peculiar semantics

- The parent invokes **`fork`**`()`

- The OS clones the parent

- *Both* the parent and the child return from fork

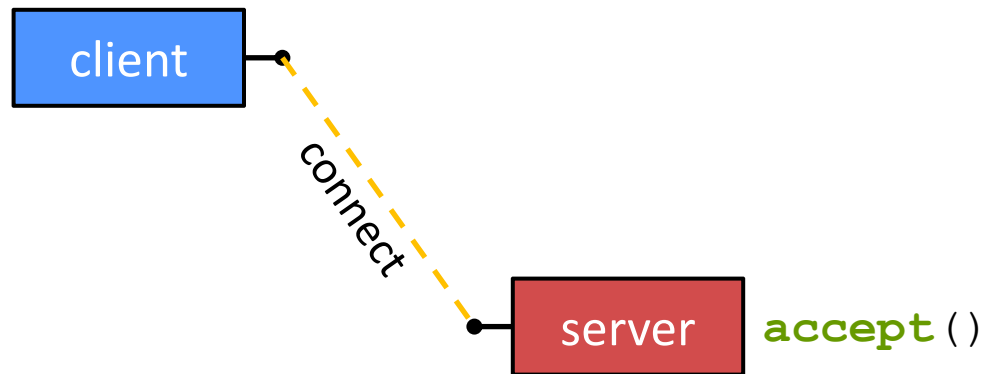 - Parent receives child's pid

  - Child receives a 0

parent

`fork``()`

OS

`fork()`

- ❖ **`fork`**`()` has peculiar semantics

  - ▪ The parent invokes **`fork`**`()`

  - ▪ The OS clones the parent

  - ▪ *Both* the parent and the child return from fork

    - • Parent receives child's pid

    - • Child receives a 0

| parent | child |

clone

OS

`fork()`

❖ **`fork`**`()` has peculiar semantics

  ▪ The parent invokes **`fork`**`()`

  ▪ The OS clones the parent

  ▪ *Both* the parent and the child return from fork

    • Parent receives child's pid

    • Child receives a 0

parent   child

child pid   0

OS
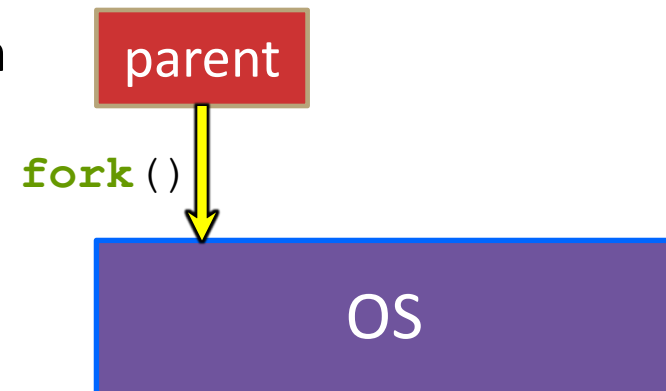
❖ See `fork_example.cc`

Concurrent Server with Processes

❖ The **parent** process blocks on `accept()`, waiting for a new client to connect

- When a new connection arrives, the parent calls `fork()` to create a **child** process
- The child process handles that new connection and `exit()`'s when the connection terminates

❖ Remember that children become "zombies" after termination
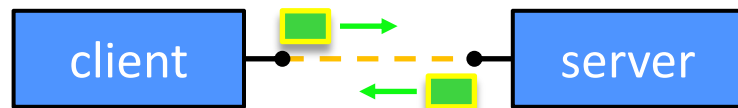
- <u>Option A</u>: Parent calls `wait()` to "reap" children
- <u>Option B</u>: Use a double-fork trick

