中国矿业大学计算机学院

2019-2020(2)本科生 Linux 操作系统课程报告

内容范围 进程控制编程

指 标 点 1.2 占 比 20%

学生姓名 袁孝健 学 号 06172151

专业班级 信息安全 2017-01 班

任课教师 杨东平

课程基础理论掌握程度	熟练 □	较熟练 □	一般 □	不熟练 □
综合知识应用能力	强 □	较强 □	一般 □	差 □
报告内容	完整 □	较完整 □	一般 □	不完整 □
报告格式	规范 □	较规范 □	一般 □	不规范 □
实验完成状况	好 □	较好 □	一般 □	差 □
工作量	饱满 □	适中 □	一般 □	欠缺 □
学习、工作态度	好 □	较好 □	一般 □	差 □
抄袭现象	无 □	有 □	姓名:	
存在问题				
总体评价				

综合成绩: 任课教师签字:

年 月

摘　要

计算机系统的各种硬件资源是有限的，在现代多任务操作系统上同时运行的多个进程都需要访问这些资源，为了更好的管理这些资源进程是不允许直接操作的，所有对这些资源的访问都必须有操作系统控制。也就是说操作系统是使用这些资源的唯一入口，而这个入口就是操作系统提供的系统调用。在 Linux 中系统调用是用户空间访问内核的唯一手段，除异常和陷入外，他们是内核唯一的合法入口。本文介绍了 fork、exec、exit、wait 四个常用的 Linux 系统调用，对每个系统调用从原型、功能、原理、实例四个方面进行了详细的阐述。通过对系统调用的学习，使得对 Linux 操作系统下的 C 编程以及其内部的实现原理有了进一步的理解。

关键词： Linux 系统；系统调用；内核

Abstract

Various hardware resources of the computer system are limited. Multiple processes running on a modern multitasking operating system need to access these resources. In order to better manage these resources, processes are not allowed to operate directly. All of these resources Access must be controlled by the operating system. In other words, the operating system is the only entry point for using these resources, and this entry point is the system call provided by the operating system. In Linux, system calls are the only means for user space to access the kernel. Except for exceptions and traps, they are the only legal entry point for the kernel. This article introduces the four commonly used Linux system calls of fork, exec, exit, and wait. Each system call is elaborated from four aspects: prototype, function, principle, and example. Through the study of the system call, the C programming under the Linux operating system and its internal realization principles have been further understood.

Keywords: Linux system; system call; kernel

目　　录

3 进程控制编程

要求: (1)掌握进程的创建 fork 系统调用的原理。

(2)掌握 exec 系统调用的原理。

(3)掌握 exit 系统调用的原理。

(4)掌握 wait 系统调用的原理。

根据阐述的详细和完整程度给出相应的分值。

说明：如果撰写规范不符合《计算机学院考查类课程报告撰写规范》要求的，整体上酌情扣除 1-10 分。

3.1 fork 系统调用

3.1.1 函数原型

（1）原型:

```
pid_t fork(void)
```

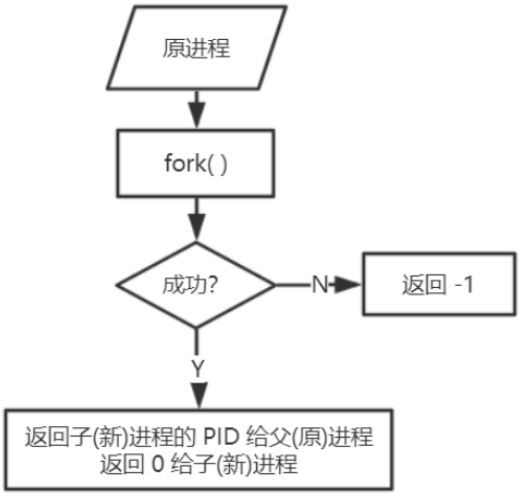
（2）头文件:

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

（3）返回值：

fork 被调用一次却能够返回两次且可能有三种不同的返回值:

- 在父进程中，fork 返回新创建子进程的进程 ID(通常为父进程 PID+1)。
- 在子进程中，fork 返回 0。
- 如果出现错误，fork 返回一个负值。



（4）fork 出错的原因:

