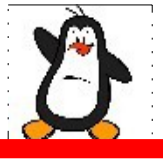


Video Lecture # 24
Overview of UNIX IPC
&
Signals on the Shell

Course: SYSTEM PROGRAMMING

Instructor: Arif Butt

Punjab University College of Information Technology (PUCIT)
University of the Punjab

Today's Agenda

- Ways to Share Information Among UNIX Processes
- Taxonomy of InterProcess Communication
  - Communication
  - Synchronization
  - Signals
- Persistence of IPC Objects
- Overview of Signals Concepts
- Signal Handling on BASH Shell
- Important Signals and their disposition
- Ignoring signals and writing signal handlers (on the shell)

Introduction to UNIX IPC

Application Design

Option 1

One huge monolithic program that does every thing

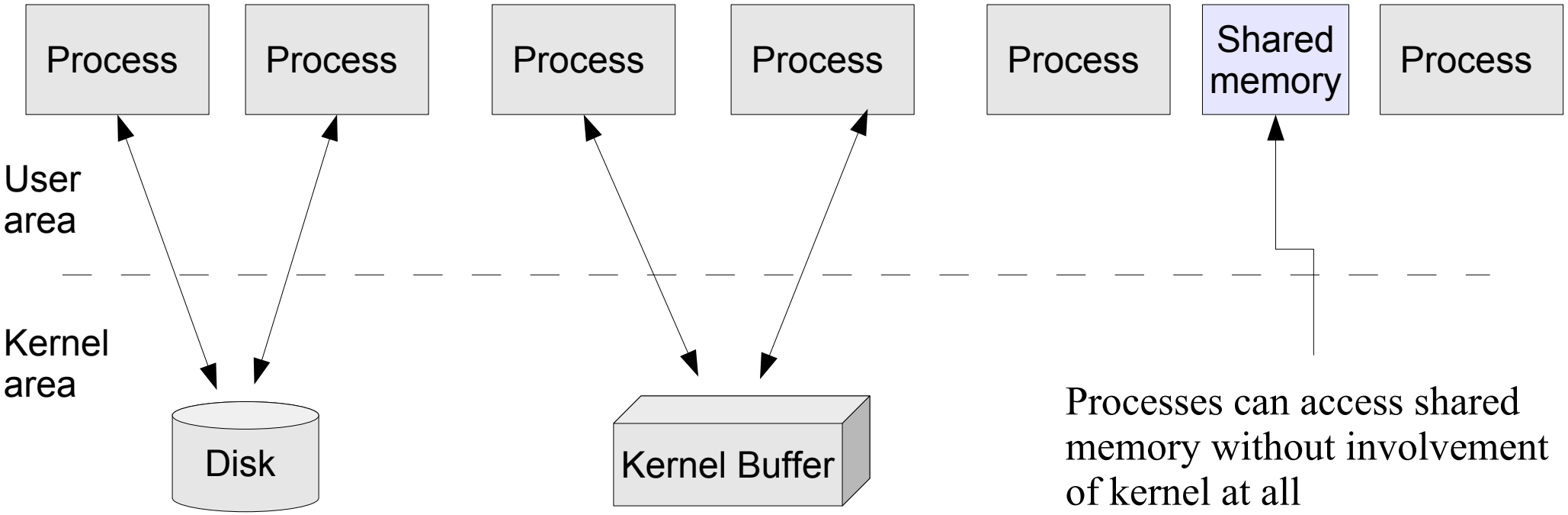
Option 2

Multi_threaded programs

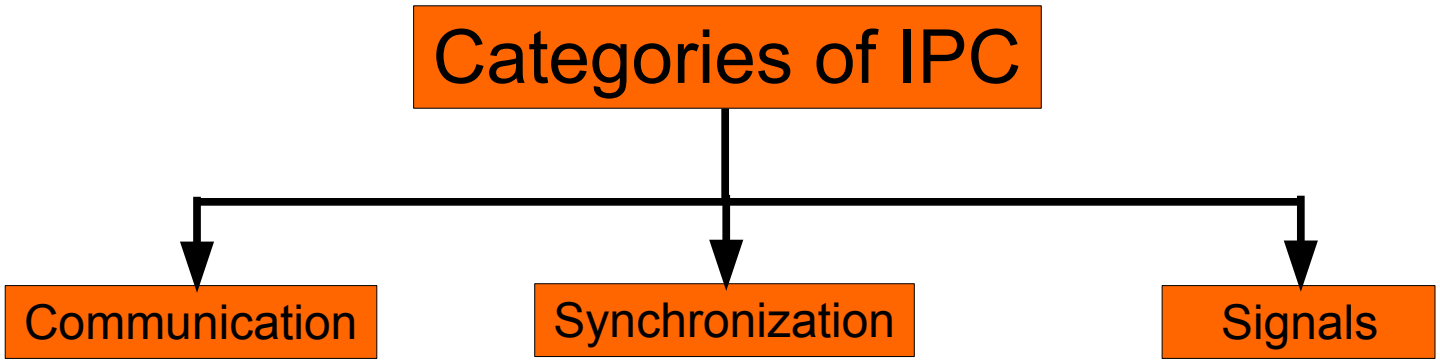
Option 3

Multiple programs using `fork()` that communicate with each other using some form of Inter Process Communication (IPC)

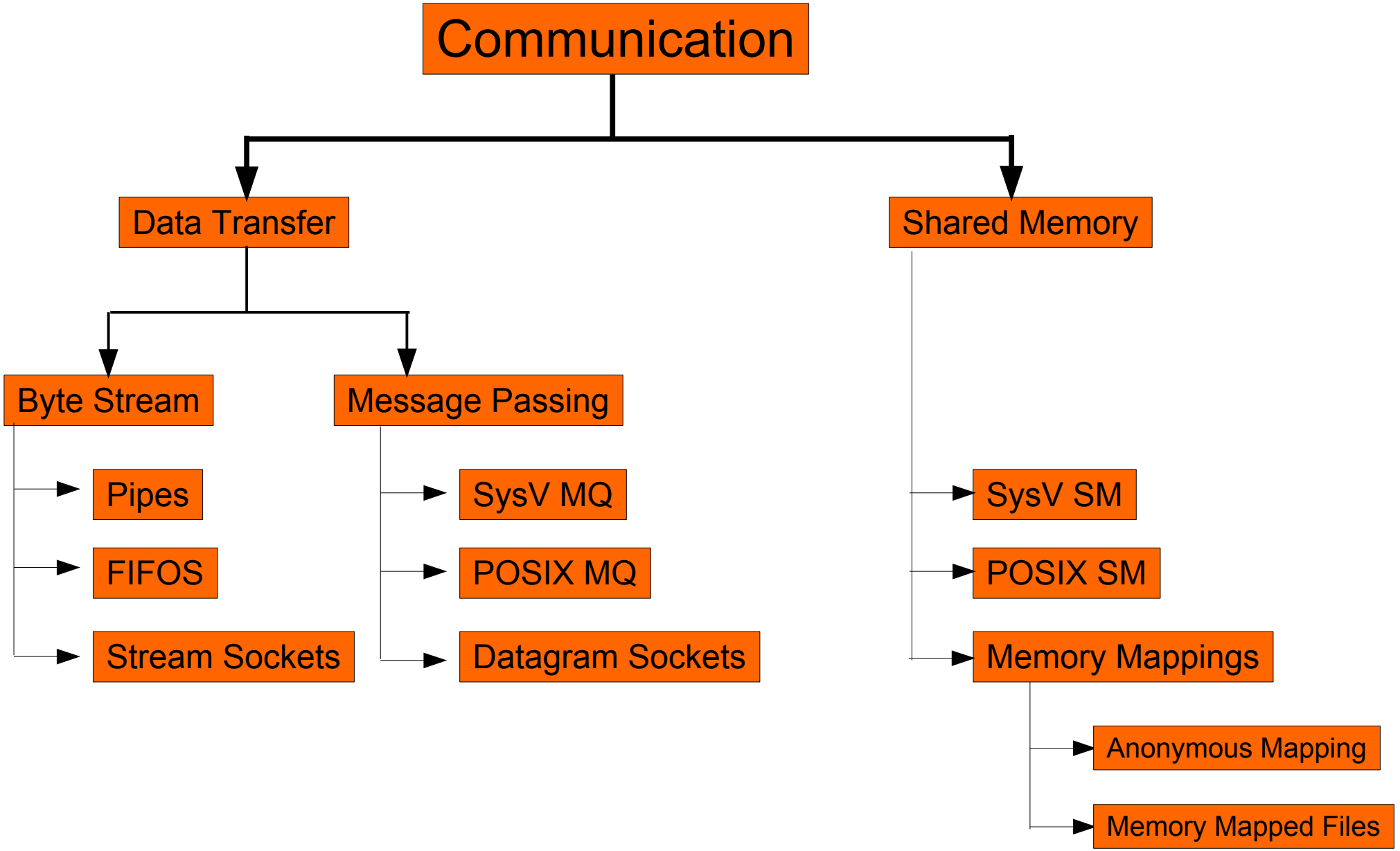
Ways to Share Information b/w UNIX Processes



