Chapter3 Page tables

Paging hardware

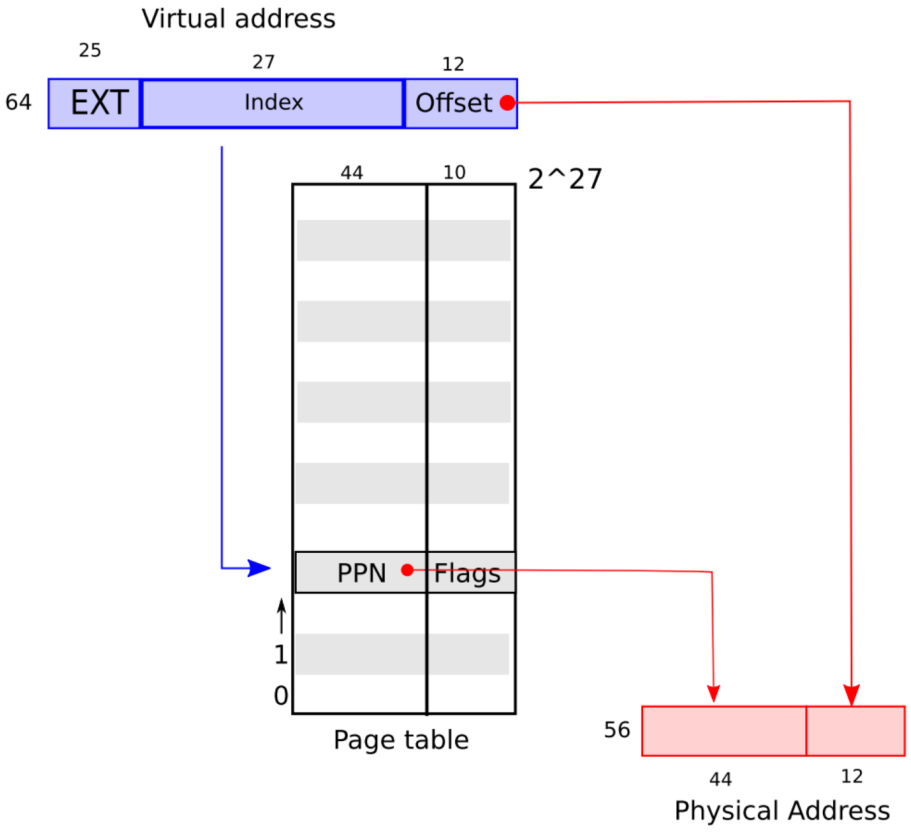
只使用了64位虚拟地址的后39位，高25位没被使用。

Sv39 RISC-V的页表逻辑上是一个2^27页表项的页表，每个PTE包含44位的物理页号(PPN)和一些符号位。

分页硬件通过39位虚拟地址高27位作为索引，在页表中寻找到PTE，并生成一个56位的物理地址，高44位来自PTE中的PPN，后12位来自原始虚拟地址的低12位。

页表允许操作系统控制在2^12对齐块粒度上的虚拟地址转换，这样的块叫做页面(page)。

在Sv39 RISC-V中，虚拟地址的高25位不用来做转换。



真正的地址翻译用三步，页表在物理内存中以三层的树状结构存储。树根是一个4096B的页表页面，其中包含512个PTEs，包含下一层级的页表页面的物理地址。这些页面每个都包含了最后一级的512个PTEs。

分页硬件使用27位的最高9位来选择跟页表页面的PTE，用中间9位去选择下一层级的页表页面，在最后的9位用来选择最终的PTE。

如果地址转换所需的三个PTE中任何一个不存在，分页硬件就抛出一个page-fault exception，让内核来处理这个异常。

层级结构，可以使在大片虚拟内存没有使用时，节省空间。

每个PTE也包含一些标志位，告诉分页硬件如何使用相关的虚拟地址。如PTE_V表示这个PTE是否存在，PTE_R控制是否指令被允许去读这个页面，PTE_W控制是否指令被允许去写这个页面，PTE_X控制是否CPU可以将页面的内容解释为指令并运行，PTE_U控制是否允许用户模式下的指令访问页面。

