lab3 - address space and memory

为了能够更好理解和我个人之前的一些学习和理解,先做出一下关于`page tables`的个人总结,再进行实验的解答

Keywords

- VA (Virtual Address): 虚拟地址，进程看到的地址空间中的地址，由页表映射到物理地址。
- PA (Physical Address): 物理地址，实际内存单元在 DRAM 中的物理位置。
- VPN (Virtual Page Number): 虚拟页号，用于页表查找。
- PPN (Physical Page Number): 物理页号，对应物理页帧编号。
- PFN (Page Frame Number): 同 PPN，指代物理页帧编号。
- PTE (Page Table Entry): 页表项，保存虚拟页到物理页的映射与权限位。
- PT (Page Table): 页表，存储 PTE 的结构，用于地址转换。
- Pagetable_t (Page Table Type): xv6 中定义的页表类型，指向根页表。
- PGSIZE (Page Size): 页大小，xv6 默认为 4KB。
- SUPERPGSIZE (Super Page Size): 超级页大小，通常为 2MB，用于减少 TLB miss。
- TLB (Translation Lookaside Buffer): 地址转换缓存，加速 VA 到 PA 转换。
- MMU (Memory Management Unit): 内存管理单元，负责根据页表进行地址翻译。
- PTE_V (Valid Bit): 页表项有效标志。
- PTE_R/W/X/U (Readable/Writable/Executable/User Bits): 页表项权限位。
- PTE_PPN (Physical Page Number bits): 页表项中的物理页号字段。
- SATP (Supervisor Address Translation and Protection register): 保存根页表物理地址及模式的寄存器。
- KSTACK (Kernel Stack): 进程在内核态使用的栈。
- TRAMPOLINE (Trampoline Page): 保存从用户态陷入/返回的汇编入口代码。
- TRAPFRAME (Trap Frame Page): 保存陷入内核时的寄存器上下文。
- USYSCALL (User Syscall Page): 用户态可读的共享页。
- KERNBASE (Kernel Base Address): 内核映射起始虚拟地址（xv6 中为 0x80000000）。
- PHYSTOP (Physical Memory Limit): xv6 可用的物理内存上限。
- guard page (Guard Page): 栈下方的保护页，防止栈溢出。
- Sv39 (Supervisor-mode Virtual Memory 39-bit): RISC-V 的 39 位虚拟内存模式，三级页表结构。
- L0/L1/L2 (Level 0/1/2 Page Table): Sv39 的三级页表层级。
- PTE Flags (Page Table Entry Flags): 页表项标志控制访问权限。
- PTE_A/D (Accessed/Dirty Bits): 表示页是否被访问或修改过。
- kalloc() (Kernel Allocator Function): 分配一个物理页（4KB）。
- mappages() (Map Pages Function): 建立 VA 到 PA 的映射并设置权限。
- walk() (Page Table Walk Function): 在页表中查找或创建相应的 PTE。
- kvminit() (Kernel VM Init Function): 创建内核页表并建立映射。
- userinit() (User Init Function): 创建第一个用户进程及其页表。
- proc_pagetable() (Process Page Table Create Function): 为进程创建新的用户页表并映射必要页面。
- uvmfirst() (User VM First Page Function): 为用户页表分配第一页并加载 initcode。
- uvmalloc() (User VM Allocate Function): 为用户空间分配并映射内存页。
- uvmcopy() (User VM Copy Function): fork() 时复制父进程页表。
- uvmfree() (User VM Free Function): 释放用户页表和对应物理页。

Content

> **Tip**

以下纯属个人理解总结,总有错误,毕竟我自己也是经过很久理清楚内容以及一些之间的关系. 还有很多内容没有一一阐述出来,因为很多东西一旦要一一阐述,还不如再去看一下文档或者书📖

先看几张图片:

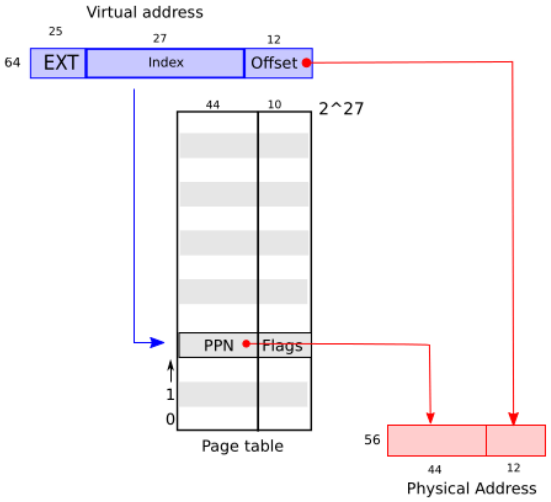


Figure 3.1: RISC-V virtual and physical addresses, with a simplified logical page table.

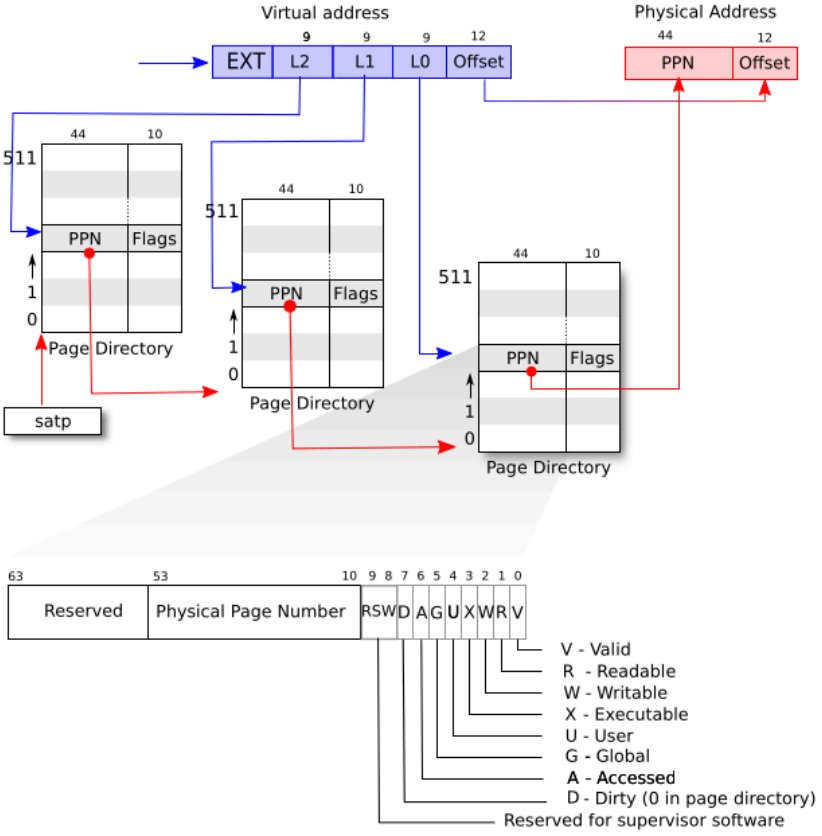


Figure 3.2: RISC-V address translation details.

