# Operating System

## Lab Assignment 4

Comprehensive study of different categories of Linux system calls, categorised as:

### 1. Process Management System Calls

- **fork():**
  In Linux systems, `fork()` is traditionally used to generate a new process by copying the currently executing process. The result is a child process that runs concurrently with the parent. Though shell scripting lacks a direct `fork()` command, its effect can be replicated using background execution (`&`) or sub-shells (`()`).

- **exec():**
  The `exec()` call is designed to replace the existing process with another program, keeping the same process ID. In shell scripts, the `exec` command behaves similarly—it starts a new program in place of the current shell process, effectively ending the original script's execution from that point onward.

- **wait():**
  The `wait` command in bash allows the script to pause until background tasks complete. It operates similarly to the `wait()` system call in C, which suspends the parent process until the child finishes its task.

- **exit():**
  Used to terminate a shell script, `exit` sends a status code back to the shell that launched the script. This status helps indicate whether the script completed successfully (`0`) or encountered an error (non-zero values).



>_ main.sh

```
1   # Simulate fork using background process
2   echo "Parent Process: $$"
3   (sleep 2; echo "Child Process: $$") &
4
5   # exec replaces current shell, comment to continue script
6   # exec ls
7
8   # Wait for background process to complete
9   wait
10  echo "Background process finished."
11
12  # Exit with status 0
13  exit 0
```

```
~/workspace$ bash main.sh
Parent Process: 900
Child Process: 900
Background process finished.
```

---

## 2. File Management System Calls

- **open():**
  In low-level programming, `open()` is used to access files using file descriptors. Bash scripts don't use `open()` directly. Instead, redirection symbols (`>`, `>>`, `<`) and `exec` with custom file descriptors help achieve similar outcomes.

- **read():**
  This system call retrieves data from a file or input stream. In shell, the `read` command serves this purpose by accepting input from the user or files, usually reading one line at a time.

- **write():**
  `write()` outputs data to a destination like a file or terminal. Shell scripts achieve this using `echo`, `printf`, or redirection to transfer text to a target file or output stream.

- **close():**
  The `close()` system call is used to release file descriptors. In bash, we manually close file descriptors opened via `exec` by using `exec fd>&-` (for output) or `exec fd<&-` (for input).



>_ main.sh

```
1   # Redirecting output to file (write)
2   echo "Hello, World!" > file.txt
3
4   # Reading from file
5   while read line; do
6     echo "Read: $line"
7   done < file.txt
8
9   # Closing file descriptor not required unless using exec
10  exec 3>logfile.txt
11  echo "Logging something" >&3
12  exec 3>&-
```

```
~/workspace$ bash main.sh
Read: Hello, World!
```

---

## 3. Device Management System Calls

- **read() and write():**
  These calls enable reading from and writing to devices (e.g., keyboard, screen). Bash accomplishes this by treating devices as files. Inputs are gathered using `read`, and outputs are sent via `echo` or `printf`.

- **ioctl():**
  This call is used for low-level device control. While not directly accessible in bash, tools like `stty`, `tput`, or `setterm` allow similar control over terminal settings and behavior.

- **select():**
  Bash includes `select` as a built-in mechanism to generate interactive menus. It allows users to choose from options and executes different commands depending on their selection.

main.sh

```
# Using stty to control terminal (ioctl alternative)
stty -a

# Using tput to get terminal info
cols=$(tput cols)
echo "Terminal has $cols columns."

# select example (menu)
select option in "Option1" "Option2" "Quit"; do
  case $option in
    Option1) echo "You chose Option1";;
    Option2) echo "You chose Option2";;
    Quit) break;;
    *) echo "Invalid option";;
  esac
done
```



```
~/workspace$ bash main.sh
speed 38400 baud; rows 42; columns 71; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
eol2 = <undef>; swtch = <undef>; start = ^Q; stop = ^S; susp = ^Z;
rprnt = ^R; werase = ^W; lnext = ^V; discard = ^O; min = 1; time = 0;
-parenb -parodd -cmspar cs8 -hupcl -cstopb cread -clocal -crtscts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon
-ixoff -iuclc -ixany -imaxbel iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0
vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop
-echoprt echoctl echoke -flusho -extproc
Terminal has 71 columns.
1) Option1
2) Option2
3) Quit
#?
```

## 4. Network Management System Calls

- **socket():**
  In network programming, `socket()` is used to create a communication endpoint. Though bash lacks native socket support, tools like `netcat (nc)` simulate socket functionality for sending/receiving data over networks.

- **connect():**
  Used to establish client-to-server connections after socket creation. In bash, similar connections can be made using `telnet`, `nc`, `curl`, or `ssh`, which encapsulate `connect()` behavior internally.

- **send():**
  This function transmits data through an established connection. In shell scripts, this can be mimicked by piping `echo` or `printf` to tools like `nc`, which send the message over the network.

