Concurrency: Processes
CSE 333

Instructor: Alex Sanchez-Stern

Teaching Assistants:
Justin Tysdal
Sayuj Shahi
Nicholas Batchelder
Leanna Mi Nguyen

Administrivia

- Last exercise due this morning woohoo! 🎉
- hw4 due Wednesday night
  - Usual late days (2 max) apply if you have any remaining
- Final exam Fri. August 16th, 1:10-2:10, SMI 211
  - Topic list on the web; exam will be somewhat weighted towards 2nd half of the quarter
  - Old exams also available on the website.
  - Closed book but you may have two 5x8 cards with handwritten notes
    - Free blank cards available after class

Administrivia

- We'll do course evaluations on Wednesday, bring a pencil
- Section this week is an exam review… show up!

Administrivia

- Extra final points for coming to office hours next week
  - +5 points on the final (out of 100), but can't go above 100 total
  - Must go to an existing, in-person office hours and bring a problem set to work on; either from the extra-problems in the slides, or an old midterm question
  - Make sure the TA writes down your name

Search Server Versions

- 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)

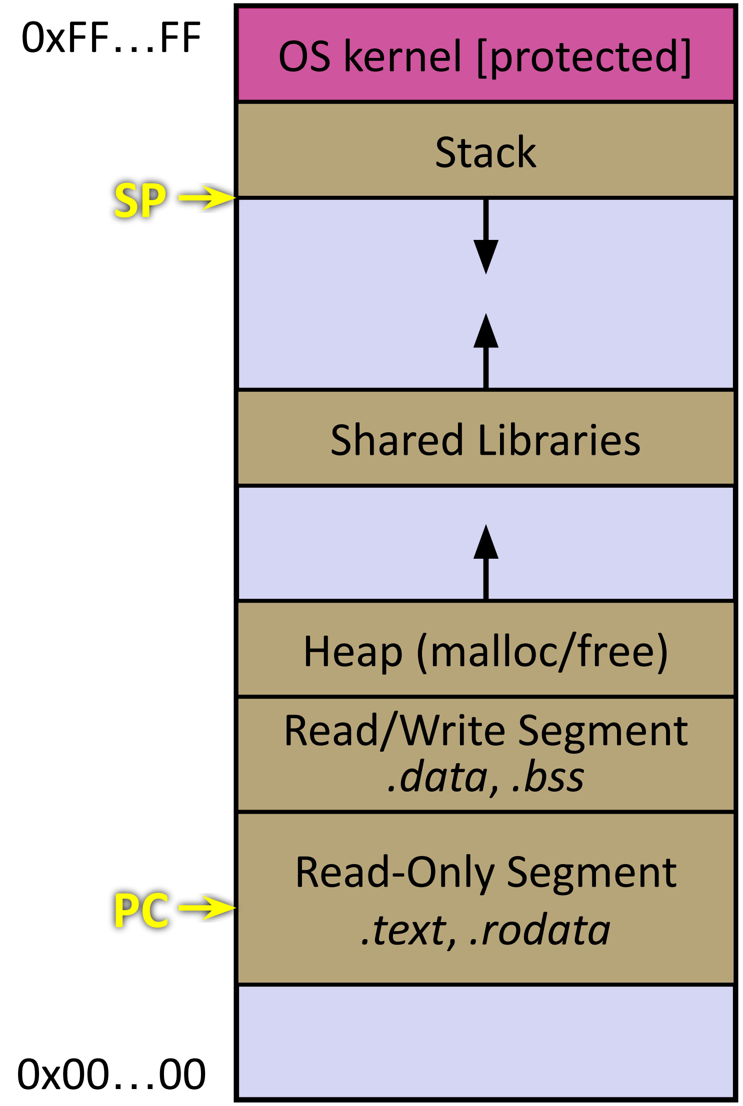
Creating New Processes

```
pid_t fork(void);
```

- Creates a new process (the "child") that is a *clone* of the current process (the "parent")
- Primarily used in two patterns:
  - Adding concurrency to an existing program, for instance a web server
    - Fork a child, then that child executes a subroutine
  - Starting another program, for instance using a shell
    - Fork a child, then that child uses exec to swap it's executable for another.

