xv6 代码阅读报告 Ch03 页表

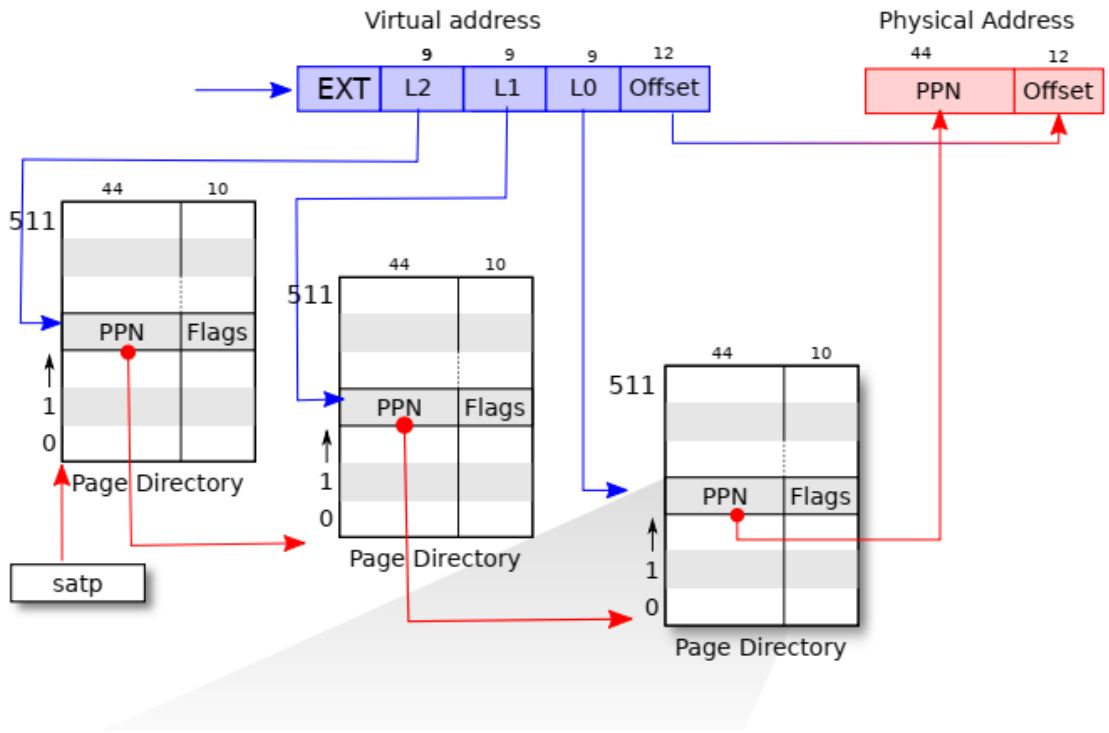
王集 2020012389

OS 需要隔离开不同的进程，最重要的一步就是隔离开不同进程之间的内存，让他们不能互相访问。实现这一点的机制主要有两种：分段和分页。
最好用的方式是分页：将物理内存划分为固定大小的块，进程占用其中的几个块。通过页表，将进程地址空间中的虚拟地址转换为真实内存的物理地址。

