第7章　UNIX进程的环境

7.1　引言

下一章将介绍进程控制原语，在此之前需先了解进程的环境。本章中将学习：当执行程序时，其main函数是如何被调用的，命令行参数是如何传送给执行程序的；典型的存储器布局是什么样式；如何分配另外的存储空间；进程如何使用环境变量；进程终止的不同方式等。另外，还将说明longjmp和setjmp函数以及它们与栈的交互作用。本章结束之前，还将查看进程的资源限制。

7.2　main函数

C程序总是从main函数开始执行。main函数的原型是：

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

其中，*argc*是命令行参数的数目，*argv*是指向参数的各个指针所构成的数组。7.4节将对命令行参数进行说明。

当内核起动C程序时(使用一个exec函数，8.9节将说明exec函数)，在调用main前先调用一个特殊的起动例程。可执行程序文件将此起动例程指定为程序的起始地址——这是由连接编辑程序设置的，而连接编辑程序则由C编译程序(通常是cc)调用。起动例程从内核取得命令行参数和环境变量值，然后为调用main函数作好安排。

7.3　进程终止

有五种方式使进程终止：

(1) 正常终止：
  (a) 从main返回。
  (b) 调用exit。
  (c) 调用_exit。

(2) 异常终止：
  (a) 调用abort(见第10章)。
  (b) 由一个信号终止(见第10章)。

上节提及的起动例程是这样编写的，使得从main返回后立即调用exit函数。如果将起动例程以C代码形式表示(实际上该例程常常用汇编语言编写)，则它调用main函数的形式可能是：

```
exit( main(argc, argv) );
```

7.3.1　exit和_exit函数

exit和_exit函数用于正常终止一个程序：_exit立即进入内核，exit则先执行一些清除处理(包括调用执行各终止处理程序，关闭所有标准I/O流等)，然后进入内核。

```
#include <stdlib.h>

void exit(int status);

#include <unistd.h>

void _exit (int status);
```

我们将在8.5节中讨论这两个函数对其他进程，例如终止进程的父、子进程的影响。

> 使用不同头文件的原因是：exit是由ANSI C说明的，而_exit则是由POSIX.1说明的。

由于历史原因， exit函数总是执行一个标准 I/O库的清除关闭操作：对于所有打开流调用fclose函数。回忆5.5节，这造成缓存中的所有数据都被刷新(写到文件上)。

exit和_exit都带一个整型参数，称之为终止状态（exit status）。大多数UNIX shell都提供检查一个进程终止状态的方法。如果 (a)若调用这些函数时不带终止状态，或 (b)main执行了一个无返回值的return语句，或(c)main执行隐式返回，则该进程的终止状态是未定义的。这就意味着，下列经典性的C语言程序：

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

是不完整的，因为main函数没有使用return语句返回(隐式返回)，它在返回到C的起动例程时并没有返回一个值(终止状态)。另外，若使用：

```
return(0);
```

或者

```
exit(0);
```

则向执行此程序的进程 (常常是一个shell进程)返回终止状态0。另外， main函数的说明实际上应当是：

```
int main(void)
```

下一章将了解进程如何使程序执行，如何等待执行该程序的进程完成，然后取得其终止状态。

> 将main说明为返回一个整型以及用exit代替return，对某些C编译程序和UNIX lint(1)程序而言会产生不必要的警告信息，因为这些编译程序并不了解 main中的exit与return语句的作用相同。警告信息可能是“ control reaches end of nonvoid function（控制到达非void函数的结束处）”。避开这种警告信息的一种方法是：在main中使用return语句而不是exit。但是这样做的结果是不能用UNIX的grep公用程序来找出程序中所有的exit调用。另外一个解决方法是将main说明为返回void而不是int，然后仍旧调用 exit。这也避开了编译程序的警告，但从程序设计角度看却并不正确。本章将main表示为返回一个整型，因为这是 ANSI C和POSIX.1所定义的。我们将不理会编译程序不必要的警告。

7.3.2 atexit函数

按照ANSI C的规定，一个进程可以登记多至32个函数，这些函数将由exit自动调用。我们称这些函数为终止处理程序（exit handler），并用atexit函数来登记这些函数。

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

返回：若成功则为0，若出错则为非0

其中，atexit的参数是一个函数地址，当调用此函数时无需向它传送任何参数，也不期望它返回一个值。exit以登记这些函数的相反顺序调用它们。同一函数如若登记多次，则也被调用多次。

> 终止处理程序这一机制由ANSI C最新引进。SVR4和4.3+BSD都提供这种机制。系统V的早期版本和4.3BSD则都不提供此机制。

根据ANSI C和POSIX.1，exit首先调用各终止处理程序，然后按需多次调用fclose，关闭所有打开流。图7-1显示了一个C程序是如何起动的，以及它终止的各种方式。

