# Linux Internals

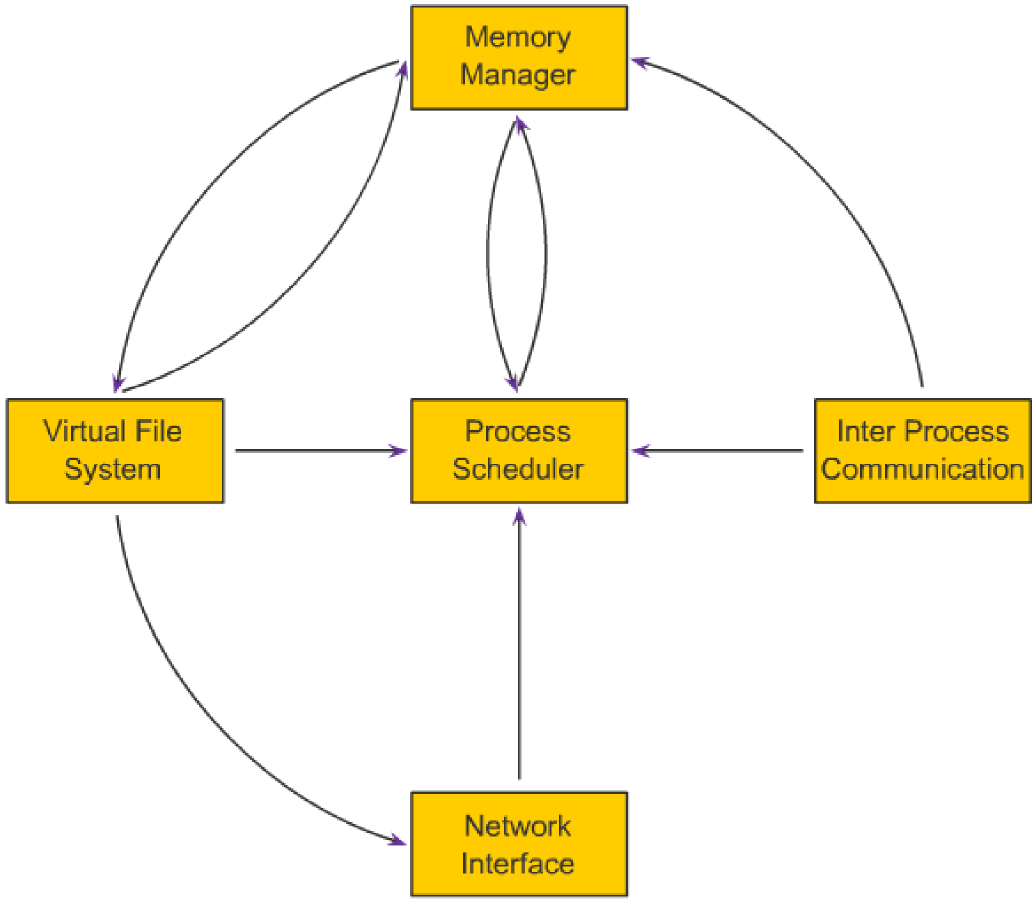
## Day 2

Team Emertxe

# Linux Kernel Subsystem

# Introduction

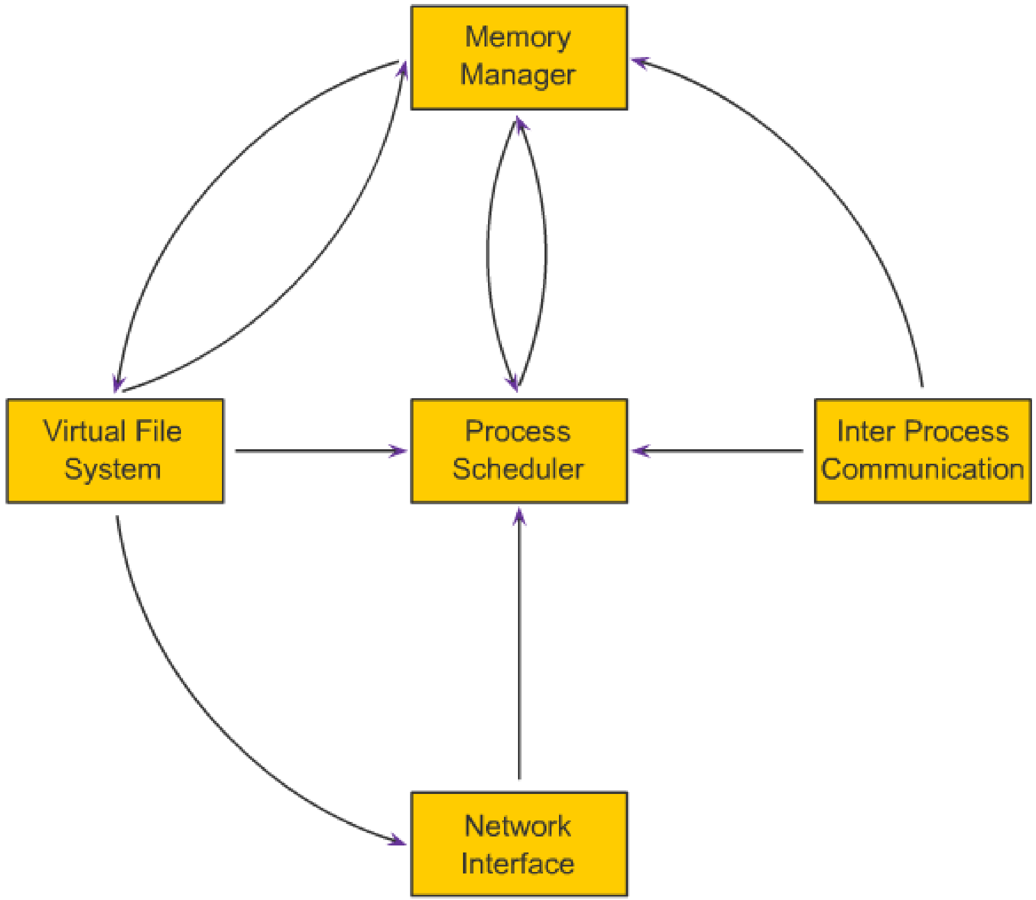
## Linux Kernel Subsystem



- Process Scheduler (SCHED):
  - To provide control, fair access of CPU to process, while interacting with HW on time
- Memory Manager (MM):
  - To access system memory securely and efficiently by multiple processes. Supports Virtual Memory in case of huge memory requirement
- Virtual File System (VFS):
  - Abstracts the details of the variety of hardware devices by presenting a common file interface to all devices

# Introduction

## Linux Kernel Subsystem



- Network Interface (NET):
  - provides access to several networking standards and a variety of network hardware
- Inter Process Communications (IPC):
  - supports several mechanisms for process-to-process communication on a single Linux system

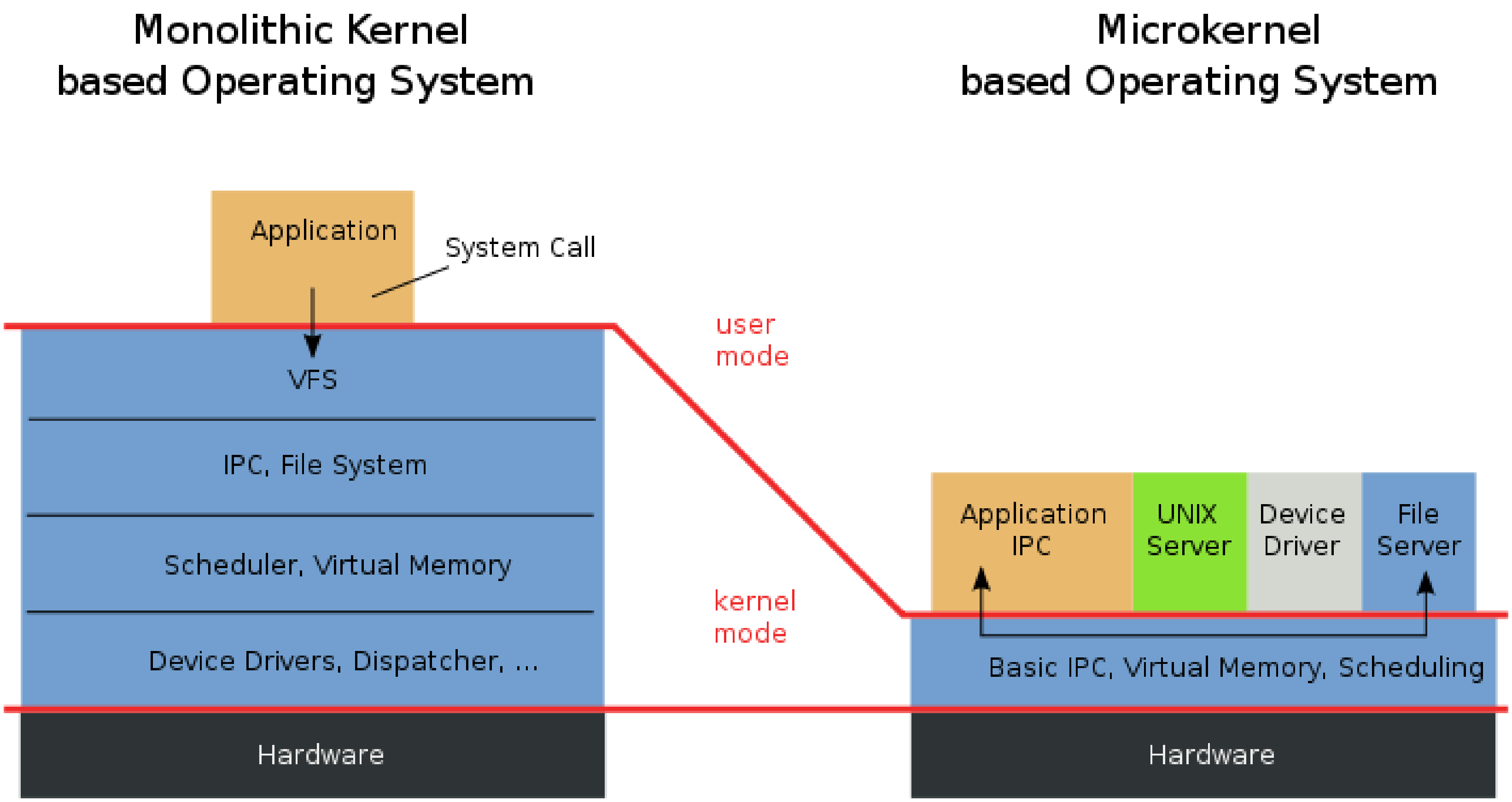
# Introduction

## Linux Kernel Architecture

- Most older operating systems are monolithic, that is, the whole operating system is a single executable file that runs in 'kernel mode'
- This binary contains the process management, memory management, file system and the rest (Ex: UNIX)
- The alternative is a microkernel-based system, in which most of the OS runs as separate processes, mostly outside the kernel
- They communicate by message passing. The kernel's job is to handle the message passing, interrupt handling, low-level process management, and possibly the I/O (Ex: Mach)

