Linux虚拟地址空间布局以及进程栈和线程栈总结

目录(?)[+]

本文转自多个博客，以及最后有我的总结。我没有单独从头到尾写一个总结的原因是别人已经写得很好了，我不花大量时间是无法达到这水平的。

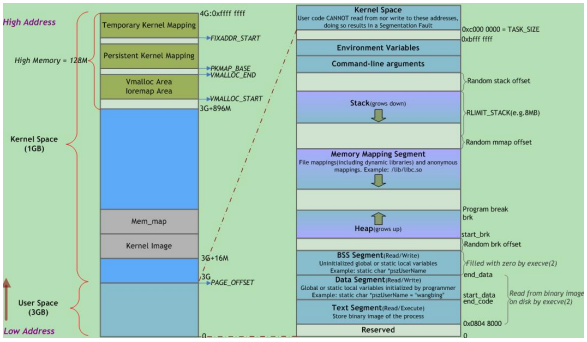
一：**Linux虚拟地址空间布局**

（转自：Linux虚拟地址空间布局)

在多任务操作系统中，每个进程都运行在属于自己的内存沙盘中。这个沙盘就是虚拟地址空间(Virtual Address Space)，在32位模式下它是一个4GB的内存块

虚拟地址通过页表(Page Table)映射到物理内存，页表由操作系统维护并被处理器引用。内核空间在页表中拥有较高特权级，因此用户态程序试图访问这些

Linux进程在虚拟内存中的标准内存段布局如下图所示：



其中，用户地址空间中的蓝色条带对应于映射到物理内存的不同内存段，灰白区域表示未映

上图中Random stack offset和Random mmap offset等随机值意在防止恶意程序。Linux通过

用户进程部分分段存储内容如下表所示(按地址递减顺序)：

名称　存储内容

栈　局部变量、函数参数、返回地址等

堆　动态分配的内存

BSS段未初始化或初值为0的全局变量和静态局部变量

数据段已初始化且初值非0的全局变量和静态局部变量

代码段可执行代码、字符串字面值、只读变量

在将应用程序加载到内存空间执行时，操作系统负责代码段、数据段和BSS段的加载，并在内存中为这些段分配空间。栈也由操作系统分配和管理；堆由

BSS段、数据段和代码段是可执行程序编译时的分段，运行时还需要栈和堆。

以下详细介绍各个分段的含义。

1 内核空间

内核总是驻留在内存中，是操作系统的一部分。内核空间为内核保留，不允许应用程序读写该区域的内容或直接调用内核代码定义的函数。

2 栈(stack)

栈又称堆栈，由编译器自动分配释放，行为类似数据结构中的栈(先进后出)。堆栈主要有三个用途：

- 为函数内部声明的非静态局部变量(C语言中称“自动变量”)提供存储空间。
- 记录函数调用过程相关的维护性信息，称为栈帧(Stack Frame)或过程活动记录(Procedure Activation Record)。它包括函数返回地址，不适合装入
- 临时存储区，用于暂存长算术表达式部分计算结果或alloca()函数分配的栈内内存。

持续地重用栈空间有助于使活跃的栈内存保持在CPU缓存中，从而加速访问。进程中的每个线程都有属于自己的栈。向栈中不断压入数据时，若超出其容

Linux中ulimit -s命令可查看和设置堆栈最大值，当程序使用的堆栈超过该值时, 发生栈溢出(Stack Overflow)，程序收到一个段错误(Segmentation Fault)。

堆栈既可向下增长(向内存低地址)也可向上增长, 这依赖于具体的实现。本文所述堆栈向下增长。

堆栈的大小在运行时由内核动态调整。

3 内存映射段(mmap)

此处，内核将硬盘文件的内容直接映射到内存, 任何应用程序都可通过Linux的mmap()系统调用或Windows的CreateFileMapping()/MapViewOfFile()请求这

