



10年口碑积累，成功培养60000多名研发工程师，铸就专业品牌形象

华清远见的企业理念是不仅要良心教育、做专业教育，更要受人尊敬的职业教育。

嵌入式应用程序设计综合教程

作者：华清远见

专业始于专注 卓识源于远见

第3章 Linux 多任务编程

本章目标

通俗地讲，多任务是指用户可以在同一时间运行多个应用程序。像其他主流的操作系统一样，Linux 不仅支持多进程、多线程等多任务处理，而且实现了多种任务间通信机制。

本章主要内容：

- Linux 下多任务机制的介绍；
- 任务、进程、线程的特点以及它们之间的关系；
- 多进程编程；
- 守护进程。

3.1 Linux 下多任务机制的介绍

多任务处理是指用户可以在同一时间内运行多个应用程序，每个正在执行的应用程序被称为一个任务。Linux 就是一个支持多任务的操作系统，比起单任务系统它的功能增强了许多。

多任务操作系统使用某种调度策略支持多个任务并发执行。事实上，（单核）处理器在某一时刻只能执行一个任务。每个任务创建时被分配时间片（几十到上百毫秒），任务执行（占用 CPU）时，时间片递减。操作系统会在当前任务的时间片用完时调度执行其他任务。由于任务会频繁地切换执行，因此给用户多个任务同时运行的感觉。多任务操作系统中通常有 3 个基本概念：任务、进程和线程。

3.1.1 任务

任务是一个逻辑概念，指由一个软件完成的活动，或者是为实现某个目的的一系列操作。通常一个任务是一个程序的一次运行，一个任务包含一个或多个完成独立功能的子任务，这个独立的子任务是进程或者是线程。例如，一个杀毒软件的一次运行是一个任务，目的是从各种病毒的侵害中保护计算机系统，这个任务包含多个独立功能的子任务（进程或线程），包括实时监控功能、定时查杀功能、防火墙功能以及用户交互功能等。任务、进程和线程之间的关系如图 3.1 所示。

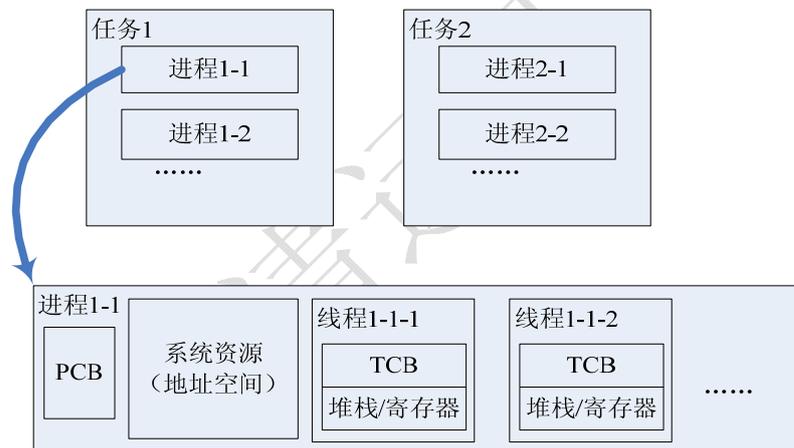


图 3.1 任务、进程和线程之间的关系

3.1.2 进程

1. 进程的基本概念

进程是指一个具有独立功能的程序在某个数据集合上的一次动态执行过程，它是操作系统进行资源分配和调度的基本单元。一次任务的运行可以发激活多个进程，这些进程相互合作来完成该任务的一个最终目标。

进程具有并发性、动态性、交互性和独立性等主要特性。

(1) 并发性：指的是系统中多个进程可以同时并发执行，相互之间不受干扰。

(2) 动态性：指的是进程都有完整的生命周期，而且在进程的生命周期内，进程的状态是不断变化的，另外进程具有动态的地址空间（包括代码、数据和进程控制块等）。

(3) 交互性：指的是进程在执行过程中可能会与其他进程发生直接和间接的通信，如进程同步和进程互斥等，需要为此添加一定的进程处理机制。

(4) 独立性：指的是进程是一个相对完整的资源分配和调度的基本单位，各个进程的地址空间是相互独立的，只有采用某些特定的通信机制才能实现进程之间的通信。

进程和程序是有本质区别的：程序是一段静态的代码，是保存在非易失性存储器上的指令和数据的有序集合，没有任何执行的概念；而进程是一个动态的概念，它是程序的一次执行过程，包括了动态创建、调度、执行和消亡的整个过程，它是程序执行和资源管理的最小单位。

从操作系统的角度看，进程是程序执行时相关资源的总称。当进程结束时，所有资源被操作系统自动回收。

Linux 系统中主要包括下面几种类型的进程。

(1) 交互式进程：这类进程经常与用户进行交互，需要等待用户的输入（键盘和鼠标操作等）。当接收到用户的输入之后，这类进程能够立刻响应。典型的交互式进程有 shell 命令进程、文本编辑器和图形应用程序运行等。

(2) 批处理进程：这类进程不必与用户进行交互，因此通常在后台运行。因为这类进程通常不必很快地响应，因此往往不会优先调度。典型的批处理进程是编译器的编译操作、数据库搜索引擎等。

(3) 守护进程：这类进程一直在后台运行，和任何终端都不关联。通常系统启动时开始执行，系统关闭时才结束。很多系统进程（各种服务）都是以守护进程的形式存在。

2. Linux 下的进程结构

进程不但包括程序的指令和数据，而且包括程序计数器和处理器的所有寄存器以及存储临时数据的进程堆栈。

因为 Linux 是一个多任务的操作系统，所以其他的进程必须等到操作系统将处理器使用权分配给自己之后才能运行。当正在运行的进程需要等待其他的系统资源时，Linux 内核将取得处理器的控制权，按照某种调度算法将处理器分配给某个正在等待执行的进程。

内核将所有进程存放在双向循环链表（进程链表）中，链表的每一项都是 `task_struct`，称为进程控制块的结构。该结构包含了与一个进程相关的所有信息，在 `<include/Linux/sched.h>` 文件中定义。`task_struct` 内核结构比较大，它能完整地描述一个进程，如进程的状态、进程的基本信息、进程标识符、内存相关信息、父进程相关信息、与进程相关的终端信息、当前工作目录、打开的文件信息、所接收的信号信息等。

下面详细讲解 `task_struct` 结构中最为重要的两个域：`state`（进程状态）和 `pid`（进程标识符）。

(1) 进程状态。

Linux 中的进程有以下几种主要状态。

① 运行状态 (`TASK_RUNNING`)：进程当前正在运行，或者正在运行队列中等待调度。

② 可中断的阻塞状态 (`TASK_INTERRUPTIBLE`)：进程处于阻塞（睡眠）状态，正在等待某些事件发生或能够占用某些资源。处在这种状态下的进程可以被信号中断。接收到信号或被显式地唤醒呼叫（如调用 `wake_up` 系列宏：`wake_up`、`wake_up_interruptible` 等）唤醒之后，进程将转变为 `TASK_RUNNING` 状态。