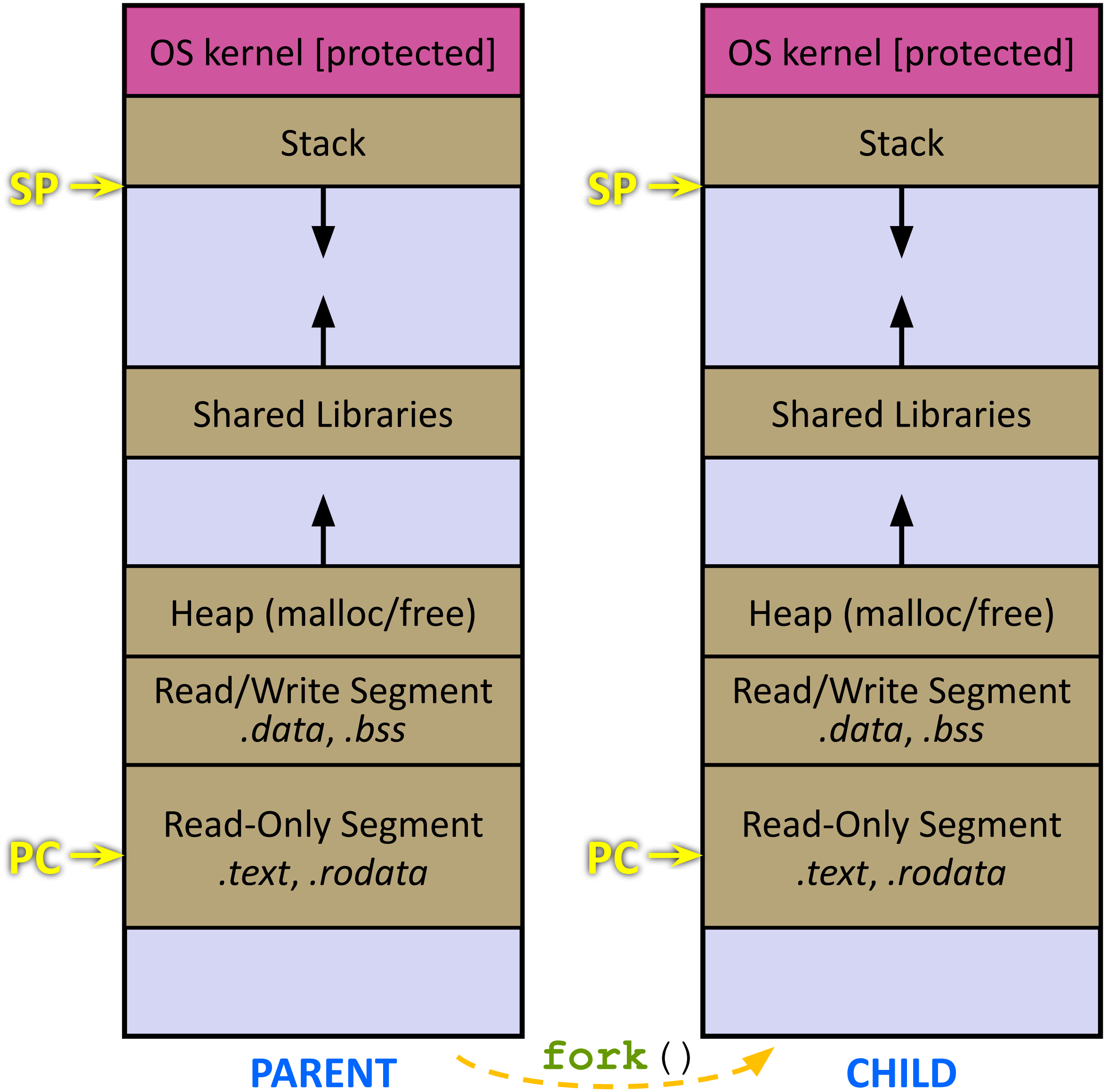
`fork()` and Address Spaces

- 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)



