

# Process Management

A process is simply an abstraction of a running program.

- Processes include a set of resources such as open files and pending signals, internal kernel data, processor state, an address space, one or more threads of execution, and a data section containing global variables.

Whereas,

- Threads are the units of execution within a program.
- In Linux Operating system, the scheduling of thread or processes take place with a common concept of tasks--

A Linux Scheduler will schedule tasks and these tasks could be

1. *a single-threaded process (e.g. created by fork function without any thread library)*
2. *any thread inside a multi-threaded process (including its main thread)*
3. *kernel tasks, which are started internally in the kernel and stay in kernel space (e.g kworker, kswapd . etc...)*

- Each thread within a process has a 1. unique program counter, 2. process stack, 3. and set of processor registers.
- Threads are light weight. They don't have their own memory spaces and other resources unlike processes. All processes start with a single thread. So they behave like lightweight processes but are always tied to a parent "thick" process. So, creating a new process is a slightly heavy task and involves allocating all these resources while creating a thread does not. Killing a process also involves releasing all these resources while a thread does not. However, killing a thread's parent process releases all resources of the thread.
- A process is suspended by itself and resumed by itself. Same with a thread but if a thread's parent process is suspended then the threads are all suspended.
- In Linux, a process is created by means of the `fork()` system call, which creates a new process by duplicating an existing one.
- The new process is an exact copy of the old process. (Now, a question might come immediately to our mind- who creates the first process?? The first process is the init process which is literally created from scratch during booting).
- The process that calls `fork()` is the **parent**, whereas the new process is the **child**. The parent resumes execution and the child starts execution at the same place, where the call returns. The `fork()` system call returns from the kernel twice: once in the parent process and again in the newborn child(see the below diagram).

## fork()

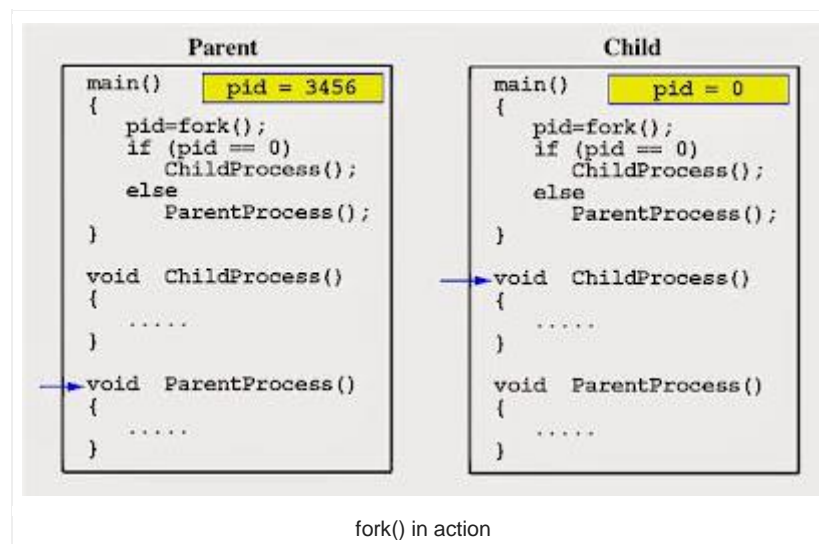
- The `fork()` function is used to create a new process by duplicating the existing process from which it is called.
- The existing process from which this function is called becomes the parent process and the newly created process becomes the child process.
- Inside the newly created address space `exec*()` family of function calls is used to create a new address space and load a new program into it. In modern Linux kernels, `fork()` is actually implemented via the `clone()` system call,

- Finally, a program exits via the `exit()` system call. This function terminates the process and frees all its resources.
- Fork is the only mechanism to create new process, any other mechanism will inherently use fork inside.
- A parent process can inquire about the status of a terminated child via the `wait4()` system call, which enables a process to wait for the termination of a specific process.
- The child is a duplicate copy of the parent but there are some exceptions to it.

