CS35L Software Construction Laboratory

Lab 5: Sneha Shankar

Week 6; Lecture 1

Assignment 10 rubrics

- Presentation (50%):
  - Organization
  - Relevance to topic
  - Technical Details and Subject Knowledge
  - Presentation abilities (Elocution and Eye contact)
  - Content of slides (not dull and boring)
  - Ability to answer questions and interactivity with audience

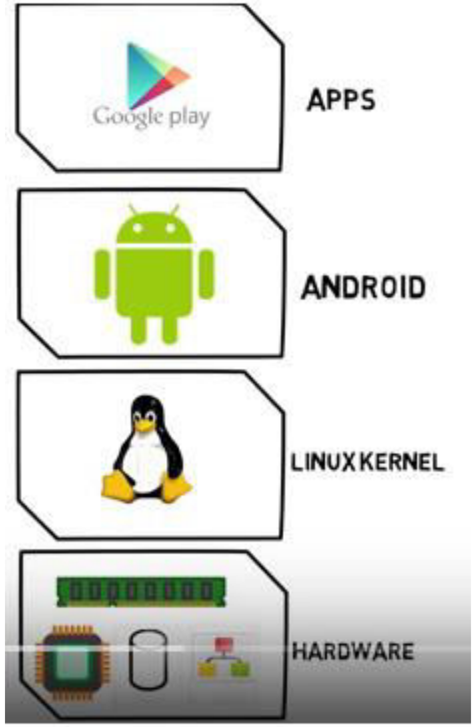
- Report (50%)

System Call Programming and Debugging

Kernel

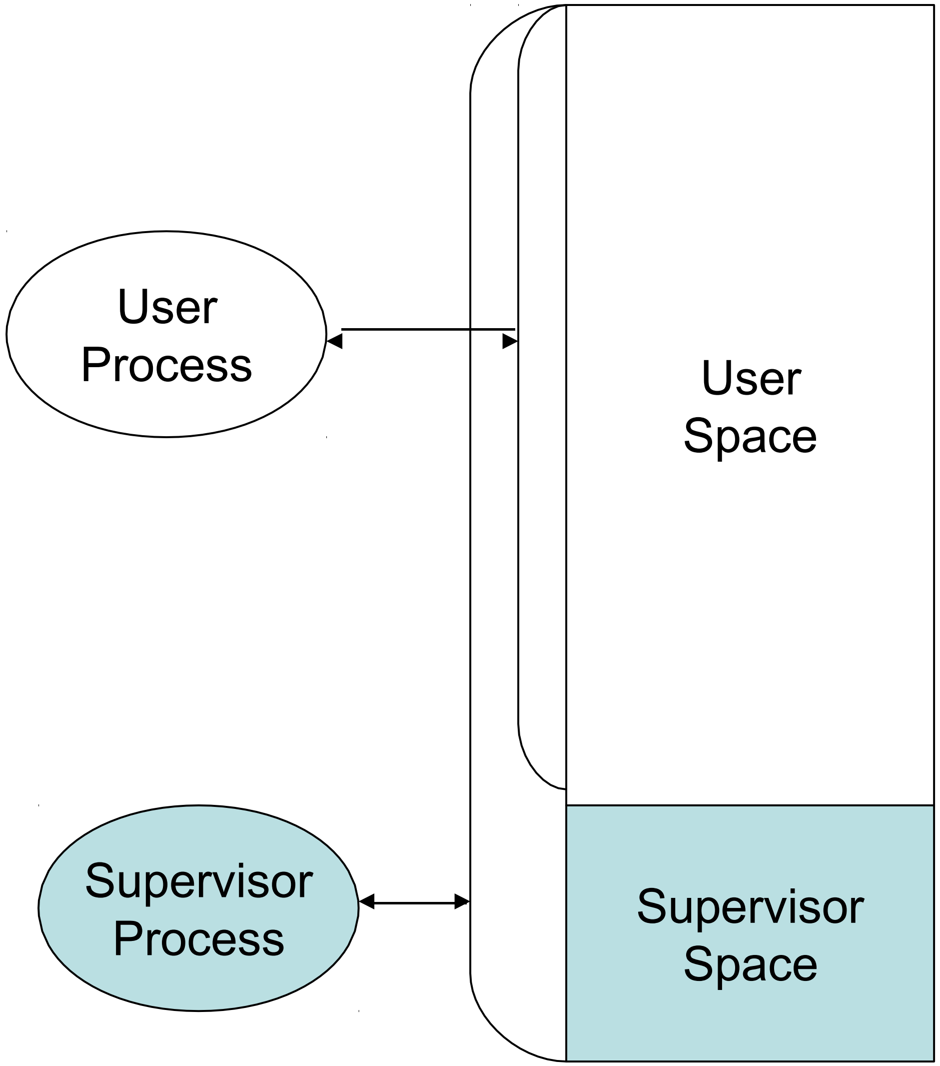
- kernel is the core of the OS
  - interface between hardware and software
  - controls access to system resources: memory, I/O, CPU
  - Manages CPU resources, memory resources, processes
  - Lowest layer above the CPU
  - ensure protection and fair allocations

