Chapter 13

Concurrent Programming

As we learned in Chapter 8, logical control flows are *concurrent* if they overlap in time. This general phenomenon, known as *concurrency*, shows up at many different levels of a computer system. Hardware exception handlers, processes, and Unix signal handlers are all familiar examples.

To this point, we have treated concurrency mainly as a mechanism that the kernel uses to run multiple application programs. But concurrency is not just limited to the kernel. It can play an important role in application programs as well. For example, we have seen how Unix signal handlers allow applications to respond to asynchronous events such as the user typing `ctrl-c`, or the program accessing an undefined area of virtual memory. Application-level concurrency is useful in other ways as well:

- *Computing in parallel on multiprocessors.* In a *uniprocessor* with a single CPU, concurrent flows are interleaved, with only one flow actually executing on the CPU at any point in time. However, there are machines with multiple CPUs, called *multiprocessors*, that can truly execute multiple flows simultaneously. *Parallel applications* that are partitioned into concurrent flows can sometimes run much faster on such machines. This is especially important for large-scale database and scientific applications.

- *Accessing slow I/O devices.* When an application is waiting for data to arrive from a slow I/O device such as a disk, the kernel keeps the CPU busy by running other processes. Individual applications can exploit concurrency in a similar way by overlapping useful work with I/O requests.

- *Interacting with humans.* People who interact with computers demand the ability to perform multiple tasks at the same time. For example, they might want to resize a window while they are printing a document. Modern windowing systems use concurrency to provide this capability. Each time the user requests some action (say by clicking the mouse), a separate concurrent logical flow is created to perform the action.

- *Reducing latency by deferring work.* Sometimes, applications can use concurrency to reduce the latency of certain operations by deferring other operations and performing them concurrently. For example, a dynamic storage allocator might reduce the latency of individual `free` operations by deferring coalescing to a concurrent "coalescing" flow that runs at a lower priority, soaking up spare CPU cycles as they become available.

- *Servicing multiple network clients.* The iterative network servers that we studied in Chapter 12 are unrealistic because they can only service one client at a time. Thus, a single slow client can deny service to every other client. For a real server that might be expected to service hundreds or thousands of clients per second, it is not acceptable to allow one slow client to deny service to the others. A better approach is to build a *concurrent server* that creates a separate logical flow for each client. This allows the server to service multiple clients concurrently, and precludes slow clients from monopolizing the server.

Applications that use application-level concurrency are known as *concurrent programs*. Modern operating systems provide three basic approaches for building concurrent programs:

- *Processes.* With this approach, each logical control flow is a process that is scheduled and maintained by the kernel. Since processes have separate virtual address spaces, flows that want to communicate with each other must use some kind of explicit *interprocess communication* (IPC) mechanism.
- *I/O multiplexing.* This is a form of concurrent programming where applications explicitly schedule their own logical flows in the context of a single process. Logical flows are modeled as state machines that the main program explicitly transitions from state to state as a result of data arriving on file descriptors. Since the program is a single process, all flows share the same address space.
- *Threads.* Threads are logical flows that run in the context of a single process and are scheduled by the kernel. You can think of threads as a hybrid of the other two approaches, scheduled by the kernel like process flows, and sharing the same virtual address space like I/O multiplexing flows.

This chapter investigates these different concurrent programming techniques. To keep our discussion concrete, we will work with the same motivating application throughout – a concurrent version of the iterative echo server from Section 12.4.9.

13.1 Concurrent Programming With Processes

The simplest way to build a concurrent program is with processes, using familiar functions such as `fork`, `exec`, and `waitpid`. For example, a natural approach for building a concurrent server is to accept client connection requests in the parent, and then create a new child process to service each new client.

To see how this might work, suppose we have two clients and a server that is listening for connection requests on a listening descriptor (say 3). Now suppose that the server accepts a connection request from client 1 and returns a connected descriptor (say 4), as shown in Figure 13.1.

