Linux Kernel Process Management

By Robert Love

Date: Apr 15, 2005

Sample Chapter is provided courtesy of Sams.

Return to the article

This chapter looks at the famed operating system abstraction of the process. Topics covered include the generalities of the process, why it is important, and the relationship between processes and threads. Specifically, this chapter covers how Linux stores and represents processes.

The *process* is one of the fundamental abstractions in Unix operating systems[1]. A process is a program (object code stored on some media) in execution. Processes are, however, more than just the executing program code (often called the *text section* in Unix). They also 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. Processes, in effect, are the living result of running program code.

Threads of execution, often shortened to *threads*, are the objects of activity within the process. Each thread includes a unique program counter, process stack, and set of processor registers. The kernel schedules individual threads, not processes. In traditional Unix systems, each process consists of one thread. In modern systems, however, multithreaded programs—those that consist of more than one thread—are common. As you will see later, Linux has a unique implementation of threads: It does not differentiate between threads and processes. To Linux, a thread is just a special kind of process.

On modern operating systems, processes provide two virtualizations: a virtualized processor and virtual memory. The virtual processor gives the process the illusion that it alone monopolizes the system, despite possibly sharing the processor among dozens of other processes. Chapter 4, "Process Scheduling," discusses this virtualization. Virtual memory lets the process allocate and manage memory as if it alone owned all the memory in the system. Virtual memory is covered in Chapter 11, "Memory Management." Interestingly, note that threads *share* the virtual memory abstraction while each receives its own virtualized processor.

A program itself is not a process; a process is an *active* program and related resources. Indeed, two or more processes can exist that are executing the *same* program. In fact, two or more processes can exist that share various resources, such as open files or an address space.

A process begins its life when, not surprisingly, it is created. In Linux, this occurs by means of the `fork()` system call, which creates a new process by duplicating an existing one. 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.

Often, immediately after a fork it is desirable to execute a new, different, program. The `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, which is discussed in a following section.

Finally, a program exits via the `exit()` system call. This function terminates the process and frees all its resources. A parent process can inquire about the status of a terminated child via the `wait4()`[2] system call, which enables a process to wait for the termination of a specific process. When a process exits, it is placed into a special zombie state that is used to represent terminated processes until the parent calls `wait()` or `waitpid()`.

Another name for a process is a *task*. The Linux kernel internally refers to processes as tasks. In this book, I will use the terms interchangeably, although when I say *task* I am generally referring to a process from the kernel's point of view.

Process Descriptor and the Task Structure

The kernel stores the list of processes in a circular doubly linked list called the *task list*[3]. Each element in the task list is a *process descriptor* of the type `struct task_struct`, which is defined in `<linux/sched.h>`. The process descriptor contains all the information about a specific process.

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 (see Figure 3.1).

Figure 3.1 The process descriptor and task list.

Allocating the Process Descriptor

The `task_struct` structure is allocated via the *slab allocator* to provide object reuse and cache coloring (see Chapter 11, "Memory Management"). Prior to the 2.6 kernel series, `struct task_struct` was stored at the end of the kernel stack of each process. This allowed architectures with few registers, such as x86, to calculate the location of the process descriptor via the *stack pointer* without using an extra register to store the location. With the process descriptor now dynamically created via the slab allocator, a new structure, `struct thread_info`, was created that again lives at the bottom of the stack (for stacks that grow down) and at the top of the stack (for stacks that grow up)[4]. See Figure 3.2.

The new structure also makes it rather easy to calculate offsets of its values for use in assembly code.

The `thread_info` structure is defined on x86 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`.

Figure 3.2 The process descriptor and kernel stack.

Storing the Process Descriptor

The system identifies processes by a unique *process identification* value or *PID*. The PID is a numerical value that is represented by the opaque type[5] `pid_t`, which is typically an `int`. Because of backward compatibility with earlier Unix and Linux versions, however, the default maximum value is only 32,768 (that of a `short int`), although the value can optionally be increased to the full range afforded the type. The kernel stores this value as `pid` inside each process descriptor.

This maximum value is important because it is essentially the maximum number of processes that may exist concurrently on the system. Although 32,768 might be sufficient for a desktop system, large servers may require many more processes. The lower the value, the sooner the values will wrap around, destroying the useful notion that higher values indicate later run processes than lower values. If the system is willing to break compatibility with old applications, the administrator may increase the maximum value via `/proc/sys/kernel/pid_max`.

