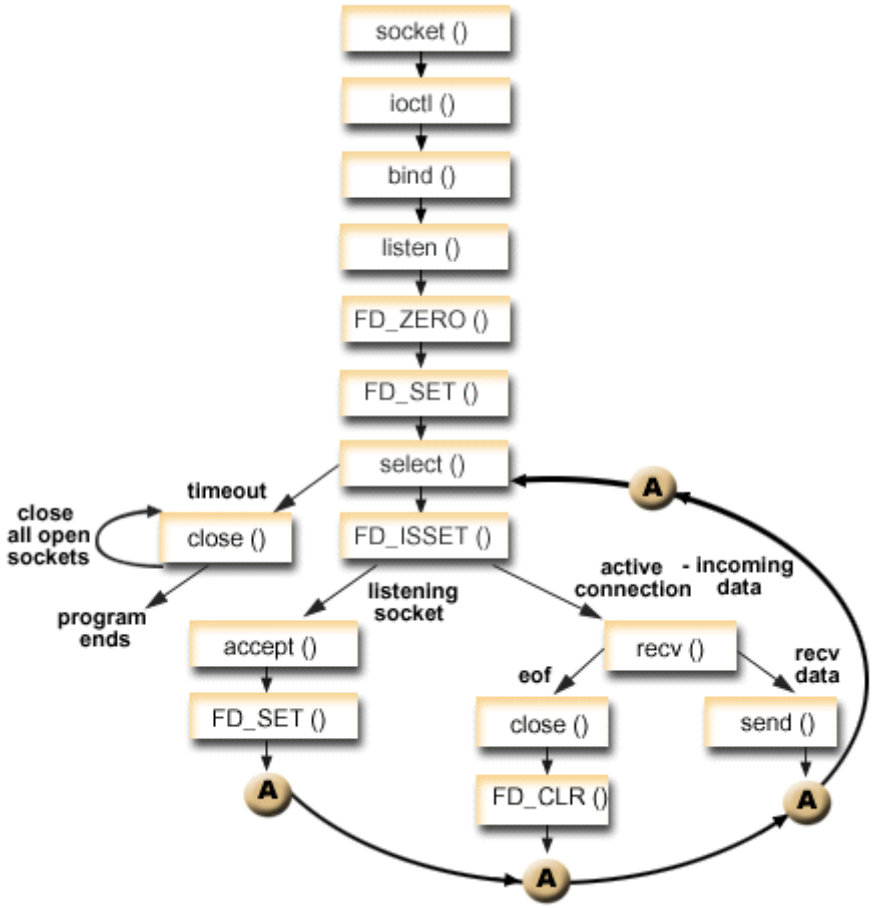
Socket flow of events: Server that uses nonblocking I/O and `select()`

This sample program illustrates a server application that uses nonblocking and the `select()` API.



The following calls are used in the example:

1. The `socket()` API returns a socket descriptor, which represents an endpoint. The statement also identifies that the `INET` (Internet Protocol) address family with the TCP transport (`SOCK_STREAM`) is used for this socket.
2. The `ioctl()` API allows the local address to be reused when the server is restarted before the required wait time expires. In this example, it sets the socket to be

nonblocking. All of the sockets for the incoming connections are also nonblocking because they inherit that state from the listening socket.

3. After the socket descriptor is created, the `bind()` gets a unique name for the socket.
4. The `listen()` allows the server to accept incoming client connections.
5. The server uses the `accept()` API to accept an incoming connection request. The `accept()` API call blocks indefinitely, waiting for the incoming connection to arrive.
6. The `select()` API allows the process to wait for an event to occur and to wake up the process when the event occurs. In this example, the `select()` API returns a number that represents the socket descriptors that are ready to be processed.

   0 Indicates that the process times out. In this example, the timeout is set for 3 minutes.

   -1 Indicates that the process has failed.

