对于 xv6 操作系统的深度解析

---结合《操作系统原型---xv6 分析与实验》

2022 年 08 月 04 号 华中农业大学 刘浩

引言

列夫·托尔斯泰说过：“正确的道路是这样的：汲取你的前辈所做的一切，然后再继续往前走”，如无根浮萍，终无安所。

有疑问可发送邮箱 2226958871@qq.com，希望对大家有帮助。

xv6 概述

xv6 代码主要分为启动扇区代码和内核代码。

启动扇区代码的任务是加载某些内核启动前所准备的东西，比如硬件中断向量和设置堆栈。启动扇区代码即 bootblock 文件（一般是放在 BIOS 中，BIOS 一般放在主板上的 ROM 中），其是由 bootblock.S 和 bootmain.c 两个文件用 kernel.ld 脚本链接的（类似于 Windows 的 BootLoader 和 Linux 的 GRUB），从链接脚本可见，“.=0x80100000”可知，内核代码的地址布局是从 0x800000 开始的，但是 14 行指出其装载地址是 0x100000，所以 PA=0x00100000，运行地址 0x80100000。其中 ENTRY(_start)，即将_start 作为程序入口地址。（链接脚本的语法规则可见附录）

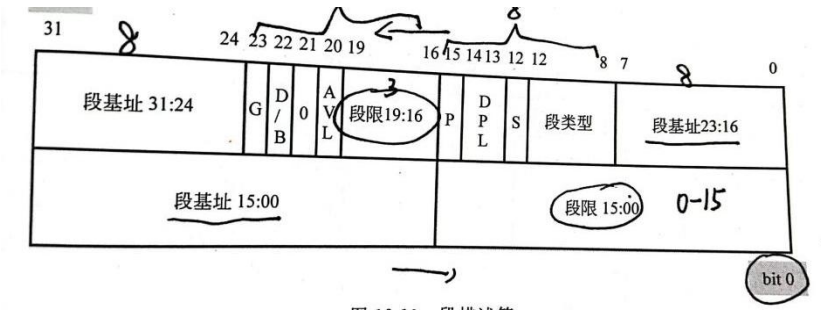
在 BIOS 加载的过程中会把中断向量表放在内存中，然后进行加电自检等操作，然后 BIOS 默认把 bootblock 加载到 0x7c00，然后把控制权交给 bootblock。

可知在 0x7c00 处是 start 处，因此从此处开始执行，首先关闭中断，然后把三个段寄存器清零，开启 A20 根地址线。

为何要开启 A20 地址线？

A0~A19 能寻址 1MB 的空间（0x00000~0xFFFFF），但是 8086/8088 是 16 位地址模式，为了让其能访问 1M 内存，Intel采用了分段的模式：16 位基址+16 位偏移=20 位地址但是此时引起了新的问题，通过上述分段模式，能够表示的最大内存为 10FFEFh，比 1M 多出来的部分被称为高端内存区 HMA，但是 8086/8088 只有 20 根地址线，如果要访问 100000h~10FFEFh 的区域，则必须有第 21 根地址线，所以当程序员给出超过 1M 的内存时，系统并不认为其是访问越界，而是自动从 0 开始重新计算，也就是求模（warp-around 技术），在 Intel80286 处理器中，系统的地址总线发展位 24 根，这样能访问的内存达到 16M，在实模式下能够和 8086/8088 兼容，IBM 使用键盘控制器上剩余的一根输出线来管理第 21 根地址线，所以称为 A20Gate，让其可以访问高于 1MB 的物理内存。现在许多新型 PC 存在着一种通过芯片直接控制 A20Gate 的 BIOS 功能，比键盘控制器的 A20Gate。

然后为了进行保护模式，设置好 GDT 段描述符表，将其地址装入 GDTR 寄存器，初始化有三项，第一项为 NULL，第二项为 bootblock 代码段，第三项为 bookblock 数据段，起点均为 0x0。之后设置好 CR0 寄存器，进入保护模式。



以下为保护模式，保护模式的程序地址是用逻辑地址来表示的，逻辑地址通常保存为“段：偏移”的形式。此处将所有段的起点都设为为 0，实际上弱化了 x86 的段管理功能。

