pollev.com/cse333j

About how long did Exercise 17 take you?

A. [0, 2) hours
B. [2, 4) hours
C. [4, 6) hours
D. [6, 8) hours
E. 8+ Hours
F. I didn't submit / I prefer not to say

Systems Programming
Concurrency and Processes

Instructors:

Justin Hsia Amber Hu

Teaching Assistants:

Ally Tribble	Blake Diaz	Connor Olson
Grace Zhou	Jackson Kent	Janani Raghavan
Jen Xu	Jessie Sun	Jonathan Nister
Mendel Carroll	Rose Maresh	Violet Monserate

Relevant Course Information

- You are done with 333 exercises! 🎉 🥳
- Homework 4 due Thursday (3/12) @ 11:59 PM
  - Submissions accepted until Sunday (3/15) @ 11:59 PM
- Course evaluations (see Ed #?) 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 Exam: Wednesday (3/18), 12:30–2:20 PM
  - In KNE 110 & 120; see Ed #630
  - Final Review Session this Friday @ 4:30 PM

Concurrency 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

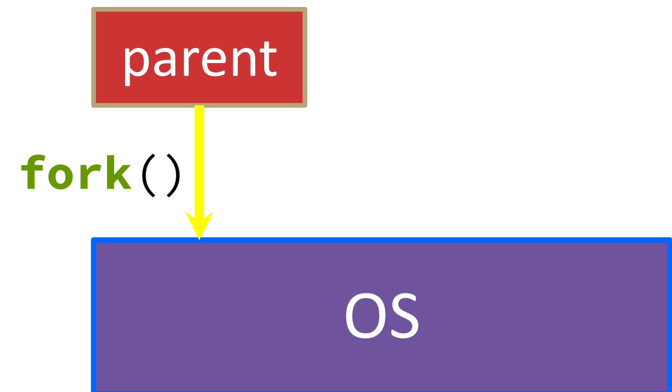
- Process Isolation is a set of mechanisms implemented to protect processes from each other and protect the kernel from user processes.
  - Processes have separate address spaces
  - Processes have privilege levels to restrict access to resources
  - If one process crashes, others will keep running

- Inter-Process Communication (IPC) is limited, but possible
  - Pipes via **`pipe`**`()`
  - Sockets via **`socketpair`**`()`
  - Shared Memory via **`shm_open`**`()`

Review: Creating New Processes (1/3)

- `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

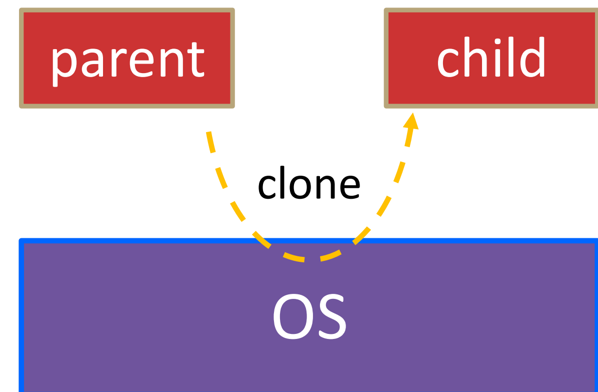
- `fork()` has peculiar semantics
  - The parent invokes `fork()`



Review: Creating New Processes (2/3)

- `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

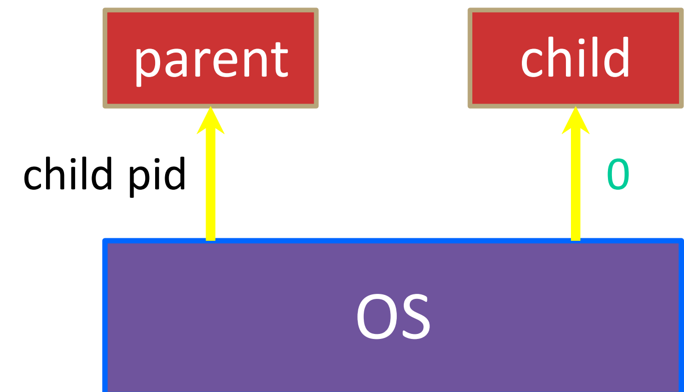
- `fork()` has peculiar semantics
  - The parent invokes `fork()`
  - The OS clones the parent



Review: Creating New Processes (3/3)

