Today’s Lecture

• Project 1
  – Memory Manipulation
• Client / Server Programming
  – Finish Up
• Internetworking

Refresher – Memory Manipulation

• Reading in information
  – Socket
  – File
• How do we read in quantities more than a byte from the socket or file?
Option 1 – Buffer / Copy

```c
uint32_t theValue;
char byBuffer[300];
// Assume we have a socket named cs
read(cs, byBuffer, sizeof(uint32_t));
memcpy(&theValue, byBuffer, sizeof(uint32_t));
theValue = ntohl(theValue);
```

Option 2 – Direct Write

```c
uint32_t theValue;
char byBuffer[300];
// Assume we have a socket named cs
read(cs, (char *) &theValue, sizeof(uint32_t));
theValue = ntohl(theValue);
```

Other Reminders

- Make sure your pointer points somewhere

```c
struct timeval * pStartTime;
gettimeofday(pStartTime);
```

Where does pStartTime point?
Binary Data

- Binary data uses size rather than delimiter

```c
// Assume client socket is named cs
nBytes = read(cs, byBuffer, 16000);
byBuffer[nBytes] = '\0';
```

How do binary and string data compare to data headers from Chapter 2?

Tips – Fixing Size

- Types
  - Swap over from `short`, `int`, `long`, etc.
  - `long` on netscaleXX gives 64 bits
  - Not 32 bits as discussed in class
- Explicit typing
  - `#include <stdint.h>`
  - `uint8_t`
  - `uint16_t`
  - `uint32_t`
  - `uint64_t`

Thread Programming

- **Fork** a new process
  - Expensive (time, memory)
  - Interprocess communication is hard
- **Threads** are ‘lightweight’ processes
  - One process, many threads
  - Execute the same program in different parts
  - Share instructions, global memory, open files, and signal handlers.
Mutex – Mutual Exclusion

Server Models

- Iterative servers: process one request at a time.
- Concurrent server: process multiple requests simultaneously.
- Concurrent: better use of resources (service others while waiting) and incoming requests can start being processed immediately after reception.
- Basic server types:
  - Iterative connectionless.
  - Iterative connection-oriented.
  - Concurrent connectionless.
  - Concurrent connection-oriented.

Iterative Server

```c
int fd, newfd;
while (1) {
  newfd = accept(fd, ...);
  handle_request(newfd);
  close(newfd);
}
```

- simple
- potentially low resource utilization
- potentially long waiting queue (response times high, rejected requests)
Concurrent Connection-Oriented

1. Master: create a socket, bind it to a well-known address.
2. Master: Place the socket in passive mode.
3. Master: Repeatedly call accept to receive next request from a client, create a new slave process/thread to handle the response.
4. Slave: Begin with a connection passed from the master.
5. Interact with client using this connection (read request, send response).
6. Close the connection and exit.

select() Approach

- Single process manages multiple connections.
- Request treatment needs to be split into non-blocking stages.
- Data structure required to maintain state of each concurrent request.

1. Create a socket, bind to well-known port, add socket to list of those with possible I/O.
2. Use select() to wait for I/O on socket(s).
3. If ‘listening’ socket is ready, use accept to obtain a new connection and add new socket to list of those with possible I/O.
4. If some other socket is ready, receive request, form a response, send back.
5. Continue with step 2.
select()

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

- `nfds`: highest number assigned to a descriptor.
- block until 1 file descriptors have something to be read, written, or timeout.
- set bit mask for descriptors to watch using FD_SET.
- returns with bits for ready descriptor set: check with FD_ISSET.
- cannot specify amount of data ready.

fd_set

- void FD_ZERO(fd_set *fdset);
- void FD_SET(int fd, fd_set *fdset);
- void FD_CLR(int fd, fd_set *fdset);
- int FD_ISSET(int fd, fd_set *fdset);

- Create fd_set.
- Clear it with FD_ZERO.
- Add descriptors to watch with FD_SET.
- Call select.
- When select returns: use FD_ISSET to see if I/O is possible on each descriptor.

Example (simplified)

```c
int main(int argc, char *argv[])
{
    /* variables */
    s = socket(...); /* create socket */
    sin.sin_family = AF_INET;
    sin.sin_port = htons(atoi(argv[1]));
    sin.sin_addr.s_addr = INADDR_ANY;
    bind(s, ...);
    listen(s, 5);
    tv.tv_sec = 10;
    tv.tv_usec = 0;
    FD_ZERO(&rfds);
    if (s > 0) FD_SET(s, &rfds);
    ```
Example (contd)

```c
while (1) {
    n = select(FD_SETSIZE, &rfds, NULL, NULL, &tv);
    if (n == 0) printf("Timeout!");
    else if (n > 0) {
        if (FD_ISSET(s, &rfds)) {
            t = 0;
            while (t = accept(...) > 0) {
                FD_SET(t, &rfds);
            }
        }
    }
}
```

Example (contd)

```c
for (i = ...) {
    if (FD_ISSET(i, &rfds)) {
        handle_request(i);
    } 
... 

