

第三讲 linux 应用程序设计

3.0 linux 应用程序设计概述

3.0.1 linux 平台下的应用程序设计

linux 下的应用程序主要有两种特殊的文件来代表：可执行文件和脚本程序。

- I 可执行文件是能够被计算机直接执行的程序，相当于 win 平台下 exe 文件。
- I 脚本程序则是一组指令，这些指令将由另外一个程序（一般来说是 shell 解释器）来执行，相当于 dos 下的 bat 文件。

Linux 不要求可执行文件或者脚本程序具备某种特定的文件名或者某种特定类的扩展名。某个文件是否可以执行将由文件的系统属性来决定（chmod 命令）。

从使用者的角度，可以把两者进行交换，而用户不会发现由什么不同。

在文本模式下，你在登陆系统后，和你打交道的是一个 shell 命令解释器程序，一般来说是 sh（其他的诸如 csh, bash, ksh 等等）。由它来调用执行其他的程序。它的工作原理和 dos 下的 command.com 是一样的。它在一组给定的子目录集合里面，按照你给出的文件名查找到与之同名的那个文件，并把它当作你打算执行的程序。

将被搜索的那些子目录都被保存到一个名字为 PATH 的 shell 变量里面，和 dos 下的情况差不多。PATH 是由系统管理员预先配置好了的，通常包括 /bin, /usr/bin, /usr/local/bin, 你可以使用 echo \$PATH 来显示。如果你是使用的 root 登陆系统，则还会多包括 /sbin, /usr/sbin 目录。

如果你打算学习如何开发一个 shell 命令解释器，当然可以阅读 bash 的源代码，或者参考李善平的边学边干—linux 内核源代码分析一书的 2.2：开发一个简单的 shell 程序。

3.0.2 linux 平台下的 c 开发环境

- I 可以使用 linux 操作系统+vi/vim 编辑器或者 emacs 编辑器+gcc 编译和链接器
- I 可以使用 windows 操作系统+cgywin 编译环境
- I 还可以使用 linux 操作系统+kdeveloper 等 IDE 环境

3.0.3 本讲要求和目的

- I 要求掌握 vi 编辑器的常用命令，具体参考 vim 用户手册中文版.pdf 文档
- I 要求掌握 gcc 的常用命令和工作过程
- I 要求掌握简单的 makefile 的写法
- I 要求掌握的 gdb 程序基本调试方法
- I 要求掌握简单的 shell 程序设计方法
- I 要求掌握基本的进程、信号和文件 API 函数的使用方法、结合简单的实例
- I 了解其他 linux 应用程序设计的知识范围

3.1 第一个程序的编写

为了快速获得对 linux 平台下应用程序设计的体验，我们首先编写经典的 hello world 程序。

下面的代码来自 c programming language 第一章

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
}
```

3.1.1 利用 vi 进行编辑

首先，进入 linux 的控制台界面。

然后，在某个目录建立一个空的 c 程序文件，比如利用 `touch /home/zhaohui/linuxApp/hello.c` 命令

然后 `cd` 到当前目录，调用 `vi` 编辑器对这个空文件进行编辑，输入这个程序的源代码，（利用 `vi hello.c` 命令，进入 `vi` 编辑器，然后进入 `vi` 的编辑状态，然后输入程序，然后进入 `vi` 的命令状态，进行源程序的保存和 `vi` 的退出，回到 `shell` 提示符。）

```
-----hello.c-----
#include <stdio.h>
int main()
{
    printf("Hello, Linux programming
           world!\n");

    return 0;
}
-----
```

3.1.2 gcc 编译和程序运行

然后在当前目录输入编译这个程序的命令，在程序没有错误的情况下，会在当前目录得到一个可执行文件 `hello`。

程序的运行如图所示。

```
$ gcc hello.c -o hello
$ ./hello
Hello, Linux programming world!
```

注意：如果你只是输入 `gcc hello.c` 则生成的可执行文件的名字是 `a.out`

3.1.3 过程总结

这个命令会依次调用 gcc 的预编译器 (cpp)，汇编器 (生成.s 文件)，编译器 (生成.o 文件) 和链接器程序 (ld)。

```
$ gcc -E hello.c -o hello.cpp
$ gcc -x cpp-output -c hello.cpp -o hello.o
$ gcc hello.o -o hello
```

提示：命令 `$ gcc -O2 -S hello.c` 可以得到 .s 汇编文件。如果打算学习某个 c 语言程序所对应的汇编代码。可以参考 csapp 的第三章。诸如变量、数组、指针、控制流、子程序等。

当然也可以使用 `objdump -d hello` 来查看汇编代码。诸如 `objdump` 等这些命令称为 `binutils`，是非常强大的代码分析工具。比如 `addr2line`, `ar`, `as`, `gprof`, `ld`, `nm`, `objcopy`, `objdump`, `ranlib`, `size`, `strings`, `strip` 等。具体可以参考其 `man` 命令手册。

3.2 linux 程序开发预备知识

【参考 Beginning linux programming 2nd 的第一章第 8 节】

【另外参考了 csapp 的第三章和第七章的一部分】

3.2.1 linux 程序开发的头文件

对 c 语言来说，这些头文件一般存在于目录 `/usr/include` 下面

而对于 linux 操作系统的头文件，一般存在于目录 `/usr/include/sys` 和 `/usr/include/linux` 下。

其他程序设计系统也可能有自己单独的目录，比如 `/usr/include/X11` 或者 `/usr/include/g++` 等。

Gcc 的一个可选参数：-I，用于在编译阶段，指定非标准位置的头文件。比如：

`gcc -I /usr/xxx/include test.c` 会在指定的位置查找 `test.c` 文件所 `include` 的一些头文件。

3.2.2 库文件

定义：是一些预先编译好的函数的集合，那些函数都是按照可再使用的原则编写的。它们通常是一组相互关联的用来完成某项常见工作的函数构成（比如 c 库里面的标准输入输出函数、时间函数和数学函数等）。

在链接阶段，会搜索一些默认的目录，比如 /lib 和 /usr/lib 等来查找需要的库文件。

命名：一般是以 lib 打头，然后后面接上表示函数库功能的名字部分。比如 libc、libm 和 libcap 等。

类型：分成静态库和共享库，后缀名分别是 a 和 so（一般在系统中的 /lib 中，两个版本都存在）。而在 win 平台下面，分别是 lib 和 dll。

在编译程序的时候，为了保证链接的正常进行，可以告诉 gcc 库文件的位置（当然，标准的 c 库是不需要指定的，其他的非标准库则要指定）。指定的方法有两个：一个之简写法，一个是完整法。比如：

```
$gcc -o fred fred.c /usr/lib/libm.a
$gcc -o fred fred.c -lm
```

简单法使用的是 gcc 的 -l 选项，用于在链接阶段，知道要链接的处于标准目录下的（动态）库文件的名字（的一部分）

完整法的好处：The -lm (no space between the l and the m) is shorthand (Shorthand is much valued in UNIX circles.) for the library called libm.a in one of the standard library directories (in this case /usr/lib). An additional advantage of the -lm notation is that the compiler will automatically choose the shared library when it exists.

Gcc 的一个可选参数：-L，用于在链接阶段，指定非标准库文件的位置。比如：

```
$gcc -o x11fred -L/usr/openwin/lib
x11fred.c -lX11
```

3.2.3 静态库

3.2.3.1 概述

库文件的含义：就是一组处于可以“拿来就用”的状态下的二进制目标代码。当有程序需要用的函数库中的某个函数的时候，就会通过 `include` 语句引用对此函数做出声明的头文件。编译器和链接程序负责把程序代码和库函数结合在一起成为一个独立的可执行程序。如果使用的不是标准的 `c` 语言运行库而是某个扩展库，则必须指定它的位置和名字（使用 `-l`，`-L` 和 `-I` 参数）。

静态库也叫归档文件，英文 `archive`，后缀名是 `a`，比如 `libc.a` 和 `libX11.a` 等。

自己建立归档文件的方法介绍：编写库函数的时候，尽量把不同类型函数实现编写到不同的源代码文件里面；然后使用 `gcc` 的编译命令对各个文件进行独立编译，从而得到各自的目标文件；然后使用 `ar` 命令把各个目标文件打包在一起。

3.2.3.2 举例：如何建立和使用归档文件

1) 编写一个小函数库，这个函数库里面有两个函数。分别使用两个源代码文件。

2) 各自代码如下：

```
-----fred.c-----  
#include <stdio.h>  
void fred(int arg)  
{  
    printf("fred: you passed %d\n", arg);  
}
```

```
-----bill.c-----  
#include <stdio.h>  
void bill(char *arg)  
{  
    printf("bill: you passed %s\n", arg);  
}
```

3) 进行两个源代码文件的编译

```
$ gcc -c bill.c fred.c
$ ls *.o
bill.o fred.o
```

4) 利用 ar 命令, 把目标代码添加到一个库文件中去。

```
$ ar crv libfoo.a bill.o fred.o
a - bill.o
a - fred.o
```

5) 为了让库的使用者可以重复利用库所提供的函数, 要编写一个头文件, 在其中声明库都是对外提供了哪些函数。如果另外一个程序员打算使用库所提供的服务, 则必须在头文件中 include 这个头文件。

```
-----lib.h-----
/*
This is lib.h. It declares the functions
fred and bill for users
*/

void bill(char *);
void fred(int);
```

6) 编写一个测试程序, 使用库中的一个函数。
源代码如下:

```
-----program.c-----

#include "lib.h"
int main()
{
    bill("Hello World");
    exit(0);
}
```

7) 编译和运行库的测试程序

```
-----第一个方法-----
$ gcc -c program.c
$ gcc -o program program.o bill.o
```

```
$ ./program
bill: you passed Hello World
$
```

-----第二个方法-----

```
$ gcc -o program program.o libfoo.a
$ ./program
bill: you passed Hello World
$
```

-----第三个方法-----

```
$gcc -o program program.o -L. -lfoo
```

说明: The `-L.` option tells the compiler to look in the current directory for libraries. The `-lfoo` option tells the compiler to use a library called `libfoo.a` (or a shared library, `libfoo.so` if one is present).

3.2.3.2 总结归档文件

To see which functions are included in an object file, library or executable program, we can use the `nm` command. 【引入nm命令】

If we take a look at `program` and `lib.a`, we see that the library contains both `fred` and `bill`, but that `program` contains only `bill`. 【可执行文件只是包括了一个】

When the program is created, it only includes functions from the library that it actually needs. Including the header file, which contains declarations for all of the functions in the library, doesn't cause all of the library to be included in the final program. 【归档库的特点: 只是包括需要的, 而不是全部】

3.2.2 共享库

静态库的缺点是：如果我们在同一时间运行多个程序，而它们又都使用着来自同一个函数库里的函数时，内存里面就会有许多份同一个函数的拷贝，在程序文件本身里面也有许多份同样的拷贝。这些会造成大量的硬盘和内存空间的浪费。而使用共享库可以解决上面的两个问题。

存在：共享库的存放位置和静态库是一样的，但是后缀名是 **so**。比如，你可以利用 **ls** 命令显示 **/lib** 目录下的库的名字，可能的一个输出是 **libc.so.6**（它是一个软链接文件），它是 **c** 语言标准库的共享版本，**6** 代表的是主版本号。

特点：如果一个程序使用了共享库，它的链接方式是这样的：程序本身不再包括函数的代码，而只保存共享代码的调用线索（**references**），共享代码是在该程序运行的时候才加入到其中的。当编译好的程序被加载到内存中准备执行的时候，函数的调用线索被解析（**the function references are resolved**），程序向共享库发出调用，共享库只在必要的时候才被加载到内存。

好处：通过这种方法，在内存里，系统就可以只保留一份共享库的拷贝供许多程序使用，在硬盘上也只需要保存一份拷贝就可以了。另外一个好处是共享库的升级不再会影响到依赖于它的那些程序。比如，我们只需要修改从 **/lib/libc.so.6** 到实际库文件的升级版本的符号链接就可以了。

Linux 中的加载器：对于 **linux** 系统而言，负责加载共享库并解析 **app** 程序中的函数调用线索的程序（也就是共享库的动态加载器）是 **ld.so** 或者 **ld-linux.so.2**。查找共享库的其他目录是在 **/etc/ld.so.conf** 文件里面配置的；如果这个文件被修改了（比如，系统里面添加了 **X11** 共享库），可以使用工具程序 **ldd** 来处理（其实就是让 **ld-linux.so.2** 重新读取配置文件的内容到内存中）。

ldd 命令：如果想了解某个程序要求使用的是哪一个共享库，可以使用工具程序 **ldd** 来查看。比如：

```
$ ldd program
libc.so.6 => /lib/libc.so.6 (0x4001a000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2
(0x40000000)
```

从这个举例可以看出来,program 需要两个共享库,同时还列举出了版本号和加载的位置(下一讲会结合一个图来描述)。

3.3 gcc 介绍

【参考 linux programming unleashed】

【参考 gcc 中文手册: 来源《中国 Linux 论坛 man 手册页翻译计划》<http://cmpp.linuxforum.net/>】

3.3.1 gcc 的特点

四个独立的处理过程: 使用 gcc, 程序员能够对编译过程有更多的控制, 编译过程分成四个阶段: 预处理(preprocessing), 编译(compilation), 汇编(assembly)和连接(linking)

在前面的举例后, 我们已经列举了一些简单的命令, 已经体现了独立的特性。

```
$ gcc -E hello.c -o hello.cpp
$ gcc -x cpp-output -c hello.cpp -o hello.o
$ gcc hello.o -o hello
$ gcc -O2 -S hello.c
```

