Lab3
实验报告

姓名 李晓畅

学号 20307130261

班级 计算机科学技术

实现思路

PART A

> Some operating systems (e.g., Linux) speed up certain system calls by sharing data in a read-only region between userspace and the kernel. This eliminates the need for kernel crossings when performing these system calls. To help you learn how to insert mappings into a page table, your first task is to implement this optimization for the getpid() system call in xv6.
>
> When each process is created, map one read-only page at USYSCALL (a virtual address defined in *memlayout.h*). At the start of this page, store a struct usyscall (also defined in *memlayout.h*), and initialize it to store the PID of the current process. For this lab, ugetpid() has been provided on the userspace side and will automatically use the USYSCALL mapping. You will receive full credit for this part of the lab if the ugetpid test case passes when running pgtbltest.

- 考虑到需要和内核共享内存，在进程中添加一个包含内存地址的结构或许是个不错的选择。另外 trapframe 也是这样实现的。

```
struct usyscall* usyscall;    //to kernel
```

  把它添加到 ***proc.h*** 中。

  应该直接保存物理地址就好，否者还要多次映射反而可能不太好了。

  实际上我们之后会发现对应的函数也是这样要求的。

- Don't forget to allocate and initialize the page in allocproc().

  因而在为进程创建分配资源的时候也应该为它对应分配资源

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

// Allocate a page.
if((p->usyscall = (struct usyscall *)kalloc()) == 0){
  freeproc(p);
  release(&p->lock);
  return 0;
}
memmove(p->usyscall,&p->pid,8);
//printf("%d\n",p->pid);
```

参照上面分配栈帧的写法就可以了。就是说分配一个块空间存储 ***pid***，如果失败就还原。这里不需要添加释放刚刚分配资源的部分，因为这会在 ***freeproc*** 中实现。我们只需要调用 ***freeproc*** 并释放锁就可以了。

```
// 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;
}
```

Kalloc 用于分配一个页，返回的是物理地址。另外我们需要类型转换为对应的指针才可以赋值。

最后将 ***pid*** 拷贝到对应位置，然后就可以访问到了。

- You can perform the mapping in proc_pagetable() in *kernel/proc.c*.

所以在创建用户页的时候也应该增加在内核中共享内存页的初始化，以及对共享内存块的页表初始化

```
// 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;

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

  a = PGROUNDDOWN(va);
  last = PGROUNDDOWN(va + size - 1);
  for(;;){
    if((pte = walk(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;
}
```

有必要先看看 ***mappages*** 是干什么的，具体来说：

- 首先将要映射的虚拟地址页对齐。
- 将要映射的大小也页对齐，然后计算映射的结尾地址。
- 通过 ***walkpgdir*** 获得一个 ***pte*** 入口。
- 检查次 ***pte*** 是否已经被使用。
- 设置 ***pte*** 条目。
- 检查是否映射结束。
- 更新虚拟地址和物理地址，然后继续上述步骤。

  总之就是映射物理页到逻辑页。

```
// map the trapframe page just below the trampoline page, for
// trampoline.S.
if(mappages(pagetable, TRAPFRAME, PGSIZE,
            (uint64)(p->trapframe), PTE_R | PTE_W) < 0){
  uvmunmap(pagetable, TRAMPOLINE, 1, 0);
  uvmfree(pagetable, 0);
  return 0;
}

if(mappages(pagetable, USYSCALL, PGSIZE,
            (uint64)(p->usyscall), PTE_R | PTE_U) < 0){
  uvmunmap(pagetable, TRAMPOLINE, 1, 0);
  uvmunmap(pagetable, TRAPFRAME, 1, 0);
  uvmfree(pagetable, 0);
  return 0;
}

return pagetable;
```

实现方法还是和栈帧类似。因为不希望修改上面的分配代码，所以把它写在最下面，并且在分配失败时应该回收所有分配的空间。

Choose permission bits that allow userspace to only read the page.

选在 USYSCALL 参照 ***memlayout*** 中的说明。