1. ***The child has a unique PID like any other process running in the operating system.***
2. ***The child has a parent process ID which is same as the PID of the process that created it.***
3. ***Resource utilization and CPU time counters are reset to zero in child process.***
4. ***Set of pending signals in child is empty.***
5. ***Child does not inherit any timers from its parent***
6. ***The child does not inherit its parent's memory locks (`mlock(2)`, `mlockall(2)`)***
7. ***The child does not inherit outstanding asynchronous I/O operations from its parent.***

## What is the return value of fork()?

- If the `fork()` function is successful then it returns twice.
- Once it returns in the child process with return value zero and then it returns in the parent process with child's PID as return value.
- This behavior is because of the fact that once the fork is called, child process is created and since the child process shares the text segment with parent process and continues execution from the next statement in the same text segment so fork returns twice (once in parent and once in child).



```

#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    pid_t pid;
    char *message;
    int n;

    printf("fork program is starting\n");
    pid = fork();
    switch(pid)
    {
        case -1:
            perror("fork failed");
            exit(1);
        case 0:
            message = "This is the child";
            n = 5;
            break;
        default:
            message = "This is the parent";
            n = 3;
            break;
    }

    for(; n > 0; n--) {
        puts(message);
        sleep(1);
    }
    exit(0);
}

```

The parent gets over before the child so we can see the shell prompt appears mixed with the output of child process.

Now, lets try to finish the child process first.

```

#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    pid_t pid;
    char *message;
    int n;

    printf("fork program starting\n");
    pid = fork();

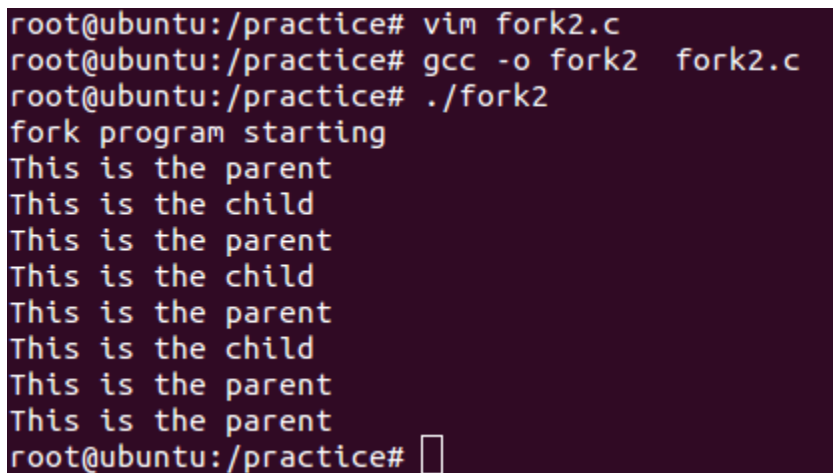
```

```

switch(pid)
{
case -1:
    perror("fork failed");
    exit(1);
case 0:
    message = "This is the child";
    n = 3;
    break;
default:
    message = "This is the parent";
    n = 5;
    break;
}

for(; n > 0; n--) {
    puts(message);
    sleep(1);
}
exit(0);
}

```



```

root@ubuntu:/practice# vim fork2.c
root@ubuntu:/practice# gcc -o fork2 fork2.c
root@ubuntu:/practice# ./fork2
fork program starting
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the parent
root@ubuntu:/practice#

```

When the child process is over, its

association with the parent is not finished. The association remains until the parent calls the wait system call or the parent survives normally. Till this association remains after the child process is over the child process is called as a zombie process( the last ritual hasn't been performed till now).

NB: when the parent terminates but the child is still running then it is running as an orphan process.

Waiting for the child--->

A parent can wait for its child by calling wait system call. This call returns the child's PID as return value.

```

#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

```

```

int main()
{

```

```

pid_t pid;
char *message;
int n;
int exit_code;

printf("fork program starting\n");
pid = fork();
switch(pid)
{
case -1:
    exit(1);
case 0:
    message = "This is the child";
    n = 5;
    exit_code = 37;
    break;
default:
    message = "This is the parent";
    n = 3;
    exit_code = 0;
    break;
}

for(; n > 0; n--) {
    puts(message);
    sleep(1);
}

/* This section of the program waits for the child process to finish. */

if(pid) {
    int stat_val;
    pid_t child_pid;

    child_pid = wait(&stat_val);

    printf("Child has finished: PID = %d\n", child_pid);
    if(WIFEXITED(stat_val))
        printf("Child exited with code %d\n", WEXITSTATUS(stat_val));
    else
        printf("Child terminated abnormally\n");
}
exit (exit_code);
}

```

## **fork() vs vfork() vs exec() vs system() vs clone()**

Let us first see the standard definition of these [system calls](#).

