

Linux Internals & Networking

System programming using Kernel interfaces

Team Emertxe



Contents



Linux Internals & Networking

Contents

- .Introduction
- .Transition to OS programmer
- .System Calls
- .Process
- .IPC
- .Signals
- .Networking
- .Threads
- .Synchronization
- .Process Management
- .Memory Management



Process

Process



- Running instance of a program is called a **PROCESS**
- On a single-user system, a user may be able to run several programs at one time:
 - a word processor,
 - a Web browser,
 - gcc compiler etc...
- A process includes
 - the stack
 - a data section
 - a heap

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

Team Emertxe



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
- One program can be several processes
- Consider multiple users executing the same program

Factor	Process	Program
Storage	Dynamic Memory	Secondary Memory
State	Active	Passive

Process vs Program



Program

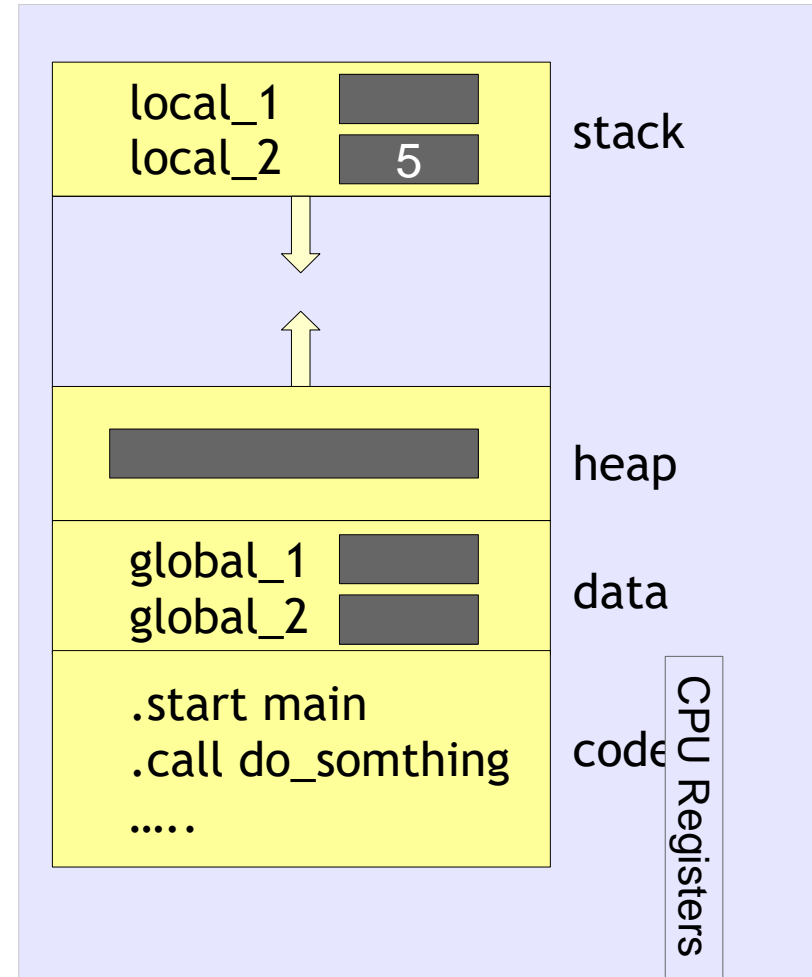
```
int global_1 = 0;
int global_2 = 0;

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

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

    do_somthing();
    ....
}
```

