Lecture 06: Process Creation

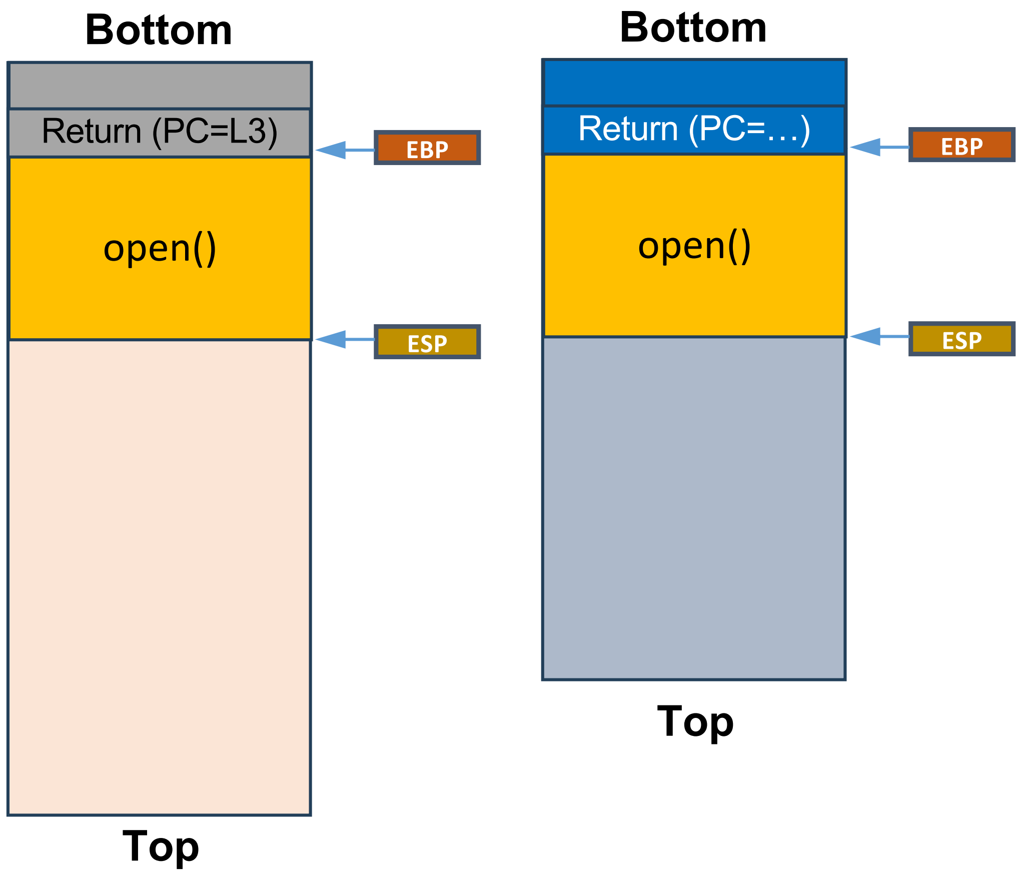
Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Today's Class

