Operating Systems

Lecture 4.1

Overview of Linux IPC and Signal Handling

Instructor: Muhammad Arif Butt, PhD

Lecture Agenda

- Taxonomy of Inter Process Communication
- Overview of Signals
- Synchronous vs Asynchronous Signals
- Signal Disposition
- Passing signals using shell commands `kill, killall, pkill, trap`
- Signal Handling using `kill(), alarm(), pause(), raise(), abort()`
- Ignoring / Writing Signal Handlers using `signal()`
- Masking Signals to Avoid Signal Races using `sigprocmask()`

Introduction to UNIX IPC

Independent vs Cooperating Processes

Processes executing concurrently in the operating system can be:

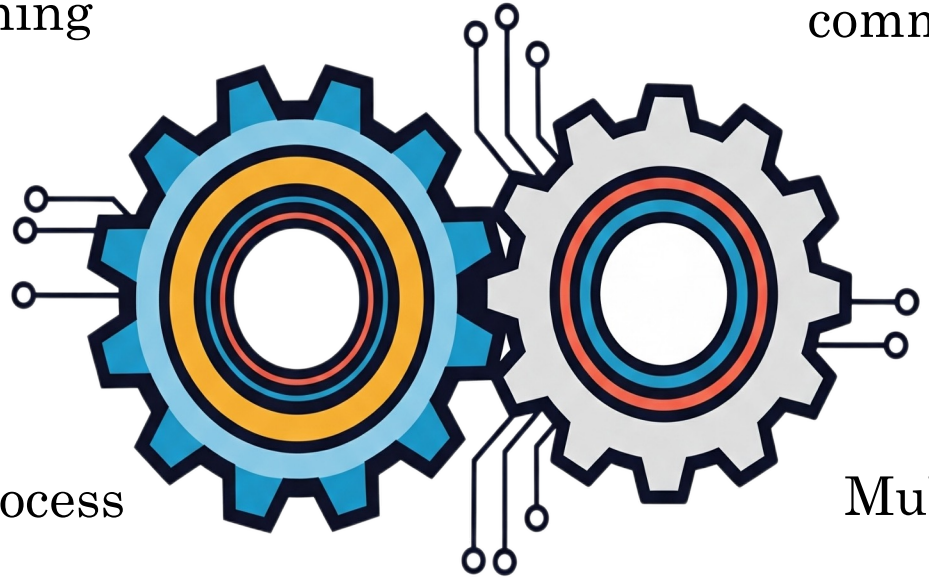
- **Independent Process:** A process that cannot affect or cannot be affected by the execution of another process. A process that does not share data with another process is independent
- **Cooperating Process:** A process that can affect or can be affected by the execution of another process. A process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

Application Design Options

Inter Process Communication (IPC) is a mechanism that allows process to communicate (exchange data) and coordinate with each other

Design using a large monolithic program that does everything

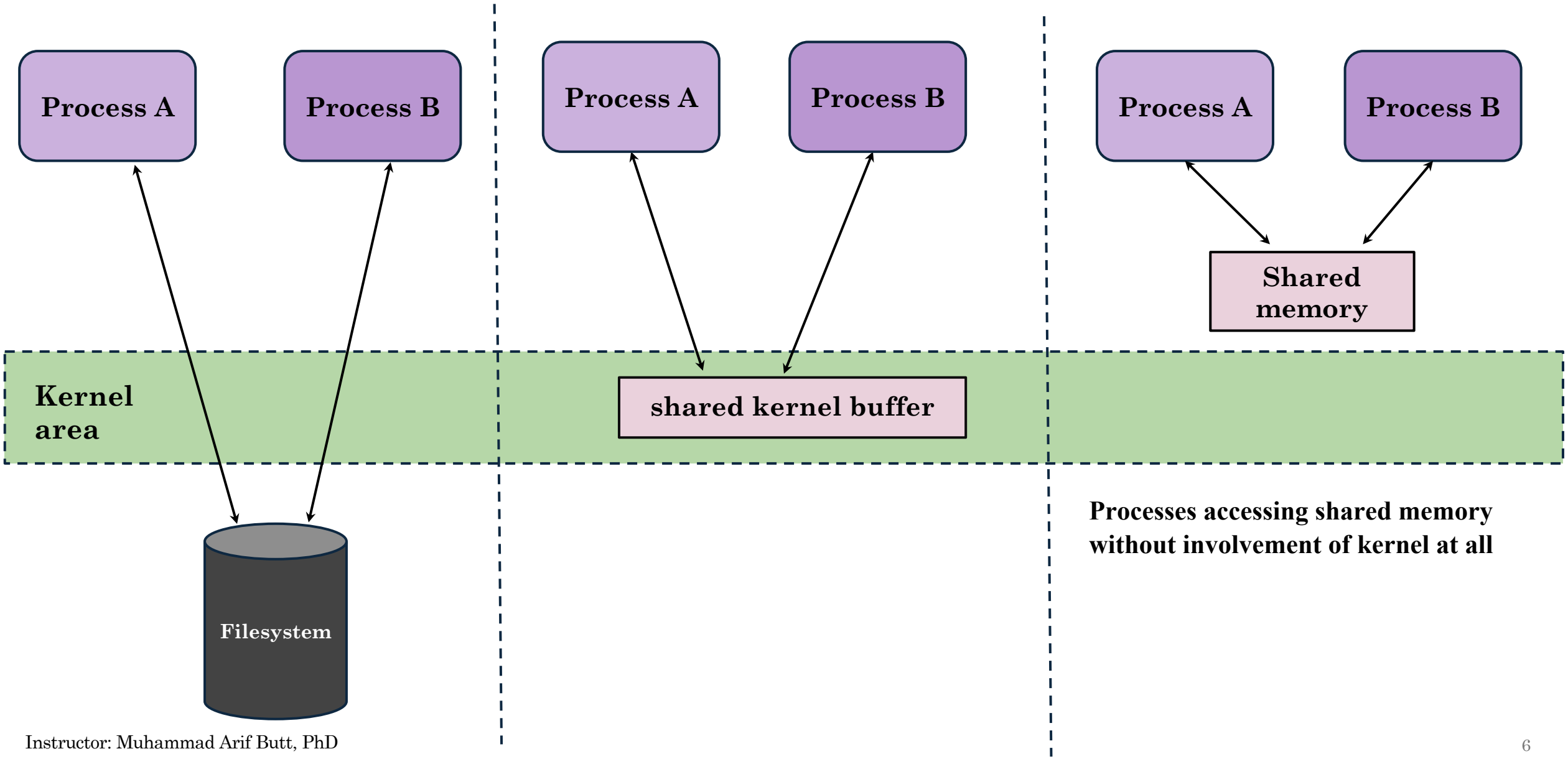
Multiple processes (using `fork`) that that communicate using some form of IPC



