

# Compulsory bad joke

- ❖ **Client:** “Do you know any jokes?”
- ❖ **Multi-threaded Server:** “I have a wonderful concurrency. joke about ”

-- The Internet

# Concurrency: Processes

## CSE 333 Autumn 2021

**Instructor:** Chris Thachuk

**Teaching Assistants:**

Arpad (John) Depaszthory

Ian Hsiao

Logan Gnanapragasam

Mengqi (Frank) Chen

Angela Xu

Khang Vinh Phan

Maruchi Kim

Cosmo Wang

# Administrivia

- ❖ Exercise 16 due Wednesday (Dec. 8)
  - Concurrency via pthreads
- ❖ hw4 due next Thursday (Dec. 9)
- ❖ Next week: more systems programming goodness
  - Rust, writing faster code by leveraging advanced CPU instructions, fun demos
  - No exercises / homework on that material

# Lecture Outline

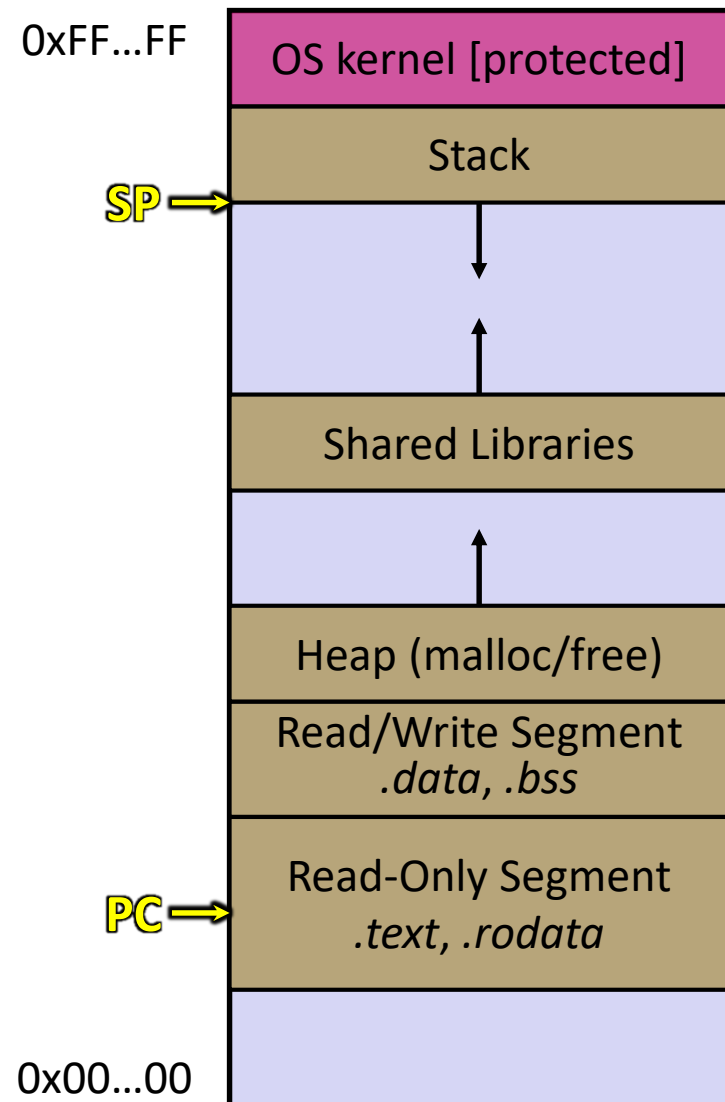
- ❖ **Review: Processes vs Threads**
- ❖ `fork()` details
  - Concurrent Server with Processes
- ❖ Conclusion

# Previous Lectures

- ❖ `searchserver`
  - 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)

# Review: Address Spaces

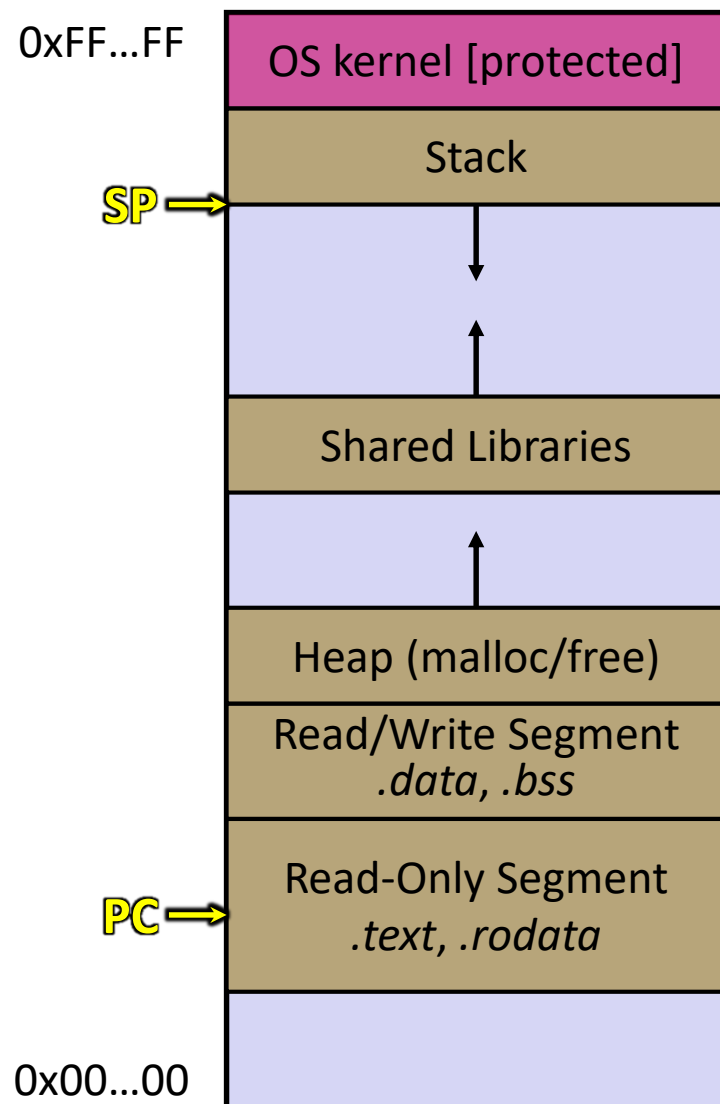
- ❖ A process has its own *address space*
  - Includes segments for different parts of memory
  - A process usually has one or more threads
    - A thread tracks its current state using the **stack pointer** (SP) and **program counter** (PC)



