关于分段和分页的思考

1. 分段的好处是
2. 利用段寄存器可以使地址进行扩充
3. 使得具有相关性质的汇编代码可以集中管理，如code可读，可执行，有些数据段是只读的
4. 分段使得逻辑地址可以转换为线性地址，也就是相关的代码的上下文是在一起的

2.分页的好处是

1．分页使得线性地址可以转换为物理地址，也就说可以根据程序的需要动态的加载适当的页到内存中，使得程序脱离真实的物理地址，使得程序的无线扩展成为可能。使得物理地址由操作系统进行管理。这样可以避免由于人工地址使用不当造成的错误

值得注意的是

分段和分页都有 段表和页表，这样引入了虚拟内存的概念

malloc 背后的系统知识

原创 2017-04-01 伯乐在线/TKL [Linux爱好者](http://mp.weixin.qq.com/s/sjfkYyvqmsgYUN5EOxU3vQ##)

(点击上方蓝字，快速关注我们)

来源：伯乐在线 - legendtkl

[如有好文章投稿，请点击 → 这里了解详情](http://mp.weixin.qq.com/s?__biz=MzAxODI5ODMwOA==&mid=418672320&idx=1&sn=02c4f68fd3147d29b401235a79eacc4f&scene=21#wechat_redirect)

面试的时候经常会被问到 malloc 的实现。从操作系统层面来说，malloc 确实是考察面试者对操作系统底层的存储管理理解的一个很好的方式，涉及到虚拟内存、分页/分段等。下面逐个细说。

**1. 虚拟内存**

首先需要知道的是程序运行起来的话需要被加载的物理内存中，具体到计算机硬件就是内存条。操作系统启动的时候先把自己加载到物理内存的固定位置（一般为底部），物理内存的其他位置就用来运行用户程序。程序就是一堆指令，程序运行可以简单抽象为把指令加载到内存中，然后 CPU 将指令从内存载入执行。

**1. 为什么需要虚拟内存？**

CPU 对内存的寻址最简单的方式就是直接使用物理内存地址，这种方式一般叫做物理寻址。早期的 PC 使用物理寻址，而且像数字信号处理器、嵌入式微控制器也使用物理寻址。物理寻址的好处是简单，坏处也有很多，比如：

不安全：操作系统的地址直接暴露给用户程序，用户程序可以破坏操作系统。这种解决方案是采用特殊的硬件保护。

同时运行多个程序比较困难：多个用户程序如果都直接引用物理地址，很容易互相干扰。那么是不是可以通过不断交换物理内存和磁盘来保证物理内存某一时间自由一个程序在运行呢？当时是可以的，但是这引入很多不必要和复杂的工作。

用户程序大小受限：受制于物理内存大小。我们现在的错觉是应用程序大小都小于物理内存，这主要是因为现在 PC 的物理内存都比较大。实际上只有 1G 物理内存的 PC 是可以运行 2G 的应用程序的。

综合上面各种缺点，虚拟内存出现了。

**2. 虚拟内存概览**

虚拟内存的基本思想是：每个程序拥有独立的地址空间（也就是虚拟内存地址，或者称作虚拟地址），互不干扰。地址空间被分割成多个块，每一块称作一页（page），每一页有连续的地址范围。虚拟地址的页被映射到物理内存（通过 MMU，Memory Management Unit），但是并不是所有的页都必须在内存中才能运行程序。当程序引用到一部分在物理内存中的地址空间时，由硬件立刻执行必要的映射。当程序引用到一部分不在物理内存中的地址空间时，由操作系统负责将确实的部分装入物理内存。虚拟地址寻址（也叫做虚拟寻址）的示意图如下。

**3.虚拟内存实现**

1.虚拟内存大小

一般是和 CPU 字长相关，比如 32 位对应的虚拟地址空间大小为：0 ~ 2^31。

2. MMU

CPU 将虚拟地址发送给 MMU，然后 MMU 将虚拟地址翻译成物理地址，再寻址物理内存。那么虚拟地址和物理地址具体是怎么映射的呢？完成映射还需要另一个重要的数据结构的参与：页表（page table）。页表完成虚拟地址和物理地址的映射，MMU 每次翻译的时候都需要读取页表。页表的一种简单表示如下。

