CSC209 Summer 2015 — Software Tools and Systems Programming

www.cdf.toronto.edu/~csc209h/summer/

Week 11 — July 23, 2015

Peter McCormick
pdm@cs.toronto.edu

Some materials courtesy of Karen Reid
Announcements

• Final exam date has been determined:
  • Tuesday, August 11 (evening)

• http://www.artsci.utoronto.ca/current/exams/reminder

• No tutorial tonight
Agenda

- Last week recap
- `setsockopt`
- I/O multiplexing
Last Week Recap
Web of Hyper Links vs Network of Hops
Web of Hyper Links vs Network of Networks
Web of Hyper Links vs Network of Networks
Web of Hyper Links vs Network of Networks

source

destination
Packets as Onions

IPv4 or IPv6

TCP or UDP or ICMP

Application Payload
TCP: HTTP, SSH, ...
UDP: DNS, low latency, ...
ICMP: ping
Assorted Terminology

- Packet contains **headers** as metadata (source and destination information, ports, protocol specific bits, etc.)
- Packets travel end-to-end by being forwarded over multiple hops by multiple interior **routers**
- Locations identified by numeric IPv4 **addresses**, example 128.100.31.200
- DNS to **resolve** names to addresses
- TCP as a **client/server** model for conversational, connection-oriented interactions
- An unreliable IP network made to appear **reliable** thanks to TCP
- **Request/response** protocols like HTTP
**TCP Server**

- `socket()`
- `bind()`
- `listen()`
- `accept()` (block until connection from client)
- `read()`
- `write()`
- `close()`

**TCP Client**

- `socket()`
- `connect()` (Connection establishment (3-way handshake))
- `write()`
- `read()` (data transfer)
- `close()` (end-of-file notification)
TCP Server

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

TCP Client

- socket()
- connect()
- write()
- read()
- close()

Connection establishment (3-way handshake)

Connection establishment (3-way handshake)

Data transfer

End-of-file notification

Connection establishment (3-way handshake)

Data transfer

End-of-file notification

Connection establishment (3-way handshake)

Data transfer

End-of-file notification
TCP Server

socket()
bind()
listen()
accept()
block until connection from client
read()
write()
close()

TCP Client

socket()
connect()
write()
read()
close()

data transfer
Connection establishment (3-way handshake)
end-of-file notification
Connection-Oriented

Server:
• Create a socket: `socket()`
• Assign a name to a socket: `bind()`
• Establish a queue for connections: `listen()`
• Get a connection from the queue: `accept()`

Client:
• Create a socket: `socket()`
• Initiate a connection: `connect()`
**netcat (nc)**: a command line utility for acting as either a socket client or a socket server

**Use /bin/nc on CDF!**
Run a server and client using netcat

Server (listening):

```bash
wolf:~$ /bin/nc -vlk localhost 209##
```

Client (connecting):

```bash
wolf:~$ /bin/nc -v localhost 209##
```

**NB:** Other students may be using the same port number so if necessary find one that is free!
setsockopt — set options on a socket
setsockopt — set options on a socket

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

- Remember that the socket API’s can be used with more than just TCP/IP stream sockets, so we need to generally remind system calls about what we need!

- Like setting `sin_family` to `AF_INET` in struct `sockaddr_in`'s
setsockopt — set options on a socket

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

- `level`: indicate which part of the networking stack should interpret this option
  - `SOL_SOCKET`: set a socket-level option
  - `IPPROTO_TCP`: set options referring to the TCP layer
setsockopt — set options on a socket

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

- **optname**: a level-dependent option name
- For **SOL_SOCKET** level, a few examples:
  - **SO_REUSEADDR**: allow reuse of local addresses
  - **SO_KEEPALIVE**: attempt to keep a connection open even when nothing is being transmitted
- See `socket(7)` for more!
setsockopt — set options on a socket

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

- Since this is a generic interface, `optval` is an opaque `void` pointer to a variable type that will depend on `level` and `optname`.

- The size of that value must be passed in via `optlen`.

