



CS241 Systems Programming

System Calls and I/O

Lawrence Angrave
Lecture 4

CS241 Administrative

This week

SMP0 due on Monday 9am morning

Discussion section

Anyone cannot find a discussion section that fit his/her schedule? Anyone needs to change discussion section?

Quizzes

Self-assessment quiz on campus

Pop quiz in lecture

Lecture & Book content

Typically selected from self-assessment quiz

Wednesday or Friday

MP quiz in lecture

Test programming knowledge associated with current lecture topics & MP

Monday after MP submission

This lecture

Goals:

- Get you familiar with necessary basic system & I/O calls to do programming

- Get basic things out of your way before moving to more complex things

- Enable you to work on your MP and any future MPs

Things covered in this lecture

- Basic file system

- I/O calls

- Signals

Note: we will come back later to discuss the above things at the concept level

More C review URLs

C Pointers Guide

[C Tutorial - Lesson 8: An Introduction To Pointers - Dereferencing](#)
[C/C++ Tutorial: An Introduction To Pointers](#)
[A Tutorial on Pointers and Arrays in C](#)

C Guide

[The GNU C Programming Tutorial](#)
[Programming in C \(new, by Marshall\)](#)
[Programming in C: A Tutorial \(old, by Kernighan\)](#)
["The C programming language" book website](#)

File system from a user's point of view

A file system: *A hierarchical arrangement of directories.*

In Unix, the root file system starts with "/"

File system from a user's point of view

A file system: *A hierarchical arrangement of directories.*

In Unix, the root file system starts with "/"

To see the filesystems on your machine, type **"df"**.

```
yyzhou@carmen|~|[1]% df
Filesystem      1k-blocks    Used Available Use% Mounted on
/dev/sda1        5162796    3967004   933536  81% /
/dev/sda5        63322504   31773716  28332176  53% /mounts/carmen/disks/0
/dev/sdb1        70557052   44202760  22770196  67% /mounts/carmen/disks/1
/dev/sdc1        70557052   45750108  21222848  69% /mounts/carmen/disks/2
/dev/sdd1        70557052   25866092  41106864  39% /mounts/carmen/disks/3
none            1034536     305740    728796  30% /dev/shm
/dev/sda3        1035692     489184    493896  50% /var
```

File name & Path name

Filename: *The name of a file as it appears in a directory.*

Pathname: *A sequence of zero or more filenames, separated by slashes.*

List all filenames in the current directory: *ls -a*

Change directory: *cd dir*

Change to home directory: *cd*

Copy directory: *cp -r dir1 dir2*

Create a directory: *mkdir dir*

Print the current directory path: *pwd*

System Calls

System Calls

A request to the operating system to perform some activity

Looks like a procedure call, but it's different

System calls are expensive----Why?

The system needs to perform many things before executing a system call

The computer (hardware) save its state (Why?)

Let the operating system take control of the CPU (why?)

The OS examines the parameters you passed (why?)

Have the operating system perform some function

Have the operating system save its state

Have the operating system give control of the CPU back to you

Examples of System Calls

Can anyone give me an example?

`getuid()` //get the user ID

`execv()` //executing a program

`times()` // get the execution time

Don't mix system calls with standard library calls

Differences?

Is *printf()* a system call?

Is `rand()` a system call?

System Calls for I/O

There are 5 basic system calls that Unix provides for file I/O
(check `man -s 2 open`)

```
int open(char *path, int flags [ , int mode ] );
```

```
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);
```

They look like regular procedure calls. They are different

A system call makes a request to the operating system.

