

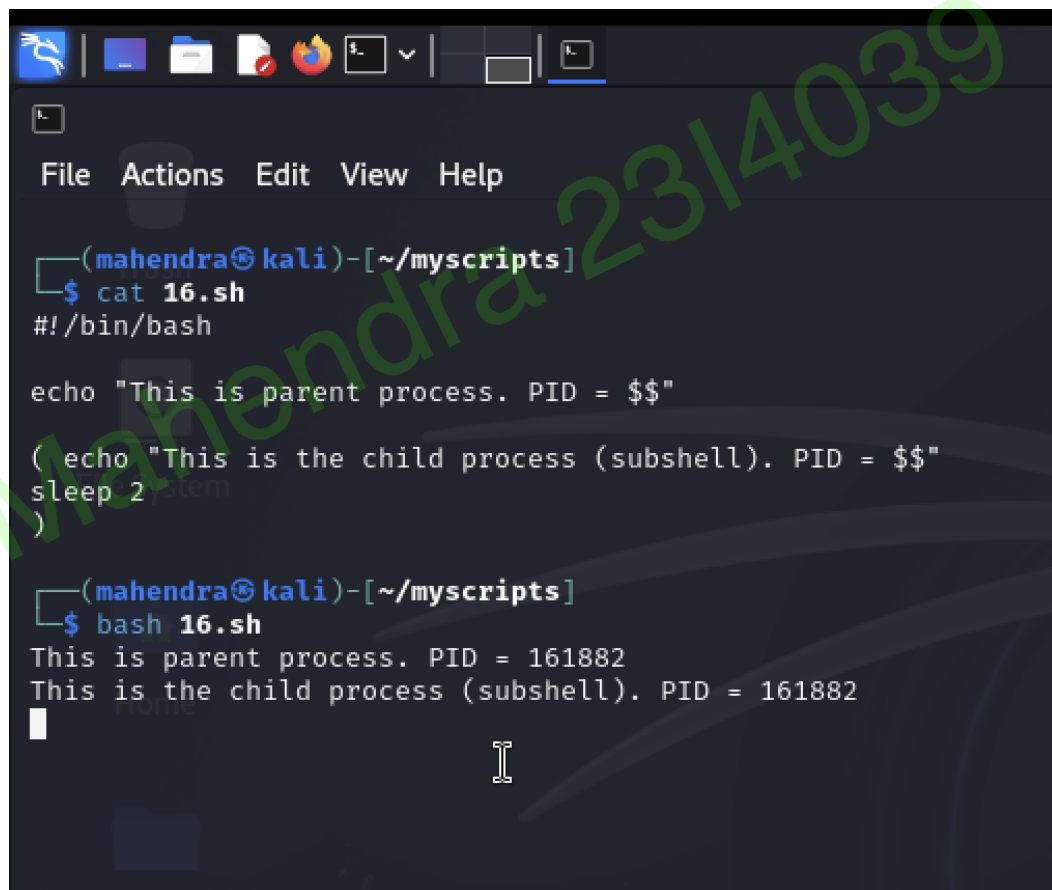
4ITRC2 Operating System Lab

Lab Assignment 4

Comprehensive study of different categories of Linux system calls, categorized as

