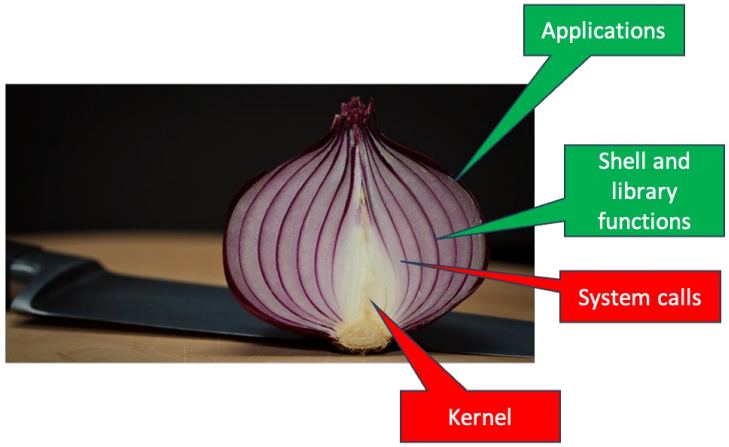
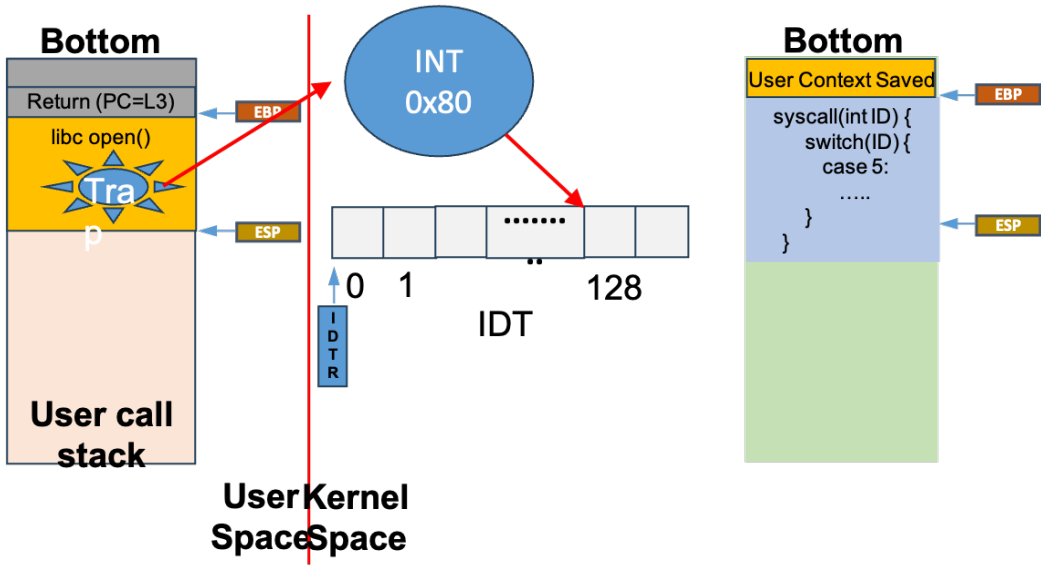
Lecture 07: Process Creation and Termination

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

Last Lecture



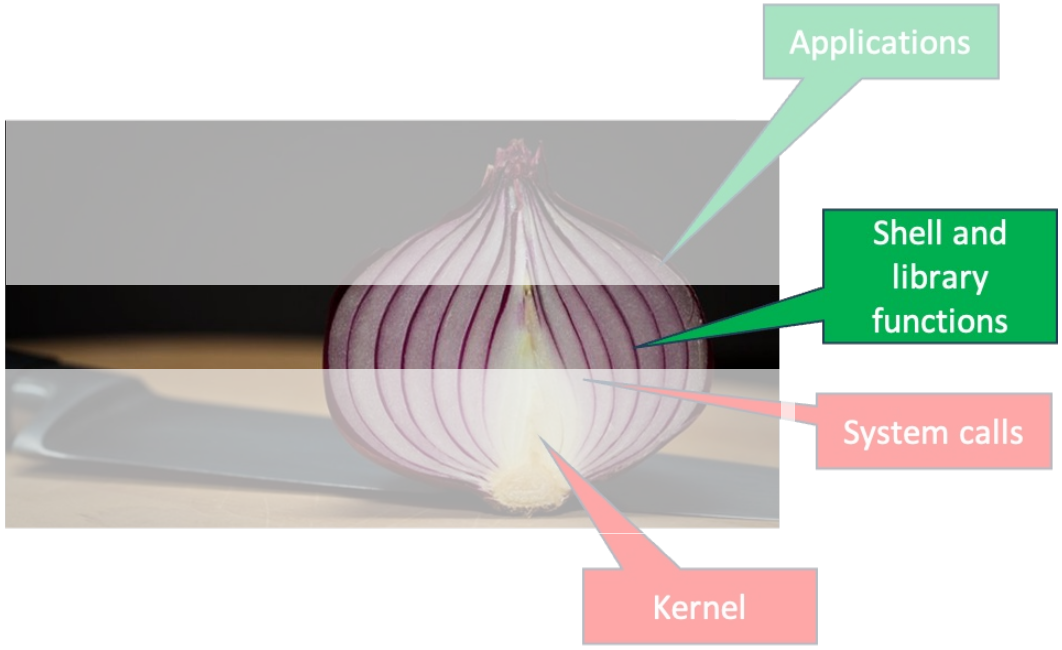


- Protection rings (kernel and User mode) in Unix-like OS
- Interrupts and system call