③ 不可中断的阻塞状态 (`TASK_UNINTERRUPTIBLE`)：此进程状态类似于可中断的阻塞状态 (`TASK_INTERRUPTIBLE`)，只是它不会处理信号，把信号传递到这种状态下的进程不能改变它的状态。在一些特定的情况下（进程必须等待，直到某些不能被中断的事件发生），这种状态是很有用的。只有在它所等待的事件发生时，进程才被显式地唤醒呼叫唤醒。

④ 暂停状态 (`TASK_STOPPED`)：进程的执行被暂停，当进程收到 `SIGSTOP`、`SIGTSTP`、`SIGTTIN`、`SIGTTOU` 等信号，就会进入暂停状态。

⑤ 僵死状态 (`EXIT_ZOMBIE`)：子进程运行结束，父进程尚未使用 `wait` 函数族（如使用 `waitpid()` 函数）等系统调用来回收退出状态。处在该状态下的子进程已经放弃了几乎所有的内存空间，没有任何可执行代码，也不能被调度，仅仅在进程列表中保留一个位置，记载该进程的退出状态等信息供其父进程收集。

⑥ 消亡状态 (`EXIT_DEAD`)：这是最终状态，父进程调用 `wait` 函数族回收之后，子进程彻底由系统删除。

它们之间的转换关系如图 3.2 所示。

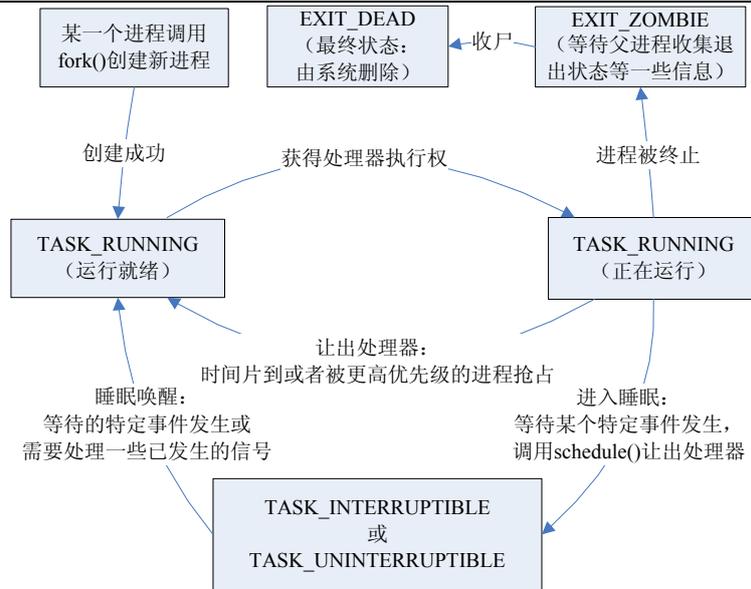


图 3.2 进程状态转换关系图

内核可以使用 `set_task_state` 和 `set_current_state` 宏来改变指定进程的状态和当前执行进程的状态。

(2) 进程标识符。

Linux 内核通过唯一的进程标识符 PID 来标识每个进程。PID 存放在 `task_struct` 的 `pid` 字段中。系统中可以创建的进程数目有限制，读者可以查看 `/proc/sys/kernel/pid_max` 来确定上限。

当系统启动后，内核通常作为某一个进程的代表。一个指向 `task_struct` 的宏 `current` 用来记录正在运行的进程。`current` 经常作为进程描述符结构指针的形式出现在内核代码中，例如，`current->pid` 表示处理器正在执行的进程的 PID。当系统需要查看所有的进程时，则调用 `for_each_process()` 宏，这 will 比系统搜索数组的速度要快得多。

在 Linux 中获得当前进程的进程号 (PID) 和父进程号 (PPID) 的系统调用函数分别为 `getpid()` 和 `getppid()`。

3. 进程的创建、执行和终止

(1) 进程的创建和执行。

许多操作系统都提供的是产生进程的机制，也就是首先在新的地址空间里创建进程、读入可执行文件，最后再开始执行。Linux 中进程的创建很特别，它把上述步骤分解到两个单独的函数中取执行：`fork()` 和 `exec` 函数族。首先，`fork()` 通过复制当前进程创建一个子进程，子进程与父进程的区别仅仅在于不同的 PID、PPID 和某些资源及统计量。`exec` 函数族负责读取可执行文件并将其载入地址空间开始运行。

要注意的是，Linux 中的 `fork()` 使用的是写时复制 (copy on write) 的技术，也就是内核在创建进程时，其资源并没有立即被复制过来，而是被推迟到需要写入数据的时候才发生。在此之前只是以只读的方式共享父进程的资源。写时复制技术可以使 Linux 拥有快速执行的能力，因此这个优化是非常重要的。

(2) 进程的终止。

进程终止也需要做很多烦琐的收尾工作，系统必须保证进程所占用的资源回收，并通知父进程。Linux 首先把终止的进程设置为僵死状态。这个时候，进程已经无法运行。它的存在只为父进程提供信息。父进程在某个时间调用 `wait` 函数族，回收子进程的退出状态，随后子进程占用的所有资源被释放。

4. 进程的内存结构

Linux 操作系统采用虚拟内存管理技术，使得每个进程都有独立的地址空间。该地址空间是大小为 4GB 的线性虚拟空间，用户所看到和接触到的都是该虚拟地址，无法看到实际的物理内存地址。利用这种虚拟地址不但更安全 (用户不能直接访问物理内存)，而且用户程序可以使用比实际物理内存更大的地址空间。

4GB 的进程地址空间会被分成两个部分——用户空间与内核空间。用户地址空间是 0~3GB (0xC0000000)，内核地址空间占据 3~4GB。用户进程在通常情况下只能访问用户空间的虚拟地址，不能访问内核空间虚拟地址。只有用户进程使用系统调用（代表用户进程在内核态执行）时才可以访问到内核空间。每当进程切换，用户空间就会跟着变化；而内核空间是由内核负责映射，它并不会跟着进程改变，是固定的。内核空间地址有自己对应的页表，用户进程各自有不同的页表。每个进程的用户空间都是完全独立、互不相干的。进程的虚拟内存地址空间如图 3.3 所示。



图 3.3 进程地址空间的分布

用户空间包括以下几个功能区域。

(1) 只读段：包含程序代码 (.init 和 .text) 和只读数据 (.rodata)。

(2) 数据段：存放的是全局变量和静态变量。其中可读可写数据段 (.data) 存放已初始化的全局变量和静态变量，BSS 数据段 (.bss) 存放未初始化的全局变量和静态变量。

(3) 栈：由系统自动分配释放，存放函数的参数值、局部变量的值、返回地址等。

(4) 堆：存放动态分配的数据，一般由程序员动态分配和释放，若程序员不释放，程序结束时可能由操作系统回收。

(5) 共享库的内存映射区域：这是 Linux 动态链接器和其他共享库代码的映射区域。

因为在 Linux 系统中每一个进程都会有“/proc”文件系统下的与之对应的一个目录（例如，init 进程的相关信息存放在“/proc/1”目录下），因此通过 proc 文件系统可以查看某个进程的地址空间的映射情况。例如，运行一个应用程序，如果它的进程号为 13703，则输入“cat /proc/13703/maps”命令，可以查看该进程的内存映射情况。

```
$ cat /proc/13703/maps
/* 只读段：代码段、只读数据段 */
08048000-08049000 r-xp 00000000 08:01 876817 /home/Linux/test
08049000-0804a000 r--p 00000000 08:01 876817 /home/Linux/test
/* 可读写数据段 */
0804a000-0804b000 rw-p 00001000 08:01 876817 /home/Linux/test
0804b000-0804c000 rw-p 0804b000 00:00 0
08502000-08523000 rw-p 08502000 00:00 0 5 [heap] /* 堆 */
b7dec000-b7ded000 rw-p b7dec000 00:00 0
/* 动态共享库 */
b7ded000-b7f45000 r-xp 00000000 08:01 541691
```