**Fork :** The fork call is used to duplicate the current process, the new process identical in almost every way except that it has its own PID. The return value of the function fork distinguishes the two processes, zero is returned in the child and PID of child in parent process.

**Exec :** The exec call is a way to basically replace the entire current process with a new program. It loads the program into the current process space and runs it from the entry point. As a new process is not created, the process identifier (PID) does not change, but the machine code, data, heap, and stack of the process are replaced by those of the new program. exec() replaces the current process with a the executable pointed by the function. Control never returns to the original program unless there is an exec() error. exec system call can be executed as execl, execlp, execlx, execlv, execlvp, execlvpe

**Vfork:** The basic difference between vfork and fork is that when a new process is created with vfork(), the parent process is temporarily suspended, and the child process might borrow the parent's address space. This strange state of affairs continues until the child process either exits, or calls execve(), at which point the parent process continues.

This means that the child process of a vfork() must be careful to avoid unexpectedly modifying variables of the parent process. In particular, the child process must not return from the function containing the vfork() call, and it must not call exit() (if it needs to exit, it should use \_exit()); actually, this is also true for the child of a normal fork()).

The intent of vfork was to eliminate the overhead of copying the whole process image if you only want to do an `exec*` in the child. Because `exec*` replaces the whole image of the child process, there is no point in copying the image of the parent.

```
if ((pid = vfork()) == 0) {
    execl(..., NULL); /* after a successful execl the parent should be resumed */
    _exit(127); /* terminate the child in case execl fails */
}
```

For other kinds of uses, vfork is dangerous and unpredictable.

With most current kernels, however, including Linux, the primary benefit of vfork has disappeared because of the way fork is implemented. Rather than copying the whole image when fork is executed, copy-on-write techniques are used.

**Clone :** Clone, as fork, creates a new process. Unlike fork, these calls allow the child process to share parts of its execution context with the calling process, such as the memory space, the table of file descriptors, and the table of signal handlers.

When the child process is created with clone, it executes the function application fn(arg). (This differs from fork, where execution continues in the child from the point of the original fork call.) The fn argument is a pointer to a function that is called by the child process at the beginning of its execution. The arg argument is passed to the fn function.

When the fn(arg) function application returns, the child process terminates. The integer returned by fn is the exit code for the child process. The child process may also terminate explicitly by calling exit(2) or after receiving a fatal signal.

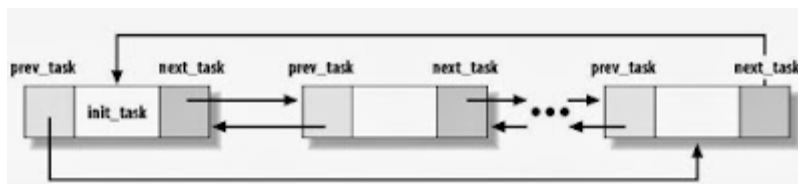
**System :** The **system()** library function uses fork(2) to create a child process that executes the shell command specified in *command* using execl(3) as follows: execl("/bin/sh", "sh", "-c", *command*, (char \*) 0); **system()** returns after the command has been completed. During execution of the *command*, **SIGCHLD** will be blocked, and **SIGINT** and **SIGQUIT** will be ignored, in the process that calls **system()** (these signals will be handled according to their defaults inside the child process that

executes *command*). If *command* is NULL, then **system()** returns a status indicating whether a shell is available on the system

- In situations where performance is critical and/or memory limited, `vfork` + `exec*` can therefore be a good alternative to `fork` + `exec*`. The problem is that it is less safe and the man page says `vfork` is likely to become deprecated in the future.
- Because memory page tables are not duplicated, `vfork` is much faster than `fork` and `vfork`'s execution time is not affected by the amount of memory the parent process uses
- `system()` will invoke your system's default command shell, which will execute the command string passed as an argument, that itself may or may not create further processes, that would depend on the command and the system. Either way, at least a command shell process will be created.
- With `system()` you can invoke any command, whereas with `exec()`, you can only invoke an executable file. Shell scripts and batch files must be executed by the command shell.