Today's Class

- Process's life lessons

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);
}
```

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

- 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!

A Process's Life Lessons (contd.)

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

The Obedient Child

```
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");
        exit(0);
    } else {
        printf("I am the parent Shell\n");
    }
    ....
    return 0;
}
```

- **exit** syscall allows to send a specific termination code (exit status) from a child process to the parent upon termination
  - "signal" is sent to the parent (inter-process communication)
  - In case of abnormal termination of child, the exit status is generated and send by the kernel
- exit carries out process cleanup – reclaiming memory, flushes buffers, closing fds, etc.
- But how the parent can get the exit status (next slide)?
  - Remember child is not going to return to the parent just like a callee method returns to a caller method

The Act of Good Parenting

```
int create_process_and_run(char* command) {
    int status = fork();
    if(status < 0) {
        printf("Something bad happened\n");
        exit(0);
    } else if(status == 0) {
        printf("I am the child (%d)\n",getpid());
    } else {
        int ret;
        int pid = wait(&ret);
        if(WIFEXITED(ret)) {
          printf("%d Exit =%d\n",pid,WEXITSTATUS(ret));
        } else {
          printf("Abnormal termination of %d\n",pid);
        }
        printf("I am the parent Shell\n");
    }
    return 0;
}
```

- **wait** and **waitpid** allows the parent process to block until the child process terminates
  - **wait** will block only for the first child, whereas **waitpid** can be used for a specific child
  - Returns the child's PID
  - Used for retrieving exit status from child
- Will there be deterministic execution of printfs from parent and child processes (notice there is no sleep)?
- Good parents avoid making their child as **Zombies or Orphan**
  - Child is zombie when it has terminated but has its exit code remaining in the process table as it is waiting for the parent to read the status
  - **Orphaned** children outliving their parent's lifetime are adopted by the mother-of-all-processes (**init**)

An Act of Kindness From a Bad Parent

```
int create_process_and_run(char* command) {
    int status = fork();
    if(status < 0) {
        printf("Something bad happened\n");
        exit(0);
    } else if(status == 0) {
        int status2 = fork();
        if(status2<0) printf("Kindness failed\n");
        else if (status2 == 0) {
           printf("Child will not live like Zombie\n");
        } else {
            _exit(0);
        }
    } else {
        printf("I am the parent Shell\n");
    }
    return 0;
}
```

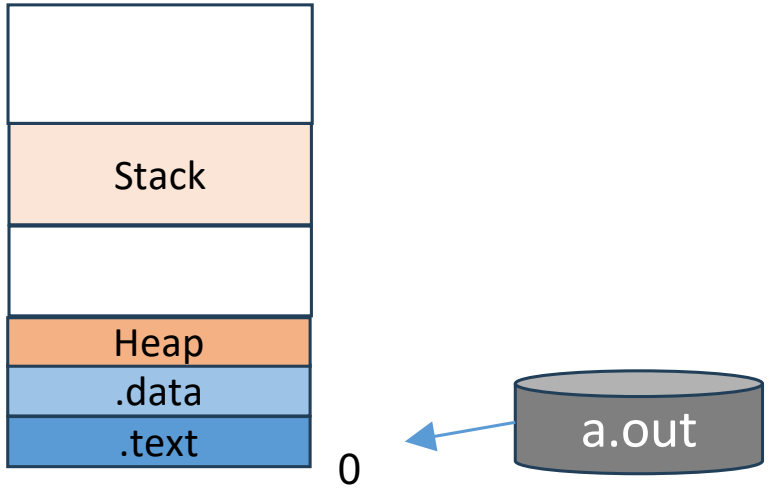
- There is a really bad parent who don't have a patience for good parenting
  - Get grandchildren and let them do the real work instead of the immediate children who will suffer a premature death
- The immediate child's only job is to get an offspring and die a quick death
  - The child should call **_exit** to ensure the offspring can inherit it's resources *(some kindness)*
- The mother-of-all process (**init**) will kindly adopt this process as her own child by issuing a **wait** call
  - No more Zombie!

Child Should Not Run Family Business

```
int create_process_and_run(char* command) {
    int status = fork();
    if(status < 0) {
        printf("Something bad happened\n");
        exit(0);
    } else if(status == 0) {
        printf("I am the child process\n");
        char* args[2] = {"./fib", "40"};
        execv(args[0], args);
        printf("I should never print\n");
    } else {
        printf("I am the parent Shell\n");
    }
    ....
    return 0;
}
```

- Main goal for creating a child process is to let it live its own free life without depending on its parent
  - The child won't let go off the parent's property (code path) until its forced to call exec
- An exec calls the OS loader internally that loads the ELF file with its command line argument as specified in the argument list
- There are seven different versions of exec which are collectively referred as exec function

exec Behind the Curtain



- It's only job is to construct the process's address space
  - Unload current process address space (segments)
  - Read ELF file from the disk
  - Create the user part of the address space lazily
    - E.g., space for .data will be allocated only after some global variable is accessed during program execution from the .text segment
- Note that PID remains the same after the process calls exec

Next Lecture

- Inter-process communication