Aside Unix IPC

You have already encountered several examples of IPC in this text. The `waitpid` function and signals from Chapter 8 are primitive IPC mechanisms that allow processes to send tiny messages to processes running on the same host. The sockets interface from Chapter 11 is an important form of IPC that allows processes on different hosts to exchange arbitrary byte streams. However, the term *Unix IPC* is typically reserved for a hodgepodge of techniques that allow processes to communicate with other processes that are running on the same host. Examples include pipes, FIFOs, System V shared memory, and System V semaphores. These mechanisms are beyond our scope. The book by Kerrisk [62] is an excellent reference.

12.2 Concurrent Programming with I/O Multiplexing

Suppose you are asked to write an echo server that can also respond to interactive commands that the user types to standard input. In this case, the server must respond to two independent I/O events: (1) a network client making a connection request, and (2) a user typing a command line at the keyboard. Which event do we wait for first? Neither option is ideal. If we are waiting for a connection request in `accept`, then we cannot respond to input commands. Similarly, if we are waiting for an input command in `read`, then we cannot respond to any connection requests.

One solution to this dilemma is a technique called *I/O multiplexing*. The basic idea is to use the `select` function to ask the kernel to suspend the process, returning control to the application only after one or more I/O events have occurred, as in the following examples:

- Return when any descriptor in the set {0, 4} is ready for reading.
- Return when any descriptor in the set {1, 2, 7} is ready for writing.
- Time out if 152.13 seconds have elapsed waiting for an I/O event to occur.

`Select` is a complicated function with many different usage scenarios. We will only discuss the first scenario: waiting for a set of descriptors to be ready for reading. See [62, 110] for a complete discussion.

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

int select(int n, fd_set *fdset, NULL, NULL, NULL);
                          Returns: nonzero count of ready descriptors, −1 on error

FD_ZERO(fd_set *fdset);           /* Clear all bits in fdset */
FD_CLR(int fd, fd_set *fdset);    /* Clear bit fd in fdset */
FD_SET(int fd, fd_set *fdset);    /* Turn on bit fd in fdset */
FD_ISSET(int fd, fd_set *fdset);  /* Is bit fd in fdset on? */
                                      Macros for manipulating descriptor sets
```

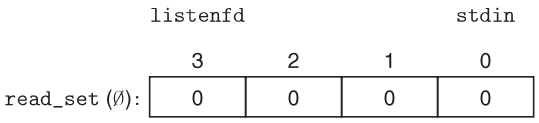
The `select` function manipulates sets of type `fd_set`, which are known as *descriptor sets*. Logically, we think of a descriptor set as a bit vector (introduced in Section 2.1) of size n:

$$b_{n-1}, \ldots, b_1, b_0$$

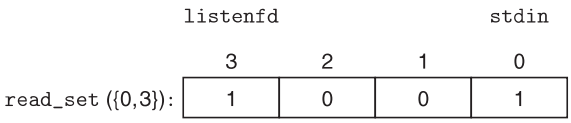
Each bit b_k corresponds to descriptor k. Descriptor k is a member of the descriptor set if and only if $b_k = 1$. You are only allowed to do three things with descriptor sets: (1) allocate them, (2) assign one variable of this type to another, and (3) modify and inspect them using the FD_ZERO, FD_SET, FD_CLR, and FD_ISSET macros.

For our purposes, the `select` function takes two inputs: a descriptor set (`fdset`) called the *read set*, and the cardinality (`n`) of the read set (actually the maximum cardinality of any descriptor set). The `select` function blocks until at least one descriptor in the read set is ready for reading. A descriptor k is *ready for reading* if and only if a request to read 1 byte from that descriptor would not block. As a side effect, `select` modifies the `fd_set` pointed to by argument `fdset` to indicate a subset of the read set called the *ready set*, consisting of the descriptors in the read set that are ready for reading. The value returned by the function indicates the cardinality of the ready set. Note that because of the side effect, we must update the read set every time `select` is called.

The best way to understand `select` is to study a concrete example. Figure 12.6 shows how we might use `select` to implement an iterative echo server that also accepts user commands on the standard input. We begin by using the `open_listenfd` function from Figure 11.19 to open a listening descriptor (line 16), and then using FD_ZERO to create an empty read set (line 18):



Next, in lines 19 and 20, we define the read set to consist of descriptor 0 (standard input) and descriptor 3 (the listening descriptor), respectively:



At this point, we begin the typical server loop. But instead of waiting for a connection request by calling the `accept` function, we call the `select` function, which blocks until either the listening descriptor or standard input is ready for reading (line 24). For example, here is the value of `ready_set` that `select` would return if the user hit the enter key, thus causing the standard input descriptor to

code/conc/select.c

