Lab 3: page tables

1950787 杨鑫

实验目的：

- 熟悉操作系统中页表的概念，以及虚拟地址和物理地址等
- 熟悉页表结构信息，尝试改写内部页表结构以加深理解
- 通过编程实现一些页表机制的改进

实验步骤：

首先切换分支至 pgtbl branch，以获取本次实验的内容

同时清理文件，以获得纯净的初始文件系统

```
git checkout pgtbl
make clean
```

a. Print a page table

实验目的：

定义一个名为 vmprint() 的函数。 它应该接受一个 pagetable_t 参数，并以下面描述的格式打印该页表。 在返回 argc 之前的 exec.c 中插入 if(p->pid==1) vmprint(p->pagetable) 以打印第一个进程的页表。

现在，当您启动 xv6 时，它应该像这样打印输出，描述第一个进程在刚刚完成 exec() 初始化时的页表：

```
page table 0x0000000087f6e000
..0: pte 0x0000000021fda801 pa 0x0000000087f6a000
.. ..0: pte 0x0000000021fda401 pa 0x0000000087f69000
.. .. ..0: pte 0x0000000021fdac1f pa 0x0000000087f6b000
.. .. ..1: pte 0x0000000021fda00f pa 0x0000000087f68000
.. .. ..2: pte 0x0000000021fd9c1f pa 0x0000000087f67000
..255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
.. ..511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
.. .. ..510: pte 0x0000000021fdd807 pa 0x0000000087f76000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
```

实验分析：

此实验需要实现一个打印页表信息的函数。我们可以在 exec 处检测其 pid 是否为 1（即是否是第一个进程）来决定是否打印页表信息（即执行此函数）。

因为 xv6 的操作系统是采用的三级页表，所以要检测其 level，对应的级别打印对应的消息，总体上应该是递归函数的形式。

实验核心代码：

vm.c

```
int
vmprintchild(pagetable_t pagetable, int level)
{
  for(int i = 0; i < 512; i++){
    pte_t pte = pagetable[i];
    if(pte & PTE_V){
      uint64 child = PTE2PA(pte);
      if (level == 1) {
        printf("..%d: pte %p pa %p\n", i, pte, child);
        vmprintchild((pagetable_t)child, level + 1);
      } else if (level == 2) {
        printf(".. ..%d: pte %p pa %p\n", i, pte, child);
        vmprintchild((pagetable_t)child, level + 1);
      } else {
        printf(".. .. ..%d: pte %p pa %p\n", i, pte, child);
      }
    }
  }
  return 0;
}

int
vmprint(pagetable_t pagetable)
{
  printf("page table %p\n", pagetable);
  vmprintchild(pagetable, 1);
  return 0;
}
```

exec.c

```
if(p->pid==1){
    vmprint(p->pagetable);
  }
```

测试：

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
page table 0x0000000087f64000
..0: pte 0x0000000021fd8001 pa 0x0000000087f60000
.. ..0: pte 0x0000000021fd7c01 pa 0x0000000087f5f000
.. .. ..0: pte 0x0000000021fd841f pa 0x0000000087f61000
.. .. ..1: pte 0x0000000021fd780f pa 0x0000000087f5e000
.. .. ..2: pte 0x0000000021fd741f pa 0x0000000087f5d000
..255: pte 0x0000000021fd8c01 pa 0x0000000087f63000
.. ..511: pte 0x0000000021fd8801 pa 0x0000000087f62000
.. .. ..510: pte 0x0000000021fed807 pa 0x0000000087fb6000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
$
```

如图，在一运行此系统时，会打印出第一个进程的页表信息

评分：