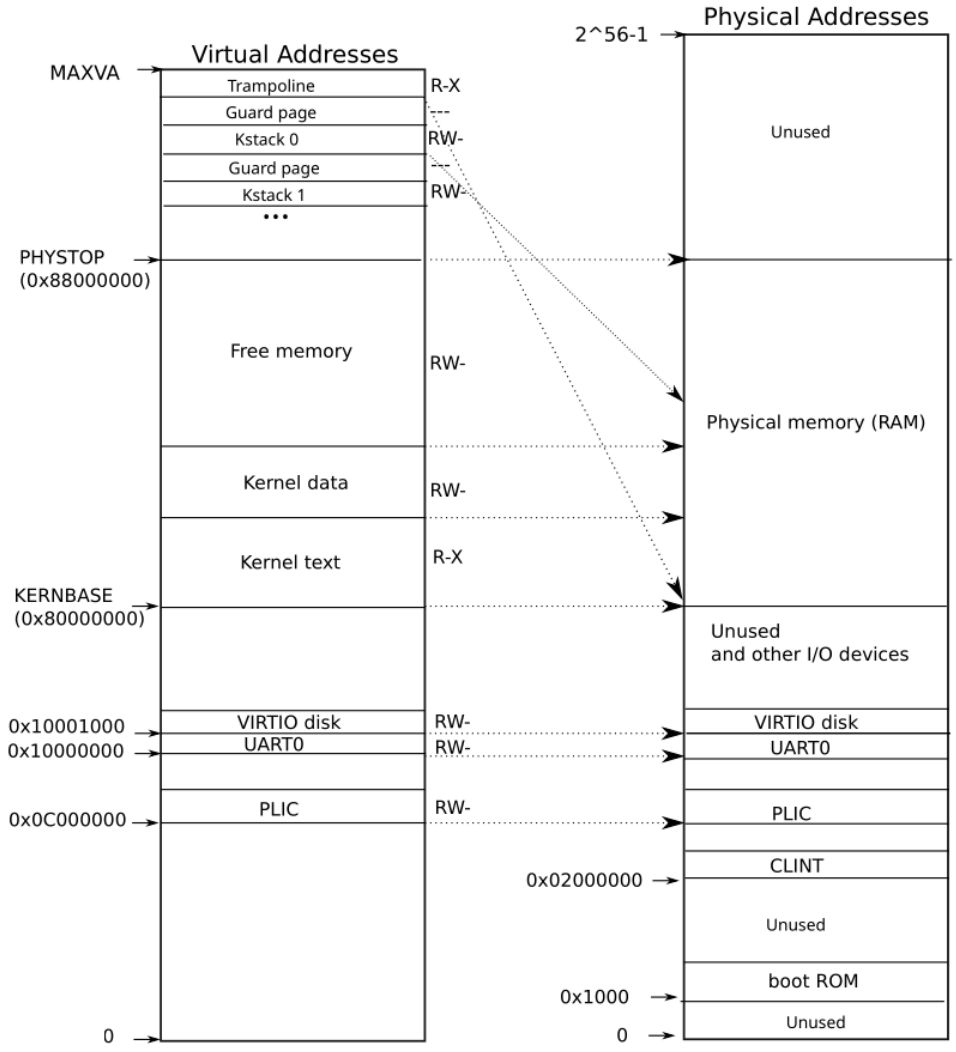


Figure 3.3: On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see.

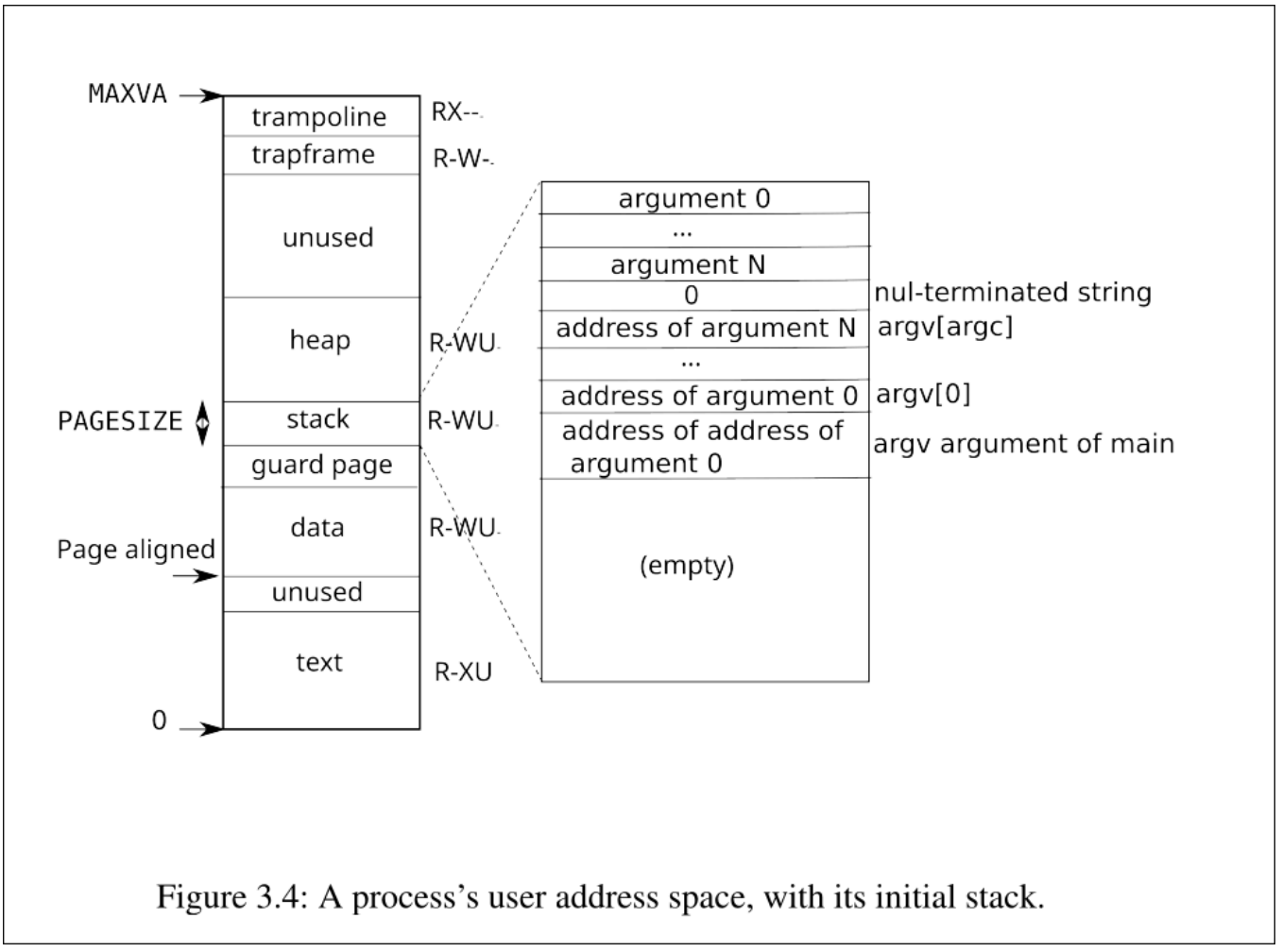


Figure 3.4: A process's user address space, with its initial stack.

这两张图片就是`xv6`中内核地址空间和用户地址空间的示意图,这张照片是高度凝练的.接下来,让我们慢慢剖析.

