[Team LiB] PREVIOUS NEXT

20.1 Introduction

In this chapter, we will describe *broadcasting* and in the next chapter, we will describe *multicasting*. All the examples in the text so far have dealt with *unicasting:* a process talking to exactly one other process. Indeed, TCP works with only unicast addresses, although UDP and raw IP support other paradigms. Figure 20.1 shows a comparison of the different types of addressing.

Figure 20.1. Different forms of addressing.

graphics/20fig01.gif

IPv6 has added *anycasting* to the addressing architecture. An IPv4 version of anycasting, which was never widely deployed, is described in RFC 1546 [Partridge, Mendez, and Milliken 1993]. IPv6 anycasting is defined in RFC 3513 [Hinden and Deering 2003]. Anycasting allows addressing one (usually the "closest" by some metric) system out of a set of systems that usually provides identical services. With an appropriate routing configuration, hosts can provide anycasting services in either IPv4 or IPv6 by injecting the same address into the routing protocol in multiple locations. However, RFC 3513's anycasting only permits routers to have anycast addresses; hosts may not provide anycasting services. As of this writing, there is no API defined for using anycast addresses. There is work in progress to refine the IPv6 anycast architecture, and hosts may be able to dynamically provide anycasting services in the future.

The important points in Figure 20.1 are:

- Multicasting support is optional in IPv4, but mandatory in IPv6.
- Broadcasting support is not provided in IPv6. Any IPv4 application that uses broadcasting must be recoded for IPv6 to use multicasting instead.
- Broadcasting and multicasting require datagram transport such as UDP or raw IP; they cannot work with TCP.

One use for broadcasting is to locate a server on the local subnet when the server is assumed to be on the local subnet but its unicast IP address is not known. This is sometimes called *resource discovery*. Another use is to minimize the network traffic on a LAN when there are multiple clients communicating with a single server. There are numerous examples of Internet applications that use broadcasting for this purpose. Some of these can also use multicasting.

- ARP— Although this is a protocol that lies underneath IPv4, and not a user application, ARP broadcasts a request on the local subnet that says, "Will the system with an IP address of a.b.c.d please identify yourself and tell me your hardware address?" ARP uses link-layer broadcast, not IP-layer, but is an example of a use of broadcasting.
- DHCP— The client assumes a server or relay is on the local subnet and sends its request to the broadcast address (often 255.255.255.255 since the client doesn't yet know its IP address, its subnet mask, or the limited broadcast address of the subnet).
- Network Time Protocol (NTP)— In one common scenario, an NTP client is configured with the IP address of one or more servers to use, and the client polls the servers at some frequency (every 64 seconds or longer). The client updates its clock using sophisticated algorithms based on the time-of-day returned by the servers and the RTT to the servers. But on a broadcast LAN, instead of making each of the clients poll a single server, the server can broadcast the current time every 64 seconds for all the clients on the local subnet, reducing the amount of network traffic.
- Routing daemons— The oldest routing daemon, `routed`, which implements RIP version 1, broadcasts its routing table on a LAN. This allows all other routers attached to the LAN to receive these routing announcements, without each router having to be configured with the IP addresses of all its neighboring routers. This feature can also be used by hosts on the LAN listening to these routing announcements and updating their routing tables accordingly. RIP version 2 permits the use of either multicast or broadcast.

We must note that multicasting can replace both uses of broadcasting (resource discovery and reducing network traffic) and we will describe the problems with broadcasting later in this chapter and the next chapter.

[Team LiB] PREVIOUS NEXT

[Team LiB] PREVIOUS NEXT

20.2 Broadcast Addresses

If we denote an IPv4 address as {*subnetid*, *hostid*}, where *subnetid* represents the bits that are covered by the network mask (or the CIDR prefix) and *hostid* represents the bits that are not, then we have two types of broadcast addresses. We denote a field containing all one bits as –1.

1. Subnet-directed broadcast address: {*subnetid*, –1}—This addresses all the interfaces on the specified subnet. For example, if we have the subnet 192.168.42/24, then 192.168.42.255 would be the subnet-directed broadcast address for all interfaces on the 192.168.42/24 subnet.

   Normally, routers do not forward these broadcasts (pp. 226–227 of TCPv2). In Figure 20.2, we show a router connected to the two subnets 192.168.42/24 and 192.168.123/24.

   Figure 20.2. Does a router forward a subnet-directed broadcast?

   graphics/20fig02.gif

   The router receives a unicast IP datagram on the 192.168.123/24 subnet with a destination address of 192.168.42.255 (the subnet-directed broadcast address of another interface). The router normally does not forward the datagram on to the 192.168.42/24 subnet. Some systems have a configuration option that allows subnet-directed broadcasts to be forwarded (Appendix E of TCPv1).

