WEEK 6: Task

Date: 15/09/2019

4	Threads: Build a Multi-Threaded Server to transfer a file from server to client using Producer-Consumer Model.

OBJECTIVE:

- **Familiarizing with basic system calls open(), read(), write() and close() to do System Programming using C language.**
- **Understand the concepts of Threads and its functionalities in comparison with processes.**
- **Understand the concepts of Semaphores used to synchronize the processes.**
- **Understand socket inter process communication and its system calls used. (Skeleton socket client server program will be shared for lab tasks)**

Introduction to System Calls

System programming starts with system calls. System calls (often shorted to syscalls) are function invocations made from user space—your text editor, favorite game, and so on—into the kernel (the core internals of the system) in order to request some service or resource from the operating system.

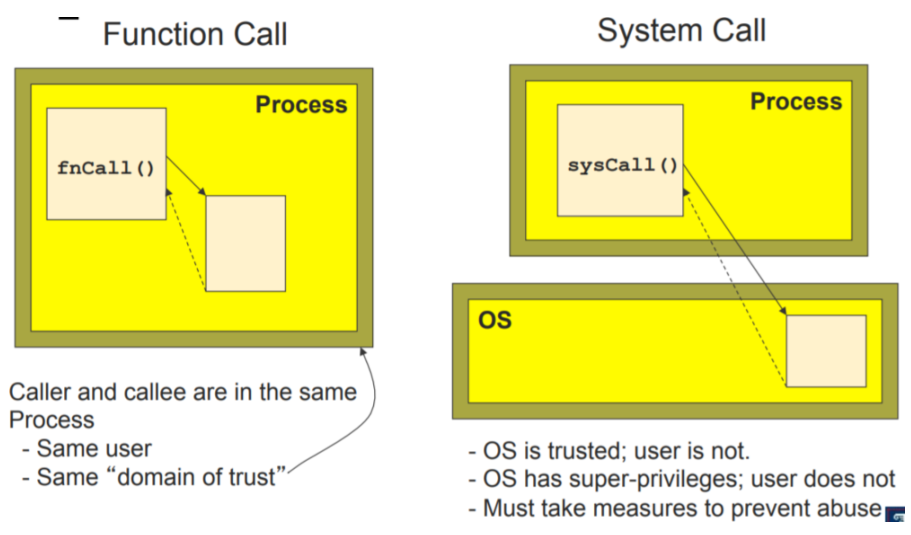
It is not possible to directly link user-space applications with kernel space. For reasons of security and reliability, user-space applications must not be allowed to directly execute kernel code or manipulate kernel data. Instead, the kernel must provide a mechanism by which a user-space application can "signal" the kernel that it wishes to invoke a system call. The applications can then trap into the kernel through this well-defined mechanism, and execute only code that the kernel allows it to execute.

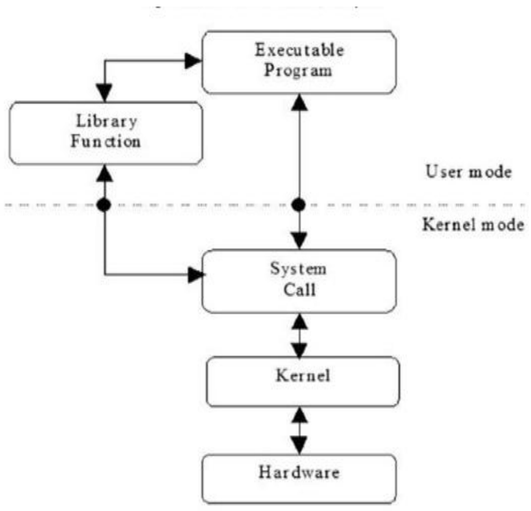
System Calls are

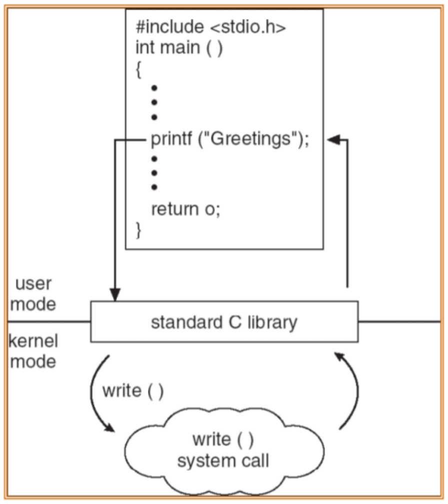
- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface** (**API**) rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

- System Calls
  - A request to the operating system to perform some activity
- System calls are expensive
  - The system needs to perform many things before executing a system call
    - The computer (hardware) saves its state
    - The OS code takes control of the CPU, privileges are updated.
    - The OS examines the call parameters
    - The OS performs the requested function
    - The OS saves its state (and call results)
    - The OS returns control of the CPU to the caller

System Calls versus Function Calls?


