CS35L – Winter 2019

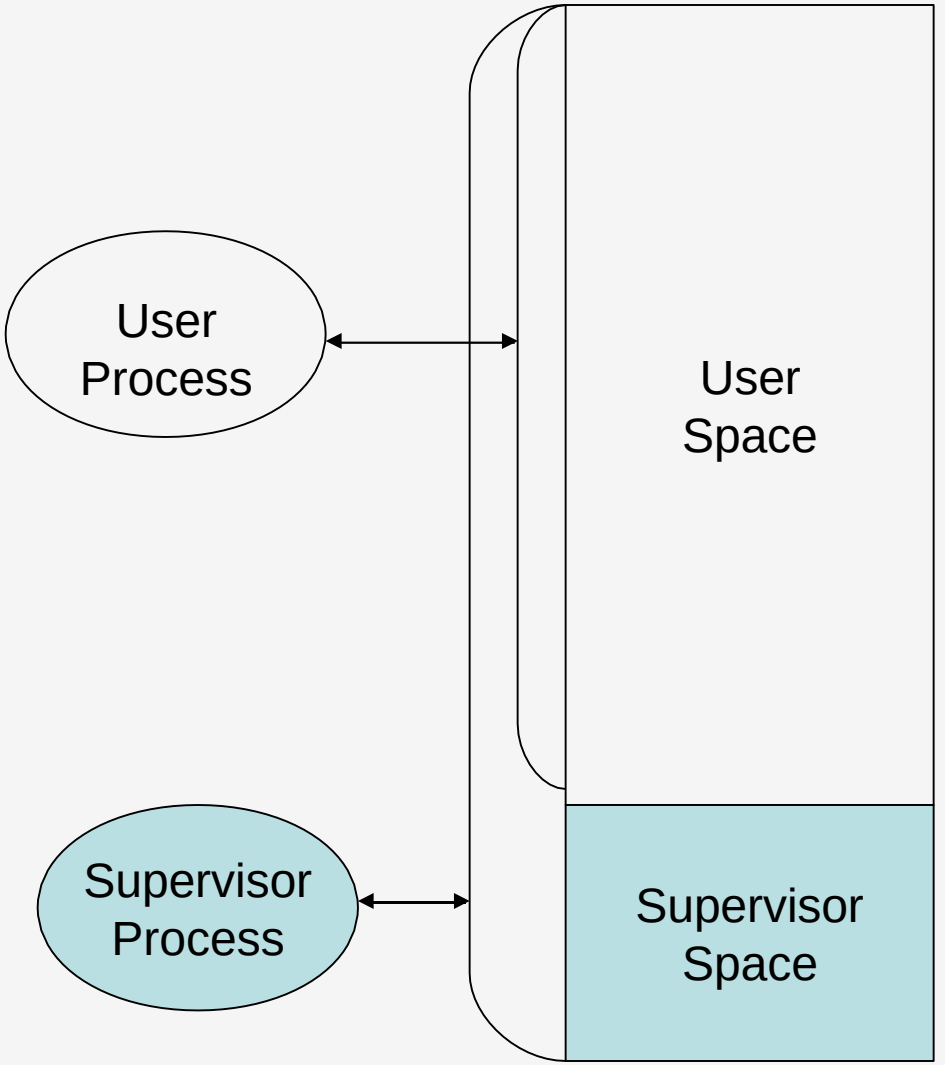
Slide set:	5.1
Slide topics:	System call programming
Assignment:	5

Processor 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



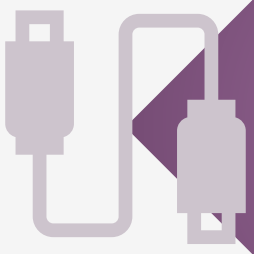
User Mode vs. Kernel Mode

- 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
- Supervisor/kernel mode
  - CPU is **unrestricted**, can use all instructions, access all areas of memory and take over the CPU anytime

Why Dual-Mode Operation?

