Problem Set 1 Solutions

1. *This code is trying to wait until something happens on file descriptor* `fd`*, or a signal is caught, whichever comes first. It relies on the fact that Unix blocking system calls, including* `select`*, exit early when the program receives a signal. Does this program work?*

```
int fd;
int have_received_a_signal;
extern "C" int signal_handler(int signal_number) {
    have_received_a_signal = 1;
}
int wait_for_signal_or_fd_activity() {
    fd_set read_fds, write_fds, except_fds;
    FD_ZERO(&read_fds);
    FD_SET(fd, &read_fds);
    write_fds = except_fds = read_fds;
    if (have_received_a_signal)
        return SAW_SIGNAL;
    /* Race condition: A signal might occur right here! */
    int retval = select(fd + 1, &read_fds, &write_fds, &except_fds, 0);
    if (retval > 0 && (FD_ISSET(fd, &read_fds) || FD_ISSET(fd, &write_fds)
                       || FD_ISSET(fd, &except_fds)))
        return SAW_FD_ACTIVITY;
    else if (retval < 0 && errno == EINTR) {
        assert(have_received_a_signal);
        return SAW_SIGNAL;
    } else if (retval < 0)
        /* Maybe the kernel ran out of memory */
        return -1;
    else
        /* This should never happen! */
        return WORLD_EXPLODING;
}
```

There's a race condition: a signal might arrive after we check `have_received_a_signal`, but before we call `select` (the boldfaced line above). If this happened, we might wait forever even though a signal had already arrived.

The `pselect` system call can be used to avoid this deadlock, as follows.

```
int fd;
int have_received_a_signal;
extern "C" int signal_handler(int signal_number) {
    have_received_a_signal = 1;
}
int wait_for_signal_or_fd_activity() {
    fd_set read_fds, write_fds, except_fds;
    sigset_t sigmask, original_sigmask;
    FD_ZERO(&read_fds);
    FD_SET(fd, &read_fds);
    write_fds = except_fds = read_fds;
    /* Temporarily block signals from arriving. */
    sigemptyset(&sigmask);
    sigaddset(&sigmask, SIGUSR1);  /* Repeat for other signals */
    sigprocmask(SIG_BLOCK, &sigmask, &original_sigmask);
    /* Did a signal arrive already? */
    if (have_received_a_signal) {
        sigprocmask(SIG_SETMASK, &original_sigmask);
        return SAW_SIGNAL;
    }
    int retval = pselect(fd + 1, &read_fds, &write_fds, &except_fds, 0,
              &original_sigmask);
    sigprocmask(SIG_SETMASK, &original_sigmask);
    if (retval > 0 && (FD_ISSET(fd, &read_fds) || FD_ISSET(fd, &write_fds)
                       || FD_ISSET(fd, &except_fds)))
        /* Note: 'have_received_a_signal' might be true here if a
        signal arrived after pselect returned, but we still know
        the file descriptor became active first. */
        return SAW_FD_ACTIVITY;
    else if (retval < 0 && errno == EINTR) {
        assert(have_received_a_signal);
        return SAW_SIGNAL;
    } else if (retval < 0)
        /* Maybe the kernel ran out of memory */
        return -1;
    else
        /* This should never happen! */
        return WORLD_EXPLODING;
}
```

Pretty horrible! Unix signals cause all kinds of concurrency problems and race conditions like this. That should make you think. Several people suggested solutions for Problem 3a based on *upcalls*, or callback functions activated directly by the kernel. Signals are a form of upcall. Could you improve upcall semantics to avoid such race conditions, or are race conditions inherent in the very idea of an upcall?

People also pointed out that `fd` should be checked against `FD_SETSIZE`, that not all signals can be caught, that the operating system itself can't say which event came first with total precision, that you should add `fd` to `except_fds` to catch the case when `fd` is closed, and so forth.

2. *Asynchronous I/O lets us build fast servers, and additionally solves other problems just as a matter of course. Sketch a deadlock scenario where a client and a server are communicating over a socket using blocking I/O.*

A classical, but subtle, deadlock is when the client pipelines many requests to the server at the same time as the server sends a long reply to the client:

```
Client                            Server
write(server_fd, buffer, 20000);  write(client_fd, buffer2, 20000);
```

The client's send buffer will fill up because the server isn't reading, and the server's send buffer will fill up because the client isn't reading. Each process will block waiting for space in its send buffer; that space would only materialize if the other process woke up and read.

The lesson here is to remember to make *writes* asynchronous as well as reads, even if your client and server are running on the same machine, and even if you wrote both client and server (so neither is malicious).

Many people came up with more-or-less convoluted scenarios involving reads, including unreliable transport (a packet is dropped and both endpoints block on `read`), unexpected lengths (the server sent back less data than the client expects, then waits for the next request: both endpoints block on `read`), and so forth. The unreliable transport example is arguably uncommon; programmers know what "unreliable" means and expect replies to get lost.

Some people suggested a malicious client or server, but that isn't really deadlock. Technically, deadlock should involve at least two blocked processes (or threads or requests), each one waiting for a resource held by another. In the `write` scenario above, each process is waiting for send buffer space that can only be freed by the other process. If a server hangs waiting for data from a client, we say that the server is *blocked*; but there is no deadlock unless the client is simultaneously blocked waiting for data from the server.

Here's a fun scenario from Ryan Cunningham: Say the client wrote a request to the server, then blocked to read the reply. I/O packages frequently *buffer* their writes to avoid excessive system calls, however; the client must make sure to flush its write before reading, or the server will never get the request. The C `stdio` library generally buffers its reads and writes, but the straight `read` and `write` calls do not.

***3a.** Sketch a comprehensive design for an asynchronous I/O subsystem, including disks, or argue that one of the current designs is best. Think about buffer management, event notification, the cost of system calls, ease of use, and portability to application-level interfaces. (Many servers provide an I/O-like interface; think of an HTTP server, which is basically* `open/read` *if you ignore* `POST`*. Could your design apply there too?) Don't be afraid to present something partially-baked, as long as it's interesting and different.*

Note: There were no "right" answers to Problem 3, so I'm not including explicit solutions.

One common suggestion was to have I/O system calls include a callback, which the kernel would call once the I/O was complete. In some ways this is the obvious interface, but there are some important details. When might the callback be called? At any time, or only after `select/poll`? If the function might be called at any time, then you have all sorts of concurrency issues. How might the user protect critical sections of code? If the function might be called only after `select` or a similar system call, then the callback interface could be implemented all at user level.

There were several suggestions that provided no way to cancel an outstanding I/O request. Any call that queues a request for later processing should probably return a token that can be used to control the request – to change its scheduling or cancel it, for example. The conventional `read/write` asynchronous I/O calls don't need to worry about this, since requests are never queued. The POSIX `aio_` calls use `struct aiocb` pointers as request tokens.

Also make sure you conside how many interface calls are required to complete a request *in the best case* as well as the worst case. An asynchronous `read` of n bytes requires a best case of one kernel-user crossing if the data is already in the buffer cache. However, it may take $O(n)$ crossings in the worst case, even with proper use of `select/poll`, if only one byte becomes available at a time. The POSIX `aio_` calls, and many of your solutions, require 2–3 kernel crossings in *every* case. Could you imagine combining the advantages of these systems?