LABORATORY

SOCKET PROGRAMMING

What is a socket?

Using sockets

- Types (Protocols)
- Associated functions
- Styles

WHAT IS A SOCKET?

An interface between application and network

- The application creates a socket
- The socket *type* dictates the style of communication
  - reliable vs. best effort
  - connection-oriented vs. connectionless

Once configured the application can

- pass data to the socket for network transmission
- receive data from the socket (transmitted through the network by some other host)

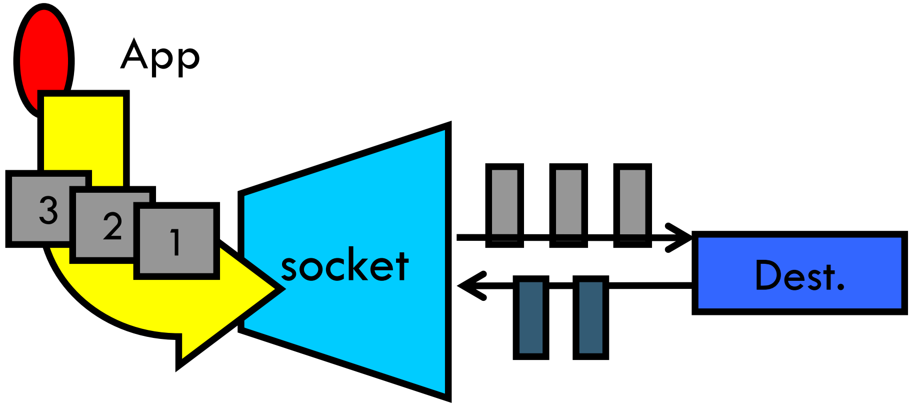
TWO ESSENTIAL TYPES OF SOCKETS

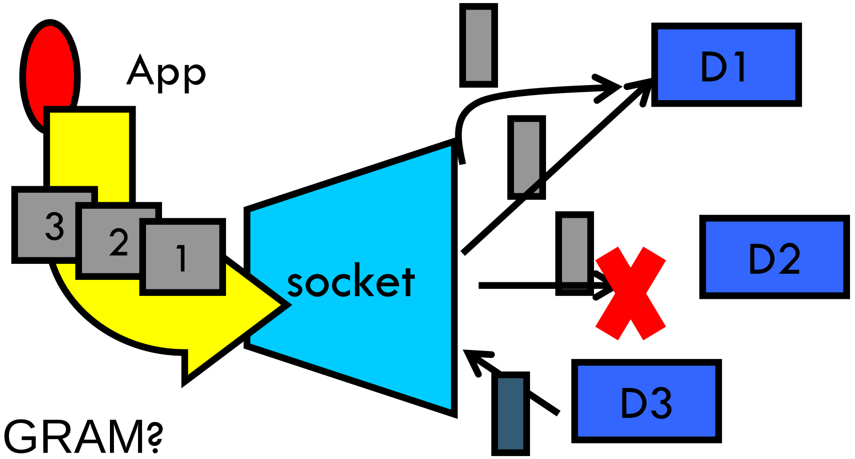
SOCK_STREAM

- a.k.a. TCP
- reliable delivery
- in-order guaranteed
- connection-oriented
- bidirectional

SOCK_DGRAM

- a.k.a. UDP
- unreliable delivery
- no order guarantees
- no notion of “connection” – app indicates dest. for each packet
- can send or receive





Q: why have type SOCK_DGRAM?

