实验报告

Lab 3

1. 实验原理

在正式开始本次实验之前，我们有必要了解xv6中虚拟内存和页表的工作机制，以及考察相应的代码实现。

1.1 虚拟内存的实现理念

每个进程都有自己的地址空间（虚拟内存），进程内的指令处理的都是**虚拟地址**。

我们通过**页表**来实现**虚拟地址到物理地址的转换**。在xv6中，虚拟地址和物理地址都是由64位（即8字节）来表示的，但都没有被完全使用。对于虚拟地址而言，只使用了**低39位**，高25位被置为全0；对于物理地址而言，只使用**低56位**，多余的位被保留。

当指令需要处理一个虚拟地址时，我们通过查页表的方式来获取相应的物理地址。页表同样储存在内存中，所以需要一个寄存器`satp`，它**储存了页表的物理地址**。

如果将39位的虚拟地址直接映射到物理地址，那么每个页表应有2^{39}个表项，这显然是不现实的。因此我们需要对虚拟地址和对应的物理地址做进一步的划分。

1.2 虚拟地址和物理地址的划分方式

实际上并没有必要为每个物理地址都创建一个表项，而是考虑**为每页(Page)创建一个表项(Page Table Entry,PTE)**。一页的大小为4KB,即4096字节。

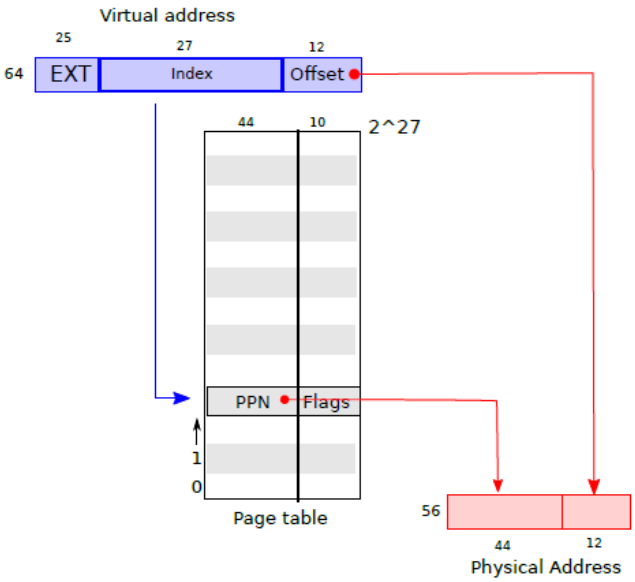
大体上，虚拟地址被划分为两个部分：**索引(Index)和偏移(Offset)**。索引占39位的高27位，用于确定page；偏移占39位的低12位，对应了页的大小$2^{12} = 4096$字节，所以通过偏移能具体确定一个字节（地址）。

相应地，物理地址的56位被划分为44位和12位。低12位与虚拟地址的低12位完全相同，用于确定页内地址；高44位为页表中对应的**物理页号(Physical Page Number,PPN)**。

1.3 页表的结构与虚拟地址到物理地址的转换方式

页表由若干表项组成。每个表项包含高44位的PPN，以及低10位的一些状态标志Flags。这些Flag决定该页表项能否被读/写/执行，是否合法等。但**每个表项的大小为规范，实际上是64位**，也就是有一些位没有被使用。

当查询页表时，我们相当于是已知虚拟地址va，求物理地址pa。其过程为，通过`satp`寄存器获得内存中页表的地址（即首表项的地址）。将虚拟地址的高27位用于在页表中定位表项（**索引=表项的下标**），表项存储了对应的PPN。通过将获得的PPN和虚拟地址的低12位组合，就获得了对应的物理地址。

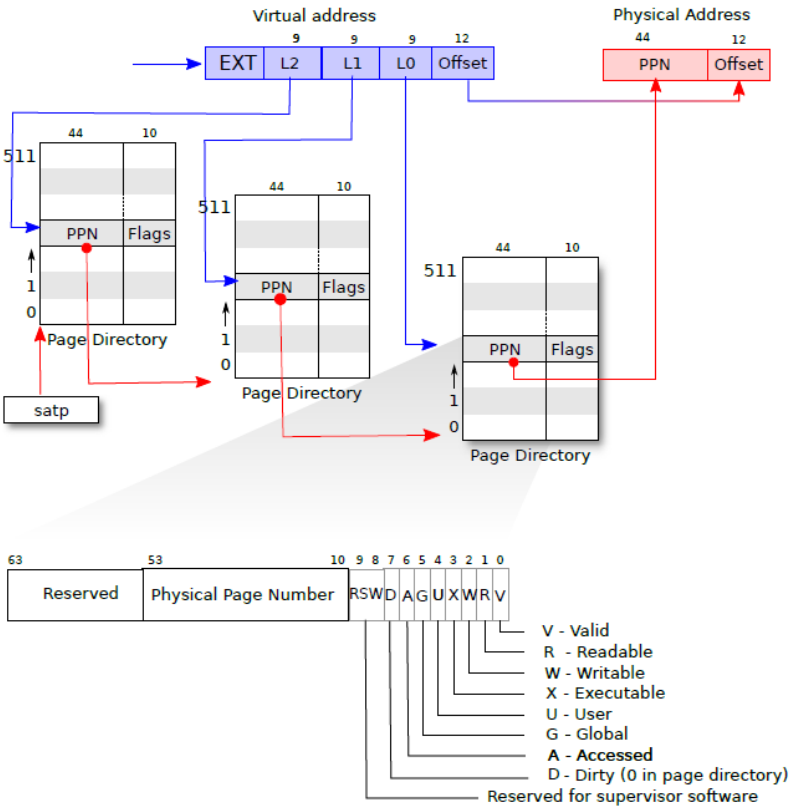


1.4 多级页表

上述的页表并不是xv6中具体实现页表的方式。事实上上述的页表组织形式显然是不合理的——因为虚拟地址高27位就对应了2^{27}个表项，对应到存储就至少应有$2^{27} \times 8 =$1TB，显然我们不可能为每个进程提供如此之大的页表。因此，实际上采用多级页表（xv6中是3级页表）的形式来组织页表。

具体地，将虚拟地址的27位索引平分为3个9位。每个9位作为页表的索引，分别记作L_2、L_1和L_0，并对应了三个页表（分别称为二级页表、一级页表和零级页表）。现在每个页表有$2^9 = 512$个表项，每个表项大小仍为64位（8字节），所以每个页表的大小为$512 \times 8 = 4096B$=4KB，这是合乎常理的页表大小了，并且巧合的是，页表的大小与页的大小一样，都是4KB。