- 当前的进程数已经达到了系统规定的上限，这时 errno 的值被设置为 EAGAIN。
- 系统内存不足，这时 errno 的值被设置为 ENOMEM。

3.1.2 函数功能

（1）创建一个与原来进程几乎完全相同的进程，即两个进程可以做完全相同的事，但如果初始参数或者传入的变量不同，两个进程也可以做不同的事。

（2）一个进程调用 fork 函数后，系统先给新的进程分配资源，例如存储数据和代码的空间。然后把原来的进程的所有值都复制到新的进程中，只有少数值与原来的进程的值不同，相当于克隆了一个自己。

3.1.3 底层原理

（1）Linux 通过 clone()系统调用实现 fork()。fork()、vfork()和_clone()库函数都根据各自需要的参数标志去调用 clone()，然后由 clone()去调用 do_fork()。

（2）do_fork()完成了创建中的大部分工作，它定义 kernel/for 中，该函数调用 copy_process()。

（3）接下来 copy_process()实现的工作如下:

① 调用 dup_task_struct()为新进程创建一个内核栈，threaad_info 结构和 task_struct,这些值与当前进程的值相同。此时，子进程和父进程的描述符是完全相同的。

② 检查新创建的这个子进程后，当前用户所拥有的进程数目没有超出给他分配的资源的限制。

③ 子进程着手使自己与父进程区别拷来，进程描述符内的许多成员都要被清 0 或设为初始值。进程描述符的成员值并不是继承而来的，而主要是统计信息，进程描述符中大多数的数据都是共享的。

④ 接下来，子进程的状态被设置为 TASK_UNINTERRUPTIBLE(不可中断)以保证它不会投入运行。

⑤ copy_process()调用 copy_flags()以更新 task_struct 的 flags 成员。表明进程是否拥有超级用户权限的 PF_SUPERPRIV 标志被清 0.表名进程还没有调用 exec()函数的 PF_FORKNOEXEC 标志被设置。

⑥ 调用 get_pid()为新进程获取一个有效的 PID。

⑦ 根据传递给 clone()的参数标志，copy_process()拷贝或共享打开的文件，文件系统信息，信号处理函数，进程地址空间和命名空间等。在一般情况下，这些资源会被给定进程的所有线程共享；否则，这些资源对每个进程是不同的，因此被拷贝到这里。

⑧ 让父进程和子进程平分剩余的时间片

⑨ 最后 copy_process()做扫尾工作并返回一个指向子进程的指针。

（4）再回到 do_fork()函数，如果 copy_process 函数成功返回，新创建的子进程被唤醒并让其投入运行。内核有意选择子进程首先执行。因为一般子进程都会马上调用 exec()函数，这样可以避免写时拷贝的额外开销，如果父进程首先执行的话，有可能会开始向地址空间写入。

3.1.4 代码实例

（1）代码

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main () {
    pid_t p1, p2;
    p1 = fork();
    if (p1 < 0) {
        printf("Error in fork!");
    } else if (p1 == 0) {
        printf("Child process b, pid: %d\n", getpid());
    } else {
        p2 = fork();
        if (p2 < 0) {
            printf("Error in fork!");
        } else if (p2 == 0) {
            printf("Child process c, pid: %d\n", getpid());
        } else {
            printf("Parent process a, pid: %d\n", getpid());
        }
    }
    return 0;
}
```

（2）运行截图

