0.1 Sockets

- A socket is a bidirectional communication device that can be used to communicate with another process on the same machine or with a process running on other machines.
- Internet programs such as Telnet, rlogin, FTP, talk, and the World Wide Web use sockets.

0.1.1 Socket Concepts

- Specify three parameters:
  - communication style;
    - When data is sent through a socket, it is packaged into chunks called **packets**. The communication style determines how these packets are handled and how they are addressed from the sender to the receiver.
    - *Connection* styles guarantee delivery of all packets in the order they were sent. If packets are lost or reordered by problems in the network, the receiver automatically requests their retransmission from the sender.
    - *Datagram* styles do not guarantee delivery or arrival order. Packets may be lost or reordered in transit due to network errors or other conditions.
  - namespace;
    - A socket namespace specifies how socket addresses are written. A socket address identifies one end of a socket connection. For example, socket addresses
      - In the *local namespace* are ordinary filenames.
      - In *Internet namespace*, a socket address is composed of the Internet address of a host attached to the network and a port number. The port number distinguishes among multiple sockets on the same host.
  - protocol;
    - A protocol specifies how data is transmitted.
    - Some protocols are TCP/IP, the primary networking protocols used by the Internet; the AppleTalk network protocol; and the UNIX local communication protocol.

0.1.2 System Calls

- These are the system calls involving sockets:
  - **socket**, Creates a socket
  - **close**, Destroys a socket
  - **connect**, Creates a connection between two sockets
  - **bind**, Labels a server socket with an address
  - **listen**, Configures a socket to accept conditions
  - **accept**, Accepts a connection and creates a new socket for the connection

  Sockets are represented by file descriptors.

- **Creating and Destroying Sockets**
  - The **socket** and **close** functions create and destroy sockets, respectively.
  - When you create a socket, specify the three socket choices: namespace, communication style, and protocol.
    - **PF_LOCAL** or **PF_UNIX** specifies the local namespace, and **PF_INET** specifies Internet namespaces.
    - **SOCK_STREAM** for a connection-style socket, or use **SOCK_DGRAM** for a datagram-style socket.
    - Each protocol is valid for a particular namespace-style combination. Because there is usually one best protocol for each such pair, specifying 0 is usually the correct protocol.
  - If **socket** succeeds, it returns a file descriptor for the socket. You can *read from* or *write to* the socket using read, write, and so on, as with other file descriptors.
  - When you are finished with a socket, call **close** to remove it.

0.1.3 Servers

- A server's life cycle consists of
  - the creation of a connection-style (tcp) socket,
  - binding an address to its socket,
  - placing a call to **listen** that enables connections to the socket,
  - placing calls to **accept** incoming connections,
  - and then closing the socket.
- Data isn't read and written directly via the server socket; instead, each time a program accepts a new connection, Linux creates a separate socket to use in transferring data over that connection.
  - An address must be bound to the server's socket using **bind** if a client is to find it.
  - When an address is bound to a connection-style socket, it must invoke **listen** to indicate that it is a server.
  - A server accepts a connection request from a client by invoking **accept**.
    - The call to accept creates a new socket for communicating with the client and returns the corresponding file descriptor.
    - The original server socket continues to accept new client connections.

0.1.4 Local Sockets

- Sockets connecting processes on the same computer can use the local namespace represented by the synonyms **PF_LOCAL** and **PF_UNIX**.
- These are called local sockets or UNIX-domain sockets. Their socket addresses, specified by filenames, are used only when creating connections.
- The only permissible protocol for the local namespace is 0.