> **Tip**
>
> 我不会对代码进行讲解,我只会讲解一些重要的逻辑概念和代码的一些实现上的呼应

address space

在`xv6`中,每个进程都有自己独立的地址空间,这个地址空间分为两大部分:用户空间(user space)和内核空间(kernel space).

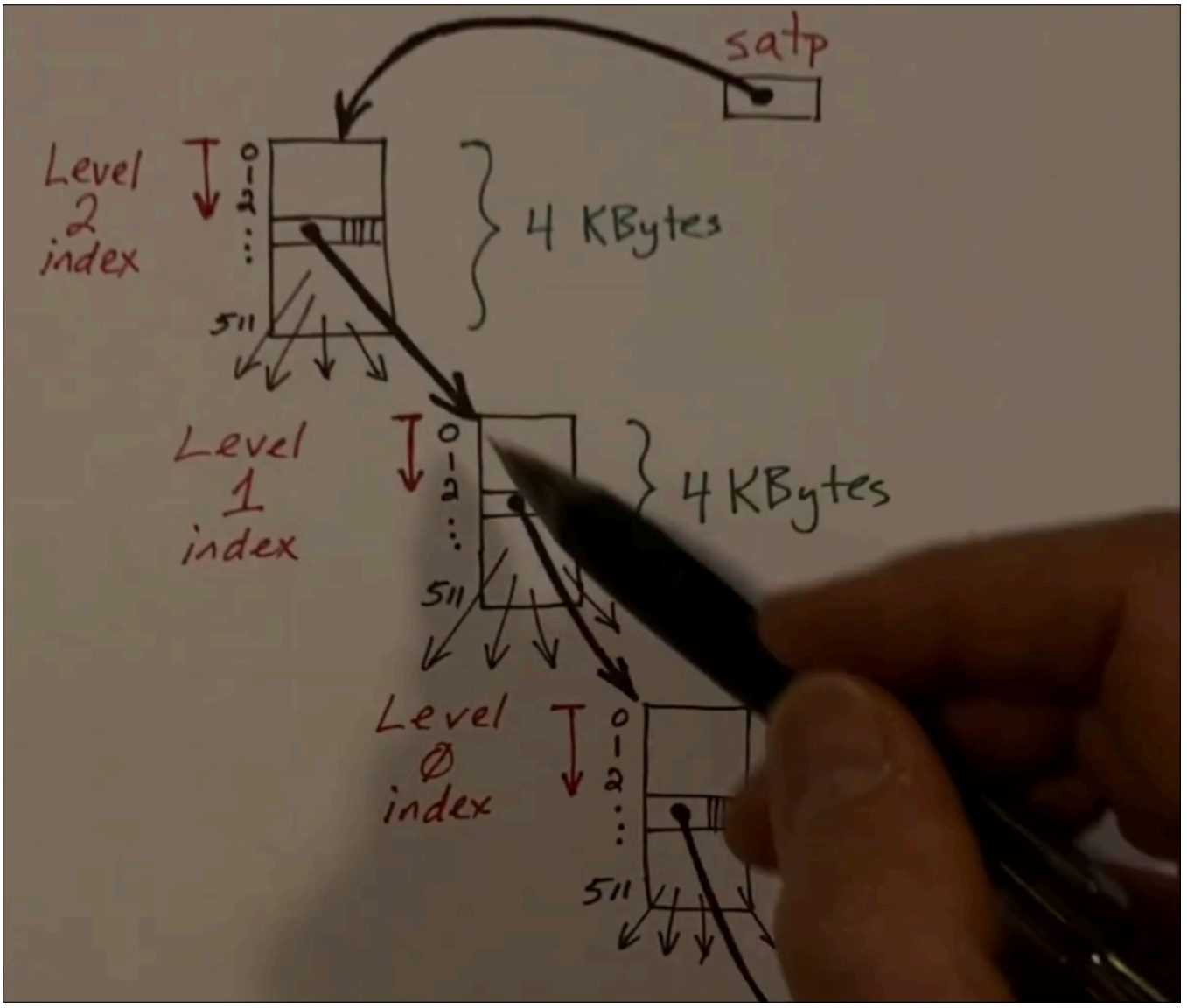
用户空间是进程可以直接访问的部分,而内核空间则是操作系统内核运行的地方,用户进程无法直接访问内核空间.

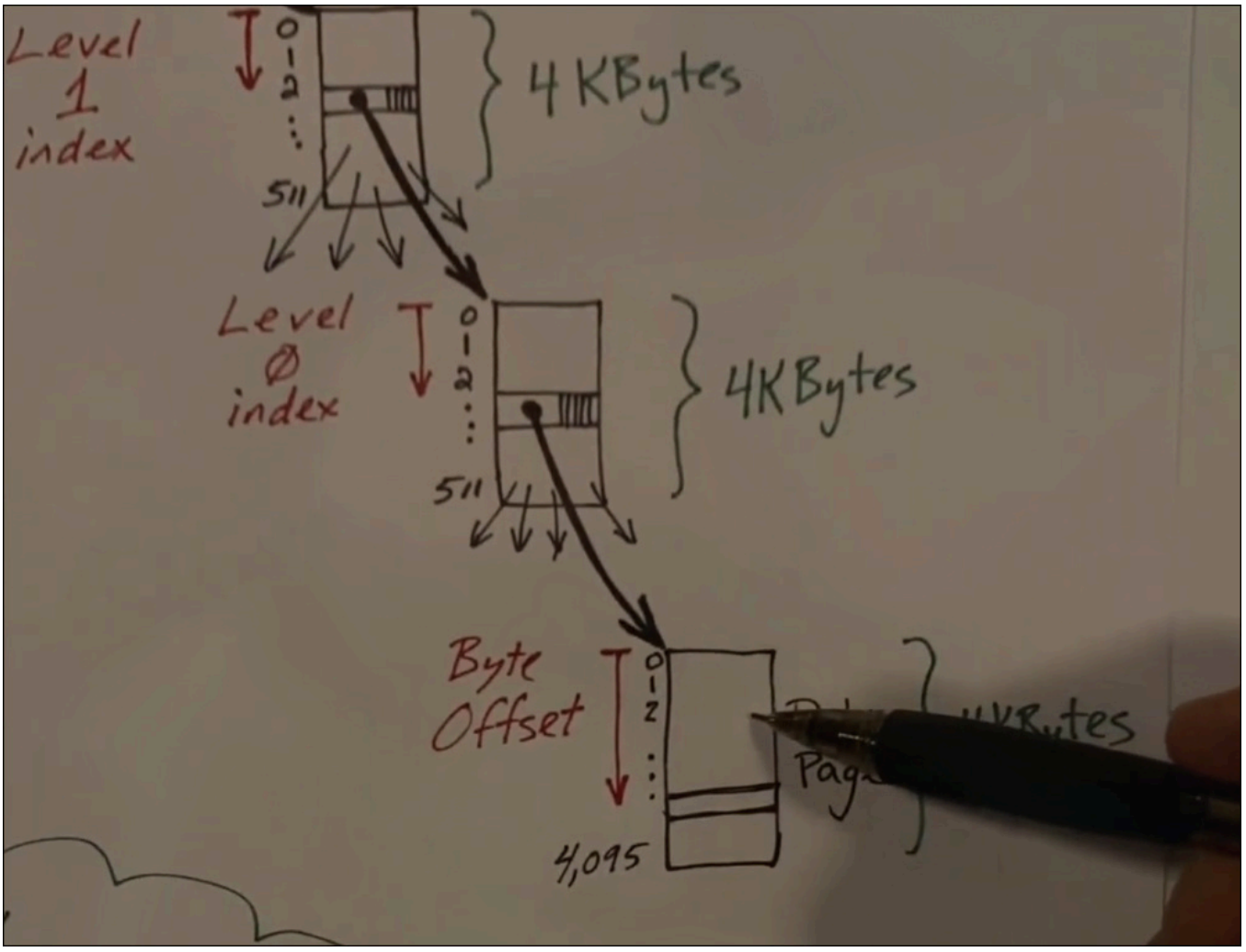
而且,`xv6`采取的是Sv39的虚拟内存管理机制,这意味着每个虚拟地址是39位的,并且采用三级页表结构进行地址转换.

