Concurrent Server Design Alternatives

One child per client
- Traditional Unix server:
  - TCP: after call to `accept()`, call `fork()`.
  - UDP: after `recvfrom()`, call `fork()`.
- Each process needs only a few sockets.
- Small requests can be serviced in a small amount of time.
- Parent process needs to clean up after children!!!! (call `wait()`).

One thread per client
- Almost like using fork - call `pthread_create` instead.
- Using threads makes it easier (less overhead) to have sibling processes share information.
- Sharing information must be done carefully (use `pthread_mutex`)

Example

```
main(int argc, char **argv)
{
    listenfd=socket(...)
    Bind(listenfd...)
    Listen(listenfd, LISTENQ);
    Signal(SIGCHLD, sig_child);
    Signal(SIGINT, sig_int);
    For( ; ; ) {
        connfd = Accept(listenfd, ...);
        if ( (pid = Fork())==0) {
            Close(listenfd);
            doit(connfd);
            Close(connfd);
            exit(0);
        }
    }
    Close(connfd);
}
```

```
main(int argc, char **argv)
{
    pthread_t tid;
    listenfd=socket(...)
    Bind(listenfd);
    Listen(listenfd, LISTENQ);
    For( ; ; ) {
        connfd = Accept(listenfd, ...);
        pthread_create(tid, ...);
        doit(connfd);
        Close(connfd);
    }
    static void doit (void *arg)
    {
        Close( (int) arg);
        return NULL;
    }
```

Process version

Thread version
Prefork()’d Server

- Creating a new process for each client is expensive.
- We can create a bunch of processes, each of which can take care of a client.
- Each child process is an iterative server.

Prefork()’d TCP Server

- Initial process creates socket and binds to well known address.
- Process now calls fork() a bunch of times.
- All children call accept().
- The next incoming connection will be handed to one child.

Example

```c
main(int argc, char **argv)
{
    listenfd = socket(…);
    Bind(listenfd, LISTENQ);
    Listen(listenfd, LISTENQ);
    Signal(SIGCHLD, sig_chld);
    Signal(SIGINT, sig_int);
    for(i=0;i<nchildren;i++) {
        if ((pid = Fork())==0) {
            for(;;) {
                connfd = Accept(listenfd,…);
                doit(connfd);
                Close(connfd);
            }
        } 
    }
}
```

Listen()

Preforking

- As the book shows, having too many preforked children can be bad.
- Using dynamic process allocation instead of a hard-coded number of children can avoid problems.
- The parent process just manages the children, doesn’t worry about clients.

Sockets library vs. system call

- A preforked TCP server won’t usually work the way we want if sockets is not part of the kernel:
  - calling accept() is a library call, not an atomic operation.
- We can get around this by making sure only one child calls accept() at a time using some locking scheme.
Prethreaded Server
• Same benefits as preforking.
• Can also have the main thread do all the calls to accept() and hand off each client to an existing thread.

What's the best server design for my application?
• Many factors:
  □ expected number of simultaneous clients.
  □ Transaction size (time to compute or lookup the answer)
  □ Variability in transaction size.
  □ Available system resources (perhaps what resources can be required in order to run the service).

Server Design
• It is important to understand the issues and options.
• Knowledge of queuing theory can be a big help.
• You might need to test a few alternatives to determine the best design.

Socket Options
• Socket Options
  ▪ Posix name/address conversion

  It's important to know about some of these topics, although it might not be apparent how and when to use them.

  Details are in the book - we are just trying to get some idea of what can be done.

Socket Options
• Various attributes that are used to determine the behavior of sockets.
• Setting options tells the OS/Protocol Stack the behavior we want.
• Support for generic options (apply to all sockets) and protocol specific options.
Option types

- Many socket options are Boolean flags indicating whether some feature is enabled (1) or disabled (0).

- Other options are associated with more complex types including int, timeval, in_addr, sockaddr, etc.

Read-Only Socket Options

- Some options are readable only (we can’t set the value).

Setting and Getting option values

getsockopt() gets the current value of a socket option.

setsockopt() is used to set the value of a socket option.

```
#include <sys/socket.h>
```

Get-Only Socket Options

```
int getsockopt( int sockfd, int level, int optname, void *opval, socklen_t *optlen);
```

level specifies whether the option is a general option or a protocol specific option (what level of code should interpret the option).

