

pollev.com/cse333sp

What has been your favorite topic group so far?

- A. **Memory Management: pointers, references, malloc/free, new/delete, memory bugs, smart pointers**
- B. **Data Structures: arrays, structs, containers**
- C. **Object-Oriented Programming: classes, inheritance**
- D. **Modularization: compilation, interfaces, templates**
- E. I/O: files, buffering, network programming
- F. Concurrency
- G. I prefer not to say

Concurrency: Processes

CSE 333 Spring 2023

Instructor: Chris Thachuk

Teaching Assistants:

Byron Jin

Deeksha Vawani

Humza Lala

Noa Ferman

Seulchan (Paul) Han

Tim Mandzyuk

CJ Reith

Edward Zhang

Lahari Nidadavolu

Saket Gollapudi

Timmy Yang

Wui Wu

Relevant Course Information

- ❖ Homework 4 due Thursday (6/1) @ 11:59 pm
 - Submissions accepted until Sunday (6/4) @ 11:59 pm
- ❖ Course evaluations due Sunday night
 - Please fill them out. They help all staff members improve their skills as educators and allow us to improve the course for future offerings. 😊
- ❖ Final starts Monday (6/5), closes Wednesday (6/7) @ 1pm
 - Ed post this evening with details
- ❖ Friday's lecture *will be fun!*
 - Writing fast(er) code, dog pictures, attempts at humor
 - Competition announcement with a prize sponsored by ACE

Outline

- ❖ We'll look at different `searchserver` implementations
 - 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)

Why Concurrent Processes?

❖ Advantages:

- Processes are isolated from one another
 - No shared memory between processes
 - If one crashes, the other processes keep going
- No need for language support (OS provides `fork`)

❖ Disadvantages:

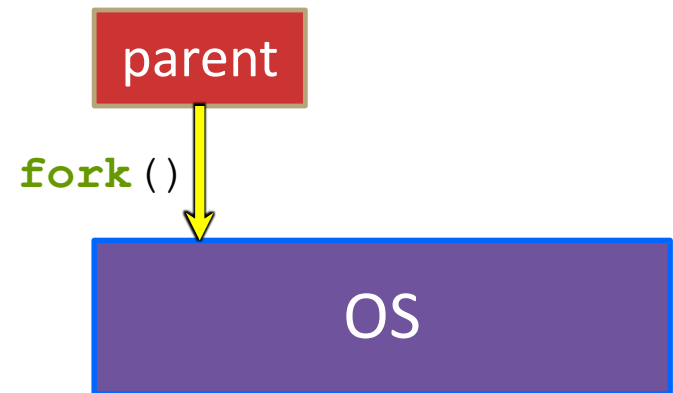
- Processes are heavyweight
 - Relatively slow to fork
 - Context switching latency is high
- Communication between processes is complicated

Process Isolation

- ❖ **Process Isolation** is a set of mechanisms implemented to protect processes from each other and protect the kernel from user processes.
 - Processes have separate address spaces
 - Processes have privilege levels to restrict access to resources
 - If one process crashes, others will keep running
- ❖ Inter-Process Communication (IPC) is limited, but possible
 - Pipes via `pipe()`
 - Sockets via `socketpair()`
 - Shared Memory via `shm_open()`

Creating New Processes (Review)

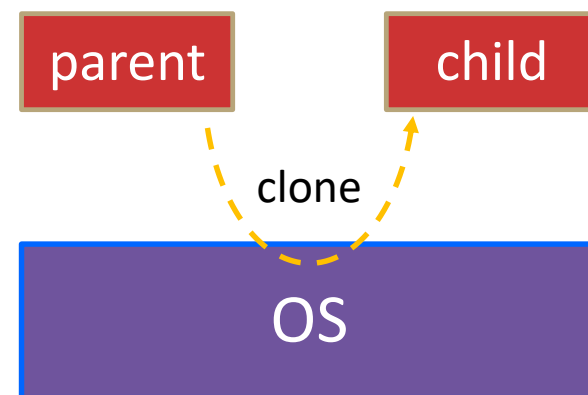
- ❖ `pid_t fork ();`
 - Creates a child process that is an *exact clone* (except threads) of the current/parent process
 - Child process has a separate virtual address space from the parent
- ❖ `fork ()` has peculiar semantics
 - The parent invokes `fork ()`