## Process Descriptor and the Task Structure

- The kernel stores the list of processes in a circular doubly linked list called the task list.
- **Process descriptor is nothing but each element of this task list** of the type `struct task_struct`, which is defined in `<linux/sched.h>`. The process descriptor contains all the information about a specific process.
- Some texts on operating system design call this list the task array. Because the Linux implementation is a linked list and not a static array, it is called the task list.
- The `task_struct` is a relatively large data structure, at around 1.7 kilobytes on a 32-bit machine.
- This size, however, is quite small considering that the structure contains all the information that the kernel has and needs about a process.
- The process descriptor contains the data that describes the executing program open files, the process's address space, pending signals, the process's state, and much more.
- This linked list is stored in kernel space.
- There is one more structure, `thread_info` which holds more architecture-specific data than the `task_struct`.



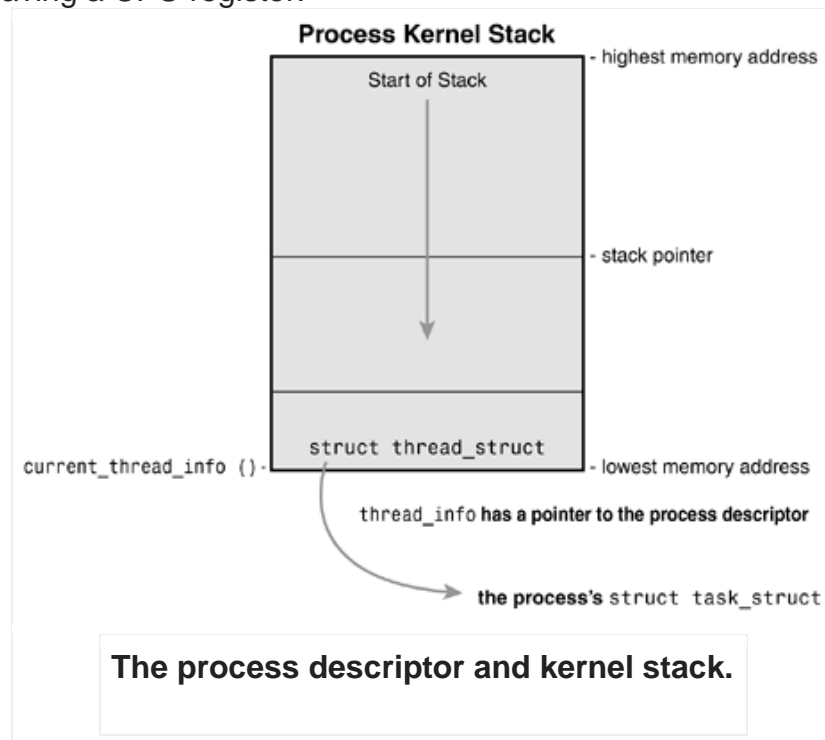
## Allocating the Process Descriptor

- Threads in Linux are treated as processes that just happen to share some resources.

1. `thread_info` is architecture dependent. `task_struct` is generic.
2. `thread_info` consumes the space of the kernel stack for that process, so it should be kept small.

- Each thread has its own `thread_info`. There are two basic reasons why there are two such structures.

"`thread_info`" is placed at the bottom of the stack as a micro-optimization that makes it possible to compute its address from the current stack pointer by rounding down by the stack size saving a CPU register.