   1 Indicates only one descriptor is ready to be processed. In this example, when a 1 is returned, the `FD_ISSET` and the subsequent socket calls complete only once.

   n Indicates that multiple descriptors are waiting to be processed. In this example, when an n is returned, the `FD_ISSET` and subsequent code loops and completes the requests in the order they are received by the server.
7. The accept() and recv() APIs are completed when the EWOULDBLOCK is returned.
8. The send() API echoes the data back to the client.
9. The close() API closes any open socket descriptors.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <errno.h>

#define SERVER_PORT  12345

#define TRUE             1
#define FALSE            0

main (int argc, char *argv[])
{
   int    i, len, rc, on = 1;
   int    listen_sd, max_sd, new_sd;
   int    desc_ready, end_server = FALSE;
   int    close_conn;
   char   buffer[80];
   struct sockaddr_in6 addr;
   struct timeval      timeout;
   fd_set              master_set, working_set;

   /*************************************************************/
```

```
    /* Create an AF_INET6 stream socket to receive incoming      */
    /* connections on                                            */
    /*************************************************************/
    listen_sd = socket(AF_INET6, SOCK_STREAM, 0);
    if (listen_sd < 0)
    {
       perror("socket() failed");
       exit(-1);
    }

    /*************************************************************/
    /* Allow socket descriptor to be reuseable                   */
    /*************************************************************/
    rc = setsockopt(listen_sd, SOL_SOCKET,  SO_REUSEADDR,
                    (char *)&on, sizeof(on));
    if (rc < 0)
    {
       perror("setsockopt() failed");
       close(listen_sd);
       exit(-1);
    }

    /*************************************************************/
    /* Set socket to be nonblocking. All of the sockets for      */
    /* the incoming connections will also be nonblocking since   */
    /* they will inherit that state from the listening socket.   */
    /*************************************************************/
    rc = ioctl(listen_sd, FIONBIO, (char *)&on);
    if (rc < 0)
    {
       perror("ioctl() failed");
       close(listen_sd);
       exit(-1);
    }

    /*************************************************************/
    /* Bind the socket                                           */
    /*************************************************************/
    memset(&addr, 0, sizeof(addr));
    addr.sin6_family      = AF_INET6;
    memcpy(&addr.sin6_addr, &in6addr_any, sizeof(in6addr_any));
    addr.sin6_port        = htons(SERVER_PORT);
    rc = bind(listen_sd,
              (struct sockaddr *)&addr, sizeof(addr));
    if (rc < 0)
    {
       perror("bind() failed");
       close(listen_sd);
       exit(-1);
    }

    /*************************************************************/
    /* Set the listen back log                                   */
    /*************************************************************/
    rc = listen(listen_sd, 32);
    if (rc < 0)
    {
       perror("listen() failed");
       close(listen_sd);
       exit(-1);
    }

    /*************************************************************/
    /* Initialize the master fd_set                              */
    /*************************************************************/
    FD_ZERO(&master_set);
    max_sd = listen_sd;
    FD_SET(listen_sd, &master_set);
```

```
    /*************************************************************/
    /* Initialize the timeval struct to 3 minutes.  If no        */
    /* activity after 3 minutes this program will end.           */
    /*************************************************************/
    timeout.tv_sec  = 3 * 60;
    timeout.tv_usec = 0;

    /*************************************************************/
    /* Loop waiting for incoming connects or for incoming data   */
    /* on any of the connected sockets.                          */
    /*************************************************************/
    do
    {
       /**********************************************************/
       /* Copy the master fd_set over to the working fd_set.     */
       /**********************************************************/
       memcpy(&working_set, &master_set, sizeof(master_set));

       /**********************************************************/
       /* Call select() and wait 3 minutes for it to complete.   */
       /**********************************************************/
       printf("Waiting on select()...\n");
       rc = select(max_sd + 1, &working_set, NULL, NULL, &timeout);

       /**********************************************************/
       /* Check to see if the select call failed.                */
       /**********************************************************/
       if (rc < 0)
       {
          perror("  select() failed");
          break;
       }

       /**********************************************************/
       /* Check to see if the 3 minute time out expired.         */
       /**********************************************************/
       if (rc == 0)
       {
          printf("  select() timed out.  End program.\n");
          break;
       }