这里页大小为 p 位。虚拟内存的页和物理内存的页大小一样。虚拟地址的高 n-p 位，又叫做虚拟页号（Virtual Page Number, VPN），用来索引物理页号（Physical Page Number，PPN），最后将 PPN 和低 p 位组合在一起就得到了物理地址。

3. 页表的两个问题

前面说到用 VPN 来做页表索引，也就是说页表的大小为虚拟地址位数 / 页的大小。比如 32 位机器，页大小一般为 4K ，则页表项有 2^32 / 2^12 = 2^20 条目。如果机器字长 64 位，页表项就更多了。那么怎么解决呢？一般有两种方法：

1. 倒排页表。物理页号做索引，映射到多个虚拟地址。通过虚拟地址查找的时候就需要通过虚拟地址的中间几位来做索引了。
2. 多级页表。以两级页表为例。一级页表中的每个 PTE （page table entry）映射虚拟地址空间的一个 4MB 的片，每一片由1024 个连续的页面组成。一级 PTE 指向二级页表的基址。这样 32 位地址空间使用 1024 个一级 PTE 就可以表示。需要的二级页表总条目还是 2^32 / 2^12 = 2^20 个。这里的关键在于如果一级 PTE i 中的页面都未被分配，一级 PTE 就为空。多级页面的一个简单示意图如下。

多级页表减少内存占用的关键在于：

1. 如果一级页表中的一个 PTE 为空，那么相应的二级页表就根本不会存在。这是一种巨大的潜在节约。
2. 只有一级页表才需要常驻内存。虚拟内存系统可以在需要时创建、页面调入或者调出二级页表，从而减轻内存的压力。

第二个问题是页表是在内存中，而 MMU 位于 CPU 芯片中，这样每次地址翻译可能都需要先访问一次内存中的页表（CPU L1,L2,L3 Cache Miss 的时候访问内存），效率非常低下。对应的解决方案是引入页表的高速缓存：TLB（Translation Lookaside Buffer）。加入 TLB，整个虚拟地址翻译的过程如下两图所示。

关于虚拟内存还有一些内容比如 page fault 处理，这里就不再赘述了。

**2. 分段**

**1. 分段概述**

前面介绍了分页内存管理，可以说通过多级页表，TLB 等，分页内存管理方法已经相当不错了。那么分页有什么缺点呢？

1. 共享困难：通过共享页面来实现共享当然是可以的。这里的问题在于我们要保证页面上只包含可以共享的内容并不是一件容易的事儿，因为进程空间是直接映射到页面上的。这样一个页面上很可能包含不能共享的内容（比如既包含代码又包含数据，代码可以共享，而数据不能共享）。早期的 PDP-11 实现的一种解决方法是为指令和数据设置分离的地址空间，分别称为 I 空间和 D 空间（其实这已经和分段很像了）。
2. 程序地址空间受限于虚拟地址：我们将程序全部映射到一个统一的虚拟地址的问题在于不好扩张。不如我们程序的地址按先代码放在一起，然后把数据放在一起，然后再放 XXX，这样其中某一部分的空间扩张起来都会影响到相邻的空间，非常不方便。

上面的问题一个比较直观的解决方法是提供多个独立的地址空间，也就是段（segment）。每个段的长度视具体的段不同而不同，而且是可以在运行期动态改变的。因为每个段都构成了一个独立的地址空间，所以它们可以独立的增长或者减小而不会影响到其他的段。如果一个段比较大，把它整个保存到内存中可能很不方便甚至是不可能的，因此可以对段采用分页管理，只有那些真正需要的页面才会被调入内存。

