嵌入式 Linux 性能详解

史子旺
loughsky@sina.com

1. 序

书名为《嵌入式 Linux 内存与性能详解》，说实话我有些心虚。嵌入式设备的种类太多，各种用法千奇百怪，关于 Linux 操作系统的剪裁方面的文章就更多了，我不可能都十分了解，我只了解我目前正在使用的系统，智能手机和 Linux 2.6（MontaVista 3.4.3），但我还是坚信这本书中的有些东西还是通用的，希望对您有所帮助。也正是因为我在一家手机公司工作，所以我大多以手机举例。

上学的时候，老师就告诉我们，“不要关注内存，现在内存条很便宜；不要太多的强调技巧，因为现在 CPU 速度已经很快了，我们应该更多看重的是程序的结构和良好的扩展性、可读性。”

在这里，我首先欢迎你加入到嵌入式 Linux 软件开发的行列，你在一个告诉发展，最有前途的行业，全球的著名 IT 公司都在这里竞争，而其还处在发展阶段，有的是供你发展的空间。其次我要告诉你几个坏消息，希望你有心里准备：

1、 目前还是战国时代，公司与公司之间，平台与平台之间竞争激烈，这对我们而言就意味着加班。
2、 环境很恶劣，现在大部分智能设备的 CPU 还处在 200～500Mhz 之间，与我 2000 年买的奔腾 II 计算机处于一个水平；内存 32M～64M，我就不明白，在当前 1G 内存也就 100 多元的时代，公司为什么还这么吝啬；更可恨的是，在手机上没有显卡和声卡，全部由 CPU 来负荷，多重的担子。
   而这就对我们的程序提出了更高的要求。
3、 调试手段单一，目前我们的调试还处于一种打 log 的状态，虽然也有 gdb，也有图形的 IDE 环境，可就是没用起来，这对于 UNIX 程序员来讲是想当然，而对于从 windows 转移过来的程序员来讲，则是十分丑陋。

不过这不应该由我来抱怨，正因为此，才有我的存在，才有我们作为系统优化人员存在的价值。

在书店里，充斥着关于 Linux 的图书，教你如何编写程序的图书，教你解读 Linux 内核，但是很少有告诉你程序是如何运行的，libc 是如何进行内存管理，程序与系统如何交互等等，而这正是我们做系统优化所不得不关心的问题。
这些知识往往面很杂，分布在不同的地方，需要你去整理。做优化，快两年了，我逐渐尝试着把工作过程中所学习到的知识，进行总结，不清楚的地方去研究，争取做到系统化，这样一方面对自己也是一个提高，另一方面，也希望对大家的工作有些帮助。

真的没想到系统化是这么累的一项工作，感谢我的妻子，是她在我想要放弃的时候，给了我支持，这本书才得以完成。

这本书主要介绍两个方面的内容：

1、 如何节省系统内存；

2、 如何加快进程运行速度。

这本书主要面向的人员：
1、 嵌入式 Linux 软件开发人员
2、 系统优化人员
3、 系统构架师。

2. 内存

记得，那是前年的一个冬天，我们都很高兴，因为新设备上要增加的功能已经完成的差不多了，我们可以在上班时喝喝茶、聊聊天，幻想着周末去滑雪。这时老板把我们召集到一起了个会。
我们走进会议室，老板在那里摆弄着手机，同时招手示意我们坐下。
我们坐在那里，静静的很疑惑的看着老板。

“好了”，老板从座位上站了起来，把手机递给了我们。

我们一看，手机上显示“Memory Low”（内存不足）。再看看手机上运行着 地址本，Java 小游戏还有 MP3。

“这不是我们的问题，哪个手机也不能同时这么多进程。”
“用户就不该这么用，它应该退出 Java 游戏，然后再去听歌。”
……..
老板摆了摆手，“不要说这些，确实存在这样的问题。内存，我们现在最重要的问题是把系统使用内存减下去，每个进程都要减 20%。”

我们知道老板是对的，这样的东西是无法交付给用户的。会后每个小组都去检查自己的进程内存使用，而我的问题则是如何推动内存的优化。

要想改进一个事物，你必须了解它目前出于什么水平，都包括哪些东西，想办法去改善，评估优化后的结果。

那么我们面临的第一件事，便是如何知道当前系统的内存使用情况。

第一篇 内存的测量

我当前最关心的便是当前系统有多少内存可以使用！如果是在 windows 平台，我们可以很方便的从任务管理器，获得当前系统有多少可用内存，每个进程的内存占用量，但在 Linux 中我们该从何处获得这些信息呢？

2.1. 系统当前可用内存

你可以在 Linux 中，敲入 free 命令获得当前系统的内存使用情况。
#busybox free

	total	used	free	shared	buffers
Mem:	55636	52808	2828	0	3132
Swap:	0	0	0		
Total:	55636	52808	2828		

当我满心欢喜的敲入上面的命令时，收到的却是一阵阵的冷汗，2828k，我的系统只剩下了 2M 多的内存，系统还能跑吗，是不是命令有错误!!!

让我们来看看在 PC 机上的 Linux 系统中，结果是什么样子的。
#free

	total	used	free	shared	buffers	cached
Mem:	4091524	4021016	70508	0	7656	1824312
-/+ buffers/cache:		2189048	1902476			
Swap:	4088532	2891732	1196800			

这里我先解释一下几个关键的概念。
buffers：主要是用来给 Linux 系统中块设备做缓冲区。
cached：用来缓冲我们所打开的文件。
在系统中内存是很宝贵的资源，Linux 的思想是，如果内存充足，不用白不用，它会使用内存来 cache 一些文件，从而加快进程的运行速度；当内存不足时，这些内存又会被回收，供程序使用。
所以真正可用的内存=free+buffers+cached=70508+7656+1824312=1902476。

这样，我们就很清楚了，的确是 busybox 在实现 free 时有缺陷，它缺少了关键的一列 cache，因此我们无法获得确切的空闲内存值。

既然 free 命令行不通，那么我们的重点放在了 free 的数据是从哪里来的！
答案是：/proc

/proc 目录是一个特殊的文件系统，它不占用磁盘空间，该目录下的内容是根据用户请求的信息，由 Linux 内核实时生成的。所以我们可以通过 proc 目录，来访问 Linux 内核中的一部分数据。

我们可以从 proc 目录下的 meminfo 文件，获得当前系统的内存使用情况。

```
# cat /proc/meminfo
MemTotal:        55880 kB
MemFree:          2252 kB
Buffers:          3760 kB
Cached:          26112 kB
SwapCached:          0 kB
Active:          34652 kB
Inactive:         8716 kB
HighTotal:           0 kB
HighFree:            0 kB
LowTotal:        55880 kB
LowFree:          2252 kB
SwapTotal:           0 kB
SwapFree:            0 kB
Dirty:               0 kB
Writeback:           0 kB
Mapped:          34220 kB
Slab:             4504 kB
CommitLimit:     27940 kB
Committed_AS:   109704 kB
PageTables:       1876 kB
VmallocTotal:   196608 kB
VmallocUsed:      2360 kB
VmallocChunk:  4112384 kB
```

从上面的结果，我们可以知道系统当前总共拥有 55880k 物理内存，

其中

free	2252kB
Buffers	3760 kB
Cached	26112 kB

那么当前可用的物理内存=2252+3760+26112=32124kB，应该说还是不少的。

终于我们经过千辛万苦得到了现在系统可用内存，也就可以很容易的计算出

使用的物理内存=55880-32124=23756kB

下面的问题便是，这 23M 的内存都是被谁使用了呢？

2.2. 进程的内存使用

有了上面的经验，这次我们就很自然的又想到了 proc 目录，英雄所见略同啊！

我们进到 proc 目录下，查看一下都有什么内容。

```
# cd /proc
# ls
1                    298                  7
10                   299                  72
100                  3                    8
……
……
14                   345                  emg
159                  381                  kallsyms
16                   384                  kmsg
161                  388                  loadavg
```

这些蓝颜色的数字，代表着一个个目录；同时这些数字也与当前系统中运行进程的 PID 一一对应。在这些目录下面的文件，记录着这些进程在 Linux 内核中相应的数据。

在介绍相关文件之前，我不得不先停下来，引入一个概念“虚拟内存”。

2.2.1. 虚拟内存与物理内存

在我们编写程序，我们不用去考虑物理内存。在 32 位的操作系统，程序员面对的是每个进程 4G 的内存空间；而实际上呢，一台 64M 物理内存的设备上，可能要跑着几十个、甚至上百个这样的进程。我们将程序员所面对的 4G 内存空间，称为虚拟内存，操作系统为我们屏蔽了物理内存的使用。

在 Linux 中采用了延迟分配物理内存的策略，针对进程的内存分配请求，它只是在内核中分配一段虚拟地址，只有当确实使用这块内存时，系统才会为其分配物理地址。

下面我们来看一段代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    char *p =(char *) malloc(10);
    char *p1=（char *）malloc(200);
    strcpy(p,”123”);
    return 0;
}
```

问题一：虚拟内存、物理内存与代码的对应关系？

在 char *p=(char *)malloc(10)，只是分配了虚拟内存，kernel 不会分配物理页面给进程。

在 strcpy(p,“123”)，进程需要使用这块内存了，kernel 会产生一个页故障，从而为系统分配一个物理页面。

因此，系统为 p1 只是分配了一个虚拟空间，而对 p 分配了相应的物理内存。

问题二：虚拟内存与物理内存有多大？

虚拟内存是 210 个字节，物理内存是 4K Byte，因为 kernel 分配物理内存的最小单位为一个物理页面，一个物理页面为（4K Byte）。

上面我们说 p1 只是对应虚拟空间，是不准确的；确切的，应该是在 strcpy(p,”123”)之前 p 对应的是虚拟内存，在这之后，它确实对应着物理内存。

2.2.2. /proc/{pid}

在介绍了物理内存和虚拟内存的区别之后，我们继续介绍在 proc 目录下，有关进程的文件。

为了能够让大家更清楚的了解/proc/{pid}目录下的文件，我在这里写一个简单的例子，在系统中运行，然后我们查看 proc 目录下对应的信息。

hello.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
int main()
{
        char *p=(char *)malloc(20);
        strcpy(p,"123");
        pid_t pid=getpid();
        printf("pid:%d\n",pid);
        pause();                        //主要是为了进程停在这里，以便我们观看相关信息。
        return 0;
}
```

使用 gcc 编译后，下载到手机上。

```
gcc –o hello hello.c
```

在手机上运行：

```
./hello
pid:491
```

现在进程停在这里，我们可以另外起一个 telnet 窗口，到 proc 目录下去查看其信息。

```
#cd /proc
# cd 491
# ls
attr        auxv        cmdline    cwd        environ    exe        fd        maps        memmap
```

mounts nodemap root stat statm status task wchan

注意：在 Linux 的不同版本中，其下的文件会有所不同，在这里我只介绍几个与内存相关的文件，其余的大家可以在互联网上查到。

#cat statm
345 87 74 1 0 58 0

这里有 7 个数，它们以页（4K）为单位。
Size (total pages) 任务虚拟地址空间的大小
Resident(pages) 应用程序正在使用的物理内存的大小
Shared(pages) 共享页数
Trs(pages) 程序所拥有的可执行虚拟内存的大小
Lrs(pages) 被映像到任务的虚拟内存空间的库的大小
Drs(pages) 程序数据段和用户态的栈的大小
dt(pages) 脏页数量（已经修改的物理页面）
其中 Size Trs Lrs Drs 对应于进程的虚拟内存；Resident shared dr 对应于物理内存。
注：在我们现在的平台上，dt 的代码被修改了，直接返回 0

不看不知道，一看吓一跳。
如此简单的一个程序，运行时居然占用了，虚存居然占了 345 个页面，也就是 345*4=1380K 内存，实在太夸张了。

下面就让我们看看进程为什么使用了 1389k 的内存。
#cat maps

00008000-00009000 r-xp 00000000 1f:12 288 /mnt/msc_int0/hello
00010000-00011000 rw-p 00000000 1f:12 288 /mnt/msc_int0/hello
00011000-00032000 rwxp 00011000 00:00 0
40000000-40002000 rw-p 40000000 00:00 0
41000000-41017000 r-xp 00000000 1f:0d 817360 /lib/ld-2.3.3.so
4101e000-41020000 rw-p 00016000 1f:0d 817360 /lib/ld-2.3.3.so
41028000-41120000 r-xp 00000000 1f:0d 817593 /lib/libc-2.3.3.so
41120000-41128000 ---p 000f8000 1f:0d 817593 /lib/libc-2.3.3.so
41128000-41129000 r--p 000f8000 1f:0d 817593 /lib/libc-2.3.3.so
41129000-4112c000 rw-p 000f9000 1f:0d 817593 /lib/libc-2.3.3.so
4112c000-4112e000 rw-p 4112c000 00:00 0
befeb000-bf000000 rwxp befeb000 00:00 0

下面我们以第一行为例，解释各列的内容

第一列：00008000-00009000，代表该内存段的虚拟地址。

第二列：r-xp，代表着该内存的权限，其值含义为：

r=读，w=写,x=执行,s=共享,p=私有；

第三列：00000000，代表偏移量库在进程里地址范围

第四列：1f:12，映射文件的主设备号和次设备号。

我们可以通过 cat /proc/devices 来查看设备信息：

cd /proc

cat devices

Character devices:

1 mem

4 /dev/vc/0

5 /dev/tty

…….

252 mxc_ipc

254 devfs

Block devices:

1 ramdisk

7 loop

31 mtdblock

243 mmc

1f 转换成 10 进制为 31，我们可以知道该段内存映射位于 mtdblock 设备的文件。

第五列：288，映像文件的节点号；

第六列：/mnt/msc_int0/hello，映像文件的路径。正好对应着我们执行文件所对应的目录。

从 maps 中我们可以知道，在进程的内存空间中，不光包括进程本身，还包括 ld-2.3.3.so 和 libc-2.3.3.so 两个动态库。

在这里，我们先抛开动态库的内存不提，只关注进程自身内存的使用：

问题一：下面各行都是什么含义？

00008000-00009000 r-xp 00000000 1f:12 288 /mnt/msc_int0/hello

00010000-00011000 rw-p 00000000 1f:12 288 /mnt/msc_int0/hello

00011000-00032000 rwxp 00011000 00:00 0

40000000-40002000 rw-p 40000000 00:00 0

4112c000-4112e000 rw-p 4112c000 00:00 0

befeb000-bf000000 rwxp befeb000 00:00 0

回答：

00008000-00009000 r-xp 00000000 1f:12 288 /mnt/msc_int0/hello

从权限 r-xp，可以得知其权限为只读、可执行，该段内存地址对应于进程的代码段，不要忘了，程序的代码也需要加载到内存才可以执行的哦。

00010000-00011000 rw-p 00000000 1f:12 288 /mnt/msc_int0/hello

从权限 rw-p，可以得知其权限为读写，不可执行，该段内存地址对应与进程的数据段，主要存储进程所用到的全局变量。

00011000-00032000 rwxp 00011000 00:00 0

从权限 rwxp，可以得知其权限是读写可执行，内存地址向上增长，而且不对应文件，其为堆段，由于进程申请内存，而生成的该段内存地址，其为堆段，我们使用 malloc 申请的内存都放在这里。

细心的读者会发现一个问题，为什么我们的代码只申请了 20 个字节，而系统却为其分配了 132k 虚拟内存呢？

这个问题，在后面专门讲堆段时会提到。

40000000-40002000 rw-p 40000000 00:00 0

从权限来看，象数据段，但没有映射的文件；从没有映射文件这一条，象堆段，但其权限又不对。它是什么呢？

这个问题，现在我也回答不上来，往后看吧！

befeb000-bf000000 rwxp befeb000 00:00 0

其位于地址的顶端，内存区域向下增长，这段内存为栈段。

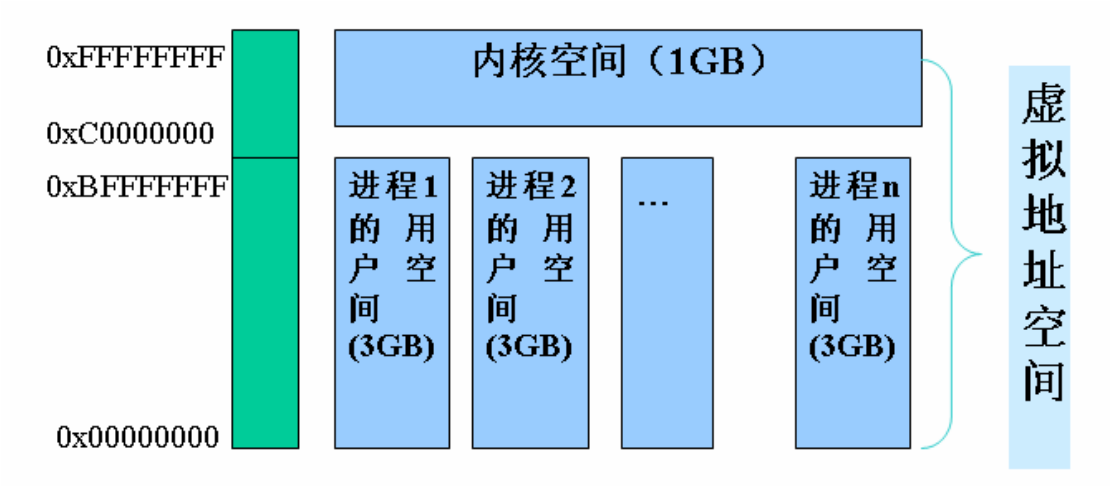
这里可能有人不禁问，为什么堆和栈的内存权限是可执行的呢？

关于堆栈段是否具有可执行功能，linux 的版本之间并没有严格的规定。有些是可以在堆栈中加一些可执行的代码的。

问题二：有个程序员朋友禁不住跳出来要问，在 32 位操作系统中，每个进程拥有 4G 的虚拟内存空间，既然栈是从顶端向下增长，那么栈顶的地址应该是 0xff000000，而不是 0xbf000000。

我们先看看 0xbf000000 是多少？转化成 10 进制 3,000,000,00。不多不少整整 3G。

每个进程通过系统调用进入内核，Linux 内核空间由系统内的所有进程共享。从进程的角度来看，每个进程拥有 4GB 的虚拟地址空间(也叫虚拟内存)。每个进程有各自的私有用户空间（0-3G），这个空间对系统中的其他进程是不可见的。最高的 1GB 内核空间则为所有进程以及内核所共享。



虚拟内存一共 4G 字节，分为两部分：

内核空间（最高的 1G 字节）

用户空间（较低的 3G 字节）

因此，更确切的说每个进程最大拥有 3G 字节私有虚存内存空间。

问题三：我们知道堆地址是向上增长，而在 maps 中

00011000-00032000 rwxp 00011000 00:00 0　堆地址

40000000-40002000 rw-p 40000000 00:00 0　共享库的一段地址

如果堆地址一直增长到 40000000 的话，堆该怎么办？

这的确是个问题，看来我不得不在埋伏一个伏笔，再后面讲到进程堆段内存管理时，会详细说明。

现在我们来回答 Hello 占用了 1380K 内存的问题：

41000000-41017000 r-xp 00000000 1f:0d 817360　　/lib/ld-2.3.3.so　（92K）

41028000-41120000 r-xp 00000000 1f:0d 817593　　/lib/libc-2.3.3.so　（992K）

也就是说光这两个库的代码段就占用了 92+992=1084K 内存。所以从 statm 得出来的内存占用量，包含了其依赖动态库占用的内存，不准确，几乎没有什么参考价值。

以上涉及的全部为虚拟内存的概念，不涉及物理内存的使用。

现在我们了解了一个进程的虚拟内存情况，那么我们能不能得知其物理内存使用情况呢？

可以，通过 memmap 文件，来了解虚拟内存每一个页面与物理内存对应情况。

cat memmap

2

1

1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1 1

494949494949494949491649494949494949494949494949

1 1

4917171717171116171717171717171717174949 0 049 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 049 0 0 0 0 0 0 0 0 0 0 044 042424242 0 0 0 041 0 0 0 0 0 0 0 0 0 0 0 0 0 0 04847 0 0 0 0 0 0 0 04747484949494949494949 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 049 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 04849 0 0 0 04849 0 0 047 049 0 0 0 0 0 0 0 0 0 0 04949 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 046 0 0 0 0 0 0 0 0 0 0 0 0 0 0 044 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0

49

1 1 1

1 1

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1

比较晕，满篇不知所云的数字。

解析这个文件的内容，你需要和 maps 文件结合来看，在 memmap 中的每一行与 maps 中的每一行顺序对应，比如：

memmap 的第一行：

2

对应于 maps 的第一行：

00008000-00009000 r-xp 00000000 1f:12 288 /mnt/msc_int0/hello

memmap 的第二行：

1

对应于 maps 的第二行：

00010000-00011000 rw-p 00000000 1f:12 288 /mnt/msc_int0/hello

等等。

在 memmap 中每两个字符对应于虚拟内存中的一个页面（4K）。如果这两个字符为 0，则代表该页面空闲，系统没有为其分配虚拟内存；如果这两个字符有数值，则代表系统为其分配了物理内存，并有多少个进程引用了该页面。

我们注意到在 maps 中，堆段为：

00011000-00032000 rwxp 00011000 00:00 0　系统为其分配了 132K 的虚拟地址，那实际使用了多少物理内存呢。

堆段在 memmap 中对应于：

1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

其正好有 26 个字符，对应于 maps 中的 132K 虚拟地址，并且表示系统只为该堆分配了一个物理页面，也就是说堆占用了 4K 的物理内存，后面的 128 虚拟内存并没有被使用。

注意：memmap 并不是每一个 linux 版本都具备的，如果没有的话，你只能通过书写自己的驱动程序，来进程 Linux 内核空间，才能获取相应信息。

现在我们再来关注一下，ld-2.3.3.so 和 libc-2.3.3.so 代码段物理内存使用情况。实际上在我们的程序中，几乎没有使用什么 libc 函数，但其却占用了相当多的物理内存，这是为什么？

代码段是整个系统共享的，也就是说其他进程运行指令，占用了代码段物理内存，也会反映到我们的测试程序上，实际上其本身并没有使用那么多内存，大部分都是其他进程使用的。因此，从 memmap 中，统计代码段所占物理内存，将是非常不准确的，没有什么实际意义。

而数据段、堆段、栈段则是每个进程私有的，所以通过 memmap 统计出来的结果是准确的。

正是由于代码段这种系统共享的特性，导致了我们想了解一个进程使用了多少内存，给我一个数，这样一个简单的要求，变得很复杂，几乎不可能。

忙活了半天，我们还是没有找到一种合理的评测进程使用内存的机制，没法向老板交代啊。

研究了半天内存的使用，下面我们说说内存的回收。

2.2.3. 内存回收

在 Linux 中，随着程序的运行，其代码段所占的内存将越来越多，有可能只是运行了一次，以后再也不用了，难道这块内存就再也没办法释放了吗？

Linux 针对于这种情况，给出了自己的解决方案：Linux 内存回收机制。
在 Linux 系统中，你可以找到一个守护进程 kswapd，它会定期的检查系统中空闲内存的数量，一旦发现空闲内存数量小于一个阀值的时候，就会将若干页面换出。熟悉 Linux 系统的

人可能马上就想起了交换分区，在我们安装 Linux 系统时都要设置这样的一个分区。

同时我们又马上会发现，在我们的嵌入式设备中，居然没有交换分区。天啊，为什么呢，难道我们的系统最多只能使用现有的物理内存吗，这未免也太弱智了吧。

我想，这可能有两方面的原因：
1、 一旦使用了交换分区，系统的性能将下降的很快。
2、 我们现在的嵌入式设备一般使用 Flash 做为存储介质，而 Flash 写的次数是有限的。如果在 Flash 上面建立交换分区的话，必然导致对 Flash 的频繁写，进而影响 Flash 的寿命。

那么没有交换分区，Linux 怎么做内存回收呢？
对于那些没有改写的页面，那么这块内存就不需要写到交换分区上，直接回收。
对于那些已经改写了的页面，没办法，只能保留在系统中。

在 Linux 物理内存中，每个页面有一个 dirty 的标志，如果该页面被改写了，我们称之为 dirty page。总的来说，所有非 dirty page 的物理页面都可以被回收。

让我们来看看进程中各个段的 dirty page 情况：
1、代码段，其权限是只读属性，不可能被改写，所以其所占的物理内存，全部不是 dirty page。
2、数据段，其权限是可读、可写，所以其所占的物理内存，可能是 dirty page，也可能不是 dirty page。
3、堆段，其没有对应的映射文件，内容都是通过程序改写的，所以其所占的物理内存，全部是 dirty page。
4、栈段，和堆段相同，其所占的物理内存，全部是 dirty page。

因此，代码段所占的物理内存全部可以回收，堆、栈段所占的物理内存全部不能回收，数据段所占的物理内存有的可以回收，有的不可以。

从这个角度上来看，一个进程所占的 dirty page，是一个非常适合做为评价一个进程所占物理内存的指标，它消除了由代码段系统共享所带来的影响，终于可以向老板交差了。

但是一个物理页面是否为 dirty page 不能从 proc 目录下的 maps 和 memmap 文件中获得，我们不得不编写一个模块，加载到 Linux 内核中，从而获得进程中每一个页面的 dirty 属性。这个内容不再本书范围之内。

第二篇 进程内存优化

我终于能拿出了一份了可以令众人信服的报告，老板看了之后也很满意。很快，老板又召集了一个会议，把各个组的组长都叫了过来，讨论这份报告。

现在我们对每个进程占据了多少内存很清楚了，问题是谁对系统内存减少负责？
让某一个 daemon 进程负责，不太可能，毕竟系统中所有的 daemon 进程都有份。

那么每个进程要减多少呢？老板需要一个硬指标，每个团队也需要一个硬指标。
可这个指标实在是难出，每个进程的情况不一样，其内存挖掘的潜力也不同，连开发团队在这方面都不清楚，更何况老板了。

可从前一段各个团队优化内存的情况来看，进展缓慢，如果没有一个指标，没有一个压力，这个事情就很难推动。

最后敲定两条标准：
1、 所有 daemon 进程所占内存只能比我们发布的上一个版本所占的内存少，不能多。
2、 Dirty Page 排在前 10 的 10 个进程，dirty page 要减少 20%。

这个结果，可以说几家欢乐，几家愁啊，虽然有可能误杀，但是命中率还是蛮高的啊。

下一步，我们就要开始真正的内存优化之旅了，我们的程序员朋友有些等不及了吧。
要想做内存优化，就要首先清楚都哪些因素对内存使用。为此，我将分为 3 个部分对其进行说明。
1、 进程所占用的内存（这里的进程，特指不包括动态库、线程）
2、 动态库对内存的影响
3、 线程对内存的影响。

这里我们所指的进程，指进程中不包含动态库的动态库的部分，动态库对进程内存的影响我们将在下一章节详细描述。

2.3. 进程

在前面我们提到了代码段、数据段、堆段和栈段，那么有些人不禁要问，这个我们写的程序有什么关系呢？

下面我将举一个例子，说明程序中各个元素所处内存的位置。

代码段、数据段、堆段和栈段，是我们从 Linux 内核的角度来看待我们写的程序，现在我们从程序员的角度来划分各个元素，对齐进行说明。

一个进程运行时，所占用的内存，可以分为如下几个部分：
1、栈区（stack）：由编译器自动分配释放，存放函数的参数值，局部变量的值等。
2、堆区（heap）：一般由程序员分配释放，若程序员不释放，程序结束时可能由 OS 回收。
3、全局变量、静态变量：初始化的全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。程序结束后有系统释放。
4、文字常量：常量字符串就是放在这里的，程序结束后由系统释放。
5、程序代码：存放函数体的二进制代码。

下面我们看一个例子程序：

```
//main.cpp
int a = 0; 全局初始化区
char *p1; 全局未初始化区
main()
{
        int b; 栈
        char s[] = "abc"; 栈
        char *p2; 栈
        char *p3 = "123456"; 123456\0 在常量区，p3 在栈上。
        static int c =0；全局（静态）初始化区
        p1 = (char *)malloc(10);
        p2 = (char *)malloc(20);
        分配得来得 10 和 20 字节的区域就在堆区。
        strcpy(p1, "123456"); 123456\0 放在常量区，编译器可能会将它与 p3 所指向的"123456"
优化成一个地方。
}
```

程序中的栈区对应于进程的栈段；
程序中的堆区对应于进程的堆段；
全局变量、静态变量因为其是可读写的，故存储在进程的数据段；
文字常量、程序代码因为其是只读的，故存储在进程的代码段。

下面几章，我们将详细的介绍进程中堆段、栈段、数据段、代码段的特性。

2.3.1. 堆段

一提到堆段，相信很多人立即会想到 C 语言的 malloc、C++中的 new，以及让人头疼的内存泄漏。
malloc、free、new、delete 是我们从程序的角度去看待堆内存；而在 Linux 内核中会专门为进程分配一段内存地址，用来存放堆的内容；随着进程申请内存的增加，进程会通过系统调用 brk，来让 Linux 内核扩展这段内存空间，从而分配给进程更多的内存；当进程释放内存时，进程又会通过系统调用 brk，来告诉内核缩减这段内存空间，Linux 内核便会将其一部分物理内存进行回收。

那么是不是我们的程序每一次调用 malloc、new 时，都回去调用系统调用 brk 调整堆顶地址呢？

不会，因为它面对着两个难题：
1、 在用户态进程申请内存是以字节为单位，而在内核中内存的管理是以页面（4K）为单位，这中间存在一个差距。
2、 太多的 brk 系统调用，会使进程的速度变得很慢。

我们再来看下面的一个例子：

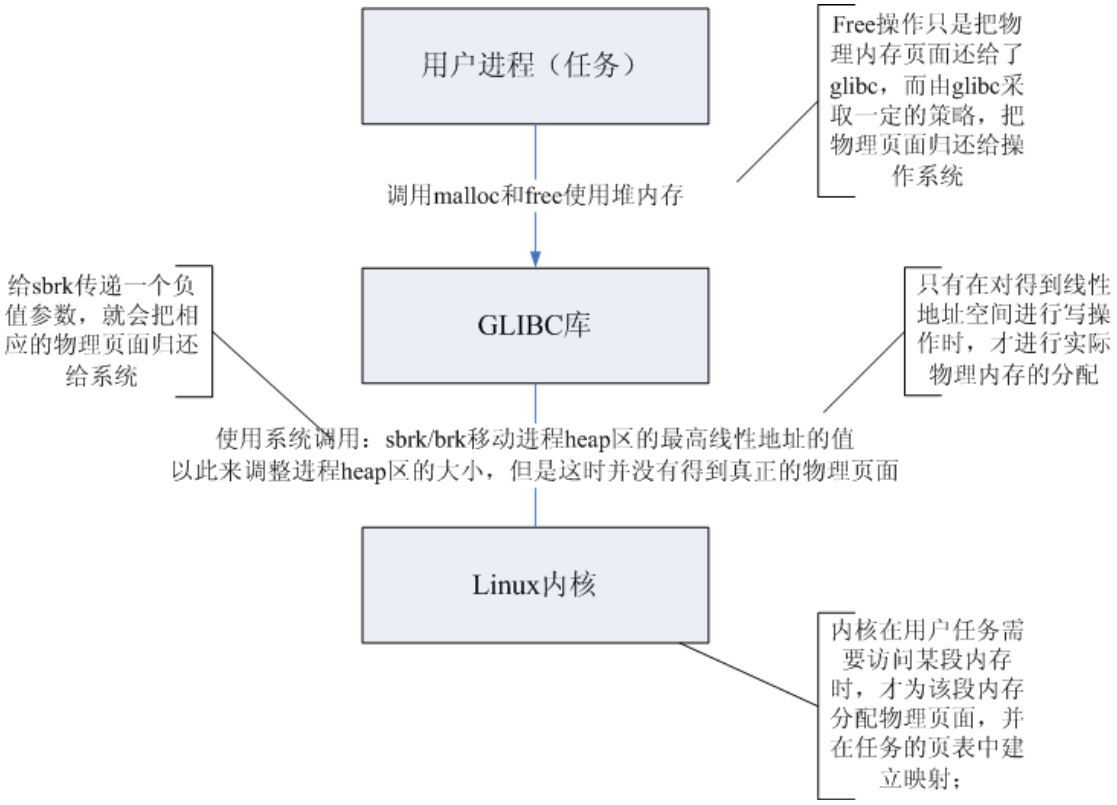
```
#include <stdlib.h>
#include <stdio.h>


int main()
{
1     char*    p;
2     p = malloc(20);
3     strcpy(p,”Hello,world!”);
      printf(“%s\n”,p);
      free(p);
      return 0;
}
```

程序很简单，就是从堆中申请了 20 个字节。

为了解决前面所提到了两个难题，我们需要一个代理的机制，它所要完成的工作：
1、 接收程序的内存请求，将其积少成多，然后统一向 Linux 内核申请内存。
2、 要想办法避免频繁的系统调用导致进程速度变慢。
3、 减少内存碎片的产生。

要想同时达到上述几个目标，是相当困难的，需要我们在效率与空间方面做出平衡。目前也有很多聪明人在从事相关的研究工作。编写内存管理已经超出了本书的内容，在此我们不做描述。

我们还是回到上面的例子，在程序调用 malloc 的时候，实际上在 libc 内置的内存管理器接收了该请求，具体的流程可以参考下图：



问题一：为什么在 malloc 时，要求输入内存区域大小；而在 free 时，不需要输入内存区域大小？

首先，当我们调用函数 malloc 申请内存时，并不是直接向操作系统申请，而是 glibc 自己所做的堆管理。也许有的人知道，在(p-4)这个地址，就是记录着 malloc 空间的大小，glibc 在 free 这块内存的时候，在（p-4）的这个地方，获得分配内存的大小，真的这样吗？让我们来看一看：

printf(“0x%x\n”,*(p-4);
结果: 0x19
0x19 换算成 10 进制，应该为 25。为什么在 glibc 中记录的是 25，而不是 20 呢？

在 malloc 函数内部定义了一个结构 malloc_chunk 来定义 malloc 分配或释放的内存块。

struct malloc_chunk

{
    INTERNAL_SIZE_T prev_size;
    INTERNAL_SIZE_T size;
    标志位；
    用户数据；
}
具体格式

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

上一个块的大小（prev_size）			
当前块大小(size)		M	P
H	e	l	l
o	,	w	o

注意这里面有两个标志位：

P=1　　表示上一块正在被使用，这时 prev_size 通常为 0；
P=0　　表示上一块空闲，这时 prev_size 通常为上一块的大小。
M=1　　表示该内存块通过 mmap 来分配，只有在分配大块内存时，才采用 mmap 的方式，那么在释放时会由 munmap_chunk()去释放;否则,释放时由 chunk_free()完成.
M=0　　则表示该内存块不采用 mmap 方式分配。

由于 malloc 实现中是 8 字节对齐的,size 的低 3 位总是不会被使用的,所以在实际计算 chunk 大小时,要去掉标志位.

那么我们去掉 0x19 的低三位，则其为 0x18,十进制为 24，还是不等于 20，为什么呢？

还记得上面说的吗，malloc 实现的是 8 字节对齐，那么 20 个字节，如果实现 8 字节对齐的话，正好等于 24 个字节。

补充一点: 一次 malloc 最小分配的长度至少为 16 字节。因此每次只申请 1、2 个字节的操作是十分浪费的。

问题二：回到前面的一个问题

00011000-00032000 rwxp 00011000 00:00 0　堆地址

40000000-40002000 rw-p 40000000 00:00 0　共享库的一段地址

如果堆地址一直增长到 40000000 的话，堆该怎么办？

40000000 这个地址有多大？1G。实际上在 glibc 的内存管理中，采用 brk 的方式，只能管理 1G 地址空间以下的内存，如果大于了 1G，glibc 将采用 mmap 的方式，为堆申请一块内存。

针对堆的管理，内核提供了两个系统调用 brk 和 mmap。brk 用于更改堆顶地址，而 mmap 则为进程分配一块虚拟地址空间。

2.3.1.1. 小块内存分配

我们先来看一下下面的例子：

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

int main()
```

```
{
    mallopt(M_MXFAST,0);
    char *p=malloc(10);
    char *p1=malloc(10);
    char *p2=malloc(20);
    printf("%p    %p    %p\n",p,p1,p2);
    free(p);
    free(p1);
    char *p3=malloc(20);
    printf("%p\n",p3);
    return 0;
}
```

上面的例子中，我们先为 p1 和 p2 分别分配了 10 个字节，然后又将其释放掉，这块内存被回收，然后我们又为 p3 分配了 20 个字节，按理来说 p3 应该使用 p 和 p1 释放的 20 个字节，p3 的地址应该和 p 的地址一样。

让我们来看一下结果：

```
# ./hello
0x11050    0x11060    0x11070
0x11088
```

p3 的地址是 0x11088，p 的地址为 0x11050，也就是 p3 并没有使用由 p 和 p1 释放而节省出来的内存，而选择了在堆顶新申请了一块，这是为什么呢？

我们知道在 glibc 的内存管理中，跟踪每一块内存的分配和分配，为了减少内存碎片，还会使用一定的算法，对相邻的空闲内存进行合并。为了提高速度，glibc 的 malloc 实现了 fastbins，对于一些小块的内存不去尝试合并以节省 CPU 和内存（因为跟踪内存是需要代价的），我们可以通过函数来设置小块内存的阀值。

```
#include <malloc.h>
int mallopt (int param, int value)
```

param 的取值分别为 M_MXFAST。

value 是以字节为单位。

M_MXFAST

定义 fastbins 的小块内存阀值，小于该阀值的小块空闲内存将不会去尝试合并，其缺省值为 64。

下面我们来将 M_MXFAST 设置为 0，禁止掉 fastbins，使其跟踪每一个小块内存并尝试进行合并。

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

int main()
```

```
{
	mallopt(M_MXFAST,0);
	char *p=malloc(10);
	char *p1=malloc(10);
	char *p2=malloc(20);
	printf("%p	%p	%p\n",p,p1,p2);
	free(p);
	free(p1);
	char *p3=malloc(20);
	printf("%p\n",p3);
	return 0;
}
```

我们来看下结果：

```
# ./hello1
0x11050	0x11060	0x11070
0x11050
```

果真 p3 复用了 p 和 p1 释放的内存。

那是不是说小于 M_MXFAST 的小块释放内存都不能复用了呢？
我们将 p3 申请的内存从 20 减到 10：

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

int main()
{
	mallopt(M_MXFAST,0);
	char *p=malloc(10);
	char *p1=malloc(10);
	char *p2=malloc(20);
	printf("%p	%p	%p\n",p,p1,p2);
	free(p);
	free(p1);
	char *p3=malloc(10);
	printf("%p\n",p3);
	return 0;
}
```

```
# ./hello2
0x11050	0x11060	0x11070
0x11050
```

实际上 p3 还是复用了 p 所释放的内存。

这里我们来总结一下：
1、 我们可以使用 mallopt 来设置 M_MXFAST 的阀值。
2、 对于所有小于 M_MXFAST 阀值的小块内存释放时，不会去尝试合并，还会被复用。

那么从内存使用的角度，M_MXFAST 到底是节省内存呢，还是会浪费内存呢？
这里要考虑两个因素：
1、 M_MXFAST 由于对许多小块内存不会去尝试合并，因此内存管理器就可以节省出跟踪每小块内存的数据结构，从而节省内存。
2、 由于小块内存不能进行合并，会增加进程中的内存碎片，从而占用更多的内存。

总体来说，M_MXFAST 对于内存使用的影响没有一个定论，但它肯定会加快进行的运行速度。

2.3.1.2. 大块内存分配

当进程向 glibc 申请内存时，如果申请内存的数量大于一个阀值的时候，glibc 会采用 mmaps 为进程分配一块虚拟地址空间，而不是采用 brk，来扩展栈顶的指针。
例如
char *p=malloc(1024*521);

通过查看 maps 文件，你可以发现增加一段线性区。
请注意：这块线性区的权限可以通过 maps 来查看，其权限为 rw-p。其与进程缺省堆内存的权限 rwxp 是不同的。而有些内存统计工具是根据这个权限来判断内存的属性，例如 maps_parser，会导致统计的失误。

而我们在释放这块内存的时候:
free(p)
glibc 将通过系统调用 unmmap 的方式，告诉内核来删除这段内存空间。

这里面提到的阀值，在缺省情况下是 128K，这个值的大小你可以通过函数来设置。
#include <malloc.h>
int mallopt (int param, int value)
param 的取值分别为 M_MMAP_THRESHOLD、M_MMAP_MAX。
value 是以字节为单位。

M_MMAP_THRESHOLD
libc中大块内存阀值，大于该阀值的内存申请，内存管理器将使用mmap系统调用申请内存；如果小于该阀值的内存申请，内存管理其使用brk系统调用来扩展堆顶指针。该阀值缺省值为128kB。

M_MMAP_MAX

该进程中最多使用mmap分配地址段的数量。

2.3.1.3. 内存释放

现在我们了解了在 libc 中有个内存管理，会接收程序内存申请、释放的请求，那么当我们 free 掉一块内存时，内存管理其也不会很简单就把这块内存归还给系统。

我们再来看一个例子

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

int main()
{
	char *p[11];
	int i;
	for(i=0;i<11;i++)
	{
		p[i]=(char *)malloc(1024*2);
		strcpy(p[i],"123");
	}

	for(i=10;i>0;i--)
	{
		free(p[i]);
	}
	pid_t pid=getpid();
	printf("pid:%d\n",pid);
	pause();

	return 0;
}
```

上面的例子，我们连续申请了 11 块 2K 大的内存，然后释放了的 10 块 2K 的内存，那么现在的堆应该还剩下 2K 的内存。

我们可以通过/proc/{pid}/memmap 来验证：

```
#cat memmap
1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

结果却发现进程占用了 6 个物理页面，也就是 24k 物理内存！

那些内存我明明 free 释放掉了，为什么进程没有归还给系统呢？

你有些惊讶，不敢相信自己的眼睛，原来构造的内存管理理念似乎将要被颠覆。

我们试想下面一种情况，如果一个进程频繁的申请、释放一个页面的话，必然会导致大量的系统调用，从而进程的效率。如果内存管理在进程释放一块内存时，不是直接返还给系统，而是将其 cache 住，留待下次分配使用，这样就可以减少系统调用的次数，从而提高进程的效率。它的代价是释放的内存不立刻返还给系统，以内存空间来换取进程的性能。

在 libc 中，只有当堆顶有连续的 128k 空闲内存时，libc 才会掉用 brk，来通知内核释放这段内存，将其返还给系统。

我们再来看看上面的 memmap 信息：
1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

总共有 33 个页面，共 132kB，在前 6 个页面中，第 1 个页面程序中还在使用；后面 5 个页面程序虽然释放了，但由于刚好没有满足释放内存的条件，所以这 20K 的物理内存还保留在进程中，以便下次使用。

堆顶 128K 空闲内存对于嵌入式设备来讲，未免有些苛刻。我们可以通过 mallopt，来修改这个阀值。

#include <malloc.h>
int mallopt (int param, int value)
param 的取值分别为 M_TRIM_THRESHOLD、M_TOP_PAD。
value 是以字节为单位。

M_TRIM_THRESHOLD
　　堆顶内存回收阀值，当堆顶连续空闲内存数量大于该阀值时，libc 的内存管理其将调用系统调用 brk，来调整堆顶地址，释放内存。该值缺省为 128k。

M_TOP_PAD
　　该参数决定了，当 libc 内存管理器调用 brk 释放内存时，堆顶还需要保留的空闲内存数量。该值缺省为 0.

这里我还要介绍另外一个参数 TRIM_FASTBINS：当释放一个小块内存时候，是否立即对 fastbin 进行合并；设置为 1 会进行立即合并可以减少内存消耗，但降低分配释放效率。

TRIM_FASTBINS 与堆顶内存回收存在着一个接口：在 TRIM_FASTBINS=0，当小于或者等于 MXFAST 的小块内存释放时，并不会触发堆段顶端内存释放，堆顶内存释放被延迟到大于 MXFAST 的内存释放时触发，这样有可能会增加内存的碎片；在 TRIM_FASTBINS=1 时，小于 MXFAST 的小块内存释放，也将触发堆顶内存释放。

TRIM_FASTBINS 不是一个可以配置的参数，它是一个编译时的预定义变量，在编译 Libc 库时，已经确定了，缺省为 0。

如果我们想使小块内存释放也会触发堆顶内存释放，有两种方法：
1、 加上 DTRIM_FASTBINS=1，来重新编译 libc 库。
2、 我们可以调整 MXFAST 的值，将其设置为 0，使得所有内存分配的大小都将大于 MXFAST；

2.3.1.4. 内存空洞

现在我们知道堆内存释放，是从堆顶开始。那么如果堆中间的一块区域，大部分内存都释放了，堆顶还有一些会怎么样呢？

我们来看个例子：

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

int main()
{
	char *p[11];
	int i;
	for(i=0;i<10;i++)
	{
		p[i]=(char *)malloc(1024*2);
		strcpy(p[i],"123");
	}
	p[10]=(char *)malloc(1024*2);
	strcpy(p[10],"123");

	for(i=0;i<10;i++)
	{
		free(p[i]);
	}
	pid_t pid=getpid();
	printf("pid:%d\n",pid);
	pause();

	return 0;
}
```

这里我申请了 11 个 2k 的内存，然后将前 10 个 2k 内存释放掉，只剩下堆顶一个 2k。

按我们正常的逻辑，进程应该只是占用了一个物理页面，可实际上呢？
我们查看 memmap 的结果，可以看到
1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

注意：这和我们上一章的例子还不一样。上一章的例子，内存是从堆顶开始释放，因为没有满足连续 128k 空闲内存，而没能真正释放，返还给系统。这个例子是，堆顶指针没变，堆顶下面的内存被释放了。

让我们来想想 libc 会怎么办？
第一个想法，也是最合理的方法，堆顶下面哪个页面空闲了，就将那个物理页面释放掉，返还给系统。

Linux 内核只能通过缩小线性内存区的方式来释放物理内存。
方法一，通过使用系统调用 brk，来改变堆顶地址释放内存。
优点：
　　算法简单，系统调用少，效率高。
缺点：
　　堆顶下方的物理页面即使空闲也无法及时释放。

方法二：，通过将对应堆的线性区拆分，将中间的物理页面释放掉。
优点：
　　堆顶下方的内存能够得到即使的释放。
缺点：
　　算法复杂，涉及到线性区的拆分与合并，有可能会导致进程堆段形成多个不连续的小块内存空间，对进程的性能影响较大。

综合以上因素，Linux 内核选则了通过调整堆顶来扩展和释放内存空间。

它也决定了，只要堆顶部还有内存在使用，堆顶下方不管释放了多少内存都不会被释放，这也就是我们经常所说的内存空洞。

除了通过 brk 扩展堆顶地址外，我们还提到了另外一种内存分配方式 mmap。当 libc 在处理大块内存分配时，其会调用 mmap 来分配一块地址空间；当释放时，直接调用 unmmap 释放掉该段内存空间，因此对于这种大块内存分配的申请和释放，就不会存在内存空洞的问题。你可以通过使用 mmap 分配内存的阀值，来减少内存空洞的概率。代价是，可能会使用更多的系统调用，降低进程的性能。

要想消除内存空洞的影响，就要求我们在申请和释放内存时，要严格依照就近原则，最先释放堆顶地址的内存。可控制内存的申请和释放的顺序难度十分的大；另外由于内存碎片的影响，每次申请得到的内存地址都带有一定的随机性，后面申请的内存，并不一定就意味着在堆顶；这简直是 Mission impossible。

做为一个程序员，我很受挫折，我无能为力，我调用 free 都释放内存，可它并不一定会返还到系统中，我无法完全控制我程序的行为，谁又能保证堆顶没有那么一块正在使用的内存

呢？

这也给很多程序员以藉口，在测试中我们程序内存使用量增长，要求他们去检查时，他们往往会说，“内存我都释放了，这是内存空洞造成的，我也无能为力。”。老板也无话可说，内存空洞简直成了我们的噩梦。

内存空洞和内存泄漏造成的内存增长是有区别的：
内存泄漏是申请了的内存没有释放，如果你在做多次同样的操作，进程所使用的内存应该保持同样的一个速度进行增长。
内存空洞是申请并释放了的内存由于不处于堆顶无法返还给系统，但是这些内存还是能留给进程自身使用，所以如果你做多次同样的操作，进程所使用的内存应该停留在一个水平线上，不怎么增长，或者增长不多。

程序员可以根据这个现象来判断，你的进程中存在的是内存泄漏还是内存空洞。

在实际情况中，内存空洞的现象并很多，通过我的观察在堆中，更多的是内存的碎片而不是内存空洞。因此程序员对于内存空洞只要了解这个概念，在申请分配内存时，本着就近原则就可以了，需要的时候才分配内存，不需要了立刻释放。不必去严格的追求申请和释放的顺序，也做不到。

后面我将会介绍一种方法，通过它你可以观察当前进程的内存分布情况，你可以清楚的知道进程内存碎片和内存空洞的情况。

虽然我给大家吃了很多宽心丸，但很多人还是不放心，担心他自己成为内存空洞的受害者。难道我们对于内存空洞真的是无能为力吗？

实际上堆的内存管理机制有很多种，你也可以按照自己的要求来自己编写内存管理机制。前面我们介绍的是 libc 中自带的内存管理机制 Doug Lea Malloc。
Doug Lea Malloc：Doug Lea Malloc 实际上是完整的一组分配程序，其中包括 Doug Lea 的原始分配程序，GNU libc 分配程序和 ptmalloc。 Doug Lea 的分配程序加入了索引，这使得搜索速度更快，并且可以将多个没有被使用的块组合为一个大的块。它还支持缓存，以便更快地再次使用最近释放的内存。

这里我简单介绍另外一种内存管理机制 BSD Malloc。
BSD Malloc：BSD Malloc 是随 4.2 BSD 发行的实现，包含在 FreeBSD 之中，这个分配程序可以从预先确定大小的对象构成的池中分配对象。它有一些用于对象大小的 size 类，这些对象的大小为 2 的若干次幂减去某一常数。所以，如果您请求给定大小的一个对象，它就简单地分配一个与之匹配的 size 类。这样就提供了一个快速的实现，但是可能会浪费内存。
据说这种算法可以消除内存空洞，因为它把原来的一个内存段，编程了多个内存段，从而减小了内存空洞的影响，而性能下降不多。
有人尝试着将其移植到嵌入式 Linux 设备上，据说每个进程的内存的堆可以减少 1/3，但是否有负作用不清楚，有待检验。有兴趣的朋友可以尝试一下。

2.3.1.5. 堆内存管理器参数总结

在进程中，我们可以调用 mallopt 函数，来调整 libc 的内存管理器的行为。

#include <malloc.h>
int mallopt (int param, int value)

value 的单位是字节。
param 的取值范围是：M_TRIM_THRESHOLD、M_TOP_PAD、M_MMAP_THRESHOLD、M_MMAP_MAX。

M_TRIM_THRESHOLD
堆顶内存回收阀值，当堆顶连续空闲内存数量大于该阀值时，libc 的内存管理其将调用系统调用 brk，来调整堆顶地址，释放内存。该值缺省为 128k。
M_TOP_PAD
该参数决定了，当 libc 内存管理器调用 brk 释放内存时，堆顶还需要保留的空闲内存数量。该值缺省为 0.
M_MMAP_THRESHOLD
libc中大块内存阀值，大于该阀值的内存申请，内存管理器将使用mmap系统调用申请内存；如果小于该阀值的内存申请，内存管理其使用brk系统调用来扩展堆顶指针。该阀值缺省值为128kB。
M_MMAP_MAX
该进程中最多使用mmap分配地址段的数量。

2.3.1.6. 内存的跟踪

使用 mtrace 检测内存泄漏
1、 引入头文件 #include <mcheck.h>
2、 在需要跟踪的程序中需要包含头文件，而且在 main()函数的最开始包含一个函数调用：mtrace()。由于在 main 函数的最开头调用了 mtrace()，所以该进程后面的一切分配和释放内存的操作都可以由 mtrace 来跟踪和分析。
3、 定义一个环境变量，用来指示一个文件。该文件用来输出 log 信息。如下的例子：
$export MALLOC_TRACE=mymemory.log
4、 正常运行程序。此时程序中的关于内存分配和释放的操作都可以记录下来。

例子：

```
#include <stdlib.h>
#include <stdio.h>
#include <mcheck.h>
int main()
{
    mtrace();
```

```
    char *p=(char *)malloc(10);
    malloc(20);
    free(p);
    muntrace();
    malloc(30);
}

# export MALLOC_TRACE=a.log
#./hello
#cat a.log
= Start
@ ./hello:(malloc+0xf4)[0x84c4] + 0x113d8 0xa
@ ./hello:(malloc+0x100)[0x84d0] + 0x113e8 0x14
@ ./hello:(malloc+0x108)[0x84d8] - 0x113d8
= End
```

为了方便调试，glibc 为用户提供了 malloc 等函数的钩子函数，这些函授都在<malloc.h>中声明。

Variable: **__malloc_hook**

```
void *function (size_t size, const void *caller)
```

Variable: **__realloc_hook**

```
void *function (void *ptr, size_t size, const void *caller)
```

Variable: **__free_hook**

```
void function (void *ptr, const void *caller)
```

Variable: **__memalign_hook**

```
void *function (size_t alignment, size_t size, const void *caller)
```

```
    #include <malloc.h>

    static void my_init_hook (void);
    static void *my_malloc_hook (size_t, const void *);
    static void my_free_hook (void*, const void *);

    void (*__malloc_initialize_hook) (void) = my_init_hook;

    static void
    my_init_hook (void)
    {
      old_malloc_hook = __malloc_hook;
      old_free_hook = __free_hook;
      __malloc_hook = my_malloc_hook;
```

```
    __free_hook = my_free_hook;
}

static void
my_malloc_hook (size_t size, const void *caller)
{
    printf ("malloc (%u) returns %p\n", (unsigned int) size, result);
}

static void
my_free_hook (void *ptr, const void *caller)
{
    printf ("freed pointer %p\n", ptr);
}

main ()
{
    ...
}
```

详细参见：http://www.gnu.org/software/libc/manual/html_node/Hooks-for-Malloc.html

2.3.1.7. 堆内存优化

关于进程堆段内存基本介绍到这里，这里给出一些建议：

1、 堆内存的最小单位为 16 个字节，所以尽量减少小块内存的申请，避免内存浪费。
2、 调整 M_MMAP_THRESHOLD，降低 mmap 的门槛，会降低内存空洞的风险，但也会增加系统调用，降低性能。
3、 调整 M_TRIM_THRESHOLD，减少堆顶连续内存门槛，释放更多的堆顶内存。

2.3.2. 栈

前面我们讲述堆的管理，下面我们来说下栈：我们知道栈中的内存是由程序自动来维护，栈段内存紧密排列，不会出现内存碎片的问题；不需要手动的来申请和释放。在进程进入函数时，会自动的将参数和局部变量加入栈中，而在函数返回时，会自动将这块内存返还给系统，很省心，不必再存在内存泄漏的问题。

2.3.2.1. 栈上申请内存

大家都知道动态分配的内存，一定要释放掉，否则就会有内存泄露。可能鲜有人知，动态分配的内存，可以不用释放。alloca 就是这样一个函数，最后一个 a 代表 auto，即自动释放的意思。

alloca 是在栈中分配内存的。即然是在栈中分配，就像其它在栈中分配的临时变量一样，在当前函数调用完成时，这块内存自动释放。

```
#include <stdio.h>
int main(intargc, char* argv[])
{
	int n = 0;
	int* p = alloca(1024);
	printf("&n=%p p=%p\n", &n, p);
	return 0;
}
```

在栈上分配内存的好处，就在于不会有内存泄漏的问题。

使用 alloca 在栈中分配内存时，在复制时所占用的物理内存与在函数中通过对临时变量复值所得到的结果是不同的。

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

int main()
{
	int i;
	char *p;
	for(i=0;i<1024*10;i++)
	{
		p=(char *)alloca(1024);
		printf("%p\n",p);
	}
	pid_t pid=getpid();
	printf("pid:%d\n",pid);
	pause();
	return 0;
}
```

查看其栈段，虚拟空间 be5eb000-bf000000 rwxp be5eb000 00:00 0 扩展到了 10M，但是通过

memmap，发现这 10M 空间，全部占用了物理内存。

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

int a()
{
        char p[1024*1024];
        pause();
}
int main()
{
        pid_t pid=getpid();
        printf("pid:%d\n",pid);
        a();

        return 0;
}
```

其所对应的栈的物理内存

```
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
```

只是在栈顶用了一个物理页面。

现在的问题是 bf000000,是指 start_stack 吗？

```
int a()
{
        char p[1024*1024];
        memset(p,'a',1024*10);
        pause();
}
int main()
{
        pid_t pid=getpid();
        printf("pid:%d\n",pid);
        a();
        return 0;
}
```

其所对应的物理内存为

```
1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
其说明对于栈来讲，使用了多少内存，就占用多少物理页面，这与堆来讲是不一样的。

2.3.2.2. 栈的扩展

在前面我们谈到，进程通过系统调用 brk 和 sbrk 调整堆顶的地址，来扩展或释放堆段内存，那么我们很自然会想到栈段是如何来扩展或释放其内存呢？
这个问题扩展了我很长时间，我查阅了 Linux 中所有的系统调用，结果没有发现一个与栈段有关的。难道 Linux 的栈段不是通过系统调用来扩展的吗？那么它是如何来申请和释放物理内存的呢？

一个偶然的机会，我找到了答案：栈需要多少空间，就给多少空间，不需要通过系统调用去扩展栈顶指针。当进程采取压栈动作后，栈顶指针减小，如果进程访问相应内存时，会触发页故障。

下面我们来看一下 Linux 内核中，处理页故障的函数。

```
asmlinkage void do_page_fault(struct pt_regs *regs, unsigned long error_code)
{
    vma = find_vma(mm, address);   /*  查找 address < vm_end 的线性区,如果没有返回空  */
    if (!vma)
        goto bad_area;
    if (vma->vm_start <= address)
        goto good_area;
/*  如果这个地址在一个线性区期间，那么说明这是一个合法的地址，采用调入页等处理方式  */
    if (!(vma->vm_flags & VM_GROWSDOWN))
        goto bad_area;
/*  如果该地址没在这个线性区，检测该线性区是是否可以向下扩展的*/
    if (expand_stack(vma, address))
        goto bad_area;
/*  如果可以向下扩展，那么我们就可以扩展这块内存，来扩展栈空间。
    注意，在进程中只有栈所对应的线性区是向下扩展的。*/
}
```

从上面的内核代码我们可以知道，进程不需要系统调用来扩展栈段所在的内存空间，而是随着压栈的操作，栈顶指针的缩小而触发页故障，触发 Linux 内核扩展栈段所在的内存空间。由于不涉及系统调用，所以栈段内存的扩展要比堆段内存扩展更加方便、快捷。

2.3.2.3. 栈的释放

前面我们解决了栈段内存扩展的问题，下一个问题就是栈段内存如何回收。

利用函数压栈时，栈顶指针减少，从而触发页故障来扩展栈内存空间是有效的，可这个机制并不能完成栈段内存回收。

在进程从函数中返回时，释放临时变量以及返回到上一级函数，栈顶寄存器 esp 将自动增大，这时候，进程应该释放的这块内存，同时调整在 Linux 内核中该进程的栈段所对应的线性区。需要有一个事件，来触发上面的动作。

很遗憾的是，在 Linux 内核中，我找不到这样一个合适的事件，同时我也找不到一个与 expand_stack 想对的缩减栈的函数，这时我不得不怀疑在 Linux 中栈所使用的物理内存是否真的如我们过去所学到的那样，在从函数中返回时释放掉。

这里我们可以做一个试验，利用递归函数来扩展栈的空间。

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

int num=10000;                  //用来控制函数递归的层数。
int funca()
{
      num—;
      if(num==0)
      {
            return 0;
      }
      funca();
}
int main()
{
      funca();
      pid_t pid=getpid();
      printf("pid:%d\n",pid);
      pause();
      return 0;
}
```

在上面的代码中，首先我利用递归函数，扩展了进程的栈空间。在函数 main 中，随着递归函数的完成，栈段内存应该返还给了系统，这时候栈空间应该很少才对。然后我们利用 pause 使程序停住，以便我们来查看其内存使用情况。

我们来查看其 maps：
#cat maps
………………
befc5000-bf000000 rwxp befc5000 00:00 0   (236K)
惊讶，栈段的内存似乎真的没有释放，证实了我的猜想。

让我们再来看看物理内存：
#cat memmap
……………….
……………….
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1
该进程的栈使用了 236k 的物理内存。

完全证实了我的判断，在 Linux 系统中进程栈段所使用的物理内存是只能增长，不会减少的，其大小等于其在运行过程中所使用的最大的栈空间。

以前从来没有人来告诉我这个事实的真相。

不太清楚为什么会这么设计，我猜测可能有两方面的原因：
1、 确实没有合适的事件来触发栈段内存的回收。
2、 Linux 的栈段虽然在函数退出时不会被释放，但在下次进入函数时可以复用，因此可能认为 Linux 的栈段内存释放问题对于整个进程的内存使用影响不大。

而对我们来讲，慎用递归函数及在函数内分配大块内存，因为那是有代价的。

2.3.2.4. 栈内存的优化

因为栈的内存一经申请，就不会释放，所以
1、 尽量避免在栈空间申请大量内存；
2、 尽量避免使用递归函数。

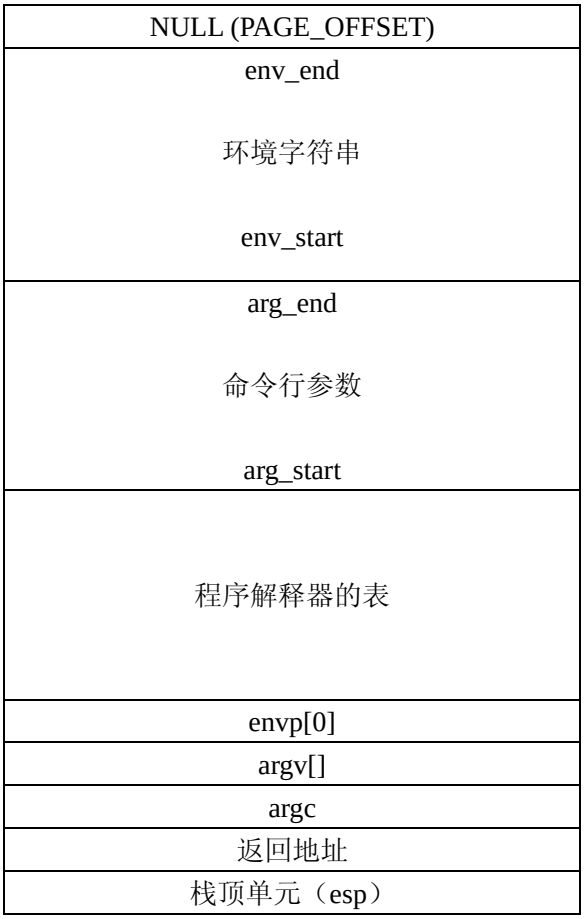
2.3.3. 环境变量及参数

在谈代码段和数据段之前，进程运行的环境变量和参数。

在程序运行时，其肯定与一些环境变量打交道，例如最常用的 LD_LIBRARY_PATH 等。这些环境变量存储在什么地方，又是如何传递的呢？

首先环境变量与进程有关，在系统 fork 出一个子进程后，会同时将自己的环境变量复制到子进程中，将其存储在栈的顶端，从而实现了环境变量在父子进程之间的继承关系。

在 Linux 内核中，环境变量的存储位置如下：



从 maps 里面的来看，它应该与栈段合用一个线性区。有时候你会发现，你刚进入 main 函数，根本没有用到什么栈空间，但其栈顶已经用了 2 个物理页面，那两个物理页面就应该是用来保存环境字符串和命令行参数的。

2.3.3.1. 环境变量的存储

下面我们来讲述，环境变量在进程的栈中是如何存储：

下面是一个打印环境变量的范例程序。

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

extern char **environ;
int main()
{
    char **env=environ;
    printf("environ:%p\n",environ);
```

```
    while(*env)
    {
        printf("env:%p    %p    %s\n",env,*env,*env);
        env++;
    }
}
```

在这里 envirom 为一个字符串指针数组，我们将数组的内容和地址，连同字符串信息一同打印出来。

```
./hello
environ:0xbefffeac
env:0xbefffeac    0xbeffff6b    USER=root
env:0xbefffeb0    0xbeffff75    OLDPWD=/root
env:0xbefffeb4    0xbeffff82    HOME=/root
env:0xbefffeb8    0xbeffff8d    PS1=#
env:0xbefffebc    0xbeffff94    LOGNAME=root
env:0xbefffec0    0xbefffa1    PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
env:0xbefffec4    0xbefffd3    SHELL=/bin/ash
env:0xbefffec8    0xbefffe2    PWD=/mnt/msc_int0
```

从结果我们可以看出，环境变量的字符串全部顺序从地址 0xbeffff6b 开始，保存在栈段的顶端。environ 保存着环境变量字符串指针数组的地址 0xbefffea，接下来的地址，均保存着对应的环境变量字符串地址。

从上面的结果我们可以看得出，这些环境变量的字符串排列的非常紧密，没有一点空隙。这样虽然使节省了内存，但也带来了问题。

如果我们新增一个环境变量，环境字符串将如何保存？字符串指针数组也有填满的可能。
如果我们删除一个环境变量，保存该环境变量所占用内存将如何释放？
如果我们修改一个环境变量，将其字符串延长，系统将如何处理呢？

下面，我们将试着一一回答这个问题。

2.3.3.2. 新增环境变量

在这里我写一个试验程序，先打印出当前环境变量存储情况，然后新增一个环境变量，最后再打印出增加后的环境变量存储情况，前后进行比较，我们就能知道结果了。

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

extern char **environ;

int main()
```

```
{
    char **env=environ;
    printf("environ:%p\n",environ);
    while(*env)
    {
        printf("env:%p    %p    %s\n",env,*env,*env);
        env++;
    }
    setenv("ezx","123",1);

    env=environ;
    printf("environ:%p\n",environ);
    while(*env)
    {
        printf("env:%p    %p    %s\n",env,*env,*env);
        env++;
    }

    return 0;
}
```

运行结果：

```
environ:0xbefffeac
env:0xbefffeac    0xbeffff6b    USER=root
env:0xbefffeb0    0xbeffff75    OLDPWD=/root
env:0xbefffeb4    0xbeffff82    HOME=/root
env:0xbefffeb8    0xbeffff8d    PS1=#
env:0xbefffebc    0xbeffff94    LOGNAME=root
env:0xbefffec0    0xbefffa1    PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
env:0xbefffec4    0xbefffd3    SHELL=/bin/ash
env:0xbefffec8    0xbefffe2    PWD=/mnt/msc_int0
environ:0x11050
env:0x11050    0xbeffff6b    USER=root
env:0x11054    0xbeffff75    OLDPWD=/root
env:0x11058    0xbeffff82    HOME=/root
env:0x1105c    0xbeffff8d    PS1=#
env:0x11060    0xbeffff94    LOGNAME=root
env:0x11064    0xbefffa1    PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
env:0x11068    0xbefffd3    SHELL=/bin/ash
env:0x1106c    0xbefffe2    PWD=/mnt/msc_int0
env:0x11070    0x11080    ezx=123
```

