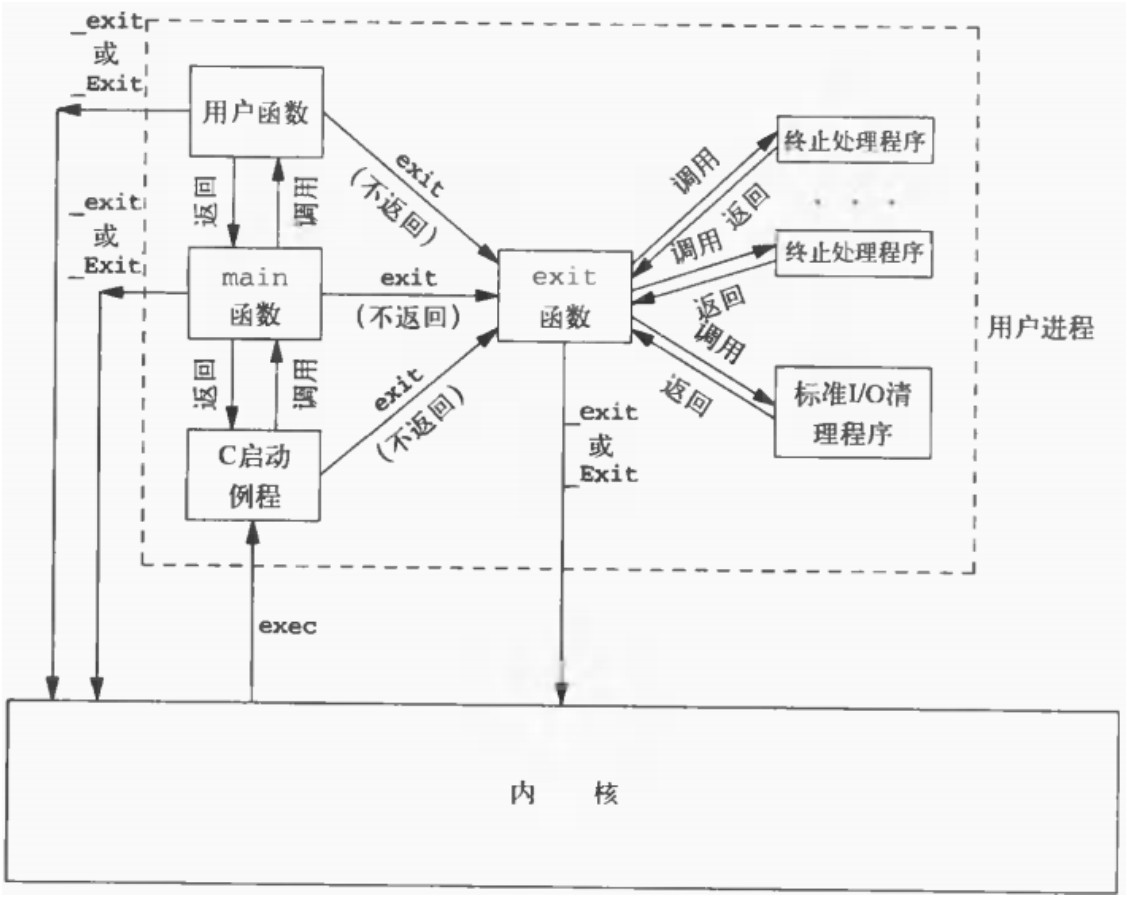
## 第七章 进程环境




http://linuxgazette.net/23/flower/context.html

进程上下文:

每次进程从正在运行的处理器上移除时，它当前运行状态的充足信息必须被保存。这样，当它在此被调入处理器中时，它可以从原来的地方恢复运行。该已知的操作状态数据就是其 ”上下文(context)”，且正执行在处理器中该进程的线程，被移除 (及替换它为其它)的行为，就是大家常说的进程切换或上下文切换。

这里对它们之间的差异做出了说明，其中一种是，进程被从处理器中完全移除，并替换成另一个进程（进程切换），另一种是进程状态被保存，且执行被临时中断（上下文切换）。注意，一个进程的切换是在执行上下文切换操作所需的超集。后面情况也许是由一个外部中断所引起，或者是由一个必须从用户模式切换到系统模式的系统调用引起。首先在进程切换时，为了后续恢复进程上下文，有很多信息需要被保存，接着进程继续常驻内存，而它执行中的线程会被中断。

一个进程的上下文包括了它的地址空间，栈空间，虚拟地址空间，寄存器设置镜像（例如：计数器(PC)，栈指针(SP)，指令寄存器(IR)，程序状态字(PSW)及其它通用处理器寄存器），更新分析或统计信息，实现一个与它相关联的内核数据结构的快照镜像及更新进程的当前状态（等待，就绪等）。