   > Forwarding subnet-directed broadcasts enables a class of denial-of-service attacks called "amplification" attacks; for instance, sending an ICMP echo request to a subnet-directed broadcast address can cause multiple replies to be sent for a single request. Combined with a forged IP source address, this results in a bandwidth utilization attack against the victim system, so it's advisable to leave this configuration option off.
   >
   > For this reason, it's inadvisable to design an application that relies on forwarding of subnet-directed broadcasts except in a controlled environment, where you know it's safe to turn them on.

2. Limited broadcast address: {–1, –1, –1} or 255.255.255.255—Datagrams destined to this address must never be forwarded by a router.

   > 255.255.255.255 is to be used as the destination address during the bootstrap process by applications such as BOOTP and DHCP, which do not yet know the node's IP address.
   >
   > The question is: What does a host do when an application sends a UDP datagram to 255.255.255.255? Most hosts allow this (assuming the process has set the `SO_BROADCAST` socket option) and convert the destination address to the subnet-directed broadcast address of the outgoing interface. It is often necessary to access the datalink directly (Chapter 29) to send a packet to 255.255.255.255.
   >
   > Another question is: What does a multihomed host do when the application sends a UDP datagram to 255.255.255.255? Some systems send a single broadcast on the primary interface (the first interface that was configured) with the destination IP address set to the subnet-directed broadcast address of that interface (p. 736 of TCPv2). Other systems send one copy of the datagram out from each broadcast-capable interface. Section 3.3.6 of RFC 1122 [Braden 1989] "takes no stand" on this issue. For portability, however, if an application needs to send a broadcast out from all broadcast-capable interfaces, it should obtain the interface configuration (Section 17.6) and do one `sendto` for each broadcast-capable interface with the destination set to that interface's broadcast address.

[Team LiB] PREVIOUS NEXT

[Team LiB] PREVIOUS NEXT

20.3 Unicast versus Broadcast

Before looking at broadcasting, let's make certain we understand the steps that take place when a UDP datagram is sent to a unicast address. Figure 20.3 shows three hosts on an Ethernet.

Figure 20.3. Unicast example of a UDP datagram.

graphics/20fig03.gif

The subnet address of the Ethernet is 192.168.42/24 with 24 bits in the network mask, leaving 8 bits for the host ID. The application on the left host calls `sendto` on a UDP socket, sending a datagram to 192.168.42.3, port 7433. The UDP layer prepends a UDP header and passes the UDP datagram to the IP layer. IP prepends an IPv4 header, determines the outgoing interface, and in the case of an Ethernet, ARP is invoked to map the destination IP address to the corresponding Ethernet address: `00:0a:95:79:bc:b4`. The packet is then sent as an Ethernet frame with that 48-bit address as the destination Ethernet address. The frame type field of the Ethernet frame will be `0x0800`, specifying an IPv4 packet. The frame type for an IPv6 packet is `0x86dd`.

The Ethernet interface on the host in the middle sees the frame pass by and compares the destination Ethernet address to its own Ethernet address `00:04:ac:17:bf:38`). Since they are not equal, the interface ignores the frame. With a unicast frame, there is no overhead whatsoever to this host. The interface ignores the frame.

The Ethernet interface on the host on the right also sees the frame pass by, and when it compares the destination Ethernet address with its own Ethernet address, they are equal. This interface reads in the entire frame, probably generates a hardware interrupt when the frame is complete, and the device driver reads the frame from the interface memory. Since the frame type is `0x0800`, the packet is placed on the IP input queue.

When the IP layer processes the packet, it first compares the destination IP address (192.168.42.3) with all of its own IP addresses. (Recall that a host can be multihomed. Also recall our discussion of the strong end system model and the weak end system model in Section 8.8.) Since the destination address is one of the host's own IP addresses, the packet is accepted.

The IP layer then looks at the protocol field in the IPv4 header, and its value will be 17 for UDP. The IP datagram is passed to UDP.

The UDP layer looks at the destination port (and possibly the source port, too, if the UDP socket is connected), and in our example, places the datagram onto the appropriate socket receive queue. The process is awakened, if necessary, to read the newly received datagram.

The key point in this example is that a unicast IP datagram is received by only the one host specified by the destination IP address. No other hosts on the subnet are affected.

We now consider a similar example, on the same subnet, but with the sending application writing a UDP datagram to the subnet-directed broadcast address: 192.168.42.255. Figure 20.4 shows the arrangement.