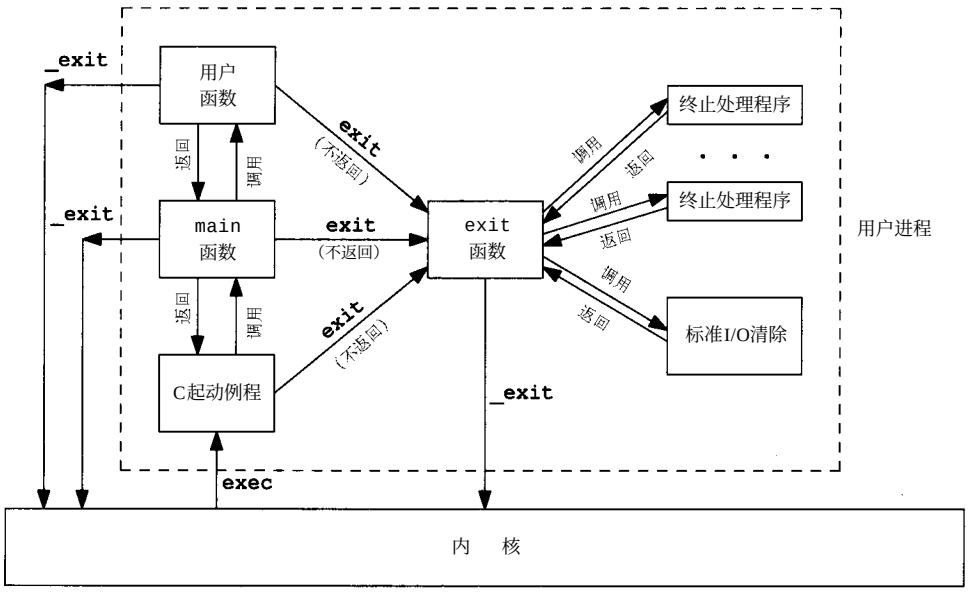

图7-1 一个C程序是如何起动和终止的

注意，内核使程序执行的唯一方法是调用一个exec函数。进程自愿终止的唯一方法是显式或隐式地(调用exit)调用_exit。进程也可非自愿地由一个信号使其终止(图7-1中没有显示)。

实例

程序7-1说明了如何使用atexit函数。执行程序7-1产生：

```
$ a.out
main is done
```

```
first exit handler
first exit handler
second exit handler
```

注意，在main中没有调用exit，而是用了return语句。

程序7-1 终止处理程序实例

```
#include    "ourhdr.h"

static void my_exit1(void), my_exit2(void);

int
main(void)
{
    if (atexit(my_exit2) != 0)
        err_sys("can't register my_exit2");

    if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");
    if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");

    printf("main is done\n");
    return(0);
}

static void
my_exit1(void)
{
    printf("first exit handler\n");
}

static void
my_exit2(void)
{
    printf("second exit handler\n");
}
```

7.4 命令行参数

当执行一个程序时，调用exec的进程可将命令行参数传递给该新程序。这是UNIX shell的一部分常规操作。在前几章的很多实例中，我们已经看到了这一点。

实例

程序7-2将其所有命令行参数都回送到标准输出上 (UNIX echo(1)程序不回送第0个参数)。编译此程序，并将其可执行代码文件定名为echoarg，则：

```
$ ./echoarg arg1 TEST foo
argv[0]: ./echoarg
argv[1]: arg1
argv[2]: TEST
argv[3]: foo
```

程序7-2 将所有命令行参数回送到标准输出

```
#include    "ourhdr.h"

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

```
{
    int     i;

    for (i = 0; i < argc; i++)      /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);
    exit(0);
}
```

ANSI C和POSIX.1都要求argv [argc] 是一个空指针。这就使我们可以将参数处理循环改写为：

```
for(i = 0; argv[i] != NULL; i++)
```

7.5 环境表

每个程序都接收到一张环境表。与参数表一样，环境表也是一个字符指针数组，其中每个指针包含一个以null结束的字符串的地址。全局变量environ则包含了该指针数组的地址。

```
extern char **environ;
```

例如：如果该环境包含五个字符串，那么它看起来可能如图 7-2中所示。

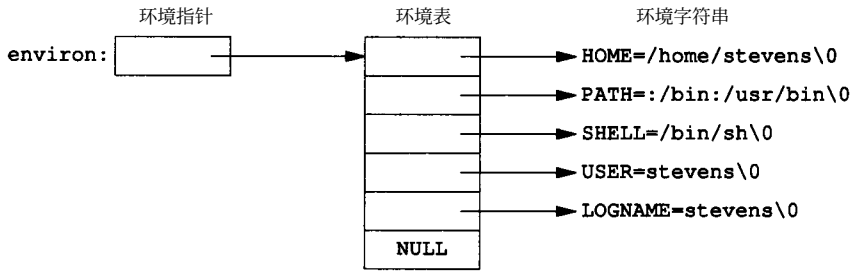

图7-2 由五个字符串组成的环境

其中，每个字符串的结束处都有一个null字符。我们称environ为环境指针，指针数组为环境表，其中各指针指向的字符串为环境字符串。

按照惯例，环境由：

name=value

这样的字符串组成，这与图 7-2中所示相同。大多数预定义名完全由大写字母组成，但这只是一个惯例。

在历史上，大多数UNIX系统对main函数提供了第三个参数，它就是环境表地址：

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

因为ANSI C规定main函数只有两个参数，而且第三个参数与全局变量 environ相比也没有带来更多益处，所以POSIX.1也规定应使用environ而不使用第三个参数。通常用 getenv和putenv函数(7.9节将说明)来存取特定的环境变量，而不是用 environ变量。但是，如果要查看整个环境，则必须使用environ指针。

7.6 C程序的存储空间布局

由于历史原因，C程序一直由下列几部分组成：

- 正文段。这是由CPU执行的机器指令部分。通常，正文段是可共享的，所以即使是经常

执行的程序(如文本编辑程序、C编译程序、shell等)在存储器中也只需有一个副本，另外，正文段常常是只读的，以防止程序由于意外事故而修改其自身的指令。

• 初始化数据段。通常将此段称为数据段，它包含了程序中需赋初值的变量。例如， C程序中任何函数之外的说明：