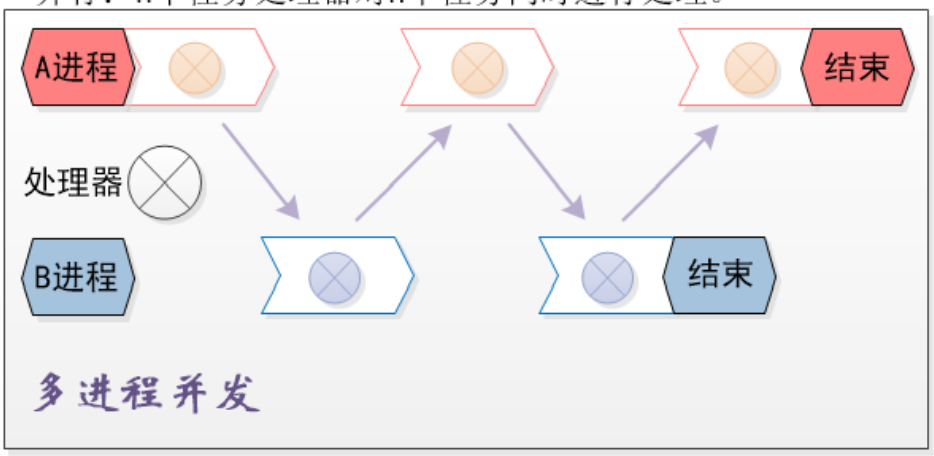
该状态信息保存在进程的进程控制块处，然后将其移动到适当的调度队列中。新进程是通过复制PCB信息到适当的位置被移到CPU的，（例如：计数器加载为下一个将被执行的指令地址处）。

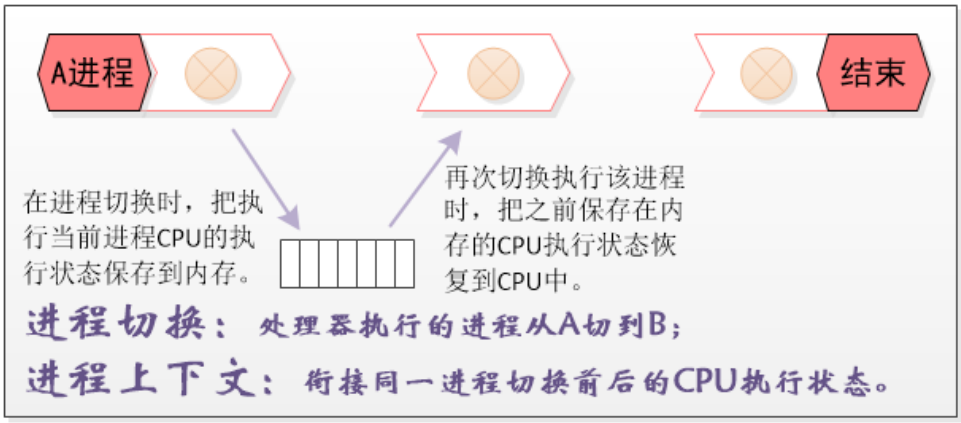
每个进程上下文被一个task_struct结构体所描述。task_struct所持有的数据有如调度策略、调度优先级、实时优先级、处理器允许的时间计数器、处理器寄存器、文件处理（files_struct）、虚拟内存（mm_struct）。

当一个进程切换时，产生一个调度，存储进程的task_struct，并使当前任务的指针指到新进程的task_struct处，恢复它的内存访问及寄存器上下文。这也许关联到硬件部分。

比喻：进程就像一个公司，是一个容器。公司里面处理具体事务的是相关的部门。一个公司只少要有一个处理事务的部门。当然也可以有多个。里面的部门，就像进程里的线程一样。处理具体事务逻辑。

并发：从宏观上看，多个任务同时进行。实际上的实现，
一般为多个任务分时轮流享用处理资源。
并行：n个任务处理器对n个任务同时进行处理。




http://technet.microsoft.com/zh-cn/library/hh439648
http://linuxgazette.net/23/flower/vmem.html

虚拟内存：

虚拟内存提供了一种可以在计算机物理内存地址空间中运行多个进程（可以把它们物理化）的方法。

每个进程是当前运行在 CPU上进程的候补者，是已分配的虚拟内存区域的拥有者，虚拟内存区域定义了逻辑地址集，拥有它的进程可以在此访问、履行其所携的任务。因为虚拟内存区域的总量是非常巨大的（通常受限于处理器的位宽和它所支持的进程最大数），每个进程运作在一个可被分配的巨大逻辑地址空间中（通常3Gb）。

进程保活由虚拟内存管理器负责，且该区域希望在访问时，按要求被重映射到物理内存上。该实现通过交换方法或分页所需的部分来达到物理内存置换的目的。交换部分包含在内存中替换一个完整的进程到另一个，然而分页却涉及了进程映射内存页（通常2-4KB）的移除和从另一个进程替换页内存。因为这可能是一个使计算机资源紧张且耗时的任务，

解决措施是尽量减少这类开销。这样做是由于一些算法的设计利用代码相关部分的共同局部性的优势，这些也仅是进行一些操作，如内存复制或当必须读取时（已知的技术有写拷贝、懒分页和按需分页）。

进程拥有的虚拟内存里包含代码、来自多个源的数据。执行代码也许以共享库的方式在进程间被共享，因为这些区域是只读的，所以被损坏的几率很小。进程可以在使用它们之前分配和链接虚拟内存。

http://linuxgazette.net/23/flower/psimage.html

进程镜像

可执行文件以一种定义好了的格式存储在磁盘上，不同的操作系统在实际的定义上会有不同，但通常它们共有的元素在存储时会被存储。在unix上常用的格式，例如在linux上被称为可执行连接格式（Extensible Linked Format (ELF)）。一个ELF的安排由一个ELF头，一个程序头表，一个区段只和一个可选区段的头表。头部包含了内核创建进程镜像的所有必要信息（即加载程序到内存，并为将要执行的它分配资源）。