       /**********************************************************/
       /* One or more descriptors are readable.  Need to         */
       /* determine which ones they are.                         */
       /**********************************************************/
       desc_ready = rc;
       for (i=0; i <= max_sd  &&  desc_ready > 0; ++i)
       {
          /*******************************************************/
          /* Check to see if this descriptor is ready            */
          /*******************************************************/
          if (FD_ISSET(i, &working_set))
          {
             /****************************************************/
             /* A descriptor was found that was readable - one   */
             /* less has to be looked for.  This is being done   */
             /* so that we can stop looking at the working set   */
             /* once we have found all of the descriptors that   */
             /* were ready.                                      */
             /****************************************************/
             desc_ready -= 1;

             /****************************************************/
             /* Check to see if this is the listening socket     */
             /****************************************************/
             if (i == listen_sd)
             {
                printf("  Listening socket is readable\n");
```

```
            /*********************************************/
            /* Accept all incoming connections that are  */
            /* queued up on the listening socket before we   */
            /* loop back and call select again.          */
            /*********************************************/
            do
            {
               /*********************************************/
               /* Accept each incoming connection.  If      */
               /* accept fails with EWOULDBLOCK, then we    */
               /* have accepted all of them.  Any other     */
               /* failure on accept will cause us to end the */
               /* server.                                   */
               /*********************************************/
               new_sd = accept(listen_sd, NULL, NULL);
               if (new_sd < 0)
               {
                  if (errno != EWOULDBLOCK)
                  {
                     perror("  accept() failed");
                     end_server = TRUE;
                  }
                  break;
               }

               /*********************************************/
               /* Add the new incoming connection to the    */
               /* master read set                           */
               /*********************************************/
               printf("  New incoming connection - %d\n", new_sd);
               FD_SET(new_sd, &master_set);
               if (new_sd > max_sd)
                  max_sd = new_sd;

               /*********************************************/
               /* Loop back up and accept another incoming  */
               /* connection                                */
               /*********************************************/
            } while (new_sd != -1);
         }

         /*********************************************/
         /* This is not the listening socket, therefore an   */
         /* existing connection must be readable      */
         /*********************************************/
         else
         {
            printf("  Descriptor %d is readable\n", i);
            close_conn = FALSE;
            /*********************************************/
            /* Receive all incoming data on this socket  */
            /* before we loop back and call select again.    */
            /*********************************************/
            do
            {
               /*********************************************/
               /* Receive data on this connection until the  */
               /* recv fails with EWOULDBLOCK.  If any other */
               /* failure occurs, we will close the         */
               /* connection.                               */
               /*********************************************/
               rc = recv(i, buffer, sizeof(buffer), 0);
               if (rc < 0)
               {
                  if (errno != EWOULDBLOCK)
                  {
                     perror("  recv() failed");
                     close_conn = TRUE;
                  }
```

```
                   break;
                }

                /*********************************************/
                /* Check to see if the connection has been   */
                /* closed by the client                      */
                /*********************************************/
                if (rc == 0)
                {
                   printf("  Connection closed\n");
                   close_conn = TRUE;
                   break;
                }

                /*********************************************/
                /* Data was received                         */
                /*********************************************/
                len = rc;
                printf("  %d bytes received\n", len);

                /*********************************************/
                /* Echo the data back to the client          */
                /*********************************************/
                rc = send(i, buffer, len, 0);
                if (rc < 0)
                {
                   perror("  send() failed");
                   close_conn = TRUE;
                   break;
                }

             } while (TRUE);

             /***************************************************/
             /* If the close_conn flag was turned on, we need */
             /* to clean up this active connection.  This     */
             /* clean up process includes removing the        */
             /* descriptor from the master set and            */
             /* determining the new maximum descriptor value  */
             /* based on the bits that are still turned on in */
             /* the master set.                               */
             /***************************************************/
             if (close_conn)
             {
                close(i);
                FD_CLR(i, &master_set);
                if (i == max_sd)
                {
                   while (FD_ISSET(max_sd, &master_set) == FALSE)
                      max_sd -= 1;
                }
             }
          } /* End of existing connection is readable */
       } /* End of if (FD_ISSET(i, &working_set)) */
    } /* End of loop through selectable descriptors */