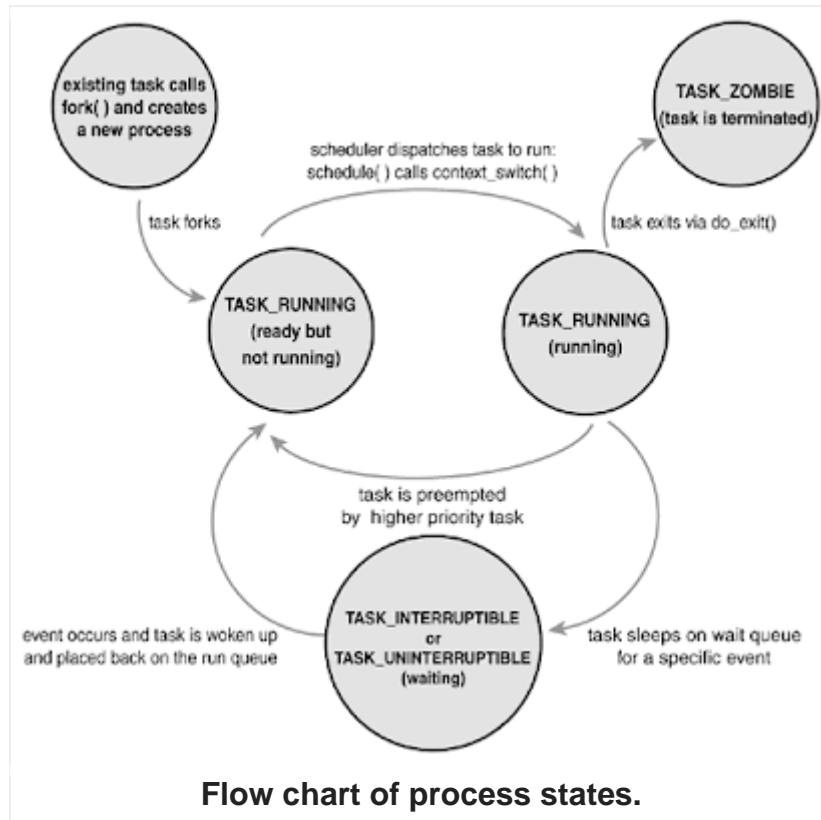


- The `thread_info` structure is defined on x86 as well as in ARM in `<asm/thread_info.h>` as

```
struct thread_info {
    struct task_struct *task;
    struct exec_domain *exec_domain;
    unsigned long flags;
    unsigned long status;
    __u32 cpu;
    __s32 preempt_count;
    mm_segment_t addr_limit;
    struct restart_block restart_block;
    unsigned long previous_esp;
    __u8 supervisor_stack[0];
};
```

- Each task's `thread_info` structure is allocated at the end of its stack. The `task` element of the structure is a pointer to the task's actual `task_struct`.

## Process State



- The `state` field of the process descriptor describes the current condition of the process.
- Each process on the system is in exactly one of five different states. This value is represented by one of five flags:

1. `TASK_INTERRUPTIBLE` - The process is sleeping (that is, it is blocked), waiting for some condition

to exist. When this condition exists, the kernel sets the process's state to `TASK_RUNNING`.

The process also awakes prematurely and becomes runnable if it receives a signal.

2. `TASK_UNINTERRUPTIBLE` - This state is identical to `TASK_INTERRUPTIBLE` except that it does not

wake up and become runnable if it receives a signal. This is used in situations where the process must wait without interruption or when the event is expected to occur quite quickly.

Because the task does not respond to signals in this state, `TASK_UNINTERRUPTIBLE` is less often

used than `TASK_INTERRUPTIBLE`

3. `TASK_ZOMBIE` - The task has terminated, but its parent has not yet issued a `wait4()` system call.

The task's process descriptor must remain in case the parent wants to access it. If the parent

calls `wait4()`, the process descriptor is deallocated.

4. `TASK_STOPPED` - Process execution has stopped; the task is not running nor is it eligible to run.

This occurs if the task receives the `SIGSTOP`, `SIGTSTP`, `SIGTTIN`, or `SIGTTOU` signal or if it receives any signal while it is being debugged.

5. `TASK_RUNNING` - The process is runnable; it is either currently running or on a runqueue waiting to run. This is the only possible state for a process executing in user-space; it can also apply to a process in kernel-space that is actively running.

## Manipulating the Current Process State

- Kernel code often needs to change a process's state. The preferred mechanism is

using `set_task_state(task, state);` /\* set task 'task' to state 'state' \*/

- This function sets the given task to the given state. If applicable, it also provides a

memory barrier to force ordering on other processors (this is only needed on SMP systems). Otherwise, it is equivalent to `task->state = state;`

- The method `set_current_state(state)` is synonymous to `set_task_state(current, state)`.