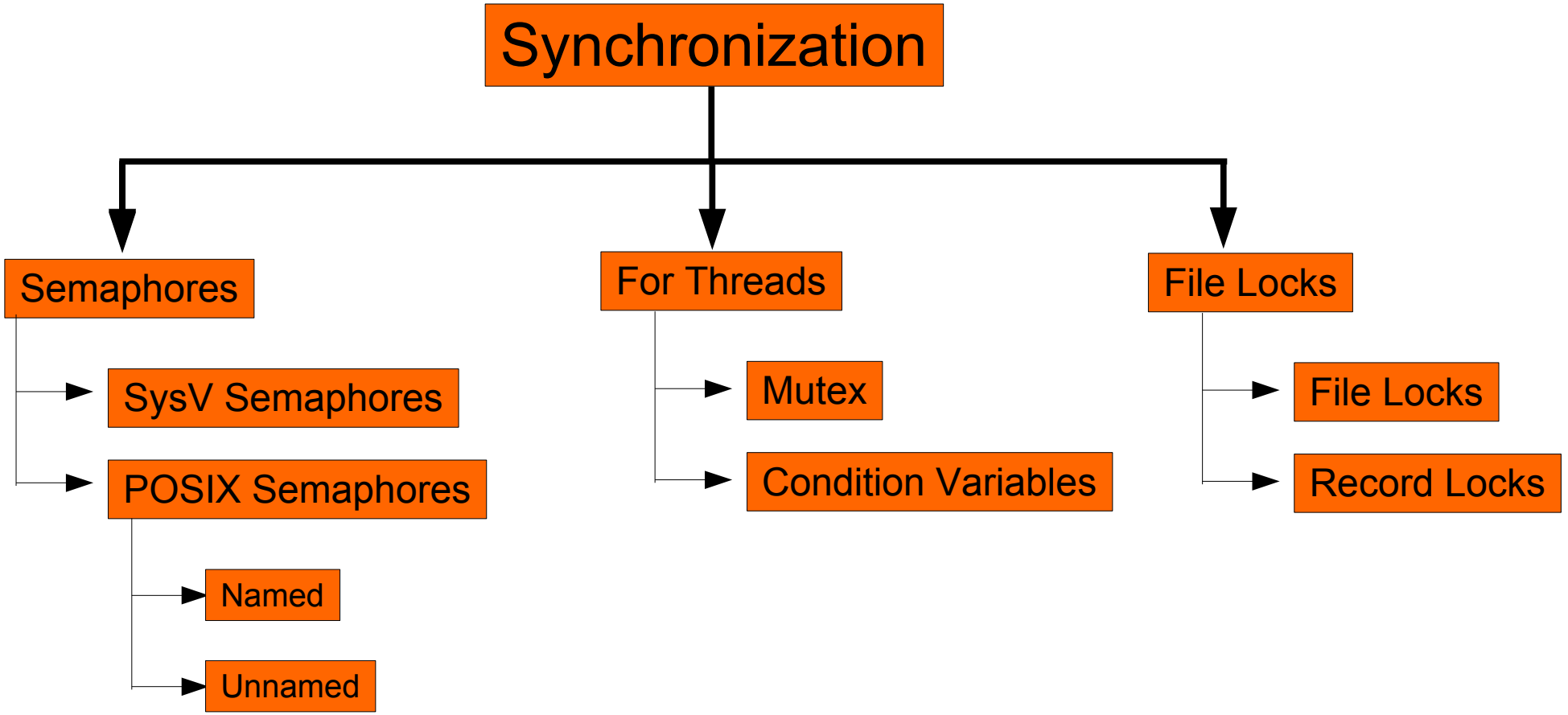
Taxonomy of IPC



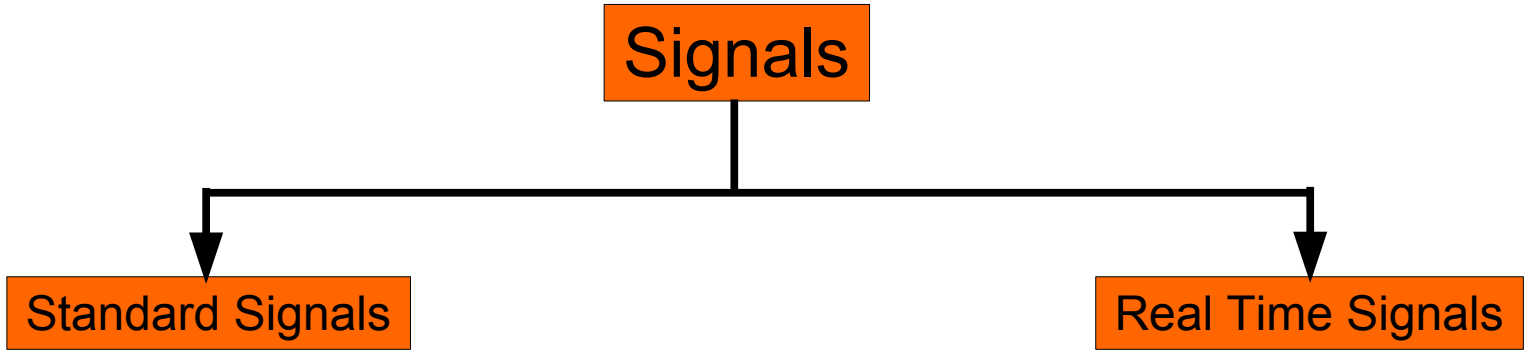
Taxonomy of IPC

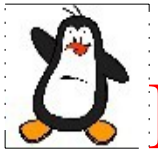


Taxonomy of IPC (cont...)



Taxonomy of IPC (cont...)





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

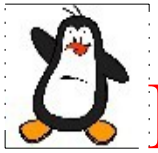
Kernel Persistence

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

File system Persistence

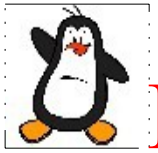
- Exists until IPC objects is explicitly deleted, or file system crashes
- Message queues, semaphores & shared memory can be file system persistent if implemented using mapped files

