CHAPTER

12 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 Linux signal handlers are all familiar examples.

Thus far, we have treated concurrency mainly as a mechanism that the operating system 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 Linux 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:

- *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 11 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.
- *Computing in parallel on multi-core machines.* Many modern systems are equipped with multi-core processors that contain multiple CPUs. Applications that are partitioned into concurrent flows often run faster on multi-core machines than on uniprocessor machines because the flows execute in parallel rather than being interleaved.

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 three 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 11.4.9.

12.1 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 12.1. 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 shown in Figure 12.2, 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

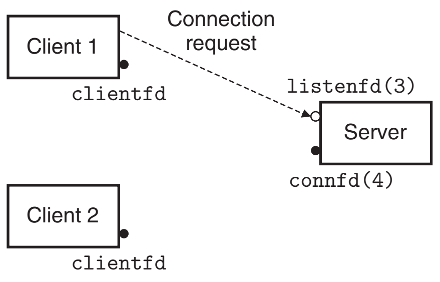


Figure 12.1
Step 1: Server accepts connection request from client.

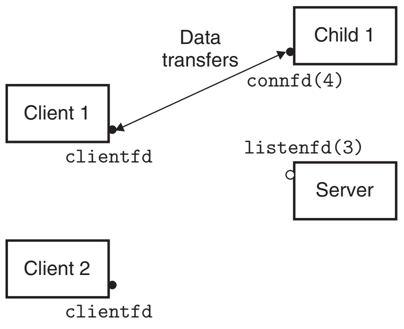


Figure 12.2
Step 2: Server forks a child process to service the client.

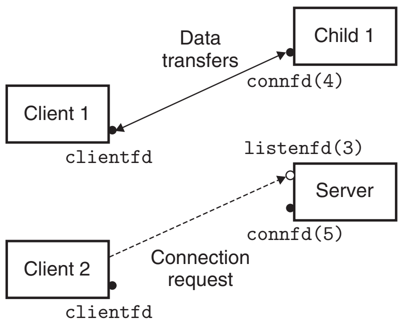


Figure 12.3
Step 3: Server accepts another connection request.

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.

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 12.3. The parent then forks another child, which begins servicing its client using connected descriptor 5, as shown in Figure 12.4. At this point, the parent is waiting for the next connection request and the two children are servicing their respective clients concurrently.

12.1.1 A Concurrent Server Based on Processes

Figure 12.5 shows the code for a concurrent echo server based on processes. The `echo` function called in line 29 comes from Figure 11.22. There are several important points to make about this server:

- First, servers typically run for long periods of time, so we must include a SIGCHLD handler that reaps zombie children (lines 4–9). Since SIGCHLD signals are blocked while the SIGCHLD handler is executing, and since Linux 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 33 and 30, respectively). As we have mentioned, this is especially im-

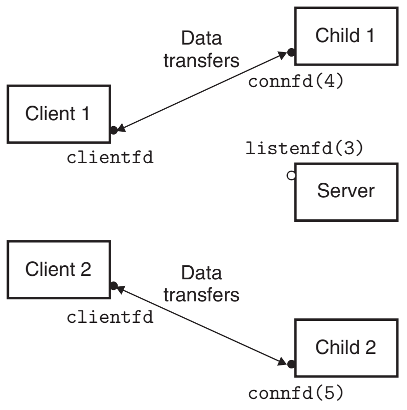


Figure 12.4
Step 4: Server forks another child to service the new client.

portant 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.

12.1.2 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 accidentally 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 the Aside on page 1013.) Another disadvantage of process-based designs is that they tend to be slower because the overhead for process control and IPC is high.

Practice Problem 12.1 (solution page 1072)

Figure 12.5 demonstrates a concurrent server in which the parent process creates a child process to handle each new connection request. Trace the value of the reference counter for the associated file table for Figure 12.5.

Practice Problem 12.2 (solution page 1072)

If we were to delete line 33 of Figure12.5, which closes the connected descriptor, the code would still be correct, in the sense that there would be no memory leak. Why?

code/conc/echoserverp.c

```c
#include "csapp.h"
void echo(int connfd);

void sigchld_handler(int sig)
{
    while (waitpid(-1, 0, WNOHANG) > 0)
        ;
    return;
}

int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }

    Signal(SIGCHLD, sigchld_handler);
    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        if (Fork() == 0) {
            Close(listenfd); /* Child closes its listening socket */
            echo(connfd);    /* Child services client */
            Close(connfd);   /* Child closes connection with client */
            exit(0);         /* Child exits */
        }
        Close(connfd); /* Parent closes connected socket (important!) */
    }
}
```

code/conc/echoserverp.c

Figure 12.5 **Concurrent echo server based on processes.** The parent forks a child to handle each new connection request.

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

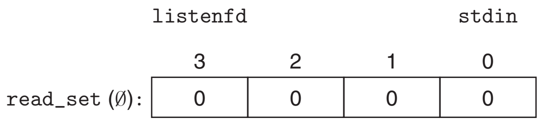
The select function manipulates sets of type fd_set, which are known as *descriptor sets*. Logically, we think of a descriptor set as a bit vector (introduced in Section 2.1) of size n:

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

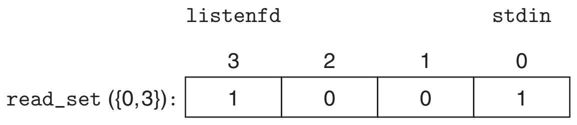
Each bit b_k corresponds to descriptor k. Descriptor k is a member of the descriptor set if and only if $b_k = 1$. You are only allowed to do three things with descriptor sets: (1) allocate them, (2) assign one variable of this type to another, and (3) modify and inspect them using the FD_ZERO, FD_SET,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



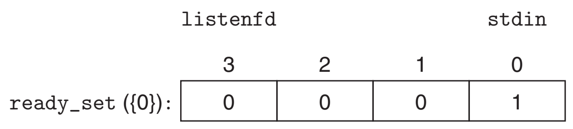
At this point, we begin the typical server loop. But instead of waiting for a connection request by calling the accept function, we call the select function, which blocks until either the listening descriptor or standard input is ready for reading (line 24). For example, here is the value of ready_set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

```c
#include "csapp.h"
void echo(int connfd);
void command(void);

int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    fd_set read_set, ready_set;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    listenfd = Open_listenfd(argv[1]);

    FD_ZERO(&read_set);              /* Clear read set */
    FD_SET(STDIN_FILENO, &read_set); /* Add stdin to read set */
    FD_SET(listenfd, &read_set);     /* Add listenfd to read set */

    while (1) {
        ready_set = read_set;
        Select(listenfd+1, &ready_set, NULL, NULL, NULL);
        if (FD_ISSET(STDIN_FILENO, &ready_set))
            command(); /* Read command line from stdin */
        if (FD_ISSET(listenfd, &ready_set)) {
            clientlen = sizeof(struct sockaddr_storage);
            connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
            echo(connfd); /* Echo client input until EOF */
            Close(connfd);
        }
    }
}

void command(void) {
    char buf[MAXLINE];
    if (!Fgets(buf, MAXLINE, stdin))
        exit(0); /* EOF */
    printf("%s", buf); /* Process the input command */
}
```

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:



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").

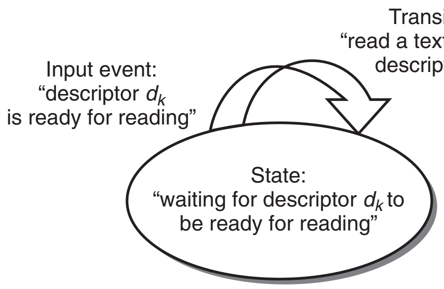

Transition:
"read a text line from descriptor d_k"
Input event:
"descriptor d_k is ready for reading"
State:
"waiting for descriptor d_k to be ready for reading"

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

```c
#include "csapp.h"

typedef struct { /* Represents a pool of connected descriptors */
    int maxfd;        /* Largest descriptor in read_set */
    fd_set read_set;  /* Set of all active descriptors */
    fd_set ready_set; /* Subset of descriptors ready for reading  */
    int nready;       /* Number of ready descriptors from select */
    int maxi;         /* High water index into client array */
    int clientfd[FD_SETSIZE];    /* Set of active descriptors */
    rio_t clientrio[FD_SETSIZE]; /* Set of active read buffers */
} pool;

int byte_cnt = 0; /* Counts total bytes received by server */

int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    static pool pool;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    listenfd = Open_listenfd(argv[1]);
    init_pool(listenfd, &pool);

    while (1) {
        /* Wait for listening/connected descriptor(s) to become ready */
        pool.ready_set = pool.read_set;
        pool.nready = Select(pool.maxfd+1, &pool.ready_set, NULL, NULL, NULL);

        /* If listening descriptor ready, add new client to pool */
        if (FD_ISSET(listenfd, &pool.ready_set)) {
            clientlen = sizeof(struct sockaddr_storage);
            connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
            add_client(connfd, &pool);
        }

        /* Echo a text line from each ready connected descriptor */
        check_clients(&pool);
    }
}
```

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

```c
void init_pool(int listenfd, pool *p)
{
    /* Initially, there are no connected descriptors */
    int i;
    p->maxi = -1;
    for (i=0; i< FD_SETSIZE; i++)
        p->clientfd[i] = -1;

    /* Initially, listenfd is only member of select read set */
    p->maxfd = listenfd;
    FD_ZERO(&p->read_set);
    FD_SET(listenfd, &p->read_set);
}
```

code/conc/echoservers.c

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

code/conc/echoservers.c

```c
void add_client(int connfd, pool *p)
{
    int i;
    p->nready--;
    for (i = 0; i < FD_SETSIZE; i++) /* Find an available slot */
        if (p->clientfd[i] < 0) {
            /* Add connected descriptor to the pool */
            p->clientfd[i] = connfd;
            Rio_readinitb(&p->clientrio[i], connfd);

            /* Add the descriptor to descriptor set */
            FD_SET(connfd, &p->read_set);

            /* Update max descriptor and pool high water mark */
            if (connfd > p->maxfd)
                p->maxfd = connfd;
            if (i > p->maxi)
                p->maxi = i;
            break;
        }
    if (i == FD_SETSIZE) /* Couldn't find an empty slot */
        app_error("add_client error: Too many clients");
}
```

code/conc/echoservers.c

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

code/conc/echoservers.c

```
void check_clients(pool *p)
{
    int i, connfd, n;
    char buf[MAXLINE];
    rio_t rio;

    for (i = 0; (i <= p->maxi) && (p->nready > 0); i++) {
        connfd = p->clientfd[i];
        rio = p->clientrio[i];

        /* If the descriptor is ready, echo a text line from it */
        if ((connfd > 0) && (FD_ISSET(connfd, &p->ready_set))) {
            p->nready--;
            if ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
                byte_cnt += n;
                printf("Server received %d (%d total) bytes on fd %d\n",
                       n, byte_cnt, connfd);
                Rio_writen(connfd, buf, n);
            }

            /* EOF detected, remove descriptor from pool */
            else {
                Close(connfd);
                FD_CLR(connfd, &p->read_set);
                p->clientfd[i] = -1;
            }
        }
    }
}
```

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.

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 they share the entire contents of the process virtual address space, including its code, data, heap, shared libraries, and open files.

12.3.1 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 12.12. 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, either 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.

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

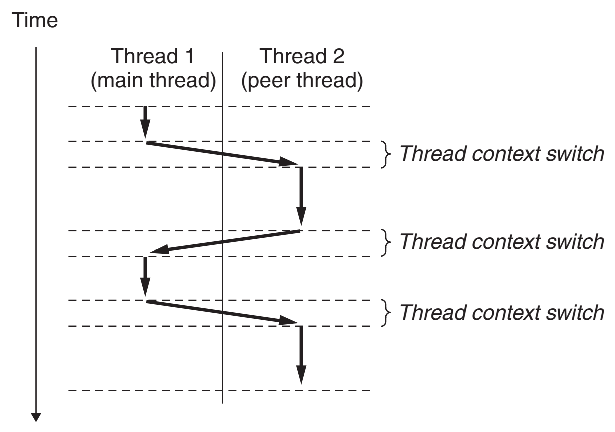


Figure 12.12 **Concurrent thread execution.**

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.

12.3.2 Posix Threads

Posix threads (Pthreads) is a standard interface for manipulating threads from C programs. It was adopted in 1995 and is available on all Linux 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 12.13 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`. This is the first threaded program we have seen, so let us dissect it carefully. The code and local data for a thread are 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

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

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.

