下载APP

03 | 内存布局：应用程序是如何安排数据的？

2021-10-29 海纳

《编程高手必学的内存知识》 课程介绍 >

讲述：海纳
时长 21:16 大小 19.48M

你好，我是海纳。

在前边的课程里，我们学习了计算机物理地址和虚拟地址的概念。有了虚拟地址之后，运行在系统里的用户进程看到的地址空间范围，都是虚拟地址空间范围（32 位计算机的地址范围是 4G；64 位计算机的地址范围是 256T）。这样的话，就不用再担心内存地址不够用，以及与其他进程之间产生内存地址冲突的问题了。

前面几节课，我们关注的是如何解决进程之间的冲突，从这节课起，我们一起来看下进程内部的虚拟内存布局，或者说单一进程是如何安排自己的各种数据的。

学习了这节课，你将理解全局变量和 static 变量在内存中的位置以及初始化时机，在这个基础上，你还将明白在栈上创建对象和在堆上创建对象有什么不同等问题。这些问题的核

心都可以归结到“内存是如何布局的”这个问题上，所以只有深刻地掌握了内存布局的知识，你才能做到以不变应万变，面对各种具体问题才有了分析的方向和思路，进而，你才能写出更加“内存安全”的代码。

首先，我们来看一下，对于一个典型的进程来说，它的内存空间是由哪些部分组成的？每个部分又被安置在空间的什么位置？

抽象内存布局

我们知道，CPU 运行一个程序，实质就是在顺序执行该程序的机器码。一个程序的机器码会被组织到同一个地方，这个地方就是**代码段**。

另外，程序在运行过程中必然要操作数据。这其中，对于有初值的变量，它的初始值会存放在程序的二进制文件中，而且，这些数据部分也会被装载到内存中，即程序的**数据段**。数据段存放的是程序中已经初始化且不为 0 的全局变量和静态变量。

对于未初始化的全局变量和静态变量，因为编译器知道它们的初始值都是 0，因此便不需要再在程序的二进制映像中存放这么多 0 了，只需要记录他们的大小即可，这便是 **BSS 段**。BSS 段这个缩写名字是 Block Started by Symbol，但很多人可能更喜欢把它记作 Better Save Space 的缩写。

数据段和 BSS 段里存放的数据也只能是部分数据，主要是全局变量和静态变量，但程序在运行过程中，仍然需要记录大量的临时变量，以及运行时生成的变量，这里就需要新的内存区域了，即程序的**堆空间**跟**栈空间**。与代码段以及数据段不同的是，堆和栈并不是从磁盘中加载，它们都是由程序在运行的过程中申请，在程序运行结束后释放。

