哈爾濱工業大學（深圳）

HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

操作系统 (Operating System)

xv6原理简析：内存管理

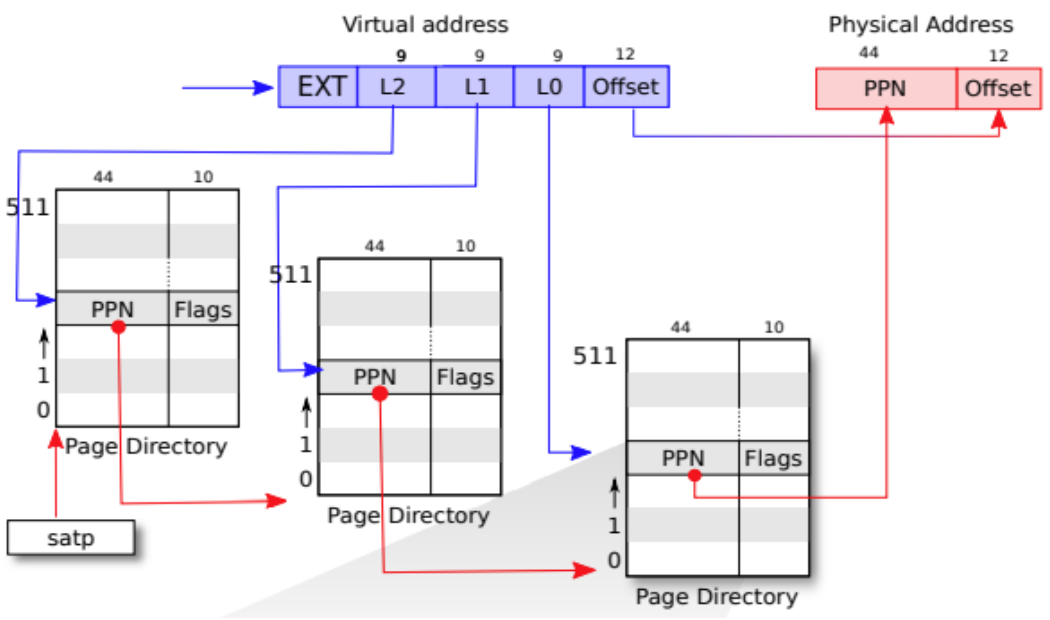
操作系统课程组
哈尔滨工业大学（深圳）

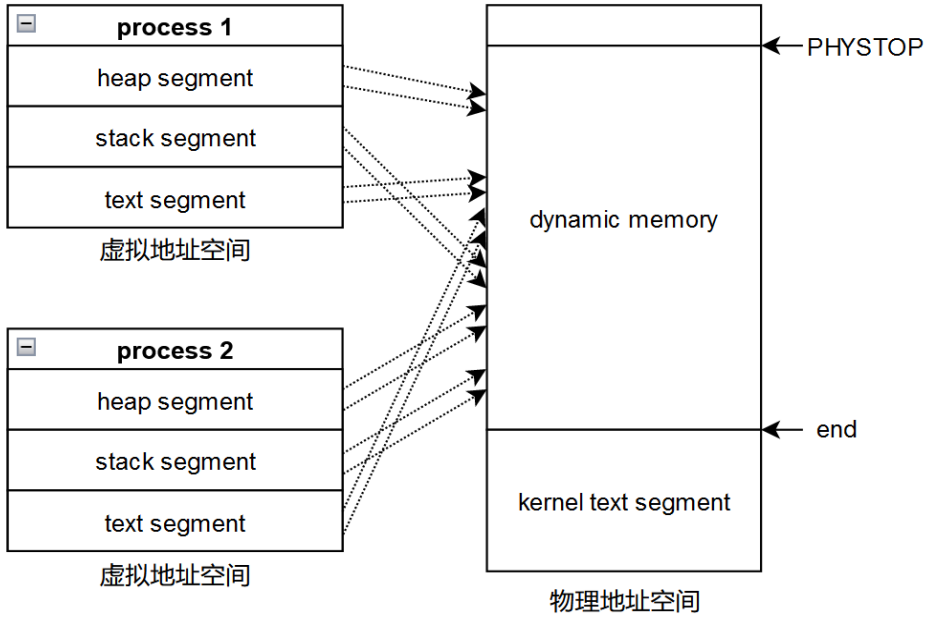
2022年秋季

Email：xiawen@hit.edu.cn

页式内存管理

- xv6使用分页方式管理用户程序的内存，每个用户进程拥有独立的虚拟地址空间，操作系统通过页表来维护虚拟地址到物理地址的映射。
- 为了提高速度，CPU会使用TLB来加速虚拟地址的转换。TLB是页表的cache，qemu也提供有这样的配置。
- xv6使用了sv39页表，使用三级地址转换，最大可管理512GB的空间。
- xv6不支持内存-磁盘页面交换，可能出现内存不足的情况，这样内存就分配失败。





