Lecture 19

Advanced Sockets Programming

CPE 401 / 601
Computer Network Systems

slides are modified from **Dave Hollinger**

Advanced Sockets Programming

- Socket Options
- Posix name/address conversion
- Out-of-Band Data
- Signal Driven I/O
- Deamons
  - inetd

- It's important to know about some of these topics,
  - it might not be apparent how and when to use them
- Details are in the book

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.
- 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()

- used to set the value of a socket option.

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

getsockopt()

```
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

setsockopt()

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

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

SO_DONTROUTE

SO_ERROR

SO_KEEPALIVE

SO_LINGER

SO_RCVBUF,SO_SNDBUF

SO_REUSEADDR

SO_BROADCAST

- enables/disables sending of broadcast msgs
  - *Boolean option*
- 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 only!
  - by calling **`getsockopt()`**
- Reading clears any pending error.

SO_KEEPALIVE

- Enabled means that STREAM sockets should send a *probe* to peer if no data flow for a "long time"
  - *Boolean option*
- 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

SO_LINGER

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

SO_RCVBUF & SO_SNDBUF

- Integer values options
- Change the receive and send buffer sizes.
- Can be used with STREAM and DGRAM sockets.
- With TCP, this option effects the window size used for flow control
  - must be established before connection is made.

SO_REUSEADDR

- Enables binding to an address (port) that is already in use
  - Boolean option
- Used by servers that are transient
  - allows binding a passive socket to a port currently in use (with active sockets) by other processes.
- Can be used to establish separate servers for the same service on different interfaces