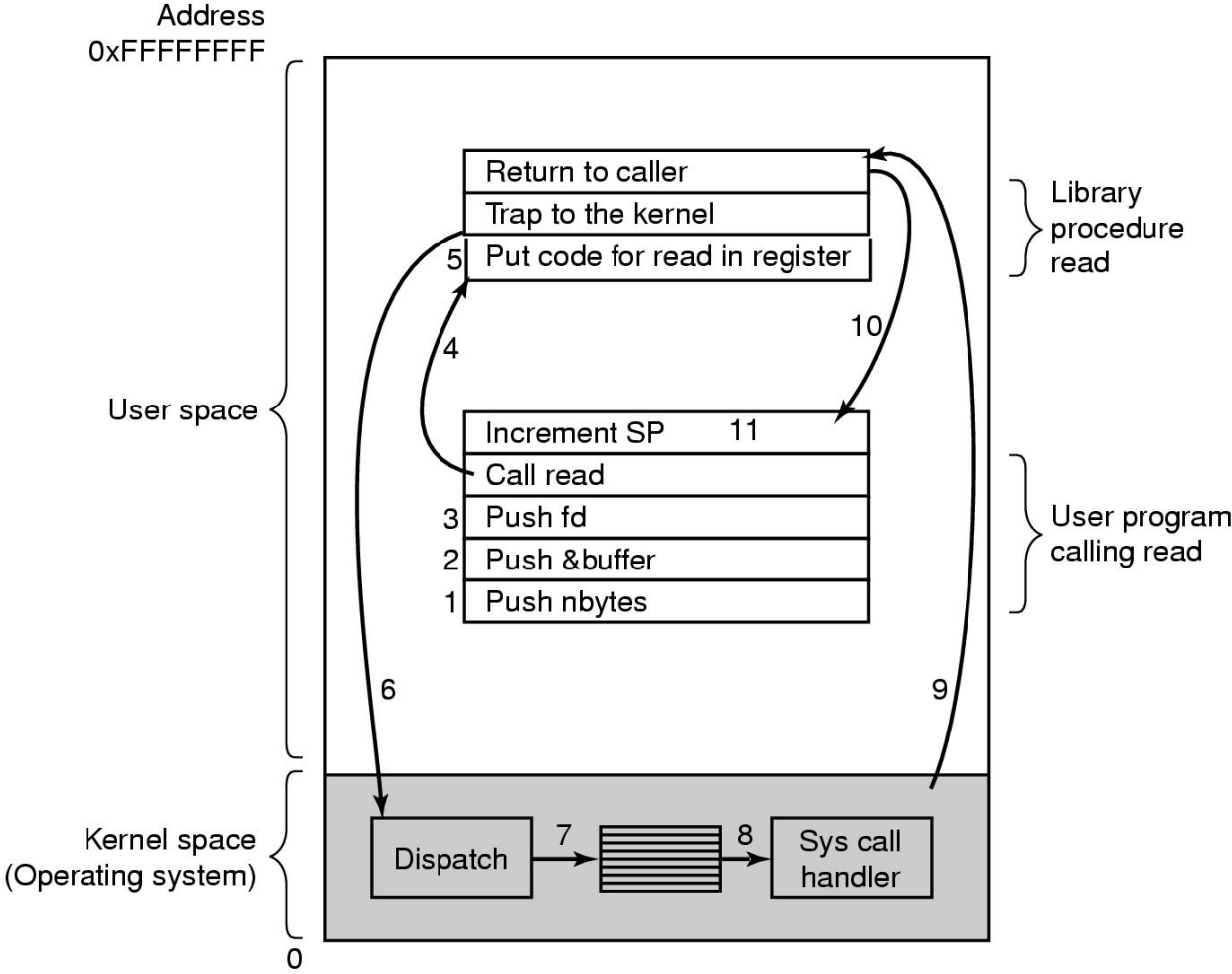
A procedure call just jumps to a procedure defined elsewhere in your program.

Some library calls may themselves make a system call

E.g. `fopen()` calls `open()`



Steps for Making a System Call (review)



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 people's programs that may be running at the same time or later.

Fairness

Make sure other programs have a fair use of device

For experienced students

In what types of systems, the OS does NOT control I/O?

Open

int open(char *path, int flags [, int mode]) makes a request to the operating system to use a file.

The '**path**' argument specifies what file you would like to use

The '**flags**' and '**mode**' arguments specify how you would like to use it.