# Review: Address Spaces

- ❖ A process has its own *address space*
  - Includes segments for different parts of memory
  - A process usually has one or more threads
    - A thread tracks its current state using the **stack pointer** (SP) and **program counter** (PC)
- ❖ New processes are created with:

```
pid_t fork();
```



# Main Uses of `fork()`

# Main Uses of `fork()`

- Fork a child to handle some work
  - Server forks to handle a new connection
  - Web browser forks to render a new website
    - Mainly for security purposes (separate address spaces)



# Main Uses of `fork()`

- Fork a child to handle some work
  - Server forks to handle a new connection
  - Web browser forks to render a new website
    - Mainly for security purposes (separate address spaces)
- Fork a child that then `exec`'s a new program
  - Shell forks and `execs` the program you want to run
  - 333 grading script **forks** and **execs** your executable



# Main Uses of `fork()`

- Fork a child to handle some work
  - Server forks to handle a new connection
  - Web browser forks to render a new website
    - Mainly for security purposes (separate address spaces)
- Fork a child that then `exec`'s a new program
  - Shell forks and `execs` the program you want to run
  - 333 grading script **forks** and **execs** your executable
- Fork a “background” process that is not interactive
  - Runs independently, doesn't need a user to be logged in
  - Called a “Daemon” Process in Linux



# Main Uses of `fork()`

- Fork a child to handle some work
  - Server forks to handle a new connection
  - Web browser forks to render a new website
    - Mainly for security purposes (separate address spaces)
- Fork a child that then exec's a new program
  - Shell forks and execs the program you want to run
  - 333 grading script **forks** and **execs** your executable
- Fork a “background” process that is not interactive
  - Runs independently, doesn't need a user to be logged in
  - Called a “Daemon” Process in Linux



# Process Isolation

- ❖ **Process Isolation** is a set of mechanisms implemented to protect processes from each other and protect the kernel from user processes.

# 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

# 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()`

# How Fast is `fork()` ?

See [forklatency.cc](http://forklatency.cc)

- maximum of  $(1000/0.5) = 2,000$  connections/sec/core
- ~175 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, ...
  - Processes are known to be even slower on Windows

# How Fast is `fork()` ?

maximum of  $(1000/0.5) = 2,000$  connections/sec/core

See [forklatency.cc](http://forklatency.cc)

- ❖ **~ 0.5 milliseconds per fork\***
  - ~175 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
      - 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,

# How Fast is `fork()` ?

maximum of  $(1000/0.5) = 2,000$  connections/sec/core

See [forklatency.cc](http://forklatency.cc)

- ❖ **~ 0.5 milliseconds per fork\***
  - ~175 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
      - 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, ...
- ❖ \*Past measurements are not indicative of future performance – depends on hardware, OS

# How Fast is `fork()` ?

maximum of  $(1000/0.5) = 2,000$  connections/sec/core

See [forklatency.cc](http://forklatency.cc)

- ❖ **~ 0.5 milliseconds per fork\***
  - ~175 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
      - 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, ...
  - Processes are known to be even slower on Windows

# How Fast is `pthread_create()` ?

See [threadlatency.cc](http://threadlatency.cc)

- ❖ **0.05 milliseconds** per thread creation\*
  - ~10x faster than `fork()`
  - $\therefore$  maximum of  $(1000/0.05) = 20,000$  connections/sec/core
  - ~2 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()`

# How Fast is `pthread_create()` ?

2 billion connections/day/core

maximum of  $(1000/0.05) = 20,000$  connections/sec/core

10x faster than `fork()`

**0.05 milliseconds** per thread creation\*

See [threadlatency.cc](http://threadlatency.cc)

- ~2 billion connections/day/core
  - ~10x faster than `fork()`
  - $\therefore$  maximum of  $(1000/0.05) = 20,000$  connections/sec/core
  - ~2 billion connections/day/core
- ❖ Much faster, but writing safe multithreaded code can be serious voodoo

# How Fast is `pthread_create()` ?

2 billion connections/day/core

maximum of  $(1000/0.05) = 20,000$  connections/sec/core

10x faster than `fork()`

0.05 milliseconds per thread creation\*

See [threadlatency.cc](http://threadlatency.cc)

- ~2 billion connections/day/core
  - ~10x faster than `fork()`
  - $\therefore$  maximum of  $(1000/0.05) = 20,000$  connections/sec/core
  - ~2 billion connections/day/core
- ❖ Much faster, but writing safe multithreaded code can be serious voodoo

# How Fast is `pthread_create()` ?

2 billion connections/day/core

maximum of  $(1000/0.05) = 20,000$  connections/sec/core

10x faster than `fork()`

0.05 milliseconds per thread creation\*

See [threadlatency.cc](http://threadlatency.cc)