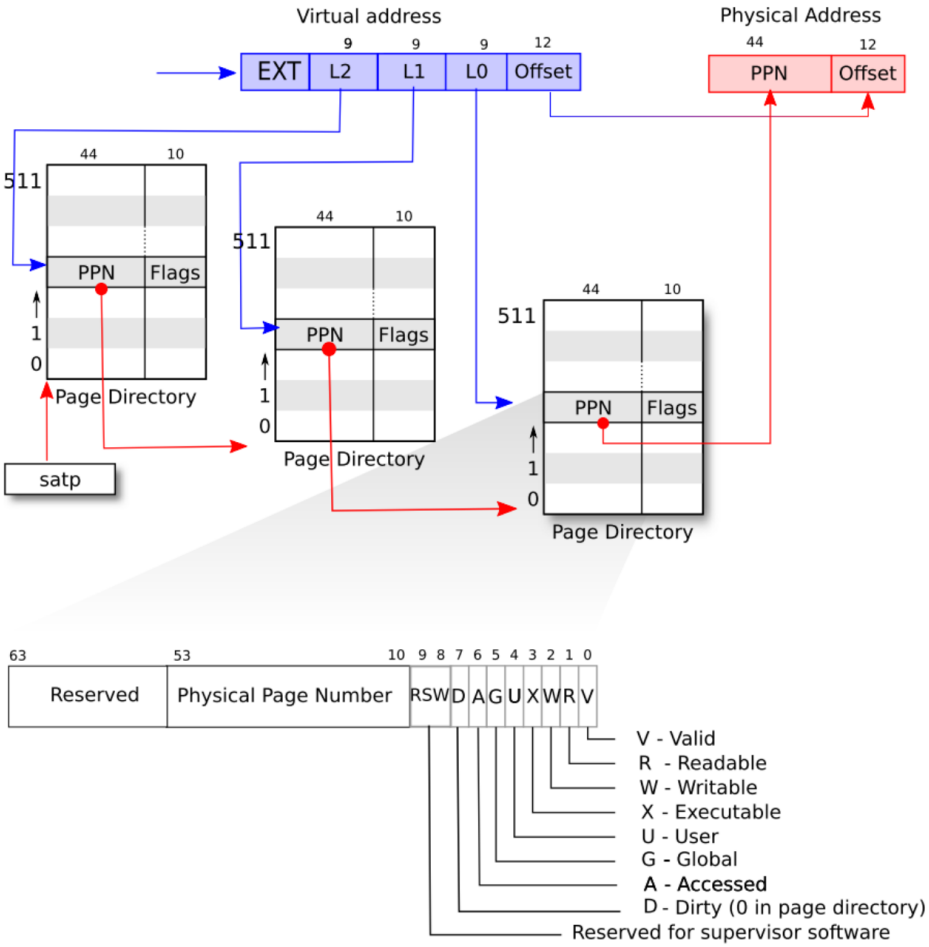


Figure 3.2: RISC-V address translation details.

内核必须将跟页表页面的物理地址写进satp寄存器，来让硬件使用页表。每个CPU有自己的satp。

一些术语的注意事项。物理存储器是指DRAM中的存储单元，一个物理内存的byte有一个地址，叫做物理地址。指令只使用虚拟地址，分页硬件将其转换为物理地址，接着送入DRAM硬件中读写数据。与物理内存和虚拟地址不同，虚拟内存不是一个物理对象，而是指内核提供的用于管理物理内存和虚拟地址的抽象和机制的集合。

Kernel address space

xv6位每个进程维护一个页表，描述每个进程的**用户地址空间**，以及一个描述**内核地址空间**的页表。内核配置其地址空间的分布，以允许自己访问**可预测的**(predictable)物理内存和各种硬件资源。下图展示了这种布局如何将内核虚拟地址映射到物理地址。

QEMU模拟了一个计算机，包含RAM，从物理地址0x80000000开始到至少0x86400000（xv6称其为PYSSTOP）。QEMU模拟也包括了IO设备，QEMU将设备接口作为位于物理地址空间0x80000000以下的memory-mapped控制寄存器公开给软件。内核可以通过读写这些特殊的物理地址与设备进行交互。

内核使用**直接映射**获取RAM和memory-mapped device registers。也就是说将资源映射到与物理地址**相等**的虚拟地址。比如内核自己在虚拟地址空间和物理地址空间中都被放置在KERNBASE=0x80000000上。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给子进程时，直接将该地址做为虚拟地址使用。

但有一些内核虚拟地址是没有使用直接映射的：

- trampoline page. 它被映射到内核虚拟地址空间的顶部，**用户页表具有相同的映射。**一个物理页面（包括trampoline代码）在内核虚拟地址空间中被映射了两次，一次在虚拟地址空间的顶部，一次是直接映射。
- kernel stack pages. 每个进程有自己的内核栈，内核栈被映射到较高的位置，以便xv6可以在他下面留下一个未映射的guard page。guard page的PTE是invalid的（PTE_V未设置），所以如果内核的内核栈溢出了，会产生一个异常。如果没有guard page，就会覆盖其他内核内存，导致错误的操作。

内核使用PTE_R和PTE_X权限，映射用于trampoline和kernel text的页面。即内核可以读写这些指令。内核使用PTE_R和PTE_W权限映射其他页面，以便能够读写这些页面的内存。guard pages被设置为invalid的。