A single multi-threaded process

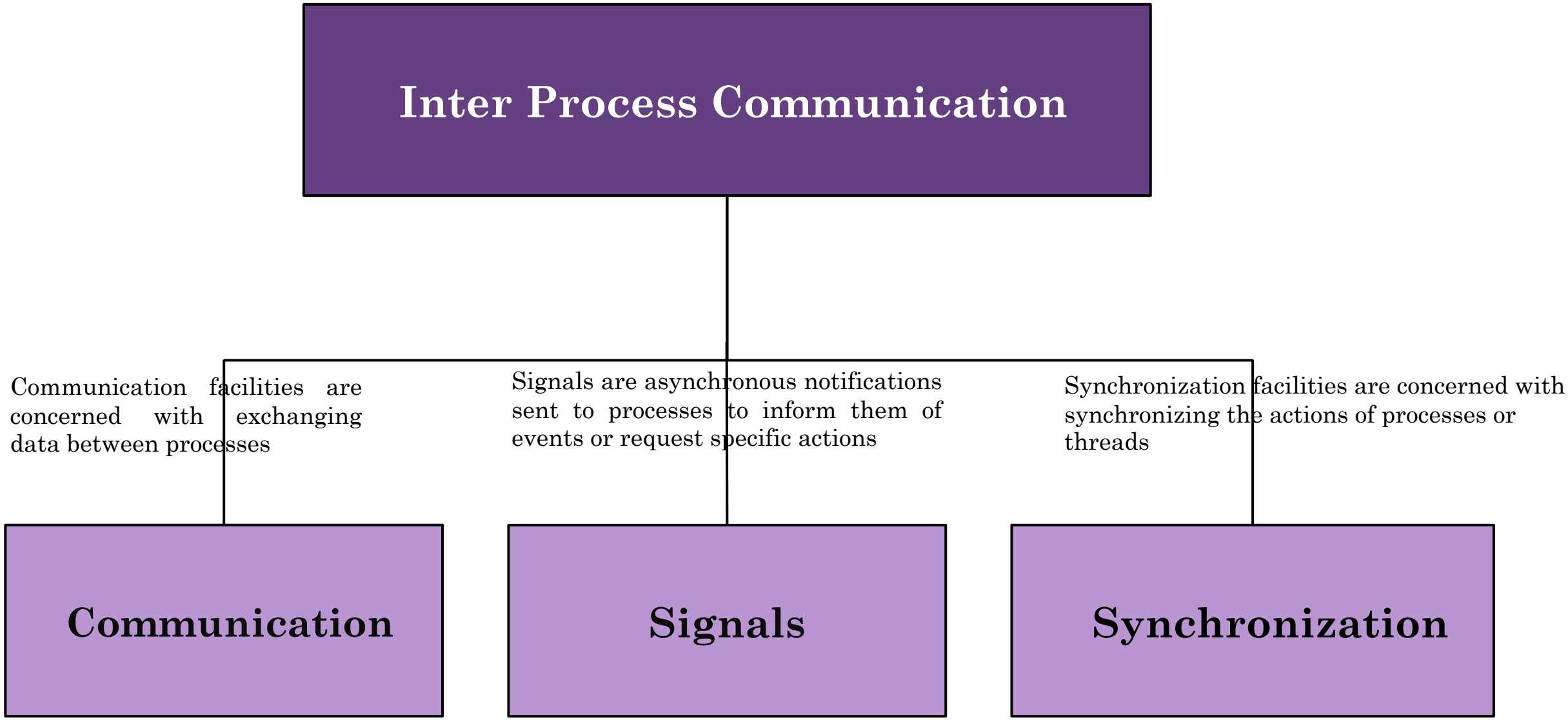
Multiple multi-threaded processes

Ways to share information b/w UNIX processes

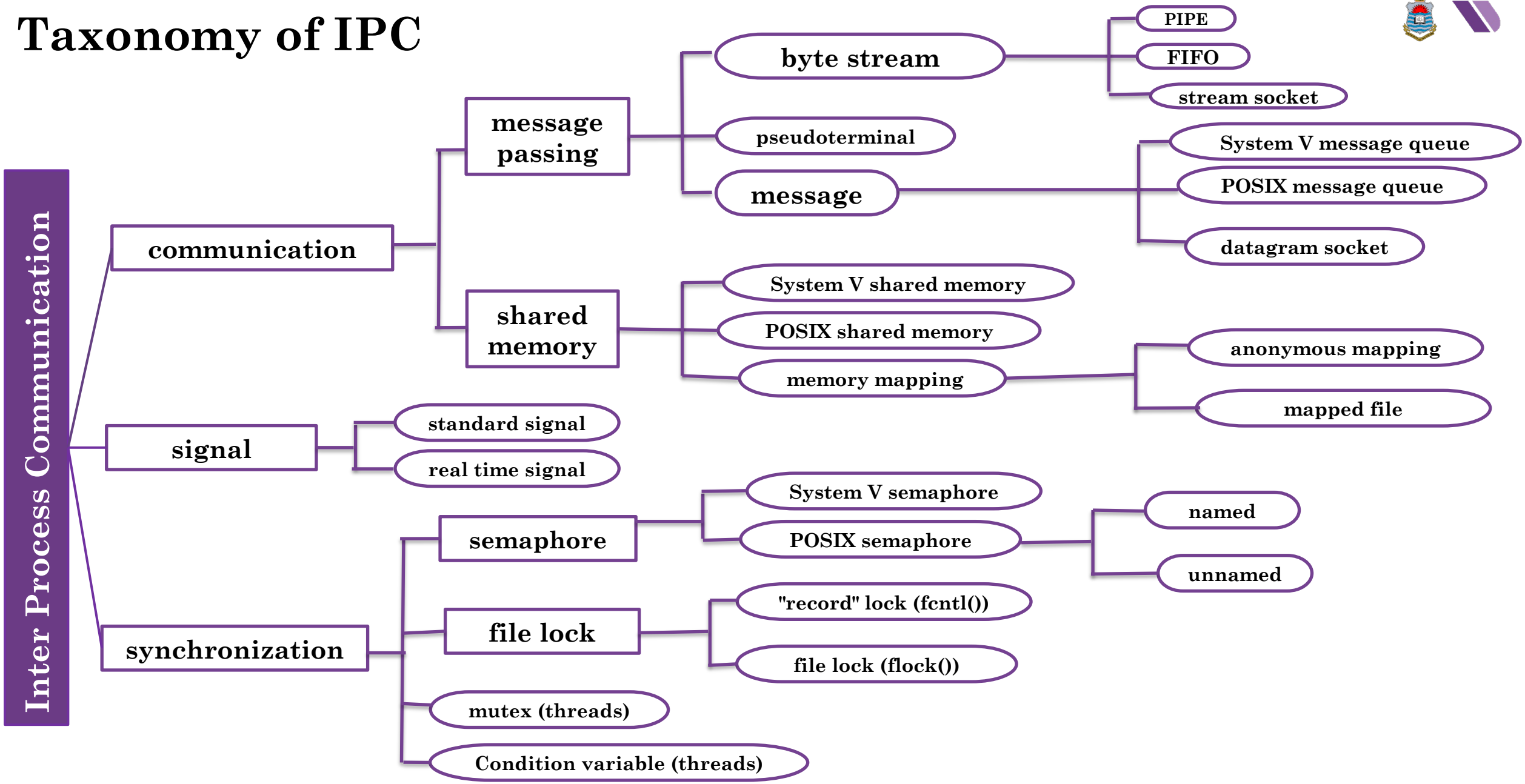


Processes accessing shared memory without involvement of kernel at all

Taxonomy of IPC



Taxonomy of IPC



Communication - Message Passing

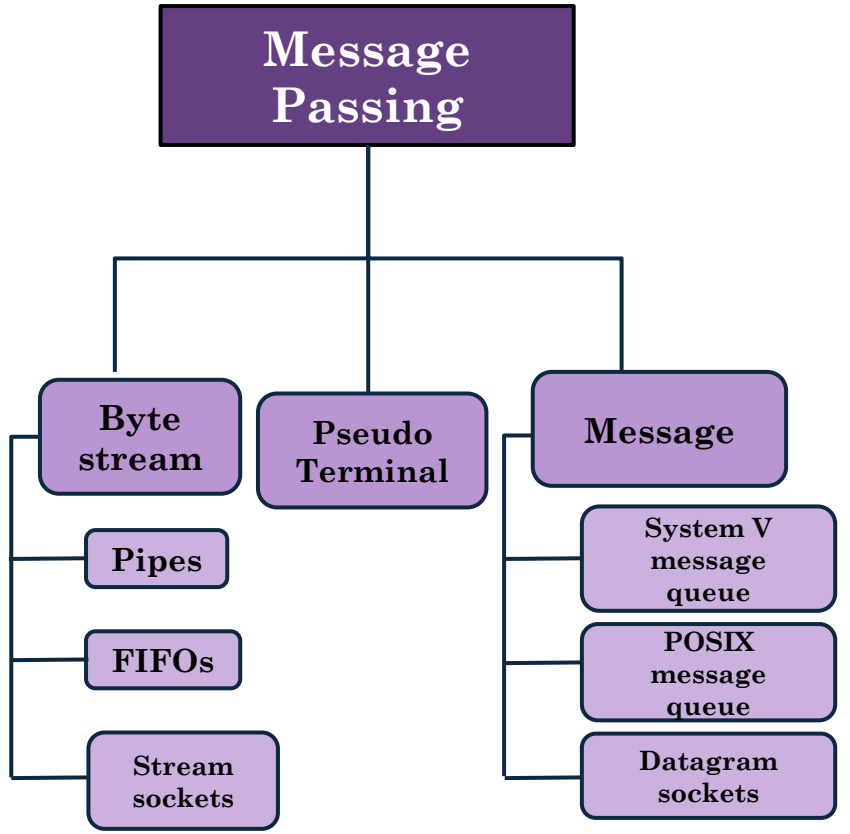
- In Message passing, the processes communicates by sending and receiving **discrete messages.**
- One process writes data to the IPC facility, and another process reads the data and vice versa.
- The kernel often manages the data exchange.
- Message passing facilities require two data transfers between user memory and kernel memory: one transfer from user memory to kernel memory during writing, and another transfer from kernel memory to user memory during reading.
- Message passing facilities have **destructive read semantics** ie; the message will be erased when the reader reads it. So, if another reader attempts to fetch data from a data-transfer facility that currently has no data, then (by default) the read operation will block until some process writes data to the facility.
- The **synchronization between reader and writer is implicit** as the kernel handles the message ordering and delivery.

Communication - Message Passing (cont...)

Byte stream: Data is sent as a **continuous stream of bytes**. The receiving process reads the data in the order it was sent, but there are no defined message boundaries Each read operation may read an arbitrary number of bytes from the IPC facility, regardless of the size of blocks written by the writer.
Example: Pipes , FIFOs and stream sockets.