从上面的**第一张图片**可以看出: `va to pa`就是一个虚拟地址借助`page table`转换到物理地址的转换过程.

从**第二张图片**,在`xv6`中,每个虚拟地址被划分为三个部分:VPN2(第一级页表索引),VPN1(第二级页表索引),VPN0(第三级页表索引)和页内偏移(offset).通过这三个索引,可以**逐级查找page table的PTE**,最终实现转换从而找到对应的物理页号(PPN),从而得到物理地址(PA).

其中就是`page table`就是如下图片的**逻辑结构**:


satp
Level 2 index
0
1
2
511
4 KBytes
Level 1 index
0
1
2
511
4 KBytes
Level Ø index
0
1
2
511



至此,初步的对于`page table`和`address转换`有了一个基本的印象

> **Question**
>
> 那么,`page table`的结构是什么呢?

从上面看,是不是就以为是一个`tree`的结构?

从**抽象逻辑概念**上看,的确`page table`是一个`tree`的结构,但在`xv6`中,`page table`的具体实现是通过三个`page`通过`PTE`进行连接,但在**根本上的实现**就是一个指针加上`kalloc`进行分配*4096 bytes*大小实现的一个数组.每个`page`都是由`PTE`组成的,而`PTE`是一个`uint64`类型的变量(*8 bytes*),所以每个`page`中可以存放`512`个`PTE`.

> **Note**
>
> 其实我们看一些文档或者书的时候,会提到`page table page`其实就是类似上面的这三张`page`,就是想说明`page table`这个`tree struct`也是由`page`这个来存储的

那么,`page table`的结构就可以理解为一个**多级索引数组**,每一级索引数组指向下一级索引数组,直到最后一级索引数组指向实际的物理页帧.

我们在做`print page table`这个实验题的时候,我们操作page table就像一个数组的,其实是通过指针移动罢了.

> **Note**
>
> 不管是`kernel address space`还是`user address space`,它们的都是一个`page table`结构,根本上就是上面的图片似的一个大数组,只是这个数组可能还会在*free memory*或者*heap*这些中还会细分一些`page table`

出来使用

> ⚠ **Attention**
>
> 只有一个**kernel page table**,每个`core`都共享这个`kernel page table`. 还有就是,在`kernel`中,`va` --> `pa`都是*direct mapping*的,除了`trampoline page`和`kernel stack pages`以外. 但是在`user mode`下,每个`process`都有一个`process page table`这个的话,这个就**不是direct mapping**了

kernel address space

`kernel address space`的创建

启动阶段 (main.c: main函数)

```
1. kinit()          → initialize physical memory allocator(freelist)
2. kvminit()        → create kernel page table(only core 0)
3. kvminithart()    → set SATP register to point to kernel page table(physical address)
```

第1步: kvminit() - 创建内核页表

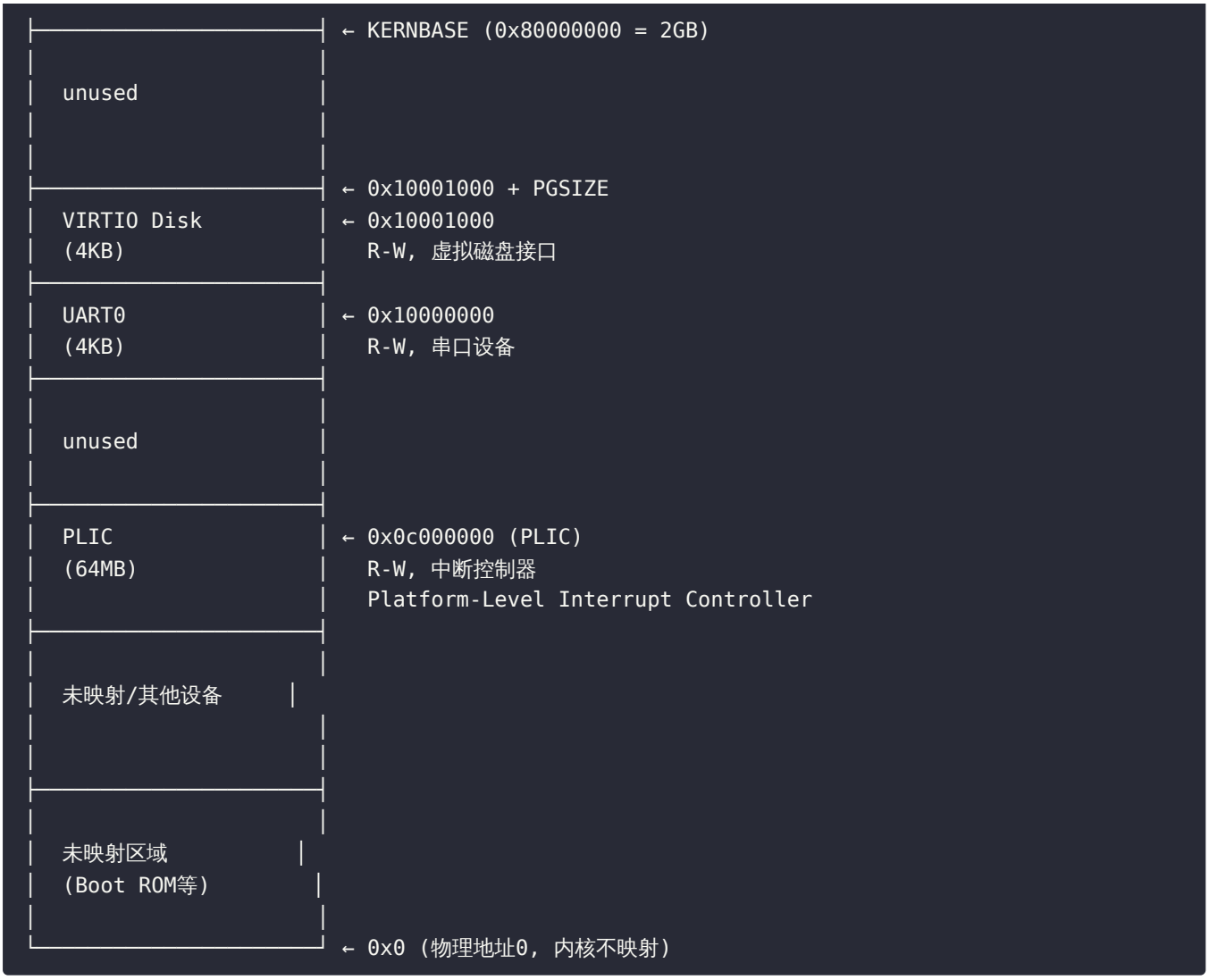
C c

```c
// vm.c
void kvminit(void) {
  kernel_pagetable = kvmmake();
}

pagetable_t kvmmake(void) {
  pagetable_t kpgtbl = (pagetable_t) kalloc();  // 分配L2页表
  memset(kpgtbl, 0, PGSIZE);

  // 映射设备寄存器 (使用identity mapping)
  kvmmap(kpgtbl, UART0, UART0, PGSIZE, PTE_R | PTE_W);
  kvmmap(kpgtbl, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);
  kvmmap(kpgtbl, PLIC, PLIC, 0x4000000, PTE_R | PTE_W);

  // 映射内核代码段 (虚拟0x80000000 = 物理0x80000000)
  kvmmap(kpgtbl, KERNBASE, KERNBASE, (uint64)etext-KERNBASE,
         PTE_R | PTE_X);

  // 映射内核数据段和物理RAM
  kvmmap(kpgtbl, (uint64)etext, (uint64)etext,
         PHYSTOP-(uint64)etext, PTE_R | PTE_W);

  // 映射trampoline到最高虚拟地址
  kvmmap(kpgtbl, TRAMPOLINE, (uint64)trampoline, PGSIZE,
         PTE_R | PTE_X);

  // 为每个进程映射内核栈
  proc_mapstacks(kpgtbl);

  return kpgtbl;
}
```

