



# CMP325

## Operating Systems

### Lecture 06, 07

## I/O Redirection & IPC

**Muhammad Arif Butt, PhD**

### **Note:**

Some slides and/or pictures are adapted from course text book and Lecture slides of

- Dr Syed Mansoor Sarwar
- Dr Kubiatoicz
- Dr P. Bhat
- Dr Hank Levy
- Dr Indranil Gupta

For practical implementation of operating system concepts discussed in these slides, students are advised to watch and practice following video lectures:

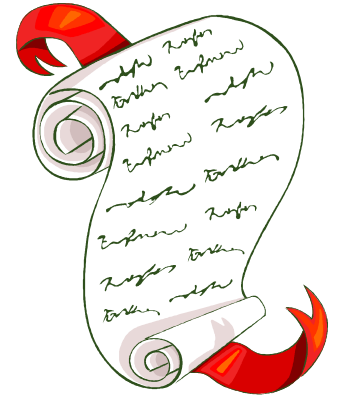
### **OS with Linux:**

[https://www.youtube.com/playlist?list=PL7B2bn3G\\_wfBuJ\\_WtHADcXC44piWLRzr8](https://www.youtube.com/playlist?list=PL7B2bn3G_wfBuJ_WtHADcXC44piWLRzr8)

### **System Programming:**

[https://www.youtube.com/playlist?list=PL7B2bn3G\\_wfC-mRpG7cxJMnGWdPAQTViW](https://www.youtube.com/playlist?list=PL7B2bn3G_wfC-mRpG7cxJMnGWdPAQTViW)

# Today's Agenda



- Review of previous lecture
- File Management in Linux
- IO Redirection
- Cooperating Processes
- Taxonomy of Inter Process Communication
- Persistence of IPC Objects
- Use of Pipes on the Shell
- Use of FIFOs on the Shell
- Use of Signals on the Shell

# Connection of an Opened File

# File Management in Linux

Following are the four key system calls for performing file I/O (programming languages and software packages typically employ these calls indirectly via I/O libraries):

- **fd = open(pathname, flags, mode)** opens the file identified by pathname, returning a file descriptor used to refer to the open file in subsequent calls. If the file doesn't exist, open() may create it, depending on the settings of the flags bit. The flags argument also specifies whether the file is to be opened for reading, writing, or both. The mode argument specifies the permissions to be placed on the file if it is created by this call. If the open() call is not being used to create a file, this argument is ignored and can be omitted
- **numread = read(fd, buffer, count)** reads at most count bytes from the open file referred to by fd and stores them in buffer. The read() call returns the number of bytes actually read. On eof, read() returns 0.
- **numwritten = write(fd, buffer, count)** writes up to count bytes from buffer to the open file referred to by fd. The write() call returns the number of bytes actually written, which may be less than count
- **status = close(fd)** is called after all I/O has been completed, in order to release the file descriptor fd and its associated kernel resources

# File Descriptor to File Contents

PPFDT

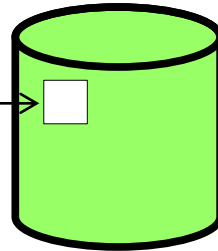
	Fd flags	File ptr
0		
1		
2		
3		
4		
5		

System Wide File Table

	File offset	Status flags	Inode pointer
0			
12			
54			
75			
93			

Inode Table

	Type	Pmns	Owner	Locks	....
13					
233					



File Descriptor	Purpose	POSIX Name	stdio Stream
0	Standard input	STDIN_FILENO	stdin
1	Standard output	STDOUT_FILENO	stdout
2	Standard error	STDERR_FILENO	stderr

# File Descriptor to File Contents

- Each process in UNIX has an associated PPFDT, whose size is equal to the number of files that a process can open simultaneously
- File descriptor is an integer returned by `open()` system call, and is used as an index in the PPFDT
- File descriptor is used in `read()`, `write()` and `close()` system call
- Kernel uses this descriptor to index the PPFDT, which contain a pointer to another table called **System Wide File Table**
- In the **System Wide File Table**, other than some information there is another pointer to a table called **Inode Table**
- The inode table contains a unique Inode to every unique file on disk

# Standard Descriptors in UNIX / Linux

- Three files are automatically opened for every process to read its input from and to send its output and error messages to.
- These files are called standard files:
  - **0 - Standard Input (stdin).** Default input to a program is from the user terminal (keyboard), **if no file name is given.**
  - **1 - Standard Output (stdout).** A simple program's output normally goes to the user terminal (monitor), **if no file name is given.**
  - **2 - Standard Error (stderr).** Default output of error messages from a program normally goes to the user terminal, **if no file name is given.**
- These numbers are called File Descriptors - System calls use them to refer to files.

# Examples - File Handling

```
int main() {  
    char buff[256];  
    read(0, buff, 255);  
    write(1, buff, 255);  
    return 0;  
}
```

```
int main() {  
    char buff[256];  
    while(1) {  
        int n = read(0, buff, 255);  
        write(1, buff, n);  
    }  
    return 0;  
}
```



# Example - File Handling

```
int main() {  
    char buff[2000];  
    int fd = open ("/etc/passwd", O_RDONLY);  
    int n;  
    for(;;) {  
        n = read(fd, buff, 1000);  
        if (n <= 0) {  
            close(fd);  
            exit(-n);  
        }  
        write(1, buff, n);  
    }  
    return 0;  
}
```

# Example - File Handling

```
int main() {  
    int n;  
    char buff[1024];  
    int  
    fd=open("file.txt",O_CREAT|O_TRUNC|O_RDWR,0666);  
    for(;;){  
        n = read(0, buff, 1023);  
        if (n <= 0) {  
            printf("Error in reading kb.\n");  
            exit(-n);  
        }  
        write(fd, buff, n);  
    }  
    close(fd);  
    return 0;  
}
```

# **I/O Redirection**

# Stdin and Stdout for Commands

## Example 1:

```
$ cat
This is GR8
This is GR8
<CTRL + D>
$
```

## Example 2:

```
$ sort
rauf
arif
kamal
<CTRL+D>
arif
kamal
rauf
$
```

PPFDT		
Fd flags	File ptr	
0		→ stdin
1		→ stdout
2		→ stderr
3		
4		
5		
OPENMAX-1		

cp is generally preferred for copying files, as it handles metadata and permissions better.

cat <f1 >f2 is a simpler way to duplicate the content but does not preserve the original file's metadata.

# Redirecting Input of a Command (0<)

- By default, **cat** and **sort** commands takes their input from the standard input, i.e. key board. We can detach the key board from stdin and attach some file to it; i.e. **cat** command will now read input from this file and not from the key board

```
# cat 0< f1.txt
```

```
# sort 0< f1.txt
```

```
cat 2>err <f1 >op    error in err if f1 does not exist
```

```
cat <f1 1>op 2>err    error on screen
```

```
cat f1 1>op 2>err    error in file
```

```
cat f1 f2 f3 >operr 2>operr if  
f1 exist and others dont , error  
in err
```

```
cat f1 2>&1
```

```
1>operr
```

```
cat: f1: No such file or  
directory
```

```
cat f1 1>operr 2>&1 error in  
..
```

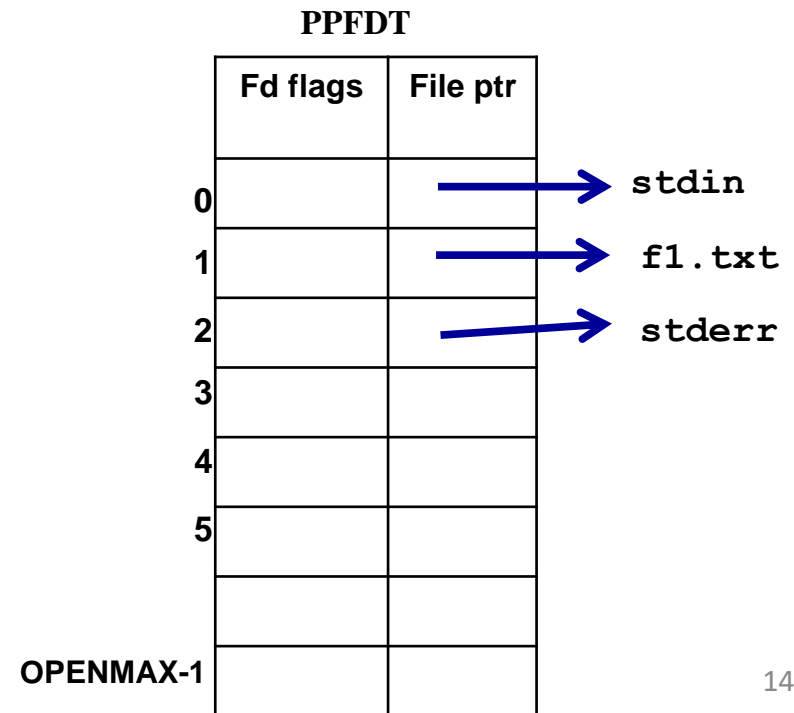
PPFDT

	Fd flags	File ptr
0		→ f1.txt
1		→ stdout
2		→ stderr
3		
4		
5		
OPENMAX-1		

# Redirecting Output of a Command (1>)

Similarly, by default **cat** and **sort** commands sends their outputs to user terminal. We can detach the display screen from stdout and attach a file to it; i.e. **cat** command will now write its output to this file and not to the display screen

```
# cat 1> f1.txt
```






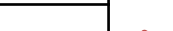
# Redirecting Error of a Command (2>)

Similarly, by default all commands send their error messages on stderr, which is also connected to the VDU. We can detach the VDU from the error stream and can connect it to a file instead. This is called error redirection