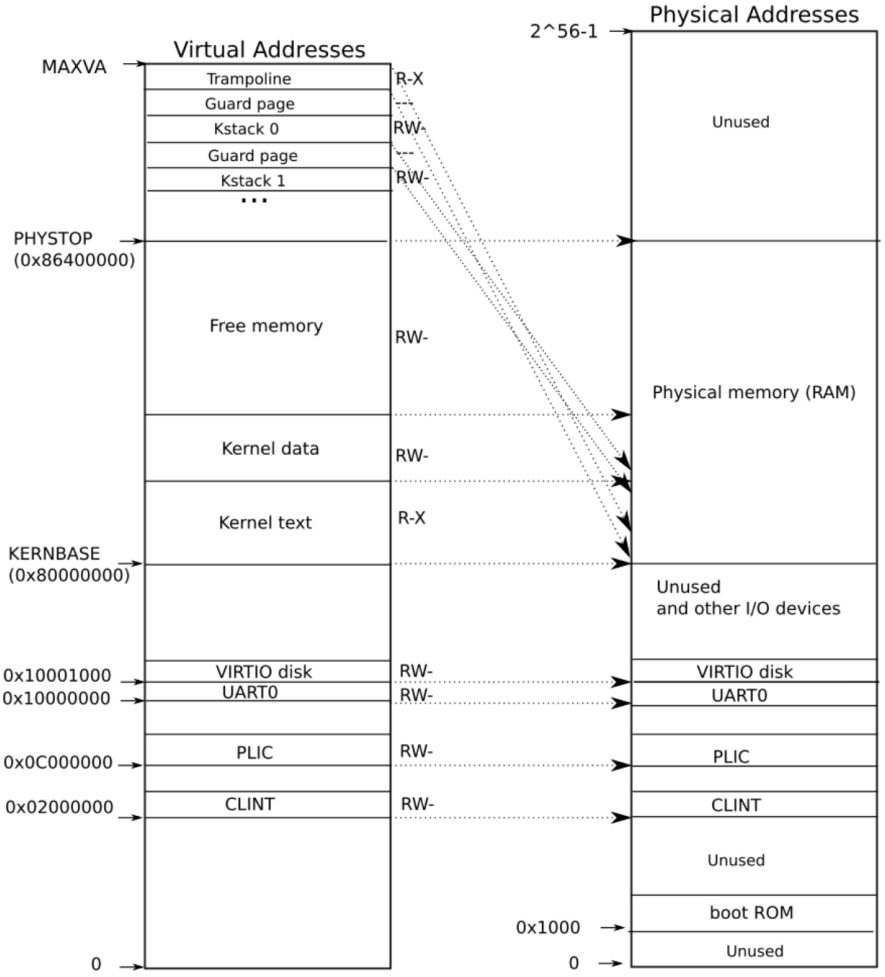


Figure 3.3: On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see.

Code: creating an address space

前情回顾：

在系统boot时的过程如下

kernel > C main.c > ...

```c
#include "types.h"
#include "param.h"
#include "memlayout.h"
#include "riscv.h"
#include "defs.h"

volatile static int started = 0;

// start() jumps here in supervisor mode on all CPUs.
void
main()
{
  if(cpuid() == 0){
    consoleinit();
    printfinit();
    printf("\n");
    printf("xv6 kernel is booting\n");
    printf("\n");
    kinit();         // physical page allocator
    kvminit();       // create kernel page table
    kvminithart();   // turn on paging
    procinit();      // process table
    trapinit();      // trap vectors
    trapinithart();  // install kernel trap vector
    plicinit();      // set up interrupt controller
    plicinithart();  // ask PLIC for device interrupts
    binit();         // buffer cache
    iinit();         // inode cache
    fileinit();      // file table
    virtio_disk_init(); // emulated hard disk
    userinit();      // first user process
    __sync_synchronize();
    started = 1;
  } else {
    while(started == 0)
      ;
    __sync_synchronize();
    printf("hart %d starting\n", cpuid());
    kvminithart();    // turn on paging
    trapinithart();   // install kernel trap vector
    plicinithart();   // ask PLIC for device interrupts
  }

  scheduler();
}
```

