Chapter 3 Page tables

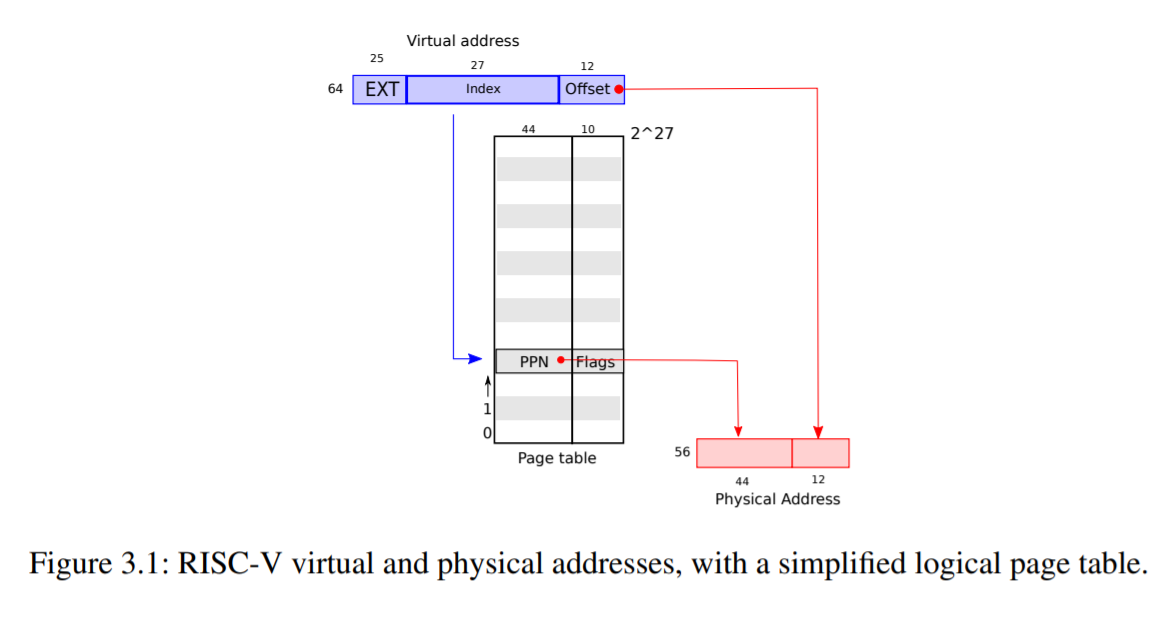
页表是操作系统为每个进程提供自己私有地址空间和内存的机制。页表决定了内存地址的含义，以及物理内存的哪些部分可以被访问。它们允许xv6隔离不同进程的地址空间，并将它们映射到物理内存上。页表还提供了一个间接层次，允许xv6执行一些技巧：在几个地址空间中映射同一内存(trampoline页)，以及用一个未映射页来保护内核和用户的栈。本章其余部分将解释RISC-V硬件提供的页表以及xv6如何使用它们。

### 3.1 Paging hardware

提醒一下，RISC-V指令(包括用户和内核)操作的是虚拟地址。机器的RAM，或者说物理内存，是用物理地址来做索引的，RISC-V分页硬件[[1]](#footnote-1)将这两种地址联系起来，通过将每个虚拟地址映射到物理地址上。

xv6 运行在 Sv39 RISC-V 上，这意味着只使用 64 位虚拟地址的底部 39 位，顶部 25 位未被使用。在这种Sv39配置中，一个RISC-V 页表在逻辑上是一个2^27（134,217,728）***页表项（Page Table Entry, PTE）***的数组。每个PTE包含一个44位的**物理页号(Physical Page Number，PPN)**和一些标志位。分页硬件通过利用39位中的高27位索引到页表中找到一个**PTE**来转换一个虚拟地址，并计算出一个56位的物理地址，它的前44位来自于**PTE**中的**PPN**，而它的后12位则是从原来的虚拟地址复制过来的。图3.1显示了这个过程，在逻辑上可以把页表看成是一个简单的**PTE**数组（更完整的描述见图3.2）。页表让操作系统控制虚拟地址到物理地址的转换，其粒度为4096（2^12）字节的对齐块。这样的分块称为页。