`cat nofile.txt 2> errors.txt` error in file

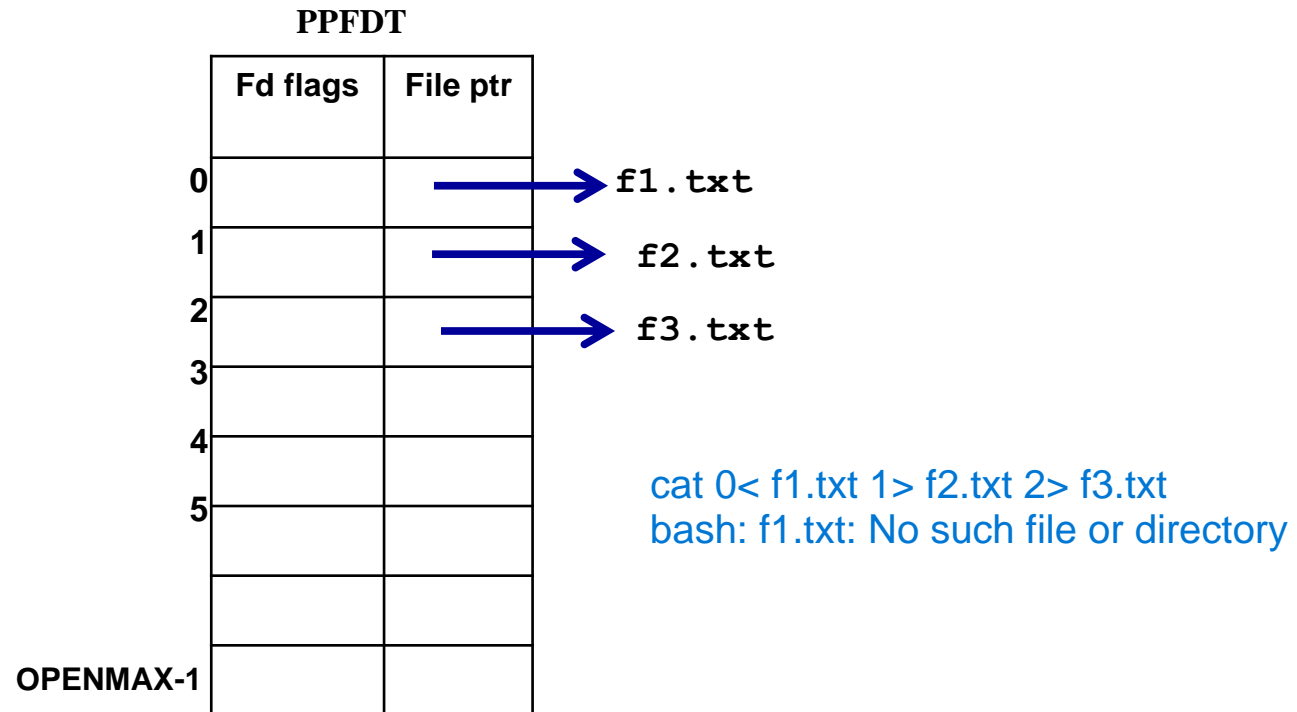
```
# cat nofile.txt 2> errors.txt
```

PPFDT

	Fd flags	File ptr	
0			stdin
1			stdout
2			errors.txt
3			nofile.txt
4			
5			
OPENMAX-1			

# Redirecting Input, Output and Error

```
$ cat 0< f1.txt 1> f2.txt 2> f3.txt
```



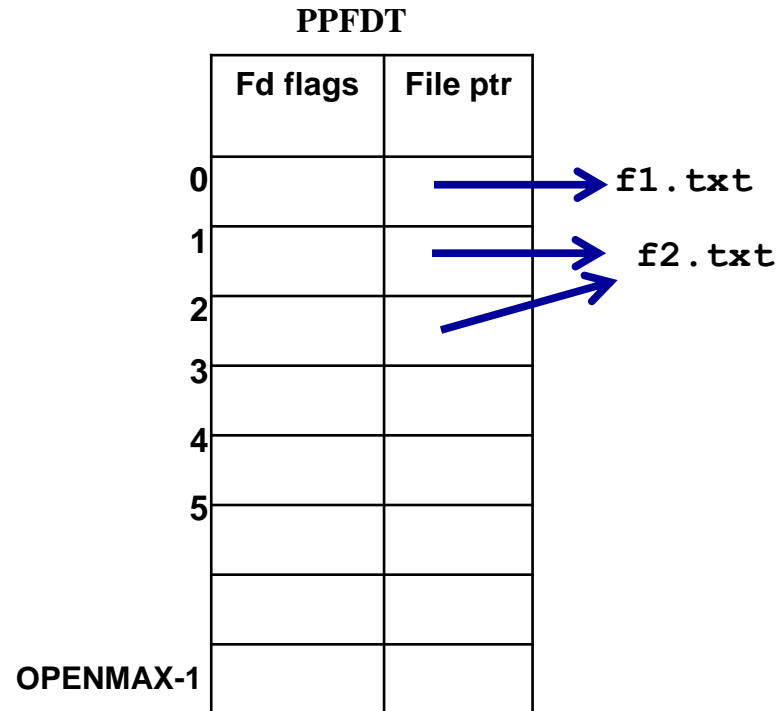


# Duplicating a File Descriptor

```
$ cat 0< f1.txt 1> f2.txt 2>&1
```

```
cat 0< f1.txt 1> f2.txt 2>&1
```

```
bash: f1.txt: No such file or directory
```



# Draw PPFDT of following Commands

- Differentiate between following two commands (if file2 do not exist)

```
$ cat < file1 > file2
```

```
$ cp file1 file2
```

- Differentiate between following two commands (if lab1 do not exist)

```
$ cat lab1.txt 1> output.txt 2> error.txt
```

```
$ cat 0< lab1.txt 1> output.txt 2> error.txt
```

- Differentiate between following two commands

```
$ find /etc/ -name passwd 2> f1 1>&2
```

```
$ find /etc/ -name passwd 2> f1 2>&1
```

if no input redirection is  
performed the whole command  
will be passed

- Explain behavior of following commands

```
$ cat 1> output.txt 0< input.txt 2> error.txt
```

```
$ cat 2> error.txt 1> output.txt 0< input.txt
```

```
$ cat f1.txt 2>&1 1> f2.txt
```

```
$ cat 0< f1.txt 1> f2.txt 2>&1
```

```
$ cat 2>&1 1> f2.txt 0< f1.txt
```

```
$ cat 1> f1.txt 2>&1 0< f1.txt
```

## Quiz 03

Question 1: Assume that file f1, f2 and f3 do not exist. Draw the PPFDT of following command. Will f1 and f2 be created and what will be their contents?

```
$ cat f1 1> f2 2> f3
```

Question 2: Assume that file f1, f2 and f3 do not exist. Draw the PPFDT of following command. Will f1 and f2 be created and what will be their contents?

```
$ cat 0< f1 1> f2 2> f3
```

# **Taxonomy of Inter Process Communication**

# Cooperating Processes

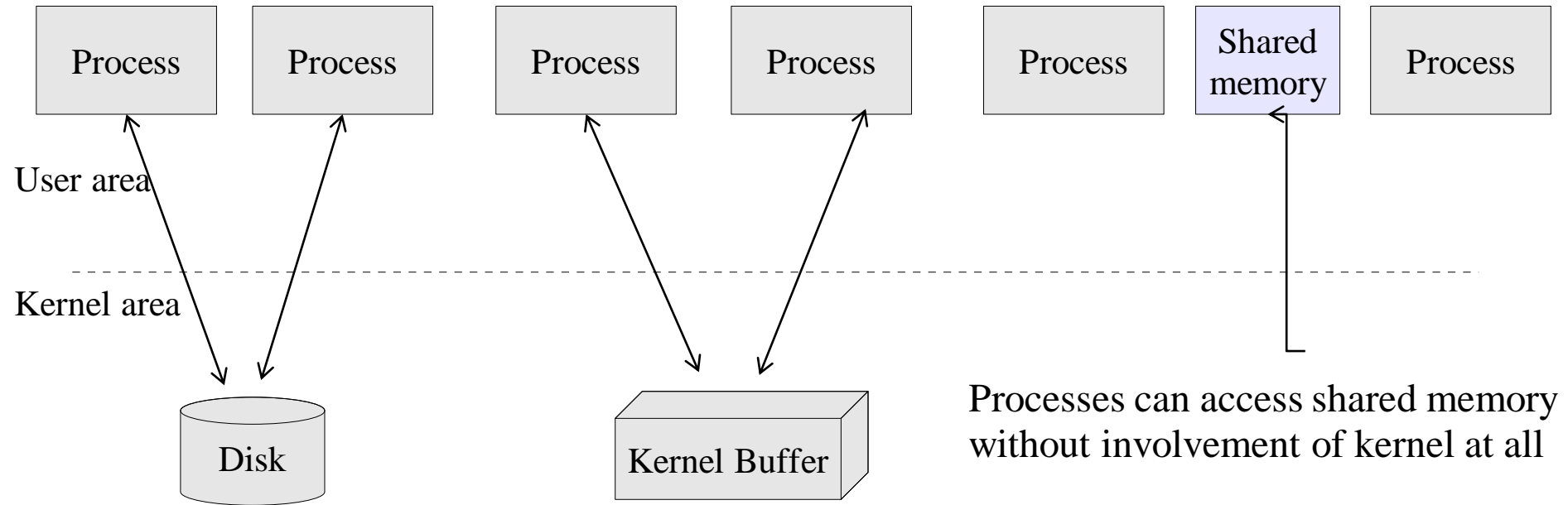
- **Independent process** is a process that cannot affect or cannot be affected by the execution of another process. A process that does not share data with another process is independent
- **Cooperating process** is a process that can affect or can be affected by the execution of another process. A process that share data with other process is a cooperating process
- Advantages of Cooperating processes:
  - Information sharing
  - Computation speed up
  - Modularity
  - Convenience