具体来讲，`satp`寄存器存储二级页表的地址，通过L_2确定表项，该表项的**高44位加上12位的0作为下一级页表的地址**（因为页表的大小和页一样，所以页表地址的低12位必定为0）。同样地，通过L_1确定一级页表的表项，获得零级页表的地址；通过L_0确定零级页表的表项，将**高44位加上虚拟地址的低12位的偏移作为最终的物理地址**。



使用多级页表虽然增多了页表的数量（二级、一级、零级页表的数量都不止一个，或称它们不止一页），但我们可以根据自己的需要建立相应的页表（见后面`mappages`和`walk`函数的实现），因此极大减少了空间的浪费。

当进程切换时，`satp`寄存器也随之切换，因此获取不同的页表，从而不同进程中相同的虚拟地址能映射到不同的物理地址。

另外，除了进程的页表外，xv6也为内核单独维护一个页表，相应的结构见xv6 book。

1.5 相关代码

在`kernel/riscv.h`中，含有各种与虚拟内存机制相关的定义与实现。我们指出一些比较关键的代码。

```
typedef uint64 pte_t;
typedef uint64 *pagetable_t; // 512 PTEs

#endif // __ASSEMBLER__

#define PGSIZE 4096 // bytes per page
#define PGSHIFT 12  // bits of offset within a page

#define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))

#define PTE_V (1L << 0) // valid
#define PTE_R (1L << 1)
#define PTE_W (1L << 2)
#define PTE_X (1L << 3)
#define PTE_U (1L << 4) // user can access

// shift a physical address to the right place for a PTE.
#define PA2PTE(pa) ((((uint64)pa) >> 12) << 10)

#define PTE2PA(pte) (((pte) >> 10) << 12)

#define PTE_FLAGS(pte) ((pte) & 0x3FF)

// extract the three 9-bit page table indices from a virtual address.
#define PXMASK          0x1FF // 9 bits
#define PXSHIFT(level)  (PGSHIFT+(9*(level)))
#define PX(level, va) ((((uint64) (va)) >> PXSHIFT(level)) & PXMASK)

// one beyond the highest possible virtual address.
// MAXVA is actually one bit less than the max allowed by
// Sv39, to avoid having to sign-extend virtual addresses
// that have the high bit set.
#define MAXVA (1L << (9 + 9 + 9 + 12 - 1))
```

首先可以看到，xv6将表项PTE定义为64位整数，将页表定义为指向64位整数的指针(也就是一个**表项的数组，大小应为512**)。每页大小PGSIZE为4096字节，页内偏移位数PGSHIFT为12位。这些定义与之前的分析完全吻合。

接下来是`PGROUNDUP`和`PGROUNDDOWN`两个函数。这两个函数是用于将地址向页大小对齐的，可以参考https://stackoverflow.com/questions/43289022/what-do-pgroundup-and-pgrounddown-in-xv6-mean。

`PTE_V` 接下来的五个宏定义对应了相应Flag的位置。分别对应1,2,4,8,16。

`PA2PTE` 用于将物理地址pa转换为相应的PTE。考虑到pa和PTE的结构，注意到它们的中间有一段完全相同的PPN，因此，将pa转换为PTE只需要右移12位消去偏移量，再左移10位（默认Flags都为0）； `PTE2PA` 是同理的，得到的物理地址是不带偏移量的（所以还需要后续处理加上偏移量）。 `PTE_FLAGS` 将pte与0x3FF(即$2^{10}-1$)位与，表示取低10位的Flags。

`PX_MASKS` 为0x1FF,表示为二进制即9个1。 `PXSHIFT` 将PGSHIFT(前面定义了为12)加上9×level，表示各级页表实际操作的地址段，举个例子，比如二级页表，则PXSHIFT=12+9×2=30，对应了二级页表的索引段为虚拟地址的[38,30]这9位；一级页表和零级页表同理。

`PX` 根据给定的level和虚拟地址va，先将va右移 `PXSHIFT(level)` 位，表示处理对应的地址段（如二级页表，[38,30]这9位被移动到[8,0]最低的9位），并将结果与 `PX_MASKS` 位与，表示只取低9位（这样在处理如零级页表时，更高位就被忽略掉）

`MAXVA` 是虚拟地址的上界（比最大值多1）。可以看到该值为2^{38}，对应了256GB。

2. 实验过程

Part A: Speed up system calls

1. 实验要求

根据要求，操作系统可以**通过在用户空间和内核之间的一块只读区域中共享数据加速系统调用**。因为这样就消除了执行系统调用时切换到内核态的需求。在本次实验中，我们以对系统调用 `getpid` 的优化来演示这点。

当进程被创建时，要求将用户虚拟地址 `USYSCALL` 映射到物理地址上。

```
// map the trampoline page to the highest address,
// in both user and kernel space.
#define TRAMPOLINE (MAXVA - PGSIZE)

// map kernel stacks beneath the trampoline,
// each surrounded by invalid guard pages.
#define KSTACK(p) (TRAMPOLINE - (p)*2*PGSIZE - 3*PGSIZE)

// 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)

struct usyscall {
  int pid;  // Process ID
};
#endif
```

根据描述，用户虚拟空间分布大致如下：`trampoline`被定义为`MAXVA-PGSIZE`，表示其为最高页；`trapframe`为`trampoline`的下面一页，再面一页就是我们定义的`USYSCALL`。之后就是堆区了。大致如图:

页名	说明
trampoline	trampoline段
trapframe	中断帧
USYSCALL	**我们需要的只读段**
heap	动态分配的堆区

页名	说明
...	...

