嵌入式与移动开发系列

NITE 国家信息技术紧缺人才培养工程
National Information Technology Education Project
国家信息技术紧缺人才培养工程系列丛书

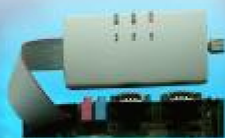
众多专家、厂商联合推荐 • 业界权威培训机构的经验总结

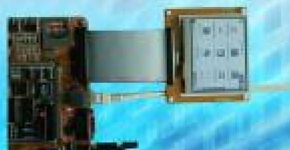
嵌入式Linux应用程序开发标准教程（第2版）

华清远见嵌入式培训中心 编著

提供36小时嵌入式专家讲座视频和教学课件

Embeded Linux Application Development





光盘内容
本书源代码
本书配套PPT
嵌入式专家讲座视频

第 7 章　进程控制开发

本章目标

文件是 **Linux** 中最常见、最基础的操作对象，而进程则是系统资源的单位，本章主要讲解进程控制开发部分，通过本章的学习，读者将会掌握以下内容。

- 掌握进程相关的基本概念
- 掌握 Linux 下的进程结构
- 掌握 Linux 下进程创建及进程管理
- 掌握 Linux 下进程创建相关的系统调用
- 掌握守护进程的概念
- 掌握守护进程的启动方法
- 掌握守护进程的输出及建立方法
- 学会编写多进程程序
- 学会编写守护进程

华清远见

7.1 Linux 进程概述

7.1.1 进程的基本概念

1．进程的定义

进程的概念首先是在 20 世纪 60 年代初期由 MIT 的 Multics 系统和 IBM 的 TSS/360 系统引入的。在 40 多年的发展中，人们对进程有过各种各样的定义。现列举较为著名的几种。

（1）进程是一个独立的可调度的活动（E. Cohen，D. Jofferson）。

（2）进程是一个抽象实体，当它执行某个任务时，要分配和释放各种资源（P. Denning）。

（3）进程是可以并行执行的计算单位。（S. E. Madnick，J. T. Donovan）。

以上进程的概念都不相同，但其本质是一样的。它指出了进程是一个程序的一次执行的过程，同时也是资源分配的最小单元。它和程序是有本质区别的，程序是静态的，它是一些保存在磁盘上的指令的有序集合，没有任何执行的概念；而进程是一个动态的概念，它是程序执行的过程，包括了动态创建、调度和消亡的整个过程。它是程序执行和资源管理的最小单位。因此，对系统而言，当用户在系统中键入命令执行一个程序的时候，它将启动一个进程。

2．进程控制块

进程是 Linux 系统的基本调度和管理资源的单位，那么从系统的角度看如何描述并表示它的变化呢？在这里，是通过进程控制块来描述的。进程控制块包含了进程的描述信息、控制信息以及资源信息，它是进程的一个静态描述。在 Linux 中，进程控制块中的每一项都是一个 task_struct 结构，它是在 include/linux/sched.h 中定义的。

3．进程的标识

在 Linux 中最主要的进程标识有进程号（PID，Process Idenity Number）和它的父进程号（PPID，parent process ID）。其中 PID 惟一地标识一个进程。PID 和 PPID 都是非零的正整数。

在 Linux 中获得当前进程的 PID 和 PPID 的系统调用函数为 getpid()和 getppid()，通常程序获得当前进程的 PID 和 PPID 之后，可以将其写入日志文件以做备份。getpid()和 getppid()系统调用过程如下所示：

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

int main()
{
```

华清远见

```
        /*获得当前进程的进程 ID 和其父进程 ID*/
        printf("The PID of this process is %d\n", getpid());
           printf("The PPID of this process is %d\n", getppid());
    }
```

使用 arm-linux-gcc 进行交叉编译，再将其下载到目标板上运行该程序，可以得到如下结果，该值在不同的系统上会有所不同：

