4.6 LINUX PROCESS AND THREAD MANAGEMENT

Linux Tasks

A process, or task, in Linux is represented by a `task_struct` data structure. The `task_struct` data structure contains information in a number of categories:

- **State:** The execution state of the process (executing, ready, suspended, stopped, zombie). This is described subsequently.

- **Scheduling information:** Information needed by Linux to schedule processes. A process can be normal or real time and has a priority. Real-time processes are scheduled before normal processes, and within each category, relative priorities can be used. A counter keeps track of the amount of time a process is allowed to execute.
- **Identifiers:** Each process has a unique process identifier and also has user and group identifiers. A group identifier is used to assign resource access privileges to a group of processes.
- **Interprocess communication:** Linux supports the IPC mechanisms found in UNIX SVR4, described in Chapter 6.
- **Links:** Each process includes a link to its parent process, links to its siblings (processes with the same parent), and links to all of its children.
- **Times and timers:** Includes process creation time and the amount of processor time so far consumed by the process. A process may also have associated one or more interval timers. A process defines an interval timer by means of a system call; as a result a signal is sent to the process when the timer expires. A timer may be single use or periodic.
- **File system:** Includes pointers to any files opened by this process, as well as pointers to the current and the root directories for this process.
- **Address space:** Defines the virtual address space assigned to this process.
- **Processor-specific context:** The registers and stack information that constitute the context of this process.

Figure 4.18 shows the execution states of a process. These are as follows:

- **Running:** This state value corresponds to two states. A Running process is either executing or it is ready to execute.
- **Interruptible:** This is a blocked state, in which the process is waiting for an event, such as the end of an I/O operation, the availability of a resource, or a signal from another process.
- **Uninterruptible:** This is another blocked state. The difference between this and the Interruptible state is that in an uninterruptible state, a process is waiting directly on hardware conditions and therefore will not handle any signals.
- **Stopped:** The process has been halted and can only resume by positive action from another process. For example, a process that is being debugged can be put into the Stopped state.
- **Zombie:** The process has been terminated but, for some reason, still must have its task structure in the process table.

Linux Threads

Traditional UNIX systems support a single thread of execution per process, while modern UNIX systems typically provide support for multiple kernel-level threads per process. As with traditional UNIX systems, older versions of the Linux kernel offered no support for multithreading. Instead, applications would need to be written with a set of user-level library functions, the most popular of which is

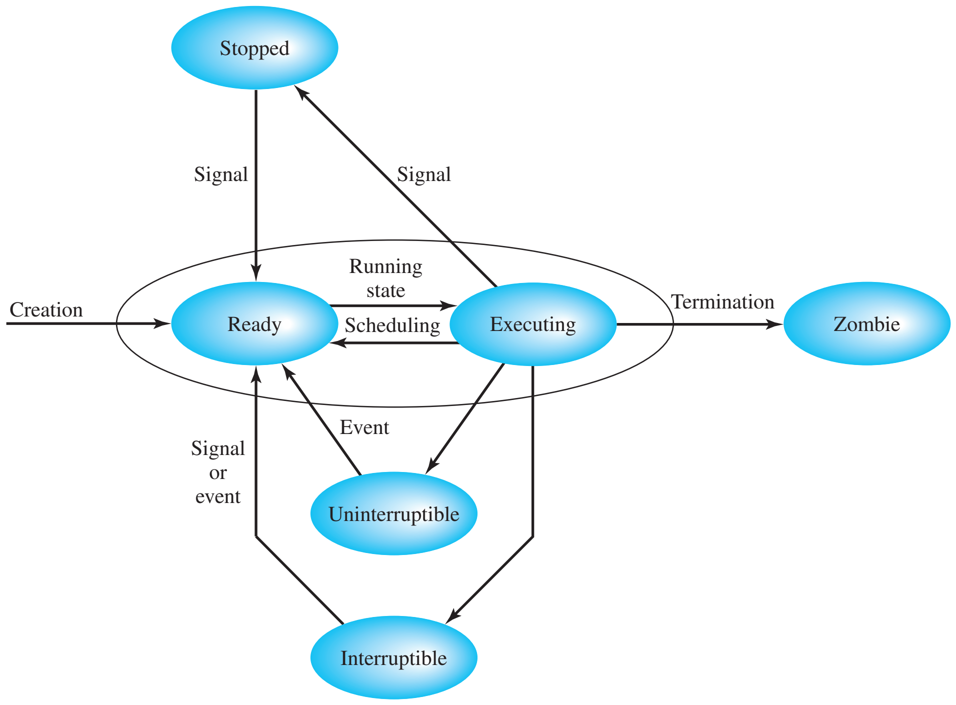

Figure 4.18 Linux Process/Thread Model

known as *pthread (POSIX thread) libraries*, with all of the threads mapping into a single kernel-level process.[11] We have seen that modern versions of UNIX offer kernel-level threads. Linux provides a unique solution in that it does not recognize a distinction between threads and processes. Using a mechanism similar to the lightweight processes of Solaris, user-level threads are mapped into kernel-level processes. Multiple user-level threads that constitute a single user-level process are mapped into Linux kernel-level processes that share the same group ID. This enables these processes to share resources such as files and memory and to avoid the need for a context switch when the scheduler switches among processes in the same group.

A new process is created in Linux by copying the attributes of the current process. A new process can be *cloned* so that it shares resources, such as files, signal handlers, and virtual memory. When the two processes share the same virtual memory, they function as threads within a single process. However, no separate type of data structure is defined for a thread. In place of the usual fork() command, processes are created in Linux using the clone() command. This command includes a set of flags as arguments, defined in Table 4.5. The traditional fork() system call is implemented by Linux as a clone() system call with all of the clone flags cleared.

[11]POSIX (Portable Operating Systems based on UNIX) is an IEEE API standard that includes a standard for a thread API. Libraries implementing the POSIX Threads standard are often named *Pthreads*. Pthreads are most commonly used on UNIX-like POSIX systems such as Linux and Solaris, but Microsoft Windows implementations also exist.

Table 4.5 Linux clone () flags

CLONE_CLEARID	Clear the task ID.
CLONE_DETACHED	The parent does not want a SIGCHLD signal sent on exit.
CLONE_FILES	Shares the table that identifies the open files.
CLONE_FS	Shares the table that identifies the root directory and the current working directory, as well as the value of the bit mask used to mask the initial file permissions of a new file.
CLONE_IDLETASK	Set PID to zero, which refers to an idle task. The idle task is employed when all available tasks are blocked waiting for resources.
CLONE_NEWNS	Create a new namespace for the child.
CLONE_PARENT	Caller and new task share the same parent process.
CLONE_PTRACE	If the parent process is being traced, the child process will also be traced.
CLONE_SETTID	Write the TID back to user space.
CLONE_SETTLS	Create a new TLS for the child.
CLONE_SIGHAND	Shares the table that identifies the signal handlers.
CLONE_SYSVSEM	Shares System V SEM_UNDO semantics.
CLONE_THREAD	Inserts this process into the same thread group of the parent. If this flag is true, it implicitly enforces CLONE_PARENT.
CLONE_VFORK	If set, the parent does not get scheduled for execution until the child invokes the *execve()* system call.
CLONE_VM	Shares the address space (memory descriptor and all page tables):

When the Linux kernel performs a switch from one process to another, it checks whether the address of the page directory of the current process is the same as that of the to-be-scheduled process. If they are, then they are sharing the same address space, so that a context switch is basically just a jump from one location of code to another location of code.

Although cloned processes that are part of the same process group can share the same memory space, they cannot share the same user stacks. Thus the clone() call creates separate stack spaces for each process.