```
$ make qemu-gdb
pte printout: OK (3.1s)
```

可以通过所有测试

b. A kernel page table per process

实验目的：

Xv6 有一个内核页表，每当它在内核中执行时都会使用它。内核页表直接映射到物理地址，因此内核虚拟地址 x 映射到物理地址 x。 Xv6 还为每个进程的用户地址空间提供了一个单独的页表，仅包含该进程的用户内存的映射，从虚拟地址零开始。因为内核页表不包含这些映射，所以用户地址在内核中是无效的。因此，当内核需要使用在系统调用中传递的用户指针（例如，传递给 write() 的缓冲区指针）时，内核必须首先将指针转换为物理地址。本节和下一节的目标是允许内核直接使用用户指针。

您的第一项工作是修改内核，以便每个进程在内核中执行时都使用自己的内核页表副本。修改struct proc为每个进程维护一个内核页表，并修改调度器在切换进程时切换内核页表。对于这一步，每个进程的内核页表应该与现有的全局内核页表相同。

实验分析：

该实验要求修改 proc 结构体，使得每个进程有一个内核页表。这就使得在初始化的时候不是使用公共的内核页表，而是自己创建一个（但是应当与全局内核页表相同）；在 schedule 的时候，保证执行期间使用的是进程内核页表；在 kvmpa 执行时要修改为进程的内核页表。最后，还要在 freeproc 里面增加释放内核页表的动作。

实验核心代码：

vm.c

```
pagetable_t
kvmnew()
{
  pagetable_t kernel_pagetable = (pagetable_t) kalloc();
  memset(kernel_pagetable, 0, PGSIZE);

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

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

  // PLIC
  mappages(kernel_pagetable, PLIC, 0x400000, PLIC, PTE_R | PTE_W);

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

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

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

  return kernel_pagetable;
}

/*
 * create a direct-map page table for the kernel.
 */
void
kvminit()
{
  kernel_pagetable = kvmnew();
  // CLINT
  mappages(kernel_pagetable, CLINT, 0x10000, CLINT, PTE_R | PTE_W);
}


void
uvmfree2(pagetable_t pagetable, uint64 va, uint page)
{
  if (page > 0) uvmunmap(pagetable, va, page, 1);
  freewalk(pagetable);
}
```

proc.c

```
// Look in the process table for an UNUSED proc.
// If found, initialize state required to run in the kernel,
// and return with p->lock held.
```

```
// If there are no free procs, or a memory allocation fails, return 0.
static struct proc*
allocproc(void)
{
  struct proc *p;

  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->state == UNUSED) {
      goto found;
    } else {
      release(&p->lock);
    }
  }
  return 0;

found:
  p->pid = allocpid();

  // Allocate a trapframe page.
  if((p->trapframe = (struct trapframe *)kalloc()) == 0){
    release(&p->lock);
    return 0;
  }

  // Allocate a kernel page table.
  p->kernal_pagetable = kvmnew();
  char* pa;
  if ((pa = kalloc()) == 0) panic("kalloc");
  mappages(p->kernal_pagetable, KSTACK((int)(p - proc)), PGSIZE, (uint64)pa,
PTE_R | PTE_W);
  p->kstack = KSTACK((int)(p - proc));

  // An empty user page table.
  p->pagetable = proc_pagetable(p);
  if(p->pagetable == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
  }

  // Set up new context to start executing at forkret,
  // which returns to user space.
  memset(&p->context, 0, sizeof(p->context));
  p->context.ra = (uint64)forkret;
  p->context.sp = p->kstack + PGSIZE;

  return p;
}

// free a proc structure and the data hanging from it,
// including user pages.
// p->lock must be held.
static void
freeproc(struct proc *p)
{
  if(p->trapframe)
    kfree((void*)p->trapframe);
```

```
  p->trapframe = 0;
  if(p->kernal_pagetable)
    proc_freekernalpagetable(p->kernal_pagetable, p->kstack, p->sz);
  p->kernal_pagetable = 0;
  if(p->pagetable)
    proc_freepagetable(p->pagetable, p->sz);
  p->pagetable = 0;
  p->sz = 0;
  p->pid = 0;
  p->parent = 0;
  p->name[0] = 0;
  p->chan = 0;
  p->killed = 0;
  p->xstate = 0;
  p->state = UNUSED;
}


