|  |
| --- |
|  |
| 内存管理阅读报告 |
|  |

|  |
| --- |
| **姓名: 余晶 学号：S1748036**  **姓名: 方明 学号：S1748007**  **日期: 2018.03.31** |

[内容一：任务完成情况 3](#_Toc508263543)

[任务完成列表（Y/N） 3](#_Toc508263544)

[具体Exercise的完成情况 3](#_Toc508263545)

[内容二：参考文献 10](#_Toc508263546)

## 引言

内存管理是操作系统一个非常重要的概念。 我们在课程中学到了很多相关的概念。 例如连续内存管理，非连续内存管理， 虚拟内存管理。 更细节一点，我们知道了段式，页式，块表等。 本报告首先回顾一下，内存管理一些重要的基本概念以及在Linux系统中的应用。 然后我们将研究XV6内存管理。最后做一下比较。

## 内存管理与Linux

### 基本概念

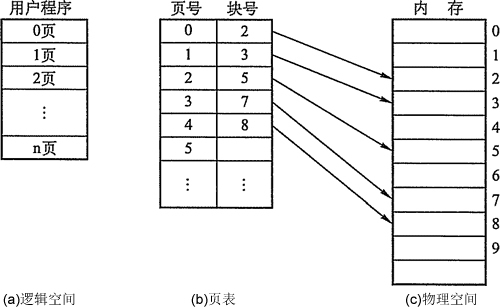
逻辑地址和物理地址：

物理地址用于内存芯片级的单元寻址， 与处理器和CPU连接的地址总线想对应。为了方便的理解，我们可以把它理解为一个从0字节一直到最大空量逐字节的编号的数组。

逻辑地址： 这个概念来源于Intel段式内存管理， 逻辑地址指的是机器语言指令中，用来指定一个操作数或者是一条指令的地址。 一个逻辑地址，是由一个段标识符加上一个段内相对地址的偏移量， 表示为[段标识符：段内偏移量]

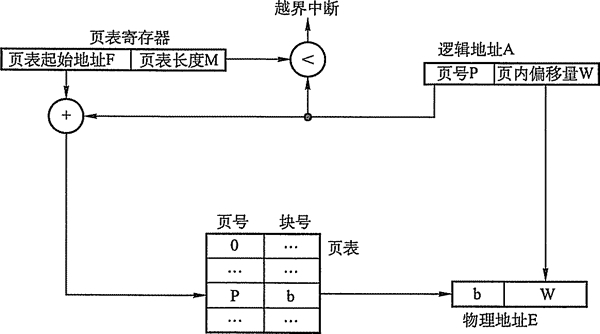
虚拟地址：也叫线性地址， 逻辑地址需要经过分段转化为线性地址， 线性地址经过分页转化为物理地址。

页表和页表项：为了便于在内存中找到进程的每个页面所对应的物理块，系统为每个进程建立一张页表，记录页面在内存中对应的物理块号，页表一般存放在内存中。一个页表是有一组页表项组成。



进程中的块成为页（Page）， 内存中的块称为页框(Page Frame), 外存中的快直接称为块(Block)。

下面我们看看分页存储管理系统中是如何进行地址变换的：



在系统中通常设置一个页表寄存器（PTR）， 存放页表在内存的起始地址F和页表长度M。进程未执行时，页表的始址和长度存放在进程控制块中，当进程执行时，才将页表始址和长度存入页表寄存器。具体的变化过程是：

1. 通过页表起始地址和逻辑地址的页号P可以在页表中找到对应的页表项。
2. 通过该页表项中的块号与逻辑地址的偏移组合，得到了新的物理地址。

段式管理方式按照用户进程中的自然段划分逻辑空间。例如，用户进程由主程序、两个子程序、栈和一段数据组成，于是可以把这个用户进程划分为5个段，每段从0 开始编址，并分配一段连续的地址空间（段内要求连续，段间不要求连续，因此整个作业的地址空间是二维的）。其逻辑地址由段号S与段内偏移量W两部分组成。