After accepting the connection request, the server forks a child, which gets a complete copy of the server's descriptor table. The child closes its copy of listening descriptor 3, and the parent closes its copy of connected descriptor 4, since they are no longer needed. This gives us the situation in Figure 13.2, where the child process is busy servicing the client. Since the connected descriptors in the parent and child each point to the same file table entry, it is crucial for the parent to close its copy of the connected descriptor. Otherwise, the file table entry for connected descriptor 4 will never be released, and the resulting memory leak will eventually consume the available memory and crash the system.

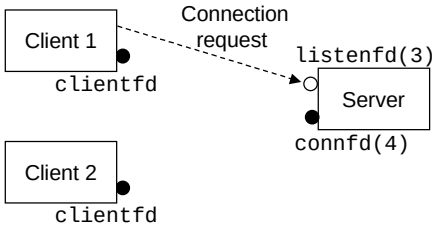

Figure 13.1: **Step 1: Server accepts connection request from client.**

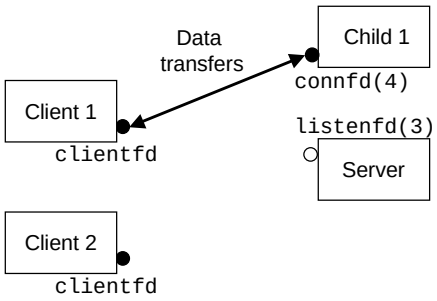

Figure 13.2: **Step 2: Server forks a child process to service the client.**

Now suppose that after the parent creates the child for client 1, it accepts a new connection request from client 2 and returns a new connected descriptor (say 5), as shown in Figure 13.3.

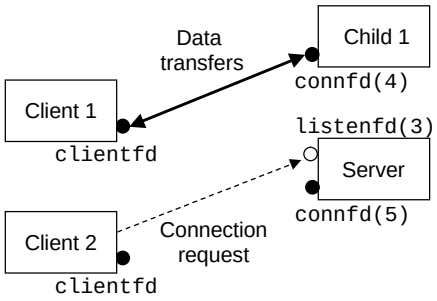

Figure 13.3: **Step 3: Server accepts another connection request.**

The parent then forks another child, which begins servicing its client using connected descriptor 5, as shown in Figure 13.4. At this point, the parent is waiting for the next connection request and the two children are servicing their respective clients concurrently.

13.1.1 A Concurrent Server Based on Processes

Figure 13.5 shows the code for a concurrent echo server based on processes. The `echo` function called in line 28 comes from Figure 12.21. There are several points of interest to make about this server:

- First, servers typically run for long periods of time, so we must include a SIGCHLD handler that

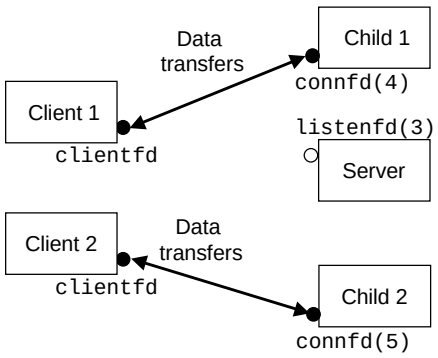

Figure 13.4: **Step 4: Server forks another child to service the new client.**

reaps zombie children (lines 4–9). Since SIGCHLD signals are blocked while the SIGCHLD handler is executing, and since Unix signals are not queued, the SIGCHLD handler must be prepared to reap multiple zombie children.

- Second, the parent and the child must close their respective copies of `connfd` (lines 32 and 29 respectively). As we have mentioned, this is especially important for the parent, which must close its copy of the connected descriptor to avoid a memory leak.
- Finally, because of the reference count in the socket's file table entry, the connection to the client will not be terminated until both the parent's and child's copies of `connfd` are closed.

