System Calls and I/O

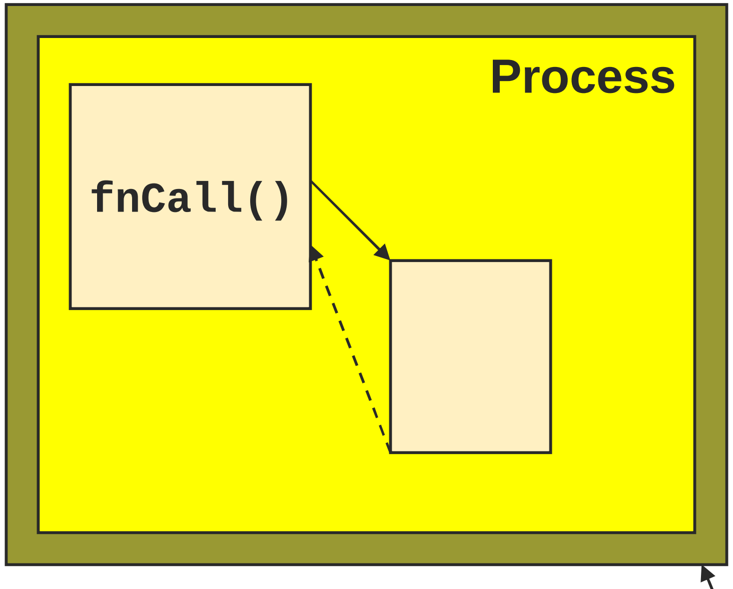
This lecture

- 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

System Calls versus Function Calls?