```
1  #include "csapp.h"
2  void echo(int connfd);
3  void command(void);
4
5  int main(int argc, char **argv)
6  {
7      int listenfd, connfd;
8      socklen_t clientlen;
9      struct sockaddr_storage clientaddr;
10     fd_set read_set, ready_set;
11
12     if (argc != 2) {
13         fprintf(stderr, "usage: %s <port>\n", argv[0]);
14         exit(0);
15     }
16     listenfd = Open_listenfd(argv[1]);
17
18     FD_ZERO(&read_set);              /* Clear read set */
19     FD_SET(STDIN_FILENO, &read_set); /* Add stdin to read set */
20     FD_SET(listenfd, &read_set);     /* Add listenfd to read set */
21
22     while (1) {
23         ready_set = read_set;
24         Select(listenfd+1, &ready_set, NULL, NULL, NULL);
25         if (FD_ISSET(STDIN_FILENO, &ready_set))
26             command(); /* Read command line from stdin */
27         if (FD_ISSET(listenfd, &ready_set)) {
28             clientlen = sizeof(struct sockaddr_storage);
29             connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
30             echo(connfd); /* Echo client input until EOF */
31             Close(connfd);
32         }
33     }
34 }
35
36 void command(void) {
37     char buf[MAXLINE];
38     if (!Fgets(buf, MAXLINE, stdin))
39         exit(0); /* EOF */
40     printf("%s", buf); /* Process the input command */
41 }
```

code/conc/select.c

Figure 12.6 An iterative echo server that uses I/O multiplexing. The server uses `select` to wait for connection requests on a listening descriptor and commands on standard input.

become ready for reading:

	listenfd			stdin
	3	2	1	0
ready_set ({0}):	0	0	0	1

Once `select` returns, we use the FD_ISSET macro to determine which descriptors are ready for reading. If standard input is ready (line 25), we call the `command` function, which reads, parses, and responds to the command before returning to the main routine. If the listening descriptor is ready (line 27), we call `accept` to get a connected descriptor and then call the `echo` function from Figure 11.22, which echoes each line from the client until the client closes its end of the connection.

While this program is a good example of using `select`, it still leaves something to be desired. The problem is that once it connects to a client, it continues echoing input lines until the client closes its end of the connection. Thus, if you type a command to standard input, you will not get a response until the server is finished with the client. A better approach would be to multiplex at a finer granularity, echoing (at most) one text line each time through the server loop.

Practice Problem 12.3 **(solution page 1072)**

In Linux systems, typing Ctrl+D indicates EOF on standard input. What happens if you type Ctrl+D to the program in Figure 12.6 while it is echoing each line of the client?

12.2.1 A Concurrent Event-Driven Server Based on I/O Multiplexing

I/O multiplexing can be used as the basis for concurrent *event-driven* programs, where flows make progress as a result of certain events. The general idea is to model logical flows as state machines. Informally, a *state machine* is a collection of *states*, *input events*, and *transitions* that map states and input events to states. Each transition maps an (input state, input event) pair to an output state. A *self-loop* is a transition between the same input and output state. State machines are typically drawn as directed graphs, where nodes represent states, directed arcs represent transitions, and arc labels represent input events. A state machine begins execution in some initial state. Each input event triggers a transition from the current state to the next state.

For each new client k, a concurrent server based on I/O multiplexing creates a new state machine s_k and associates it with connected descriptor d_k. As shown in Figure 12.7, each state machine s_k has one state ("waiting for descriptor d_k to be ready for reading"), one input event ("descriptor d_k is ready for reading"), and one transition ("read a text line from descriptor d_k").

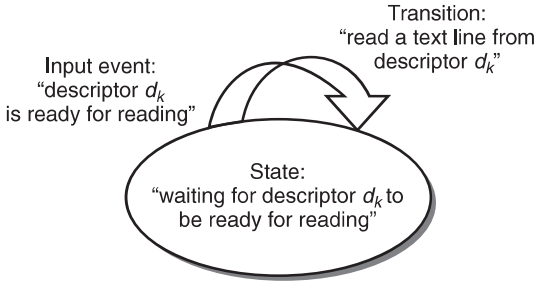


Figure 12.7
State machine for a logical flow in a concurrent event-driven echo server.

The server uses the I/O multiplexing, courtesy of the `select` function, to detect the occurrence of input events. As each connected descriptor becomes ready for reading, the server executes the transition for the corresponding state machine—in this case, reading and echoing a text line from the descriptor.

Figure 12.8 shows the complete example code for a concurrent event-driven server based on I/O multiplexing. The set of active clients is maintained in a `pool` structure (lines 3–11). After initializing the pool by calling `init_pool` (line 27), the server enters an infinite loop. During each iteration of this loop, the server calls the `select` function to detect two different kinds of input events: (1) a connection request arriving from a new client, and (2) a connected descriptor for an existing client being ready for reading. When a connection request arrives (line 35), the server opens the connection (line 37) and calls the `add_client` function to add the client to the pool (line 38). Finally, the server calls the `check_clients` function to echo a single text line from each ready connected descriptor (line 42).

The `init_pool` function (Figure 12.9) initializes the client pool. The `clientfd` array represents a set of connected descriptors, with the integer −1 denoting an available slot. Initially, the set of connected descriptors is empty (lines 5–7), and the listening descriptor is the only descriptor in the `select` read set (lines 10–12).

The `add_client` function (Figure 12.10) adds a new client to the pool of active clients. After finding an empty slot in the `clientfd` array, the server adds the connected descriptor to the array and initializes a corresponding RIO read buffer so that we can call `rio_readlineb` on the descriptor (lines 8–9). We then add the connected descriptor to the `select` read set (line 12), and we update some global properties of the pool. The `maxfd` variable (lines 15–16) keeps track of the largest file descriptor for `select`. The `maxi` variable (lines 17–18) keeps track of the largest index into the `clientfd` array so that the `check_clients` function does not have to search the entire array.

The `check_clients` function in Figure 12.11 echoes a text line from each ready connected descriptor. If we are successful in reading a text line from the descriptor, then we echo that line back to the client (lines 15–18). Notice that in line 15, we are maintaining a cumulative count of total bytes received from all clients. If we detect EOF because the client has closed its end of the connection, then we close our end of the connection (line 23) and remove the descriptor from the pool (lines 24–25).

code/conc/echoservers.c

