Concurrency: Processes
CSE 333

Instructor: Alex Sanchez-Stern

Teaching Assistants:
Audrey Seo
Deeksha Vatwani
Derek de Leuw
Katie Gilchrist

Administrivia

- Last exercise due this morning woohoo! 🎉
- hw4 due Wednesday night
  - Usual late days (2 max) apply if you have any remaining
- Final exam Fri. August 22nd, 1:10-2:10, HRC 155
  - Topic list on the web; exam will mostly cover 2nd half of the quarter
  - Old exams also available on the website.
  - Closed book but you may have a 3x5 card with handwritten notes
    - Blank cards available after class

Administrivia

- Course evaluations open as of Friday
  - Your feedback fuels me 🧛
  - https://uw.iasystem.org/survey/311914
- Section this week is an exam review… show up, and bring exam questions!

Administrivia

- Extra final points for coming to office hours this week
  - +5 points on the final (out of 100), but can't go above 100 total
  - Must go to an existing, in-person office hours and bring a problem set to work on; either from the extra-problems in the slides, or an old final question
  - Make sure the staff member writes down your name

Search Server Versions

- We've been looking at different `searchserver` implementations
  - Sequential
  - Concurrent via dispatching threads: `pthread_create()`
  - **Concurrent via forking processes: `fork()`**

Reference: *Computer Systems: A Programmer's Perspective*, Chapter 12 (CSE 351 book)

Concurrency and Processes

- To implement a "process", the operating system gives us:
  - Resources such as file handles and sockets
  - Call stack + registers to support (eg, PC, SP)
  - Virtual memory (page tables, TLBs, etc …)
- Minimal set to implement concurrency: Call stack and registers
- But what if we want more than the minimum

"Worker" 1

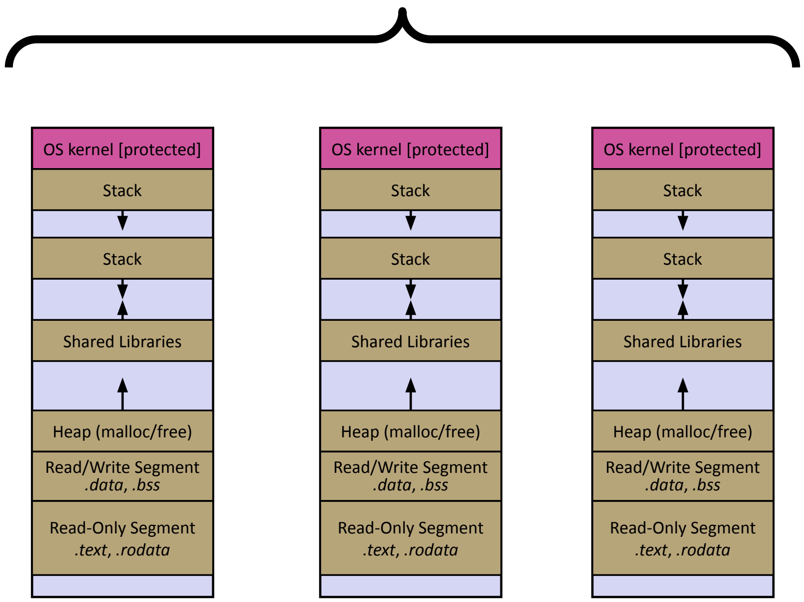
```
bucket = hash(word);
hitlist = file.read(bucket);
```