System Calls versus Function Calls

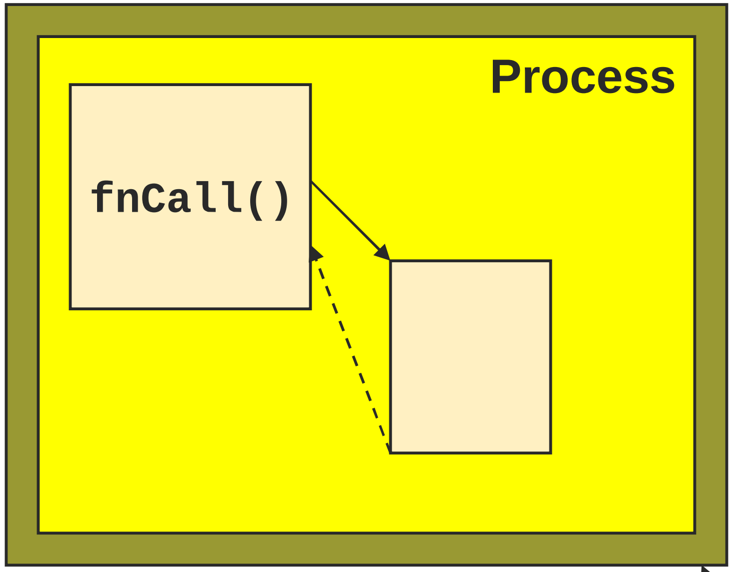
Function Call



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

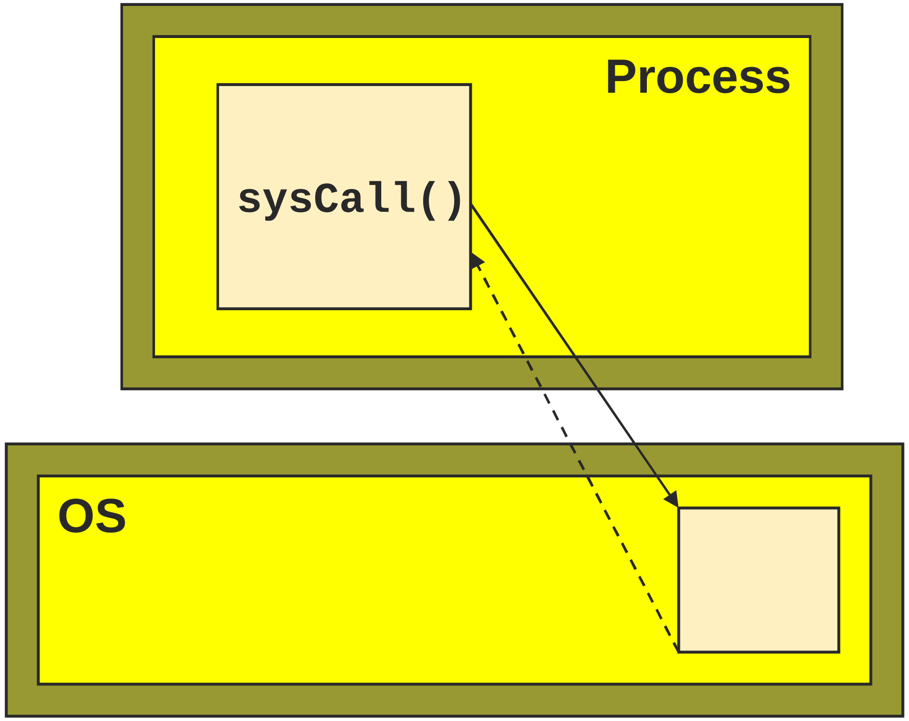
System Calls versus Function Calls

Function Call



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

System Call



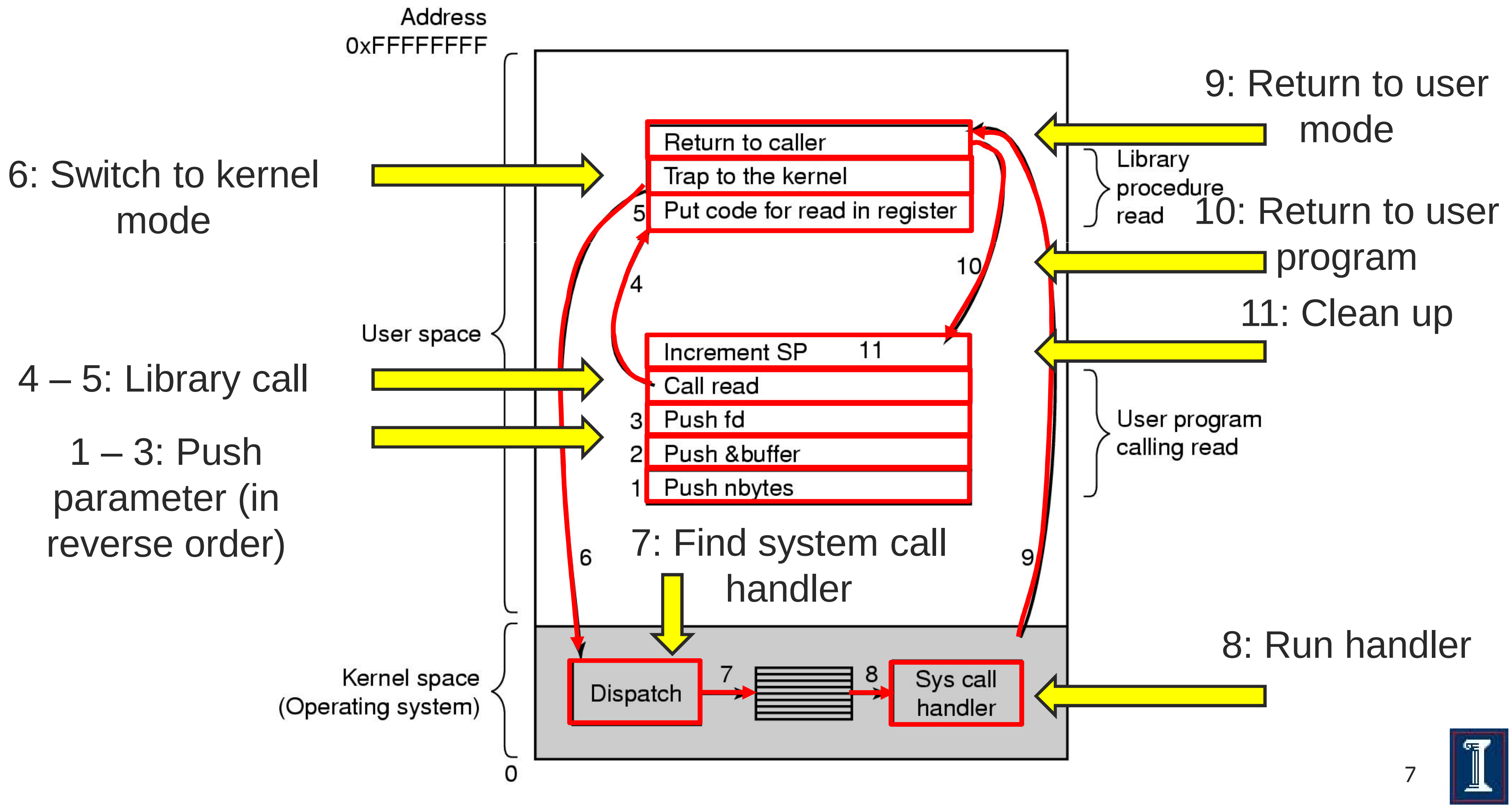
- OS is trusted; user is not.
- OS has super-privileges; user does not
- Must take measures to prevent abuse