1. Process Management System calls

- **fork()** : In Linux and UNIX systems, fork() is a **system call** used to create a new process by duplicating the calling (parent) process. The newly created process is called the **child process** and gets a unique Process ID (PID). In shell scripts there is no command named as fork but we can simulate the same behaviour as fork using subshells or using background process.



```
(mahendra@kali)-[~/myscripts]
$ cat 16.sh
#!/bin/bash

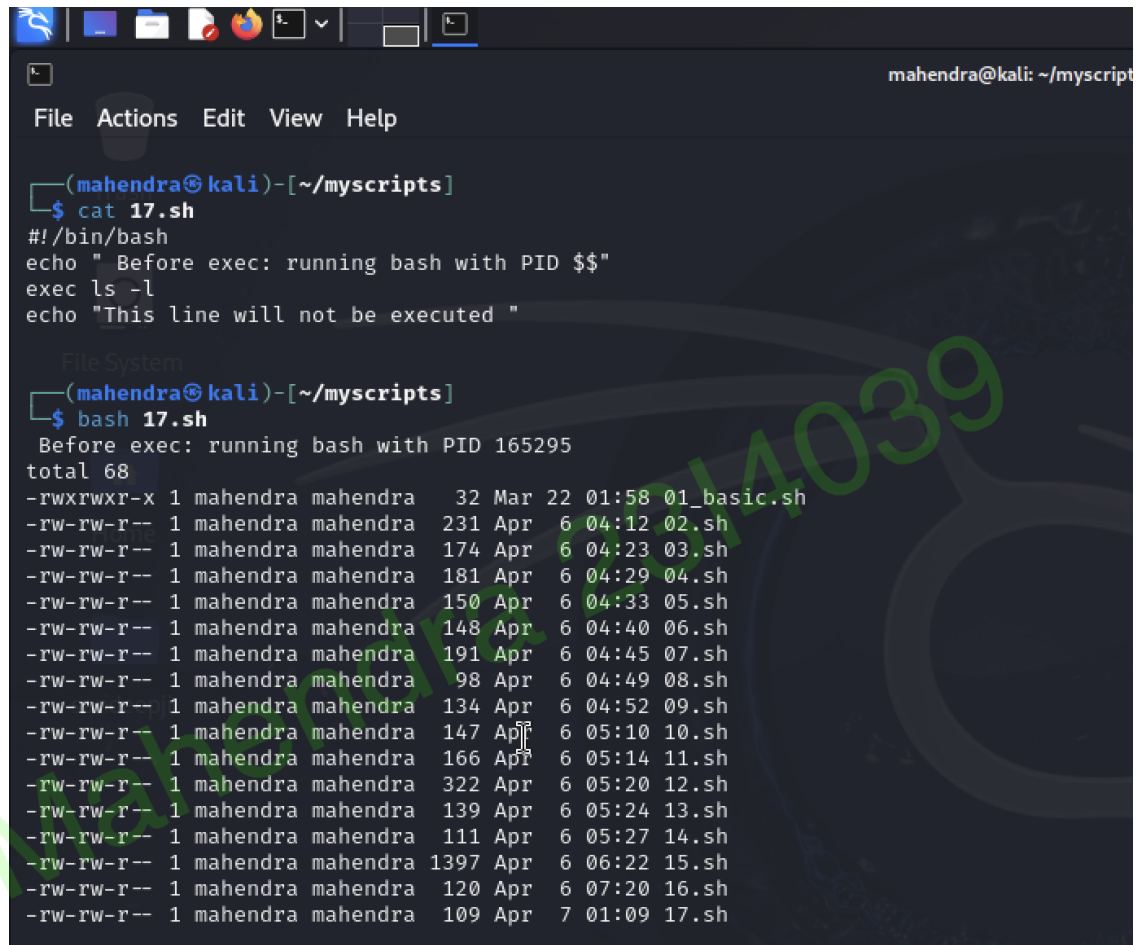
echo "This is parent process. PID = $$"

( echo "This is the child process (subshell). PID = $$"
  sleep 2
)

(mahendra@kali)-[~/myscripts]
$ bash 16.sh
This is parent process. PID = 161882
This is the child process (subshell). PID = 161882
```

- **exec()** : In Linux, exec() is a system call that **replaces the current process image with a new process**. It doesn't create a new process; it **loads a new program into the current process space**. After exec() is called, the **original process is replaced** and doesn't continue.

In **shell scripting**, the `exec` command behaves similarly—it **replaces the current shell** with a new command or script. Once `exec` is invoked, the shell does **not return to the script**, unless `exec` is used only to redirect file descriptors.