xv6中大部分用于操作地址空间和页表的代码在vm.c中。

核心数据结构是pagetable_t，实际上是一个指向RISC-V跟页表页面的指针，pagetable_t可以是内核页表，也可能是进程页表。

核心函数是walk，它为虚拟地址找到PTE；以及mappages，为新的映射设置PTEs。

以kvm开头的看书操作内核页表，以uvm开头的函数操作用户页表，其他函数两个都可以使用。

copyout和copyin将数据拷贝到作为系统调用参数提供的用户虚拟地址，或从其拷贝来。

在boot的过程中，main调用了kvminit来创建内核的页表。这个调用发生在xv6启用分页之前，因此地址直接引用物理内存。kvminit首先分配一个物理内存页来保存根页表页面（分配的内存在内核的结束位置和PHYSTOP之间），然后调用kvmmap来安装内核需要的转换，这些转换包括内核的指令和数据，物理内存直到PHYSTOP，以及设备对应的内存。

```
/*
 * create a direct-map page table for the kernel.
 */
void
kvminit()
{
  kernel_pagetable = (pagetable_t) kalloc();
  memset(kernel_pagetable, 0, PGSIZE);

  // uart registers
  kvmmap(UART0, UART0, PGSIZE, PTE_R | PTE_W);

  // virtio mmio disk interface
  kvmmap(VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);

  // CLINT
  kvmmap(CLINT, CLINT, 0x10000, PTE_R | PTE_W);

  // PLIC
  kvmmap(PLIC, PLIC, 0x400000, PTE_R | PTE_W);

  // map kernel text executable and read-only.
  kvmmap(KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R | PTE_X);

  // map kernel data and the physical RAM we'll make use of.
  kvmmap((uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext, PTE_R | PTE_W);

  // map the trampoline for trap entry/exit to
  // the highest virtual address in the kernel.
  kvmmap(TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);
}
```

kvmmap调用了mappages，将映射安装到页表中，将一系列虚拟地址映射到相应的物理地址范围。它以页为间隔，为范围内的每个虚拟地址分别执行此操作。对于每个要被映射的虚拟地址，mappages调用walk去查找该地址的PTE地址，然后初始化PTE以保存相关的物理代码，所需权限，和PET_V来标记PTE为有效。

疑问：为何mappages要以页为间隔。因为最底层的PTE对应了一个页面....。

walk模仿RISC-V分页硬件，寻找虚拟地址的PTE。walk使用每层的9位虚拟地址来查找下一层的页表或最终页的PTE。如果PTE不是valid，则所需的页面还没被分配；如果设置了alloc参数，walk将分配一个新的页表页面，并将其物理地址放在PTE中。它返回树中最低层的PTE的地址。

```c
// add a mapping to the kernel page table.
// only used when booting.
// does not flush TLB or enable paging.
void
kvmmap(uint64 va, uint64 pa, uint64 sz, int perm)
{
  if(mappages(kernel_pagetable, va, sz, pa, perm) != 0)
    panic("kvmmap");
}
```

```c
// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned. Returns 0 on success, -1 if walk() couldn't
// allocate a needed page-table page.
int
mappages(pagetable_t pagetable, uint64 va, uint64 size, uint64 pa, int perm)
{
  uint64 a, last;
  pte_t *pte;

  a = PGROUNDDOWN(va);
  last = PGROUNDDOWN(va + size - 1);
  for(;;){
    if((pte = walk(pagetable, a, 1)) == 0)
      return -1;
    if(*pte & PTE_V)
      panic("remap");
    *pte = PA2PTE(pa) | perm | PTE_V;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

```
// Return the address of the PTE in page table pagetable
// that corresponds to virtual address va.  If alloc!=0,
// create any required page-table pages.
//
// The risc-v Sv39 scheme has three levels of page-table
// pages. A page-table page contains 512 64-bit PTEs.
// A 64-bit virtual address is split into five fields:
//   39..63 -- must be zero.
//   30..38 -- 9 bits of level-2 index.
//   21..29 -- 9 bits of level-1 index.
//   12..20 -- 9 bits of level-0 index.
//    0..11 -- 12 bits of byte offset within the page.
pte_t *
walk(pagetable_t pagetable, uint64 va, int alloc)
{
  if(va >= MAXVA)
    panic("walk");

  for(int level = 2; level > 0; level--) {
    pte_t *pte = &pagetable[PX(level, va)];
    if(*pte & PTE_V) {
      pagetable = (pagetable_t)PTE2PA(*pte);
    } else {
      if(!alloc || (pagetable = (pde_t*)kalloc()) == 0)
        return 0;
      memset(pagetable, 0, PGSIZE);
      *pte = PA2PTE(pagetable) | PTE_V;
    }
  }
  return &pagetable[PX(0, va)];
}
```

上述代码依赖于物理内存直接映射到内核虚拟地址空间这个事实。例如当walk降低页表级别时，从PTE中提取下一层页表的（物理）地址，然后使用该地址作为下一层页表的虚拟地址。

main调用kvminithard来安装内核页表，它将根页表页面的物理地址写入寄存器satp。之后，CPU将使用内核页表转换地址，即启动了分页功能。由于内核使用恒等映射，下一条指令的当前虚拟地址将映射到正确的物理内存地址。

```
// Switch h/w page table register to the kernel's page table,
// and enable paging.
void
kvminithart()
{
  w_satp(MAKE_SATP(kernel_pagetable));
  sfence_vma();
}
```

```
// use riscv's sv39 page table scheme.
#define SATP_SV39 (8L << 60)

#define MAKE_SATP(pagetable) (SATP_SV39 | (((uint64)pagetable) >> 12))

// supervisor address translation and protection;
// holds the address of the page table.
static inline void
w_satp(uint64 x)
{
  asm volatile("csrw satp, %0" : : "r" (x));
}
```

