Chapter 7 Process Architecture and Control

Concepts Covered

Memory architecture of a process
Memory structures
Viewing memory layout
Process structure
Executable file format
Process creation
Process synchronization
file, nohup, pgrep, ps, psg, readelf, strings
API: atexit, brk, sbrk, malloc, calloc, fork, execve, execlp, ..., exit, on_exit, vfork, wait, waitpid, waitid

7.1 Introduction

In a typical operating systems course, a process is defined to be a program in execution, or something similar to that. This is a true and accurate abstraction. A program such as the bash shell can have many, many instances running on a multi-user machine and each individual instance is a separate and distinct process, although each and every one of these is executing the exact same executable file. What this definition does not tell you is what a process is in concrete terms. It is like saying that a baseball game is an instance of the implementation of a set of rules created by Alexander Cartwright in 1845 by which two teams compete against each other on a playing field. Neither definition gives you a mental picture of the thing being defined.

In this chapter, we focus on the concrete representation of a process in UNIX: how it is represented within the kernel, what kinds of resources it requires, how those resources are materialized and managed, what attributes it has, and what system calls are related to its control and management. As a first step we will look at processes from the command level. Afterward, we will look at how UNIX systems arrange their address spaces and manage them in virtual and physical memory. Then we will look at how processes are created and how they communicate and synchronize with each other.

7.2 Examining Processes on the Command Line

The `ps` command is used for viewing one or more processes currently known to the operating system. I say "currently known" as opposed to "active" because the list may include processes that are technically not active, so called *zombie* processes. The set of options available for the `ps` command is very system dependent. There were different versions of it in BSD systems and in Version 7, and then there are options added by GNU. RedHat Linux systems support all of the historical options, and so there are many different ways to use this command in Linux. In Linux, users also have the option of running the top command. The `top` command is very much like `ps`, except that it displays the dynamic, real-time view of the state of the system. It is like the Windows task manager, and also like a terminal-based version of the Gnome System Monitor. Here we will describe the standard syntax rather than the BSD style or GNU syntax.

The `ps` command without options displays the list of processes of the login id that executes it, in a short form:

```
$ ps
  PID TTY      TIME CMD
14244 pts/1    00:00:00 bash
14572 pts/1    00:00:00 ps
```

You will notice that it always contains a line for the command itself because it itself has to run to do its job, obviously. The `-f` option causes it to display a full listing:

```
$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
sweiss    2508  2507  0 12:09 pts/8    00:00:00 -bash
sweiss    3132  2508  0 12:22 pts/8    00:00:00 ps -f
```

The `UID` column is the user login name. The `PID` column is the process id of the process. The `PPID` column is the parent process's process id. The `C` field is rarely of interest; it gives processor utilization information. The `STIME` field is the starting time in hours, minutes, and seconds. The `TIME` column is the cumulative execution time, which for short commands will be zero, since the smallest unit of measurement in this column is seconds. The `CMD` column is the command line being executed by this process; in this case there are two: `bash` and "`ps -f`". All command line arguments are displayed; if any are suppressed, the command will appear in square brackets:

```
root      3080     1  0 Jan29 ?        00:00:00 [lockd]
```

The `TTY` column is the controlling terminal attached to this process. Some processes have no terminal, in which case a "?" will appear.

The `-e` option displays all processes, which will be quite long. If I want to know which process is the parent of my `bash` process, I can use `ps -ef` and filter using `grep`:

```
$ ps -ef | grep  2507
sweiss    2507  2504  0 12:09 ?        00:00:00 sshd: sweiss@pts/8
sweiss    2508  2507  0 12:09 pts/8    00:00:00 -bash
sweiss    3207  2508  0 12:30 pts/8    00:00:00 grep 2507
```

From this output you see that I am connected via `ssh` on pseudo-terminal `pts/8` and that the `ssh` daemon `sshd` is the parent of my `bash` process.

You can learn a lot about a system just by running `ps`. For example, on our Linux system, the first few processes in the system are:

```
$ ps -ef | head -4
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 Jan29 ?        00:00:01 init [5]
root         2     1  0 Jan29 ?        00:00:02 [migration/0]
root         3     1  0 Jan29 ?        00:00:00 [ksoftirqd/0]
```

whereas on our Solaris 9 server, the list is:

```
$ ps -ef | head -4
UID   PID  PPID  C    STIME TTY      TIME CMD
root     0     0  0   Mar 13 ?        0:23 sched
root     1     0  0   Mar 13 ?        0:00 /etc/init -
root     2     0  0   Mar 13 ?        0:00 pageout
```

Notice that in Solaris, the (CPU) process scheduler itself is the very first process in the system. It is absent in Linux. In all UNIX systems, the process with PID 1 is always init. In Solaris, the `pageout` process is responsible for writing pages to disk, and `fsflush` flushes system buffers to disk.

The `-u` and `-U` options are useful for viewing all of your processes or those of others in a supplied user list. The list of users must be comma-separated, with no intervening spaces. For example:

```
$ ps -f -U sweiss,wgrzemsk
UID        PID  PPID  C STIME TTY          TIME CMD
sweiss    2507  2504  0 12:09 ?        00:00:00 sshd: sweiss@pts/8
sweiss    2508  2507  0 12:09 pts/8    00:00:00 -bash
wgrzemsk  2572  2570  0 12:10 ?        00:00:00 sshd: wgrzemsk@notty
wgrzemsk  2575  2573  0 12:10 ?        00:00:00 /bin/sh
```

While there are dozens of other options, I will only mention one more: the `-o` option. You can customize the output of the `ps` command to include any of the dozens of attributes available to be displayed using `-o`. The man page gives the general format for this. Some examples from the man page:

```
ps -eo pid,tid,class,rtprio,ni,pri,psr,pcpu,stat,wchan:14,comm
ps axo stat,euid,ruid,tty,tpgid,sess,pgrp,ppid,pid,pcpu,comm
ps -eopid,tt,user,fname,tmout,f,wchan
```

Note that there are no spaces in the list. In general never use spaces in any of the lists because the shell will then treat them as separate words rather than a single word to be passed to the `ps` command itself.

A related command is `pgrep`. If you need the process id of a command or program that is running, typing `pgrep` <*executable name*> will give you a list of processes running that program, one per line. For example

```
$ pgrep bash
2508
3502
3621
```

showing that three instances of `bash` are running, with pids 2508, 3502, and 3621.

7.3 Process Groups

UNIX systems allow processes to be placed into groups. There are several reasons for grouping processes. One is that a signal can be sent to an entire process group rather than a single process. For example, the shell arranges that all processes created in order to carry out the command line are in a single group, so that if the user needs to terminate that command, a single signal sent via `Ctrl-C` will kill all processes in the group. The alternative would require using the `ps` command to find all processes that were created to carry out the command.

Every process has a *process group-id* (of type `pid_t`). There is a single process in each group that is considered to be the *leader* of the group. It can be identified easily, because it is the only process whose *process group-id is the same as its process-id.* You can view the process group-id of a process in the output of `ps` by using the `-o` option and specifying the format in either AIX format, such as

```
ps -o'%U %p %P %r %C %x %y %a'
```

or in standard format, as in

```
ps -ouser,pid,ppid,pgrp,%cpu,cputime,tty,args
```

If you run the command

```
$ cat | sort -u | wc
```

and then view the processes using one of the above `ps` commands, you will see the group formation:

```
$ psg -u sweiss | egrep 'TIME|cat|sort|wc'
USER PID PPID PGID %CPU TIME TTY COMMAND
sweiss 17198 17076 17198 0.0 00:00:00 pts/2 cat
sweiss 17199 17076 17198 0.0 00:00:00 pts/2 sort -u
sweiss 17200 17076 17198 0.0 00:00:00 pts/2 wc
```

Notice that the `cat` command's process group-id (`pgid`) is the same as its process-id (`pid`) and that the three processes belong to the same group. If the full listing were displayed you would see that no other process is in this group.

7.4 Foreground and Background Processes

UNIX allows processes to run in the *foreground* or in the *background*. Processes invoked from a shell command line are *foreground processes*, unless they have been explicitly placed into the background by appending an ampersand '`&`' to the command line. There can be only one process group in the foreground at any time, because you cannot enter a new command until the currently running one terminates and the shell prompt returns. Foreground processes can read from and write to the terminal.

In contrast, there is no limit to the number of background processes, but in a POSIX-compliant system, they cannot read from or write to the terminal. If they try to do either, they will be stopped by the kernel (via `SIGTTIN` or `SIGTTOU` signals). The default action of a `SIGTTOU` signal is to stop the process, but many shells override the default action to allow background processes to write to the terminal.

Background and foreground processes use the terminal as their control terminal, but background processes do not receive all signals from that terminal. A `Ctrl-C`, for example, will not cause a `SIGINT` to be sent to background processes. However, a `SIGHUP` will be sent to all processes that use that terminal as their control terminal, including background processes. This is so that, if a terminal connection is broken, all processes can be notified of it and killed by default. If you want to start a background process and then logout from a session, you can use the `nohup` command to run it while ignoring `SIGHUP` signals, as in

```
$ nohup do_backup &
```

which will let `do_backup` run after the terminal is closed. In this case, the `do_backup` program must not read or write a terminal.

7.5 Sessions

Every process belongs to a *session*. More accurately, every process group belongs to a session, and by transitivity, each process belongs to a session. Every session has a unique *session-id* of type `pid_t`. The primary purpose of sessions is to organize processes around their controlling terminals. When a user logs on, the kernel creates a session, places all processes and process groups of that user into the session, and links the session to the terminal as its controlling terminal. Sessions usually consist of a single foreground process group and zero or more background process groups. Just as process groups have leaders, *sessions have leaders.* The session leader can be distinguished because its process-id is the same as the session-id.

Processes may secede from their sessions, and unlike countries, they can do this without causing wars. Any process other than a process group leader can form a new session and automatically be placed into a new group as well. The new session will have no control terminal. This is exactly how a *daemon* is created – it detaches itself from the session into which it was born and goes off on its own. Later we will see how programs can do this.

You can add output to the `ps` command to see the session-id by adding the `"sid"` output format to the standard syntax, as in

```
$ ps -ouser,pid,ppid,pgrp,sid,%cpu,cputime,tty,args
```

7.6 The Memory Architecture of a Process

Although earlier chapters made allusions as to how a process is laid out in virtual memory, here the process layout is described in detail. In addition, we provide a program that displays enough information about the locations of its own symbols in virtual memory that one can infer its layout from them. In particular, the program will display the addresses of various local and global symbols in hexadecimal and decimal notation. These locations are clues to how the process is laid out in its logical address space.

