

A person wearing a dark jacket is pulling a thick white rope with blue stripes through a pulley system on a boat. The rope is coiled around the pulley. The background shows the blue ocean and a clear sky with some clouds. The text "File Descriptors & System Calls" is overlaid in green.

File Descriptors & System Calls

Dr. Vimal Baghel
Assistant Professor
SCSET, BU

Contents

- File Descriptor
- System Calls
- System call Vs Function Call
- Sequence of making system call
- Q & A

File Descriptors

- A *file descriptor* is the Unix abstraction for an open I/O stream:
 - a file, a network connection, a pipe, a terminal, etc.
- **File descriptor table:** an array of kernel objects
- The file descriptors that applications manipulate are *indexes* into this table.

File Descriptors

- Logically, a file descriptor comprises
 - a **file reference** (/home/vimal/data.txt) and
 - a **file position** (an offset into the file)
- There can be many file descriptors simultaneously open for the same file reference, each with a different position.

File Descriptors

- For disk files, the position can be explicitly changed:
 - a process can rewind and re-read part of a file or skip around.
- These files are called *seekable*.
- However, not all types of file descriptor are seekable.

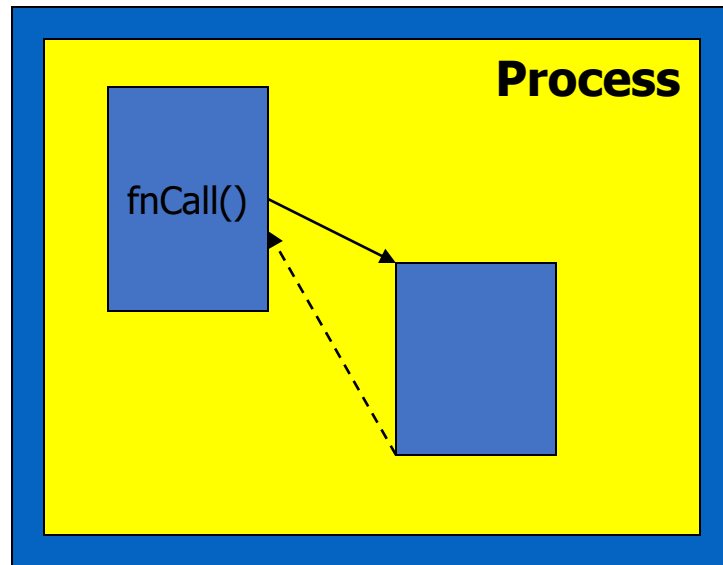
System Calls

- **System Calls**
 - A request to the OS 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?

System Calls versus Function Calls

Function Call

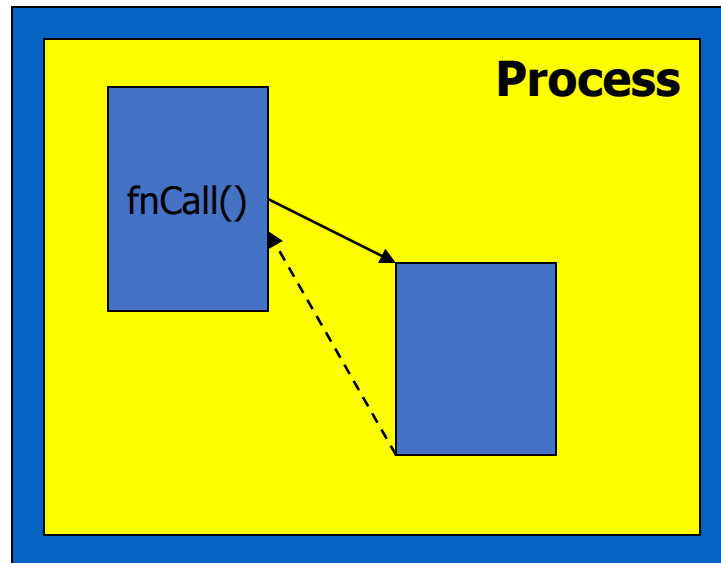


Caller and callee are in the same Process

- Same user
- Same "domain of trust"