```
$ gcc killerapp.c helper.c -o killerapp
```

gcc 的特性: 1) 如果需要, gcc 能够在生成的二进制执行文件中加入不同数量和种类的调试代码; 2) gcc 可以对代码执行优化(但是在分析汇编代码的时候, 不要使用这个特性) 3) gcc 有 30 多个警告和 3 个警告级别 4) 支持交叉编译, 能够在当前 cpu 平台上面为不同体系结构的硬件平台开发程序 5) gcc 对 c 进行了大量的扩展, 这些扩展中的大部分能够提高程序的执行效率, 或者有助于编译器进行代码优化, 或者使得编程变得更加容易。然而这些是以降低可移植性(编译平台)为代价的。

3.3.2 gcc 对文件扩展名的解释

.c	C 源程序;预处理,编译,汇编
.C	C++源程序;预处理,编译,汇编
.cc	C++源程序;预处理,编译,汇编
.cxx	C++源程序;预处理,编译,汇编
.m	Objective-C 源程序;预处理,编译,汇编
.i	预处理后的 C 文件;编译,汇编
.ii	预处理后的 C++文件;编译,汇编
.s	汇编语言源程序;汇编
.S	汇编语言源程序;预处理,汇编
.h	预处理器文件;通常不出现在命令行上

3.3.3 一些常用的参数和选项

gcc 可以支持的命令行参数很多,这里只是列举最常用的一些:

- 1) **-o**: 指定输出文件为 **file**. 该选项不在乎GCC产生什么输出,无论是可执行文件,目标文件,汇编文件还是预处理后的C代码.
- 2) **-c**: 编译或汇编源文件,但是不作连接.编译器输出对应于源文件的目标文件.
- 3) **-DF00=BAR**: 在命令行上定义宏F00,宏的内容定义为BAR
- 4) **-IDIRNAME**: 把DIRNAME加入到头文件的搜索目录列表中
- 5) **-LDIRNAME**: 把DIRNAME加入到库文件的搜索目标列表中
- 6) **-lF00**: 链接libF00函数库
- 7) **-static**: 执行静态链接

- 8) `-g`: 在可执行文件中包含标准调试信息
- 9) `-ggdb`: 在可执行文件中包括大量只有gdb才可以识别的调试信息
- 10) `-ON`: 执行优化级别: Specify an optimization level N, $0 \leq N \leq 3$.
- 11) `-Wall` Emit all generally useful warnings that gcc can provide.
- 11) `-v` Show the commands used in each step of compilation.

举例: `$gcc myapp.c -L/home/fred/lib`
`-I/home/fred/include -lnew`

`$ gcc cursesapp.c -lcurses -static`

3.3.4 优化选项

01 到 03 共三个级别, 数字越大, 优化程度越高。
具体的含义和内容略。

3.3.5 调试选项

`-g` 和 `-ggdb` 可以加入一些调试信息到可执行文件当中, 以便于对程序进行调试。显然调试信息的加入会使用程序的大型剧增。

3.3.6 GNU 的 c 扩展

gcc 对 c 语言提供了不少的扩展, 这可以提供 c 语言的效率, 但是是以牺牲程序的可移植性为代价的。

在阅读内核源代码的时候, 一个常见的扩展是 `asm` 命令, 它可以在 c 语言中嵌入汇编代码。【具体的讲解可以参考赵炯的书的 5.5.3.1】。另外一个 `attribute` 关键字。

3.4 Makefile 介绍

【参考中文 Makefile 的教程】

【参考 linux programming unleashed 的第四章】

【参考 Beginning linux programming 2nd 的 8.2】

3.4.1 使用 make 的原因

重要性:什么是 makefile? 或许很多 Windows 的程序员都不知道这个东西, 因为那些 Windows 的 IDE 都为你做了这个工作, 但我觉得要作一个好的和 professional 的程序员, makefile 还是要懂。

好处: makefile 带来的好处就是——“自动化编译”, 一旦写好, 只需要一个 make 命令, 整个工程完全自动编译, 极大的提高了软件开发的效率。

定义: make 是一个命令工具, 是一个解释 makefile 中指令的命令工具, 一般来说, 大多数的 IDE 都有这个命令, 比如: Delphi 的 make, Visual C++ 的 nmake, Linux 下 GNU 的 make。可见, makefile 都成为了一种在工程方面的编译方法。

Make 和 makefile 的关系: make 命令执行时, 需要一个 Makefile 文件(好比 makefile 的配置文件), 以告诉 make 命令需要怎么样的去编译和链接程序。

默认的情况下, make 命令会在当前目录下按顺序找寻文件名为 “GNUmakefile”、“makefile”、“Makefile” 的文件, 找到了解释这个文件。在这三个文件名中, 最好使用 “Makefile” 这个文件名, 因为, 这个文件名第一个字符为大写, 这样有一种显目的感觉。最好不要用 “GNUmakefile”, 这个文件是 GNU 的 make 识别的。有另外一些 make 只对全小写的 “makefile” 文件名敏感, 但是基本上来说, 大多数的 make 都支持 “makefile” 和 “Makefile” 这两种默认文件名。【三个可能的 makefile 的名字】

3.4.2 编写 makefile

3.4.2.1 举例

```
-----  
foo.o:  foo.c  defs.h      # foo模块  
      gcc  -c  foo.c  
-----
```

程序源代码编写完成后，输入make就可以完成程序的编译，得到foo.o文件。

3.4.2.2 总结

makefile是一个文本格式的文件，里面包括了一系列的规则，由这些规则来告诉make命令需要编译哪些文件，以及怎样编译这些文件。

```
-----  
target : dependency dependency [...]  
command  
command  
[...]  
-----
```

三个关键字的解析：

target 也就是一个目标文件，可以是 Object File，也可以是执行文件。还可以是一个标签（Label），对于标签也可以称为“伪目标”。

dependency 就是，要生成那个 target 所需要的文件或是目标。即 target 所依赖的东西。

command 也就是 make 需要执行的命令。（任意的 Shell 命令）

这是一个文件的依赖关系，也就是说，target 这一个或多个的目标文件依赖于 dependency 中的文件，其生成规则定义在 command 中。

说白了就是说，dependency 中如果有一个以上的文件比 target 文件要新的话，command 所定义的命令就会被执行。这就是 Makefile 的规则。也就是 Makefile 中最核心的内容。

工作过程：make 会比较 targets 文件和 prerequisites 文件的修改日期，如果 prerequisites 文件的日期要比 targets 文件的日期要新，或者 target 不存在的话，那么，make 就会执行后续定义的命令。

3.4.2.3 一个复杂一些的举例

```
-----  
edit : main.o kbd.o command.o display.o \  
insert.o search.o files.o utils.o  
gcc -o edit main.o kbd.o command.o display.o \  
insert.o search.o files.o utils.o
```

```
main.o : main.c defs.h  
gcc -c main.c
```

```
kbd.o : kbd.c defs.h command.h  
gcc -c kbd.c
```

```
command.o : command.c defs.h command.h  
gcc -c command.c
```

```
display.o : display.c defs.h buffer.h  
gcc -c display.c
```

```
insert.o : insert.c defs.h buffer.h  
gcc -c insert.c
```

```
search.o : search.c defs.h buffer.h  
gcc -c search.c
```

```
files.o : files.c defs.h buffer.h command.h  
gcc -c files.c
```

```
utils.o : utils.c defs.h  
gcc -c utils.c
```

```
clean :  
rm edit main.o kbd.o command.o display.o \  
insert.o search.o files.o utils.o  
-----
```

输入 `make` 会生成 `edit` 程序，而输入 `make clean`，则会删除所有文件，包括 `edit` 和中间的目标代码文件。

这里要说明一点的是，`clean` 不是一个文件，它只不过是一个动作名字，有点像 C 语言中的 `lable` 一样，其冒号后什么也没有，那么，`make` 就不会自动去找文件的依赖性，也就不会自动执行其后所定义的命令。要执行其后的命令，就要在 `make` 命令后明显得指出这个 `lable` 的名字。这样的方法非常有用，我们可以在一个 `makefile` 中定义不用的编译或是和编译无关的命令，比如程序的打包，程序的备份，等等。【`clean` 这个特殊的伪目标】

3.4.3makefile 的编写规则

3.4.3.1 伪目标

伪目标并不对应于实际的文件，比如 `clean`。由于它不需要相关文件，它所指定的命令不自动执行，这是由 `make` 的工作机制决定的。此时，必须输入 `make clean` 才可以来执行这个伪目标。

可以利用伪目标来完成很多有用的工作，比如我们在编译内核的时候，`make dep`，`make install`，`make menuconfig` 等等。

3.4.3.2 变量

我们可以看到`[.o]`文件的字符串被重复了两次，如果我们的工程需要加入一个新的`[.o]`文件，那么我们需要在两个地方加（应该是三个地方，还有一个地方在`clean`中）。当然，我们的`makefile`并不复杂，所以在两个地方加也不累，但如果`makefile`变得复杂，那么我们就有可能会忘掉一个需要加入的地方，而导致编译失败。所以，为了`makefile`的易维护，在`makefile`中我们可以使用变量。`makefile`的变量也就是一个字符串，理解成 C 语言

中的宏可能会更好。【引入需要，用变量来代替经常使用的字符串】

```
objects = main.o kbd.o command.o display.o \
insert.o search.o files.o utils.o

edit : $(objects)
cc -o edit $(objects)
```

格式: VARNAME = some_text [...]

```
另外举例:  TOPDIR = /home/kwall/myproject
            SRCDIR = $(TOPDIR)/src
```

3.4.3.3 环境变量、自动变量和预定义变量

除了用户定义的变量意外,make 也允许使用环境变量、自动变量和预定义变量。

使用环境变量非常简单,在启动时,make 读取已定义的环境变量,并且创建了与之同名同值的变量。

如果 makefile 中有同名的变量,则这个变量将取代与之相应的环境变量。

列举:

TABLE 4.1 AUTOMATIC VARIABLES	
Variable	Description
\$@	The filename of a rule's target
\$<	The name of the first dependency in a rule
\$^	Space-delimited list of all the dependencies in a rule
\$?	Space-delimited list of all the dependencies in a rule that are newer than the target
\$(@D)	The directory part of a target filename, if the target is in a subdirectory
\$(@F)	The filename part of a target filename, if the target is in a subdirectory

TABLE 4.2 CONTINUED	
Variable	Description
CC	Program for compiling C programs; default value = cc
CPP	C Preprocessor program; default value = cpp
RM	Program to remove files; default value = "rm -f"
ARFLAGS	Flags for the archive-maintenance program; default = rv
ASFLAGS	Flags for the assembler program; no default
CFLAGS	Flags for the C compiler; no default
CPPFLAGS	Flags for the C preprocessor; no default
LDFLAGS	Flags for the linker (ld); no default

3.4.3.4 隐式规则

除了在 `makefile` 中定义的显示规则以外，`make` 还有一系列的隐式规则，也可以称为预定义规则。其中的大部分是出于特殊的目的，这里只是列举一些最常用的。

结合下面的一个 `makefile` 的举例：

```
1  OBJS = editor.o screen.o keyboard.o
2  editor : $(OBJS)
3      cc -o editor $(OBJS)
4
5  .PHONY : clean
6
7  clean :
8      rm editor $(OBJS)
```

在这个 `makefile` 提及了三个目标文件：也就是 `editor.o`, `screen.o` 和 `keyboard.o`，但是没有给如何编译这些目标文件的规则，此时，`make` 就使用所谓的隐式规则：对每一个 `XXX.o` 目标文件，找到与之对应的源代码文件 `XXX.c`，然后使用编译命令生成之。

3.4.3.5 模式规则

用户可以自己定义隐式规则，这样 `make` 提供了扩展其本身的隐式规则的方法。

模式规则类似于普通规则，但是它的目标必定含有符号%，这个符号可以与任何非空字符串匹配；为与目标中的%匹配，这个规则的相关文件也必须使用%。

实际上，`make` 就是使用其自己预先定义的模式规则来生成隐式规则的。比如：

```
%.o : %.c
    $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

3.4.3.6 注释

和 `c++` 语言类似，如果要给一行添加注释，则使用 `#` 这个符号。可以提高 `makefile` 的可读性。

3.4.4 其他 `make` 命令参数

具体的可以参考 `man make` 的输出。这里列举几个常用的。

TABLE 4.3 COMMON `make` COMMAND-LINE OPTIONS

Option	Description
<code>-f file</code>	Specify an alternatively-named makefile file.
<code>-n</code>	Print the commands that would be executed, but don't actually execute them.
<code>-Idirname</code>	Specify <code>dirname</code> as a directory in which <code>make</code> should search for included makefiles.
<code>-s</code>	Don't print the commands as they are executed.
<code>-w</code>	If <code>make</code> changes directories while executing, print the current directory names.
<code>-Wfile</code>	Act as if file has been modified; use with <code>-n</code> to see how <code>make</code> would behave if file had been changed.
<code>-r</code>	Disable all of <code>make</code> 's built-in rules.
<code>-d</code>	Print lots of debugging information.
<code>-i</code>	Ignore non-zero error codes returned by commands in a makefile rule. <code>make</code> will continue executing even if a command returns a non-zero exit status.
<code>-k</code>	If one target fails to build, continue to build other targets. Normally, <code>make</code> terminates if a target fails to build successfully.
<code>-jN</code>	Run <code>N</code> commands at once, where <code>N</code> is a non-zero integer.

3.4.4 有用的 `makefile` 伪目标

这里列举几个常用的。如 `install`，`uninstall` 来进行安装。`disk` 可以用来生成准备发布的软件包。`test` 或者 `check` 可以用来进行验证显示适当的诊断信息。

对于内核编译而言，其 `makefile` 比较复杂。可以参考：赵炯的书的 2.10 一节

3.5 进程控制和信号处理

【参考 Beginning linux programming 2nd 第 10 章】

【参考 系统调用跟我学 系列文档】

【参考 linux programming unleashed 的第 11 章】

【参考 csapp 的第 8 章一部分】

重要性: Processes and signals form a fundamental part of the UNIX operating environment. They control almost all activities performed by a UNIX computer system. An understanding of how UNIX manages processes will hold any systems programmer, applications programmer or system administrator in good stead.

3.5.1 什么是进程

官方定义: The Single UNIX Specification, Version 2 (UNIX98) and its predecessor Version 1 (UNIX95), defines a process as “an address space with one or more threads executing within that address space, and the required system resources for those threads.”【一个空间+必要的资源】

我们在学习操作系统的定义: 进程是可并发执行的程序在一个数据集合上运行的过程。

通俗的定义: 硬盘上的一个可执行文件通常被称为程序, 在操作系统中, 一个程序开始执行后, 在开始执行到执行完毕退出的这段时间内, 它在内存中 (也包括被置换出去) 的部分就被称为一个进程

3.5.2 进程的结构

3.5.2.0 举例认识 linux 下的进程

比如: 两个 linux 用户 neil 和 rick, 各自在不同的控制台上输入了 grep 程序, 来在不同的文件里面搜索各自关心的字符串。则两个进程的结构简单的描述图如下:



图中的两个方框代表的是两个进程的虚拟地址空间，具体的格式，我们会在下一讲里面描述。

进程标识符 PID: 每一个进程都会被内核分配一个独一无二的正整数，范围一般是从 2 到 32768。当一个新的进程产生的时候，内核会选择下一个没有被使用的正整数来使用。当已经转过一轮的时候，则会重新从编号 2 开始。1 对应 `init` 进程。

代码共享: 两个进程共享同一个程序 `grep`，也就是说，这段代码只是被从硬盘上加载到内存中一次。代码是以只读的方式加载到内存中的。

C 库函数代码是共享库，也只是加载一次。

进程的私有代码和数据: 不同的进程，因为对于的程序不是一个，因此只读代码有可能是不同的。就算是同一个程序所对应的两个进程，一般来说，数据部分也不是相同的。如图中的 `data` 所示。

另外，每一个进程还有自己的堆栈空间，有自己的环境空间，有自己的程序计数器等寄存器状态信息。

进程支持虚拟存储器，也就是这个内存框图的一部分也许是在二级（硬盘）存储器上面的。

3.5.2.1 进程表

`linux` 的进程表，也就是我们在操作系统中学习的 `pcb` 表，在 `linux` 内核中，它实际上是一个链表，链表的每一个元素是一个执行 `pcb` 的指针。利用这个表，内核可以快速、高效的访问任何一个系统中

的进程。

3.5.2.2 查看进程

在学习 linux 常用命令的时候，我们已经讲解过了 ps 命令。这里举例分析其输出的一些重要信息。

```
$ ps -af
UID          PID  PPID  C  STIME TTY          TIME CMD
root         433    425  0  18:12 tty1        00:00:00 [bash]
rick         445    426  0  18:12 tty2        00:00:00 -bash
rick         456    427  0  18:12 tty3        00:00:00 [bash]
root         467    433  0  18:12 tty1        00:00:00 sh /usr/X11R6/bin/startx
root         474    467  0  18:12 tty1        00:00:00 xinit /etc/X11/xinit/xinitrc --
root         478    474  0  18:12 tty1        00:00:00 /usr/bin/gnome-session
root         487      1  0  18:12 tty1        00:00:00 gnome-smproxy --sm-client-id def
root         493      1  0  18:12 tty1        00:00:01 [enlightenment]
root         506      1  0  18:12 tty1        00:00:03 panel --sm-client-id default8
root         508      1  0  18:12 tty1        00:00:00 xscreensaver --no-splash -timeout
root         510      1  0  18:12 tty1        00:00:01 gmc --sm-client-id default10
root         512      1  0  18:12 tty1        00:00:01 gnome-help-browser --sm-client-i
root         649    445  0  18:24 tty2        00:00:00 su
root         653    649  0  18:24 tty2        00:00:00 bash
neil         655    428  0  18:24 tty4        00:00:00 -bash
root         713      1  2  18:27 tty1        00:00:00 gnome-terminal
root         715    713  0  18:28 tty1        00:00:00 gnome-pty-helper
root         717    716  13  18:28 pts/0        00:00:01 emacs
root         718    653  0  18:28 tty2        00:00:00 ps af
```