 } while (end_server == FALSE);

 /*************************************************************/
 /* Clean up all of the sockets that are open                 */
 /*************************************************************/
 for (i=0; i <= max_sd; ++i)
 {
    if (FD_ISSET(i, &master_set))
       close(i);
 }
}
```

The `poll()` API is part of the Single Unix Specification and the UNIX 95/98 standard. The `poll()` API performs the same API as the existing `select()` API. The only difference between these two APIs is the interface provided to the caller.

The `select()` API requires that the application pass in an array of bits in which one bit is used to represent each descriptor number. When descriptor numbers are very large, it can overflow the `30KB` allocated memory size, forcing multiple iterations of the process. This overhead can adversely affect performance.

The `poll()` API allows the application to pass an array of structures rather than an array of bits. Because each pollfd structure can contain up to `8 bytes`, the application only needs to pass one structure for each descriptor, even if descriptor numbers are very large.

Socket flow of events: Server that uses `poll()`

The following calls are used in the example:

1. The `socket()` API returns a socket descriptor, which represents an endpoint. The statement also identifies that the `AF_INET6` (Internet Protocol version 6) address family with the TCP transport (`SOCK_STREAM`) is used for this socket.
2. The `setsockopt()` API allows the application to reuse the local address when the server is restarted before the required wait time expires.
3. The `ioctl()` API sets the socket to be nonblocking. All of the sockets for the incoming connections are also nonblocking because they inherit that state from the listening socket.
4. After the socket descriptor is created, the `bind()` API gets a unique name for the socket.
5. The `listen()` API call allows the server to accept incoming client connections.
6. The `poll()` API allows the process to wait for an event to occur and to wake up the process when the event occurs. The `poll()` API might return one of the following values.
   0 Indicates that the process times out. In this example, the timeout is set for 3 minutes (in milliseconds).
   -1 Indicates that the process has failed.

1 Indicates only one descriptor is ready to be processed, which is processed only if it is the listening socket.

1++ Indicates that multiple descriptors are waiting to be processed. The `poll()` API allows simultaneous connection with all descriptors in the queue on the listening socket.

7. The `accept()` and `recv()` APIs are completed when the `EWOULDBLOCK` is returned.
8. The `send()` API echoes the data back to the client.
9. The `close()` API closes any open socket descriptors.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/poll.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <errno.h>

#define SERVER_PORT  12345

#define TRUE             1
#define FALSE            0

main (int argc, char *argv[])
{
  int    len, rc, on = 1;
  int    listen_sd = -1, new_sd = -1;
  int    desc_ready, end_server = FALSE, compress_array = FALSE;
  int    close_conn;
  char   buffer[80];
  struct sockaddr_in6   addr;
  int    timeout;
  struct pollfd fds[200];
  int    nfds = 1, current_size = 0, i, j;

  /*************************************************************/
  /* Create an AF_INET6 stream socket to receive incoming      */