总的来说，一个程序想要运行起来所需要的几块基本内存区域：代码段、数据段、BSS 段、堆空间和栈空间。下面就是内存布局的示意图：

抽象内存布局

这是程序运行起来所需要的最小功能集，如果你尝试去看 Linux 0.11 的内核代码的话，会发现它所支持的 a.out 文件格式和内存布局就是上边的样子。

除了上面所讲的基本内存区域外，现代应用程序中还会包含其他的一些内存区域，主要有以下几类：

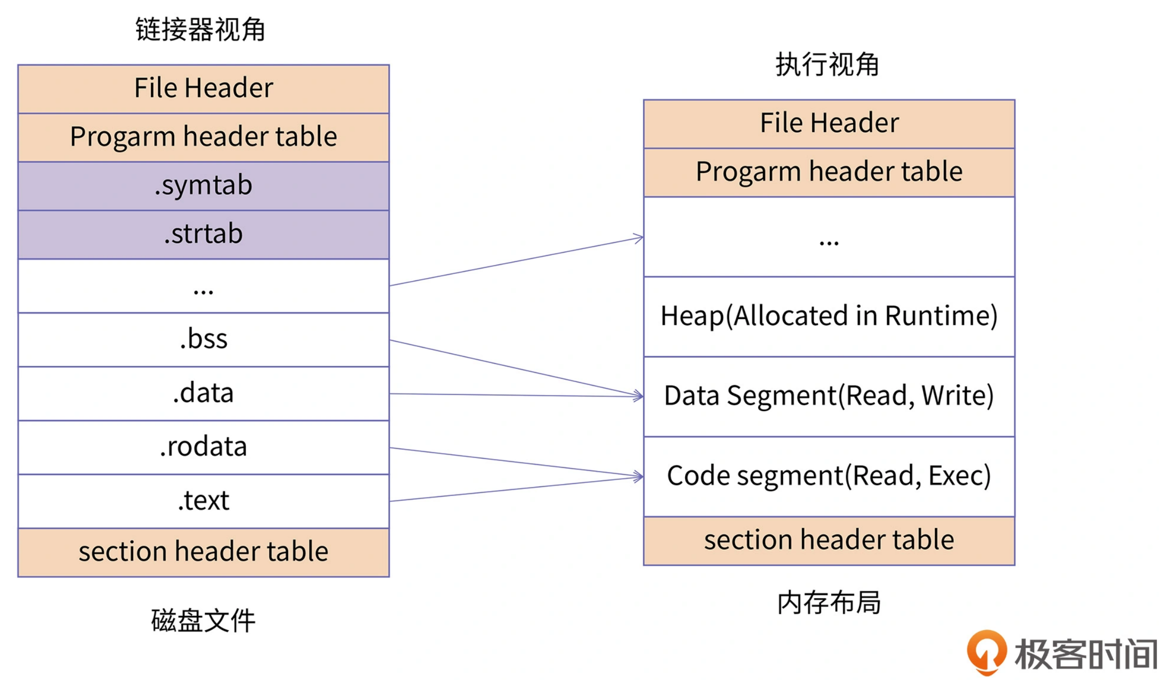
存放加载的共享库的内存空间：如果一个进程依赖共享库，那对应的，该共享库的代码段、数据段、BSS 段也需要被加载到这个进程的地址空间中。

共享内存段：我们可以通过系统调用映射一块匿名区域作为共享内存，用来进行进程间通信。

内存映射文件：我们也可以将磁盘的文件映射到内存中，用来进行文件编辑或者是类似共享内存的方式进行进程通信。

这样我们就初步了解了一个进程内存中需要哪些区域。

在上面的讨论中，我们并没有区分磁盘的程序段 (Section)，以及内存程序段 (Segment) 的概念，这两个词在国内往往都被翻译成 “段” ，导致大多数同学会混淆它们。这里我来给你做一个区分。



上图从两个视角展示了应用程序的分布，左边是程序在磁盘中的文件布局结构，右边是程序加载到内存中的内存布局结构。

对于磁盘的程序，每一个单元结构称为 Section。我们可以通过 readelf -S 的选项，来查看二进制文件中所有的 Section 信息。对于右边的内存镜像，每一个单元结构称为 Segment。我们可以通过 readelf -l 的选项，来查看二进制文件加载到内存之后的 Segment 布局信息。