```
int     maxcount = 99;
```

使此变量以初值存放在初始化数据段中。

• 非初始化数据段。通常将此段称为bss段，这一名称来源于早期汇编程序的一个操作符，意思是“block started by symbol（由符号开始的块）”，在程序开始执行之前，内核将此段初始化为0。函数外的说明：

```
long    sum[1000] ;
```

使此变量存放在非初始化数据段中。

• 栈。自动变量以及每次函数调用时所需保存的信息都存放在此段中。每次函数调用时，其返回地址、以及调用者的环境信息（例如某些机器寄存器）都存放在栈中。然后，新被调用的函数在栈上为其自动和临时变量分配存储空间。通过以这种方式使用栈， C函数可以递归调用。

• 堆。通常在堆中进行动态存储分配。由于历史上形成的惯例，堆位于非初始化数据段顶和栈底之间。

图7-3显示了这些段的一种典型安排方式。这是程序的逻辑布局——虽然并不要求一个具体实现一定以这种方式安排其存储空间。尽管如此，这给出了一个我们便于作有关说明的一种典型安排。

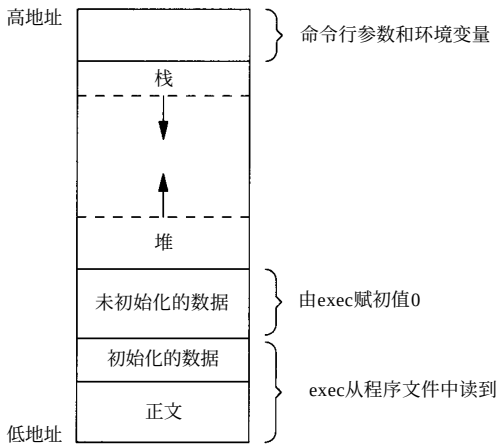

图7-3 典型的存储器安排

对于VAX上的4.3+BSD，正文段从0位置开始，栈顶则在0x7fffffff之下开始。在VAX机器上，堆顶和栈底之间未用的虚地址空间很大。

从图7-3还可注意到未初始化数据段的内容并不存放在磁盘程序文件中。需要存放在磁盘程序文件中的段只有正文段和初始化数据段。

size(1)命令报告正文段、数据段和bss段的长度(单位：字节)。例如：

```
$ size /bin/cc /bin/sh
text    data    bss     dec     hex
81920   16384   664     98968   18298   /bin/cc
90112   16384   0       106496  1a000   /bin/sh
```

第4和第5列是分别以十进制和十六进制表示的总长度。

7.7 共享库

现在，很多UNIX系统支持共享库。Arnold [1986] 说明了系统V上共享库的一个早期实现，Gingell等 [1987] 则说明了SunOS上的另一个实现。共享库使得可执行文件中不再需要包含常用的库函数，而只需在所有进程都可存取的存储区中保存这种库例程的一个副本。程序第一次执行或

者第一次调用某个库函数时，用动态连接方法将程序与共享库函数相连接。这减少了每个可执行文件的长度，但增加了一些运行时间开销。共享库的另一个优点是可以用库函数的新版本代替老版本而无需对使用该库的程序重新连接编辑。(假定参数的数目和类型都没有发生改变。)

不同的系统使用不同的方法使说明程序是否要使用共享库。比较典型的有 cc(1)和ld(1)命令的可选项。作为长度方面发生变化的例子，下列可执行文件 (典型的hello.c程序)先用无共享库方式创建：

```
$ ls -1 a.out
-rwxrwxr-x   1 stevens      104859 Aug  :2 a.out
$ size a.out
text    data    bss     dec     hex
49152   49152   0       98304   18000
```

如果我们再编译此程序使其使用共享库，则可执行文件的正文和数据段的长度都显著减小：

```
$ ls -1 a.out
-rwxrwxr-x   1 stevens      24576 Aug  :2 a.out
$ size a.out
text    data    bss     dec     hex
8192    8192    0       16384   4000
```

7.8 存储器分配

ANSI C说明了三个用于存储空间动态分配的函数。

(1) malloc。分配指定字节数的存储区。此存储区中的初始值不确定。

(2) calloc。为指定长度的对象，分配能容纳其指定个数的存储空间。该空间中的每一位(bit)都初始化为0。

(3) realloc。更改以前分配区的长度(增加或减少)。当增加长度时，可能需将以前分配区的内容移到另一个足够大的区域，而新增区域内的初始值则不确定。

```
#include <stdlib.h>

void *malloc(size_t size);

void *calloc(size_t nobj, size_t size);

void *realloc(void *ptr, size_t newsize);