内核的虚拟地址空间

- xv6内核态也运行在虚拟地址空间。
- 内核虚拟地址空间映射如右图(xv6-book-p32)
  - trampoline：用户态-内核态跳板
  - kstack ?：每个进程的内核栈空间
  - free memory：页分配器管理的地址空间
  - kernel data/text：数据段/代码段
  - KERNBASE：内核态基址
  - VIRTIO disk：磁盘IO操作地址
  - UART0：串口IO操作地址
  - PLIC：平台级中断控制器地址
  - CLINT：core-local中断控制器地址
- 为了方便管理物理内存，xv6的内核态页表具备所有可用物理内存的直接映射，可以直接访问物理内存。

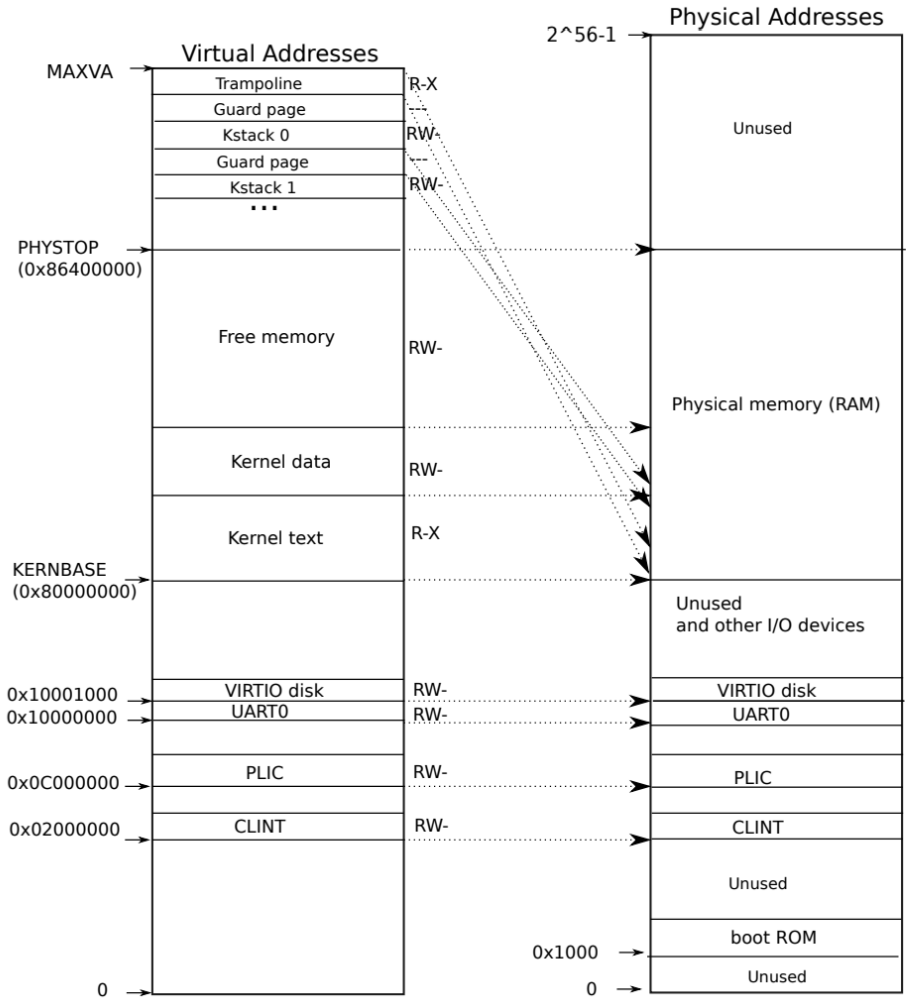


Figure 3.3: On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see.

用户的虚拟地址空间

- xv6的用户地址空间非常简单，如右图。所有页面都从空闲物理内存区域分配。
- 通常操作系统会把用户栈放置在比堆更高的位置，两者同时向中间扩展。xv6的用户栈只分配了一个页，放置在了比堆更低的位置。
- 程序初始化时堆没有分配任何空间。用户程序可以通过sbrk系统调用调整堆分配的空间，这会把新内存映射到页表中，也可以从页表中移除映射，释放内存。
- 用户虚拟地址空间和内核虚拟地址空间是独立的。因为两个状态运行在不同的页表上。
- lab4要求用户态和内核态共享地址空间，此时heap的最大地址必须小于内核态的最小地址。