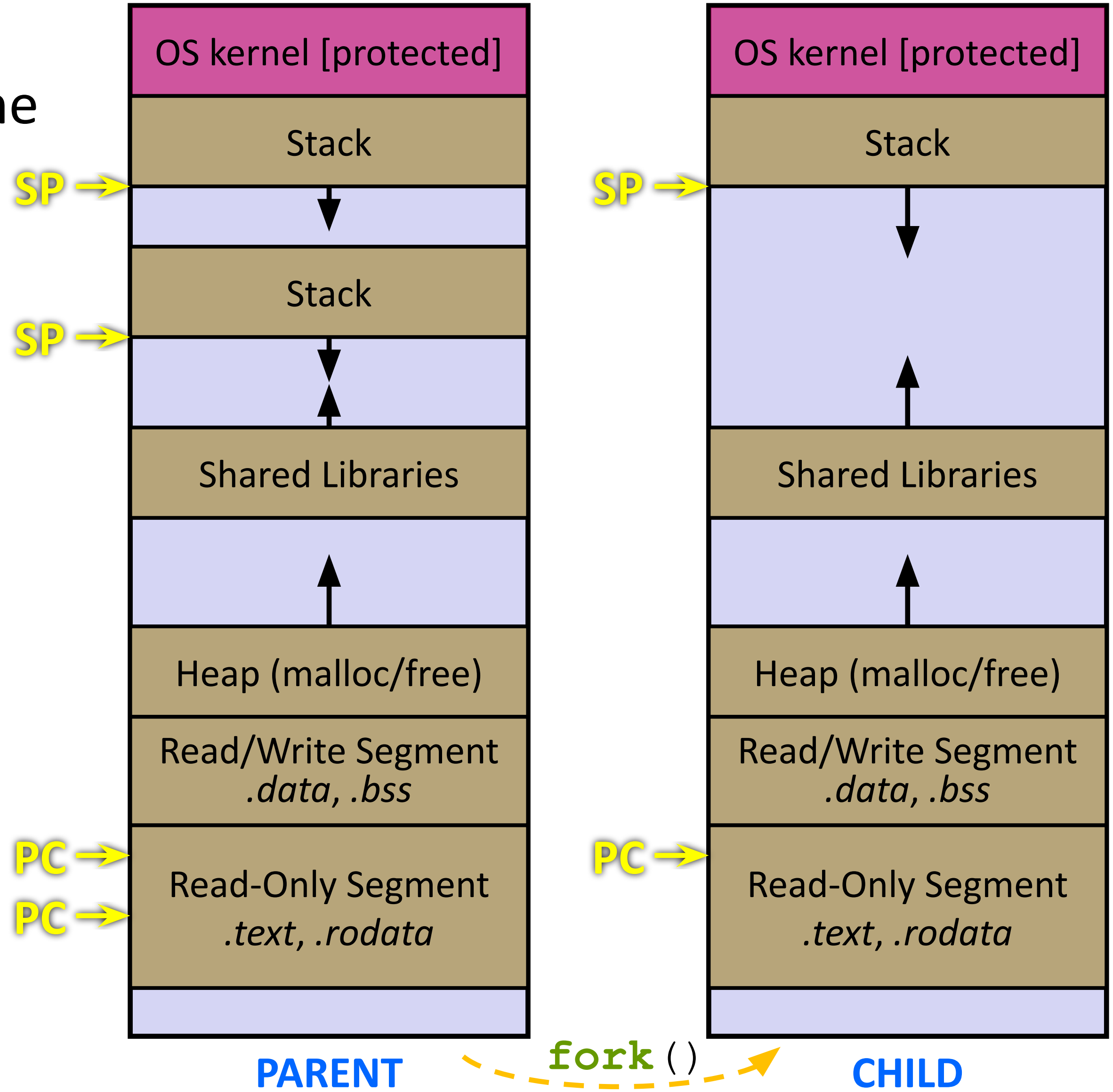
`fork()` and Address Spaces

- Fork cause the OS to clone the address space and registers
  - 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.