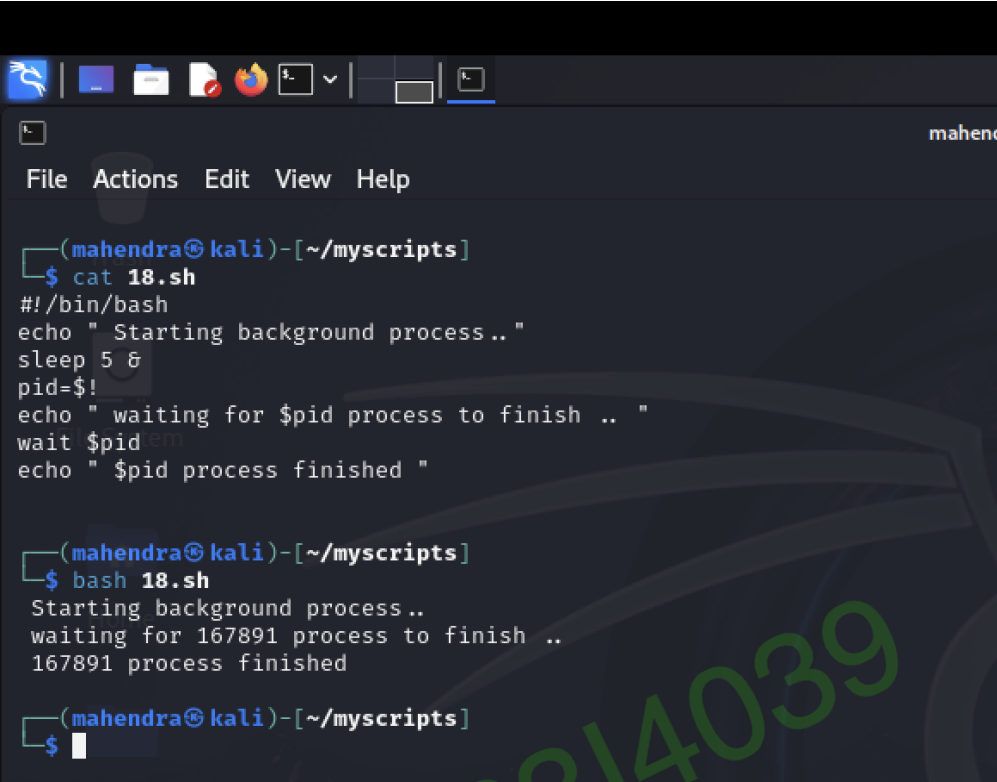


The screenshot shows a terminal window with a dark background. At the top, there's a menu bar with 'File', 'Actions', 'Edit', 'View', and 'Help'. The terminal prompt is '(mahendra@kali)-[~/myscripts]'. The user enters '\$ cat 17.sh', which displays the contents of the script. The script starts with '#!/bin/bash', followed by an echo statement, then 'exec ls -l', and another echo statement. The user then enters '\$ bash 17.sh', which replaces the prompt and shows the output of 'ls -l'. The output lists various files in the directory, including '01_basic.sh' through '17.sh'. A large green watermark '4039' is visible across the terminal output.

```
(mahendra@kali)-[~/myscripts]
$ cat 17.sh
#!/bin/bash
echo " Before exec: running bash with PID $$"
exec ls -l
echo "This line will not be executed "
```

```
(mahendra@kali)-[~/myscripts]
$ bash 17.sh
Before exec: running bash with PID 165295
total 68
-rwxrwxr-x 1 mahendra mahendra 32 Mar 22 01:58 01_basic.sh
-rw-rw-r-- 1 mahendra mahendra 231 Apr 6 04:12 02.sh
-rw-rw-r-- 1 mahendra mahendra 174 Apr 6 04:23 03.sh
-rw-rw-r-- 1 mahendra mahendra 181 Apr 6 04:29 04.sh
-rw-rw-r-- 1 mahendra mahendra 150 Apr 6 04:33 05.sh
-rw-rw-r-- 1 mahendra mahendra 148 Apr 6 04:40 06.sh
-rw-rw-r-- 1 mahendra mahendra 191 Apr 6 04:45 07.sh
-rw-rw-r-- 1 mahendra mahendra 98 Apr 6 04:49 08.sh
-rw-rw-r-- 1 mahendra mahendra 134 Apr 6 04:52 09.sh
-rw-rw-r-- 1 mahendra mahendra 147 Apr 6 05:10 10.sh
-rw-rw-r-- 1 mahendra mahendra 166 Apr 6 05:14 11.sh
-rw-rw-r-- 1 mahendra mahendra 322 Apr 6 05:20 12.sh
-rw-rw-r-- 1 mahendra mahendra 139 Apr 6 05:24 13.sh
-rw-rw-r-- 1 mahendra mahendra 111 Apr 6 05:27 14.sh
-rw-rw-r-- 1 mahendra mahendra 1397 Apr 6 06:22 15.sh
-rw-rw-r-- 1 mahendra mahendra 120 Apr 6 07:20 16.sh
-rw-rw-r-- 1 mahendra mahendra 109 Apr 7 01:09 17.sh
```

- **wait()**: In shell scripting, the `wait` command is used to **pause the execution of the current shell script** until all background processes started by that shell have finished, or until a specific process ID (PID) has finished.
It is based on the **wait() system call** in Linux, which allows a parent process to wait for its child process to terminate.