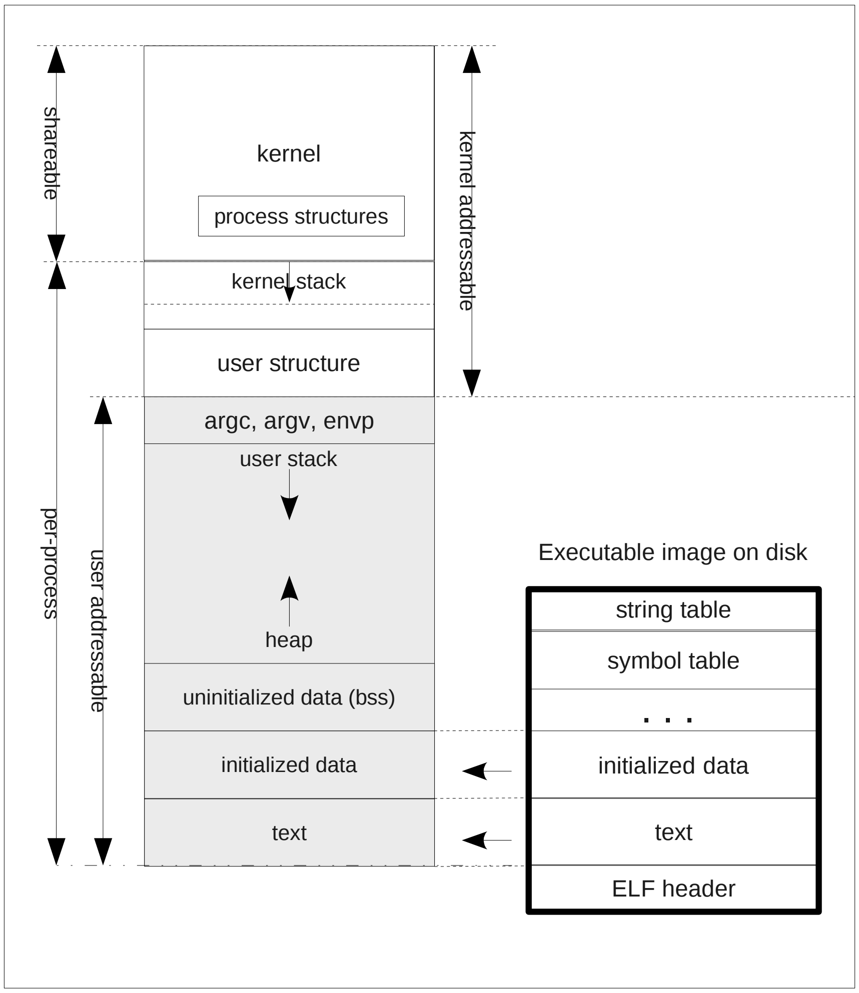

Figure 7.1: Typical layout of a process in virtual memory.

7.6.1 Overview

To start, we look at the big picture of what a process looks like in its logical address space. This picture should enable you to construct a mental image of the physical layout of a process in its own logical address space, how it looks in a file, and how that file relates to the virtual memory image of the process. Different UNIX systems use different layouts for processes, but for the most part, most modern systems adhere to a format known as the *Executable and Linkable Format* (*ELF*).

The resources needed by a process executing in user mode include the CPU state (general purpose registers, program status register, stack related registers, etc.), some environment information, and three memory segments called the *text segment*, the *data segment*, and the *stack segment*. The text segment contains the program code. The stack segment is reserved for the run-time stack of the user phase[1] of the process, and the data segment is for program data. More will be said about these below.

The resources needed when the process is executing in kernel phase include the CPU state (same as above) as well as the resources needed for the kernel to provide the services for the process and schedule it appropriately. These resources include the parameters to the system call, the process's identity and properties, scheduling information, open file descriptors, and so on. This set of kernel resources is separated into two main structures, a *process structure* and a *user structure*, and a few minor ones. Together these structures constitute what is usually called the *process image*. The process structure and user structure are kept in kernel memory. The layout of the process address space is shown in Figure 7.1.

The process structure contains the information that must be memory-resident even when the process is swapped out, including the process privileges and rights, identifiers associated with the process, its memory map, descriptors, pending events, and maximum and current resource utilization. The user structure contains information that is not needed in memory when the process is swapped out, including the *process control block*[2], accounting and statistics, and a few other pieces of information, and is therefore swapped out along with the rest of the process.

7.6.2 The Process Structure

The kernel maintains a process structure for every running process. This structure contains the information that the kernel needs to manage the process, such as various process descriptors (process-id, process group-id, session-id, and so on), a list of open files, a memory map and possibly other process attributes. In all of the versions of UNIX with which I am familiar, this structure is known as the *process structure*. In Linux, the terms process structure and *task structure* are used interchangeably, and the actual data structure that represents it is the `task_struct`. In some versions of UNIX, there is much less information in the process structure and more in the user structure.

Up until the introduction of threads, or so called *light-weight processes* (*LWPs*), the process structure was a very "heavy" structure filled with a large amount of information. One reason that the concept of a light-weight process was invented was to reduce the amount of information associated with each executable unit, so that they did not take up as much memory and so that creating new ones would be faster. The process structure was redesigned in 4.4BSD to support these threads by moving much of the information that had been in it into smaller structures that could be pointed to by the process structure. Each thread could share the information in the substructures by pointing to them, rather than keeping complete copies of them. This way, each thread could have its own unique identifiers, such as a process-id, and also have access to the shared data, such as open files and memory maps.

The exact information present in any process structure will vary from one implementation to another, but all process structures minimally include

- Process id

[1]User phase and user mode are used interchangeably here.

[2]The process control block is used in UNIX only to store the state of the CPU – the contents of the registers and so on.

- Parent process id (or pointer to parent's process structure)
- Pointer to list of children of the process
- Process priority for scheduling, statistics about CPU usage and last priority.
- Process state
- Signal information (signals pending, signal mask, etc.)
- Machine state
- Timers

Usually the process structure is a very large object containing much more additional information. The task structure in Linux 2.6.x, for example, may contain over 150 different members. Typical substructures referenced in the process structure may include such things as the

- Process's group id
- User ids associated with the process
- Memory map for the process (where all segments start, and so on)
- File descriptors
- Accounting information
- Other statistics that are reported such as page faults, etc.
- Signal actions
- Pointer to the user structure.

The substructure generally contains information that all threads would share, and the process structure itself contains thread-specific information.

The process structure is located in kernel memory. Different versions of UNIX store it in different ways. In BSD, the kernel maintains two lists of process structures called the *zombie list* and the *allproc list*. Zombies are processes that have terminated but that cannot be destroyed, the reasons for which will be made clear a little later in this chapter. Zombie processes have their structures on the zombie list. The allproc list contains those that are not zombies. In Linux 2.6.x, the process or task structures are kept in one, circular, doubly-linked list. In Solaris, the process structure is in a `struct proc_t` and the collection of these are maintained in a table.

7.6.3 The User Structure

The user structure contains much less information than the process structure. The user structure gets swapped in and out with the process; keeping it small reduces the swapping overhead. Historically, the most important purpose of the user structure was that it contained the *per-process execution stack* for the kernel.

Every process in UNIX needs its own, small kernel stack. When a process issues a system call and the kernel phase begins, the kernel needs a stack for its function calls. Since the kernel might be interrupted in the middle of servicing a call, it must be able to switch from one process's service call to another. This implies that it needs a different stack for each process. The UNIX kernel designers carefully designed all functions in the kernel so that they are non-recursive and do not use large automatic variables. Furthermore, they can trace the possible sequences of call chains in the kernel so that they know exactly the largest possible stack size. Thus, unlike ordinary user programs, the kernel itself has a known upper bound on its required stack size. Because the stack size is known in advance, the kernel stack can be allocated in a fixed size chunk of virtual address space. This is why, in Figure 7.1, you see that it is bounded above and below by fixed boundaries. As you can see from the figure, the stack is at the high end of the address space, above the environment variables and program parameters. Its exact placement varies from one version of UNIX to another.

Depending on the version of UNIX, the user structure can contain various other pieces of information. In BSD, the memory maps are all in the process structure or its substructures. In Linux, the user structure, defined in `struct user` (in `<user.h>`) contains the memory maps. The memory maps generally include the starting and ending addresses of the text, data, and stack segments, the various base and limit registers for the rest of the address space, and so on1. The user structure usually contains the process control block. This contains the CPU state and virtual memory state (page table base registers and so on.)

7.6.4 The Text Segment

The text segment (also called the *instruction segment*) is the program's executable code. It is almost always a sharable, read-only segment, shared by all other processes executing the same program. The C compiler, by default, creates shared text segments. The advantage of using shared text segments is not so much that it conserves memory to do so, but that it reduces the overhead of swapping. When a process is active, its text segment resides in primary memory. There are various reasons why the process might be swapped to secondary storage. Often it is that it issues a wait for a slow event; in this case the system will swap it to secondary storage to make room in memory for other more productive processes. When it becomes active again, it is brought back into memory. Since a read-only text segment can never be modified, there is no reason to copy it to secondary storage when a process executing it is swapped out. Similarly, if a copy of a text segment already resides in primary memory, there is no reason to copy the text segment from secondary storage into primary memory. Thus, there is a savings in swapping overhead.

UNIX keeps track of the read-only text segment of each user process. It records the location of the segment in secondary storage and, if it is loaded, its primary memory address, and a count of the number of processes that are currently executing it. When a process first executes the segment, the segment is loaded from secondary storage, the count is set to one, and a table entry is created. When a process terminates, the count is decremented. When the count reaches zero, the segment is freed and its primary and secondary memory are de-allocated.

7.6.5 The Stack Segment

The user stack segment serves as the run-time stack for the user phase of the process. That is, when the process makes calls to other parts of the user code, the calls are stacked in this segment. The stack segment provides storage for the automatic identifiers and register variables, and serves its usual role of managing the linkage of subroutines called by the user process. The stack is always "upside-down" in UNIX, meaning that pushes cause the top to become a smaller memory address. If the stack ever meets the top of the heap, it causes an exception. In a 32-bit architecture, a user process is typically allocated a virtual address space of 4 Gbytes. If the stack meets the heap, the process exceeds its virtual memory allotment and it is time to port the application to a 64-bit machine!

7.6.6 The Data Segment

The data segment is memory allocated for the program's initialized and uninitialized data. The initialized data is separated from the uninitialized data, which is stored in a section called the *bss*, which is an acronym for *Block Started by Symbol*, an old FORTRAN machine instruction. Initialized data are items such as named constants and initialized static variables. They come from the symbol table. Uninitialized data has no starting value. The system only needs to reserve the space for them, which it does by setting the address of the top of the data segment. The data segment grows or shrinks by explicit memory requests to shift its boundary. The system call to shift the boundary is the `brk()` call; `sbrk()` is a C wrapper function around `brk()`.The data segment always grows toward the high end of memory, i.e., the `brk()` call increases the boundary to increase memory. Most programmers use the C library functions `malloc()` and `calloc()` to allocate more memory instead of the low-level `brk()` primitives. These C routines call `brk()` to adjust the size of the data segment.

Programmers often refer to the part of memory that is used by `malloc()` and `calloc()`, the "*heap*". People usually think of the heap as the part of memory that is not yet allocated, lying between the top of the stack and the end of the bss.