An example: Android OS



Modes

- Operating modes that place restrictions on the type of operations that can be performed by running processes
  - User mode: restricted access to system resources
  - Kernel/Supervisor mode: unrestricted access



User Mode vs. Kernel Mode

- These are the two modes in which a program executes
- Hardware contains a mode-bit, e.g. 0 means kernel mode, 1 means user mode
- User mode
  - CPU **restricted** to unprivileged instructions and a specified area of memory
  - Less privileged
  - Exception will crash single process
- Kernel mode
  - CPU is **unrestricted**, can use all instructions, access all areas of memory and take over the CPU anytime
  - High privilege
  - Exception will crash the entire OS

User space v/s Kernel Space

- User space - where normal user processes run
  - limited access to system resources: memory, I/O, CPU

- Kernel space
  - stores the code of the kernel, which manages processes
  - prevent processes messing with each other and the machine
  - only the kernel code is trusted

Which Code is Trusted?
=> The Kernel *ONLY*

- Core of OS software **executing in supervisor** state
- **Trusted software:**
  - Manages hardware resources (CPU, Memory and I/O)
  - Implements protection mechanisms that could not be changed through actions of untrusted software in user space

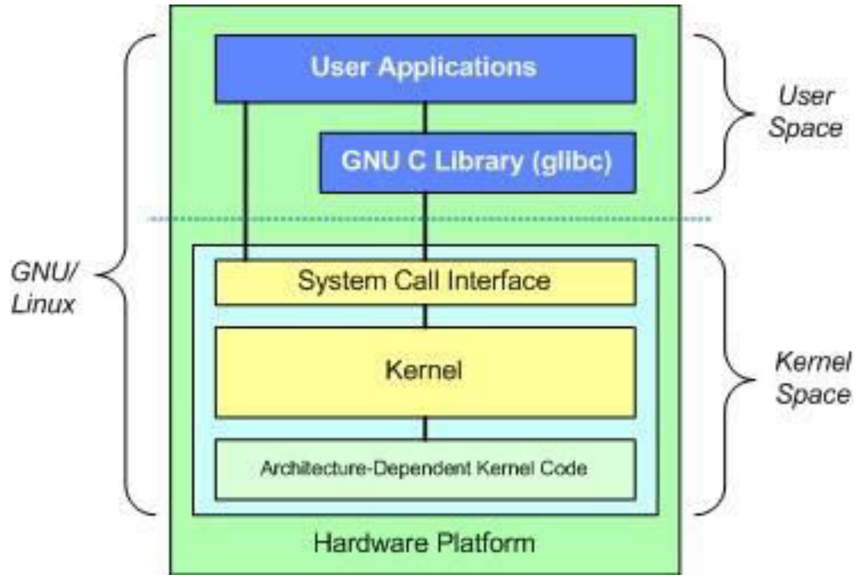
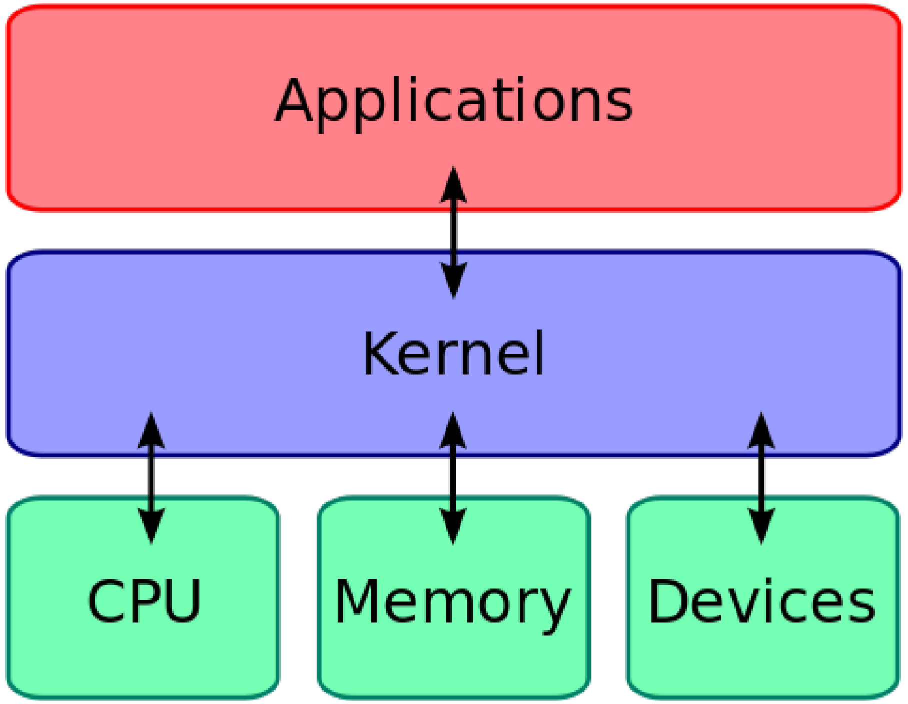


Image by: Tim Jones (IBM)

