OS view of networking –
Sockets API
(an exercise in planning for the future)

David E. Culler
CS162 – Operating Systems and Systems Programming
Lecture 5
Sept. 10, 2014

Adjustment on Culler Office Hours:
- Tue 9-10, Wed 2-3, Th 1-2 in 449 Soda

Reading: OSC 2.7, 3.6
HW: 1 is out, due 9/15
Proj:
Real Reading

• Unix Network Programming. The Sockets Networking API, Stevens (et al), Ch 3-5 “Elementary Sockets”
• Lots of on-line tutorials
• This lecture and the code
  • http://cs162.eecs.berkeley.edu/static/lectures/code05/eclient.c
  • http://cs162.eecs.berkeley.edu/static/lectures/code05/eserver.c
  • http://cs162.eecs.berkeley.edu/static/lectures/code05/feserver.c
Communication between processes

```c
write(wfd, wbuf, wlen);

n = read(rfd, rbuf, rmax);
```

- Producer and Consumer of a file may be distinct processes
- May be separated in time (or not)
Communication Across the world looks like file IO

write(wfd, wbuf, wlen);

n = read(rfd, rbuf, rmax);

• But what’s the analog of open?
• What is the namespace?
• How are they connected in time?
Request Response Protocol

Client (issues requests)  Server (performs operations)

write(rqfd, rqbuf, buflen);

| requests | | | |

wait

n = read(rfd, rbuf, rmax);

| service request |

write(wfd, respbuf, len);

| responses | | | |

n = read(resfd, resbuf, resmax);
Request Response Protocol

Client (issues requests) | Server (performs operations)

write(rqfd, rqbuf, buflen);

n = read(rfd, rbuf, rmax);

write(wfd, respbuf, len);

n = read(resfd, resbuf, resmax);
Client-Server Models

- File servers, web, FTP, Databases, ...
- Many clients accessing a common server
Sockets

• Mechanism for inter-process communication
• Data transfer like files
  – Read / Write against a descriptor
• Over ANY kind of network
  – Local to a machine
  – Over the internet (TCP/IP, UDP/IP)
  – OSI, Appletalk, SNA, IPX, SIP, NS, ...
Silly Echo Server – running example

Client (issues requests)

Server (performs operations)

write(fd, buf, len);

n = read(fd, buf,);

print

wait

gets(fd, sndbuf, ...);

write(fd, buf, len);

n = read(fd, recvbuf,);

print

print
void client(int sockfd) {
    int n;
    char sndbuf[MAXIN]; char rcvbuf[MAXOUT];
    getreq(sndbuf, MAXIN); /* prompt */
    while (strlen(sndbuf) > 0) {
        write(sockfd, sndbuf, strlen(sndbuf)); /* send */
        memset(rcvbuf,0,MAXOUT); /* clear */
        n=read(sockfd, rcvbuf, MAXOUT-1); /* receive */
        write(STDOUT_FILENO, rcvbuf, n); /* echo */
        getreq(sndbuf, MAXIN); /* prompt */
    }
}

void server(int consockfd) {
    char reqbuf[MAXREQ];
    int n;
    while (1) {
        memset(reqbuf,0, MAXREQ);
        n = read(consockfd,reqbuf,MAXREQ-1); /*Recv*/
        if (n <= 0) return;
        n = write(STDOUT_FILENO, reqbuf, strlen(reqbuf));
        n = write(consockfd, reqbuf, strlen(reqbuf)); /* echo*/
    }
}
char *getreq(char *inbuf, int len) {
    /* Get request char stream */
    printf("REQ: ");       /* prompt */
    memset(inbuf,0,len);    /* clear for good measure */
    return fgets(inbuf,len,stdin); /* read up to a EOL */
}
Socket creation and connection

- File systems provide a collection of permanent objects in structured name space
  - Processes open, read/write/close them
  - Files exist independent of the processes
- Sockets provide a means for processes to communicate (transfer data) to other processes.
- Creation and connection is more complex
- Form 2-way pipes between processes
  - Possibly worlds away
Sockets in concept

Client

Create Client Socket

Connect it to server (host:port)

write request

read response

Close Client Socket

Server

Create Server Socket

Bind it to an Address (host:port)

Listen for Connection

Accept connection

Connection Socket

read request

write response

Close Connection Socket

Close Server Socket
char *hostname;
int sockfd, portno;
struct sockaddr_in serv_addr;
struct hostent *server;

server = buildServerAddr(&serv_addr, hostname, portno);

/* Create a TCP socket */
sockfd = socket(AF_INET, SOCK_STREAM, 0)

/* Connect to server on port */
connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)
printf("Connected to %s:%d\n",server->h_name, portno);

/* Carry out Client-Server protocol */
client(sockfd);

/* Clean up on termination */
close(sockfd);
/* Create Socket to receive requests*/
lstnsockfd = socket(AF_INET, SOCK_STREAM, 0);

/* Bind socket to port */
bind(lstnsockfd, (struct sockaddr *)&serv_addr,sizeof(serv_addr));
while (1) {
    /* Listen for incoming connections */
    listen(lstnsockfd, MAXQUEUE);

    /* Accept incoming connection, obtaining a new socket for it */
    consockfd = accept(lstnsockfd, (struct sockaddr *) &cli_addr, &clilen);

    server(consockfd);

    close(consockfd);
}
close(lstnsockfd);
Administrative break
How does the server protect itself?