# Introduction

## Linux Kernel Architecture



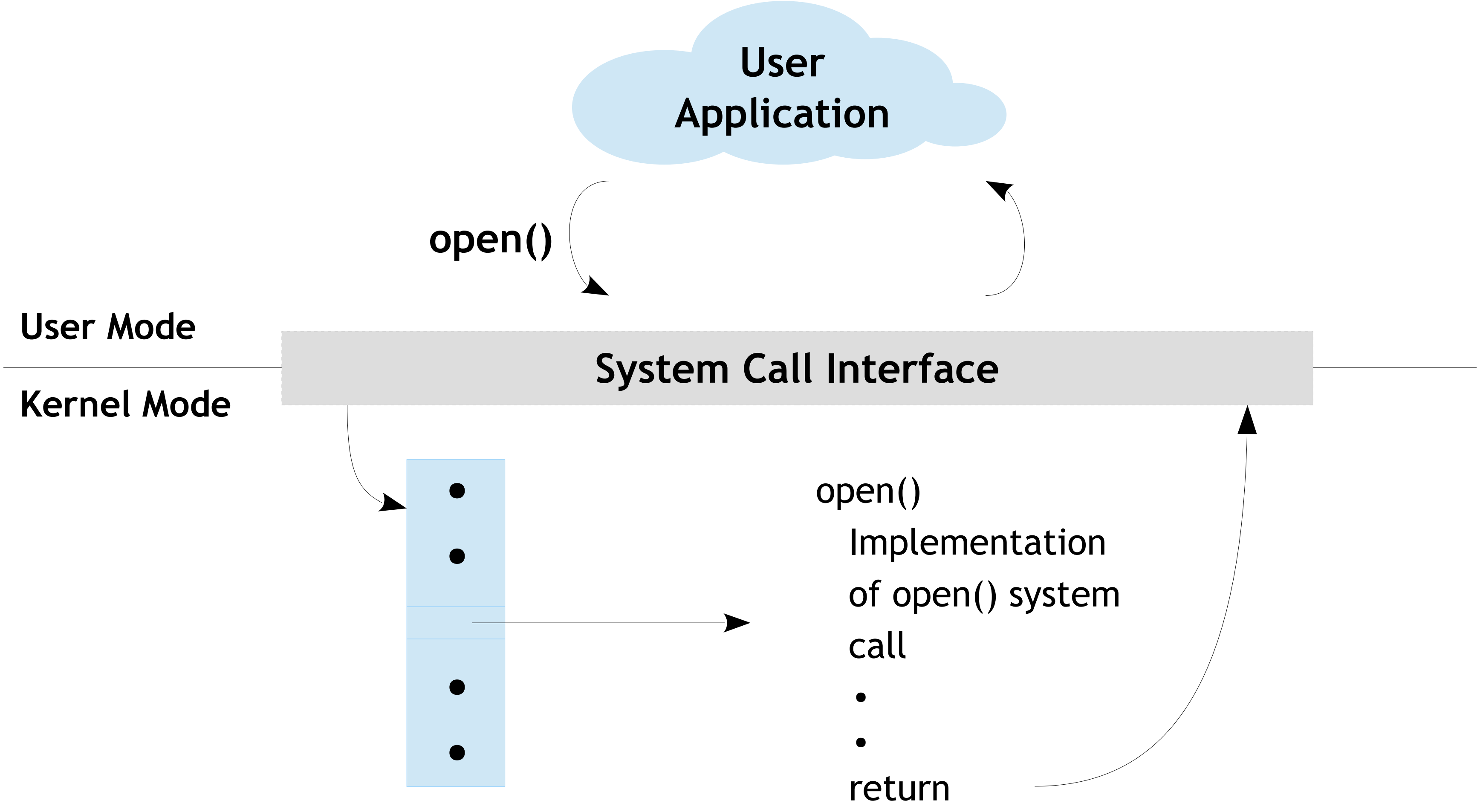
# System Calls

# System calls

- A set of interfaces to interact with hardware devices such as the CPU, disks, and printers.
- Advantages:
  - Freeing users from studying low-level programming
  - It greatly increases system security
  - These interfaces make programs more portable

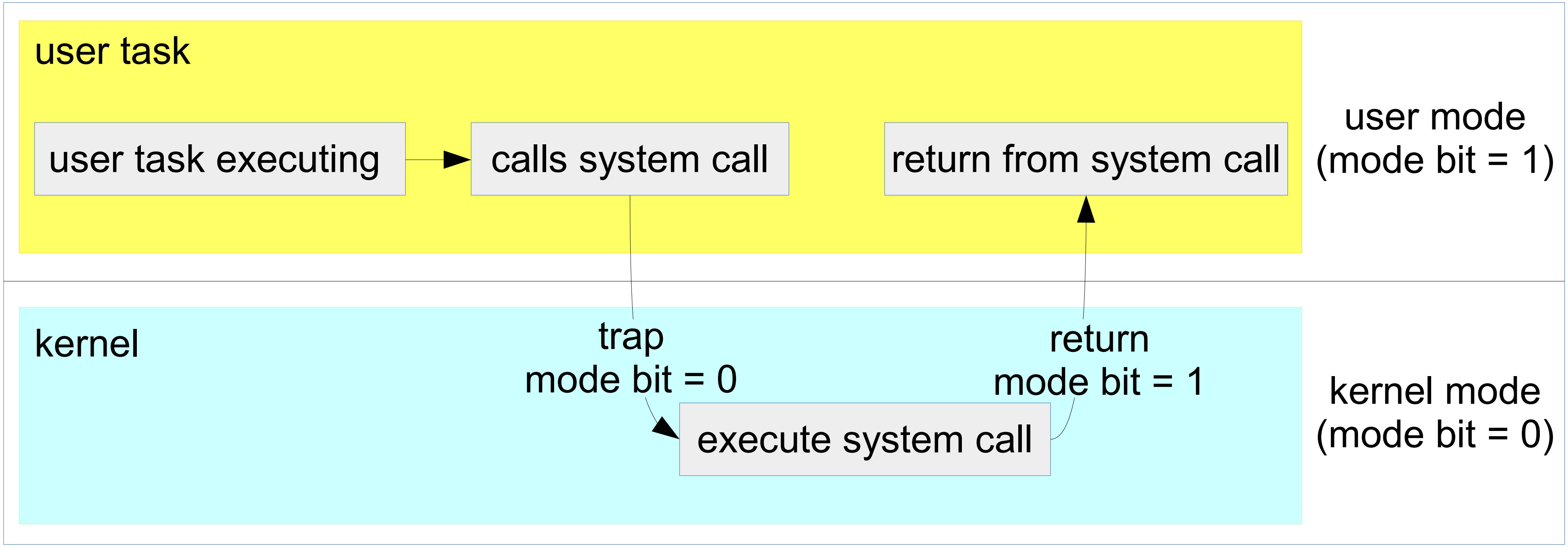
For a OS programmer, calling a system call is no different from a normal function call. But the way system call is executed is way different.

# System calls



# System Call

## Calling Sequence



Logically the system call and regular interrupt follow the same flow of steps. The source (I/O device v/s user program) is very different for both of them. Since system call is generated by user program they are called as ‘Soft interrupts’ or ‘Traps’

# System Call
## vs Library Function

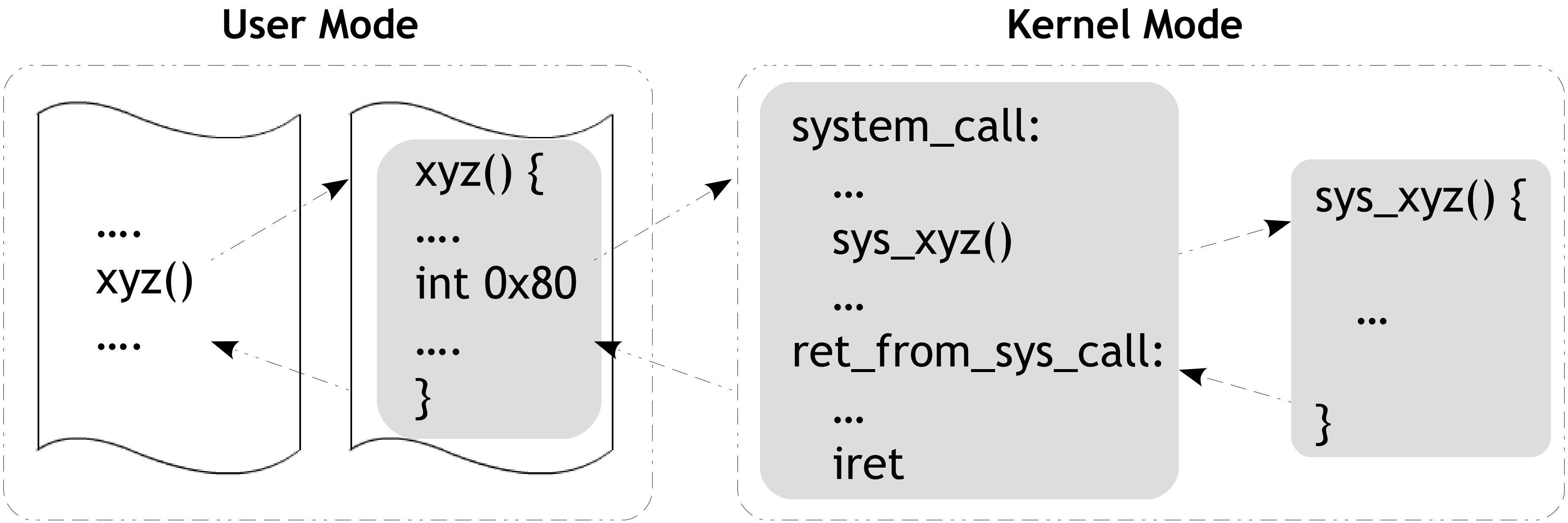
- A library function is an ordinary function that resides in a library external to your program. A call to a library function is just like any other function call
- A system call is implemented in the Linux kernel and a special procedure is required in to transfer the control to the kernel
- Usually, each system call has a corresponding wrapper routine, which defines the API that application programs should employ

✓ Understand the differences between:
  - Functions
  - Library functions
  - System calls

✓ From the programming perspective they all are nothing but simple C functions

# System Call

## Implementation



```
....
xyz()
....
```

```
xyz() {
....
int 0x80
....
}
```

```
system_call:
   …
   sys_xyz()
   …
ret_from_sys_call:
   …
   iret
```

```
sys_xyz() {

   …

}
```

System Call Invocation in application program

Wrapper routine in libc standard library

System call handler

System call service routine

# System Call

## Information: strace

- The strace command traces the execution of another program, listing any system calls the program makes and any signals it receives

  *E.g.: strace hostname*

- Each line corresponds to a single system call.
- For each call, the system call's name is listed, followed by its arguments and its return value

# System Call

## Example: fcntl

- The fcntl system call is the access point for several advanced operations on file descriptors.
- Arguments:
  - An open file descriptor
  - Value that indicates which operation is to be performed

# System Call

## Example: gettimeofday()

- Gets the system’s wall-clock time.
- It takes a pointer to a struct timeval variable. This structure represents a time, in seconds, split into two fields.
  - tv_sec field - integral number of seconds
  - tv_usec field - additional number of usecs

# System Call

## Example: nanosleep()

- A high-precision version of the standard UNIX sleep call
- Instead of sleeping an integral number of seconds, ***nanosleep*** takes as its argument a pointer to a ***struct timespec*** object, which can express time to nanosecond precision.
  - tv_sec field - integral number of seconds
  - tv_nsec field - additional number of nsecs

# System Call

## Example: Others

- open
- read
- write
- exit
- close
- wait
- waitpid
- getpid
- sync
- nice
- kill etc..