- ~2 billion connections/day/core
- ❖ Much faster, but writing safe multithreaded code can be serious voodoo
  - $\therefore$  maximum of  $(1000/0.05) = 20,000$  connections/sec/core
  - ~2 billion connections/day/core
- ❖ Much faster, but writing safe multithreaded code can be

# How Fast is `pthread_create()` ?

2 billion connections/day/core

maximum of  $(1000/0.05) = 20,000$  connections/sec/core

10x faster than `fork()`

0.05 milliseconds per thread creation\*

See [threadlatency.cc](http://threadlatency.cc)

- ~2 billion connections/day/core
- ❖ Much faster, but writing safe multithreaded code can be serious voodoo
  - $\therefore$  maximum of  $(1000/0.05) = 20,000$  connections/sec/core
- ❖ \*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()`
- ❖ Much faster, but writing safe multithreaded code can be

# Lecture Outline

- ❖ Review: Processes vs Threads
- ❖ `fork()` details
  - **Concurrent Server with Processes**
- ❖ Conclusion

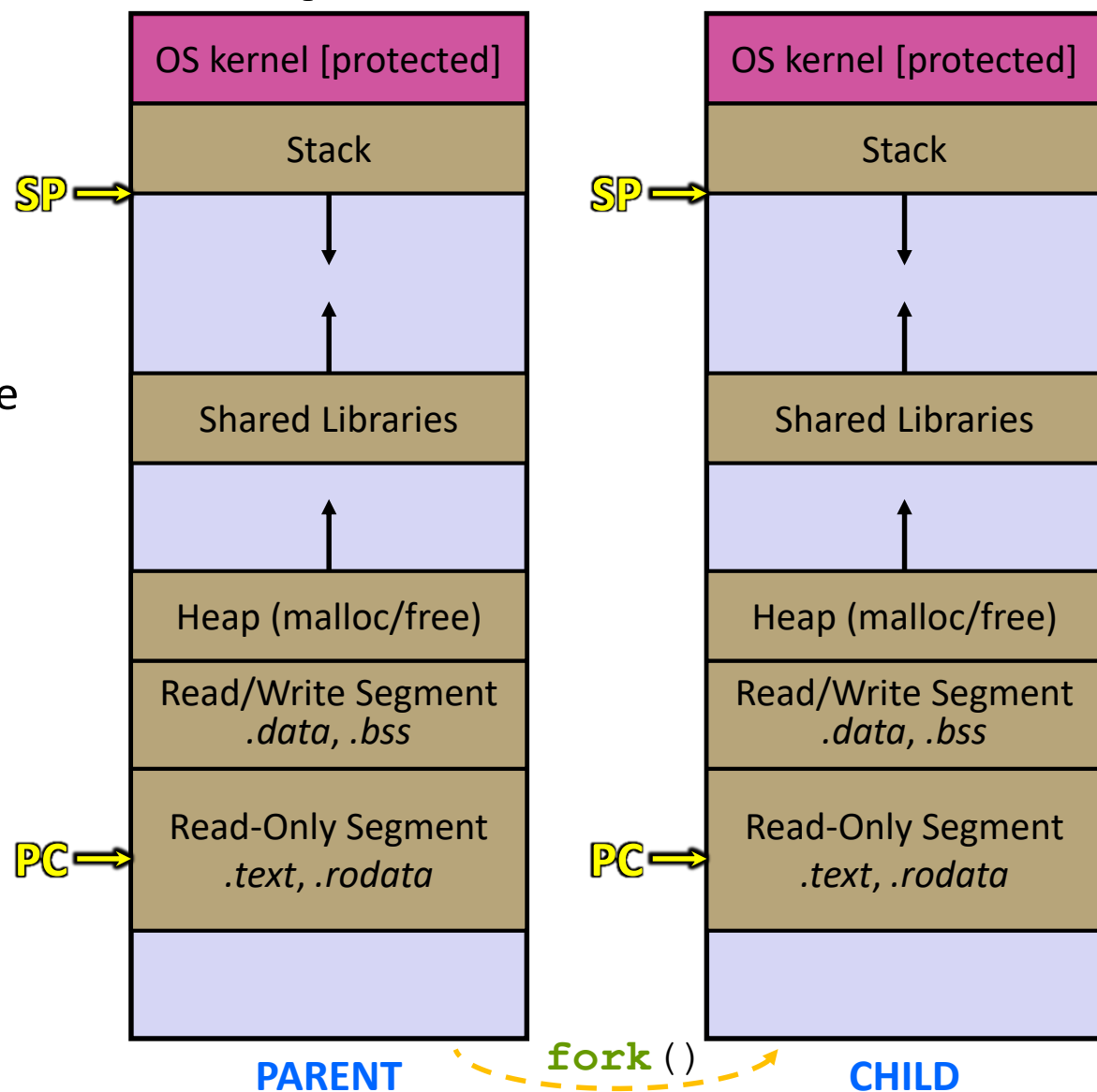
# Creating New Processes

❖ `pid_t fork();`

- Creates a new process (the “child”) that is an *exact clone*\* of the current process (the “parent”)
  - \*Everything is cloned except threads. Sockets, file descriptors, variables, etc.
- The new process has a separate virtual address space from the parent

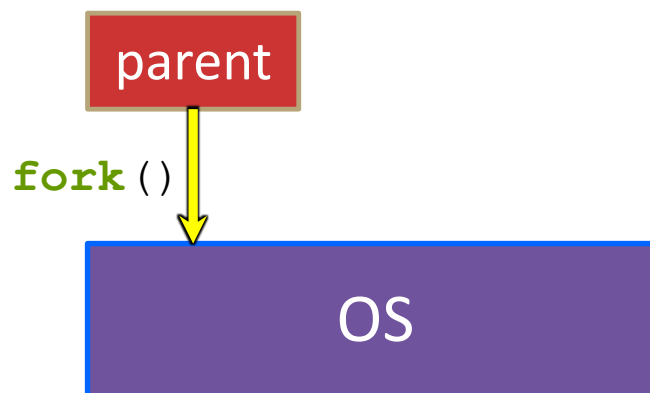
# fork () and Address Spaces

- ❖ 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.