```
// flush the TLB.
static inline void
sfence_vma()
{
  // the zero, zero means flush all TLB entries.
  asm volatile("sfence.vma zero, zero");
}
```

procinit函数（在proc.c中），在main函数中被调用，它为每个进程分配一个内核栈（xv6中最多有64个进程，所以开了64个kernel stack）。它将每个堆栈映射到KSTACK生成的虚拟地址，并且为invalid的stack-guard page留下了空间。kvmmap将映射的PTE添加到内核页表中，对kvminithart的调用将内核页表重新加载到satp中，以便硬件知道新的PTEs。

```c
// initialize the proc table at boot time.
void
procinit(void)
{
  struct proc *p;

  initlock(&pid_lock, "nextpid");
  for(p = proc; p < &proc[NPROC]; p++) {
      initlock(&p->lock, "proc");

      // Allocate a page for the process's kernel stack.
      // Map it high in memory, followed by an invalid
      // guard page.
      char *pa = kalloc();
      if(pa == 0)
        panic("kalloc");
      uint64 va = KSTACK((int) (p - proc));
      kvmmap(va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
      p->kstack = va;
  }
  kvminithart();
}
```

```c
// map the trampoline page to the highest address,
// in both user and kernel space.
#define TRAMPOLINE (MAXVA - PGSIZE)

// map kernel stacks beneath the trampoline,
// each surrounded by invalid guard pages.
#define KSTACK(p) (TRAMPOLINE - ((p)+1)* 2*PGSIZE)
```

每个RISC-V CPU在Translation Look-aside Buffer(TLB)中缓存页表项，xv6更改页表时，必须通知CPU使相应的缓存TLB项失效。若没有，那么在稍后的某个时刻，TLB可能会使用一个就的缓存映射，指向一个错误的物理页面，这个物理页面可能已经分配给了另一个进程。

RISC-V有一个指令sfence.vma，刷新目前CPU的TLB。xv6在重新加载satp寄存器后通过调用kvminithart使用sfence.vma，以及在返回用户空间之前在trampoline代码中切换到用户页表时使用sfence.vma。

Physical memory allocation

内核必须在运行时为页表、用户内存、内核栈和管道缓冲区分配和释放物理内存。

xv6使用内核末端和PHYSTOP之间的物理内存进行运行时分配。它一次分配和释放整个4096B的页面。它通过在页面本身中使用一个链表来跟中哪些页面是空闲的。分配内存是从链表中删除一个页面，释放内存是将释放的页面添加到列表中。

Code: Physical memory allocator

分配器的代码在kalloc.c中。分配器的数据结构是，包含可以用来分配的物理内存页的空闲链表。每个空闲页面的链表元素都是一个结构体*run*。

分配器将每个空闲页面的*run*结构体存储在空闲页面自身中，因为是空闲的，没有存储任何有用的信息。

空闲链表由一个spin lock保护。

链表和锁放在一个结构体中，为了明确锁保护这个结构体的区域。现在不考虑锁的获取和释放，在第六章将详细讨论。

```
struct run {
  struct run *next;
};

struct {
  struct spinlock lock;
  struct run *freelist;
} kmem;
```

main函数调用kinit来初始化分配器。

```c
void
kinit()
{
  initlock(&kmem.lock, "kmem");
  freerange(end, (void*)PHYSTOP);
}

void
freerange(void *pa_start, void *pa_end)
{
  char *p;
  p = (char*)PGROUNDUP((uint64)pa_start);
  for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE)
    kfree(p);
}

// Free the page of physical memory pointed at by v,
// which normally should have been returned by a
// call to kalloc().  (The exception is when
// initializing the allocator; see kinit above.)
void
kfree(void *pa)
{
  struct run *r;

  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(pa, 1, PGSIZE);

  r = (struct run*)pa;

  acquire(&kmem.lock);
  r->next = kmem.freelist;
  kmem.freelist = r;
  release(&kmem.lock);
}
```

```
// Allocate one 4096-byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
void *
kalloc(void)
{
  struct run *r;

  acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  release(&kmem.lock);