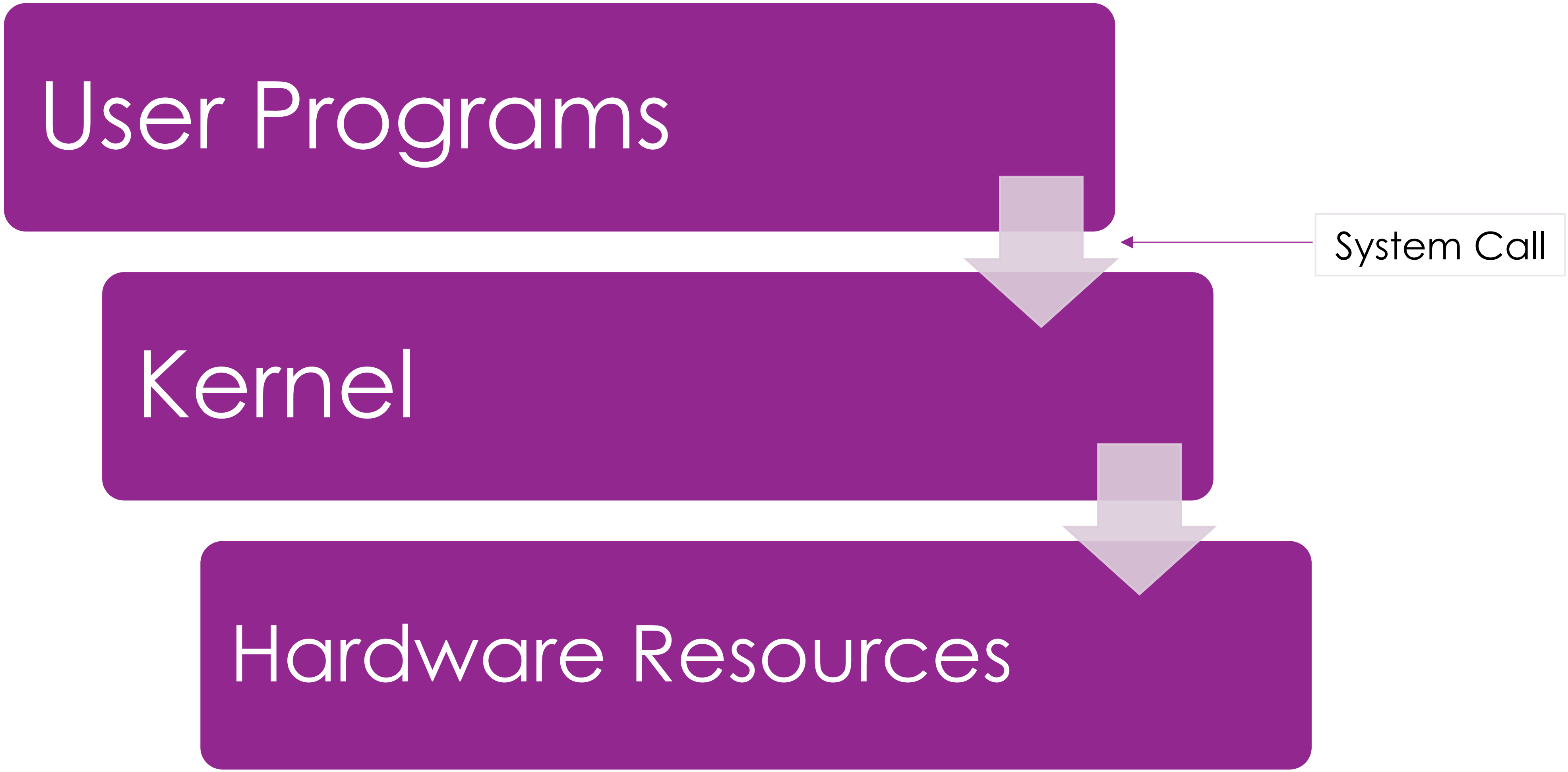
What About User Processes?

- The kernel executes privileged operations on behalf of untrusted user processes



