20.5 Race Conditions

A *race condition* is usually when multiple processes are accessing data that is shared among them, but the correct outcome depends on the execution order of the processes. Since the execution order of processes on typical Unix systems depends on many factors that may vary between executions, sometimes the outcome is correct, but sometimes the outcome is wrong. The hardest type of race conditions to debug are those in which the outcome is normally correct and only occasionally is the outcome wrong. We will talk more about these types of race conditions in Chapter 26, when we discuss mutual exclusion variables and condition variables. Race conditions are always a concern with threads programming since so much data is shared among all the threads (e.g., all the global variables).

Race conditions of a different type often exist when dealing with signals. The problem occurs because a signal can normally be delivered at anytime while our program is executing. POSIX allows us to *block* a signal from being delivered, but this is often of little use while we are performing I/O operations.

An example is an easy way to see this problem. A race condition exists in Figure 20.5; take a few minutes and see if you can find it. (*Hint:* Where can we be executing when the signal is delivered?) You can also force the condition to occur as follows: Change the argument to `alarm` from 5 to 1, and add `sleep(1)` immediately before the `printf`.

When we make these changes to the function and then type the first line of input, the line is sent as a broadcast and we set the `alarm` for one second in the future. We block in the call to `recvfrom`, and the first reply then arrives for our socket, probably within a few milliseconds. The reply is returned by `recvfrom`, but we then go to sleep for one second. Additional replies are received, and they are placed into our socket's receive buffer. But while we are asleep, the `alarm` timer expires and the `SIGALRM` signal is generated: Our signal handler is called, and it just returns and interrupts the `sleep` in which we are blocked. We then loop around and read the queued replies with a one-second pause each time we print a reply. When we have read all the replies, we block again in the call to `recvfrom`, but the timer is not running. Thus, we will block forever in `recvfrom`. The fundamental problem is that our intent is for our signal handler to interrupt a blocked `recvfrom`, but the signal can be delivered at any time, and we can be executing anywhere in the infinite `for` loop when the signal is delivered.

We now examine four different solutions to this problem: one incorrect solution and three different correct solutions.

Blocking and Unblocking the Signal

Our first (incorrect) solution reduces the window of error by blocking the signal from being delivered while we are executing the remainder of the `for` loop. Figure 20.6 shows the new version.

Figure 20.6 Block signals while executing within the `for` loop (incorrect solution).

bcast/dgclibcast3.c

```
#include     "unp.h"

static void recvfrom_alarm(int);

void
dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
    int     n;
    const int on = 1;
    char    sendline[MAXLINE], recvline[MAXLINE + 1];
    sigset_t sigset_alrm;
    socklen_t len;
    struct sockaddr *preply_addr;

    preply_addr = Malloc(servlen);

    Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));

    Sigemptyset(&sigset_alrm);
    Sigaddset(&sigset_alrm, SIGALRM);

    Signal(SIGALRM, recvfrom_alarm);

    while (Fgets(sendline, MAXLINE, fp) != NULL) {

        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

        alarm(5);
        for ( ; ; ) {
            len = servlen;
            Sigprocmask(SIG_UNBLOCK, &sigset_alrm, NULL);
            n = recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
            Sigprocmask(SIG_BLOCK, &sigset_alrm, NULL);
            if (n < 0) {
```

```
26                 if (errno == EINTR)
27                     break;      /* waited long enough for replies */
28                 else
29                      err_sys("recvfrom error");
30             } else {
31                 recvline[n] = 0;      /* null terminate */
32                 printf("from %s: %s",
33                        Sock_ntop_host(preply_addr, len), recvline);
34             }
35         }
36     }
37     free(preply_addr);
38 }

39 static void
40 recvfrom_alarm(int signo)
41 {
42     return;                  /* just interrupt the recvfrom() */
43 }
```

Declare signal set and initialize

14–15 We declare a signal set, initialize it to the empty set (`sigemptyset`), and then turn on the bit corresponding to `SIGALRM` (`sigaddset`).