System Calls versus Function Calls

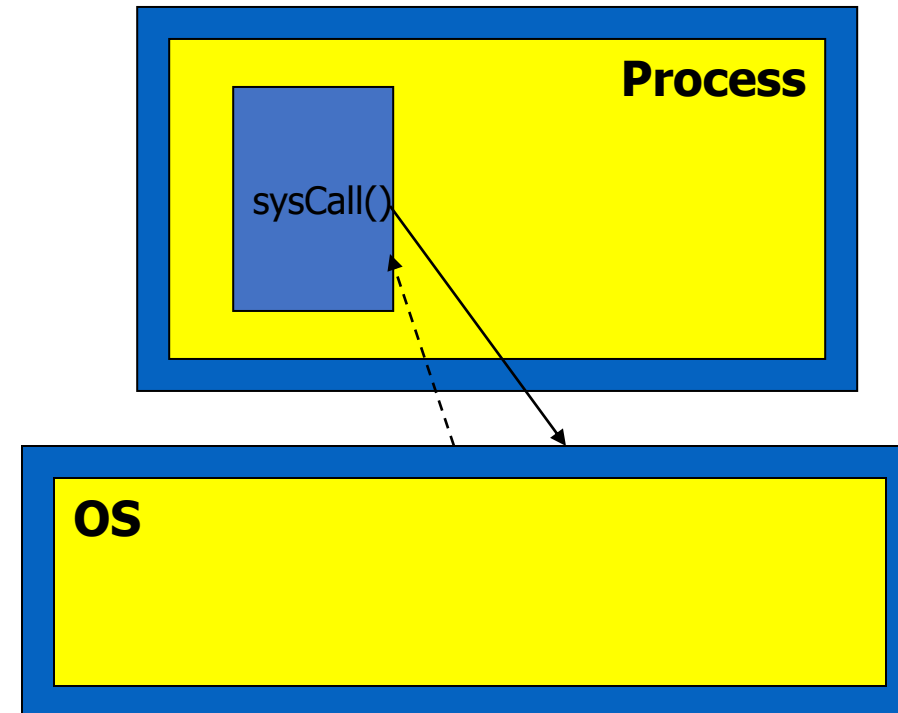
Function Call



Caller and callee are in the same Process

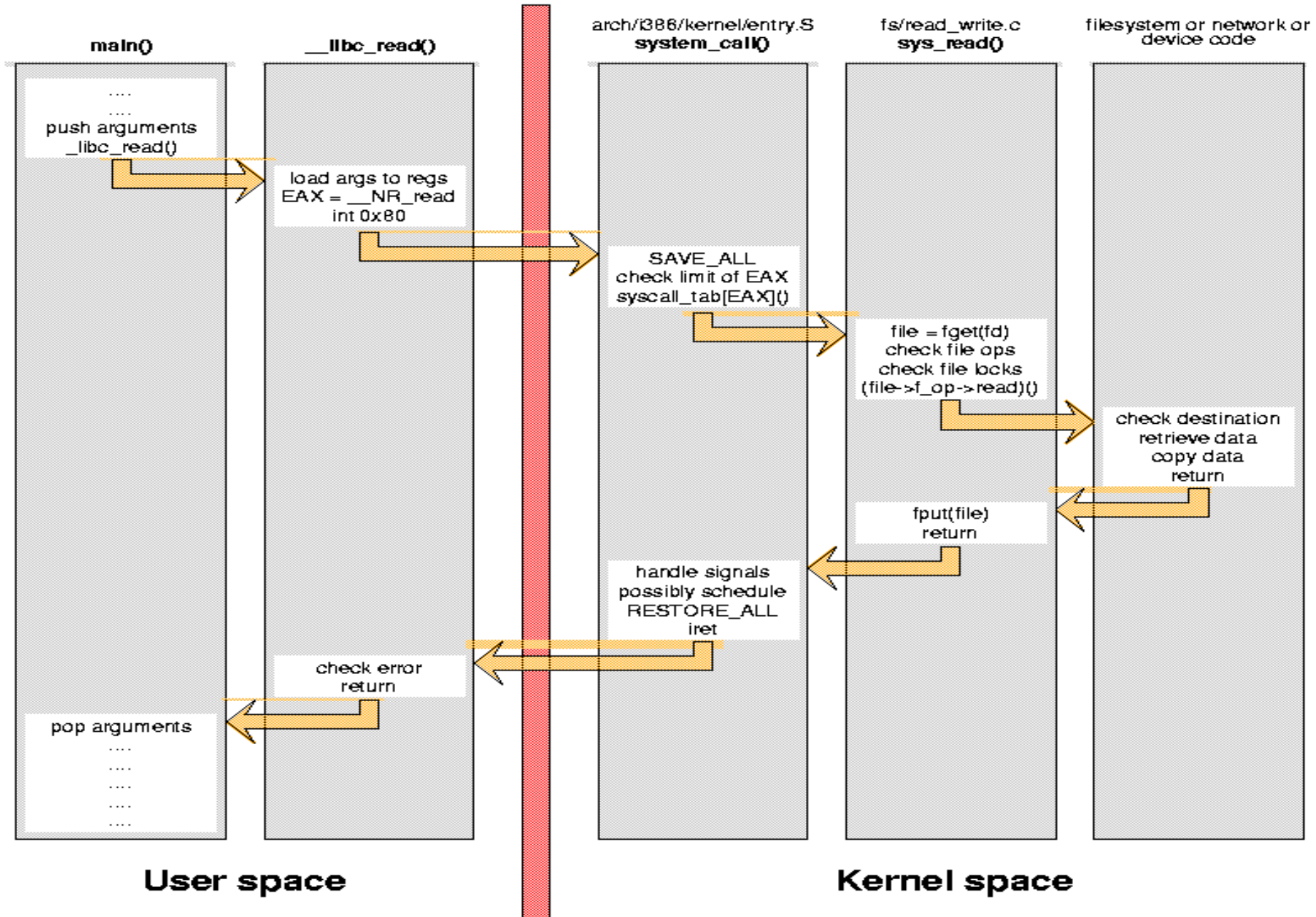
- Same user
- Same "domain of trust"

