Copyright ©: University of Illinois CS 241 Staff

This lecture

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

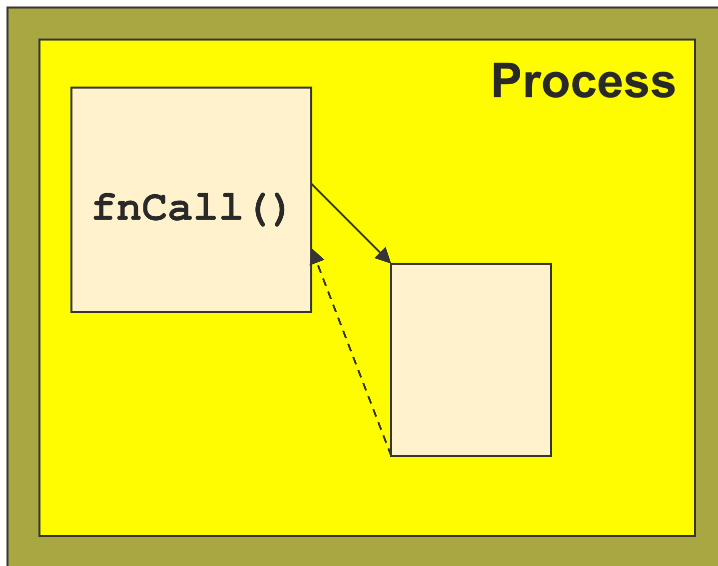
- Things covered in this lecture
  - Basic file system calls
  - Basic concepts about UNIX I/O

Copyright ©: University of Illinois CS 241 Staff

System Calls versus Function Calls?

System Calls versus Function Calls?

Function Call

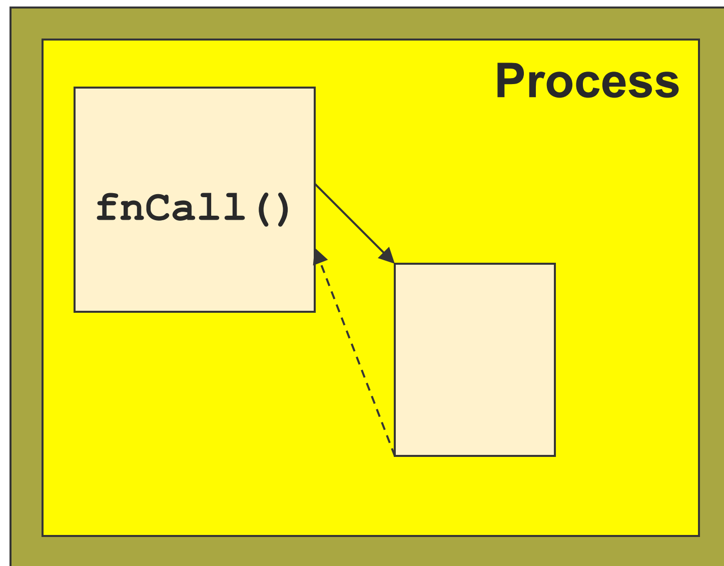


Caller and callee are in the same Process
- Same user
- Same "domain of trust"