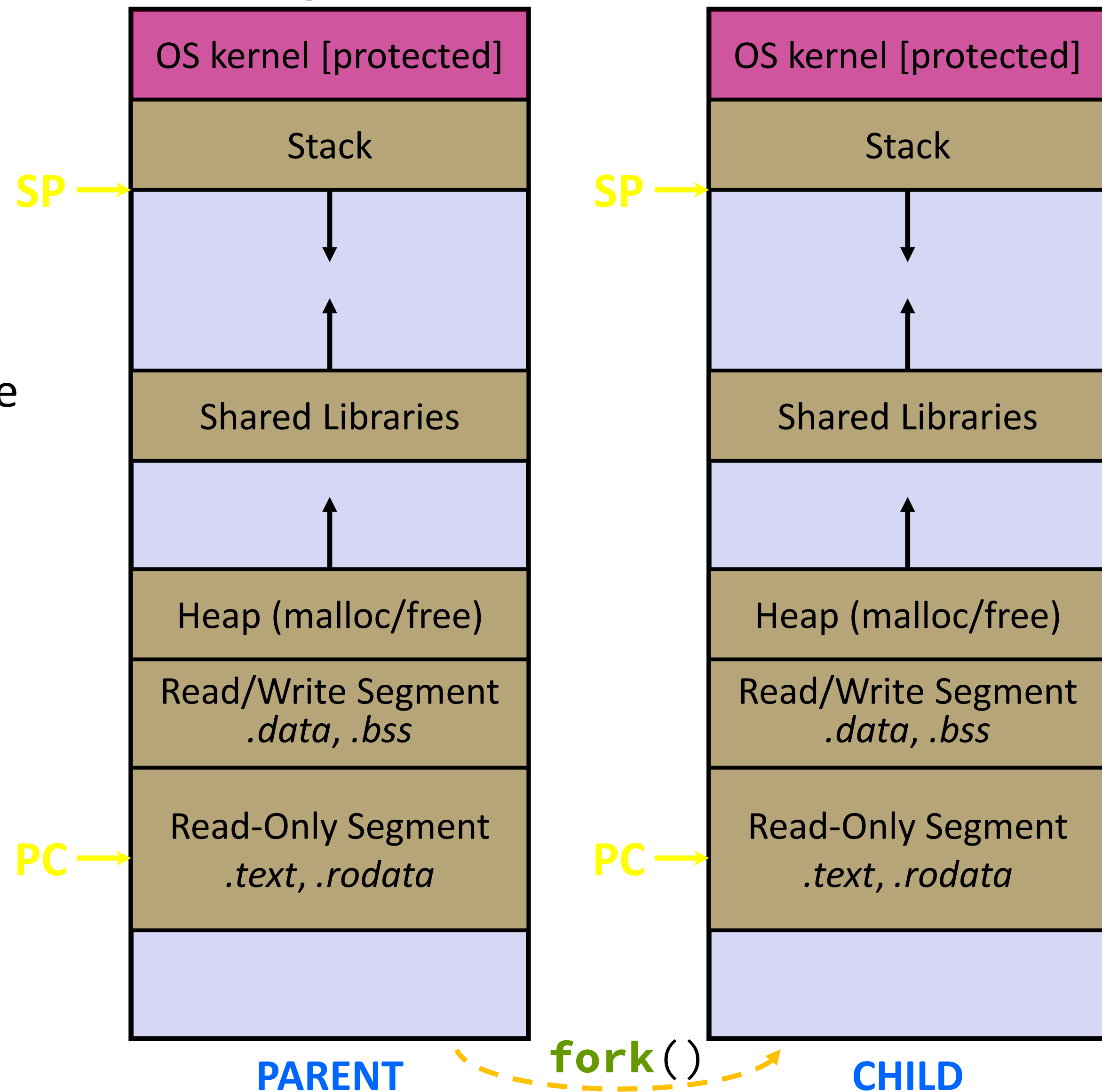
- `pid_t fork();`
  - Creates a child process that is an *exact clone* (except threads) of the current/parent process
  - Child process has a separate virtual address space from the parent

- `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.



Review: Zombies

- 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

How Fast is `pthread_create()`?