```
$ ./pid
The PID of this process is 78
THe PPID of this process is 36
```

另外，进程标识还有用户和用户组标识、进程时间、资源利用情况等，这里就不做一一介绍，感兴趣的读者可以参见 W.Richard Stevens 编著的《Advanced Programming in the UNIX Environmen》。

4．进程运行的状态

进程是程序的执行过程，根据它的生命周期可以划分成 3 种状态。

- 执行态：该进程正在运行，即进程正在占用 CPU。
- 就绪态：进程已经具备执行的一切条件，正在等待分配 CPU 的处理时间片。
- 等待态：进程不能使用 CPU，若等待事件发生（等待的资源分配到）则可将其唤醒。

它们之间转换的关系如图 7.1 所示。

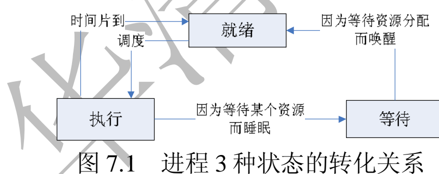


图 7.1　进程 3 种状态的转化关系

华清远见

7.1.2　Linux 下的进程结构

Linux 系统是一个多进程的系统，它的进程之间具有并行性、互不干扰等特点。也就是说，每个进程都是一个独立的运行单位，拥有各自的权利和责任。其中，各个进程都运行在独立的虚拟地址空间，因此，即使一个进程发生异常，它也不会影响到系统中的其他进程。

Linux 中的进程包含 3 个段，分别为“数据段”、“代码段”和“堆栈段”。

- “数据段”存放的是全局变量、常数以及动态数据分配的数据空间，根据存放的数据，数据段又可以分成普通数据段（包括可读可写/只读数据段，存放静态初始化的全局变量或常量）、BSS 数据段（存放未初始化的全局变量）以及堆（存放动态分配的数据）。
- “代码段”存放的是程序代码的数据。
- “堆栈段”存放的是子程序的返回地址、子程序的参数以及程序的局部变量等。如图 7.2 所示。

7.1.3　Linux 下进程的模式和类型

在 Linux 系统中，进程的执行模式划分为用户模式和内核模式。如果当前运行的是用户程序、应用程序或者内核之外的系统程序，那么对应进程就在用户模式下运行；如果在用户程序执行过程中出现系统调用或者发生中断事件，那么就要运行操作系统（即核心）程序，进程模式就变成内核模式。在内核模式下运行的进程可以执行机器的特权指令，而且此时该进程的运行不受用户的干扰，即使是 root 用户也不能干扰内核模式下进程的运行。

用户进程既可以在用户模式下运行，也可以在内核模式下运行，如图 7.3 所示。

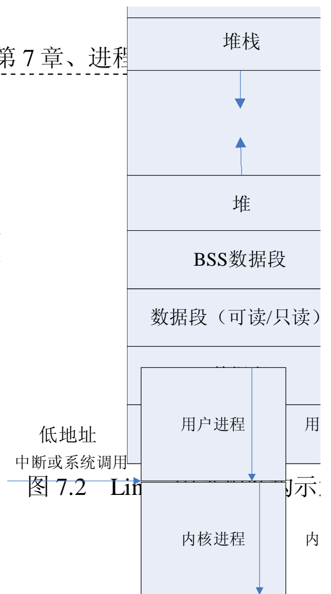


图 7.2　Lin……示

图 7.3 用户进程的两种运行模

7.1.4　Linux 下的进程管理

Linux 下的进程管理包括启动进程和调度进程，下面就分别对这两方面进行简要讲解。

1．启动进程

Linux 下启动一个进程有两种主要途径：手工启动和调度启动。手工启动是由用户输入命令直接启动进程，而调度启动是指系统根据用户的设置自行启动进程。

（1）手工启动。

手工启动进程又可分为前台启动和后台启动。

- 前台启动是手工启动一个进程的最常用方式。一般地，当用户键入一个命令如“ls -l”时，就已经启动了一个进程，并且是一个前台的进程。
- 后台启动往往是在该进程非常耗时，且用户也不急着需要结果的时候启动的。比如用户要启动一个需要长时间运行的格式化文本文件的进程。为了不使整个 shell 在格式化过程中都处于“瘫痪”状态，从后台启动这个进程是明智的选择。

（2）调度启动。

有时，系统需要进行一些比较费时而且占用资源的维护工作，并且这些工作适合在深夜无人值守的时候进行，这时用户就可以事先进行调度安排，指定任务运行的时间或者场合，到时候系统就会自动完成这一切工作。

使用调度启动进程有几个常用的命令，如 at 命令在指定时刻执行相关进程，cron 命令可以自动周期性地执行相关进程，在需要使用时读者可以查看相关帮助手册。

2．调度进程

调度进程包括对进程的中断操作、改变优先级、查看进程状态等，在 Linux 下可以使用相关的系统命令实现其操作，在表 7.1 中列出了 Linux 中常见的调用进程的系统命令，读者在需要的时候可以自行查找其用法。

表 7.1　　Linux 中进程调度常见命令

选　　项	参 数 含 义

华清远见

ps	查看系统中的进程
top	动态显示系统中的进程
nice	按用户指定的优先级运行
renice	改变正在运行进程的优先级
kill	向进程发送信号（包括后台进程）
crontab	用于安装、删除或者列出用于驱动 cron 后台进程的任务。
bg	将挂起的进程放到后台执行

7.2　Linux 进程控制编程

1．fork()

在 Linux 中创建一个新进程的惟一方法是使用 fork()函数。fork()函数是 Linux 中一个非常重要的函数，和读者以往遇到的函数有一些区别，因为它看起来执行一次却返回两个值。难道一个函数真的能返回两个值吗？希望读者能认真地学习这一部分的内容。

（1）fork()函数说明。

fork()函数用于从已存在的进程中创建一个新进程。新进程称为子进程，而原进程称为父进程。使用 fork()函数得到的子进程是父进程的一个复制品，它从父进程处继承了整个进程的地址空间，包括进程上下文、代码段、进程堆栈、内存信息、打开的文件描述符、信号控制设定、进程优先级、进程组号、当前工作目录、根目录、资源限制和控制终端等，而子进程所独有的只有它的进程号、资源使用和计时器等。

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，这两个进程不可能做不同的事。

实际上是在父进程中执行 fork()函数时，父进程会复制出一个子进程，而且父子进程的代码从 fork()函数的返回开始分别在两个地址空间中同时运行。从而两个进程分别获得其所属 fork()的返回值，其中在父进程中的返回值是子进程的进程号，而在子进程中返回 0。因此，可以通过返回值来判定该进程是父进程还是子进程。

同时可以看出，使用 fork()函数的代价是很大的，它复制了父进程中的代码段、数据段和堆栈段里的大部分内容，使得 fork()函数的系统开销比较大，而且执行速度也不是很快。

（2）fork()函数语法。

表 7.2 列出了 fork()函数的语法要点。

表 7.2　　fork()函数语法要点

所需头文件	#include <sys/types.h> // 提供类型 pid_t 的定义 #include <unistd.h>

华清远见

函数原型	pid_t fork(void)
函数返回值	0：子进程
	子进程 ID（大于 0 的整数）：父进程
	–1：出错

（3）fork()函数使用实例。

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

int main(void)
{
    pid_t result;

    /*调用 fork()函数*/
    result = fork();

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

将可执行程序下载到目标板上，运行结果如下所示：

