Operating Systems

Lecture 3.2

Process Management (Part-II)

Instructor: Muhammad Arif Butt, PhD

Lecture Agenda

- Monitoring child processes using `wait()`
- Overwriting process address space using `exec()`
- How Linux Shell Execute a Command?
- Effect of `fork` and `exec` on process attributes
- Writing your own `system()` library function
- Overview of Background and Daemon processes
  - Job Control States
  - Intro to Daemon Processes
  - Job Scheduling using `cron` utility
  - Managing services using `systemctl` utility
- Writing your own daemon (programmatically)

Monitoring Child Process

wait() System call

```
pid_t wait(int *status)
```

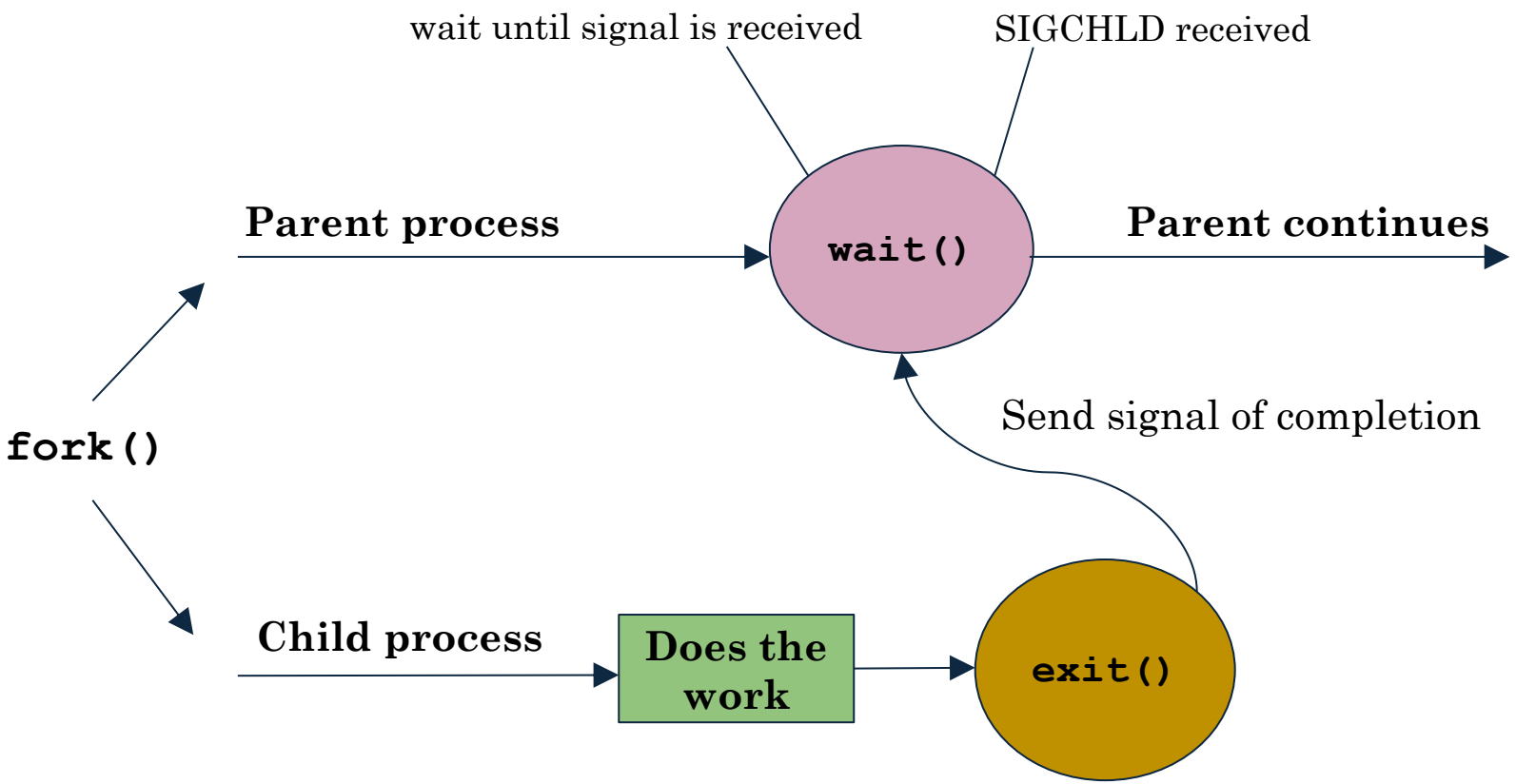
- The `wait()` system call is used for reaping and cleaning zombies from system, and serves two purposes
  - Notify parent that a child process finished running.
  - Tell the parent how a child process finished.
- The parent process calls the `wait()` system call and gets blocked till any one of its child terminates.
- The child process returns its termination status using the `exit()` call and that integer value is received by the parent inside the status argument.
- On the shell, we can check this value in the `$?` environment variable.
- On success, the `wait()` system call returns PID of the terminated child and in case of error returns a -1
- If a process wants to wait for termination of all its children, then

```
while(wait(null) > 0);
```

`wait()` System call

Two purposes of `wait()` system call:

- Notify parent that a child process finished running
- Tell the parent how a child process finished



Demonstration

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

`wait()` Status Argument

A process can end in four ways:

- **Normal Termination:** On successful completion of the task, programs call `exit(0)` or `return 0` from `main()` function. In case of failure, programs call `exit()` with a non-zero value (1-255). The exit codes should be documented in manual pages for proper error handling.
- **Terminated by a Signal:** A process can be terminated (killed) by signals such as `SIGKILL(9)`, `SIGTERM(15)`, or `SIGINT(2)`.
- **Stopped by a Signal:** A process receives `SIGSTOP(19)` or `SIGTSTP(20)` and suspends execution temporarily. The process remains in memory but doesn't consume CPU time Process state becomes "stopped" or "traced".
- **Continued by a Signal:** A process might get `SIGCHLD(17)`, `SIGCONT(18)` and resumes its execution.

All this information is encoded in the `status` argument of the `wait()` system call.
A programmer can decipher this information using bit operators or using available macros.

Decipher Status Argument (using bit-operators)

Normal Termination:

- If the lower eight bits contains zero that means normal termination, and the exit status is in the upper 8 bits.
- To check for normal exit: `(status & 0x7F) == 0`.
- To extract exit code: `(status >> 8) & 0xFF`.

