Concurrency: Processes

CSE 333 Summer 2020

Instructor: Travis McGaha

Teaching Assistants:

Jeter Arellano Ramya Challa Kyrie Dowling
Ian Hsiao Allen Jung Sylvia Wang

Poll Everywhere

pollev.com/cse33320su

About how long did Exercise 17 take?

A. 0-1 Hours
B. 1-2 Hours
C. 2-3 Hours
D. 3-4 Hours
E. 4+ Hours
F. I didn't submit / I prefer not to say

Side question:
What's your favourite CSE course so far?

Administrivia

- hw4 due Thursday (8/20)
  - Submissions accepted until Sunday (8/23) @ 11:59 pm
  - If you want to use late day(s), you **MUST** let staff know. Make a private post on ed or send an email to staff letting us know you want to use late day(s).

- Course evaluations!
  - Please fill them out. They help staff members improve our skills as educators and allow us to improve the course for future offerings. ☺

- Grades for various assignments have been posted. **PLEASE CHECK THESE** and contact staff if something seems incorrect!!!

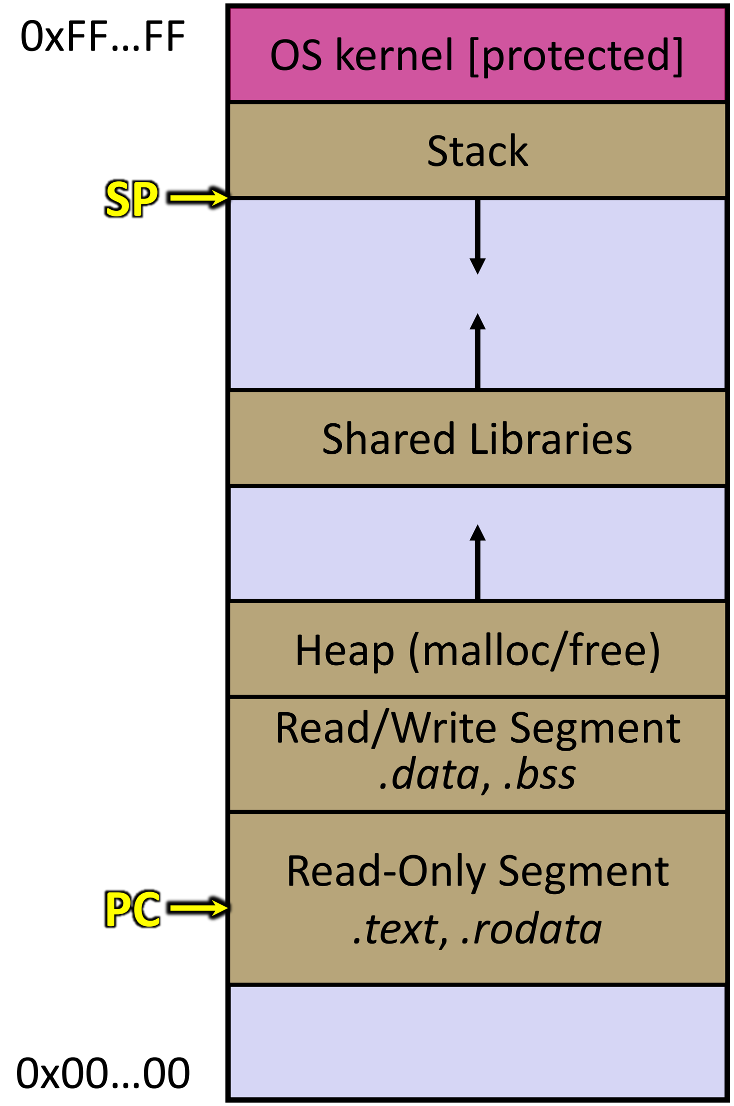
Outline

- `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)

Review: 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)