这个结果说明什么呢？

1、“ezx=123”，其地址为 0x11080。

说明该字符位于堆段，程序发现在栈的顶部无法保存新的环境变量，其便在堆段申请了一段内存用来保存环境变量。

2、 环境变量字符串指针数组的地址从 0xbefffeac 变为 0x11050。

因为环境变量字符串指针数组又多了一个指针，原来的栈的顶部就没有空间保存了，所以进程在堆段又申请了一段内存，用来保存所有的环境变量指针。

因此如果进程新增一个环境变量，系统将消耗的内存=4×系统环境变量总数+新增环境变量的长度+1。

2.3.3.3. 修改环境变量

我们用同样的方法来分析一下，修改环境变量，程序的行为。

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

extern char **environ;

int main()
{
    char **env=environ;
    printf("environ:%p\n",environ);
    while(*env)
    {
        printf("env:%p    %p    %s\n",env,*env,*env);
        env++;
    }
    setenv("PATH","123",1);

    env=environ;
    printf("environ:%p\n",environ);
    while(*env)
    {
        printf("env:%p    %p    %s\n",env,*env,*env);
        env++;
    }

    return 0;
}
```

运行：./hello

```
environ:0xbefffeac
env:0xbefffeac    0xbeffff6b    USER=root
```

```
env:0xbefffeb0    0xbeffff75    OLDPWD=/root
env:0xbefffeb4    0xbeffff82    HOME=/root
env:0xbefffeb8    0xbeffff8d    PS1=#
env:0xbefffebc    0xbeffff94    LOGNAME=root
env:0xbefffec0    0xbefffa1    PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
env:0xbefffec4    0xbefffd3    SHELL=/bin/ash
env:0xbefffec8    0xbeffffe2    PWD=/mnt/msc_int0
environ:0xbefffeac
env:0xbefffeac    0xbeffff6b    USER=root
env:0xbefffeb0    0xbeffff75    OLDPWD=/root
env:0xbefffeb4    0xbeffff82    HOME=/root
env:0xbefffeb8    0xbeffff8d    PS1=#
env:0xbefffebc    0xbeffff94    LOGNAME=root
env:0xbefffec0    0x11050    PATH=123
env:0xbefffec4    0xbefffd3    SHELL=/bin/ash
env:0xbefffec8    0xbeffffe2    PWD=/mnt/msc_int0
```

从结果上，我们分析可以知道：

1、 不论环境变量字符串增大或者减少，都会在堆段分配出一块内存来保存环境变量字符串。
2、 不会重新分配环境变量字符串指针数组的内存，会更新对应字符串指针的指向。

如果我再次修改 PATH 环境变量的值，系统会释放原来在堆中申请到的保存该环境变量的内存吗？

我们可以通过 glibc 的源码，来获得相关的信息，详细可以参见/glibc/sysdeps/generic/setenv.c 的__add_to_environ 函数。

当我们更新一个环境变量时，libc 并不会去判断对应的环境变量是否保存在堆中，然后去试图释放它；而是直接另外分配一块堆内存，来保存新的环境变量值，这时会有一点的内存泄漏，好在并不严重。

2.3.3.4. 释放环境变量

我们可以用函数 unsetenv 来释放环境变量，它会释放环境变量所占用的内存吗？

同样，我们在 glibc 来分析该函数，该函数位于 glibc/sysdeps/generic/setenv.c 中 00。

```
unsetenv (name)
      const char *name;
{
  size_t len;
  char **ep;
```

```
  len = strlen (name);
  LOCK;
  ep = __environ;
  while (*ep != NULL)
    if (!strncmp (*ep, name, len) && (*ep)[len] == '=')
      {
    /* 找到了对应的环境变量字符串数组下标，之后的事情是把环境变量字符串指针数组向前移动一位。/
    char **dp = ep;
    do
      dp[0] = dp[1];
    while (*dp++);
      }
    else
      ++ep;
  UNLOCK;
  return 0;
}
```

通过代码我们可以了解到，释放某个环境变量，只是简单的更新环境变量字符串指针数组，并没有释放相应的字符串资源，所以 unsetenv 并不会释放内存。

2.3.3.5. 环境变量内存优化

尽可能在程序启动前设置好环境变量，这样环境变量紧密的排列在栈空间。在程序内增加、修改环境变量将会导致在堆中申请内存。

对于程序员比较熟悉的堆段和栈段，已经基本阐述完了，下面我将要讲述进程的代码段和数据段。对于这两个内存段，相信很多人只是有一个轮廓，代码和全局变量到底是如何存储的都不是很清楚。

写到这里，我有些犹豫。我到底要不要详细的对这两个段进行论述，有可能只是从程序的角度说个其然，这样内容将更加浅显易懂；可如果不说其所以然的话，很多东西又不是那么容易理解。

反复思量，我决定在介绍代码段和数据段之前，我先介绍一下 ELF 文件，它是理解程序代码段和数据段的基础。

2.3.4. ELF 文件

我们先从一个最简单的应用程序“Hello，world!”入手。

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

我们先对齐进行编译：
gcc-o hello hello.c

有没有人注意过编译出来的 hello 有多大？
> ls -al hello
-rwxr-xr-x 1 jkvp74 devsrc 8953 Apr 23 10:29 hello
将近 9k，为什么如此简单的一个程序，会占据这么多空间呢？这个执行文件里都包含哪些内容呢？

下面我们将一一回答这些问题。

一个小技巧：
在 Linux 系统中，你可以通过 file 命令来得知文件的格式。
>file hello
ELF 32-bit LSB executable Version 1, dynamically linked, not stripped

2.3.4.1. 常用工具

当我们满怀对新知识的渴望，面对 ELF 文件时，第一盆凉水便是 ELF 文件采用二进制格式，不利于阅读。
有恒心的朋友，可以去尝试学习 ELF 文件的格式，你通过 google 可以很容易的找到；
懒惰的朋友，也别着急，这里有几个常用的工具，帮助我们来了解 ELF 文件。

下面的这些工具属于 binutils 工具包，在嵌入式开发前期创建交叉编译工具链时就已经生成。

命令	说明
strings	输出 ELF 文件中的所有字符串。
strip	删除 ELF 文件中一些无用的信息。
nm	列举目标文件符号.
size	显示目标文件段（section）大小，以及目标文件大小。
readelf	显示 elf 格式文件的内容。
objdump	显示目标文件信息，可作为反汇编用。
ar	建立 static library(Insert Delete List Extract)
addr2line	将地址转换成文件、行号。

演示程序

hello.c

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

int main()
{
        printf("Hello,world!\n");
        pid_t pid=getpid();
        printf("pid:%d\n",pid);
        pause();
        return 0;
}
```

我们将其编译成 ELF 文件。

```
gcc –o hello hello.c
```

```
#strings hello
_Jv_RegisterClasses
__gmon_start__
libc.so.6
pause
printf
getpid
abort
__libc_start_main
GLIBC_2.4
Hello,world!
pid:%d
```

```
#nm hello
000105b0      D    _DYNAMIC
00010698      D    _GLOBAL_OFFSET_TABLE_
0000857c      R    _IO_stdin_used
              w    _Jv_RegisterClasses
000085a0      r    __FRAME_END__
……
0000835c      T    _init
000083cc      T    _start
              U    abort@@GLIBC_2.4
00008404      t    call_gmon_start
000106c8      b    completed.0
```

```
000106c0    W   data_start
0000844c    t   frame_dummy
            U   getpid@@GLIBC_2.4
00008474    T   main
            U   pause@@GLIBC_2.4
            U   printf@@GLIBC_2.4
```

第一列，为符号的起始地址；
第二列，为符号的类型；

A	Global absolute
a	Local absolute
B	Global bss
b	Local bss
D	Global data
d	Local data
f	源文件名称符号。
T	Global text
t	Local text
U	未定义符号。

第三列，为符号的名称。

#size hello

```
   text    data     bss     dec     hex filename
    994     284       4    1282     502 hello
```

text 表示文件中指令的大小；
data 表示文件有初值的全局变量和静态变量的大小；
bss 表示文件中未赋初值或初值为 0 的全局变量和静态变量的大小；
dec=text+data+bss
hex 只不过是 dec 的 16 进制表示。

2.3.4.2. ELF 文件

下面我们主要使用 readelf 来查看 ELF 文件的内容。

我们可以直接运行 readelf，不带参数，来查看其帮助文件。

```
#readelf
Usage: readelf <option(s)> elf-file(s)
 Display information about the contents of ELF format files
 Options are:
  -a --all               Equivalent to: -h -l -S -s -r -d -V -A -I
  -h --file-header       Display the ELF file header
  -l --program-headers   Display the program headers
```

```
      --segments            An alias for --program-headers
  -S --section-headers      Display the sections' header
      --sections            An alias for --section-headers
  -g --section-groups       Display the section groups
  -e --headers              Equivalent to: -h -l -S
  -s --syms                 Display the symbol table
       --symbols            An alias for --syms
  -n --notes                Display the core notes (if present)
  -r --relocs               Display the relocations (if present)
  -u --unwind               Display the unwind info (if present)
  -d --dynamic              Display the dynamic section (if present)
  -V --version-info         Display the version sections (if present)
  -A --arch-specific        Display architecture specific information (if any).
  -D --use-dynamic          Use the dynamic section info when displaying symbols
  -x --hex-dump=<number> Dump the contents of section <number>
  -w[liaprmfFsoR]                                                    or
    --debug-dump[=line,=info,=abbrev,=pubnames,=aranges,=macro,=frames,=str,=loc,=Ranges]
                            Display the contents of DWARF2 debug sections
  -I --histogram            Display histogram of bucket list lengths
  -W --wide                 Allow output width to exceed 80 characters
  -H --help                 Display this information
  -v --version              Display the version number of readelf
```

对于上面的这些选项不清楚，没有关系，我们使用-a 选项，列出 elf 文件的所有内容，然后意义对其进行解释。

在运行 readelf 之前，我先整体上介绍一下 ELF 文件。
在 ELF 文件中，有两种视图：
一种主要是面向程序编译和链接，称之为链接视图。
它将 ELF 文件中所需要保存的信息按照信息的类型、格式的不同，分别保存在文件的不同区域。这些区域，中文我们叫它“节”，英文称之为“section”。为了访问这些节区，在 ELF 文件中又包含了一个这些节区位置的索引，节区头部表 section headers。
另一种主要是面向程序的加载和运行，称之为执行视图。
在执行视图中，它又会对前面提到的 section，按照运行时的需要，划分为不同的组，中文称之为“段”，英文称之为“segment”。为了说明段（segment）与节（section）的关系，在 ELF 文件中又引入了程序头部表 program headers。

而在文件开始处是一个 ELF 头部（ELF Header），用来描述整个文件的组织，并告诉我们节区头部表和程序头部表在 ELF 文件中的位置。

这里我用一个表来说明 ELF 文件的格式。

链接视图	执行视图
ELF 头部	ELF 头部

程序头部表（可选）	程序头部表
节区 1	段 1
…….	
节区 n	段 2
……	
…….	……
节区头部表	节区头部表（可选）

好了，很多人已经等的不耐烦了，下面让我们开始对 ELF 文件的探险之旅。

```
#readelf –a hello
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x836c
  Start of program headers:          52 (bytes into file)
  Start of section headers:          5308 (bytes into file)
  Flags:                             0x4000002, has entry point, Version4 EABI
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         8
  Size of section headers:           40 (bytes)
  Number of section headers:         39
  Section header string table index: 36
```

下面对于一些重要的选项，进行说明。

Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00

该行给出了ELF文件的一些标识信息，

7f 45 4c 46	表明该文件是ELF文件
01	表明该文件运行在32位的操作系统上
01	表明数据采用小端排序，高位在前

Entry point address: 0x836c

程序入口的虚拟地址。如果目标文件没有程序入口，可以为 0。在程序加载完成后，loader 会将程序的焦点转移到该地址。

Start of program headers: 52 (bytes into file)

表明程序头部表在 ELF 文件中的位置。

Start of section headers: 5308 (bytes into file)

表明节区头部表在 ELF 文件中的位置。

Size of this header: 52 (bytes)

表明 ELF 文件头部的大小。

注意这个值为 52，而程序头部表在 ELF 文件的位置也是 52，说明程序头部表在文件中紧挨着 ELF 文件头部。

Size of program headers: 32 (bytes)

表明程序头部表每行的大小为 32 个字节。

Number of program headers: 8

表明在程序头部表中共有 8 行，也就意味着程序中有 8 个段。

Size of section headers: 40 (bytes)

表明节区头部表每行的大小为 40 个字节。

Number of section headers: 39

表明在节区头部表中共有 39 行，也就意味着程序中有 39 个节。

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00	0	0	0	
[1]	.interp	PROGBITS	00008134	000134	000013	00	A	0	0	1
[2]	.note.ABI-tag	NOTE	00008148	000148	000020	00	A	0	0	4
[3]	.note.numapolicy	NOTE	00008168	000168	000074	00	A	0	0	4
[4]	.hash	HASH	000081dc	0001dc	00002c	04	A	5	0	4
[5]	.dynsym	DYNSYM	00008208	000208	000060	10	A	6	1	4
[6]	.dynstr	STRTAB	00008268	000268	000057	00	A	0	0	1
[7]	.gnu.version	VERSYM	000082c0	0002c0	00000c	02	A	5	0	2
[8]	.gnu.version_r	VERNEED	000082cc	0002cc	000020	00	A	6	1	4
[9]	.rel.dyn	REL	000082ec	0002ec	000008	08	A	5	0	4
[10]	.rel.plt	REL	000082f4	0002f4	000020	08	A	5	12	4
[11]	.init	PROGBITS	00008314	000314	000014	00	AX	0	0	4
[12]	.plt	PROGBITS	00008328	000328	000044	04	AX	0	0	4
[13]	.text	PROGBITS	0000836c	00036c	00017c	00	AX	0	0	4
[14]	.fini	PROGBITS	000084e8	0004e8	000010	00	AX	0	0	4
[15]	.rodata	PROGBITS	000084f8	0004f8	000014	00	A	0	0	4
[16]	.ARM.extab	PROGBITS	0000850c	00050c	000000	00	A	0	0	1
[17]	.ARM.exidx	ARM_EXIDX	0000850c	00050c	000008	00	AL	13	0	4
[18]	.eh_frame	PROGBITS	00008514	000514	000004	00	A	0	0	4
[19]	.init_array	INIT_ARRAY	00010518	000518	000004	00	WA	0	0	4
[20]	.fini_array	FINI_ARRAY	0001051c	00051c	000004	00	WA	0	0	4

[21] .jcr	PROGBITS	00010520	000520	000004	00	WA	0	0	4
[22] .dynamic	DYNAMIC	00010524	000524	0000e8	08	WA	6	0	4
[23] .got	PROGBITS	0001060c	00060c	000020	04	WA	0	0	4
[24] .data	PROGBITS	0001062c	00062c	000008	00	WA	0	0	4
[25] .bss	NOBITS	00010634	000634	000004	00	WA	0	0	1
[26] .comment	PROGBITS	00000000	000634	0001bc	00	0	0	1	
[27] .debug_aranges	PROGBITS	00000000	0007f0	0000b8	00	0	0	8	
[28] .debug_pubnames	PROGBITS	00000000	0008a8	00007a	00	0	0	1	
[29] .debug_info	PROGBITS	00000000	000922	00040c	00	0	0	1	
[30] .debug_abbrev	PROGBITS	00000000	000d2e	00015c	00	0	0	1	
[31] .debug_line	PROGBITS	00000000	000e8a	000255	00	0	0	1	
[32] .debug_frame	PROGBITS	00000000	0010e0	00007c	00	0	0	4	
[33] .debug_str	PROGBITS	00000000	00115c	0001ae	00	0	0	1	
[34] .note.gnu.arm.ide	NOTE	00000000	00130a	00001c	00	0	0	1	
[35] .debug_ranges	PROGBITS	00000000	001326	000018	00	0	0	1	
[36] .shstrtab	STRTAB	00000000	00133e	00017e	00	0	0	1	
[37] .symtab	SYMTAB	00000000	001ad4	000750	10	38	87	4	
[38] .strtab	STRTAB	00000000	002224	0002f5	00	0	0	1	

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)
There are no section groups in this file.

在讲述每个节之前，我们先来说下每个列的含义。

Type：为这个节的内容进行分类。

NULL　　此值标志节区头部是非活动的，没有对应的节区。此节区头部中的其他成员取值无意义。

PROGBITS　此节区包含程序定义的信息，其格式和含义都由程序来解释。

SYMTAB　　此节区包含一个符号表。目前目标文件对每种类型的节区都只能包含一个，不过这个限制将来可能发生变化。

STRTAB　　此节区包含字符串表。目标文件可能包含多个字符串表节区。

RELA　　此节区包含重定位表项，其中可能会有补齐内容（addend），例如 32 位目标文件中的 Elf32_Rela 类型。目标文件可能拥有多个重定位节区。

HASH　　此节区包含符号哈希表。所有参与动态链接的目标都必须包含一个符号哈希表

DYNAMIC　此节区包含动态链接的信息。

NOTE　　此节区包含以某种方式来标记文件的信息。

NOBITS　　这种类型的节区不占用文件中的空间，其他方面和 SHT_PROGBITS 相似。

DYNSYM　　作为一个完整的符号表，它可能包含很多对动态链接而言不必要的符号。

INIT_ARRAY 在 main 函数之前，运行的函数指针数组。

FINI_ARRAY 在退出 main 函数之后，运行的函数指针数组。

Addr： 如果节区将出现在进程的内存映像中，此成员给出节区的第一个字节应处的位置。否则，此字段为 0

Off： 表示该节内容在距离文件起始的偏移地址。

Size： 表示该节在文件中的大小。

ES： 某些节区中包含固定大小的项目，如符号表。对于这类节区，此成员给出每个表项的长度字节数。 如果节区中并不包含固定长度表项的表格，此成员取值为 0。

Flag： 表示该节的内存分配属性。

A 表示分配内存

X 表示可执行

W 表示可写

Lk： 此成员给出节区头部表索引链接。

AL： 某些节区带有地址对齐约束。

在对每个列的含义说明之后，下面我们介绍几个经常用到的节。

[4] .hash HASH 000081dc 0001dc 00002c 04 A 5 0 4

此节区包含了一个符号哈希表。

[5] .dynsym DYNSYM 00008208 000208 000060 10 A 6 1 4

此节区包含了动态链接符号表。

[6] .dynstr STRTAB 00008268 000268 000057 00 A 0 0 1

此节区包含用于动态链接的字符串，大多数情况下这些字符串代表了与符号表项相关的名称。

[9] .rel.dyn REL 000082ec 0002ec 000008 08 A 5 0 4

[10] .rel.plt REL 000082f4 0002f4 000020 08 A 5 12 4

这些节区中包含了重定位信息。如果文件中包含可加载的段，段中有重定位内容，节区的属性将包含 SHF_ALLOC 位，否则该位置 0。

[11] .init PROGBITS 00008314 000314 000014 00 AX 0 0 4

此节区包含了可执行指令，是进程初始化代码的一部分。当程序开始执行时，系统要在开始调用主程序入口之前（通常指 C 语言的 main 函数）执行这些代码。

[12] .plt PROGBITS 00008328 000328 000044 04 AX 0 0 4

此节区包含过程链接表（procedure linkage table）。

[13] .text PROGBITS 0000836c 00036c 00017c 00 AX 0 0 4

此节区包含程序的可执行指令。

[14] .fini PROGBITS 000084e8 0004e8 000010 00 AX 0 0 4

此节区包含了可执行的指令，是进程终止代码的一部分。程序正常退出时，系统将安排执行这里的代码。

[15] .rodata PROGBITS 000084f8 0004f8 000014 00 A 0 0 4

这些节区包含只读数据，这些数据通常参与进程映像的只读代码段。

[19] .init_array INIT_ARRAY 00010518 000518 000004 00 WA 0 0 4

进程初始化的所运行的函数指针数组

[20] .fini_array FINI_ARRAY 0001051c 00051c 000004 00 WA 0 0 4

进程退出时的所运行的函数指针数组

[22] .dynamic DYNAMIC 00010524 000524 0000e8 08 WA 6 0 4

此节区包含动态链接信息。

[23] .got PROGBITS 0001060c 00060c 000020 04 WA 0 0 4

此节区包含全局偏移表，其与 plt 一起协作完成符号的动态查找。

[24] .data PROGBITS 0001062c 00062c 000008 00 WA 0 0 4

这些节区包含初始化了的数据，将出现在程序的内存映像中。

[25] .bss NOBITS 00010634 000634 000004 00 WA 0 0 1

包含将出现在程序的内存映像中的为初始化数据。根据定义，当程序开始执行，系统将把这些数据初始化为 0。此节区不占用文件空间。

[26] .comment PROGBITS 00000000 000634 0001bc 00 0 0 1

[26] .comment PROGBITS 00000000 000634 0001bc 00 0 0 1

[27] .debug_aranges PROGBITS 00000000 0007f0 0000b8 00 0 0 8

[28] .debug_pubnames PROGBITS 00000000 0008a8 00007a 00 0 0 1

[29] .debug_info PROGBITS 00000000 000922 00040c 00 0 0 1

[30] .debug_abbrev PROGBITS 00000000 000d2e 00015c 00 0 0 1

[31] .debug_line PROGBITS 00000000 000e8a 000255 00 0 0 1

[32] .debug_frame PROGBITS 00000000 0010e0 00007c 00 0 0 4

[33] .debug_str PROGBITS 00000000 00115c 0001ae 00 0 0 1

此节区包含用于符号调试的信息。

[34] .note.gnu.arm.ide NOTE 00000000 00130a 00001c 00 0 0 1

[35] .debug_ranges PROGBITS 00000000 001326 000018 00 0 0 1

[36] .shstrtab STRTAB 00000000 00133e 00017e 00 0 0 1

此节区包含节区名称。

[37] .symtab SYMTAB 00000000 001ad4 000750 10 38 87 4

此节区包含一个符号表。如果文件中包含一个可加载的段，并且该段中包含符号表，那么节区的属性中包含 SHF_ALLOC 位，否则该位置为 0。

[38] .strtab STRTAB 00000000 002224 0002f5 00 0 0 1

此节区包含一个符号表。如果文件中包含一个可加载的段，并且该段中包含符号表，那么节区的属性中包含 SHF_ALLOC 位，否则该位置为 0。

这里我们需要注意到一点，Flags 属性为 A 和 AX 的节，在 ELF 文件的分布是连续的，中间没有穿插不 AW 或不需要内存的节，同样所有属性为 AW 的节都顺序的排列在一起。这为进程运行时划分为只读代码段和可读写的数据段奠定了基础。

There are no section groups in this file.

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
EXIDX	0x000598	0x00008598	0x00008598	0x00008	0x00008	R	0x4
PHDR	0x000034	0x00008034	0x00008034	0x00100	0x00100	RE	0x4
INTERP	0x000134	0x00008134	0x00008134	0x00013	0x00013	R	0x1
[Requesting program interpreter: /lib/ld-linux.so.3]							
LOAD	0x000000	0x00008000	0x00008000	0x005a4	0x005a4	RE	0x8000
LOAD	0x0005a4	0x000105a4	0x000105a4	0x00124	0x00128	RW	0x8000
DYNAMIC	0x0005b0	0x000105b0	0x000105b0	0x000e8	0x000e8	RW	0x4
NOTE	0x000148	0x00008148	0x00008148	0x00020	0x00020	R	0x4
NOTE	0x000168	0x00008168	0x00008168	0x00074	0x00074	R	0x4

我们还是先介绍一下程序头部表各列的含义。

Type：各段的类型。

PHDR： 此类型的数组元素如果存在，则给出了程序头部表自身的大小和位置，既包括在文件中也包括在内存中的信息。

INTERP： 数组元素给出一个 NULL 结尾的字符串的位置和长度，该字符串将被当作解释器调用。对于 ELF 文件来讲，该段给出了启动进程的 Loader。

LOAD： 此数组元素给出一个可加载的段，段的大小由 p_filesz 和 p_memsz 描述。文件中的字节被映射到内存段开始处。

. DYNAMIC： 数组元素给出动态链接信息。

NOTE： 此数组元素给出附加信息的位置和大小。

Offset ：该段在距离文件开头的偏移地址。

VirtAddr ：此成员给出段的第一个字节将被放到内存中的虚拟地址。

PhysAddr：此成员仅用于与物理地址相关的系统中。

FileSiz ：此成员给出段在文件映像中所占的字节数。

MemSiz ：此成员给出段在内存映像中占用的字节数。

注意，MemSize 可能与 FileSize 不等，主要是因为.bss 节只占据内存空间，不占据文件空间。

Flg ：该段的属性。

R： 表示可读

E： 表示可执行

W： 表示可写

Align ：表示该段的要求字节对齐的属性。

下面我们着重介绍程序头部表的某些行：

PHDR 0x000034 0x00008034 0x00008034 0x00100 0x00100 RE 0x4

偏移值-0x34，换算为 10 进制为 52，正好是程序头部表在 ELF 文件的位置。

INTERP 0x000134 0x00008134 0x00008134 0x00013 0x00013 R 0x1

[Requesting program interpreter: /lib/ld-linux.so.3]

该段给出了运行该文件所需要的解释器，针对 ELF 文件来讲，其特指加载 ELF 文件到内存，做符号解析，直到把运行焦点返还给进程的 loader。

LOAD 0x000000 0x00008000 0x00008000 0x005a4 0x005a4 RE 0x8000

该段的内容将会被加载到内存中，并且其权限为只读、可执行，所以该段对应于我们常说的代码段。

LOAD 0x0005a4 0x000105a4 0x000105a4 0x00124 0x00128 RW 0x8000

该段的内容被加载到内存中，并且其权限为可读、可写，所以该段对应于我们常说的数据段。

DYNAMIC 0x0005b0 0x000105b0 0x000105b0 0x000e8 0x000e8 RW 0x4

该段给出了此 ELF 所需要的动态链接信息。

现在我们知道了程序运行中代码段和数据段的来历，但各段的具体内容还是不清楚。

别急，下面的结果就给出了段与节的映射关系。

Section to Segment mapping:

```
 Segment Sections...
  00     .ARM.exidx
  01
  02     .interp
  03     .interp .note.ABI-tag .note.numapolicy .hash .dynsym .dynstr .gnu.version .gnu.versi
on_r .rel.dyn .rel.plt .init .plt .text .fini .rodata .ARM.exidx .eh_frame
  04     .init_array .fini_array .jcr .dynamic .got .data .bss
  05     .dynamic
  06     .note.ABI-tag
  07     .note.numapolicy
```

在这里面的每一行，按照顺序与前面在程序头部表的每一行一一对应。

因此，我们可以看到

代码段：

```
LOAD           0x000000       0x00008000     0x00008000     0x005a4  0x005a4  RE 0x8000
  03     .interp .note.ABI-tag .note.numapolicy .hash .dynsym .dynstr .gnu.version .gnu.versi
on_r .rel.dyn .rel.plt .init .plt .text .fini .rodata .ARM.exidx .eh_frame
```

数据段：

```
LOAD           0x0005a4       0x000105a4     0x000105a4     0x00124  0x00128  RW 0x8000
  04     .init_array .fini_array .jcr .dynamic .got .data .bss
```

同时我们也可以看到，在代码段中的各节在文件中是顺序排列的；在数据段中的各节也是在文件中数序排列的。
这是因为，在程序运行前期，loader 会将 ELF 文件的代码段和数据段使用 mmap 将其映射到内存中，这就要求各段所包含的节在文件中必须是连续的。

```
Dynamic section at offset 0x5b0 contains 24 entries:
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [libc.so.6]
 0x0000000c (INIT)                       0x835c
 0x0000000d (FINI)                       0x856c
 0x00000019 (INIT_ARRAY)                 0x105a4
 0x0000001b (INIT_ARRAYSZ)               4 (bytes)
 0x0000001a (FINI_ARRAY)                 0x105a8
 0x0000001c (FINI_ARRAYSZ)               4 (bytes)
 0x00000004 (HASH)                       0x81dc
 0x00000005 (STRTAB)                     0x8290
 0x00000006 (SYMTAB)                     0x8210
 0x0000000a (STRSZ)                      100 (bytes)
 0x0000000b (SYMENT)                     16 (bytes)
 0x00000015 (DEBUG)                      0x0
```

0x00000003	(PLTGOT)	0x10698
0x00000002	(PLTRELSZ)	48 (bytes)
0x00000014	(PLTREL)	REL
0x00000017	(JMPREL)	0x832c
0x00000011	(REL)	0x8324
0x00000012	(RELSZ)	8 (bytes)
0x00000013	(RELENT)	8 (bytes)
0x6ffffffe (	VERNEED)	0x8304
0x6fffffff	(VERNEEDNUM)	1
0x6ffffff0	(VERSYM)	0x82f4
0x00000000	(NULL)	0x0

这个节主要包含了动态链接信息，其余的我们不管，只关注下面一行

0x00000001	(NEEDED)	Shared library: [libc.so.6]

我们知道在程序中，可以会调用一些动态库的接口，这样在程序运行时，loader 就需要把这些动态库加载到系统中。那么 loader 怎么知道该程序依赖于哪几个动态库呢？

没错，就是通过 dynamic section 节，Type 为 NEEDED 就代表程序依赖于该动态库。

reaelf –a hello 的输出后面还有很多，在这里我就不一一列出了，有兴趣的朋友可以去阅读相关的 ELF 文档。

2.3.4.3. 程序瘦身

有些细心的朋友可能会发现，在节头部表中有 39 个节，而这 39 个节并没有全部在“节--段映射表”中出现。

比如说下面的这个节：

[27] .debug_aranges	PROGBITS	00000000	0007f0	0000b8	00	0	0	8
[28] .debug_pubnames	PROGBITS	00000000	0008a8	00007a	00	0	0	1
[29] .debug_info	PROGBITS	00000000	000922	00040c	00	0	0	1
[30] .debug_abbrev	PROGBITS	00000000	000d2e	00015c	00	0	0	1
[31] .debug_line	PROGBITS	00000000	000e8a	000255	00	0	0	1
[32] .debug_frame	PROGBITS	00000000	0010e0	00007c	00	0	0	4
[33] .debug_str	PROGBITS	00000000	00115c	0001ae	00	0	0	1
[34] .note.gnu.arm.ide	NOTE	00000000	00130a	00001c	00	0	0	1
[35] .debug_ranges	PROGBITS	00000000	001326	000018	00	0	0	1
[36] .shstrtab	STRTAB	00000000	00133e	00017e	00	0	0	1
[37] .symtab	SYMTAB	00000000	001ad4	000750	10	38	87	4
[38] .strtab	STRTAB	00000000	002224	0002f5	00	0	0	1

实际上这些节只是保存着一些调试信息，并不参与程序的运行，所以其不在“节—段映射表”中。

既然与运行无关，那么我们是否可以删除它们呢？
可以，由于这些节所包含的调试信息往往占据 ELF 文件一半的大小，鉴于嵌入式设备的 Flash 有限，往往我们会将 ELF 文件中这些与运行无关的节删除掉，减少 ELF 文件的大小后，才放到嵌入式设备当中去。

删除这些与运行无关的节，我们可以使用 strip 命令。

我们还以上面编译出的 hello 为例：

1、查看一下 hello 文件的大小。
> ls -al hello
-rwxr-xr-x 1 jkvp74 devsrc 9172 Apr 23 14:09 hello

2、调用 strip，删除与运行无关的节
>strip hello
注意，该命令会直接修改源文件。

3、 我们再来查看一下 hello 文件的大小。
> ls -al hello
-rwxr-xr-x 1 jkvp74 devsrc 3628 Apr 23 15:45 hello

可以看到从原来的 9172 个字节，缩减到 3628，效果很明显，不是吗。

4、 我们使用 readelf 来查看修改后的 ELF 文件的节的情况。

>readelf –S hello
There are 29 section headers, starting at offset 0x9a4:

Section Headers:

[Nr] Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]	NULL	00000000	000000	000000	00		0	0	0
[1] .interp	PROGBITS	00008134	000134	000013	00	A	0	0	1
[2] .note.ABI-tag	NOTE	00008148	000148	00002000		A	0	0	4
[3] .note.numapolicy	NOTE	00008168	000168	000074	00	A	0	0	4
[4] .hash	HASH	000081dc	0001dc	000034	04	A	5	0	4
[5] .dynsym	DYNSYM	00008210	000210	000080	10	A	6	1	4
[6] .dynstr	STRTAB	00008290	000290	000064	00	A	0	0	1
[7] .gnu.version	VERSYM	000082f4	0002f4	000010	02	A	5	0	2
[8] .gnu.version_r	VERNEED	00008304	000304	000020	00	A	6	1	4
[9] .rel.dyn	REL	00008324	000324	000008	08	A	5	0	4
[10] .rel.plt	REL	0000832c	00032c	000030	08	A	5	12	4
[11] .init	PROGBITS	0000835c	00035c	000014	00	AX	0	0	4
[12] .plt	PROGBITS	00008370	000370	00005c	04	AX	0	0	4
[13] .text	PROGBITS	000083cc	0003cc	0001a0	00	AX	0	0	4

```
[14] .fini             PROGBITS        0000856c 00056c 000010 00  AX  0   0  4
[15] .rodata           PROGBITS        0000857c 00057c 00001c 00   A  0   0  4
[16] .ARM.extab        PROGBITS        00008598 000598 000000 00   A  0   0  1
[17] .ARM.exidx        ARM_EXIDX       00008598 000598 000008 00  AL  0   0  4
[18] .eh_frame         PROGBITS        000085a0 0005a0 000004 00   A  0   0  4
[19] .init_array       INIT_ARRAY      000105a4 0005a4 000004 00  WA  0   0  4
[20] .fini_array       FINI_ARRAY      000105a8 0005a8 000004 00  WA  0   0  4
[21] .jcr              PROGBITS        000105ac 0005ac 000004 00  WA  0   0  4
[22] .dynamic          DYNAMIC         000105b0 0005b0 0000e8 08  WA  6   0  4
[23] .got              PROGBITS        00010698 000698 000028 04  WA  0   0  4
[24] .data             PROGBITS        000106c0 0006c0 000008 00  WA  0   0  4
[25] .bss              NOBITS          000106c8 0006c8 000004 00  WA  0   0  1
[26] .comment          PROGBITS        00000000 0006c8 0001bc 00      0   0  1
[27] .note.gnu.arm.ide NOTE            00000000 000884 00001c 00      0   0  1
[28] .shstrtab         STRTAB          00000000 0008a0 000103 00      0   0  1
```

我们成功的将 section 的数量从 39 个减少到 29 个。

5、 我们还可以把.comment 节从 ELF 文件中删除。

```
>strip --remove-section=.comment hello
```

6、 我们再来看看 hello 文件的大小。

```
> ls -al hello
-rwxr-xr-x     1 jkvp74     devsrc          3136 Apr 23 16:06 hello
```

文件从 3638 个字节，减少到 3136 个字节。

我们也可以直接运行

```
>strip –remove-section=.comment –strip-all hello
```

来一次性的完成上面的步骤。

我们可以只运行 strip 来输出帮助，了解其所有选项。

```
>strip
strip <option(s)> in-file(s)
 Removes symbols and sections from files
 The options are:
  -I --input-target=<bfdname>      Assume input file is in format <bfdname>
  -O --output-target=<bfdname>     Create an output file in format <bfdname>
  -F --target=<bfdname>            Set both input and output format to <bfdname>
  -p --preserve-dates              Copy modified/access timestamps to the output
  -R --remove-section=<name>       Remove section <name> from the output
  -s --strip-all                   Remove all symbol and relocation information
  -g -S -d --strip-debug           Remove all debugging symbols & sections
      --strip-unneeded             Remove all symbols not needed by relocations
      --only-keep-debug            Strip everything but the debug information
```

```
  -N --strip-symbol=<name>          Do not copy symbol <name>
  -K --keep-symbol=<name>           Only copy symbol <name>
  -w --wildcard                     Permit wildcard in symbol comparison
  -x --discard-all                  Remove all non-global symbols
  -X --discard-locals               Remove any compiler-generated symbols
  -v --verbose                      List all object files modified
  -V --version                      Display this program's version number
  -h --help                         Display this output
       --info                       List object formats & architectures supported
  -o <file>                         Place stripped output into <file>
```

注意：strip 虽然可以减少 ELF 文件的大小，但它并不会修改代码段和数据段中的节，所以其不会减少进程运行时的内存使用。

在粗略的介绍完 ELF 文件之后，我们可以开始步入正体进程的数据段和代码段。

2.3.4.4. 程序的运行

不论我们是在 telnet 终端
检查存放文件前 128 字节中的一些魔数以确认可执行格式。如果魔数不匹配，则返回错误码 -ENOEXEC。
读可执行首部。这个首部描述程序的段和所需要的共享库。
从可执行文件获得程序解释器的路径名，用程序解释器来确定共享库的位置并把他们映射到内存。
获得程序解释器的目录项对象。
检查程序解释器的执行许可权。
把程序解释器的前 128 字节拷贝到缓冲区。
对程序解释器的类型执行一些一致性检查。
调用 flush_old_exec 函数释放前一个计算所占用的几乎所有资源。
建立进程新的个性，即建立进程描述符的 persionality 字段。
清进程描述符的 PF_FORKNOEXEC 标志。
为进程的用户态堆栈分配一个新的线性区描述符。
使用 do_mmap 函数创建一个线性区来对可执行文件正文段（即代码）进行映射。
使用 do_mmap 函数创建一个线性区来对可执行文件数据段进行映射。
为可执行文件的其他专用段分配另外的线性区。
调用一个装入程序解释器的函数。
吧可执行格式的 linux_binfmt 对象的地址存放在进程描述符的 binfmt 字段中。
确定进程新的权能。
创建特定的程序解释器表并把他们存放在用户堆栈。
设置进程的内存描述符的 start_code end_code end_data start_brk brk 以及 start_stack 字段。
调用 do_brk 函数创建一个新的匿名线性区来映射程序的 bss 段。这个线性区的大小是在可执行程序被链接时计算出来的。

转到程序的入口。

2.3.5. 数据段

我们可以通过解析 ELF 文件，来得知进程的数据段都包括哪些内容。

```
    04        .init_array .fini_array .jcr .dynamic .got .data .bss
```

.init_array 和.fini_array 是进程在启动和退出时，所要运行的函数指针数组，这节我们会在讲述非内置类型全局变量时讲到。

.dynamic 主要涉及进程的动态链接信息，对于进程的数据段影响不大，暂且忽略。
.got 节主要与.plt 节合作完成符号的动态查找和链接，对于我们的优化也不大。

我们将重点放在.data 和.bss 节上面。

2.3.5.1. .bss 与 .data 的区别

.bss 和.data 应该说是我们在做数据段优化的一个大头，在此我们着重声明。

我们先来看看从程序的角度着两个 section 的区别。
.bss：　　主要用来保存未初始化或初始化不为 0 的全局变量和静态变量。
.data：　　主要用来保存初始化不为 0 的全局变量或静态变量。

为什么初值是否为 0，变得如此关键呢？
这主要是因为 loader 可以对初值为 0 的变量采取一定的优化措施。
上文我们说到，loader 在加载进程时，会使用 mmap，将 ELF 文件的数据段映射到内存中。
1、 对于那些初值不为 0 的位于数据段的变量，其初始值必须保存在.data 节，需要占用文件大小，当访问这些变量时，便会触发页故障，将文件中对应的初值加载到内存中，完成初始化。
2、 对于那些初值为 0 的位于数据段的变量，不必将初值保存到文件中，loader 只需要将这段内存映射到一个全 0 的页面即可，这样.bss 节并不占据 ELF 文件的空间。

下面我举个例子，做一下对比。
bss.c

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

int bss_array[1024 * 1024] = {0};
int main(int argc, char* argv[])
{
    pause();
```

```
    return 0;
}
```

上面的例子，我声明了一个初值为 0 的大小为 4M 的数组，这个数组应该位于.bss 节。

```
>gcc –o bss bss.c
>ls –al bss
-rwxr-xr-x    1 jkvp74    devsrc          9501 Jan 15 17:43 bss
```

生成文件的大小为 9501 个字节。

data.c

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

int data_array[1024 * 1024] = {1};
int main(int argc, char* argv[])
{
    pause();
    return 0;
}
```

上面的例子，我声明了一个初值为 1 的大小为 4M 的数组，这个数组应该位于.data 节。

```
>gcc –o data data.c
>ls –al data
-rwxr-xr-x    1 jkvp74    devsrc      4203805 Jan 15 17:44 hello
```

生成文件的大小为 4,203,805 个字节。

这样，我们基本证明了.bss 节的变量不会占据 ELF 文件空间。

让我们再来看看运行过程中，loader 加载数据段时会有什么不同！

```
#./bss
```

我们来查看其 maps

```
00008000-00009000 r-xp 00000000 1f:12 303              /mnt/msc_int0/bss
00010000-00011000 rw-p 00000000 1f:12 303              /mnt/msc_int0/bss
00011000-00411000 rwxp 00011000 00:00 0
40000000-40001000 rw-p 40000000 00:00 0
```

```
#./data
```

我们来查看其 maps

```
00008000-00009000 r-xp 00000000 1f:12 304              /mnt/msc_int0/data
00010000-00411000 rw-p 00000000 1f:12 304              /mnt/msc_int0/data
40000000-40001000 rw-p 40000000 00:00 0
```

我们前后比较两个 maps 的不同：

bss 进程
其数据段很小，只有一个页面
00010000-00011000 rw-p 00000000 1f:12 303 /mnt/msc_int0/bss
但在代码中，我没有调用 malloc 分配堆内存，为什么这里却有一个 4M 的堆段呢？
00011000-00411000 rwxp 00011000 00:00 0

data 进程
其就完全符合我们原来的理论，其拥有一个 4M 的数据段，由于没有申请堆内存，所以没有堆段。

为什么会这样呢？
我们使用 readelf 来查看 bss 的程序头部表。
>readelf –l bss

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg Align
LOAD	0x000000	0x00008000	0x00008000	0x00510	0x00510	R E 0x8000
LOAD	0x000510	0x00010510	0x00010510	0x0011c	0x400120	RW 0x8000

我们可以使用 readelf 来查看 data 的程序头部表。
>readelf –l data

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg Align
LOAD	0x000000	0x00008000	0x00008000	0x00510	0x00510	R E 0x8000
LOAD	0x000510	0x00010510	0x00010510	0x40011c	0x400120	RW 0x8000

我们可以看到它们数据段的 FileSiz 是不同的，MemSize 相同。
Loader 在处理这种情况时，其首先根据 FileSiz 的大小，来创建数据段，这样 bss 进程的数据段只有 4k，而 data 进程的数据段就有 4M 多。
为了容纳 bss 中 4M 的 bss 数据，loader 将在进程的堆段申请出足够的内存来容纳它，这些页面的属性定义为全 0，所以我们发现在 bss 进程，虽然我们没有申请内存，但却有了一个 4M 的堆段。

了解这点，对于我们分析堆段内存时非常有用，并不是只有 malloc 和 new 等分配堆段内存，有时候我们还要考虑 bss 数据对器的影响。

这里还有一个差别：
当程序读取 data 节的数据时，这时系统会触发页故障，从而分配相应的物理内存。
当程序读取 bss 节的数据时，这时内核会将其转到一个全零的页面，不会触发页故障，也不会为其分配物理内存。

这里会有一个问题，我们看下下面的例子：

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

```
#include <stdio.h>

int data[2661]={1};
int bss;
int main()
{
	printf("data=%p,bss=%p\n",data,&bss);
	pause();
	return 0;
}
```

其运行结果是：

```
# ./hello
data=0x1067c,bss=0x12680
```

其对应的 maps：

```
# cat maps
00008000-00009000 r-xp 00000000 1f:12 279                /mnt/msc_int0/hello
00010000-00013000 rw-p 00000000 1f:12 279                /mnt/msc_int0/hello
```

我们看到实际上位于.data 节的 data 数组和位于.bss 节的 bss 整型，都处在数据段，这是为什么呢？

在 Linux 内核中内存管理是以页面为单位的，进程的数据段也必须是页面对齐的，可问题是数据段做映射时，往往.data 节无法正好填满最后一个页面，会剩余一些字节，这时 loader 会试图用.bss 节的数据去填充它。
前面我们提到对于.bss 的数据，内存会映射到一个全零的页面，而对于这个既有.data 节数据，又有.bss 节数据的页面，它是无法这么做的。
这时，loader 在将数据段映射完成后，在最后的一个页面，.data 节填充后剩余的字节，使用.bss 节数据进行填充，并将这些剩余的字节全部填充为 0，这同时会造成对最后一个页面的写操作，产生 dirty page。

我们看下上面例子中的 memmap：

```
# cat memmap
 2
 1 0 1
```

从程序中可以看到，我们没有对.data 节的 data 和.bss 节的 bss 进行读写操作，但还是在最后一个页面产生了一个物理页面。这就说明，loader 在加载进程数据段时，对数据段最后一个页面做补齐操作时，对内存进行写操作。

2.3.5.2. 变量所在内存区域

上面我们讲述了.bss 和.data 节的数据在进程运行时，表现行为的不同；下面我们讲述在程序代码中，.bss 和.data 定义的区别。

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

#define b 10
#define c "123"

int bss[10]={0};
int data[10]={1};
const int a=10;
static int m1;

int main()
{
        static int m2;
        pid_t pid=getpid();
        return 0;
}
```

提问：请分别说出上面变量所在物理内存的位置？

bss 是初始化为 0 的全局变量，所以其在.bss 节；
data 是初始化不为 0 的全局变量，所以其在.data 节；
注意 a，虽然它是初始化不为 0 的全局变量，但它前面有个 const，说明其数据是只读的，对于这种变量，将会存储在.rodata 节，位于代码段。
m1 是没有初始化的全局静态变量，其位于.bss 节。
m2 不是全局变量，但前面有 static，是一个未初始化的静态局部变量，所以其位于.bss 节。
pid 是一个局部变量，在进程运行期间，其将存储在栈中。

至于 b 和 c，其属于宏定义，在预处理的时候，这两个标志符就已经被替换掉了。
至于其多代表的值，10 将被认为是立即数，编译到代码中；"123"字符串常量，将被编译到.rodata section 中。

另外在这里我告诉大家一种验证的方法。

```
#nm -f sysv hello'
Symbols from hello:

Name                  Value   Class        Type         Size     Line  Section
```

a	|00008500|	R	|	OBJECT|	00000004|	|.rodata
bss	|00010658|	B	|	OBJECT|	00000028|	|.bss
data	|0001062c|	D	|	OBJECT|	00000028|	|.data
m1	|00010684|	b	|	OBJECT|	00000004|	|.bss
m2.0	|00010680|	b	|	OBJECT|	00000004|	|.bss
main	|00008414|	T	|	FUNC|	00000024|	|.text

很简单，不是吗：）。

下面我再出一道难的。

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

char *s1="Hello,world\n";
char s2[]="Hello,world\n";

int main()
{
……..
    return 0;
}
```

在上面的例子中，我们定义了两个字符串，一个使用字符串指针定义，一个使用了字符串数组，我的问题前后两个“Hello,world\n”存储在什么地方？

```
char *s1="Hello,world\n";
```

在这里只是定义了一个字符串指针，该指针指向一个常量。
所以“Hello,world\n”做为一个字符串常量存在，存储在.rodata 节，s1 做为一个指针，存储在.data 节，而指针指向了位于.rodata 的字符串。
如果你尝试对 s1 所指向的字符串做修改时，进程会报错。

```
char s2[]="Hello,world\n";
```

在这里定义了一个字符串数组，s2 不光是一个字符串指针，而且编译器还要为 s2 分配相应的空间来容纳字符串。
所以“Hello,world\n”将存储在.data 区，s2 做为一个指针也将存储在.data 节，指针指向位于.data 节的字符串。
你可以尝试着修改 s2 所指向的字符串，只要不超过数组的长度即可。

下面我们再来在 C++类中内存分布情况。

```
class c1
{
public:
    static int nCount;
```

```
    int nValue;
    char c;
    c1();
    virtual ~c1();

    int getValue(void);
    virtual void foo(void);
    static void addCount();
}
```

我们可以通过 sizeof()得到 c1 对象的大小为 12 个字节。

1、 函数 c1，～c1()，getValue,foo,addCount 为函数，其位于程序的代码段，多个对象共享，因此不算在 c1 的 size 中。
2、 static int nCount，因为该变量为静态变量，在 c1 所定义的对象之间共享，其位于程序的数据段。其不会随着对象数据的增加而增加。
3、 nValue 和 c 占据内存，其中 nValue 使用了 4 个字节，c 虽然使用了 1 个字节，但由于内存对齐的缘故，其也使用了 4 个字节，这样总共占据了 8 个字节。
4、 因为有虚函数，每个类对象要有一个指向虚函数表的指针，每个对象一个，占据 4 个字节，虚函数表是位于程序的代码段。

这样 c1 对象的大小为 12 个字节。

总结一下：

1、 静态成员和非静态成员函数，主要占据代码段内存，生成对象，不会再占用内存。
2、 非静态数据成员是影响对象占据内存大小的主要因素，随着对象数据的增加，非静态数据成员占据的内存会相应增加。
3、 所有的对象共享一份静态数据成员，所以静态数据成员占据的内存的数量不会随着对象的数据的增加而增加。
4、 如果对象中包含虚函数，会增加 4 个字节的空间，不论是有多少个虚函数。

对于 C++中的非内置类型的全局变量，其是属于.data 还是.bss 呢？

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

class c1
{
public:
    c1();
    c1(int i);
    ~c1();
    int n1;
};
```

```
c1::c1()
{
    n1=0;
    printf("n1=%d\n",n1);
};
c1::c1(int i)
{
    n1=i;
    printf("n1=%d\n",n1);
};

c1::~c1()
{
    ;
}

c1 g1;
c1 g2=10;

int main()
{
      pause();
      return 0;
}
```

这是一个 C++的例子，我的问题是 g1 和 g2 分别在哪一个节呢？

按照我们原来的标准，未赋初值的全局变量 g1，将位于.bss 节，赋初值了的 g2，将位于.data 节。可细一想，又不对劲，非内置类型的全局对象，需要调用构造函数将其构造出来，不能只通过 mmap 将其映射到内存就可以完成的。
头有些大了。

下面我来回答这个问题，实际上 g1 和 g2 全部位于.bss 节，编译器只是为其划分出了一段内存空间。
我们来验证一下：

```
>nm –f sysv hello
```

g1	|00010a08|	B	|	OBJECT|00000004|	|.bss
g2	|00010a0c|	B	|	OBJECT|00000004|	|.bss

那什么时候，对对象的成员变量赋值呢？

我们先来运行一下进程。

./hello
n1=0
n1=10

在上面的程序中，main 函数的第一句是 pause()，所以 main 函数刚一进入就停住了，而我们依然能够看 g1 和 g2 的构造函数打印出来的结果，很显然进入 main 函数之前，运行了 g1 和 g2 的构造函数。

还记得我们前面提到的.init_array 节吗，loader 在将程序焦点转移到 main 函数之前，其会运行.init_array 函数指针数组中的所有函数。

让我们来查看一下.init_array 中都有那些内容。

>objdump -s hello
……..
Contents of section .init_array:
 108fc 7c840000 14870000
……

108fc 是内存地址的一个序号，我们可以不用管它。7c840000 14870000 才是 init_array 中真正的内容。
在这里是以小端排序，我们试着翻译一下：
7c840000　　应该为　　0000847c
14870000　　应该为　　00008714

我们可以通过查看符号表，看看这两个地址都对应着什么内容。

>nm -n -C hello
0000847c t frame_dummy
000084a4 T c1::c1()
000084e8 T c1::c1()
0000852c T c1::c1(int)
00008574 T c1::c1(int)
000085bc T c1::~c1()
000085e0 T c1::~c1()
00008604 T main
0000861c t __static_initialization_and_destruction_0(int, int)
000086c4 t __tcf_1
000086ec t __tcf_0
00008714 t global constructors keyed to _ZN2c1C2Ev
00008734 T __libc_csu_init

这样就很清楚了，进程运行时，在调用 main 之前，要运行 frame_dummy 和 global constructors

keyed to _ZN2c1C2Ev。

如果还有兴趣的朋友，可以尝试着对进程进行反编译看看这两个函数到底做了什么事。

我们前文说到，对于 C 程序编写的进程来讲，在运行时，只是通过 mmap 为其数据段分配了一段虚拟内存，只有在实际用到才会分配物理内存。

而对于 C++编写的程序来讲，那些非内置类型的全局变量，由于在 main 函数之前，需要运行构造函数，为其成员变量赋值，这时虽然在你的程序里还没有用到，但它已经开始占用了物理内存。

2.3.5.3. 关于数据段的优化

我刚一跳到数据段的优化，可能有的人嘀咕，一个整型的全局变量也就 4 个字节，为这几个字节值得吗？

我的意见是，对于进程自身的数据段做优化，影响有限；但如果我们对动态库的数据段做优化，其作用将会比较明显。
比如一个动态库被 50 个进程所依赖，那么在这 50 个进程同时运行时，在系统中就会存在 50 个该动态库的数据段，每减少一个整型的全局变量，它将节省 4*50=200 个字节。

优化数据段的方法：
1、 尽可能的减少全局变量和静态变量。
我们可以使用“nm”来列出所有在.data 和.bss 节的变量，方便我们检查。

查看.data 节数据
#nm --format=sysv yourlib | grep -w .data

查看.bss 节数据
#nm --format=sysv yourlib | grep -w .bss

2、 对于非内置类型的全局变量，尽可能使用类指针来代替。
进程会 main 函数之前，运行所有非内置类型的全局变量的构造函数，一方面会降低进程的启动速度，另一方面，即使没有使用该全局变量，其也已经开始占用物理内存，造成浪费。
对于静态的非内置类型的静态对象，其在第一引用时创建出来。

3、 将只读的全局变量，加上 const，从而使其转移到代码段，利用代码段是系统共享的特性，来节省内存使用。但是对于非内置类型的变量，即使你使用 const 也不能将其转移到.rodata 段，因为其要运行构造函数，有可能对其成员变量赋值。

4、 关于字符串数组的优化：
例如：
static const char *errstr[]= { "message for err1", "message for err2", "message for

something"};

注：每个字符串的长度都不一样，第 3 个要比前两个长。

优化方法 1：

```
static const char errstr[][21]={...};
```

优化方法 2：

```
static const char msgstr[] = "message for err1\0"
                             "message for err2\0"
                             "message for something\0";
static const size_t msgidx[] = {
                             0,
                             sizeof ("message for err1"),
                             sizeof("message for err2"),
                             sizeof("message for something")
                            };
const char * errstr(int nr){
    return msgstr+ msgidx[nr];
}
```

优化方法 3：

```
static const char* getErrString(int id)
{
    switch(id)
    {
        case 0:
            retrun "message for err1";
            break;
        case 1:
            retrun "message for err2";
            break;
        case 2:
            retrun "message for something";
            break;
       default:
            return "";
     }
}
```

5、 不要在头文件中定义变量

我们来下看下面一段代码：

header.h

```
const int m=10;
```

f1.c

```
#include <stdlib.h>
#include <stdio.h>
#include "header.h"

void funca()
{
    printf("%d\n",m);
}
```

f2.c

```
#include <stdlib.h>
#include <stdio.h>
#include "m.h"

void funca();

int main()
{
    funca();
    return 0;
}
```

在上面的例子中，我们在 header.h 头文件中定义了一个常量 m=10；然后在 f1.c 和 f2.c 分别引用了这个头文件。

我们使用 gcc 进行编译：

```
gcc -o hello f1.c f2.c'
/tmp/ccl4RJIU.o(.data+0x0): multiple definition of `m'
/tmp/cc646fCS.o(.data+0x0): first defined here
collect2: ld returned 1 exit status
```

报错，说明 m 在多个文件里重复定义。

我们再使用 g++进行编译：

```
g++ -o hello a1.c a2.c'
```

很神奇，居然成功了。

下面我们再来查看一下编译出来的 hello 的符号表

```
nm hello
……
000106a8 W data_start
```

00008414 t frame_dummy
00008540 r m
00008544 r m
0000845c T main

这里，我们发现两个常量，都叫 m，具体原因，我们稍后再说。

我们再来修改一下 header.h，去掉 const。
header.h
int m=10;

我们再使用 g++，对齐进行编译。
g++ -o hello a1.c a2.c'
/tmp/ccYqaAsp.o(.data+0x0): multiple definition of `m'
/tmp/ccauTm0I.o(.data+0x0): first defined here
collect2: ld returned 1 exit status
这次却报错了，说明 const 对于在头文件中定义变量是有区别的。

下面我们就来说明一下原因。
编译器在编译文件时：
1、 首先将#include 的头文件，拷贝到当前的源文件中，合并成一个编译单元。
2、 对于所有的变量，在编译过程中具备一个属性，其是内部链接，还是一个外部链接。
内部链接：该变量只是在当前的编译单元生效，在同一个编译单元中，不允许有同名的变量。
外部链接：该变量不只局限于当前的编译单元，在所有编译单元中生效。
在同一编译单元中，同一标识符不应该同时具有内部链接和外部链接两种声明。
具备外部链接的标识符，应该只声明一次。

3、我们先来说明，为什么在 gcc 编译时，报告 multiple definition of `m'
gcc 中 const 声明的标识符，将具有外部链接属性，在所有的编译单元生效。
这样，在编译 a1.c 中，const int m=10 将被拷贝到 a1.c 中，这时在 a1.c 的编译单元中，定义了全局变量 m，在所有的编译单元生效。之后在编译 a1.c 中，const int m=10 将被拷贝到 a2.c 中，编译器在试图定义全局变量 m 的时候，发现了在 a1.c 编译单元中，定义同样的全局变量 m，故报错。

4、为什么在 g++编译时，就成功了呢？
在 g++中，const 声明的标识符，将具有内部链接属性，这是与 gcc 所不同的。这样在编译 a1.c 时，全局变量 m 只是在 a1.c 中生效，其他的编译单元，看不到该全局变量 m；在编译 a2.c 时，编译器又生成一个全局变量 m，其只是在 a2.c 中生效。

所以，我们才会看到在编译出来的 hello 中，有两个都叫 m 的符号
00008540 r m
00008544 r m
它们指向不同的地址。

而当我们修改 header.h 文件，去掉 const 时，变量的属性又被改回到外部链接，所以在编译时报错。

对于 static 声明的变量，不论是 gcc，还是 g++，都将是内部链接。

下面，我来总结一下：

1、 对于普通的全局变量来讲，其定义应该放在源程序（分配空间）中，在头文件中应该使用 extern 声明该变量（只声明，不分配空间）。
这样多个编译单元用到该全局变量时，将使用的是同一地址。

2、 对于 const 限定的全局变量，放在头文件中
使用 gcc 进行编译，该全局变量将具备外部链接属性，如果在多个编译单元中使用，将报错。
使用 g++进行编译，该全局变量将具备内部链接属性，如果在多个编译单元中使用，则编译器将创建多个同名，但不同地址的全局变量。

3、 对于 static 限定的全局变量，放在头文件中
该全局变量将具备内部链接属性，如果在多个编译单元中使用，则编译器将创建多个同名，但不同地址的全局变量。

因此，在头文件中定义变量，一方面有可能会使程序逻辑不对，另外也有会为不同的编译单元分配多块内存，造成内存的浪费。

而我们在检查应用程序、动态库的符号，发现有重名时，往往就是因为该标识符具备内部链接的属性，只对于某一个编译单元有效。

2.3.6. 代码段

对于代码段由于其在整个系统内共享，而且在内存不足时还能够回收，因此，实际上代码段对我们系统的内存使用影响不大。

这里我只介绍几点：

1、 在编译主程序时，不要使用“-export-dynamic”。

在缺省情况下，主程序不会导出其内部定义的函数和变量名。如果你想导出的话，可以在编译的时候，加上“-Wl ,-export-dynamic”选项。但这会增加代码段和数据的大小，占据更多的内存。

2、 删除冗余代码

有些人可能会说，既然是冗余代码，肯定就不会被用到，也就不会占用物理内存。但是由于有冗余代码的存在，有可能会使原本可以在一个物理页面存在的代码，要使用两个物理页面。因此，我们可以说冗余代码有可能导致代码段使用的物理内存增加。

另外，从性能的角度来看，由于冗余代码的存在，会增加页故障的数量，从而导致进程运行效率的下降。

我们可以使用 gcc 的编译选项，来检测冗余代码，并显示 warning 信息。

-Wunused：检查无用代码。
--Wunreachable-code：检查从未使用的代码。

我们举个例子，来说明两者的区别：

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

int func()
{
	return 10;
}

int main()
{
	int i=10;
	return 0;
}
```

func 进程根本不会运行到该函数，所以其为从未使用的代码。
int i=0，进程会运行到该函数，但是该行对最终结果没有任何效果，所以为无用代码。

```
>gcc -o hello hello.c -Wunused
hello.c: In function `main':
hello.c:12: warning: unused variable `i'
```

2.3.7. 使用 Thumb 指令

为兼容数据总线宽度为 16 位的应用系统， ARM 体系结构除了支持执行效率很高的 32 位 ARM 指令集以外，同时支持 16 位的 Thumb 指令集。 Thumb 指令集是 ARM 指令集的一个子集，允许指令编码为 16 位的长度。与等价的 32 位代码相比较， Thumb 指令集在保留 32 代码优势的同时，大大的节省了系统的存储空间。
所有的 Thumb 指令都有对应的 ARM 指令，而且 Thumb 的编程模型也对应于 ARM 的编程模型，在应用程序的编写过程中，只要遵循一定调用的规则， Thumb 子程序和 ARM 子程序就可以互相调用。当处理器在执行 ARM 程序段时，称 ARM 处理器处于 ARM 工作状态，当处理器在执行 Thumb 程序段时，称 ARM 处理器处于 Thumb 工作状态。
与 ARM 指令集相比较， Thumb 指令集中的数据处理指令的操作数仍然是 32 位，指令地址也为 32 位，但 Thumb 指令集为实现 16 位的指令长度，舍弃了 ARM 指令集的一些特性，如大多数的 Thumb 指令是无条件执行的，而几乎所有的 ARM 指令都是有条件执行的；大多数的 Thumb 数据处理指令的目的寄存器与其中一个源寄存器相同。

由于 Thumb 指令的长度为 16 位，即只用 ARM 指令一半的位数来实现同样的功能，所以，要实现特定的程序功能，所需的 Thumb 指令的条数较 ARM 指令多。在一般的情况下， Thumb 指令与 ARM 指令的时间效率和空间效率关系为：

Thumb 代码所需的存储空间约为 ARM 代码的 60％～70％

Thumb 代码使用的指令数比 ARM 代码多约 30％～40％

若使用 32 位的存储器，ARM 代码比 Thumb 代码快约 40％

若使用 16 位的存储器，Thumb 代码比 ARM 代码快约 40％～50％

与 ARM 代码相比较，使用 Thumb 代码，存储器的功耗会降低约 30％

显然， ARM 指令集和 Thumb 指令集各有其优点，若对系统的性能有较高要求，应使用 32 位的存储系统和 ARM 指令集，若对系统的成本及功耗有较高要求，则应使用 16 位的存储系统和 Thumb 指令集。当然，若两者结合使用，充分发挥其各自的优点，会取得更好的效果。

2.3.7.1. Thumb 指令的编译

hello.c

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

int main()
{
        printf("hello world!\n");
        return 0;
}
```

编译：

```
#gcc -o hello -mthumb a.c
```

注意，如果需要编译成 Thumb 指令，需要使用-mthumb 选项。

下面，我们来看一下，编译后的 Thumb 指令：

```
#objdump -d hello
00008418 <main>:
    8418:       b580            push    {r7, lr}
    841a:       466f            mov     r7, sp
    841c:       4802            ldr     r0, [pc, #8]    (8428 <.text+0xb8>)
    841e:       ff9ff7ff        bl      8360 <__plt_thumb_printf@@GLIBC_2.4>
    8422:       2000            mov     r0, #0
    8424:       46bd            mov     sp, r7
    8426:       bd80            pop     {r7, pc}
    8428:       84f4            strh    r4, [r6, #38]
```

从上面可以看出，其指令为 2 个字节，16 位。

2.3.7.2. ARM 程序和 Thumb 程序混合使用

通常，Thumb 程序比 ARM 程序更加紧凑，而且对于内存为 8 位或者 16 位的系统，使用 Thumb 程序效率更高。但是在下面一些场合下，程序必须运行在 ARM 状态，这时就需要混合使用 ARM 程序和 Thumb 程序。

1、 强调速度的场合：在有些系统中需要某些代码运行速度尽可能地快，这时应该使用 ARM 程序。系统可以采用少量的 32 位内存，将这段 ARM 程序在 32 位的内存中运行，从而尽可能地提高运行速度。
2、 有一些功能只有 ARM 程序能够完成。例如，使用或者禁止异常中断就只能在 ARM 状态下完成。
3、 当处理器进入异常中断处理程序时，程序状态自动切换到 ARM 状态。即在异常中断处理程序入口的一些指令是 ARM 指令，然后根据需要程序可以切换到 Thumb 状态，在异常中断处理程序返回前，程序再切换到 ARM 状态。
4、 ARM 处理器总是从 ARM 状态开始执行。因而，如果要在调试器中运行 Thumb 程序，必须为该 Thumb 程序添加一个 ARM 程序头，然后再切换到 Thumb 状态，调用该 Thumb 程序。

a1.c

```
#include <stdlib.h>
#include <stdio.h>
#include "m.h"

void funcb();

void funca()
{
    funcb();
    printf("%d\n",m);
}
```

a2.c

```
#include <stdlib.h>
#include <stdio.h>
#include "m.h"

void funcb();

void funca()
{
    funcb();
    printf("%d\n",m);
}

zch07elgi01:/home/jkvp74/bookcode> cat a2.c
```

```
#include <stdlib.h>
#include <stdio.h>
#include "m.h"

void funca();

void funcb()
{
    printf("abcd\n");
}

int main()
{
    funca();
    return 0;
}
```

我们将 a1.c 编译成 thumb 指令:

```
#gcc -o a1.o -c -mthumb a1.c
```

将 a2.c 编译成 ARM 指令：

```
#gcc -o a2.o -c a2.c'
```

将其链接成一个可执行文件

```
#gcc -o hello a1.o a2.o
```

我们下载到设备上，运行。

```
# ./hello
abcd
10
```

下面我们再来对编译後的 hello，进行反编译：

```
#objdump -d hello
……
00008418 <funca>:
    8418:    b580        push    {r7, lr}
    841a:    466f        mov     r7, sp
    841c:    f880f000    bl      8520 <__funcb_from_thumb>
    8420:    4803        ldr     r0, [pc, #12]    (8430 <.text+0xc0>)
    8422:    4b04        ldr     r3, [pc, #16]    (8434 <.text+0xc4>)
    8424:    681b        ldr     r3, [r3, #0]
    8426:    1c19        mov     r1, r3           (add r1, r3, #0)
    8428:    ff9af7ff    bl      8360 <__plt_thumb_printf@@GLIBC_2.4>
    842c:    46bd        mov     sp, r7
```

```
    842e:       bd80            pop     {r7, pc}
    8430:       8548            strh    r0, [r1, #42]
    8432:       0000            lsl     r0, r0, #0
    8434:       067c            lsl     r4, r7, #25
    8436:       0001            lsl     r1, r0, #0

00008438 <funcb>:
    8438:       e1a0c00d        mov     ip, sp
    843c:       e92dd800        stmdb   sp!, {fp, ip, lr, pc}
    8440:       e24cb004        sub     fp, ip, #4      ; 0x4
    8444:       e59f0004        ldr     r0, [pc, #4]    ; 8450 <.text+0xe0>
    8448:       ebffffc5        bl      8364 <__plt_arm_printf@@GLIBC_2.4>
    844c:       e89da800        ldmia   sp, {fp, sp, pc}
    8450:       0000854c        andeq   r8, r0, ip, asr #10

00008454 <main>:
    8454:       e1a0c00d        mov     ip, sp
    8458:       e92dd800        stmdb   sp!, {fp, ip, lr, pc}
    845c:       e24cb004        sub     fp, ip, #4      ; 0x4
    8460:       eb000030        bl      8528 <__funca_from_arm>
    8464:       e3a00000        mov     r0, #0  ; 0x0
    8468:       e89da800        ldmia   sp, {fp, sp, pc}

……

00008520 <__funcb_from_thumb>:
    8520:       4778            bx      pc
    8522:       46c0            nop                     (mov r8, r8)

00008524 <__funcb_change_to_arm>:
    8524:       eaffffc3        b       8438 <funcb>

