The World of Select()

So just why am I so hyped on `select()`?

One traditional way to write network servers is to have the main server block on `accept()`, waiting for a connection. Once a connection comes in, the server `fork()`s, the child process handles the connection and the main server is able to service new incoming requests.

With `select()`, instead of having a process for each request, there is usually only one process that "multi-plexes" all requests, servicing each request as much as it can.

So one main advantage of using `select()` is that your server will only require a single process to handle all requests. Thus, your server will not need shared memory or synchronization primitives for different 'tasks' to communicate.

One major disadvantage of using `select()`, is that your server cannot act like there's only one client, like with a `fork()`'ing solution. For example, with a `fork()`'ing solution, after the server `fork()`s, the child process works with the client as if there was only one client in the universe -- the child does not have to worry about new incoming connections or the existence of other sockets. With `select()`, the programming isn't as transparent.

Okay, so how do you use `select()`?

`select()` works by blocking until something happens on a file descriptor (aka a socket). What's 'something'? Data coming in or being able to write to a file descriptor -- you tell `select()` what you want to be woken up by. How do you tell it? You fill up a `fd_set` structure with some macros.

Most `select()`-based servers look pretty much the same:

- Fill up a `fd_set` structure with the file descriptors you want to know when data comes in on.
- Fill up a `fd_set` structure with the file descriptors you want to know when you can write on.
- Call `select()` and block until something happens.
- Once `select()` returns, check to see if any of your file descriptors was the reason you woke up. If so, 'service' that file descriptor in whatever particular way your server needs to (i.e. read in a request for a Web page).
- Repeat this process forever.

Quit with the pseudo-code, show me some real code!