0.1.5 An Example Using Local Namespace Sockets

- The server program, (see Fig. 1), creates a local namespace socket and listens for connections on it.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
/* Read text from the socket and print it out.  Continue until the
   socket closes.  Return non-zero if the client sent a "quit"
   message, zero otherwise.  */
int server (int client_socket)
{
  while (1) {
    int length;
    char* text;
    /* First, read the length of the text message from the socket.  If
       read returns zero, the client closed the connection.  */
    if (read (client_socket, &length, sizeof (length)) == 0)
      return 0;
    /* Allocate a buffer to hold the text.  */
    text = (char*) malloc (length);
    /* Read the text itself, and print it.  */
    read (client_socket, text, length);
    printf ("%s\n", text);
    /* Free the buffer.  */
    free (text);
    /* If the client sent the message "quit", we're all done.  */
    if (!strcmp (text, "quit"))
      return 1;
  }
}
int main (int argc, char* const argv[])
{
  const char* const socket_name = argv[1];
  int socket_fd;
  struct sockaddr_un name;
  int client_sent_quit_message;
  /* Create the socket.  */
  socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);
  /* Indicate this is a server.  */
  name.sun_family = AF_LOCAL;
  strcpy (name.sun_path, socket_name);
  bind (socket_fd, &name, SUN_LEN (&name));
  /* Listen for connections.  */
  listen (socket_fd, 5);
  /* Repeatedly accept connections, spinning off one server() to deal
     with each client.  Continue until a client sends a "quit" message.  */
  do {
    struct sockaddr_un client_name;
    socklen_t client_name_len;
    int client_socket_fd;
    /* Accept a connection.  */
    client_socket_fd = accept (socket_fd, &client_name, &client_name_len);
    /* Handle the connection.  */
    client_sent_quit_message = server (client_socket_fd);
    /* Close our end of the connection.  */
    close (client_socket_fd);
  }
  while (!client_sent_quit_message);
  /* Remove the socket file.  */
  close (socket_fd);
  unlink (socket_name);
  return 0;
}
```

Figure 1: Local Namespace Socket Server.

- The socket-server program takes the path to the socket as its command-line argument.
- The socket-client program, (see Fig. 2), connects to a local namespace socket and sends a message. The name path to the socket and the message are specified on the command line.

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
/* Write TEXT to the socket given by file descriptor SOCKET_FD.  */
void write_text (int socket_fd, const char* text)
{
  /* Write the number of bytes in the string, including
     NUL-termination.  */
  int length = strlen (text) + 1;
  write (socket_fd, &length, sizeof (length));
  /* Write the string.  */
  write (socket_fd, text, length);
}
int main (int argc, char* const argv[])
{
  const char* const socket_name = argv[1];
  const char* const message = argv[2];
  int socket_fd;
  struct sockaddr_un name;
  /* Create the socket.  */
  socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);
  /* Store the server's name in the socket address.  */
  name.sun_family = AF_LOCAL;
  strcpy (name.sun_path, socket_name);
  /* Connect the socket.  */
  connect (socket_fd, &name, SUN_LEN (&name));
  /* Write the text on the command line to the socket.  */
  write_text (socket_fd, message);
  close (socket_fd);
  return 0;
}
```

Figure 2: (socket-client.c) Local Namespace Socket Client.

```
$ ./socket-server /yourdirectory/socket
```

In another window, run the client a few times, specifying the same socket path plus messages to send to the client:

```
$ ./socket-client /yourdirectory/socket "Hello, world."
$ ./socket-client /yourdirectory/socket "This is a test."
$ ./socket-client /yourdirectory/socket "quit"
```

0.1.6 Internet-Domain Sockets

- **UNIX-domain** sockets can be used only for communication between two processes on the same computer.
- **Internet-domain** sockets, on the other hand, may be used to connect processes on different machines connected by a network.
- Sockets connecting processes through the Internet use the Internet namespace represented by **PF_INET**. The most common protocols are TCP/IP.
  - Internet socket addresses contain two parts: a machine and a port number.This information is stored in a struct **sockaddr_in** variable.
  - Fig. 3 illustrates the use of Internet-domain sockets. The program obtains the home page from the Web server whose hostname is specified on the command line.

```
$ ./socket-inet siber.cankaya.edu.tr
```