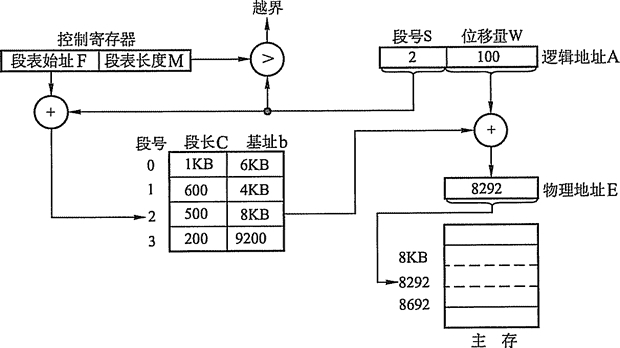
分段系统中的逻辑地址结构：



段表项：

http://c.biancheng.net/cpp/uploads/allimg/140701/1-140F10101563W.png

接着我们看看分段式存储管理是如何进行地址变换的：



与分页类似：

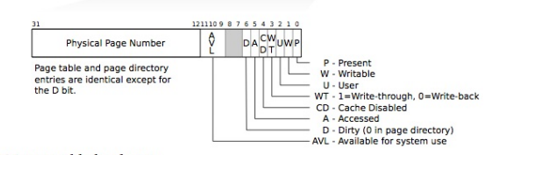
1. 通过对应的段表起始位置F和逻辑地址的段号找到对应的段表项。
2. 物理地址 = 逻辑地址的位移100　 +　段的起始位置8KB

## Linux

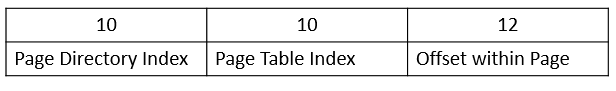
Linux内核使用的是页式内存管理，应用程序给出的内存是虚拟地址， 它需要经过若干级的页表的一级一级的变换，才能变成真正的物理地址。

## XV6内存管理

XV6同样也采用了页式管理。 它使用了一个二级页表。一个页表就包含了2^20条页表条目PTE的数组。每条PTE包含了一个20位的物理页号PPN及一些标志位，常用的标志位PTU\_W(0x002), PTU\_U(0x004)。 页表项的结构为：

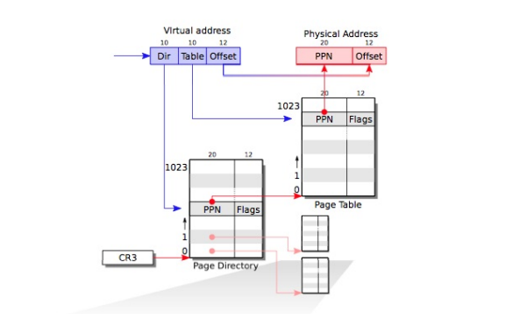


XV6的虚拟地址包含三个部分， 页目录索引（Page Directory Index）， 页表索引（Page Table Index），页内偏移(Offset within Page)，其示意结构为：



如果虚拟地址为va, 那么函数PDX(va) 得到的就是页目录的索引， PTX(va)得到的是页表索引。

有了以上信息我们可以知道XV6从虚拟地址到物理地址的转换了，示意图为：



XV6是一个二级页表， 第一级是页表目录，CR3是一个寄存器，它存放的是当前进程的页表目录地址。地址转换的步骤为：

1. 从寄存器CR3中读取该进程的页表目录地址。
2. 读取虚拟地址的“Dir”部分，然后在该页表目录中找到对应的页表项。该页表项的PPN就是页表的其实地址。
3. 读取虚拟地址的“Table”部分，在上一步找到的页表中找到对应的页表项。
4. 该页表项的PPN和虚拟地址的“Offset”组合起来就是真实的物理地址。