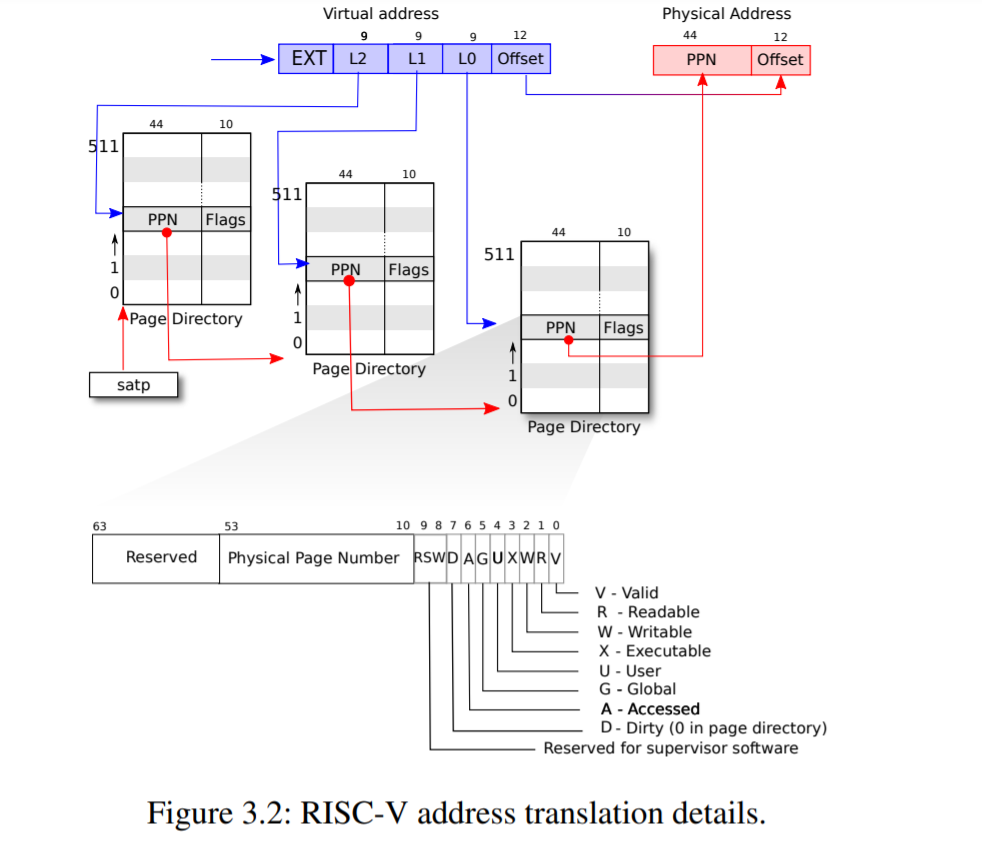
在Sv39 RISC-V中，虚拟地址的前25位不用于转换地址；将来，RISC-V可能会使用这些位来定义更多的转换层。物理地址也有增长的空间：在**PTE**格式中，物理页号还有10位的增长空间。



如图3.2所示，实际转换分三步进行。一个页表以三层树的形式存储在物理内存中。树的根部是一个4096字节的页表页，它包含512个PTE，这些PTE包含树的下一级页表页的物理地址。每一页都包含512个PTE，用于指向下一个页表或物理地址。分页硬件用27位中的顶9位选择根页表页中的PTE，用中间9位选择树中下一级页表页中的PTE，用底9位选择最后的PTE。

如果转换一个地址所需的三个PTE中的任何一个不存在，分页硬件就会引发一个页面错误的异常(***page-fault exception***)，让内核来处理这个异常（见第4章）。这种三层结构的一种好处是，当有大范围的虚拟地址没有被映射时，可以省略整个页表页。

每个PTE包含标志位，告诉分页硬件如何允许使用相关的虚拟地址。**PTE\_V**表示PTE是否存在：如果没有设置，对该页的引用会引起异常（即不允许）。PTE\_R控制是否允许指令读取到页。**PTE\_W**控制是否允许指令向写该页。**PTE\_X**控制CPU是否可以将页面的内容解释为指令并执行。**PTE\_U**控制是否允许用户模式下的指令访问页面；如果不设置**PTE\_U**，PTE只能在监督者模式下使用。图3.2显示了这一切的工作原理。标志位和与页相关的结构体定义在(kernel/riscv.h)。