Terminated by Signal:

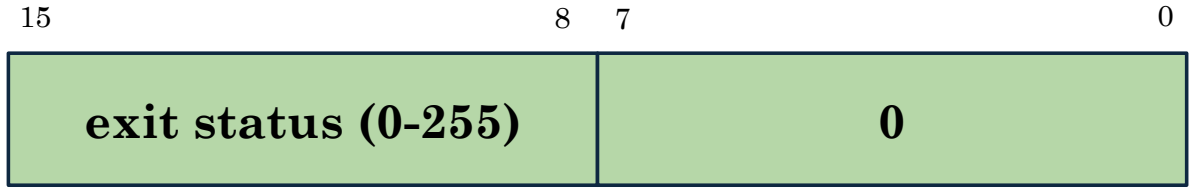
- If the lower eight bits do not contain zero that means process is terminated by signal.
- To check for normal exit: `(status & 0x7F) != 0`
- To extract signal number: `status & 0x7f`
- Core dump occurred if bit#7 is set: `status & 0x80`

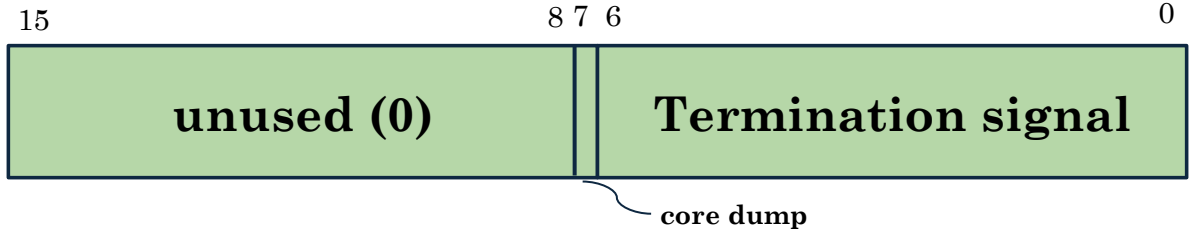
Stopped by Signal:

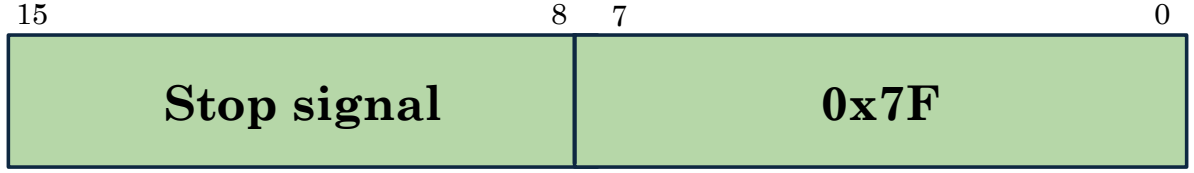
- If the lower eight bits are 0x7F, that means process is stopped by signal, and the upper 8 bits contains stop signal.
- Stop detection: `(status & 0xFF) == 0x7F`
- To extract signal number: `(status >> 8) & 0xFF`

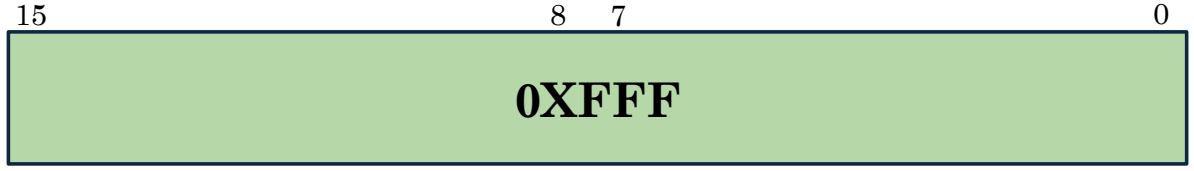
Continued by Signal:

- Continue detection: `status == 0xFFFF`
- This might work on some Linux systems, but is not reliable or portable. Better use the `WIFCONTINUED()` macro.







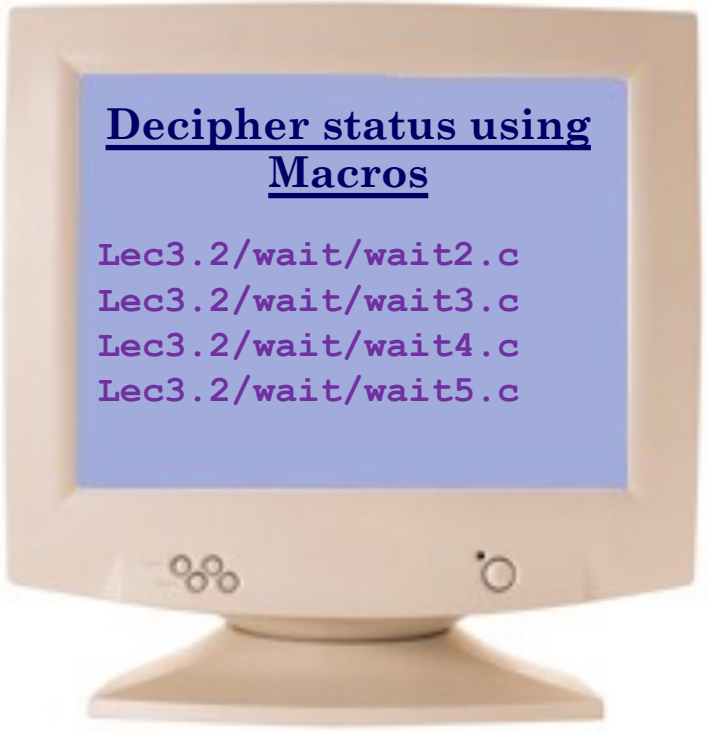


Decipher Status Argument (using Macros)

Instead of bit operators, we can use macros to decipher the status argument of `wait()`, defined in `./usr/include/x86_64-linux-gnu/bits/waitstatus.h`

WIFEXITED(status)	This macro returns true if child process exited normally `WEXITSTATUS (status)` returns exit status of the child process
WIFSIGNALED(status)	This macro returns true if child process is killed by a signal `WTERMSIG(status)` returns the number of signal that killed the process `WCOREDUMP(status)` returns a non-zero value if the child process created a core dump file
WIFSTOPPED(status)	This macro returns true if child process is stopped by a signal `WSTOPSIG(status)` returns the number of signal that stopped the process
WIFCONTINUED(status)	This macro returns true if child process was resumed by SIGCONT

Demonstration



GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Limitations of `wait()` System Call

Using `wait()`, it is not possible for parent to:

- **Detect why child was stopped:** Retrieve information about stopped children or the signal that stopped them (SIGSTOP, SIGTSTP, etc.)
- **Be notified of child continuation:** Detect when a stopped child resumes execution after receiving SIGCONT
- **Wait for specific child:** Wait for a particular child by PID; can only wait for the first available child that changes state
- **Perform non-blocking wait:** Check child status without blocking; always waits until a child terminates or changes state

waitpid() System call

```
pid_t waitpid(pid_t pid, int* status, int options);
```

- The `waitpid()` system call is used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed.
- The **pid argument** enables the selection of the child to be waited for:
  - If **`pid > 0`** : waits for the child whose PID equals the value of pid
  - If **`pid == -1`**: waits for any child

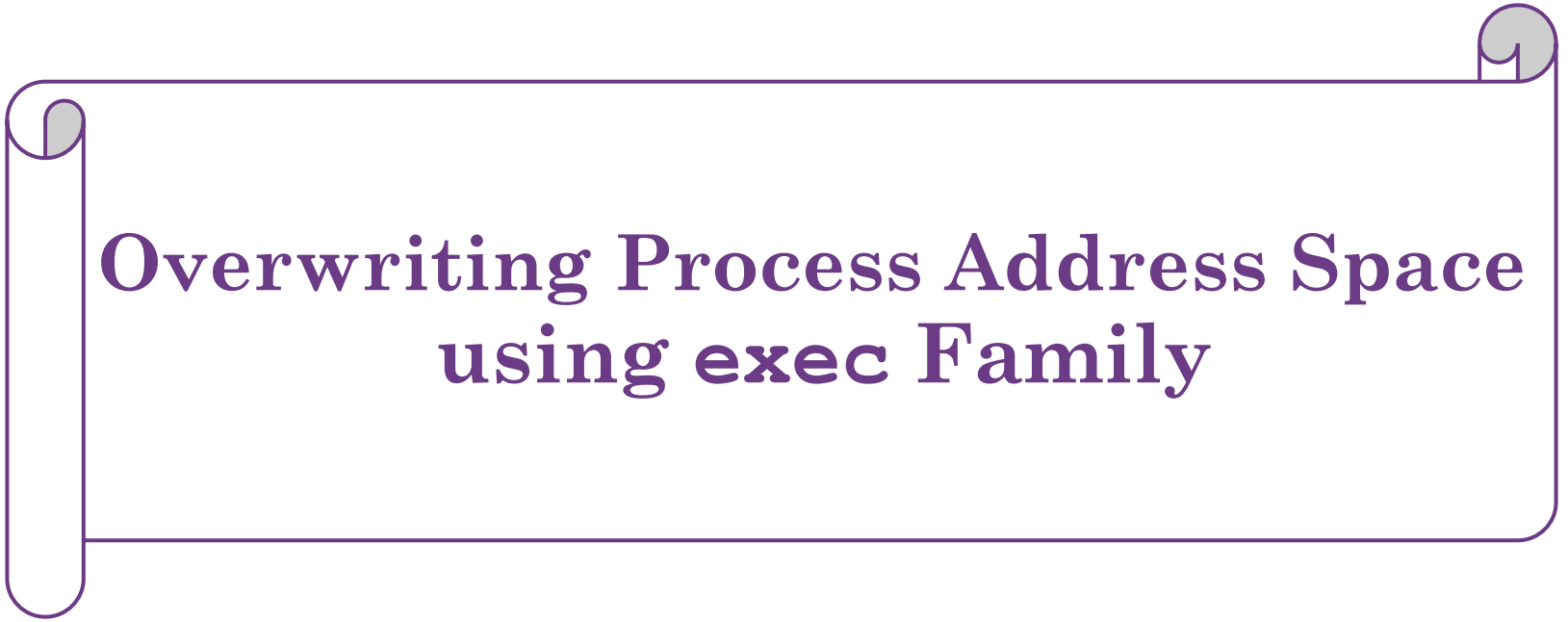
    `wait(&status) <=> waitpid(-1, &status, 0)`
  - If **`pid == 0`**: waits for any child process whose process Group ID is the same as the calling/parent process
  - If **`pid < -1`**: waits for any child process whose process Group ID equals the absolute value of pid argument

- The **third argument** of `waitpid()` call is a bit mask of zero or more of the following flags, defined in `/usr/include/wait.h` file:

WUNTRACED	Also returns information when a child is stopped by a signal
WCONTINUED	Also return information about stopped children that have been resumed by delivery of SIGCONT signal
WNOHANG	Performs polling. If no child specified by pid has yet changed state, then return immediately, instead of blocking

Demonstration

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*



Overwriting Process Address Space using exec Family

exec Family of Functions

- A process can replace its current program image with a new executable using the `exec()` family of functions
- This action overwrites the entire address space of the calling process, including code, data, heap, and stack with the new program
- The new program starts execution from its main function
- There are five library functions of `exec` family and all are layered on top of the `execve()` system call. Each of these functions provides a different interface to the same functionality
- There is no return after a successful `exec` call
- The `exec` functions return only if an error has occurred. The return value is -1, and `errno` is set to indicate the error

exec Family of Functions (cont...)

```
int execl (const char *pathname, const char* arg0,...,(char*)0);
int execlp (const char *filename, const char* arg0,...,(char*)0);
int execle (const char* pathname, const char* arg0,...,(char*)0,char* const envp[]);
```

- The first argument to this family of exec() calls, is the name of the executable, which on success will overwrite the address space of the calling process with a new program from the secondary storage
- The **l** after the exec means that command line arguments to the new program will be passed as a comma separated list of strings with a '\0' character at the end
- The **p** stands for path. It means that the program specified as the first argument should be searched in all directories listed in the PATH variable. However, using absolute path to program is more secure than relying on PATH variable, which can be more easily altered by malicious users
- The **e** stands for environment. It means that after the command line arguments, the program should pass an array of pointers to null terminated strings, specifying the new environment of the program to be executed. Otherwise, the caller environment will be used

exec Family of Functions (cont...)

```
int execv  (const char *pathname, char *const argv[]);
int execvp (const char *filename, char* const argv[]);
int execve (const char* pathname,char* const argv[], char* const envp[]);
```

