Administrivia

HW4 is due on Thursday

- <panic> if you haven’t started yet </panic>
Administrivia (2)

HW4 due Thursday night, 11 pm
- Don’t run past available late days - no credit if you do

No sections Thursday - TAs in the labs 8:30+ε – 11:00+ε

Final exam next Wednesday, here, 8:30 am
- Review Q&A Tuesday, 4:30, EE 045
Some common HW4 bugs

Your server works, but is really really slow
- check the 2nd argument to the QueryProcessor constructor

Funny things happen after the first request
- make sure you’re not destroying the HTTPConnection object too early (e.g., falling out of scope in a while loop)

Server crashes on blank request
- make sure you handle the case that read() [or WrappedRead] returns 0
Previously

We implemented hw3 searchserver, 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 searchserver

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

If we have 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);
    resp = ProcessQuery(buf);
    write(client_fd, resp);
    close(client_fd);
}
```

look at `searchserver_sequential/`
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( )

\[ \text{pid}_t \ \text{fork}(\text{void}); \]

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() and address spaces

Remember this picture...?
- a process executes within an **address space**
- the address space includes:
  - a stack (for stack frames)
  - heap (for dynamically allocated data)
  - text segment (containing code)
  - etc.
fork() and address spaces

Fork causes the OS to clone the address space, creating a brand new process

- the new process starts life as a copy the old process in (nearly) every way
- the copies of the heap, stack, text segment, etc. are (nearly) identical
- the new process has copies of the parent’s data structures, stack-allocated variables, open file descriptors, and so on
fork( )

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( )

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( )

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( )

fork_example.cc
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

client

connect

server
Graphically

client

server

server

fork() child
Graphically
Graphically

client -- server

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

client ─────────── server

server parent closes its client connection

server
Graphically

client ➔ server ➔ server
Graphically
Graphically
Graphically

client → server → client → server

client → server → client → server

client → server → client → server

client → server → client → server

client → server → client → server

client → server → client → server
Concurrent with processes

*look at* `searchserver_processes`
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?

0.25 ms per fork

- maximum of (1000 / 0.25) = 4,000 connections per second per core
- ~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 $O(750 \text{ billion})$ page views per day
    • would need 3,000 -- 6,000 cores just to handle fork(), i.e., without doing any work for each connection!
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 and the address space

Pre-thread create

- one thread of execution running in the address space
  - the “main” thread
  - therefore, one stack, SP, PC

- that main thread invokes a function to create a new thread
  - typically “pthread_create( )”
threads and the address space

Post- thread create

- two threads of execution running in the address space
  - the “main” thread (parent)
  - the child thread
  - thus, two stacks, SPs, PCs
- both threads share the heap and text segment (globals)
  - they can cooperatively modify shared data
threads

see thread_example.cc
Concurrent server 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

server

accept()
Graphically

client → connect 
  
  accept() 

server
Graphically

client

server

accept( )
Graphically

client

pthread_create()

server
Graphically

client

client

server

pthread_create( )
Graphically

client

client

client

client

client

shared data structures

server
Concurrent with threads

look at searchserver_threads/
Whither concurrent threads?

Benefits

- straight-line code
  ‣ still the case that much of the code is identical to sequential!
- 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 (1000 / 0.018) = ~60,000 connections per second
- ~10 billion connections per day per core
  ‣ much better

But, writing safe multithreaded code can be serious voodoo
Threads and races

What happens if two threads try to mutate the same data structure?

- they might interfere in painful, non-obvious ways, depending on the specifics of the data structure
  - imagine if two threads try to push an item onto the head of the linked list at the same time
  - depending on how the threads interleave, you might end up with a correct answer, or you might break the data structure altogether
Simple “race” example

If no milk, buy some more
- liveness: if out, somebody buys
- safety: at most one person buys

What happens with multiple threads?

```c
if (!milk) {
    buy milk
}
```
Simple “race” example

Does this fix the problem?

if (!note) {
    if (!milk) {
        leave note
        buy milk
        remove note
    }
}
Synchronization

Synchronization is the act of preventing two (or more) concurrently running threads from interfering with each other when operating on shared data

- need some mechanism to coordinate the threads
  - “let me go first, then you go”
- many different coordination mechanisms have been invented
  - take cse451 for details
Locks

lock acquire
- wait until the lock is free, then take it

lock release
- release the lock
- if other threads are waiting for it
  ‣ wake up exactly one of them
  ‣ give it the lock

simplifies concurrent code
- prevents more than one thread from entering a critical section

... non-critical code ...

lock.acquire();
critical section
lock.release();

... non-critical code ...
Simple “race” solution

What is the critical section?
- checking for milk
- buying more milk if out

These two steps must be uninterrupted, i.e., \textit{atomic}
- solution: protect the critical section with a lock

```c
milk_lock.lock();
if (!milk) {
    buy milk
}
milk_lock.unlock();
```
pthreads and locks

pthread_mutex_init( )
- creates a mutex (a.k.a. a lock)

pthread_mutex_lock( )
- grabs the lock

pthread_mutex_unlock( )
- releases the lock

see lock_example.cc
See you on Friday!
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