该区域用于映射可执行文件用到的动态链接库。在Linux 2.4版本中，若可执行文件依赖共享库，则系统会为这些动态库在从0x40000000开始的地址分配

从进程地址空间的布局可以看到，在有共享库的情况下，留给堆的可用空间还有两处：一处是从.bss段到0x40000000，约不到1GB的空间；另一处是从共

4 堆(heap)

堆用于存放进程运行时动态分配的内存段，可动态扩张或缩减。堆中内容是匿名的，不能按名字直接访问，只能通过指针间接访问。当进程调用malloc(C

分配的堆内存是经过字节对齐的空间，以适合原子操作。堆管理器通过链表管理每个申请的内存，由于堆申请和释放是无序的，最终会产生内存碎片。堆

堆的末端由break指针标识，当堆管理器需要更多内存时，可通过系统调用brk()和sbrk()来移动break指针以扩张堆，一般由系统自动调用。

使用堆时经常出现两种问题：1) 释放或改写仍在使用的内存(“内存破坏”)；2)未释放不再使用的内存(“内存泄漏”)。当释放次数少于申请次数时，可能已造

注意，堆不同于数据结构中的”堆”，其行为类似链表。

【扩展阅读】栈和堆的区别

①管理方式：栈由编译器自动管理；堆由程序员控制，使用方便，但易产生内存泄露。

②生长方向：栈向低地址扩展(即”向下生长”)，是连续的内存区域；堆向高地址扩展(即”向上生长”)，是不连续的内存区域。这是由于系统用链表来存储空闲内

③空间大小：栈顶地址和栈的最大容量由系统预先规定(通常默认2M或10M)；堆的大小则受限于计算机系统中有效的虚拟内存，32位Linux系统中堆内存可达

④存储内容：栈在函数调用时，首先压入主调函数中下条指令(函数调用语句的下条可执行语句)的地址，然后是函数实参，然后是被调函数的局部变量。本次

⑤分配方式：栈可静态分配或动态分配。静态分配由编译器完成，如局部变量的分配。动态分配由alloca函数在栈上申请空间，用完后自动释放。堆只能动态

⑥分配效率：栈由计算机底层提供支持：分配专门的寄存器存放栈地址，压栈出栈由专门的指令执行，因此效率较高。堆由函数库提供，机制复杂，效率比栈

⑦分配后系统响应：只要栈剩余空间大于所申请空间，系统将为程序提供内存，否则报告异常提示栈溢出。

操作系统为堆维护一个记录空闲内存地址的链表。当系统收到程序的内存分配申请时，会遍历该链表寻找第一个空间大于所申请空间的堆结点，然后将该

此外，由于找到的堆结点大小不一定正好等于申请的大小，系统会自动将多余的部分重新放入空闲链表中。

⑧碎片问题：栈不会存在碎片问题，因为栈是先进后出的队列，内存块弹出栈之前，在其上面的后进的栈内容已弹出。而频繁申请释放操作会造成堆内存空

可见，堆容易造成内存碎片；由于没有专门的系统支持，效率很低；由于可能引发用户态和内核态切换，内存申请的代价更为昂贵。所以栈在程序中应用

使用栈和堆时应避免越界发生，否则可能程序崩溃或破坏程序堆、栈结构，产生意想不到的后果。

5 BSS段

BSS(Block Started by Symbol)段中通常存放程序中以下符号：

- 未初始化的全局变量和静态局部变量
- 初始值为0的全局变量和静态局部变量(依赖于编译器实现)
- 未定义且初值不为0的符号(该初值即common block的大小)