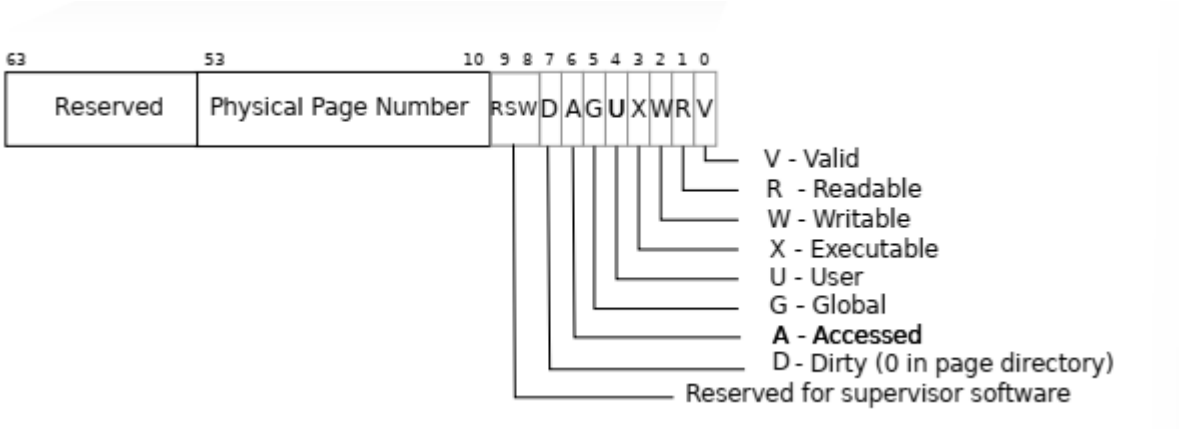
硬件支持

多级页表转换

一页的大小通常为4KB，这样的话，页表的体积就太大了。因此，xv6采用了三级页表的存储结构。根页表的首存储在satp寄存器里，地址转换由硬件完成。示意图如下：



每一个页表中的词条，除了物理页数（PPN）之外，还有若干标识（flags）：

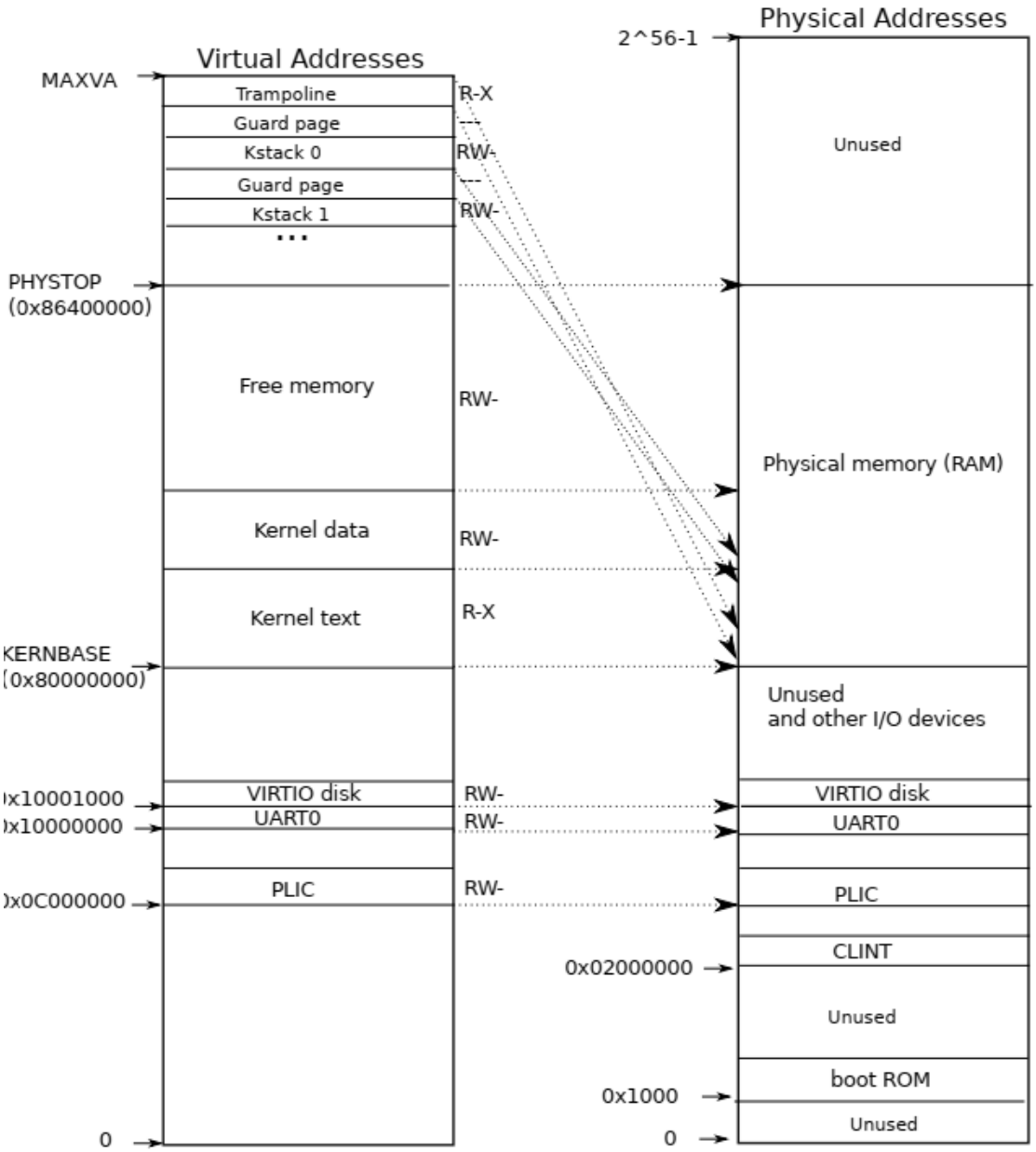


- V有效位 判断翻译是否有效 没有使用的空间是无效的，访问会trap到OS
- RWX保护位 指示这页可不可以读写运行
- U用户位：用户模式下能否访问这一页
- D脏位 指示页最近有没有被修改
- A访问位 指示页最近有没有被访问 决定是否留下，这对**页替换**很有用。

TLB

Translation Look-aside Buffer，是一个缓存区域，将已经访问过的词条放在里面，这样，下次需要的时候可以直接命中。以此减小页表转换的开支。

内核页表布局



- 0×80000000以下是外设。QEMU中，包含磁盘，IO设备，网络等等。xv6通过直接映射的方式访问这些外设。
- 内核数据，内核代码等等，大多是直接映射，只有两个特例
- trampoline page 所有地址空间的顶部都会映射到这里，然后再映射到物理内存中的代码。两次映射。
- kstack 内核栈 每个进程都有一个内核栈。把内核栈在虚拟内存里下面添上一个保护页，就可以防止栈溢出覆盖其他数据。因为保护页是虚拟内存里的，不是物理内存里的，所以保护页其实不占内存。这样非常便利。
-

创建第一个地址空间

- 页表的数据结构
  页表是一个 pagetable_t 类型的数据结构。pagetable_t 实际上只是 uint64 指针。
  所以页表仅仅是一些被链接起来的数组。我们的内核页表只是一个指向RISC-V根页表的指针。

- 第一个页表
  kernel/vm.c 中的 kvminit 函数创建了第一个页表。这个函数非常短，只有一句话。

```
void
kvminit(void)
{
  kernel_pagetable = kvmmake();
}
```

- kvmmake() 的功能是创建根页表。这个函数首先调用 kalloc() 在物理内存中为根页表分配空间，然后调用 kvmmap 函数把虚拟内存映射到物理内存里去。通过这个函数，内核页表获得了上一节【内核页表布局】中我曾写到过的的结构。