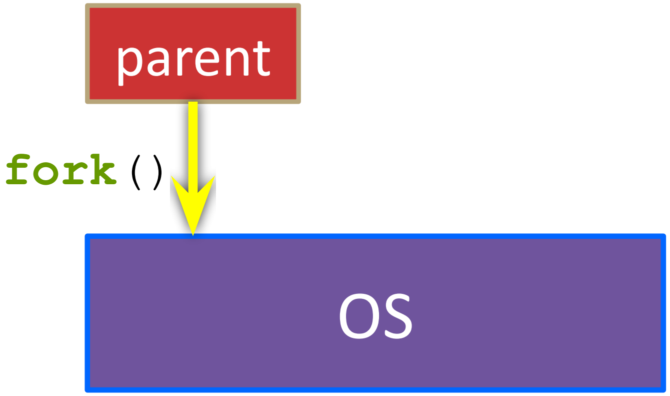
fork() and Address Spaces

- Fork does *not* clone threads
  - Only the thread that called fork is duplicated
  - If the parent had multiple stacks for threads, the child only has one.
  - This can be a source of bugs; try to only use concurrent processes **or** threads, not both.



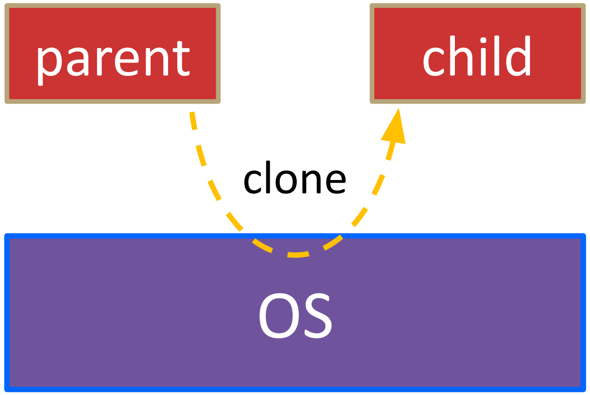
fork()

- fork() has peculiar semantics
  - The parent invokes fork()



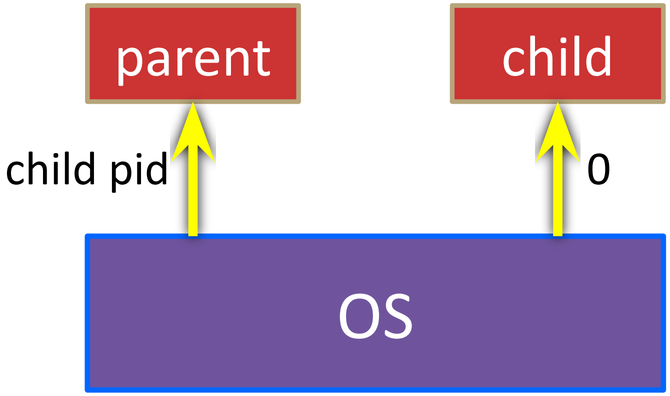
fork()

- fork() has peculiar semantics
  - The parent invokes fork()
  - The OS clones the parent



`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

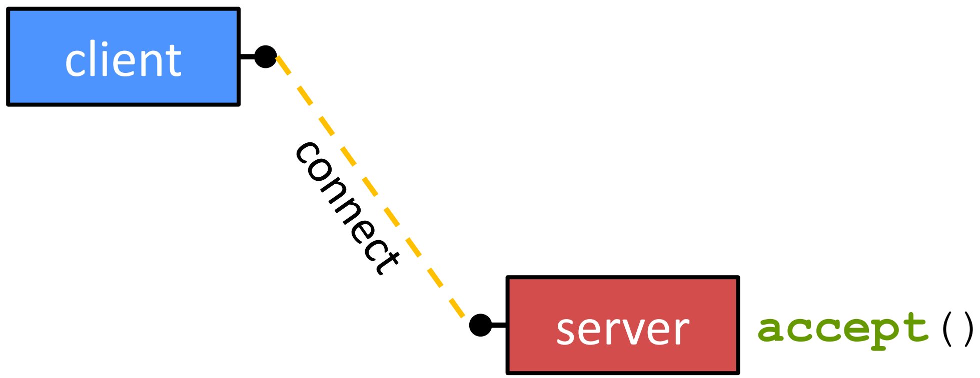
- The **parent** process blocks on `accept()`, waiting for a new client to connect
- When a new connection arrives, the parent calls `fork()` to create a **child** process
- The child process handles that new connection and `exit()`'s when the connection terminates

