Multiplexing I/O

CSC209H5: Software Tools & Systems Programming

Robert (Rupert) Wu
rupert.wu@utoronto.ca

Department of Computer Science
University of Toronto

March 27, 2023

Lesson Plan

1. Reading From Multiple Sources
2. I/O Models
3. Multiplexing I/O with `select()`
4. More on Multiplexing
5. Practice with Multiplexing
6. Multiplexing I/O Alternatives

Acknowledgements

Some material was borrowed from Andi Bergen and Karen Reid.

Section 1

I/O Models for Multiple Sources

Assume that a process `p0` has any two file descriptors open for reading (e.g., from a socket, regular file, pipe). Keep in mind that `read()` is blocking.

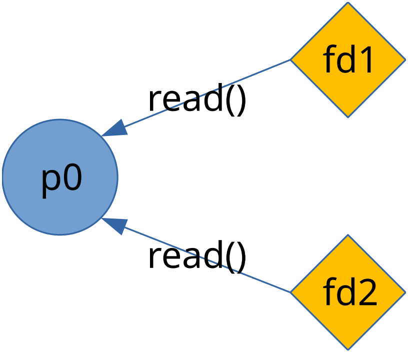

Figure 1: a process `p0` reading from two file descriptors `fd1`, `fd2`

If `p0` reads from `fdX`, it will block until `fdX` has data ready to read. But what if the other `fdY` already has data available to be read?

Reading From Multiple Sources: Another Perspective

Another way to view the problem is the following.

```
while true
    accept a new connection
    for each existing connection
        read
        write
```

Which of the system calls might block indefinitely?

- `read()` and `accept()`

So what happens if there is only one connection?

- The program will stall until the one `read()` has available data.

And what if there are multiple connections?

- The program will stall on the first `read()` that doesn't have available data and not be able to operate on existing connections or accept new ones.

`p0` can `fork` one child process per file descriptor to be read from; each child calls `read` on one file descriptor and communicates data to parent over a pipe.

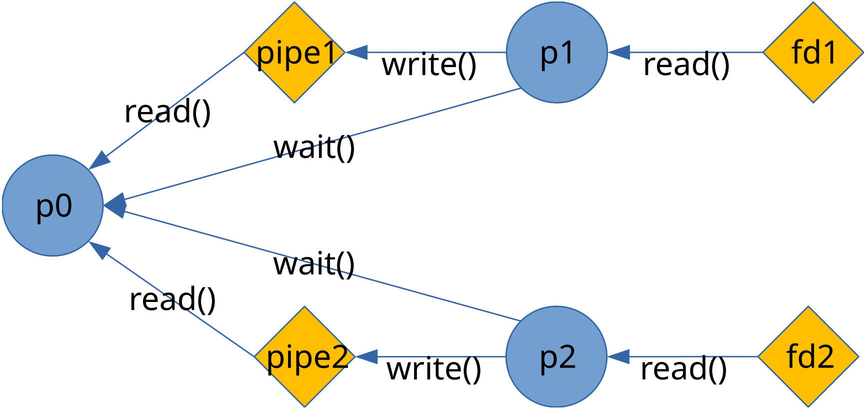

Figure 2: using a new server-side `pipe()`+`fork()` to handle each `read()`

Reading From Multiple Sources: Using `fork()` and Sockets

- It is common for server software to `fork()` a new process for each client that connects: SSH does exactly that.
- *Performance* benefit: Solves the issue of blocking `read()` calls that we just discussed.
- *Security* benefit: Each process has its own memory space, making it less likely for there to be a bug that allows one user to read confidential information that belongs to another user
- Drawback: Each process takes up memory.

Section 2

I/O Models

I/O Models

Let's review some different I/O models...

- blocking I/O
- nonblocking I/O
- signal driven I/O (`SIGIO`)
- I/O multiplexing (`select` and `poll`)
- asynchronous I/O (the POSIX `aio_functions`)

Most of the time, there are two distinct phases of input operation.

