

# OSN Tutorial 1

---

# Kernel Mode and User mode


---

- A processor in a computer running an OS can be thought of having two different modes: user mode and kernel mode. The processor switches between the two modes depending on what type of code is running on the processor. Applications run in user mode, and core operating system components run in kernel mode.
- Initially on boot the processor starts in kernel mode, most ISA have special instructions that can only be run in kernel mode. Like setting some hardware specific registers. The kernel then runs user programs in user mode
- . • This allows kernel control over the hardware but we have to allow user programs to use this hardware safely without disturbing other user programs.

# Interrupts and ISR

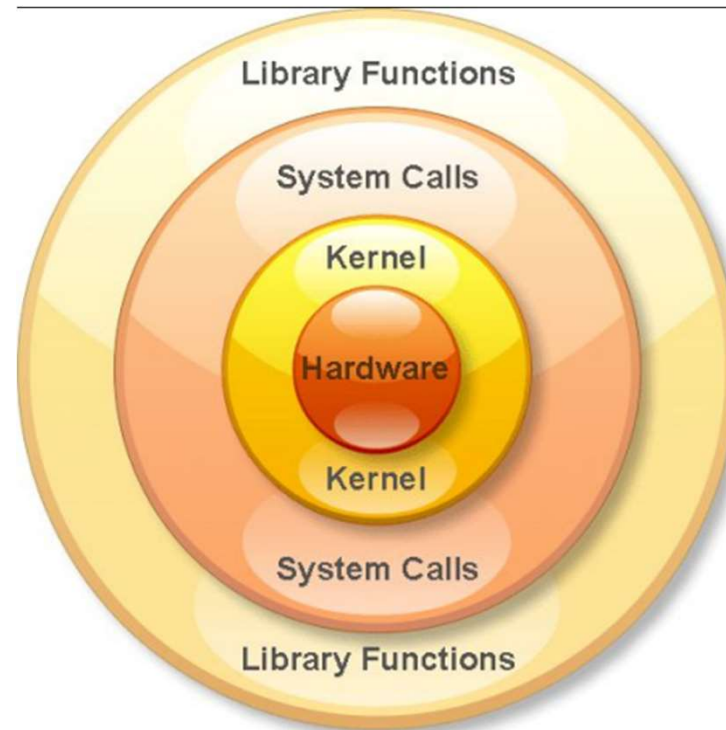
---


- User programs ask operating system to do things for them, how does the operating systems know some user program is asking stuff.
- The user program would call the operating system.
- But a call instruction won't work as we have to switch modes, a user program can't just go into kernel mode on a whim.
- We also cannot have the OS poll for events from time to time(Wasteful).
- We need to interrupt the CPU from executing user code and go to the kernel.
- We need hardware support.

- 
- The way we do this is using hardware interrupts. You might have seen interrupts in arduino programming.
  - Similar to arduinos, most CPUs have a INT pin that when sent a signal can be used to run routines in kernel mode.
  - For legacy intel architectures this translates to the INT 0x80 instruction
  - syscall instruction is the default way of entering kernel mode on x86-64.
  - The kernel sets the code to be run during these routines when it first boots up, These are called the interrupt handlers.
  - You can also pass arguments to interrupts via registers.
- 

# Syscalls

---



- 
- Syscalls are the mechanism used in modern operating systems to do this.
  - That means the code that gets executed during syscalls is part of the kernel.
  - In modern systems you will never do syscalls directly.
  - In case of C this is through glibc(The C standard library).
- 

---

Types of System Calls : There are many different categories of system calls –

1. Process control: end, abort, create, terminate, allocate and free memory.
2. File management: create, open, close, delete, read file etc.
3. Device management: read, device, write, get device attributes, release device, etc.
4. Information maintenance: system data, Get Process ID, set time or date, get time or date, set system data, etc.
5. Communication: CreatePipe() CreateFileMapping() MapViewOfFile()

# Write System Call

---

It writes data from a buffer declared by the user to a given device, such as a file. This is the primary way to output data from a program by directly using a system call

Write takes 3 arguments:

1. The file code (file descriptor fd)
2. The pointer to buffer where the data is stored (buf)
3. The number of bytes to write from the buffer

Ex:-

```
ssize_t write(int fd, const void *buf, size_t nbytes);
```



Argument	Description
<b>fd</b>	It is the file descriptor which has been obtained from the call to open. It is an integer value. The values 0, 1, 2 can also be given, for standard input, standard output & standard error, respectively .
<b>buf</b>	It points to a character array, with content to be written to the file pointed to by fd.
<b>nbytes</b>	It specifies the number of bytes to be written from the character array into the file pointed to by fd.

# Open system call

---

A program initializes access to a file in a file system using the open system call.

This allocates resources associated to the file (the file descriptor), and returns a handle that the process will use to refer to that file. In some cases the open is performed by the first access

Operations on the descriptors such as moving the file pointer or closing it are independent—they do not affect other descriptors for the same file.

If the file is expected to exist and it does, the file access, as restricted by permission flags within the file meta data or access control list, is validated against the requested type of operations.

Arguments:

1. The pathname to the file,
2. The kind of access requested on the file (read, write, append etc.),
3. The initial file permission is requested using the third argument called mode. This argument is relevant only when a new file is being created.

Ex:-

```
int open(const char *path, int oflag, .../*,mode_t mode */);
```

---

The value returned is a file descriptor which is a reference to a process specific structure which contains, among other things, a position pointer that indicates which place in the file will be acted upon by the next operation.

Open may return `-1` indicating a failure with `errno` detailing the error.

# lseek system call

---

lseek (C System Call): lseek is a system call that is used to change the location of the read/write pointer of a file descriptor. The location can be set either in absolute or relative terms.

int fildes : The file descriptor of the pointer that is going to be moved

off\_t offset : The offset of the pointer (measured in bytes).

int whence : The method in which the offset is to be interpreted (relative, absolute, etc.). Legal values for this variable are provided at the end.

return value : Returns the offset of the pointer (in bytes) from the beginning of the file. If the return value is -1, then there was an error moving the pointer.

Ex:-

```
lseek (f_write, n, SEEK_CUR);
```

# read system call

---

`read()` attempts to read up to `count` bytes from file descriptor `fd` into the buffer starting at `buf`.

If `count` is zero, `read()` returns zero and has no other results. If `count` is greater than `SSIZE_MAX`, the result is unspecified.

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number.

On error, `-1` is returned, and `errno` is set appropriately. In this case it is left unspecified whether the file position (if any) changes.

Ex:-

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

# write system call

---

write() writes up to count bytes to the file referenced by the file descriptor fd from the buffer starting at buf. POSIX requires that a read() which can be proved to occur after a write() has returned returns the new data.

Ex:-

```
ssize_t write(int fd, const void *buf, size_t count);
```

On success, the number of bytes written are returned (zero indicates nothing was written). On error, -1 is returned, and errno is set appropriately.

If count is zero and the file descriptor refers to a regular file, 0 may be returned, or an error could be detected. For a special file, the results are not portable