```
$ arm-linux-gcc fork.c -o fork （或者修改 Makefile）
$ ./fork
The returned value is 76      /* 在父进程中打印的信息 */
In father process!!
My PID is 75
The returned value is :0      /* 在子进程中打印的信息 */
In child process!!
My PID is 76
```

从该实例中可以看出，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 PID，而子进程的返回值为 0。

（4）函数使用注意点。

fork()函数使用一次就创建一个进程，所以若把 fork()函数放在了 if else 判断语句中则要小心，不能多次使用 fork()函数。

小知识

由于 fork()完整地复制了父进程的整个地址空间，因此执行速度是比较慢的。为了加快 fork()的执行速度，有些 UNIX 系统设计者创建了 vfork()。vfork()也能创建新进程，但它不产生父进程的副本。它是通过允许父子进程可访问相同物理内存从而伪装了对进程地址空间的真实拷贝，当子进程需要改变内存中数据时才复制父进程。这就是著名的“写操作时复制”（copy-on-write）技术。

现在很多嵌入式 Linux 系统的 fork()函数调用都采用 vfork()函数的实现方式，实际上 uClinux 所有的多进程管理都通过 vfork()来实现。

华清远见

2．exec 函数族

（1）exec 函数族说明。

fork()函数是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，但是，这个新创建的进程如何执行呢？这个 exec 函数族就提供了一个在进程中启动另一个程序执行的方法。它可以根据指定的文件名或目录名找到可执行文件，并用它来取代原调用进程的数据段、代码段和堆栈段，在执行完之后，原调用进程的内容除了进程号外，其他全部被新的进程替换了。另外，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。

- 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。
- 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。

下表 7.3 列举了 exec 函数族的 6 个成员函数的语法。

表 7.3　　　　exec 函数族成员函数语法

所需头文件	#include <unistd.h>
函数原型	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[])
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。

- 查找方式。

读者可以注意到，表 7.3 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”所指定的路径进行查找。

- 参数传递方式。

exec 函数族的参数传递有两种方式：一种是逐个列举的方式，而另一种则是将所有参数整体构造指针数组传递。

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，其语法为 char *arg；字母为“v”（vertor）的表示将所有参数整体构造指针数组传递，其语法为*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 表示结束，如果使用逐个列举方式，那么要把它强制转化成一个字符指针，否则 exec 将会把它解释为一个整型参数，如果一个整型数的长度 char *的长度不同，那么 exec 函数就会报错。

- 环境变量。

exec 函数族可以默认系统的环境变量，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 7.4 是对这 4 个函数中函数名和对应语法的小结，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 7.4　　　　exec 函数名对应含义

华清远见

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

（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>

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在 shell 中使用的命令名和选项。并且当使用文件名进行查找时，系统会在默认的环境变量 PATH 中寻找该可执行文件。读者可将编译后的结果下载到目标板上，运行结果如下所示：