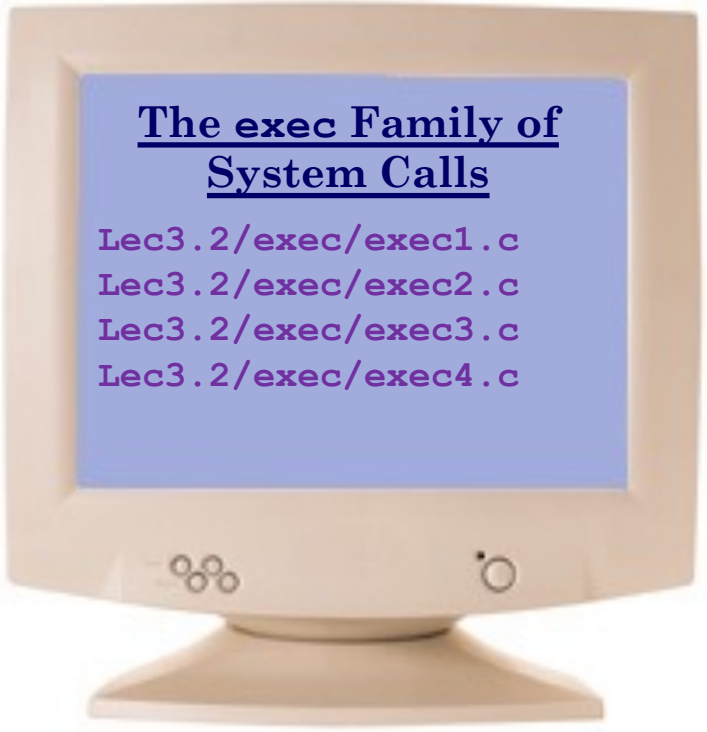
- The first argument to this family of exec() calls, is the name of the executable, which on success will overwrite the address space of the calling program with a new program
- The **v** after the exec means that command line arguments to these functions will be passed as an array of pointers to null terminated strings
- The **p** stands for path. It means that the program specified as the first argument should be searched in all directories listed in the PATH variable. However, using absolute path to program is more secure than relying on PATH variable, which can be more easily altered by malicious users
- The **e** stands for environment. It means that after the command line arguments, the program should pass an array of pointers to null terminated strings, specifying the new environment of the program to be executed. Otherwise, the caller environment will be used

exec Family of Functions (cont...)

- All `exec` functions do not return on success -they completely replace the current process image
- If an `exec` call does return, it always returns -1, indicating that an error occurred
- However, there's no need to explicitly check the return value, as the mere fact that control returned to the caller implies failure. In such cases, the global variable `errno` can be inspected to determine the specific error

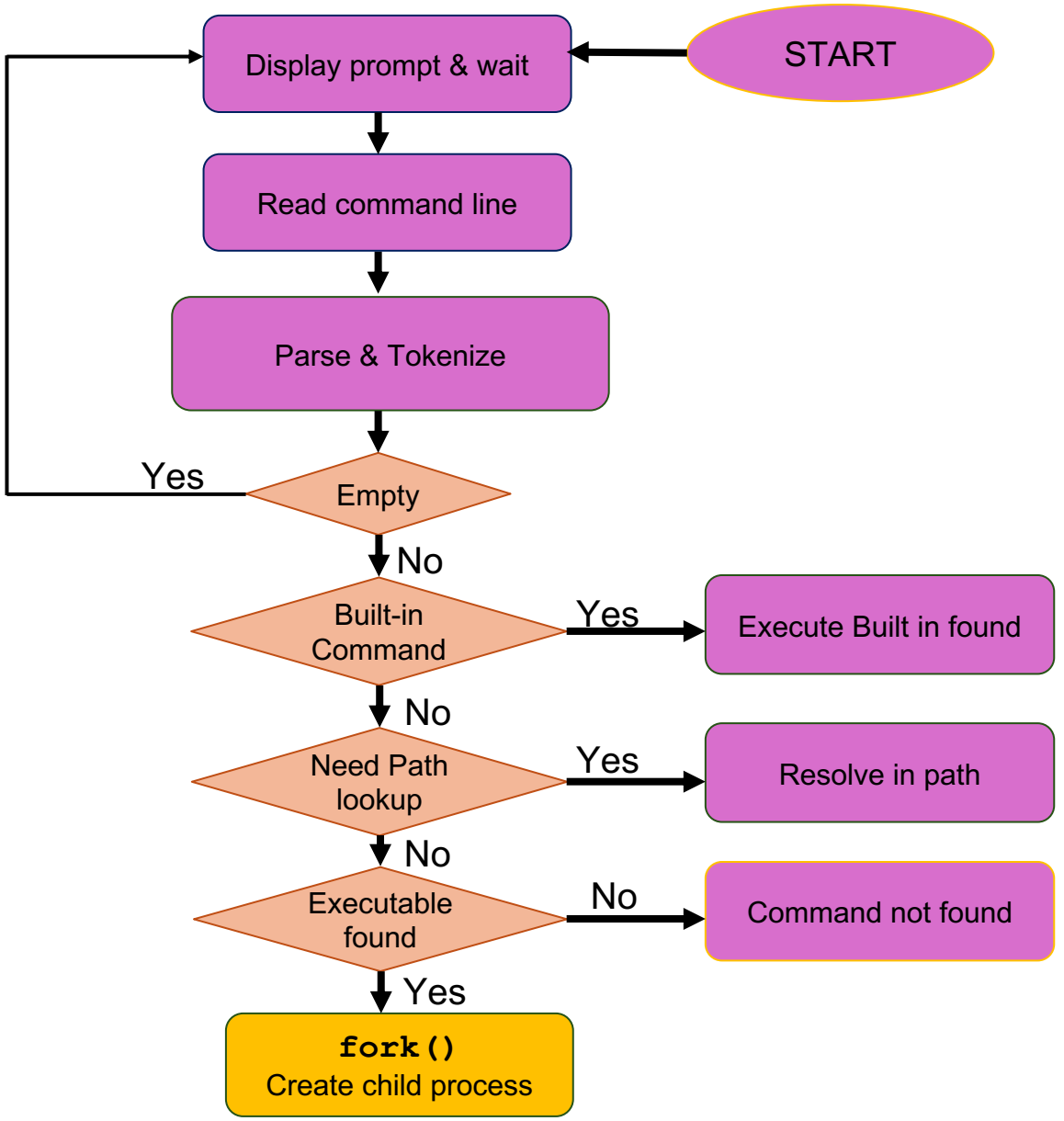
EACCES	The specified program is not a regular file, or doesn't have execute permissions enabled or one of the directory components of pathname is not searchable
ENOENT	The specified program does not exist
ENOEXEC	The specified program is not in a recognizable executable format
ETXTBSY	The specified program is open for writing by another process
E2BIG	The total space required by the argument list & environment list exceeds the allowed maximum