- See `thread_latency.cc`

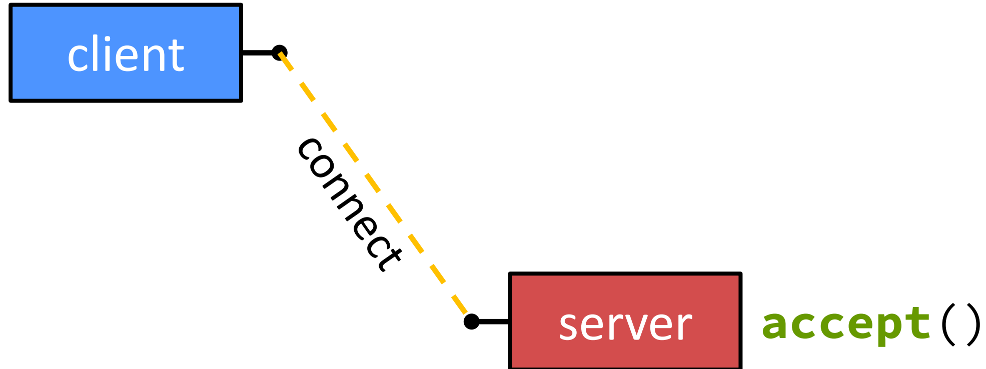
Concurrent Server with Processes

- 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

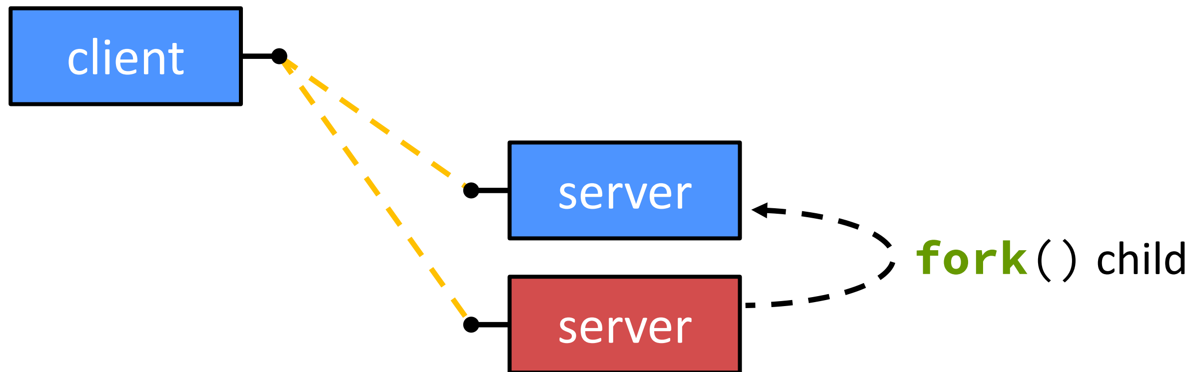
- How do we avoid zombie processes from consuming all of our memory?
  - Option A: Parent calls `wait()` to "reap" children
  - Option B: Use a double-fork trick

Double-fork Trick (1/10)

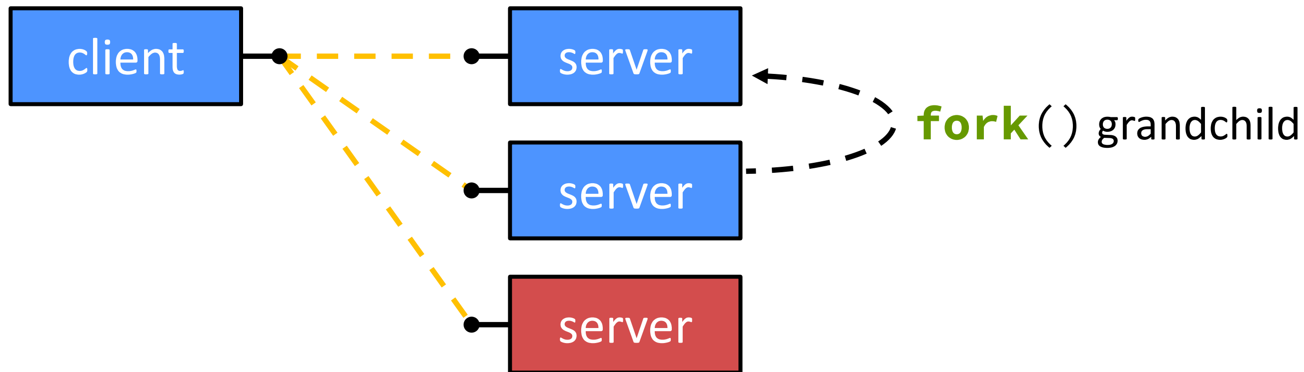
Double-fork Trick (2/10)



