REMOVE

6.3 select Function

This function allows the process to instruct the kernel to wait for any one of multiple events to occur and to wake up the process only when one or more of these events occurs or when a specified amount of time has passed.

As an example, we can call `select` and tell the kernel to return only when

- any of the descriptors in the set {1, 4, 5} are ready for reading, or
- any of the descriptors in the set {2, 7} are ready for writing, or
- any of the descriptors in the set {1, 4} have an exception condition pending, or
- after 10.2 seconds have elapsed.

That is, we tell the kernel what descriptors we are interested in (for reading, writing, or an exception condition) and how long to wait. The descriptors in which we are interested are not restricted to sockets: any descriptor can be tested using `select`.

> Berkeley-derived implementations have always allowed I/O multiplexing with any descriptor. SVR3 originally limited I/O multiplexing to descriptors that were streams devices (Chapter 33), but this limitation was removed with SVR4.

```
#include <sys/select.h>
#include <sys/time.h>

int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
           const struct timeval *timeout);
```

Returns: positive count of ready descriptors, 0 on timeout, –1 on error

We start our description of this function with its final argument, which tells the kernel how long to wait for one of the specified descriptors to become ready. A `timeval` structure specifies the number of seconds and microseconds.

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

There are three possibilities.

1. Wait forever: return only when one of the specified descriptors is ready for I/O. For this, we specify the *timeout* argument as a null pointer.
2. Wait up to a fixed amount of time: return when one of the specified descriptors is ready for I/O, but do not wait beyond the number of seconds and microseconds specified in the `timeval` structure pointed to by the *timeout* argument.
3. Do not wait at all: return immediately after checking the descriptors. This is called *polling*. To specify this, the *timeout* argument must point to a `timeval`

select/strcliselect01.c

```
 1 #include    "unp.h"

 2 void
 3 str_cli(FILE *fp, int sockfd)
 4 {
 5     int     maxfdp1;
 6     fd_set  rset;
 7     char    sendline[MAXLINE], recvline[MAXLINE];

 8     FD_ZERO(&rset);
 9     for ( ; ; ) {
10         FD_SET(fileno(fp), &rset);
11         FD_SET(sockfd, &rset);
12         maxfdp1 = max(fileno(fp), sockfd) + 1;
13         Select(maxfdp1, &rset, NULL, NULL, NULL);

14         if (FD_ISSET(sockfd, &rset)) {  /* socket is readable */
15             if (Readline(sockfd, recvline, MAXLINE) == 0)
16                 err_quit("str_cli: server terminated prematurely");
17             Fputs(recvline, stdout);
18         }
19         if (FD_ISSET(fileno(fp), &rset)) {  /* input is readable */
20             if (Fgets(sendline, MAXLINE, fp) == NULL)
21                 return;         /* all done */
22             Writen(sockfd, sendline, strlen(sendline));
23         }
24     }
25 }
```

select/strcliselect01.c

Figure 6.9 Implementation of `str_cli` function using `select` (improved in Figure 6.13).

Handle readable input

19-23 If the standard input is readable, a line is read by `fgets` and written to the socket using `writen`.

Notice that the same four I/O functions are used as in Figure 5.5: `fgets`, `writen`, `readline`, and `fputs`, but the order of flow within the function has changed. Instead of the function flow being driven by the call to `fgets`, it is now driven by the call to `select`. With only a few additional lines of code in Figure 6.9, compared to Figure 5.5, we have added greatly to the robustness of our client.

6.5 Batch Input

Unfortunately, our `str_cli` function is still not correct. First let's go back to our original version, Figure 5.5. It operates in a stop-and-wait mode, which is fine for interactive use: it sends a line to the server and then waits for the reply. This amount of time is one RTT (round-trip time) plus the server's processing time (which is close to 0 for a simple echo server). We can therefore estimate how long it will take for a given number of lines to be echoed, if we know the RTT between the client and server.

standard input or the socket to be readable. Figure 6.8 shows the various conditions that are handled by our call to `select`.