Inside the kernel, tasks are typically referenced directly by a pointer to their `task_struct` structure. In fact, most kernel code that deals with processes works directly with `struct task_struct`. Consequently, it is very useful to be able to quickly look up the process descriptor of the currently executing task, which is done via the `current` macro. This macro must be separately implemented by each architecture. Some architectures save a pointer to the `task_struct` structure of the currently running process in a register, allowing for efficient access. Other architectures, such as x86 (which has few registers to waste), make use of the fact that `struct thread_info` is stored on the kernel stack to calculate the location of `thread_info` and subsequently the `task_struct`.

On x86, `current` is calculated by masking out the 13 least significant bits of the stack pointer to obtain the `thread_info` structure. This is done by the `current_thread_info()` function. The assembly is shown here:

```
movl $-8192, %eax
andl %esp, %eax
```

This assumes that the stack size is 8KB. When 4KB stacks are enabled, 4096 is used in lieu of 8192.

Finally, `current` dereferences the `task` member of `thread_info` to return the `task_struct`:

```
current_thread_info()->task;
```

Contrast this approach with that taken by PowerPC (IBM's modern RISC-based microprocessor), which stores the current `task_struct` in a register. Thus, `current` on PPC merely returns the value stored in the register `r2`. PPC can take this approach because, unlike x86, it has plenty of registers. Because accessing the process descriptor is a common and important job, the PPC kernel developers deem using a register worthy for the task.

Process State

The `state` field of the process descriptor describes the current condition of the process (see Figure 3.3). Each process on the system is in exactly one of five different states. This value is represented by one of five flags:

- **TASK_RUNNING**—The process is runnable; it is either currently running or on a runqueue waiting to run (runqueues are discussed in Chapter 4, "Scheduling"). 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.
- **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.
- **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`[6].
- **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.
- **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.

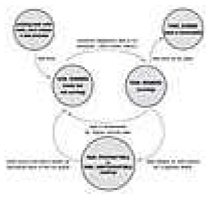

Figure 3.3 Flow chart of process states.

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

One of the most important parts of a process is the executing program code. This code is read in from an *executable file* and executed within the program's address space. Normal program execution occurs in *user-space*. When a program executes a system call (see Chapter 5, "System Calls") or triggers an exception, it enters *kernel-space*. At this point, 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[7]. Upon exiting the kernel, 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.

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.

The Process Family Tree

A distinct hierarchy exists between processes in Unix systems, and Linux is no exception. All processes are descendents of the `init` process, whose PID is one. The kernel starts `init` in the last step of the boot process. The `init` process, in turn, reads the system *initscripts* and executes more programs, 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`. Consequently, 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;
```

Similarly, it is possible to iterate over a process's children with

```
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 */
}
```

The `init` task's process descriptor is statically allocated as `init_task`. A good example of the relationship between all processes is the fact that this code will always succeed:

```
struct task_struct *task;

for (task = current; task != &init_task; task = task->parent)
    ;
/* task now points to init */
```

In fact, you can follow the process hierarchy from any one process in the system to *any* other. Oftentimes, however, it is desirable simply to iterate over *all* processes in the system. This is easy because the task list is a circular doubly linked list. To obtain the next task in the list, given any valid task, use:

```
list_entry(task->tasks.next, struct task_struct, tasks)
```

Obtaining the previous works the same way:

```
list_entry(task->tasks.prev, struct task_struct, tasks)
```

These two routines are provided by the macros `next_task(task)` and `prev_task(task)`, respectively. Finally, the macro `for_each_process(task)` is provided, which iterates over the entire task list. On each iteration, `task` points to the next task in the list:

```
struct task_struct *task;

for_each_process(task) {
    /* this pointlessly prints the name and PID of each task */
    printk("%s[%d]\n", task->comm, task->pid);
}
```

Note: It can be expensive to iterate over every task in a system with many processes; code should have good reason (and no alternative) before doing so.

Process Creation

Process creation in Unix is unique. Most operating systems implement a *spawn* mechanism to create a new process in a new address space, read in an executable, and begin executing it. Unix takes the unusual approach of separating these steps into two distinct functions: `fork()` and `exec()`[8]. The first, `fork()`, creates a child process that is a copy of the current task. It differs from the parent only in its PID (which is unique), its PPID (parent's PID, which is set to the original process), and certain resources and statistics, such as pending signals, which are not inherited. The second function, `exec()`, loads a new executable into the address space and begins executing it. The combination of `fork()` followed by `exec()` is similar to the single function most operating systems provide.

Copy-on-Write

Traditionally, upon `fork()` all resources owned by the parent are duplicated and the copy is given to the child. This approach is significantly naïve and inefficient in that it copies much data that might otherwise be shared. Worse still, if the new process were to immediately execute a new image, all that copying would go to waste. In Linux, `fork()` is implemented through the use of *copy-on-write* pages. Copy-on-write (or *COW*) is a technique to delay or altogether prevent copying of the data. Rather than duplicate the process address space, the parent and the child can share a single copy. The data, however, is marked in such a way that if it is written to, a duplicate is made and each process receives a unique copy. Consequently, the duplication of resources occurs only when they are written; until then, they are shared read-only. This technique delays the copying of each page in the address space until it is actually written to. In the case that the pages are never written—for example, if `exec()` is called immediately after `fork()`—they never need to be copied. The only overhead incurred by `fork()` is the duplication of the parent's page tables and the creation of a unique process descriptor for the child. In the common case that a process executes a new executable image immediately after forking, this optimization prevents the wasted copying of large amounts of data (with the address space, easily tens of megabytes). This is an important optimization because the Unix philosophy encourages quick process execution.

fork()

Linux implements `fork()` via the `clone()` system call. This call takes a series of flags that specify which resources, if any, the parent and child process should share (see the section on "The Linux Implementation of Threads" later in this chapter for more about the flags). The `fork()`, `vfork()`, and `__clone()` library calls all invoke the `clone()` system call with the requisite flags. The `clone()` system call, in turn, calls `do_fork()`.

The bulk of the work in forking is handled by `do_fork()`, which is defined in `kernel/fork.c`. This function calls `copy_process()`, and then starts the process running. The interesting work is done by `copy_process()`:

- It calls `dup_task_struct()`, which creates a new kernel stack, `thread_info` structure, and `task_struct` for the new process. The new values are identical to those of the current task. At this point, the child and parent process descriptors are identical.
- It then checks that the new child will not exceed the resource limits on the number of processes for the current user.
- Now the child needs to differentiate itself from its parent. Various members of the process descriptor are cleared or set to initial values. Members of the process descriptor that are not inherited are primarily statistically information. The bulk of the data in the process descriptor is shared.
- Next, the child's state is set to `TASK_UNINTERRUPTIBLE`, to ensure that it does not yet run.
- Now, `copy_process()` calls `copy_flags()` to update the `flags` member of the `task_struct`. The `PF_SUPERPRIV` flag, which denotes whether a task used super-user privileges, is cleared. The `PF_FORKNOEXEC` flag, which denotes a process that has not called `exec()`, is set.
- Next, it calls `get_pid()` to assign an available PID to the new task.