13.1.2 Pros and Cons of Processes

Processes have a clean model for sharing state information between parents and children: File tables are shared and user address spaces are not. Having separate address spaces for processes is both an advantage and a disadvantage. It is impossible for one process to accidently overwrite the virtual memory of another process, which eliminates a lot of confusing failures – an obvious advantage.

On the other hand, separate address spaces make it more difficult for processes to share state information. To share information, they must use explicit IPC (interprocess communications) mechanisms. (See Aside.) Another disadvantage of process-based designs is that they tend to be slower because the overhead for process control and IPC is high.

Aside: Unix IPC
You have already encountered several examples of IPC in this text. The `waitpid` function and Unix 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 12 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 hodge-podge 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 Stevens [80] is a good reference. **End Aside.**

Practice Problem 13.1:

code/conc/echoserverp.c

```
1  #include "csapp.h"
2  void echo(int connfd);
3
4  void sigchld_handler(int sig)
5  {
6      while (waitpid(-1, 0, WNOHANG) > 0)
7          ;
8      return;
9  }
10
11 int main(int argc, char **argv)
12 {
13     int listenfd, connfd, port, clientlen=sizeof(struct sockaddr_in);
14     struct sockaddr_in clientaddr;
15
16     if (argc != 2) {
17         fprintf(stderr, "usage: %s <port>\n", argv[0]);
18         exit(0);
19     }
20     port = atoi(argv[1]);
21
22     Signal(SIGCHLD, sigchld_handler);
23     listenfd = Open_listenfd(port);
24     while (1) {
25         connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
26         if (Fork() == 0) {
27             Close(listenfd); /* Child closes its listening socket */
28             echo(connfd);    /* Child services client */
29             Close(connfd);   /* Child closes connection with client */
30             exit(0);         /* Child exits */
31         }
32         Close(connfd); /* Parent closes connected socket (important!) */
33     }
34 }
```

code/conc/echoserverp.c

Figure 13.5: **Concurrent echo server based on processes.** The parent forks a child to handle each new connection request.

After the parent closes the connected descriptor in line 32 of the concurrent server in Figure 13.5, the child is still able to communicate with the client using its copy of the descriptor. Why?

Practice Problem 13.2:

If we were to delete line 29 of Figure 13.5 that closes the connected descriptor, the code would still be correct, in the sense that there would be no memory leak. Why?

13.2 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.
- Timeout 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 [76, 81] for a complete discussion.

```
#include <unistd.h>
#include <sys/types.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 turned 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 mask 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 an 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. 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 one 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 13.6 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 12.17 to open a listening descriptor (line 16), and then using FD_ZERO to create an empty read set:

	listenfd			stdin
	3	2	1	0
`read_set` ($\emptyset$):	0	0	0	0

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

	listenfd			stdin
	3	2	1	0
`read_set` ($\{0,3\}$):	1	0	0	1

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 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 12.21, 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

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, port, clientlen = sizeof(struct sockaddr_in);
8     struct sockaddr_in clientaddr;
9     fd_set read_set, ready_set;
10
11    if (argc != 2) {
12        fprintf(stderr, "usage: %s <port>\n", argv[0]);
13        exit(0);
14    }
15    port = atoi(argv[1]);
16    listenfd = Open_listenfd(port);
17
18    FD_ZERO(&read_set);
19    FD_SET(STDIN_FILENO, &read_set);
20    FD_SET(listenfd, &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            connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
29            echo(connfd); /* echo client input until EOF */
30        }
31    }
32 }
33
34 void command(void) {
35    char buf[MAXLINE];
36    if (!Fgets(buf, MAXLINE, stdin))
37        exit(0); /* EOF */
38    printf("%s", buf); /* Process the input command */
39 }
```

code/conc/select.c

Figure 13.6: **An 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.

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. (See Problem 13.3.)

13.2.1 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 13.7, 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").