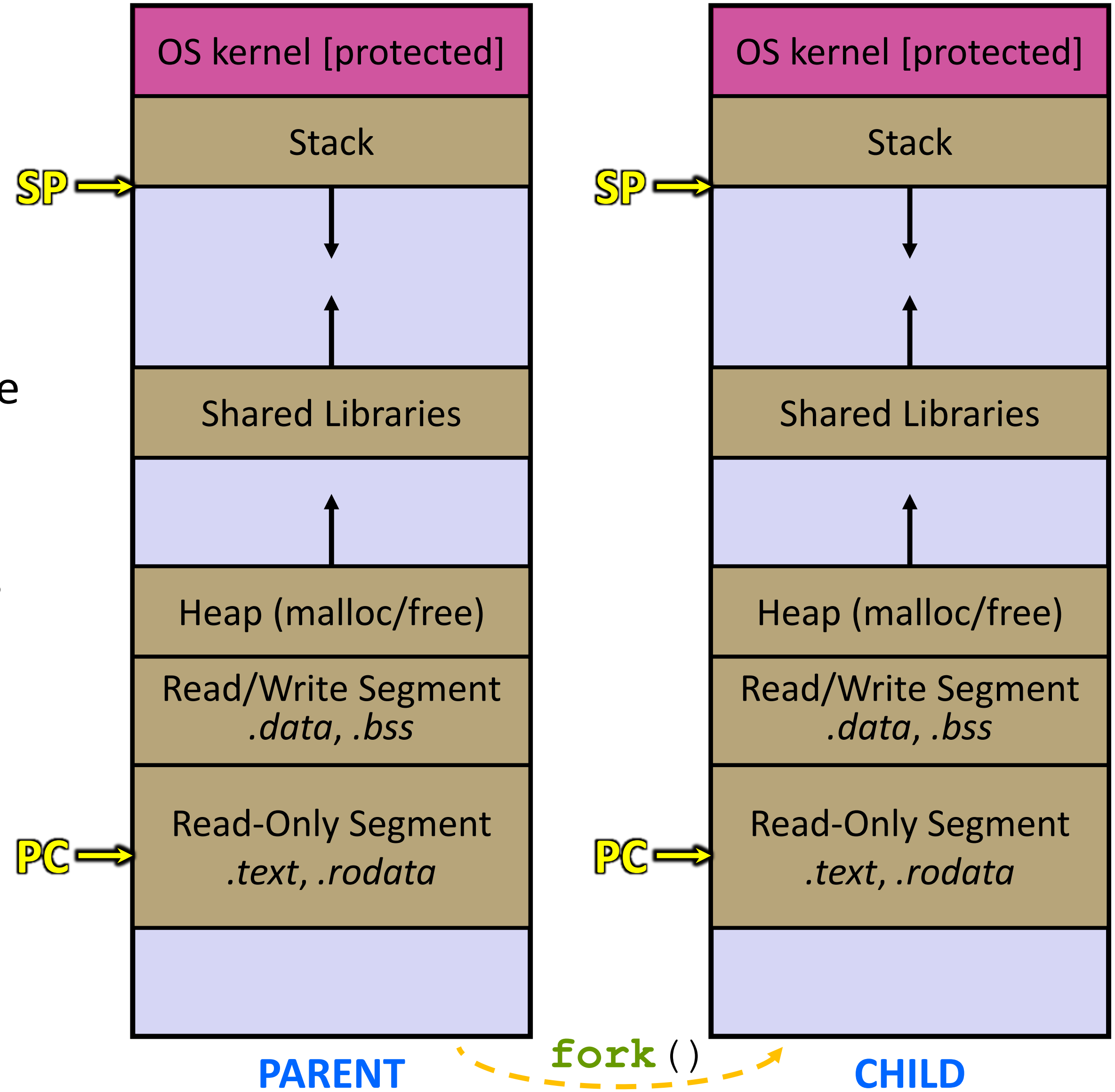


Creating New Processes

- `pid_t fork();`
  - Creates a new process (the "child") that is an *exact clone** of the current process (the "parent")
    - *Everything is cloned except threads. Sockets, file descriptors, virtual address space, variables, etc.
  - The new process has a separate virtual address space from the parent

`fork()` and Address Spaces

- Fork causes 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.