If the operating system approves your request, it will return a *file descriptor* to you. This is a non-negative integer. Any future accesses to this file needs to provide this file descriptor

If it returns -1, then you have been denied access, and check the value of the variable "**errno**" to determine why (use **perror()**).

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

```
main()
```

```
{
```

```
    int fd;
```

```
    fd = open("foo.txt", O_RDONLY);
```

```
    printf("%d\n", fd);
```

```
}
```

What could be the output?

What did I forget?

Why specify mode?

Close

int close(int fd) tells the operating system that you are done with a file descriptor.

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

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

After close, can you still use the file descriptor?

Why do we need to close a file?

read(...)

*int read(int fd, char *buf, int size)* tells the operating system

To read "**size**" bytes from the file specified by "**fd**" into the memory location pointed to by "**buf**".

It returns many bytes were actually read (**why?**)

0 ---- end of the file

< size ---- fewer bytes are read to the buffer (**why?**)

== size ---- read the specified # of bytes

NEVER > size (**why?**)

Things to be careful

buf needs to point to a valid memory location with length not smaller than the specified size

Otherwise, what could happen?

fd should be a valid file descriptor returned from open to perform read operation

Otherwise, what could happen?

Example

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

```
c = (char *) calloc(100, sizeof(char));
```

Any problem?

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

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 actually written, which is almost invariably "**size**"
Returns a value smaller than size--- an indication for error

Things to be careful

buf needs to be at least as long as specified by "size"
The file needs to be opened for write operations

Example

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


    fd = open("out3", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd < 0) { perror("r1"); exit(1); }

    sz = write(fd, "cs360\n", strlen("cs360\n"));

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

    close(fd);
}
```

Why?



lseek

All open files have a "file pointer" associated with them to record the current position for the next file operation

When the file is opened, the file pointer points to the beginning of the file

After reading/write 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' variable of lseek 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 will you know to include sys/types.h andunistd.h?

type "man -s 2 lseek"

lseek example

```
c = (char *) calloc(100, sizeof(char));  
fd = open("foo.txt", O_RDONLY);  
if (fd < 0) { perror("r1"); exit(1); }
```

```
sz = read(fd, c, 10);  
printf("We have opened in1, 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 that the current offset is %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);
```

...

Standard Input, Output and Error

Now, every process in Unix starts out with three file descriptors predefined:

File descriptor 0 is standard input.

File descriptor 1 is standard output.

File descriptor 2 is standard error.

You can read from standard input, using **read(0, ...)**, and write to standard output using **write(1, ...)** or using two **library** calls

printf

scanf

Why did UNIX design this way?

I/O Library Calls

Each system call has analogous procedure calls from the standard I/O library:

System Call

open

close

read/write

lseek

Standard I/O call

fopen

fclose

getchar/putchar

getc/putc

fgetc/fputc

fread/fwrite

gets/puts

fgets/fputs

scanf/printf

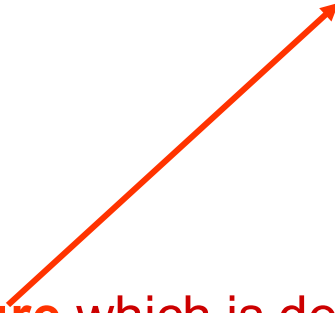
fscanf/fprintf

fseek

Files in C

- Location of the file on the disk
- Type of the file
- Allowed operations
- Present position in the file

```
typedef struct
{
    char*   _ptr;
    int     _cnt;
    char*   _base;
    int     _flag;
    int     _file;
    int     _charbuf;
    int     _bufsiz;
    char*   _tmpfname;
} FILE;
```



This information is kept in a **structure** which is defined as type **FILE** in the header-file `stdio.h`.

The details of the `FILE` structure are never regarded. You only need to provide a **pointer** to it:

```
FILE *ifp; /* pointer to be used for input file */
FILE *ofp; /* pointer to be used for output file */
```

The pointers are assigned when opening the file and then used for reading from it or writing to it, and finally closing it.

fopen() and fclose()

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