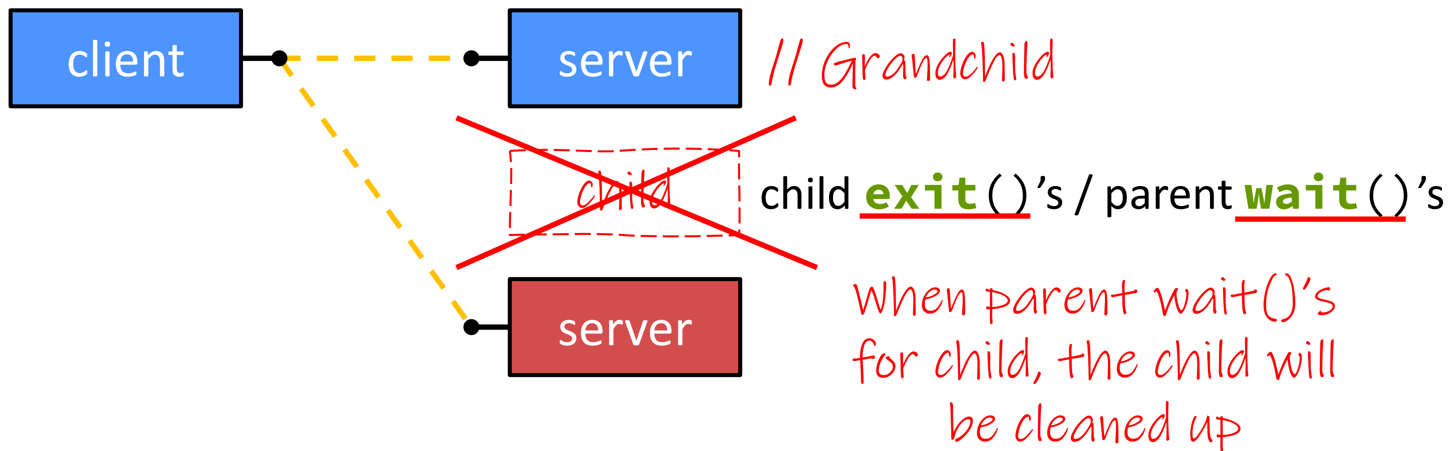
Double-fork Trick (3/10)



Double-fork Trick (4/10)



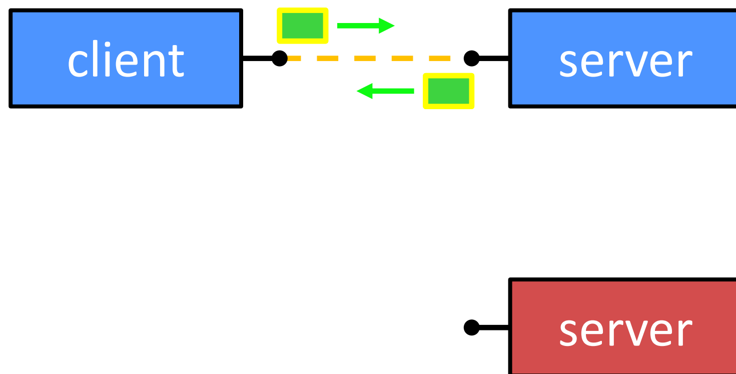
Double-fork Trick (5/10)



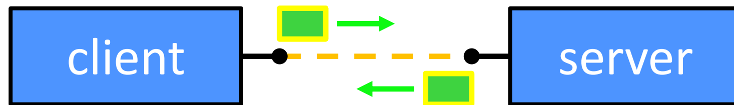
Double-fork Trick (6/10)

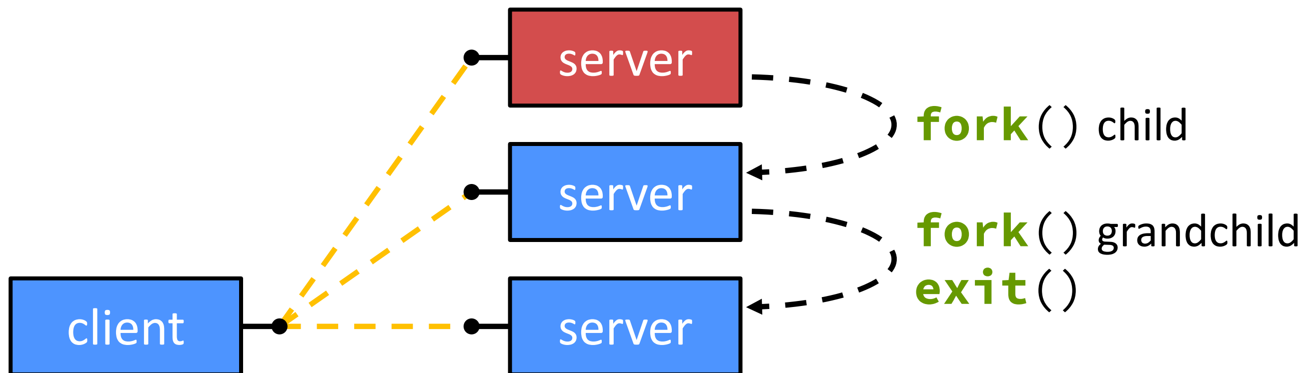
parent closes its client connection

Double-fork Trick (7/10)