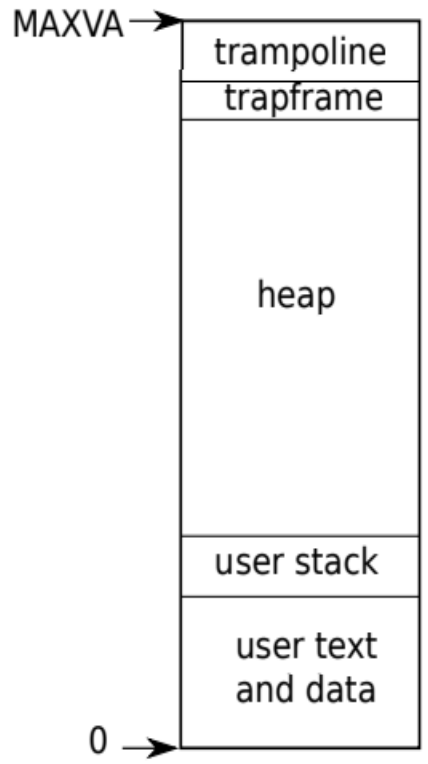
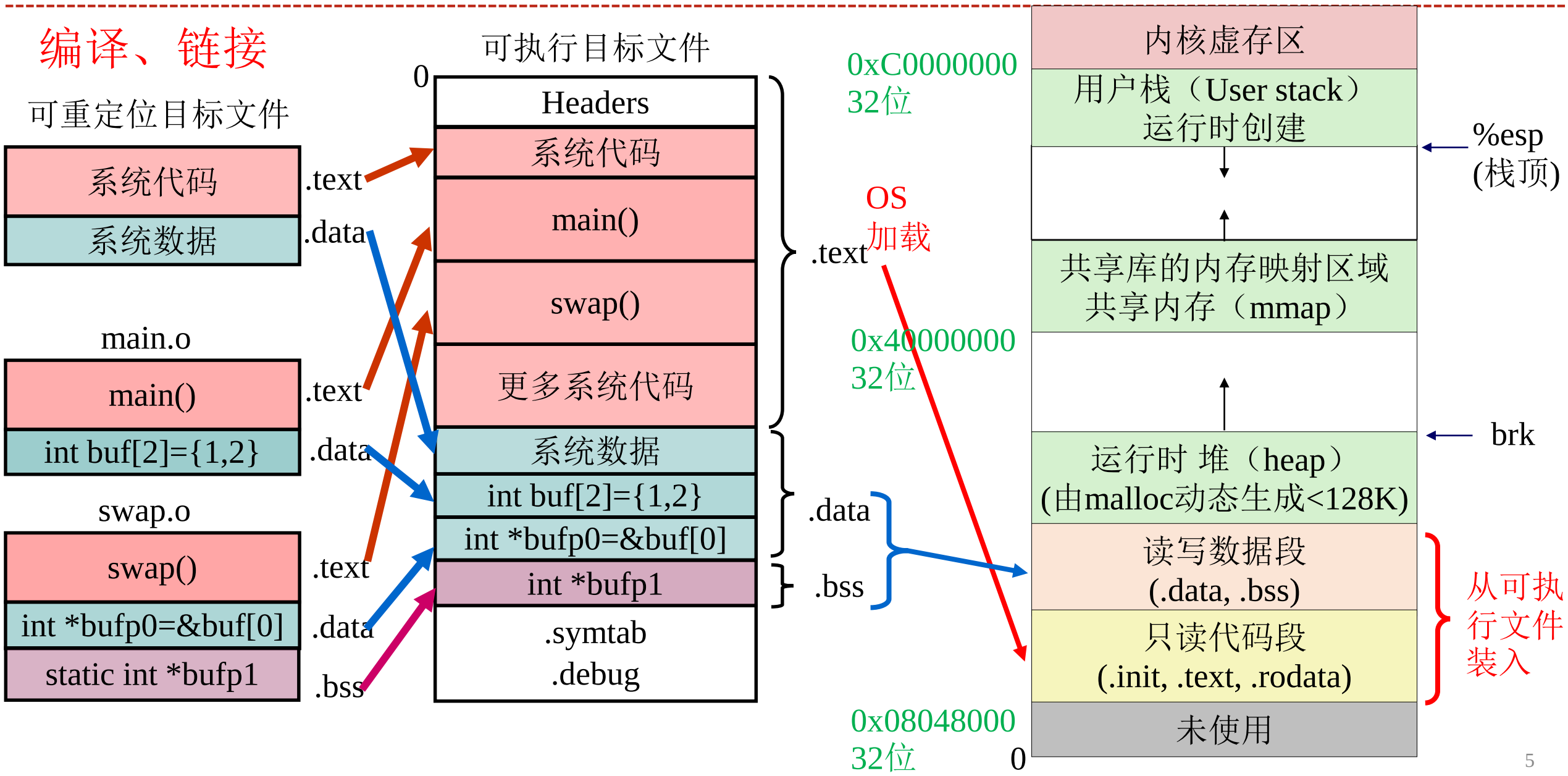


