CIS 330: UNIX and C/C++
HTTP, Wrapup

Prof. Kevin Butler
Winter 2012
Administivia

• Last problem set exercises: due Friday
• Assignment 4: due Saturday (can use grace days)

• Final exam
  ‣ Thursday March 22, 8:00am, in this room
  ‣ closed book
  ‣ C, C++ topics
HTTP is a “protocol”

- **Protocol**: the rules governing the exchange of messages, and the format of those messages, in a computing system
  - what messages can a client exchange with a server?
    - what do the messages mean?
    - what are legal replies to a message?
    - what is the syntax of a message?
  - what sequence of messages is legal?
    - how are errors conveyed?
- A protocol is (roughly) the network equivalent of an API
- Hypertext transport protocol
  - a request / response protocol
  - a client (web browser) sends a request to a web server
    - the server processes the request, sends a response
  - typically, a request asks the server to retrieve a resource
    - a resource is an object or document, named by a URI
  - a response indicates whether the server succeeded
    - and, if so, it provides the content of the requested response
How fast is `pthread_create`?

• run `threadlatency.cc`
An HTTP request

- [METHOD] [request-uri] HTTP/[version]\r\n
- [fieldname1]: [fieldvalue1]\r\n
- [fieldname2]: [fieldvalue2]\r\n
- [...]\n
- [fieldnameN]: [fieldvalueN]\r\n
- \r\n
- [request body, if any]
HTTP methods

• There are three commonly used HTTP methods
  ‣ **GET**: “please send me the named document”
  ‣ **POST**: “I’d like to submit data to you, such as form content”
  ‣ **HEAD**: “send me the headers for the named object, but not the object. (I’d like to see if my cached copy is still valid.)”

• There are several rarely used methods:
  ‣ **PUT, DELETE, TRACE, OPTIONS, CONNECT, PATCH, ...**
    • **TRACE**: “if there are any proxies or caches in between me and the server, please speak up!”
HTTP versions

- Most browsers and servers speak HTTP/1.1
  - “version 1.1 of the HTTP protocol”
    - http://www.w3.org/Protocols/rfc2616/rfc2616.html
  - introduced around 1996 to fix shortcomings of HTTP/1.0
    - better performance, richer caching features, better support for multi-homed servers, and much more
    - more complicated to implement than HTTP/1.0
Client headers

• The client can provide zero or more request “headers”
  ‣ they provide information to the server, or modify how the server should process the request

• You’ll encounter many in practice
  ‣ Host: the DNS name of the server  [why?]
  ‣ User-Agent: an identifying string naming the browser  [why?]
  ‣ Accept: the content types the client prefers or can accept
  ‣ Cookie: an HTTP cookie previously set by the server
Example...

GET /foo/bar.html HTTP/1.1
Host: futureproof.cs.washington.edu:5555
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_7; en-us) AppleWebKit/533.21.1 (KHTML, like Gecko) Version/5.0.5 Safari/533.21.1
Accept: application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Cookie: __utmz=59807807.1547453334.1214335349.1301330421.1301339949.30; __utmz=59807807.1300728257.27.14.utmcsr=google|utmccn=(organic)|utmcmd=organic|utmctr=csgordon@u.washington.edu; __utmz=80390417.1521666831.1201286098.1302710464.1302717901.34; __utmz=80390417.1301950604.31.15.utmcsr=cs.washington.edu|utmccn=(referral)|utmcmd=referral|utmct=/education/courses/cse333/11sp/; __qca=P0-1872143622-1294952393928
Connection: keep-alive
An HTTP response

- **HTTP/** [version] [status code] [reason]\r\n- [fieldname1]: [fieldvalue1]\r\n- [fieldname2]: [fieldvalue2]\r\n- [...]\n- [fieldnameN]: [fieldvalueN]\r\n- \r\n- [response body, if any]

*let’s use “telnet” to see a real response*
Status codes, reason phrase

• Code: a computer-readable outcome of the request
  ‣ three digit integer; first digit identifies the response category
    • 1xx: some kind of informational message
    • 2xx: success of some kind
    • 3xx: redirects the client to a different URL
    • 4xx: the client’s request contained some error
    • 5xx: the server experienced an error

• Reason phrase: human-readable explanation
  ‣ e.g., “OK” or “Moved Temporarily”
Common status lines

• **HTTP/1.1 200 OK**
  ‣ the request succeeded, the requested object is sent

• **HTTP/1.1 404 Not Found**
  ‣ the requested object was not found

• **HTTP/1.1 301 Moved Permanently**
  ‣ the object exists, but its name has changed
  ‣ the new URL is given in the “Location:” header

• **HTTP/1.1 500 Server Error**
  ‣ the server had some kind of unexpected error
Server headers

• The server can provide zero or more request “headers”
  ‣ they provide information to the client, or modify how the client should process the response

• You’ll encounter many in practice
  ‣ **Server**: a string identifying the server software
  ‣ **Content-Type**: the type of the requested object
  ‣ **Content-Length**: size of requested object [why?]
  ‣ **Last-Modified**: a date indicating the last time the request object was modified [why?]
Example