Function Call
Process
fnCall()
Caller and callee are in the same Process
- Same user
- Same "domain of trust"
System Call
Process
sysCall()
OS
- OS is trusted; user is not.
- OS has super-privileges; user does not
- Must take measures to prevent abuse



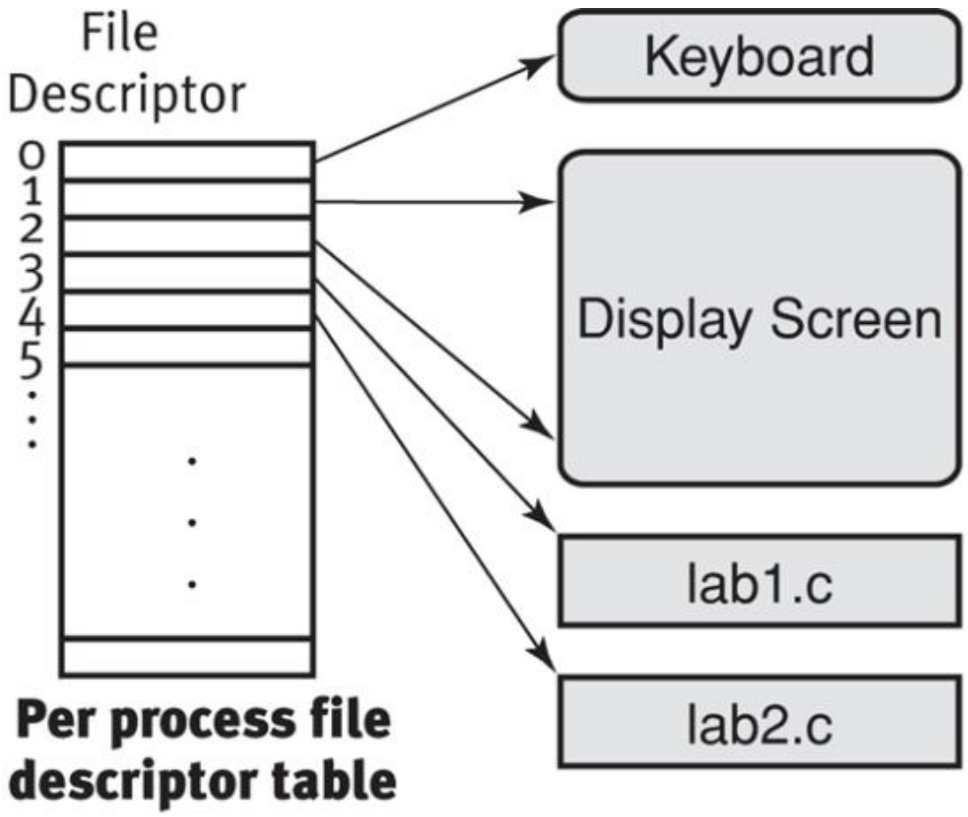


File Structure Related System Calls

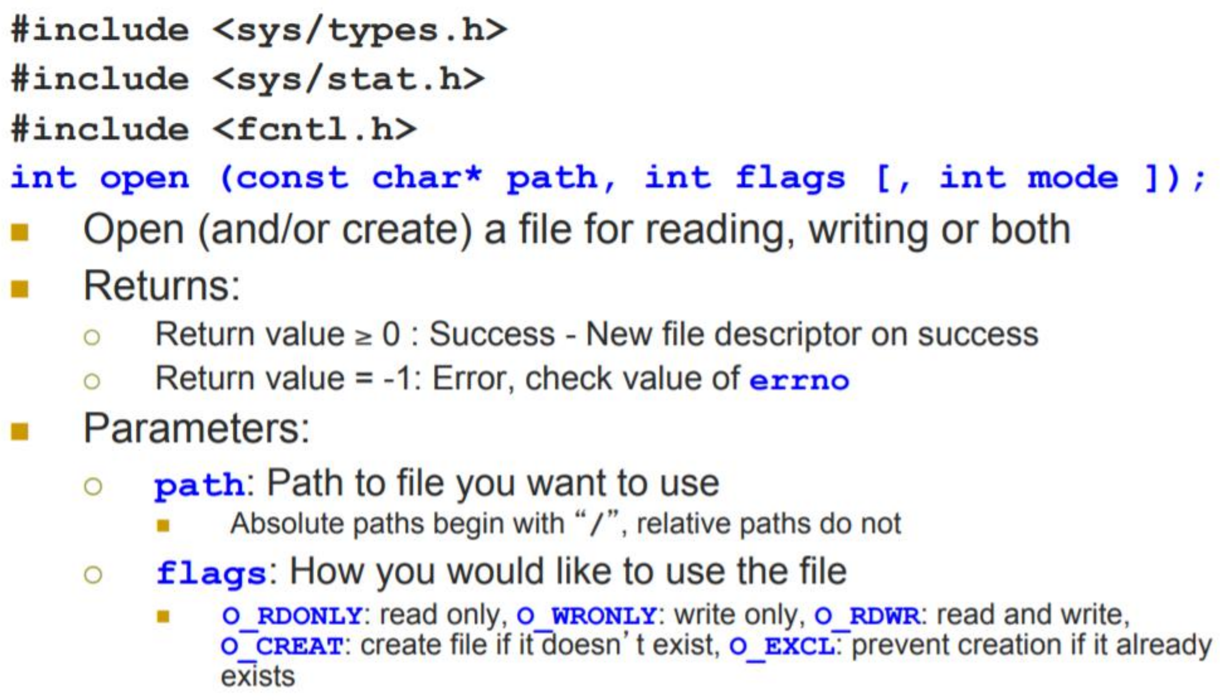
The file structure related system calls available in the UNIX system let us use the following system calls.

