CS61 Scribe Notes 12/2

Administrative Things

- final on 12/18 in SC
- review session near the end of reading week
- doodle poll for code review
- all work for the class, except for final, is due by the end of reading week - will not accept anything after reading week is over - Midnight next Wednesday

Problem Set 6 Handout Code

Server and Client (pong61) communicate with HTTP Requests
Use strace to trace system calls - strace -o strace.txt ./pong61

- write(3, “POST /test/reset HTTP/1.0\r\nHost:”...)
    - method: POST
    - locator: /test/rest
    - protocol ID: HTTP/1.0
- read(3, “HTTP/1.1 200 OK…” ...)
    - response:
        - version: HTTP/1.1 (which HTTP version the server supports)
        - status code: 200
            - text description of status: OK

In linux, clone() does both fork() and new_threat()

- check the flags: CLONE_THREAD - create new thread, not new process
    - CLONE_VM - share memory state
    - CLONE_FS - share FDs
- a new thread is very like a new process, it just shares more

In handout code:

- each move of the ball is handled by a new thread
- DIAGRAM - each sublevel represents a child thread of the parent level)
- main
- mutex_init
- cond_init
- pong_args
- pthread_create(&pt) - new thread created
        - copy arg (this is part of a possible race condition with the destruction of pong_args)
        - connects to server
        - sends request
        - receives response
        - closes connection

- signal the **main thread** to continue using the condvar - this is only done by THIS thread
- exit

- lock
- cond_wait - this is used to resolve the race condition discussed below; this code is only run AFTER the child thread has signaled the **main thread** using condvar
- unlock
- usleep
- than loop (go back to pong_args)
- after look, destroy pong_args - we need to make sure this happend BEFORE copy_args in the new thread!

(RESOLVED) Possible race conditions - pong_args are a local variable and are initialized in a block (the loop)
pong_args are destroyed when the block ends

In handout code, now we go to phase 2:

- Here, the server delays the full response after the reset request by the client
    - First part of the response is sent, and only much later does the server say “DONE”
    - Partial responses to clients are inevitable
        - They are sent over TCP/IP protocol. Responses are divided into multiple packets which aren’t al sent at the same time
- Where does the delay happen in the dependency diagram?
    - It happens in the receive_response, because the child thread only signals **after** it has received a response from the server
        - Pong thread is waiting on server, main thread is waiting on signal from the pong thread, therefore main thread is waiting on the server…
    - How to fix?
        - Could we just move the signal below copy_args? This is what we want to lock anyway.
        - **MAKE SURE** not to put it before copy_args in child thread
        - Did this work? No! However, we made good progress with a simple movement of code - we need to make
- New problem - ordering of pinged balls: there is nothing prevent the server from responding out of order because we don’t wait until connect to continue the main thread which might launch new threads.
    - Solution - put the signal after the connect
- New problem - we have too many concurrent connections (server caps at 30)
    - How to fix? (need a count of threds)
        - Keep track of the number of threads that we are running!
        - In the main thread, nthreads = 0 (nthreads is a global variable)
            - while (nthreads >= 30) do nothing

- In child threads, decrement thread count right before you exit

- New problem - we have added race conditions in two new locations to the code (when we increment and decrement the nthreads)
  - What is the critical section in the code?
    - The updates to nthreads must be done only by one thread
  - use pthread function calls to deal with critical sections in the code
    - pthread_mutex_lock & pthread_mutex_unlock around increments and decrements of nthreads
    - later in pset we will need to do more data structure maintenance, and when updating a global storing threads or connections, we also want to lock
- Concurrency Networking Synthesis
- Eight Fallacies of Distributed Computing - Peter Deutsch
  - read them online. All assumptions that are untrue
  - 1) Network is reliable
    - When you need to send a large amount of information, it is broken down into smaller pieces that fit into packets.
    - These packets are then sent over the network and reassembled into the original piece of information
    - BUT
      - network is allowed to:
        - drop pieces
        - reorder pieces
        - duplicate pieces
        - delay pieces
      - Why?
        - over time the network will change.
        - Imagine a scenario
          - You are asking for info from google
          - google sends packets 1, 2, 3, 4, 5
          - speed boat cuts cable
          - only packets 1, 2 make it to you
          - google doesn’t know what info was sent to you, will have to guess at what they need to resend.
          - This can cause all kinds of problems with lost packets, reordered packets, duplicate packets
          - If we assume that the endpoint is smart enough to handle these cases, the network can be much easier to manage
          - Called End-to-End principle
    - 4 Layers
      - Application Layer
        - HTTP

- Transport Layer
  - TCP
  - TCP takes dropped pieces, reordered pieces, duplicate pieces, and delayed pieces and reassembles them
- Network layer
  - IP
- Physical Layer
  - Ethernet protocol

- Problem Set 6 shows why the network is not reliable/secure
- l24 directory from lectures repository
  - shows sort algorithms
  - sort01 uses qsort (quicksort) call
    - in linereader.h
      - line = char* s and size_t length
      - lineset = array of lines with pointers to the front and end
  - with 48 cores, how can we optimize this?
    - idea #1 - split lineset into 48 pieces and then perform a standard merge
      - 48 pieces into 1 piece directly
    - idea #2 - split lineset into 32 pieces and merge 2 by 2
      - 32 pieces -> 16 pieces -> 8 pieces -> 4 pieces -> 2 pieces -> 1 piece
  - sort02 does this
    - runs slightly faster than quicksort, but spends a lot of time in malloc
    - quicksort is a sort in place algorithm, but mergesort needs to malloc new space to merge into
  - sort08 is most optimized
    - runs about 3 times faster
- Next step if you liked CS61
  - CS146 - Architectures
  - Programming languages
  - Operating systems
  - Compilers
  - Seminar CS260r - taught by Eddy
    - Detecting nasal demons (better ways to debug!) - run code backwards