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

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

```
count = read(fd, buffer, nbytes);
```



Examples of System Calls

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

`man syscalls`

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

Major System Calls

Directory and File System Management	
`s = mkdir(name, mode)`	Create a new directory
`s = rmdir(name)`	Remove an empty directory
`s = link(name, name)`	Create a new entry, name, pointing to name
`s = unlink(name)`	Remove a directory entry
`s = mount(special, name, flag)`	Mount a file system
`s = umount(special)`	Unmount a file system
Miscellaneous	
`s = chdir(dirname)`	Change the working directory
`s = chmod(name, mode)`	Change a file's protection bits
`s = kill(pid, signal)`	Send a signal to a process
`seconds = time(&seconds)`	Get the elapsed time since January 1, 1970

File System and I/O Related System Calls

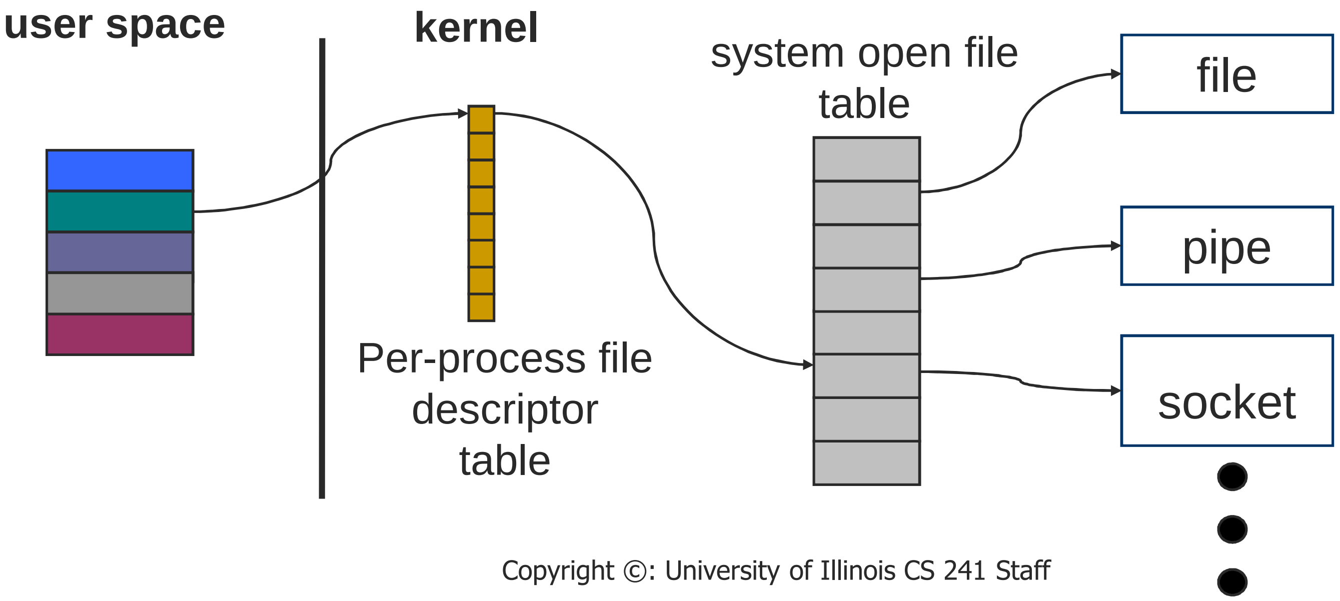
- A file system
  - A means to organize, retrieve, and updated data in persistent storage
  - A hierarchical arrangement of directories
  - Bookkeeping information (file metadata)
    - File length, # bytes, modified timestamp, etc
- Unix file system
  - Root file system starts with "/"

Why does the OS control I/O?