- creat()
- open()
- close()
- read()
- write()
- lseek()
- dup()
- link()
- unlink()
- stat()
- fstat()
- access()
- chmod()
- chown()
- umask()
- ioctl()

- Every process in Unix has three predefined file descriptors
  - File descriptor 0 is standard input (`STDIN`)
  - File descriptor 1 is standard output (`STDOUT`)
  - File descriptor 2 is standard error (`STDERR`)
- Read from standard input,
  - `read(0, ...);`
- Write to standard output
  - `write(1, ...);`

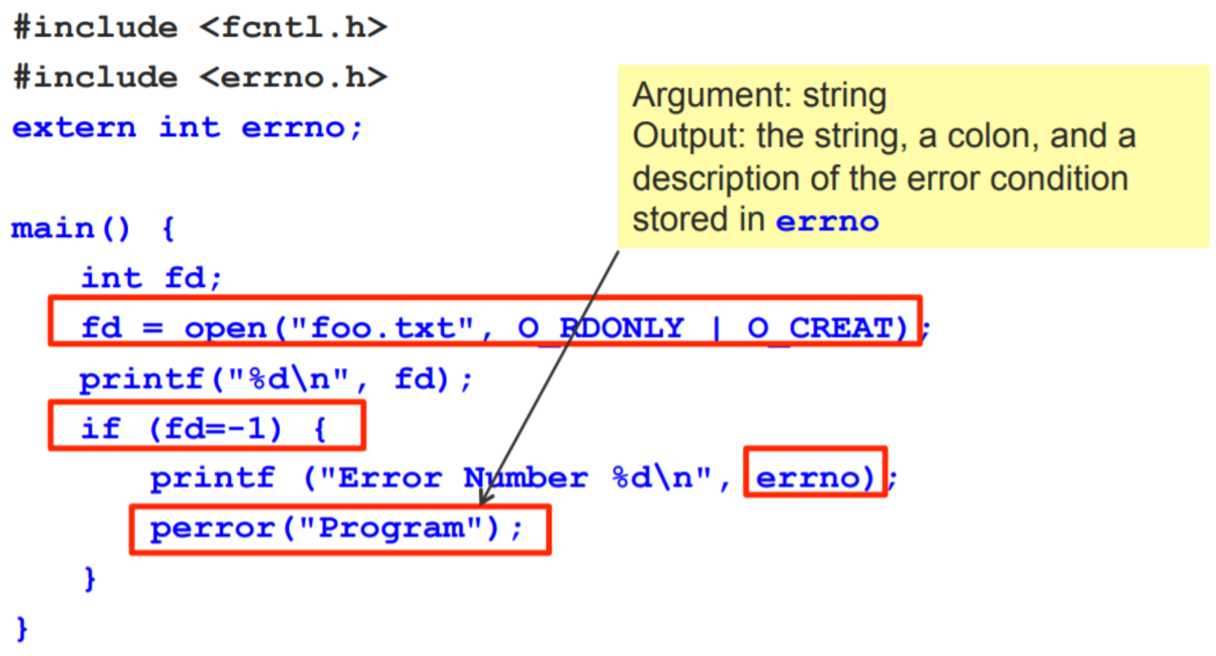


1. open() System Call



```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open (const char* path, int flags [, int mode ]);
```