System Call



- OS is trusted; user is not.
- OS has super-privileges; user does not
- Must take measures to prevent abuse

Sequence for Making a System Call (Example: read call)



Steps for Making a System Call (Example: read call)

- A system call is implemented by a “*software interrupt*” that transfers control to kernel code; in Linux/i386 this is “interrupt 0x80”.
- The specific system call being invoked is stored in the EAX register, and its arguments are held in the other CPU registers.
- *After the switch to kernel mode, the CPU must save all of its registers and dispatch execution to the proper kernel function, after checking whether EAX is out of range.*
- The read finally performs the data transfer and all the previous steps are unwound up to the calling user function.

Cost for Making a System Call (Example:read)

- Each arrow in the figure represents a jump in CPU instruction flow, and each jump may require flushing the prefetch queue and possibly a *“cache miss” event*.
- Transitions between user and kernel space are especially important, as they are the most expensive in processing time and prefetch behavior.

File Locking & Record Locking

- **File locking** is the capability to prevent other processes from reading/writing any part of a file
- **Record locking** is the capability to prevent other processes from reading/writing particular records.
- `fcntl(fd, cmd, arg);` system call is used to implement file/record locking
 - `fd`= file descriptor, `cmd`= type of lock (R/W), `arg`=lock type/byte offset,
- Kernel releases the locks as soon as the process closes the file

System calls Examples

- System calls to access the existing files:
 - `open()`, `read()`, `write()`, `lseek()`, and `close()`
 - `getuid()` //get the user ID
 - `fork()` //create a child process
 - `exec()` //executing a program => `execve (2)` in Linux
 - System calls to create new files:
 - `Create()`, `mknod()`
 - System calls to modify the inode or maneuver through the filesystem:
 - `chdir()`, `chroot()`, `chown()`, `chmod()`, `stat()`, `fstat()`
 - Advanced System calls :
 - `pipe()`, `dup()`, `mount()`, `umount()`, `link()`, `unlink()`
- Don't confuse system calls with *libc* calls
 - Differences?
 - Is `printf()` a system call?
 - Is `rand()` a system call?

System calls vs. *libc*

Each I/O system call has corresponding procedure calls from the standard I/O library.

