Unix/Linux 程序设计实验指导

石家庄经济学院信息工程学院

2014.7

目录

第 1 章 Linux 使用基础

诸多类 Unix 操作系统的设计令人惊叹。即便是在数十年后的今天，Unix 式的操作系统架构仍是有史以来的最佳设计之一。这种架构最重要的一个特性就是命令行界面或 shell。Shell 环境使得用户能与操作系统的核心功能进行交互，为用户提供使用操作系统的接口。它是命令语言、命令解释程序及程序设计语言的统称。如果把 Linux 内核想象成一个球体的中心，shell 就是围绕内核的外层。用户可以通过输入要执行的命令或利用 Shell 脚本编程向 Linux 传递指令，内核会做出相应的反应。

本章使用的是 Bash(Bourne Again Shell)，它是目前大多数 GNU/Linux 系统默认的 shell 环境。本章的主要目的是让读者了解并熟悉 shell 环境的基本特性，掌握 Linux 基本命令的使用方法。

1.1 Linux 的基本命令

1.1.1 ls 命令

语法：ls[选项][目录或文件列表]

说明：显示指定工作目录下之内容（列出目前工作目录所含之档案及子目录)。说明：这是用户常用的一个命令之一，因为用户需要不时地查看某个目录的内容。该命令类似于 DOS 下的 dir 命令。在 Linux 系统中，dir 命令也一样可用。

例 1-1 ls 命令列出/usr 目录下详细的信息，包括文件权限、大小、文件的拥有权等。

```
[root@localhost usr]# ls -l
总用量 144
drwxr-xr-x   2 root   root    40960  4 月   4 11:00 bin
drwxr-xr-x   2 root   root     4096 2003-01-25  dict
drwxr-xr-x   2 root   root     4096 2003-01-25  etc
drwxr-xr-x   2 root   root     4096 2003-01-25  games
drwxr-xr-x 105 root   root     8192  7 月  11 09:23 include
drwxr-xr-x   3 root   root     4096 2013-01-28  java
drwxr-xr-x   8 root   root     4096 2013-01-28  kerberos
drwxr-xr-x  82 root   root    45056 2013-01-28  lib
drwxr-xr-x   7 root   root     4096 2013-01-28  libexec
drwxr-xr-x  12 root   root     4096 2013-01-28  local
drwxr-xr-x   2 root   root     4096 2013-01-28  opt
drwxr-xr-x   2 root   root     8192 2013-01-28  sbin
drwxr-xr-x 185 root   root     4096 2013-01-28  share
drwxr-xr-x   5 root   root     4096 2013-01-28  src
lrwxrwxrwx   1 root   root       10 2013-01-28  tmp -> ../var/tmp
drwxr-xr-x   8 root   root     4096 2013-01-28  X11R6
```

ls 有很多功能选项，这些选项赋予 ls 命令很大的能力。使用 man 命令可以查看这些选项的具体用法。

1.1.2 cd 命令

功能：改变工作目录

语法：cd [dirName]

说明：变换工作目录至 dirName。 其中 dirName 表示法可为绝对路径或相对路径。若目录名称省略，,则变换至使用者的 home directory (也就是刚 login 时所在的目录)。另外,"~" 也表示为 home directory 的意思，"." 则是表示目前所在的目录，".." 则表示目前目录位置的上一层目录。为了改变到指定的目录，用户必须拥有对指定目录的执行和读权限。

例 1-2 cd 命令的使用

跳到 /usr/bin/:

cd /usr/bin

跳到自己的 home directory:

cd ~

跳到目前目录的上上两层:

cd ../..

1.1.3 cp命令

功能：文件复制命令，将给出的文件或目录复制到其他文件或目录中。

语法： cp [options] source dest

cp [options] source... directory

选项：

-r 若 source 中含有目录名，则将目录下之文件亦皆依序拷贝至目的地。

-f 若目的地已经有同名的文件存在，则在复制前先予以删除再行复制。

例 1-3 将文件 aaa 复制(已存在)，并命名为 bbb

cp aaa bbb

例 1-4 将所有的 C 语言程式拷贝至 Finished 子目录中

cp *.c Finished

1.1.4 rm 命令

功能：删除一个目录中的一个或多个文件或目录，它可以将某个目录及其下的所有文件及子目录均删除。对于链接文件，只是断开了链接，原文件保持不变。

语法：rm [options] name...

选项： -i 删除前逐一询问确认。

-f 即使原文件属性设为只读，亦直接删除，无需逐一确认。

-r 将目录及以下之文件逐一删除。

例 1-5 删除所有 C 语言程序；删除前逐一询问确认。

rm -i *.c

例 1-6 将 Finished 子目录及子目录中所有文件删除。

rm -r Finished

1.1.5 mv 命令

功能：文件移动命令，将文件或目录改名或将文件由一个目录移动入另一个目录中。该命令如同 MS-DOS 下的 rename 和 move 的组合。

语法：mv [options] source dest

mv [options] source... directory

选项：-i 若目的地已有同名档案,则先询问是否覆盖旧档。

例 1-7 将文件 aaa 更名为 bbb:

mv aaa bbb

例 1-8 将所有的 C 语言程式移至 Finished 子目录中:

mv -i *.c

1.1.6 mkdir 命令

功能：创建一个目录。

语法：mkdir [options] 目录

说明：该命令创建由 dir-name 命名的目录。要求创建目录的用户在当前目录中(dir-name 的父目录中)具有写权限，并且 dirname 不能是当前目录中已有的目录或文件名称。

例 1-9 在 test 目录下创建 test 目录

mkdir /home/test/test2

例 1-10 创建多层目录

mkdir –p /home/test/test2

1.1.7 rmdir 命令

说明：删除一个目录

语法：rmdir [-p] dirName

说明：该命令从一个目录中删除一个或多个字目录项。需要注意的是，一个目录被删除之前必须是空的。

例 1-11 将工作目录下，名为 AAA 的子目录删除:

rmdir AAA

1.1.8 chmod 命令

功能：改变某个文件或目录的访问权限

语法：chmod [options] mode [,mode] 文件

chomd [options] octal-mode 文件

chmod [options] –reference=rfile 文件

说明：chmode 命令非常重要，用于改变文件或目录的访问权限。Linux 系统中的每个文件和目录都有访问权限，用它来确定谁可以通过何种方式对文件和目录进行访问和操作。这个命令有多种用法，如包含字母和操作符表达式的文字设定法和包含数字的数字设定法。其中文本设定法中的选项含义如下：

(1) 文字设定法中的选项含义如下：

• u 表示用户，即文件或目录的所有者。

• g 表示组内用户，即与文件属主有相同组 ID 的所有用户。

• o 表示其他用户。

• a 表示所有用户，它是系统默认值。

(2) 操作符号有 3 种

• +表示添加权限

• -表示取消权限

• =表示赋予给定权限并取消其他所有权限

(3) mode 所表示的权限可以用下列字母组合

• r：可读

• w：可写

• x：可执行

• u：与文件属主拥有一样的权限

• g：与和文件属主同组的用户拥有一样的权限

• o：与其他用户拥有一样的权限

例 1-12 将文件 test.c 设置为所有人可 读

chmod ugo+r test.c

(4) 数字设定法中数字表示的属性含义

每种用户的权限可用 3 位二进制位表示，其中 000、001、010、100 分别对应数字 0、1、2、4，对应含义分别为：0 表示没有权限、1 表示可执行权限，2 表示可写权限，4 表示可读权限。所有数字属性的格式应为 3 个从 0~7 的八进制数，其顺序是(u)(g)(0)。若要设置属性 rwx 属性，则 4+2+1=7；若要设置 rw-属性，则 4+2=6；若要设置 r - x 属性 ，则 4+1=5.

例 1-13 设置文件 test.c 权限为 rwx r-x - -x

chmod 751 test.c

例 1-14 设置文件 test.c 权限为所有用户可读、可写、可执行

chmod a=rwx test.c

chmod 777 test.c

1.1.9 chown 命令

功能：更改某个文件或目录的属主和属组。

语法： chown [options] 用户或组 文件

chown [options] .组 文件

chown [options] -reference = rfile 文件

说明：这是一个常用命令，例如 root 用户把自己的一个文件复制给用户 sunny，为了让 sunny 能够存取这个文件，root 用户应该把这个文件的属主设为 sunny，否则 sunny 无法存取这个文件。

例 1-15 sunny 用户属于 Goup1 组，auto.c 文件现在不属于 sunny 用户，使 sunny 用户能够存取 auto.c 的命令

chown +R sunny. Group1 auto.c

1.1.10 cat 命令

功能：通常 cat 命令用于读取、显示或拼接文件内容。

使用方式：cat [options] [fileName…]

例 1-16 使用 cat 打印文件内容

cat file1 file2 file3

例 1-17 将 file1 的文件内容加上行号后输入 file2 文件中。

cat -n file1 >file2

例 1-18 将 file1 和 file2 的文件内容加上行号(空行不加)之后将其内容追加到 file3 中

cat -b file1 file2 >> file3

1.1.11 touch 命令

功能：touch 命令用于改变文件的时间属性，用 ls –l 可以显示文件的时间属性。

最简单的使用方式，将文件的时间属性改为现在的时间。若文件不存在，系统会建立一个新的文件。

例 1-19 用 touch 建立一个新文件

touch newfile

1.1.12 find 命令

功能：查找满足条件的文件，可以指定需要查找的条件为文件名、类别、时间、大小、权限等组合形式，只有完全相符的才会被打印出来。

例 1-20 将当前目录及其子目录下所有扩展名是 c 的文件列出来。

find . -name "*.c"

例 1-21 将当前目录及其下子目录中所有普通文件列出来。

find . -ftype f

例 1-22 将当前目录及其子目录下所有最近 20 分钟内更新过的文件列出来

find . -ctime -20

1.1.13 grep 命令

功能：grep 命令是 Unix 中用于文本搜索的神奇工具，能够接受正则表达式，生成各种格式的输出。它有大量有趣的选项。

语法： grep [options] PATTERN [FILE...]

grep [options] [-e PATTERN | -f FILE] [FILE...]

说明：grep 命令若要使用正则表达式，需要添加-E 选项，这意味着使用扩展(extended)正则表达式。

例 1-23 搜索包含特定模式的文本行，例如包含字符串“pattern”

grep “pattern” file.txt

this is the line containing pattern

例 1-24 使用 grep 在多个文件中搜索含有特定模式“pattern”行

grep “pattern” file1 file2 file3

例 1-25 使用 grep 搜索包含 a-z 的字符行

grep –E “[a-z]+” file.txt

例 1-26 统计文件或文本中包含匹配字符串的行数

grep –c “test” file.txt

例 1-27 搜索多个文件并找出匹配文本位于哪一个文件中

grep –l “linux” sample1.txt sample2.txt

1.1.14 wc 命令

用法：wc 是一个用于统计的工具，它是 WordCount(单词统计)的缩写。

语法：wc [options] [file…]

说明：可以使用 wc 的各选项来统计行数、单词数和字符数。

例 1-28 统计行数

Wc –l file

cat file | wc –l

例 1-29 统计单词数

wc –w file

cat file | wc –w

例 1-30 统计字符数

wc –c file

cat file | wc –c

1.1.14 ps 命令

功能：报告进程状态。

语法：ps [options]

说明：ps 命令可以查看系统进程的状态，确定有哪些进程正在运行和运行的状态以及进程是否结束，进程有没有僵死，哪些进程占用了过多的资源等。总之大部分信息都可以通过执行 ps 命令得到。

例 1-31 ps 命令

$ps -u

USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND

root 3426 0.0 0.4 5800 1256 pts/0 S 09:47 0:00 bash

root 16216 0.0 0.2 2612 664 pts/0 R 13:37 0:00 ps –u

1.1.15 top 命令

功能：实时监控系统处理器的状态。显示系统中 CPU 最“活跃”的任务列表。

语法：top [-] [d delay] [q] [c][s][S][i]

说明：top 命令与 ps 相似，但是通常会全屏显示，而且会随着进程状态的变化不断更新。对于那些经常引发问题，而用 ps 又难以查看的程序而言，这个命令很有用。也会显示真个系统的信息，为查找问题提供了方便。

1.2 Linux 文件系统

Linux 以目录结构的方式组织文件。每个目录具有一个名字，它可包含其他的文件和目录。目录按照一定的顺序组织在其他目录之下，形成了一种树状结构的组织形式。这种结构可使用户通过将相关文件放到一个目录下，实现对大量文件的管理。每个用户拥有一个主目录，根据需要可在该目录下建立更多的子目录，如图 1-1 所示。表 1-1 对 Linux 文件系统进行详细描述。

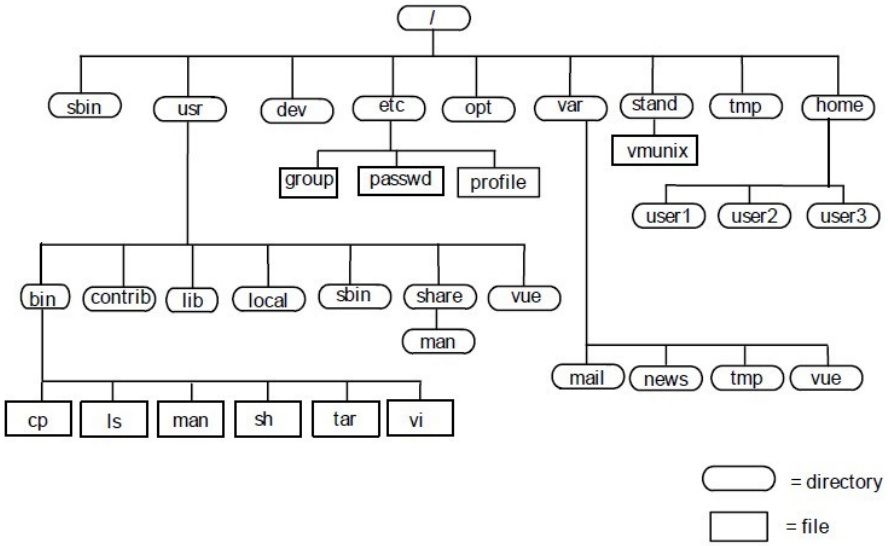

图 1-1 Linux 文件系统结构

表 1-1 Linux 文件系统

/	根目录。文件系统的入口，最高一级目录。“/”是唯一必须挂载的目录也是衍生以下挂载点的根源 。
/bin	主要存放用户经常使用的命令，如 ls, cp, mkdir 等。 这个目录中的文件都是可执行的，一般的用户都可以使用。
/boot	包含 Linux 内核及系统引导程序所需要的文件，比如 vmlinuz initrd.img 文件都位于这个目录中。在一般情况下，GRUB 或 LILO 系统引导管理器也位于这个目录；
/dev	设备文件存储目录。Linux 文件系统的一个闪亮的特性——所有对象都是文件或目录。比如/dev/hda 代表第一块 IDE 硬盘；/dev/cdrom 和/dev/fd0 代表 CDROM 驱动器和 floppy 驱动器，它们都可以读出和写入。又如/dev/dsp 代表扬声器等。
/etc	存放系统程序或者一般工具的配置文件。

/home	普通用户默认存放目录。 Linux 是多用户环境，所以每一个用户都有一个只有自己可以访问的目录(当然管理员也可以访问)。比如有个用户 user，它的主目录就是/home/user。这个目录也保存一些应用对于这个用户的配置，比如 IRC, X 等。
/lib	库文件存放目录。主要存放系统最基本的函数库，几乎所有的应用程序都要用到这些函数库。
/lost found	平时是空的，当系统不正常关机后，这里保存一些文件的片段。
/media	即插即用型存储设备的挂载点自动在这个目录下创建，比如 USB 盘系统自动挂载后，会在这个目录下产生一个目录 ；CDROM/DVD 自动挂载后，也会在这个目录中创建一个目录，类似 cdrom 的目录。这个只有在最新的发行套件上才有。
/misc	存放不好归类的东西。
/mnt	于存放挂载储存设备的挂载目录。比如光驱可以挂载到/mnt/cdrom 。这是一个普通的加载目录，在这里可以加载用户的文件系统或设备。建立/mnt 只是为了使系统更工整的惯例。
/net	主要存和网络相关的文件。
/opt	用来安装可选的应用程序。这个目录包含所有默认系统安装之外的软件和添加的包。
/proc	操作系统运行时，进程(正在运行中的程序)信息及内核信息(比如 cpu、硬盘分区、内存信息等)存放在这里。“/proc”是一个虚拟的目录，由系统运行时产生，是系统内存的映射，可以通过直接访问这个目录来获取系统信息。注意：这个目录的内容不在硬盘上而是在内存里。
/root	超级用户(也叫系统管理员或根用户)的主目录 。
/sbins	主要存放系统管理员使用的管理程序，其他的还有有/usr/sbin、/nsf/local/sbin。
/srv	存放一些服务启动之后需要服务的文件
/sys	系统的核心文件，这个目录是 2.6 内核的一个很大的变化，该目录下安装了 2.6 内核中新出现的一个系统文件 Sysfs，Sysfs 文件系统集成了下面三种文件系统信息：针对进程信息的 proc 文件系统、针对设备的 devfs 文件系统、针对伪终端的 derpts 文件系统。
/tmp	临时文件目录，有时用户运行程序的时候，会产生临时文件。 /tmp 就用来存放临时文件的。/var/tmp 目录和这个目录相似。许多程序在这里建立 lock 文件和存储临时数据。有些系统会在启动或关机时清空此目录。
/usr	包含系统的主要程序、用户自行安装的程序、图形界面需要的文件、共享的目录与文件、命令程序文件、程序库，手册和其他文件等，这些文件一般不需要修改。
/var	含系统执行过程中的经常变化的文件。例如打印机、邮件、新闻等假脱机目录、日志文件、格式化后的手册页以及一些应用程序的数据文件等。建议单独的放在一个分区 。

1.3 在线系统文档

1.3.1 man 手册

通过 man 命令可以访问系统参考手册页(man 手册页)。如果需要阅读 man 自身的参考手册，在命令行下运行“man man”即可。Man 手册页一般只包含参考文档，而不包含指导信息，其信息量以短小精悍出名。

有三个程序提供了对 man 手册页的访问。man 程序显示单个的 man 手册页，apropos 和 whatis 命令可以在一个 man 手册页的集合里搜索关键字。apropos 和 whatis 命令搜索同一数据库；它们的区别是 whatis 命令只显示匹配行，而 apropos 显示包含该单词的所有行。可以尝试如下命令，观察它们的不同之处。

$whatis man

$apropos man

Linux系统中的许多man手册页是LDP[1](Linux文档工程)汇编的巨大文档包的一部分。Linux的man手册共有以下几个章节：

1 Standard commands (标准命令)

2 System calls (系统调用)

3 Library functions (库函数)

4 Special devices (设备说明）

5 File formats （文件格式）

6 Games and toys （游戏和娱乐）

7 Miscellaneous （杂项）

8 Administrative Commands （管理员命令）

如果你想在某一个节进行查询，可以在需要查询的手册页前加一个节号。

在阅读 man 手册页的时候，要记住：许多系统调用和库函数使用同样的名字。在大多数情况下，你希望知道你想链接的库函数信息，而不是这个库函数最终调用的系统调用，那么要用 man 3 function 来获取库函数的接收。因为一些库函数与第 2 节系统调用具有相同的名字。

1.3.2 其他文档

/usr/share/doc 是其他未分类的文档的存放目录。系统安装的大多数软件都把 README 文件、其他格式的文档(包括纯文本、PostScript、PDF 和 HTML)，以及例子安装在/usr/share/doc 目录下。每一个软件包都有自己的子目录，与软件包的名字同名，同时一般都带有软件包的版本号。

[1] Linux文档工程(LDP)为Linux社区提供多种自由文档资源,包括向导 (guide)，常见问答 (FAQ)，入门 (HOWTO) 以及手册页 (man-pages).