- Unix architecture
- System calls
- Process creation and termination

Return Address on Stack can be Modified

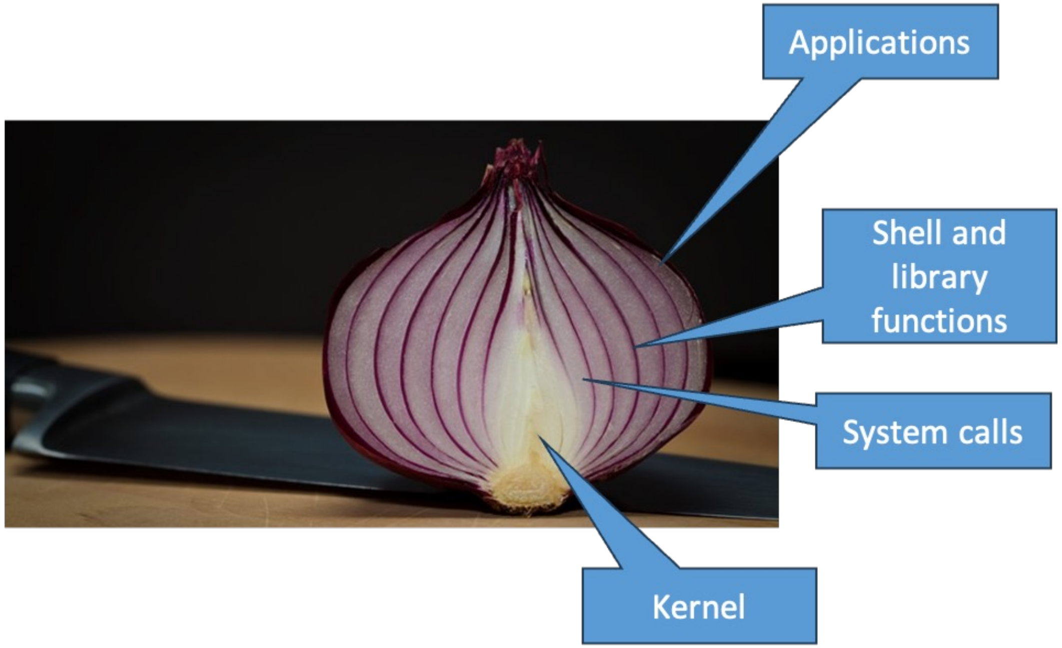
L1: main () {
    <save_regs>
    <save(PC)>
L2: open();
    <restore(PC)>
    <restore_regs>
L3: }



- Issues

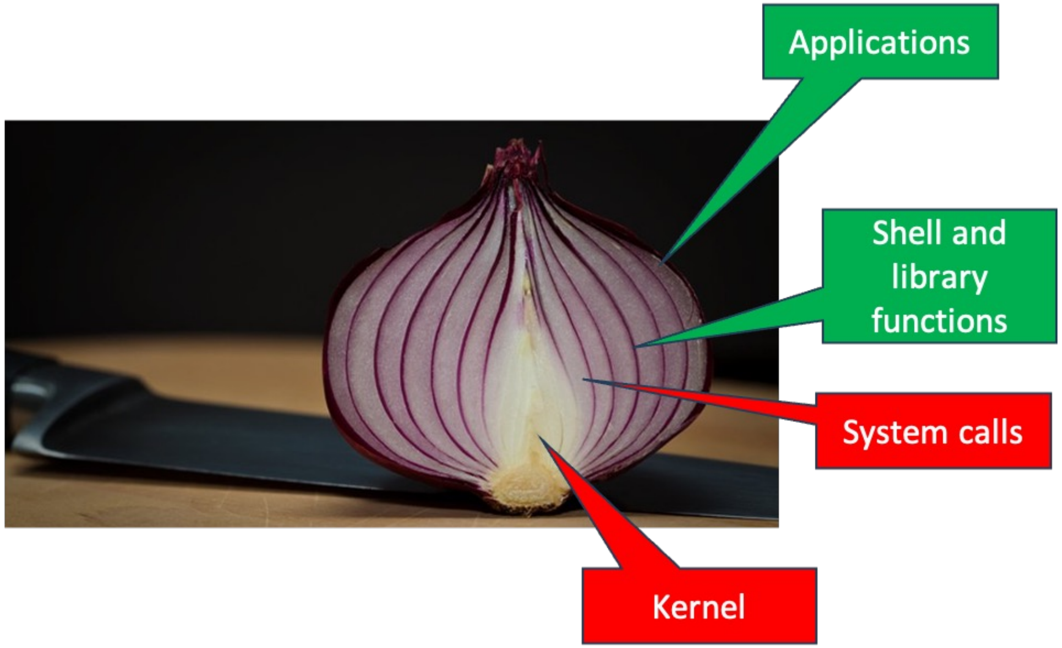
1. If **open**() is a regular method call then it's machine code won't be able to have a foolproof check if the calling process has the permission to access this file
   - The loader can simply patch the machine code with the call to open() at the runtime
2. The main() called **open**(), but then **open** could be **forced** to return to some malicious process call stack

Unix Architecture



- A major goal of OS is to support portability across various architecture
- A layered approach helps in achieving this goal as the innermost layer is only one that has to interact with the hardware
  - This innermost layer is called as OS **kernel**
  - It is relatively small, controls the hardware resources, and provides an environment under which programs can run
- System call is the interface to the kernel (e.g., read, open, etc.)
- Library functions are are built on top of system calls that applications can use
  - E.g., C-library function malloc uses sbrk system call, open(), etc.
- Shell is a command line interpreter that reads user input and execute commands
  - E.g., bash, csh, etc.