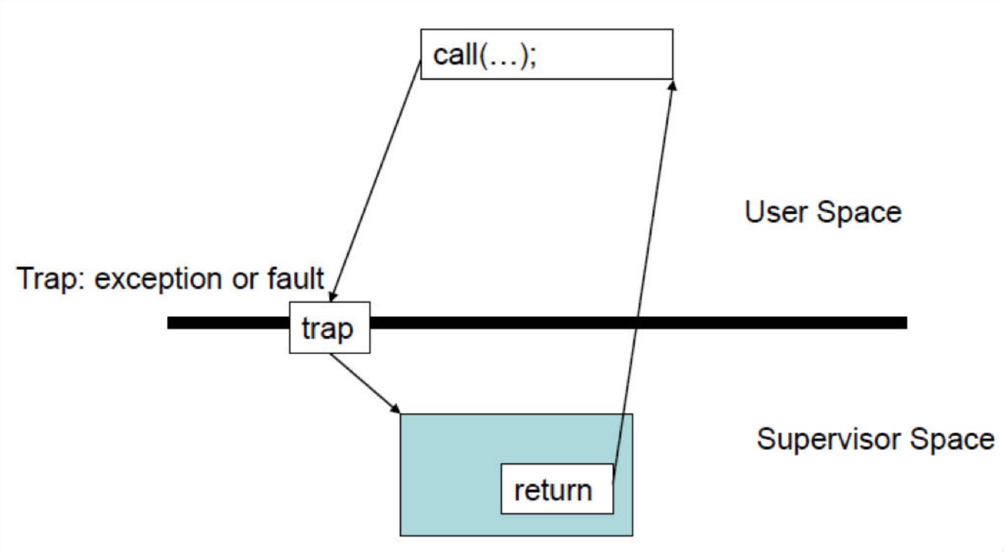
System Calls

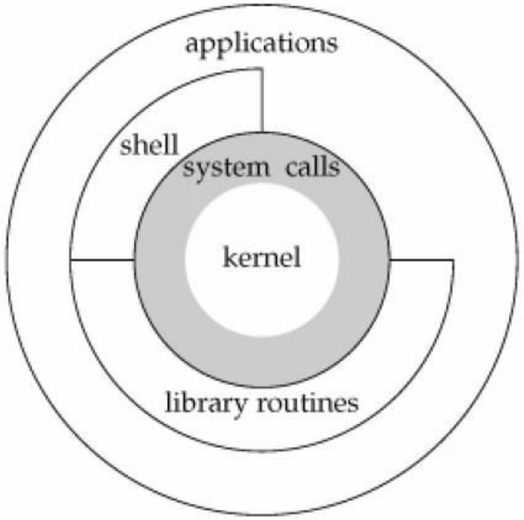
- Special type of function that:
  - Provide interface between user programs and OS
  - Used by user-level processes to request a service from the kernel
  - Changes the CPU’s mode from user mode to kernel mode to enable more capabilities
  - Is part of the kernel of the OS
  - Verifies that the user should be allowed to do the requested action and then does the action (kernel performs the operation on behalf of the user)
  - Is the ***only way*** a user program can perform privileged operations


User Programs
Kernel
Hardware Resources
System Call

System Calls

- When a system call is made, the program being executed is interrupted and control is passed to the kernel
- If operation is valid the kernel performs it





System Call Overhead

- System calls are expensive and can hurt performance
- The system must do many things
  - Process is interrupted & computer saves its state
  - OS takes control of CPU & verifies validity of op.
  - **OS performs requested action**
  - OS restores saved context, switches to user mode
  - OS gives control of the CPU back to user process

What actually happens?

- System call generates an interrupt
- OS gains control of the CPU
- OS finds out the type of system call
- OS creates the corresponding **interrupt handler**
- Routine is executed with this interrupt handler

Making a System Call

- System calls are directly available and used in high level languages like c and C++
- Hence, easy to use system calls in programs
- For a programmer, system calls are same as calling a procedure or function
- So, what is the difference between a system call and a normal function?
  - System call enters a kernel
  - Normal function does not and cannot enter a function!

Making a System Call

