Concurrency Via Processes

CSE 333

Instructor: Hannah C. Tang

Teaching Assistants:

Deeksha Vatwani	Hannah Jiang	Jen Xu
Justin Tysdal	Leanna Nguyen	Sayuj Shahi
Wei Wu	Yiqing Wang	Youssef Ben Taleb

Poll Everywhere

pollev.com/uwcse333

- Consider two threads running within the same process. Which of the following do they SHARE?
  - In-use resources such as file handles and sockets
  - System-available resources such as the file system or IP addresses
  - Call stack
  - Registers (eg, PC, SP)
  - Virtual memory (page tables, TLBs, etc …)

Administrivia

- HW4 due tomorrow night
  - With late days, can even be Thursday night (if you have them)

- Ex17 due Wednesday morning
- Ex 18 (the last one!) is Gradescope-only, due Friday
  - Final-exam prep

- Awkward amount of time on Wednesday's lecture (~20m of topics); choose your own adventure!
  - Conversion tracking / Tracking pixels
  - Event-based concurrency

Lecture(s) Outline

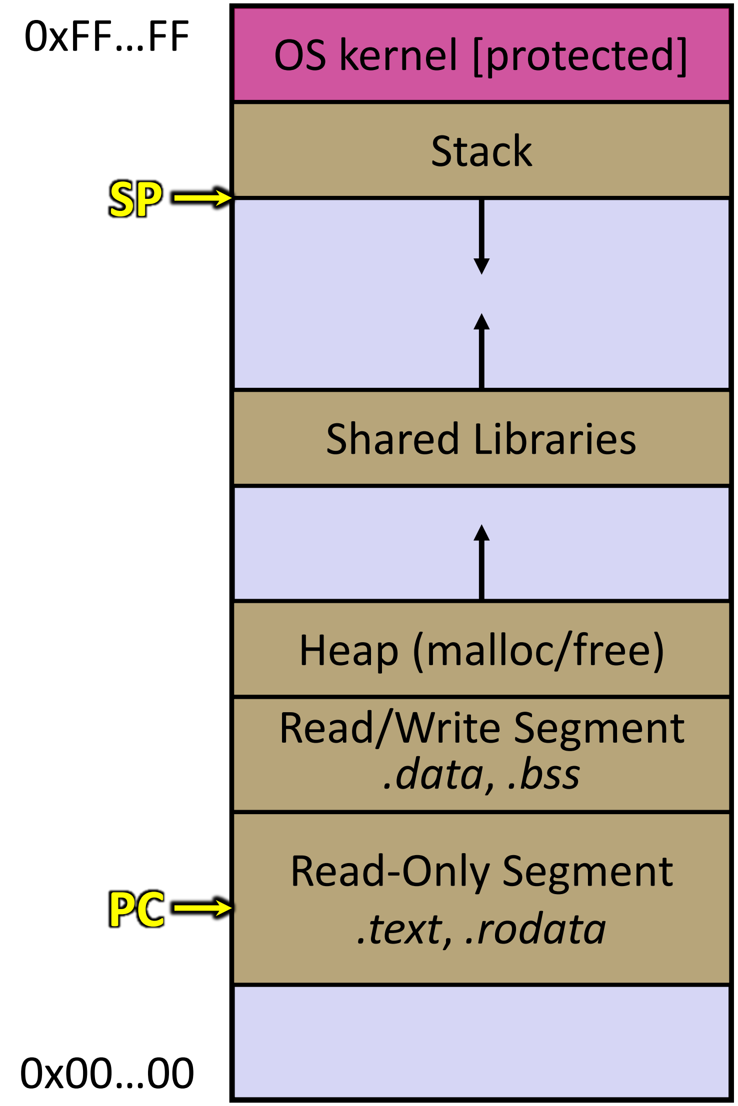
- `searchserver`
  - Sequential
  - Concurrent via threads: `pthread_create()`
    - Implementation using dispatching threads
    - Data Races
  - **Concurrent via forking processes: `fork()`**
  - *Supplement: Concurrent via events:* `select()`
  - Conclusion

Creating New Processes

- `pid_t fork(void);`
  - Creates a new process (the "child") that is a *clone** of the current process (the "parent")
    - * Everything is cloned except threads
    - Variables, file descriptors, open sockets, the virtual address space (code, globals, heap, stack), etc are all cloned
  - Primarily used in two patterns:
    - Servers: fork a child to handle a connection
    - Shells: fork a child that then exec's a new program