```
// User memory layout.
// Address zero first:
//   text
//   original data and bss
//   fixed-size stack
//   expandable heap
//   ...
//   USYSCALL (shared with kernel)
//   TRAPFRAME (p->trapframe, used by the trampoline)
//   TRAMPOLINE (the same page as in the kernel)
#define TRAPFRAME (TRAMPOLINE - PGSIZE)
#ifdef LAB_PGTBL
#define USYSCALL (TRAPFRAME - PGSIZE)
```

因为要和内存共享，应该要映射在 ***USYSCALL*** 中

- Make sure to free the page in freeproc().

同上面分配空间相对应的是释放空间。

一方面是要释放为进程额外分配的空间：

```
if(p->trapframe)
  kfree((void*)p->trapframe);

if(p->usyscall)
  kfree((void*)p->usyscall);
```

在 ***freeproc*** 中参考释放栈帧实现，释放。

另一方面是释放分配页时额外分配的空间：

```
// 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);
  uvmunmap(pagetable, USYSCALL , 1, 0);
  uvmfree(pagetable, sz);
}
```

测试结果截图：

```
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
```

运行成功

PART B

Write a function that prints the contents of a page table.

Define a function called vmprint(). It should take a pagetable_t argument, and print that pagetable in the format described below. Insert if(p->pid==1) vmprint(p->pagetable) in *exec.c* just before the return argc, to print the first process's page table.

「Print format」

The first line displays the argument to vmprint. After that there is a line for each PTE, including PTEs that refer to page-table pages deeper in the tree. Each PTE line is indented by a number of ".." that indicates its depth in the tree. Each PTE line shows the PTE index in its page-table page, the pte bits, and the physical address extracted from the PTE. Don't print PTEs that are not valid.

In the example, the top-level page-table page has mappings for entries 0 and 255. The next level down for entry 0 has only index 0 mapped, and the bottom-level for that index 0 has entries 0, 1, and 2 mapped.

- 不妨先看看建议参考的函数　The function freewalk may be inspirational.

```
// Recursively free page-table pages.
// All leaf mappings must already have been removed.
void
freewalk(pagetable_t pagetable)
{
  // there are 2^9 = 512 PTEs in a page table.
  for(int i = 0; i < 512; i++){
    pte_t pte = pagetable[i];
    if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
      // this PTE points to a lower-level page table.
      uint64 child = PTE2PA(pte);
      freewalk((pagetable_t)child);
      pagetable[i] = 0;
    } else if(pte & PTE_V){
      panic("freewalk: leaf");
    }
  }
  kfree((void*)pagetable);
}
```

总之就是遍历页表中所有可用的 PTE，对于非叶子的节点就要进入下一级页表映射，直到遍历每一层的所有 PTE。

freewalk 中判断到达叶节点的方式是检查 ***PTE_R***, ***PTE_W*** 或 ***PTE_X*** 是否至少有一个被置位了，如果都没有那说明不是叶节点。这样判断的依据是，中间节点标志位的设置是在 ***walk*** 函数中进行的，

```
*pte = PA2PTE(pagetable) | PTE_V;
```

walk 函数只给 PTE_V 置位了，其余标志都是 0；而叶节点，一定会包含 ***PTE_R***, ***PTE_W***, ***PTE_X*** 中的至少一个，因为实际的页应该至少要能读或写或执行，不然为什么要创建。

其实看完这个函数我们要如何实现就很显然了。

- You can put vmprint() in *kernel/vm.c*.