7.6.7 From File to Memory

How is an executable program file arranged and how does it get loaded? When you run the `gcc` compiler and do not specifically name the executable file, as in

```
$ gcc myprog.c
```

the compiler (actually the linker) creates a file named `a.out`. The name `a.out` is short for "assembler output" and was not just the name of the output file, but was also name of the format of all binary executable files on UNIX systems for many years. The `a.out` file format could be read in the `a.out` man page as well as in the `<a.out>` header file.

In the mid 1990's, a more portable and extensible format known as *Executable and Linking Format* (*ELF*) was created by the UNIX System Laboratories as part of the *Application Binary Interface* (*ABI*). It was later revised by the Tool Interface Standards (TIS) Committee, an industry consortium that included most major companies (Intel, IBM, Microsoft, and so on). While compilers continue to create files named `a.out`, they are ELF files on most modern machines.

The ELF specification defines exactly how an executable file is organized, and is general enough to encompass three different types of files:

- *Relocatable files* holding code and data suitable for linking with other object files, to create an executable or a shared object file (`.o` files),

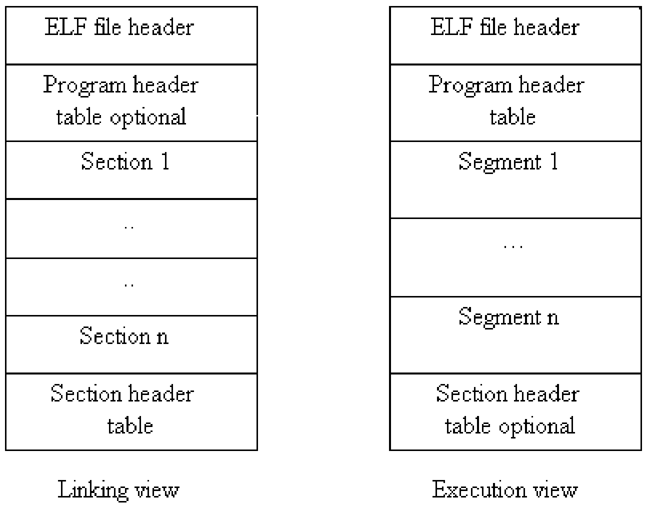

Figure 7.2: Linking and execution views of an ELF file.

- *Executable files* holding a program suitable for execution, and
- *Shared object files* that can be used by the dynamic linker to create a process image (`.so` files).

An ELF file begins with a structure called the *ELF header.* This header is essentially a road map to the rest of the file. It contains identification information and the addresses and sizes of the rest of the components of the file. An ELF file is characterized by the fact that it contains two different, parallel, yet overlapping views: the *linking view* and the *execution view* as shown in Figure 7.2.

The linking view is the view of the file needed by the link editor in order to link and relocate components in the file. Information within it is organized into *sections*, which contain such things as the instructions, data, symbol table, string table, and relocation information. A *section header table* serves as a table of contents for the sections and is generally located at one end of the file.

The execution view, in contrast, is the view of the file needed in order to execute it. It organizes its information in *segments*. Segments correspond conceptually to virtual memory segments; when the executable is loaded into memory, ELF segments are mapped to virtual memory segments. Thus, for example, for an executable program, there is a text segment containing instructions, an uninitialized data segment, and an initialized data segment, as well as several others. A *program header table* serves as a table of contents for the segments and usually follows the ELF header.

Neither the sections nor the segments in the file have to be in any particular order because the tables define their positions. The two separate views are overlapped in the file, as shown in Figure 7.3. The figure also shows that segments can consist of multiple sections.

The *symbol table* is a table that the compiler creates to map symbolic names to logical addresses and store the attributes of these symbols. The compiler uses the table to construct the code, but in addition, it is the symbol table that makes it possible for a debugger to associate memory addresses with the names of variables. When the debugger runs, the symbol table is loaded into memory with the program. The *string table* is a table containing all of the strings used in the program. The `strings` command is a handy command to know about – it displays a list of all of the strings in a binary file. The strings command works because the string table is part of the file and the command simple has examine it.

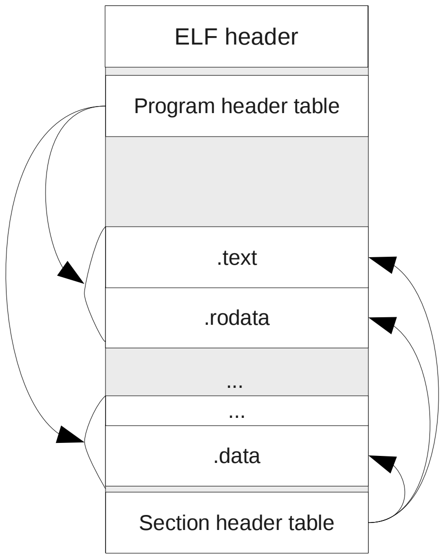

Figure 7.3: Overlapped views of the ELF file.

The `readelf` command can be used to examine an executable file.

```
$ readelf [options] elffile ...
```

The information displayed depends upon the options provided on the command line. With `-a`, all information is provided. The `-h` option displays the contents of the ELF header (in human readable form of course):

```
$ readelf -h /bin/bash
ELF Header:
Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Class: ELF32
Data: 2's complement, little endian
Version: 1 (current)
OS/ABI: UNIX - System V
ABI Version: 0
Type: EXEC (Executable file)
Machine: Intel 80386
Version: 0x1
Entry point address: 0x805c7b0
Start of program headers: 52 (bytes into file)
Start of section headers: 733864 (bytes into file)
Flags: 0x0
Size of this header: 52 (bytes)
Size of program headers: 32 (bytes)
```

```
Number of program headers: 8
Size of section headers: 40 (bytes)
Number of section headers: 32
Section header string table index: 31
```

You can see from this output that the ELF header has information about the executable's format: 32-bit, 2's complement, for Intel 80386 on UNIX System V. It also has the location of the program header (byte 52 into the file) and the section header table (733864 bytes into the file).

The `file` command uses the ELF header to provide its output:

```
$ file /bin/bash
/bin/bash: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.9, stripped
```

You can experiment with some of the executables to see how much you can learn about the structure of executable programs.

Notice in the output above that the entry point address of the executable is `0x805c7b0`. This is the address of the first executable instruction in the bash executable. The starting virtual address is not `0`. In modern UNIX systems, the starting address is always after `0x8048000`. The addresses below that are reserved for the system to use. One reason for this is that the debugger will run in the lower addresses when the program runs under the debugger.

Figure 7.4 shows the virtual addresses of a hypothetical executable, taken from a version of the ELF standard. Notice in this example that the text segment is not at the start of the virtual address space, and that each segment is padded as needed so that it aligns on `0x1000` (4096) byte boundaries, because page sizes are 4096 bytes. Notice too that the data segment and uninitialized data segment follow the text segment.

7.6.8 A Program To Display The Virtual Memory Boundaries

In this section we will explore the virtual address space of a process with the aid of a program that I found in a book on interprocess communication [1] and subsequently modified. It displays the boundaries of the different components of the virtual address space of its executable image. The program declares the following types of memory objects:

- Global, initialized pointer variable: `cptr`
- Global uninitialized string: `buffer1`
- Automatic variable in main program: `i`
- Parameters to main program: `argc`, `argv`, `envp`
- Static uninitialized local in main program: `diff`
- Main function `main()`
- Non-main function `showit()`

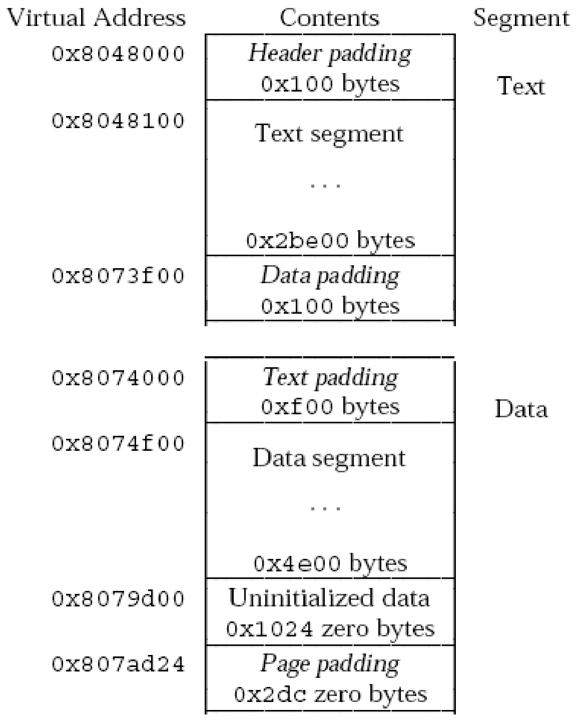

Figure 7.4: Example of ELF process image segments.

- Automatic pointer variable, dynamically allocated: `buffer2`
- Automatic variable in non-main function: `num`

It displays the locations of each of these objects as hexadecimal and decimal virtual addresses. The locations of these objects will lie within the ranges that are specified by the text, data, and stack segment positions described earlier. For example, the location of the uninitialized global `buffer1` and the initialized global `*cptr` will be between the first and last addresses of the data segment, one in the `bss` and the other, not. The remaining variables each pinpoint the location of a particular segment: `"Hello World"`, the pointer variable `cptr` itself, the symbols `main()` and `showit`(), which are globals, and the local variables `num` and `buffer2` in `showit`(). The location of a local variable, which is supposed to be on the run-time stack, will show where that stack is in virtual memory.

Three external integer variables `etext`, `edata`, and `end` are defined in global scope in the C library and may be accessed by any C program[3]. They are boundaries of three specific segments:

`etext`	The address of `etext` is the first location after the program text.
`edata`	The address of `edata` is the first location after the initialized data region.
`end`	The address of `end` is the first location after the uninitialized data region.

The value of `etext` is an upper bound on the size of the executable image. The initialized and uninitialized data regions are the regions where constants and globals are stored. Uninitialized data

[3]They may be declared as macros.

are globals whereas initialized data are constants. These symbols are incorporated into the program, displayed in Listing 7.1.