Socket and IP-layer socket options

```
int setsockopt( int sockfd, int level, int optname, const void *opval, socklen_t optlen);
```

<table>
<thead>
<tr>
<th>Level</th>
<th>Opname</th>
<th>Get</th>
<th>Set</th>
<th>Flag</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOL_SOCKET</td>
<td>SO_ERROR</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>SO_LINGER</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>linger</td>
</tr>
<tr>
<td></td>
<td>SO_KEEPALIVE</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>int</td>
</tr>
<tr>
<td>IPPROTO_IP</td>
<td>IP_HDRINCL</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>IP_TOS</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>int</td>
</tr>
<tr>
<td>IPPROTO_TCP</td>
<td>TCP_MAXSEG</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>TCP_NODELAY</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>int</td>
</tr>
</tbody>
</table>
Example: SO_LINGER

- Specifies how the close function operates for a connection-oriented protocol.

```c
#include <unistd.h>
int close(int socketfd);
```

- Decrease the reference count for the descriptor
- If the reference count is 0:
  - send any data that is already queued to be sent to the other end
  - Normal TCP Connection termination sequence

SO_LINGER usage

- By default, calling `close()` on a TCP socket will return immediately.
- The closing process has no way of knowing whether or not the peer received all data.
- Setting SO_LINGER means the closing process can determine that the peer machine has received the data (but not that the data has been `read()`).

```c
struct linger {
    int l_onoff; /* 0 = off */
    int l_linger; /* time in seconds */
};
```

- Used to control whether and how long a call to close will wait for pending ACKS.
- connection-oriented sockets only.

SO_LINGER vs SO_LINGER

- `l_onoff = 1 & l_linger = 0`
  - TCP aborts the connections when it is closed

- `l_onoff = 1 & l_linger != 0`
  - close return if either:
    - all the data is sent and acked
    - the linger time has expired.

  - Check an example

shutdown

- Starts TCP's normal connection termination sequence, regardless of the reference count

```c
#include <sys/socket.h>
int shutdown(int sockfd, int howto);
```

- howto
  - SHUT_RD: the read half of the connection is closed
  - SHUT_WR: the write half of the connection is closed
  - SHUT_RDWR: the read half and the write half of the connection are both closed

shutdown() vs SO_LINGER

- close returns immediately without waiting at all
- close lingers until the ACK of our FIN is received
- shutdown followed by a `read` waits until we receive the peer's FIN
General Options

- Protocol independent options.
- Handled by the generic socket system code.
- Some general options are supported only by specific types of sockets (SOCK_DGRAM, SOCK_STREAM).

Some Generic Options

- **SO_BROADCAST**
  - Boolean option: enables/disables sending of broadcast messages.
  - Underlying DL layer must support broadcasting!
  - Applies only to SOCK_DGRAM sockets.
  - Prevents applications from inadvertently sending broadcasts (OS looks for this flag when broadcast address is specified).

- **SO_DONTROUTE**
  - Boolean option: enables bypassing of normal routing.
  - Used by routing daemons.

- **SO_ERROR**
  - Integer value option.
  - The value is an error indicator value (similar to `errno`).
  - Readable (get’able) only!
  - Reading (by calling `getsockopt()`) clears any pending error.

- **SO_KEEPALIVE**
  - Boolean option: enabled means that STREAM sockets should send a probe to peer if no data flow for a “long time”.
  - Used by TCP - allows a process to determine whether peer process/host has crashed.
  - Consider what would happen to an open telnet connection without keepalive.
  - Detect half-open connections and terminate them.
SO_RCVBUF
SO_SNDBUF
- Integer values options - change the receive and send buffer sizes.
- Can be used with STREAM and DGRAM sockets.
- With TCP, When should this option be set?
  - this option effects the window size used for flow control - must be established before connection is made.

SO_REUSEADDR
- Boolean option: enables binding to an address (port) that is already in use.
- By default, bind fails when the listening server is trying to bind a port that is part of an existing connection.
- How?