                                三个函数返回：若成功则为非空指针，若出错则为 NULL

void free(void *ptr)
```

这三个分配函数所返回的指针一定是适当对齐的，使其可用于任何数据对象。例如，在一个特定的系统上，如果最苛刻的对齐要求是 double，则对齐必须在8的倍数的地址单元处，那么这三个函数返回的指针都应这样对齐。

回忆1.6节中对类属void *指针和函数原型的讨论。因为这三个 alloc函数都返回类属指针，如果在程序中包括了<stdlib.h>(包含了函数原型)，那么当我们将这些函数返回的指针赋与一个不同类型的指针时，不需要作类型强制转换。

函数free释放*ptr*指向的存储空间。被释放的空间通常被送入可用存储区池，以后可在调用

分配函数时再分配。

realloc使我们可以增、减以前分配区的长度 (最常见的用法是增加该区)。例如，如果先分配一个可容纳长度为 512的数组的空间，并在运行时填充它，但又发现空间不够，则可调用realloc扩充该存储空间。如果在该存储区后有足够的空间可供扩充，则可在原存储区位置上向高地址方向扩充，并返回传送给它的同样的指针值。如果在原存储区后没有足够的空间，则realloc分配另一个足够大的存储区，将现存的 512个元素数组的内容复制到新分配的存储区。因为这种存储区可能会移动位置，所以不应当使用任何指针指在该区中。习题 4.18显示了在getcwd中如何使用realloc，以处理任何长度的路径名。程序 15-27是使用realloc的另一个例子，用其可以避免使用编译时固定长度的数组。

注意，realloc的最后一个参数是存储区的*newsize*(新长度)，不是新、旧长度之差。作为一个特例，若*ptr*是一个空指针，则realloc的功能与malloc相同，用于分配一个指定长度*newsize*的存储区。

> 此功能是由ANSI C新引进的。如果传送一个 null指针，realloc的早期版本会失败。早期版本允许realloc自上次malloc, realloc或calloc以来所释放的块。这种技巧可回溯到V7，它利用malloc实现存储器紧缩的搜索策略。 4.3+BSD仍支持这一功能，而SVR4则不支持。这种功能不应再使用。

这些分配例程通常通过sbrk(2)系统调用实现。该系统调用扩充(或缩小)进程的堆(见图7-3)。malloc和free的一个样本实现请见Kernighan和Ritchie [1988] 的8.7节。

虽然sbrk可以扩充或缩小一个进程的存储空间，但是大多数 malloc和free的实现都不减小进程的存储空间。释放的空间可供以后再分配，但将它们保持在 malloc池中而不返回给内核。

应当注意的是，大多数实现所分配的存储空间比所要求的要稍大一些，额外的空间用来记录管理信息——分配块的长度，指向下一个分配块的指针等等。这就意味着如果写过一个已分配区的尾端，则会改写后一块的管理信息。这种类型的错误是灾难性的，但是因为这种错误不会很快就暴露出来，所以也就很难发现。将指向分配块的指针向后移动也可能会改写本块的管理信息。

其他可能产生的致命性的错误是：释放一个已经释放了的块；调用 free时所用的指针不是三个alloc函数的返回值等。

因为存储器分配出错很难跟踪，所以某些系统提供了这些函数的另一种实现方法。每次调用这三个分配函数中的任意一个或 free时都进行附加的出错检验。在调用连接编辑程序时指定一个专用库，则在程序中就可使用这种版本的函数。此外还有公共可用的资源（例如由4.3+BSD所提供的），在对其进行编译时使用一个特殊标志就会使附加的运行时间检查生效。

因为存储空间分配程序的操作对某些应用程序的运行时间性能非常重要，所以某些系统提供了附加能力。例如， SVR4提供了名为mallopt的函数，它使进程可以设置一些变量，并用它们来控制存储空间分配程序的操作。还可使用另一个名为 mallinfo的函数，以对存储空间分配程序的操作进行统计。请查看所使用系统的 malloc(3)手册页，弄清楚这些功能是否可用。

alloca函数

还有一个函数也值得一提，这就是 alloca。其调用序列与malloc相同，但是它是在当前函数的栈帧上分配存储空间，而不是在堆中。其优点是：当函数返回时，自动释放它所使用的

栈帧，所以不必再为释放空间而费心。其缺点是：某些系统在函数已被调用后不能增加栈帧长度，于是也就不能支持alloca函数。尽管如此，很多软件包还是使用alloca函数，也有很多系统支持它。

7.9 环境变量

如同前述，环境字符串的形式是：

name=value

UNIX内核并不关心这种字符串的意义，它们的解释完全取决于各个应用程序。例如， shell使用了大量的环境变量。其中某一些在登录时自动设置（如HOME，USER等），有些则由用户设置。我们通常在一个shell起动文件中设置环境变量以控制shell的动作。例如，若设置了环境变量MAILPATH，则它告诉Bourne shell和KornShell到哪里去查看邮件。

ANSI C定义了一个函数getenv，可以用其取环境变量值，但是该标准又称环境的内容是由实现定义的。

```
#include <stdlib.h>

char *getenv(const char *name);
```

返回：指向与*name*关联的*value*的指针，若未找到则为NULL

注意，此函数返回一个指针，它指向*name=value*字符串中的*value*。我们应当使用getenv从环境中取一个环境变量的值，而不是直接存取environ。

POSIX.1和XPG3定义了某些环境变量。表 7-1列出了由这两个标准定义并受到 SVR4和4.3+BSD支持的环境变量。SVR4和4.3+BSD还使用了很多依赖于实现的环境变量。

FIPS 151-1要求登录shell必须要定义环境变量HOME和LOGNAME。

表7-1 环境变量

变 量	标 准		实 现		说 明
	POSIX.1	XPG3	SVR4	4.3+BSD	
HOME	•	•	•	•	起始目录
LANG	•	•	•		本地名
LC_ALL	•	•	•		本地名
LC_COLLATE	•	•	•		本地排序名
LC_CTYPE	•	•	•		本地字符分类名
LC_MONETARY	•	•	•		本地货币编辑名
LC_NUMERIC	•	•	•		本地数字编辑名
LC_TIME	•	•	•		本地日期/时间格式名
LOGNAME	•	•	•	•	登录名
NLSPATH		•	•		消息类模板序列
PATH	•	•	•	•	搜索可执行文件的路径前缀表
TERM	•	•	•	•	终端类型
TZ	•	•	•	•	时区信息

除了取环境变量值，有时也需要设置环境变量，或者是改变现有变量的值，或者是增加新的环境变量。（在下一章将会了解到，我们能影响的是当前进程及其后生成的子进程的环境，但不能影响父进程的环境，这通常是一个shell进程。尽管如此，修改环境表的能力仍然是很有用的。）不幸的是，并不是所有系统都支持这种能力。表7-2列出了由不同的标准及实现支持的各种函数。

表7-2 对于各种环境表函数的支持

函 数	标 准			实 现	
	ANSI C	POSIX.1	XPG3	SVR4	4.3+BSD
getenv	•	•	•	•	•
putenv		（可能）	•	•	•
setenv					•
unsetenv					•
clearenv		（可能）			

POSIX.1标准说明putenv和clearenv正被考虑加到POSIX.1的修订版中。

在表7-2中，中间三个函数的原型是：