Figure 20.4. Example of a broadcast UDP datagram.

graphics/20fig04.gif

When the host on the left sends the datagram, it notices that the destination IP address is the subnet-directed broadcast address and maps this into the Ethernet address of 48 one bits: `ff:ff:ff:ff:ff:ff`. This causes *every* Ethernet interface on the subnet to receive the frame. The two hosts on the right of this figure that are running IPv4 will both receive the frame. Since the Ethernet frame type is `0x0800`, both hosts pass the packet to the IP layer. Since the destination IP address matches the broadcast address for each of the two hosts, and since the protocol field is 17 (UDP), both hosts pass the packet up to UDP.

The host on the right passes the UDP datagram to the application that has bound UDP port 520. Nothing special needs to be done by an application to receive a broadcast UDP datagram: It just creates a UDP socket and binds the application's port number to the socket. (We assume the IP address bound is `INADDR_ANY`, which is typical.)

On the host in the middle, no application has bound UDP port 520. The host's UDP code then discards the received datagram. This host must *not* send an ICMP "port unreachable," as doing so could generate a *broadcast storm*: a condition where lots of hosts on the subnet generate a response at about the same time, leading to the network being unusable for a period of time. In addition, it's not clear what the sending host would do with an ICMP error: What if some receivers report errors and others don't?

In this example, we also show the datagram that is output by the host on the left being delivered to itself. This is a property of broadcasts: By definition, a broadcast goes to every host on the subnet, which includes the sender (pp. 109–110 of TCPv2). We also assume that the sending application has bound the port that it is sending to (520), so it will receive a copy of each broadcast datagram it sends. (In general, however, there is no requirement that a process bind a UDP port to which it sends datagrams.)

> In this example, we show a logical loopback performed by either the IP layer or the datalink layer making a copy (pp. 109–110 of TCPv2) and sending the copy up the protocol stack. A network could use a physical loopback, but this can lead to problems in the case of network faults (such as an unterminated Ethernet).

This example shows the fundamental problem with broadcasting: Every IPv4 host on the subnet that is not participating in the application must completely process the broadcast UDP datagram all the way up the protocol stack, through and including the UDP layer, before discarding the datagram. (Recall our discussion following Figure 8.21.) Also, every non-IP host on the subnet (say a host running Novell's IPX) must also receive the entire frame at the datalink layer before discarding the frame (assuming the host does not support the frame type, which would be `0x0800` for an IPv4 datagram). For applications that generate IP datagrams at a high rate (audio or video, for example), this unnecessary processing can severely affect these other hosts on the subnet. We will see in the next chapter how multicasting gets around this problem to some extent.

> Our choice of UDP port 520 in Figure 20.4 is intentional. This is the port used by the `routed` daemon to exchange RIP packets. All routers on a subnet that are using RIP version 1 will send a broadcast UDP datagram every 30 seconds. If there are 200 systems on the subnet, including two routers using RIP, 198 hosts will have to process (and discard) these broadcast datagrams every 30 seconds, assuming none of the 198 hosts is running `routed`. RIP version 2 uses multicast to avoid this very problem.

[Team LiB]   PREVIOUS NEXT

20.4 dg_cli Function Using Broadcasting

We modify our dg_cli function one more time, this time allowing it to broadcast to the standard UDP daytime server (Figure 2.18) and printing all replies. The only change we make to the main function (Figure 8.7) is to change the destination port number to 13.

```
servaddr.sin_port = htons(13);
```

We first compile this modified main function with the unmodified dg_cli function from Figure 8.8 and run it on the host freebsd.

```
     freebsd % udpcli01 192.168.42.255
     hi
     sendto error: Permission denied
```

The command-line argument is the subnet-directed broadcast address for the secondary Ethernet. We type a line of input, the program calls sendto, and the error EACCES is returned. The reason we receive the error is that we are not allowed to send a datagram to a broadcast destination address unless we explicitly tell the kernel that we will be broadcasting. We do this by setting the SO_BROADCAST socket option (Section 7.5).

> Berkeley-derived implementations implement this sanity check. Solaris 2.5, on the other hand, accepts the datagram destined for the broadcast address even if we do not specify the socket option. The POSIX specification requires the SO_BROADCAST socket option to be set to send a broadcast packet.
>
> Broadcasting was a privileged operation with 4.2BSD and the SO_BROADCAST socket option did not exist. This option was added to 4.3BSD and any process was allowed to set the option.

We now modify our dg_cli function as shown in Figure 20.5. This version sets the SO_BROADCAST socket option and prints all the replies received within five seconds.

Allocate room for server's address, set socket option

11–13 malloc allocates room for the server's address to be returned by recvfrom. The SO_BROADCAST socket option is set and a signal handler is installed for SIGALRM.