Unblock and block signal

21–24 Before calling `recvfrom`, we unblock the signal (so that it can be delivered while we are blocked) and then block it as soon as `recvfrom` returns. If the signal is generated (i.e., the timer expires) while it is blocked, the kernel remembers this fact, but cannot deliver the signal (i.e., call our signal handler) until it is unblocked. This is the fundamental difference between the *generation* of a signal and its *delivery*. Chapter 10 of APUE provides additional details on all these facets of POSIX signal handling.

If we compile and run this program, it appears to work fine, but then most programs with a race condition work most of the time! There is still a problem: The unblocking of the signal, the call to `recvfrom`, and the blocking of the signal are all independent system calls. Assume `recvfrom` returns with the final datagram reply and the signal is delivered between the `recvfrom` and the blocking of the signal. The next call to `recvfrom` will block forever. We have reduced the window, but the problem still exists.

A variation of this solution is to have the signal handler set a global flag when the signal is delivered.

```
static void
recvfrom_alarm(int signo)
{
    had_alarm = 1;
    return;
}
```

The flag is initialized to 0 each time `alarm` is called. Our `dg_cli` function checks this flag before calling `recvfrom` and does not call it if the flag is nonzero.

```
for ( ; ; ) {
    len = servlen;
    Sigprocmask(SIG_UNBLOCK, &sigset_alrm, NULL);
    if (had_alarm == 1)
        break;
    n = recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
```

If the signal was generated during the time it was blocked (after the previous return from `recvfrom`), and when the signal is unblocked in this piece of code, it will be delivered before `sigprocmask` returns, setting our flag. But there is still a small window of time between the testing of the flag and the call to `recvfrom` when the signal can be generated and delivered, and if this happens, the call to `recvfrom` will block forever (assuming, of course, no additional replies are received).

Blocking and Unblocking the Signal with `pselect`

One correct solution is to use `pselect` (Section 6.9), as shown in Figure 20.7.

Figure 20.7 Blocking and unblocking signals with `pselect`.

bcast/dgclibcast4.c

```
 1 #include     "unp.h"
 2 static void recvfrom_alarm(int);

 3 void
 4 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 5 {
 6     int     n;
 7     const int on = 1;
 8     char    sendline[MAXLINE], recvline[MAXLINE + 1];
 9     fd_set  rset;
10     sigset_t sigset_alrm, sigset_empty;
11     socklen_t len;
12     struct sockaddr *preply_addr;

13     preply_addr = Malloc(servlen);

14     Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));

15     FD_ZERO(&rset);

16     Sigemptyset(&sigset_empty);
17     Sigemptyset(&sigset_alrm);
18     Sigaddset(&sigset_alrm, SIGALRM);

19     Signal(SIGALRM, recvfrom_alarm);

20     while (Fgets(sendline, MAXLINE, fp) != NULL) {
21         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

22         Sigprocmask(SIG_BLOCK, &sigset_alrm, NULL);
23         alarm(5);
24         for ( ; ; ) {
25             FD_SET(sockfd, &rset);
26             n = pselect(sockfd + 1, &rset, NULL, NULL, NULL, &sigset_empty);
27             if (n < 0) {
28                 if (errno == EINTR)
29                     break;
30                 else
31                    err_sys("pselect error");
32             } else if (n != 1)
33                 err_sys("pselect error: returned %d", n);

34             len = servlen;
35             n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
36             recvline[n] = 0;     /* null terminate */
37             printf("from %s: %s",
38                    Sock_ntop_host(preply_addr, len), recvline);
39         }
40      }
41      free(preply_addr);
42 }
43 static void
44 recvfrom_alarm(int signo)
45 {
46     return;                     /* just interrupt the recvfrom() */
47 }
```

22–33 We block `SIGALRM` and call `pselect`. The final argument to `pselect` is a pointer to our `sigset_empty` variable, which is a signal set with no signals blocked, that is, all signals are unblocked. `pselect` will save the current signal mask (which has `SIGALRM` blocked), test the specified descriptors, and block if necessary with the signal mask set to the empty set. Before returning, the signal mask of the process is reset to its value when `pselect` was called. The key to `pselect` is that the setting of the signal mask, the testing of the descriptors, and the resetting of the signal mask are atomic operations with regard to the calling process.

