

3.2.3 Context Switch

As mentioned in Section 1.2.1, interrupts cause the operating system to change a CPU core from its current task and to run a kernel routine. Such operations happen frequently on general-purpose systems. When an interrupt occurs, the system needs to save the current **context** of the process running on the CPU core so that it can restore that context when its processing is done, essentially suspending the process and then resuming it. The context is represented in the PCB of the process. It includes the value of the CPU registers, the process state (see Figure 3.2), and memory-management information. Generically, we perform a **state save** of the current state of the CPU core, be it in kernel or user mode, and then a **state restore** to resume operations.

Switching the CPU core to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a **context switch** and is illustrated in Figure 3.6. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context-switch time is pure overhead, because the system does no useful work while switching. Switching speed varies from machine to machine, depending on the

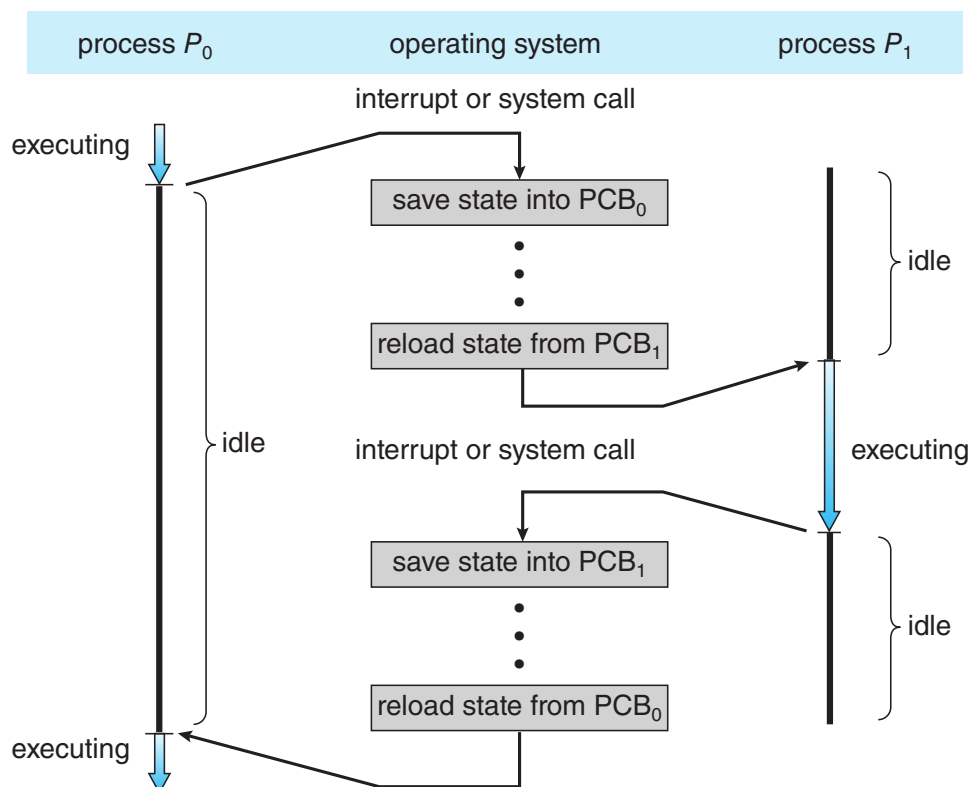


Figure 3.6 Diagram showing context switch from process to process.

MULTITASKING IN MOBILE SYSTEMS

Because of the constraints imposed on mobile devices, early versions of iOS did not provide user-application multitasking; only one application ran in the foreground while all other user applications were suspended. Operating-system tasks were multitasked because they were written by Apple and well behaved. However, beginning with iOS 4, Apple provided a limited form of multitasking for user applications, thus allowing a single foreground application to run concurrently with multiple background applications. (On a mobile device, the **foreground** application is the application currently open and appearing on the display. The **background** application remains in memory, but does not occupy the display screen.) The iOS 4 programming API provided support for multitasking, thus allowing a process to run in the background without being suspended. However, it was limited and only available for a few application types. As hardware for mobile devices began to offer larger memory capacities, multiple processing cores, and greater battery life, subsequent versions of iOS began to support richer functionality for multitasking with fewer restrictions. For example, the larger screen on iPad tablets allowed running two foreground apps at the same time, a technique known as **split-screen**.

Since its origins, Android has supported multitasking and does not place constraints on the types of applications that can run in the background. If an application requires processing while in the background, the application must use a **service**, a separate application component that runs on behalf of the background process. Consider a streaming audio application: if the application moves to the background, the service continues to send audio data to the audio device driver on behalf of the background application. In fact, the service will continue to run even if the background application is suspended. Services do not have a user interface and have a small memory footprint, thus providing an efficient technique for multitasking in a mobile environment.

memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). A typical speed is a several microseconds.