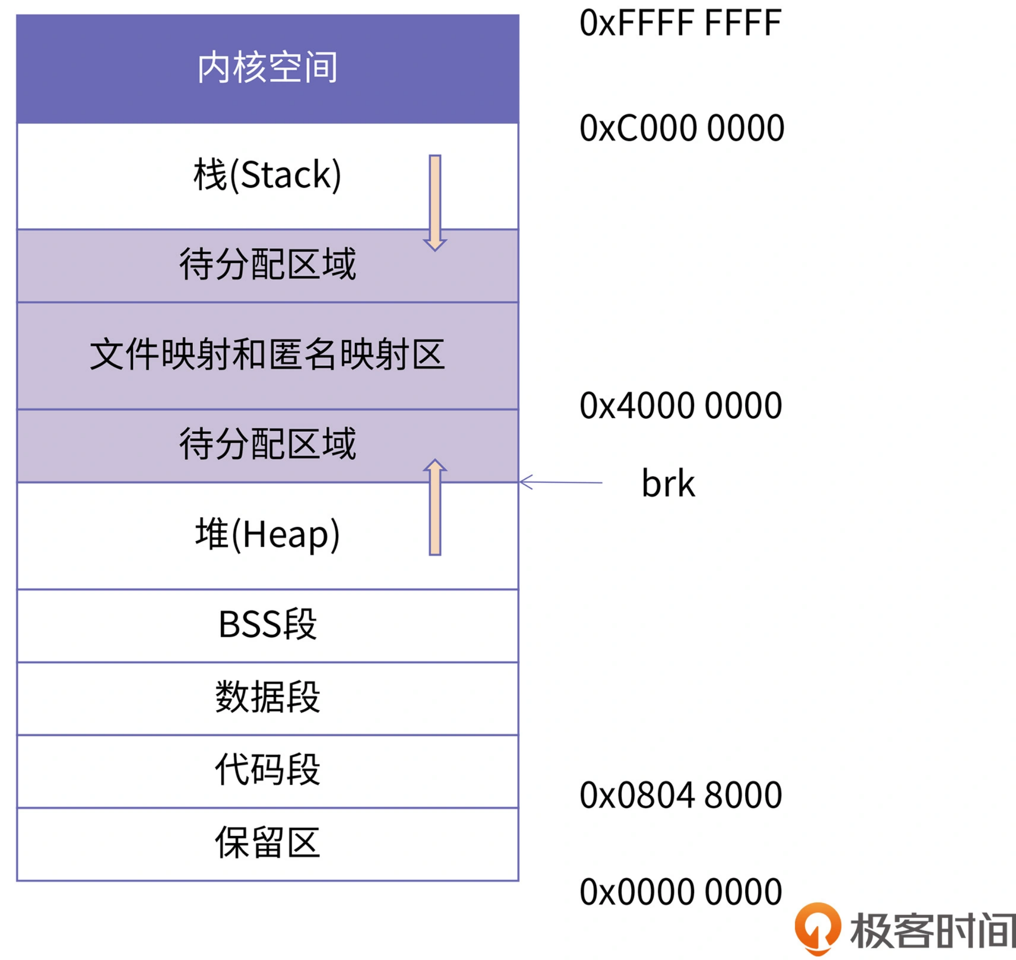
同时我们也可以看到，往往多个 Section 会对应一个 Segment，例如.text、.rodata 等一些只读的 Section，会被映射到内存的一个只读 / 执行的 Segment 里；而.data、.bss 等一些可读写的 Section，则会被映射到内存的一个具有读写权限的 Segment 里。并且对于磁盘二进制中一些辅助信息的 Section，例如.symtab、.strtab 等，不需要在内存中进行映射。

总的来说，**Section 主要是指在磁盘中的程序段，而 Segment 则用来指代内存中的程序段，Segment 是将具有相同权限属性的 Section 集合在一起，系统为它们分配的一块内存空间。**

接下来，我们就具体看下 Linux 系统下内存布局是怎样的。

IA-32 机器上的 Linux 进程内存布局

在 32 位机器上，每个进程都具有 4GB 的寻址能力。Linux 系统会默认将高地址的 1GB 空间分配给内核，剩余的低 3GB 是用户可以使用的用户空间。下图是 32 位机器上 Linux 进程的一个典型的内存布局。在实践中，我们可以通过`cat /proc/pid/maps`来查看某个进程的实际虚拟内存布局。



现在，我们从低地址到高地址，依次来解释下图中的布局情况。

首先，我们发现在 32 位 Linux 系统下，从 0 地址开始的内存区域并不是直接就是代码段区域，而是一段不可访问的保留区。这是因为在大多数的系统里，我们认为比较小数值的地址不是一个合法地址，例如，我们通常在 C 的代码里会将无效的指针赋值为 NULL。因此，这里会出现一段不可访问的内存保留区，防止程序因为出现 bug，导致读或写了一些小内存地址的数据，而使得程序跑飞。