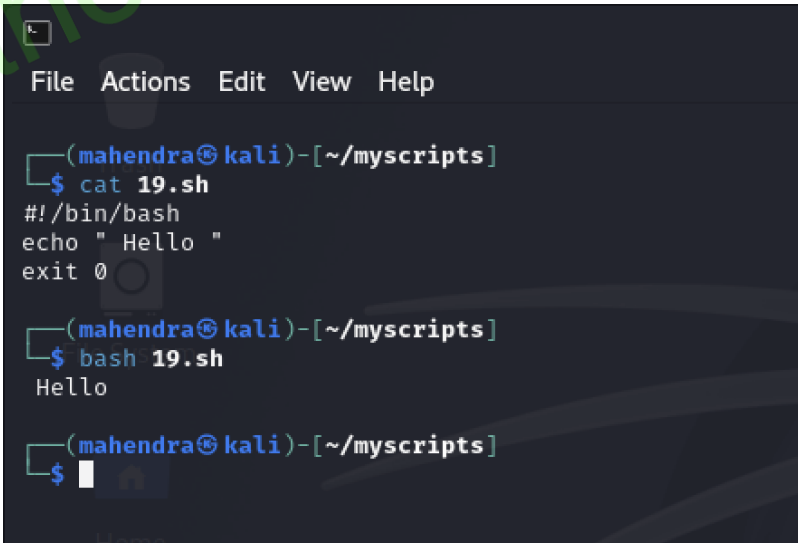


```
(mahendra@kali)-[~/myscripts]
$ cat 18.sh
#!/bin/bash
echo " Starting background process.."
sleep 5 &
pid=$!
echo " waiting for $pid process to finish .. "
wait $pid
echo " $pid process finished "
```

```
(mahendra@kali)-[~/myscripts]
$ bash 18.sh
Starting background process..
waiting for 167891 process to finish ..
167891 process finished

(mahendra@kali)-[~/myscripts]
$
```

- **exit():** The exit command in shell scripting is used to **terminate a script** and return an **exit status code** to the calling process (usually the shell or parent script). This helps indicate whether a script **completed successfully** or **encountered an error**.



```
(mahendra@kali)-[~/myscripts]
$ cat 19.sh
#!/bin/bash
echo " Hello "
exit 0
```

```
(mahendra@kali)-[~/myscripts]
$ bash 19.sh
Hello

(mahendra@kali)-[~/myscripts]
$
```

2. File Management System calls

- **open()** : In low-level languages like **C**, open() is a **system call** used to open a file and return a **file descriptor**, which can then be used for reading, writing, or both. But in **shell scripting**, we don't use open() directly. Instead, **shell provides high-level syntax** using:
 - Redirection operators (>, >>, <)
 - exec command to open files with custom file descriptors

```

(mahendra@kali)-[~/myscripts]
$ cat 20.sh
#!/bin/bash
filename="demo.txt"
echo "this is a demo line" > "$filename"
exec 3< "$filename"
read line <&3
echo "read from $filename : $line"
exec 3<&-

(mahendra@kali)-[~/myscripts]
$ bash 20.sh
read from demo.txt : this is a demo line

(mahendra@kali)-[~/myscripts]
$

```

- **read()** : The read() system call is used to **receive input** from a file descriptor into a buffer. In shell scripting, read is a built-in command that reads **user input** from standard input (keyboard) or from **files via file descriptors**. It is commonly used to **prompt users**, read line-by-line data, or capture values from scripts for further use.
- **write()** : The write() system call is used to **output data** to a file descriptor. In shell scripting, this is mimicked using echo, printf, or redirection operators (>, >>). It allows you to write strings or outputs into files, terminals, or other writable streams.
- **close()** : The close() system call is used to **release a file descriptor** once operations are done, preventing memory or resource leaks. In shell scripting, file descriptors opened with exec must be closed using exec fd>&- for output or exec fd<&- for input.

The example of all the function is given together in a single script which is given below.

```
mahendra@kali: ~/myscripts
File Actions Edit View Help

(mahendra@kali)-[~/myscripts]
$ cat 21.sh
#!/bin/bash
exec 3> demo.txt
echo "This is the first line written using write(). " >&3
echo "This is the second line written to demo.txt " >&3

exec 3>&-
echo "Write is complete and file descriptor 3 is closed "
exec 4< demo.txt
echo "reading from demo.txt using read() ;"
while read -r line <&4; do
    echo "Read line :$line"
done
exec 4<&-
echo " Read complete and file descriptor 4 closed "
```

```
(mahendra@kali)-[~/myscripts]
$ bash 21.sh
Write is complete and file descriptor 3 is closed
reading from demo.txt using read() ;
Read line :This is the first line written using write().
Read line :This is the second line written to demo.txt
Read complete and file descriptor 4 closed

(mahendra@kali)-[~/myscripts]
$
```