Task

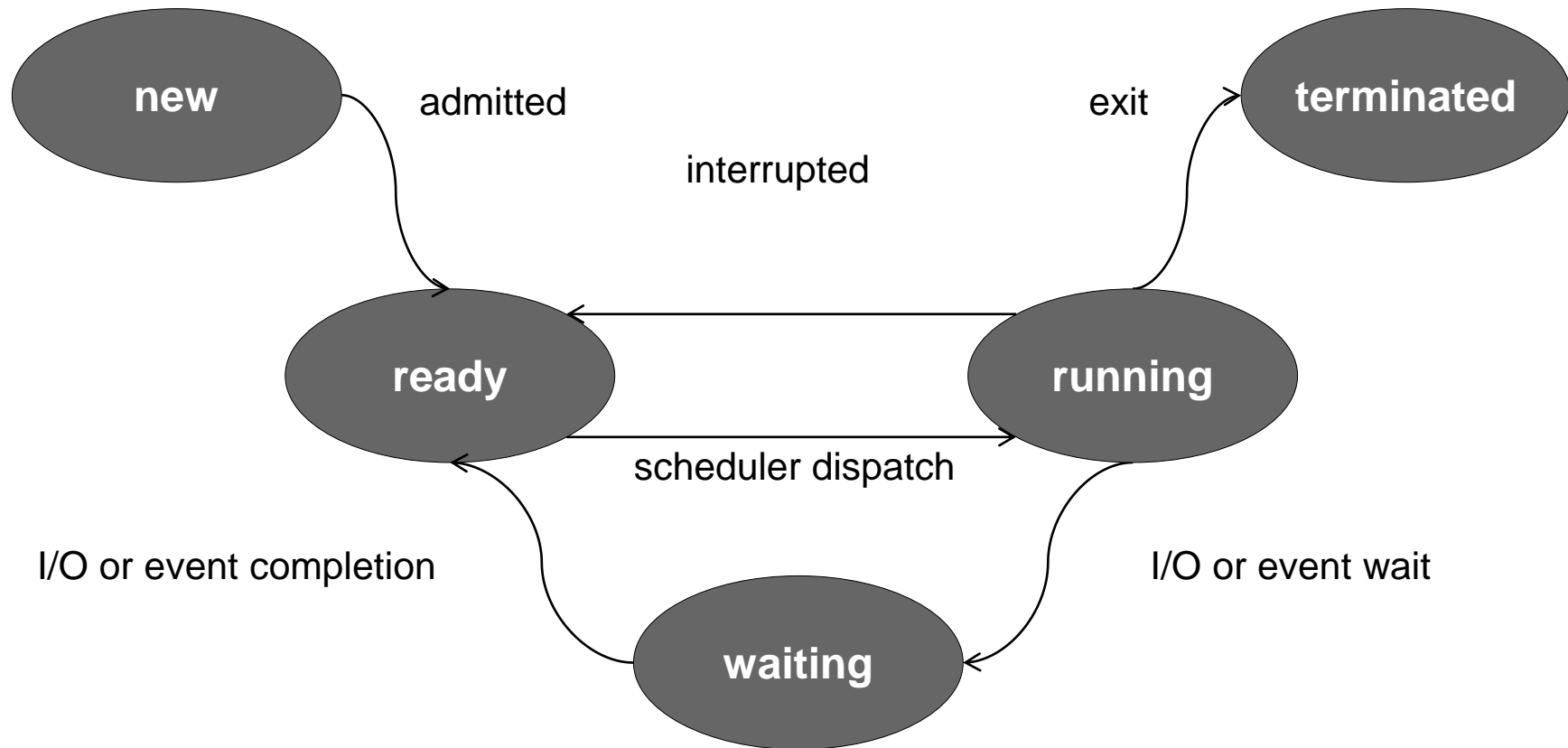


Team Emertxe



Process

State Transition Diagram

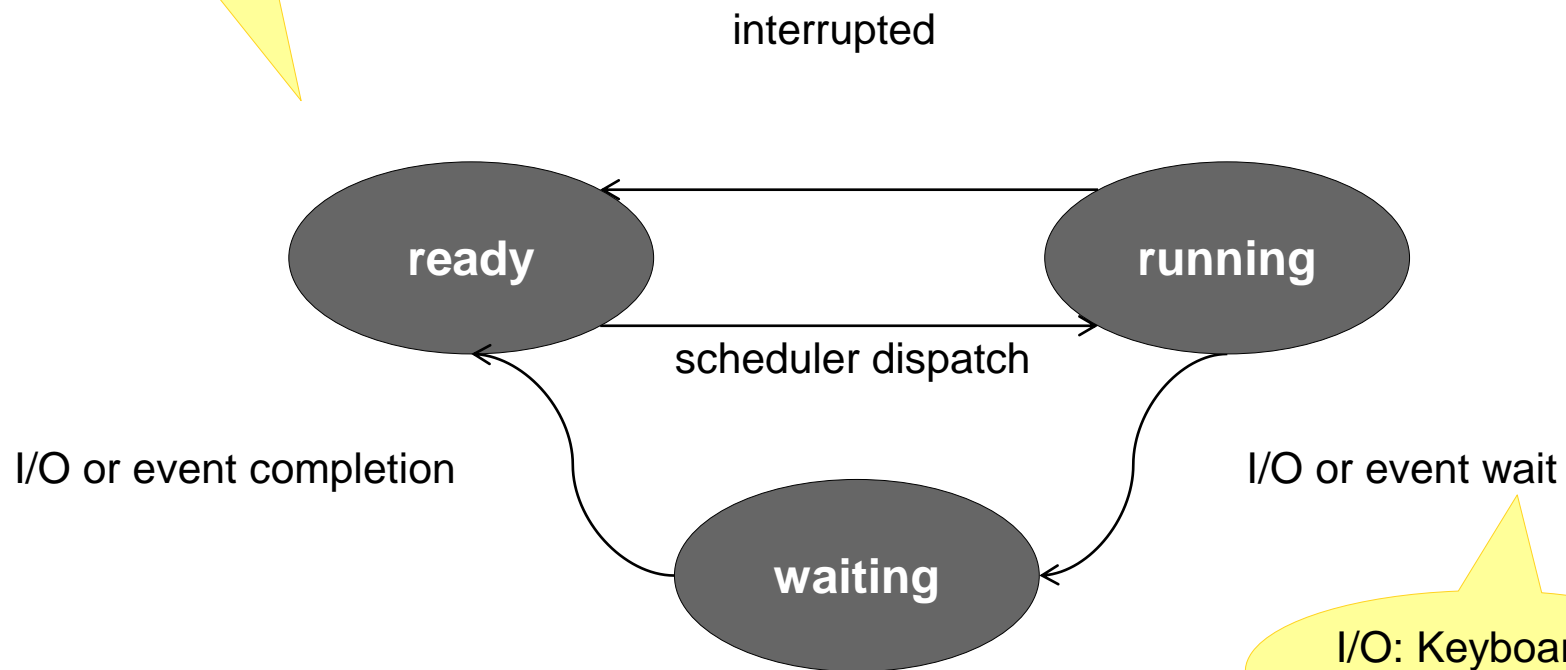


Process

State Transition Diagram



Priority
Round Robin
FCFS
Preemptive



I/O: Keyboard
Even: Signal



•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



.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 (PCB / TCB)



Pointer	Process State
Process ID	
Program Counter	
Registers	
Memory Limits	
List of Open Files	
.	
.	
.	
.	
.	
.	

.Information associated with each process.

–Process state

.running, waiting, etc

–Program counter

.location of instruction to next execute

–CPU registers

.contents of all process centric registers

–CPU scheduling information

.priorities, scheduling queue pointers

–Memory-management information

.memory allocated to the process

–Accounting Information

PCB: Process Control Block / TCB: Task Control Block

–I/O Status Information

.I/O devices allocated to process, list of open files

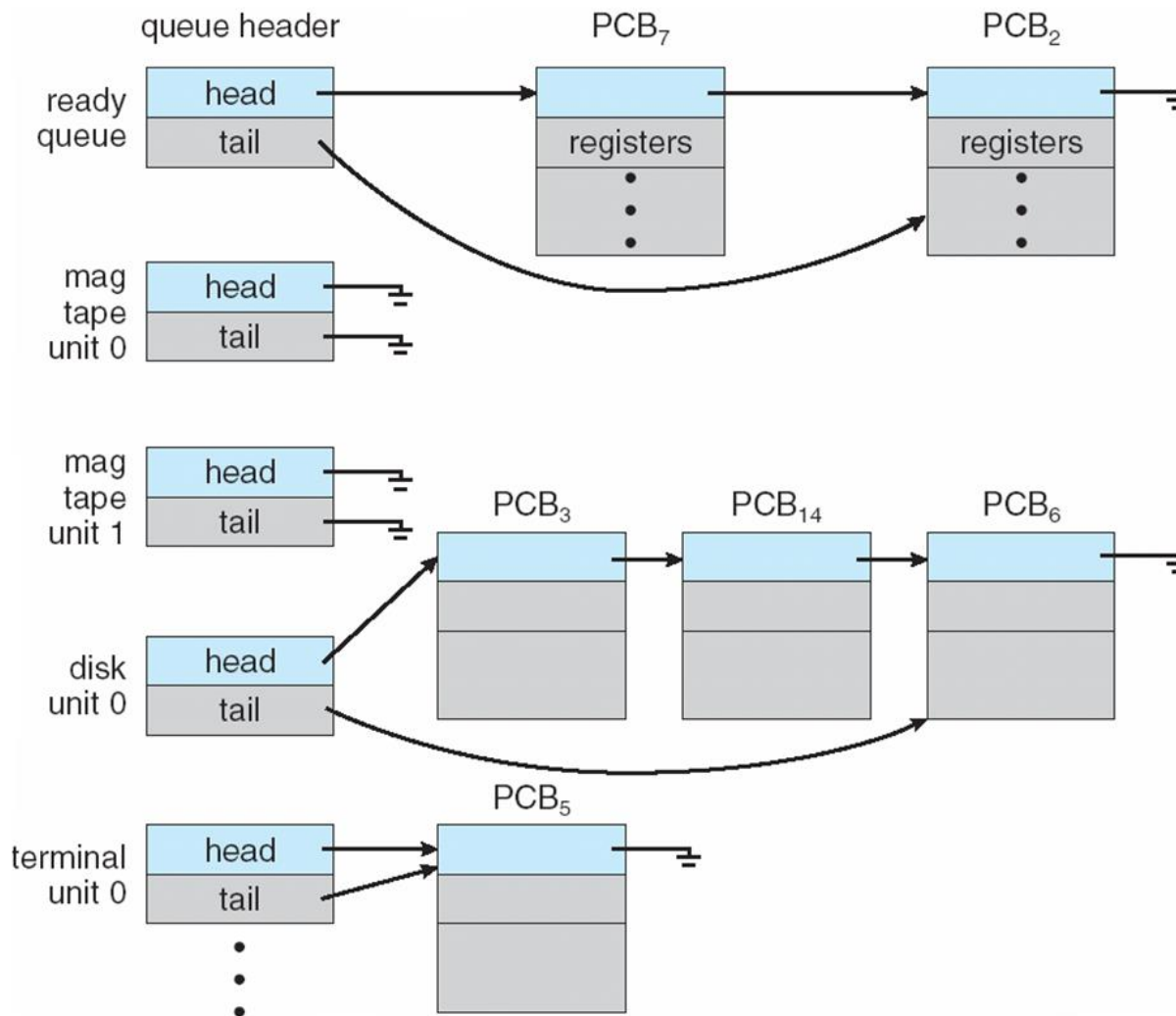
Process Scheduling



- Maximize CPU use, quickly switch processes onto CPU for time sharing
- Process scheduler selects among available processes for next execution on CPU
- Maintains scheduling queues of processes
 - Job queue – set of all processes in the system
 - Ready queue – set of all processes residing in main memory, ready and waiting to execute
 - Device queues – set of processes waiting for an I/O device
- Processes migrate among the various queues

Process

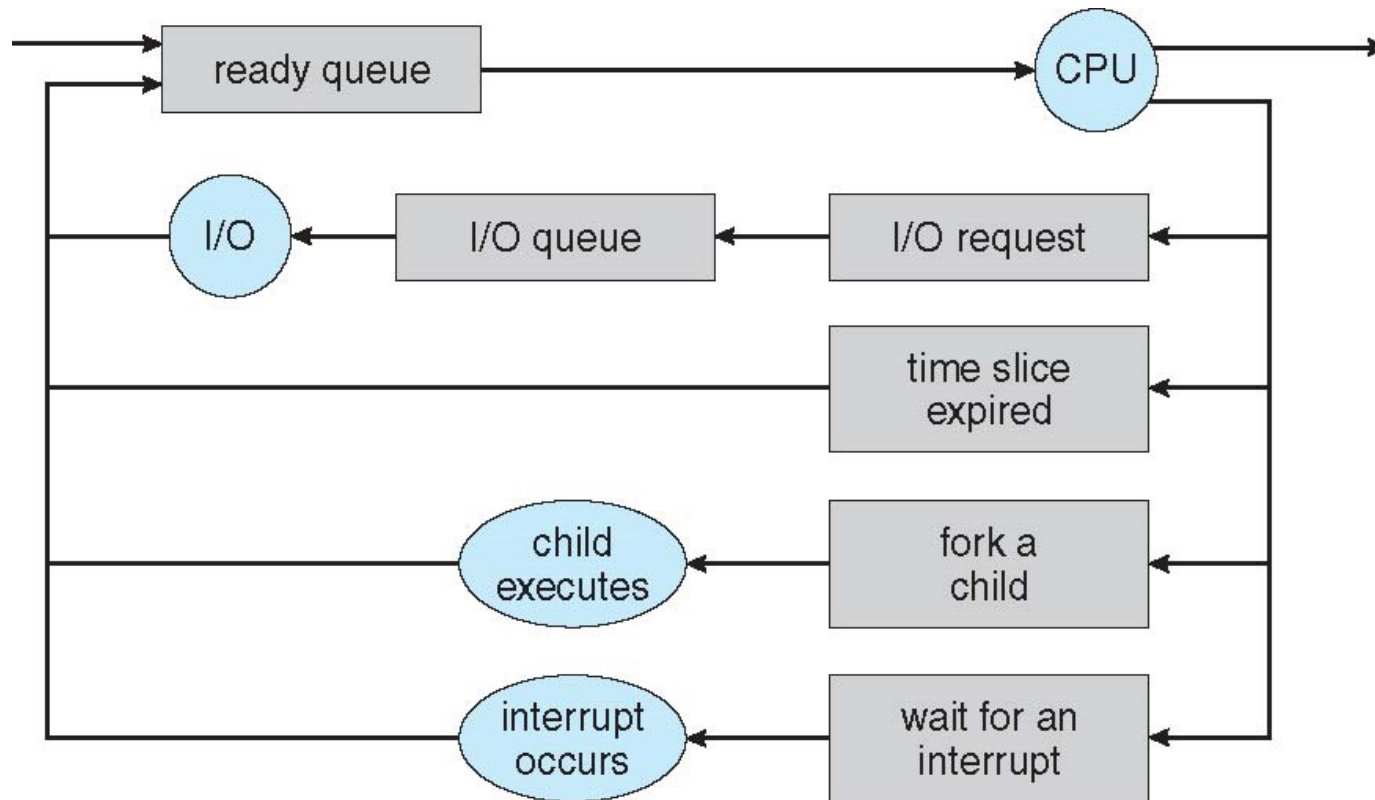
Ready & Various I/O Device Queues



Process

Scheduling – Queueing Diagram

Queueing diagram represents queues, resources, flows



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()