Copyright ©: University of Illinois CS 241 Staff

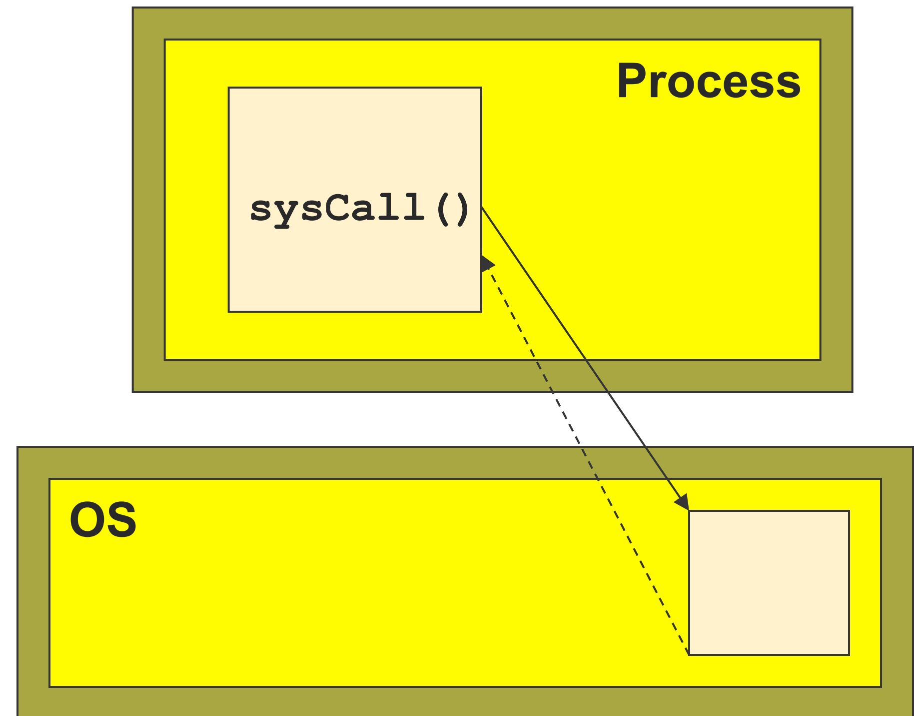
System Calls versus Function Calls?

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 ©: University of Illinois CS 241 Staff

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
    - Store sys_call arguments in registers and switch to kernel mode
    - The OS code takes control of the CPU, privileges are updated, and value of all registers is saved
    - The OS examines the call parameters and type of sys_call
    - The OS performs the requested function
    - The OS restores value of all registers
    - The OS returns control of the CPU to the caller

Copyright ©: University of Illinois CS 241 Staff

Copyright ©: Nahrstedt, Angrave, Abdelzaher

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

main()

....
....
push arguments
_libc_read()

__libc_read()

load args to regs
EAX = __NR_read
int 0x80

arch/i386/kernel/entry.S
system_call()

SAVE_ALL
check limit of EAX
syscall_tab[EAX]()

fs/read_write.c
sys_read()

file = fget(fd)
check file ops
check file locks
(file->f_op->read)()

filesystem or network or device code

check destination
retrieve data
copy data
return

fput(file)
return

handle signals
possibly schedule
RESTORE_ALL
iret

check error
return

pop arguments
....
....
....
....
....

User space

Kernel space

Copyright ©: Nahrstedt, Angrave, Abdelzaher

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 processor registers. In our example, the number associated to read is __NR_read, defined in <asm/ unistd.h>.

- After the switch to kernel mode, the processor must save all of its registers and dispatch execution to the proper kernel function, after checking whether EAX is out of range. The system call we are looking at is implemented in the sys_read function. The read finally performs the data transfer and all the previous steps are unwound up to the calling user function.

Copyright ©: Nahrstedt, Angrave, Abdelzaher

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

- The cost of using a system call
  - 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.

Examples of System Calls

- Examples
  - **`getuid()`** //get the user ID
  - **`fork()`** //create a child process
  - **`execve()`** //executing a program
- Don't mix system calls with standard library calls **`See man syscalls`**
  - Differences?
  - Is **`printf()`** a system call?
  - Is **`rand()`** a system call?

Copyright ©: University of Illinois CS 241 Staff

How do we know what is and what isn't a system call?

open(2) - Linux man page

die.net

site search

Google Custom Search

Library
linux docs
linux man pages
online dictionary
page load time

Toys
world sunlight
moon phase
trace explorer

strcat(3) - Linux man page

Name

strcat, strncat - concatenate two strings

Synopsis

```
#include <string.h>

char *strcat(char *dest, const char *s

char *strncat(char *dest, const char *s
```

Description

The strcat() function appends the src string to t
not overlap, and the dest string must have enou

The strncat() function is similar, except that

*

it will use at most n characters from src; and

*

Library calls often invoke system calls!