第 2 章 开发工具和环境

Linux 下可以使用很多种不同的应用程序开发工具和环境。其中一些比较重要的程序开发工具是每个程序源都应该熟练掌握的。Linux 的发行版本中包括许多稳定的并经实践证明是有效的开发工具；绝大部分工具已经在 Unix 的开发系统中使用了很多年。这些工具既不浮华也不奇特；它们中绝大多数是命令行式的工具。因此，值得每一个程序员花时间去学习。

2.1 Vim 编辑器

Unix 程序开发人员传统上都有很强的尤其是对于编辑器的偏爱。他们所使用的编辑器中最通用的两个是 vi 和 emacs。这两个编辑器的功能都比它们第一眼看上去功能强大得多，而且在本质上它们也有很多不同。Emacs 很大，它有一种属于它自己的操作环境。Eamcs 包含了一个很广泛的手册集合，Emacs 的功能非常强大，读者可以通过参考资料自行学习。

Vi(Visual Interface)是 Linux 系统的第一个全屏幕交互式编辑工具，历经数十年的发展，成为人们主要使用的文本编辑工具。Vi 可以执行输出、删除、查找、替换等文本操作。而且用户可以根据自己的需要对其进行定制，这是其他编辑程序所没有的。

最通用的 vi 版本是 vim（“Vi Imporved”），它是 Vi 编辑器的升级，是 Unix/Linux 操作系统下标准的编辑器。

2.1.1 Vim 基本工作模式

Vim(及 Vi)有 3 个模式：插入模式、命令模式、底行模式。

输入模式：在此模式下可以输入字符，按 ESC 将回到命令模式。

命令模式：可以移动光标、删除字符等。

底行模式：可以保存文件、退出 vim、设置 vim、查找等功能(底行模式也可以看作是特殊的命令模式)。

进入 Vim 时，默认的模式是命令行模式，用户可以根据需要在三种工作模式之间切换，切换方法如图 2-1 所示。

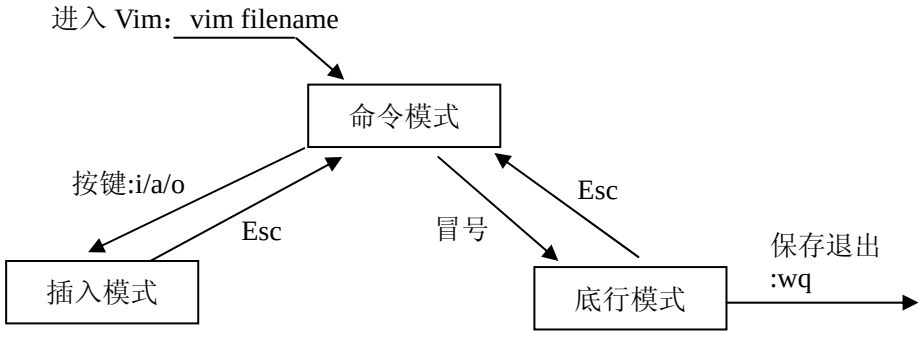

图 2-1 模式切换图

说明

(1)要进入输入模式输入数据时，可以用下列按键：

①按“a”键：从目前光标所在位置的下一个字符开始输入。

②按“i”键：从光标所在位置开始插入新输入的字符。

③按“o”键：新增加一行，并将光标移到下一行的开头。

(2)Vim 的底行模式是指可以在界面最底部的一行显示的输入命令，一般用来执行查找特定的字符串、保存及退出等任务。在命令行模式下输入冒号“：”，就可以进入最后行模式了，还可以使用“？”和“/”键进入最后行模式。

2.1.2 Vim 的启动

在终端界面中输入 vim 即可启动 Vim。也可以输入如下命令：

```
vim file.txt
```

现在 Vim 开始编辑一个名为 file.txt 的文件了。由于这是一个新建文件，因此得到一个空的窗口。屏幕看起来会象下面这样：

```
+----------------------------------------+
|#                                       |
|~                                       |
|~                                       |
|~                                       |
|~                                       |
|"file.txt" [New file]                   |
+----------------------------------------+
        ("#"是当前光标的位置)
```

以波纹线(~)开头的行表示该行在文件中不存在。换句话说，如果 Vim 打开的文件不能充满这个显示的屏幕，它就会显示以波纹线开头的行。在屏幕的底部，有一个消息行指示文件名为 file.txt 并且说明这是一个新建的文件。这行信息是临时的，新的信息可以覆盖它。

2.1.2 Vim 命令行模式操作

1. 插入操作

Vim 是一个多模式的编辑器。就是说，在不同模式下，编辑器的响应是不同的。在命令模式下，你敲入的字符只是命令；而在插入模式，敲入的字符就成为插入的文本。

Vim 刚启动时，它处在命令模式。通过敲入"i"命令(i 是 Insert 的缩写)可以启动插入模式输入文字，这些文字将被插入到文件中。输入错误可以随时修改。例如，在 vim 中编译如下文本：

```
iA very intelligent turtle
Found programming UNIX a hurdle
```

输入"turtle"后，通过输入回车开始一个新行。最后，输入 <Esc> 键退出插入模式而回到命令模式。现在在 Vim 窗口中就有两行文字了：

```
+----------------------------------------+
|A very intelligent turtle               |
|Found programming UNIX a hurdle         |
|~                                       |
|~                                       |
|                                        |
+----------------------------------------+
```

现在，如果再次输入 "i" 命令，Vim 会在窗口的底部显示 --INSERT--（中文模式显示的是--插入-- —— 译者注），这表示正处于插入模式。

```
+----------------------------------------+
|A very intelligent turtle               |
```

```
|Found programming UNIX a hurdle              |
|~                                            |
|~                                            |
|-- INSERT --                                 |
+---------------------------------------------+
```

如果你输入<Esc>回到命令模式，最后一行又变成空白。

在使用 Vim 时经常忘记在什么模式下，可以通过敲<Esc>键，如果 Vim 发出“嘀”一声，就表示已经处于命令模式了。

常用的插入命令如表 2-1 所示。

表 2-1 插入命令

命令	操作说明
i	在当前光标前插入
I	在当前行首插入
a	在当前光标后插入
A	在当前行尾插入
o	在当前行下新开一行
O	在当前行上新开一行
r	替换当前字符
R	替换当前字符及其后的字符，直至按 Esc 键
s	从当前光标位置开始，以输入的文本替换指定数目的字符
S	删除指定数目的行，并以所输入文本替代

2. 移动光标

Vim 移动光标的方式可以分以下几类：

(1) 字符移动

在命令模式下，可以使用以下命令来移动光标。

表 2-2 字符移动命令

命令	操作说明
h	向左移动光标
j	向下移动光标
k	向上移动光标
l	向右移动光标

(2) 移动单词

在命令模式下，使用 w 命令可以将光标向后移动一个单词。在 w 命令前面指定一个数字前缀，光标会移动指定数据的单词，如 5w 表示将光标向后移动 5 个单词。

命令 b 可以将光标向前移动一个单词，也可以加上数字前缀表示移动多个单词。e 命令可以将光标移动到下一个单词的最后一个字符。而命令 be 可以将光标移动到前一个单词的最后一个字符。

(3) 移动行

Vim 的行移动命令如表 2-3 所示。

命令	操作说明
$	将光标移动到当前行的尾部，类似 End 键。可以配合数字前缀表示向后移动到若干行的行尾。例如 5$表示移动到第 5 行的行尾
0	可将光标移动到当前行的第一个字符上，相当于 Home 键，该命令不接受数字前缀
^	将光标移动到当前行的第一个非空白字符上。该命令前面加上数字没有任何效果。
:	：加具体行号，光标会移动到指定的行上，例如 :20。
j	使用 j 命令可以向下跳转若干行。加数字前缀，表示可以跳转出相应的行数。
G	G 命令把光标定位到指定的行上。加数字前缀，可以跳转到相应的行数，例如 10G 表示把光标定位到第 10 行。若未指定数字，则定位到最后一行。
gg	命令 gg 表示跳转到第 1 行，等同于 1G
%	在% 命令之前指定一个数字，可以将文件定位到这个指定百分比的位置上，如 95%会把光标定位到接近文件结尾的位置。

3. 删除

Vim 可以使用命令对光标处字符进行删除、也可以对单词或整行进行删除。删除命令参考表 2-4。

表 2-4 删除命令

命令	操作说明
x	删除光标所指向的当前字符
nx	删除光标所指向的前 n 个字符
:1, #d	删除行 1 至行#的文字
X	删除光标前面一个字符
D	删除至行尾
dw	删除光标右侧的字
ndw	删除光标右侧的 n 个字
db	删除光标左侧的字
ndb	删除光标左侧的 n 个字
dd	删除光标所在行
ndd	删除 n 行内容

4. 撤销与恢复

在编辑时如果由于错误操作而修改了原有的文本，可以使用撤销命令来取消之前的修改操作。Vim 可以多次取消以前的操作，常用命令参考表 2-5。

表 2-5 取消命令

命令	操作说明
.	重复上一次修改动作
u	取消上一次修改(撤销)
U	将当前行恢复至修改前的状态(恢复)

U 命令一次撤销对一行的全部操作，第二次使用 U 命令则会撤销前一个 U 操作的命令。连续使用 u 命令或“.”命令可以多次执行取消或重复上一次操作。

5. 复制与粘贴

Vim 可以释放复制、粘贴及移动操作，常见命令参考表 2-6。

表 2-6 删除命令

命令	操作说明
yw	复制光标处之字到字尾至缓冲区
yy	复制光标所在支行至缓冲区
#yy	如 5yy，复制光标所在之行以下 5 行至缓冲区
P	把缓冲区的资料粘贴在所在行之后
p	把缓冲区的资料粘贴在所在行之前

6. 移动

当使用"d"，"x" 或者其它命令删除文本的时候，这些文字会被存起来。因此可以用 p 命令重新粘贴出来。当删除的是一整行，"p" 命令把该行插入到光标下方。如果删除的是一行的一部分(例如一个单词)，"p" 命令会把它插入到光标的后面。

7. 查找

查找命令是 "/String"。例如，要查找单词 "include"，使用如下命令：

```
/include
```

你会注意到，当你输入 "/" 的时候，光标移到了 Vim 窗口的最后一行，这与 "冒号命令" 一样，在那里你可以输入要查找的字符串。

要查找下一个匹配可以使用 "n" 命令。用下面命令查找光标后的第一个 #include：

```
/#include
```

然后输入 "n" 数次。你会移动到后面每一个 #include。如果你知道你想要的是第几个，可以在这个命令前面增加次数前缀。这样，"3n" 表示移动到第三个匹配点。要注意，"/" 不支持次数前缀。

"?" 命令功能与 "/" 的功能类似，但是是反方向查找：

```
?word
```

"N" 命令在反方向重复前一次查找。因此，在 "/" 命令后执行 "N" 命令是向后查找，在 "?" 命令后执行 "N" 命令是向前查找。

"/>" 是一个特殊的记号，表示只匹配单词末尾。类似地，"/<" 只匹配单词的开头。这样，要匹配一个完整的单词 "the"，只需：

```
/<the>
```

2.1.3 Vim 的底行模式操作

Vim 的底行模式也叫末行模式，就是在界面最底部进行命令的输入，底行模式一般用来执行保存和退出等任务。只要在命令模式下输入冒号，就可以进入底行模式。

1. 显示和取消行号

表 2-7 显示和取消行号

命令	操作说明
: set nu	显示行号
: set nonu	不显示行号

2. 字符串搜素

表 2-8 字符串搜索命令

命令	操作说明
:/str	正向搜索，将光标移到下一个包含字符串 str 的行，按 n 可向下继续找
:?str	反向搜索，将光标移到上一个包含字符串 str 的行，按 n 可向上继续找
:/str/ w file	正向搜索，并将第一个包含字符串 str 的行写入 file 文件

3. 删除正文

表 2-9 删除正文

命令	操作说明
:d	删除光标所在行
:3 d	删除 3 行
:.,$ d	删除当前行至正文的末尾
:/str1/,/str2/ d	删除从字符串 str1 到 str2 的所有行

4. 退出命令

表 2-10 退出命令

命令	操作说明
:wq 或:x	先保存再退出 Vim
:w 或:w fname	保存/保存为 fname 名的文件
:q	退出(如果文件被修改会有提示)
:q!和:quit	不保存退出 Vim
:wq!	强制保存，并退出

5. 恢复命令

命令“:recover”可以恢复文件。

由于 Vi/Vim 是基于命令交互的方式，对于用惯了可视化工具的用户来说，在开始会有短暂的不适应，但相信经过慢慢实践，大家一定会喜欢上这款工具的。

2.2 GCC 编译器

在 Unix/Linux 环境下做开发，绝大多数数情况下使用的都是 C 语言，因此几乎每一个 Linux 程序员面临的首要问题都是如何灵活运用 C 语言编译器。目前 Linux 下最常用的 C 语言编译器是 GCC(GNU Compiler Collection)，它是 GNU 项目中符合 ANSI C 标准的编译系统，能够编译用 C、C++和 Object C 等语言编写的程序。GCC 不仅功能非常强大，结构也异常灵活。它还是一个交叉平台编译器，能够在当前的 CPU 平台上为多种不同的体系结构的硬件开发软件。因此 GCC 不仅适合于 Unix/Linux 环境下的应用级、系统级开发编译，还适合于嵌入式领域的开发编译。Linux 系统下的 GCC 是 GNU 推出的功能强大、性能优越的多平台编译器，是 GNU 的代表作品之一。

2.2.1. GCC 编译流程

编译过程隐藏着很多的细节，了解隐藏的细节可以帮助我们深入地理解产生的大量警告和错误信息，从而可以更加精确地编译程序和控制连接。

C 和 C++编译器是集成的。对于 GUN 编译器来说，程序的编译都要用四个步骤中的一个或多个处理输入文件: 预处理(preprocessing)，编译(compilation)，汇编(assembly)和连接(linking)。执行流程如图 8-4 所示。

(1) 预处理

在预处理阶段，输入的是 C 语言的源文件，通常为*.c。它们通常带有.h 之类头文件的包含文件。这个阶段主要处理源文件中的#ifdef、 #include 和#define 命令。

预处理常用的选项是“-E”，它的作用是告诉编译器，当预处理结束后停止编译。

预处理的输入是.c 文件，输出时.i 文件。

可以利用下面的示例命令：

gcc -E test.c -o test.i

但实际工作中通常不用专门生成这种文件，因为基本上用不到。

(2) 编译

编译阶段是对代码的规范性、语法的正确性进行检查，并编译成汇编语言。

编译阶段常用的选项是“-s”，它的作用是告诉编译器，当编译阶段结束时，停止编译。

编译的输入是中间文件*.i，编译后生成汇编语言文件*.s 。这个阶段对应的 GCC 命令如下所示：

gcc -s test.i -o test.s

(3) 汇编

在汇编阶段，编译器将输入的汇编文件*.s 转换成机器语言*.o。汇编常用的选项是“-c”，它的作用是告诉编译器，当汇编阶段结束时停止编译。

这个阶段对应的 GCC 命令如下所示：

gcc -c test.s -o test.o

(4) 链接

最后，在链接阶段将输入的机器代码文件*.s(与其它的机器代码文件和库文件)汇集成一个可执行的二进制代码文件。

链接并不需要选项，只是需要指定生成的可执行文件。可以利用下面的示例命令完成：

gcc test.o -o test

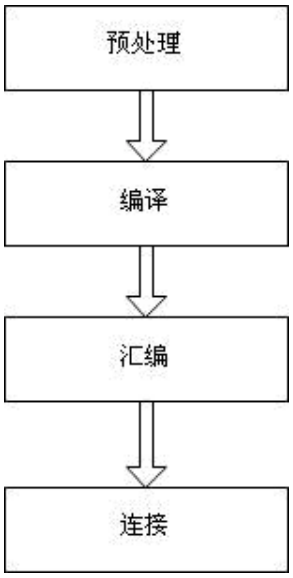

图 2-2 程序的编译流程

2.2.2 GCC 编译模式

（1） GCC 支持编译的文件

由于 GCC 支持多种语言，因此 GCC 能够支持多种文件后缀的编译。表 8-7 列出了 GCC 支持的源文件后缀名情况。

源文件后缀名标识源文件的语言，但是对编译器来说，后缀名控制着缺省设定：

gcc：认为预处理后的文件(.i)是 C 文件，并且设定 C 形式的连接。

g++：认为预处理后的文件(.i)是 C++文件，并且设定 C++形式的连接。

表 2-11 GCC 支持的常用的后缀名表

后缀名	意义	后期操作
.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
.o	目标文件	链接
.a	归档库文件	链接