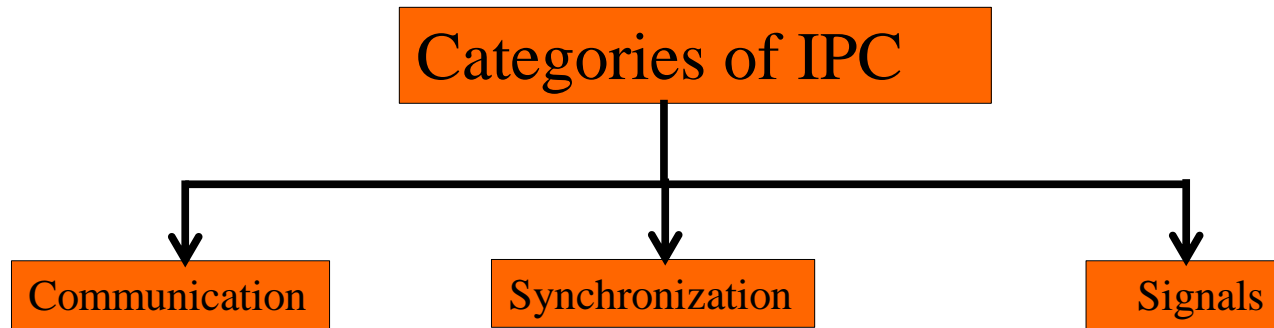
# Application Design

- Option 1: One huge monolithic program that does every thing
- Option 2: Multi\_threaded programs
- Option 3: Multiple programs using fork() that communicate with each other using some form of Inter Process Communication (IPC)

# Ways to Share Information among Processes



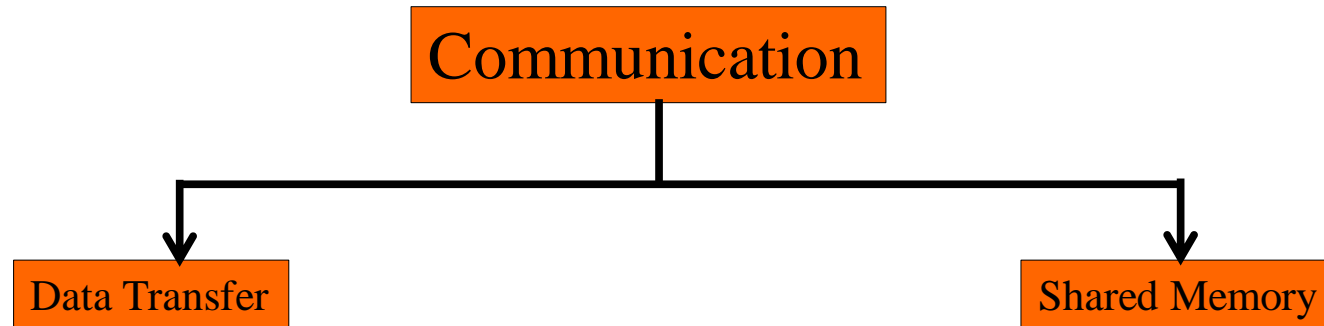
# Taxonomy of IPC



- Communication: These facilities are concerned with exchange of data among cooperating processes
- Synchronization: These facilities are concerned with synchronizing actions among cooperating processes
- Signals: Although signals are primarily for other purposes, they can be used as synchronization primitives in certain circumstances