# Process

# Process

- Running instance of a program is called a PROCESS
- If you have two terminal windows showing on your screen, then you are probably running the same terminal program twice-you have two terminal processes
- Each terminal window is probably running a shell; each running shell is another process
- When you invoke a command from a shell, the corresponding program is executed in a new process
- The shell process resumes when that process complete

# Process

## vs Program

- A program is a passive entity, such as file containing a list of instructions stored on a disk
- Process is a active entity, with a program counter specifying the next instruction to execute and a set of associated resources.
- A program becomes a  process when an executable file is loaded into main memory

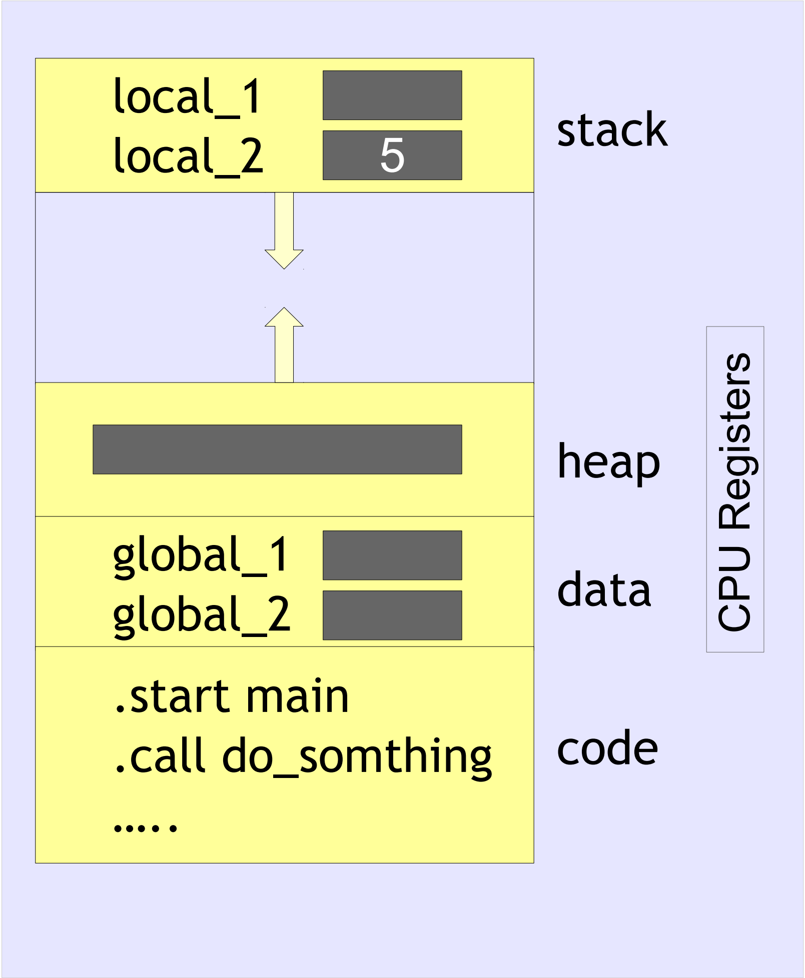
| Factor | Process | Program |
|---|---|---|
| Storage | Dynamic Memory | Secondary Memory |
| State | Active | Passive |

# Process
## vs Program

**Program**

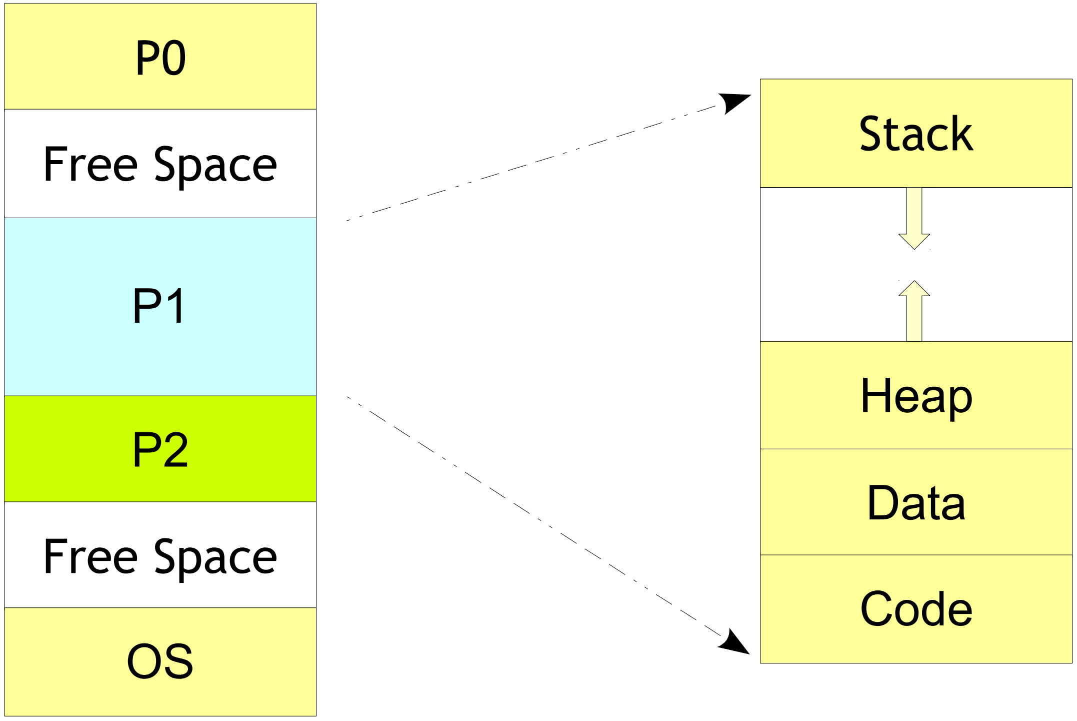
```
int global_1 = 0;
int global_2 = 0;

void do_somthing()
{
        int local_2 = 5;
        local_2 = local_2 + 1;
}

int main()
{
        char *local_1 = malloc(100);

        do_somthing();
        …..
}
```

**Task**



# Process

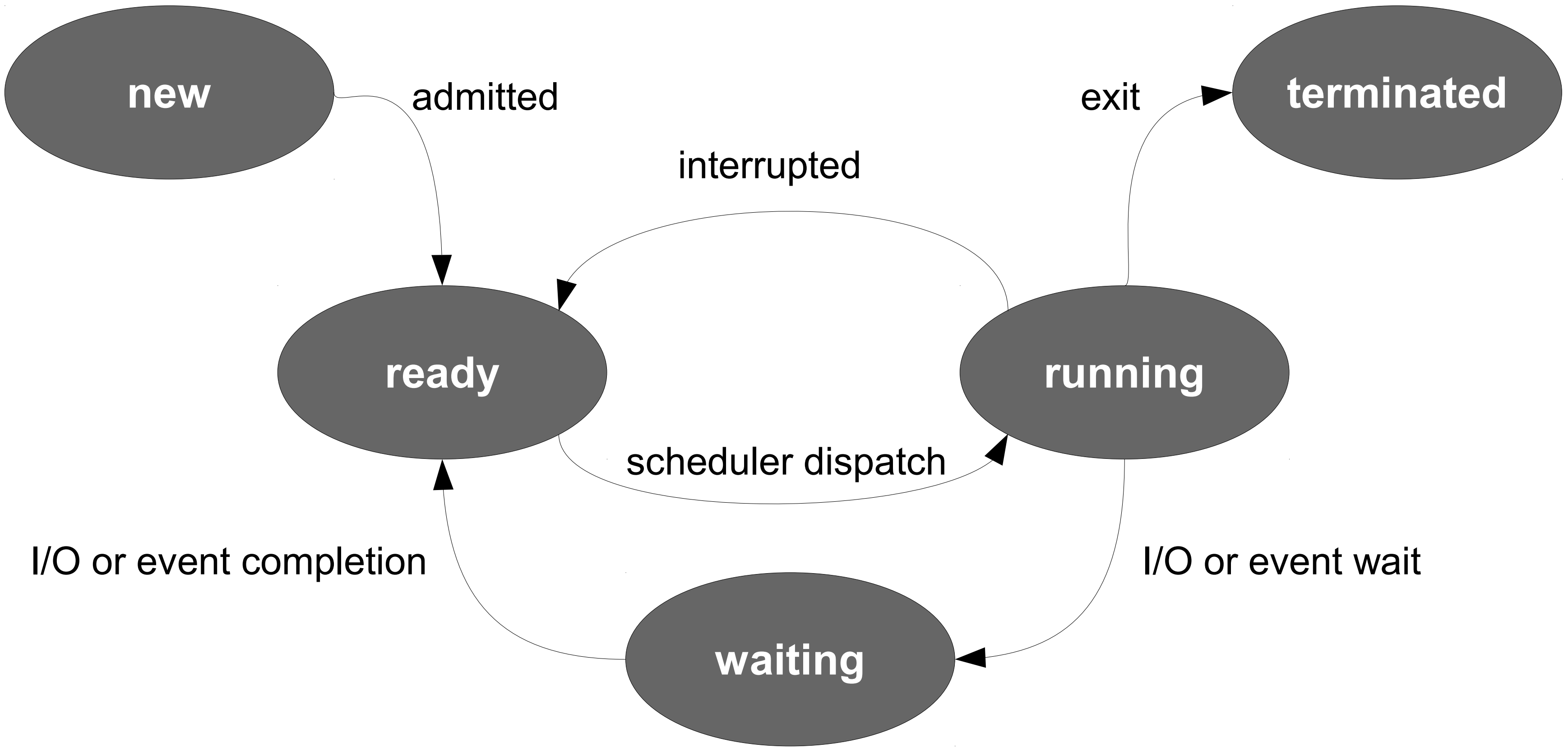
## More processes in memory!