(2) GCC 编译方式

GCC 指令的一般格式为：gcc [选项] 源文件名 [选项] 目标文件名

根据 GCC 采用命令的不同，可以讲 GCC 的编译模式分以下三种情况：默认模式、编译模式、编译连接模式。下面通过实例，以理论结合实例的方式，引领读者更好的理解。

a) GCC 编译模式

例 2-1 hello.c 程序

```
#include <stdio.h>
int main()
{
    printf("hello world!\n");
    return 0;
}
```

编译模式一：默认方式

采用默认方式编译源文件，系统会生成 a.out 可执行文件，调用 a.out 文件可查看执行结

```
[root@localhost programs]# gcc hello.c
[root@localhost programs]# ls
a.out  hello.c
[root@localhost programs]# ./a.out
hello world!
```

图 2-3 默认编译方式

果，如图 2-3 所示。

编译模式二：编译模式

```
[root@localhost programs]# gcc -o hello hello.c
[root@localhost programs]# ls
hello  hello.c
[root@localhost programs]# ./hello
hello world!
```

图 2-4 编译模式

在该模式下，采用“-o”选项，直接将源文件编译成目标文件，如图 2-4 所示。

编译模式三：编译链接模式

在该模式下，采用先编译成目标文件，再编译成可执行文件。首先利用“-c”选项，生成

```
[root@localhost programs]# gcc -c hello.c
[root@localhost programs]# ls
hello.c  hello.o
[root@localhost programs]# gcc -o hello hello.o
[root@localhost programs]# ls
hello  hello.c  hello.o
[root@localhost programs]# ./hello
hello world!
```

图 2-5 编译链接模式

目标文件 hello.o，再利用“-o”选项，生成可执行文件 hello。

b)多源文件编译方法

以上 GCC 编译针对一个源文件进行。实际中，一个程序的源代码通常包含在多个源文件之中。因此，对于多个源文件，如何用 GCC 处理呢？通常有以下两种方法。

方法一：多个文件一起编译

例 2-2 多文件一起编译

[root@localhost programs]# gcc -o test first.c second.c third.c

说明 该命令将同时编译三个源文件，即 first.c、second.c 和 third.c，然后将它们连接成一个可执行程序，名为 test。

方法二：分别编译各个源文件，之后对编译后输出的目标文件链接。

例 2-3 多文件编译

[root@localhost programs] #gcc -c first.c //将 first.c 编译成 first.o

[root@localhost programs] #gcc -c second.c //将 second.c 编译成 second.o

[root@localhost programs] #gcc -c third.c //将 third.c 编译成 third.o

[root@localhost programs] #gcc -o test first.o second.o third.o //将 first.o、second.o 和 third.o 链接成 test

说明

①第一种方法编译时需要所有文件重新编译，而第二种方法可以只重新编译修改的文件，未修改的文件不用重新编译。

②要生成可执行程序时，一个程序无论有有一个源文件还是多个源文件，所有被编译和连接的源文件中必须有且仅有一个 main 函数，因为 main 函数是该程序的入口点(换句话说，当系统调用该程序时，首先将控制权授予程序的 main 函数)。但如果仅仅是把源文件编译成目标文件的时候，因为不会进行链接，所以 main 函数不是必需的。

2.2.3 GCC 常用命令选项

(1) 总体选项

表 2-12　总体选项常用参数表

参数名	意义
-E	只进行预处理
-c	将源文件生成目标代码
-S	生成汇编代码
-o outfile	生成指定的输出文件，用在生成可执行文件时。
-g	在可执行程序中，添加调试代码
-v	打印出编译器编译过程中的信息
-static	静态链接库
-library	连接名为 library 的库文件

例 2-4 library 选项的应用：使用 pthread 库编译程序

[root@localhost programs]gcc –o test test.c –lpthread

gcc test.c –lpthread

(2) 目录选项

目录选项用于指定搜索路径，用于查找头文件、库文件或编译器的某些成员。

表 2-13　GCC 的目录选项参数

参数名	意义
-I dir	在头文件的搜索路径列表中添加 dir 目录。
-L dir	制定编译时，搜索库的路径
-B prefix	指出在何处寻找可执行文件、库文件及编译器自己的数据文件。

例 2-5 头文件和源文件会单独存放在不同的目录中

假设存放源文件的子目录名为./src，而包含文件则放在层次的其他目录下，如./inc。当我们在./src 目录下进行编译工作时，如何告诉 GCC 到哪里找头文件呢？方法如下所示：

[root@localhost programs] #gcc test.c –I../inc -o test

说明 命令告诉 GCC 包含文件存放在./inc 目录下，在当前目录的上一级。

如果在编译时需要的包含文件存放在多个目录下，可以使用多个-I 来指定各个目录：

[root@localhost programs] # gcc test.c –I../inc –I../../inc2 -o test

说明 命令告诉 GCC 包含文件存放在./inc 和./inc2 两个文件夹中，其中 inc 文件夹在在当前目录的上一级，另一个包含子目录 inc2，较之前目录它还要在再上两级才能找到。

(3) 警告选项

GCC 在编译过程中检查出错误的话，它就会中止编译；但检测到警告时却能继续编译生成可执行程序，因为警告只是针对程序结构的诊断信息，它不能说明程序一定有错误，而是存在风险，或者可能存在错误。虽然 GCC 提供了非常丰富的警告，但前提是你已经启用了它们，否则它不会报告这些检测到的警告。

表 2-14 GCC 的警告选项参数

参数名	意义
-w	禁止所有警告信息
-wall	允许发出所有有用的警告信息
-werror	把所有的警告化成错误信息，并在出现任何警告时放弃编译

在众多的警告选项之中，最常用的就是-Wall 选项。该选项能发现程序中一系列的常见错误警告。例如设置警告调试选项的命令如下：

[root@localhost programs]#gcc –wall test.c –o test

(4) 调试选项

调试选项是用于用户程序的调试。

表 2-15 GCC 的调试选项

参数名	意义
-g	产生调试信息，GDB 能使用这些调试信息
-ggdb	以本地格式（如果支持）输出调试信息，尽可能包括 GDB 扩展
-gstabs	以 stabs 格式（如果支持）输出调试信息，不包括 GDB 扩展。这是大多数 BSD 系统上 DBX 使用的格式
-gstabls+	以 stabs 方式输出调试信息，不过只有 GDB 能读取这些调试信息
-gcoff	以 coff 格式输出调试信息
-gxcoff	以 xcoff 格式输出调试信息

(5)优化选项

优化选项用于 GCC 对可执行代码进行优化操作，通常会用到的是 O、O0、O1、O2、O3 五种优化方式。通常 O 后面的数字代表优化的效果，数字越大，优化的内容就越多。

表 2-16 GCC 的优化选项参数

参数名	意义
-O	主要进行线程跳转与延时退栈两种优化，另外此时的优化能够有效减少目标代码空间以及程序的执行时间。
-O0	这个参数是指不进行优化。
-O1	对于大函数，优化编译占用稍微多的时间和相当大的内存。
-O2	除了涉及空间和速度交换的优化选项，执行几乎所有的优化工作，例如不进行循环展开和函数内嵌，和-O 选项比较，这个选项既增加了编译时间，也提高了生成代码的运行效果。
-O3	除了打开 O2 的所有优化外，还加强了程序对于处理器特性的优化，同时也进

	行循环展开的优化。

优化选项的添加能够提升程序的执行效率，所以一般在程序发布时，添加优化选项进行编译，但这里建议读者在程序调试时，尽量不要用优化选项，因为优化会在一定程度上改变程序的执行流程，会对调试工作产生一些不利影响。

2.3 Make 工程管理器

在编写小型的 Linux 应用程序时，一般情况下会只有少数几个源文件。这样程序员能够很容易地理清它们之间的包含和引用关系。但随着软件项目逐渐变大，对源文件的处理也将变得越来越复杂。此时单纯依赖于手工方式进行管理的做法就显得有些力不从心。为此，Linux 系统专门为软件开发提供了一个自动化管理工具 GNU Make。

在 Unix/Linux 环境下 GNU 的 Make 工程管理器是用于自动编译、链接程序的实用工具。包含多个源文件的项目在编译时都有长而复杂的命令行，使用 Make 工程管理器可以通过把这些命令行保存在 Makefile 文件中而简化这个工作，同时使用 Make 工程管理器还可以减少重新编译所需要的时间，因为 Make 工程管理器可以识别出那些修改的文件，而只编译这些文件。

Make 工程管理器是一个命令集，它解释 Makefile 中的指令(Makefile 中称为规则)。Makefile 文件中则描述了整个工程所有源文件的编译顺序、编译规则以及编译方法。

2.3.1 认识 Makefile

一个简单的 makefile 文件，示例来源于 GNU 的 make 使用手册。此例子由 3 个头文件和 8 个 C 文件组成，我们将书写一个简单的 Makefile，来描述如何创建最终的可执行文件“edit”，此可执行文件依赖于 8 个 C 源文件和 3 个头文件。

Makefile 文件的内容如下：

```
edit : main.o kbd.o command.o display.o \
        insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
            insert.o search.o files.o utils.o
main.o : main.c defs.h
            cc -c main.c
kbd.o : kbd.c defs.h command.h
            cc -c kbd.c
command.o : command.c defs.h command.h
            cc -c command.c
display.o : display.c defs.h buffer.h
            cc -c display.c
insert.o : insert.c defs.h buffer.h
            cc -c insert.c
search.o : search.c defs.h buffer.h
            cc -c search.c
files.o : files.c defs.h buffer.h command.h
            cc -c files.c
utils.o : utils.c defs.h
            cc -c utils.c
clean :
```

*rm edit main.o kbd.o command.o display.o *
insert.o search.o files.o utils.o

2.3.2 Makefile 文件的构成

一个完整的 Makefile 里主要由显式规则、隐晦规则、变量定义、指示符和注释构成。

1. 显式规则

显式规则说明了，如何生成一个或多的的目标文件。这是由书写 Makefile 规则时需要明确指出目标文件、目标的依赖文件列表以及更新目标文件所需要的命令(有些规则没有命令，这些规则只是纯粹地描述文件之间的依赖关系)。显示规则就是 Makefile 的书写者明确给出的所有规则。显式规则的一般形式如下：

```
target : dependency_files
command
...
...
```

target：规则目标，通常就是一个目标文件。通常是最后需要生成的文件名或者为了实现这个目的而必需的中间过程文件名。可以是.o 文件、也可以是可执行程序的文件名等。另外，目标也可以是一个 make 执行的动作的名称，如目标“clean”。

dependency：规则的依赖。生成规则目标 target 所需要的文件名列表。

command：规则的命令行。是规则所要执行的动作(任意的 shell 命令或者是可在 shell 下执行的程序)。它限定了 make 执行这条规则时所需要的动作。

例如：main.o : main.c defs.h

```
        cc -c main.c
```

2. 隐晦规则

隐晦规则是指 Make 工程管理器根据某一类文件的特征(如根据文件扩展名后缀)按照特定的对应关系，为创建(或者更新)这个文件而自动推导出来的规则。由于 Make 工程管理器有自动推导功能，可以根据目标文件的文件名，自动产生目标的依赖文件和生成目标的命令。

例如：foo: foo.o bar.o

```
  cc –o foo   foo.o bar.o $(CFLAGS)   $(LDFLAGS)
```

因为此处提及了文件 foo.o，但是却没有给出它的规则，make 将自动寻找一条隐含规则，该规则告诉 make 怎样更新该文件。无论 foo.o 文件存在与否，make 都会这样执行。Make 会找到如下规则生成 foo.o：

$(CC) –c $(CPPFLAGS) $(CFLAGS)

表 2-17 给出了常见的隐式规则[2]。

表 2-17 Makefile 中的常见隐式规则

对应语言后缀名	规则
C 编译：.c 变为.o	$(CC) –c $(CPPFLAGS) $(CFLAGS)
C++编译：.cc 或.c 变为.o	$(CXX) –c $(CPPFLAGS) $(CXXFLAGS)
Pascal 编译：.p 变为.o	$(PC) –c $(PFLAGS)
Fortran 编译：.r 变为.o	$(FC) –c $(FFLAGS)

表 2-18 编译器额外的表示变量

[2] 隐式规则只能找到相同文件名的不同后缀文件，如foo.o文件必须由foo.c文件生成

选项	说明
ASFLAGS	用于汇编编译器的额外标识
CFLAGS	用于 C 编译器的额外标识
CXXFLAGS	用于 C++编译器的额外标识
CPPFLAGS	用于 C 预处理以及使用它的程序的额外标识
LDFLAGS	用于调用 linker(ld)的编译器的额外标识

3. 变量的定义

在 Makefile 中需要定义一系列的变量，变量一般都是字符串，这个有点你 C 语言中的宏，当 Makefile 被执行时，其中的变量都会被扩展到相应的引用位置上。

例如：CC = gcc

4. 文件指示符

包括三个部分，第一部分是在一个 Makefile 中引用另一个 Makefile，就像 C 语言中的 include 一样包含进来；第二部分是指根据某些情况指定 Makefile 中的有效部分，就像 C 语言中的预编译宏一样；第三部分是定义一个多行的命令。

5. 注释

Makefile 中只有行注释，和 UNIX 的 Shell 脚本一样，其注释是用“#”字符，这个就像 C/C++中的“//”一样。如果你要在你的 Makefile 中使用“#”字符，可以用反斜框进行转义，如：“\#”。

注意，在 Makefile 中的命令必须要以 Tab 键开始。

2.3.3 Makefile 变量

在 Makefile 中，变量类似于环境变量，大小写敏感，一般使用大写字母。变量是一个名字(像 C 语言中的宏)，代表一个文本字符串(变量的值)。Makefile 中定义变量的一般形式如下；

(1) 变量名、赋值符、变量值

变量名习惯上只使用字母、数字和下划线，并且不以数字开头，不可以包括“：”、“#”、“=”、前置空白和尾空白的任何字符串。赋值符主要有 4 个，即“=”、“：=”、“+=”、“？=”。变量值是一个文本字符串。另外有一些变量名只包含了一个或很少几个特殊的字符(符号)，如“$<”等，它们称为**自动变量**。

表 2-19 Makefile 中常见的自动变量

自动变量	含义
$?	当前目标所依赖的文件列表中比目前目标文件还要新的文件
$@	当前目标文件的名字
$<	第一个依赖文件的名字
$*	不包含扩展名的当前依赖文件的名字
$+	所有的依赖文件，以空格分开，并以出现的先后为序，可能包含重复的依赖文件
$^	所有不重复的依赖文件，以空格分开

(2) 引用变量

定义一个变量之后，就可以在 Makefile 的很多地方使用这个变量。变量的引用方式是“$(变量名)”或者“${变量名}”。例如，$(fun)或者${fun}就是取变量 fun 的值。对一个变量的引用可以在 Makefile 的任何上下文、目标、依赖、命令、绝大多数指示符和新变量的赋值中。

Makefile 中的变量分为用户自定义变量、预定义变量、自动变量及环境变量。自定义变量的值由用户自行设定，而预定义变量和自动变量为通常在 Makefile 中都会出现的变量，其中部分由默认值，也就是常见的设定值，当然用户也可以对其进行修改。

预定义变量包含了常见编译器、汇编器的名称及其编译选项，如表 2-19 列出了 Makefile 中常见的预定义变量。

表 2-20 Makefile 中常见的预定义变量

命令格式	含义
$*	不包含扩展名的目标文件名称
$+	所有的依赖文件以空格分开，以出现的先后为序，包含重复的依赖文件
$<	第一个依赖文件的名称
AR	库管理命令，默认值为 ar
AS	汇编程序的名称，默认值为 as
CC	C 编译器的名称，默认值为 cc
CPP	C 预编译器的名称，默认值为$(CC)-E
CXX	C++编译器的名称，默认值为 g++
FC	FORTRAN 编译器的名称，默认值为 f77
PM	文件删除程序的名称，默认值为 rm －f
ARFLAGS	库文件维护程序的选项，无默认值
ASFLAGS	汇编程序的选项，无默认值
CFLAGS	C 编译器的选项，无默认值
CPPFLAGS	C 预编译的选项，无默认值
CXXFLAGS	C++编译器的选项，无默认值
FFLAGS	FORTRAN 编译器的选项，无默认值
HOMEDIR	当前项目的主目录
INCDIRS	搜索头文件的目录
LIBDIRS	搜索库文件的目录