Demonstration

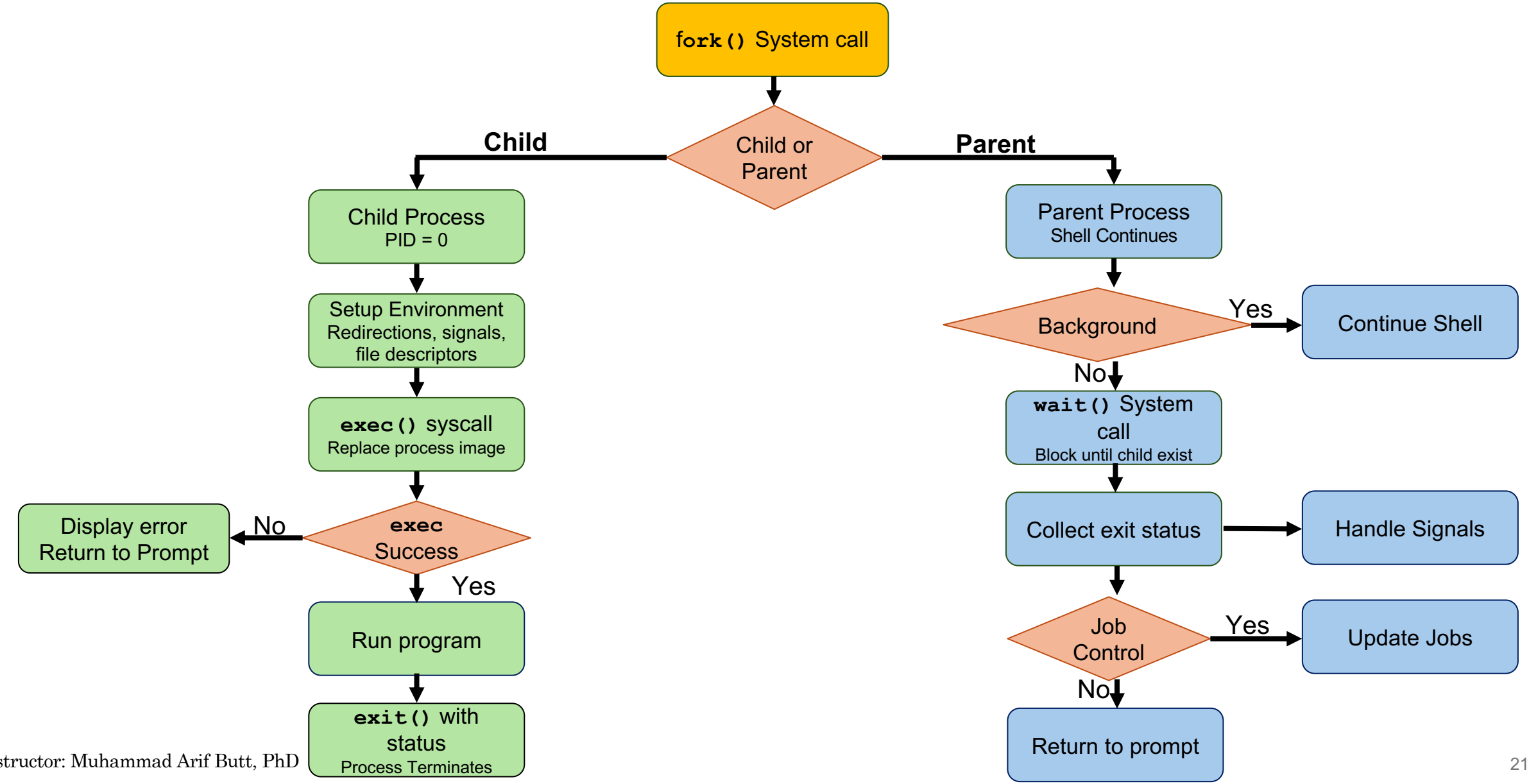


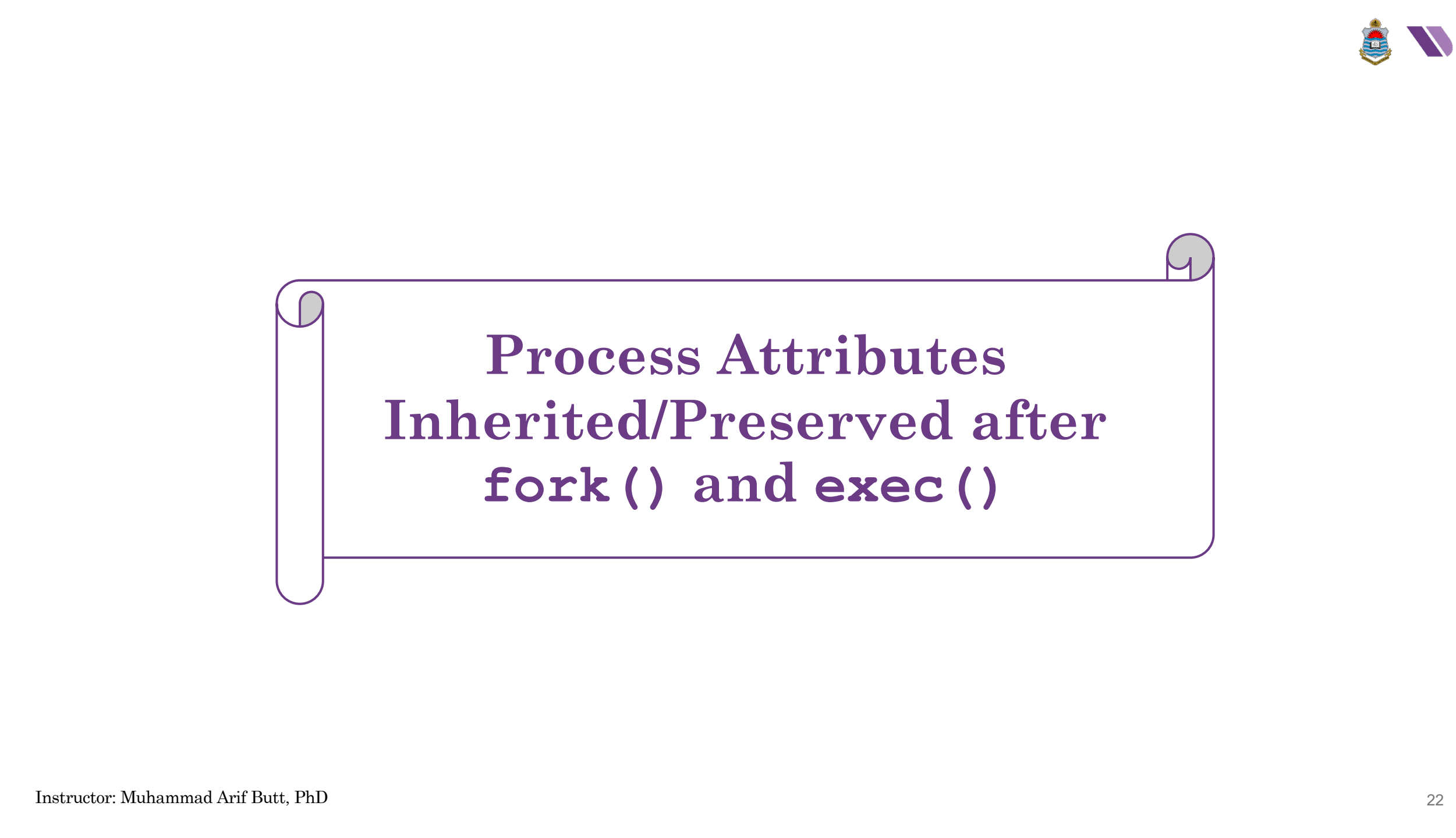
GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

How Shell Execute Commands?



