Week 6 - Friday

COMP 3400

Last time

- What did we talk about last time?
- Networking

Questions?

Assignment 4

Exam 1 Post Mortem

Sockets

Sockets

- The last class was a high-level overview of networking
- Now, we'll look at how to turn those ideas into code
- The most basic element of the networking arsenal is the **socket**
- A socket is half of a two-way connection between hosts
- We create a socket with a call to **`socket()`**

```
int socket (int domain, int type, int protocol);
```

  - Returns an **`int`**, essentially a file descriptor
  - Is similar to calling **`open()`** on a file
  - We can call **`read()`** and **`write()`** on socket file descriptors

IP addresses

- Computers on the Internet have addresses, not names
- **Google.com** is actually **`74.125.67.100`**
- **Google.com** is called a **domain**
- The Domain Name System or DNS turns the name into an address

IPv4

- Old-style IP addresses are often written in this form:
  - **`74.125.67.100`**
- 4 numbers between 0 and 255, separated by dots
- That's a total of 256^4 = 4,294,967,296 addresses
- But there are 8 billion people on earth...

IPv6

- IPv6 are the new IP addresses that are beginning to be used by modern hardware
  - 8 groups of 4 hexadecimal digits each
  - **`2001:0db8:85a3:0000:0000:8a2e:0370:7334`**
  - 1 hexadecimal digit has 16 possibilities
  - How many different addresses is this?
  - $16^{32} = 2^{128} \approx 3.4 \times 10^{38}$ is enough to have 500 trillion addresses for every cell of every person's body on Earth
  - Will it be enough?!

Details for socket()

```
int socket (int domain, int type, int protocol);
```

- **domain**
  - What the socket will be used for
  - Typical values are IPv4, IPv6, or local communication
- **type**
  - Determines the transport layer
  - Usually TCP or UDP for this class
- **protocol**
  - Usually not used and set to **0**
  - Can be used for special raw sockets used for packer sniffers

Field	Constant	Purpose
domain	AF_INET	IPv4 addresses
	AF_INET6	IPv6 addresses
	AF_LOCAL	Unix domain socket for IPC
	AF_NETLINK	Netlink socket for kernel messages
	AF_PACKET	Raw socket type
type	SOCK_STREAM	Byte-stream communication, for TCP transport
	SOCK_DGRAM	Fixed-size messages, for UDP transport
	SOCK_RAW	Raw data that is not processed by transport layer
protocol	IPPROTO_RAW	IP datagrams without transport-layer processing
	ETH_P_ALL	Ethernet frames without network-layer processing

Example calls to socket ( )

Purpose	Call
IPv4 socket for TCP	`socketfd = socket (AF_INET, SOCK_STREAM, 0);`
IPv6 socket for TCP	`socketfd = socket (AF_INET6, SOCK_STREAM, 0);`
IPv4 socket for UDP	`socketfd = socket (AF_INET, SOCK_DGRAM, 0);`
IPv6 socket for UDP	`socketfd = socket (AF_INET6, SOCK_DGRAM, 0);`
Raw socket for sniffing unprocessed Ethernet frames	`socketfd = socket (AF_PACKET, SOCK_RAW, htons (ETH_P_ALL));`

Networking data structures

- Different data structures are needed to specify addresses depending on what kind of networking is being done
- Since C doesn't have inheritance, structs with the same size are treated interchangeably and then cast to each other when appropriate
- One of these is **`struct sockaddr`**, which is 16 bytes in size

```
// generic address structure
struct sockaddr {
  sa_family_t sa_family; // two bytes: AF_INET, etc.
  char sa_data[14];
};
```

IPv4 socket addresses

- The structure for holding IPv4 addresses is identical in size to **`struct sockaddr`**

```
// IPv4 address structure
struct sockaddr_in {
  sa_family_t sin_family;
  in_port_t sin_port;
  struct in_addr sin_addr;
  char sin_zero[8];
};
struct in_addr {
  in_addr_t s_addr; // in_addr_t is an alias for uint32_t
};
```

Type	struct sockaddr															
Fields	sa_family		sa_data													
Data	02	00	00	50	5d	b8	d8	22	00	00	00	00	00	00	00	00
Fields	sin_family		sin_port		sin_addr				sin_zero							
Type	struct sockaddr_in															

IPv6 socket addresses

- IPv6 addresses are longer and consequently require bigger (and stranger looking) structs

```
// IPv6 address structure
struct sockaddr_in6 {
  sa_family_t sin6_family;
  in_port_t sin6_port;
  uint32_t sin6_flowinfo;
  struct in6_addr sin6_addr;  // IPv6 addresses are 128-bit
  uint32_t sin6_scope_id;
};

struct in6_addr {
  union {
    uint8_t  __u6_addr8[16];  // aliased as s6_addr
    uint16_t __u6_addr16[8];  // aliased as s6_addr16
    uint32_t __u6_addr32[4];  // aliased as s6_addr32
  } __u6_addr;
};
```