- Many `SOL_SOCKET` socket options expect an integer sized boolean flag.
Enable the \texttt{SO\_REUSEADDR} option on sockfd

```c
int optval = 1; // Boolean true

int rc = setsockopt(
    sockfd,
    \texttt{SOL\_SOCKET},
    \texttt{SO\_REUSEADDR},
    (void *) \&optval,
    sizeof (optval) /* ==\texttt{sizeof (int)} */
);
```

\texttt{setsockopt} — set options on a socket
setsockopt — set options on a socket

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

- See also `getsockopt` to *retrieve* current options
What does `SO_REUSEADDR` do, and why do we care about any of these options?
bind: Address already in use
simpleserver1.c

- socket
- bind
- listen
- repeatedly accept
  - read until end-of-stream
  - close
Enabling `SO_REUSEADDR` fixes this for us
Dealing with Multiple Connections with I/O Multiplexing

Kerrisk 63
Dealing with Multiple Connections

- The `read` system call accepts only a single descriptor, and (for sockets) will block until the other end of that socket connection sends us something.
Blocking I/O Model

- **Application**
  - `read` system call
  - process blocks in a call to `read`

- **Kernel**
  - no data ready
  - data ready
  - copy data from kernel to user
  - copy complete
  - wait for data
  - return OK

- Process data
Dealing with Multiple Connections

• What if you have more than one socket to read from at a time?
dualclient1.c
Dealing with Multiple Connections

• It’s possible to put sockets into a *non-blocking* mode, where the `read` system call returns immediately and signals an error if it would otherwise have to block waiting for data.
Non-blocking I/O Model

application

read

system call

EWONDBLOCK

read

system call

EWONDBLOCK

read

system call

process repeatedly calls read waiting for an OK (polling)

process data

return OK

kernel

no data ready

wait for data

data ready

copy data from kernel to user

copy complete

read system call

read system call

return OK

read system call
Polling Busy-loop Approach:

set all sockets to be non-blocking...

while (1) {
    for each sockfd that is currently connected {
        int rc = read(sockfd, …)
        if (rc == -1 && errno == EWOULDBLOCK):
            continue;

        process read data …
    }
}

rinse & repeat…
It’s also possible to setup the kernel to deliver *signals* to your process when I/O is ready.
Signal Driven I/O Model

- Application:
  - Establish SIGIO handler
  - Signal handler
  - Read
  - Process data
  - Process continues executing

- Kernel:
  - No data ready
  - Copy data
  - Copy complete
  - Wait for data

- System Call:
  - System call
  - Deliver SIGIO

- Signal Action:
  - Sigaction
  - Return
soloserver.c

• socket and setsockopt
• bind
• listen
• repeatedly accept
  • read until end-of-stream
  • close
What happens if two clients simultaneously try to connect to soloserver?

- socket and setsockopt
- bind
- listen
- repeatedly accept
  - read until end-of-stream
  - close
What We Want: a mechanism whereby the kernel tells us which descriptors are available to read *now*, and then read *only* from those
select
I/O Multiplexing Model

application

select

system call

no data ready

kernel

wait for data

data ready

copy data

copy complete

process

blocks

waiting for

one of

many fds

read

system call

return readable

process

blocks

process data

return OK

process data

copy data from kernel to user
select — synchronous I/O multiplexing

```c
int select(int max_fd_plus_1,
           fd_set *readfds,
           fd_set *writefds,
           fd_set *exceptfds,
           struct timeval *timeout);
```

- The most sophisticated system call interface we have seen yet!
select — synchronous I/O multiplexing

```
int select(int max_fd_plus_1,
           fd_set *readfds,
           fd_set *writefds,
           fd_set *exceptfds,
           struct timeval *timeout);
```

- Returns when *(blocks until)* either:
  - The (optionally non-NULL) timeout has expired, *or*
  - When at least one of the file descriptors in one of the sets is *ready* for I/O
- If timeout duration is 0, returns immediately after checking descriptors
Readiness

- Ready to *read* when:
  - There is data in the receive buffer to be read
  - End-of-file state on file descriptor
  - The socket is a listening socket and there is a connection pending
  - A socket error is pending
- Generally most interested in read readiness
Readiness

• Ready to *write* when:
  • There is space available in the write buffer
  • A socket error is pending
  • Useful if you plan on writing a lot of bytes (otherwise connection can block waiting for buffer space to become available)
Readiness

• Ready to handle *exception condition* when:
  
  • TCP out-of-band data
struct timeval

```c
struct timeval {
    long tv_sec;    /* seconds */
    long tv_usec;   /* microseconds */
};
```

- The optional `timeout` specifies how long we are willing to wait for descriptors readiness before returning anyways

- If timeout is `NULL`, block waiting forever (or until a signal is delivered)

- If timeout durations are 0, test descriptors and return immediately
Descriptor Sets

• `fd_set` is a datatype for holding sets of file descriptors

• Since descriptors are non-negative integers, they are typically implemented as a *bit set* (using an array of integers)

  • Bit $N$ is set to 1 iff file descriptor $N$ is in the set
Descriptor Sets

- We indicate our interest in the read/write/exception readiness of a file descriptor by adding it to the appropriate set before the `select` call.