结果: 创建了`kernel virtual address space`，包括:

- UART0, VIRTIO disk, PLIC等设备映射
- Kernel text (R-X权限)
- Kernel data (R-W权限)
- Free memory区域
- TRAMPOLINE页面
- 64个进程的kernel stack

最终的kernel address space布局



user/proc address space

`user/proc address space`的创建

用户进程启动 (main → userinit → exec)

```
1. userinit()           → create the first user process
2. allocproc()          → allocate process resources
3. proc_pagetable()     → create user page table framework
4. uvmfirst()           → load initcode
5. exec("/init")        → load the real program
```

第1步: userinit() - 创建第一个用户进程

C c

```c
// proc.c
void userinit(void) {
  struct proc *p;

  p = allocproc();  // 分配进程结构
  initproc = p;

  // 分配一页并复制initcode到其中
  uvmfirst(p->pagetable, initcode, sizeof(initcode));
  p->sz = PGSIZE;
```

```
  // 设置用户程序入口
  p->trapframe->epc = 0;      // 从虚拟地址0开始
  p->trapframe->sp = PGSIZE;  // 栈顶指针

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

  p->state = RUNNABLE;
  release(&p->lock);
}
```

第2步: allocproc() - 分配进程资源

C c

```
// proc.c
static struct proc* allocproc(void) {
  struct proc *p;

  // 找一个UNUSED的proc槽位
  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->state == UNUSED) goto found;
    release(&p->lock);
  }
  return 0;

found:
  p->pid = allocpid();
  p->state = USED;

  // 分配trapframe页 (物理页)
  p->trapframe = (struct trapframe *)kalloc();

  // 分配usyscall页 (物理页)
  p->usyscall = (struct usyscall *)kalloc();
  p->usyscall->pid = p->pid;

  // 创建空的用户页表
  p->pagetable = proc_pagetable(p);

  return p;
}
```

第3步: proc_pagetable() - 创建用户页表框架

C c

```
// proc.c
pagetable_t proc_pagetable(struct proc *p) {
  pagetable_t pagetable;

  // 创建空页表 (只有L2根页表)
  pagetable = uvmcreate();

  // 映射TRAMPOLINE (虚拟MAXVA → 物理trampoline代码)
  mappages(pagetable, TRAMPOLINE, PGSIZE,
           (uint64)trampoline, PTE_R | PTE_X);
```

```c
  // 映射TRAPFRAME (虚拟TRAPFRAME → 物理trapframe页)
  mappages(pagetable, TRAPFRAME, PGSIZE,
           (uint64)(p->trapframe), PTE_R | PTE_W);

  // 映射USYSCALL (虚拟USYSCALL → 物理usyscall页)
  mappages(pagetable, USYSCALL, PGSIZE,
           (uint64)(p->usyscall), PTE_R | PTE_U);

  return pagetable;
}
```

此时的用户地址空间布局:

```
MAXVA         → trampoline     (R-X, 无PTE_U)
TRAPFRAME     → trapframe页    (R-W, 无PTE_U)
USYSCALL      → usyscall页     (R-U)
中间全部未映射
```

第4步: uvmfirst() - 加载initcode

C c

```c
// vm.c
void uvmfirst(pagetable_t pagetable, uchar *src, uint sz) {
  char *mem;

  if(sz >= PGSIZE)
    panic("uvmfirst: more than a page");

  mem = kalloc();  // 分配物理页
  memset(mem, 0, PGSIZE);

  // 映射虚拟地址0 → 物理页mem
  mappages(pagetable, 0, PGSIZE, (uint64)mem, PTE_W|PTE_R|PTE_X|PTE_U);

  // 复制initcode到物理页
  memmove(mem, src, sz);
}
```

此时的用户地址空间:

```
MAXVA         → trampoline
TRAPFRAME     → trapframe
USYSCALL      → usyscall
...未映射...
0x1000        未映射
0x0000        → text (initcode, R-W-X-U)
```

第5步: exec() - 加载真正的程序

当initcode执行`exec("/init")`时:

C c

```c
// exec.c
int exec(char *path, char **argv) {
  struct proc *p = myproc();
```