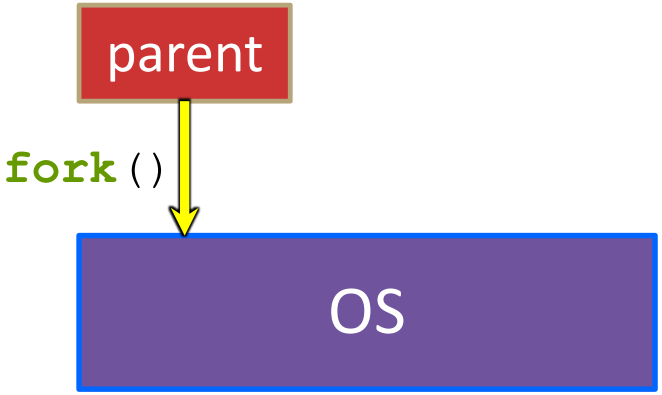


Main Uses of `fork()`

- Fork a child to handle some work
  - Server forks to handle a new connection
  - Web browser forks to render a new website
    - Mainly for security purposes (separate address spaces)
- Fork a child that then exec's a new program
  - Shell forks and execs the program you want to run
  - 333 grading script forks and execs your executable
    - Using Python `subprocess`

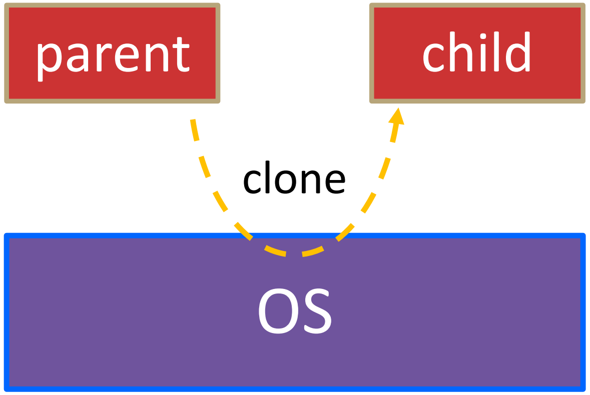
`fork()`

- `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



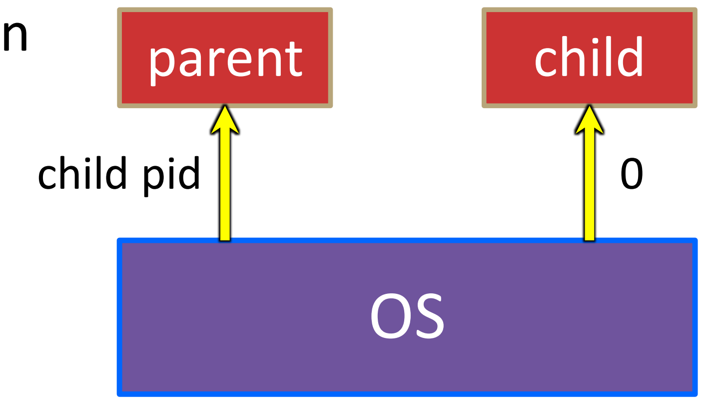
`fork()`

- `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

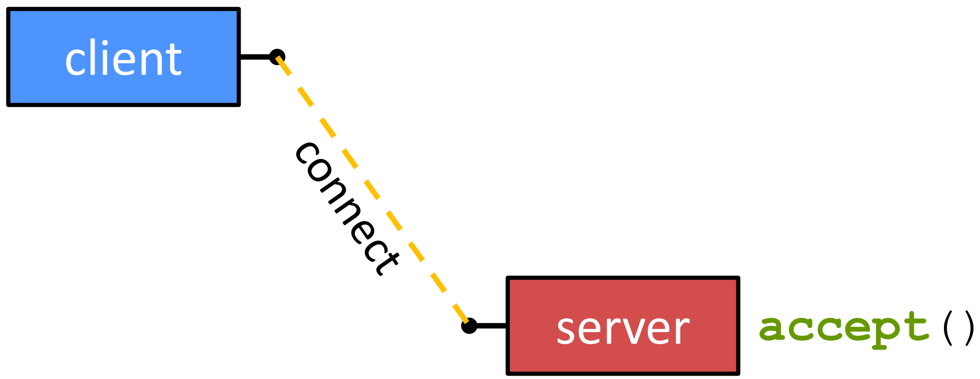
CSE 351 review:
When a child process exits, it is a zombie until its parent reaps it.