- kvmmap(pagetable_t kpgtbl, uint64 va, uint64 pa, uint64 sz, int perm) 的功能是在内核上建立起 虚拟地址 -- 物理地址 的映射关系。函数的形式变量 kpgtbl 代表内核页表，va 代表虚拟地址的开始，pa 代表物理地址的开始，sz 代表从va开始有sz大小的地址映射过去，perm代表这段地址上允许的操作（读/写/执行）

- mappages(pagetable_t pagetable, uint64 va, uint64 size, uint64 pa, int perm) 随即被调用。这个函数接受的参数与kvmmap一样。实际上kvmmap只是mappages的一个包装，处理了exception的情况。

因为这个函数非常重要，所以下面展开讲讲 mappages 的代码

```
int
mappages(pagetable_t pagetable, uint64 va, uint64 size, uint64 pa,
int perm)
{
  uint64 a, last;
  pte_t *pte;
```

```
  if(size == 0)
    panic("mappages: size");

  // PGROUNDDOWN() PGROUNDDOWN() 是两个用来取整的宏。因为地址是按页分配的，不可能把开始的物理地址放在某一页的中间。所以需要舍入取整。
  a = PGROUNDDOWN(va);
  last = PGROUNDDOWN(va + size - 1);
  for(;;){
    // 调用函数 walk 寻找 a（虚拟地址） 在页表中对应的PTE位置
    if((pte = walk(pagetable, a, 1)) == 0)
      return -1;
    if(*pte & PTE_V)
      panic("mappages: remap");
    // 若成功找到，则将物理地址标记上可用和读写后，存入PTE
    *pte = PA2PTE(pa) | perm | PTE_V;
    // 完成任务，跳出
    if(a == last)
      break;
    // 处理下一页的映射关系
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

- walk(pagetable_t pagetable, uint64 va, int alloc) 函数用来寻找va在页表中对应的PTE的位置。返回的是一个指向那个PTE的指针。这个函数也很重要，所以在下面展开讲讲。

```
pte_t *
walk(pagetable_t pagetable, uint64 va, int alloc)
{
  if(va >= MAXVA)
    panic("walk");

