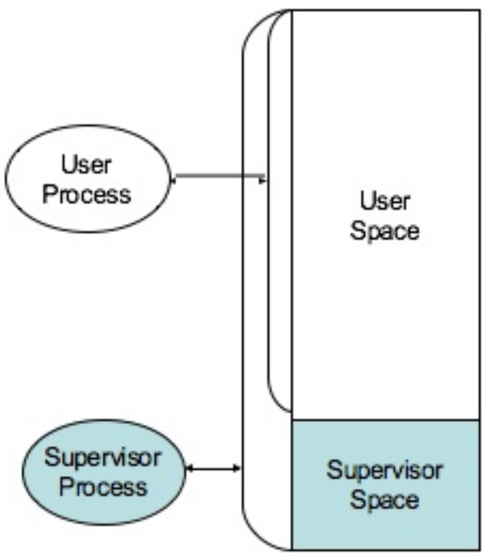
Week 7: System Call Programming

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
  - **kerner/supervisor mode**: unrestricted access



User Mode vs. Kernel Mode

- hardware contains a mode-bit
  - **0**: kernel mode
  - **1**: 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
- can 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

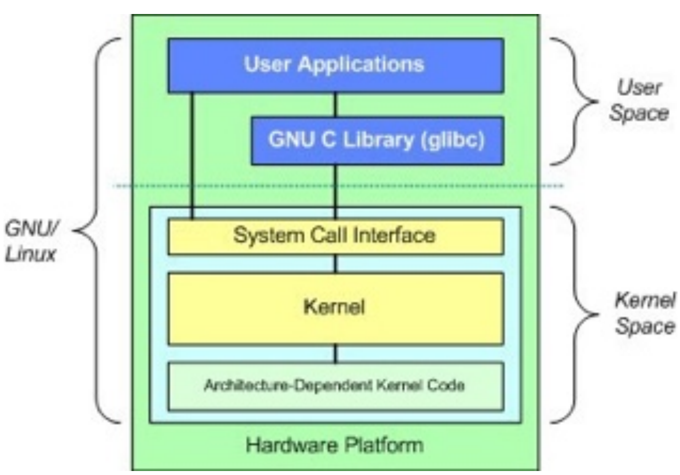
Goals for Protection and Fairness

- **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
- instructions that might affect goals are privileged and can only be executed by *trusted code*

Which Code is Trusted?

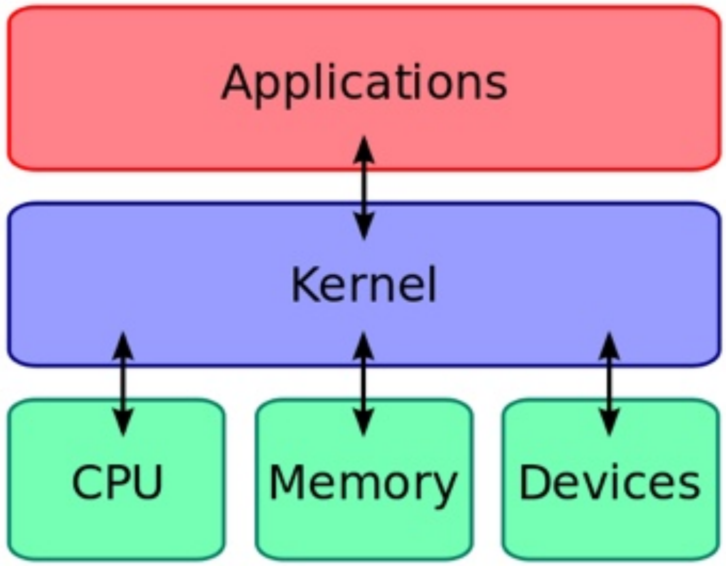
Kernel ONLY

- core of OS software executing in **supervisor** state
- **trusted software**:
    - manages hardware resources (CPU, memory, 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



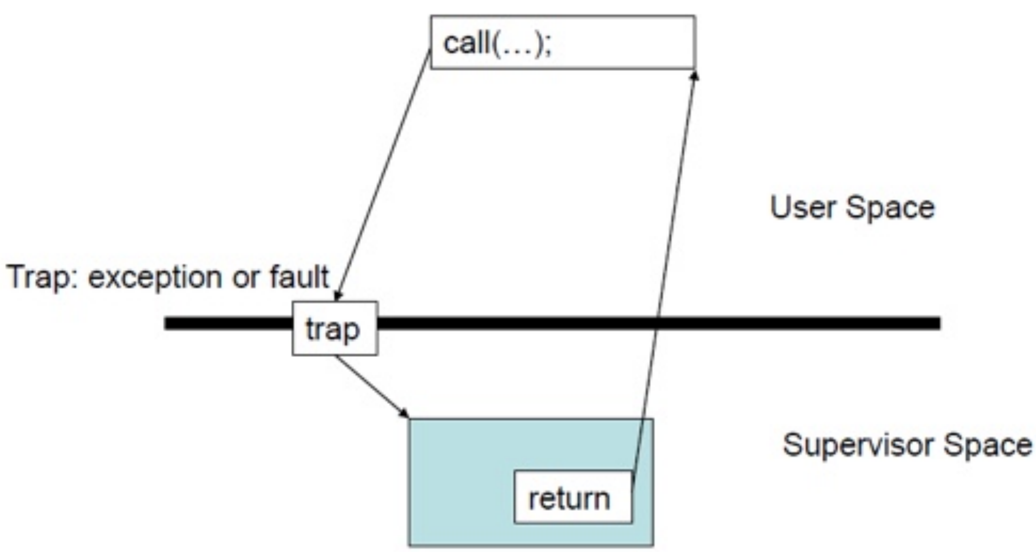
What About User Processes?

- 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 preforms the operation on behalf of the user
  - is the **only way** a user program can perform privileged operations
- when a system call is made, the program being executed is interrupted and control is passed to the kernel
- if operation is valid, then 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 operation
    - **OS performs requested action**
    - OS restores saved context, switches to user mode
    - OS gives control of the CPU back to user process

Example System Calls

```
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 filedes, 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

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

int fstat(int filedes, struct stat *buf);
// Returns information about the file with the descriptor filedes into buff
```

```
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.
- these functions perform privileged operations
    - they make system calls

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
- library functions make fewer system calls
    - non-frequent switches from user mode to kernel mode ➤ less overhead

Lab 7

- Write `tr2b` and `tr2u` programs in 'C' that transliterates bytes. They take two arguments 'from' and 'to'. The programs will transliterate every byte in 'from' to corresponding byte in 'to'
  - `./tr2b 'abcd' 'wxyz' < bigfile.txt`
    - Replace 'a' with 'w', 'b' with 'x', etc
  - `./tr2b 'mno' 'pqr' < bigfile.txt`
- `tr2b` uses **getchar** and **putchar** to read from STDIN and write to STDOUT.
- `tr2u` uses **read** and **write** to read and write each byte, instead of using getchar and putchar. The nbyte argument should be 1 so it reads/writes a single byte at a time.
- Test it on a big file with 5000000 bytes

  `$ head --bytes=# /dev/urandom > output.txt`

`time` and `strace`

```
time [options] command [arguments...]
// Output
// - real 0m4.866s: elapsed time as read from a wall clock
// - user 0m0.001s: the CPU time used by your process
// - sys 0m0.021s: the CPU time used by the system on behalf of your process

strace
// intercepts and prints out system calls to stderr or to an output file
// $ strace –o strace_output ./tr2b ‘AB’ ‘XY’ < input.txt
// $ strace –o strace_output2 ./tr2u ‘AB’ ‘XY’ < input.txt
```