长跳转指令“ljmp $(SEG_KCODE)<<3”跳转到 start32 的标号所在的代码处（此处需要左移 3 位的原因是段选择子的低 3 位索引不是索引编号，因此为 GDT 中索引为 1 的位置即 bootblock 的代码段，从 0x0 开始，偏移为 start32 的位置为标号为 start32 的位置）。从此处开始是 32 位代码。首先给 ds、ss 和 es 三个段选择子都索引到了数据段，fs 和 gs 则执行无效段（GDT 的编号为 0 的段），并将 esp 设置为 start，即 0x7c00 之前。然后跳转到 C 代码执行 bootmain()。

在知晓 ELF 文件格式的情况下，继续探究 bootmain 函数。先是通过 readseg()函数将磁盘最开头（跳过启动扇区 bootblock）的 4KB 字节读入到内存 0x100000 地址处（读取模式为 LBA 模式。

LBA 模式访问

早期的硬盘 IDE 的读取方式比较复杂，需要分别指定 CHS（柱面/磁头/扇区）；后来的磁盘都支持 LBA（logical block addressing）模式，所有扇区都统一编号，只需指出扇区号即可完成访问。为何要换成 LBA 模式？我个人认为有两点：第一点，是为了访问磁盘块的便利性；由磁盘自定义扇区位置和编号（CHS 模式是连续访问）可以更高效的访问访问磁盘块（在同一磁道中有多个磁头的时候，因为 CPU 在同一时间只能接受一个磁头传输的信息，而一个磁头还在传输的过程中，此时另一个磁头必须多转一圈才能读取邻近扇区的值，因此可以相隔一个扇区进行读取，这样就不用多转一圈，极大的提高了读取和传输效率）。

为何不放在 0x0 处，是因为从 640kb 到 0x100000 的地方用于 I/O 设备的映射），最后获取装载入口地址 0x10000c，即标号为 start，跳转执行。在 0x10000c 处为 _start=V2P_WO(entry)，而 entry 的地址为 0x8010000c，所以该地址应该为 0x10000c，即标号为 entry 的位置。首先设置 CR4 的 PSE 为 1，这说明采用大页模式，页帧为 4MB，同时设置了页表基址寄存器指向 entrypgdir，其设置 0~4MB 和

0x80000000~0x80000000+4MB的空间的访问均映射到0~4MB的物理地址中(此处设置0~4MB映射到0~4MB是因为此时bootmain还运行在0~4MB下),置位CR0的PG位,开启分页模式。同时设置esp栈顶指向为stack+KSTACKSIZE(KSTACKSIZE为4KB,stack是在该文件的末尾声明的,最后位置为0x8010a5d0),然后执行main()函数。

因为该ELF内核文件在编译时的运行地址为0x8000000,所以main函数的位置在0x80000000+,因为此时所有的操作应当在0x80000000~0x80000000+4MB位置。