Double-fork Trick

Double-fork Trick

client

connect

server   `accept()`

Double-fork Trick

client

server

server

fork() child

Double-fork Trick

Double-fork Trick

client — server
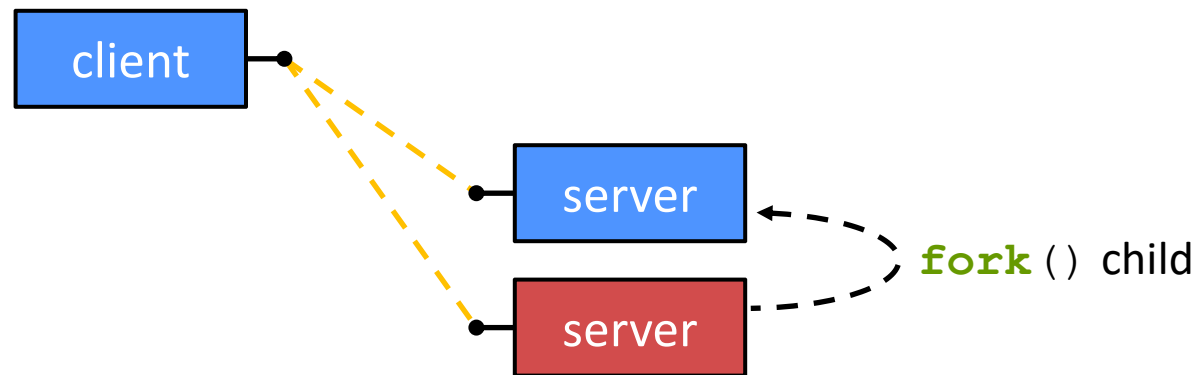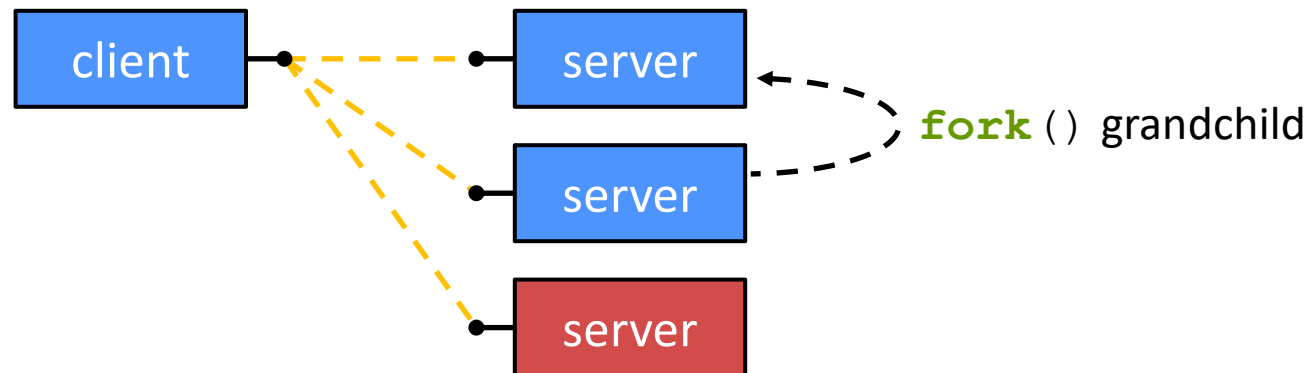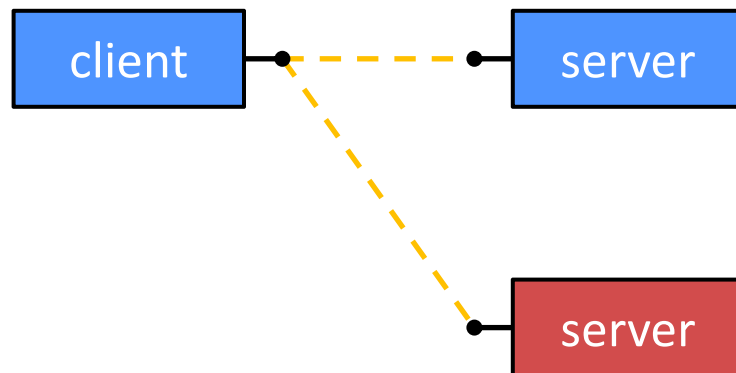
child **exit**()'s / parent **wait**()'s