3. Device Management System calls

- **read()** : The read() system call is used to **receive input** from a file descriptor into a buffer. In shell scripting, read is a built-in command that reads **user input** from standard input (keyboard) or from **files via file descriptors**. It is commonly used to **prompt users**, read line-by-line data, or capture values from scripts for further use.
- **write()** : The write() system call is used to **output data** to a file descriptor. In shell scripting, this is mimicked using echo, printf, or redirection operators (>, >>). It allows you to write strings or outputs into files, terminals, or other writable streams.

```
mahendra@kali: ~/myscripts
File Actions Edit View Help

(mahendra@kali)-[~/myscripts]
$ cat 21.sh
#!/bin/bash
exec 3> demo.txt
echo "This is the first line written using write(). " >&3
echo "This is the second line written to demo.txt " >&3

exec 3>&-
echo "Write is complete and file descriptor 3 is closed "
exec 4< demo.txt
echo "reading from demo.txt using read() ;"
while read -r line <&4; do
    echo "Read line :$line"
done
exec 4<&-
echo " Read complete and file descriptor 4 closed "
```

```
(mahendra@kali)-[~/myscripts]
$ bash 21.sh
Write is complete and file descriptor 3 is closed
reading from demo.txt using read() ;
Read line :This is the first line written using write().
Read line :This is the second line written to demo.txt
Read complete and file descriptor 4 closed
```

- **ioctl()** : **ioctl()** stands for **Input/Output Control**. It is a **system call** in **C/C++** used to configure or control **low-level device parameters** such as terminals, network interfaces, or hardware devices. It allows user-space programs to send **device-specific commands** to drivers.
In **shell scripting**, you can't use **ioctl()** directly, but you can perform **equivalent actions** using shell tools like:
 - stty (to configure terminal I/O)
 - setterm (for terminal behavior)
 - tput (to query terminal capabilities)

```
File Actions Edit View Help

(mahendra@kali)-[~/myscripts]
$ cat 22.sh
#!/bin/bash
echo -n "Enter password : "
stty -echo # Disable echoing
read password
stty echo # re-enable echoing
echo " "
echo " Hidden for security "
echo "$password"

(mahendra@kali)-[~/myscripts]
$ bash 22.sh
Enter password :
Hidden for security
12345
```

- **select()** : The select keyword is built into **bash**, and it's used for:
 - Creating numbered menus
 - Capturing user input as a choice
 - Running commands based on the user's selection

```
File Actions Edit View Help

(mahendra@kali)-[~/myscripts]
$ cat 23.sh
select option in "Date" "Calender" "Quit"
do
    case $option in
        "Date") date ;;
        "Calender") cal ;;
        "Quit") break ;;
        *) echo "Invalid option"
    esac
done

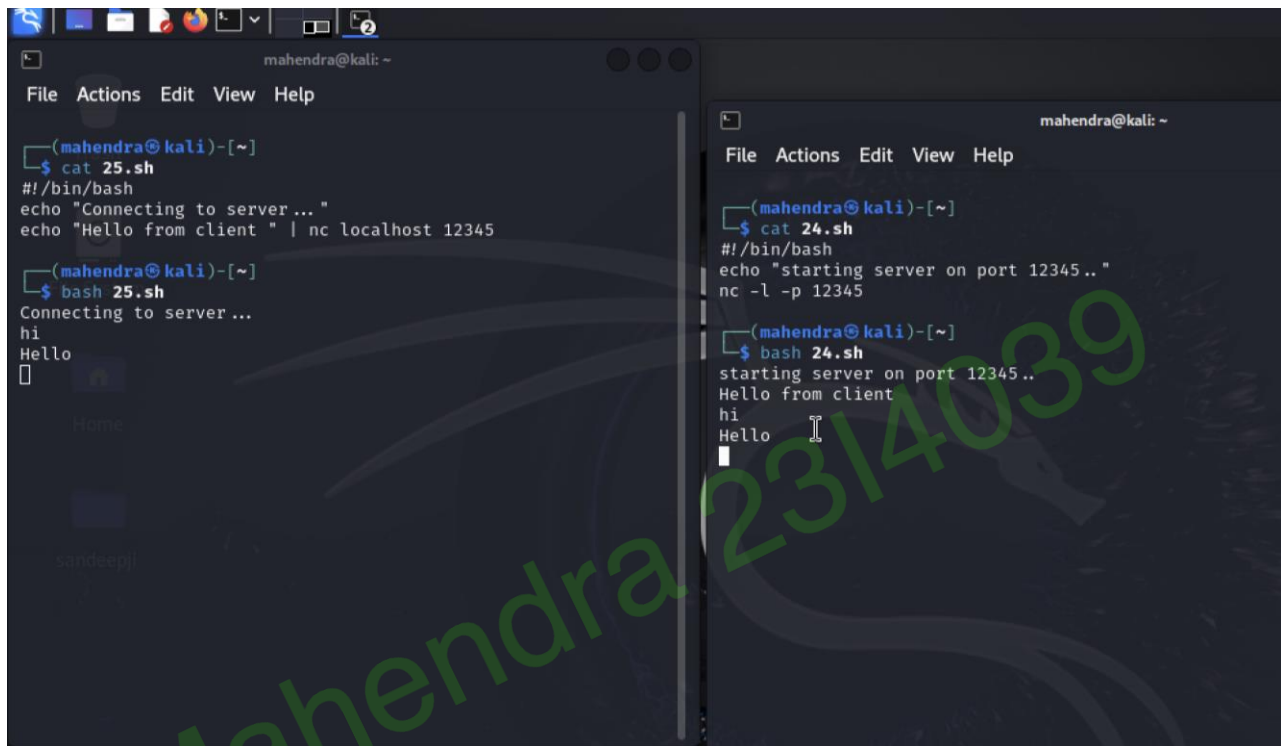
(mahendra@kali)-[~/myscripts]
$ bash 23.sh
1) Date
2) Calender
3) Quit
#? 1
Mon Apr 7 02:07:51 IST 2025
#? 2
April 2025
Su Mo Tu We Th Fr Sa
    1  2  3  4  5
  6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30
#? 3
```