因为子进程几乎是父进程的完全复制，所以父子两个进程会运行同一个程序。因此需要用一种方式来区分它们，并使它们照此运行，否则，这两个进程只能做相同的事。

父子进程一个很重要的区别是：`fork()` 的返回值不同。父进程中的返回值是子进程的进程号，而子进程中返回 0。可以通过返回值来判定该进程是父进程还是子进程。

注意：子进程没有执行 `fork()` 函数，而是从 `fork()` 函数调用的下一条语句开始执行。

(2) `fork()` 函数语法。

表 3.1 列出了 `fork()` 函数的语法要点。

表 3.1 `fork()` 函数语法要点

所需头文件	<code>#include <sys/types.h> /* 提供类型 pid_t 的定义 */</code> <code>#include <unistd.h></code>
函数原型	<code>pid_t fork (void);</code>
函数返回值	0: 子进程
	子进程 PID (大于 0 的整数): 父进程
	-1: 出错

`fork()` 函数的简单的示例程序如下。

```
int main (void)
{
    pid_t ret;

    /*调用 fork() 函数*/
    ret = fork();
    /*通过 ret 的值来判断 fork() 函数的返回情况，首先进行出错处理*/
    if (ret == -1)
    {
        perror ("fork error");
        return -1;
    }
    else if (ret == 0) /*返回值为 0 代表子进程*/
    {
        printf ("In child process!! ret is %d, My PID is %d\n", ret, getpid());
    }
    else /*返回值大于 0 代表父进程*/
    {
        printf ("In parent process!! ret is %d, My PID is %d\n", ret, getpid());
    }

    return 0;
}
```

编译并执行程序，结果如下。

```
$ gcc fork.c -o fork -Wall
$ ./fork
In parent process!! ret is 3876, My PID is 3875
In child process!! ret is 0, My PID is 3876
```

从该示例中可以看出，使用 `fork()` 函数新建了一个子进程，其中的父进程返回子进程的进程号，而子进程的返回值为 0。

2. exec 函数族

(1) exec 函数族说明。

`fork()` 函数用于创建一个子进程，该子进程几乎复制了父进程的全部内容。我们能否让子进程执行一个新的程序呢？`exec` 函数族就提供了一个在进程中执行另一个程序的方法。它可以根据指定的文件名或目录名找到可执行文件，并用它来取代当前进程的数据段、代码段和堆栈段。在执行完之后，当前进程除了进程号外，其他内容都被替换了。这里的可执行文件既可以是二进制文件，也可以是 Linux 下任何可执行的脚本文件。

在 Linux 中使用 `exec` 函数族主要有两种情况。

① 当进程认为自己不能再为系统和用户做出任何贡献时，就可以调用 `exec` 函数族中的任意一个函数让自己重生。

② 如果一个进程想执行另一个程序，那么它就可以调用 `fork()` 函数新建一个进程，然后调用 `exec` 函数族中的任意一个函数，这样看起来就像通过执行应用程序而产生了一个新进程（这种情况非常普遍）。

(2) exec 函数族语法。

实际上，在 Linux 中并没有 `exec()` 函数，而是有 6 个以 `exec` 开头的函数，它们之间语法有细微差别，本书在下面会详细讲解。

表 3.2 所示为 `exec` 函数族的 6 个成员函数的语法。

表 3.2 exec 函数族成员函数语法

所需头文件	<code>#include <unistd.h></code>
函数原型	<code>int execl (const char *path, const char *arg, ...);</code>
	<code>int execv (const char *path, char *const argv[]);</code>
	<code>int execl_e (const char *path, const char *arg, ..., char *const envp[]);</code>
	<code>int execve (const char *path, char *const argv[], char *const envp[]);</code>
	<code>int execlp (const char *file, const char *arg, ...);</code>
	<code>int execvp (const char *file, char *const argv[]);</code>
函数返回值	-1: 出错

这 6 个函数在函数名和使用语法的规则上都有细微的区别，下面就可执行文件查找方式、参数表传递方式及环境变量这几个方面进行比较。

① 查找方式。读者可以注意到，表 3.2 中的前 4 个函数的查找方式都是完整的文件目录路径，而最后 2 个函数（也就是以 p 结尾的两个函数）可以只给出文件名，系统就会自动按照环境变量“PATH”所包含的路径进行查找。

② 参数传递方式。exec 函数族的参数传递方式有两种：一种是逐个列举的方式，另一种是将所有参数通过指针数组传递。在这里是以函数名的第 5 位字母来区分的，字母为 l (list) 的表示逐个列举参数的方式，其类型为 `const char *arg`；字母为 v (vector) 的表示通过指针数组传递，其类型为 `char *const argv[]`。读者可以观察 `execl()`、`execle()`、`execlp()` 的语法与 `execv()`、`execve()`、`execvp()` 的区别。它们具体的用法在后面的实例讲解中会具体说明。

这里的参数实际上就是用户在使用这个可执行文件时所需的全部命令选项字符串（包括该可执行程序命令本身）。要注意的是，这些参数必须以 NULL 结尾。

③ 环境变量。exec 函数族可以使用默认的环境变量，也可以传入指定的环境变量。这里以 e (environment) 结尾的两个函数 `execle()` 和 `execve()` 就可以在 `envp[]` 中指定当前进程所使用的环境变量。

表 3.3 对这 4 个函数中函数名和对应语法做一个小结，主要指出了函数名中每一位所表明的含义，希望读者结合此表加以记忆。

表 3.3 exec 函数名对应含义

前 4 位	统一为 exec	
第 5 位	l: 参数传递为逐个列举方式	<code>execl</code> 、 <code>execle</code> 、 <code>execlp</code>
	v: 参数传递为构造指针数组方式	<code>execv</code> 、 <code>execve</code> 、 <code>execvp</code>
第 6 位	e: 可传递新进程环境变量	<code>execle</code> 、 <code>execve</code>
	p: 可执行文件查找方式为文件名	<code>execlp</code> 、 <code>execvp</code>

事实上，这 6 个函数中真正的系统调用只有 `execve()`，其他 5 个都是库函数，它们最终都会调用 `execve()` 这个系统调用。在使用 exec 函数族时，一定要加上错误判断语句。exec 很容易执行失败，其中最常见的原因如下。

- 找不到文件或路径，此时 `errno` 被设置为 `ENOENT`。
- 数组 `argv` 和 `envp` 忘记用 NULL 结束，此时 `errno` 被设置为 `EFAULT`。
- 没有对应可执行文件的运行权限，此时 `errno` 被设置为 `EACCES`。