章的内容。

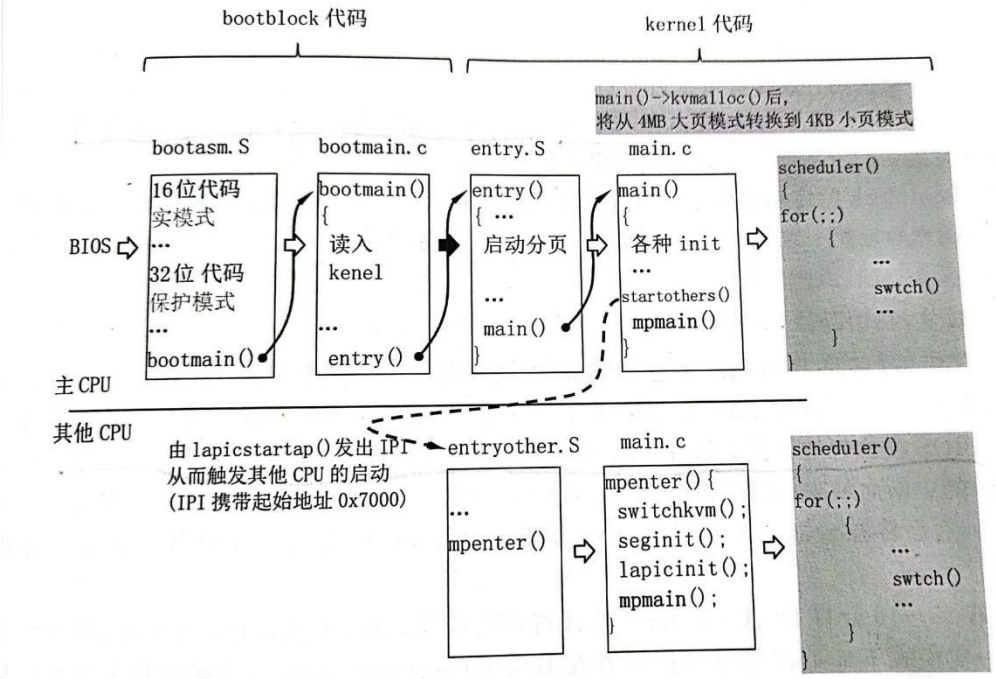


图4-2 xv6启动过程一览

系统启动

首先是kinit1()和kvmalloc()两个函数进行内存初始化。

先来看看kinit1()函数,传递的参数为在4MB空间中除去kernel占据的末尾部分和到0x80000000+4MB部分。在内存管理中有一个kmem结构体,其中有一个freelist结构体,指的是空闲链表,每个链表的位置相隔4KB内存空间,即一个页帧。使用

freerange()函数将该 vstart-vend 部分初始化为页帧。

然后调用 kvm alloc()函数，使用 setupkvm ()函数，根据 km ap 结构体构建内核页表映射（km ap 数组描述了 I/O 空间、内核代码段+ 可读数据段、内核数据段、其他设备段的划分使用情况），最后返回一个内核页表，使用内联汇编函数 lc3()值得页表基址寄存器指向 kpgdir。（此处采用的是直接映射或一致映射，虚地址和物理地址之间恒定差一个常数偏移，这样很容易在物理地址和内核虚地址之间进行转换，当然，我们此时是假设内核代码全部在内存中才行，没有缺页中断）。

随后在 m pinit()多核启动初始化函数，此处不做深入讨论。

然后是 lapicinit()本地中断控制器初始化函数，因为此处我假设使用的是单核，所以直接使用主 PIC 芯片，即使用 8259A 中断控制器来处理中断；如果是多核处理器，则采用 APIC 来处理中断。

然后是 seginit()函数，该还是那会对段表中的段进行初始化，在原来的两个内核段的基础上增加了用户态段，（如果有多个处理器，则在每个处理器上都要执行一次）即 SEG_UCODE 和 SEG_UDATA。

然后进入硬件设置，即中断使能、中断控制器、控制台和串口设备函数。

LAPIC 和 IOAPIC 的关系

APIC 经历了 APIC、xAPIC 和 x2APIC，APIC 使用专用的 APIC 总线来传递中断消息，而 xAPIC 使用系统总线。x86 多核系统上的中断采用 APIC 机制，其中断控制器划分为两个部分

- 一个是在 I/O 系统中的 IOAPIC，用于分发中断到某个处理器

- 另一部分是关联在每一个处理器上的 LAPIC（本地 APIC/Local APIC），用来处理发送给它的中断。

硬件部分较为麻烦，后续会在后面附加上。

然后是对内存区域的一些初始化，分别是进程控制块、中断向量、文件表和相关缓冲区的初始化。

进程表 ptable 的初始化在 pinit()函数中，主要是对进程表 ptable 初始化一个互斥锁（spinlock 为自旋锁），锁名为 ptable。

中断处理函数初始化是在 tvinit()中。该函数主要初始化了 256 个中断向量，即将 vector[]中的地址转换成了“门”填写到了 IDT 中。

vector[]是什么？

可见 vectors.S 文件，其在末尾初始化了 256 个 vector#，分别冲 vector0/1/2……vector255，而前面的各个标号则是对应 vector# 的处理函数，做的事情差不多，都是将对应的编号压入堆栈（内核栈）后跳转到公共入口代码 alltraps，在弃用了内核数据段和 per-CPU 段之后，alltraps 中根据编号进一步调用 trap()。