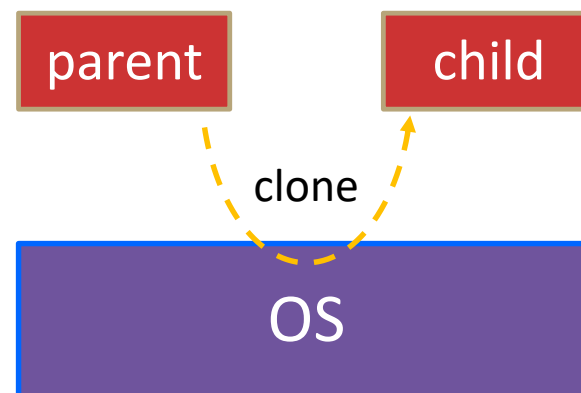
# fork ()

- ❖ **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



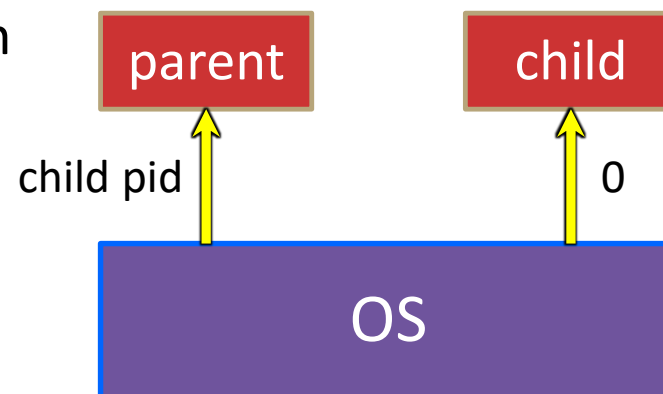
# fork ()

- ❖ **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



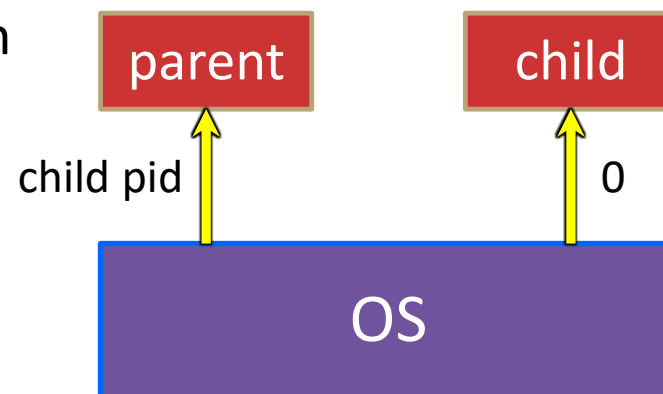
# fork ()

- ❖ **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



# fork ()

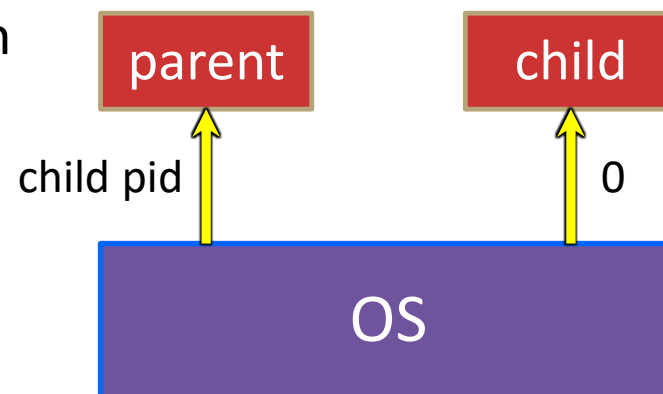
- ❖ **fork ()** has peculiar semantics
  - The parent invokes **fork ()**
  - The OS clones the parent
  - *Both* the parent and the child return from fork
    - Parent receives child's pid
    - Child receives a 0



- ❖ See `fork_example.cc`

# 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`

CSE 351 review:  
When a child process exits, it is a zombie until its parent reaps it.

# 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

- ❖ 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 death

# 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 death
  - Option A: Parent calls **wait** ( ) to “reap” children

# 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 death
  - Option A: Parent calls **wait** ( ) to “reap” children

Don't want to block, waiting for client to finish being handled

# Concurrent Server with Processes

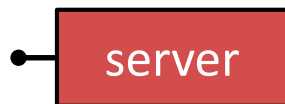
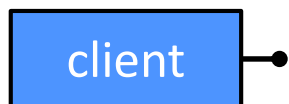
- ❖ The **parent** process blocks on **accept** ( ) , waiting for a new client to connect
  - When a new connection arrives, the parent calls **fork** ( ) to create a **child** process
  - The child process handles that new connection and **exit** ( ) 's when the connection terminates
- ❖ Remember that children become “zombies” after death
  - Option A: Parent calls **wait** ( ) to “reap” children
  - Option B: Use a **double-fork trick**