1. Waiting for data to be ready (arriving from the network to the kernel's buffer).
2. Copying the data from the kernel to the process (from kernel's buffer to application's buffer).

Source

www.masterraghu.com/subjects/np/introduction/unix_network_programming_v1.3/ch06le

I/O Models: Blocking

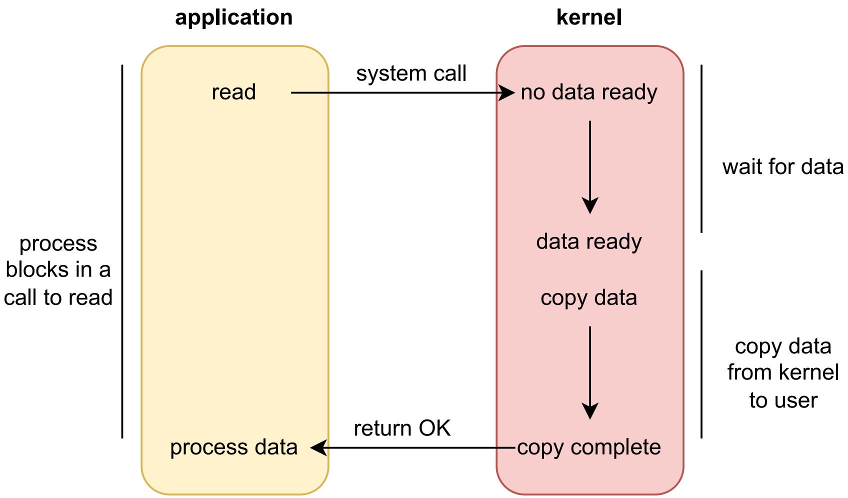

Figure 3: a blocking I/O model; based on Haviland 7.1.6

I/O Models: Non-Blocking

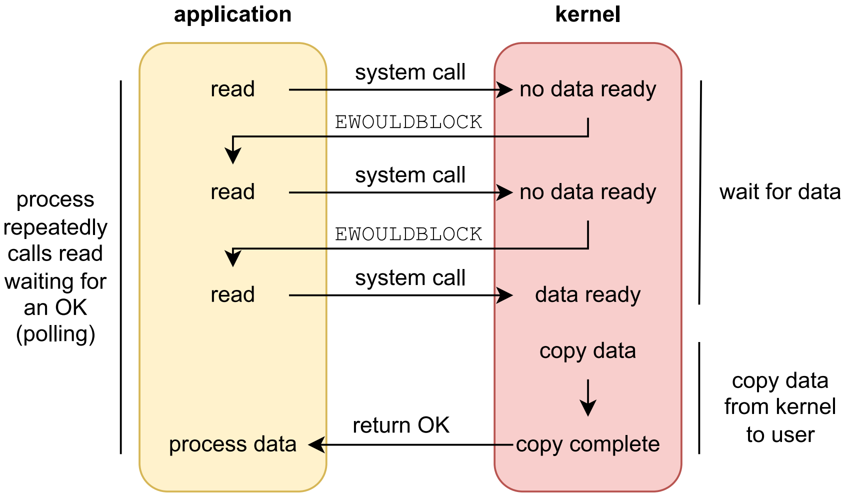

Figure 4: a non-blocking I/O model; based on Haviland 7.1.6

I/O Models: Signal-Driven

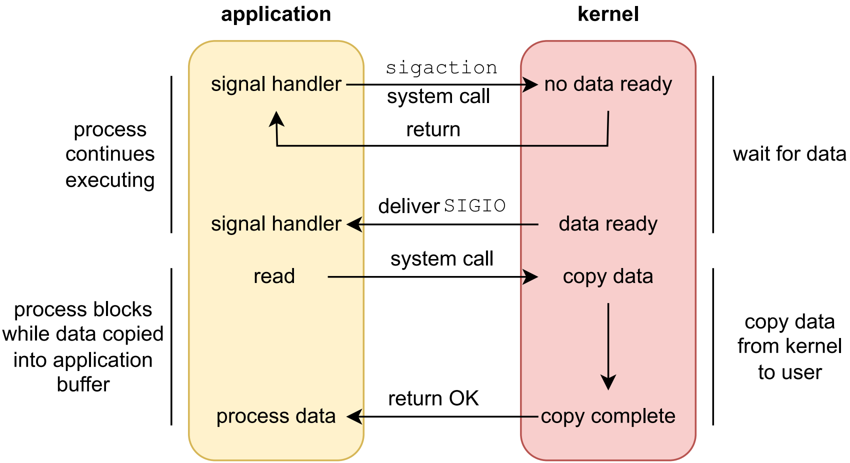

Figure 5: a signal-driven I/O model; based on Haviland 7.1.6

I/O Models: Multiplexing

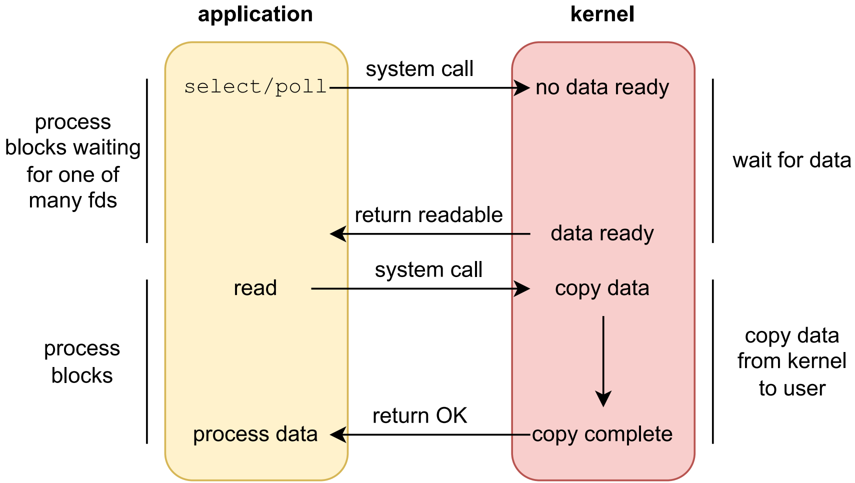

Figure 6: a multiplexiing I/O model; based on Haviland 7.1.6

I/O Models: Asynchronous

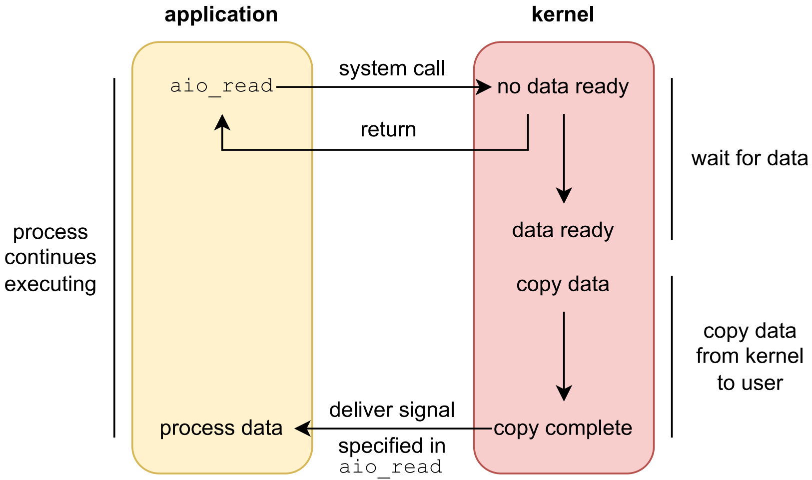

Figure 7: an async I/O model; based on Haviland 7.1.6

I/O Models: Comparison

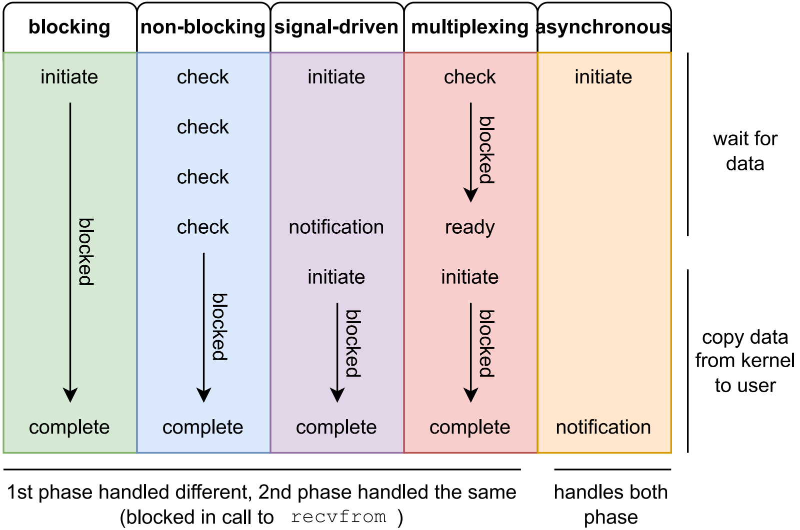

Figure 8: how the models compare in control flow; based on Haviland 7.1.6

Section 3

Multiplexing I/O with `select()`

Multiplexing I/O with select()

select() monitors several file descriptors (FDs) simultaneously, without needing to fork() children processors to handle them.

```
#include "sys/select.h"
int select(int nfds, fd_set *restrict readfds, fd_set *restrict writefds,
    fd_set *restrict exceptfds, struct timeval *restrict timeout);
```