// free a proc structure and the data hanging from it,
// including user pages.
// p->lock must be held.
static void
freeproc(struct proc *p)
{
  if(p->trapframe)
    kfree((void*)p->trapframe);
  p->trapframe = 0;
  if(p->kernal_pagetable)
    proc_freekernalpagetable(p->kernal_pagetable, p->kstack, p->sz);
  p->kernal_pagetable = 0;
  if(p->pagetable)
    proc_freepagetable(p->pagetable, p->sz);
  p->pagetable = 0;
  p->sz = 0;
  p->pid = 0;
  p->parent = 0;
  p->name[0] = 0;
  p->chan = 0;
  p->killed = 0;
  p->xstate = 0;
  p->state = UNUSED;
}

// Free a process's page table, and free the
// physical memory it refers to.
void
proc_freepagetable(pagetable_t pagetable, uint64 sz)
{
  uvmunmap(pagetable, TRAMPOLINE, 1, 0);
  uvmunmap(pagetable, TRAPFRAME, 1, 0);
  uvmfree(pagetable, sz);
}


// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run.
```

```
//  - swtch to start running that process.
//  - eventually that process transfers control
//    via swtch back to the scheduler.
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();

  c->proc = 0;
  for(;;){
    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();

    int found = 0;
    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        // Switch to chosen process.  It is the process's job
        // to release its lock and then reacquire it
        // before jumping back to us.
        p->state = RUNNING;
        c->proc = p;
        w_satp(MAKE_SATP(p->kernal_pagetable));
        sfence_vma();
        swtch(&c->context, &p->context);
        kvminithart();

        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0;

        found = 1;
      }
      release(&p->lock);
    }
#if !defined (LAB_FS)
    if(found == 0) {
      intr_on();
      asm volatile("wfi");
    }
#else
    ;
#endif
  }
}
```

评分：

```
init: starting sh
$ usertests
usertests starting
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test truncate1: OK
test truncate2: OK
test truncate3: OK
test reparent2: OK
test pgbug: OK
test sbrkbugs: usertrap(): unexpected scause 0x000000000000000c pid=3234
            sepc=0x0000000000005406 stval=0x0000000000005406
usertrap(): unexpected scause 0x000000000000000c pid=3235
            sepc=0x0000000000005406 stval=0x0000000000005406
OK
test badarg: OK
test reparent: OK
test twochildren: OK
test forkfork: OK
test forkforkfork: OK
test argptest: OK
```

```
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
```

可以通过所有测试

c. Simplify `copyin/copyinstr`

实验目的:

内核的 copyin 函数读取用户指针指向的内存。它通过将它们转换为物理地址来实现这一点，内核可以直接解析引用。它通过在软件中遍历进程页表来执行此转换。您在这部分实验中的工作是将用户映射添加到每个进程的内核页表（在上一节中创建），从而允许 copyin（和相关的字符串函数 copyinstr）直接取消引用用户指针。
将 kernel/vm.c 中的 copyin 主体替换为对 copyin_new 的调用（在 kernel/vmcopyin.c 中定义）；对

copyinstr 和 copyinstr_new 执行相同的操作。将用户地址的映射添加到每个进程的内核页表，以便 copyin_new 和 copyinstr_new 工作。
该方案依赖于用户虚拟地址范围，不与内核用于其自己的指令和数据的虚拟地址范围重叠。 Xv6 使用从零开始的虚拟地址作为用户地址空间，幸运的是内核的内存从更高的地址开始。但是，这种方案确实将用户进程的最大大小限制为小于内核的最低虚拟地址。内核启动后，该地址是 xv6 中的 0xC000000，即 PLIC 寄存器的地址；参见 kernel/vm.c、kernel/memlayout.h 中的 kvminit() 以及文中的图 3-4。您需要修改 xv6 以防止用户进程变得大于 PLIC 地址。

实验分析：

在上一个实验中，我们已经为每个用户进程添加了他们自己的内核页表，但是内核页表不能直接使用用户地址，因为内核页表中不含有用户区的地址映射关系。所以我们就要为进程的内核页表添加用户地址的映射。

在 userinit （用户进程初始化时）将进程页表的地址映射复制一份给内核页表；在 fork 时也要复制这份映射给子进程的内核页表；在 exec 处先取消之前的内核页表映射，然后将新建立的进程内核页表建立新的映射；在 sys_sbrk 处，内存扩张或缩小时，也要相应的更改进程内核页表的映射信息；最后是 freeproc 处要取消进程内核页表的映射。

最后，还需要将原来使用的 copyin 等替换为相应的不需要之前那样遍历寻找页表的 copyin_new 函数。

实验核心代码：

vm.c