C语言中，未显式初始化的静态分配变量被初始化为0(算术类型)或空指针(指针类型)。由于程序加载时，BSS会被操作系统清零，所以未赋初值或初值为0

注意，尽管均放置于BSS段，但初值为0的全局变量是强符号，而未初始化的全局变量是弱符号。若其他地方已定义同名的强符号(初值可能非0)，则弱符

某些编译器将未初始化的全局变量保存在common段，链接时再将其放入BSS段。在编译阶段可通过-fno-common选项来禁止将未初始化的全局变量放入

此外，由于目标文件不含BSS段，故程序烧入存储器(Flash)后BSS段地址空间内容未知。U-Boot启动过程中，将U-Boot的Stage2代码(通常位于lib_xxxx/

【扩展阅读】BSS历史

BSS(Block Started by Symbol，以符号开始的块)一词最初是UA-SAP汇编器(United Aircraft Symbolic Assembly Program)中的伪指令，用于为符号预留一

后来该词被作为关键字引入到了IBM 709和7090/94机型上的标准汇编器FAP(Fortran Assembly Program)，用于定义符号并且为该符号预留指定字数的未

在采用段式内存管理的架构中(如Intel 80x86系统)，BSS段通常指用来存放程序中未初始化全局变量的一块内存区域，该段变量只有名称和大小却没有值。

BSS段不包含数据，仅维护开始和结束地址，以便内存能在运行时被有效地清零。BSS所需的运行时空间由目标文件记录，但BSS并不占用目标文件内的

6 数据段**(Data)**

数据段通常用于存放程序中已初始化且初值不为0的全局变量和静态局部变量。数据段属于静态内存分配(静态存储区)，可读可写。

数据段保存在目标文件中(在嵌入式系统里一般固化在镜像文件中)，其内容由程序初始化。例如，对于全局变量int gVar = 10，必须在目标文件数据段中保

数据段与BSS段的区别如下：

1) BSS段不占用物理文件尺寸，但占用内存空间；数据段占用物理文件，也占用内存空间。

对于大型数组如int ar0[10000] = {1, 2, 3, ...}和int ar1[10000]，ar1放在BSS段，只记录共有10000*4个字节需要初始化为0，而不是像ar0那样记录每个数据