```
void
vmprintl(pagetable_t pagetable,int level)
{
  // there are 2^9 = 512 PTEs in a page table.
  for(int i = 0; i < 512; i++){
    pte_t pte = pagetable[i];
    pte_t pa=PA2PTE(pte);
    if((pte & PTE_V)){
      //printf("%d",level);
      for(int k=0;k<level-1;k++){
        printf(".. ");
      }
      printf("..%d: pte %p pa %p\n",i,pte,pa);
    }
    if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
      // this PTE points to a lower-level page table.
      uint64 child = PTE2PA(pte);
      vmprintl((pagetable_t)child,level+1);
    }
  }
}

void
vmprint(pagetable_t pagetable){
  printf("page table %p\n",pagetable);
  vmprintl(pagetable,1);
}
```

因为每层打印的结果都不同，我们需要传递层数作为参数；但是在调用 ***Vmprint*** 的时候我们希望打印所有 PTE，所以可以用两个分别实现。

遍历表元，打印可用的 PTE，如果不是叶子节点，就要递归调用函数打印其子节点。

Use %p in your printf calls to print out full 64-bit hex PTEs and addresses as shown in the example.

另外需要注意的是，打印 PTE 和地址应该用%p 而非%d。

- Define the prototype for vmprint in *kernel/defs.h* so that you can call it from *exec.c*.

为函数添加声明

```
void        vmprint(pagetable_t);
```

- print that pagetable in the format described below. Insert if(p->pid==1) vmprint(p->pagetable) in *exec.c* just before the return argc, to print the first process's page table.

修改第一个进程函数：

```
if (p->pid==1) vmprint(p->pagetable);
return argc; // this ends up in a0, the first argument to main(argc, argv)
```

测试结果截图：

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
page table 0x0000000087f6b000
..0: pte 0x0000000021fd9c01 pa 0x00000000087f6400
.. ..0: pte 0x0000000021fd9801 pa 0x00000000087f6400
.. .. ..0: pte 0x0000000021fda01b pa 0x00000000087f6800
.. .. ..1: pte 0x0000000021fd9417 pa 0x00000000087f6400
.. .. ..2: pte 0x0000000021fd9007 pa 0x00000000087f6400
.. .. ..3: pte 0x0000000021fd8c17 pa 0x00000000087f6000
..255: pte 0x0000000021fda801 pa 0x00000000087f6800
.. ..511: pte 0x0000000021fda401 pa 0x00000000087f6800
.. .. ..509: pte 0x0000000021fdcc13 pa 0x00000000087f7000
.. .. ..510: pte 0x0000000021fdd007 pa 0x00000000087f7400
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000008000400
init: starting sh
```

运行成功

PART C

Some garbage collectors (a form of automatic memory management) can benefit from information about which pages have been accessed (read or write). In this part of the lab, you will add a new feature to xv6 that detects and reports this information to userspace by inspecting the access bits in the RISC-V page table. The RISC-V hardware page walker marks these bits in the PTE whenever it resolves a TLB miss.

Your job is to implement pgaccess(), a system call that reports which pages have been accessed. The system call takes three arguments. First, it takes the starting virtual address of the first user page to check. Second, it takes the number of pages to check. Finally, it takes a user address to a buffer to store the results into a bitmask (a datastructure that uses one bit per page and where the first page corresponds to the least significant bit). You will receive full credit for this part of the lab if the pgaccess test case passes when running pgtbltest.

- 不妨先看看测试函数

Read pgaccess_test() in *user/pgtlbtest.c* to see how pgaccess is used.

```
void
pgaccess_test()
{
  char *buf;
  unsigned int abits;
  printf("pgaccess_test starting\n");
  testname = "pgaccess_test";
  buf = malloc(32 * PGSIZE);
  if (pgaccess(buf, 32, &abits) < 0)
    err("pgaccess failed");
  buf[PGSIZE * 1] += 1;
  buf[PGSIZE * 2] += 1;
  buf[PGSIZE * 30] += 1;
  if (pgaccess(buf, 32, &abits) < 0)
    err("pgaccess failed");
  if (abits != ((1 << 1) | (1 << 2) | (1 << 30)))
    err("incorrect access bits set");
  free(buf);
  printf("pgaccess_test: OK\n");
}
```

大概就是分配空间，修改一些位置，检查被访问过的页是否被正确检测。

- Start by implementing sys_pgaccess() in *kernel/sysproc.c*.