PID 和 PPID 都很好理解。

STIME 表示启动的时间

TTY 表示是从哪一个虚拟终端是启动的

CMD 表示本进程运行的程序

3.5.2.3 系统进程

可以利用 ps 命令的 a 选择查看系统进程。也就是由 linux 在启动的时候，由 init 直接创建的一些进程。它们在后台提供一些服务。

```
$ ps -ax
  PID TTY  STAT  TIME COMMAND
    1  ?    S      0:00 init
    7  ?    S      0:00 update (bdf flush)
   40  ?    S      0:01 /usr/sbin/syslogd
   46  ?    S      0:00 /usr/sbin/lpd
   51  ?    S      0:00 sendmail: accepting connections
   88 v02  S      0:00 /sbin/agetty 38400 tty2
  109  ?    R      0:41 X :0
  192 pp0  R      0:00 ps -ax
```

一般来说，系统进程是不使用 TTY 的

init 是所有系统进程和用户进程的祖先。一般的启动顺序是 init—getty—login—shell。

启动一个新的进程，然后等待其完成，是整个系统的基石，后面我们会看到，这个是通过系统调用函数 `fork+exec+wait` 来实现的。

3.5.2.4 进程的调度

上面的举例中，输出的 R 表示这个进程是可以运行的，它也许正在占有 `cpu`，也许是等待分配给 `cpu`。

Linux 内核系统中，有一个进程调度函数，名字是 `scheduler`，它的功能是：在需要进行进程切换的时候（有大概 5 类这样的时刻），由它决定下一个时间片应该分配给哪一个进程。`Scheduler` 的选择依据就是进程的优先级。

在 `linux` 系统中，是从“优先级基数”开始，参考进程的行为来确定其优先级的。具体的实现，参考 `scheduler` 函数。`nice` 命令可以修改进程的优先级基数。

3.5.3linux 的 API 函数

3.5.3.1 系统调用概述

什么是系统调用：Linux 内核中设置了一组用于实现各种系统功能的子程序，称为系统调用。用户可以通过系统调用命令在自己的应用程序中调用它们。从某种角度来看，系统调用和普通的函数调用非常相似。区别仅仅在于，系统调用由操作系统核心提供，运行于核心态；而普通的函数调用由函数库或用户自己提供，运行于用户态。

为什么要用系统调用：实际上，很多已经被我们习以为常的 C 语言标准函数，在 Linux 平台上的实现都是靠系统调用完成的，所以如果想对系统底层的原理作深入的了解，掌握各种系统调用是初步的要求。进一步，若想成为一名 Linux 下编程高手，也就是我们常说的 `Hacker`，其标志之一也是能对各种系统调用有透彻的了解

系统调用是怎么工作的：一般的，进程是不能访问内核的。它不能访问内核所占内存空间也不能调用内核函数。CPU 硬件决定了这些（这就是为什么它被称作"保护模式"）。系统调用是这些规则的一个例外。其原理是进程先用适当的值填充寄存器，然后调用一个特殊的指令，这个指令会跳到一个事先定义的内核中的一个位置（当然，这个位置是用户进程可读但是不可写的）。

在 Intel CPU 中，这个由中断 0x80 实现。硬件知道一旦你跳到这个位置，你就不是在限制模式下运行的用户，而是作为操作系统的内核--所以你就可以为所欲为。进程可以跳转到的内核位置叫做 `sysem_call`。这个过程检查系统调用号，这个号码告诉内核进程请求哪种服务。然后，它查看系统调用表(`sys_call_table`)找到所调用的内核函数入口地址。接着，就调用函数，等返回后，做一些系统检查，最后返回到进程（或到其他进程，如果这个进程时间用尽）。如果你希望读这段代码，它在<内核源码目录>/kernel/entry.S，`Entry(system_call)`的下一行。

如何使用系统调用：一般来说，包括相应头文件，然后直接调用就可以了。但是，真实的过程不是这样的，具体现在略。

errno 是什么：为防止和正常的返回值混淆，系统调用并不直接返回错误码，而是将错误码放入一个名为 `errno` 的全局变量中。

如果一个系统调用失败，你可以读出 `errno` 的值来确定问题所在。`errno` 不同数值所代表的错误消息定义在 `errno.h` 中，你也可以通过命令"`man 3 errno`"来察看它们。

需要注意的是，`errno` 的值只在函数发生错误时设置，如果函数不发生错误，`errno` 的值就无定义，并不会被置为 0。另外，在处理 `errno` 前最好先把它的值存入另一个变量，因为在错误处理过程中，即使像 `printf()` 这样的函数出错时也会改变 `errno` 的值。

该如何学习使用 Linux 系统调用呢：你可以用"man 2 系统调用名称"的命令来查看各条系统调用的介绍，但这首先要求你要有一定的英语基础，其次还得有一定的程序设计和系统编程的功底，man pages 不会涉及太多的应用细节，因为它只是一个手册而非教程。如果 man pages 所提供的东西不能使你感到非常满意，则要参考一些编程的书籍先，比如我们这个课程第一次列举的。还是先框架，然后细节，在细节学习的时候，要不断的举例尝试。

3.5.4 进程控制 API

3.5.4.1 获得进程的 PID

原型：

```
-----  
#include<sys/types.h> /* 提供类型 pid_t 的定义 */  
#include<unistd.h> /* 提供函数的定义 */  
  
pid_t getpid(void);  
-----
```

功能：

getpid 的作用很简单，就是返回当前进程的进程 ID。

举例和运行：

```
-----  
/* getpid_test.c */  
#include<unistd.h>  
  
main()  
{  
    printf("The current process ID is  
%d\n",getpid());  
}  
-----
```

```
$gcc getpid_test.c -o getpid_test
$./getpid_test
The current process ID is 1980
```

在运行一次，输出会不一样的。【原因前面我讲解了的。】

3.5.4.2 fork 系统调用

原型:

```
-----
#include<sys/types.h> /* 提供类型 pid_t 的定义 */
#include<unistd.h> /* 提供函数的定义 */
```

```
pid_t fork(void);
-----
```

功能:

fork 系统调用的作用是复制一个进程。当一个进程调用它，完成后就出现两个几乎一模一样的进程，我们也由此得到了一个新进程。据说 fork 的名字就是来源于这个与叉子的形状颇有几分相似的工作流程。

特点:

调用一次、返回两次；并发执行；相同的但是独立的进程地址空间；共享文件；【这个是来源于 csapp 的资料】【在讲解完举例后，结合自己画两个进程空间图后，再来看这四个特点】

举例和分析:

```
-----
/* fork_test.c */

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

main()
{
    pid_t pid;
```

```

/*此时仅有一个进程*/
pid=fork();

/*此时已经有两个进程在同时运行*/
if(pid<0)
    printf("error in fork!");

else if(pid==0)
    printf("I am the child process, my process
           ID is %d\n",getpid());
else
    printf("I am the parent process, my
           process ID is %d\n",getpid());
}

```

而 Beginning linux programming 2nd 第 10 章里面的写法是：

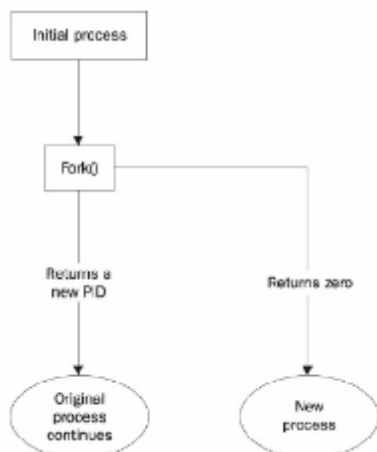
```

pid_t new_pid;

new_pid = fork();

switch(new_pid) {
case -1 :    /* Error */
    break;
case 0 :    /* We are child */
    break;
default :    /* We are parent */
    break;
}

```



【上面的程序的运行结果】

```
$gcc fork_test.c -o fork_test
$./fork_test
I am the parent process, my process ID is 1991
I am the child process, my process ID is 1992
```

【总结】

看这个程序的时候，头脑中必须首先了解一个概念：在语句 `pid=fork()` 之前，只有一个进程在执行这段代码，但在这条语句之后，就变成两个进程在运行了，这两个进程的代码部分完全相同，将要执行的下一条语句都是 `if(pid==0)……`。

两个进程中，原先就存在的那个被称作“父进程”，新出现的那个被称作“子进程”。父子进程的区别除了进程标志符（process ID）不同外，变量 `pid` 的值也不相同，`pid` 存放的是 `fork` 的返回值。`fork` 调用的一个奇妙之处就是它仅仅被调用一次，却能够返回两次，它可能有三种不同的返回值：

- I 在父进程中，`fork` 返回新创建子进程的进程 ID；
- I 在子进程中，`fork` 返回 0；
- I 如果出现错误，`fork` 返回一个负值；

`fork` 出错可能有两种原因：（1）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`。（2）系统内存不足，这时 `errno` 的值被设置为 `ENOMEM`。（关于 `errno` 的意义，请参考系统调用跟我学系列的第一篇文章。）

`fork` 系统调用出错的可能性很小，而且如果出错，一般都为第一种错误。如果出现第二种错误，说明系统已经没有可分配的内存，正处于崩溃的边缘，这种情况对 Linux 来说是很罕见的。

说到这里，聪明的读者可能已经完全看懂剩下的代码了，如果 `pid` 小于 0，说明出现了错误；`pid==0`，就说明 `fork` 返回了 0，也就说明当前进程是子进程，就去执行 `printf("I am the child!")`，否则（`else`），当前进程就是父进程，执行 `printf("I am the parent!")`。

完美主义者会觉得这很冗余，因为两个进程里都各有一条它们永远执行不到的语句。不必过于为此耿耿于怀，毕竟很多年以前，UNIX 的鼻祖们在当时内存小得无法想象的计算机上就是这样写程序的，以我们如今的“海量”内存，完全可以把这几个字节的顾虑抛到九霄云外。

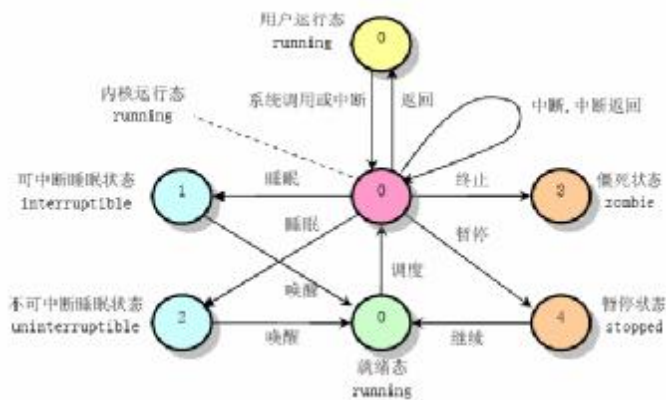
说到这里，可能有些读者还有疑问：如果 `fork` 后子进程和父进程几乎完全一样，而系统中产生新进程唯一的方法就是 `fork`，那岂不是系统中所有的进程都要一模一样吗？那我们要执行新的应用程序时候怎么办呢？【答案在后面的学习中】

3.5.4.3 `exit()` 系统调用

原型：在 2.4.4 版内核中，`exit` 是第 1 号调用，其在 Linux 函数库中的原型是：

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

功能：不像 `fork` 那么难理解，从 `exit` 的名字就能看出，这个系统调用是用来终止一个进程的。无论在程序中的什么位置，只要执行到 `exit` 系统调用，进程就会进入 `TASK_ZOMBIE` 状态（后面的内核分析会具体的进行讲解）。【注意：在系统调用跟我学(3)里面有具体的描述：在一个进程调用了 `exit` 之后，该进程并非马上就消失掉，而是留下一个称为僵尸进程（Zombie）的数据结构。在 Linux 进程的所有状态中，僵尸进程是非常特殊的一种，它已经放弃了几乎所有内存空间，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。僵尸进程却除了留下一些供人凭吊的信息，对系统毫无作用。】



【下面参考的为 csapp 的 8.4.3】

僵死状态 (zombie) 的说明：当一个进程由于某种原因终止时，内核并不是立即把它从系统中清除。取而代之的是，进程被保存在一种终止状态中，直到被它的父进程回收。当父进程回收已经终止的子进程时，内核将子进程的退出状态传递给父进程，然后抛弃已经终止的子进程，从这个时候开始，该子进程就不存在了。因此，一个终止了的但是还没有被回收的子进程就称为了僵死进程。因为在西方的传说中，僵尸是活着的尸体，一个半生半死的实体。因此这是一个类比的称呼。

下面举两个例子来使用 `exit` 和查看僵尸进程

举例 1 及其运行：

```

/* exit_test1.c */
#include<stdlib.h>
main()
{
    printf("this process will exit!\n");
    exit(0);
    printf("never be displayed!\n");
}
  
```

编译和运行这个程序：

```

$gcc exit_test1.c -o exit_test1
$./exit_test1
this process will exit!
  