接下来，我们可以看到，代码段从 0x08048000 的位置开始排布（需要注意的是，以上地址需要 gcc 编译的时候不开启 pie 的选项）。就像我们前面提到的，代码段、数据段都是从可执行文件映像中装载到内存中；BSS 段则是根据 BSS 段所需的大小，在加载时生成一段 0 填充的内存空间。

紧接着，排在 BSS 段后边的就是堆空间了。在图中，堆的空间里有一个向上的箭头，这里标明了堆地址空间的增长方向，也就是说，**每次在进程向内核申请新的堆地址时候，其地址的值是在增大的**。与之对应的是栈空间，有一个向下的箭头，说明栈增长的方向是向低地址方向增长，也就是说，**每次进程申请新的栈地址时，其地址值是在减少的**。

对此，我们可以想象堆和栈分别由两个指针控制，堆指针指明了当前堆空间的边界，栈指针指明了当前栈空间的边界。当堆申请新的内存空间时，只需要将堆指针增加对应的大小，回收地址时减少对应的大小即可。而栈的申请刚好相反。这其实就是内核对堆跟栈使用的最根本的方式，其中，堆的指针叫做“Program break”，栈的指针叫做“Stack pointer”，也就是 x86 架构下的 sp 寄存器。我们在后续的课程中会分别展开堆空间跟栈空间的实现原理。

继续往下看，就到了内存映射区域，这里最常见的就是程序所依赖的共享库，例如 libc.so。共享库的代码段、数据段、BSS 段都会被装载到这里。

这里我要说明一点，我们上述的布局分析都是基于 Linux 系统下关闭了进程地址随机化的选项。如果打开进程地址随机化的模式，其中的堆空间、栈空间和共享库映射的地址，在每次程序运行下都会不一样。这是因为内核在加载的过程中，会对这些区域的起始地址增加一些随机的偏移值，这能增加缓冲区溢出的难度。

对于这个进程地址随机化选项，我们可以通过 `sudo sysctl -w kernel.randomize_va_space=val`的命令来设置。其中，val=0 表示关闭内存地址随

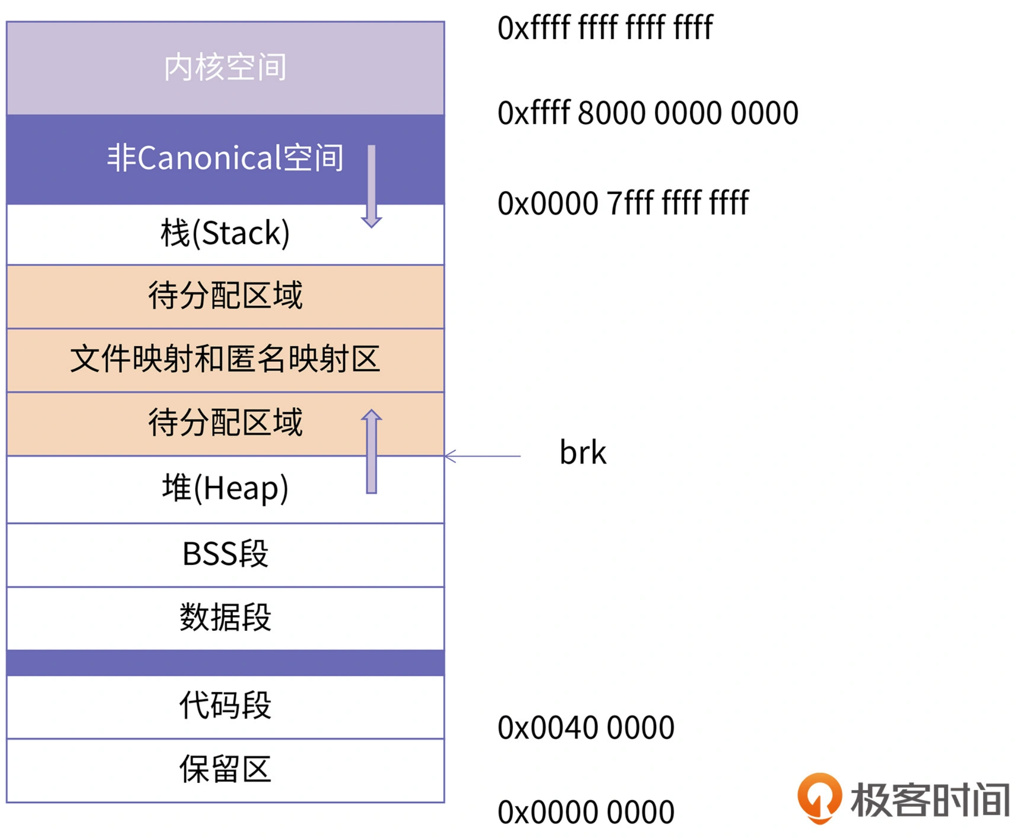
机化；val=1 表示使得 mmap 的基地址、栈地址和 VDSO 的地址随机化；val=2 则是在 1 的基础上增加堆地址的随机化。