12.3.3 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

12.3.4 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. 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>

void pthread_exit(void *thread_return);
```

Never returns

- Some peer thread calls the Linux `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

12.3.5 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 generic `(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 Linux `wait` function, the `pthread_join` function can only wait for a specific thread to terminate. There is no way to instruct `pthread_join` 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 [110].

12.3.6 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 be either 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.

12.3.7 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 the same `once_control` variable 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 12.5.5.

12.3.8 A Concurrent Server Based on Threads

Figure 12.14 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

code/conc/echoservert.c

```c
#include "csapp.h"

void echo(int connfd);
void *thread(void *vargp);

int main(int argc, char **argv)
{
    int listenfd, *connfdp;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    pthread_t tid;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    listenfd = Open_listenfd(argv[1]);

    while (1) {
        clientlen=sizeof(struct sockaddr_storage);
        connfdp = Malloc(sizeof(int));
        *connfdp = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        Pthread_create(&tid, NULL, thread, connfdp);
    }
}

/* Thread routine */
void *thread(void *vargp)
{
    int connfd = *((int *)vargp);
    Pthread_detach(pthread_self());
    Free(vargp);
    echo(connfd);
    Close(connfd);
    return NULL;
}
```

code/conc/echoservert.c

Figure 12.14 **Concurrent echo server based on threads.**

the connected descriptor to the peer thread when we call `pthread_create`. The obvious approach is to pass a pointer to the descriptor, as in 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 21–22. We will return to the issue of races in Section 12.7.4.

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 31). Further, we must be careful to free the memory block that was allocated by the main thread (line 32).

Practice Problem 12.5 **(solution page 1072)**

In the process-based server in Figure 12.5, we observed that there is no memory leak and the code remains correct even when line 33 is deleted. In the threads-based server in Figure 12.14, are there any chances of memory leak if lines 31 or 32 are deleted. Why?

12.4 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 *code/conc/sharing.c*

```c
#include "csapp.h"
#define N 2
void *thread(void *vargp);

char **ptr;  /* Global variable */

int main()
{
    int i;
    pthread_t tid;
    char *msgs[N] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < N; i++)
        Pthread_create(&tid, NULL, thread, (void *)i);
    Pthread_exit(NULL);
}

void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int cnt = 0;
    printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
```

code/conc/sharing.c

Figure 12.15 Example program that illustrates different aspects of sharing.

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 12.15 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.

12.4.1 Threads Memory Model

A pool of concurrent threads runs in the context of a process. Each thread has its own separate *thread context*, which includes a thread ID, stack, stack pointer, program counter, condition codes, and general-purpose register values. Each thread shares the rest of the process context with the other threads. This includes the entire user virtual address space, which consists of read-only text (code), read/write data, the heap, and any shared library code and data areas. The threads also share the same set of open files.

In an operational sense, it is impossible for one thread to read or write the register values of another thread. On the other hand, any thread can access any location in the shared virtual memory. If some thread modifies a memory location, then every other thread will eventually see the change if it reads that location. Thus, registers are never shared, whereas virtual memory is always shared.

The memory model for the separate thread stacks is not as clean. These stacks are contained in the stack area of the virtual address space and are *usually* accessed independently by their respective threads. We say *usually* rather than *always*, because different thread stacks are not protected from other threads. So if a thread somehow manages to acquire a pointer to another thread's stack, then it can read and write any part of that stack. Our example program shows this in line 26, where the peer threads reference the contents of the main thread's stack indirectly through the global `ptr` variable.

12.4.2 Mapping Variables to Memory

Variables in threaded C programs are mapped to virtual memory according to their storage classes:

Global variables. A *global variable* is any variable declared outside of a function. At run time, the read/write area of virtual memory contains exactly one instance of each global variable that can be referenced by any thread. For example, the global `ptr` variable declared in line 5 has one run-time instance in the read/write area of virtual memory. When there is only one instance of a variable, we will denote the instance by simply using the variable name—in this case, `ptr`.

Local automatic variables. A *local automatic variable* is one that is declared inside a function without the `static` attribute. At run time, each thread's stack contains its own instances of any local automatic variables. This is true even if multiple threads execute the same thread routine. For example, there is one instance of the local variable `tid`, and it resides on the stack of the main thread. We will denote this instance as `tid.m`. As another example, there are two instances of the local variable `myid`, one instance on the stack of peer thread 0 and the other on the stack of peer thread 1. We will denote these instances as `myid.p0` and `myid.p1`, respectively.

Local static variables. A *local static variable* is one that is declared inside a function with the `static` attribute. As with global variables, the read/write area of virtual memory contains exactly one instance of each local static variable declared in a program. For example, even though each peer thread in our example program declares `cnt` in line 25, at run time there is only one instance of `cnt` residing in the read/write area of virtual memory. Each peer thread reads and writes this instance.

12.4.3 Shared Variables

We say that a variable v is *shared* if and only if one of its instances is referenced by more than one thread. For example, variable `cnt` in our example program is shared because it has only one run-time instance and this instance is referenced by both peer threads. On the other hand, `myid` is not shared, because each of its two instances is referenced by exactly one thread. However, it is important to realize that local automatic variables such as `msgs` can also be shared.

Practice Problem 12.6 (solution page 1072)

A. Using the analysis from Section 12.4, fill each entry in the following table with "Yes" or "No" for the example program in Figure 12.15. In the first column, the notation $v.t$ denotes an instance of variable v residing on the local stack for thread t, where t is either `m` (main thread), `p0` (peer thread 0), or `p1` (peer thread 1).

Variable	Referenced by		
instance	main thread?	peer thread 0?	peer thread 1?
`ptr`	______	______	______
`cnt`	______	______	______
`i.m`	______	______	______
`msgs.m`	______	______	______
`myid.p0`	______	______	______
`myid.p1`	______	______	______

B. Given the analysis in part A, which of the variables `ptr`, `cnt`, `i`, `msgs`, and `myid` are shared?

12.5 Synchronizing Threads with Semaphores

Shared variables can be convenient, but they introduce the possibility of nasty *synchronization errors*. Consider the `badcnt.c` program in Figure 12.16, which creates two threads, each of which increments a global shared counter variable called `cnt`.

Since each thread increments the counter `niters` times, we expect its final value to be 2 × `niters`. This seems quite simple and straightforward. However, when we run `badcnt.c` on our Linux system, we not only get wrong answers, we get different answers each time!

code/conc/badcnt.c

```
/* WARNING: This code is buggy! */
#include "csapp.h"

void *thread(void *vargp);  /* Thread routine prototype */

/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    /* Check input argument */
    if (argc != 2) {
        printf("usage: %s <niters>\n", argv[0]);
        exit(0);
    }
    niters = atoi(argv[1]);

    /* Create threads and wait for them to finish */
    Pthread_create(&tid1, NULL, thread, &niters);
    Pthread_create(&tid2, NULL, thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}

/* Thread routine */
void *thread(void *vargp)
{
    long i, niters = *((long *)vargp);

    for (i = 0; i < niters; i++)
        cnt++;

    return NULL;
}
```

code/conc/badcnt.c

Figure 12.16 `badcnt.c`: **An improperly synchronized counter program.**

```
linux>  ./badcnt 1000000
BOOM! cnt=1445085

linux>  ./badcnt 1000000
BOOM! cnt=1915220

linux>  ./badcnt 1000000
BOOM! cnt=1404746
```

So what went wrong? To understand the problem clearly, we need to study the assembly code for the counter loop (lines 40–41), as shown in Figure 12.17. We will find it helpful to partition the loop code for thread i into five parts:

H_i: The block of instructions at the head of the loop

L_i: The instruction that loads the shared variable `cnt` into the accumulator register `%rdx`$_i$, where `%rdx`$_i$ denotes the value of register `%rdx` in thread i

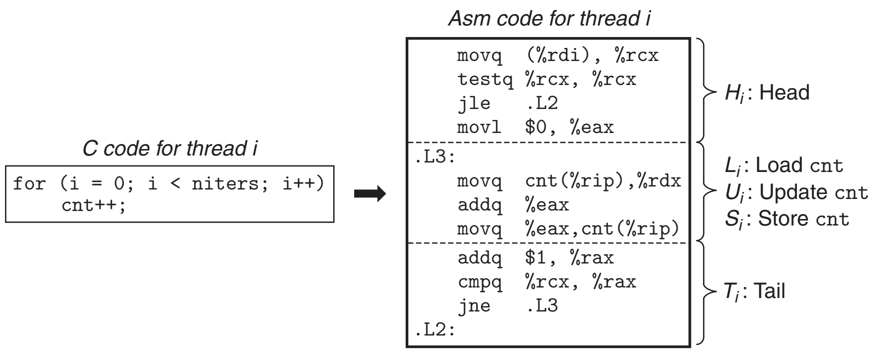
U_i: The instruction that updates (increments) `%rdx`$_i$

S_i: The instruction that stores the updated value of `%rdx`$_i$ back to the shared variable `cnt`

T_i: The block of instructions at the tail of the loop

Notice that the head and tail manipulate only local stack variables, while L_i, U_i, and S_i manipulate the contents of the shared counter variable.

When the two peer threads in `badcnt.c` run concurrently on a uniprocessor, the machine instructions are completed one after the other in some order. Thus, each concurrent execution defines some total ordering (or interleaving) of the instructions in the two threads. Unfortunately, some of these orderings will produce correct results, but others will not.