- Arguments include the upper bound of FDs (nfds), sets of FDs to monitor (readfds, writefds, exceptfds) and a timeout.

This function **blocks** until some monitored FD is "ready" or when timeout exceeded. Then, it returns the number of FDs that are ready, or -1 on error.

Motivation

The bottom line is that we **never** want to block on any calls to read() or accept().

- Otherwise, we risk the possibility of waiting forever, even when there might be data ready to be read from other FDs.
- Instead, we write our client/server programs to block **only** on select().

Multiplexing I/O with select(): Parameters

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

`select()` takes several arguments...

1. `nfds`: The (exclusive) upper bound on the FDs that're monitored.
   - Set it to just above the highest-numbered FD of interest
   - If you're interested in FDs 3, 9, 50, let `nfds` be 51.
2. `readfds`: set of FDs to monitor for *reading*.
3. `writefds`: set of FDs to monitor for *writing*.
4. `exceptfds`: set of FDs to monitor for *exceptions*.
5. `timeout`: how long we're willing to wait for a given FD to be ready.

More About Exceptions

www.gnu.org/software/libc/manual/html_node/Out_002dof_002dBand-Data.html

Multiplexing I/O with `select()`: Readiness

`select()` **blocks** until some monitored FD is "ready". What does that mean?

Ready for Reading (`readfds`)

