Network Programming

COSC 1176/1179

Lecture 8

Multiplexing I/O & Assignment 2
Lecture Overview

Introduce assignment 2

Sketch of web server using multiplexed I/O - FSM & select()
Multiplexed Files & Socket I/O

- Files
  - `open()` files with `O_NONBLOCK` option
  - `O_NONBLOCK` permits operations (reads, writes, etc.) in non-blocking mode - `open()` will return immediately with a file handle.
  - If `open()` fails, later use of handle will fail

- Sockets
  - Set sockets to non-blocking - use `fcntl()`
  - Don't just set the `O_NONBLOCK` flag, you might clear other flags. Use:
    
    ```c
    flags = fcntl(sockFD, F_GETFL, 0)
    flags |= O_NONBLOCK
    fcntl(sockFD, F_SETFL, flags)
    ```
  - Always check `fcntl()` result for errors
Select()

- input: 3 sets of sockets in which you are interested
- output: 3 sets containing only sockets where a read/write/... will not block
- returns: number of sockets in sets
- watch out for:
  - 1st argument must be “highest FD in any set” + 1
  - updates sets (readerSet, writerSet, exceptSet) in place
  - may return -1 and errno == EINTR indicating call was interrupted, just retry
  - shouldn't need to use 3rd set (exceptSet), pass NULL
Multiplexed Server

Basic idea for web server with multiplexed I/O

- open files, and set sockets, to be non-blocking
- use select( ) to see which sockets are usable
- perform actions on just those sockets

- FSM is modeled on graph of states, each has single 'blocked' operation at start of state
- create separate instances of FSM machine for each connection
- each instance has own state (local variables)
Main routine

mainLoop():
create listenSocket, assign address, set to listening
while not finished // listening or instances unfinished
build readFDset, including listenSocket
build writeFDset
select( ) to see which sockets are ready for action
if listenSocket in readFDset // can accept a connection
   accept( ) new connection // accept shouldn't block
   create new FSM instance for this connection
for each FSM instance i
   if FSM waiting to read and waitingFD in readFDset
      performNextAction( i ) // next read shouldn't block, so do it
   if FSM waiting to write and waitingFD in writeFDset
      performNextAction( i ) // write to socket shouldn't block...
Instance State

State variables typically required by each FSM instance (for simple web server)

nextAction = ( enumerated list of possible states )
reqSocket // socket connected to client
localFile // FD for file being copied to the client
waitingFD // socket/file required for next action
waitType = (waitRead, waitWrite, notWait) // notWait is error sentinel
reqBuf [] // buffer to accumulate client request
reqBufPtr
rplyBuf [] // buffer for copying file data to client
rplyBufPtr
Sketch of single-threaded web server

```plaintext
handleRequest( con ) do // stAccumRequest
    read(con) → reqBuf
until (socket error or socket closed) or request complete
parseRequest( reqBuf ) → filename
open( filename, O_NONBLOCK ) → localFile
    if open ok, status ← ok, else status ← not-found
rplyBuf ← “HTTP status header”
while anything in rplyBuf // stWriteHeader
    write( con ) ← rplyBuf
    do
        read(localFile) → rplyBuf // stReadData
        while anything in rplyBuf
            write(con) ← rplyBuf // stWriteData
        until end of localFile
    close(con) // stCloseConn
```

Convert single threaded into FSM

- Map your code into steps. Each **starts** with a potentially blocking action, each has ONE potentially **blocking action**
- Convert your nice, structured code into steps
- Invent state “GOTOs” between steps to handle state transitions and errors
performNextAction( ) procedure

**PerformNextAction( )**

waitType = notWait  // to catch state/wait errors

case nextAction:
  stAccumRequest: doAccumRequest( )
  stWriteHeader: doWriteHeader( )
... and so on
... which uses ...

doAccumRequest()
  read( con ) → reqBuf
  if error reading socket | socket is closed
    nextAction = stCloseConn
    waitType = waitWrite
    waitFD = con
  else if requestComplete( reqBuf )
    nextAction = stWriteHeader
    waitType = waitWrite
    waitFD = con
  else // more request to come
    nextAction = stAccumRequest
    waitType = waitRead
    waitFD = con
... and so on ...

doWriteHeader( )

write( con ) ← rplyBuf

if error writing to socket

    nextAction = stCloseConn
    waitType = waitWrite
    waitFD = con

else if buffer empty // all of header written

    nextAction = stReadData
    waitType = waitRead
    waitFD = localFile

else // more header still to write

    nextAction = stWriteHeader
    waitType = waitWrite
    waitFD = con
Notes

- **SO_REUSEADDR socket option**
  - permits server to use the port address,
  - even if that address currently in time-wait state

- **shutdown( )**
  - initiates our end of the 4-way TCP-close protocol
  - indicates to other end that no more data is coming
  - remote close indicates to us that all data has arrived
  - `shutdown( )` your end of a sending socket when you have sent all the data or request
  - when the shutdown socket no longer required, `close( )` it