  /* connections on                                            */
  /*************************************************************/
  listen_sd = socket(AF_INET6, SOCK_STREAM, 0);
  if (listen_sd < 0)
  {
    perror("socket() failed");
    exit(-1);
  }

  /*************************************************************/
  /* Allow socket descriptor to be reuseable                   */
  /*************************************************************/
  rc = setsockopt(listen_sd, SOL_SOCKET,  SO_REUSEADDR,
                  (char *)&on, sizeof(on));
  if (rc < 0)
  {
    perror("setsockopt() failed");
    close(listen_sd);
    exit(-1);
  }

  /*************************************************************/
```

```
  /* Set socket to be nonblocking. All of the sockets for      */
  /* the incoming connections will also be nonblocking since   */
  /* they will inherit that state from the listening socket.   */
  /*************************************************************/
  rc = ioctl(listen_sd, FIONBIO, (char *)&on);
  if (rc < 0)
  {
    perror("ioctl() failed");
    close(listen_sd);
    exit(-1);
  }

  /*************************************************************/
  /* Bind the socket                                           */
  /*************************************************************/
  memset(&addr, 0, sizeof(addr));
  addr.sin6_family      = AF_INET6;
  memcpy(&addr.sin6_addr, &in6addr_any, sizeof(in6addr_any));
  addr.sin6_port        = htons(SERVER_PORT);
  rc = bind(listen_sd,
            (struct sockaddr *)&addr, sizeof(addr));
  if (rc < 0)
  {
    perror("bind() failed");
    close(listen_sd);
    exit(-1);
  }

  /*************************************************************/
  /* Set the listen back log                                   */
  /*************************************************************/
  rc = listen(listen_sd, 32);
  if (rc < 0)
  {
    perror("listen() failed");
    close(listen_sd);
    exit(-1);
  }

  /*************************************************************/
  /* Initialize the pollfd structure                           */
  /*************************************************************/
  memset(fds, 0 , sizeof(fds));

  /*************************************************************/
  /* Set up the initial listening socket                        */
  /*************************************************************/
  fds[0].fd = listen_sd;
  fds[0].events = POLLIN;
  /*************************************************************/
  /* Initialize the timeout to 3 minutes. If no                */
  /* activity after 3 minutes this program will end.           */
  /* timeout value is based on milliseconds.                   */
  /*************************************************************/
  timeout = (3 * 60 * 1000);

  /*************************************************************/
  /* Loop waiting for incoming connects or for incoming data   */
  /* on any of the connected sockets.                          */
  /*************************************************************/
  do
  {
    /***********************************************************/
    /* Call poll() and wait 3 minutes for it to complete.      */
    /***********************************************************/
    printf("Waiting on poll()...\n");
    rc = poll(fds, nfds, timeout);

    /***********************************************************/
```