a FD is ready for *reading* when `read()` can be called *once* without blocking; this happens on socket errors, or under the following conditions:

- there is data in the receive buffer to be read;
- end-of-file state (EOF) is detected on the FD;
- the socket is a listening socket and there's a pending connection.

Most of the time, we are interested in monitoring FDs in `readfds`.

Ready for Writing (`writefds`)

FD is ready for *writing* if there's space in the write buffer or a pending socket error.

Ready for Exceptions (`exceptfds`)

In the TCP protocol, some data is *urgent*, so the receiver should process it immediately outside the buffer stream. a FD can have an exception condition when it has this "out-of-band" data.

Multiplexing I/O with `select()`: Readiness is Level- or Edge-Triggered

When is an FD ready? In the following scenarios:

1. *Level-triggered*: when an operation (e.g. `read`) won't block; or
2. *Edge-triggered*: when there is new action on the FD since the last check.

`select()` is level-triggered: if you don't read everything, `select()` will keep telling you that the FD is ready

Multiplexing I/O with select(): Descriptor Sets

```
#include "sys/select.h"
#define __FDS_BITS(set) ((set)->__fds_bits)
typedef struct { // for POSIX/Berkeley/BSD sockets
    __fd_mask __fds_bits[__FD_SETSIZE / __NFDBITS];
} fd_set;
```

File descriptor sets (`fd_set`'s) are similar to signal sets but typically implemented as an array of integers where each bit corresponds to a FD.