- The `select` call will modify all of the sets, clearing all bits except for the ones corresponding to file descriptors which are now ready.

- After the call, we check each relevant bit of the set to see what is ready.
Before select:

<table>
<thead>
<tr>
<th>fd0</th>
<th>fd1</th>
<th>fd2</th>
<th>fd3</th>
<th>fd4</th>
<th>fd5</th>
<th>fd6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

```
readfds  1  0  0  1  1  0  1  ...  
```

```
maxfd + 1 = 7
```

After select:

<table>
<thead>
<tr>
<th>fd0</th>
<th>fd1</th>
<th>fd2</th>
<th>fd3</th>
<th>fd4</th>
<th>fd5</th>
<th>fd6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

```
readfds  0  0  0  1  0  0  0  ...  
```

select has informed you that FD 3 is ready for reading!
Descriptor Sets

- Implementation details are hidden in the `fd_set` data type.

  - `FD_SETSIZE` is the number of descriptors in the data type.
    - This is a fixed maximum, thus there is a hard limit on the number of descriptors you can select over.

  - `max_fd_plus_1` specifies the number of descriptors to test, so that the call doesn’t have to check all of the fixed maximum of descriptors unnecessarily.
Descriptor Sets

- `void FD_ZERO(fd_set *fdset);`
  - Zero out all bits in the set (removing all descriptors)

- `void FD_SET(int fd, fd_set *fdset);`
  - Add a specific descriptor (set a bit) to the set

- `void FD_CLR(int fd, fd_set *fdset);`
  - Remove a specific descriptor (clear a bit) to the set

- `int FD_ISSET(int fd, fd_set *fdset);`
  - Check whether a specific descriptor is in the set (whether the bit is 1)
Using `select`

1. *Setup* read/write/except sets for your descriptors of interest (noting the largest FD you added)

2. *Setup* an optional timeout value

3. *Call* `select`

4. For each descriptor you initially added:
   - Check whether the FD is still in the given set. If it is, then you know that it is ready for I/O
selectstdin.c
dualclient2.c
multiserver.c
• **select** has some problems
  
  • Fixed number of descriptors in sets
  
  • Large overhead of doing the setup, call and checking return values in order to process ~1 descriptor (doesn’t scale for large servers)
  
• Alternative Unix APIs for multiplexing I/O exist
  
  • **poll**
  
  • **epoll** (Linux specific)
  
  • **kqueue** (FreeBSD and Mac OS X specific)
  
• Windows has its own APIs and idioms
Suggested Exercises

Next Week

- Regularly scheduled office hour on Tuesday
- A4 to be released within next couple of days
Extra Slides
Implementing Bit Sets in C
Arrays of bit strings

- FD_SETSIZE is bigger than 32.

```c
struct bits {
    unsigned int field[N];
};

typedef struct bits Bitstring;
Bitstring a, b;
setzero(&a);
b = a;
a.field[0] = ~0;
```
Setting and unsetting

```c
int set(unsigned int bit, Bitstring *b) {
    int index = bit / 32;
    b->field[index] |= 1 << (bit % 32);
    return 1;
}

int unset(unsigned int bit, Bitstring *b) {
    int index = bit / 32;
    b->field[index] &= ~(1 << (bit % 32));
}
```
Testing and emptying

int ifset(unsigned int bit, Bitstring *b) {
    int index = bit / 32;
    return ( (1 << (bit % 32))
            & b->field[index]);
}

int setzero(Bitstring *b){
    if(memset(b,0, sizeof(Bitstring)) == NULL)
        return 0;
    else
        return 1;
}
Printing

```c
char *intToBinary(unsigned int number) {
    char *binaryString = malloc(32+1);
    int i;
    binaryString[32] = '\0';
    for (i = 31; i >= 0; i--) {
        binaryString[i] = ((number & 1) + '0');
        number = number >> 1;
    }
    return binaryString;
}
```