```

我们可以看到，程序并没有打印后面的 "never be displayed!\n"，因为在此之前，在执行到 `exit(0)` 时，进程就已经终止了。

exit 系统调用带有一个整数类型的参数 **status**，我们可以利用这个参数传递进程结束时的状态，比如说，该进程是正常结束的，还是出现某种意外而结束的。

一般来说，0 表示没有意外的正常结束；其他的数值表示出现了错误，进程非正常结束。我们在实际编程时，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。关于 **wait** 的详细情况，我们将在后面进行介绍。

举例 2 及其运行： 目的：查看僵尸进程

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

main()
{
    pid_t pid;

    pid=fork();

    if(pid<0) /* 如果出错 */
        printf("error occurred!\n");

    else if(pid==0) /* 如果是子进程 */
        exit(0);

    else /* 如果是父进程 */
    {
        sleep(60); /* 休眠 60 秒,这段时间里,
父进程什么也干不了 */
        wait(NULL); /* 收集僵尸进程 */
    }
}
```

编译

```
$ gcc zombie.c -o zombie
```

后台运行

```
$ ./zombie &
```

```
[1] 1577
```

显示当前进程

```
$ ps -ax
```

```
... ..
```

```
1177 pts/0    S      0:00 -bash
1577 pts/0    S      0:00 ./zombie
1578 pts/0    Z      0:00 [zombie <defunct>]
1579 pts/0    R      0:00 ps -ax
```

这个输出的原因：父亲还在睡觉，而儿子已经结束了，因此也就是父亲还没有来得及回收儿子的终止状态。

3.5.4.4 `_exit()` 系统调用

概述：作为系统调用而言，`_exit` 和 `exit` 是一对孪生兄弟，它们究竟相似到什么程度，我们可以从 Linux 的源码中找到答案：

```
/*摘自文件 include/asm-i386/unistd.h 第 334 行
*/
#define __NR__exit __NR_exit
```

“`__NR_`”是在 Linux 的源码中为每个系统调用加上的前缀，请注意第一个 `exit` 前有 2 条下划线，第二个 `exit` 前只有 1 条下划线。

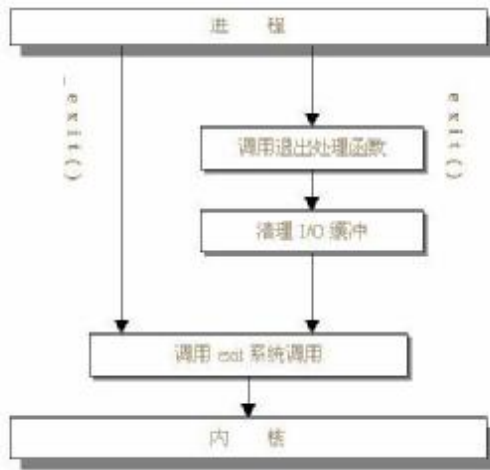
这时随便一个懂得 C 语言并且头脑清醒的人都会说，`_exit` 和 `exit` 没有任何区别，但我们还要讲一下这两者之间的区别，这种区别主要体现在它们在函数库中的定义。

原型：`_exit` 在 Linux 函数库中的原型是：

```
#include<unistd.h>
void _exit(int status);
```

和 `exit` 比较一下，`exit()` 函数定义在 `stdlib.h` 中，而 `_exit()` 定义在 `unistd.h` 中，从名字上看，`stdlib.h` 似乎比 `unistd.h` 高级一点，那么，它们之间到底有什么区别呢？让我们先来看流程图，通过下图，我们会对这两个系统调用的执行过程产生

一个较为直观的认识。



从图中可以看出：**_exit()**函数的作用最为简单：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（不包括 PCB）；**exit()**函数则在这些基础上作了一些包装，在执行退出之前加了若干道工序，也是因为这个原因，有些人认为 **exit** 已经不能算是纯粹的系统调用。

总结：**exit()**函数与**_exit()**函数最大的区别就在于 **exit()**函数在调用 **exit** 系统调用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（这个是最基本的工作）。

知识扩展：在 Linux 的标准函数库中，有一套称作“高级 I/O”的函数，我们熟知的 **printf()**、**fopen()**、**fread()**、**fwrite()**都在此列，它们也被称作“缓冲 I/O (buffered I/O)”，其特征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，会多读出若干条记录，这样下次读文件时就可以直接从内存的缓冲区中读取，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(达到一定数量，或遇到特定字符，如换行符\n 和文件结束符 EOF)，再将缓冲区中的内容一次性写入文件，这样就大大增加了文件读写速度，但也为我们编程带来了一点点麻烦。如果有一些数据，我们认为已经写入了文件，实际上因为没有满足特定的条件，它们还只是保存在缓冲区内，这时我们用**_exit()**函数直接将进程关闭，缓

缓冲区中的数据就会丢失，反之，如果想保证数据的完整性，就一定要使用 `exit()` 函数。

举例之 1：证明 `exit` 可以同步 `buffer` 空间

```
/* exit2.c */
#include<stdlib.h>
main()
{
    printf("output begin\n");
    printf("content in buffer");
    exit(0);
}
```

运行结果：

```
$gcc exit2.c -o exit2
```

```
$/exit2
```

```
output begin
```

```
content in buffer
```

举例之 2：

```
/* _exit1.c */
#include<unistd.h>
main()
{
    printf("output begin\n");
    printf("content in buffer");
    _exit(0);
}
```

运行结果：

```
$gcc _exit1.c -o _exit1
```

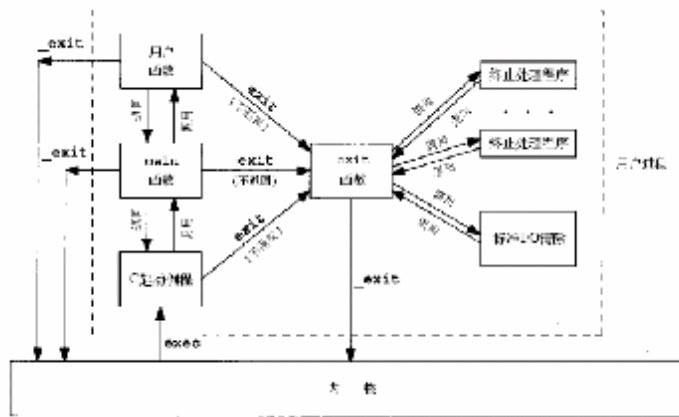
```
$/_exit1
```

```
output begin
```

补充说明：在 `Linux` 中，标准输入和标准输出都是作为文件处理的，虽然是一类特殊的文件，但从程序员的角度来看，它们和硬盘上存储数据的普通文

件并没有任何区别。与所有其他文件一样，它们在打开后也有自己的缓冲区。

【补充说明 1：来自 APU 第七章】



【补充说明 2：来自 CSAPP 的 7.9】

【说明：等到 3.5.4.11 后，再回来讲解这里】

【旁注：我的心得：1】当前的问题后面可能会自动解决；2）学习的圆圈效应 3】利用汇编来理解 c 语言】

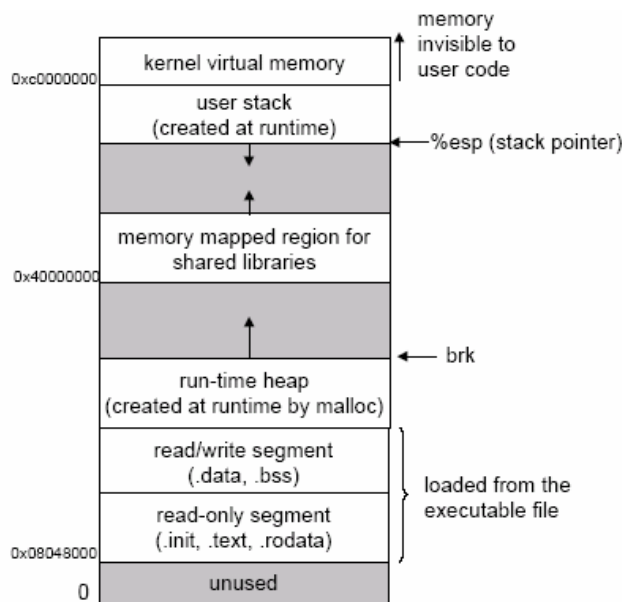
To run an executable object file `p`, we can type its name to the Unix shell's command line:

```
unix> ./p
```

Since `p` does not correspond to a built-in shell command, the shell assumes that `p` is an executable object file, which it runs for us by invoking some memory-resident operating system code known as the loader. Any Unix program can invoke the loader by calling the `execve` function, which we will describe in detail in Section 8.4.6. The loader copies the code and data in the executable object file from disk into memory, and then runs the program by jumping to its first instruction, or entry point. This process of copying the program into memory and then running it is known as *loading*.

Every Unix program has a run-time memory image similar to the one in Figure 7.13. On Linux systems, the code segment always starts at address `0x08048000`. The data segment follows at the next 4-KB aligned address. The run-time *heap* follows on the first 4-KB aligned address past the read/write segment and grows up via calls to the `malloc` library. (We will describe `malloc` and the heap in detail in Section 10.9). The segment starting at address `0x40000000` is reserved for shared libraries. The user stack always starts at address `0xbfffffff` and grows down (towards lower memory addresses). The segment starting above the stack at address `0xc0000000` is reserved for the code and data in the memory-resident part of the operating system known as the *kernel*.

When the loader runs, it creates the memory image shown in Figure 7.13. Guided by the segment header table in the executable, it copies chunks of the executable into the code and data segments. Next, the loader jumps to the program's entry point, which is always the address of the `start` symbol. The *startup code* at the `start` address is defined in the object file `crt1.o` and is the same for all C programs. Figure 7.14 shows the specific sequence of calls in the startup code. After calling initialization routines in from the `.text` and `.init` sections, the startup code calls the `atexit` routine, which appends a list of routines that should be called when the application calls the `exit` function. The `exit` function runs the functions registered by `atexit`, and then returns control to the operating system by calling `_exit`). Next, the startup code calls the application's `main` routine, which begins executing our C code. After the application returns, the startup code calls the `_exit` routine, which returns control to the operating system.



```

1 0x08048300 _start:      /* entry point in .text */
2    call __libc_init_first /* startup code in .text */
3    call __init          /* startup code in .init */
4    call __atexit         /* startup code in .text */
5    call main             /* application main routine */
6    call _exit            /* returns control to OS */
7 /* control never reaches here */

```

[在main中，如果调用了exit，则标号6的语句不会被run]

Figure 7.14: Pseudo-code for the `crt1.o` startup routine in every C program. Note: The code that pushes the arguments for each function is not shown.

【本课件的后面参考了 shown 的图，呵呵。】

3.5.4.5 wait()和waitpid 的引入

引入：我们已经学习了系统调用 `exit`，它的作用是使进程退出，但也仅仅限于将一个正常的进程变成一个僵尸进程，并不能将其完全销毁。僵尸进程虽然对其他进程几乎没有什么影响，不占用 CPU 时间，消耗的内存也几乎可以忽略不计，但有它在那里呆着，还是让人觉得心里很不舒服。而且 Linux 系统中进程数目是有限制的，在一些特殊的情况下，如果存在太多的僵尸进程，也会影响到新进程的产生。那么，我们该如何来消灭这些僵尸进程呢？（这个是 `wait` 函数族的两个功能之一）

僵尸进程的功能复习：再次来复习一下僵尸进程的来由，我们知道，Linux 和 UNIX 总有着剪不断理还乱的亲缘关系，僵尸进程的概念也是从 UNIX 上继承来的，而 UNIX 的先驱们设计这个东西并非是因为闲来无聊想烦烦其他的程序员。僵尸进程中保存着很多对程序员和系统管理员非常重要的信息，**首先**，这个进程是怎么死亡的？是正常退出呢（从这个意义上理解，exit(0) 和 exit(n) 都是正常的退出），还是出现了错误，还是被其它进程强迫退出的？**其次**，这个进程占用的总系统 CPU 时间和总用户 CPU 时间分别是多少？发生页错误的数目和收到信号的数目。这些信息都被存储在僵尸进程中，试想如果没有僵尸进程，进程一退出，所有与之相关的信息都立刻归于无形，而此时程序员或系统管理员需要用到，就只好干瞪眼了。

那么，我们如何收集这些信息，并终结这些僵尸进程呢？就要靠我们下面要讲到的 `waitpid` 调用和 `wait` 调用。这两者的作用都是收集僵尸进程留下的信息，同时使这个进程彻底消失。

3.5.4.6 wait ()系统调用

原型：

```
#include <sys/types.h> /* 提供类型 pid_t 的定义 */
#include <sys/wait.h>
```

```
pid_t wait(int *status)
```

功能：进程一旦调用了 `wait`，就立即阻塞自己（这里讲解一下阻塞函数的含义），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（两个功能）；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：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，只想把这个僵尸进程消灭掉，（事实上绝大多数情况下，我们都会这样想），我们就可以设定这个参数为

NULL，就象下面这样：

```
pid = wait(NULL); /*这样，我们只是使用了一个功能*/
```

返回值：如果成功，`wait` 会返回被收集的子进程的进程 ID，如果调用进程没有子进程，调用就会失败，此时 `wait` 返回-1，同时 `errno` 被置为 `ECHILD`。

举例 1 及其运行：功能：儿子 `over` 了父亲才继续

```
/* wait1.c */
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>

main()
{
    pid_t pc,pr;

    pc=fork();

    if(pc<0)          /* 如果出错 */
        printf("error ocurred!\n");

    else if(pc==0)
    { /* 如果是子进程 */
        printf("This is child process with
pid of %d\n",getpid());
        sleep(10);      /* 睡眠 10 秒钟 */
    }

    else
    { /* 如果是父进程 */
        pr=wait(NULL);  /* 在这里等待 */

        printf("I caught a child process
with pid of %d\n"),pr);
    }

    exit(0); /*父子进程都要运行的 */
}
```

```
}
```

举例 1 的编译和运行:

```
$ gcc wait1.c -o wait1
```

```
$ ./wait1
```

```
This is child process with pid of 1508
```

```
I caught a child process with pid of 1508
```

可以明显注意到，在第 2 行结果打印出来前有 10 秒钟的等待时间，这就是我们设定的让子进程睡眠的时间，只有子进程从睡眠中苏醒过来，它才能正常退出，也就才能被父进程捕捉到。其实这里我们不管设定子进程睡眠的时间有多长，父进程都会一直等待下去，读者如果有兴趣的话，可以试着自己修改一下这个数值，看看会出现怎样的结果。

参数的另外一个使用方式:

如果参数 **status** 的值不是 NULL，**wait** 就会把子进程退出时的状态取出并存入其中，这是一个整数值 (**int**)，指出了子进程是正常退出还是被非正常结束的（一个进程也可以被其他进程用信号结束，我们将在以后介绍），以及正常结束时的返回值，或被哪一个信号结束的等信息。由于这些信息被存放在一个整数的不同二进制位中，所以用常规的方法读取会非常麻烦，人们就设计了一套专门的宏 (**macro**) 来完成这项工作，下面我们来学习一下其中最常用的两个：

1, **WIFEXITED(status)** 这个宏用来指出子进程是否为正常退出的，如果是，它会返回一个非零值。

（请注意，虽然名字一样，这里的参数 **status** 并不同于 **wait** 唯一的参数 -- 指向整数的指针 **status**，而是那个指针所指向的整数，切记不要搞混了。）【见下面的例子】

2, **WEXITSTATUS(status)** 当 **WIFEXITED** 返回非零值时，我们可以用这个宏来提取子进程的返回值，如果子进程调用 **exit(5)** 退出，**WEXITSTATUS(status)** 就会返回 5；如果子进程调用 **exit(7)**，**WEXITSTATUS(status)** 就会返回 7。请注意，如果进程不是正常退出的，也就是说，

WIFEXITED 返回 0，这个值就毫无意义。

举例 2 及其运行：使用两个宏来使用第二个功能

```
/* wait2.c */
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

main()
{
    int status;
    pid_t pc,pr;

    pc=fork();

    if(pc<0) /* 如果出错 */
        printf("error occurred!\n");

    else if(pc==0)
    { /* 子进程 */

        printf("This is child process with pid
of %d.\n",getpid());
        exit(3); /* 子进程返回 3 */
    }

    else
    { /* 父进程 */
        pr=wait(&status);

        if(WIFEXITED(status))
        { /* 如果 WIFEXITED 返回非零值 */
            printf("the child process %d exit
normally.\n",pr);
            printf("the return code is
%d.\n",WEXITSTATUS(status));
        }
        else /* 如果 WIFEXITED 返回零 */
            printf("the child process %d exit
abnormally.\n",pr);
    }
}
```

```
    }  
}
```

举例 2 的编译和运行:

```
$ cc wait2.c -o wait2  
$ ./wait2  
This is child process with pid of 1538.  
the child process 1538 exit normally.  
the return code is 3.
```

3.5.4.7 wait()系统调用的应用: 进程同步

进程同步的需求: 有时候, 父进程要求子进程的运算结果进行下一步的运算, 或者子进程的功能是为父进程提供了下一步执行的先决条件 (如: 子进程建立文件, 而父进程写入数据), 此时父进程就必须在某一个位置停下来, 等待子进程运行结束, 而如果父进程不等待而直接执行下去的话, 可以想见, 会出现极大的混乱。

定义: 这种情况称为进程之间的同步, 更准确地说, 这是进程同步的一种特例。进程同步就是要协调好 2 个以上的进程, 使之以安排好地次序依次执行。对于我们假设的这种情况, 则完全可以用 `wait` 系统调用简单的予以解决。请看下面这个例子:

```
#include <sys/types.h>  
#include <sys/wait.h>  
  
main()  
{  
    pid_t pc, pr;  
    int status;  
  
    pc=fork();  
  
    if(pc<0)  
        printf("Error occured on  
forking.\n");
```

```

else if(pc==0)
{
    /* 子进程的具体工作代码 */
    exit(0);
}
else
{
    /* 父进程的工作 */
    pr=wait(&status);
    /* 利用子进程的结果 */
}
}

```

这段程序只是个例子，不能真正拿来执行，但它却说明了一些问题，首先，当 `fork` 调用成功后，父子进程各做各的事情，但当父进程的工作告一段落，需要用到子进程的结果时，它就停下来调用 `wait`，一直等到子进程运行结束，然后利用子进程的结果继续执行，这样就圆满地解决了我们提出的进程同步问题

3.5.4.8 waitpid() 系统调用

原型：

```

#include <sys/types.h> /* 提供类型 pid_t 的定义 */
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status,
              int options)

```

功能：从本质上讲，系统调用 `waitpid` 和 `wait` 的作用是完全相同的，但 `waitpid` 多出了两个可由用户控制的参数 `pid` 和 `options`，从而为我们编程提供了另一种更灵活的方式。

参数 1: pid

从参数的名字 `pid` 和类型 `pid_t` 中就可以看出，这里需要的是一个进程 ID。但当 `pid` 取不同的值时，在这里有不同的意义。

1 `pid>0` 时，只等待进程 ID 等于 `pid` 的子进程，不管其它已经有多少子进程运行结束退出了，

只要指定的子进程还没有结束，waitpid 就会一直等下去。

- I pid=-1 时，等待任何一个子进程退出，没有任何限制，此时waitpid和wait的作用一模一样。
- I pid=0 时，等待同一个进程组中的任何子进程，如果子进程已经加入了别的进程组，waitpid不会对它做任何理睬。
- I pid<-1 时，等待一个指定进程组中的任何子进程，这个进程组的 ID 等于 pid 的绝对值。

参数 2: options

options 提供了一些额外的选项来控制 waitpid，目前在 Linux 中只支持 WNOHANG 和 WUNTRACED 两个选项，这是两个常数，可以用 "|" 运算符把它们连接起来使用，比如：

```
ret=waitpid(-1,NULL,WNOHANG | WUNTRACED);
```

如果我们不想使用它们，也可以把 options 设为 0，如：ret=waitpid(-1,NULL,0); 【阻塞函数】

如果使用了 WNOHANG 参数调用 waitpid，即使没有子进程退出，它也会立即返回，不会像 wait 那样永远等下去。【这个就是非阻塞函数了】

而 WUNTRACED 参数，由于涉及到一些跟踪调试方面的知识，加之极少用到，这里就不多讲了，有兴趣的同学可以自行查阅相关材料。

联系：看到这里，聪明的读者可能已经看出端倪了--wait 不就是经过包装的 waitpid 吗？没错，察看<内核源码目录>/include/unistd.h 文件 349-352 行就会发现以下程序段：

```
static inline pid_t wait(int * wait_stat)
{
    return waitpid(-1,wait_stat,0);
}
```

返回值：

waitpid 的返回值比 wait 稍微复杂一些，一共有 3 种情况：

- | 当正常返回的时候，waitpid 返回收集到的子进程的进程 ID；
- | 如果设置了选项 WNOHANG，而调用中 waitpid 发现没有已退出的子进程可收集，则返回 0；
- | 如果调用中出错，则返回-1，这时 errno 会被设置成相应的值以指示错误所在； 比如：当 pid 所指示的子进程不存在，或此进程存在，但不是调用进程的子进程，waitpid 就会出错返回，这时 errno 被设置为 ECHILD；

举例的源代码及其运行：【看 html 会更加清楚】

【好比查询式，还可以利用信号实现中断式】

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

main()
{
    pid_t pc, pr;

    pc=fork();

    if(pc<0)          /* 如果 fork 出错 */
        printf("Error occured on forking.\n");

    else if(pc==0)
    { /* 如果是子进程 */
        sleep(10); /* 睡眠 10 秒 */
        exit(0);
    }

    /* 如果是父进程 */
    do
    {

        pr=waitpid(pc, NULL, WNOHANG);
        /* 使用了 WNOHANG 参数，waitpid 不会在这里等待 */

        if(pr==0)
        { /* 如果没有收集到子进程 */
```

```

        printf("No child exited\n");
        sleep(1);
    }

    }while(pr==0); /* 没有收集到子进程，就回去继续尝试 */

    if(pr==pc)
        printf("successfully get child %d\n",
pr);
    else
        printf("some error occurred\n");
}

```

举例的运行:

```

$ cc waitpid.c -o waitpid
$ ./waitpid
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
successfully get child 1526

```

父进程经过 10 次失败的尝试之后，终于收集到了退出的子进程。

因为这只是一个例子程序，不便写得太复杂，所以我们就让父进程和子进程分别睡眠了 10 秒钟和 1 秒钟，代表它们分别作了 10 秒钟和 1 秒钟的工作。父子进程都有工作要做，父进程利用工作的简短间歇察看子进程的是否退出，如退出就收集它。

3.5.4.9 exec() 系统调用

引入: 目前，我们学习到这里，还有一个很大的疑

惑没有解决：既然所有新进程都是由 `fork` 产生的，而且由 `fork` 产生的子进程和父进程几乎完全一样，那岂不是意味着系统中所有的进程都应该一模一样了吗？而且，就我们的常识来说，当我们执行一个程序的时候，新产生的进程的内容应就是程序的内容才对。是我们理解错了吗？显然不是，要解决这些疑惑，就必须提到我们下面要介绍的 `exec` 系统调用。

原型：

说是 `exec` 系统调用，实际上在 Linux 中，并不存在一个 `exec()` 的函数形式，`exec` 指的是一组函数，一共有 6 个，分别是：

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

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

其中只有 `execve` 是真正意义上的系统调用，其它都是在此基础上经过包装的库函数。

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。

返回值：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，颇有些神似"三十六计"中的"金蝉脱壳"。看上去还是旧的躯壳，却已经注入了新的灵魂。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。

应用：现在我们应该明白了，Linux 下是如何执行新程序的，每当有进程认为自己不能为系统和拥护做出任何贡献了，他就可以发挥最后一点余热，调用任何一个 **exec**，让自己以新的面貌重生；或者，更普遍的情况是，如果一个进程想执行另一个程序，它就可以 **fork** 出一个新进程，然后调用任何一个 **exec**，这样看起来就好像通过执行应用程序而产生了一个新进程一样。【后者更加常用】

实现上的效率问题：事实上第二种情况被应用得如此普遍，以至于 Linux 专门为其作了优化，我们已经知道，**fork** 会将调用进程的所有内容原封不动的拷贝到新产生的子进程中去，这些拷贝的动作很消耗时间，而如果 **fork** 完之后我们马上就调用 **exec**，这些辛辛苦苦拷贝来的东西又会被立刻抹掉，这看起来非常不划算，于是人们设计了一种"写时拷贝（copy-on-write）"技术，使得 **fork** 结束后并不立刻复制父进程的内容，而是到了真正实用的时候才复制，这样如果下一条语句是 **exec**，它就不会白白作无用功了，也就提高了效率。

参数：在学习前，复习一些基础知识：

下面这个 **main** 函数的形式可能有些出乎我们的意料：

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

它可能与绝大多数教科书上描述的都不一样，但实际上，这才是 **main** 函数真正完整的形式。

参数 **argc** 指出了运行该程序时命令行参数的个数，数组 **argv** 存放了所有的命令行参数，数组 **envp** 存放了所有的环境变量。

环境变量指的是一组值，从用户登录后就一直存

在，很多应用程序需要依靠它来确定系统的一些细节，我们最常见的环境变量是 PATH，它指出了应到哪里去搜索应用程序，如/bin；HOME 也是比较常见的环境变量，它指出了我们在系统中的个人目录。环境变量一般以字符串"XXX=xxx"的形式存在，XXX 表示变量名，xxx 表示变量的值。

值得一提的是，`argv` 数组和 `envp` 数组存放的都是指向字符串的指针，这两个数组都以一个 `NULL` 元素表示数组的结尾。【下面的两个图来自 `csapp` 的第八章】



Figure 8.17: Organization of an argument list.



Figure 8.18: Organization of an environment variable list.

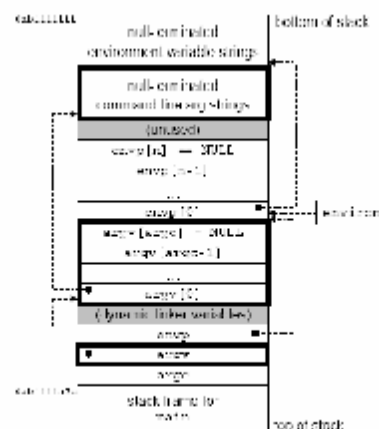


Figure 8.19: Typical organization of the user stack when a new program starts.

可以通过一个举例来输出这两个字符串指针的数组。

```
/* main.c */
int main(int argc, char *argv[], char *envp[])
{
    printf("\n### ARGV ###\n", argc);

    printf("\n### ARGV ###\n");
    while(*argv)
        printf("%s\n", *(argv++));

    printf("\n### ENVP ###\n");
    while(*envp)
        printf("%s\n", *(envp++));

    return 0;
}
```

编译它：

```
$ cc main.c -o main
```

运行它：

```
$ ./main -xx 000
```

```
### ARGV ###
```

```
3
```

```
### ARGV ###
```

```
./main
```

```
-xx
```

```
000
```

```
### ENVP ###
```

```
PWD=/home/lei
```

```
REMOTEHOST=dt.laser.com
```

```
HOSTNAME=localhost.localdomain
```

```
QTDIR=/usr/lib/qt-2.3.1
```

```
LESSOPEN=|/usr/bin/lesspipe.sh %s
```

```
KDEDIR=/usr
```

```
USER=lei
LS_COLORS=
MACHTYPE=i386-redhat-linux-gnu
MAIL=/var/spool/mail/lei
INPUTRC=/etc/inputrc
LANG=en_US
LOGNAME=lei
SHLVL=1
SHELL=/bin/bash
HOSTTYPE=i386
OSTYPE=linux-gnu
HISTSIZE=1000
TERM=ansi
HOME=/home/lei
PATH=/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/lei/bin
_=./main
```

我们看到,程序将"./main"作为第1个命令行参数,所以我们一共有 3 个命令行参数。这可能与大家平时习惯的说法有些不同,小心不要搞错了。

exec 函数族的参数:

现在回过头来看一下 `exec` 函数族,先把注意力集中在 `execve` 上:

```
int execve(const char *path, char *const argv[], char *const envp[]);
```

对比一下 `main` 函数的完整形式,看出问题了吗? 是的,这两个函数里的 `argv` 和 `envp` 是完全一一对应的关系。`execve` 第 1 个参数 `path` 是被执行应用程序的完整路径,第 2 个参数 `argv` 就是传给被执行应用程序的命令行参数,第 3 个参数 `envp` 是传给被执行应用程序的环境变量。

分成两组:留心看一下这 6 个函数还可以发现,前 3 个函数都是以 `execl` 开头的,后 3 个都是以 `execv` 开头的,它们的区别在于,`execv` 开头的函数是以

"char *argv[]"这样的形式传递命令行参数，而 `execl` 开头的函数采用了我们更容易习惯的方式，把参数一个一个列出来，然后以一个 `NULL` 表示结束。这里的 `NULL` 的作用和 `argv` 数组里的 `NULL` 作用是一样的。

环境变量的设置和默认值：在全部 6 个函数中，只有 `execle` 和 `execve` 使用了 `char *envp[]` 传递环境变量，其它的 4 个函数都没有这个参数，这并不意味着它们不传递环境变量，这 4 个函数将把默认的环境变量不做任何修改地传给被执行的应用程序。而 `execle` 和 `execve` 会用指定的环境变量去替代默认的那些。

参数中 `Path` 和 `file` 的区别：还有 2 个以 `p` 结尾的函数 `execlp` 和 `execvp`，乍看起来，它们和 `execl` 与 `execv` 的差别很小，事实也确是如此，除 `execlp` 和 `execvp` 之外的 4 个函数都要求，它们的第 1 个参数 `path` 必须是一个完整的路径，如 `"/bin/ls"`；而 `execlp` 和 `execvp` 的第 1 个参数 `file` 可以简单到仅仅是一个文件名，如 `"ls"`，这两个函数可以自动到环境变量 `PATH` 制定的目录里去寻找。

举例 1 及其源代码：（第一个使用方式）

```
/*pexec.c的源代码 */
#include <unistd.h>
#include <stdio.h>
int main()
{
printf("Running ps with execlp\n");

execlp("ps", "ps", "-ax", 0);

printf("Done.\n");
exit(0);

}
```

举例 1 的运行：注：列举的进程里面没有 `pexec` 自己

```
$ ./pexec
```

```
Running ps with execlp
PID TTY STAT TIME COMMAND
1 ? S 0:00 init
7 ? S 0:00 update (bdfush)
...
146 v01 S N 0:00 oclock
294 pp0 R 0:00 ps -ax
```

举例 2 及其源代码：（6 个 function 都调用一次） （第二个使用方式）

```
/* exec.c */
#include <unistd.h>

main()
{
    char *envp[]={ "PATH=/tmp",
                   "USER=lei",
                   "STATUS=testing",
                   NULL};

    char *argv_execl[]={ "echo", "excuted by execl", NULL};
    char *argv_execlp[]={ "echo", "executed by execlp",
                           NULL};
    char *argv_execlve[]={ "env", NULL};

    if(fork()==0)
        if(execl("/bin/echo", "echo",
                 "executed by execl", NULL)<0)
            perror("Err on execl");

    if(fork()==0)
        if(execlp("echo", "echo",
                  "executed by execlp", NULL)<0)
            perror("Err on execlp");

    if(fork()==0)
        if(execl("/usr/bin/env", "env",
                 NULL, envp)<0)
            perror("Err on execlve");
```

```

        if(fork()==0)
            if(execv("/bin/echo",
argv_execv)<0)
                perror("Err on execv");

        if(fork()==0)
            if(execvp("echo", argv_execvp)<0)
                perror("Err on execvp");

        if(fork()==0)
            if(execve("/usr/bin/env",
argv_execve, envp)<0)
                perror("Err on execve");
    }

```

举例 2 及其运行:

程序里调用了 2 个 Linux 常用的系统命令，**echo** 和 **env**。**echo** 会把后面跟的命令行参数原封不动的打印出来，**env** 用来列出所有环境变量。

由于各个子进程执行的顺序无法控制，所以有可能出现一个比较混乱的输出--各子进程打印的结果交杂在一起，而不是严格按照程序中列出的次序。

最常见的错误:

大家在平时的编程中，如果用到了 **exec** 函数族，一定记得要加错误判断语句。因为与其他系统调用比起来，**exec** 很容易受伤，被执行文件的位置，权限等很多因素都能导致该调用的失败。

最常见的错误是:

- I 找不到文件或路径，此时 **errno** 被设置为 **ENOENT**;
- I 数组 **argv** 和 **envp** 忘记用 **NULL** 结束，此时 **errno** 被设置为 **EFAULT**;
- I 没有对要执行文件的运行权限，此时 **errno** 被设置为 **EACCES**。

3.5.4.10 system() 系统调用

复习：在 Linux 中，创造新进程的方法只有一个，就是我们正在介绍的 `fork`。其他一些库函数，如 `system()`，看起来似乎它们也能创建新的进程，如果能看一下它们的源码就会明白，它们实际上也在内部调用了 `fork`。包括我们在命令行下运行应用程序，新的进程也是由 `shell` 调用 `fork` 制造出来的。

原型:

```
#include <stdlib.h>
int system (const char *string);
```

功能:

The system function runs the command passed to it as string and waits for it to complete. The command is executed as if the command `$ sh -c string` has been given to a shell.