程序代码在一个多任务操作系统上必须可重入。这表示它可以被多个进程共享。可重入的代码必须在任何时间都不能修改其自身，且数据必须从指令文本处就分开存储（这样每个独立的进程可以维护它自己的数据空间）。

当一个程序被加载为一个进程时，它所分配到的虚拟内存部分构成了它可用的地址空间。在此进程镜像中，通常至少包含 4种元素：

程序代码（或文本）：程序被独立执行。注意，在一个程序运行时，这部分不是处理器必须要读到物理内存的一个进程的总量，而不是由一个进程已知？动态分页？下一个独立块加载所需且或许共享与进程之间。

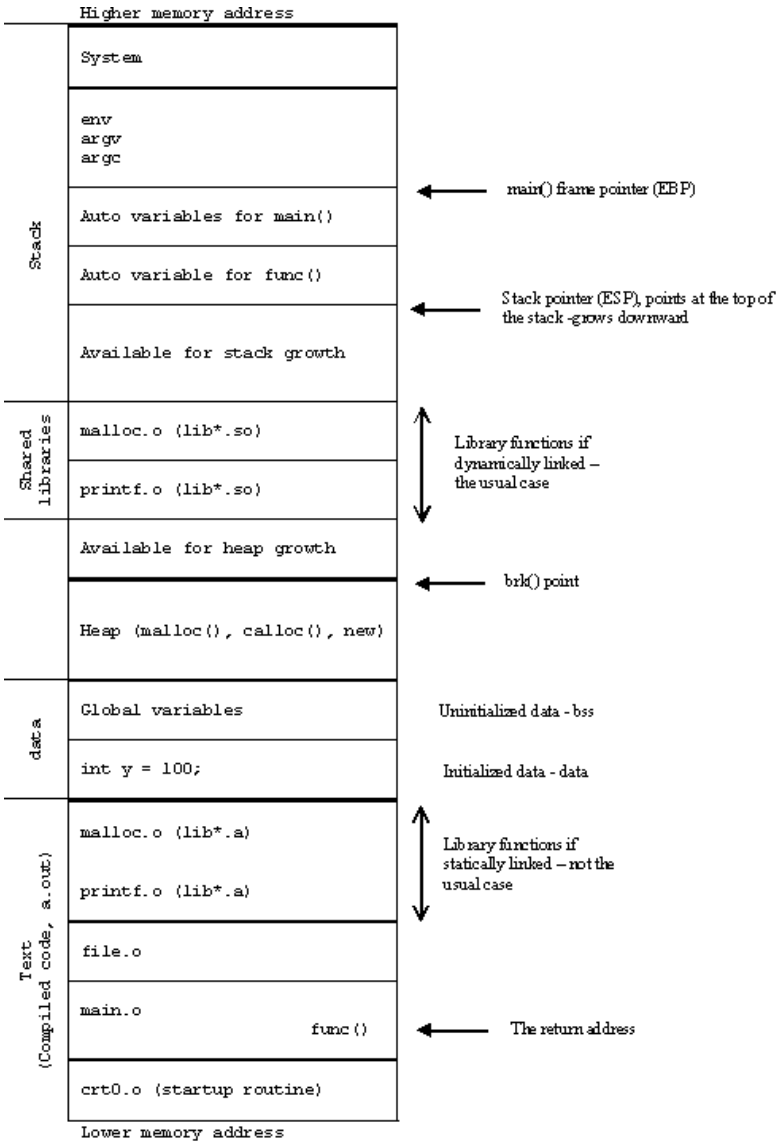
程序数据：可分为初始化变量和未初始化变量（已知的如：在 Unix系统上衍生出来的bss区域），初始化变量包含外部全局变量和静态变量。数据块默认不被在进程间共享。

栈：一个进程通常又两个栈（后进先出 (LIFO)），用户模式的用户栈和内核模式的内核栈。

进程控制块：操作系统控制进程所必须的信息。

http://www.x86-64.org/documentation/abi.pdf
http://www.sco.com/developers/devspecs/gabi41.pdf
http://blog.csdn.net/colin719/article/details/1482088

变量类型：全局(外部)变量、静态变量、全局静态变量、局部(自动)变量、初始化变量、未初始化变量；




喻：系统中的进程，就像一个空荡荡的教室一样(一个空间(计算机中叫做地址空间))，教室里面有前左角、讲台、门角、后左角、后门脚及桌椅区。每个教室间的格局是一样的(即以教室力的某一点为参考点，里面区域相对是一样的)，但不同教室间相同区域放置的东西会不相同。系统像一个学校，会有很多教室。并且在一定限制下，可以任意创建和消失一个教室。

http://soft.chinabyte.com/os/51/12324551.shtml

申请方式

stack:

由系统自动分配。 例如，声明在函数中一个局部变量int b; 系统自动在栈中为b开辟空间

heap:

需要程序员自己申请，并指明大小，在c中malloc函数

如p1 = (char *)malloc(10);

在C++中用new运算符

如p2 = new char[20];//(char *)malloc(10);

但是注意p1、p2本身是在栈中的。

## Special Sections

Various sections hold program and control information. Sections in the list below are used by the system and have the indicated types and attributes.