到这里，我们对 32 位机器下 Linux 进程的内存布局有了一个清晰的认知。对于 64 位系统而言，它的基本框架与 32 位架构是一致的，但在一些细节上，还是有所不同。

Intel 64 机器上的 Linux 进程内存布局

64 位系统理论的寻址范围是 2^64，也就是 16EB。但是，从目前来看，我们的系统和应用往往用不到这么庞大的地址空间。因此，在目前的 Intel 64 架构里定义了 canonical address 的概念，即在 64 位的模式下，如果地址位 63 到地址的最高有效位被设置为全 1 或全零，那么该地址被认为是 canonical form。目前，Intel 64 处理器往往支持 48 位的虚拟地址，这意味着 canonical address 必须将第 63 位到第 48 位设置为零或一（这取决于第 47 位是零还是一）。

所以，目前的 64 系统下的寻址空间是 2^48，即 256TB。而且根据 canonical address 的划分，地址空间天然地被分割成两个区间，分别是 0x0 - 0x00007fffffffffff 和 0xffff800000000000 - 0xffffffffffffffff。这样就直接将低 128T 的空间划分为用户空间，高 128T 划分为内核空间。下面这张图展示了 Intel 64 机器上的 Linux 进程内存布局：



从图中你可以看到，在用户空间和内核空间之间有一个巨大的内存空洞。这块空间之所以用更深颜色来区分，是因为这块空间的不可访问是由 CPU 来保证的（这里的地址都不满足 Intel 64 的 Canonical form）。

对于 64 位的程序，你在查看 /proc/pid/maps 的过程中，会发现代码段跟数据段的中间还有一段不可以读写的保护段，它的作用也是防止程序在读写数据段的时候越界访问到代码段，这个保护段可以让越界访问行为直接崩溃，防止它继续往下运行。

在所有的内存区域中，程序员打交道最多、接触最广泛的就是堆空间。所以，我们接下来重点关注操作系统所提供的，用于管理堆的系统调用是怎样的。这里我会先给你讲如何通过系统调用申请堆空间，关于堆空间更精细的管理，我们将在第 9 节课介绍。

申请堆空间

其实，不管是 32 位系统还是 64 位系统，内核都会维护一个变量 brk，指向堆的顶部，所以，**brk 的位置实际上就决定了堆的大小**。Linux 系统为我们提供了两个重要的系统调用来修改堆的大小，分别是 sbrk 和 mmap。接下来，我们来学习这两个系统调用是如何使用的。我们先来看 sbrk。

sbrk

sbrk 函数的头文件和原型定义如下：

复制代码

```
1 #include <unistd.h>
2
3 void* sbrk(intptr_t incr);
```

sbrk 通过给内核的 brk 变量增加 incr，来改变堆的大小，incr 可以为负数。当 incr 为正数时，堆增大，当 incr 为负数时，堆减小。如果 sbrk 函数执行成功，那返回值就是 brk 的旧值；如果失败，就会返回 -1，同时会把 errno 设置为 ENOMEM。