Message: Data is sent and received in **discrete units** called messages. Each message has a clear boundary, and the operating system often provides a queue to hold messages until they are received. Each read operation reads a whole message, as written by the writer process. It is not possible to read part of a message, leaving the remainder on the IPC facility; nor is it possible to read multiple messages in a single read operation.
Example: System V message queue, POSIX message queue, datagram sockets.

Pseudo Terminals: This is a pair of devices - a master and a slave - that provides a way for a user-space process to control and communicate with another process, as if it were a terminal. It is a more specialized form of stream-based communication, often used for applications like SSH and terminal emulators

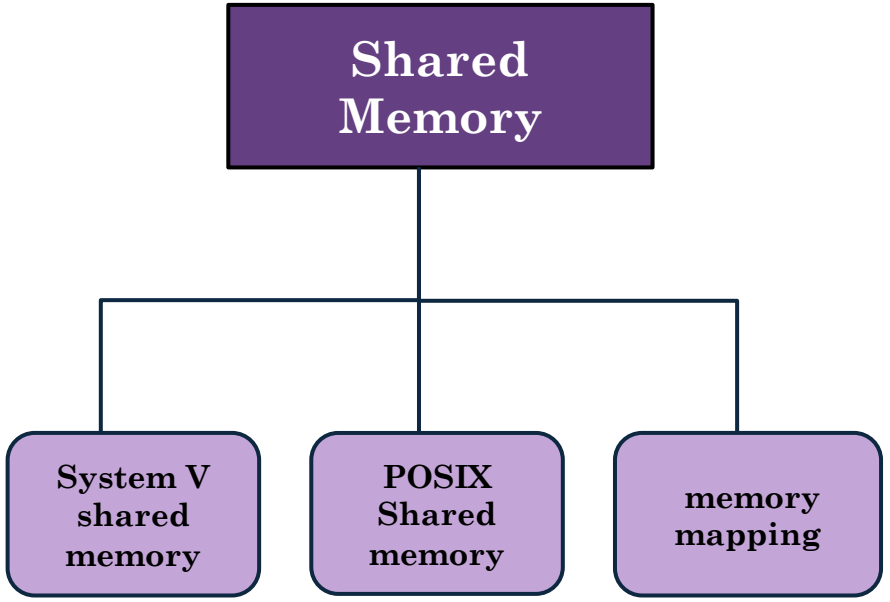


Communication - Shared Memory

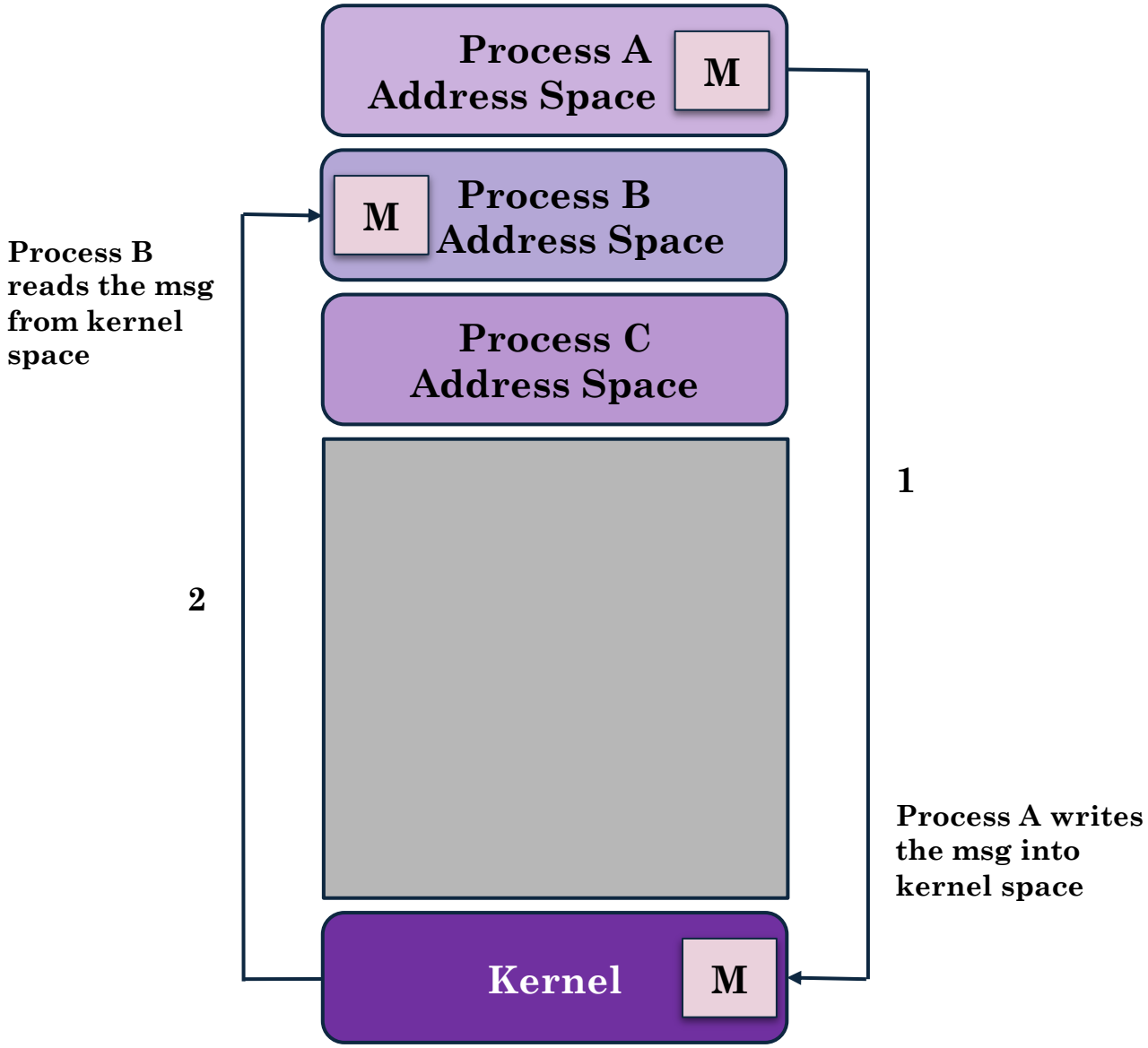
- The processes exchange data by **directly reading and writing to a shared memory region.**
- One process writes data to the shared region and other process reads the data from the shared memory.
- The kernel accomplishes this by making page table entries in each process point to the same pages of RAM.
- Once the memory is mapped, then data transfer **does not require kernel involvement**.
- This method is faster because no data transfer between user and kernel memory is involved.
- In Shared memory, read is **non destructive**, i.e., data placed in shared memory can be read by any number of processes any number of times.
- **Synchronization is not implicit**, and developers must implement explicit synchronization to manage access to shared region