其他配置其实已经完成了，所以直接实现就可以了。

```
74  int
75  sys_pgaccess(void)
76  {
77    uint64 va;
78    int number;
79    uint64 bitmask;
80    argaddr(0, &va) ;
81    argaddr(2, &bitmask);
82    argint(1, &number) ;
83
84    struct proc* p=myproc();
85    pagetable_t pagetable=p->pagetable;
86    uint64 res=0;
87    //printf("%d",number);
88    for (int i=0;i<number;i++){
89      if(va>MAXVA){
90        printf("NO @_@\n");
91        panic("Over range");
92      }
93      //printf("%d: %p\n",i,va);
94      pte_t *pte=walk(pagetable,va,0);
95      if(pte == 0 || (*pte & PTE_V) == 0 ||(*pte & PTE_U) == 0)
96        continue;
97      if(*pte & PTE_A) {
98        res = res | (1L << i);    //set
99        *pte = *pte & (~PTE_A);   //clear
100     }
101     //printf("Done\n");
102     va+=PGSIZE;
103   }
104   printf("Done\n");
105   return copyout(p->pagetable,bitmask, (char*) &res, sizeof(int));
106 }
```

获取参数，然后遍历所有需要检查的逻辑页，映射到物理地址，跳过不可访问的页，对被访问过的页置位，拷贝结果。

You'll need to parse arguments using argaddr() and argint().

正如前一次实验中提到的那样，获取参数要用 ***argaddr*** 和 ***argint***。

argint 利用用户空间的 ***%esp*** 寄存器定位第 n 个参数：***%esp*** 指向系统调用结束后的返回地址。参数就恰好在 ***%esp*** 之上 ***(%esp+4)*** 。因此第 n 个参数就在 ***%esp+4+4*n***；***argaddr*** 也是类似的。

For the output bitmask, it's easier to store a temporary buffer in the kernel and copy it to the user (via copyout()) after filling it with the right bits.

说得很对，我们应该维持一个缓冲，在结束检查后再写回用户态下的存储。

```
// Copy from kernel to user.
// Copy len bytes from src to virtual address dstva in a given page table.
// Return 0 on success, -1 on error.
int
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
```

Copyout 是把一块内核中的空间拷贝到一个逻辑内存指示的地方。注意参数类型。

walk() in *kernel/vm.c* is very useful for finding the right PTEs.

```
// 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)
```

简单来说就是传入一级页表，把虚拟地址转换为实际 PTE 地址，然后这里不需要分配空间。

You'll need to define PTE_A, the access bit, in *kernel/riscv.h*. Consult the RISC-V privileged architecture manual to determine its value.

参考 RISC-V privileged architecture:

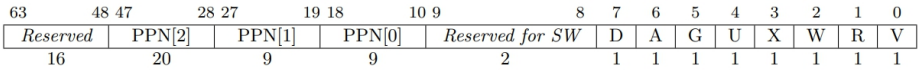


Figure 4.16: Sv39 page table entry.

Each leaf PTE maintains an accessed (A) and dirty (D) bit. When a virtual page is read, written, or fetched from, the implementation sets the A bit in the corresponding PTE. When a virtual page is written, the implementation additionally sets the D bit in the corresponding PTE. The PTE updates are exact and are observed in program order by the local hart. The ordering on loads and stores provided by FENCE instructions and the acquire/release bits on atomic instructions also orders the PTE updates associated with those loads and stores as observed by remote harts.

所以 access 位应该是 6，定义 PTE_A:

```
#define PTE_A (1L << 6)
```

Be sure to clear PTE_A after checking if it is set. Otherwise, it won't be possible to determine if the page was accessed since the last time pgaccess() was called (i.e., the bit will be set forever).

清除就置位为 0 就可以了。PTE_A 在前面已经定义过了。

vmprint() may come in handy to debug page tables.