- Implementation is hidden in the `fd_set` data type (e.g. Windows doesn't use integer arrays).
- `FD_SETSIZE` is the number of FDs in the data type.
- The argument `nfds` specifies the number of FDs (counting from `0`) to test.

You can use these macro functions to manipulate/operate on `fd_set`'s:

```
#include "sys/select.h"
void FD_CLR(int fd, fd_set *set);    // remove fd from set
int  FD_ISSET(int fd, fd_set *set);  // check if fd in set
void FD_SET(int fd, fd_set *set);    // add fd to set
void FD_ZERO(fd_set *set);           // empty/zero out set
```

Multiplexing I/O with `select()`: Timeout

`select()` returns when (and blocks until) some monitored FD is "ready" or when `timeout` exceeded. The `timeout` specifies how long we're willing to wait for a FD to become ready.

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

Depending on a value of the `timeval` struct, `select()` might wait.

- If timeout is `0`, test and return immediately.
- If timeout is `NULL`, wait forever (or until we catch a signal).
- Otherwise wait up to specified timeout.

Multiplexing I/O with select(): A Variant

There exists a variant of `select()` called `pselect()`.

```
#define _POSIX_C_SOURCE 200112L
#include <sys/select.h>
int pselect(int nmsgsfds, fd_set *__restrict__ readlist,
            fd_set *__restrict__ writelist, fd_set *__restrict__ exceptlist,
            const struct timespec *__restrict__ timeout,
            const sigset *__restrict__ sigmask);
```

The key differences are as follows:

- `select()` takes a `timeval` while `pselect()` takes `timespec`.
  - What's the difference? *Their second members (`tv_usec`, `tv_nsec`) are in microseconds and nanoseconds, respectively.*
- `pselect()` adds a `sigmask` argument. If `sigmask` is a null pointer, this is equivalent to `select()`.
  - What does this do? *It's a bit vector indicating which signals to ignore.*
- Unlikely `select()`, `pselect()` cannot modify `timeout` upon success.
  - Why would you want to? *You can modify `timeout` to indicate remaining time.*

Source: www.ibm.com/docs/en/zos/2.2.0

Arnold provided some examples:

- `[www]/lectures/src/select/selectExample0.c`
- `[www]/lectures/src/select/selectExample1.c`
- `[www]/lectures/src/select/selectExample2.c`
- `[www]/lectures/src/select/muffinman.c`

`[www]` = `mcs.utm.utoronto.ca/~209/23s`

Chat Room Demo

And of course, chat rooms were a motivating use case..

- `mcs.utm.utoronto.ca/~209/23s/lectures/src/select/charserver.c`
- `mcs.utm.utoronto.ca/~209/23s/lectures/src/select/charserver2.c`
- `github.com/kirintwn/socket-chat-room`

Section 4

More On Multiplexing

Reading From Clients

When a server does a `read()`, it is not guaranteed to receive a complete line or all of the desired bytes. For example:

- The client could be sending each character separately.
- The client could send data that gets split over several segments.

Want to operate only on full lines? The server must keep each partial line in a buffer until it gets the newline from the client.

Buffering for Full Lines

The following code assumes there's at most one line in the buffer.

```
struct client {
    int fd;
    char buf[300];
    int inbuf;
    struct client *next;
};
```

The server should keep a buffer for each client, and keep track of the number of bytes in each buffer following the previous message.

Buffering for Full Lines...

Read bytes, check for errors, and null-terminate the string.

```
void myread(struct client *p) {
    int room = sizeof(p->buf) - p->inbuf;
    if (room <= 1) { ... } // clean up this client: buffer full

    char *startbuf = p->buf + p->inbuf;
    char *tok, *cr, *lf;
    int crlf;

    int len = read(p->fd, startbuf, room - 1);
    if (len <= 0) { ... } // clean up this client: eof or error

    p->inbuf += len;
    p->buf[p->inbuf] = '\0';
    ...
}
```

Making sure to start at the point up to which the buffer's filled.

