Project 1

15-441: Computer Networks

Matt Mukerjee
David Naylor
Ben Wasserman

Agenda

- Handling Concurrency
- Project 1 Checkpoint 1
- Q & A

Flashback!

- getaddrinfo() - Prepare to launch!
- socket() - Get the file descriptor!
- bind() - Which port am I on?
- listen() - Will someone please call me?
- connect() - Hey, you!
- accept() - Thank you for calling port 8080!
- send() and recv() - Talk to me, please!
- close and shutdown() - Get out!

What do you want to build?
A webserver that can handle multiple concurrent connections!

What's the problem?

Blocking!

What's the solution?
Threading or select()

Threading approach

- Did in 15-213??
- Main server blocks on accept()
- Accept incoming connection
- Fork() child process for each connection
- Pain!
  - Need to manage a pool of threads
  - And what if tasks have to communicate?

World of select()

- Event driven programming!
- Single process that multiplexes all requests.
- Caveat
  - Programming is not so transparent!
  - Server no longer acts like it has only one client!

How to use select()?

- Give select a set of sockets/file descriptors.
- select() blocks till something happens.
  - Data coming in on some socket.
  - Able to write to a socket.
  - Exception at the socket.
- Once woken up, check for the event and service it the way the server would do.

select()

```
#include <sys/select.h>


int select (int  nfds, fd_set*  readfds,
             fd_set*  writefds, fd_set*  exceptfds,
            struct timeval *timeout);
```

fd_set Datastructure

- Remember, file descriptor is just an integer!
- Datastructure is basically a bit array!
- Helper macros:

```
FD_ZERO(fd_set* fdset);  /* initializes fdset to have 0s for all fds */
FD_SET(int fd, fd_set* fdset);  /* sets the bit for fd in fdset */
FD_CLR(int fd, fd_set* fdset);  /* clears the bit for fd in fdset */
FD_ISSET(int fd, fd_set* fdset); /* returns 0 if fd is set else non-0 */
```

select() Parameters

- The FDs between 0 to nfds-1 are checked.
- Check for reading in readfds.
- Check for writing in writefds.
- Check for exception in exceptfds.
- These fd_sets can be NULL.
- timeout
  - NULL – blocking
  - else how long to wait for the required condition before returning to the caller.

Return value, Error states

- Success – number of ready descriptors.
  - readfds, writefds and exceptfds are modified
- Time expired – returns 0 (errno set to EINTR)
- Failure – returns -1
  - EBADF, EINTR, EINVAL , ENOMEM

Pseudo-code of Usage

- nfds = 0
- Initialize readfds, writefds, exceptfds using FD_ZERO
- Add the listener socket to readfds using FD_SET and update nfds
- For each active connection
  - If connection has available read buffer, add fd to readfds (FD_SET)
  - If connection has available write buffer, add to writefds (FD_SET)
  - Add to exceptfds (FD_SET) – not really needed for this project.
  - Update nfds to ensure that the fd falls in the range
- select_return = select(nfds, readfds, writefds, exceptfds, NULL)
- If select_return > 0
  - Handle exceptions if any fd in exceptfds is set to 1 (FD_ISSET)
  - Read data from connections for which fd in readfds is set to 1 (FD_ISSET)
  - Write data from connections for which fd in writefds is set to 1 (FD_ISSET)
  - If listener socket is set to read, accept and handle new connection.
- Else handle error states

cp1_checker.py

- ./cp1_checker.py <ip> <port> <#trials> <#writes and reads per trial> <max # bytes to write at a time> <#connections>
  - Starts #connections connections to server at ip and port
  - Repeat #trials number of times
    - Sample #writes and reads per trials connections.
    - Send random number of random bytes to each of these connections (with a limit of max # bytes to write at a time).
    - Receive and check if all the bytes received are same as the ones that are sent.
  - If your server cannot handle multiple connections
    - Set #connections to 1 and #writes and reads per trial to 1

Okay, so you can handle multiple connections!
But that is not enough...

Reading data

- Check return value of recv()
  - Error – handle the error and clear up state.
  - If peer shutdown the connection, clear up state.
- Maintain state
  - Maintain a read buffer
  - Keep track of the number of bytes left to be read
  - May need multiple reads to get all data
  - But only one read per socket when select() returns.

Writing data

- Check return value of send()
  - Error – handle the error and clear up state.
  - If peer shutdown the connection, clear up state.
- Maintain state
  - Maintain a write buffer
  - Keep track of the number of bytes left to be written
  - May need multiple writes to send all data
  - Number of bytes actually sent should be checked from the return value
  - Only one write per socket when select() returns.

Exceptfds

- For handling out of band data
- Should be read one byte at a time!
- Not really needed for this project.

Checkpoint 1 Docs

- Makefile - make sure nothing is hard coded specific to your user; should build a file which runs the echo server (name it lisod)
- All of your source code - all .c and .h files
- readme.txt - file containing a brief description of your current implementation of server
- tests.txt - file containing a brief description of your testing methods for server
- replay.test - a file containing bytes that can be sent to your server as a test case
- replay.out - a file containing expected bytes that should be sent as a response from your server when provided replay.test
- vulnerabilities.txt - identify at least one vulnerability in your current implementation

Remember

- Code quality
- Code documentation
- Robustness
  - Handle all errors
  - Buffer overflows
  - Connection reset by peer

Peek into the future

- Checkpoint 2
  - Implement HTTP 1.1 parser and persistent connections
- Checkpoint 3
  - Implement HTTPS handshaking and persistent connections via TLS
  - Implement CGI server-side.

Any other questions?

Come to our office hours!
Matt Gates 7501 Wednesday 11-Noon
David Gates xxxx Tuesday 4:30 – 5:30 PM
Ben Gates xxxx Thursday 3 – 4 PM