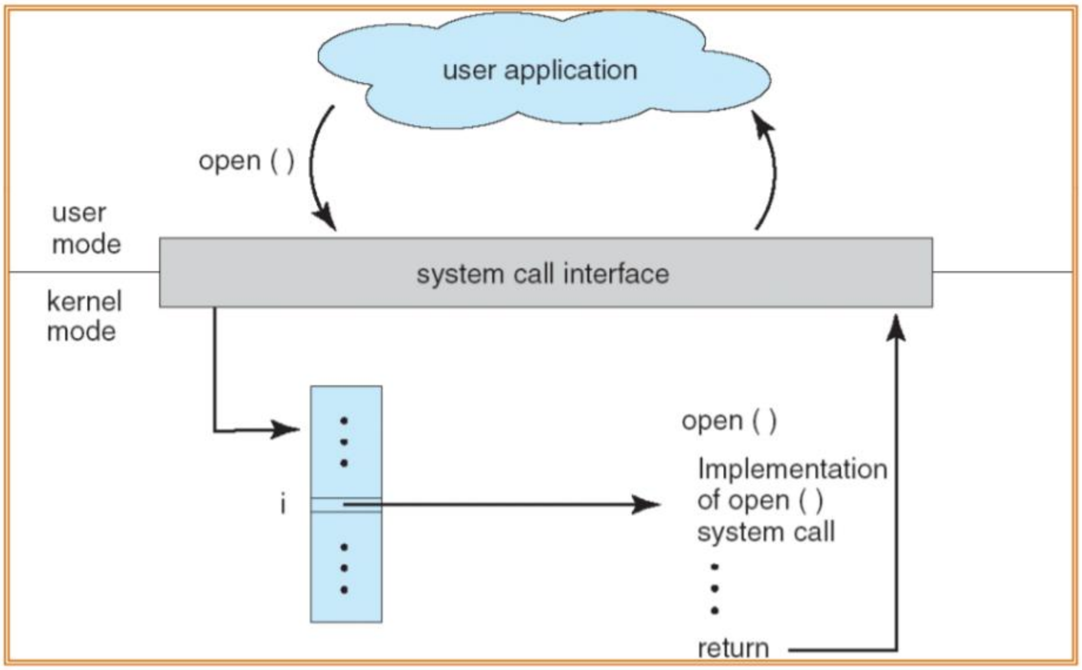
- Open (and/or create) a file for reading, writing or both
- Returns:
  - Return value ≥ 0 : Success - New file descriptor on success
  - Return value = -1: Error, check value of `errno`
- Parameters:
  - `path`: Path to file you want to use
    - Absolute paths begin with "/", relative paths do not
  - `flags`: How you would like to use the file
    - `O_RDONLY`: read only, `O_WRONLY`: write only, `O_RDWR`: read and write, `O_CREAT`: create file if it doesn't exist, `O_EXCL`: prevent creation if it already exists



```
#include <fcntl.h>
#include <errno.h>
extern int errno;

main() {
    int fd;
    fd = open("foo.txt", O_RDONLY | O_CREAT);
    printf("%d\n", fd);
    if (fd=-1) {
        printf ("Error Number %d\n", errno);
        perror("Program");
    }
}
```

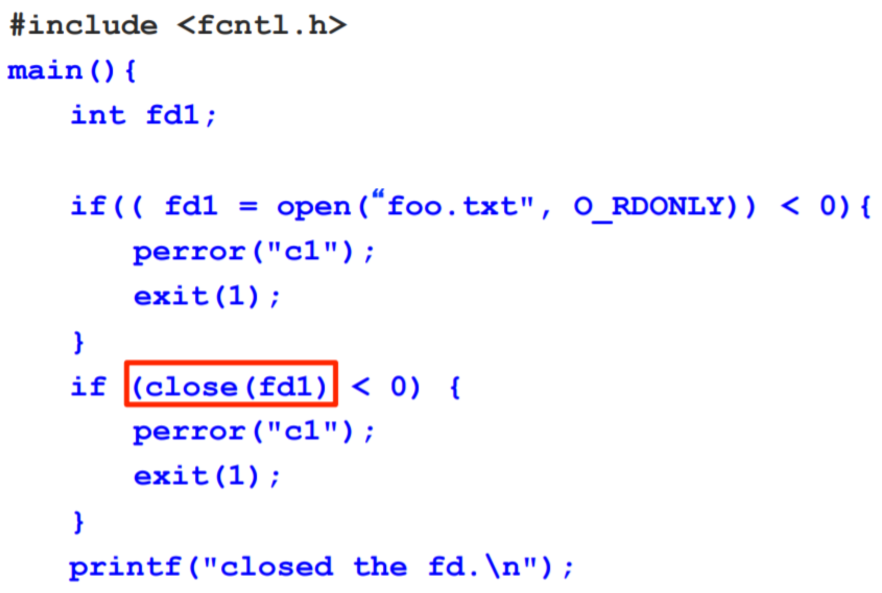
Argument: string
Output: the string, a colon, and a description of the error condition stored in `errno`