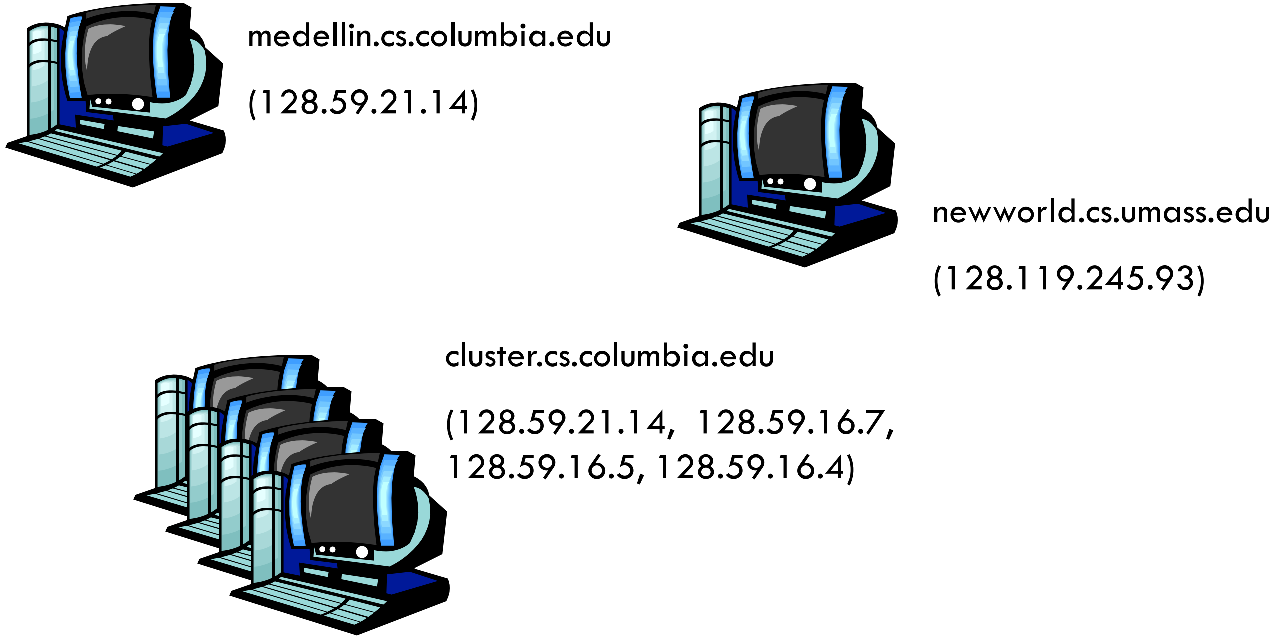
SOCKET CREATION IN C: SOCKET

int s = socket(domain, type, protocol);

- S: socket descriptor, an integer (like a file-handle)
- domain: integer, communication domain
  - e.g., PF_INET (IPv4 protocol) – typically used
- type: communication type
  - SOCK_STREAM: reliable, 2-way, connection-based service
  - SOCK_DGRAM: unreliable, connectionless,
  - other values: need root permission, rarely used, or obsolete
- protocol: specifies protocol (see file /etc/protocols for a list of options) - usually set to 0

NOTE: socket call does not specify where data will be coming from, nor where it will be going to – it just creates the interface!

A SOCKET-EYE VIEW OF THE INTERNET



medellin.cs.columbia.edu

(128.59.21.14)

newworld.cs.umass.edu

(128.119.245.93)

cluster.cs.columbia.edu

(128.59.21.14, 128.59.16.7, 128.59.16.5, 128.59.16.4)