2) 当程序读取数据段的数据时，系统会出发缺页故障，从而分配相应的物理内存；当程序读取BSS段的数据时，内核会将其转到一个全零页面，不会发生

运行时数据段和BSS段的整个区段通常称为数据区。某些资料中“数据段”指代数据段 + BSS段 + 堆。

7 代码段**(text)**

代码段也称正文段或文本段，通常用于存放程序执行代码(即CPU执行的机器指令)。一般c语言执行语句都编译成机器代码保存在代码段。通常代码段是可

代码段指令根据程序设计流程依次执行，对于顺序指令，只会执行一次(每个进程)；若有反复，则需使用跳转指令；若进行递归，则需要借助栈来实现。

代码段指令中包括操作码和操作对象(或对象地址引用)。若操作对象是立即数(具体数值)，将直接包含在代码中；若是局部数据，将在栈区分配空间，然后

代码段最容易受优化措施影响。

8 保留区

位于虚拟地址空间的最低部分，未赋予物理地址。任何对它的引用都是非法的，用于捕捉使用空指针和小整型值指针引用内存的异常情况。

它并不是一个单一的内存区域，而是对地址空间中受到操作系统保护而禁止用户进程访问的地址区域的总称。大多数操作系统中，极小的地址通常都是不

在32位X86架构的Linux系统中，用户进程可执行程序一般从虚拟地址空间0x08048000开始加载。该加载地址由ELF文件头决定，可通过自定义链接器脚本

通过cat /proc/self/maps命令查看加载表如下：

```
00405000-00406000 r-xp 00405000 00:00 0          [vdso]
0051b000-00535000 r-xp 00000000 fd:00 28871142   /lib/ld-2.5.so
00535000-00536000 r-xp 00019000 fd:00 28871142   /lib/ld-2.5.so
00536000-00537000 rwxp 0001a000 fd:00 28871142   /lib/ld-2.5.so
0053d000-0067c000 r-xp 00000000 fd:00 28871143   /lib/libc-2.5.so
0067c000-0067d000 --xp 0013f000 fd:00 28871143   /lib/libc-2.5.so
0067d000-0067f000 r-xp 0013f000 fd:00 28871143   /lib/libc-2.5.so
0067f000-00680000 rwxp 00141000 fd:00 28871143   /lib/libc-2.5.so
00680000-00683000 rwxp 00680000 00:00 0
08048000-0804d000 r-xp 00000000 fd:00 21594152   /bin/cat
0804d000-0804e000 rw-p 00004000 fd:00 21594152   /bin/cat
09cc1000-09ce2000 rw-p 09cc1000 00:00 0          [heap]
b7fc6000-b7fc8000 rw-p b7fc6000 00:00 0
bfd04000-bfd19000 rw-p bffea000 00:00 0          [stack]
```

【扩展阅读】分段的好处

进程运行过程中，代码指令根据流程依次执行，只需访问一

当程序被装载后，数据和指令分别映射到两个虚存区域。数

现代CPU具有极为强大的缓存(Cache)体系，程序必须尽量提

当系统中运行多个该程序的副本时，其指令相同，故内存中

此外，临时数据及需要再次使用的代码在运行时放入栈区中

二：Linux 中的各种栈：进程栈 线程栈 内核栈 中断栈

（转自：Linux 中的各种栈：进程栈 线程栈 内核栈 中断栈，不过我只转了他的部分内容，感兴趣可以去看）

Linux 中有几种栈？各种栈的内存位置？

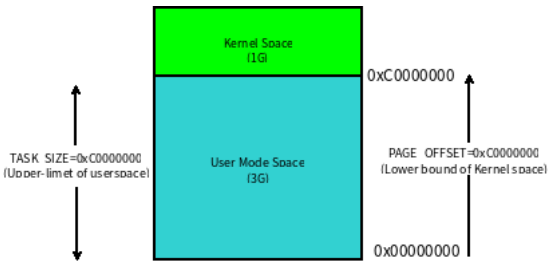
介绍完栈的工作原理和用途作用后，我们回归到 Linux 内核上来。内核将栈分成四种：

- 进程栈
- 线程栈
- 内核栈
- 中断栈

一、进程栈

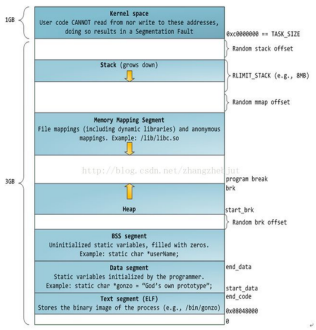
进程栈是属于用户态栈，和进程 虚拟地址空间 (Virtual Address Space) 密切相关。那我们先了解下什么是虚拟地址空间：在 32 位机器下，虚拟地址空间大小

Linux 内核将这 4G 字节的空间分为两部分，将最高的 1G 字节（0xC0000000-0xFFFFFFFF）供内核使用，称为内核空间。而将较低的3G字节（0x000000



Linux虚拟地址空间

Linux 对进程地址空间有个标准布局，地址空间中由各个不同的内存段组成 (Memory Segment)，
- 程序段 (Text Segment)：可执行文件代码的内存映射
- 数据段 (Data Segment)：可执行文件的已初始化全局变量的内存映射
- BSS段 (BSS Segment)：未初始化的全局变量或者静态变量（用零页初始化）
- 堆区 (Heap) : 存储动态内存分配，匿名的内存映射
- 栈区 (Stack) : 进程用户空间栈，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
- 映射段(Memory Mapping Segment)：任何内存映射文件