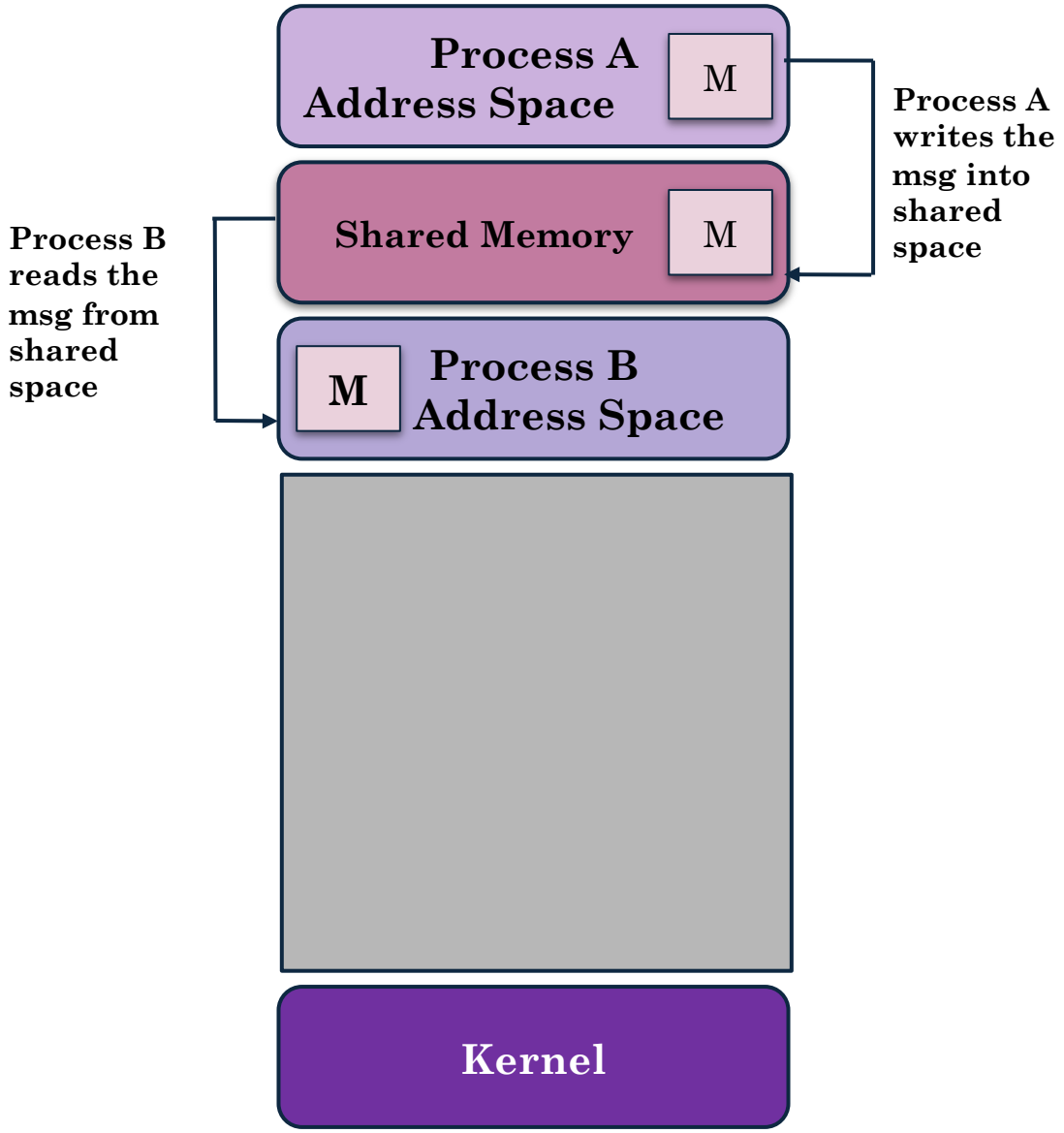
Communication - Shared Memory (cont...)

- Most modern UNIX systems provide **three flavors of shared memory**: System V shared memory, POSIX shared memory, and memory mappings.
- Following are some important points to be kept in mind about shared memory:
  - Although shared memory provides fast communication, this speed advantage is offset by the need to synchronize operations on the shared memory. For example, one process should not attempt to access a data structure in the shared memory while another process is updating it. A semaphore is the usual synchronization method used with shared memory.
  - Data placed in shared memory is visible to all of the processes that share that memory. This contrasts with the destructive read semantics described before for data-transfer facilities.



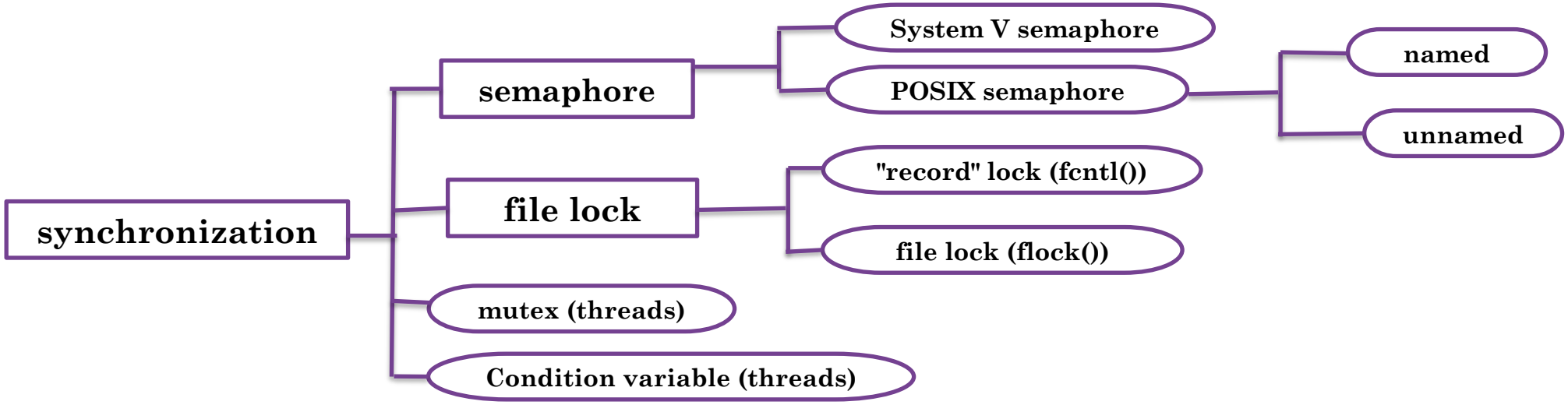
Data Transfer vs Shared Memory





Synchronization Facilities

Synchronization facilities allow processes to coordinate their actions. Allowing processes to avoid doing things such as simultaneously updating a shared memory region or the same part of a file. Without synchronization, such simultaneous updates could cause an application to produce incorrect results.