Listing 7.1: displayvm.c

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

/*********************************************************************/
/*                          Global Constants                         */
/*********************************************************************/

#define SHW_ADR(ID,I) \
printf("      %s \t is at addr:%8X\t%20u\n",\
         ID,(unsigned int)(&I),(unsigned int)(&I))

/* These are system variables, defined in the unistd.h header file */
extern int    etext, edata, end;

char     *cptr = "Hello World\n";  /* cptr is an initialized global */
char     buffer1[40];              /* uninitialized  global         */

void showit(char *);         /* Function prototype -- has no storage */

int main(int argc, char* argv[], char* envp[])
{
    int  i = 0;          /* on stack */
    static int  diff;    /* global in uninitialized data segment */

    strcpy(buffer1, "    Layout of virtual memory\n");
    write(1,buffer1,strlen(buffer1)+1);

    printf("Adr etext: %8X \t Adr edata: %8X \t Adr end: %8X \n\n",
           (unsigned int)&etext,(unsigned int) &edata,(unsigned int) &end);

    printf("        ID \t             HEX_ADDR\t         DECIMAL_ADDR\n");
    SHW_ADR("main",  main);
    SHW_ADR("showit", showit);
    SHW_ADR("etext", etext);
    diff = (int) &showit - (int) &main;
    printf("           showit is %d bytes above main\n", diff);
    SHW_ADR("cptr", cptr);
    diff = (int) &cptr - (int) &showit;
    printf("           cptr is %d bytes above showit\n", diff);
    SHW_ADR("buffer1", buffer1);
    SHW_ADR("diff", diff);
    SHW_ADR("edata", edata);
    SHW_ADR("end", end);
    SHW_ADR("argc", argc);
    SHW_ADR("argv", argv);
    SHW_ADR("envp", envp);
    SHW_ADR("i", i);
    showit(cptr);
    return 0;
}
/*********************************************************************/

void showit(char * p)
{
    char  *buffer2;
    SHW_ADR("buffer2",buffer2);
    if ( (buffer2= (char *) malloc((unsigned)(strlen(p)+1))) != NULL){
        strcpy(buffer2, p);
        printf("%s", buffer2);
        free(buffer2);
    }
    else {
        printf("Allocation error.\n");
        exit(1);
    }
}
```

When you run this program, you should get output similar to the following. The last column is the decimal value of the location. Notice that `i` in `main()` and `buffer2` in `showit()` are high addresses. They are in the stack. Notice that `envp`, which is a pointer to an array of strings, is above the stack, as Figure 7.1 depicts. Notice that the addresses of these descend, because the stack grows downward. Notice too that `diff` in `main()` and `buffer1` lie between `edata` and `end`, showing that they are in the uninitialized data region, but that `cptr` is between `etext` and `edata` because it is initialized. You should see that `buffer1` is the last variable in the *bss*, since it is 40 bytes long and its address is 40 bytes from the end. Also, notice that the address of `etext` is the same as the value of `etext`. This is because `etext` is not a real variable. It is just a mnemonic name for the actual location, as determined by the linker.

```
     Layout of virtual memory
Adr etext:  8048958      Adr edata:  8049C90      Adr end:  8049CE8

        ID                  HEX_ADDR            DECIMAL_ADDR
       main        is at addr: 8048544             134513988
       showit      is at addr: 8048806             134514694
       etext       is at addr: 8048958             134515032
           showit is 706 bytes above main
       cptr        is at addr: 8049C8C             134519948
           cptr is 5254 bytes above showit
       buffer1     is at addr: 8049CC0             134520000
       diff        is at addr: 8049CA8             134519976
       edata       is at addr: 8049C90             134519952
       end         is at addr: 8049CE8             134520040
       argc        is at addr:BF9ECDC0            3214855616
       argv        is at addr:BF9ECDC4            3214855620
       envp        is at addr:BF9ECDC8            3214855624
       i           is at addr:BF9ECD9C            3214855580
       buffer2     is at addr:BF9ECD6C            3214855532
Hello World
```

Notice that the main program starts at virtual address `0x08048544`. The start of the virtual address space is at `0x08048000`. The difference is 1348 bytes. If you look at the output of the `readelf`

`-a` command, you will see that there are several sections of code that reside in this small pocket before the main program. There you will find the code that must be run before `main()` starts (what I like to call the "glue" routine), the code that runs when the program finishes, and special code for dynamic linking and relocation. When a program starts, before the operating system transfers control to the program, there are initializations and possible linking and relocation; the code there serves that purpose.

7.7 Creating New Processes Using fork

Now that you have a better idea of what processes actually are, we can start exploring their world. We will begin with process creation, since that is, after all, the beginning of all things.

Once the operating system has bootstrapped itself, the only way for any process to be created is via the `fork()` system call[4]. All processes are created with `fork()`:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

The `fork()` call is a hard one to accept at first; you probably have never seen a function quite like it before. It is very appropriately named, because the statement

```
pid_t processid = fork();
```

causes the kernel to create a new process that is almost an exact copy of the calling process, such that after the call, there are two processes, each continuing its execution at the point immediately after the call in the executing program! So before the instruction is executed, there is a single process about to execute the instruction, and by the time it has returned, there are two. There has been a fork in the stream of instructions, just like a fork in a road. It is almost like process mitosis.

The process that calls `fork()` is called the *parent process* and the new one is the *child process*. They are distinguished by the values returned by `fork()`. Each process is able to identify itself by the return value. The value returned to the parent is the process-id of the newly created child process, which is never 0, whereas the value returned to the child is 0. Therefore, each process simply needs to test whether the return value is 0 to know who it is. The child process is given an independent *copy* of the memory image of the parent and the same set of resources. It is essentially a clone of the parent and is almost identical in every respect[5]. Unlike cellular mitosis though, it is not symmetric.

The typical use of the `fork()` call follows the pattern

```
processid = fork();
```

[4]This is a simplification. There are other system calls, depending on the version of UNIX. For example, Linux revived the old BSD `vfork()` system call, which was introduced in 3.0BSD and later removed in 4.4BSD for good reason. Linux also provides a `clone()` library function built on top of the kernel's `sys_clone()` function. Section 7.7.1 below.

[5]There are a few minor differences. For example, a call to `getppid()` will return different values, since this returns the process-id of the parent process.

```
if (processid == 0)
// child's code here
else
// parent's code here
```

The true branch of the if-statement is code executed by the child and not by the parent. The false branch is just the opposite. This may seem like a useless way to create processes, since they always have to share the same code as the parent, so they do nothing different. The `fork()` call is valuable when used with `exec()` and `wait()`, as will be shown shortly.

Our first example, `forkdemo1.c`, in Listing 7.2 demonstrates a bit about how `fork()` works.

Listing 7.2: forkdemo1.c

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int global = 10;

int main(int argc, char* argv[])
{
    int    local = 0;
    pid_t  pid;

    printf("Parent process: (Before fork())");
    printf(" local = %d, global = %d \n", local, global);

    if ( ( pid = fork() ) == -1 ) {
        perror("fork");
        exit(1);
    }
    else if ( 0 == pid ) {
        /* child executes this branch */
        printf("After the fork in the child:");
        local++;
        global++;
        printf(" local = %d, global = %d\n", local, global);
    }
    else {
        /* parent executes this branch */
        sleep(2);  /* sleep long enough for child's output to appear */
    }

    /* both processes execute this print statement */
    printf("pid = %d, local = %d, global = %d \n",
            getpid(), local, global);

    return 0;
}
```

In this program, the return value from `fork()` is stored in `pid`, which is tested in the if-statement. `fork()` returns `-1` on failure, and one should always check whether it failed or not.

The child's code is the next branch, in which `0 == pid` is true. The child increments the values of two variables, one, `global`, declared globally and one, `local`, locally in `main()`. It then prints out their values, jumps over the parent code and executes the `printf()`, obtaining its process-id using the `getpid()` system call and displaying the values of `local` and `global` again.

In the meanwhile, the parent sleeps for two seconds, enough time so that the child's output will appear first. It then executes the `printf()`, obtaining its process-id using the `getpid()` system call and displaying the values of `local` and `global`. The `sleep()` prevents intermingling of the output, which can happen because the child shares the terminal with the parent.

Bearing in mind that the child is given a *copy* of the memory image of the parent, what should the output of the parent be? Hopefully you said that `local = 0` and `global = 10`. This is because the child changed their values in the copy of the memory image, not in a shared memory. The point of this program is simply to demonstrate this important fact.

Let us make sure we understand `fork()` arithmetic before continuing. Before running the program `forkdemo2.c` shown in Listing 7.3, predict the number of lines of output. Do not redirect the output or pipe it. This will be explained afterward.

Listing 7.3: forkdemo2.c

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char* argv[] )
{
    int i;
    printf("About to create many processes...\n");
    for ( i = 0; i < N; i++ )
        if ( -1 == fork() )
            exit(1);

    printf("Process id = %d\n", getpid());
    fflush(stdout);      /* force output before shell prompt        */
    sleep(1);            /* give time to the shell to diplay prompt */
    return 0;
}
```

Did you correctly predict?

Each time the bottom of the loop is reached, the number of processes in existence is twice what it was before the loop was entered, because each existing process executes the `fork()` call, making a copy of itself. If N were 1, the loop would execute once and there would be 2 processes, each printing their process ids to the screen. If N were 2, the loop would execute a second time, and the 2 processes would make 2 more, 4 in total. In general, there will be 2^N processes when the loop finishes, and that many lines of output on the screen, together with the first line,

```
About to create many processes...
```

for a total, when $N = 4$, of 17 lines. Now try redirecting the output of the program to `wc`, to make it easier to count how many lines are there.

```
$ forkdemo2 | wc
     32     144     832
$
```

Why are there 32 lines and not 17? Try instead redirecting the output to a file and looking at it there:

```
$ forkdemo2 > temp; wc temp
 24 108 624 temp
$
```

If you look at the file `temp`, you will see something like

```
About to create many processes...
Process id = 6708
About to create many processes...
Process id = 6707
About to create many processes...
Process id = 6717
About to create many processes...
Process id = 6709
About to create many processes...
Process id = 6716
...
```

and there will be fewer than 16 lines with process ids. So what is going on here? Why is that line replicated for each process and why are there fewer than 16 lines stating the process ids?

- Remember this important fact about `fork()`: when a process is created by a call to `fork()`, it is an almost exact duplicate of the original process. In particular it gets copies of all open file descriptors and naturally, all of the process's user space memory image.
- What this means is that when a child process is created, its standard output descriptor points to the same open file structure as the parent and all other processes forked by the parent, and therefore the children and parent share the file position pointer.
- Operations such as `printf()` are part of the C I/O library and act on objects of type `FILE`, which are called *streams.* The C I/O Library uses *stream buffering* for all operations that act of `FILE` streams. (We noted this in Chapter 5.) There are three different kinds of buffering strategies:
  - *Unbuffered streams*: Characters written to or read from an unbuffered stream are transmitted individually to or from the file as soon as possible.
  - *Line buffered streams*: Characters written to a line buffered stream are transmitted to the file in blocks when a newline character is found.
  - *Fully buffered streams*: Characters written to or read from a fully buffered stream are transmitted to or from the file in blocks of arbitrary size.

- By default, when a stream is opened, it is fully buffered, except for streams connected to terminal devices, which are line buffered.
- The buffers created by the C I/O Library are in the process's own address space, not the kernel's address space. (When your program calls a function such as `printf()`, the library is linked into that program; all memory that it uses is in its virtual memory.) This means that when `fork()` is called, the child gets a copy of the parent's library buffers, and all children get copies of these buffers. *They are not shared*; *they are duplicated.*
- The C I/O library flushes all output buffers
  - When the process tries to do output and the output buffer is full.
  - When the stream is closed.
  - When the process terminates by calling `exit()`.
  - When a newline is written, if the stream is line buffered.
  - Whenever an input operation on any stream actually reads data from its file.
  - When `fflush()` is called on that buffer.
- As a corollary to the preceding statement, until the buffer has been flushed, it contains all characters that were written to it since the last time it was flushed.
- No C I/O Library function is atomic. It is entirely possible that output can be intermingled or even lost if the timing of calls by separate processes sharing a file position pointer leads to this.

Now we put these facts together. The `forkdemo2` program begins with the instruction