- System resources are shared among processes
- OS must ensure:
  - **Protection**
    - an incorrect/malicious program cannot cause damage to other processes or the system as a whole
  - **Fairness**
    - Make sure processes have a fair use of devices and the CPU

Goals for Protection and Fairness

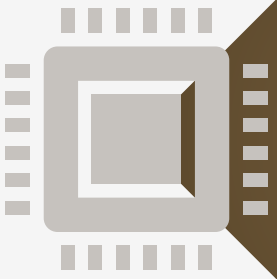


I/O Protection

- Prevent processes from performing illegal I/O operations

Memory Protection

- Prevent processes from accessing illegal memory and modifying kernel code and data structures



CPU Protection

- Prevent a process from using the CPU for too long

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
- **System call interface** is a **safe way** to expose privileged functionality and services of the processor

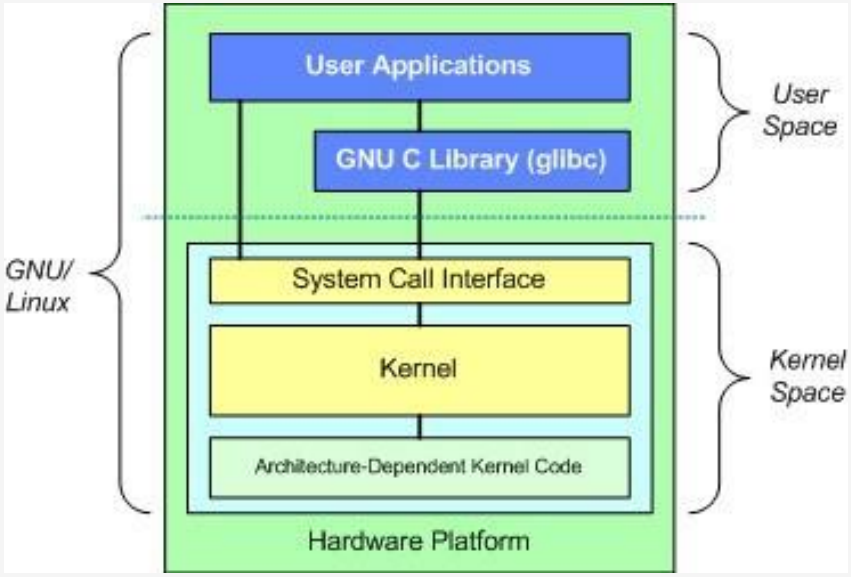
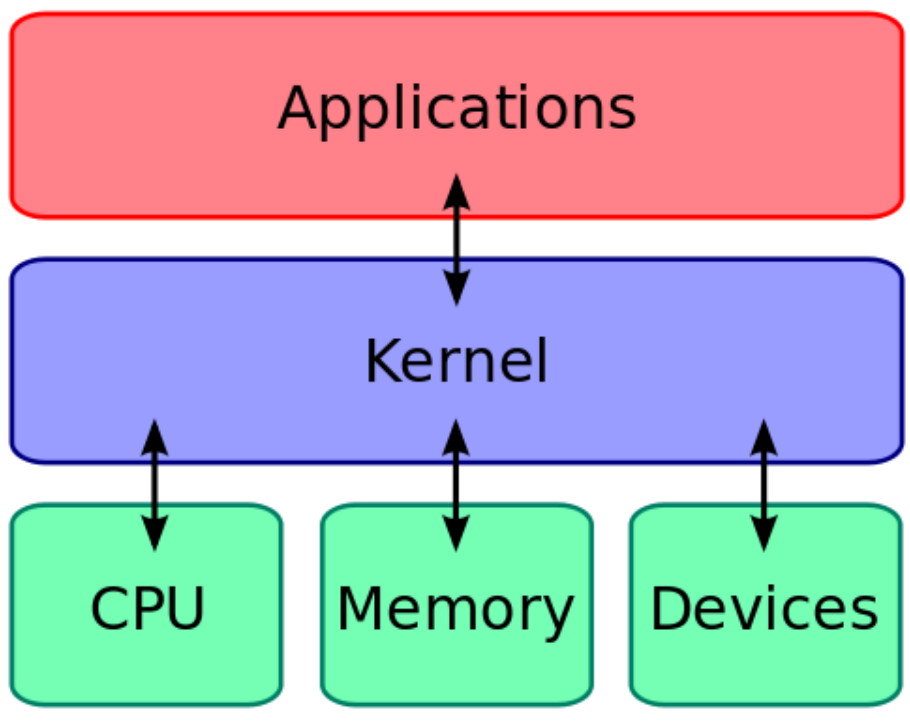