  if(r)
    memset((char*)r, 5, PGSIZE); // fill with junk
  return (void*)r;
}
```

kinit初始化空闲列表来保存内核结束位置到PHYSTOP之间的每一个页面。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。但是相反，xv6假设机器有128M的RAM。

kinit调用freearange，通过每页调用kfree将内存添加到空闲列表中。PTE只能引用在4096B边界上对齐（可以被4096整除）的物理地址，因此freearange使用PGROUNDUP保证只释放对齐的物理地址。分配器开始时没有内存可供管理，这些kfree的调用给了它一些可以管理的空闲空间。

分配器有时将地址视为整数，以便对其执行算数运算（例如freearange上对所有页面进行遍历），有时使用地址作为指针来读写内存（例如操作存储在每个页面中的run结构体）；地址的双重作用是分配器代码充满C类型强制转换的主要原因，另一个原因是释放和分配会改变内存的类型。

函数kfree首先将释放的内存中的每个字节设置为值1，这将导致使用内存的代码读取释放后的内存（使用“悬空引用”），会读取到垃圾而不是旧的有效的内容，这样程序崩溃得更快。然后kfree将页面放到空闲链表最前端，它将pa强制转换为指向struct run的指针，在r->next中记录空闲链表旧的头，然后将表头设置为r。kalloc删除并放回空闲列表中的第一个元素。

Process address space

每个进程都有一个单独的页表，当xv6在进程之间切换时，它还会更改页表。进程的用户内存从虚拟地址0开始，增长到MAXVA，原则上允许进程寻址256G内存。

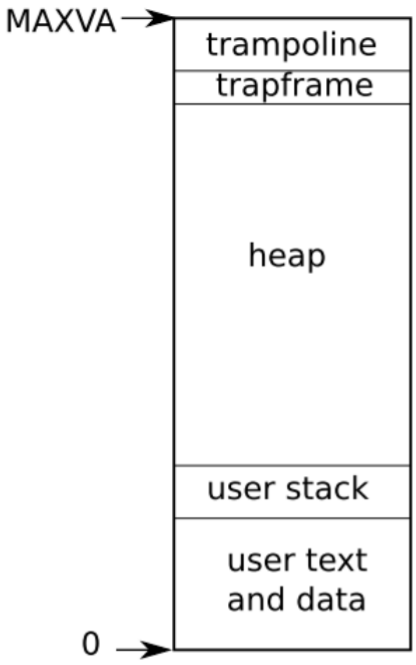


Figure 2.3: Layout of a process's virtual address space

```
// one beyond the highest possible virtual address.
// MAXVA is actually one bit less than the max allowed by
// Sv39, to avoid having to sign-extend virtual addresses
// that have the high bit set.
#define MAXVA (1L << (9 + 9 + 9 + 12 - 1))
```

当进程向xv6请求更多的用户内存时，xv6首先使用kalloc分配物理页面。然后将PTE添加到进程的页表中，这些页指向新的物理页。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将未使用的PTEs的PTE_V设置为空。

这里可以看到使用页表的优点。首先，不同进程的页表将用户地址转换到物理内存的不同页，这样每个进程都有私有的用户内存。其次，每个进程都将其内存视为具有从0开始的连续虚拟地址，而进程的物理内存可以是不连续的。第三，内核在用户地址空间的顶部用trampoline代码映射一个页面，从而在所有地址空间中显示一个物理内存页面。

下图详细展现了xv6中正在执行的进程的用户内存布局。**堆栈是一个单独的页面**，并显示由exec创建的初始内容。包含命令行参数的字符串，以及指向他们的指针数组位于堆栈的最顶端。这下面存的是可以让程序启动main函数的值，就像调用了函数main(argc, argv)。

为了检测用户堆栈溢出已分配的堆栈内存，xv6将invalid的guard page放在堆栈下方，如果用户堆栈溢出，并且进程试图使用堆栈以下的地址，由于guard page的PTE_V为invalid，所以硬件将生成一个page fault。

现在的操作系统可能会当溢出时自动为用户堆栈分配更多的内存。

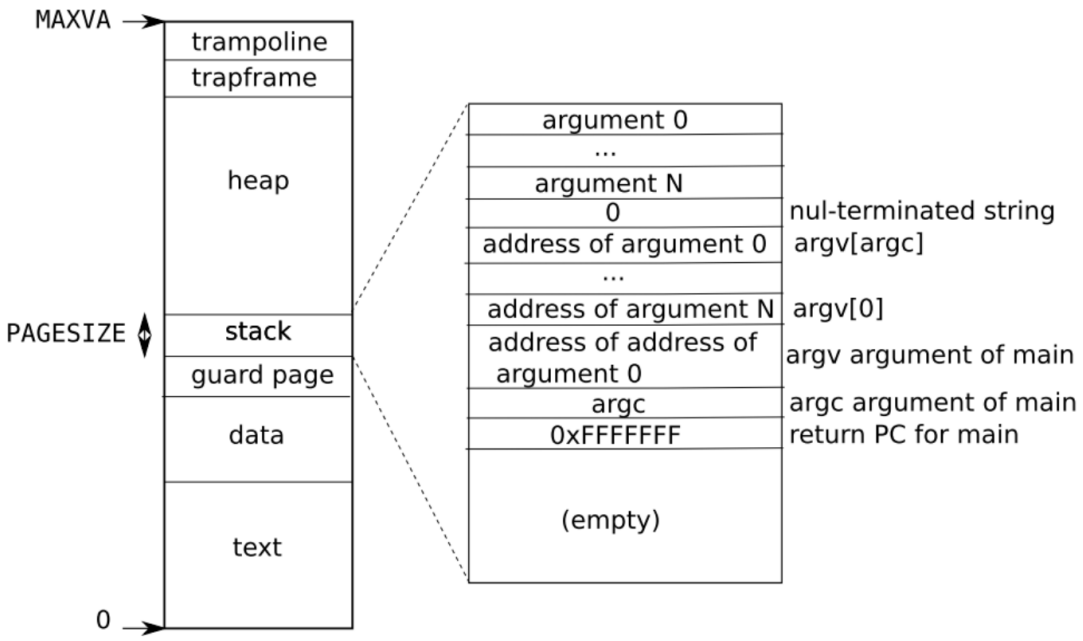


Figure 3.4: A process's user address space, with its initial stack.

Code: sbrk

sbrk是一个系统调用，用来减少或增加当前进程的内存。系统调用由函数growproc实现（位于kernel/proc.c中）。growproc调用uvmalloc或者uvmdealloc，取决于n是正还是负。

uvmalloc(位于kernel/vm.c中)使用kalloc分配物理内存，并且使用mappages将PTEs添加到用户页表中。如果不能分配足够的物理内存或者创建PTE的时候缺少物理内存，就用uvmdealloc释放分配的内存，然后返回0。

uvmdealloc调用uvmunmap(位于kernel/vm.c中)，使用walk寻找PTEs，取消map，并使用kfree释放物理内存。