整体上我们已经了解了XV6的内存管理，下面我们深入阅读代码来了解一些细节，如内核是如何来管理物理内存的？虚拟地址空间是什么样？

XV6一个物理内存页块大小为2^12, 也就是4096B。所有空闲的物理内存页通过数据结构kmen管理的，它内面存放了一个用户线程保护的锁和空闲物理地址的首地址freelist。其中 freelist是一个链表，我们可以通过链表的插入，删除操作来完成空闲内存的释放和分配工作。

物理空闲内存的初始化，有两个步骤:

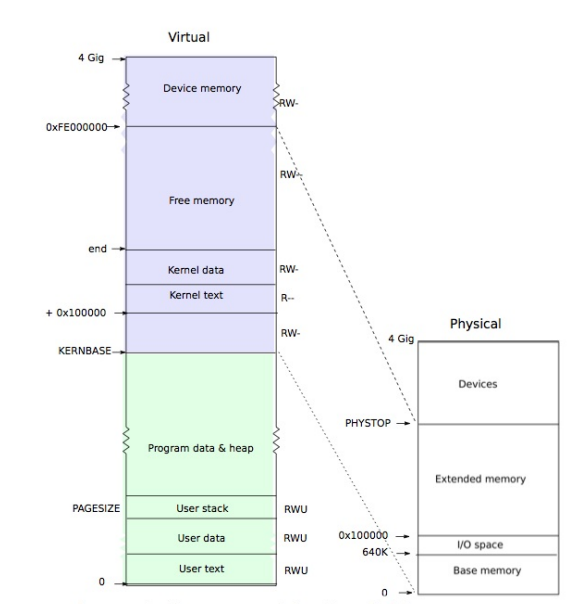
1. Main() 调用kinit1(end, P2V(4\*1024\*1024))， 我们可以知道XV6每页的大小为4k， 这个函数就是分配了1K个物理内存页到XV6内核中。
2. Main()调用kinit2(P2V(4\*1024\*1024), P2V(PHYSTOP))， PHYSTOP 为0xE00000, 有房了2.5K个物理页到内核中。

虚拟地址的初始化：

每个进程都有一个页表，如果CPU没有运行任何进程， 将使用的页表为kpgdir。 内核在系统调用和中断时使用当前进程的页表，其中，页保护位用来防止用户代码使用内核的映射。

进程的用户内存是从0开始，最多能够增长到KERNBASE(0x80000000)，这使得一个进程最多使用2GB的内存。当进程想XV6要更多的物理内存时，XV6首先要找到空闲的物理页， 然后把这些页对应的PTE加入该进程的页表中。

XV6在每个进程的页表中都包含了内核运行所需要的所有映射，而这些映射都出现在KERNBASE之上。它将虚拟地址KERNBASE… KERNBASE+EXTMEM映射到了物理地址的0..EXTMEM（0X100000），作为I/O空间, 其具体的分配图如下：