server

Double-fork Trick

client - - - - - - server

server   parent closes its
          client connection

Double-fork Trick

Double-fork Trick

fork() child

fork() grandchild
exit()
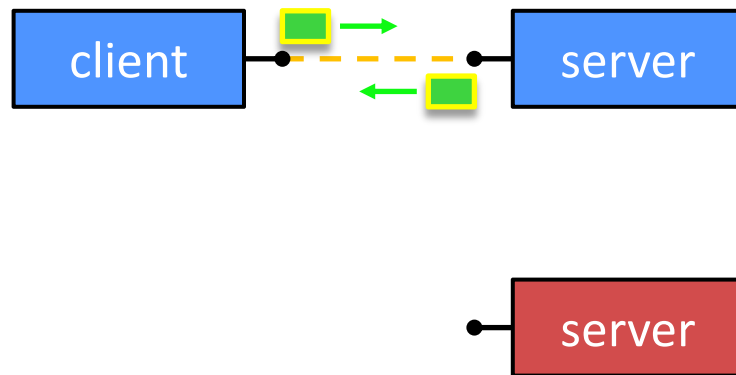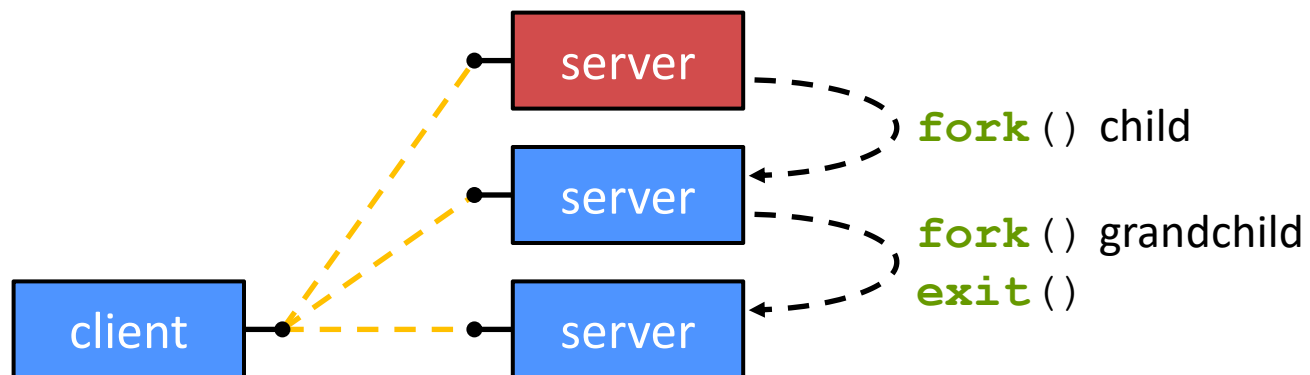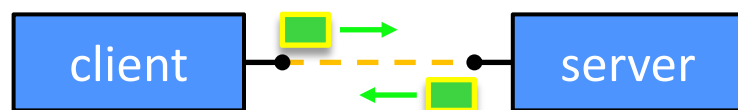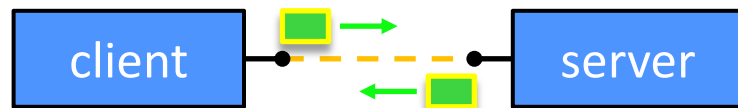
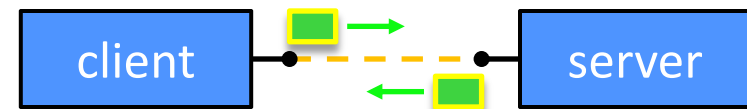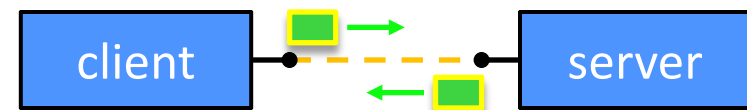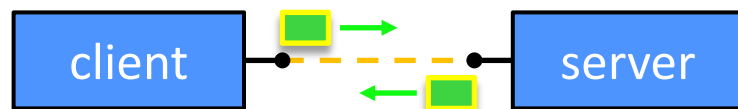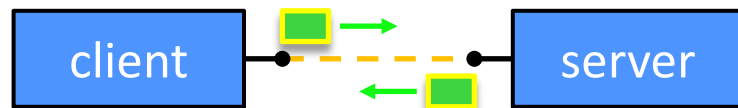# Double-fork Trick