00008528 <__funca_from_arm>:
    8528:       e59fc000        ldr     ip, [pc, #0]    ; 8530 <__funca_from_arm+0x8>
    852c:       e12fff1c        bx      ip
    8530:       00008419        andeq   r8, r0, r9, lsl r4
```

从上面我们可以看到，为了 funca 为 Thumb 指令，funcb 和 main 为 ARM 指令，为了支持 ARM 指令和 Thumb 指令的调用，GCC 又安排生成了 3 个函数__funcb_from_thumb、__funcb_change_to_arm 和__funca_from_arm。

2.4. 动态库

动态库技术是当前程序设计中经常采用的技术。其目的减少程序的大小，节省空间，提高效率，具有很高的灵活性。采用动态库技术对于升级软件版本更加容易。与静态库不同，动态库里面的函数不是执行程序本身的一部分，而是根据执行需要按需载入，其执行代码可以同时在多个程序中共享。

天下没有免费的午餐，由于在编译过程中无法知道动态库函数的地址，不得不选择运行期间查找，对程序的性能会有所损失。

动态库有两种其加载方式：
1、静态加载
在程序编译的时候加上“-l”选项，指定其所依赖的动态库，这个库的名字将记录在 ELF 文件的.dynamic 节。在程序运行时，loader 会预先将程序所依赖的所有动态库都加载在进程空间中。

静态加载的优点：
动态库的接口调用简单，可以直接调用。

缺点：
动态库的生存周期等于进程的生存周期，其加载时机不灵活。

2、动态加载
我们还可以在程序中编码来指定加载动态库的时机，经常使用的函数 dlopen dlclose。

动态加载的优点：
动态库加载的时机非常灵活，可以非常细致的定义动态库的生存周期。

动态加载的缺点:
动态库的接口调用起来比较麻烦，同时还要关注动态库的生存周期。

前面我们在介绍进程时，分为了 4 段：只读的代码段、可修改的数据段、堆段和栈段。

对于共享库来讲，它只包括 2 个段：只读的代码段和可修改的数据段。堆和栈段，只有进程才有。如果你在共享库的函数里，分配了一块内存，这段内存将被算在调用该函数的进程的堆中。

对于共享库的代码段和数据段，系统的处理和对主程序的代码段、数据段一样：
代码段由于其内容是对每个进程都是一样的，所以它在系统中是唯一的，系统只为其分配一块内存，多个进程之间共享。
数据段由于其内容对每个进程是不一样的，所以在链接到进程空间后，系统会为每个进程创建相应的数据段。

也就是说如果一个共享库，被 N 个进程链接，当这 N 个进程同时运行时，同时共享一个代码段，每个进程拥有一个数据段，系统中共有 N 个数据段。

2.4.1. 数据段

2.4.1.1. 共享库中的 bss

我们前面讲过，在进程中的 bss，如果数据段容纳不下，那么它将使用堆段内存。那么在共享库中，它没有对应的堆段，会如何呢？

让我们看一下下面的例子：

a.c

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

int bss[1024*128]={0};
void a1()
{
    printf("bss:%p\n",bss);
    return;
}
```

```
>gcc –shared –fPIC   a.c –o liba.so
```

hello.c

```
#include <stdlib.h>
#include <stdio.h>
int funca();
int main()
{
   int pid=getpid();
   printf("pid:%d\n",pid);
   funca();
   pause();
}
```

```
>gcc –L./ -la hello.c –o hello
```

下载到嵌入式设备中

```
# ./hello
pid:458
bss:0x40009798
```

```
# cat maps
00008000-00009000 r-xp 00000000 1f:12 294        /mnt/msc_int0/hello
00010000-00011000 rw-p 00000000 1f:12 294        /mnt/msc_int0/hello
40000000-40001000 rw-p 40000000 00:00 0
40001000-40002000 r-xp 00000000 1f:12 295        /mnt/msc_int0/liba.so
40002000-40009000 ---p 00001000 1f:12 295        /mnt/msc_int0/liba.so
40009000-4000a000 rw-p 00000000 1f:12 295        /mnt/msc_int0/liba.so
4000a000-4008b000 rw-p 4000a000 00:00 0
41000000-41017000 r-xp 00000000 1f:0d 818683     /lib/ld-2.3.3.so
4101e000-41020000 rw-p 00016000 1f:0d 818683     /lib/ld-2.3.3.so
41028000-41120000 r-xp 00000000 1f:0d 818916     /lib/libc-2.3.3.so
41120000-41128000 ---p 000f8000 1f:0d 818916     /lib/libc-2.3.3.so
41128000-41129000 r--p 000f8000 1f:0d 818916     /lib/libc-2.3.3.so
41129000-4112c000 rw-p 000f9000 1f:0d 818916     /lib/libc-2.3.3.so
4112c000-4112e000 rw-p 4112c000 00:00 0
befeb000-bf000000 rwxp befeb000 00:00 0
```

```
# cat memmap
 2
 1
 1
 2
 0 0 0 0 0 0 0
 1
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
5050505050505050501750505050505050505050505050
 1 1
5018181818121818181818181818185050 0 050 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 050 0 0 0
0 0 0 0 0 0 0 045 043434343 0 0 0 042 0 0 0 0 0 0 0 0 0 0 0 0 0 04947 0 0 0 0 0 0 0 0474749 0 0 0
0 0 05050 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 050 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 04950 0 0 0 049 0 0 0 047 050 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 047 0 0 0 0 0 0 0 0 0 0 0 0 0 045 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0
50
 1 1 1
 1 1
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
```

可以看到对于共享库的 bss 节的数据，如果数据段不能容纳的话，进程将会新创建一内存段来容纳 bss 节的数据。虽然 bss 的数据我们没有使用，但它仍然占用了一个物理页面。

如果在进程中引用了共享库的全局变量将会如何呢？

进程会扩展它的堆栈段，并将 bss 段的数据拷贝到堆栈段中来。

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

extern int bss[1024*128];
int main()
{
        a1();
        pid_t pid=getpid();
        printf("pid:%d a:%p p:\n",pid,bss);
        pause();
        return 0;
}
```

运行结果

```
bss:0x10908
pid:464 a:0x10908
```

查看 maps

```
00008000-00009000 r-xp 00000000 1f:12 288              /mnt/msc_int0/hello
00010000-00011000 rw-p 00000000 1f:12 288              /mnt/msc_int0/hello
00011000-00091000 rwxp 00011000 00:00 0
40000000-40001000 rw-p 40000000 00:00 0
40001000-40002000 r-xp 00000000 1f:12 289              /mnt/msc_int0/liba.so
40002000-40009000 ---p 00001000 1f:12 289              /mnt/msc_int0/liba.so
40009000-4000a000 rw-p 00000000 1f:12 289              /mnt/msc_int0/liba.so
4000a000-4008a000 rw-p 4000a000 00:00 0
40093000-40094000 rw-p 40093000 00:00 0
41000000-41017000 r-xp 00000000 1f:0d 818669         /lib/ld-2.3.3.so
4101e000-41020000 rw-p 00016000 1f:0d 818669         /lib/ld-2.3.3.so
41028000-41120000 r-xp 00000000 1f:0d 818902         /lib/libc-2.3.3.so
41120000-41128000 ---p 000f8000 1f:0d 818902         /lib/libc-2.3.3.so
41128000-41129000 r--p 000f8000 1f:0d 818902         /lib/libc-2.3.3.so
41129000-4112c000 rw-p 000f9000 1f:0d 818902         /lib/libc-2.3.3.so
4112c000-4112e000 rw-p 4112c000 00:00 0
befeb000-bf000000 rwxp befeb000 00:00 0
```

我们可以看到进程分配了一个堆栈区，00011000-00091000 rwxp 00011000 00:00 0；
同时也为 liba.so 分配了一个数据段 4000a000-4008a000 rw-p 4000a000 00:00 0。

如果我们查看 memmap 的话，会发现

```
2
1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1
 2
 0 0 0 0 0 0 0
 1
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

虽然我们没有使用 bss 数据，但系统还是占用了物理页面，这可能与进程复制共享库的数据段所造成的。

因此，不要在进程中通过 extern 的方式引用共享库中的全局变量；一旦引用，不论其是否使用，都将会占用物理内存。同时还会增加系统启动时，内存复制的代价，会导致性能的下降。

2.4.1.2. 共享库数据段对进程数据段的影响

前面说了，如果全局变量声明在共享库中，在进程中使用，则该变量被复制到进程的数据段，同时修改使用该变量的共享库的指向。

那么它对进程的数据段有什么影响呢？

hello.c

```
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>
#include <unistd.h>
extern int a[1024*30];
int main()
{
        a1();
        int i=1;
        for(i=0;i<1024*30;i++)
              a[i]=1;
        pid_t pid=getpid();
        printf("pid:%d a:%p p:%p\n",pid,a,p);
        pause();
        return 0;
}
```

a.c

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

int a[1024*30]={1};
void a1()
```

```
{
        return;
}
```

前面我们说过，全局变量的声明，将会影响生成可执行文件的大小。
编译出来之后，发现 liba.so 文件尺寸大了很多，而 hello 的文件大小基本没变。这说明，liba.so 的数据段增大了很多，而 hello 的数据段则没什么变化。
运行一下看看：

```
# ./hello
pid:700 a:0x10988 p:0x2f050
```

查看对应的 maps

```
00010000-00011000 rw-p 00000000 1f:12 308            /mnt/msc_int0/hello
00011000-0002f000 rwxp 00011000 00:00 0
40009000-40028000 rw-p 00000000 1f:12 303            /mnt/msc_int0/liba.so
```

通过 maps 我们发现，进程的数据段映射没有变化，而 liba.so 的数据段大小对应着声明的数组大小。
而我们打印出来的全局变量的数组指针，则明确的指向了进程的数据段，这是为什么呢？
我们可以看到堆栈段异常增大，在主程序没有一个 malloc 的情况下，居然为堆申请了 120k 的空间。
我们也注意到，数据段的地址与堆栈段的地址是连着的，这也就是说，当主程序链接了一个共享库的全局变量时，它会为该变量定义一个地址，但它不会影响数据段的大小，将该值拷贝到这个地址上。如果地址段不够用，它将占用堆段，系统将调用 brk 来扩展堆段。

我们通过 strace 可以来验证这个观点：

```
execve("./hello1", ["./hello1"], [/* 9 vars */]) = 0
uname({sys="Linux", node="(none)", ...}) = 0
brk(0)                                    = 0x2f000
```

我们再来查看一下其物理内存的使用情况：

```
cat maps
00010000-00011000 rw-p 00000000 1f:12 269            /mnt/msc_int0/hello
00011000-00411000 rwxp 00011000 00:00 0
40009000-4040a000 rw-p 00000000 1f:12 267            /mnt/msc_int0/liba.so
1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1

我们可以看到，在堆段占用了 4M 的内存，而在 liba 的数据段，其数据段的物理页面都被引用了 2 次，因此其是复用的，整个进程应该只是使用了 4M 的内存。

我们可以通过系统整体的空闲内存变化，来确认。

2.4.2. 代码段

2.4.2.1. 符号解析

hello.c

```
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>
#include <unistd.h>
void a();
extern int b;
int main()
{
        a();
        pid_t pid=getpid();
        printf("pid:%d b=%d %p\n",pid,b,&b);
        pause();
        return 0;
}
```

a.c

```
#include <stdlib.h>
#include <stdio.h>
int b=10;
int c=1;
void a()
{
        printf("function a b:%p c:%p\n",&b,&c);
        b++;
        return;
}
```

我们将 a.c 编译成 liba.so

```
gcc -fPIC -shared    a.c -o liba.so
```

编译 hello.c 为 hello，并且联接上 liba.so。

运行结果为：

```
# ./hello
function a b:0x108f0 c:0x400097f0
pid:644 b=11 0x108f0
```

问题一：在进程 hello 中，如何将 void a()，与 extern int b 与共享库 liba.so 里面的信息对应上的？
这这里不想很细致的讲解进程是如何一步一步 load 共享库的，先一知半解吧。

前面我们讲解过 elf 文件的一个大概结构。
elf 其中有两个 section：
.rel.dyn .rel.plt 两个节主要负责共享库的重定向工作，那么我们就来看看他们的内容。

readelf –r hello

Relocation section '.rel.dyn' at offset 0x4e8 contains 2 entries:
 Offset Info Type Sym.Value Sym. Name
00010860 00000114 R_ARM_COPY 00010860 b

Relocation section '.rel.plt' at offset 0x4f8 contains 5 entries:
 Offset Info Type Sym.Value Sym. Name
00010848 00001016 R_ARM_JUMP_SLOT 00008560 a
00010850 00001316 R_ARM_JUMP_SLOT 00008578 printf

我们可以看到变量 b 在.rel.dyn 节中，而 hello 中用到的外部函数 a,printf 在.rel.plt 中。

下面我们再来看看 liba.so。
readelf –a liba.so
我们在符号表中，可以看到下面的信息。
Relocation section '.rel.dyn' at offset 0x464 contains 8 entries:
 Offset Info Type Sym.Value Sym. Name
000087d4 00000915 R_ARM_GLOB_DAT 000087ec b

Symbol table '.dynsym' contains 28 entries:
 Num: Value Size Type Bind Vis Ndx Name
 9: 00008774 4 OBJECT GLOBAL DEFAULT 20 b
 23: 000005a0 48 FUNC GLOBAL DEFAULT 10 a

在这里我们可以看到 a 和 b 的定义。

因此，我们可以通过进程的.rel.dyn 和.rel.pltsection 和共享库中的符号，找出进程与共享库之间的调用关系。

问题二：同样是在 liba.so 中声明的全局变量，为什么 b 的地址为 0x108f0，而 c 的地址为 0x400097f0。
我们先根据打印出来的 pid 值，到 proc 对应的目录下，查看 maps 信息。
00008000-00009000 r-xp 00000000 1f:12 293 /mnt/msc_int0/hello
00010000-00011000 rw-p 00000000 1f:12 293 /mnt/msc_int0/hello
40000000-40001000 rw-p 40000000 00:00 0

```
40001000-40002000 r-xp 00000000 1f:12 295          /mnt/msc_int0/liba.so
40002000-40009000 ---p 00001000 1f:12 295          /mnt/msc_int0/liba.so
40009000-4000a000 rw-p 00000000 1f:12 295          /mnt/msc_int0/liba.so
4000a000-4000b000 rw-p 4000a000 00:00 0
41000000-41017000 r-xp 00000000 1f:0d 819429       /lib/ld-2.3.3.so
4101e000-41020000 rw-p 00016000 1f:0d 819429       /lib/ld-2.3.3.so
41028000-41120000 r-xp 00000000 1f:0d 819662       /lib/libc-2.3.3.so
41120000-41128000 ---p 000f8000 1f:0d 819662       /lib/libc-2.3.3.so
41128000-41129000 r--p 000f8000 1f:0d 819662       /lib/libc-2.3.3.so
41129000-4112c000 rw-p 000f9000 1f:0d 819662       /lib/libc-2.3.3.so
4112c000-4112e000 rw-p 4112c000 00:00 0
befeb000-bf000000 rwxp befeb000 00:00 0
```

从这里，我们可以得出 0x108f0 属于 hello 的数据段，而 0x400097f0 则属于 liba.so 的数据段，这是为什么呢？
在 a.c 的.rel.dyn 节中，我们发现，不论 b,c 是否被外界应用，都被添加到该节中，以供进程或其他共享库重定位。

而我们在 hello 的.rel.dyn 中发现
00010860 00000114 R_ARM_COPY 00010860 b

R_ARM_COPY ：
链接编辑器创建这种重定位类型的目的是支持动态链接。其偏移量成员引用某个可写段中的某个位置。符号表索引规定符号应该既存在于当前目标文件中，也存在于某个共享目标中。在执行过程中，动态链接器把与共享目标的符号相关的数据复制到由偏移给出的位置。
也就是说，在加载 liba.so 后，以及查找全局变量 b 的时候，会将 liba.so 中的 b 复制到自己的数据段，并且修改 liba.so 中的 b 的指向。
这主要是因为进程不会为这些共享库的变量，做重定向，它只是把该数据复制到自己的数据段中，然后要求对应的共享库修改其对应的指向。

问题三：如果我在主进程声明一个全局变量，而在共享库中引用，那么该全局变量会在哪个数据段呢？
答案:该变量在进程的数据段中，系统只需要修改共享库中该变量的重定向即可。

问题四：如果我在一个共享库中声明了一个全局变量，而在另外一个共享库中，使用，那么该全局变量将会如何呢？
该全局变量将位于声明它的共享库的数据段。

总结：
1、 如果全局变量声明在进程中，在共享库中使用，则该变量位于进程的数据段。
2、 如果全局变量声明在共享库中，在进程中使用，则该变量被复制到进程的数据段，同时修改使用该变量的共享库的指向。
3、 如果该变量在共享库中声明，在共享库中使用，则该变量位于声明它的共享库的数据段中。

2.4.2.2. 关于代码段共享

gcc-shared –fPIC -o 1.so1.c
这里有一个-fPIC 参数
PIC 就是 position independent code
PIC 使.so 文件的代码段变为真正意义上的共享。
如果不加-fPIC,则加载.so 文件的代码段时,代码段引用的数据对象需要重定位, 重定位会修改代码段的内容,这就造成每个使用这个.so 文件代码段的进程在内核里都会生成这个.so 文件代码段的 copy.每个 copy 都不一样,取决于这个.so 文件代码段和数据段内存映射的位置.

2.4.2.3. 导出函数对代码段的影响

在共享库中所定义的函数，缺省都是导出函数，让我们来看看函数的增加对代码段的影响。
每增加一个函数
.hash=.hash+4
.dynsym=.dynsym+16
.dynstr=.dynstr+函数名长度+1
.gnu.version=.gnu.version+2
.text 节按函数代码长度有所增长。

每增加一个全局变量
.hash=.hash+4
.dynsym=.dynsym+16
.dynstr=.dynstr+变量名长度+1
.gnu.version=.gnu.version+2
.data 节按函数代码长度有所增长。

从这里可以看出，我们每增长一个导出函数，不包括函数自身代码变化外，需要增加 24+函数名长度个字节。
而实际上，我们在共享库代码中，有很多函数和变量，只是在共享库内部使用，不需要导出，因此只要我们可以精确定义出外部使用的函数，那么我们就可以节省.dynsym 和.dynstr 等 section 的大小，从而节省内存。
现在我们使用 gcc 的--version-script 选项，来指定我们只是导出 a1()，其余函数只是共享库内部可见。
a.ld

```
{
      global:
          a1;
      local:*;
}
```

gcc –fPIC –c a.c –o a.o
ld –shared –version-script=a.ld –o liba.so a.o

2.4.2.4. 删除多余的导出符号

这里我们还是只关注直接依赖，因为直接依赖对于我们程序的编译、链接是直接相关的。

找出多余的导出符号
1、使用 readelf 读取动态库的 dynamic 节，找到其所有直接依赖的动态库。
2、为每一个动态库建立一个依赖于它的动态库和进程表。
3、将该动态库所有的导出符号减去依赖于它的进程（或动态库）所有 weak 或未定义的符号，剩下的就是多余的符号。

2.4.3. 动态库的优化

由于一个动态库的数据段所占内存，会随着依赖它的进程的数量成线性增长；而其代码段则系统共享，所以我们还是将优化的重点放在动态库的数据段。

不要太小看了动态库的优化，按照现在系统中有 50 个进程依赖于这个动态库，这个动态库的数据段减少 4K，那么系统整个的内存将节约 200K。

2.4.3.1. 减少 bss 节的数据

在做动态库数据段优化的时候，有这么一个概念：在加载动态库的时候，.bss 节的数据将被单独放在一个数据段中，试想 1k 的 data 数据和 1k 的 bss 数据，其要分别占用 2 个 4K 的物理页面，浪费了 6k。如果将 bss 的数据全部赋上初值，转化为 data 段，这样就可以转化成 1 个 2k 的数据段，占用一个物理页面，从而节省出一个物理页面。

这种优化思路，关键在于在加载动态库的时候 bss 数据单独分出一个内存段来。

可实际上，在最后一个页面，如果还有剩余空间的话，会使用 bss 去补齐，因此就不会出现上面的问题。

所以说减少 bss 节的数据，并不会带来内存数量的减少。

2.4.3.2. 无用的动态库

我们现在的程序有个很不好的习惯，为了自己写程序方便，共享库不管是否真正的使用，都链接进来再说，反正 Linux 是延迟分配内存，只要是不真正用到，就不会占用实际内存。

实际上并非如此，loader 在加载动态库时需要做很多事情，对于代码段来讲其要做一些重定

位的工作，至少占用一个物理页面；对于数据段来讲，首先要做一些重定位的工作，在映射数据段时，还要在最后一个页面，.data 节未填充完的剩余字节填充为 0，其最少要占用一个物理页面。也就是说因为一个无用的共享库，我们至少损失了 8k 的物理内存。对于每个不管是直接依赖还是间接依赖该动态库的进程，都将损失 8K 的物理内存。

如何来发现无用的动态库？
不要让程序员一个个去验证其负责程序所依赖的动态库，这将是一个很令人烦躁的工作，我推荐使用 perl 脚本来编写一个工具，可以定期的去检测。

例如我们来检测动态库 A 其是否链接了无用的动态库。
1、 使用 readelf 读取动态库 A 的 dynamic 节，获取其所直接依赖的动态库。
注意：不要使用 ldd，那样结果会包含间接依赖的动态库，对我们的结果有害无异。
2、 读取动态库 A 中所有 Weak 和 undefine 的符号；
3、 读取动态库 A 所直接依赖动态库所有已经定义的符号；
4、 将上面的结果取个交集，剔除一些系统的符号，如果交集不为空，我们就认为两个库有实际的依赖关系，否则就是链接了无用的动态库。

2.4.3.3. 动态库的合并

试想一个情景，动态库 A 的数据段为 1K，动态库 B 的数据段为 1K，进程加载两个动态库，就需要 2 个页面 8K 内存。如果我们将这两个动态库合并到一起，那么合并后动态库的数据段所占内存为 4K，节省出一个物理页面。
不要小看动态库数据段内存减少的一个物理页面，其节约的内存是与所依赖于它的进程成线性关系，如果有 50 个进程依赖于它，那么会节省 50*4=200k。

问题的关键是：大库和小库。

大库
优点：数据段都合并到一起，节约了内存。
缺点：进程可能会因此引入很多无用的内容，对于 C 语言的动态库来讲，在加载动态库时，只是为数据段分配了一块虚存，没有用到的全局变量并不会占用更多的内存；而对于 C++来讲，对于非内置类型的全局变量，在加载完后需要调用其构造函数，会产生 dirty page，反而有可能增加内存使用。

小库：
优点：进程可以只加载那些它所需要的内容，对于 C++的库来讲有可能会节省内存。
缺点：会产生很多的内存段，造成最后一个页面部分空间的浪费。

因此，对于 C 语言写的库来讲，动态库的合并会通过合并数据段来节省内存。对于 C++编写的动态库来讲，有可能会由于无用的非内置类型的全局变量的增加而增加内存消耗。

但如果两个动态库在一个进程中同时出现，那么动态库的合并从内存角度来讲对于进程有益无害。

关于动态库的合并，我们可以总结如下几点：
1、 对于 C 语言编写的动态库，合并没有什么害处。
2、 对于 C++语言编写的动态库，可以考虑将一些不常用的功能分拆，使用 dlopen 的方式加载，来节省内存。
3、 防止由于动态库之间的合并，导致进程加载很多无关的内容。
4、 对于经常一同出现的动态库，可以合并。

2.4.3.4. 仅被依赖一次的动态库

下面我们再来考虑一种特殊情形：
一个动态库 A 只被某一个动态库 B 或进程 B 所依赖，那么我们可以说这个动态库 A 和动态库（或进程）B 是 100%同时出现的，那么我们可以将动态库 A 做成一个静态库，然后将其与动态库（或进程）B 进行合并，从内存的角度来讲这是有益无害的。

查找仅使用一次的动态库：
这里我们还是要关注直接依赖，我们要检测整个系统中所有的动态库。
1、 使用 readelf 读取动态库的 dynamic 节，找到其所有直接依赖的动态库。
2、 为每一个动态库建立一个依赖于它的动态库和进程表。

表长度为 1 的动态库，即是是仅被依赖一次的动态库。

2.4.3.5. 使用 dlopen 来控制动态库的生存周期

动态加载共享库 dlopen

我们在启动程序后，会发现进程加载了很多的库，有些库是我们很少用到的，有些库我们可能只用一次，而进程一上来就全部加载进来，每个库最少 8k（4k 代码段，4K 数据段）实在显得是非常浪费。更要命的是，动态库的数据段一旦被使用后，便无法释放，将会导致动态库所占用的内存越来越多。

好在 Linux 中提供了动态加载共享库的机制，我们可以按照自己的需要来加载共享库。dlopen 函数的使用，我在这里就不多说了，我讲一下它的实现机制。
根据前面 ELF 文件的介绍，在 dynamic 节记载了这个库依赖于哪些共享库。
Dynamic section at offset 0x6c8 contains 24 entries:

Tag	Type	Name/Value
0x00000001	(NEEDED)	Shared library: [libc.so.6]

在 dlopen 一个共享库时，
1、 进程会加载该共享库的 txt 段和数据段，同时为这个共享库计数加 1。
2、 进程查找该共享库的 dynamic 节，查看其所依赖的共享库。

3、 首先检查所依赖库是否已经被加载，如果以被加载，则为这个共享库计数加 1.
如果未被加载，则加载其 txt 段和 data 段，然后为这个共享库计数加 1。
4、 再查找这些库所依赖的库。

最终进程会为每个加载的共享库维护一个依赖的计数。

在 dlclose 共享库时：
1、 首先将该共享库的计数减 1，如果该共享库依赖计数为 0，则卸载该共享库。
2、 在 dynamic 节中，查找其所依赖的共享库。
3、 为每个共享库的计数减 1，如果该共享库依赖计数为 0，则卸载该共享库。
4、 重复上面的步骤。

dlopen 的好处在于：
1、 可以在程序启动的时候，减少加载库的数量，这样可以加快进程的启动速度和减少加载库的内存使用。
2、 为进程提供了卸载共享库的机会，这样就可以回收共享库代码段和数据段所占用的内存。

缺点在于：
1、 对于程序员编码来讲，很不方便。

现在使用 dlopen 来提高程序性能和节约内存，用的越来越多。

下面我列举一些采用 dlopen 优化时，经常遇到的问题：

1、 dlopen 的嵌套

为了说明这个问题，我们举一个例子:
进程 P;
共享库 liba.so:
依赖于:libb.so
共享库 libc.so:
依赖于 libd.so.

1、 进程 P，通过 dlopen 调用 liba.so
进程将 liba.so 和 libb.so 加载到进程中。
2、 进程 P，调用 liba.so 的库函数，dlopen 加载 libc.so。
进程将 libc.so 和 libd.so 加载到进程中。
3、 这时进程 P，调用 dlclose 来关闭 liba.so。
dlclose 会查找其所依赖的库，那么它只能查到 libb.so，所以它将关闭 libb.so 以及 liba.so.

这样虽然进程 P 关闭了 liba.so，但是进程所加载的共享库却增加了 libc.so 和 libd.so，导致共享库没有完全卸载。

这个问题的难点在于，我们需要一个时机，在我们 dlclose 一个库时，将这个库所有 dlopen 的库关闭。如果我们在每个 dlclose 的时候，去检查这个库是否又打开了其他的库，那么就需要上层的函数需要了解库的详细的实现细节，不符合封装的概念，而且对于上层函数实现起来也十分困难。

好在 libc 的共享库提供了一个机制，帮助我们实现这个封装。

库的构造和析构函数：

关于构建与析构函数，一般不需要自己去编程实现。如果你一定要自己做，下面是函数原型：

void __attribute__ ((constructor)) my_init(void);

void __attribute__ ((destructor)) my_fini(void);

在编译共享库时，不能使用"-nonstartfiles"或"-nostdlib"选项，否则，构建与析构函数将不能正常执行(除非你采取一定措施)。

2、lopen 有可能导致内存泄漏

下面我举一个常见的例子：

在动态库中，我要使用进程中唯一的对象 mInstance,标准的写法：

liba.so 的代码

```
static Myclass *mInstance;          //声明一个静态 Myclass 对象指针。

Myclass getInstance()
{
        if(NULL == mInstance)
        {
                mInstance = new Myclass;
         }
         return mInstance;
}
```

上面的代码，大家应该非常熟悉了，而在引入了 dlopen 之后，有可能会导致内存泄漏。

下面，我使用伪代码来说明这个问题。

```
1           dlopen(liba.so);
2           pInstance=getInstance();
            …….
3           dlclose(liba.so);
            …….
4           dlopen(liba.so);
5           pInstance=getInstance();
            …….
```

我来解释一下有可能导致的内存泄漏。

1、 加载了动态库 liba.so，同时初始化了其数据段，这时 mInstance 应该为空。
2、 在 getInstance 中，看到 mInstance 为空，则在堆中分配了一块内存，生成一个 Myclass 实例，同时为数据段的 mInstance 赋值。
3、 卸载了动态库 liba.so，这时 mInstance 是不存在的，也就意味着我们丢失了在堆中生成的 Myclass 对象实例。
4、 加载了动态库 liba.so，同时初始化了其数据段，这时 mInstance 应该为空。
5、 在 getInstance 中，看到 mInstance 为空，则在堆中又分配了一块内存，生成一个 Myclass 实例，同时为数据段的 mInstance 赋值。

这样，每做一次这样的时间循环，将会导致一个 Myclass 对象内存泄漏。

这个问题的实质是：
在我们的心目中，一个 static 的对象的生存周期是贯穿在进程始终的，实际上不是这样。在动态库中的 static 对象，其生命周期等于该动态库的生命周期。采用静态链接的方式，动态库的生命周期等于进程的声明周期；而采用动态加载的方式，则是不同的。

为了避免上面的问题出现，我们要在动态库卸载的时候，释放掉该块内存。

有两个方法：
1、 我们写一个释放内存的函数，在调用 dlclose 之前，调用该函数。
2、 我们在动态库的析构函数中，释放这块内存。

```
void __attribute__ ((destructor)) my_fini(void);
```

在这里我推荐第二个方法，它对不需要上层用户做任何改动，对其完全透明。

2.5. 线程

如果我们在嵌入式设备上，ps 列出当前所有的进程，你可以发现有很多具有同样名字的进程，他们的 PID 是不同的。那么它很有可能是一个进程中的多个线程。你可以通过在 /proc/pid/status 里面查看其父进程，来判断他们之间的关系。

那么对于线程来讲，不同的 gid，他们之间的代码段、数据段、堆段、已经栈段又是如何呢？在 linux 中线程有两种实现方式，一种 pthread，另一种是 NPTL，在这里我就不详细介绍了。关于内存方面，我这里只讲述最常用的一种 pthread。

从编程的角度来讲，线程区别进程的重要一点，就是进程拥有自己的地址空间，而线程则是完全共享的。从这个角度来讲，线程之间的数据段是完全共享，任何一个线程都可以访问进程的全局变量；堆段也是全局共享的，一个线程中申请到的一块内存，在另一个线程也能访问，堆是相对于进程的概念；栈段，每个线程都要有自己的运行过程，如果多个线程的栈交织在一起，那么线程将会如何运行呢？

我们来看个例子：

```
#include <stdio.h>
#include <pthread.h>
void* thread_proc(void* param)
{
	int	first = 0;
	int* p0 = malloc(1024);
	printf("(0x%x): first=%p\n",	pthread_self(), &first);
	printf("(0x%x): p0=%p p1=%p \n", pthread_self(), p0);
	return 0;
}
#define N 5
int main(int argc, char* argv[])
{
	int first = 0;
	int i= 0;
	void* ret = NULL;
	pthread_t tid[N] = {0};
	printf("first=%p\n", &first);
	for(i = 0; i < N; i++)
	{
		pthread_create(tid+i, NULL, thread_proc, NULL);
	}
	pause();
	return 0;
}
```

运行结果：

```
first=0xbefffe8c
(0x4002): first=0xbe7ffb28
(0x4002): p0=0x13038
(0x8003): first=0xbe5ffb28
(0x8003): p0=0x13440
(0xc004): first=0xbe3ffb28
(0xc004): p0=0x13848
(0x10005): first=0xbe1ffb28
(0x10005): p0=0x13c50
(0x14006): first=0xbdfffb28
(0x14006): p0=0x14058
```

我们再查看 maps：

```
bde00000-bde01000 ---p bde00000 00:00 0
bde01000-be000000 rwxp bde01000 00:00 0
be000000-be001000 ---p be000000 00:00 0
```

```
be001000-be200000 rwxp be001000 00:00 0
be200000-be201000 ---p be200000 00:00 0
be201000-be400000 rwxp be201000 00:00 0
be400000-be401000 ---p be400000 00:00 0
be401000-be600000 rwxp be401000 00:00 0
be600000-be601000 ---p be600000 00:00 0
be601000-be800000 rwxp be601000 00:00 0
befeb000-bf000000 rwxp befeb000 00:00 0
```

我们可以看到，每创建一个线程，新创建的线程将调用 mmap 在虚拟内存顶部分配一个 2M 的内存虚拟内存，并且使用一个页面做隔离保护，以此做为线程的栈空间。这也就是说，主进程的栈空间为 8M，每个进程的栈内存空间为 2M，如果大于 2M 将会导致栈溢出。

同时我们还应该注意到一个现象，待我们去查看 maps 的时候，thread_proc 应该已经执行完了，这时线程已经退出，但我们在查看 maps 的时候发现，为每个线程分配的栈空间依然存在，并没有释放，为什么呢？

在线程退出后，进程需要调用 pthread_join 结束,否则栈内存就不释放

```
int main(int argc, char* argv[])
{
    int first = 0;
    int i= 0;
    void* ret = NULL;
    pthread_t tid[N] = {0};
    printf("first=%p\n", &first);
    for(i = 0; i < N; i++)
    {
        pthread_create(tid+i, NULL, thread_proc, NULL);
    }
    for(i=0;i<N;i++)
    {
        pthread_join(tid[i],&ret);
    }
    pause();
    return 0;
}
```

通过 maps，我们可以看到对应的线程栈空间内存释放。

你也可以通过 pthread 的函数来设置线程栈的大小。

我们知道，在 linux 中每个线程都有一个 pid，那么/proc/pid 的信息会如何呢？
他们的 maps，memmap,statm 都是完全一样的，这对于我们在统计整个 phone 内存使用时，要注意，以免重复计算。

2.5.1. 设置进程栈空间

我们可以使用 ulimit 命令，来查看和设置一个进程的栈空间的大小：

```
$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d)
unlimitedscheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 16365
max locked memory       (kbytes, -l) 32
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8196
cpu time               (seconds, -t) unlimited
max user processes              (-u) 16365
virtual memory          (kbytes, -v) unlimitedf
ile locks                       (-x) unlimited
```

从这里，我们可以看到，进程的栈空间为 8M。

下面，我们编写一个使用栈大于 8M 的测试程序，看看当栈空间大于 8M，会出现什么情况。

```
int main()
{
	int i = 1024 * 1024 * 8;
	char p[ i ];
	memset( p, 0, sizeof( char ) * i );
	while( i-- )
	{
		p[ i ] = i;
	}
	return 0;
}
```

程序运行时，出现段错误。

我们可以使用 ulimit 来重新设置栈空间的大小。

```
ulimit –s 16384
```

运行上面的程序后，就不会再有问题了。

2.5.2. 设置线程栈空间

通常，线程栈是从页边界开始的，任何指定的大小都被向上舍入到下一个页边界，不具备访问权限的页将被附加到栈的溢出端。

在 maps 中，下面两段内存空间，代表着一个线程的栈。
be000000-be001000 ---p be000000 00:00 0
be001000-be200000 rwxp be001000 00:00 0
其中 be001000-be200000 作为线程的栈空间，而 be000000-be001000 一个没有任何权限的只读页面，作为一个线程栈的保护页面，隔开了多个线程的栈，当线程的栈溢出时，会触发段错误，从而终止进程。

从上面得知，一个线程的栈内存空间为 2M。有些情况下我们需要调整线程栈空间的大小，比如说为了节省内存，我们通过将多个进程改为线程而合并到一个进程，这样 2M 的线程栈空间可能对由进程修改而成的栈来说，显得有些小，我们需要将线程的栈空间扩大；还有对于服务器来讲，其服务进程有可能创建数千个线程，那么每个线程 2M 的空间，又显得太奢侈了，我们需要将线程的栈空间减小。

我们可以使用 pthread_attr_setstacksize 来设置，在减小栈空间的时候，一定要慎重，防止过小而导致的栈溢出。

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
int ret;

size_t size = PTHREAD_STACK_MIN + 0x4000;
/* initialized with default attributes */
ret = pthread_attr_init(&tattr);

/* setting the size of the stack also */
ret = pthread_attr_setstacksize(&tattr, size);

/* only size specified in tattr*/
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

2.5.3. 减少线程的数量

一般情况下，一个进程所拥有的线程数量很小，大概 10 个以内，如果每个线程的栈使用 20K 的内存的话，那么总共消耗 200K 的内存，对系统的影响还是很小的。

可是对于某些网络服务器的进程，对于每个用户请求创建一个线程，为其服务。如果每个线程的工作时间很长，不能及时退出的话，会导致进程中同时并发大量的线程，这时其线程栈所占用的内存，就不可以轻视了。比如说，我们进程有 100 个线程，每个线程使用 20k 的内存做为栈，那么总共消耗 2M 的内存，这对于内存稀缺的嵌入式来讲，就是一个不小的消耗。

对于线程众多的进程，我们需要考虑使用异步通讯的的方式来管理所有的通讯的方式，替代以前线程+同步通讯的方式，来达到减少线程的目的。

这样的好处，一方面可以达到减少内存的目的，同时也可以减少线程数量，减轻 Linux 在内核做进程调度时的负担。

2.6. 共享内存

这里提到的共享内存，主要特指系统 V 共享内存，在这里并不是如何编程来实现共享内存，而是共享内存之后的一些故事？

问题一：内核是如何支持共享内存的？
进程间需要共享的数据被放在一个叫做 IPC 共享内存区域的地方，所有需要访问该共享区域的进程都要把该共享区域映射到本进程的地址空间中去。系统 V 共享内存通过 shmget 获得或创建一个 IPC 共享内存区域，并返回相应的标识符。内核在保证 shmget 获得或创建一个共享内存区，初始化该共享内存区相应的 shmid_kernel 结构注同时，还将在特殊文件系统 shm 中，创建并打开一个同名文件，并在内存中建立起该文件的相应 dentry 及 inode 结构，新打开的文件不属于任何一个进程（任何进程都可以访问该共享内存区）。所有这一切都是系统调用 shmget 完成的。

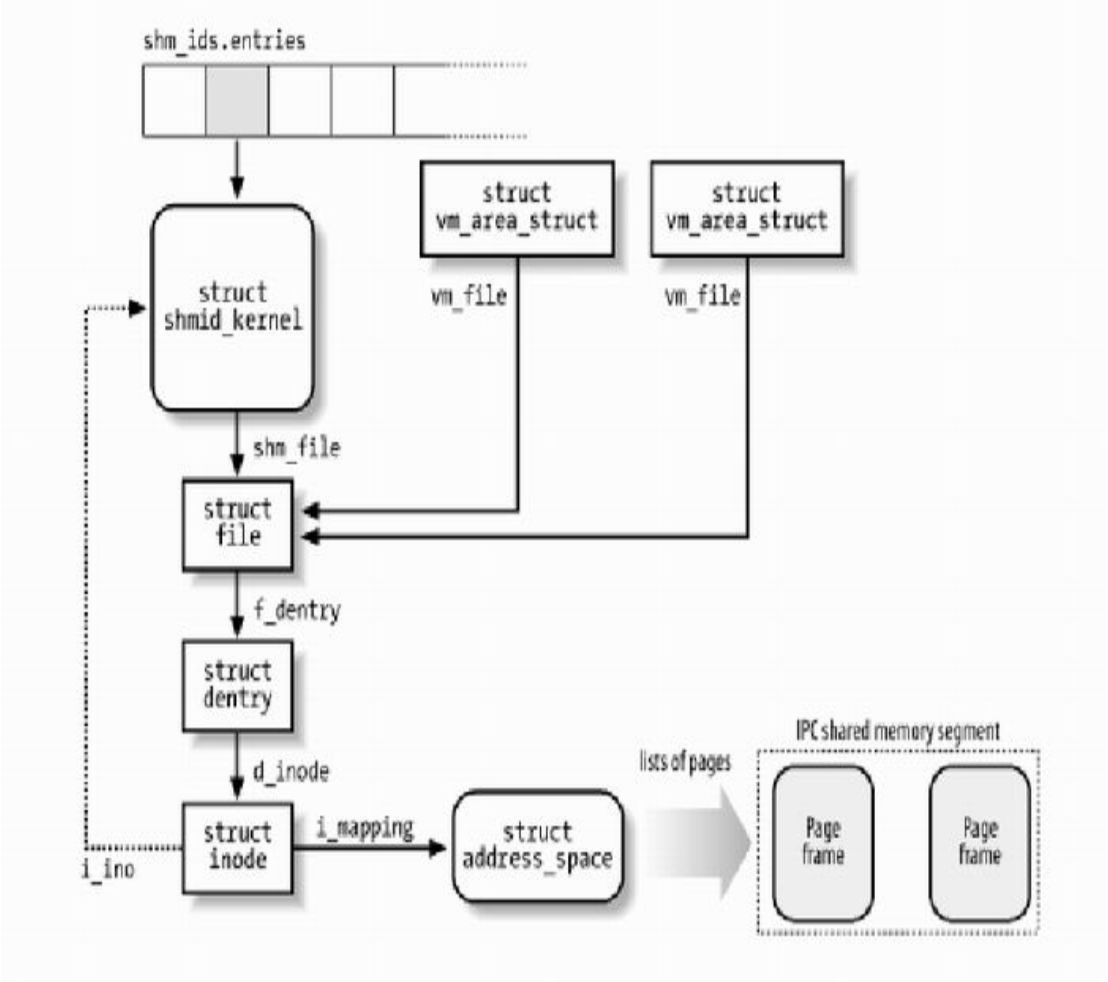
注：每一个共享内存区都有一个控制结构 struct shmid_kernel，shmid_kernel 是共享内存区域中非常重要的一个数据结构，它是存储管理和文件系统结合起来的桥梁，定义如下：

```
struct shmid_kernel /* private to the kernel */
{
    struct kern_ipc_perm    shm_perm;
    struct file *           shm_file;
    int                     id;
    unsigned long           shm_nattch;
    unsigned long           shm_segsz;
    time_t                  shm_atim;
```

```
    time_t          shm_dtim;
    time_t          shm_ctim;
    pid_t           shm_cprid;
    pid_t           shm_lprid;
};
```

该结构中最重要的一个域应该是 shm_file，它存储了将被映射文件的地址。每个共享内存区对象都对应特殊文件系统 shm 中的一个文件，一般情况下，特殊文件系统 shm 中的文件是不能用 read()、write()等方法访问的，当采取共享内存的方式把其中的文件映射到进程地址空间后，可直接采用访问内存的方式对其访问。

这里我们采用[1]中的图表给出与系统 V 共享内存相关数据结构：



正如消息队列和信号灯一样，内核通过数据结构 struct ipc_ids shm_ids 维护系统中的所有共享内存区域。上图中的 shm_ids.entries 变量指向一个 ipc_id 结构数组，而每个 ipc_id 结构数组中有个指向 kern_ipc_perm 结构的指针。到这里读者应该很熟悉了，对于系统 V 共享内存区来说，kern_ipc_perm 的宿主是 shmid_kernel 结构，shmid_kernel 是用来描述一个共享内存区域的，这样内核就能够控制系统中所有的共享区域。同时，在 shmid_kernel 结构的 file 类型指针 shm_file 指向文件系统 shm 中相应的文件，这样，共享内存区域就与 shm 文件系统中的文件对应起来。

在创建了一个共享内存区域后，还要将它映射到进程地址空间，系统调用 shmat()完成此项

功能。由于在调用 shmget()时，已经创建了文件系统 shm 中的一个同名文件与共享内存区域相对应，因此，调用 shmat()的过程相当于映射文件系统 shm 中的同名文件过程，原理与 mmap()大同小异。

问题二：共享内存在不同的进程中，是否地址相同？
这个不一定。

问题三：在 proc 文件系统中，共享内存在不同的进程中，物理内存数量统计是一样的？
不是的，在不同的进程中先使用虚拟地址映射到对应 shm 文件系统中，当对其访问时，触发页故障，但这时不是分配新的物理页面，然后读入文件内容到物理页面中，它是直接将这个虚拟页面映射到 shm 文件系统的物理页面上，从而达到多个进程物理内存共享的目的。因为各个进程对共享内存的访问不同，所以共享内存所占用的物理内存数量在进程所对应 proc 目录下显示的信息是不同的。

2.7. 内存调试

在做内存优化时，我发现人们最常问的是：
为什么我的程序使用了这么多的内存？
怎样检测内存泄漏？

我实在不是很明白，编码技术发展了这么多年，为什么还是没有有效的工具来帮我们解决这两个问题。

下面，我来试着一步一步解决这个问题。

2.7.1. mtrace

首先我来一下 glibc，针对内存内存泄漏给出的解决方，在 glibc 库中实现了一个钩子函数 mtrace。

1、 加入头文件<mcheck.h>
2、 在需要内存泄漏检查的代码的开始调用 void mtrace()，在需要内存泄漏检查代码的结束调用 void muntrace()。
一般情况下不要调用 muntrace，而让程序自然结束，因为可能有些内存释放代码要到 muntrace 之后运行。
3、 用 debug 模式编译检查代码（-g 或-ggdb）
4、 在运行程序钱，先设置环境变量 MALLOC_TRACE 为一文件名，这一文件将存有内存分配信息。
5、 运行程序，内存分配的 log 将输出到 MALLOC_TRACE 所指向的文件中。

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

```
int main(void)
{
	mtrace();

	char *p=malloc(10);
	return 0;
}
```

输出结果为

```
= Start
@ ./t2:(mtrace+0xdb)[0x8048397] + 0x8b333c8 0xa
```

我使用这种方法，在嵌入式设备上做了一些测试，发现两个问题：
1、 程序运行很慢；
2、 Log 文件大小上升的很快。

对于程序运行慢的问题，主要有两个因素：
1、 程序需要把 log 写到 flash 文件中，读写 flash 是比较慢的。
一般我们都通过 telnet 的方式来调试设备，那么我们可以想办法，让 mtrace 的 log 输出在终端窗口，由终端软件记录下来。
将 MALLOC_TRACE 指向/dev/stdout。
export MALLOC_TRACE=/dev/stdout
在有些嵌入式设备中，没有 stdout，我们可以将 MALLOC_TRACE=/proc/self/fd/1 来实现。

```
export MALLOC_TRACE=/proc/self/fd/1
```

这样 mtrace 的 log 就不会记录在 flash 中。

2、 mtrace 中试图根据调用 malloc 的代码指针，解析出对应的函数，这会导致进程的速度变得很慢。
我们可以考虑在钩子函数中，只是打印出运行时的代码地址，而不对齐进行解析，在拿到 log 之后，我们再来考虑利用 PC 对其事后地址解析。

可问题是那我们就需要修改 mtrace 函数。

幸好 libc 中为我们提供了 malloc 和 free 的钩子函数，这样我们就可以自由定义了。

2.7.2. malloc and free 钩子函数

在 glibc 中，提供了 malloc、free、realloc、memalign 的钩子函数。
你可以按照钩子函数的原型，定义自己的函数，并在 glibc 中设置相应的钩子函数，这样 glibc 在处理函数时，会调用你的钩子函数，从而获得相应的信息。

详细，你可以参见：http://www.gnu.org/software/libtool/manual/libc/Hooks-for-Malloc.html

范例：

```
#include <malloc.h>
#include <execinfo.h>

static void* (* old_malloc_hook) (size_t,const void *);
static void (* old_free_hook)(void *,const void *);
static void my_init_hook(void);
static void* my_malloc_hook(size_t,const void*);
static void    my_free_hook(void*,const void *);


static void my_init_hook(void)
{
      old_malloc_hook = __malloc_hook;
      old_free_hook = __free_hook;
      __malloc_hook = my_malloc_hook;
      __free_hook = my_free_hook;
}

static void* my_malloc_hook(size_t size,const void *caller)
{
      void *result;
      __malloc_hook = old_malloc_hook;
      result = malloc(size);
      old_malloc_hook = __malloc_hook;
      printf("@@@ %p + %p 0x%x\n",caller,result,(unsigned long int)size);
      __malloc_hook = my_malloc_hook;

      return result;
}

static void my_free_hook(void *ptr,const void *caller)
{
      __free_hook = old_free_hook;
      free(ptr);
      old_free_hook = __free_hook;
      printf("@@@ %p - %p\n",caller,ptr);
      __free_hook = my_free_hook;
}


char *p[10];
```

```
int func1()
{
	int i=0;
	for(i=0;i<10;i++)
	{
		p[i]=(char *)malloc(i);
	}
	return 0;
}

int func2()
{
	int i=0;
	for(i=0;i<9;i++)
	{
		free(p[i]);
	}
	return 0;
}

int main()
{
	my_init_hook();

	func1();
	func2();
	pause();
	return 0;
}
```

你可以在你的钩子函数中，打印出进程的栈，这样你对每块内存的分配将十分了解。

```
# ./hello
@@@ 0x8694 + 0x11050 0x0
@@@ 0x8694 + 0x11060 0x1
@@@ 0x8694 + 0x11070 0x2
@@@ 0x8694 + 0x11080 0x3
@@@ 0x8694 + 0x11090 0x4
@@@ 0x8694 + 0x110a0 0x5
@@@ 0x8694 + 0x110b0 0x6
@@@ 0x8694 + 0x110c0 0x7
@@@ 0x8694 + 0x110d0 0x8
@@@ 0x8694 + 0x110e0 0x9
@@@ 0x86f4 - 0x11050
@@@ 0x86f4 - 0x11060
@@@ 0x86f4 - 0x11070
```

```
@@@ 0x86f4 - 0x11080
@@@ 0x86f4 - 0x11090
@@@ 0x86f4 - 0x110a0
@@@ 0x86f4 - 0x110b0
@@@ 0x86f4 - 0x110c0
@@@ 0x86f4 - 0x110d0
```

我们将这些打印出来的 log，保存到文件中 1.log。
下一步，我们要想办法，把那些申请了并释放了的内存过滤掉，找出没有释放的内存。

我们可以写一个 perl 脚本对齐进行分析：
pmleak

```
#! /usr/bin/perl

my $log = shift (@ARGV);
open flog ,"<$log" or die "cannot open $log:$!";
while(<flog>)
{
    if(/^@@@/)
    {
        @items = split/\s+/;
        if($items[2] eq '+')
        {
            $size=hex(substr($items[4],2));
            $memory{$items[3]}=$size;
        }
        if($items[2] eq '-')
        {
            delete $memory{$items[3]};
        }
    }
}
foreach $key(sort keys %memory)
{
    print "$key    $memory{$key}\n";
}
```

我们可以在 Linux 机器上运行 perl 脚本。

```
> ./pt1 1.log
0x110e0    9
```

perl 脚本对于分析文本文件很方便，不是吗，建议每个人都学习一下。

现在我们能够检查到哪块内存没有释放，但这还远远不够，我们需要知道这块内存是哪个在

哪个函数分配的，这很重要。
这时我们就需要解析在 malloc 的钩子函数中的 caller 地址。

第一步，修改上面的 perl 脚本，使其在分期内存泄漏时，也能打印出 caller 的地址。
#! /usr/bin/perl

```
my $log = shift (@ARGV);
open flog ,"<$log" or die "cannot open $log:$!";
while(<flog>)
{
    if(/^@@@/)
    {
        @items = split/\s+/;
        if($items[2] eq '+')
        {
            $size=hex(substr($items[4],2));
            $memory{$items[3]}=$size;
            $caller{$items[3]}=$items[1];
        }
        if($items[2] eq '-')
        {
            delete $memory{$items[3]};
            delete $caller{$items[3]};
        }
    }
}
foreach $key(sort keys %memory)
{
    print "$key    $memory{$key}    $caller{$key}\n";
}
```

我们再来看一下运行结果：

```
> ./pt1 1.log
0x110e0    9    0x8694
```

现在我们知道 caller 的地址是 0x8694，但这个地址对程序员来讲还是很困难。

第二步，我们将 caller 的地址，转换成函数。
这里我们需要利用程序的符号表。

```
>nm -n hello
……..
0000848c t call_gmon_start
000084b8 t __do_global_dtors_aux
000084d4 t frame_dummy
000084fc t my_init_hook
```

```
0000855c t my_malloc_hook
000085e4 t my_free_hook
00008658 T func1
000086b8 T func2
00008714 T main
00008738 T __libc_csu_init
00008798 T __libc_csu_fini
000087ec T _fini
……
```

将其保存到文件中 sym.log。

注意：在获取符号表时，一定要选用-n 选项，使符号按顺序打印出来，这对于我们下面解析地址非常重要。

现在我们再来看一下 0x8694 这个地址，它应该正好在：

```
00008658 T func1
000086b8 T func2
```

之间。

func1 的起始地址是 00008658，func2 的起始地址是 000086b5，因为分配内存的这个地址应该在函数中间的某个位置，它距离起始地址应该有个偏移，所以我们可以断定 0x8694 这个地址位于 func1 函数内。

下面我们再改写 perl 脚本对齐进行解析。

```
#! /usr/bin/perl

sub getsym
{
    my($addr)=hex($_[0]);
    my($sym)="";
    if(!open fsym,"<$symfile")
    {
        print "can't open libray $symfile\n";
        return -1;
    }

    while(<fsym>)
    {
        chomp($_);
        $term[0]=substr($_,0,8);
        $term[1]=substr($_,9,1);
        $term[2]=substr($_,11);
        if(($term[1] eq "t")||($term[1] eq "T"))
        {
            if($addr < hex($term[0])
```

```
                {
                        $sym=$term[2];
                }else
                {
                        last;
                }
        }
    }
    close fsym;
    return $sym;
}

my $log = shift (@ARGV);
my $symfile = shift (@ARGV);

open flog ,"<$log" or die "cannot open $log:$!";
while(<flog>)
{
    if(/^@@@/)
    {
        @items = split/\s+/;
        if($items[2] eq '+')
        {
                $size=hex(substr($items[4],2));
                $memory{$items[3]}=$size;
                $caller{$items[3]}=$items[1];
        }
        if($items[2] eq '-')
        {
                delete $memory{$items[3]};
                delete $caller{$items[3]};
        }
    }
}

foreach $key(sort keys %memory)
{
    $mysym=&getsym($caller{$key});
    print "$key    $memory{$key} $caller{$key} $mysym\n";
}
```

这里，主要增加了一个函数 getsym，其基本算法是根据已知地址，查找到小于它并最接近的一个函数符号。

我们再来看一下结果：
> ./pt1 ./1.log ./sym
0x110e0 9 0x8694 func1
我们已经知道了哪个函数分配了这块内存。

我们需要在编译的时候，加上调试信息。
>gcc -o hello -g hello.c

再使用 addr2line，来将地址指针转化称为具体的代码行。
>addr2line -e hello 0x8694
/home/jkvp74/bookcode/hello.c:48

从结果上，我们可以得知是 hello.c 的 48 行，分配了该块内存。
其正好对应着：
p[i]=(char *)malloc(i);

现在我们能找到内存泄漏，也能找到其对应的具体的代码行，但这还不够。

我们来看下面的程序：

```
char *p[10];

void func1()
{
    int i=0;
    for(i=0;i<10;i++)
    {
        p[i]=(char *)malloc(i);
    }
}

void func2()
{
    int i=0;
    for(i=0;i<10;i++)
    {
        free(p[i]);
    }
}

void func3()
{
    func1();
```

```
}

int main()
{
    my_init_hook();

    func1();
    func2();
    func3();
    pause();
    return 0;
}
```

我们在程序前面加上钩子函数，然后让我们来看一下结果。

编译程序

```
>gcc –o hello –g hello.c
```

得到符号文件

```
>nm –n hello > sym
```

运行程序

```
# ./hello
@@@ 0x8694 + 0x11050 0x0
@@@ 0x8694 + 0x11060 0x1
@@@ 0x8694 + 0x11070 0x2
@@@ 0x8694 + 0x11080 0x3
@@@ 0x8694 + 0x11090 0x4
@@@ 0x8694 + 0x110a0 0x5
@@@ 0x8694 + 0x110b0 0x6
@@@ 0x8694 + 0x110c0 0x7
@@@ 0x8694 + 0x110d0 0x8
@@@ 0x8694 + 0x110e0 0x9
@@@ 0x86f0 - 0x11050
@@@ 0x86f0 - 0x11060
@@@ 0x86f0 - 0x11070
@@@ 0x86f0 - 0x11080
@@@ 0x86f0 - 0x11090
@@@ 0x86f0 - 0x110a0
@@@ 0x86f0 - 0x110b0
@@@ 0x86f0 - 0x110c0
@@@ 0x86f0 - 0x110d0
@@@ 0x86f0 - 0x110e0
@@@ 0x8694 + 0x110e0 0x0
```

```
@@@ 0x8694 + 0x110d0 0x1
@@@ 0x8694 + 0x110c0 0x2
@@@ 0x8694 + 0x110b0 0x3
@@@ 0x8694 + 0x110a0 0x4
@@@ 0x8694 + 0x11090 0x5
@@@ 0x8694 + 0x11080 0x6
@@@ 0x8694 + 0x11070 0x7
@@@ 0x8694 + 0x11060 0x8
@@@ 0x8694 + 0x11050 0x9
```

我们将其保存到文件，memory.log，使用我们上面的工具进行分析。

```
>./pt1    ./memory.log ./sym
0x11050    9 0x8694 func1
0x11060    8 0x8694 func1
0x11070    7 0x8694 func1
0x11080    6 0x8694 func1
0x11090    5 0x8694 func1
0x110a0    4 0x8694 func1
0x110b0    3 0x8694 func1
0x110c0    2 0x8694 func1
0x110d0    1 0x8694 func1
0x110e0    0 0x8694 func1
```

现在我们虽然定位到了 func1 产生了内存泄漏，但我们需要知道程序是如何调到 func1。
我们需要修改 malloc 的钩子函数，调用 backtrace，将程序对应的栈打印出来。

我们修改一下 my_malloc_hook 函数，增加打印栈的内容。

```
static void* my_malloc_hook(size_t size,const void *caller)
{
    void *array[10];
    size_t size1;
    size1 = backtrace (array, 5);
    void *result;
    __malloc_hook = old_malloc_hook;
    result = malloc(size);
    old_malloc_hook = __malloc_hook;
    __malloc_hook = my_malloc_hook;
    switch(size1)
    {
    case 1:
        printf("@@@ %p + %p 0x%x %p\n",caller,result,(unsigned long
int)size,array[0]);
        break;
```

```
    case 2:
        printf("@@@ %p + %p 0x%x %p %p\n",caller,result,(unsigned long
int)size,array[0], array[1]);
        break;
    case 3:
        printf("@@@ %p + %p 0x%x %p %p %p\n",caller,result,(unsigned long
int)size,array[0],array[1],array[2]);
        break;
    case 4:
        printf("@@@ %p + %p 0x%x %p %p %p %p\n",caller,result,(unsigned long
int)size,array[0],array[1],array[2],array[3]);
        break;
    case 5:
        printf("@@@ %p + %p 0x%x %p %p %p %p %p\n",caller,result,(unsigned long
int)size,array[0],array[1],array[2],array[3],array[4]);
        break;
     }
     return result;
}
```

修改完成后，我们看看打印出来的结果。

#./hello

```
@@@ 0x87e8 + 0x11050 0x0 0x85b4 0x41087a40 0x8888 0x4103b8ac
@@@ 0x87e8 + 0x11060 0x1 0x85b4 0x41087a40 0x8888 0x4103b8ac
@@@ 0x87e8 + 0x11070 0x2 0x85b4 0x41087a40 0x8888 0x4103b8ac
@@@ 0x87e8 + 0x11080 0x3 0x85b4 0x41087a40 0x8888 0x4103b8ac
@@@ 0x87e8 + 0x11090 0x4 0x85b4 0x41087a40 0x8888 0x4103b8ac
@@@ 0x87e8 + 0x110a0 0x5 0x85b4 0x41087a40 0x8888 0x4103b8ac
@@@ 0x87e8 + 0x110b0 0x6 0x85b4 0x41087a40 0x8888 0x4103b8ac
@@@ 0x87e8 + 0x110c0 0x7 0x85b4 0x41087a40 0x8888 0x4103b8ac
@@@ 0x87e8 + 0x110d0 0x8 0x85b4 0x41087a40 0x8888 0x4103b8ac
@@@ 0x87e8 + 0x110e0 0x9 0x85b4 0x41087a40 0x8888 0x4103b8ac
@@@ 0x8844 - 0x11050
@@@ 0x8844 - 0x11060
@@@ 0x8844 - 0x11070
@@@ 0x8844 - 0x11080
@@@ 0x8844 - 0x11090
@@@ 0x8844 - 0x110a0
@@@ 0x8844 - 0x110b0
@@@ 0x8844 - 0x110c0
@@@ 0x8844 - 0x110d0
@@@ 0x8844 - 0x110e0
@@@ 0x87e8 + 0x110e0 0x0 0x85b4 0x41087a40 0x8870 0x8890 0x4103b8ac
@@@ 0x87e8 + 0x110d0 0x1 0x85b4 0x41087a40 0x8870 0x8890 0x4103b8ac
```

```
@@@ 0x87e8 + 0x110c0 0x2 0x85b4 0x41087a40 0x8870 0x8890 0x4103b8ac
@@@ 0x87e8 + 0x110b0 0x3 0x85b4 0x41087a40 0x8870 0x8890 0x4103b8ac
@@@ 0x87e8 + 0x110a0 0x4 0x85b4 0x41087a40 0x8870 0x8890 0x4103b8ac
@@@ 0x87e8 + 0x11090 0x5 0x85b4 0x41087a40 0x8870 0x8890 0x4103b8ac
@@@ 0x87e8 + 0x11080 0x6 0x85b4 0x41087a40 0x8870 0x8890 0x4103b8ac
@@@ 0x87e8 + 0x11070 0x7 0x85b4 0x41087a40 0x8870 0x8890 0x4103b8ac
@@@ 0x87e8 + 0x11060 0x8 0x85b4 0x41087a40 0x8870 0x8890 0x4103b8ac
@@@ 0x87e8 + 0x11050 0x9 0x85b4 0x41087a40 0x8870 0x8890 0x4103b8ac
```

我们可以通过编写 perl 脚本，对其栈进行解析。

```
#! /usr/bin/perl

sub by_linedes      {$mem_line{$a}<=>$mem_line{$b}}

sub getsym
{
      my($addr)=hex($_[0]);
      my($symfile)=$_[1];

      my($sym)="";
      if(!open fsym,"<$symfile")
      {
            print "can't open libray $symfile\n";
            return -1;
      }

      while(<fsym>)
      {
            chomp($_);
            $term[0]=substr($_,0,8);
            $term[1]=substr($_,9,1);
            $term[2]=substr($_,11);
            if(($term[1] eq "t")||($term[1] eq "T"))
            {
                  if($addr < hex($term[0]))
                  {
                        last;
                  }else
                  {
                        $sym=$term[2];
                  }
            }
      }
      close fsym;
```

```
        return $sym;
}

my $log = shift (@ARGV);
my $symfile = shift (@ARGV);

my($mline)=0;
open flog ,"<$log" or die "cannot open $log:$!";
while(<flog>)
{
        if(/^@@@/)
        {
                chomp($_);
                undef @items;
                @items = split/\s+/;
                $i=0;
                foreach(@items)
                {
                        if($i==3)
                        {
                                $point=hex(substr($items[3],2));

                                if($items[2] eq '+')
                                {
                                        $size=hex(substr($items[4],2));
                                        $memory{$items[3]}=$size;
                                        $caller{$items[3]}=$items[1];
                                        if(! exists $stack{$items[1]})
                                        {
                                                $stack{$items[1]}="";
                                        }
                                        $mem_line{$items[3]}=$mline;
                                }
                                if($items[2] eq '-')
                                {
                                        delete $memory{$items[3]};
                                        delete $caller{$items[3]};
                                        delete $mem_line{$items[3]};
                                }
                        }
                        if($i>4)
                        {
                                if(! exists $stack{$_})
                                {
```

```
                                $stack{$_}="";
                            }
                        }
                        $i=$i+1;
                    }
                }
                $mline=$mline+1;
}
close flog;


foreach $key(sort keys %memory)
{
        $mysym=&getsym($caller{$key},$symfile);
        print "$key    $memory{$key} $caller{$key} $mysym\n";
}

open flog ,"<$log" or die "cannot open $log:$!";
my($mline)=0;
foreach $key(sort by_linedes keys%mem_line)
{
        $line_no=$mem_line{$key};
        while(<flog>)
        {
                if(/^@@@/)
                {
                        if($mline==$line_no)
                        {
                                chomp($_);
                                undef @items;
                                @items = split/\s+/;
                                $i=0;
                                print "@items\n";
                                foreach(@items)
                                {
                                        if($i>4)
                                        {
                                                $myaddr=$_;
                                                $myfunc=&getsym($myaddr,$symfile);
                                                print " $myaddr\t$myfunc\n";
                                        }
                                        $i=$i+1;
                                }
                                print "\n";
```

```
                        $mline=$mline+1;
                        last;
                    }
                }else
                {
                        if($_ ne "\n")
                        {
                            print "$_";
                        }
                }
                $mline=$mline+1;
        }
}
close flog;

> ./pt1 ./1.log ./sym
0x11050    9 0x87e8 func1
0x11060    8 0x87e8 func1
0x11070    7 0x87e8 func1
0x11080    6 0x87e8 func1
0x11090    5 0x87e8 func1
0x110a0    4 0x87e8 func1
0x110b0    3 0x87e8 func1
0x110c0    2 0x87e8 func1
0x110d0    1 0x87e8 func1
0x110e0    0 0x87e8 func1
@@@ 0x87e8 + 0x110e0 0x0 0x85b4 0x41087a40 0x8870 0x8890 0x4103b8ac
 0x85b4 my_malloc_hook
 0x41087a40          __do_global_dtors_aux_fini_array_entry
 0x8870 func3
 0x8890 main
 0x4103b8ac          __do_global_dtors_aux_fini_array_entry

@@@ 0x87e8 + 0x110d0 0x1 0x85b4 0x41087a40 0x8870 0x8890 0x4103b8ac
 0x85b4 my_malloc_hook
 0x41087a40          __do_global_dtors_aux_fini_array_entry
 0x8870 func3
 0x8890 main
 0x4103b8ac          __do_global_dtors_aux_fini_array_entry

@@@ 0x87e8 + 0x110c0 0x2 0x85b4 0x41087a40 0x8870 0x8890 0x4103b8ac
 0x85b4 my_malloc_hook
 0x41087a40          __do_global_dtors_aux_fini_array_entry
 0x8870 func3
```

```
 0x8890 main
 0x4103b8ac        __do_global_dtors_aux_fini_array_entry

@@@ 0x87e8 + 0x110b0 0x3 0x85b4 0x41087a40 0x8870 0x8890 0x4103b8ac
 0x85b4 my_malloc_hook
 0x41087a40        __do_global_dtors_aux_fini_array_entry
 0x8870 func3
 0x8890 main
 0x4103b8ac        __do_global_dtors_aux_fini_array_entry

@@@ 0x87e8 + 0x110a0 0x4 0x85b4 0x41087a40 0x8870 0x8890 0x4103b8ac
 0x85b4 my_malloc_hook
 0x41087a40        __do_global_dtors_aux_fini_array_entry
 0x8870 func3
 0x8890 main
 0x4103b8ac        __do_global_dtors_aux_fini_array_entry

@@@ 0x87e8 + 0x11090 0x5 0x85b4 0x41087a40 0x8870 0x8890 0x4103b8ac
 0x85b4 my_malloc_hook
 0x41087a40        __do_global_dtors_aux_fini_array_entry
 0x8870 func3
 0x8890 main
 0x4103b8ac        __do_global_dtors_aux_fini_array_entry

@@@ 0x87e8 + 0x11080 0x6 0x85b4 0x41087a40 0x8870 0x8890 0x4103b8ac
 0x85b4 my_malloc_hook
 0x41087a40        __do_global_dtors_aux_fini_array_entry
 0x8870 func3
 0x8890 main
 0x4103b8ac        __do_global_dtors_aux_fini_array_entry

@@@ 0x87e8 + 0x11070 0x7 0x85b4 0x41087a40 0x8870 0x8890 0x4103b8ac
 0x85b4 my_malloc_hook
 0x41087a40        __do_global_dtors_aux_fini_array_entry
 0x8870 func3
 0x8890 main
 0x4103b8ac        __do_global_dtors_aux_fini_array_entry

@@@ 0x87e8 + 0x11060 0x8 0x85b4 0x41087a40 0x8870 0x8890 0x4103b8ac
 0x85b4 my_malloc_hook
 0x41087a40        __do_global_dtors_aux_fini_array_entry
 0x8870 func3
 0x8890 main
 0x4103b8ac        __do_global_dtors_aux_fini_array_entry
```

```
@@@ 0x87e8 + 0x11050 0x9 0x85b4 0x41087a40 0x8870 0x8890 0x4103b8ac
 0x85b4 my_malloc_hook
 0x41087a40        __do_global_dtors_aux_fini_array_entry
 0x8870 func3
 0x8890 main
 0x4103b8ac        __do_global_dtors_aux_fini_array_entry
```