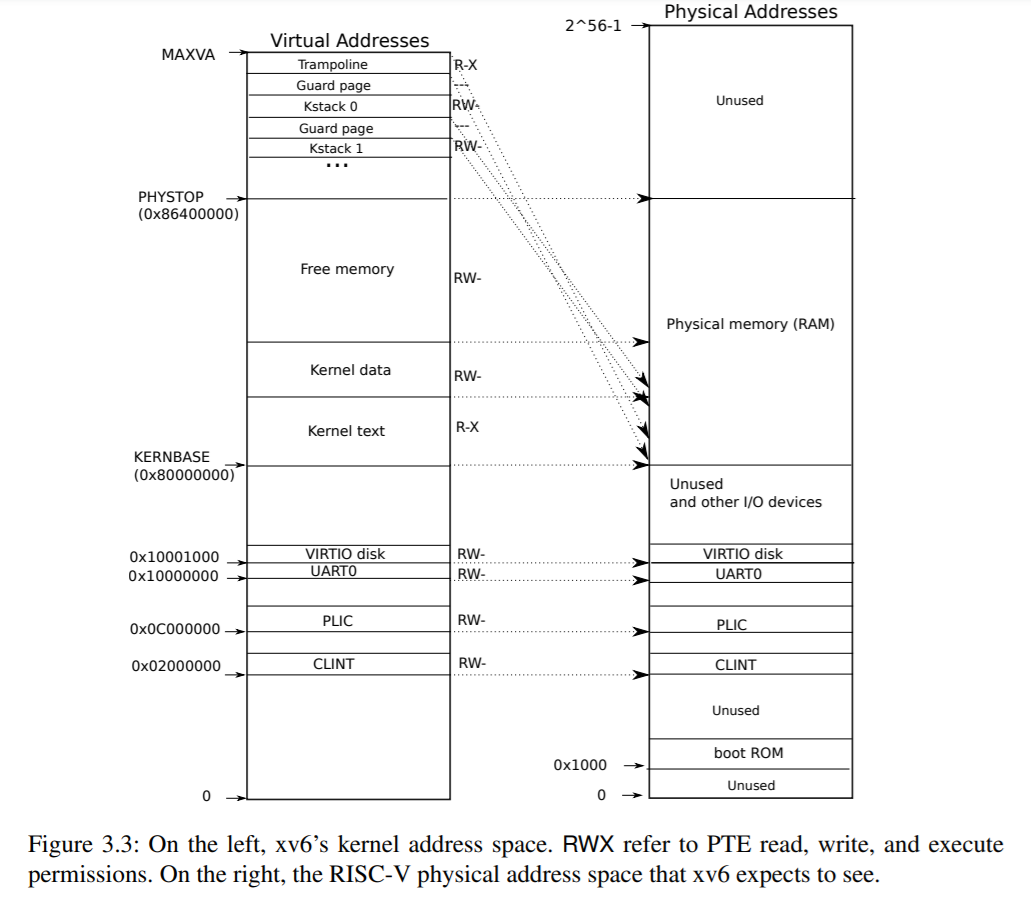


要告诉硬件使用页表，内核必须将根页表页的物理地址写入**satp**寄存器中。每个CPU都有自己的**satp**寄存器。一个CPU将使用自己的**satp**所指向的页表来翻译后续指令产生的所有地址。每个CPU都有自己的**satp**，这样不同的CPU可以运行不同的进程，每个进程都有自己的页表所描述的私有地址空间。

关于术语的一些说明。物理内存指的是**DRAM**中的存储单元。物理存储器的一个字节有一个地址，称为物理地址。当指令操作虚拟地址时，分页硬件会将其翻译成物理地址，然后发送给DRAM硬件，以读取或写入存储。不像物理内存和虚拟地址，虚拟内存不是一个物理对象，而是指内核提供的管理物理内存和虚拟地址的抽象和机制的集合。

### 3.2 Kernel address space

Xv6为每个进程维护页表，一个是进程的用户地址空间，外加一个内核地址空间的单页表。内核配置其地址空间的布局，使其能够通过可预测的虚拟地址访问物理内存和各种硬件资源。图3.3显示了这个设计是如何将内核虚拟地址映射到物理地址的。文件(kernel/memlayout.h)声明了xv6内核内存布局的常量。