(3) exec 使用实例。

下面的第一个示例说明了如何使用文件名的方式来查找可执行文件，同时使用参数列表的方式，这里用的函数是 `execlp()`。

```

/*execlp.c*/
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main()
    
```

```

{
    if (fork() == 0)
    {
        /*调用 execlp() 函数，这里相当于调用了“ps -ef”命令*/
        if ((ret = execlp ("ps", "ps", "-ef", NULL)) < 0)
        {
            printf ("execlp error\n");
        }
    }
}
    
```

在该程序中，首先使用 fork() 函数创建一个子进程，然后在子进程里使用 execlp() 函数。读者可以看到，这里的参数列表列出了要执行的程序名和选项。运行结果如下。

```

$ ./execlp
PID TTY      Uid        Size State Command
  1   root      root        1832  S    init
  2   root      root         0     S    [keventd]
  3   root      root         0     S    [ksoftirqd_CPU0]
  4   root      root         0     S    [kswapd]
  5   root      root         0     S    [bdflush]
  6   root      root         0     S    [kupdated]
  7   root      root         0     S    [mtdblockd]
  8   root      root         0     S    [khubd]
 35   root      root        2104  S    /bin/bash /usr/etc/rc.local
 36   root      root        2324  S    /bin/bash
 41   root      root        1364  S    /sbin/inetd
 53   root      root       14260  S    /Qtopia/qtopia-free-1.7.0/bin/qpe -qws
 54   root      root       11672  S    quicklauncher
 65   root      root         0     S    [usb-storage-0]
 66   root      root         0     S    [scsi_ah_0]
 83   root      root        2020  R    ps -ef
    
```

此程序的运行结果与在 shell 中直接输入命令 “ps -ef” 是一样的。

接下来的示例使用完整的文件目录来查找对应的可执行文件。注意目录必须以 “/” 开头，否则将其视为文件名。

```

/*execl.c*/
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    if (fork() == 0)
    {
        /*调用 execl() 函数，注意这里要给出 ps 程序所在的完整路径*/
    }
}
    
```

```

        if (execl ("/bin/ps", "ps", "-ef", NULL) < 0)
        {
            printf ("execl error\n");
        }
    }
}

```

下面的示例利用函数 `execl()` 将环境变量添加到新建的子进程中，这里的 `env` 是查看当前进程环境变量的命令。

```

/* execl.c */
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    /*命令参数列表，必须以 NULL 结尾*/
    char *envp[]={"PATH=/tmp", "USER=harry", NULL};

    if (fork() == 0)
    {
        /*调用 execl()函数，注意这里也要指出 env 的完整路径*/
        if (execl ("/usr/bin/env", "env", NULL, envp) < 0)
        {
            printf ("execl error\n");
        }
    }
}

```

程序的运行结果如下。

```

$ ./execl
PATH=/tmp
USER=harry

```

最后一个示例使用 `execve()` 函数，通过构造指针数组的方式来传递参数，注意参数列表一定要以 `NULL` 作为结尾标识符。其代码和运行结果如下。

```

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

int main()
{
    /*命令参数列表，必须以 NULL 结尾*/
    char *arg[] = {"env", NULL};
    char *envp[] = {"PATH=/tmp", "USER=harry", NULL};

    if (fork() == 0)
    {
        if (execve ("/usr/bin/env", arg, envp) < 0)

```

```

    {
        printf("execve error\n");
    }
}
}

```

程序的运行结果如下。

```

$ ./execve
PATH=/tmp
USER=harry

```

3. exit()和_exit()

(1) exit()和_exit()函数说明。

exit()和_exit()函数都是用来终止进程的。当程序执行到exit()或_exit()时，进程会无条件地停止剩下的所有操作，清除各种数据结构，并终止本进程的运行。但是，这两个函数还是有区别的，这两个函数的调用过程如图3.5所示。

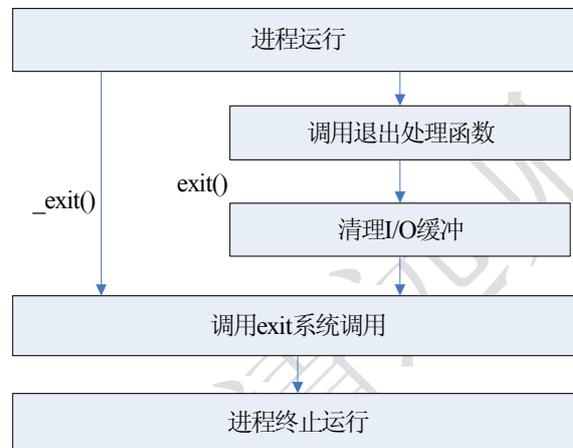


图 3.5 exit()和_exit()函数流程图

从图3.5中可以看出，_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；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()函数与_exit()函数最大的区别就在于exit()函数在终止当前进程之前要检查该进程打开了哪些文件，并把文件缓冲区中的内容写回文件，就是图中的“清理I/O缓冲”一项。

由于在Linux的标准函数库中，有一种被称为“缓冲I/O (buffered I/O)”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，会连续读出若干条记录，这样在下次读文件时就可以直接从内存的缓冲区中读取；同样，每次写文件的时候，也仅仅是写入内存中的缓冲区，等满足了一定的条件（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，但也为编程带来了一些麻烦。比如有些数据，认为已经被写入文件中，实际上因为没有满足特定的条件，它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，若想保证数据的完整性，最好使用exit()函数。

(2) exit()和_exit()函数语法。

表3.4所示为exit()和_exit()函数的语法规范。

表 3.4 exit()和_exit()函数族语法

所需头文件	exit: #include <stdlib.h>
	_exit: #include <unistd.h>

函数原型	exit: void exit (int status); <hr/> _exit: void _exit (int status);
函数传入值	status 是一个整型的参数，可以利用这个参数传递进程结束时的状态。一般来说，0 表示正常结束；其他的数值表示出现了错误，进程非正常结束 在实际编程时，可以用 wait() 系统调用接收子进程的返回值，从而针对不同的情况进行不同的处理

(3) exit() 和 _exit() 使用示例。

这两个示例比较了 exit() 和 _exit() 两个函数的区别。由于标准输出流 stdout 是行缓冲，遇到“\n”换行符时才会实际写入终端。示例中就是利用这个性质来进行比较的。示例 1 的代码如下。

```

/* exit.c */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    printf("Using exit...\n");
    printf("This is the content in buffer");
    exit(0);
}
    
```

```

$ ./exit
Using exit...
This is the content in buffer $
    
```

读者从输出的结果中可以看到，调用 exit() 函数时，缓冲区中的内容也能正常输出。示例 2 的代码如下。

```

/* _exit.c */
#include <stdio.h>
#include <unistd.h>

int main()
{
    printf("Using _exit...\n");
    printf("This is the content in buffer"); /* 加上回车符之后结果又如何 */
    _exit(0);
}
    
```

```

$ ./_exit
Using _exit...
$
    
```

可以看到，调用 _exit() 函数，进程结束时没有输出缓冲区中的内容。

4. wait() 和 waitpid()

(1) wait() 和 waitpid() 函数说明。

wait() 函数用于使父进程（也就是调用 wait() 的进程）阻塞，直到一个子进程结束或者该进程收到了一个指定的信号为止。如果该父进程没有子进程或者他的子进程已经结束，则 wait() 会立即返回-1。

waitpid()的作用和wait()一样,但它并不一定等待第一个终止的子进程。waitpid()有若干选项,可提供一个非阻塞版本的wait()功能。实际上wait()函数只是waitpid()函数的一个特例,在Linux内部实现wait()函数时直接调用的就是waitpid()函数。