Context-switch times are highly dependent on hardware support. For instance, some processors provide multiple sets of registers. A context switch here simply requires changing the pointer to the current register set. Of course, if there are more active processes than there are register sets, the system resorts to copying register data to and from memory, as before. Also, the more complex the operating system, the greater the amount of work that must be done during a context switch. As we will see in Chapter 9, advanced memory-management techniques may require that extra data be switched with each context. For instance, the address space of the current process must be preserved as the space of the next task is prepared for use. How the address space is preserved, and what amount of work is needed to preserve it, depend on the memory-management method of the operating system.

3.3 Operations on Processes

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination. In this section, we explore the mechanisms involved in creating processes and illustrate process creation on UNIX and Windows systems.

3.3.1 Process Creation

During the course of execution, a process may create several new processes. As mentioned earlier, the creating process is called a parent process, and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a **tree** of processes.

Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique **process identifier** (or **pid**), which is typically an integer number. The pid provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel.

Figure 3.7 illustrates a typical process tree for the Linux operating system, showing the name of each process and its pid. (We use the term *process* rather loosely in this situation, as Linux prefers the term *task* instead.) The `systemd` process (which always has a pid of 1) serves as the root parent process for all user processes, and is the first user process created when the system boots. Once the system has booted, the `systemd` process creates processes which provide additional services such as a web or print server, an ssh server, and the like. In Figure 3.7, we see two children of `systemd`—`logind` and `sshd`. The `logind` process is responsible for managing clients that directly log onto the system. In this example, a client has logged on and is using the bash shell, which has been assigned pid 8416. Using the bash command-line interface, this user has created the process `ps` as well as the `vim` editor. The `sshd` process is responsible for managing clients that connect to the system by using ssh (which is short for *secure shell*).

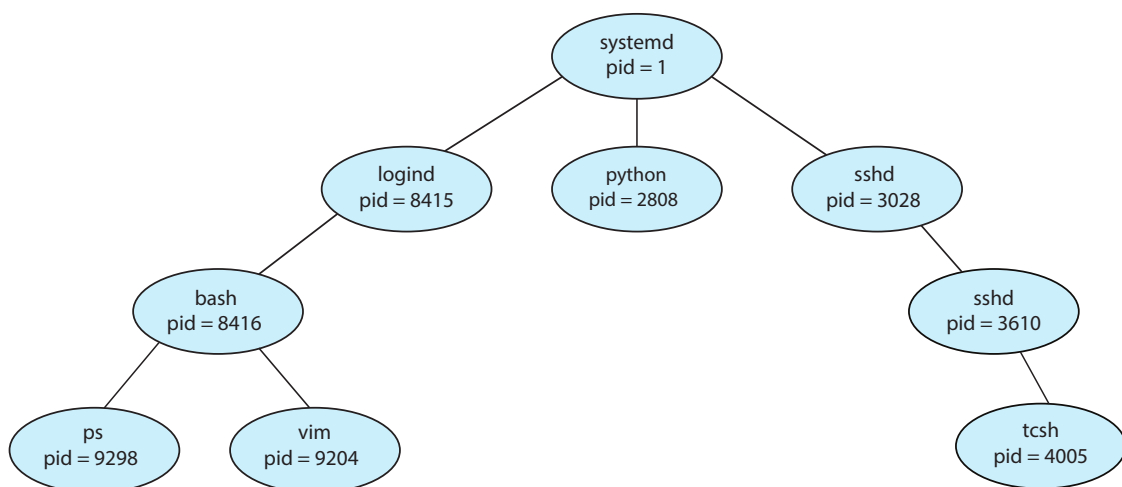


Figure 3.7 A tree of processes on a typical Linux system.

THE `init` AND `systemd` PROCESSES

Traditional UNIX systems identify the process `init` as the root of all child processes. `init` (also known as **System V `init`**) is assigned a pid of 1, and is the first process created when the system is booted. On a process tree similar to what is shown in Figure 3.7, `init` is at the root.

Linux systems initially adopted the System V `init` approach, but recent distributions have replaced it with `systemd`. As described in Section 3.3.1, `systemd` serves as the system's initial process, much the same as System V `init`; however it is much more flexible, and can provide more services, than `init`.

On UNIX and Linux systems, we can obtain a listing of processes by using the `ps` command. For example, the command

```
ps -el
```

will list complete information for all processes currently active in the system. A process tree similar to the one shown in Figure 3.7 can be constructed by recursively tracing parent processes all the way to the `systemd` process. (In addition, Linux systems provide the `pstree` command, which displays a tree of all processes in the system.)

In general, when a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes.

In addition to supplying various physical and logical resources, the parent process may pass along initialization data (input) to the child process. For example, consider a process whose function is to display the contents of a file—say, `hw1.c`—on the screen of a terminal. When the process is created, it will get, as an input from its parent process, the name of the file `hw1.c`. Using that file name, it will open the file and write the contents out. It may also get the name of the output device. Alternatively, some operating systems pass resources to child processes. On such a system, the new process may get two open files, `hw1.c` and the terminal device, and may simply transfer the datum between the two.