QEMU模拟的计算机包含RAM（物理内存），从物理地址**0x80000000**，至少到0x86400000，xv6称之为**PHYSTOP**。QEMU模拟还包括I/O设备，如磁盘接口。QEMU将设备接口作为***memory-mapped(内存映射)***控制寄存器暴露给软件，这些寄存器位于物理地址空间的**0x80000000**以下。内核可以通过读取/写入这些特殊的物理地址与设备进行交互；这种读取和写入与设备硬件而不是与RAM进行通信。第4章解释了xv6如何与设备交互。

内核使用“直接映射”**RAM**和***内存映射***设备寄存器，也就是在虚拟地址上映射硬件资源，这些地址与物理地址相等。例如，内核本身在虚拟地址空间和物理内存中的位置都是**KERNBASE=0x80000000**。直接映射简化了读/写物理内存的内核代码。例如，当fork为子进程分配用户内存时，分配器返回该内存的物理地址；**fork**在将父进程的用户内存复制到子进程时，直接使用该地址作为虚拟地址。

有几个内核虚拟地址不是直接映射的:

1．trampoline 页。它被映射在虚拟地址空间的顶端；用户页表也有这个映射。第4章讨论了trampoline 页的作用，但我们在这里看到了页表的一个有趣的用例；一个物理页（存放trampoline代码）在内核的虚拟地址空间中被映射了两次：一次是在虚拟地址空间的顶部，一次是直接映射。

2．内核栈页。每个进程都有自己的内核栈，内核栈被映射到地址高处，所以在它后面xv6可以留下一个未映射的守护页。守护页的PTE是无效的（设置**PTE\_V**），这样如果内核溢出内核stack，很可能会引起异常，内核会报错。如果没有防护页，栈溢出时会覆盖其他内核内存，导致不正确的操作。报错还是比较好的

当内核通过高地址映射使用stack时，它们也可以通过直接映射的地址被内核访问。另一种的设计是只使用直接映射，并在直接映射的地址上使用stack。在这种安排中，提供保护页将涉及到取消映射虚拟地址，否则这些地址将指向物理内存，这将很难使用。

内核为trampoline和text(可执行程序的代码段)映射的页会有**PTE\_R**和**PTE\_X**权限。内核从这些页读取和执行指令。内核映射的其他page会有**PTE\_R**和**PTE\_W**权限，以便内核读写这些页面的内存。守护页的映射是无效的（设置**PTE\_V**）；

### 3.3 Code: creating an address space

大部分用于操作地址空间和页表的xv6代码都在**vm.c**(kernel/vm.c:1)中。核心数据结构是**pagetable\_t**，它实际上是一个指向RISC-V根页表页的指针；**pagetable\_t**可以是内核页表，也可以是进程的页表。核心函数是**walk**和**mappages**，前者通过虚拟地址得到PTE，后者将虚拟地址映射到物理地址。以 **kvm** 开头的函数操作内核页表；以 uvm 开头的函数操作用户页表；其他函数用于这两种页表。**copyout** 可以将内核数据复制到用户虚拟地址，**copyin**可以将用户虚拟地址的数据复制到内核空间地址，用户虚拟地址由系统调用的参数指定；它们在 **vm.c** 中，因为它们需要显式转换这些地址，以便找到相应的物理内存。

在启动序列的前面，**main**调用**kvminit**(kernel/vm.c:22)来创建内核的页表。这个调用发生在xv6在RISC-V启用分页之前，所以地址直接指向物理内存。**Kvminit**首先分配一页物理内存来存放根页表页。然后调用**kvmmap**将内核所需要的硬件资源映射到物理地址。这些资源包括内核的指令和数据，KERNBASE到PHYSTOP（0x86400000）的物理内存，以及实际上是设备的内存范围。

**kvmmap** (kernel/vm.c:118) 调用 **mappages** (kernel/vm.c:149)，它将一个虚拟地址范围映射到一个物理地址范围。它将范围内地址分割成多页（忽略余数），每次映射一页的顶端地址。对于每个要映射的虚拟地址（页的顶端地址），mapages调用walk找到该地址的最后一级PTE的地址。然后，它配置PTE，使其持有相关的物理页号、所需的权限(**PTE\_W、PTE\_X和/或PTE\_R**)，以及**PTE\_V**来标记PTE为有效**(kernel/vm.c:161)**。