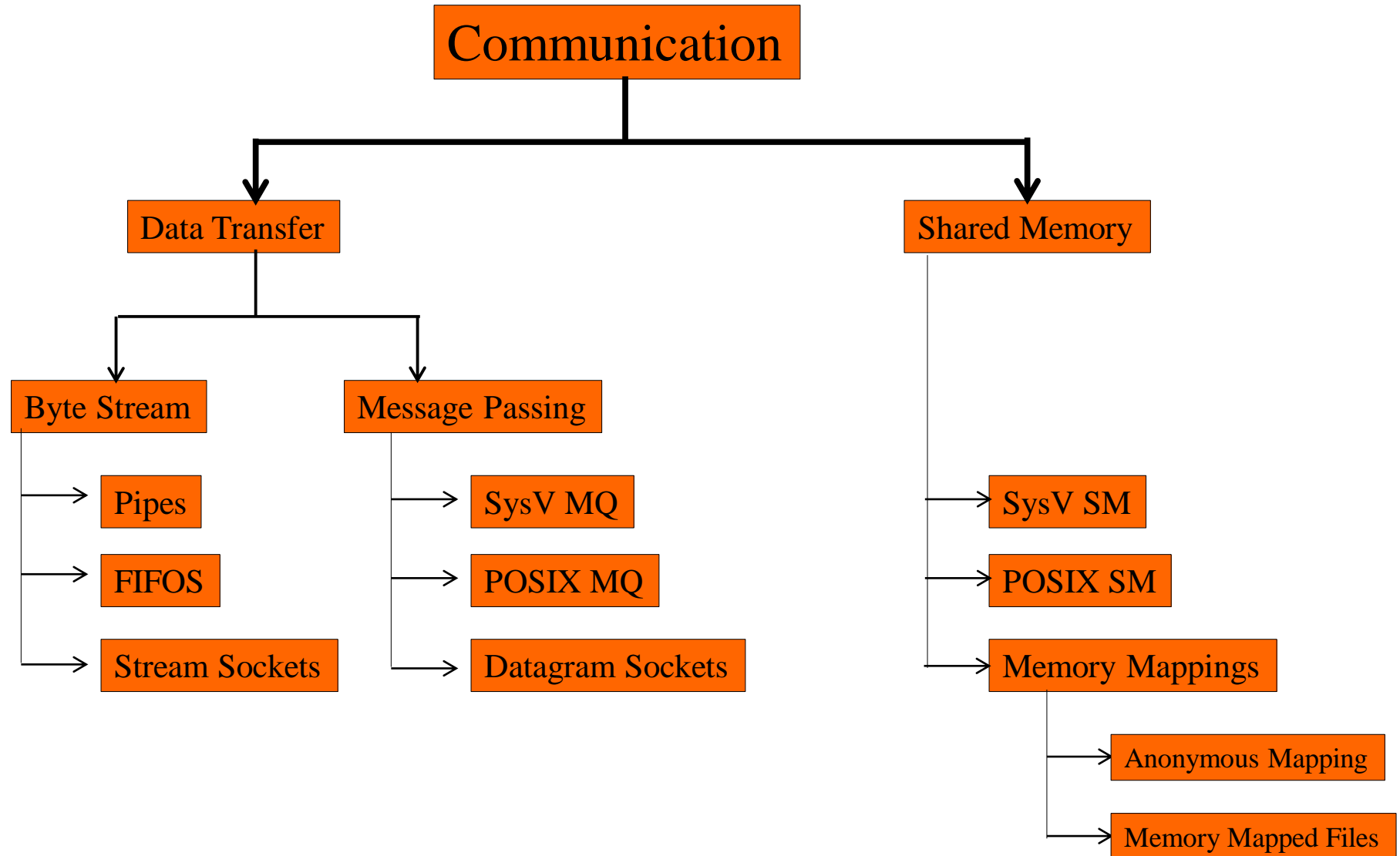


# Taxonomy of IPC

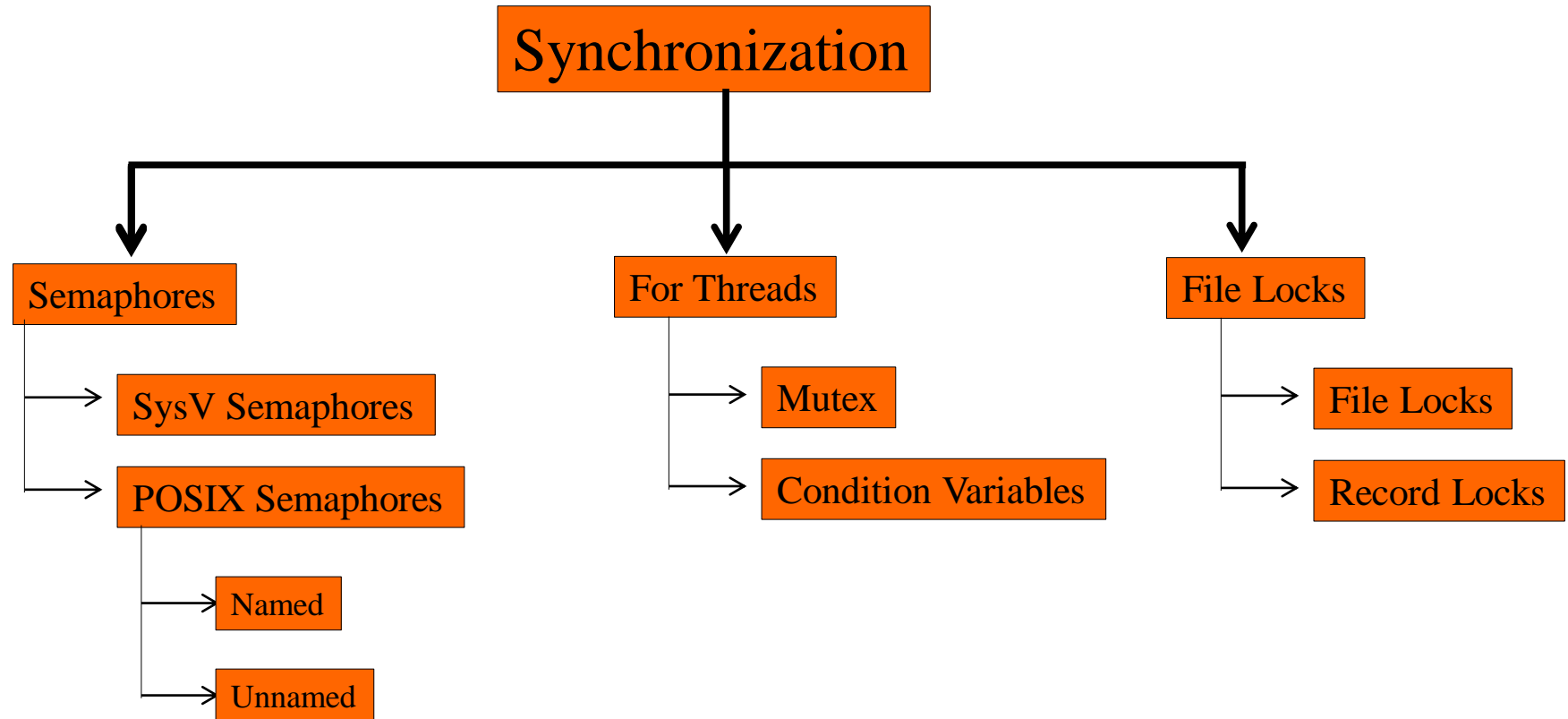


- Data Transfer: One process writes data to the IPC facility and other process reads the data. It has destructive read semantics and synchronization between the reader and writer is implicit
- Shared Memory: A region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In shared memory reading is non-destructive, however synchronization is not implicit rather is the baby of the programmer

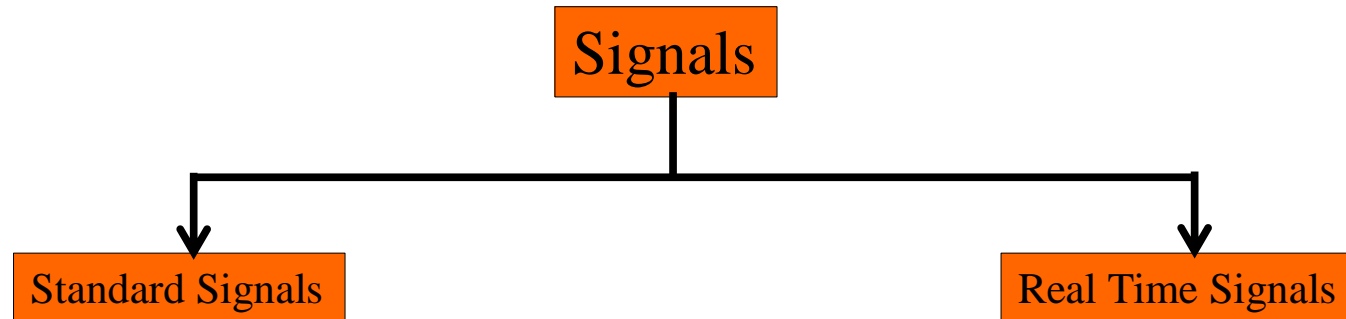
# Taxonomy of IPC



# Taxonomy of IPC



# Taxonomy of IPC



# Persistence of IPC Objects

Process  
Persistence

- .Exists as long as it is held open by a process
- .Pipes and FIFOs
- .TCP, UDP sockets
- .Mutex, condition variables, read write locks
- .POSIX memory based semaphores

Kernel  
Persistence

- .Exists until kernel reboots or IPC objects is explicitly deleted
- .Message Queues, semaphores & shared memory are at least kernel persistent

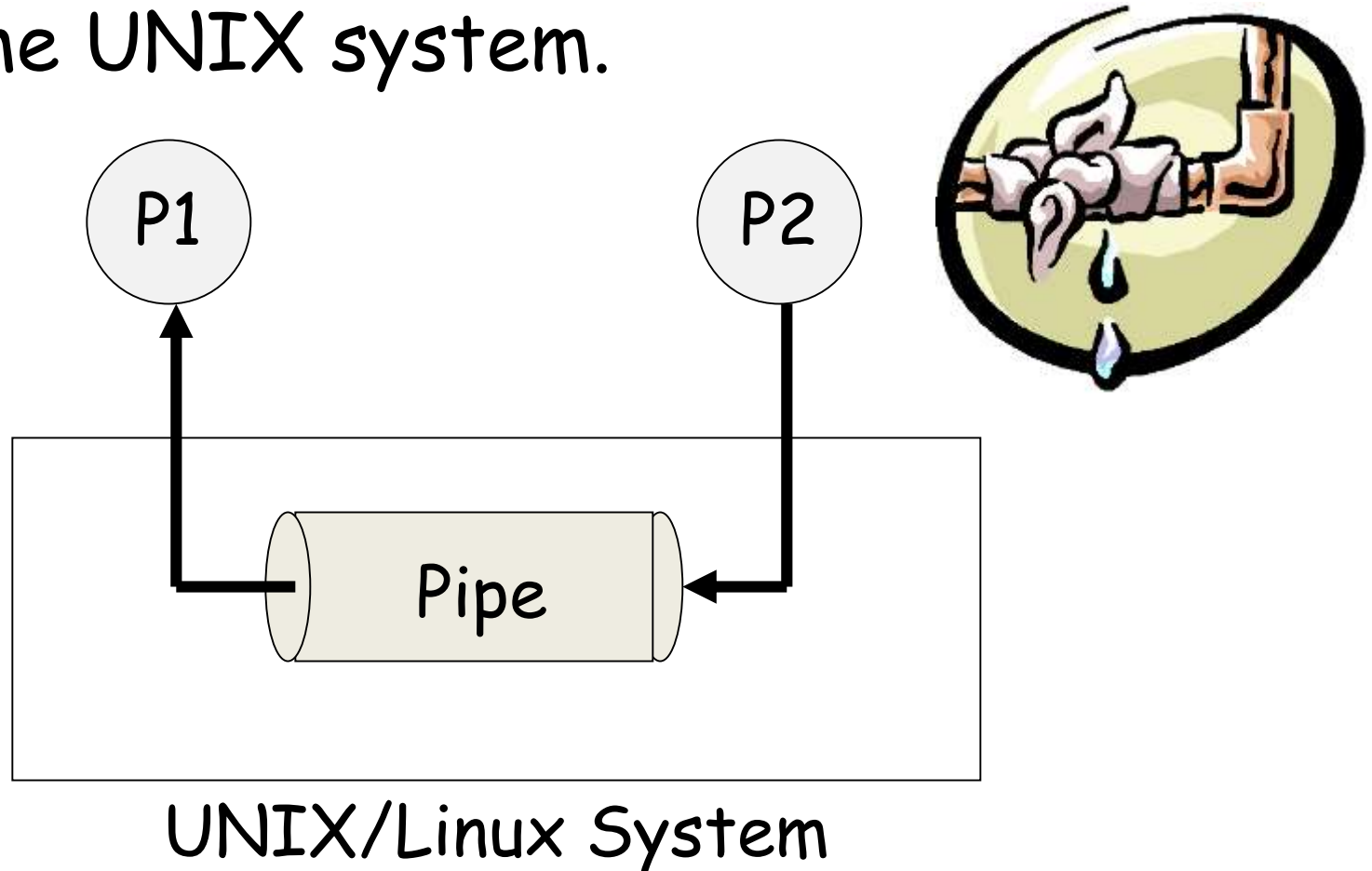
File system  
Persistence

- .Exists until IPC objects is explicitly deleted, or file system crashes
- .Message queues, semaphores & shared memory can be file system persistent if implemented using mapped files

# Use of Pipes on the Shell

# UNIX IPC Tool: Pipes

- **Pipes** are used for communication between related processes (parent-child-sibling) on the same UNIX system.



# UNIX IPC Tool: Pipes

- **History of Pipes:** Pipes history goes back to 3<sup>rd</sup> edition of UNIX in 1973. They have no name and can therefore be used only between related processes. This was corrected in 1982 with the addition of FIFOs
- **Byte stream:** When we say that a pipe is a byte stream, we mean that there is no concept of message boundaries when using a pipe. Each read operation may read an arbitrary number of bytes regardless of the size of bytes written by the writer. Furthermore, the data passes through the pipe sequentially, bytes are read from a pipe in exactly the order they were written. It is not possible to randomly access the data in a pipe using `lseek()`
- **Pipes are unidirectional:** Data can travel only in one direction. One end of the pipe is used for writing, and the other end is used for reading

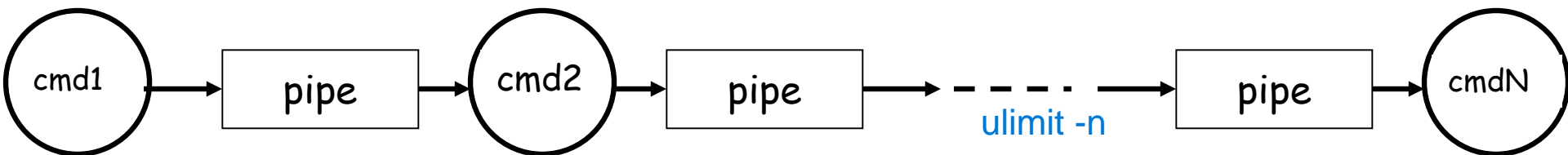


# UNIX IPC Tool: Pipes

- The UNIX system allows **stdout** of a command to be connected to **stdin** of another command using the pipe operator **|**.