Concurrent Server with Processes

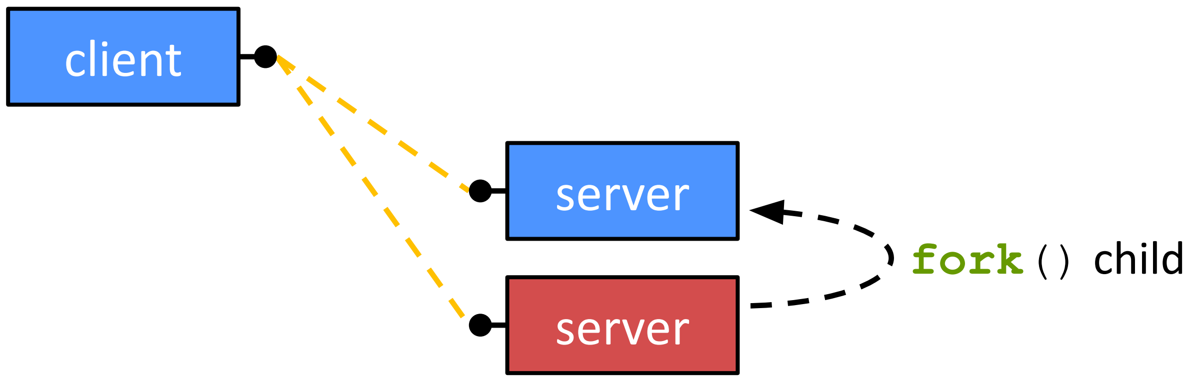
- Remember that children become "zombies" after termination
- The OS is waiting for someone to read their exit code before getting rid of them
- Two ways to handle this:
  - Option A: Parent calls `wait()` to "reap" children and receive their exit codes.
  - Option B: Use the double-fork trick