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

Basic Unix Concepts

- Input/Output – I/O
  - Per-process table of I/O channels
  - Table entries describe files, sockets, devices, pipes, etc.
  - Table entry/index into table called “file descriptor”
  - Unifies I/O interface



Basic Unix Concepts

- 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 */
```

System Calls for I/O

- Get information about a file

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

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

```
int open (const char* name, in flags);
```

- Read data from one buffer to file descriptor

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

- Write data from file descriptor into buffer

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

- Close a file

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

System Calls for I/O

- They look like regular procedure calls but are different
  - A system call makes a request to the operating system by trapping into kernel mode
  - A procedure call just jumps to a procedure defined elsewhere in your program
- Some library procedure calls may themselves make a system call
  - e.g., **`fopen()`** calls **`open()`**

File: Statistics

```
#include <sys/stat.h>
int stat(const char* name, struct stat* buf);
```

- Get information about a file
- Returns:
  - 0 on success
  - -1 on error, sets **errno**
- Parameters:
  - **name**: Path to file you want to use
    - Absolute paths begin with "/", relative paths do not
  - **buf**: Statistics structure
    - **off_t st_size**: Size in bytes
    - **time_t st_mtime**: Date of last modification. Seconds since January 1, 1970
- Also

```
int fstat(int filedes, struct stat *buf);
```

Example - (stat())

```
#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
int main(int argc, char **argv) {
    struct stat fileStat;
    if(argc != 2)
        return 1;
    if(stat(argv[1], &fileStat) < 0)
        return 1;
    printf("Information for %s\n",argv[1]);
    printf("---------------------------\n");
    printf("File Size: \t\t%d bytes\n", fileStat.st_size);
    printf("Number of Links: \t%d\n", fileStat.st_nlink);
    printf("File inode: \t\t%d\n", fileStat.st_ino);
```

Example - (stat())

```
    printf("File Permissions: \t");
    printf( (S_ISDIR(fileStat.st_mode)) ? "d" : "-");
    printf( (fileStat.st_mode & S_IRUSR) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWUSR) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXUSR) ? "x" : "-");
    printf( (fileStat.st_mode & S_IRGRP) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWGRP) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXGRP) ? "x" : "-");
    printf( (fileStat.st_mode & S_IROTH) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWOTH) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXOTH) ? "x" : "-");
    printf("\n\n"); printf("The file %s a symbolic link\n",
    (S_ISLNK(fileStat.st_mode)) ? "is" : "is not");
    return 0;
}
```

Useful Macros: File types

- Is file a symbolic link
  - **`S_ISLNK`**
- Is file a regular file
  - **`S_ISREG`**
- Is file a character device
  - **`S_ISCHR`**
- Is file a block device
  - **`S_ISBLK`**
- Is file a FIFO
  - **`S_ISFIFO`**
- Is file a unix socket
  - **`S_ISSOCK`**

Useful Macros: File Modes

- **S_IRWXU**
  - read, write, execute/search by owner
- **S_IRUSR**
  - read permission, owner
- **S_IWUSR**
  - write permission, owner
- **S_IXUSR**
  - execute/search permission, owner
- **S_IRGRP**
  - read permission, group
- **S_IRWXO**
  - read, write, execute/search by others

Example - (stat())

```
Information for testfile.sh
---------------------------
File Size: 36 bytes
Number of Links: 1
File inode: 180055
File Permissions: -rwxr-xr-x

The file is not a symbolic link
```

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`**: prevent creation if it already exists

Example (open())

```
#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");
    }
}
```

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

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

Example (close())

```
#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");
```

Example (close())

```
#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");
```

After close, can you still use the file descriptor?

Why do we need to close a file?

File: Read

```
#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

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 could this happen?

Example (**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);
}
```

File: Write

```
#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

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 could this happen?

Copyright ©: University of Illinois CS 241 Staff

Example (write())

```
#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);
}
```

File Pointers

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

File: Seek

```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

- Explicitly set the file offset for the open file
- Return: Where the file pointer is
  - the new offset, in bytes, from the beginning of the file
  - -1 on error, sets `errno`, file pointer remains unchanged
- Parameters:
  - `fd`: file descriptor
  - `offset`: indicates relative or absolute location
  - `whence`: How you would like to use `lseek`
    - `SEEK_SET`, set file pointer to `offset` bytes from the beginning of the file
    - `SEEK_CUR`, set file pointer to `offset` bytes from current location
    - `SEEK_END`, set file pointer to `offset` bytes from the end of the file

File: Seek Examples

- Random access
  - Jump to any byte in a file
- Move to byte #16

