Poll Everywhere

pollev.com/cse333

About how long did Exercise 11 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

Concurrency: Processes

CSE 333 Summer 2023

Instructor: Timmy Yang

Teaching Assistants:

Jennifer Xu Leanna Nguyen Pedro Amarante
Sara Deutscher Tanmay Shah

Relevant Course Information

- Exercise 12 due Monday (8/14) @ ~~11:00 am~~ 1:00 pm
- Homework 4 due Wednesday (8/16) @ 11:59 pm
  - Submissions accepted until Friday (8/18) @ 11:59 pm
- Quiz 4 (Wednesday, 8/16 – Friday, 8/18)
  - Same policies as previous quizzes
  - ex10-ex12, hw4, overall course questions!

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)

Aside: Thinking about Threads

- Recall: More instructions per thread = higher likelihood of interleaving
  - Even seemingly simple lines can interleave in strange ways.

- Let's look at the following example…

Aside: Thinking about Threads

- What are some possible outputs?
- What's the range of possible outputs?

g=1, g=6 , g=6, g=12 , g=6, g=7

g=12 g=12

max = 12

min = 1 ?

```
int g = 0;
void *worker(void *ignore) {
 for (int k = 1; k <= 3; k++) {
   g = g + k;
 }
 printf("g = %d\n", g);
 return NULL;
}

int main() {
 pthread_t t1, t2;
 int ignore;
 ignore = pthread_create(&t1, NULL, &worker, NULL);
 ignore = pthread_create(&t2, NULL, &worker, NULL);
 pthread_join(t1, NULL);
 pthread_join(t2, NULL);
 return EXIT_SUCCESS;
}
```

Aside: Thinking about Threads

- What are some possible outputs?
  - g = 6/g = 12 | g = 12/g = 12 | g = 7/g = 9 | g = 6/g = 11

```
int g = 0;
void *worker(void *ignore) {
 for (int k = 1; k <= 3; k++) {
   g = g + k;
 }
 printf("g = %d\n", g);
 return NULL;
}

int main() {
 pthread_t t1, t2;
 int ignore;
 ignore = pthread_create(&t1, NULL, &worker, NULL);
 ignore = pthread_create(&t2, NULL, &worker, NULL);
 pthread_join(t1, NULL);
 pthread_join(t2, NULL);
 return EXIT_SUCCESS;
}
```

Aside: Thinking about Threads

- What's the range of possible outputs?
  - g = [4, 12]

```c
int g = 0;
void *worker(void *ignore) {
 for (int k = 1; k <= 3; k++) {
   g = g + k;
 }
 printf("g = %d\n", g);
 return NULL;
}

int main() {
 pthread_t t1, t2;
 int ignore;
 ignore = pthread_create(&t1, NULL, &worker, NULL);
 ignore = pthread_create(&t2, NULL, &worker, NULL);
 pthread_join(t1, NULL);
 pthread_join(t2, NULL);
 return EXIT_SUCCESS;
}
```

Interleaving at the Instruction Level

- Context-switching can happen between any instruction.

```
Instructions for g = g + k:
mov     0x2ebf(%rip),%edx      ⟵ Loads global g into local register
mov     -0x4(%rbp),%eax        ⟵ Loads k into %eax register
add     %edx,%eax              ⟵ Adds copy of g in %edx to %eax register
mov     %eax,0x2eb4(%rip)      ⟵ Stores addition result back into global g
```

- Why does this matter?
  - Remember, each thread has its own Stack *and* register values.
    - Allows for %rax to be used between multiple threads as a return reg.