2. close() System Call

```
#include <fcntl.h>
int close(int fd);
```

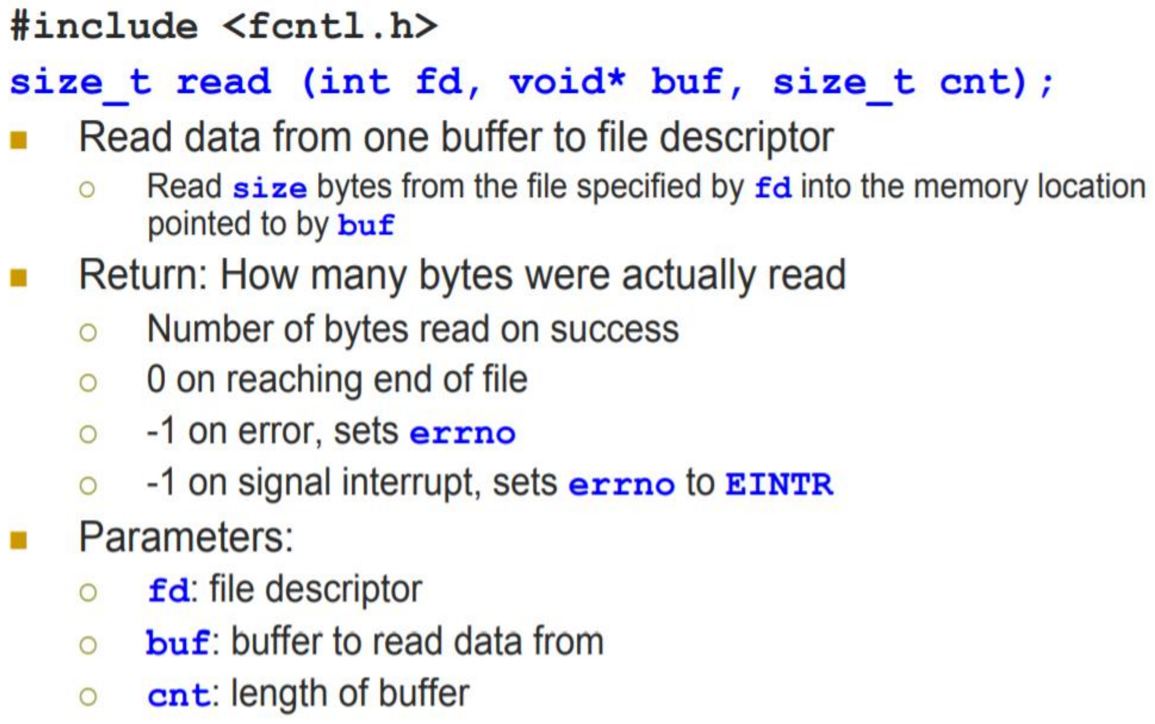
- Close a file
  - Tells the operating system you are done with a file descriptor
- Return:
  - 0 on success
  - -1 on error, sets `errno`
- Parameters:
  - `fd`: file descriptor



```
#include <fcntl.h>
main(){
    int fd1;

    if(( fd1 = open("foo.txt", O_RDONLY)) < 0){
        perror("c1");
        exit(1);
    }
    if (close(fd1) < 0) {
        perror("c1");
        exit(1);
    }
    printf("closed the fd.\n");
```

3. read() System Call

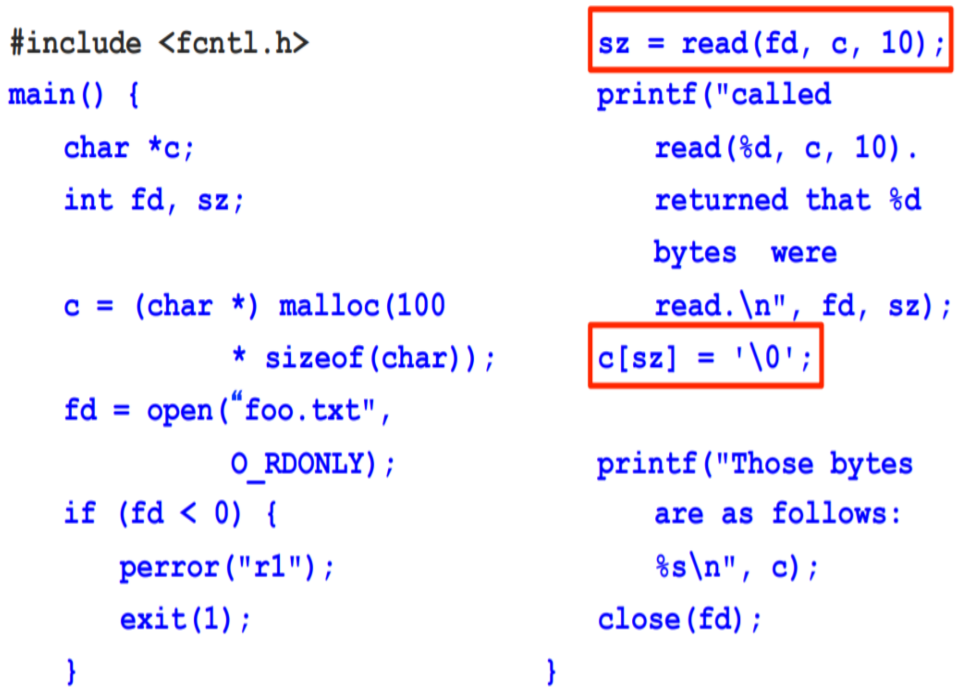


```
#include <fcntl.h>
size_t read (int fd, void* buf, size_t cnt);
```

- Read data from one buffer to file descriptor
  - Read **size** bytes from the file specified by **fd** into the memory location pointed to by **buf**
- Return: How many bytes were actually read
  - Number of bytes read on success
  - 0 on reaching end of file
  - -1 on error, sets **errno**
  - -1 on signal interrupt, sets **errno** to **EINTR**
- Parameters:
  - **fd**: file descriptor
  - **buf**: buffer to read data from
  - **cnt**: length of buffer

Things to be careful about

- buf needs to point to a valid memory location with length not smaller than the specified size .
- fd should be a valid file descriptor returned from open() to perform read operation.
- cnt is the requested number of bytes read, while the return value is the actual number of bytes read.