Figure 2.3: Layout of a process's virtual address space

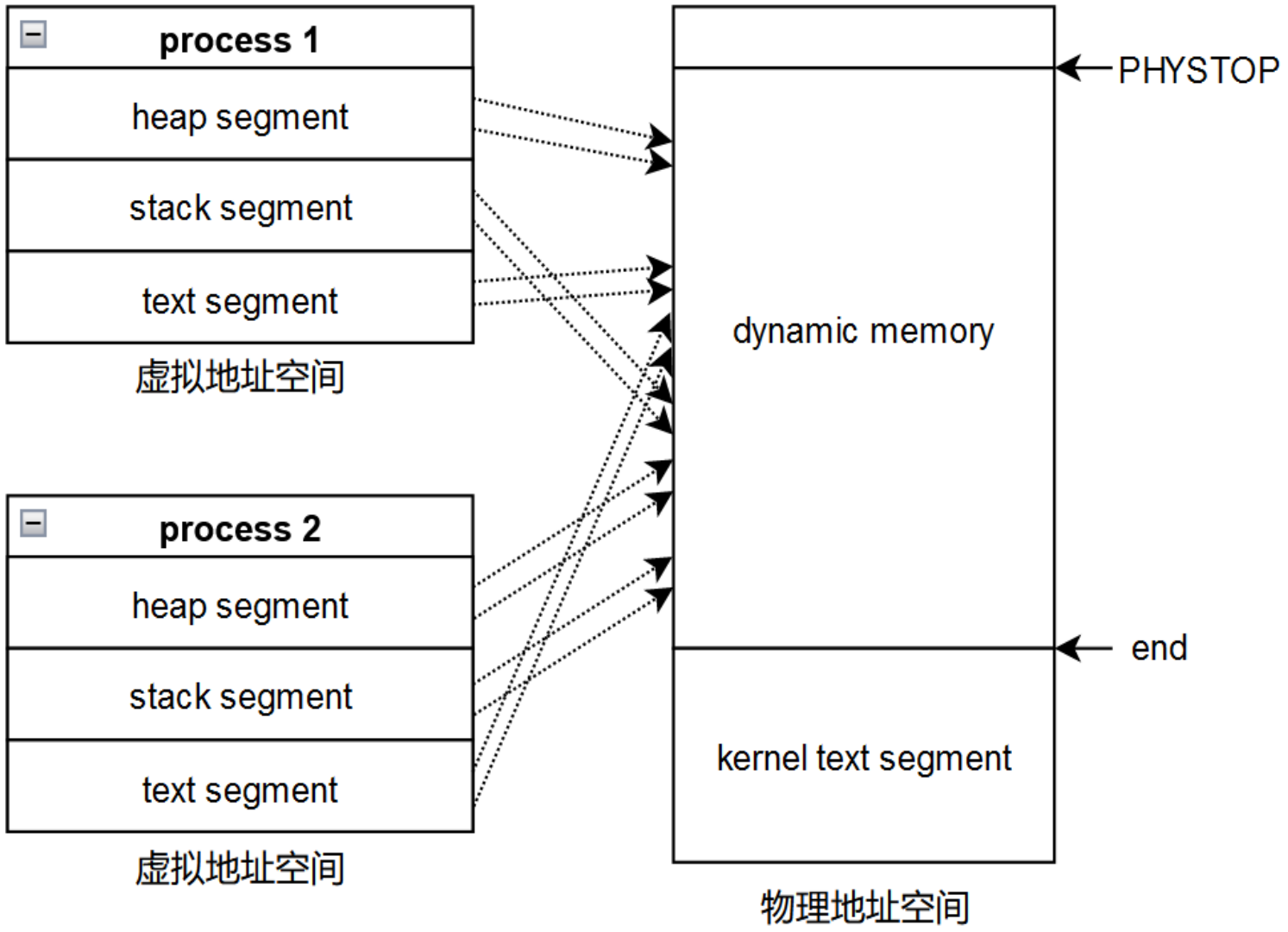
虚拟内存映像【回顾理论课】



xv6逻辑地址到物理地址的映射

- xv用采用的页式内存管理；

- **再次注意：xv6不支持内存-磁盘页面交换**，可能出现内存不足的情况。当内存不足时，xv6的页分配器会返回空指针，这让使用内存的函数进入错误路径，返回-1等错误码。



物理内存页的分配器相关实现

kernel/main.c

```
9   // start() jumps here in supervisor mode on all CPUs.
10  void
11  main()
12  {
13    if(cpuid() == 0){
14      consoleinit();
15      printfinit();
16      printf("\n");
17      printf("xv6 kernel is booting\n");
18      printf("\n");
19      kinit();         // physical page allocator
20      kvminit();       // create kernel page table
21      kvminithart();   // turn on paging
22      procinit();      // process table
23      trapinit();      // trap vectors
24      trapinithart();  // install kernel trap vector
25      plicinit();      // set up interrupt controller
26      plicinithart();  // ask PLIC for device interrupts
27      binit();         // buffer cache
28      iinit();         // inode cache
29      fileinit();      // file table
30      virtio_disk_init(); // emulated hard disk
31      userinit();      // first user process
```

