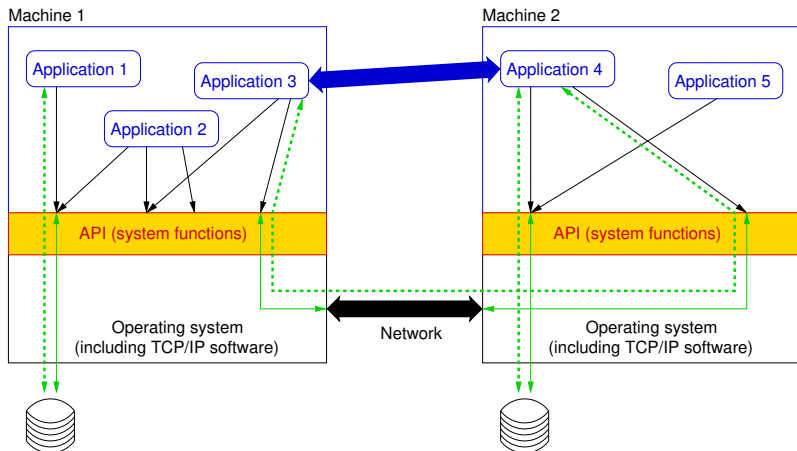


Application Programming Interfaces

Stefan D. Bruda

CS 464/564, Fall 2023

SYSTEM CALLS





- Where do we find API descriptions?
 - In **Section 2** of the **manual pages**
 - **Section 3** is dedicated to **library functions**, which are wrappers over APIs (implemented using calls to the respective API)
 - Examples:
 - man -S2 open (often man 2 open will also work)
 - man -S3 printf (or man 3 printf)
- Some interesting (for us) APIs:
 - Process management (`fork` and company — already seen)
 - File I/O
 - Terminal I/O (`cin`, `cout` / `scanf`, `printf`)
 - The TCP/IP implementation (**Berkeley sockets**)



- Where do we find API descriptions?
 - In **Section 2** of the **manual pages**
 - **Section 3** is dedicated to **library functions**, which are wrappers over APIs (implemented using calls to the respective API)
 - Examples:
 - man -S2 open (often man 2 open will also work)
 - man -S3 printf (or man 3 printf)
- Some interesting (for us) APIs:
 - Process management (`fork` and company — already seen)
 - File I/O
 - Terminal I/O (`cin`, `cout` / `scanf`, `printf`) — **library functions, not API!**
 - The TCP/IP implementation (**Berkeley sockets**)



Operation	Meaning
open	prepares a file for I/O operations returns a file descriptor (<code>int</code>) used by all the other operations
close	terminates the use of a previously opened file/device
read	obtain data from a file/device
write	write data to a file/device
lseek	move to some position in the file/device (not applicable to all devices)

- For example,

```
char result[256];
int file = open("echo", O_RDONLY);
if (file == -1)
    return 1;
read(file, result, 255);
close(file);
```



FILE I/O (CONT'D)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>

int main (int argc, char** argv) {
    int file = open(argv[1],O_WRONLY|O_CREAT|O_APPEND,S_IRUSR|S_IWUSR);
    int i = 0; char prefix[12]; char message[256] = "something";
    if (file == -1)
        return 1;
    while (true) {
        i++;
        printf("Message: ");
        fgets(message,255,stdin);
        if (message[strlen(message)-1] == '\n')
            message[strlen(message)-1] == '\0';
        if (strlen(message) == 0)
            break;
        snprintf(prefix,12,"%d: ",i);
        write(file,prefix,strlen(prefix));
        write(file,message,strlen(message));
        write(file,"\n",1);
    }
    close(file);
}
```



```
int readline(int fd, char* buf_str, size_t max) {
    size_t i;
    for (i = 0; i < max; i++) {
        char tmp;
        int what = read(fd,&tmp,1);
        if (what == 0 || tmp == '\n') {
            buf_str[i] = '\0';
            return i;
        }
        buf_str[i] = tmp;
    }
    buf_str[i] = '\0';
    return i;
}
```

```
int dbf = open("myfile",O_RDONLY);
if (dbf == -1) {
    perror("myfile");
    exit(1);
}
char message[256];
int nc = readline(dbf,message,255);
```



```
const int recv_nodata = -2;

int readline(const int fd, char* buf, const size_t max) {
    size_t i;
    int begin = 1;

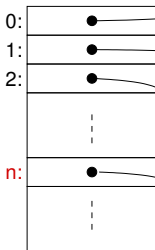
    for (i = 0; i < max; i++) {
        char tmp;
        int what = read(fd,&tmp,1);
        if (what == -1) return -1;
        if (begin) {
            if (what == 0)
                return recv_nodata;
            begin = 0;
        }
        if (what == 0 || tmp == '\n') {
            buf[i] = '\0';
            return i;
        }
        buf[i] = tmp;
    }
    buf[i] = '\0';
    return i;
}
```


FILE DESCRIPTORS