例 2-6 一个 Makefile 文件

```
OBJS = foo.o bar.o
INCDIRS = -I /usr/include
CC = gcc
CFLAGS = -Wall –O –g
myprog : $(OBJS)
    $(CC) $(OBJS) –o myprog
foo.o : foo.c foo.h bar.h
    $(CC) $(CFLAGS) –c $(INCDIRS)  foo.c –o foo.o
bar.o : bar.c  bar.h
    $(CC) $(CFLAGS)  -c $(INCDIRS) $<  -o  $@
clean:
    rm  –f  *.o  core  *  ~
realclean: clean
   rm –f  myprog
```

说明：

(1) 开始 3 行定义变量：OBJS、INCDIRS、CC、CFLAGS。

(2) 这里有 5 条规则，第 1 条一般作为默认规则，即如果在 shell 命令行中只输入 make 的话就从这里开始。

(3) 第 1 条规则时编译 foo.o 和 bar.o，编译完后再执行 gcc foo.o bar.o –o myprog，从而生成 myprog 文件。

(4) 第 2 条规则时如何编译 foo.o，它执行命令：gcc –Wall –O –g –c –I /usr/include foo.c –o foo.o。

(5) 第 3 条规则比第 2 条规则复杂一点，$<是指 bar.c，$@是指 bar.o，所以命令是：

gcc –Wall –o –g –c –I /usr/include bar.c bar.o

(6) 第 4 条规则定义一个没有依赖关系的名为 clean 的目标。当一个目标没有依赖关系时，无论何时被调用，它的命令都被执行。Clean 删除目标文件(*.o)、核心文件(core)以及备份文件(*~)。

(7) 第 5 条规则定义称为 realclean 的目标，它用第 4 条规则作为它的一个依赖关系，使 make 创建 clean 目标，删除 myprog 二进制代码。

2.3.4 Make 工程管理器的使用

使用 Make 工程管理器非常简单，只需要在 make 命令的后面输入目标名即可建立指定的目标，如果直接运行 make，则建立 Makefile 中的第一个目标。

此外，Make 工程管理器还有丰富的命令行选项，可以完成各种不同的功能。表 2-21 列出了常用的 Make 工程管理器的命令行选项。

表 2-21 Makefile 中常见的预定义变量

命令格式	含义
-C dir	读入指定目录下的 Makefile
-f file	读入当前目录下的 file 文件作为 Makefile
-i	忽略所有的命令执行错误
-I dir	指定被包含的 Makefile 所在目录
-n	只打印要执行的命令，但不执行这些命令
-p	显示 Make 变量数据库的隐式规则
-s	在执行命令时不显示命令
-w	如果 Make 在执行过程中改变目录，打印当前目录名

2.3.5 多文件编译实例

1. 程序源代码

程序由 5 个源文件组成，文件为：include/func1.h，include/func2.h；src/func1.c，src/func2.c，src/main.c。下面列出其源代码。

(1) func1.h

```
#ifndef __FUNC1_H__
#define __FUNC1_H__
void func1print();
#endif
```

(2) func2.h

```
#ifndef __FUNC2_H__
#define __FUNC2_H__
void func2print();
#endif
```

(3) func1.c

```
#include <stdio.h>
#include "func1.h"

void func1print()
{
    printf("This is func1 print!\n");
}
```

(4) func2.c

```
#include <stdio.h>
#include "func1.h"
#include "func2.h"
void    func2print()
{
    printf("In func2, first, call func2:");
    func1print();

    printf("This is func2 print!\n");
}
```

(5) main.c

```
#include "func1.h"
#include "func2.h"

int main(void)
{
    func1print();
    func2print();
    return 0;
}
```

从上面的源代码可以很清晰地看到，这是一个比较简单的例子，func1.h 和 func2.h 中定义了两个功能函数，分别是 func1print()和 func2print()；其中 func2 中的功能函数 func2print()调用 func1.h 中定义的 func1print()函数。

Main()函数定义在 main.c 文件中，main()函数分别调用 func1.h 和 func2.h 中的两个功能函数。

2. Makefile 的编写流程

编写任何一个工程的 Makefile 脚本，都要按照相应的步骤。下面列出编写一个 Makefile 脚本的常用步骤。

第 1 步：编写可执行文件生成规则。

```
myapp : main.o func1.o func2.o
    gcc -o myapp main.o func1.o func2.o
```

第 2 步：编写各目标文件生成规则。

(1)main.o 文件：它由 main.c 文件生成，而 main.c 文件又包含了头文件 func1.h 和 func2.h，因此它的生成规则为：

```
main.o : src/main.c include/func1.h include/func2.h
```

gcc -c -I ./include src/main.c

(2)func1.o 文件：它由同名的 C 文件生成，可以利用以下规则：

func1.o : src/func1.c include/func1.h

gcc -c -I ./include src/func1.c

(3) func2.o 文件：它是由 func2.c 生成，但是在 func2 中调用了 func1.c 文件中的函数，包含了 func1.h，即它还依赖于 func1.h，因此它的生成规则为：

func2.o : src/func2.c include/func2.h include/func1.h

gcc -c -I ./include src/func2.c

第 3 步：编写伪目标规则。一般要在后面定义一个伪目标：clean。它的生成规则为：

clean:

$(RM) *.o myapp

将上述规则合并在一起，便得到所需要的 makefile 文件。

3.Make 执行

针对编写的 Makefile 脚本，进行测试，其中测试命令如下：

$make

gcc -c -I ./include src/main.c

gcc -c -I ./include src/func1.c

gcc -c -I ./include src/func2.c

gcc -o myapp main.o func1.o func2.o

$./myapp

This is func1 print!

In func2, first, call func2:This is func1 print!

This is func2 print!

输入 make 命令，若有错误，可以根据错误的提示，进行相应的修改。这里可以看到，执行 make 命令后，生成目标文件与可执行文件，证明 Makefile 文件编写正确。

GNU Make 包括一个很不错的、容易阅读的文本信息格式的用户手册，可以通过学习 Make 手册了解 Make 工程管理更强大的功能。

2.4 GDB 调试器

调试是软件开发过程中一个必不可少的组成部分。如何找到软件中存在的问题及症结所在，是每一个开发人员都要面对的问题。通常说来，软件项目的规模越大，调试起来就越困难，越需要一个强大而高效的调试器作为后盾。对于 Linux 程序员来讲，目前可供使用的调试器非常多，GDB(GNU Debuger)就是其中较为优秀的一个。虽然这种工具使用起来不如一些可视化工具随心所欲，但它的一些高效特性为程序员调试程序指明了方向，也可以帮助软件工程师提高工作效率，进而增进工程项目的进度。本节主要介绍如何使用 GDB 调试。

2.4.1 GDB 概述

GDB(GNU Symbolic Debugger)简单地说就是一个调试工具。它是一个受通用公共许可证即 GPL 保护的自由软件。

像所有的调试器一样，GDB 可以让你调试一个程序，包括让程序在预设的地方停下，此时可以查看变量，寄存器，内存及堆栈。更进一步可以修改变量及内存值。GDB 是一个功能很强大的调试器，它可以调试多种语言。在此我们仅涉及 C 和 C++ 的调试，而不包括其它语言。

GDB 调试器，不象 VC 一样是一个集成环境。你可以使用一些前端工具如 XXGDB，DDD 等，它们都有图形化界面，因此使用更方便，但它们仅是 GDB 的一层外壳。因此，熟悉 GDB 的使用仍是必要的。下面介绍 GDB 的一些重要的常用命令。

在开始调试程序之前，需要注意增加编译选项***-g*** 或其它相应的选项，将调试信息加到要调试的程序中。例如：gcc -g -o hello hello.c

GDB 主要能做 4 件事情(包括为了完成某些事而附加的功能)，以帮助开发人员找出被调试程序中的错误。

其一，运行被调试程序，设置所有的能影响该程序的参数和变量。

其二，保证被调试程序在指定的条件下停止运行。

其三，当被调试程序停止时，让开发工程师检查发生了什么。

其四，根据每次调试器的提示信息来做相应的改变，那样可以修正某个错误引起的问题，然后继续查找其他的错误。

以上 4 个过程是周而复始的过程，在实际调试过程中，发挥着重要作用。

2.4.2 GDB 的启用和退出

GDB 有两种加载被调试程序的方式：一种是在启动时输入 gdb filename，其中 filename 是可执行文件名，可以自动加载被调试文件；另一种是在启动 GDB 后输入 file filename，其中 file 是 GDB 内置命令，用于指明要加载的调试文件。

下面以 hello.c 文件的编译和调试为例，讲解 GDB 如何加载调试文件。

1. 启动

切换终端目录到 hello.c 文件所在目录，输入编译命令：gcc–g –o hello　hello.c；

(1)启动方式一

输入调试命令 gdb hello，然后输入 run，查看运行结果如图 2-6 所示。

```
[root@localhost programs]# gcc -g -o hello hello.c
[root@localhost programs]# gdb hello
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or 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.
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb) run
Starting program: /home/user/programs/hello
hello world!

Program exited normally.
(gdb)
```

图 2-6 GDB 加载调试文件(一)

(2)启动方式二

输入调试命令 gdb 启动 GDB 调试器后，输入 file hello 命令，然后输入 run，查看运行结果如图 2-7 所示。

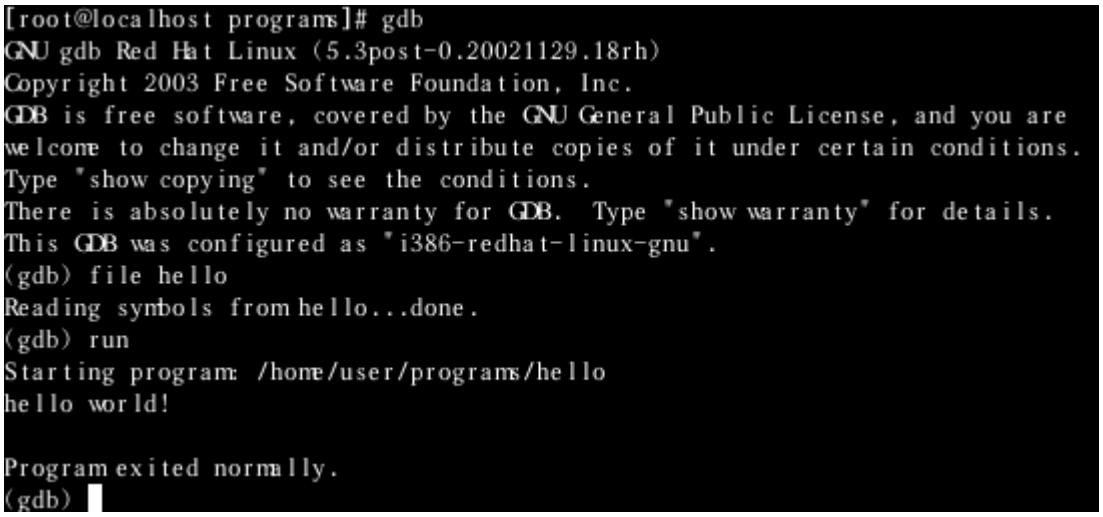

图 2-7 GDB 加载调试程序(二)

2. GDB 退出

退出 GDB 的命令是 quit。若只终止被调试的可执行程序，可输入命令 kill，kill 的意义是终止当前正在调试的程序。

2.4.3 GDB 调试实例

例 2-7 通过该实例练习以下 GDB 调试命令：break(设置断点)、next(单步调试)、where(错误定位)、where(定位)。

该例题的源代码（有错误的程序）：

```
#include <stdio.h>
#include <stdlib.h>
static char buff[256];
static char *string;
int main()
{
    printf("Please input a string:");
    gets(string);
    printf("\n Your string is :%s\n",string);
}
```

程序调试过程如下：

(1) 编译源文件并执行

```
[root@localhost programs]# gcc -g -o bug eg_file_bug1.c
/tmp/cc8YH5A2.o(.text+0x2a): In function `main':
/home/user/programs/eg_file_bug1.c:8: the `gets' function is dangerous and should not be used.
[root@localhost programs]# ./bug
Please input a string:hello
段错误
```

说明

为了在目标文件中产生调试信息，需要在进行编译时，在 gcc(或 g++)下使用额外的-g 选项。

(2) 分析原因：这个程序目的是接收用户的键盘输入，并将结果输出。编译提示警告错误提示：gets 函数存在问题。

(3) 调试过程

Step 1：运行 gdb bug 命令，装入 bug 可执行程序

```
[root@localhost programs]# gdb bug
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or 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.
This GDB was configured as "i386-redhat-linux-gnu"...
```

Step 2：使用 list（缩写为 l）命令，查看代码

```
(gdb) l
1       #include <stdio.h>
2       #include <stdlib.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("\n Your string is :%s\n",string);
10      }
```

Step 3：设置断点，位置 gets 函数

```
(gdb) break gets
Breakpoint 1 at 0x804827c
```

Step 4：通过 run 命令运行程序，并且程序在 Breakpoint 1 处暂停

```
(gdb) run
Starting program: /home/user/programs/bug
Breakpoint 1 at 0x42062f86

Breakpoint 1, 0x42062f86 in gets () from /lib/tls/libc.so.6
```

Step 5：使用 next 命令继续程序的执行，此时程序提示 ges 函数需要参数，如何字符串“hello”后，程序执行，并接受到 SIGSEGV 信号，即出现段错误

```
(gdb) next
Single stepping until exit from function gets,
which has no line number information.
Please input a string:hello

Program received signal SIGSEGV, Segmentation fault.
0x42063050 in gets () from /lib/tls/libc.so.6
```

Step 6：使用 where 命令查看程序出错的位置，可以看出当前程序在 0#栈出错，此时处于 gets 函数中

(gdb) where

#0　0x42063050 in gets () from /lib/tls/libc.so.6

#1　0x0804838a in main () at eg_file_bug1.c:8

#2　0x42015574 in __libc_start_main () from /lib/tls/libc.so.6

通过以上信息可以明确，gets 函数出错。从代码中可以看出，唯一能够导致 gets 函数出错的原因就是变量 string。因此使用 print（缩写 p）命令查看 string 变量

Step 7：打印字符串 string 的值，进一步确认

(gdb) p string

$1 = 0x0

在 gdb 中，我们可以直接修改变量的值，只要将 string 取一个合法的指针值就可以了，为此，我们在第 8 行处设置断点 break 8；

程序重新运行到第 8 行处停止，这时，我们可以用 set variable 命令修改 string 的取值； 然后继续运行，将看到正确的程序运行结果。

Step 8：使用 quit（缩写 q）命令退出 GDB 调试器。

(gdb) q

到此为止，可以确定出错是由于 string 变量未赋初值的缘故。

2.4.4 停下来环顾程序

1.断点概述

有 3 种方式可以通知 GDB 暂停程序的执行。

(1) 断点：通知 GDB 在程序中的特定位置暂停执行。

(2) 监视点：通知 GDB 当特定内存位置(或者涉及一个或多个位置的表达式)的位置发生变化时暂停执行。

(3) 捕获点：通知 GDB 当特定事件发生时暂停执行。

在 GDB 文档中将这 3 种机制都称为断点。例如在帮助文档中可以看到，GDB 关于删除断点的命令 delete 命令，其含义是用 delete 命令删除所有断点、监视点和捕获点。

(gdb) help delete

Delete some breakpoints or auto-display expressions.

Arguments are breakpoint numbers with spaces in between.

To delete all breakpoints, give no argument.