kernel/kalloc.c

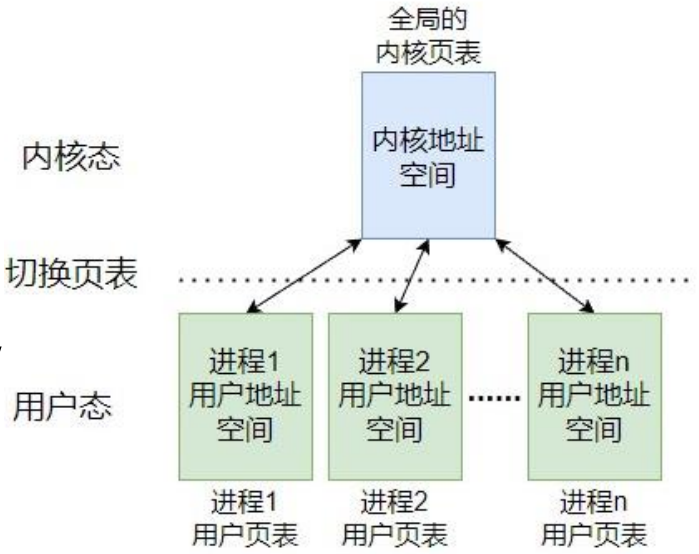
```
26  void
27  kinit()
28  {
29    initlock(&kmem.lock, "kmem");
30    freerange(pa_start: end, pa_end: (void*)PHYSTOP);
31  }
32
33  void
34  freerange(void *pa_start, void *pa_end)
35  {
36    char *p;
37    p = (char*)PGROUNDUP((uint64)pa_start);
38    for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE)
39      kfree(pa: p);
40  }
```

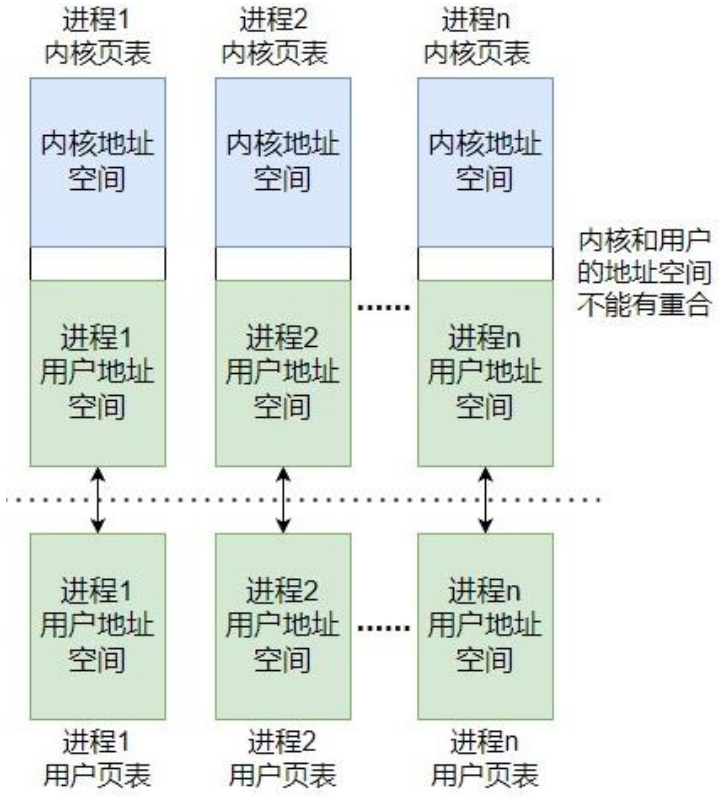
kernel/kalloc.c

```
68  void *
69  kalloc(void)
70  {
71    struct run *r;
72
73    acquire(&kmem.lock);
74    r = kmem.freelist;
75    if(r)
76      kmem.freelist = r->next;
77    release(&kmem.lock);
78
79    if(r)
80      memset((char*)r, 5, PGSIZE);
81    return (void*)r;
82  }
```