How to Get 4

	Thread 1	Thread 2	
	`reg1 ⇐ g`	`reg2 ⇐ g`	Load 0 into `reg`
	`g ⇐ reg1 + 1`	`g ⇐ reg2 + 1`	Write `g=1`
	`reg1 ⇐ g`	`reg2 ⇐ g`	
	`g ⇐ reg1 + 2`	`g ⇐ reg2 + 2`	
Load 1 into `reg`	`reg1 ⇐ g`	`reg2 ⇐ g`	
Write `g=4`	`g ⇐ reg1 + 3`	`g ⇐ reg2 + 3`	

`g = 4`

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

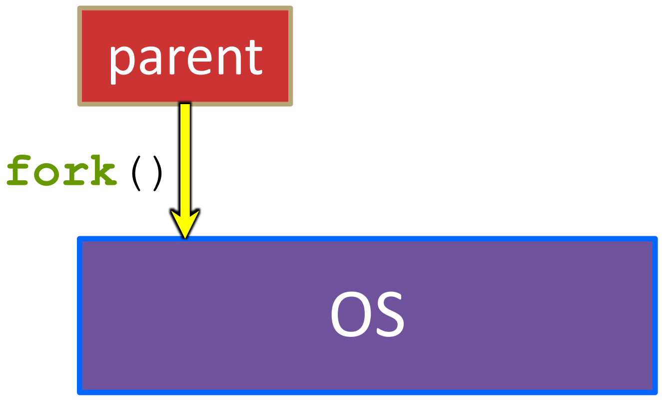
- Process Isolation is a set of mechanisms implemented to protect processes from each other and protect the kernel from user processes.
  - Processes have separate address spaces
  - Processes have privilege levels to restrict access to resources
  - If one process crashes, others will keep running

- 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

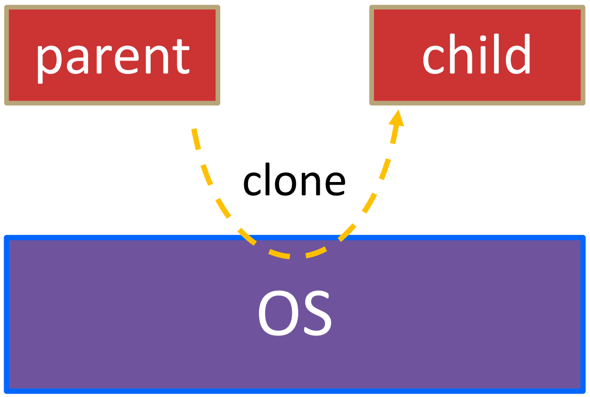
- `fork()` has peculiar semantics
  - The parent invokes `fork()`



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

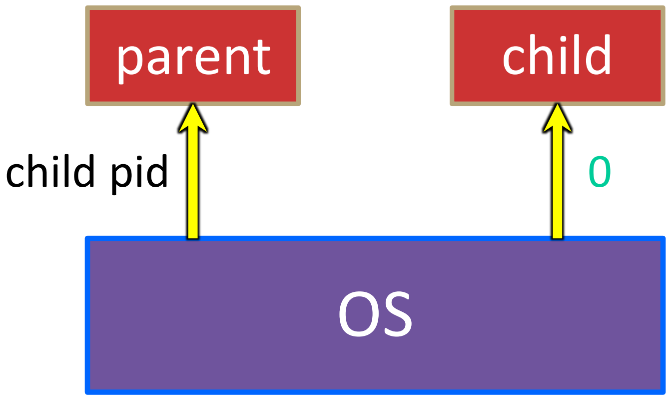
- `fork()` has peculiar semantics
  - The parent invokes `fork()`
  - The OS clones the parent



Creating New Processes (Review)