应该是有用的吧。

测试结果截图：

```
hart 2 starting
hart 1 starting
page table 0x0000000087f6b000
..0: pte 0x0000000021fd9c01 pa 0x00000000087f6400
.. ..0: pte 0x0000000021fd9801 pa 0x00000000087f6400
.. .. ..0: pte 0x0000000021fda01b pa 0x00000000087f6800
.. .. ..1: pte 0x0000000021fd9417 pa 0x00000000087f6400
.. .. ..2: pte 0x0000000021fd9007 pa 0x00000000087f6400
.. .. ..3: pte 0x0000000021fd8c17 pa 0x00000000087f6000
..255: pte 0x0000000021fda801 pa 0x00000000087f6800
.. ..511: pte 0x0000000021fda401 pa 0x00000000087f6800
.. .. ..509: pte 0x0000000021fdcc13 pa 0x00000000087f7000
.. .. ..510: pte 0x0000000021fdd007 pa 0x00000000087f7400
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000008000400
init: starting sh
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgaccess_test: OK
pgtbltest: all tests succeeded
```

测试成功

问题回答

其他系统调用

```
uint64
sys_getpid(void)
{
  return myproc()->pid;
}
```

观察 sys_getpid,不难发现这是一个系统调用，但是得到的是一个用户进程的参数，而且不会改变。这样共享内存就可以先在内核缓存对应的数值，使得内核可以直接读取参数，加速调用。

所以类似的还有：

- getppid, 获取父进程 pid

虚拟内存有什么用处?

- 高效使用内存：VM 将主存看成是存储在磁盘上的地址空间的高速缓存，主存中保存热的数据，根据需要在磁盘和主存之间传送数据。这样不常用的内存就可以存储在磁盘中，使得内存被更好的利用；
- 简化内存管理：VM 为每个进程提供了一致的地址空间，从而简化了链接、加载、内存共享等过程。这样就允许在程序运行时动态将逻辑地址映射到物理地址，内存管理更容易；
- 内存保护：保护每个进程的地址空间不被其他进程破坏。

为什么现代操作系统采用多级页表?

- 使用多级页表可以使得页表在内存中离散存储。多级页表实际上是增加了索引，有了索引就可以定位到具体的项。为了快速随机访问，应该为一个页表分配一块连续的内存单元。这在范围不大时或许可以，但在范围较大时或许就不太理想。

比如考虑我们实验中用到的 RISK-V 系统：

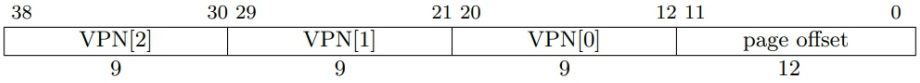


Figure 4.14: Sv39 virtual address.

39 位地址空间中有 27 位表示页序号。换句话说，页表总计有 2^27 项。

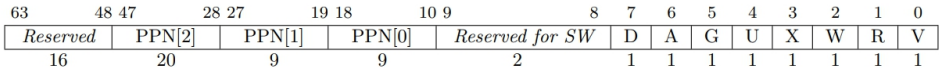


Figure 4.16: Sv39 page table entry.

一个页表项需要 64bit，换言之 8B

总计需要：8B*2^27=2^30B=1G

哪来那么多连续空间。

而多级页表就不需要连续了。连续也不会更快。

- 使用多级页表可以节省页表内存。使用一级页表，需要连续的内存空间来存放所有的页表项。多级页表通过只为进程实际使用的那些虚拟地址内存区请求页表来减少内存使用量。多级页表相当于一棵树，如果没有那么多的地址空间需要寻址，那么就没有必要建完全树了。就是这个例子：

> The three-level structure of Figure 3.2 allows a memory-efficient way of recording PTEs, compared to the single-level design of Figure 3.1. In the common case in which large ranges of virtual addresses have no mappings, the three-level structure can omit entire page directories. For example, if an application uses only a few pages starting at address zero, then the entries 1 through 511 of the top-level page directory are invalid, and the kernel doesn't have to allocate pages those for 511 intermediate page directories. Furthermore, the kernel also doesn't have to allocate pages for the bottom-level page directories for those 511 intermediate page directories. So, in this example, the three-level design saves 511 pages for intermediate page directories and 511×512 pages for bottom-level page directories.