Creating New Processes (Review)

❖ `pid_t fork();`

- Creates a child process that is an *exact clone* (except threads) of the current/parent process
 - Child process has a separate virtual address space from the parent
- ❖ `fork()` has peculiar semantics
- The parent invokes `fork()`
 - The OS clones the parent



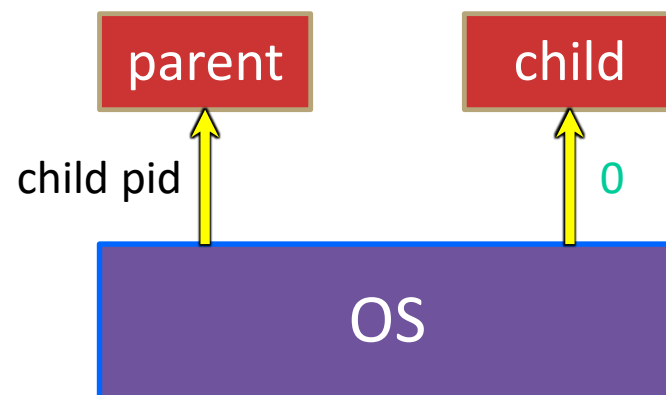
Creating New Processes (Review)

❖ `pid_t fork();`

- Creates a child process that is an *exact clone* (except threads) of the current/parent process
- Child process has a separate virtual address space from the parent