System calls	Library calls
open	fopen
close	fclose
read	fread, getchar, scanf, fscanf, getc, fgetc, gets, fgets
write	fwrite, putchar, printf, fprintf putc, fputc, puts, fputs
lseek	fseek

Use man -s 2

Use man -s 3

Define a new system call

- Create a directory hello in the kernel source directory:
 - **mkdir hello**
- Change into this directory
 - **cd hello**
- Create a “hello.c” file in this folder and add the definition of the system call to it as given below using any text editor.

```
#include <linux/kernel.h>  asmlinkage long
sys_hello(void)
{
    printk(KERN_INFO "Hello world\n"); return 0;
}
```


Create Makefile

- Create a “Makefile” in the hello folder and add the given line to it.
 - **gedit Makefile**
- add the following line to it:-
 - **objy := hello.o**
- This is to ensure that the hello.c file is compiled and included in the kernel source code.

Add the hello directory to the kernel's Makefile

- change back into the linux-3.16 folder and open Makefile.
 - gedit Makefile
 - goto line number 842 which says :- "core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ "
 - change this to "core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ hello/"
- This is to tell the compiler that the source files of our new system call (sys_hello()) are present in the hello directory.

Add system call to the table

- If your system is a 64 bit system you will need to alter the **syscall_64.tbl** file else **syscall_32.tbl**.
 - `cd arch/x86/syscalls`
 - `gedit syscall_32.tbl`
- Add the following line in the end of the file :-

358 **i386** **hello** **sys_hello**

358 – It is the number of the system call .

It should be one plus the number of the last system call. (it was 358 in my system).
This has to be noted down to make the system call in the userspace program.

Add system call to header file

- **cd include/linux/**
- **gedit syscalls.h**
 - ✓ add the following line to the end of the file just before the #endif statement at the very bottom.
 - **asmlinkage long sys_hello(void);**
- This defines the prototype of the function of our system call.
- “**asmlinkage**” is a key word used to indicate that all parameters of the function would be available on the stack.

Now compile the linux source code according to the standard procedure.

Test the system call

- Create a “userspace.c” program in your home folder and type in the following code :-

```
#include <stdio.h>
#include <linux/kernel.h> #include
<sys/syscall.h> #include <unistd.h>
int main()
{
    long int r = syscall(358);
    printf("System call sys_hello returned %ld\n", r); return 0;
}
```

Test the system call

- Now compile this program using the following command.
 - **gcc userspace.c**
- If all goes well you will not have any errors else, rectify the errors. Now run the program using the following command.
 - **./a.out**
- You will see the following line getting printed in the terminal if all the steps were followed correctly.
 - **“System call sys_hello returned 0“.**
- Now to check the message of the kernel you can run the following command.
 - **dmesg**

Why does the OS control I/O?

Why does the OS control I/O?

- **Safety:** The computer must ensure that if my program has a bug in it, then it doesn't crash or mess up
 - the system,
 - other programs that may run at the same time or later.
- **Fairness:** Make sure other programs have a fair use of device

System Calls for I/O

- There are 5 basic system calls that Unix provides for file I/O
 - `int open(char *path, int flags [, int mode]);` (check `man -s 2 open`)
 - `int close(int fd);`
 - `int read(int fd, char *buf, int size);`
 - `int write(int fd, char *buf, int size);`
 - `off_t lseek(int fd, off_t offset, int whence);`
- Some library calls themselves make a system call
 - (e.g. **`fopen()`** calls **`open()`**)

Open

- **int open(char *path, int flags [, int mode])** makes a request to the OS to use a file.
 - The '**path**' argument specifies the file you would like to use
 - The '**flags**' and '**mode**' arguments specify how you would like to use it.
 - If the OS approves your request, it will return a *FD*.
 - This is a non-negative integer. Any future accesses to this file needs to provide this FD
 - If it returns -1, then you have been denied access; check the value of global variable "**errno**" to determine why (or use **perror()** to print corresponding error message).

perror() & errno

- When a system call fails, it usually returns -1 and sets the variable errno to a value describing what went wrong.
- (These values can be found in <errno.h>.) Many library functions do likewise.
- The function perror() serves to translate this error code into human-readable form.