  // 一共三层，所以需要跳转两次
  for(int level = 2; level > 0; level--) {
    // PX 这个宏用来提取虚拟地址的第level级目录
    pte_t *pte = &pagetable[PX(level, va)];
    // 可以访问，那就直接跳转访问
    if(*pte & PTE_V) {
      pagetable = (pagetable_t)PTE2PA(*pte);
```

```
    } else {
    // 不然，分配内存空间
      if(!alloc || (pagetable = (pde_t*)kalloc()) == 0)
        return 0;
      memset(pagetable, 0, PGSIZE);
      *pte = PA2PTE(pagetable) | PTE_V;
    }
  }
  // 最后，把这个PTE的地址返回
  return &pagetable[PX(0, va)];
}
```

- 这样，我们成功建立了内核页表。下一步是要安装上页表，让页表机制正式启用。这是函数 kvminithart() 的作用。这个函数一共只有两句话：

```
    w_satp(MAKE_SATP(kernel_pagetable)); // 设置寄存器 satp 为内核页表的地址，页表机制正式启用！
    sfence_vma(); // 刷新缓存（TLB）
```

物理内存分配

xv6 利用链表记录空闲页面，一个页面是4096KB。链表的结点为struct run; run中只有一个元素run *next，是指向下一个元素的指针。内存的地址用run本身的地址表示。列表的表头是一个叫 kmem 的结构，其中存有一个自旋锁和一个freelist指针，指向第一个空闲的区域。

功能扩展的可能性：使用其他数据结构记录空闲页面（树）；像linux一样，实现大页面，例如2MB的页；Buddy Allocator

kernel/main.c 中，用 kinit() 函数初始化物理内存分配器。 kinit() 函数调用 freerange() 将内存添加到空闲列表中。

函数 freerange(void *pa_start, void *pa_end) 接受两个变量，释放从 pa_start 到 pa_end 之间的内存。前面的 kinit() 传入的两个分别为 end PHYSTOP，是整个DRAM的开始和结尾。这样，整个内存都被加入了 freelist。

```
void
freerange(void *pa_start, void *pa_end)
```

```
{
  char *p;
  // 毫无疑问，要取整
  p = (char*)PGROUNDUP((uint64)pa_start);
  for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE)
    kfree(p);
}
```

在释放的时候，freerange 调用了函数 kfree(void *pa) 这个函数的功能是释放pa处的这一页，之后将这一页插入freelist。

```
void
kfree(void *pa)
{
  struct run *r;

// 处理异常情况
  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >=
PHYSTOP)
    panic("kfree");

// 添上垃圾，以使非法访问更快崩溃
  memset(pa, 1, PGSIZE);

  r = (struct run*)pa;