(2) wait()和waitpid()函数格式说明。

表3.5所示为wait()函数的语法规范。

表 3.5 wait()函数族语法

所需头文件	#include <sys/types.h> #include <sys/wait.h>
函数原型	pid_t wait (int *status);
函数传入值	status 指向的整型对象用来保存子进程结束时的状态。另外,子进程的结束状态可由Linux中一些特定的宏来测定
函数返回值	成功: 已回收的子进程的进程号 失败: -1

表3.6所示为waitpid()函数的语法规范。

表 3.6 waitpid()函数语法

所需头文件	#include <sys/types.h> #include <sys/wait.h>
函数原型	pid_t waitpid (pid_t pid, int *status, int options);
函数传入值	pid pid > 0: 回收进程ID等于pid的子进程 pid = -1: 回收任何一个子进程,此时和wait()作用一样 pid = 0: 回收其组ID等于调用进程的组ID的任一子进程 pid < -1: 回收其组ID等于pid的绝对值的任一子进程
函数传入值	status 同wait() options WNOHANG: 若指定的子进程没有结束,则waitpid()不阻塞而立即返回,此时返回值为0 WUNTRACED: 为了实现某种操作,由pid指定的任一子进程已被暂停,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,直到指定的子进程退出
函数返回值	>0: 已经结束运行的子进程的进程号 0: 使用选项WNOHANG且没有子进程退出

-1: 出错

(3) waitpid() 使用示例。

由于 wait() 函数的使用较为简单，在此以 waitpid() 为例进行讲解。本例中首先使用 fork() 创建一个子进程，然后让子进程暂停 5s (使用了 sleep() 函数)。接下来对原有的父进程使用 waitpid() 函数，并使用参数 WNOHANG 使该父进程不会阻塞。若有子进程退出，则 waitpid() 返回子进程号；若没有子进程退出，则 waitpid() 返回 0，并且父进程每隔 1s 循环判断一次。该程序的流程图如图 3.6 所示。

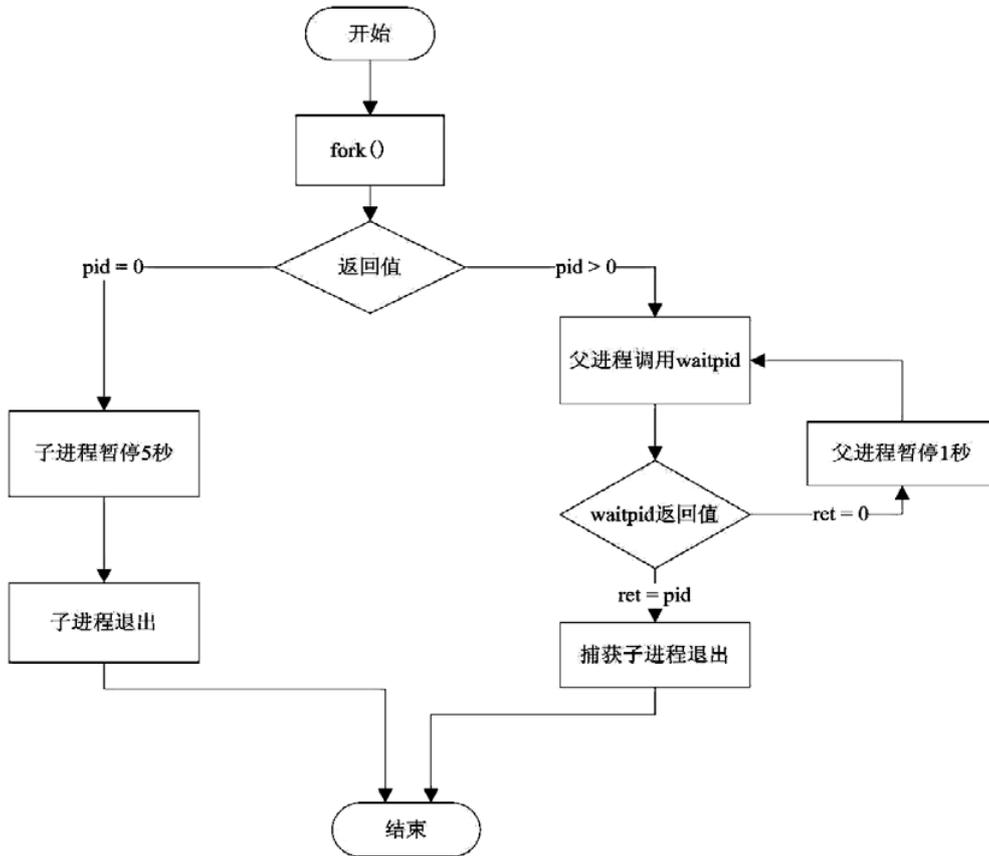


图 3.6 waitpid 示例程序流程图

该程序源代码如下。

```

/* waitpid.c */
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    pid_t pid, ret;

    if ((pid = fork()) < 0)
    {
        printf ("Error fork\n");
    }
}
  
```

```

else if (pid == 0) /*子进程*/
{
    /*子进程暂停 5s*/
    sleep (5) ;
    /*子进程正常退出*/
    exit (0) ;
}
else /*父进程*/
{
    /*循环测试子进程是否退出*/
    do
    {
        /*调用 waitpid, 且父进程不阻塞*/
        ret = waitpid (pid, NULL, WNOHANG) ;

        /*若子进程还未退出, 则父进程暂停 1s*/
        if (ret == 0)
        {
            printf ("The child process has not exited\n") ;
            sleep (1) ;
        }
    } while (ret == 0) ;

    /*若发现子进程退出, 打印出相应情况*/
    if (pid == ret)
    {
        printf ( "child process exited\n" ) ;
    }
    else
    {
        printf ("some error occured.\n") ;
    }
}
}
    
```

程序运行结果如下。

```

$ ./waitpid
The child process has not exited
child process exited
    
```

如果把 `ret = waitpid (pid, NULL, WNOHANG) ;` 改为 `ret = waitpid (pid, NULL, 0) ;` 父进程会一直阻塞, 直到子进程结束为止, 运行的结果如下。

```

$ ./waitpid
child process exited
    
```

3.2.2 Linux 守护进程

1. 守护进程概述

守护进程也就是通常所说的 Daemon 进程，它是 Linux 中的后台服务进程。它是一个生存期较长的进程，通常独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。守护进程常常在系统启动时开始执行，在系统关闭时终止。Linux 中很多系统服务都是通过守护进程实现的。

由于在 Linux 中，每一个系统与用户进行交流的界面称为终端。每一个从此终端开始运行的进程都会依附于该终端，这个终端称为这些进程的控制终端。当控制终端关闭时，相应的进程都会自动结束。但是守护进程却能够突破这种限制，不受终端关闭的影响。反之，如果希望某个进程不因为用户、终端或者其他的变化而受到影响，那么就必须把这个进程变成一个守护进程。