Concurrent Server with Processes

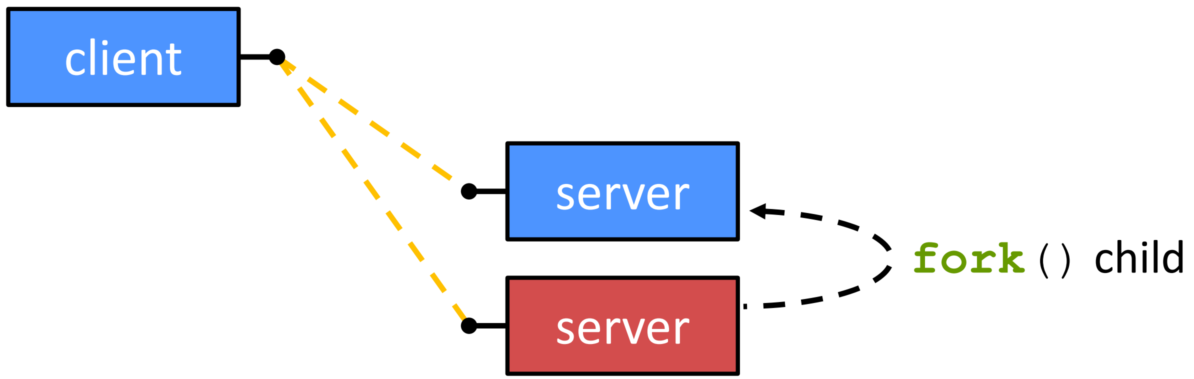
- The **parent** process blocks on `accept()`, waiting for a new client to connect
  - When a new connection arrives, the parent calls `fork()` to create a **child** process
  - The child process handles that new connection and `exit()`'s when the connection terminates
- Remember that children become "zombies" after death
  - Option A: Parent calls `wait()` to "reap" children
  - Option B: Use a double-fork trick