// 插入空闲列表
  acquire(&kmem.lock);
  r->next = kmem.freelist;
  kmem.freelist = r;
  release(&kmem.lock);
}
```

最后，函数 kalloc() 返回一个指针，指向一个空闲的页，然后把这一页从 freelist中剔除。

进程地址空间

每一个进程都有一个自己独特的页表，因此进程之间获得了隔离的地址空间。
进程的地址空间结构如图所示：

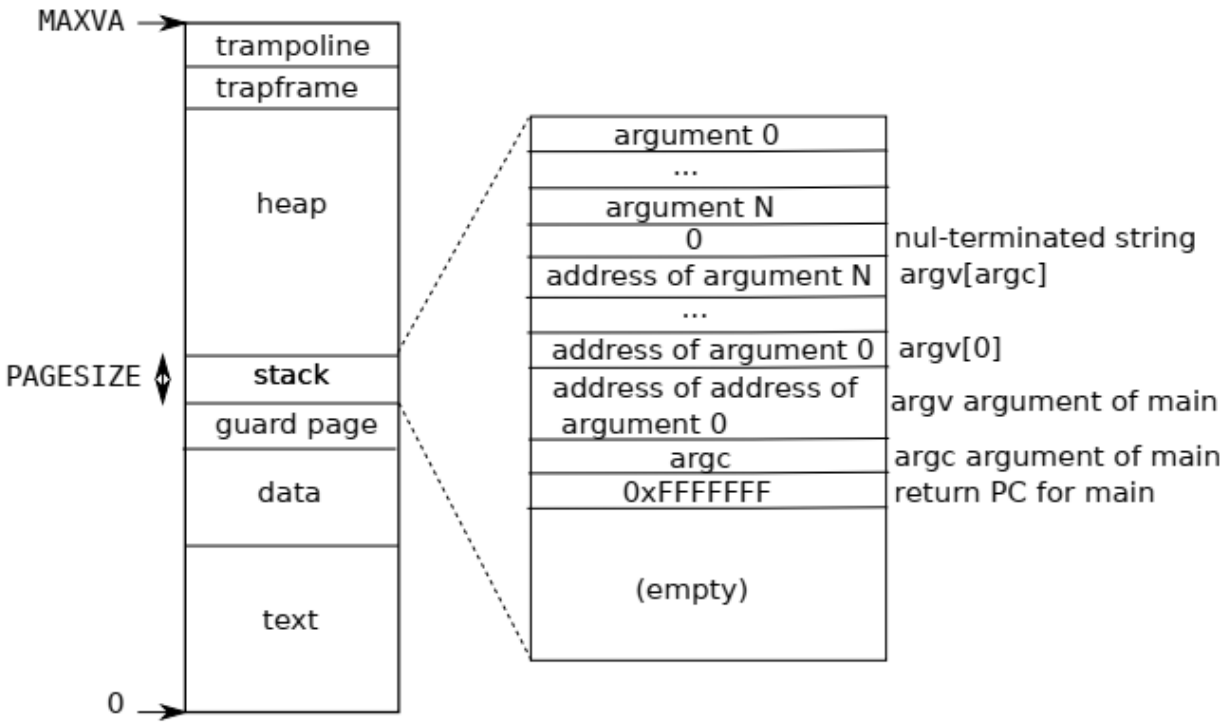


Figure 3.4: A process's user address space, with its initial stack.

解释一下这张图：

- MAXVA是一个常量，指定了进程地址空间的最大大小
- trampoline是一个所有地址空间都有的代码片段，位于相同的地址。是在trap的时候用的
- 栈区下面有一个虚拟的守卫区，用来检测栈溢出，并相应行动。
- heap data text区 很普通的堆，数据，代码区
- 每个进程的地址空间都是连续的，但实际内存却不一定。

进程利用 sbrk 系统调用来获得或缩小内存，sbrk 实现为 growproc 函数。
growproc(int n) 接受参数n，若n为正数则增加内存，若n为负数则减小内存。增加内存调用的是 uvmalloc 函数，而减小内存调用的是 uvmdealloc 函数。这两个函数都接受这三个参数：页表，旧大小，新大小。

uvmalloc 函数仅仅使用 kalloc 分配内存，如果遇到困难，就回退一页的分配，然后返回。
uvmdealloc 函数更加复杂一些，它必须先计算需要删除几页。然后，调用函数 uvmunmap 解除虚拟页与物理页之间的映射关系。

函数 uvmunmap 的作用是解除虚拟页与物理页之间的映射关系，程序员可以指定是否顺便清空物理页的内存。
接受三个参数，第一个是页表；第二个是虚拟地址开始的位置；第三个是移除的页数；第四个是标记是否释放内存。

```
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){
    // 找a对应的物理地址在哪里
    if((pte = walk(pagetable, a, 0)) == 0)
      panic("uvmunmap: walk");
    // 检查这个物理地址是否有效
    if((*pte & PTE_V) == 0)
      panic("uvmunmap: not mapped");
    // 只是标记了有效，其他没有任何标记，大概不是一个最低级的页表
    if(PTE_FLAGS(*pte) == PTE_V)
      panic("uvmunmap: not a leaf");
    // 可选：释放内存
    if(do_free){
      uint64 pa = PTE2PA(*pte);
      kfree((void*)pa);
    }
    *pte = 0;
  }
}
```

exec系统调用：创建进程地址空间

运行一个程序，需要先把这个程序的数据加载到内存里。xv6用exec()系统调用实现这一点。一个用户进程调用exec的时候，首先是trap进了内核模式，把自己的栈换成了kstack，效果是切换到了目前这个进程的内核控制进程。通过这个内核控制进程，程

序把自己的用户页表换成了新的页表。这样，当程序再度返回到用户模式的时候，它的页表就是新的页表了，所以它访问的就是新的程序。