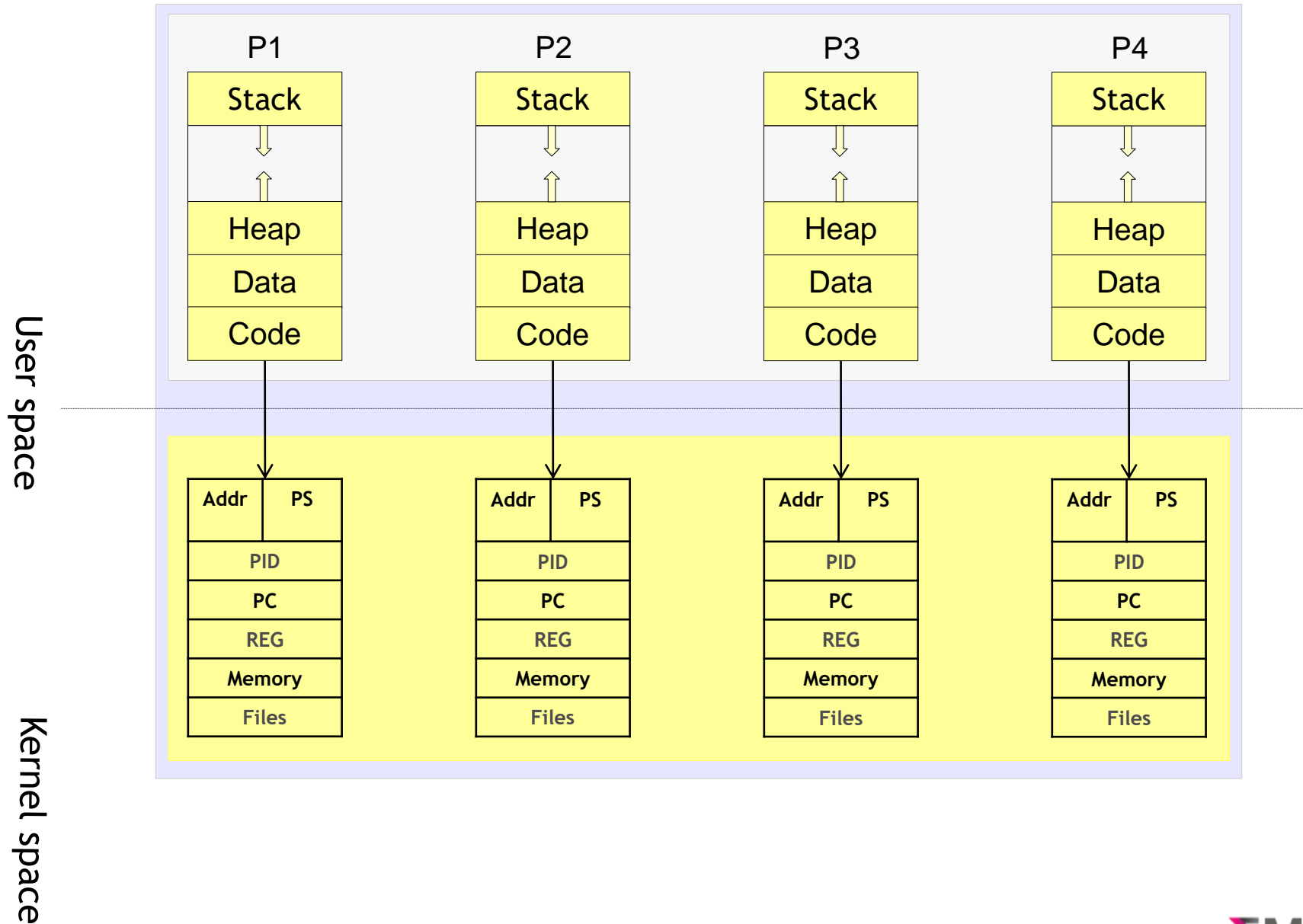
Process

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 Schedule



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]
```

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]

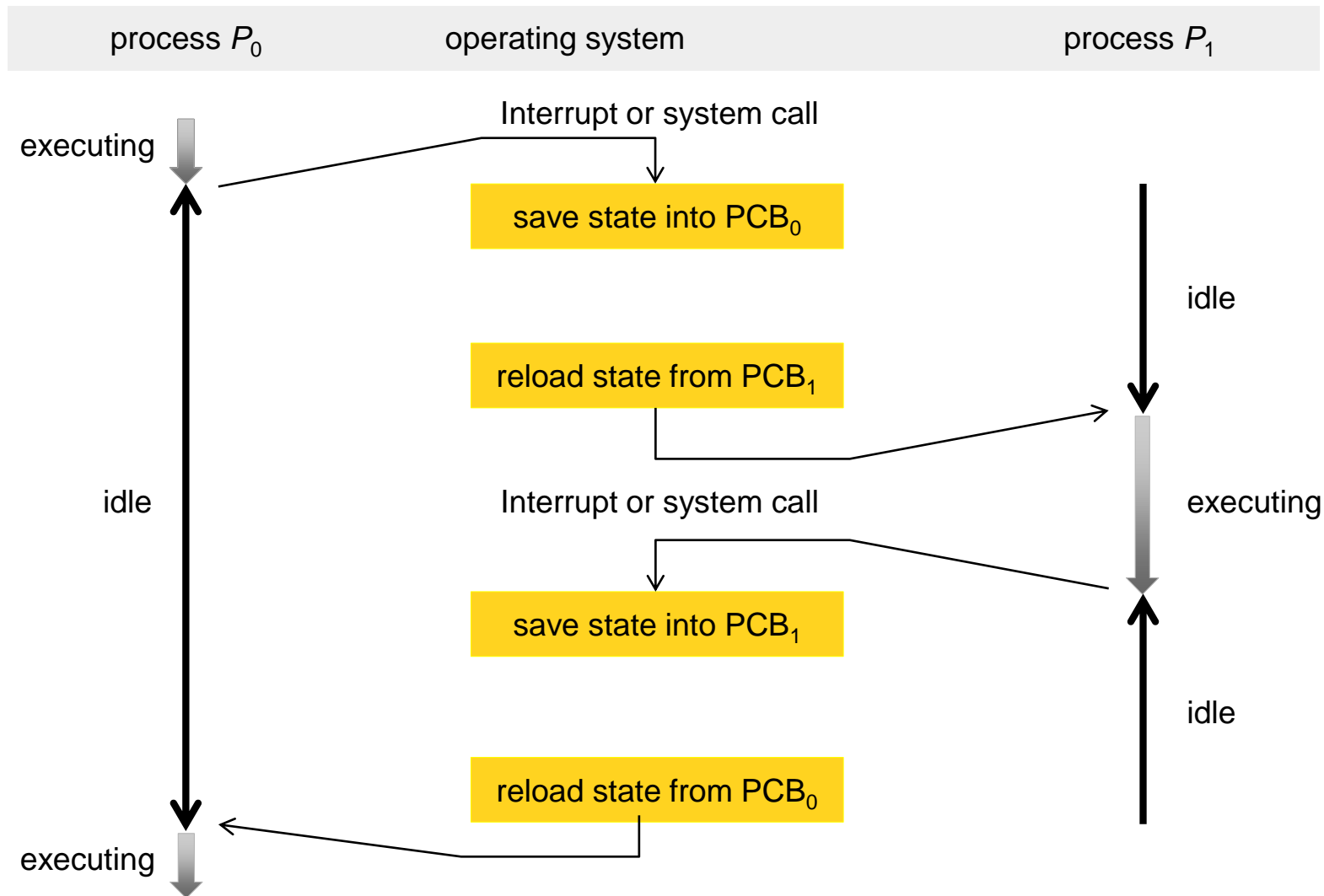
Process

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

Context Switching





.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.
- The child obtains copies of the parent's stack, data, heap, and text segments.

Process

Creation - fork()

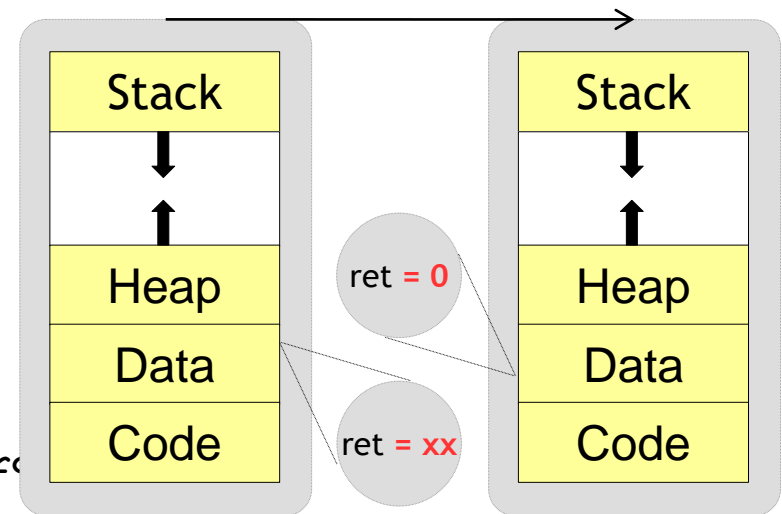


.The execution context for the child process is a copy of parent's context at the time of the call