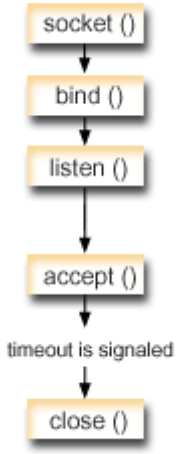
```
    /* Check to see if the poll call failed.                   */
    /***********************************************************/
    if (rc < 0)
    {
      perror("  poll() failed");
      break;
    }

    /***********************************************************/
    /* Check to see if the 3 minute time out expired.          */
    /***********************************************************/
    if (rc == 0)
    {
      printf("  poll() timed out.  End program.\n");
      break;
    }


    /***********************************************************/
    /* One or more descriptors are readable.  Need to          */
    /* determine which ones they are.                          */
    /***********************************************************/
    current_size = nfds;
    for (i = 0; i < current_size; i++)
    {
      /*********************************************************/
      /* Loop through to find the descriptors that returned    */
      /* POLLIN and determine whether it's the listening       */
      /* or the active connection.                             */
      /*********************************************************/
      if(fds[i].revents == 0)
        continue;

      /*********************************************************/
      /* If revents is not POLLIN, it's an unexpected result,  */
      /* log and end the server.                               */
      /*********************************************************/
      if(fds[i].revents != POLLIN)
      {
        printf("  Error! revents = %d\n", fds[i].revents);
        end_server = TRUE;
        break;

      }
      if (fds[i].fd == listen_sd)
      {
        /*******************************************************/
        /* Listening descriptor is readable.                   */
        /*******************************************************/
        printf("  Listening socket is readable\n");

        /*******************************************************/
        /* Accept all incoming connections that are            */
        /* queued up on the listening socket before we         */
        /* loop back and call poll again.                      */
        /*******************************************************/
        do
        {
          /*****************************************************/
          /* Accept each incoming connection. If               */
          /* accept fails with EWOULDBLOCK, then we            */
          /* have accepted all of them. Any other              */
          /* failure on accept will cause us to end the        */
          /* server.                                           */
          /*****************************************************/
          new_sd = accept(listen_sd, NULL, NULL);
          if (new_sd < 0)
          {
            if (errno != EWOULDBLOCK)
```

```
        {
          perror("  accept() failed");
          end_server = TRUE;
        }
        break;
      }

      /*************************************************/
      /* Add the new incoming connection to the        */
      /* pollfd structure                              */
      /*************************************************/
      printf("  New incoming connection - %d\n", new_sd);
      fds[nfds].fd = new_sd;
      fds[nfds].events = POLLIN;
      nfds++;

      /*************************************************/
      /* Loop back up and accept another incoming      */
      /* connection                                    */
      /*************************************************/
    } while (new_sd != -1);
  }

  /*************************************************************/
  /* This is not the listening socket, therefore an            */
  /* existing connection must be readable                      */
  /*************************************************************/

  else
  {
    printf("  Descriptor %d is readable\n", fds[i].fd);
    close_conn = FALSE;
    /*************************************************/
    /* Receive all incoming data on this socket      */
    /* before we loop back and call poll again.      */
    /*************************************************/

    do
    {
      /*************************************************/
      /* Receive data on this connection until the     */
      /* recv fails with EWOULDBLOCK. If any other     */
      /* failure occurs, we will close the             */
      /* connection.                                   */
      /*************************************************/
      rc = recv(fds[i].fd, buffer, sizeof(buffer), 0);
      if (rc < 0)
      {
        if (errno != EWOULDBLOCK)
        {
          perror("  recv() failed");
          close_conn = TRUE;
        }
        break;
      }

      /*************************************************/
      /* Check to see if the connection has been       */
      /* closed by the client                          */
      /*************************************************/
      if (rc == 0)
      {
        printf("  Connection closed\n");
        close_conn = TRUE;
        break;
      }

      /*************************************************/
      /* Data was received                             */
```

```
          /*****************************************************/
          len = rc;
          printf("  %d bytes received\n", len);

          /*****************************************************/
          /* Echo the data back to the client                  */
          /*****************************************************/
          rc = send(fds[i].fd, buffer, len, 0);
          if (rc < 0)
          {
            perror("  send() failed");
            close_conn = TRUE;
            break;
          }

        } while(TRUE);

        /*****************************************************/
        /* If the close_conn flag was turned on, we need       */
        /* to clean up this active connection. This            */
        /* clean up process includes removing the              */
        /* descriptor.                                         */
        /*****************************************************/
        if (close_conn)
        {
          close(fds[i].fd);
          fds[i].fd = -1;
          compress_array = TRUE;
        }


      }  /* End of existing connection is readable             */
    } /* End of loop through pollable descriptors              */

    /***********************************************************/
    /* If the compress_array flag was turned on, we need       */
    /* to squeeze together the array and decrement the number  */
    /* of file descriptors. We do not need to move back the    */
    /* events and revents fields because the events will always*/
    /* be POLLIN in this case, and revents is output.          */
    /***********************************************************/
    if (compress_array)
    {
      compress_array = FALSE;
      for (i = 0; i < nfds; i++)
      {
        if (fds[i].fd == -1)
        {
          for(j = i; j < nfds; j++)
          {
            fds[j].fd = fds[j+1].fd;
          }
          i--;
          nfds--;
        }
      }
    }

  } while (end_server == FALSE); /* End of serving running.    */

  /*************************************************************/
  /* Clean up all of the sockets that are open                 */
  /*************************************************************/
  for (i = 0; i < nfds; i++)
  {
    if(fds[i].fd >= 0)
      close(fds[i].fd);
  }
}
```