"Worker" 2

```
foreach hit in hitlist {
  doclist.append(file.read(hit));
}
```

Concurrency and Processes

Multi-Process Program



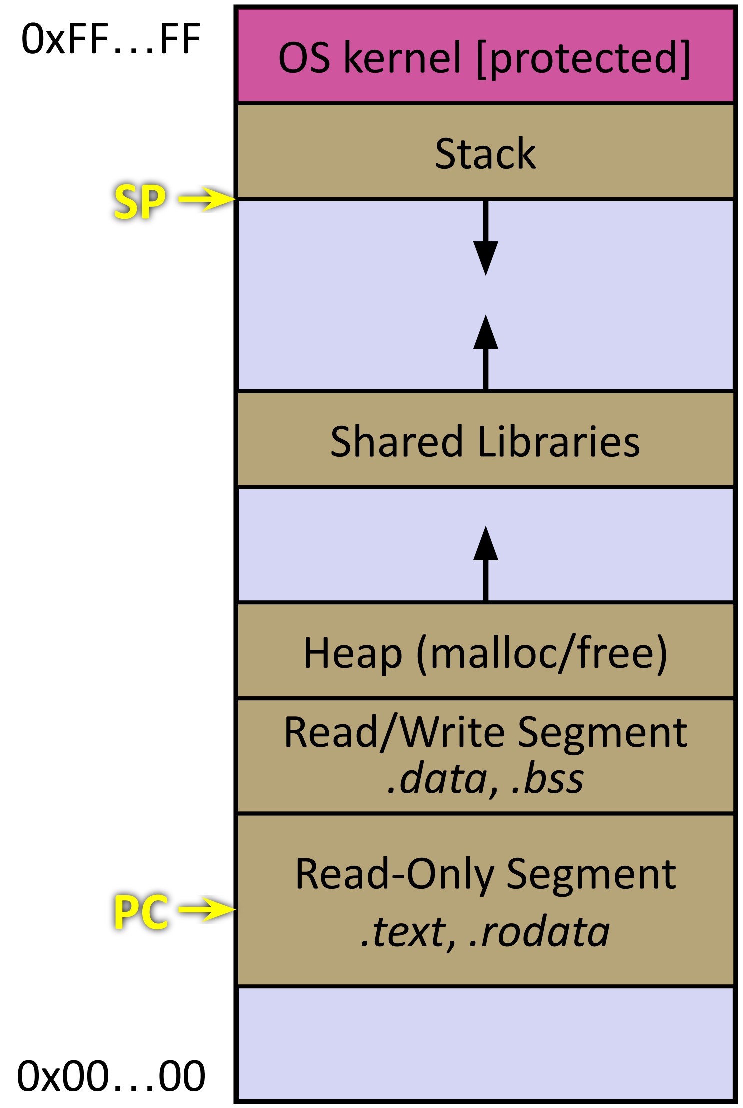
Creating New Processes

```
pid_t fork(void);
```

- Creates a new process (the "child") that is a *clone* of the current process (the "parent")
- Primarily used in two patterns:
  - Adding concurrency to an existing program, for instance a web server
    - `fork` a child, then that child executes a subroutine
  - Starting another program, for instance using a shell
    - `fork` a child, then that child uses `exec` to swap it's executable for another.