```
int child_pid;
int child_status;

int main()
{
    int ret;

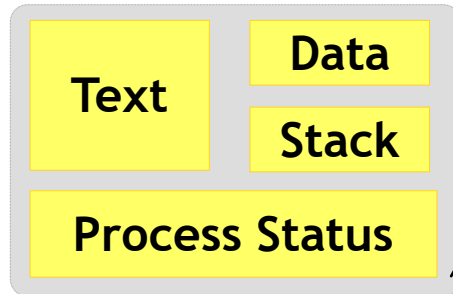
    ret = fork();
    switch (ret)
    {
        case -1:
            perror("fork");
            exit(1);
        case 0:
            <code for child process>
            exit(0);
        default:
            <code for parent process>
            wait(&child_status);
    }
}
```



Process

fork() - The Flow

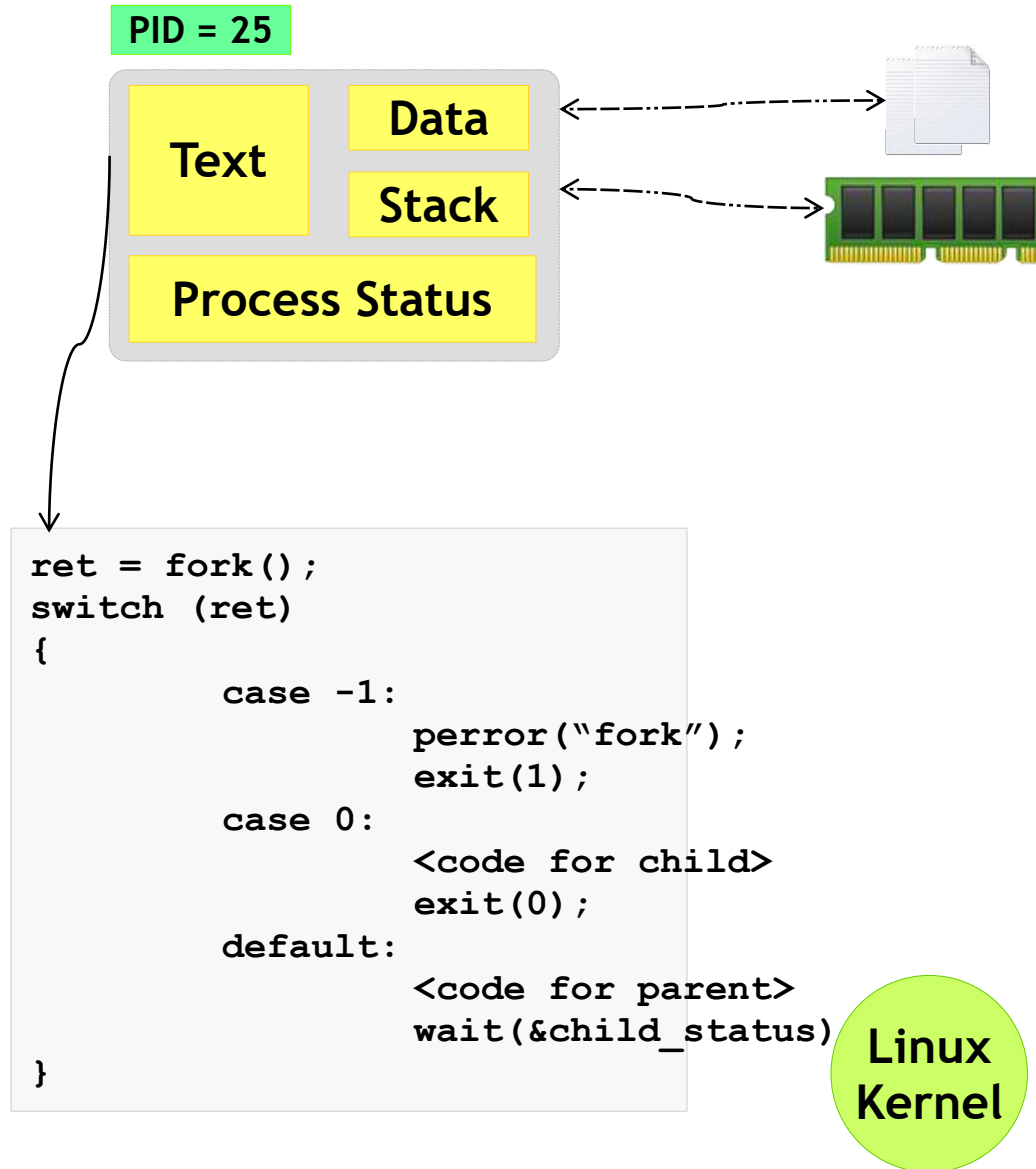
PID = 25



Linux
Kernel

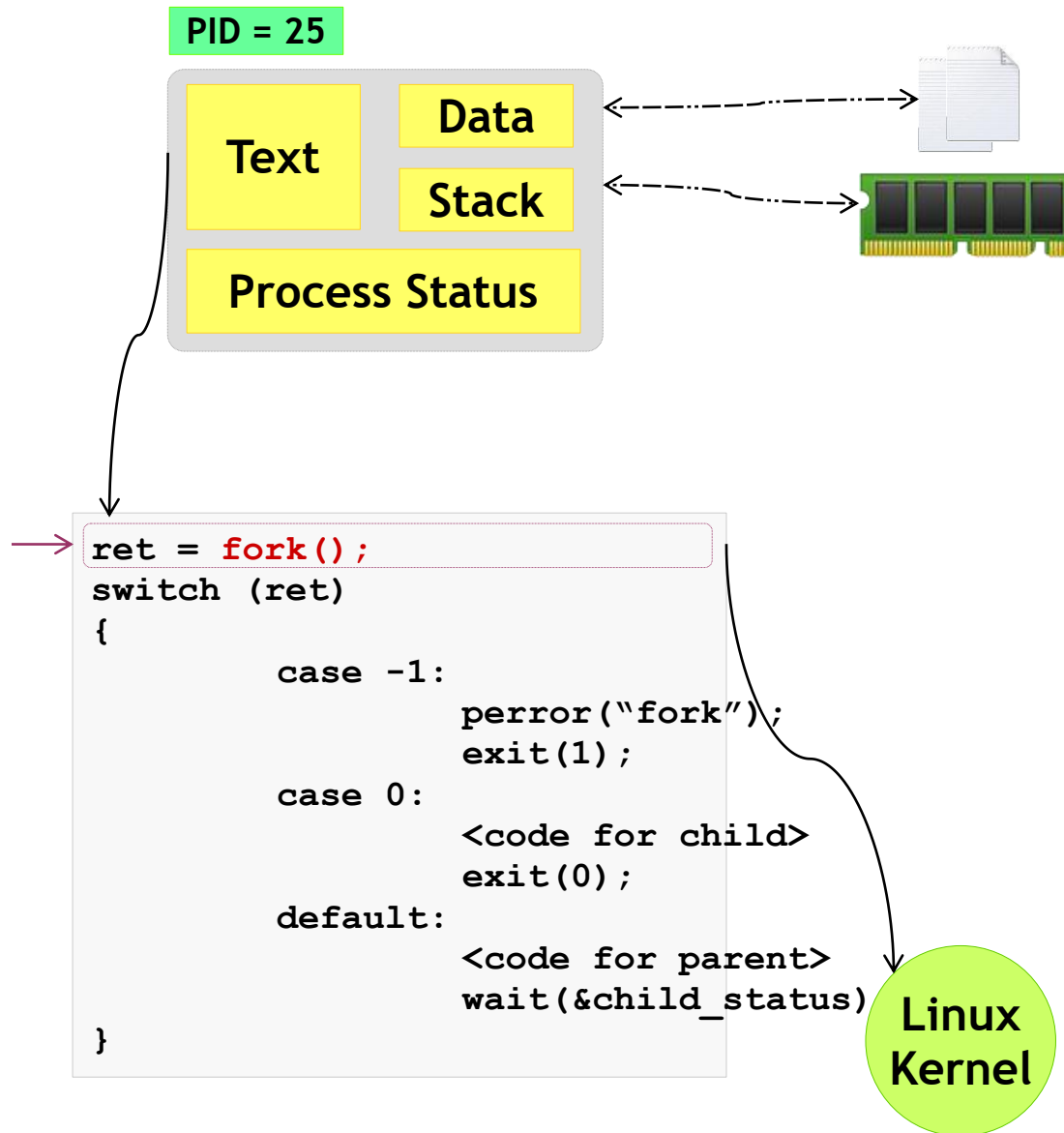
Process

fork() - The Flow



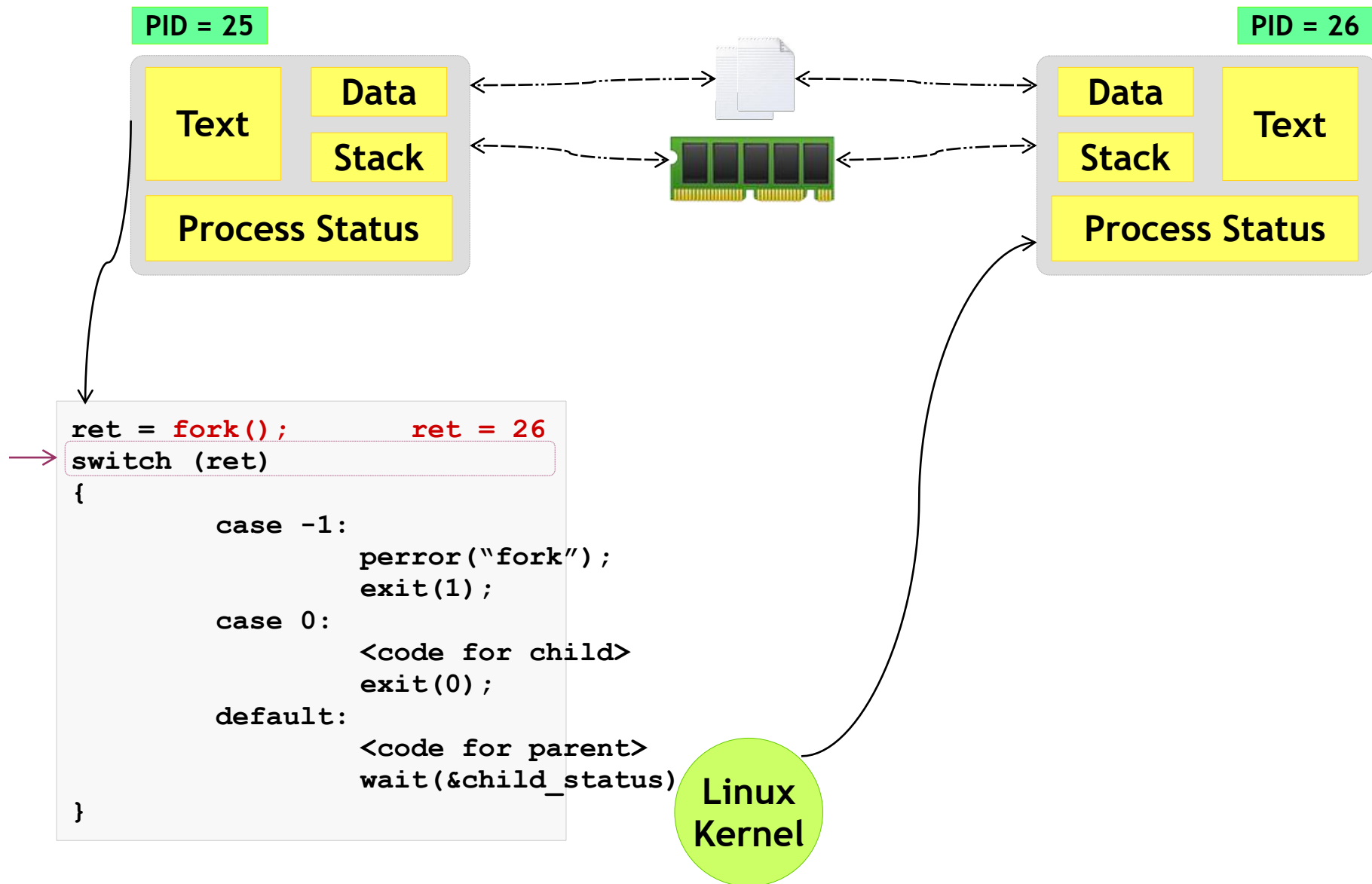
Process

fork() - The Flow



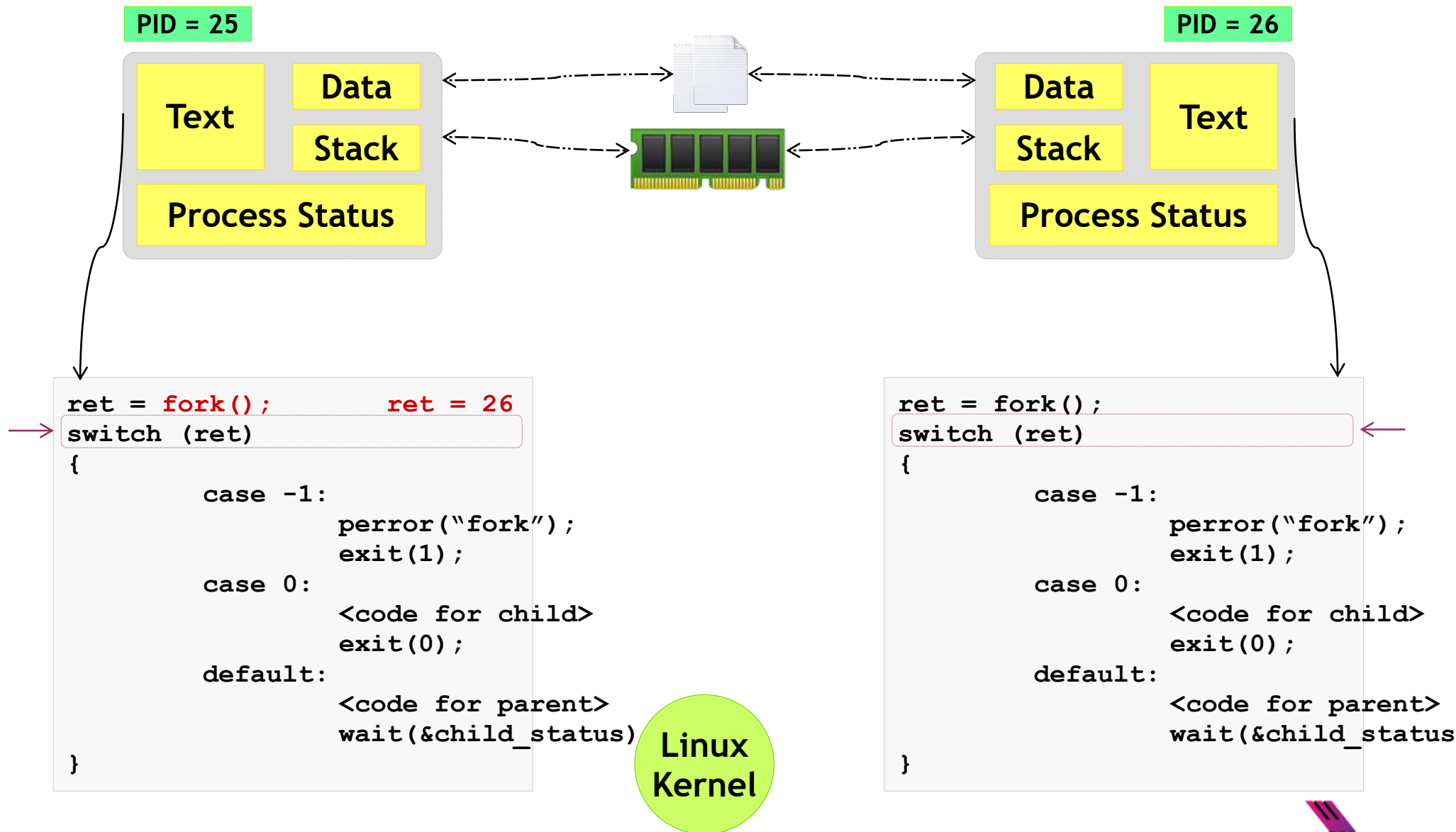
Process

fork() - The Flow



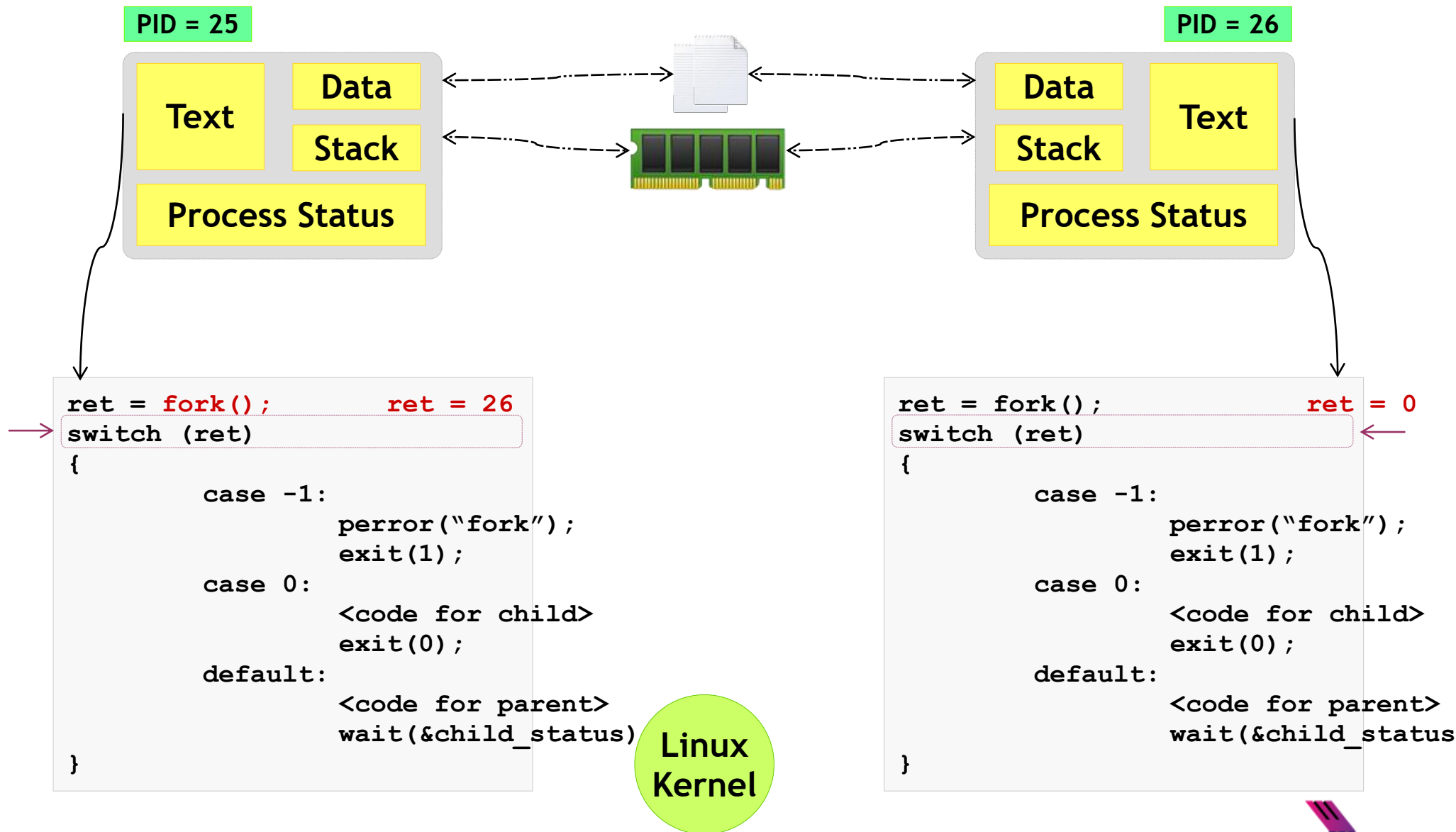
Process