4. Network Management System calls

- **socket()** : socket() is a system call used in networking to create an **endpoint for communication** between two machines (or processes) over a **network**. It

returns a **socket descriptor**, which is like a file descriptor, used for sending or receiving data. In shell scripting, we can **simulate socket behavior** using tools like netcat (nc), since bash doesn't support raw socket() calls like C.

Shell doesn't have a native socket() system call, but **netcat (or nc)** is a command-line tool that allows socket-based communication and can act as both **server** and **client**.



The image shows two terminal windows side-by-side on a Kali Linux system. The left window shows a client script (25.sh) being executed, which connects to a server on localhost 12345 and sends the message 'Hello from client'. The right window shows a server script (24.sh) being executed, which listens on port 12345 and receives the message 'Hello from client'. Both windows show the output of the scripts, including the connection status and the received message.

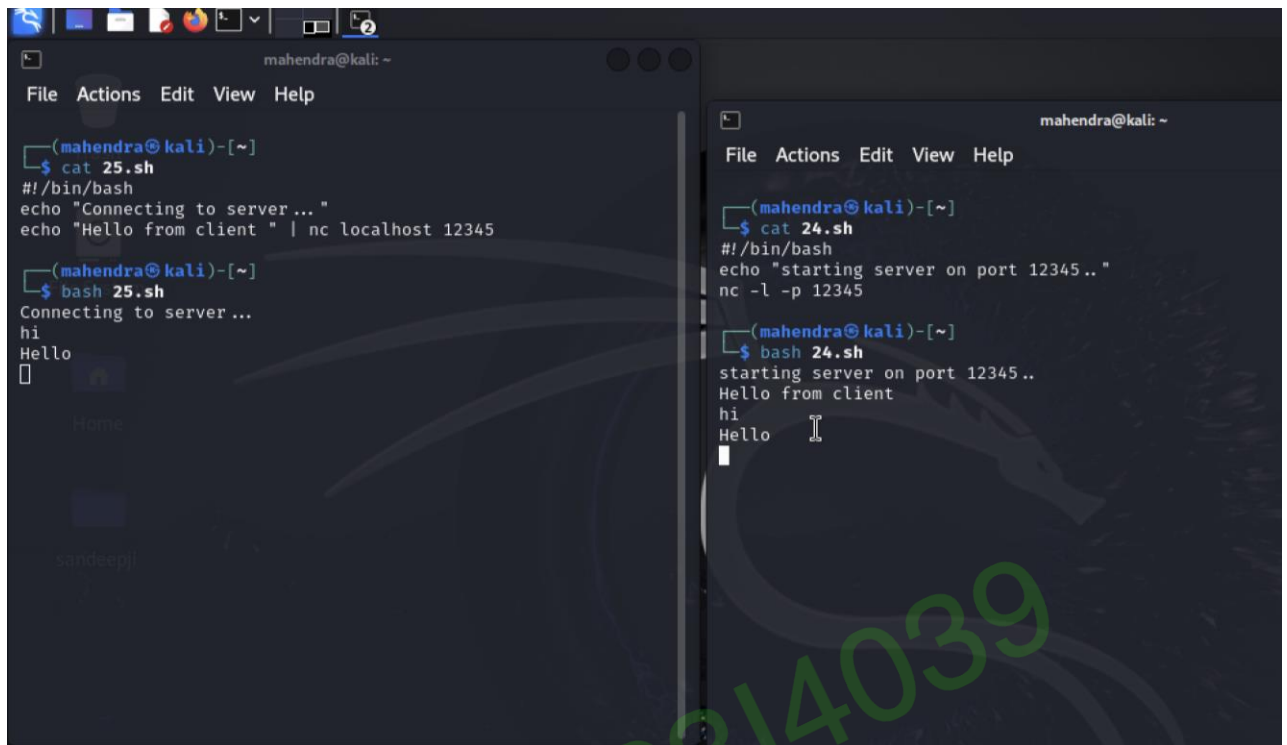
```
mahendra@kali: ~  
File Actions Edit View Help  
(mahendra@kali)-[~]  
$ cat 25.sh  
#!/bin/bash  
echo "Connecting to server..."  
echo "Hello from client " | nc localhost 12345  
  
(mahendra@kali)-[~]  
$ bash 25.sh  
Connecting to server...  
hi  
Hello  
[~]  
Home  
sandeep!
```

```
mahendra@kali: ~  
File Actions Edit View Help  
(mahendra@kali)-[~]  
$ cat 24.sh  
#!/bin/bash  
echo "starting server on port 12345.."  
nc -l -p 12345  
  
(mahendra@kali)-[~]  
$ bash 24.sh  
starting server on port 12345..  
Hello from client  
hi  
Hello  
[~]
```