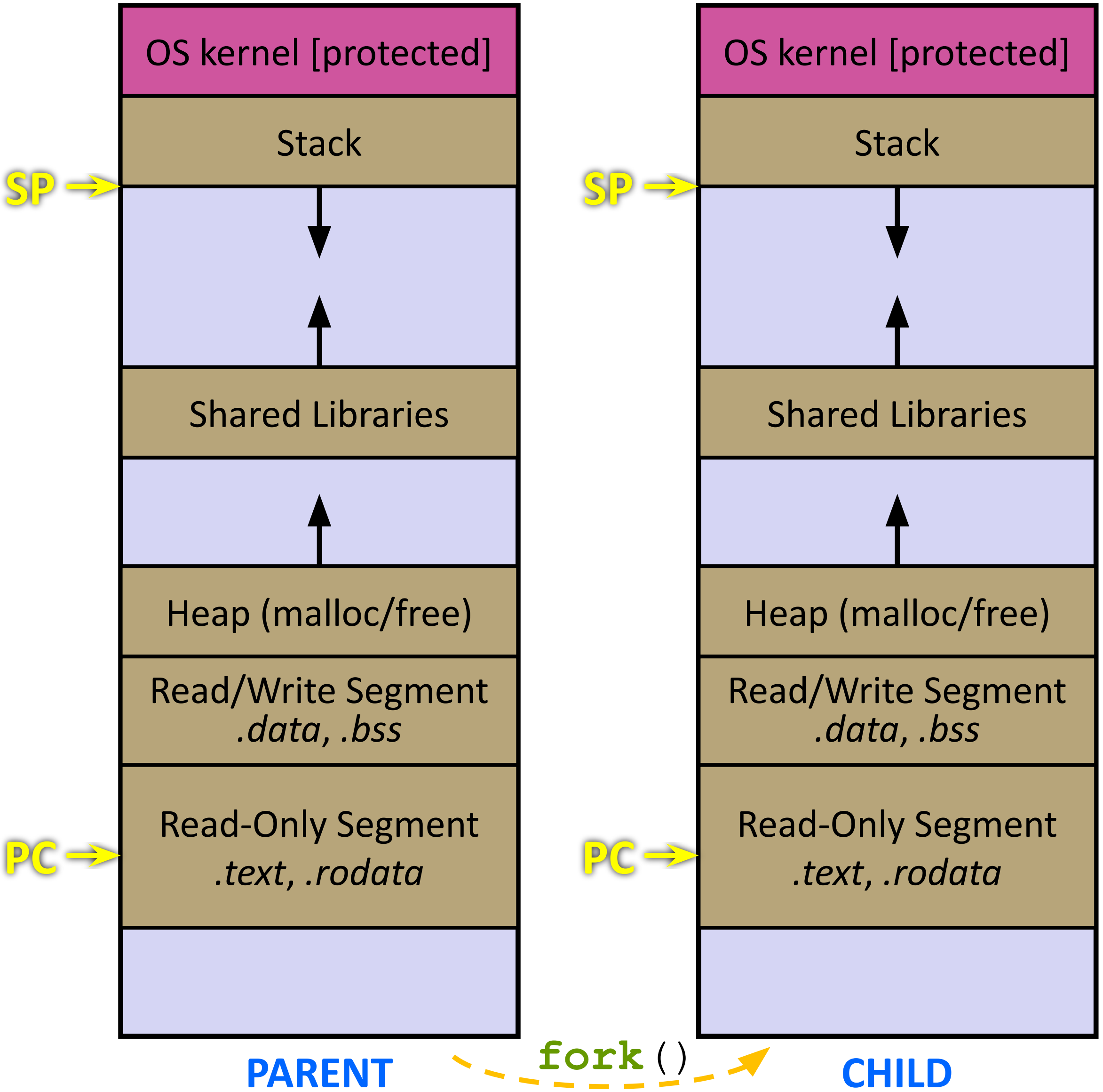
`fork()` and Address Spaces

- A process executes within an *address space*
  - Process tracks its current state using the stack pointer (SP) and program counter (PC)



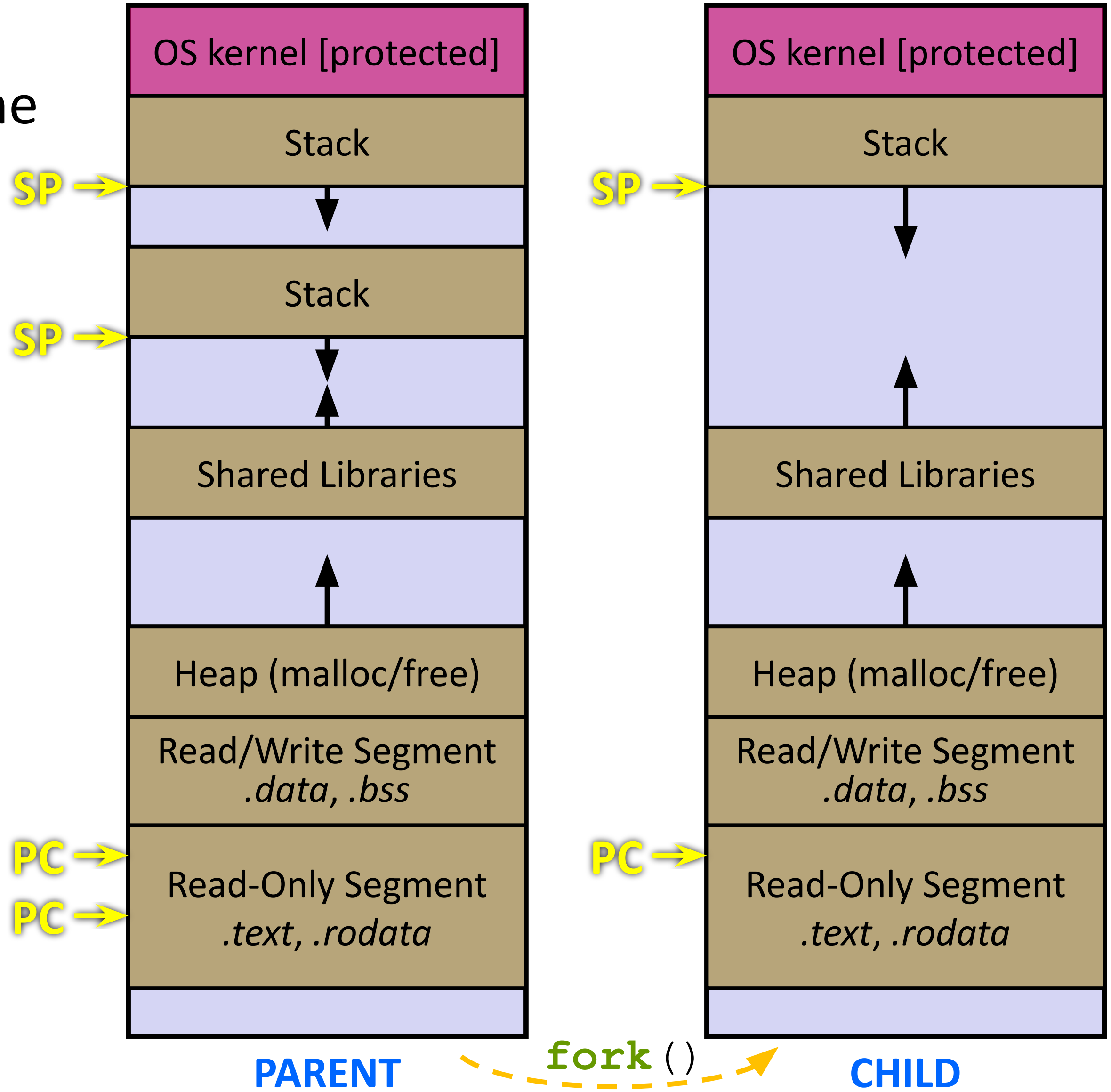
`fork()` and Address Spaces

- `fork()` causes the OS to clone the process state
  - 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.