fork() - The Flow



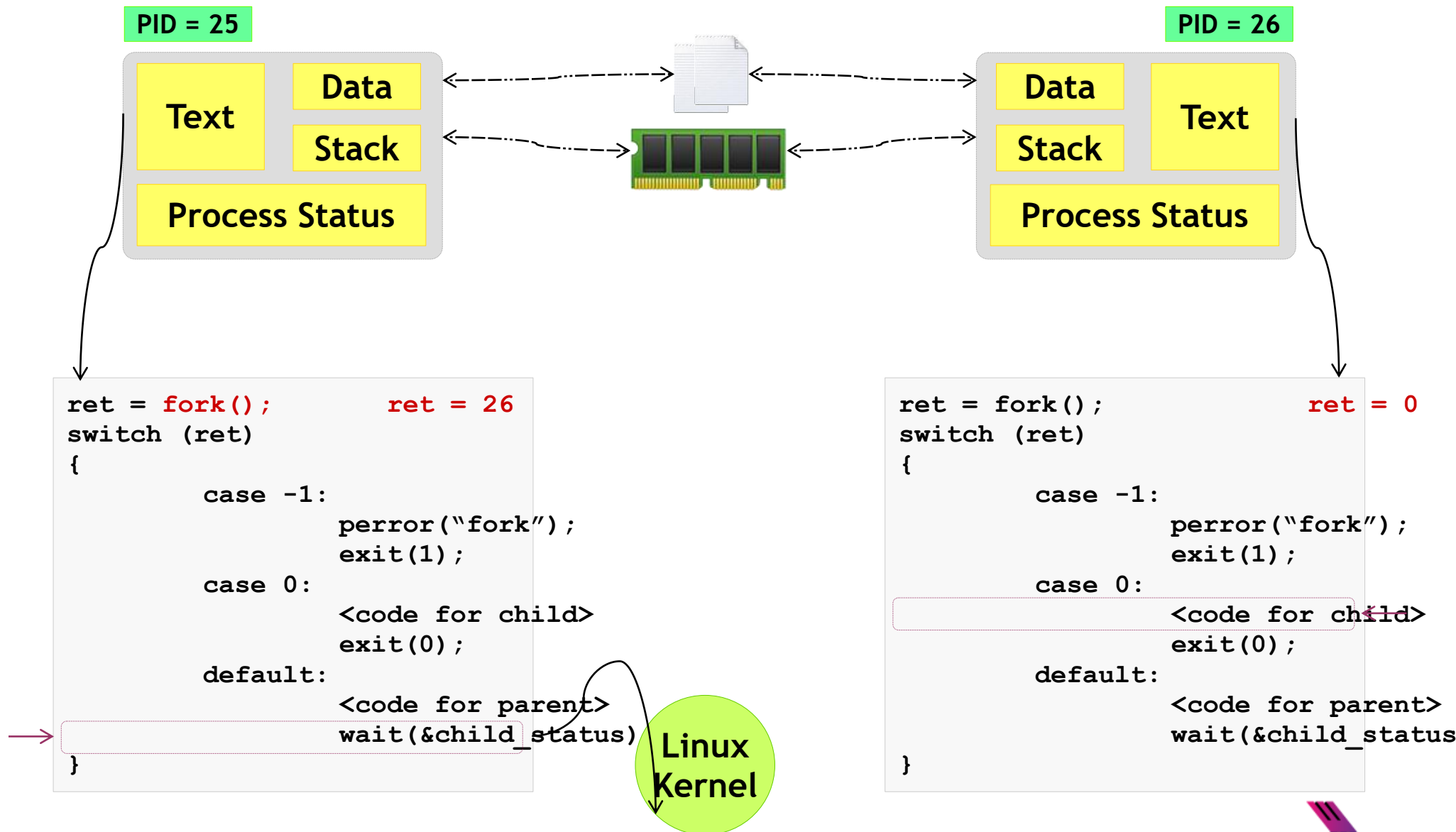
Process

fork() - The Flow



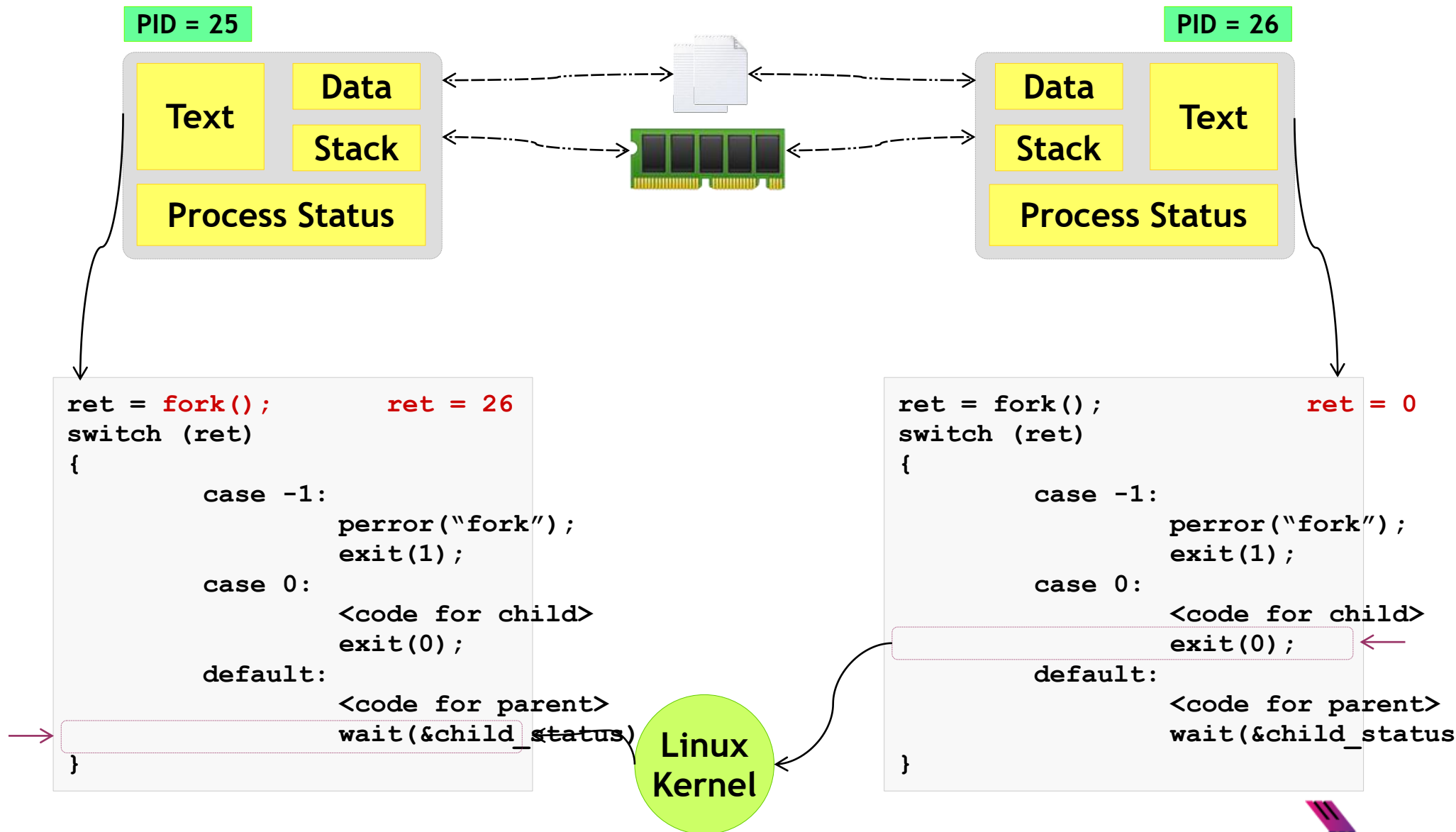
Process

fork() - The Flow



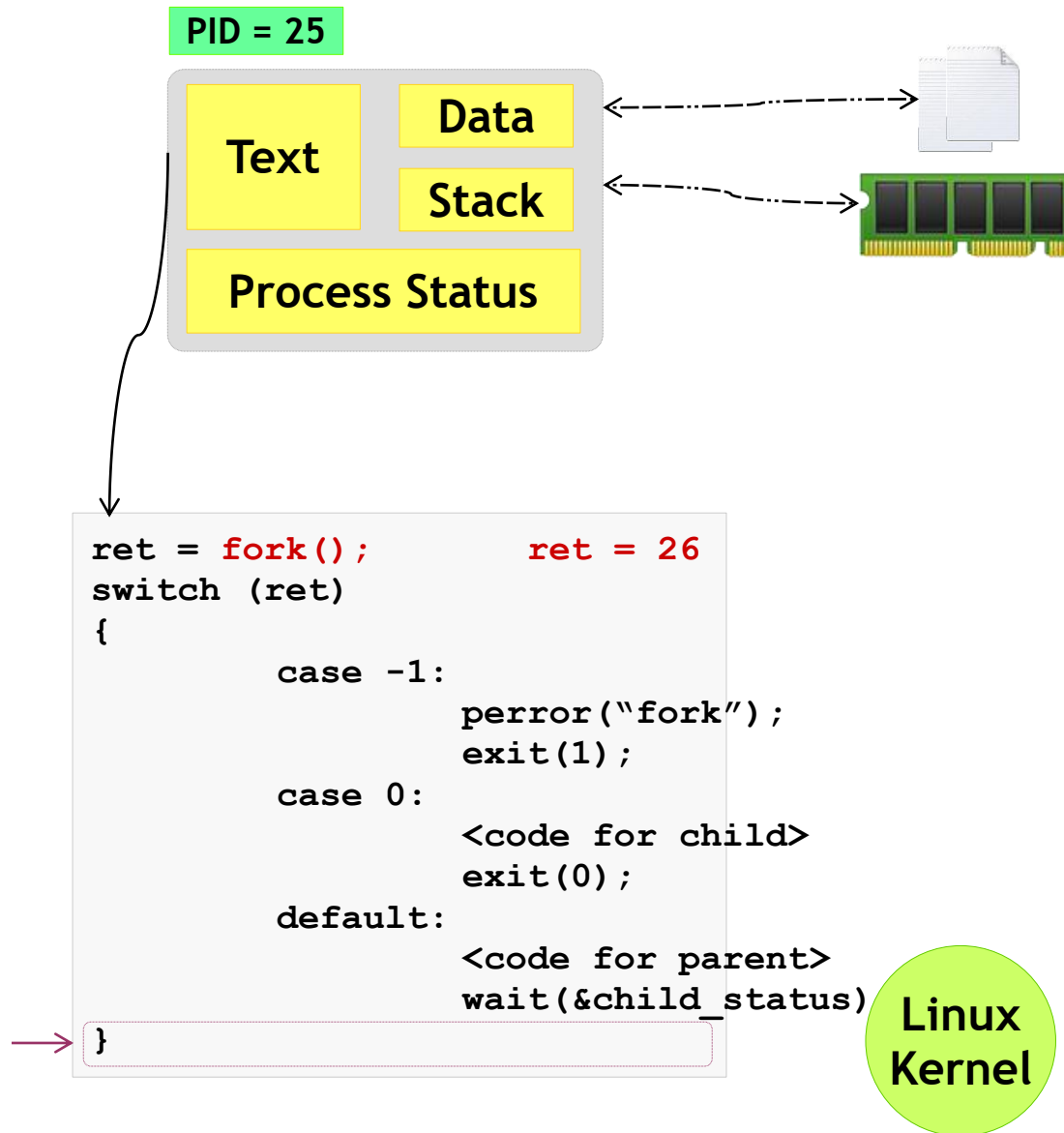
Process

fork() - The Flow



Process

fork() - The Flow



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

fork() - Example



.What would be output of the following program?

