OSN Tutorial 1

INTRO TO SYSTEM CALLS

DUAL MODE IN OPERATING SYSTEMS

- A processor running an OS can be thought of having two different modes :
  1. Kernel Mode
  2. User Mode
- The difference in the two modes is the level of access that the processor has to functions like accessing hardware.
- User Applications run in User Mode while core Operating System components run in kernel mode.
- On boot, the processor runs in kernel mode and loads special instructions which are restricted in user mode.

INTERRUPTS AND ISR

- User programs can ask the OS to do things for them rather than accessing the restricted functionality directly.
- The user program needs to call the operating system. However, a call instruction won't work as we have to switch modes. (User program can't be allowed to go into Kernel Mode on a whim).
- We also cannot have the OS poll for events since it's wasteful.
- Hence, the solution is to interrupt CPU from executing user code and then safely switch to Kernel Mode, hence, hardware support is required.
- Hardware interrupts are used to facilitate this.

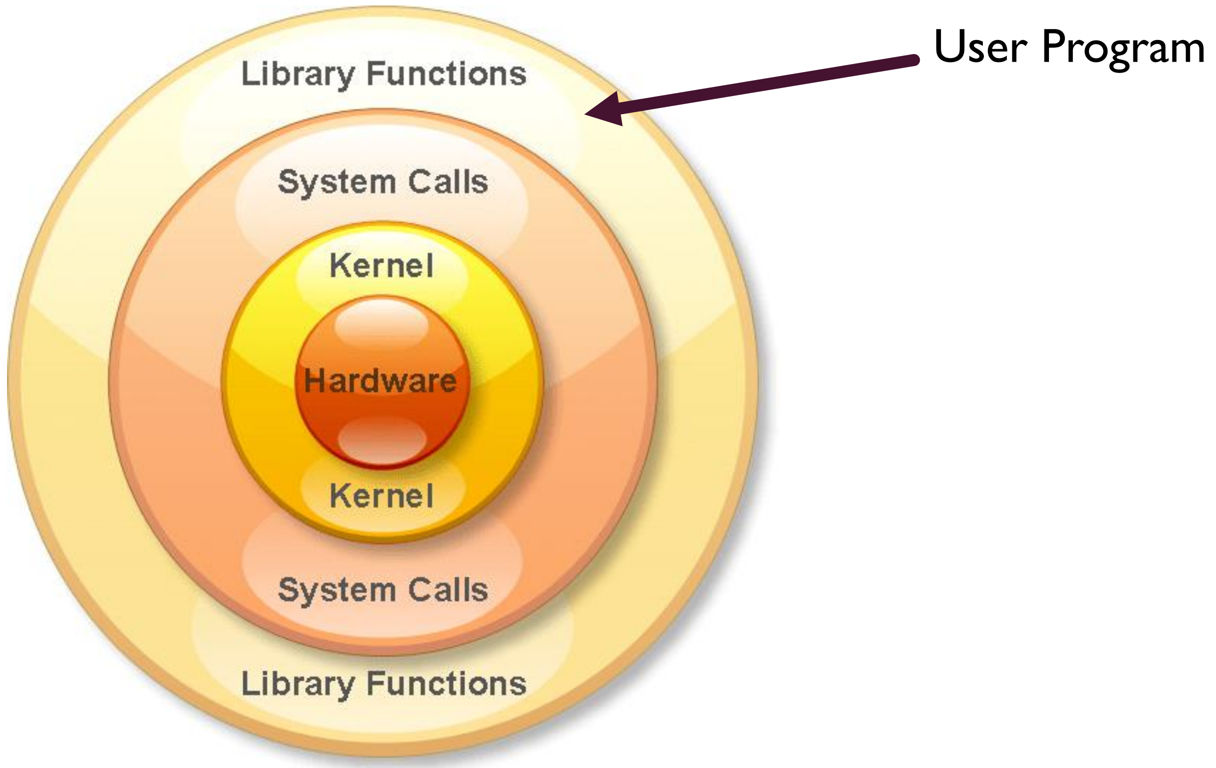
INTERRUPTS AND ISR

- There is a special INT pin in CPUs which can be used to execute routines in kernel mode when sent a signal.
- Syscall is the default way of entering kernel mode on x86-64 CPUs
- Kernel sets up the code to be run in the routines on boot up and are called ISR (Interrupt Service Routine).
- As the name suggests, these are routines invoked in kernel mode in response to an interrupt.
- Hence, the code that gets executed during syscalls is part of the kernel.

ANATOMY OF LINUX SYSTEM CALLS

The System Calls layer here ensures that no user program can access the kernel mode and subsequently the hardware as they like. Unrestricted access to Kernel (and hardware) can cause harm to the system.

In modern systems, syscalls aren't made directly but usually through Library functions. In case of C, this is glibc(The C standard library).