Image by: Tim Jones (IBM)

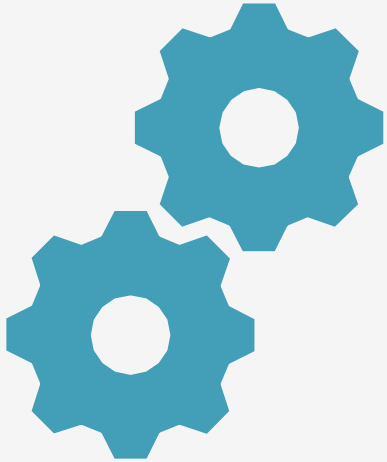
What About User Processes?

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



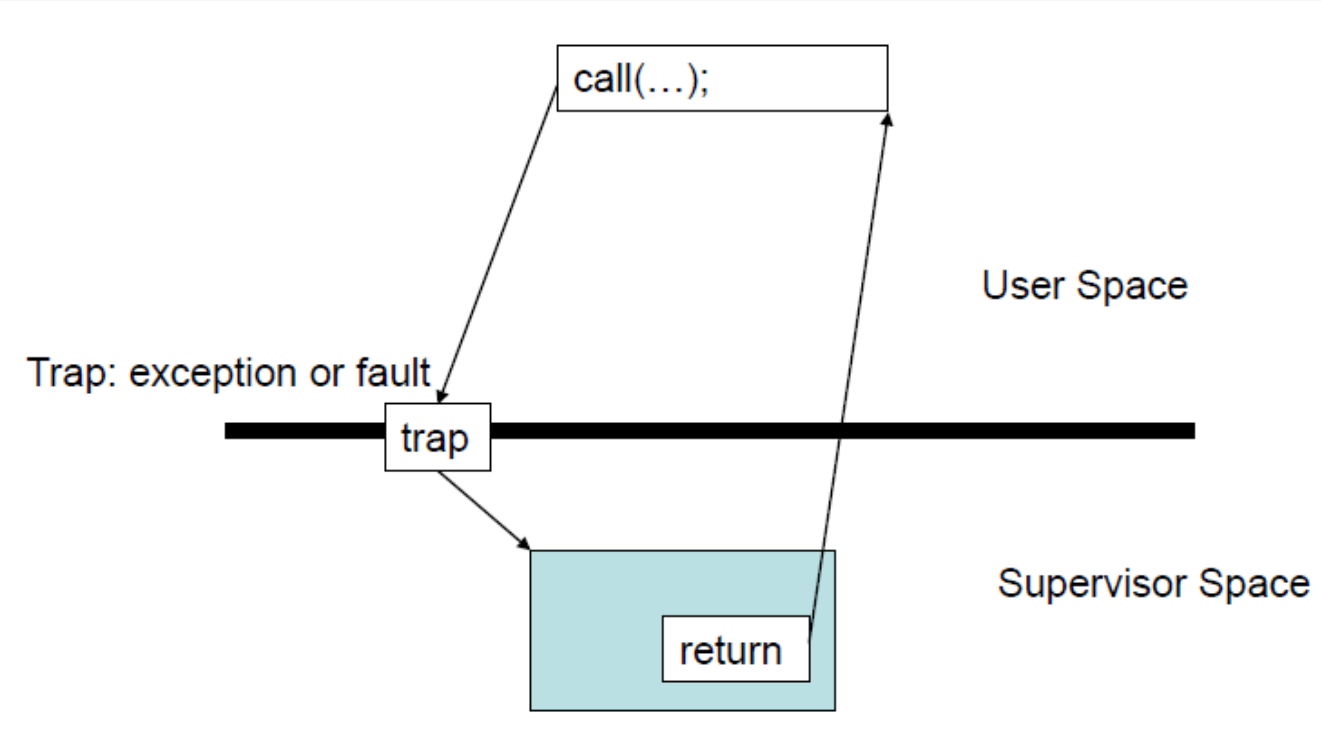
System Calls

- Special type of function that:
  - Is 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



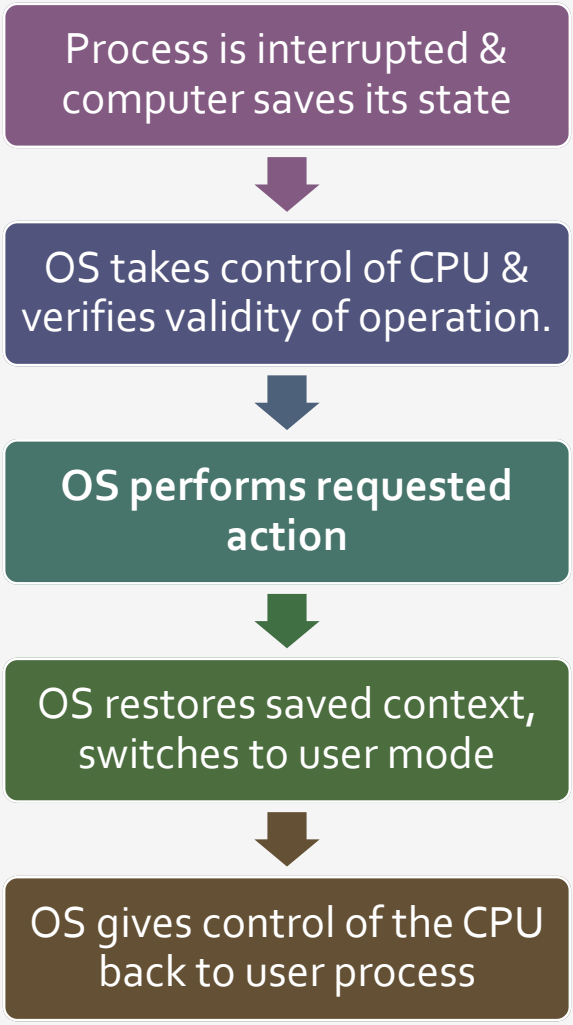
System Calls

- 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



Example System Calls

```
#include<unistd.h>
```

- `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
- `int` **`open`**`(const char *pathname, int flags, mode_t mode);`
- `int` **`close`**`(int fd);`
- File descriptors
  - 0 stdin
  - 1 stdout
  - 2 stderr