```
ubuntu@VM-0-14-ubuntu:~/learn$ ./fork_sample
Parent process a, pid: 28085
ubuntu@VM-0-14-ubuntu:~/learn$ Child process c, pid: 28088
Child process b, pid: 28087
```

3.2 exec 系统调用

3.2.1 函数原型

（1）原型:

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

上述函数中只有 execve 函数是系统调用，其它的都是库函数，其它几个最终都会调用它

（2）头文件:

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

（3）返回值:

出错则返回 -1，失败原因记录在 error 中。

（4）区别:

① 查找方式不同

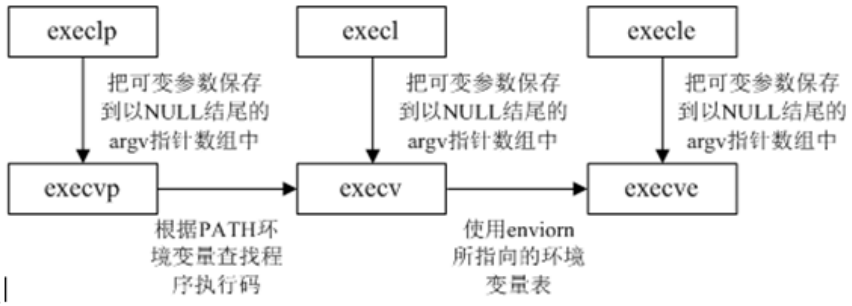
- 前 4 个函数的查找方式都是完整的文件目录路径。
- 后 2 个函数(以 p 结尾的两个函数)可以只给出文件名，系统会自动从环境变量 $PATH 所指出的路径中进行查找。

② 参数传递方式不同

- 数名的第 5 位字母为 l(list)的表示逐个列举的方式。
- 函数名的第 5 位字母为 v(vertor)的表示将所有参数整体构造成指针数组传递，然后将该数组的首地址当作参数传给它，数组中的最后一个指针要求是 NULL。

③ 环境变量不同：

- 以 e(environment)结尾的两个函数 execle、execve 可以在 envp[]中指定当前进程所使用的环境变量替换掉该进程继承的环境变量。
- 其它函数把调用进程的环境传递给新进程。



3.2.2 函数功能

exec 函数族的作用是根据指定的文件名找到可执行文件，并用它来取代调用进程的内容，换句话说，就是在调用进程内部执行一个可执行文件。这里的可执行文件既可以是二进制文件，也可以是任何 Linux 下可执行的脚本文件。

与一般情况不同，exec 函数族的函数执行成功后不会返回，因为调用进程的实体，包括代码段，数据段和堆栈等都已经被新的内容取代，只留下进程 ID 等一些表面上的信息仍保持原样。只有调用失败了，它们才会返回一个-1，从原程序的调用点接着往下执行。

系统调用 exec 和 fork 都可以运行一个可执行文件，不同的是 exec 是以新的进程去代替原来的进程，但进程的 PID 保持不变。也就是说，exec 并没有创建新的进程，只是替换了原来进程上下文的内容。一般是在 fork 函数新建一个进程后，再让进程去执行 exec 系统调用，这样可以很好地提高效率。

3.2.3 底层原理

内核中实际执行 execv()或 execve()系统调用的程序是 do_execve()，这个函数先打开目标映像文件，并从目标文件的头部（第一个字节开始）读入若干（当前 Linux 内核中是 128）字节（实际上就是填充 ELF 文件头，下面的分析可以看到），然后调用另一个函数 search_binary_handler()，在此函数里面，它会搜索我们上面提到的 Linux 支持的可执行文件类型队列，让各种可执行程序的处理程序前来认领和处理。如果类型匹配，则调用 load_binary 函数指针所指向的处理函数来处理目标映像文件。在 ELF 文件格式中，处理函数是 load_elf_binary 函数：

```
sys_execve() > do_execve() > do_execveat_common > search_binary_handler() > load_elf_binary()
```

3.2.4 代码实例

（1）代码：

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

int main(void)
{
    printf("Entering a new process!\n");
    //利用 execl 将当前进程 main 替换掉
    execl("/bin/ls","ls","-l",NULL);
    //因为不出错是不会返回的,所以下面这句不会输出
    printf("Exiting the process!\n");
    return 0;
 }
```

运行截图:、

```
ubuntu@VM-0-14-ubuntu:~/learn$ ./exec_sample1
Entering a new process!
total 160
-rwxrwxr-x 1 ubuntu ubuntu  178 Mar 29 16:33 1.sh
-rwxrwxr-x 1 ubuntu ubuntu 8344 Mar 30 10:30 beexec
-rw-rw-r-- 1 ubuntu ubuntu  198 Mar 30 10:28 beexec.c
-rwxrwxr-x 1 ubuntu ubuntu 8712 Apr  3 15:03 comprehensive
-rwxrwxr-x 1 ubuntu ubuntu 8576 Apr  3 16:38 comprehensive2
-rw-rw-r-- 1 ubuntu ubuntu  450 Apr  3 17:26 comprehensive2.c
-rw-rw-r-- 1 ubuntu ubuntu  749 Apr  3 15:02 comprehensive.c
-rw-rw-r-- 1 ubuntu ubuntu   38 Apr  7 17:17 cumt.txt
-rwxrwxr-x 1 ubuntu ubuntu 8344 Mar 30 10:29 doexec
-rw-rw-r-- 1 ubuntu ubuntu  204 Mar 30 10:29 doexec.c
-rw-rw-r-- 1 ubuntu ubuntu  214 Apr  8 21:25 exec.c
-rwxrwxr-x 1 ubuntu ubuntu 8344 Apr  8 21:26 exec_sample1
-rwxrwxr-x 1 ubuntu ubuntu 8392 Apr  8 18:28 fork_sample
-rw-rw-r-- 1 ubuntu ubuntu  450 Apr  8 18:28 fork_sample.c
-rwxrwxr-x 1 ubuntu ubuntu  494 Mar 31 16:26 guess.sh
-rwxrwxr-x 1 ubuntu ubuntu  281 Apr  1 19:59 judgeAlive.sh
-rwxrwxr-x 1 ubuntu ubuntu 8576 Apr  3 16:52 nameedpipe_read
-rw-rw-r-- 1 ubuntu ubuntu  367 Apr  3 16:52 nameedpipe_read.c
```