采用分段和分页结合的方式管理内存，一个地址由两个部分组成：段和段内地址。段内地址又进一步分为页号和页偏移。在进行内存访问时，过程如下：

1. 根据段号找到段描述符（存放段基址）。
2. 检查该段的页表是否在内存中。如果在，则找到它的位置，如果不在，则产生段错误。
3. 检查所请求的虚拟页面的页表项，如果该页面不在内存中则产生缺页中断，如果在内存中就从页表项中取出这个页面在内存中的起始地址。
4. 将页面起始地址和偏移量进行拼接得到物理地址，然后完成读写。

**2. 进程的段**

每个 Linux 程序都有一个运行时内存映像，也就是各个段的布局，简单如下图所示。

注意上图只是一个相对位置图，实际上这些段并不是相邻的。主要的段包括只读代码段、读写段、运行时堆、用户栈。在分配栈、堆段运行时地址的时候，链接器会使用空间地址空间布局随机化（ASLR），但是相对位置不会变。上图中 .data 等是对应进程中的不同数据的 section ，或者叫做节。简介如下。

* .text: 已编译程序的机器代码。
* .rodata: 只读数据。
* .data: 已初始化的全局和静态变量。局部变量保存在栈上。
* .bss: 未初始化的全局和静态变量，以及所有被初始化为 0 的全局或者静态变量。在目标文件中这个节不占据实际的空间，它仅仅是一个占位符。

**3. malloc 实现**

**1. 堆内存管理**

我们常说的 malloc 函数是 glibc 提供的库函数。glibc 的内存管理使用的方法是 ptmalloc，除此之后还有很多其他内存管理方案，比如 tcmalloc （golang 使用的就是 tcmalloc）。

ptmalloc 对于申请内存小于 128KB 时，分配是在堆段，使用系统调用 brk() 或者 sbrk()。如果大于 128 KB 的话，分配在映射区，使用系统调用 mmap()。

**2. brk, sbrk**

在堆段申请的话，使用系统调用 brk 或者 sbrk。

int brk(const void \*addr);

void \*sbrk(intptr\_t incr);

brk 将 brk 指针放置到指定地址处，成功返回 0，否则返回 -1。sbrk 将 brk 指针向后移动指定字节，返回依赖于系统实现，或者返回移动前的 brk 位置，或者返回移动后的 brk 位置。下面使用 sbrk 实现一个巨简单的 malloc。

void \*malloc(size\_t size) {

    void \*p = sbrk(0);

    void \*request = sbrk(size);

    if (request == (void\*) -1) {

        return NULL; // sbrk failed.

    } else {

        assert(p == request); // Not thread safe.

        return p;

    }

}

**3. mmap**

linux 系统调用 mmap 将一个文件或者其它对象映射进内存。

#include <sys/mman.h>

void \*mmap(void \*addr, size\_t length, int prot, int flags, int fd, off\_t offset);

mmap 的 flags 可选多种参数，当选择 MAP\_ANONYMOUS 时，不需要传入文件描述符，malloc 使用的就是 MAP\_ANONYMOUS 模式。mmap 申请的内存在操作系统的映射区。比如 32 位系统，映射区从 3G 虚拟地址粗向下生长，但是因为程序的其他段也会占用空间（比如代码段必须以特定的地址开始），所以并不能申请 3G 的大小。

**4. malloc 和物理内存有关系吗？**

可以说没关系，malloc 申请的地址是线性地址，申请的时候并没有进行映射。访问到的时候触发缺页异常，这个时候才会进行物理地址映射。

**5. ptmalloc**

ptmalloc 只是 glibc 使用的内存管理策略，篇幅有限，这里就不细说了。我之前写了一篇 tcmalloc 的介绍，大家可以对比着看。

**4. 参考：**

1. 《深入理解计算机系统》
2. 《现代操作系统》
3. StackOverFlow
4. mmap manpage
5. tcmalloc 介绍
6. a malloc tutorial
7. malloc manpage
8. understanding glibc malloc
9. advanced memory allocation