How Shell Execute Commands? (cont...)




Process Attributes
Inherited/Preserved after
fork() and exec()

Attributes Inherited after `fork()` & `exec()`

Process IDs	fork()	exec()
PID	No	Preserved
PPID	No	Preserved
PGID	Inherited	Preserved
SID	Inherited	Preserved
Real IDs	Inherited	Preserved
Effective and Saved SUIDs	Inherited	Preserved
Supplementary Group IDs	Inherited	Preserved

Process Address Space	fork()	exec()
Text Segment	Shared	No
Stack Segment	No	No
Data and Heap Segment	Inherited	No
Environment Variables	Inherited	–
Memory Mappings	Inherited	No
Memory Locks	No	No

Files and Directories	fork()	exec()
PPFDT	Inherited	Preserved
Close-on-exec Flag	Inherited	Preserved
File offsets	Shared	Preserved
Open file status flags	Shared	Preserved
Directory streams	Inherited	No
Present working directory	Inherited	Preserved
File mode creation mask	Inherited	Preserved

Scheduling , Resources	fork()	exec()
Nice value	Inherited	Preserved
Priority	Inherited	Preserved
Scheduling policy	Inherited	Preserved
Resource limits	Inherited	Preserved
Resource usage	No	Preserved
CPU times	No	Preserved
Exit Handlers	Inherited	No

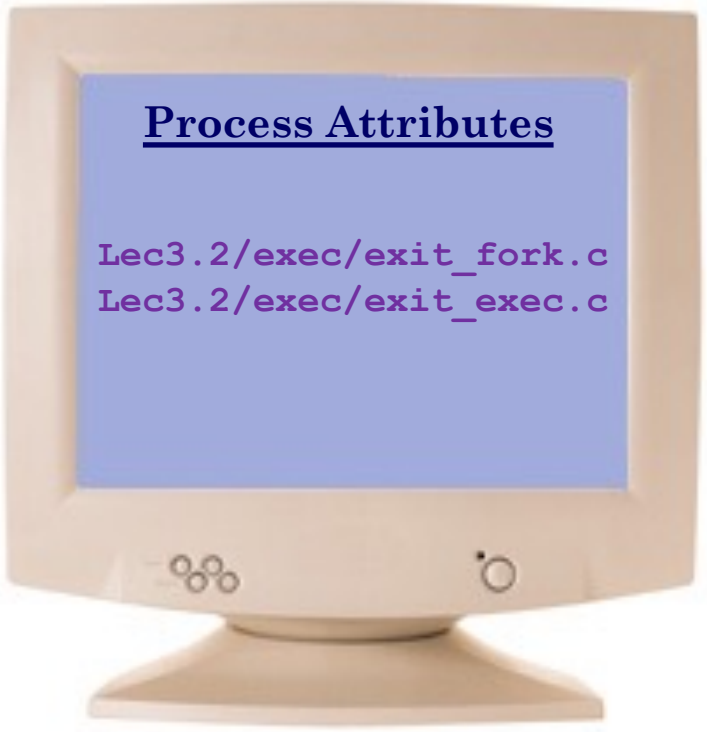
The `exec` command

- The **`exec`** is a shell built-in that replaces the current shell process with the specified command / program.

`exec [command] [command arguments]`

- For example the command `exec ls -l` will replace the current shell with the `ls` command. If successful, exec never returns because the shell process is completely replaced. So after `ls` completes, the terminal session ends, as there is no parent shell to return.
- Similarly, the command `exec > output.txt` will redirect all future stdout of the current shell to a file.
- Use Cases of **`exec`** command:
  - **Daemon Processes:** Used in start-up scripts where the wrapper script should be replaced by the actual daemon.
  - **Avoiding Process Spawning:** No `fork()` system call - current process is replaced directly. Saves memory and eliminates unnecessary parent process.
  - **Efficient I/O Redirection:** Permanently redirects file descriptors for the entire shell session. More efficient than repeated redirection in individual commands.

Demonstration



GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Executing shell command using system()

```
int system(const char* command);
```

The **system(char* cmd)** is a standard C library function, which is used to execute shell commands from within a program. It is passed a string and it spawns a shell program `/bin/sh` and use it to execute the argument passed as cmd (`/bin/sh -c "cal"`). The following two examples describes its behaviour:

```
//Lec3/2/system/system1.c
#include <stdio.h>
#include <stdlib.h>
int main() {
    system("cal");
   printf("Done…Bye\n");
    return 0;
}
```

```
(kali㉿kali)-[~/IS/module3/3.1/programs]
$ ./system1
    October 2024
Su Mo Tu We Th Fr Sa
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31

Done ... Bye
```

```
//Lec3/2/system/system2.c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[]) {
    system(argv[1]);
    printf("Done…Bye\n");
    return 0;
}
```

```
(kali㉿kali)-[~/IS/module3/3.1/programs]
$ ./system2 date
Wed Oct 30 09:41:17 AM PKT 2024

Done ... Bye
```

Executing shell command using `system()`

Now study the code of the following program, that prompts the user to enter a filename as input and constructs a command like `system("cat " + filename)` and display the contents of the file (`snprintf()` formats and stores a series of characters and values in the array as per the format string)