**Figure 4-13: Special Sections**

| Name | Type | Attributes |
|---|---|---|
| .bss | SHT_NOBITS | SHF_ALLOC + SHF_WRITE |
| .comment | SHT_PROGBITS | none |
| .data | SHT_PROGBITS | SHF_ALLOC + SHF_WRITE |
| .data1 | SHT_PROGBITS | SHF_ALLOC + SHF_WRITE |
| .debug | SHT_PROGBITS | none |
| .dynamic | SHT_DYNAMIC | see below |
| .dynstr | SHT_STRTAB | SHF_ALLOC |
| .dynsym | SHT_DYNSYM | SHF_ALLOC |
| .fini | SHT_PROGBITS | SHF_ALLOC + SHF_EXECINSTR |
| .got | SHT_PROGBITS | see below |

**Figure 4-13: Special Sections** (continued)

| | | |
|---|---|---|
| .hash | SHT_HASH | SHF_ALLOC |
| .init | SHT_PROGBITS | SHF_ALLOC + SHF_EXECINSTR |
| .interp | SHT_PROGBITS | see below |
| .line | SHT_PROGBITS | none |
| .note | SHT_NOTE | none |
| .plt | SHT_PROGBITS | see below |
| .rel*name* | SHT_REL | see below |
| .rela*name* | SHT_RELA | see below |
| .rodata | SHT_PROGBITS | SHF_ALLOC |
| .rodata1 | SHT_PROGBITS | SHF_ALLOC |
| .shstrtab | SHT_STRTAB | none |
| .strtab | SHT_STRTAB | see below |
| .symtab | SHT_SYMTAB | see below |
| .text | SHT_PROGBITS | SHF_ALLOC + SHF_EXECINSTR |

.bss This section holds uninitialized data that contribute to the program's memory image. By definition, the system initializes the data with zeros when the program begins to run. The section occupies no file space, as indicated by the section type, SHT_NOBITS.

.comment This section holds version control information.

.data and .data1
These sections hold initialized data that contribute to the program's memory image.

.debug This section holds information for symbolic debugging. The contents are unspecified. All section names with the prefix .debug are reserved for future use in the ABI.

.dynamic This section holds dynamic linking information. The section's attributes will include the SHF_ALLOC bit. Whether the SHF_WRITE bit is set is processor specific. See Chapter 5 for more information.

.dynstr This section holds strings needed for dynamic linking, most commonly the strings that represent the names associated with symbol table entries. See Chapter 5 for more information.

.dynsym This section holds the dynamic linking symbol table, as "Symbol Table" describes. See Chapter 5 for more information.

`.fini` This section holds executable instructions that contribute to the process termination code. That is, when a program exits normally, the system arranges to execute the code in this section.

`.got` This section holds the global offset table. See ''Coding Examples'' in Chapter 3, ''Special Sections'' in Chapter 4, and ''Global Offset Table'' in Chapter 5 of the processor supplement for more information.

`.hash` This section holds a symbol hash table. See ''Hash Table'' in Chapter 5 for more information.

`.init` This section holds executable instructions that contribute to the process initialization code. That is, when a program starts to run, the system arranges to execute the code in this section before calling the main program entry point (called `main` for C programs).

`.interp` This section holds the path name of a program interpreter. If the file has a loadable segment that includes the section, the section's attributes will include the `SHF_ALLOC` bit; otherwise, that bit will be off. See Chapter 5 for more information.

`.line` This section holds line number information for symbolic debugging, which describes the correspondence between the source program and the machine code. The contents are unspecified.

`.note` This section holds information in the format that ''Note Section'' in Chapter 5 describes.

`.plt` This section holds the procedure linkage table. See ''Special Sections'' in Chapter 4 and ''Procedure Linkage Table'' in Chapter 5 of the processor supplement for more information.

`.rel`*name* and `.rela`*name*
These sections hold relocation information, as ''Relocation'' below describes. If the file has a loadable segment that includes relocation, the sections' attributes will include the `SHF_ALLOC` bit; otherwise, that bit will be off. Conventionally, *name* is supplied by the section to which the relocations apply. Thus a relocation section for `.text` normally would have the name `.rel.text` or `.rela.text`.

`.rodata` and `.rodata1`
These sections hold read-only data that typically contribute to a non-writable segment in the process image. See ''Program Header'' in Chapter 5 for more information.

`.shstrtab` This section holds section names.

`.strtab` This section holds strings, most commonly the strings that represent the names associated with symbol table entries. If the file has a loadable segment that includes the symbol string table, the section's attributes will include the `SHF_ALLOC` bit; otherwise, that bit will be off.

`.symtab` This section holds a symbol table, as ''Symbol Table'' in this chapter describes. If the file has a loadable segment that includes the symbol table, the section's attributes will include the `SHF_ALLOC` bit; otherwise, that bit will be off.

`.text` This section holds the ''text,'' or executable instructions, of a program.