SO_REUSEADDR
- A listening server is started.
- A connection request arrives and a child process is spawned to handle that client.
- The listening server terminates, but the child continues to service the client on the existing connections.
- The listening server is restarted.

IP Options (IPv4): IPPROTO_IP
- IP_HDRINCL: used on raw IP sockets when we want to build the IP header ourselves.
- IP_TOS: allows us to set the “Type-of-service” field in an IP header.
- IP_TTL: allows us to set the “Time-to-live” field in an IP header.

TCP socket options (IPPROTO_TCP)
- TCP_MAXSEG: set the maximum segment size sent by a TCP socket.
another TCP socket option

- **TCP_NODELAY**: can disable TCP’s Nagle algorithm that delays sending small packets if there is unACK’d data pending.

- **TCP_NODELAY** also disables delayed ACKs (TCP ACKs are cumulative).

Socket Options Summary

- This was just an overview
  - there are many details associated with the options described.
  - There are many options that haven’t been described.
  - Our text is one of the best sources of information about socket options.

- Let’s see an example:
  ```c
  getssockopt(fd, IPPROTO_TCP, TCP_MAXSEG, &val, &len);
  ```

Posix Name/Address Conversion

- We’ve seen `gethostbyname` and `gethostbyaddr` - these are protocol dependent.
- Not part of sockets library.
- Posix includes protocol independent functions:
  ```c
  getaddrinfo() getnameinfo()
  ```

**gethostbyname**

```c
struct hostent *gethostbyname(
    const char *hostname);
```

`struct hostent` is defined in `netdb.h`:

```c
#include <netdb.h>
```

```
struct hostent {
    char *h_name;
    char **h_aliases;
    int h_addrtype;
    int h_length;
    char **h_addr_list;
};
```

```
struct hostent {
    char *h_name;
    char **h_aliases;
    int h_addrtype;
    int h_length;
    char **h_addr_list;
};
```
getaddrinfo, getnameinfo
- These functions provide name/address conversions as part of the sockets library.
- In the future it will be important to write code that can run on many protocols (IPV4, IPV6).

Why getaddrinfo()?
- Puts protocol dependence in library (where it belongs).
  - Same code can be used for many protocols (IPV4, IPV6)
  - Re-entrant function - gethostbyname is not!
    - Important to threaded applications.

getaddrinfo()

int getaddrinfo(
  const char *hostname,
  const char *service,
  const struct addrinfo* hints,
  struct addrinfo **result);

getaddrinfo() replaces both
gethostbyname() and getservbyname()

getaddrinfo() parameters
hostname is a hostname or an address string
  (dotted decimal string for IP).
service is a service name or a decimal port number string.

struct addrinfo
struct addrinfo {
  int ai_flags;
  int ai_family;
  int ai_socktype;
  int ai_protocol;
  size_t ai_addrlen;
  char *canonname;
  struct sockaddr *ai_addr;
  struct addrinfo *ai_next;
};

getaddrinfo() hints
hints is an addrinfo * (can be NULL) that can contain:
- ai_flags (AI_PASSIVE, AI_CANONNAME)
- ai_family (AF_XXX)
- ai_socktype (SOCK_XXX)
- ai_protocol (IPPROTO_TCP, etc.)
getaddrinfo() result

result is returned with the address of a pointer to an addrinfo structure that is the head of a linked list.

It is possible to get multiple structures:
- multiple addresses associated with the hostname.
- The service is provided for multiple socket types.

addrinfo usage

- Used in call to socket():
  int socket(int family, int type, int proto);
- Used in call to bind(), connect() or sendto():
  int bind(int sockfd, const struct sockaddr *myaddr, int addrlen);
  int sendto(int sockfd, const void *buf, socklen_t len, int flags, const struct sockaddr *to, socklen_t tolen);

getnameinfo()

int getnameinfo(const struct sockaddr *sockaddr, socklen_t addrlen, char *host, size_t hostlen, char *serv, size_t servlen, int flags);

getnameinfo() looks up a hostname and a service name given a sockaddr.

Assignment & Next time

- Reading:
  - UNP 7.1-7.6, 11.3, 11.4, 11.6, 11.17, 30

- Next Lecture:
  - Daemons and inetd