Protection Rings



- Protection rings are hardware supported mechanisms used by the OS for protection of data and functionality
  - E.g., x86 supports four protection levels (0-3)
  - Level-0 is called as **kernel mode** (supervisor mode) and Level-3 is **user mode**
- Unix-like OS execute user applications, shell, and library functions in the user mode whereas kernel/syscalls in the kernel mode

Today's Class

- Unix architecture
- System calls
- Process creation and termination

Interrupts: Gateway to the Kernel Mode

- Interrupts are signals to the CPU that something special has to happen
  - INT instruction on x86 to generate an interrupt
  - Interrupts are handled only by the kernel
- Three kinds of interrupts
  - Exceptions
    - Attempt to access invalid memory address, divide by zero, etc.
      - Covered in next lecture
  - **System calls (software interrupt)** – today's focus
  - Hardware interrupt
    - Signal generated by some hardware device. E.g., disk can generate an interrupt when a block of memory has been read and is ready to be retrieved
- As per Unix terminology, "interrupts" refer to the hardware interrupts, whereas "trap" refers to software interrupts (e.g., some special type of exceptions and all **system calls**)

libc Syscall Wrapper v/s Direct Syscall

- Code portability across different OS versions
- Easy to use APIs simplify coding experience as compared to using raw system call APIs
- Error handling support

Steps for Making a System Call (1/5)

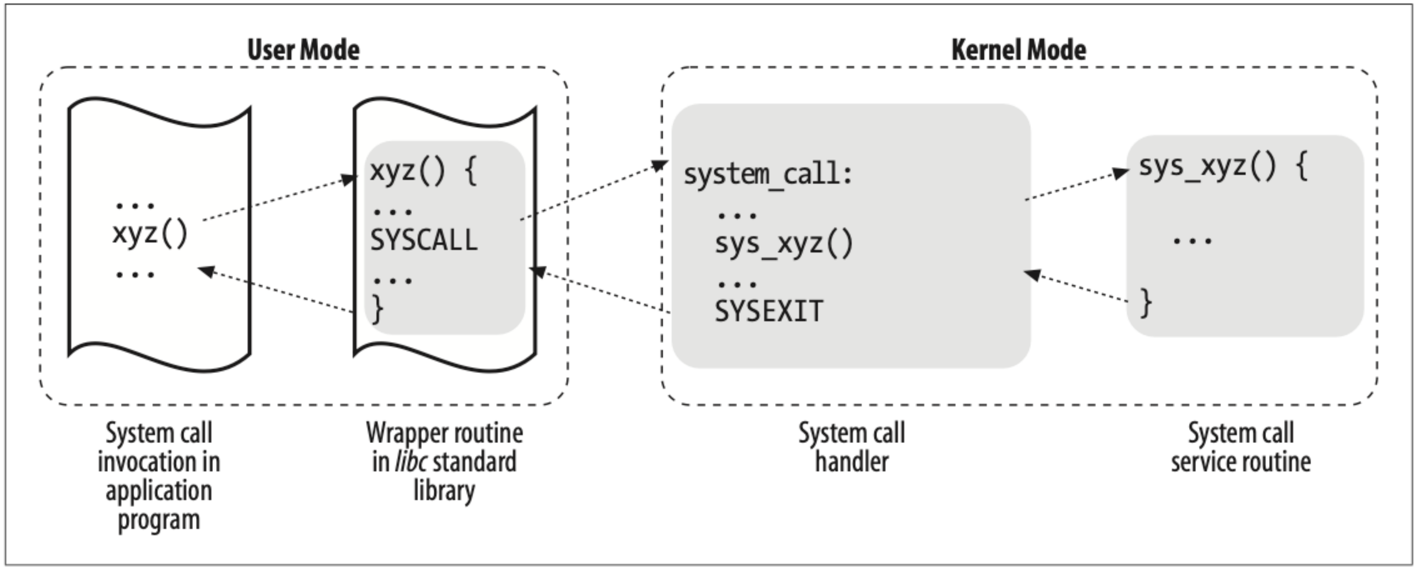
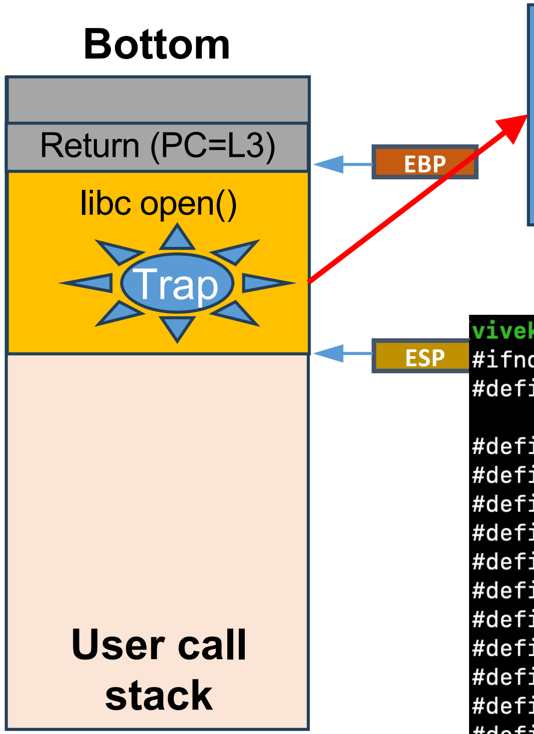