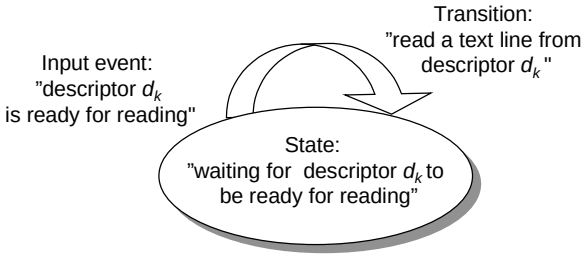

Figure 13.7: **State machine for a logical flow in a concurrent event-driven echo server.**

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 13.8 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 28), 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: (a) a connection request arriving from a new client, and (b) a connected descriptor for an existing client being ready for reading. When a connection request arrives (line 35), the server opens the connection (line 36) and calls the `add_client` function to add the client to the pool (line 37). Finally, the server calls the `check_client` function to echo a single text line from each ready connected descriptor (line 41).

The `init_pool` function (Figure 13.9) initializes the client pool. The `clientfd` array represents a set of connected descriptors, with -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 13.10) adds a new client to the pool of active clients. After finding

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;         /* highwater 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, port, clientlen = sizeof(struct sockaddr_in);
18     struct sockaddr_in clientaddr;
19     static pool pool;
20
21     if (argc != 2) {
22         fprintf(stderr, "usage: %s <port>\n", argv[0]);
23         exit(0);
24     }
25     port = atoi(argv[1]);
26
27     listenfd = Open_listenfd(port);
28     init_pool(listenfd, &pool);
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             connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
37             add_client(connfd, &pool);
38         }
39
40         /* Echo a text line from each ready connected descriptor */
41         check_clients(&pool);
42     }
43 }
```

code/conc/echoservers.c

Figure 13.8: **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 13.9: **`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 highwater 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 13.10: **`add_client`: Adds a new client connection to the pool.**

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` functions does not have to search the entire array.

The `check_clients` function echoes a text line from each ready connected descriptor. If we are success-

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 13.11: **`check_clients`: Services ready client connections.**

ful 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).

In terms of the finite state model in Figure 13.7, 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.

13.2.2 Pros and Cons of I/O Multiplexing

The server in Figure 13.8 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-driver 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.

Practice Problem 13.3:

In most Unix systems, typing `ctrl-d` indicates EOF on standard input. What happens if you type `ctrl-d` to the program in Figure 13.6 while it is blocked in the call to `select`?

Practice Problem 13.4:

In the server in Figure 13.8, we are careful to reinitialize the `pool.ready_set` variable immediately before every call to `select`. Why?

13.3 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. 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. This section introduces a third approach – based on threads – that is a hybrid of these two.

A *thread* is a logical flow that runs in the context of a process. Thus far in this book, our programs have consisted of a single thread per process. But modern systems also allow us to write programs that have multiple threads running concurrently in a single process. The threads are scheduled automatically by the kernel. Each thread has its own *thread context*, including a unique integer *thread ID (TID)*, stack, stack pointer, program counter, general-purpose registers, and condition codes. All threads running in a process share the entire virtual address space of that process.

Logical flows based on threads combine qualities of flows based on processes and I/O multiplexing. Like processes, threads are scheduled automatically by the kernel and are known to the kernel by an integer ID. Like flows based on I/O multiplexing, multiple threads run in the context of a single process, and thus share the entire contents of the process virtual address space, including its code, data, heap, shared libraries, and open files.

13.3.1 Thread Execution Model

The execution model for multiple threads is similar in some ways to the execution model for multiple processes. Consider the example in Figure 13.12. Each process begins life as a single thread called the *main thread*. At some point, the main thread creates a *peer thread*, and from this point in time the two threads run concurrently. Eventually, control passes to the peer thread via a context switch, because the main thread executes a slow system call such as `read` or `sleep`, or because it is interrupted by the system's interval timer. The peer thread executes for a while before control passes back to the main thread, and so on.

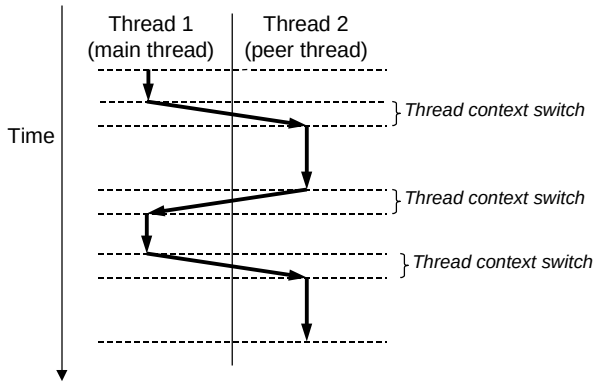

Figure 13.12: **Concurrent thread execution.**

Thread execution differs from processes in some important ways. Because a thread context is much smaller than a process context, a thread context switch is faster than a process context switch. Another difference is that threads, unlike processes, are not organized in a rigid parent-child hierarchy. The threads associated with a process form a *pool* of peers, independent of which threads were created by which other threads. The

main thread is distinguished from other threads only in the sense that it is always the first thread to run in the process. The main impact of this notion of a pool of peers is that a thread can kill any of its peers, or wait for any of its peers to terminate. Further, each peer can read and write the same shared data.

13.3.2 Posix Threads

Posix threads (Pthreads) is a standard interface for manipulating threads from C programs. It was adopted in 1995 and is available on most Unix systems. Pthreads defines about 60 functions that allow programs to create, kill, and reap threads, to share data safely with peer threads, and to notify peers about changes in the system state.

Figure 13.13 shows a simple Pthreads program. The main thread creates a peer thread and then waits for it to terminate. The peer thread prints "`Hello, world!\n`" and terminates. When the main thread detects that the peer thread has terminated, it terminates the process by calling `exit`.

code/conc/hello.c