- **Semaphores** is a simple signaling mechanism used to control access to a shared resource by multiple processes. Linux provides both System V semaphores and POSIX semaphores, which have essentially similar functionality.
- **Mutexes and condition variables** are synchronization facilities normally used with POSIX threads.
- **File locks:** File locks are a synchronization method explicitly designed to coordinate the actions of multiple processes operating on the same file. File locks come in two flavors: read (shared) locks and write (exclusive) locks. Any number of processes can hold a read/shared lock on the same file (or region of a file). However, when one process holds a write/exclusive lock on a file (or file region), other processes are prevented from holding either read or write locks on that file (or file region). Linux provides file-locking facilities via the `flock()` and `fcntl()` system calls.

Persistence of IPC objects

The term persistence refers to the lifetime of an IPC object. It can have three levels:

1. **Process Persistence:** A process-persistent IPC object remains in existence as long as it is held open by at least one process. When the last process closes the object (or terminates), the object is automatically destroyed. Pipes, FIFOs, and sockets are examples of IPC facilities with process persistence.
2. **Kernel Persistence:** A kernel-persistent IPC object exists until either it is explicitly deleted or the system is shut down. System V message queues, semaphores, and shared memory are kernel persistent.
3. **File-system Persistence:** An IPC object with file-system persistence retains its information even when the system is rebooted. POSIX message queues, semaphores, and shared memory can be file-system persistent if the implementation supports it, but this is implementation-dependent and not guaranteed.

Overview of Signals

Introduction to Signals

Suppose a program is running in a `while(1)` loop in the foreground, and the user presses `Ctrl+C` key. The program dies. How does this happens?

- User presses `Ctrl+C` on keyboard.
- The `tty` driver receives the keystroke, recognizes the character combination as predefined 'interupt' character.
- The `tty` driver calls signal system.
- The signal system sends `SIGINT(2)` to currently running process.
- Process receives `SIGINT(2)` signal and performs the default action for `SIGINT` i.e., terminates.
- The process shuts down.
- Actually `Ctrl+C`, ask the kernel to send `SIGINT` to the currently running foreground process.

Note: The key `Ctrl+C` is not hardcoded, you can use `stty` command or the system call `tcsetattr()` to replace the current *intr* control character with another keystroke.

Introduction to Signals (cont...)

Signal is a software interrupt delivered to a process by OS because:

1. **The process did something :** These signal are computer way of saying "Something went wrong inside the program. These signals are typically generated by system:
   - `SIGSEGV`: process tried to access the memory location it was not allowed to
   - `SIGILL`: process tried to execute an instruction that the CPU didn't understand
   - `SIGFPE`: process tried to do a mathematical operation like division by zero

2. **The user did something:** Signals generated by keyboard when user want to control a running program. The user may want to send :
   - `SIGINT`: <CTRL + C> to terminate the program
   - `SIGQUIT`: <CTRL + \> to forcefully terminate a program
   - `SIGTSTP`: <CTRL + Z> to pause the execution of a program

3. **One process wants to tell another process something:** The child process sends `SIGCHLD` to parent, to tell the parent process that it has terminated or stopped

Introduction to Signals (cont...)

- Signals are a limited form of inter-process communication in UNIX. They are usually delivered to a process by OS, typically to **notify it of an asynchronous events**, such as user requests to terminate a process or a process accessing an invalid area of memory.
- Each signal has a **symbolic name** and an **integer value** associated with it (e.g., SIGSEGV or 11) that differentiate it from other signals defined in system header file `/usr/include/asm-generic/signal.h`
- You can see a list of signals on your system using **`kill -l`** command.
- Linux supports 32 real time signals ranging from `SIGRTMIN (32)` to `SIGRTMAX (63)`. Unlike standard signals, real time signals have no predefined meanings, are used for application specific purposes.
- Whenever a process receives a signal, it is interrupted from whatever it is doing, and a kernel function specific to that signal is executed called **signal handler**. When the signal handler function returns, the process continues execution as if this interruption has never occurred.

Synchronous and Asynchronous Signals

Synchronous Signals are generated as a direct result of executing a specific instruction in the program. For example SIGFPE (illegal arithmetic operation) and SIGSEGV (accessing memory location it does not have permission for. Characteristics of synchronous signals are:

- Delivered immediately when the triggering event occurs.
- Directly related to the instruction being executed.
- Predictable and deterministic.

Asynchronous Signals are ggenerated by events external to the program's execution. They can arrive at any time during program execution and are not directly caused by the current instruction. For example, when a user press <Ctrl + C>, when a process sends a signal using kill() system call, when a timer expires (SIGALRM). Characteristics of asynchronous signals are:

- Can be delivered at any point during program execution.
- Generated by external events or other processes.
- Unpredictable timing.

Signal Disposition

Each signal has a current disposition, which determines how the process behaves when it receives that signal. Once generated, the system delivers the signal to the target process. The process can then:

1. **Accept the default signal action:** This is the standard behavior for a signal as defined by OS.
2. **Ignore the signal:** A process can explicitly ignore a signal. The process will take no action and continue its execution, except for SIGKILL and SIGSTOP, which cannot be ignored.
3. **Catch the signal:** Define and install a custom function that will be executed when the process receives the signal called signal handler

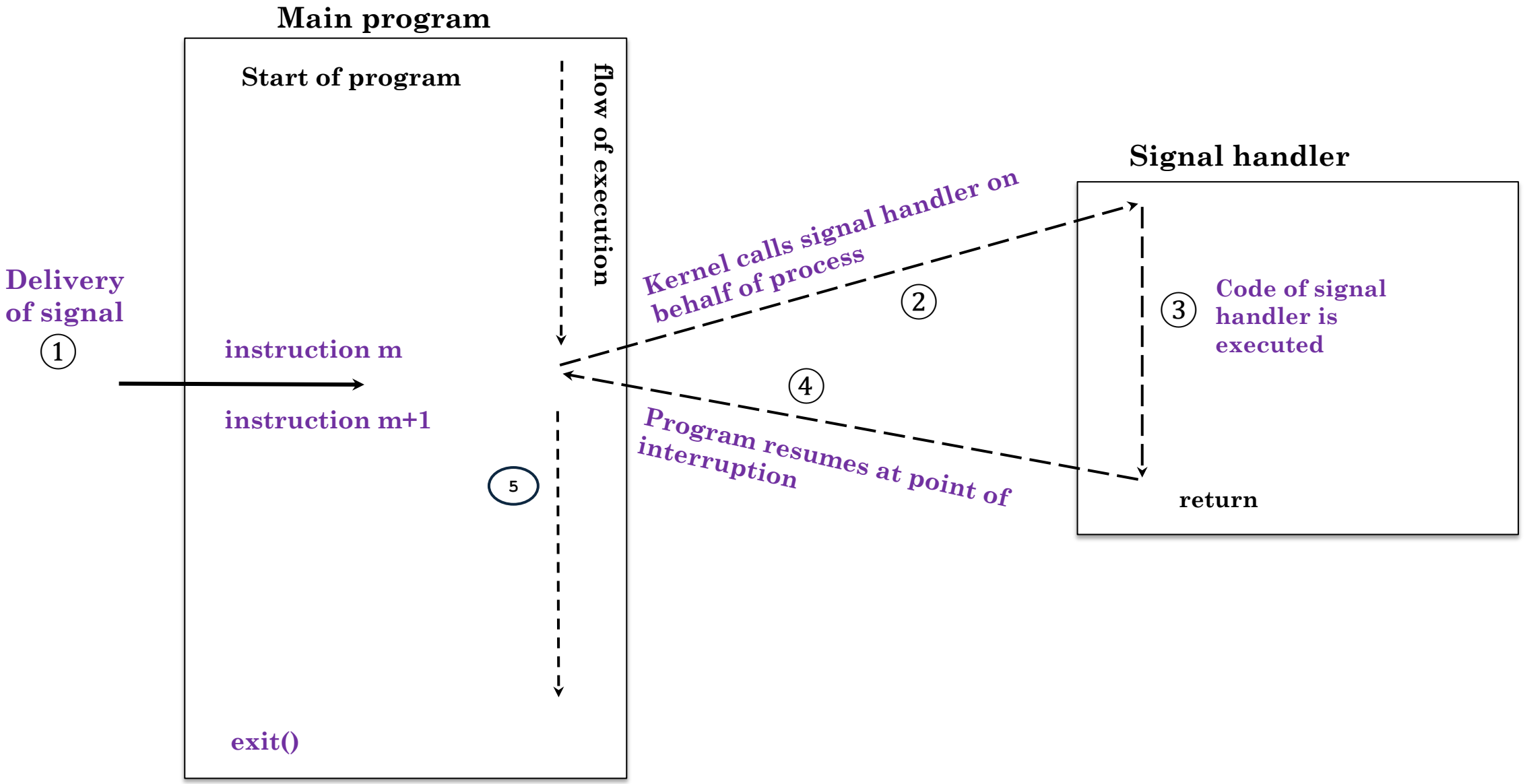
Signal Disposition (cont...)