Buffering for Full Lines...

If a full line exists, process it and shift it out of the buffer.

```
void myread(struct client *p) {
    ...
    lf = strchr(p->buf, '\n');
    cr = strchr(p->buf, '\r');
    if (!lf && !cr) return; // no complete line

    tok = strtok(p->buf, "\r\n");
    if (tok) { ... } // use tok (complete string)

    // compute how many bytes we're removing
    if (!lf) crlf = cr - p->buf;
    else if (!cr) crlf = lf - p->buf;
    else crlf = ((lf > cr) ? lf : cr) - p->buf;
    crlf++; // include the CRLF

    p->inbuf -= crlf; // shift the remainder towards the head
    memmove(p->buf, p->buf + crlf, p->inbuf);
}
```

Suppose you're writing to a broken pipe/socket generates a `SIGPIPE`. By default, most signals (including `sigpipe`) will terminate your program. Here's how you can protect against `sigpipe`:

```
/*
 * Turn off SIGPIPE: write() to a socket that
 * is closed on the other end will return -1
 * with errno set to EPIPE, instead of generating
 * a SIGPIPE signal that terminates the process.
 */
if (signal(SIGPIPE, SIG_IGN) == SIG_ERR) {
    perror("signal");
    exit(1);
}
```

Non-Blocking Reads

You can change the behaviour of `read()` so that it returns `-1` and sets `errno` to `EAGAIN` if no data is available.

- In this mode, `read()` will never block.
- Downside is that it will lead to inefficient code, e.g., using an infinite loop that repeatedly calls `read()`.
  - Remember, `read()` will return **immediately** in non-blocking mode, so you will be calling it **many** times per second.

Non-Blocking Reads: Sample Code

```
char buf[1024];
ssize_t bytesread;
/* set O_NONBLOCK flags on fd1 and fd2 */
if (fcntl(fd1, F_SETFL, O_NONBLOCK) == -1) perror("fcntl"); exit(1);
if (fcntl(fd2, F_SETFL, O_NONBLOCK) == -1) perror("fcntl"); exit(1);

for (;;) {
    bytesread = read(fd1, buf, sizeof(buf));
    if ((bytesread == -1) && (errno != EAGAIN))
        return; // real error
    else if (bytesread > 0)
        doSomething(buf, bytesread);

    bytesread = read(fd2, buf, sizeof(buf));
    if ((bytesread == -1) && (errno != EAGAIN))
        return; // real error
    else if (bytesread > 0)
        doSomething(buf, bytesread);
}
```

Section 5

Practice with Multiplexing

Practice with Multiplexing: Descriptor Sets

You can use these macro functions to manipulate/operate on `fd_set`'s:

```
#include "sys/select.h"
void FD_CLR(int fd, fd_set *set);     // remove fd from set
int  FD_ISSET(int fd, fd_set *set);   // check if fd in set
void FD_SET(int fd, fd_set *set);     // add fd to set
void FD_ZERO(fd_set *set);            // empty/zero out set
```

Suppose you have a server `S` and two clients `c1`, `c2`.

1. `S` should read on FDs 4 and 6 from `c1` and `c2`, respectively.
2. FD 3 on `S` should also be flagged for everything.
3. No other FDs should be marked as ready.
4. *Some time after* calling `select()`, let's also mark FD 3 as not ready at all.

Try drawing a graph between `S`, `c1`, `c2` and writing the representative bit vectors for the sets right before and after `select()`. Then use the macro function calls to perform the above operations. Start here:

```
fd_set *readfds, *writefds, *exceptfds;
/* some code */
```

Practice with Multiplexing: Descriptor Sets

Suppose you have a server `S` and two clients `c1`, `c2`.

1. `S` should read on FDs 4 and 6 from `c1` and `c2`, respectively.
2. FD 3 on `S` should also be flagged for everything.
3. No other FDs should be marked as ready.
4. *Some time after* calling `select()`, let's also mark FD 3 as not ready at all.