```
int dbf = open("myfile", O_RDONLY);
```

Descriptor table
(one per process)



data structure for file
0 (stdin)

data structure for file
1 (stdout)

data structure for file
2 (stderr)

data structure for file
n ("myfile")

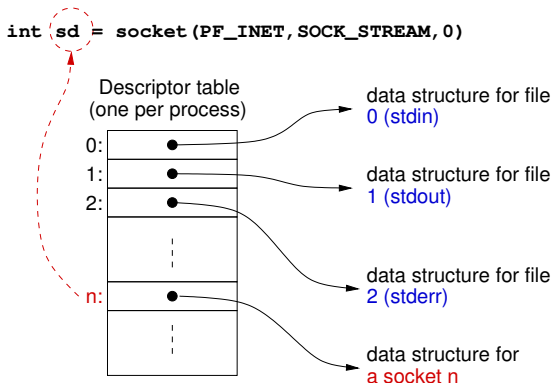


- TCP/IP is a **loosely specified protocol**
- In other words, the TCP/IP standard does **not** specify the API, only the functionality
- A TCP/IP interface must support the following operations:
 - **Allocate** resources for communication
 - Specify **communication endpoints**
 - **Initiate** a connection (client) or **wait** for connection (server)
 - **Send** and **receive** data
 - Identify **data arrival**
 - Generate **urgent data** and handle incoming urgent data
 - **Terminate** a connection, **handle connection termination**, **abort communication**
 - Handle **error conditions**
 - **Release** resources
- We use one (by far the most popular) TCP/IP implementation called **Berkeley sockets**



BERKELEY SOCKETS

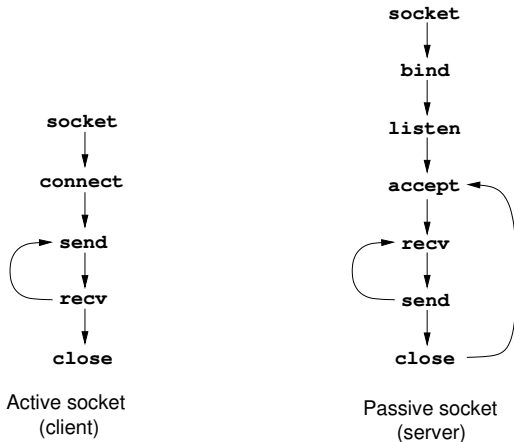
- Recall that an application that wants to access a file uses `open`
 - Further recall that `open` returns a file descriptor
- An application that wants to communicate creates a socket using `socket`
 - `socket` also returns a **descriptor**



WHAT TO DO WITH SOCKETS



- We have a socket, what do we do with it?
- We can convince it to be either **passive** (for servers) or **active** (for clients).





Who	Call	Meaning
Both	<code>socket</code>	Creates a socket, returns a descriptor
Client	<code>connect</code>	Connects to a remote server (client)
Server	<code>bind</code>	Associates (“binds”) the socket to a local IP address and a port number
	<code>listen</code>	Place the socket into passive mode (also sets the length of the queue)
	<code>accept</code>	Accept the next incoming connection (server)
Both	<code>send</code>	Send data (can also use <code>write</code>)
	<code>recv</code>	Receive data (can also use <code>read</code>)
	<code>close</code>	Terminate communication and destroys the descriptor

- Necessary headers:

```
#include <sys/types.h>
#include <sys/socket.h>
```



- In principle, there exist **address families** to hold different type of addresses
- In practice, TCP/IP uses one family of addresses, called AF_INET
- An address (within an address family) should contain at least the **address of some machine** and a **port number**
- The TCP/IP API includes C structures for address endpoints:

```
struct sockaddr_in {
    ...
    sa_family_t      sin_family; /* Address family, AF_INET for us */
    unsigned short int sin_port;  /* Port number */
    struct in_addr    sin_addr;  /* IP address */
    ...
};

struct in_addr {
    unsigned int s_addr; /* 32 bits, i.e., 4 bytes */
};
```



FINDING AN ADDRESS

- Usually, a server address is given as a **machine name** or an **IP address in dotted decimal notation**:

`cs.ubishops.ca` or `207.162.99.35`

- No matter how the server is specified (as dotted address or name), we have to obtain the actual IP address
- `gethostbyname` does the job no matter how the server is specified and puts the result in the following structure:

```
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;    /* Address from name server   */
#define h_addr h_addr_list[0] /* Address, for backward compatibility */
};
```



FINDING AN ADDRESS (CONT'D)

- Actually, `h_addr_list` is an array that contains the **bytes** of a network address
- They are given in **network byte order** (aka big endian, “big end first”), i.e., exactly in the right order to be put in a `sockaddr_in` structure

```
#include <stdio.h>
#include <netdb.h>
```

```
extern int h_errno;
```

```
int main (int argc, char** argv) {
    struct hostent* host_info = gethostbyname(argv[1]);
    if (host_info != NULL) {
        printf("%s is ", argv[1]);
        for (int i = 0; i < host_info -> h_length; i++)
            printf("%d.", (unsigned char)((host_info -> h_addr)[i]));
        printf("\n");
    }
    else
        printf("Host lookup error %d: %s\n", h_errno, hstrerror(h_errno));
}
```




FINDING A WELL-KNOWN PORT

- If you know the name of a standard service, you can find the port number associated to that service using `getservbyname` which puts the result in the following structure:

```
struct servent {
    char *s_name;          /* Official service name. */
    char **s_aliases;     /* Alias list. */
    int s_port;           /* Port number. */
    char *s_proto;        /* Protocol to use. */
};
```

- For example:

```
struct servent* service = getservbyname("ftp","tcp");
if (service != NULL) {
    printf("%s\n%d\n%s\n",
        service -> s_name,
        service -> s_port,
        service -> s_proto);
}
```

```
< godel:slides/code > ./a.out
ftp
21
tcp
```