Don't want to block, waiting for client to finish being handled

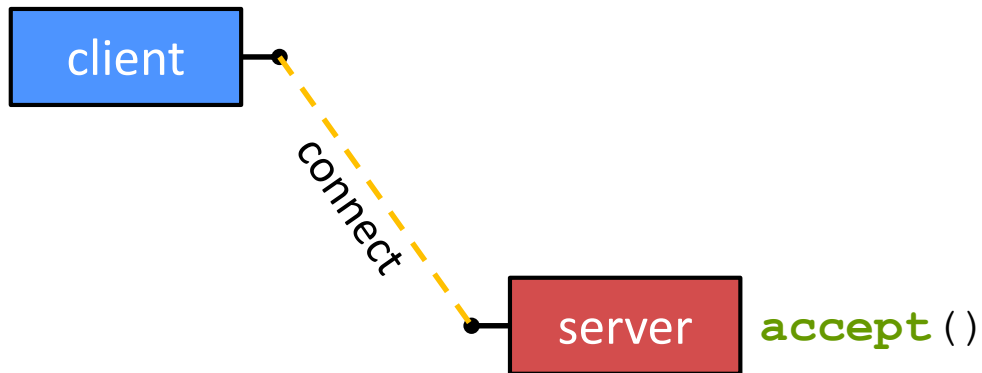
# Double-fork Trick



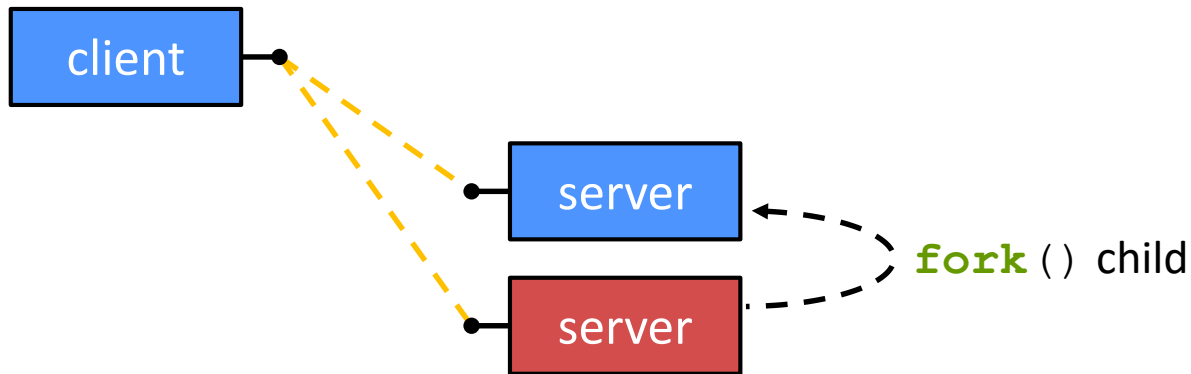
# Double-fork Trick



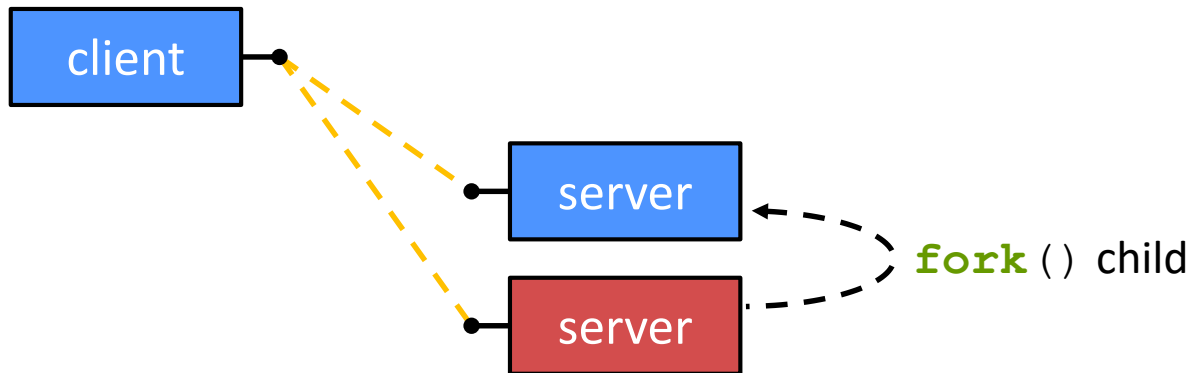
# Double-fork Trick



# Double-fork Trick



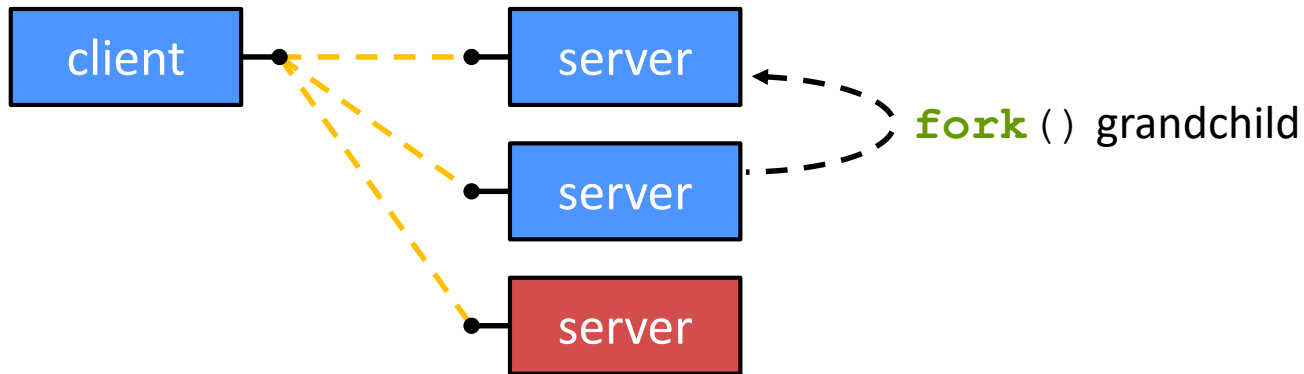
# Double-fork Trick



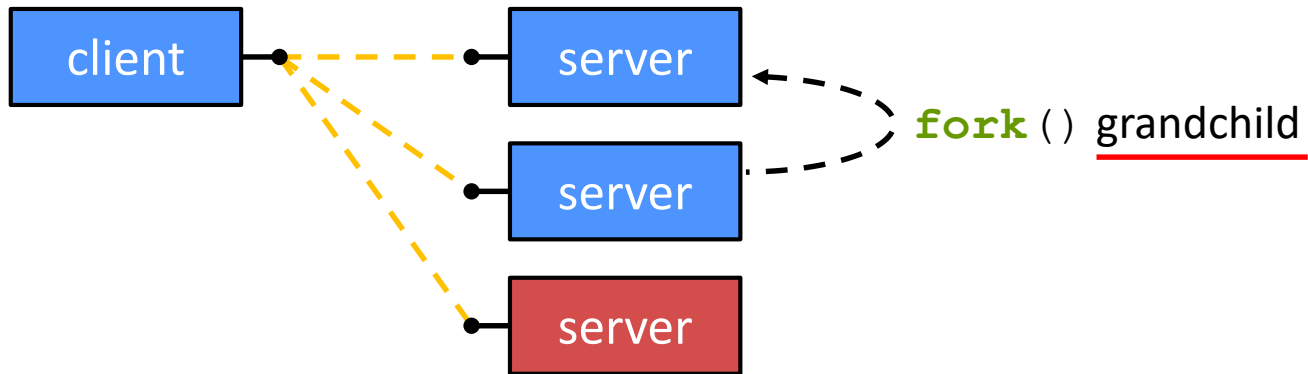
Reminder:

Fork() copies the file descriptor table from parent, so the child has connection to the client too.