Overview of Signals



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



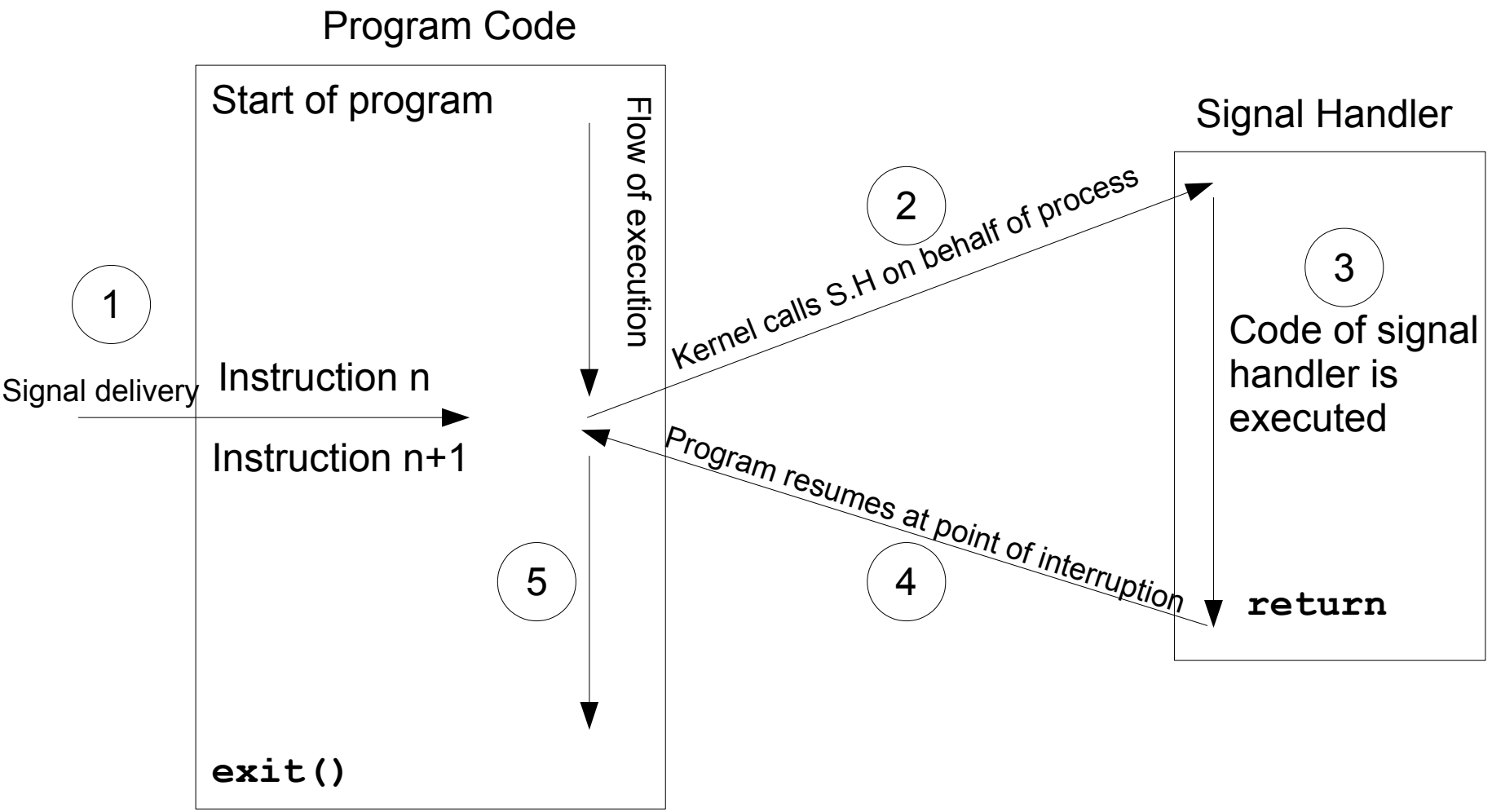
Introduction to Signals (cont...)

- 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 (**SIGCHILD (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 & 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 Handling on the Shell
Proof of Concept

I hear & I forget;
I see & I remember;
I do & I understand.

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 Disposition

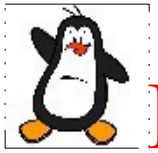
Upon delivery of a signal, a process carries out one of the following default actions, depending on the signal: **`[$man 7 signal]`**

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 Disposition (cont...)

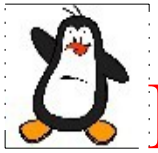
- Each signal has a current disposition which determines how the process behave when the OS delivers it the signal
- If you install no signal handler, the run time environment sets up a set of default signal handlers for your program. Different default actions for signals are:

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



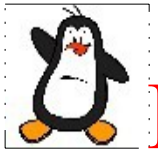
Important Signals (Default Behavior: Term)

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)

SIGQUIT(3)	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
SIGILL(4)	This signal is sent to a process if it tries to execute an illegal (i.e., incorrectly formed) machine-language instruction module
SIGFPE(9)	Generate by floating point Arithmetic Exception
SIGSEGV(11)	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 (cont...)

Default Behavior: Stop

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

Default Behavior: Cont

SIGCHILD(17)	This signal is sent (by the kernel) to a parent process when one of its children terminates (either by calling `exit()` 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
SIGCONT(18)	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

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)

Things To Do

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