execl 将当前进程 main 替换，正常情况下不返回，所以看不到最后一句的输出。

（2）代码

① argv.c

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

int main(int argc, char *argv[], char *envp[])
{
    extern char **environ;
    int i;

    printf("pid: %d\n", getpid());
    // 打印 argv
    printf("%s\n", "argv");
    for (i=0; i < argc; i++) {
        printf("  argv[%d]: %s\n", i, argv[i]);
    }

    //打印 envp
    printf("%s\n", "envp(0~4)");
    for (i=0; envp[i] && i != 5; i++) {
        printf("  envp[%d]: %s\n", i, envp[i]);
    }

    //打印 environ,需要和 envp 相同
    printf("%s\n", "environ(0~4)");
    for (i=0; environ[i] && i != 5; i++) {
        printf("  environ[%d]: %s\n", i, environ[i]);
    }
    return 0;
}
```

② execve_sample.c

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

int main(int argc, char *argv[], char *envp[])
{
    char *const pg_argv[] = {"./argv", "-dummy-param", NULL};
    char *pg_envp[]     = {"some_key_val_pair", NULL};

    printf("pid: %d\n", getpid());

    printf("---------- before execve ----------\n");
```

```
    if (execve("./argv", pg_argv, pg_envp) == -1)
        perror("execve error: ");

    return 0;
}
```

运行截图：

```
ubuntu@VM-0-14-ubuntu:~/learn$ gcc argv.c -o argv
ubuntu@VM-0-14-ubuntu:~/learn$ gcc execve_sample.c -o execve_sample
ubuntu@VM-0-14-ubuntu:~/learn$ chmod +x execve_sample
ubuntu@VM-0-14-ubuntu:~/learn$ ./execve_sample
pid: 3546
---------- before execve -----------
pid: 3546
argv
  argv[0]: ./argv
  argv[1]: -dummy-param
envp(0~4)
  envp[0]: some_key_val_pair
environ(0~4)
  environ[0]: some_key_val_pair
```

3.3 exit 系统调用

3.3.1 函数原型

（1）原型:

```
void _exit(int status)
void exit(int status)
```

（2）头文件:

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

（3）返回值:

staus 是程序返回的状态码，exit 系统调用没有返回值。

（4）exit()和_exit()的区别:

- _exit() 的作用最简单：直接使进程停止运行，清除其使用的内存空间，并销毁其在内核中的各种数据结构.
- exit() 在终止进程之前要检查文件的打开情况，把文件。
  缓冲区中的内容写回文件，即“清理 I/O 缓冲"。
- 两者最终都要将控制权交给内核。
- 因此，要想保证数据的完整性，就一定要使用 exit()。

3.3.2 函数功能

exit 系统调用用来终止发出调用的进程，status 是返回给父进程的状态值，父进程可通过 wait 系统调用获得。

通常在 Linux 中让一个进程退出，分为以下两种:

（1）正常退出:

- main() 函数执行完成或在 main() 函数中执行 return。
- 调用 exit() 函数。
- 调用 _exit() 函数。

（2）异常退出:

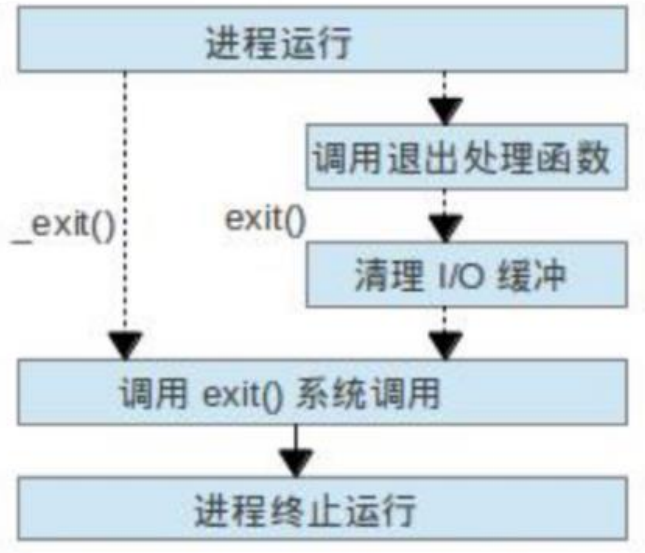
- 调用 abort() 函数。
- 进程收到某个信号，而该信号使程序终止。

（3）注意

- exit()终止进程时，将使终止的进程进入僵死状态，放它占有的资源，撤除进程上下文，但仍保留 proc 结构。
- 子进程还未终止，但父进程已终止时，将交由 init 进程处理。

3.3.3 底层原理

当进程执行到 exit()或_exit()函数时，进程会无条件的停止剩下的所有操作,清理各种数据结构，并终止本进程的运行。 这两个函数还是有区别的，它们调用过程如下:



由上图可以看出：

① _exit()函数的作用是: 直接使进程停止运行，清除其使用的内存空间，并清除其在内核中的各种数据结构;

② 而 exit()函数则在这些基础上做了一些包装，在执行退出之前加了若干道工序。

exit()函数和_exit()函数的最大区别就在于 exit()函数在终止当前进程之前要检查

该进程打开过那些文件，把文件缓冲区中的内容写回文件，也就是图中的“清理 I/O 缓冲”一项。

在 Linux 的标准函数库中，有一种被称为“缓冲 I/O”的操作，其特征就是对应每一个打开的文件，在内存中都有一片缓冲区。每次读文件时，会联系读出若干条记录，这样在下次读文件时就可以直接从内存的缓冲区当中读取；同样的，每次写文件的时候，也仅仅是写入内存的缓冲区，等满足了一定的条件时(如达到一定数量或遇到特定字符等)，再将缓冲区中的内容一次性写入文件。

这种技术大大增加了文件读写的速度，但是也给编程带来了一点麻烦. 比如有些数据你认为已经被写入到文件中，实际上因为没有满足特定的条件，他们还只是被保存在缓冲区内，这时用_exit()函数直接将进程关闭掉，缓冲区中的数据就会丢失. 因此，为了数据的完整性，请使用 exit()函数。

3.3.4 代码实例

（1）exit 系统调用

代码：

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

int main ()
{
   printf("Hello,World!\n");
    printf("I am a student from CUMT.");
    exit(2);
    return 0;
}
```

运行截图：

```
ubuntu@VM-0-14-ubuntu:~/learn$ ./exit_sample
Hello,World!
I am a student from CUMT.ubuntu@VM-0-14-ubuntu:~/learn$ echo $?
2
```

exit 在退出的时候，会刷新内存缓冲区，所以两句话都会打印出来，并且退出码为 2。

（2）_exit 系统调用

代码：

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

int main ()
```

```
{
    printf("Hello,World!\n");
    printf("I am a student from CUMT.");
    _exit(2);
    return 0;
}
```

运行截图：

```
ubuntu@VM-0-14-ubuntu:~/learn$ ./_exit_sample
Hello,World!
ubuntu@VM-0-14-ubuntu:~/learn$ echo $?
2
```

_exit 会直接结束掉进程，所以我们可以看到第一句话，第二句不会输出在显示屏上，且退出码为 2。

3.4 wait 系统调用

3.3.1 函数原型

（1）原型：

```
pid_t wait(int *status)
pid_t waitpid(pid_t pid,int *status,int options)
```

（2）头文件：

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

（3）返回值：

成功则返回成功结束运行的子进程的进程号 PID，否则返回-1，失败原因存于 errno 中。

（4）参数：

① status：返回子进程退出时的状态。

② pid：

- pid>0 时：等待进程号为 pid 的子进程结束。
- pid=0 时：等待组 ID 等于调用进程组 ID 的子进程结束。
- pid=-1 时：等待任一子进程结束，等价于调用 wait()。
- pid<-1 时：等待组 ID 等于 PID 的绝对值的任一子进程结束。

