Concurrency and Processes

CSE 333 Spring 2018

Instructor: Justin Hsia

Teaching Assistants:

Danny Allen
Kevin Bi
Michael Poulain
Waylon Huang
Dennis Shao
Jack Xu
Renshu Gu
Wei Lin
Eddie Huang
Matthew Neldam
Robby Marver

Administrivia

- hw4 due tomorrow (5/31)
  - Submissions accepted until Sunday (6/3)

- Final is Tuesday (6/5), 12:30-2:20 pm, KNE 120
  - Review Session: Sunday (6/3), 4-6:30 pm, EEB 125
  - *Two* double-sided, handwritten sheets of notes allowed
  - Topic list and past finals on Exams page on website

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 ☹

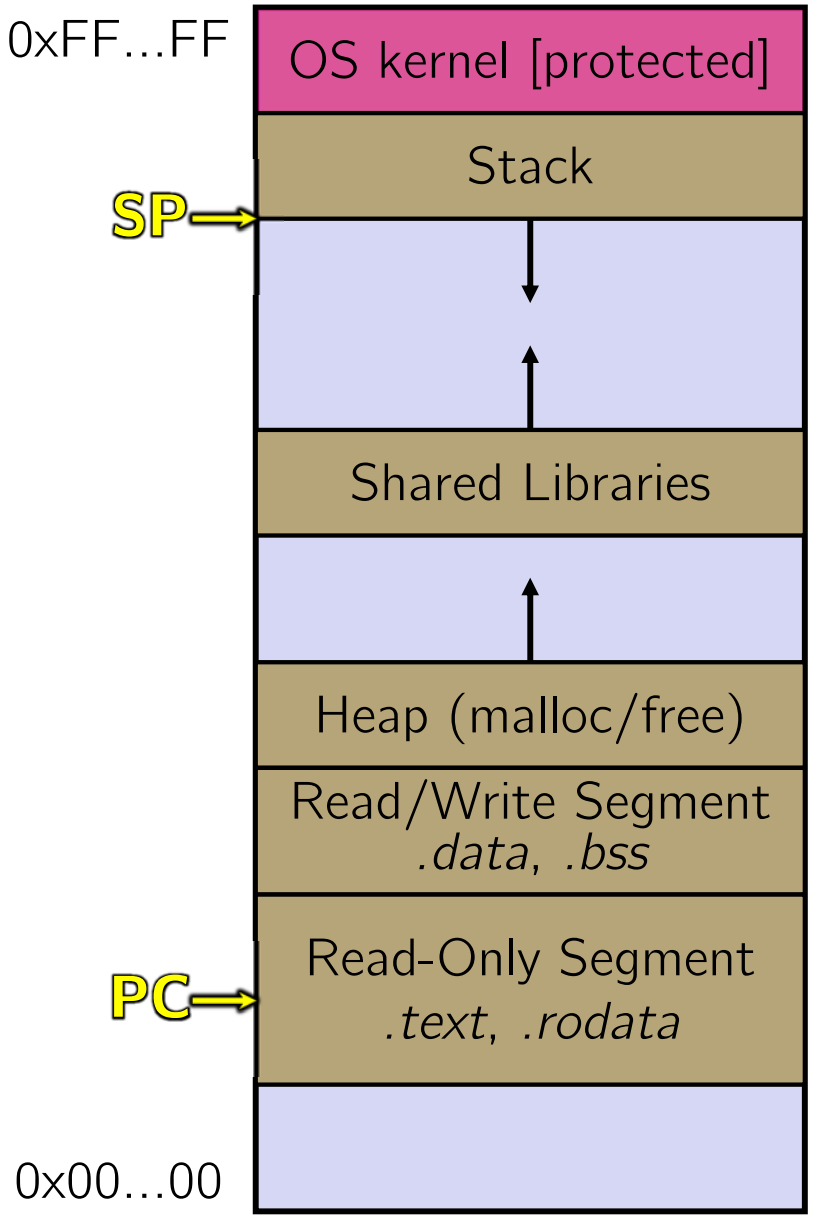
- Reference: *CSPP*, Chapter 12

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, 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