```
#include <stdlib.h>

int putenv(const char *str);

int setenv(const char *name, const char *value, int rewrite);
```

两个函数返回：若成功则为0，若出错则为非0

```
void unsetenv(const char *name);
```

这三个函数的操作是：

• putenv取形式为*name*=*value*的字符串，将其放到环境表中。如果*name*已经存在，则先删除其原来的定义。

• setenv将*name*设置为*value*。如果在环境中*name*已经存在，那么(a)若*rewrite*非0，则首先删除其现存的定义；(b)若*rewrite*为0，则不删除其现存定义（*name*不设置为新的*value*，而且也不出错）。

• unsetenv删除*name*的定义。即使不存在这种定义也不算出错。

这些函数在修改环境表时是如何进行操作的呢？回忆一下图 7-3，其中，环境表（指向实际*name*=*value*字符串的指针数组）和环境字符串典型地存放在进程存储空间的顶部（栈之上）。删除一个字符串很简单——只要先找到该指针，然后将所有后续指针都向下移一个位置。但是增加一个字符串或修改一个现存的字符串就比较困难。栈以上的空间因为已处于进程存储空间的顶部，所以无法扩充，即无法向上扩充，也无法向下扩充。

(1) 如果修改一个现存的*name*:

(a) 如果新*value*的长度少于或等于现存*value*的长度，则只要在原字符串所用空间中写入新字符串。

(b) 如果新*value*的长度大于原长度，则必须调用malloc为新字符串分配空间，然后将新字符写入该空间中，然后使环境表中针对*name*的指针指向新分配区。

(2) 如果要增加一个新的*name*，则操作就更加复杂。首先，调用malloc为*name=value*分配空间，然后将该字符串写入此空间中。

(a) 然后，如果这是第一次增加一个新*name*，则必须调用malloc为新的指针表分配空间。将原来的环境表复制到新分配区，并将指向新*name=value*的指针存在该指针表的表尾，然后又将一个空指针存在其后。最后使environ指向新指针表。再看一下图7-3，如果原来的环境表位于栈顶之上(这是一种常见情况)，那么必须将此表移至堆中。但是，此表中的大多数指针仍指向栈顶之上的各*name=value*字符串。

(b) 如果这不是第一次增加一个新*name*，则可知以前已调用malloc在堆中为环境表分配了空间，所以只要调用realloc，以分配比原空间多存放一个指针的空间。然后将该指向新*name=value*字符串的指针存放在该表表尾，后面跟着一个空指针。

7.10 setjmp和longjmp函数

在C中，不允许使用跳越函数的goto语句。而执行这种跳转功能的是函数setjmp和longjmp。这两个函数对于处理发生在很深的嵌套函数调用中的出错情况非常有用。

考虑一下程序7-3的骨干部分。其主循环是从标准输入读1行，然后调用do_line处理每一输入行。该函数然后调用get_token从该输入行中取下一个记号。一行中的第一个记号假定是某种形式的一条命令，于是switch语句就实现命令选择。我们的程序只处理一条命令，对此命令调用cmd_add函数。

程序7-3 进行命令处理的典型程序的骨干部分

```
#include    "ourhdr.h"

#define TOK_ADD    5

void    do_line(char *);
void    cmd_add(void);
int     get_token(void);

int
main(void)
{
    char    line[MAXLINE];

    while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
    exit(0);
}

char    *tok_ptr;       /* global pointer for get_token() */

void
do_line(char *ptr)      /* process one line of input */
{
    int     cmd;

    tok_ptr = ptr;
    while ( (cmd = get_token()) > 0) {
        switch (cmd) {  /* one case for each command */
        case TOK_ADD:
                cmd_add();
```

```
                break;
            }
        }
    }

    void
    cmd_add(void)
    {
        int     token;

