Concurrency: Processes and Events
CSE 333 Autumn 2019

Instructor: Hannah C. Tang

Teaching Assistants:
Dao Yi           Farrell Fileas       Lukas Joswiak
Nathan Lipiarski Renshu Gu        Travis McGaha
Yibo Cao         Yifan Bai          Yifan Xu
About how long did Exercise 17 take?

A. 0-1 Hours
B. 1-2 Hours
C. 2-3 Hours
D. 3-4 Hours
E. 4+ Hours
F. I prefer not to say
Administrivia

❖ 🎉 No more exercises! 🎉 😊

❖ HW4 due on Thursday (12/05) 🥱
  ▪ You can use at most ONE late day

❖ Guest lecture on Wednesday (12/04)
  ▪ Albert J. Wong, Google: threat modeling and system design
Administrivia

- Final exam on Wednesday (12/11)
  - Final review sessions this weekend!

- Course evals
  - Please fill them out! Your feedback is extremely valuable to us
  - Comments are helpful!
  - Your honesty is even more helpful!
Lecture Outline

❖ Processes
  ▪ fork() and wait()
  ▪ Concurrency using Processes
  ▪ Threads vs. Processes: A Story of Efficiency

❖ Event-based Concurrency

❖ Concurrency Wrapup
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)
Review: Multi-threaded Address Spaces

- After creating a thread
  - *Two* threads of execution running in the address space
    - Original thread (parent) and new thread (child)
    - New stack created for child thread
    - Child thread has its own *values* of the PC and SP
  - Both threads share the other segments (code, heap, globals)
    - They can cooperatively modify shared data
Creating New Processes

- Creates a new process (the “child”) that is an exact clone* of the current process (the “parent”)
  - Variables, file descriptors, open sockets, the virtual address space (code, globals, heap, stack), etc.
  - *Everything is cloned except threads

- 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

- **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() 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

- Remember that processes become “zombies” after death
waitpid() 

- **Block** until the passed-in process has changed state (usually terminated)
  - Detailed process status available in `status` output parameter.

```c
pid_t waitpid(pid_t pid, int *status, int options);
```
I need a `fork()`ing demo!

- See `fork_example.cc`
Lecture Outline

❖ Processes
  ▪ `fork()` and `waitpid()`
  ▪ Concurrency using Processes
  ▪ Threads vs. Processes: A Story of Efficiency

❖ Event-based Concurrency

❖ Concurrency Wrapup
Multi-processes Search Engine: Architecture

- 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 subsequent I/O, calls `exit()`'s when the connection terminates
Double-fork Trick

- There is no “process version” of `pthread_detach()`
  - How do we tell the OS to clean up the process when it’s dead?

- Remember that processes become “zombies” after death
  - **Option A**: Parent calls `waitpid()` to “reap” children
  - **Option B**: Parent terminates, causing children to be “adopted” by the root process (“init” or “systemd”)
Multi-process Search Engine: Request Flow
Multi-process Search Engine: Request Flow

client

connect

server

accept()

In-memory resources
Multi-process Search Engine: Request Flow

client

server

In-memory resources

fork() child

server

In-memory resources
Multi-process Search Engine: Request Flow

client

server

server

In-memory resources

fork() child

In-memory resources
Multi-process Search Engine: Request Flow

client

server

In-memory resources

server

In-memory resources

server

In-memory resources

fork () grandchild
Multi-process Search Engine: Request Flow

- Client connects to server.
- Server provides resources.
- Child process exits / parent process waits.
Multi-process Search Engine: Request Flow

Client

Server

In-memory resources

Parent closes its client connection
Multi-process Search Engine: Request Flow
Multi-process Search Engine: Request Flow
Multi-process Search Engine: Request Flow

- Client
- Server
- In-memory resources
Multi-process Search Engine: Request Flow
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...
Lecture Outline

❖ Processes
  ▪ `fork()` and `waitpid()`
  ▪ Concurrency using Processes
  ▪ Threads vs. Processes: A Story of Efficiency

❖ Event-based Concurrency

❖ Concurrency Wrapup
How Fast is `fork()`?

- See `forklatency.cc`

- ~0.500 ms per fork*
  - maximum of (1000/0.50) = 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, ...
How Fast is `pthread_create()`?

- See `threadlatency.cc`

- \(~0.070\) ms per thread creation*
  - \(~10\)x faster than `fork()`
  - \(\therefore\) maximum of \(\frac{1000}{0.036}\) = 28,000 connections/sec
  - \(~2.4\) 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()`
Lecture Outline

❖ Processes
  - `fork()` and `waitpid()`
  - Concurrency using Processes
  - Threads vs. Processes: A Story of Efficiency

❖ Event-based Concurrency

❖ Concurrency Wrapup
"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.

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

while (1) {
    event = OS.GetNextEvent();
    state = GetState(event);
    ProcessOneTask(state);
}
```
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
Multi-Step Event-Driven Programming

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

```c
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;
            ...
        }
    }

    while (1) {
        event = OS.GetNextEvent();
        task = lookup(event);
        dispatch(task, event);
    }
```
Multi-Step, Event-Driven w/Async I/O
Lecture Outline

❖ Processes
  ▪ `fork()` and `waitpid()`
  ▪ Concurrency using Processes
  ▪ Threads vs. Processes: A Story of Efficiency

❖ Event-based Concurrency

❖ Concurrency Wrapup
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

Pairs naturally with event-based programming (but also works with “traditional” threaded programming)
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
  ▪ Fewer things to synchronize – but when you do need to synchronize, it’s hard!
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