```
$ ./execlp
  PID TTY     Uid        Size State Command
    1         root       1832     S    init
    2         root          0     S    [keventd]
    3         root          0     S    [ksoftirqd_CPU0]
    4         root          0     S    [kswapd]
    5         root          0     S    [bdflush]
    6         root          0     S    [kupdated]
    7         root          0     S    [mtdblockd]
```

```
      8         root          0     S    [khubd]
     35         root       2104     S    /bin/bash /usr/etc/rc.local
     36         root       2324     S    /bin/bash
     41         root       1364     S    /sbin/inetd
     53         root      14260     S    /Qtopia/qtopia-free-1.7.0/bin/qpe
-qws
     54         root      11672     S    quicklauncher
     65         root          0     S    [usb-storage-0]
     66         root          0     S    [scsi_eh_0]
     83         root       2020     R    ps -ef
    $ env
    ……
    PATH=/Qtopia/qtopia-free-1.7.0/bin:/usr/bin:/bin:/usr/sbin:/sbin
        ……
```

此程序的运行结果与在 shell 中直接键入命令“ps -ef”是一样的，当然，在不同系统的不同时刻都可能会有不同的结果。

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");
        }
    }
}
```

同样下载到目标板上运行，运行结果同上例。

下面的示例利用函数 execle()，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，如下所示：

```
/* execle.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=david", NULL};

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

下载到目标板后的运行结果如下所示：

```
$./execle
PATH=/tmp
USER=sunq
```

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所示：

```
#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=david", NULL};

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

```
            printf("Execve error\n");
           }
        }
    }
```

下载到目标板后的运行结果如下所示：

```
$ ./execve
PATH=/tmp
USER=david
```

（4）exec 函数族使用注意点。

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，其中最常见的原因有：

- 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。

小知识　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。

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，清除包括 PCB 在内的各种数据结构，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 7.4 所示。

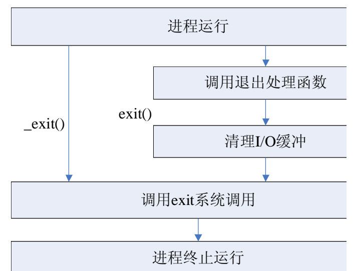


图 7.4　exit 和_exit 函数流程图

从图中可以看出，_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()函数在调用 exit 系统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，就是图中的“清理 I/O 缓冲”一项。

由于在 Linux 的标准函数库中，有一种被称作“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，缓冲区中的数据就会丢失。因此，若想保证数据的完整性，就一定要使用 exit()函数。

（2）exit()和_exit()函数语法。

表 7.5 列出了 exit()和_exit()函数的语法规范。

表 7.5　　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)
	_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()使用实例。

