Networking
Multiple Clients

Adapted by Joseph Lunderville
from slides by Dr. Brian Fraser
and course material by Dr. Steve Ko

Topics

- How can one program handle (very) many requests?
  - Specifically a server handle many TCP clients?

TCP Server Recap

- Recall that on a TCP server
  - We open the first socket and call `accept()`
  - `accept()` will return a new socket file descriptor for the new client connection
- How can we make our server work with multiple client sockets?

Idea 1: Thread per Connection

- Idea 1
  - Server creates a new thread (or child process) for each accepted connection
  - This thread handles the new client's socket
- Pros
  - Handle multiple clients cleanly (i.e. without much modification)
    - Very obvious code organization
- Cons
  - Higher overhead of creating new processes or threads
    - Each thread has its own stack, i.e. VM space reserved to accommodate the worst case
    - Also a bunch of process info in the kernel

Idea 2: Non-Blocking Sockets

- *Idea 2*
  - Non-blocking calls can allow writing concurrent code without threads…
  - Create array of open sockets and poll with non-blocking calls
  - Server loops indefinitely, using
    - Non-blocking `accept()` to add any new socket to array
    - Non-blocking read/write on each socket in array as needed
- Pros:
  - Avoids creating new processes/threads
  - Idle sockets use *much* less memory!
- Cons:
  - **Busy-wait loop checking sockets wastes CPU**

Idea 3: Kernel Notify on Socket Event

- *Idea 3*
  - Kernel notifies program on socket event
  - Use non-blocking sockets and kernel notifies program on socket events
- ***I/O Multiplexing***
  - Use syscalls to monitor multiple file descriptors
  - Program is notified when a monitored file descriptor is ready for read or write (or on error)
  - Use: `select()`, `poll()`, and `epoll()`

Idea 3: (cont'd)

- *Generally speaking, this is how I/O multiplexing works*
  - Add file descriptors to the monitored list
  - Indicate what events we want to monitor the file descriptors for (read, write, …)
  - Call the blocking function to wait for an event (`select()` or `epoll()`)
  - When it returns, check which file descriptors can perform I/O
  - We perform the I/O
- Pros
  - Like (2), but no thread overhead, little-to-no polling
- Cons:
  - More complex to maintain list of file descriptors to monitor!

Idea 3: Implementing Sketch with epoll

- *3 Calls to implement I/O Multiplexing with `epoll*()`*
  - `epoll_create()`
    - Returns an epoll instance
    - We can think of this as a monitor object that maintains the monitoring list
  - `epoll_ctl()`
    - Allows us to add, remove, or modify a file descriptor to the epoll instance
    - Start by monitoring socket for `accept()`
    - Each new FD from `accept()` is added to set to monitor
  - `epoll_wait()`
    - Waits for a file descriptor to be available for I/O

Async I/O vs. Threads

- *Software engineering perspective*
  - It's great to worry about how efficient a program is, but you have to *write it* first
- *Thread pros*
  - Threads give a more obvious code organization
  - Debugging, breakpoints, exceptions, etc. work normally
  - You get parallelism when doing work by default
  - If a thread blocks waiting on a resource, the dependency is visible to the scheduler
- *Async (polling or epoll) pros*
  - You get atomicity between polls automatically
    - Data races are much less likely to occur accidentally!
- In a language with good async support, exceptions and program flow might work well with polling/epoll

Audience Participation - Server Choices

- Match the server implementation idea with the problem it suffers:
  - Non-blocking IO in a loop
  - epoll() to watch sockets
  - Thread per client

a) More complex code

b) Only handle one socket at a time.

c) More likely to use too much system resources (such as RAM), or too high kernel overhead.

d) Wastes CPU Time

Summary

- *`accept()` returns a new socket for each TCP client*
- *Server must likely handle many sockets at once*
  - Can create a new thread per socket
  - Can use non-blocking IO to busy-wait checking for ready sockets
  - Can use `epoll()` or `select()` to have kernel monitor sockets