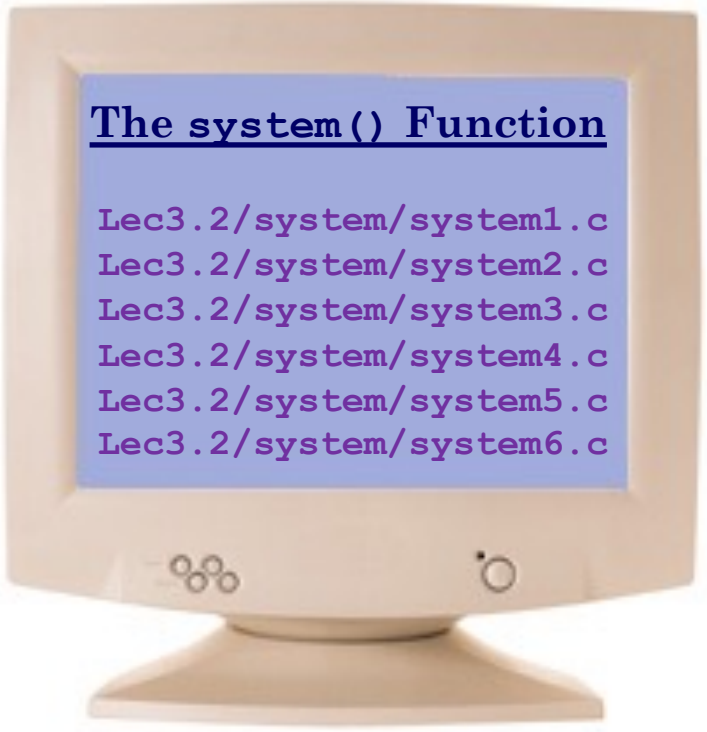
The `system()` function is really great however, it can introduce significant security vulnerabilities if not used carefully. If user input is passed to `system()`, w/o any validation, an attacker can manipulate that input to execute arbitrary commands. If the command string contains special characters like `;`, `&`, or `|`, it can lead to command execution beyond what the programmer intended.

```
//Lec3/2/system/system4.c
#include <stdio.h>
#include <stdlib.h>
int main() {
    char filename[100];
    printf("Enter a filename to display its content: ");
    fgets(filename, sizeof(filename), stdin);
    char command[150];
    snprintf(command, sizeof(command), "cat %s", filename);
    system(command);  // Potentially dangerous!
    return 0;
}
```

Try running the above program with a file name say **`f1.txt`**.

What will happen if the user gives this input: **`f1.txt;/bin/sh`**. What will happen, do you get a shell? Is this a shell with root privileges? If not, can we get a shell with root privileges?

Demonstration



GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Overview of Background Processes and Daemons

Introduction to Daemon Processes

- A **daemon** is a background system process that performs automated tasks for system administration (`crond`), manages kernel-level operations (`kthreadd`), or provides network services to clients and other processes (`httpd`).
- **Key Characteristics:**
  - A daemon is **long-lived**, often created at system startup and runs until the system is shut down.
  - It runs in the **background** and has no controlling terminal, and operates independently of user sessions.
  - The lack of a controlling terminal ensures that the kernel never automatically generates any terminal-related signals such as `SIGINT`, `SIGQUIT`, `SIGTSTP`, and `SIGHUP` for a daemon.
  - Modern Linux systems use `systemd` to manage daemons efficiently, starting services in parallel and launching them only when needed.
- **Example Daemon Processes:**
  - **`crond`:** a daemon that executes commands at a scheduled time,
  - **sshd:** the secure shell daemon enables remote system access with encrypted communication.
  - **`httpd/ngnix`:** Web server daemons serving content via HTTP/HTTPS protocols.
  - **xinetd:** is the Extended Internet daemon (super server). If you do not want the services (like daytime, echo, telnet, etc) to be started at system initialization time by `systemd`, and be dormant until a connection request arrives, `xinetd` is the only daemon process started. When a request comes in for any one of the above services, `xinetd` starts the appropriate service.

Common Unix job scheduling commands

Linux provides several tools for scheduling tasks to run automatically or at a specified time

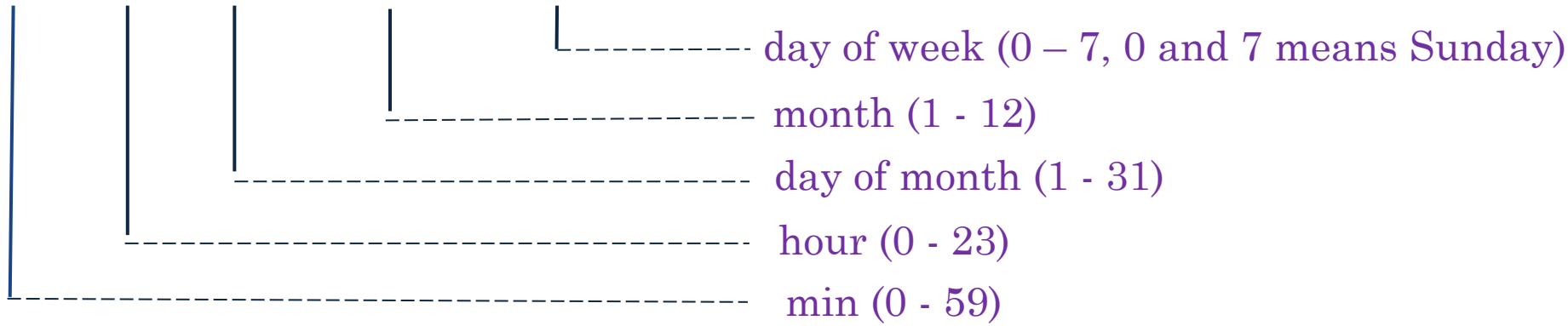
- **`at` utility:** Schedules one-time tasks for specific future times when you know exactly when something needs to happen; perfect for temporary jobs, system reboots, or ad-hoc administrative tasks. For example, to schedule a one-time system reboot at 11:30 PM today, you can use the command: `echo "sudo reboot" | at 23:30`
- **`batch` utility:** Schedules one-time tasks that run automatically when system load drops below a threshold (typically 1.5 load average); best for resource-intensive jobs like file compression, database maintenance, or large data processing that shouldn't interfere with peak system usage. For example, to schedule a database optimization to run when system load is low, you can use the command: `echo "/usr/bin/mysql_optimize.sh" | batch`
- **`cron utility`:** Schedules recurring tasks (hourly, daily, weekly, monthly) that need to run consistently regardless of system load. It is ideal for system maintenance, backups, and regular monitoring tasks that must execute on schedule. For example, edit crontab and add the following line to run backup script every day at 2:30 AM

```
30 2 * * * /usr/local/bin/backup.sh
```

The Linux cron utility

- Use the `crontab -e` command to edit current user crontab file (**`/var/spool/cron/crontabs/<user>`**), add your crontab entries using the syntax below. Once you save and exit, cron automatically validates and loads the new schedule. To verify the entries, you can use the `crontab -l` command.

```
m    h   dom    mon    dow        <command>
*    *    *      *      *         <command>
```