Each host machine has an IP address

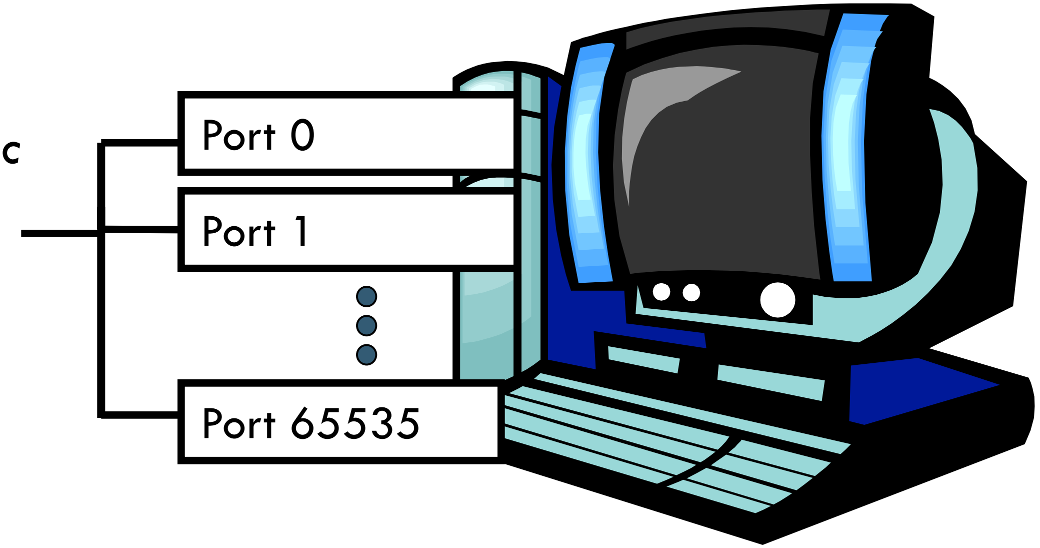
When a packet arrives at a host

PORTS

Each host has 65,536 ports

Some ports are *reserved for specific apps*

- 20,21: FTP
- 23: Telnet
- 80: HTTP
- see RFC 1700 (about 2000 ports are reserved)



❐ A socket provides an interface to send data to/from the network through a port

ADDRESSES, PORTS AND SOCKETS

Like apartments and mailboxes

- You are the application
- Your apartment building address is the address
- Your mailbox is the port
- The post-office is the network
- The socket is the key that gives you access to the right mailbox (one difference: assume outgoing mail is placed by you in your mailbox)

Q: How do you choose which port a socket connects to?

THE BIND FUNCTION

associates and (can exclusively) reserves a port for use by the socket

int status = bind(sockid, &addrport, size);

- status: error status, = -1 if bind failed
- sockid: integer, socket descriptor
- addrport: struct sockaddr, the (IP) address and port of the machine (address usually set to INADDR_ANY – chooses a local address)
- size: the size (in bytes) of the addrport structure

bind can be skipped for both types of sockets. When and why?

SKIPPING THE BIND

SOCK_DGRAM:

- if only sending, no need to bind. The OS finds a port each time the socket sends a pkt
- if receiving, need to bind

SOCK_STREAM:

- destination determined during conn. setup
- don't need to know port sending from (during connection setup, receiving end is informed of port)

CONNECTION SETUP (SOCK_STREAM)

Recall: no connection setup for SOCK_DGRAM

A connection occurs between two kinds of participants

- passive: waits for an active participant to request connection
- active: initiates connection request to passive side

Once connection is established, passive and active participants are “similar”

- both can send & receive data
- either can terminate the connection

CONNECTION SETUP CONT'D

Passive participant

- step 1: listen (for incoming requests)
- step 3: accept (a request)
- step 4: data transfer

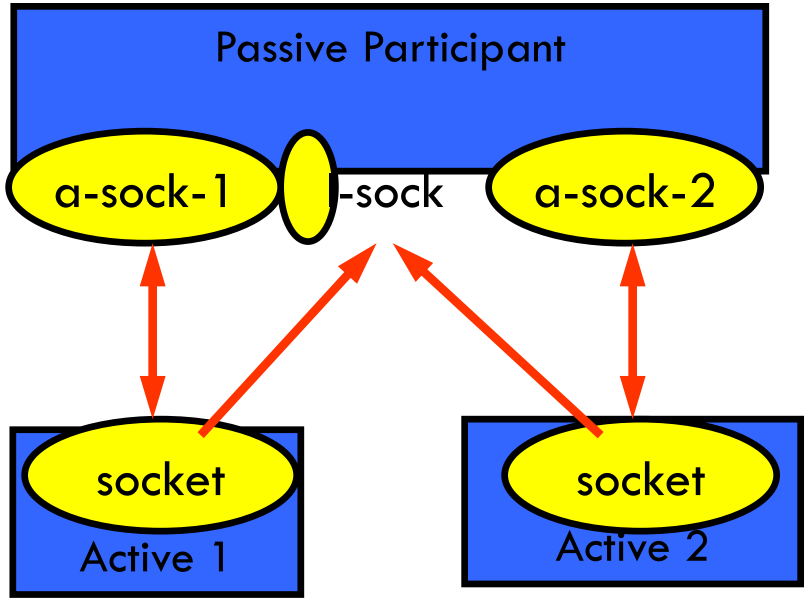
The accepted connection is on a new socket

The old socket continues to listen for other active participants

Why?