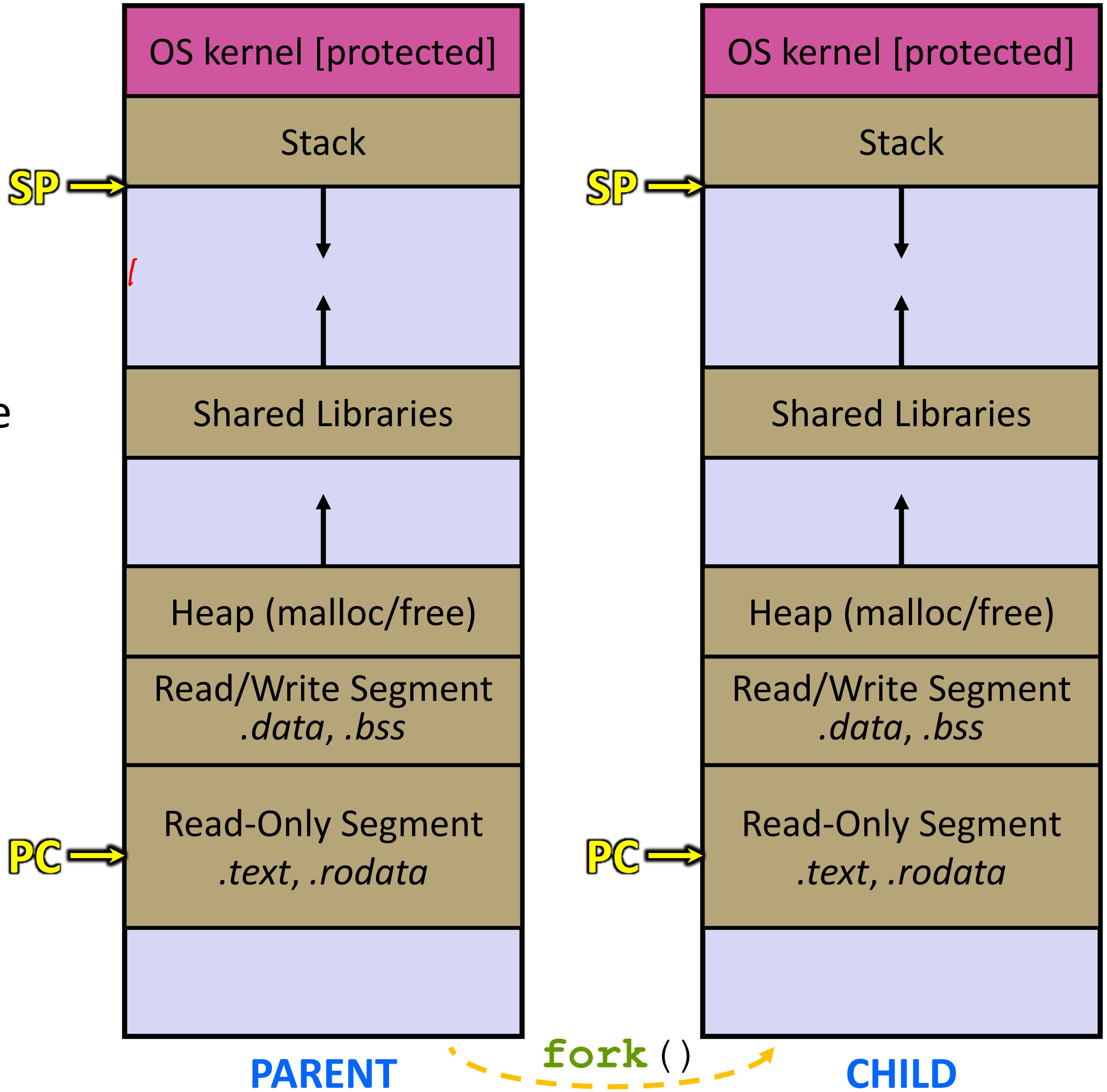
`fork()` and Address Spaces

- 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)

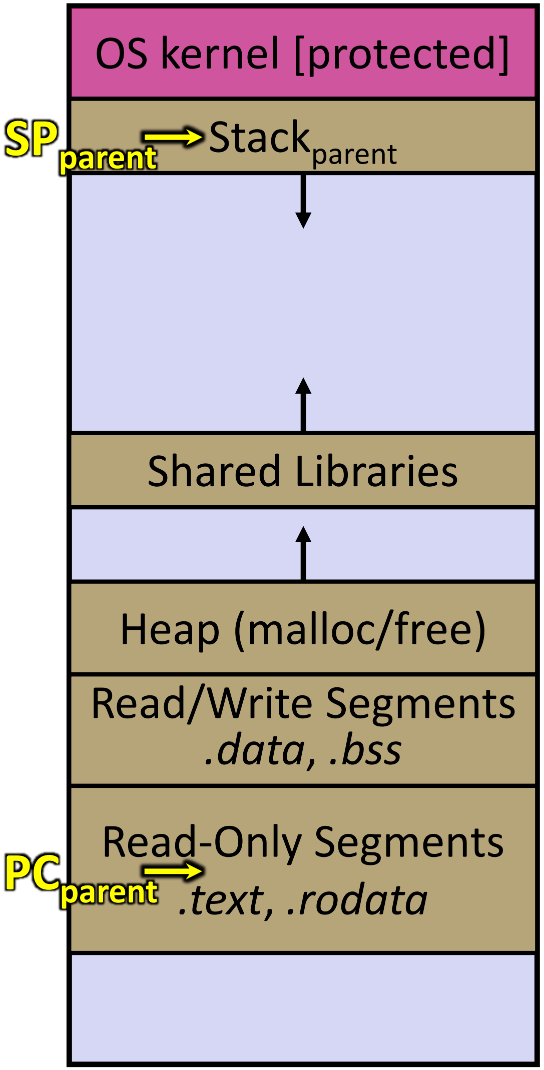


`fork()` and Address Spaces

- Fork cause 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.