Standard Input, Output and Error

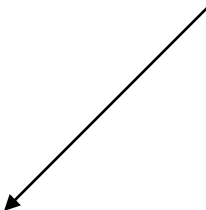
- Now, every process in Unix starts out with three FDs predefined:
 - File descriptor 0 is standard input.
 - File descriptor 1 is standard output.
 - File descriptor 2 is standard error.
- We can read from standard input, using **read(0, ...)**, and write to standard output using **write(1, ...)** or using two **library** calls
 - printf
 - scanf

Example 1

```
#include <fcntl.h>
#include <errno.h>
```

```
main(int argc, char** argv) {
    int fd;
    fd = open("foo.txt", O_RDONLY);
    printf("%d\n", fd);
    if (fd==-1) {
        fprintf (stderr, "Error Number %d\n", errno);
        perror("Program example1_open");
    }
}
```

Other flag options are:
O_WRONLY or O_RDWR
(see man pages for more flags!)



Close()

- **int close(int fd)**

Tells the OS you are done with a FD.

```
#include <fcntl.h>
main(){
    int fd1, fd2;

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

Why do we need to close a file?

After close, can you still use the file descriptor?

read(...)

- `int read(int fd, char *buf, int size)` tells the OS:
 - To read "**size**" bytes from the file specified by "**fd**" into the memory location pointed to by "**buf**".
 - It returns how many bytes were actually read (**why?**)
 - 0 : at end of the file
 - < size : fewer bytes are read to the buffer (**why?**)
 - == size : read the specified # of bytes
- Things to be careful about
 - buf must point to valid memory not smaller than the specified size
 - Otherwise, what could happen?
 - fd should be a valid file descriptor returned from open() to perform read operation

read(...)

- It is not an error and if the number of read bytes is smaller than the number of bytes requested
 - this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe)
 - it can also happen because read() was interrupted by a signal.

Example 2

```
#include <fcntl.h>
main(int argc, char** argv) {
    char *c;
    int fd, sz;
    c = (char *) malloc(100 * sizeof(char));

    fd = open("foo.txt", O_RDONLY);
    if (fd < 0) { perror("foo.txt"); exit(1); }

    sz = read(fd, c, 10);
    printf("called read(%d, c, 10), which read %d bytes.\n", fd, sz);
    c[sz] = '\0';
    printf("Those bytes are as follows: %s\n", c);

    close(fd);
}
```

write(...)

- **int write(int fd, char *buf, int size)** writes the bytes stored in **buf** to the file specified by **fd**
 - It returns the number of bytes written, which is usually “**size**” unless there is an error
- Things to be careful about
 - buf must be at least as long as “size”
 - The file must be open for write operations

Example 3

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

If the file does not exist,
it will be created

the file offset is set to the end of the
file before each write

```
    fd = open("out3", O_RDWR | O_CREAT | O_APPEND, 0644);
    if (fd < 0) { perror("out3"); exit(1); }
```

```
    sz = write(fd, "cs241\n", strlen("cs241\n"));
```

```
    printf("called write(%d, \"cs241\\n\", %d), which returned %d\n",      fd,
           strlen("cs241\n"), sz);
```

```
    close(fd);
}
```

lseek()

- All opened files have a "file pointer" associated with them to record the current position for the next file operation
 - When file is opened, file pointer points to the beginning of the file
 - After reading/writing *m* bytes, the file pointer moves *m* bytes forward
- **off_t lseek(int fd, off_t offset, int whence)** moves the file pointer explicitly
 - The 'whence' argument specifies how the seek is to be done
 - from the beginning of the file
 - from the current value of the pointer, or
 - from the end of the file
 - The return value is the offset of the pointer after the lseek
- How would you know to include sys/types.h and unistd.h?
 - Read "man -s 2 lseek"

Lseek() example

```
c = (char *) malloc(100 * sizeof(char));
fd = open("foo.txt", O_RDONLY);
if (fd < 0) { perror("foo.txt"); exit(1); }

sz = read(fd, c, 10);
printf("We have opened foo.txt, and called read(%d, c, 10).\n", fd);
c[sz] = '\\0';
printf("Those bytes are as follows: %s\n", c);

i = lseek(fd, 0, SEEK_CUR);
printf("lseek(%d, 0, SEEK_CUR) returns the current offset = %d\n\n", fd, i);

printf("now, we seek to the beginning of the file and call read(%d, c, 10)\n", fd);
lseek(fd, 0, SEEK_SET);
sz = read(fd, c, 10);
c[sz] = '\\0';
printf("The read returns the following bytes: %s\n", c);
...:
```



Thanks

Q & A