cmd1 | cmd2 | cmd3 | ... | cmdN

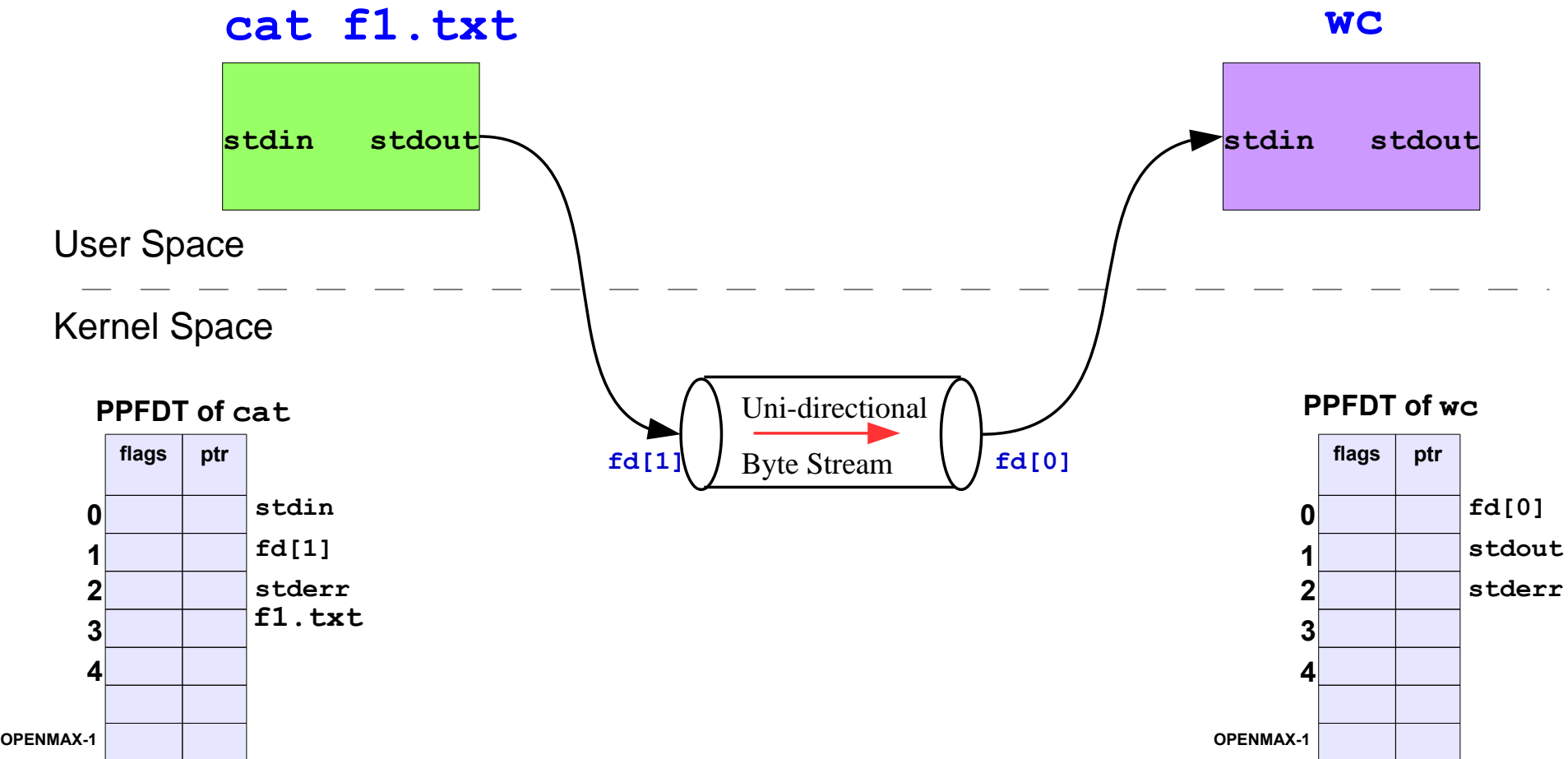
- Stdout of cmd1 is connected to stdin of cmd2, stdout of cmd2 is connected to stdin of cmd3, ... and stdout of cmdN-1 is connected to stdin of cmdN.



Question: On a shell, how many commands can be connected using pipes?

# UNIX IPC Tool: Pipes

```
$ cat f1.txt | wc
```



# UNIX IPC Tool: Pipes

- Example Write a command that displays the contents of **/etc/** directory, one page at a time

```
$ ls -l /etc/ 1> temp
```

```
$ less 0< temp
```

This will display the contents of file temp on screen one page at a time. After this we need to remove the temp file as well.

```
$ rm temp
```

So we needed three commands to accomplish the task, and the command sequence is also extremely slow because file read and write operations are involved.

Lets use the **pipe** symbol

```
$ ls -l /etc/ | less
```

The net effect of this command is the same. It does not use a disk to connect standard output of **ls -l** to standard input of **less** because **pipe** is implemented in the main memory.

# UNIX IPC Tool: Pipes

- Example Write a command that will sort the file friends.txt and display its contents on stdout after removing duplication if any

```
$ sort friends.txt | sort
```

- Example Write a command that will count the number of lines in the man page of ls

```
$ man ls | wc -l
```

- Example Write a command that will count the number of lines containing string ls in the man page of ls

```
$ man ls | grep ls | wc -l
```

Try drawing the above commands pictorially and also draw their respective PPFDTs for better understanding

# UNIX IPC Tool: tee

- **tee** command reads from stdin and writes to stdout

```
$ tee
```

- **tee** command reads from stdin and writes to stdout as well as the two files f1.txt and f2.txt

```
$ tee f1.txt f2.txt
```

- **tee** command is used to redirect the stdout of a command to one or more files, as well as to another command

```
cmd1 | tee file1 ... fileN | cmd2
```

- Stdout of cmd1 is connected to stdin of tee, and **tee** sends its output to files file1, ... fileN and also to stdin of cmd2

```
$ who | tee who.txt
```

# Pipe, tee and I/O Redirection

- Example Write a command that reads a file f1.txt and stores the line containing string pucit in another file f2.txt by using pipes and I/O redirection

```
$ cat f1.txt | grep pucit 1> f2.txt
```

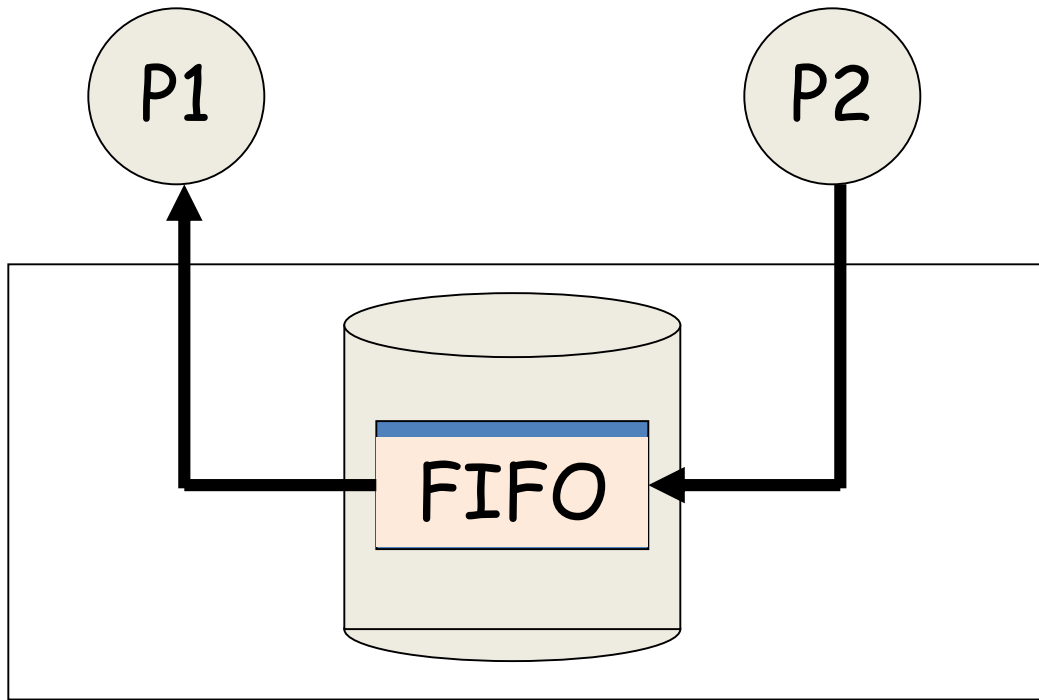
- Example Repeat above command so that the output is also displayed on stdout as well

```
$ cat f1.txt | grep pucit | tee f2.txt | cat
```

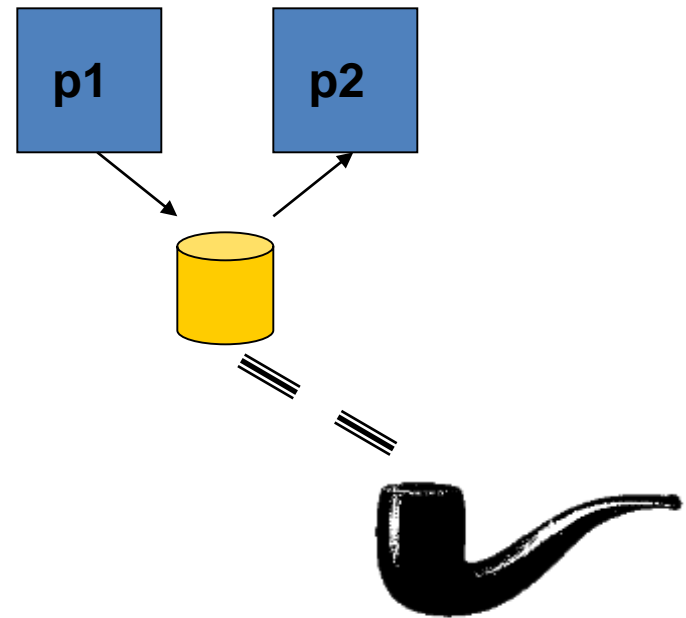
# Use of Named Pipes on the Shell

# UNIX IPC Tool: FIFO

- Named pipes (FIFO) are used for communication between related or unrelated processes on the same UNIX system.