Each Process will have its own Code, Data, Heap and Stack

# Process

## State Transition Diagram



# Process

## States

- A process goes through multiple states ever since it is created by the OS

| State | Description |
|---|---|
| New | The process is being created |
| Running | Instructions are being executed |
| Waiting | The process is waiting for some event to occur |
| Ready | The process is waiting to be assigned to processor |
| Terminated | The process has finished execution |

# Process
## Descriptor

- To manage tasks:
  - OS kernel must have a clear picture of what each task is doing.
  - Task's priority
  - Whether it is running on the CPU or blocked on some event
  - What address space has been assigned to it
  - Which files it is allowed to address, and so on.
- Usually the OS maintains a structure whose fields contain all the information related to a single task

# Process
## Descriptor

| Pointer | Process State |
|---|---|
| Process ID | |
| Program Counter | |
| Registers | |
| Memory Limits | |
| List of Open Files | |
| •<br>•<br>•<br>•<br>•<br>• | |

- Information associated with each process.
- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- I/O status information

# Process

## Descriptor – State Field

- State field of the process descriptor describes the state of process.
- The possible states are:

| State | Description |
|---|---|
| TASK_RUNNING | Task running or runnable |
| TASK_INTERRUPTIBLE | process can be interrupted while sleeping |
| TASK_UNINTERRUPTIBLE | process can't be interrupted while sleeping |
| TASK_STOPPED | process execution stopped |
| TASK_ZOMBIE | parent is not issuing wait() |

# Process

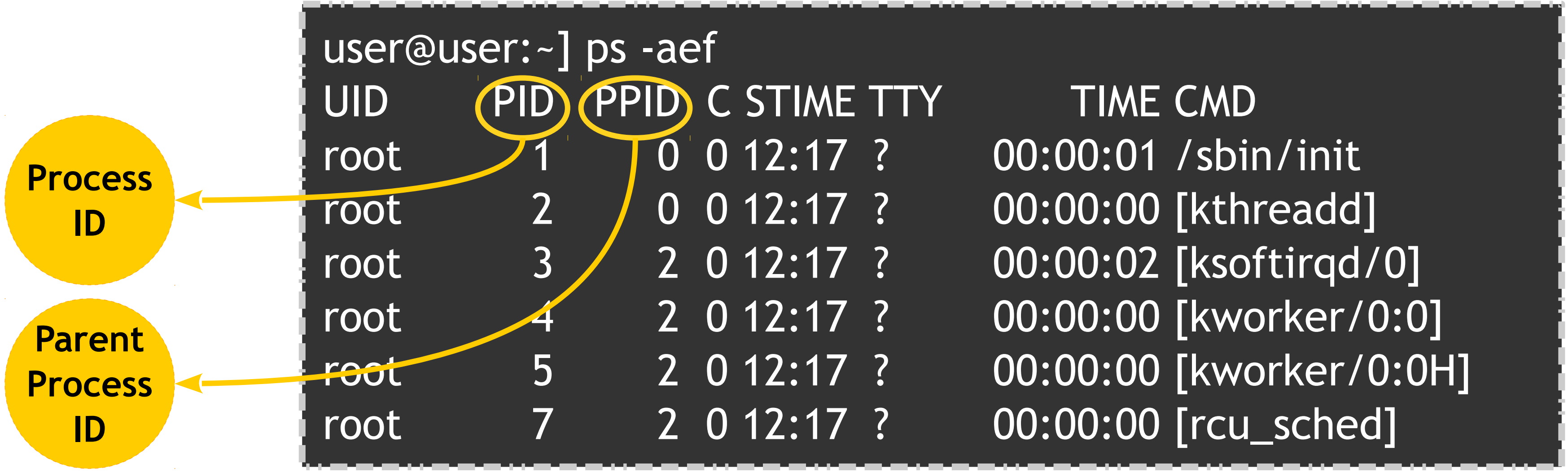
## Descriptor - ID

- Each process in a Linux system is identified by its unique process ID, sometimes referred to as PID
- Process IDs are numbers that are assigned sequentially by Linux as new processes are created
- Every process also has a parent process except the special init process
- Processes in a Linux system can be thought of as arranged in a tree, with the init process at its root
- The parent process ID or PPID, is simply the process ID of the process's parent

# Process

## Active Processes

- The *ps* command displays the processes that are running on your system
- By default, invoking ps displays the processes controlled by the terminal or terminal window in which ps is invoked
- For example (Executed as “ps –aef”):



```
user@user:~] ps -aef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 12:17 ?        00:00:01 /sbin/init
root         2     0  0 12:17 ?        00:00:00 [kthreadd]
root         3     2  0 12:17 ?        00:00:02 [ksoftirqd/0]
root         4     2  0 12:17 ?        00:00:00 [kworker/0:0]
root         5     2  0 12:17 ?        00:00:00 [kworker/0:0H]
root         7     2  0 12:17 ?        00:00:00 [rcu_sched]
```

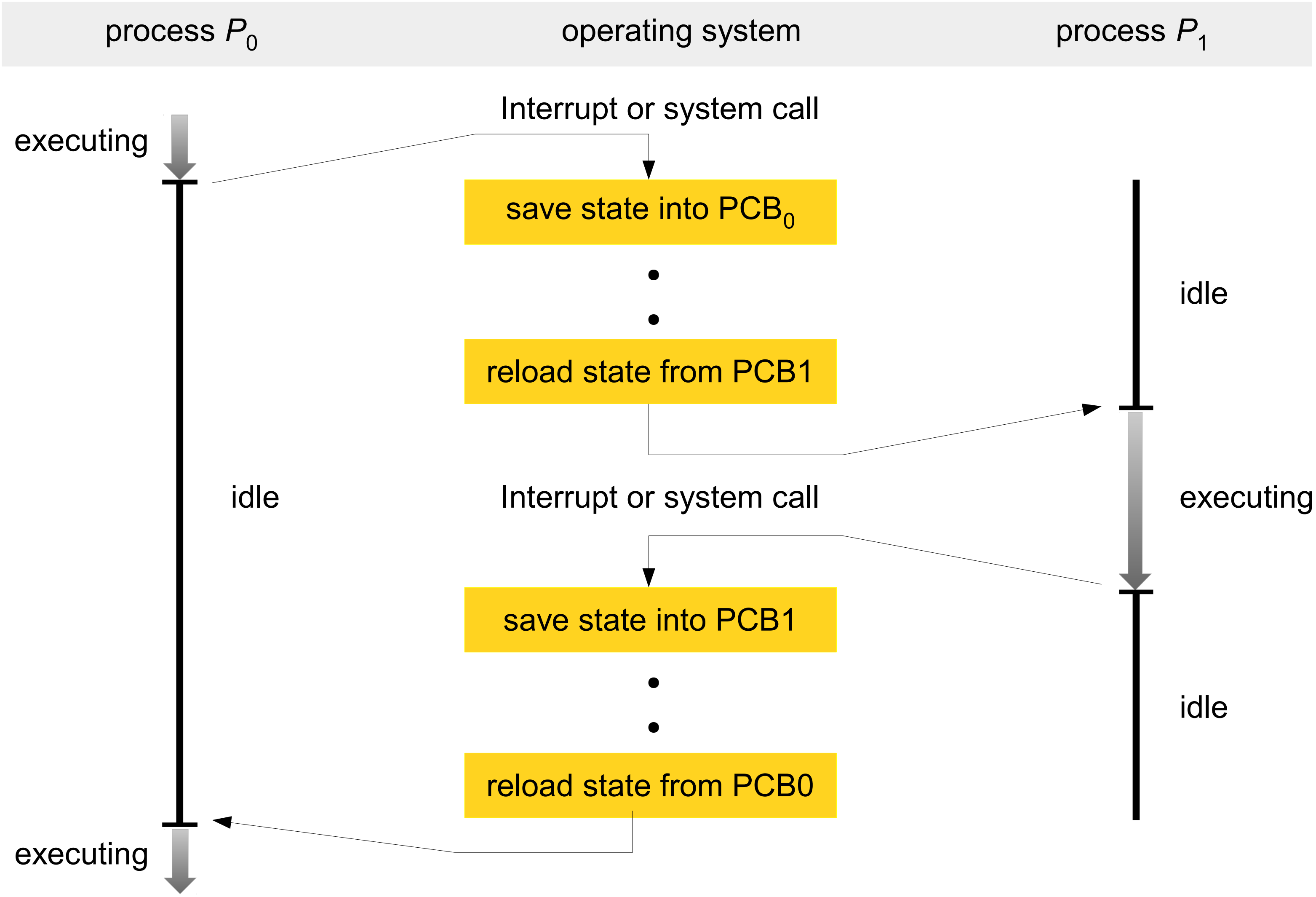
# Process

## Context Switching

- Switching the CPU to another task requires saving the state of the old task and loading the saved state for the new task
- The time wasted to switch from one task to another without any disturbance is called context switch or scheduling jitter
- After scheduling the new process gets hold of the processor for its execution

# Process

## Context Switching



# Process

## Creation

- Two common methods are used for creating new process
- Using system(): Relatively simple but should be used sparingly because it is inefficient and has considerably security risks
- Using fork() and exec(): More complex but provides greater flexibility, speed, and security

# Process

## Creation - system()

- It creates a sub-process running the standard shell
- Hands the command to that shell for execution
- Because the system function uses a shell to invoke your command, it's subject to the features and limitations of the system shell
- The system function in the standard C library is used to execute a command from within a program
- Much as if the command has been typed into a shell

# Process

## Creation - fork()

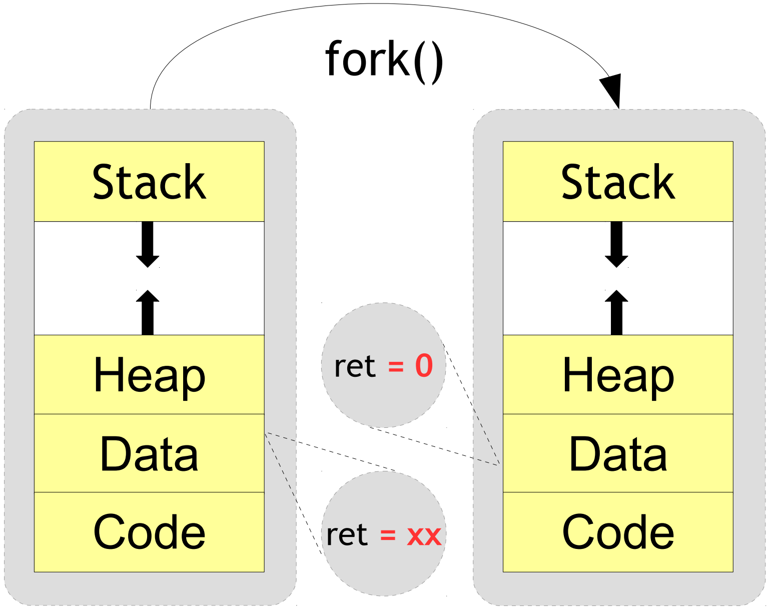
- fork makes a child process that is an exact copy of its parent process
- When a program calls fork, a duplicate process, called the child process, is created
- The parent process continues executing the program from the point that fork was called
- The child process, too, executes the same program from the same place
- All the statements after the call to fork will be executed twice, once, by the parent process and once by the child process