34–38 If our socket is readable, we call `recvfrom`, knowing it will not block.

As we mentioned in Section 6.9, `pselect` is new with the POSIX specification; of all the systems in Figure 1.16, only FreeBSD and Linux support the function. Nevertheless, Figure 20.8 shows a simple, albeit incorrect, implementation. Our reason for showing this incorrect implementation is to show the three steps involved: setting the signal mask to the value specified by the caller along with saving the current mask, testing the descriptors, and resetting the signal mask.

Figure 20.8 Simple, incorrect implementation of `pselect`.

lib/pselect.c

```
#include     "unp.h"

int
pselect(int nfds, fd_set *rset, fd_set *wset, fd_set *xset,
        const struct timespec *ts, const sigset_t *sigmask)
{
    int     n;
    struct timeval tv;
16     sigset_t savemask;

17     if (ts != NULL) {
18         tv.tv_sec = ts->tv_sec;
19         tv.tv_usec = ts->tv_nsec / 1000;     /* nanosec -> microsec */
20     }

21     sigprocmask(SIG_SETMASK, sigmask, &savemask);     /* caller's mask */
22     n = select(nfds, rset, wset, xset, (ts == NULL) ? NULL : &tv);
23     sigprocmask(SIG_SETMASK, &savemask, NULL); /* restore mask */

24     return (n);
25 }
```

Using `sigsetjmp` and `siglongjmp`

Another correct way to solve our problem is not to use the ability of a signal handler to interrupt a blocked system call, but to call `siglongjmp` from the signal handler instead. This is called a *nonlocal goto* because we can use it to jump from one function back to another. Figure 20.9 demonstrates this technique.

Figure 20.9 Use of `sigsetjmp` and `siglongjmp` from signal handler.

bcast/dgclibcast5.c

```
 1 #include     "unp.h"
 2 #include     <setjmp.h>

 3 static void recvfrom_alarm(int);
 4 static sigjmp_buf jmpbuf;

 5 void
 6 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 7 {
 8     int     n;
 9     const int on = 1;
10     char    sendline[MAXLINE], recvline[MAXLINE + 1];
11     socklen_t len;
12     struct sockaddr *preply_addr;

13     preply_addr = Malloc(servlen);

14     Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));

15     Signal(SIGALRM, recvfrom_alarm);

16     while (Fgets(sendline, MAXLINE, fp) != NULL) {

17         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

18         alarm(5);
19         for ( ; ; ) {
20             if (sigsetjmp(jmpbuf, 1) != 0)
21                 break;
22             len = servlen;
23             n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
24             recvline[n] = 0;     /* null terminate */
25             printf("from %s: %s",
26                    Sock_ntop_host(preply_addr, len), recvline);
27         }
28      }
29      free(preply_addr);
30 }

31 static void
32 recvfrom_alarm(int signo)
33 {
34     siglongjmp(jmpbuf, 1);
35 }
```

Allocate jump buffer

4 We allocate a jump buffer that will be used by our function and its signal handler.

Call `sigsetjmp`

20–23 When we call `sigsetjmp` directly from our `dg_cli` function, it establishes the jump buffer and returns 0. We proceed on and call `recvfrom`.

Handle `SIGALRM` and call `siglongjmp`

31–35 When the signal is delivered, we call `siglongjmp`. This causes the `sigsetjmp` in the `dg_cli` function to return with a return value equal to the second argument (1), which must be a nonzero value. This will cause the `for` loop in `dg_cli` to terminate.