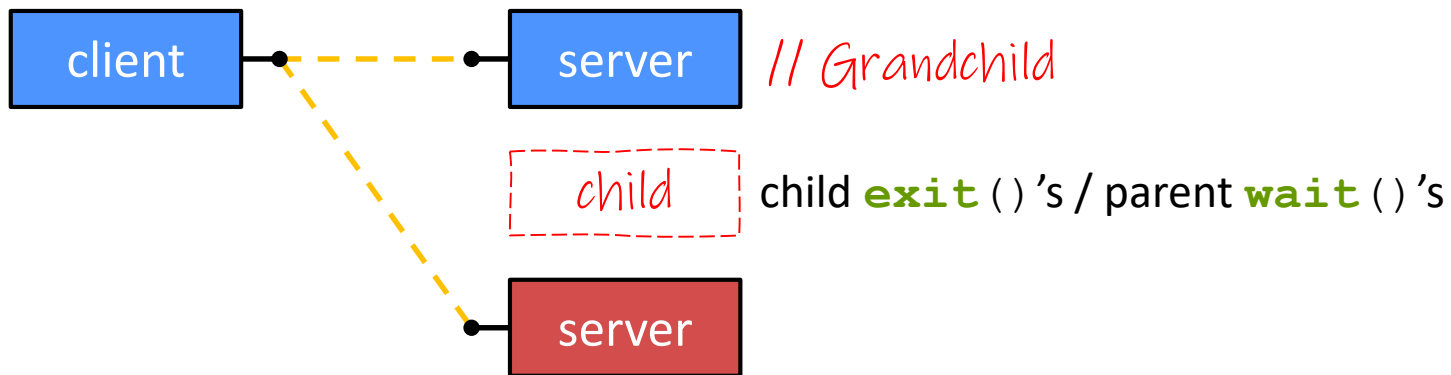
# Double-fork Trick



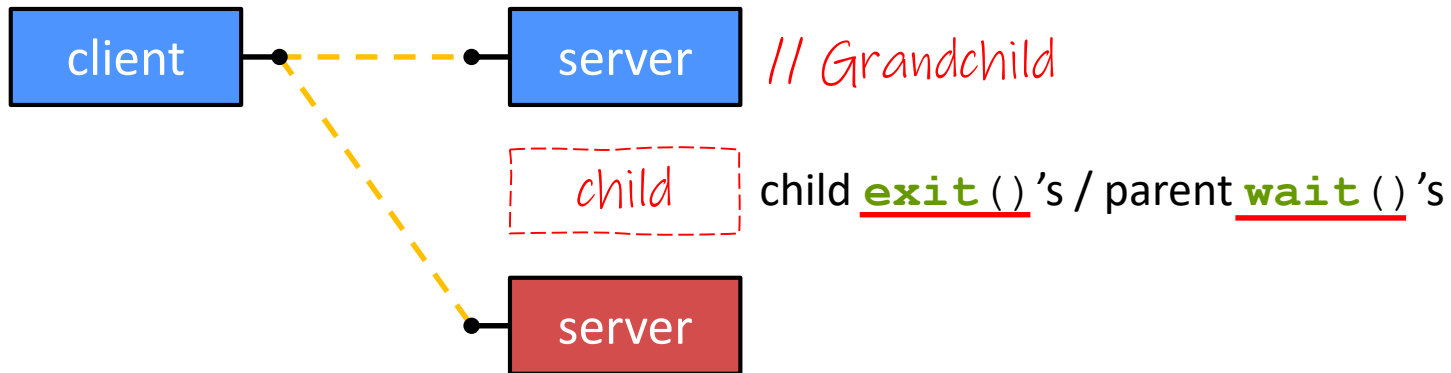
# Double-fork Trick



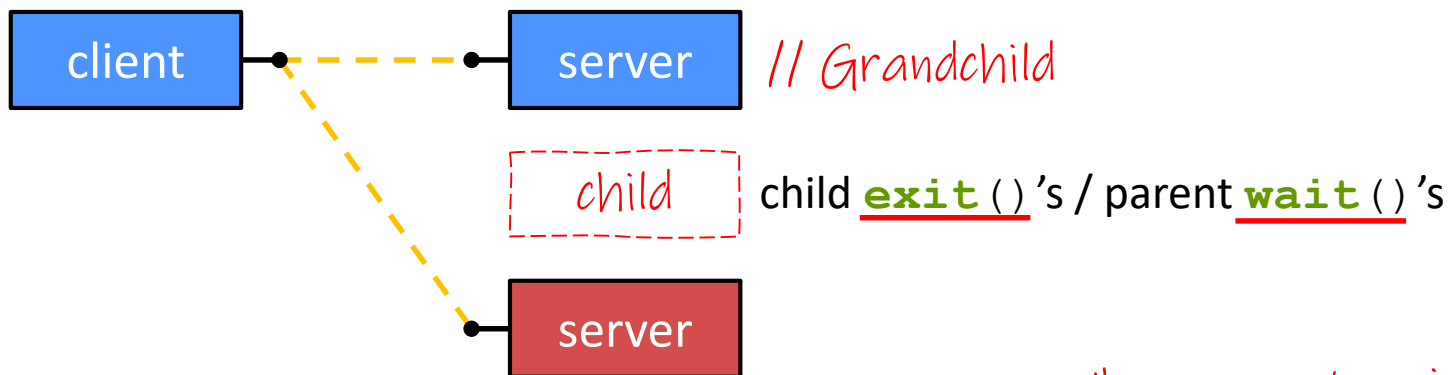
# Double-fork Trick



# Double-fork Trick

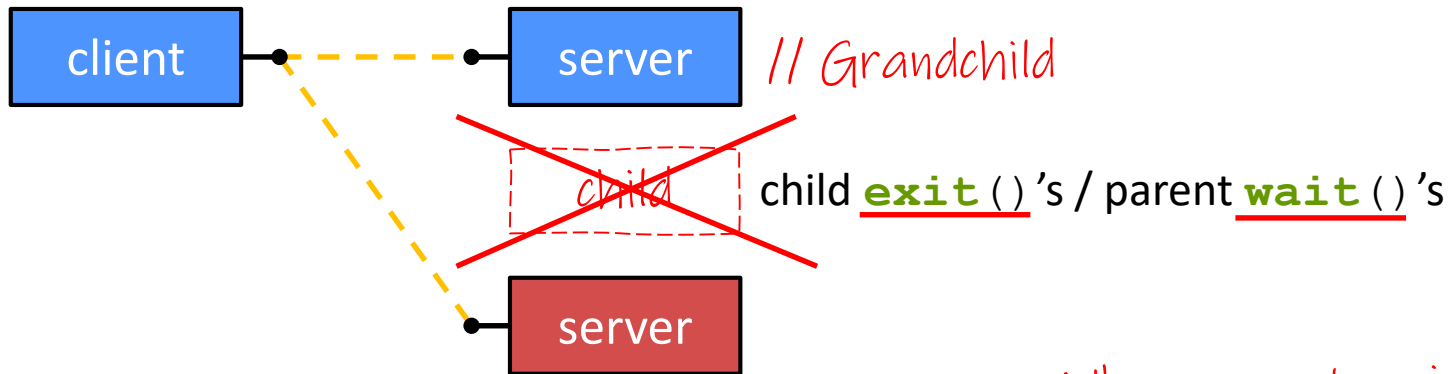


# Double-fork Trick



When parent `wait()`'s  
for child, the child will  
be cleaned up

# Double-fork Trick

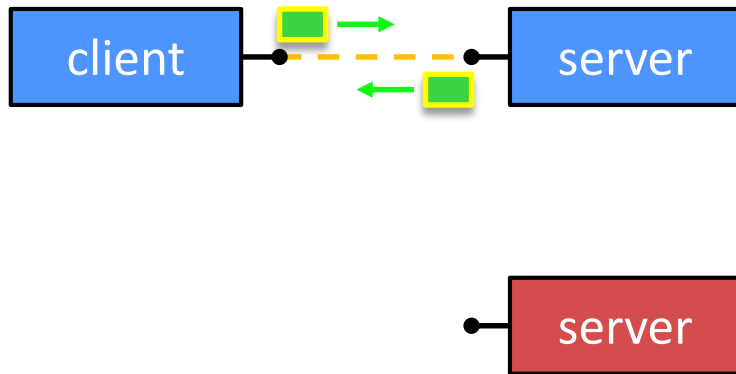


When parent `wait()`'s  
for child, the child will  
be cleaned up

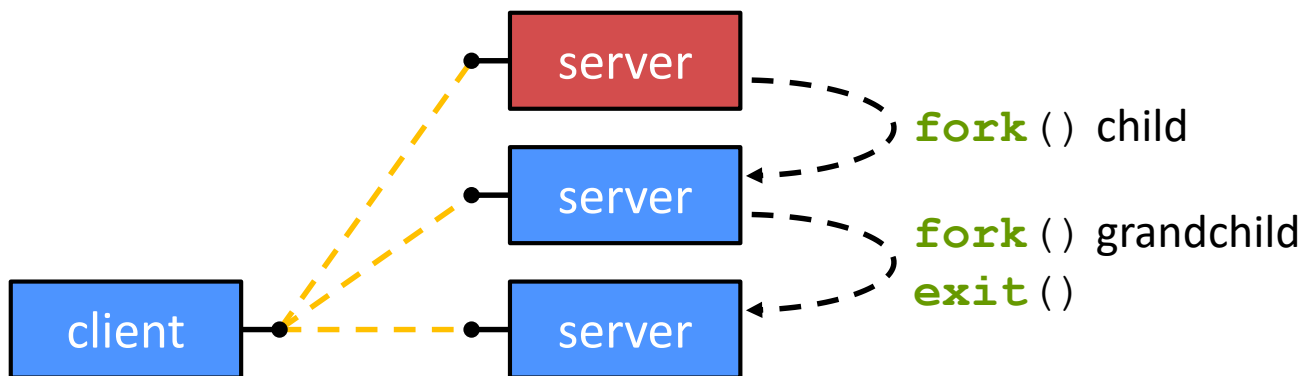
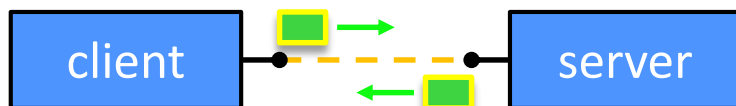
# Double-fork Trick



# Double-fork Trick



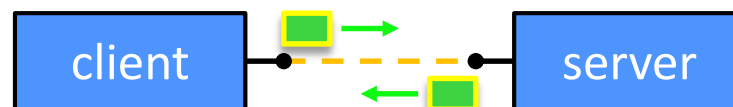
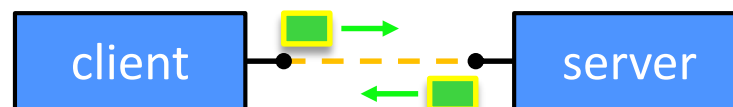
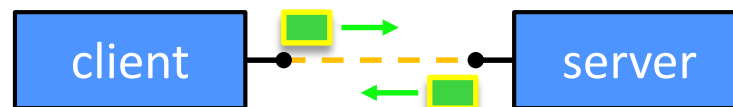
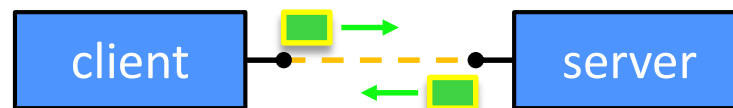
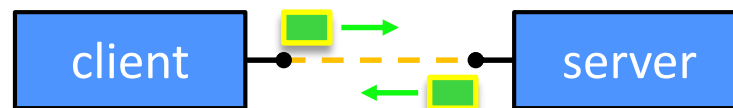
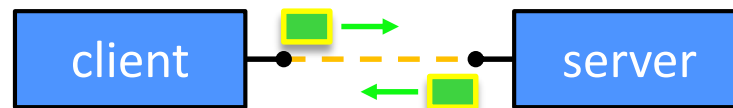
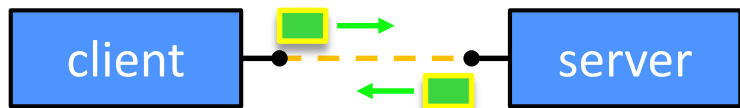
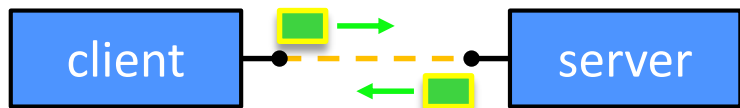
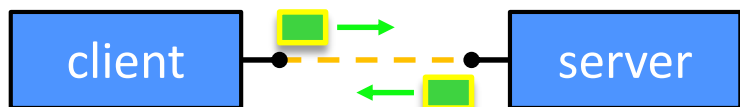