在USYSCALL页开头，应存储结构`usyscall`，其成员存储进程的pid。在本实验中，`ugetpid`函数已在用户空间中提供(`user/ulib.c`)，这个函数**直接从对应的虚拟地址读数据**，因而`ugetpid`函数应当是一个比系统调用`getpid`实现更快的版本。

```
int
ugetpid(void)
{
  struct usyscall *u = (struct usyscall *)USYSCALL;
  return u->pid;
}
```

因此，我们需要做的是:

1. 在用户页表中添加USYSCALL的映射。这段内存与内核态共享。
2. 为USYSCALL页分配内存，并初始化。
3. 为USYSCALL添加相应的释放处理。

2. 具体实现

2.1 添加USYSCALL映射

根据提示，我们应在`kernel/proc.c`的`proc_pagetable`函数中实现添加映射的要求。
为了理解运作原理，我们有必要考察相应的函数。它们都位于`kernel/vm.c`中。

```
// create an empty user page table.
// returns 0 if out of memory.
pagetable_t
uvmcreate()
{
  pagetable_t pagetable;
  pagetable = (pagetable_t) kalloc();
  if(pagetable == 0)
    return 0;
  memset(pagetable, 0, PGSIZE);
  return pagetable;
}
```

`uvmcreate`函数简单的调用`kalloc`函数（先忽略其实现）申请内存，并分配PGSIZE大小的内存给pagetable。

```
// 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;
  // 注意这里pte是PTE项的指针，其实本质相当于页表(pagetable_t)
  pte_t *pte;

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

  // va和va+size-1分别是虚拟地址的开头和结尾
  // 分别对它们向下取页大小的整，方便以页为单位处理
  a = PGROUNDDOWN(va);
  last = PGROUNDDOWN(va + size - 1);
  for(;;){
    // walk函数为给定的va找到对应的PTE
    // 由于alloc=1，一定能找到对应的PTE
    // 如果申请失败（walk返回0的情况见walk函数）返回-1
    if((pte = walk(pagetable, a, 1)) == 0)
      return -1;
    // 如果有效，表明该PTE已被其他va映射了
    if(*pte & PTE_V)
      panic("mappages: remap");
    // 这里处理的应该是那些0级页表建立的新页
    // 因为从walk函数可以看出，2级和1级页表建立新页后会设置有效位为1
    // 正因如此，我们才能将需要的物理地址pa直接写到相应的PTE中
    // 中间页表的物理地址是随机申请的，但最终目标物理地址由自己指定
    // 将物理地址转换为PTE格式，并添加标志位perm和有效位（应该都在低10位）
    // 注意pte是通过walk找到的，所以这里为*pte赋值就完成了新映射的建立
    *pte = PA2PTE(pa) | perm | PTE_V;
    // 如果已经处理完毕就结束
    if(a == last)
      break;
    // 每次处理完一页就让虚拟地址和物理地址都增大一页
    a += PGSIZE;
    pa += PGSIZE;
```

```
  }
  return 0;
}
```

`mappages`函数为页表添加新的表项(PTEs)，要建立从va开始虚拟地址的到pa开始的物理地址的一系列映射，范围为给定的size。perm是需要的标志位信息。具体分析见注释。

```
// 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)
{
  if(va >= MAXVA)
    panic("walk");

  // 分级页表，从高级别往低级别找（注意，只处理2级和1级）
  for(int level = 2; level > 0; level--) {
    // 获取va对应的PTE(PX操作已经分析过)
    pte_t *pte = &pagetable[PX(level, va)];
    // 若PTE项非空且有效
    if(*pte & PTE_V) {
      // 获取相应的物理地址（指向下一级页表）
      pagetable = (pagetable_t)PTE2PA(*pte);
    } else {
      // 若没有找到相应的PTE且alloc为1，
      // 为PTE指向的页表分配新页
      // 否则若alloc=0或者申请失败，返回0。
      if(!alloc || (pagetable = (pde_t*)kalloc()) == 0)
        return 0;
      // 页表的新页设为全0（新页的有效位都是0）
      // 这样下次循环时还是要建立新页，直到循环结束(level=0)
      memset(pagetable, 0, PGSIZE);
      // 更新PTE，内容为新分配页表的地址，并标记有效
      *pte = PA2PTE(pagetable) | PTE_V;
    }
  }
  // 返回0级页表对应的PTE（其内容为pa对应的物理地址）
```

```
  return &pagetable[PX(0, va)];
}
```

`walk`函数为输入的va获取相应的物理地址pa。如果没有找到PTE，且alloc为1，则调用`kalloc`函数申请内存并为（本级页表建立的）新页分配内存，再更新本级页表的PTE和有效位。具体分析见注释。注意xv6的实现中普遍使用`pte_t*`类型来代表PTE，它实际是一个指向PTE（内容）的指针。

现在回过头来看`kernel/proc.c`的`proc_pagetable`函数。

```
// Create a user page table for a given process, with no user memory,
// but with trampoline and trapframe pages.
pagetable_t
proc_pagetable(struct proc *p)
{
  pagetable_t pagetable;

  // An empty page table.
  pagetable = uvmcreate();
  if(pagetable == 0)
    return 0;

  // map the trampoline code (for system call return)
  // at the highest user virtual address.
  // only the supervisor uses it, on the way
  // to/from user space, so not PTE_U.
  if(mappages(pagetable, TRAMPOLINE, PGSIZE,
              (uint64)trampoline, PTE_R | PTE_X) < 0){
    uvmfree(pagetable, 0);
    return 0;
  }

  // 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;
  }

  return pagetable;
}
```

该函数首先调用`uvmcreate`创建空页表，再调用`mappages`将虚拟地址`TRAMPOLINE`映射到物理地址`trampoline`，并设置其可读可执行。再将虚拟地址`TRAPFRAME`映射到物理地址`p->trapframe`，并设置其可读可写。

这里简单提一下`uvmfree`和`uvmunmap`的功能。如果`mappages`的结果不正常的话，就需要做相应的处理工作。`uvmfree`这里两次调用都只用于清空页表(其调用`freewalk`函数，递归地清空页表内容并释放页表内存)；`uvmunmap`用于释放指定页数的映射，在`TRAPFRAME`的`mappages`的调用作用是释

放`TRAMPOLINE`该页的映射（设置为全0）。具体见源代码。

那么我们需要做的就是仿照上面两个例子，将虚拟地址`USYSCALL`映射到物理地址,并设置其可读且用户可访问(`PTE_R | PTE_U`)，如果出错的话也要释放前两页的映射。

现在还剩下一个最关键的问题：**应该将USYSCALL映射到哪?**

到这里有些没有头绪了，我们不妨先看看提示中提到的`allocproc`函数，因为按照要求，我们应该在该函数中分配并初始化这段内存。

```
// 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();
  p->state = USED;

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

  // 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;
}
```

这个函数的工作机理很简单，在进程表中寻找未使用的进程，如果找到，为其分配trapframe页、一个空用户页表和相应的上下文(context)。这里我们能看到一个进程的trapframe是怎样形成的：直接用`kalloc`申请内存即可。因此，我们可以考虑：对进程新添加一个成员变量`usyscall`，它按照trapframe同样的方式进行分配，并且在`proc_pagetable`中将`USYSCALL`映射到`usyscallpage`。