2. 编写守护进程

编写守护进程看似复杂，但实际上也是遵循一个特定的流程。下面就分 5 个步骤来讲解怎样创建一个简单的守护进程。在讲解的同时，会同时介绍与创建守护进程相关的概念和函数，希望读者能很好地掌握。

(1) 创建子进程，父进程退出。

这是编写守护进程的第一步。由于守护进程是脱离控制终端的，因此，完成第一步后子进程变成后台进程，给用户感觉程序已经运行完毕。之后的所有工作都在子进程中完成，而用户通过 shell 可以执行其他的命令，从而在形式上做到了与控制终端的脱离。

到这里，有心的读者可能会问，父进程创建了子进程之后退出，此时该子进程不就没有父进程了吗？守护进程中确实会出现这么一个有趣的现象。由于父进程已经先于子进程退出，会造成子进程没有父进程，从而变成一个孤儿进程。在 Linux 中，每当系统发现一个孤儿进程，就会自动由 1 号进程（也就是 init 进程）收养它，这样，原先的子进程就会变成 init 进程的子进程了。其实现代码如下。

```
pid = fork();
if (pid > 0)
{
    exit (0); /*父进程退出*/
}
```

(2) 在子进程中创建新会话。

这个步骤是创建守护进程中最重要的一步，虽然它的实现非常简单，但它的意义却非常重大。在这里使用的是函数 setsid()。在具体介绍 setsid() 之前，读者首先要了解两个概念：进程组和会话期。

① 进程组。进程组是一个或多个进程的集合。进程组由进程组 ID 来唯一标识。除了进程号 (PID) 之外，进程组 ID 也是一个进程的必备属性。

每个进程组都有一个组长进程，其组长进程的进程号等于进程组 ID。且进程组 ID 不会因组长进程的退出而受到影响。

② 会话期。会话组是一个或多个进程组的集合。通常，一个会话开始于用户登录，终止于用户退出；或者开始于终端打开，结束于终端关闭。会话期的第一个进程称为会话组长。在此期间该用户运行的所有进程都属于这个会话期，它们之间的关系如图 3.7 所示。

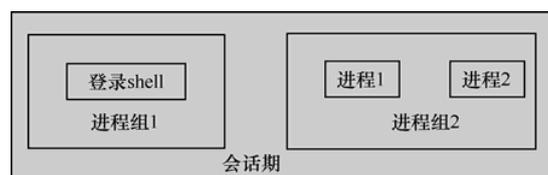


图 3.7 进程组和会话期之间的关系图

接下来就可以具体介绍 setsid() 的相关内容。

(1) setsid() 函数作用。

setsid() 函数用于创建一个新的会话，并担任该会话组的组长。调用 setsid() 有下面的 3 个作用。

- ① 让进程摆脱原会话的控制。
- ② 让进程摆脱原进程组的控制。
- ③ 让进程摆脱原控制终端的控制。

那么，在创建守护进程时为什么要调用 setsid() 函数呢？读者可以回忆一下创建守护进程的第一步，在那里调用了 fork() 函数来创建子进程再令父进程退出。由于在调用 fork() 函数时，子进程全盘复制了父进程的会话期、进程组和控制终端等。虽然父进程退出了，但原先的会话期、进程组和控制终端等并没有改变，因此，还不是真正意义上的独立，而 setsid() 函数能够使进程完全独立出来，从而脱离所有其他进程的控制。

(2) setsid() 函数格式。

表 3.7 所示为 setsid() 函数的语法规范。

表 3.7 setsid() 函数语法

所需头文件	#include <sys/types.h> #include <unistd.h>
函数原型	pid_t setsid (void);
函数返回值	成功：该进程组 ID
	出错：-1

(3) 改变当前目录。

这一步也是必要的步骤。使用 fork() 创建的子进程继承了父进程的当前工作目录。由于在进程运行过程中，当前目录所在的文件系统（比如“/mnt/usb”等）是不能卸载的，这对以后的使用会造成诸多的麻烦（比如系统由于某种原因要进入单用户模式）。因此，通常的做法是让“/”作为守护进程的当前工作目录，这样就可以避免上述的问题。当然，如有特殊需要，也可以把当前工作目录换成其他的路径，如“/tmp”。改变工作目录的函数是 chdir()。

(4) 重设文件权限掩码。

文件权限掩码（通常用 8 进制表示）的作用是屏蔽文件权限中的对应位。例如，如果文件权限掩码是 050，它表示屏蔽了文件组拥有者的可读与可执行权限。由于使用 fork() 函数新建的子进程继承了父进程的文件权限掩码，这就给该子进程使用文件带来了一定的影响。因此，把文件权限掩码设置为 0，可以增强该守护进程的灵活性。设置文件权限掩码的函数是 umask()。在这里，通常的使用方法为 umask (0)。

(5) 关闭文件描述符。

同文件权限掩码一样，用 fork() 函数新建的子进程会从父进程那里继承一些已经打开了的文件。这些被打开的文件可能永远不会被守护进程访问，但它们一样占用系统资源，而且还可能导致所在的文件系统无法被卸载。

特别是守护进程和终端无关，所以指向终端设备的标准输入、标准输出和标准错误流已经失去了存在的价值，应当被关闭。通常按如下方式关闭文件描述符。

```
int num;
num = getdtablesize(); // 获取当前进程文件描述符表大小
for (i = 0; i < num; i++)
{
    close (i);
}
```

这样，一个简单的守护进程就建立起来了，创建守护进程的流程图如图 3.8 所示。

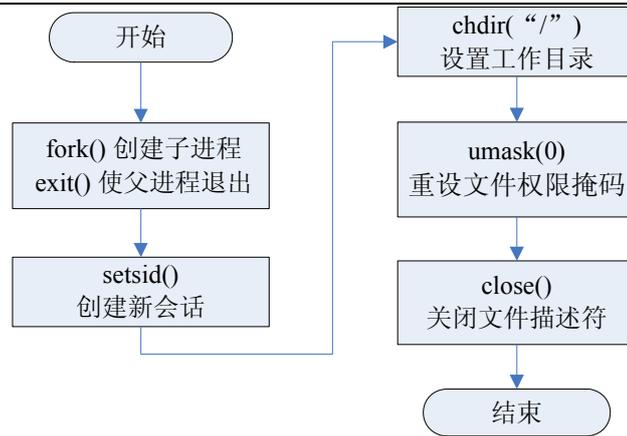


图 3.8 创建守护进程流程图

下面是实现守护进程的一个完整示例，该示例首先按照以上的创建流程建立了一个守护进程，然后让该守护进程每隔 2s 向日志文件 “/tmp/daemon.log” 写入字符串。

```

/* daemon.c 创建守护进程实例 */
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<fcntl.h>
#include<sys/types.h>
#include<unistd.h>
#include<sys/wait.h>

int main()
{
    pid_t pid;
    int i, fd;
    char *buf = "This is a Daemon\n";

    pid = fork(); /* 第一步 */
    if (pid < 0)
    {
        printf("Error fork\n");
        exit(1);
    }
    else if (pid > 0)
    {
        exit(0); /* 父进程推出 */
    }

    setsid(); /*第二步*/
    chdir("/tmp"); /*第三步*/
    umask(0); /*第四步*/
    for (i = 0; i < getdtablesize(); i++) /*第五步*/
    {
        close(i);
    }
}
  