1 Devices

- LINUX, like most operating systems, interacts with hardware devices via modularized software components called **device drivers**.
- A device driver hides the peculiarities of a hardware device's communication protocols from the operating system and allows the system to interact with the device through a standardized interface.
- Under Linux, device drivers are part of the kernel and may be either linked statically into the kernel or loaded on demand as kernel modules.
- Device drivers run as part of the kernel and aren't directly accessible to user processes. However, Linux provides a mechanism by which processes can communicate with a device driver -and through it with a hardware device- via file-like objects.

```
#include <stdlib.h>
#include <stdio.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/socket.h>
#include <unistd.h>
#include <string.h>
/* Print the contents of the home page for the server's socket.
   Return an indication of success.  */
void get_home_page (int socket_fd)
{
  char buffer[10000];
  ssize_t number_characters_read;
  /* Send the HTTP GET command for the home page.  */
  sprintf (buffer, "GET /\n");
  write (socket_fd, buffer, strlen (buffer));
  /* Read from the socket.  read may not return all the data at one
     time, so keep trying until we run out.  */
  while (1) {
    number_characters_read = read (socket_fd, buffer, 10000);
    if (number_characters_read == 0)
      return;
    /* Write the data to standard output.  */
    fwrite (buffer, sizeof (char), number_characters_read, stdout);
  }
}
int main (int argc, char* const argv[])
{
  int socket_fd;
  struct sockaddr_in name;
  struct hostent* hostinfo;
  /* Create the socket.  */
  socket_fd = socket (PF_INET, SOCK_STREAM, 0);
  /* Store the server's name in the socket address.  */
  name.sin_family = AF_INET;
  /* Convert from strings to numbers.  */
  hostinfo = gethostbyname (argv[1]);
  if (hostinfo == NULL)
    return 1;
  else
    name.sin_addr = *((struct in_addr *) hostinfo->h_addr);
  /* Web servers use port 80.  */
  name.sin_port = htons (80);
  /* Connect to the web server  */
  if (connect (socket_fd, &name, sizeof (struct sockaddr_in)) == -1) {
    perror ("connect");
    return 1;
  }
  /* Retrieve the server's home page.  */
  get_home_page (socket_fd);
  return 0;
}
```

Figure 3: Read from a WWW Server.

- These objects appear in the file system, and programs can open them, read from them, and write to them practically as if they were normal files.

1.1 Device Types

- Device files aren't ordinary files -they do not represent regions of data on a disk based file system.

- Data read from or written to a device file is communicated to the corresponding device driver, and from there to the underlying device. Device files come in two flavors:

  - A character device represents a hardware device that reads or writes a serial stream of data bytes. Serial and parallel ports, tape drives, terminal devices, and sound cards are examples of character devices.
  - A block device represents a hardware device that reads or writes data in fixed-size blocks. Unlike a character device, a block device provides random access to data stored on the device. A disk drive is an example of a block device.

1.2 Device Numbers

- Linux identifies devices using two numbers: the **major device number** and the **minor device number**.
- The major device number specifies which driver the device corresponds to. The same major device number may correspond to two different drivers, one a character device and one a block device.
- Minor device numbers distinguish individual devices or components controlled by a single driver.

1.3 Device Entries

- A device entry is in many ways the same as a regular file.
- If you try to copy a device entry using **cp**, though, you'll read bytes from the device (if the device supports reading) and write them to the destination file.
- If you try to overwrite a device entry, you'll write bytes to the corresponding device instead.
- You can create a device entry in the file system using the **mknod** command.
- Creating a device entry in the file system doesn't automatically imply that the corresponding device driver or hardware device is present or available; the device entry is merely a portal for communicating with the driver, if it's there.

```
$ mknod ./lp0 c 6 0
$ ls -l lp0
$ rm ./lp0
```

1.3.1 The /dev Directory

- By convention, a GNU/Linux system includes a directory **/dev** containing the full complement of character and block device entries for devices that Linux knows about.
- Entries in **/dev** have standardized names corresponding to major and minor device numbers.

```
$ ls -l /dev/hda /dev/hda1
$ ls -l /dev/lp0
```

- In most cases, you should not use **mknod** to create your own device entries. Use the entries in **/dev** instead.