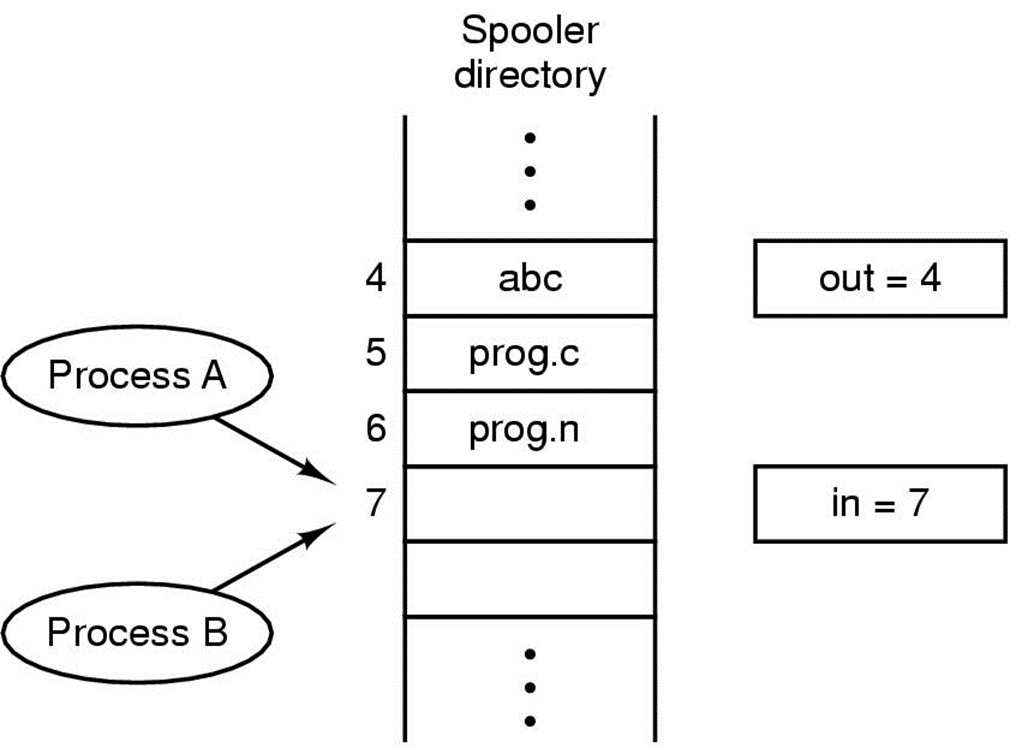
物理内存管理：

Xv6的物理内存是通过kmen来管理的，它内面存放了一个用户线程保护的锁和空闲物理地址的首地址freelist. Freelist是一个链表，我们可以通过链表的插入，删除操作来完成空闲内存的释放和分配工作。

### 具体Exercise的完成情况

1. 什么是临界区？什么是同步和互斥？什么是竞争状态？临界区操作时中断是否应该开启？中断会有什么影响？XV6的锁是如何实现的，有什么操作？xchg是什么指令，该指令有何特性？

* 临界区：系统中某些资源一次只允许一个进程使用，称这样的资源为临界资源或互斥资源或共享变量。各个进程中对某个临界资源或共享变量实施操作的程序片段称为临界区［1］；
* 由于进程间要求使用共享资源（变量、文件等），而这些资源需要排他性使用。进程之间竞争使用这些资源，这一关系称为进程互斥。系统中多个进程中发生的事件存在某种时序关系，需要相互合作，共同完成一项任务。一个进程运行到某一点时，要求另一伙伴进程为它提供消息，在未获得消息之前，该进程进入阻塞态，获得消息后被唤醒进入就绪态，这一关系称为进程同步［1］；
* 竞争状态：两个或多个进程读写某些共享数据，而最后的结果取决于进程运行的精确时序［3］；



* 在单处理器系统中，最简单的方法是使每个进程在刚刚进入临界区的时候立即屏蔽中断，并在离开之前再打开中断。屏蔽中断后，时钟中断也被屏蔽。CPU只有发生时钟中断或者其它中断才会进行进程切换，这样在屏蔽后CPU不会被切换到其它进程。但是这种方案并不好，一个进程如果屏蔽中断后，若不打开中断，造成整个系统因此而终止。如果是多处理器，屏蔽中断只对单CPU有效，其它CPU仍然继续运行访问临界资源。其次，对于内核来说，中断屏蔽是很有效的手段。但是当就绪进程队列之类的数据状态不一致时发生中断，将会导致竞争条件。所以说屏蔽中断对操作系统而言是有效，但对于用户进程则不是一种合适的互斥机制。如果临界区过大，造成长时间在临界区，限制了CPU的并发能力；
* 通过阅读spinlock的代码知，XV6使用结构体spinlock来代表锁。变量locked代表锁的状态，为1表示上锁，为0表示未上锁。Name表示锁的名字。spinlock还附带有调试信息，比如锁的名字，当前占有锁的CPU和调用栈。