`malloc(3)` calls **`sbrk(2)`**

- 2: System Call
- 3: Library Call

Copyright ©: University of Illinois CS 241 Staff

Major System Calls

Process Management	
`pid = fork( )`	Create a child process identical to the parent
`pid = waitpid(pid, &statloc, options)`	Wait for a child to terminate
`s = execve(name, argv, environp)`	Replace a process' core image
`exit(status)`	Terminate process execution and return status
File Management	**Today**
`fd = open(file, how, ...)`	Open a file for reading, writing or both
`s = close(fd)`	Close an open file
`n = read(fd, buffer, nbytes)`	Read data from a file into a buffer
`n = write(fd, buffer, nbytes)`	Write data from a buffer into a file
`position = lseek(fd, offset, whence)`	Move the file pointer
`s = stat(name, &buf)`	Get a file's status information

Copyright ©: University of Illinois CS 241 Staff

File System and I/O Related System Calls

- A file system
  - A means to organize, retrieve, and update data in persistent storage
  - A hierarchical arrangement of directories
  - Bookkeeping information (file metadata)
    - File length, # bytes, modified timestamp, etc
- Unix file system
  - Root file system starts with "/"

Copyright ©: University of Illinois CS 241 Staff

System Calls for I/O

- Open (and/or create) a file for reading, writing or both

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

- Read data from file referenced by fd into a buffer

```
size_t read (int fd, void* buf, size_t nbyte);
```

- Write data from buffer to file referenced by fd

```
size_t write (int fd, void* buf, size_t nbyte);
```

- Close a file

```
int close(int fd);
```

- Get information about a file

```
int stat(const char* name, struct stat* buf);
```

Copyright ©: University of Illinois CS 241 Staff

UNIX I/O

- A UNIX file is a sequence of m bytes
  - $B_0, B_1, ... , B_k , ... , B_{m-1}$
- All I/O devices (e.g., network, disks, terminals) are modeled as files
  - A simple and elegant low-level interface
  - For example:
    - /dev/sda   Hard disk
    - /dev/tty   terminal for the current process

Copyright ©: University of Illinois CS 241 Staff

How UNIX represents open files

- Case of two descriptors referencing two distinct open files

File descriptor table
(one table per process)

stdin	fd=0
stdout	fd=1
stderr	fd=2
	fd=3
	fd=4

List of open file obj.
(shared by all processes)

File "foo.txt"

file offset
file object's usage cnt = 1
…

File "readme"

file offset
file object's usage cnt = 1
…

List of v-nodes
(shared by all processes)

file type
file size
# of hard links
…

file type
file size
# of hard links
…

File sharing in UNIX

- Two file descriptors sharing the same file ➔ call open(“foo.txt”,…) twice

File descriptor table
(one table per process)

stdin	fd=0
stdout	fd=1
stderr	fd=2
	fd=3
	fd=4

List of open file obj.
(shared by all processes)

File “foo.txt”

file offset
file object's usage cnt = 1
…

File “foo.txt”

file offset
file object's usage cnt = 1
…

List of v-nodes
(shared by all processes)

file type
file size
# of hard links
…

The two open file objects share the same file but NOT the file offset

I/O Redirection: Output (Input)

- unix> ls > foo.txt ➔ redirect output by calling sys_call: **dup2(4,1)**

File descriptor table (one table per process)	
stdin	fd=0
stdout	fd=1
stderr	fd=2
	fd=3
	fd=4

List of open file obj. (shared by all processes)

File "foo.txt"

file offset
file object's usage cnt = 2
…

List of v-nodes (shared by all processes)

file type
file size
# of hard links
…

System call
#include <unistd.h>
int dup2(int oldfd, int newfd);

Copyright ©: University of Illinois CS 241 Staff

File offset

- All open files have a "file offset" associated with them to record the current position for the next file operation
- On open
  - File offset points to the beginning of the file
- After reading/writing m bytes
  - File offset moves m bytes forward