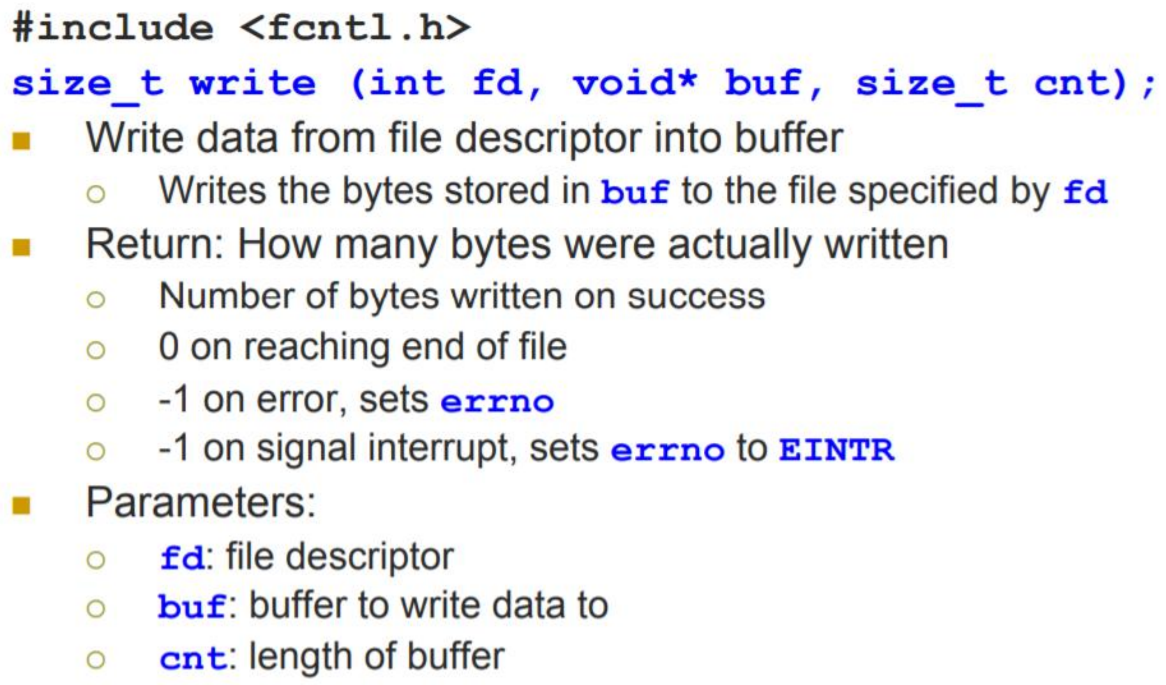


```
#include <fcntl.h>
main() {
   char *c;
   int fd, sz;

   c = (char *) malloc(100
            * sizeof(char));
   fd = open(“foo.txt",
            O_RDONLY);
   if (fd < 0) {
      perror("r1");
      exit(1);
   }
   sz = read(fd, c, 10);
   printf("called
       read(%d, c, 10).
       returned that %d
       bytes  were
       read.\n", fd, sz);
   c[sz] = '\0';

   printf("Those bytes
       are as follows:
       %s\n", c);
   close(fd);
}
```

4. write() system call

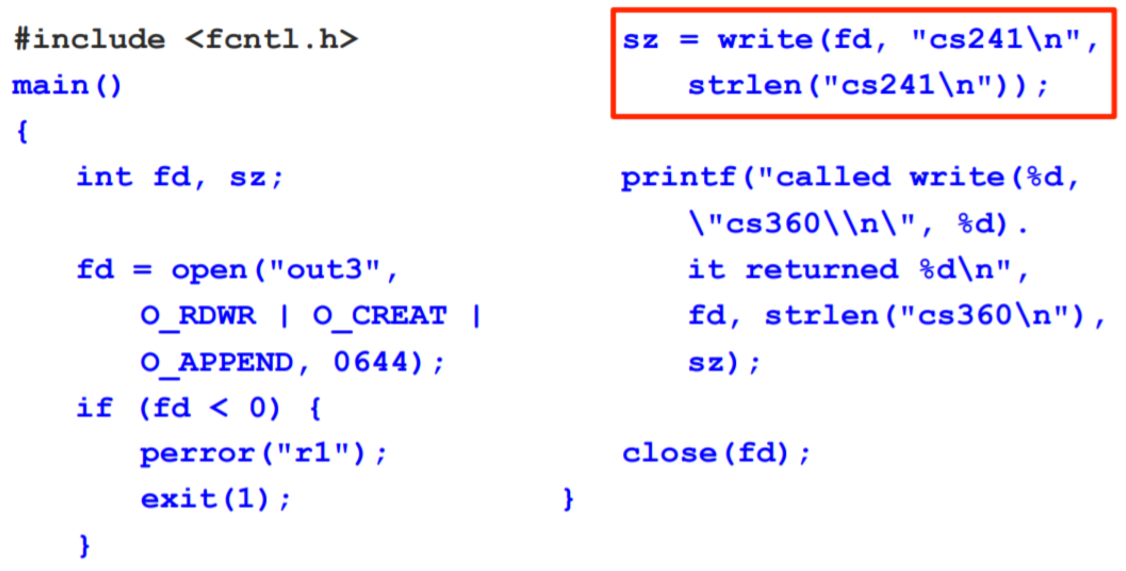


```
#include <fcntl.h>
size_t write (int fd, void* buf, size_t cnt);
```