TYPES OF SYSTEM CALLS (SYSCALLS)

Syscalls can be categorized in five categories as follows :

1. Process Control
2. File Management
3. Device Management
4. Information Maintenance
5. Communication

For Mini Project 0, you will be working with the first 2, Process Control and File Management.

FILE MANAGEMENT SYSTEM CALLS

These system calls are used to handle files which includes creating files, deleting files, opening files, reading files etc. We will focus on the following system calls for the sake of this Tutorial :

1. open
2. read
3. write
4. close
5. lseek

OPEN SYSTEM CALL

As the name suggests, the open system call is used initialize access to a file in the file system. This allocates resources associated to the file (the file descriptor) and returns a handler which can be used by the process to refer to that file.

A file can have multiple descriptors which are independent of each other for operations like moving or closing the file descriptor.

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. Pathname to the file
2. Flags : Kind of access requested (read, write, append etc.) and other flags for creation, status etc.
3. Mode (optional) : The initial file permission which is relevant only when a new file is being created.

Syntax :

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

WRITE SYSTEM CALL

Writes data from a buffer declared by the user to a given device, such as a file.

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

write() writes up to nbytes bytes to the file reference by the file descriptor fd from the buffer starting at buf. The OS ensures that any read() following a write() operation can access the new data (even if it is cached and not on disk yet).

On success, the number of bytes written are returned by write() while on error, -1is returned and errno is set appropriately. File position is advanced by this number on success.

The 3 arguments are :

1. fd : File Descriptor obtained after system call to open is made, it is a positive integer value. Value 0, 1 and 2 is for stdout, stdin and stderr by default and can be used as well.
2. buf : Points to a character array with content to be written to the file pointed by fd.
3. nbytes : Specifies **maximum** number of bytes to be written from character array to file.

READ SYSTEM CALL

Reads maximum nbytes from file specified with file descriptor fd into the buffer starting at buf.

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

If nbytes is 0, then read() returns 0 with no other effect. If it's greater than SSIZE_MAX (maximum size of object of type SSIZE_T), then result is unspecified.

On success, the number of bytes read is returned (zero indicated EOF), and file position is advanced by this number. On error, -1 is returned and errno is set accordingly.

The 3 arguments are :

1. fd : Explained before.
2. buf : Points to a character array in which content is stored after being read from file.
3. nbytes : Specifies **maximum** number of bytes to be read from file to character array.

LSEEK SYSTEM CALL

lseek is a system call that can be used to change the location of the read/write pointer of a file descriptor. Location can be either set in absolute or relative terms.

```
off_t lseek(int fd, off_t offset, int whence);
```

It repositions the file offset of file associated with file descriptor fd to the argument offset according to directive whence as follows :

1. SEEK_SET : File offset is set to offset bytes (from start of file).
2. SEEK_CUR : File offset is set to its current location plus offset bytes.
3. SEEK_END : File offset is set to size of file plus offset bytes (offset can be –ve).

On success, lseek returns the resulting offset location as measured in bytes from beginning of file. On error, -1 is returned and errno is set accordingly.