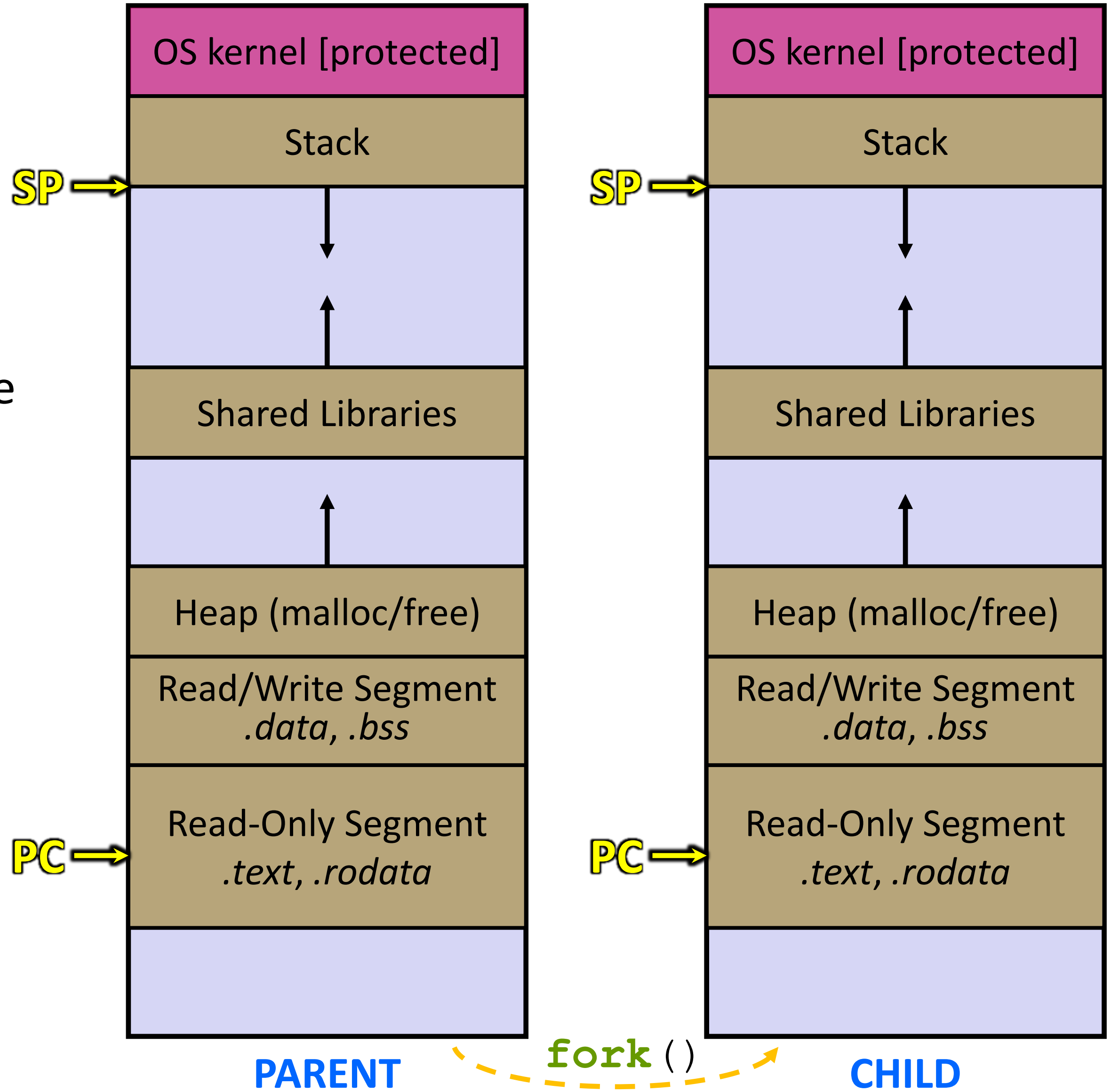
- `pid_t fork();`
  - Creates a child process that is an *exact clone* (except threads) of the current/parent process
  - Child process has a separate virtual address space from the parent

- `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

Concurrent Server with Processes

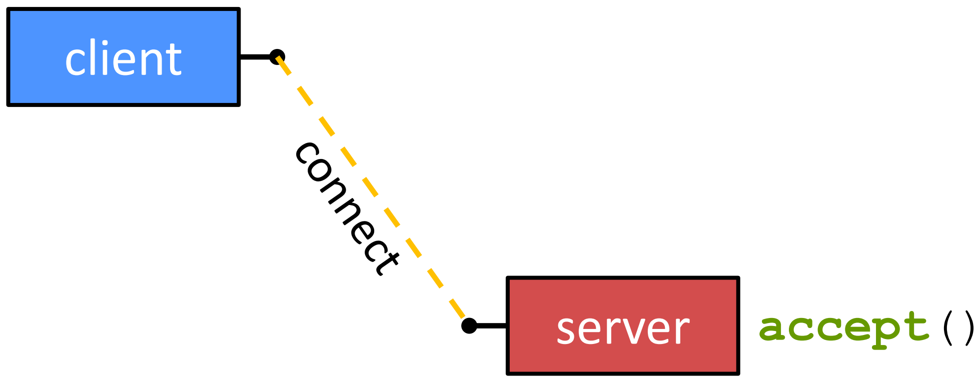
- The **parent** process blocks on `accept()`, waiting for a new client to connect
  - When a new connection arrives, the parent calls `fork()` to create a **child** process
  - The child process handles that new connection and `exit()`'s when the connection terminates

