< open Table of Content

Team LiB

PREVIOUS NEXT

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.

> [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.

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.
`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.
`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.

PREVIOUS NEXT