```
1 #include "csapp.h"
2 void *thread(void *vargp);
3
4 int main()
5 {
6     pthread_t tid;
7     Pthread_create(&tid, NULL, thread, NULL);
8     Pthread_join(tid, NULL);
9     exit(0);
10 }
11
12 void *thread(void *vargp) /* thread routine */
13 {
14     printf("Hello, world!\n");
15     return NULL;
16 }
```

code/conc/hello.c

Figure 13.13: **`hello.c`: The Pthreads "Hello, world!" program.**

This is the first threaded program we have seen, so let us dissect it carefully. The code and local data for a thread is encapsulated in a *thread routine*. As shown by the prototype in line 2, each thread routine takes as input a single generic pointer and returns a generic pointer. If you want to pass multiple arguments to a thread routine, then you should put the arguments into a structure and pass a pointer to the structure. Similarly, if you want the thread routine to return multiple arguments, you can return a pointer to a structure.

Line 4 marks the beginning of the code for the main thread. The main thread declares a single local variable `tid`, which will be used to store the thread ID of the peer thread (line 6). The main thread creates a new peer thread by calling the `pthread_create` function (line 7). When the call to `pthread_create` returns, the main thread and the newly created peer thread are running concurrently, and `tid` contains the ID of

the new thread. The main thread waits for the peer thread to terminate with the call to `pthread_join` in line 8. Finally, the main thread calls `exit` (line 9), which terminates all threads (in this case just the main thread) currently running in the process.

Lines 12–16 define the thread routine for the peer thread. It simply prints a string, and then terminates the peer thread by executing the `return` statement in line 15.

13.3.3 Creating Threads

Threads create other threads by calling the `pthread_create` function.

```
#include <pthread.h>
typedef void *(func)(void *);

int pthread_create(pthread_t *tid, pthread_attr_t *attr, func *f, void *arg);
```
Returns 0 if OK, nonzero on error

The `pthread_create` function creates a new thread and runs the *thread routine* `f` in the context of the new thread and with an input argument of `arg`. The `attr` argument can be used to change the default attributes of the newly created thread. Changing these attributes is beyond our scope, and in our examples, we will always call `pthread_create` with a NULL `attr` argument.

When `pthread_create` returns, argument `tid` contains the ID of the newly created thread. The new thread can determine its own thread ID by calling the `pthread_self` function.