- **connect()** : The connect() system call is used in **network programming** to establish a **connection from a client to a server** over a socket. It is part of the **socket API** and is used after creating a socket with socket().

But in **shell scripting**, we can't directly use connect(). Instead, we simulate it using tools like nc (netcat), telnet, or even curl and ssh—these tools abstract connect()

behind simple commands.



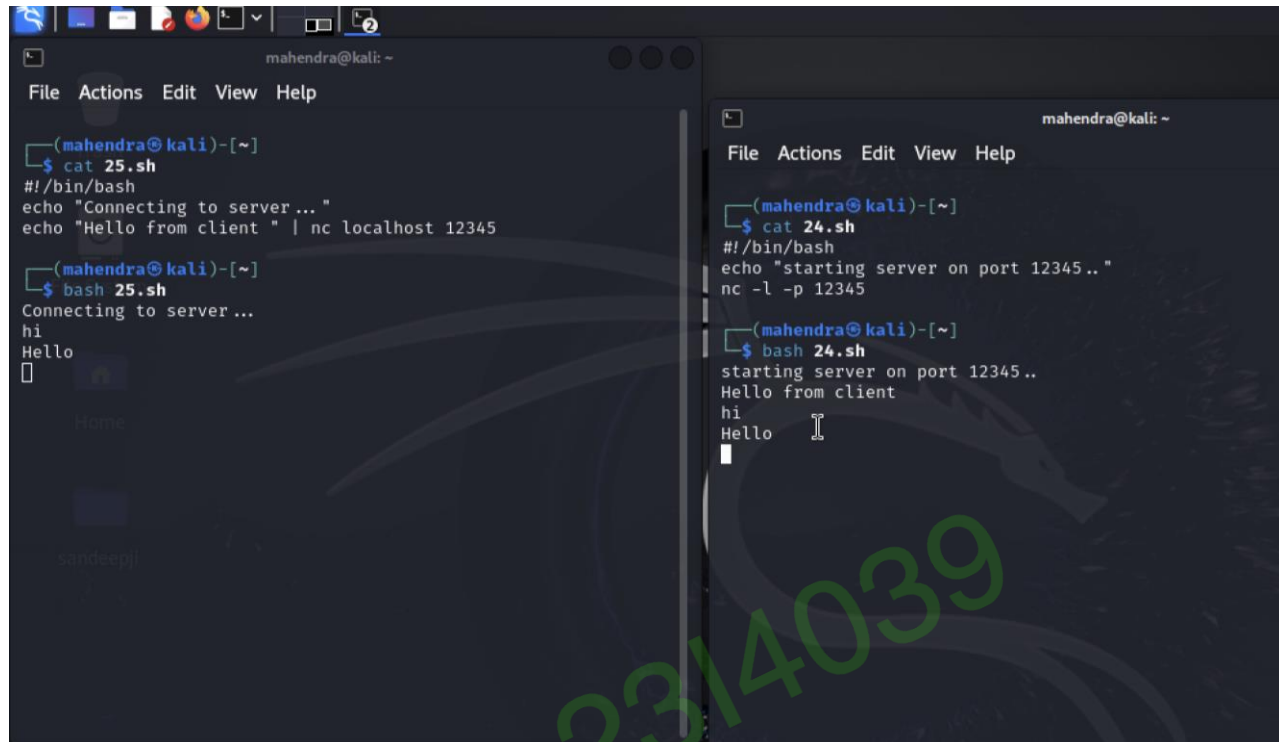
The image shows two terminal windows side-by-side on a Kali Linux system. The left window shows a client script (25.sh) being executed, which connects to localhost 12345 and sends the message 'Hello from client'. The right window shows a server script (24.sh) being executed, which listens on port 12345 and receives the message 'Hello from client'. Both windows show the output of the scripts, including the connection status and the received message.

```
mahendra@kali: ~  
File Actions Edit View Help  
(mahendra@kali)-[~]  
$ cat 25.sh  
#!/bin/bash  
echo "Connecting to server..."  
echo "Hello from client" | nc localhost 12345  
(mahendra@kali)-[~]  
$ bash 25.sh  
Connecting to server...  
hi  
Hello  
[~]  
Home  
sandeepj
```

```
mahendra@kali: ~  
File Actions Edit View Help  
(mahendra@kali)-[~]  
$ cat 24.sh  
#!/bin/bash  
echo "starting server on port 12345.."  
nc -l -p 12345  
(mahendra@kali)-[~]  
$ bash 24.sh  
starting server on port 12345..  
Hello from client  
hi  
Hello  
[~]
```