【也就是说，system要使用一个shell来启动指定的程序】

返回值:

system returns 127 if a shell can't be started to run the command and -1 if another error occurs. Otherwise, system returns the exit code of the command.

举例及其源代码:

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
    printf("Running ps with system\n");

    system("ps -ax");

    printf("Done.\n");

    exit(0);
}
```

举例的运行:

```
$ ./system
Running ps with system

PID TTY STAT TIME COMMAND
1 ? S 0:00 init
7 ? S 0:00 update (bdf flush)
...
146 v01 S N 0:00 oclock
256 pp0 S 0:00 ./system
257 pp0 R 0:00 ps -ax

Done.
```

缺点:

In general, using system is a far from ideal way to start other processes, because it invokes the desired program using a shell. This is both inefficient, because a shell is started then the program is started, and also quite dependent on the installation for the shell and environment that is used.

3.5.4.11 小节

下面就让我用一些形象的比喻，来对进程短暂的一生作一个小小的总结：

- l 随着一句 **fork**，一个新进程呱呱落地，但它这时只是老进程的一个克隆。
- l 然后随着 **exec**，新进程脱胎换骨，离家独立，开始了为人民服务的职业生涯。
- l 人有生老病死，进程也一样，它可以是自然死亡，即运行到 **main** 函数的最后一个"}"，从容地离我们而去；也可以是自杀，自杀有 2 种方式，一种是调用 **exit** 函数，一种是在 **main** 函数内使用 **return**，无论哪一种方式，它都可以留下遗书，放在返回值里保留下来；它还甚至能被谋杀，被其它进程通过另外一些方式结束他的生命。（例如信号机制）
- l 进程死掉以后，会留下一具僵尸，**wait** 和 **waitpid** 充当了殓尸工，把僵尸推去火化，使

其最终归于无形。

这就是进程完整的一生。

3.5.5 信号处理 API

3.5.5.1 信号概述

我们在学习微机原理的时候，知道 `cpu` 支持内部中断（一般又称为异常）和外部中断。而在应用程序开发中，异常和中断的体现是利用信号来完成的。

简单的说，一个信号就是一条消息，它通知一个进程某种类型的事件已经在系统中发生了。它是异步产生和异步处理的【从概念上讲，它类似于 windows 程序开发中的消息，但是实现上要简单的多。比如它没有复杂的消息队列机制，而且范围也少的多，比如（当然是废话了，linux 内核本身不支持 GUI）不支持鼠标单击，双击，窗口大小变化等消息。】

具体的说，信号是 linux 系统响应某些状况而产生的事件，进程在接收到信号的时候，会采取相应的行动。信号是因为某些错误条件（比如内存段冲突、浮点处理器错误或者非法指令），或者由 `shell` 和终端管理器产生而引起的中断（比如用户输入了 `ctrl+D`），或者是程序员在编程中显示的产生指定的自定义信号。信号可以明确的从一个进程产生而发送给另外一个进程，所以，利用信号可以传递简单的信息或者协调进程间的操作行为。【总结三类：硬件、内核系统和用户定义的】

进程可以产生信号，捕获信号和屏蔽信号。

信号的名称是在 `signal.h` 文件里面定义的，下面列举了 linux 支持的常用信号及其含义。

Number	Name	Default action	Corresponding event
1	SIGILL	terminate	Terminated time hangup
2	SIGINT	terminate	Interrupt from keyboard
3	SIGQUIT	terminate	Quit from keyboard
4	SIGILL	terminate	Illegal instruction
5	SIGTRAP	terminate and dump core	Trace trap
6	SIGABRT	terminate and dump core	Abort signal from a math library
7	SIGBUS	terminate	Bus error
8	SIGFPE	terminate and dump core	Floating point exception
9	SIGKILL	terminate*	Kill program
10	SIGUSR1	terminate	User-defined signal 1
11	SIGSEGV	terminate and dump core	Invalid memory reference (seg fault)
12	SIGUSR2	terminate	User-defined signal 2
13	SIGPIPE	terminate	Write to a pipe with no reader
14	SIGALRM	terminate	Timer signal from a law timer
15	SIGTERM	terminate	Software termination signal
16	SIGSTKFLT	terminate	Stack fault on copy-on-write
17	SIGCHLD	ignore	A child process has stopped or terminated
18	SIGCONT	ignore	Continue process if stopped
19	SIGSTOP	stop until next SIGCONT*	Stop signal not from terminal
20	SIGTSTP	stop until next SIGCONT	Stop signal from terminal
21	SIGTTIN	stop until next SIGCONT	Background process read from terminal
22	SIGTTOU	stop until next SIGCONT	Background process write to terminal
23	SIGURG	ignore	Urgent condition on socket
24	SIGXCPU	terminate	CPU time limit exceeded
25	SIGXFSZ	terminate	File size limit exceeded
26	SIGVTALRM	terminate	Virtual timer expired
27	SIGPROF	terminate	Profiling timer expired
28	SIGWINCH	ignore	Window size changed
29	SIGIO	terminate	I/O now possible on a descriptor
30	SIGPWR	terminate	Power failure

经常使用的：SIGINT(ctrl+c 来自键盘的中断),SIGKILL（杀死进程）,SIGCHLD（一个子进程停止了），SIGQUIT（来自键盘的退出）,SIGSTP(来自中断的停止信号，ctrl+Z)

3.5.5.2 信号的发送

方式一：利用 kill 命令发送信号：

```
unix> kill -9 15213
```

9 号就是 SIGKILL 信号

方式二：从键盘或者终端发送信号（其实给发送前台进程组的每一个进程），比如 ctrl+c 或者 ctrl+z 等。

方式三：利用 kill 函数发送信号

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

方式四：利用 alarm 函数向自己发送信号

```
#include <unistd.h>
unsigned int alarm(unsigned int secs);
```

3.5.5.3 信号的处理

处理的方式有三个：The process can either ignore the signal, terminate, or *catch* the signal by executing a user-level function called a *signal handler*.

定义：以上三个方式的任何一个发生了，我们可以称信号被接收了：*Receiving a signal. A destination process receives a signal when it is forced by the kernel to react in some way to the delivery of the signal.*

特点：任何时刻，一个发送出的而没有被处理的信号（我们称之为待处理信号）【A signal that has been sent but not yet received is called a *pending signal*. At any point in time, there can be at most one pending signal of a particular type.】

引入函数：每一个信号都有一个默认的处理方式，在上面的表中我们已经列举了。程序员可以通过 `signal()` 函数来修改和某个信号相关的处理方法（SIGSTOP 和 SIGKILL 的默认处理方法是不能修改的）

原型：

```
#include <signal.h>
typedef void handler_t(int)

handler_t *signal(int signum, handler_t
*handler)
```

举例及其源代码：

```

1 #include "csapp.h"
2
3 void handler(int sig)
4 {
5     static int boop = 0;
6
7     printf("handler: %d\n", sig);
8     if (boop < 5)
9         alarm(1); /* next SIGALRM will be delivered in 1s */
10    else {
11        printf("boomer!\n");
12        exit(0);
13    }
14 }
15
16 int main()
17 {
18     signal(SIGALRM, handler); /* install SIGALRM handler */
19     alarm(1); /* until SIGALRM will be delivered in 1s */
20
21     while (1) {
22         /* signal handling routine executed here each time */
23     }
24     exit(0);
25 }

```

【注意程序来源于csapp，里面使用了Signal（）】

举例运行：

```
unix> ./alarm
```

BEEP

BEEP

BEEP

BEEP

BEEP

BOOM!

3.5.5.4 信号处理问题

当一个程序需要捕捉多个信号的时候，有一些细微的问题需要考虑：

- n 待处理信号被阻塞（*Pending signals can be blocked.*）
- n 待处理信号不会排队（*Pending signals are not queued.*）
- n 系统调用可以被中断（*System calls can be interrupted.*）

具体的讲解略。Csapp上列举了三个很好的例子。请同学作为课后作业，自行完成测试和运行。

3.5.5.5 可移植的信号处理

引入：由于不同的 unix 系统直接，对信号处理的语义有差别，比如：一个被中断的系统调用是重启还是永久放弃，不同的 unix 实现可能有不同。因此为了保证程序的可移植性。POSIX 定义了一个标准的 `sigaction()` 函数，它运行用户显示的指定信号处理函数的语义。

原型：

```
#include <signal.h>

int sigaction(int signum, struct sigaction
*act, struct sigaction *oldact);
```

3.6 文件的读写

3.6.1 概述

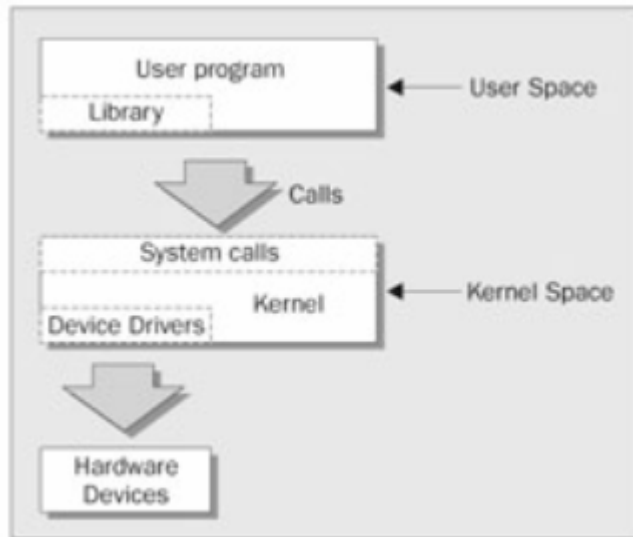
在 linux 系统中，对文件进行读写的函数有两组，一个是直接调用系统 API 函数，还有一个就是利用我们在 c 语言课程里面学习的标准 c 函数库。

这两组函数的命名很相似，比如 `open()` 和 `fopen()`，`read()` 和 `fread()` 等等。

在文件的输入和输出操作中直接使用 API 函数，可能会产生效率比较低的问题，原因有两个：一个是调用 API 函数的时候，进程会不停的在用户态和内核态切换，而态的切换是要耗费时间和系统资源的；另外一个原因就是不同的硬件的驱动实现不同，因此读写的底层实现上，可能一次读写的数据大小是不同的。

因此一般的标准 c 库本身实现了一个缓冲处理，在必要的时候，才产生真正意义上的硬盘硬件的读写操作。【注意：c 库的 io 操作和 API 的 io 操作直接有 buffer 处理，而 `read()` 和 `write()` 的实现中，本身也使用了 buffer，这个 buffer 是文件系统和硬盘驱动程序直接的。】

因此，各种文件函数、用户、设备驱动程序和内核之间的关系可以用下面的图来说明。



3.6.2 常用文件相关的 API

3.6.2.1 write () 函数

原型:

```
#include <unistd.h>

size_t write(int fildes, const void *buf, size_t
nbytes);
```

举例程序:

```
#include <unistd.h>
#include <stdlib.h>
int main()
{
if ((write(1, "Here is some data\n", 18)) != 18)

    write(2, "A write error has occurred on file
descriptor 1\n",46);

exit(0);
}
```

3.6.2.2 read () 函数

原型:

```
#include <unistd.h>

size_t read(int fildes, void *buf, size_t nbytes);
```

举例程序:

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

int main()
{
    char buffer[128];
    int nread;

    nread = read(0, buffer, 128);
    if (nread == -1)
        write(2, "A read error has occurred\n", 26);

    if ((write(1,buffer,nread)) != nread)
        write(2, "A write error has occurred\n",27);

    exit(0);
}
```

```
$ echo hello there | simple_read
hello there
$ simple_read < draft1.txt
```

3.6.2.3 open () 函数

原型:

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

int open(const char *path, int oflags);
int open(const char *path, int oflags, mode_t mode);
```

oflags 指定读写方式, mode 在创建一个文件的时候, 指定 rwx 权限 (还记得 chmod 命令么?)

举例程序:

```
open ("myfile", O_CREAT, S_IRUSR|S_IXOTH);
```

```
$ ls -ls myfile
```

```
0 -r-----x 1 neil software 0 Sep 22 08:11 myfile
```

3.6.2.4 close () 函数

原型:

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

```
int close(int fildes);
```

3.6.3 程序举例

下面的程序来自于 stevens 的 APU

程序3-3 将标准输入复制到标准输出

```
#include "ourhdr.h"

#define BUFFSIZE 8192

int
main(void)
{
    int     n;
    char    buf[BUFFSIZE];

    while ( (n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");

    if (n < 0)
        err_sys("read error");

    exit(0);
}
```

表3-1 用不同缓存长度进行读操作的时间结果

BUFSIZE	用户 CPU (秒)	系 统 CPU (秒)	时钟时间 (秒)	循环次数
1	23.8	397.9	423.4	1 468 802
2	12.3	202.0	215.2	734 401
4	6.1	100.6	107.2	367 201
8	3.0	50.7	54.0	183 601
16	1.5	25.3	27.0	91 801
32	0.7	12.8	13.7	45 901
64	0.3	6.6	7.0	22 951
128	0.2	3.3	3.6	11 476
256	0.1	1.8	1.9	5 738
512	0.0	1.0	1.1	2 869
1 024	0.0	0.6	0.6	1 435
2 048	0.0	0.4	0.4	718
4 096	0.0	0.4	0.4	359
8 192	0.0	0.3	0.3	180
16 384	0.0	0.3	0.3	90
32 768	0.0	0.3	0.3	45
65 536	0.0	0.3	0.3	23
131 072	0.0	0.3	0.3	12

此测试所用的文件系统是伯克利快速文件系统，其块长为8192字节。(块长也就是block，是一个文件系统的基本大小单位，在linux中，创建一个文件系统的命令mkfs，默认的块的大小就是8192)。系统CPU时间的最小值开始出现在BUFSIZE为8192处，继续增加缓存长度对此时间并无影响。

3.7 gdb 调试

3.7.1 概述

介绍：Linux 包含了一个叫 gdb 的 GNU 调试程序。gdb 是一个用来调试 C 和 C++ 程序的强力调试器。它使你能在程序运行时观察程序的内部结构和内存的使用情况。

功能：

- l 它使你能监视你程序中变量的值。
- l 它使你能设置断点以使程序在指定的代码行上停止执行。
- l 它使你能一行行的执行你的代码。

准备：为了调试一个可执行程序，必须在编译可执行程序的时候，添加调试信息到其中。利用下面的命令完成：