在程序中的特定“位置”设置断点，当到达那一点时，调试器会暂停程序的执行。GDB 中关于“位置”的含义非常灵活，它可以指各种源代码行、代码地址、源代码文件中的行号或者函数入口等。

2.设置断点

使用 GDB 时，需要指示在何处暂停执行，以便使用 GDB 的命令行提示符执行调试活动。GDB 有许多指定断点的方式，下面介绍最常见的几种。

命令格式：break 断点

断点可以通过函数名、当前文件内的行号来设置，也可以先指定文件名再指定行号，还可以指定与暂时位置的偏移量，或者用地址来设置。

命令格式如下：

break 函数名

例：(gdb) break main

break 行号

break 文件名:行号
break 文件名:函数名
break +偏移量
break －偏移量
break *地址

例 2-8 设置断点

(gdb) break main
Breakpoint 1 at 0x804860c: file process_append.c, line 10.
(gdb) b 19
Breakpoint 2 at 0x804868e: file process_append.c, line 19.
(gdb) break process_append.c:14
Breakpoint 3 at 0x8048637: file process_append.c, line 14.
(gdb) b +3
Breakpoint 4 at 0x80486db: file process_append.c, line 24.
(gdb) b *0x8048637
Note: breakpoint 3 also set at pc 0x8048637.
Breakpoint 5 at 0x8048637: file process_append.c, line 14. (gdb) break main
可以使用 **info break** 命令查看断点信息。
(gdb) info break

Num	Type	Disp	Enb	Address	What
1	breakpoint	keep	y	0x0804860c	in main at process_append.c:10
2	breakpoint	keep	y	0x0804868e	in main at process_append.c:19
3	breakpoint	keep	y	0x08048637	in main at process_append.c:14
4	breakpoint	keep	y	0x080486db	in main at process_append.c:24
5	breakpoint	keep	y	0x08048637	in main at process_append.c:14

说明

使用 break +偏移量或 break –偏移量，可以在当前选中栈帧中正在执行的源代码之前后之后 offset 行设置断点。

使用 break *地址这种形式可用来在虚拟内存地址处设置断点。这对于程序没有调试信息的部分(比如找不到源代码时或对于共享库)是必须的。

3. 删除和禁止断点

在调试会话期间，会遇到大量断点，对于经常重复的循环结构或函数，这种情况使得调试极不方便。在这种情况下，可以使用断点禁用或启用命令。如果要保留断点以便以后使用，同时又不希望 GDB 停止执行，可以禁用它们在以后需要时再启用。

只有当 GDB 遇到启用的断点时，才会暂停程序的执行；它会忽略禁用的断点。默认情况下，断点的声明期从启用时开始。

禁用/启用断点的命令格式如下：

禁用断点：disable <断点号列表>

启用断点：enable <断点号列表>

对于确定不再使用的断点(因为修复了那个特定的程序错误)，可以删除该断点。删除断点的命令有 delete 和 clear。具体使用方法如下：

(1) delete <断点号列表>

删除断点号列表中的断点，断点使用数字标识符，例如 delete 2 4。

(2) delete

删除所有断点。

(3) clear

清除 GDB 将执行的下一指令处的断点。这种方法适用于要删除 GDB 已经到达的断点的情况。

(4) clear 根据位置清除断点

命令格式：clear 函数名、clear 文件名：函数名、clear 行号、clear 文件名：行号

例 2-9 删除断点

```
(gdb) clear 10
Deleted breakpoint 1
(gdb) info break
Num Type           Disp Enb Address    What
2   breakpoint     keep y   0x0804868e in main at process_append.c:19
3   breakpoint     keep y   0x08048637 in main at process_append.c:14
4   breakpoint     keep y   0x080486db in main at process_append.c:24
5   breakpoint     keep y   0x08048637 in main at process_append.c:14
(gdb) delete 2
(gdb) info break
Num Type           Disp Enb Address    What
3   breakpoint     keep y   0x08048637 in main at process_append.c:14
4   breakpoint     keep y   0x080486db in main at process_append.c:24
5   breakpoint     keep y   0x08048637 in main at process_append.c:14
```

4. 恢复执行

在断点处恢复执行与设置断点同样重要，GDB 提供了 3 类恢复执行的方法。第一类是使用 step 和 next“单步”调试程序，仅执行代码的下一行然后再次暂停。第二类由使用 continue 组成，使 GDB 无条件地恢复程序的执行，直到它遇到另一个断点或者程序结束。第三类是用 finish 或 unitl 命令恢复，在这种情况下，GDB 会恢复执行，程序继续运行直到遇到某个预先确定的条件(比如，到达函数的末尾)，或者到达另一个断点，或者程序完成。

(1) 使用 step 和 next 单步调试

使用单步跟踪的典型的技术是在函数的入口点设置一个断点，或者在可能发生错误的程序段上设置，运行程序直到它在断点上中断，再在这个嫌疑区域单步执行，检测感兴趣的变量，直到你发现错误发生。一旦 GDB 在断点处停止，可以使用 step(缩写为 s)和 next(缩写为 n)命令来单步调试代码。

其中，step 命令遇到函数会在函数的第一条语句处暂停，因此称为**单步进入(Stepping Into)**；而 next 命令遇到函数，会执行函数，而不会在其中暂停，因此称为**单步越过(Stepping Over)**。

可以在 next 或 step 命令后采用一个可选数值参数，表示使用该命令执行的额外行。命令格式如下：

①step [count]

继续单步执行，不过执行 count 次。如果执行到一个断点，或者一个与单步跟踪无关的信号在 count 次单步执行发生时，单步执行会立即中断。

②next [count]

在当前栈帧(最内层)上继续执行到 count 行源码行。

(2) 使用 continue 继续程序执行

使用 continue(缩写为 c)命令可以恢复继续程序的执行，直至触发断点或者程序结束。

常用命令格式：continue [ignore-count]

该命令表示在程序最近中断的地方重新执行；ignore-count 指定在这个位置上忽略断点次数。注意该参数只有在程序由于断点中断时才有意义。其他时候，continue 的 ignore-count 参数会被忽略。

例如：continue 3，表示该命令接受整数参数 3，使 GDB 恢复执行，并忽略接下来的 3 个断点。

(3) 使用 finish 恢复程序执行

使用 finish 命令可以**使 GDB 恢复执行，直到恰好在当前栈帧上的函数完成为止。**即当处于某函数中，使用 finish 命令，可以跳过该函数其余部分，返回到这个函数之前 GDB 所在的调用函数。

注意，在递归函数中，finish 只会将调用带到递归的上一层。

(4) 使用 until 恢复程序执行

命令 until(缩写为 u)通常**用来在不进一步在循环中暂停(除了循环的中间设置断点)的情况下完成正在执行的循环。**使用 until 执行循环的其余部分，让 GDB 在循环后面的第一行代码处暂停。这个命令用来避免多次单步执行一个循环。

不带参数的 until 命令将使用单个指令的单步执行方式，可以使用带参数的 until 命令来提高效率。

命令格式为：until location

该命令表示继续执行直到程序执行到指定的位置，或者从当前栈帧返回。location 可以是行号、函数地址、偏移量等各种形式。这个命令形式使用临时断点，因此比不带参数的 until 命令要快。注意：只有当这个指定的位置在当前帧上时，它才会真的被执行到。这就意味着 until 可以用来跳过函数嵌套调用。

例 2-10 使用单步调试技术

源程序代码:

```
#include <stdio.h>
void swap(int a,int b);
int main(void)
{
        int i = 4;
        int j = 6;
        printf("i: %d, j: %d\n",i,j);
        swap(i,j);
        printf("i: %d, j: %d\n",i, j);
        return 0;
}

void swap(int a, int b)
{
        int c = a;
        a = b;
        b = c;
}
```

编译并启动调试:

$ gcc -g swapper.c
$ gdb a.out
(gdb) start
Temporary breakpoint 1 at 0x80483ed: file swapper.c, line 5.
Starting program: /home/nachos/system/debug/a.out

Temporary breakpoint 1, main () at swapper.c:5
5 int i = 4;
(gdb) n
6 int j = 6;
(gdb) step
7 printf("i: %d, j: %d\n",i,j);
(gdb) step
i: 4, j: 6
8 swap(i,j);
(gdb) step
swap (a=4, b=6) at swapper.c:15
15 int c = a;
(gdb) finish
Run till exit from #0 swap (a=4, b=6) at swapper.c:15
main () at swapper.c:9
9 printf("i: %d, j: %d\n",i, j);
(gdb) n
i: 4, j: 6
10 return 0;
(gdb)
例 2-12 单步调试技术练习
源程序代码：

```
#include <stdio.h>
int main(void)
{
        int i;
        for(i = 0; i < 10; i++)
                printf("hello wrold");
        return 0;
}
```

编译并启动调试:
gcc -g until_anomaly.c
$ gdb a.out
(gdb) break main
Breakpoint 1 at 0x80483ed: file until_anomaly.c, line 5.
(gdb) run
Starting program: /home/nachos/system/debug/a.out

Breakpoint 1, main () at until_anomaly.c:5

5 for(i = 0; i < 10; i++)

(gdb) n

6 printf("hello wrold");

(gdb) until

5 for(i = 0; i < 10; i++)

(gdb) until

7 return 0;

(gdb)

说明

如果在循环的末尾执行 until 命令导致回跳到循环顶部，这时只要再次执行 until 就可以离开当前循环体。

5. 条件断点

当怀疑某个变量在某处得到了一个伪值时，可以通过设置条件断点，告诉编译器只有在有问题的代码处当变量呈现该怀疑值时才在断点处暂停。

设置条件断点的命令格式如下：

break break-args if (condition)

其中 break-args 表示 break 命令指定的断点位置的任何参数；condition 可以是任意布尔表达式。

若是需要为条件断点删除条件或者添加条件，可以使用 condition 命令，格式如下：

删除指定断点的条件：condition break-args

为指定断点增加条件：condition break-args if(condition)

例 2-13 设置条件断点

(1) 在 main()处中断，并依据命令行参数设置条件

break main if argc > 1

(2) 为循环结构设置条件断点

break if(i = = 100)

参考代码：

```
for( i=0; i<=1000; i++)
    do_something(i);
```

6. 监视点

监视点是一种特殊的断点，它与普通断点的区别在于，监视点没有固定于某一行源代码中，而是要求 GDB 每当某个表达式改变了值就暂停程序执行的指令。因此若需要确定变量在何处被改变，使用监视点是非常有效的。

设置监视点的命令格式：watch var。

此处的 var 可以是常量、变量或一个表达式。

说明

①监视点与变量的作用域相关

监视点只能监视“存在且在作用域内”的变量，一旦变量不再存在(当包含局部变量的函数返回时)，GDB 会自动删除监视点。

②监视点一般用来监视单个线程中的变量

同断点查看命令一样，用 info watchpoints 可以查看监视点的设置情况。

例 2-14 设置监视点

参考源代码：

```
#include <stdio.h>
int i = 0;
int main(void)
{
        i = 3;
        printf("i is %d\n", i);
        i = 5;
        printf("i is %d\n", i);
        return 0;
}
```

编译并启动调试：

```
$ gcc -g watch_break.c
$ gdb a.out
(gdb) b main
Breakpoint 1 at 0x80483ed: file watch_break.c, line 5.
(gdb) run
Starting program: /home/nachos/system/debug/a.out

Breakpoint 1, main () at watch_break.c:5
5       i = 3;
(gdb) watch i > 4
Hardware watchpoint 2: i > 4
(gdb) c
Continuing.
i is 3
Hardware watchpoint 2: i > 4

Old value = 0
New value = 1
main () at watch_break.c:8
8       printf("i is %d\n", i);
(gdb) n
i is 5
9       return 0;
```

2.4.5 检查和设置变量

在 GDB 中调试过程中，需要通过 print、display 等方式检查调试过程中的各种变量的信息，下面介绍常用的检查变量的方法。

1. print 命令

```
print <expr>
print /<f> <expr>
```

其中，<expr>是表达式；<f>表示输出格式。

print 命令可以接受表达式，gdb 会根据当前程序运行的数据来计算表达式。表达式可以是当前程序运行中的常量、变量和函数等内容，其语法应该是当前程序所用语言的语法。

一般来说，gdb 会根据变量的类型输出变量的值。但也可以自定义 gdb 的输出格式。

例 2-15 利用 print 查看命令行参数

```
(gdb) run 12 8 5 19 16     //启动程序，并传递命令行参数为 12,8,5,19,16
Starting program: /home/nachos/system/debug/a.out 12 8 5 19 16
Breakpoint 1, main (argc=6, argv=0xbffff4a4) at bintree.c:59
59          insert(&root,atoi(argv[i]));
(gdb) p argv
$2 = (char **) 0xbffff4a4
(gdb) p *argv
$3 = 0xbffff626 "/home/nachos/system/debug/a.out"
(gdb) p argv[1]
$4 = 0xbffff646 "12"
```

2. display 命令

GDB 的 display 命令(简写 disp)会在 GDB 执行中每次有暂停(由于有断点，使用 next 或 display 命令等)时就输出指定信息。

例 2-15 利用 display 查看变量信息

```
(gdb) display *argv
1: *argv = 0xbffff626 "/home/nachos/system/debug/a.out"
(gdb) n
58          for(i=1; i<argc; i++){
1: *argv = 0xbffff626 "/home/nachos/system/debug/a.out"    //在每一次暂停都执行了显示*argv
(gdb) n

Breakpoint 1, main (argc=6, argv=0xbffff4a4) at bintree.c:59
59                  insert(&root,atoi(argv[i]));
1: *argv = 0xbffff626 "/home/nachos/system/debug/a.out"
(gdb) n
58          for(i=1; i<argc; i++){
1: *argv = 0xbffff626 "/home/nachos/system/debug/a.out"
```

3. set 命令

GDB 的命令 set 用于在调试过程中改变变量的值。

命令格式：set variable <变量>=<表达式>

例 2-16 使用 set 在调试过程中设定变量的值

参考源代码：

```
#include <stdio.h>

int w[4]={12,8,5,19};
int main(void)
{
        w[2] = 88;
        return 0;
}
```

```
    (gdb) b main
Breakpoint 1 at 0x80483b7: file array.c, line 6.
```

```
(gdb) run
Starting program: /home/nachos/system/debug/a.out

Breakpoint 1, main () at array.c:6
6           w[2] = 88;
(gdb) n
7           return 0;
(gdb) set $i = 0
(gdb) p w[$i++]
$1 = 12
(gdb)
$2 = 8
(gdb)
$3 = 88
(gdb)
$4 = 19
```

2.4.6 GDB 高级检查

1. 显示栈帧

命令 backtrace 命令可以在程序暂停执行时显示栈帧。该命令简写为 bt。

命令格式：

(1) bt　显示所有栈帧

(2) bt N　只显示开头 N 个栈帧

(3) bt –N　只显示最后 N 个栈帧

命令 frame 可以指定显示某一栈帧的信息。

命令格式：　frame N　显示第 N 栈帧信息。

例 2-17　显示栈帧信息

```
(gdb) run
Starting program: /home/nachos/system/debug/a.out

Breakpoint 1, main () at swapper.c:5
5           int i = 4;
(gdb) n
6           int j = 6;
(gdb) n
7           printf("i: %d, j: %d\n",i,j);
(gdb) n
i: 4, j: 6
8           swap(i,j);
(gdb) s
swap (a=4, b=6) at swapper.c:15
15          int c = a;
(gdb) bt
#0    swap (a=4, b=6) at swapper.c:16
#1    0x0804842e in main () at swapper.c:8
```

(gdb) frame
#0 swap (a=4, b=6) at swapper.c:16
16   a = b;
(gdb) frame 0
#0 swap (a=4, b=6) at swapper.c:16
16   a = b;
(gdb) frame 1
#1 0x0804842e in main () at swapper.c:8
8   swap(i,j);
(**gdb) info frame 1**
Stack frame at 0xbffff420:
 eip = 0x804842e in main (swapper.c:8); saved eip 0x144bd6
 caller of frame at 0xbffff3f0
 source language c.
 Arglist at 0xbffff418, args:
 Locals at 0xbffff418, Previous frame's sp is 0xbffff420
 Saved registers:
  ebp at 0xbffff418, eip at 0xbffff41c
(gdb) info frame 0
Stack frame at 0xbffff3f0:
 eip = 0x804846a in swap (swapper.c:18); saved eip 0x804842e
 called by frame at 0xbffff420
 source language c.
 Arglist at 0xbffff3e8, args: a=6, b=4
 Locals at 0xbffff3e8, Previous frame's sp is 0xbffff3f0
 Saved registers:
  ebp at 0xbffff3e8, eip at 0xbffff3ec

2. info 命令

命令 info frame N 可以显示第 N 个栈的详细信息。

例 2-18 显示栈帧信息

(继续例 2-17)

(**gdb) info frame 1**
Stack frame at 0xbffff420:
 eip = 0x804842e in main (swapper.c:8); saved eip 0x144bd6
 caller of frame at 0xbffff3f0
 source language c.
 Arglist at 0xbffff418, args:
 Locals at 0xbffff418, Previous frame's sp is 0xbffff420
 Saved registers:
  ebp at 0xbffff418, eip at 0xbffff41c
(gdb) info frame 0
Stack frame at 0xbffff3f0:
 eip = 0x804846a in swap (swapper.c:18); saved eip 0x804842e
 called by frame at 0xbffff420

source language c.