```
int exec(char *path, char **argv);
```

这个函数接受path，argv参数，通过path找到可执行文件，argv是传给程序的参数。我们知道windows下的可执行文件是exe格式。而linux、unix、xv6中的可执行文件都为ELF格式。ELF文件由四部分组成，分别为ELF头(header)、程序头表(program header table)、节(section)和节头表(section header table)组成。
那么，exec究竟如何读取ELF文件呢?

首先，exec需要先验证给定的ELF二进制文件到底是不是一个可执行文件，也就是如下所示的片段。注意为了突出主干，我删去了一些部分。

```
  if(readi(ip, 0, (uint64)&elf, 0, sizeof(elf)) != sizeof(elf))
    goto bad;
  if(elf.magic != ELF_MAGIC)
    goto bad;
```

ELF文件的开头会有一组字符0x464C457FU。检测到了，才说明是一个完好的ELF文件，

接着，exec建立了一个没有任何映射的新页表，并且用uvmalloc为每一个ELF段分配页表。最后，用loadseg函数将ELF段加载到物理内存里。

```
if((pagetable = proc_pagetable(p)) == 0)
    goto bad;

  // Load program into memory.
  for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, 0, (uint64)&ph, off, sizeof(ph)) != sizeof(ph))
      goto bad;
    if(ph.type != ELF_PROG_LOAD)
      continue;
    if(ph.memsz < ph.filesz)
      goto bad;
    if(ph.vaddr + ph.memsz < ph.vaddr)
      goto bad;
```

```
    uint64 sz1;
    if((sz1 = uvmalloc(pagetable, sz, ph.vaddr + ph.memsz)) == 0)
      goto bad;
    sz = sz1;
    if((ph.vaddr % PGSIZE) != 0)
      goto bad;
    if(loadseg(pagetable, ph.vaddr, ip, ph.off, ph.filesz) < 0)
      goto bad;
  }
```

再下一步，分配用户栈。

```
sz = PGROUNDUP(sz);
  uint64 sz1;
  if((sz1 = uvmalloc(pagetable, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
  sz = sz1;
  uvmclear(pagetable, sz-2*PGSIZE);
  sp = sz;
  stackbase = sp - PGSIZE;
```

这里分配了两页，是把其中的一页当成保护页。实际上那一页不分配任何物理内存。

```
 // Push argument strings, prepare rest of stack in ustack.
  for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
      goto bad;
    sp -= strlen(argv[argc]) + 1;
    sp -= sp % 16; // riscv sp must be 16-byte aligned
    if(sp < stackbase)
      goto bad;
    if(copyout(pagetable, sp, argv[argc], strlen(argv[argc]) + 1) <
0)
      goto bad;
    ustack[argc] = sp;
  }
  ustack[argc] = 0;
// push the array of argv[] pointers.
  sp -= (argc+1) * sizeof(uint64);
  sp -= sp % 16;
```

```
    if(sp < stackbase)
      goto bad;
    if(copyout(pagetable, sp, (char *)ustack, (argc+1)*sizeof(uint64))
  < 0)
      goto bad;
```

这里，我们将参数压入栈中。最后，再把argv这个指针给压进去。这些都要调用copyout()函数，把内核模式下的数据拷贝到用户的物理内存里去。这些都做完以后，sp指针指向argv指针所在的地址。

现在，我们已经把ELF文件的内容和参数都装载到了内存里。因此，接下来要设置运行程序所必需的一些寄存器。

```
p->trapframe->a1 = sp;
p->pagetable = pagetable;
p->sz = sz;
p->trapframe->epc = elf.entry;  // initial program counter = main
p->trapframe->sp = sp; // initial stack pointer
return argc;
```

我们知道，一个C程序的开头是main函数，接受argc和argv[]两个参数。寄存器中的a0就对应第一个参数，a1对应第二个参数。从上面，我们知道，栈指针指向argv指针所在的地址。所以，要把p的trapframe中的a1寄存器设置为sp，也就是栈指针。

然后，需要设置p的页表为刚刚弄好的pagetable。接着，把p中的sz，epc（程序计数器，记录目前执行哪个命令），sp（栈指针）全部更新。

最后的return argc会把argc放到p的a0寄存器中。关于trap的细节不在这里展开，这是第四章的内容。从系统调用回到用户模式的时候，trampoline重新将储存在trapframe里的数据放回到寄存器里。这样，程序就会从elf.entry开始，有两个参数，argc就是刚刚return的a0，argv就是之前放到a1里的sp。

就这样，完全透明地，我们打开了一个新的程序。