CSE 333
Lecture 18 -- fork, pthread_create

Steve Gribble
Department of Computer Science & Engineering
University of Washington
Previously

We implemented an echo server, but it was sequential

- it processed requests one at a time, in spite of client interactions blocking for arbitrarily long periods of time
  - this led to terrible performance

Servers should be concurrent

- process multiple requests simultaneously
  - issue multiple I/O requests simultaneously
  - overlap the I/O of one request with computation of another
  - utilize multiple CPUs / cores
Today

We’ll go over three versions of the ‘echo’ server

- sequential
- concurrent
  ‣ processes [fork()]
  ‣ threads [pthread_create()]

Next time: non-blocking, event driven version
  ‣ non-blocking I/O [select()]
Sequential

pseudocode:

```c
listen_fd = Listen(port);
while(1) {
    client_fd = accept(listen_fd);
    buf = read(client_fd);
    write(client_fd, buf);
    close(client_fd);
}
```

look at `echo_sequential.cc`
Whither sequential?

Benefits
- super simple to build

Disadvantages
- incredibly poorly performing
  - one slow client causes all others to block
  - poor utilization of network, CPU
Fork is used to create a new process (the “child”) that is an exact clone of the current process (the “parent”)

- everything is cloned (except threads)
  - all variables, file descriptors, open sockets, etc.
  - the heap, the stack, etc.
- primarily used in two patterns
  - servers: fork a child to handle a connection
  - shells: fork a child, which then exec’s a new program
fork() has peculiar semantics

- the parent invokes fork()
- the operating system clones the parent
- **both** the parent and the child return from fork
  - parent receives child’s pid
  - child receives a “0” as pid
fork() has peculiar semantics

- the parent invokes fork()
- the operating system clones the parent
- both the parent and the child return from fork
  ‣ parent receives child’s pid
  ‣ child receives a “0” as pid
fork() has peculiar semantics

- the parent invokes fork()
- the operating system clones the parent
- both the parent and the child return from fork
  - parent receives child’s pid
  - child receives a “0” as pid
fork()
Concurrency 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 the double-fork trick
Graphically

server
Graphically

client

server
Graphically
Graphically
Graphically

client → server
server → server
fork() → grandchild
Graphically

client ➔ server ➔ parent wait()’s

server ➔ child exit()’s ➔ parent wait()’s
Graphically

client -> parent closes its client connection -> server

client <-> server
Graphically
Graphically

- Client
- Server
- Fork child
- Fork grandchild
- Exit
Graphically

client → server

server

client → server
Graphically
Concurrent with processes

look at echo_concurrent_processes.cc
Whither concurrent processes?

Benefits

- almost as simple as sequential
  ‣ in fact, most of the code is identical!
- parallel execution; good CPU, network utilization

Disadvantages

- processes are heavyweight
  ‣ relatively slow to fork
  ‣ context switching latency is high
- communication between processes is complicated
How slow is fork?

run forklatency.cc
Implications?

**0.31 ms** per fork

- maximum of \( \frac{1000}{0.31} = 3,500 \) connections per second per core
- \( \sim 0.5 \) billion connections per day per core
  
  ‣ fine for most servers
  
  ‣ too slow for a few super-high-traffic front-line web services
    
    • Facebook serves \( O(750 \) billion) page views per day
    
    • guess \( \sim 1-20 \) HTTP connections per page
    
    • would need 3,000 -- 60,000 cores just to handle \texttt{fork()}, i.e., without doing any work for each connection!
threads

Threads are like lightweight processes

- like processes, they execute concurrently
  - multiple threads can run simultaneously on multiple cores/CPUs
- unlike processes, threads cohabit the same address space
  - the threads within a process see the same heap and globals
    - threads can communicate with each other through variables
    - but, threads can interfere with each other: need synchronization
  - each thread has its own stack
threads

see thread_example.cc
Concurrency with threads

A single **process** handles all of the connections

- but, a parent **thread** forks (or dispatches) a new thread to handle each connection

- the child thread:
  - handles the new connection
  - exits when the connection terminates
Graphically

accept()
Graphically
Graphically
Graphically

client

pthread_create()

server
Graphically
Graphically

client

client

pthread_create()

server
Graphically

```
server

shared data structures

client

client

client

client

client

client

client
```
Concurrent with threads

look at echo_concurrent_threads.cc
Whither concurrent threads?

Benefits
- straight-line code, line processes or sequential
  ▸ still the case that much of the code is identical!
- parallel execution; good CPU, network utilization
  ▸ lower overhead than processes
- shared-memory communication is possible

Disadvantages
- synchronization is complicated
- shared fate within a process; one rogue thread can hurt you badly
How fast is pthread_create?

run threadlatency.cc
Implications?

0.036 ms per thread create; ~10x faster than process forking

- maximum of \( \frac{1000}{0.036} \) = ~30,000 connections per second
- ~5 billion connections per day per core
  ▸ much better

But, writing safe multithreaded code can be serious voodoo
See you on Wednesday!
Exercise 1

Write a simple “proxy” server

- forks a process for each connection
- reads an HTTP request from the client
  ‣ relays that request to www.cs.washington.edu
- reads the response from www.cs.washington.edu
  ‣ relays the response to the client, closes the connection

Try visiting your proxy using a web browser :)
Exercise 2

Write a client program that:

- loops, doing “requests” in a loop. Each request must:
  - connect to one of the echo servers from the lecture
  - do a network exchange with the server
  - close the connection
- keeps track of the latency (time to do a request) distribution
- keeps track of the throughput (requests / s)
- prints these out