```
for (i = 0; i < niters; i++)
    cnt++;
```

```
    movq  (%rdi), %rcx
    testq %rcx, %rcx
    jle   .L2
    movl  $0, %eax
.L3:
    movq  cnt(%rip),%rdx
    addq  %eax
    movq  %eax,cnt(%rip)
    addq  $1, %rax
    cmpq  %rcx, %rax
    jne   .L3
.L2:
```

Figure 12.17 **Assembly code for the counter loop (lines 40–41) in `badcnt.c`.**

(a) Correct ordering

Step	Thread	Instr.	`%rdx`$_1$	`%rdx`$_2$	`cnt`
1	1	H_1	—	—	0
2	1	L_1	0	—	0
3	1	U_1	1	—	0
4	1	S_1	1	—	1
5	2	H_2	—	—	1
6	2	L_2	—	1	1
7	2	U_2	—	2	1
8	2	S_2	—	2	2
9	2	T_2	—	2	2
10	1	T_1	1	—	2

(b) Incorrect ordering

Step	Thread	Instr.	`%rdx`$_1$	`%rdx`$_2$	`cnt`
1	1	H_1	—	—	0
2	1	L_1	0	—	0
3	1	U_1	1	—	0
4	2	H_2	—	—	0
5	2	L_2	—	0	0
6	1	S_1	1	—	1
7	1	T_1	1	—	1
8	2	U_2	—	1	1
9	2	S_2	—	1	1
10	2	T_2	—	1	1

Figure 12.18 **Instruction orderings for the first loop iteration in `badcnt.c`.**

Here is the crucial point: *In general, there is no way for you to predict whether the operating system will choose a correct ordering for your threads.* For example, Figure 12.18(a) shows the step-by-step operation of a correct instruction ordering. After each thread has updated the shared variable `cnt`, its value in memory is 2, which is the expected result.

On the other hand, the ordering in Figure 12.18(b) produces an incorrect value for `cnt`. The problem occurs because thread 2 loads `cnt` in step 5, after thread 1 loads `cnt` in step 2 but before thread 1 stores its updated value in step 6. Thus, each thread ends up storing an updated counter value of 1. We can clarify these notions of correct and incorrect instruction orderings with the help of a device known as a *progress graph*, which we introduce in the next section.

Practice Problem 12.7 (solution page 1073)

Complete the table for the following instruction ordering of `badcnt.c`:

Step	Thread	Instr.	`%rdx`$_1$	`%rdx`$_2$	`cnt`
1	1	H_1	—	—	0
2	1	L_1	______	______	______
3	2	H_2	______	______	______
4	2	L_2	______	______	______
5	2	U_2	______	______	______
6	2	S_2	______	______	______
7	1	U_1	______	______	______
8	1	S_1	______	______	______
9	1	T_1	______	______	______

10	2	T_2			

Does this ordering result in a correct value for `cnt`?

12.5.1 Progress Graphs

A *progress graph* models the execution of n concurrent threads as a trajectory through an n-dimensional Cartesian space. Each axis k corresponds to the progress of thread k. Each point $(I_1, I_2, \ldots, I_n)$ represents the state where thread k ($k = 1, \ldots, n$) has completed instruction I_k. The origin of the graph corresponds to the *initial state* where none of the threads has yet completed an instruction.

Figure 12.19 shows the two-dimensional progress graph for the first loop iteration of the `badcnt.c` program. The horizontal axis corresponds to thread 1, the vertical axis to thread 2. Point (L_1, S_2) corresponds to the state where thread 1 has completed L_1 and thread 2 has completed S_2.

A progress graph models instruction execution as a *transition* from one state to another. A transition is represented as a directed edge from one point to an adjacent point. Legal transitions move to the right (an instruction in thread 1 completes) or up (an instruction in thread 2 completes). Two instructions cannot complete at the same time—diagonal transitions are not allowed. Programs never run backward so transitions that move down or to the left are not legal either.

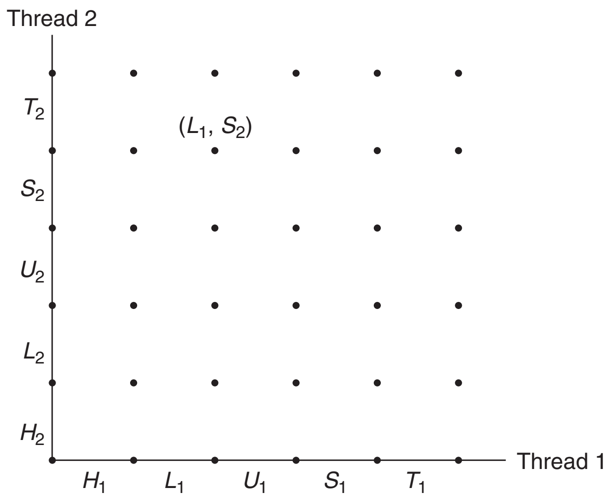


Figure 12.19
Progress graph for the first loop iteration of `badcnt.c`.

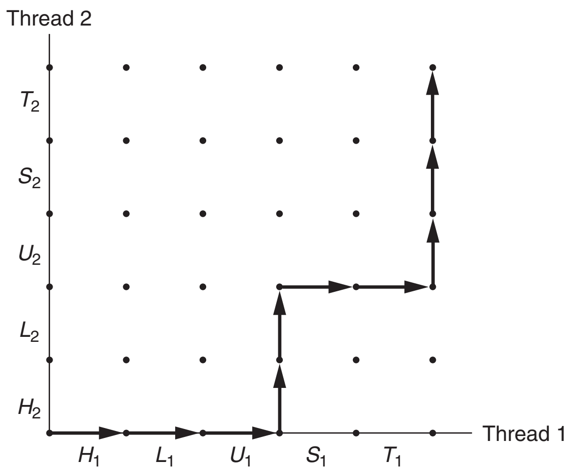


Figure 12.20
An example trajectory.

The execution history of a program is modeled as a *trajectory* through the state space. Figure 12.20 shows the trajectory that corresponds to the following instruction ordering:

$$H_1, L_1, U_1, H_2, L_2, S_1, T_1, U_2, S_2, T_2$$

For thread i, the instructions (L_i, U_i, S_i) that manipulate the contents of the shared variable `cnt` constitute a *critical section* (with respect to shared variable `cnt`) that should not be interleaved with the critical section of the other thread. In other words, we want to ensure that each thread has *mutually exclusive access* to the shared variable while it is executing the instructions in its critical section. The phenomenon in general is known as *mutual exclusion*.

On the progress graph, the intersection of the two critical sections defines a region of the state space known as an *unsafe region*. Figure 12.21 shows the unsafe region for the variable `cnt`. Notice that the unsafe region abuts, but does not include, the states along its perimeter. For example, states (H_1, H_2) and (S_1, U_2) abut the unsafe region, but they are not part of it. A trajectory that skirts the unsafe region is known as a *safe trajectory*. Conversely, a trajectory that touches any part of the unsafe region is an *unsafe trajectory*. Figure 12.21 shows examples of safe and unsafe trajectories through the state space of our example `badcnt.c` program. The upper trajectory skirts the unsafe region along its left and top sides, and thus is safe. The lower trajectory crosses the unsafe region, and thus is unsafe.

Any safe trajectory will correctly update the shared counter. In order to guarantee correct execution of our example threaded program—and indeed any concurrent program that shares global data structures—we must somehow *synchronize* the threads so that they always have a safe trajectory. A classic approach is based on the idea of a semaphore, which we introduce next.

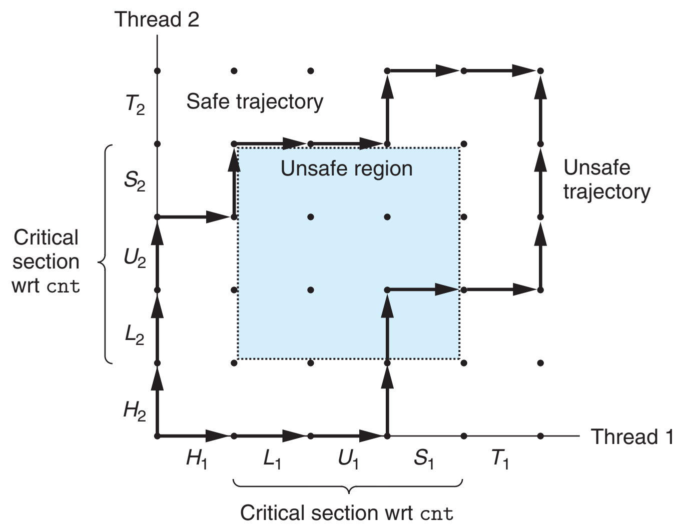


Figure 12.21
Safe and unsafe trajectories. The intersection of the critical regions forms an unsafe region. Trajectories that skirt the unsafe region correctly update the counter variable.

Practice Problem 12.8 **(solution page 1074)**

Using the progress graph in Figure 12.21, classify the following trajectories as either *safe* or *unsafe*.

A. $H_1, L_1, U_1, S_1, H_2, L_2, U_2, S_2, T_2, T_1$

B. $H_2, L_2, H_1, L_1, U_1, S_1, T_1, U_2, S_2, T_2$

C. $H_1, H_2, L_2, U_2, S_2, L_1, U_1, S_1, T_1, T_2$

12.5.2 Semaphores

Edsger Dijkstra, a pioneer of concurrent programming, proposed a classic solution to the problem of synchronizing different execution threads based on a special type of variable called a *semaphore*. A semaphore, s, is a global variable with a nonnegative integer value that can only be manipulated by two special operations, called P and V:

$P(s)$: If s is nonzero, then P decrements s and returns immediately. If s is zero, then suspend the thread until s becomes nonzero and the thread is restarted by a V operation. After restarting, the P operation decrements s and returns control to the caller.

$V(s)$: The V operation increments s by 1. If there are any threads blocked at a P operation waiting for s to become nonzero, then the V operation restarts exactly one of these threads, which then completes its P operation by decrementing s.

Aside Origin of the names P and V

Edsger Dijkstra (1930–2002) was originally from the Netherlands. The names P and V come from the Dutch words *proberen* (to test) and *verhogen* (to increment).

The test and decrement operations in P occur indivisibly, in the sense that once the semaphore s becomes nonzero, the decrement of s occurs without interruption. The increment operation in V also occurs indivisibly, in that it loads, increments, and stores the semaphore without interruption. Notice that the definition of V does *not* define the order in which waiting threads are restarted. The only requirement is that the V must restart exactly one waiting thread. *Thus, when several threads are waiting at a semaphore, you cannot predict which one will be restarted as a result of the* V.

The definitions of P and V ensure that a running program can never enter a state where a properly initialized semaphore has a negative value. This property, known as the *semaphore invariant*, provides a powerful tool for controlling the trajectories of concurrent programs, as we shall see in the next section.

The Posix standard defines a variety of functions for manipulating semaphores.

```
#include <semaphore.h>

int sem_init(sem_t *sem, 0, unsigned int value);
int sem_wait(sem_t *s);   /* P(s) */
int sem_post(sem_t *s);   /* V(s) */
```

Returns: 0 if OK, −1 on error

The `sem_init` function initializes semaphore `sem` to `value`. Each semaphore must be initialized before it can be used. For our purposes, the middle argument is always 0. Programs perform P and V operations by calling the `sem_wait` and `sem_post` functions, respectively. For conciseness, we prefer to use the following equivalent P and V wrapper functions instead:

```
#include "csapp.h"

void P(sem_t *s);   /* Wrapper function for sem_wait */
void V(sem_t *s);   /* Wrapper function for sem_post */
```

Returns: nothing

12.5.3 Using Semaphores for Mutual Exclusion

Semaphores provide a convenient way to ensure mutually exclusive access to shared variables. The basic idea is to associate a semaphore s, initially 1, with

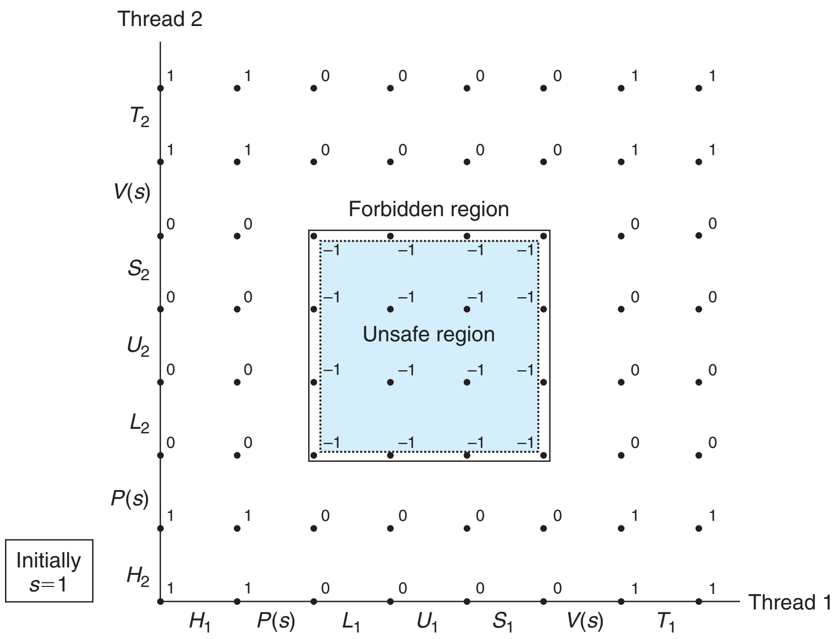


Figure 12.22 Using semaphores for mutual exclusion. The infeasible states where $s < 0$ define a *forbidden region* that surrounds the unsafe region and prevents any feasible trajectory from touching the unsafe region.

each shared variable (or related set of shared variables) and then surround the corresponding critical section with $P(s)$ and $V(s)$ operations.

A semaphore that is used in this way to protect shared variables is called a *binary semaphore* because its value is always 0 or 1. Binary semaphores whose purpose is to provide mutual exclusion are often called *mutexes*. Performing a P operation on a mutex is called *locking* the mutex. Similarly, performing the V operation is called *unlocking* the mutex. A thread that has locked but not yet unlocked a mutex is said to be *holding* the mutex. A semaphore that is used as a counter for a set of available resources is called a *counting semaphore*.

The progress graph in Figure 12.22 shows how we would use binary semaphores to properly synchronize our example counter program.

Each state is labeled with the value of semaphore s in that state. The crucial idea is that this combination of P and V operations creates a collection of states, called a *forbidden region*, where $s < 0$. Because of the semaphore invariant, no feasible trajectory can include one of the states in the forbidden region. And since the forbidden region completely encloses the unsafe region, no feasible trajectory can touch any part of the unsafe region. Thus, every feasible trajectory is safe, and regardless of the ordering of the instructions at run time, the program correctly increments the counter.

Aside Limitations of progress graphs

Progress graphs give us a nice way to visualize concurrent program execution on uniprocessors and to understand why we need synchronization. However, they do have limitations, particularly with respect to concurrent execution on multiprocessors, where a set of CPU/cache pairs share the same main memory. Multiprocessors behave in ways that cannot be explained by progress graphs. In particular, a multiprocessor memory system can be in a state that does not correspond to any trajectory in a progress graph. Regardless, the message remains the same: always synchronize accesses to your shared variables, regardless if you're running on a uniprocessor or a multiprocessor.

In an operational sense, the forbidden region created by the *P* and *V* operations makes it impossible for multiple threads to be executing instructions in the enclosed critical region at any point in time. In other words, the semaphore operations ensure mutually exclusive access to the critical region.

Putting it all together, to properly synchronize the example counter program in Figure 12.16 using semaphores, we first declare a semaphore called `mutex`:

```
volatile long cnt = 0; /* Counter */
sem_t mutex;           /* Semaphore that protects counter */
```

and then we initialize it to unity in the main routine:

```
Sem_init(&mutex, 0, 1);  /* mutex = 1 */
```

Finally, we protect the update of the shared `cnt` variable in the thread routine by surrounding it with *P* and *V* operations:

```
for (i = 0; i < niters; i++) {
    P(&mutex);
    cnt++;
    V(&mutex);
}
```

When we run the properly synchronized program, it now produces the correct answer each time.

```
linux> ./goodcnt 1000000
OK cnt=2000000

linux> ./goodcnt 1000000
OK cnt=2000000
```

12.5.4 Using Semaphores to Schedule Shared Resources

Another important use of semaphores, besides providing mutual exclusion, is to schedule accesses to shared resources. In this scenario, a thread uses a semaphore

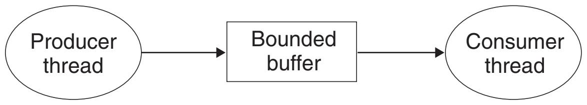


Figure 12.23 **Producer-consumer problem.** The producer generates items and inserts them into a bounded buffer. The consumer removes items from the buffer and then consumes them.

operation to notify another thread that some condition in the program state has become true. Two classical and useful examples are the *producer-consumer* and *readers-writers* problems.

Producer-Consumer Problem

The *producer-consumer* problem is shown in Figure 12.23. A producer and consumer thread share a *bounded buffer* with *n slots*. The producer thread repeatedly produces new *items* and inserts them in the buffer. The consumer thread repeatedly removes items from the buffer and then consumes (uses) them. Variants with multiple producers and consumers are also possible.

Since inserting and removing items involves updating shared variables, we must guarantee mutually exclusive access to the buffer. But guaranteeing mutual exclusion is not sufficient. We also need to schedule accesses to the buffer. If the buffer is full (there are no empty slots), then the producer must wait until a slot becomes available. Similarly, if the buffer is empty (there are no available items), then the consumer must wait until an item becomes available.

Producer-consumer interactions occur frequently in real systems. For example, in a multimedia system, the producer might encode video frames while the consumer decodes and renders them on the screen. The purpose of the buffer is to reduce jitter in the video stream caused by data-dependent differences in the encoding and decoding times for individual frames. The buffer provides a reservoir of slots to the producer and a reservoir of encoded frames to the consumer. Another common example is the design of graphical user interfaces. The producer detects mouse and keyboard events and inserts them in the buffer. The consumer removes the events from the buffer in some priority-based manner and paints the screen.

In this section, we will develop a simple package, called SBUF, for building producer-consumer programs. In the next section, we look at how to use it to build an interesting concurrent server based on prethreading. SBUF manipulates bounded buffers of type `sbuf_t` (Figure 12.24). Items are stored in a dynamically allocated integer array (`buf`) with `n` items. The `front` and `rear` indices keep track of the first and last items in the array. Three semaphores synchronize access to the buffer. The `mutex` semaphore provides mutually exclusive buffer access. Semaphores `slots` and `items` are counting semaphores that count the number of empty slots and available items, respectively.

code/conc/sbuf.h

