Overview

- Deciding on type of connection
  - Packet oriented or connection oriented
- Deciding on type of server
  - Iterative or concurrent
  - Stateless or stateful
- Basic paradigms

Type of Connection

- Connection-Oriented vs Connectionless
  - Connection-Oriented use TCP
  - Connectionless use UDP
- Application will dictate which one to use
  - UDP
    - Quick one time transfer
    - Loss can be tolerated
  - TCP
    - Need for message control
    - Can accept additional overhead
- UDP
  - Quick one time transfer
  - Loss can be tolerated
- TCP
  - Need for message control
  - Can accept additional overhead
Type of Server

- Iterative
  - Processes 1 request at a time
  - Easier to build
  - Useful for simple, quick connections – e.g. time
- Concurrent
  - Processes multiple requests at a time
  - More difficult to build
  - Useful for connections that involve asynchronous user input

State vs Stateless servers

- Maintaining state information must be done carefully since it can cause problems in an unreliable network or with client crashes
- The “best” servers are those that maintain no state – they are called idempotent
- If the server doesn’t maintain state, the client must.
  - Cookies provide one example of how to maintain state information. The SUN NFS protocol is another example
- Having the client maintain state increases network traffic

Sockets and Servers

![Diagram of Sockets and Servers]

- Server
  - socket bind
- OS
  - socket bound to an address
What Constitutes a Connectionless Connection?

- In connectionless systems, the client sends a message using sendto and the server reads it with recvfrom and responds using sendto
- Each message is considered unique and generally speaking servers of these types respond based only on the data in the message.

The Algorithm Graphically

```
socket
bind
recvfrom
sendto
```

Connectionless Server - Getting Client Address

- Recvfrom is a blocking call that reads a message from the bound socket. The address of the sending client is in from
  
  ```c
  retcode = recvfrom(s, msg,len,flags,from,fromlen);
  /*
   * s = socket descriptor
   * msg = address to hold incoming message
   * len = length of buf (bytes)
   * from = sockaddr structure for address used in sendto call
   * fromlen = will contain length of source address
  */
  ```
Connectionless Server Reply

• sendto sends the response to the address obtained from the recvfrom
  
  Retcode= sendto(s,message,len,flags,toaddr,toaddrlen);
  
  s = socket descriptor
  message = data buffer address
  len = length of message buffer (bytes)
  flags = control info
  toaddr and toaddrlen = sockaddr client address

What Constitutes a Connection Oriented Connection?

• How do multiple connections to the same port on a server get distinguished?
• How do multiple connections to one machine get distinguished when they all come from a single machine.

Iterative Connection-Oriented Server Algorithm

• Create a socket – socket of type TCP
• Bind to a port -- bind
• Place socket in passive mode -- listen
• Wait for a connection request, getting a new socket to handle the request -- accept
• Using the socket returned by accept, read client request, process request and respond
• When finished, close the slave socket and return to accept another request on the master socket
Comer Utility Function

- passiveTCP() function
  - Arguments are (service, qlen)
  - Calls utility function passivesock
- passivesock() function
  - Arguments are (service, “tcp”, qlen)
  - Returns value for success (0) or failure (-1)
  - Define structures for service entity (servent), protocol entity (protoent), and Internet endpoint address (sin).
  - Place values into sin (AF_INET, INADDR_ANY)
  - Map the service and protocol name.
  - Allocate a socket, bind it, and for tcp listen on it

The Algorithm Graphically

- socket
- bind
- listen
- accept
- read
- write
- close

Master/Slave Sockets

- SERVER
  - Socket for connection requests
  - Socket for individual Client connections
**Binding and INADDR_ANY**

- `bind` specifies an endpoint address that is in the structure sockaddr_in (IP and port)
- If the server connected to more than 1 network, or has multiple IP addresses, which one will be used?
  - The constant INADDR_ANY tells the OS to process datagrams for this service from any host IP addresses that are valid for the host