- App developers do not have direct access to system calls
- They have to invoke the API
- The functions in the API invoke the actual system calls
- Advantages:
  - Portability: as long as a system supports an API, any program using that API can compile and run
  - Ease of Use: using API is significantly easier than the actual system call

Types of System Calls

- 5 categories:

1. Process Control
   - A running program needs to be able to stop execution
   - Normally or abnormally
   - If abnormally, dump of memory is created and taken for examination by a debugger
2. File Management
   - To perform operations on files
   - Create, delete, read, write, reposition, close
   - Many a times, OS provides an API to make these system calls

Types of System Calls

3. Device Management
   - Process usually requires several resources to execute
   - If available, access granted
   - Resources = devices
   - Eg: physical I/O devices attached
4. Information Management
   - To transfer information between user program and OS
   - Eg: time, date
5. Communication
   - Interprocess communication
     - Message passing model
     - Shared memory model

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()

Executing a System Call

- Follows a sequence of steps
- There is a need to pass various parameters of system call to the OS
- Three methods:
  - Register method: parameters are stored in the registers of the CPU
  - If size of parameters are huge, a block of memory is used. Address of that block is stored in Registers
  - Stack Method (in memory): parameters are pushed in the stack and OS pops it out
- These are later saved in the processor registers
- OS checks the system call code and creates the interrupt handler
- Control is passed to the interrupt handler

What if there were no System Calls?

