

# Concurrency: Processes

## CSE 333 Spring 2022

**Instructor:** Hal Perkins

**Teaching Assistants:**

Esau Abraham

Nour Ayad

Ramya Challa

Cleo Chen

Sanjana Chintalapati

Dylan Hartono

Kenzie Mihardja

Brenden Page

Aakash bin Srazali

Justin Tysdal

Julia Wang

Timmy Yang

# Administrivia

- ❖ hw4 due Thur. night
  - (Plus late days – everyone can use 2 late days on hw4!)
- ❖ Please nominate great TAs for the Bandes award
- ❖ Please fill out course evals while they are available
- ❖ Wrapup sections tomorrow and class on Friday
- ❖ Final exam Wed. June 8, 2:30-4:20, CSE2 G20
  - Review session Tue., June 7, 4:30-~5:30, also CSE2 G20
  - Topic list on the web now; exam will be somewhat weighted towards 2<sup>nd</sup> half of the quarter
  - Closed book but you may have two 5x8 cards (or equivalent) with handwritten notes

# 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)

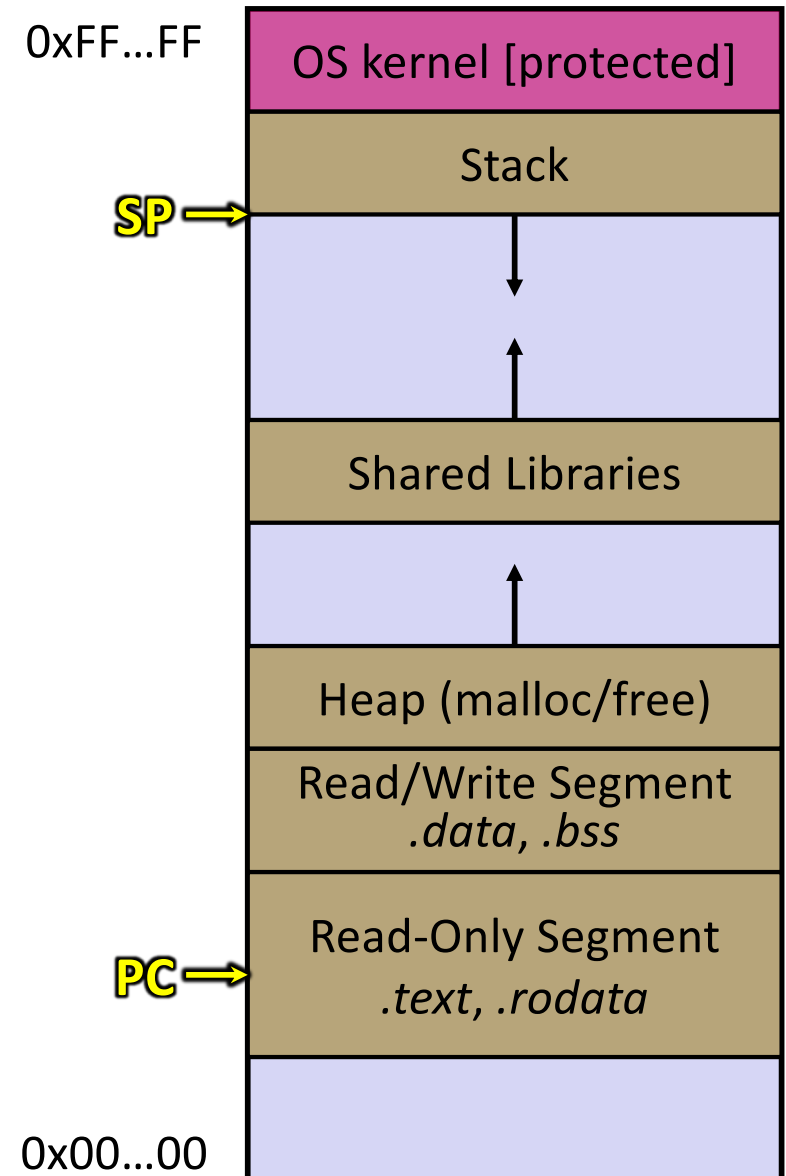
# Creating New Processes

❖ `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

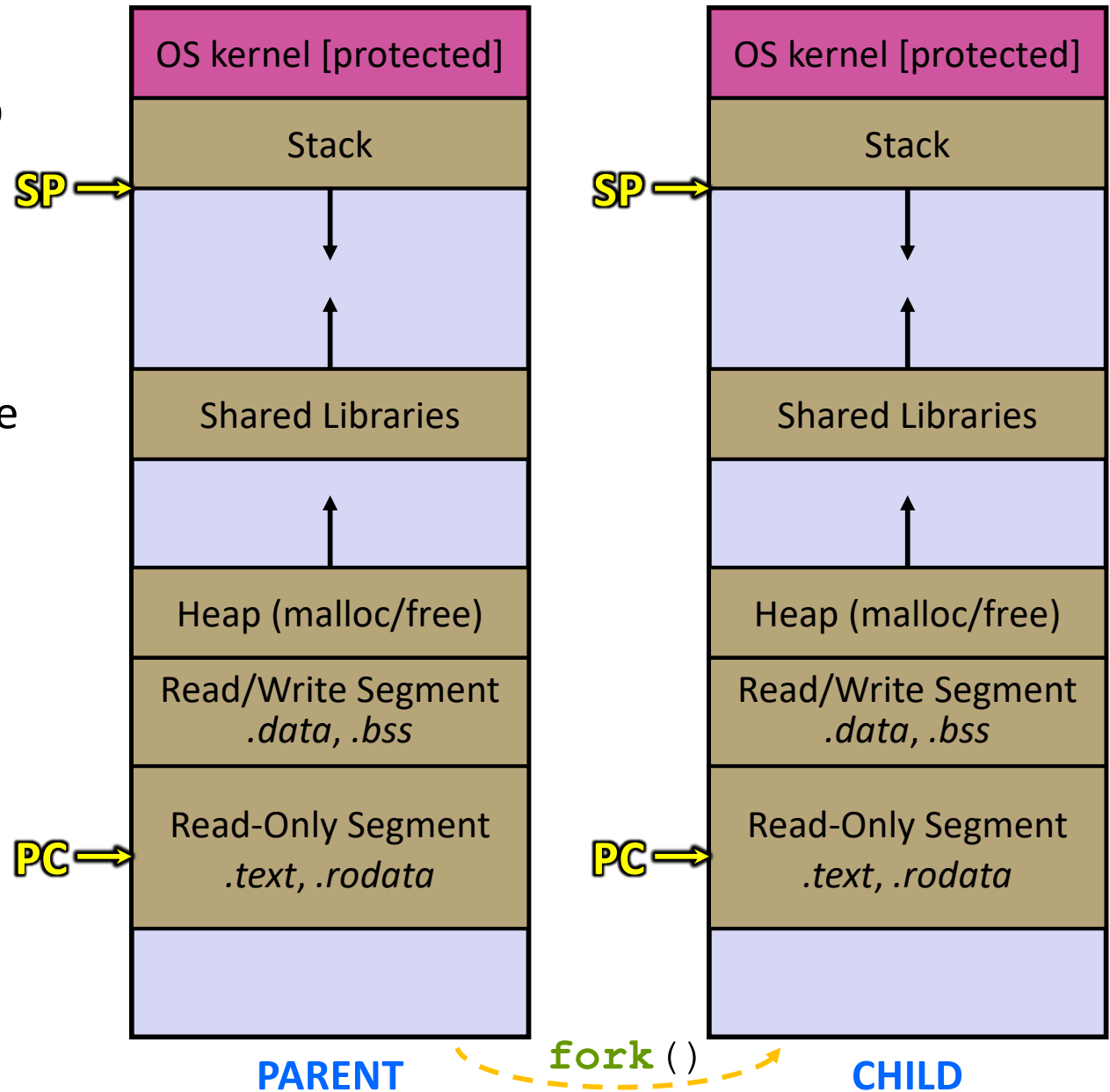
# fork () and Address Spaces

- ❖ 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)