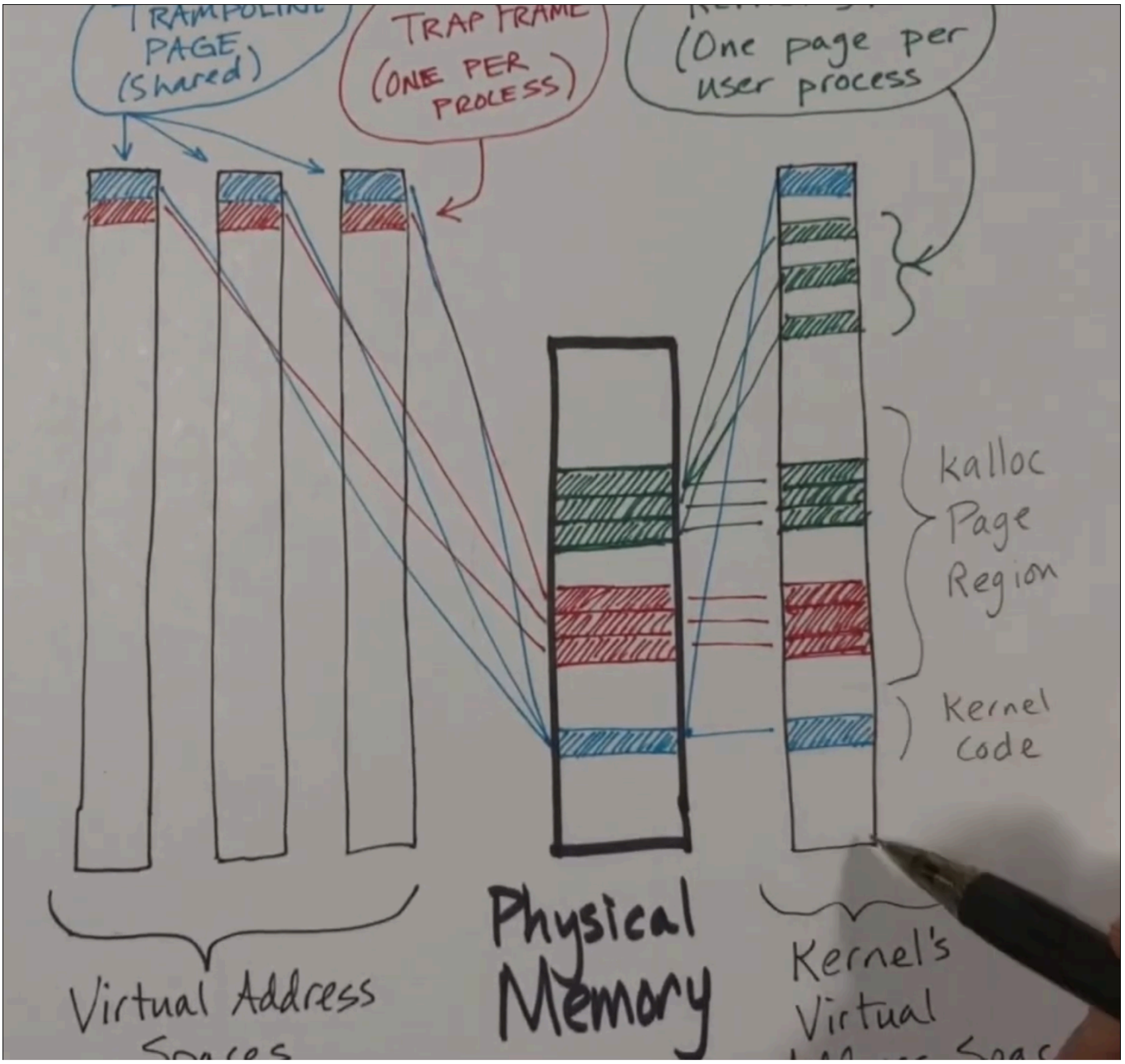
```
  // 创建新的页表
  pagetable = proc_pagetable(p);

  // 解析ELF文件,加载所有program header
  for(i=0; i<elf.phnum; i++) {
    if(ph.type != ELF_PROG_LOAD) continue;

    // 为每个段分配内存并建立映射
    sz = uvmalloc(pagetable, sz, ph.vaddr + ph.memsz,
                  flags2perm(ph.flags));

    // 从文件加载内容到内存
    loadseg(pagetable, ph.vaddr, ip, ph.off, ph.filesz);
  }

  // 现在sz指向程序段结束位置
  sz = PGROUNDUP(sz);

  // 分配guard page + stack
  sz = uvmalloc(pagetable, sz, sz + (USERSTACK+1)*PGSIZE, PTE_W);
  uvmclear(pagetable, sz-(USERSTACK+1)*PGSIZE);  // guard page清除PTE_U
  sp = sz;
  stackbase = sp - USERSTACK*PGSIZE;

  // 将argv字符串push到栈上
  for(argc = 0; argv[argc]; argc++) {
    sp -= strlen(argv[argc]) + 1;
    sp -= sp % 16;
    copyout(pagetable, sp, argv[argc], strlen(argv[argc]) + 1);
    ustack[argc] = sp;
  }

  // 将argv[]指针数组push到栈上
  sp -= (argc+1) * sizeof(uint64);
  sp -= sp % 16;
  copyout(pagetable, sp, (char *)ustack, (argc+1)*sizeof(uint64));

  // 设置寄存器
  p->trapframe->a1 = sp;           // argv参数
  p->trapframe->epc = elf.entry;   // 入口点
  p->trapframe->sp = sp;           // 栈顶

  // 切换到新页表
  oldpagetable = p->pagetable;
  p->pagetable = pagetable;
  p->sz = sz;

  proc_freepagetable(oldpagetable, oldsz);
  return argc;
}
```

最终的user address space布局:

```
MAXVA (256GB)
    ↓
┌──────────────────┐
│  trampoline      │ ← MAXVA, R-X, 无PTE_U (内核使用)
```

> **Note**
>
> 之后`process address space`的创建,我们可以通过`fork`这个可以知道, 就是我们会复制父进程(`/user/init.c`的`sh`之后的第二个进程就是这样)的`page table`,然后子进程和父进程共享

The relationship of physical memory, kernel address space and user address space



这张图片很好的将*physical memory*, *kernel address space*和*user address space*的关系总结了出来.而且,这里还提到了有关于*mapping*的关系.

Optimize

1. **Pages Are Allocated On-Demand**

Key Principle: "Don't allocate memory until you actually need it, otherwise you'll run out of memory."

> **Tip**
>
> `xv6`不使用2MB而是使用4kb是因为,虽然可以提高**TLB**命中率以及减少`page table page`等这些, 但是会造成小空间的浪费以及内存的消耗(因为大多都是小容量的申请)

- Growing Process Memory (`uvmalloc`)

C c

```c
// Allocate PTEs and physical memory to grow process from oldsz to newsz
uint64 uvmalloc(pagetable_t pagetable, uint64 oldsz, uint64 newsz, int xperm) {
  char *mem;
```

```c
  uint64 a;

  oldsz = PGROUNDUP(oldsz);
  for(a = oldsz; a < newsz; a += PGSIZE){
    mem = kalloc();              // ← Allocate physical page ON-DEMAND
    if(mem == 0){
      uvmdealloc(pagetable, a, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);

    // Create page table mapping
    if(mappages(pagetable, a, PGSIZE, (uint64)mem, PTE_R|PTE_U|xperm) != 0){
      kfree(mem);
      uvmdealloc(pagetable, a, oldsz);
      return 0;
    }
  }
  return newsz;
}
```

What happens:

1. Process calls `sbrk()` to grow heap
2. Kernel allocates pages one by one
3. Each page gets mapped into page table
4. Only allocates what's needed!

- Creating Page Tables On-Demand (`walk`)

C c

```c
pte_t *walk(pagetable_t pagetable, uint64 va, int alloc) {
  for(int level = 2; level > 0; level--) {
    pte_t *pte = &pagetable[PX(level, va)];
    if(*pte & PTE_V) {
      // Page table exists, follow it
      pagetable = (pagetable_t)PTE2PA(*pte);
    } else {
      // Page table DOESN'T exist yet!
      if(!alloc)
        return 0;  // Caller said don't allocate

      // Allocate a new page table page NOW
      pagetable = (pde_t*)kalloc();
      if(pagetable == 0)
        return 0;
      memset(pagetable, 0, PGSIZE);
      *pte = PA2PTE(pagetable) | PTE_V;  // Link it in
    }
  }
  return &pagetable[PX(0, va)];
}
```

Examples:

- Lazy allocation

- Copy-on-Write COW
- Demand Paging

2. **TLB**

`xv6` 没有办法自己定义**TLB**,好像*real world*中,都是通过一个硬件实现的**TLB**.