```
typedef struct {
    int *buf;          /* Buffer array */
    int n;             /* Maximum number of slots */
    int front;         /* buf[(front+1)%n] is first item */
    int rear;          /* buf[rear%n] is last item */
    sem_t mutex;       /* Protects accesses to buf */
    sem_t slots;       /* Counts available slots */
    sem_t items;       /* Counts available items */
} sbuf_t;
```

code/conc/sbuf.h

Figure 12.24 `sbuf_t`**: Bounded buffer used by the SBUF package.**

Figure 12.25 shows the implementation of the SBUF package. The `sbuf_init` function allocates heap memory for the buffer, sets `front` and `rear` to indicate an empty buffer, and assigns initial values to the three semaphores. This function is called once, before calls to any of the other three functions. The `sbuf_deinit` function frees the buffer storage when the application is through using it. The `sbuf_insert` function waits for an available slot, locks the mutex, adds the item, unlocks the mutex, and then announces the availability of a new item. The `sbuf_remove` function is symmetric. After waiting for an available buffer item, it locks the mutex, removes the item from the front of the buffer, unlocks the mutex, and then signals the availability of a new slot.

Practice Problem 12.9 (solution page 1074)

Let p denote the number of producers, c the number of consumers, and n the buffer size in units of items. For each of the following scenarios, indicate whether the mutex semaphore in `sbuf_insert` and `sbuf_remove` is necessary or not.

A. $p = 1, c = 1, n > 1$

B. $p = 1, c = 1, n = 1$

C. $p > 1, c > 1, n = 1$

Readers-Writers Problem

The *readers-writers problem* is a generalization of the mutual exclusion problem. A collection of concurrent threads is accessing a shared object such as a data structure in main memory or a database on disk. Some threads only read the object, while others modify it. Threads that modify the object are called *writers*. Threads that only read it are called *readers*. Writers must have exclusive access to the object, but readers may share the object with an unlimited number of other readers. In general, there are an unbounded number of concurrent readers and writers.

code/conc/sbuf.c

```c
#include "csapp.h"
#include "sbuf.h"

/* Create an empty, bounded, shared FIFO buffer with n slots */
void sbuf_init(sbuf_t *sp, int n)
{
    sp->buf = Calloc(n, sizeof(int));
    sp->n = n;                       /* Buffer holds max of n items */
    sp->front = sp->rear = 0;        /* Empty buffer iff front == rear */
    Sem_init(&sp->mutex, 0, 1);      /* Binary semaphore for locking */
    Sem_init(&sp->slots, 0, n);      /* Initially, buf has n empty slots */
    Sem_init(&sp->items, 0, 0);      /* Initially, buf has zero data items */
}

/* Clean up buffer sp */
void sbuf_deinit(sbuf_t *sp)
{
    Free(sp->buf);
}

/* Insert item onto the rear of shared buffer sp */
void sbuf_insert(sbuf_t *sp, int item)
{
    P(&sp->slots);                          /* Wait for available slot */
    P(&sp->mutex);                          /* Lock the buffer */
    sp->buf[(++sp->rear)%(sp->n)] = item;   /* Insert the item */
    V(&sp->mutex);                          /* Unlock the buffer */
    V(&sp->items);                          /* Announce available item */
}

/* Remove and return the first item from buffer sp */
int sbuf_remove(sbuf_t *sp)
{
    int item;
    P(&sp->items);                          /* Wait for available item */
    P(&sp->mutex);                          /* Lock the buffer */
    item = sp->buf[(++sp->front)%(sp->n)];  /* Remove the item */
    V(&sp->mutex);                          /* Unlock the buffer */
    V(&sp->slots);                          /* Announce available slot */
    return item;
}
```

code/conc/sbuf.c

Figure 12.25 **SBUF: A package for synchronizing concurrent access to bounded buffers.**

Readers-writers interactions occur frequently in real systems. For example, in an online airline reservation system, an unlimited number of customers are allowed to concurrently inspect the seat assignments, but a customer who is booking a seat must have exclusive access to the database. As another example, in a multi-threaded caching Web proxy, an unlimited number of threads can fetch existing pages from the shared page cache, but any thread that writes a new page to the cache must have exclusive access.

The readers-writers problem has several variations, each based on the priorities of readers and writers. The *first readers-writers problem*, which favors readers, requires that no reader be kept waiting unless a writer has already been granted permission to use the object. In other words, no reader should wait simply because a writer is waiting. The *second readers-writers problem*, which favors writers, requires that once a writer is ready to write, it performs its write as soon as possible. Unlike the first problem, a reader that arrives after a writer must wait, even if the writer is also waiting.

Figure 12.26 shows a solution to the first readers-writers problem. Like the solutions to many synchronization problems, it is subtle and deceptively simple. The `w` semaphore controls access to the critical sections that access the shared object. The `mutex` semaphore protects access to the shared `readcnt` variable, which counts the number of readers currently in the critical section. A writer locks the `w` mutex each time it enters the critical section and unlocks it each time it leaves. This guarantees that there is at most one writer in the critical section at any point in time. On the other hand, only the first reader to enter the critical section locks `w`, and only the last reader to leave the critical section unlocks it. The `w` mutex is ignored by readers who enter and leave while other readers are present. This means that as long as a single reader holds the `w` mutex, an unbounded number of readers can enter the critical section unimpeded.

A correct solution to either of the readers-writers problems can result in *starvation*, where a thread blocks indefinitely and fails to make progress. For example, in the solution in Figure 12.26, a writer could wait indefinitely while a stream of readers arrived.

Practice Problem 12.10 **(solution page 1074)**

The solution to the first readers-writers problem in Figure 12.26 gives priority to readers, but this priority is weak in the sense that a writer leaving its critical section might restart a waiting writer instead of a waiting reader. Describe a scenario where this weak priority would allow a collection of writers to starve a reader.

12.5.5 Putting It Together: A Concurrent Server Based on Prethreading

We have seen how semaphores can be used to access shared variables and to schedule accesses to shared resources. To help you understand these ideas more clearly, let us apply them to a concurrent server based on a technique called *prethreading*.

```
/* Global variables */
int readcnt;    /* Initially = 0 */
sem_t mutex, w; /* Both initially = 1 */

void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&mutex);

        /* Critical section */
        /* Reading happens  */

        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
    }
}

void writer(void)
{
    while (1) {
        P(&w);

        /* Critical section */
        /* Writing happens  */

        V(&w);
    }
}
```

Figure 12.26 **Solution to the first readers-writers problem.** Favors readers over writers.

In the concurrent server in Figure 12.14, we created a new thread for each new client. A disadvantage of this approach is that we incur the nontrivial cost of creating a new thread for each new client. A server based on prethreading tries to reduce this overhead by using the producer-consumer model shown in Figure 12.27. The server consists of a main thread and a set of worker threads. The main thread repeatedly accepts connection requests from clients and places

Aside Other synchronization mechanisms

We have shown you how to synchronize threads using semaphores, mainly because they are simple, classical, and have a clean semantic model. But you should know that other synchronization techniques exist as well. For example, Java threads are synchronized with a mechanism called a *Java monitor* [48], which provides a higher-level abstraction of the mutual exclusion and scheduling capabilities of semaphores; in fact, monitors can be implemented with semaphores. As another example, the Pthreads interface defines a set of synchronization operations on *mutex* and *condition* variables. Pthreads mutexes are used for mutual exclusion. Condition variables are used for scheduling accesses to shared resources, such as the bounded buffer in a producer-consumer program.

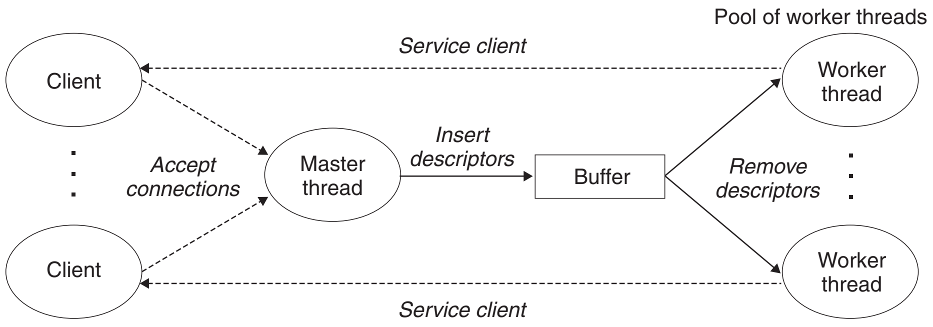


Figure 12.27 Organization of a prethreaded concurrent server. A set of existing threads repeatedly remove and process connected descriptors from a bounded buffer.

the resulting connected descriptors in a bounded buffer. Each worker thread repeatedly removes a descriptor from the buffer, services the client, and then waits for the next descriptor.

Figure 12.28 shows how we would use the Sbuf package to implement a prethreaded concurrent echo server. After initializing buffer `sbuf` (line 24), the main thread creates the set of worker threads (lines 25–26). Then it enters the infinite server loop, accepting connection requests and inserting the resulting connected descriptors in `sbuf`. Each worker thread has a very simple behavior. It waits until it is able to remove a connected descriptor from the buffer (line 39) and then calls the `echo_cnt` function to echo client input.

The `echo_cnt` function in Figure 12.29 is a version of the `echo` function from Figure 11.22 that records the cumulative number of bytes received from all clients in a global variable called `byte_cnt`. This is interesting code to study because it shows you a general technique for initializing packages that are called from thread routines. In our case, we need to initialize the `byte_cnt` counter and the `mutex` semaphore. One approach, which we used for the Sbuf and Rio packages, is to require the main thread to explicitly call an initialization function. Another approach, shown here, uses the `pthread_once` function (line 19) to call *code/conc/echoservert-pre.c*

```c
#include "csapp.h"
#include "sbuf.h"
#define NTHREADS  4
#define SBUFSIZE  16

void echo_cnt(int connfd);
void *thread(void *vargp);

sbuf_t sbuf; /* Shared buffer of connected descriptors */

int main(int argc, char **argv)
{
    int i, listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    pthread_t tid;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    listenfd = Open_listenfd(argv[1]);

    sbuf_init(&sbuf, SBUFSIZE);
    for (i = 0; i < NTHREADS; i++)  /* Create worker threads */
        Pthread_create(&tid, NULL, thread, NULL);

    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        sbuf_insert(&sbuf, connfd); /* Insert connfd in buffer */
    }
}

void *thread(void *vargp)
{
    Pthread_detach(pthread_self());
    while (1) {
        int connfd = sbuf_remove(&sbuf); /* Remove connfd from buffer */
        echo_cnt(connfd);                /* Service client */
        Close(connfd);
    }
}
```

code/conc/echoservert-pre.c

Figure 12.28 **A prethreaded concurrent echo server.** The server uses a producer-consumer model with one producer and multiple consumers.

code/conc/echo-cnt.c

