the kernel like the swapper), although it does run with superuser privileges. Later in this chapter we’ll see how init becomes the parent process of any orphaned child process.

On some virtual memory implementations of Unix, process ID 2 is the pagedaemon. This process is responsible for supporting the paging of the virtual memory system. Like the swapper, the pagedaemon is a kernel process.

In addition to the process ID, there are other identifiers for every process. The following functions return these identifiers.

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

pid_t getpid(void); // Returns: process ID of calling process

pid_t getppid(void); // Returns: parent process ID of calling process

uid_t getuid(void); // Returns: real user ID of calling process

uid_t geteuid(void); // Returns: effective user ID of calling process

gid_t getgid(void); // Returns: real group ID of calling process

gid_t getegid(void); // Returns: effective group ID of calling process
```

Note that none of these functions has an error return. We’ll return to the parent process ID in the next section when we discuss the fork function. The real and effective user and group IDs were discussed in Section 4.4.

8.3 fork Function

The only way a new process is created by the Unix kernel is when an existing process calls the fork function. (This doesn’t apply to the special processes that we mentioned in the previous section—the swapper, init, and the pagedaemon. These processes are created specially by the kernel as part of the bootstrapping.)

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

pid_t fork(void); // Returns: 0 in child, process ID of child in parent, -1 on error
```

The new process created by fork is called the child process. This function is called once but returns twice. The only difference in the returns is that the return value in the child is 0 while the return value in the parent is the process ID of the new child. The reason the child’s process ID is returned to the parent is because a process can have more than one child, so there is no function that allows a process to obtain the process IDs of its

Example

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

pid_t fork(void);
```

In general, this direct use of fork is discouraged because it makes the code difficult to understand and maintain. Instead, it is recommended to use process creation utilities such as the ones provided by the C runtime library. These utilities abstract away the details of process creation and provide safer, more reliable, and more robust interfaces for creating and managing processes.
fork Function

Later in child process...

The fork function returns 0 to the child because a process can have only a single parent, so the child can call getpid to obtain the process ID of its parent. (Process ID 0 is always in use by the swapper, so it's not possible for 0 to be the process ID of a child.)

Both the child and parent continue executing with the instruction that follows the call to fork. The child is a copy of the parent. For example, the child gets a copy of the parent's data space, heap, and stack. Note that this is a copy for the child—the parent and child do not share these portions of memory. Often the parent and child share the text segment (Section 7.6), if it is read-only.

Many current implementations don't perform a complete copy of the parent's data, stack, and heap, since a fork is often followed by an exec. Instead, a technique called copy-on-write (COW) is used. These regions are shared by the parent and child and have their protection changed by the kernel to read-only. If either process tries to modify these regions, the kernel then makes a copy of that piece of memory only, typically a "page" in a virtual memory system. Section 9.2 of Bach [1986] and Section 5.7 of Leffler et al. [1989] provide more detail on this feature.

Example

Program 8.1 demonstrates the fork function. If we execute this program we get

```bash
$ a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89 child's variables were changed
pid = 429, glob = 6, var = 88 parent's copy were not changed
$ a.out > temp.out
$ cat temp.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89
before fork
pid = 431, glob = 6, var = 88
```

In general, we never know if the child starts executing before the parent or vice versa. This depends on the scheduling algorithm used by the kernel. If it's required that the child and parent synchronize with each other, some form of interprocess communication is required. In Program 8.1 we just have the parent put itself to sleep for 2 seconds, to let the child execute. There is no guarantee that this is adequate, and we talk about this and other types of synchronization in Section 8.8 when we talk about race conditions. In Section 10.16 we show how to synchronize a parent and child after a fork using signals.

Note the interaction of fork with the I/O functions in Program 8.1. Recall from Chapter 3 that the write function is not buffered. Since write is called before the fork, its data is written once to standard output. The standard I/O library, however, is buffered. Recall from Section 5.12 that standard output is line buffered if it's connected to a terminal device, otherwise it's fully buffered. When we run the program interactively we get only a single copy of the printf line, because the standard output buffer...
```c
#include <sys/types.h>
#include "ourhdr.h"

int glob = 6; /* external variable in initialized data */
char buf[] = "a write to stdout\n";

int main(void)
{
    int var; /* automatic variable on the stack */
    pid_t pid;

    var = 88;
    if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("write error");
    printf("before fork\n"); /* we don't flush stdout */
    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) { /* child */
        glob++;
        /* modify variables */
        var++;
    } else
        sleep(2); /* parent */
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

Program 8.1 Example of fork function.

is flushed by the newline. But when we redirect standard output to a file we get two copies of the printf line. What has happened in this second case is that the printf before the fork is called once, but the line remains in the buffer when fork is called. This buffer is then copied into the child, when the parent’s data space is copied to the child. Both the parent and child now have a standard I/O buffer with this line in it. The second printf, right before the exit, just appends its data to the existing buffer. When each process terminates, its copy of the buffer is finally flushed.

File Sharing

Another point to note from Program 8.1 is, when we redirect the standard output of the parent, the child’s standard output is also redirected. Indeed, one characteristic of fork is that all descriptors that are open in the parent are duplicated in the child. We say “duplicated” because it’s as if the dup function had been called for each descriptor. The parent and child share a file table entry for every open descriptor (recall Figure 3.4).

Consider a process that has three different files opened for standard input, standard output, and standard error. On return from fork we have the arrangement shown in Figure 8.1.