Concurrency: Processes
CSE 333 Spring 2019

Guest Instructor: Andrew Hu

Teaching Assistants:
Andrew Hu          Austin Chen          Brennan Stein
Cheng Ni           Cosmo Wang           Diya Joy
Guramrit Singh     Menqi Chen           Pat Kosakanchit
Rehaan Bhimani     Renshu Gu            Travis McGaha
Zachary Keyes
Administrivia

- Classes have moved online to avoid virus transmission
  - Stay healthy everyone!
- **Final Exam is cancelled**
  - Grade weights will be updated, more info coming soon
- Lectures posted online through Panopto on Canvas
- Section will be online through Zoom
- Office Hours are online through Zoom

- hw4 due Thursday (3/12)
  - Submissions accepted through Sunday, as usual

- Course evaluations! (see Piazza)
Outline

- 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 😞

Review: 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)
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
  - The new process has a separate virtual address space from the parent
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
    - Using Python `subprocess`
**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.
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**
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**
Double-fork Trick
Double-fork Trick

client

connect

server accept()
Double-fork Trick

- client
- server
- server

fork() child
Double-fork Trick
Double-fork Trick

client

server

server

grow child

child exit()’s / parent wait()’s
Double-fork Trick

client → server

parent closes its client connection
Double-fork Trick
Double-fork Trick

```
client -> server
```

```
server

fork() child
fork() grandchild
exit()
```

```
server <-> server
```

```
client
```

```
server
```
Double-fork Trick
Double-fork Trick
Review Question

- 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/`

```c
... // Server set up
while (1) {
    sock_fd = accept();
    pid = fork();
    if (pid == 0) {
        // ??? process
        Child or Parent?
    } else {
        // ??? process
    }
}
```
Concurrent with Processes Pseudocode

- See `searchserver_processes/`

```c
... // 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/`

```c
... // 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/`

```c
... // 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/`

```c
... // 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 {
            // Clean up resources...
            exit();
        }
    } else {
        // Parent process

    }
}
```
Concurrent with Processes Pseudocode

- See `searchserver_processes/`
Why 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
How Fast is `fork()`?

- See `forklatency.cc`

- ~ 0.5 milliseconds per fork*
  - ∴ 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, ...
- Tested on `attu` (3/2/2020)
How Fast is `pthread_create()`?

- See `threadlatency.cc`
  - 0.5 milliseconds per thread creation*  
  - ~10x faster than `fork()`  
  - maximum of (1000/0.05) = 20,000 connections/sec/core  
  - ~2 billion connections/day/core

- Mush 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()  
- Tested on `attu` (3/2/2020)
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)