Figure 10-1. Invoking a system call

- **At a high level** the steps are as follows:
  1. Save registers in kernel mode stack
  2. Handle the system call by invoking a corresponding system call service routine
  3. Restore the registers and switch back into user mode

Figure source: from the book "Understanding the Linux Kernel"

Steps for Making a System Call (2/5)



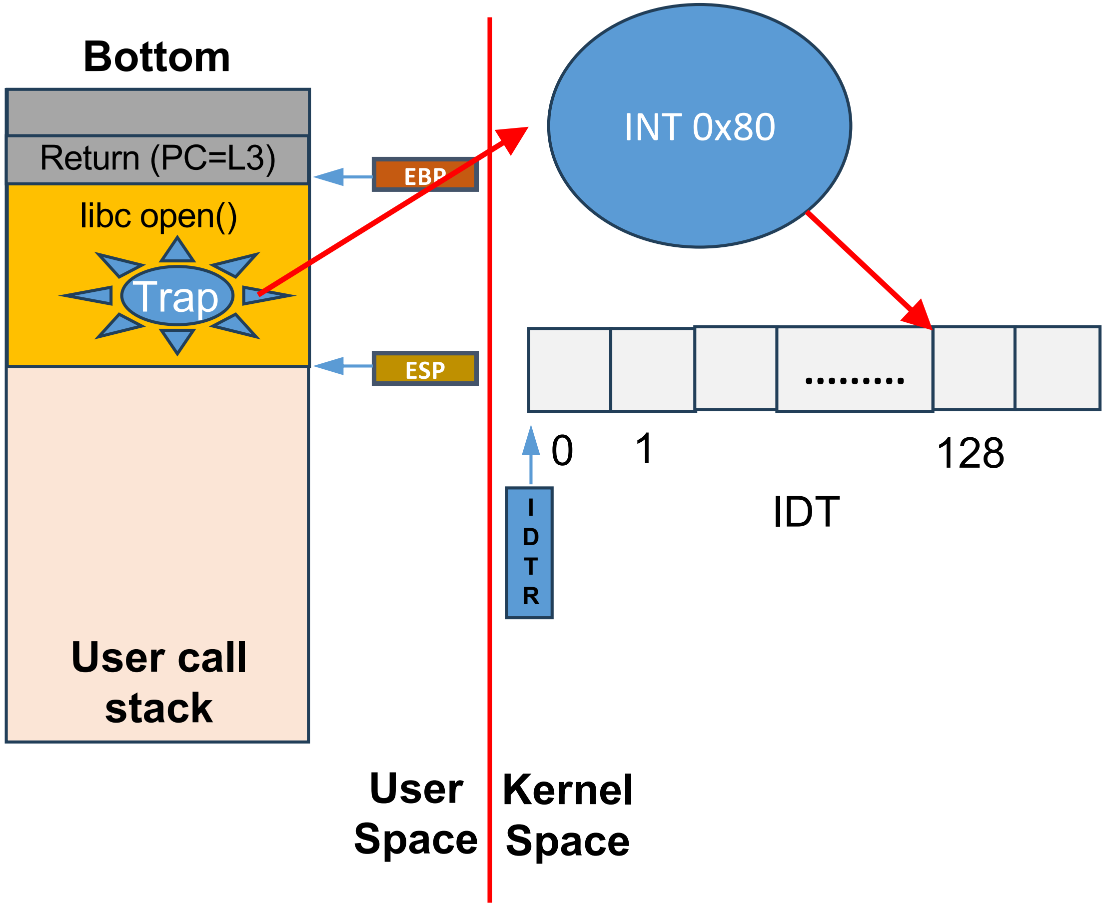
Find the unique number associated with the system call and store in EAX register

```
vivek@possum:~$ cat /usr/include/asm/unistd_32.h
#ifndef _ASM_UNISTD_32_H
#define _ASM_UNISTD_32_H

#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
#define __NR_waitpid 7
#define __NR_creat 8
#define __NR_link 9
#define __NR_unlink 10
#define __NR_execve 11
#define __NR_chdir 12
#define __NR_time 13
#define __NR_mknod 14
#define __NR_chmod 15
```