**walk (kernel/vm.c:72)**模仿RISC-V分页硬件查找虚拟地址的PTE(见图3.2)。walk每次降低3级页表的9位。它使用每一级的9位虚拟地址来查找下一级页表或最后一级（kernel/vm.c:78）的PTE。如果PTE无效，那么所需的物理页还没有被分配；如果**alloc**参数被设置**true**，**walk**会分配一个新的页表页，并把它的物理地址放在PTE中。它返回PTE在树的最低层的地址(kernel/vm.c:88)。

**main** 调用 **kvminithart** (kernel/vm.c:53) 来映射内核页表。它将根页表页的物理地址写入寄存器**satp**中。在这之后，CPU将使用内核页表翻译地址。由于内核使用唯一映射，所以指令的虚拟地址将映射到正确的物理内存地址。

**procinit** (kernel/proc.c:26)，它由**main**调用，为每个进程分配一个内核栈。它将每个栈映射在**KSTACK**生成的虚拟地址上，这就为栈守护页留下了空间。**Kvmmap**栈的虚拟地址映射到申请的物理内存上，然后调用**kvminithart**将内核页表重新加载到**satp**中，这样硬件就知道新的PTE了。

每个RISC-V CPU都会在***Translation Look-aside Buffer(TLB)***中缓存页表项，当xv6改变页表时，必须告诉CPU使相应的缓存TLB项无效。如果它不这样做，那么在以后的某个时刻，TLB可能会使用一个旧的缓存映射，指向一个物理页，而这个物理页在此期间已经分配给了另一个进程，这样的话，一个进程可能会在其他进程的内存上“乱写乱画“。RISC-V有一条指令**sfence.vma**，可以刷新当前CPU的TLB。xv6在重新加载satp寄存器后，在**kvminithart**中执行**sfence.vma**，也会在从内核空间返回用户空间前，切换到用户页表的trampoline代码中执行**sfence.vma**(kernel/trampoline.S:79)。

### 3.4 Physical memory allocation

内核必须在运行时为页表、用户内存、内核堆栈和管道缓冲区分配和释放物理内存。 xv6使用内核地址结束到**PHYSTOP**之间的物理内存进行运行时分配。它每次分配和释放整个4096字节的页面。它通过保存空闲页链表，来记录哪些页是空闲的。分配包括从链表中删除一页；释放包括将释放的页面添加到空闲页链表中。

### 3.5 Code: Physical memory allocator

分配器在**kalloc.c(kernel/kalloc.c:1)**中。分配器的数据结构是一个可供分配的物理内存页的***空闲页链表***，每个空闲页的链表元素是一个结构体**run（kernel/kalloc.c:17）**。分配器从哪里获得内存来存放这个结构体呢？它把每个空闲页的**run**结构体存储在空闲页本身，因为那里没有其他东西存储。空闲链表由一个***自旋锁***保护(kernel/kalloc.c:21-24)。链表和锁被包裹在一个结构体中，以明确锁保护的是结构体中的字段。现在，请忽略锁以及**acquire**和**release**的调用；第6章将详细研究锁。

**main**函数调用**kinit**来初始化分配器(kernel/kallo .c:27)。**kinit**初始空闲页链表，以保存内核地址结束到**PHYSTOP**之间的每一页。xv6应该通过解析硬件提供的配置信息来确定有多少物理内存可用。但是它没有做，而是假设机器有128M字节的RAM。**Kinit**通过调用**freerange**来添加内存到空闲页链表，**freerange**则对每一页都调用**kfree**。PTE只能引用按4096字节边界对齐的物理地址(4096的倍数)，因此**freerange**使用**PGROUNDUP**来确保它只添加对齐的物理地址到空闲链表中。分配器开始时没有内存;这些对**kfree**的调用给了它一些内存管理。

分配器有时把地址当作整数来处理，以便对其进行运算（如**freerange**遍历所有页），有时把地址当作指针来读写内存（如操作存储在每页中的**run**结构体）；这种对地址的双重使用是分配器代码中充满C类型转换的主要原因。另一个原因是，释放和分配本质上改变了内存的类型。