Active participant

- step 2: request & establish connection
- step 4: data transfer



CONNECTION SETUP: LISTEN & ACCEPT

Called by passive participant

int status = listen(sock, queuelen);

- status: 0 if listening, -1 if error
- sock: integer, socket descriptor
- queuelen: integer, # of active participants that can “wait” for a connection
- listen is **non-blocking**: returns immediately

int s = accept(sock, &name, &namelen);

- s: integer, the new socket (used for data-transfer)
- sock: integer, the orig. socket (being listened on)
- name: struct sockaddr, address of the active participant
- namelen: sizeof(name): value/result parameter
  - must be set appropriately before call
  - adjusted by OS upon return
- accept is **blocking**: waits for connection before returning

CONNECT CALL

int status = connect(sock, &name, namelen);

- status: 0 if successful connect, -1 otherwise
- sock: integer, socket to be used in connection
- name: struct sockaddr: address of passive participant
- namelen: integer, sizeof(name)

connect is **blocking**

SENDING / RECEIVING DATA

With a connection (SOCK_STREAM):

- int count = send(sock, &buf, len, flags);
  - count: # bytes transmitted (-1 if error)
  - buf: char[], buffer to be transmitted
  - len: integer, length of buffer (in bytes) to transmit
  - flags: integer, special options, usually just 0
- int count = recv(sock, &buf, len, flags);
  - count: # bytes received (-1 if error)
  - buf: void[], stores received bytes
  - len: # bytes received
  - flags: integer, special options, usually just 0
- Calls are **blocking** [returns only after data is sent (to socket buf) / received]

SENDING / RECEIVING DATA (CONT'D)

Without a connection (SOCK_DGRAM):

- int count = sendto(sock, &buf, len, flags, &addr, addrlen);
  - count, sock, buf, len, flags: same as send
  - addr: struct sockaddr, address of the destination
  - addrlen: sizeof(addr)
- int count = recvfrom(sock, &buf, len, flags, &addr, &addrlen);
  - count, sock, buf, len, flags: same as recv
  - name: struct sockaddr, address of the source
  - namelen: sizeof(name): value/result parameter

Calls are **blocking** [returns only after data is sent (to socket buf) / received]

CLOSE

When finished using a socket, the socket should be closed:

status = close(s);

- status: 0 if successful, -1 if error
- s: the file descriptor (socket being closed)

Closing a socket

- closes a connection (for SOCK_STREAM)
- frees up the port used by the socket

THE STRUCT SOCKADDR

The generic:

```
struct sockaddr {
    u_short sa_family;
    char sa_data[14];
};
```

- sa_family
  - specifies which address family is being used
  - determines how the remaining 14 bytes are used

The Internet-specific:

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

- sin_family = AF_INET
- sin_port: port # (0-65535)
- sin_addr: IP-address
- sin_zero: unused