具体修改如下：

首先，在`kernel/proc.h`中添加成员变量`usyscallpage`，它应是一个指向usyscall类型的指针。

```
// Per-process state
struct proc {
  ...
  // my edit
  struct usyscall* usyscallpage;
  // edit ends
};
```

在`proc_pagetable`中按类似的方式，添加需要的映射。

```
// Create a user page table for a given process, with no user memory,
// but with trampoline and trapframe pages.
pagetable_t
proc_pagetable(struct proc *p)
{
  ...
  // my edit
  // map the USYSCALL page just below the trapframe page
  if(mappages(pagetable,USYSCALL,PGSIZE,
              (uint64)(p->usyscallpage),PTE_R | PTE_U)<0)
  {
    // need to clean these two pagetables and self's pagetable
    uvmunmap(pagetable,TRAMPOLINE,1,0);
    uvmunmap(pagetable,TRAPFRAME,1,0);
    uvmfree(pagetable,0);
  }
  // edit ends

  return pagetable;
}
```

2.2 为usyscallpage分配内存和初始化

按照上面的分析，只需要在`allocproc`函数中类似trapframe分配内存即可。另外记得按照要求，初始化usycallpage保存当前进程的pid。

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

2.3 为usyscallpage提供释放方法

按照要求，应当在`freeproc`函数中提供对usyscallpage的相应释放代码。`freeproc`函数如下:

```
// 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;
  //my edit
  if(p->usyscallpage)
    kfree((void*)p->usyscallpage);
  p->usyscallpage = 0;
  // edit ends
  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;
}
```

这个函数就是简单的把`proc`类型的成员变量都初始化了。我们只需要类似trapframe添加对usyscallpage的初始化就行，具体见上。

另外还有一个提示没提到的函数`proc_freepagetable`，这个函数是用于释放进程的页表的。如果不更改的话会提示`panic: freewalk:leaf`。

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

3. 实验结果

输入命令`pgtbltest`后结果如下图：

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

Part B: Print a page table

1. 实验要求

按照要求，我们应提供函数打印出页表的内容。该函数接收`pagetable_t`参数，以指定格式打印该页表。具体格式不再赘述。

2. 具体实现

2.1 实现`vmprint()`函数

根据提示，我们不妨参考函数`freewalk`的写法。在上面的分析中提到，`freewalk`函数用于**递归地清空页表内容**。而我们的vmprint函数也正是需要按树状打印各级页表，很容易联想到递归的写法。

`freewalk`函数如下:

```c
// 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);
}
```

仿照上述函数，我们可写出`vmprint`函数于`kernel/vm.c`：

```c
// my edit
// recursively print out the content of pagetables(using helper function)
void vmprint(pagetable_t pagetable)
{
  printf("page table %p\n",pagetable);
  helper(pagetable,0);
}

// recursive function
void helper(pagetable_t pagetable,int depth)
{
  // there are 2^9 = 512 PTEs in a page table.
  for(int i = 0; i < 512; i++)
  {
    // note that this is not a pointer(PTE content)
    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);

      // print out the contents of this level
      // the PTE index
      for(int j = 0;j < depth; ++j)
        printf(".. ");
      printf("..%d:",i);
      // the PTE bits
      printf("pte %p ",pte);
      // the pa
      printf("pa %p\n",PTE2PA(pte));
      // increases the depth
      helper((pagetable_t)child,depth+1);
    }

    // this PTE is a leaf
    else if((pte & PTE_V))
    {
      // print out the contents of this level
      // the PTE index
      for(int j = 0;j < depth; ++j)
```

```
        printf(".. ");
      printf("..%d:",i);
      // the PTE bits
      printf("pte %p ",pte);
      // the pa
      printf("pa %p\n",PTE2PA(pte));
    }
  }
}
// edit ends
```

这里我们需要确定递归的部分，首先在最开始需要打印页表的地址，这个内容只需要打印一次，因此不作为递归部分；其次为了保存递归深度，需要给递归函数增加一个深度参数`depth`，但按要求，函数`vmprint`只接受一个参数`pagetable_t`。基于以上两个原因，考虑增加一个函数`helper`，它作为递归主体，而在`vmprint`函数中就只负责打印页表地址，再调用`helper(pagetable,0)`。

`helper`函数的结构和`freewalk`大致相同。用`for`循环遍历当前页表，检测当前PTE是否指向下一级页表（判断条件同`freewalk`），如果是，根据`depth`打印出当前PTE内容，并递归调用`helper`；如果不是（当前PTE指向va对应的pa），则直接打印出PTE并返回。

2.2 添加定义

根据提示，在`kernel/def.h`中，添加`vmprint`和`helper`的声明。

```
// vm.c
...
void            vmprint(pagetable_t);
void            helper(pagetable_t,int);
```

2.3 调用`vmprint`

根据提示，在`kernel/exec.c`的`exec`函数中添加`vmprint`的调用。

```
int
exec(char *path, char **argv)
{
  ...
  // my edit
  if(p->pid==1)
    vmprint(p->pagetable);
  // edit ends

  return argc; // this ends up in a0, the first argument to main(argc, argv)
  ...
}
```

3. 实验结果

在目录下`make qemu`后结果如下:

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
page table 0x0000000087f6b000
..0:pte 0x0000000021fd9c01 pa 0x0000000087f67000
.. ..0:pte 0x0000000021fd9801 pa 0x0000000087f66000
.. .. ..0:pte 0x0000000021fda01b pa 0x0000000087f68000
.. .. ..1:pte 0x0000000021fd9417 pa 0x0000000087f65000
.. .. ..2:pte 0x0000000021fd9007 pa 0x0000000087f64000
.. .. ..3:pte 0x0000000021fd8c17 pa 0x0000000087f63000
..255:pte 0x0000000021fda801 pa 0x0000000087f6a000
.. ..511:pte 0x0000000021fda401 pa 0x0000000087f69000
.. .. ..509:pte 0x0000000021fdcc13 pa 0x0000000087f73000
.. .. ..510:pte 0x0000000021fdd007 pa 0x0000000087f74000
.. .. ..511:pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
$
```

Part C: Detect which pages have been detected

1. 实验要求

我们需要完成一个系统调用`pgaccess()`，它的作用是报告哪些页被访问(accessed)。该系统调用应接收三个参数va、number和addr：va为检查的第一个用户页面的虚拟地址，number为需要检查的页数，

addr是bitmask的地址，用于将结果存到bitmask中。(bitmask是一个二进制数，用每一位代表一页是否被访问，第一页对应最低位)。

2. 具体实现

2.1 `pgaccess_test()`阅读

根据提示，我们先看一下`pgacess`系统调用是怎么被使用的。