  - or different IP addresses on the same interface
- Virtual Web Servers can work this way

IP Options (IPv4)

- IP_HDRINCL: used on raw IP sockets
  - when we want to build the IP header ourselves
- IP_TOS: Sets the "Type-of-service" field in IP header.
- IP_TTL: Sets the "Time-to-live" field in IP header.

TCP socket options

- TCP_KEEPALIVE: set the idle time used when SO_KEEPALIVE is enabled.
- TCP_MAXSEG: set the maximum segment size sent by a TCP socket.
- TCP_NODELAY: can disable TCP's Nagle alg.
  - Nagle algorithm delays sending small packets if there is unACK'd data pending.
  - also disables delayed ACKS

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.

Posix Name/Adress Conversion

- We've seen **`gethostbyname`** and **`gethostbyaddr`**
  - these are protocol dependent
  - Not part of sockets library

- Posix includes protocol *independent* functions:

  `getaddrinfo()` **`getnameinfo()`**

getaddrinfo, getnameinfo

- These functions provide name/address conversions as part of the sockets library
- It is important to write code that can run on many protocols
  - IPV4, IPV6
- **Getaddrinfo** puts protocol dependence in library
  - Same code can be used for many protocols
    - IPV4, IPV6
  - re-entrant function
    - Important to threaded applications.
    - **gethostbyname** is not!

getaddrinfo()

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

- **hostname** is a hostname or an address string
  - dotted decimal string for IP
- **service** is a service name or a decimal port number string

getaddrinfo() hints

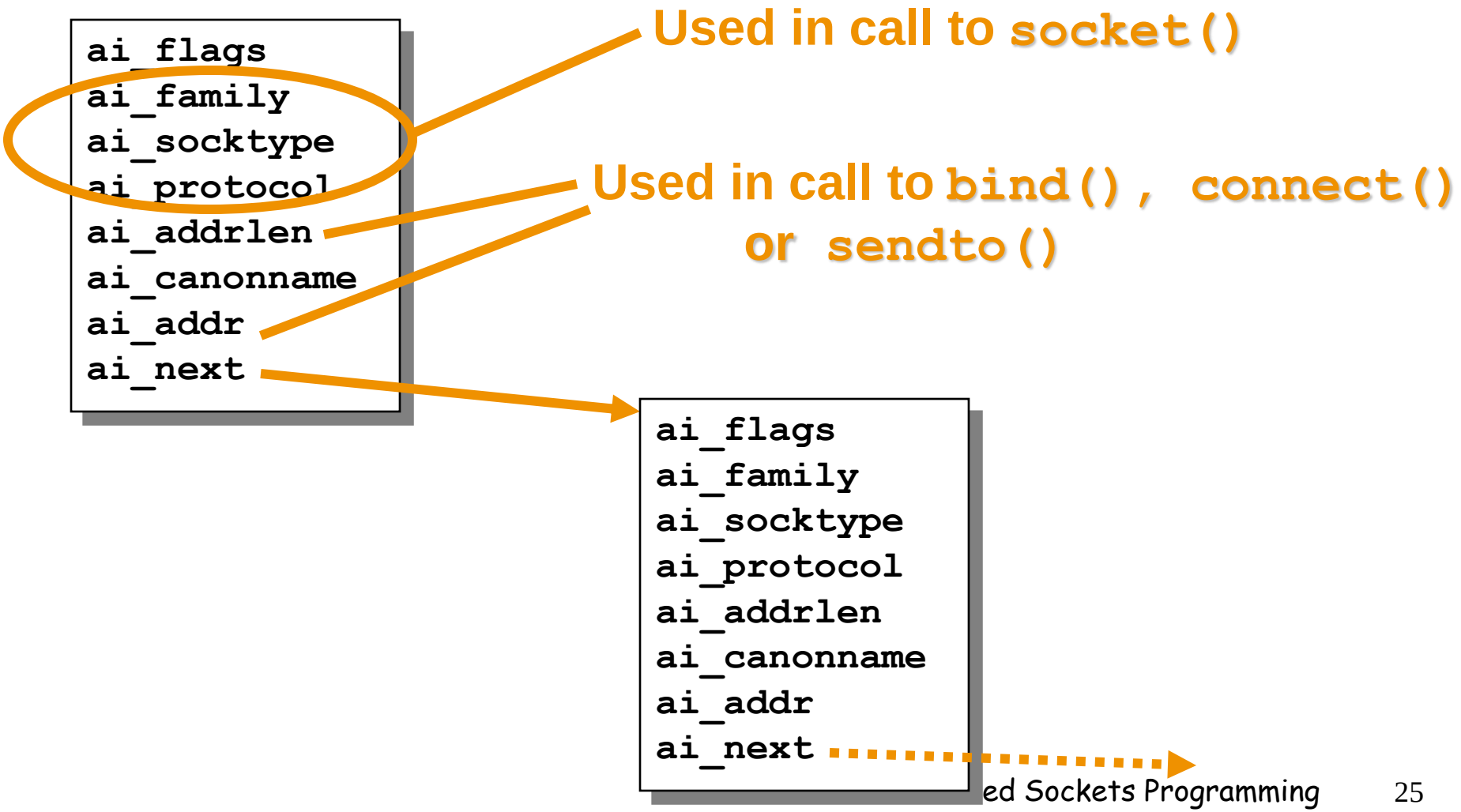
- **hints** is an **addrinfo *** (can be **NULL**) that can contain:
  - **ai_flags** (**AI_PASSIVE** , **AI_CANONNAME**)
  - **ai_family** (**AF_*XXX***)
  - **ai_socktype** (**SOCK_*XXX***)
  - **ai_protocol** (**IPPROTO_TCP, etc.**)

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



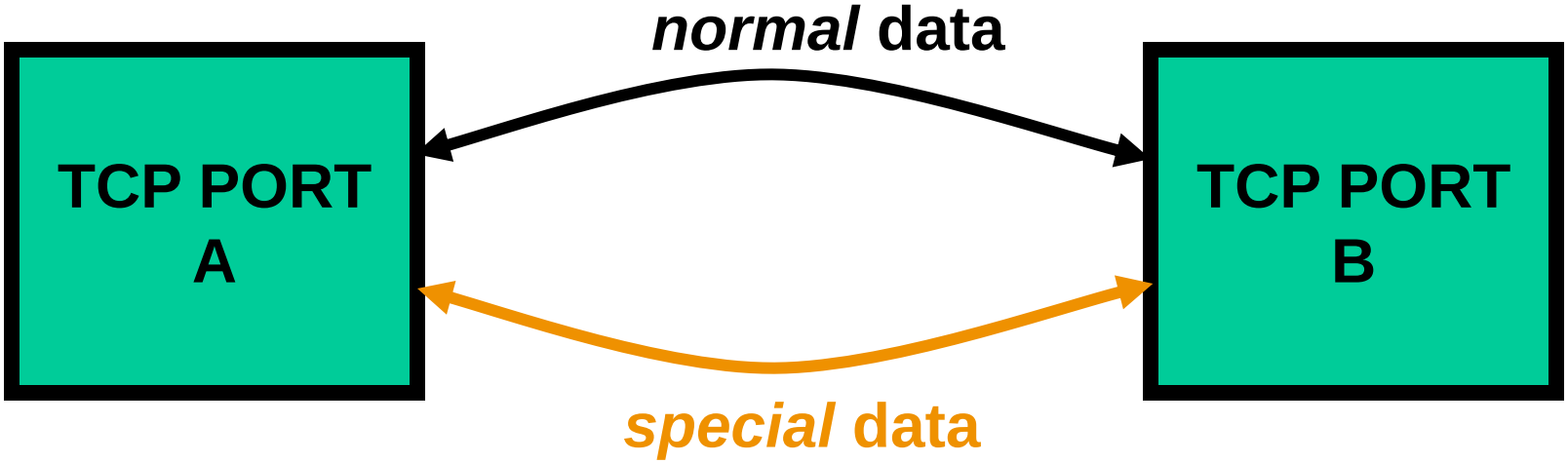
getnameinfo()