Section names with a dot (`.`) prefix are reserved for the system, although applications may use these sections if their existing meanings are satisfactory. Applications may use names without the prefix to avoid conflicts with system sections. The object file format lets one define sections not in the list above. An object file may have more than one section with the same name.

Section names reserved for a processor architecture are formed by placing an abbreviation of the architecture name ahead of the section name. The name should be taken from the architecture names used for `e_machine`. For instance .FOO.psect is the psect section defined by the FOO architecture. Existing extensions are called by their historical names.

Pre-existing Extensions

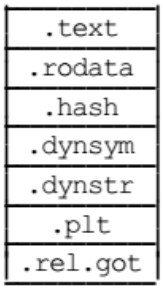
| | |
|---|---|
| `.sdata` | `.tdesc` |
| `.sbss` | `.lit4` |
| `.lit8` | `.reginfo` |
| `.gptab` | `.liblist` |
| `.conflict` | |

## Segment Contents

An object file segment comprises one or more sections, though this fact is transparent to the program header. Whether the file segment holds one or many sections also is immaterial to program loading. Nonetheless, various data must be present for program execution, dynamic linking, and so on. The diagrams below illustrate segment contents in general terms. The order and membership of sections within a segment may vary; moreover, processor-specific constraints may alter the examples below. See the processor supplement for details.

Text segments contain read-only instructions and data, typically including the following sections described in Chapter 4. Other sections may also reside in loadable segments; these examples are not meant to give complete and exclusive segment contents.

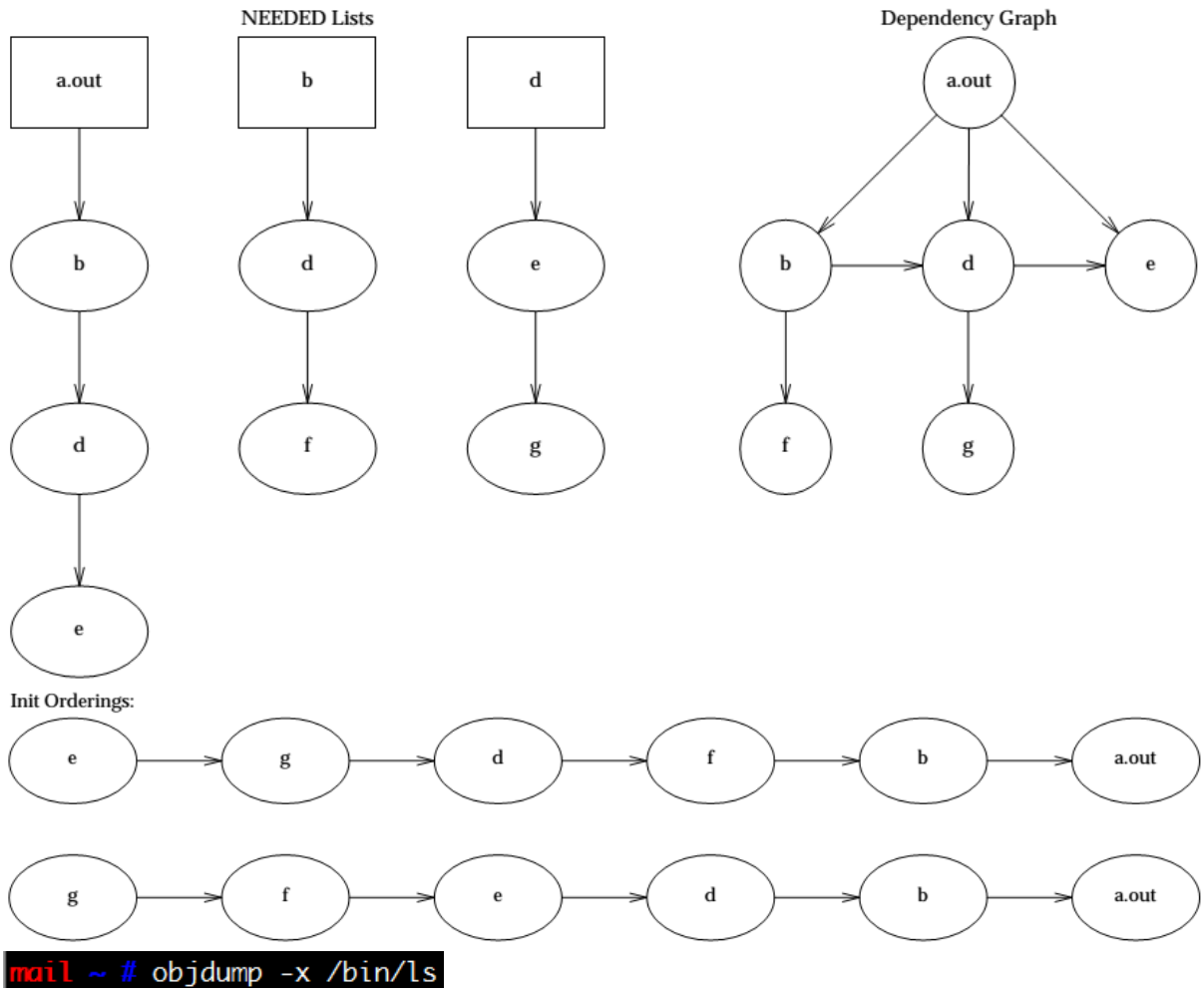
**Figure 5-5: Text Segment**




Data segments contain writable data and instructions, typically including the following sections.