# Process

## Creation - fork()

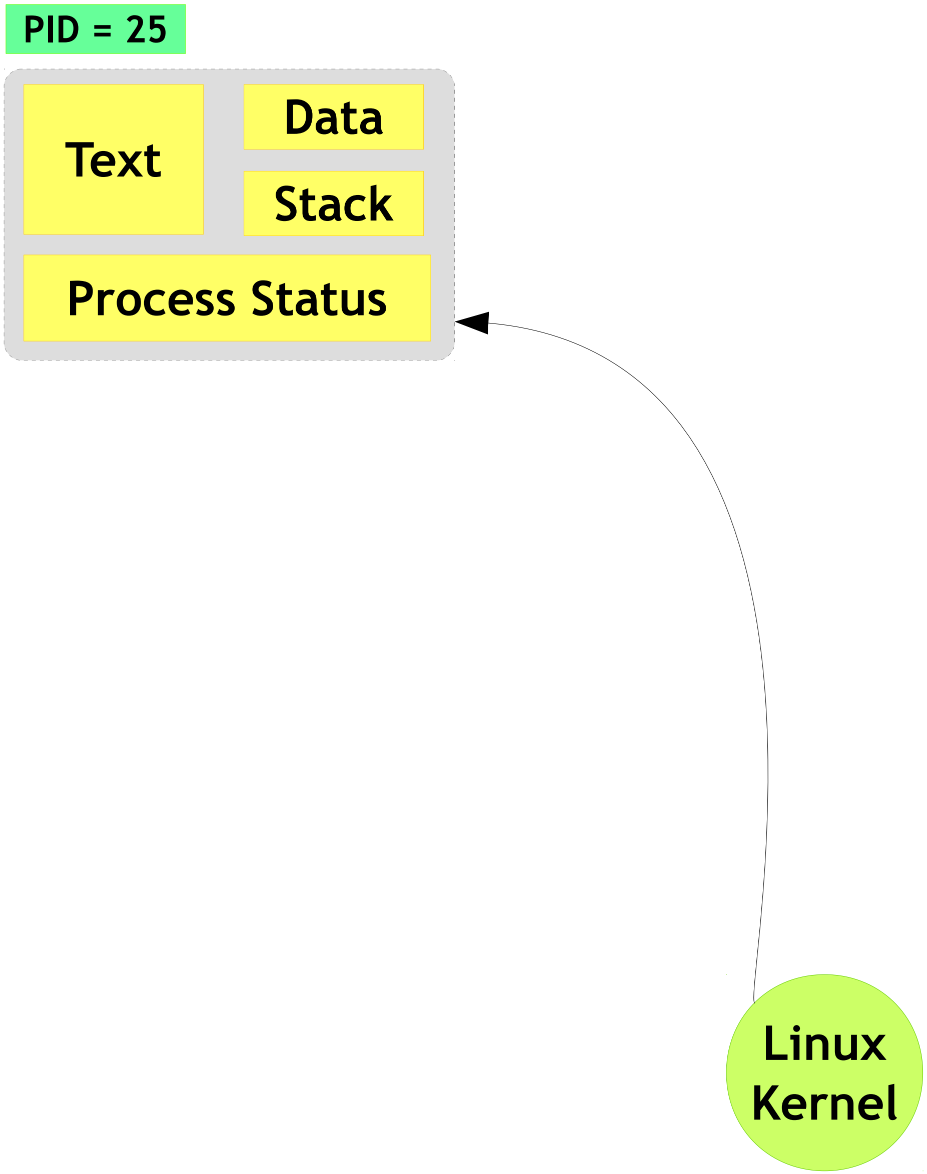
- The execution context for the child process is a copy of parent's context at the time of the call

```
int child_pid;
int child_status;

int main()
{
    int ret;

    ret = fork();
    switch (ret)
    {
        case -1:
            perror(“fork”);
            exit(1);
        case 0:
            <code for child process>
            exit(0);
        default:
            <code for parent process>
            wait(&child_status);
    }
}
```



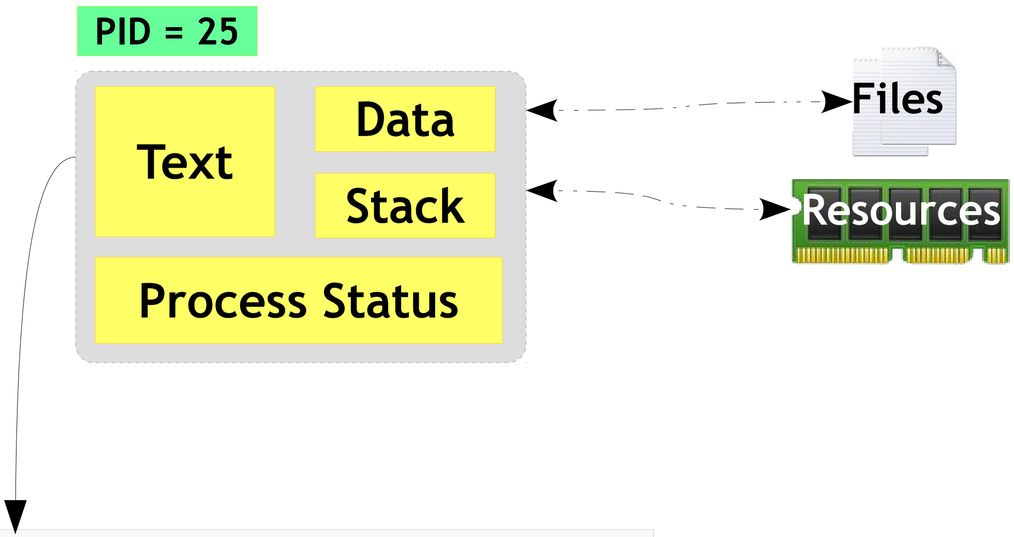
# Process

## fork() - The Flow



# Process

## fork() - The Flow

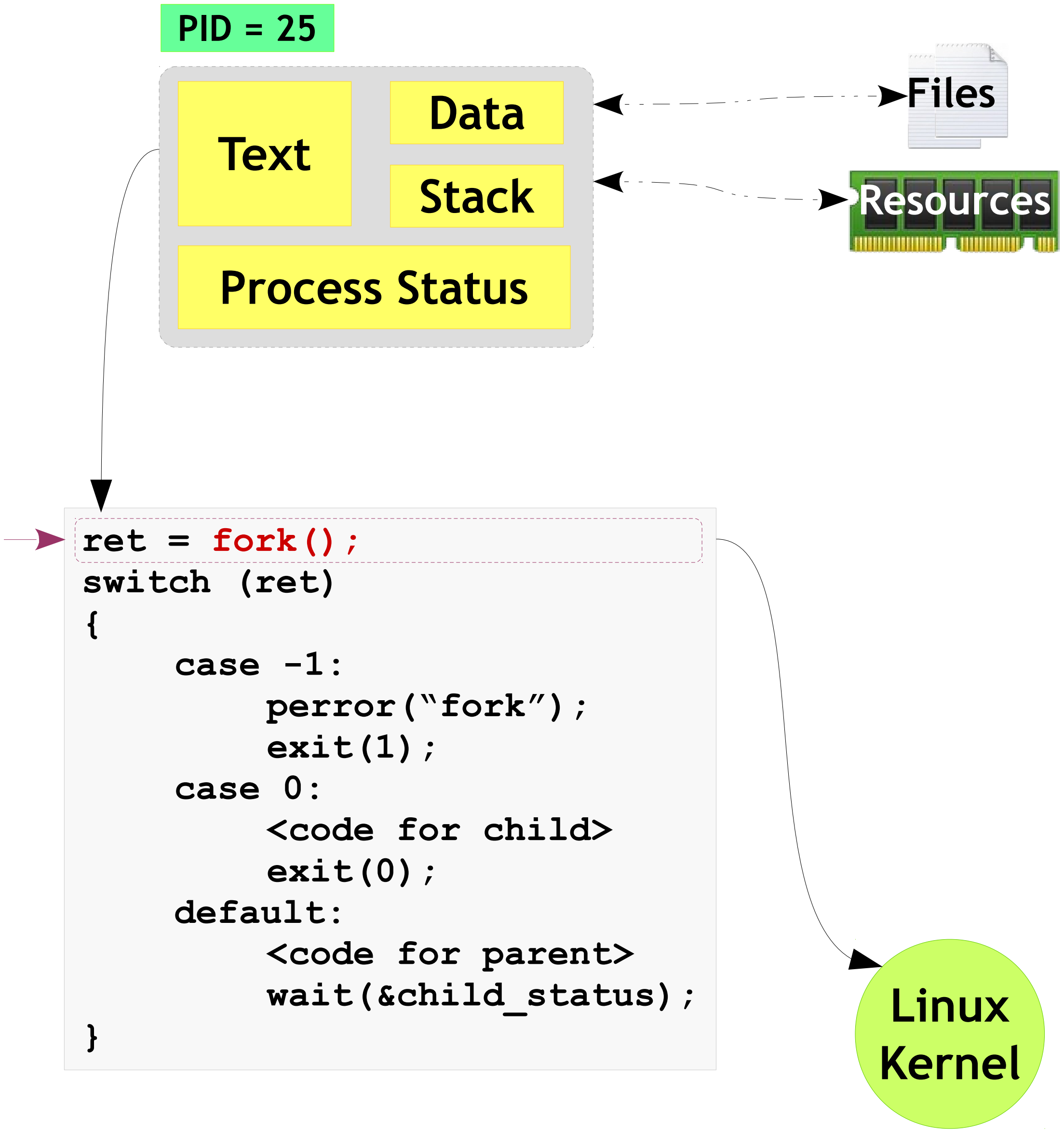


```
ret = fork();
switch (ret)
{
    case -1:
        perror(“fork”);
        exit(1);
    case 0:
        <code for child>
        exit(0);
    default:
        <code for parent>
        wait(&child_status);
}
```