这两个示例比较了 exit()和_exit()两个函数的区别。由于 printf()函数使用的是缓冲 I/O 方式，该函数在遇到“\n”换行符时自动从缓冲区中将记录读出。示例中就是利用这个性质来进行比较的。以下是示例 1 的代码：

```
/* 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，缓冲区中的记录也能正常输出。

以下是示例 2 的代码：

```
/* _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...
$
```

读者从最后的结果中可以看到，调用_exit()函数无法输出缓冲区中的记录。

小知识	在一个进程调用了 exit()之后，该进程并不会立刻完全消失，而是留下一个称为僵尸进程（Zombie）的数据结构。僵尸进程是一种非常特殊的进程，它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他进程收集，除此之外，僵尸进程不再占有任何内存空间。

4．wait()和 waitpid()

（1）wait()和 waitpid()函数说明。

wait()函数是用于使父进程（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接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()就会立即返回。

waitpid()的作用和 wait()一样，但它并不一定要等待第一个终止的子进程，它还有若干选项，如可提供一个非阻塞版本的 wait()功能，也能支持作业控制。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 7.6 列出了 wait()函数的语法规范。

表 7.6　　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	这里的 status 是一个整型指针，是该子进程退出时的状态 • status 若不为空，则通过它可以获得子进程的结束状态 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：已结束运行的子进程的进程号 失败：−1

表 7.7 列出了 waitpid()函数的语法规范。

表 7.7　　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)

华清远见

续表

函数传入值	Pid	pid > 0：只等待进程 ID 等于 pid 的子进程，不管已经有其他子进程 结束退出了，只要指定的子进程还没有结束，waitpid()就会一直等下
		pid = –1：等待任何一个子进程退出，此时和 wait()作用一样
		pid = 0：等待其组 ID 等于调用进程的组 ID 的任一子进程
		pid < –1：等待其组 ID 等于 pid 的绝对值的任一子进程
	status	同 wait()
	options	WNOHANG：若由 pid 指定的子进程不立即可用，则 waitpid()不阻 此时返回值为 0
		WUNTRACED：若实现某支持作业控制，则由 pid 指定的任一子 状态已暂停，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
函数返回值	正常：已经结束运行的子进程的进程号	
	使用选项 WNOHANG 且没有子进程退出：0	
	调用出错：–1	

3）waitpid()使用实例。

由于 wait()函数的使用较为简单，在此仅以 waitpid()为例进行讲解。本例中首先使用 fork()创建一个子进程，然后让其子进程暂停 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，并且父进程每隔一秒循环判断一次。该程序的流程图如图 7.5 所示。

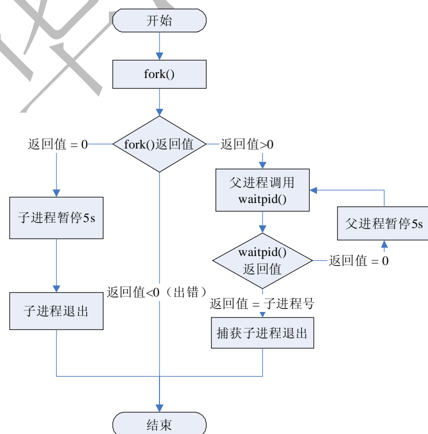


图 7.5　waitpid 示例程序流

华清远见

该程序源代码如下所示：

```
/* waitpid.c */
#include <sys/types.h>
#include <sys/wait.h>
```

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

int main()
{
    pid_t pc, pr;

    pc = fork();
    if (pc < 0)
    {
        printf("Error fork\n");
    }
    else if (pc == 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*/
            pr = waitpid(pc, NULL, WNOHANG);

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

        /*若发现子进程退出，打印出相应情况*/
        if (pr == pc)
        {
           printf("Get child exit code: %d\n",pr);
```

```
            }
            else
            {
                printf("Some error occured.\n");
            }
        }
    }
```

将该程序交叉编译，下载到目标板后的运行结果如下所示：

```
$./waitpid
The child process has not exited
The child process has not exited
The child process has not exited
The child process has not exited
The child process has not exited
Get child exit code: 75
```