binit()是磁盘缓冲区初始化。完成了互斥锁的初始化，并将缓冲区构成链表。

fileinit()通过 initlock()完成对 ftable 的互斥锁的初始化，ftable 记录系统所打开的文件，总数不超过 NFILE=100。

ideinit()是对 IDE 控制器的初始化，包括互斥锁的初始化、相应的中断使能以及检测是否有 driver1。

kinit2()函数将 4MB~240MB 地址范围的空闲页帧构成链表。

userinit()创建第一个用户态进程。

mpmain()将 IDT 表的起始地址装入到 IDTR 寄存器中，然后开始执行 schedule()内核执行流。schedule()执行流即 schedule 调度器的无限循环（不再返回），每隔一个 tick 时钟中断就选取下一个就绪进程来执行。

内存管理

前面说过 kinit1()函数将内核末尾~4MB 的空间形成了空闲链表，kinit2()将 4MB~PHYSTOP 部分形成了空闲链表，而在 0~内核末尾的部分则是直接被 mappage()到了 PD 表和 PE 表中（此处确实有些神迷，因为此时内核已经完全在内存中了，再去映射未免显得有些虚假）。

在内核初始化的时候对内存区域管理已经大致了解了，接下来谈一谈用户空间映像。用户空间的分配和回收是由 allocuvm()和 deallocuvm()两个函数来完成的。allocuvm()函数主要用来分配页帧和建立页表映射：调用 kalloc()函数，获得一个页帧的首地址并执行清 0 操作，然后在 mappages()中构建页表映射；deallocuvm()同理：kfree()一个页帧并挂到空闲链表头部，然后将相对应的页表项清 0 即可。

fork()所产生的操作是复制父进程的所有资源，因此有 copyuvm()函数用于从父进程复制出一个新的页表并分配新的内存、复制内存数据，新的内存布局和父进程的完全一样。

除此之外，还有一个 sbrk()函数，用于调整进程空间的大小。

进程管理

这里简要描述一下进程的调度过程：在 schedule 执行流的过程中，如果进程调度算法采用的是轮转调度法，那么每次时钟中断都进入到 xv6 内核代码，完成 IRQ_TIMER 相关的处理，然后原因 yield()让出当前 CPU（切换到其他就绪进程），调度下一个就绪进程。当然，也会有一些其他时机，也会触发调度。

一般情况下，我们把构建一个名为进程控制块的结构，负责记录和组织所有进程的 PCB，每个 PCB 包括进程的进程号、父进程、进程名等。在 xv6 内核中，进程有六种状态，分别是未使用态 UNUSED、创建中 EMBRYO、睡眠阻塞态 SLEEPING、就绪态 RUNNABLE、

运行态 RUNNING 和僵尸态 ZOMBIE。在初始化 PCB 中所有的进程都被标位 UNUSED。

此处有一个结构：trapframe—陷阱帧。在中断的时候会压入某个信息用来保存，为了在中断之后进行返回恢复寄存器信息，如硬件中断会在堆栈中压入 ss、esp、eflags、cs 和 ip 等几个寄存器，然后进入 IDT 中寻找相关中断号压入堆栈并进入到 alltraps 中。除了硬件自动压入的一些寄存器之后还会压入一些其他寄存器，如 ds、es、fs 和 gs，然后 pushal指令压入 eax、ecx、edx、ebx、oesp、ehp、esi和 edi等几个寄存器，此时才算构造了完整的 trapframe，保存了用户态断点的所有信息，也就是恢复被中断的用户态执行现场所需的全部信息。（如果中断是从内核态发生的，则 trapframe 顶部没有 ss 和 esp，直接从 eflags 开始）。在内核切换现场中（此处的内核切换现场指的是进程切换，而不是进程从用户态到内核态），在进程切换前要保存您当前进程的内核态执行现场，并恢复切入进程的执行现场，由于进程切换都发生在内核态，而内核态的段寄存器都是相同的，因此无需保存段寄存器，而 eax、ecx 和 edx 也不需要保存（按照 x86 的固定是调用者保存），esp 也不需要保存，因为 context本身就是堆栈栈顶位置，于是进程内核断点切换现场 context只有成员 edi、esi、ebx、ebp 和 eip。