```
$gcc -g -o debug debug.c
```

\$ gdb debug

运行：如果直接在控制台上输入 **gdb** 命令，输出如下：

```
GDB is free software and you are welcome to distribute
copies of it under certain conditions; type "show
copying" to see the conditions. There is absolutely no
warranty for GDB; type "show warranty" for details. GDB
4.14 (i486-slakware-linux), Copyright 1995 Free
Software Foundation, Inc.
(gdb)
```

查询帮助：在 **gdb** 提示符处键入 **help**，将列出命令的分类，主要的分类有：

- * **aliases:** 命令别名
- * **breakpoints:** 断点定义；
- * **data:** 数据查看；
- * **files:** 指定并查看文件；
- * **internals:** 维护命令；
- * **running:** 程序执行；
- * **stack:** 调用栈查看；
- * **statu:** 状态查看；
- * **tracepoints:** 跟踪程序执行。

键入 **help** 后跟命令的分类名，可获得该类命令的详细清单。

基本命令：

命 令	描 述
file	装入想要调试的可执行文件
kill	终止正在调试的程序
list	列出产生执行文件的源代码的一部分
next	执行一行源代码但不进入函数内部
step	执行一行源代码而且进入函数内部
run	执行当前被调试的程序
quit	终止 gdb
watch	使你能监视一个变量的值而不管它何时被改变
break	在代码里设置断点，这将使程序执行到这里时被挂起
make	使你能不退出 gdb 就可以重新产生可执行文件
shell	使你能不离开 gdb 就执行 UNIX shell 命令

3.7.2 举例

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

```
void my_print (char *string)
{
    printf ("The string is %s\n", string);
}
```

```
void my_print2 (char *string)
{
    char *string2;
    int size, i;

    size = strlen (string);
    string2 = (char *) malloc (size + 1);

    for (i = 0; i < size; i++)
        string2[size - i] = string[i];

    string2[size+1] = '\0';

    printf ("The string printed backward is
%s\n", string2);
}
```

```
int main ()
{
    char * my_string = "hello world";

    my_print(my_string);
    my_print2(my_string);
}
```

编译程序:

```
gcc -o test -g test.c
```

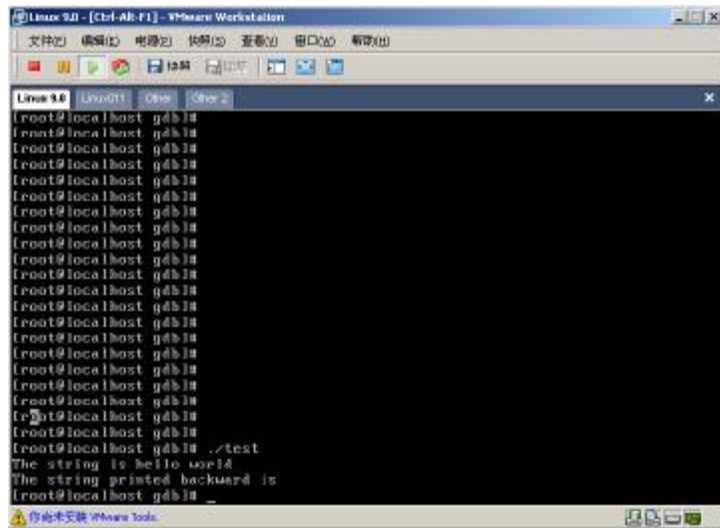
运行程序: 这个程序执行时显示如下结果:

The string is hello world

The string printed backward is

分析：输出的第一行是正确的，但第二行打印出的东西并不是我们所期望的。我们所设想的输出应该是：

The string printed backward is dlrow olleh



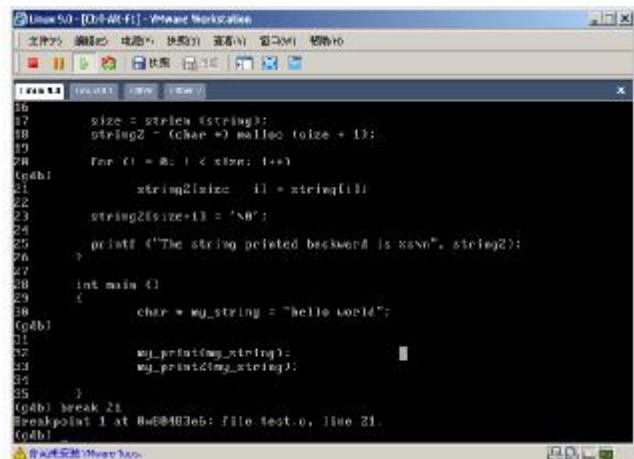
运行 gdb 程序来调试 test：\$ gdb test

这时你能用 gdb 的 run 命令来运行 greeting 了。当它在 gdb 里被运行后结果大约会象这样：

```
(gdb) run
Starting program: /root/greeting
The string is hello there
The string printed backward is
Program exited with code 040
```

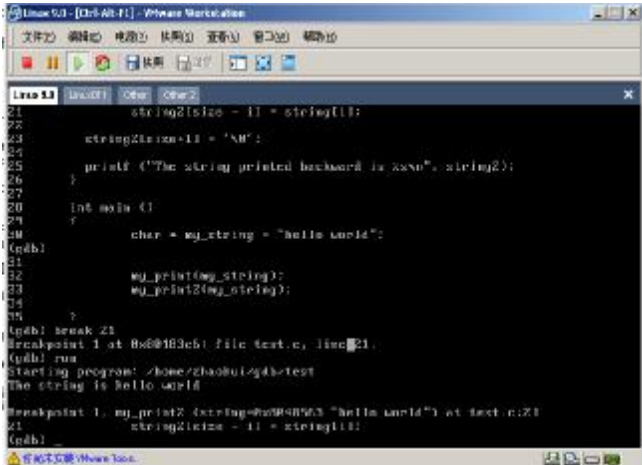
这个输出和在 gdb 外面运行的结果一样。问题是，为什么反序打印没有工作？为了找出症结所在，我们可以在 my_print2 函数的 for 语句后设一个断点，具体的做法是在 gdb 提示符下键入 list 命令四次，列出源代码：

```
(gdb) list 0
(gdb) list
(gdb) list
(gdb) list
```

```
Linux 5.0 - [root@f1] - VMware Workstation
文件(F) 编辑(E) 视图(V) 窗口(W) 帮助(H)
编译 运行 调试 断点 窗口 帮助
Line 11
16 size = strlen(string);
17 string2 = (char *) malloc (size + 1);
18
19 for (i = 0; i < strlen; i++)
20 {
21     string2[size - i] = string[i];
22     string2[size+1] = '\0';
23 }
24 printf ("The string printed backward is %s\n", string2);
25
26 }
27
28 int main ()
29 {
30     char * my_string = "hello world";
31
32     my_printing_string();
33     my_printing_string();
34 }
35
(gdb) break 21
Breakpoint 1 at 0x004030b5: file test.c, line 21.
(gdb)
(gdb) run
Starting program: /home/zhao@f1/gdb-test
The string is hello world
Breakpoint 1, my_print2 (string=0x004030b3 "hello world") at test.c:21
21 string2[size - i] = string[i];
(gdb) _
```

现在再键入 `run` 命令，将产生如下的输出：

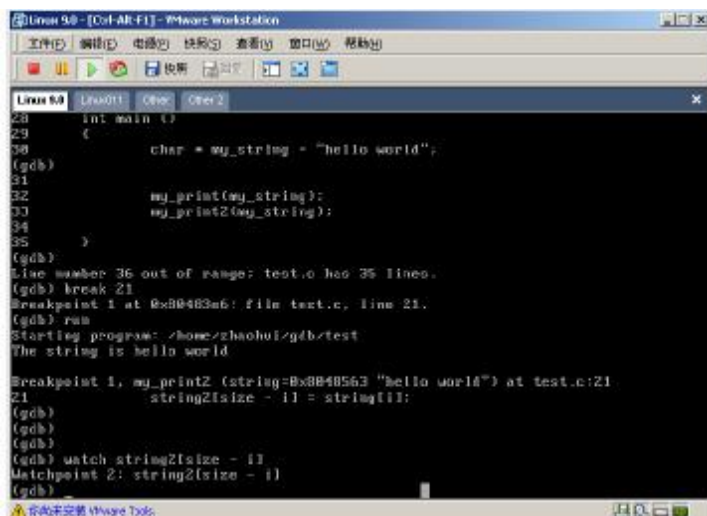


```
Linux 5.0 - [root@f1] - VMware Workstation
文件(F) 编辑(E) 视图(V) 窗口(W) 帮助(H)
编译 运行 调试 断点 窗口 帮助
Line 11
21 string2[size - i] = string[i];
22
23 string2[size+1] = '\0';
24
25 printf ("The string printed backward is %s\n", string2);
26
27 }
28
29 int main ()
30 {
31     char * my_string = "hello world";
32
33     my_printing_string();
34     my_printing_string();
35 }
36
(gdb) break 21
Breakpoint 1 at 0x004030b5: file test.c, line 21.
(gdb) run
Starting program: /home/zhao@f1/gdb-test
The string is hello world
Breakpoint 1, my_print2 (string=0x004030b3 "hello world") at test.c:21
21 string2[size - i] = string[i];
(gdb) _
```

你可以通过设置一个观察 `string2[size - i]` 变量的值的观察点来看出错误是怎样产生的，做法是键入：

`(gdb) watch string2[size - i]`

`gdb` 将作出如下回应：



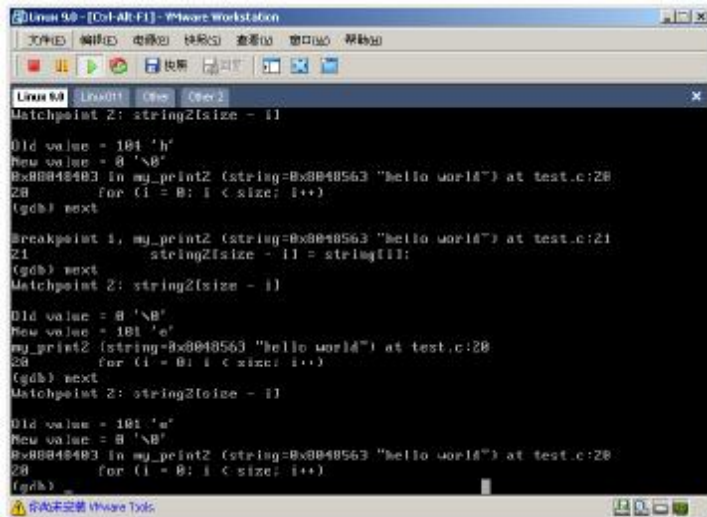
```
Linux 98 - [Ctrl+Alt+F1] - VMware Workstation
文件(F) 编辑(E) 电脑(C) 快捷(S) 查看(V) 窗口(W) 帮助(H)
Linux 98  Linux 98:1  Other  Other:2
28  int main ()
29  {
30      char * my_string = "hello world";
(gdb)
31
32      my_print(my_string);
33      my_print2(my_string);
34
35  }
(gdb)
Line number 36 out of range: test.o has 35 lines.
(gdb) break 21
Breakpoint 1 at 0x80483a6: file test.c, line 21.
(gdb) run
Starting program: /home/zhao/hu/gdb/test
The string is hello world

Breakpoint 1, my_print2 (string=0x8048563 "hello world") at test.c:21
21      string2[size - 1] = string[i];
(gdb)
(gdb)
(gdb)
(gdb) watch string2[size - 1]
Watchpoint 2: string2[size - 1]
(gdb)
```

现在可以用 `next` 命令来一步步的执行 `for` 循环了：

(gdb) next

经过第一次循环后，`gdb` 告诉我们 `string2[size - i]` 的值是 `'h'`。 `gdb` 用如下的显示来告诉你这个信息：



```
Linux 98 - [Ctrl+Alt+F1] - VMware Workstation
文件(F) 编辑(E) 电脑(C) 快捷(S) 查看(V) 窗口(W) 帮助(H)
Linux 98  Linux 98:1  Other  Other:2
Watchpoint 2: string2[size - 1]
Old value = 101 '\0'
New value = 0 '\0'
0x8048483 in my_print2 (string=0x8048563 "hello world") at test.c:28
28      for (i = 0; i < size; i++)
(gdb) next

Breakpoint 1, my_print2 (string=0x8048563 "hello world") at test.c:21
21      string2[size - 1] = string[i];
(gdb) next
Watchpoint 2: string2[size - 1]
Old value = 0 '\0'
New value = 101 'h'
0x8048483 in my_print2 (string=0x8048563 "hello world") at test.c:28
28      for (i = 0; i < size; i++)
(gdb) next
Watchpoint 2: string2[size - 1]
Old value = 101 'h'
New value = 0 '\0'
0x8048483 in my_print2 (string=0x8048563 "hello world") at test.c:28
28      for (i = 0; i < size; i++)
(gdb)
```

如果你再把循环执行下去，你会看到已经没有值分配给 `string2[0]` 了，而它是新串的第一个字符，因为 `malloc` 函数在分配内存时把它们初始化为空(`null`)字符。所以 `string2` 的第一个字符是空字符。这解释了为什么在打印 `string2` 时没有任何输出了。

现在找出了问题出在哪里，修正这个错误是很容易的。你得把代码里写入 `string2` 的第一个字符的偏移量改为 `size - 1` 而不是 `size`。这是因为 `string2` 的大小为 12，但起始偏移量是 0，串内的字符从偏移量 0 到 偏移量 10，偏移量 11 为空字符保留。

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

```
void my_print (char *string)
{
    printf ("The string is %s\n", string);
}
```

```
void my_print2 (char *string)
{
    char *string2;
    int size, i, size2;

    size = strlen (string);
    size2 = size - 1;

    string2 = (char *) malloc (size + 1);

    for (i = 0; i < size; i++)
        string2[size2 - i] = string[i];

    string2[size] = '\0';

    printf ("The string printed backward is
%s\n", string2);
}
```

```
int main ()
{
    char * my_string = "hello world";

    my_print(my_string);
}
```



```
    my_print2(my_string);  
}
```

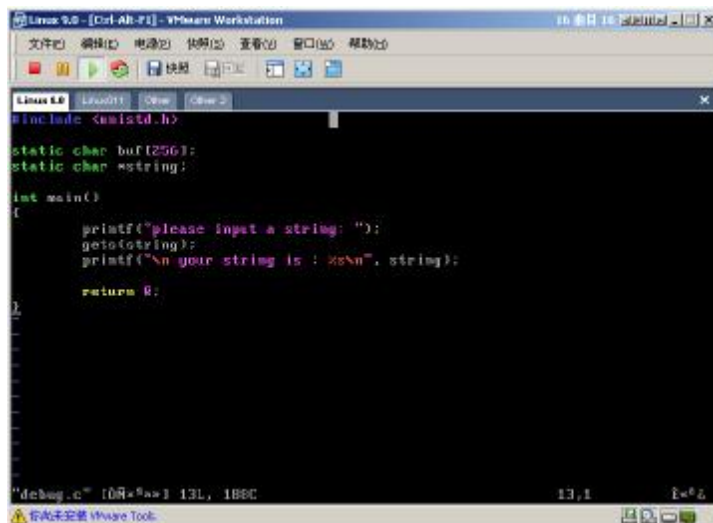
3.7.3 练习

源程序:

*****gdb 使用范例*****

清单 一个有错误的 C 源程序 debug.c

```
1  #include <unistd.h>  
2  #include <stdio.h>  
  
3  static char buff [256];  
  
4  static char* string;  
  
5  int main ()  
6  {  
7      printf ("Please input a string: ");  
8      gets (string);  
9      printf ("\nYour string is: %s\n",  
              string);  
  