Linux Kernel

# Process

## fork() - The Flow



PID = 25

Text

Data

Stack

Process Status

Files

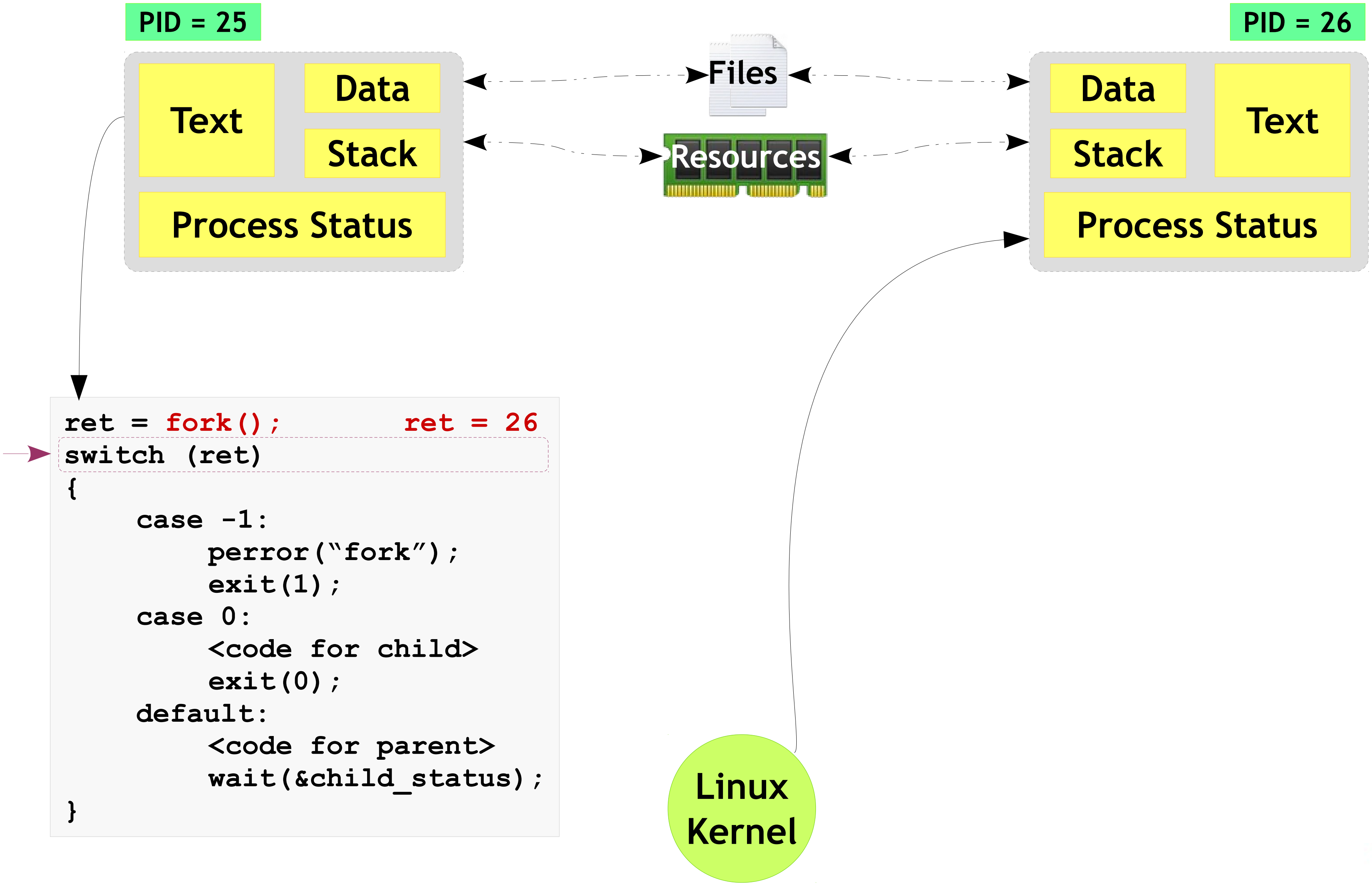
Resources

```
ret = fork();
switch (ret)
{
     case -1:
          perror(“fork”);
          exit(1);
     case 0:
          <code for child>
          exit(0);
     default:
          <code for parent>
          wait(&child_status);
}
```

Linux Kernel

# Process

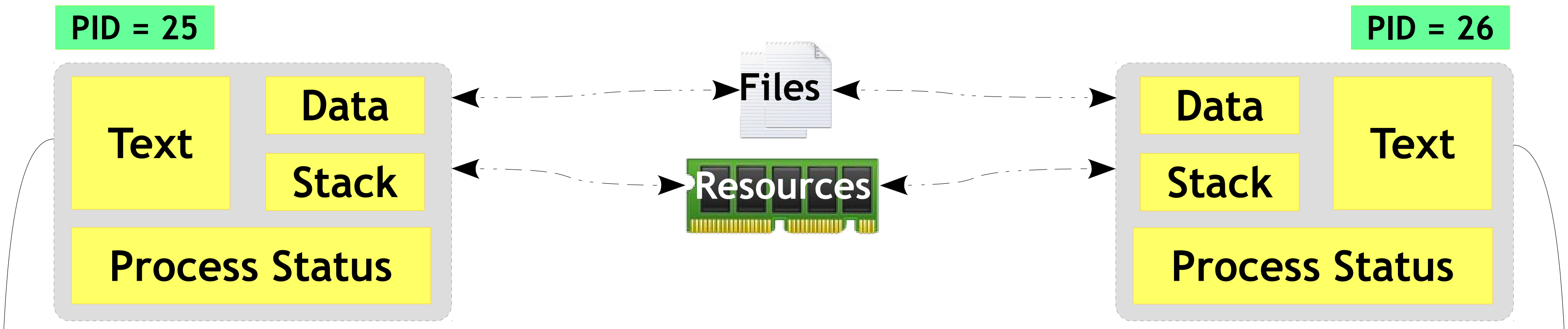
## fork() - The Flow



```
ret = fork();          ret = 26
switch (ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0:
        <code for child>
        exit(0);
    default:
        <code for parent>
        wait(&child_status);
}
```

# Process

## fork() - The Flow



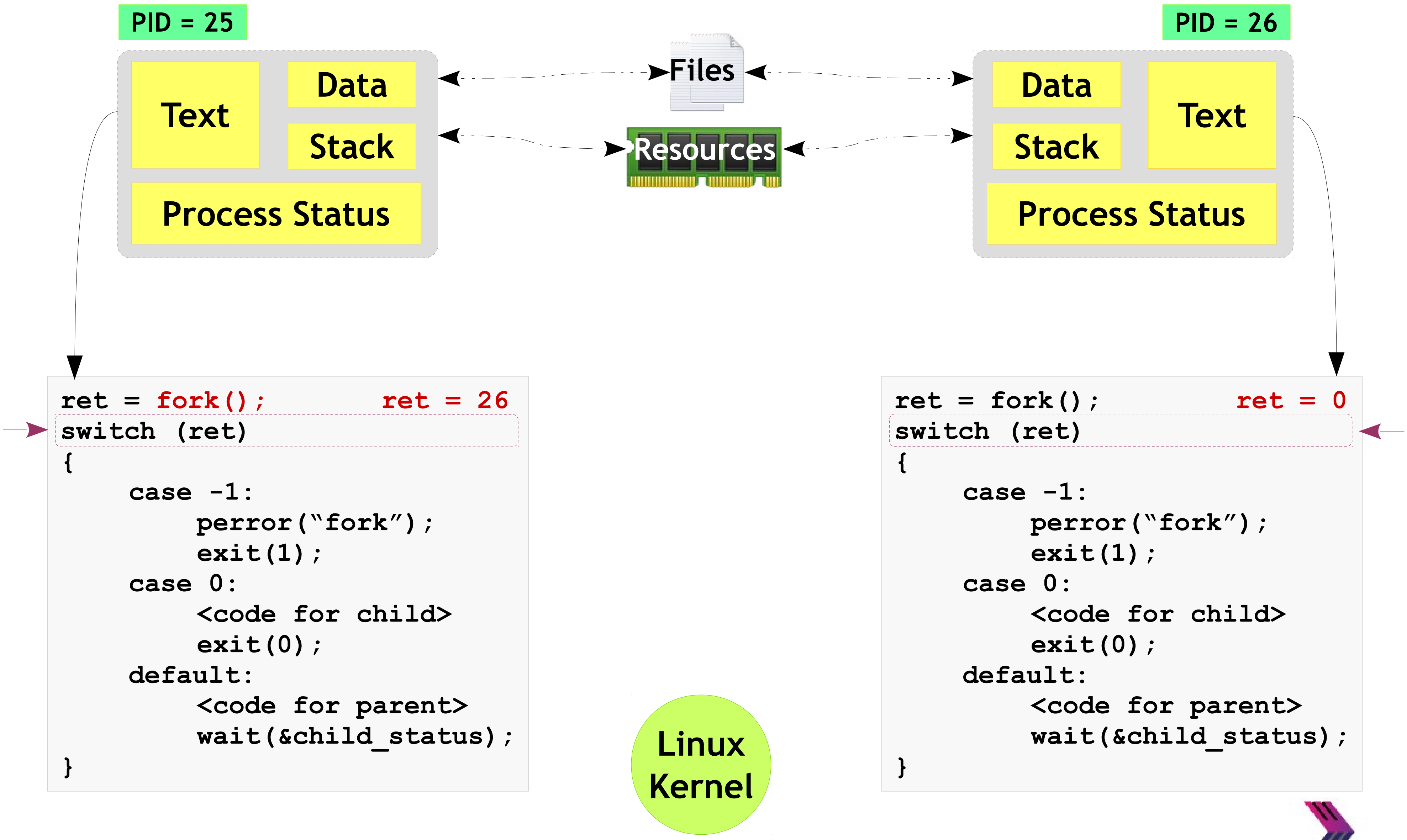
```
ret = fork();          ret = 26
switch (ret)
{
    case -1:
        perror(“fork”);
        exit(1);
    case 0:
        <code for child>
        exit(0);
    default:
        <code for parent>
        wait(&child_status);
}
```

Linux Kernel

```
ret = fork();
switch (ret)
{
    case -1:
        perror(“fork”);
        exit(1);
    case 0:
        <code for child>
        exit(0);
    default:
        <code for parent>
        wait(&child_status);
}
```

# Process

## fork() - The Flow



PID = 25

Text | Data | Stack

Process Status

Files

Resources

PID = 26

Data | Text | Stack

Process Status

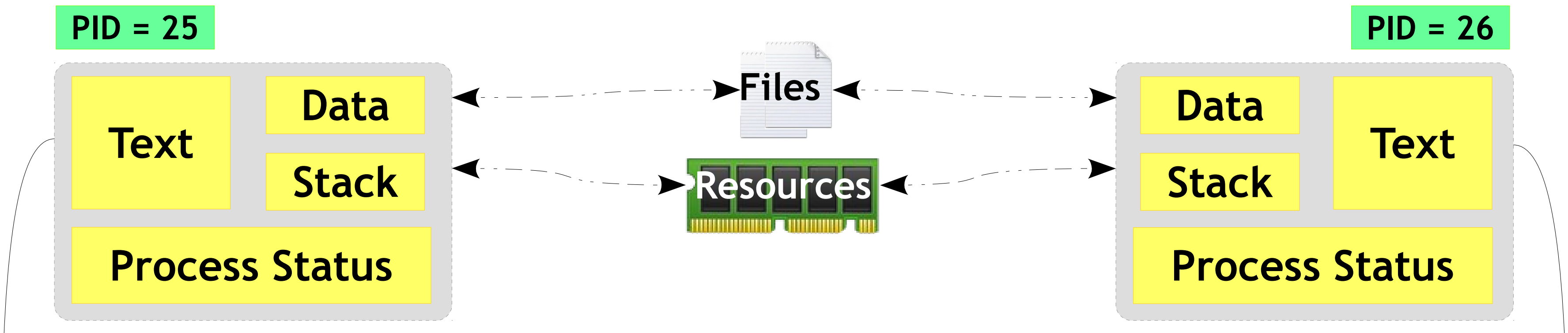
```
ret = fork();           ret = 26
switch (ret)
{
     case -1:
          perror("fork");
          exit(1);
     case 0:
          <code for child>
          exit(0);
     default:
          <code for parent>
          wait(&child_status);
}
```

```
ret = fork();             ret = 0
switch (ret)
{
     case -1:
          perror("fork");
          exit(1);
     case 0:
          <code for child>
          exit(0);
     default:
          <code for parent>
          wait(&child_status);
}
```