关于进程独立内核页表（Lab4）

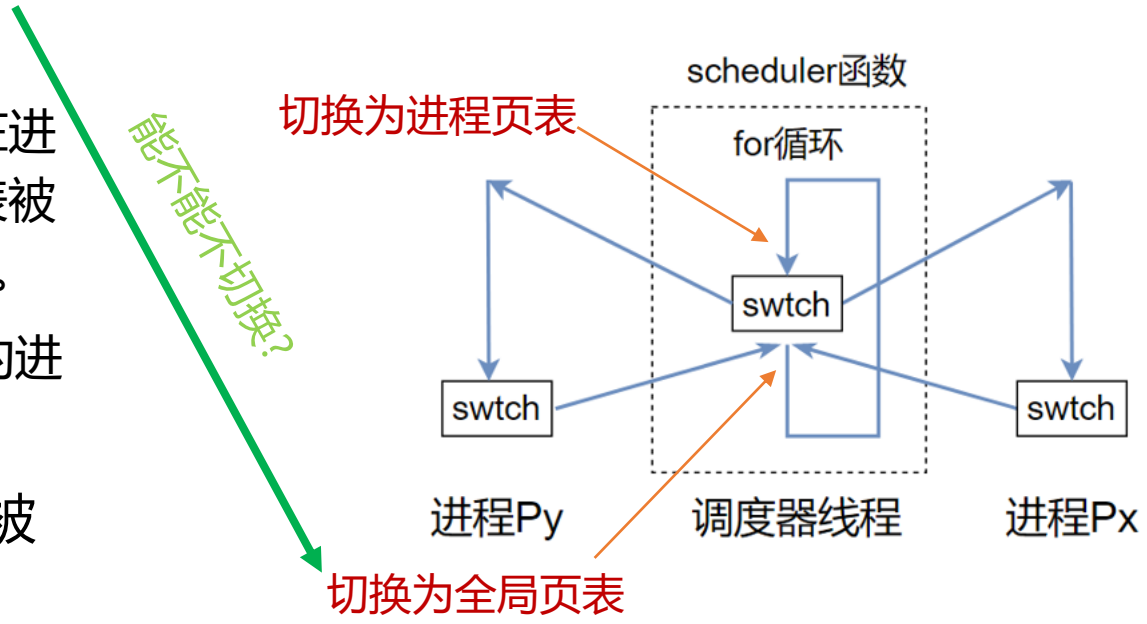
- 在xv6中（除了lab4），内核地址空间与用户地址空间**互相独立**，所有进程共享一个**全局内核页表**。
  - 内核态无法直接访问用户态虚拟地址，需要手动查找用户地址空间页表获取相关物理地址。
- lab4要求用户态和内核态**共享地址空间**，让xv6可以在内核态直接访问用户态指针，因此需要把用户地址空间映射到内核。
  - 每个进程都需要分配**独立的进程内核态页表**，把用户地址空间映射到该页表中。





进程独立内核页表的小细节（Lab4）

- scheduler调度器运行在哪个页表?
  - 运行在全局页表（main函数里面的初始页表，也是内核页表）
- scheduler函数为什么不运行在某个进程（比如进程Px）的内核页表上?
  - 因为进程内核页表可能会被回收：假设sheduler运行在进程Px的内核页表时，Px在另一个CPU上exit了，Px页表被回收，这会导致正在使用Px内核页表的scheduler崩溃。
  - 为了保证这一点，调度器切换到Px进程前要切换至Px的进程页表，Px进程切换回调度器后立刻切换回全局页表。
- 用户地址空间和内核地址空间不能重合。有些低地址被用来操作外设，用户不能占用这些地址。
  - 现代操作系统通常将整个内核放置在高位地址空间，并重新映射这些操作外设的低地址（加个偏移量），因此用户程序可以利用整个低地址空间。



内核独立页表的小细节（部分相关代码）

- 全局页表相关代码：main函数调用kvminit()

kernel/vm.c

```
21  void
22  kvminit()
23  {
24    kernel_pagetable = (pagetable_t) kalloc();
25    memset(kernel_pagetable, 0, PGSIZE);
26
27    // uart registers
28    kvmmap(va: UART0, pa: UART0, sz: PGSIZE, perm: PTE_R | PTE_W);
29
30    // virtio mmio disk interface
31    kvmmap(va: VIRTIO0, pa: VIRTIO0, sz: PGSIZE, perm: PTE_R | PTE_W);
32
33    // CLINT
34    kvmmap(va: CLINT, pa: CLINT, sz: 0x10000, perm: PTE_R | PTE_W);
35
36    // PLIC
37    kvmmap(va: PLIC, pa: PLIC, sz: 0x400000, perm: PTE_R | PTE_W);
38
39    // map kernel text executable and read-only.
40    kvmmap(va: KERNBASE, pa: KERNBASE, sz: (uint64)etext-KERNBASE, perm: PTE_R | PTE_X);
41
42    // map kernel data and the physical RAM we'll make use of.
43    kvmmap(va: (uint64)etext, pa: (uint64)etext, sz: PHYSTOP-(uint64)etext, perm: PTE_R | PTE_W);
44
45    // map the trampoline for trap entry/exit to
46    // the highest virtual address in the kernel.
47    kvmmap(va: TRAMPOLINE, pa: (uint64)trampoline, sz: PGSIZE, perm: PTE_R | PTE_X);
48  }
```