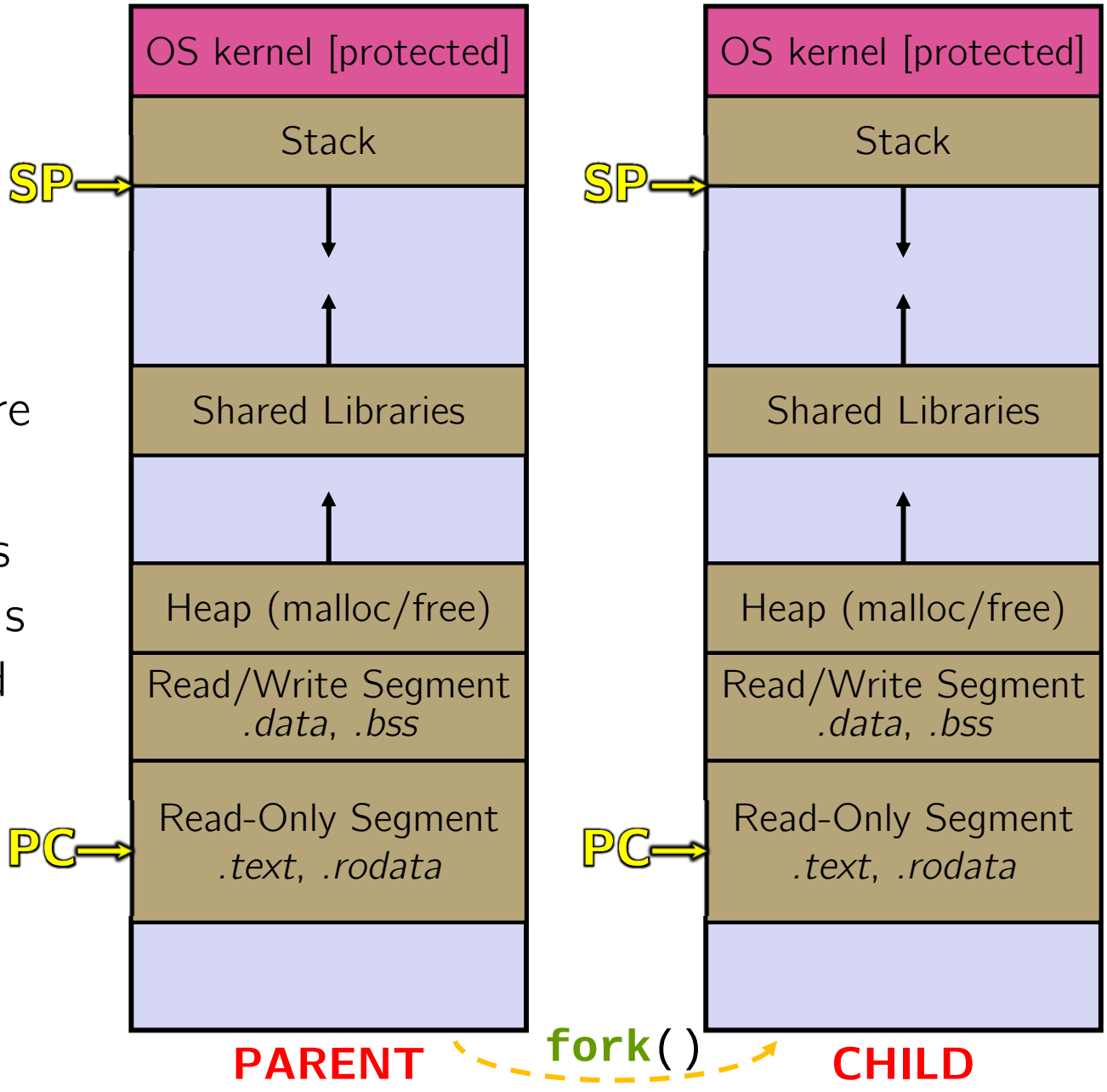
fork() and Address Spaces

- 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)