- Write data from file descriptor into buffer
  - Writes the bytes stored in `buf` to the file specified by `fd`
- Return: How many bytes were actually written
  - Number of bytes written on success
  - 0 on reaching end of file
  - -1 on error, sets `errno`
  - -1 on signal interrupt, sets `errno` to `EINTR`
- Parameters:
  - `fd`: file descriptor
  - `buf`: buffer to write data to
  - `cnt`: length of buffer

Things to be careful about

- The file needs to be opened for write operations.
- buf needs to be at least as long as specified by cnt.
- cnt is the requested number of bytes to write, while the return value is the actual number of bytes written.

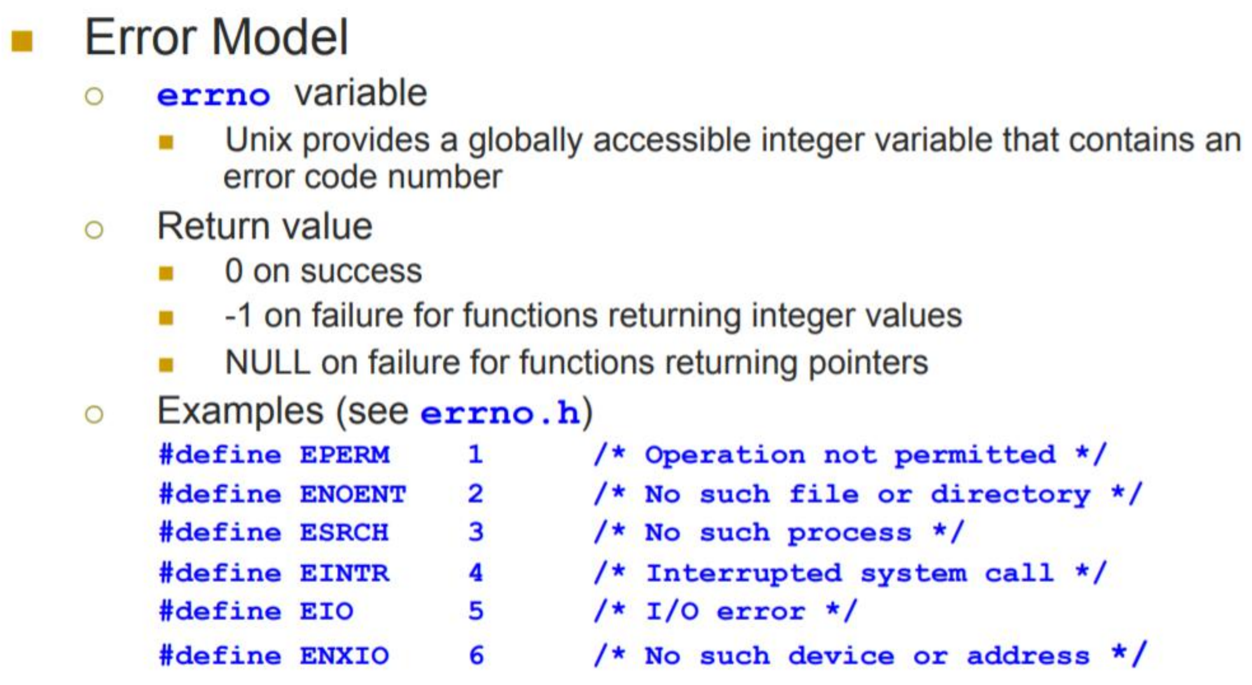


```
#include <fcntl.h>
main()
{
    int fd, sz;

    fd = open("out3",
        O_RDWR | O_CREAT |
        O_APPEND, 0644);
    if (fd < 0) {
        perror("r1");
        exit(1);
    }
    sz = write(fd, "cs241\n",
        strlen("cs241\n"));

    printf("called write(%d,
        \"cs360\\n\", %d).
        it returned %d\n",
        fd, strlen("cs360\n"),
        sz);

    close(fd);
}
```