When a process or an application becomes blocked, signals allow you to be notified. They also provide a time limit for blocking processes.

In this example, the signal occurs after five seconds on the `accept()` call. This call normally blocks indefinitely, but because there is an alarm set, the call blocks only for five seconds. Because blocked programs can hinder performance of an application or a server, you can use signals to diminish this impact. The following example shows how to use signals with blocking socket APIs.



Socket flow of events: Using signals with blocking socket

The following sequence of API calls shows how you can use signals to alert the application when the socket has been inactive:

1. The `socket()` API returns a socket descriptor, which represents an endpoint. The statement also identifies that the `AF_INET6` (Internet Protocol version 6) address family with the TCP transport (`SOCK_STREAM`) is used for this socket.
2. After the socket descriptor is created, a `bind()` API gets a unique name for the socket. In this example, a port number is not specified because the client application does not connect to this socket. This code snippet can be used within other server programs that use blocking APIs, such as `accept()`.
3. The `listen()` API indicates a willingness to accept client connection requests. After the `listen()` API is issued, an alarm is set to go off in five seconds. This alarm or signal alerts you when the `accept()` call blocks.

4. The `accept()` API accepts a client connection request. This call normally blocks indefinitely, but because there is an alarm set, the call only blocks for five seconds. When the alarm goes off, the accept call is completed with -1 and with an errno value of `EINTR`.
5. The `close()` API ends any open socket descriptors.

```
/*********************************************************************/
/* Example shows how to set alarms for blocking socket APIs          */
/*********************************************************************/

/*********************************************************************/
/* Include files                                                     */
/*********************************************************************/
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>

/*********************************************************************/
/* Signal catcher routine.  This routine will be called when the     */
/* signal occurs.                                                    */
/*********************************************************************/
void catcher(int sig)
{
   printf("   Signal catcher called for signal %d\n", sig);
}

/*********************************************************************/
/* Main program                                                      */
/*********************************************************************/
int main(int argc, char *argv[])
{
   struct sigaction sact;
   struct sockaddr_in6 addr;
   time_t t;
   int sd, rc;

/*********************************************************************/
/* Create an AF_INET6, SOCK_STREAM socket                            */
/*********************************************************************/
   printf("Create a TCP socket\n");
   sd = socket(AF_INET6, SOCK_STREAM, 0);
   if (sd == -1)
   {
      perror("   socket failed");
      return(-1);
   }

/*********************************************************************/
/* Bind the socket.  A port number was not specified because         */
/* we are not going to ever connect to this socket.                  */
/*********************************************************************/
   memset(&addr, 0, sizeof(addr));
   addr.sin6_family = AF_INET6;
   printf("Bind the socket\n");
   rc = bind(sd, (struct sockaddr *)&addr, sizeof(addr));
   if (rc != 0)
   {
      perror("   bind failed");
```

```
      close(sd);
      return(-2);
   }

/*****************************************************************/
/* Perform a listen on the socket.                               */
/*****************************************************************/
   printf("Set the listen backlog\n");
   rc = listen(sd, 5);
   if (rc != 0)
   {
      perror("   listen failed");
      close(sd);
      return(-3);
   }

/*****************************************************************/
/* Set up an alarm that will go off in 5 seconds.                */
/*****************************************************************/
   printf("\nSet an alarm to go off in 5 seconds. This alarm will cause the\n");
   printf("blocked accept() to return a -1 and an errno value of EINTR.\n\n");
   sigemptyset(&sact.sa_mask);
   sact.sa_flags = 0;
   sact.sa_handler = catcher;
   sigaction(SIGALRM, &sact, NULL);
   alarm(5);

/*****************************************************************/
/* Display the current time when the alarm was set               */
/*****************************************************************/
   time(&t);
   printf("Before accept(), time is %s", ctime(&t));

/*****************************************************************/
/* Call accept.  This call will normally block indefinitely,     */
/* but because we have an alarm set, it will only block for      */
/* 5 seconds.  When the alarm goes off, the accept call will     */
/* complete with -1 and an errno value of EINTR.                 */
/*****************************************************************/
   errno = 0;
   printf("   Wait for an incoming connection to arrive\n");
   rc = accept(sd, NULL, NULL);
   printf("   accept() completed.  rc = %d, errno = %d\n", rc, errno);
   if (rc >= 0)
   {
      printf("   Incoming connection was received\n");
      close(rc);
   }
   else
   {
      perror("   errno string");
   }

/*****************************************************************/
/* Show what time it was when the alarm went off                 */
/*****************************************************************/
   time(&t);
   printf("After accept(), time is %s\n", ctime(&t));
   close(sd);
   return(0);
}
```