**Pending Requests**

- The LISTEN call has a parameter which defines the maximum number of connection requests to queue.
- Requests that arrive while the server is processing a request are queued up to limit
- Requests that arrive in excess of the limit are rejected
- ACCEPT returns the descriptor of the socket to be used for the new connection

**Concurrent Processing**

- Servers try to provide concurrent processing to clients
  - There is no such thing as actual concurrent processing unless the machine has multiple CPUs
  - Apparent concurrency and network based concurrency are a function of the difference in view from the client and the server
- Server concurrency occurs via process forking
  - Forked processes use different data sets, instruction pointers, and stacks
  - Parents and children in a fork and share I/O ports open before the fork
**Concurrent Connection-Oriented Server**

- Server creates a new process (child) using “fork”
- The child process uses the second or slave socket returned by accept to communicate with the client. The child process should:
  - Close the master socket when created
  - Close the slave socket and exit when done
- The main server or parent process never interacts with client. The parent process should:
  - Close the slave socket after forking
  - Return immediately to accept new requests
  - Handle signals from child processes

**Master/Slave Sockets**

[Diagram showing Parent and Child processes with Master/Slave Sockets]

- Socket for connection requests
- Socket for individual Client connections

**Concurrent Connection-Oriented Server Algorithm**

- Create a socket and bind to well known address for the service - leave unconnected
- Place socket in passive mode - listen
- Repeatedly call ACCEPT to receive requests
  - Create a new child process to handle requests
  - Child process reads requests and responds
  - When finished, close connection and exit
Separate Programs as Slaves

- When service is complex, it is more convenient to use a separately compiled program to handle the service
- Slave process call a separate program with the EXECVE system call after the call to fork();

Single Process Implementation of a Concurrent Server

- Uses a list of sockets connected to multiple clients
- Use the SELECT system call to determine which one has I/O activity
  - If original socket ready, then use ACCEPT for a new socket and add to list for SELECT
  - else use READ to form a response and WRITE to send a reply

Concurrent Connection-Oriented Server

- Master Process
- slave 1
- slave 2
- ...
Functionality for Server Processes

- All servers (iterative and concurrent) should
  - Fork and reparent the Master process
  - Disconnect from terminal I/O
  - Change file I/O defaults
  - Optional alarm on server
  - Provide for an interactive restart
- Concurrent Servers should
  - Fork on accept closing unneeded sockets
  - Define a signal handler for SIGCHLD
  - Develop interprocess communication capability (most simply a file)

Fork and Reparent the Master Process

```c
pid_t pid, sid;

/* Fork off the parent process */
pid = fork();
if (pid < 0)
    { exit(EXIT_FAILURE); }
/* If we got a good PID, exit the parent process */
if (pid > 0)
    { exit(EXIT_SUCCESS); }
/* Create a new SID for the child process */
sid = setsid();
if (sid < 0)
    { /* Log any failure here */ exit(EXIT_FAILURE); }
```

Terminal Disconnect

- To prevent terminal I/O from blocking and to prevent user logout from killing children, the server should be disconnected from the terminal and from the shell group:
  - (void) close(0);
  - (void) close(1);
  - (void) close(2);
  - (void) open("/", O_RDONLY);
  - (void) dup2(0, 1);
  - (void) dup2(0, 2);
- Don’t ask me about the “/”
Umask

- when using open to create a file, the third parameter specifies its protection
- the system default protection for a file and directories is:
  - files: rw-rw-rw- (0666)
  - directories: rwxrwxrwx (0777)
- System default permissions are “masked” to determine the user permissions. Take the binary of the base permissions and perform a logical AND operation on the ones complement representation of the binary umask.
  - A umask of 0000 yields the system default
  - A umask of 0077 yields the following (which is your default):
    - Files: rw-rw-rw- AND rwx------ or rw-------
- The open specifies a set of protections. A bitwise-AND is performed on the file-mode bits and the corresponding bits in the complement of the process's file-mode creation mask.
  - If the umask is set to 0077, 0600 would be ANDed with the value in open
  - If the umask is set to 0400, 0466 would be ANDed with the value in open