Arglist at 0xbffff3e8, args: a=6, b=4

Locals at 0xbffff3e8, Previous frame's sp is 0xbffff3f0

Saved registers:

ebp at 0xbffff3e8, eip at 0xbffff3ec

命令 info registers 可以显示寄存器，简写为 info reg。使用命令 p/格式 变量可以显示各寄存器的内容。

常用显示格式如下：:

x 按十六进制格式显示变量。

d 按十进制进制格式显示变量。

u 按十进制进制格式显示无符号整数。

o 按八进制格式显示变量。

t 按二进制格式显示变量。

c 按字符格式显示变量。

f 按浮点数格式显示变量。

例 2-19 显示寄存器值

(继续例 2-19)

```
(gdb) info register
eax            0x4    4
ecx            0xbffff3d8 -1073744936
edx            0x285360  2642784
ebx            0x283ff4   2637812
esp            0xbffff3f0  0xbffff3f0
ebp            0xbffff418 0xbffff418
esi            0x0    0
edi            0x0    0
eip            0x804842e  0x804842e <main+74>
eflags         0x200286   [ PF SF IF ID ]
cs             0x73   115
ss             0x7b   123
ds             0x7b   123
es             0x7b   123
fs             0x0    0
gs             0x33   51
(gdb) p $eax
$1 = 4
```

2.4.7 多进程调试

1. attach 命令跟踪指定进程

GDB 有附着(attach)到正在运行的进程的功能，即 attach <pid>命令。因此可以利用该命令 attach 到子进程然后进行调试。

GDB 调试器的 attach 命令对已经运行的程序进行加载调试。不过这种做法要改变程序中的内容。通常的做法是：

在进程创建的过程中，在子进程中加入一些代码：

if（（pid = fork（））==0）

```
{
    //子进程
    int pause = 1;   //可以理解为一个开关
    while（pause）
    {
        sleep（1）;
        return 0;
    }
    ...
}
```

这样在程序的 int pause... 部分创建断点，在调试的过程中，会提示系统 Detaching ...，这其实是制定了子进程的 pid 号码；也可以在终端中调用 ls -ef 查看子进程的进程号，接着调用 attach 命令加载进程号。当进程挂起在断点的时候，我们可以改变 pause 的值为 0，从而跳过循环过程，重新执行子进程中的内容。这里要注意的一个地方是子进程中的结尾部分要有 return 0 语句(如果你的函数定义的是类似 int func(...))，这样 GDB 在调试中 detach 的时候可以自动结束或挂起进程，否则系统就会一直等在那里，什么都做不了了。

2.设置 follow-fork-mode

默认情况下，gdb 会继续跟踪父进程，无法对子进程进行调试。可以通过设置 follow-fork-mode 来实现对子进程的跟踪。

follow-fork-mode 格式：set follow-fork-mode [parent|child]

其中选项 parent 表示 fork 之后继续调试父进程，子进程不受影响；选项 child 表示 fork 之后调试子进程，父进程不受影响。

如果需要调试子进程，则在 GDB 启动后，执行命令(gdb)set follow-fork-mode child，并在子进程中设置断点，即可实现跟踪子进程执行流。

此外还有 detach-on-fork 参数，指示 GDB 在 fork 之后是否断开(detach)某个进程的调试，或者都交由 GDB 控制。

设置 detach-on-fork 参数：set detach-on-fork [on|off]

其中选项 on 表示断开调试 follow-fork-mode 指定的进程；off 表示 gdb 将控制父进程和子进程。若预设 follow-fork-mode 指定的进程将被调试，另一个进程置于暂停(suspended)状态，需要将 detach-on-fork 设置为 off 即可。

例 2-20 多进程调试案例

(1)存在错误代码的源程序

```
#include<stdio.h>
int main()
{
        if(fork() == 0)
        {
                int b = 9;
                sleep(60);
                int a =1;
                int c = 90;
                int d =5;
                printf("child\n");
        }
```

```
        else
        {
                wait(NULL);
                printf("parant\n");
        }
        return 0;
}
```

⑵分析

为了能够跟踪进入子进程，需要首先后台运行该程序，可以得到进程 ID。然后启动 GDB，并使用 attach ID 使执行流附着于子进程中。

(3)调试过程

Step1：初始工作

GDB 调试程序的前提条件就是你编译程序时必须加入调试符号信息，即使用“-g”编译选项。首先编译我们的源程序“gcc –g –o eg1 eg1.c”。编译之后产生调试目标文件 eg1。由于在调试过程中需要多个工具配合，所以最好多打开几个终端窗口，另外一点需要注意的是最好在 eg1 的“当前工作目录”下下执行 GDB 程序，否则会提示“No symbol table is loaded”。

Step2：开始运行

Step2.1 进程 eg1 进入后台运行

```
[root@localhost degbugprg]#./eg1 &
[1]4037
```

Step2.2 通过 ps 命令查看进程 id

```
[root@localhost degbugprg]#ps -fu
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
Root 2887 0.00.35744992pts/0S12:100:00 bash
Root 4037 0.00.01340236pts/0S13:260:00 ./eg1
Root 4038 0.00.01340232pts/0S13:260:00 \_./eg1
Root 4039 0.00.22616668pts/0R13:260:00 ps–fu

(gdb) attach 4038
Attaching to program: /home/purerain/test/f, process 12606
Symbols already loaded for /lib/tls/libc.so.6
Symbols already loaded for /lib/ld-linux.so.2
0xffffe002 in ?? ()
(gdb) stop     /*防止子进程自己跑完,在子进程中设置断点*/
(gdb) b 12
Breakpoint 8 at 0x8048402: file fork.c, line 12.
(gdb) c
Continuing.

Breakpoint 7, main () at fork.c:11
11                              int d =5;
(gdb) s
12                              printf("child\n");
(gdb) s
```

```
19                      return 0;
(gdb) s
20          }
(gdb) s
0x42015574 in __libc_start_main () from /lib/tls/libc.so.6
(gdb) s
Single stepping until exit from function __libc_start_main,
which has no line number information.

Program exited normally.
```

例 2-21 多进程程序调试案例

(1)存在错误代码的源程序

```
int wib(intno1,intno2)
{
        intresult,diff;
        diff=no1-no2;
        result=no1/diff;
        returnresult;
}
int main()
{
        pid_t pid;
        pid=fork();
        if(pid<0){
            printf("forkerr\n");
            exit(-1);
        }else if(pid==0){
            /*inchildprocess*/
            sleep(60);----------(!)
                intvalue=10;
            int div=6;
            int total=0;
            int i=0;
            int result=0;
            for(i=0;i<10;i++){
                result=wib(value,div);
                total+=result;
                div++;
                value--;
            }
            printf("%dwibedby%dequals%d\n",value,div,total);
            exit(0);
        }else{
            /*inparentprocess*/
```

```
            sleep(4);
            wait(-1);
            exit(0);
        }
}
```

(2)分析

该测试程序中子进程运行过程中会在 wib 函数中出现一个“除 0”异常。现在我们就要调试该子进程。

(3)调试原理

测试程序在父进程 fork 后，子进程调用 sleep 睡眠 60 秒。这个 sleep 本来是不该存在于子进程代码中的，而是为了使用 GDB 调试后加入的，它是调试的一个关键点。目的是为了在子进程睡眠期间，利用 shell 命令获取其进程 PID，然后再利用 GDB 调试外部进程的方法 attach 到该 PID 上，调试该进程。

(4)调试过程

Step1：初始工作

编译产生可调试的目标文件：gcc –g –o eg2 ineg1.c

Step2：开始运行

Step2.1 进程 eg2 进入后台运行

[root@localhostprograms]#./eg2&

[1]4052

Step2.2 通过 ps 命令查看进程 id

[root@localhostprograms]#ps -fu

USERPID%CPU%MEMVSZRSSTTYSTATSTARTTIMECOMMAND

Root 2887 0.00.35744992pts/0S12:100:00 bash

Root 4052 0.00.01340236pts/0S13:260:00 ./eg2

Root 4053 0.00.01340232pts/0S13:260:00 _./eg2

Root 4054 0.00.22616668pts/0R13:260:00 ps–fu

step3:调试阶段

step3.1 启动 gdb

[root@localhostprograms]#gdb

GNUgdbRedHatLinux(5.3post-0.20021129.18rh)

Copyright2003FreeSoftwareFoundation,Inc.

GDBisfreesoftware,coveredbytheGNUGeneralPublicLicense,andyouare

welcometochangeitand/ordistributecopiesofitundercertainconditions.

Type"showcopying"toseetheconditions.

ThereisabsolutelynowarrantyforGDB.Type"showwarranty"fordetails.

ThisGDBwasconfiguredas"i386-redhat-linux-gnu".

Step3.2:利用 attach 命令，进入指定进程的调试，这里 4053 是通过 ps 命令获得的子进程 PID.

(gdb)attach 4053

Attaching to process4053

Reading symbols from /home/user/programs/eg2...done.

Reading symbols from /lib/tls/libc.so.6...done.

Loaded symbols for /lib/tls/libc.so.6

Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
0xffffe002 in ??()
Step3.3:通过 stop 命令，先暂停子进程，然后设置断点
(gdb)stop
(gdb)l (list)
6{
7intresult,diff;
8diff=n1-n2;
9result=n1/diff;
10returnresult;
11}
12intmain()
13{
14pid_t pid;
15pid=fork();
(gdb)
16if(pid<0)
17{
18printf("forkerror\n");
19exit(-1);
20}else if(pid==0)
21{
22/*inchildprocess*/
23sleep(60);
24int value=10;
25int div=6;
(gdb) 回车 (l 或 list)
26int total=0;
27int i=0;
28int result=0;
29for(i=0;i<10;i++)
30{
31result=wib(value,div);
32total+=result;
33div++;
34value--;
35}
Step3.4:在 main()中语句 result=wib(value,div);这行设置断点。
(gdb)break 31
Breakpoint1at0x8048497:fileeg2.c,line31.
(gdb)continue
Continuing.

Breakpoint1,main()ateg2.c:31
31result=wib(value,div);
(gdb)step
wib(n1=10,n2=6)ateg2.c:8
8diff=n1-n2;
(gdb)continue
Continuing.

Breakpoint1,main()ateg2.c:31
31result=wib(value,div);
(gdb)step
wib(n1=9,n2=7)ateg2.c:8
8diff=n1-n2;
(gdb)continue
Continuing.

Breakpoint1,main()ateg2.c:31
31result=wib(value,div);
(gdb)step
wib(n1=8,n2=8)ateg2.c:8
8diff=n1-n2;
(gdb)next
9result=n1/diff;
Step3.5:查看变量值
(gdb)print diff
$1=0/*除数为 0!，找到错误原因*/
(gdb)next

ProgramreceivedsignalSIGFPE,Arithmeticexception.
0x0804840binwib(n1=8,n2=8)ateg2.c:9
9result=n1/diff;
(gdb)quit
The program is running.Quit any way(and detach it)?(y or n)y
Detaching from program:/home/user/programs/eg2,process 4053
[root@localhostprograms]#

2.4.8 多线程调试

GDB 提供了多线程调试的功能。主要使用的命令有：

(1) thread THEADNO：用于线程间切换

(2) info threads： 显示当前线程信息

使用 GDB 调试的时候，通常只有一个线程为活动线程，如果希望得到其他线程的结果，必须使用命令 thread THREADNO 切换至指定的 THREADNO 线程，才能对该线程进行调试和观察。

例 2-22 多线程调试

(程序源代码见附录 threads_debug.c)，关于程序的说明：

此实例实现经典的埃拉托色尼筛选法求素数。要求 2-n 之间的素数，首先列出所有这些数字，然后去掉所有 2 的倍数，再去掉所有 3 的倍数，以此类推。最后剩下的就是素数。

程序运行编译及运行形式：

gcc -g threads_debug.c -lpthread –lm //-lpthread –lm 为了链接 Pthreads 和数学函数库

./a.out 100 4　　//100 表示求 2-100 之间的素数；　4 表示创建 4 个线程。

下面是调试过程。

$ gdb a.out

GNU gdb (GDB) 7.1-ubuntu

Copyright (C) 2010 Free Software Foundation, Inc.

License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

This is free software: you are free to change and redistribute it.

There is NO WARRANTY, to the extent permitted by law.　Type "show copying"

and "show warranty" for details.

This GDB was configured as "i486-linux-gnu".

For bug reporting instructions, please see:

<http://www.gnu.org/software/gdb/bugs/>...

Reading symbols from /home/nachos/system/debug/a.out...done.

(gdb) run 100 4　　//表示

Starting program: /home/nachos/system/debug/a.out 100 4

[Thread debugging using libthread_db enabled]

[New Thread 0xb7fe9b70 (LWP 2790)]

[New Thread 0xb77e8b70 (LWP 2791)]

[New Thread 0xb6fe7b70 (LWP 2792)]

[New Thread 0xb67e6b70 (LWP 2793)]　<-------- 程序在此处阻塞

^C　<---------- 用户按下 ctrl+c 中断程序

Program received signal SIGINT, Interrupt.

0x0012d422 in __kernel_vsyscall ()

(gdb) info threads　　<-----查看系统中线程信息

　5 Thread 0xb67e6b70 (LWP 2793)　0x0012d422 in __kernel_vsyscall ()

　4 Thread 0xb6fe7b70 (LWP 2792)　0x0012d422 in __kernel_vsyscall ()

　3 Thread 0xb77e8b70 (LWP 2791)　0x0012d422 in __kernel_vsyscall ()

　2 Thread 0xb7fe9b70 (LWP 2790)　0x0012d422 in __kernel_vsyscall ()

* 1 Thread 0xb7fea6c0 (LWP 2787)　0x0012d422 in __kernel_vsyscall ()

(gdb) bt　　<----------　bt 命令查看栈信息

#0　0x0012d422 in __kernel_vsyscall ()

#1　0x00134b5d in pthread_join () from /lib/tls/i686/cmov/libpthread.so.0

#2　0x080486de in main (argc=3, argv=0xbffff4b4) at threads_debug.c:75

(gdb) thread 1　　<----------切换到第 1 个线程

[Switching to thread 1 (Thread 0xb7fea6c0 (LWP 2787))]#0　0x0012d422 in __kernel_vsyscall ()

(gdb) bt　　<---------- 由栈信息可以看出线程 1 是主线程

#0　0x0012d422 in __kernel_vsyscall ()

#1　0x00134b5d in pthread_join () from /lib/tls/i686/cmov/libpthread.so.0

#2　0x080486de in main (argc=3, argv=0xbffff4b4) at threads_debug.c:75

(gdb) thread 2

[Switching to thread 2 (Thread 0xb7fe9b70 (LWP 2790))]#0 0x0012d422 in __kernel_vsyscall ()

(gdb) bt

#0 0x0012d422 in __kernel_vsyscall ()

#1 0x0013aaf9 in __lll_lock_wait () from /lib/tls/i686/cmov/libpthread.so.0

#2 0x0013613b in _L_lock_748 () from /lib/tls/i686/cmov/libpthread.so.0

#3 0x00135f61 in pthread_mutex_lock () from /lib/tls/i686/cmov/libpthread.so.0

#4 0x080485a9 in worker (tn=0) at threads_debug.c:40

#5 0x0013396e in start_thread () from /lib/tls/i686/cmov/libpthread.so.0

#6 0x0023aa0e in clone () from /lib/tls/i686/cmov/libc.so.6

(gdb) thread 3

[Switching to thread 3 (Thread 0xb77e8b70 (LWP 2791))]#0 0x0012d422 in __kernel_vsyscall ()

(gdb) bt

#0 0x0012d422 in __kernel_vsyscall ()

#1 0x0013aaf9 in __lll_lock_wait () from /lib/tls/i686/cmov/libpthread.so.0

#2 0x0013613b in _L_lock_748 () from /lib/tls/i686/cmov/libpthread.so.0

#3 0x00135f61 in pthread_mutex_lock () from /lib/tls/i686/cmov/libpthread.so.0

#4 0x080485a9 in worker (tn=1) at threads_debug.c:40

#5 0x0013396e in start_thread () from /lib/tls/i686/cmov/libpthread.so.0

#6 0x0023aa0e in clone () from /lib/tls/i686/cmov/libc.so.6

(gdb) thread 4

[Switching to thread 4 (Thread 0xb6fe7b70 (LWP 2792))]#0 0x0012d422 in __kernel_vsyscall ()

(gdb) bt

#0 0x0012d422 in __kernel_vsyscall ()

#1 0x0013aaf9 in __lll_lock_wait () from /lib/tls/i686/cmov/libpthread.so.0

#2 0x0013613b in _L_lock_748 () from /lib/tls/i686/cmov/libpthread.so.0

#3 0x00135f61 in pthread_mutex_lock () from /lib/tls/i686/cmov/libpthread.so.0

#4 0x080485a9 in worker (tn=2) at threads_debug.c:40

#5 0x0013396e in start_thread () from /lib/tls/i686/cmov/libpthread.so.0

#6 0x0023aa0e in clone () from /lib/tls/i686/cmov/libc.so.6

(gdb) thread 5

[Switching to thread 5 (Thread 0xb67e6b70 (LWP 2793))]#0 0x0012d422 in __kernel_vsyscall ()

(gdb) bt

#0 0x0012d422 in __kernel_vsyscall ()

#1 0x0013aaf9 in __lll_lock_wait () from /lib/tls/i686/cmov/libpthread.so.0

#2 0x0013613b in _L_lock_748 () from /lib/tls/i686/cmov/libpthread.so.0

#3 0x00135f61 in pthread_mutex_lock () from /lib/tls/i686/cmov/libpthread.so.0

#4 0x080485a9 in worker (tn=3) at threads_debug.c:40

#5 0x0013396e in start_thread () from /lib/tls/i686/cmov/libpthread.so.0

#6 0x0023aa0e in clone () from /lib/tls/i686/cmov/libc.so.6

由以上观察，在 bt 输出的栈#4 中显示了线程号。同时可以看到，在除了主线程外的其他各线程的栈帧#3 中，都去申请锁，而没有线程释放锁，因此造成了所有线程都在等待锁，导致程序挂起。

例 2-23 常用线程命令举例

(gdb) info threads //给出关于当前所有线程的信息，*标记当前活动线程

* 5 Thread 0xb67e6b70 (LWP 2793) 0x0012d422 in __kernel_vsyscall ()

4 Thread 0xb6fe7b70 (LWP 2792) 0x0012d422 in __kernel_vsyscall ()

3 Thread 0xb77e8b70 (LWP 2791) 0x0012d422 in __kernel_vsyscall ()

2 Thread 0xb7fe9b70 (LWP 2790) 0x0012d422 in __kernel_vsyscall ()

1 Thread 0xb7fea6c0 (LWP 2787) 0x0012d422 in __kernel_vsyscall ()

(gdb) thread 3 //切换到线程 3

[Switching to thread 3 (Thread 0xb77e8b70 (LWP 2791))]#0 0x0012d422 in __kernel_vsyscall ()

(gdb) info threads

5 Thread 0xb67e6b70 (LWP 2793) 0x0012d422 in __kernel_vsyscall ()

4 Thread 0xb6fe7b70 (LWP 2792) 0x0012d422 in __kernel_vsyscall ()

* 3 Thread 0xb77e8b70 (LWP 2791) 0x0012d422 in __kernel_vsyscall ()

2 Thread 0xb7fe9b70 (LWP 2790) 0x0012d422 in __kernel_vsyscall ()

1 Thread 0xb7fea6c0 (LWP 2787) 0x0012d422 in __kernel_vsyscall ()

(gdb) break 10 thread 3 //当线程 3 到达源程序行 10 时停止执行

Breakpoint 1 at 0x804855a: file threads_debug.c, line 10.

(gdb) break 10 thread 3 if x==y

No symbol "x" in current context.

(gdb) break 10 thread 3 if base == lim

No symbol "base" in current context.

(gdb) break 10 thread 3 if n==2 //当线程 3 到达源程序行 10 且变量 n==2 是停止执行

Note: breakpoint 1 (thread 3) also set at pc 0x804855a.

Breakpoint 2 at 0x804855a: file threads_debug.c, line 10.

