CSE 333
Lecture 22 -- fork, pthread_create, select

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

HW4 out on Monday
- you’re gonna love it

Final exam
- Wednesday, June 8th, 2:30-4:20pm, in this room
- will not be offering it early or late
Last time

We implemented a simple 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 four versions of the ‘echo’ server

- sequential
- concurrent
  - processes [ fork() ]
  - threads [ pthread_create() ]
  - non-blocking [ 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
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
Graphically
Graphically

client

fork() child

server

server
Graphically

client

server

fork() child

server

CSE333 lec 22 network.4 // 05-20-11 // gribble
Graphically

```plaintext
server

fork()

grandchild

server

server

client
```
Graphically

client -> server -> fork() -> grandchild

server

server

server
Graphically

client ➔ server

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

server ➔ server
Graphically

client

server

server

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

- Client
- Server

parent closes its client connection
Graphically
Graphically

client → server

fork( ) child
fork( ) grandchild
exit( )
Graphically
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?

0.18 ms per fork

- maximum of \(\frac{1000}{0.18} = 5,555.5\) connections per second
- 0.5 billion connections per day per machine

  ‣ 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 ~1-20 HTTP connections per page
    • would need 3,000 -- 60,000 machines just to handle fork(), i.e., without doing any work for each connection!
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

client

pthread_create()

server
Graphically
Graphically

```
pthread_create()
```
Graphically
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?

0.021 ms per thread create; 10x faster than process forking
- maximum of \((1000 / 0.021) = \sim 50,000\) connections per second
- 4 billion connections per day per machine
  ‣ much, much better

But, writing safe multithreaded code is serious voodoo
Non-blocking I/O

Warning: an unfamiliar and slightly non-intuitive topic...

Why did the sequential implementation do badly?

- it relied on **blocking** system calls
  - accept() blocked until a new connection arrived
  - read() blocked until new data arrived
  - write() potentially blocked until the write buffer had room
- nothing else could happen while the main thread blocks
Non-blocking I/O

An alternative: **non-blocking** system calls

- non-blocking accept( )
  - if a connection is waiting, accept( ) succeeds and returns it
  - if no connection is waiting, accept( ) fails and returns immediately

- non-blocking read( )
  - if data is waiting, read( ) succeeds and returns it
  - if no data is waiting, read( ) fails and returns immediately

- non-blocking write( )
  - if buffer space is available, write( ) deposits data and returns
  - if no buffer space is available, write( ) fails and returns immediately
A (bad) first attempt [N clients]

```c
state  s[N];         // clients’ state field
int    fd[N], readfd[N]; // clients’ file descriptors
char *data[N], *fdata[N]; // buffers holding clients’ data

while (1) {
    for (int i = 0; i < N; i++) {
        if (s[i] == NET_READING) {
            if (nb_read(fd[i], data[i]))
                s[i] = FILE_READING;
        }

        if (s[i] == FILE_READING) {
            if (nb_read(getfd(data[i]), fdata[i]))
                s[i] = NET_WRITING;
        }

        if (s[i] == NET_WRITING) {
            if (nb_write(fd[i], fdata[i])
                s[i] = NET_READING;
        }
    }
}
```
Compare with threaded

```c
pthread_create(t1, handleclient, fd1);
pthread_create(t2, handleclient, fd2);

handleclient(int fd) {
    while (1) {
        data = geturldata(fd);
        do_netwrite(fd, filedta);  // NET_WRITING
    }
}

char *geturldata(int fd) {
    filename = read(fd);  // NET_READING
    return readdir(filename);  // FILE_READING
}

void do_write(int fd, char *data) {
    write(fd, data);
}

char *readfile(char *filename) {
    return do_read(fopen(filename));
}
```
Pictorially

<table>
<thead>
<tr>
<th></th>
<th>buffer</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>fd1</td>
<td>NET_WRITING</td>
<td>buffer</td>
<td></td>
</tr>
<tr>
<td>fd2</td>
<td>FILE_READING</td>
<td>buffer</td>
<td></td>
</tr>
<tr>
<td>fd3</td>
<td>NET_READING</td>
<td>buffer</td>
<td></td>
</tr>
</tbody>
</table>

main

NON BLOCKING

THREADED

data = fileread()

buffer = nb_netread()
nb_netwrite(data)

netread

fileread

geturldata:
    netread()
    fileread()

write()

geturldata():
do_write();

while (1) {
    accept();
    thread_create(start);
}

main
Task state
- kept in a table in the heap

Task concurrency, threads
- single thread dispatches “I/O is available” event
- program *is* task scheduler

Call graph
- only one “procedure” deep
- code path is sliced at what used to be blocking I/O

main

<table>
<thead>
<tr>
<th>fd1</th>
<th>NET_WRITING</th>
<th>buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>fd2</td>
<td>FILE_READING</td>
<td>buffer</td>
</tr>
<tr>
<td>fd3</td>
<td>NET_READING</td>
<td>buffer</td>
</tr>
</tbody>
</table>

data = fileread()

buffer = nb_netread()

nb_netwrite(data)
**THREADED**

**Task state**
- kept in each thread’s stack

**Task concurrency, threads**
- each thread spurts computation between long blocking IOs
- OS is the scheduler

**Call graph**
- many procedures deep; stack trace lines up with task progress

```c
while (1) {
    accept();
    thread_create(start);
}

geturldata:
    netread()
    fileread()

write()

netread

fileread

geturldata();
do_write();

while (1) {
    accept();
    thread_create(start);
}

main
```
Problem with first attempt

It burns up the CPU, constantly looping

- testing each connection to see if it received an event
  - if so, dispatch the event
- which events?
  - fd is read’able
  - fd is write’able
  - fd is accept’able
  - fd closed / in an error state

```c
while (1) {
    for (int i = 0; i < N; i++) {
        if (s[i] == NET_READING) {
            if (nb_read(fd[i], data[i]))
                s[i] = FILE_READING;
        }
        if (s[i] == FILE_READING) {
            if (nb_read( ... )
                s[i] = NET_WRITING;
        }
        if (s[i] == NET_WRITING) {
            if (nb_write( ... )
                s[i] = NET_READING;
        }
    }
}
```
An idea

Instead of constantly polling each file descriptor, why not have one blocking call?

- “hey OS, please tell me when the next event arrives”

```c
while (1) {
    (fd, event) = wait_for_next_event( fd_array );

    switch (event) {
        case NET_WRITEABLE:
            do_netwrite(fd, lookup_state(fd));
            break;
        case NET_READABLE:
            do_netread(fd, lookup_state(fd));
            break;
        case FILE_READABLE:
            do_fileread(fd, lookup_state(fd));
            break;
        case NET_CLOSED:
            close(fd);
            break;
    }
}
```
select( )

```c
int select(int nfds,
           fd_set *read_fds,
           fd_set *write_fds,
           fd_set *error_fds,
           struct timeval *timeout);
```

Waits (up to timeout) for one or more of the following:

- readable events on (read_fds)
- writeable events on (write_fds)
- error events on (error_fds)
See you on Monday!
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