UNIX/Linux System



*Ceci, n'est pas une pipe.*



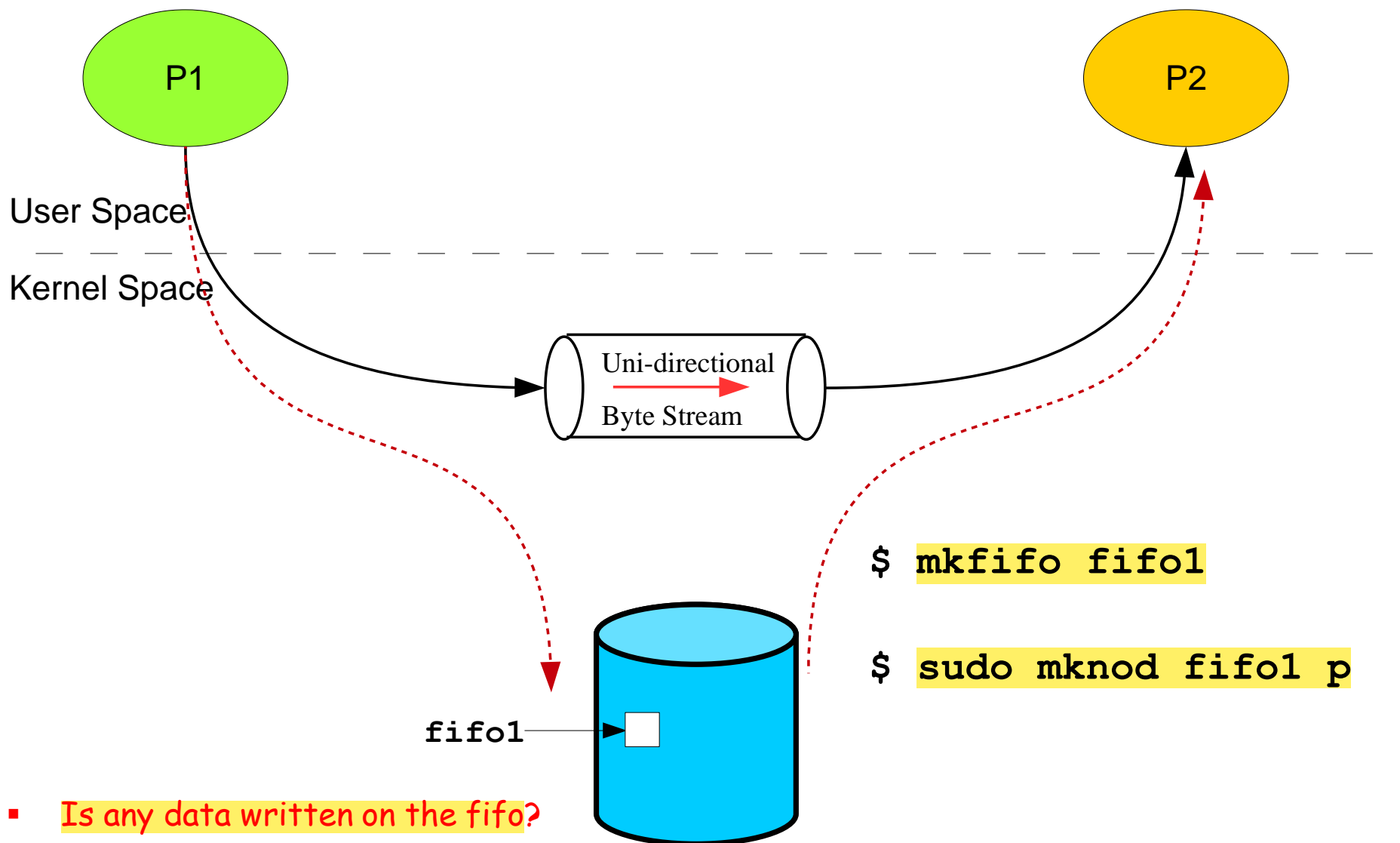
# UNIX IPC Tool: FIFO

- Pipes have no names, and their biggest disadvantage is that they can be only used between processes that have a parent process in common (ignoring descriptor passing)
- UNIX FIFO is similar to a pipe, as it is a one way (half duplex) flow of data. But unlike pipes a FIFO has a path name associated with it allowing unrelated processes to access a single pipe
- FIFOs/named pipes are used for communication between related or unrelated processes executing on the same machine
- A FIFO is created by one process and can be opened by multiple processes for reading or writing. When processes are reading or writing data via FIFO, kernel passes all data internally without writing it to the file system. Thus a FIFO file has no contents on the file system; the file system entry merely serves as a reference point so that processes can access the pipe using a name in the file system

# UNIX IPC Tool: FIFO

```
$ echo "Hello PUCIT" 1> fifo1
```

```
$ cat fifo1
```



```
$ mkfifo fifo1
```

```
$ sudo mknod fifo1 p
```

- Is any data written on the fifo?
- Why echo, or cat blocks while writing or reading on fifo
- What if there are multiple readers on this fifo, who gets the data?

# Use of Signals on the Shell

# Introduction to Signals

- Suppose a program is running in a **while(1)** loop and you press **Ctrl+C** key. The program dies. How does this happens?
  - User presses **Ctrl+C**
  - The **tty** driver receives character, which matches **intr**
  - The **tty** driver calls signal system
  - The signal system sends **SIGINT (2)** to the process
  - Process receives **SIGINT (2)**
  - Process dies
- Actually by pressing **<ctrl+c>**, you ask the kernel to send **SIGINT** to the currently running foreground process. To change the key combination you can use **stty(1)** or **tcsetattr(2)** to replace the current **intr** control character with some other key combination

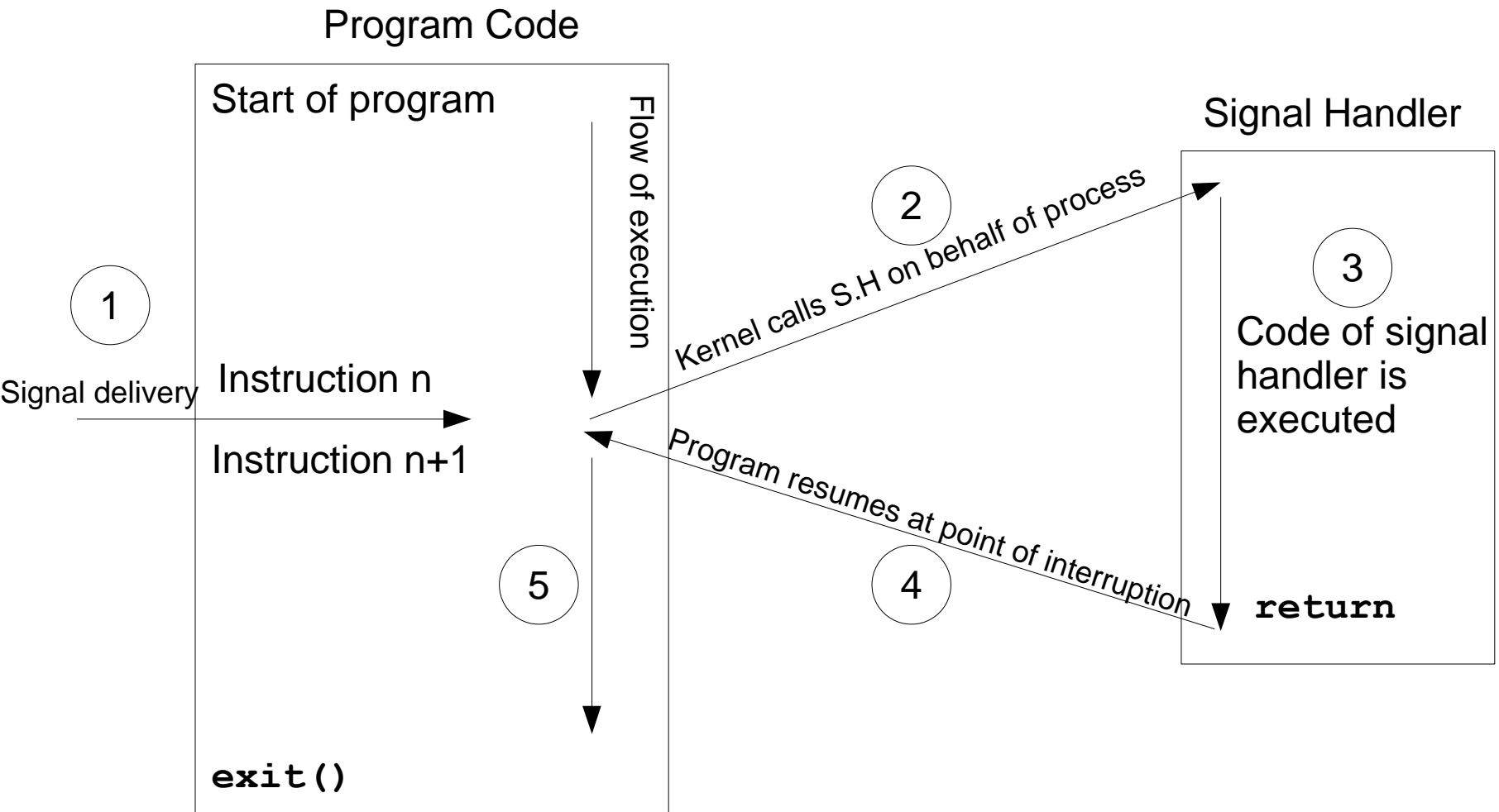
# Introduction to Signals

- Signal is a software interrupt delivered to a process by OS because:
    - The process did something (SIGFPE (8), SIGSEGV (11), SIGILL (4))
    - The user did something (SIGINT (2), SIGQUIT (3), SIGTSTP (20))
    - One process wants to tell another process something (SIGCHLD (17))
  - **Signals are usually used by OS to notify processes that some event has occurred, without these processes needing to poll for the event**
  - Whenever a process receives a signal, it is interrupted from whatever it is doing and forced to execute a piece of code called signal handler. When the signal handler function returns, the process continues execution as if this interruption has never occurred
  - A **signal handler** is a function that gets called when a process receives a signal. Every signal may have a specific handler associated with it. A signal handler is called in *asynchronous mode*. Failing to handle various signals, would likely cause our application to terminate, when it receives such signals
-