- fork操作里面创建一个空的页表

kernel/proc.c

```
157  pagetable_t
158  proc_pagetable(struct proc *p)
159  {
160    pagetable_t pagetable;
161
162    // An empty page table.
163    pagetable = uvmcreate();
164    if(pagetable == 0)
165      return 0;
166
167    // map the trampoline code (for system call return)
168    // at the highest user virtual address.
169    // only the supervisor uses it, on the way
170    // to/from user space, so not PTE_U.
171    if(mappages(pagetable, TRAMPOLINE, PGSIZE,
172                (uint64)trampoline, PTE_R | PTE_X) < 0){
173      uvmfree(pagetable, 0);
174      return 0;
175    }
176
177    // map the trapframe just below TRAMPOLINE, for trampoline.S.
178    if(mappages(pagetable, TRAPFRAME, PGSIZE,
179                (uint64)(p->trapframe), PTE_R | PTE_W) < 0){
180      uvmunmap(pagetable, TRAMPOLINE, 1, 0);
181      uvmfree(pagetable, 0);
182      return 0;
183    }
184
185    return pagetable;
186  }
```

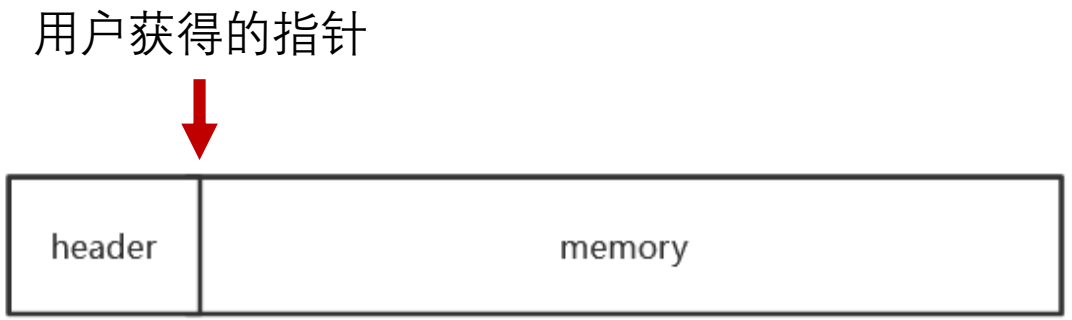
用户态的动态内存分配

- xv6内核态没有提供可变长度的动态内存分配机制（定长4KB）。
- xv6用户态提供了分配不定长内存的函数malloc和free，它是怎么做到的?
  - xv6用户态内存分配器基于链表来实现，采用首次适配方式。

```
 9   typedef long Align;
10
11   union header {
12     struct {
13       union header *ptr;
14       uint size;
15     } s;
16     Align x;
17   };
```