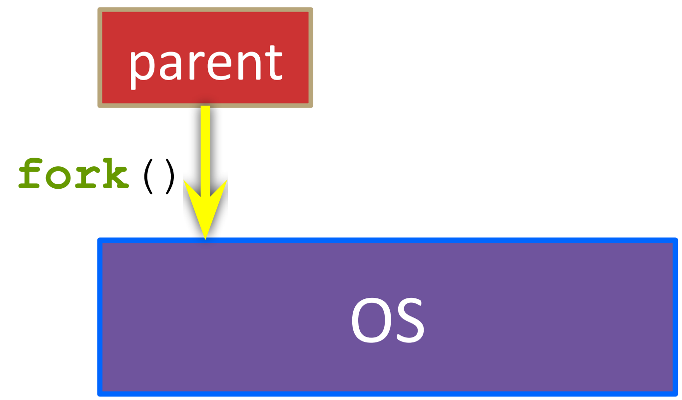
`fork()` and Address Spaces

- Fork does ***not*** clone threads
  - Only the thread that called fork is duplicated
  - If the parent had multiple stacks for threads, the child only has one.
  - This can be a source of bugs; try to only use concurrent processes **or** threads, not both.



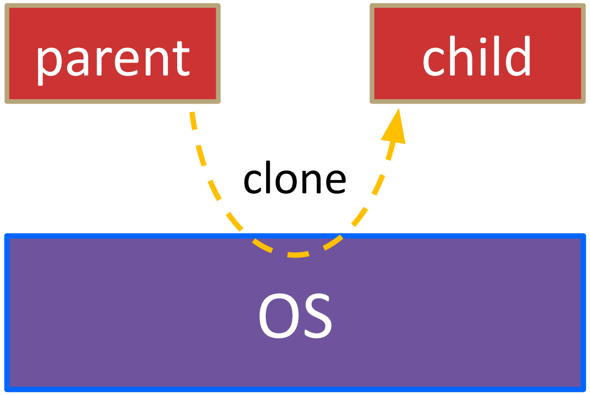
`fork()`

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



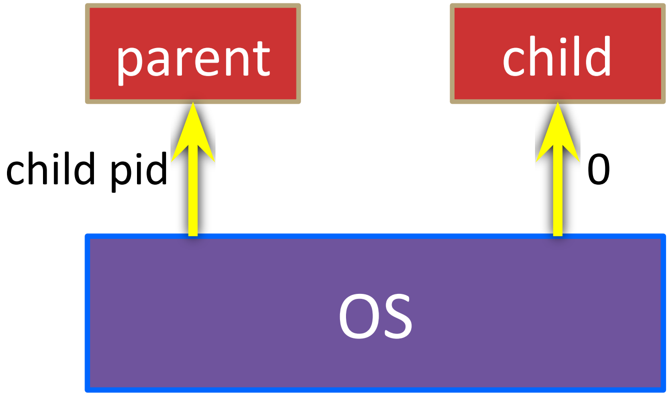
`fork()`

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



`fork()`

- `fork()` has peculiar semantics
  - The parent invokes `fork()`
  - The OS clones the parent
  - *Both* the parent and the child return from fork
    - Only return value differs:
      - Parent receives child's pid
      - Child receives a 0



Ending a fork()

- Since all processes are launched through **`fork()`**, parents must handle the exit codes of their children
- ```
  pid_t waitpid(pid_t pid, int* status, int options);
  ```