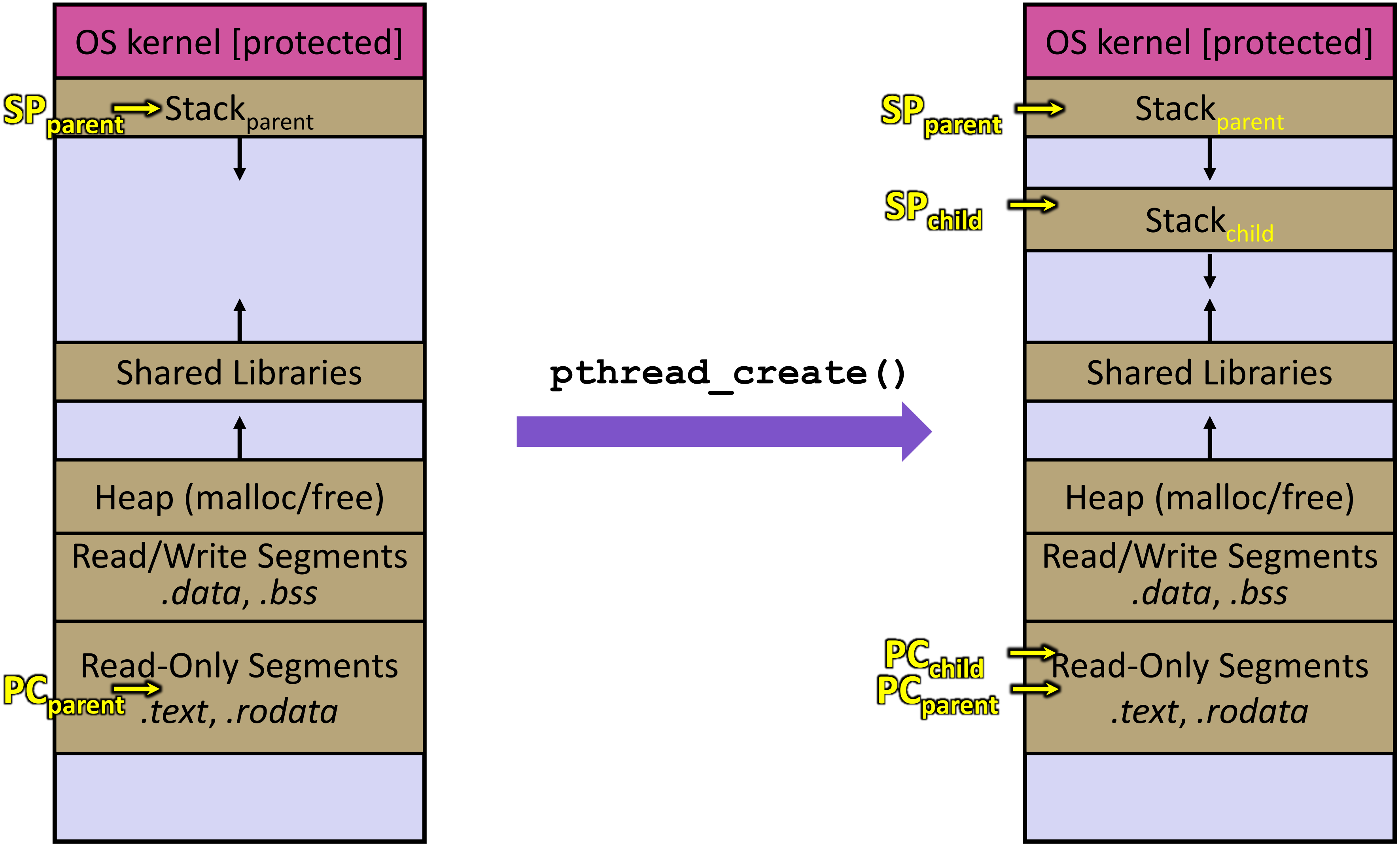


Threads vs. Processes

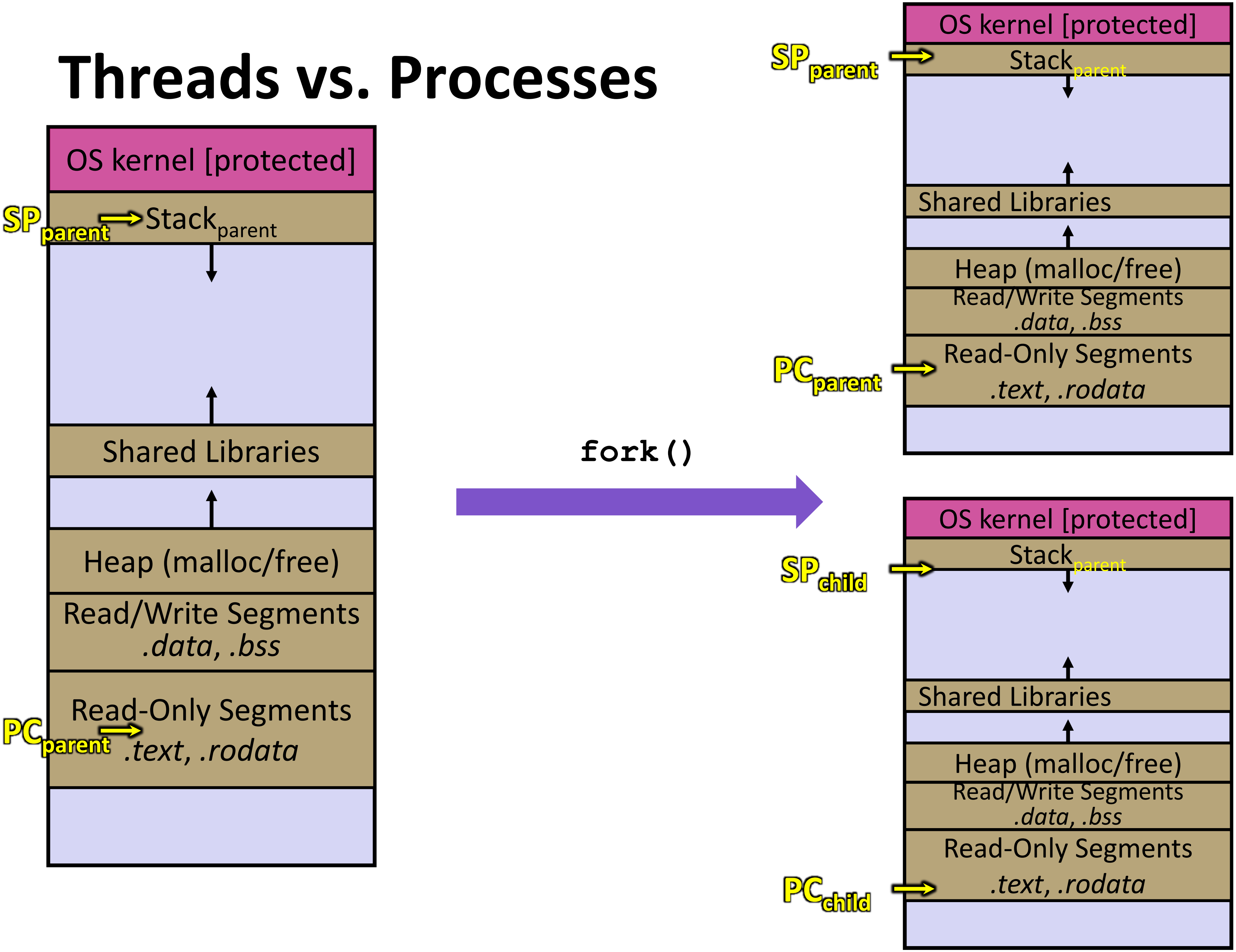


- Before creating a thread
  - One thread of execution running in the address space
    - One PC, stack, SP
  - That main thread invokes a function to create a new thread
    - Typically `pthread_create()`

Threads vs. Processes

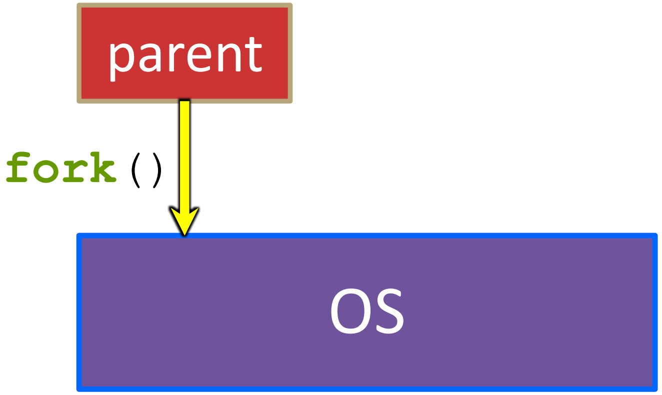


Threads vs. Processes



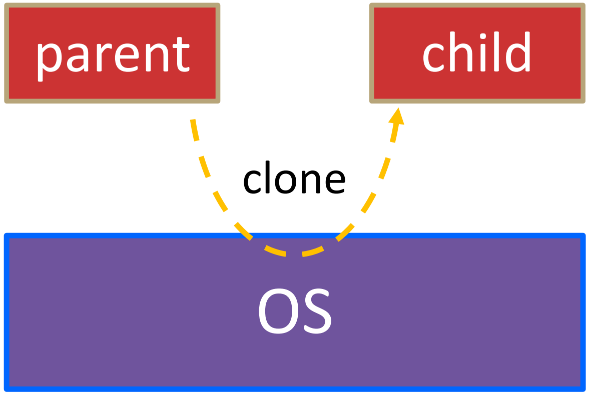
fork()