```
1   #include "csapp.h"
2
3   typedef struct { /* Represents a pool of connected descriptors */
4       int maxfd;        /* Largest descriptor in read_set */
5       fd_set read_set;  /* Set of all active descriptors */
6       fd_set ready_set; /* Subset of descriptors ready for reading  */
7       int nready;       /* Number of ready descriptors from select */
8       int maxi;         /* High water index into client array */
9       int clientfd[FD_SETSIZE];    /* Set of active descriptors */
10      rio_t clientrio[FD_SETSIZE]; /* Set of active read buffers */
11  } pool;
12
13  int byte_cnt = 0; /* Counts total bytes received by server */
14
15  int main(int argc, char **argv)
16  {
17      int listenfd, connfd;
18      socklen_t clientlen;
19      struct sockaddr_storage clientaddr;
20      static pool pool;
21
22      if (argc != 2) {
23          fprintf(stderr, "usage: %s <port>\n", argv[0]);
24          exit(0);
25      }
26      listenfd = Open_listenfd(argv[1]);
27      init_pool(listenfd, &pool);
28
29      while (1) {
30          /* Wait for listening/connected descriptor(s) to become ready */
31          pool.ready_set = pool.read_set;
32          pool.nready = Select(pool.maxfd+1, &pool.ready_set, NULL, NULL, NULL);
33
34          /* If listening descriptor ready, add new client to pool */
35          if (FD_ISSET(listenfd, &pool.ready_set)) {
36              clientlen = sizeof(struct sockaddr_storage);
37              connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
38              add_client(connfd, &pool);
39          }
40
41          /* Echo a text line from each ready connected descriptor */
42          check_clients(&pool);
43      }
44  }
```

code/conc/echoservers.c

Figure 12.8 Concurrent echo server based on I/O multiplexing. Each server iteration echoes a text line from each ready descriptor.

code/conc/echoservers.c

```
1  void init_pool(int listenfd, pool *p)
2  {
3      /* Initially, there are no connected descriptors */
4      int i;
5      p->maxi = -1;
6      for (i=0; i< FD_SETSIZE; i++)
7          p->clientfd[i] = -1;
8
9      /* Initially, listenfd is only member of select read set */
10     p->maxfd = listenfd;
11     FD_ZERO(&p->read_set);
12     FD_SET(listenfd, &p->read_set);
13 }
```

code/conc/echoservers.c

Figure 12.9 `init_pool` **initializes the pool of active clients.**

code/conc/echoservers.c