可见，该程序在经过 5 次循环之后，捕获到了子进程的退出信号，具体的子进程号在不同的系统上会有所区别。

读者还可以尝试把“pr = **waitpid(pc, NULL, WNOHANG);**”这句改为“pr = waitpid(pc, NULL, 0);”或者“pr = wait(NULL);”，运行的结果为：

```
$./waitpid
Get child exit code: 76
```

可见，在上述两种情况下，父进程在调用 waitpid()或 wait()之后就将自己阻塞，直到有子进程退出为止。

7.3 Linux 守护进程

7.3.1 守护进程概述

守护进程，也就是通常所说的 Daemon 进程，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引导载入时启动，在系统关闭时终止。Linux 有很多系统服务，大多数服务都是通过守护进程实现的，如本书在第二章中讲到的多种系统服务都是守护进程。同时，守护进程还能完成许多系统任务，例如，作业规划进程 crond、打印进程 lqd 等（这里的结尾字母 d 就是 Daemon 的意思）。

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这个终端，这个终端就称为这些进程的控制终端，当控制终端被关闭时，相应的进程都会自动关闭。但是守护进程却能够突破这种限制，它从被执行开始运转，直到整个系统关闭时才会退出。如果想让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。可见，守

护进程是非常重要的。

7.3.2 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。只要将此流程掌握了，就能很方便地编写出用户自己的守护进程。下面就分 4 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，会配合介绍与创建守护进程相关的几个系统函数，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，完成第一步后就会在 shell 终端里造成一种程序已经运行完毕的假象。之后的所有工作都在子进程中完成，而用户在 shell 终端里则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。其关键代码如下所示：

```
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。在这里使用的是系统函数 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。

- 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 来惟一标识。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。且该进程 ID 不会因组长进程 的退出而受到影响。

- 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，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 7.6 所示。

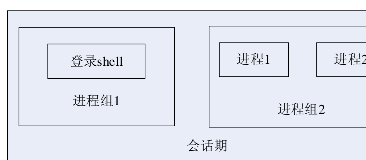


图 7.6 进程组和会话期之间的关系

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。

- 让进程摆脱原会话的控制。
- 让进程摆脱原进程组的控制。
- 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 7.8 列出了 setsid()函数的语法规范。

表 7.8　　　　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．改变当前目录为根目录

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。改变工作目录的常见函数是 chdir()。

4．重设文件权限掩码

文件权限掩码是指屏蔽掉文件权限中的对应位。比如，有一个文件权限掩码是 050，它就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诸多的麻烦。因此，把文件权限掩码设置为 0，可以大大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读或写，但它们一样消耗系统资源，而且可能导致所在的文件系统无法被卸载。

在上面的第二步之后，守护进程已经与所属的控制终端失去了联系。因此从终端

输入的字符不可能达到守护进程，守护进程中用常规方法（如 printf()）输出的字符也不可能在终端上显示出来。所以，文件描述符为 0、1 和 2 的 3 个文件（常说的输入、输出和报错这 3 个文件）已经失去了存在的价值，也应被关闭。通常按如下方式关闭文件描述符：

```
for(i = 0; i < MAXFILE; 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 7.7 所示。

下面是实现守护进程的一个完整实例，该实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 10s 向日志文件/tmp/daemon.log 写入一句话。

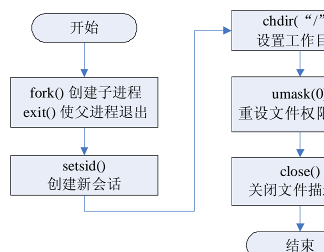


图 7.7 创建守护进程流程图

```
/* 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("/"); /*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("/tmp/daemon.log",
                            O_CREAT|O_WRONLY|O_APPEND, 0600)) < 0)
            {
                printf("Open file error\n");
                exit(1);
            }
            write(fd, buf, strlen(buf) + 1);
            close(fd);
            sleep(10);
        }
        exit(0);
    }
```