Double-fork Trick

Double-fork Trick

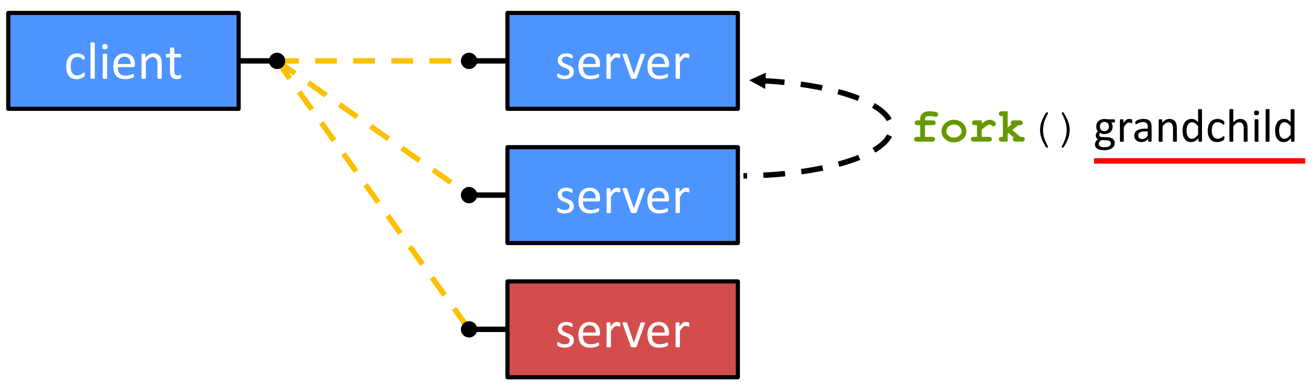


Double-fork Trick

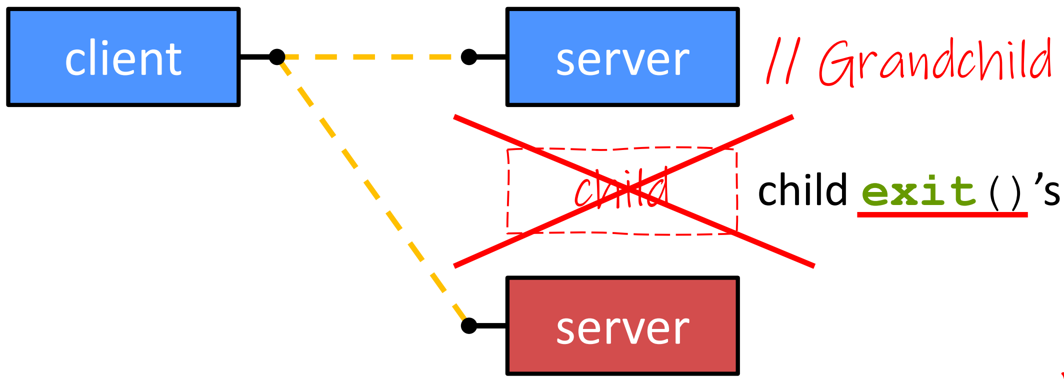


Reminder:
Fork() copies the file descriptor table from parent, so the child has connection to the client too.

Double-fork Trick



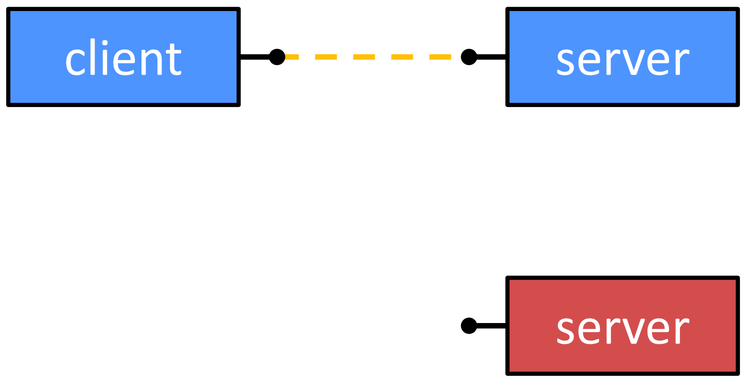
Double-fork Trick



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

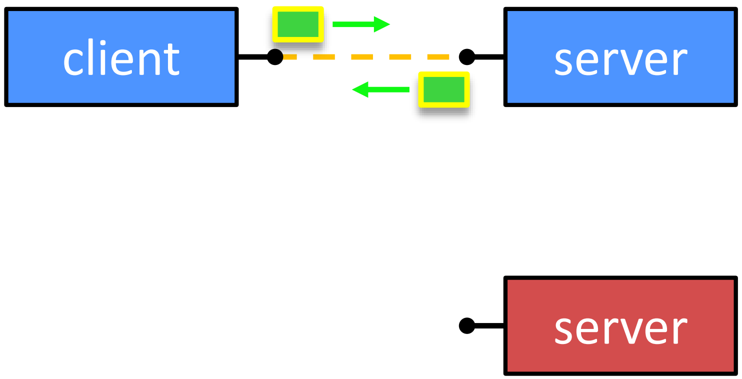
When parent wait()'s for child, the child will be cleaned up