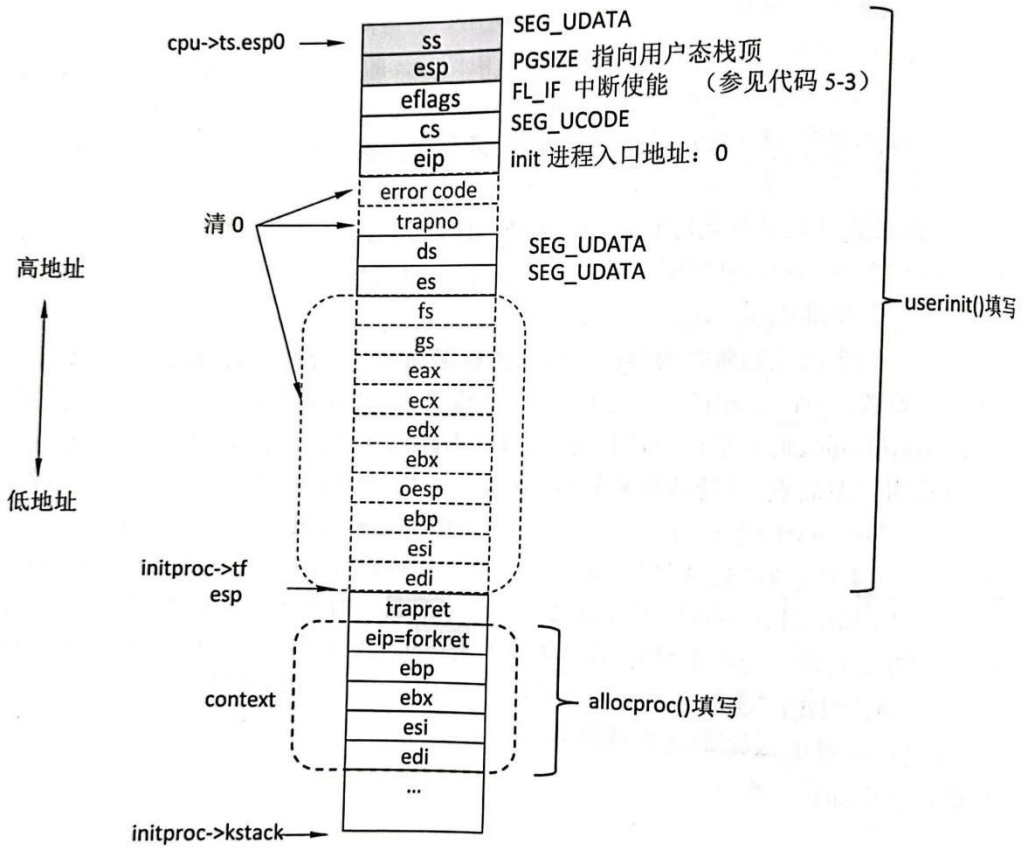


图 6-4 init 进程刚创建时的内核栈及其“伪造”的 trapframe 和 context

此处着重描述一下 init 进程。在 userinit()中将完成第一个用户进程 init 的创建工作，这个进程映像很快随着 initcode 的执行而通过 SYS_exec 系统调用替换成磁盘上的“/init”进程映像，启动 sh 程序。使用专门的全局变量 initproc 来记录这个进程，然后通过 setupkvm ()给 init 进程创建初始页表，由于只映射了内核空间，因此函数名使用 kvm ，此时还没有用户态页表（不能访问用户态空间），接着通过 initkvm ()完成用户空间的建立，由于 init 的代码已经在内核映像 kernel中随着启动过程装载到_binary_initcode_start 地址（initcode.S 在 kernel中，文件偏移为 0xa460，PA 为 0x10a460，VA 为 0x80 10a460），因此只需要新分配一个页帧，然后将该页帧映射到 0 地址（进程用户代码和数据使用虚拟地址空间的低地址部分，高地址留给内核），再将 init 代码复制到该地址即可（使用 m em m ove()函数）。init 进程是从 0 地址开始存放的（虚拟地址空间），然后填