```
1  void add_client(int connfd, pool *p)
2  {
3      int i;
4      p->nready--;
5      for (i = 0; i < FD_SETSIZE; i++) /* Find an available slot */
6          if (p->clientfd[i] < 0) {
7              /* Add connected descriptor to the pool */
8              p->clientfd[i] = connfd;
9              Rio_readinitb(&p->clientrio[i], connfd);
10
11             /* Add the descriptor to descriptor set */
12             FD_SET(connfd, &p->read_set);
13
14             /* Update max descriptor and pool high water mark */
15             if (connfd > p->maxfd)
16                 p->maxfd = connfd;
17             if (i > p->maxi)
18                 p->maxi = i;
19             break;
20         }
21     if (i == FD_SETSIZE) /* Couldn't find an empty slot */
22         app_error("add_client error: Too many clients");
23 }
```

code/conc/echoservers.c

Figure 12.10 `add_client` **adds a new client connection to the pool.**

code/conc/echoservers.c

```
1  void check_clients(pool *p)
2  {
3      int i, connfd, n;
4      char buf[MAXLINE];
5      rio_t rio;
6
7      for (i = 0; (i <= p->maxi) && (p->nready > 0); i++) {
8          connfd = p->clientfd[i];
9          rio = p->clientrio[i];
10
11         /* If the descriptor is ready, echo a text line from it */
12         if ((connfd > 0) && (FD_ISSET(connfd, &p->ready_set))) {
13             p->nready--;
14             if ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
15                 byte_cnt += n;
16                 printf("Server received %d (%d total) bytes on fd %d\n",
17                        n, byte_cnt, connfd);
18                 Rio_writen(connfd, buf, n);
19             }
20
21             /* EOF detected, remove descriptor from pool */
22             else {
23                 Close(connfd);
24                 FD_CLR(connfd, &p->read_set);
25                 p->clientfd[i] = -1;
26             }
27         }
28     }
29 }
```

code/conc/echoservers.c

Figure 12.11 `check_clients` **services ready client connections.**

In terms of the finite state model in Figure 12.7, the `select` function detects input events, and the `add_client` function creates a new logical flow (state machine). The `check_clients` function performs state transitions by echoing input lines, and it also deletes the state machine when the client has finished sending text lines.

Practice Problem 12.4 (solution page 1072)

In the server in Figure 12.8, `pool.nready` is reinitialized with the value obtained from the call to `select`. Why?

Aside Event-driven Web servers

Despite the disadvantages outlined in Section 12.2.2, modern high-performance servers such as Node.js, nginx, and Tornado use event-driven programming based on I/O multiplexing, mainly because of the significant performance advantage compared to processes and threads.

12.2.2 Pros and Cons of I/O Multiplexing

The server in Figure 12.8 provides a nice example of the advantages and disadvantages of event-driven programming based on I/O multiplexing. One advantage is that event-driven designs give programmers more control over the behavior of their programs than process-based designs. For example, we can imagine writing an event-driven concurrent server that gives preferred service to some clients, which would be difficult for a concurrent server based on processes.

Another advantage is that an event-driven server based on I/O multiplexing runs in the context of a single process, and thus every logical flow has access to the entire address space of the process. This makes it easy to share data between flows. A related advantage of running as a single process is that you can debug your concurrent server as you would any sequential program, using a familiar debugging tool such as GDB. Finally, event-driven designs are often significantly more efficient than process-based designs because they do not require a process context switch to schedule a new flow.

A significant disadvantage of event-driven designs is coding complexity. Our event-driven concurrent echo server requires three times more code than the process-based server. Unfortunately, the complexity increases as the granularity of the concurrency decreases. By *granularity*, we mean the number of instructions that each logical flow executes per time slice. For instance, in our example concurrent server, the granularity of concurrency is the number of instructions required to read an entire text line. As long as some logical flow is busy reading a text line, no other logical flow can make progress. This is fine for our example, but it makes our event-driven server vulnerable to a malicious client that sends only a partial text line and then halts. Modifying an event-driven server to handle partial text lines is a nontrivial task, but it is handled cleanly and automatically by a process-based design. Another significant disadvantage of event-based designs is that they cannot fully utilize multi-core processors.

12.3 Concurrent Programming with Threads

To this point, we have looked at two approaches for creating concurrent logical flows. With the first approach, we use a separate process for each flow. The kernel schedules each process automatically, and each process has its own private address space, which makes it difficult for flows to share data. With the second approach, we create our own logical flows and use I/O multiplexing to explicitly schedule the flows. Because there is only one process, flows share the entire address space.