```
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`

Out-of-Band Data

- TCP (and other transport layers) provide a mechanism for delivery of "*high priority*" data ahead of "*normal data*".
- We can almost think of this as 2 streams:



TCP OOB Data

- TCP supports something like OOB data using URGENT MODE
  - a bit is set in a TCP segment header
- A TCP segment header field contains an indication of the location of the urgent data in the stream
  - the byte number

Sending OOB Data

```
send(sd,buff,1,MSG_OOB);
```

- Use **send()** to put a single byte of urgent data in a TCP stream.
- The TCP layer adds some segment header info to let the other end know there is some OOB data.

Receiving OOB Data

- TCP layer generates a **`SIGURG`** signal in the receiving process.
- **`select()`** will indicate an exception condition is present
- Reading URG data
  - the data can be read using **`recv()`** with a **`MSG_OOB`** flag set
  - The data can be read *inline* and the receiving process can monitor the *out-of-band-mark* for the connection
    - using **`sockatmark()`**

So what?

- OOB Data might be used:
  - a heartbeat between the client and server to detect early failure
  - A way to communicate an exceptional condition to a peer even when flow control has stopped the sender

Signal Driven I/O

- We can tell the kernel to send us a **SIGIO** signal whenever something happens to a socket descriptor.

- The signal handler must determine what conditions caused the signal and take appropriate action.

Signal Driven UDP

- SIGIO occurs whenever:
  - an incoming datagram arrives.
  - An asynchronous error occurs.
    - Could be ICMP error (unreachable, invalid address, etc).
- Could allow process to handle other tasks and still watch for incoming UDP messages.
- Eg: NTP Network Time Protocol
  - Record timestamp of arrival of UDP datagram

Signal Driven TCP (very rare)

- **SIGIO** occurs whenever:
  - an incoming connection has completed.
  - Disconnect request initiated.
  - Disconnect request completed.
  - Half a connection shutdown.
  - Data has arrived.
  - Data has been sent
    - indicating there is buffer space
  - asynchronous error

Daemons

- A daemon is a process that:
  - runs in the background
  - not associated with any terminal
    - output doesn't end up in another session.
    - terminal generated signals (^C) aren't received.
- Unix systems typically have many daemon processes.
  - Most servers run as a daemon process

Common Daemons

- Web server (httpd)
- Mail server (sendmail)
- SuperServer (inetd)
- System logging (syslogd)
- Print server (lpd)
- router process (routed, gated)