写陷阱帧内容，等同于“好像曾经”从用户态经过中断而形成的 trapfram e。

唤醒-等待机制。即 w akeup1 ()和 sleep ()函数。sleep ()函数传入参数为一个资源和一个互斥锁，当该资源繁忙的时候，调用 sleep ()函数，让当前进程进入睡眠阻塞态，并进行调度切换（注意此处用的调度切换是 sched ()函数，schedule ()函数是无限运行的），此处未设置等待队列，而是使用 proc-chan= chan 的操作来模拟一个资源等待，并且使用 acquire (& ptable .lock)来锁住该资源（ptable 具体唯一性，此处是睡眠阻塞，当资源繁忙结束的时候它应当是第一个获得该资源的进程(只有当第一个检测出该资源忙碌的进程才能获得 ptable 锁，当唤醒的时候也只有该进程能唤醒），此处是切换成就绪态，不能直接切换成运行态），然后在唤醒的时候通过遍历获得该进程并检验条件即可。当进程切换执行流回来的时候继续执行该代码，释放 ptable 互斥锁让其他进程获得该资源即可。

此处我想对 sh 程序进行一个简要说明。sh 即 shell程序，可以执行外部程序命令、重定向命令和管道命令。shell会通过 getcm d()函数不断读入命令行的命令并执行该命令，除了 cd 直接通过 chdir()实现外，其他的命令（含内部命令）通过创建子进程去执行 runcm d(buf)而完成，其主要是根据命令类型去分发处理（一般有 EXEC、RED IR、PIPE、LIST 和 BAKC 五种类型）：EXEC—利用 exec ()系统调用完成，需要从外部传入命令中将外部程序名和命令行参数传递给 exec ()系统调用；RED IR—需要线将标准输出/输出文件替换为指定文件之后，将修改后的命令再次交给 runcm d()去处理；PIPE—首先通过系统调用 pipe ()创建管道，然后创建两个下一级的进程，然后将一个进程的标准输出文件修改为管道的输出端，另一个进程的标准输入文件修改为管道的输入端、其中管道命令左端的命令通过 runcm d()提交给第一个进程，管道右边的命令通过 runcm d()提交给第二个进程；LIST：一般会出现多个命令列表；BACK：属于后台命令，创建一个子进程并执行 runcm d()完成该操作，但无需等待子进程的结束，runcm d()可直接返回，从而使得 shell可以输出

提示符等待新命令，shell又出现在终端的前台。

文件管理

参考 EX2 文件系统，此处不做概述

高级实验

最后来看看高级实验吧，有四部分：内核线程、文件系统实验、虚拟内存和用户终端实现，此处主要讲一下难度较大的内核线程和虚拟内存实现。

首先想内核线程。

不管是内核线程还是用户进程，它们对内核来说都是用 PCB 来表示的一个调度实体，进程和线程的基本的数据结构是一致的。内核线程一般只在内核虚拟地址空间范围内活动。内核线程和用户线程最大的区别就在于此，而且，内核线程使用内核函数的时候不用使用陷阵帧，直接调用即可，而用户线程在内核看来是一个进程，进程从用户态到内核态要保存信息到 tram fram e 中。

在 xv6 中的实验我认为是用户线程，只不过为何它要定义为内核线程。

一般情况下线程要自己的 TCB、进程有自己的 PCB，此处将进程和线程放在同一个 TCB 中，除此之外，线程是有自己的线程栈的，它不用进程（主线程，我们一般把产生线程的进程称为主线程）的栈，作为私有资源。在内核态，线程的初始状态栈被设置了线程执行的起点（伪造返回的 PC 指针）传递的参数值

虚拟内存实验。

一般情况下，仅允许进程空间堆栈后面 sbrk()分配的空间换出，堆栈前面的内容不换出（exec()中生成的进程空间）。虚拟内存需要和请求式分页相结合，所以在申请的物理页帧之前，系统不会进行分配操作，直到产生缺页中断，才调用相应的中断程序进行分配。

此处仍然默认使用 xv6 的文件系统作为交换磁盘。

当页帧被换出之后，换出的页帧所在的盘块号直接保存在其 pte 的高位，其低 12 位仍用做标志用途。交换机制的作用范围仅限于本进程的 sbrk()之上的物理页帧，而且一个进程并不去抢占其他进程的物理页帧。