注意象 0x41087a40 __do_global_dtors_aux_fini_array_entry，这样的解析是错误的。主要是因为该地址属于动态库的地址，根据加载地址的不同而有所变化，而动态库生成的符号表，则是根据虚拟地址 0x00000000 来形成的。

在某些编译选项，会导致我们上面打印栈的方法失效。
例如：

```
>gcc -o hello -O2 hello.c
```

很常用，我们使用编译选项 O2 对程序进行优化；

我们再来看看其在嵌入设备中执行的结果。

```
# ./hello
@@@ 0x872c + 0x11050 0x0 0x8554
@@@ 0x872c + 0x11060 0x1 0x8554
@@@ 0x872c + 0x11070 0x2 0x8554
@@@ 0x872c + 0x11080 0x3 0x8554
@@@ 0x872c + 0x11090 0x4 0x8554
@@@ 0x872c + 0x110a0 0x5 0x8554
@@@ 0x872c + 0x110b0 0x6 0x8554
@@@ 0x872c + 0x110c0 0x7 0x8554
@@@ 0x872c + 0x110d0 0x8 0x8554
@@@ 0x872c + 0x110e0 0x9 0x8554
@@@ 0x8764 - 0x11050
@@@ 0x8764 - 0x11060
@@@ 0x8764 - 0x11070
@@@ 0x8764 - 0x11080
@@@ 0x8764 - 0x11090
@@@ 0x8764 - 0x110a0
@@@ 0x8764 - 0x110b0
@@@ 0x8764 - 0x110c0
@@@ 0x8764 - 0x110d0
@@@ 0x8764 - 0x110e0
@@@ 0x872c + 0x110e0 0x0 0x8554
@@@ 0x872c + 0x110d0 0x1 0x8554
@@@ 0x872c + 0x110c0 0x2 0x8554
@@@ 0x872c + 0x110b0 0x3 0x8554
@@@ 0x872c + 0x110a0 0x4 0x8554
```

```
@@@ 0x872c + 0x11090 0x5 0x8554
@@@ 0x872c + 0x11080 0x6 0x8554
@@@ 0x872c + 0x11070 0x7 0x8554
@@@ 0x872c + 0x11060 0x8 0x8554
@@@ 0x872c + 0x11050 0x9 0x8554
```

可以看到，栈的结果只打印出来一列，函数调用栈根本没有打印出来。这个问题，我们将在下一章栈的回溯讲到。

我们再来将测试程序改动一下：

a.c

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

char *p[10];
void func3()
{
    int i=0;
    for(i=0;i<10;i++)
    {
        p[i]=(char *)malloc(i);
    }
}

void func2()
{
    int i=0;
    for(i=0;i<9;i++)
    {
        free(p[i]);
    }
}


void func1()
{
    func3();
}
```

编译成动态链接库：

```
>gcc –shared –fPIC –o liba.so a.c
```

hello.c

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

```
#include <malloc.h>
#include <execinfo.h>

static void* (* old_malloc_hook) (size_t,const void *);
static void (* old_free_hook)(void *,const void *);
static void my_init_hook(void);
static void* my_malloc_hook(size_t,const void*);
static void    my_free_hook(void*,const void *);

static void my_init_hook(void)
{
      old_malloc_hook = __malloc_hook;
      old_free_hook = __free_hook;
      __malloc_hook = my_malloc_hook;
      __free_hook = my_free_hook;
}

static void* my_malloc_hook(size_t size,const void *caller)
{
      void *array[10];
      size_t size1;
      size1 = backtrace (array, 5);
      void *result;
      __malloc_hook = old_malloc_hook;
      result = malloc(size);
      old_malloc_hook = __malloc_hook;
      __malloc_hook = my_malloc_hook;
      switch(size1)
      {
      case 1:
            printf("@@@ %p + %p 0x%x %p\n",caller,result,(unsigned long int)size,array[0]);
            break;
      case 2:
            printf("@@@ %p + %p 0x%x %p %p\n",caller,result,(unsigned long int)size,array[0],
array[1]);
            break;
      case 3:
            printf("@@@ %p + %p 0x%x %p %p %p\n",caller,result,(unsigned long
int)size,array[0],array[1],array[2]);
            break;
      case 4:
            printf("@@@ %p + %p 0x%x %p %p %p %p\n",caller,result,(unsigned long
int)size,array[0],array[1],array[2],array[3]);
            break;
```

```
        case 5:
                printf("@@@ %p + %p 0x%x %p %p %p %p %p\n",caller,result,(unsigned long int)size,array[0],array[1],array[2],array[3],array[4]);
                break;
        case 6:
                printf("@@@ %p + %p 0x%x %p %p %p %p %p\n",caller,result,(unsigned long int)size,array[0],array[1],array[2],array[3],array[4],array[5]);
                break;

        }
        return result;
}

static void my_free_hook(void *ptr,const void *caller)
{
        __free_hook = old_free_hook;
        free(ptr);
        old_free_hook = __free_hook;
        printf("@@@ %p - %p\n",caller,ptr);
        __free_hook = my_free_hook;
}
void func1();
void func2();

int main()
{
        my_init_hook();

        func1();
        func2();
        pause();
        return 0;
}
```

>gcc –L./ -la –o hello hello.c

我们来看看运行结果：

```
# ./hello
@@@ 0x40001658 + 0x11050 0x0 0x87e0 0x41087a40 0x400016f4 0x8a38 0x4103b8ac
@@@ 0x40001658 + 0x11060 0x1 0x87e0 0x41087a40 0x400016f4 0x8a38 0x4103b8ac
@@@ 0x40001658 + 0x11070 0x2 0x87e0 0x41087a40 0x400016f4 0x8a38 0x4103b8ac
@@@ 0x40001658 + 0x11080 0x3 0x87e0 0x41087a40 0x400016f4 0x8a38 0x4103b8ac
@@@ 0x40001658 + 0x11090 0x4 0x87e0 0x41087a40 0x400016f4 0x8a38 0x4103b8ac
@@@ 0x40001658 + 0x110a0 0x5 0x87e0 0x41087a40 0x400016f4 0x8a38 0x4103b8ac
```

@@@ 0x40001658 + 0x110b0 0x6 0x87e0 0x41087a40 0x400016f4 0x8a38 0x4103b8ac
@@@ 0x40001658 + 0x110c0 0x7 0x87e0 0x41087a40 0x400016f4 0x8a38 0x4103b8ac
@@@ 0x40001658 + 0x110d0 0x8 0x87e0 0x41087a40 0x400016f4 0x8a38 0x4103b8ac
@@@ 0x40001658 + 0x110e0 0x9 0x87e0 0x41087a40 0x400016f4 0x8a38 0x4103b8ac
@@@ 0x400016c4 - 0x11050
@@@ 0x400016c4 - 0x11060
@@@ 0x400016c4 - 0x11070
@@@ 0x400016c4 - 0x11080
@@@ 0x400016c4 - 0x11090
@@@ 0x400016c4 - 0x110a0
@@@ 0x400016c4 - 0x110b0
@@@ 0x400016c4 - 0x110c0
@@@ 0x400016c4 - 0x110d0

下面我们先来说说 caller 的地址 0x40001658，怎么解析?
从我们的代码来讲，这个地址应该对应着 liba.so 的 func3。那么我们现在就来看看 liba.so 的符号表。

…….
000005c4 t frame_dummy
00000610 T func3
0000067c T func2
000006e4 T func1
000006f8 T _fini
00000708 r __FRAME_END__
00000708 A __exidx_end
00000708 A __exidx_start
……
0x40001658 和 00000610 相距甚远。这主要是因为，进程在加载动态库时，其基地址是随机的，所以导致具体函数地址的变化。
我们的方法是，要找到这个地址与加载地址的偏移，根据此再在函数表中查找。
我们查看当前进程的 maps 信息。
00008000-00009000 r-xp 00000000 1f:12 318 /mnt/msc_int0/hello
00010000-00011000 rw-p 00000000 1f:12 318 /mnt/msc_int0/hello
00011000-00032000 rwxp 00011000 00:00 0
40000000-40001000 rw-p 40000000 00:00 0
40001000-40002000 r-xp 00000000 1f:12 319 /mnt/msc_int0/liba.so
40002000-40009000 ---p 00001000 1f:12 319 /mnt/msc_int0/liba.so
40009000-4000a000 rw-p 00000000 1f:12 319 /mnt/msc_int0/liba.so
4000a000-4000b000 rw-p 4000a000 00:00 0
我们可以看到 liba.so 的代码段开始地址为 40001000。
计算 0x40001658 的偏移量：0x40001658-0x40001000=0x658。
我们再将 0x658 查找 liba.so 的符号表，可以很清楚的看到它是属于 func3 函数。

因此，如果我们想解析进程运行时动态库的地址，那么我们就需要其 maps 信息，计算偏移

量，再去查找符号表。

上面的数据只是阐明了当前哪些内存申请了，但没有释放。

有了上面基础之后，现在我们再来尝试解决前面提到的两个问题。

为什么我的程序占用了这么多内存？
1、在 main 函数添加钩子函数；
2、在程序启动时，开始抓取 log，直到你要查看的点；
3、使用工具对其进行分析，获取内存分布情况，在分析工具中可以根据需要做一些统计，着重关注大块内存分配。

有时，往往一个进程中是由几个 team 共同完成的，那么经常遇到的一个问题就是，进程经过一个操作后，内存大量增加了，我们要查明它的原因。
1、 在 main 函数中添加钩子函数；
2、 在刚开始的时候不抓取 log，在操作开始之前抓取 log，在操作完成之后停止抓取 log。
3、 使用分析工具对其进行分析，过滤掉那些申请并释放的内存，剩余的 log 将记录了内存的增长。

解决内存泄漏的问题：
通过对分配并被释放了内存记录进行过滤，剩下的是没有被释放的内存，但这不代表它们是内存泄漏，有可能是确实是程序需要而分配的内存，如何将内存泄漏与之分离开来，这是一个难题。

这点单纯从 log，我们是无法区分的。
对于一个偶然事件，造成的内存泄漏，其危害性不大，而查找起来困难却很大，只能通过分析源代码的方式，一个个确认。；但对于一些经常使用的周期性事件，在每个周期哪怕泄漏几个字节，积少成多也会造成很大的危害，这也是我们查找的重点。

在这里针对周期性的内存泄漏，我给出一个方法：
对于一个周期的操作来说，比如说我现在在界面 A，按一个键进入界面 B，然后又按了一个键返回界面 A。
1、 在按键进入界面 B 的时候，进程会申请一些内存。
2、 在按键返回界面 A 的时候，进程成应该释放掉在步骤申请的内存；否则如果不断重复上面的动作，就将会导致系统内存被耗尽。
这就是我们查找内存泄漏的理论基础。

我的方法是：
1、 使用上面提到了 hook 机制，为每一次申请和释放记录 log。
2、 抓取一个事件周期完整的 log。
3、 使用我们上面提到的工具，过滤掉所有申请并释放了的 log 记录。

这样，我们能通过程序里面的 log，来区分出这个事件周期，然后查找这个事件周期内剩余

的内存记录，我们就认为是内存泄漏。

实际情况并不是这么简单。
有时程序出于性能上的考虑，它会缓存一些内存，比如：
1、 用户现在在界面 A，按一个键进入界面 B。
程序申请了一块内存，用于事件队列；
2、 按一个键，返回界面 A。
程序并不释放上面申请的内存，如果使用我们上面提到的方法，会认为这块内存为内存泄漏。
3、 用户按键，进入界面 B。
程序不会再申请一块内存，用于事件队列，而是复用前面申请的内存。因此，理论上来讲，我们不应该认为这块内存为内存泄漏。
4、 用户按键，返回界面 A。

我们该如何解决上面的问题呢？
1、针对某一个事件周期，在抓取 log 前，对于多做几次该事件周期，目的就是将程序中的缓存填满，消除缓存对于我们分析的影响。

2、开始抓取 log 记录，并做几个事件周期，通过我们上面的方法来分析内存泄漏。

注意，最后一个事件周期在分析时，我们要将它忽略掉。因为最后一个事件周期中的一些内存分配记录，有可能会在下一个周期开始被释放掉。

2.7.3. 栈的回溯

在前面我们做内存调试的时候，在 malloc 的 hook 函数中调用 backtrace 打印出堆栈，但如果进程（或动态库）在使用 O2 优化后，打印出来的栈只有一个函数地址，这是为什么呢？

在这里，我们不得不先讲述一下栈帧回溯的原理：
在 ARM 的 APCS（ARM 过程调用标准中，对此做了规定）。
在 APCS 中，R11 也称 fp，被用作保留帧指针。
栈是链接起来的‘桢’的一个列表，通过一个叫做‘回溯结构’的东西来链接它们。这个结构存储在每个桢的高端。按递减地址次序分配栈的每一块。寄存器 sp 总是指向在最当前桢中最低的使用的地址。这符合传统上的满降序栈。
寄存器 fp (桢指针)应当是零或者是指向栈回溯结构的列表中的最后一个结构，提供了一种追溯程序的方式，来反向跟踪调用的函数。
回溯结构是:
地址高端　保存代码指针
[fp]　　fp 指向这里　返回 lr 值
[fp, #-4]　　返回 sp 值
[fp, #-8]　　返回 fp 值
[fp, #-12]　　指向下一个结构
…….

保存寄存器，和当前函数中的临时变量
……..
这个结构包含 4 至 27 个字，在方括号中的是可选的值。如果它们存在，则必须按给定的次序存在(例如，在内存中保存的 a3 下面可以是保存的 f4，但 a2-f5 则不能存在)。浮点值按‘内部格式’存储并占用三个字(12 字节)。

fp 寄存器指向当前执行的函数的栈回溯结构。返回 fp 值应当是零，或者是指向由调用了这个当前函数的函数建立的栈回溯结构的一个指针。而这个结构中的返回 fp 值是指向调用了调用了这个当前函数的函数的函数的栈回溯结构的一个指针；并以此类推直到第一个函数。

在函数退出的时候，把返回连接值、返回 sp 值、和返回 fp 值装载到 pc、sp、和 fp 中。

```
#include <stdio.h>
void one(void);
void two(void);
void zero(void);
int main(void)
{
	one();
	return 0;
}
void one(void)
{
	zero();
	two();
	return;
}
void two(void)
{
	printf("main...one...two\n");
	return;
}
void zero(void)
{
	return;
}
```

当它在屏幕上输出消息的时候， APCS 回溯结构将是:

```
fp ----> two_structure
		return link
		return sp
		return fp    ----> one_structure                    ...
				return link
				return sp
```

return fp ----> main_structure
return link
return sp
return fp---->0

所以，我们可以检查 fp 并参看给函数‘two’的结构，它指向给函数‘one’的结构，它指向给‘main’的结构，它指向零来终结。在这种方式下，我们可以反向追溯整个程序并 确定我们是如何到达当前的崩溃点的。值得指出‘zero’函数，因为它已经被执行并退出了，此时我们正在做它后面的打印，所以它曾经在回溯结构中，但现在不在了。值得指出的还有对于给定代码不太可能总是生成象上面那样的一个 APCS 结构。原因是不调用任何其他函数的函数不要求完全的 APCS 头部。

实际上呢，这个帧的结构不是必须的，GCC 在做优化的时候，为了节省出寄存器 fp，往往不维护这个帧的结构，但却给我们的调试带来很多的麻烦。

我们先来看一下带有正常帧的汇编结果。

```
00008414 <main>:
    8414:       e1a0c00d        mov     ip, sp
    8418:       e92dd800        stmdb   sp!, {fp, ip, lr, pc}
    841c:       e24cb004        sub     fp, ip, #4      ; 0x4
    8420:       eb000001        bl      842c <one>
    8424:       e3a00000        mov     r0, #0  ; 0x0
    8428:       e89da800        ldmia   sp, {fp, sp, pc}

0000842c <one>:
    842c:       e1a0c00d        mov     ip, sp
    8430:       e92dd800        stmdb   sp!, {fp, ip, lr, pc}
    8434:       e24cb004        sub     fp, ip, #4      ; 0x4
    8438:       eb000008        bl      8460 <zero>
    843c:       eb000000        bl      8444 <two>
    8440:       e89da800        ldmia   sp, {fp, sp, pc}

00008444 <two>:
    8444:       e1a0c00d        mov     ip, sp
    8448:       e92dd800        stmdb   sp!, {fp, ip, lr, pc}
    844c:       e24cb004        sub     fp, ip, #4      ; 0x4
    8450:       e59f0004        ldr     r0, [pc, #4]    ; 845c <.text+0xf0>
    8454:       ebffffc1        bl      8360 <__plt_header+0x38>
    8458:       e89da800        ldmia   sp, {fp, sp, pc}
    845c:       00008538        andeq   r8, r0, r8, lsr r5

00008460 <zero>:
    8460:       e1a0c00d        mov     ip, sp
    8464:       e92dd800        stmdb   sp!, {fp, ip, lr, pc}
    8468:       e24cb004        sub     fp, ip, #4      ; 0x4
```

```
    846c:       e89da800        ldmia   sp, {fp, sp, pc}
```

这三句是形成标准栈帧的最小结构。

```
mov     ip, sp
stmdb   sp!, {fp, ip, lr, pc}
sub     fp, ip, #4      ; 0x4
```

我们再来看看优化后的，不形成标准栈帧的反汇编结果。

```
00008414 <zero>:
    8414:       e12fff1e        bx      lr

00008418 <two>:
    8418:       e59f0000        ldr     r0, [pc, #0]    ; 8420 <.text+0xb4>
    841c:       eaffffc6        b       833c <__plt_header+0x14>
    8420:       00008508        andeq   r8, r0, r8, lsl #10

00008424 <one>:
    8424:       eaffffb         b       8418 <two>

00008428 <main>:
    8428:       e52de004        str     lr, [sp, #-4]!
    842c:       e24dd004        sub     sp, sp, #4      ; 0x4
    8430:       ebfffffb        bl      8424 <one>
    8434:       e3a00000        mov     r0, #0  ; 0x0
    8438:       e28dd004        add     sp, sp, #4      ; 0x4
    843c:       e8bd8000        ldmia   sp!, {pc}
```

可以看到优化后的代码，十分简单，象 one 函数，甚至都没有将自己的地址入栈的动作。

在我们经常使用的优化选项 O2 中，缺省就不形成标准的栈帧。这个选项给我们带来了性能的提高同时，也会为我们的代码调试带来不便。

比如说，当程序 coredump 的时候，无法准确打印出线程的栈出来，从而给代码的调试带来很多的困难。

这时候，我们可以在编译的时候加上-mapcs-frame，形成标准的栈帧就可以了。

2.7.4. 化整为零法

虽然通过上面的方法，我们能够了解到每一块内存的来历，和它的生存周期，但由于分配的内存的 log 实在是太多，而且中间又夹杂着泄漏和 cache、内存碎片等等种种情况，注定了检查内存泄漏将是一个非常累人的事情。

我也经常遇到，程序执行到一个操作，内存立刻就上去了，操作完成后，内存又恢复不到原

来的水平，十分令人头疼。

实际上，我们也注意到一般只有 daemon 进程，才会去花大力气去检测内存泄漏。对于进程的内存，首先我们要考虑争取其不是 daemon 进程，或者简化其 daemon 的部分。

1、 检查进程之所以是 daemon 进程的原因，如果是因为启动速度的原因，可以按后面的章节去进行优化。

2、 如果是因为需要侦听事件，那么争取把侦听部分设计成 daemon，代码和逻辑做到简单，检查内存泄漏也方便；对于业务部分，另起一个进程来完成，在完成业务后，该进程自动退出，从而不占用系统内存。inet 的设计就是一个很好的例子。

3、 将某些相对独立，对内存影响较大的部分，分离出来通过进程来实现，这样业务逻辑完成后，其所占用的内存会随进程的销毁而得到释放。

比如说，我们要在地址簿里，为某一个名片拍张照片。

起初的实现方式：将拍照片的这个功能采用动态库来实现，地址簿的进程调用相应接口就可以了。

其优点是：进程通过调接口的方式，实现拍照功能，进程内部通讯，实现起来简单。

缺点是：拍照片所分配的内存都将算在地址簿的堆内存中，如果拍照片的接口存在内存泄漏，将会影响到地址簿这个应用程序。

我们可以将拍照片这个功能，通过进程来实现，地址簿调用这样一个非 daemon 的进程完成拍照工作。

其优点是：如果拍照片的功能存在内存泄漏，那么其分配的内存将会随着这个进程的销毁而释放。

缺点是：这涉及到进程之间的通讯，需要妥善解决。

下面我们再来介绍几种常用的 memory 检查工具，dmalloc 和 valgrind。

2.7.5. dmalloc

dmalloc 的使用：

1、 从从www.dmalloc.com上下载dmalloc的源代码。

2、 按照你嵌入式平台的要求对其进行编译，生成静态链接库 libdmalloc.a。

3、 在需要检测的源代码中，包含 dmalloc.h。

```
#include <dmalloc.h>
```

4、 重新编译程序代码。

```
gcc –ldmalloc –o program code.c
```

5、 在嵌入式设备中，设置环境变量，定义 log 的输出文件。

```
Export DMALLOC_OPTIONS=log=logname,debug=0x3
```

6、 运行你的程序，待程序退出后，会将内存信息输出到 log 文件中。

```
1134286917: 23: Dmalloc version '5.4.2' from 'http://dmalloc.com/'
```

```
1134286917: 23: flags = 0x3, logfile 'logfile'
1134286917: 23: interval = 0, addr = 0, seen # = 0, limit = 0
1134286917: 23: starting time = 1134286917
1134286917: 23: process pid = 12088
1134286917: 23: WARNING: tried to free(0) from 'module.c:23'
1134286917: 27: WARNING: tried to free(0) from 'module.c:23'
1134286917: 27: Dumping Chunk Statistics:
1134286917: 27: basic-block 4096 bytes, alignment 8 bytes
1134286917: 27: heap address range: 0x111000 to 0x49c000, 3715072 bytes
1134286917: 27:         user blocks: 4 blocks, 16048 bytes (39%)
1134286917: 27:       admin blocks: 6 blocks, 24576 bytes (60%)
1134286917: 27:       total blocks: 10 blocks, 40960 bytes
1134286917: 27: heap checked 0
1134286917: 27: alloc calls: malloc 14, calloc 0, realloc 0, free 13
1134286917: 27: alloc calls: recalloc 0, memalign 0, valloc 0
1134286917: 27: alloc calls: new 0, delete 0
1134286917: 27:     current memory in use: 432 bytes (3 pnts)
1134286917: 27:   total memory allocated: 6204 bytes (14 pnts)
1134286917: 27:   max in use at one time: 4988 bytes (12 pnts)
1134286917: 27: max alloced with 1 call: 4124 bytes
1134286917: 27: max unused memory space: 4740 bytes (48%)
1134286917: 27: top 10 allocations:
1134286917: 27:   total-size   count in-use-size   count   source
1134286917: 27:         4124       1           0       0   ra=0x6e5910
1134286917: 27:          864       6           0       0   module.c:43
1134286917: 27:          432       3           0       0   module.c:133
1134286917: 27:          432       3         432       3   module.c:185
1134286917: 27:          352       1           0       0   ra=0x6b276d
1134286917: 27:         6204      14         432       3   Total of 5
1134286917: 27: Dumping Not-Freed Pointers Changed Since Start:
1134286917: 27:   not freed: '0x111400|s1' (144 bytes) from 'module.c:185'
1134286917: 27:   not freed: '0x111500|s1' (144 bytes) from 'module.c:185'
1134286917: 27:   not freed: '0x111600|s1' (144 bytes) from 'module.c:185'
1134286917: 27:   total-size   count   source
1134286917: 27:          432       3   module.c:185
1134286917: 27:          432       3   Total of 1
1134286917: 27: ending time = 1134286917, elapsed since start = 0:00:00
```

dmalloc 的原理；
在 malloc.h 中，我们可以看到

```
#define malloc(size) \
  dmalloc_malloc(__FILE__, __LINE__, (size), DMALLOC_FUNC_MALLOC, 0, 0)
#undef calloc
```

```
#define calloc(count, size) \
  dmalloc_malloc(__FILE__, __LINE__, (count)*(size), 
DMALLOC_FUNC_CALLOC, 0, 0)
#undef realloc
#define realloc(ptr, size) \
  dmalloc_realloc(__FILE__, __LINE__, (ptr), (size), 
DMALLOC_FUNC_REALLOC, 0)
#undef recalloc
#define recalloc(ptr, size) \
  dmalloc_realloc(__FILE__, __LINE__, (ptr), (size), 
DMALLOC_FUNC_RECALLOC, 0)
#undef memalign
#define memalign(alignment, size) \
  dmalloc_malloc(__FILE__, __LINE__, (size), DMALLOC_FUNC_MEMALIGN, \
        (alignment), 0 /* no xalloc */)
```

dmallo 通过宏定义，将我们代码中的 malloc、realloc 等函数转到了自身定义的函数中，从而获得了控制权，在自身的代码中去跟踪和检测堆内存的分配和释放。

dmalloc：
1、 运行起来简单；
2、 能够将内存分配与代码行对应起来。

dmalloc 的局限：
1、 需要修改所有使用到 malloc 等函数的文件，对于大的项目来讲，使用起来非常麻烦。
2、 必须拥有源码，随着现代软件的复杂度越来越高，一个 process 往往使用了很多个动态库，如果检查一个 process 的内存情况，那么动态库中调用了 malloc 等函数，也需要添加头文件，工作量将非常的大。
3、 只能等待进程运行结束后，才能获得结果，这对于那些 daemon 进程不适用。
4、 虽然能够获得内存分配的 log 记录，精确到哪个文件哪行，但是没有如何调用到该函数的栈信息。
使用 dmalloc 将增加系统的健壮性、但不能完全检测出所有的内存错误。

因此，对于一些小的非 daemon 的 process，我们可以考虑使用 dlmalloc。

2.7.6. valgrind

Valgrind 可以说一款非常强大的产品，它可以帮助程序员程序员检测无效的内存分配，访问未初始化的内存，内存泄漏等等，其功能几乎可以与在 windows 下的 Rational purity 工具相媲美，更加难能可贵的其是基于 GPL 的开源工具，我们可以免费使用。

Valgrind 可以运行在 x86,amd64 和 ppc32 的架构上，可惜不能执行在 arm 平台上面，如果你嵌入式软件可以在 x86 的虚拟机上进行测试的话，可以考虑使用。
valgrind 包含一个核心，它提供一个虚拟的 CPU 运行程序，还有一系列的工具，它们完成调

试，剖析和一些类似的任务。
它的一个好处在于，它是对程序二进制代码进行跟踪、调试，不必修改代码、重新编译程序等，使用起来极其方便。

valgrind的官方网址是：http://valgrind.org，你可以在这里获得源码，并进行编译。

valgrind 包含几个标准的工具，它们是：
1、memcheck
memcheck 探测程序中内存管理存在的问题。它检查所有对内存的读/写操作，并截取所有的 malloc/new/free/delete 调用。因此 memcheck 工具能够探测到以下问题：
1）使用未初始化的内存
2）读/写已经被释放的内存
3）读/写内存越界
4）读/写不恰当的内存栈空间
5）内存泄漏
6）使用 malloc/new/new[]和 free/delete/delete[]不匹配。

2、cachegrind
cachegrind 是一个 cache 剖析器。它模拟执行 CPU 中的 L1，D1 和 L2 cache，因此它能很精确的指出代码中的 cache 未命中。如果你需要，它可以打印出 cache 未命中的次数，内存引用和发生 cache 未命中的每一行代码，每一个函数，每一个模块和整个程序的摘要。如果你要求更细致的信息，它可以打印出每一行机器码的未命中次数。在 x86 和 amd64 上，cachegrind 通过 CPUID 自动探测机器的 cache 配置，所以在多数情况下它不再需要更多的配置信息了。

3、helgrind
helgrind 查找多线程程序中的竞争数据。helgrind 查找内存地址，那些被多于一条线程访问的内存地址，但是没有使用一致的锁就会被查出。这表示这些地址在多线程间访问的时候没有进行同步，很可能会引起很难查找的时序问题。

使用 valgrind 检测你的程序：
valgrind 被设计成非侵入式的，它直接工作于可执行文件上，因此在检查前不需要重新编译、连接和修改你的程序。要检查一个程序很简单，只需要执行下面的命令就可以了

valgrind --tool=tool_name program_name

比如我们要对 ls -l 命令做内存检查，只需要执行下面的命令就可以了

valgrind --tool=memcheck ls -l

不管是使用哪个工具，valgrind 在开始之前总会先取得对你的程序的控制权，从可执行关联库里读取调试信息。然后在 valgrind 核心提供的虚拟 CPU 上运行程序，valgrind 会根据选择的工具来处理代码，该工具会向代码中加入检测代码，并把这些代码作为最终代码返回给 valgrind 核心，最后 valgrind 核心运行这些代码。

如果要检查内存泄漏，只需要增加--leak-check=yes 就可以了，命令如下

valgrind --tool=memcheck --leak-check=yes ls -l

不同工具间加入的代码变化非常的大。在每个作用域的末尾，memcheck 加入代码检查每一片内存的访问和进行值计算，代码大小至少增加 12 倍，运行速度要比平时慢 25 到 50 倍。

valgrind 模拟程序中的每一条指令执行，因此，检查工具和剖析工具不仅仅是对你的应用程序，还有对共享库，GNU C 库，X 的客户端库都起作用。

三、现在开始

首先，在编译程序的时候打开调试模式（gcc 编译器的-g 选项）。如果没有调试信息，即使最好的 valgrind 工具也将中能够猜测特定的代码是属于哪一个函数。打开调试选项进行编译后再用 valgrind 检查，valgrind 将会给你的个详细的报告，比如哪一行代码出现了内存泄漏。

当检查的是 C++程序的时候，还应该考虑另一个选项 -fno-inline。它使得函数调用链很清晰，这样可以减少你在浏览大型 C++程序时的混乱。比如在使用这个选项的时候，用 memcheck 检查 openoffice 就很容易。当然，你可能不会做这项工作，但是使用这一选项使得 valgrind 生成更精确的错误报告和减少混乱。

一些编译优化选项(比如-O2 或者更高的优化选项)，可能会使得 memcheck 提交错误的未初始化报告，因此，为了使得 valgrind 的报告更精确，在编译的时候最好不要使用优化选项。

如果程序是通过脚本启动的，可以修改脚本里启动程序的代码，或者使用--trace-children=yes 选项来运行脚本。

下面是用 memcheck 检查 ls -l 命令的输出报告，在终端下执行下面的命令

valgrind --tool=memcheck ls -l

程序会打印出 ls -l 命令的结果，最后是 valgrind 的检查报告如下：

==4187==

==4187== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 19 from 2)

==4187== malloc/free: in use at exit: 15,154 bytes in 105 blocks.

==4187== malloc/free: 310 allocs, 205 frees, 60,093 bytes allocated.

==4187== For counts of detected errors, rerun with: -v

==4187== searching for pointers to 105 not-freed blocks.

==4187== checked 145,292 bytes.

==4187==

==4187== LEAK SUMMARY:

==4187== definitely lost: 0 bytes in 0 blocks.

==4187== possibly lost: 0 bytes in 0 blocks.

==4187== still reachable: 15,154 bytes in 105 blocks.

==4187== suppressed: 0 bytes in 0 blocks.

==4187== Reachable blocks (those to which a pointer was found) are not shown.

==4187== To see them, rerun with: --show-reachable=yes

这里的“4187”指的是执行 ls -l 的进程 ID，这有利于区别不同进程的报告。memcheck 会给出报告，分配置和释放了多少内存，有多少内存泄漏了，还有多少内存的访问是可达的，检查了多少字节的内存。

下面举两个用 valgrind 做内存检查的例子

例子一 (test.c)：

```
#include <string.h>

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

	ptr = (char*) malloc(10);
	strcpy(ptr, "01234567890");

	return 0;
}
```

编译程序

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

用 valgrind 执行命令

valgrind --tool=memcheck --leak-check=yes ./test

报告如下

==4270== Memcheck, a memory error detector.

==4270== Copyright (C) 2002-2006, and GNU GPL'd, by Julian Seward et al.

==4270== Using LibVEX rev 1606, a library for dynamic binary translation.

==4270== Copyright (C) 2004-2006, and GNU GPL'd, by OpenWorks LLP.

==4270== Using valgrind-3.2.0, a dynamic binary instrumentation framework.

==4270== Copyright (C) 2000-2006, and GNU GPL'd, by Julian Seward et al.

==4270== For more details, rerun with: -v

==4270==

==4270== Invalid write of size 1

==4270== at 0x4006190: strcpy (mc_replace_strmem.c:271)

==4270== by 0x80483DB: main (test.c:8)

==4270== Address 0x4023032 is 0 bytes after a block of size 10 alloc'd

==4270== at 0x40044F6: malloc (vg_replace_malloc.c:149)

==4270== by 0x80483C5: main (test.c:7)

==4270==

==4270== Invalid write of size 1

==4270== at 0x400619C: strcpy (mc_replace_strmem.c:271)

==4270== by 0x80483DB: main (test.c:8)

==4270== Address 0x4023033 is 1 bytes after a block of size 10 alloc'd

==4270== at 0x40044F6: malloc (vg_replace_malloc.c:149)

==4270== by 0x80483C5: main (test.c:7)

==4270==

==4270== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 12 from 1)

==4270== malloc/free: in use at exit: 10 bytes in 1 blocks.

==4270== malloc/free: 1 allocs, 0 frees, 10 bytes allocated.

==4270== For counts of detected errors, rerun with: -v

==4270== searching for pointers to 1 not-freed blocks.

==4270== checked 51,496 bytes.

==4270==

==4270==

==4270== 10 bytes in 1 blocks are definitely lost in loss record 1 of 1

==4270== at 0x40044F6: malloc (vg_replace_malloc.c:149)

==4270== by 0x80483C5: main (test.c:7)

==4270==

==4270== LEAK SUMMARY:

==4270== definitely lost: 10 bytes in 1 blocks.

==4270== possibly lost: 0 bytes in 0 blocks.

==4270== still reachable: 0 bytes in 0 blocks.

==4270== suppressed: 0 bytes in 0 blocks.

==4270== Reachable blocks (those to which a pointer was found) are not shown.

==4270== To see them, rerun with: --show-reachable=yes

从这份报告可以看出，进程号是 4270，test.c 的第 8 行写内存越界了，引起写内存越界的是 strcpy 函数，

第 7 行泄漏了 10 个字节的内存，引起内存泄漏的是 malloc 函数。

例子二（test2.c)

```
#include <stdio.h>

int foo(int x)
{
	if (x < 0) {
		printf("%d ", x);
	}

	return 0;
}

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

	foo(x);

	return 0;
}
```

编译程序

```
gcc -g -o test2 test2.c
```

用 valgrind 做内存检查

```
valgrind --tool=memcheck ./test2
```

输出报告如下

```
==4285== Memcheck, a memory error detector.

==4285== Copyright (C) 2002-2006, and GNU GPL'd, by Julian Seward et al.

==4285== Using LibVEX rev 1606, a library for dynamic binary translation.

==4285== Copyright (C) 2004-2006, and GNU GPL'd, by OpenWorks LLP.
```

==4285== Using valgrind-3.2.0, a dynamic binary instrumentation framework.

==4285== Copyright (C) 2000-2006, and GNU GPL'd, by Julian Seward et al.

==4285== For more details, rerun with: -v

==4285==

==4285== Conditional jump or move depends on uninitialised value(s)

==4285== at 0x8048372: foo (test2.c:5)

==4285== by 0x80483B4: main (test2.c:16)

==4285==p p

==4285== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 12 from 1)

==4285== malloc/free: in use at exit: 0 bytes in 0 blocks.

==4285== malloc/free: 0 allocs, 0 frees, 0 bytes allocated.

==4285== For counts of detected errors, rerun with: -v

==4285== All heap blocks were freed -- no leaks are possible.

从这份报告可以看出进程 PID 是 4285，test2.c 文件的第 16 行调用了 foo 函数，在 test2.c 文件的第 5 行 foo 函数使用了一个未初始化的变量。

valgrind 还有很多使用选项，具体可以查看 valgrind 的 man 手册页和 valgrind 官方网站的在线文档。

2.8. 嵌入式系统

系统内存的优化主要包括两个目的：
1、 系统空闲内存尽可能的多，这样可以在系统中同时运行多个进程。
2、 在经过长时间运行后，系统的空闲内存还要保持在一个比较高的水平，这样系统的性能才不会下降的太厉害。

系统空闲内存尽可能的多：
1、 减少常驻内存进程的数量。
2、 对于常驻内存的进程，要去优化是内存的使用，包括我们前面提到的各个部分。

系统经过长时间运行，空闲内存仍然保持一个较高的水平：
针对于常驻内存的进程，检测是否存在内存泄漏。这个问题虽然说起来很容易，在实际调试过程中，往往花费很大的精力。

2.8.1. tmpfs 分区

在 linux 中，为了加快对文件的读写，基于内存建立与了一个文件系统我们称之为 ramdisk 或者 tmpfs。对于该文件系统的访问，要比访问 flash 快的多。在对这个文件分区进行读写时，我们要时刻提醒自己，它是占用物理内存的。对于这块内存的使用，也要注意及时释放。

因此我们在做系统的内存优化的时候，不要忘掉这块内存。

在 linux 下，你可以使用 df 的命令，来查看分区所占内存空间的大小。

2.8.2. Cache 与 Buffer

如何查看系统的空闲内存有多少呢？
有两种方法：
cat /proc/meminfo
MemTotal: 55892 kB
MemFree: 2472 kB
Buffers: 3048 kB
Cached: 24052 kB
SwapCached: 0 kB
……..
或者使用 free 命令：
busybox free

	total	used	free	shared	buffers
Mem:	55892	53444	2448	0	3060
Swap:	0	0	0		
Total:	55892	53444	2448		

total=used +free.
上面可以看到，系统总共有 56M 内存，但 free 的 2.448M。只有 2.448M 内存，系统还能跑的起来吗？
实际上对于已经进程来讲，它可以使用的内存应该是=buffer+cache+free=2472+3048+24052.

什么是 Cache？
在这里不要与 CPU 上面的一级 Cache 和二级 Cache 相混淆，不可能为一个手机配几十 M 的 Cache。
Linux 中有一个概念，内存这种东西不用白不用，与其放在那里浪费，不如尽可能的多用内存，来提高系统性能。这里的 Cache 指的是文件 Cache。
在 Linux 操作系统中，当应用程序需要读取文件中的数据时，操作系统先分配一些内存，

将数据从存储设备读入到这些内存中，然后再将数据分发给应用程序；当需要往文件中写数据时，操作系统先分配内存接收用户数据，然后再将数据从内存写到磁盘上。文件 Cache 管理指的就是对这些由操作系统分配，并用来存储文件数据的内存的管理。 Cache 管理的优劣通过两个指标衡量：一是 Cache 命中率，Cache 命中时数据可以直接从内存中获取，不再需要访问低速外设，因而可以显著提高性能；二是有效 Cache 的比率，有效 Cache 是指真正会被访问到的 Cache 项，如果有效 Cache 的比率偏低，则相当部分磁盘带宽会被浪费到读取无用 Cache 上，而且无用 Cache 会间接导致系统内存紧张，最后可能会严重影响性能。

/proc/sys/vm/pagecache
该文件与 /proc/sys/vm/buffermem 的工作内容一样，但它是针对文件的内存映射和一般高速缓存。
/proc/sys/vm/pgcache-max，可以用来设置最大的 pagecache 页面。

Linux 内核中文件预读算法的具体过程是这样的：对于每个文件的第一个读请求，系统读入所请求的页面并读入紧随其后的少数几个页面(不少于一个页面，通常是三个页面)，这时的预读称为同步预读。对于第二次读请求，如果所读页面不在 Cache 中，即不在前次预读的 group 中，则表明文件访问不是顺序访问，系统继续采用同步预读；如果所读页面在 Cache 中，则表明前次预读命中，操作系统把预读 group 扩大一倍，并让底层文件系统读入 group 中剩下尚不在 Cache 中的文件数据块，这时的预读称为异步预读。无论第二次读请求是否命中，系统都要更新当前预读 group 的大小。此外，系统中定义了一个 window，它包括前一次预读的 group 和本次预读的 group。任何接下来的读请求都会处于两种情况之一：第一种情况是所请求的页面处于预读 window 中，这时继续进行异步预读并更新相应的 window 和 group；第二种情况是所请求的页面处于预读 window 之外，这时系统就要进行同步预读并重置相应的 window 和 group。图 5 是 Linux 内核预读机制的一个示意图，其中 a 是某次读操作之前的情况，b 是读操作所请求页面不在 window 中的情况，而 c 是读操作所请求页面在 window 中的情况。

Linux 内核中文件 Cache 替换的具体过程是这样的：刚刚分配的 Cache 项链入到 inactive_list 头部，并将其状态设置为 active，当内存不够需要回收 Cache 时，系统首先从尾部开始反向扫描 active_list 并将状态不是 referenced 的项链入到 inactive_list 的头部，然后系统反向扫描 inactive_list，如果所扫描的项的处于合适的状态就回收该项，直到回收了足够数目的 Cache 项。Cache 替换算法如图 6 的算法描述伪码所示。

Buffer：

Buffer：缓冲区，一个用于存储速度不同步的设备或优先级不同的设备之间传输数据的区域。通过缓冲区，可以使进程之间的相互等待变少，从而使从速度慢的设备读入数据时，速度快的设备的操作进程不发生间断。

/proc/sys/vm/buffermem
该文件控制用于缓冲区内存的整个系统内存的数量（以百分比表示）。它有三个值，通过把用空格相隔的一串数字写入该文件来设置这三个值。
用于缓冲区的内存的最低百分比

如果发生所剩系统内存不多，而且系统内存正在减少这种情况，系统将试图维护缓冲区内存的数量。

用于缓冲区的内存的最高百分比

缺省设置：2 10 60

进程管理中另一个程序 pdflush 用于将内存中的内容和文件系统进行同步，比如说，当一个文件在内存中进行修改，pdflush 负责将它写回硬盘。

ps -ef | grep pdflush

root 28 3 0 23:01 ? 00:00:00 [pdflush]

root 29 3 0 23:01 ? 00:00:00 [pdflush]

每当内存中的垃圾页（dirty page）超过 10%的时候，pdflush 就会将这些页面备份回硬盘。这个比率是可以调节的，通过参数 vm.dirty_background_ratio。

sysctl -n vm.dirty_background_ratio 10

pdflush 同 PFRA 是独立运行的，当内核调用 LMR 时，LMR 就触发 pdflush 将垃圾页写回硬盘。

2.8.3. 内存交换与回收

对于 Linux 系统中的内存回收，主要包括两个部分：

1、 对于那些没有修改的物理页面，也就是非 dirty page，我们可以将该物理页面回收，返还系统。
2、 对于那些已经修改的物理页面，也就是 diry page，我们需要使用交换分区，将这些物理页面写到磁盘上进行保存，然后释放返还给系统。

不知道大家有没有注意到，实际上我们所使用的嵌入式设备很少有交换分区，这是为什么呢？

主要有两个方面的原因：

1、 在嵌入式设备中，大多使用 flash 卡，做为存储设备，而这种设备的擦写次数是有限的。如果在 flash 上面做交换分区的话，那么将会造成频繁的读写，非常影响 flash 卡的寿命。
2、 如果使用交换分区的话，系统也将变得很慢。

没有了交换分区，是不是意味着就没有了内存的回收机制了呢？

不是，在 Linux 系统中，内存回收机制，不是基于进程，而是基于页面，所以它也就更加灵活。你可以看到一个内核守护进程 kswapd，它就主要负责页面的回收。

由于系统没有交换分区，所以只能释放那些非 dirty page 的页面。

让我们再来回顾一下，进程中都有哪些内存有可能是非 dirty page 页面。

1、代码段，其权限是只读属性，不可能被改写，所以其所占的物理内存，全部不是 dirty page。

2、数据段，其权限是可读、可写，所以其所占的物理内存，可能是 dirty page，也可能不

是 dirty page。
3、堆段，其没有对应的映射文件，内容都是通过程序改写的，所以其所占的物理内存，全部是 dirty page。
4、栈段，和堆段相同，其所占的物理内存，全部是 dirty page。
5、共享内存，其所占的物理内存，全部是 dirty page。

也就是说，在我们的系统中能够回收的内存只能是代码段和未修改的数据段。如果你做过长时间的系统进程内存使用分析，你会发现：
在进程跑了一段时间后，其代码段所占的物理内存减少了，这就是代码段内存回收的结果。

在 kswapd 中，有 2 个阀值，pages_hige 和 pages_low。当空闲内存页的数量低于 pages_low 的时候，kswapd 进程就会扫描内存并且每次释放出 32 个 free pages，直到 free page 的数量到达 pages_high。

我们可以在下面的文件，了解到 kswapd 的这两个阀值。
/proc/sys/vm/freepages
该文件控制系统怎么样应对各种级别的可用内存。它有三个值，通过把用空格相隔的一串数字写入该文件来设置这三个值。
如果系统中可用页面的数目达到了最低限制，则只允许内核分配一些内存。
如果系统中可用页面的数目低于这一限制，则内核将以较积极的方式启动交换，以释放内存，从而维持系统性能。
内核将试图保持这个数量的系统内存可用。低于这个值将启动内核交换。
缺省设置：512 768 1024

kswapd 在回收内存过程中还有两个重要的方法，一是 LMR（Low on memory reclaiming），另一个是 OMK(Out of Memory Killer)。当分配内存失败的时候 LMR 将会其作用，失败的原因是 kswapd 不能提供足够的空闲内存，这个时候 LMR 会每次释放 1024 个垃圾页直到内存分配成功。当 LMR 不能快速释放内存的时候，OMK 就开始其作用，OMK 会采用一个选择算法来决定杀死某些进程。当选定进程时，就会发送信号 SIGKILL，这就会使内存立即被释放。OMK 选择进程的方法如下：
1、进程占用大量的内存；
2、进程只会损失少量工作；
3、进程具有低的静态优先级；
4、进程不属于 root 用户。

/proc/sys/vm/kswapd
该文件控制允许内核怎么样交换内存。它有三个值，通过把用空格相隔的一串数字写入该文件来设置这三个值：
内核试图一次释放的最大页面数目。如果想增加内存交换过程中的带宽，则需要增加该值。
内核在每次交换中试图释放页面的最少次数。
内核在一次交换中所写页面的数目。这对系统性能影响最大。这个值越大，交换的数据越多，花在磁盘寻道上的时间越少。然而，这个值太大会因“淹没”请求队列而反过来影响系统性能。
缺省设置：512 32 8

2.8.4. 减少守护进程数量

在我们的系统中，有些进程在系统一启动就开始运行，直到关机。这类的进程对于整个系统内存影响重大，因此也是我们优化的重点。

守护进程对系统内存的影响：
1、 由于守护进程一直存活，所以其占用的内存将不会被释放。
2、 一个进程即使什么事情都不做，它引用了一些动态库，也会占用大量的物理内存。
3、 由于守护进程的生存周期很长，其哪怕只有一点内存泄漏，也会导致系统的内存耗尽。

我们的目标：
没有守护进程，所需要的服务都是在进程需要的时候启动，在不需要的时候退出。

为什么会有守护进程？
根据我的经验，可以总结为 2 个：
1、 系统需要守护进程来等待某些事件；
2、 有些进程提供服务的响应时间，达不到要求所以转变成了守护进程。

系统需要守护进程来等待某些事件
我们可以考虑将几个守护进程的等待事件部分合并到一个守护进程，当这个负责侦听的守护进程接收到消息后，再去启动相应的服务进程。
在 Linux 中 inetd 守护进程就是最好的例子。

现在我们守护进程的最大问题是没有区分中常驻部分和非常驻部分，往往是我们的程序业务逻辑运行完转入侦听时，非常驻部分并没有释放掉，还保存在内存中，占用了大量内存。并且由于没有区分常驻和非常驻部分，所以弄得侦听事件时，逻辑很复杂，增加了内存泄漏的概率。

有些进程提供服务的响应时间，达不到要求所以转变成了守护进程
我们要努力加快进程的启动速度，从而使服务进程达到按需启动的要求。这个问题我们将在后面的章节详细讲到，除了优化加载动态库外，我们还可以采用将服务进程改成线程，Preload 的方式等。

另外我们还可以采用将守护进程分解为若干个非驻留的子进程的方法，来降低守护进程的复杂度。

总之，一句话尽量减少守护进程的数量。

2.8.5. /proc/sys/vm/优化

1) /proc/sys/vm/block_dump
该文件表示是否打开 Block Debug 模式，用于记录所有的读写及 Dirty Block 写回动作。

缺省设置：0，禁用 Block Debug 模式
2) /proc/sys/vm/dirty_background_ratio
该文件表示脏数据到达系统整体内存的百分比，此时触发 pdflush 进程把脏数据写回磁盘。

缺省设置：10
3) /proc/sys/vm/dirty_expire_centisecs
该文件表示如果脏数据在内存中驻留时间超过该值，pdflush 进程在下一次将把这些数据写回磁盘。

缺省设置：3000（1/100 秒）
4) /proc/sys/vm/dirty_ratio
该文件表示如果进程产生的脏数据到达系统整体内存的百分比，此时进程自行把脏数据写回磁盘。

缺省设置：40
5) /proc/sys/vm/dirty_writeback_centisecs
该文件表示 pdflush 进程周期性间隔多久把脏数据写回磁盘。

缺省设置：500（1/100 秒）
6) /proc/sys/vm/vfs_cache_pressure
该文件表示内核回收用于 directory 和 inode cache 内存的倾向；缺省值 100 表示内核将根据 pagecache 和 swapcache，把 directory 和 inode cache 保持在一个合理的百分比；降低该值低于 100，将导致内核倾向于保留 directory 和 inode cache；增加该值超过 100，将导致内核倾向于回收 directory 和 inode cache。

缺省设置：100
7) /proc/sys/vm/min_free_kbytes
该文件表示强制 Linux VM 最低保留多少空闲内存（Kbytes）。

缺省设置：724（512M 物理内存）
8) /proc/sys/vm/nr_pdflush_threads
该文件表示当前正在运行的 pdflush 进程数量，在 I/O 负载高的情况下，内核会自动增加更多的 pdflush 进程。

缺省设置：2（只读）
9) /proc/sys/vm/overcommit_memory
该文件指定了内核针对内存分配的策略，其值可以是 0、1、2。
0， 表示内核将检查是否有足够的可用内存供应用进程使用；如果有足够的可用内存，内存

申请允许；否则，内存申请失败，并把错误返回给应用进程。
1， 表示内核允许分配所有的物理内存，而不管当前的内存状态如何。
2， 表示内核允许分配超过所有物理内存和交换空间总和的内存（参照 overcommit_ratio）。

缺省设置：0
10) /proc/sys/vm/overcommit_ratio
该文件表示，如果 overcommit_memory=2，可以过载内存的百分比，通过以下公式来计算系统整体可用内存。
系统可分配内存=交换空间+物理内存*overcommit_ratio/100

缺省设置：50（%）
11) /proc/sys/vm/page-cluster
该文件表示在写一次到 swap 区的时候写入的页面数量，0 表示 1 页，1 表示 2 页，2 表示 4 页。

缺省设置：3（2 的 3 次方，8 页）
12) /proc/sys/vm/swapiness
该文件表示系统进行交换行为的程度，数值（0-100）越高，越可能发生磁盘交换。

缺省设置：60
13) legacy_va_layout
该文件表示是否使用最新的 32 位共享内存 mmap()系统调用，Linux 支持的共享内存分配方式包括 mmap()，Posix，System VIPC。
0， 使用最新 32 位 mmap()系统调用。
1， 使用 2.4 内核提供的系统调用。

缺省设置：0
14) nr_hugepages
该文件表示系统保留的 hugetlb 页数。
15) hugetlb_shm_group
该文件表示允许使用 hugetlb 页创建 System VIPC 共享内存段的系统组 ID。
16) 待续。。。
四、/proc/sys/fs/优化
1) /proc/sys/fs/file-max
该文件指定了可以分配的文件句柄的最大数目。如果用户得到的错误消息声明由于打开文件数已经达到了最大值，从而他们不能打开更多文件，则可能需要增加该值。

缺省设置：4096
建议设置：65536
2) /proc/sys/fs/file-nr
该文件与 file-max 相关，它有三个值：
已分配文件句柄的数目
已使用文件句柄的数目
文件句柄的最大数目

该文件是只读的，仅用于显示信息。

2.8.6. 系统内存分析

对于项目的管理者来言，其要控制整个系统的内存消耗，在系统内存出现问题的时候，要能够定位到某一个进程，否则会导致各个 team 互相扯皮。

系统内存分析主要面临两个问题：
1、 在某一个时间点上，空闲内存太少。
2、 随着使用时间的增长，空闲内存越来越少。

先来说第一个问题：在某一个时间点，空闲内存太少。
1、 查看 tmpfs 分区，是否有不再使用，但没有删除的文件。
2、 统计所有进程所栈的 dirty page，对于那些占用大量 dirty page 的 TOP10 的进程，要求其优化内存使用。
3、 减少守护进程的数量。
4、 要求每个 Team，按照前面提到的方法，优化自己的程序。

第二个问题：随着使用时间的增长，空闲内存越来越少。
1、 要求所有的守护进程，检查内存泄漏的问题。
2、 对于某些常用的操作，做专项测试，检查是否有内存泄漏。
3、 对系统进行日常使用随机测试，模拟正常用户使用，连续运行若干天，查看各个进程内存使用情况。如果某个进程内存使用大量增长，则要求其检查内存泄漏。

系统内存分析和测试，开始的越早越好，不必等功能完全开发完毕即可开始。

3. 速度

内存终于告一段落，下面我们将重点放在优化的另一个主题：速度。

虽然嵌入式设备的心（CPU）不如 PC，虽然用户的要求比 PC 更为苛刻，但我们无怨无悔，我期望我的程序飞起来，就象赛车手在不断挑战速度的极限一样。

速度令我兴奋。

3.1. 性能评价

如果老板对你说，这个程序跑的有些慢，你去优化一下。

我们面临的第一个问题便是如何定义“慢”。慢这是和每个人的感觉相关，因人而异。对于程序员来讲，我们需要定义什么是“快”，什么是“慢”，需要一个具体可测量的指标。这个指标应该由产品部门，根据用户的感觉给出。

性能的指标应该覆盖面尽可能的全，主要关注用户使用时的感受，在笔者优化实践中，描述一个产品的性能指标就将近 100 个。

性能的指标要相对精确，对于消费电子产品，一般时间精确到十分之一秒。

性能的评测，测试人员在对性能评测时，也要以数据来说话，最好使用高速摄影机来进行测量。

有了性能指标后，程序员才好工作，否则很有可能测试人员说慢，开发人员说快，最后不了了之；另外程序的优化，往往花费程序员大量的时间和精力，对程序的移植性和可读性也有所损害，有了性能指标，程序员才好适可而止，不必浪费太多的精力。

下面，我给出一张比较常用的性能数据列表。

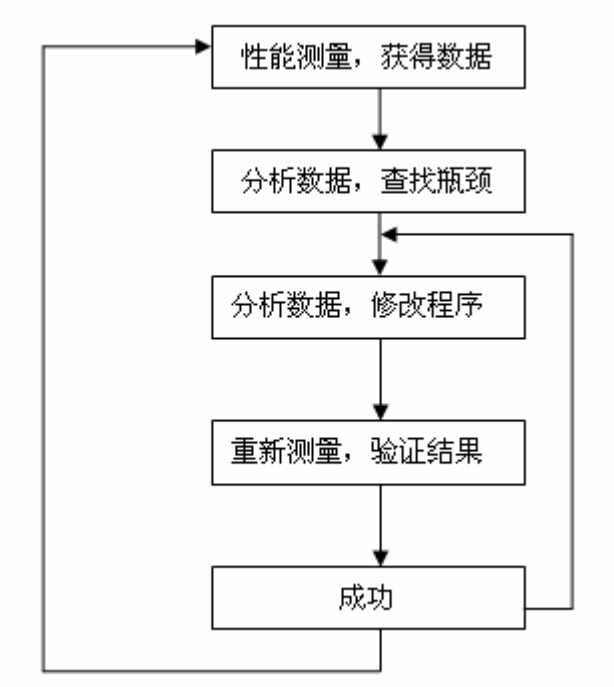
性能	理想	下限	版本 1	版本 2
开机	21000	30000		
关机	10000	20000		

3.2. 性能优化的方法

在知道了具体的指标之后，我们就开始了性能优化之旅：

1、 测量程序当前的状况，距离我们的目标还有多远，做到心中有数。
2、 分析我们的程序，查找到性能瓶颈，一般来讲 20%的代码占用了 80%的时间。
3、 修改程序
4、 重新测试，验证我们优化的结果。
5、 如果满足要求，就停止优化，过度的优化，有可能导致代码可读性的下降；如果不满足要求，我们需要分析测量结果，查找瓶颈。

整个优化过程，我们以一个图表来显示：



程序的优化是一个不断循环迭代的过程，这需要优化人员付出很多的时间和精力。

下面我们将分为几章介绍性能的优化：
1、 性能的评测
2、 shell 脚本的性能优化
3、 C 和 C++程序性能优化

3.3. 性能的评测

程序性能的问题，有很多原因，我们需要找到性能的瓶颈，来对症下药：

1、 程序的运算量很大，导致 CPU 过于繁忙，CPU 是瓶颈。
2、 程序需要做大量的 I/O，读写文件、内存操作等等，CPU 更多的是处于等待，I/O 部分称为程序性能的瓶颈。
3、 程序之间相互等待，结果 CPU 利用率很低，但运行速度依然很慢，事务间的共享与死锁制约了程序的性能。

因为每种性能瓶颈的优化方法是不一样的，所以在开始着手优化之前，我们需要弄清楚程序性能的瓶颈在于是什么类型，找准方向。

同样要了解 CPU 和 I/O 的性能，我们可以通过 proc 目录来找到我们所需要的数据。

3.3.1. Proc 目录

3.3.1.1. 系统相关

到目前为止，我们对系统效率的原因还是一无所知。我们目前
cat /proc/stat
cpu 5116 0 7801 249195 60 41 55
cpu0 5116 0 7801 249195 60 41 55
intr 364129 0 0 0 0 0 0 0 0 7 7 93 0 0 0 0 69 0 0 0 0 3 4250 0 0 1 0 0 0 8561 262014 0 0 0 85936 839 2 0 0 1157 0 1152 0 33 0 0 0 0 1 2 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
ctxt 231056
btime 1167696676
processes 515
procs_running 1
procs_blocked 0

因为嵌入式设备中一般只有一个 CPU，所以我们只关注第一行。
cpu 5116 0 7801 249195 60 41 55

cpu 后面数值分别代表着 CPU 在不同状态下所用的时间，其单位为 jiffies（也就是）。01 秒），7 个数值的含义分别是：
user（5116）：从系统启动开始累计到当前时刻，用户态的 CPU 时间，不包含 nice 值为负的进程。
nice（0）：从系统启动开始累计到当前时刻，nice 值为负的进程所占用的 CPU 时间。
system(7801)：从系统启动开始累计到当前时刻，内核所占用的 CPU 时间。
idle(259195)：从系统启动开始累计到当前时刻，除硬盘 IO 等待时间以外其它等待时间。
iowait(60)：从系统启动开始累计到当前时刻，硬盘 IO 等待时间。
irq（41）：从系统启动开始累计到当前时刻，硬中断时间。
softirq（55）：从系统启动开始累计到当前时刻，软中断时间。

CPU 时间=user+system+nice+idle+iowait+irq+softirq

从这些数据中，我们可以分析出性能瓶颈在哪个状态。

CPU 的利用率=1-（idle）/（user+system+nice+idle+iowait+irq+softirq）。
根据 CPU 的利用率，我们可以知道当前系统的 CPU 的负载情况。

如果当前 CPU 很繁忙，导致系统整体很慢，会有几种情况。

1、我们程序代码有问题，导致占用了大量的 CPU，我们可以通过
CPU 用户态利用率=(user+nice)/(user+system+nice+idle+iowait+irq+softirq)。
来了解。

2、我们程序代码调用了大量的系统调用，导致 Linux 内核占用了大量的 CPU，我们可以通过
CPU 内核态利用率=（system）/（user+system+nice+idle+iowait+irq+softirq)。
来了解。

3、系统和 FLASH、内存等有大量的交互和等待，从而导致系统性能下降，我们可以通过
IO 利用率=(iowait)/（user+system+nice+idle+iowait+irq+softirq）
来了解。

通过这些我们可粗略定位系统的瓶颈为何种类型，从而进行有目的的优化。

cat /proc/loadavg
0.64 0.53 0.43 3/121 482
loadavg 主要检查当前的系统的负载情况。
0.64　　1-分钟平均负载
0.53　　5-分钟平均负载
0.43　　15-分钟平均负载
3　　在采样时刻，运行队列的任务的数目，与/proc/stat 的 procs_running 表示相同意思
121　　在采样时刻，系统中活跃的任务的个数（不包括运行已经结束的任务）
482　　最大的 pid 值，包括轻量级进程，即线程。

3.3.1.2. 进程相关

在 proc 目录下面，有很多以数字为名字的目录，这些数字对应着当前系统中的进程和线程，进入目录，可以看到很多文件，这些文件则标识着每个进程和线程的相关信息。
cat stat
510 (telnetd) S 367 367 367 0 -1 256 223 0 1 0 0 6 0 0 15 0 1 0 25058 3690496 155 4294967295 32768 645464 3204447952 3204447088 1091367176 0 0 4 0 3221834488 0 0 17 0 0 0 c2477820
每个参数解释如下：
pid=510 进程(包括轻量级进程，即线程)号
comm=telnetd 应用程序或命令的名字
task_state=S 任务的状态，R:runnign, S:sleeping (TASK_INTERRUPTIBLE), D:disk sleep (TASK_UNINTERRUPTIBLE), T: stopped, T:tracing stop,Z:zombie, X:dead
ppid=367 父进程 ID
pgid=367 线程组号
sid=367 该任务所在的会话组 ID
tty_nr=0(pts/3) 该任务的 tty 终端的设备号，INT（34817/256）=主设备号，（34817-主设备号）=次设备号

tty_pgrp=-1 终端的进程组号，当前运行在该任务所在终端的前台任务(包括 shell 应用程序)的 PID。
task->flags=256 进程标志位，查看该任务的特性
min_flt=223 该任务不需要从硬盘拷数据而发生的缺页（次缺页）的次数
cmin_flt=0 累计的该任务的所有的 waited-for 进程曾经发生的次缺页的次数目
maj_flt=1 该任务需要从硬盘拷数据而发生的缺页（主缺页）的次数
cmaj_flt=0 累计的该任务的所有的 waited-for 进程曾经发生的主缺页的次数目
utime=0 该任务在用户态运行的时间，单位为 jiffies
stime=6 该任务在核心态运行的时间，单位为 jiffies
cutime=0 累计的该任务的所有的 waited-for 进程曾经在用户态运行的时间，单位为 jiffies
cstime=0 累计的该任务的所有的 waited-for 进程曾经在核心态运行的时间，单位为 jiffies
priority=15 任务的动态优先级
nice=0 任务的静态优先级
num_threads=1 该任务所在的线程组里线程的个数
it_real_value=0 由于计时间隔导致的下一个 SIGALRM 发送进程的时延，以 jiffy 为单位.
start_time=25058 该任务启动的时间，单位为 jiffies
vsize=3690496 该任务的虚拟地址空间大小
rss=155(page) 该任务当前驻留物理地址空间的大小
Number of pages the process has in real memory,minu 3 for administrative purpose.
这些页可能用于代码，数据和栈。
rlim=4294967295（bytes） 该任务能驻留物理地址空间的最大值
start_code=32768 该任务在虚拟地址空间的代码段的起始地址
end_code=645464 该任务在虚拟地址空间的代码段的结束地址
start_stack=3204447952 该任务在虚拟地址空间的栈的结束地址
kstkesp=3204447088 esp(32 位堆栈指针) 的当前值, 与在进程的内核堆栈页得到的一致.
kstkeip=1091357176 指向将要执行的指令的指针, EIP(32 位指令指针)的当前值.
pendingsig=0 待处理信号的位图，记录发送给进程的普通信号
block_sig=0 阻塞信号的位图
sigign=4 忽略的信号的位图
sigcatch=0 被俘获的信号的位图
wchan=3221834488 如果该进程是睡眠状态，该值给出调度的调用点
nswap=0 被 swapped 的页数，当前没用
cnswap=0 所有子进程被 swapped 的页数的和，当前没用
exit_signal=17 该进程结束时，向父进程所发送的信号
task_cpu(task)=0 运行在哪个 CPU 上
task_rt_priority=0 实时进程的相对优先级别
task_policy=0 进程的调度策略，0=非实时进程，1=FIFO 实时进程；2=RR 实时进程

计算进程 CPU 占用率
主要思路：进程 CPU 占用率=进程占用 CPU 时间/系统总的时间。
进程占用 CPU 时间，可以从进程的 stat 文件获得（utime、stime、cutime、cstime），系统总的时间，需要通过其他方式获得，例如通过/proc/stat 或者通过 C 函数 gettime 获得。
要想获得进程 CPU 占用率，需要两个采样点：
采样点 1：

系统时间记为：sys1.
进程时间分别为：utime1　stime1　cutime1　cstime1
采样点 2：
系统时间记为：sys2
进程时间分别为:：utime2　stime2　cutime2　cstime2

进 程　CPU　占 用 率 =((utime2+stime2-cutime2-cstime2)-(utime1+stime1-cutime1-cstime1))/sys2-sys1

进程用户态占用率=((utime2-cutime2)-(utime1-cutime1))/sys2-sys1

进程内核态占用率=((stime2-cstime2)-(stime1-cstime1))/sys2-sys1

3.3.2. 相关工具

3.3.2.1. top

top 是最常用来监控系统范围内进程活动的工具，它提供运行在系统上的与 CPU 关系最密切的进程列表，以及许多有意义的统计值，例如负载平均、进程数量、以及使用的存储器和页面空间的数量。

一般我们使用

3.3.2.2. lmbench

我们使用的内存有多快？
这里我们介绍一个工具 lmbench。由于 lmbench 具有多种测量方法的能力，因此这里集中在四个特殊的方法上。前三个方法测量带宽：内存读的速度，内存写的速度和内存复制的速度。最后一个是内存读写等待时间的测量。

内存复制基准测试
内存复制基准测试分配一大块以 0 填充的内存，然后计算将内存的前半块复制到后半块所需的时间。结果以每秒移动的多少兆字节给出。

内存读基准测试
内存读基准测试分配一块以 0 填充的内存，然后计算在整数载入和相加时读这个内存所需的时间；每个字节整数被载入和累加到一个内存变量中。

内存写基准测试
内存写基准测试分配一块以 0 填充的内存，然后计算在一连串的 4 字节整数存储和递增时写这个内存所需的时间。

内存延迟基准测试

内存延迟基准测试计算从一块内存读一个字节所需的时间。结果以每个载入花费多少纳秒的形式给出。全部数据分级内存体系受到评价，包括第一缓存、第二缓存和主内存的等待时间，以及 TLB 错失的等待时间影响。为了从 lmbenc 内存延迟基准测试中获得更多的信息，请试着将数据转化为延迟的函数与测试所用的阵列大小对比。

```
                L M B E N C H  2 . 0  S U M M A R Y
                ------------------------------------


Basic system parameters
----------------------------------------------------
Host              OS Description            Mhz

--------- ------------- ---------------------- ----
Phone         Linux 2.6.10_                   arm  531

Processor, Processes - times in microseconds - smaller is better
----------------------------------------------------------------
Host              OS  Mhz null null     open selct sig  sig  fork exec sh
                        call  I/O stat clos TCP   inst hndl proc proc
proc
--------- ------------- ---- ---- ---- ---- ---- ----- ---- ---- ---- ----
----
Phone    Linux 2.6.10_  531 0.49 1.74 19.8 28.9  58.3 2.41 5.07 734. 968.
12.K

Context switching - times in microseconds - smaller is better
-------------------------------------------------------------
Host              OS 2p/0K 2p/16K 2p/64K 8p/16K 8p/64K 16p/16K 16p/64K
                     ctxsw  ctxsw  ctxsw ctxsw  ctxsw   ctxsw   ctxsw
--------- ------------- ----- ------ ------ ------ ------ ------- -------
Phone     Linux 2.6.10_ 3.710   21.9  164.2   64.3  396.9   115.6   403.0

*Local* Communication latencies in microseconds - smaller is better
---------------------------------------------------------------------
Host              OS 2p/0K  Pipe AF    UDP  RPC/   TCP  RPC/ TCP
                     ctxsw      UNIX        UDP         TCP conn
--------- ------------- ----- ----- ---- ----- ----- ----- ----- ----
Phone     Linux 2.6.10_ 3.710  19.6 36.8 101.1      133.7      630.