1.3.2 Accessing Devices by Opening Files

- How do you use these devices? In the case of character devices, it can be quite simple: Open the device as if it were a normal file, and read from or write to it.

```
$ cat document.txt > /dev/lp0
In a program, sending data to a device is
int fd = open ("/dev/lp0", O_WRONLY);
write (fd, buffer, buffer_length);
close (fd);
```

1.4 Hardware Devices

- Some common block (character) devices are listed in Fig. 4 (5).
- A terminal program might access a modem directly through a serial port device. Data *written to* or *read from* the devices is transmitted via the modem to a remote computer.
- A program can write directly to the first virtual terminal writing data to **/dev/tty1**. Terminal windows running in a graphical environment, or remote login terminal sessions, are not associated with virtual terminals; instead, they're associated with pseudo-terminals.
- A program can play sounds through the system's sound card by sending audio data to **/dev/audio**.

```
$ cat /usr/share/sndconfig/sample.au > /dev/audio
```

Device	Name	Major	Minor
First floppy drive	`/dev/fd0`	2	0
Second floppy drive	`/dev/fd1`	2	1
Primary IDE controller, master device	`/dev/hda`	3	0
Primary IDE controller, master device, first partition	`/dev/hda1`	3	1
Primary IDE controller, secondary device	`/dev/hdb`	3	64
Primary IDE controller, secondary device, first partition	`/dev/hdb1`	3	65
Secondary IDE controller, master device	`/dev/hdc`	22	0
Secondary IDE controller, secondary device	`/dev/hdd`	22	64
First SCSI drive	`/dev/sda`	8	0
First SCSI drive, first partition	`/dev/sda1`	8	1
Second SCSI disk	`/dev/sdb`	8	16
Second SCSI disk, first partition	`/dev/sdb1`	8	17
First SCSI CD-ROM drive	`/dev/scd0`	11	0
Second SCSI CD-ROM drive	`/dev/scd1`	11	1

Figure 4: Some common block devices.

Device	Name	Major	Minor
Parallel port 0	`/dev/lp0` or `/dev/par0`	6	0
Parallel port 1	`/dev/lp1` or `/dev/par1`	6	1
First serial port	`/dev/ttyS0`	4	64
Second serial port	`/dev/ttyS1`	4	65
IDE tape drive	`/dev/ht0`	37	0
First SCSI tape drive	`/dev/st0`	9	0
Second SCSI tape drive	`/dev/st1`	9	1
System console	`/dev/console`	5	1
First virtual terminal	`/dev/tty1`	4	1
Second virtual terminal	`/dev/tty2`	4	2
Process's current terminal device	`/dev/tty`	5	0
Sound card	`/dev/audio`	14	4

Figure 5: Some common character devices.

1.5 Special Devices

Linux also provides several character devices that don't correspond to hardware devices. These entries all use the major device no. 1, which is associated with the Linux kernel's memory device instead of a device driver.

/dev/null

- The entry **/dev/null**, the null device, is very handy. It serves two purposes;
  - Linux discards any data written to **/dev/null**. Specify **/dev/null** as an output file in some context where the output is unwanted.

    ```
    $ verbose_command > /dev/null
    ```

  - Reading from **/dev/null** always results in an end-of-file. For instance, if you open a file descriptor to **/dev/null** using **open** and then attempt to read from the file descriptor, **read** will read no bytes and will return 0.

    ```
    $ cp /dev/null empty-file
    $ ls -l empty-file
    ```

/dev/zero

The device entry **/dev/zero** behaves as if it were an infinitely long file filled with 0 bytes. As much data as you'd try to read from **/dev/zero**, Linux generates enough 0 bytes.

```
$ hexdump -v /dev/zero
$ hexdump anyfile
```

/dev/full

The entry **/dev/full** behaves as if it were a file on a file system that has no more room. A write to **/dev/full** fails and sets errno to ENOSPC, which ordinarily indicates that the written-to device is full.

```
$ cp /etc/fstab /dev/full
cp: writing '/dev/full': No space left on device
```

Random Number Devices

- The special devices **/dev/random** and **/dev/urandom** provide access to the Linux kernel's built-in random number generation facility.
- Most software functions for generating random numbers, such as the **rand** function in the standard C library, actually generate pseudorandom numbers.
- Although these numbers satisfy some properties of random numbers, they are reproducible: If you start with the same *seed* value, you'll obtain the same sequence of pseudorandom numbers every time.
- By measuring the time delay between your input actions, such as keystrokes and mouse movements, Linux is capable of generating an unpredictable stream of high-quality random numbers.You can access this stream by reading from **/dev/random** and **/dev/urandom**. The data that you read is a stream of randomly generated bytes.

```
$ od -t a (d2,x1) /dev/random
$ od -t x1 /dev/urandom
$ man od
```

- Using random numbers from **/dev/random** in a program is easy, too. The program in Fig. 6 presents a function that generates a random number using bytes read from in **/dev/random**.
- Remember that **/dev/random** blocks a read until there is enough randomness available to satisfy it; you can use **/dev/urandom** instead if fast execution is more important and you can live with the potential lower quality of random numbers.