Linux标准进程内存段布局

而上面进程虚拟地址空间中的栈区，正指的是我们所说的进程栈

【扩展阅读】：如何确认进程栈的大小

我们要知道栈的大小，那必须得知道栈的起始地址和结束地址。

```
/* file name: stacksize.c */

void *orig_stack_pointer;

void blow_stack() {
    blow_stack();
}

int main() {
    __asm__("movl %esp, orig_stack_pointer");

    blow_stack();
    return 0;
}<ul><li>1</li><li>2</li><li>3</li><li>4</li><li>5</li><li>6</li><li>7</li><li>8</li><li>9</li><li>10</li><li>11</li><li>12<,

$ g++ -g stacksize.c -o ./stacksize
$ gdb ./stacksize
(gdb) r
Starting program: /home/home/misc-code/setrlimit

Program received signal SIGSEGV, Segmentation fault.
blow_stack () at setrlimit.c:4
4       blow_stack();
(gdb) print (void *)$esp
$1 = (void *) 0xfffffffffff7ff000
(gdb) print (void *)orig_stack_pointer
$2 = (void *) 0xffffc800
(gdb) print 0xffffc800-0xff7ff000
$3 = 8378368    // Current Process Stack Size is 8M<ul><li>1</li><li>2</li><li>3</li><li>4</li><li>5</li><li>6</li><li>7</li><li>
```

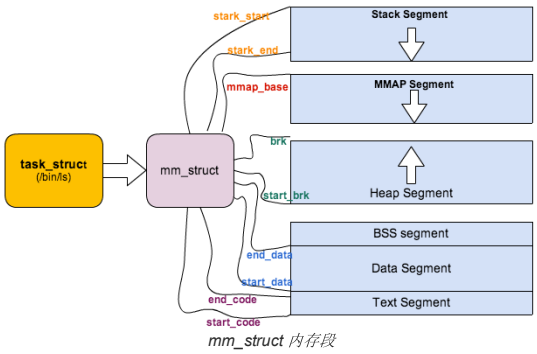
上面对进程的地址空间有个比较全局的介绍，那我们看下 Linux 内核中是怎么体现上面内存布局的。内核使用内存描述符来表示进程的地址空间，该描述符表

```
struct mm_struct {
    struct vm_area_struct *mmap;           /* 内存区域链表 */
    struct rb_root mm_rb;                  /* VMA 形成的红黑树 */
    ...
    struct list_head mmlist;               /* 所有 mm_struct 形成的链表 */
    ...
    unsigned long total_vm;                /* 全部页面数目 */
    unsigned long locked_vm;               /* 上锁的页面数据 */
    unsigned long pinned_vm;               /* Refcount permanently increased */
    unsigned long shared_vm;               /* 共享页面数目 Shared pages (files) */
    unsigned long exec_vm;                 /* 可执行页面数目 VM_EXEC & ~VM_WRITE */
    unsigned long stack_vm;                /* 栈区页面数目 VM_GROWSUP/DOWN */
    unsigned long def_flags;
    unsigned long start_code, end_code, start_data, end_data;    /* 代码段、数据段 起始地址和结束地址 */
    unsigned long start_brk, brk, start_stack;                   /* 栈区 的起始地址，堆区 起始地址和结束地址 */
    unsigned long arg_start, arg_end, env_start, env_end;        /* 命令行参数 和 环境变量的 起始地址和结束地址 */
    ...
    /* Architecture-specific MM context */
    mm_context_t context;                  /* 体系结构特殊数据 */

    /* Must use atomic bitops to access the bits */
    unsigned long flags;                   /* 状态标志位 */
    ...
    /* Coredumping and NUMA and HugePage 相关结构体 */
};<ul><li>1</li><li>2</li><li>3</li><li>4</li><li>5</li><li>6</li><li>7</li><li>8</li><li>9</li><li>10</li><li>11</li><li>12<

            </div>
```