**Figure 5-6: Data Segment**

**Figure 5-13: Initialization Ordering Example**




```
mail ~ # objdump -x /bin/ls
```

```
mail ~ # readelf -a /bin/ls
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x404c44
  Start of program headers:          64 (bytes into file)
  Start of section headers:          108184 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         10
  Size of section headers:           64 (bytes)
  Number of section headers:         27
  Section header string table index: 26
```

```
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000400270  00000270
       000000000000001c  0000000000000000   A       0     0     1
```

PT_NULL — The array element is unused; other members' values are undefined. This type lets the program header table have ignored entries.

PT_LOAD — The array element specifies a loadable segment, described by p_filesz and p_memsz. The bytes from the file are mapped to the beginning of the memory segment. If the segment's memory size (p_memsz) is larger than the file size (p_filesz), the "extra"

bytes are defined to hold the value 0 and to follow the segment's initialized area. The file size may not be larger than the memory size. Loadable segment entries in the program header table appear in ascending order, sorted on the `p_vaddr` member.

`PT_DYNAMIC` The array element specifies dynamic linking information. See ''Dynamic Section'' below for more information.

`PT_INTERP` The array element specifies the location and size of a null-terminated path name to invoke as an interpreter. This segment type is meaningful only for executable files (though it may occur for shared objects); it may not occur more than once in a file. If it is present, it must precede any loadable segment entry. See ''Program Interpreter'' below for further information.

`PT_NOTE` The array element specifies the location and size of auxiliary information. See ''Note Section'' below for details.

`PT_SHLIB` This segment type is reserved but has unspecified semantics. Programs that contain an array element of this type do not conform to the ABI.

`PT_PHDR` The array element, if present, specifies the location and size of the program header table itself, both in the file and in the memory image of the program. This segment type may not occur more than once in a file. Moreover, it may occur only if the program header table is part of the memory image of the program. If it is present, it must precede any loadable segment entry. See ''Program Interpreter'' below for further information.

`PT_LOPROC` through `PT_HIPROC`
Values in this inclusive range are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.

```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x0000000000000230 0x0000000000000230  R E    8
  INTERP         0x0000000000000270 0x0000000000400270 0x0000000000400270
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000006bc 0x00000000000006bc  R E    200000
  LOAD           0x0000000000000e18 0x0000000000600e18 0x0000000000600e18
                 0x0000000000000200 0x0000000000000210  RW     200000
  DYNAMIC        0x0000000000000e40 0x0000000000600e40 0x0000000000600e40
                 0x00000000000001a0 0x00000000000001a0  RW     8
  NOTE           0x000000000000028c 0x000000000040028c 0x000000000040028c
                 0x0000000000000020 0x0000000000000020  R      4
  GNU_EH_FRAME   0x00000000000005ec 0x00000000004005ec 0x00000000004005ec
                 0x000000000000002c 0x000000000000002c  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     8
  GNU_RELRO      0x0000000000000e18 0x0000000000600e18 0x0000000000600e18
                 0x00000000000001e8 0x00000000000001e8  R      1
  PAX_FLAGS      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000         8
```

```
 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .hash .gnu.hash .dynsym .dynstr .gnu.version .gnu.v
ersion_r .rela.dyn .rela.plt .init .plt .text .fini .rodata .eh_frame_hdr .eh_frame

   03     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag
   06     .eh_frame_hdr
   07
   08     .ctors .dtors .jcr .dynamic .got
   09
```

习题：

7-1、返回值为最后输出的字符串长度；

三个exit函数都带一个整型参数，称之为终止状态（或退出状态，exit status）。大多数UNIX shell都提供检查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没有声明返回类型为整型，则该进程的终止状态是未定义的。但是，若main的返回类型是整型，并且main执行到最后一条语句时返回（隐式返回），那么该进程的终止状态是0。

这种处理是ISO C标准1999版引入的。历史上，若main函数终止时没有显式使用return语句或调用exit函数，那么进程的终止状态是未定义的。

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

main(int argc, const char *argv[]) {
    printf("String length of output: %ld\n", strlen("Hello World!\n"));
    printf("Hello World!\n");
}
```

```
mail cpp # gcc 7-1.c -o 7-1
mail cpp # ./7-1
String length of output: 13
Hello World!
mail cpp # echo $?
13
```

7-2、验证了图7-1；

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static void fun_01() {
    printf("fun_01 print!\n");
}
static void fun_02() {
    printf("fun_02 print!\n");
}

int main(int argc, const char *argv[]) {
    if (0 != atexit(fun_01)) {
        printf("Error by atexit fun_01!\n");
    }
    if (0 != atexit(fun_02)) {
        printf("Error by atexit fun_02!\n");
    }
    printf("Main print!\n");
    setvbuf(stderr, _IOFBF, 0, 0); 把标准错误置为全缓冲
    fwrite("StdIO print\n", sizeof(char), strlen("StdIO print\n"), stderr);
    sleep(3);
    printf("After sleeping!\n");
    return 0;
}
/opt/cpp/7-2.c [FORMAT=unix:utf-8] [TYPE=C] [COL=001] [ROW=001/25(4%)]
```