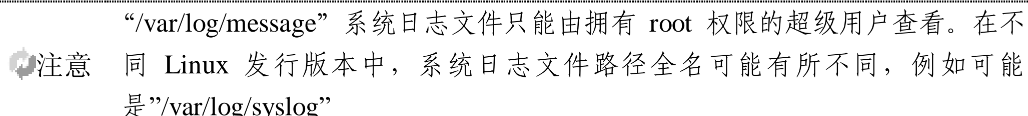
将该程序下载到开发板上，可以看到该程序每隔 10s 就会在对应的文件中输入相关内容。并且使用 ps 可以看到该进程在后台运行。如下所示：

```
$ 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
```

7.3.3 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来通知程序员，

即使使用 gdb 也无法正常调试。那么，守护进程的进程要如何调试呢？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（例如："/var/log/messages"），从而可以直观地看到程序的问题所在。



注意　"/var/log/message" 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"。该文件决定了不同种类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）syslog 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()函数是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()函数是用于关闭系统日志服务的连接。

（2）syslog 相关函数格式。

表 7.9 列出了 openlog()函数的语法规范。

表 7.9　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_PERROR：将消息也同时送到 stderr 上
		LOG_PID：在每条消息中包含进程的 PID
	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 7.10 列出了 syslog()函数的语法规范。

表 7.10　　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 printf 中的格式

表 7.11 列出了 closelog()函数的语法规范。

表 7.11　　closelog 函数语法

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

（3）使用实例。

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，源代码如下所示：

```
/* 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 */
```

华清远见

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

     /* 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到系统日志文件（例如“/var/log/messages”）中，如下所示：

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

7.4　实验内容

7.4.1　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 7.8 所示。

华清远见

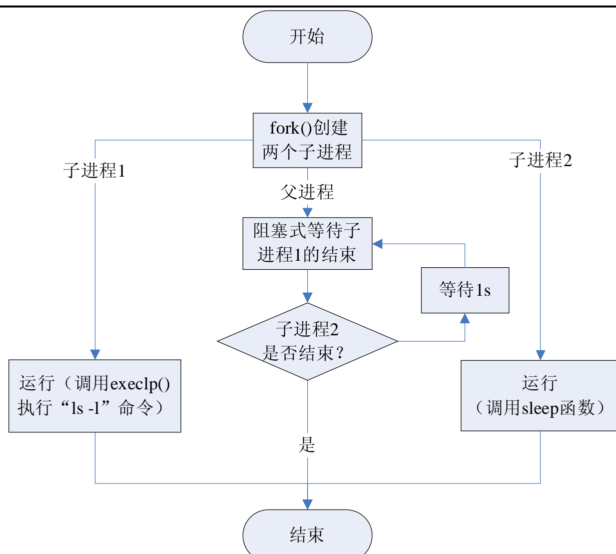


图 7.8　实验 7.4.1 流程图

（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");
        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);
    }
```

（3）首先在宿主机上编译调试该程序：

```
$ gcc multi_proc.c -o multi_proc（或者使用 Makefile）
```

（4）在确保没有编译错误后，使用交叉编译该程序：

```
$ arm-linux-gcc multi_proc.c -o multi_proc （或者使用 Makefile）
```

（5）将生成的可执行程序下载到目标板上运行。

4．实验结果

在目标板上运行的结果如下所示（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:                            /* 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!
The child2 process has not exited!
The child2 process has not exited!
The child2 process has not exited!
```

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

因为几个子进程的执行有竞争关系，因此，结果中的顺序是随机的。读者可以思考，怎样才可以保证子进程的执行顺序呢？

7.4.2 编写守护进程

1．实验目的

通过编写一个完整的守护进程，使读者掌握守护进程编写和调试的方法，并且进一步熟悉如何编写多进程程序。

2．实验内容

在该实验中，读者首先建立起一个守护进程，然后在该守护进程中新建一个子进程，该子进程暂停 10s，然后自动退出，并由守护进程收集子进程退出的消息。在这里，子进程和守护进程的退出消息都在系统日志文件（例如“/var/log/messages”，日志文件的全路径名因版本的不同可能会有所不同）中输出。子进程退出后，守护进程循环暂停，其间隔时间为 10s。