- `pid`: the child pid to wait for
  - -1 means wait for any child
  - Can also be one of a few other values (see man pages)
- `status`: an output parameter for the child return code
- Returns the process id of the child that finished
  - -1 on error
  ```

Ending a fork()

- `pid_t waitpid(pid_t pid, int* status, int options);`

  Usually 0; can be used to not block or some other esoteric options (see man page)

- If the parent calls **`waitpid()`** before the child terminates, it will block until the child is done

- If the child terminates first, it will not clean up fully until the parent calls **`waitpid()`**
  - called becoming a "zombie".
- Can also wait for **any** child to exit with:
  - `pid_t wait(int* status);`

fork()

See:

`fork_example.cc`

https://courses.cs.washington.edu/courses/cse333/25su/lecture/23-processes-example

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

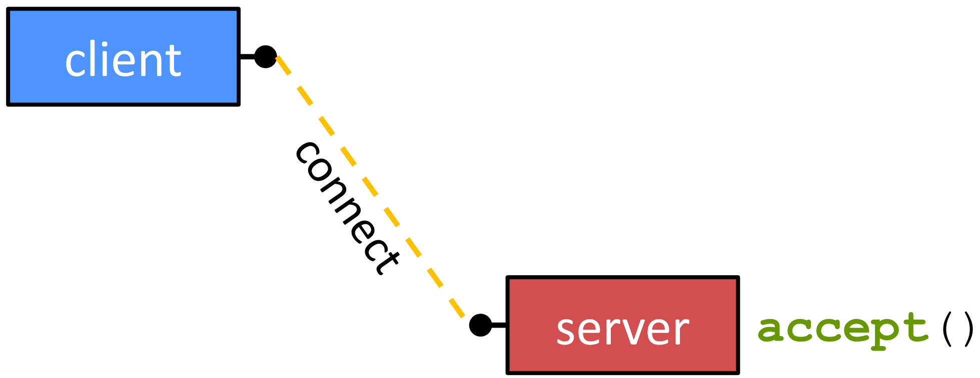
Concurrent Server with Processes