- Never edit the crontab file directly, rather use `crontab -e`, because it validates syntax before saving.
- The `crond` is typically installed and enabled on modern Linux systems. You can check its status using the `systemctl` command

Managing services using `systemctl`

- The **`systemd`** is a *system and service manager for Linux operating systems*. When run as first process on boot (as PID 1), it acts as init system that brings up and maintains user space services. The `/sbin/init` and `/bin/systemd` are both a soft link to `/lib/systemd/system`
- The **`systemctl`** is a program that is *used to introspect and control the state of the `systemd` system* and service manager. To check the status of xinetd on your system use:

```
# systemctl status/start/stop/enable/disable xinetd
```

- Try checking and changing the status of following services:
  - **`networking.service`**
  - **`sshd.service`**
  - **`cron.service`**
  - **`apache2.service`**
  - **`mysql.service`**
  - **`postgresql.service`**
  - **`xinetd.service`**
  - **`firewalld.service`**
  - **`docker.service`**
  - **`containerd.service`**
  - **`bluetooth.service`**
  - **`named.service`**

Writing a Daemon Process

Writing your own Daemon

Step-I (fork and Parent exit): Perform a `fork()`, after which the parent exits and the child continues. The child process inherits the PGID of the parent ensuring that the child is not a process group leader. The daemon process becomes the child of `/lib/systemd/systemd` process having PID of 992, which is further the child of `/sbin/init` process having a PID of 1

```
pid_t cpid = fork();
if(cpid > 0)
   exit(0);
```

Step-II (Make the daemon session leader): The `setsid()` creates a new session and process group. It detaches completely from controlling terminal

```
setsid();
```

Step-III (Close inherited file descriptors): Get maximum file descriptor limit and close all open file descriptors (except `stdin`, `stdout`, and `stderr`) that the daemon may have inherited from its parent.

```
struct rlimit r;
getrlimit(RLIMIT_NOFILE, &r);
for(i=3; i<r.rlim_max; i++)
     close(i);
```

Step-IV (Single Instance Protection): Only a single instance of a daemon process should run. For example, if multiple instances of `cron` start running, each would run a scheduled operation. So we can use the opposite logic to ensure that if one instance of a program is running, no user should be able to run another instance. Create a file `f1.txt` and achieve exclusive write lock on it using `fcntl()` system call. If lock fails, that means another instance is already running.

```
int fd = open("f1.txt", O_CREAT|O_TRUNC|O_RDWR, 0666);
struct flock lock;
lock.l_start = 0;
lock.l_len = 0;
lock.l_type = F_WRLCK;
lock.l_whence = SEEK_SET;
int rv = fcntl(fd, F_SETLK, &lock);
if(rv == -1){
   printf("Process is already running\n");
   close(fd);
   exit(1); }
```

Writing your own Daemon

Step-V (Redirect Standard I/O to `/dev/null`): Make the file descriptors 0, 1, and 2 of PPFDT point to the file `/dev/null`. This is done to ensures that if the daemon calls library functions that perform I/O on these descriptors, those functions won't unexpectedly fail.

```
int fd0 = open("/dev/null", O_RDWR);
dup2(fd0, 0);
dup2(fd0, 1);
dup2(fd0, 2);
close(fd0);
```

Step-VI (Environment Setup): Set the file mode creation mask to 0 by calling `umask(0)`, to ensure that, when the daemon creates files and directories, they have exactly the same access privileges as mentioned in the mode specified in an `open()` or `creat()` system call. Change the process's current working directory, typically to the root directory (/). This is necessary because a daemon usually runs until system shutdown; if the daemon's current working directory is on a file system other than the one containing /, then that file system can't be unmounted.

```
umask(0);
chdir("/");
```

Step-VII (Signal. Handling): Handle the SIGHUP signal, so that when this signal arrives, the daemon should ignore it

```
signal(SIGHUP,SIG_IGN);
```

Compiling and Testing your own Daemon

- **Compile and run the daemon:**

```
$ gcc mydaemon.c –o mydaemon
$ ./mydaemon &
Daemon has started running with PID: 248444
```

- **Check if any process has the f1.txt opened and locked:**

```
$ lsof f1.txt
…
```

- **Try to run multiple instances of daemon:**

```
$ ./mydaemon &
This process is already running
```

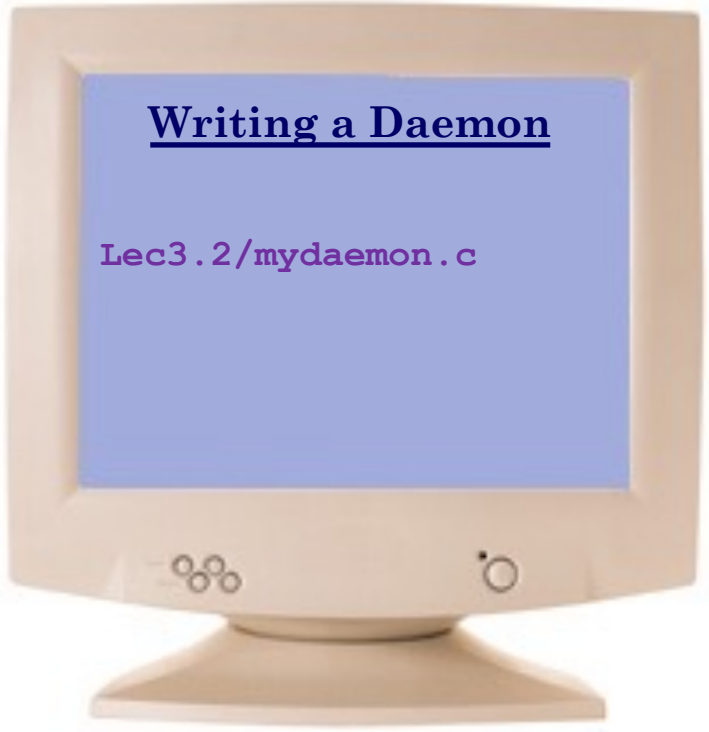
- **Check attributes of daemon:**

```
$ ps -ef | head -1; ps ef | grep mydaemon
…
```

- **Once done, you may kill the daemon process:**

```
$ kill <PID>
$ pkill mydaemon
```

Demonstration



GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

To Do

- Watch SP video on Process Management-I:
  https://youtu.be/R_01xGLp0ZQ?si=NQfovVEPiM0t1PMh

- Watch SP video on Process Management-II:
  https://youtu.be/91qzstPN1p8?si=4VnUXtlGw9hJadyV

Coming to office hours does NOT mean that you are academically weak!