The representative bit vectors immediately before/after `select()`:

- `readfds`: 00011010... → 00010000...
- `writefds/exceptfds`: 00010000... (don't change)

And the code would look like this:

```
fd_set *readfds, *writefds, *exceptfds; /* some code */
FD_ZERO(readfs); FD_ZERO(writefds); FD_ZERO(exceptfds); // why clear?
FD_SET(4, readfds); FD_SET(6, readfds);
FD_SET(3, readfds); FD_SET(3, writefds); FD_SET(3, exceptfds);
select(7, readfds, writefds, exceptfds, NULL); // why 7?
FD_ZERO(writefds);
```

What does a `timeout` of value `NULL` do?

Practice with Multiplexing: Office Hours Analogy

Consider the following man page of an imaginary system call `office_hours()`.

```
OFFICE_HOURS(2)              BSD System Calls Manual              OFFICE_HOURS(2)

NAME: IS_CLR, IS_ISSET, IS_SET, IS_ZERO, office_hours

SYNOPSIS
    void IS_CLR(is, is_set *isset); int IS_ISSET(is, is_set *isset);
    void IS_SET(is, is_set *isset); void IS_ZERO(is_set *isset);
    int office_hours(is_set *instr, struct timeval window);

DESCRIPTION: office_hours() examines the schedules for the instructors in the instructor
set instrs to see which have office hours scheduled within the given window from the
current time. office_hours() replaces the instructor set with the subset of instructors
who have office hours in the given window.

RETURN VALUE: office_hours() returns the number of instructors from the is_set who have
office hours in the window, or -1 if an error occurs. If office_hours() returns with an
error, the descriptor sets will be unmodified and the global variable errno will be set
to indicate the error.
```

`is_set`, `office_hours()` and the (macro) functions are very similar analogs to `fd_set`, `select()`, and those in `sys/select.h`.

Practice with Multiplexing: Office Hours Analogy

`is_set`, `office_hours()` and the (macro) functions are very similar analogs to `fd_set`, `select()`, and those in `sys/select.h`. Suppose that (like FDs) instructors are represented by small integers and there are instructors defined as follows:

```
#define ANDREW 1
#define ARNOLD 2
#define BAHAR 3
#define RENATO 4
#define RUPERT 5
#define TINGTING 6
```

You want help in CSC258 in the next 24 hours. Finish the program on the next slide, so that it calls `office_hours()` and then prints either the message "Andrew has office hours" or the message "Renato has office hours" or both messages as appropriate.

- As an additional exercise, properly check for errors on a system call by writing the code to give the conventional behaviour if `office_hours()` fails.

Practice with Multiplexing: Office Hours Analogy (Solution)

```c
#include "stdio.h"
#include "sys/select.h"
int main() {
    struct timeval window = {24 * 60 * 60, 0};

    // set up first argument to office_hours()
    is_set instrs; IS_ZERO(&instrs);
    IS_SET(ANDREW, &instrs);
    IS_SET(RENATO, &instrs);

    // call office_hours()
    if (office_hours(&instrs, window) == -1) {
        perror("office_hours");
        exit(1);
    }

    // print the appropriate message
    if (IS_ISSET(RENATO, &instrs)) printf("Renato has office hours\n");
    if (IS_ISSET(ANDREW, &instrs)) printf("Andrew has office hours\n");
    return 0;
}
```

Practice with Multiplexing: Other Questions

Suppose your server is written to block on both `read()` and `write()` calls.

1. Recalling last lecture, how might a `write()` call block?
2. Why might you write it this way? How is it useful?
3. How do we solve this?

Practice with Multiplexing: Other Questions

Suppose your server is written to block on both `read()` and `write()` calls.