PART C 的 DETECT 流程

- 申请空间，为它分配对应的页。
- 访问对应的页。硬件会负责为被访问过的页页表中的 PTE_A 位置 1。
- 找到要检测页的 PTE，检查它的 PTE_A 位的值，修改 bitmask。
- 复位，这样下次还能检测。

问题和解决

调试和检查

在 PART A 中，会报这个错：

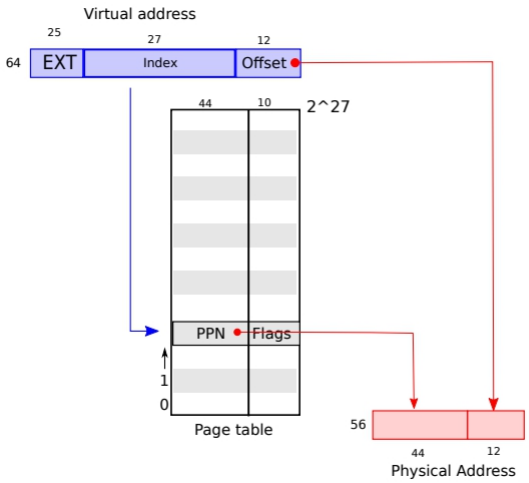
```
xv6 kernel is booting

panic: mappages: remap
```

利用添加输出语句、GDB 调试方法并加以分析，定位问题是重复映射。检查发现是一个参数写错了。对应修改即可。

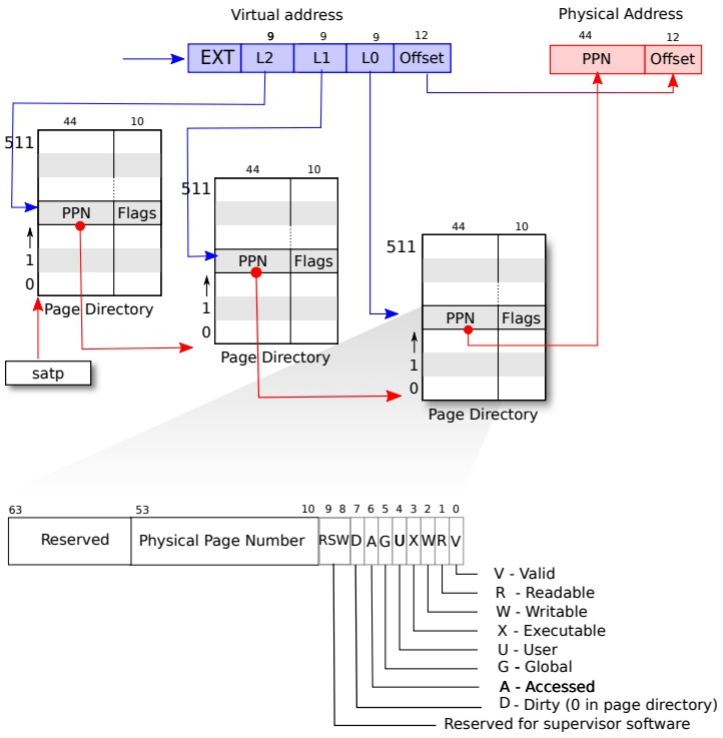
内存实现

在 PART C 中直接在页表中检查 PTE_A 是不对的。

逻辑上说确实是这样：



但实际上是三层页表：



所以要参照建议使用 walk 函数，或者自己实现映射。

实验感想

数据结构的作用

在本次学习的页表中，深刻体现出灵活运用数据结构在提高效率方面的作用。简单的页表类似于 map，比起直接管理物理内存，使用页表更加方便。多级页表类似于树，比起单级页表它有利于离散存储，节约空间。

类比

Xv6 的函数不像很多库那样有非常完备的注释，因而理解它的作用或许需要观察它在其他函数中的使用来实现。参照其他地方编写映射、调用函数，这样同样可以实现目的。换句话说，我们理解代码的方式其实是多种多样的。