ADDRESS AND PORT BYTE-ORDERING

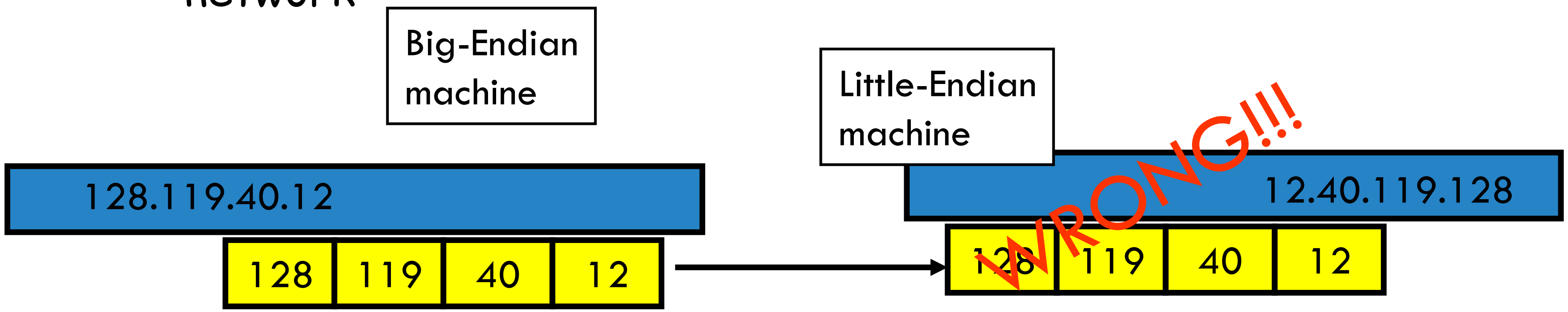
Address and port are stored as integers

- u_short sin_port; (16 bit)
- in_addr sin_addr; (32 bit)

```
struct in_addr {
  u_long s_addr;
};
```

❐ Problem:

- different machines / OS's use different word orderings
  - little-endian: lower bytes first
  - big-endian: higher bytes first
- these machines may communicate with one another over the network



Big-Endian machine

128.119.40.12

128	119	40	12

Little-Endian machine

12.40.119.128

128	119	40	12

WRONG!!!

SOLUTION: NETWORK BYTE-ORDERING

Defs:

- Host Byte-Ordering: the byte ordering used by a host (big or little)
- Network Byte-Ordering: the byte ordering used by the network – always big-endian

Any words sent through the network should be converted to Network Byte-Order prior to transmission (and back to Host Byte-Order once received)

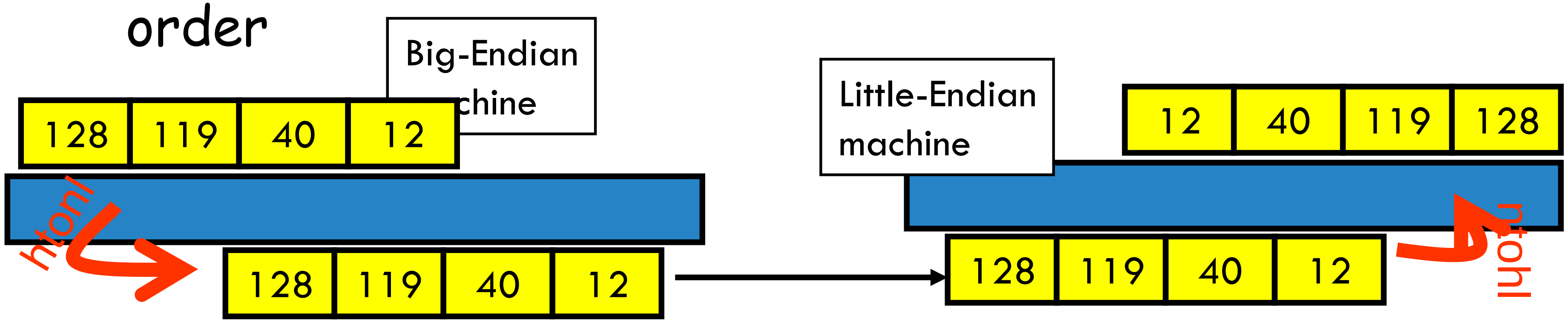
Q: should the socket perform the conversion automatically?

❐ Q: Given big-endian machines don't need conversion routines and little-endian machines do, how do we avoid writing two versions of code?

UNIX'S BYTE-ORDERING FUNCS

```
u_long htonl(u_long x);        u_long ntohl(u_long x);
u_short htons(u_short x);      u_short ntohs(u_short x);
```

- ❒ On big-endian machines, these routines do nothing
- ❒ On little-endian machines, they reverse the byte order



- ❒ Same code would have worked regardless of endian-ness of the two machines

DEALING WITH BLOCKING CALLS

Many of the functions we saw block until a certain event

- accept: until a connection comes in
- connect: until the connection is established
- recv, recvfrom: until a packet (of data) is received
- send, sendto: until data is pushed into socket's buffer
  - Q: why not until received?