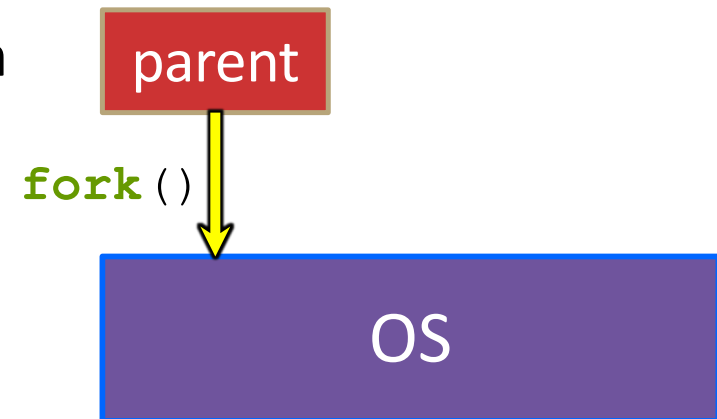
# fork () and Address Spaces

- ❖ 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.



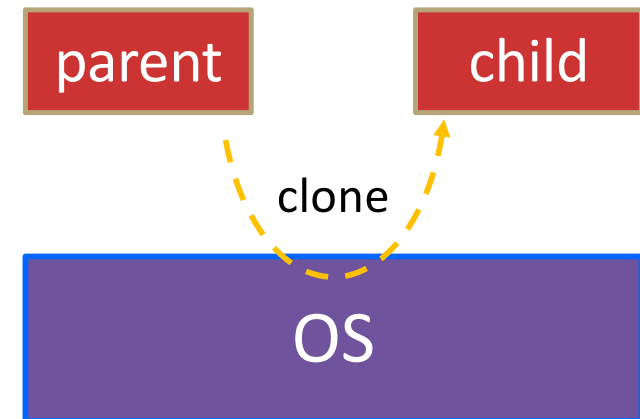
# fork ()

- ❖ **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



# fork ()

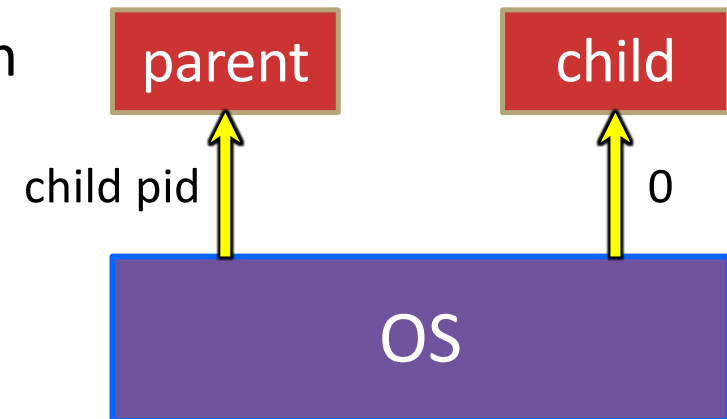
- ❖ **fork ()** has peculiar semantics
  - The parent invokes **fork ()**
  - The OS clones the parent
  - *Both* the parent and the child return from fork
    - Parent receives child's pid
    - Child receives a 0





# 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

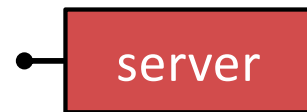


- ❖ See `fork_example.cc`

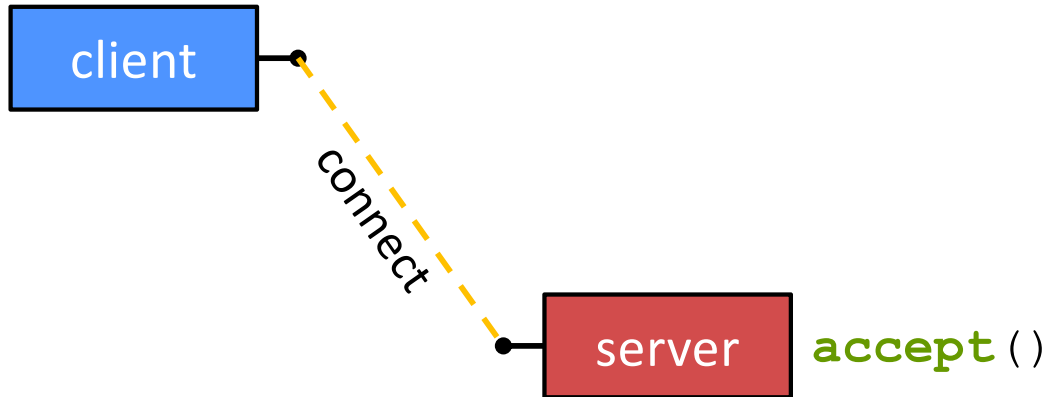
# Concurrent Server with Processes

- ❖ 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 death
  - Option A: Parent calls **wait** ( ) to “reap” children
  - Option B: Use a **double-fork trick**