2.4.9 信号处理函数的程序调试

信号是一种软中断，是一种处理异步事件的方法。一般来说，操作系统都支持许多信号，尤其是在 Unix/Linux 中，比较重要的应用程序一般都会处理信号。Linux 定义了许多信号，比如 SIGINT 表示中断信号，也就是 Ctr+C 的信号，SIGBUS 表示硬件故障的信号；SIGCHLD 表示子进程状态改变信号；SIGKILL 表示终止程序运行的信号等。信号量编程是 Linux 下非常重要的一种技术。GDB 有能力在调试程序时处理任何一种信号，可以告诉 GDB 需要处理哪一种信号。可以要求 GDB 收到所指定的信号后立刻停止正在运行的程序，以便进行调试。可以用 GDB 的 handle 命令来完成这一功能。如下所示：

(gdb) handle <signal> <keyword…>

在 GDB 中定义一个信号处理。信号 signal 可以以 SIG 开头或不以 SIG 开头，可以定义一个要处理的信号的范围(如 SIGIO-SIGKILL，表示处理从 SIGIO 信号到 SIGKILL 的信号，其中包括了 SIGIO、SIGIOT 和 SIGKILL3 个信号)，也可以使用关键字 all 来表明要处理的所有信号。一旦被调试的程序接收到信号，运行程序马上被 GDB 停住，以供调试。其中 keyword 可以是表 2-22 中列出的一个或多个关键字。

表 2-22 keyword 关键字

断点名称	含义
nostop	GDB 不会停止程序的运行，但会打出消息告诉你收到这种信号
stop	GDB 会停止程序
print	GDB 会显示出一条信息
noprint	GDB 不会告诉你收到信号的信息
pass noignore	GDB 不处理信号，这表示 GDB 会把这个信号交给被调试程序处理
nopass ignore	GDB 不会让被调试程序来处理这个信号
info signals… info handl…	查看有哪些信号在被 GDB 检测中

2.5 小结

任何应用程序及系统程序的开发都离不开编辑器、编译器及调试器。本章介绍的 Linux 编辑器 Vim，编译器 GCC，调试器 GDB 及 Make 工程管理器是 Unix/Linux 系统中的优秀的编辑、编译和调试功工具。掌握这些开发工具至关重要，它直接影响到程序开发的效率和质量。

第 3 章 基本实验单元

3.1 实验一 Linux 基本操作

一、 实验目的

(1) 了解 Linux 操作系统的文件构成。

(2) 了解 UNIX/Linux Shell 的作用及基本应用。

(3) 掌握编辑工具 vi 的基本使用方法。

(4) 掌握 GCC 编译器的基本用法。

(5) 掌握 GDB 的基本用法。

二、 实验内容

1. 在 shell 命令模式下使用命令行操作

常用命令：man、cd、cat、more、ls、ps、chmod、kill、ln、cp、mv、rm、cd、pwd、mkdir、chown、who、w、wc、whoami、date、uname、touch 等。

(1) 利用 cd 命令进入当前用户的工作目录，并在其下分别建立以下子目录：fileio，process，thread，test

(2) 进入 fileio 目录建立以下子目录：include，src，obj，bin

(3) 进入 test 目录，完成附录 1 中的命令操作。

2. 基于 Vi、GCC、GDB 的基础编程

编程实现 cp 命令

(1)使用 vi 编辑源程序

(2) 使用 gcc 编译

```
# gcc -g -c -I ./include src/cp.c -o obj/cp.o
# gcc -g obj/io.o obj/cp.o -o bin/cp
# bin/cp f1 f2
# bin/cp f1.txt f2.txt
# diff f1.txt f2.txt
```

(3) 使用 gdb 调试

要求:

使用 diff 比较工具比对 cp 操作的源文件和目标文件，若没有给出任何提示，则说明它们的内容完全相同。

阅读与系统调用相关的联机帮助，学习系统调用的使用方法

3.2 实验二 文件 I/O

一、实验目的

(1) 理解 Linux 系统中文件系统的基本概念。

(2) 掌握 Linux 系统中文件 I/O 系统调用的使用方法；

(3) 深入理解文件重定向的概念

(4) 能应用各系统调用实现文件系统的综合设计；

(5) 掌握基本的 Linux 环境编程编译和调试技术。

二、背景知识

1. 文件描述符与文件指针 FILE 结构

2. 文件 I/O 系统调用：open()、read()、write()、close()、lseek()、dup()、dup2()等；

三、实验项目

1. 模拟实现 cp 命令

2. 模拟实现 cat 命令

3. 实现用户自己定义的 stdio 库

四、实验内容

1. 模拟实现 cp 命令

命令 cp 的典型用法是：cp source-file target-file

标准的 cp 命令实现时，如果 target-file 所指定的文件不存在，则 cp 创建这个文件，如果已存在就覆盖，而不给出任何提示；target-file 的内容与 source-file 相同。

2. 模拟实现 cat 命令

命令 cat 的用法有以下几种形式：

cat:从标准输入内容，在标准输出显示。

cat file1 file2 ...：将 file1,file2 文件的内容在标准输出显示。

cat > file：将从标准输入的内容写入文件 file 中。

cat < file：将文件 file 的内容显示到标准输出。

cat < file1 > fil2：将文件 file1 的内容写入 file2 中。

请使用文件 I/O 系统调用模拟实现 cat 以上功能。

3. 模拟实现 stdio 库函数

使用文件 I/O 系统调用模拟实现 stdio 库函数 fopen、fclose、fgetc、fputc、。

提示：FILE 结构体

FILE 结构体的构建时机及内存位置？

FILE 结构体中存放哪些内容？

为什么设计 FILE 结构体，通过文件描述符

拓展实验：

(1) 完善 stdio 库函数，在以上基本实验基础上，继续完整的标准 I/O 函数功能，例如：fread、fwrite、fputs、fgets、flseek 等。

3.3 实验三 进程管理

一、实验目的

(1)了解进程的运行环境，加深对进程概念的理解。

(2)掌握进程控制相关系统调用的使用。

(3)掌握 GDB 关于进程的调试技术。

二、背景知识

1. 进程运行环境，包括命令行、环境表、内存映像；
2. 进程描述符；
3. 进程管理核心系统调用：fork、wait、exec、_exit
4. 进程组及前(后)台进程

四、实验项目

1. 基本进程管理命令的练习
2. 模拟实现简单的 shell
3. 模式实现 system

五、实验内容

1. 常用进程管理命令的使用

(1) ps 命令

ps 命令是收集进程信息的重要工具，它提供的信息包括：拥有进程的用户、进程的起始时间、进程对应的命令行路径、PID、进程所属的终端(TTY)、进程占用的 CPU 时间等。可以通过 man ps 了解 ps 命令的详细使用说明。通过以下命令熟悉 ps 命令的使用。

```
$ps
$ps -ef
$ps -ef |grep bash
$ps -ef | head
$ps -aux | head
$ps -o pid,ppid,cmd | head
```

说明：

-o 选项可以使用不同的参数，这些参数及其描述如下表所示。

参数	描述	参数	描述
pcpu	CPU 占用率	Nice	优先级
Pid	进程 ID	Time	累计的 CPU 时间
Ppid	父进程 ID	Etime	进程启动后流逝的时间
Pmem	内存使用率	Tty	所关联的 TTY 设备
Comm	可执行文件名	Euid	有效用户 ID
Cmd	简单命令	Stat	进程状态
User	启动进程的用户		

(2) top 命令

top 命令默认输出一个占用 CPU 最多的进程列表。输出结果每隔几秒就会更新。可以通过 man top 了解 top 命令的详细使用说明。

练习命令：

```
$top
```

(3) pstree 命令

pstree 命令以树状显示进程关系及进程信息。

练习：

$pstree

$pstree –p

2. 模拟实现简单的 system

(1) system 基本功能

凡是在 shell 中使用的任意命令字符串，都可以通过以下两种方式执行：

[1]命令形式：$ bash –c　cmdstring

[2] 函数形式：system(cmdstring)

请尝试使用 system()和 bash -c 执行以下命令：

cat /etc/passwd

cat /etc/passwd | grep root

cat /etc/passwd > ./s.txt

(2)system 的运行机制

system 对于任意的 shell 命令都可以执行，特别命令中会包含如“|”或“>、<”等符号。而 system 的实现，需要创建子进程，且子进程启动后要进行代码替换。

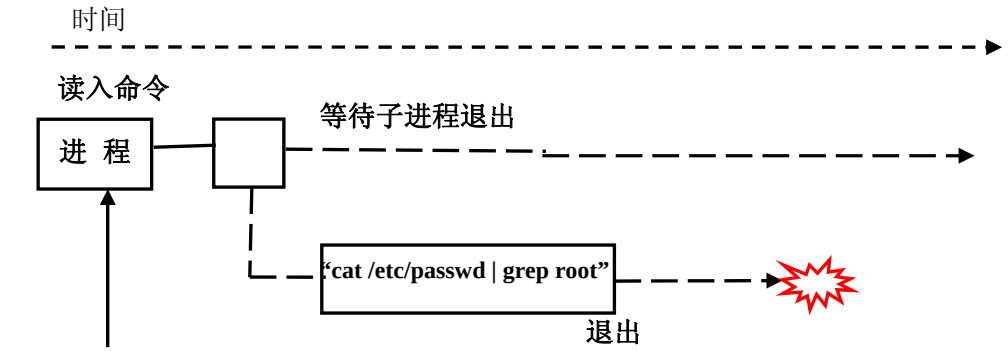

图 1 system()工作模型

(3) system 设计

通过对 system 的工作原理分析可知，为了实现 system()，需要使用 fork()来创建一个子进程，然后调用 exec()以 bash 程序作为新代码进行替换。为了收集 system()所创建的子进程状态，还以指定的子进程 ID 调用 waitpid()(使用 wait()并不合适，因为 wait()等待的是任一子进程，因而无意间所获取的子进程状态可能属于其他子进程)。

(4) system 实现

(略)

3. 模拟实现 shell

(1) shell 基本功能介绍

shell 是一个管理进程和运行程序的程序。Unix 系统有很多种可用的 shell，每种都有各自的风格和优势。所有常用的 shell 都具有运行程序、执行命令的功能：

A) 执行命令

Shell 可以看命令解释器，可以执行各种系统命令。这些命令分两类：内部命令和实用程序。可以使用 type 命令查看系统命令是否是内置命令。 其中内置命令如 cd、pwd、export 等，这些命令的解释程序构造在 shell 内部，可以有效提高命令执行效率；而实用程序如 date、

cat 等都是一些普通的程序，被编程成机器语言，由 shell 将它们载入内存并执行。

B)用户程序

用户程序经过编译生成可执行文件后，也可作为 Shell 命令运行。

(2) shell 的运行机制

Shell 打印提示符后，用户输入命令。Shell 就运行这个命令，然后 shell 再次打印提示符。那么 shell 的内部运行机制是怎样的呢？

例：$ ls

$ cat /etc/passwd

用时间轴来表示事件发生次序。其中时间从左向右消逝，shell 由标识为 sh 的方块代表，它随时间的流逝从左到右移动。Shell 从用户读入字符串“ls”。Shell 建立一个新的进程，然后在那个进程上运行新程序，并等待那个进程结束。

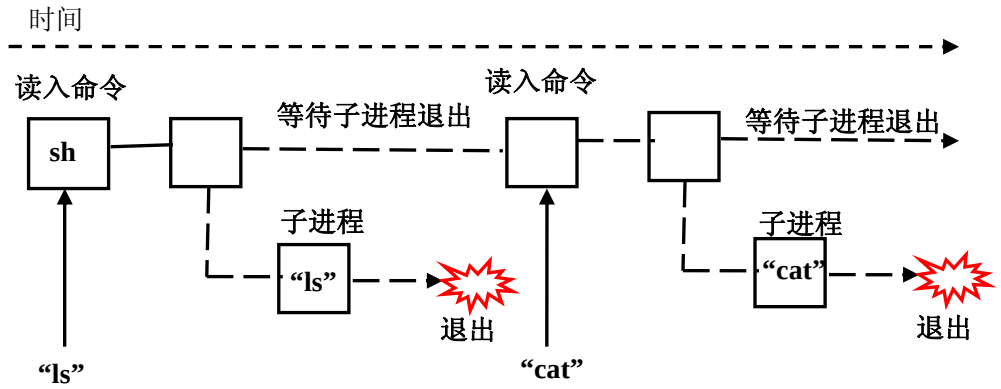

图 2 shell 工作模型

(3) shell 设计

通过对 shell 的工作原理可知，shell 用 fork 创建新进程，用 exec 在新进程中运行用户指定的程序，最后 shell 用 wait 等待新进程结束。Wait 系统调用同时从内核取得退出状态或信号序号以告知子进程是如何结束的。

Shell 工作模型如下图所示。

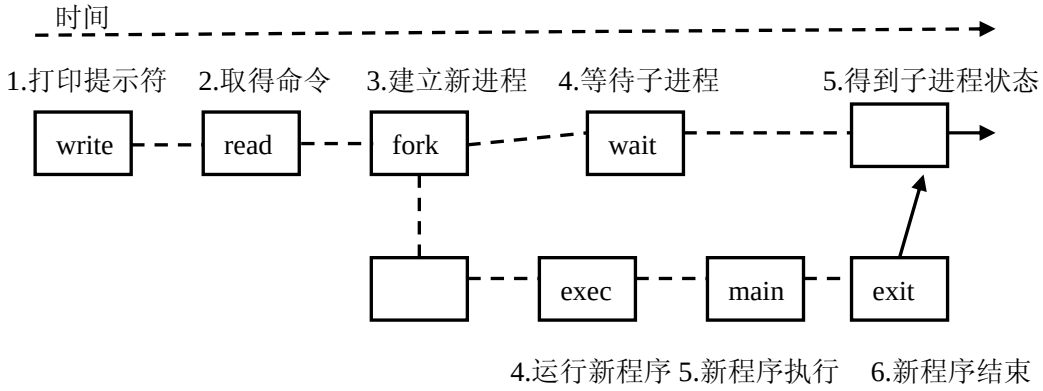

图 3 shell 的 fork()、exec()和 wait()循环

(4) 难点解析

[1]命令行参数

shell 将命令行读入缓冲区，再将缓冲区切割为参数列表。

例：cat /etc/passwd

argv[0]=”cat”;

argv[1]=”/etc/passwd”;

argv[2]=NULL;

execvp(argv[0],argv);

[2]内部命令的处理

由于内部命令的实现是内嵌在 shell 内部的，在模拟实现 shell 时，对于 shell 的内部命令，需要用户自己编码实现其功能。

(5)shell 实现

[1]数据结构

```
typedef struct
{
        pid_t pid;
        char **cmds;
}Program;
```

[2]函数说明

```
Program* creat_program(char *line);
void free_program(Program *prog);
void execute_program(Program *prog);
```

请自行补充代码，并进行全面测试

3.4 实验四 中断机制

一、实验目的

(1) 掌握信号、信号集特点及处理方式。

(2) 了解 Linux 系统信号机制的内核实现基本原理。

(3) 了解信号对进程执行的影响及使用策略

(4) 掌握信号机制相关系统调用的使用。

(5) 熟练应用 Vi 编辑器

(6) 熟练应用 GCC 编译器

(7) 掌握 makefile 的使用技术。

二、基础知识

1. 信号特点及处理方式。

2. 理解信号屏蔽字和信号未决字的概念及内核结构。

3. 信号相关系统调用：signal、kill 等。

4. 信号集操作的使用方法。

三、实验项目

1. kill 命令的使用

2. 改进 system

3. 模拟实现 time 命令

3. 信号实现“读者——-写者”问题

四、实验内容

1. kill 命令的使用

信号是 Linux 中的一种进程间通信机制。当进程接收到一个信号时，它会通过执行相应的信号处理程序(signal handler)来进行响应。在 shell 中可以发送、接收并响应信号。Kill 是用于终止进程的信号。像 ctrl+c、ctrl+z 这种事件也会发送各自的信号。Kill 命令可用来向进程发送信号。

(1) 列出所有可用的信号

形式：$kill -l

(2) 终止进程

形式：$kill PROCESS_ID_LIST

kill 命令默认发出一个 TERM 信号。进程 ID 列表使用空格作为进程 ID 之间的定界符。

(3) 通过 kill 命令向进程发送指定的信号

形式：$kill –s SIGNAL PID

参数 SIGNAL 要么是信号名称，要么是信号编号。经常用到的信号具体如下：

SIGINT 2——当按下 Ctrl+C 时发送该信号

SIGKILL 9——用于强行杀死进程

SIGTERM 15——默认用于终止进程

SIGTSTP 20——当按下 Ctrl+Z 时发送该信号

SIGCONT 18——使暂停的进程继续运行。

(4) 强行杀死进程

形式：$kill –s SIGKILL PID

$kill -9 PID

请设计实例练习 kill 命令的使用方法。

2. 改进 system

(1) system()中的信号处理

实验 3 实现了一个缺乏信号处理的 system()。本实验需要在 system()内部正确处理信号，需要考虑 SIGCHLD 和 SIGINT 及 SIGQUIT 信号的正确处理。

首先观察：SIGCHLD

例如：system(“cat …”)；

system() 的程序直接创建了子进程(执行 cat…)。在这种情况下，当由 system()所创建的子进程退出并产生 SIGCHLD 信号时，在 system()有机会调用 waitpid()之前，主程序的信号处理器程序可能率先得以执行(收集到子进程的状态)。这是竞争条件，会产生两种不良后果：