```
#include <pthread.h>

pthread_t pthread_self(void);
```
Returns thread ID of caller

13.3.4 Terminating Threads

A thread terminates in one of the following ways:

- The thread terminates *implicitly* when its top-level thread routine returns.
- The thread terminates *explicitly* by calling the `pthread_exit` function, which returns a pointer to the return value `thread_return`. If the main thread calls `pthread_exit`, it waits for all other peer threads to terminate, and then terminates the main thread and the entire process with a return value of `thread_return`.

```
#include <pthread.h>

int pthread_exit(void *thread_return);
```
Returns 0 if OK, nonzero on error

- Some peer thread calls the Unix `exit` function, which terminates the process and all threads associated with the process.
- Another peer thread terminates the current thread by calling the `pthread_cancel` function with the ID of the current thread.

```
#include <pthread.h>

int pthread_cancel(pthread_t tid);
```
Returns 0 if OK, nonzero on error

13.3.5 Reaping Terminated Threads

Threads wait for other threads to terminate by calling the `pthread_join` function.

```
#include <pthread.h>

int pthread_join(pthread_t tid, void **thread_return);
```
Returns 0 if OK, nonzero on error

The `pthread_join` function blocks until thread `tid` terminates, assigns the `(void *)` pointer returned by the thread routine to the location pointed to by `thread_return`, and then *reaps* any memory resources held by the terminated thread.

Notice that, unlike the Unix `wait` function, the `pthread_join` function can only wait for a specific thread to terminate. There is no way to instruct `pthread_wait` to wait for an arbitrary thread to terminate. This can complicate our code by forcing us to use other, less intuitive mechanisms to detect process termination. Indeed, Stevens argues convincingly that this is a bug in the specification [81].

13.3.6 Detaching Threads

At any point in time, a thread is *joinable* or *detached*. A joinable thread can be reaped and killed by other threads. Its memory resources (such as the stack) are not freed until it is reaped by another thread. In contrast, a detached thread cannot be reaped or killed by other threads. Its memory resources are freed automatically by the system when it terminates.

By default, threads are created joinable. In order to avoid memory leaks, each joinable thread should either be explicitly reaped by another thread, or detached by a call to the `pthread_detach` function.

```
#include <pthread.h>

int pthread_detach(pthread_t tid);
```
Returns 0 if OK, nonzero on error

The `pthread_detach` function detaches the joinable thread `tid`. Threads can detach themselves by calling `pthread_detach` with an argument of `pthread_self()`.

Although some of our examples will use joinable threads, there are good reasons to use detached threads in real programs. For example, a high-performance Web server might create a new peer thread each time it receives a connection request from a Web browser. Since each connection is handled independently by a separate thread, it is unnecessary – and indeed undesirable – for the server to explicitly wait for each peer thread to terminate. In this case, each peer thread should detach itself before it begins processing the request so that its memory resources can be reclaimed after it terminates.

13.3.7 Initializing Threads

The `pthread_once` function allows you to initialize the state associated with a thread routine.

```
#include <pthread.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;

int pthread_once(pthread_once_t *once_control, void (*init_routine)(void));
```
Always returns 0

The `once_control` variable is a global or static variable that is always initialized to PTHREAD_ONCE_INIT. The first time you call `pthread_once` with an argument of `once_control`, it invokes `init_routine`, which is a function with no input arguments that returns nothing. Subsequent calls to `pthread_once` with an argument of `pthread_once` do nothing. The `pthread_once` function is useful whenever you need to dynamically initialize global variables that are shared by multiple threads. We will look at an example in Section 13.6.

13.3.8 A Concurrent Server Based on Threads

Figure 13.14 shows the code for a concurrent echo server based on threads. The overall structure is similar to the process-based design. The main thread repeatedly waits for a connection request and then creates a peer thread to handle the request. While the code looks simple, there are a couple of general and somewhat subtle issues we need to look at more closely. The first issue is how to pass the connected descriptor to the

code/conc/echoservert.c