Okay, let's take a look at a sample server (original) included with Vic Metcalfe's Socket Programming FAQ (my comments are in **red**):

```
/*
   PLEASE READ THE FILE NB-APOLOGY!!!!  There are some things you should
   know about this source before you read it.  Thanks.


   Quang Ngo alerted me to a bug where the variable listnum in deal_with_data()
   wasn't being passed in by parameter, thus it was always garbage. I have
   quick-fixed this in the code below. - Spencer (October 12, 1999)
```

```
   Non blocking server demo
   By Vic Metcalfe (vic@acm.org)
   For the unix-socket-faq
*/

#include "sockhelp.h"
#include <ctype.h>
#include <sys/time.h>
#include <fcntl.h>

int sock;            /* The socket file descriptor for our "listening"
                        socket */
int connectlist[5];  /* Array of connected sockets so we know who
                        we are talking to */
fd_set socks;        /* Socket file descriptors we want to wake
                        up for, using select() */
int highsock;        /* Highest #'d file descriptor, needed for select() */

void setnonblocking(sock)
int sock;
{
        int opts;

        opts = fcntl(sock,F_GETFL);
        if (opts < 0) {
                perror("fcntl(F_GETFL)");
                exit(EXIT_FAILURE);
        }
        opts = (opts | O_NONBLOCK);
        if (fcntl(sock,F_SETFL,opts) < 0) {
                perror("fcntl(F_SETFL)");
                exit(EXIT_FAILURE);
        }
        return;
}

void build_select_list() {
        int listnum;         /* Current item in connectlist for for loops */

        /* First put together fd_set for select(), which will
           consist of the sock veriable in case a new connection
           is coming in, plus all the sockets we have already
           accepted. */


        /* FD_ZERO() clears out the fd_set called socks, so that
                it doesn't contain any file descriptors. */

        FD_ZERO(&socks);

        /* FD_SET() adds the file descriptor "sock" to the fd_set,
                so that select() will return if a connection comes in
                on that socket (which means you have to do accept(), etc. */

        FD_SET(sock,&socks);

        /* Loops through all the possible connections and adds
                those sockets to the fd_set */

        for (listnum = 0; listnum < 5; listnum++) {
                if (connectlist[listnum] != 0) {
                        FD_SET(connectlist[listnum],&socks);
                        if (connectlist[listnum] > highsock)
                                highsock = connectlist[listnum];
                }
        }
}

void handle_new_connection() {
```

```
        int listnum;         /* Current item in connectlist for for loops */
        int connection; /* Socket file descriptor for incoming connections */

        /* We have a new connection coming in!  We'll
        try to find a spot for it in connectlist. */
        connection = accept(sock, NULL, NULL);
        if (connection < 0) {
                perror("accept");
                exit(EXIT_FAILURE);
        }
        setnonblocking(connection);
        for (listnum = 0; (listnum < 5) && (connection != -1); listnum ++)
                if (connectlist[listnum] == 0) {
                        printf("\nConnection accepted:   FD=%d; Slot=%d\n",
                                connection,listnum);
                        connectlist[listnum] = connection;
                        connection = -1;
                }
        if (connection != -1) {
                /* No room left in the queue! */
                printf("\nNo room left for new client.\n");
                sock_puts(connection,"Sorry, this server is too busy.  "
                                        Try again later!\r\n");
                close(connection);
        }
}

void deal_with_data(
        int listnum                     /* Current item in connectlist for for loops */
        ) {
        char buffer[80];     /* Buffer for socket reads */
        char *cur_char;      /* Used in processing buffer */

        if (sock_gets(connectlist[listnum],buffer,80) < 0) {
                /* Connection closed, close this end
                   and free up entry in connectlist */
                printf("\nConnection lost: FD=%d;  Slot=%d\n",
                        connectlist[listnum],listnum);
                close(connectlist[listnum]);
                connectlist[listnum] = 0;
        } else {
                /* We got some data, so upper case it
                   and send it back. */
                printf("\nReceived: %s; ",buffer);
                cur_char = buffer;
                while (cur_char[0] != 0) {
                        cur_char[0] = toupper(cur_char[0]);
                        cur_char++;
                }
                sock_puts(connectlist[listnum],buffer);
                sock_puts(connectlist[listnum],"\n");
                printf("responded: %s\n",buffer);
        }
}

void read_socks() {
        int listnum;         /* Current item in connectlist for for loops */

        /* OK, now socks will be set with whatever socket(s)
           are ready for reading.  Lets first check our
           "listening" socket, and then check the sockets
           in connectlist. */

        /* If a client is trying to connect() to our listening
                socket, select() will consider that as the socket
                being 'readable'. Thus, if the listening socket is
                part of the fd_set, we need to accept a new connection. */

        if (FD_ISSET(sock,&socks))
```

```
                handle_new_connection();
        /* Now check connectlist for available data */

        /* Run through our sockets and check to see if anything
                happened with them, if so 'service' them. */

        for (listnum = 0; listnum < 5; listnum++) {
                if (FD_ISSET(connectlist[listnum],&socks))
                        deal_with_data(listnum);
        } /* for (all entries in queue) */
}

int main (argc, argv)
int argc;
char *argv[];
{
        char *ascport;  /* ASCII version of the server port */
        int port;       /* The port number after conversion from ascport */
        struct sockaddr_in server_address; /* bind info structure */
        int reuse_addr = 1;  /* Used so we can re-bind to our port
                                while a previous connection is still
                                in TIME_WAIT state. */
        struct timeval timeout;  /* Timeout for select */
        int readsocks;       /* Number of sockets ready for reading */

        /* Make sure we got a port number as a parameter */
        if (argc < 2) {
                printf("Usage: %s PORT\r\n",argv[0]);
                exit(EXIT_FAILURE);
        }

        /* Obtain a file descriptor for our "listening" socket */
        sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
                perror("socket");
                exit(EXIT_FAILURE);
        }
        /* So that we can re-bind to it without TIME_WAIT problems */
        setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &reuse_addr,
                sizeof(reuse_addr));

        /* Set socket to non-blocking with our setnonblocking routine */
        setnonblocking(sock);

        /* Get the address information, and bind it to the socket */
        ascport = argv[1]; /* Read what the user gave us */
        port = atoport(ascport); /* Use function from sockhelp to
                                    convert to an int */
        memset((char *) &server_address, 0, sizeof(server_address));
        server_address.sin_family = AF_INET;
        server_address.sin_addr.s_addr = htonl(INADDR_ANY);
        server_address.sin_port = port;
        if (bind(sock, (struct sockaddr *) &server_address,
          sizeof(server_address)) < 0 ) {
                perror("bind");
                close(sock);
                exit(EXIT_FAILURE);
        }

        /* Set up queue for incoming connections. */
        listen(sock,5);

        /* Since we start with only one socket, the listening socket,
           it is the highest socket so far. */
        highsock = sock;
        memset((char *) &connectlist, 0, sizeof(connectlist));

        while (1) { /* Main server loop - forever */
                build_select_list();
```

```
                timeout.tv_sec = 1;
                timeout.tv_usec = 0;

                /* The first argument to select is the highest file
                        descriptor value plus 1. In most cases, you can
                        just pass FD_SETSIZE and you'll be fine. */

                /* The second argument to select() is the address of
                        the fd_set that contains sockets we're waiting
                        to be readable (including the listening socket). */

                /* The third parameter is an fd_set that you want to
                        know if you can write on -- this example doesn't
                        use it, so it passes 0, or NULL. The fourth parameter
                        is sockets you're waiting for out-of-band data for,
                        which usually, you're not. */

                /* The last parameter to select() is a time-out of how
                        long select() should block. If you want to wait forever
                        until something happens on a socket, you'll probably
                        want to pass NULL. */

                readsocks = select(highsock+1, &socks, (fd_set *) 0,
                  (fd_set *) 0, &timeout);

                /* select() returns the number of sockets that had
                        things going on with them -- i.e. they're readable. */

                /* Once select() returns, the original fd_set has been
                        modified so it now reflects the state of why select()
                        woke up. i.e. If file descriptor 4 was originally in
                        the fd_set, and then it became readable, the fd_set
                        contains file descriptor 4 in it. */

                if (readsocks < 0) {
                        perror("select");
                        exit(EXIT_FAILURE);
                }
                if (readsocks == 0) {
                        /* Nothing ready to read, just show that
                           we're alive */
                        printf(".");
                        fflush(stdout);
                } else
                        read_socks();
        } /* while(1) */
} /* main */
```

Okay, so maybe that wasn't the best example...

Got some suggests? Corrections? Please let me know.

In the mean time, here's some references to other sources of information about `select()`:

- Socket FAQ Question about `select()`
  Straight from the excellent socket FAQ.
- Unix Programming FAQ Question about `select()`
  Straight from the *other* excellent FAQ.
- Unix Socket Programming FAQ Examples
  The above `nbserver.c` sample code and more socket stuff.
- thttpd - tiny/turbo/throttling HTTP server
  Nifty little single-threaded, non-`fork()`'ing, `select()` based Web server.
- BOA

Another single-threaded, `select()` based Web server.

[Back to Spencer's Socket Site](#)