与标准输出分开，不然会让后面的printf标准输出，把前面的fwrite标准输出给挤出来

```
mail cpp # gcc 7-2.c -o 7-2
mail cpp # ./7-2
Main print!
After sleeping!
fun_02 print!
fun_01 print!
StdIO print
```

注意输出顺序

7-3、对该题进行一下扩展，加入初始化程序参数，有下可看出，用环境变量能实现该题基本要求；

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void load_args(const char *argv[], int i) {
    if (0 == strcmp("-l", argv[i])) {
        setenv("LISTEN_ADDR", argv[i+1], 1);
    } else if (0 == strcmp("-p", argv[i])) {
        setenv("LISTEN_PORT", argv[i+1], 1);
    } else {
        printf("Invalid parameter %s!\n", argv[i]);
    }
}

void display_env() {
    printf("Listen Addr: %s\n", getenv("LISTEN_ADDR"));
    printf("Listen Port: %s\n", getenv("LISTEN_PORT"));
}

int main(int argc, const char *argv[]) {
    int i;
    for (i=1; NULL != argv[i]; i++) {
        if (1 == (i % 2)) {
            load_args(argv, i);
        }
    }
    display_env();
    return 0;
}
```

/opt/cpp/7-3.c [FORMAT=unix:utf-8] [TYPE=C] [COL=001] [ROW=001/29(3%)]

```
mail cpp # gcc 7-3.c -o 7-3
mail cpp # ./7-3 -l 0.0.0.0 -p 8888
Listen Addr: 0.0.0.0
Listen Port: 8888
```

7-4、无法解释；

7-5、该题意在让读者把下图红框中的函数指针定义为一个简短类型；

参考链接：http://www.cnblogs.com/leon3000/archive/2007/11/15/2037989.html

```c
/* Register a function to be called when `exit' is called.  */
extern int atexit (void (*__func) (void)) __THROW __nonnull ((1));
```

typedef void (*Exitfunc) (void);
int atexit(Exitfunc pEfun);
或者
typedef void (Exitfunc) (void);
int atexit(Exitfunc *pEfun);

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

#ifdef _FIRST_
    typedef void (*Exitfunc) (void);
    int atexit(Exitfunc pEfun);
#else
    typedef void (Exitfunc) (void);
    int atexit(Exitfunc *pEfun);
#endif

void fun_01() {
    printf("fun_01 print!\n");
}

int main(int argc, const char *argv[])
{
    if (0 != atexit(&fun_01)) {
        printf("Error by atexit fun_01!\n");
    }
    printf("Main print!\n");
    return 0;
}
```

```
mail cpp # gcc 7-5.c -o 7-5 -D_FIRST_
mail cpp # ./7-5
Main print!
fun_01 print!
mail cpp # gcc 7-5.c -o 7-5
mail cpp # ./7-5
Main print!
fun_01 print!
```

7-6、分配的指针地址，显然不为空(0)；

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

int main(int argc, const char *argv[]) {
    long *arr_long = calloc( 10, sizeof(long));
    printf("Arr_long: %ld\n", arr_long[0]);
    printf("Arr_long: %ld\n", arr_long[9]);
    void *pArr = calloc( 10, sizeof(void *));
    printf("pArr: %p\n", &pArr[0]);
    printf("pArr: %p\n", &pArr[9]);
    return 0;
}
```