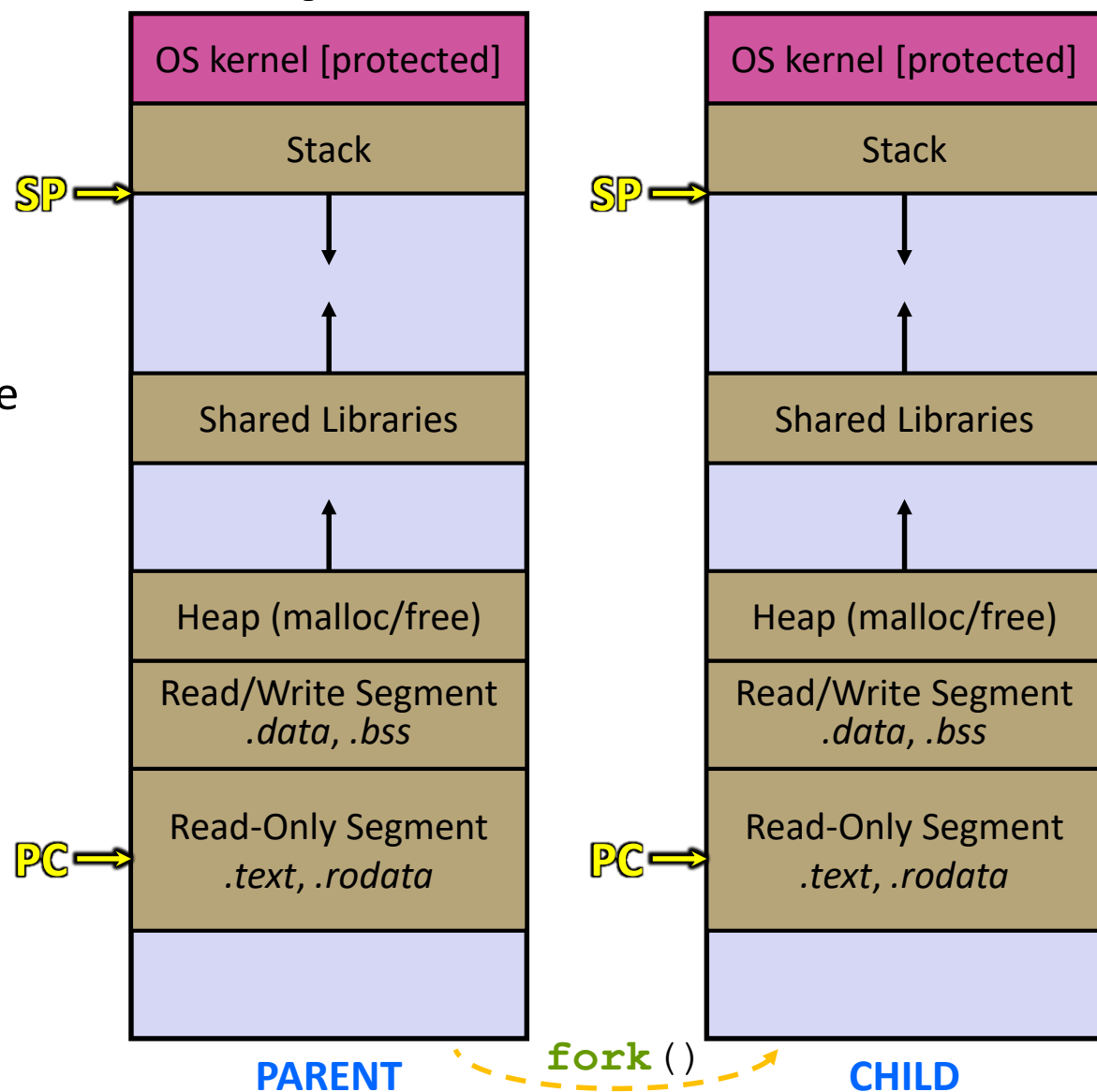
❖ `fork()` has peculiar semantics

- 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 () 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.



Zombies (Review)

- ❖ When a process terminates, its resources (*e.g.*, its address space) hang around as the process sits in a *zombie* state
 - Process terminates by `return` from `main` or calling `exit()`
- ❖ A zombie process needs to be *reaped*
 - Done automatically when its parent process terminates
 - Can be done explicitly by its parent process by calling `wait()` or `waitpid()`, which also returns the *status code*
 - If the parent process terminates before the child becomes a zombie, then `init/systemd` is responsible for reaping it
- ❖ See `fork_example.cc`
 - `ps -u` displays the user's currently running processes

Main Uses of `fork`

- ❖ Fork a child to handle some work
 - *e.g.*, server forks to handle a new connection
 - *e.g.*, web browser forks to render a new website (for security purposes)
- ❖ Fork a child that then starts a new program via `execv`
 - *e.g.*, a shell forks and starts the program you want to run
 - *e.g.*, the 333 grading scripts `fork` and `exec` your executable
- ❖ Fork a background (“daemon”) process that runs independently



How Fast is `fork()` ?

- ❖ See fork_latency.cc
- ❖ **~0.26 milliseconds per fork***
 - \therefore maximum of $(1000/0.5) = 3,800$ connections/sec/core
= ~ 332 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 2-3k 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, ...
- ❖ Tested on `attu4` (3/5/2022)

How Fast is `pthread_create()` ?

- ❖ See `thread_latency.cc`
- ❖ **~0.02 milliseconds** per thread creation*
 - ~13x faster than `fork()`
 - \therefore maximum of $(1000/0.02) = 50,000$ connections/sec/core
= ~4.3 billion connections/day/core
 - Much faster, but writing safe multithreaded code can be serious voodoo, as we've seen
- ❖ *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()`
- ❖ Tested on `attu4` (3/5/2022)

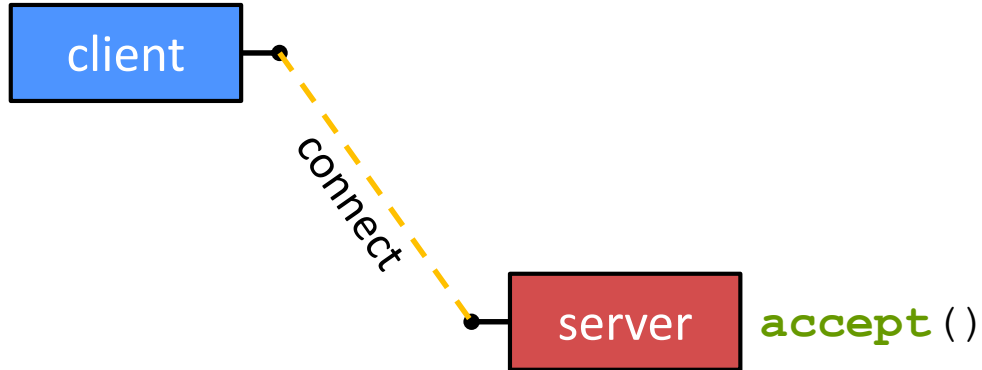
Concurrent Server with Processes

- ❖ 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
- ❖ How do we avoid zombie processes from consuming all of our memory?
 - Option A: Parent calls **wait** () to “reap” children
 - Option B: Use a **double-fork trick**