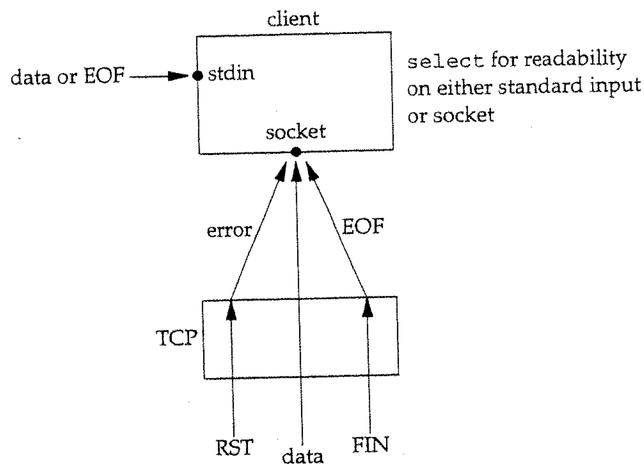

Figure 6.8 Conditions handled by `select` in `str_cli`.

Three conditions are handled with the socket.

1. If the peer TCP sends data, the socket becomes readable and `read` returns greater than 0 (i.e., the number of bytes of data).
2. If the peer TCP sends a FIN (the peer process terminates), the socket becomes readable and `read` returns 0 (end-of-file).
3. If the peer TCP sends an RST (the peer host has crashed and rebooted), the socket becomes readable and `read` returns –1 and `errno` contains the specific error code.

Figure 6.9 shows the source code for this new version.

Call `select`

8–13 We only need one descriptor set—to check for readability. This set is initialized by `FD_ZERO` and then two bits are turned on using `FD_SET`: the bit corresponding to the standard I/O file pointer `fp` and the bit corresponding to the socket `sockfd`. The function `fileno` converts a standard I/O file pointer into its corresponding descriptor. `select` (and `poll`) work only with descriptors.

`select` is called after calculating the maximum of the two descriptors. In the call the write-set pointer and the exception-set pointer are both null pointers. The final argument (the time limit) is also a null pointer since we want the call to block until something is ready.

Handle readable socket

14–18 If, on return from `select`, the socket is readable, the echoed line is read with `readline` and output by `fputs`.

Condition	readable?	writable?	exception?
data to read	•		
read-half of the connection closed	•		
new connection ready for listening socket	•		
space available for writing		•	
write-half of the connection closed		•	
pending error	•	•	
TCP out-of-band data			•

Figure 6.7 Summary of conditions that cause a socket to be ready for `select`.

per process (often limited only by the amount of memory and any administrative limits), so the question is: how does this affect `select`?

Many implementations have declarations similar to the following, which are taken from the 4.4BSD `<sys/types.h>` header:

```
/*
 * Select uses bitmasks of file descriptors in longs.  These macros
 * manipulate such bit fields (the filesystem macros use chars).
 * FD_SETSIZE may be defined by the user, but the default here should
 * be enough for most uses.
 */
#ifndef FD_SETSIZE
#define FD_SETSIZE      256
#endif
```

This makes us think that we can just `#define FD_SETSIZE` to some larger value before including this header to increase the size of the descriptor sets used by `select`. Unfortunately, this normally does not work.

> To see what is wrong, notice that Figure 16.53 of TCPv2 declares three descriptor sets within the kernel and also uses the kernel's definition of `FD_SETSIZE` as the upper limit. The only way to increase the size of the descriptor sets is to increase the value of `FD_SETSIZE` and then recompile the kernel. Changing the value without recompiling the kernel is inadequate.

Some vendors are changing their implementation of `select` to allow the process to define `FD_SETSIZE` to a larger than default value. BSD/OS has changed the kernel implementation to allow larger descriptor sets, and it also provides four new `FD_`*xxx* macros to dynamically allocate and manipulate these larger sets. From a portability standpoint, however, beware of using large descriptor sets.

6.4 `str_cli` Function (Revisited)

We can now rewrite our `str_cli` function from Section 5.5, this time using `select`, so we are notified as soon as the server process terminates. The problem with that earlier version was that we could be blocked in the call to `fgets` when something happened on the socket. Our new version blocks in a call to `select` instead, waiting for either

2. A socket is ready for writing if any of the following three conditions is true:
   a. The number of bytes of available space in the socket send buffer is greater than or equal to the current size of the low-water mark for the socket send buffer *and* either (i) the socket is connected, or (ii) the socket does not require a connection (e.g., UDP). This means that if we set the socket nonblocking (Chapter 15), a write operation will not block and will return a positive value (e.g., the number of bytes accepted by the transport layer). We can set this low-water mark using the `SO_SNDLOWAT` socket option. This low-water mark normally defaults to 2048 for TCP and UDP sockets.
   b. The write-half of the connection is closed. A write operation on the socket will generate `SIGPIPE` (Section 5.12).
   c. A socket error is pending. A write operation on the socket will not block and will return an error (–1) with `errno` set to the specific error condition. These *pending errors* can also be fetched and cleared by calling `getsockopt` with the `SO_ERROR` socket option.
3. A socket has an exception condition pending if there exists out-of-band data for the socket or the socket is still at the out-of-band mark. (We describe out-of-band data in Chapter 21.)