```
1  #include "csapp.h"
2
3  void echo(int connfd);
4  void *thread(void *vargp);
5
6  int main(int argc, char **argv)
7  {
8      int listenfd, *connfdp, port, clientlen=sizeof(struct sockaddr_in);
9      struct sockaddr_in clientaddr;
10     pthread_t tid;
11
12     if (argc != 2) {
13         fprintf(stderr, "usage: %s <port>\n", argv[0]);
14         exit(0);
15     }
16     port = atoi(argv[1]);
17
18     listenfd = Open_listenfd(port);
19     while (1) {
20         connfdp = Malloc(sizeof(int));
21         *connfdp = Accept(listenfd, (SA *) &clientaddr, &clientlen);
22         Pthread_create(&tid, NULL, thread, connfdp);
23     }
24 }
25
26 /* thread routine */
27 void *thread(void *vargp)
28 {
29     int connfd = *((int *)vargp);
30     Pthread_detach(pthread_self());
31     Free(vargp);
32     echo(connfd);
33     Close(connfd);
34     return NULL;
35 }
```

code/conc/echoservert.c

Figure 13.14: **Concurrent echo server based on threads.**

peer thread when we call `pthread_create`. The obvious approach is to pass a pointer to the descriptor, such as the following

```
connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
Pthread_create(&tid, NULL, thread, &connfd);
```

Then we have the peer thread dereference the pointer and assign it to a local variable, as follows

```
void *thread(void *vargp) {
    int connfd = *((int *)vargp);
    ...
}
```

This would be wrong, however, because it introduces a *race* between the assignment statement in the peer thread and the `accept` statement in the main thread. If the assignment statement completes before the next `accept`, then the local `connfd` variable in the peer thread gets the correct descriptor value. However, if the assignment completes *after* the `accept`, then the local `connfd` variable in the peer thread gets the descriptor number of the *next* connection. The unhappy result is that two threads are now performing input and output on the same descriptor. In order to avoid the potentially deadly race, we must assign each connected descriptor returned by `accept` to its own dynamically allocated memory block, as shown in lines 20–21. We will return to the issue of races in Section 13.7.4.

Another issue is avoiding memory leaks in the thread routine. Since we are not explicitly reaping threads, we must detach each thread so that its memory resources will be reclaimed when it terminates (line 30). Further, we must be careful to free the memory block that was allocated by the main thread (line 31).

Practice Problem 13.5:

In the process-based server in Figure 13.5, we were careful to close the connected descriptor in two places: the parent and child processes. However, in the threads-based server in Figure 13.14, we only closed the connected descriptor in one place: the peer thread. Why?

13.4 Shared Variables in Threaded Programs

From a programmer's perspective, one of the attractive aspects of threads is the ease with which multiple threads can share the same program variables. However, this sharing can be tricky. In order to write correctly threaded programs, we must have a clear understanding of what we mean by sharing and how it works.

There are some basic questions to work through in order to understand whether a variable in a C program is shared or not: (1) What is the underlying memory model for threads? (2) Given this model, how are instances of the variable mapped to memory? (3) Finally, how many threads reference each of these instances? The variable is *shared* if and only if multiple threads reference some instance of the variable.

To keep our discussion of sharing concrete, we will use the program in Figure 13.15 as a running example. Although somewhat contrived, it is nonetheless useful to study because it illustrates a number of subtle points about sharing. The example program consists of a main thread that creates two peer threads. The main thread passes a unique ID to each peer thread, which uses the ID to print a personalized message, along with a count of the total number of times that the thread routine has been invoked.