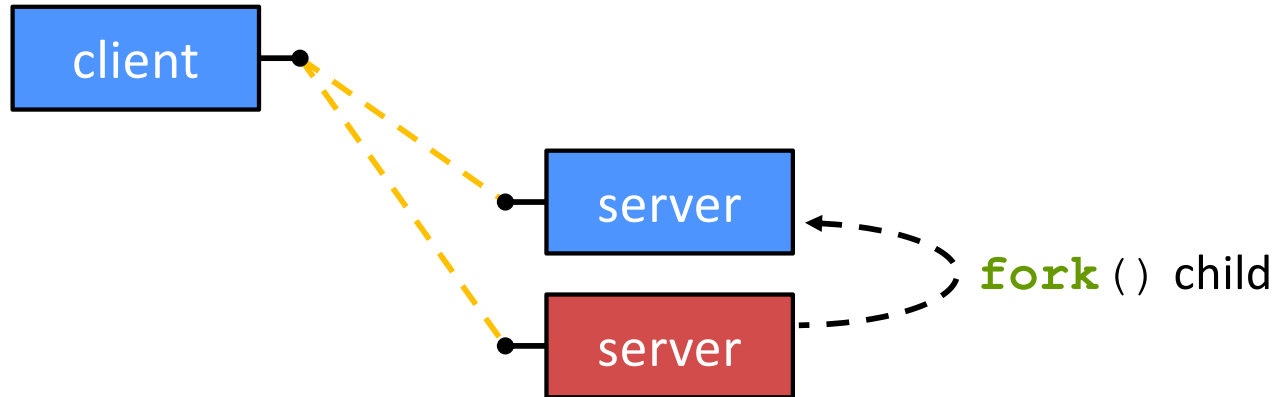
# Double-fork Trick



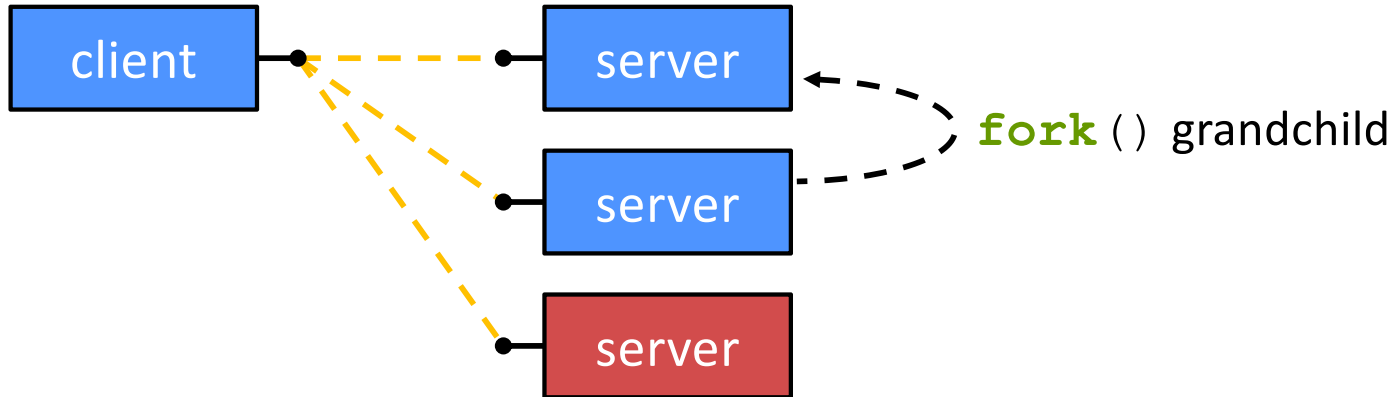
# Double-fork Trick



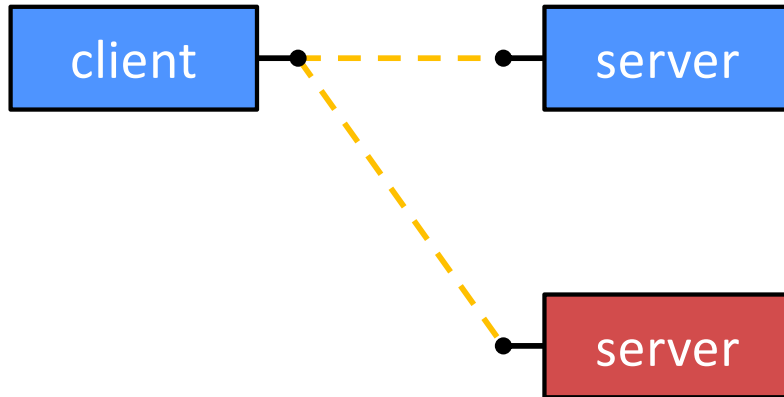
# Double-fork Trick



# Double-fork Trick