> Our definitions of "readable" and "writable" are taken directly from the kernel's `soreadable` and `sowriteable` macros on pp. 530–531 of TCPv2. Similarly our definition of the "exception condition" for a socket is from the `soo_select` function on these same pages.

Notice that when an error occurs on a socket it is marked as both readable and writable by `select`.

The purpose of the receive and send low-water marks is to give the application control over how much data must be available for reading or how much space must be available for writing before `select` returns readable or writable. For example, if we know that our application has nothing productive to do unless at least 64 bytes of data are present, we can set the receive low-water mark to 64 to prevent `select` from waking us up if less than 64 bytes are ready for reading.

As long as the send low-water mark for a UDP socket is less than the send buffer size (which should always be the default relationship), the UDP socket is always writable, since a connection is not required.

Figure 6.7 summarizes the conditions just described that cause a socket to be ready for `select`.

Maximum Number of Descriptors for `select`?

We said earlier that most applications do not use lots of descriptors. It is rare, for example, to find an application that uses hundreds of descriptors. But these applications do exist, and they often use `select` to multiplex the descriptors. When `select` was originally designed, the operating system normally had an upper limit on the maximum number of descriptors per process (the 4.2BSD limit was 31), and `select` just used this same limit. But current versions of Unix allow for an unlimited number of descriptors test a specific descriptor in an `fd_set` structure. Any descriptor that is not ready on return will have its corresponding bit cleared in the descriptor set. To handle this we turn on all the bits in which we are interested in all the descriptor sets each time we call `select`.

> The two most common programming errors when using `select` are to forget to add one to the largest descriptor number and to forget that the descriptor sets are value–result. The second error results in `select` being called with a bit set to 0 in the descriptor set, when we think that bit is 1. The author also wasted 2 hours debugging an example for this text that uses `select` by forgetting to add one to the first argument.

The return value from this function indicates the total number of bits that are ready across all the descriptor sets. If the timer value expires before any of the descriptors are ready, a value of 0 is returned. A return value of –1 indicates an error (which can happen, for example, if the function is interrupted by a caught signal).

> Early releases of SVR4 had a bug in their implementation of `select`: if the same bit was on in multiple sets, say a descriptor was ready for both reading and writing, it was counted only once. Current releases fix this bug.

Under What Conditions Is a Descriptor Ready?

We have been talking about waiting for a descriptor to become ready for I/O (reading or writing) or to have an exception condition pending on it (out-of-band data). While readability and writability are obvious for descriptors such as regular files, we must be more specific about the conditions that cause `select` to return "ready" for sockets (Figure 16.52 of TCPv2).

1. A socket is ready for reading if any of the following four conditions is true:
   a. The number of bytes of data in the socket receive buffer is greater than or equal to the current size of the low-water mark for the socket receive buffer. A read operation on the socket will not block and will return a value greater than 0 (i.e., the data that is ready to be read). We can set this low-water mark using the `SO_RCVLOWAT` socket option. It defaults to 1 for TCP and UDP sockets.
   b. The read-half of the connection is closed (i.e., a TCP connection that has received a FIN). A read operation on the socket will not block and will return 0 (i.e., end-of-file).
   c. The socket is a listening socket and the number of completed connections is nonzero. An `accept` on the listening socket will normally not block, although we describe a timing condition in Section 15.6 under which the `accept` can block.
   d. A socket error is pending. A read operation on the socket will not block and will return an error (–1) with `errno` set to the specific error condition. These *pending errors* can also be fetched and cleared by calling `getsockopt` specifying the `SO_ERROR` socket option.

We allocate a descriptor set of the fd_set datatype, we set and test the bits in the set using these macros, and we can also assign it to another descriptor set across an equals sign in C.

What we are describing, an array of integers using one bit per descriptor, is just one possible way to implement select. Nevertheless, it is common to refer to the individual descriptors within a descriptor set as *bits*, as in "turn on the bit for the listening descriptor in the read set."

We will see in Section 6.10 that the poll function uses a completely different representation: a variable-length array of structures with one structure per descriptor.