看完本文有收获？请分享给更多人

**关注「Linux 爱好者」，提升Linux技能**

[阅读原文](http://mp.weixin.qq.com/s/sjfkYyvqmsgYUN5EOxU3vQ##)



微信扫一扫  
关注该公众号

值得注意的是

1. 使用分页的时候，一级页表和二级页表的，页表设计方式是不一样的。
2. 使用二级页表的时候，常驻在内存中的是二级页表中的一级页表。第二级页表在需要的时候动态加载
3. 引入一级页表的作用是 原本程序无法使用4g 32位 内存，但是引入一级页表之后，程序就可以访问4g内存了，原因是当使用物理地址的时候，由于其他程序在占用，如果直接使用4g内存，则会导致内存冲突。

当一级页表的设计 使用 12位表示页表号的时候，

具体的分页设计为

虚拟地址 =页表号（12）+偏移地址（20）

则可以表示的地址方位是 2^20\*2^20=2^40 >>2^32

如果页表号采用 14位

可以表示的地址范围是 2^18\*2^18=2^36>>2^32

如果页表号采用16为呢 则 2^16\*2^16==2^32

页表号不同，可以表示的地址范围不同，但是可以表示的地址范围越大，那么可以使用的基址个数就越少

如 12 4\*1024

14 16\*1024

16 64\*1024

基址个数的作用是操系统中，可以同时运行的程序的个数，所以使用的时候可以综合考虑。

当采用一级页表的时候，使用4g地址的问题已经解决了，因为可以通过硬盘对内存进行扩容，也就是只有使用或者是将要使用的页表才会移入内存，其他的页表则会留在硬盘。这样的好处是更多的程序可以共存了，但是每个独立的程序，他们看来好像是自己独享4g内存一样。

假如 一级页表，每个页表项站4个字节4b，那么当页表号是12位时，一级页表占用的空间是多大呢

12位 可以有 2^12=4\*1024 =4K个页表项

占用的空间是 4b\*4k=16kb ，也就是说如果采用一级页表，那么有多少个程序就有多少个16kb。

为了解决多程序共存，也就是多基址，并且程序需要大内存的问题，这样就引入了多级页表基准。接下来以二级页表进行分析

二级页表的设计

虚拟地址=一级页表号+二级页表号+偏移号

现在 一级页表号 12 二级页表号12 偏移号 8

那么可以表示的基准个数是

2^20 表示一级页表中可以表示二级页表的个数

2^12 表示一个第二级页表中，可以表示基址的数目

2^20 表示一个第二级页表中，可以表示的地址范围

基址个数为 2^20\*2^12=2^32

地址范围 2^20\*2^20=2^40

同一级页表一样，二级页表也可以根据需求动态的改变二级页表的设计

二级页表中基址个数和 地址范围的含义是

当前系统中可以同时存在2^32个程序，每个程序可以使用的地址范围是 2^40 比特位

更不用说三级页表或者更高级的了

假设 每一个页表项站 4个字节 4b，那么使用二级页表进行表示地址的时候。需要的内存占用是多少呢

由于1个一级页表项指向一个二级页表，所以只用1个一级页表项就可以使得 2^12个程序可以共存

那么一个程序需要用的内存是 4b 一级页表项+ 4b 二级页表项=8b<<16kb

并且一级页表项是可以公用了，而且可以使用的地址范围 同16kb的一级页表项相同。

特别注意的是 程序可以使用地址和程序可以使用的地址范围是两回事

可以使用的地址 只与 偏移号有关，默认是只使用一个页表号来表示当前程序的地址，如果程序想使用更多的地址，那么可以给程序多分配几个页表号即可。

可以使用的地址范围是 与 基址位数和偏移号有关。上文已经计算过了。

二级页表相对一级页表最明显的区别是随着程序并行规模的不断扩大，那么二级页表在内存占用方面的优点将会越来越明显。

比如系统中有1025个程序

一级页表 无法使其共存

二级页表 2个一级页表项几个，其中一个第二级页表满员，另一个只有一项占用。