- 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



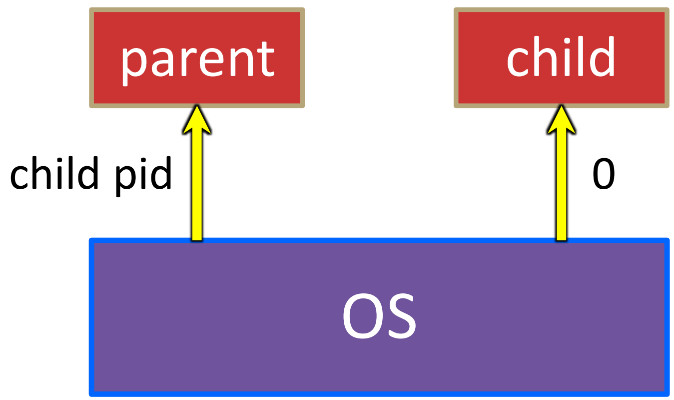
`fork()`

- `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



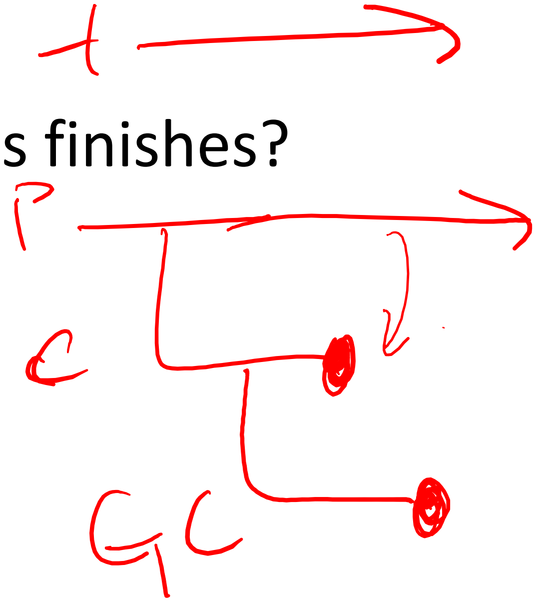
fork()

- 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

Poll Everywhere pollev.com/uwcse333



What happens when a grandchild process finishes?

A. Zombie until grandparent exits
B. ~~Zombie until grandparent reaps~~
C. Zombie until `systemd` reaps
D. ZOMBIE FOREVER!!!
E. I’m not sure…

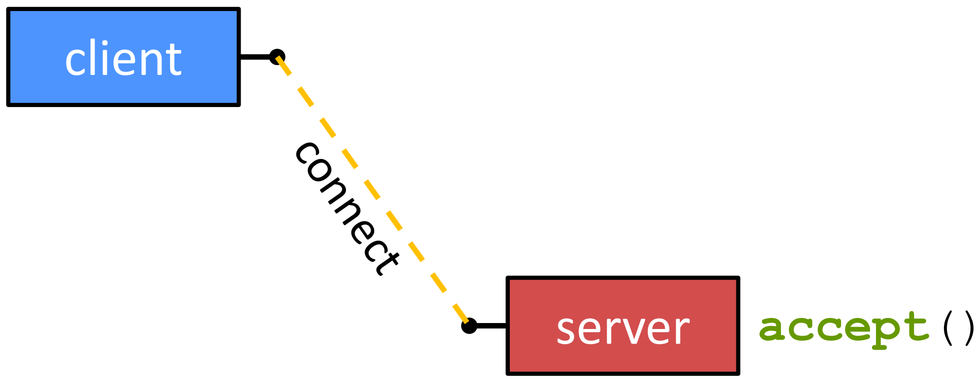
Concurrent Server with Processes

- 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

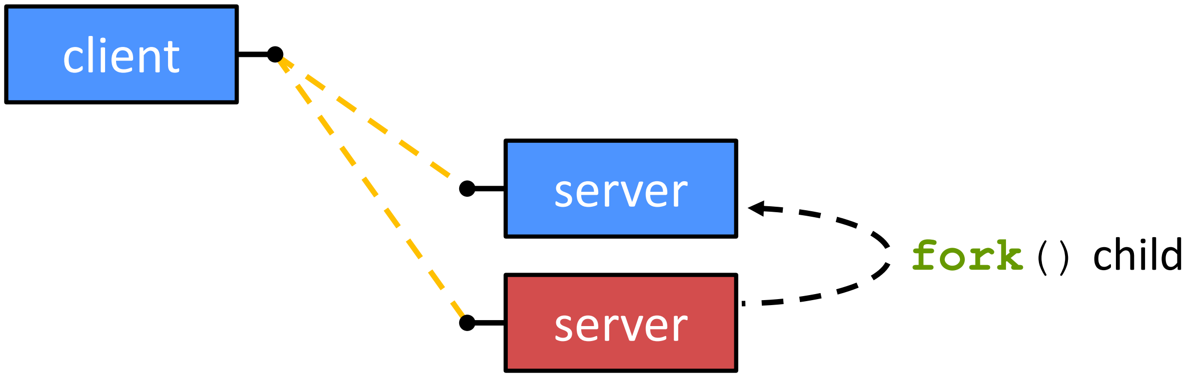
- Remember that children become "zombies" after termination
  - Option A: Parent calls `wait()` to "reap" children
  - Option B: Use a double-fork trick