```
void
pgaccess_test()
{
  char *buf;
  unsigned int abits;
  printf("pgaccess_test starting\n");
  testname = "pgaccess_test";
  // buf为第一页的起始地址
  buf = malloc(32 * PGSIZE);
  if (pgaccess(buf, 32, &abits) < 0)
    err("pgaccess failed");
  // 访问第2、3、31页
  buf[PGSIZE * 1] += 1;
  buf[PGSIZE * 2] += 1;
  buf[PGSIZE * 30] += 1;
  if (pgaccess(buf, 32, &abits) < 0)
    err("pgaccess failed");
  // 检查上面访问情况是否对应于access bits
  if (abits != ((1 << 1) | (1 << 2) | (1 << 30)))
    err("incorrect access bits set");
  free(buf);
  printf("pgaccess_test: OK\n");
}
```

具体见注释。可以看到，函数`pgacess`接收如上所说的三个参数，并且应该返回`int`类型的值表示调用情况。

2.2 实现 `sys_pgacess()` 函数

2.2.1 获取参数

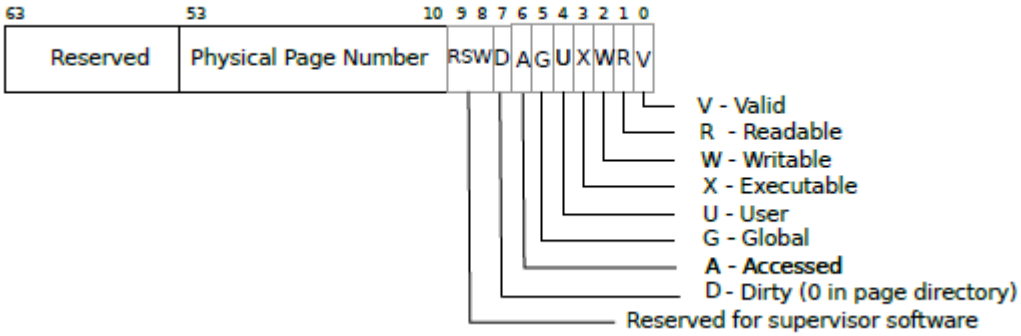
就像上次实现系统调用一样，应该在 `kernel/sysproc.c` 中进行具体的实现。类似地，参数应该使用 `argaddr()` 和 `argint()` 来获取。

```
uint64 buf;
int number;
uint64 bitmask;
argaddr(0,&buf);
argint(1,&number);
argaddr(2,&bitmask);

if(number>32)
    return -1;
```

顺便根据提示，设置了一个最大页数32（因为unsigned int只有32位），超过则返回-1。

2.2.2 定义 `PTE_A`

`PTE_A` 是对应于access bit的flag位。可以在 `kernel/riscv.h` 中类似上面 `PTE_V` 进行定义。根据RISC-V FLAGS的分布，第6位是作为access bit。



```
#define PTE_V (1L << 0) // valid
#define PTE_R (1L << 1)
#define PTE_W (1L << 2)
#define PTE_X (1L << 3)
#define PTE_U (1L << 4) // user can access

// my edit
// PTE_A,the access bit
#define PTE_A (1L << 6)
// edit ends
```

2.2.3 获取PTE并检查，为mask赋值

既然我们传入了起始页的虚拟地址，那么我们就能通过查页表对应的PTE的FLAGS来判定是否被access。我们应该查哪个页表呢？很显然，应该获取当前进程的页表`myproc()->pagetable`。再应用`walk`函数，就能获取虚拟地址对应的PTE。

```
  //temp mask
  uint64 tempmask = 0;
  // get pagetable
  pagetable_t pagetable = myproc()->pagetable;

  for(int i=0;i<number;++i)
  {
    //current page address
    uint64 addr = buf + PGSIZE * i;
    pte_t* pte = walk(pagetable,addr,1);
    if(*pte & PTE_A)
      tempmask = tempmask | (1 << i);
  }
```

按照提示，定义一个临时变量`tempmask`记录临时的mask值。对于`number`页遍历，每页的起始地址应为`buf + PGSIZE * i`，其中`buf`为第一页的起始地址。通过`walk`函数获取PTE(这里alloc取1或0应该都行，不需要新建)。然后将PTE的值(即`*pte`，因为`walk`的返回值类型是`pte_t*`)和定义的`PTE_A`相与。如果结果非0，说明该PTE的FLAG中access bit为1，让`tempmask`对应位置的值为1(通过和`1 << i`相或)。

2.2.4 将`tempmask`的值复制给`bitmask`

遍历完毕后，用`copyout`函数将`tempmask`的内容复制给用户态变量`bitmask`，这个函数在实验2中已经使用过，用法不再赘述。

```
// using copyout() to copy tempmask to user space
  if(copyout(pagetable,bitmask,(char*)&tempmask,sizeof(tempmask))<0)
    return -1;
```

2.2.5 清除PTE的`PTE_A`位

根据提示，在检查了PTE的access位后，需要清除该位，否则被访问过一次后该位一直为1，从而达不到判断从上次调用`pgaccess`为之某页是否被访问过（也就是说，是否被访问成为了累积性的，而不是两次调用之间的）。

通过让PTE的内容`*pte`和`~PTE_A`相与即可消去PTE的access位，因为`~PTE_A`除了第7位(access位)都是1，所以`*pte`的其他位不变，access位清0。

```
    // clean access bit of PTE
    *pte = *pte & ~PTE_A;
```

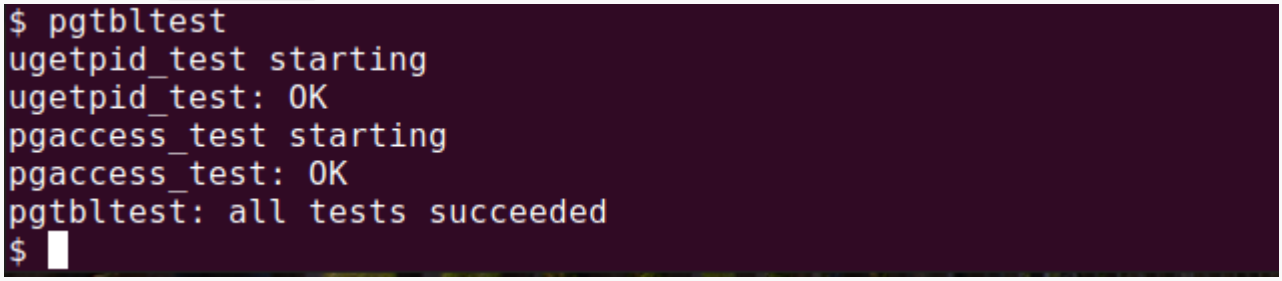
2.2.6 完整代码