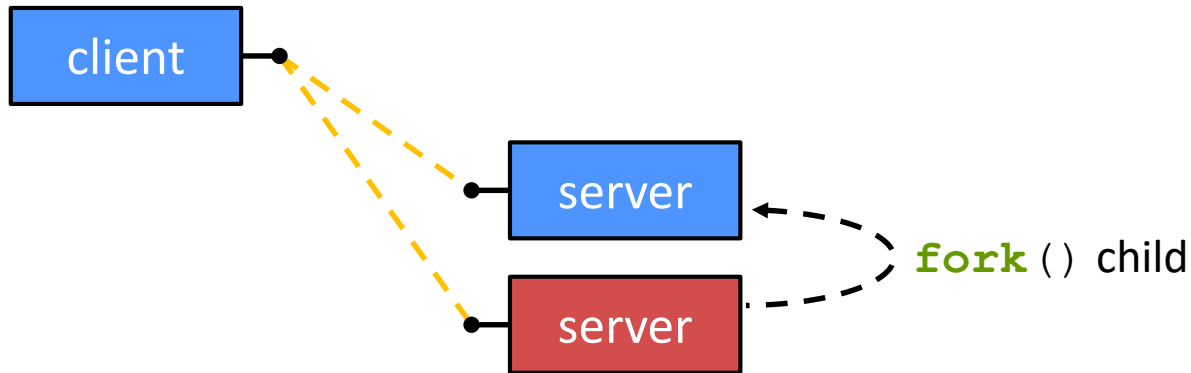
Double-fork Trick



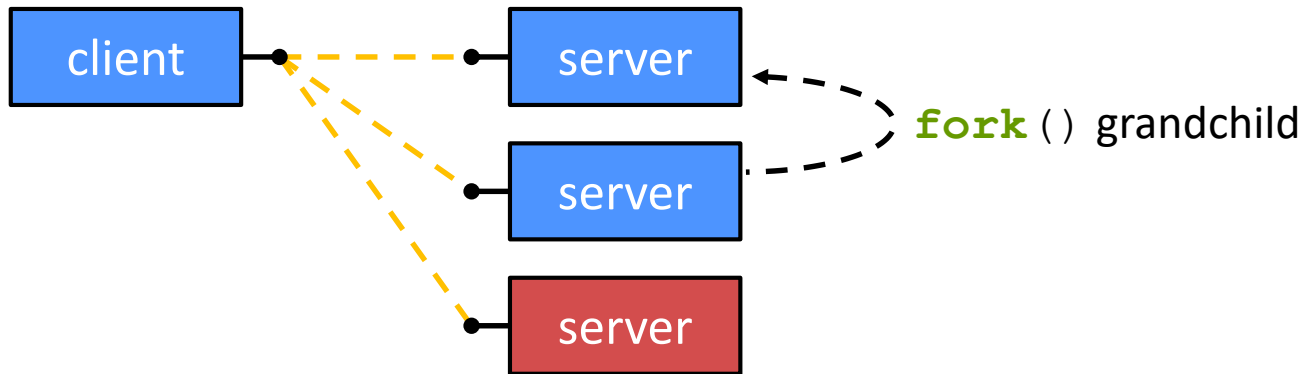
Double-fork Trick



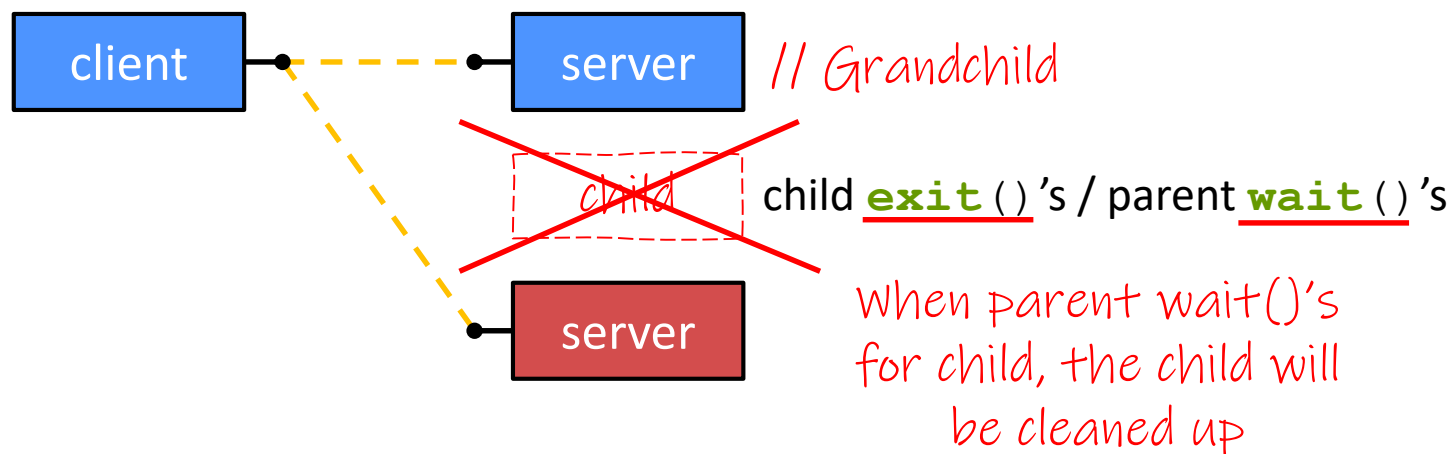
Double-fork Trick