Read line, send to socket, read all replies

14–24 The next two steps, fgets and sendto, are similar to previous versions of this function. But since we are sending a broadcast datagram, we can receive multiple replies. We call recvfrom in a loop and print all the replies received within five seconds. After five seconds, SIGALRM is generated, our signal handler is called, and recvfrom returns the error EINTR.

Print each received reply

25–29 For each reply received, we call sock_ntop_host, which in the case of IPv4 returns a string containing the dotted-decimal IP address of the server. This is printed along with the server's reply.

If we run the program specifying the subnet-directed broadcast address of 192.168.42.255, we see the following:

```
freebsd % udpcli01 192.168.42.255
hi
from 192.168.42.2: Sat Aug 2 16:42:45 2003
from 192.168.42.1: Sat Aug 2 14:42:45 2003
from 192.168.42.3: Sat Aug 2 14:42:45 2003
hello
from 192.168.42.3: Sat Aug 2 14:42:57 2003
from 192.168.42.2: Sat Aug 2 16:42:57 2003
from 192.168.42.1: Sat Aug 2 14:42:57 2003
```

Each time we must type a line of input to generate the output UDP datagram. Each time we receive three replies, and this includes the sending host. As we said earlier, the destination of a broadcast datagram is *all* the hosts on the attached network, including the sender. Each reply is unicast because the source address of the request, which is used by each server as the destination address of the reply, is a unicast address.

All the systems report the same time because all run NTP.

Figure 20.5 `dg_cli` function that broadcasts.

bcast/dgclibcast1.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];
    socklen_t len;
    struct sockaddr *preply_addr;

    preply_addr = Malloc(servlen);

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

    Signal(SIGALRM, recvfrom_alarm);

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

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

        alarm(5);
        for ( ; ; ) {
            len = servlen;
            n = recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
            if (n < 0) {
                if (errno == EINTR)
                    break;      /* waited long enough for replies */
                else
                   err_sys("recvfrom error");
            } else {
                recvline[n] = 0; /* null terminate */
                printf("from %s: %s",
                       Sock_ntop_host(preply_addr, len), recvline);
            }
        }
    }
    free(preply_addr);
}

static void
recvfrom_alarm(int signo)
{
    return;                     /* just interrupt the recvfrom() */
}
```

IP Fragmentation and Broadcasts

Berkeley-derived kernels do not allow a broadcast datagram to be fragmented. If the size of an IP datagram that is being sent to a broadcast address exceeds the outgoing interface MTU, `EMSGSIZE` is returned (pp. 233–234 of TCPv2). This is a policy decision that has existed since 4.2BSD. There is nothing that prevents a kernel from fragmenting a broadcast datagram, but the feeling is that broadcasting puts enough load on the network as it is, so there is no need to multiply this load by the number of fragments.

We can see this scenario with our program in Figure 20.5. We redirect standard input from a file containing a 2,000-byte line, which will require fragmentation on an Ethernet.

```
freebsd % udpcli01 192.168.42.255 < 2000line
sendto error: Message too long
```

AIX, FreeBSD, and MacOS implement this limitation. Linux, Solaris, and HP-UX fragment datagrams sent to a broadcast address. For portability, however, an application that needs to broadcast should determine the MTU of the outgoing interface using the `SIOCGIFMTU ioctl`, and then subtract the IP and transport header lengths to determine the maximum payload size. Alternately, it can pick a common MTU, like Ethernet's 1500, and use it as a constant.

[Team LiB] PREVIOUS NEXT

[Team LiB]

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

```
 9 #include     "unp.h"

10 int
11 pselect(int nfds, fd_set *rset, fd_set *wset, fd_set *xset,
12         const struct timespec *ts, const sigset_t *sigmask)
13 {
14     int     n;
15     struct timeval tv;
```

```
    sigset_t savemask;

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

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

    return (n);
}
```

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

[Team LiB] PREVIOUS NEXT

20.6 Summary

Broadcasting sends a datagram that all hosts on the attached subnet receive. The disadvantage in broadcasting is that every host on the subnet must process the datagram, up through the UDP layer in the case of a UDP datagram, even if the host is not participating in the application. For high data rate applications, such as audio or video, this can place an excessive processing load on these hosts. We will see in the next chapter that multicasting solves this problem because only the hosts that are interested in the application receive the datagram.

Using a version of our UDP echo client that sends a broadcast to the daytime server and then prints all the replies that are received within five seconds, we looked at race conditions with the `SIGALRM` signal. Since the use of the `alarm` function and the `SIGALRM` signal is a common way to place a timeout on a read operation, this subtle error is common in networking applications. We showed one incorrect way to solve the problem, and three correct ways:

- Using `pselect`
- Using `sigsetjmp` and `siglongjmp`
- Using IPC (typically a pipe) from the signal handler to the main loop

[Team LiB] PREVIOUS NEXT