Double-fork Trick

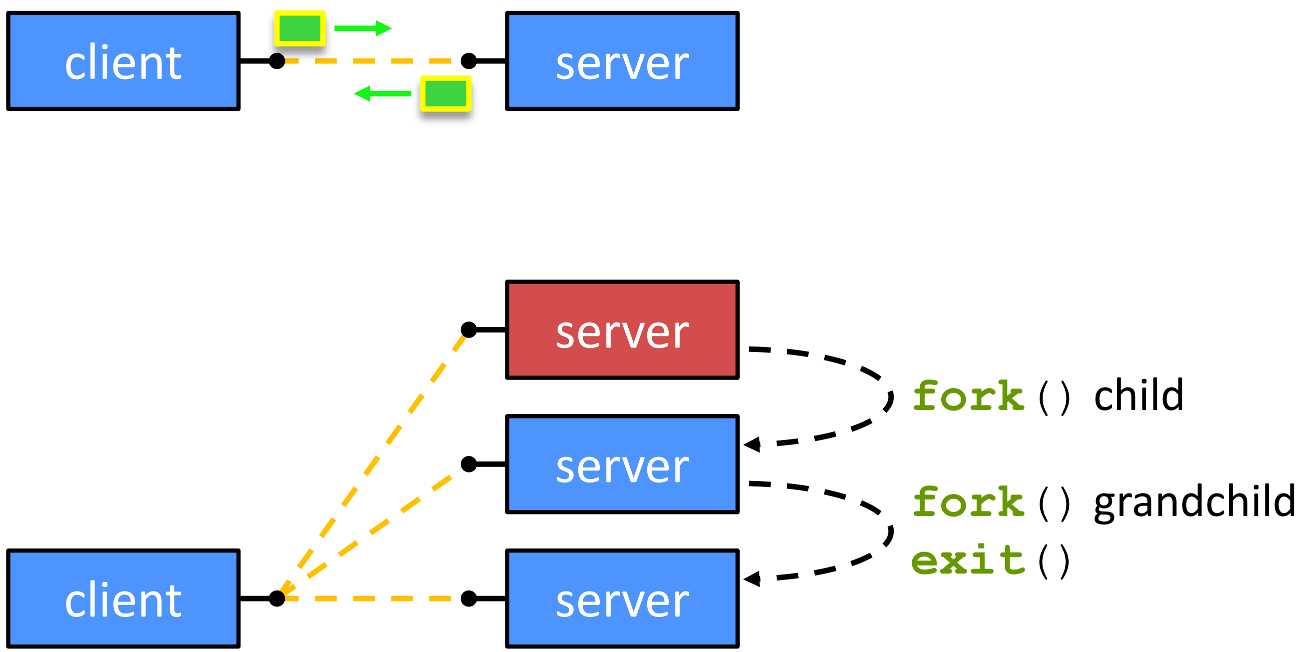


parent closes its
client connection

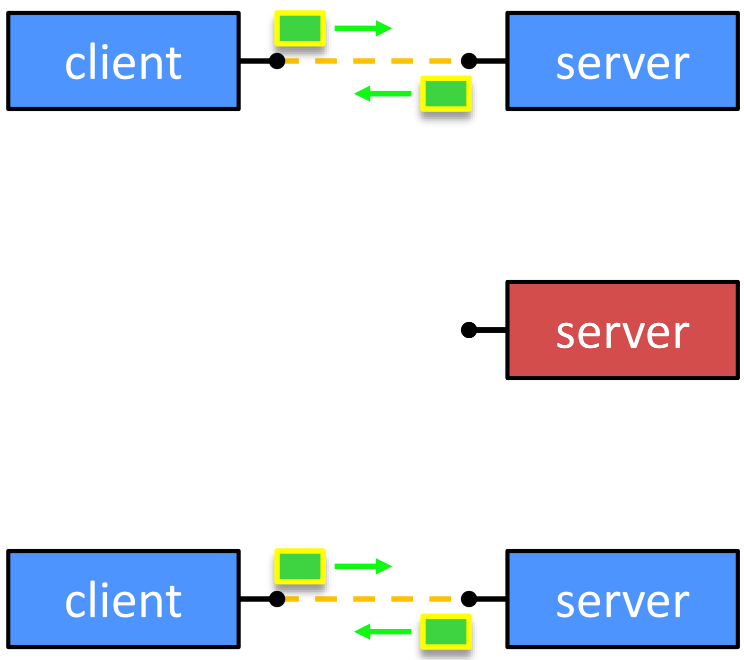
Double-fork Trick



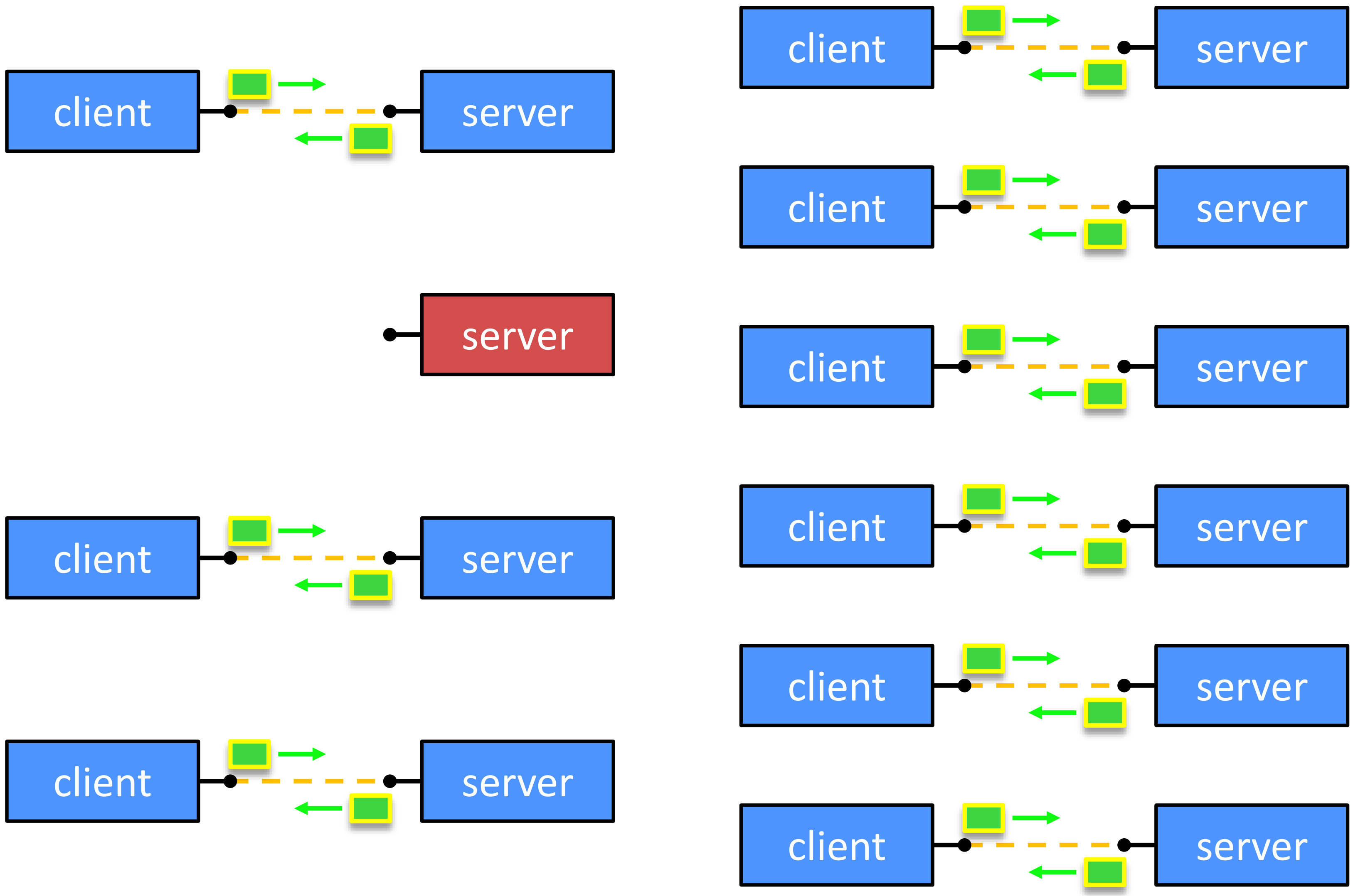
Double-fork Trick



Double-fork Trick



Double-fork Trick



Poll Everywhere

pollev.com/cse33320su

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

A. Zombie until grandparent exits
B. Zombie until grandparent reaps
C. Zombie until init reaps
D. ZOMBIE FOREVER!!!
E. We're lost…

Poll Everywhere

pollev.com/cse33320su

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

A. Zombie until grandparent exits
B. Zombie until grandparent reaps with wait
C. Zombie until init reaps
D. ZOMBIE FOREVER!!!
E. We're lost…

Double fork is done to have process cleaned up without waiting/blocking program

Concurrent with Processes Pseudocode

- See `searchserver_processes/`

```
... // Server set up
while (1) {
  sock_fd = accept();
  pid = fork();
  if (pid == 0) {
    // ??? process









  } else {
    // ??? process




  }
}
```

Concurrent with Processes Pseudocode

- See `searchserver_processes/`

```
... // Server set up
while (1) {
  sock_fd = accept();
  pid = fork();
  if (pid == 0) {
    // Child process







  } else {