Linux Kernel

# Process

## fork() - The Flow



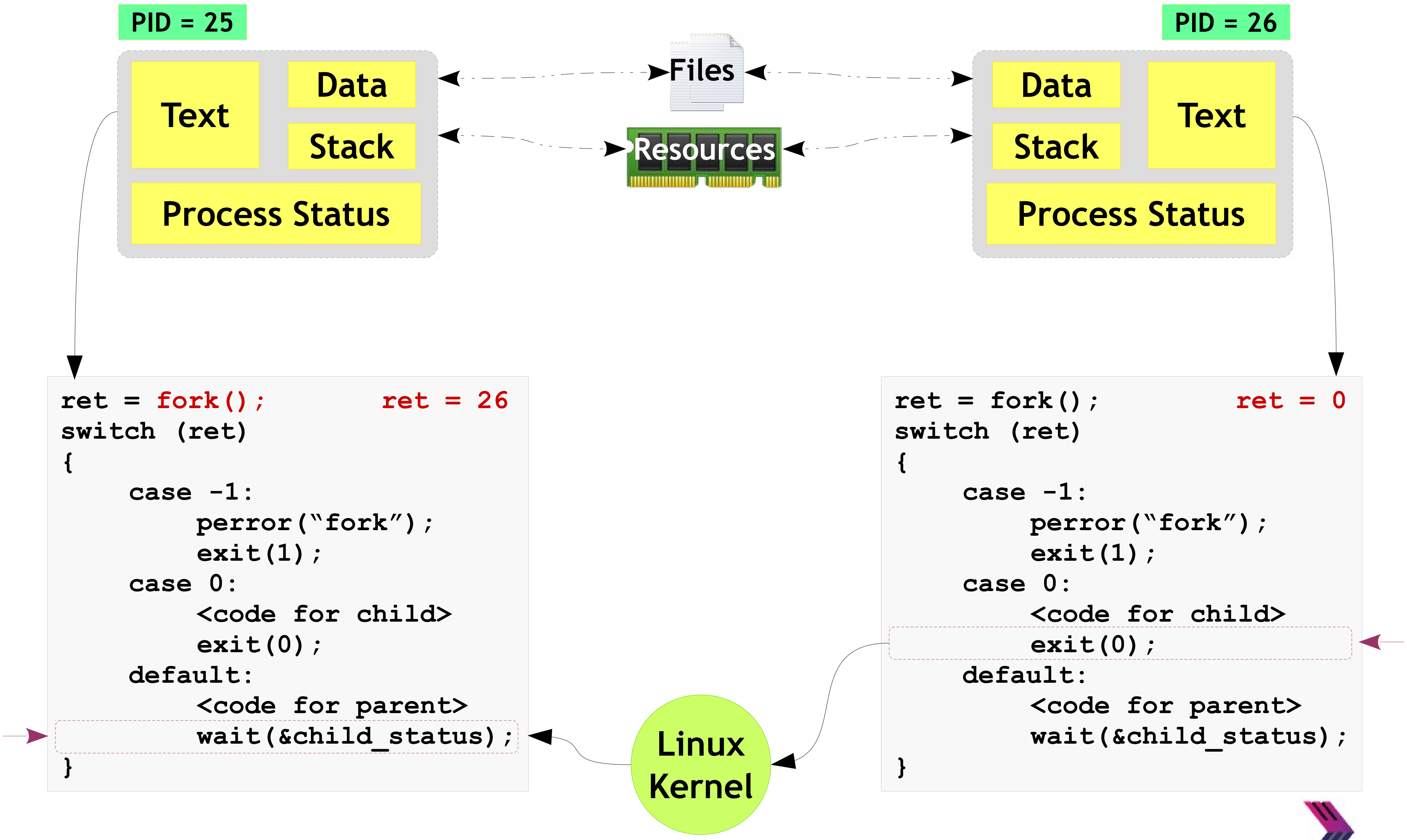
```
ret = fork();          ret = 26
switch (ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0:
        <code for child>
        exit(0);
    default:
        <code for parent>
        wait(&child_status);
}
```

```
ret = fork();           ret = 0
switch (ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0:
        <code for child>
        exit(0);
    default:
        <code for parent>
        wait(&child_status);
}
```

Linux Kernel

# Process

## fork() - The Flow

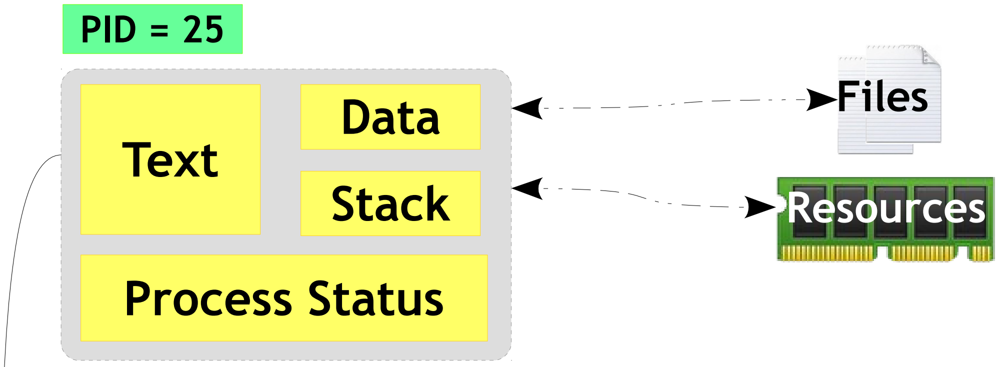


```
ret = fork();          ret = 26
switch (ret)
{
     case -1:
          perror(“fork”);
          exit(1);
     case 0:
          <code for child>
          exit(0);
     default:
          <code for parent>
          wait(&child_status);
}
```

```
ret = fork();          ret = 0
switch (ret)
{
     case -1:
          perror(“fork”);
          exit(1);
     case 0:
          <code for child>
          exit(0);
     default:
          <code for parent>
          wait(&child_status);
}
```

# Process

## fork() - The Flow



```
ret = fork();         ret = 26
switch (ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0:
        <code for child>
        exit(0);
    default:
        <code for parent>
        wait(&child_status);
}
```

Linux Kernel

# Process

## fork() - How to Distinguish?

- First, the child process is a new process and therefore has a new process ID, distinct from its parent’s process ID
- One way for a program to distinguish whether it’s in the parent process or the child process is to call getpid
- The fork function provides different return values to the parent and child processes
- One process “goes in” to the fork call, and two processes “come out,” with different return values
- The return value in the parent process is the process ID of the child
- The return value in the child process is zero

# Process

## Overlay - exec()

- The exec functions replace the program running in a process with another program
- When a program calls an exec function, that process immediately ceases executing and begins executing a new program from the beginning
- Because exec replaces the calling program with another one, it never returns unless an error occurs
- This new process has the same PID as the original process, not only the PID but also the parent process ID, current directory, and file descriptor tables (if any are open) also remain the same
- Unlike fork, exec results in still having a single process

# Process
## Termination

- When a parent forks a child, the two process can take any turn to finish themselves and in some cases the parent may die before the child
- In some situations, though, it is desirable for the parent process to wait until one or more child processes have completed
- This can be done with the wait() family of system calls.
- These functions allow you to wait for a process to finish executing, enable parent process to retrieve information about its child's termination

# Process
## Wait

- There are four different system calls in wait family
- Simplest one is wait(). It blocks the calling process until one of its child processes exits (or an error occurs).
- It returns a status code via an integer pointer argument, from which you can extract information about how the child process exited.
- The waitpid function can be used to wait for a specific child to exit, instead of any child process.
- The wait3 function returns resource usage information about the exiting child process.

# Process
## Zombie

- Zombie process is a process that has terminated but has not been cleaned up yet
- It is the responsibility of the parent process to clean up its zombie children
- If the parent does not clean up its children, they stay around in the system, as zombie
- When a program exits, its children are inherited by a special process, the init program, which always runs with process ID of 1 (it's the first process started when Linux boots)
- The init process automatically cleans up any zombie child processes that it inherits.

# Inter Process Communications (IPC)

# Inter Process Communications
## Introduction

- *Inter process communication (IPC)* is the mechanism whereby one process can communicate, that is exchange data with another processes
- Example, you may want to print the filenames in a directory using a command such as ls | lpr
- The shell creates an ls process and separate lpr process, connecting the two with a pipe, represented by the “|” symbol.

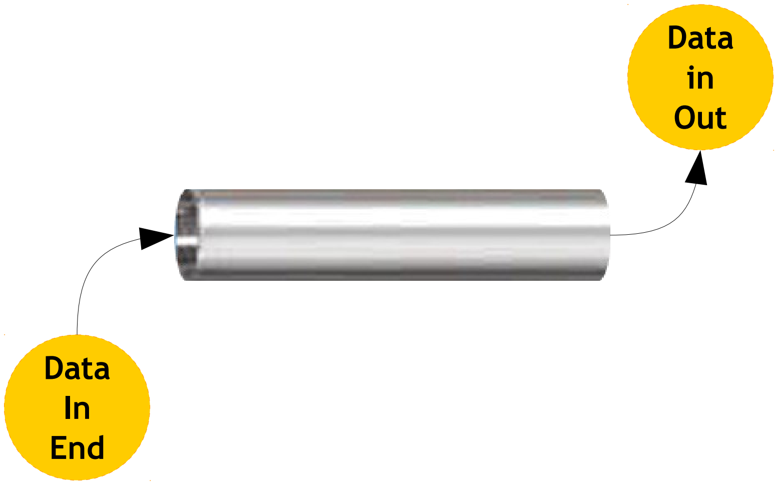
# Pipes

# Inter Process Communications

## Pipes

- A pipe is a communication device that permits unidirectional communication
- Data written to the “write end” of the pipe is read back from the “read end”
- Pipes are serial devices; the data is always read from the pipe in the same order it was written

Water In End
Water Out End



# Inter Process Communications

## Pipes - Creation

- To create a pipe, invoke the pipe system call
- Supply an integer array of size 2
- The call to pipe stores the reading file descriptor in array position 0
- Writing file descriptor in position 1

# FIFO

# Inter Process Communications

## FIFO - Properties

- A *first-in, first-out (FIFO)* file is a pipe that has a name in the file-system
- FIFO file is a pipe that has a name in the file-system
- FIFOs are also called Named Pipes
- FIFOs is designed to let them get around one of the shortcomings of normal pipes

# Inter Process Communications

## FIFO vs Pipes

- Unlike pipes, FIFOs are not temporary objects, they are entities in the file-system
- Any process can open or close the FIFO
- The processes on either end of the pipe need not be related to each other
- When all I/O is done by sharing processes, the named pipe remains in the file system for later use

# Inter Process Communications

## FIFO - Creation

- FIFO can also be created using

  mknod(“myfifo”, S_IFIFO | 0644, 0);
- The FIFO file will be called “myfifo”
- Creation mode (permission of pipe)
- Finally, a device number is passed. This is ignored when creating a FIFO, so you can put anything you want in there.