```
newpos = lseek(fd, 16, SEEK_SET);
```

- Move forward 4 bytes

```
newpos = lseek(fd, 4, SEEK_CUR);
```

- Move to 8 bytes from the end

```
newpos = lseek(fd, -8, SEEK_END);
```

Example (lseek())

```
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("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, 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, ...);`**
- Two additional library functions
  - **`printf();`**
  - **`scanf();`**

I/O Library Calls

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

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

- 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`

Stream Processing - `fgetc()`

```
int fgetc(FILE *stream);
```

- Read the next character from **`stream`**
- Return
  - An *unsigned char* cast to an *int*
  - **`EOF`** on end of file
  - Error

Similar functions for writing:

```
int fputc(int c, FILE *stream);
int putchar(int c);
int putc(int c, FILE *stream);
```

```
int getchar(void);
```

  - Read the next character from **`stdin`**

```
int getc(void);
```

  - Similar to , but implemented as a macro, faster and potentially unsafe

Stream Processing - `fgets()`

```
char *fgets(char *s, int size, FILE
   *stream);
```

- Read in at most one less than **`size`** characters from **`stream`**
  - Stores characters in buffer pointed to by *s*.
  - Reading stops after an **`EOF`** or a newline.
  - If a newline is read, it is stored into the buffer.
  - A **`'\0'`** is stored after the last character in the buffer.
- Return
  - **`s`** on success
  - NULL on error or on **`EOF`** and no characters read

Similar:

```
int fputs(const char *s, FILE *stream);
```

Stream Processing

```
char *gets(char *s);
```

- NOTE: DO NOT USE
  - Reading a line that overflows the array pointed to by *s* causes undefined results.
  - The use of is `fgets()` recommended

Stream Processing - `fputs()`

```
int fputs(const char *s, FILE *stream);
```

- Write the null-terminated string pointed to by **`s`** to the stream pointed to by **`stream`**.
  - The terminating null byte is not written
- Return
  - Non-neg number on success
  - **`EOF`** on error

```
char *puts(char *s);
```

- Write to **`stdout`**
  - Appends a newline character

Example: (fgets()- fputs())

```
#include <stdio.h>
int main() {
   FILE * fp = fopen("test.txt", "r");
   char line[100];
   while( fgets(line, sizeof(line), fp) != NULL )
      fputs(line, stdout);
   fclose(fp);
   return 0;
}
```

Stream Processing - `fscanf()`

```
int scanf(const char *format, ... );
```

- Read from the standard input stream **`stdin`**
  - Stores read characters in buffer pointed to by *s*.
- Return
  - Number of successfully matched and assigned input items
  - **`EOF`** on error

```
int fscanf(FILE *stream, const char *fmt, ... );
```

  - Read from the named input **`stream`**

```
int sscanf(const char *s, const char *fmt, ... );
```

  - Read from the string **`s`**

Example: (**scanf()**)

- Input: **56789 56a72**

```
#include <stdio.h>
int main() {
   int i;
   float x;
   char name[50];
   scanf("%2d%f %[0123456789]", &i, &x, name);
}
```

What are **i**, **x**, and **name** after the call to **scanf()**?

What will a subsequent call to **getchar()** return?

Example: **stdin**

```
int x;
char st[31];

/* read first line of input */
printf("Enter an integer: ");
scanf("%d", &x);

/* read second line of input */
printf("Enter a line of text: ");
fgets(st, 31, stdin);
```

What will this code really do?

Example: **stdin**

```
int x;
char st[31];

/* read first line of input */
printf("Enter an integer: ");
scanf("%d", &x);

/* read second line of input */
printf("Enter a line of text: ");
fgets(st, 31, stdin);
```

What will this code really do?

Input is buffered, but **`scanf()`** did not read all of the first line

Example: **stdin**

```
int x;
char st[31];
/* read first line */
printf("Enter an
   integer: ");
scanf("%d", &x);
dump_line(stdin);
/* read second line */
printf("Enter a line of
   text: ");
fgets(st, 31, stdin);
```

```
void dump_line( FILE * fp
   ) {
   int ch;
   while((ch = fgetc(fp))
      != EOF &&
      ch != '\n' )
      /* null body */;
   }
```

Read and dump all characters from input buffer until a **`'\n'`** after **`scanf()`**