在实际应用中，我们很少直接使用 sbrk 来申请堆内存，而是使用 C 语言提供的 malloc 函数进行堆内存的分配，然后用 free 进行内存释放。关于 malloc 和 free 的具体实现，我们将在第 8 节课进行详细讲解。这里你要注意的是，malloc 和 free 函数不是系统调用，而是 C 语言的运行时库。Linux 上的主流运行时库是 glibc，其他影响力比较大的运行时库还有 musl 等。C 语言的运行时库多是以动态链接库的方式实现的，关于动态链接库的相关知识，我们会在第 7 节课加以介绍。

在 C 语言的运行时库里，malloc 向程序提供分配一小块内存的功能，当运行时库的内存分配完之后，它会使用 sbrk 方法向操作系统再申请一块大的内存。我们可以将 C 语言的运行时库类比为零售商，它从操作系统那里批发一块比较大的内存，然后再通过零售的方式一点点地提供给程序员使用。

mmap

另一个可以申请堆内存的系统调用是 mmap，它是最重要的内存管理接口。mmap 的头文件和原型如下所示：

复制代码

```
1   #include <unistd.h>
2     #include <sys/mman.h>
3
4 void* mmap(void* start, size_t length, int prot, int flags, int fd, off_t offs
```

我来解释一下上述代码中的各个变量的意义：

- start 代表该区域的起始地址；
- length 代表该区域长度；
- prot 描述了这块新的内存区域的访问权限；
- flags 描述了该区域的类型；
- fd 代表文件描述符；
- offset 代表文件内的偏移值。

mmap 的功能非常强大，根据参数的不同，它可以用于创建共享内存，也可以创建文件映射区域用于提升 IO 效率，还可以用来申请堆内存。决定它的功能的，主要是 prot, flags 和 fd 这三个参数，我们分别来看看。

prot 的值可以是以下四个常量的组合：

- PROT_EXEC，表示这块内存区域有可执行权限，意味着这部分内存可以看成是代码段，它里面存储的往往是 CPU 可以执行的机器码。
- PROT_READ，表示这块内存区域可读。
- PROT_WRITE，表示这块内存区域可写。
- PROT_NONE，表示这块内存区域的页面不能被访问。

而 flags 的值可取的常量比较多，你可以通过 man mmap 查看，这里我只列举一下最重要的四种可取值常量：

- **MAP_SHARED**：创建一个共享映射的区域，多个进程可以通过共享映射的方式，来共享同一个文件。这样一来，一个进程对该文件的修改，其他进程也可以观察到，这就实现了数据的通讯。
- **MAP_PRIVATE**：创建一个私有的映射区域，多个进程可以使用私有映射的方式，来映射同一个文件。但是，当一个进程对文件进行修改时，操作系统就会为它创建一个独立的副本，这样它对文件的修改，其他进程就看不到了，从而达到映射区域私有的目的。

MAP_ANONYMOUS：创建一个匿名映射，也就是没有关联文件。使用这个选项时，fd 参数必须为空。

MAP_FIXED：一般来说，addr 参数只是建议操作系统尽量以 addr 为起始地址进行内存映射，但如果操作系统判断 addr 作为起始地址不能满足长度或者权限要求时，就会另外再找其他适合的区域进行映射。如果 flags 的值取是 MAP_FIXED 的话，就不再把 addr 看成是建议了，而是将其视为强制要求。如果不能成功映射，就会返回空指针。