```
// Copy from user to kernel.
// Copy len bytes to dst from virtual address srcva in a given page table.
// Return 0 on success, -1 on error.
int
copyin(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len)
{
  return copyin_new(pagetable, dst, srcva, len);
}

// Copy a null-terminated string from user to kernel.
// Copy bytes to dst from virtual address srcva in a given page table,
// until a '\0', or max.
// Return 0 on success, -1 on error.
int
copyinstr(pagetable_t pagetable, char *dst, uint64 srcva, uint64 max)
{
  return copyinstr_new(pagetable, dst, srcva, max);
}
```

proc.c

```
// Set up first user process.
void
userinit(void)
{
  struct proc *p;
  pte_t *pte, *kernelPte;
```

```
  p = allocproc();
  initproc = p;

  // allocate one user page and copy init's instructions
  // and data into it.
  uvminit(p->pagetable, initcode, sizeof(initcode));
  p->sz = PGSIZE;

  //将进程页表的mapping，复制一份到进程内核页表
  pte = walk(p->pagetable, 0, 0);
  kernelPte = walk(p->kernal_pagetable, 0, 1);
  *kernelPte = (*pte) & ~PTE_U;

  // prepare for the very first "return" from kernel to user.
  p->trapframe->epc = 0;      // user program counter
  p->trapframe->sp = PGSIZE;  // user stack pointer

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  p->state = RUNNABLE;

  release(&p->lock);
}
```

exec.c

```
  uvmunmap(p->kernal_pagetable, 0, PGROUNDUP(oldsz) / PGSIZE, 0);
  for (j = 0; j < sz; j += PGSIZE){
    pte = walk(pagetable, j, 0);
    kernelPte = walk(p->kernal_pagetable, j, 1);
    *kernelPte = (*pte) & ~PTE_U;
  }
```

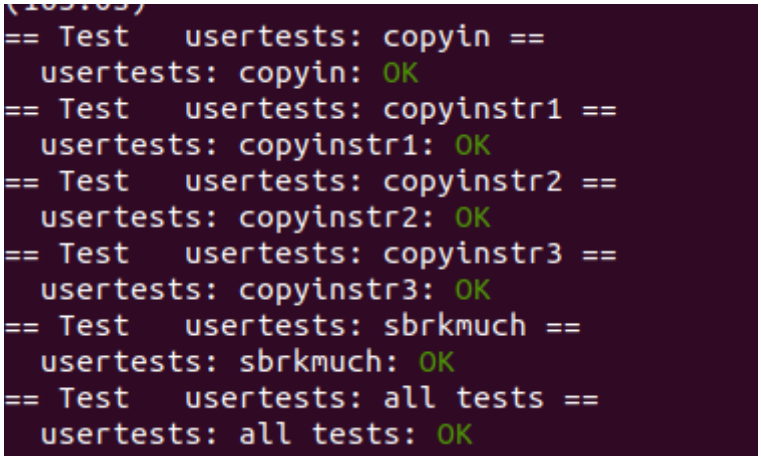
sysproc.c

```
uint64
sys_sbrk(void)
{
  int addr;
  int n, j;
  struct proc *p = myproc();
  pte_t *pte, *kernelPte;

  if(argint(0, &n) < 0)
    return -1;
  addr = p->sz;
  if (addr + n >= PLIC){
    return -1;
  }
  if(growproc(n) < 0)
    return -1;
  if (n > 0){
    //将进程页表的mapping，复制一份到进程内核页表
    for (j = addr; j < addr + n; j += PGSIZE){
```