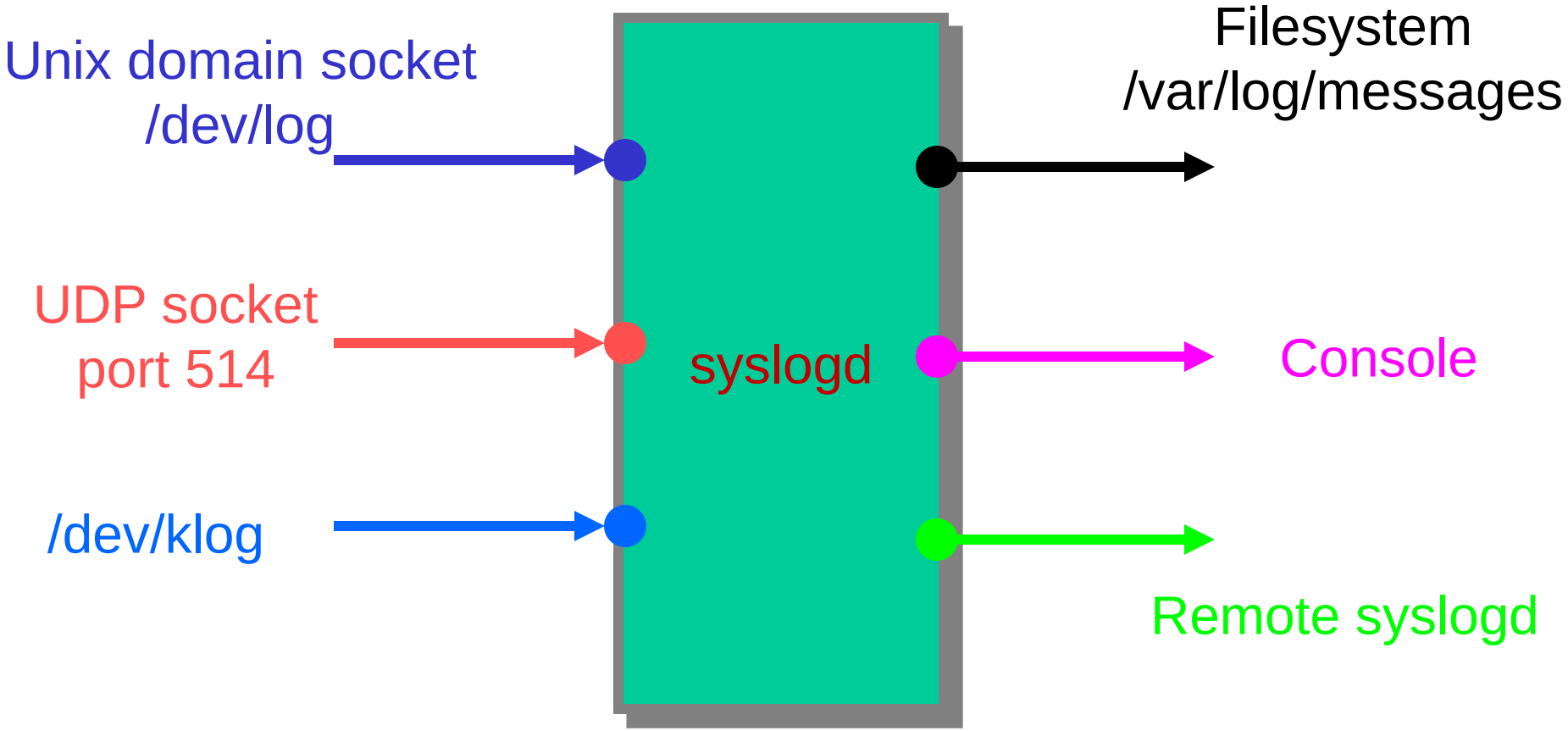
Daemon Output

- No terminal
  - must use something else:
    - file system
    - central logging facility
- Syslog is often used
  - provides central repository for system logging.
- **`syslogd`** daemon provides system logging services to "clients"
- Simple API for "clients"
  - A library provided by O.S.

Centralized Administration

- A system administrator can control logging functions by specifying:
  - where messages should go
  - what kinds of messages are important
  - what can be ignored
- Examples
  - Sysadmin could set LOG_EMERG messages to be sent to the console
  - low priority messages from lpr could be thrown away
  - Medium priority message from the mail server could be saved in a file

syslogd


Unix domain socket
/dev/log
UDP socket
port 514
/dev/klog
syslogd
Filesystem
/var/log/messages
Console
Remote syslogd

Syslog messages

- A server that accepts messages.
- Each message includes a number of fields
  - Message: (*level*,*facility*,*string*)
  - a *level* indicating the importance (8 levels)
    - **`LOG_EMERG`** highest priority
    - **`LOG_DEBUG`** lowest priority
  - a *facility* that indicates the type of process that sent the message:
    - **`LOG_MAIL`, `LOG_AUTH`, `LOG_USER`, `LOG_KERN`, `LOG_LPR`, . . .**
  - A text *string*

/etc/syslog.conf

- Syslogd reads a configuration file that specifies
  - how various messages should be handled
  - where they should go

- The sysadmin controls all logged messages by editing this file.