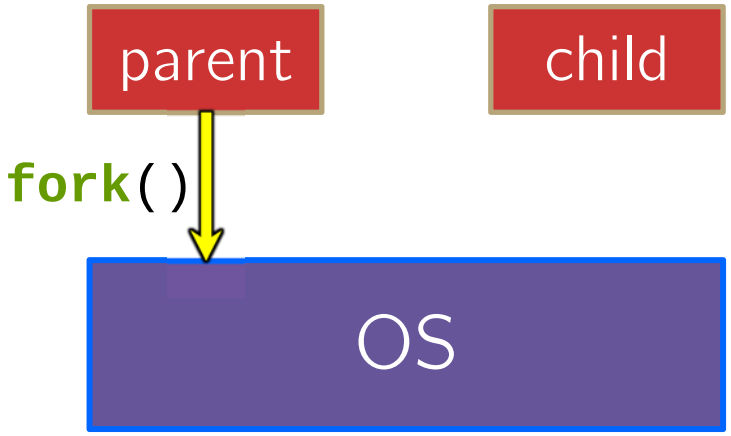
fork() and Address Spaces

- 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.



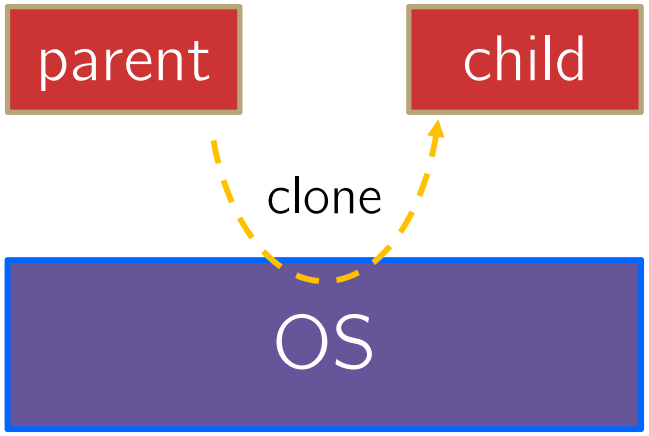
fork()

- **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



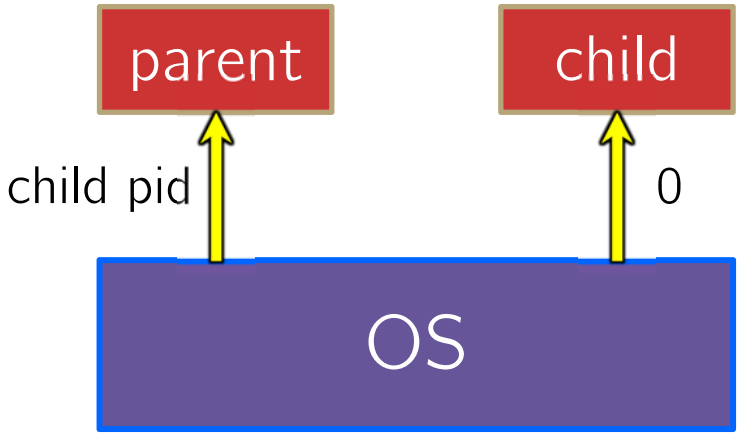
fork()