收藏到代码笔记



mm_struct 内存段

【扩展阅读】：进程栈的动态增长实现

进程在运行的过程中，通过不断向栈区压入数据，当超出栈区容量时，就会耗尽

如果栈的大小低于 RLIMIT_STACK（通常为8MB），那么一般情况下栈会被加长

动态栈增长是唯一一种访问未映射内存区域而被允许的情形，其他任何对未映射

二、线程栈

从 Linux 内核的角度来说，其实它并没有线程的概念。Linux 把所有线程都当做

```
 if (clone_flags & CLONE_VM) {
    /*
     * current 是父进程而 tsk 在 fork() 执行期间是共享子进程
     */
    atomic_inc(&current->mm->mm_users);
    tsk->mm = current->mm;
  }<ul><li>1</li><li>2</li><li>3</li><li>4</li><li>5</li><li>6</li><li>7</li></ul><ul><li>1</li><li>2</li><li>3</li><li>4</li>
```

收藏到代码笔记

收藏到代码笔记 虽然线程的地址空间和进程一样，但是对待其地址空间的 stack 还是有些区别的。对于 Linux 进程或者说主线程，其 stack 是在 fork 的时候生

```
mem = mmap (NULL, size, prot,
            MAP_PRIVATE | MAP_ANONYMOUS | MAP_STACK, -1, 0);<ul><li>1</li><li>2</li></ul><ul><li>1</li><li>2</li></ul><div>
```

收藏到代码笔记

收藏到代码笔记 由于线程的 mm->start_stack 栈地址和所属进程相同，所以线程栈的起始地址并没有存放在 task_struct 中，应该是使用 pthread_attr_

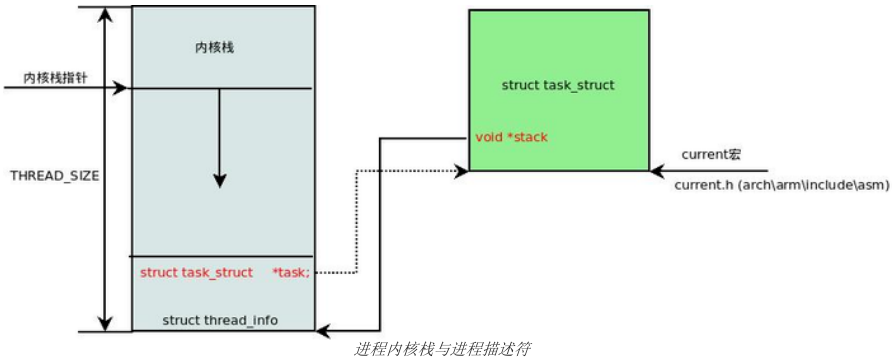
三、进程内核栈

在每一个进程的生命周期中，必然会通过到系统调用陷入内核。在执行系统调用陷入内核之后，这些内核代码所使用的栈并不是原先进程用户空间中的栈，而

```
union thread_union {
        struct thread_info thread_info;
        unsigned long stack[THREAD_SIZE/sizeof(long)];
};                                                    <ul><li>1</li><li>2</li><li>3</li><li>4</li></ul><ul><li>1</li><li>2</li>
```

收藏到代码笔记

收藏到代码笔记 thread_union 进程内核栈 和 task_struct 进程描述符有着紧密的联系。由于内核经常要访问 task_struct，高效获取当前进程的描述符是



进程内核栈与进程描述符

有了上述关联结构后，内核可以先获取到栈顶指针 esp，然后

这段描述，也就是 current 宏的实现方法：

```
register unsigned long current_stack_pointer asm

static inline struct thread_info *current_thread
{
        return (struct thread_info *)
                (current_stack_pointer & ~(THREA
}

#define get_current() (current_thread_info()->ta

#define current get_current()
```

收藏到代码笔记

收藏到代码笔记 四、中断栈

进程陷入内核态的时候，需要内核栈来支持内核函数调用。中断也是如此，当系统收到中断事件后，进行中断处理的时候，也需要中断栈来支持函数调用。

X86 上中断栈就是独立于内核栈的；独立的中断栈所在内存空间的分配发生在 arch/x86/kernel/irq_32.c 的irq_ctx_init() 函数中(如果是多处理器系统