Double-fork Trick
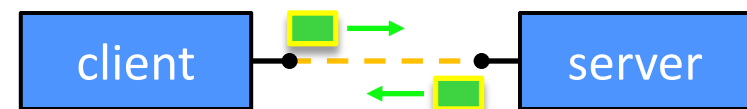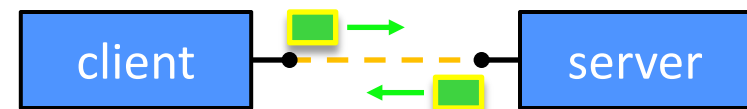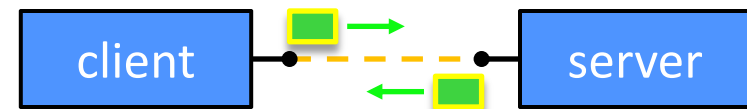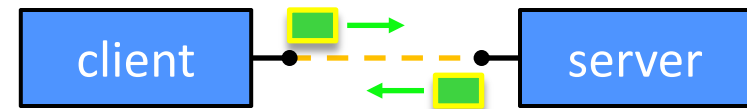
Concurrent with Processes

❖ See `searchserver_processes/`

Whither Concurrent Processes?

❖ Advantages:

- Almost as simple to code as sequential

  - In fact, most of the code is identical!

- Concurrent execution leads to better CPU, network utilization

❖ Disadvantages:

- Processes are heavyweight

  - Relatively slow to fork

  - Context switching latency is high

- Communication between processes is complicated

How Fast is `fork()`?

❖ See `forklatency.cc`

❖ ~ **0.25 ms** per fork*

  ▪ ∴ maximum of (1000/0.25) = 4,000 connections/sec/core

  ▪ ~350 million connections/day/core

    • This is fine for most servers

    • Too slow for super-high-traffic front-line web services

      – Facebook served ~ 750 billion page views per day in 2013!
        Would need 3-6k cores just to handle **fork**`()`, *i.e.* without doing any work
        for each connection

❖ *Past measurements are not indicative of future performance – depends on hardware, OS, software versions, …

How Fast is `pthread_create()`?

❖ See `threadlatency.cc`

❖ **~0.036 ms** per thread creation*

  ▪ ~10x faster than **fork**`()`

  ▪ ∴ maximum of (1000/0.036) = 28,000 connections/sec

  ▪ ~2.4 billion connections/day/core

❖ Much faster, but writing safe multithreaded code can be serious voodoo

❖ *Past measurements are not indicative of future performance – depends on hardware, OS, software versions, …, but will typically be an order of magnitude faster than fork()

Aside: Thread Pools

❖ In real servers, we'd like to avoid overhead needed to create a new thread or process for every request

❖ Idea: Thread Pools:

  ▪ Create a fixed set of worker threads or processes on server startup and put them in a queue

  ▪ When a request arrives, remove the first worker thread from the queue and assign it to handle the request

  ▪ When a worker is done, it places itself back on the queue and then sleeps until dequeued and handed a new request

More Administrivia

❖ We're done with concurrency

❖ So what should we do about class on Wednesday?

❖ Proposal: no class Wednesday

   ▪ Use the extra time to wrap up hw4

   ▪ Should be able to have TAs in lecture room (G20) at regular time for extra office hours

❖ See you on Friday to wrap up the quarter

   ▪ But be sure to go to sections on Thursday!!