File & VM system latencies in microseconds - smaller is better
```

```
-------------------------------------------------------------
Host                 OS   0K File     10K File     Mmap    Prot    Page
                        Create Delete Create Delete  Latency Fault   Fault
--------- ------------- ------ ------ ------ ------  ------- -----
-----
phone    Linux 2.6.10_  771.0 1512.9 4219.4 2451.0   4343.0 1.110    12.0

*Local* Communication bandwidths in MB/s - bigger is better
-----------------------------------------------------------
Host                 OS  Pipe AF    TCP  File   Mmap  Bcopy  Bcopy  Mem
Mem
                           UNIX     reread reread (libc) (hand) read write
--------- ------------- ---- ---- ---- ------ ------ ------ ------ ----
-----
phone    Linux 2.6.10_ 79.1 94.7 13.3   47.1  135.9   78.6   77.8 135.
205.9

Memory latencies in nanoseconds - smaller is better
   (WARNING - may not be correct, check graphs)
------------------------------------------------
Host                 OS   Mhz  L1 $   L2 $    Main mem    Guesses
--------- -------------  ---- ----- ------    --------    -------
phone    Linux 2.6.10_   531 7.506   39.4  221.9
```

3.3.2.3. vmstat

vmstat 是 Virtual Meomory Statistics（虚拟内存统计）的缩写，是实时系统监控工具。该命令通过使用 knlist 子程序和/dev/kmen 伪设备驱动器访问这些数据，输出信息直接打印在屏幕。vmstat 反馈的与 CPU 相关的信息包括：

（1）多少任务在运行
（2）CPU 使用的情况
（3）CPU 收到多少中断
（4）发生多少上下文切换

下面只介绍 Vmstat 与 CPU 相关的参数
vmstat 的语法如下：

```
vmstat [delay [count]]
```

参数的含义如下：
参数 解释
delay 相邻的两次采样的间隔时间

count 采样的次数，count 只能和 delay 一起使用

当没有参数时，vmstat 则显示系统启动以后所有信息的平均值。有 delay 时，第一行的信息自系统启动以来的平均信息。从第二行开始，输出为前一个 delay 时间段的平均信息。当系统有多个 CPU 时，输出为所有 CPU 的平均值。

与 CPU 有关的输出的含义 (采用进一法)
参数 解释 从/proc/stat 获得数据
任务的信息
r 在 internal 时间段里，运行队列里等待 CPU 的任务（任务）的个数，即不包含 vmstat 进程 procs_running-1
b 在 internal 时间段里，被资源阻塞的任务数（I/0，页面调度，等等.），通常情况下是接近 0 的 procs_blocked
CPU 信息 所有值取整（四舍五入）
us 在 internal 时间段里，用户态的 CPU 时间(%)，包含 nice 值为负进程 (user+ nice)/ total*100
sy 在 internal 时间段里，核心态的 CPU 时间(%) (system+ irq+ softirq)/ total*100
id 在 internal 时间段里，cpu 空闲的时间，不包括等待 i/o 的时间(%) idle/ total*100
wa 在 internal 时间段里，等待 i/o 的时间(%) iowait/ total*100
系统信息
in 在 internal 时间段里，每秒发生中断的次数 intr/interval
cs 在 internal 时间段里，每秒上下文切换的次数，即每秒内核任务交换的次数 ctxt/interval

total_cur=user+system+nice+idle+iowait+irq+softirq
total_pre=pre_user+ pre_system+ pre_nice+ pre_idle+ pre_iowait+ pre_irq+ pre_softirq
total=total_cur-total_pre

范例 1：average mode (粗略信息)
当 vmstat 不带参数时，对应的输出值是从系统启动以来的平均值，而 r 和 b 则对应的是完成这一命令时，系统的值。从下面例子，可以看出系统基本出去闲置状态（idle）。自启动以来，CPU 在用户态消耗时间为 5%，在核心态消耗为本 1%，剩下的为闲置时间。需要指出的是：这里的用户态时间包括 nice 值为负的进程的时间。

```
[root@localhost ~]# vmstat
procs -----------memory---------- ---swap-- -----io---- --system-- ----cpu----
r b swpd free buff cache si so bi bo in cs us sy id wa
1 0 0 4580 428 98516 0 0 49 6 15 19 2 1 96 1
[root@localhost ~]#
```

范例 2：average mode (详细信息)
命令格式：
vmstat –s

这里只讨论与 CPU 相关信息。“CPU ticks”表示自系统启动 CPU 运行时间，这里以 tick 为时间单位。用 tick 来西安市 us,sy id 和 wa 的时间；forks 指自从系统启动以来，所创建的新任务的个数。这些信息从/proc/stat 的第一行和”processes”行获得。

[root@localhost ~]# vmstat -s
255280 total memory
244216 used memory
206624 active memory
21208 inactive memory
11064 free memory
628 buffer memory
91396 swap cache
255992 total swap
24 used swap
255968 free swap
973400 non-nice user cpu ticks
477 nice user cpu ticks
206168 system cpu ticks
43567714 idle cpu ticks
373234 IO-wait cpu ticks
62732 IRQ cpu ticks
1972 softirq cpu ticks
22366502 pages paged in
88756936 pages paged out
0 pages swapped in
0 pages swapped out
135634319 interrupts
137288441 CPU context switches
1134440368 boot time
208990 forks
[root@localhost ~]#

结果解释

参数 描述 /proc/stat
non-nice user cpu ticks 自系统启动以来，CPU 在用户态下运行非 nice 进程的时间，单位为 jiffies user
nice user cpu ticks 自系统启动以来，CPU 在用户态下运行 nice 进程的时间，单位为 jiffies nice
system cpu ticks 自系统启动以来，CPU 处于系统状态的时间，单位为 jiffies sys
idle cpu ticks 自系统启动以来，CPU 处于闲置状态的时间，单位为 jiffies idle
IO-wait cpu ticks 自系统启动以来，CPU 处理 IO 中断的时间，单位为 jiffies iowait
IRQ cpu ticks 自系统启动以来，CPU 处理硬中断的时间，单位为 jiffies irq
softing cpu ticks 自系统启动以来，CPU 处理软中断的时间，单位为 jiffies Softirq
interrupts 自系统启动以来，发生的所有的中断的次数目 Intr

CPU context switches 自系统启动以来，发生的上下文交换的次数 Ctxt
boot time 自系统启动以来到现在运行的时间，单位为秒。 btime
forks 自系统启动以来所创建的任务的个数目。 Process

范例 3：定期采样(delay [count])
定期采样数据是指每隔 delay 时间，采样一次。当 count 为 0 时，vmstat 将不停地定期报告信息；否则当报告 count 次后，vmstat 命令停止运行。
第一行的信息如同范例 1，是自系统启动以来的平均信息。从第二行开始，每行的意思是：r 和 b 采样那一时刻系统运行队列和等待队列的情况；而 usystem 参数（in,cs）以及 CPU 参数（us,sy,id,wa）对应的输出值是系统在前一个 delay 的情况。
从下面例子可以看出上下文交换的次数小于中断的发生次数。当系统大部分时间是空闲并且中断大部分是时间中断时，这种现象极可能发生。当时间中断发生时, 因为调度器没有什么任务可调度，所以很少发生上下文切换。

```
[root@localhost ~]# vmstat 2 4
procs -----------memory---------- ---swap-- -----io---- --system-- ----cpu----
r b swpd free buff cache si so bi bo in cs us sy id wa
1 0 24 11032 652 91396 0 0 49 6 15 19 2 1 96 1
0 0 24 11032 652 91396 0 0 0 0 377 464 1 0 99 0
0 0 24 11024 652 91396 0 0 0 0 387 476 1 0 100 0
0 0 24 11024 652 91396 0 0 0 0 323 377 0 0 100 0
[root@localhost ~]#
```

vmstat 命令详解
vmstat 是用来实时查看内存使用情况,反映的情况比用 top 直观一些.
如果直接使用,只能得到当前的情况,最好用个时间间隔来采集
vmstat T 其中 T 用具体的时间标示,单位是 秒 例如:vmstat 5 表格每隔 5 秒采集一次.
这样在刷新的时候就能比较系统的看到那个列不正常的
procs:
r-->;在运行队列中等待的进程数
b-->;在等待 io 的进程数
w-->;可以进入运行队列但被替换的进程

memoy
swap-->;现时可用的交换内存（k 表示）
free-->;空闲的内存（k 表示）

pages
re－－》回收的页面
mf－－》非严重错误的页面
pi－－》进入页面数（k 表示）
po－－》出页面数（k 表示）
fr－－》空余的页面数（k 表示）

de－－》提前读入的页面中的未命中数
sr－－》通过时钟算法扫描的页面

disk 显示每秒的磁盘操作。 s 表示 scsi 盘，0 表示盘号

fault 显示每秒的中断数
in－－》设备中断
sy－－》系统中断
cy－－》cpu 交换

cpu 表示 cpu 的使用状态
cs－－》用户进程使用的时间
sy－－》系统进程使用的时间
id－－》cpu 空闲的时间

其中:
如果 r 经常大于 4 ，且 id 经常少于 40，表示 cpu 的负荷很重。
如果 pi，po 长期不等于 0，表示内存不足。
如果 disk 经常不等于 0， 且在 b 中的队列 大于 3， 表示 io 性能不好。

3.4. 优化基本原则

1、等效原则：优化前后程序实现的功能一致。
2、有效原则：优化后要比优化前运行速度快或占用存储空间小，或二者兼有。
3、经济原则：优化程序要付出较小的代价，取得较好的结果。

在优化方面，我们有 5 个基本纲领，为我们提供前进的方向：

1、Do it faster。

为了达成一个功能，程序可以有很多种写法，有的方法速度会很慢，有的会很快，我们就是要找到最有效率的方法，来提高程序的运行速度。

2、Do it in parallel

程序往往要实现很多功能，如果需要串行执行的话，那么很有可能会使 CPU 等资源无法充分利用。如果程序能够并行执行的话，那么可以充分利用 CPU 等资源，从而达到加快速度的功能。

3、Do it later

对于时间紧迫的功能，有些不必要的功能，可以考虑延后执行，使程序的任务减轻，从而腾出资源，来做最重要的事。

4、Don't do it at all

在我们写程序中，为了模块，可读性好的目的，往往使程序多做了很多无用的事，如果我们把这些无用的事去掉，则可以加快程序的速度。

5、Do it before.

速度并不一直是用户关注的焦点，只是在某些特定点，如果我们把一些工作在系统空闲时完成，则可以减轻繁忙时程序的负担。

3.5. shell 脚本优化

在嵌入式 Linux 中，bash 脚本占有很大的比重。优化 shell 脚本有助于缩短系统的启动时间，进程的执行速度等等。在嵌入式 Linux 中，bash shell 一般是由 BusyBox 来实现的。在 BusyBox 中，命令主要被分为两类：built-ins 和 applets。Builts-in 在 busybox 中只是简单的函数，而 applets 则意味着需要调用“fork/exec”创建子进程来执行，并且 Busybox 也可以使用外部命令。
出于性能方面的考虑，倾向于使用 builts-in 来取代 applets 和外部命令，主要因为创建子进程来执行是非常耗时的，从而使脚本变得低效。
当初设计 busybox 时，将精力主要放在了减小文件大小方面，所以内置的 builts-in 比较少，大部分功能都需要通过 fork 子进程来完成。

3.5.1. Builts-in 和 applets

如何知道 busybox 支持哪些脚本？
你可以在系统中直接运行 busybox,它将打印出它所支持的所有功能：
busybox
BusyBox v1.01 (---) multi-call binary

Currently defined functions:
[, addgroup, adduser, arping, ash, awk, basename, bunzip2, busybox, bzcat, cat, chgrp, chmod, chown, chroot, chvt, clear, cmp, cp, cpio, crond,
……..

我们怎么样才能知道那些功能是 builts-in 与 applets 呢？
很遗憾这里没有命令来帮你定位，也很少有相关的文档，尤其是不同版本的 busybox 其支持的 builts-in 和 applets 有可能会变化。还好，我们有源码，源码就是最好的文档。

你可以到http://www.busybox.net/，下载busybox的源码。
在/include/applets.h 中，定义了 busybox 所支持的所有功能。
在/docs/nofork_noexec.txt 中，说明了 builts-in 和 applets 的区别。

实际上在 busybox 的源码中，并不区分 builts-in 和 applets，它全部都认为是 applets。但对 applets 进行了分类：
APPLET
APPLET_NOUSAGE
APPLET_ODDNAME
APPLET_NOFORK
APPLET_NOEXEC
APPLET：也即我们所说的 applets，它由 busybox 创建一个 fork 出一个子进程，然后调用 exec 执行相应的功能，待执行完毕后，返还控制给父进程。
APPLET_NOUSAGE：它属于 APPLET，只是进一步规定了没有相应的帮助文档。

APPLET_ODDNAME：
APPLET_NOEXEC：系统将调用 fork 创建子进程，然后执行 busybox 中对应的功能，在执行完毕后，返回控制给父进程。
APPLET_NOFORK：它相当于 builts-in，只是执行 busybox 的内部函数，不必创建子进程，所以其效率最高。

在 busybox 1.9 中，属于 APPLET_NOFORK 的功能有：

[basename cat dirname echo false hostid
length logname mkdir pwd rm rmdir seq
sleep sync touch true usleep whoami yes

属于 APPLET_NOEXEC 的功能有：

awk chgrp chmod chown cp cut dd find
hexdump ln sort test xargs

3.5.2. bash 脚本

在脚本中命令，还有一些特殊的约定：
1、 包含在 pipe 中的 builts-in 将被创建子进程来执行。
2、 包含在 ` 的命令将创建子进程来执行。

3.5.3. 如何优化 Busybox bash 脚本

1、 去掉脚本中无用的代码。
2、 尽可能的使用 busybox 中 built-ins 替换外部命令。
3、 尽可能的不使用 pipe
4、 减少 pipe 中的命令数
5、 尽可能不使用 `

3.5.4. Bash 脚本优化

3.6. 进程启动速度

在消费式电子产品中，非常注重用户的响应时间，这就与进程的启动速度高度相关。有些时候，由于进程的启动速度达不到要求，我们便不得不把他们改成 dameon 进程，来提高用户的响应速度。但我们知道系统中 dameon 进程的数量过多，会占用大量的物理内存，也会导致系统的整体性能下降。

关键的问题，我们还是要提高进程的启动速度。

进程的启动时间主要包括两部分：

1、 加载共享库。

2、 进程的启动代码。

进程的代码优化，在后面我们讲到。这里主要关注加载共享库的时间，我们很少关注，可是随着程序复杂性的越来越大，一个进程往往依赖几十个，甚至接近 100 个动态库，我们以 addrbk 为例，其在启动过程中要加载 72 个动态库，而加载这 72 个动态库，所用时间大约为 0.8898 秒，将近 1 秒，已经很多了。
因此，优化加载动态库，是提高进程启动速度很重要的一部分。

3.6.1. 查看动态库的加载过程

这里我们可以使用两个工具，来查看进程加载动态库的过程：

strace

```
# strace
usage: strace [-dffhiqrtttTvVxxn] [-a column] [-e expr] ... [-o file]
                      [-p pid] ... [-s strsize] [-u username] [-E var=val] ...
                      [command [arg ...]]
   or: strace -c [-e expr] ... [-O overhead] [-S sortby] [-E var=val] ...
                      [command [arg ...]]
-c -- count time, calls, and errors for each syscall and report summary
-f -- follow forks, -ff -- with output into separate files
-F -- attempt to follow vforks, -h -- print help message
-i -- print instruction pointer at time of syscall
-q -- suppress messages about attaching, detaching, etc.
-r -- print relative timestamp, -t -- absolute timestamp, -tt -- with usecs
-T -- print time spent in each syscall, -V -- print version
-v -- verbose mode: print unabbreviated argv, stat, termio[s], etc. args
-x -- print non-ascii strings in hex, -xx -- print all strings in hex
-a column -- alignment COLUMN for printing syscall results (default 40)
-e expr -- a qualifying expression: option=[!]all or option=[!]val1[,val2]...
    options: trace, abbrev, verbose, raw, signal, read, or write
-o file -- send trace output to FILE instead of stderr
-O overhead -- set overhead for tracing syscalls to OVERHEAD usecs
-p pid -- trace process with process id PID, may be repeated
-s strsize -- limit length of print strings to STRSIZE chars (default 32)
-S sortby -- sort syscall counts by: time, calls, name, nothing (default time)
-u username -- run command as username handling setuid and/or setgid
-E var=val -- put var=val in the environment for command
-E var -- remove var from the environment for command
```

-n -- print print the syscall number of each call.

使用 LD_DEBUG 来跟踪 loader 的过程：
LD_DEBUG=help ./hello
Valid options for the LD_DEBUG environment variable are:

libs	display library search paths
reloc	display relocation processing
files	display progress for input file
symbols	display symbol table processing
bindings	display information about symbol binding
versions	display version dependencies
all	all previous options combined
statistics	display relocation statistics
unused	determined unused DSOs
help	display this help message and exit

To direct the debugging output into a file instead of standard output
a filename can be specified using the LD_DEBUG_OUTPUT environment variable

3.6.2. 减少加载动态库的数量

在进程启动时加载的动态库，并不一定在进程启动时都要用到。不需要的动态库，我们可以在进程启动时加载动态库的清单中去掉，从而加载进程的启动速度。

如果我们想使用某一个动态库，又不想它在进程一启动就加载，那么我们可以使用 dlopen 来动态加载共享库。

dlopen 的好处在于可以精确控制动态库的生存周期，一方面可以减少动态库的内存使用；另一方面可以减少进程启动时加载动态库的时间。

将一些动态库，改变成静态库，最终编译到进程中，从而减少加载动态库的数量。

3.6.3. 共享库的搜索路径

在进程加载动态库时，它首先要去很多路径去搜索动态库，其搜索顺序是：

1、DT_NEED 入口中包含的路径
2、DT_RPATH 入口给出的路径（存在的话）
3、环境变量 LD_LIBRARY_PATH 路径（setuid 类的程序排除）
4、LD_RUNPATH 入口给出的路径（存在的话）
5、库高速缓存文件 ld.so.conf 中给出的路径

6、/lib /usr/lib

其中 DT_RPATH 和 LD_RUNPATH 是在程序编译时加的选项，这些信息会记录在进程中，在 loader 分析进程文件时，获取这些信息。

你可以使用-rpath 来设置 DT_RPATH。
gcc -o test test.c -L. -lfoo -lbar -Wl,-rpath=/home/app

实际上，上面的搜索路径还不完全，有一种比 DT_RPATH 更高优先级的目录搜索机制，HWCAP。
LD_DEBUG=libs ./hello
500: find library=libpthread.so.0; searching
500: search cache=/etc/ld.so.cache
500: search path=/lib/tls/v6l/fast-mult/half:/lib/tls/v6l/fast-mult:/lib/tls/v6l/half:/lib/tls/v6l:/lib/tls/fast-mult/half:/lib/tls/fast-mult:/lib/tls/half:/lib/tls:/lib/v6l/fast-mult/half:/lib/v6l/fast-mult:/lib/v6l/half:/lib/v6l:/lib/fast-mult/half:/lib/fast-mult:/lib/half:/lib:/usr/lib/tls/v6l/fast-mult/half:/usr/lib/tls/v6l/fast-mult:/usr/lib/tls/v6l/half:/usr/lib/tls/v6l:/usr/lib/tls/fast-mult/half:/usr/lib/tls/fast-mult:/usr/lib/tls/half:/usr/lib/tls:/usr/lib/v6l/fast-mult/half:/usr/lib/v6l/fast-mult:/usr/lib/v6l/half:/usr/lib/v6l:/usr/lib/fast-mult/half:/usr/lib/fast-mult:/usr/lib/half:/usr/lib (system search path)
500: trying file=/lib/tls/v6l/fast-mult/half/libpthread.so.0
500: trying file=/lib/tls/v6l/fast-mult/libpthread.so.0
500: trying file=/lib/tls/v6l/half/libpthread.so.0
500: trying file=/lib/tls/v6l/libpthread.so.0
500: trying file=/lib/tls/fast-mult/half/libpthread.so.0
500: trying file=/lib/tls/fast-mult/libpthread.so.0
500: trying file=/lib/tls/half/libpthread.so.0
500: trying file=/lib/tls/libpthread.so.0
500: trying file=/lib/v6l/fast-mult/half/libpthread.so.0
500: trying file=/lib/v6l/fast-mult/libpthread.so.0
500: trying file=/lib/v6l/half/libpthread.so.0
500: trying file=/lib/v6l/libpthread.so.0
500: trying file=/lib/fast-mult/half/libpthread.so.0
500: trying file=/lib/fast-mult/libpthread.so.0
500: trying file=/lib/half/libpthread.so.0
500: trying file=/lib/libpthread.so.0
500: find library=libc.so.6; searching
500: search cache=/etc/ld.so.cache
500: search path=/lib:/usr/lib/tls/v6l/fast-mult/half:/usr/lib/tls/v6l/fast-mult:/usr/lib/tls/v6l/half:/usr/lib/tls/v6l:/usr/lib/tls/fast-mult/half:/usr/lib/tls/fast-mult:/usr/lib/tls/half:/usr/lib/tls:/usr/lib/v6l/fast-mult/half:/usr/lib/v6l/fast-mult:/usr/lib/v6l/half:/usr/lib/v6l:/usr/lib/fast-mult/half:/usr/lib/fast-mult:/usr/lib/half:/usr/lib (system search path)
500: trying file=/lib/libc.so.6

500: calling init: /lib/libpthread.so.0

从上面的结果可以看出，为了寻找 libpthread.so.0，系统搜索了大量的无关路径=/lib/tls/v6l/fast-mult/half:/lib/tls/v6l/fast-mult:/lib/tls/v6l/half:/lib/tls/v6l:/lib/tls/fast-mult/half:/lib/tls/fast-mult:/lib/tls/half:/lib/tls:/lib/v6l/fast-mult/half:/lib/v6l/fast-mult:/lib/v6l/half:/lib/v6l:/lib/fast-mult/half:/lib/fast-mult:/lib/half:/lib:/usr/lib/tls/v6l/fast-mult/half:/usr/lib/tls/v6l/fast-mult:/usr/lib/tls/v6l/half:/usr/lib/tls/v6l:/usr/lib/tls/fast-mult/half:/usr/lib/tls/fast-mult:/usr/lib/tls/half:/usr/lib/tls:/usr/lib/v6l/fast-mult/half:/usr/lib/v6l/fast-mult:/usr/lib/v6l/half:/usr/lib/v6l:/usr/lib/fast-mult/half:/usr/lib/fast-mult:/usr/lib/half:/usr/lib。这些路径就来自于 Linux 的 HWCAP 机制。

在各个不同的系统种，系统的硬件功能是不一致的，有的系统支持浮点运算，有的则不支持。这样，由于系统硬件功能的不同，软件需要加载不同的共享库，glibc 的 HWCAP 就是为了这个功能所设计的。
上面的 tls,v6l half fast-muli 则代表着不同的硬件功能特性。

这个功能虽然带来了灵活，但也导致我们的进程在加载动态库时，搜索了很多无效的路径，浪费了宝贵的时间。
我们可以通过环境变量 LD_HWCAP_MASK 来屏蔽一些硬件功能选项，从而减少搜索路径。
export LD_HWCAP_MASK=0x00000000
LD_DEBUG=libs ./hello
508: find library=libpthread.so.0; searching
508: search cache=/etc/ld.so.cache
508: search path=/lib/tls/v6l:/lib/tls:/lib/v6l:/lib:/usr/lib/tls/v6l:/usr/lib/tls:/usr/lib/v6l:/usr/lib (system search path)
508: trying file=/lib/tls/v6l/libpthread.so.0
508: trying file=/lib/tls/libpthread.so.0
508: trying file=/lib/v6l/libpthread.so.0
508: trying file=/lib/libpthread.so.0
508: find library=libc.so.6; searching
508: search cache=/etc/ld.so.cache
508: search path=/lib:/usr/lib/tls/v6l:/usr/lib/tls:/usr/lib/v6l:/usr/lib (system search path)
508: trying file=/lib/libc.so.6
508: calling init: /lib/libpthread.so.0
508: calling init: /lib/libc.so.6
508: initialize program: ./hello
508: transferring control: ./hello

上面描述了很多关于搜索动态库路径的机制，其首要的目标就是在进程加载共享库时，能最快的找到动态库，搜索短时间。
可以使用的方法有：
1、 设置 LD_HWCAP_MASK，禁掉一些不用的硬件特性。
2、 将所有的动态库都放在一个目录下，并将该目录放在 LD_LIBRARY_PATH 的开始。

3、 不能放在一个目录的，在进程中加入-rpath 选项，指定搜索路径。

将所有的动态库放在一个目录下，并且在 LD_LIBRARY_PATH 的第一个搜索路径指向它，从而减少搜索的次数，从而降低搜索动态库的时间。

3.6.4. 动态库的层次

由于库的依赖关系，我们可以画出库之间的一个层次关系，它可以是一棵树，也可能是一个线型结构。从一个库到最底层库之间的最长路径，我们称之为库的层次。
有一种说法，降低库的层次，使库的层次扁平话，可以降低加载动态库的时间，他们这么做了，也的确收到了效果，可为什么呢？

我为此做了一个试验：
Test1：
我做了一个可执行文件，hello 和 82 个动态库，这 82 个动态库之间不依赖。可以说从 hello，到最底层的 libc.so 的层次为 2.
我使用 strace –tt 来获取加载库的时间：0.645112

Test2:
我做了一个可执行文件，hello 和 82 个动态库，这 82 个库是相互依赖的，lib1 依赖于 lib2,lib2 依赖于 lib3….
lib81 依赖 lib82。
可以说从 hello 到最底层的 libc.so 的层次为 83.
使用 strace –tt 来获取加载库的时间：0.650055

从结果可以看出，Test2 的确比 Test1 的时间要少，但仅少 0.004943，几乎可以忽略不计。

那为什么降低动态库的层次，又的确缩短了加载动态库的时间呢？
在我们降低库的层次时，经常采用的手段是合并不必要的库，使用 dlopen 来动态加载库，这样的做的效果会减少进程加载库的数量，这才是缩短加载库时间的根本，而不是库的依赖形态。

3.6.5. 动态库的初始化

在加载器完成了对动态库的内存映射之后，需要运行动态库的一些初始化函数，来完成设置动态库的一些基本环境。
这里主要包含两个部分：

1、 动态库的构造和析构函数机制。
2、 动态库的全局变量初始化工作。

3.6.5.1. 动态库的构造和析构函数机制

在 Linux 中，为我们提供了一个机制，在加载和卸载动态库时，可以编写程序，做一些相应的动作。

```
void __attribute__ ((constructor)) my_init(void);
void __attribute__ ((destructor)) my_fini(void);
```

在编译共享库时，不能使用"-nonstartfiles"或"-nostdlib"选项，否则，构建与析构函数将不能正常执行(除非你采取一定措施)。

注意，构造函数其参数必须为空，返回值也必须为空。

在这里我举个例子：

动态库文件

a.c

```
void __attribute__ ((constructor)) my_init(void)
{
    printf("init library\n");
}
```

```
gcc -fPIC -shared b2.c -o liba.so
```

主程序

Hello.c

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

int main()
{
    pause();
    return 0;
}
```

编译:

```
gcc –L./ -la hello.c –o hello
```

运行 hello：

```
# ./hello
init library
```

也就是说，在运行 hello 时，加载完 liba.so 后，自动运行 liba.so 的初始化函数。

3.6.5.2. 全局变量初始化

这里我先举个例子：

liba.so

```
int reti()
{
      printf("reti\n");
      return 10;
}
int g1=reti();
```

在这里，g1 是个全局变量。
首先我们使用 gcc 对其进行编译：
b2.c:23: error: initializer element is not constant

我们使用 g++对齐进行编译：
成功！

运行 hello
./hello
reti

这说明，进程在加载 liba.so 后，为了初始化全局变量 g1，其会运行 reti 来初始化 g1。

我们再来看一个例子：
liba.so

```
class c1
{
public:
    c1();
    int c1_i1;
};

c1::c1()
{
    printf("library 0\n");
};

c1 g1;
```

在动态库 liba.so 中，我们声明了一个类型为 c1 的全局变量 g1。
运行 hello
./hello
library 0

这说明，进程在加载 liba.so 后，为了初始化全局变量 g1，将会运行 c1 的构造函数。

而这些初始化函数的运行，必然会导致加载动态库时间缓慢，甚至在程序还没有做任何事情时，就已经占用了大量的物理内存。
在系统优化过程中，我们曾经在一个进程的 main 函数中第一条指令增加了一个 pause 语句，我们很惊奇的发现，在进程还没有做任何事情的时候，仅仅是加载动态库，就使用了 800K 的物理内存。

3.6.6. Prelink

动态库的确给我们的程序带来良好的扩充性和减少内存的使用量，但这是有代价的。

```
#include<stdio.h>

int main()
{
    printf(“hello\n”);
    return 0;
}
```

我们知道 printf 是在 libc 中定义的，比如说我们不使用动态库，那是将 libc 静态联编到进程中的话，那么 printf 函数的地址在运行之前就已知的了，这里可以很简单的一句地址转移，就可以完成了。
可是如果采用动态库的话，在程序编译阶段，我们是无法得知 printf 的函数地址，因为动态库的加载的内存地址是随机的。那么对于动态库的情况，针对 printf 是如何寻址的呢？

在程序启动时，当调用 printf 的时候，程序会将处理权交给 loader，由其负责在进程以及其链接的动态库中查找 printf 的函数地址。由于 loader 不知道 printf 是在哪个动态库，所以它将在整个的进程和动态库的范围内查找，更糟糕的是在 C++程序中，符号的命名是类名+函数名，这导致在做字符串比较时，往往直到字符串的结尾才能获得结果。

这就导致了，在进程启动过程中，符号查找往往占据了大部分时间。在 Linux 的 KDE 进程启动过程中，符号查表竟占据了进程启动 80%的时间。

这里我们可以使用一个工具，来查看进程在启动过程中，符号查找的情况。

有没有什么办法来改进？
如果进程在运行前，我们就能够获知动态库的加载地址，那么我们函数调用的地址就应该是已知的，我们就可以通过修改执行代码，来避免符号的查找，从而节省进程的启动时间。

实际上 Prelink 也正是这么做的。Prelink 最早是在 redhat 中引用的，用来加速 KDE 的启动速度。那时 prelink 做为系统的一个进程，不定期的启动，对系统中的进程和动态库进行优化，这对于系统中进程和动态库不怎么变化非常实用。

而对于嵌入式 Linux 系统来讲，我们可以在完成 build 所有文件后，运行 prelink 来加速程序

运行，再将修改后的进程做成 img，烧到设备上，非常实用。

Prelink 的使用：
Prelink 使用方式
http://www.linuxcommand.org/man_pages/prelink8.html

我们可以用下列指令来"预先连结"所有列在 /etc/prelink.conf 中的目录里的执行档.

代码 3.1: 预先连结清单中的档案
prelink -afmR
警告： 有人发现如果你在磁碟空间吃紧的时候"预先连结"系统上所有执行档, 你的执行档有可能会被截断, 这样会弄爆你的系统. 你可以用 file 或 readelf 来检查执行档的状态. 或者每次在进行"预先连结"前先用 df -h 检查硬碟的剩余空间.

每个选项的解说:
-a "All": 对所有执行档进行"预先连结".
-f 强制 prelink 重新"预先连结"已经做过"预先连结"的执行档. 加上这个选项是因为 prelink 在看见做过"预先连结"的执行档的时候会 中止执行, 即使相依的函式库有更动过.
-m 节省虚拟定址分配. 如果你有一卡车的函式库要"预先连结"就会需要这个选项. (译注: 这里的原文 virtual memory space 是有问题的, 应该是 virtual address space 比较正确.)
-R Random -- 用乱数进行定址分配, 这样可以增进安全性对缓冲区溢出(buffer overflow) 攻击的抵抗能力.

在做 prlink 时，你需要为其指定需要做 prilink 的进程和动态库的目录，prlink 需要做以下几件事情：
1、 分析所有的进程和动态库，为每个动态库指定一块唯一的内存地址。
2、 分析进程和动态库中，所有需要重定位的函数，全局变量等，使用 loader 进程符号查找，对齐地址进行解析。
3、 修改进程和动态库的二进制文件。

这里可能有人会问：
如果我们的系统中，动态库的数据非常多，为每个动态库指定一段唯一的内存地址，那地址段够吗？
1、首先我们来看看到底有多少内存地址可用？
我们来查看一下 maps
00008000-00009000 r-xp 00000000 1f:12 288 /mnt/msc_int0/hello

00010000-00011000 rw-p 00000000 1f:12 288 /mnt/msc_int0/hello

00011000-00032000 rwxp 00011000 00:00 0

40000000-40002000 rw-p 40000000 00:00 0

41000000-41017000 r-xp 00000000 1f:0d 817360 /lib/ld-2.3.3.so

4101e000-41020000 rw-p 00016000 1f:0d 817360 /lib/ld-2.3.3.so

41028000-41120000 r-xp 00000000 1f:0d 817593 /lib/libc-2.3.3.so

41120000-41128000 ---p 000f8000 1f:0d 817593 /lib/libc-2.3.3.so

41128000-41129000 r--p 000f8000 1f:0d 817593 /lib/libc-2.3.3.so

41129000-4112c000 rw-p 000f9000 1f:0d 817593 /lib/libc-2.3.3.so

4112c000-4112e000 rw-p 4112c000 00:00 0

befeb000-bf000000 rwxp befeb000 00:00 0

3G 以上为操作系统使用，在 00000000~40000000 归进程的代码段、数据段和堆段使用。
从 3G 往下归栈段使用。
基本上我们可以认为从 1G~3G 的地址空间可以用来指定动态库的加载地址。地址空间还是很丰富的。

2、Prelink 关于这个问题，做了两个约定。
总是一同出现的动态库，其动态库的加载地址一定不能重叠；
总是不同时出现的动态库，其动态库的加载地址可以重叠。

有了这两个约定之后，基本上就能保证，为每个动态库指定加载地址，从而在运行前就能获知函数等的地址。

问题二：如何指定加载地址。
在 prelink 之前
readelf -l libezxaddrbk.so
Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x000000	0x00000000	0x00000000	0xdbccc	0xdbccc	R E	0x8000
LOAD	0x0dc000	0x000e4000	0x000e4000	0x083a8	0x087f8	RW	0x8000
DYNAMIC	0x0dc0ec	0x000e40ec	0x000e40ec	0x001b0	0x001b0	RW	0x4
NOTE	0x0dbc58	0x000dbc58	0x000dbc58	0x00074	0x00074	R	0x4

在 prelink 之后
Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x000000	0x430d8000	0x430d8000	0xdbccc	0xdbccc	R E	0x8000
LOAD	0x0dc000	0x431bc000	0x431bc000	0x083a8	0x087f8	RW	0x8000
DYNAMIC	0x0dc0ec	0x431bc0ec	0x431bc0ec	0x001b0	0x001b0	RW	0x4
NOTE	0x0dbc58	0x431b3c58	0x431b3c58	0x00074	0x00074	R	0x4

我们可以看到，prelink 会修改动态库的 Program Headers 段。

我们来查看一下 Prelink 之后的效果。
Prelink 之前
LD_DEBUG=statistics ./addrbk

```
533:                     number of relocations: 9368
533:                     number of relocations from cache: 26700
533:                     number of relative relocations: 0
```

Prelink 之后

516: number of relocations: 0
516: number of relocations from cache: 588
516: number of relative relocations: 0

应该说效果还是十分显著的。

使用 Prelink：

1、 在编译你的动态库时，加上-fPIC 选项，生成位置无关代码。
2、 Glibc 的版本要高于 2.3.1
3、 Prelink 对于 dlopen 打开的动态库没有效果。

Prelink.conf

```
# This config file contains a list of directories both with binaries
# and libraries prelink should consider by default.
# If a directory name is prefixed with `-l ', the directory hierarchy
# will be walked as long as filesystem boundaries are not crossed.
# If a directory name is prefixed with `-h ', symbolic links in a
# directory hierarchy are followed.

# System libraries
-l /lib
-l /usr/lib
-l /usr/local/lib

# System binaries
-l /bin
-l /usr/bin
-l /usr/local/bin
-l /sbin
-l /usr/sbin
-l /usr/local/sbin

# Qtopia
-l /opt/Qtopia/lib
-l /opt/Qtopia/bin
-l /opt/Qtopia/plugins
-l /opt/Qtopia/qt_plugins
```

3.6.6.1. Prelink 会带来内存使用上的增长

我们先举个例子：

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

```
#include <stdlib.h>

int main()
{
    printf("Hello,world\n");
    malloc(10);
    return 0;
}
```

在 Prelink 之前，我们查看一下生成 ELF 文件的情况。

```
Readelf –S ./hello
Section Headers:
[Nr] Name Type Addr Off Size ES Flg Lk Inf Al
[ 0] NULL 00000000 000000 000000 00 0 0 0
[ 1] .interp PROGBITS 08048114 000114 000013 00 A 0 0 1
[ 2] .note.ABI-tag NOTE 08048128 000128 000020 00 A 0 0 4
[ 3] .hash HASH 08048148 000148 000024 04 A 4 0 4
[ 4] .dynsym DYNSYM 0804816c 00016c 000040 10 A 5 1 4
[ 5] .dynstr STRTAB 080481ac 0001ac 000045 00 A 0 0 1
[ 6] .gnu.version VERSYM 080481f2 0001f2 000008 02 A 4 0 2
[ 7] .gnu.version_r VERNEED 080481fc 0001fc 000020 00 A 5 1 4
[ 8] .rel.dyn REL 0804821c 00021c 000008 08 A 4 0 4
[ 9] .rel.plt REL 08048224 000224 000008 08 A 4 b 4
[10] .init PROGBITS 0804822c 00022c 000017 00 AX 0 0 4
 ...
[22] .bss NOBITS 080494f8 0004f8 000004 00 WA 0 0 4
[23] .comment PROGBITS 00000000 0004f8 000132 00 0 0 1
[24] .shstrtab STRTAB 00000000 00062a 0000be 00 0 0 1
```

我们再来看一下prelink完之后的情况：

```
[Nr] Name Type Addr Off Size ES Flg Lk Inf Al
[ 0] NULL 00000000 000000 000000 00 0 0 0
[ 1] .interp PROGBITS 08048114 000114 000013 00 A 0 0 1
[ 2] .note.ABI-tag NOTE 08048128 000128 000020 00 A 0 0 4
[ 3] .hash HASH 08048148 000148 000024 04 A 4 0 4
[ 4] .dynsym DYNSYM 0804816c 00016c 000040 10 A 23 1 4
[ 5] .gnu.liblist GNU_LIBLIST 080481ac 0001ac 000028 14 A 23 0 4
[ 6] .gnu.version VERSYM 080481f2 0001f2 000008 02 A 4 0 2
[ 7] .gnu.version_r VERNEED 080481fc 0001fc 000020 00 A 23 1 4
[ 8] .rel.dyn REL 0804821c 00021c 000008 08 A 4 0 4
[ 9] .rel.plt REL 08048224 000224 000008 08 A 4 b 4
[10] .init PROGBITS 0804822c 00022c 000017 00 AX 0 0 4
 ...
[22] .bss PROGBITS 080494f8 0004f8 000004 00 WA 0 0 4
```

```
[23] .dynstr STRTAB 080494fc 0004fc 000058 00 A 0 0 1
[24] .gnu.conflict RELA 08049554 000554 0000c0 0c A 4 0 4
[25] .comment PROGBITS 00000000 000614 000132 00 0 0 1
[26] .gnu.prelink_undo PROGBITS 00000000 000748 0004d4 01 0 0 4
[27] .shstrtab STRTAB 00000000 000c1c 0000eb 00 0 0 1
```

从prelink前后的section变化情况，我们可以发现：

1、 增加了3个section：

```
[ 5] .gnu.liblist GNU_LIBLIST 080481ac 0001ac 000028 14 A 23 0 4
[24] .gnu.conflict RELA 08049554 000554 0000c0 0c A 4 0 4
[26] .gnu.prelink_undo PROGBITS 00000000 000748 0004d4 01 0 0 4
[23] .dynstr STRTAB 080494fc 0004fc 000058 00 A 0 0 1
```

2、 另外.dynstr节也从0045 增加到0058。

这里有一个需要大家注意的地方，有所变化的这4个section， .gnu.liblist .gnu.conflict .gnu.prelink_undo和.dynstr其属性都是A，分配内存只读。意思是说，对于动态链接库来讲，这些节可以位于代码段，在系统的所有进程之间共享。

但是由于要插入3个节，和扩大.dynstr，prelink需要重新来安排各个section的位置，这里它有一套自己的规则：

1、 prelink首先在各个section之间查看，看看是否有空隙，如果有，则把这几个节插进去。（注这几个节并不要求放在一起）。
2、 查看SHT_NOBITS节是否能够与前面的SHT_PROGBITS节放到一个页面中，并且在这个页的SHT_NOBITS节之后，还有些空间。Prelink将把SHT_NOBITS节转换成SHT_PROGBITS，并填充为0，然后把新的section附加在后面。
3、 在IA-32体系架构的可执行文件由于历史原因，其基地址为0x8048000，但这不是必须，prelink可以降低基地址的空间，来为新加入的section腾出空间。
4、 如果以上方法都不满足的话，prelink会新创建一个PT_LOAD的段，来容纳这几个节。由于这个节后面紧跟着堆段（因为Linux有个约定，堆段紧挨着最后一个PT_LOAD段），所以其权限属性不得不可读写。

如果Prelink将新加入的section放在了SHT_NOBITS，或新增了一个段来容纳它的话，那么对动态库来讲，意味着原本可以在系统内所有进程之间共享的几个section，不得不为每个进程复制一份，从而导致内存大量的浪费。

因此，我们要想办法，使Prelink新加入的节放在代码段，从而达到共享的目的。我们可以利用规则1，来人为的在section之间创建空隙。

我们要利用loader的link脚本，来完成这个事情。

1、 使用loader获得缺省的link 脚本。

```
ld –verbose > test.lds
```

我们可以查看一下这个缺省的link脚本。

```
GNU ld version 2.15.94 20041215
  Supported emulations:
   armelf_linux_eabi
using internal linker script:
==================================================
/* Script for -z combreloc: combine and sort reloc sections */
OUTPUT_FORMAT("elf32-littlearm", "elf32-bigarm",
              "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SEARCH_DIR("=/usr/local/lib"); SEARCH_DIR("=/lib"); SEARCH_DIR("=/usr/lib");
/* Do we need any of these for elf?
   __DYNAMIC = 0;    */
SECTIONS
{
  /* Read-only sections, merged into text segment: */
  PROVIDE (__executable_start = 0x00008000); . = 0x00008000 + SIZEOF_HEADERS;
  .interp         : { *(.interp) }
  .hash           : { *(.hash) }
  .dynsym         : { *(.dynsym) }
  .dynstr         : { *(.dynstr) }
  .gnu.version    : { *(.gnu.version) }
  .gnu.version_d  : { *(.gnu.version_d) }
  .gnu.version_r  : { *(.gnu.version_r) }
  .rel.dyn        :
    {
      *(.rel.init)
      *(.rel.text .rel.text.* .rel.gnu.linkonce.t.*)
      *(.rel.fini)
      *(.rel.rodata .rel.rodata.* .rel.gnu.linkonce.r.*)
      *(.rel.data.rel.ro*)
      *(.rel.data .rel.data.* .rel.gnu.linkonce.d.*)
      *(.rel.tdata .rel.tdata.* .rel.gnu.linkonce.td.*)
      *(.rel.tbss .rel.tbss.* .rel.gnu.linkonce.tb.*)
      *(.rel.ctors)
      *(.rel.dtors)
      *(.rel.got)
      *(.rel.bss .rel.bss.* .rel.gnu.linkonce.b.*)
    }
    ……

      .debug_weaknames 0 : { *(.debug_weaknames) }
      .debug_funcnames 0 : { *(.debug_funcnames) }
      .debug_typenames 0 : { *(.debug_typenames) }
```

```
        .debug_varnames   0 : { *(.debug_varnames) }
        .note.gnu.arm.ident 0 : { KEEP (*(.note.gnu.arm.ident)) }
        /DISCARD/ : { *(.note.GNU-stack) }
      }
```

==========

2、我们需要把这个缺省的link脚本，首尾的一些注释去掉，才能使用。

```
GNU ld version 2.15.94 20041215
  Supported emulations:
   armelf_linux_eabi
using internal linker script:
==================================================
/* Script for -z combreloc: combine and sort reloc sections */
OUTPUT_FORMAT("elf32-littlearm", "elf32-bigarm",
              "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
……..

   .debug_weaknames 0 : { *(.debug_weaknames) }
   .debug_funcnames 0 : { *(.debug_funcnames) }
   .debug_typenames 0 : { *(.debug_typenames) }
   .debug_varnames   0 : { *(.debug_varnames) }
   .note.gnu.arm.ident 0 : { KEEP (*(.note.gnu.arm.ident)) }
   /DISCARD/ : { *(.note.GNU-stack) }
  }
```

==========

3、下一步，我们需要在 .gnu.version_r 和 .rel.dyn 之间插入空隙。

```
  .gnu.version_d   : { *(.gnu.version_d) }
  .gnu.version_r   : { *(.gnu.version_r) }
  . +=512;
  .rel.dyn        :
    {
      *(.rel.init)
      *(.rel.text .rel.text.* .rel.gnu.linkonce.t.*)
      *(.rel.fini)
      *(.rel.rodata .rel.rodata.* .rel.gnu.linkonce.r.*)
      *(.rel.data.rel.ro*)
      *(.rel.data .rel.data.* .rel.gnu.linkonce.d.*)
      *(.rel.tdata .rel.tdata.* .rel.gnu.linkonce.td.*)
      *(.rel.tbss .rel.tbss.* .rel.gnu.linkonce.tb.*)
      *(.rel.ctors)
```

```
        *(.rel.dtors)
        *(.rel.got)
        *(.rel.bss .rel.bss.* .rel.gnu.linkonce.b.*)
    }
```

在这里，我们增加512个字节。

3、 编译，链接，生成ELF文件。

gcc –o hello hello.c –wl , -T , test.lds。

这里-T是用来指定link脚本。

我们可以使用readelf –S hello来验证结果。

```
[Nr] Name Type Addr Off Size ES Flg Lk Inf Al
[ 0] NULL 00000000 000000 000000 00 0 0 0
[ 1] .interp PROGBITS 08048114 000114 000013 00 A 0 0 1
[ 2] .note.ABI-tag NOTE 08048128 000128 000020 00 A 0 0 4
[ 3] .hash HASH 08048148 000148 000024 04 A 4 0 4
[ 4] .dynsym DYNSYM 0804816c 00016c 000040 10 A 5 1 4
[ 5] .dynstr STRTAB 080481ac 0001ac 000045 00 A 0 0 1
[ 6] .gnu.version VERSYM 080481f2 0001f2 000008 02 A 4 0 2
[ 7] .gnu.version_r VERNEED 080481fc 0001fc 000020 00 A 5 1 4
[ 8] .rel.dyn REL 0804841c 00041c 000008 08 A 4 0 4
[ 9] .rel.plt REL 08048424 000424 000008 08 A 4 b 4
[10] .init PROGBITS 0804842c 00042c 000017 00 AX 0 0 4
 ...
[22] .bss NOBITS 080496f8 0006f8 000004 00 WA 0 0 4
[23] .comment PROGBITS 00000000 0006f8 000132 00 0 0 1
[24] .shstrtab STRTAB 00000000 00082a 0000be 00 0 0 1
```

在Prelink之后，我们可以看到：

```
[Nr] Name Type Addr Off Size ES Flg Lk Inf Al
[ 0] NULL 00000000 000000 000000 00 0 0 0
[ 1] .interp PROGBITS 08048114 000114 000013 00 A 0 0 1
[ 2] .note.ABI-tag NOTE 08048128 000128 000020 00 A 0 0 4
[ 3] .hash HASH 08048148 000148 000024 04 A 4 0 4
[ 4] .dynsym DYNSYM 0804816c 00016c 000040 10 A 8 1 4
[ 5] .gnu.liblist GNU_LIBLIST 080481ac 0001ac 000028 14 A 8 0 4
[ 6] .gnu.version VERSYM 080481f2 0001f2 000008 02 A 4 0 2
[ 7] .gnu.version_r VERNEED 080481fc 0001fc 000020 00 A 8 1 4
[ 8] .dynstr STRTAB 0804821c 00021c 000058 00 A 0 0 1
[ 9] .gnu.conflict RELA 08048274 000274 0000c0 0c A 4 0 4
[10] .rel.dyn REL 0804841c 00041c 000008 08 A 4 0 4
[11] .rel.plt REL 08048424 000424 000008 08 A 4 d 4
[12] .init PROGBITS 0804842c 00042c 000017 00 AX 0 0 4
 ...
```

```
[24] .bss NOBITS 080496f8 0006f8 000004 00 WA 0 0 4
[25] .comment PROGBITS 00000000 0006f8 000132 00 0 0 1
[26] .gnu.prelink_undo PROGBITS 00000000 00082c 0004d4 01 0 0 4
[27] .shstrtab STRTAB 00000000 000d00 0000eb 00 0 0 1
```

从而我们将原来移到数据段的.gnu.conflict .gnu.prelink_undo .dynstr 都转移到了代码段，从而达到节省内存的目的。

3.6.7. 提高进程启动速度

在我们做完上面所有的优化后，你也许依然会发现进程的启动速度还是无法满足用户需要，多么令人沮丧的一件事情。
进程就是需要那么多库，而加载库就是需要花费那么多时间，难道需要我们去修改加载器吗？就是因为进程启动速度的限制，很多进程不得不改为了 daemon 进程，常驻内存，而我们也知道，daemon 进程的增多，会给系统带来很多的问题，内存的占用，和 CPU 调度的增加等等。

我就遇到了这样的一个问题：
通过测试，进程加载动态库的时间，花费了 300ms，而要求进程完成初始化，显示用户界面的时间为 400ms，这样留给用户逻辑的代码就只有 100ms，这简直是一个不可完成的任务。

我们难道就没有其他方法了吗？

别灰心，还是有办法的：

1、 将进程改为线程
2、 Preload 进程

3.6.7.1. 进程改为线程

我们知道，在 Linux 中线程是没有自己的内存空间，它与别的线程一起共享进程的内存空间。那么在启动线程的时候，我们就不必再去加载动态库等费时的操作。

从这个思路上来看，我们可以把原来的进程分割为两个部分：

1、常驻内存部分，其主要是负责加载进程所需要的动态库，等待用户信号，创建和销毁用户逻辑线程；

2、完成用户逻辑部分，按用户需求完成用户逻辑。

这样我们就可以所读进程的响应时间，来满足用户需求。

我们还可以再引申一下，将原来的多个 daemon 进程的常驻内存部分进行合并，根据用户逻辑的需求，创建不同的线程。

需要注意的两个问题：

1、 因为进程不会退出，所以动态库的数据段将会保留以前的变化，全局变量和静态数据不会随着用户线程的销毁，而重置。
2、 如果多个业务逻辑的线程，共用一个守护进程的话，注意多个线程可能公用一个动态库的全局变量，而产生冲突。
3、 进程的栈段缺省大小为 8M，而线程的栈段缺省为 2M，有可能会因为将进程改为线程，而带来栈溢出。这个问题，你可以通过手工设置线程栈空间大小来修改。

好处：

1、 缩短进程的响应时间
2、 多个业务逻辑共享动态库，避免了系统为每个业务逻辑创建动态库的数据段，从而节省了大量的内存。

缺点：

1、 由原来的进程改为线程，工作量比较大，代码修改上存在一定的风险。
2、 多个业务逻辑线程之间共享动态库，有可能会带来全局变量的冲突。
3、 由于还是存在 daemon 进程部分，所以其堆所栈内存不会被释放，多个业务逻辑线程所存在的内存泄漏会纠缠在一起，从而是问题更加复杂。

3.6.7.2. Preload 进程

实际问题很简单，进程响应时间=进程加载动态库+用户逻辑部分。那么我们可不可以将用户进程启动分为两个部分呢？

1、 我们在进程的 main 函数中，插入一行语句

pause();

这样，当进程启动时，加载完动态库后，就会停在这里，不会运行用户逻辑。

2、当我们需要响应用户时，向该进程发送一个信号，这样用户就会继续前进，处理用户逻辑，这样我们就节省了进程加载动态库的过程。

这里我们，需要注册一个信号处理函数。

```
void sigCont(int unused)
{
    return ;
}
int main(int argc, char** argv)
{
   signal(SIGCONT,sigCont);
   pause();
```

3、当用户逻辑执行完成后，就退出进程，同时在启动该进程，这时进程会在加载完动态库后，停留在那里。

好处：

1、 该方法对于目前的进程来讲，修改起来十分容易。
2、 由于没有了 daemon 部分，堆段在进程退出后，所以进程的内存泄漏问题影响不大。

缺点：
1、 由于在进程退出后，需要重新启动进程，会带来调度上复杂性。
2、 在进程等待的时候，其加载的动态库还是要占用一部分内存。

在这里我还是推荐这种方法，修改起来简单，而且不容易出问题，即使加载动态库会占用一些内存，但那与堆段的内存泄漏和内存碎片比起来，还是小的很多。

3.7. 优化思路

我们想我们的程序跑的飞开，可当我们面对大量代码的时候，却无从下手，怎么能让程序跑的更快呢？

首先我们要想到的是著名的 8/2 法则，往往 20%的代码占用了 80%的运行时间，我们优化的重点便在于这 20%的代码。

第一步：找到性能瓶颈代码
在 Linux 下，也提供了很多类似的工具，象 gprof 和 opofile。这两个工具，我们都将在后面提到。

第二步：代码优化

11.2 提高程序的效率
程序的时间效率是指运行速度，空间效率是指程序占用内存或者外存的状况。

全局效率是指站在整个系统的角度上考虑的效率，局部效率是指站在模块或函数角度上考虑的效率。

l 【规则 11-2-1】不要一味地追求程序的效率，应当在满足正确性、可靠性、健壮性、可读性等质量因素的前提下，设法提高程序的效率。

l 【规则 11-2-2】以提高程序的全局效率为主，提高局部效率为辅。

l 【规则 11-2-3】在优化程序的效率时，应当先找出限制效率的“瓶颈”，不要在无关紧要之处优化。

l 【规则 11-2-4】先优化数据结构和算法，再优化执行代码。

l 【规则 11-2-5】有时候时间效率和空间效率可能对立，此时应当分析那个更重要，作出适当的折衷。例如多花费一些内存来提高性能。

1 【规则 11-2-6】不要追求紧凑的代码，因为紧凑的代码并不能产生高效的机器码。

11.3 一些有益的建议
2 【建议 11-3-1】当心那些视觉上不易分辨的操作符发生书写错误。

我们经常会把“＝＝”误写成“＝”，象“||”、“&&”、“<=”、“>=”这类符号也很容易发生“丢 1”失误。然而编译器却不一定能自动指出这类错误。

2 【建议 11-3-2】变量（指针、数组）被创建之后应当及时把它们初始化，以防止把未被初始化的变量当成右值使用。

2 【建议 11-3-3】当心变量的初值、缺省值错误，或者精度不够。

2 【建议 11-3-4】当心数据类型转换发生错误。尽量使用显式的数据类型转换（让人们知道发生了什么事），避免让编译器轻悄悄地进行隐式的数据类型转换。

2 【建议 11-3-5】当心变量发生上溢或下溢，数组的下标越界。

2 【建议 11-3-6】当心忘记编写错误处理程序，当心错误处理程序本身有误。

2 【建议 11-3-7】当心文件 I/O 有错误。

2 【建议 11-3-8】避免编写技巧性很高代码。

2 【建议 11-3-9】不要设计面面俱到、非常灵活的数据结构。

2 【建议 11-3-10】如果原有的代码质量比较好，尽量复用它。但是不要修补很差劲的代码，应当重新编写。

2 【建议 11-3-11】尽量使用标准库函数，不要“发明”已经存在的库函数。

2 【建议 11-3-12】尽量不要使用与具体硬件或软件环境关系密切的变量。

2 【建议 11-3-13】把编译器的选择项设置为最严格状态。

2 【建议 11-3-14】如果可能的话，使用 PC-Lint、LogiScope 等工具进行代码审查。

3.8. 查找性能瓶颈

3.8.1. gprof

通过在编译和链接你的程序的时候（使用 -pg 编译和链接选项），gcc 在你应用程序的每个

函数中都加入了一个名为 mcount (or “_mcount” , or “__mcount”，依赖于编译器或操作系统)的函数，也就是说你的应用程序里的每一个函数都会调用 mcount, 而 mcount 会在内存中保存一张函数调用图，并通过函数调用堆栈的形式查找子函数和父函数的地址。这张调用图也保存了所有与函数相关的调用时间，调用次数等等的所有信息。

3.8.1.1. 基本用法：

1、使用 -pg 编译和链接你的应用程序，如果要得到带注释的源码清单，则需要增加-g 选项。
2、执行你的应用程序，会在当前目录下产生 gmon.out 文件。
3、使用 gprof 程序分析 gmon.out 文件，需要把它和产生它的应用程序关联起来：

gprof hello gmon.out –p 得到每个函数占用的执行时间。

gprof hello gmon.out –q 得到 call graph，包含了每个函数的调用关系，调用次数，执行时间等信息。

gprof hello gmon.out –A 得到一个带注释的源代码清单，它会注释源码，指出每个函数的执行次数。这需要在编译的时候增加-g 选项。

范例：

```
#include <stdio.h>

void funca()
{
    int i=0;
    int n=0;
    for(i=0;i<10000000;i++)
    {
        n++;
        n--;
    }
}

void funcb()
{
    int i=0;
    int n=0;
    for(i=0;i<10000000;i++)
    {
        n++;
        n--;
    }
}
int main(int argc, char* argv[])
{
```

```
    int i=0;
    for(i=0;i<10;i++)
    {
        funca();
    }
    funcb();
    return 0;
}
```

编译： gcc hello.c -pg -o hello
运行： ./hello
产生 gmon.out 文件
使用 gprof 进行解析：
gprof hello gmon.out –p
Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
90.92	5.71	5.71	10	571.00	571.00	funca
9.08	6.28	0.57	1	570.00	570.00	funcb

% time　the percentage of the total running time of the 函数使用时间占所有时间的百分比。

cumulative seconds　a running sum of the number of seconds accounted for by this function and those listed above it. 函数和上列函数累计执行的时间。

self seconds　the number of seconds accounted for by this function alone. This is the major sort for this listing. 函数本身所执行的时间。

calls　the number of times this function was invoked, if this function is profiled, else blank. 函数被调用的次数

self ms/call　the average number of milliseconds spent in this function per call, if this function is profiled, else blank. 每一次调用花费在函数的时间 microseconds。

total ms/call　the average number of milliseconds spent in this function and its descendents per call, if this function is profiled, else blank. 每一次调用，花费在函数及其衍生函数的平均时间 microseconds。

name the name of the function. This is the minor sort
for this listing. The index shows the location of
the function in the gprof listing. If the index is
in parenthesis it shows where it would appear in
the gprof listing if it were to be printed.
函数名

gprof hello gmon.out –q
Call graph (explanation follows)

granularity: each sample hit covers 4 byte(s) for 0.16% of 6.28 seconds

index	% time	self	children	called	name
					<spontaneous>
[1]	100.0	0.00	6.28		main [1]
		5.71	0.00	10/10	funca [2]
		0.57	0.00	1/1	funcb [3]

		5.71	0.00	10/10	main [1]
[2]	90.9	5.71	0.00	10	funca [2]

		0.57	0.00	1/1	main [1]
[3]	9.1	0.57	0.00	1	funcb [3]

很简单，很方便不是吗，它可以把代码时间统计分析精确到函数。
gprof 的一个大的缺陷是：它只能分析应用程序在运行 过程中所消耗掉的用户 时间。通常来说，应用程序在运行时既要花费一些时间来运行用户代码，也要花费一些时间来运行“系统代码”，例如内核系统调用。

3.8.1.2. gprof 与动态库

创建动态库 liba.c

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

void funca()
{
    int i=0;
    int n=0;
    for(i=0;i<10000000;i++)
    {
        n++;
        n--;
```

```
    }
}

void funcb()
{
    int i=0;
    int n=0;
    for(i=0;i<10000000;i++)
    {
        n++;
        n--;
    }
}
```

编译：　　gcc -fPIC -shared -pg a.c -o liba.so

创建进程 hello

```
#include <stdio.h>
void funca();
void funcb();

int main(int argc, char* argv[])
{
    int i=0;
    for(i=0;i<10;i++)
    {
        funca();
    }
    funcb();
    return 0;
}
```

编译：　　gcc hello.c -pg -L./ -la -o hello

运行：./hello
生成 gmon.out

gprof hello gmon.out –p
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
0.00	0.00	0.00	11	0.00	0.00	data_start

数据不对，实际上 gprof 对于几乎不支持动态库，具体的分析可以参见：
http://blog.csdn.net/absurd/archive/2007/01/08/1477532.aspx

在这里就不做过多分析了。

3.8.1.3. gprof 与多线程

gprof 在支持线程方面也很有问题，不过也有解决方法，具体可以参见：
http://sam.zoy.org/writings/programming/gprof.html

由于 gprof 存在着动态库和多线程的种种限制，基本上决定了其的实用性不大，这也是在这里不做太多说明的原因。

我们将重点放在 oprofile 上面。

3.8.2. OProfile

Oprofile 是用于 Linux 的若干评测和性能监控工具的一种。它可以工作在不同的体系上，包括 IA32，IA64 和 AMD Athlon 系列。它的开销小，被包含在（Linux）2.6 版的内核中。
Oprofile 使用了一个内核模块和一个用户空间守护进程，前者可以访问性能计数寄存器，后在后台运行，负责从这些寄存器收集数据。在启动守护进程之前，Oprofile 将配置事件类型以及每种事件的样本计数。如果没有配置任何事件，那么 Oprofile 将使用默认事件，即 CYCLES，该事件对处理器循环进行计数。时间的样本技术将决定事件每发生多少次计数器才增加一次。Oprofile 被设计成可以在低开销下运行，从而使后台运行的守护进程不会扰乱系统性能。
通过以上计数，可以帮助用户识别诸如循环的展开、高速缓存的使用率低、低效的类型转换和冗余操作、错误预测转移等问题。

oprofile的帮助文件可以参见：http://oprofile.sourceforge.net/doc/

3.8.2.1. Oprofile 范例

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

int fast_multiply(x,    y)
{
      return x * y;
}
int slow_multiply(x, y)
```

```
{
	int i, j, z;
	for (i = 0, z = 0; i < x; i++)
		z = z + y;
	return z;
}
int main()
{
	int i,j;
	int x,y;
	for (i = 0; i < 2000; i ++)
	{
		for (j = 0; j < 300 ; j++)
		{
			x = fast_multiply(i, j);
			y = slow_multiply(i, j);
		}
	}
	return 0;
}
```

编译：gcc -g hello.c -o hello

运行：

```
# opcontrol –reset                          //重置 Oprofile 的统计数据
# opcontrol --no-vmlinux                    //不分析内核
# opcontrol -i /mnt/msc_int0/hello
# opcontrol -e CPU_CYCLES:5000:0:1:1        //设置跟踪事件
# opcontrol –start                          //开始采样
Using 2.6+ OProfile kernel interface.
Using log file /var/lib/oprofile/oprofiled.log
Daemon started.
Profiler running.
# ./hello                                   //运行进程
# opcontrol –shutdown                       //停止采样
Stopping profiling.
Killing daemon.
```

查看结果：

```
# opreport

CPU: ARM/ARM11 PMU, speed 0 MHz (estimated)
Counted CPU_CYCLES events (clock cycles counter) with a unit mask of 0x00 (No unit mask) count 5000
```

```
   CPU_CYCLES:5000|
   samples|          %|
------------------
   2352763 100.000 hello

# opreport -l
CPU: ARM/ARM11 PMU, speed 0 MHz (estimated)
Counted CPU_CYCLES events (clock cycles counter) with a unit mask of 0x00 (No unit mask) count 5000
samples    %             symbol name
2345696    99.6996    slow_multiply
3968          0.1687    main
3099          0.1317    fast_multiply
```

精确到了函数，还能不能更精确呢？

可以

```
# opreport –d

CPU: ARM/ARM11 PMU, speed 0 MHz (estimated)
Counted CPU_CYCLES events (clock cycles counter) with a unit mask of 0x00 (No unit mask) count 5000
vma          samples    %              symbol name
00008414 2345675    99.7040    slow_multiply
   00008414 954           0.0407
   00008418 19           8.1e-04
   0000841c 308           0.0131
……
   00008474 12           5.1e-04
   00008478 102           0.0043
0000847c 4003          0.1701    main
   00008494 7             0.1749
   0000849c 1             0.0250
.....
   000084f8 3             0.0749
   00008500 5             0.1249
000083e4 2960          0.1258    fast_multiply
   000083e4 560          18.9189
   000083e8 110          3.7162
.....
   0000840c 133           4.4932
   00008410 227           7.6689
```

怎样才能使之与代码相结合呢？

```
# opannotate --source ./hello

opannotate (warning): unable to open for reading: /home/jkvp74/bookcode/hello.c
/*
 * Total samples for file : "/home/jkvp74/bookcode/hello.c"
 *
 * 2352577 100.000
 */
```

因为没有源文件，所以 opannotate 不能将时间与源代码对应上。我可以把源代码拷贝到嵌入式设备上，但不一定可以保证开发时的环境。Opannotate 提供相关的选项。
--base-dir：指定在 build 软件时，源文件的根路径。
--search-dir：指定嵌入式设备中，文档保存的根路径。

```
# opannotate --source --base-dirs=/home/jkvp74/bookcode/ --search-dirs=/mnt/msc_int0/ ./hello
                    :#include <stdlib.h>
                    :#include <stdio.h>
                    :
                    :int fast_multiply(x,    y)
 /* fast_multiply total:      3020    0.1284 */
   1413    0.0601 :       return x * y;
    355    0.0151 :}
                    :int slow_multiply(x, y)
 /* slow_multiply total: 2345582 99.7027 */
                    :       int i, j, z;
1626169 69.1229 :       for (i = 0, z = 0; i < x; i++)
716317 30.4482 :                z = z + y;
   1278    0.0543 :       return z;
    123    0.0052 :}
                    :int main()
 /* main total:      3975    0.1690 */
                    :       int i,j;
                    :       int x,y;
     15 6.4e-04 :       for (i = 0; i < 2000; i ++)
                    :       {
   1805    0.0767 :               for (j = 0; j < 300 ; j++)
                    :               {
    656    0.0279 :                       x = fast_multiply(i, j);
   1499    0.0637 :                       y = slow_multiply(i, j);
                    :               }
                    :       }
                    :       return 0;
                    :}
```

从代码中，我们可以看到 slow_multiply 使用了大量的采样，其中
1626169 69.1229 :　　for (i = 0, z = 0; i < x; i++)
716317 30.4482 :　　　　z = z + y;
占用大量的机器时间，优化时需要重点考虑。