- **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`

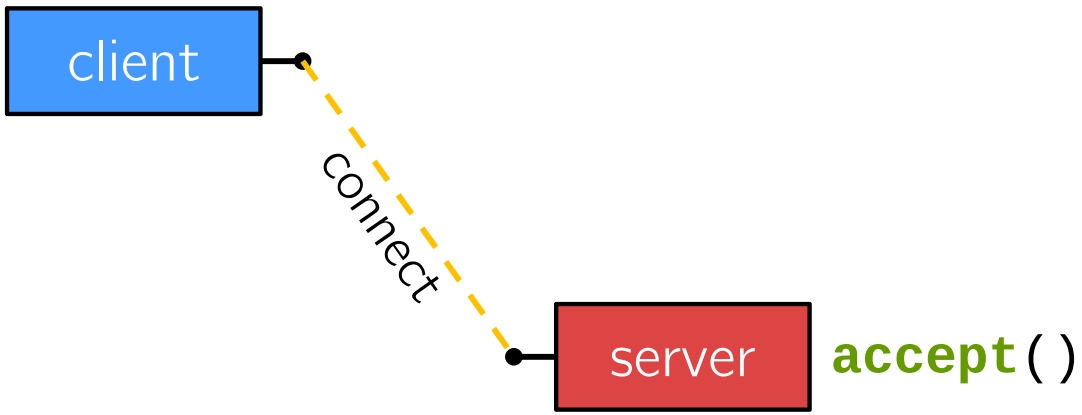
Concurrency with Processes

- 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

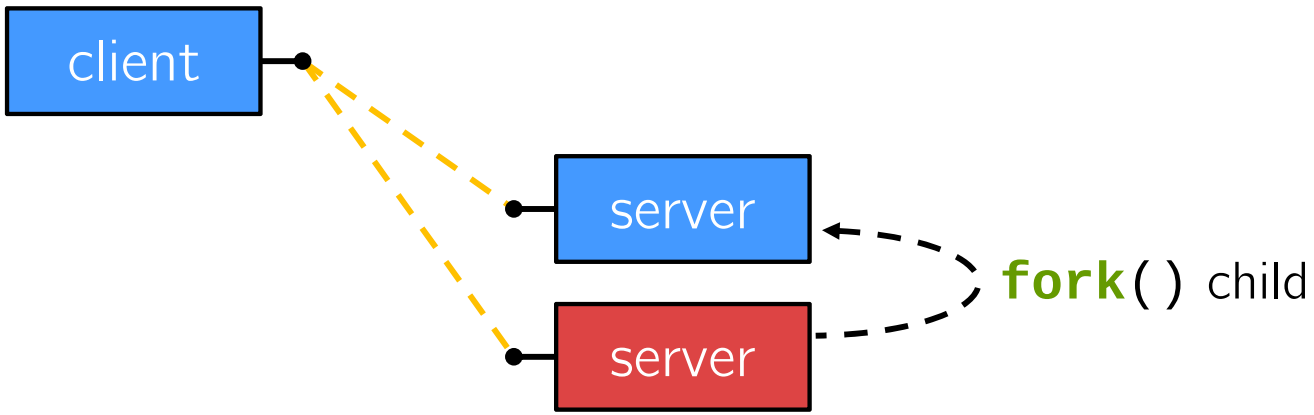
- Remember that children become "zombies" after death
  - Option A: Parent calls **`wait()`** to "reap" children
  - Option B: Use a double-fork trick