# Double-fork Trick



# Double-fork Trick



# Double-fork Trick



 **Poll Everywhere**[pollev.com/cse333](https://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...**

 **Poll Everywhere**[pollev.com/cse333](https://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...**

# 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);
    }
}
```

# 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);
    }
}
```

Grandchild has copy of socket, we can close our copy

# Lecture Outline

- ❖ Review: Processes vs Threads
- ❖ `fork()` details
  - Concurrent Server with Processes
- ❖ **Conclusion**

# Why Concurrent Processes?

## ❖ Advantages:

- No shared memory between processes
- No need for language support; OS provides “fork”
- Concurrent execution leads to better CPU, network utilization

# Why Concurrent Processes?

## ❖ Advantages:

- No shared memory between processes      *<- Process Isolation*
- No need for language support; OS provides “fork”
- Concurrent execution leads to better CPU, network utilization

# Why Concurrent Processes?

## ❖ Advantages:


- No shared memory between processes      *<- Process Isolation*
- No need for language support; OS provides “fork”
- 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

# Why Concurrent Processes?

## ❖ Advantages:

- ❖  No shared memory between processes *<- Process Isolation*
- No need for language support; OS provides “fork”
- 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

# Why Concurrent Processes?

## ❖ Advantages:

- ❖ No shared memory between processes *<- Process Isolation*
- No need for language support; OS provides “fork”
- 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

# Aside: Thread Pools

- ❖ In real servers, we'd like to avoid overhead needed to create a new thread or process for every request
  - We wrote a Thread Pool implementation for you in HW4

# Aside: Thread Pools

- ❖ In real servers, we'd like to avoid overhead needed to create a new thread or process for every request
  - We wrote a Thread Pool implementation for you in HW4
- ❖ Idea: Thread Pools:
  - 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)