Good news and bad news

- The good news is that you (usually) don't have to muck around in the parts of the structs that represent actual IP addresses
  - These are bytes laid out in specific patterns
  - Not user-friendly representations like `74.125.67.100`
  - Functions handle the translation for you
- The bad news is that some values inside of these structs are sensitive to **endianness**
  - Which byte is considered the most significant in a machine
  - When networking, important data like ports and addresses are sent between machines with potentially different endianness

Endian conversion

- Rather than try to keep straight what the endianness of our machine and the endianness of the network is, we use a family of functions:
  - **`hton`**: host to network endianness
  - **`ntoh`**: network to host endianness
  - They come in **`l`** (long) versions (for 32-bit integers) or **`s`** (short) versions (for 16-bit integers)

```
uint32_t htonl (uint32_t hostlong);   // 32-bit from host to network
uint16_t htons (uint16_t hostshort);  // 16-bit from host to network
uint32_t ntohl (uint32_t netlong);    // 32-bit from network to host
uint16_t ntohs (uint16_t netshort);   // 16-bit from network to host
```

Getting addresses from a host name

- DNS converts a host name to an IP address
- The **`getaddrinfo()`** function lets us get a linked list of matching addresses

```
int getaddrinfo (const char *name, const char *service,
const struct addrinfo *hints, struct addrinfo **results)
```

- The only annoying bit is that we have to fill out a hints structure
- A utility function **`freeaddrinfo()`** is provided to free the linked list structure when done with it

```
void freeaddrinfo (struct addrinfo *info);
```

The `addrinfo` struct

- The result of **`getaddrinfo()`** is stored into the pointer given by the last argument

```
struct addrinfo {
  int ai_flags;
  int ai_family;
  int ai_socktype;
  int ai_protocol;
  socklen_t ai_addrlen;
  char *ai_canonname;
  struct sockaddr *ai_addr;  // Pointer to address we need
  struct addrinfo *ai_next;  // Pointer to next addrinfo in linked list
};
```

Getting address example

```
struct addrinfo hints, *server_list = NULL, *server = NULL;
memset (&hints, 0, sizeof (hints));
hints.ai_family = AF_INET;         // IPv4
hints.ai_socktype = SOCK_STREAM; // Byte-streams (TCP)
hints.ai_protocol = IPPROTO_TCP; // TCP
assert (getaddrinfo (hostname, "http", &hints, &server_list) == 0); // Get addresses

for (server = server_list; server != NULL; server = server->ai_next)
 {
    if (server->ai_family == AF_INET) // Only take IPv4
      {
        // Cast to IPv4 socket
        struct sockaddr_in *addr = (struct sockaddr_in *)server->ai_addr;
        printf ("IPv4 address: %s\n", inet_ntoa (addr->sin_addr));
      }
  }
freeaddrinfo (server_list);
```

Confusing structs!

- Here's a visualization of the **addrinfo** and **sockaddr** structs that might come back from **getaddrinfo()**

struct addrinfo	
ai_flags	…
ai_family	AF_INET
ai_socktype	SOCK_STREAM
ai_protocol	IPPROTO_TCP
ai_addrlen	4
ai_canonname	…
ai_addr	●
ai_next	●

struct addrinfo	
ai_flags	…
ai_family	AF_INET6
ai_socktype	SOCK_STREAM
ai_protocol	IPPROTO_TCP
ai_addrlen	16
ai_canonname	…
ai_addr	●
ai_next	NULL

struct sockaddr_in6	
sin6_family	AF_INET6
sin6_port	80
sin6_flowinfo	…
sin6_addr	2606:2800:220:1: 248:1893:25C8:1946
sin6_scope_id	…

struct sockaddr_in	
sin_family	AF_INET
sin_port	80
sin_addr	93.184.216.34
sin_zero	0

Programming practice

- Adapt the code on the previous slide:
  - Read a host or IP address from the user
  - Read a service or port name from the user
  - Print out the resulting IP addresses

Note the following common port names and services:

Port	Name	Service
21	FTP	Insecure file transfer
22	SSH	Secure shell
23	Telnet	Insecure remote access
25	SMTP	Email delivery
53	DNS	IP address lookup
67	DHCP	IP address assignment
68	DHCP	IP address assignment
80	HTTP	Web page
88	Kerberos	Authentication

Port	Name	Service
110	POP3	POP email access
123	NTP	Time synchronization
143	IMAP	IMAP email access
194	IRC	Internet chat service
389	LDAP	Authentication
443	HTTPS	Secure web page
530	RPC	Remote procedure call
631	IPP	Internet printing
993	IMAPS	Secure IMAP access

Ticket Out the Door

Upcoming

Next time...

- TCP socket programming

Reminders

- Work on Assignment 4
  - Due next Monday
- Start on Project 2!
- Read section 4.5