. XV6使用acquire和release操作来代表获得锁和解除锁，使用时它们必须成对使用；

. pushcli 通过调用cli关闭外部中断并递增关闭中断的次数。原因是如果上了两个锁，那么只有当两个锁都被释放后中断才会开启。相应的acquire 一定要在可能获得锁的 xchg 之前调用 pushcli。如果颠倒，则会在几个时钟周期里，中断仍被允许，也可获得锁。在这个时候如果发生了中断，会造成系统出现死锁。同样在 release 使用时也需要在释放锁的 xchg 之后再调用 popcli；

. holding 用于检查是否当前CPU持有锁；

* xchg是两个寄存器，寄存器和[内存变量](https://baike.baidu.com/item/%E5%86%85%E5%AD%98%E5%8F%98%E9%87%8F)之间内容的交换指令。通过循环xchg：如果返回值为1，代表占用锁，继续循环等待xchg；如果xchg返回为0，代表目前没有人占用锁，将锁的数据结构locked置为1，然后跳出循环。这样，在访问一个可能出现竞争条件的数据结构时，必须提前获得这个数据的锁，如果暂时无法获得锁，则不断循环测试。在处理完数据后再解除锁的占用，以便其它进程能重新占用锁；

1. 基于XV6的spinlock，请给出实现信号量、读写锁、信号机制的设计方案。

* 下面用伪码实现了一下读者写者问题。用锁来实现读者写者问题。运行两个写者线程，两个读者线程。通过加速保护缓冲区，使得同一时刻同一段缓冲区只有一个写者可以访问。读取的时候，要保证该缓冲区不能被写者访问。定义一个计数器，第一读者进入时，给缓冲区加锁，不允许写；最后一个读者读完，解锁；

spinlock rlock;

spinlock wlock;

int rCount = 0;

initlock(&rlock, "Reader");

initlock(&wlock, "Writer");

void Read()

{

acquire(&rlock);

rCount++;

if(1 == rCount) { acquire(wlock); }

release(&rlock);

...

Read something from buffer;

...

acquire(&rlock);

rCount--;

if(0 == rCount) { release(wlock); }

release(&rlock);

}

void Write()

{

acquire(&wlock);

acquire(&rlock);

...

Write something to buffer;

...

release(&rlock);

release(&wlock);

}

1. XV6初始化之后到执行main.c时，内存布局是怎样的（其中已有哪些内容）？

* 在计算机启动的时候，首先执行的代码是主板上的BIOS。BIOS所做的主要是一些硬件自检的工作。在这些工作做完之后，它会从启动盘里读取第一扇区的512字节数据到内存中，即 bootloader。在导入完之后，BIOS就会把CPU的控制权给bootloader。BIOS会把bootloader导入到地址0x7c00开始的地方，然后把PC指针设成此地址，以完成跳转。XV6 的bootloader的汇编部分代码bootasm.S，首先关中断，然后清理寄存器，ds、es、ss实际上存放的就是各个段的基地址，其中ds代表数据段，es代表扩展段，ss代表堆栈段，这里把寄存器清0；

seta20.1:

inb $0x64,%al # Wait for not busy

testb $0x2,%al

jnz seta20.1

movb $0xd1,%al # 0xd1 -> port 0x64

outb %al,$0x64

seta20.2:

inb $0x64,%al # Wait for not busy

testb $0x2,%al

jnz seta20.2

movb $0xdf,%al # 0xdf -> port 0x60

outb %al,$0x60

* 然后我们离开实模式，进入保护模式。在我们转换到保护模式之前，我们必须保证已经建好了段描述符表，lgdt命令把段描述符表的基址和限界存到GDTR这个寄存其中，硬件就可以找到这个段描述表了。gdtdesc实际上就是分配了六个字节，前两个字节存放的是gdt的大小减一，后四个字节是gdt表的地址。lgdt负责把这段空间的基地址传到GDTR寄存器中，段描述符表就建立好了。建立好段描述表之后，可以切换到保护模式了，通过置位CR0的PE位打开保护模式。CR0是个控制寄存器。然后我们跳到start32处执行，设置各个段的索引，可以看到ds,es,ss都索引到了数据段，fs和gs段值为0。最后设置了一个栈顶指针，然后跳到C代码执行，bootloader的汇编部分至此结束；
* bootmain(void)函数，主要是从磁盘读出kernel，然后跳到入口点开始执行kernel的代码。kernel是按照elf的格式存到硬盘中的，内核读出来是放到地址空间0x10000处。bootmain的最后一个操作就是跳到内核执行了，entry的地址这里是0x10000c,下面开始的就是真正的内核代码了（entry.S）；
* bootloader把内核从硬盘导入到0x100000处，由于640kb到0x100000开始的地方是被用于IO device的映射，所以为了保持内核代码的连续性，就从1MB的内存区域开始存放。在XV6里面，内核对应的虚拟地址实际上是从0x800000处开始的，那么为什么不干脆把内核导入到物理地址的0x800000处呢？这个原因主要是有的小型PC是没有这么高的地址的，所以放在0x100000处显然是个更好的选择。在 entry.S 中，首先我们定义一个全局变量\_start来保存entry的地址，由于我们还没有开始虚拟地址，所以这里保存的还是entry的实际物理地址。 在entry中，我们首先打开CR4的PSE位，即页大小扩展位，如果PSE等于0，每页的大小只能是4KB，如果PSE等于1，每页的容量可以达到4MB。然后就要打开分页模式了，在这之前，我们必须要确保页目录表的地址已经存放到cr3中了。cr3也是一个控制寄存器，里面存放了全局页目录表的地址。entrydir中只定义了两项，分别把虚拟地址0~4MB和KERNBASE(0x800000) ~ KERNBASE + 4MB的空间映射到了0~4MB的物理地址，映射0~4MB是因为我们在entry之前可能还会用到这段区域，0x800000则是分页之后内核认为的虚拟地址所在地。然后打开分页模式。打开分页模式主要是置位CR0的PG位。在分页之后，设置栈指针，KSTACKSIZE的定义是4096,即4K.这意味着用4KB开始向下的这一段区域当做内核的栈。最后，我们跳到main函数执行，在main函数中初始化一些变量并且开始创立我们的第一个进程；
* bootasm.S bootmain.c引导代码放在第0扇区，链接到0x7c00。BIOS自检后会跳转到这个地址执行，并将控制权交给操作系统；initcode: initcode.S 第一个用户进程，链接到0。BSP (bootstrap processor) 引导及初始化完成后会回到用户态，开始执行这个用户进程；initcode 编译后生成ELF文件，放在第1扇区。SP初始化的过程中会将其加载内核中的640K，之后解析ELF格式将代码段、数据段等链接到相应的地址，即initcode.S以及kernel就可以链接到正确的地址了；main.c主要设置了BSP的数据以及一些控制器的初始化工作： ksegment() 设置CPU结构体数据。picinit() 初始化主从中断控制器。 kinit() 释放了从end-16M的内存空间，构成了一个空闲内存链表，每个节点对应1页 (4K)，头结点为kmem.freelist。接下来跳转到mainc继续执行。整个内存布局示意图如下；



1. XV6的动态内存管理是如何完成的？有一个kmem（链表），用于管理可分配的物理内存页。（vend=0x00400000，也就是可分配的内存页最大为4Mb）
2. XV6的虚拟内存是如何初始化的？画出XV6的虚拟内存布局图，请说出每一部分对应的内容是什么。见memlayout.h和vm.c的kmap上的注释。
3. 关于XV6的内存页式管理。发生中断时，用哪个页表？一个内页是多大？页目录有多少项？页表有多少项？最大支持多大的内存？画出从虚拟地址到物理地址的转换图。在XV6中，是如何将虚拟地址与物理地址映射的（调用了哪些函数实现了哪些功能）？

## 内容二：参考文献

1. 陈向群. 同步互斥机制［Z］.
2. 陈向群. 存储管理方案课件［Z］.
3. Andrew S.Tanenbaum.现代操作系统［M］. 北京：机械工业出版社，2017．