Double-fork Trick

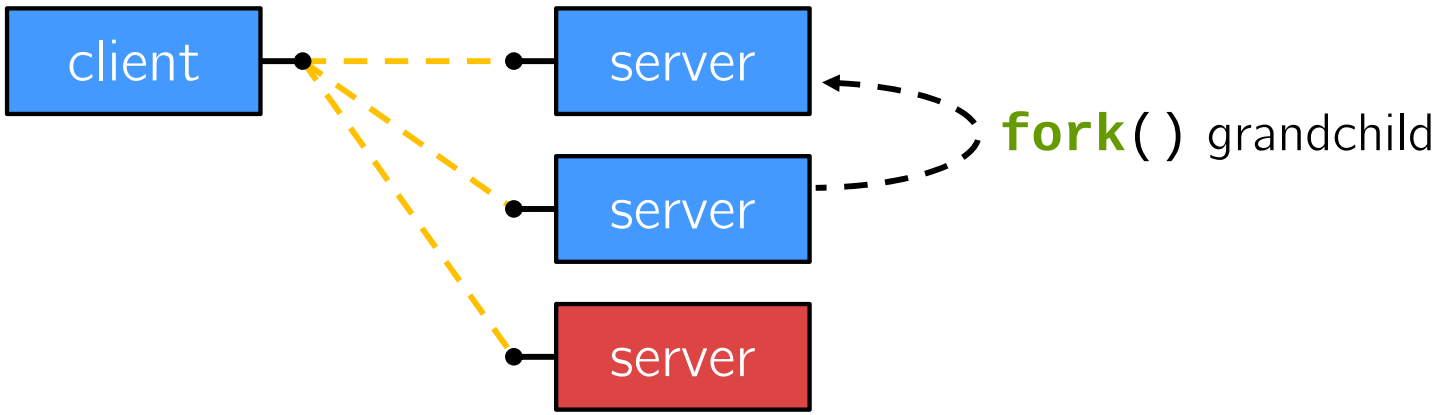
Double-fork Trick



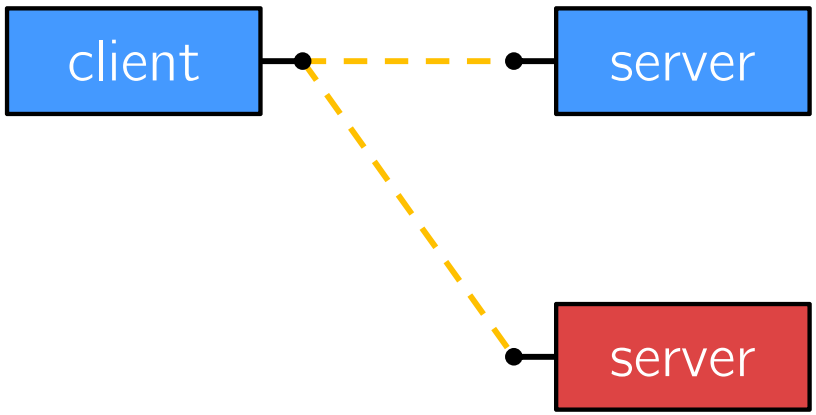
Double-fork Trick



Double-fork Trick



Double-fork Trick

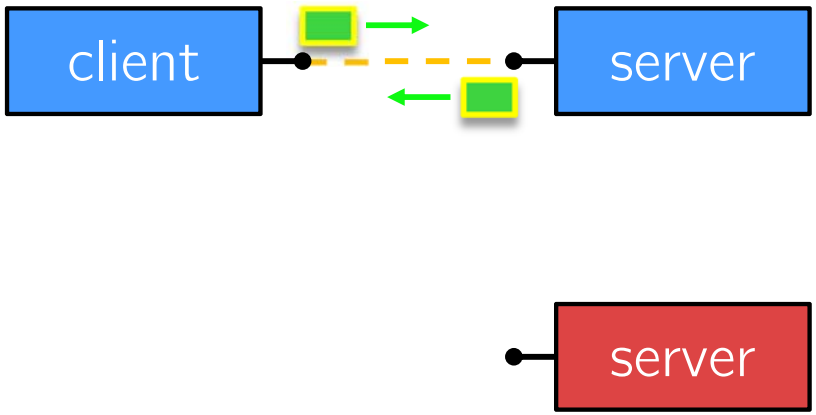


child `exit()`'s / parent `wait()`'s

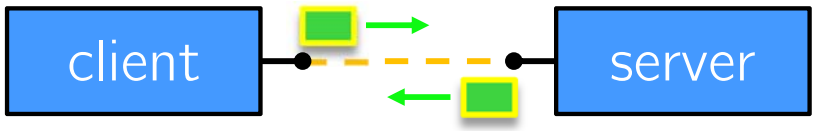
Double-fork Trick

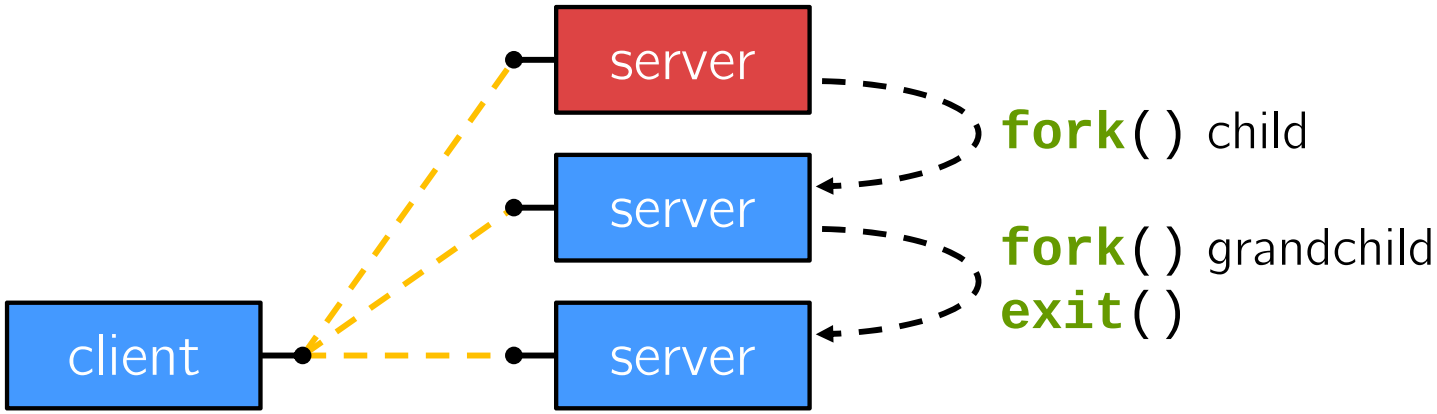
parent closes its client connection

Double-fork Trick

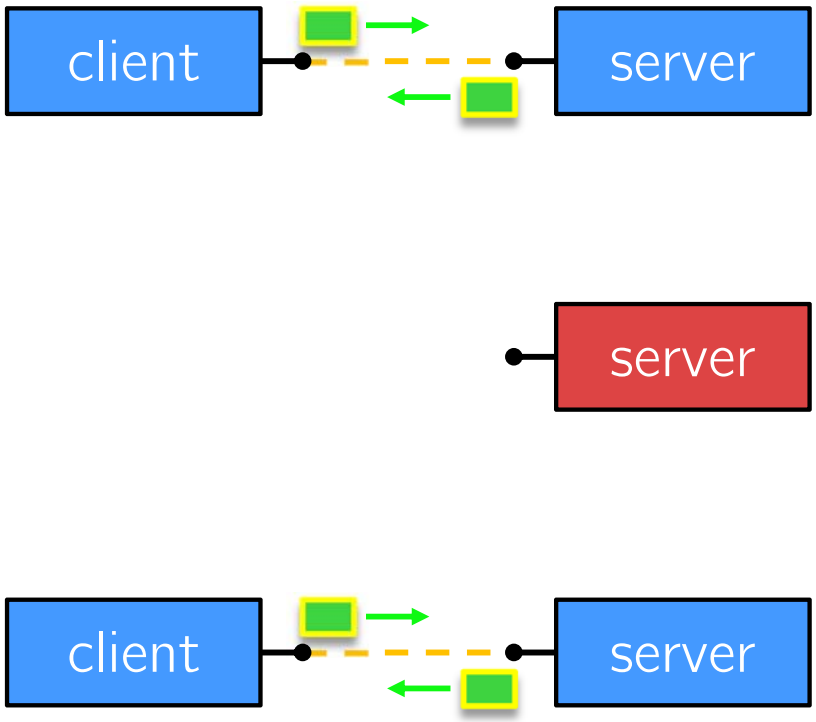


Double-fork Trick

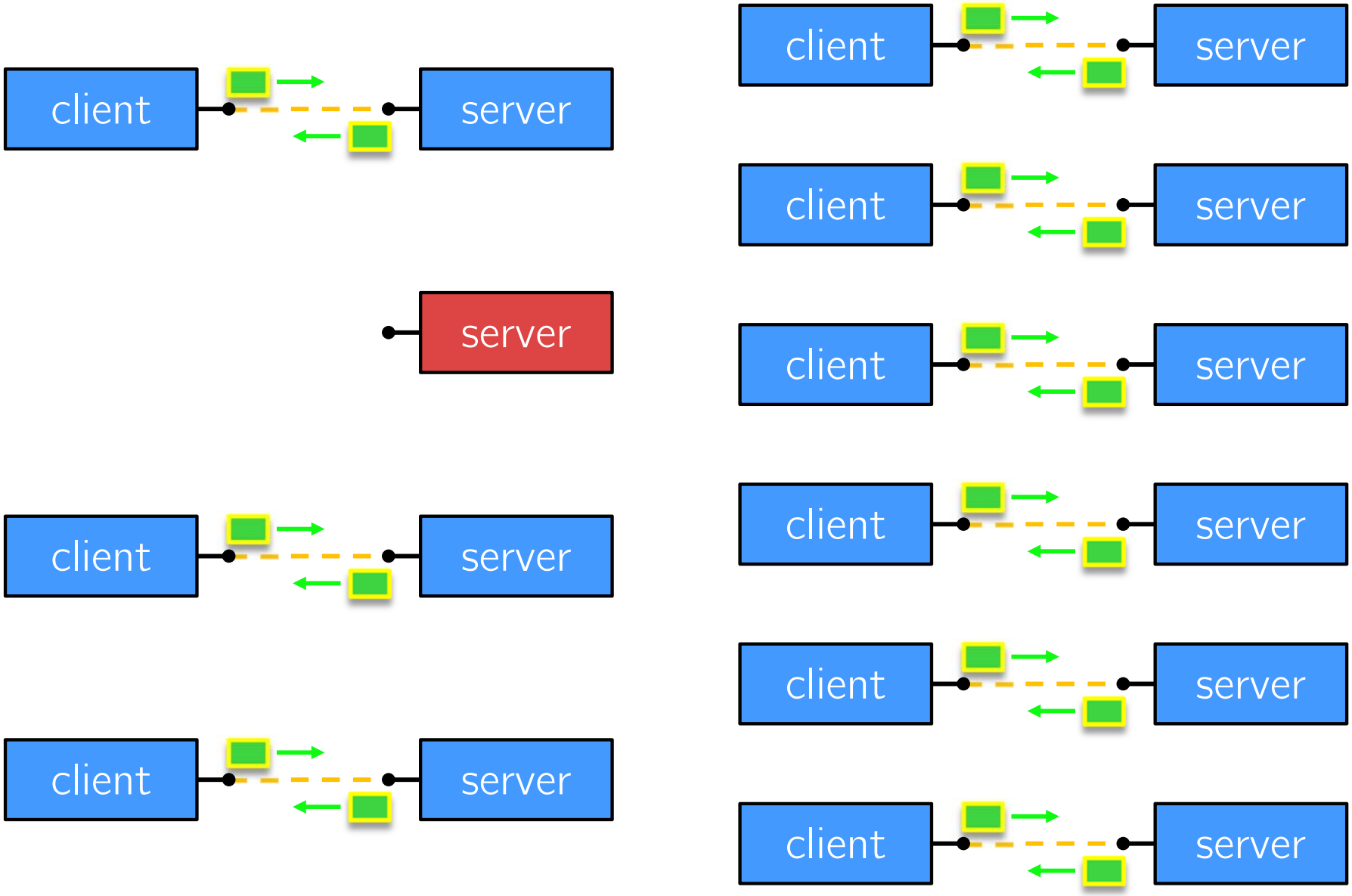




Double-fork Trick



Double-fork Trick



Peer Instruction Question

- What will happen when one of the grandchildren processes finishes?

A. Zombie until ~~grand~~parent exits

B. Zombie until ~~grand~~parent reaps

C. Zombie until init reaps

D. ZOMBIE FOREVER!!!

E. We're lost…

only works for parent-zombie child

wait() only works for children grandparent doesn't generally have pid of grandchild for waitpid()

init/systemd is the "mother of all processes" and cleans up orphaned children processes

double-fork trick lets parent block for very short periods (until child exits) while init handles clean up of grand children when client connections close

Concurrent with Processes

- See `searchserver_processes/`

pseudo code:

```
while (1) {
  ...
  sock_fd = accept();

  pid = fork();
  if (pid == 0) {
    //child
    pid = fork();
    if (pid == 0) {
      //grandchild
      HandleClient();
      //clean up
      return 0;
    }
    //clean up
    return 0;
  }
  waitpid(pid);
  close(sock_fd);
}
```

Why 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

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

- Mush faster, but writing safe multithreaded code can be serious voodoo

Aside: Thread Pools

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

- 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