# Inter Process Communications

## FIFO - Access

- Access a FIFO just like an ordinary file
- To communicate through a FIFO, one program must open it for writing, and another program must open it for reading
- Either low-level I/O functions (open, write, read, close and so on) or C library I/O functions (fopen, fprintf, fscanf, fclose, and so on) may be used.

# Inter Process Communications

## FIFO – Access Example

- For example, to write a buffer of data to a FIFO using low-level I/O routines, you could use this code:

```
int fd = open(fifo_path, O_WRONLY);
write(fd, data, data_length);
close(fd);
```

- To read a string the FIFO using C library I/O functions, you could use this code:

```
FILE* fifo = fopen(fifo_path, “r”);
fscanf(fifo, “%s”, buffer);
fclose(fifo);
```

# Inter Process Communications

## Broken Pipe

- In the previous examples, terminate read while write is still running. This creates a condition called “Broken Pipe”.
- What has happened is that when all readers for a FIFO close and the writers is still open, the write will receive the signal SIGPIPE the next time it tries to write().
- The default signal handler prints “Broken Pipe” and exits. Of couse, you can handle this more gracefully by catching SIGPIPE through the signa()l call.

# Message Queues

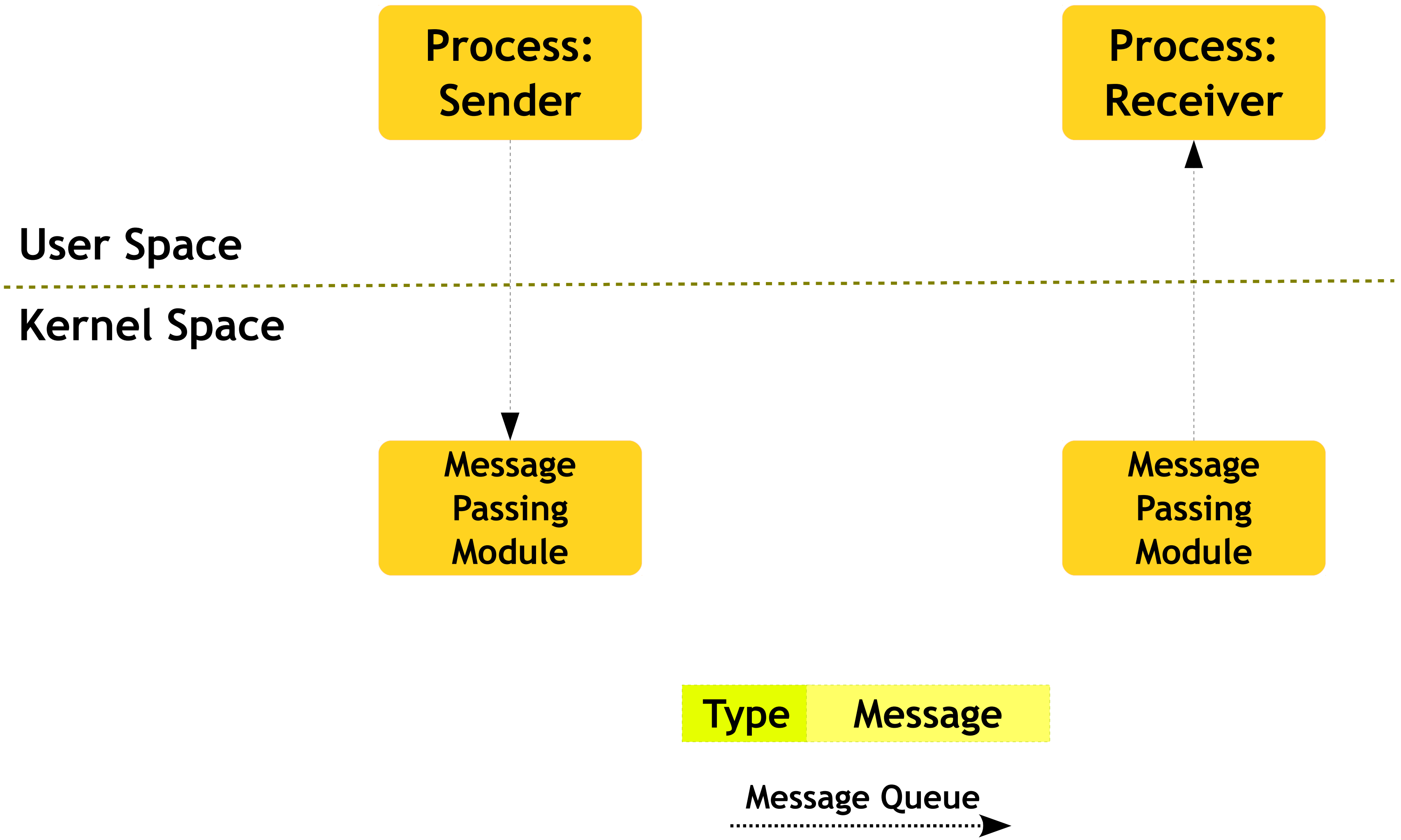
# Inter Process Communications

## Message Queues

- Message queues are two way IPC mechanism for communicating structured messages
- Works well for applications like protocols where there is a meaning behind every message
- Asynchronous communication mechanism, applied in group applications
- Queue full and queue empty situations
- Automatic synchronizations

# Inter Process Communications

## Message Queues - Flow



# Shared Memory

# Inter Process Communications

## Shared Memories

- Shared memory allows two or more processes to access the same memory
- When one process changes the memory, all the other processes see the modification
- Shared memory is the fastest form of Inter process communication because all processes share the same piece of memory
- It also avoids copying data unnecessarily

# Inter Process Communications

## Shared Memories - Procedure

- To start with one process must allocate the segment
- Each process desiring to access the segment must attach to it
- Reading or Writing with shared memory can be done only after attaching into it
- After use each process detaches the segment
- At some point, one process must de-allocate the segment

# Inter Process Communications

## Shared Memories – Process & Memory

- Under Linux, each process's virtual memory is split into pages.
- Each process maintains a mapping from its memory address to these virtual memory pages, which contain the actual data.
- Even though each process has its own addresses, multiple processes mappings can point to the same page, permitting sharing of memory.

# Inter Process Communications

## Shared Memories – Procedure

- Allocating a new shared memory segment causes virtual memory pages to be created.
- Because all processes desire to access the same shared segment, only one process should allocate a new shared segment
- Allocating an existing segment does not create new pages, but it does return an identifier for the existing pages
- To permit a process to use the shared memory segment, a process attaches it, which adds entries mapping from its virtual memory to the segment's shared pages

# Inter Process Communications

## Shared Memories – Example

- This invocation of the shmget creates a new shared memory (or access to an existing one, if shm_key is already used) that;s readable and writable to the owner but not other users

```
int segment_id;

segment_id = shmget(shm_key, getpagesize(), IPC_CREAT | S_IRUSR | S_IWUSR);
```

- If the call succeeds, shmget returns a segment identifier

# Socket

# Sockets

- A sockets is communication mechanism that allow client / server system to be developed either locally on a single machine or across networks.
- It is well defined method of connecting two processes locally or across networks

# Sockets

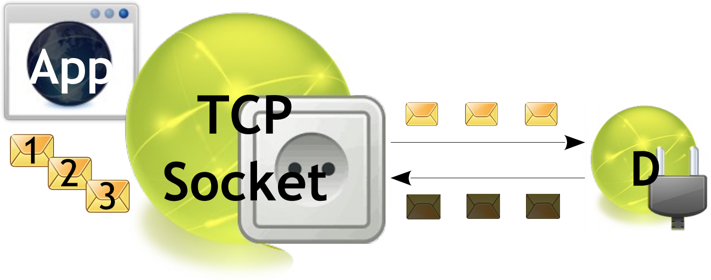
## The APIs

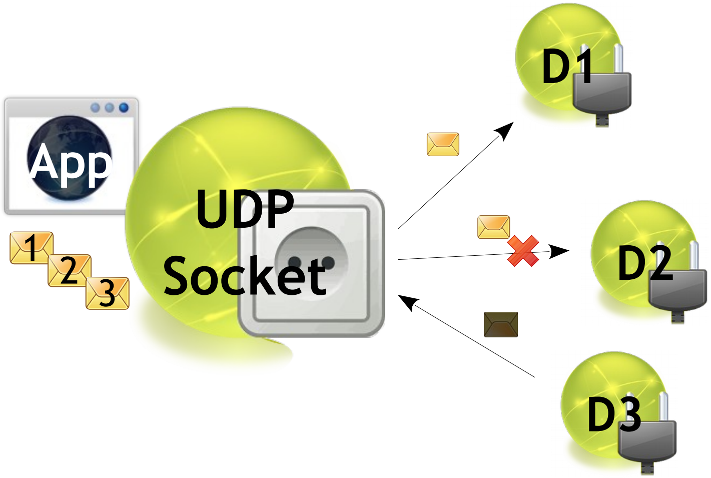
- int socket(int domain, int type, int protocol);
  - Domain
    - AF_UNIX, AF_INET, AF_INET6 etc.
  - Type
    - SOCK_STREAM, SOCK_DGRAM, SOCK_RAW
- int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
- int listen(int sockfd, int backlog);
- int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
- int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);

# Sockets

## Types - TCP and UDP

| TCP socket (SOCK_STREAM) | UDP socket (SOCK_DGRAM) |
|---|---|
| • Connection oriented TCP<br>• Reliable delivery<br>• In-order guaranteed<br>• Three way handshake<br>• More network BW | • Connectionless UDP<br>• Unreliable delivery<br>• No-order guarantees<br>• No notion of "connection"<br>• Less network BW |





# Stay connected

**About us:** Emertxe is India's one of the top IT finishing schools & self learning kits provider. Our primary focus is on Embedded with diversification focus on Java, Oracle and Android areas

**Branch Office:**
Emertxe Information Technologies,
No-1, 9th Cross, 5th Main,
Jayamahal Extension,
Bangalore, Karnataka 560046

**Corporate Headquarters:**
Emertxe Information Technologies,
83, Farah Towers, 1st Floor,
MG Road,
Bangalore, Karnataka - 560001
T: +91 809 555 7333 (M), +91 80 41289576 (L)
E: training@emertxe.com

https://www.facebook.com/Emertxe

https://twitter.com/EmertxeTweet

https://www.slideshare.net/EmertxeSlides

# Thank You