HTTP/1.1 200 OK
Date: Fri, 27 May 2011 17:05:53 GMT
Server: Apache/2.2.19 (Fedora)
Last-Modified: Fri, 27 May 2011 17:04:51 GMT
ETag: "2740640-52-4a444ef9392c0"
Accept-Ranges: bytes
Content-Length: 82
Content-Type: text/html
Content-Language: en
X-Pad: avoid browser bug

<html><body>
<font color="chartreuse" size="18pt">Awesome!!</font>
</body></html>
Cool HTTP/1.1 features

• Persistent connections
  ‣ establishing a TCP connection is costly
    • multiple network “round trips” just to set up the TCP connection
    • TCP has a feature called “slow start”; slowly grows the rate at which a TCP connection transmits to avoid overwhelming networks
  ‣ a web page consists of multiple objects, and a client probably visits several pages on the same server
    • bad idea: separate TCP connection for each object
    • better idea: single TCP connection, multiple requests
    • *try it on* [www.cs.uoregon.edu](http://www.cs.uoregon.edu)
Today

• Sales pitch for upper-level systems-y courses

• Get some in-person feedback from you
Systems course map

- Programming languages
- Internet systems
- Distributed systems
- DB
- Compilers
- OS
- Network
- HW
- Computer design & organization

Modeling and Simulation
• Major topics:
  ‣ processes and threads (abstraction, implementation)
  ‣ synchronization (locks, semaphores, monitors, deadlock)
  ‣ CPU scheduling
  ‣ virtual memory, paging, page replacement
  ‣ storage: file system design, RAID
  ‣ IPC
Networking (CIS 432)

• Major topics:
  ‣ physical layer: bits, errors, a little bit of information theory
  ‣ link level protocols and media access control: Ethernet, ARP, switches, bridges
  ‣ network protocols: routing (BGP, Internet structure)
  ‣ transport protocols: TCP, congestion control, retransmission
  ‣ application-level protocols: Web, DNS
Databases (CIS 451)

• Major topics:
  ‣ SQL, views, and schema design
  ‣ database architecture
  ‣ transactions: concurrency control, recovery
  ‣ storage and indexing
  ‣ relational algebra
  ‣ query processing: algorithms, optimization
  ‣ parallel and distributed databases
Security (CIS 433)

• Major topics:
  ✦ software security (stack smashing, injection attacks, etc.)
  ✦ intro to cryptography
  ✦ web security
  ✦ authentication
  ✦ botnets
  ✦ privacy and anonymity
  ✦ network security
Distributed systems (CS 630)

• Major topics:
  ‣ time, clocks, and distributed synchronization
  ‣ rollback recovery
  ‣ fault tolerance through replication
  ‣ consensus: byzantine and fail-stop
  ‣ clusters and large-scale distributed systems design
  ‣ authentication in distributed systems
Goals for the course

• 1. Teach you C, C++
  ‣ language mechanics, pitfalls, bridging low-level to language

• 2. Teach you how to interact with the OS
  ‣ file system, network stack, asynchrony and concurrency

• 3. Teach you how to think in a systems-y way
  ‣ contending with multiple layers of abstraction simultaneously
  ‣ interfaces and architecture

• 4. Cultivate a sense of adventure
  ‣ courage to wade into a large system and understand it
Lectures

• Focus primarily on teaching you C, C++
  ‣ with occasional diversions into less hands-on, more abstract ideas

• Was this the right balance?
Project sequence

• Take you from the lowest-level, simplest data structure all the way to a fully-functioning search engine
  ‣ see each layer of abstraction as we build it up
  ‣ see the implications of good (and bad) design and interface decisions downstream in a project
  ‣ read and write a lot of code
    • experience more than just “100 line mini-projects”

• Hard part
  ‣ balancing how much we give you with how much you do on your own
Sections

• Original goal:
  ‣ reinforce lecture material, interesting additional topics, help with hard parts of projects

• An alternate model we could adopt:
  ‣ make the sections be a hands-on lab
    • some small-ish programming project each week
    • complete within the hour, with TAs there to guide you
Exercises

• If you did them all, you’d really master the topics
  ‣ but, we made some optional
  ‣ right balance having two problems a week?
Review

• Look at the topics covered for the midterm
• What have we covered since then?
Concurrency

• Sequential processes: when are they bad?
• What are alternatives?
  ‣ multiprocessing (fork)
  ‣ multithreading
• Event-driven programming
Inheritance

• C++ single and multiple inheritance
• Public, protected, private classes
• Dispatch: static and dynamic
  ‣ virtual function table
  ‣ virtual table pointer
• Constructors and destructors in derived classes
Smart pointers

• Why smart pointers? STL issues
• Slicing
• auto_ptr
• Casting
  ‣ static, dynamic, const, reinterpret
• Implicit casting
Networks

- OSI model
- File descriptors and socket descriptors
- Socket programming
- Stream and datagram sockets
- DNS resolution
- Clients and servers
- Blocking and concurrency
Thanks!