```
int main()
{
    fork();
    fork();
    fork();

    printf("Hello World\n");

    return 0;
}
```

Process

fork() - Example



```
→ int main()  
{  
    fork();  
    fork();  
    fork();  
  
    printf("Hello World\n");  
  
    return 0;  
}
```

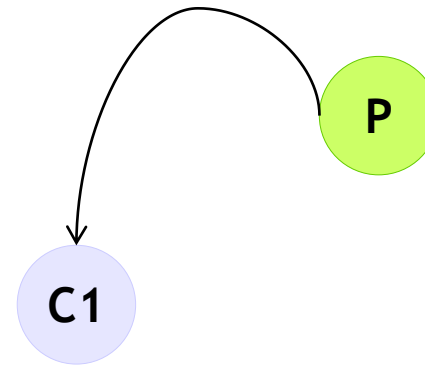

Process

fork() - Example

```
int main()
{
    fork();
    fork();
    fork();

    printf("Hello World\n");

    return 0;
}
```



Team Emertxe



Team Emertxe



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.

Process

Orphan



- An orphan process is a computer process whose parent process has finished or terminated, though it remains running itself.
- Orphaned children are immediately "adopted" by init .
- An orphan is just a process. It will use whatever resources it uses. It is reasonable to say that it is not an "orphan" at all since it has a parent but "adopted".
- Init automatically reaps its children (adopted or otherwise).
- So if you exit without cleaning up your children, then they will not become zombies.

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

Overlay - exec()

Let us consider an example of `execlp` (variant of `exec()` function) shown below