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**. 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.
5. Execution of the process is **resumed**, which was previously stopped.

A process can change the disposition of a signal using `signal()` or `sigaction()` system calls.

Signal Delivery and Handler Execution



Pending Signals

- A **pending signal** is one that's been generated but not yet delivered to the process. This can happen when the signal is generated while the process is not scheduled or it's blocked. The signal stays in a pending state until delivery conditions are met.
- 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.
- For standard (non–real-time) signals, Linux ensures there's at most one pending signal of each type (duplicate standard signals are not queued).
- You can use `sigpending()` to retrieve the set of signals currently pending delivery to the calling thread.

Signal Handling on Shell

Signal Related Shell Commands

Command	Description
`kill`	Send signals to processes by PID
`killall`	Send signals to all instances of a process by name
`pkill`	Send signals to processes matching criteria (name, user, etc.)
`trap`	Catch and handle signals in shell scripts
`nohup`	Run commands immune to hangup signals
`fg`	Bring background job to foreground (sends SIGCONT if stopped)
`bg`	Put stopped job in background (sends SIGCONT)
`jobs`	List active jobs and their signal status

```
pkill -u user123 chrome       # Kill chrome processes for user123
pkill -f "python3 my_script" # Match full command line
pkill -SIGKILL sshd          # Force kill all sshd processes
```

Signals on Shell: `kill`, `killall`, `pkill`

- The `kill` command is used to send a signal to a process by its ID:

  `$ kill <-SIGXXX> <pid>`

- To view the available signal numbers and their corresponding constants use following command:

  `$ kill -l`

- The default signal for `kill` is `SIGTERM (15)` which is polite request to terminate the process, which will allow the process to free resources (memory, files, sockets etc) and execute the exit handlers registered using `atexit()` or signal handlers registered using `signal()` or `sigaction()`.

  `$ kill <pid>`

- To forcefully terminate a process, send `SIGKILL(9)`, which cannot be caught or ignored:

  `$ kill -9 <pid>`

- You can use **`killall`** command to send a signal to all processes using a specific name:

  `$ killall chrome` # Kill all instances of chrome processes

  `$ killall vlc` # Kill all instances of vlc processes

- You can use **`pkill`** command to send a signal to processes matching a pattern(name, full command, user etc)

  `$ pkill -f "python3 my_script"` # Match full command line

  `$ pkill -u kakamanna chrome` # Kill chrome processes for user kakamanna

Signals on Shell: `trap`

- The `trap` command in Bash is used to specify commands to execute automatically when the shell receives a specific signal or event, such as `SIGINT`, `SIGTERM`, or shell exit.

```
$ trap 'commands_to_execute' <signal_name_or_number>
```

- The define a custom handler for `SIGINT(2)` use the following command:

```
$ trap 'echo "Received SIGINT. Cleaning up..."' SIGINT
```

- To list all currently defined traps use following command:

```
$ trap -p
```

- To ignore a signal use the following command:

```
$ trap '' SIGINT
```

- To reset the signal handler to default use the following command:

```
$ trap - SIGINT
```

- Most signals may be caught by the process but there are few signals that the process cannot be caught or ignore and cause the process to terminate:
  - **SIGKILL(9):** This signal immediately and unconditionally terminates a process. A process cannot catch, block or ignore it.
  - **SIGSTOP(19):** This signal immediately suspends a process's execution.
- System **`shutdown`** process first sends `SIGTERM(15)` to all processes, waits a while and after allowing them a grace period to shut down cleanly, it kills which are still running processes using `SIGKILL(9)`

Important Signals

Default behavior : Term

SIGHUP	1	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 (<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	The sure kill signal, can't be blocked, ignored, or caught by a handler, and 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()`
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 (cont...)

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. This signal will terminate the process and generate a core dump file, which developer can load inside gdb debugger to perform postmortem of the process.
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	8	Generate by floating point Arithmetic Exception
SIGSEGV	11	Generated when a program makes an invalid memory reference. In C, this signal is delivered to a process when it tries to dereference 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)	19	This is the sure stop signal. It can't be blocked, ignored, or caught by a handler; thus, it always stops the execution of a process.
SIGTSTP(20)	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 : Continue

SIGCHLD	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.