```
printf("About to create many processes...\n");
```

If output has not been redirected, then `stdout` is pointed to a terminal device and it is line buffered. The string `"About to create many processes...\n"` is written to the terminal and removed from the buffer. When the process forks the children, they get empty buffers and write their individual messages to the terminal. Unless by poor timing a line is written over by another process, each process will produce exactly one line of output. It is quite possible that this will happen if there are a large enough number, N, of processes, as the probability of simultaneous writes increases rapidly towards 1.0 as N increases.

Let us do a bit of mathematical modeling. The `printf()` instruction

```
printf("Process id = %d\n", getpid());
```

writes to standard output. If the fraction of time that each process spends in the portion of the `printf()` function in which a race condition might occur is p, then there is a probability of $1-(1-p)^N$ that at least two processes are in that portion of code at the same time. If, for example, $p = 0.05$ and $N = 16$, then the probability of a race (and hence lost output) is $1 - 0.95^16 \approxeq 0.56$. If $N = 32$, it is 0.81 and when $N = 64$, it is 0.96. So you see that as the number of processes increases, it becomes almost inevitable that lines will be lost, regardless of whether they are written to the terminal or to a different file descriptor, because the race condition is independent of how the output stream is buffered.

If standard output is redirected to a file or to a pipe, it no longer points to a terminal device and the library will fully buffer it instead of line buffering it. The block size used for buffering is much larger than the total size of the strings given to the `printf()` function. The consequence is that the string `"About to create many processes...\n"` will remain in the buffers of all child processes when they are forked, and when they each call

```
printf("Process id = %d\n", getpid());
fflush(stdout);
```

each line of the output will be of the form

```
About to create many processes...
Process id = 8810
```

and there will be twice as many lines *written* as there were to the terminal.

Since the command

```
forkdemo2 | wc
```

redirects the standard output of `forkdemo2` to a pipe, `wc` will see twice as many lines as appear on the terminal. Similarly, the command

```
forkdemo2 > temp
```

redirects the standard output to a file, and the file will contain twice as many lines as what appears on the terminal.

The foregoing statement about the output to the file is true *only if* the executions of the `printf()` instructions do not overlap and no output is lost. We return to this issue shortly. The claim regarding the pipe is unconditionally true.

How can we make the behavior of the program the same regardless of whether it is to a terminal or is redirected? We can force the first string to be flushed from the buffer by calling `fflush(stdout)`. Since there is no need to do this if it is a terminal, we can insert the two lines

```
if ( !isatty(fileno(stdout)) );
fflush(stdout);
```

just after the first `printf()`.

What about the problem of lost output? How can we prevent this race condition? The answer is that we must not use the stream library but must use the lower level `write()` system call and file descriptors. Writes are unbuffered and we can set the `O_APPEND` flag on file descriptor 1 so that the race condition is eliminated. (Recall from Chapter 4 that this is how writes to the `utmp` file avoid race conditions.)

To use `write()`, we must first create the output string using the `sprintf()` function:

```
char str[32];
sprintf(str, "Process id = %d\n", getpid());
```

Then we can call `write()`:

```
write(1, str, strlen(str));
```

But first we must start the program by setting `O_APPEND` on standard output's descriptor:

```
int flags;
flags = fcntl(1, F_GETFL);
flags |= (O_APPEND);
if (fcntl( 1, F_SETFL,flags) == -1 )
    exit(1);
```

This solves the problems. The corrected program is listed below.

Listing 7.4: forkdemo3.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include   <fcntl.h>
#include <termios.h>

int main( int argc, char* argv[] )
{
    int     i;
    int     N;
    char    str[32];
    int     flags;

    /* Put standard output into atomic append mode */
    flags = fcntl(1, F_GETFL);
    flags |= (O_APPEND);
    if (fcntl( 1, F_SETFL,flags) == -1 )
        exit(1);

    /* Get the command line value and convert to an int.
       If none, use default of 4. If invalid, exit. */
   N = ( argc > 1 )? atoi(argv[1]):4;
    if ( 0 == N )
        exit(1);

    /* Print a message and flush it if this is not a terminal */
    printf("About to create many processes...\n");
    if ( !isatty(fileno(stdout)) )
        fflush(stdout);
    /* Now fork the child processes. Check return values and exit
       if we have a problem. Note that the exit() may be executed
       only for some children and not others. */
    for ( i = 0; i < N; i++ )
        if ( -1 == fork() )
            exit(1);

    /* Create the output string that the process wil/ write, and write using
       system call. */
    sprintf(str, "Process id = %d\n", getpid());
    write(1, str, strlen(str));
    fflush(stdout);         /* to force output before shell prompt */
    sleep(1);               /* to give time to the shell to diplay prompt */
    return 0;
}
```

7.7.1 Other Versions of fork()

The `vfork()` system call is a different version of the `fork()` call that is designed to be more efficient. Rather than making a complete copy of the address space of the old process, the `vfork()` call creates a new process without copying the data and stack segments of the parent and instead allows the child process to share these. This saves time and memory but also raises the possibility that the child will inadvertently corrupt the state of the parent process. It is not intended to be used to allow the child and parent to share data; on the contrary, its purpose is to avoid the extensive memory copying in the case that the child will replace its code anyway using `exec()` (to be discussed soon.)

There is also a `clone()` system call in Linux systems. The `clone()` function, which is technically a library routine wrapping a system call, allows the child to share the address space with its parent, and also lets the programmer pass a function and arguments for the child to execute. We will look at it in detail later.

7.7.2 Synchronizing Processes with Signals

The next program, `synchdemo1.c`, demonstrates how to use `fork()` and signals to synchronize a child and its parent.

Listing 7.5: synchdemo1.c

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

void  c_action(int signum)
{
    /* nothing to do here */
}

/*****************************************************************/
```

```
int main(int argc, char* argv[])
{
    pid_t  pid;
    int    status;
    static struct sigaction childAct;


    switch (pid = fork())  {
    case -1:
        /* fork failed! */
        perror("fork() failed!");
        exit(1);

    case 0: {
        /* child executes this branch   */
        /* set SIGUSR1 action for child */
        int i, x=1;
        childAct.sa_handler = c_action;
        sigaction(SIGUSR1, &childAct, NULL);
        pause();
        printf("Child  process: starting computation...\n");
        for ( i = 0; i < 10; i++ ) {
            printf("2^%d = %d\n", i, x);
            x = 2*x;
        }
        exit(0);
    }
    default:
        /* parent code */
        printf("Parent process: "
               "Will wait 2 seconds to prove child waits.\n");
        sleep(2); /* to prove that child waits for signal */
        printf("Parent process: "
               "Sending child notice to start computation.\n");
        kill(pid,SIGUSR1);

        /* parent waits for child to return here */
        if ((pid = wait(&status)) == -1)
        {
            perror("wait failed");
            exit(2);
        }
        printf("Parent process: child terminated.\n");
        exit(0);
    }
}
```

Comments.

- First note that the style of this program is slightly different. It uses the `switch` statement to distinguish the failed `fork()`, child, and parent.
- The `SIGUSR1` signal is a signal value that is reserved for user programs to use as they choose.

In this program, we can use it to synchronize two processes. One process delays itself using `pause()`, and waits for a signal to arrive. The second sends the signal to wake up the first. The signal handler does not have to do anything special in this case.

- The parent calls `wait()`, a function we will explore shortly. The `wait()` call makes the parent wait until the child terminates or is killed.
- The program displays output on the terminal just to demonstrate how the signaling works.

The next demo, `synchdemo2.c`, is a little more interesting than the preceding one. It demonstrates how the parent and child can work in lockstep using the `SIGUSR1` signal. It also shows that the child process inherits the open files of the parent, and that writes by the child and parent to the same descriptor advance the shared file position pointer.

Listing 7.6: synchdemo2.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*****************************************************************/
void p_action(int sig);
void c_action(int sig);
void on_sigint(int sig);
/*****************************************************************/

int                    nSignals = 0;
volatile sig_atomic_t  sigint_received = 0;

/*****************************************************************/
int main(int argc, char* argv[])
{
    pid_t pid, ppid;

    static struct  sigaction parentAct, childAct;
    int            fd;
    int            counter = 0;
    char           childbuf[40];
    char           parentbuf[40];

    if ( argc < 2 ) {
        printf("usage: synchdemo2 filename\n");
        exit(1);
    }

    if ( -1 == (fd = open(argv[1], O_CREAT|O_WRONLY|O_TRUNC, 0644 )) )
    {
        perror(argv[1]);
        exit(1);
    }
    switch( pid = fork() ) {
    case -1:
        perror("failed");
        exit(1);
    case 0:
        /* set action for child */
        childAct.sa_handler = c_action;
        sigaction(SIGUSR1, &childAct, NULL);
        ppid = getppid();  /* get parent id */
        for(;;) {
            sprintf(childbuf, "Child counter = %d\n", counter++);
            write(fd, childbuf, strlen(childbuf) );
            printf("Sending signal to parent -- ");
            fflush(stdout);
            kill(ppid, SIGUSR1);
            sleep(3);
        }

    default:
        /* set SIGUSR1 action for parent */
        parentAct.sa_handler = p_action;
        sigaction(SIGUSR1, &parentAct, NULL);

        /* set SIGINT handler for parent */
        parentAct.sa_handler = on_sigint;
        sigaction(SIGINT, &parentAct, NULL);

        for(;;) {
            sleep(3);
            sprintf(parentbuf, "Parent counter = %d\n", counter++);
            write(fd, parentbuf, strlen(parentbuf) );
            printf("Sending signal to child  -- ");
            fflush(stdout);
            kill(pid, SIGUSR1);
            if ( sigint_received ) {
                close(fd);
                exit(0);
            }
        }
    }
}

/**********************************************************************/

void p_action(int sig)
{
    printf("Parent caught signal %d\n", ++nSignals);
}

void c_action(int sig)
{
    printf("Child caught signal %d\n", ++nSignals);
}
```

```
void on_sigint( int sig )
{
    sigint_received = 1;
}
```

Comments.

- Writes to the file are in lockstep and there is no race condition because of the arrangement of the `sleep()` and `kill()` calls in the child and parent. The parent writes after it is awakened from its `sleep()` and before it signals the child, whereas the child writes before it signals the parent the first time, and then after it is awakened by the parent. If the parent is writing, the child must be sleeping, and vice verse.
- The use of the `sleep()` instead of `pause()` prevents deadlock. Had we used `pause()`, then there would be a very small but nonzero probability that one process could issue a `kill()` to the other and, before it then executes its `pause()`, the other is woken up, executes all of its code and issues a `kill()` to the first one. In this case, the signal would be lost, because it happened before the pause(). That process would then be blocked waiting for a second signal to wake it, but the other process will enter its `pause()` and never be able to send that signal. They are thus deadlocked. The `sleep()` will eventually terminate, so nether process will wait indefinitely.
- The call to `fflush()` is needed to force writes to the screen by each process to happen immediately, otherwise they will occur in the wrong order.
- The main program has a `SIGINT` handler so that the program can clean up after itself. When `Ctrl-C` is typed, both the parent and the child will receive it. The parent closes the open file descriptor before exiting, and the child is automatically killed.