When a process creates a new process, two possibilities for execution exist:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two address-space possibilities for the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
2. The child process has a new program loaded into it.

To illustrate these differences, let's first consider the UNIX operating system. In UNIX, as we've seen, each process is identified by its process identifier, which is a unique integer. A new process is created by the `fork()` system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the `fork()`, with one difference: the return code for the `fork()` is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

After a `fork()` system call, one of the two processes typically uses the `exec()` system call to replace the process's memory space with a new program. The `exec()` system call loads a binary file into memory (destroying the memory image of the program containing the `exec()` system call) and starts

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

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```

Figure 3.8 Creating a separate process using the UNIX `fork()` system call.

its execution. In this manner, the two processes are able to communicate and then go their separate ways. The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a `wait()` system call to move itself off the ready queue until the termination of the child. Because the call to `exec()` overlays the process's address space with a new program, `exec()` does not return control unless an error occurs.

The C program shown in Figure 3.8 illustrates the UNIX system calls previously described. We now have two different processes running copies of the same program. The only difference is that the value of the variable `pid` for the child process is zero, while that for the parent is an integer value greater than zero (in fact, it is the actual `pid` of the child process). The child process inherits privileges and scheduling attributes from the parent, as well certain resources, such as open files. The child process then overlays its address space with the UNIX command `/bin/ls` (used to get a directory listing) using the `exec1p()` system call (`exec1p()` is a version of the `exec()` system call). The parent waits for the child process to complete with the `wait()` system call. When the child process completes (by either implicitly or explicitly invoking `exit()`), the parent process resumes from the call to `wait()`, where it completes using the `exit()` system call. This is also illustrated in Figure 3.9.

Of course, there is nothing to prevent the child from *not* invoking `exec()` and instead continuing to execute as a copy of the parent process. In this scenario, the parent and child are concurrent processes running the same code instructions. Because the child is a copy of the parent, each process has its own copy of any data.

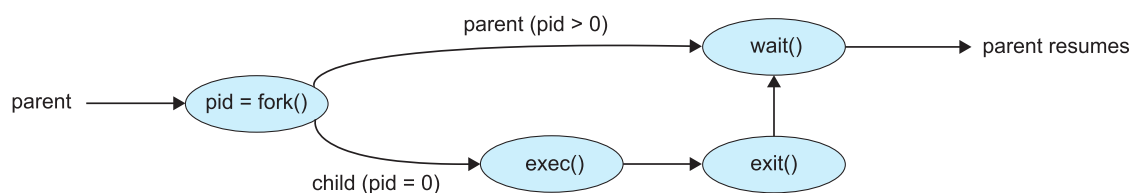


Figure 3.9 Process creation using the `fork()` system call.

3.3.2 Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the `exit()` system call. At that point, the process may return a status value (typically an integer) to its waiting parent process (via the `wait()` system call). All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated and reclaimed by the operating system.

Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, `TerminateProcess()` in Windows). Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, a user—or a misbehaving application—could arbitrarily kill another user's processes. Note that a parent needs to know the identities of its children if it is to terminate them. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

- The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Some systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as **cascading termination**, is normally initiated by the operating system.

To illustrate process execution and termination, consider that, in Linux and UNIX systems, we can terminate a process by using the `exit()` system call, providing an exit status as a parameter:

```
/* exit with status 1 */  
exit(1);
```


In fact, under normal termination, `exit()` will be called either directly (as shown above) or indirectly, as the C run-time library (which is added to UNIX executable files) will include a call to `exit()` by default.

A parent process may wait for the termination of a child process by using the `wait()` system call. The `wait()` system call is passed a parameter that allows the parent to obtain the exit status of the child. This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated:

```
pid_t pid;
int status;

pid = wait(&status);
```

When a process terminates, its resources are deallocated by the operating system. However, its entry in the process table must remain there until the parent calls `wait()`, because the process table contains the process's exit status. A process that has terminated, but whose parent has not yet called `wait()`, is known as a **zombie** process. All processes transition to this state when they terminate, but generally they exist as zombies only briefly. Once the parent calls `wait()`, the process identifier of the zombie process and its entry in the process table are released.

Now consider what would happen if a parent did not invoke `wait()` and instead terminated, thereby leaving its child processes as **orphans**. Traditional UNIX systems addressed this scenario by assigning the `init` process as the new parent to orphan processes. (Recall from Section 3.3.1 that `init` serves as the root of the process hierarchy in UNIX systems.) The `init` process periodically invokes `wait()`, thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.

Although most Linux systems have replaced `init` with `systemd`, the latter process can still serve the same role, although Linux also allows processes other than `systemd` to inherit orphan processes and manage their termination.