Double-fork Trick

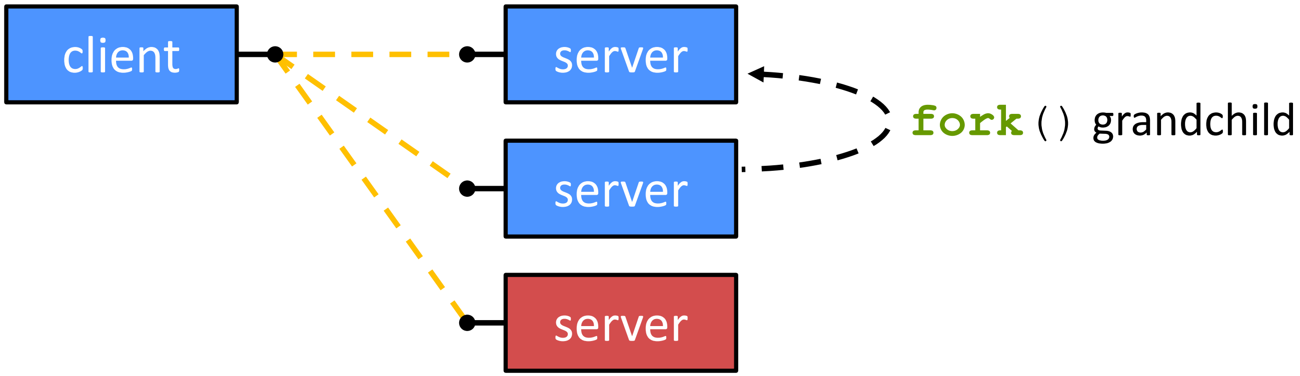
Double-fork Trick



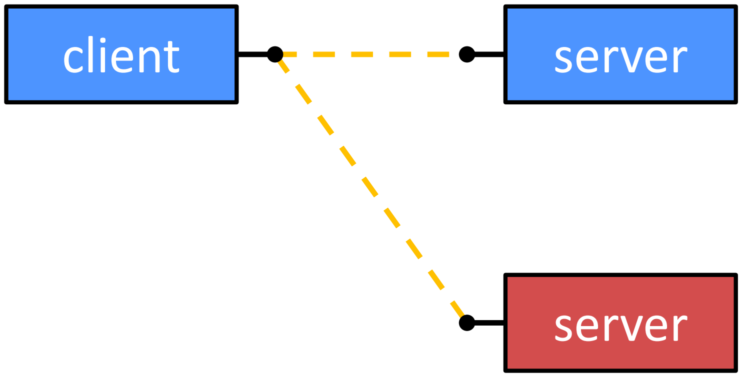
Double-fork Trick



Double-fork Trick

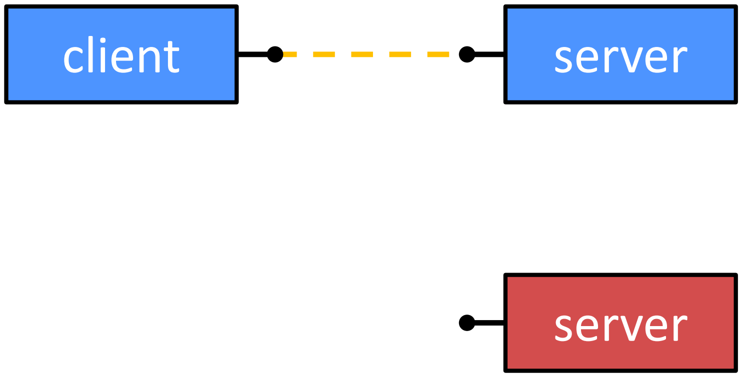


Double-fork Trick



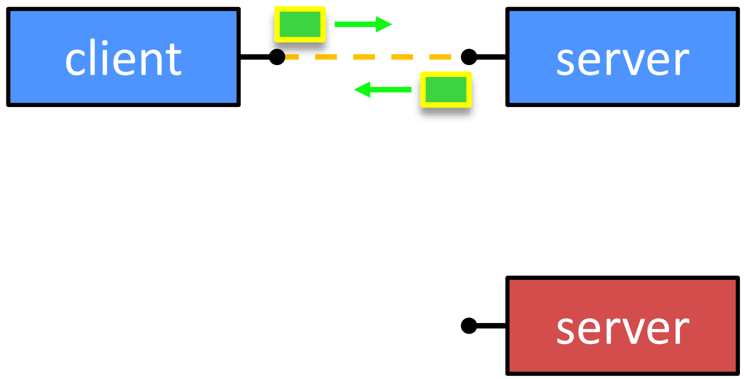
child **exit**()'s / parent **wait**()'s