```

```

/*这时创建完守护进程，以下开始正式进入守护进程工作*/
while (1)
{
    if ((fd = open ("daemon.log", O_CREAT|O_WRONLY|O_TRUNC, 0600)) < 0)
    {
        printf ("Open file error\n");
        exit (1);
    }
    write (fd, buf, strlen (buf));
    close (fd);
    sleep (2);
}
exit (0);
}
    
```

程序运行时每隔 2s 就会在对应的文件中写入字符串。使用 ps 可以看到该进程在后台运行，命令如下。

```

$ tail -f /tmp/daemon.log
This is a Daemon
This is a Daemon
This is a Daemon
This is a Daemon
...
$ ps -ef|grep daemon
    76          root          1272  S   ./daemon
    85          root          1520  S   grep daemon
    
```

3. 守护进程的出错处理

读者在前面编写守护进程的具体调试过程中会发现，由于守护进程完全脱离了控制终端，因此，不能像其他普通进程一样将错误信息输出到控制终端。那么，守护进程要如何调试呢？一种通用的办法是使用 syslog 服务，将程序中的出错信息输入到系统日志文件中（如“/var/log/messages”），从而可以直观地看到程序的问题所在（“/var/log/messages”系统日志文件只能由拥有 root 权限的超级用户查看。在不同 Linux 发行版本中，系统日志文件路径全名可能有所不同，例如，可能是“/var/log/syslog”）。

syslog 是 Linux 中的系统日志管理服务，通过守护进程 syslogd 来维护。该守护进程在启动时会读一个配置文件“/etc/syslog.conf”。该文件决定了不同类型的消息会发送向何处。例如，紧急消息可被送向系统管理员并在控制台上显示，而警告消息则可被记录到一个文件中。

该机制提供了 3 个 syslog 相关函数，分别为 openlog()、syslog() 和 closelog()。下面就分别介绍这 3 个函数。

(1) 相关函数说明。

通常，openlog() 函数用于打开系统日志服务的一个连接；syslog() 函数用于向日志文件中写入消息，在这里可以规定消息的优先级、消息输出格式等；closelog() 函数用于关闭系统日志服务的连接。

(2) 相关函数格式。

表 3.8 所示为 openlog() 函数的语法规范。

表 3.8 openlog() 函数语法

所需头文件	#include <syslog.h>
函数原型	void openlog (char *ident, int option, int facility);

函数传入值	ident	要向每个消息加入的字符串，通常为程序的名称
	option	LOG_CONS: 如果消息无法送到系统日志服务，则直接输出到系统控制终端
		LOG_NDELAY: 立即打开系统日志服务的连接。在正常情况下，直接发送到第一条消息时才打开连接
		LOG_PID: 在每条消息中包含进程的PID
函数传入值	facility: 指定程序发送的消息类型	LOG_AUTHPRIV: 安全/授权信息
		LOG_CRON: 时间守护进程 (cron 及 at)
		LOG_DAEMON: 其他系统守护进程
		LOG_KERN: 内核信息
		LOG_LOCAL[0~7]: 保留
		LOG_LPR: 行打印机子系统
		LOG_MAIL: 邮件子系统
		LOG_NEWS: 新闻子系统
		LOG_SYSLOG: syslogd 内部所产生的信息
LOG_USER: 一般使用者等级信息		
LOG_UUCP: UUCP 子系统		

表 3.9 所示为 syslog() 函数的语法规范。

表 3.9 syslog() 函数语法

所需头文件	#include <syslog.h>	
函数原型	void syslog (int priority, char *format, ...);	
函数传入值	priority: 指定消息的重要性	LOG_EMERG: 系统无法使用
		LOG_ALERT: 需要立即采取措施
		LOG_CRIT: 有重要情况发生
		LOG_ERR: 有错误发生
		LOG_WARNING: 有警告发生
		LOG_NOTICE: 正常情况，但也是重要情况
		LOG_INFO: 信息消息
	LOG_DEBUG: 调试信息	
format	以字符串指针的形式表示输出的格式，类似于 printf 中的格式	

表 3.10 所示为 closelog() 函数的语法规范。

表 3.10 closelog() 函数语法

所需头文件	#include <syslog.h>
函数原型	void closelog (void);

(3) 使用示例。

这里将上一节中的示例程序用 syslog 服务进行重写，其中有区别的地方用加粗的字体表示，源代码如下。

```

/* syslog_daemon.c 利用 syslog 服务的守护进程实例 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <syslog.h>

int main()
{
    pid_t pid, sid;
    int i, fd;
    char *buf = "This is a Daemon\n";

    pid = fork(); /* 第一步 */
    if (pid < 0)
    {
        printf("Error fork\n");
        exit(1);
    }
    else if (pid > 0)
    {
        exit(0); /* 父进程退出 */
    }
    /* 打开系统日志服务, openlog */
    openlog("daemon_syslog", LOG_PID, LOG_DAEMON);
    if ((sid = setsid()) < 0) /*第二步*/
    {
        syslog(LOG_ERR, "%s\n", "setsid");
        exit(1);
    }
    if ((sid = chdir("/")) < 0) /*第三步*/
    {
        syslog(LOG_ERR, "%s\n", "chdir");
        exit(1);
    }
    umask(0); /*第四步*/
    for (i = 0; i < getdtablesize(); i++) /*第五步*/
    {
        close(i);
    }
    /*这时创建完守护进程, 以下开始正式进入守护进程工作*/
    while (1)
    {

```

```

if ((fd = open ("/tmp/daemon.log", O_CREAT|O_WRONLY|O_APPEND, 0600)) < 0)
{
    syslog (LOG_ERR, "open");
    exit (1);
}
write (fd, buf, strlen (buf) + 1);
close (fd);
sleep (10);
}
closelog();
exit (0);
}
    
```

读者可以尝试用普通用户的身份执行此程序, 由于这里的 open() 函数必须具有 root 权限, 因此, syslog 就会将错误信息写入系统日志文件 (如 "/var/log/messages") 中, 信息如下。

Jan 30 18:20:08 localhost daemon_syslog[612]: open

3.3 实验内容编写多进程程序

1. 实验目的

通过编写多进程程序, 使读者熟练掌握 fork()、exec()、wait() 和 waitpid() 等函数的使用, 进一步理解在 Linux 中多进程编程的步骤。

2. 实验内容

该实验有 3 个进程, 其中一个为父进程, 其余两个是该父进程创建的子进程, 其中一个子进程运行 "ls -l" 指令, 另一个子进程在暂停 5s 之后异常退出, 父进程先用阻塞方式等待第一个子进程的结束, 然后用非阻塞方式等待另一个子进程的退出, 待收集到第二个子进程结束的信息, 父进程就返回。