```
#ifdef LAB_PGTBL
int
sys_pgaccess(void)
{
  // lab pgtbl: your code here.
  uint64 buf;
  int number;
  uint64 bitmask;
  argaddr(0,&buf);
  argint(1,&number);
  argaddr(2,&bitmask);

  //temp mask
  uint64 tempmask = 0;

  // upper limit for number
  if(number>32)
    return -1;

  // get pagetable
  pagetable_t pagetable = myproc()->pagetable;

  for(int i=0;i<number;++i)
  {
    //current page address
    uint64 addr = buf + PGSIZE * i;
    pte_t* pte = walk(pagetable,addr,1);

    if(*pte & PTE_A)
      tempmask = tempmask | (1 << i);

    // clean access bit of PTE
    *pte = *pte & ~PTE_A;
  }

  // using copyout() to copy tempmask to user space
  if(copyout(pagetable,bitmask,(char*)&tempmask,sizeof(tempmask))<0)
    return -1;

  return 0;
```

```
}
#endif
```

2.2.7 实验结果

输入命令`pgtbltest`后结果如下图:



3. 问题回答

（1）在Part A 加速系统调用部分，除了`getpid()`系统调用函数，你还能想到哪些系统调用函数可以如此加速?

需要进入内核只获取数据的系统调用函数都可以这样加速，如`uptime()`、 `dup()`等。

（2）虚拟内存有什么用处?

一方面，使用虚拟内存能使每个进程有自己独立的地址空间并使其受到保护，从而省去了考虑进程间是否产生内存冲突的忧虑，并简化了内存的管理；另一方面，使用虚拟内存使程序员能以逻辑的思维考虑问题，而不需要在物理地址上编程，简化了编程逻辑。

（3）为什么现代操作系统采用多级页表?

如果直接使用单级页表，根据上面的分析，在xv6系统中页表项数可达2^{27}，且大多数PTE都是不可用的（空闲的），既占内存，又浪费空间。使用多级页表后，每张页表的大小仅为4KB，且对于不需要的映射就可以相应的减少页表的建立。例如，假如仅有一个虚拟地址`va`到物理地址`pa`的映射（即一页大小的映射），那么对应产生的页表只有1个二级页表、1个一级页表和1个零级页表，无论是使用的空间还是空间利用率都远超单级页表形式。

（4）简述Part C的detect 流程。

首先在用户态调用函数`pgaccess()`，转到内核态执行系统调用`sys_pgaccess()`（这部分在实验2中已经详细叙述过）。

进入`sys_pgaccess()`函数后，首先使用`argint()`和`argaddr()`函数获取传入的参数。先检测传入的页数是否超过设定的最大值，超过则直接返回-1表示不合理。用临时变量`tempmask`记录mask值。用for循环遍历`number`个页，当前页的起始(虚拟)地址为`buf + PGSIZE * i`。用`walk()`函数查询当前进程页表`myproc()->pagetable`，获得虚拟地址对应的PTE(的指针)`pte`。用位与操作符`&`检查获取的`*pte`的第7位(access bit)是否为1，如果为1则将`tempmask`对应位的值设为1（通过与`1 << i`位或）。检测后，

清除`*pte`的access bit为0，这样保证记录的访问信息是从上次调用`pgaccess()`函数开始算起而非累计的。最后，使用`copyout()`函数将内核态变量`tempmask`的内容复制到用户态变量`bitmask`。

4. 实验问题

4.1 搞不清楚将`USYSCALL`映射到哪个物理地址

因为这部分实验没有给出很明确的提示，包括你需要在`proc`结构体里添加新成员变量。但实际上也不是很难想到，因为确实没有其他的路可走了。

在`allocproc()`函数中看到`trapframe`的初始化给了我启发，就能类似写出对`USYSCALL`的操作。Part A的实验基本就是"`trapframe`怎么做我就怎么做"就可以了。

4.2 不清楚如何按要求格式打印页表

主要是实验要求`vmprint()`函数只接受一个页表参数，如果不记录递归深度的话这个我不知道怎么写，且`vmprint()`函数开头还要打印一个页表地址，如果递归`vmprint()`的话页表地址就重复打印了。

最后是考虑用了一个辅助函数`helper()`，这个函数记录递归深度，且让`vmprint()`单独只打印一次页表地址，问题就很轻松的解决了。

如果辅助函数也不能使用的话，我觉得只能是用一个全局变量记录递归深度，感觉很麻烦，还需要自己手动调整深度，不如把递归深度作为函数参数。

4.3 不清除PTE的access bit的结果

最开始我完全没看懂提示说的

> Be sure to clean PTE_A after checking if it is set.

是什么意思（做完了再看，感觉是他表述问题，因为`PTE_A`是定义的数`1L << 6`，“清除PTE_A”让人看不懂，改成清除access bit可能更好理解一些），所以我一开始没有清除PTE的access bit，得到结果如下:

```
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgtbltest: pgaccess_test failed: incorrect access bits set, pid=3
$
```

根据提示，得到的access bit不匹配正确结果。于是我按照提示，在`sys_pgaccess()`函数中添加了相

应的`printf()`函数和`vmprint()`函数以debug。

```
printf("buf:%p\n",buf);
...
vmprint(pagetable);
...
for(...)
{
    ...
    printf("addr:%p ",addr);
    ...
    printf("pte:%p\n",*pte);
    ...
}
...
printf("%d\n",tempmask);
...
```

得到结果如下：