Double-fork Trick

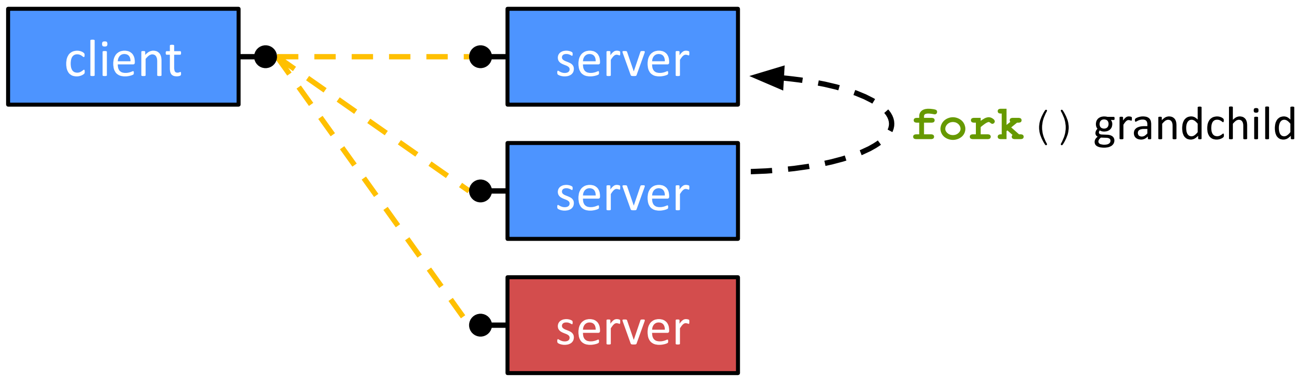
Double-fork Trick



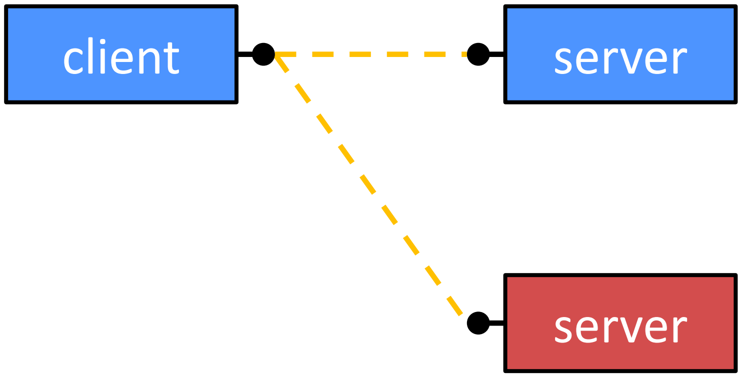
Double-fork Trick



Double-fork Trick

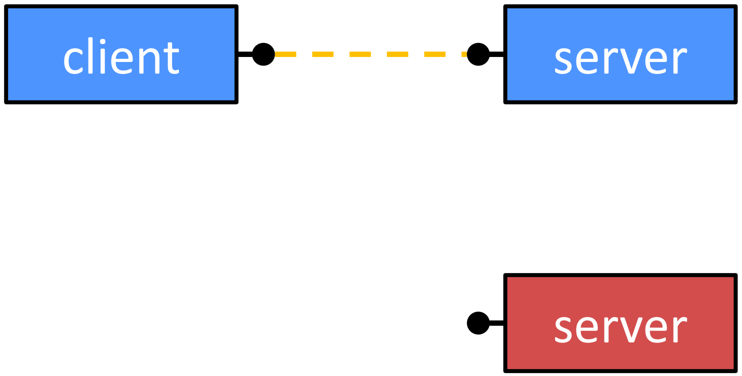


Double-fork Trick



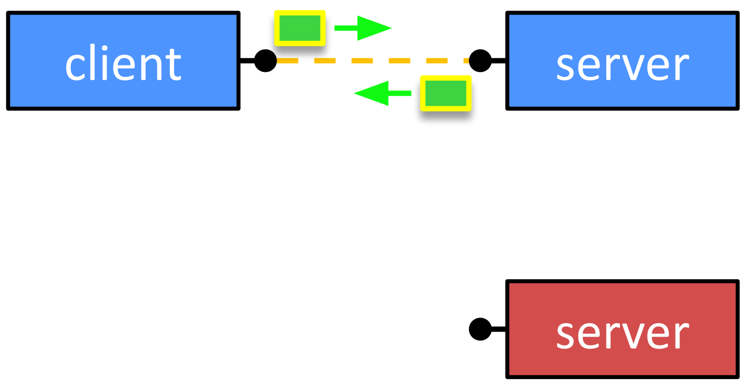
child **exit()**'s / parent **wait()**'s