user/umalloc.c

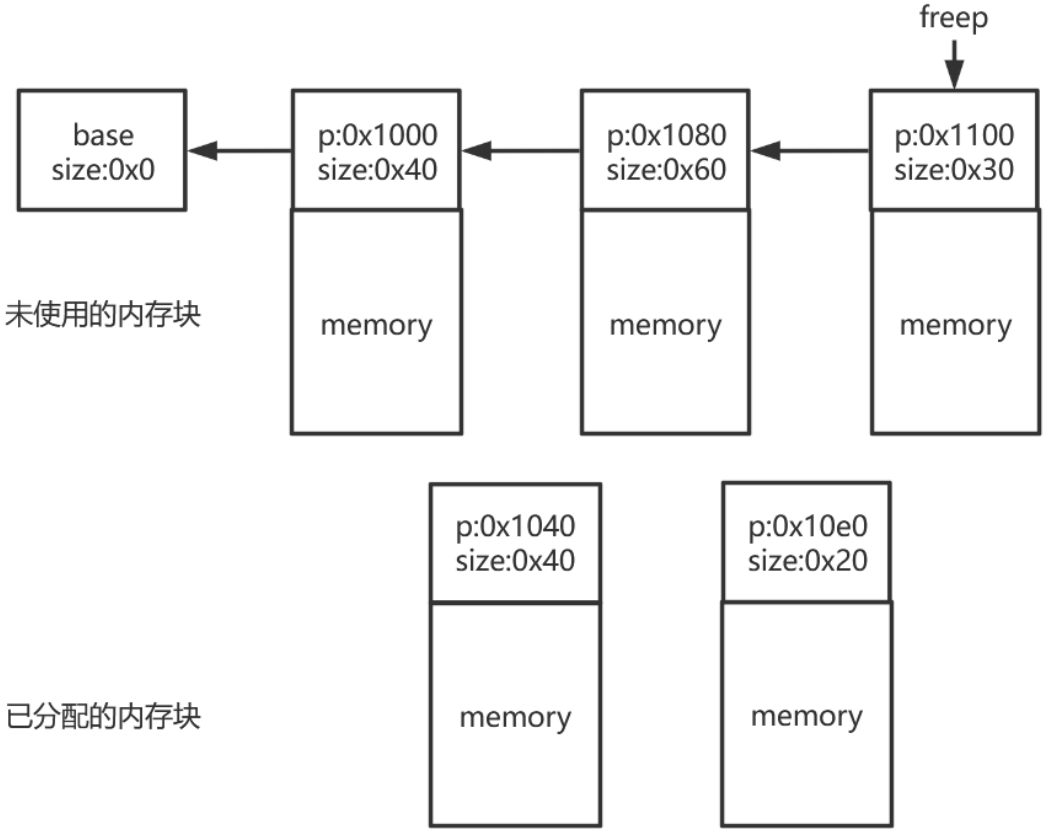
一个内存分配单元



它对应的虚拟内存块

用户态的动态内存分配

- 空闲的内存块按从小到大的顺序连成链表...
- malloc
  - 从freep开始遍历链表，查找第一个放得下的段（从高地址向低地址查找）
  - 如果找到了，从链表中取出这个段并切除内存块，返回偏移header后的地址
  - 如果找不到，用**sbrk系统调用**分配内存。
- free
  - 遍历链表，找到释放内存块在链表中的位置。
  - 检查当前内存块是否可以和前后内存块合并。
  - 如果不可合并就把内存块加入链表。
- 这个内存分配器速度很慢，很容易产生内存碎片。



动态内存分配

user/umalloc.c

```
63  void*
64  malloc(uint nbytes)
65  {
66    Header *p, *prevp;
67    uint nunits;
68
69    nunits = (nbytes + sizeof(Header) - 1)/sizeof(Header) + 1;
70    if((prevp = freep) == 0){
71      base.s.ptr = freep = prevp = &base;
72      base.s.size = 0;
73    }
74    for(p = prevp->s.ptr; ; prevp = p, p = p->s.ptr){
75      if(p->s.size >= nunits){
76        if(p->s.size == nunits)
77          prevp->s.ptr = p->s.ptr;
78        else {
79          p->s.size -= nunits;
80          p += p->s.size;
81          p->s.size = nunits;
82        }
83        freep = prevp;
84        return (void*)(p + 1);
85      }
86      if(p == freep)
87        if((p = morecore(nunits)) == 0)
88          return 0;
89    }
90  }
```

user/umalloc.c

```
24  void
25  free(void *ap)
26  {
27    Header *bp, *p;
28
29    bp = (Header*)ap - 1;
30    for(p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
31      if(p >= p->s.ptr && (bp > p || bp < p->s.ptr))
32        break;
33    if(bp + bp->s.size == p->s.ptr){
34      bp->s.size += p->s.ptr->s.size;
35      bp->s.ptr = p->s.ptr->s.ptr;
36    } else
37      bp->s.ptr = p->s.ptr;
38    if(p + p->s.size == bp){
39      p->s.size += bp->s.size;
40      p->s.ptr = bp->s.ptr;
41    } else
42      p->s.ptr = bp;
43    freep = p;
44  }
```

xv6虚拟地址映射的其它问题（一）

- 在实验一中许多人写了这样的函数...
  - xv6可以正常地printf。
  - linux/windows上程序直接崩溃了。

```
void func() {
    char* buf = "";
    buf[0] = '@';
    buf[1] = 0;
    printf(Format: "get %s\n", buf);
}
```

这里buf可不会在栈上分配空间！

- 为什么会这样?
  - 因为xv6的elf解析系统太过简陋，为所有页面赋予了RWX权限。
  - 现代操作系统会为页面赋予它应有的权限，buf处于只读段，当它被写入时会发生异常。

```
memset(mem, 0, PGSIZE);
if(mappages(pagetable, va: a, size: PGSIZE, pa: (uint64)mem, perm: PTE_W|PTE_X|PTE_R|PTE_U) != 0){
  kfree(mem);
  uvmdealloc(pagetable, oldsz: a, newsz: oldsz);
  return 0;
}
```

kernel/vm.c中的uvmalloc函数片段

xv6虚拟地址映射的其它问题（二）

- 还有另一种代码…
  - xv6居然可以从空指针里读出数据！
  - linux/windows上程序直接崩溃了。

```
void func2() {
    int* p = 0;
    int v = *p;
    printf(Format: "get %d\n", v);
}
```

- 为什么会这样？
  - xv6映射了从0开始直到最大程序虚拟地址的每一个页面，空指针也处于范围内。
  - 实验用的gcc编译器默认使用了位置无关方式编译（PIC选项），这时代码段是包括第0页的（可以用gdb看看函数的地址是不是小于0x1000），因此0地址必定被映射。
  - 现代操作系统会将使用PIC编译选项的程序的地址空间随机偏移（保证第0个页不会被映射），所以访问空指针会报错；但xv6没有这么做，出现了上述错误。

总结页

- xv6使用**页式内存管理**来管理用户进程的虚拟地址空间，以及内核地址空间
- xv6使用链表简单管理可用物理内存页（现代操作系统采用伙伴内存分配系统）
- xv6-lab4需要实现**进程独立的内核页表**，需要仔细处理页表切换的细节
- xv6用户态使用有序链表管理空闲的动态分配内存（malloc与free）
- xv6地址映射比较简单，容易出现奇怪的访存问题，比如可以访问空指针