Copyright ©: University of Illinois CS 241 Staff

File: Open

```
#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`: if this flag is specified in conjunction with O_CREAT, and pathname already exists, then open() will fail.

Copyright ©: University of Illinois CS 241 Staff

File: Close

```
#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

Copyright ©: University of Illinois CS 241 Staff

File: Read

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

- Read data from file referenced by fd into a buffer
  - Read `cnt` 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 where to store data to
  - `cnt`: length (in bytes) of data to be read

Copyright ©: University of Illinois CS 241 Staff

File: Read

```
size_t read (int fd, void* buf, size_t cnt);
```

- Things to be careful about
  - `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?
  - `cnt` is the requested number of bytes read, while the return value is the actual number of bytes read
    - How a “short count” could happen?

Copyright ©: University of Illinois CS 241 Staff

File: Write

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

- Write data from buffer to file referenced by fd
  - 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
  - -1 on error, sets `errno`
  - -1 on signal interrupt, sets `errno` to `EINTR`

- Parameters:
  - `fd`: file descriptor
  - `buf`: buffer
  - `cnt`: length of buffer

Copyright ©: University of Illinois CS 241 Staff

File: Write

```
size_t write (int fd, void* buf, size_t cnt);
```

- 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`
    - If not, what will happen?
  - `cnt` is the requested number of bytes to write, while the return value is the actual number of bytes written
    - How a "short count" could happen?

Copyright ©: University of Illinois CS 241 Staff

Standard Input, Standard Output and Standard Error

- 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, ...);`
- Example of library functions from standard I/O Library (**stdio.h**)
  - `fprintf(FILE *stream, const char *format, …);`
  - `scanf(const char *format, …);`
  - FILEs are a buffering wrapper around UNIX file descriptors

Copyright ©: University of Illinois CS 241 Staff

I/O Library Calls

- Every system call has paired procedure calls from the standard I/O library:

Unbuffered I/O

- System Call
  - **`open`**
  - **`close`**
  - **`read/write`**
  - **`lseek`**

Buffered I/O (stream stderr is <u>unbuffered</u>)

- Standard I/O call (**`stdio.h`**)
  - **`fopen`**
  - **`fclose`**
  - **`getchar/putchar, getc/putc, fgetc/fputc, fread/fwrite, gets/puts, fgets/fputs, scanf/printf, fscanf/fprintf`**
  - **`fseek`**

Copyright ©: University of Illinois CS 241 Staff

Basic Unix Concepts

- Error Model
  - `errno` variable
    - Unix provides a globally accessible integer variable that contains an error code number
  - 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.
  - Note that errno is undefined after a successful library call: this call may well change this variable, even though it succeeds, for example because it internally used some other library function that failed. Thus, if a failing call is not immediately followed by a call to perror(), the value of errno should be saved.

Copyright ©: University of Illinois CS 241 Staff

Basic Unix Concepts

- Error Model
  - 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 */
```

Read (write) with short count

- Sometimes read and write can transfer fewer bytes than the application requested. It is not an error!
  - **`Reaching EOF on reads`**
  - **`Reading text lines from terminal`**
  - …

- A special case: read/write returns -1; errno == EINTR
  - The system call was interrupted by a signal handler
  - It is a recoverable error ➔ just call read/write again!

Copyright ©: University of Illinois CS 241 Staff

An example: robust reading with Rio package

```
ssize_t rio_readn(int fd, void *usrbuf, size_t n)
{
    size_t nleft = n;
    ssize_t nread;
    char *bufp = usrbuf;

    while (nleft > 0) {
        if ((nread = read(fd, bufp, nleft)) < 0) {
            if (errno == EINTR) /* Interrupted by sig handler return */
                nread = 0;      /* and call read() again */
            else
                return -1;      /* errno set by read() */
        }
        else if (nread == 0)
            break;              /* EOF */
        nleft -= nread;
        bufp += nread;
    }
    return (n - nleft);         /* return >= 0 */
}
```

Downloaded from http://csapp.cs.cmu.edu/public/ics2/code/src/csapp.c
Online public document from "Computer Systems: A Programmer's Perspective"