```
mail cpp # gcc 7-6.c -o 7-6
7-6.c: In function 「main」:
7-6.c:9:31: 警告：提領 「void *」 指標 [enabled by default]
7-6.c:9:26: 警告：taking address of expression of type 「void」 [enabled by default]
7-6.c:10:31: 警告：提領 「void *」 指標 [enabled by default]
7-6.c:10:26: 警告：taking address of expression of type 「void」 [enabled by default]
mail cpp # ./7-6
Arr_long: 0
Arr_long: 0
pArr: 0x12d8070
pArr: 0x12d8079
```

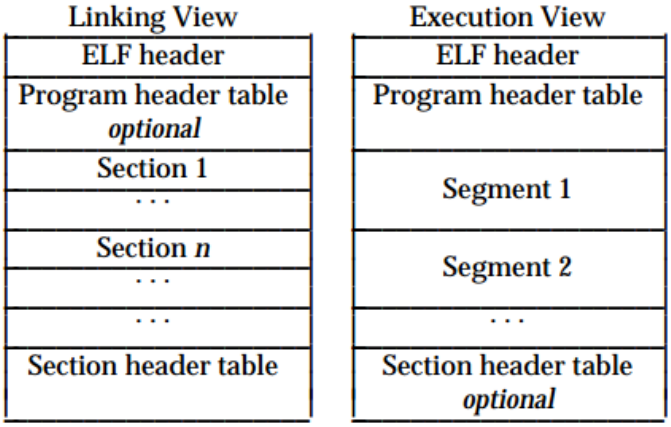
7-7、堆和栈是当程序被载入为进程时，由系统分配的，分配之前是不存在于程序中的；

7-8、size仅显示出程序的主要几个部分（text、data、bss），程序除了之前三个部分，还有其它信息保存在该可执行文件中；

参考链接：http://blog.chinaunix.net/uid-28596231-id-3841970.html

gabi41.pdf

**Figure 4-1: Object File Format**




a.out中还有若干其他类型的段，例如，包含符号表的段、包含调试信息的段以及包含动态共享库链接表的段等等。这些部分并不装载到进程执行的程序映像中。

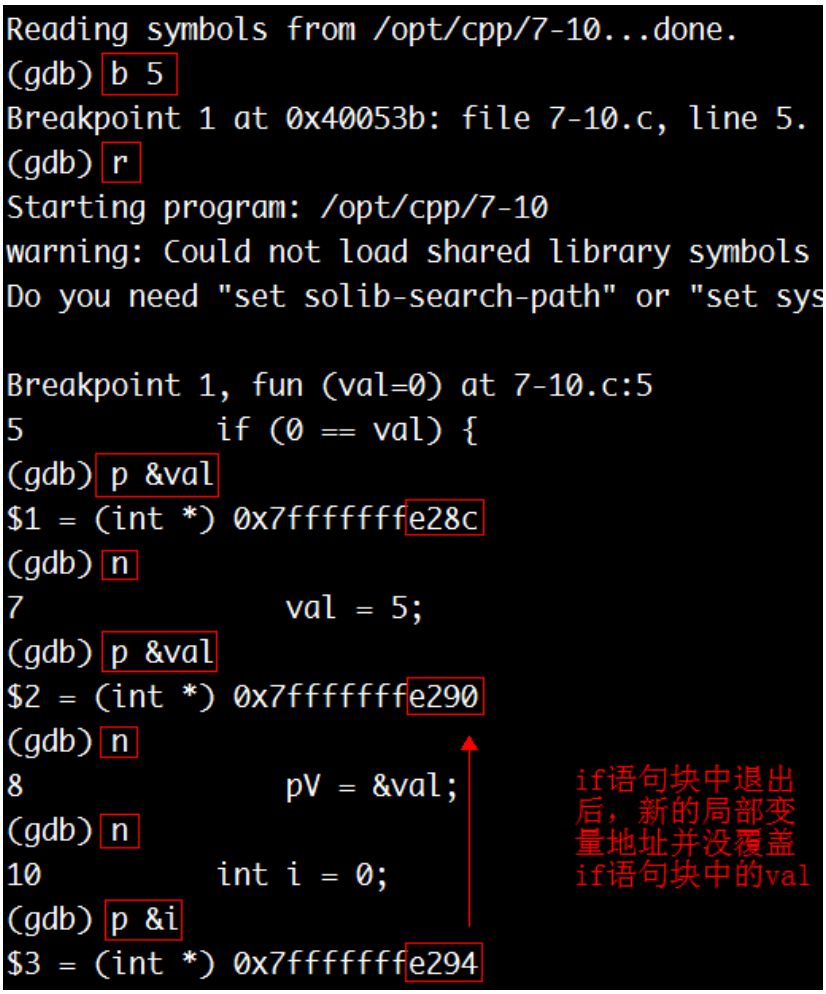
7-9、静态编译把原来动态链接的对象文件都放到本文件中了；

```
mail cpp # gcc 7-9.c -o 7-9
mail cpp # ls -l 7-9
-rwxr-xr-x 1 root root 7781  1月 13 18:19 7-9
mail cpp # size 7-9
   text    data     bss     dec     hex filename
   1080     512      16    1608     648 7-9
mail cpp # readelf -d 7-9

Dynamic section at offset 0xe40 contains 21 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000c (INIT)               0x4003c0
```

```
mail cpp # gcc -static 7-9.c -o 7-9
mail cpp # ls -l 7-9
-rwxr-xr-x 1 root root 837799  1月 13 18:20 7-9
mail cpp # size 7-9
   text    data     bss     dec     hex filename
 747206    5904   10408  763518   ba67e 7-9
mail cpp # readelf -d 7-9

There is no dynamic section in this file.
```

7-10、正确。首先，函数里面的所有自动变量所占的内存空间，只在函数返回时才被释放。该程序中，在if语句块退出后，我们虽然无法直接访问if语句里面的变量val，但其所占的内存空间并未被释放，但仍可通过指针来访问它所在地址的内容。其次，函数返回值是个int型，函数最后返回的是(*ptr + 1)的结果，也是int型，两者刚好匹配，完美接收。所以程序运行不会有误。

```
mail cpp # gcc -g 7-10.c -o 7-10
mail cpp # gdb 7-10
GNU gdb (Gentoo 7.5.1 p2) 7.5.1
```


Reading symbols from /opt/cpp/7-10...done.
(gdb) b 5
Breakpoint 1 at 0x40053b: file 7-10.c, line 5.
(gdb) r
Starting program: /opt/cpp/7-10
warning: Could not load shared library symbols
Do you need "set solib-search-path" or "set sys
Breakpoint 1, fun (val=0) at 7-10.c:5
5 if (0 == val) {
(gdb) p &val
$1 = (int *) 0x7fffffffe28c
(gdb) n
7 val = 5;
(gdb) p &val
$2 = (int *) 0x7fffffffe290
(gdb) n
8 pV = &val;
(gdb) n
10 int i = 0;
(gdb) p &i
$3 = (int *) 0x7fffffffe294
if语句块中退出后，新的局部变量地址并没覆盖if语句块中的val