- **send()** : send() is a system call used in socket programming to **transmit data from one socket to another over a network**. It is typically used by the **client or server** to **send messages** through a connected socket. In C, it sends data in the form of bytes from the calling process to the destination.
In shell scripting, we simulate send() using tools like **echo piped into nc (netcat)**, which sends data to a specific IP and port.
- **recv()** : recv() is used to **receive data from a connected socket**. It's the counterpart of send(), typically used on the **server-side** to **accept messages sent by the client**. It listens for incoming data on a socket and reads it into a buffer.

In shell, we use **nc -l** (netcat in listening mode) to simulate `recv()`. It listens on a port and displays any data received.



The image shows two terminal windows side-by-side. The left window is titled 'mahendra@kali: ~' and shows a script '25.sh' being executed. The script contains the following code: `#!/bin/bash`, `echo "Connecting to server ..."`, and `echo "Hello from client " | nc localhost 12345`. The output shows 'Connecting to server ...', 'hi', and 'Hello'. The right window is also titled 'mahendra@kali: ~' and shows a script '24.sh' being executed. The script contains the following code: `#!/bin/bash`, `echo "starting server on port 12345.."`, and `nc -l -p 12345`. The output shows 'starting server on port 12345..', 'Hello from client', 'hi', and 'Hello'.

5. System Information Management System calls

- **getpid()** : `getpid()` is a system call that returns the **process ID (PID)** of the currently running process. In shell scripting, the special variable `$$` is used to retrieve the PID of the current script or shell session. It is helpful for identifying, tracking, and managing processes, especially in background or parallel execution scenarios.
- **getuid()** : `getuid()` returns the **User ID (UID)** of the user executing the current process. In shell scripting, the command `id -u` or `whoami` is used to get the UID or username. It's crucial for permission checks, access control, or running scripts conditionally based on user identity.
- **gethostname()** : `gethostname()` fetches the **name of the current machine or host**. In shell scripts, you use the `hostname` command or `uname -n`. It is often used in scripts that log system activity, generate host-specific reports, or automate tasks across different servers in a networked environment.
- **sysinfo()** : `sysinfo()` gives detailed information about the **system's uptime, memory, and load average**. In shell scripting, this is accessed using a combination of `uname`, `uptime`, and `free`. It's useful in monitoring, diagnostics,

or when making decisions in scripts based on resource availability or system load.



```
(mahendra@kali)-[~]
$ cat 26.sh
#!/bin/bash
# File: sysinfo_full.sh
# getpid()
echo "Process id (pid) : $$"
# getuid()
echo "User id (uid) : $(id-u)"
echo "user name : $(whoami)"
# gethostname()
echo "Hostname : $(hostname)"
# sysinfo()
echo " System name : $(uname -s)"
echo "kernel verison : $(uname -r)"
echo "System Uptime : "
uptime
echo "Memory Information : "
free -h

(mahendra@kali)-[~]
$ bash 26.sh
Process id (pid) : 233811
26.sh: line 6: id-u: command not found
User id (uid) :
user name : mahendra
Hostname : kali
System name : Linux
kernel verison : 6.6.15-arm64
System Uptime :
01:28:10 up 7:52, 1 user, load average: 0.00, 0.00, 0.00
Memory Information :
              total        used        free      shared  buff/cache   available
Mem:          3.8Gi         771Mi         2.1Gi          14Mi         1.2Gi         3.1Gi
Swap:          975Mi           0B          975Mi
```