Example System Calls

`pid_t` **`getpid`**`(void)`

- Returns the process ID of the calling process

`int` **`dup`**`(int fd)`

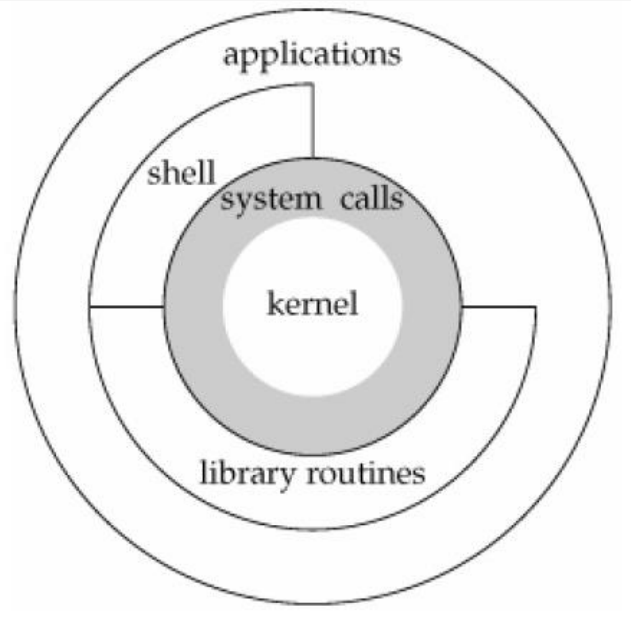
- Duplicates a file descriptor fd. Returns a second file descriptor that points to the same file table entry as fd does.

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

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

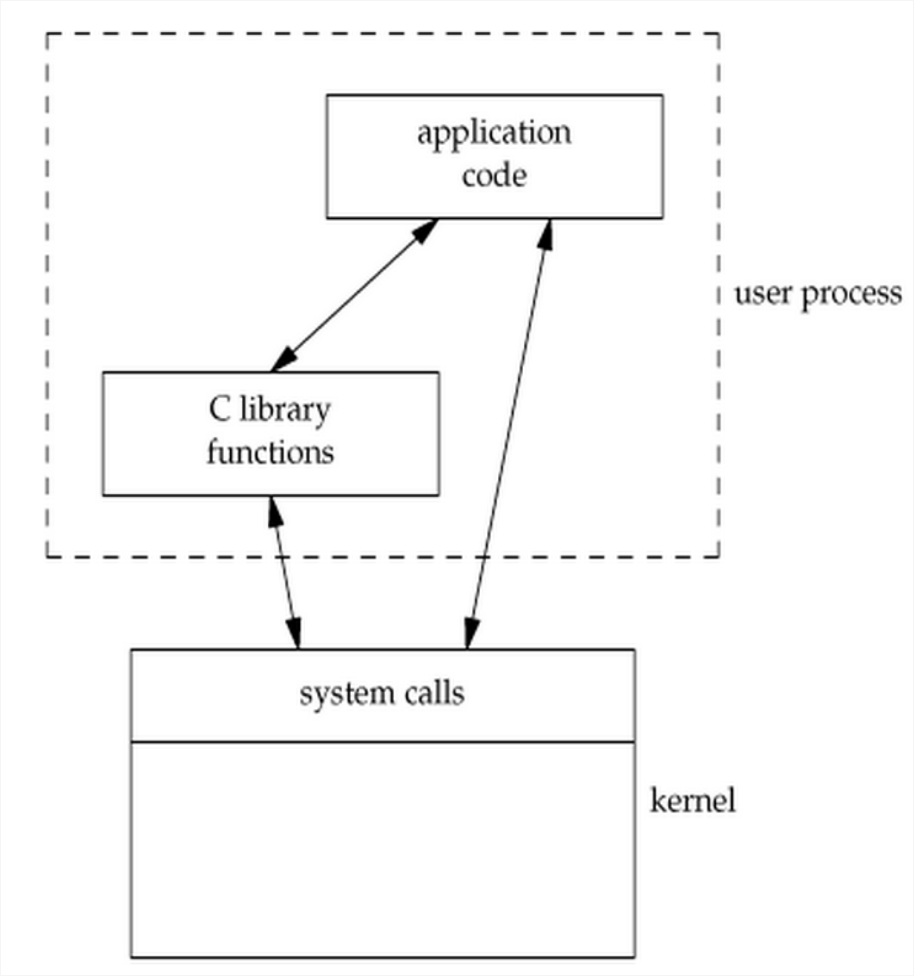
```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group 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 an internal 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