- 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

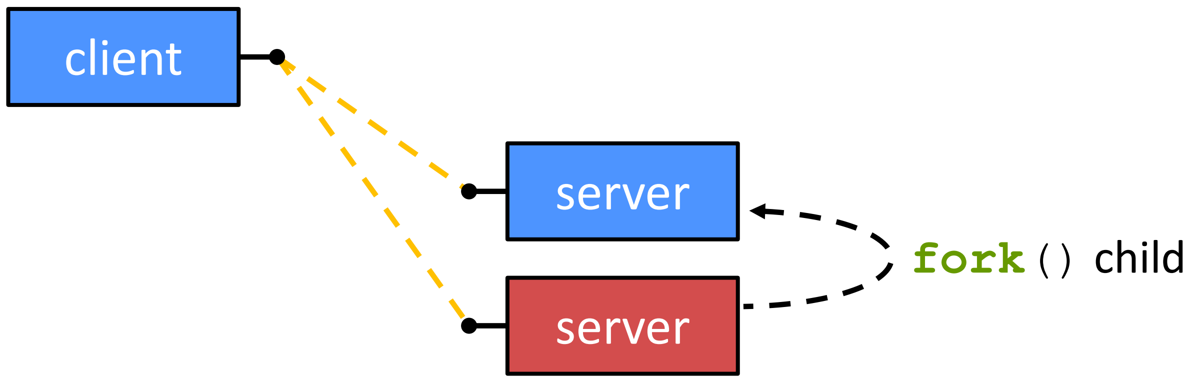
block until child exits()