3. 实验步骤

(1) 画出该实验流程图。

该实验流程图如图 3.9 所示。

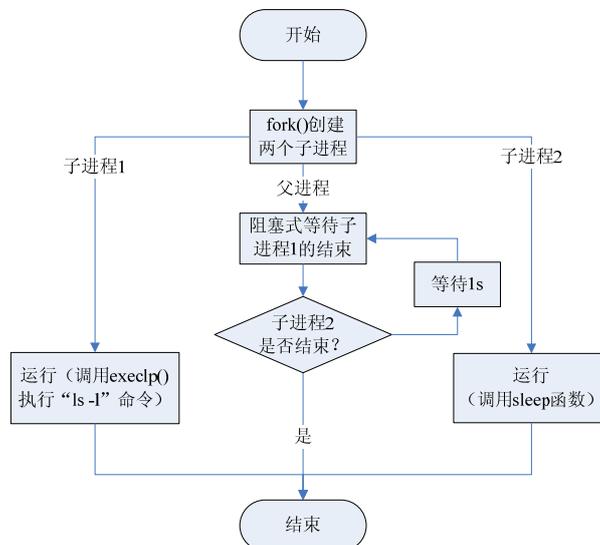


图 3.9 实验流程图

(2) 实验源代码。

先看一下下面的代码，这个程序能得到我们所希望的结果吗？它的运行会产生几个进程？请读者回忆一下 fork() 调用的具体过程。

```
/* multi_proc_wrong.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main (void)
{
    pid_t child1, child2, child;
    /*创建两个子进程*/
    child1 = fork();
    child2 = fork();
    /*子进程 1 的出错处理*/
    if (child1 == -1)
    {
        printf ("Child1 fork error\n");
        exit (1);
    }
    else if (child1 == 0) /*在子进程 1 中调用 execlp() 函数*/
    {
        printf ("In child1: execute 'ls -l'\n");
        if (execlp ("ls", "ls", "-l", NULL) < 0)
        {
            printf ("Child1 execlp error\n");
        }
    }

    if (child2 == -1) /*子进程 2 的出错处理*/
    {
        printf ("Child2 fork error\n");
        exit (1);
    }
    else if ( child2 == 0 ) /*在子进程 2 中使其暂停 5s*/
    {
        printf ("In child2: sleep for 5 seconds and then exit\n");
        sleep (5);
        exit (0);
    }
    else /*在父进程中等待两个子进程的退出*/
    {
        printf ("In father process:\n");
        child = waitpid (child1, NULL, 0); /* 阻塞式等待 */
    }
}
```

```

        if (child == child1)
        {
            printf ("Get child1 exit code\n");
        }
        else
        {
            printf ("Error occured!\n");
        }

        do
        {
            child = waitpid (child2, NULL, WNOHANG) ;/* 非阻塞式等待 */
            if (child == 0)
            {
                printf ("The child2 process has not exited!\n");
                sleep (1);
            }
        } while (child == 0);

        if (child == child2)
        {
            printf ("Get child2 exit code\n");
        }
        else
        {
            printf ("Error occured!\n");
        }
    }
    exit (0);
}
    
```

编译和运行以上代码，并观察其运行结果。它的结果是我们所希望得到的吗？

看完前面的代码之后，再观察下面的代码，它们之间有什么区别，会解决哪些问题？

```

/* multi_proc.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main (void)
{
    pid_t child1, child2, child;

    /*创建两个子进程*/
    child1 = fork();
    /*子进程 1 的出错处理*/
    
```

```

if (child1 == -1)
{
    printf ("Child1 fork error\n");
    exit (1);
}
else if (child1 == 0) /*在子进程 1 中调用 execlp()函数*/
{
    printf ("In child1: execute 'ls -l'\n");
    if (execlp ("ls", "ls", "-l", NULL) < 0)
    {
        printf ("Child1 execlp error\n");
    }
}
else /*在父进程中再创建进程 2，然后等待两个子进程的退出*/
{
    child2 = fork();
    if (child2 == -1) /*子进程 2 的出错处理*/
    {
        printf ("Child2 fork error\n");
        exit (1);
    }
    else if (child2 == 0) /*在子进程 2 中使其暂停 5s*/
    {
        printf ("In child2: sleep for 5 seconds and then exit\n");
        sleep (5);
        exit (0);
    }

    printf ("In father process:\n");

    ..... (以下部分跟前面程序的父进程执行部分相同)

}
exit (0);
}
    
```

(3) 编译程序并运行。

4. 实验结果

运行的结果如下（具体内容与各自的系统有关）。

```

$ ./multi_proc

In child1: execute 'ls -l'          /* 子进程 1 的显示，以下是“ls -l”的运行结果 */

total 28
-rwxr-xr-x 1 david root  232 2008-07-18 04:18 Makefile
-rwxr-xr-x 1 david root 8768 2008-07-20 19:51 multi_proc
-rw-r--r-- 1 david root 1479 2008-07-20 19:51 multi_proc.c
-rw-r--r-- 1 david root 3428 2008-07-20 19:51 multi_proc.o
    
```

```
-rw-r--r-- 1 david root 1463 2008-07-20 18:55 multi_proc_wrong.c

In child2: sleep for 5 seconds and then exit /* 子进程 2 的显示 */

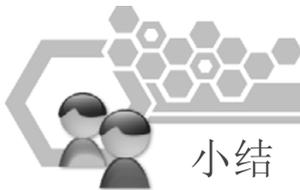
In father process:    27                                /* 以下是父进程显示 */

Get child1 exit code                                /* 表示子进程 1 结束 (阻塞等待) */

The child2 process has not exited!                 /* 等待子进程 2 结束 (非阻塞等待) */

The child2 process has not exited!

Get child2 exit code                                /* 表示子进程 2 结束*/
```



小结

Linux 是一种支持多任务的操作系统。Linux 支持多进程、多线程等多任务处理和任务之间的多种通信机制。

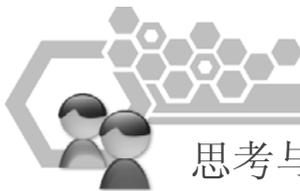
本章主要介绍任务、进程、线程的基本概念和特性以及它们之间的关系。这些概念也是嵌入式 Linux 应用编程的最基本的内容，因此，读者一定要牢牢掌握。

接下来，本章具体介绍了进程的生命周期、进程的内存结构等内容。

本章的编程部分讲解多进程编程，包括创建进程、exec 函数族、等待/退出进程等多进程编程的基本内容，并且举实例加以区别。exec 函数族较为庞大，希望读者能够仔细比较它们之间的区别，认真体会并理解。

最后本章讲解了 Linux 守护进程的编写，包括守护进程的概念、编写守护进程的步骤以及守护进程的出错处理。由于守护进程非常特殊，因此，在编写时有不少细节需要特别注意。守护进程的编写实际上涉及进程控制编程的很多部分，需要加以综合应用。

本章的实验安排了多进程编程，希望读者能够认真完成。



思考与练习

1. 什么叫多任务系统？任务、进程、线程分别是什么，它们之间有何区别？
2. 讲述 Linux 下进程状态如何切换。

联系方式

集团官网: www.hqyj.com

嵌入式学院: www.embedu.org

移动互联网学院: www.3g-edu.org

企业学院: www.farsight.com.cn

物联网学院: www.topsight.cn

研发中心: dev.hqyj.com

集团总部地址: 北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

全国免费咨询电话: 400-706-1880

双休日及节假日请致电值班手机: 15010390966

在线咨询: 张老师 QQ (619366077), 王老师 QQ (2814652411), 杨老师 QQ (1462495461)

企业培训洽谈专线: 010-82600901

院校合作洽谈专线: 010-82600350, 在线咨询: QQ (248856300)

华清远见