Double-fork Trick

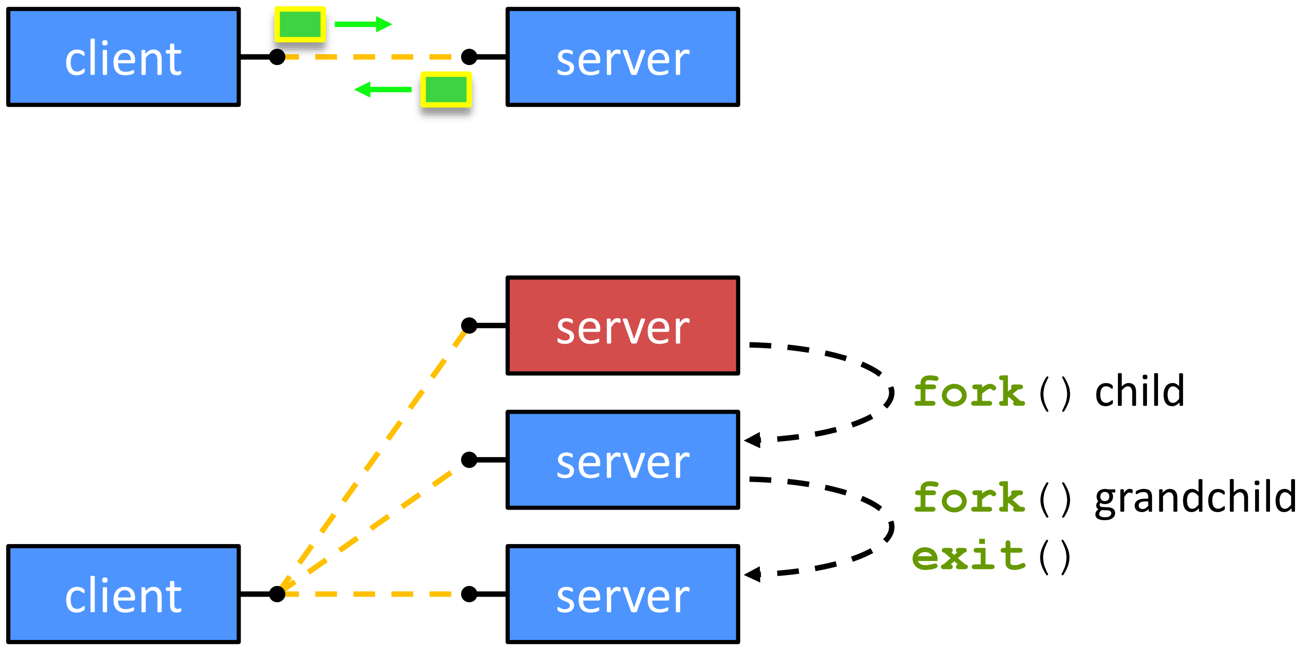


parent closes its
client connection

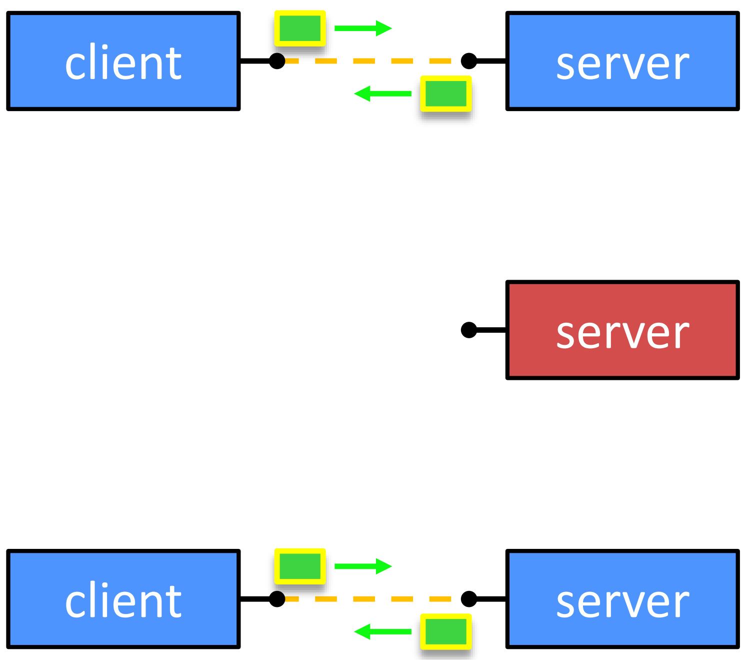
Double-fork Trick



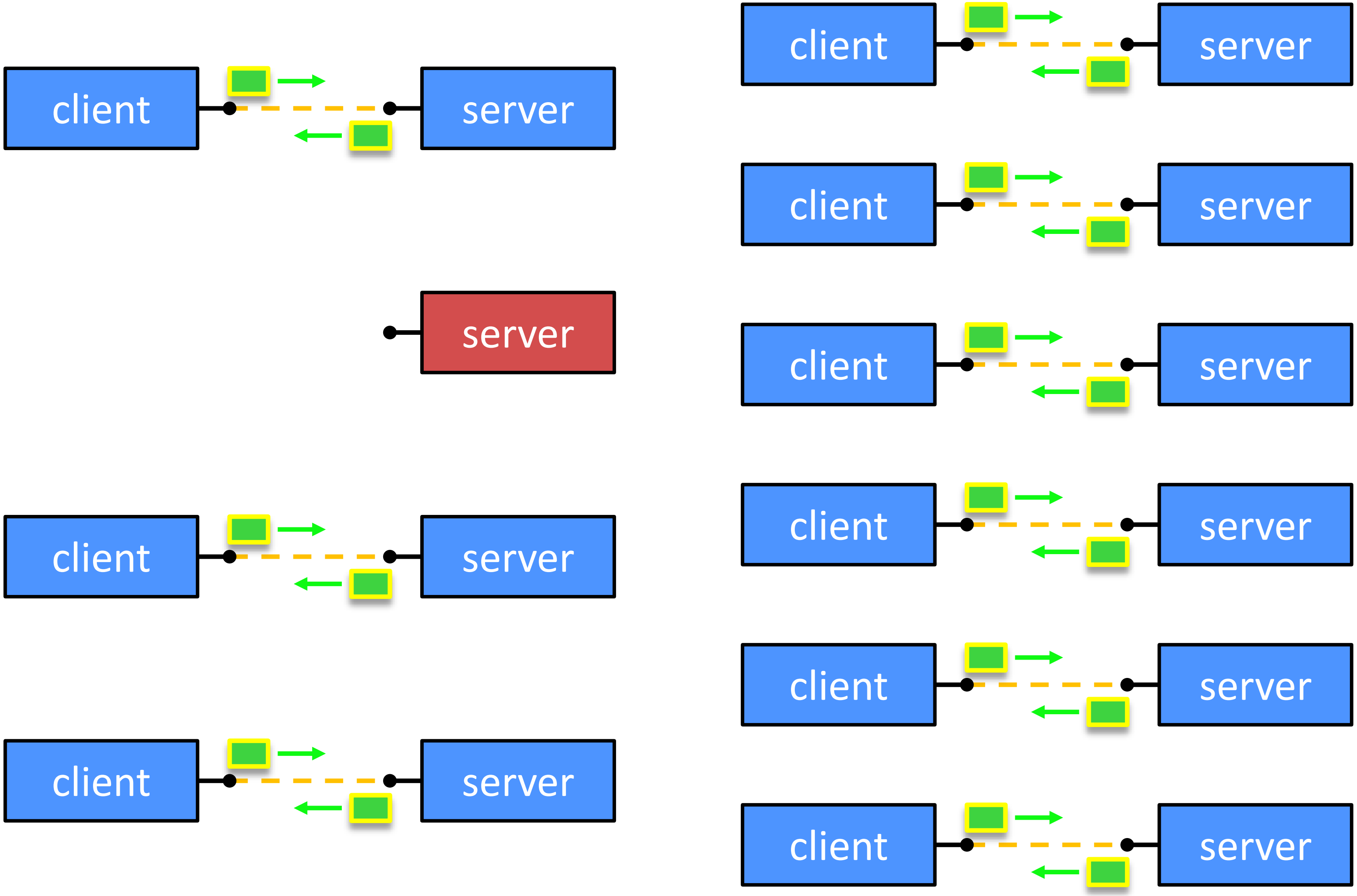
Double-fork Trick



Double-fork Trick



Double-fork Trick



Concurrent with Processes

- See `searchserver_processes/`

Wherefore Concurrent Processes?