函数**kfree** (kernel/kalloc.c:47)将被释放的内存中的每个字节设置为1。这将使得释放内存后使用内存的代码(使用悬空引用)读取垃圾而不是旧的有效内容；希望这将导致这类代码更快地崩溃。然后**kfree**将页面预存入释放列表：它将**pa**（物理地址）转为指向结构体**run**的指针，在**r->next**中记录空闲链表之前的节点，并将释放列表设为**r**，**kalloc**移除并返回空闲链表中的第一个元素。

### 3.6 Process address space

每个进程都有一个单独的页表，当xv6在进程间切换时，也会改变页表。如图2.3所示，一个进程的用户内存从虚拟地址0开始，可以增长到MAXVA(kernel/riscv.h:348)，原则上允许一个进程寻址256GB的内存。

当一个进程要求xv6提供更多的用户内存时，xv6首先使用**kalloc**来分配物理页，然后将指向新物理页的PTE添加到进程的页表中。然后它将指向新物理页的PTE添加到进程的页表中。Xv6在这些PTE中设置**PTE\_W**、**PTE\_X**、**PTE\_R**、**PTE\_U**和**PTE\_V**标志。大多数进程不使用整个用户地址空间；xv6使用**PTE\_V**来清除不使用的PTE。

我们在这里看到了几个例子，是关于使用页表的。首先，不同的进程页表将用户地址转化为物理内存的不同页，这样每个进程都有私有的用户内存。第二，每个进程都认为自己的内存具有从零开始的连续的虚拟地址，而进程的物理内存可以是不连续的。第三，内核会映射带有**trampoline** 代码的页，该**trampoline**处于用户地址空间顶端，因此，在所有地址空间中都会出现一页物理内存。

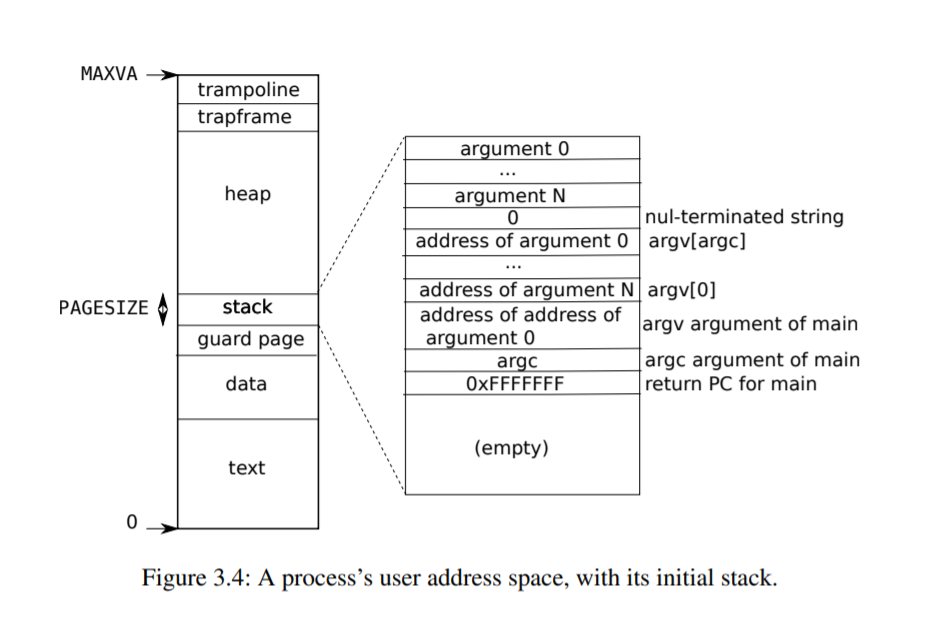


图3.4更详细地显示了xv6中执行进程的用户内存布局。栈只有一页，图中显示的是由**exec**创建的初始内容。字符串的值，以及指向这些参数的指针数组，位于栈的最顶端。下面是允许程序在**main**启动的值，就像函数**main(argc, argv)**刚刚被调用一样。

为了检测用户栈溢出分配的栈内存，xv6会在stack的下方放置一个无效的保护页。如果用户栈溢出，而进程试图使用栈下面的地址，硬件会因为该映射无效而产生一个页错误异常。现实世界中的操作系统可能会在用户栈溢出时自动为其分配更多的内存。