- Remember that children become "zombies" after termination
- The OS is waiting for someone to read their exit code before getting rid of them

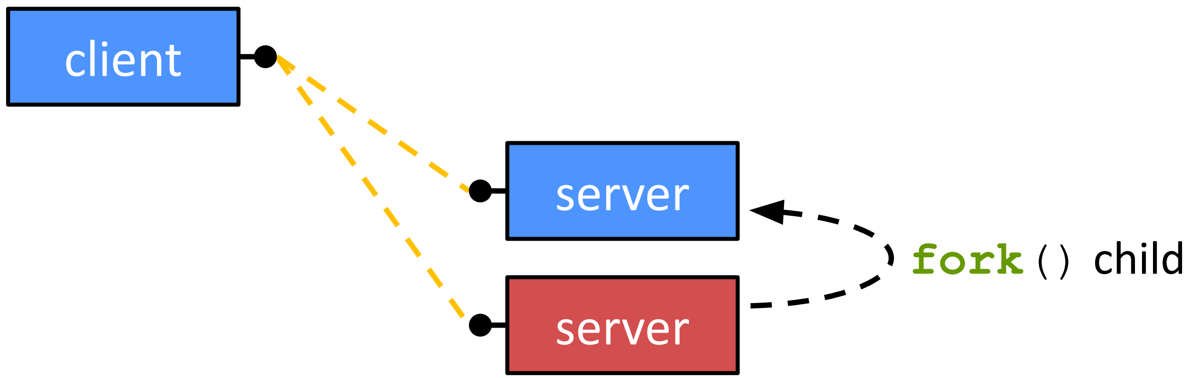
- Two ways to handle this:
  - Option A: Parent calls `wait()` to "reap" children and receive their exit codes.
  - Option B: Use the double-fork trick

Double-fork Trick

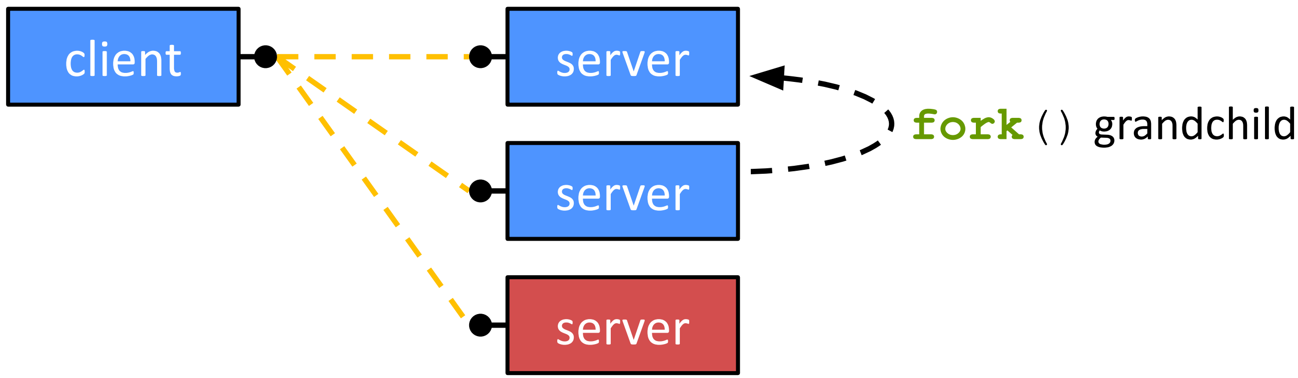
Double-fork Trick



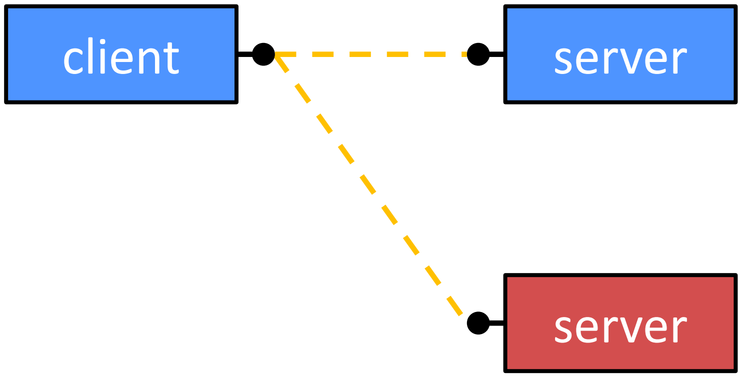
Double-fork Trick



Double-fork Trick

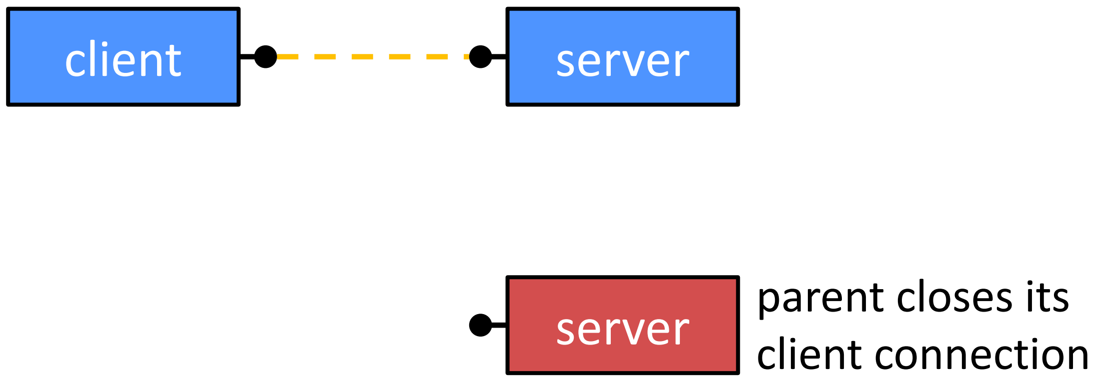


Double-fork Trick

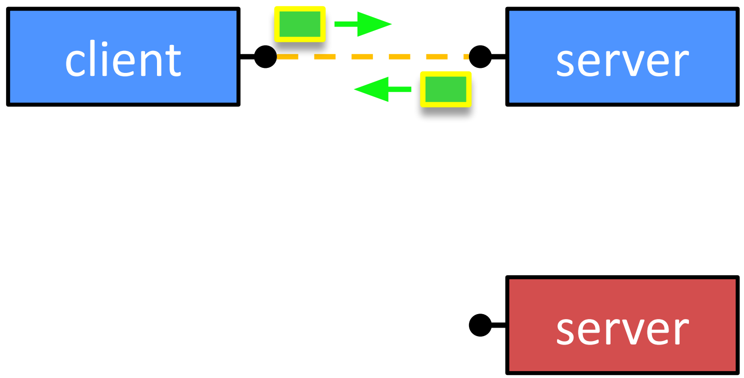


child **`exit`**()'s / parent **`wait`**()'s

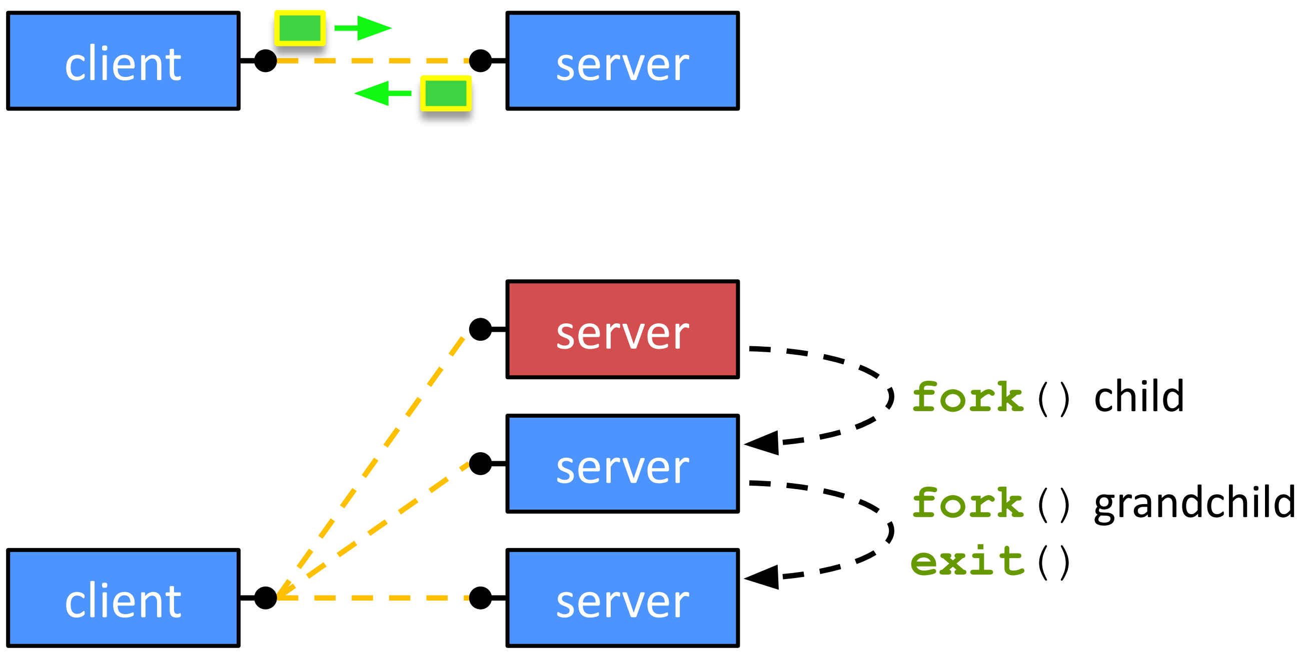
Double-fork Trick



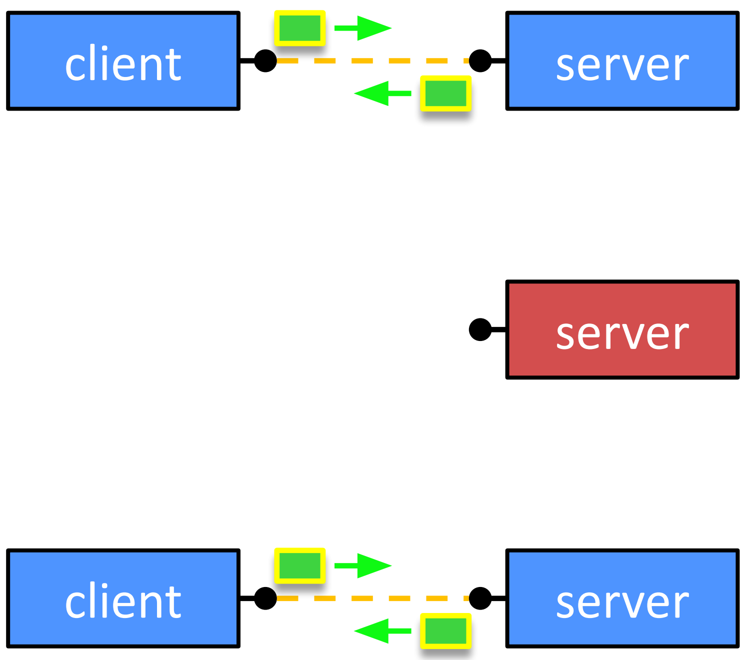
Double-fork Trick



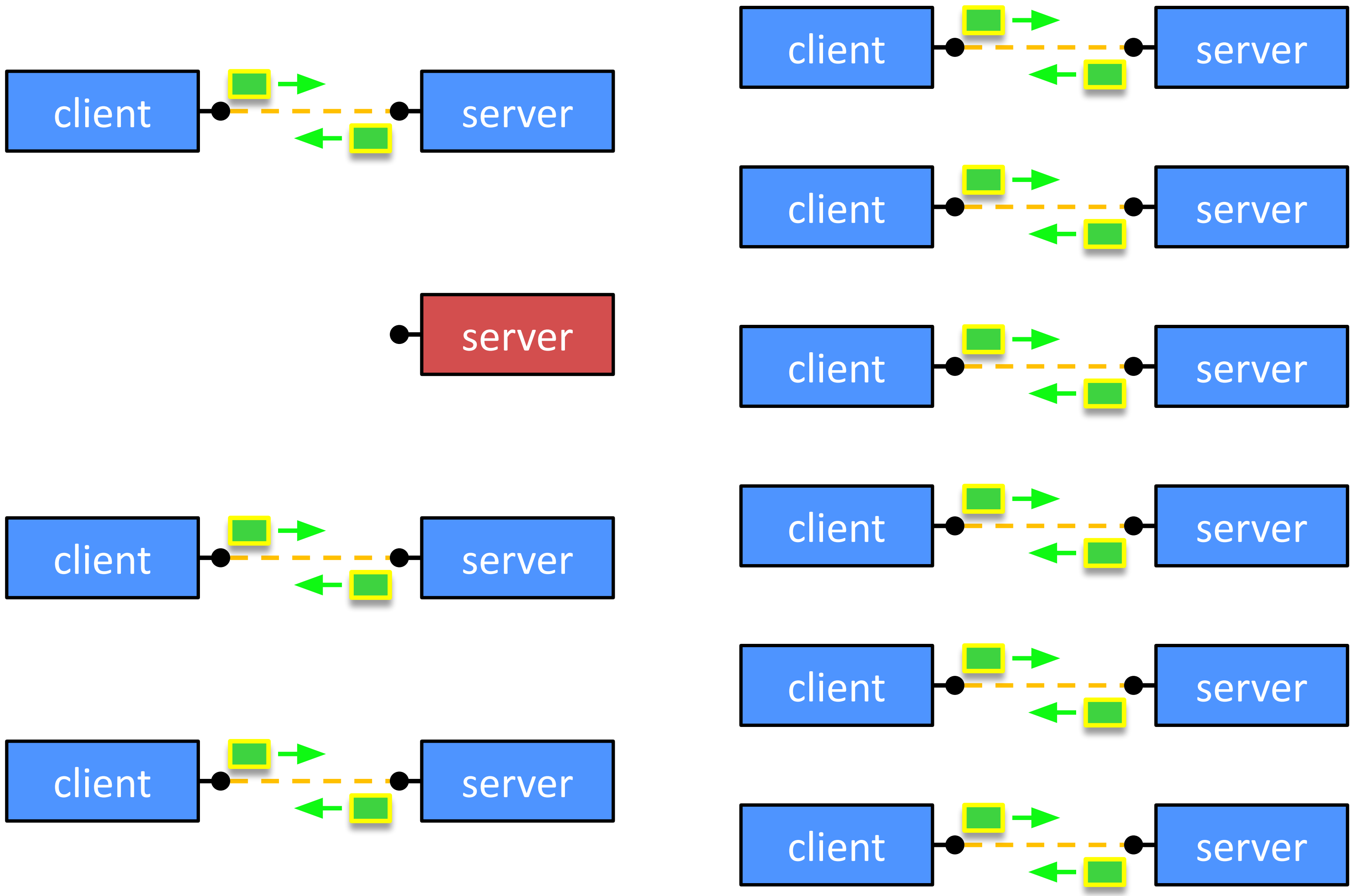
Double-fork Trick



Double-fork Trick



Double-fork Trick



Double-fork Trick

- With the double fork trick:
  - There's no parent to read the exit code
  - Therefore the OS knows to clean it up right away.