3.8.2.2.　多个文件

现在我们将上面的范例，拆成两个文件：

hello.c

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

int fast_multiply(int x,int y);
int slow_multiply(int x,int y);

int main()
{
	int i,j;
	int x,y;
	for (i = 0; i < 2000; i ++)
	{
		for (j = 0; j < 300 ; j++)
		{
			x = fast_multiply(i, j);
			y = slow_multiply(i, j);
		}
	}
	return 0;
}
```

a.c

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

int fast_multiply(x,    y)
{
	return x * y;
}
int slow_multiply(x, y)
```

```
{
        int i, j, z;
        for (i = 0, z = 0; i < x; i++)
                z = z + y;
        return z;
}

gcc -g hello.c a.c -o hello

# opcontrol --reset
# opcontrol --no-vmlinux
# opcontrol -i /mnt/msc_int0/hello
# opcontrol -e CPU_CYCLES:5000:0:1:1
# opcontrol --start
Using 2.6+ OProfile kernel interface.
Using log file /var/lib/oprofile/oprofiled.log
Daemon started.
Profiler running.
# ./hello
# opcontrol --shutdown
Stopping profiling.
Killing daemon.

# opreport
   samples|          %|
------------------
   2352612 100.000 hello

# opreport -l
samples    %              symbol name
2345494    99.6974    slow_multiply
4091         0.1739    main
3026         0.1286    fast_multiply
1            4.3e-05    __libc_csu_fini

# opannotate --source --base-dirs=/home/jkvp74/bookcode/ --search-dirs=/mnt/msc_int0/ ./hello
/*
 * Total samples for file : "/home/jkvp74/bookcode/a.c"
 *
 * 2348520 99.8261
 */
                        :#include <stdlib.h>
                        :#include <stdio.h>
                        :
```

```
                    :int fast_multiply(x,    y)
 /* fast_multiply total:      3026    0.1286 */
   1460    0.0621 :       return x * y;
    308    0.0131 :}
                    :int slow_multiply(x, y)
 /* slow_multiply total: 2345494 99.6974 */
                    :       int i, j, z;
1632461 69.3893 :       for (i = 0, z = 0; i < x; i++)
709944 30.1768 :                z = z + y;
   1264    0.0537 :       return z;
     98    0.0042 :}
                    :
/*
 * Total samples for file : "/home/jkvp74/bookcode/hello.c"
 *
 *     4091    0.1739
 */
                    :#include <stdlib.h>
                    :#include <stdio.h>
                    :
                    :int fast_multiply(int x,int y);
                    :int slow_multiply(int x,int y);
                    :
                    :int main()
 /* main total:      4091    0.1739 */
                    :       int i,j;
                    :       int x,y;
      11 4.7e-04 :      for (i = 0; i < 2000; i ++)
                    :       {
   1929    0.0820 :               for (j = 0; j < 300 ; j++)
                    :               {
    654    0.0278 :                       x = fast_multiply(i, j);
   1497    0.0636 :                       y = slow_multiply(i, j);
                    :               }
                    :       }
                    :       return 0;
                    :}
                    :
opannotate (warning): unable to open for reading:
/vobs/jem/cee4_lsp/rpm/tmp/BUILD/glibc-2.3.3/csu/elf-init.c
/*
 * Total samples for file : "/vobs/jem/cee4_lsp/rpm/tmp/BUILD/glibc-2.3.3/csu/elf-init.c"
 *
 *          1 4.3e-05
```

```
 */
```

可见 oprofile 对于一个文件或多个文件编译合成的 process 操作来讲，是一样的。

3.8.2.3. Oprofile 与动态库

我们再来看看动态库对 oprofile 的影响。

a.c

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

int fast_multiply(x,    y)
{
        return x * y;
}
int slow_multiply(x, y)
{
        int i, j, z;
        for (i = 0, z = 0; i < x; i++)
                z = z + y;
        return z;
}
```

编译生成动态库：gcc -fPIC -shared -g a.c -o liba.so

hello.c

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

int fast_multiply(int x,int y);
int slow_multiply(int x,int y);

int main()
{
        int i,j;
        int x,y;
        for (i = 0; i < 2000; i ++)
        {
                for (j = 0; j < 300 ; j++)
                {
                        x = fast_multiply(i, j);
                        y = slow_multiply(i, j);
                }
```

```
    }
    return 0;
}
```

编译主进程：gcc -g hello.c -L./ -la -o hello

```
# opcontrol --reset
# opcontrol --no-vmlinux
# opcontrol -i /mnt/msc_int0/hello
# opcontrol -e CPU_CYCLES:5000:0:1:1
# opcontrol --start
Using 2.6+ OProfile kernel interface.
Using log file /var/lib/oprofile/oprofiled.log
.Daemon started.
Profiler running.
# ./hello
# opcontrol --shutdown
Stopping profiling.
Killing daemon.
```

查看结果

```
# opreport
CPU: ARM/ARM11 PMU, speed 0 MHz (estimated)
Counted CPU_CYCLES events (clock cycles counter) with a unit mask of 0x00 (No unit mask) count 5000
  CPU_CYCLES:5000|
  samples|          %|
------------------
         54 100.000 hello

# opreport -l
CPU: ARM/ARM11 PMU, speed 0 MHz (estimated)
Counted CPU_CYCLES events (clock cycles counter) with a unit mask of 0x00 (No unit mask) count 5000
samples  %             symbol name
42           77.7778   main
12           22.2222   .plt
```

我们并不能看到具体的函数热点信息。

```
#opannotate --source --base-dirs=/home/jkvp74/bookcode/ --search-dirs=/mnt/msc_int0/
/*
 * Total samples for file : "/home/jkvp74/bookcode/hello.c"
 *
```

```
 *        42 77.7778
 */



                    :#include <stdlib.h>
                    :#include <stdio.h>
                    :
                    :int fast_multiply(int x,int y);
                    :int slow_multiply(int x,int y);
                    :
                    :int main()
 /* main total:       42 77.7778 */
                    :       int i,j;
                    :       int x,y;
                    :       for (i = 0; i < 2000; i ++)
                    :       {
      14 25.9259 :              for (j = 0; j < 300 ; j++)
                    :               {
      12 22.2222 :                      x = fast_multiply(i, j);
      16 29.6296 :                      y = slow_multiply(i, j);
                    :               }
                    :       }
                    :       return 0;
                    :}
                    :
```

从这里我们可以看到只能看到进程自身源代码热点信息，如何才能看到动态库中的信息呢？

Oprofile 提供了—sepatate 选项用来提供对动态库等的支持。

```
# opcontrol --separate
        Separate profiles as follows :

        none:         no profile separation
        library:    separate shared library profiles per-application
        kernel:     same as library, plus kernel profiles
        thread:     per-thread/process profiles
        cpu:          per CPU profiles
        all:          all of the above

# opcontrol –reset
# opcontrol --no-vmlinux
# opcontrol -e CPU_CYCLES:5000:0:1:1
# opcontrol --separate=lib
# opcontrol --start
```

```
Using 2.6+ OProfile kernel interface.
Using log file /var/lib/oprofile/oprofiled.log
Daemon started.
Profiler running.
# ./hello
# opcontrol --shutdown
Stopping profiling.
Killing daemon.

# opreport
CPU: ARM/ARM11 PMU, speed 0 MHz (estimated)
CPU: ARM/ARM11 PMU, speed 0 MHz (estimated)
Counted CPU_CYCLES events (clock cycles counter) with a unit mask of 0x00 (No unit mask) count 5000
   CPU_CYCLES:5000|
   samples|          %|
------------------
   2385750 100.000 helloc
                CPU_CYCLES:5000|
                samples|          %|
              ------------------
                2354508 98.6905 liba.so
                   24174    1.0133 libc-2.3.3.so
                    6986    0.2928 helloc
                      82    0.0034 ld-2.3.3.so

# opreport -l
CPU: ARM/ARM11 PMU, speed 0 MHz (estimated)
Counted CPU_CYCLES events (clock cycles counter) with a unit mask of 0x00 (No unit mask) count 5000
samples    %             image name                    symbol name
2340189    98.0903    liba.so                      slow_multiply
24174        1.0133    libc-2.3.3.so              (no symbols)
12774        0.5354    liba.so                      fast_multiply
5714          0.2395    helloc                        main
1545          0.0648    liba.so                      .plt
1272          0.0533    helloc                        .plt
82            0.0034    ld-2.3.3.so               (no symbols)

# opannotate --source --base-dirs=/home/jkvp74/bookcode --search-dirs=/mnt/msc_int0
/*
 * Command line: opannotate --source --base-dirs=/home/jkvp74/bookcode --search-dirs=/mnt/msc_int0
```

```
 *
 * Interpretation of command line:
 * Output annotated source file with samples
 * Output all files
 *
 * CPU: ARM/ARM11 PMU, speed 0 MHz (estimated)
 * Counted CPU_CYCLES events (clock cycles counter) with a unit mask of 0x00 (No unit mask) count 5000
 */
/*
 * Total samples for file : "/home/jkvp74/bookcode/c.c"
 *
 * 2355744 99.6900
 */


                    :#include <stdlib.h>
                    :#include <stdio.h>
                    :
                    :int fast_multiply(int x,int    y)
 /* fast_multiply total:    10172    0.4305 */
   1124    0.0476 :       return x * y;
     92    0.0039 :}
                    :int slow_multiply(int x,int y)
 /* slow_multiply total: 2345572 99.2596 */
                    :       int i, j, z;
1692341 71.6162 :       for (i = 0, z = 0; i < x; i++)
649922 27.5033 :                z = z + y;
    999    0.0423 :       return z;
     80    0.0034 :}
                    :
/*
 * Total samples for file : "/home/jkvp74/bookcode/b.c"
 *
 *     6046    0.2559
 */


                    :#include <stdlib.h>
                    :#include <stdio.h>
                    :
                    :int fast_multiply(int x,int    y);
                    :int slow_multiply(int x,int y);
                    :
```

```
                    :int main()
 /* main total:     6046    0.2559 */
                    :       int i,j;
                    :       int x,y;
      16 6.8e-04 :      for (i = 0; i < 2000; i ++)
                    :       {
   1879    0.0795 :              for (j = 0; j < 300 ; j++)
                    :               {
   1665    0.0705 :                      x = fast_multiply(i, j);
   2486    0.1052 :                      y = slow_multiply(i, j);
                    :               }
                    :       }
                    :       return 0;
                    :}
                    :
```

3.8.2.4. Oprofile 与多线程

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

int fast_multiply(x,    y)
{
      return x * y;
}
int slow_multiply(x, y)
{
      int i, j, z;
      for (i = 0, z = 0; i < x; i++)
            z = z + y;
      return z;
}

void* thread_proc(void* param)
{
      int i,j;
      for(i=0;i<2000;i++)
      {
            for(j=0;j<300;j++)
            {
                  slow_multiply(i,j);
            }
      }
```

```
	return 0;
}

int main()
{
	pthread_t tid=0 ;

	pthread_create(&tid, NULL, thread_proc, NULL);

	pthread_join(tid,NULL);

	int i,j;
	for(i=0;i<2000;i++)
	{
		for(j=0;j<300;j++)
		{
			fast_multiply(i,j);
		}
	}

	return 0;
}

gcc -L./ -lpthread -g b.c -o helloc

opcontrol --reset
# opcontrol --no-vmlinux
# opcontrol -e CPU_CYCLES:5000:0:1:1
# opcontrol --start
Using 2.6+ OProfile kernel interface.
Using log file /var/lib/oprofile/oprofiled.log
Daemon started.
Profiler running.
# ./helloc
# opcontrol --shutdown
Stopping profiling.
Killing daemon.
# opreport
CPU: ARM/ARM11 PMU, speed 0 MHz (estimated)
Counted CPU_CYCLES events (clock cycles counter) with a unit mask of 0x00 (No unit mask) count 5000
   CPU_CYCLES:5000|
   samples|          %|
------------------
```

```
   2353605 100.000 helloc
                CPU_CYCLES:5000|
                samples|          %|
              ------------------
                2353446 99.9932 helloc
                     119    0.0051 ld-2.3.3.so
                      20 8.5e-04 libc-2.3.3.so
                      20 8.5e-04 libpthread-0.10.so
# opreport -l
CPU: ARM/ARM11 PMU, speed 0 MHz (estimated)
Counted CPU_CYCLES events (clock cycles counter) with a unit mask of 0x00 (No unit mask) count 5000
samples    %            image name                 symbol name
2345283    99.6464    helloc                     slow_multiply
3203         0.1361    helloc                     thread_proc
2554         0.1085    helloc                     fast_multiply
2406         0.1022    helloc                     main
119          0.0051    ld-2.3.3.so              (no symbols)
20           8.5e-04   libc-2.3.3.so            (no symbols)
20           8.5e-04   libpthread-0.10.so       (no symbols)

# opannotate --source --base-dirs=/home/jkvp74/bookcode --search-dirs=/mnt/msc_int0
/*
 * Total samples for file : "/home/jkvp74/bookcode/b.c"
 *
 * 2353446 99.9932
 */



                      :#include <stdlib.h>
                      :#include <stdio.h>
                      :#include <pthread.h>
                      :
                      :int fast_multiply(x,    y)
 /* fast_multiply total:      2554    0.1085 */
   1673    0.0711 :       return x * y;
    209    0.0089 :}
                      :int slow_multiply(x, y)
 /* slow_multiply total: 2345283 99.6464 */
                      :       int i, j, z;
1610974 68.4471 :       for (i = 0, z = 0; i < x; i++)
731309 31.0719 :                z = z + y;
   1370    0.0582 :       return z;
    276    0.0117 :}
```

```
                    :
                    :void* thread_proc(void* param)
 /* thread_proc total:     3203    0.1361 */
                    :       int i,j;
      18 7.6e-04 :      for(i=0;i<2000;i++)
                    :       {
   2579    0.1096 :            for(j=0;j<300;j++)
                    :             {
    606    0.0257 :                  slow_multiply(i,j);
                    :             }
                    :       }
                    :       return 0;
                    :}
                    :
                    :int main()
 /* main total:      2406    0.1022 */
                    :       pthread_t tid=0 ;
                    :
                    :       pthread_create(&tid, NULL, thread_proc, NULL);
                    :
                    :       pthread_join(tid,NULL);
                    :
                    :       int i,j;
      13 5.5e-04 :      for(i=0;i<2000;i++)
                    :       {
   1778    0.0755 :            for(j=0;j<300;j++)
                    :             {
    615    0.0261 :                  fast_multiply(i,j);
                    :             }
                    :       }
                    :
                    :       return 0;
                    :}
```

由此可见 OProfile 很完美的支持了动态库、多线程。

3.8.2.5. 在嵌入式环境下的 OProfile

在嵌入式环境中，Oprofile 运行的环境有所不同的。

首先嵌入式设备中，其 Flash 是有限的，这就决定了在放到嵌入式设备中的进程和动态库其都应该是经过 strip 过的，不具备符号表。这样我们就不能通过 opreport –l 来获得哪个函数被采样次数。

```
# opreport -l
CPU: ARM/ARM11 PMU, speed 0 MHz (estimated)
Counted CPU_CYCLES events (clock cycles counter) with a unit mask of 0x00 (No unit mask)
```

count 5000

samples	%	image name	symbol name
6770	33.2973	libqte-mt.so.2.3.8	(no symbols)
6553	32.2300	libc-2.3.3.so	(no symbols)
2080	10.2302	libpthread-0.10.so	(no symbols)

如果无法定位到函数，只是面对一些内存地址，Oprofile 实际上对我们几乎没有任何帮助。难道这么好的工具我们就白白的浪费在那里吗。
我们还有办法！
我们可以在程序和动态库编译出来之后，strip 之前，来获得符号表。然后，使用 opreport –d 来获得详细的信息，然后在手机外使用符号表对齐进行解析。
#opreport -d
cat op.log
CPU: ARM/ARM11 PMU, speed 0 MHz (estimated)
Counted CPU_CYCLES events (clock cycles counter) with a unit mask of 0x00 (No unit mask) count 5000

vma	samples	%	image name	symbol name
00000000	6770	33.2973	libqte-mt.so.2.3.8	(no symbols)
0004fa70	1	0.0148		
0004fa78	1	0.0148		
0004fb3c	1	0.0148		
0004fbfc	1	0.0148		
0004fcc8	1	0.0148		
0004fe90	3	0.0443		
0004fe98	1	0.0148		
0004fea8	1	0.0148		
0004fee4	1	0.0148		
0004ff08	3	0.0443		
0004ff8c	2	0.0295		

这里第一列是被采样时指令地址，第二列是被采样的次数，第三列是被采样次数占总采样次数的百分比。

我们可以根据符号表，将第一列指令地址转换为所对应的具体地址，然后把对应该函数的采样次数加在一起，从而得到每个函数具体被采样的次数，并进行排列。

为此，我用 perl 自己写了工具，来解析 Oprofile –d 的结果。

./op_sym ./report.log ./sym
结果：

4016	0.0320	libgcc_s.so.1@@000021a4
5209	0.0415	libqte-mt.so.2.3.8@@QGfxRaster<24, 1>::blt(int, int, int, int, int, int)
7363	0.0587	libqte-mt.so.2.3.8@@QPixmap::QPixmap(int, int, unsigned char const*, bool)
8248	0.0657	libc-2.3.3.so@@00064dd0

8511 0.0678 libqte-mt.so.2.3.8@@operator>>(QDataStream&, BMP_INFOHDR&)
21145 0.1685 libqte-mt.so.2.3.8@@read_dib(QDataStream&, int, int, QImage&, QImageIO*)

我曾经一度以为，精确到函数已经够了，可实际上并非如此，当我们面对一个函数，长达上百行时，我知道我错了。
能不能精确到代码行？这对于我提出了一个更高的要求。
同样由于 Flash 大小的限制，我们在编译进程和动态库的时候不能加上-g 的选项，这也就导致我们无法使用 opannotate 来将所用时间与具体代码对应上。

同样，我们能否考虑使用 opreport –d 的结果，来在设备外面解析出结果来呢？

还是有办法的，你还记得一个工具 addr2line 吗。它就是专门将代码指令转化成具体的代码行，但有个先决的条件，你编译的时候需要加上-g 的选项。

首先通过我们上面的努力，将 opreport 的结果精确到函数之后，
1、 我们需要将这个函数内，被采样的指令地址打印出来。
2、 我们需要将对应的进程或动态库，加上-g 选项，重新编译。
3、 我们使用 addr2line，来将地址指针，转化成具体的代码的行号。

3.9. 优化的层次

在我们了解到了程序热点之后，接下来我们就要考虑对代码进行优化：

程序的优化，主要包括四个层次：
1、 算法和数据结构的优化。
2、 编译器优化
3、 代码优化
4、 硬加速

一个优化新手最容易犯的错误就是：当他找到程序热点时，过于兴奋，急匆匆的就开始局部代码的优化。这时候，我们最需要的是冷静，我们需要在更大的范围内分析，是什么导致了这个程序热点的存在，有时可能根本就不需要改动你的程序。

我在做优化的时候，就遇到了这么一个例子：测试人员发现，手机在播放 gif 动画时，CPU 的利用率很高，到达了 50%~70%，要求我们将 CPU 利用率降下来。GIF 文件的播放器是 QT 开源的代码，我细细的读了一遍，感觉写的很好，几乎没什么可优化的地方。也试着改动了几个地方，效果甚微。
通过 oprofile 的结果，我发现播放器在做图像解压缩时花费了大量时间。我们都知道 GIF 动画文件，是有很多的图像文件合并到一起时，为了减小合并后的图像文件的尺寸，采用了 GIF-LZW 算法进行了压缩，其压缩比率可以达到 3：1，而且由于 gif 文件广泛的应用在互联网中，为了加快传输，现在市面上又做出了很多软件，加大压缩比，缩小文件，加快网络传输过程。
经过我的测试，文件的压缩比越高，系统在播放该文件的时候 CPU 的利用率将越高，而在文件不压缩的时候，CPU 的利用率可以降到 30%左右，满足了我们的要求。这样，我们对

gif 文件的优化，是采用降低压缩比，减少 CPU 在解压缩方面所占用的时间；而目前市面上的 gif 优化软件，大都是加大压缩比，减少图像文件的大小。

在我们提出了这个优化方案的时候，也有些顾虑，比如说压缩后 200K 的文件，不压缩后其为 600K 左右，在播放时会不会带来额外的内存占用呢？
针对这个问题，我们也做了专门的测试，发现随着压缩比的减小，文件越来越大，引擎在播放 gif 文件时，所占的内存越来越小，与我们原来预计的结果正好相反。主要原因是，引擎在播放 gif 文件时，其在做解压缩时占用了大量的内存，如果 gif 文件不进行压缩，那么这块内存就可以省出来了。

在这里，我举这个例子，主要是想告诉大家，在着手做优化想，要更多的关注业务逻辑，关注程序的结构，不要一头就扎进到程序的细节。

算法的选择是决定软件运行速度快慢的最重要的因素。好的算法能以快速高效的方式解决问题，而糟糕的算法无论怎样实现和调试都无法获得很好的性能。
我们以数据排序算法为例：
冒泡排序或许是最简单也是最慢的排序算法。它的计算复杂度为 O（n2），意味着待排序元素的数目增加一倍时，排序所花费的时间增加为原来的 4 倍。而快速排序法具有更好的计算复杂度 O(n log n)。
下面我们来进行一下比较：

	200 元素	1000 元素	10,000 元素
冒泡排序	65,000	1,000,000	100,000,000
快速排序	2,048	9,965	133,000

在后面的章节中，我们将介绍在优化中常用的查表法。

编译器优化，这也是我们使用起来最为简单、最为常用的优化方法，在编译是加上-O2 等选项，在这里我们只讲述目前最为常用的 GCC 编译器。这里我们将讲述-O2、-O3 等背后的故事，同时也会提到编译器优化为我们程序调试所带来的困难。
在这里我们需要在编译器优化和调试方面做出权衡。

编译器只能做一些最基本的优化，而一些需要创造性、复杂的优化工作只能由我们人为的去完成。这部分的内容将会很多、很繁杂。这个章节我们主要关注于 C 语言、C++语言、以及一些与芯片体系结构相关优化，我们将主要介绍 ARM 体系结构。

对于一些视频游戏等，对性能要求极高，软件优化的方法不能满足时，就需要采用硬件加速的方式。在这里，我们介绍一些硬件加速的基本知识，比如 ARM 芯片中新增加的 SIMD 指令，以及 GPU 加速的原理等，最后我们还要讲述一点，修改 GCC 编译器，来扩展系统所支持的指令。

3.10. 算法优化

由于我们所面临的事物千差万别，其算法的改进也将各种各样，在这里我只介绍在针对计算密集型代码中，采用查表法来优化代码的方法，主要是给各位朋友来扩展一下思路。

3.10.1. 查表法

为了描述查表法，我举一个例子：

我们的任务：将一个彩色图片，转换成黑白图片。

其公式也很简单：
Y = 0.299 * R + 0.587 * G + 0.114 * B

我们实现的方法：

```
void calc_lum()
{     int i;
      for(i=0;i<IMGSIZE;i++)
      {
            double r,g,b,y;
            unsigned char yy;
            r=in[i].r; g=in[i].g; b=in[i].b;
            y=0.299*r+0.587*g+0.114*b;
            yy=y; out[i]=yy;
      }
}
```

在设备上跑，转换一张图片大概 120 秒。

完全按照公式实现的，看似没有什么可以优化的。

优化 1：使用整数运算，代替浮点运算。
公式 Y = 0.299 * R + 0.587 * G + 0.114 * B 可以转化为 Y = (299 * R + 587 * G + 114 * B)/1000。
这样，我们就把一个浮点运算，转换为了整数运算。

```
void calc_lum()
{     int i;
      for(i=0;i<IMGSIZE;i++)
```

```
    {
        double r,g,b,y;
        unsigned char yy;
        r=in[i].r; g=in[i].g; b=in[i].b;
        y=(299*r+587*g+114*b)/1000;
        yy=y; out[i]=yy;
    }
}
```

在设备上跑，转换一张图片大概 45 秒，提高了近 3 倍的样子。

优化 2：将除法运算，转换为移位操作。
转换为移位操作的关键，在于除数必须为 2 的 N 次方。
公式 Y = (299 * R + 587 * G + 114 * B)/1000 转换为 Y=(R*1224+G*2404+B*467)/4096。

代码优化为：

```
void calc_lum()
{   int i;
    for(i=0;i<IMGSIZE;i++)
    {
        double r,g,b,y;
        unsigned char yy;
        r=in[i].r; g=in[i].g; b=in[i].b;
        y=(1224*r+2404*g+467*b)>>10;
        yy=y; out[i]=yy;
    }
}
```

在设备上跑，转换一张图片大概 30 秒。

到这里是不是极限了呢？

优化 3：现在我们的查表法，马上就要登场了。

现在程序性能的瓶颈在于，对于每一个像素点，都要做 3 个乘法 2 个加法和 1 个移位计算。我们注意到，r g b 其取值范围为 0～255，那么我们就可以考虑在转换图片之前，针对其所有可能值进行计算，得到转换后的数值，放在内存中。在做图像转换时，只需要查表就可以得到对应的数值，从而节省 CPU 计算的时间。

初始化转换表：

```
void table_init()
{   int i;
    for(i=0;i<256;i++)
    {   D[i]=i*1224; D[i]=D[i]>>12;
```

```
            E[i]=i*2404; E[i]=E[i]>>12;
            F[i]=i*467; F[i]=F[i]>>12;
        }
}
```

这样，针对每个 r g b 的初值，我们获得了其相应的数值数组，其所占内存为：
256(数组含有 256 个元素)*4（每个元素占 4 个字节）*3（共有 3 个数组）=3K。

由数值计算，转换为查表：

```
void calc_lum()
{     int I;
      for(i=0;i<IMGSIZE;i++)
      {     int r,g,b,y;
            r=D[in[i].r]; g=E[in[i].g]; b=F[in[i].b]; //查表
            y=r+g+b;
            out[i]=y;
      }
}
```

这样，图像转换中，我们就由 3 个乘法 2 个加法和一个移位运算，优化为 2 个加法，从而大大的节省了时间。

在设备上跑，转换一张图片大概 2 秒。

这样，转换的效率又提高了 15 倍。

我们再来回顾一下优化的过程：
1、 优化前，转换一张图片 120 秒
2、 使用乘除替代浮点运算，转换一张图片 45 秒，提高了 3 倍。
3、 使用移位代替除法运算，转换一张图片 20 秒。
4、 使用查表法，转换一张图片 2 秒，又提高了 10 倍。

从这个优化的过程来看，我们也能清楚的看到算法的优化，对程序性能的提高起着非常重要的作用。

查表法在现实世界中应用非常广泛，其是一种典型的以内存换速度的优化方法。

由于算法优化，更多的涉及数学、现实事物模型等方面的知识，超出了本书的范围，只能点到为止，更多的只能是大家具体问题具体分析。

3.10.2. 使用异步通讯替代多线程

在这里我引用《UNIX 网络编程 第一卷套接口 API》的一个例子，来讲述 socket 通讯的各个架构对于性能的影响。

服务器端：

```
int main(int argc,char **argv)
{
    int listenfd,connfd;
    pid_t childpid;
    socklen_t clien;
    struct    sockaddr_in cliaddr,servaddr;
    bzero(&servaddr,sizeof(servaddr));
    servaddr.sin_family=AF_INET;
    servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
    servaddr.sin_port=htons(SERV_PORT);
    bind(listenfd,&servaddr,sizeof(servaddr));
    listen(listenfd,LISTENQ);

    for(;;)
    {
        clilen=sizeof(cliaddr);
        connfd = accept(listenfd, &cliaddr,&clilen);
        if((childpid=fork()) == 0)
        {
            close(listenfd);
            str_echo(connfd);
            exit(0);
        }
        close(connfd);
    }
}

void str_echo(int sockfd)
{
    ssize_t     n;
    char        buf[MAXLINE];
    while((n = read(sockfd,buf,MAXLINE))>0)
        writen(sockfd,buf,n);
    if(n<0 && errno == EINTR)
        goto again;
    else if(n<0)
        err_sys("str_ech0:read error");
}
```

这个服务器很简单，就是当接收到一个请求时，创建一个进程，然后从 socket 中读取输入，然后将内容再输出给客户端。

客户端程序：

```
int main(int argc,char **argv)
{
	int sockfd;
	struct sockaddr_in	servaddr;

	sockfd=socket(AF_INET,SOCK_STREAM,0);
	bzero(&servaddr,sizeof(servaddr));
	servaddr.sin_family = AF_INET;
	servaddr.sin_port = htons(SERV_PORT);
	inet_pton(AF_INET,argv[1],&servaddr.sin_addr);

	connect(sockfd,(SA *)&servaddr,sizeof(servaddr));
	str_cli(stdin,sockfd);

	exit(0);
}
```

客户端的主要功能是，从标准输入中读入一行文本，写道服务器上，然后再接收服务器的返回，并把返回的内容显示在标准输出上。

其主要逻辑下面我们使用 5 种方法，来实现其客户端，然后比较其性能。

停等版本

```
void str_cli(FILE *fp,int sockfd)
{
	char sendline[MAXLINE],recvline[MAXLINE];
	while(fgets(sendline,MAXLINE,fp)!=NULL)
	{
		writen(sockfd,sendline,strlen(sendline));
		if (readline(sockfd,recvline,MAXLINE) == 0)
			err_quit("str_cli server terminated prematurely");
		fputs(recvline,stdout);
	}
}
```

select 加阻塞 I/0 版本

```
void str_cli(FILE *fp,int sockfd)
{
	int maxfdp1,stdineof;
	fd_set	rset;
	char	buf[MAXLINE];
	int n;

	stdineof = 0;
```

```
    FD_ZERO(&rset);

    for(;;)
    {
        if(stdineof ==0)
            FD_SET(fileno(fp),&rset);
        FD_SET(sockfd,&rset);
        maxfdp1 = max(fileno(fp),sockfd) + 1;
        select(maxfdp1,&rset,NULL,NULL,NULL);
        if(FD_ISSET(sockfd,&rset))
        {
            if( (n = Read(sockfd,buf,MAXLINE)) = 0)
            {
                if(stdineof == 1)
                    return;
                else
                    err_quit("str_cli:server terminated prematurely");
            }
            write(fileno(stdout),buf,n);
        }
        if(FD_ISSET(fileno(fp),&rset))
        {
            if( (n = Read(fileno(fp),buf,MAXLINE))== 0)
            {
                stdineof=1;
                shutdown(sockfd,SHUT_WR);
                FD_CLR(fileno(fp),&rset);
                continue;
            }
            write(sockfd,buf,n);
        }
    }
}
```

非阻塞 I/O 版本

```
void str_cli(file * fp,int sockfd)
{
    int         maxfdp1,val,stdineof
    ssize_t    n,nwritten;
    fd_set      rset,wset;
    char    to[MAXLINE],fr[MAXLINE];
    char      *toiptr,*tooptr,*friptr,*froptr;
    val=fcntl(sockfd,F_GETFL,0);
    fcntl(sockfd,F_SETFL,val | O_NONBLOCK);
```

```
val=fcntl(STDIN_FILENO,F_GETFL,0);
fcntl(STDIN_FILENO,F_SETFL,val | O_NONBLOCK);
val=fcntl(STDOUT_FILENO,F_GETFL,0);
fcntl(STDOUT_FILENO,F_SETFL,val | O_NONBLOCK);

toiptr=tooptr=to;
friptr=froptr=fr;
stdineof=0;
maxfdp1=max(max(STDIN_FILENO,STDOUT_FILENO),sockfd)+1;
for(;;)
{
        fd_zero(&rset);
        fd_zero(&wset);
        if(stdineof == 0 && toipr < &to[MAXLINE])
              fd_set(STDIN_FILENO,&rset);
        if(fripr<&fr[MAXLINE])
              fd_set(sockfd,&wset);
        if(froptr != friptr)
              fd_set(STDOUT_FILENO,&wset);
        select(maxfdp1,&rset,&wret,NULL,NULL);

        if(FD_ISSET(STDIN_FILENO,&rset)
        {
               if((n=read(STDIN_FILENO,toipr,&to[MAXLINE]-toipr))<0)
               {
                      if(errno!=EWOULDBLOCK)
                             err_sys("read error on stdin");

               }else if (n==0)
               {
                      stdineof=1;
                      if(tooptr== toiptr)
                             shutdown(sockfd,SHUT_WR);
               }else
               {
                      toiptr += n;
                      FD_SET(sockfd,&wset);
               }
        }

        if(FD_ISSET(sockfd,&rset))
        {
               if((n=read(sockfd,friptr,&fr[MAXLINE]-friptr))<0)
               {
```

```
                        if(errno != EWOULDBLOCK)
                                err_sys("read error on socket");
                    }else if(n==0)
                    {
                        if(stdineof)
                            return;
                        else
                            err_quit("str_cli:server terminated prematurely");
                     }else
                     {
                        friptr+=n;
                        FD_SET(STDOUT_FINLENO,&wset);
                     }
            }
      }
}
```

fork 版本

```
void str_cli(FILE *fp,int sockfd)
{
        pid_t pid;
        char sendline[MAXLINE],recvline[MAXLINE];
        if((pid=fork())==0)
        {
            while(readline(sockfd,recvline,MAXLINE)>0)
                fputs(recvline,stdout);
            kill(getppid(),SIGTERM);
            exit(0);
        }
        while(fgets(sendline,MAXLINE,fp)!=NULL)
            writen(sockfd,sendline,strlen(sendline));

        shutdown(sockfd,SHUT_WR);
        pause();
        return;
}
```

线程化版本

```
void str_cli(FILE *fp_arg,int sockfd_arg)
{
      char         recvline[MAXLINE];
      pthread_t    tid;
      sockfd=sockfd_arg;
      fp=fp_arg;
```

```
        pthread_create(&tid,NULL,copyto,NULL);
        while(readline(sockfd,recvline,MAXLINE)>0)
                fputs(recvline,stdout);
}

void * copyto(void *arg)
{
        char sendline[MAXLINE];
        while(fgets(sendline,MAXLINE,fp)!=NULL)
                writen(sockfd,sendline,strlen(sendline));

        shutdown(sockfd,SHUT_WR);
        return(NULL);
}
```

测试环境是从一个 solaris 客户主机到 RTT 为 175 毫秒的一个服务器主机拷贝 2000 行文本：

停等版本，354.0 秒。
select 加阻塞 I/O 版本，12.3 秒。
非阻塞 I/O 版本，6.9 秒
fork 版本，8.7 秒
线程化版本，8.5 秒

从上面的测试结果可以看出，非阻塞版本几乎快出 select 加阻塞 I/O 版本一倍。fork 版本比非阻塞版本稍慢，但是非阻塞版本代码相比 fork 版本代码的复杂的多。

但是，上面的测试只是停留在具有少量并发访问的基础上，当同时面对上百个请求时，使用 fork 或线程化版本，系统将很难承受，这时非阻塞 I/O 的版本将最适宜的。

3.11. GCC 编译优化

编译器在不断的进步，不断的尝试帮我们多做一些工作，这也使得我们的程序员越来越懒，疏于去了解如何去写高效的代码。

编译器的确能帮我们做很多优化工作，但它同时也会为我们的调试带来负面作用，比如在 GCC 采用-O2 优化代码后，程序在运行过程中，不会产生标准的栈帧，这样在产生 coredump 后就无法准确定位，给调试工作带来了很大的困扰。

因此，我们需要了解编译器都做了哪些优化，这样我们才能在代码优化与易于调试之间获取一个平衡。
在实现代码优化方面，各个编译器的实现各不相同，本书仅以 GCC 为准，讲述 GCC 的优化选项。

3.11.1. 体系无关优化

在 GCC 中，我们最常用的优化选项为-O；可以是-O1 –O2 -O3 和-Os。-O0 关闭编译器优化。-O 和-O1（第一级优化）是等价的，告诉 GCC 优化代码。使用-O 或-O1，编译器将在不显著增加编译时间的基础上，尝试减少代码尺寸、缩短执行时间。使用-O2 和-O3 将增加优化级别，同时仍然保留-O1 所采取的优化方法。如果想最大限度的减小代码尺寸，可以使用-Os。
下面，我们将列出 GCC 每个优化级别所包含的优化选项，如果你想关掉某一个优化选项，你可以在-f 和优化选项之间增加“no”。

例如
gcc myprog.c –o myprog –O1 –fno-defer-pop。

第一级 GCC 优化

选项	描述
-fcprop-registers	因为在函数中把寄存器分配给变量，所以编译器执行第二次检查以便减少调度依赖性(两个段要求使用相同的寄存器)并且删除不必要的寄存器复制操作。
-fdefer-pop	这种优化技术与汇编语言代码在函数完成时如何进行操作有关。一般情况下，函数的输入值被保存在堆栈种并且被函数访问。函数返回时，输入值还在堆栈种。 一般情况下，函数返回之后，输入值被立即弹出堆栈。这样做会使堆栈种的内容有些杂乱。
-fdelayed-branch	这种技术试图根据指令周期时间重新安排指令。 它还试图把尽可能多的指令移动到条件分支前，以便最充分的利用处理器的治理缓存。
-fguess-branch-probability	就像其名称所暗示的，这种技术试图确定条件分支最可能的结果，并且相应的移动指令，这和延迟分支技术类似。 因为在编译时预测代码的安排，所以使用这一选项两次编译相同的 c 或者 c++代码很可能会产生不同的汇编语言代码， 这取决于编译时编译器认为会使用那些分支。因为这个原因，很多程序员不喜欢采用这个特性，并且专门地使用-fno-guess-branch-probability 选项关闭这个特性
-fif-conversion	if-then 语句应该是应用程序中仅次于循环的最消耗时间的部分。简单的 if-then 语句可能在最终的汇编语言代码中产生众多的条件分支。 通过减少或者删除条件分支，以及使用条件传送 设置标志和使用运算技巧来替换他们，编译器可以减少 if-then 语句中花费的时间量。
-fif-conversion2	这种技术结合更加高级的数学特性， 减少实现 if-then 语句所需的条件分支。
-floop-optimize	过优化如何生成汇编语言中的循环， 编译器可以在很大程序上提高应用程序的性能。通常，程序由很多大型且复杂的循环构成。通过删除在循环内没有改变值的变量赋值操作，可以减少循环内执行指令的数量，在很大程度上提高性能。 此外优化那些确定何

	时离开循环的条件分支， 以便减少分支的影响。
-fmerge-constants	使用这种优化技术，编译器试图合并相同的常量. 这一特性有时候会导致很长的编译时间，因为编译器必须分析 c 或者 c++程序中用到的每个常量，并且相互比较他们.
-fomit-frame-pointer	
-ftree-ccp	
-ftree-ch	
-ftree-copyrename	
-ftree-dce	
-ftree-dominator-opts	
-ftree-dse	
-ftree-fre	
-ftree-lrs	
-ftree-sra	
-ftree-ter	

第二级优化

-falign-functions	这个选项用于使函数对准内存中特定边界的开始位置。 大多数处理器按照页面读取内存，并且确保全部函数代码位于单一内存页面内，就不需要叫化代码所需的页面。
-falign-jumps	
-falign-labels	
-fcaller-saves	这个选项指示编译器对函数调用保存和恢复寄存器，使函数能够访问寄存器值，而且不必保存和恢复他们。 如果调用多个函数，这样能够节省时间，因为只进行一次寄存器的保存和恢复操作，而不是在每个函数调用中都进行。
-fcrossjumping	这是对跨越跳转的转换代码处理， 以便组合分散在程序各处的相同代码。 这样可以减少代码的长度， 但是也许不会对程序性能有直接影响。
-fcse-follow-jumps	这种特别的通用子表达式消除技术扫描跳转指令，查找程序中通过任何其他途径都不会到达的目标代码。 这种情况最常见的例子就式 if-then-else 语句的 else 部分。
-fcse-skip-blocks	
-fdelete-null-pointer-checks	这种优化技术扫描生成的汇编语言代码，查找检查空指针的代码。 编译器假设间接引用空指针将停止程序。 如果在间接引用之后检查指针， 它就不可能为空。
-fexpensive-optimizations	这种技术执行从编译时的角度来说代价高昂的各种优化技术，但是它可能对运行时的性能产生负面影响。
-fforce-mem	这种优化再任何指令使用变量前，强制把存放再内存位置中的所有变量都复制到寄存器 中。 对于只涉及单一指令的变量，这样也许不会有很大的优化效果. 但是对于再很多指令(必须数学操作)中都涉及到的变量来说，这会时很显著的优化，因为和访

	问内存中的值相比 ,处理器访问寄存器中的值要快的多。
-fgcse	这种技术对生成的所有汇编语言代码执行全局通用表达式消除历程。 这些优化操作试图分析生成的汇编语言代码并且结合通用片段， 消除冗余的代码段。如果代码使用计算性的 goto, gcc 指令推荐使用-fno-gcse 选项。
-fgcse-lm	
-fgcse-sm	
-foptimize-sibling-calls	这种技术处理相关的和/或者递归的函数调用。 通常，递归的函数调用可以被展开为一系列一般的指令， 而不是使用分支。 这样处理器的指令缓存能够加载展开的指令并且处理他们，和指令保持为需要分支操作的单独函数调用相比，这样更快。
-fpeephole2	这个选项允许进行任何计算机特定的观察孔优化。
-fregmove	编译器试图重新分配 mov 指令中使用的寄存器，并且将其作为其他指令操作数，以便最大化捆绑的寄存器的数量。
-freorder-blocks	
-freorder-functions	这种优化技术允许重新安排指令块以便改进分支操作和代码局部性。
-frerun-cse-after-loop	这种技术在对任何循环已经进行过优化之后重新运行通用子表达式消除例程。这样确保在展开循环代码之后更进一步地优化还编代码。
-frerun-loop-opt	
-fsched-interblock	这种技术使编译器能够跨越指令块调度指令。 这可以非常灵活地移动指令以便等待期间完成的工作最大化。
-fsched-spec	
-fschedule-insns	编译器将试图重新安排指令，以便消除等待数据的处理器。 对于在进行浮点运算时有 延迟的处理器来说， 这使处理器在等待浮点结果时可以加载其他指令。
-fschedule-insns2	
-fstrength-reduce	这种优化技术对循环执行优化并且删除迭代变量。 迭代变量是捆绑到循环计数器的变量，比如使用变量，然后使用循环计数器变量执行数学操作的 for-next 循环。
-fstrict-aliasing	这种技术强制实行高级语言的严格变量规则。 对于 c 和 c++程序来说，它确保不在数据类型之间共享变量. 例如，整数变量不和单精度浮点变量使用相同的内存位置。
-fthread-jumps	用这种优化技术与编译器如果处理汇编代码中的条件和非条件分支有关。 在某些情况下，一条跳转指令可能转移到另一条分支语句。 通过一连串跳转，编译器确定多个跳转之间的最终目标并且把第一个跳转重新定向到最终目标。
-ftree-pre	
-funit-at-a-time	这种优化技术指示编译器在运行优化例程之前读取整个汇编语言代码。 这使编译器可以重新安排不消耗大量时间的代码以便优化指令缓存。 但是，这会在编译时花费相当多的内存，对于小型计算机可能是一个问题。
-fweb	构建用于保存变量的伪寄存器网络。 伪寄存器包含数据，就像

	他们是寄存器一样，但是可以使用各种其他优化技术进行优化，比如 cse 和 loop 优化技术。

代码尺寸优化

选项	描述
-falign-functions	
-falign-jumps	
-falign-labels	
-falign-loops	
-fprefetch-loop-arrays	
-freorder-blocks	
-freorder-blocks-and-partition	
-ftree-ch	

O3 优化

选项	描述
-fgcse-after-reload	这中技术在完全重新加载生成的且优化后的汇编语言代码之后执行第二次 gcse 优化,帮助消除不同优化方式创建的任何冗余段。
-finline-functions	这种优化技术不为函数创建单独的汇编语言代码， 而是把函数代码包含在调度程序的代码中。 对于多次被调用的函数来说，为每次函数调用复制函数代码。 虽然这样对于减少代码长度不利，但是通过最充分的利用指令缓存代码，而不是在每次函数调用时进行分支操作，可以提高性能。
-funswitch-loops	

手动优化选项

选项	描述
-fbounds-check	
-fdefault-inline	
-ffast-match	
-ffinite-math-only	
-ffloat-store	
-fforce-addr	
-ffunction-cse	
-finline	
-finline-functions	
-finline-limit=n	
-fkeep-inline-functions	
-fkeep-static-consts	

-fmath-errno	
-fmerge-all-constants	
-ftrapping-math	
-ftrapv	
-funsafe-math-optimizations	

3.11.2. 内部连接和外部连接

在说内部连接与外部连接前，先说明一些概念。

1.声明

一个声明将一个名称引入一个作用域;

在 c++中，在一个作用域中重复一个声明是合法的

以下都是声明：

int foo(int,int); //函数前置声明

typedef int Int; //typedef 声明

class bar; //类前置声明

extern int g_var; //外部引用声明

class bar; //类前置声明

typedef int Int; //typedef 声明

extern int g_var; //外部引用声明

friend test; //友员声明

using std::cout; //名字空间引用声明

friend test; //友员声明

using std::cout; //名字空间引用声明

```
int foo(int,int); //函数前置声明
```

在同一个作用域中你可以多次重复这些声明。

有两种声明不能重复，那就是类成员函数及静态数据成员的声明

```
class foo
{
   static int i;
   static int i;//不可以
   public:
      int foo();
      int foo();//不可以
};
```

2.定义

一个定义提供一个实体(类型、实例、函数)在一个作用域的唯一描述。

在同一作用域中不可重复定义一个实体。

以下都是定义。

```
int y;

class foo ;

struct bar ;

foo* p;

static int i;

enum Color;

const double PI = 3.1415;

union Rep;

void test(int p) {};

foo a;

bar b;
```

3.编译单元

当一个 c 或 cpp 文件在编译时，预处理器首先递归包含头文件，形成一个含有所有必要信息的单个源文件,这个源文件就是一个编译单元。这个编译单元会被编译成为一个与 cpp 文件名同名的目标文件(.o 或是.obj)。连接程序把不同编译单元中产生的符号联系起来，构成一个可执行程序。

4.自由函数

如果一个函数是自由函数，那么这个函数不是类的成员函数，也不是友元函数。

下面来看内部连接和外部连接

内部连接：如果一个名称对于它的编译单元来说是局部的，并且在连接时不会与其它编译单元中的同样的名称相冲突，那么这个名称有内部连接(注：有时也将声明看作是无连接的，这里我们统一看成是内部连接的)。

以下情况有内部连接:

a)所有的声明

b)名字空间(包括全局名字空间)中的静态自由函数、静态友元函数、静态变量的定义

c)enum 定义

d)inline 函数定义(包括自由函数和非自由函数)

e)类的定义

f)名字空间中 const 常量定义

g)union 的定义

外部连接:在一个多文件程序中，如果一个名称在连接时可以和其它编译单元交互，那么这个名称就有外部连接。

以下情况有外部连接:

a)类非 inline 函数总有外部连接。包括类成员函数和类静态成员函数

b)类静态成员变量总有外部连接。

c)名字空间(包括全局名字空间)中非静态自由函数、非静态友元函数及非静态变量

下面举例说明：

a)声明、enum 定义、union 定义有内部连接

所有的声明、enum 定义及 union 定义在编译后不会产生连接符号，也就是在不同编译单元中有相同名称的声明及 enum、union 定义并不会在连接时发生发现多个符号的错误。

```
// main.cpp

typedef int Int; //typedef  声明，内部连接

enum Color; //enum 定义,内部连接

union X //union 定义，内部连接
{
   long a;
   char b[10];
};

int main(void)
{
Int i = red;
return i;
}
```

```
// a.cpp

typedef int Int; //在 a.cpp 中重声明一个 int 类型别名，在连接时不会发生错误
enum Color; //在 a.cpp 中重定义了一个 enum Color，在连接时不会发生错误
const Int i =blue; //const 常量定义，内部连接
union X //union 定义，内部连接
{
   long a;
   char b[10];
};
```

b)名字空间中静态自由函数、静态友元函数、静态变量、const 常量定义有内部连接

```
// main.cpp

namespace test
{
   int foo(); //函数声明，内部连接
```

```
    static int i = 0; //名字空间静态变量定义，内部连接
    static int foo() { return 0;} //名字空间静态函数定义，内部连接
}

static int i = 0; //全局静态变量定义，内部连接
static int foo() {return 1;} //全局静态函数定义，内部连接
const int k = 0; //全局 const 常量定义，内部连接
int main(void)
{
    return 0;
}

//a.cpp

namespace test
{
    int i = 0; //名字空间变量定义，外部连接
    int foo() {return 0;} //名字空间函数定义，外部连接
}

int i = 0; //全局变量定义，外部连接
int k = 0; //全局变量定义，外部连接
int foo() { return 2;} //全局函数定义，外部连接
```

在全局名字空间中，main.cpp 中定义了静态变量 i,常量 k,及静态自由函数 foo 等，这些都有内部连接。如果你将这些变量或函数的 static 或是 const 修饰符去掉，在连接时就会现 multiply defined symbols 错误，它们与 a.cpp 中的全局变量、全局函数发生冲突。

c)类定义总有内部连接,而非 inline 类成员函数定义总有外部连接，不论这个成员函数是静态、虚拟还是一般成员函数，类静态数据成员定义总有外部连接。

1.类的定义有内部连接。如果不是，想象一下你在 4 个 cpp 文件中 include 定义了类 Base 的头文件，在 4 个编译单元中的类 Base 都有外部连接，在连接的时候就会出错。
2007-12-14 16:51 ripvt
看下面的例子:

```
//main.cpp

class B //类定义，内部连接
{
    static int s_i; //静态类成员声明，内部连接
    public:
        void foo() { ++s_i;} //类 inline 函数，内部连接
};
```

```
struct D
{
   void foo(); //类成员函数声明，内部连接
};

int B::s_i = 0; //类静态数据成员定义，外部连接
void D::foo() //类成员函数定义，外部连接
{
   cout << "D::foo in main.cpp" <
}

int main() //main 函数，全局自由函数，外部连接
{
   B b;
   D d;
   return 0;
}

//a.cpp

class B
{
   int k;
};

struct D
{
   int d;
};
```

在这个例子中，main.cpp 与 a.cpp 中都有 class B 和 class D 的定义，但在编译这两个 cpp 文件时并不发生 link 错误。

2.类的非 inline 成员函数(一般，静态，虚拟都是)总有外部连接，这样当你 include 了某个类的头文件，使用这个类的函数时，就能连接到正确的类成员函数上，继续以上面为例子，如果把 a.cpp 中的 struct D 改为

```
struct D //类定义
{
   int d;
   void foo(); //类成员函数声明
};
void D::foo() //类成员函数定义，外部连接
{
```

```
   cout << " D::foo in a.cpp" <
}
```

这时 main.cpp 与 a.cpp 中的 D::foo 都有外部连接，在连接就会出现 multiply defined symbols 错。

3.类的静态数据成员有外部连接，如上例的 B::s_i,这样当你在 main.cpp 中定义了类静态数据成员，其它编译单元若使用了 B::s_i,就会连接到 main.cpp 对应编译单元的 s_i。

d)inline 函数总有内部连接，不论这个函数是什么函数

```
// main.cpp

inline int foo() { return 1;} //inline 全局函数，内部连接
class Bar //类定义，内部连接
{
   public:
      static int f() { return 2;} //inline 类静态函数，内部连接
      int g(int i) { return i;} //inline 类成员函数，内部连接
};

class Base
{
   public:
      inline int k();  //类成员函数声明，内部连接
};

inline int Base::k(){return 5;} //inline 类成员函数，内部连接
int main(void)
{
   return 0;
}
```

如果你的 Base 类是定义在 Base.h 中，而 Base 的 inline 函数是在 Base.cpp 中定义的，那么在 main.cpp 中 include "Base.h"编译不会出现问题，但在连接时会找不到函数 k，所以类的 inline 函数最好放到头文件中，让每一个包含头文件的 cpp 都能找到 inline 函数。

现在对 c++中的连接有了一个认识，能清楚的知道是什么原因产生连接时错误。当你在连接时产生连接不到的错误，这说明所有的编译单元都没有这个实体的外部连接；当你在连接时发现有多个连接实体，这说明有多个编译单元提供了同名的有外部连接的实体。同时，在进行程序设计时，也要注意不要使只有本编译单元用到的函数、类、变量等有外部连接，减少与其它编译单元的连接冲突。

不过在这里没有说明 template 函数及 template class 的连接性，并且对一些特别的情况

也没有作出说明(比如 inline 函数不能被 inline)。

3.12. 程序的优化

3.12.1. 数据类型

在标准 C 中有几种内置数据类型：char　short　int　long unsigned char 等，在 Linux 32 位操作系统上，其对应大小分别为：

char	8 位
short	16 位
int	32 位
long	32 位
unsigned int	32 位

哪种类型最高效，是 char、short、int、unsigned int、unsigned long?

我们知道 RISC 系统中，所有运算都是在 32 位的寄存器中完成的。因此 8 位和 16 位变量要转换成 32 位才能运算。
而 8 位和 16 位的值域是有所不同的，比如说
char i=-47
在 8 位中表示为：11010001
在 16 位中表示为：11111111　11010001
在 32 位寄存器中表示为：11111111　11111111　11111111　11010001。

因此有一种说法：
8 位和 16 位的变量，在做完加法操作后，需要在 32 位的寄存器中进行符号扩展，其中带符号的变量，要用逻辑左移（LSL）接算术右移（ASR）两条指令才能完成符号扩展；无符号的变量，要使用一条逻辑与（AND）指令对符号位进行清零。
例如：

int a;
a=a+1
ADD a1,a1,#1
	short a;
	a=a+1
	ADD a1,a1,#1
	MOV a1,a1,LSL #16
	MOV a1,a1,ASR #16
		char a;
		a=a+1
		ADD a1,a1,#1
		AND a1,a1,#&ff

从这个角度上来讲，得出 int、unsigned int、long 等 32 位的数据类型效率最高。

我专门为此做了测试：

```
void printtime(struct timeval a1,struct timeval a2)
{
        long time = 1000000*(a2.tv_sec-a1.tv_sec)+a2.tv_usec-a1.tv_usec;
        printf("loop1 time:%u\n",time);
}
```

```
int main()
{
	char p=1;
	int i=0;

	struct timeval a1,a2;
	gettimeofday(&a1,NULL);
	for(i=0;i<1024*1024;i++)
		p++;
	gettimeofday(&a2,NULL);
	printtime(a1,a2);


	short s1=2;
	gettimeofday(&a1,NULL);
	for(i=0;i<1024*1024;i++)
		s1++;
	gettimeofday(&a2,NULL);
	printtime(a1,a2);

	int n1=3;
	gettimeofday(&a1,NULL);
	for(i=0;i<1024*1024;i++)
		n1++;
	gettimeofday(&a2,NULL);
	printtime(a1,a2);

	unsigned int ui=4;
	gettimeofday(&a1,NULL);
	for(i=0;i<1024*1024;i++)
		ui++;
	gettimeofday(&a2,NULL);
	printtime(a1,a2);

	long l1=5;
	gettimeofday(&a1,NULL);
	for(i=0;i<1024*1024;i++)
		l1++;
	gettimeofday(&a2,NULL);
	printtime(a1,a2);

	return 0;
}
```

运行结果：
loop1 time:128052
loop1 time:127870
loop1 time:127685
loop1 time:126831
loop1 time:127380

相差无几，与上面的推论完全不符，为什么呢？
我们可以通过反汇编来详细看一下内部结构

char	unsigned char

char p=0;
p++;
 	
mov r3, #0
strsb r3, [fp, #-13]
ldrsb r3, [fp, #-13]
add r3, r3, #1
strsb r3, [fp, #-13]
	short s1=1;
	s1++;
	
	mov r3, #1
	strb r3, [fp, #-13]
	ldrb r3, [fp, #-13]
	add r3, r3, #1
	strb r3, [fp, #-13]

我们可以看到对于 char 和 unsigned char，其存取寄存器的指令是不同的。

ldrb 指令用于从存储器中将一个 8 位的字节数据传送到目的寄存器中，同时将寄存器的高 24 位清零。
ldrsb 指令用于从存储器中将一个 8 位的字节数据传送到目的寄存器中，同时将符号位扩展到高 24 位。

对与 short 类型，同样对应着 ldrh 和 ldrsh。

而对于将 8 位和 16 位数据从寄存器写到对应的内存中，则不存在符号位扩展的问题。

针对 8 位和 16 位，ARM 是在将数据加载到寄存器中，完成符号位扩展，而 ldr、ldrsb、ldrb、lsdh、lsdsh、ldr 等其指令周期是一样的，所以 char、short、int 其效率是相同的。

3.12.2. 慢操作

3.12.3. if 与 switch 的性能比较

我们通过范例来比较 if 和 switch 的性能。

if	switch

if(a==0)
{
}elseif(a==1)
{
}elseif(a==2)
{
}elseif(a=3)
{
}elseif(a==4)
{
}else
{
}
	switch(a)
	{
	case 0:
	b=1;
	break;
	case 2:
	b=3;
	break;
	case 1:
	b=2;
	break;
	default:
	b=6;
	break;
	}

我们通常认为，在 switch 中，程序是按照 case 的值挨个比较执行的，如果运行到 default，则需要 3 次比较；那么我们将最有可能发生的放在第一个，可以提高性能。

可实际上呢？GCC 对于 switch 做了相关的优化。
假设我们程序的源代码是下面的样子：

```
switch(a)
{
    case 1:…….;
    case 2:…….;
    case 3:…….;
    case 4:…….;
    case 5:…….;
    case 6:…….;
    case 7:…….;
    case 8:…….;
    case 9:…….;
    case 10:…….;
    case 11:…….;
}
```

该代码所对应的非优化逻辑树的高度为 11 个节点（左边的树）。该树根的节点左分支含有 10 个子节点，而右分支只是 case 的处理代码。如果我们能够从 case 的中间值开始比较，那么就可以降低逻辑树的高度，从而提高 switch 的搜索效率。

```
switch(a)
{
case 0:
```

```
            b=1;
            break;
    case 2:
            b=3;
            break;
    case 1:
            b=2;
            break;
    default:
            b=6;
            break;
    }
```

gcc –S a.c

```
            mov     r3, #0
            str     r3, [fp, #-16]
            ldr     r3, [fp, #-16]
            str     r3, [fp, #-24]
            ldr     r3, [fp, #-24]
            cmp     r3, #1
            beq     .L4
            ldr     r3, [fp, #-24]
            cmp     r3, #1
            bgt     .L6
            ldr     r3, [fp, #-24]
            cmp     r3, #0
            beq     .L3
            b       .L2
.L6:
            ldr     r3, [fp, #-24]
            cmp     r3, #2
            beq     .L5
            b       .L2
.L3:
            mov     r3, #1
            str     r3, [fp, #-20]
            b       .L2
.L4:
            mov     r3, #2
            str     r3, [fp, #-20]
            b       .L2
.L5:
            mov     r3, #3
            str     r3, [fp, #-20]
```

从上面代码我们可以看到，并不是 switch 的第一个分支效率最高，而是中间值的效率最高。
实际上，switch 除了逻辑树外，我们还可以使用跳转表来进一步优化。
如果 case 值变化不大（太大的话，会导致跳转表过大，造成内存的浪费），有足够的分支（大于 4 个分支），跳转表会带来性能极大的提升。

```
    switch(a)
    {
    case 0:
            b=1;
            break;
    case 2:
            b=3;
            break;
    case 1:
            b=2;
            break;
    case 3:
            b=4;
            break;
    default:
            b=6;
            break;
    }

            mov         r3, #0
            str         r3, [fp, #-16]
            ldr         r3, [fp, #-16]
            cmp         r3, #3
            ldrls       pc, [pc, r3, asl #2]
            b           .L7
            .p2align 2
.L8:
            .word       .L3
            .word       .L5
            .word       .L4
            .word       .L6
.L3:
            mov         r3, #1
            str         r3, [fp, #-20]
            b           .L2
.L4:
            mov         r3, #3
            str         r3, [fp, #-20]
            b           .L2
```

```
.L5:
        mov     r3, #2
        str     r3, [fp, #-20]
        b       .L2
.L6:
        mov     r3, #4
        str     r3, [fp, #-20]
        b       .L2
.L7:
        mov     r3, #6
        str     r3, [fp, #-20]
.L2:
        mov     r0, #0
        sub     sp, fp, #12
        ldmfd   sp, {fp, sp, pc}
```

从上面可知，在跳转表实现的 switch 中，default 值的效率是最高的。

3.12.4. 循环

循环是我优化的重点，大量的重复执行，造成程序的局部热点，一丁点的优化往往带来性能的巨大提升。
我们知道现在的处理器很多都采用了流水线技术。
ARM7 提供 3 级流水线；ARM9、ARM9E 系列采用了 5 级整数流水线；ARM10E 提供 6 级整数流水线；ARM11 提供了 8 级流水线。
流水线级数越高，对分支操作越加敏感。在程序正常运行期间处理当前代码时，预取模块有时间去读取并解码下一部分指令而不使存储总线闲置下来。如果遇到分支语句，那么它们通过清除流水线而立刻使预先进行的读取和解码工作无效。流水线级数越多，系统就花费更多的时间来填充流水线，在这期间 CPU 空闲下来。因此，分支语句越少，性能越高。

循环的展开
通过循环分支的展开，我们可以降低循环的次数，从而减少分支语句对循环的影响。

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>

void printtime(struct timeval a1,struct timeval a2)
{
        long time = 1000000*(a2.tv_sec-a1.tv_sec)+a2.tv_usec-a1.tv_usec;
        printf("loop time:%u\n",time);
}

int main()
{
```

```
int n=0,n1;
struct timeval a1,a2;
gettimeofday(&a1,NULL);
for(n = 0; n <1024*1024 ;n++)
{
        n1++;
}
gettimeofday(&a2,NULL);
printtime(a1,a2);

gettimeofday(&a1,NULL);
for(n = 0; n <1024*512 ;n++)
{
        n1++;
        n1++;
}
gettimeofday(&a2,NULL);
printtime(a1,a2);

gettimeofday(&a1,NULL);
for(n = 0; n <1024*256 ;n++)
{
        n1++;
        n1++;
        n1++;
        n1++;
}
gettimeofday(&a2,NULL);
printtime(a1,a2);

gettimeofday(&a1,NULL);
for(n = 0; n <1024*128 ;n++)
{
        n1++;
        n1++;
        n1++;
        n1++;
        n1++;
        n1++;
        n1++;
        n1++;
}
gettimeofday(&a2,NULL);
printtime(a1,a2);
```

```
        gettimeofday(&a1,NULL);
        for(n = 0; n <1024*64 ;n++)
        {
                n1++;
                n1++;
                n1++;
                n1++;
                n1++;
                n1++;
                n1++;
                n1++;
                n1++;
                n1++;
                n1++;
                n1++;
                n1++;
                n1++;
                n1++;
                n1++;
        }
        gettimeofday(&a2,NULL);
        printtime(a1,a2);
        return 0;
}
```

```
loop time:103576
loop time:41505
loop time:31708
loop time:16783
loop time:14985
```

从运行结果上看，通过循环展开，对性能提高明显；但随着展开的指令增加，性能的收益递减，其代价是代码尺寸的增加。

循环的合并
如果两个循环具有同样的循环首部，那么将它们组合成一个共同的循环是可能的。例如：

```
for(b=0;b<10;b++)
    m[b]=b;

for(b=0;b<10;b++)
    n[b]=b;
```

可以将两个循环合并到一起，既可以提高程序的性能并减少代码尺寸，同时不影响功能。

```
for(b=0;b<10;b++)
{
    m[b]=b;
    n[b]=b;
}
```

用减 1 指令替换循环加 1 指令

```
        gettimeofday(&a1,NULL);
        for(n = 0; n <1024*1024 ;n++)
        {
                n1++;
        }
        gettimeofday(&a2,NULL);
        printtime(a1,a2);

        n1=1024*1024;
        gettimeofday(&a1,NULL);
        for(n = 1024*1024; n !=0 ;n--)
        {
                n1--;
        }
        gettimeofday(&a2,NULL);
        printtime(a1,a2);
```

```
loop time:127777
loop time:63751
```

while、do while、for 循环比较

```
        int n=0,n1;
        struct timeval a1,a2;
        gettimeofday(&a1,NULL);
        for(n = 0; n <1024*1024 ;n++)
        {
                n1++;
        }
        gettimeofday(&a2,NULL);
        printtime(a1,a2);

        n1=0;
        n=0;
        gettimeofday(&a1,NULL);
        while(n<1024*1024)
        {
            n1++;
```

```
        n++;
    }
    gettimeofday(&a2,NULL);
    printtime(a1,a2);

    n1=0;
    n=0;
    gettimeofday(&a1,NULL);
    do{
        n1++;
        n++;
    }while(n<1024*1024);
    gettimeofday(&a2,NULL);
    printtime(a1,a2);
```

```
loop1 time:126648
loop1 time:127349
loop1 time:127562
```

效率上都差不多。

3.12.5. 函数

关于纯函数和常函数：

纯函数（pure function）是指不会影响它自己范围外任何事情的函数。它可以读取全局变量或者通过指针传递的变量，但是不可以对这些变量进行些操作。它也不可以读取 volatile 变量和外部资源（比如文件）。“函数的结果只依赖参数”。

常函数（const function）是纯函数的更严格版本。它不会读取和写除参数外的任何数据，也不可以使用指针参数来读取数据。“函数的结果依赖于参数和全局及内存变量”。

对于纯函数和常函数，如果返回值为 void 类型，则没有任何意义。GCC 可以通过这些信息来进行 CSE 优化，比如对于常函数，如果参数在多次调用时不变，则 GCC 将替换后面的函数调用为前面已执行函数的返回值。

函数属性是 C 语言的 GNU 扩展，允许程序员为 GCC 提供更多的信息。它作为函数声明的一部分，通过关键字__attribute__来制定。比如，int func(int a, int b) __attribute__ (pure);。一个函数可以具有多个属性。

现在针对函数调用，ARM 体系中存在两个标准：AAPCS（Procedure Call Standard for the ARM Architecture）和 ATPCS（ARM-Thumb Procedure Call Standard）。

AAPCS
AAPCS 定义了程序如何单独编写，单独编译，协同工作。它阐述了子程序见调用规定的规范。而 ATPCS 关于于 ARM 程序和 Thumb 程序中子程序调用的基本规则。

ATPCS
为了使单独编译的 C 语言程序和汇编程序之间能够相互调用，必须为子程序间的调用规定一定的规则。ATPCS 规定了一些子程序间的调用的基本规则。这些基本规则包括子程序调用过程中寄存器的使用规则，数据栈的使用规则，参数的传递规则。一般情况下，编译出来的程序都符合该规则。
寄存器使用规则：
1、 子程序间通过寄存器 R0～R3 来传递参数。被调用的子程序在返回前无需恢复寄存器 R0～R3 的内容。
2、 在子程序中，使用寄存器 R4～R11 来保存局部变量。如果在子程序中使用了寄存器 R4～R11 中的某些寄存器，子程序进入时必须保存这些寄存器的值，在返回前必须恢复这些寄存器的值；对于子程序中没有用到的寄存器则不必进行这些操作。
3、 R12 用作子程序间 scratch 寄存器，记作 ip。
4、 R13 用作数据栈指针，记作 sp。
5、 R14 称为连接寄存器，记作 lr。它用来保存子程序的返回地址。
6、 R15 是程序计数器，记作 pc。
7、 子程序返回结果为一个 32 位整数时，可以通过寄存器 R0 返回；结果为一个 64 位整数时，可以通过寄存器 R0 和 R1 返回，依次类推。

函数的优化：
函数的参数最好不多于 4 个

4 个以下的形参可以通过寄存器来传递；4 个以上的参数，则要通过栈来传递。如果有更多的参数调用，则可将相关的参数组织在一个结构体内，用传递结构体指针来代替参数。
在 C++中，成员函数的参数个数位最好小于等于 3 个，因为对象指针本身占用了一个参数。

减少局部变量的个数
1、 尽量限制函数内部循环所用局部变量的数目，最多不超过 12 个，以便编译器能把变量分配到寄存器。
2、 如果没有局部变量保存到栈中，系统也将不必设置和恢复栈指针。

3. 关于函数参数类型
如果不需要返回值，则尽量定义为 void，这样可以节省一个寄存器 R0。
如果需要返回值，则最好限定在 4 个字节，使用一个寄存器 R0 即可。

我们可以看个例子：
```
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
```

```
int func(int a,int b)
{
	int c=a+1;
	int d=b+2+c;
	printf("%d %d\n",c,d);
	return d;
}
int main()
{
	int a=0;
	int b=2;
	func(a,b);
	return 0;
}
```