        token = get_token();
        /* rest of processing for this command */
    }

    int
    get_token(void)
    {
        /* fetch next token from line pointed to by tok_ptr */
    }
```

程序7-3在读命令、确定命令的类型、然后调用相应函数处理每一条命令这类程序中是非常典型的。图 7-4显示了调用了cmd_add之后栈的大致使用情况。

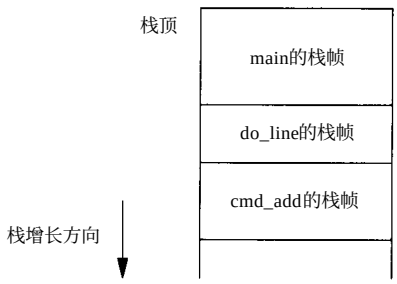

图7-4 调用cmd_add后的各个栈帧

自动变量的存储单元在每个函数的栈帧中。数组 line在main的栈帧中，整型 cmd在do_line的栈帧中，整型token在cmd_add的栈帧中。

如上所述，这种形式的栈安排是非常典型的，但并不要求非如此不可。栈并不一定要向低地址方向扩充。某些系统对栈并没有提供特殊的硬件支持，此时一个 C实现可能要用连接表实现栈帧。

在编写如程序7-3这样的程序中经常会遇到的一个问题是，如何处理非致命性的错误。例如，若cmd_add函数发现一个错误，譬如说一个无效的数，那么它可能先打印一个出错消息，然后希望忽略输入行的余下部分，返回main函数并读下一输入行。用C语言比较难做到这一点。(在本例中，cmd_add函数只比main低两个层次，在有些程序中往往低五或更多层次。）如果我们不得不以检查返回值的方法逐层返回，那就会变得很麻烦。

解决这种问题的方法就是使用非局部跳转——setjmp和longjmp函数。非局部表示这不是在一个函数内的普通的C语言goto语句，而是在栈上跳过若干调用帧，返回到当前函数调用路径上的一个函数中。

```
#include <setjmp.h>

int setjmp(jmp_buf env);
```

返回：若直接调用则为0，若从longjmp返回则为非0

```
void longjmp(jmp_buf env, int val);
```

在希望返回到的位置调用setjmp，在本例中，此位置在main函数中。因为我们直接调用该函数，所以其返回值为0。setjmp的参数*env*是一个特殊类型jmp_buf。这一数据类型是某种形式的数组，其中存放在调用longjmp时能用来恢复栈状态的所有信息。一般，*env*变量是个全局变量，因为需从另一个函数中引用它。

当检查到一个错误时，例如在cmd_add函数中，则以两个参数调用longjmp函数。第一个就是在调用setjmp时所用的*env*；第二个*val*，是个非0值，它成为从setjmp处返回的值。使用第二个参数的原因是对于一个 setjmp可以有多个longjmp。例如，可以在cmd_add中以*val*为1调用longjmp，也可在get_token中以*val*为2调用longjmp。在main函数中，setjmp的返回值就会是1或2，通过测试返回值就可判断是从cmd_add还是从get_token来的longjmp。

再回到程序实例中，表 7-1中给出了经修改过后的 main和cmd_add函数(其他两个函数，do_line和get_token未经更改)。

程序7-4　setjmp和longjmp实例

```
#include    <setjmp.h>
#include    "ourhdr.h"

#define TOK_ADD    5

jmp_buf jmpbuffer;

int
main(void)
{
    char    line[MAXLINE];

    if (setjmp(jmpbuffer) != 0)
        printf("error");
    while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
    exit(0);
}

 . . .

void
cmd_add(void)
{
    int     token;

    token = get_token();
    if (token < 0)      /* an error has occurred */
        longjmp(jmpbuffer, 1);
    /* rest of processing for this command */
}
```

执行main时，调用 setjmp，它将所需的信息记入变量 jmpbuffer中并返回 0。然后调用do_line，它又调用cm_add，假定在其中检测到一个错误。在 cmd_add中调用longjmp之前，栈的形式如图 7-4中所示。但是 longjmp使栈反绕到执行 main函数时的情况，也就是抛弃了cmd_add和do_line的栈帧。调用longjmp造成main中setjmp的返回，但是，这一次的返回值是1(longjmp的第二个参数)。

7.10.1 自动、寄存器和易失变量

下一个问题是：“在main函数中，自动变量和寄存器变量的状态如何？”当longjmp返回到main函数时，这些变量的值是否能恢复到以前调用 setjmp时的值（即滚回原先值），或者这些变量的值保持为调用do_line时的值(do_line调用cmd_add，cmd_add又调用longjmp)?不幸的是，对此问题的回答是“看情况”。大多数实现并不滚回这些自动变量和寄存器变量的值，而所有标准则说它们的值是不确定的。如果你有一个自动变量，而又不想使其值滚回，则可定义其为

具有volatile属性。说明为全局和静态变量的值在执行longjmp时保持不变。

实例

下面我们通过程序7-5说明在调用longjmp后，自动变量、寄存器变量和易失变量的不同情况。如果以不带优化和带优化对此程序分别进行编译，然后运行它们，则得到的结果是不同的：

```
$ cc testjmp.c              不进行任何优化的编译
$ a.out
in f1(): count = 97, val = 98, sum = 99
after longjmp: count = 97, val = 98, sum = 99
$ cc -O testjmp.c           进行全部优化的编译
$ a.out
in f1(): count = 97, val = 98, sum = 99
after longjmp: count = 2, val = 3, sum = 99
```

注意，易失变量(sum)不受优化的影响，在longjmp之后的值，是它在调用f1时的值。在我们所使用的setjmp(3)手册页上说明存放在存储器中的变量将具有 longjmp时的值，而在CPU和浮点寄存器中的变量则恢复为调用 setjmp时的值。这确实就是在运行程序 7-5时所观察到的值。不进行优化时，所有这三个变量都存放在存储器中 (亦即对val的寄存器存储类被忽略)。而进行优化时，count和val都存放在寄存器中(即使count并未说明为register),volatile变量则仍存放在存储器中。通过这一例子要理解的是，如果要编写一个使用非局部跳转的可移植程序，则必须使用volatile属性。

程序7-5 longjmp对自动，寄存器和易失变量的影响