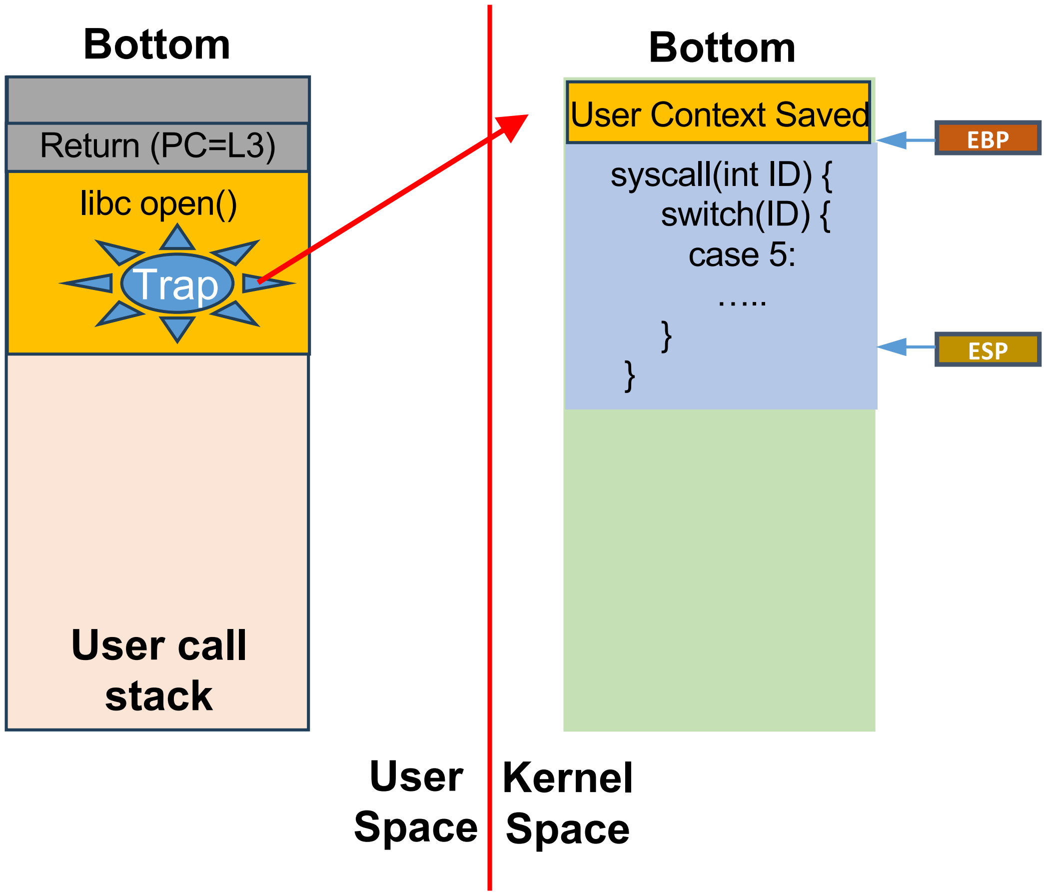
- Each system call has a unique number associated that is declared inside the file unistd.h
- Store this number in the EAX register
- Process is still executing inside user space

Steps for Making a System Call (3/5)