For simple programs, blocking is convenient

What about more complex programs?

- multiple connections
- simultaneous sends and receives
- simultaneously doing non-networking processing

DEALING W/ BLOCKING (CONT'D)

Options:

- create multi-process or multi-threaded code
- turn off the blocking feature (e.g., using the fcntl file-descriptor control function)
- use the select function call.

What does select do?

- can be permanent blocking, time-limited blocking or non-blocking
- input: a set of file-descriptors
- output: info on the file-descriptors' status
- i.e., can identify sockets that are "ready for use": calls involving that socket will return immediately

SELECT FUNCTION CALL

int status = select(nfds, &readfds, &writefds, &exceptfds, &timeout);

- status: # of ready objects, -1 if error
- nfds: 1 + largest file descriptor to check
- readfds: list of descriptors to check if read-ready
- writefds: list of descriptors to check if write-ready
- exceptfds: list of descriptors to check if an exception is registered

- timeout: time after which select returns, even if nothing ready - can be 0 or ∞

  (point timeout parameter to NULL for ∞)

TO BE USED WITH SELECT:

Recall select uses a structure, struct fd_set

- it is just a bit-vector
- if bit *i* is set in [readfds, writefds, exceptfds], select will check if file descriptor (i.e. socket) *i* is ready for [reading, writing, exception]

Before calling select:

- FD_ZERO(&fdvar): clears the structure
- FD_SET(i, &fdvar): to check file desc. *i*

After calling select:

- int FD_ISSET(i, &fdvar): boolean returns TRUE iff *i* is “ready”

OTHER USEFUL FUNCTIONS

bzero(char* c, int n): 0's n bytes starting at c

gethostname(char *name, int len): gets the name of the current host

gethostbyaddr(char *addr, int len, int type): converts IP hostname to structure containing long integer

inet_addr(const char *cp): converts dotted-decimal char-string to long integer

inet_ntoa(const struct in_addr in): converts long to dotted-decimal notation

Warning: check function assumptions about byte-ordering (host or network). Often, they assume parameters / return solutions in network byte-order

RELEASE OF PORTS

Sometimes, a “rough” exit from a program (e.g., ctrl-c) does not properly free up a port

Eventually (after a few minutes), the port will be freed

To reduce the likelihood of this problem, include the following code:

```
#include <signal.h>
void cleanExit(){exit(0);}
```

- in socket code:

```
signal(SIGTERM, cleanExit);
signal(SIGINT, cleanExit);
```

FINAL THOUGHTS

Make sure to #include the header files that define used functions

Check man-pages and course web-site for additional info

EXERCISE

WWW Client

The simple WWW client program should do the following activities:

1. From the command-line, read (1) the URL from which you can extract the name of the remote WWW server and the file to retrieve and (2) the server port number. Create a socket that is connected to the server machine at the specified port (e.g., HTTP port 80) [getservbyname, gethostbyname, socket, connect].

2. Send a request to the WWW server using the HTTP protocol format. This will look something like this: `GET /index.html HTTP/1.0\n\n` Note that it's very important to include the two trailing newlines -- they are required by the HTTP protocol.

3. Read all the data from the HTTP connection and write it to a temporary file created in your WWW cache (e.g., /tmp/*yourloginname*) on the local host [creat,read/write].

4. Spawn an external viewer [fork/exec] to display the file. You can determine the type of viewer to spawn in two ways:
    1. ***Client-side file suffix*** -- The client can use the file suffix (e.g., *.ps should spawn ghostview, *.gif should spawn xv, an html file should spawn lynx, and a regular text file should spawn /usr/ucb/more, etc.). If the file is compressed (e.g., *.gz, *.Z, or *.zip) then uncompress it before viewing it.
    2. ***Server-side MIME content type information*** -- A more robust way to determine what type of the viewer to spawn is to utilize the `Content-type:` header returned by the server. For instance, the .

The client should simply print out the appropriate error message [perror] and exit with a return status of 1 if any of the system calls fail to work properly. If everything works correctly, the program should exit with a return status of 0.