```
      pte = walk(p->pagetable, j, 0);
      kernelPte = walk(p->kernal_pagetable, j, 1);
      *kernelPte = (*pte) & ~PTE_U;
    }
  }else {
    for (j = addr - PGSIZE; j >= addr + n; j -= PGSIZE){
      uvmunmap(p->kernal_pagetable, j, 1, 0);
    }
  }
  return addr;
}
```

评分



可以通过所有测试

实验评分：

按照实验要求对本次实验的所有小实验进行评分，结果如图：

```
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (3.1s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test count copyin ==
$ make qemu-gdb
count copyin: OK (0.7s)
== Test usertests ==
$ make qemu-gdb
(158.5s)
== Test   usertests: copyin ==
  usertests: copyin: OK
== Test   usertests: copyinstr1 ==
  usertests: copyinstr1: OK
== Test   usertests: copyinstr2 ==
  usertests: copyinstr2: OK
== Test   usertests: copyinstr3 ==
  usertests: copyinstr3: OK
== Test   usertests: sbrkmuch ==
  usertests: sbrkmuch: OK
== Test   usertests: all tests ==
  usertests: all tests: OK
== Test time ==
time: OK
Score: 66/66
```

通过了所有测试

问题以及解决办法：

a.页表信息的组织架构

打印页表信息的时候开始没有充分了解页表结构，导致一直出错，后来查看了文档，明白了这里的页表为三级页表结构（如图1），此外还看到了虚拟地址和物理地址的映射关系（图2）

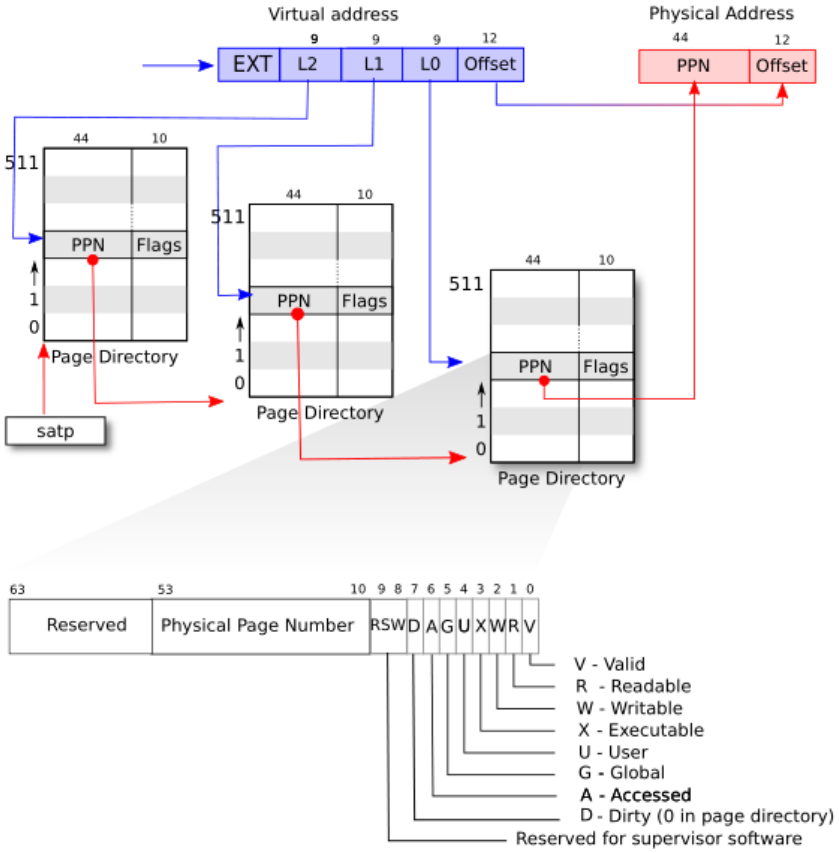


Figure 3.2: RISC-V address translation details.

图 1

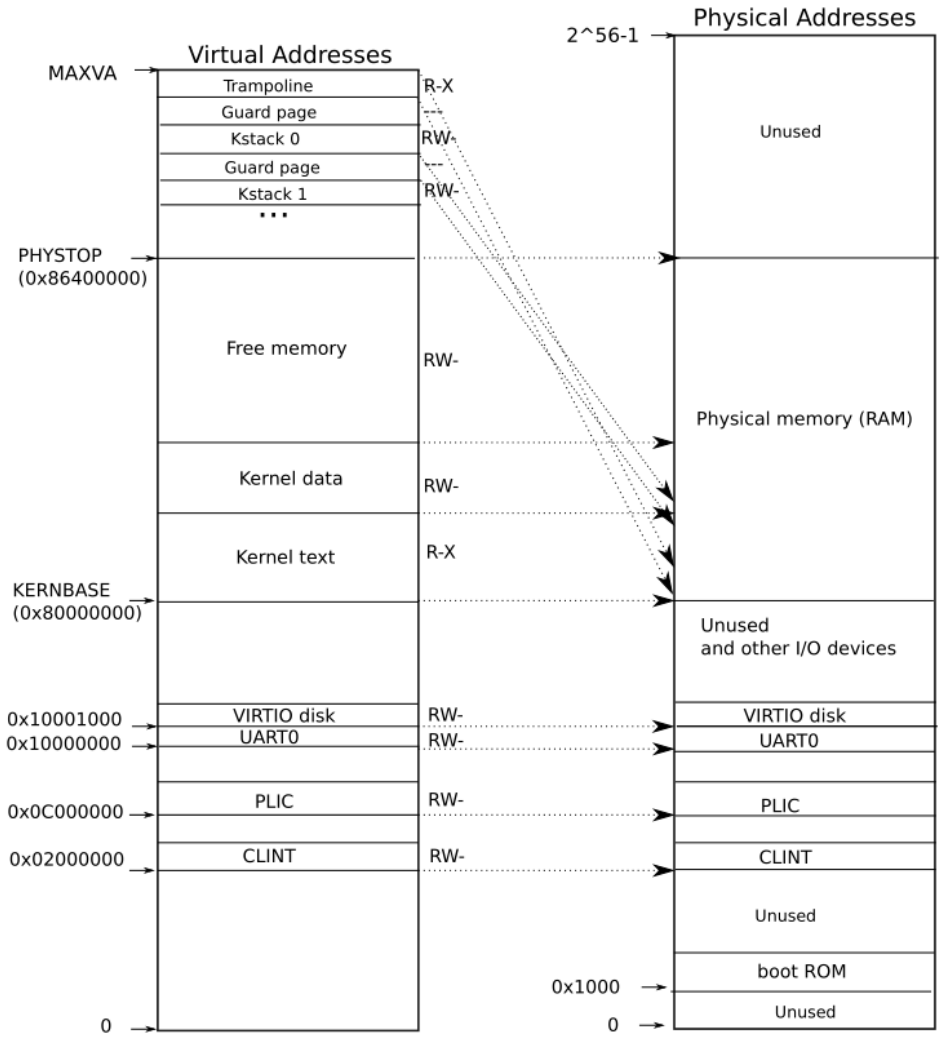


图 2

此外，还有许多相关的信息，对实验很有帮助，这里不一一列举了。

b.为何要将内核页表为每个进程复制一份

一开始内核页表是共有的，所有进程使用同一个，这次实验要求将其改为每个进程独享一个内核页表，一开始不知道为何这样做。其实做了第三个实验就会明白，这样是为了方便为内核页表添加用户地址的映射，这样每个进程的内核页表就可以直接使用用户态的地址，而不必每次使用 walk 遍历来找到其物理地址。

实验心得：

1. 通过本次实验，加深了对页表机制的理解，以及页表机制的具体实现的了解
2. 对于多级页表机制的体会得到了加深，并知道了怎么获取页表相关信息
3. 通过修改页表结构，让页表满足一些特定的功能，这样对于页表机制及其总体的使用逻辑等有了更深的理解
4. 对于虚拟地址，物理地址，映射等基本概念更加了解