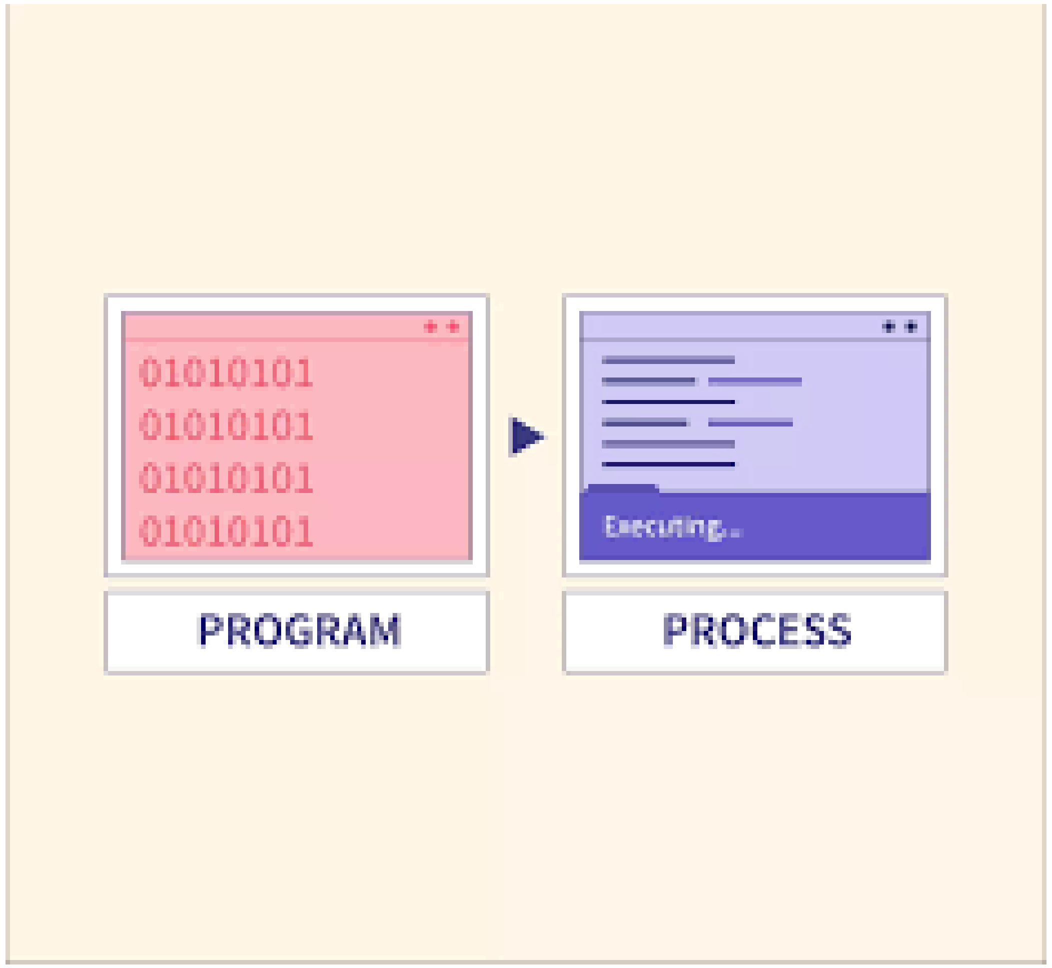
WHAT IS A PROCESS?

A process can be defined as an instance of a running program.

…

What is a program?

It is a set of instructions that are used to complete a specific task.



PROCESS ID & GETPID()

- Process ID (PID) is a unique id that is assigned to every process in the system.
- It can be any number lesser than 32768 (2^{15}) and can be increased to 4194304 (2^{22}) on 64-bit systems.
- getpid() function can be called which returns the process ID of the calling process.
- Process management refers to the activities involved in managing the execution of multiple processes in an operating system.

PROCESS MANAGEMENT

Process Management includes creating, scheduling and terminating processes.

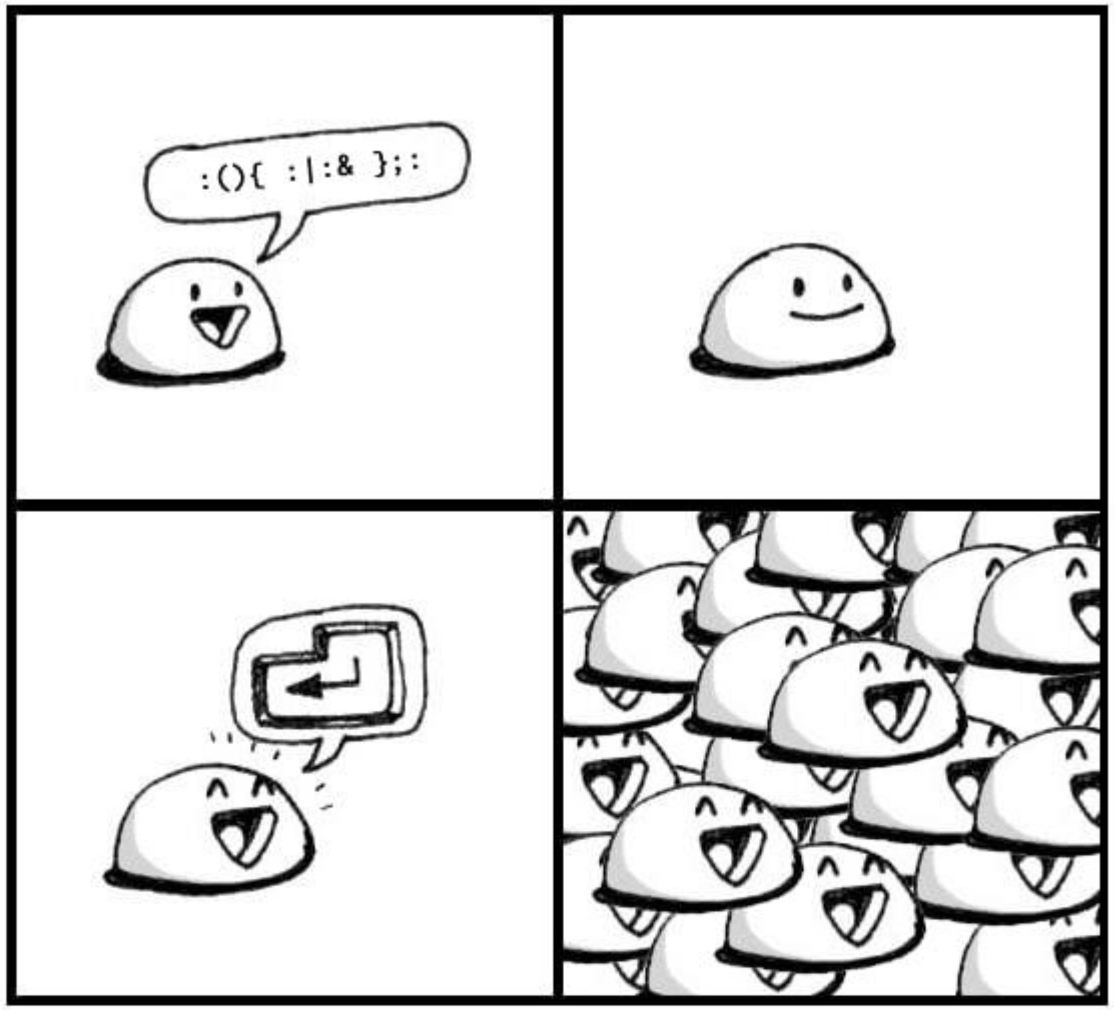
A process that is not needed or has gone haywire might need to be terminated by the OS.