```
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
buf:0x0000000000004010
page table 0x0000000087f40000
..0:pte 0x0000000021fcf001 pa 0x0000000087f3c000
.. ..0:pte 0x0000000021fcec01 pa 0x0000000087f3b000
.. .. ..0:pte 0x0000000021fcf45b pa 0x0000000087f3d000
.. .. ..1:pte 0x0000000021fce8d7 pa 0x0000000087f3a000
.. .. ..2:pte 0x0000000021fce407 pa 0x0000000087f39000
.. .. ..3:pte 0x0000000021fce0d7 pa 0x0000000087f38000
.. .. ..4:pte 0x0000000021fdb8d7 pa 0x0000000087f6e000
.. .. ..5:pte 0x0000000021fdc017 pa 0x0000000087f70000
.. .. ..6:pte 0x0000000021fd7c17 pa 0x0000000087f5f000
.. .. ..7:pte 0x0000000021fdbc17 pa 0x0000000087f6f000
.. .. ..8:pte 0x0000000021fd0c17 pa 0x0000000087f43000
.. .. ..9:pte 0x0000000021fd1817 pa 0x0000000087f46000
.. .. ..10:pte 0x0000000021fd0817 pa 0x0000000087f42000
.. .. ..11:pte 0x0000000021fd1017 pa 0x0000000087f44000
.. .. ..12:pte 0x0000000021fd8017 pa 0x0000000087f60000
.. .. ..13:pte 0x0000000021fd1417 pa 0x0000000087f45000
.. .. ..14:pte 0x0000000021fd0417 pa 0x0000000087f41000
.. .. ..15:pte 0x0000000021fd1c17 pa 0x0000000087f47000
.. .. ..16:pte 0x0000000021fd2017 pa 0x0000000087f48000
.. .. ..17:pte 0x0000000021fd2417 pa 0x0000000087f49000
.. .. ..18:pte 0x0000000021fd2817 pa 0x0000000087f4a000
.. .. ..19:pte 0x0000000021fd2c17 pa 0x0000000087f4b000
.. .. ..20:pte 0x0000000021fd3017 pa 0x0000000087f4c000
.. .. ..21:pte 0x0000000021fd3417 pa 0x0000000087f4d000
.. .. ..22:pte 0x0000000021fd3817 pa 0x0000000087f4e000
.. .. ..23:pte 0x0000000021fd3c17 pa 0x0000000087f4f000
.. .. ..24:pte 0x0000000021fd4017 pa 0x0000000087f50000
.. .. ..25:pte 0x0000000021fd4417 pa 0x0000000087f51000
.. .. ..26:pte 0x0000000021fd4817 pa 0x0000000087f52000
.. .. ..27:pte 0x0000000021fd4c17 pa 0x0000000087f53000
.. .. ..28:pte 0x0000000021fdb417 pa 0x0000000087f6d000
.. .. ..29:pte 0x0000000021fd8417 pa 0x0000000087f61000
.. .. ..30:pte 0x0000000021fd8817 pa 0x0000000087f62000
.. .. ..31:pte 0x0000000021fcdc17 pa 0x0000000087f37000
.. .. ..32:pte 0x0000000021fcd817 pa 0x0000000087f36000
```

```
.. .. ..33:pte 0x0000000021fcd417 pa 0x0000000087f35000
.. .. ..34:pte 0x0000000021fcd017 pa 0x0000000087f34000
.. .. ..35:pte 0x0000000021fccc17 pa 0x0000000087f33000
.. .. ..36:pte 0x0000000021fcc817 pa 0x0000000087f32000
..255:pte 0x0000000021fcfc01 pa 0x0000000087f3f000
.. ..511:pte 0x0000000021fcf801 pa 0x0000000087f3e000
.. .. ..509:pte 0x0000000021fdc413 pa 0x0000000087f71000
.. .. ..510:pte 0x0000000021fd78c7 pa 0x0000000087f5e000
.. .. ..511:pte 0x0000000020001c4b pa 0x0000000080007000
addr:0x0000000000004010 pte:0x0000000021fdb8d7
addr:0x0000000000005010 pte:0x0000000021fdc017
addr:0x0000000000006010 pte:0x0000000021fd7c17
addr:0x0000000000007010 pte:0x0000000021fdbc17
addr:0x0000000000008010 pte:0x0000000021fd0c17
addr:0x0000000000009010 pte:0x0000000021fd1817
addr:0x000000000000a010 pte:0x0000000021fd0817
addr:0x000000000000b010 pte:0x0000000021fd1017
addr:0x000000000000c010 pte:0x0000000021fd8017
addr:0x000000000000d010 pte:0x0000000021fd1417
addr:0x000000000000e010 pte:0x0000000021fd0417
addr:0x000000000000f010 pte:0x0000000021fd1c17
addr:0x0000000000010010 pte:0x0000000021fd2017
addr:0x0000000000011010 pte:0x0000000021fd2417
addr:0x0000000000012010 pte:0x0000000021fd2817
addr:0x0000000000013010 pte:0x0000000021fd2c17
addr:0x0000000000014010 pte:0x0000000021fd3017
addr:0x0000000000015010 pte:0x0000000021fd3417
addr:0x0000000000016010 pte:0x0000000021fd3817
addr:0x0000000000017010 pte:0x0000000021fd3c17
addr:0x0000000000018010 pte:0x0000000021fd4017
addr:0x0000000000019010 pte:0x0000000021fd4417
addr:0x000000000001a010 pte:0x0000000021fd4817
addr:0x000000000001b010 pte:0x0000000021fd4c17
addr:0x000000000001c010 pte:0x0000000021fdb417
addr:0x000000000001d010 pte:0x0000000021fd8417
addr:0x000000000001e010 pte:0x0000000021fd8817
addr:0x000000000001f010 pte:0x0000000021fcdc17
addr:0x0000000000020010 pte:0x0000000021fcd817
addr:0x0000000000021010 pte:0x0000000021fcd417
addr:0x0000000000022010 pte:0x0000000021fcd017
addr:0x0000000000023010 pte:0x0000000021fccc17
1
```