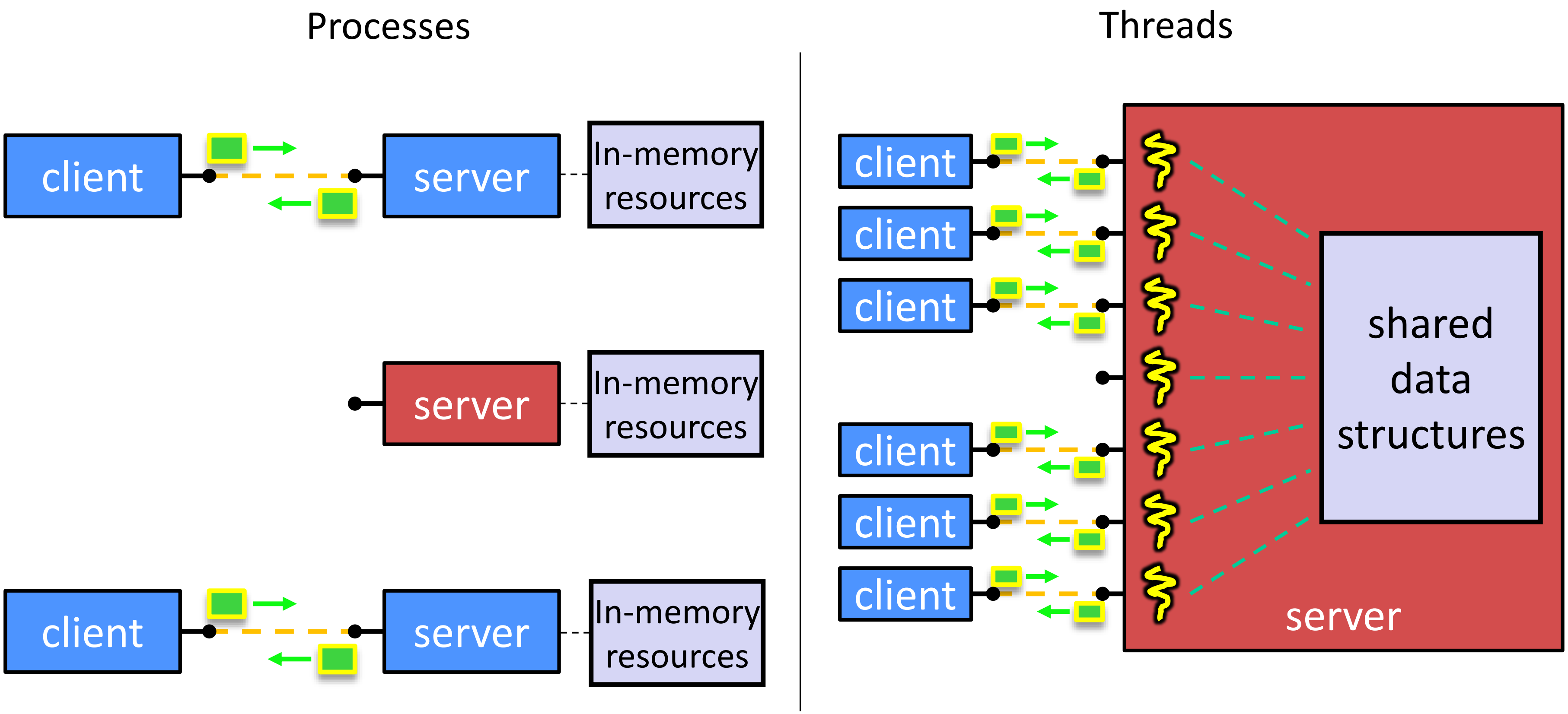
- Advantages:
  - Almost as simple to code as sequential
    - In fact, most of the code is identical!
  - Concurrent execution leads to better CPU, network utilization

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

Lecture(s) Outline

- `searchserver`
  - Sequential
  - Concurrent via threads: `pthread_create()`
    - Implementation using dispatching threads
    - Data Races
  - Concurrent via forking processes: fork()
  - ***Supplement: Concurrent via events: `select()`***
  - Conclusion

Review: Multi-“worker” Search Engine



*“The child process/thread handles that new connection **and subsequent I/O**, then calls `exit()`/`pthread_exit()` when the connection terminates”*

Event-Driven Programming

- Your program is structured as an *event-loop* consisting of (mostly) independent, stateless tasks executing in any order

any necessary state is held outside of your event handler

```
void ProcessOneTask(state) {
  query_words = state.buffer;
  for (idx : state.indices) {
    ...
  }
  ...
}

while (1) {
  event = OS.GetNextEvent();
  state = GetState(event);
  ProcessOneTask(state);
}
```

your application code ("event handler"). Typically a dispatcher into more specialized sub-handlers

typically framework code ("event loop")

Asynchronous I/O and Event-Driven Programming

- Use asynchronous or non-blocking I/O
- Your program begins processing a query
  - When your program needs to read data to make further progress, it registers interest in the data with the OS and then switches to a different query
  - The OS handles the details of issuing the read on the disk, or waiting for data from the console (or other devices, like the network)
  - When data becomes available, the OS lets your program know
- Your program (almost never) blocks on I/O

One Way to Think About It

- Threaded code:
  - OS and thread scheduler switch between threads for you
  - Each thread executes its task sequentially, and per-task state is naturally stored in the thread's stack

- Event-driven code:
  - ***You*** (or your framework) are the scheduler
    - You (or your framework) also manages scheduling-related resources, such as the connection
  - You have to bundle up task state into *continuations* (data structures describing what-to-do-next); tasks do not have their own stacks

    the "state" in our pseudocode
  - … what if your logic required multiple steps?
    - Read from one index, then read from another index, then …