(a) 调用程序误以为其所创建的某个子进程已经终止了。

(b) system()函数却无法获取其所创建子进程的终止状态。

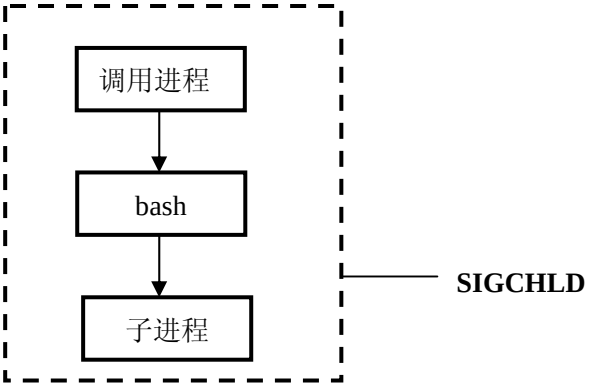

图 1

其次：观察 SIGINT 和 SIGQUIT

由终端的中断(interrupt)和退出(quit)所产生的 SIGINT 和 SIGQUIT 信号。在执行如下调用时的后果：

system(“sleep 20”);

此时此刻会产生 3 个进程，执行调用程序的进程，一个 shell 进程以及 sleep 进程，如图 2 所示所有进程构成终端的前台进程组的一部分。所以，在输入中断或退出符时，会将相应信号发送给所有 3 个进程，这与实际不符合。

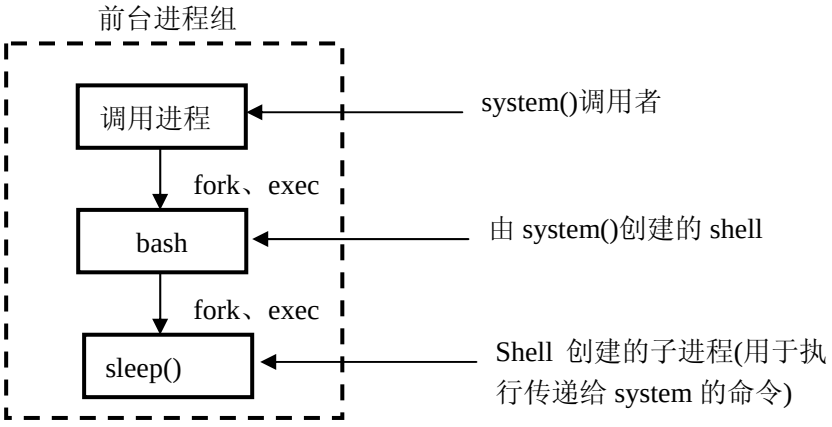

图 2

(2) 带信号处理的 system()

[1] 调用进程(调用 system())在运行期间必须阻塞 SIGCHLD 信号。

[2]调用进程(调用 system())在执行命令期间应当忽略 SIGINT 和 SIGQUIT 信号。

[3]子进程对 SIGINT 和 SIGQUIT 的处理，如同调用进程调用了 fork()和 exec()一般，将对已处理信号的处置重置为默认值，而对其他信号的处置则保持不变。

(3) 编码实现

(略)

3. 模拟实现 time 命令

(1) 进程时间

在 Linux 中，time 命令可以测量任意进程的 3 种运行时间。

$time ps

PID TTY TIME CMD

3438 pts/0 00:00:00 bash

5683 pts/0 00:00:00 ps

real 0m0.233s

user 0m0.020s

sys 0m0.020s

(2) 分析

模拟实现 time 的功能，统计进程运行时在用户模式、内核模式及总的运行时间。需要使用 3 个定时器来测量进程的处理器使用率，同时使用 signal 机制建立信号处理程序，记录和统计进程的实际、虚拟的和活跃的 3 种时间。

可以使用 setitimer()函数设置定时器，用户可以指定 3 种策略：ITIMER_REAL、ITIMER_VITRUAL、ITIMER_PROF，通过在一个进程中设置 3 种定时器，统计一个进程在用户模式、内核模式及总的运行时间。

信号：SIGALRM、SIGVTALRM、SIGPROF

(3) 设计方案

[A.1]定义 3 个计时器变量，分别记录进程的 3 种运行时间，定义 3 中不同的定时器，设置这些定时器的发生时间，并将这些定时器的发生频率设置为一直发生。

[A.2] 每当定时器超时并产生一个信号后，只需要增加相应计时变量的值便粗略地得到了进程运行的 3 种时间。

(4)编码实现

4. 信号实现“读者——写者”问题

参考《Unix 环境高级编程》第 10 章 第 272 页 程序清单 10-17 设计的信号同步函数，实现读者、写者进程之间的同步。

3.5 实验五 进程通信

一、实验目的

(1)了解 Linux 系统中进程通信的基本原理。

(2)掌握匿名管道及命名管道通信机制，学会通过管道实现进程间通信。

(3)掌握和使用共享内存实现进程间通信。

(4)掌握和使用消息队列实现进程间通信。

(5)掌握和使用信号量实现进程同步。

(6) 熟练应用 Vi 编辑器

(7) 熟练应用 GCC 编译器

(8) 掌握 makefile 的使用技术。

二、基础知识

1. System V IPC 的特点。

2. 管道及命名管道实现机制及读写规则。

3. 消息队列通信机制及相关系统调用。

4. 共享内存通信机制及相关系统调用。

三、实验项目

1. ipc 命令的使用

2. 简单聊天室的实现

四、实验内容

1. ipc 命令的使用

Linux 中，与 IPC 相关的命令包括：ipcs、ipcrm(释放 IPC)。ipcs 命令是 Linux 下显示进程间通信设施状态的工具，可以查看共享内存、信号量、消息队列的状态。可以使用 man ipcs 了解 ipcs 命令详细用法，现列举 ipcs 命令的常用方法。

(1) 显示所有的 IPC 设施

ipcs -a

(2) 显示所有的消息队列 Message Queue

ipcs -q

(3) 显示所有的信号量

ipcs -s

(4) 显示所有的共享内存

ipcs -m

(5) 显示 IPC 设施的详细信息

ipcs -q -i id

说明：id 对应 shmid、semid、msgid 等。-q 对应设施的类型（队列），查看信号量详细情况使用-s，查看共享内存使用-m。

(6) 显示 IPC 设施的限制大小

ipcs -m -l

说明：-m 对应设施类型，可选参数包括-q、-m、-s。

(7) 显示 IPC 设施的权限关系

ipcs -c

ipcs -m -c

ipcs -q -c

ipcs -s -c

(8) 显示最近访问过 IPC 设施的进程 ID。

ipcs -p

ipcs -m -p

ipcs -q -p

(9) 显示 IPC 设施的最后操作时间

ipcs -t

ipcs -q -t

ipcs -m -t

ipcs -s -t

(10) 显示 IPC 设施的当前状态

ipcs –u

2. 使用 IPC 机制实现简单的聊天室

假设 lucy 和 peter 使用聊天室聊天，lucy 必须等到 peter 在线才能说话，然后需要等待 peter 应答才能继续发话，这样一来一往的聊天模式，如果不想聊了，说 quit 即可。

(1) 使用 FIFO 构建简单的聊天室

(2) 使用消息队列构建简单的聊天室

(3) 使用共享内存构建简单的聊天室

3.6 实验六 进程同步

一、实验目的

(1) 深入理解进程同步机制。
(2) 掌握各种进程同步的实现方法
(4) 灵活应用信号量及信号、管道实现进程的同步。
(6) 熟练应用 Vi 编辑器
(7) 熟练应用 GCC 编译器
(8) 掌握 makefile 的使用技术。

二、基础知识

1. 进程同步及互斥的概念。
2. 管道实现进程同步的基本方法。
3. 信号实现进程同步的基本方法。
4. 信号量机制及其应用。

三、实验内容

1. 请使用共享内存段和信号量实现“读者写者问题”
2. 请使用信号量实现“哲学家就餐问题”
3. 利用信号的方式，父子进程通过消息队列/共享内存进行数据交换；
4. 利用管道的方式，父子进程对消息队列进行/共享内存数据交换。

3.7 实验七 线程管理

一、实验目的

(1) 加深对线程概念的理解，明确线程与进程的关系与区别。

(2) 掌握多线程并发程序设计相关系统调用的使用。

(3) 掌握多线程程序设计的基本方法。

(4) 熟练应用 Vi 编辑器

(5) 熟练应用 GCC 编译器

(6) 掌握 GDB 关于多线程程序的调试技术。

二、基础知识

1. 线程与进程的关系。

2. 线程控制相关系统调用：pthread_creat、pthread_join、pthread_exit 等。

3. 线程属性及基本操作。

三、实验内容

模拟实现 wc 功能

Unix 系统中的 wc 程序的作用是计算一个或多个文件中的行、单词以及字符个数。不过 wc 是一个典型的单线程程序。请设计一个多线程程序来计算并打印行数、单词数。

区分单词的规则可以参考：凡是一个非字母或数字的字符跟在字母或数字的后面，那么这个字母或数字就是单词的结尾。

要求：负责统计的每个线程设置自己的计数器，当线程处理完毕返回时，将自己的计数器值返还给主线程，由其将计数器值相加得到最后的结果。

3.8 实验八 线程同步

一、实验目的

(1) 加深对线程概念的理解，明确线程与进程的关系与区别。

(2) 掌握多线程并发程序同步机制。

(3) 熟练应用 Vi 编辑器

(4) 熟练应用 GCC 编译器

二、基础知识

1. 线程与进程的关系。

2. 线程控制相关系统调用：pthread_creat、pthread_join、pthread_exit 等。

3. 线程同步机制：互斥量、读写锁、条件变量

三、实验内容

1. 请使用线程同步机制实现“读者——写者”问题

2. 请使用线程同步机制实现“哲学家就餐”问题

第 4 章 并发程序综合设计

4.1 综合一 模拟实现 shell

Unix/Linux 操作系统中的 shell 是功能强大的命令解释器，本节将展示 Unix/Linux 的一个相对完善的 shell 子集。这个 shell 要支持：

(1) 简单的内置命令

(2) 命令执行

(3) 输入、输出重定向(<、>、|等)

(4) 作业控制

4.2 综合二 并发算法实现

实验项目：多进程(多线程)实现快速排序

在 Linux 系统环境下，编写一个多进程(多线程)进行快速排序的程序，使用的是有 1000，000 个随机数的文件。要求给出测试结果并对测试程序和结果做出说明。

实现提示：

(1) 每次数据分割后产生两个新的进程(线程)处理分割后的数据。

(2) 每个进程(线程)处理的数据小于 1000 以后不再分割

(3) 可以利用一些技巧使分割尽可能均匀

附录 1 常用 Linux 命令练习

(1) cd 命令将工作目录转换为/boot/grub,并用返回用户主目录。

提示：cd /boot/grub

cd

(3) 用 man 命令和--help 选项分别查看 ls 命令。

提示：man ls

ls –help

(4) 显示/root 文件夹下所有文件和目录（包括隐含文件和子文件夹下内容）

提示：ls –al

(5) 用 cat 命令显示/etc/passwd 文件，要求显示文件的每行必须有行号。

提示：cat /etc/passwd

(6) 用 more 和 less 命令显示/etc/passwd 文件，感受各种翻页命令。

(7) 使用 head 和 tail 显示/etc/passwd 文件的前 5 行和后 10 行，要求带有行号。

(8) 用命令清除当前终端内容。

提示：clear

(9) 用 wc 命令统计/etc/passwd 文件的行数。

(10) 用 cat 命令创建 f1 和 f2 文件，将 f1 和 f2 文件内容合并到 f3。

提示：**$** cat >f1

This is file1

$cat >f2

This is f2

$cat f1 f2 > f3

(11) 将 f3 文件中的内容合并到 f1，要求不能删除 f1 中原有的内容。

$cat f3 >> f1

(12) 利用管道统计/etc 文件夹下文件与子目录的个数并将统计结果放到 count 文件中。

提示：ls /etc | wc –l > count

$ ls /etc | wc -l > count

$ more count

226

(13)为 ls /root| wc –l 命令设置别名为 count，再次实现(12)的功能。

提示：alias 'count'='ls /etc | wc -l'

命令：

$ alias 'count'='ls /etc | wc -l'

$ count

226

(14) 将 f1 复制(已存在)，并命名为 fff，并比对 f1 和 fff 的内容。

提示：cp f1 fff

diff f1 fff

命令：

$ cp f1.txt fff

$ diff f1.txt fff

(15) 将当前目录及其子目录下所有扩展名是.c 的文件列出来。

提示：**find . –name “*.c”**

附录 2 GDB 调试部分参考代码

thread_debug.c

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

#define MAX_N 100000000
#define MAX_THREADS 100

int nthreads; //number of threads
int n;    //upper bound of rang in which of find primes
int prime[MAX_N+1]; ///in the end, prime[i] = 1 if i prime, eles 0
int nextbase;    //next sieve multiplier to be used

int work[MAX_THREADS];    //to measure how much work each threads does,
                                    //in terms of number of sieve multipliers checked

//lock index of the shared variable nextbase
pthread_mutex_t nextbaselock = PTHREAD_MUTEX_INITIALIZER;

//ID structs for the threads
pthread_t id[MAX_THREADS];

//""CROSSES OUT" all multiples of k, from k*k on
void crossout(int k)
{
        int i;
        for(i = k; i*k <= n; i++){
                prime[i*k] = 0;
        }

}
//worker thread routine
void *worker(int tn)    //tn is the thread number(0,1,...)
{
        int lim, base;
        // no need to check multipliers bigger than sqrt(n)
        lim = n;
```

```
        do{
                //get next sieve multiplier, avoiding duplication across threads
                pthread_mutex_lock(&nextbaselock);
                base = nextbase +=2;
                //pthread_mutex_unlock(&nextbaselock);
                if(base <= lim){
                        work[tn]++;    //log work done by this thread
                        //don't bother with crossing out if base is known to be
                        //composite
                        if(prime[base])
                                crossout(base);
                }else
                        return;
        }while(1);
}
int main(int argc, char *argv[])
{
        int nprimes; //number of primes found
        int totwork; //number of base values checked
        int i;
        void *p;

        n = atoi(argv[1]);
        nthreads = atoi(argv[2]);
        for(i = 2; i<=n; i++)
                prime[i] = 1;
        crossout(2);
        nextbase = 1;
        //get threads started
        for(i=0; i< nthreads; i++){
                pthread_create(&id[i],NULL,(void*)worker,(void*)i);
        }

        //wait for all done
        totwork = 0;
        for(i = 0; i < nthreads; i++){
                pthread_join(id[i],&p);
```

```
                printf("%d values of base done\n",work[i]);
                totwork += work[i];
        }
        printf("%d total values of base done\n",totwork);

        //report results
        nprimes = 0;
        for(i = 2; i <= 2; i++)
                if(prime[i])    nprimes ++;
        printf("the number of primes found was %d\n",nprimes);

}
```

附录 3 Vim 配置

在终端下使用 vim 进行编辑时，默认情况下，编辑的界面上是没有显示行号、语法高亮度显示、智能缩进等功能的。为了更好的在 vim 下进行工作，需要手动设置一个配置文件：.vimrc。

在启动 vim 时，当前用户根目录下的.vimrc 文件会被自动读取，该文件可以包含一些设置甚至脚本，所以，一般情况下把.vimrc 文件创建在当前用户的根目录下比较方便，即创建的命令为：

```
$vi ~/.vimrc
```

设置完后

```
$:x 或者 $wq
```

进行保存退出即可。

下面给出一个例子，其中列出了经常用到的设置，详细的设置信息请参照参考资料：

```
“双引号开始的行作为注释行，下同 。
-----------------------------------------------------------------------------------------------------------
“去掉讨厌的有关 vi 一致性模式，避免以前版本的一些 bug 和局限
set nocompatible
“显示行号
set nummber
“检测文件的类型
filetype on
“记录历史的行数
set history=1000
“背景使用黑色
set background=dark
“语法高亮度显示
syntax on
-----------------------------------------------------------------------------------------------------------
“下面在进行编写代码设置格式时，很有用。
“第一行，vim 使用自动对起，也就是把当前行的对起格式应用到下一行；
```

“第二行，依据上面的对起格式，智能的选择对起方式，对于类似 C 语言编写上很有用
set autoindent
set smartindent
“第一行设置 tab 键为 4 个空格，第二行设置当行之间交错时使用 4 个空格
set tabstop=4
set shiftwidth=4

“设置匹配模式，类似当输入一个左括号时会匹配相应的那个右括号
set showmatch

“去除 vim 的 GUI 版本中的 toolbar
set guioptions-=T

“当 vim 进行编辑时，如果命令错误，会发出一个响声，该设置去掉响声
set vb t_vb=

“在编辑过程中，在右下角显示光标位置的状态行
set ruler

“默认情况下，寻找匹配是高亮度显示的，该设置关闭高亮显示
set nohls

“查询时非常方便，如要查找 book 单词，当输入到/b 时，会自动找到第一 “个 b 开头的单词，当输入到/bo 时，会自动找到第一个 bo 开头的单词，依“次类推，进行查找时，使用此设置会快速找到答案，当你找要匹配的单词时，别忘记回车。
set incsearch

“修改一个文件后，自动进行备份，备份的文件名为原文件名加“~“后缀
if has(“vms”)
set nobackup
else
set backup
endif

如果去除注释后，一个完整的.vimrc 配置信息如下所示：
set nocompatible
set nummber
filetype on
set history=1000
set background=dark
syntax on
set autoindent

```
set smartindent
set tabstop=4
set shiftwidth=4
set showmatch
set guioptions-=T
set vb t_vb=
set ruler
set nohls
set incsearch
if has(“vms”)
set nobackup
else
set backup
endif
```

如果设置完后，发现功能没有起作用，检查一下系统下是否安装了 vim-enhanced 包，查询命令为：

```
$rpm –q vim-enhanced
```