当我们使用 gcc a.c –o hello

func 对应的代码：

```
    8414:	e1a0c00d	mov	ip, sp
    8418:	e92dd800	stmdb	sp!, {fp, ip, lr, pc}
    841c:	e24cb004	sub	fp, ip, #4	; 0x4
    8420:	e24dd010	sub	sp, sp, #16	; 0x10
    8424:	e50b0010	str	r0, [fp, #-16]
    8428:	e50b1014	str	r1, [fp, #-20]
    842c:	e51b3010	ldr	r3, [fp, #-16]
    8430:	e2833001	add	r3, r3, #1	; 0x1
    8434:	e50b3018	str	r3, [fp, #-24]
    8438:	e51b2014	ldr	r2, [fp, #-20]
    843c:	e51b3018	ldr	r3, [fp, #-24]
    8440:	e0823003	add	r3, r2, r3
    8444:	e2833002	add	r3, r3, #2	; 0x2
    8448:	e50b301c	str	r3, [fp, #-28]
    844c:	e59f0018	ldr	r0, [pc, #24]	; 846c <.text+0x100>
    8450:	e51b1018	ldr	r1, [fp, #-24]
    8454:	e51b201c	ldr	r2, [fp, #-28]
    8458:	ebffffc0	bl	8360 <__plt_header+0x38>
    845c:	e51b301c	ldr	r3, [fp, #-28]
    8460:	e1a00003	mov	r0, r3
    8464:	e24bd00c	sub	sp, fp, #12	; 0xc
    8468:	e89da800	ldmia	sp, {fp, sp, pc}
    846c:	00008570	andeq	r8, r0, r0, ror r5
```

当我们使用 gcc –O2 a.c –o hello

func 对应的代码：

```
    8414:	e2803001	add	r3, r0, #1	; 0x1
    8418:	e0810003	add	r0, r1, r3
```

```
841c:       e92d4010        stmdb   sp!, {r4, lr}
8420:       e2804002        add     r4, r0, #2      ; 0x2
8424:       e1a01003        mov     r1, r3
8428:       e1a02004        mov     r2, r4
842c:       e59f0008        ldr     r0, [pc, #8]    ; 843c <.text+0xd0>
8430:       ebfffca         bl      8360 <__plt_header+0x38>
8434:       e1a00004        mov     r0, r4
8438:       e8bd8010        ldmia   sp!, {r4, pc}
843c:       00008524        andeq   r8, r0, r4, lsr #10
```

把本地函数声明为静态的(static)
如果一个函数在实现它的文件外未被使用的话，把它声明为静态的(static)以强制使用内部连接。否则，默认的情况下会把函数定义为外部连接。这样可能会影响某些编译器的优化——比如，自动内联。

3.12.6. 数组

尽量使用数组的大小是 4 或 8 的倍数，用此倍数展开循环体
一维数组和多维数组

3.12.7. 浮点

从 ARM11 开始，浮点运算是一个可供用户选择的设计。用户可以在向 ARM 要求授权的时候选择是否包含浮点处理器的内核。
当 ARM 没有浮点处理器，需要一个模拟浮点机制，在配置内核时一定要选择一个浮点模拟器 NWFPW。
--- At least one math emulation must be selected | |
| | <*> NWFPE math emulation | |
| | [ ] Support extended precision | |
| | < > FastFPE math emulation (EXPERIMENTAL)

NWFPE 模拟浮点是利用了 undefined instruction handler 机制，即每次浮点指令操作，都会发生一次未定义的指令异常。在这个异常的 handler 里面，用软件的方法仿真一个浮点指令。如果软件里的浮点运算比较多，那就不要不停的请求 CPU 执行 undefined instruction，产生异常，大大增加了终端延迟，对性能的开销极大。

在 glibc 库中提供了软浮点。当使用软浮点工具链编译浮点操作时，编译器会用内联的浮点库替换掉浮点操作，使得生成的机器码完全不含浮点指令，但是又能完成正确的浮点操作，由于是在编译时优化，这种方式能够让 CPU 即使做浮点运算时也能够执行连续的指令，使用软浮点工具链编译产生的浮点操作要比 NWFPE 模拟快一个数量级。

使用软浮点。
gcc test.c –msoft-float –o test

即使使用软浮点，速度还是很慢，我们可以采用将浮点运算转换成乘法和除法。
例如
i*0.4　我们可以转发为 i*4/10，可以转化为 i<<2/10。

我们怎么知道交叉工具链是否支持软浮点呢？
很简单，使用编译命令时加上-msoft-float 这个 CFLAGS 就能够了，比如
arm-linux-gcc test。c -msoft-float -o test
假如工具链支持软浮点，那么就能够生成可执行文档
如出错，对不起，该工具链不支持
另一个鉴定的方法，拿到一个编译产生的文档，使用
arm-linux-readelf -e test
能够打印出 elf 文档头信息
在里面看到 software FP 就是软浮点格式，否则不是
Flags: 0x202, has entry point, GNU EABI, software FP

对于计算密集型的算法，我们可以通过转化为查表来进行优化。

如何查看 CPU 是否支持浮点运算
cat cpuinfo
Processor : Some Random V6 Processor rev 1 (v6l)
BogoMIPS : 88.66
Features : swp half thumb fastmult vfp edsp java
CPU implementer : 0x41
CPU architecture: 6TEJ
CPU variant : 0x1
CPU part : 0xb36
CPU revision : 1
Cache type : write-back
Cache clean : cp15 c7 ops
Cache lockdown : format C
Cache format : Harvard
I size : 16384
I assoc : 4
I line length : 32
I sets : 128
D size : 16384
D assoc : 4
D line length : 32
D sets : 128

Hardware : Motorola Product - SCM-A11 Phone

Revision : 0021
Serial : 0000000000000000

在 Features 中，我们可以看到 vfp，即 Vector Floating-Point，说明该 CPU 支持浮点运算。

3.12.7.1. 软浮点与硬浮点

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>

void printtime(struct timeval a1,struct timeval a2)
{
        long time = 1000000*(a2.tv_sec-a1.tv_sec)+a2.tv_usec-a1.tv_usec;
        printf("loop1 time:%u\n",time);
}

int main()
{
    int i=0;
    float fret=0.058;

    struct timeval a1,a2;
    gettimeofday(&a1,NULL);

    for(i=0;i<1000000;i++)
    {
        fret+=0.23;
    }
    gettimeofday(&a2,NULL);
    printtime(a1,a2);
    return 0;
}
```

硬浮点：

```
#gcc -o hfloat f1.c
# ./hfloat
loop1 time:368196
```

软浮点

```
#gcc -o sfloat -msoft-float f1.c
# ./sfloat
loop1 time:1292024
```

从上面的结果可以看出，软浮点的加法要比硬浮点慢了将近 4 倍。

前面我们提到，对于浮点运算，我们可以采用将其转化为乘除运算来进行优化。那么如果我们拥有了浮点协处理器的话，两者的性能会将如何呢？

在这里我们看个例子：

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>

void printtime(struct timeval a1,struct timeval a2)
{
        long time = 1000000*(a2.tv_sec-a1.tv_sec)+a2.tv_usec-a1.tv_usec;
        printf("loop1 time:%u\n",time);
}

void floattest(int n)
{
    float fret=0.058;

    int i=0;
    for(i=0;i<1000000;i++)
    {
        fret+=n*0.8;
    }
    printf("%f\n",fret);
}

void divtest(int n)
{
    float fret=0.058;

    int i=0;
    for(i=0;i<1000000;i++)
    {
        fret+=n*4/5;
    }
    printf("%f\n",fret);
}

int main()
{
    struct timeval a1,a2;
    gettimeofday(&a1,NULL);
```

```
    floattest(20);
    gettimeofday(&a2,NULL);
    printtime(a1,a2);

    gettimeofday(&a1,NULL);
    divtest(20);
    gettimeofday(&a2,NULL);
    printtime(a1,a2);

    return 0;
}
```

其运算结果：

```
# ./f1
16000000.000000
loop1 time:471741
16000000.000000
loop1 time:445588
```

从结果上我们可以看出，即使是拥有了浮点协处理器，将浮点运算转换为整数的乘除运算仍然具有更高的性能。

3.12.8. 结构体

◎ 小的元素放在结构体的开始，大的元素放在结构体的最后
◎ 避免使用过大的结构体，用层次话的小结构体代替。
◎ 人工对 API 的结构体增加填充位以提高移植性。
◎ 枚举类型要慎用，因为它的大小与编译器相关。

跳转预测及管理

跳转指令通常都是条件执行的。问题在于那些决定是否跳转的条件要在跳转指令被译码的后 3~4 个周期才能就绪。如果不做特殊处理，跳转指令必须等待，这样使指令执行效率变得让人难以忍受。跳转预测就是用来帮助解决这种延迟的。

ARM11 处理器提供两种技术来对跳转作出预测—动态预测和静态预测。

动态预测：在 ARM11 处理器中包含了 64 个 4 状态跳转地址缓存器（4-state branch target address cache）来保存最近发生的跳转指令的结果。通过对这些历史纪录的查找，处理器可以预测出当前的跳转指令是否会被执行。

静态预测：当在动态预测的缓存器中无法查到和当前指令匹配的记录，ARM11 处理器就从跳转的方式来判断是否执行。如果是向回跳转，大多数情况是遇到一个循环，处理器会假设这条指令被执行。如果是向前跳转，处理器会假设这条指令不被执行。

动态预测和静态预测的组合使 ARM11 处理器能达到 85%的预测正确性，对于每一个正确的预测，给指令执行带来的是减少 5 个时钟周期的等待时间。

3.12.9. 除法

在 ARM 指令集中没有除法指令，其除法是通过 C 库函数实现的，根据执行情况和输入操作数的范围，一个 32 位的除法通常需要 20～140 个时钟周期。因此，除法和模运算成了一个程序效率的瓶颈，应该避免使用。

1、 当除数为 2 的幂次方时，使用移位操作代替除法。

2、 如果不能避免除法运算，那么就应该使除数和被除数是无符号整数。有符号的整数除法执行起来更慢，因为它们首先要取得除数和被除数的绝对值，再调用无符号除法运算，最后再确定结果的符号。

① 比如 n/2 写为 n>>1 这个是常用的方法，不过要注意的是这两个不是完全等价的！因为：如果 n＝3 的话，n/2=1;n>>1=1;但是，如果 n＝-3 的话，n/2=-1;n>>1=-2 所以说在正数的时候，他们都是向下取整，但是负数的时候就不一样了。（在 JPG2000 中的整数 YUV 到 RGB 变换一定要使用>>来代替除法就是这个道理）

x86 系列的 CPU 对于位运算、加、减等基本指令都能在 1 个 CPU 周期内完成(现在的 CPU 还能乱序执行，从而使指令的平均 CPU 周期更小)；现在的 CPU,做乘法也是很快的(需要几个 CPU 周期，每个周期可能启动一个新的乘指令(x87))，但作为基本指令的除法却超出很多人的预料，它是一条很慢的操作，整数和浮点的除法都慢；我测试的英特尔 P5 赛扬 CPU 浮点数的除法差不多是 37 个 CPU 周期，整数的除法是 80 个 CPU 周期，AMD2200+浮点数的除法差不多是 21 个 CPU 周期，整数的除法是 40 个 CPU 周期。(改变 FPU 运算精度对于除法无效) (SSE 指令集的低路单精度数除法指令 DIVPS 18 个 CPU 周期,四路单精度数除法指令 DIVSS 36 个 CPU 周期)　(x86 求余运算和除法运算是用同一条 CPU 指令实现的；据说，很多 CPU 的整数除法都是用数学协处理器的浮点除法器完成的；有一个推论就是，浮点除法和整数除法不能并行执行.　(ps:intel 的 p4 imul 指令可能有 14 周期(或 15-18)的延迟才能得到结果)

本文将给出一些除法的优化方法或替代算法　(警告:某些替代算法并不能保证完全等价!)

1.尽量少用除法

比如：　if (x/y>z) ...

改成：　if (((y>0)&&(x>y*z))||((y<0)&&(x<y*z))) ...

(少用求余)

比如：　++index; if (index>=count) index=index % count;　//assert(index<count*2);

改成：　++index; if (index>=count) index=index - count;

2.用减法代替除法

如果知道被除数是除数的很小的倍数，那么可以用减法来代替除法

比如：　uint32 x=200;

uint32 y=70;

uint32 z=x/y;

改成：　uint z=0;

while (x>=y)

{

```
            x-=y;    ++z;
        }
```

一个用减法和移位完成的除法 (如果你没有除法指令可用:)

```
uint32 div(uint64 u,uint32 z) //return u/z
{
    uint32 x=(uint32)(u>>32);
    uint32 y=(uint32)u;
    //y 保存商  x 保存余数

    for (int i=0;i<32;++i)
    {
        uint32 t=((int32)x) >> 31;
        x=(x<<1)|(y>>31);
        y=y<<1;
        if ((x|t)>=z)
        {
            x-=z;
            ++y;
        }
    }
    return y;
}
```

(该函数经过了测试; z==0 需要自己处理; 对于有符号除法，可以用取绝对值的方法(当然也不是轻松就能
写出完全等价的有符号除法的:)； 如果不需 s 的 64bit 长度，仅需要 32bit,那么可以化简这个函数，但改进不多)

3.用移位代替除法 (很多编译器能自动做好这个优化)

要求除数是 2 的次方的常量； (同理：对于某些应用，可以优先选取这样的数来做除数)

比如：
```
uint32 x=213432575;
uint32 y=x/8;
```
改成：
```
y=x>>3;
```

对于有符号的整数；

比如：
```
int32 x=213432575;
int32 y=x/8;
```
改成：
```
if (x>=0)    y=x>>3;
else    y=(x+(1<<3-1))>>3;
```

4.合并除法 (替代方法不等价，很多编译器都不会帮你做这种优化)

适用于不能用其它方法避免除法的时候；

比如：
```
double x=a/b/c;
```
改成：
```
double x=a/(b*c);
```

比如： double x=a/b+c/b;
改成： double x=(a+c)/b;

比如： double x=a/b;
double y=c/d;
double z=e/f;
改成： double tmp=1.0/(b*d*f);
double x=a*tmp*d*f;
double y=c*tmp*b*f;
double z=e*tmp*b*d;

5.把除法占用的时间充分利用起来
CPU 在做除法的时候，可以不用等待该结果(也就是后面插入的指令不使用该除法结果)，而插入多条简单整数
指令(不包含整数除法，而且结果不能是一个全局或外部变量等)，把除法占用的时间节约出来；
(当除法不可避免的时候，这个方法很有用)

6.用查表的方法代替除法
（适用于除数和被除数的可能的取值范围较小的情况，否则空间消耗太大）
比如 uint8 x; uint8 y;
uint8 z=x/y;
改成 uint8 z=table[x][y]; //其中 table 是预先计算好的表，table[i][j]=i/j;
//对于除零的情况需要根据你的应用来处理
或者：uint8 z=table[x<<8+y]; //其中 table[i]=(i>>8)/(i&(1<<8-1));

比如 uint8 x;
uint8 z=x/17;
改成 uint8 z=table[x]; //其中 table[i]=i/17;

7.用乘法代替除法
(替代方法不等价，很多编译器都不会帮你做这种优化)
比如： double x=y/21.3432575;
改成： double x=y*(1/21.3432575); //如果编译器不能优化掉(1/21.3432575),请预先计算出该结果

对于整数，可以使用与定点数相似的方法来处理倒数
(该替代方法不等价)
比如： uint32 x,y; x=y/213;
改成： x=y*((1<<16)/213) >> 16;
某些应用中 y*((1<<16)/213)可能会超出值域，这时候可以考虑使用 int64 来扩大值域
uint32 x=((uint64)y)*((1<<31)/213) >> 31;
也可以使用浮点数来扩大值域

uint32 x=(uint32)(y*(1.0/213)); (警告: 浮点数强制类型转换到整数在很多高级语言里都是
一条很慢的操作，在下几篇文章中将给出其优化的方法)

对于这种方法，某些除法是有与之完全等价的优化方法的：
无符号数除以 3： uint32 x,y; x=y/3;
推 理 : y y y (1<<33)+1 y
x=y/3 => x=[-] => x=[- + ---------] => x=[--------- * -------] // []表示取整
3 3 3*(1<<33) 3 (1<<33)
y 1
(可证明: 0 <= --------- < -)
3*(1<<33) 3
即 ： x=(uint64(y)*M)>>33; 其 中 魔 法 数 M=((1<<33)+1)/3=2863311531=0xAAAAAAAB;

无符号数除以 5： uint32 x,y; x=y/5; (y<(1<<31))
推 理 : y y 3*y (1<<33)+3 y
x=y/5 => x=[-] => x=[- + ---------] => x=[--------- * -------]
5 5 5*(1<<33) 5 (1<<33)
3*y 1
(可证明: 0 <= --------- < -)
5*(1<<33) 5
即 ： x=(uint64(y)*M)>>33; 其 中 魔 法 数 M=((1<<33)+3)/5=1717986919=0x66666667;

无符号数除以 7： uint32 x,y; x=y/7;
推理 ：略
即 ： x=((uint64(y)*M)>>33+y)>>3; 其 中 魔 法 数 M=((1<<35)+3)/7-(1<<32)=613566757=0x24924925;

对于这种完全等价的替代,还有其他替代公式不再讨论，对于有符号除法的替代算法没有讨论，很多数都有完全等价的替代算法，那些魔法数也是有规律可循的；有“进取心”的编译器应该帮助用户处理掉这个优化(vc6 中就已经见到! 偷懒的办法是直接看 vc6 生成的汇编码:)；

8.对于已知被除数是除数的整数倍数的除法，能够得到替代算法；或改进的算法；
这里只讨论除数是常数的情况；
比如对于(32 位除法)：x=y/a; //已知 y 是 a 的倍数,并假设 a 是奇数
(如果 a 是偶数,先转化为 a=a0*(1<<k); y/a==(y/a0)>>k;a0 为奇数)
求得 a',使 (a'*a) % (1<<32) = 1；

那么: x=y/a => x=(y/a)*((a*a') %(1<<32)) => x=(y*a') % (1<<32); //这里并不需要实际做一个求余指令

(该算法可以同时支持有符号和无符号除法)

比如 uint32 y;

uint32 x=y/7; //已知 y 是 7 的倍数

改成 uint32 x=(uint32)(y*M); //其中 M=(5*(1<<32)+1)/7

9.近似计算除法 (该替代方法不等价)

优化除数 255 的运算(257 也可以,或者 1023,1025 等等)(1026 也可以,推导出的公式略有不同)

比如颜色处理中 ： uint8 color=colorsum/255;

改成： uint8 color=colorsum/256; 也就是 color=colorsum>>8;

这个误差在颜色处理中很多时候都是可以接受的

如果要减小误差可以改成: uint8 color=(colorsum+(colorsum>>8))>>8;

推导: x/255=(x+x/255)/(255+1)=(x+A)>>8; A=x/255;

把 A 改成 A=x>>8 (引入小的误差);带入公式就得到了: x/255 约等于(x+(x>>8))>>8 的公式

同理可以有 x/255 约等于(x+(x>>8)+(x>>16))>>8 等其它更精确的公式(请推导出误差项已确定是否精度足够)

10. 牛顿迭代法实现除法

(很多 CPU 的内部除法指令就是用该方法或类似的方法实现的)

假设有一个函数 y=f(x); 求 0=f(x)时，x 的值;(这里不讨论有多个解的情况或逃逸或陷入死循环或陷入混沌状态的问题)

(参考图片)

求函数的导函数为 y=f'(x); //可以这样来理解这个函数的意思：x 点处函数 y=f(x)的斜率；

a.取一个合适的 x 初始值 x0; 并得到 y0=f(x0);

b.过(x0,y0)作一条斜率为 f'(x0)的直线，与 x 轴交于 x1;

c.然后用得到的 x1 作为初始值，进行迭代；

当进行足够多次的迭代以后，认为 x1 将会非常接近于方程 0=f(x)的解，这就是牛顿迭代；

把上面的过程化简,得到牛顿迭代公式: x(n+1)=x(n)-y(x(n))/y'(x(n))

这里给出利用牛顿迭代公式求倒数的方法； (用倒数得到除法: y = x/a = x* (1/a))

求 1/a,

令 a=1/x; 有方程 y=a-1/x;

求导得 y'=1/x^2;

代入牛顿迭代方程 x(n+1)=x(n)-y(x(n))/y'(x(n));

有迭代式 x_next=(2-a*x)*x; //可证明：该公式为 2 阶收敛公式； 也就是说计算出的解的有效精度成倍增长；

证明收敛性:令 x=(1/a)+dx; //dx 为一个很小的量

则有 x_next-(1/a)=(2-a*(1/a+dx))*(1/a+dx)-1/a
=(-a)*dx^2 //^表示指数运算符
证毕.

程序可以用该方法来实现除法，并按照自己的精度要求来决定迭代次数；
(对于迭代初始值，可以使用查表法来得到,或者利用 IEEE 浮点格式得到一个近似计算的表达式;在 SSE 指令集中有一条 RCPPS(RCPSS)指令也可以用来求倒数的近似值；有了初始值就可以利用上面的迭代公式得到更精确的结果)

附录: 用牛顿迭代法来实现开方运算
//开方运算可以表示为 y=x^0.5=1/(1/x^0.5)； 先求 1/x^0.5
求 1/a^0.5,
令 a=1/x^2; 有方程 y=a-1/x^2;
求导得 y'=2/x^3;
代入牛顿方程 x(n+1)=x(n)-y(x(n))/y'(x(n));
有迭代式 x_next=(3-a*x*x)*x*0.5; //可证明：该公式为 2 阶收敛公式 //方法同上 证明过程略

3.12.10. inline 内联函数

在调用函数时，进程需要对栈进程操作、保存运行环境，代价比较大。为了节省函数进出栈，所花费的时间，我们可以使用宏和 inline 函数，在函数调用的地方进行展开。由于宏不具备类型检查等功能，很容易出现问题，所以目前比较推崇的是使用 inline 函数。

```
inline int halve(double x)
{
    return(x / 2.0);
}
```

既然内联代码有上面的好处，为什么不都采用内联函数呢？
天下没有免费的午餐，内联函数的引入会导致目标代码迅速的增大，使程序的代码段占据大量的物理内存。并且由于内存代码使用内存的增长，会导致触发页故障增大和降低高速缓存的命中率，而使代码的运行速度减慢。

用法相信大家都已经知道了，我在这里说几个注意事项：
1、 为了调试方便，当函数在编译时，没有采用-O 指定优化级别，内联函数不做展开，除非用户使用 always_inline，来声明函数。
2、 编译器展开内联函数的前提是，在同一个编译单元能够找到该函数的定义，否则就不能展开内联函数。因此，内联函数的声明和定义一般都包含在头文件中。
3、 inline 修饰符并非强制性的，编译器有可能会对它置之不理。例如，递归函数通常不会被编译成 inline 函数，编译器有权自行决定是否要将定义成 inline 的函数编译成 inline。
4、 展开 inline 函数并不一定会导致代码量增大。因为在函数调用时，需要使用一些指令来安排进出函数时，栈结构的变化；而如果 inline 函数展开，则省去了这些指令。如果 inline

函数体的代码量小于安排栈帧的代码量，那么将函数标识为 inline，反而会使代码量减小。

5、 GCC 在编译时，如果使用了-O3 优化编译开关，其会将一些代码量小的函数转变成 inline 来处理，即使这个函数没有使用 inline 标识符。

如果是一个手写汇编的函数，那样的话很有可能破坏参数。gcc 里有强制不内联的，用法如下

void foo() __attribute__((noinline));

但是有的 gcc 可能会忽略 noinline。

那么你可以将你实现的这个函数写到调用函数之后，就不会被 inline 了。这是因为编译器 gcc 只内联当前函数之前可见(实现代码在前)的函数。

6、 在C++中，在类的内部定义了函数体的函数，被默认为是内联函数。而不管你是否有inline关键字。

```
Class TableClass{
private:
    int I,j;
public:
    int add() { return I+j;};
    inline int dec() { return I-j;}
}
```

3.12.11. C++

3.12.11.1. 在声明对象时进行初始化

对于以前定义的或者用户自定义的类对象，最好在声明的时候初始化：

例如

```
Myclass obj = data;
```

上面这种方法要比下面的方法更快，更加简洁

```
Myclass obj;
obj = data;
```

下面的这种方法，将首先调用对象的构造函数，构造出对象，然后调用类的赋值运算，需要两个操作；而对于第一个例子来讲，只需要调用构造函数，就可以完成对象的初始化工作。

3.12.11.2. 构造函数和析构函数

在我们编写代码中，经常会看到这样的情形。

```
class A;
……..
for（int i=0;….
{
      A a;
    ……..
}
```

我们在循环中声明了类对象，很多人会把它想象成 int a 一样，只是一个声明而已对性能不会有什么影响。

而实际上，在每一次 for 循环中，都要运行一次 A 的构造函数和析构函数。而如果 A 是一个派生类的话，那么 A 的构造函数将转变称为一个递归操作，在每层递归内部的操作都尊徐严格的次序。递归模式为首先执行父类的构造函数（父类的构造函数操作也相应的包括执行初始化和执行构造函数体两部分），父类构造函数返回后构造该类自己的成员变量。

析构函数的调用和构造函数的调用一样，也是类似的递归操作。但是有两点不同：一是析构函数没有与构造函数相对应的初始化操作部分；二是析构函数执行的递归与构造函数刚好相反，先析构子类，再析构父类。

如果 A 位于一个很复杂的类结构顶端，那么运行一次 A 的构造函数和析构函数将会花费很长的时间。而这一点，往往是我们所经常忽略的。

3.12.11.3. 构造函数初始化列表

我们来比较一下，下面两种构造函数的写法。

第一种：

```
class c1: public Base
{
public:
      c1():i(10)
      {
      ……
      }
private:
      int i;
}
```

第二种

```
class c1: public Base
{
public:
      c1()
      {
          i=10;
```

```
    }
private:
    int i;
}
```

在构造类 c1 时，分为两个阶段：一是初始化成员变量；二是运行构造函数。
在初始化成员变量时，又遵循着两条法则：一是严格按照成员变量在类中的声明顺序进行，而与其在初始化类表中的出现的顺序无关；二是当有些成员变量或父类对象没有在初始化列表中出现时，这些对象仍然被执行构造函数，这时执行的是“默认构造函数”。

在第一种情况构造类 c1 对象时：
1、因为有初始化列表，所以在初始化成员变量时，i=10;
2、运行 c1 的构造函数。

在第二中情况构造类 c1 对象时：
1、在初始化成员变量时,i=0;（默认构造函数的原因）。
2、运行 c1 的构造函数，i=10;

因此，在第二种情况，构造对象要比第一种情况多做了些事情，而且纯属浪费。因此对于类的成员变量初始化，尽量使用构造函数初始化列表。

3.12.11.4. 参数与返回值

减少对象创建/销毁的一个很简单且常见的方法就是在函数声明中将所有的值传递给常量引用传递。

比如：
int func(Object a);
改为
Int func(const Object &a);

首先我们要记住著名的 2/8 法则，20%的代码占用了程序 80%的时间。

速度瓶颈的类型：
    计算密集型
    IO 瓶颈
    相互等待

性能调整方法
定义指标

性能调优

设计层面
算法方面
Code 方面

定位热点

gprof
oprofile

code 调优方法：

shell script 调优

编译方法：

内存

Cache

缩短 ap 启动时间

3.12.12. 使用 mmap 来优化大文件操作

利用 mmap 实现的一个文件拷贝例子

```
/*
 * gcc -Wall -O3 -o copy_mmap copy_mmap.c
  */
#include   < stdio.h >
#include   < stdlib.h >
#include   < string .h >      /*   for memcpy   */
#include   < strings.h >
#include   < sys / mman.h >
#include   < sys / types.h >
#include   < sys / stat.h >
#include   < fcntl.h >
#include   < unistd.h >

#define   PERMS 0600

int   main (   int   argc,   char    *   argv[] )
{
       int                src, dst;
       void               * sm,   * dm;
       struct   stat   statbuf;
```

```
    if  ( argc  !=  3  )
    {
        fprintf( stderr,  "  Usage: %s <source> <target>\n " , argv[ 0 ] );
        exit( EXIT_FAILURE );
    }
    if  ( ( src  =  open( argv[ 1 ], O_RDONLY ) )  <  0  )
    {
        perror(  " open source "  );
        exit( EXIT_FAILURE );
    }
    /*  为了完成复制，必须包含读打开，否则 mmap()失败  */
    if  ( ( dst  =  open( argv[ 2 ], O_RDWR  |  O_CREAT  |  O_TRUNC, PERMS ) )
<  0  )
    {
        perror(  " open target "  );
        exit( EXIT_FAILURE );
    }
    if  ( fstat( src,  & statbuf )  <  0  )
    {
        perror(  " fstat source "  );
        exit( EXIT_FAILURE );
    }
    /*
    * 参看前面 man 手册中的说明，mmap()不能用于扩展文件长度。所以这里必须事
    * 先扩大目标文件长度，准备一个空架子等待复制。
     */
    if  ( lseek( dst, statbuf.st_size  -  1 , SEEK_SET )  <  0  )
    {
        perror(  " lseek target "  );
        exit( EXIT_FAILURE );
    }
    if  ( write( dst,  & statbuf,  1  )  !=  1  )
    {
        perror(  " write target "  );
        exit( EXIT_FAILURE );
    }

    /*  读的时候指定 MAP_PRIVATE 即可  */
    sm  =  mmap(  0 , ( size_t )statbuf.st_size, PROT_READ,
                MAP_PRIVATE  |  MAP_NORESERVE, src,  0  );
    if  ( MAP_FAILED  ==  sm )
    {
        perror(  " mmap source "  );
        exit( EXIT_FAILURE );
```

```
    }
     /*   这里必须指定 MAP_SHARED 才可能真正改变静态文件   */
    dm   =   mmap(   0 , ( size_t )statbuf.st_size, PROT_WRITE,
                    MAP_SHARED, dst,   0   );
     if   ( MAP_FAILED   ==   dm )
    {
        perror(   " mmap target "   );
        exit( EXIT_FAILURE );
    }
    memcpy( dm, sm, ( size_t )statbuf.st_size );
     /*
     *  可以不要这行代码
     *
     * msync( dm, ( size_t )statbuf.st_size, MS_SYNC );
      */
     return ( EXIT_SUCCESS );
}
```

mmap()好处是处理大文件时速度明显快于标准文件 I/O，无论读写，都少了一次用户空间与内核空间之间的复制过程。操作内存还便于设计、优化算法。

文件 I/O 操作/proc/self/mem 不存在页边界对齐的问题，但至少 Linux 的 mmap()的最后一个形参 offset 并未强制要求页边界对齐，如果提供的值未对齐，系统自动向上舍入到页边界上。malloc()分配得到的地址不见得对齐在页边界上。

/proc/self/mem 和/dev/kmem 不同。root 用户打开/dev/kmem 就可以在用户空间访问到内核空间的数据，包括偏移 0 处的数据，系统提供了这样的支持。显然代码段经过/proc/self/mem 可写映射后已经可写，无须 mprotect()介入。

3.12.13. ARM 体系结构相关

在当前嵌入式设备中，很多都采用了 ARM 体系结构的芯片，因为其影响力巨大，所以我们在这里单独列出一个章节，针对 ARM 体系结构做代码优化。

ARM 芯片具有用 RISC 体系的一般特点：

1、 具有大量的寄存器。
2、 绝大多数操作都在寄存器完成，通过 Load/Store 的体系结构在内存和寄存器之间传递数据。
3、 寻址方式简单。
4、 采用固定长度的指令格式。

同时 ARM 体系采用了一些特别的技术，在保证高性能的同时尽量减少芯片的体积，减少芯片的功耗。

1、 在同一条数据处理指令中包含算数逻辑处理单元和移位处理。

2、 使用地址自动增加（减少）来优化程序中循环处理。
3、 Load/Store 指令可以批量传输数据，从而提高数据传输效率。
4、 所有指令都可以根据前面指令执行结果，决定是否执行，以提高指令执行的效率。

3.12.13.1. ARM 指令简介

ARM 指令的一般编码格式

31 28	27 25	24 21	20	19 16	15 12	11 0
cond	001	opcode	S	Rn	Rd	shifter_operand

其中
opcode　指令操作符编码
cond　　执行执行的条件编码
S　　决定指令的操作是否影响 CPSR 的值。
Rd　　目标寄存器编码
Rn　　包含第一个操作数的寄存器编码
shift_operand　　表示第 2 个操作数。

CPSR 程序状态寄存器

31	30	29	28	27	26 7	6	5	4	3	2	1	0
N	Z	C	V	Q	DNM（RAZ）	I	F	T	M4	M3	M2	M1	M0

在这里我们主要关注条件标志位 N（Negative）、Z（Zero）、C（Carry）及 V（oVerflow）。

标志位	含义
N	本位设置成当前指令运算结果的 bit[31]的值。 当两个补码表示的有符号整数运算时，N=1 表示运算的结果为负数；N=0 表示结果为正数或 0
Z	Z=1 表示运算的结果为零；Z=0 表示运算的结果不为零。 对于 CMP 指令，Z=1 表示进行比较的两个数大小相等。
C	在加法指令中（包含比较指令 CMN），当结果产生了进位，则 C=1，表示无符号数运算发生上溢出；其他情况下 C=0 在剑法指令中（包含比较指令 CMP），当运算发生错位，则 C=0，表示无符号数运算发生下溢出；其他情况下 C=1 对于包含移位操作的非加/减法运算指令，C 中包含最后一次被溢出的位的数值。 对于其他非加/减法运算指令，C 位的值通常不受影响。
V	对于加/减法运算指令，当操作数和运算结果为二进制的补码表示的带符号数时，V=1 表示符号为溢出。

ARM 指令的条件码
大多数 ARM 指令都可以条件执行，也就是根据 CPSR 中的条件标志位来决定是否执行指令。

当条件满足时执行该指令，条件不满足时该指令被当作一条 NOP 指令，这时处理器进行判断中断请求等操作，然后专项下一条指令。

条件码 <cond>	条件码助记符	含义	CPSR 中条件标志位值
0000	EQ	相等	Z=1
0001	NE	不相等	Z=0
0010	CS/HS	无符号数大于/等于	C=1
0011	CC/LO	无符号数小于	C=0
0100	MI	负数	N=1
0101	PL	非负数	N=0
0110	VS	上溢出	V=1
0111	VC	没有上溢出	V=0
1000	HI	无符号数大于	C=1 且 Z=0
1001	LS	无符号数小于等于	C=0 且 Z=1
1010	GE	带符号数大于等于	N=1 且 V=1　或 N=0 且 V=0
1011	LT	带符号数小于	N=1 且 V=0　或 N=0 且 V=1
1100	GT	带符号数大于	Z=0 且 N=V
1101	LE	带符号数小于/等于	Z=1 或 N!=V
1110	AL	无条件执行	
1111	NV	该指令从不执行	

根据 ARM 指令的条件执行的特性，我们可以做如下优化。
1、 在做比较运算时，尽量与 0 比较，这样我们就可以利用条件执行的优势，减少一条指令。
例如

```
int i;
for(i=0;i<20;i++)
{
    ;
}

0x000010: MUL R2,R1,R2
0x000014: ADD R1,R1,#1
0x000018: CMP R1,R0
0x00001c: BLE 0x10
```

修改为

```
int i;
for(i=20;i>0;i--)
{
    ;
}
0x000010: MUL R0,R1,R0
0x000014: SUBS R1,R1,#1
```

0x000018: BNE 0x10

我们可以发现累加法比递减法多用了一条指令，当循环次数大的时候，这两段代码就会在性能上产生出明显的差异。其本质原因，在于当进行一个非零常数比较时，需要用专门的 CMP 指令来执行；而当一个变量与零进行比较时，ARM 指令可以直接利用条件执行的特性来进行判别。

3.12.13.2. ARM 寄存器

ARM 处理器有 7 种运行模式。

处理器模式	描述
用户模式	正常程序执行的模式
快速中断模式	用于高速数据传输和通道处理
外部中断模式	用于通常的终端处理
特权模式	供操作系统使用的一种保护模式
数据访问中止模式	用于虚拟存储及存储保护
未定义指令中止模式	用于支持通过软件仿真硬件的协处理器模式
系统模式	用于运行特权级的操作系统任务。

ARM 处理器共有 37 个寄存器。其中包括：
1、31 个通用寄存器，包括程序计数器（PC）在内。这些寄存器都是 32 位寄存器。
2、6 个状态寄存器，这些寄存器都是 32 位寄存器，但目前只使用了其中 12 位。

在用户态可用的有 16 个通用寄存器（R0～R15）和 1 个状态寄存器（CPSR）。
R13 在 ARM 中常用作栈指针，这只是一个习惯的用法，并没有任何强制性的，用户也可以使用其他的寄存器做为栈指针。
R14 又被称为连接寄存器（LR），在调用函数时，其用来保存函数的返回地址。
R15 又被称为程序计数器（PC），用来指向程序执行指令的地址。

3.12.14. 内存

对于程序员来讲，内存是一块线性区域，其速度要比 Flash 快一些，所以我们代码中的变量和一些常用的数据结构都保存在内存中。实际上这种看法极大的简化了内存实现上的复杂，这种简化对程序员编码有利，但也带来了内存性能上的浪费。

这里我先来讲述一下 CPU 和内存主频搭配的问题。
在过去的一段时间中，PC 所使用的 CPU 主频越来越快，达到了几 G 的主频，内存以 SDRAM 为主，而 CPU 与内存之间的总线频率却一直停留在 100Mhz、133Mhz，这就导致了内存成为了制约系统性能的瓶颈。

这个瓶颈到底有严重？CPU 在访问内存要等待多少个周期呢？

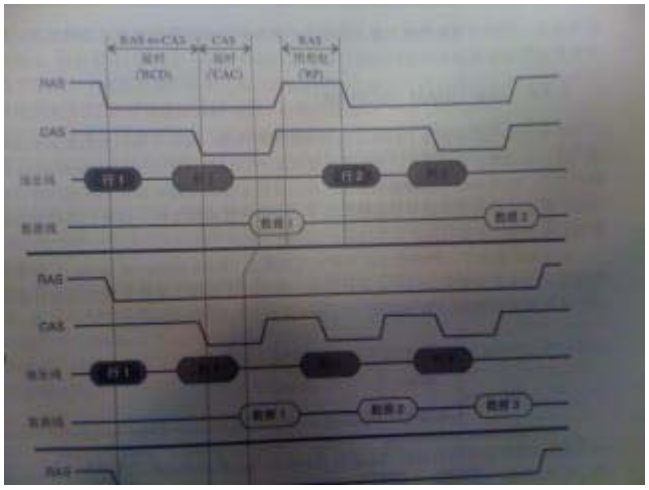
下面我们来算一下。

在计算之前，我先简单介绍一下 RAM 的基本原理。
这里我举一个例子，只是描述一下基本原理，实际上并不完全按照如此实现。
虽然在程序中，内存我们认为是一片线性地址空间，如果在制造芯片中，也采用线性排列的话，会有很多实际的苦难，所以芯片商们将其转变称为了多行的矩阵排列来实现。比如 1K 的物理内存，可以转变成 32×32 的矩阵来实现。
在程序访问内存时，内存控制器先将线性的内存地址转换成行地址和列地址，先发送行地址，找到内存中对应的行；然后发送列地址，定位到具体的物理地址，之后该地址所对应的数据在数据总线上输出。

这样我们就可以通过 3 个数值来描述内存的访问性能。
从读取最后一个单元到提供新行地址的延迟称为 RAS 预充电时间（RP）。
从提供行地址到列编号之间的延迟称为 RAS-CAS 延迟时间（RCD）。
从提供列编号到在数据总线上输出数据的延迟称为 CAS 延迟时间（CAC）。
以上这三个指标，内存芯片厂商都会提供。

比如说：PC100 SDRAM-222，其表示的含义为，内存工作频率为 100Mhz，其中第一个数字 2 表示为 RAS-CAS 时间；第二个数字 2 表示 CAS 延迟时间；第三个数字 2 表示 RAS 预充电时间。

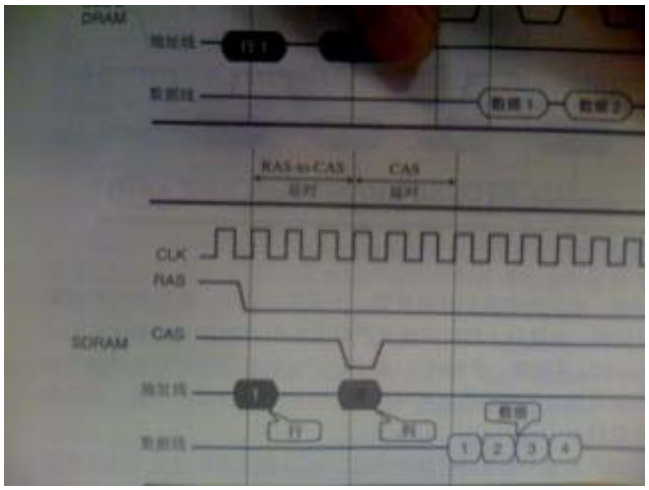
下面我们来看一下 DRAM 的时序图：



我们看到对于 DRAM 内存来讲，访问内存的时间=RAS 预充电时间+RAS-CAS 延迟时间+CAS 延迟时间。

为了提高 DRARM 的性能，出现了同步 DRAM，也就是我们目前经常使用的 SDRAM。
SDRAM 主要做了如下改动：
1、针对于同一行数据的存取，前一单元数据输出后，不必再经历 RAS 预充电、RAS-to-CASE 延迟的过程，直接发送下一个 CAS 信号，从而使读取同行单元的延迟从 RAS 预充电时间+RAS-CAS 延迟时间+CAS 延迟时间，缩短到只是一个 CAS 延迟时间。

时序图如下：



这种方法，对于位于不同行的单元无效。

2、SDRAM 实现了一种经过改进的突发交换模式。在内存控制器读取到第一个 64 位的数据后，其内部计数器可以直接把下一个相邻列地址所决定的基本存储单元的数据传送到输出缓存，而传送这些数据 64 位只需要一个时钟周期。
这种方式与 CPU 上面的 Cache 密切相关。

3、增加了存储器阵列的数目，SDRAM 的板块先从 1 增加到 2，然后到 4。这为在一个板块的内部电路充电期间存取另外一个板块上的内存单元提供了可能性。

4、数据线的位容量从 32 位提高到了 64 位。

倍速 SDRAM，也就是我们通常所说的 DDR 内存。
倍速 SDRAM 针对 SDRAM 的改进主要在于：数据可以在时钟脉冲的首尾两端同时传输，而在 SDRAM 中，数据仅仅在首端传输。这样其有效的频率提高了一倍，结果使 100MHz 的 DDR SDRAM 的性能与 2000MHz 的 SDRAM 的性能相当。

在当前嵌入式设备中，由于嵌入式设备中的 CPU 一般主频较低，还处于几百 Mhz 的水平，因此一般采用 SDRAM，目前 ARM 公司也在尝试使用 DDR 内存。本书中，我们还是以 SDRAM 为主。

现在我们开始着手计算 CPU 访问内存所造成的延迟。
CPU
主频 400Mhz，采用哈佛结构；
包含一级 cache 32K
I size : 16384
I assoc : 4
I line length : 32
I sets : 128
D size : 16384
D assoc : 4

D line length : 32
D sets : 128

总线
数据总线为 32 位，频率为 100Mhz。

内存
PC100 SDRAM-222

RAS 预充电时间（RP）	2 个时钟
RAS-CAS 延迟时间（RCD）	2 个时钟
CAS 延迟时间（CAC）	2 个时钟

在 CPU 要试图访问一块内存时，其会首先查看这块内存地址是否在 Cache 中，如果在 Cache 中，则不会访问内存。
如果不在 Cache 中，CPU 则访问内存，将包括该地址的 32 个字节写到 Cache 中，在从 cache 中找到其所需要的数据。
这里，我在多说一句：在 32 位的嵌入式系统中，系统一般有 32 根数据总线，却只有 27 根地址总线，这是为什么呢？
难道说在 32 位的嵌入式系统中，CPU 寻址空间仅为为 2**27 吗？
我们知道在 CPU 中内嵌了高速缓存 Cache，从而加快 CPU 访问内存的速度。在 Cache 中每个槽的大小为 32 个字节，正好是 2 的 5 次方。
在 CPU 中读取时时，它首先根据 27 根地址总线定位地址总线的高 27 位，然后将接下来的 32 个字节，一次 32 位，分为 8 次，传送到 Cache 缓存起来。CPU 在根据地址的低 5 位，从 Cache 定位到具体的字节。

CPU 从 cache 中读取的数据的速度很快，1～2 个 CPU 时钟周期，在这里我们先忽略不计。
那么 CPU 访问一块内存的时间，就等价于访问一块内存，并从内存中读取 32 个字节的时间。

下面我们列出整个内存系统各个阶段的延迟时间：

1、 CPU 到地址总线之间的延迟，1 个时钟周期。
2、 地址总线到 SDRAM 芯片的延迟，1 个时钟周期。
3、如果芯片组访问过某一 Bank 的某一行后，这一行处于“开”的状态，那么再次访问该行的数据，则只需要 CAS 的延迟，2 个时钟周期。如果芯片组某一 Bank 内，没有行处于打开状态，那么访问某一行的数据，则需要 RAS-CAS 的延迟时间+CAS 的延迟时间，4 个时钟周期。 如果芯片组某一 Bank 内，有行处于打开状态，而要访问另一行的数据时，需要先将原来打开行的数据写回，需要 RAS 预充电时间。所以其总共延迟为 6 个时钟。
4、1 个时钟周期用于传送数据。
5、DRAM 中的输出缓存中的数据传送到 CPU，需要 2 个时钟周期。
6、Cache 中一行为 32 个字节，而到这时我们读取了 4 个字节，接下来每个时钟周期传递 4 个字节，共需 7 个时钟周期。

这样 CPU 从内存中读取一个 32 位的数据，其总共需要 14～18 个总线的时钟周期。
这里我们还需要注意到 CPU 的主频为 400Mhz，而总线的主频为 100Mhz，这样 CPU 需要

等待的时间应该为 56～72 个 CPU 的时钟周期。

从这个角度上，我们可以看出程序对内存的操作，对程序自身的性能还是有很大的影响。下面我们将介绍一些方法，使程序尽可能的利用内存的特性，加快程序与内存之间的访问。

3.12.14.1. 消除数据的相关性

前面我们了解到，在 CPU 访问不在 Cache 中的内存时，会有很长时间的等待，实际上这种说法不完全正确。
在 CPU 试图访问一块内存，而要等待期间，其会去查看下面的指令是否依赖于当前访问的内存。如果依赖，那么将等待当前内存访问完成，然后执行后面的指令；如果不依赖，那么 CPU 可以继续执行下面的指令，内存返回数据或者后面的指令依赖于当前访问的内存。。

下面我们举个例子：

```
a=p[next];
b=next+1;
a++;
```

CPU 执行的顺序：
1、 CPU 试图访问内存 p[next]，由于访问内存速度比较慢，所以在几十个时钟内将不会得到应答。
2、 CPU 在发送完 p[next]的内存请求后，由于当前 next 的值已知，所以可以执行运行 b=next+1。
3、 当运行到 a++指令时，由于 a 的值未知，所以需要等待内存访问 p[next]返回，对 a 赋值后，才能执行。

所以最后的执行顺序是：

```
b=next+1;
a=p[next];
a++;
```

这也就是现代 CPU 技术中常说的乱序执行。注意，这里所说乱序执行，并不是把编译后的指令顺序打乱，而是 CPU 在执行过程中，根据指令运行的实际情况，而打乱原来的顺序。

如果 p[next]，保留在 Cache 中，就不会存在访问内存等待的问题，在 CPU 中的运行顺序将会是：

```
a=p[next];
b=next+1;
a++;
```

针对于 CPU 的这个特点，我们可以通过在 CPU 访问内存期间，增加一些非相关的指令，来达到充分利用 CPU 的目的，提高程序的运行速度。

非优化版本：

```
for(int i=0;i<1024;i++)
{
        a+=1;
        b+=i;
        c=p[i];
}
```

优化后的版本

```
for(int i=0;i<1024;i++)
{
        c=p[i];
        a+=1;
        b+=i;
}
```

这两个版本，从运行的结果上来看，完全是等价的。
优化版本，主要是里利用 CPU 在访问 p[i]内存时的等待时间，来执行 a+=1;b+=i;从而达到节省时间的目的。

3.12.14.2. 同时向内存控制器发送多个查询

上面我们看到，CPU 再访问一块内存

3.12.14.3. 请求按不少于 32 个字节的增量方式读取数据

3.12.14.4. 建议

1、 展开读取内存的循环
2、 消除数据相关性
3、 同时向存储控制器发送多个查询
4、 请求按不少于 32 个字节的增量方式读取数据
5、 使用所有经过请求的页面
6、 以一种排除了对相同 DRAM 页面进行选取的增量方式来处理数据。
7、 对齐数据源地址
8、 组合执行存取内存的代码
9、 成组进行读写操作
10、仅仅在必要时才访问内存。

3.12.14.5. 消除数据相关性

如果请求的 RAM 单元存在地址-数据相关性（也就是说，一个单元含有另外一个单元的地址），那么 CPU 不能并行处理他们，而在得到地址之前必须等待。
比如 while(next=p[next])
在处理器得到 next 变量之前，它不会知道下一个单元的地址而不能加载它。
while(a=p[next++])
处理器请求芯片组加载 p[next]单元，并立即将 next 加 1。用不着等待应答（因为下一个单元的地址是已知的），处理器向芯片组一个接一个地发送请求信息。

3.12.15. cache

增强的存储器访问

在 ARM11 处理器中，指令和数据可以更长时间的被保存在 Cache 中。一方面是由于物理地址 Cache 的实现，使上下文切换避免了反复重载 Cache，另一方面是由于 ARM11 的 Cache 还有很多其它新颖的技术特点。

如果数据的访问引起了 C a c h e Miss，Cache 将到存储器中读取需要的数据。但是 ARM11 处理器的流水线并不会停止下来。只要后面的指令没有用到 Cache Miss 将读回来的数据，ARM11 处理器就会继续执行下去。即使下一条指令还是存储器访问指令，只要数据存放在 Cache 中，ARM11 也会允许这条指令被执行。只有这条指令又引起一次 Cache Miss，处理器才会停止下来。在大多数应用中，经过编译器调配后，这种情况并不多见。

3.13. 硬加速

对于某些特定的应用，也许我们使用了浑身解数，还是无法满足性能上的要求，比如说面向实时性要求很强，计算量很大的视频播放的功能，这时我们就得求助于硬加速的功能。对于某一个功能，如果采用软件实现的化，几十个甚至上百个周期，而采用采用硬件实现的往往只需要几个时钟周期，所以硬加速一般可以带来上百倍的性能提高。

3.13.1. SIMD 加速

ARM 公司发布了新的媒体和信号处理 NEON?技术,将加速多种应用。ARM? NEON 技术适用于手机和消费娱乐电子,可灵活地实现多种视频编/解码、三维图像、语音处理、音频解码、图像处理和基带功能。NEON 技术将应用在将来的 ARM 处理器中,该技术也将获得 ARM 和

第三方工具提供商的广泛支持。ARM 公司（伦敦证交所：ARM;纳斯达克：ARMHY）是全球著名的 16/32 位嵌入式 RISC 微处理器技术方案供应商。
NEON 技术是 64/128 位单指令多数据流（SIMD）指令集,用于新一代媒体和信号处理应用加速。NEON 技术下执行 MP3 音频解码器,CPU 频率可低于 10 兆赫;运行 GSM AMR 语音数字编解码器,CPU 频率仅为 13 兆赫。新技术包含指令集、独立的寄存器及可独立的执行硬件。NEON 支持 8 位、16 位、32 位、64 位整数及单精度浮点 SIMD 操作,以进行音频/视频、图像和游戏处理。

GCC 3.4.6　开始支持 iwmmxt
GCC4.3.0　开始支持 neon。

3.14. 整体调优

对于在系统中生命周期很短的进程，由于其对内存影响很小，而我们对于其所耗费的时间有更高的要求，那么我们可以通过静态将动态库编译进来，从而减少加载动态库的数量，提高诉。

3.14.1. 进程执行速度与空闲内存的关系

不知道大家有没有这种感觉，设备用的时间长了，软件的执行速度越来越慢；有时程序第一次起来的时候比较慢，退出后再次启动，速度会快好多，这是为什么呢？理论上来讲软件的执行速度只应该与我们的代码和 CPU 有关系，而这两者都没变。

随着设备运行时间的延长，系统中的空闲内存会越来越少，而这与进程的运行速度有什么关系呢？
前面，我们讲到 Linux 的 cache 机制，在 Linux 中对于内存的态度是“尽量使用，闲着浪费”，因此在读取一个文件时，会争取将更多的内容从 flash 中加载到内存 cache 起来，这样在程序运行过程中，其触发页故障的几率就少，从而节省了进程的运行时间。
而当系统的空闲内存越来越少时，Linux 会自动的回收一些物理内存，这样在程序运行过程中，其触发的页故障的机率就越大，从而使进程的执行速度减慢。

这也就引入了通过控制 Cache 的方式，来加快进程执行速度的想法：
我们能否控制在 cache 的内存回收时，对于某些关键的进程所占用的 cache 尽量少的回收。人为的加大某一个进程的 cache 内存的数量，从而提高该进程的运行速度。

在 linux 中，有这样的一个开源的项目 preload，就是利用控制 Linux 中的 cache，来加快进程的启动速度。

preload 是一个很小的程序，通过 preload 这个小家伙，可以让你经常使用的程序启动速度更快。而且，小家伙是自适应的，它可以检测用户使用的程序，分析数据，预测用户可能会打开的程序，然后把需要的二进制代码或依赖的东西预先调入内存，这样就可以快速启动了。

比如我们经常打开的 firefox。

你可以在http://sourceforge.net/projects/preload，获得preload的代码，集成到你的系统中。

这里有一片文章http://www.techthrob.com/tech/preload.php，讲述如何配置和使用preload，有兴趣的朋友可以参考。

3.14.2. 调整进程的优先级

当多个进程同时运行，竞争 CPU 资源的时候，为了保证某个进程运行速度，我们可以采用调整进程的调度策略和优先级的方法。

在 Linux 内核中，支持两种进程：实时进程和普通进程。

实时进程
实时进程的优先级是静态设定的，而且始终大于普通进程的优先级。因此只有当运行队列中没有实时进程的情况下，普通进程才能够获得调度。

实时进程采用两种调度策略：SCHED_FIFO 和 SCHED_RR。FIFO 采用先进先出的策略，对于所有相同优先级的进程，最先进入 runqueue 的进程总能优先获得调度；Round Robin 采用更加公平的轮转策略，使得相同优先级的实时进程能够轮流获得调度。

对于实时进程来讲，其使用绝对优先级的概念，绝对优先级取值范围是 0～99，数字越大，优先级别越高。

普通进程
在 Linux 2.6 中，普通进程的绝对优先级取值是 0。
在普通进程之间，其又具备静态优先级和动态优先级之分。静态优先级，我们可以通过程序来进行修改。同时系统在运行过程中，会在静态优先级基础上，不断的动态计算出每个进程的动态优先级，拥有最高动态优先级的进程被调度器选中。拥有最高优先级的进程被调度器选中。一般来讲，静态优先级越高，进程所能分配的时间片越长，用户可以通过 nice 系统调用修改进程的静态优先级。

动态优先级的公式：
动态优先级=max(100,min(静态优先级-bonus +5,139))
其中 bonus 取决于进程的平均睡眠时间。由此可以看出，在 linux2.6 中，一个普通进程的优先级和平均睡眠时间的关系为：平均睡眠时间越长，其 bonus 越大，从而得到更高的优先级。

对于优先级非常的进程，我们可以将其设置为实时进程，来保证其运行速度。
我们可以调用 sched_setscheduler，来修改进程的调度策略。

#include <sched.h>

int sched_setscheduler (pid t pid, int policy, const struct sched param *param)；
这个函数可以同时设置一个进程的静态优先级和调度策略。pid用来指名所要设置的进程号，如果pid为0，则表示为当前进程；policy设置进程的调度策略；param用来设置进程的绝对优先级。

参数policy
取值范围：
SCHE_OTHER　　普通进程
SCHE_FIFO　　实时进程，采用先进先出策略。
SCHE_RR　　实时进程，采用轮转策略

参数 param
struct sched_param
{
　　int sched_priority;
};
sched_priority为进程的绝对优先级，其取值范围0～99。

返回值：
　　如果设置成功，返回值为0；如果失败，返回指为-1。

相关函数：
int sched_getscheduler (pid t pid)；
　　获得调度策略。

int sched_setparam (pid t pid, const struct sched param *param)；
　　设置进程的绝对优先级

int sched_getparam (pid t pid, const struct sched param *param)
　　获得进程的绝对优先级

上面，我们讲述了如何将进程设置为实时进程，从而获得最高的运行优先级。下面我们将主要针对普通进程的调度。

普通进程
对于普通进程来讲，其绝对优先级为0，我们可以通过修改的进程的nice值来来影响进程的调度。nice的取值-20～19，负值或0表示对高优点级，对其它进程不谦让，也就是拥有优先占用系统资源的权利。

#include <sys/resource.h>
int setpriority (int class, int id, int niceval)；
class的取值为：
PRIO_PROCESS　　id为进程的pid。
PRIO_PGRP　　id为进程所在组的组ID。

PRIO_USER　　id为用户ID
niceval　　为进程的nice值，其取值范围为-20～19。

返回值
　　成功返回0；失败返回-1。

另外一个常用的函数为int nice (int increment)；

```
int    nice (int increment)
{
      int result, old = getpriority (PRIO_PROCESS, 0);
      result = setpriority (PRIO_PROCESS, 0, old + increment);
      if (result != -1)
            return old + increment;
      else
      return -1;
}
```

increment为所设置进程nice值的增量。

在使用 shell 脚本启动进程时，我们也可以使用 nice 来设置进程的优先级。

nice -n 5　gaim &　　注：运行 gaim 程序，并为它指定谦让度增量为 5；
所以 nice 的最常用的应用就是：
nice　-n　谦让度的增量值　　程序

线程的优先级：
在 Linux 中，线程本身就是一个执行单位，有其自己的优先级。它与其进程的优先级没有依赖关系。

如果我们想将线程设置为实时的线程，我们可以调用 LinuxThreads 库的 pthread_setschedparam 函数，其是 sched_setscheduler 的线程版本，用于动态修改运行着的线程的调度优先级和策略：
int pthread_setschedparam(pthread_t target_thread, int policy, const struct sched_param *param);。
target_thread　　线程的 pid；
policy　　线程的调度策略；
param　　线程的优先级；

对于普通线程，我们仍然可以使用 setpriority 和 nice 函数。

3.14.3. shutdown、reboot 的区别

系统关机和重启，有两种实现方式，一种是通过 script 来实现；一种是通过编程来实现。

脚本的方式很简单:

#reboot
就可以了

编程实现

```
#include <unistd.h>
#include <linux/reboot.h>
#include <stdio.h>
#include <stdlib.h>
int main() {
	int ret;
	/*
	 * Flush filesystem buffers and the time according to buffer
	 * size.    If need, add usleep here to wait flush finish.
	 */
	system("sync");

	/* Manipulate hardware to power off phone directly. */
	printf("Shutting down the phone!\n");
	ret = reboot(LINUX_REBOOT_CMD_POWER_OFF);
	if(ret != 0)
		printf("the return value : %d \n", ret);
	return ret;
}
```

这两种实现的方式：

脚本实现的 reboot
系统的初始化，也就是内核最后一个初始化动作就是启动 init 程序，在我的系统中，init 程序是由 busybox 来实现的。。脚本的解释和执行都是由 busybox 来执行的；在 busybox 执行脚本，运行到 reboot 这一行时，一般它不会直接触发 Linux 内核，要求其重启动；而是由 init 程序接管，给用户提供一个配置关机脚本的机会。

而对于程序中调用的 reboot 来讲，其 reboot 命令将直接与 Linux 内核打交道，立即生效，这样就不会去运行系统所配置的 shutdown 脚本，会有所遗漏。

3.14.4. 设备的启动时间

缩短 phone 的启动时间，我们首先要弄明白设备的启动时间都包括哪些。
起机时间主要包括：

1、 Bootloader 启动，加载内核
2、 Linux 内核初始化
3、 运行起机脚本。

缩短加载 Linux 内核时间：

1、 使用 XIP 技术启动 Linux 内核
2、 使用 DMA 技术，将 Linux 内核和文件系统拷贝到 RAM 中。

Linux 内核自身加速
1、 Disable console
在 Linux 内核的启动参数中加上”quiet”选项，关掉 console，来节省内核中使用 Printk 打印字符时间。
其可节省时间 0.1~02.秒。
在 Linux 启动之后，我们可以通过/proc/cmdline 来查看内核启动参数。

2、 缩短硬件检测时间
因为我们面向的是特定的嵌入式设备，其硬件是固定的，那么我们完全可以通过修改 Linux 内核来缩短甚至去除不必要的硬件检测。

3、 关注内核加载模块时间
在 2.4 的内核中，一些外部的大的内核模块，在其加载时可能会用到 3 秒的时间。

用户态及应用程序加速
在 Bootloader 和 Linux 启动完之后，就是用户态的进程加载，其按规定的顺序执行一系列的脚本，来完成用户系统初始化的动作。
1、 这个执行的过程，有很多是由 bash 脚本来完成的，我们需要按前面的说法去优化 bash 脚本来优化执行时间。
2、 Bash 脚本一般是按顺序执行的，在不同的时段，CPU 的利用率是不同的，这就要求我们调整脚本的执行顺序，以达到充分利用 CPU 等各种资源，从而提高系统的速度。
3、 对用户感觉有影响的关键的进程需要先执行，无关的进程可以放在后面执行。这需要我们了解进程之间的相互依赖关系，从而调整进程的顺序。
4、 将 bash 脚本的某些功能转由 C 编写的应用程序来完成，进而节省时间。
5、
6、 在 busybox 的 init 中，其是顺序串行的执行脚本，这样往往无法充分利用系统资源。fastboot 是一个开源的项目，其通过并行执行 bash 脚本来提高启动速度。
你可以访问其网站http://www.fastboot.org/，详细了解。

3.14.5. damon 的数量

在系统中长期停留的进程，我们称其为 damon 进程，它一般为其他进程提供服务或传递消息。而这类进程对系统整个的性能尤为重要。
1、 即使 damon 进程十分简单，它也会调用动态链接库，系统再为每个动态链接库分配代码段和数据段，从而占用大量的物理内存。
2、 代码越复杂的程序，其造成内存泄漏的概率越大，而对于长存生存的 damon 进程来讲，一丁点的内存泄漏，都有可能导致系统的内存耗尽。
3、 在系统中存活的 damon 进程越多，cpu 的负担越重，也会导致系统的整体性能下降。

减少 damon 进程的数量：

1、 很多 damon 进程之所以成为 damon 进程，主要是因为进程的响应速度跟不上。比如说要求按一个按钮，弹出一个画面的时间要在 0.5 秒；而如果这时，启动一个进程的话，从启动进程，load 动态库，执行用户代码的时间不满足要求，我们没办法只好把它做成一个 damon 进程，驻留在系统中。这时我们要想办法尽量去缩短进程的启动时间。
2、 将 damon 进程划分为两个部分：一部分代码简单，稳定可靠，长期驻留在内存中；另一部分代码复杂，实现用户逻辑，用完了就释放。这样的好处在于因为具有长期驻留部分的代码，在启动用户逻辑时，省去了创建进程、加载动态库的时间，同时又能够把由 damon 部分代码造成的内存泄漏减到最小。
   我们可以这样设计：
   damon 部分：启动进程，加载动态库，等待用户事件。
   用户逻辑部分：创建出子线程，实现用户逻辑，完成后删除子线程。
3、 damon 进程合并，在系统中我们经常需要驻留很多个进程，来为各种各样的需求提供服务。我们可以把这些具有类似功能的 damon 进行合并，从而降低 damon 的数目。

设计守护进程的要点

1、 出于提高启动速度的考虑，我们需要调整守护进程的启动顺序，这就涉及到了进程之间的相互依赖的问题。一方面，我们要全面掌握进程之间的相互依赖关系；另一方面，我们也需要将进程之间之间的相互依赖调整到最低。
2、 在运行过程中有些进程可能会由于错误突然死掉，我们要提供重新重新启动的机制；由于进程可能因为死锁等原因停在那里，我们要提供 watchdog 机制。
3、 由于守护进程的重启，可能导致依赖其的进程找不到相应的服务，我们需要为其设计一套寻找服务的机制。

3.14.6. 文件系统

1. 基于 FLASH 的文件系统

Flash(闪存)作为嵌入式系统的主要存储媒介，有其自身的特性。Flash 的写入操作只能把对应位置的 1 修改为 0，而不能把 0 修改为 1(擦除 Flash 就是把对应存储块的内容恢复为 1)，因此，一般情况下，向 Flash 写入内容时，需要先擦除对应的存储区间，这种擦除是以块(block)为单位进行的。

闪存主要有 NOR 和 NAND 两种技术(简单比较见附录)。Flash 存储器的擦写次数是有限的，NAND 闪存还有特殊的硬件接口和读写时序。因此，必须针对 Flash 的硬件特性设计符合应用要求的文件系统；传统的文件系统如 ext2 等，用作 Flash 的文件系统会有诸多弊端。

在嵌入式 Linux 下，MTD(Memory Technology Device,存储技术设备)为底层硬件(闪存)和上层(文件系统)之间提供一个统一的抽象接口，即 Flash 的文件系统都是基于 MTD 驱动层的(参见上面的 Linux 下的文件系统结构图)。使用 MTD 驱动程序的主要优点在于，它是专门针对各种非易失性存储器(以闪存为主)而设计的，因而它对 Flash 有更好的支持、管理和基于扇区的擦除、读/写操作接口。

顺便一提，一块 Flash 芯片可以被划分为多个分区，各分区可以采用不同的文件系统；两块 Flash 芯片也可以合并为一个分区使用，采用一个文件系统。即文件系统是针对于存储器分区而言的，而非存储芯片。

(1) jffs2

JFFS 文件系统最早是由瑞典 Axis Communications 公司基于 Linux2.0 的内核为嵌入式系统开发的文件系统。JFFS2 是 RedHat 公司基于 JFFS 开发的闪存文件系统，最初是针对 RedHat 公司的嵌入式产品 eCos 开发的嵌入式文件系统，所以 JFFS2 也可以用在 Linux, uCLinux 中。

Jffs2: 日志闪存文件系统版本 2 (Journalling Flash FileSystem v2)

主要用于 NOR 型闪存，基于 MTD 驱动层，特点是：可读写的、支持数据压缩的、基于哈希表的日志型文件系统，并提供了崩溃/掉电安全保护，提供“写平衡”支持等。缺点主要是当文件系统已满或接近满时，因为垃圾收集的关系而使 jffs2 的运行速度大大放慢。

目前 jffs3 正在开发中。关于 jffs 系列文件系统的使用详细文档，可参考 MTD 补丁包中 mtd-jffs-HOWTO.txt。

jffsx 不适合用于 NAND 闪存主要是因为 NAND 闪存的容量一般较大，这样导致 jffs 为维护日志节点所占用的内存空间迅速增大，另外，jffsx 文件系统在挂载时需要扫描整个 FLASH 的内容，以找出所有的日志节点，建立文件结构，对于大容量的 NAND 闪存会耗费大量时间。

(2) yaffs：Yet Another Flash File System

yaffs/yaffs2 是专为嵌入式系统使用 NAND 型闪存而设计的一种日志型文件系统。与 jffs2 相比，它减少了一些功能(例如不支持数据压缩)，所以速度更快，挂载时间很短，对内存的占用较小。另外，它还是跨平台的文件系统，除了 Linux 和 eCos，还支持 WinCE, pSOS 和 ThreadX 等。

yaffs/yaffs2 自带 NAND 芯片的驱动，并且为嵌入式系统提供了直接访问文件系统的 API，用户可以不使用 Linux 中的 MTD 与 VFS，直接对文件系统操作。当然，yaffs 也可与 MTD 驱动程序配合使用。

yaffs 与 yaffs2 的主要区别在于，前者仅支持小页(512 Bytes) NAND 闪存，后者则可支持大页(2KB) NAND 闪存。同时，yaffs2 在内存空间占用、垃圾回收速度、读/写速度等方面均有大幅提升。

(3) Cramfs：Compressed ROM File System

Cramfs 是 Linux 的创始人 Linus Torvalds 参与开发的一种只读的压缩文件系统。它也基于 MTD 驱动程序。

在 cramfs 文件系统中，每一页(4KB)被单独压缩，可以随机页访问，其压缩比高达 2:1, 为嵌入式系统节省大量的 Flash 存储空间，使系统可通过更低容量的 FLASH 存储相同的文件，从而降低系统成本。

Cramfs 文件系统以压缩方式存储，在运行时解压缩，所以不支持应用程序以 XIP 方式运行，所有的应用程序要求被拷到 RAM 里去运行，但这并不代表比 Ramfs 需求的 RAM 空间要大一点，因为 Cramfs 是采用分页压缩的方式存放档案，在读取档案时，不会一下子就耗用过多的内存空间，只针对目前实际读取的部分分配内存，尚没有读取的部分不分配内存空间，当我们读取的档案不在内存时，Cramfs 文件系统自动计算压缩后的资料所存的位置，

再即时解压缩到 RAM 中。

另外，它的速度快，效率高，其只读的特点有利于保护文件系统免受破坏，提高了系统的可靠性。

由于以上特性，Cramfs 在嵌入式系统中应用广泛。

但是它的只读属性同时又是它的一大缺陷，使得用户无法对其内容对进扩充。

Cramfs 映像通常是放在 Flash 中，但是也能放在别的文件系统里，使用 loopback 设备可以把它安装别的文件系统里。

(4) Romfs

传统型的 Romfs 文件系统是一种简单的、紧凑的、只读的文件系统，不支持动态擦写保存，按顺序存放数据，因而支持应用程序以 XIP(eXecute In Place，片内运行)方式运行，在系统运行时，节省 RAM 空间。uClinux 系统通常采用 Romfs 文件系统。

其他文件系统：fat/fat32 也可用于实际嵌入式系统的扩展存储器(例如 PDA, Smartphone, 数码相机等的 SD 卡)，这主要是为了更好的与最流行的 Windows 桌面操作系统相兼容。ext2 也可以作为嵌入式 Linux 的文件系统，不过将它用于 FLASH 闪存会有诸多弊端。

2. 基于 RAM 的文件系统

(1) Ramdisk

Ramdisk 是将一部分固定大小的内存当作分区来使用。它并非一个实际的文件系统，而是一种将实际的文件系统装入内存的机制，并且可以作为根文件系统。将一些经常被访问而又不会更改的文件(如只读的根文件系统)通过 Ramdisk 放在内存中，可以明显地提高系统的性能。

在 Linux 的启动阶段，initrd 提供了一套机制，可以将内核映像和根文件系统一起载入内存。

(2)ramfs/tmpfs

Ramfs 是 Linus Torvalds 开发的一种基于内存的文件系统，工作于虚拟文件系统(VFS)层，不能格式化，可以创建多个，在创建时可以指定其最大能使用的内存大小。(实际上，VFS 本质上可看成一种内存文件系统，它统一了文件在内核中的表示方式，并对磁盘文件系统进行缓冲。)

Ramfs/tmpfs 文件系统把所有的文件都放在 RAM 中，所以读/写操作发生在 RAM 中，可以用 ramfs/tmpfs 来存储一些临时性或经常要修改的数据，例如/tmp 和/var 目录，这样既避免了对 Flash 存储器的读写损耗，也提高了数据读写速度。

Ramfs/tmpfs 相对于传统的 Ramdisk 的不同之处主要在于：不能格式化，文件系统大小可随所含文件内容大小变化。

Tmpfs 的一个缺点是当系统重新引导时会丢失所有数据。

3.14.7. 系统待机时间

共享库的结构
耗电：
频率调整
电压的调整

/tmp 目录

GPU 对多媒体的影响

不使用 atime 属性

gcc 的编译选项
让我们先看看 Makefile 规则中的编译命令通常是怎么写的。

大多数软件包遵守如下约定俗成的规范：

#1,首先从源代码生成目标文件(预处理,编译,汇编)，"-c"选项表示不执行链接步骤。$(CC) $(CPPFLAGS) $(CFLAGS) example.c -c -o example.o#2,然后将目标文件连接为最终的结果(连接)，"-o"选项用于指定输出文件的名字。$(CC) $(LDFLAGS) example.o -o example#有一些软件包一次完成四个步骤：$(CC) $(CPPFLAGS) $(CFLAGS) $(LDFLAGS) example.c -o example
当然也有少数软件包不遵守这些约定俗成的规范，比如：

#1,有些在命令行中漏掉应有的 Makefile 变量(注意：有些遗漏是故意的)$(CC) $(CFLAGS) example.c -c -o example.o$(CC) $(CPPFLAGS) example.c -c -o example.o$(CC) example.o -o example$(CC) example.c -o example#2,有些在命令行中增加了不必要的 Makefile 变量$(CC) $(CFLAGS) $(LDFLAGS) example.o -o example$(CC) $(CPPFLAGS) $(CFLAGS) $(LDFLAGS) example.c -c -o example.o
当然还有极个别软件包完全是"胡来"：乱用变量(增加不必要的又漏掉了应有的)者有之，不用$(CC)者有之，不一而足.....