```
1   #include "csapp.h"
2
3   static int byte_cnt;  /* Byte counter */
4   static sem_t mutex;   /* and the mutex that protects it */
5
6   static void init_echo_cnt(void)
7   {
8       Sem_init(&mutex, 0, 1);
9       byte_cnt = 0;
10  }
11
12  void echo_cnt(int connfd)
13  {
14      int n;
15      char buf[MAXLINE];
16      rio_t rio;
17      static pthread_once_t once = PTHREAD_ONCE_INIT;
18
19      Pthread_once(&once, init_echo_cnt);
20      Rio_readinitb(&rio, connfd);
21      while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
22          P(&mutex);
23          byte_cnt += n;
24          printf("server received %d (%d total) bytes on fd %d\n",
25                 n, byte_cnt, connfd);
26          V(&mutex);
27          Rio_writen(connfd, buf, n);
28      }
29  }
```

code/conc/echo-cnt.c

Figure 12.29 `echo_cnt`**: A version of** `echo` **that counts all bytes received from clients.**

the initialization function the first time some thread calls the `echo_cnt` function. The advantage of this approach is that it makes the package easier to use. The disadvantage is that every call to `echo_cnt` makes a call to `pthread_once`, which most times does nothing useful.

Once the package is initialized, the `echo_cnt` function initializes the Rio buffered I/O package (line 20) and then echoes each text line that is received from the client. Notice that the accesses to the shared `byte_cnt` variable in lines 23–25 are protected by P and V operations.

Aside Event-driven programs based on threads

I/O multiplexing is not the only way to write an event-driven program. For example, you might have noticed that the concurrent prethreaded server that we just developed is really an event-driven server with simple state machines for the main and worker threads. The main thread has two states ("waiting for connection request" and "waiting for available buffer slot"), two I/O events ("connection request arrives" and "buffer slot becomes available"), and two transitions ("accept connection request" and "insert buffer item"). Similarly, each worker thread has one state ("waiting for available buffer item"), one I/O event ("buffer item becomes available"), and one transition ("remove buffer item").

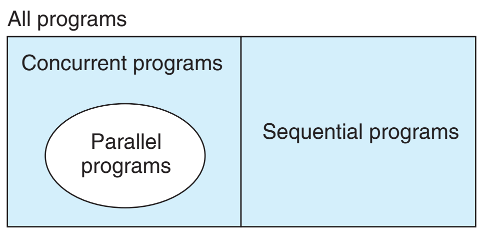


Figure 12.30 **Relationships between the sets of sequential, concurrent, and parallel programs.**

12.6 Using Threads for Parallelism

Thus far in our study of concurrency, we have assumed concurrent threads executing on uniprocessor systems. However, most modern machines have multi-core processors. Concurrent programs often run faster on such machines because the operating system kernel schedules the concurrent threads in parallel on multiple cores, rather than sequentially on a single core. Exploiting such parallelism is critically important in applications such as busy Web servers, database servers, and large scientific codes, and it is becoming increasingly useful in mainstream applications such as Web browsers, spreadsheets, and document processors.

Figure 12.30 shows the set relationships between sequential, concurrent, and parallel programs. The set of all programs can be partitioned into the disjoint sets of sequential and concurrent programs. A sequential program is written as a single logical flow. A concurrent program is written as multiple concurrent flows. A parallel program is a concurrent program running on multiple processors. Thus, the set of parallel programs is a proper subset of the set of concurrent programs.

A detailed treatment of parallel programs is beyond our scope, but studying a few simple example programs will help you understand some important aspects of parallel programming. For example, consider how we might sum the sequence of integers $0, \ldots, n-1$ in parallel. Of course, there is a closed-form solution for this particular problem, but nonetheless it is a concise and easy-to-understand exemplar that will allow us to make some interesting points about parallel programs.

The most straightforward approach for assigning work to different threads is to partition the sequence into t disjoint regions and then assign each of t different threads to work on its own region. For simplicity, assume that n is a multiple of t, such that each region has n/t elements. Let's look at some of the different ways that multiple threads might work on their assigned regions in parallel.

The simplest and most straightforward option is to have the threads sum into a shared global variable that is protected by a mutex. Figure 12.31 shows how we might implement this. In lines 28–33, the main thread creates the peer threads and then waits for them to terminate. Notice that the main thread passes a small integer to each peer thread that serves as a unique thread ID. Each peer thread will use its thread ID to determine which portion of the sequence it should work on. This idea of passing a small unique thread ID to the peer threads is a general technique that is used in many parallel applications. After the peer threads have terminated, the global variable `gsum` contains the final sum. The main thread then uses the closed-form solution to verify the result (lines 36–37).

Figure 12.32 shows the function that each peer thread executes. In line 4, the thread extracts the thread ID from the thread argument and then uses this ID to determine the region of the sequence it should work on (lines 5–6). In lines 9–13, the thread iterates over its portion of the sequence, updating the shared global variable `gsum` on each iteration. Notice that we are careful to protect each update with P and V mutex operations.

When we run `psum-mutex` on a system with four cores on a sequence of size $n = 2^{31}$ and measure its running time (in seconds) as a function of the number of threads, we get a nasty surprise:

	Number of threads				
Version	1	2	4	8	16
`psum-mutex`	68	432	719	552	599

Not only is the program extremely slow when it runs sequentially as a single thread, it is nearly an order of magnitude slower when it runs in parallel as multiple threads. And the performance gets worse as we add more cores. The reason for this poor performance is that the synchronization operations (P and V) are very expensive relative to the cost of a single memory update. This highlights an important lesson about parallel programming: *Synchronization overhead is expensive and should be avoided if possible. If it cannot be avoided, the overhead should be amortized by as much useful computation as possible.*

One way to avoid synchronization in our example program is to have each peer thread compute its partial sum in a private variable that is not shared with any other thread, as shown in Figure 12.33. The main thread (not shown) defines a global array called `psum`, and each peer thread i accumulates its partial sum in `psum[i]`. Since we are careful to give each peer thread a unique memory location to update, it is not necessary to protect these updates with mutexes. The only necessary synchronization is that the main thread must wait for all of the children to finish. After the peer threads have terminated, the main thread sums up the elements of the `psum` vector to arrive at the final result.

code/conc/psum-mutex.c

```c
#include "csapp.h"
#define MAXTHREADS 32

void *sum_mutex(void *vargp); /* Thread routine */

/* Global shared variables */
long gsum = 0;           /* Global sum */
long nelems_per_thread;  /* Number of elements to sum */
sem_t mutex;             /* Mutex to protect global sum */

int main(int argc, char **argv)
{
    long i, nelems, log_nelems, nthreads, myid[MAXTHREADS];
    pthread_t tid[MAXTHREADS];

    /* Get input arguments */
    if (argc != 3) {
        printf("Usage: %s <nthreads> <log_nelems>\n", argv[0]);
        exit(0);
    }
    nthreads = atoi(argv[1]);
    log_nelems = atoi(argv[2]);
    nelems = (1L << log_nelems);
    nelems_per_thread = nelems / nthreads;
    sem_init(&mutex, 0, 1);

    /* Create peer threads and wait for them to finish */
    for (i = 0; i < nthreads; i++) {
        myid[i] = i;
        Pthread_create(&tid[i], NULL, sum_mutex, &myid[i]);
    }
    for (i = 0; i < nthreads; i++)
        Pthread_join(tid[i], NULL);

    /* Check final answer */
    if (gsum != (nelems * (nelems-1))/2)
        printf("Error: result=%ld\n", gsum);

    exit(0);
}
```

code/conc/psum-mutex.c

Figure 12.31 **Main routine for `psum-mutex`.** Uses multiple threads to sum the elements of a sequence into a shared global variable protected by a mutex.

code/conc/psum-mutex.c

```c
/* Thread routine for psum-mutex.c */
void *sum_mutex(void *vargp)
{
    long myid = *((long *)vargp);          /* Extract the thread ID */
    long start = myid * nelems_per_thread; /* Start element index */
    long end = start + nelems_per_thread;  /* End element index */
    long i;

    for (i = start; i < end; i++) {
        P(&mutex);
        gsum += i;
        V(&mutex);
    }
    return NULL;
}
```

code/conc/psum-mutex.c

Figure 12.32 **Thread routine for** `psum-mutex`. Each peer thread sums into a shared global variable protected by a mutex.

code/conc/psum-array.c

```c
/* Thread routine for psum-array.c */
void *sum_array(void *vargp)
{
    long myid = *((long *)vargp);          /* Extract the thread ID */
    long start = myid * nelems_per_thread; /* Start element index */
    long end = start + nelems_per_thread;  /* End element index */
    long i;

    for (i = start; i < end; i++) {
        psum[myid] += i;
    }
    return NULL;
}
```

code/conc/psum-array.c

Figure 12.33 **Thread routine for** `psum-array`. Each peer thread accumulates its partial sum in a private array element that is not shared with any other peer thread.

When we run `psum-array` on our four-core system, we see that it runs orders of magnitude faster than `psum-mutex`:

	Number of threads				
Version	1	2	4	8	16
`psum-mutex`	68.00	432.00	719.00	552.00	599.00
`psum-array`	7.26	3.64	1.91	1.85	1.84

In Chapter 5, we learned how to use local variables to eliminate unnecessary memory references. Figure 12.34 shows how we can apply this principle by having each peer thread accumulate its partial sum into a local variable rather than a global variable. When we run `psum-local` on our four-core machine, we get another order-of-magnitude decrease in running time:

	Number of threads				
Version	1	2	4	8	16
`psum-mutex`	68.00	432.00	719.00	552.00	599.00
`psum-array`	7.26	3.64	1.91	1.85	1.84
`psum-local`	1.06	0.54	0.28	0.29	0.30

code/conc/psum-local.c

```c
/* Thread routine for psum-local.c */
void *sum_local(void *vargp)
{
    long myid = *((long *)vargp);          /* Extract the thread ID */
    long start = myid * nelems_per_thread; /* Start element index */
    long end = start + nelems_per_thread;  /* End element index */
    long i, sum = 0;

    for (i = start; i < end; i++) {
        sum += i;
    }
    psum[myid] = sum;
    return NULL;
}
```

code/conc/psum-local.c

Figure 12.34 Thread routine for `psum-local`. Each peer thread accumulates its partial sum in a local variable.

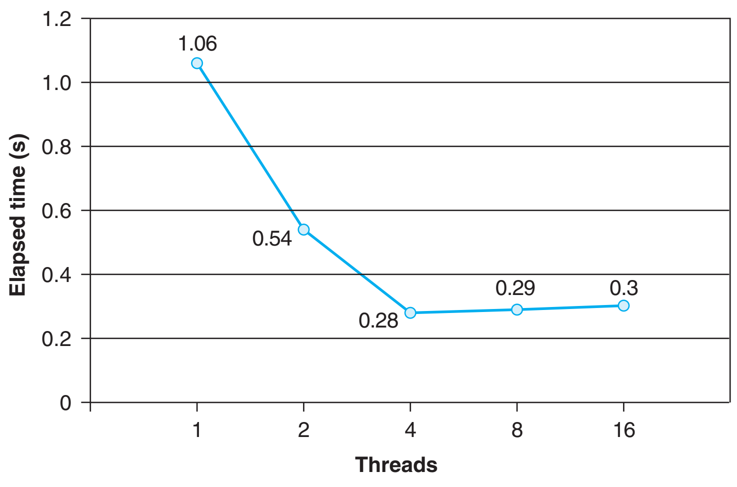


Figure 12.35
Performance of `psum-local` (Figure 12.34). Summing a sequence of 2^{31} elements using four processor cores.

An important lesson to take away from this exercise is that writing parallel programs is tricky. Seemingly small changes to the code have a significant impact on performance.

Characterizing the Performance of Parallel Programs

Figure 12.35 plots the total elapsed running time of the `psum-local` program in Figure 12.34 as a function of the number of threads. In each case, the program runs on a system with four processor cores and sums a sequence of $n = 2^{31}$ elements. We see that running time decreases as we increase the number of threads, up to four threads, at which point it levels off and even starts to increase a little.

In the ideal case, we would expect the running time to decrease linearly with the number of cores. That is, we would expect running time to drop by half each time we double the number of threads. This is indeed the case until we reach the point ($t > 4$) where each of the four cores is busy running at least one thread. Running time actually increases a bit as we increase the number of threads because of the overhead of context switching multiple threads on the same core. For this reason, parallel programs are often written so that each core runs exactly one thread.

Although absolute running time is the ultimate measure of any program's performance, there are some useful relative measures that can provide insight into how well a parallel program is exploiting potential parallelism. The *speedup* of a parallel program is typically defined as

$$S_p = \frac{T_1}{T_p}$$

where p is the number of processor cores and T_k is the running time on k cores. This formulation is sometimes referred to as *strong scaling*. When T_1 is the execution

Threads (t)	1	2	4	8	16
Cores (p)	1	2	4	4	4
Running time (T_p)	1.06	0.54	0.28	0.29	0.30
Speedup (S_p)	1	1.9	3.8	3.7	3.5
Efficiency (E_p)	100%	98%	95%	91%	88%

Figure 12.36 Speedup and parallel efficiency for the execution times in Figure 12.35.

time of a sequential version of the program, then S_p is called the *absolute speedup*. When T_1 is the execution time of the parallel version of the program running on one core, then S_p is called the *relative speedup*. Absolute speedup is a truer measure of the benefits of parallelism than relative speedup. Parallel programs often suffer from synchronization overheads, even when they run on one processor, and these overheads can artificially inflate the relative speedup numbers because they increase the size of the numerator. On the other hand, absolute speedup is more difficult to measure than relative speedup because measuring absolute speedup requires two different versions of the program. For complex parallel codes, creating a separate sequential version might not be feasible, either because the code is too complex or because the source code is not available.

A related measure, known as *efficiency*, is defined as

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}$$

and is typically reported as a percentage in the range (0, 100]. Efficiency is a measure of the overhead due to parallelization. Programs with high efficiency are spending more time doing useful work and less time synchronizing and communicating than programs with low efficiency.

Figure 12.36 shows the different speedup and efficiency measures for our example parallel sum program. Efficiencies over 90 percent such as these are very good, but do not be fooled. We were able to achieve high efficiency because our problem was trivially easy to parallelize. In practice, this is not usually the case. Parallel programming has been an active area of research for decades. With the advent of commodity multi-core machines whose core count is doubling every few years, parallel programming continues to be a deep, difficult, and active area of research.

There is another view of speedup, known as *weak scaling*, which increases the problem size along with the number of processors, such that the amount of work performed on each processor is held constant as the number of processors increases. With this formulation, speedup and efficiency are expressed in terms of the total amount of work accomplished per unit time. For example, if we can double the number of processors and do twice the amount of work per hour, then we are enjoying linear speedup and 100 percent efficiency.

Weak scaling is often a truer measure than strong scaling because it more accurately reflects our desire to use bigger machines to do more work. This is particularly true for scientific codes, where the problem size can be easily increased and where bigger problem sizes translate directly to better predictions of nature. However, there exist applications whose sizes are not so easily increased, and for these applications strong scaling is more appropriate. For example, the amount of work performed by real-time signal-processing applications is often determined by the properties of the physical sensors that are generating the signals. Changing the total amount of work requires using different physical sensors, which might not be feasible or necessary. For these applications, we typically want to use parallelism to accomplish a fixed amount of work as quickly as possible.

Practice Problem 12.11 **(solution page 1074)**

Fill in the blanks for the parallel program in the following table. Assume strong scaling.

Threads (t)	1	4	8
Cores (p)	1	4	8
Running time (T_p)	16	8	4
Speedup (S_p)	1	____	____
Efficiency (E_p)	100%	____	____

12.7 Other Concurrency Issues

You probably noticed that life got much more complicated once we were asked to synchronize accesses to shared data. So far, we have looked at techniques for mutual exclusion and producer-consumer synchronization, but this is only the tip of the iceberg. Synchronization is a fundamentally difficult problem that raises issues that simply do not arise in ordinary sequential programs. This section is a survey (by no means complete) of some of the issues you need to be aware of when you write concurrent programs. To keep things concrete, we will couch our discussion in terms of threads. Keep in mind, however, that these are typical of the issues that arise when concurrent flows of any kind manipulate shared resources.

12.7.1 Thread Safety

When we program with threads, we must be careful to write functions that have a property called thread safety. A function is said to be *thread-safe* if and only if it will always produce correct results when called repeatedly from multiple concurrent threads. If a function is not thread-safe, then we say it is *thread-unsafe*.

We can identify four (nondisjoint) classes of thread-unsafe functions:

Class 1: *Functions that do not protect shared variables.* We have already encountered this problem with the `thread` function in Figure 12.16, which *code/conc/rand.c*