```
int main(void)  
{
```

```
    int a, sum = 0;  
    FILE *ifp, *ofp; /* input file pointer,  
                    output file pointer */
```

```
    ifp = fopen("my_file", "r"); /* open for reading */  
    ofp = fopen("outfile", "w"); /* open for writing */  
    .....
```

```
    fclose(ifp);  
    fclose(ofp);
```

Pointers to structure



The modes for opening files



File modes for fopen()

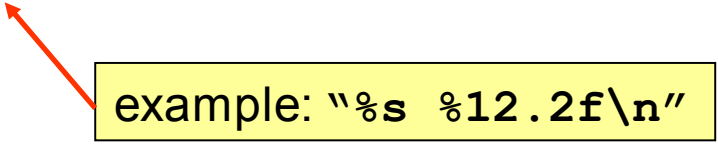
MODE	used for	if file doesn't exist	if file exists
"a"	Appending	New file created	File appended to
"a+"	Reading and appending	New file created	File appended to
"r"	Read only	NULL returned	
"r+"	Reading and writing	NULL returned	
"w"	Write only	New file created	File overwritten
"w+"	Reading and writing	New file created	File overwritten

fprintf() and fscanf()

To write to files: function `fprintf()` analogous to `printf()`
To read from files: function `fscanf()` analogous to `scanf()`

In the file i/o functions, a file pointer has to be specified.

```
fprintf(ofp, control_string, other_args);  
fscanf (ifp, control_string, other_args);
```



example: "%s %12.2f\n"

getc() and putc()

To write character to file:

`putc(c, ofp)` analogous to `putchar(c)`

To read character from file:

`getc(ifp)` analogous to `getchar()`

```
int c;  
c = getc(ifp);
```

```
putc(c, ofp);
```

EOF (end-of-file) character is defined as -1 and therefore **not** in the usual ASCII character set (0-255),



Use **int** instead of **char** if EOF is to be read

`getchar()` and `putchar(c)` are defined as macros:

```
#define getchar() getc(stdin)  
#define putchar(c) putc(c, stdout)
```

Moving around in files

`rewind(file_ptr) ;` resets the position indicator to start of file


`ftell(file_ptr) ;` returns the current value of file position
indicator: i.e. no. of bytes from start of file

`fseek(file_ptr, offset, place) ;`

moves the file position indicator to *offset* bytes from *place*.
place can be `SEEK_SET`, `SEEK_CUR` or `SEEK_END`
(actually 0, 1 or 2) for beginning, current position or end-of-file
espectively. Offset can be positive or negative.

Example: file access

```
/* output a file backwards */
#include <stdio.h>
#define MAXSTRING 100
int main(void)
{
    char fname[MAXSTRING];
    int c;
    FILE *ifp;
    fprintf(stdout, "\nInput a filename: ");
    scanf("%s", fname);
    ifp = fopen(fname, "r");          /* open for reading */
    fseek(ifp, 0, SEEK_END);         /* move to end of file */
    fseek(ifp, -1, SEEK_CUR);        /* back up one char */
    while (ftell(ifp) > 0) {
        c = getc(ifp);              /* move ahead one char */
        putchar(c);
        fseek(ifp, -2, SEEK_CUR);    /* back up 2 chars */
    }
    return 0;
}
```



back up 2 chars, because reading in one char increments the file position indicator by 1.

Signals

Signal: *An interruption of the program*

Signal Handler: *The mechanism by which the program may gracefully deal with signals*

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

```
main()
{
    int i;

    i = 0;
    while (1) {
        i++;
        printf("%d\n", i);
        sleep(5);
    }
}
```

- What does it do?
- If you type **ctrl-z** to this program, what will happen?
 - > Send a **stop** signal to the program
- what about **ctrl-c**?
 - > Send a **kill** signal to the program

Summary

System I/O calls

I/O library calls

Read R&R : pp: 16-48, pp. 812-814

Read S: Chapter 1 pp. 47-50 Chapter 2 pp. 94-102

Next lecture

Compile, link, debug a C program