尽管将源代码编译为二进制文件的四个步骤由不同的程序(cpp,gcc/g++,as,ld)完成，但是事实上 cpp, as, ld 都是由 gcc/g++ 进行间接调用的。换句话说，控制了 gcc/g++ 就等于控制了

所有四个步骤。从 Makefile 规则中的编译命令可以看出，编译工具的行为全靠 CC/CXX CPPFLAGS CFLAGS/CXXFLAGS LDFLAGS 这几个变量在控制。当然理论上控制编译工具行为的还应当有 AS ASFLAGS ARFLAGS 等变量，但是实践中基本上没有软件包使用它们。

那么我们如何控制这些变量呢？一种简易的做法是首先设置与这些 Makefile 变量同名的环境变量并将它们 export 为全局，然后运行 configure 脚本，大多数 configure 脚本会使用这同名的环境变量代替 Makefile 中的值。但是少数 configure 脚本并不这样做(比如 GCC 和 Binutils 的脚本就不传递 LDFLAGS)，你必须手动编辑生成的 Makefile 文件，在其中寻找这些变量并修改它们的值，许多源码包在每个子文件夹中都有 Makefile 文件，真是一件很累人的事！

CC 与 CXX
这是 C 与 C++ 编译器命令。默认值一般是 "gcc" 与 "g++"。这个变量本来与优化没有关系，但是有些人因为担心软件包不遵守那些约定俗成的规范，害怕自己苦心设置的 CFLAGS/CXXFLAGS/LDFLAGS 之类的变量被忽略了，而索性将原本应当放置在其它变量中的选项一股老儿塞到 CC 或 CXX 中，比如：CC="gcc -march=k8 -O2 -s"。这是一种怪异的用法，本文不提倡这种做法，而是提倡按照变量本来的含义使用变量。

CPPFLAGS
这是用于预处理阶段的选项。不过能够用于此变量的选项，看不出有哪个与优化相关。如果你实在想设一个，那就使用下面这两个吧：

-DNDEBUG
"NDEBUG"是一个标准的 ANSI 宏，表示不进行调试编译。
-D_FILE_OFFSET_BITS=64
大多数包使用这个来提供大文件(>2G)支持。
CFLAGS 与 CXXFLAGS
CFLAGS 表示用于 C 编译器的选项，CXXFLAGS 表示用于 C++ 编译器的选项。这两个变量实际上涵盖了编译和汇编两个步骤。大多数程序和库在编译时默认的优化级别是"2"(使用 "-O2" 选项) 并且带有调试符号来编译，也就是 CFLAGS="-O2 -g", CXXFLAGS=$CFLAGS 。事实上，"-O2"已经启用绝大多数安全的优化选项了。另一方面，由于大部分选项可以同时用于这两个变量，所以仅在最后讲述只能用于其中一个变量的选项。[提醒]下面所列选项皆为非默认选项，你只要按需添加即可。

先说说"-O3"在"-O2"基础上增加的几项：

-finline-functions
允许编译器选择某些简单的函数在其被调用处展开，比较安全的选项，特别是在 CPU 二级缓存较大时建议使用。
-funswitch-loops
将循环体中不改变值的变量移动到循环体之外。
-fgcse-after-reload
为了清除多余的溢出，在重载之后执行一个额外的载入消除步骤。

另外：

-fomit-frame-pointer
对于不需要栈指针的函数就不在寄存器中保存指针，因此可以忽略存储和检索地址的代码，同时对许多函数提供一个额外的寄存器。所有"-O"级别都打开它，但仅在调试器可以不依靠栈指针运行时才有效。在 AMD64 平台上此选项默认打开，但是在 x86 平台上则默认关闭。建议显式的设置它。
-falign-functions=N
-falign-jumps=N
-falign-loops=N
-falign-labels=N
这四个对齐选项在"-O2"中打开，其中的根据不同的平台 N 使用不同的默认值。如果你想指定不同于默认值的 N，也可以单独指定。比如，对于 L2-cache>=1M 的 cpu 而言，指定 -falign-functions=64 可能会获得更好的性能。建议在指定了 -march 的时候不明确指定这里的值。
调试选项：

-fprofile-arcs
在使用这一选项编译程序并运行它以创建包含每个代码块的执行次数的文件后，程序可以再次使用 -fbranch-probabilities 编译，文件中的信息可以用来优化那些经常选取的分支。如果没有这些信息，gcc 将猜测哪个分支将被经常运行以进行优化。这类优化信息将会存放在一个以源文件为名字的并以".da"为后缀的文件中。
全局选项：

-pipe
在编译过程的不同阶段之间使用管道而非临时文件进行通信，可以加快编译速度。建议使用。
目录选项：

--sysroot=dir
将 dir 作为逻辑根目录。比如编译器通常会在 /usr/include 和 /usr/lib 中搜索头文件和库，使用这个选项后将在 dir/usr/include 和 dir/usr/lib 目录中搜索。如果使用这个选项的同时又使用了 -isysroot 选项，则此选项仅作用于库文件的搜索路径，而 -isysroot 选项将作用于头文件的搜索路径。这个选项与优化无关，但是在 CLFS 中有着神奇的作用。
代码生成选项：

-fno-bounds-check
关闭所有对数组访问的边界检查。该选项将提高数组索引的性能，但当超出数组边界时，可能会造成不可接受的行为。
-freg-struct-return
如果 struct 和 union 足够小就通过寄存器返回，这将提高较小结构的效率。如果不够小，无法容纳在一个寄存器中，将使用内存返回。建议仅在完全使用 GCC 编译的系统上才使用。
-fpic
生成可用于共享库的位置独立代码。所有的内部寻址均通过全局偏移表完成。要确定一个地址，需要将代码自身的内存位置作为表中一项插入。该选项产生可以在共享库中存放并从中

加载的目标模块。
-fstack-check
为防止程序栈溢出而进行必要的检测，仅在多线程环境中运行时才可能需要它。
-fvisibility=hidden
设置默认的 ELF 镜像中符号的可见性为隐藏。使用这个特性可以非常充分的提高连接和加载共享库的性能，生成更加优化的代码，提供近乎完美的 API 输出和防止符号碰撞。我们强烈建议你在编译任何共享库的时候使用该选项。参见 -fvisibility-inlines-hidden 选项。
硬件体系结构相关选项[仅仅针对 x86 与 x86_64]：

-march=cpu-type
为特定的 cpu-type 编译二进制代码(不能在更低级别的 cpu 上运行)。Intel 可以用：pentium2, pentium3(=pentium3m), pentium4(=pentium4m), pentium-m, divscott, nocona, core2(GCC-4.3 新增)。AMD 可以用：k6-2(=k6-3), athlon(=athlon-tbird), athlon-xp(=athlon-mp), k8(=opteron=athlon64=athlon-fx)
-mfpmath=sse
P3 和 athlon-xp 级别及以上的 cpu 支持"sse"标量浮点指令。仅建议在 P4 和 K8 以上级别的处理器上使用该选项。
-malign-double
将 double, long double, long long 对齐于双字节边界上；有助于生成更高速的代码，但是程序的尺寸会变大，并且不能与未使用该选项编译的程序一起工作。
-m128bit-long-double
指定 long double 为 128 位，pentium 以上的 cpu 更喜欢这种标准，并且符合 x86-64 的 ABI 标准，但是却不附合 i386 的 ABI 标准。
-mregparm=N
指定用于传递整数参数的寄存器数目(默认不使用寄存器)。0<=N<=3 ；注意：当 N>0 时你必须使用同一参数重新构建所有的模块，包括所有的库。
-msseregparm
使用 SSE 寄存器传递 float 和 double 参数和返回值。注意：当你使用了这个选项以后，你必须使用同一参数重新构建所有的模块，包括所有的库。
-mmmx
-msse
-msse2
-msse3
-m3dnow
-mssse3(没写错!GCC-4.3 新增)
-msse4.1(GCC-4.3 新增)
-msse4.2(GCC-4.3 新增)
-msse4(含 4.1 和 4.2,GCC-4.3 新增)
是否使用相应的扩展指令集以及内置函数，按照自己的 cpu 选择吧！
-maccumulate-outgoing-args
指定在函数引导段中计算输出参数所需最大空间，这在大部分现代 cpu 中是较快的方法；缺点是会明显增加二进制文件尺寸。
-mthreads
支持 Mingw32 的线程安全异常处理。对于依赖于线程安全异常处理的程序，必须启用这个

选项。使用这个选项时会定义"-D_MT"，它将包含使用选项"-lmingwthrd"连接的一个特殊的线程辅助库，用于为每个线程清理异常处理数据。

-minline-all-stringops

默认时 GCC 只将确定目的地会被对齐在至少 4 字节边界的字符串操作内联进程序代码。该选项启用更多的内联并且增加二进制文件的体积，但是可以提升依赖于高速 memcpy, strlen, memset 操作的程序的性能。

-minline-stringops-dynamically

GCC-4.3 新增。对未知尺寸字符串的小块操作使用内联代码，而对大块操作仍然调用库函数，这是比"-minline-all-stringops"更聪明的策略。决定策略的算法可以通过"-mstringop-strategy"控制。

-momit-leaf-frame-pointer

不为叶子函数在寄存器中保存栈指针，这样可以节省寄存器，但是将会使调试变的困难。注意：不要与 -fomit-frame-pointer 同时使用，因为会造成代码效率低下。

-m64

生成专门运行于 64 位环境的代码，不能运行于 32 位环境，仅用于 x86_64[含 EMT64]环境。

-mcmodel=small

[默认值]程序和它的符号必须位于 2GB 以下的地址空间。指针仍然是 64 位。程序可以静态连接也可以动态连接。仅用于 x86_64[含 EMT64]环境。

-mcmodel=kernel

内核运行于 2GB 地址空间之外。在编译 linux 内核时必须使用该选项！仅用于 x86_64[含 EMT64]环境。

-mcmodel=medium

程序必须位于 2GB 以下的地址空间，但是它的符号可以位于任何地址空间。程序可以静态连接也可以动态连接。注意：共享库不能使用这个选项编译！仅用于 x86_64[含 EMT64]环境。

其它优化选项：

-fforce-addr

必须将地址复制到寄存器中才能对他们进行运算。由于所需地址通常在前面已经加载到寄存器中了，所以这个选项可以改进代码。

-finline-limit=n

对伪指令数超过 n 的函数，编译程序将不进行内联展开，默认为 600。增大此值将增加编译时间和编译内存用量并且生成的二进制文件体积也会变大，此值不宜太大。

-fmerge-all-constants

试图将跨编译单元的所有常量值和数组合并在一个副本中。但是标准 C/C++要求每个变量都必须有不同的存储位置，所以该选项可能会导致某些不兼容的行为。

-fgcse-sm

在全局公共子表达式消除之后运行存储移动，以试图将存储移出循环。gcc-3.4 中曾属于"-O2"级别的选项。

-fgcse-las

在全局公共子表达式消除之后消除多余的在存储到同一存储区域之后的加载操作。gcc-3.4 中曾属于"-O2"级别的选项。

-floop-optimize

已废除(GCC-4.1 曾包含在"-O1"中)。

-floop-optimize2
使用改进版本的循环优化器代替原来"-floop-optimize"。该优化器将使用不同的选项(-funroll-loops, -fpeel-loops, -funswitch-loops, -ftree-loop-im)分别控制循环优化的不同方面。目前这个新版本的优化器尚在开发中，并且生成的代码质量并不比以前的版本高。已废除，仅存在于 GCC-4.1 之前的版本中。
-funsafe-loop-optimizations
假定循环不会溢出，并且循环的退出条件不是无穷。这将可以在一个比较广的范围内进行循环优化，即使优化器自己也不能断定这样做是否正确。
-fsched-spec-load
允许一些装载指令执行一些投机性的动作。
-ftree-loop-linear
在 trees 上进行线型循环转换。它能够改进缓冲性能并且允许进行更进一步的循环优化。
-fivopts
在 trees 上执行归纳变量优化。
-ftree-vectorize
在 trees 上执行循环向量化。
-ftracer
执行尾部复制以扩大超级块的尺寸，它简化了函数控制流，从而允许其它的优化措施做的更好。据说挺有效。
-funroll-loops
仅对循环次数能够在编译时或运行时确定的循环进行展开，生成的代码尺寸将变大，执行速度可能变快也可能变慢。
-fdivfetch-loop-arrays
生成数组预读取指令，对于使用巨大数组的程序可以加快代码执行速度，适合数据库相关的大型软件等。具体效果如何取决于代码。
-fweb
建立经常使用的缓存器网络，提供更佳的缓存器使用率。gcc-3.4 中曾属于"-O3"级别的选项。
-ffast-math
违反 IEEE/ANSI 标准以提高浮点数计算速度，是个危险的选项，仅在编译不需要严格遵守 IEEE 规范且浮点计算密集的程序考虑采用。
-fsingle-divcision-constant
将浮点常量作为单精度常量对待，而不是隐式地将其转换为双精度。
-fbranch-probabilities
在使用 -fprofile-arcs 选项编译程序并执行它来创建包含每个代码块执行次数的文件之后，程序可以利用这一选项再次编译，文件中所产生的信息将被用来优化那些经常发生的分支代码。如果没有这些信息，gcc 将猜测那一分支可能经常发生并进行优化。这类优化信息将会存放在一个以源文件为名字的并以".da"为后缀的文件中。
-frename-registers
试图驱除代码中的假依赖关系，这个选项对具有大量寄存器的机器很有效。gcc-3.4 中曾属于"-O3"级别的选项。
-fbranch-target-load-optimize
-fbranch-target-load-optimize2
在执行序启动以及结尾之前执行分支目标缓存器加载最佳化。
-fstack-protector

在关键函数的堆栈中设置保护值。在返回地址和返回值之前，都将验证这个保护值。如果出现了缓冲区溢出，保护值不再匹配，程序就会退出。程序每次运行，保护值都是随机的，因此不会被远程猜出。

-fstack-protector-all

同上，但是在所有函数的堆栈中设置保护值。

--param max-gcse-memory=xxM

执行 GCSE 优化使用的最大内存量(xxM)，太小将使该优化无法进行，默认为 50M。

--param max-gcse-passes=n

执行 GCSE 优化的最大迭代次数，默认为 1。

传递给汇编器的选项：

-Wa,options

options 是一个或多个由逗号分隔的可以传递给汇编器的选项列表。其中的每一个均可作为命令行选项传递给汇编器。

-Wa,--strip-local-absolute

从输出符号表中移除局部绝对符号。

-Wa,-R

合并数据段和正文段，因为不必在数据段和代码段之间转移，所以它可能会产生更短的地址移动。

-Wa,--64

设置字长为 64bit，仅用于 x86_64，并且仅对 ELF 格式的目标文件有效。此外，还需要使用"--enable-64-bit-bfd"选项编译的 BFD 支持。

-Wa,-march=CPU

按照特定的 CPU 进行优化：pentiumiii, pentium4, divscott, nocona, core, core2; athlon, sledgehammer, opteron, k8 。

仅可用于 CFLAGS 的选项：

-fhosted

按宿主环境编译，其中需要有完整的标准库，入口必须是 main()函数且具有 int 型的返回值。内核以外几乎所有的程序都是如此。该选项隐含设置了 -fbuiltin，且与 -fno-freestanding 等价。

-ffreestanding

按独立环境编译，该环境可以没有标准库，且对 main()函数没有要求。最典型的例子就是操作系统内核。该选项隐含设置了 -fno-builtin，且与 -fno-hosted 等价。

仅可用于 CXXFLAGS 的选项：

-fno-enforce-eh-specs

C++标准要求强制检查异常违例，但是该选项可以关闭违例检查，从而减小生成代码的体积。该选项类似于定义了"NDEBUG"宏。

-fno-rtti

如果没有使用'dynamic_cast'和'typeid'，可以使用这个选项禁止为包含虚方法的类生成运行时表示代码，从而节约空间。此选项对于异常处理无效(仍然按需生成 rtti 代码)。

-ftemplate-depth-n

将最大模版实例化深度设为'n'，符合标准的程序不能超过 17，默认值为 500。

-fno-optional-diags
禁止输出诊断消息，C++标准并不需要这些消息。
-fno-threadsafe-statics
GCC 自动在访问 C++局部静态变量的代码上加锁，以保证线程安全。如果你不需要线程安全，可以使用这个选项。
-fvisibility-inlines-hidden
默认隐藏所有内联函数，从而减小导出符号表的大小，既能缩减文件的大小，还能提高运行性能，我们强烈建议你在编译任何共享库的时候使用该选项。参见 -fvisibility=hidden 选项。
LDFLAGS
LDFLAGS 是传递给连接器的选项。这是一个常被忽视的变量，事实上它对优化的影响也是很明显的。

[提示]以下选项是在完整的阅读了 ld-2.18 文档之后挑选出来的选项。http://blog.chinaunix.net/u1/41220/showart_354602.html 有 2.14 版本的中文手册。

-s
删除可执行程序中的所有符号表和所有重定位信息。其结果与运行命令 strip 所达到的效果相同，这个选项是比较安全的。
-Wl,options
options 是由一个或多个逗号分隔的传递给链接器的选项列表。其中的每一个选项均会作为命令行选项提供给链接器。
-Wl,-On
当 n>0 时将会优化输出，但是会明显增加连接操作的时间，这个选项是比较安全的。
-Wl,--exclude-libs=ALL
不自动导出库中的符号，也就是默认将库中的符号隐藏。
-Wl,-m<emulation>
仿真<emulation>连接器，当前 ld 所有可用的仿真可以通过"ld -V"命令获取。默认值取决于 ld 的编译时配置。
-Wl,--sort-common
把全局公共符号按照大小排序后放到适当的输出节，以防止符号间因为排布限制而出现间隙。
-Wl,-x
删除所有的本地符号。
-Wl,-X
删除所有的临时本地符号。对于大多数目标平台，就是所有的名字以'L'开头的本地符号。
-Wl,-zcomberloc
组合多个重定位节并重新排布它们，以便让动态符号可以被缓存。
-Wl,--enable-new-dtags
在 ELF 中创建新式的"dynamic tags"，但在老式的 ELF 系统上无法识别。
-Wl,--as-needed
移除不必要的符号引用，仅在实际需要的时候才连接，可以生成更高效的代码。
-Wl,--no-define-common
限制对普通符号的地址分配。该选项允许那些从共享库中引用的普通符号只在主程序中被分配地址。这会消除在共享库中的无用的副本的空间，同时也防止了在有多个指定了搜索路径

的动态模块在进行运行时符号解析时引起的混乱。
-Wl,--hash-style=gnu
使用 gnu 风格的符号散列表格式。它的动态链接性能比传统的 sysv 风格(默认)有较大提升，但是它生成的可执行程序和库与旧的 Glibc 以及动态链接器不兼容。

最后说两个与优化无关的系统环境变量，因为会影响 GCC 编译程序的方式，下面两个是咱中国人比较关心的：

LANG
指定编译程序使用的字符集，可用于创建宽字符文件、串文字、注释；默认为英文。[目前只支持日文"C-JIS,C-SJIS,C-EUCJP"，不支持中文]
LC_ALL
指定多字节字符的字符分类，主要用于确定字符串的字符边界以及编译程序使用何种语言发出诊断消息；默认设置与 LANG 相同。中文相关的几项："zh_CN.GB2312 , zh_CN.GB18030 , zh_CN.GBK , zh_CN.UTF-8 , zh_TW.BIG5"。

ARM Options
These `-m' options are defined for Advanced RISC Machines (ARM) architectures:

-mabi=name
Generate code for the specified ABI. Permissible values are: `apcs-gnu', `atpcs', `aapcs' and `iwmmxt'.

-mapcs-frame
Generate a stack frame that is compliant with the ARM Procedure Call Standard for all functions, even if this is not strictly necessary for correct execution of the code. Specifying -fomit-frame-pointer with this option will cause the stack frames not to be generated for leaf functions. The default is -mno-apcs-frame.

-mapcs
This is a synonym for -mapcs-frame.

-mthumb-interwork
Generate code which supports calling between the ARM and Thumb instruction sets. Without this option the two instruction sets cannot be reliably used inside one program. The default is -mno-thumb-interwork, since slightly larger code is generated when -mthumb-interwork is specified.

-mno-sched-prolog
Prevent the reordering of instructions in the function prolog, or the merging of those instruction with the instructions in the function's body. This means that all functions will start with a

recognizable set of instructions (or in fact one of a choice from a small set of different function prologues), and this information can be used to locate the start if functions inside an executable piece of code. The default is -msched-prolog.

-mhard-float
Generate output containing floating point instructions. This is the default.

-msoft-float
Generate output containing library calls for floating point. Warning: the requisite libraries are not available for all ARM targets. Normally the facilities of the machine's usual C compiler are used, but this cannot be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation.
-msoft-float changes the calling convention in the output file; therefore, it is only useful if you compile all of a program with this option. In particular, you need to compile libgcc.a, the library that comes with GCC, with -msoft-float in order for this to work.

-mfloat-abi=name
Specifies which ABI to use for floating point values. Permissible values are: `soft', `softfp' and `hard'.
`soft' and `hard' are equivalent to -msoft-float and -mhard-float respectively. `softfp' allows the generation of floating point instructions, but still uses the soft-float calling conventions.

-mlittle-endian
Generate code for a processor running in little-endian mode. This is the default for all standard configurations.

-mbig-endian
Generate code for a processor running in big-endian mode; the default is to compile code for a little-endian processor.

-mwords-little-endian
This option only applies when generating code for big-endian processors. Generate code for a little-endian word order but a big-endian byte order. That is, a byte order of the form `32107654'. Note: this option should only be used if you require compatibility with code for big-endian ARM processors generated by versions of the compiler prior to 2.8.

-mcpu=name
This specifies the name of the target ARM processor. GCC uses this name to determine what kind of instructions it can emit when generating assembly code. Permissible names are: `arm2', `arm250', `arm3', `arm6', `arm60', `arm600', `arm610', `arm620', `arm7', `arm7m', `arm7d', `arm7dm', `arm7di', `arm7dmi', `arm70', `arm700', `arm700i', `arm710', `arm710c', `arm7100', `arm7500', `arm7500fe', `arm7tdmi', `arm7tdmi-s', `arm8', `strongarm', `strongarm110',

`strongarm1100', `arm8', `arm810', `arm9', `arm9e', `arm920', `arm920t', `arm922t', `arm946e-s', `arm966e-s', `arm968e-s', `arm926ej-s', `arm940t', `arm9tdmi', `arm10tdmi', `arm1020t', `arm1026ej-s', `arm10e', `arm1020e', `arm1022e', `arm1136j-s', `arm1136jf-s', `mpcore', `mpcorenovfp', `arm1176jz-s', `arm1176jzf-s', `xscale', `iwmmxt', `ep9312'.

-mtune=name

This option is very similar to the -mcpu= option, except that instead of specifying the actual target processor type, and hence restricting which instructions can be used, it specifies that GCC should tune the performance of the code as if the target were of the type specified in this option, but still choosing the instructions that it will generate based on the cpu specified by a -mcpu= option. For some ARM implementations better performance can be obtained by using this option.

-march=name

This specifies the name of the target ARM architecture. GCC uses this name to determine what kind of instructions it can emit when generating assembly code. This option can be used in conjunction with or instead of the -mcpu= option. Permissible names are: `armv2', `armv2a', `armv3', `armv3m', `armv4', `armv4t', `armv5', `armv5t', `armv5te', `armv6', `armv6j', `iwmmxt', `ep9312'.

-mfpu=name

-mfpe=number

-mfp=number

This specifies what floating point hardware (or hardware emulation) is available on the target. Permissible names are: `fpa', `fpe2', `fpe3', `maverick', `vfp'. -mfp and -mfpe are synonyms for -mfpu=`fpe'number, for compatibility with older versions of GCC.

If -msoft-float is specified this specifies the format of floating point values.

-mstructure-size-boundary=n

The size of all structures and unions will be rounded up to a multiple of the number of bits set by this option. Permissible values are 8, 32 and 64. The default value varies for different toolchains. For the COFF targeted toolchain the default value is 8. A value of 64 is only allowed if the underlying ABI supports it.

Specifying the larger number can produce faster, more efficient code, but can also increase the size of the program. Different values are potentially incompatible. Code compiled with one value cannot necessarily expect to work with code or libraries compiled with another value, if they exchange information using structures or unions.

-mabort-on-noreturn

Generate a call to the function abort at the end of a noreturn function. It will be executed if the function tries to return.

-mlong-calls

-mno-long-calls

Tells the compiler to perform function calls by first loading the address of the function into a register and then performing a subroutine call on this register. This switch is needed if the target function will lie outside of the 64 megabyte addressing range of the offset based version of subroutine call instruction.
Even if this switch is enabled, not all function calls will be turned into long calls. The heuristic is that static functions, functions which have the `short-call' attribute, functions that are inside the scope of a `#pragma no_long_calls' directive and functions whose definitions have already been compiled within the current compilation unit, will not be turned into long calls. The exception to this rule is that weak function definitions, functions with the `long-call' attribute or the `section' attribute, and functions that are within the scope of a `#pragma long_calls' directive, will always be turned into long calls.

This feature is not enabled by default. Specifying -mno-long-calls will restore the default behavior, as will placing the function calls within the scope of a `#pragma long_calls_off' directive. Note these switches have no effect on how the compiler generates code to handle function calls via function pointers.

-mnop-fun-dllimport
Disable support for the dllimport attribute.

-msingle-pic-base
Treat the register used for PIC addressing as read-only, rather than loading it in the prologue for each function. The run-time system is responsible for initializing this register with an appropriate value before execution begins.

-mpic-register=reg
Specify the register to be used for PIC addressing. The default is R10 unless stack-checking is enabled, when R9 is used.

-mcirrus-fix-invalid-insns
Insert NOPs into the instruction stream to in order to work around problems with invalid Maverick instruction combinations. This option is only valid if the -mcpu=ep9312 option has been used to enable generation of instructions for the Cirrus Maverick floating point co-processor. This option is not enabled by default, since the problem is only present in older Maverick implementations. The default can be re-enabled by use of the -mno-cirrus-fix-invalid-insns switch.

-mpoke-function-name
Write the name of each function into the text section, directly preceding the function prologue. The generated code is similar to this:

```
t0
    .ascii "arm_poke_function_name", 0
    .align
t1
```

```
        .word 0xff000000 + (t1 - t0)
arm_poke_function_name
        mov     ip, sp
        stmfd   sp!, {fp, ip, lr, pc}
        sub     fp, ip, #4
```

When performing a stack backtrace, code can inspect the value of pc stored at fp + 0. If the trace function then looks at location pc - 12 and the top 8 bits are set, then we know that there is a function name embedded immediately preceding this location and has length ((pc[-3]) & 0xff000000).

-mthumb
Generate code for the 16-bit Thumb instruction set. The default is to use the 32-bit ARM instruction set.

-mtpcs-frame
Generate a stack frame that is compliant with the Thumb Procedure Call Standard for all non-leaf functions. (A leaf function is one that does not call any other functions.) The default is -mno-tpcs-frame.

-mtpcs-leaf-frame
Generate a stack frame that is compliant with the Thumb Procedure Call Standard for all leaf functions. (A leaf function is one that does not call any other functions.) The default is -mno-apcs-leaf-frame.

-mcallee-super-interworking
Gives all externally visible functions in the file being compiled an ARM instruction set header which switches to Thumb mode before executing the rest of the function. This allows these functions to be called from non-interworking code.

-mcaller-super-interworking
Allows calls via function pointers (including virtual functions) to execute correctly regardless of whether the target code has been compiled for interworking or not. There is a small overhead in the cost of executing a function pointer if this option is enabled.

qmake 选项
去掉-O2
/vobs/ezx_qt/qt/code/mkspecs/qws/linux-all/qmake.conf

QMAKE_CFLAGS_RELEASE = -O2

去掉单个动态库的 O2
在 pro 文件中，加入 QMAKE_CFLAGS_RELEASE =

4. 附录

4.1. gcc 与 g++的不同

gcc 和 g++都是 GNU 的一个编译器。

很多人都认为 gcc 只能编译 C 程序，而 g++只能编译 C++代码。实际上而这都可以。
1.后缀为.c 的，gcc 把它当作是 C 程序，而 g++当作是 c++程序；后缀为.cpp 的，两者都会认为是 c++程序，注意，虽然 c++是 c 的超集，但是两者对语法的要求是有区别的。C++的语法规则更加严谨一些。
2.编译阶段，g++会调用 gcc，对于 c++代码，两者是等价的，但是因为 gcc 命令不能自动和 C＋＋程序使用的库联接，所以通常用 g++来完成链接，为了统一起见，干脆编译/链接统统用 g++了，这就给人一种错觉，好像 cpp 程序只能用 g++似的。

在这里有一篇文章做了详细阐述。
http://blog.chinaunix.net/u/30686/showart.php?id=519752

这里我要着重说明的是，在使用 g++编译的时候，无论这个程序是否使用了类，都将会将依赖于 C++的动态库（libstdc++.so）。

下面我们写了一个测试程序，然后分别使用 gcc 和 g++分别编译，然后对比其内存使用情况。

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

int main()
{
	pause();
	return 0;
}
```

编译：
```
gcc –o hello hello.c
```

运行
```
# ./hello

# cat maps
```

```
00008000-00009000 r-xp 00000000 1f:12 291          /mnt/msc_int0/hello
00010000-00011000 rw-p 00000000 1f:12 291          /mnt/msc_int0/hello
40000000-40001000 rw-p 40000000 00:00 0
41000000-41017000 r-xp 00000000 1f:0d 698815       /lib/ld-2.3.3.so
4101e000-41020000 rw-p 00016000 1f:0d 698815       /lib/ld-2.3.3.so
41028000-41120000 r-xp 00000000 1f:0d 699047       /lib/libc-2.3.3.so
41120000-41128000 ---p 000f8000 1f:0d 699047       /lib/libc-2.3.3.so
41128000-41129000 r--p 000f8000 1f:0d 699047       /lib/libc-2.3.3.so
41129000-4112c000 rw-p 000f9000 1f:0d 699047       /lib/libc-2.3.3.so
4112c000-4112e000 rw-p 4112c000 00:00 0
befeb000-bf000000 rwxp befeb000 00:00 0
```

我们可以看到，hello 进程只依赖于 libc 一个动态库。

我们使用 g++对齐进行编译：

编译
g++ -o hello hello.c

运行：
./hello

```
# cat maps
00008000-00009000 r-xp 00000000 1f:12 293          /mnt/msc_int0/hello
00010000-00011000 rw-p 00000000 1f:12 293          /mnt/msc_int0/hello
40000000-40001000 rw-p 40000000 00:00 0
41000000-41017000 r-xp 00000000 1f:0d 698815       /lib/ld-2.3.3.so
4101e000-41020000 rw-p 00016000 1f:0d 698815       /lib/ld-2.3.3.so
41028000-41120000 r-xp 00000000 1f:0d 699047       /lib/libc-2.3.3.so
41120000-41128000 ---p 000f8000 1f:0d 699047       /lib/libc-2.3.3.so
41128000-41129000 r--p 000f8000 1f:0d 699047       /lib/libc-2.3.3.so
41129000-4112c000 rw-p 000f9000 1f:0d 699047       /lib/libc-2.3.3.so
4112c000-4112e000 rw-p 4112c000 00:00 0
411b0000-41219000 r-xp 00000000 1f:0d 1871995      /lib/libm-2.3.3.so
41219000-41220000 ---p 00069000 1f:0d 1871995      /lib/libm-2.3.3.so
41220000-41222000 rw-p 00068000 1f:0d 1871995      /lib/libm-2.3.3.so
41228000-412e4000 r-xp 00000000 1f:0d 8228665      /usr/lib/libstdc++.so.6.0.3
412e4000-412e8000 ---p 000bc000 1f:0d 8228665      /usr/lib/libstdc++.so.6.0.3
412e8000-412f0000 rw-p 000b8000 1f:0d 8228665      /usr/lib/libstdc++.so.6.0.3
412f0000-412f5000 rw-p 412f0000 00:00 0
41308000-41310000 r-xp 00000000 1f:0d 1871213      /lib/libgcc_s.so.1
41310000-41311000 rw-p 00008000 1f:0d 1871213      /lib/libgcc_s.so.1
befeb000-bf000000 rwxp befeb000 00:00 0
```

我们会发现 hello 所依赖的库增加了 3 个 libstdc++.so　libm.so　libgcc_s.so,其中 libm.so 和 libgcc_s.so 是由 libstdc++.so 引入的。不要小看这三个库，进程每加载一个库，就要为其分配相应的数据段，做一些重定向工作等，每个库的数据段最少要占据 4K 的物理内存，也就是说使用 g++编译出的 hello，要比使用 gcc 编译出来的，最少内存多了 12K。

4.2. 代码段和数据段优化

在 linux 内核中，物理内存是以页为单位了，也就是说物理内存最少以 4K 为单位。
程序的代码段和数据段，在编译出来之后，就是固定的；它不会随着程序的运行，扩大和缩小。因此我们可以静态的来分析程序代码段和数据段使用内存。

问题一：在动态库中，bss 和 data 节是分别属于两个不同的段。
这有可能造成内存的浪费，比如说 bss 段实际使用了 1K 的物理内存，data 段实际使用了 1K 的物理内存，但由于 linux 物理内存最小单位是 4K，所以系统将不得不为该程序分配两个物理页面，来容纳两个段，也就是 8K 内存。
如果把 bss 和 data 合并的话，那么系统只需要为其分配一个物理页面，这样就节省出了 4K 的空间。
同理，代码段也可以 data 段进行合并，但由于效果有限，一般不采用。

在 gcc 中，可以通过修改连接器脚本，来自由定义 section。
ld --verbose 查看默认链接脚本
ld 把一定量的目标文件跟档案文件连接起来,并重定位它们的数据,连接符号引用.一般在编译一个程序时,最后一步就是运行 ld。
实例 1：

```
SECTIONS
{
        . = 0x10000;
        .text : { *(.text) }
        . = 0x8000000;
        .data : { *(.data) }
        .bss : { *(.bss) }
}
```

注释：“.”是定位计数器，设置当前节的地址。

实例 2：

```
floating_point = 0;
      SECTIONS
      {
. = ALIGN(4);
         .text :
           {
             *(.text)
              _etext = .;
```

```
PROVIDE(etext = .);
    }

. = ALIGN(4);
        _bdata = (. + 3) & ~ 3;
        .data : { *(.data) }
    }
```

注释：定义一个符合_etext，地址为.text 结束的地方，注意源程序中不能在此定义该符合，否则链接器会提示重定义，而是应该象下面这样使用：

```
extern char _etext;
```

但是可以在源程序中使用 etext 符合，连接器不导出它到目标文件。

实例 3：

```
  SECTIONS {
        outputa 0x10000 :
          {
          all.o
          foo.o (.input1)
          }
        outputb :
          {
          foo.o (.input2)
          foo1.o (.input1)
          }
        outputc :
          {
          *(.input1)
          *(.input2)
          }
  }
```

这个例子是一个完整的连接脚本。它告诉连接器去读取文件 all.o 中的所有节，并把它们放到输出节 outputa 的开始位置处， 该输出节是从位置 0x10000 处开始的。从文件 foo.o 中来的所有节.input1 在同一个输出节中紧密排列。 从文件 foo.o 中来的所有节.input2 全部放入到输出节 outputb 中，后面跟上从 foo1.o 中来的节.input1。来自所有文件的所有余下的.input1 和.input2 节被写入到输出节 outputc 中。

示例 4：连接器填充法则：

```
    SECTIONS { .text : { *(.text) } LONG(1) .data : { *(.data) } }                    错误
     SECTIONS { .text : { *(.text) ; LONG(1) } .data : { *(.data) } }          正确
```

示例 5：VMA 和 LMA 不同的情况

```
    SECTIONS
      {
      .text 0x1000 : { *(.text) _etext = . ; }
```

```
        .mdata 0x2000 :
          AT ( ADDR (.text) + SIZEOF (.text) )
          { _data = . ; *(.data); _edata = . ;    }
        .bss 0x3000 :
          { _bstart = . ;    *(.bss) *(COMMON) ; _bend = . ;}
    }
```

程序：

```
    extern char _etext, _data, _edata, _bstart, _bend;
    char *src = &_etext;
    char *dst = &_data;

    /* ROM has data at end of text; copy it. */
    while (dst &lt; &_edata) {
      *dst++ = *src++;
    }

    /* Zero bss */
    for (dst = &_bstart; dst&lt; &_bend; dst++)
      *dst = 0;
```

示例 6：linux-2.6.14/arch/i386/kernel $ vi vmlinux.lds.S

linux 内核的链接脚本，自行分析吧，有点复杂哦。

Parallel add /subtract	Signed add high 16 + 16, low 16 + 16, set GE flags	`SADD16{cond} <Rd>, <Rn>, <Rm>`
	Saturated add high 16 + 16, low 16 + 16	`QADD16{cond} <Rd>, <Rn>, <Rm>`
	Signed high 16 + 16, low 16 + 16, halved	`SHADD16{cond} <Rd>, <Rn>, <Rm>`
	Unsigned high 16 + 16, low 16 + 16, set GE flags	`UADD16{cond} <Rd>, <Rn>, <Rm>`

	Saturated unsigned high 16 + 16, low 16 + 16	`UQADD16{cond} <Rd>, <Rn>, <Rm>`
	Unsigned high 16 + 16, low 16 + 16, halved	`UHADD16{cond} <Rd>, <Rn>, <Rm>`
	Signed high 16 + low 16, low 16 - high 16, set GE flags	`SADDSUBX{cond} <Rd>, <Rn>, <Rm>`
	Saturated high 16 + low 16, low 16 - high 16	`QADDSUBX{cond} <Rd>, <Rn>, <Rm>`
	Signed high 16 + low 16, low 16 - high 16, halved	`SHADDSUBX{cond} <Rd>, <Rn>, <Rm>`
	Unsigned high 16 + low 16, low 16 - high 16, set GE flags	`UADDSUBX{cond} <Rd>, <Rn>, <Rm>`
	Saturated unsigned high 16 + low 16, low 16 - high 16	`UQADDSUBX{cond} <Rd>, <Rn>, <Rm>`
	Unsigned high 16 + low 16, low 16 - high 16, halved	`UHADDSUBX{cond} <Rd>, <Rn>, <Rm>`

	Signed high 16 - low 16, low 16 + high 16, set GE flags	SSUBADDX{cond} <Rd>, <Rn>, <Rm>
	Saturated high 16 - low 16, low 16 + high 16	QSUBADDX{cond} <Rd>, <Rn>, <Rm>
	Signed high 16 - low 16, low 16 + high 16, halved	SHSUBADDX{cond} <Rd>, <Rn>, <Rm>
	Unsigned high 16 - low 16, low 16 + high 16, set GE flags	USUBADDX{cond} <Rd>, <Rn>, <Rm>
	Saturated unsigned high 16 - low 16, low 16 + high 16	UQSUBADDX{cond} <Rd>, <Rn>, <Rm>
	Unsigned high 16 - low 16, low 16 + high 16, halved	UHSUBADDX{cond} <Rd>, <Rn>, <Rm>
	Signed high 16-16, low 16-16, set GE flags	SSUB16{cond} <Rd>, <Rn>, <Rm>
	Saturated high 16 - 16, low 16 - 16	QSUB16{cond} <Rd>, <Rn>, <Rm>

	Signed high 16 - 16, low 16 - 16, halved	SHSUB16{cond} <Rd>, <Rn>, <Rm>
	Unsigned high 16 - 16, low 16 - 16, set GE flags	USUB16{cond} <Rd>, <Rn>, <Rm>
	Saturated unsigned high 16 - 16, low 16 - 16	UQSUB16{cond} <Rd>, <Rn>, <Rm>
	Unsigned high 16 - 16, low 16 - 16, halved	UHSUB16{cond} <Rd>, <Rn>, <Rm>
	Four signed 8 + 8, set GE flags	SADD8{cond} <Rd>, <Rn>, <Rm>
	Four saturated 8 + 8	QADD8{cond} <Rd>, <Rn>, <Rm>
	Four signed 8 + 8, halved	SHADD8{cond} <Rd>, <Rn>, <Rm>
	Four unsigned 8 + 8, set GE flags	UADD8{cond} <Rd>, <Rn>, <Rm>
	Four saturated unsigned 8 + 8	UQADD8{cond} <Rd>, <Rn>, <Rm>
	Four unsigned 8 + 8, halved	UHADD8{cond} <Rd>, <Rn>, <Rm>
	Four signed 8 - 8, set GE flags	SSUB8{cond} <Rd>, <Rn>, <Rm>
	Four saturated 8 - 8	QSUB8{cond} <Rd>, <Rn>, <Rm>

	Four signed 8 - 8, halved	`SHSUB8{cond} <Rd>, <Rn>, <Rm>`
	Four unsigned 8 - 8	`USUB8{cond} <Rd>, <Rn>, <Rm>`
	Four saturated unsigned 8 - 8	`UQSUB8{cond} <Rd>, <Rn>, <Rm>`
	Four unsigned 8 - 8, halved	`UHSUB8{cond} <Rd>, <Rn>, <Rm>`
	Sum of absolute differences	`USAD8{cond} <Rd>, <Rm>, <Rs>`
	Sum of absolute differences and accumulate	`USADA8{cond} <Rd>, <Rm>, <Rs>, <Rn>`
Sign/zero extend and add	Two low 8/16, sign extend to 16 + 16	`SXTAB16{cond} <Rd>, <Rn>, <Rm>{, <rotation>}`
	Low 8/32, sign extend to 32, + 32	`SXTAB{cond} <Rd>, <Rn>, <Rm>{, <rotation>}`
	Low 16/32, sign extend to 32, + 32	`SXTAH{cond} <Rd>, <Rn>, <Rm>{, <rotation>}`
	Two low 8/16, zero extend to 16, + 16	`UXTAB16{cond} <Rd>, <Rn>, <Rm>{, <rotation>}`
	Low 8/32, zero extend to 32, + 32	`UXTAB{cond} <Rd>, <Rn>, <Rm>{, <rotation>}`
	Low 16/32, zero extend to 32, + 32	`UXTAH{cond} <Rd>, <Rn>, <Rm>{, <rotation>}`
	Two low 8, sign extend to 16, packed 32	`SXTB16{cond} <Rd>, <Rm>{, <rotation>}`
	Low 8, sign extend to	`SXTB{cond} <Rd>, <Rm>{,`

	32	`<rotation>}`
	Low 16, sign extend to 32	`SXTH{cond} <Rd>, <Rm>{, <rotation>}`
	Two low 8, zero extend to 16, packed 32	`UXTB16{cond} <Rd>, <Rm>,{, <rotation>}`
	Low 8, zero extend to 32	`UXTB{cond} <Rd>, <Rm>{, <rotation>}`
	Low 16, zero extend to 32	`UXTH{cond} <Rd>, <Rm>{, <rotation>}`
Signed multiply and multiply, accumulate	Signed (high 16 x 16) + (low 16 x 16) + 32, and set Q flag.	`SMLAD{cond} <Rd>, <Rm>, <Rs>, <Rn>`
	As SMLAD, but high x low, low x high, and set Q flag	`SMLADX{cond} <Rd>, <Rm>, <Rs>, <Rn>`
	Signed (high 16 x 16) - (low 16 x 16) + 32	`SMLSD{cond} <Rd>, <Rm>, <Rs>, <Rn>`
	As SMLSD, but high x low, low x high	`SMLSDX{cond} <Rd>, <Rm>, <Rs>, <Rn>`
	Signed (high 16 x 16) + (low 16 x 16) + 64	`SMLALD{cond} <RdLo>, <RdHi>, <Rm>, <Rs>`

	As SMLALD, but high x low, low x high	SMLALDX{cond} <RdLo>, <RdHi>, <Rm>, <Rs>
	Signed (high 16 x 16) - (low 16 x 16) + 64	SMLSLD{cond} <RdLo>, <RdHi>, <Rm>, <Rs>
	As SMLSLD, but high x low, low x high	SMLSLDX{cond} <RdLo>, <RdHi>, <Rm>, <Rs>
	32 + truncated high 16 (32 x 32)	SMMLA{cond} <Rd>, <Rm>, <Rs>, <Rn>
	32 + rounded high 16 (32 x 32)	SMMLAR{cond} <Rd>, <Rm>, <Rs>, <Rn>
	32 - truncated high 16 (32 x 32)	SMMLS{cond} <Rd>, <Rm>, <Rs>, <Rn>
	32 -rounded high 16 (32 x 32)	SMMLSR{cond} <Rd>, <Rm>, <Rs>, <Rn>
	Signed (high 16 x 16) + (low 16 x 16), and set Q flag	SMUAD{cond} <Rd>, <Rm>, <Rs>
	As SMUAD, but high x low, low x high, and set Q flag	SMUADX{cond} <Rd>, <Rm>, <Rs>
	Signed (high 16 x 16) - (low 16 x 16)	SMUSD{cond} <Rd>, <Rm>, <Rs>

	As SMUSD, but high x low, low x high	SMUSDX{cond} <Rd>, <Rm>, <Rs>
	Truncated high 16 (32 x 32)	SMMUL{cond} <Rd>, <Rm>, <Rs>
	Rounded high 16 (32 x 32)	SMMULR{cond} <Rd>, <Rm>, <Rs>
	Unsigned 32 x 32, + two 32, to 64	UMAAL{cond} <RdLo>, <RdHi>, <Rm>, <Rs>
Saturate, select, and pack	Signed saturation at bit position n	SSAT{cond} <Rd>, #<immed_5>, <Rm>{, <shift>}
	Unsigned saturation at bit position n	USAT{cond} <Rd>, #<immed_5>, <Rm>{, <shift>}
	Two 16 signed saturation at bit position n	SSAT16{cond} <Rd>, #<immed_4>, <Rm>
	Two 16 unsigned saturation at bit position n	USAT16{cond} <Rd>, #<immed_4>, <Rm>
	Select bytes from Rn/Rm based on GE flags	SEL{cond} <Rd>, <Rn>, <Rm>
	Pack low 16/32, high 16/32	PKHBT{cond} <Rd>, <Rn>, <Rm>{, LSL #<immed_5>}
	Pack high 16/32, low 16/32	PKHTB{cond} <Rd>, <Rn>, <Rm>{, ASR #<immed_5>}

4.3. Advanced SIMD data-processing instructions

Table B.1 lists the UAL equivalents of the legacy Advanced SIMD data-processing assembly language mnemonics used in this manual. This table lists only those mnemonics that are different in the UAL syntax.

Table B.1. Advanced SIMD mnemonics

Legacy		UAL
Three registers of the same length:		
	VFMX	VPMAX
	VFMN	VPMIN
	VQ{R}DMLH	VQ{R}DMULH
	VSUM	VPADD
	VCAGE	VACGE
	VCAGT	VACGT
Three registers of different lengths:		
	VADD long	VADDL
	VSUB long	VSUBL
	VADD wide	VADDW
	VSUB wide	VSUBW
	V{R}ADH	V{R}ADDHN
	VABA long	VABAL
	V{R}SBH	V{R}SUBHN
	VABD long	VABDL
	VMLA long	VMLAL
	VQDMLA long	VQDMLAL

Legacy		UAL
	VMLS long	VMLSL
	VQDMLS long	VQDMLSL
	VMUL long	VMULL
	VQDMUL long	VQDMULL
	VMUL polynomial	VMULL
Two registers and a scalar:		
	VMLA long	VMLAL
	VQDMLA long	VQDMLAL
	VMLS long	VMLSL
	VQDMLS long	VQDMLSL
	VMUL long	VMULL
	VQDMUL long	VQDMULL
	VQ{R}DMLH	VQ{R}DMULH
Two registers and a shift amount:		
	VQSHL	VQSHL{U}
	V{R}SHR narrow	V{R}SHRN
	VQ{R}SHR narrow	VQ{R}SHR{U}N
	VSHL wide	VSHLL
Two registers, miscellaneous:		
	VSUM long	VPADDL
	VNOT	VMVN
	VSMA long	VPADAL
	VCGTZ	VCGT #0

Legacy		UAL
	VCGEZ	VCGE #0
	VCEQZ	VCEQ #0
	VCLEZ	VCLE #0
	VCLTZ	VCLT #0
	VMOV narrow	VMOVN
	VQMOV narrow	VQMOV{U}N
	VMVH wide	VSHLL
Move data element to all lanes of a register:		
	VMOV	VDUP

Advanced SIMD integer shift instructions

Instruction	Register format	Cycles	Source 1	Source 2	Source3	Source4	Result 1	Result 2
VSHR VSHL	Dd,Dm,#IMM	1	Dm:N1	-	-	-	Dd:N3	-
	Qd,Qm,#IMM	1	QmLo:N1	QmHi:N1	-	-	QdLo:N3	QdHi:N3
VQSHL VRSHR	Dd,Dm,#IMM	1	Dm:N1	-	-	-	Dd:N4	-
	Qd,Qm,#IMM	1	QmLo:N1	QmHi:N1	-	-	QdLo:N4	QdHi:N4
VSHR	Dd,Qm,#IMM (narrow)	1	QmLo:N1	QmHi:N1	-	-	Dd:N3	-
VQSHR VQMO	Dd,Qm,#IMM (narrow)	1	QmLo:N1	QmHi:N1	-	-	Dd:N4	-

Instruction	Register format	Cycles	Source 1	Source 2	Source3	Source4	Result 1	Result 2
VVRSHR VQRSHR								
VSHL[1] VMVH	Qd,Dm,[#IMM] (long, wide)		Dm:N1	-	-	-	QdLo:N3	QdHi:N3
VSLI VSRI	Dd,Dm,#IMM	1	Dm:N1	Dd:N1	-	-	Dd:N3	-
	Qd,Qm,#IMM	1 2	QmLo:N1 QmHi:N1	QdLo:N1 QdHi:N1	– –	– –	QdLo:N3 QdHi:N3	– –
VSHL	Dd,Dm,Dn	1	Dm:N1	Dn:N1	-	-	Dn:N1	-
	Qd,Qm,Qn	1 2	QmLo:N1 QmHi:N1	QnLo:N1 QnHi:N1	– –	– –	QdLo:N3 QdHi:N3	– –
VQSHL VRSHL VQRSHL	Dd,Dm,Dn	1	Dm:N1	Dn:N1	-	-	Dd:N4	-
	Qd,Qm,Qn	1 2	QmLo:N1 QmLo:N1	QnLo:N1 QnHi:N1	– –	– –	QdLo:N4 QdHi:N4	– –
VSRA VRSRA	Dd,Dm,#IMM	1	Dm:N1	-	Dd:N3	-	Dd:N6	-
	Qd,Qm,#IMM	1	QmLo:N1	QmHi:N1	QdLo:N3	QdHi:N3	QdLo:N6	QdHi:N6

4.3.1. Advanced SIMD integer ALU instructions

Table 16.15 shows the operation of the Advanced SIMD integer ALU instructions.

Table 16.15. Advanced SIMD integer ALU instructions

Instruction	Register format	Cycles	Source				Result	
			1	2	3	4	1	2
VADD VAND VORR VEOR VBIC VORN	Dd,Dn,Dm	1	Dn:N2	Dm:N2	-	-	Dd:N3	-
	Qd,Qn,Qm	1	QnLo:N2	QmLo:N2	QnHi:N2	QmHi:N2	QdLo:N3	QdHi:N3
VSUB	Dd,Dn,Dm	1	Dn:N2	Dm:N1	-	-	Dd:N3	-
	Qd,Qn,Qm	1	QnLo:N2	QmLo:N1	QnHi:N2	QmHi:N1	QdLo:N3	QdHi:N3
VADD VSUB	Qd,Dn,Dm (long)	1	Dn:N1	Dm:N1	-	-	QdLo:N3	QdHi:N3
	Qd,Qn,Dm (wide)	1	QnLo:N2	Dm:N1	QnHi:N2	-	QdLo:N3	QdHi:N3
VHADD VRHADD VQADD VTST	Dd,Dn,Dm	1	Dn:N2	Dm:N2	-	-	Dd:N4	-
	Qd,Qn,Qm	1	QnLo:N2	QmLo:N2	QnHi:N2	QmHi:N2	QdLo:N4	QdHi:N4
VADH VRADH	Dd,Qn,Qm (highhalf)	1	QnLo:N2	QmLo:N2	QnHi:N2	QmHi:N2	Dd:N4	-

Instruction	Register format	Cycles	Source				Result	
			1	2	3	4	1	2
VSBH VRSBH	Dd,Qn,Qm (highhalf)	1	QnLo:N2	QmLo:N2	QnHi:N2	QmHi:N1	Dd:N4	-
VHSUB VQSUB VABD VCEQ VCGE VCGT VMAX VMIN VFMX[1] VFMN[1]	Dd,Dn,Dm	1	Dn:N2	Dm:N1	-	-	Dd:N4	-
	Qd,Qn,Qm	1	QnLo:N2	QmLo:N1	QnHi:N2	QmHi:N1	QdLo:N4	QdHi:N4
VNEG	Dd,Dm	1	-	Dm:N1	-	-	Dd:N3	-
	Qd,Qm	1	-	QmLo:N1	-	QmHi:N1	QdLo:N3	QdHi:N3
VQNEG VQABS	Dd,Dm	1	-	Dm:N1	-	-	Dd:N4	-
	Qd,Qm	1	-	QmLo:N1	-	QmHi:N1	QdLo:N4	QdHi:N4
VABD	Qd,Dn,Dm (long)	1	Dn:N2	Dm:N1	-	-	QdLo:N4	QdHi:N4
VABS VCEQZ VCGEZ VCGTZ VCLEZ	Dd,Dm	1	Dm:N2	-	-	-	Dd:N4	-
	Qd,Qm	1	QmLo:N2	-	QmHi:N2	-	QdLo:N4	QdHi:N4

Instruction	Register format	Cycles	Source				Result	
			1	2	3	4	1	2
VCLTZ								
VSUM	Dd,Dn,Dm	1	Dn:N1	Dm:N1	-	-	Dd:N3	-
	Dd,Dm (long)	1	Dm:N1	-	-	-	Dd:N3	-
	Qd,Qm (long)	1	QmLo:N1	QmHi:N1	-	-	QdLo:N3	QdHi:N3
VNOT VCLS VCLZ VCNT	Dd,Dm	1	-	Dm:N2	-	-	Dd:N3	-
VNOT	Qd,Qm	1	-	QmLo:N2	-	QmHi:N2	QdLo:N3	QdHi:N3
VCLS VCLZ VCNT	Qd,Qm	1 2	– –	QmLo:N2 QmHi:N2	– –	– –	QdLo:N3 QdHi:N3	– –
VMOV VMVN	Dd,#IMM	1	-	-	-	-	Dd:N3	-
	Qd,#IMM	1	-	-	-	-	QdLo:N3	QdHi:N3
VORR VBIC	Dd,#IMM	1	Dd:N2	-	-	-	Dd:N3	-
	Qd,#IMM	1	QdLo:N2	-	Qdb:N2	-	QdLo:N3	QdHi:N3
VBIT VBIF VBSL	Dd,Dn,Dm	1	Dn:N2	Dm:N2	Dd:N2	-	Dd:N3	-
	Qd,Qn	1	QnLo:	QmLo:	QdLo:	–	QdLo:	–

Instruction	Register format	Cycles	Source				Result	
			1	2	3	4	1	2
	,Qm	2	N2 QnHi: N2	N2 QmHi: N2	N2 QdHi: N2	–	N3 QdHi: N3	–
VABA	Dd,Dn ,Dm	1	Dn:N2	Dm:N1	Dd:N3	-	Dd:N6	-
	Qd,Qn ,Qm	1 2	QnLo: N2 QnHi: N2	QmLo: N1 QmHi: N1	QdLo: N3 QdHi: N3	– –	QdLo: N6 QdHi: N6	– –
	Qd,Dn ,Dm (long)	1	Dn:N2	Dm:N1	QdLo: N3	QdHi: N3	QdLo: N6	QdHi: N6
VSMA	Dd,Dm (long)	1	Dm:N1	-	Dd:N3	-	Dd:N6	-
	Qd,Qm (long)	1	QmLo: N1	QmHi: N1	QdLo: N3	QdHi: N3	QdLo: N6	QdHi: N6
[1] VFMX and VFMN exist only for the Dd, Dn, Dm variant.								

4.3.2. Advanced SIMD floating-point instructions

Table 16.18 shows the operation of the Advanced SIMD floating-point instructions.

Table 16.18. Advanced SIMD floating-point instructions

Instruction	Register format	Cycles	Source 1	Source 2	Source 3	Source4	Result1	Result2
VADD VSUB VABD	Dd,Dn,Dm	1	Dn:N2	Dm:N2	-	-	Dd:N5	-
	Qd,Qn,Q	1	QnLo:	QmLo:	–	–	QdLo:	–

Instruction	Register format	Cycles	Source 1	Source 2	Source 3	Source4	Result1	Result2
VMUL VCEQ VCGE VCGT VCAGE VCAGT VMAX VMIN	m	2	N2 QnHi:N2	N2 QmHi:N2	–	–	N5 QdHi:N5	–
VABS VNEG VCEQZ VCGEZ VCGTZ VCLEZ VCLTZ VRECPE VRSQRTE VCVT	Dd,Dm	1	Dm:N2	-	-	-	Dd:N5	-
	Qd,Qm	1 2	QmLo:N2 QmHi:N2	– –	– –	– –	QdLo:N5 QdHi:N5	– –
VSUM VFMX VFMN	Dd,Dn,Dm	1	Dn:N1	Dm:N1	-	-	Dd:N5	-
VMUL	Dd,Dn,Dm[x] (scalar)	1	Dn:N2	Dm:N1	–	–	Dd:N5	-
	Qd,Qn,Dm[x] (scalar)	1 2	QnLo:N2 QnHi:N2	Dm:N1 –	– –	– –	QdLo:N5 QdHi:N5	– –
VMLA[1] VMLS[a]	Dd,Dn,Dm	1	Dn:N2	Dm:N2	Dd:N3	–	Dd:N9	–
	Qd,Qn,Qm	1 2	QnLo:N2 QnHi:N2	QmLo:N2 QmHi:N2	QdLo:N3 QdHi:N3	– –	QdLo:N9 QdHi:N9	– –

Instruction	Register format	Cycles	Source 1	Source 2	Source 3	Source4	Result1	Result2
	Dd,Dn,Dm[x] (scalar)	1	Dn:N2	Dm:N1	Dd:N3	–	Dd:N9	–
	Qd,Qn,Dm[x] (scalar)	1 2	QnLo:N2 QnHi:N2	Dm:N1 –	QdLo:N3 QdHi:N3	– –	QdLo:N9 QdHi:N9	– –
VRECPS[a] VRSQRTS[a]	Dd,Dn,Dm	1	Dn:N2	Dm:N2	–	–	Dd:N9	–
	Qd,Qn,Qm	1 2	QnLo:N2 QnHi:N2	QmLo:N2 QmHi:N2	– –	– –	QdLo:N9 QdHi:N9	– –

[1] The VMLA.F, VMLS.F, VRECPS.F, VRSQRTS.F instructions begin execution on the floating-point multiply pipeline. The floating-point multiply result is then forwarded to the floating-point add pipeline to complete the accumulate portion of the instructions. Therefore, these instructions are pipelined across ten stages, N1 through N10, where N10 is the writeback stage.

4.3.3. ARMv6 SIMD intrinsics

The ARM Architecture v6 Instruction Set Architecture adds over sixty SIMD instructions to ARMv6 for the efficient software implementation of high-performance media applications.
The ARM compiler supports intrinsics that map to the ARMv6 SIMD instructions. These intrinsics are available when compiling your code for an ARMv6 architecture or processor. The following list gives the function prototypes for these intrinsics. The function prototypes given in the list describe the primitive or basic forms of the ARMv6 instructions realized by the intrinsics. To obtain the name of the basic instruction realized by an intrinsic, drop the leading underscores (__) from the intrinsic

name. For example, the __qadd16 intrinsic corresponds to an ARMv6 QADD16 instruction.

4.3.4. Note

Each ARMv6 SIMD intrinsic is guaranteed to be compiled into a single, inline, machine instruction for an ARM v6 architecture or processor. However, the compiler might use optimized forms of underlying instructions when it detects opportunities to do so.

The ARMv6 SIMD instructions can set the GE[3:0] bits in the *Application Program Status Register* (APSR). The SIMD instructions might update these flags to indicate the "greater than or equal to" status of each 8/16-bit slice of a SIMD operation.

The ARM compiler treats the GE[3:0] bits as a global variable. To access these bits from within your C or C++ program, either:

- access bits 16-19 of the APSR through a named register variable
- use the __sel intrinsic to control a SEL instruction.

```
unsigned int __qadd16(unsigned int, unsigned int)
unsigned int __qadd8(unsigned int, unsigned int)
unsigned int __qasx(unsigned int, unsigned int)
unsigned int __qsax(unsigned int, unsigned int)
unsigned int __qsub16(unsigned int, unsigned int)
unsigned int __qsub8(unsigned int, unsigned int)
unsigned int __sadd16(unsigned int, unsigned int)
unsigned int __sadd8(unsigned int, unsigned int)
unsigned int __sasx(unsigned int, unsigned int)
unsigned int __sel(unsigned int, unsigned int)
unsigned int __shadd16(unsigned int, unsigned int)
unsigned int __shadd8(unsigned int, unsigned int)
unsigned int __shasx(unsigned int, unsigned int)
unsigned int __shsax(unsigned int, unsigned int)
unsigned int __shsub16(unsigned int, unsigned int)
unsigned int __shsub8(unsigned int, unsigned int)
unsigned int __smlad(unsigned int, unsigned int, unsigned int)
unsigned long long __smlald(unsigned int, unsigned int, unsigned long
long)
unsigned int __smlsd(unsigned int, unsigned int, unsigned int)
unsigned long long __smlsld(unsigned int, unsigned int, unsigned long
long)
unsigned int __smuad(unsigned int, unsigned int)
```

```
unsigned int __smusd(unsigned int, unsigned int)
unsigned int __ssat16(unsigned int, unsigned int)
unsigned int __ssax(unsigned int, unsigned int)
unsigned int __ssub16(unsigned int, unsigned int)
unsigned int __ssub8(unsigned int, unsigned int)
unsigned int __sxtab16(unsigned int, unsigned int)
unsigned int __sxtb16(unsigned int, unsigned int)
unsigned int __uadd16(unsigned int, unsigned int)
unsigned int __uadd8(unsigned int, unsigned int)
unsigned int __uasx(unsigned int, unsigned int)
unsigned int __uhadd16(unsigned int, unsigned int)
unsigned int __uhadd8(unsigned int, unsigned int)
unsigned int __uhasx(unsigned int, unsigned int)
unsigned int __uhsax(unsigned int, unsigned int)
unsigned int __uhsub16(unsigned int, unsigned int)
unsigned int __uhsub8(unsigned int, unsigned int)
unsigned int __uqadd16(unsigned int, unsigned int)
unsigned int __uqadd8(unsigned int, unsigned int)
unsigned int __uqasx(unsigned int, unsigned int)
unsigned int __uqsax(unsigned int, unsigned int)
unsigned int __uqsub16(unsigned int, unsigned int)
unsigned int __uqsub8(unsigned int, unsigned int)
unsigned int __usad8(unsigned int, unsigned int)
unsigned int __usada8(unsigned int, unsigned int, unsigned int)
unsigned int __usax(unsigned int, unsigned int)
unsigned int __usat16(unsigned int, unsigned int)
unsigned int __usub16(unsigned int, unsigned int)
unsigned int __usub8(unsigned int, unsigned int)
unsigned int __uxtab16(unsigned int, unsigned int)
unsigned int __uxtb16(unsigned int, unsigned int)
```