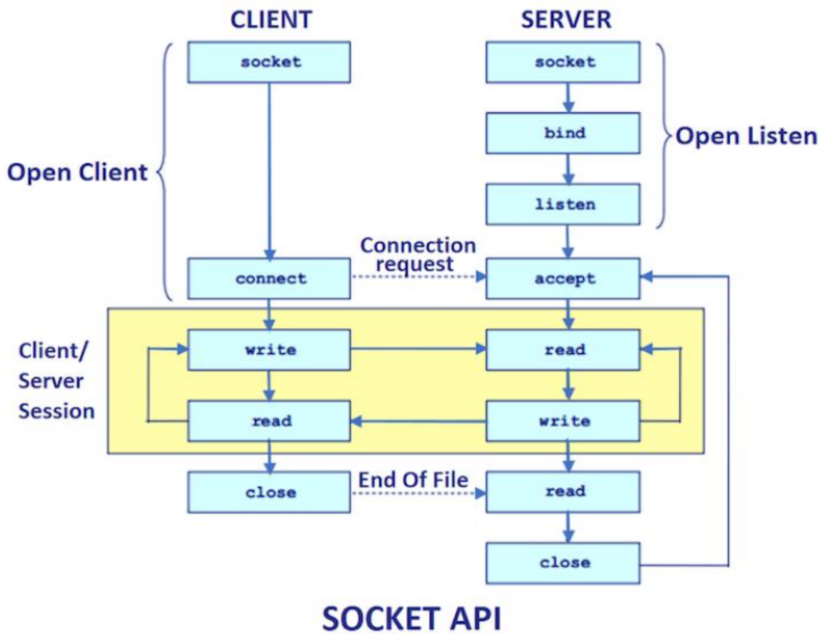


- Error Model
  - `errno` variable
    - Unix provides a globally accessible integer variable that contains an error code number
  - Return value
    - 0 on success
    - -1 on failure for functions returning integer values
    - NULL on failure for functions returning pointers
  - Examples (see `errno.h`)

```
#define EPERM     1     /* Operation not permitted */
#define ENOENT    2     /* No such file or directory */
#define ESRCH     3     /* No such process */
#define EINTR     4     /* Interrupted system call */
#define EIO       5     /* I/O error */
#define ENXIO     6     /* No such device or address */
```

Sockets for Interprocess Communication:

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 inter process communication channel. The two processes each establish their own socket.



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.

Step 1: Creating a socket:

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

Creating a socket is in some ways similar to opening a file. This function creates a file descriptor and returns it from the function call. You later use this file descriptor for reading, writing and using with other socket functions

Parameters:
`family`: AF_INET or PF_INET (These are the IP4 family)
`type`: SOCK_STREAM (for TCP) or SOCK_DGRAM (for UDP)

Step 2: Binding an address and port number

```
int bind(int socket_file_descriptor, const struct sockaddr * LocalAddress, socklen_t
AddressLength);
```

We need to associate an IP address and port number to our application. A client that wants to connect to our server needs both of these details in order to connect to our server. Notice the difference between this function and the connect() function of the client. The connect function specifies a remote address that the client wants to connect to, while here, the server is specifying to the bind function a local IP address of one of its Network Interfaces and a local port number.

The parameter *socket_file_descriptor* is the socket file descriptor returned by a call to *socket()* function. The return value of *bind()* is 0 for success and –1 for failure.

Step 3: Listen for incoming connections

Binding is like waiting by a specific phone in your house, and Listening is waiting for it to ring.

```
int listen(int socket_file_descriptor, int backlog);
```

The parameter *socket_file_descriptor* is the socket file descriptor returned by a call to *socket()* function. The return value of *listen()* is 0 for success and –1 for failure.

It is important in determining how many connections the server will connect with. Typical values for backlog are 5 – 10.

Step 4: Accepting a connection.

```
int accept (int socket_file_descriptor, struct sockaddr * ClientAddress, socklen_t
*addrlen);
```

accept() returns a new socket file descriptor for the purpose of reading and writing to the client. The original file descriptor is used usually used for listening for new incoming connections.

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.

Step 1: Create a socket : Same as in the server.

Step 2: Binding a socket: This is unnecessary for a client, what bind does is (and will be discussed in detail in the server section) is associate a port number to the application. If you skip this step with a TCP client, a temporary port number is automatically assigned, so it is just better to skip this step with the client.

Step 3: Connecting to a Server:

```
 int connect(int socket_file_descriptor, const struct sockaddr *ServerAddress,
socklen_t AddressLength);
```

Once you have created a socket and have filled in the address structure of the server you want to connect to, the next thing to do is to connect to that server. This is done with the connect function listed above.

Connect performs the three-way handshake with the server and returns when the connection is established or an error occurs.

Once the connection is established you can begin reading and writing to the socket.

Step 4: read/write: These are the same functions you use with files but you can use them with sockets as well.

Writing to a socket:

```
int write(int file_descriptor, const void * buf, size_t message_length);
```

Reading from a socket:

```
int read(int file_descriptor, char *buffer, size_t buffer_length);
```

The value returned is the number of bytes read which may not be buffer_length! It returns –1 for failure. As with write(), read() only transfers data from a buffer in the kernel to your application , you are not directly reading the byte stream from the remote host, but rather TCP is in control and buffers the data for your application.

Shutting down sockets:

After you are finished reading and writing to your socket you most call the close system call on the socket file descriptor just as you do on a normal file descriptor otherwise you waste system resources.

The close() function: `int close(int filedescriptor);`