Core dumps

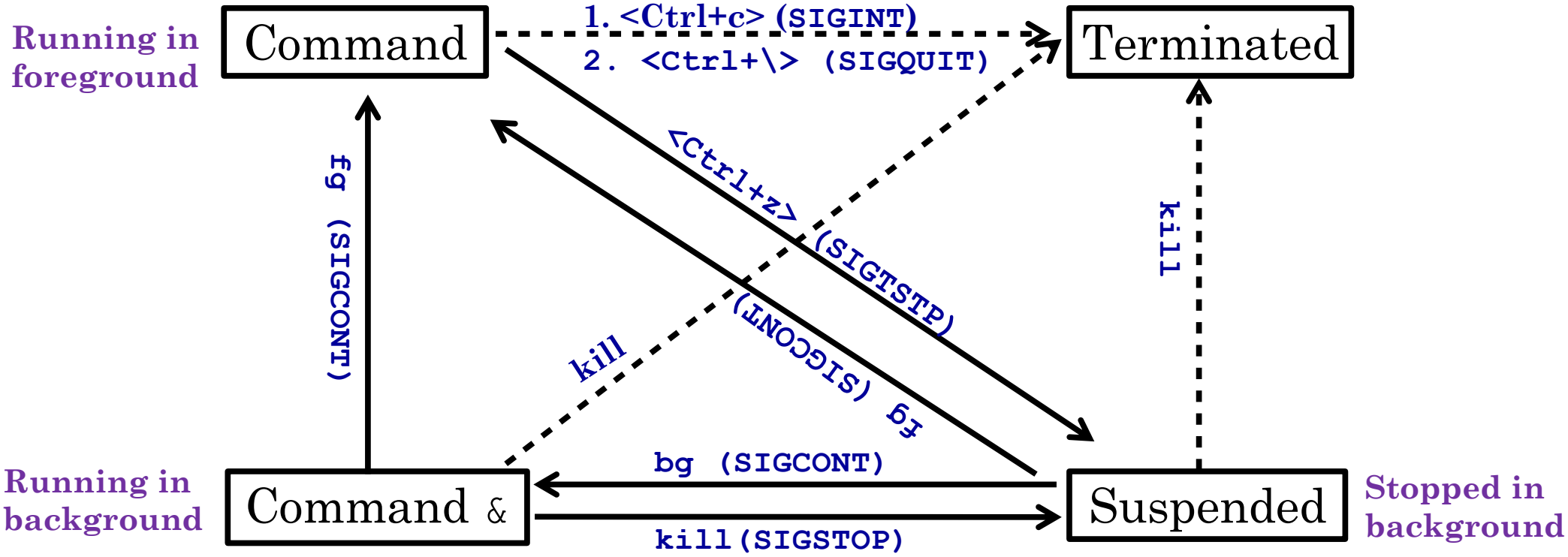
- A **core dump** file, typically named **core**, is an image of the process virtual memory at the time it was terminated. It contains a snapshot of process variables, stack and registers details at the time of failure. From a core file, the programmer can investigate the reason for termination using a debugger. By default core file creation is disabled to save the disk space.
- The **`ulimit`** command is used to view/set resource limits for the current shell and the processes started by it.
  - **Soft limit**: The current limit enforced by the kernel. A process can increase its soft limit up to the hard limit.
  - **Hard limit**: The maximum allowed limit. Only the superuser (root) can increase this.
- To display the current limits: `$ulimit -a`
- To display the core file size: `$ulimit -c` (returns zero by default).
- To enable core file creation: `$ulimit -c unlimited`
- Once a core dump is created, you can use a debugger to analyze it. The general steps are:

```
$ gcc -g -Wall divbyzero.c   (Compile your code with the -g flag to include debugging symbols)
$ ./a.out                    (Run the program, which will cause a crash and generate a core file)
$ gdb ./a.out core           (Load the core file in gdb, which will pinpoint the root of program crash)
```

Foreground vs Background Processes

- In a CLI, processes can exist in one of two states: foreground or background:
  - **Foreground Process:** This is the single process that is actively holding the terminal. It receives all keyboard input and its output is displayed directly on the screen. A program like `vim` or `less` is a classic example of a foreground process. You must wait for a foreground process to finish or be stopped before you can execute another command.
  - **Background Process:** These are processes that run without a direct connection to the terminal. They don't typically require user input and their output is not shown on the screen, allowing the user to continue using the terminal for other commands. Common examples include an audio player or a file search utility like `find`.
- Unlike a Graphical User Interface (GUI) where you can simply click to minimize or switch applications, a CLI requires specific commands and signals to move a process between these two states. For example, pressing Ctrl+Z sends a SIGTSTP signal to a foreground process, moving it to a stopped state in the background. From there, you can use the bg command to resume it in the background or fg to bring it back to the foreground

Job Control States



Managing Background & Foreground Tasks

- The **`jobs`** command is used to track and manage background or suspended tasks within the current shell session. It works alongside **`fg`**, **`bg`**, and **`kill`** to control process states interactively.
- This is especially useful in scripting, multitasking in terminal sessions, or when dealing with long-running foreground commands.
- Every command run in the shell can be:
  - **Foreground**: Takes control of the terminal.
  - **Background**: Runs while the terminal remains usable.
- **Background execution:** Use & at the end of a command → **`sleep 60 &`**
- To view background/suspended jobs → **`jobs`**

Commands	Description
`jobs`	Lists jobs with status (Running / Stopped)
`fg %n`	Brings job **n** to the foreground
`bg %n`	Resumes stopped job **n** in the background
`kill %n`	Sends terminate signal to job **n**

```
$ sleep 100 &
[1] 15018
$  jobs
[1]  + running    sleep 100
$ fg %1
[1]  + 15018 running    sleep 100
^Z
[1]  + 15018 suspended  sleep 100
$ bg %1
[1]  + 15018 continued  sleep 100
$  kill %1
[1]  + 15018 terminated  sleep 100
```


Signal Handling in C

kill() System Call

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

- One process can send a signal to another process using the **kill()** system call.
- The **pid** argument identifies one or more processes to which the signal specified by **sig** argument is to be sent
- Four different cases determine how pid is interpreted:
  - If **pid > 0**, the signal is sent to the process with the process ID specified by pid.
  - If **pid == 0**, the signal is sent to every process in the same process group as the calling process, including the calling process itself.
  - If **pid < –1**, the signal is sent to all of the processes in the process group whose ID equals the absolute value of pid
  - If **pid == –1**, the signal is sent to every process for which the calling process has permission to send a signal, except `init` and the calling process. If a privileged process makes this call, then all processes on the system will be signaled, except for these last two.
- If **sig** argument is zero then normal error checking is performed but no signal is sent. Used to determine if a specified process still exists. If it doesn't exist, a -1 is returned & `errno` is set to `ESRCH`
- If no process matches the specified **pid**, `kill()` fails and sets `errno` to `ESRCH`

raise() Function Call

```
#include <signal.h>
int raise(int sig);
```

- In UNIX-based operating systems, processes have the ability to send signals to themselves (Self signaling).
- The `raise()` is a library function that allows a process to send signal to itself.
- In a single-threaded program, a call to `raise()` is essentially a wrapper for following call to `kill()`:

```
kill(getpid(), sig);
```

- When a process sends itself a signal using `raise()` or `kill()`, the signal is delivered immediately even before the `raise()` returns to the caller.
- The only error that can occur with `raise()` is `EINVAL`, if the provided signal number is invalid.

abort() Function

```
#include <stdlib.h>

void abort();
```

- The `abort()` function terminates the calling process by raising a `SIGABRT(6)` signal and causes it to produce a core dump file.
- The default action for `SIGABRT` is to produce a core dump file before terminating the process. The core dump file can then be used within a debugger to examine the state of the program at the time of the `abort()` call
- `abort()` function never returns.

pause() System Call

```
#include <stdlib.h>

void pause();
```

- The purpose of **`pause()`** is to suspend the process execution until a signal is received and handled.
- The **`pause()`** system call causes the invoking process / thread to sleep until a signal is received that either terminates it or causes it to call a signal catching function.
- When a process calls `pause()`, it enters a sleeping state and stops consuming CPU resources. It will only wake up if a signal is delivered to it.
- The **`pause()`** system call is an efficient way for a process to wait for an event, where the event is defined as the receipt of a signal.
- If the received signal default action is to terminate the process (e.g SIGTERM), the `pause()` call will not return, as the process is terminated directly by the kernel.

alarm() System Call

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

- The **alarm()** system call is used to ask the system to send a special signal `SIGALRM(14)` to the process after a specified number of seconds.
- The `SIGALRM` signal is delivered asynchronously. If the process does not have a custom handler for `SIGALRM`, the default action is to terminate the process.
- This function returns the previously registered alarm clock for the process that has not yet expired, i.e., the number of seconds left for that alarm clock is returned as the value of this function.
- Previously registered alarm clock is cancelled and it will start a new alarm.
- If seconds = 0, no new alarm is scheduled and any pending alarm request is canceled.
- Timers are normally used to allow one to check timeout:
  - Wait for user input up to 30 seconds, else exits.
  - If a server has not responded in last 30 seconds, notify the user and exits.
  - Set an alarm before making a network call that could block indefinitely.