10     return 0;  
11 }
```

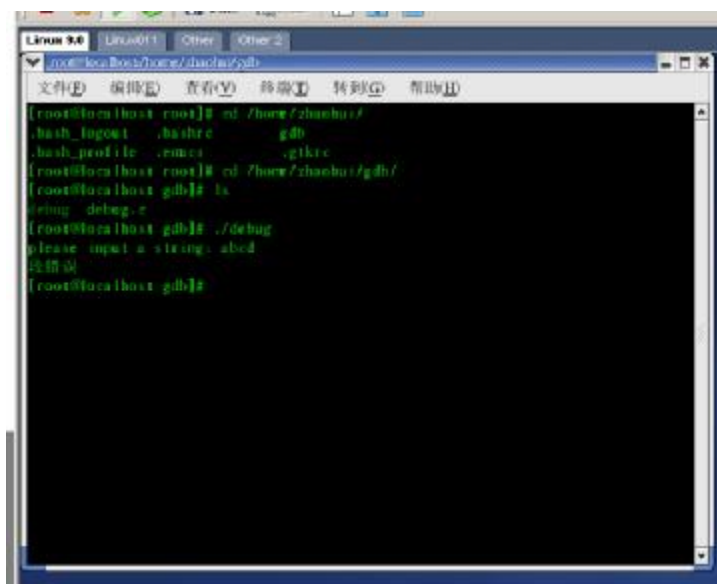


上面这个程序非常简单，其目的是接受用户的输入，然后将用户的输入打印出来。该程序使用了一个未经过初始化的字符串地址 `string`，因此，编译并运行之后，将出现 `Segment Fault` 错误：

```

$ gcc -o debug -g debug.c
$ ./debug
Please input a string: asfd
Segmentation fault

```



请利用 `gdb` 进行调试这个程序，查找该程序中出现的错误。

练习使用 `where` 命令查看程序出错的地方；利

用 `list` 命令查看代码; 利用 `print` 命令查看变量的值; 利用 `break` 命令设置断点; 利用 `set variable` 命令修改 变量的取值。

3.8 shell 程序设计

3.8.1 什么是 shell

shell 的功能之一: shell 是 Unix/Linux 系统中一个重要的层次, 它是用户与系统交互作用的界面。在以前介绍 Linux 命令时, Shell 都作为命令解释程序出现: 它接收用户输入的命令, 进行分析, 创建子进程, 由子进程实现命令所规定的功能, 等子进程终止工作后发出提示符。这是 Shell 最常见的使用方式。

shell 的功能之二: Shell 还是一种高级编程语言, 它有变量、关键字, 有各种控制语句, 如 `if`、`case`、`while`、`for` 等语句, 支持函数模块, 有自己的语法结构。利用 Shell 程序设计语言可以编写出功能很强、但代码简单的程序。特别是它把相关的 Linux 命令有机地组合在一起, 可大大提高编程的效率, 充分利用 Linux 系统的开放性能, 能够设计出适合自己要求的命令。

以上的功能二也是 unix/linux 程序设计的特色: unix/linux 是建立并高度依赖于代码再使用的基础上。你可以编写一个小巧而且简单的工具, 其他人把它作为另外一根链条上的某个环节来构成一条命令。

举例来说: `ifconfig | grep -1 eth0 | cut -s -d ' ' -f16 | grep -0 Mask | cut -d ':' -f2 > /eth0.txt`

shell 的类型: Shell 的概念最初是在 Unix 操作系统中形成和得到广泛应用的。Unix 的 Shell 有很多种类, 比如 `sh`, `csh`, `tcsh`, `ksh` 和 `bash` 等。Linux 系统继承了 Unix 系统中 Shell 的全部功能, 现在默认使用的是 `bash`。

Shell 的特点:

- (1) 把已有命令进行适当组合构成新的命令。
- (2) 提供了文件名扩展字符(通配符, 如*、?、[]), 使得用单一的字符串可以匹配多个文件名, 省去键入一长串文件名的麻烦。
- (3) 可以直接使用 Shell 的内置命令, 而不需创建新的进程, 如 Shell 中提供的 cd、echo、exit、pwd、kill 等命令。为防止因某些 Shell 不支持这类命令而出现麻烦, 许多命令都提供了对应的二进制代码, 从而也可以在新进程中运行。
- (4) Shell 允许灵活地使用数据流, 提供通配符、输入/输出重定向、管道线等机制, 方便了模式匹配、I/O 处理和数据传输。
- (5) 结构化的程序模块, 提供了顺序流程控制、条件控制、循环控制等。
- (6) Shell 提供了在后台执行命令的能力。
- (7) Shell 提供了可配置的环境, 允许创建和修改命令、命令提示符和其它的系统行为。

Shell 脚本: Shell 提供了一个高级的命令语言, 能够创建从简单到复杂的程序。这些 Shell 程序称为 Shell 脚本, 利用 Shell 脚本, 可把用户编写的可执行程序与 Unix 命令结合在一起, 当作新的命令使用, 从而便于用户开发新的命令。比如 startx 和 rc.d 目录下的各个子目录下的文件就是这样的 shell 脚本文件。

3.8.2 管道和重定向

3.8.2.1 输出数据的重新定向

```
$ ls -l > lsoutput.txt
$ kill -9 1234 > killout.txt 2>killerr.txt
```

3.8.2.2 输入数据的重新定向

```
$ more < killout.txt
```

3.8.2.3 管道

```
$ ls | more
$ ps | sort | more
$ ps -xo comm | sort | uniq | grep -v sh | more
```

3.8.3 shell 和程序设计

3.8.3.1 交互式程序

可以不建立脚本文件，而是临时的在 **shell** 提示符下输入一个小的 **shell** 程序，然后让其运行。比如：

```
$ for file in *
> do
> if grep -l POSIX $file
> then
> more $file
> fi
> done
```

3.8.3.2 编写脚本程序

过程描述：建立 **Shell** 脚本的步骤同建立普通文本文件的方式相同，利用编辑器（如 **vi**）进行程序录入和编辑加工。例如，要建立一个名为 **ex1** 的 **Shell** 脚本，可在提示符后输入命令：**\$ vi ex1** 进入 **vi** 的插入方式后，就可录入程序行。完成编辑之后，将编辑缓冲区内容写入文件中，返回到 **Shell** 命令状态。

一般而言，一个 **shell** 脚本文件是以注释开始的，比如 **#!/bin/sh** 然后就是功能描述等注释。它是注释语句的一个特殊形式，**#!** 字符告诉 **linux** 系统同一行上紧跟在后面的那个参数就是用来执行脚本文件的程序。

3.8.3.3 执行一个脚本文件

执行 **Shell** 脚本的方式基本上有三种：

(1)输入定向到 Shell 脚本

比如：\$ bash <ex1.sh

(2)以脚本名作为参数

其一般形式是：\$ bash 脚本名 [参数]

比如：

\$ bash ex2.sh /usr/meng /usr/zhang

(3)将 Shell 脚本的权限设置为可执行，然后在提示符下直接执行它。比如 \$ chmod a+x ex2.sh，然后就可以 \$/路径名/ex2.sh

3.8.4 shell 程序设计的语法

3.8.4.1 变量

shell 脚本中的变量包括自定义变量、环境变量和参数变量。

自己定义的变量，比如：

```
#!/bin/sh
myvar="Hi there"
echo $myvar
echo "$myvar"
echo '$myvar'
echo \myvar
```

环境变量：除了常用的 PATH 和 HOME

\$PS1	A command prompt, usually \$.
\$PS2	A secondary prompt, used when prompting for additional input, usually >.
\$IFS	An input field separator. A list of characters that are used to separate words when the shell is reading input, usually space, tab and new line characters.
\$0	The name of the shell script
\$#	The number of parameters passed.
\$\$	The process ID of the shell script, often used inside a script for generating unique temporary filenames, for example /tmp/tmpfile.\$\$

参数变量：

Parameter Variable	Description
\$1, \$2,	The parameters given to the script.
\$*	A list of all the parameters, in a single variable, separated by the first character in the environment variable IFS.
\$@	A subtle variation on \$*, that doesn't use the IFS environment variable.

3.8.4.2 条件测试

条件测试可以使用 **test** 命令或者 **[]** 来完成, 比如:
判断一个文件是否存在可以使用下面的语句:

```
if test -f fred.c
then
...
fi
或者
if [ -f fred.c ]
then
...
fi
```

条件测试可以分成 3 类: 字符串比较、算术比较和文件有关的测试

String Comparison	Result
string1 = string2	True if the strings are equal.
string1 != string2	True if the strings are not equal.

-n string	True if the string is not null.
-z string	True if the string is null (an empty string).

Arithmetic Comparison	Result
expression1 -eq expression2	True if the expressions are equal.
expression1 -ne expression2	True if the expressions are not equal
expression1 -gt expression2	True if expression1 is greater than expression2.
expression1 -ge expression2	True if expression1 is greater than or equal to expression2.
expression1 -lt expression2	True if expression1 is less than expression2.
expression1 -le expression2	True if expression1 is less than or equal to expression2.
! expression	True if the expression is false, and vice versa.

File Conditional	Result
-d file	True if the file is a directory.
-e file	True if the file exists.
-f file	True if the file is a regular file.
-g file	True if set-group-id is set on file.
-r file	True if the file is readable.
-s file	True if the file has non-zero size.
-u file	True if set-user-id is set on file.
-w file	True if the file is writeable.
-x file	True if the file is executable.

3.8.4.3 控制结构

if 语句:

```

if condition
then
statements
else
statements
fi
```

elif 语句:

```

if [ $timeofday = "yes" ]
then
echo "Good morning"
elif [ $timeofday = "no" ]; then
echo "Good afternoon"
else
echo "Sorry, $timeofday not recognized. Enter yes
or no"
exit 1
fi
```

for 语句:

```

for variable in values
do
statements
done
```

```

for foo in bar fud 43
do
echo $foo
done
```



```
exit 0
```

输出是:

```
bar
```

```
fud
```

```
43
```

while 语句:

```
while condition do
```

```
statements
```

```
done
```

```
while [ "$trythis" != "secret" ]; do
```

```
echo "Sorry, try again"
```

```
read trythis
```

```
done
```

until 语句:

```
until condition
```

```
do
```

```
statements
```

```
done
```

case 语句:

```
case variable in
```

```
pattern [ | pattern] ...) statements;;
```

```
pattern [ | pattern] ...) statements;;
```

```
...
```

```
esac
```

```
case "$timeofday" in
```

```
yes | y | Yes | YES ) echo "Good Morning";;
```

```
n* | N* ) echo "Good Afternoon";;
```

```
* ) echo "Sorry, answer not recognized";;
```

```
esac
```

and 和 or 操作符:

```
statement1 && statement2 && statement3 && ...
```

```
statement1 || statement2 || statement3 || ...
```

语句块：和 c 语言一样，使用{ }。

3.8.4.4 函数

```
function_name ()  
{  
    statements  
}
```

shell 中的函数也可以使用 return 返回数值，比如：

```
#!/bin/sh  
  
yes_or_no() {  
    echo "Is your name $* ?"  
    while true  
    do  
        echo -n "Enter yes or no: "  
        read x  
        case "$x" in  
            y | yes ) return 0;;  
            n | no )  return 1;;  
            * )      echo "Answer yes or no"  
        esac  
    done  
}
```

3.8.4.5 命令

shell 支持一些内部 (build-in) 命令，他们是在 shell 内部实现的，不能作为外部程序在提示符下运行和使用，大部分内部命令是 POSIX 标准的组成部分。

常用的内部命令有：break, : 命令, continue, . 命令, echo, eval, exec, exit, export, expr, printf, return, set, shift, trap, unset 等。

3.8.4.6 命令的执行

比如：echo The current users are \$(who)

When we're writing scripts, we often need to capture the result of a command's execution for use in the shell script, i.e. we want to execute a command and put the output of the command in a variable. We do this using the `$(command)` syntax,

3.8.4.7 here 文档

它从一个 `shell` 脚本向一条命令输入数据的一个特殊方法。它允许命令在执行的时候，就好像是在读一个文件或者读键盘意一样，而它实际上是从脚本程序里面得到输入数据的。

格式：<<然后一个特殊的自己定义的字符串序列。

比如：

1. Let's start with a file, `a_text_file`, containing:

```
That is line 1
That is line 2
That is line 3
That is line 4
```

2. We can edit this file using a combination of a here document and the `ed` editor:

```
#!/bin/sh

ed a_text_file <<!FunkyStuff!
3
d
.,\$/is/was/
w
q
!FunkyStuff!

exit 0
```

If we run this script, the file now contains:

```
That is line 1
That is line 2
That was line 4
```

3.8.4.8 调试脚本程序

有两个方法：一个是调用 `shell` 时加上命令行，一个是使用 `set` 命令。

Command Line Option	set Option	Description
<code>sh -n <script></code>	<code>set -o noexec</code> <code>set -n</code>	Checks for syntax errors only; doesn't execute commands.
<code>sh -v <script></code>	<code>set -o verbose</code> <code>set -v</code>	Echoes commands before running them.
<code>sh -x <script></code>	<code>set -o xtrace</code> <code>set -x</code>	Echoes commands after processing on the command line.
	<code>set -o nounset</code> <code>set -u</code>	Gives an error message when an undefined variable is used.

3.8.5 举例

下面是/etc/rc.d/rc.local 的脚本

```
#!/bin/sh
#
# This script will be executed *after* all the other init scripts.
# You can put your own initialization stuff in here if you don't
# want to do the full Sys V style init stuff.

if [ -f /etc/redhat-release ]; then
    R=$(cat /etc/redhat-release)

    arch=$(uname -m)
    a="a"
    case "$arch" in
        _a*) a="an";;
        _i*) a="an";;
    esac

    NUMPROC=$(egrep -c "^cpu[0-9]+" /proc/stat)
    if [ "$NUMPROC" -gt "1" ]; then
        SMP="$NUMPROC-processor "
        if [ "$NUMPROC" = "8" -o "$NUMPROC" =
"11" ]; then
            a="an"
        else
            a="a"
        fi
    fi

    # This will overwrite /etc/issue at every boot.  So, make
    any changes you
    # want to make to /etc/issue here or you will lose them
    when you reboot.
    echo "" > /etc/issue
    echo "$R" >> /etc/issue
    echo "Kernel $(uname -r) on $a $SMP$(uname -m)" >>
/etc/issue

    cp -f /etc/issue /etc/issue.net
```

```
echo >> /etc/issue  
fi  
exec syslogd
```

3.9 其他相关主题（列举而已）

3.9.1 Beginning linux programming

1. Unix 环境
2. 终端
3. Curse 库
4. 数据管理（内存管理和文件 locking）
5. RCS 和 CVS 源代码控制系统
6. POSIX 线程
7. 进程间通信机制
8. Socket 网络编程
9. Xwindows 程序设计
10. GTK+和 GNOME 程序设计

3.9.2 linux programming unleashed

略

第三讲作业

1. Makefile 的进一步学习：阅读《中文 Makefile 的教程》和赵炯的书的 2.10 一节。

2. 结合一个 shell 应用程序实例，锻炼自己的 shell 程序设计和编程能力，可以从 www.wrox.com 下载 Beginning_Linux_Programming_2nd_Edition code.tar.tar（也可以从我这里拷贝），阅读和分析其中第二章的实例（一个 cd 管理程序）。或者系统调用跟我学(4) 进程管理相关的系统调用之三

3. 参考网络上资源或者李善平的边学边干的第一章，学习如何利用 linux 的进程控制 API 来开发一个简单的 shell 程序。这也是复习本讲的进程控制相关 API 的一个好实例。

4. 有兴趣的同学，可以学习 pdf 文档或者 csapp 一书的第三章。从汇编语言的角度理解 c 语言。当然，c expert 和 c trap and pitfall 也有一些讲解。

5. 阅读和执行 csapp 第八章的关于进程控制和信号的程序（8.4 和 8.5 一章）。