1. Recalling last lecture, how might a `write()` call block?
   - *TCP buffers if a server is sending data faster than a client can handle.*
2. Why might you write it this way? How is it useful?
   - *Example: your server is responsible for selling tickets and clients can buy as many as possible. These tickets could be unique and involve some very complex hash composed of a large number of bytes.*
   - *Example: your server `S1` ingests local data and shards it across multiple remote clients. The datagrams sent include timestamps of shards' transmission times.*
3. How do we solve this? *Use `select()` for both reads and writes.*
4. Expanding on the latter example in Question 2, how might you write a second server downstream that concurrently reads from the clients and reconstructs the original data in order?
5. Same as Question 4, but reconstructing in reverse order.

Practice with Multiplexing: Other Questions

Scenario: your server ingests local data and shards it across multiple remote clients. The datagrams sent include timestamps and precedence of shards.

How might you write a second server `S2` downstream that concurrently reads from the clients and reconstructs the original data in order? Here are some potential solutions:

1. Use blocking I/O and call `read()` on the clients sequentially.
   - But this is problematic if the clients have their own timeouts i.e. they shutdown after their FDs have been open/ready for a while.
2. Instead, use multiplexing I/O.
   1. Introduce an ordering index (timestamps were of transmission time, not precedence) into the datagrams.
   2. Call `select()`+`read()` to write into an ordered buffer.
   3. How do you know how big the buffer should be?
      - Use dynamic memory for and `memmove()` on the buffer. But this wouldn't work in the reverse order case...
      - Instead, have `S1` transmit the sizes of the shards to `S2`. The value represnting the lenth of a shard doesn't grow in memory footprint.

Section 6

Multiplexing I/O Alternatives

Multiplexing I/O Alternatives: poll()/ppoll()

Although portable, select() has some performance limitations, and can only monitor at most FD_SETSIZE (1024, on Linux) FDs. Consider poll()/ppoll().

```
#include "poll.h"
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
int ppoll(struct pollfd *fds, nfds_t nfds, // analog to pselect()
        const struct timespec *tmo_p, const sigset_t *sigmask);
```

select() iterates over the array of all possible FDs (up to nfds) while only the active FDs are polled by poll(). By definition, $\mathcal{O}(\texttt{poll}) \leq \mathcal{O}(\texttt{select})$ and in most cases $\mathcal{O}(\texttt{poll}) \ll \mathcal{O}(\texttt{select})$ because most FDs between 0 and nfds are inactive.

Portability & Modernity

Nowadays, poll() is widespread enough that it's considered the new portable standard. In other iterations of CSC209, poll() might be taught instead.

Can we do better?

Some OS-specific system calls perform even better but are less portable. Conditional directives #ifdef/#else/#endif can sometimes offer a workaround.

Multiplexing Alternatives: More Efficient, Less Portable (epoll on Linux)

epoll() is an API that performs a similar task to poll() but is much more scalable ($\mathcal{O}(1)$). It's both level- and edge-triggered.

```
#include "sys/epoll.h"
int epoll_create(int size);
int epoll_create1(int flags);
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event *events,
              int maxevents, int timeout);
int epoll_pwait(int epfd, struct epoll_event *events,
              int maxevents, int timeout,
              const sigset_t *sigmask);
int epoll_pwait2(int epfd, struct epoll_event *events,
              int maxevents, const struct timespec *timeout,
              const sigset_t *sigmask);
```

Variations of epoll() - libevent, libev, ...

- libevent.org/
- software.schmorp.de/pkg/libev.html

There are probably more...

Multiplexing Alternatives: More Efficient, Less Portable (kqueue on BSD)

kqueue() also scales well ($\mathcal{O}(1)$) on BSD/MacOS.

```
#include "sys/event.h"
kqueue(void);
kevent(int kq, const struct kevent *changelist, int nchanges,
         struct kevent *eventlist, int nevents,
         const struct timespec *timeout);
EV_SET(&kev, ident, filter, flags, fflags, data, udata);

struct kevent {
    uintptr_t   ident;     /* identifier for this event */
    short       filter;      /* filter for event */
    u_short     flags;     /* action flags for kqueue */
    u_int       fflags;      /* filter flag value */
    int64_t     data;        /* filter data value */
    void       *udata;       /* opaque user data identifier */
};
```

man.openbsd.org/kqueue.2