# Double-fork Trick



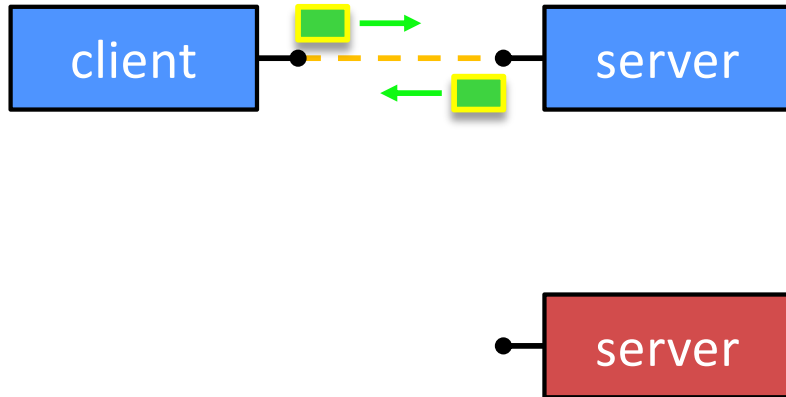
child `exit()`'s / parent `wait()`'s

# Double-fork Trick

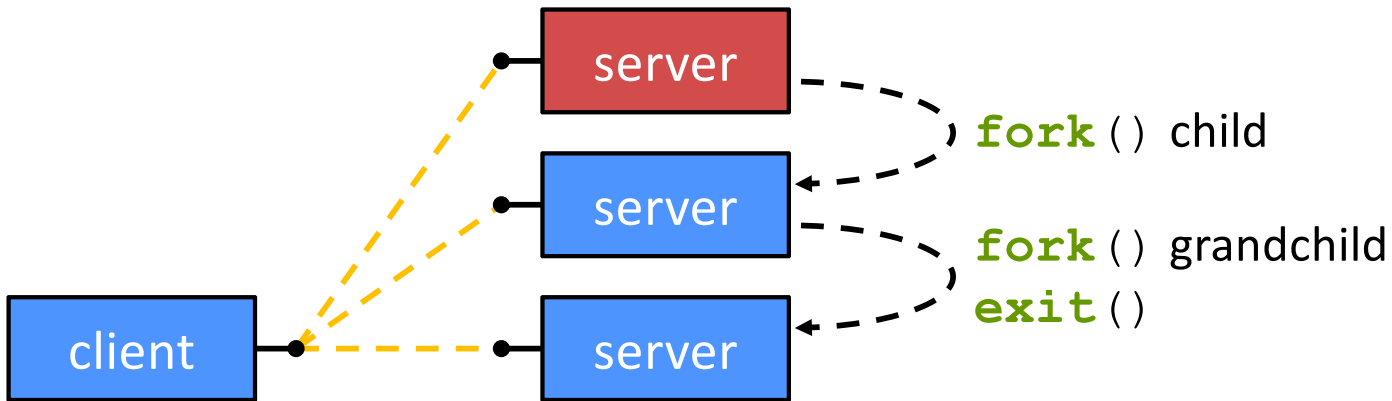
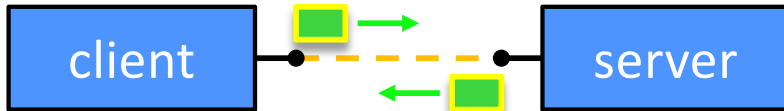