# Synchronous and Asynchronous Signals

- Signals may be generated synchronously or asynchronously
- **Synchronous signals** pertains to a specific action in the program and is delivered (unless blocked) during that action. Examples:
  - Most errors generate signals synchronously
  - Explicit request by a process to generate a signal for the same process
- **Asynchronous signals** are generated by the events outside the control of the process that receives them. These signals arrive at unpredictable times during execution. Examples include:
  - External events generate requests asynchronously
  - Explicit request by a process to generate a signal for some other process

# Signal Delivery and Handler Execution



# Signal Numbers and Strings

- Every signal has a symbolic name and an integer value associated with it, defined in `/usr/include/asm-generic/signal.h`
- You can use following shell command to list down the signals on your system:

```
$ kill -l
```

- Linux supports 32 real time signals from SIGRTMIN (32) to SIGRTMAX (63). Unlike standard signals, real time signals have no predefined meanings, are used for application defined purposes. The default action for an un-handled real time signal is to terminate the receiving process. See also `$ man 7 signal`



# Sending Signals to Processes

A signal can be issued in one of the following ways:

- **Using Key board**

- `<Ctrl+c>` gives `SIGINT (2)`
- `<Ctrl+\>` gives `SIGQUIT (3)`
- `<Ctrl+z>` gives `SIGTSTP (20)`

- **Using Shell command**

- `kill -<signal> <PID>` OR `kill -<signal> %<jobID>`
- If no signal name or number is specified then default is to send `SIGTERM (15)` to the process
- Do visit man pages for `jobs`, `ps`, `bg` and `fg` commands
- `bg` gives `SIGTSTP (20)` while `fg` gives `SIGCONT (18)`

- **Using `kill()` or `raise()` system call**

- **Implicitly by a program** (division by zero, issuing an invalid addr, termination of a child process)

# Signal Dispositions

Upon delivery of a signal, a process carries out one of the following default actions, depending on the signal: `[$man 7 signal]`

1. The signal is **ignored**; that is, it is discarded by the kernel and has no effect on the process. (The process never even knows that it occurred)
2. The process is **terminated** (killed). This is sometimes referred to as abnormal process termination, as opposed to the normal process termination that occurs when a process terminates using `exit()`
3. A **core dump** file is generated, and the process is terminated. A core dump file contains an image of the virtual memory of the process, which can be loaded into a debugger in order to inspect the state of the process at the time that it terminated
4. The process is **stopped**—execution of the process is suspended (SIGSTOP, SIGTSTP)
5. Execution of the process is **resumed** which was previously stopped (SIGCONT, SIGCHLD)

# Signal Dispositions

- Each signal has a current disposition which determines how the process behave when the OS delivers it the signal
- If you install no signal handler, the run time environment sets up a set of default signal handlers for your program. Different default actions for signals are:

<b>TERM</b>	Abnormal termination of the program with <code>_exit( )</code> i.e, no clean up. However, status is made available to <code>wait()</code> & <code>waitpid()</code> which indicates abnormal termination by the specified signal
<b>CORE</b>	Abnormal termination with additional implementation dependent actions, such as creation of core file may occur
<b>STOP</b>	Suspend/stop the execution of the process
<b>CONT</b>	Default action is to continue the process if it is currently stopped

# Important Signals (Default Behavior: Term)

## **SIGHUP (1)**

Informs the process when the user who run the process logs out. When a terminal disconnect (hangup) occurs, this signal is sent to the controlling process of the terminal. A second use of SIGHUP is with daemons. Many daemons are designed to respond to the receipt of SIGHUP by reinitializing themselves and rereading their configuration files.

## **SIGINT (2)**

When the user types the terminal interrupt character (usually <Control+C>, the terminal driver sends this signal to the foreground process group. The default action for this signal is to terminate the process.

## **SIGKILL (9)**

This is the sure kill signal. It can't be blocked, ignored, or caught by a handler, and thus always terminates a process.

## **SIGPIPE (13)**

This signal is generated when a process tries to write to a pipe, a FIFO, or a socket for which there is no corresponding reader process. This normally occurs because the reading process has closed its file descriptor for the IPC channel

## **SIGALRM (14)**

The kernel generates this signal upon the expiration of a real-time timer set by a call to `alarm()` or `setitimer()`

## **SIGTERM (15)**

Used for terminating a process and is the default signal sent by the kill command. Users sometimes explicitly send the SIGKILL signal to a process, however, this is generally a mistake. A well-designed application will have a handler for SIGTERM that causes the application to exit gracefully, cleaning up temporary files and releasing other resources beforehand. Killing a process with SIGKILL bypasses SIGTERM handler.

# Important Signals (Default Behavior: Core)

<b>SIGQUIT (3)</b>	When the user types the quit character (Control+\) on the keyboard, this signal is sent to the foreground process group. Using SIGQUIT in this manner is useful with a program that is stuck in an infinite loop or is otherwise not responding. By typing Control-\ and then loading the resulting core dump with the gdb debugger and using the backtrace command to obtain a stack trace, we can find out which part of the program code was executing
<b>SIGILL (4)</b>	This signal is sent to a process if it tries to execute an illegal (i.e., incorrectly formed) machine-language instruction module
<b>SIGFPE (9)</b>	Generate by floating point Arithmetic Exception
<b>SIGSEGV (11)</b>	Generated when a program makes an invalid memory reference. A memory reference may be invalid because the referenced page doesn't exist (e.g., it lies in an unmapped area somewhere between the heap and the stack), the process tried to update a location in read-only memory (e.g., the program text segment or a region of mapped memory marked read-only), or the process tried to access a part of kernel memory while running in user mode. In C, these events often result from dereferencing a pointer containing a bad address. The name of this signal derives from the term segmentation violation

# Important Signals

## Default Behavior: Stop

<b>SIGSTOP (19)</b>	This is the sure stop signal. It can't be blocked, ignored, or caught by a handler; thus, it always stops a process
<b>SIGTSTP (20)</b>	This is the job-control stop signal, sent to stop the foreground process group when the user types the suspend character (usually <code>&lt;Control+Z&gt;</code> ) on the keyboard.. The name of this signal derives from “terminal stop”

## Default Behavior: Cont

<b>SIGCHLD (17)</b>	This signal is sent (by the kernel) to a parent process when one of its children terminates (either by calling <code>exit()</code> or as a result of being killed by a signal). It may also be sent to a process when one of its children is stopped or resumed by a signal
<b>SIGCONT (18)</b>	When sent to a stopped process, this signal causes the process to resume (i.e., to be rescheduled to run at some later time). When received by a process that is not currently stopped, this signal is ignored by default. A process may catch this signal, so that it carries out some action when it resumes

# Ignoring Signals & Writing Signal Handlers

- trap command can be used to ignore the signals

```
$ trap " 2
```

```
$ sleep 1000
```

```
<CTRL + C>
```

- Similarly trap command can be used to mention a different signal handler to a signal.

```
$ trap 'date' 2
```

```
$<CTRL+C> Thu Sep 27 09:38:22 PKT 2018
```

# Masking of Signals

- A signal is generated by some event. Once generated, a signal is later delivered to a process, which then takes some action in response to the signal. Between the time it is generated and the time it is delivered, a signal is said to be **pending**. Normally, a pending signal is delivered to a process as soon as it is next scheduled to run, or immediately if the process is already running (e.g., if the process sent a signal to itself). There can be at most one pending signal of any particular type, i.e., standard signals are not queued
- Sometimes, however, we need to ensure that a segment of code is not interrupted by the delivery of a signal. To do this, we can add a signal to the **process's signal mask**—a set of signals whose delivery is currently blocked. If a signal is generated while it is masked/blocked, it remains pending until it is later unmasked or unblocked (removed from the signal mask)