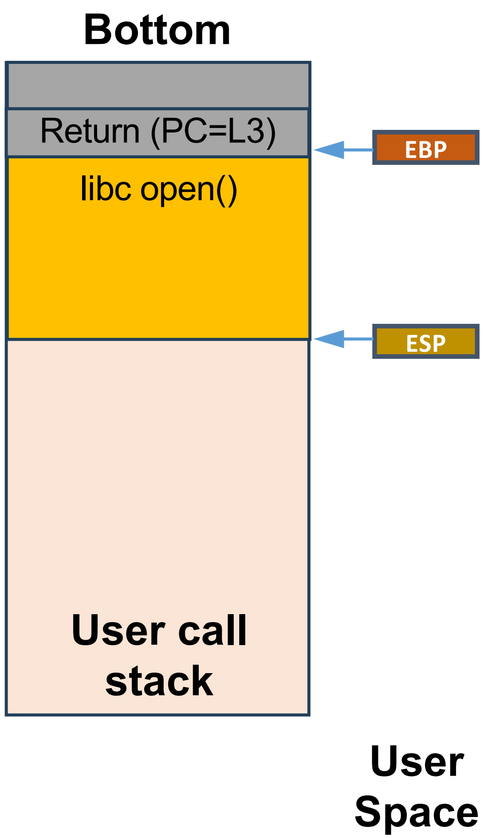
- Kernel sets up Interrupt Descriptor Table (IDT) at boot time whose address is stored in IDT register (IDTR). Total 256 entries in IDT on x86
- Each IDT index has the address of some interrupt handler
- 128th entry associated with the system call handler

Steps for Making a System Call (4/5)



- Switch to kernel stack
    - What should be the first step?
- Save registers on kernel stack. These registers are currently holding details to return back to user stack
- Look up the syscall number from EAX and process the syscall

Steps for Making a System Call (5/5)

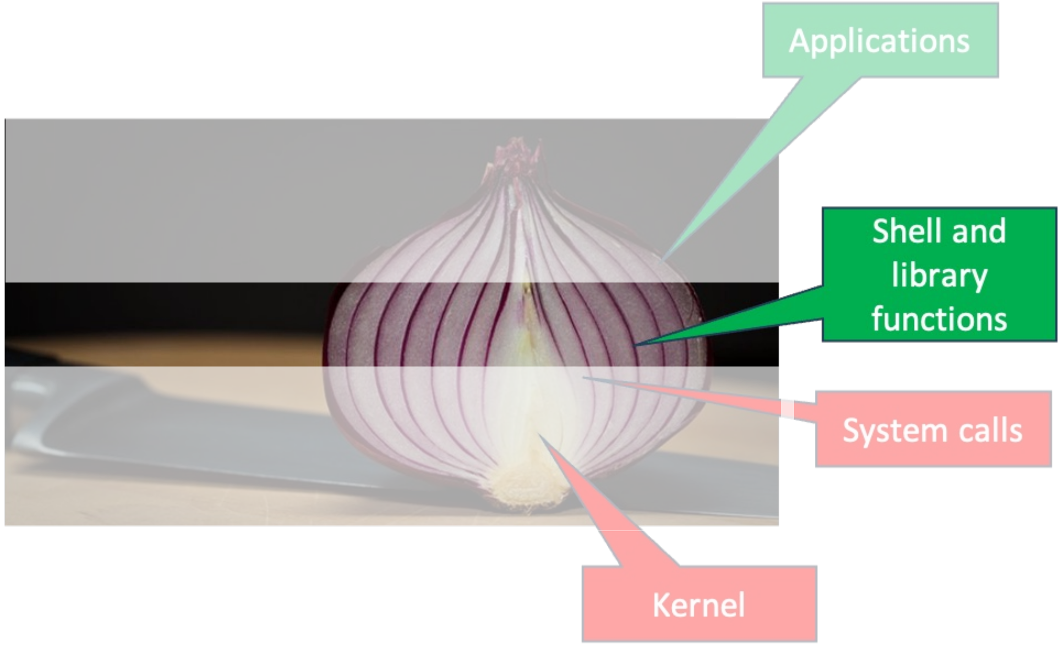


- For returning to user call stack, first pop the user registers from kernel stack and store them back into respective CPU registers
  - Return back to user call stack in the user mode
    - Protection level-3