```
buf:0x0000000000004010
page table 0x0000000087f40000
..0:pte 0x0000000021fcf001 pa 0x0000000087f3c000
.. ..0:pte 0x0000000021fcec01 pa 0x0000000087f3b000
.. .. ..0:pte 0x0000000021fcf45b pa 0x0000000087f3d000
.. .. ..1:pte 0x0000000021fce8d7 pa 0x0000000087f3a000
.. .. ..2:pte 0x0000000021fce407 pa 0x0000000087f39000
.. .. ..3:pte 0x0000000021fce0d7 pa 0x0000000087f38000
.. .. ..4:pte 0x0000000021fdb8d7 pa 0x0000000087f6e000
.. .. ..5:pte 0x0000000021fdc0d7 pa 0x0000000087f70000
.. .. ..6:pte 0x0000000021fd7cd7 pa 0x0000000087f5f000
.. .. ..7:pte 0x0000000021fdbc17 pa 0x0000000087f6f000
.. .. ..8:pte 0x0000000021fd0c17 pa 0x0000000087f43000
.. .. ..9:pte 0x0000000021fd1817 pa 0x0000000087f46000
.. .. ..10:pte 0x0000000021fd0817 pa 0x0000000087f42000
.. .. ..11:pte 0x0000000021fd1017 pa 0x0000000087f44000
.. .. ..12:pte 0x0000000021fd8017 pa 0x0000000087f60000
.. .. ..13:pte 0x0000000021fd1417 pa 0x0000000087f45000
.. .. ..14:pte 0x0000000021fd0417 pa 0x0000000087f41000
.. .. ..15:pte 0x0000000021fd1c17 pa 0x0000000087f47000
.. .. ..16:pte 0x0000000021fd2017 pa 0x0000000087f48000
.. .. ..17:pte 0x0000000021fd2417 pa 0x0000000087f49000
.. .. ..18:pte 0x0000000021fd2817 pa 0x0000000087f4a000
.. .. ..19:pte 0x0000000021fd2c17 pa 0x0000000087f4b000
.. .. ..20:pte 0x0000000021fd3017 pa 0x0000000087f4c000
.. .. ..21:pte 0x0000000021fd3417 pa 0x0000000087f4d000
.. .. ..22:pte 0x0000000021fd3817 pa 0x0000000087f4e000
.. .. ..23:pte 0x0000000021fd3c17 pa 0x0000000087f4f000
.. .. ..24:pte 0x0000000021fd4017 pa 0x0000000087f50000
.. .. ..25:pte 0x0000000021fd4417 pa 0x0000000087f51000
.. .. ..26:pte 0x0000000021fd4817 pa 0x0000000087f52000
.. .. ..27:pte 0x0000000021fd4c17 pa 0x0000000087f53000
.. .. ..28:pte 0x0000000021fdb417 pa 0x0000000087f6d000
.. .. ..29:pte 0x0000000021fd8417 pa 0x0000000087f61000
.. .. ..30:pte 0x0000000021fd8817 pa 0x0000000087f62000
.. .. ..31:pte 0x0000000021fcdc17 pa 0x0000000087f37000
.. .. ..32:pte 0x0000000021fcd817 pa 0x0000000087f36000
.. .. ..33:pte 0x0000000021fcd417 pa 0x0000000087f35000
.. .. ..34:pte 0x0000000021fcd0d7 pa 0x0000000087f34000
.. .. ..35:pte 0x0000000021fccc17 pa 0x0000000087f33000
.. .. ..36:pte 0x0000000021fcc817 pa 0x0000000087f32000
..255:pte 0x0000000021fcfc01 pa 0x0000000087f3f000
```

```
.. ..511:pte 0x0000000021fcf801 pa 0x0000000087f3e000
.. .. ..509:pte 0x0000000021fdc413 pa 0x0000000087f71000
.. .. ..510:pte 0x0000000021fd78c7 pa 0x0000000087f5e000
.. .. ..511:pte 0x0000000020001c4b pa 0x0000000080007000
addr:0x0000000000004010 pte:0x0000000021fdb8d7
addr:0x0000000000005010 pte:0x0000000021fdc0d7
addr:0x0000000000006010 pte:0x0000000021fd7cd7
addr:0x0000000000007010 pte:0x0000000021fdbc17
addr:0x0000000000008010 pte:0x0000000021fd0c17
addr:0x0000000000009010 pte:0x0000000021fd1817
addr:0x000000000000a010 pte:0x0000000021fd0817
addr:0x000000000000b010 pte:0x0000000021fd1017
addr:0x000000000000c010 pte:0x0000000021fd8017
addr:0x000000000000d010 pte:0x0000000021fd1417
addr:0x000000000000e010 pte:0x0000000021fd0417
addr:0x000000000000f010 pte:0x0000000021fd1c17
addr:0x0000000000010010 pte:0x0000000021fd2017
addr:0x0000000000011010 pte:0x0000000021fd2417
addr:0x0000000000012010 pte:0x0000000021fd2817
addr:0x0000000000013010 pte:0x0000000021fd2c17
addr:0x0000000000014010 pte:0x0000000021fd3017
addr:0x0000000000015010 pte:0x0000000021fd3417
addr:0x0000000000016010 pte:0x0000000021fd3817
addr:0x0000000000017010 pte:0x0000000021fd3c17
addr:0x0000000000018010 pte:0x0000000021fd4017
addr:0x0000000000019010 pte:0x0000000021fd4417
addr:0x000000000001a010 pte:0x0000000021fd4817
addr:0x000000000001b010 pte:0x0000000021fd4c17
addr:0x000000000001c010 pte:0x0000000021fdb417
addr:0x000000000001d010 pte:0x0000000021fd8417
addr:0x000000000001e010 pte:0x0000000021fd8817
addr:0x000000000001f010 pte:0x0000000021fcdc17
addr:0x0000000000020010 pte:0x0000000021fcd817
addr:0x0000000000021010 pte:0x0000000021fcd417
addr:0x0000000000022010 pte:0x0000000021fcd0d7
addr:0x0000000000023010 pte:0x0000000021fccc17
1073741831
pgtbltest: pgaccess_test failed: incorrect access bits set, pid=3
```

首先，可以看到起始地址为0x0410,然后后续地址每次增加一个`PGSIZE`，即0x0510,0x0610,...
我们可以在`vmprint()`打印出的页表中找到对应的行。从0级页表的第4行开始，打印出的`*pte`和页

表的PTE就完全对应，这与`walk()`函数的作用相对应。

然后可以发现，第一次调用`pgaccess()`得到的`tempmask`的结果居然是1，但这时在用户态并没有对这些页有任何访问，结果应该是0才对。我推测可能是因为`argaddr(0,&buf)`这句实际上对这些页的起始地址（即第一页）进行了访问，所以可以看到，该地址对应的PTE为`0x0000000021fdb8d7`，其第7位(access bit)为1。

在第二次调用`pgaccess()`时，对第2、3、31页进行了访问，所以在`pgaccess_test()`函数期望的结果为(1<<1)+(1<<2)+(1<<30)=1073741830，比这里打印出的1073741831少1。显然这个1就是由于未清除上次调用的access位而多出的对第一页的访问。

在发现这一点后就能明白提示的意思了，因此加上清除access bit的部分就能解决问题。

5. 实验感想

这次实验做起来比第二次顺畅挺多的，可能是经验更丰富了吧。三个Part的实验难度个人排名是A>C>B，第一个实验主要难在需要对页表添加映射、初始化页并申请内存和释放页的流程和原理都要有一个比较深入的理解，这部分比较抽象，也不太好理解，尤其是像`walk()`和`mappages()`这种函数。

6. 参考资料

1. xv6-book
2. 4.3 Page Table