```
#include    <setjmp.h>
#include    "ourhdr.h"

static void f1(int, int, int);
static void f2(void);

static jmp_buf  jmpbuffer;

int
main(void)
{
    int             count;
    register int    val;
    volatile int    sum;

    count = 2; val = 3; sum = 4;
    if (setjmp(jmpbuffer) != 0) {
        printf("after longjmp: count = %d, val = %d, sum = %d\n",
                count, val, sum);
        exit(0);
    }
    count = 97; val = 98; sum = 99;
                /* changed after setjmp, before longjmp */
    f1(count, val, sum);        /* never returns */
}

static void
f1(int i, int j, int k)
{
    printf("in f1(): count = %d, val = %d, sum = %d\n", i, j, k);
    f2();
}
```

```
static void
f2(void)
{
    longjmp(jmpbuffer, 1);
}
```

第10章讨论信号处理程序及sigsetjmp和siglongjmp时，将再次使用setjmp和longjmp函数。

7.10.2 自动变量的潜在问题

前面已经说明了处理栈帧的一般方式，与此相关我们现在可以分析一下自动变量的一个潜在出错情况。基本规则是说明自动变量的函数已经返回后，就不能再引用这些自动变量。在整个UNIX手册中，关于这一点有很多警告。

程序7-6是一个名为open_data的函数，它打开了一个标准I/O流，然后为该流设置缓存。

程序7-6 自动变量的不正确使用

```
#include    <stdio.h>

#define DATAFILE    "datafile"

FILE *
open_data(void)
{
    FILE    *fp;
    char    databuf[BUFSIZ];  /* setvbuf makes this the stdio buffer */

    if ( (fp = fopen(DATAFILE, "r")) == NULL)
        return(NULL);

    if (setvbuf(fp, databuf, _IOLBF, BUFSIZ) != 0)
        return(NULL);

    return(fp);     /* error */
}
```

问题是：当open_data返回时，它在栈上所使用的空间将由下一个被调用函数的栈帧使用。但是，标准I/O库函数仍将使用原先为databuf在栈上分配的存储空间作为该流的缓存。这就产生了冲突和混乱。为了改正这一问题，应在全局存储空间静态地 (如static或extern)，或者动态地(使用一种alloc函数)为数组databuf分配空间。

7.11 getrlimit和setrlimit函数

每个进程都有一组资源限制，其中某一些可以用getrlimit和setrlimit函数查询和更改。

```
#include <sys/time.h>
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlptr);

int setrlimit(int resource, const struct rlimit *rlptr);
```

两个函数返回：若成功则为0，若出错则为非0

对这两个函数的每一次调用都指定一个资源以及一个指向下列结构的指针。

```
struct rlimit {
  rlim_t rlim_cur;   /* soft limit: current limit */
  rlim_t rlim_max;   /* hard limit: maximum value for rlim_cur */
};
```

这两个函数不属于POSIX.1，但SVR4和4.3+BSD提供它们。

SVR4在上面的结构中使用基本系统数据类型 rlim_t，其他系统则将这两个成员定义为整型或长整型。

进程的资源限制通常是在系统初始化时由0进程建立的，然后由后续进程继承。在SVR4中，系统默认值可以查看文件/etc/conf/cf.d/mtune。在4.3+BSD中，系统默认值分散在多个头文件中。

在更改资源限制时，须遵循下列三条规则：

(1) 任何一个进程都可将一个软限制更改为小于或等于其硬限制。

(2) 任何一个进程都可降低其硬限制值，但它必须大于或等于其软限制值。这种降低，对普通用户而言是不可逆反的。

(3) 只有超级用户可以提高硬限制。

一个无限量的限制由常数RLIM_INFINITY指定。

这两个函数的*resource*参数取下列值之一。注意并非所有资源限制都受到 SVR4和4.3+BSD的支持。

• RLIMIT_CORE （SVR4及4.3+BSD）core文件的最大字节数，若其值为 0则阻止创建core文件。

• RLIMIT_CPU （SVR4及4.3+BSD）CPU时间的最大量值(秒)，当超过此软限制时，向该进程发送SIGXCPU信号。

• RLIMIT_DATA （SVR4及4.3+BSD）数据段的最大字节长度。这是图7-3中初始化数据、非初始化数据以及堆的总和。

• RLIMIT_FSIZE （SVR4及4.3+BSD）可以创建的文件的最大字节长度。当超过此软限制时，则向该进程发送SIGXFSZ信号。

• RLIMIT_MEMLOCK （4.3+BSD）锁定在存储器地址空间(尚未实现)。

• RLIMIT_NOFILE （SVR4）每个进程能打开的最多文件数。更改此限制将影响到sysconf函数在参数_SC_OPEN_MAX中返回的值（见2.5.4节）。见程序2-3。

• RLIMIT_NPROC （4.3+BSD）每个实际用户ID所拥有的最大子进程数。更改此限制将影响到sysconf函数在参数_SC_CHILD_MAX中返回的值（见2.5.4节）。

• RLIMIT_OFILE （4.3+BSD）与SVR4的RLIMIT_NOFILE相同。

• RLIMIT_RSS （4.3+BSD）最大驻内存集字节长度（RSS）。如果物理存储器供不应求，则内核将从进程处取回超过RSS的部分。

• RLIMIT_STACK （SVR4及4.3+BSD）栈的最大字节长度。见图7-3。

• RLIMIT_VMEM （SVR4）可映照地址空间的最大字节长度。这影响到 mmap函数（见12.9节）。

资源限制影响到调用进程并由其子进程继承。这就意味着为了影响一个用户的所有后续进程，需将资源限制设置构造在 shell之中。确实， Bourne shell和KornShell具有内部ulimit命令，C shell具有内部limit命令。(umask和chdir函数也必须是shell内部的。)

早期的Bourne shell，例如由伯克利提供的一种，不支持ulimit命令。

较新的KornShell的ulimit命令具有-H和-S选择项，分别检查和修改硬或软限制，但它们尚未编写入文档。

实例

程序7-7打印由系统支持的对所有资源的当前软限制和硬限制。为了在 SVR4和4.3+BSD之下运行此程序，必须条件编译不同的资源名。

程序7-7 打印当前资源限制