Today's Class

- Unix architecture
- System calls
- Process creation and termination

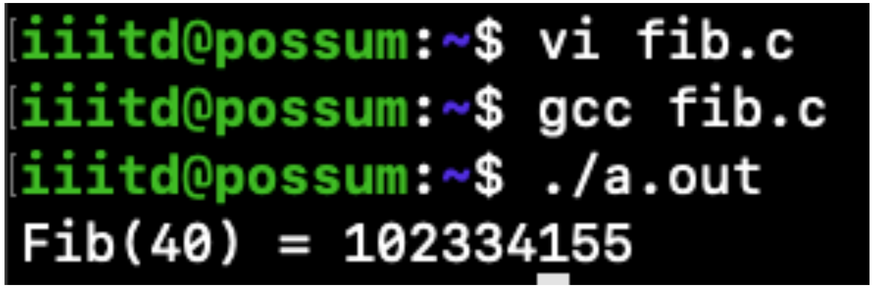
The Shell



- It is the first user process created by the OS after the bootup
- It runs in the user mode but it can create more processes by using system call
- Its main job is to execute user commands
  - Recall how we launched ./fib executable in previous lecture

Shell Pseudocode (1/2)

```
void shell_loop() {
    int status;
    do {
        printf("iiitd@possum:~$ ");
        char* command = read_user_input();
        status = launch(command);
    } while(status);
}
```



- Shell runs in an infinite loop and reads the user input to execute
- Should cease execution if it was unable to execute user command

Shell Pseudocode (2/2)

```
int launch (char *command) {
    int status;
    status = create_process_and_run(command);
    return status;
}
```

```
iiitd@possum:~$ vi fib.c
iiitd@possum:~$ gcc fib.c
iiitd@possum:~$ ./a.out
Fib(40) = 102334155
```

- The launch method accepts the user input (command name along with arguments to it)
- It will create a new process that would execute the user command and return execution status

A Process's Life Lessons

1. Processes can have children
2. Children should be obedient to their parent
3. Parent must follow the steps for good parenting
4. Children should not run their family business

Creating Child Processes (1/3)

```
int create_process_and_run(char* command) {
    int status = fork();
    if(status < 0) {
        printf("Something bad happened\n");
    } else if(status == 0) {
        printf("I am the child process\n");
    } else {
        printf("I am the parent Shell\n");
    }
    ....
    return 0;
}
```

- fork is a system call used for creating a new process
- Called once, but returns twice!
  - Return value in child process is zero, whereas child's process PID is returned in parent process
- It creates a replica of the parent process
  - Both the child and parent process are going to execute the same code with a minute difference
  - Copy-on-Write (COW) – Initially, both parent and child process have read-only access to parent's address space. Whichever process attempts a write on a memory page in parent's address space, it would get a copy of that page (lazy copy)
    - What about opened file descriptors?

Creating Child Processes (2/3)

```
int create_process_and_run(char* command) {
    int status = fork();
    if(status < 0) {
        printf("Something bad happened\n");
    } else if(status == 0) {
        printf("I am the child process\n");
    } else {
        printf("I am the parent Shell\n");
    }
    ....
    return 0;
}
```

- Which of the two printfs would get printed first?
- The output is non-deterministic as the OS can decide on its own which one of the child or parent process should be in the "running" queue
  - Imagine there is single CPU
  - Will be discussed in details in later lectures on process scheduling

Creating Child Processes (3/3)

```
int global=0;
int create_process_and_run(char* command) {
    int status = fork();
    if(status < 0) {
        printf("Something bad happened\n");
    } else if(status == 0) {
        printf("I am the child process\n");
        global++;
    } else {
        printf("I am the parent Shell\n");
        sleep(2)
    }
    printf("Global value = %d\n",global);
    ....
    return 0;
}
```

- What value of the global variable will be printed?
- Although, the child is replica of the parent process, it has its own address space (heap, call stack, etc.) and registers
  - "Replica" here means both child and parent will run the exact same executable a.out immediately after calling fork (unless child and parent path are made separate as shown – if statement)
- Although we have made the parent to sleep for 2 seconds, it is not guaranteed that this duration is adequate for the child to move into running queue and complete its execution
- Inter-process communication is required for the updated global value to be seen by the parent
  - Next lecture!

Next Lecture

- Process life lessons (contd.) and inter process communication