## Process Context vs Interrupt Context

- One of the most important parts of a process is the executing program code.

- The process and interrupt context is with reference to the kernel execution, when kernel is working on behalf of a process or it is running some kernel threads it is said to be executing in process context whereas when the kernel is handling some interrupt handler then it is said to be working in interrupt context.

- **Normal program execution occurs in user-space.** When a program executes a system call or

triggers an exception, it enters **kernel-space**.

- When system call is executed, the **kernel is said to be "executing on behalf of the**

**process"** and is in process context.

- When **in process context, the `current` macro is valid.**

- Upon exiting the kernel when the system call execution is done, the process resumes execution in user-space, unless a higher-priority process has become runnable in the interim, in which case the scheduler is invoked to select the higher priority process.

- There is no process tied to interrupt handlers and consequently no process context.

- **System calls and exception handlers are well-defined interfaces into the kernel.**

- A process can begin executing in kernel-space only through one of these interfaces all access to

the kernel is through these interfaces.

- Sleeping is not allowed in an interrupt context as inherently it is not a process and

hence there is no one backing to wake it up.

## The Process Family Tree

- A distinct hierarchy exists between processes in Unix systems, and Linux is no exception.
- All the processes running in linux are descendents of the `init` process, whose PID is one.
- The scheduler which gets to schedule all the processes has a PID of 0.
- The kernel starts `init` in the last step of the boot process.
- The `init` process, in turn, reads the system `init` scripts and executes more programs, which can be thought like spawning and eventually completing the boot process.
- Every process on the system has exactly one parent.
- Likewise, every process has zero or more children.
- Processes that are all direct children of the same parent are called siblings.
- The relationship between processes is stored in the process descriptor.
- Each `task_struct` has a pointer to the parent's `task_struct`, named `parent`, and a list of children, named `children`.
- Using simple data structure logic, given the current process, it is possible to obtain the process descriptor of its parent with the following code:

```
struct task_struct *my_parent = current->parent;
```

- Code snippet to iterate over a process's children

```
struct task_struct *task;  
struct list_head *list;  
  
list_for_each(list, &current->children) {  
    task = list_entry(list, struct task_struct, sibling);  
    /* task now points to one of current's children */  
}
```

## Kernel Threads

- A kernel thread is a kernel task running only in kernel mode; it usually has not been

created by `fork()` or `clone()` system calls. An example is `kworker` or `kswapd`.

There are 4 functions useful in context of kernel threads-

1. `start_kthread`: creates a new kernel thread. Can be called from any process context but not from interrupt. The function blocks until the thread started.
2. `stop_kthread`: stop the thread. Can be called from any process context but the thread to be terminated. Cannot be called from interrupt context. The function blocks until the thread terminated.
3. `init_kthread`: sets the environment of the new threads. Is to be called out of the created thread.
4. `exit_kthread`: needs to be called by the thread to be terminated on exit.

- It is often useful for the kernel to perform some operations in the background, and the

kernel achieves this via kernel threads standard processes that exist solely in kernel-space.

- The significant difference between kernel threads and normal processes is that kernel threads

do not have an address space (in fact, their `mm` pointer is `NULL`) as it is the address space which contains the kernel.

- They operate only in kernel-space and do not context switch into user-space.
- Kernel threads are, however, schedulable and preemptable as normal processes.
- Linux delegates several tasks to kernel threads, most notably the `pdflush` task and the `ksoftirqd`

task. These threads are created on system boot by other kernel threads. Indeed, a kernel thread can be created only by another kernel thread. The interface for spawning a new kernel thread from an existing one is

```
int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
```

- The new task is created via the usual `clone()` system call with the specified `flags` argument.

On return, the parent kernel thread exits with a pointer to the child's `task_struct`. The child executes the function specified by `fn` with the given argument `arg`. A special clone flag, `CLONE_KERNEL`, specifies the usual flags for kernel threads: `CLONE_FS`, `CLONE_FILES`, and `CLONE_SIGHAND`. Most kernel threads pass this for their `flags` parameter

- Typically, a kernel thread continues executing its initial function forever (or at least until the

system reboots, but with Linux you never know). The initial function usually implements a loop in which the kernel thread wakes up as needed, performs its duties, and then returns to sleep.