Concurrent Server with Processes

See:

`searchserver_processes/searchserver.cc`

https://courses.cs.washington.edu/courses/cse333/25su/lecture/23-processes-example

How Fast is `fork()`?

See:

`forklatency.cc`

`threadlatency.cc`

https://courses.cs.washington.edu/courses/cse333/25su/lecture/23-processes-example

How Fast is `fork()`?

- ~**0.2ms** per fork*
  - Maximum of (1000/0.2) = 5,000 connections/sec/core
  - ~430 million connections/day/core
    - This is fine for most servers
    - Two slow for super-high-traffic front-line web services
      - Facebook served ~750 billion page views per day in 2013!
      - Would need 2k cores just to handle `fork()`, i.e. without doing any work for each connection

* Exact past measurements are not indicative of future performance, just their rough ratios - actual measurement depends on hardware and software versions.

How Fast is `pthread_create()`?

- ~**0.018ms** per thread create*
  - Maximum of (1000/0.018) = 56,000 connections/sec/core
  - ~4.8 billion connections/day/core
- Much faster, but writing safe multithreaded code is really hard

* Exact past measurements are not indicative of future performance, just their rough ratios - actual measurement depends on hardware and software versions.

Aside: Thread/Process Pools

- In real servers, we'd like to avoid overhead needed to create a new thread or process for every request
- Idea: Thread/Process *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
- Provides faster client connection acceptances and more control over total resource usage.

Don't Forget

- hw4 due **Wednesday night (August 20th)**
  - Usual late days (2 max) apply if you have any remaining
- Final exam **Fri. August 22nd**, 1:10-2:10, HRC 155
- Please nominate great TAs for the Bandes award when nominations are available
- Course evals are available
- Section this week is an exam review… show up!
- Office hours this week get you extra points on the final