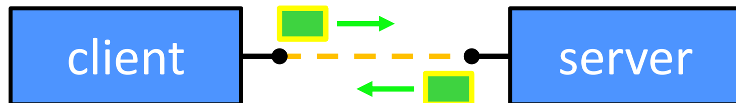


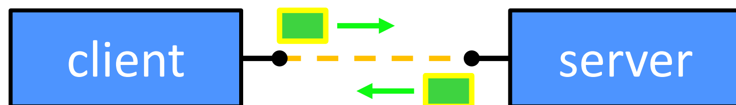
Double-fork Trick (8/10)



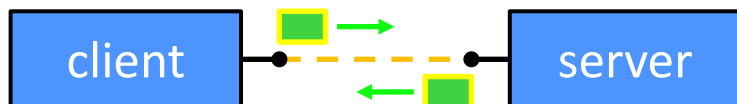


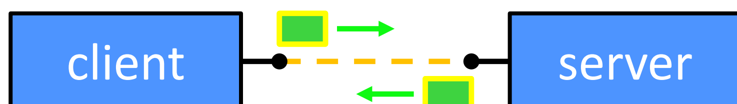
Double-fork Trick (9/10)

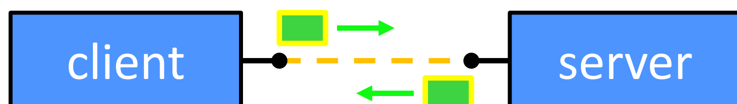


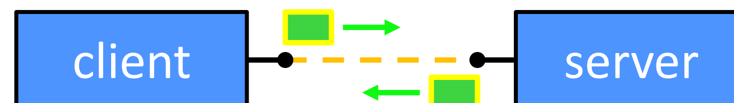


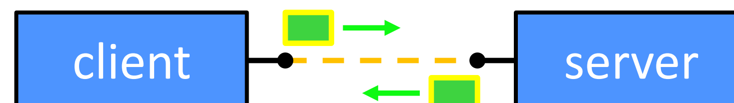
Double-fork Trick (10/10)

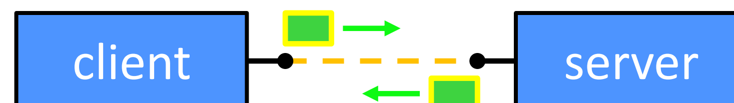


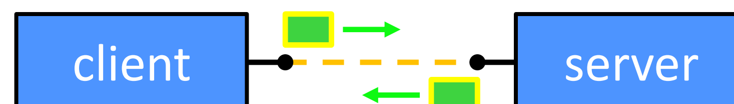


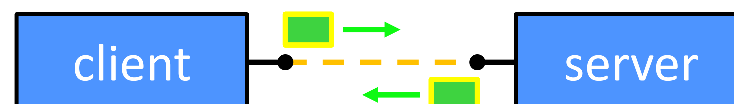


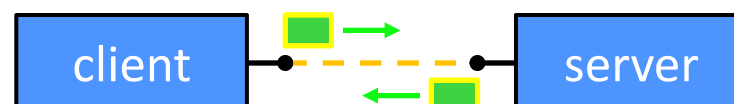












pollev.com/cse333j

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…

Process Concurrency Pseudocode (1/6)

- See `searchserver_processes/`

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









  } else {
    // ??? process




  }
}
```

Process Concurrency Pseudocode (2/6)

- See searchserver_processes/

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









  } else {
    // Parent process




  }
}
```

Process Concurrency Pseudocode (3/6)

- 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




  }
}
```

Process Concurrency Pseudocode (4/6)

- 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




  }
}
```

Process Concurrency Pseudocode (5/6)

- 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




  }
}
```

Process Concurrency Pseudocode (6/6)

- 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 terminate
    wait();
    close(sock_fd);
  }
}
```

Concurrency 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