而 ARM 上中断栈和内核栈则是共享的；中断栈和内核栈共享有一个负面因素，如果中断发生嵌套，可能会造成栈溢出，从而可能会破坏到内核栈的一些重要

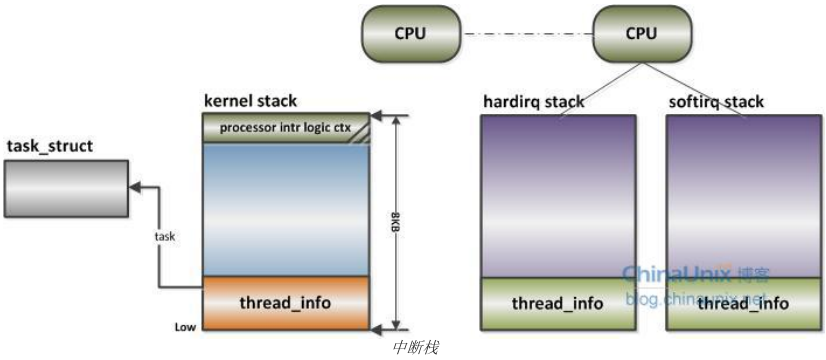
Linux 为什么需要区分这些栈？

为什么需要区分这些栈，其实都是设计上的问题。这里就我看到过的一些观点进行汇总，供大家讨论：

1.

为什么需要单独的进程内核栈？

- 所有进程运行的时候，都可能通过系统调用陷入内核态继续执行。假设第一个进程 A 陷入内核态执行的时候，需要等待读取网卡的数据，主动



2.

为什么需要单独的线程栈？

- Linux 调度程序中并没有区分线程和进程，当调度程序需要唤醒
  - 此时 A1 的栈指针 esp 如果为初始值 0x7ffc80000000，则线程
  - 如果此时线程 A1 的栈指针和父进程最后更新的值一致，esp 为

3.

进程和线程是否共享一个内核栈？

- No，线程和进程创建的时候都调用 `dup_task_struct` 来创建 t

4.

为什么需要单独中断栈？

- 这个问题其实不对，ARM 架构就没有独立的中断栈。

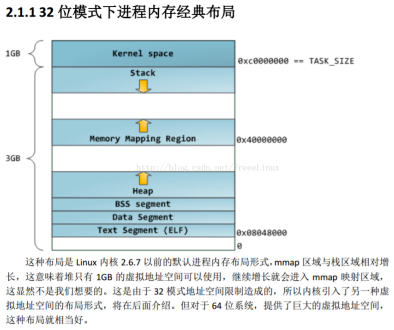
三：自己的总结

上面的图都很好，但我觉得这张图更形象，32位进程栈大小是8M，理论上堆区最大大小约为2.9G，所以还是蛮大的。

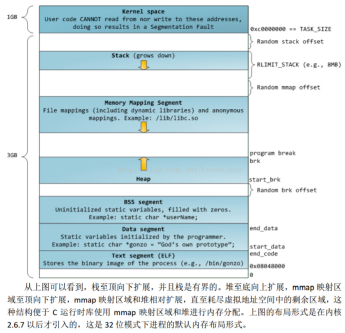
从上面两篇文章，我知道的线程栈是使用mmap系统调用分配的空间，但是mmap分配的系统空间是什么呢？也就是上图中的mmap区域或者说共享的内存映

下面两幅图给出了答案：

图一：



图二：



所以，mmap其实和堆一样，实际上可以说他们都是动态内存分配，但是

这里要提到一个很重要的概念，内存的延迟分配，只有在真正访问一个地

这篇文章关于mmap生长方向说的也挺详细的： 进程地址空间的布局（整

最后还有一个mmap机制的源代码分析博客，我水平暂时不够，只能看懂