此处要提及一些虚拟地址空间的分布，其高地址地区，即从 0x80000000 开始是内核区域，从 0~0x80000000 是用户进程地址空间，此处可通过 readelf 或 nm 工具验证。根据 exec()进程产生的原理，所有的进程都是 init 进程的子进程或者后代进程，所有的进程都是复制他们的父进程的内存映像，根据 init 的进程映像可知，其进程用户地址空间是从 0 开始的，然后开始映射。

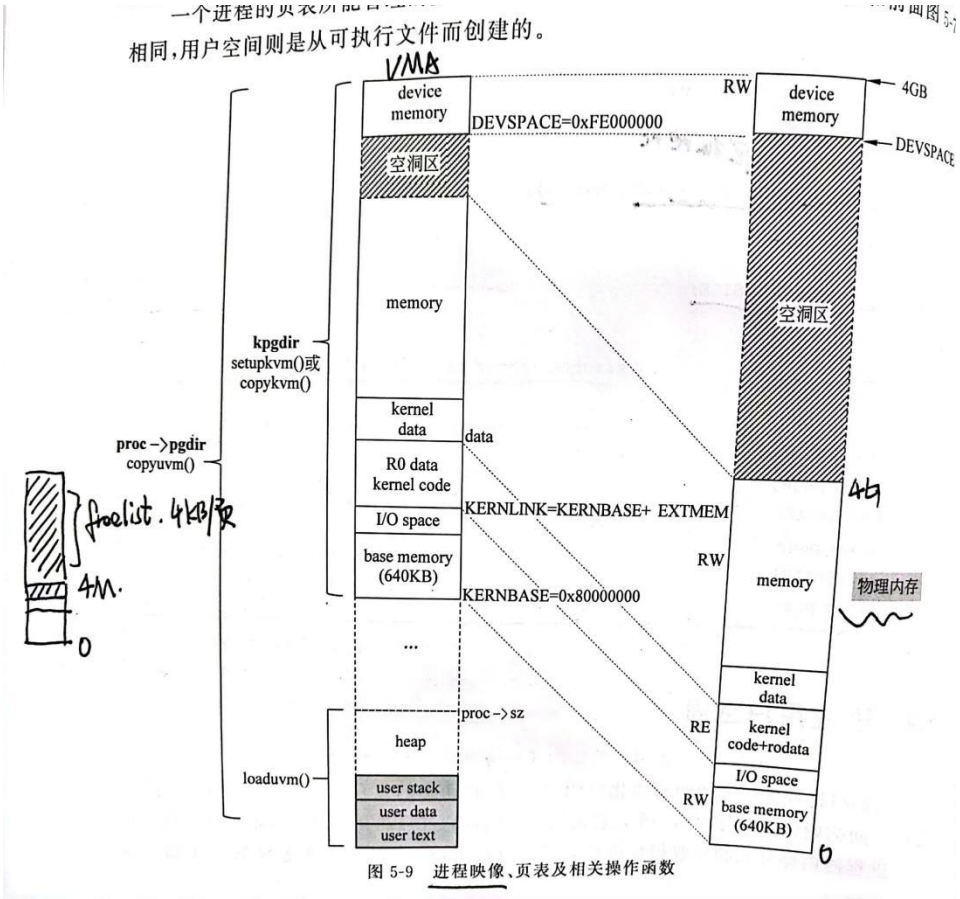


图 5-9 进程映像、页表及相关操作函数

由于我们将页面调出到磁盘后，进程再去访问就会产生缺页中断，当找到一块已经有映射的页换出去时，调用 kmalloc()函数分配一个物理页帧并修改 pte 即可。

最后来总结一下除了和 Linux 系统相似性之外我认为 xv6 的特点所在：

- 启动分页使用了大页（4MB）和 4KB 模式

- 多核系统启动，中断控制器使用 APIC

- 内核同步使用自旋锁作为互斥锁

- 实现了 sbrk()堆空间分配

- 文件系统使用了日志层，iNode 中使用了混合索引

- 缓冲区管理使用的是循环缓冲区

- 进程六态

xv6 代码：https://github.com/mit-pdos/xv6-public

我的博客：https://blog.csdn.net/qq_48322523?type=blog