### 3.7 Code: sbrk

**Sbrk**是一个进程收缩或增长内存的系统调用。该系统调用由函数**growproc**(kernel/proc.c:239)实现，**growproc**调用**uvmalloc**或**uvmdealloc**，取决于n是正数还是负数。 **uvmdealloc** 调用 **uvmunmap** (kernel/vm.c:174)，它使用 walk 来查找 PTE，使用 **kfree** 来释放它们所引用的物理内存。

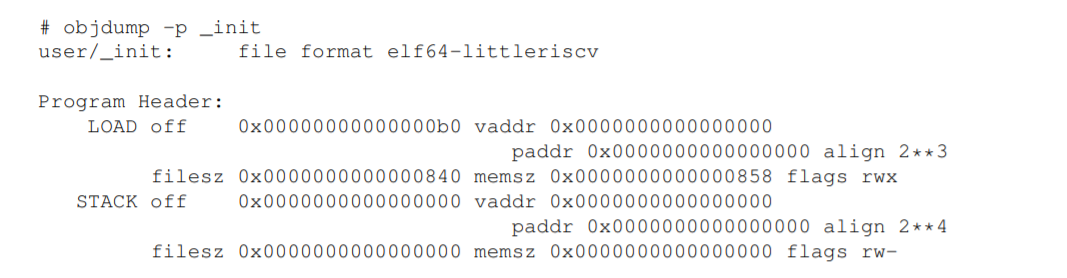
xv6使用进程的页表不仅是为了告诉硬件如何映射用户虚拟地址，也是将其作为分配给该进程的物理地址的唯一记录。这就是为什么释放用户内存（**uvmunmap**中）需要检查用户页表的原因。

### 3.8 Code: exec

**Exec**是创建用户地址空间的系统调用。它读取储存在文件系统上的文件用来初始化用户地址空间。**Exec** (kernel/exec.c:13)使用**namei** (kernel/exec.c:26)打开二进制文件路径，这在第8章中有解释。然后，它读取ELF头。xv6应用程序用ELF格式来描述可执行文件，它定义在**(kernel/elf.h)**。一个ELF二进制文件包括一个ELF头，**elfhdr**结构体**(kernel/elf.h:6)**，后面是一个程序节头(program section header)序列，程序节头为一个结构体**proghdr**(kernel/elf.h:25)。每一个**proghdr**描述了一个必须加载到内存中的程序节；xv6程序只有一个程序节头，但其他系统可能有单独的指令节和数据节需要加载到内存。

第一步是快速检查文件是否包含一个ELF二进制文件。一个ELF二进制文件以四个字节的”魔法数字“ 0x7F、`E`、`L`、`F`或ELF\_MAGIC(kernel/elf.h:3)开始。如果ELF头有正确的”魔法数字“，exec就会认为该二进制文件是正确的类型。

**Exec** 使用**proc\_pagetable(kernel/exec.c:38)**分配一个没有使用的页表，使用**uvmalloc** **(kernel/exec.c:52)**为每一个ELF 段分配内存，通过**loadseg (kernel/exec.c:10)**加载每一个段到内存中。**loadseg**使用**walkaddr**找到分配内存的物理地址，在该地址写入ELF段的每一页，页的内容通过**readi**从文件中读取。



用**exec**创建的第一个用户程序**/init**的程序部分header是上面这样的。

程序部分头的**filesz**可能小于**memsz**，说明它们之间的空隙应该用0（用于C语言全局变量）来填充，而不是从文件中读取。对于**/init**来说，**filesz**是2112字节，**memsz**是2136字节，因此**uvmalloc**分配了足够的物理内存来容纳2136字节，但只从文件**/init**中读取2112字节。

**exec**在栈页的下方放置了一个不可访问页，这样程序如果试图使用多个页面，就会出现故障。这个不可访问的页允许exec处理过大的参数；在这种情况下，exec用来复制参数到栈的copyout(kernel/vm.c:355)函数会注意到目标页不可访问，并返回-1。

在准备新的内存映像的过程中，如果exec检测到一个错误，比如一个无效的程序段，它就会跳转到标签bad，释放新的映像，并返回-1。**exec**必须延迟释放旧映像，直到它确定exec系统调用会成功：如果旧映像消失了，系统调用就不能返回-1。exec中唯一的错误情况发生在创建映像的过程中。一旦镜像完成，exec就可以提交到新的页表(kernel/exec.c:113)并释放旧的页表(kernel/exec.c:117)。