```
unsigned next_seed = 1;

/* rand - return pseudorandom integer in the range 0..32767 */
unsigned rand(void)
{
    next_seed = next_seed*1103515245 + 12543;
    return (unsigned)(next_seed>>16) % 32768;
}

/* srand - set the initial seed for rand() */
void srand(unsigned new_seed)
{
    next_seed = new_seed;
}
```

code/conc/rand.c

Figure 12.37 A thread-unsafe pseudorandom number generator. (Based on [61])

increments an unprotected global counter variable. This class of thread-unsafe functions is relatively easy to make thread-safe: protect the shared variables with synchronization operations such as P and V. An advantage is that it does not require any changes in the calling program. A disadvantage is that the synchronization operations slow down the function.

Class 2: *Functions that keep state across multiple invocations.* A pseudorandom number generator is a simple example of this class of thread-unsafe functions. Consider the pseudorandom number generator package in Figure 12.37.

The `rand` function is thread-unsafe because the result of the current invocation depends on an intermediate result from the previous iteration. When we call `rand` repeatedly from a single thread after seeding it with a call to `srand`, we can expect a repeatable sequence of numbers. However, this assumption no longer holds if multiple threads are calling `rand`.

The only way to make a function such as `rand` thread-safe is to rewrite it so that it does not use any `static` data, relying instead on the caller to pass the state information in arguments. The disadvantage is that the programmer is now forced to change the code in the calling routine as well. In a large program where there are potentially hundreds of different call sites, making such modifications could be nontrivial and prone to error.

Class 3: *Functions that return a pointer to a static variable.* Some functions, such as `ctime` and `gethostbyname`, compute a result in a `static` variable and then return a pointer to that variable. If we call such functions from *code/conc/ctime-ts.c*

```
char *ctime_ts(const time_t *timep, char *privatep)
{
    char *sharedp;

    P(&mutex);
    sharedp = ctime(timep);
    strcpy(privatep, sharedp); /* Copy string from shared to private */
    V(&mutex);
    return privatep;
}
```

code/conc/ctime-ts.c

Figure 12.38 Thread-safe wrapper function for the C standard library `ctime` function. This example uses the lock-and-copy technique to call a class 3 thread-unsafe function.

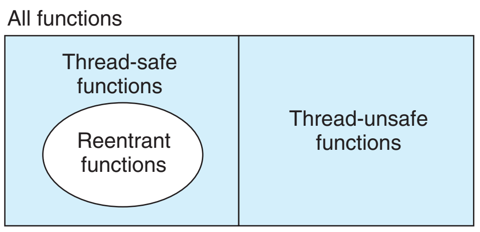
concurrent threads, then disaster is likely, as results being used by one thread are silently overwritten by another thread.

There are two ways to deal with this class of thread-unsafe functions. One option is to rewrite the function so that the caller passes the address of the variable in which to store the results. This eliminates all shared data, but it requires the programmer to have access to the function source code.

If the thread-unsafe function is difficult or impossible to modify (e.g., the code is very complex or there is no source code available), then another option is to use the *lock-and-copy* technique. The basic idea is to associate a mutex with the thread-unsafe function. At each call site, lock the mutex, call the thread-unsafe function, copy the result returned by the function to a private memory location, and then unlock the mutex. To minimize changes to the caller, you should define a thread-safe wrapper function that performs the lock-and-copy and then replace all calls to the thread-unsafe function with calls to the wrapper. For example, Figure 12.38 shows a thread-safe wrapper for `ctime` that uses the lock-and-copy technique.

Class 4: *Functions that call thread-unsafe functions.* If a function f calls a thread-unsafe function g, is f thread-unsafe? It depends. If g is a class 2 function that relies on state across multiple invocations, then f is also thread-unsafe and there is no recourse short of rewriting g. However, if g is a class 1 or class 3 function, then f can still be thread-safe if you protect the call site and any resulting shared data with a mutex. We see a good example of this in Figure 12.38, where we use lock-and-copy to write a thread-safe function that calls a thread-unsafe function.

Figure 12.39
Relationships between the sets of reentrant, thread-safe, and thread-unsafe functions.



code/conc/rand-r.c

```c
/* rand_r - return a pseudorandom integer on 0..32767 */
int rand_r(unsigned int *nextp)
{
    *nextp = *nextp * 1103515245 + 12345;
    return (unsigned int)(*nextp / 65536) % 32768;
}
```

code/conc/rand-r.c

Figure 12.40 `rand_r`**: A reentrant version of the** `rand` **function from Figure 12.37.**

12.7.2 Reentrancy

There is an important class of thread-safe functions, known as *reentrant functions*, that are characterized by the property that they do not reference *any* shared data when they are called by multiple threads. Although the terms *thread-safe* and *reentrant* are sometimes used (incorrectly) as synonyms, there is a clear technical distinction that is worth preserving. Figure 12.39 shows the set relationships between reentrant, thread-safe, and thread-unsafe functions. The set of all functions is partitioned into the disjoint sets of thread-safe and thread-unsafe functions. The set of reentrant functions is a proper subset of the thread-safe functions.

Reentrant functions are typically more efficient than non-reentrant thread-safe functions because they require no synchronization operations. Furthermore, the only way to convert a class 2 thread-unsafe function into a thread-safe one is to rewrite it so that it is reentrant. For example, Figure 12.40 shows a reentrant version of the `rand` function from Figure 12.37. The key idea is that we have replaced the static `next` variable with a pointer that is passed in by the caller.

Is it possible to inspect the code of some function and declare a priori that it is reentrant? Unfortunately, it depends. If all function arguments are passed by value (i.e., no pointers) and all data references are to local automatic stack variables (i.e., no references to static or global variables), then the function is *explicitly reentrant*, in the sense that we can assert its reentrancy regardless of how it is called.

However, if we loosen our assumptions a bit and allow some parameters in our otherwise explicitly reentrant function to be passed by reference (i.e., we allow them to pass pointers), then we have an *implicitly reentrant* function, in the sense that it is only reentrant if the calling threads are careful to pass pointers to nonshared data. For example, the rand_r function in Figure 12.40 is implicitly reentrant.

We always use the term *reentrant* to include both explicit and implicit reentrant functions. However, it is important to realize that reentrancy is sometimes a property of both the caller and the callee, and not just the callee alone.

Practice Problem 12.12 **(solution page 1074)**

The rand_r function in Figure 12.40 is implicitly reentrant. Explain.

12.7.3 Using Existing Library Functions in Threaded Programs

Most Linux functions, including the functions defined in the standard C library (such as malloc, free, realloc, printf, and scanf), are thread-safe, with only a few exceptions. Figure 12.41 lists some common exceptions. (See [110] for a complete list.) The strtok function is a deprecated function (one whose use is discouraged) for parsing strings. The asctime, ctime, and localtime functions are popular functions for converting back and forth between different time and date formats. The gethostbyaddr, gethostbyname, and inet_ntoa functions are obsolete network programming functions that have been replaced by the reentrant getaddrinfo, getnameinfo, and inet_ntop functions, respectively (see Chapter 11). With the exceptions of rand and strtok, they are of the class 3 variety that return a pointer to a static variable. If we need to call one of these functions in a threaded program, the least disruptive approach to the caller is to lock and copy. However, the lock-and-copy approach has a number of disadvantages. First, the additional synchronization slows down the program. Second, functions that return pointers to complex structures of structures require a *deep copy* of the structures in order to copy the entire structure hierarchy. Third, the lock-and-copy approach will not work for a class 2 thread-unsafe function such as rand that relies on static state across calls.

Thread-unsafe function	Thread-unsafe class	Linux thread-safe version
rand	2	rand_r
strtok	2	strtok_r
asctime	3	asctime_r
ctime	3	ctime_r
gethostbyaddr	3	gethostbyaddr_r
gethostbyname	3	gethostbyname_r
inet_ntoa	3	(none)
localtime	3	localtime_r

Figure 12.41 **Common thread-unsafe library functions.**

Therefore, Linux systems provide reentrant versions of most thread-unsafe functions. The names of the reentrant versions always end with the `_r` suffix. For example, the reentrant version of `asctime` is called `asctime_r`. We recommend using these functions whenever possible.

12.7.4 Races

A *race* occurs when the correctness of a program depends on one thread reaching point x in its control flow before another thread reaches point y. Races usually occur because programmers assume that threads will take some particular trajectory through the execution state space, forgetting the golden rule that threaded programs must work correctly for any feasible trajectory.

An example is the easiest way to understand the nature of races. Consider the simple program in Figure 12.42. The main thread creates four peer threads and passes a pointer to a unique integer ID to each one. Each peer thread copies the

code/conc/race.c

```c
/* WARNING: This code is buggy! */
#include "csapp.h"
#define N 4

void *thread(void *vargp);

int main()
{
    pthread_t tid[N];
    int i;

    for (i = 0; i < N; i++)
        Pthread_create(&tid[i], NULL, thread, &i);
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    exit(0);
}

/* Thread routine */
void *thread(void *vargp)
{
    int myid = *((int *)vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

code/conc/race.c

Figure 12.42 **A program with a race.**

ID passed in its argument to a local variable (line 22) and then prints a message containing the ID. It looks simple enough, but when we run this program on our system, we get the following incorrect result:

```
linux> ./race
Hello from thread 1
Hello from thread 3
Hello from thread 2
Hello from thread 3
```

The problem is caused by a race between each peer thread and the main thread. Can you spot the race? Here is what happens. When the main thread creates a peer thread in line 13, it passes a pointer to the local stack variable *i*. At this point, the race is on between the next increment of `i` in line 12 and the dereferencing and assignment of the argument in line 22. If the peer thread executes line 22 before the main thread increments `i` in line 12, then the `myid` variable gets the correct ID. Otherwise, it will contain the ID of some other thread. The scary thing is that whether we get the correct answer depends on how the kernel schedules the execution of the threads. On our system it fails, but on other systems it might work correctly, leaving the programmer blissfully unaware of a serious bug.

To eliminate the race, we can dynamically allocate a separate block for each integer ID and pass the thread routine a pointer to this block, as shown in Figure 12.43 (lines 12–14). Notice that the thread routine must free the block in order to avoid a memory leak.

When we run this program on our system, we now get the correct result:

```
linux> ./norace
Hello from thread 0
Hello from thread 1
Hello from thread 2
Hello from thread 3
```

Practice Problem 12.13 (solution page 1075)

In Figure 12.43, we might be tempted to free the allocated memory block immediately after line 14 in the main thread, instead of freeing it in the peer thread. But this would be a bad idea. Why?

Practice Problem 12.14 (solution page 1075)

A. In Figure 12.43, we eliminated the race by allocating a separate block for each integer ID. Outline a different approach that does not call the `malloc` or `free` functions.

