CS241 Systems Programming
System Calls and I/O

Tarek Abdelzaher
Vikram Adve

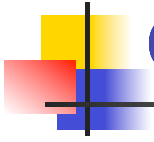
Copyright ©: Nahrstedt, Angrave, Abdelzaher

Copyright ©: Nahrstedt, Angrave, Abdelzaher

This lecture

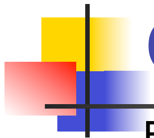
- Goals:
  - Get you familiar with necessary basic system & I/O calls to do programming
- Things covered in this lecture
  - Basic file system calls
  - I/O calls
  - Signals
- Note: we will come back later to discuss the above things at the concept level

Copyright ©: Nahrstedt, Angrave, Abdelzaher



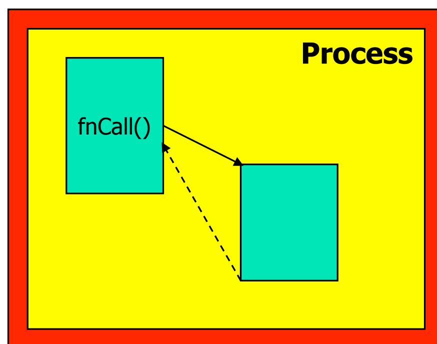
System Calls versus Function Calls?

Copyright ©: Nahrstedt, Angrave, Abdelzaher



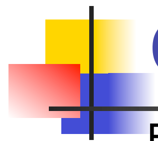
System Calls versus Function Calls

Function Call



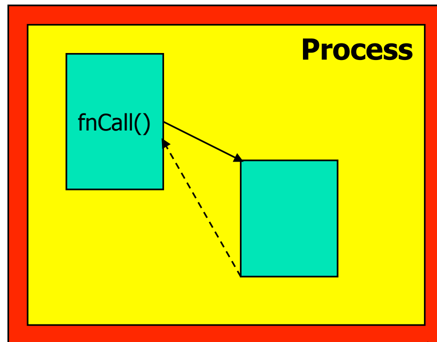
Caller and callee are in the same Process
- Same user
- Same "domain of trust"

Copyright ©: Nahrstedt, Angrave, Abdelzaher



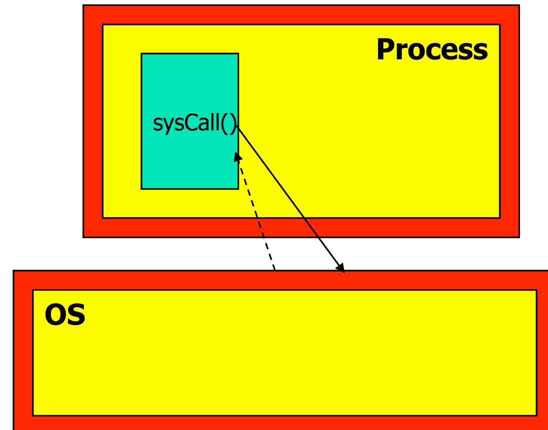
System Calls versus Function Calls

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

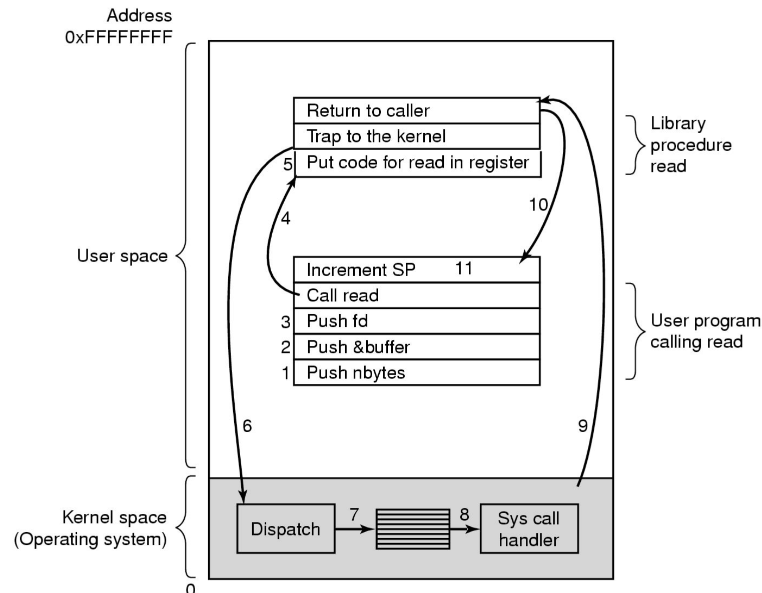
Copyright ©: Nahrstedt, Angrave, Abdelzaher

System Calls

- **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

Copyright ©: Nahrstedt, Angrave, Abdelzaher

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



Copyright ©: Nahrstedt, Angrave, Abdelzaher

Examples of System Calls

- Example:
  - getuid() //get the user ID
  - fork() //create a child process
  - exec() //executing a program
- Don't confuse system calls with *libc* calls
  - Differences?
  - Is printf() a system call?
  - Is rand() a system call?

Copyright ©: Nahrstedt, Angrave, Abdelzaher

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*

Copyright ©: Nahrstedt, Angrave, Abdelzaher

File System and I/O Related System Calls

- **A file system:** *A hierarchical arrangement of directories.*
- In Unix, the root file system starts with "/"

Copyright ©: Nahrstedt, Angrave, Abdelzaher

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

Copyright ©: Nahrstedt, Angrave, Abdelzaher

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);
- Remember: these are different from regular procedure calls
- Some library calls themselves make a system call
  - (e.g. **fopen()** calls **open()**)

Copyright ©: Nahrstedt, Angrave, Abdelzaher

Open

- **int open(char *path, int flags [, int mode])** makes a request to the operating system 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 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; check the value of global variable "**errno**" to determine why (or use **perror()** to print corresponding error message).

Copyright ©: Nahrstedt, Angrave, Abdelzaher

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

Copyright ©: Nahrstedt, Angrave, Abdelzaher

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");
  }
}
```


Copyright ©: Nahrstedt, Angrave, Abdelzaher

Example 1

How to modify the example to print the program name before the error message?

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

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

Copyright ©: Nahrstedt, Angrave, Abdelzaher

Close

- **int close(int fd)**

Tells the operating system you are done with a file descriptor.

```
#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?

Copyright ©: Nahrstedt, Angrave, Abdelzaher

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 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
    - Otherwise, what could happen?

Copyright ©: Nahrstedt, Angrave, Abdelzaher

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);
}
```


Copyright ©: Nahrstedt, Angrave, Abdelzaher

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 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

Copyright ©: Nahrstedt, Angrave, Abdelzaher

Example 3

```
#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), which returned %d\n",
          fd,  strlen("cs360\n"),  sz);

  close(fd);
}
```


Copyright ©: Nahrstedt, Angrave, Abdelzaher

lseek

- All open 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/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' 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"

Copyright ©: Nahrstedt, Angrave, Abdelzaher

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);
…:
```