Using `sigsetjmp` and `siglongjmp` in this fashion guarantees that we will not block forever in `recvfrom` because of a signal delivered at an inopportune time. However, this introduces another potential problem: If the signal is delivered while `printf` is in the middle of its output, we will effectively jump out of the middle of `printf` and back to our `sigsetjmp`. This may leave `printf` with inconsistent private data structures, for example. To prevent this, we should combine the signal blocking and unblocking from Figure 20.6 with the nonlocal goto. This makes this solution unwieldy, as the signal blocking has to occur around any function that may behave poorly as a result of being interrupted in the middle.

Using IPC from Signal Handler to Function

There is yet another correct way to solve our problem. Instead of having the signal handler just return and hopefully interrupt a blocked `recvfrom`, we have the signal handler use IPC to notify our `dg_cli` function that the timer has expired. This is somewhat similar to the proposal we made earlier for the signal handler to set the global `had_alarm` when the timer expired, because that global variable was being used as a form of IPC (shared memory between our function and the signal handler). The problem with that solution, however, was our function had to test this variable, and this led to timing problems if the signal was delivered at about the same time.

What we use in Figure 20.10 is a pipe within our process, with the signal handler writing one byte to the pipe when the timer expires and our `dg_cli` function reading that byte to know when to terminate its `for` loop. What makes this such a nice solution is that the testing for the pipe being readable is done using `select`. We test for either the socket being readable or the pipe being readable.

Create pipe

15 We create a normal Unix pipe and two descriptors are returned. `pipefd[0]` is the read end and `pipefd[1]` is the write end.

> We could also use `socketpair` and get a full-duplex pipe. On some systems, notably SVR4, a normal Unix pipe is always full-duplex and we can read from either end and write to either end.

`select` on both socket and read end of pipe

23–30 We `select` on both `sockfd`, the socket, and `pipefd[0]`, the read end of the pipe.

47–52 When `SIGALRM` is delivered, our signal handler writes one byte to the pipe, making the read end readable. Our signal handler also returns, possibly interrupting `select`. Therefore, if `select` returns `EINTR`, we ignore the error, knowing that the read end of the pipe will also be readable, and that will terminate the `for` loop.

`read` from pipe

39–42 When the read end of the pipe is readable, we `read` the null byte that the signal handler wrote and ignore it. But this tells us that the timer expired, so we `break` out of the infinite `for` loop.

Figure 20.10 Using a pipe as IPC from signal handler to our function.

bcast/dgclibcast6.c

```
 1 #include     "unp.h"

 2 static void recvfrom_alarm(int);
 3 static int pipefd[2];

 4 void
 5 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 6 {
 7     int     n, maxfdp1;
 8     const int on = 1;
 9     char    sendline[MAXLINE], recvline[MAXLINE + 1];
10     fd_set  rset;
11     socklen_t len;
12     struct sockaddr *preply_addr;
```

```
    preply_addr = Malloc(servlen);

    Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));

    Pipe(pipefd);
    maxfdp1 = max(sockfd, pipefd[0]) + 1;

    FD_ZERO(&rset);

    Signal(SIGALRM, recvfrom_alarm);

    while (Fgets(sendline, MAXLINE, fp) != NULL) {
        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

        alarm(5);
        for ( ; ; ) {
            FD_SET(sockfd, &rset);
            FD_SET(pipefd[0], &rset);
            if ( (n = select(maxfdp1, &rset, NULL, NULL, NULL)) < 0) {
                if (errno == EINTR)
                    continue;
                else
                    err_sys("select error");
            }

            if (FD_ISSET(sockfd, &rset)) {
                len = servlen;
                n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr,
                             &len);
                recvline[n] = 0;     /* null terminate */
                printf("from %s: %s",
                       Sock_ntop_host(preply_addr, len), recvline);
            }

            if (FD_ISSET(pipefd[0], &rset)) {
                Read(pipefd[0], &n, 1); /* timer expired */
                break;
            }
         }
     }
     free(preply_addr);
}
static void
recvfrom_alarm(int signo)
{
    Write(pipefd[1], "", 1);     /* write one null byte to pipe */
    return;
}
```

[Team LiB] PREVIOUS NEXT