Double-fork Trick

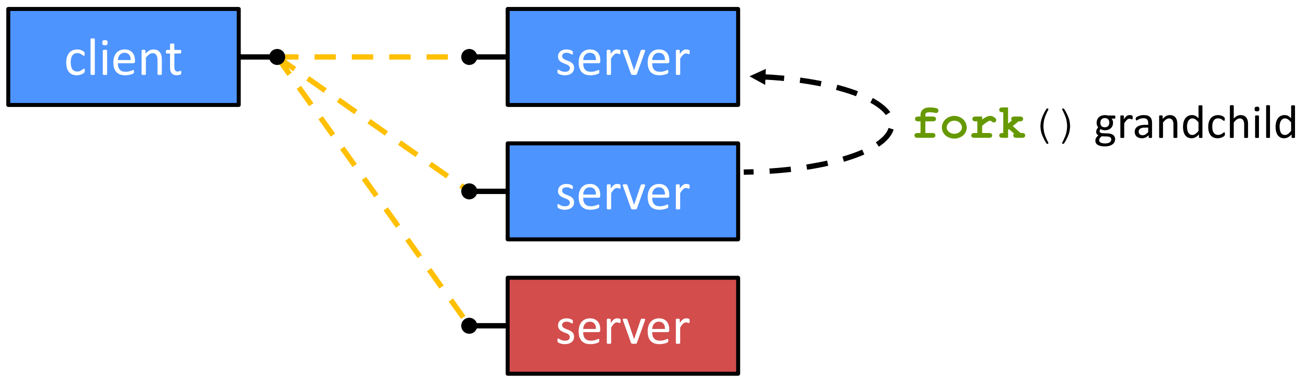
Double-fork Trick



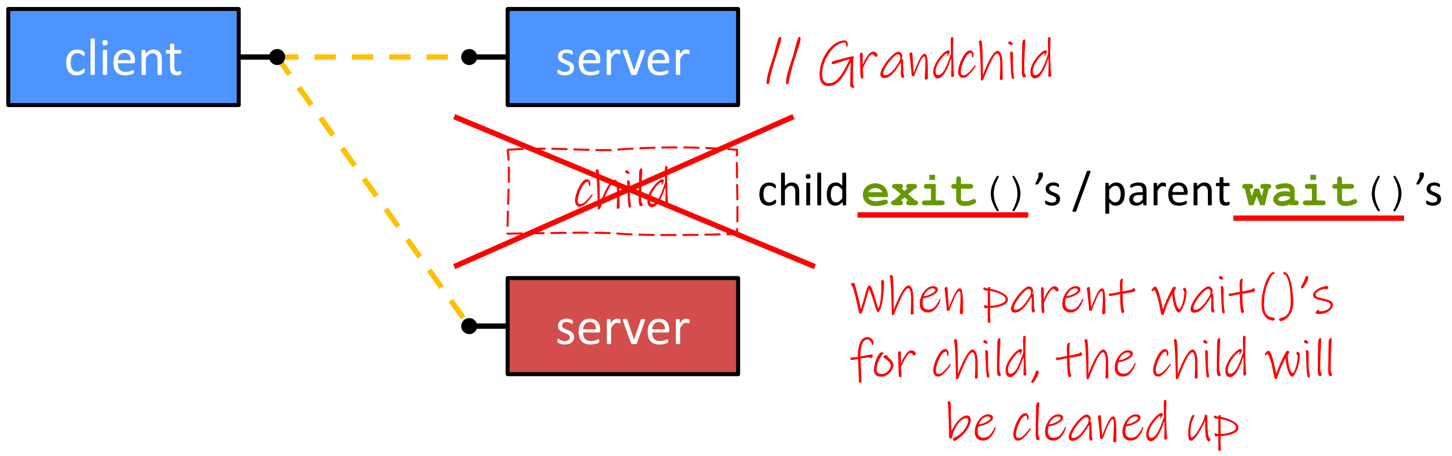
Double-fork Trick



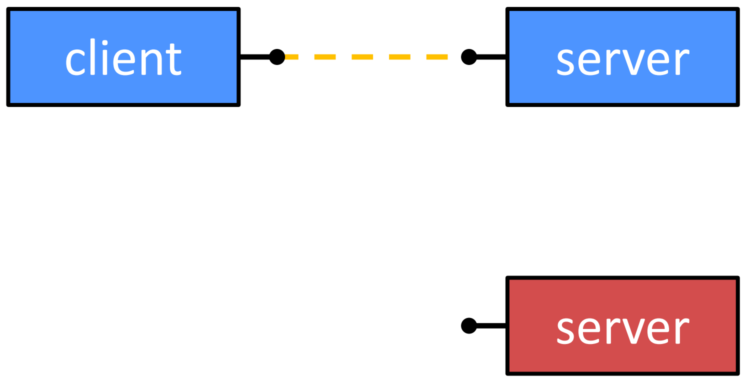
Double-fork Trick



Double-fork Trick

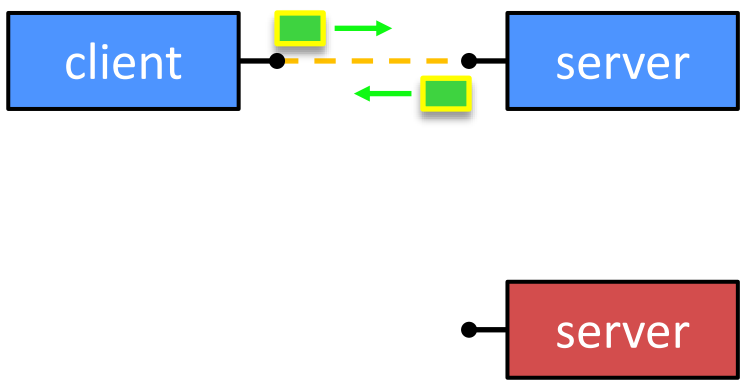


Double-fork Trick

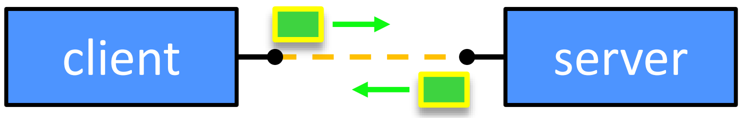


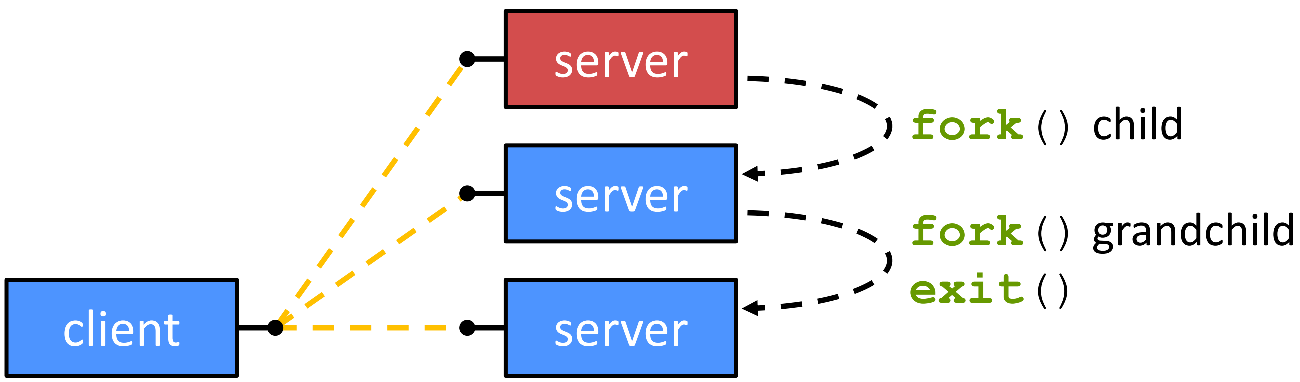