通常，我们使用私有匿名映射来进行堆内存的分配，具体的原理我们会在第 9 节课详细分析。

我们再来看参数 fd。当参数 fd 不为 0 时，mmap 映射的内存区域将会和文件关联，如果 fd 为 0，就没有对应的相关文件，此时就是匿名映射，flags 的取值必须为 MAP_ANONYMOUS。

明白了 mmap 及其各参数的含义后，你肯定想知道什么场景下才会使用 mmap，我们又该怎么使用它。

mmap 的其他应用场景

mmap 这个系统调用的能力非常强大，我们在后面还会经常遇到它。在这节课里，我们先来了解一下它最常见的用法。

根据映射的类型，mmap 有四种最常用的组合：

	私有映射	共享映射
匿名映射	私有匿名映射，常用于分配内存	共享匿名映射，常用于父子进程间共享内存，因为只有父子进程之间才能对同一个mmap的返回值进行访问
文件映射	私有文件映射，常用于加载动态库。如果只是读和执行，那么物理内存中只有一份副本。如果动态库中存在全局变量，则数据段就会被复制一份，变成每个进程独有的，而这正是我们所期望的	共享文件映射，可以用于多个进程之间的共享内存。共享内存是通过文件名建立起联系的

其中，私有匿名映射常用于分配内存，也就是我们上文讲的申请堆内存，具体原理我们会在第 9 节课讲解。而私有文件映射常用于加载动态库，它的原理我们会在第 7 节课和第 8 节课进行分析。

这里我们重点看看共享匿名映射。我们通过一个例子，来了解一下 mmap 是如何用于父子进程之间的通信的，其他的例子我会在后面的章节陆续给你介绍。它的用法示例代码如下：

```
#include <sys/mman.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid;

    char* shm = (char*)mmap(0, 4096, PROT_READ | PROT_WRITE,
        MAP_SHARED | MAP_ANONYMOUS, -1, 0);

    if (!(pid = fork())){
        sleep(1);
        printf("child got a message: %s\n", shm);
        sprintf(shm, "%s", "hello, father.");
        exit(0);
    }

```

```
19      sprintf(shm, "%s", "hello, my child");
20      sleep(2);
21      printf("parent got a message: %s\n", shm);
22
23      return 0;
24  }
```

在这个过程中，我们先是用 mmap 方法创建了一块共享内存区域，命名为 shm（第 9 行代码），接着，又通过 fork 这个系统调用创建了子进程。从第 13 行到第 16 行代码是子进程的执行逻辑，具体来讲，子进程休眠一秒后，从 shm 中取出一行字符并打印出来，然后又向共享内存中写入了一行消息（第 15 行）。

在子进程的执行逻辑之后，是父进程的执行逻辑（第 19 行以后）：父进程先写入一行消息，然后休眠两秒，等待子进程完成读取消息和发消息的过程并退出后，父进程再从共享内存中取出子进程发过来的消息。

这就是共享匿名映射在父子进程间通信的运用。我们使用 gcc 编译运行上面的例子，可以得到这样的结果：

复制代码

```
1 $ gcc -o mm mmap_shm.c
2 $ ./mm
3 child got a message: hello, my child
4 parent got a message: hello, father.
```

我想请你结合我刚才的讲解，来分析一下这个程序运行的结果，这样你就理解的更透彻了。

关于共享匿名映射，我们就讲到这里，至于 mmap 的另一个组合共享文件映射。它的作用其实和共享匿名映射相似，也可以用于进程间通讯。不同的是，共享文件映射是通过文件名来创建共享内存区域的，这就让没有父子关系的进程，也可以通过相同的文件创建共享内存区域，从而可以使用共享内存进行进程间通讯。更具体的原理分析我放在了第 10 章。

课程小结

好，这节课我们就讲到这里，现在我们来总结一下。

在这节课中，我们从抽象到具体逐步了解了程序运行时的内存布局模型。我们了解到，**一个进程的内存可以分为内核区域和用户区域**。内核区域是由操作系统内核维护的，我们通常并不关心这一块内存是如何使用的。