```
#include    <sys/types.h>
#include    <sys/time.h>
#include    <sys/resource.h>
#include    "ourhdr.h"

#define doit(name)  pr_limits(#name, name)

static void pr_limits(char *, int);

int
main(void)
{
    doit(RLIMIT_CORE);
    doit(RLIMIT_CPU);
    doit(RLIMIT_DATA);
    doit(RLIMIT_FSIZE);

#ifdef  RLIMIT_MEMLOCK
    doit(RLIMIT_MEMLOCK);
#endif
#ifdef  RLIMIT_NOFILE    /* SVR4 name */
    doit(RLIMIT_NOFILE);
#endif
#ifdef  RLIMIT_OFILE     /* 4.3+BSD name */
    doit(RLIMIT_OFILE);
#endif
#ifdef  RLIMIT_NPROC
    doit(RLIMIT_NPROC);
#endif
#ifdef  RLIMIT_RSS
    doit(RLIMIT_RSS);
#endif
    doit(RLIMIT_STACK);
#ifdef  RLIMIT_VMEM
    doit(RLIMIT_VMEM);
#endif
    exit(0);
}

static void
pr_limits(char *name, int resource)
{
    struct rlimit   limit;

    if (getrlimit(resource, &limit) < 0)
        err_sys("getrlimit error for %s", name);
    printf("%-14s  ", name);
    if (limit.rlim_cur == RLIM_INFINITY)
        printf("(infinite)  ");
```

```
        else
            printf("%10ld  ", limit.rlim_cur);
        if (limit.rlim_max == RLIM_INFINITY)
            printf("(infinite)\n");
        else
            printf("%10ld\n", limit.rlim_max);
    }
```

注意，在doit宏中使用了新的ANSI C字符串创建算符(#)，以便为每个资源名产生字符串值。例如：

```
doit(RLIMIT_CORE);
```

这将由C预处理程序扩展为：

```
pr_limits("RLIMIT_CORE",  RLIMIT_CORE);
```

在SVR4下运行此程序，得到：

```
$ a.out
RLIMIT_CORE          1048576       1048576
RLIMIT_CPU         (infinite)    (infinite)
RLIMIT_DATA          16777216      16777216
RLIMIT_FSIZE          2097152       2097152
RLIMIT_NOFILE              64          1024
RLIMIT_STACK         16777216      16777216
RLIMIT_VMEM          16777216      16777216
```

在4.3+BSD下运行此程序，得到：

```
$ a.out
RLIMIT_CORE        (infinite)    (infinite)
RLIMIT_CPU         (infinite)    (infinite)
RLIMIT_DATA           8388608      16777216
RLIMIT_FSIZE       (infinite)    (infinite)
RLIMIT_MEMLOCK     (infinite)    (infinite)
RLIMIT_OFILE               64    (infinite)
RLIMIT_NPROC               40    (infinite)
RLIMIT_RSS           27070464      27070464
RLIMIT_STACK           524288      16777216
```

在介绍了信号机构后，习题10.11将继续讨论资源限制。

7.12 小结

理解在UNIX环境中C程序的环境是理解UNIX进程控制特征的先决条件。本章说明了一个进程是如何起动和终止的，如何向其传递传数表和环境。虽然这两者都不是由内核进行解释的，但内核起到了从exec的调用者将这两者传递给新进程的作用。

本章也说明了C程序的典型存储器布局，以及一个进程如何动态地分配和释放存储器。详细地了解用于维护环境的一些函数是值得的，因为它们涉及存储器分配。本章也介绍了 setjmp和longjmp函数，它们提供了一种在进程内非本地转移的方法。最后介绍了 SVR4和4.3+BSD提供的资源限制功能。

习题

7.1 在80386系统上，无论使用SVR4或4.3+BSD，如果执行一个输出“ hello, world ”但不调用exit 或return，则程序的返回代码为13（用shell检查），解释其原因。

7.2 程序7-1中的printf函数的结果何时才被真正输出？

7.3 是否有方法不使用 (a)参数传递(b)全局变量这两种方法，将 main中的参数argc,argv传递给它所调用的其他函数？

7.4 在有些UNIX系统中执行程序时，为什么访问不到其数据段的0单元？

7.5 用C语言的typdef定义一个新的数据类型 Exitfunc处理退出，利用该类型修改 atexit的原型。

7.6 如果用 calloc分配一个long型的数组，数组的初始值是否为 0？如果用 calloc分配一个指针数组，数组的初始值是否为空指针？

7.7 在7.6节size命令的输出结果中，为什么没有给出堆和堆栈的大小？

7.8 为什么 7.7 节中两个文件的大小 (104859和24576)不等于它们各自文本和数据大小的和？

7.9 为什么7.7节中对程序使用共享库以后改变了其可执行文件的大小？

7.10 在7.10节中我们已经说明为什么不能将一个指针返回给一个自动变量，下面的程序是否正确？

```
int
f1(int val)
{
   int      *ptr;
   if (val == 0) {
       int      val;
       val = 5;
       ptr = &val;
   }
   return (*ptr + 1);
}
```