```
uint64
sys_sbrk(void)
{
  int addr;
  int n;

  if(argint(0, &n) < 0)
    return -1;
  addr = myproc()->sz;
  if(growproc(n) < 0)
    return -1;
  return addr;
}
```

```c
// Grow or shrink user memory by n bytes.
// Return 0 on success, -1 on failure.
int
growproc(int n)
{
  uint sz;
  struct proc *p = myproc();

  sz = p->sz;
  if(n > 0){
    if((sz = uvmalloc(p->pagetable, sz, sz + n)) == 0) {
      return -1;
    }
  } else if(n < 0){
    sz = uvmdealloc(p->pagetable, sz, sz + n);
  }
  p->sz = sz;
  return 0;
}
```

```c
// Allocate PTEs and physical memory to grow process from oldsz to
// newsz, which need not be page aligned.  Returns new size or 0 on error.
uint64
uvmalloc(pagetable_t pagetable, uint64 oldsz, uint64 newsz)
{
  char *mem;
  uint64 a;

  if(newsz < oldsz)
    return oldsz;

  oldsz = PGROUNDUP(oldsz);
  for(a = oldsz; a < newsz; a += PGSIZE){
    mem = kalloc();
    if(mem == 0){
      uvmdealloc(pagetable, a, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);
    if(mappages(pagetable, a, PGSIZE, (uint64)mem, PTE_W|PTE_X|PTE_R|PTE_U) != 0){
      kfree(mem);
      uvmdealloc(pagetable, a, oldsz);
      return 0;
    }
  }
  return newsz;
}

// Deallocate user pages to bring the process size from oldsz to
// newsz.  oldsz and newsz need not be page-aligned, nor does newsz
// need to be less than oldsz.  oldsz can be larger than the actual
// process size.  Returns the new process size.
uint64
uvmdealloc(pagetable_t pagetable, uint64 oldsz, uint64 newsz)
{
  if(newsz >= oldsz)
    return oldsz;

  if(PGROUNDUP(newsz) < PGROUNDUP(oldsz)){
    int npages = (PGROUNDUP(oldsz) - PGROUNDUP(newsz)) / PGSIZE;
    uvmunmap(pagetable, PGROUNDUP(newsz), npages, 1);
  }

  return newsz;
}
```

```c
// Remove npages of mappings starting from va. va must be
// page-aligned. The mappings must exist.
// Optionally free the physical memory.
void
uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
{
  uint64 a;
  pte_t *pte;

  if((va % PGSIZE) != 0)
    panic("uvmunmap: not aligned");

  for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
    if((pte = walk(pagetable, a, 0)) == 0)
      panic("uvmunmap: walk");
    if((*pte & PTE_V) == 0)
      panic("uvmunmap: not mapped");
    if(PTE_FLAGS(*pte) == PTE_V)
      panic("uvmunmap: not a leaf");
    if(do_free){
      uint64 pa = PTE2PA(*pte);
      kfree((void*)pa);
    }
    *pte = 0;
  }
}
```

Code: exec

Exec是**创建地址空间的用户部分**的系统调用。它从存储在文件系统中的文件初始化地址空间的用户部分。Exec(位置在kernel/Exec.c中)使用namei(位置在kernel/Exec.c中)打开指定的二进制路径。然后读取ELF文件头。xv6应用程序使用广泛使用的ELF格式进行描述，定义在kernel/elf.h中。一个ELF二进制文件，由一个ELF头文件组成，struct elfhdr，接下来是一系列的program section headers，struct proghdr。每个proghdr描述一个必须加载到内存中的应用程序的section，Xv6程序只有一个program section header，但其他系统可能有单独的指令和数据section。

第一步是快速检查文件是否可能包含ELF二进制文件。ELF二进制文件以4字节的“magic number”0x7F、“E”、“L”、“F”或ELF_MAGIC(定义在kernel/elf.h)开始。如果ELF头具有正确的魔法数，exec假定二进制文件是格式良好的。

exec使用proc_pagetable函数分配一个没有用户映射的新页表，使用uvmalloc为每个ELF段分配内存，并使用loadseg将每个段加载到内存中。loadseg使用walkaddr查找所分配内存的物理地址，在该地址写入ELF段的每一页，并使用readi从文件中读取。

在系统启动后exec创建的第一个用户程序init的section header如下：

