# Section 3: Syscalls, I/O, Sockets, and Networking

September 11, 2015

## Contents

# 1 Vocabulary

- **system call** - In computing, a system call is how a program requests a service from an operating system's kernel. This may include hardware-related services (for example, accessing a hard disk drive), creation and execution of new processes, and communication with integral kernel services such as process scheduling.
- **file descriptors** - File descriptors are an index into a file-descriptor table stored by the kernel. The kernel creates a file-descriptor in response to an open call and associates the file-descriptor with some abstraction of an underlying file-like object; be that an actual hardware device, or a file-system or something else entirely. Consequently a process's read or write calls that reference that file-descriptor are routed to the correct place by the kernel to ultimately do something useful.
- **int open(const char *path, int oflags)** - open is a system call that is used to open a new file and obtain its file descriptor.
- **size_t read(int fildes, void *buf, size_t nbytes)** - read is a system call used to read data into a buffer.
- **size_t write(int fildes, const void *buf, size_t nbytes)** - write is a system call that is used to write data out of a buffer.
- **int dup(int fildes)** - creates an alias for the provided file descriptor. dup always uses the smallest available file descriptor. Thus, if we called dup first thing in our program, then you could write to standard output by using file descriptor 3 (dup uses 3 because 0, 1, and 2 are already signed to stdin, stdout, stderr). You can determine the value of the new file descriptor by saving the return value from dup.
- **int dup2(int fildes, int fildes2)** - dup2 is a system call similar to dup. It duplicates one file descriptor, making them aliases, and then deleting the old file descriptor. This becomes very useful when attempting to redirect output, as it automatically takes care of closing the old file descriptor, performing the redirection in one elegant command. For example, if you wanted to redirect standard output to a file, then you would simply call dup2, providing the open file descriptor for the file as the first command and 1 (standard output) as the second command.
- **client server model, sockets** - Most interprocess communication uses the client server model. These terms refer to the two processes which will be communicating with each other. One of the two processes, the client, connects to the other process, the server, typically to make a request for information. A good analogy is a person who makes a phone call to another person. Notice that the client needs to know of the existence of and the address of the server, but the server does not need to know the address of (or even the existence of) the client prior to the connection being established.

  Notice also that once a connection is established, both sides can send and receive information.

  The system calls for establishing a connection are somewhat different for the client and the server, but both involve the basic construct of a socket. A socket is one end of an interprocess communication channel. The two processes each establish their own socket.

  The steps involved in establishing a socket on the client side are as follows:

  1. Create a socket with the `socket()` system call
  2. Connect the socket to the address of the server using the connect() system call
  3. Send and receive data. There are a number of ways to do this, but the simplest is to use the read() and write() system calls.

The steps involved in establishing a socket on the server side are as follows:

1. Create a socket with the `socket()` system call
2. Bind the socket to an address using the `bind()` system call. For a server socket on the Internet, an address consists of a port number on the host machine.
3. Listen for connections with the `listen()` system call
4. Accept a connection with the `accept()` system call. This call typically blocks until a client connects with the server.
5. Send and receive data

- **Signals** - A signal is a software interrupt, a way to communicate information to a process about the state of other processes, the operating system, and the hardware. A signal is an interrupt in the sense that it can change the flow of the program when a signal is delivered to a process, the process will stop what its doing, either handle or ignore the signal, or in some cases terminate, depending on the signal.
- **int signal(int signum, void (*handler)(int))** - signal() is the primary system call for signal handling, which given a signal and function, will execute the function whenever the signal is delivered. This function is called the signal handler because it handles the signal.

# 2 Problems

## 2.1 Signals

The following is a list of standard Linux signals:

```
Signal     Value     Action   Comment
------------------------------------------------------------------
SIGHUP        1       Term    Hangup detected on controlling terminal
                              or death of controlling process
SIGINT        2       Term    Interrupt from keyboard
SIGQUIT       3       Core    Quit from keyboard
SIGILL        4       Core    Illegal Instruction
SIGABRT       6       Core    Abort signal from abort(3)
SIGFPE        8       Core    Floating point exception
SIGKILL       9       Term    Kill signal
SIGSEGV      11       Core    Invalid memory reference
SIGPIPE      13       Term    Broken pipe: write to pipe with no
                              readers
SIGALRM      14       Term    Timer signal from alarm(2)
SIGTERM      15       Term    Termination signal
SIGUSR1   30,10,16    Term    User-defined signal 1
SIGUSR2   31,12,17    Term    User-defined signal 2
SIGCHLD   20,17,18    Ign     Child stopped or terminated
SIGCONT   19,18,25    Cont    Continue if stopped
SIGSTOP   17,19,23    Stop    Stop process
SIGTSTP   18,20,24    Stop    Stop typed at tty
SIGTTIN   21,21,26    Stop    tty input for background process
SIGTTOU   22,22,27    Stop    tty output for background process
```

#### 2.1.1 Warmup

Will Ctrl-c stop the following program? How do we stop it?

```
int main(){
  signal(SIGINT, SIG_IGN);
  while(1);
}
```

#### 2.1.2 Did you really want to quit?

Fill in the blanks for the following function using syscalls such that when we type Ctrl-c , the user is prompted with a message: "Do you really want to quit [y/n]? ", and if "y" is typed, the program quits. Otherwise, it continues along.

```
void sigint_handler(int sig)
{
    char c;
    printf(Ouch, you just hit Ctrl-C?. Do you really want to quit [y/n]?);
    c = getchar();
    if (c == 'y' || c == 'Y')
        ------------------------------;
}



int main() {
    ------------------------------;
    ...

}
```