parent closes its client connection

Double-fork Trick

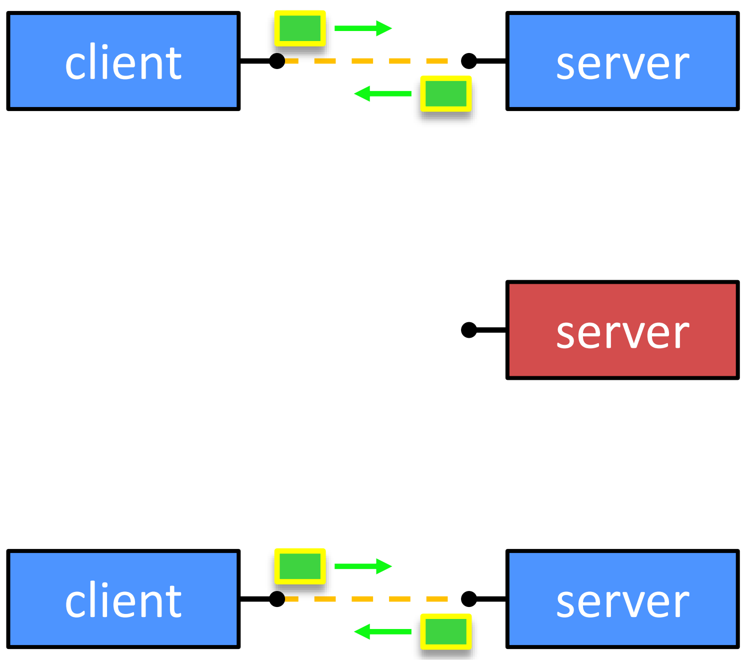


Double-fork Trick

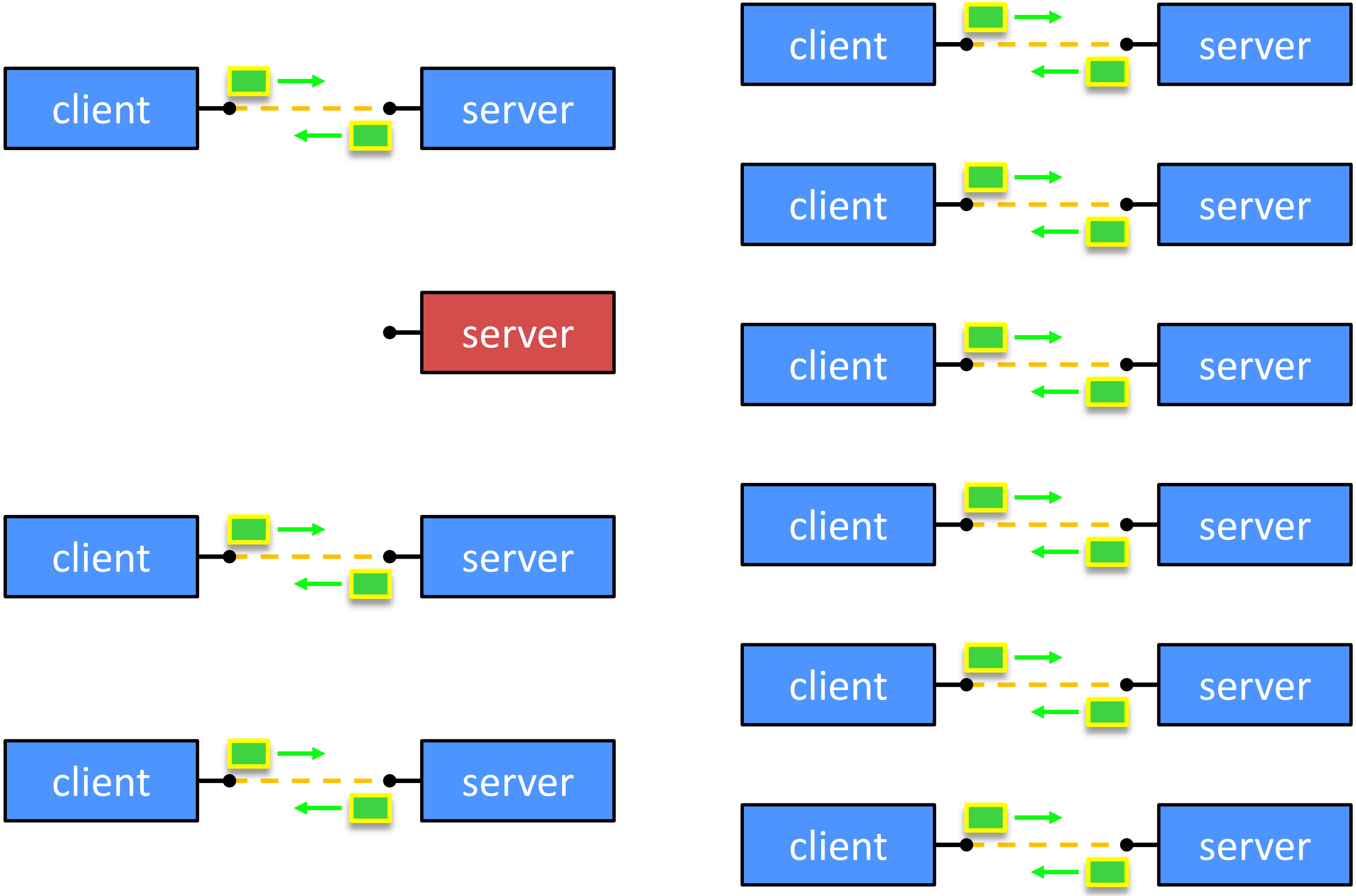




Double-fork Trick



Double-fork Trick



Poll Everywhere

pollev.com/cse333

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…

grandparent accept() loop, never exits

Not the parent process

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);
  }
}
```

How Fast is `fork()`?

- See `fork_latency.cc`

- ~**0.26 milliseconds** per fork*
  - ∴ 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`**`()`
  - ∴ maximum of (1000/0.02) = 50,000 connections/sec/core = ~4.3 billion connections/day/core

  - Mush 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)

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