For the sake of Mini Project 0, we will be working with process creation only.

A process can be created using the fork() command in LINUX.

FORK() AND SPAWNING PROCESSES

- fork() system call can be made to spawn a new process.
- This new process is called the **child** of the process which made the system call.
- The original process making the system call is called the **parent process.**
- The child process is an exact clone of the parent process and both of them execute the next instruction after fork()



SO HOW DOES THIS HELP?

- The child process is a clone of the parent process with the exception of the return value of fork().
- This return value can be used to distinguish between the parent process and the child process.

So, what does it return?

- Negative value : error while creating child process
- Zero : Returned to the newly created child process (if successful)
- Positive value : Returned to the parent process (if successful)

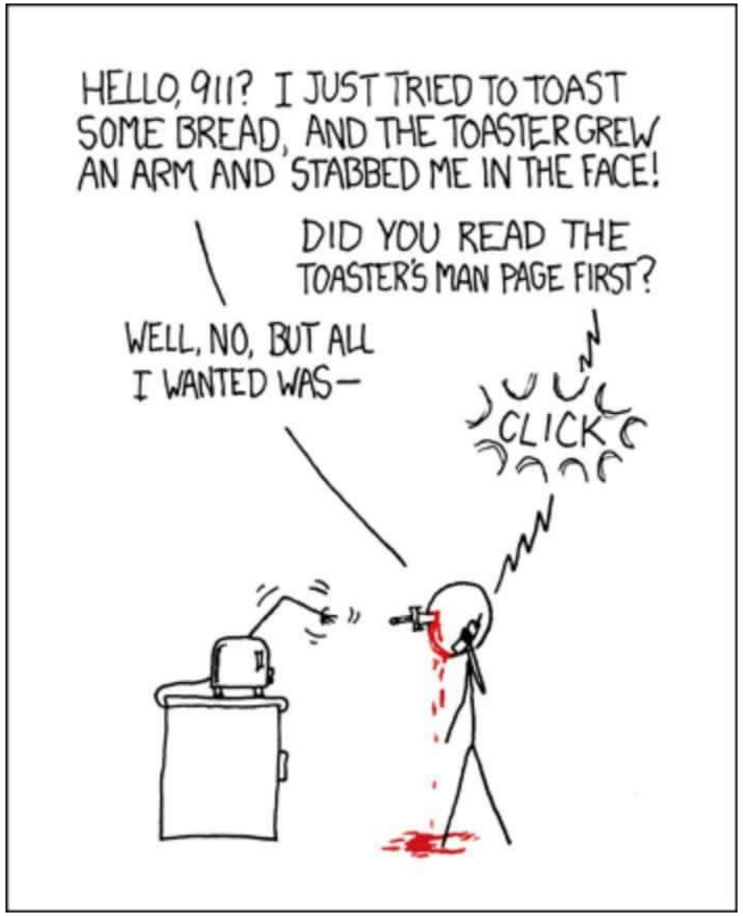
An interesting thing is that the positive value returned to parent process is the process ID of the newly created child process.

SOME GENERAL ADVICE

Read MAN pages.

Write modular code.

Start on time.


HELLO, 911? I JUST TRIED TO TOAST SOME BREAD, AND THE TOASTER GREW AN ARM AND STABBED ME IN THE FACE!
DID YOU READ THE TOASTER'S MAN PAGE FIRST?
WELL, NO, BUT ALL I WANTED WAS—
CLICK

Any Doubts?