在 `xv6` 中,我们都是通过 `sfence_vma` 指令来刷新**TLB**的内容.

使用场景:

- 切换page table
- trap的时候
- 进程调度(scheduler)的时候

Experiment Questions

Question 1

这个思路说起来就感觉简单,就是copy paste(模仿 `trampoline page` 就行),可能就是要捋清实现思路

实现思路

就是在 `proc.c` 和 `proc.h` 中进行增加功能就行

1. 在proc结构体中添加一个 `struct usyscall *usyscall` 用来存放该页面

```c
// Per-process state
struct proc {
  // ...
struct usyscall *usyscall;   // user syscall data page
  // ...
};
```

2. 在创建进程的过程中，申请一个物理页，参考trapframe.

```c
// kernel/proc.c
static struct proc*
allocproc(void)
{
    // ...

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

    // ...
}
```

3. 在创建页表时将该物理页映射到虚拟地址 `USYSCALL` ，权限位设置为 `PTE_U | PTE_R`.

```c
// kernel/proc.c
pagetable_t
proc_pagetable(struct proc *p)
{
    // ...
    // map the usyscall page just below the trapframe page

  // map the usyscall code (for getpid syscall in user mode)
  // under the trapframe page in user virtual address.
  // now only the getpid system call uses it,
  // so PTE_R and PTE_U is fine.
  if(mappages(pagetable, USYSCALL, PGSIZE,
              (uint64)(p->usyscall), PTE_R | PTE_U) < 0){
    uvmfree(pagetable, 0);
    return 0;
  }

    // ...
}
```

4. 释放页表的时候取消映射

```c
//kernel/proc.c
void
proc_freepagetable(pagetable_t pagetable, uint64 sz)
{
  uvmunmap(pagetable, TRAMPOLINE, 1, 0);
  uvmunmap(pagetable, TRAPFRAME, 1, 0);
  uvmunmap(pagetable, USYSCALL, 1, 0);
  uvmfree(pagetable, sz);
}
```

5. 在销毁进程的时候，将该页内存释放

```c
// 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->usyscall)
        kfree((void*)p->usyscall);
    p->usyscall = 0;

    // ...
}
```

Question 2

这个的话,能够理清上面的这些概念和内容基本就可以理解,就基本只需要直接实现即可.

就是`page table`是一个**多级索引数组**,数组的内容就是`pte`

> **Tip**
>
> 不过这里一开始我就觉得为什么每一行第一个值和最后一个值会不一样的,不是说`direct mapping`吗? 一开始我就没有很好理清上面的内容就导致以上的疑问. 其实,最后一个值`pa`是这个`pte`指向的下一个`index page`的`pte`或者`data page`的`physical address`. 因为,`pte`负责的就是将`va`和`pa`实现转换的一个中间工具,所以也就知道该`pte`的下一级的`pa` 第一个值应该就是这个本身的`pte`所对应的`va`,而它的`pa`应该也是这个值

实现思路

```c
#ifdef LAB_PGTBL
// helper function
void vmprint_level(pagetable_t pagetable, int level, uint64 va_base) {
  // Termination of the recursion
  if (level < 0)
    return;

  for (int i = 0; i < 512; i++) {
    pte_t pte = pagetable[i];
    if (pte & PTE_V) {
      // Calculate the virtual address for this PTE
      // Each level contributes 9 bits to the VA
      uint64 va = va_base | ((uint64)i << PXSHIFT(level));

      // Print the identation
      for (int j = 0; j <= (2 - level); j++) {
        printf(".. ");
      }

      // Print the virtual address, pte value and the physical address that the pte points
to
      printf("%p: pte %p pa %p\n", (void*)va, (void*)pte, (void*)PTE2PA(pte));

      pagetable_t child_pagetable = (pagetable_t)PTE2PA(pte);
      vmprint_level(child_pagetable, level - 1, va);
    }
  }
}

void
vmprint(pagetable_t pagetable) {
  printf("page table %p\n", (void*)pagetable);
  vmprint_level(pagetable, 2, 0); // Start from level 2, VA base = 0
}
#endif
```

Question 3

这个`super page`的实现,就感觉你要很好理解了`memory layout`以及就是`virtual address space`的一整个内容(包括page的创建,与`physical address/physical page`的映射和`pte`的这个内容)

否则,你就有的痛苦了😭😭

实现思路

- 修改内存布局，在可供分配的物理内存中预留出一块区域用于superpage

```c
// memlayout.h
#define KERNBASE 0x80000000L
#define SUPERBASE (KERNBASE + 112*1024*1024)
#define PHYSTOP (KERNBASE + 128*1024*1024)
```

- 修改kallo.c中的代码，在初始化 `kmem` 的时候预留出一定物理空间给superpage

```c
// kallo.c
void
kinit()
{
  initlock(&kmem.lock, "kmem");
  freerange(end, (void*)SUPERBASE);
  superinit();
}
```

- 增加一个 `supermem` 链表，用于管理所有空闲的superpage，并进行相应的初始化工作（模仿kmem）

```c
// kalloc.c
struct {
  struct spinlock lock;
  struct run *freelist;
} kmem, supermem;

void
kinit()
{
  initlock(&kmem.lock, "kmem");
  freerange((void*)end, (void*)SUPERBASE);

  initlock(&supermem.lock, "supermem");
  superinit();
}

void
superinit()
{
  char *p;
  p = (char*)SUPERPGROUNDUP((uint64)SUPERBASE);
  for(; p + SUPERPGSIZE <= (char*)PHYSTOP; p += SUPERPGSIZE)
    superfree(p);
}
```

- 添加 `superalloc` 和 `superfree` 函数，用于分配和释放superpage

```c
// Allocate one 2MB super page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
void *
superalloc(void)
{
  struct run *r;
```

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

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

// Free the 2MB super page of physical memory pointed at by pa,
// which normally should have been returned by a call to superalloc().
void
superfree(void *pa)
{
  struct run *r;

  if(((uint64)pa % SUPERPGSIZE) != 0 || (char*)pa < (char*)SUPERBASE || (uint64)pa >=
PHYSTOP)
    panic("superfree");

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

  r = (struct run*)pa;