### 2.2 Dup and Dup2

#### 2.2.1 Warmup

What does C print in the following code?

```
int
main(int argc, char **argv)
{
    int pid, status;
    int newfd;

    if ((newfd = open("output_file.txt", O_CREAT|O_TRUNC|O_WRONLY, 0644)) < 0) {
        exit(1);
    }
    printf("Luke, I am your...");
```

```
    dup2(newfd, 1);
    printf("father");
    exit(0);
}
```

### 2.2.2 Redirection: executing a process after dup2

Please refer to the dup2-c.c file in the section3/ folder in the Sections repo on Github. The code is also below for convenience.
Describe what happens, and what the output will be.

```
int
main(int argc, char **argv)
{
    int pid, status;
    int newfd;
    char *cmd[] = { "/bin/ls", "-al", "/", 0 };

    if (argc != 2) {
        fprintf(stderr, "usage: %s output_file\n", argv[0]);
        exit(1);
    }
    if ((newfd = open(argv[1], O_CREAT|O_TRUNC|O_WRONLY, 0644)) < 0) {
        perror(argv[1]);    /* open failed */
        exit(1);
    }
    printf("writing output of the command %s to \"%s\"\n", cmd[0], argv[1]);
    dup2(newfd, 1);
    execvp(cmd[0], cmd);
    perror(cmd[0]);     /* execvp failed */
    exit(1);
}
```

### 2.2.3 Redirecting in a new process

Please refer to the dup2-d.c file in the Sections repo on Github. If you do not have your laptop with you, share with your neighbor.

Describe what happens, and what the output will be.

## 2.3 Practice with Sockets

In this section, you will implement a client and server in C using stream sockets in the Internet domain. Please fill out the blank lines.
The following syscalls and structs may be of use:

```
int socket(int domain, int type, int protocol);

struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};

int bind(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen);

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

struct  hostent {
        char    *h_name;        /* official name of host */
        char    **h_aliases;    /* alias list */
        int     h_addrtype;     /* host address type */
        int     h_length;       /* length of address */
        char    **h_addr_list;  /* list of addresses from name server */
#define h_addr  h_addr_list[0]  /* address, for backward compatiblity */
};
```

Below are descriptions of the system calls:

- The bind() system call binds a socket to an address, in this case the address of the current host and port number on which the server will run. It takes three arguments, the socket file descriptor, the address to which is bound, and the size of the address to which it is bound. The second argument is a pointer to a structure of type sockaddr, but what is passed in is a structure of type sockaddr_in, and so this must be cast to the correct type. This can fail for a number of reasons, the most obvious being that this socket is already in use on this machine.
- The listen() system call allows the process to listen on the socket for connections. The first argument is the socket file descriptor, and the second is the size of the backlog queue, i.e., the number of connections that can be waiting while the process is handling a particular connection.
- The accept() system call causes the process to block until a client connects to the server. Thus, it wakes up the process when a connection from a client has been successfully established. It returns a new file descriptor, and all communication on this connection should be done using the new file descriptor. The second argument is a reference pointer to the address of the client on the other end of the connection, and the third argument is the size of this structure.
- The connect() function is called by the client to establish a connection to the server. It takes three arguments, the socket file descriptor, the address of the host to which it wants to connect (including the port number), and the size of this address. This function returns 0 on success and -1 if it fails.

client code:

```
int main(int argc, char *argv[])
{
    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;


    char buffer[256];
    if (argc < 3) {
       fprintf(stderr,"usage %s hostname port\n", argv[0]);
       exit(0);
    }
    portno = atoi(argv[2]);

    ____________________________; // creates a new socket
    if (sockfd < 0)
        error("ERROR opening socket");
        server = gethostbyname(argv[1]);
    if (server == NULL) {
        fprintf(stderr,"ERROR, no such host\n");
        exit(0);
    }
    ____________________________; // zero out serv_addr
    ____________________________; // set address family to internet

    ____________________________; // set up server address
    ____________________________; // set up server port no

    if (_____________________________ < 0)  // try to connect to serve_addr
        error("ERROR connecting");
    printf("Please enter the message you wish to send: ");
    ____________________________; // read from stdin into buffer
    ____________________________; // send buffer contents to server
    if (n < 0)
         error("ERROR writing to socket");
    bzero(buffer,256);
    ____________________________; // read reply into buffer
    if (n < 0)
         error("ERROR reading from socket");
    printf("%s\n",buffer);
    ____________________________; // close socket
    return 0;
}
```

server.c code:

```
/* A simple server in the internet domain using TCP
   The port number is passed as an argument */
```

```
int main(int argc, char *argv[])
{
     int sockfd, newsockfd, portno;
     socklen_t clilen;
     char buffer[256];
     struct sockaddr_in serv_addr, cli_addr;
     int n;
     if (argc < 2) {
         fprintf(stderr,"ERROR, no port provided\n");
         exit(1);
     }
     ______________________; // create socket
     if (sockfd < 0)
        error("ERROR opening socket");

    ______________________; // set up address family
    ______________________; // set IP address of the host
    ______________________; // set port no


     if (_____________________) < 0)  { // binds socket to address
              error("ERROR on binding");
     }
     ______________________; // listen on the socket for connections
     newsockfd = _____________________; // accept connection from client
     if (newsockfd < 0)
          error("ERROR on accept");
     bzero(buffer,256);
     n = ______________________; // read from socket into buffer
     if (n < 0) error("ERROR reading from socket");

     n = ___________________________; // write "I got your message" to socket
     if (n < 0) error("ERROR writing to socket");
     ______________________; // close newsockfd
    ______________________; // close sockfd
    return 0;
}
```

Currently our server can only handle a single client connection before terminating. Can you rewrite the server code so that it accepts and handles multiple connections? Hint: use fork().