We now turn to the question of how a process can change the code it executes.

7.8 Executing Programs: The exec family

Being able to create a new process is not so useful unless that new process has a way to execute a different program. The `exec()` family of calls fulfills that purpose. All versions of the `exec()` call have one thing in common – they cause the calling process to execute a program named in one way or another in the argument list, and they all are library wrappers for the `execve()` system call.

7.8.1 The execve() System Call

The man page for the `execve()` system call defines it as follows:

```
#include <unistd.h>
int execve(const char *filename, char *const argv[],
char *const envp[]);
```

`execve()` executes the program pointed to by its first argument. The filename must be a binary executable or a script whose first line is

```
#! interpreter [optional-arg]
```

The filename must be the absolute pathname or relative pathname of the program. `execve()` does not look at the `PATH` environment variable to resolve command names. The second and third arguments are `NULL`-terminated arrays of arguments and environment strings respectively. In other words, each is an array of strings followed by a `NULL` pointer. The environment strings are expected to be in the proper format: *key=value*.

You should remember that arrays are sometimes called vectors by computer scientists, and that the reason that the name of this system call is `execve` is that it expects vectors as its second and third arguments. In particular, you need to remember that `execve()` will pass these vectors to the program being executed, which will be able to access them in its own argument list:

```
int main( int argc, char* argv[], char* envp[])
```

Since all programs expect the program name in `argv[0]` and the first real argument in `argv[1]`, it is important that you arrange the argument list to satisfy this condition before you invoke `execve()`. The examples will demonstrate.

The man page provides all of the details about the call. In general, `execve()` causes the stack, data segment, bss, and text segment to be replaced, and pretty much clears all signals and closes anything the process had open before the call except for file descriptors, which remain open. The current working directory remains the same. Because the same process continues to execute, process relationships are also preserved. For the details consult the man page.

To start we will look at how to use the `execve()` system call, after which we will look at the different wrappers for it. The first program, `execdemo1.c`, is in Listing 7.7 below.

Listing 7.7: execdemo1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char * argv[], char * envp[])
{
    if ( argc < 2 ) {
        printf("usage: execdemo1 arg1 [arg2 ...]\n");
        exit(1);
    }
    execve("/bin/echo", argv, envp);
    fprintf(stderr, "execve() failed to run.\n");
    exit (1);
}
```

This program calls `execve()`, passing `/bin/echo` as the program to run, followed by its own command line arguments and environment strings. These are passed to `/bin/echo`. This program works correctly even though `argv[0]` contains the string `"execdemo1"` and not `"echo"` because `echo` pretty much ignores `argv[0]` and only starts paying attention to arguments starting with `argv[1]`. This is not the best way to use `execve()` – it only works in a few circumstances.

Are you wondering about the `printf()` after the call? The `printf()` statement will only be executed if the `execve()` call fails; the only reason that `execve()` returns is failure to execute the program.

The next program, `execvedemo2.c`, uses `execve()` to execute the first command line argument of the program, passing to it the remaining arguments from the command line. In other words, if we supply a line like

```
$ execvedemo2 /bin/ls -l ..
```

it will execute it as if you typed the `/bin/ls -l ..` on a line by itself.

Listing 7.8: execvedemo2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char * argv[], char * envp[])
{
    if (argc < 2) {
        fprintf(stderr, "Usage: %s program args\n", argv[0]);
        exit (1);
    }
    execve(argv[1], argv+1, envp);
    fprintf(stderr,"execve() failed to run.\n");
    exit(1);
}
```

Notice that it uses pointer arithmetic to pass the array `argv[1 .. argc-1]` rather than `argv[0 .. argc-1]`.

7.8.2 The exec() Library Functions

Because it is a little inconvenient to arrange everything for `execve()`, the designers of UNIX created a family of five functions that act as front-ends to `execve()`, each expecting a different set of parameters. They are

```
#include <unistd.h>
extern char **environ;

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg,..., char* const envp[]);
int execv(const char *path, char * const argv[]);
int execvp(const char *file, char * const argv[]);
```

Each of these contains either an 'l' or a 'v' in its name. The versions that contain an 'l', `execl()`, `execlp()` and `execle()`, expect a null-terminated *list* of null-terminated string arguments whereas the versions that contain a 'v', `execv()` and `execvp()`, expect a *vector* of null-terminated string arguments. All versions cause the kernel to load the executable file whose name is either path or file, given above, overlaying the current program for the process, and passing it the remaining arguments.

The functions are also characterized by whether or not they contain a 'p' in their names. The versions that contain a 'p', `execlp()` and `execvp()`, expect the first argument to be a simple file name rather than a full path name, whereas the ones that do not contain a 'p': `execl()`, `execle()`, and `execv()`, require the full pathname for the first argument. The versions containing the 'p' will use the `PATH` environment variable to search for the file whose name is supplied, provided it does not contain any slashes. If it has a slash, then that is treated as a pathname, either relative or absolute, to the file to be loaded.

For all of these functions, the parameters named `arg` or `argv` above that follow the path or file parameter are passed to the executable as its own arguments. The first of these arguments must be a pointer to the executable file, because in UNIX, by convention, the first argument to a program (`argv[0]`) is always the name of the program itself, stripped of the preceding pathname. For example, to execute `"/bin/ls -l"` using `execl()`, you would use the syntax

```
execl( “/bin/ls”, “ls”, “-l”, (char *) 0);
```

In other words, the name of the executable occurs twice – first with the full pathname, and second with just the name of the file itself.

The differences between the different versions can be summarized as follows:

`execl, execle, execlp` expect the arguments to be presented as a comma-separated list of strings, terminated by a `NULL` pointer cast to a string, as in

```
execl( “/bin/ls”,  “ls”, “-l”, (char *) 0);
```

`execv, execvp` expect the arguments to be presented as a vector (array) whose last element is a `NULL` pointer as in:

```
strcpy(argvec[0], "ls");
strcpy(argvec[1], "-l");
argvec[2] = NULL;
execv( "/bin/ls", argvec );
```

`execlp, execvp` do not require a full path name:

```
execlp( “ls”, “ls”, “-l”, (char *) 0);
strcpy(argvec[0], "ls");
strcpy(argvec[1], "-l");
argvec[2] = NULL;
execvp( "ls", argvec );
```

`execle` is the same as `execl()` except that it has a third argument that is an array of pointers to environment strings exactly as `execve()` expects, with a `NULL` pointer as its last entry.

The functions other than `execle()` obtain the environment settings for the new process from the values in the external `environ` variable.

To illustrate , the following program, `execvpdemo.c`, uses its first argument as the executable to run, and the remaining arguments as its arguments.

Listing 7.9: execvpdemo.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char * argv[], char * envp[])
{
    if (argc < 2) {
        fprintf(stderr, "Usage: %s program args\n", argv[0]);
        exit (1);
    }
    execvp( argv[1],  argv +1 );
    perror("execvp");
    exit (1);
}
```

Since it searches the `PATH` environment variable, it can be called, for example, with

```
$ execvpdemo cp origfile newfile ..
```

7.9 Synchronizing Parents and Children: wait and exit

7.9.1 Exit() Stage Left

We have used the `exit()` function many times in various programs, but the only reason for doing so was that it was a way to terminate the calling process and return a non-zero integer value when some error condition arose. The `exit()` function does much more than this. Its synopsis is

```
#include <stdlib.h>
void exit(int status);
```

Three actions take place when `exit()` is called:

1. The process's registered *exit functions* run;
2. the system gets a chance to clean up after the process; and
3. the process gets a chance to have a status value delivered to its parent.

By an *exit function*, we mean a function that is run when `exit()` is called.

Before continuing, you may wonder why we would want a special function to run when `exit()` is called. Imagine that when your program terminates, it has to update a log file. Suppose the function that does this is named `update_log()`. Suppose also that the program is very large, that there are multiple points at which `exit()` is called, and that more than one programmer is maintaining this program. If the `exit()` function did not provide a means of invoking user-defined exit routines, then each time that anyone modified the program to insert a new call to `exit()`, he or she would have to remember to call `update_log()` first. However, by registering `update_log()` to run whenever `exit()` is called, it makes the programmer's job easier, since she does not have to worry about forgetting to include the call when the program is modified.

When `exit()` is called, the following actions take place in the given order:

1. All functions registered to run with the `atexit()` or `on_exit()` functions are run (in the reverse order in which they were registered with these routines).
2. All of the file streams opened through the Standard I/O Library are flushed and closed.
3. The kernel's `_exit()` function is called, passing the status argument to it.

Programmers can register a function to run when a process calls `exit()` using either `atexit()` or `on_exit()`. The preferred choice is `atexit()` since it is more portable. The man pages for both contain the details for how to register such exit functions. If more than one function is registered, they are run in the reverse order of the order in which they were registered (i.e., in last-in-first-out order). After the registered functions run, the `exit()` function flushes the streams and closes the files. The `exit()` function then calls `_exit(status)`. The kernel's `_exit()` function makes sure that

1. any open file descriptors are closed (not just those opened through Standard I/O Library functions),
2. all memory belonging to the process is released,
3. all children of the process (including zombies, defined below) are "adopted" by the `init()` process (meaning that `init()` is made the parent of these children,
4. the low-order eight bits of the integer argument to `exit()`, called its *exit status*, are made available to the parent process, and
5. under appropriate conditions, a `SIGCHLD` signal is sent to the parent process.

Actions (4) and (5) will be explained shortly. There are other actions that must take place within the clean-up routines of the kernel. This is just a partial list, including the most basic operations.

The following example shows how `atexit()` can be used to register a few exit functions:

Listing 7.10: atexitdemo.c

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

void Worker(void)
{
    printf("Worker #1 : Finished for the day.\n");
}
```

```
void Foreman(void)
{
    printf("Foreman    : Workers can stop for the day.\n");
}

void Boss(void)
{
    printf("First Boss: Foreman, tell all workers to stop work.\n");
}

int main(void)
{
    long max_exit_functions = sysconf(_SC_ATEXIT_MAX);

    printf("Maximum number of exit functions is %ld\n",
            max_exit_functions);
    if ((atexit(Worker)) != 0) {
        fprintf(stderr, "cannot set exit function\n");
        return EXIT_FAILURE;
    }

    if ((atexit(Foreman)) != 0) {
        fprintf(stderr, "cannot set exit function\n");
        return EXIT_FAILURE;
    }

    if ((atexit(Boss)) != 0) {
        fprintf(stderr, "cannot set exit function\n");
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

7.9.2 Waiting for Children to Terminate

After a process forks a child, how will it know if and when the child has finished whatever task it set out to accomplish? Typically, a process has to wait until the child or children finish completing their tasks before it can continue. The `fork()`, `exec()`, and `exit()` system calls need one more partner to form a complete ensemble, and that is the `wait()` family of calls. Generally speaking, the purpose of `wait()` is two-fold:

- to delay the parent until a child has terminated, and
- to obtain the status of a child that has terminated.

There are only two ways for a process to terminate: either "normally" by calling one of various exit functions[6] such as `exit()`, or "abnormally" and involuntarily as a result of receiving a signal that killed it, or calling `abort()`. (When a program either reaches the end of its code or executes a `return`, this results in an implicit call to `exit()`.) In either case, the parent can call `wait()`

[6]There are two other exit functions, including `_exit()` and `_Exit()`.

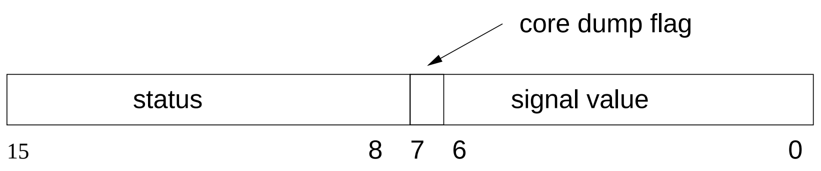

Figure 7.5: Two-byte exit status.

to determine the cause of termination. There are three different POSIX-compliant `wait()` system calls:

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

The `wait()` function causes the executing process to suspend its execution until any one of its children terminates. When a child terminates, the kernel sends a `SIGCHLD` signal to its parent, unless the parent has indicated that it does not want these signals. Upon receipt of a `SIGCHLD` signal, the parent resumes in the `wait()` code. The return value of `wait()` is the process-id of the child that just terminated or was just killed. It does not matter which child terminates. The process is resumed if any child terminates. If a process calls `wait()` but it has no children, `wait()` returns immediately with a -1.

The purpose of the status parameter is to receive information about how the child terminated. It is a pointer to a two-byte integer. If the child terminated normally using the `exit()` call, then the high-order byte of the value received by `wait()` contains the low-order byte of the integer passed in the `exit()` call's argument, and the low-order byte of the received value is 0[7]. If the child terminated abnormally because of an unhandled signal, then the low-order byte of the received value contains the signal value. If the child was terminated by a signal, then in particular bit 7 is set if there was a core dump[8]. Figure 7.5 shows how the two bytes of the status are arranged.

Based on these facts, the following code can be used for querying and extracting the status and signal state:

```
if ((status & 0x000000FF) ==0)
    /* low-order byte is zero, so high-order byte has status */
    exitStatus = status >> 8;  /* exitStatus contains exit status */
else {
    signum = status%128;  /* signum contains signal */
    if ( status & 0x00000080 )
```

[7]The convention when using `exit()` is to supply a zero on success and some non-zero value on failure.

[8]The fact that a core dump is supposed to occur does not mean that there will be a core file in your working directory. If your shell has been configured so that core dumps are disabled, then you will not see the file. On some systems, you can enable the core dump by running the command "`ulimit -c unlimited`", which allows your processes to create core files of unlimited size. The `ulimit` command is part of `bash` and you can read the `bash` man page for more details.

```
        /* core dump took place */
}
```

However, using the following macros, which are defined in `<sys/wait.h>`, makes the code more portable:

```
if (WIFEXITED(status))
    /* true implies exit() was called to terminate the child   */
    exit_status = WEXITSTATUS(status);  /* extract exit status */
else if ( WIFSIGNALED(status) ) {
    /* true if signal killed child */
    signum = WTERMSIG(status);  /* extract signal that killed child */
#ifdef WCOREDUMP
    if ( WCOREDUMP(status) )
        /* true if a core dump took place  */
#endif
}
```

- The `WIFEXITED(status)` macro returns true if the child terminated normally, i.e., by calling `exit(3)` or `_exit(2)`, or by returning from `main()`. In this case the `WEXITSTATUS(status)` macro returns the exit status of the child. This macro should only be employed if `WIFEXITED()` returned true.
- The `WIFSIGNALED(status)` macro returns true if the child process was terminated by a signal. If it returns true, then the `WTERMSIG(status)` macro returns the number of the signal that caused the child process to terminate. The `WTERMSIG()` macro should only be used if `WIFSIGNALED()` returned true.
- The `WCOREDUMP(status)` macro returns true if the child produced a core dump. This macro should only be used if `WIFSIGNALED` returned true. *This macro is not specified in POSIX.1-2001* ; only use this enclosed in

  ```
  #ifdef WCOREDUMP ... #endif.
  ```

7.9.3 Using wait()

Listing 7.11 contains an example that puts together the use of `fork()`, `exit()`, and `wait()`. It is the typical way in which these three primitives are used. In this example the user is prompted to supply an exit value for the child, which is then passed to the `exit()` call, to show that the value is then available to the parent in the `wait()` call.

Listing 7.11: waitdemo2.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <signal.h>
```

```
void child()
{
    int exit_code;
    printf("I am the child and my process id is %d.\n",getpid());
    sleep(2);
    printf("Enter a value for the child exit code:");
    scanf("%d",&exit_code);
    exit(exit_code);
}

int main(int argc, char* argv[])
{
    int pid;
    int status;

    printf("Starting up...\n");
    if ( -1 == (pid = fork())) {
        perror("fork");
        exit(1);
    }
    else if ( 0 == pid )
        child();
    else {  /* parent code */
        printf("My child has pid %d and my pid is %d.\n", pid, getpid());
        if ((pid = wait(&status)) == -1) {
            perror("wait failed");
            exit(2);
        }
        if (WIFEXITED(status))  { /* low order byte of status equals 0 */
            printf("Parent: Child %d exited with status %d.\n",
                   pid, WEXITSTATUS(status));
        }
        else if ( WIFSIGNALED(status) ) {
            printf("Parent: Child %d exited with error code %d.\n",
                   pid, WTERMSIG(status));
#ifdef WCOREDUMP
            if ( WCOREDUMP(status) )
                printf("Parent: A core dump took place.\n");
#endif
        }
    }
    return 0;
}
```

Notes.

- When the child process displays a message such as

  ```
  I am the child and my process id is 5666.
  Enter a value for the child exit code:
  ```

  enter an exit code and observe that it is printed by the parent after the parent's call to `wait()` finishes. Then run the program again but this time send a signal to the child process from another terminal using the kill command, i.e.,

```
$ kill -10 5666
```

and observe that the parent displays the message

```
Parent: Child 5666 exited with error code 6.
Parent: A core dump took place.
```

- The conditional compilation macro is used because the `WCOREDUMP` macro is not available on all UNIX systems, as noted above.

Sometimes a parent does not care very much about how its children terminate. A parent can explicitly tell the kernel that it doesn't care if and when its children terminate by setting the `SA_NOCLDWAIT` flag, which prevents the delivery of `SIGCHLD` signals to itself. It does this using the `sigaction()` call:

```
const struct sigaction act;
act.sa_flags = SA_NOCLDWAIT;
sigaction (SIGCHLD, &act, NULL);
```

Notice that it is not necessary to set a signal handler in this call; it is enough to just pass the `sa_flags` field. If a process has so indicated to the kernel its lack of interest in its children, then when a child terminates, the child's status will not be delivered to its parent. The child will be completely terminated immediately. Similarly, when the parent sets the action for `SIGCHLD` to `SIG_IGN`, the status will be discarded and the child completely terminated.

If the parent process has set neither `SA_NOCLDWAIT` nor the action for `SIGCHLD` to `SIG_IGN` and is presently executing any of the `wait()` calls described below, then the status will be delivered to the parent and a `SIGCHLD` signal sent to it. If the parent is not currently waiting, then when the parent does invoke `wait()`, it will receive the status. If the parent is not executing any form of `wait()`, though, the child process is transformed into a *zombie process*. A zombie process is an inactive process and it will be deleted at some later time when its parent process executes a `wait()` call. Zombie processes exist simply to provide their parents with their status values at a future time.

The flip side of this issue is what happens when a process terminates before its children. When a process terminates and has children, these children are not terminated also. In comes the `init()` process. The `init()` process adopts orphans, so if a process terminates and has any children, they are adopted by `init()`. When the children terminate, their exit status will be sent to `init()`.

7.9.4 Using waitpid()

The `waitpid()` function has three parameters. The first is the process-id of the child to wait for, the second is a pointer to the variable in which to store the status, and the last is an optional set of flags. If the pid is -1, then it tells the kernel that the process will wait for any child, like `wait()`. Setting the pid to 0 means to wait for only those children in the same process group as the parent. Setting the pid to -*G*, for some positive integer *G*, means to wait for any child whose process group-id is *G*.

There are three flags that can be passed to `waitpid()`:

`WNOHANG`	return immediately if no child has exited.
`WUNTRACED`	also return if a child has stopped (but not traced via `ptrace(2)`). Status for traced children which have stopped is provided even if this option is not specified.
`WCONTINUED`	(Since Linux 2.6.10) also return if a stopped child has been resumed by delivery of `SIGCONT`.

The program in Listing 7.12 below combines some of the preceding ideas and demonstrates the use of `waitpid()` with the `WNOHANG` flag.

Listing 7.12: waitpiddemo.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <signal.h>

void child()
{
    int exit_code;

    printf("I am the child and my process id is %d.\n",getpid());
    sleep(2);
    printf("Enter a value for the child exit code followed by <ENTER>.\n");
    scanf("%d",&exit_code);
    exit(exit_code);
}

int main(int argc, char* argv[])
{
    pid_t pid;
    int status;
    int signum;