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

- 用户在调用 `sbrk` 系统调用的时候，实际会去调用 `uvmalloc` 函数，因此，我们需要修改 `uvmalloc` 函数，根据参数进行不同的分配策略。具体的，我们先分配普通的page，直到虚拟地址对齐到了2MB的位置上，然后我们尽可能多的分配superpage，最后有可能还会剩下一些需要分配的内存，但是不足一个superpage，或者我们已经没有superpage可供分配了，我们继续分配普通的page，直到满足用户需求。

c

```c
// vm.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, int xperm)
{
  char *mem;
  uint64 a;
  int sz;
  if(newsz < oldsz)
    return oldsz;
  // page ... page superpage ... superpage page ... page
  oldsz = PGROUNDUP(oldsz);

  // 分配page直到对齐
  for(a = oldsz; a < SUPERPGROUNDUP(oldsz) && a < newsz; a += sz){
    sz = PGSIZE;
```

```
    mem = kalloc();
    if(mem == 0){
      uvmdealloc(pagetable, a, oldsz);
      return 0;
    }
#ifndef LAB_SYSCALL
    memset(mem, 0, sz);
#endif
    if(mappages(pagetable, a, sz, (uint64)mem, PTE_R|PTE_U|xperm) != 0){
      kfree(mem);
      uvmdealloc(pagetable, a, oldsz);
      return 0;
    }
  }

  // 尽可能多的分配superpage
  for (; a + SUPERPGSIZE < newsz; a += sz)
  {
    sz = SUPERPGSIZE;
    mem = superalloc();
    if (mem == 0){
      break;
    }
    memset(mem, 0, sz);
    if (mappages(pagetable, a, sz, (uint64)mem, PTE_R|PTE_U|xperm) != 0){
      superfree(mem);
      uvmdealloc(pagetable, a, oldsz);
      return 0;
    }
  }

  // 为剩余内存分配page
  for(; a < newsz; a += sz){
    sz = PGSIZE;
    mem = kalloc();
    if(mem == 0){
      uvmdealloc(pagetable, a, oldsz);
      return 0;
    }
    memset(mem, 0, sz);
    if(mappages(pagetable, a, sz, (uint64)mem, PTE_R|PTE_U|xperm) != 0){
      kfree(mem);
      uvmdealloc(pagetable, a, oldsz);
      return 0;
    }
  }
  return newsz;
}
```

- 在 `uvmalloc` 中申请到内存之后，我们需要在页表中添加对应的页表项。superpage只需要在一级页表中设置即可，因为一个一级页表对应512个页，即2MB。因此，我们需要修改 `mappages` 函数，来进行相应的映射。我们根据pa的值来区分当前映射的是page还是superpage，并且我们需要相应的 `superwalk` 来获取 `superpage` 对应的页表项， `walk` 获得的页表项是0级页表中的， `superwalk` 获得1级页表中的页表项。

C c

```
// Walk to a level-1 PTE and return a pointer to it.
// Used for super page (2MB) mapping.
```

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

  pte_t *pte = &pagetable[PX(2, va)];
  if(*pte & PTE_V) {
    pagetable = (pagetable_t)PTE2PA(*pte);
    return &pagetable[PX(1, va)];
  } else {
    if(!alloc || (pagetable = (pde_t*)kalloc()) == 0)
      return 0;
    memset(pagetable, 0, PGSIZE);
    *pte = PA2PTE(pagetable) | PTE_V;
    return &pagetable[PX(1, va)];
  }
}
// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa.
// va and size MUST 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;
  uint64 pgsize;
  pte_t *pte;

  if (pa >= SUPERBASE)
    pgsize = SUPERPGSIZE;
  else
    pgsize = PGSIZE;

  if((va % pgsize) != 0)
    panic("mappages: va not aligned");

  if((size % pgsize) != 0)
    panic("mappages: size not aligned");

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

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

```
  }
  return 0;
}
```

- 分配完了内存，我们也需要处理释放内存。释放内存调用 `uvmdealloc` ，该函数调用 `uvmunmap` 函数去进行实际的释放操作。我们同样根据获得的pa来判断当前释放的是page还是superpage。这里获取va对应的页表项可以统一用 `walk` ，因为 `walk` 遇到叶子结点的时候会直接返回。

```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;
  int sz;
  if((va % PGSIZE) != 0)
    panic("uvmunmap: not aligned");
  for(a = va; a < va + npages*PGSIZE; a += sz){
    sz = PGSIZE;
    if((pte = walk(pagetable, a, 0)) == 0)
      panic("uvmunmap: walk");
    if((*pte & PTE_V) == 0) {
      printf("va=%ld pte=%ld\n", a, *pte);
      panic("uvmunmap: not mapped");
    }
    if(PTE_FLAGS(*pte) == PTE_V)
      panic("uvmunmap: not a leaf");
    uint64 pa = PTE2PA(*pte);
    if (pa >= SUPERBASE){
      a += SUPERPGSIZE;
      a -= sz;
    }
    if(do_free){
      uint64 pa = PTE2PA(*pte);
      if (pa >= SUPERBASE) superfree((void*)pa);
      else kfree((void*)pa);
    }
    *pte = 0;
  }
}
```

- 执行 `fork` 的时候会拷贝内存，调用的是 `uvmcopy` ，因此我们需要进行相应的修改。其实就是遍历父进程用户空间的内存，根据内存的pa判断该页是page还是superpage，然后进行相应的申请并复制，最后在新进程的页表中添加相应的页表项即可。

```c
// Given a parent process's page table, copy
// its memory into a child's page table.
// Copies both the page table and the
// physical memory.
// returns 0 on success, -1 on failure.
// frees any allocated pages on failure.
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
```

```
{
  pte_t *pte;
  uint64 pa, i;
  uint flags;
  char *mem;
  int szinc;

  for(i = 0; i < sz; i += szinc){
    szinc = PGSIZE;
    if((pte = walk(old, i, 0)) == 0)
      panic("uvmcopy: pte should exist");
    if((*pte & PTE_V) == 0)
      panic("uvmcopy: page not present");
    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte);
    if (pa >= SUPERBASE){
      szinc = SUPERPGSIZE;
      if ((mem = superalloc()) == 0)
        goto err;
    }
    else if ((mem = kalloc()) == 0)
      goto err;
    memmove(mem, (char*)pa, szinc);
    if(mappages(new, i, szinc, (uint64)mem, flags) != 0){
      if (szinc == PGSIZE) kfree(mem);
      else superfree(mem);
      goto err;
    }
  }
  return 0;
 err:
  uvmunmap(new, 0, i / PGSIZE, 1);
  return -1;
}
```

> **Tip**
>
> 这里一开始做的时候,我还在想: 我们的 `super page` 的使用是要达到 `SUPERBASE` 这个界限值才会使用 又因为在**kernel mode**下, `va` = `pa` .那么我怎么确保这个会使用到 `super page` 呢? 其实,这里我们要准确理解上面总结的content中的关于The relationship of physical memory, kernel address space and user address space的内容就清楚了. 因为,这里的是 `process` ,所以使用的是 `process address space`

总结

总的来说,我在学习 `page table` 这一块的内容的时候,我学习的真是很难受😣.

我不是不理解,只是我感觉自己是那种懂了一点,但是好像没有懂完,思路和内容是那种有点感觉但好像又没有抓住似的,就是关系也没有理清,导致我学的很痛苦.

之前,我在学习看文档的时候,我就有在做 `super page` 这个习题内容,但是那时候我是真没有理清楚,所以我就直接借助AI,但是我发现AI真就还完全无法理解这个整个系统的框架去实现.

他只能给出他自己的实现,然后panic,然后就又继续修改,一直一直不断的修改,到最后也还是没有达到理想的一个实现.

只能说,AI现在只是一个工具罢了,思路还是要我们自己理清,我们就是那个思路提供者,AI就是一个我们思路的实现者,这样说的我们好像是一个**vibe codding**似的😄😄

还有,在*real world*中,我们不会这样来实现`super page`这样,因为这样毕竟没有高效管理内存而且也会产生很多碎片. 在我了解到的,一般都是使用`buddy system`来进行一个动态的`super page`的生成和合并这些,高效管理内存.