Sending a message to syslogd

- Standard programming interface provided by `syslog()` function:

```
#include <syslog.h>
void syslog( int priority,
             const char *message,
             . . . );
```

- Works like `printf()`

Syslog client/server

- Clients send messages to local syslogd through a unix domain (datagram) socket
- All the details are handled by **`syslog()`**
- **`syslogd`** sends/receives messages to/from other hosts using UDP.

Daemon initialization

- To force a process to run in the background,
  - just fork() and have the parent exit
- There are a number of ways to disassociate a process from any controlling terminal
  - Call **`setsid()`** and then **`fork()`** again.
- Should close all unnecessary descriptors
  - often including **`stdin, stdout, stderr`**.
- Set up for using syslog
  - Call **`openlog()`**
- Often change working directory.

Too many daemons?

- There can be many servers running as daemons
  - and idle most of the time.
- Much of the startup code is the same for these servers.
- Most of the servers are asleep most of the time,
  - but use up space in the process table.

SuperServer

- Most Unix systems provide a "SuperServer" that solves the problem:
  - executes the startup code required by a bunch of servers.
  - Waits for incoming requests destined for the same bunch of servers.
  - When a request arrives
    - starts up the right server and *gives it the request*.

inetd

- The SuperServer is named **inetd**.
- This single daemon creates multiple sockets and waits for (multiple) incoming requests.
- **inetd** typically uses **select** to watch multiple sockets for input.
- When a request arrives, **inetd** will fork and the child process handles the client.

`inetd` children

- Child process closes all unnecessary sockets
- The child **`dup`**'s the client socket to descriptors 0, 1 and 2 (`stdin,stdout,stderr`)
- The child **`exec`**'s the real server program, which handles the request and exits.
- Servers started by **`inetd`** assume that the socket holding the request is already established
- TCP servers started by **`inetd`** don't call **`accept`**.

/etc/inetd.conf

- **inetd** reads a configuration file that lists all the services it should handle.
- **inetd** creates a socket for each listed service, and adds the socket to a **fd_set** given to **select()** .
- For each service, **inetd** needs to know:
  - the port number and transport protocol
  - wait/nowait flag.
  - login name the process should run as.
  - pathname of real server program.
  - command line arguments to server program

example /etc/inetd.conf

```
# comments start with #
echo      stream  tcp  nowait  root  internal
echo      dgram   udp  wait    root  internal
chargen   stream  tcp  nowait  root  internal
chargen   dgram   udp  wait    root  internal
ftp       stream  tcp  nowait  root  /usr/sbin/ftpd ftpd -l
telnet    stream  tcp  nowait  root  /usr/sbin/telnetd telnetd
finger    stream  tcp  nowait  root  /usr/sbin/fingerd fingerd
# Authentication
auth      stream  tcp  nowait  nobody  /usr/sbin/in.identd in.identd -l -e -o
# TFTP
tftp      dgram   udp  wait    root   /usr/sbin/tftpd tftpd -s /tftpboot
```

wait/nowait

- Specifying WAIT means that **inetd** should not look for new clients for the service until the child (the real server) has terminated.

- TCP servers usually specify **nowait**
  - this means inetd can start multiple copies of the TCP server program
    - providing concurrency!

UDP & wait/nowait

- Most UDP services run with **`inetd`** told to wait until the child server has died.
- What would happen if:
  - **`inetd`** did not wait for a UDP server to die.
  - **`inetd`** gets a time slice before the real server reads the request datagram?
- Some UDP servers hang out for a while,
  - handling multiple clients before exiting.
- **`inetd`** was told to wait
  - it ignores the socket until the UDP server exits

Servers

- Servers that are expected to deal with frequent requests are typically *not* run from inetd: mail, web, NFS.

- Many servers are written so that a command line option can be used to run the server from **`inetd`**.

xinetd

- Some versions of Unix provide a service very similar to **inetd** called **xinetd**.
  - configuration scheme is different
  - basic idea (functionality) is the same...