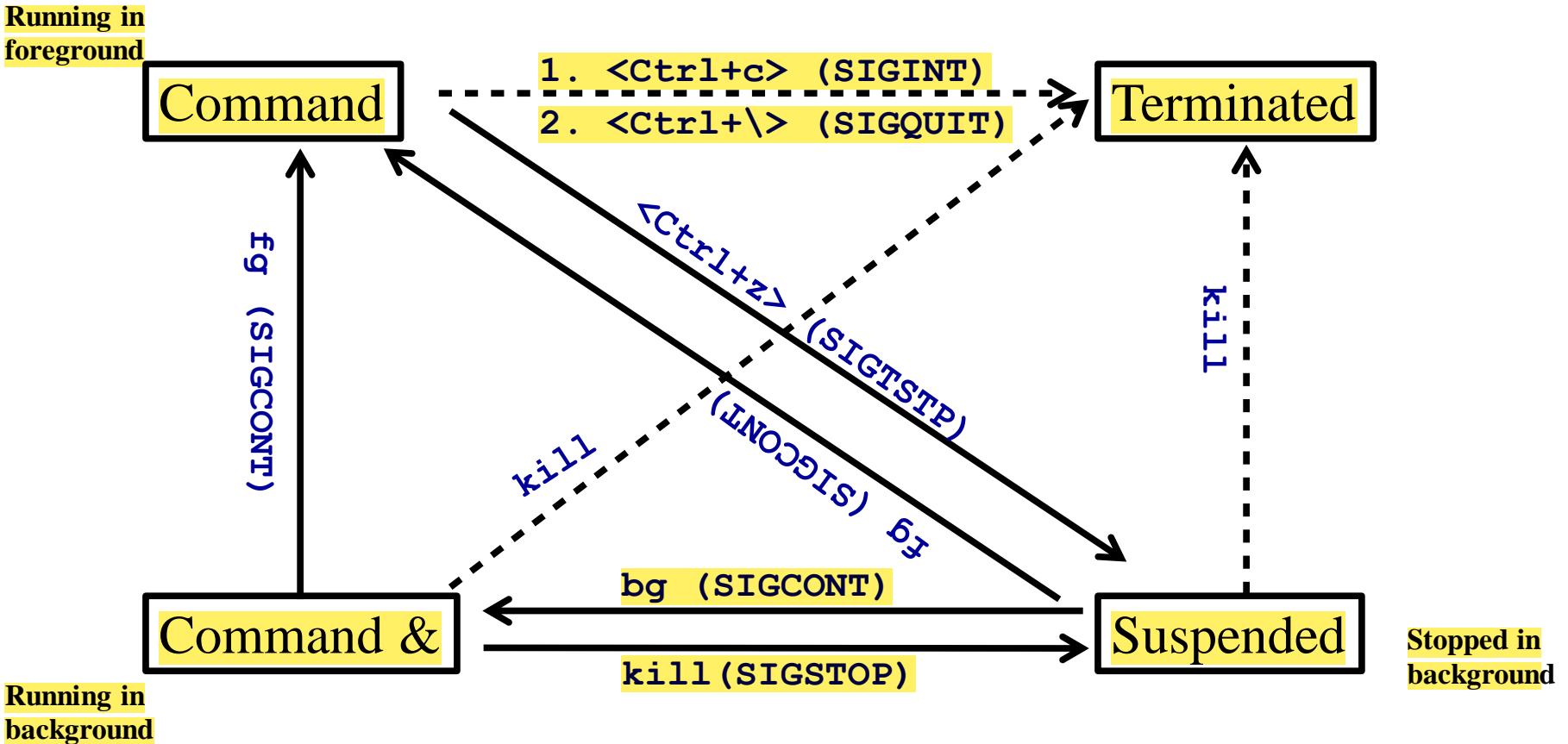


# **Job Control States**

# Foreground and Background Processes

- On a system, there is always one foreground process that is holding the terminal (VDU and keyboard). Example is vim or may be a less program
- On a system, there can be many processes which are running in the background. They usually neither need user input nor requires to give output on the monitor. Example is an audio player playing an audio song, or a find program searching a file from a huge file system. Such programs run in the background so that the terminal is available to the user for execution of other commands
- On GUI a user can simply minimize an application and run other applications, while on a CLI we need to make use of commands to move processes in these two states

# Job Control States



### 1. Command: ``$ cat < file1 > file2`` and ``$ cp file1 file2``

If `file2` does not exist\*\*:

- `cat < file1 > file2`:

- If `file1` exists, it will create `file2` and write the contents of `file1` into it.
- If `file1` does not exist, `cat` will produce an error message, and `file2` will not be created.

- `cp file1 file2`:

- If `file1` exists, it will copy `file1` to `file2` (creating `file2` if it does not exist).
- If `file1` does not exist, `cp` will return an error saying "No such file or directory," and `file2` will not be created.

### 2. Command: ``$ cat lab1.txt 1> output.txt 2> error.txt`` and ``$ cat 0< lab1.txt 1> output.txt 2> error.txt``

If `lab1.txt` does not exist\*\*:

- `cat lab1.txt 1> output.txt 2> error.txt`:

- `cat` attempts to read `lab1.txt`.
- Since `lab1.txt` does not exist, it will generate an error message which will be redirected to `error.txt`.

- `cat 0< lab1.txt 1> output.txt 2> error.txt`:

- This is equivalent to the first command, but it specifies `lab1.txt` as `stdin` explicitly.
- Since `lab1.txt` does not exist, `cat` will again generate an error message redirected to `error.txt`.

3. Command: ``$ find /etc/ -name passwd 2> f1 1>&2`` and ``$ find /etc/ -name passwd 2> f1 2>&1``

If `f1` does not exist\*\*:

- `find /etc/ -name passwd 2> f1 1>&2`:

- `2> f1` redirects `stderr` (error messages) to `f1`.
- `1>&2` redirects `stdout` to wherever `stderr` is currently going (in this case, `f1`).
- If `f1` does not exist, it will be created, and both `stdout` and `stderr` will be written there.
- If `f1` exists, its contents will be overwritten with any output or error messages from the `find` command.

- `find /etc/ -name passwd 2> f1 2>&1`:

- `2> f1` redirects `stderr` to `f1`.
- `2>&1` then redirects `stderr` to `stdout`, meaning both are sent to `f1`.
- If `f1` does not exist, it will be created, and both outputs will be written there. If it does exist, it will be overwritten with the new output.

### 4. Behavior of each command when `f1` exists and does not exist

Let's consider the commands individually:

1. `cat 1> output.txt 0< input.txt 2> error.txt`:

- If `input.txt` exists\*\*: `cat` reads from `input.txt` and writes its contents to `output.txt`, while any errors go to `error.txt`.
- If `input.txt` does not exist\*\*: `cat` will produce an error message, which will be redirected to `error.txt`. `output.txt` will be created (or overwritten) but will remain empty.

2. `cat 2> error.txt 1> output.txt 0< input.txt`:

- If `input.txt` exists\*\*: Same as above. `cat` reads `input.txt`, outputs to `output.txt`, and errors (if any) go to `error.txt`.
- If `input.txt` does not exist\*\*: An error is generated, redirected to `error.txt`. `output.txt` is created (or overwritten) and remains empty.

3. `cat f1.txt 2>&1 1> f2.txt`:

- If `f1.txt` exists\*\*: `cat` reads `f1.txt`, but `2>&1` redirects `stderr` to `stdout`, and then `1> f2.txt` redirects `stdout` to `f2.txt`. Only `stdout` is written to `f2.txt`; `stderr` does not go to `f2.txt`.
- If `f1.txt` does not exist\*\*: An error is generated. Since `stderr` was redirected to `stdout`, the error does not go to `f2.txt` and will display on the terminal instead.

4. `cat 0< f1.txt 1> f2.txt 2>&1`:

- If `f1.txt` exists\*\*: `cat` reads `f1.txt`, `stdout` goes to `f2.txt`, and `stderr` also goes to `f2.txt` (because of `2>&1`).
- If `f1.txt` does not exist\*\*: `cat` produces an error message, and since `stderr` is redirected to `stdout`, the error message will be written to `f2.txt`.

5. `cat 2>&1 1> f2.txt 0< f1.txt`:

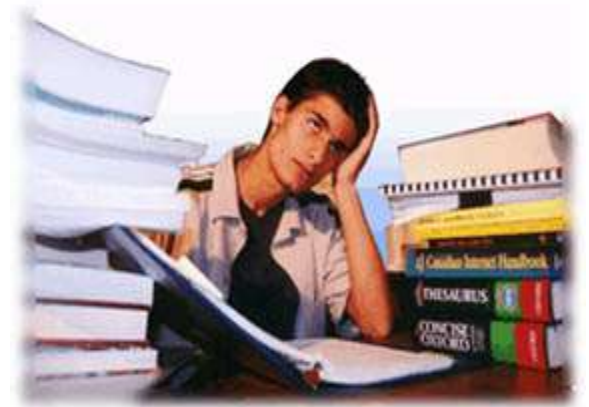
- If `f1.txt` exists\*\*: `cat` reads `f1.txt`. Here, `2>&1` redirects `stderr` to `stdout`, but `1> f2.txt` then redirects `stdout` to `f2.txt`, leaving `stderr` to display on the terminal.
- If `f1.txt` does not exist\*\*: The error message goes to the terminal, not to `f2.txt`, because of the way redirection is ordered.

6. `cat 1> f1.txt 2>&1 0< f1.txt`:

- If `f1.txt` exists\*\*: This creates an infinite loop where `cat` tries to read from and write to `f1.txt` simultaneously. This typically leads to rapid overwriting, and `f1.txt` will fill with empty content or garbage until the command is stopped.

- If `f1.txt` does not exist\*\*: The command creates `f1.txt` as an empty file, then attempts to read from it. Since it's empty, nothing happens. No error is produced, but `f1.txt` remains empty.

# We're done for now, but Todo's for you after this lecture...



- Go through the slides and Book Sections: 3.4, 3.6.3
- Go through Unix The Text Book Sections: 12.1 to 12.15
- Practice all the commands discussed in slides
- Get ready for Lab Quiz

If you have problems visit me in counseling hours. . . .