- Depending on the flags passed to `clone()`, `copy_process()` then either duplicates or shares open files, filesystem information, signal handlers, process address space, and namespace. These resources are typically shared between threads in a given process; otherwise they are unique and thus copied here.
- Next, the remaining timeslice between the parent and its child is split between the two (this is discussed in Chapter 4).
- Finally, `copy_process()` cleans up and returns to the caller a pointer to the new child.

Back in `do_fork()`, if `copy_process()` returns successfully, the new child is woken up and run. Deliberately, the kernel runs the child process first[9]. In the common case of the child simply calling `exec()` immediately, this eliminates any copy-on-write overhead that would occur if the parent ran first and began writing to the address space.

vfork()

The `vfork()` system call has the same effect as `fork()`, except that the page table entries of the parent process are not copied. Instead, the child executes as the sole thread in the parent's address space, and the parent is blocked until the child either calls `exec()` or exits. The child is *not* allowed to write to the address space. This was a welcome optimization in the old days of 3BSD when the call was introduced because at the time copy-on-write pages were not used to implement `fork()`. Today, with copy-on-write and child-runs-first semantics, the only benefit to `vfork()` is not copying the parent page tables entries. If Linux one day gains copy-on-write page table entries there will no longer be any benefit[10]. Because the semantics of `vfork()` are tricky (what, for example, happens if the `exec()` fails?) it would be nice if `vfork()` died a slow painful death. It is entirely possible to implement `vfork()` as a normal `fork()`—in fact, this is what Linux did until 2.2.

The `vfork()` system call is implemented via a special flag to the `clone()` system call:

- In `copy_process()`, the `task_struct` member `vfork_done` is set to `NULL`.

- In `do_fork()`, if the special flag was given, `vfork_done` is pointed at a specific address.
- After the child is first run, the parent—instead of returning—waits for the child to signal it through the `vfork_done` pointer.
- In the `mm_release()` function, which is used when a task exits a memory address space, `vfork_done` is checked to see whether it is `NULL`. If it is not, the parent is signaled.
- Back in `do_fork()`, the parent wakes up and returns.

If this all goes as planned, the child is now executing in a new address space and the parent is again executing in its original address space. The overhead is lower, but the design is not pretty.

The Linux Implementation of Threads

Threads are a popular modern programming abstraction. They provide multiple threads of execution within the same program in a shared memory address space. They can also share open files and other resources. Threads allow for *concurrent programming* and, on multiple processor systems, true *parallelism.*

Linux has a unique implementation of threads. To the Linux kernel, there is *no* concept of a thread. Linux implements all threads as standard processes. The Linux kernel does not provide any special scheduling semantics or data structures to represent threads. Instead, a thread is merely a process that shares certain resources with other processes. Each thread has a unique `task_struct` and appears to the kernel as a normal process (which just happens to share resources, such as an address space, with other processes).

This approach to threads contrasts greatly with operating systems such as Microsoft Windows or Sun Solaris, which have *explicit* kernel support for threads (and sometimes call threads *lightweight processes*). The name "lightweight process" sums up the difference in philosophies between Linux and other systems. To these other operating systems, threads are an abstraction to provide a lighter, quicker execution unit than the heavy process. To Linux, threads are simply a manner of sharing resources between processes (which are already quite lightweight)[11]. For example, assume you have a process that consists of four threads. On systems with explicit thread support, there might exist one process descriptor that in turn points to the four different threads. The process descriptor describes the shared resources, such as an address space or open files. The threads then describe the resources they alone possess. Conversely, in Linux, there are simply four processes and thus four normal `task_struct` structures. The four processes are set up to share certain resources.

Threads are created like normal tasks, with the exception that the `clone()` system call is passed flags corresponding to specific resources to be shared:

```
clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
```

The previous code results in behavior identical to a normal `fork()`, except that the address space, filesystem resources, file descriptors, and signal handlers are shared. In other words, the new task and its parent are what are popularly called *threads*.

In contrast, a normal `fork()` can be implemented as

```
clone(SIGCHLD, 0);
```

And `vfork()` is implemented as

```
clone(CLONE_VFORK | CLONE_VM | SIGCHLD, 0);
```

The flags provided to `clone()` help specify the behavior of the new process and detail what resources the parent and child will share. Table 3.1 lists the clone flags, which are defined in `<linux/sched.h>`, and their effect.

Table 3.1 clone() Flags

Flag	**Meaning**
`CLONE_FILES`	Parent and child share open files.
`CLONE_FS`	Parent and child share filesystem information.

Flag	Meaning
`CLONE_IDLETASK`	Set PID to zero (used only by the idle tasks).
`CLONE_NEWNS`	Create a new namespace for the child.
`CLONE_PARENT`	Child is to have same parent as its parent.
`CLONE_PTRACE`	Continue tracing child.
`CLONE_SETTID`	Write the TID back to user-space.
`CLONE_SETTLS`	Create a new TLS for the child.
`CLONE_SIGHAND`	Parent and child share signal handlers and blocked signals.
`CLONE_SYSVSEM`	Parent and child share System V `SEM_UNDO` semantics.
`CLONE_THREAD`	Parent and child are in the same thread group.
`CLONE_VFORK`	`vfork()` was used and the parent will sleep until the child wakes it.
`CLONE_UNTRACED`	Do not let the tracing process force `CLONE_PTRACE` on the child.
`CLONE_STOP`	Start process in the `TASK_STOPPED` state.
`CLONE_SETTLS`	Create a new TLS (thread-local storage) for the child.
`CLONE_CHILD_CLEARTID`	Clear the TID in the child.
`CLONE_CHILD_SETTID`	Set the TID in the child.
`CLONE_PARENT_SETTID`	Set the TID in the parent.

Flag	Meaning
`CLONE_VM`	Parent and child share address space.

Kernel Threads

It is often useful for the kernel to perform some operations in the background. The kernel accomplishes 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`). 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.

We will discuss specific kernel threads in more detail in later chapters.

Process Termination

It is sad, but eventually processes must die. When a process terminates, the kernel releases the resources owned by the process and notifies the child's parent of its unfortunate demise.

Typically, process destruction occurs when the process calls the `exit()` system call, either explicitly when it is ready to terminate or implicitly on return from the main subroutine of any program (that is, the C compiler places a call to `exit()` after `main()` returns). A process can also terminate involuntarily. This occurs when the process receives a signal or exception it cannot handle or ignore. Regardless of how a process terminates, the bulk of the work is handled by `do_exit()`, which completes a number of chores:

- First, it set the `PF_EXITING` flag in the flags member of the `task_struct`.
- Second, it calls `del_timer_sync()` to remove any kernel timers. Upon return, it is guaranteed that no timer is queued and that no timer handler is running.
- Next, if BSD process accounting is enabled, `do_exit()` calls `acct_process()` to write out accounting information.
- Now it calls `__exit_mm()` to release the `mm_struct` held by this process. If no other process is using this address space (in other words, if it is not shared), then deallocate it.
- Next, it calls `exit_sem()`. If the process is queued waiting for an IPC semaphore, it is dequeued here.
- It then calls `__exit_files()`, `__exit_fs()`, `exit_namespace()`, and `exit_sighand()` to decrement the usage count of objects related to file descriptors, filesystem data, the process namespace, and signal handlers, respectively. If any usage counts reach zero, the object is no longer in use by any process and it is removed.
- Subsequently, it sets the task's exit code, stored in the `exit_code` member of the `task_struct`, to the code provided by `exit()` or whatever kernel mechanism forced the termination. The exit code is stored here for optional retrieval by the parent.
- It then calls `exit_notify()` to send signals to the task's parent, reparents any of the task's children to another thread in their thread group or the init process, and sets the task's state to `TASK_ZOMBIE`.

- Finally, do_exit() calls schedule() to switch to a new process (see Chapter 4). Because TASK_ZOMBIE tasks are never scheduled, this is the last code the task will ever execute.

The code for do_exit() is defined in kernel/exit.c.

At this point, all objects associated with the task (assuming the task was the sole user) are freed. The task is not runnable (and in fact no longer has an address space in which to run) and is in the TASK_ZOMBIE state. The only memory it occupies is its kernel stack, the thread_info structure, and the task_struct structure. The task exists solely to provide information to its parent. After the parent retrieves the information, or notifies the kernel that it is uninterested, the remaining memory held by the process is freed and returned to the system for use.

Removal of the Process Descriptor

After do_exit() completes, the process descriptor for the terminated process still exists but the process is a zombie and is unable to run. As discussed, this allows the system to obtain information about a child process after it has terminated. Consequently, the acts of cleaning up after a process and removing its process descriptor are separate. After the parent has obtained information on its terminated child, or signified to the kernel that it does not care, the child's task_struct is deallocated.

The wait() family of functions are implemented via a single (and complicated) system call, wait4(). The standard behavior is to suspend execution of the calling task until one of its children exits, at which time the function returns with the PID of the exited child. Additionally, a pointer is provided to the function that on return holds the exit code of the terminated child.

When it is time to finally deallocate the process descriptor, release_task() is invoked. It does the following:

- First, it calls free_uid() to decrement the usage count of the process's user. Linux keeps a per-user cache of information related to how many processes and files a user has opened. If the usage count reaches zero, the user has no more open processes or files and the cache is destroyed.

- Second, release_task() calls unhash_process() to remove the process from the pidhash and remove the process from the task list.

- Next, if the task was ptraced, release_task() reparents the task to its original parent and removes it from the ptrace list.

- Ultimately, release_task(), calls put_task_struct() to free the pages containing the process's kernel stack and thread_info structure and deallocate the slab cache containing the task_struct.

At this point, the process descriptor and all resources belonging solely to the process have been freed.

The Dilemma of the Parentless Task

If a parent exits before its children, some mechanism must exist to *reparent* the child tasks to a new process, or else parentless terminated processes would forever remain zombies, wasting system memory. The solution, hinted upon previously, is to reparent a task's children on exit to either another process in the current thread group or, if that fails, the init process. In do_exit(), notify_parent() is invoked, which calls forget_original_parent() to perform the reparenting:

```
struct task_struct *p, *reaper = father;
struct list_head *list;

if (father->exit_signal != -1)
    reaper = prev_thread(reaper);
else
    reaper = child_reaper;

if (reaper == father)
    reaper = child_reaper;
```

This code sets reaper to another task in the process's thread group. If there is not another task in the thread group, it sets reaper to child_reaper, which is the init process. Now that a suitable new parent for the children is found, each child needs to be located and reparented to reaper:

```
list_for_each(list, &father->children) {
    p = list_entry(list, struct task_struct, sibling);
    reparent_thread(p, reaper, child_reaper);
}

list_for_each(list, &father->ptrace_children) {
    p = list_entry(list, struct task_struct, ptrace_list);
    reparent_thread(p, reaper, child_reaper);
}
```

This code iterates over two lists: the *child list* and the *ptraced child list*, reparenting each child. The rationale behind having both lists is interesting; it is a new feature in the 2.6 kernel. When a task is *ptraced,* it is temporarily reparented to the debugging process. When the task's parent exits, however, it must be reparented along with its other siblings. In previous kernels, this resulted in a loop over *every process in the system* looking for children. The solution, as noted previously, is simply to keep a separate list of a process's children that are being ptraced—reducing the search for one's children from every process to just two relatively small lists.

With the process successfully reparented, there is no risk of stray zombie processes. The `init` process routinely calls `wait()` on its children, cleaning up any zombies assigned to it.

Process Wrap Up

In this chapter, we looked at the famed operating system abstraction of the *process*. We discussed the generalities of the process, why it is important, and the relationship between processes and threads. We then discussed how Linux stores and represents processes (with `task_struct` and `thread_info`), how processes are created (via `clone()` and `fork()`), how new executable images are loaded into address spaces (via the `exec()` family of system calls), the hierarchy of processes, how parents glean information about their deceased children (via the `wait()` family of system calls), and how processes ultimately die (forcefully or intentionally via `exit()`).

The process is a fundamental and crucial abstraction, at the heart of every modern operating system, and ultimately the reason we have operating systems altogether (to run programs).

The next chapter discusses process scheduling, which is the delicate and interesting manner in which the kernel decides which processes to run, at what time, and in what order.

Footnotes

1. The other fundamental abstraction is files.
2. The kernel implements the wait4() system call. Linux systems, via the C library, typically provide the wait(), waitpid(), wait3(), and wait4() functions. All these functions return status about a terminated process, albeit with slightly different semantics.
3. 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*.
4. Register-impaired architectures were not the only reason for creating struct thread_info.
5. An opaque type is a data type whose physical representation is unknown or irrelevant.
6. This is why you have those dreaded unkillable processes with state D in ps(1). Because the task will not respond to signals, you cannot send it a SIGKILL signal. Further, even if you could terminate the task, it would not be wise as the task is supposedly in the middle of an important operation and may hold a semaphore.
7. Other than process context there is *interrupt context*, which we discuss in Chapter 6, "Interrupts and Interrupt Handlers." In interrupt context, the system is not running on behalf of a process, but is executing an interrupt handler. There is no process tied to interrupt handlers and consequently no process context.
8. By exec() I mean any member of the exec() family of functions. The kernel implements the execve() system call on top of which execlp(), execle(), execv(), and execvp() are implemented.
9. Amusingly, this does not currently function correctly, although the goal is for the child to run first.
10. In fact, there are currently patches to add this functionality to Linux. In time, this feature will most likely find its way into the mainline Linux kernel.
11. As an example, benchmark process creation time in Linux versus process (or even thread!) creation time in these other operating systems. The results are quite nice.

© 2009 Pearson Education, Inc. All rights reserved.
800 East 96th Street Indianapolis, Indiana 46240