    printf("Starting up...\n");
    if ( -1 == (pid = fork())) {
        perror("fork");
        exit(1);
    }
    else if ( 0 == pid ) {
        child();
    }
    else {
        /* wait for specific child process with waitpid()
           If no child has terminated, do not block in waitpid()
           Instead just sleep. (Would do something useful instead.)
        */
        while (0 == waitpid(pid, &status, WNOHANG) ) {
            printf("still waiting for child\n");
            sleep(1);
        }
        /* pid is the pid of the child that terminated */
        if (WIFEXITED(status)) {
            printf("Exit status of child %d was %d.\n",
                   pid, WEXITSTATUS(status));
        }
        else if ( WIFSIGNALED(status) ) {
            signum = WTERMSIG(status);
            printf("Parent: Child %d exited by signal %d.\n",pid,
                    signum);
#ifdef WCOREDUMP
            if ( WCOREDUMP(status) )
                printf("Parent: A core dump took place.\n");
#endif
        }
    }
    return 0;
}
```

Notes.

- The parent is in a busy waiting loop in this example, waiting for the child to terminate. The `WNOHANG` flag to `waitpid()` allows it to continue polling the `waitpid()` call and do something else in the meanwhile. The body of the loop would be replaced with a task that the parent could do while waiting for the child. The advantage of this is that the parent does not have to block, waiting for the child, but can instead do work. If there is no work to do, then this paradigm is not the one to use.
- You can send a signal to the child in the same way as for the program in Listing 7.11 to observe that the parent code detects the error return.

7.9.5 Non-blocking waits

Instead of calling `wait()` or `waitpid()`, a process can establish a `SIGCHLD` handler that will run when a child terminates. The `SIGCHLD` handler can then call `wait()`. This frees the process from having to poll the `wait()` function. It only calls `wait()` when it is guaranteed to succeed. The following example (`rabbits2.c` in the demos directory) demonstrates how this works.

Listing 7.13: rabbits2.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <limits.h>
#include <sys/wait.h>
#include <termios.h>

#define  NUM_CHILDREN   5
#define  SLEEPTIME      30

/**  child()  The code that is executed by each child process
```

```
 *   All this does is register the SIGINT signal handler and then
 *   sleep SLEEPTIME seconds. If a child is delivered a SIGINT, it
 *   exits with the exit code 99. See on_sigint() below.
 */
void child        ();

/**  on_sigint()  Signal handler for SIGINT
 *   All it does it call exit with a code of 99.
 */
void on_sigint    ( int signo );

/**  on_sigchld()  Signal handler for SIGCHLD
 *   This calls wait() to retrieve the status of the terminated child
 *   and get its pid. These are both stored into global variables that
 *   the parent can access in the main program. It also sets a global
 *   flag.
 */
void on_sigchld   ( int signum );

/* These variables are declared with the volatile qualifier to tell the
   compiler that they are used in a signal handler and their values
   change asynchronously. This prevents the compiler from performing an
   optimization that might corrupt the program state. All three are
   shared by the main parent process and the SIGCHLD handler.
 */
volatile int          status;
volatile pid_t        pid;
volatile sig_atomic_t child_terminated;

/*****************************************************************
                           Main Program
*****************************************************************/

int main(int argc, char* argv[])
{
    int        count = 0;
    const int NumChildren = NUM_CHILDREN;
    int i;
    struct sigaction newhandler;      /* for installing handlers */

    printf("About to create many little rabbits...\n");
    for ( i = 0; i < NumChildren; i++) {
        if ( -1 == (pid = fork())) {
            perror("fork");
            exit(-1);
        }
        else if ( 0 == pid ) { /* child code */
            /* Close standard output so that children do not print
               parent's output again. */
            close(1);
            child();
            exit(1);
        }
        else { /* parent code */
```

```
            if ( 0 == i )
               printf("Another ");
            else if (i < NumChildren-1 )
               printf("and another ");
            else
               printf("and another.\n");
        }
    }
    /* parent continues here*/
    /* Set up signal handling  */
    newhandler.sa_handler = on_sigchld ;
    sigemptyset(&newhandler.sa_mask);
    if ( sigaction(SIGCHLD, &newhandler, NULL) == -1 ) {
        perror("sigaction");
        return (1);
    }

    /* Enter a loop in which work could happen while the global flag
       is checked to see if any child has terminated. */

    child_terminated = 0;          /* Set flag to 0 */
    while( count < NumChildren ){
        if ( child_terminated ) {
            if ( WIFEXITED(status) )
                printf("Rabbit %d died with code %d.\n",
                        pid, WEXITSTATUS(status));
            else if (  WIFSIGNALED(status) )
                printf("Rabbit %d was killed by signal %d.\n",
                        pid, WTERMSIG(status));
            else
                printf("Rabbit %d dies with status %d.\n",pid,status);
            child_terminated = 0;
            count++;
        }
        else  {
            /* do something useful here. for now just delay a bit */
            sleep(1);
        }
    }

    printf("All rabbits have terminated and been laid to rest.\n");
    return 0; /* main returns; child never reaches here */
}

void on_sigint( int signo )
{
    exit(99);
}

void child()
{
    struct sigaction newhandler;

    newhandler.sa_handler = on_sigint;
    sigemptyset(&newhandler.sa_mask);
    if ( sigaction(SIGINT, &newhandler, NULL) == -1 ) {
        perror("sigaction");
        exit (1);
    }
    sleep(SLEEPTIME);
}

void on_sigchld ( int signum )
{
    int child_status;

    if ((pid = wait(&child_status)) == -1) {
        perror("wait failed");
    }
    child_terminated = 1;
    status = child_status;
}
```

Notes.

- The C Standard I/O Library by default uses buffered streams. This means that when a process uses the C output functions such as `printf()`, the output is placed into a buffer before being delivered to the terminal device. When each child is created, it is given a copy of the parent's buffers at the time of creation. If we did not close the file descriptor in the child immediately, then when the child terminated, its buffer would be flushed and multiple copies of the parent's output would appear in the terminal window. We cannot use `fclose(stdout)` because `fclose()` is designed to flush the buffers and would also cause the output to appear. We must use the lower-level file descriptors. Try commenting out the line "`close(1)`" and running the program.
- The child is designed to catch a `SIGINT` and exit within the signal handler. The reason for this is to allow the user to send a `SIGINT` (by issuing a `kill -2` to the child on the command line) in order to show that even if a signal caused the signal handler to run, the fact that the child called `exit()` means that the parent will see that the child died by calling `exit()`, not as a result of being sent a signal.
- If the child is not killed by a signal, then it terminates normally after a 30 second sleep.
- The `on_sigchld()` handler, after calling `wait()`, sets an atomic flag and lets the main program do the work of handling the child's exit.
- The program does not test the return value of `fork()` for failure, just to save space here.
- Although POSIX does not permit it, some systems allow signals to be lost, and it is possible to lose a `SIGCHLD` signal in this code. If multiple children terminate in a small time sequence, and the parent is in the `SIGCHLD` handler, then some of the `SIGCHLD` signals may be merged into a single signal. GNU C allows this for example. The fix requires much more complex code that maintains a list of the child processes and which inspects that list within the handler.

7.9.6 Waiting for State Changes in Children

The `wait()` family was extended in Linux 2.6.9 with the inclusion of `waitid()`, which can be used to gather information about other changes in the state of child processes besides their termination:

```
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

The `idtype` can be one of `P_PID`, `P_PGID`, or `P_ALL`. If it is `P_PID`, it waits for the process whose process-id is passed as the second argument. If it is `P_PGID`, it waits for any process whose group-id is the second argument. If `P_ALL`, the second argument is ignored and it acts like the ordinary `wait()`.

The options parameter is the OR of one or more of the flags

`WEXITED`	Wait for children that have terminated.
`WSTOPPED`	Wait for children that have been stopped by delivery of a signal.
`WCONTINUED`	Wait for (previously stopped) children that have been resumed by delivery of `SIGCONT`.

and optionally the `WNOHANG` flag described earlier as well as the `WNOWAIT` flag, which leaves the child process as if the parent never called `wait()`, in case it wants to retrieve the status at a later time.

If waitid() completes successfully, it fills in the `siginfo_t` structure pointed to by the `infop` parameter . The `siginfo_t` structure is the same structure used by the `sigaction()` function. It is a union, so the members filled in by `sigaction()` are not exactly the same as those filled in by `waitid()`. `waitid()` provides the following members:

`si_pid`	The process-id of the child
`si_uid`	The real user-id of the child
`si_signo`	`SIGCHLD`
`si_status`	Either the exit status or the signal that caused the child's state to change.
`si_code`	Exactly one of `CLD_EXITED` if the child exited, `CLD_KILLED` if the child was killed by a signal, `CLD_STOPPED` if the child was stopped by a signal, or `CLD_CONTINUED` if it was continued by a `SIGCONT`.

The way to use `waitid()` is to inspect the value of `infop->si_code` to determine the state of the child before accessing `infop-status`. The following listing, modified from the one in the man page for `wait()`, demonstrates how to use `waitid()`.

Listing 7.14: waitiddemo.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <signal.h>

#define SLEEPTIME   60
```

```
int main(int argc, char* argv[])
{
    pid_t      pid;
    siginfo_t  siginfo;

    if ( -1 == (pid = fork())) {
        perror("fork");
        exit(1);
    }
    else if ( 0 == pid ) {
        printf("Child pid is %d\n", getpid());
        sleep(SLEEPTIME);
        exit(0);
    }
    /* Parent code */
    else do {
        /* Zero out si_pid in case the sig_info_t struct does not get */
        /* initialized because no children are waitable. */
        siginfo.si_pid = 0;

        /* Wait for changes in the state of the child created above,  */
        /* specifically, stopping, resuming, exiting, and return */
        /*immediately if no child is waitable. */
        if (-1 == waitid(P_PID, pid, &siginfo,
                  WEXITED | WSTOPPED | WCONTINUED | WNOHANG) ) {
            perror("waitid");
            exit(EXIT_FAILURE);
        }
        if ( siginfo.si_pid == 0 )
            /* no child is waitable. */
            continue;
        switch ( siginfo.si_code ) {
                case CLD_EXITED:
                printf("Child exited with status %d\n",
                        siginfo.si_status );
                break;
                case CLD_KILLED:
            case CLD_DUMPED:
                printf("Child killed by signal %d\n",
                        siginfo.si_status );
                break;
                case CLD_STOPPED:
                printf("Child stopped by signal %d\n",
                        siginfo.si_status );
                break;
                case CLD_CONTINUED:
                printf("Child continued\n");
                break;
        }
    } while ( siginfo.si_code != CLD_EXITED &&
                siginfo.si_code != CLD_KILLED &&
                siginfo.si_code != CLD_DUMPED );
    return 0;
}
```

Notes.

- The `si_status` field does not need to be bit-manipulated to extract its value. The value contained there has already been shifted as necessary.
- The while-loop in the parent continues until the child is killed or terminated to give you a a chance to stop and continue the child.
- If you run this program in one terminal, and then from another issue `kill` commands, or run it in the background on one terminal and issue kill commands on the same terminal, you will see output like the following:

```
$ waitiddemo 77
Child pid is 15243
Child exited with status 77
$
```

  Then do it again without the command-line argument:

```
$ waitiddemo &
Child pid is 15245
$ kill -STOP 15245
Child stopped by signal 19
$ kill -CONT 15245
Child continued
$ kill -TERM 15245
Child killed by signal 15
[1]+ Done waitid2
$
```

7.10 Summary

The four principal tools in process creation and control are `fork()`, `exec()`, `exit()`, and `wait()` and their related functions. Add to the toolbox the things you have learned about signals and signal handling and you have the means of creating and managing processes effectively. What is still lacking is a means for these processes to exchange and share data effectively. At this point the only way they can share data other than provided by the signal mechanism is through the file system, which is extremely slow. Inter-process communication is the topic of the next chapter.

Because the new process is a copy of the parent process, it shares all open files and all library buffers. When the two processes both use the C I/O Library, care must be taken to prevent unexpected consequences of this.

Process creation is a time-consuming activity in the kernel, with high overhead in memory copying. When the objective is to use shared variables and common code, light-weight processes, or threads, are the better solution. This topic follows as well.

Bibliography

[1] Keith Haviland, Marcus Gray, and Ben Salama. *Unix System Programming.* Addison-Wesley Longman, Inc., 2nd edition, 1998.