Changing File I/O

/* Change the file mode mask */
umask(0);
/* Open any logs */
/* Change the current working directory */
if ((chdir("/")) < 0)
{
  /* Log any failure here */ exit(EXIT_FAILURE);
}

Some Simple Server Signals

- You can imagine two broad classes of signals
  - Errors
  - Utilities
- All concurrent servers need to be prepared to respond to a SIGCHLD
- The utility signals can be used in a variety of ways
  - A convenience function might be written that kills servers you happen to leave hanging around
- Many servers are designed to respond to the SIGHUP as an instruction to reload their config file
  - "The sendmail program recognizes three signals that cause it to perform certain actions. SIGINT causes sendmail to clean up after itself and exit. SIGHUP causes sendmail to re-execute itself (thus restarting and reading its configuration file anew). SIGUSR1 causes sendmail to log its file descriptors and other information."
Some Error Signals

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Default</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGILL</td>
<td>4</td>
<td>Core</td>
<td>Illegal Instruction</td>
</tr>
<tr>
<td>SIGTRAP</td>
<td>5</td>
<td>Core</td>
<td>Trace or Breakpoint Trap</td>
</tr>
<tr>
<td>SIGABRT</td>
<td>6</td>
<td>Core</td>
<td>Abort</td>
</tr>
<tr>
<td>SIGEMT</td>
<td>7</td>
<td>Core</td>
<td>Emulation Trap</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>8</td>
<td>Core</td>
<td>Arithmetic Exception</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>9</td>
<td>Exit</td>
<td>Killed</td>
</tr>
<tr>
<td>SIGBUS</td>
<td>10</td>
<td>Core</td>
<td>Bus Error</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>11</td>
<td>Core</td>
<td>Segmentation Fault</td>
</tr>
<tr>
<td>SIGSYS</td>
<td>12</td>
<td>Core</td>
<td>Bad System Call</td>
</tr>
<tr>
<td>SIGPIPE</td>
<td>13</td>
<td>Exit</td>
<td>Broken Pipe</td>
</tr>
<tr>
<td>SIGXCPU</td>
<td>30</td>
<td>Core</td>
<td>CPU time limit exceeded (see getrlimit(2))</td>
</tr>
</tbody>
</table>

Some Utility Signals

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Default</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGHUP</td>
<td>1</td>
<td>Exit</td>
<td>Hangup (see termio(7))</td>
</tr>
<tr>
<td>SIGINT</td>
<td>2</td>
<td>Exit</td>
<td>Interrupt (see termio(7))</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>3</td>
<td>Core</td>
<td>Quit (see termio(7))</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>14</td>
<td>Exit</td>
<td>Alarm Clock</td>
</tr>
<tr>
<td>SIGINT</td>
<td>16</td>
<td>Core</td>
<td>User Signal 1</td>
</tr>
<tr>
<td>SIGUSR2</td>
<td>17</td>
<td>Exit</td>
<td>User Signal 2</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>18</td>
<td>Ignore</td>
<td>Child Status Changed</td>
</tr>
<tr>
<td>SIGURG</td>
<td>21</td>
<td>Ignore</td>
<td>Urgent Socket Condition</td>
</tr>
<tr>
<td>SIGPOLL</td>
<td>22</td>
<td>Exit</td>
<td>Pollable Event (see streamio(7))</td>
</tr>
<tr>
<td>SIGSTOP</td>
<td>23</td>
<td>Stop</td>
<td>Stopped (signal)</td>
</tr>
<tr>
<td>SIGXCPU</td>
<td>30</td>
<td>Core</td>
<td>CPU time limit exceeded (see getrlimit(2))</td>
</tr>
</tbody>
</table>