For example, to define a variable of type fd_set and then turn on the bits for descriptors 1, 4, and 5, we write

```
fd_set  rset;

FD_ZERO(&rset);         /* initialize the set: all bits off */
FD_SET(1, &rset);       /* turn on bit for fd 1 */
FD_SET(4, &rset);       /* turn on bit for fd 4 */
FD_SET(5, &rset);       /* turn on bit for fd 5 */
```

It is important to initialize the set, since unpredictable results can occur if the set is allocated as an automatic variable and not initialized.

Any of the middle three arguments to select, *readset*, *writeset*, or *exceptset*, can be specified as a null pointer, if we are not interested in that condition. Indeed, if all three pointers are null, then we have a higher precision timer than the normal Unix sleep function (which sleeps for multiples of a second). The poll function provides similar functionality. Figures C.9 and C.10 of APUE show a sleep_us function implemented using both select and poll that sleeps for multiples of a microsecond.

The *maxfdp1* argument specifies the number of descriptors to be tested. Its value is the maximum descriptor to be tested, plus one (hence our name of *maxfdp1*). The descriptors 0, 1, 2, up through and including *maxfdp1–1* are tested.

The constant FD_SETSIZE, defined by including <sys/select.h>, is the number of descriptors in the fd_set datatype. Its value is often 1024, but few programs use that many descriptors. The *maxfdp1* argument forces us to calculate the largest descriptor that we are interested in and then tell the kernel this value. For example, given the previous code that turns on the indicators for descriptors 1, 4, and 5, *maxfdp1* value is 6. The reason it is 6 and not 5 is that we are specifying the number of descriptors, not the largest value, and descriptors start at 0.

The reason this argument exists along with the burden of calculating its value is purely for efficiency. Although each fd_set has room for many descriptors, typically 1024, this is much more than the number used by a typical process. The kernel gains efficiency by not copying unneeded portions of the descriptor set between the process and the kernel, and by not testing bits that are always 0 (Section 16.13 of TCPv2).

select modifies the descriptor sets pointed to by the *readset*, *writeset*, and *exceptset* pointers. These three arguments are value–result arguments. When we call the function, we specify the values of the descriptors that we are interested in and on return the result indicates which descriptors are ready. We use the FD_ISSET macro on return to structure, and the timer value (the number of seconds and microseconds specified by the structure) must be 0.

The wait in the first two scenarios is normally interrupted if the process catches a signal and returns from the signal handler.

> Berkeley-derived kernels never automatically restart `select` (p. 527 of TCPv2), while SVR4 will if the `SA_RESTART` flag is specified when the signal handler is installed. This means that for portability we must be prepared for `select` to return an error of `EINTR` if we are catching signals.

Although the `timeval` structure lets us specify a resolution in microseconds, the actual resolution supported by the kernel is often more coarse. For example, many Unix kernels round the timeout value up to a multiple of 10 ms. There is also a scheduling latency involved, meaning it takes some time after the timer expires before the kernel schedules this process to run.

The `const` qualifier on the *timeout* argument means it is not modified by `select` on return. For example, if we specify a time limit of 10 seconds, and `select` returns before the timer expires, with one or more of the descriptors ready or with an error of `EINTR`, the `timeval` structure is not updated with the number of seconds remaining when the function returns. If we wish to know this value, we must obtain the system time before calling `select`, and then again when it returns, and subtract the two.

> Current Linux systems modify the `timeval` structure. Therefore for portability, assume the `timeval` structure is undefined upon return, and initialize it before each call to `select`. Posix.1g specifies the `const` qualifier.

The three middle arguments *readset*, *writeset*, and *exceptset* specify the descriptors that we want the kernel to test for reading, writing, and exception conditions. There are only two exception conditions currently supported.

1. The arrival of out-of-band data for a socket. We describe this in more detail in Chapter 21.
2. The presence of control status information to be read from the master side of a pseudo terminal that has been put into packet mode. We do not talk about pseudo terminals in this volume.

A design problem is how to specify one or more descriptor values for each of these three arguments. `select` uses *descriptor sets*, typically an array of integers, with each bit in each integer corresponding to a descriptor. For example, using 32-bit integers, the first element of the array corresponds to descriptors 0 through 31, the second element of the array corresponds to descriptors 32 through 63, and so on. All the implementation details are irrelevant to the application and are hidden in the `fd_set` datatype and the following four macros:

```
void FD_ZERO(fd_set *fdset);          /* clear all bits in fdset */
void FD_SET(int fd, fd_set *fdset);   /* turn on the bit for fd in fdset */
void FD_CLR(int fd, fd_set *fdset);   /* turn off the bit for fd in fdset */
int  FD_ISSET(int fd, fd_set *fdset); /* is the bit for fd on in fdset ? */
```