- Kernel can be accessed by anyone!
- Threat to the security of OS

Example System Calls

```
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
```

- `int open(const char *pathname, int flags, mode_t mode);`
- `int close(int fd);`
- File descriptors
  - 0 stdin
  - 1 stdout
  - 2 stderr
- `ssize_t read(int fildes, void *buf, size_t nbyte)`
  - fildes: file descriptor
  - buf: buffer to write to
  - nbyte: number of bytes to read
- `ssize_t write(int fildes, const void *buf, size_t nbyte);`
  - fildes: file descriptor
  - buf: buffer to write from
  - nbyte: number of bytes to write

Example System Calls

```
#include <sys/stat.h>
int fstat(int filedes, struct stat *buf)
```

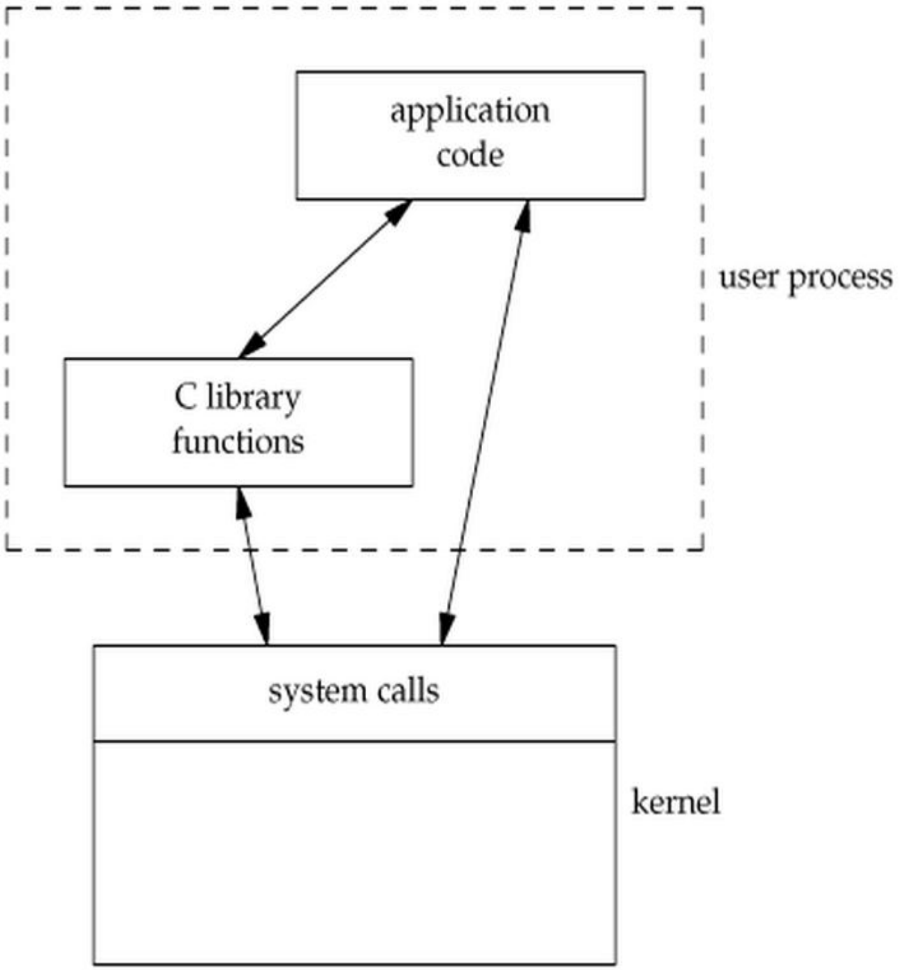
– Returns information about the file with the descriptor filedes into buf

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for file system I/O */
    blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

Library Functions

- Functions that are a part of standard C library
- To avoid system call overhead use equivalent library functions
  - getchar, putchar vs. read, write (for standard I/O)
  - fopen, fclose vs. open, close (for file I/O), etc.
- How do these functions perform privileged operations?
  - They make system calls

So What's the Point?



- Many library functions invoke system calls indirectly
- So why use library calls?
- Usually equivalent library functions make fewer system calls
- non-frequent switches from user mode to kernel mode => less overhead

Unbuffered vs. Buffered I/O

- **Unbuffered**
  - Every byte is read/written by the kernel through a system call
- **Buffered**
  - collect as many bytes as possible (in a buffer) and read more than a single byte (into buffer) at a time and use one system call for a block of bytes

=> Buffered I/O decreases the number of read/write system calls and the corresponding overhead