Double-fork Trick



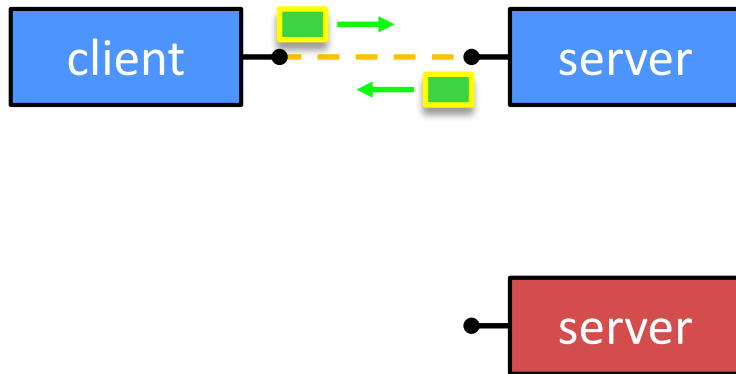
Double-fork Trick



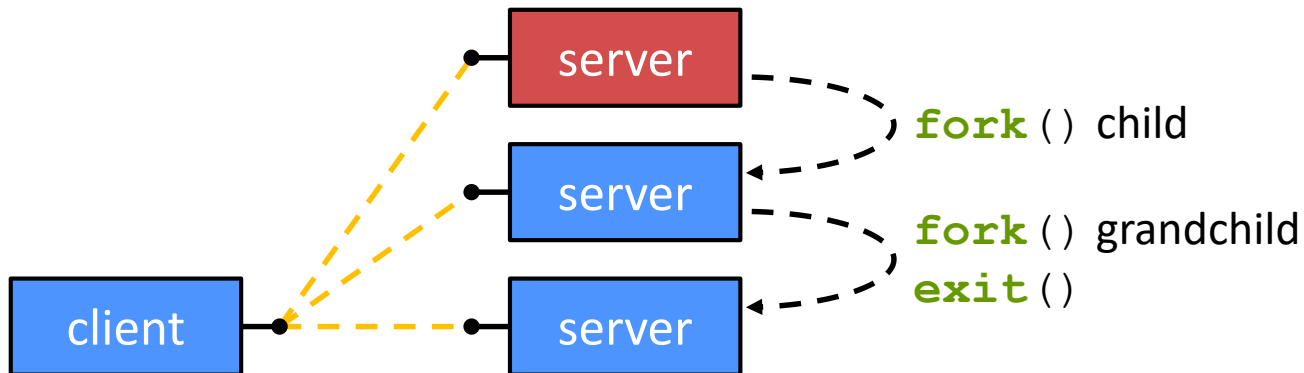
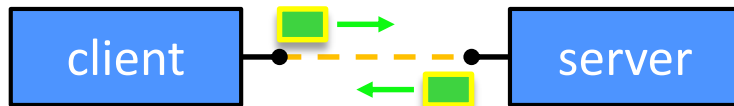
Double-fork Trick