- Isolate the handling of each connection
- By forking it off as another process
Sockets in concept

**Client**

- Create Client Socket
- Connect it to server (host:port)
- write request
- read response
- Close Client Socket

**Server**

- Create Server Socket
- Bind it to an Address (host:port)
- Listen for Connection
- Accept connection
  - Connection Socket
- write response
- read request
- Close Connection Socket
- Close Listen Socket
- Wait for child

**Connection Socket**

- Close Connection Socket
- Close Server Socket

---

9/10/14  

CS162 FA14 L5
while (1) {
    listen(lstnsockfd, MAXQUEUE);
    consockfd = accept(lstnsockfd, (struct sockaddr *) &cli_addr, &clilen);
    cpid = fork(); /* new process for connection */
    if (cpid > 0) { /* parent process */
        close(consockfd);
        tcpid = wait(&cstatus);
    } else if (cpid == 0) { /* child process */
        close(lstnsockfd); /* let go of listen socket */
        server(consockfd);
        close(consockfd);
        exit(EXIT_SUCCESS); /* exit child normally */
    }
}
close(lstnsockfd);
Concurrent Server

• Listen will queue requests
• Buffering present elsewhere
• But server waits for each connection to terminate before initiating the next
Sockets in concept

Client

1. Create Client Socket
2. Connect it to server (host:port)
   - write request
   - read response
3. Close Client Socket

Server

1. Create Server Socket
2. Bind it to an Address (host:port)
3. Listen for Connection
4. Accept connection
5. Close Connection Socket
6. Close Listen Socket
7. Close Server Socket
8. Close Connection Socket

(child) Connection Socket
(Parent) Connection Socket

while (1) {
    listen(lstnsockfd, MAXQUEUE);
    consockfd = accept(lstnsockfd, (struct sockaddr *) &cli_addr, &clilen);
    cpid = fork(); /* new process for connection */
    if (cpid > 0) { /* parent process */
        close(consockfd);
        //tcpid = wait(&cstatus);
    } else if (cpid == 0) { /* child process */
        close(lstnsockfd); /* let go of listen socket */

        server(consockfd);

        close(consockfd);
        exit(EXIT_SUCCESS); /* exit child normally */
    }  
}

close(lstnsockfd);
Server Address - itself

memset((char *) &serv_addr,0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(portno);

- Simple form
- Internet Protocol
- accepting any connections on the specified port
- In “network byte ordering”
Client: getting the server address

struct hostent *buildServerAddr(struct sockaddr_in *serv_addr,
                         char *hostname, int portno) {

        struct hostent *server;

    /* Get host entry associated with a hostname or IP address */
    server = gethostbyname(hostname);
    if (server == NULL) {
        fprintf(stderr,"ERROR, no such host\n");
        exit(1);
    }

    /* Construct an address for remote server */
    memset((char *) serv_addr, 0, sizeof(struct sockaddr_in));
    serv_addr->sin_family = AF_INET;
    bcopy((char *)server->h_addr,
          (char *)&(serv_addr->sin_addr.s_addr), server->h_length);
    serv_addr->sin_port = htons(portno);

    return server;
}
Namespaces for communication

• Hostname
  – www.eecs.berkeley.edu

• IP address
  – 128.32.244.172 (ipv6?)

• Port Number
  – 0-1023 are “well known” or “system” ports
    • Superuser privileges to bind to one
  – 1024 – 49151 are “registered” ports (registry)
    • Assigned by IANA for specific services
  – 49152–65535 ($2^{15}+2^{14}$ to $2^{16}–1$) are “dynamic” or “private”
    • Automatically allocated as “ephemeral Ports”
Recall: UNIX Process Management

• UNIX fork – system call to create a copy of the current process, and start it running
  – No arguments!
• UNIX exec – system call to change the program being run by the current process
• UNIX wait – system call to wait for a process to finish
• UNIX signal – system call to send a notification to another process
Signals – infloop.c

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>

#include <unistd.h>
#include <signal.h>

void signal_callback_handler(int signum)
{
    printf("Caught signal %d - phew!\n", signum);
    exit(1);
}

int main() {
    signal(SIGINT, signal_callback_handler);

    while (1) {}
}
if (cpid > 0) {
    mypid = getpid();
    printf("[\%d] parent of \%d\n", mypid, cpid);
    for (i=0; i<100; i++) {
        printf("[\%d] parent: %d\n", mypid, i);
        //      sleep(1);
    }
} else if (cpid == 0) {
    mypid = getpid();
    printf("[\%d] child\n", mypid);
    for (i=0; i>-100; i--) {
        printf("[\%d] child: %d\n", mypid, i);
        //      sleep(1);
    }
}
BIG OS Concepts so far

- Processes
- Address Space
- Protection
- Dual Mode
- Interrupt handlers (including syscall and trap)
- File System
  - Integrates processes, users, cwd, protection
- Key Layers: OS Lib, Syscall, Subsystem, Driver
  - User handler on OS descriptors
- Process control
  - fork, wait, signal --- exec
- Communication through sockets
- Client-Server Protocol
Course Structure: Spiral

Address Space (4)

File Systems (8)

Concurrency (6)

Reliability, Security, Cloud (8)

Distributed Systems (8)

Os Concepts (3)

intro