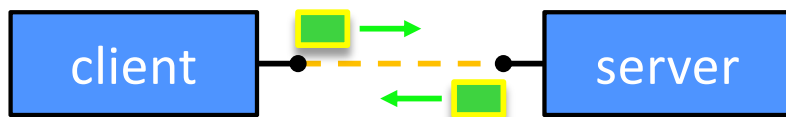
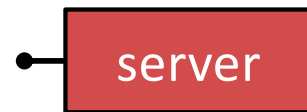
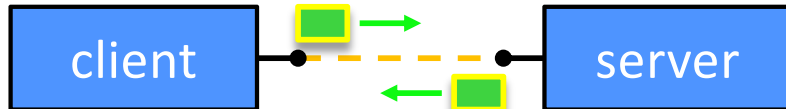
# Double-fork Trick



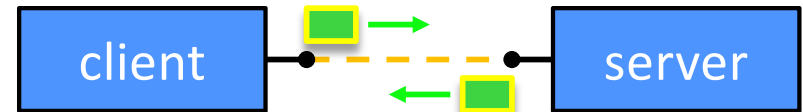
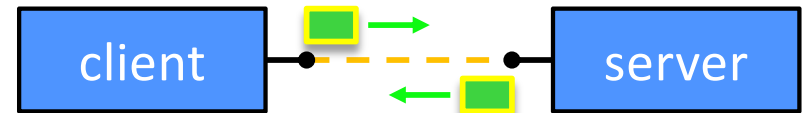
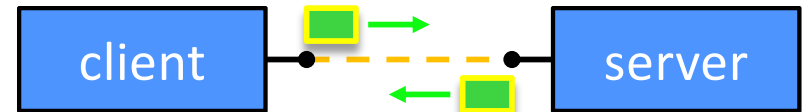
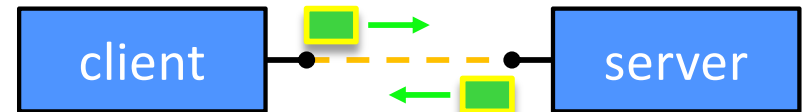
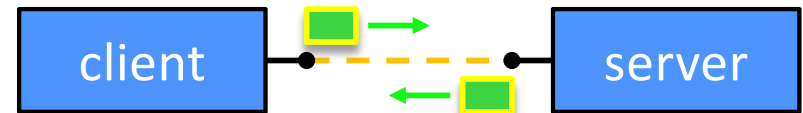
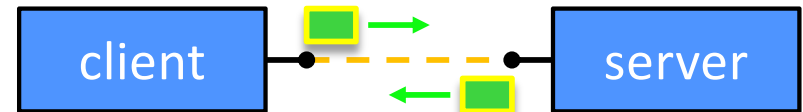
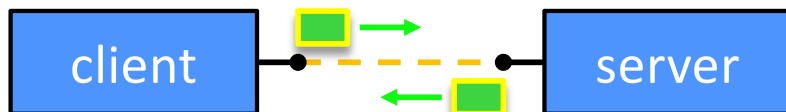
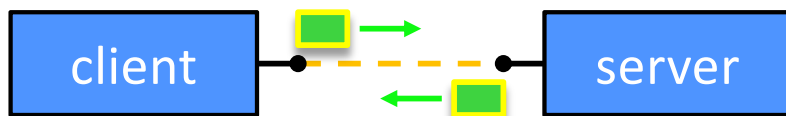
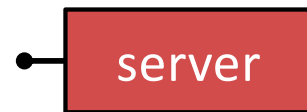
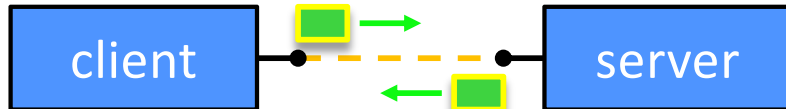
# Double-fork Trick



# 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\***
  - $\therefore$  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](http://threadlatency.cc)
- ❖ **~0.036 ms per thread creation\***
  - ~10x faster than `fork()`
  - $\therefore$  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