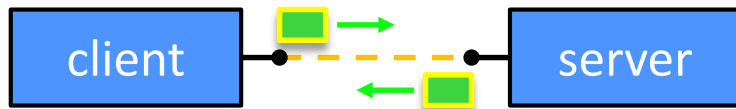
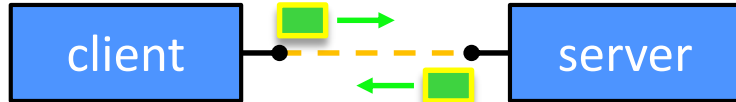
Double-fork Trick



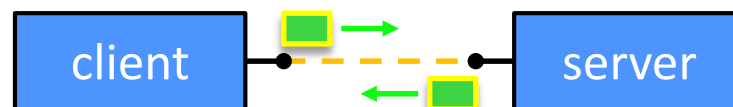
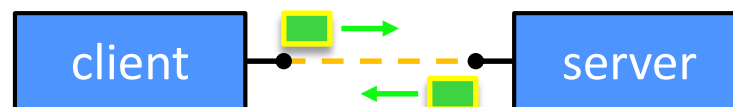
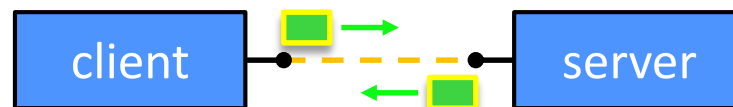
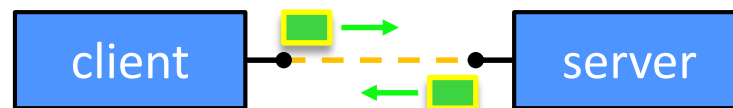
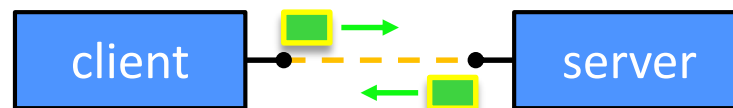
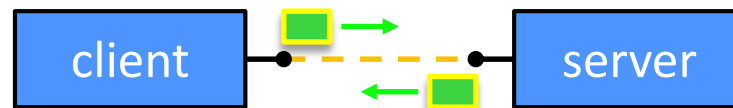
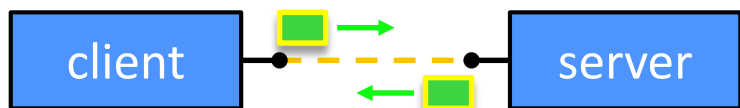
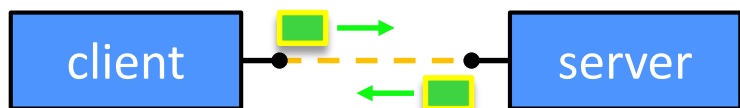
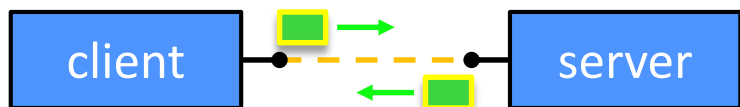
Double-fork Trick



Double-fork Trick



Double-fork Trick



 **Poll Everywhere**pollev.com/cse333sp

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...**

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);
    }
}
```


Outline (Revisited)

- ❖ We'll look at different `searchserver` implementations
 - Sequential
 - Concurrent via forking threads – `pthread_create()`
 - Concurrent via forking processes – `fork()`
 - Concurrent via non-blocking, event-driven I/O – `select()`
- ❖ Conclusions:
 - Concurrent execution leads to better CPU, network utilization
 - Writing concurrent software can be tricky and different concurrency methods have benefits and drawbacks
- ❖ In real servers, we'd like to avoid the overhead needed to create a new thread or process for every request... how?

Aside: Thread Pools

- ❖ Idea:
 - 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)

- ❖ A thread pool is written for you in Homework 4!
 - Feel free to take a look, if curious