程序员最关心的是用户空间，用户空间大致可以分为栈、堆、bss 段、数据段和代码段：

- **代码段**保存的是程序的机器指令，这一段区域的内存往往是可读可执行，但不可写；
- **数据段**保存的是程序的静态变量和全局变量；
- **bss 段**用于无初值的变量区域；
- **堆**是程序员可以自由申请的空间，当我们在写程序时要保存数据，优先会选择堆；
- **栈**是函数执行时的活跃记录，这将是我们下一节课要重点分析的内容。

这 5 个内存区域通常是由高地址向低地址顺序排列的。但这并不是绝对的，以后我们会看到各种反例，比如代码段的位置完全可以比堆的位置还要高。

接着，我们以 Linux 为例，分别研究了 IA-32 架构和 Intel64 架构上的内存布局。在这两种情况下，各个段都是按照上述功能进行划分的，区别在于 64 架构中地址空间更大，而且内核空间和用户空间是不连续的。

此外，我们还初步学习了两个用于堆管理的系统调用 sbrk 和 mmap。其中，mmap 的用法非常复杂，根据调用时传的参数，它有 4 种常见的用法，分别是私有匿名映射、私有文件映射、共享匿名映射和共享文件映射。其中，共享匿名映射是我们这节课的重点，它可以用于父子进程之间的通讯。关于 mmap 的其他功能，我们会在后面的课程逐渐展开。

在接下来的课程中，我会给你详细介绍内存布局中的堆跟栈，这两块也是我们开发人员最常打交道的内存区域，让你对程序运行时的环境和内存状态有一个更深入的理解。

思考题

在这节课的最后，我给你留一道思考题。

一块内存区域的权限一般包括可读，可写，可执行三类，请你思考一下，代码段应该被授予怎么样的权限呢？数据段和堆又该被授予怎样的权限呢？欢迎你在留言区和我交流你的

想法，我在留言区等你。欢迎把这节课分享给更多对计算机内存感兴趣的朋友。我是海纳，我们下一讲再见！

吊打面试官

- 谈谈你在哪些场景下使用过mmap?

mmap最常见的三种用途分别是使用私有匿名映射在堆上分配可用内存，私有文件映射用于加载动态链接库，以及使用共享映射来创建共享内存，以达到进程间通信的目的。

由于实现上的限制，我们经常使用匿名共享映射来实现父子进程间通信，使用文件共享映射来实现多进程之间的通信。

阿里高频面试真题

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 02 | 聊聊X86体系架构中的实模式和保护模式

精选留言 (4)

写留言

慢动作
2021-10-29

从执行视角那张图，代码段是可读可执行，数据段是可读可写，把链接器视角的.bss对应为数据段了，后面的图又把bss单独列出来，这是为什么？bss存了未初始化的数据，只有

大小，那初始化后数据放哪里？

大豆

2021-10-29

1、通过mmap来创建一块区域，那么这块区域是从brk的位置开始吗？还是找到一块满足大小的空闲区域即可。
2、像Java、dart这类高级语言，其虚拟机中的堆、栈等区域的内存分配都是在进程的堆中吧。
3、像字节码这些内容应该是加载在进程的堆中吧。

展开

送过快递的码农

2021-10-29

我说下我的猜想，代码段，可读可执行，数据段：可读可写。至于堆：这个我猜想所在，如果单从我们写一个Hello World程序的时候，堆看起来有读写权限就行。但是，由于我们有实时编译技术，就从我知道的Java来说，有jit。虽然我从来没接触过这个大神，但是它本质上无非就是把一个动态字符穿实时编译成一个可执行字节码。由于通常字符串是在堆中的，所以编译成的可执行字节码是存在堆里面的。所以我猜想，堆里面肯定是有执行...

展开

牧野

2021-10-29

代码段：可执行。数据段：可读。堆：可读可写

展开

作者回复: 正确了一半，但不完全。
代码段只可执行吗？数据段里是全局变量哦，只可读吗？再想想？