```
# objdump -p _init
user/_init:     file format elf64-littleriscv

Program Header:
    LOAD off    0x00000000000000b0 vaddr 0x0000000000000000
                                   paddr 0x0000000000000000 align 2**3
         filesz 0x0000000000000840 memsz 0x0000000000000858 flags rwx
   STACK off    0x0000000000000000 vaddr 0x0000000000000000
                                   paddr 0x0000000000000000 align 2**4
         filesz 0x0000000000000000 memsz 0x0000000000000000 flags rw-
```

应用程序的section header的filesz可能小于memsz，这表明它们之间的gap应该用0填充（对于未赋值的C全局变量），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分配了足够容纳2136字节的物理内存，但只从init文件中读取了2112个字节。

之后，exec分配并初始化用户堆栈，它只分配一个堆栈页面。exec将参数字符串复制到堆栈顶部，并在ustack中记录指向它们的指针。他在传递给main的argv列表的末尾放置一个空指针。ustack中的前三个entries分别是fake reutrn program counter, argc和argv pointer。

exec将一个不可访问的页面放在堆栈页面的下方，因此使用多个堆栈页面的程序将出错。这个不可访问的页面还允许exec处理过大的参数，这种情况下，exec用于将参数复制到堆栈的copyout函数将注意到目标页面不可访问，并返回-1.

在准备新的内存映像期间，如果exec检测到一个错误，比如一个invalid程序段，它将跳转到标签bad，释放新映像并返回-1。exec必须等待释放旧镜像，直到确定系统调用将成功：如果旧镜像已消失，系统调用不能返回-1给它。exec唯一的错误情况发生在创建映像期间，一旦映像完成，exec可以提交到新的页表并释放旧的页表。

Exec将ELF文件中的字节加载到ELF文件指定地址的内存中。用户或进程可以将他们想要的任何地址放入ELF文件中。因此exec是有风险的，因为ELF文件中的地址可能有意或无意地引用内核。一个“粗心的”内核的后果可能从崩溃到对内核隔离机制的恶意破坏(例如，安全漏洞)。Xv6执行许多检查以避免这些风险。例如,if(ph.Vaddr + ph.memsz < ph.Vaddr)检查和是否溢出64位整数。危险之处在于，用户可能使用指向用户选择的地址的ph.vaddr来构造ELF二进制文件，ph.memsz足够大，以至于总和溢出到0x1000，这看起来像一个有效值。在xv6的较老版本中，用户地址空间也包含内核(但在用户模式下不可读/写)，用户可以选择与内核内存对应的地址，从而将数据从ELF二进制文件复制到内核中。在xv6的RISC-V版本中，这是不可能发生的，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关键的检查，而真实的内核长期以来都有遗漏检查的历史，用户程序可以利用这些检查的缺失来获取内核权限。xv6很可能没有完全完成验证提供给内核的用户级数据的工作，恶意用户程序可能会利用这些数据来绕过xv6的隔离。

Real world

与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。大多数操作系统通过结合分页和页面错误异常，对分页的使用都比xv6复杂得多，我们将在第4章中讨论这一点。

由于内核使用了虚拟地址和物理地址之间的直接映射，并且假设在0x8000000地址有一个物理RAM，内核期望在这个地址上加载该地址，因此简化了Xv6。这种方法适用于QEMU，但在真正的硬件上却是个不好的设计;实际硬件将RAM和设备放置在不可预知的物理地址，因此(例如)在0x8000000处可能没有RAM，而xv6希望在0x8000000处能够存储内核。更好的内核设计利用页表将任意硬件物理内存布局转换为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不使用该特性。

在具有大量内存的机器上，使用RISC-V对“超级页面”的支持可能是有意义的。当物理内存较小时，小页面是有意义的，这样可以细粒度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，那么给它一个4兆字节的超级物理内存页就是一种浪费。较大的页面在具有大量RAM的机器上是有意义的，并且可以减少页表操作的开销。

xv6内核缺乏可以为小对象提供内存的类似malloc的分配器，这使内核无法使用需要动态分配的复杂数据结构。

如今，人们更关心速度而不是空间效率。此外，更精细的内核可能会分配许多不同大小的小块，而不是(如xv6)只分配4096字节的块;一个真正的内核分配器将需要处理小的分配和大的分配。

总结xv6与现实os的不同：

1. 对分页的使用更简单
2. 内核使用了虚拟地址和物理地址的直接映射，而且物理地址是固定的
3. xv6不支持物理地址级别的保护
4. 空间分配器不精细