③ options：

- WNOHANG：若 pid 指定的子进程没有结束，则 waitpid()不阻塞而立即返回，此时的返回值为 0。
- WUNTRACED：为了实现某种操作，由 pid 指定的任一进程已被暂停，且其状态自暂停以来还未报告过，则返回其状态。
- 0：同 wait()，阻塞父进程，等待子进程退出。

3.3.2 函数功能

wait()会暂时停止目前进程的执行,直到有信号来到或子进程结束。如果在调用 wait()时子进程已经结束，则 wait()会立即返回子进程结束状态值。

waitpid()会暂时停止目前进程的执行，直到有信号来到或子进程结束。如果在调用 waitpid()时子进程已经结束，则 waitpid()会立即返回子进程结束状态值。子进程的结束状态值会由参数 status 返回，而子进程的进程识别码也会一快返回。如果不在意结束状态值，则参数 status 可以设成 NULL。

3.3.3 底层原理

（1）僵尸进程

首先内核会释放终止进程(调用了 exit 系统调用)所使用的所有存储区，关闭所有打开的文件等，但内核为每一个终止子进程保存了一定量的信息。这些信息至少包括进 ID，进程的终止状态,以及该进程使用 CPU 时间,所以当终止子进程的父进程调用 wait 或 waitpid 时就可以得到这些信息。

而僵尸进程就是指：一个进程执行了 exit 系统调用退出，而其父进程并没有为它进行后续处理(调用 wait 或 waitpid 来获得它的结束状态)的进程。

任何一个子进程(init 除外)在 exit 后并非马上就消失，而是留下一个称外僵尸进程的数据结构，等待父进程处理。这是每个子进程都必需经历的阶段。另外子进程退出的时候会向其父进程发送一个 SIGCHLD 信号。

（2）wait 系统调用

进程一旦调用了 wait，就立即阻塞自己，由 wait 自动分析是否当前进程的某个子进程已经退出，如果让它找到了这样一个已经变成僵尸的子进程，wait 就会收集这个子进程的信息，并把它彻底销毁后返回；如果没有找到这样一个子进程，wait 就会一直阻塞在这里，直到有一个出现为止。参数 status 用来保存被收集进程退出时的一些状态，它是一个指向 int 类型的指针。但如果我们对这个子进程是如何死掉的毫不在意，我们就可以设定这个参数为 NULL。

使用下面的宏判断子进程的退出情况:

- WIFEXITED(status): 如果子进程正常结束，该宏返回一个非零值
- WEXITSTATUS(status): 如果 WIFEXITED 非零，该宏返回子进程退出码
- WIFSIGNALED(status): 如果子进程因为捕获信号而终止，返回非零值
- WTERMSIG(status): 如果 WIFSIGNALED 非零，返回信号代码
- WIFSTOPPED(status): 如果子进程被暂停，返回一个非零值
- WSTOPSIG(status): 如果 WIFSTOPPED 非零，返回一个信号代码

3.3.4 代码实例

代码：

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

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

int main(void)
{
    pid_t pid;
    pid = fork();
    if(pid < 0) {
        perror("fork error");
        exit(EXIT_FAILURE);
    }

    if(pid == 0) {
        printf("this is child process\n");
        exit(100);
    }

    int status;
    pid_t ret;
    ret = wait(&status);

    if(ret < 0) {
        perror("wait error");
        exit(EXIT_FAILURE);
    }

    printf("ret = %d pid = %d\n", ret, pid);

    if(WIFEXITED(status))
        printf("child exited normal exit status = %d\n", WEXITSTATUS(status));
    else if(WIFSIGNALED(status))
        printf("child exited abnormal signal number = %d\n",
WTERMSIG(status));
    else if(WIFSTOPPED(status))
        printf("child stopped signal number = %d\n", WSTOPSIG(status));
```

```
    return 0;
}
```

运行截图：

```
ubuntu@VM-0-14-ubuntu:~/learn$ ./wait_sample
this is child process
ret = 30114 pid = 30114
child exited normal exit status = 100
```

参考文献

[1] 鸟哥. 鸟哥的 Linux 私房菜: 基础学习篇[M]. 人民邮电出版社, 2010.

[2] Bovet Daniel P. 深入理解 LINUX 内核. 中国电力出版社, 2004.

[3] Robert Love. LINUX 系统编程. 东南大学出版社, 2009.