```
/* Program: my_ls.c */
```

```
→ int main()  
{  
    print("Executing my ls :)\n");  
    execlp("/bin/ls", "ls", NULL);  
}
```

PID

Program
Counter

Registers

Stack

Heap

Data

Code

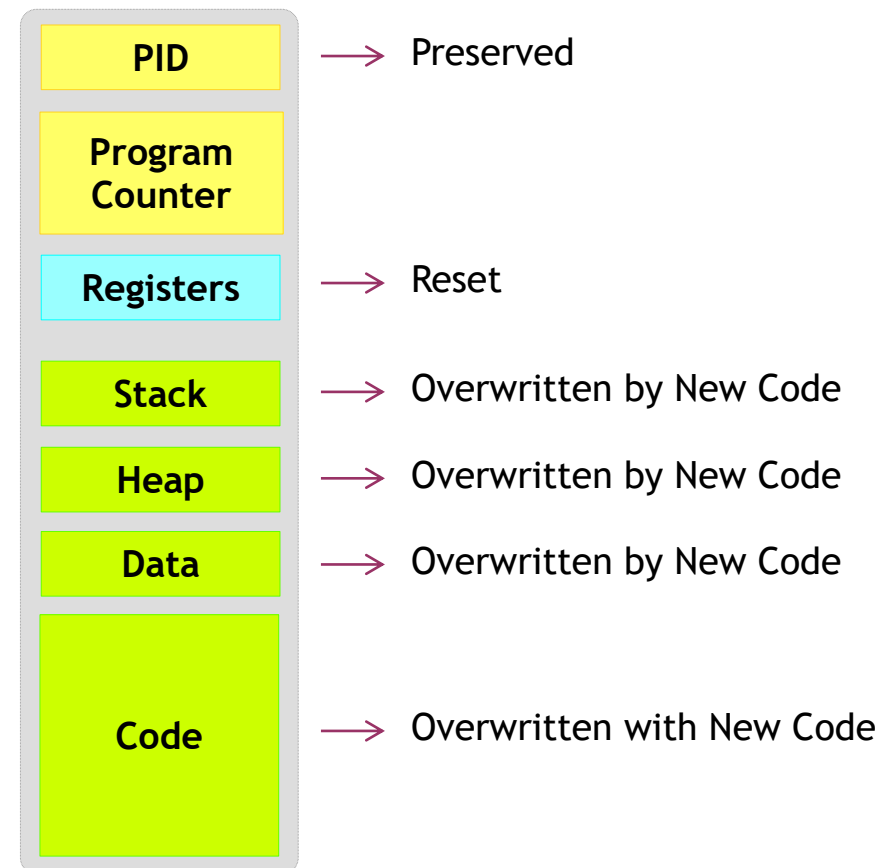
Process

Overlay - exec()



•After executing the exec function, you will note the following changes

```
/* Program: my_ls.c */  
  
int main()  
{  
    print("Executing my ls :)\n");  
    → execlp("/bin/ls", "ls", NULL);  
}
```



Process

exec() - Variants



- .The exec has a family of system calls with variations among them
- .They are differentiated by small changes in their names
- .The exec family looks as follows:

System call	Meaning
<code>execl(const char *path, const char *arg, ...);</code>	Full path of executable, variable number of arguments
<code>execlp(const char *file, const char *arg, ...);</code>	Relative path of executable, variable number of arguments
<code>execv(const char *path, char *const argv[]);</code>	Full path of executable, arguments as pointer of strings
<code>execvp(const char *file, char *const argv[]);</code>	Relative path of executable, arguments as pointer of strings

Process

Blending fork() and exec()



- Practically calling program never returns after exec()
- If we want a calling program to continue execution after exec, then we should first fork() a program and then exec the subprogram in the child process
- This allows the calling program to continue execution as a parent, while child program uses exec() and proceeds to completion
- This way both fork() and exec() can be used together

Process

COW – Copy on Write



- Copy-on-write (called COW) is an optimization strategy
- When multiple separate processes use the same copy of the same information, it is not necessary to re-create it
- Instead, they can all be given pointers to the same resource, thereby effectively using the resources
- However, when a local copy has been modified (i.e. write), the COW has to replicate the copy, as it has no other option
- For example, if `exec()` is called immediately after `fork()`, they never need to be copied; the parent memory can be shared with the child, only when a write is performed it can be re-created



- When a parent forks a child, the two processes 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 Termination



- Process can be terminated in any one of the following ways,
 - Normal program termination, i.e end of the main function.
 - By an explicit return statement in the main function.
 - By the **exit** function or **_exit** system call anywhere in the program.
 - On receipt of a signal which may terminate the process.

Process

Child's exit status



.What does the parent do when the child is executing?

- Wait to gather the child's exit status

- .Example:

- Normal shell behavior when any command is run in the command prompt.

- Continue execution without waiting for the child and pick up the exit status of the child later.

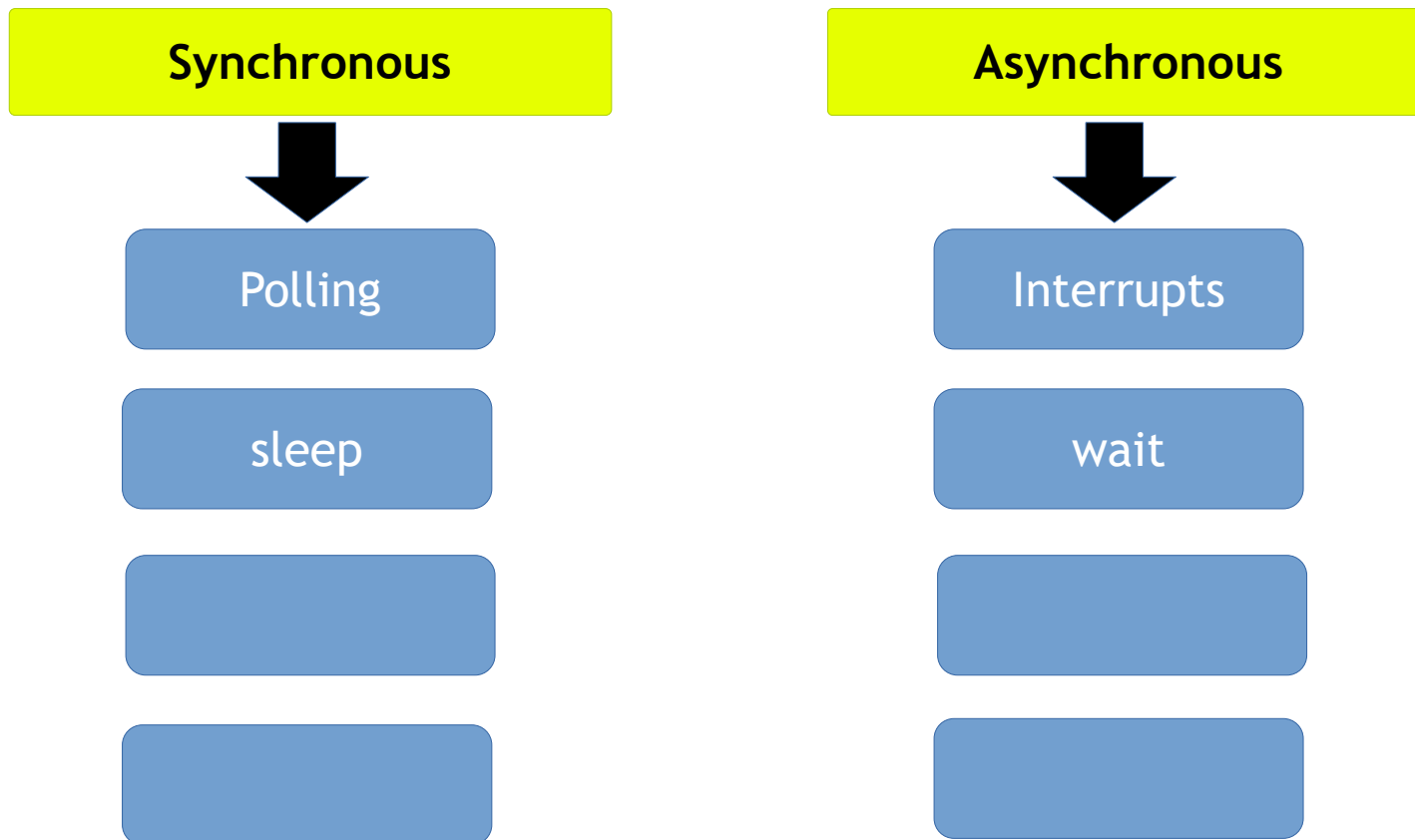
- .Example:

- Shell exhibiting non-waiting, when any command is run in the background.

Synchronous & Asynchronous



.Wait for child to finish





- `fork()` in combination with `wait()` can be used for child monitoring
- Appropriate clean-up (if any) can be done by the parent for ensuring better resource utilization
- Otherwise it will result in a ZOMBIE process
- There are four different system calls in the wait family

System call	Meaning
<code>wait(int *status)</code>	Blocks & waits the calling process until one of its child processes exits. Return status via simple integer argument
<code>waitpid (pid_t pid, int* status, int options)</code>	Similar to <code>wait</code> , but only blocks on a child with specific PID
<code>wait3(int *status, int options, struct rusage *rusage)</code>	Returns resource usage information about the exiting child process.
<code>wait4 (pid_t pid, int *status, int options, struct rusage *rusage)</code>	Similar to <code>wait3</code> , but on a specific child

Process

Resource Structure



```
struct rusage {  
    struct timeval ru_utime;           /* user CPU time used */  
    struct timeval ru_stime;           /* system CPU time  
used */  
    long   ru_maxrss;                  /* maximum resident set size  
*/  
    long   ru_ixrss;                   /* integral shared memory  
size */  
    long   ru_idrss;                   /* integral unshared data size  
*/  
    long   ru_isrss;                   /* integral unshared stack  
size */  
};
```



Team Emertxe



Thank You