Code Fragment for Alarm

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h> /* for alarm */
#include <signal.h> /* for kill, signal, and kill #defines */
#include <sys/wait.h> /* for wait #defines */

void cleanup(int i)
{
    printf("time to go\n");
    exit(EXIT_SUCCESS);
}

int main(int argc, char ** argv)
{
    signal(SIGALRM, cleanup);
    socket(); bind(); listen();
    while (1)
    {
        alarm(600); /* 600 sec = ten minutes */
        accept();
        ……
    }
}
```
Code Fragment for SIGHUP

```c
#include <stdlib.h>
#include <stdio.h>
#include <unixstd.h> /* for alarm */
#include <signal.h> /* for kill, signal, and kill #defines */
#include <sys/wait.h> /* for wait #defines */

void cleanup(int i);

int main(int argc, char ** argv)
{
    signal(SIGHUP, setconfig);
    socket(); bind(); listen();
    while (1)
    {
        accept();
    }
    setconfig(i);
    return;
}

setconfig(int i)
{
    reread configuration file and set parameters
    return;
}
```

Forking on the Accept

- Basically, we fork following the accept results in two processes.
  The parent returns to accept, the child handles the request:

  ```c
  for (;;) {
    alen = sizeof(fsin);
    ssock = accept(msock, (struct sockaddr *) & fsin, &alen);
    if (ssock < 0) { /* the for loop */
        errexit("accept: %s", sys_errlist[errno]);
    }
    switch (fork()) {
    case 0: /* child process */
      close(msock);
      exit(TCPechod(ssock));
    case -1:
      errexit("fork: %s", sys_errlist[errno]);
    default: /* parent */
      break;
    }
  }
  ```

Handling Children(1)

- When a child process exits, a signal is sent to the parent/master process.
- The parent must have code to handle this signal.
- The way this is done is to use one of two signal vector functions
  ```c
  signal(SIGCHLD, mysighandler)
  sigset(SIGCHLD, mysighandler)
  ```
- Signal is reset to the default each time the handler function is called, sigset is not. (There is yet another function called sigvec that can be used.)
Handling Children(2)

- The function calls wait3 to complete
termination of a child that exits (after exit)
- handle multiple calls so server does not
deadlock (wait3 normally blocks) use argument
WNOHANG to specify no blocking

```c
void cleanup(int signal) {
    int status;
    while (wait3(&status, WNOHANG, (struct rusage *)0) > 0);
}
```

File locking

- There are three cooperative file locking mechanisms in Unix
  - flock provides for whole file locking – doesn’t work on NFS mounted
    file systems
  - lockf provides for whole or partial file locking
  - fcntl provides a full set of full and partial locks as well as other features
    – it is not for the weak of heart
- We will use lockf in our file example and flock in our server
  example.
- lockf(fd, operation, offset) Locks or unlocks the portion of the file
  after offset. Allows you to lock just trailing portions of the file, if
  desired. Arguments:
  - F_LOCK Creates a lock. The request blocks until released
  - F_TLOCK Create a lock or return an error immediately
  - F_ULOCK Release our lock
  - F_TEST Test if the file is locked by another process

Using lockf for File Locking (2)

```c
unlink(LF);
sprintf(buf, "touch %s", LF);
system(buf);
if ((fd = open(LF, O_RDWR | O_CREAT, 0666)) == -1) {
    printf("Unable to open for r/w %s", LF);
    exit(-1);
}
lockf(fd, F_LOCK, 0);
Do some file processing
lockf(fd, F_ULOCK, 0);
close(fd);
```
Using Flock for Server Control

- Only one copy of a server should be running at a time
- Flock can be used to insure only one server

```c
if (fopen(path, O_RDWR|O_CREAT, 0640)==0)
{ /* big problem: */ exit (1); }
// couldn't find or create file
if (flock(lf, LOCK_EX| LOCK_NB)<0)
{ /* server exists; */ exit(1); }
// If not exited, could get lock, start up server
```

- Flock is automatically released at system and process crashes
- The lockfile is often used to store the server's process id

```c
sprintf(buf, "%6d
", getpid());
write(lf, buf, strlen(buf));
```