Network Programming TDC 561

Lecture # 5: Multiservice and Multiprotocols Servers

Dr. Ehab S. Al-Shaer

School of Computer Science & Telecommunication
DePaul University
Chicago, IL

1

select() Review

```
int select( int maxfd,
            fd_set *readset,
            fd_set *writeset,
            fd_set *excepset,
            const struct timeval
  *timeout);
```

maxfd : highest number assigned to a descriptor.
readset: set of descriptors we want to read from.
writeset: set of descriptors we want to write to.
excepset: descriptors to watch for exceptions.
timeout: maximum time select should wait

2

select() Review

- `select()` components (e.g., p. 143)
  - passive (original) mailbox, afds: `FD_SET` and `FD_CLR`
  - active (operational) mailbox, rfds:`FD_ISSET`
- When to use it
  - I/O Multiplexing
  - Non-blocking
  - High-precision timer

3

Multiprotocol Servers

- Provides a single service using TCP or UDP
- Motivation
  - less overhead
  - No need for replication control
  - maintainability
  - extendibility for more protocols

Multiprotocol Servers

- Algorithm
  1. Create a passive UDP socket (usock) and passive TCP socket (tsock), Bind to a port
  2. Use select() to monitor usock and tsock 3.
  3. If tsock is READY
     3.1. Accept connections
     3.2. Read and Write
     3.3. Close if finish
  4. If usock is READY
     4.1. Receive and Send
  5. Go to 2
- Example (*daytimed.c*, P. 150)
  - Is it truly concurrent?
  - No `read` in tsock, and no `close` in usock, WHY?

Multiservice Servers

- A single server that provides multiple services
- Motivation
  - less execution overhead
  - less code
- Algorithm for connectionless servers
  1. Create set of sockets & Bind each to a port
  2. Construct a mapping service table
  3. Use select() to monitor socket
  4. If any socket is READY
     4.1. Dispatch to the proper function and reply
  5. Goto 3

Multiservice Servers

- Algorithm for connection-oriented servers
  1. Create set of sockets & Bind each to a port
  2. Construct a mapping service table
  3. Use select() to monitor socket
  4. If any socket is READY
     - 4.1. Dispatch to the proper function
     - 4.2. Accept the connection
     - 4.3. Read and reply
     - 4.4. Close the accepted socket
  5. Goto 3
- How to make it concurrent
  - fork() before 4.1
  - using select(): add accepted socket to the list
- Using `exec()`

Multiservice Muliprotocol Servers

- Combination of both: super server
- Server Configuration
  - Static
    - reading from a file when starts
  - Dynamic
    - reading from a file when signal is received
    - sending control information via TCP or UNIX sockets
    - Advantages:
      - Flexible for extending, adding new services
      - non-programmer can change handlers services
- superd.c example

Daemons & inetd

- A daemon is a process that:
  - runs in the background
  - not associated with any terminal
- Unix systems typically have many daemon processes.

Common Daemons

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

Daemon Issues

- No terminal - must use something else:
  - file system
  - central logging facility
- 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, tty (see P. 439)
- Daemons should close all unnecessary descriptors (often including stdin, stdout, stderr).

Too many daemons?

- There can be many servers running as daemons - most of them idle most of the time.
- Much of the startup code is the same for all the servers.
- Most of the servers are asleep, 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 of the right server and gives it the request.

`inetd`

- The superserver is named inetd. This single daemon creates a number of sockets and waits for incoming requests.
- When a request arrives, inetd will fork and the child process handles the client.
- inetd moves the accepted socket to zero

`inetd` children

- The 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.

`inetd` based servers

- Servers that are started by inetd assume that the socket to the client is already established (descriptors 0,1 or 2).
- TCP servers started by `inetd` don't call accept, so they must call `getpeername()` if they need to know the address of the client.

/etc/inetd.conf

- Inetd reads a configuration file that lists all the services it should handle. For each service inetd needs to know (P. 170):
  - the port number and protocol
  - wait/nowait flag
  - login name the process should run as
  - pathname of real server program
  - command line arguments to server program

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 before looking for new requests AND inetd get a time slice before the real server reads the request datagram?
- Some UDP servers hang out for a while, handling multiple clients.

Super inetd

- Some versions of inetd have server code to handle simple services such as echo server, daytime server, chargen, …
- 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.

Concurrency in Clients

- Advantages of client concurrency:
  - faster response
  - asynchrony
  - parallelism
- Concurrent client Architecture:
  - Multiprocess connection-oriented (e.g., ass#2: chat)
  - single process (book example)
  - Multithreaded (e.g., Internet browsers) -WAIT!