**Exec**将ELF文件中的字节按ELF文件指定的地址加载到内存中。用户或进程可以将任何他们想要的地址放入ELF文件中。因此，Exec是有风险的，因为ELF文件中的地址可能会意外地或故意地指向内核。对于一个不小心的内核来说，后果可能从崩溃到恶意颠覆内核的隔离机制(即安全漏洞)。xv6执行了一些检查来避免这些风险。例如**if(ph.vaddr + ph.memsz < ph.vaddr)**检查总和是否溢出一个64位整数。危险的是，用户可以用指向用户选择的地址的 **ph.vaddr** 和足够大的 **ph.memsz** 来构造一个 ELF 二进制，使总和溢出到 **0x1000**，这看起来像是一个有效值。在旧版本的xv6中，用户地址空间也包含内核（但在用户模式下不可读/写），用户可以选择一个对应内核内存的地址，从而将ELF二进制中的数据复制到内核中。在RISC-V版本的xv6中，这是不可能的，因为内核有自己独立的页表；**loadseg**加载到进程的页表中，而不是内核的页表中。

内核开发人员很容易忽略一个关键的检查，现实中的内核有很长一段缺少检查的空档期，用户程序可以利用缺少这些检查来获得内核特权。xv6在验证需要提供给内核的用户程序数据的时候，并没有完全验证其是否是恶意的，恶意用户程序可能利用这些数据来绕过xv6的隔离。

### 3.9 Real world

像大多数操作系统一样，xv6使用分页硬件进行内存保护和映射。大多数操作系统对分页的使用要比xv6复杂得多，它将分页和分页错误异常结合起来，我们将在第4章中讨论。

Xv6的内核使用虚拟地址和物理地址之间的直接映射，这样会更简单，并假设在地址0x8000000处有物理RAM，即内核期望加载的地方。这在QEMU中是可行的，但是在真实的硬件上，它被证明是一个糟糕的想法；真实的硬件将RAM和设备放置在不可预测的物理地址上，例如在0x8000000处可能没有RAM，而xv6期望能够在那里存储内核。更好的内核设计利用页表将任意的硬件物理内存布局变成可预测的内核虚拟地址布局。

RISC-V支持物理地址级别的保护，但xv6没有使用该功能。

在有大量内存的机器上，使用RISC-V对超级页(4MB的页)的支持可能是有意义的。当物理内存很小的时候，小页是有意义的，可以精细地分配和分页到磁盘。例如，如果一个程序只使用8千字节的内存，那么给它整整4兆字节的超级物理内存页是浪费的。更大的页在有大量内存的机器上是有意义的，可以减少页表操作的开销。

xv6内核缺乏一个类malloc的分配器为小程序提供内存，这使得内核没有使用需要动态分配的复杂数据结构，从而简化了设计。

内存分配是一个常年的热门话题，基本问题是有效利用有限的内存和为未来未知的请求做准备[7]。如今人们更关心的是速度而不是空间效率。此外，一个更复杂的内核可能会分配许多不同大小的小块，而不是（在xv6中）只分配4096字节的块；一个真正的内核分配器需要处理小块分配以及大块分配。

### 3.10 Exercises

1、分析RISC-V的设备树（device tree），找出计算机有多少物理内存。

2、编写一个用户程序，通过调用sbrk(1)使其地址空间增加一个字节。运行该程序，研究调用sbrk之前和调用sbrk之后的程序页表。内核分配了多少空间？新内存的PTE包含哪些内容？

3、修改xv6使得内核使用超级页4M)

4、修改xv6，使用户程序间接引用[[2]](#footnote-2)一个空指针时，会收到一个异常，即修改xv6，使用户程序的虚拟地址0不被映射。

5、Unix实现的exec传统上包括对shell脚本的特殊处理。如果要执行的文件以文本#!开头，那么第一行就被认为是要运行的程序来解释文件。例如，如果调用exec运行myprog arg1，而myprog的第一行是#!/interp，那么exec执行/interp myprog arg1。在xv6中实现对这个约定的支持。

1. 一般指内存管理单元(Memory Management Unit, MMU) [↑](#footnote-ref-1)
2. dereference，就是对地址取值，例如\*p。 [↑](#footnote-ref-2)