– handle_request: reads request, sends response, closes socket if client done, calls FD_CLR
```

Summary

- Iterative
  - Single network connection at a time
- Concurrent – Threads
  - Spawn thread for each client
  - Fork process for each client
- Hybrid
  - Monitor multiple clients via select

Next week: Look at signals
http://www.cs.utah.edu/dept/old/techinfo/glibc-manual-0.02/library_21.html
Internetworking

Outline
- Best Effort Service Model
- Global Addressing Scheme

IP Internet

- Concatenation of Networks
- Protocol Stack

Network of Networks

AS = Autonomous System
Service Model - Internet

- Connectionless (datagram-like)
- Best-effort delivery (unreliable service)
  - Packets are lost
  - Packets may be delivered out of order
  - Duplicate copies of a packet may be delivered
  - Packets can be delayed for a long time
- Datagram format

IP Header

20 bytes (usually)

- **Version**: 4 or 6
- **HLen**: Length of header (in 4 byte chunks), usually 5
- **TOS**: Type of service
- **Length**: Length of packet including IP header

IP Header

- **Ident**: Identifier (for reassembly)
- **Flags**: Fragment, Don’t Fragment
- **Offset**: Offset for reassembly
- **TTL**: Time to live, decrements with each router hop
- **Protocol**: Upper layer protocol
  - 6 = TCP
  - 17 = UDP
IP Header

Checksum: Ones complement checksum of header
Source Addr: Source address
192.168.1.17
Destination Addr: Destination address
129.74.153.157
Options: May or may not be present (usually not), always pad out to 32 bits for all options

Fragmentation and Reassembly

- Each network has a MTU
  - Maximum Transfer Unit
  - Ethernet = 1500 bytes, FDDI = 4500 bytes, Modem = 512 bytes
- Design decisions
  - Fragment when necessary (MTU < Datagram)
  - Try to avoid fragmentation at source host
  - Re-fragmentation is possible
  - Fragments are self-contained datagrams
  - Use CS-PDU (not cells) for ATM
  - Delay reassembly until destination host
  - Do not recover from lost fragments

Fragmentation = bad, bad, bad, bad!

Example
Global Addresses

- Properties
  - Globally unique
  - Hierarchical: network + host

- Dot Notation
  - 10.3.2.4
  - 128.96.33.81
  - 192.12.69.77

Address Categories

- Class A - /8
  - 10.0.0.0 / 8
  - 10.*

- Class B - /16
  - 192.168.0.0 / 16
  - 192.168.*

- Class C - /24
  - 129.74.153.*
  - 129.74.153.0 / 24

Network Mask

- Bits to pay attention to when determining a subnet
  - 255.255.255.0

Subnet

- Result after applying net mask
  - 129.74.153.0

Masking

- Use bit-wise AND result
  - 1 & X =
  - 0 & X =

Address

- 129.74.20.40
- 128.74.20.0
Routing Table – netscale01

<table>
<thead>
<tr>
<th>Source IP Address</th>
<th>Destination IP Address</th>
<th>Gateway</th>
<th>Mask</th>
<th>Flags</th>
<th>Hop Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>192.168.1.0/24</td>
<td>192.168.2.0</td>
<td>192.168.2.1</td>
<td>255.255.255.0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Default</td>
<td>192.168.2.0</td>
<td>0</td>
<td>255.255.255.255</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

Local or next hop?

Mask result

Interface to pass to

Datagram Forwarding

• Strategy
  – Always have dest address
    • In IP header
  – Two choices
    • Local network (subnet)
      – Pass it off directly
    • Not on my local network
      – Pass to some router
  – Routing table
    • Maps network to next hop
    • Routing entries
    • Default router

Address Translation

• Map IP addresses into physical addresses
  – Destination host
  – Next hop router
• Techniques
  – Encode physical address in host part of IP address
  – Table-based
• ARP
  – Table of IP to physical address bindings
  – Broadcast request if IP address not in table
  – Target machine responds with its physical address
  – Table entries are discarded if not refreshed
ARP Details

- Request Format
  - HardwareType: type of physical network (e.g., Ethernet)
  - ProtocolType: type of higher layer protocol (e.g., IP)
  - HLEN & PLEN: length of physical and protocol addresses
  - Operation: request or response
  - Source/Target-Physical/Protocol addresses

- Notes
  - table entries timeout in about 15 minutes
  - update table with source when you are the target
  - update table if already have an entry
  - do not refresh table entries upon reference

ARP Packet Format

- Hardware type = 1
- ProtocolType = 0x0800
- HLen = 48
- PLen = 32
- SourceHardwareAddr (bytes 4 – 5)
- SourceProtocolAddr (bytes 0 – 1)
- TargetHardwareAddr (bytes 0 – 11)
- TargetProtocolAddr (bytes 0 – 3)

DHCP

- Dynamic Host Configuration Protocol

[Diagram showing DHCP process]
DHCP

Internet Control Message Protocol (ICMP)

- Echo (ping)
- Redirect (from router to source host)
- Destination unreachable (protocol, port, or host)
- TTL exceeded (so datagrams don’t cycle forever)
- Checksum failed
- Reassembly failed
- Cannot fragment