B. What are the advantages and disadvantages of this approach?

code/conc/norace.c

```
#include "csapp.h"
#define N 4

void *thread(void *vargp);

int main()
{
    pthread_t tid[N];
    int i, *ptr;

    for (i = 0; i < N; i++) {
        ptr = Malloc(sizeof(int));
        *ptr = i;
        Pthread_create(&tid[i], NULL, thread, ptr);
    }
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    exit(0);
}

/* Thread routine */
void *thread(void *vargp)
{
    int myid = *((int *)vargp);
    Free(vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

code/conc/norace.c

Figure 12.43 **A correct version of the program in Figure 12.42 without a race.**

12.7.5 Deadlocks

Semaphores introduce the potential for a nasty kind of run-time error, called *deadlock*, where a collection of threads is blocked, waiting for a condition that will never be true. The progress graph is an invaluable tool for understanding deadlock. For example, Figure 12.44 shows the progress graph for a pair of threads that use two semaphores for mutual exclusion. From this graph, we can glean some important insights about deadlock:

- The programmer has incorrectly ordered the P and V operations such that the forbidden regions for the two semaphores overlap. If some execution trajectory happens to reach the *deadlock state* d, then no further progress is

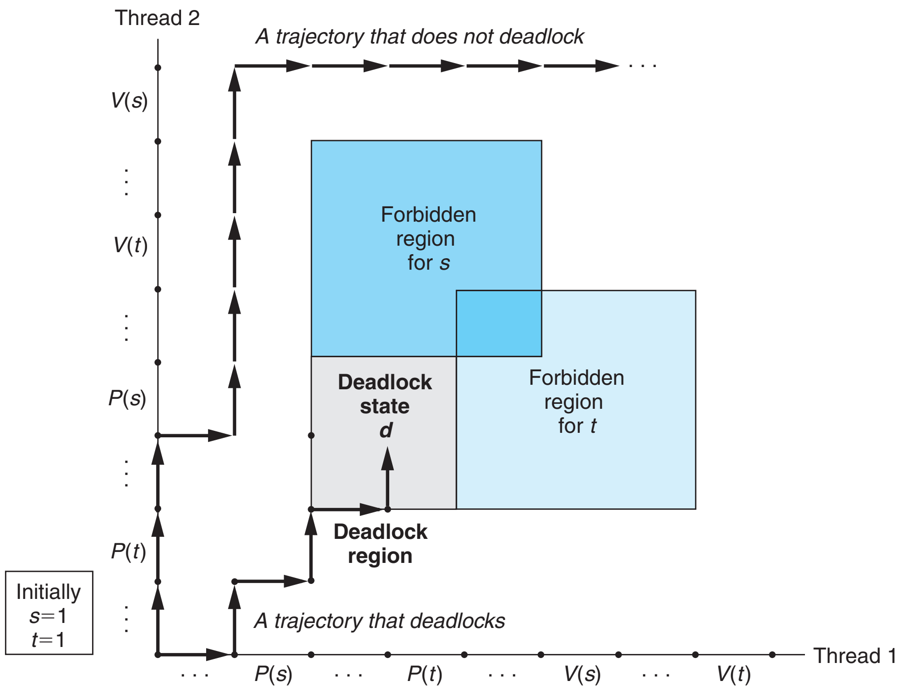


Figure 12.44 **Progress graph for a program that can deadlock.**

possible because the overlapping forbidden regions block progress in every legal direction. In other words, the program is deadlocked because each thread is waiting for the other to do a V operation that will never occur.

- The overlapping forbidden regions induce a set of states called the *deadlock region*. If a trajectory happens to touch a state in the deadlock region, then deadlock is inevitable. Trajectories can enter deadlock regions, but they can never leave.
- Deadlock is an especially difficult issue because it is not always predictable. Some lucky execution trajectories will skirt the deadlock region, while others will be trapped by it. Figure 12.44 shows an example of each. The implications for a programmer are scary. You might run the same program a thousand times without any problem, but then the next time it deadlocks. Or the program might work fine on one machine but deadlock on another. Worst of all, the error is often not repeatable because different executions have different trajectories.

Programs deadlock for many reasons, and preventing them is a difficult problem in general. However, when binary semaphores are used for mutual exclusion, as in Figure 12.44, then you can apply the following simple and effective rule to prevent deadlocks:

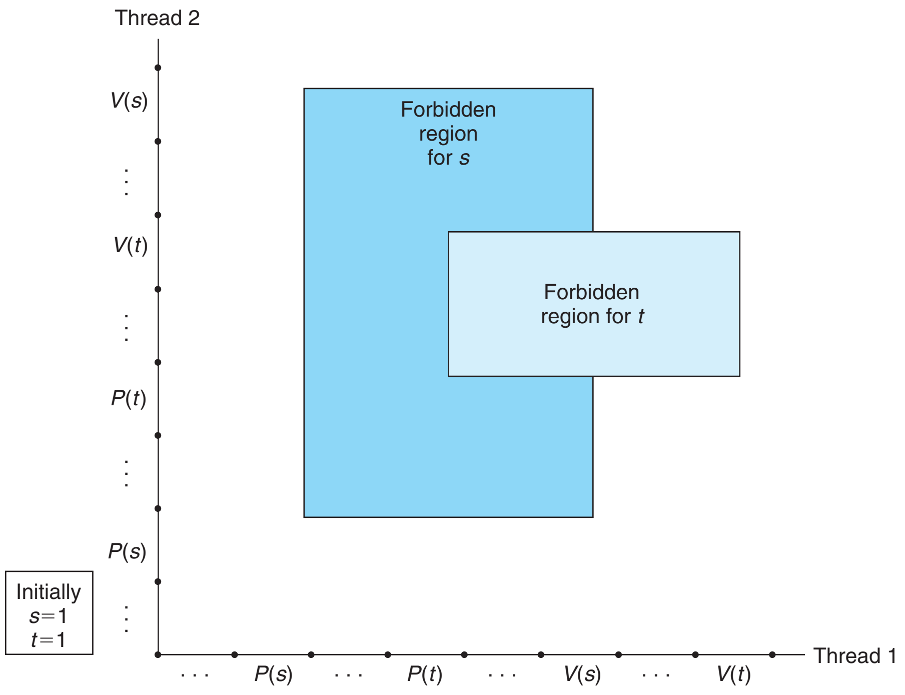


Figure 12.45 **Progress graph for a deadlock-free program.**

Mutex lock ordering rule: Given a total ordering of all mutexes, a program is deadlock-free if each thread acquires its mutexes in order and releases them in reverse order.

For example, we can fix the deadlock in Figure 12.44 by locking s first, then t, in each thread. Figure 12.45 shows the resulting progress graph.

Practice Problem 12.15 (solution page 1075)

Consider the following program, which attempts to use a pair of semaphores for mutual exclusion.

```
Initially: s = 1, t = 0.

Thread 1:        Thread 2:
   P(s);            P(s);
   V(s);            V(s);
   P(t);            P(t);
   V(t);            V(t);
```

A. Draw the progress graph for this program.

B. Does it always deadlock?

C. If so, what simple change to the initial semaphore values will eliminate the potential for deadlock?

D. Draw the progress graph for the resulting deadlock-free program.

12.8 Summary

A concurrent program consists of a collection of logical flows that overlap in time. In this chapter, we have studied three different mechanisms for building concurrent programs: processes, I/O multiplexing, and threads. We used a concurrent network server as the motivating application throughout.

Processes are scheduled automatically by the kernel, and because of their separate virtual address spaces, they require explicit IPC mechanisms in order to share data. Event-driven programs create their own concurrent logical flows, which are modeled as state machines, and use I/O multiplexing to explicitly schedule the flows. Because the program runs in a single process, sharing data between flows is fast and easy. Threads are a hybrid of these approaches. Like flows based on processes, threads are scheduled automatically by the kernel. Like flows based on I/O multiplexing, threads run in the context of a single process, and thus can share data quickly and easily.

Regardless of the concurrency mechanism, synchronizing concurrent accesses to shared data is a difficult problem. The *P* and *V* operations on semaphores have been developed to help deal with this problem. Semaphore operations can be used to provide mutually exclusive access to shared data, as well as to schedule access to resources such as the bounded buffers in producer-consumer systems and shared objects in readers-writers systems. A concurrent prethreaded echo server provides a compelling example of these usage scenarios for semaphores.

Concurrency introduces other difficult issues as well. Functions that are called by threads must have a property known as thread safety. We have identified four classes of thread-unsafe functions, along with suggestions for making them thread-safe. Reentrant functions are the proper subset of thread-safe functions that do not access any shared data. Reentrant functions are often more efficient than non-reentrant functions because they do not require any synchronization primitives. Some other difficult issues that arise in concurrent programs are races and deadlocks. Races occur when programmers make incorrect assumptions about how logical flows are scheduled. Deadlocks occur when a flow is waiting for an event that will never happen.

Bibliographic Notes

Semaphore operations were introduced by Dijkstra [31]. The progress graph concept was introduced by Coffman [23] and later formalized by Carson and Reynolds [16]. The readers-writers problem was introduced by Courtois et al [25]. Operating systems texts describe classical synchronization problems such as the dining philosophers, sleeping barber, and cigarette smokers problems in more detail [102, 106, 113]. The book by Butenhof [15] is a comprehensive description of the Posix threads interface. The paper by Birrell [7] is an excellent introduction to threads programming and its pitfalls. The book by Reinders [90] describes a C/C++ library that simplifies the design and implementation of threaded programs. Several texts cover the fundamentals of parallel programming on multi-core systems [47, 71]. Pugh identifies weaknesses with the way that Java threads interact through memory and proposes replacement memory models [88]. Gustafson proposed the weak-scaling speedup model [43] as an alternative to strong scaling.

Homework Problems

12.16 ◆

Write a version of `hello.c` (Figure 12.13) that creates and reaps n joinable peer threads, where n is a command-line argument.

12.17 ◆

A. The program in Figure 12.46 has a bug. The thread is supposed to sleep for 1 second and then print a string. However, when we run it on our system, nothing prints. Why?

B. You can fix this bug by replacing the `exit` function in line 10 with one of two different Pthreads function calls. Which ones?

code/conc/hellobug.c

```
1   /* WARNING: This code is buggy! */
2   #include "csapp.h"
3   void *thread(void *vargp);
4
5   int main()
6   {
7       pthread_t tid;
8
9       Pthread_create(&tid, NULL, thread, NULL);
10      exit(0);
11  }
12
13  /* Thread routine */
14  void *thread(void *vargp)
15  {
16      Sleep(1);
17      printf("Hello, world!\n");
18      return NULL;
19  }
```

code/conc/hellobug.c

Figure 12.46 **Buggy program for Problem 12.17.**

12.18 ◆

Using the progress graph in Figure 12.21, classify the following trajectories as either safe or unsafe.

A. $H_2, L_2, U_2, H_1, L_1, S_2, U_1, S_1, T_1, T_2$

B. $H_2, H_1, L_1, U_1, S_1, L_2, T_1, U_2, S_2, T_2$

C. $H_1, L_1, H_2, L_2, U_2, S_2, U_1, S_1, T_1, T_2$

12.19 ◆◆

The solution to the first readers-writers problem in Figure 12.26 gives a somewhat weak priority to readers because a writer leaving its critical section might restart a waiting writer instead of a waiting reader. Derive a solution that gives stronger priority to readers, where a writer leaving its critical section will always restart a waiting reader if one exists.

12.20 ◆◆◆

Consider a simpler variant of the readers-writers problem where there are at most N readers. Derive a solution that gives equal priority to readers and writers, in the sense that pending readers and writers have an equal chance of being granted access to the resource. *Hint:* You can solve this problem using a single counting semaphore and a single mutex.

12.21 ◆◆◆◆

Derive a solution to the second readers-writers problem, which favors writers instead of readers.

12.22 ◆◆

Test your understanding of the `select` function by modifying the server in Figure 12.6 so that it echoes at most one text line per iteration of the main server loop.

12.23 ◆◆

The event-driven concurrent echo server in Figure 12.8 is flawed because a malicious client can deny service to other clients by sending a partial text line. Write an improved version of the server that can handle these partial text lines without blocking.

12.24 ◆

The functions in the Rio I/O package (Section 10.5) are thread-safe. Are they reentrant as well?

12.25 ◆

In the prethreaded concurrent echo server in Figure 12.28, each thread calls the `echo_cnt` function (Figure 12.29). Is `echo_cnt` thread-safe? Is it reentrant? Why or why not?

12.26 ◆◆◆

Use the lock-and-copy technique to implement a thread-safe non-reentrant version of `gethostbyname` called `gethostbyname_ts`. A correct solution will use a deep copy of the `hostent` structure protected by a mutex.

12.27 ◆◆

Some network programming texts suggest the following approach for reading and writing sockets: Before interacting with the client, open two standard I/O streams on the same open connected socket descriptor, one for reading and one for writing:

```
FILE *fpin, *fpout;

fpin = fdopen(sockfd, "r");
fpout = fdopen(sockfd, "w");
```

When the server finishes interacting with the client, close both streams as follows:

```
fclose(fpin);
fclose(fpout);
```

However, if you try this approach in a concurrent server based on threads, you will create a deadly race condition. Explain.

12.28 ◆

In Figure 12.45, does swapping the order of the two *V* operations have any effect on whether or not the program deadlocks? Justify your answer by drawing the progress graphs for the four possible cases:

Case 1		Case 2		Case 3		Case 4	
Thread 1	Thread 2	Thread 1	Thread 2	Thread 1	Thread 2	Thread 1	Thread 2
`P(s)`	`P(s)`	`P(s)`	`P(s)`	`P(s)`	`P(s)`	`P(s)`	`P(s)`
`P(t)`	`P(t)`	`P(t)`	`P(t)`	`P(t)`	`P(t)`	`P(t)`	`P(t)`
`V(s)`	`V(s)`	`V(s)`	`V(t)`	`V(t)`	`V(s)`	`V(t)`	`V(t)`
`V(t)`	`V(t)`	`V(t)`	`V(s)`	`V(s)`	`V(t)`	`V(s)`	`V(s)`

12.29 ◆

Can the following program deadlock? Why or why not?

```
Initially: a = 1, b = 1, c = 1.

Thread 1:        Thread 2:
   P(a);            P(c);
   P(b);            P(b);
   V(b);            V(b);
   P(c);            V(c);
   V(c);
   V(a);
```

12.30 ◆

Consider the following program that deadlocks.

```
Initially: a = 1, b = 1, c = 1.

