Lecture 9
Servers
February 2, 2005

Ports
- Some well-known ports:
  - 80: http
  - 21: ftp
  - 23: telnet
  - 22: ssh
  - 79: finger
  - 42: nameserver
  - 13: daytime
  - 7: echo
  - etc/services

Ports
- Ports assigned to standard Internet applications (ftp, telnet, smtp, etc) between 1 – 255: Reserved
  - Can be used only with superuser privileges
  - Ports in range 256 – 511 are reserved, not used
  - Ports in range 1024 – 5000 are automatically assigned by UNIX to clients
  - Ports greater than 5000 are for user-developed servers

<table>
<thead>
<tr>
<th>Port Type</th>
<th>Port Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved ports (only superuser)</td>
<td>1 – 1023</td>
</tr>
<tr>
<td>Ports automatically assigned by the system</td>
<td>1024 – 5000</td>
</tr>
<tr>
<td>Ports used by rcmd()</td>
<td>512 – 1023</td>
</tr>
</tbody>
</table>

Iterative vs. Concurrent
- 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.

Connection-Oriented/Less
- TCP/IP: TCP (connection-oriented), UDP (connection-less) servers.
- TCP:
  - point-to-point communication: 2 endpoints
  - reliable connection establishment (tests network connectivity)
  - reliable delivery: same order, no loss, no duplication
  - flow-controlled transfer: fast-slow computers
  - full-duplex transfer: simult. in either direction
  - stream paradigm: byte streams, no message boundaries.

Connectionless
- UDP:
  - many-to-many connection: multiple senders/receivers, multicast, broadcast
  - unreliable service: can be lost, duplicated, out of order, no re-transmission, no signalling
  - no flow control: if too fast, packets are discarded
  - message paradigm: sender specifies data size, UDP places data in single outgoing message
Basic Server Types

- Iterative connectionless.
- Iterative connection-oriented.
- Concurrent connectionless.
- Concurrent connection-oriented.

Iterative Connectionless

1. Create socket, bind to well-known address.
2. Repeatedly read next request from a client, formulate response, send reply back.

Iterative Connection-Oriented

1. Create socket, bind to well-known address.
2. Place socket in passive mode.
3. Accept next connection request, obtain new socket.
4. Read request, write response, repeat.
5. When finished, close connection, return to step 3.

Concurrent Connectionless

1. Master: create socket, bind to well-known address.
2. Master: repeatedly call recvfrom to receive next request from a client, create a new slave process/thread to handle the response.
3. Slave: begin with specific request from the master as well as access to the socket.
4. Slave: form a reply and send it to client using sendto.
5. Slave: exit.

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.
One Slave Per Client

```c
void sig_chld(int) {
    while (waitpid(0, NULL, WNOHANG) > 0) {} 
    signal(SIGCHLD, sig_chld); }
int main() {
    int fd, newfd, pid;  
    signal(SIGCHLD, sig_chld);  
    while (1) {
        newfd = accept(fd, ...); 
        if (newfd < 0) continue;
        pid = fork(); 
        if (pid == 0) {handle_request(newfd); exit(0); } 
        else {close(newfd); } 
    }
}
```

- Several requests handled simultaneously.
- Incoming requests are handled ‘immediately’.
- `fork()` is heavy-weight.
- One thread (instead of process) per client works the same.

Processes vs. Threads

- Master process
- Slave processes

- Master thread
- Slave threads

Process Pool

```c
#define NB_PROC 10
void recv_requests(int fd) {
    int f;
    while (1) {
        f = accept(fd, ...);
        handle_request(f);
        close(f);
    }
}
int main() {
    int fd;
    for (int i=0; i<NB_PROC; i++) {
        if (fork() == 0) recv_requests(fd);
    }
    while (1) pause();
}
```

- Concurrent request treatment.
- No time wasted on `fork()`.
- Number of concurrent requests can be limited.
- Example: Apache.
- Variant 1: thread pool.
- Variant 2: dynamically sized process pool.

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.
- Example: Squid Web Cache.
select() Approach

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 *writfds, 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.

Outline

- Create fd_set.
- Clear it with FD_ZERO.
- Add descriptors to watch with FD_SET.
- Call select.
- 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(...);
    listen(s);
    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);
            }
        }
    }
```

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);
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

Multiservice Servers

- Merging multiple protocols into one server.
- Merging multiple services into one server.

```
Server
  ↓
Sockets
  ↓
  ... iterative connectionless multiservice
```

```
Server
  ↓
Sockets
  ↓
  ... iterative connectionoriented multiservice
```

```
Server
  ↓
Sockets
  ↓
  ... concurrent connectionoriented multiservice
```

```
Server
  ↓
Sockets
  ↓
  ... connectionoriented multiservice with execve
```

Multiservice Servers

- Separation of master server and actual services.
- Allows separate maintenance or replacement without recompilation.
- Called “super server”.
- Services can easily be added/removed/replaced.
- Static configuration: when server starts up (file: services + executable programs), changes require restart.
- Dynamic configuration: same as above but admin can notify server of changes (e.g., signals), no restart required.

Inetd

- Dynamically configurable (/etc/inetd.conf)
- Fields:
  - service name
  - socket type (stream, dgram)
  - protocol (tcp, udp)
  - wait status (wait, nowait)
  - userid
  - server program (program or internal)
  - arguments
**Inetd**
- Master socket for each service.
- If request arrives, uses file info to decide on how to proceed.
- `wait`: iterative, concurrent, UDP: block server from using socket until service program finishes (multiple datagrams).

**inetd**
- Server processes or daemons: routed, snmpd, in.rlogind, ftpd, mountd, telnetd, ...
- Each service one process: lots of idle processes and listening sockets.
- `inetd`: Internet services daemon.
- Opens sockets, forks processes when requests arrive.
- `stdin/stdout` connected to network.
- Problem: process creation overhead.

**Server Deadlock**
- Example: iterative connection-oriented server.
- Client requests a connection but never sends a request.
- Server blocks at read call.

**Server Deadlock**
- Client makes a connection, sends a sequence of requests, never reads the responses.
- When client's buffer fills, TCP stops transmitting data (flow control).
- Server continues to generate responses.
- Server’s buffer fills and the server blocks.

**Deadlock**
- General: a program blocks if the operating system cannot satisfy a system call.
- Slaves: the slave sending the responses blocks.
- Single-threaded: central server blocks.
- Any server using one thread can be subject to deadlock.