Loopback Devices

- A **loopback device** enables you to simulate a block device using an ordinary disk file.
- Imagine a disk drive device for which data is written to and read from a file named disk-image rather than to and from the tracks and sectors of an actual physical disk drive or disk partition.
- A loopback device enables you to use a file in this manner.
- Loopback devices are named **/dev/loop0**, **/dev/loop1**, and so on.

```
#include <assert.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
/* Return a random integer between MIN and MAX, inclusive.  Obtain
   randomness from /dev/random.  */
int random_number (int min, int max)
{
  /* Store a file descriptor opened to /dev/random in a static
     variable.  That way, we don't need to open the file every time
     this function is called.  */
  static int dev_random_fd = -1;
  char* next_random_byte;
  int bytes_to_read;
  unsigned random_value;
  /* Make sure MAX is greater than MIN.  */
  assert (max > min);
  /* If this is the first time this function is called, open a file
     descriptor to /dev/random.  */
  if (dev_random_fd == -1) {
    dev_random_fd = open ("/dev/random", O_RDONLY);
    assert (dev_random_fd != -1);
  }
  /* Read enough random bytes to fill an integer variable.  */
  next_random_byte = (char*) &random_value;
  bytes_to_read = sizeof (random_value);
  /* Loop until we've read enough bytes.  Since /dev/random is filled
     from user-generated actions, the read may block, and may only
     return a single random byte at a time.  */
  do {
    int bytes_read;
    bytes_read = read (dev_random_fd, next_random_byte, bytes_to_read);
    bytes_to_read -= bytes_read;
    next_random_byte += bytes_read;
  } while (bytes_to_read > 0);
  /* Compute a random number in the correct range.  */
  return min + (random_value % (max - min + 1));
}
```

Figure 6: Function to Generate a Random Number.

- A loopback device can be used in the same way as any other block device. In particular, you can construct a file system on the device and then mount that file system as you would mount the file system on an ordinary disk or partition (virtual file system).

- To construct a virtual file system and mount it with a loopback device, follow these steps:

  1. Create an empty file to hold the virtual file system. To construct a 10MB file named disk-image, invoke the following:

     ```
     $ dd if=/dev/zero of=/yourdirectory/disk-image count=20480
     $ ls -l /yourdirectory/disk-image
     ```

  2. The file that you've just created is filled with 0 bytes. Before you mount it, you must construct a file system. This sets up the various control structures needed to organize and store files, and builds the root directory.

     ```
     $ mke2fs -q /yourdirectory/disk-image
     ```

  3. Mount the file system using a loopback device.

     ```
     $ mkdir /yourdirectory/virtual-fs
     $ mount -o loop=/dev/loop0 /yourdirectory/disk-image
       /yourdirectory/virtual-fs
     $ df -h /yourdirectory/virtual-fs
     $ cd /yourdirectory/virtual-fs
     $ echo "Hello, world!" > test.txt
     $ ls -l
     $ cat test.txt
     $ cd /yourdirectory
     $ umount /yourdirectory/virtual-fs
     ```

1.6 ioctl

- The **ioctl** system call is an all-purpose interface for controlling hardware devices.
  - The first argument to **ioctl** is a file descriptor, which should be opened to the device that you want to control.
  - The second argument is a request code that indicates the operation that you want to perform.

```
#include <fcntl.h>
#include <linux/cdrom.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int main (int argc, char* argv[])
{
  /* Open a file descriptor to the device specified on the command line.  */
  int fd = open (argv[1], O_RDONLY);
  /* Eject the CD-ROM.  */
  ioctl (fd, CDROMEJECT);
  /* Close the file descriptor.  */
  close (fd);
  return 0;
}
```

Figure 7: Eject a CD-ROM.

- The program in Fig. 7 presents a short program that ejects the disk in a CD-ROM drive (if the drive supports this).

  ```
  $ ./cdrom-eject /dev/hdc
  ```