Thread 1:       Thread 2:       Thread 3:
   P(a);           P(c);           P(c);
   P(b);           P(b);           V(c);
   V(b);           V(b);           P(b);
   P(c);           V(c);           P(a);
   V(c);           P(a);           V(a);
   V(a);           V(a);           V(b);
```

A. For each thread, list the pairs of mutexes that it holds simultaneously.

B. If $a < b < c$, which threads violate the mutex lock ordering rule?

C. For these threads, show a new lock ordering that guarantees freedom from deadlock.

12.31 ◆◆◆

Implement a version of the standard I/O `fgets` function, called `tfgets`, that times out and returns `NULL` if it does not receive an input line on standard input within 5 seconds. Your function should be implemented in a package called `tfgets-proc.c` using processes, signals, and nonlocal jumps. It should not use the Linux `alarm` function. Test your solution using the driver program in Figure 12.47.

code/conc/tfgets-main.c

```c
#include "csapp.h"

char *tfgets(char *s, int size, FILE *stream);

int main()
{
    char buf[MAXLINE];

    if (tfgets(buf, MAXLINE, stdin) == NULL)
        printf("BOOM!\n");
    else
        printf("%s", buf);

    exit(0);
}
```

code/conc/tfgets-main.c

Figure 12.47 Driver program for Problems 12.31–12.33.

12.32 ◆◆◆

Implement a version of the `tfgets` function from Problem 12.31 that uses the `select` function. Your function should be implemented in a package called `tfgets-select.c`. Test your solution using the driver program from Problem 12.31. You may assume that standard input is assigned to descriptor 0.

12.33 ◆◆◆

Implement a threaded version of the `tfgets` function from Problem 12.31. Your function should be implemented in a package called `tfgets-thread.c`. Test your solution using the driver program from Problem 12.31.

12.34 ◆◆◆

Write a parallel threaded version of an $N \times M$ matrix multiplication kernel. Compare the performance to the sequential case.

12.35 ◆◆◆

Implement a concurrent version of the Tiny Web server based on processes. Your solution should create a new child process for each new connection request. Test your solution using a real Web browser.

12.36 ◆◆◆

Implement a concurrent version of the Tiny Web server based on I/O multiplexing. Test your solution using a real Web browser.

12.37 ◆◆◆

Implement a concurrent version of the Tiny Web server based on threads. Your solution should create a new thread for each new connection request. Test your solution using a real Web browser.

12.38 ◆◆◆◆

Implement a concurrent prethreaded version of the Tiny Web server. Your solution should dynamically increase or decrease the number of threads in response to the current load. One strategy is to double the number of threads when the buffer becomes full, and halve the number of threads when the buffer becomes empty. Test your solution using a real Web browser.

12.39 ◆◆◆◆

A Web proxy is a program that acts as a middleman between a Web server and browser. Instead of contacting the server directly to get a Web page, the browser contacts the proxy, which forwards the request to the server. When the server replies to the proxy, the proxy sends the reply to the browser. For this lab, you will write a simple Web proxy that filters and logs requests:

A. In the first part of the lab, you will set up the proxy to accept requests, parse the HTTP, forward the requests to the server, and return the results to the browser. Your proxy should log the URLs of all requests in a log file on disk, and it should also block requests to any URL contained in a filter file on disk.

B. In the second part of the lab, you will upgrade your proxy to deal with multiple open connections at once by spawning a separate thread to handle each request. While your proxy is waiting for a remote server to respond to a request so that it can serve one browser, it should be working on a pending request from another browser.

Check your proxy solution using a real Web browser.

Solutions to Practice Problems

Solution to Problem 12.1 (page 1011)

When the parent process on the concurrent server starts executing, the reference counter increments from 0 to 1 for the associated file table. When this parent process forks the child process, the reference counter is incremented from 1 to 2. When the parent closes its copy of the descriptor, the reference count is decremented from 2 to 1. Similarly, when the child's end of connection closes, the reference counter is decremented from 1 to 0.

Solution to Problem 12.2 (page 1011)

When a process terminates for any reason, the kernel closes all open descriptors. Thus, the parent's copy of the connected file descriptor will be closed automatically when the parent exits.

Solution to Problem 12.3 (page 1016)

Recall that the `echo` function from Figure 11.22 echoes each line from the client until the client loses its end of the connection. If Ctrl+D is typed when the `echo` function is under execution, the server would consider it to be the EOF and may assume that the client has closed its end of connection and hence, may stop echoing back to the client.

Solution to Problem 12.4 (page 1020)

`pool.nready` is an integer variable. We reinitialize the `pool.nready` variable with the value obtained from the call to `select` so as to store the total number of ready descriptors returned by `select`.

Solution to Problem 12.5 (page 1028)

Yes, there are chances of memory leak if lines 31 or 32 are deleted from Figure 12.14. Since the threads are not explicitly reaped, each thread must be detached so that its memory resource will be reclaimed when it terminates. Similarly, it is important to free the memory block that was allocated by the main thread.

Solution to Problem 12.6 (page 1031)

The main idea here is that stack variables are private, whereas global and static variables are shared. Static variables such as `cnt` are a little tricky because the sharing is limited to the functions within their scope—in this case, the thread routine.

A. Here is the table:

Variable instance	Referenced by main thread?	peer thread 0?	peer thread 1?
`ptr`	yes	yes	yes
`cnt`	no	yes	yes
`i.m`	yes	no	no
`msgs.m`	yes	yes	yes
`myid.p0`	no	yes	no
`myid.p1`	no	no	yes

Notes:

`ptr` A global variable that is written by the main thread and read by the peer threads.

`cnt` A static variable with only one instance in memory that is read and written by the two peer threads.

`i.m` A local automatic variable stored on the stack of the main thread. Even though its value is passed to the peer threads, the peer threads never reference it on the stack, and thus it is not shared.

`msgs.m` A local automatic variable stored on the main thread's stack and referenced indirectly through `ptr` by both peer threads.

`myid.p0` and `myid.p1` Instances of a local automatic variable residing on the stacks of peer threads 0 and 1, respectively.

B. Variables `ptr`, `cnt`, and `msgs` are referenced by more than one thread and thus are shared.

Solution to Problem 12.7 (page 1034)

The important idea here is that you cannot make any assumptions about the ordering that the kernel chooses when it schedules your threads.

Step	Thread	Instr.	`%rdx`$_1$	`%rdx`$_2$	`cnt`
1	1	H_1	—	—	0
2	1	L_1	0	—	0
3	2	H_2	—	—	0
4	2	L_2	—	0	0
5	2	U_2	—	1	0
6	2	S_2	—	1	1
7	1	U_1	1	—	1
8	1	S_1	1	—	1
9	1	T_1	1	—	1
10	2	T_2	—	1	1

Variable `cnt` has a final incorrect value of 1.

Solution to Problem 12.8 (page 1037)

This problem is a simple test of your understanding of safe and unsafe trajectories in progress graphs. Trajectories such as A and C that skirt the critical region are safe and will produce correct results.

A. $H_1, L_1, U_1, S_1, H_2, L_2, U_2, S_2, T_2, T_1$: safe

B. $H_2, L_2, H_1, L_1, U_1, S_1, T_1, U_2, S_2, T_2$: unsafe

C. $H_1, H_2, L_2, U_2, S_2, L_1, U_1, S_1, T_1, T_2$: safe

Solution to Problem 12.9 (page 1042)

A. $p = 1,\ c = 1,\ n > 1$: Yes, the mutex semaphore is necessary because the producer and consumer can concurrently access the buffer.

B. $p = 1, c = 1, n = 1$: No, the mutex semaphore is not necessary in this case, because a nonempty buffer is equivalent to a full buffer. When the buffer contains an item, the producer is blocked. When the buffer is empty, the consumer is blocked. So at any point in time, only a single thread can access the buffer, and thus mutual exclusion is guaranteed without using the mutex.

C. $p > 1, c > 1, n = 1$: No, the mutex semaphore is not necessary in this case either, by the same argument as the previous case.

Solution to Problem 12.10 (page 1044)

Suppose that a particular semaphore implementation uses a LIFO stack of threads for each semaphore. When a thread blocks on a semaphore in a P operation, its ID is pushed onto the stack. Similarly, the V operation pops the top thread ID from the stack and restarts that thread. Given this stack implementation, an adversarial writer in its critical section could simply wait until another writer blocks on the semaphore before releasing the semaphore. In this scenario, a waiting reader might wait forever as two writers passed control back and forth.

Notice that although it might seem more intuitive to use a FIFO queue rather than a LIFO stack, using such a stack is not incorrect and does not violate the semantics of the P and V operations.

Solution to Problem 12.11 (page 1056)

This problem is a simple sanity check of your understanding of speedup and parallel efficiency:

Threads (t)	1	4	8
Cores (p)	1	4	8
Running time (T_p)	16	8	4
Speedup (S_p)	1	2	4
Efficiency (E_p)	100%	50%	25%

Solution to Problem 12.12 (page 1060)

The `rand_r` function is implicitly reentrant function, because it passes the parameter by reference; i.e., the parameter `*nextp` and not by value. Explicit reentrant functions pass arguments only by value and all data references are to local automatic stack variables.

Solution to Problem 12.13 (page 1062)

If we free the block immediately after the call to `pthread_create` in line 14, then we will introduce a new race, this time between the call to `free` in the main thread and the assignment statement in line 24 of the thread routine.

Solution to Problem 12.14 (page 1062)

A. Another approach is to pass the integer `i` directly, rather than passing a pointer to `i`:

```
for (i = 0; i < N; i++)
    Pthread_create(&tid[i], NULL, thread, (void *)i);
```

   In the thread routine, we cast the argument back to an `int` and assign it to `myid`:

```
int myid = (int) vargp;
```

B. The advantage is that it reduces overhead by eliminating the calls to `malloc` and `free`. A significant disadvantage is that it assumes that pointers are at least as large as `int`s. While this assumption is true for all modern systems, it might not be true for legacy or future systems.

Solution to Problem 12.15 (page 1065)

A. The progress graph for the original program is shown in Figure 12.48 on the next page.

B. The program always deadlocks, since any feasible trajectory is eventually trapped in a deadlock state.

C. To eliminate the deadlock potential, initialize the binary semaphore `t` to 1 instead of 0.

D. The progress graph for the corrected program is shown in Figure 12.49.

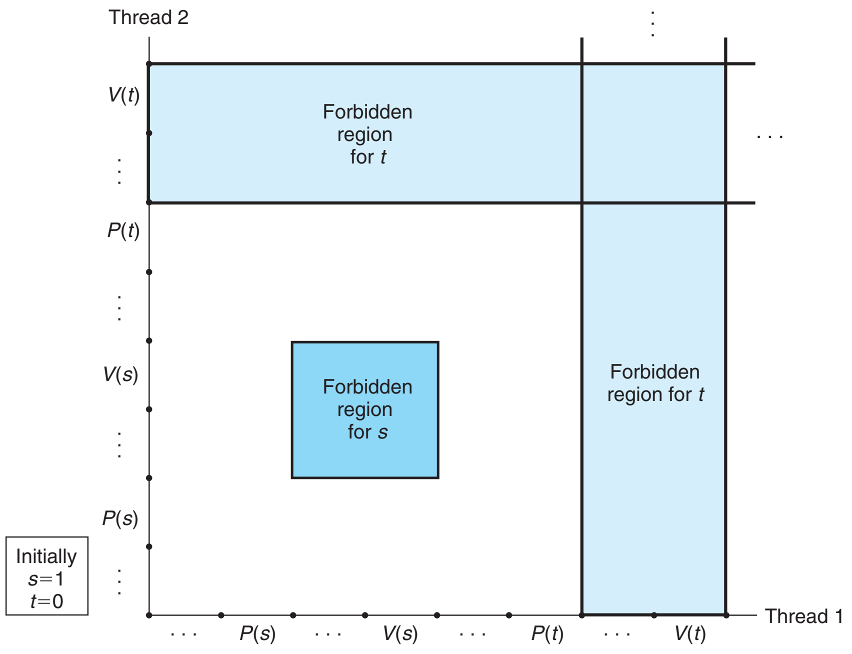


Figure 12.48 **Progress graph for a program that deadlocks.**

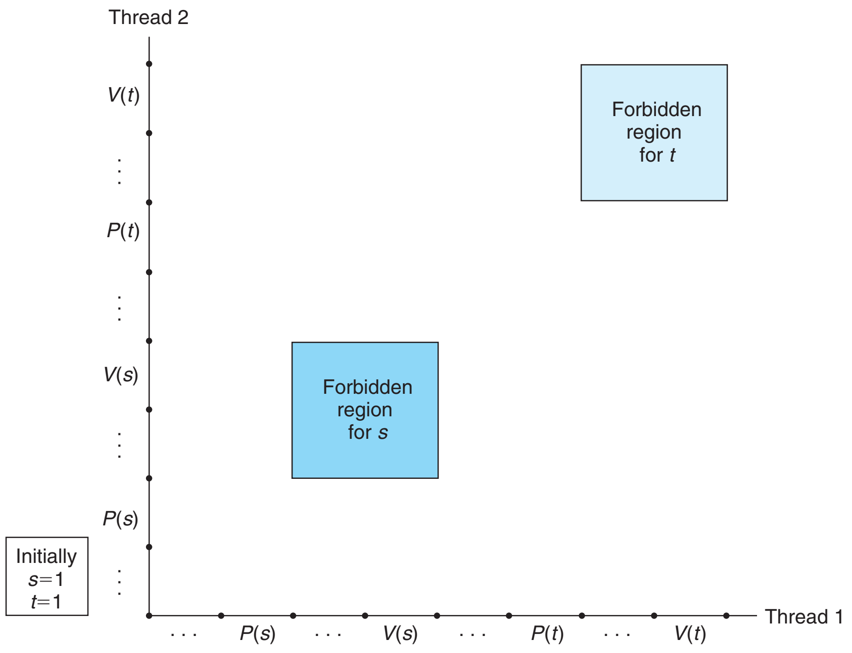


Figure 12.49 **Progress graph for the corrected deadlock-free program.**