Double-fork Trick

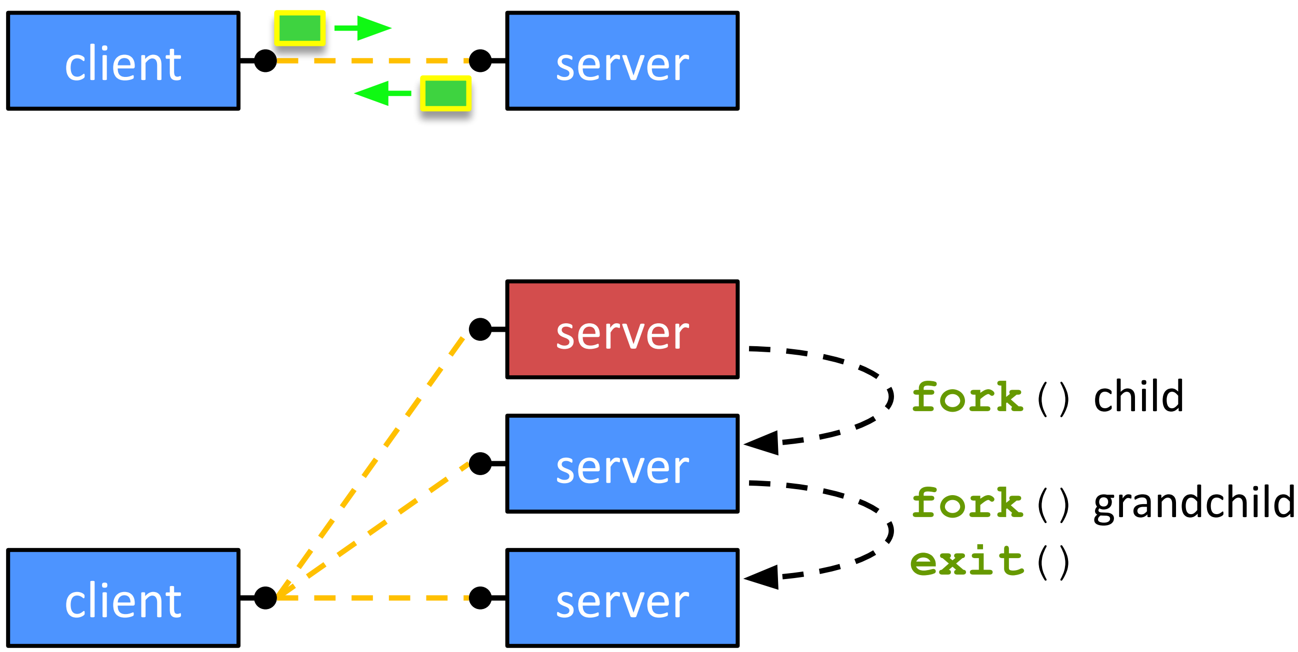


parent closes its
client connection

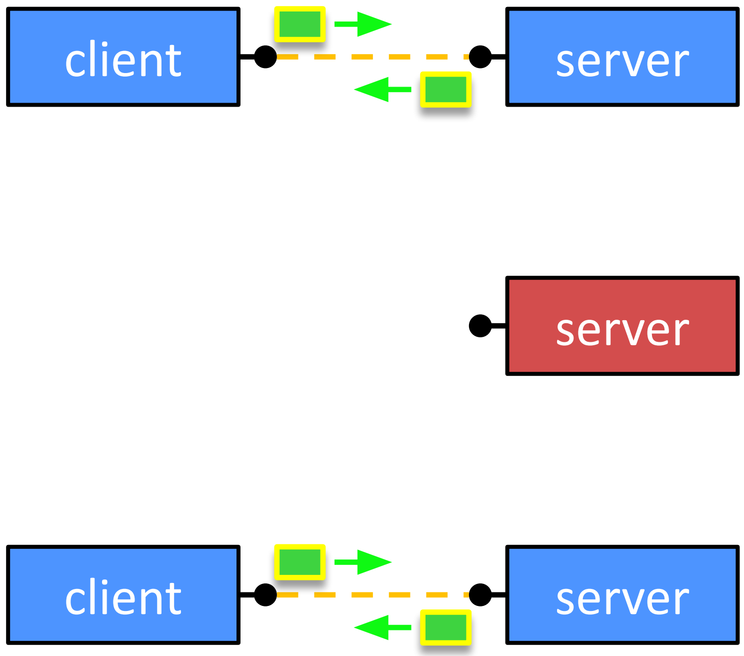
Double-fork Trick



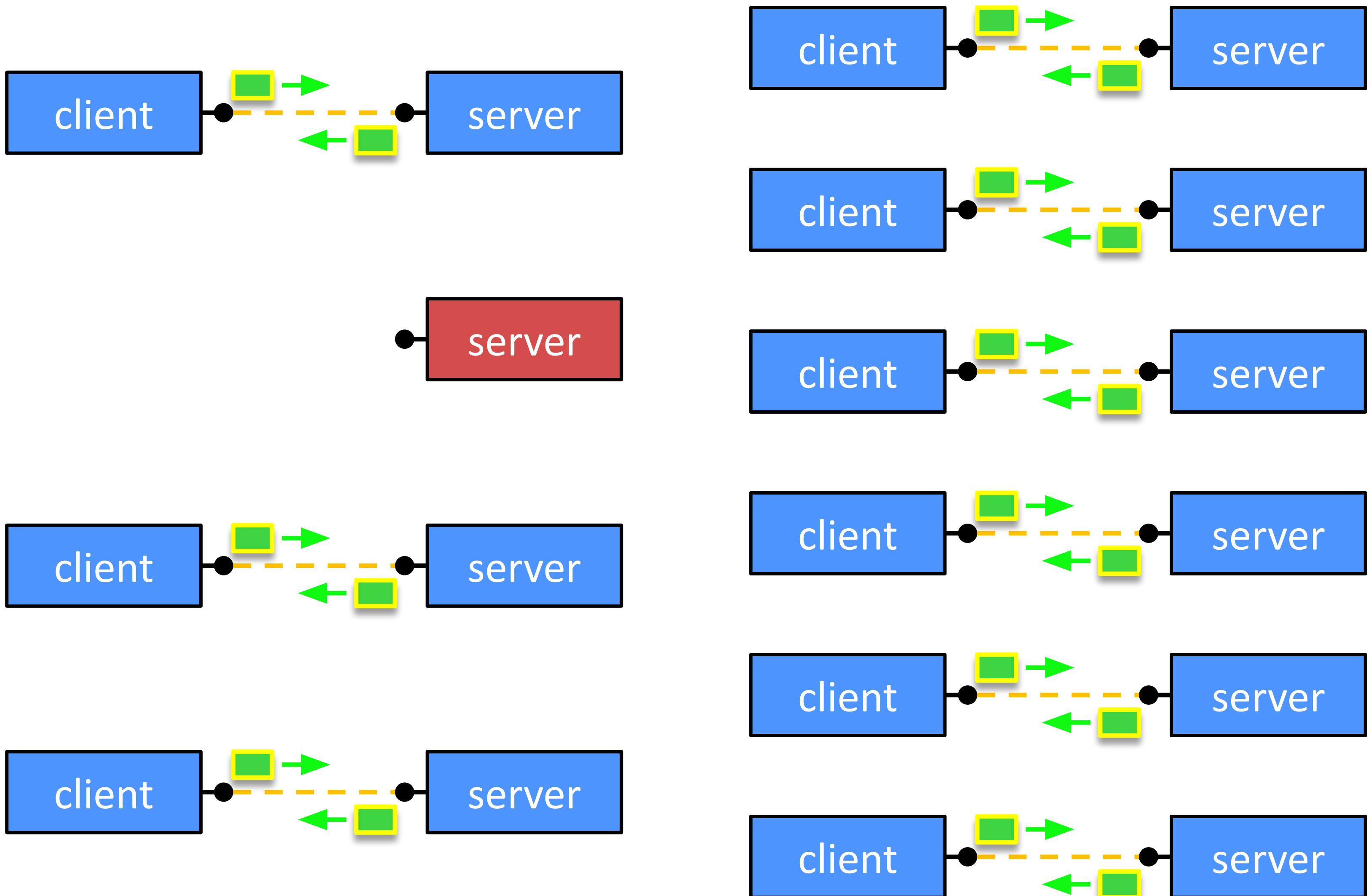
Double-fork Trick



Double-fork Trick



Double-fork Trick



Double-fork Trick

- With the double fork trick:
  - There's no parent to read the exit code
  - Therefore the OS knows to clean it up right away.

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!
  - No need for memory synchronization

- Disadvantages:
  - Processes are heavyweight
    - Relatively slow to fork
    - Context switching latency is high
  - Communication between processes is complicated (and slow)

How Fast is `fork()`?

- See `forklatency.cc`
- **~0.25ms** per fork*
  - Maximum of (1000/0.25) = 4,000 connections/sec/core
  - ~350 million connections/day/core
    - This is fine for most servers
    - Two 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

* Exact past measurements are not indicative of future performance, just their rough ratios - actual measurement depends on hardware and software versions.

How Fast is `pthread_create()`?

- See `threadlatency.cc`
- ~**0.036ms** per fork*
  - Maximum of (1000/0.036) = 28,000 connections/sec/core
  - ~2.4 million connections/day/core
- Much faster, but writing safe multithreaded code is really hard

* Exact past measurements are not indicative of future performance, just their rough ratios - actual measurement depends on hardware and software versions.

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
- Provides faster client connection acceptances and more control over total resource usage.

Don’t Forget

- hw4 due Wednesday night
  - Usual late days (2 max) apply if you have any remaining
- Final exam Fri. August 16th, 1:10-2:10, SMI 211
- Please nominate great TAs for the Bandes award when nominations are available
- We’ll do course evaluations on Wednesday, bring a pencil
- Section this week is an exam review… show up!
- Office hours this week get you extra points on the final