Multi-Step Event-Driven Programming

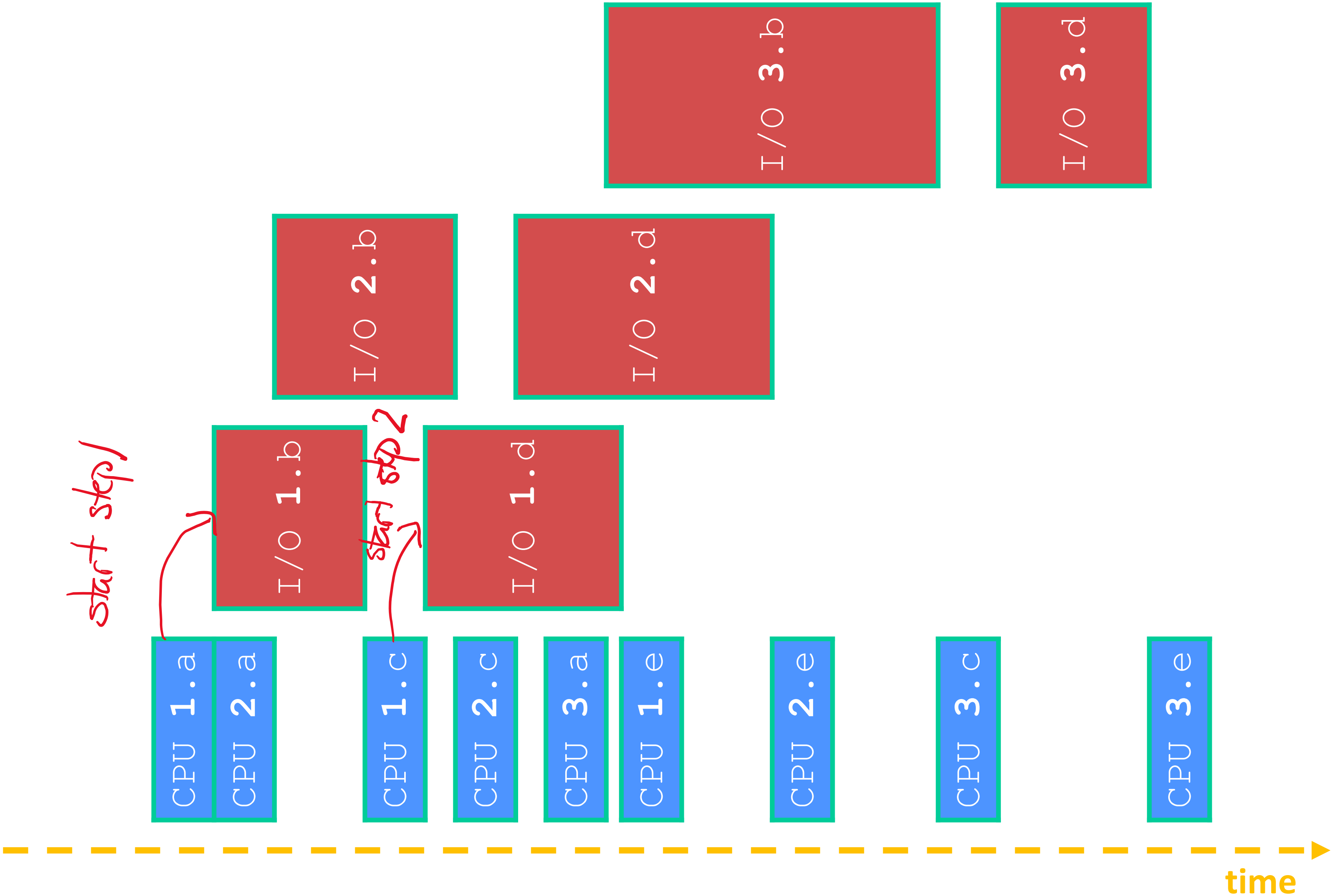
- Each step is a brand-new event
  - Task state must include information about which step we're on

dispatches into specialized sub-handlers

```
void dispatch(task, event) {
  switch (task.state) {
    case READING_FROM_CONSOLE:   step 1
      query_words = event.query;
      async_read(index, query_words[0]);
      task.state = READING_FROM_INDEX;
      return;
    case READING_FROM_INDEX:   step 2
      results = event.results;
      ...
  }                            step N
}

while (1) {
  event = OS.GetNextEvent();
  task = lookup(event);
  dispatch(task, event);
}
```

Multi-Step, Event-Driven w/Async I/O



Non-blocking I/O vs. Asynchronous I/O

- Asynchronous I/O (disk)
  - Program tells the OS to begin reading/writing
    - The "begin_read" or "begin_write" returns immediately
    - When the I/O completes, OS delivers an event to the program
  - According to the Linux specification, the disk never blocks your program (just delays it)
    - Asynchronous I/O is primarily used to hide disk latency
    - Asynchronous I/O system calls are messy and complicated ☹
  - Reading from the network can truly *block* your program
    - Remote computer may wait arbitrarily long before sending data

Non-blocking I/O vs. Asynchronous I/O

- Non-blocking I/O (network, console)
  - Your program enables non-blocking I/O on its file descriptors
  - Your program issues `read()` and `write()` system calls
    - If the read/write would block, the system call returns immediately
  - Program can ask the OS which file descriptors are readable/writeable
    - Program can choose to

Why Events?

- Advantages:
  - Don't have to worry about locks and race conditions
  - For some kinds of programs, especially GUIs, leads to a very simple and intuitive program structure
    - One event handler for each UI event

- Disadvantages:
  - Can lead to very complex structure for programs that do lots of disk and network I/O
    - Sequential code gets broken up into a jumble of small event handlers
    - You have to package up all task state between handlers

Lecture(s) Outline

- `searchserver`
  - Sequential
  - Concurrent via threads: **`pthread_create`**()
    - Implementation using dispatching threads
    - Data Races
  - Concurrent via forking processes: fork()
  - *Supplement: Concurrent via events:* **`select`**()
  - **Conclusion**

How Fast is `fork()`?

- See `forklatency.cc`

- ~ **0.25 ms** per fork*
  - ∴ maximum of (1000/0.25) = 4,000 connections/sec/core
  - ~350 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`

- ~**0.036 ms** per thread creation*
  - ~10x faster than **`fork()`**
  - ∴ maximum of (1000/0.036) = 28,000 connections/sec
  - ~2.4 billion connections/day/core

- Much 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

- Idea: Thread 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

Why Sequential?

- Advantages:
  - Simple to write, maintain, debug
  - The default. Supported everywhere!
- Disadvantages:
  - Depending on application, poor performance
    - One slow client will cause *all* others to block
    - Poor utilization of resources (CPU, network, disk)

Why Concurrent Threads?

- Advantages:
  - Almost as simple to code as sequential
  - Concurrent execution with good CPU and network utilization
  - Threads can run in parallel if you have multiple CPUs/cores
  - Shared-memory communication is possible

- Disadvantages:
  - Need language and OS support for threads
  - If threads share data, you need locks or other synchronization
  - Threads can introduce overhead (technical + cognitive)
  - Threads have a “shared fate” (eg, “rogue” thread, shared limits)

Why Concurrent Processes?

- Advantages:
  - Almost as simple to code as sequential
  - Concurrent execution with good CPU and network utilization
  - Processes almost certainly run in parallel thanks to OS time-sharing
  - No need to synchronize access to in-memory structures

- Disadvantages:
  - Processes are heavyweight
    - Relatively slow to fork and context switching latency is high
  - Communication between processes is complicated – no shared memory
  - Fewer things to synchronize – but when you do need to synchronize, it's hard! – e.g., disk based locks? shared "master" to hold lock?

Why Events?

- Advantages:
  - For some kinds of programs – those with mostly-stateless, simple responses – leads to very simple and intuitive program
    - Eg, GUIs: one event handler for each UI event

- Disadvantages:
  - Can lead to very complex structure for some programs
    - Sequential logic gets broken up into a jumble of small event handlers
    - You have to package up all task state between handlers