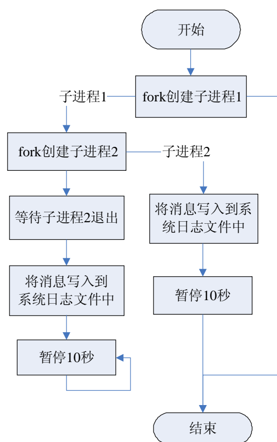


图 7.9　实验 7.4.2 流程图

3．实验步骤

（1）画出该实验流程图。

该程序流程图如图 7.9 所示。

（2）实验源代码。

具体代码设置如下：

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

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

     /*创建子进程 1*/
    child1 = fork();
    if (child1 ==  1)
    {
        perror("child1 fork");
        exit(1);
```

```
}
else if (child1 > 0)
{
    exit(0);         /* 父进程退出*/
}
/*打开日志服务*/
openlog("daemon_proc_info", LOG_PID, LOG_DAEMON);

/*以下几步是编写守护进程的常规步骤*/
setsid();
chdir("/");
umask(0);
for(i = 0; i < getdtablesize(); i++)
{
   close(i);
}

/*创建子进程 2*/
child2 = fork();
if (child2 ==  1)
{
     perror("child2 fork");
     exit(1);
}
else if (child2 == 0)
{ /* 进程 child2 */
  /*在日志中写入字符串*/
    syslog(LOG_INFO, " child2 will sleep for 10s ");
    sleep(10);
    syslog(LOG_INFO, " child2 is going to exit! ");
    exit(0);
}
else
{ /* 进程 child1*/
    waitpid(child2, NULL, 0);
    syslog(LOG_INFO, " child1 noticed that child2 has exited ");
    /*关闭日志服务*/
    closelog();
    while(1)
    {
```

```
                sleep(10);
            }
        }
    }
```

（3）由于有些嵌入式开发板没有 syslog 服务，读者可以在宿主机上编译运行。

```
    $ gcc daemon_proc.c -o daemon_proc （或者使用 Makefile）
```

（4）运行该程序。
（5）等待 10s 后，以 root 身份查看系统日志文件（例如“/var/log/messages”）。
（6）使用 ps –ef | grep daemon_proc 查看该守护进程是否在运行。

4．实验结果

（1）在系统日志文件中有类似如下的信息显示：

```
    Jul 20 21:15:08 localhost daemon_proc_info[4940]: child2 will sleep for 10s
    Jul 20 21:15:18 localhost daemon_proc_info[4940]: child2 is going to exit!
    Jul 20 21:15:18 localhost daemon_proc_info[4939]: child1 noticed that child2 has exited
```

读者可以从时间戳里清楚地看到 child2 确实暂停了 10s。
（2）使用命令 ps –ef | grep daemon_proc 可看到如下结果：

```
    david    4939    1  0 21:15 ?       00:00:00 ./daemon_proc
```

可见，daemon_proc 确实一直在运行。

7.5　本章小结

本章主要介绍进程的控制开发，首先给出了进程的基本概念，Linux 下进程的基本结构、模式与类型以及 Linux 进程管理。进程是 Linux 中程序运行和资源管理的最小单位，对进程的处理也是嵌入式 Linux 应用编程的基础，因此，读者一定要牢牢掌握。

接下来，本章具体讲解了进程控制编程，主要讲解了 fork()函数和 exec 函数族，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，在编写时有不少的细节需要特别注意。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和编写完整的守护进程两个部分。这两个实验都是较为综合的，希望读者能够认真完成。

7.6 思考与练习

查阅资料，明确 Linux 中进程处理和嵌入式 Linux 中的进程处理有什么区别？

推荐课程： **嵌入式学院-嵌入式 Linux 长期就业班**

- **招生简章：** http://www.embedu.org/courses/index.htm
- **课程内容：** http://www.embedu.org/courses/course1.htm
- **项目实战：** http://www.embedu.org/courses/project.htm
- **出版教材：** http://www.embedu.org/courses/course3.htm
- **实验设备：** http://www.embedu.org/courses/course5.htm

推荐课程： **华清远见-嵌入式 Linux 短期高端培训班**

- **嵌入式 Linux 应用开发班：**
  http://www.farsight.com.cn/courses/TS-LinuxApp.htm
- **嵌入式 Linux 系统开发班：**
  http://www.farsight.com.cn/courses/TS-LinuxEMB.htm
- **嵌入式 Linux 驱动开发班：**
  http://www.farsight.com.cn/courses/TS-LinuxDriver.htm

华清远见