    // Parent process



  }
}
```

Concurrent with Processes Pseudocode

- See `searchserver_processes/`

```
... // Server set up
while (1) {
  sock_fd = accept();
  pid = fork();
  if (pid == 0) {
    // Child process
    pid = fork();
    if (pid == 0) {
      // ??? process

    }


  } else {
    // Parent process




  }
}
```

Concurrent with Processes Pseudocode

- See `searchserver_processes/`

```
... // Server set up
while (1) {
  sock_fd = accept();
  pid = fork();
  if (pid == 0) {
    // Child process
    pid = fork();
    if (pid == 0) {
      // Grand-child process
      HandleClient(sock_fd, ...);
    }


  } else {
    // Parent process




  }
}
```

Concurrent with Processes Pseudocode

- See `searchserver_processes/`

```
... // Server set up
while (1) {
  sock_fd = accept();
  pid = fork();
  if (pid == 0) {
    // Child process
    pid = fork();
    if (pid == 0) {
      // Grand-child process
      HandleClient(sock_fd, ...);
    }
    // Clean up resources...
    exit();
  } else {
    // Parent process




  }
}
```

Concurrent with Processes Pseudocode

- See searchserver_processes/

```
... // Server set up
while (1) {
  sock_fd = accept();
  pid = fork();
  if (pid == 0) {
    // Child process
    pid = fork();
    if (pid == 0) {
      // Grand-child process
      HandleClient(sock_fd, ...);
    }
    // Clean up resources...
    exit();
  } else {
    // Parent process
    // Wait for child to immediately die
    wait();
    close(sock_fd);
  }
}
```

Grandchild has copy of socket, we can close our copy

Why Concurrent Processes?

- Advantages:
  - No shared memory between processes
  - No need for language support; OS provides "fork"
  - 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`

- ~ **0.5 milliseconds** per fork*
  - ∴ maximum of (1000/0.5) = 2,000 connections/sec/core
  - ~175 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`

- ~**0.05 milliseconds** per thread creation*
  - ~10x faster than `fork()`
  - ∴ maximum of (1000/0.05) = 20,000 connections/sec/core
  - ~2 billion connections/day/core

- Mush 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
  - We wrote a Thread Pool implementation for you in HW4
- Idea: Thread Pools:
  - Create a fixed set of worker threads when the server starts
  - When a request arrives, add it to a queue of tasks (using locks)
  - Each thread tries to remove a task from the queue (using locks)
  - When a thread is finished with one task, it tries to get a new task from the queue (using locks)