- **recv():**
  Used to accept incoming data via a socket. Bash scripts simulate this using `nc -l` (listen mode), which waits for connections and outputs received data to the console.



```
>_ main.sh
1  # Simulate socket communication using netcat
2  # Run this in one terminal to act as server:
3  # nc -l 12345
4
5  # Run this in another terminal as client:
6  echo "Hello from client" | nc localhost 12345
```

```
~/workspace$ bash main.sh
Hello from client
```

If the output is an error :

the error `nc: command not found` means that `netcat` is not installed in your current environment

---

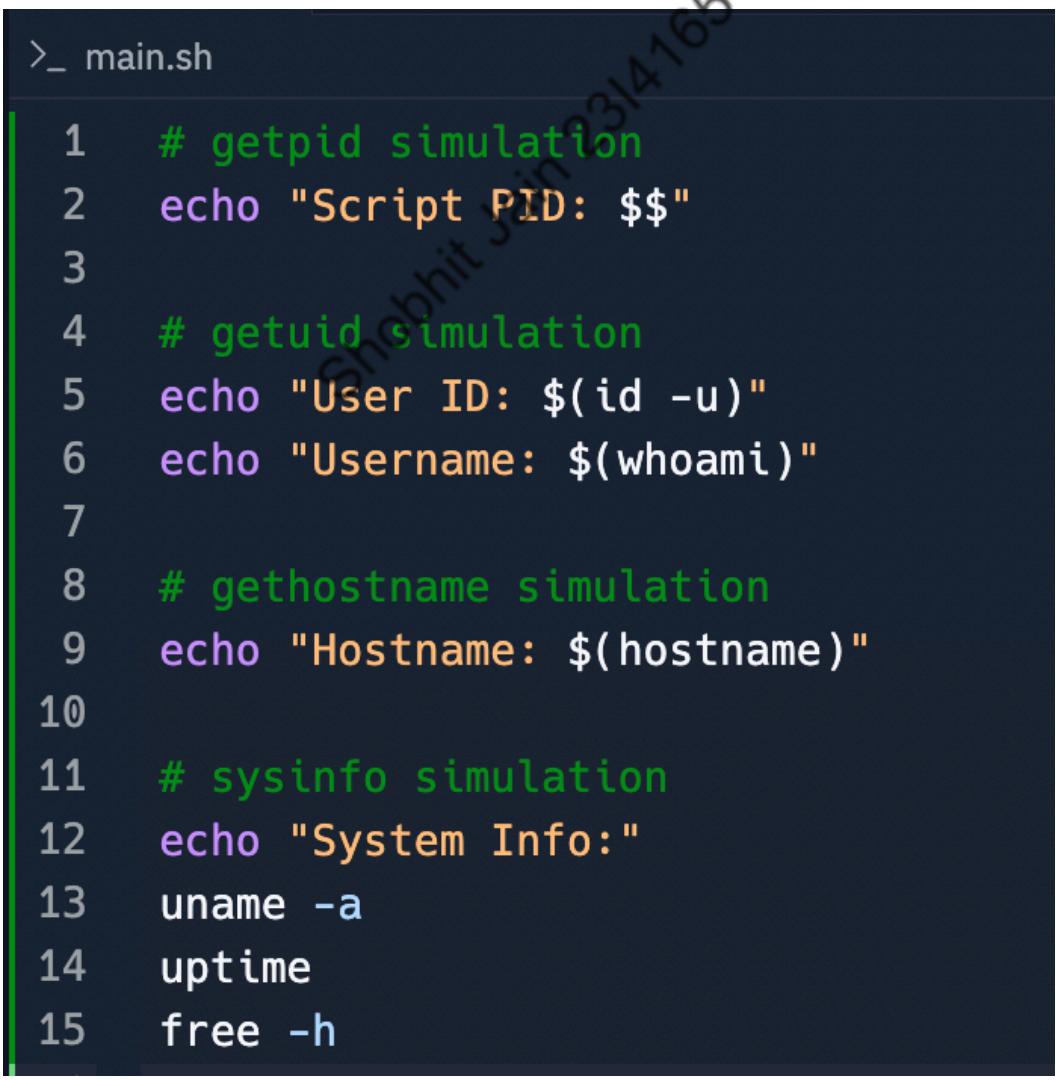
## 5. System Information Management System Calls

- **getpid():**
  Returns the process ID of the current process. In shell scripting, the special variable `$$` serves the same purpose, helping track the active shell session or script.

- **getuid():**
  Provides the user ID executing the current session. This can be obtained using `id -u` or `whoami` in bash, useful for checking privileges or performing user-specific tasks.

- **gethostname():**
  Retrieves the name of the current host machine. In shell scripts, `hostname` or `uname -n` can be used to gather this information, useful for system logs or multi-host automation.

- **sysinfo():**
  Displays system-level statistics like uptime, memory usage, and load. Commands like `uname -a`, `uptime`, and `free -h` are used in bash scripts to collect such data for monitoring or diagnostics.



```
~/workspace$ bash main.sh
Script PID: 1045
User ID: 1000
Username: runner
Hostname: 829d900354d6
System Info:
Linux 829d900354d6 6.2.16 #1-NixOS SMP PREEMPT_DYNAMIC Tue Jan  1 00:00
:00 UTC 1980 x86_64 GNU/Linux
 21:45:16  up 1 day  3:04,  0 users,  load average: 0.88, 1.50, 4.15
              total        used        free      shared  buff/cache   a
vailable
Mem:           62Gi        21Gi        24Gi        20Mi        16Gi
    40Gi
Swap:            0B          0B          0B
```

---

---