Adding a Delay: sleep

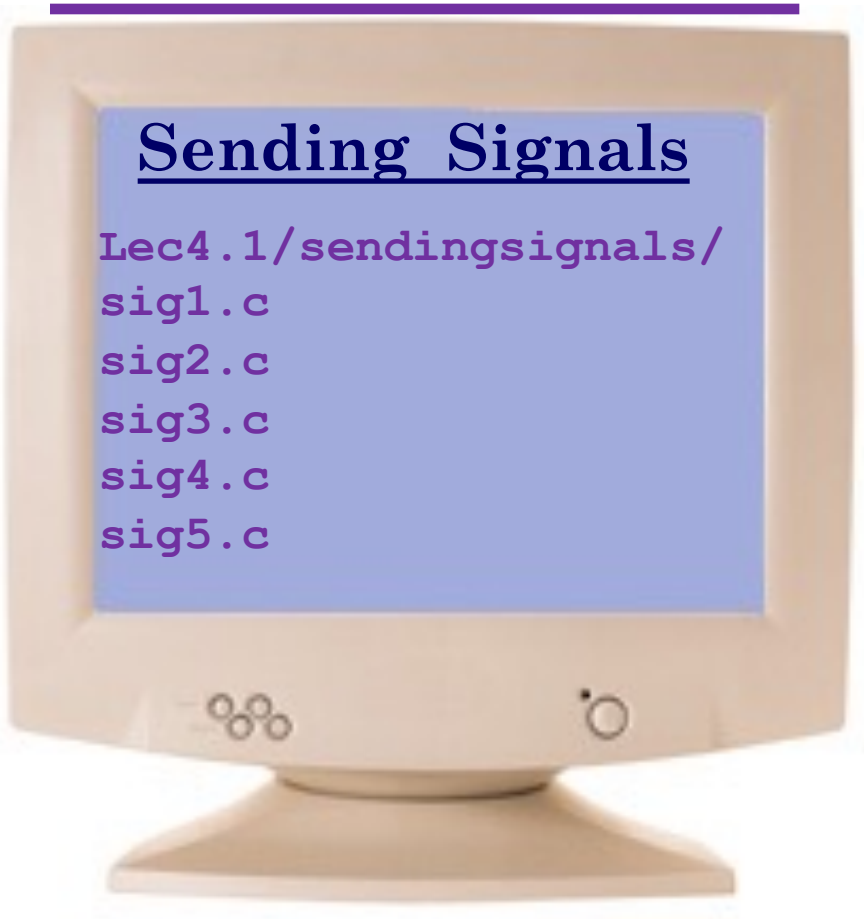
```
int sleep(unsigned int seconds);
int usleep(useconds_t usec);
int nanosleep(const struct timespec* req,struct timespec* rem);
```

- The `sleep()` and `usleep()` functions are library functions provided by the C standard library (`glibc`), and they are internally implemented using the Linux `nanosleep()` system call.
- All these calls suspends the execution of the calling process until either the specified time interval has elapsed, or the process receives a signal that interrupts the sleep.
- The `sleep()` call has a precision of seconds, `usleep()` call has a precision of micro-seconds, while the `nanosleep()` system call has a precision of nano-seconds.
- The `nanosleep()` is the most precise and modern way to sleep. It pauses the process for a duration specified in a `timespec` structure, which can define both seconds and nanoseconds:

```
struct timespec {
        time_t tv_sec;        /* No of seconds */
        long   tv_nsec;       /* No of nanoseconds */
};
```

- If the `nanosleep()` system call is interrupted by a signal, it returns a value and stores the remaining unslept time in the `rem` parameter.

Demonstration



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

Writing signal handlers

signal() System call

```
#include <signal.h>
sighandler_t signal(int signum, sighandler_t handler)
```

- The `signal()` system call installs a new signal handler for the signal with number **`signum`.**
- This allows the process to change the behavior of that signal from default action to a custom one.
- The second argument is the address of custom function to be registered. `sighandler_t` is the type of the signal handler function, which can have three values:

  i) `SIG_IGN`: the signal is ignored.

  ii) `SIG_DFL`: the default action associated with signal occur (revert the default action for that signal).

  iii) A user specified function address.
- It returns the previous handler for the specified signal, or SIG_ERR on error.
- Ignoring `SIGFPE`, `SIGILL` or `SIGSEGV` signal that was not generated by `kill()` or `raise()` functions results in undefined behavior.

Handling Signals with `signal()`

- The following code snippet, temporarily installs a custom signal handler for the SIGINT signal. It first saves the current SIGINT handler, then sets `newhandler` as the new handler using the `signal()` function.
- After executing the desired part of the program where the custom behavior is needed, it restores the original signal handler to return the program to its previous state.
- This pattern is useful when a program needs to override signal handling only during a specific portion of its execution.

```
void newhandler(int sig) {
    // ---Your code to handle SIGINT (Ctrl+C) come here---
}
int main() {
    void (*oldhandler)(int);
    oldhandler = signal(SIGINT, newhandler);
    // ---Your program logic comes here, where SIGINT is handled by newhandler ---

    signal(SIGINT, oldhandler); // Restore the original SIGINT handler
    return 0;
}
```

Demonstration

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

Masking Signals to Avoid Signal Races

Blocking a Signal using Signal Mask

- Sometimes, we need to ensure that a segment of code is not interrupted by the delivery of a signal(s). This is mostly done to protect critical sections of code from being interrupted, which helps avoid race conditions and inconsistent program states. So we need to block signal(s) from being delivered to the process.
- When a blocked signal is generated, it is not delivered to the process and remains pending until that signal is unblocked.
- Blocking a signal is different from ignoring a signal, where the signal is delivered and the process handles it by throwing it away.
- Every process has a **signal mask** represented by **`sigset_t`** data type, which acts like a bitmask, where each bit corresponds to a specific signal. If a bit is set to one, the corresponding signal is blocked. Initially, the signal mask for a new process contains all zeros, meaning no signals are blocked.

Functions related to Signal Sets

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);

int sigaddset(sigset_t *set, int sig);
int sigdelset(sigset_t *set, int sig);
```

In order to set the signal mask of a process, we can create a new `sigset_t` variable and set its bits according to our need. There are two ways you can do this:

- **Option 1:**
  - Pass the address of `set` variable to the `sigemptyset()` function, which will initializes the `set` to be empty, with all signals excluded from the `set`.
  - Then use the `sigaddset()` function, to add the specified signal to the `set`.
- **Option 2:**
  - Pass the address of `set` variable to the `sigfillset()` function, which will initializes the `set` to be full, with all signals included in the `set`.
  - Then use the `sigdelset()` function, to delete the specified signal from the `set`.

Masking signal using `sigprocmask()`

```
int sigprocmask (int how,const sigset_t *set1, sigset_t *set2)
```

- The **`sigprocmask()`** system call is used to fetch and/or change the signal mask of the calling thread. The signal mask is the set of signals whose delivery is currently blocked for the caller.
- The **`set1`** argument is the new signal mask that we want to set. If it is NULL, then the signal mask is unchanged (`how` is ignored) but the current value of the signal mask is returned in set2.
- If argument **`set2`** is non-null, the previous value of signal mask is stored in it. This is useful when we want to restore the previous masking state once we're done with our critical section.
- The **how** argument determines how the signal mask is changed and must be one of the following:

`SIG_BLOCK`	1	The set of blocked signals is the union of `set1` and the current signal set.
`SIG_UNBLOCK`	2	The signals in the `set1` are removed from the current set of blocked signals.
`SIG_SETMASK`	3	The set of blocked signals is set to the argument `set1`.

Demonstration

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

To Do

- Watch SP video overview of IPC
  https://youtu.be/EX7EWSX8-qM?si=nQ5KEAqd7ndPU0EH

- Watch SP video on Signals
  https://youtu.be/YBg9sWw4qbU?si=g2GximUuHnTlui6B

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