华东师范大学数据学院实验报告

课程名称：操作系统	年级：2022级	上机实践成绩：
指导教师：翁楚良	姓名：刘蔚璁	
上机实践名称：页表	学号：10225501443	上机实践日期：2024.5

一、实验目的

- 学习页表的实现机制，用户拷贝数据到内核态的方法

二、实验内容

- 加速 getpid()
- 递归遍历页表，碰到有效的就遍历进下一层页表，打印页表内容

三、实验环境

VSCode

四、实验过程及结果

预备知识

在开始实验前，需要先了解一下页表的相关知识。

内存管理方法

- **分段**
  将虚拟内存空间划分为代码、堆、栈等多个逻辑段，分别为每个逻辑段维护一对基址—界限寄存器。
  存在严重的内、外部碎片问题。
- **分页**
  将虚拟内存空间切成固定长度的分片，每个固定长度的单元称之为页（Page），将物理内存看成是定长槽块的阵列，这些槽块大小与页相同，每个槽块叫做页帧（Page Frame），每个页帧都可以装载一个虚拟内存页。
  xv6采用分页的手段来构建虚拟内存系统，提供页表完成虚拟地址到物理地址的转换。

页表翻译

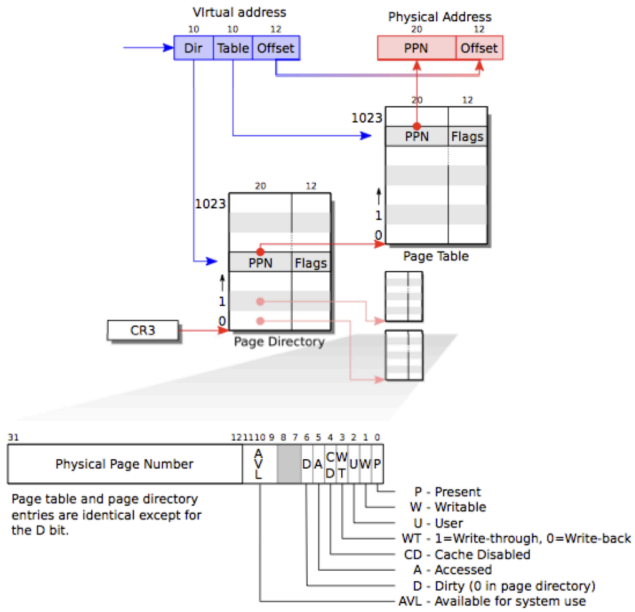


Figure 2-1. x86 page table hardware.

如图 2-1 所示，实际上，地址的翻译有两个步骤。一个页表在物理内存中像一棵两层的树。树的根是一个 4096 字节的页目录，其中包含了 1024 个类似 PTE 的条目，但其实每个条目是指向一个页表页的引用。而每个页表页又是包含 1024 个 32 位 PTE 的数组。

- 分页硬件使用虚拟地址的高 10 位来决定对应页目录条目。
- 如果想要的条目已经放在了页目录中，分页硬件就会继续使用接下来的 10 位来从页表页中选择出对应的 PTE。否则，分页硬件就会抛出错误。通常情况下，大部分虚拟地址不会进行映射，而这样的二级结构就使得页目录可以忽略那些没有任何映射的页表页。

每个 PTE 都包含一些标志位，说明分页硬件对应的虚拟地址的使用权限。（这些的标志位和页表硬件相关的结构体都在 mmu.h 定义）

- PTE_P 表示 PTE 是否陈列在页表中：如果不是，那么一个对该页的引用会引发错误（也就是：不允许被使用）。
- PTE_W 控制着能否对页执行写操作；如果不能，则只允许对其进行读操作和取指令。
- PTE_U 控制着用户程序能否使用该页；如果不能，则只有内核能够使用该页。

xv6

每个进程都有自己的页表，xv6 会在进程切换时通知分页硬件切换页表。页表将虚拟地址（x86 指令所使用的地址）映射为物理地址（处理器芯片向主存发送的地址）。

- **进程**

  进程是一个抽象概念，它让一个程序可以假设它独占一台机器，xv6 使用页表（由硬件实现）来为每个进程提供其独有的地址空间。
  一个 xv6 进程分为：
  - 用户内存空间（指令、数据、栈）

- 仅对内核可见的进程状态

在xv6中，对进程控制块 `struct proc` 的定义和相关的操作函数实现，例如进程的初始化、创建和销毁等主要在 `proc.c` 中实现。
每个进程都有一个结构体 `struct proc`，来维护一个进程的状态，其中最为重要的状态是进程的页表，内核栈，当前运行状态，在 `proc.h` 中定义：

```
struct proc {
    struct spinlock lock;       // 进程锁，用于保护进程结构的并发访问

    // p->lock must be held when using these:
    enum procstate state;       // 进程当前的运行状态
    void *chan;                 // 进程等待的通道，如果非空，则表示进程正在休眠等待某
    int killed;                 // 表示进程是否被杀死
    int xstate;                 // 返回给父进程 wait 的退出状态
    int pid;                    // 进程 ID，唯一标识一个进程

    // wait_lock must be held when using this:
    struct proc *parent;        // 父进程指针，指向当前进程的父进程

    // these are private to the process, so p->lock need not be held.
    uint64 kstack;              // 内核栈的虚拟地址，用于保存进程的内核执行栈
    uint64 sz;                  // 进程内存的大小（字节），表示进程占用的内存空间大小
    pagetable_t pagetable;      // 用户页表，用于管理进程的虚拟地址空间
    struct trapframe *trapframe; // 用户态中断帧，用于保存用户态执行的上下文信息
    struct context context;     // 进程上下文，用于进行进程间的上下文切换
    struct file *ofile[NOFILE]; // 打开文件数组，用于存储进程打开的文件描述符
    struct inode *cwd;          // 当前工作目录
    char name[16];              // 进程名字（用于调试）
};
```

可以看出，一个进程和它的页表通过 `pagetable_t pagetable;` 联系在一起。

- **创建进程**

  在 main 初始化了一些设备和子系统后，它通过调用 `userinit` 建立了第一个进程,它首先调用 `allocproc` 分配结构体 `struct proc` 并对其进行初始化。 `allocproc` 在每个进程被创建时都会被调用。
  （ `userinit` 和 `allocproc` 都在 `proc.c` 中 ）

作业一

在xv6中，如果用户态调用系统调用，就会切换到内核态，这中间一定是有开销的，至少CPU要保存用户态进程的上下文，然后CPU被内核占有，系统调用完成后再切换回来。作业一本质是

加速 getpid()，思路是为每一个进程多分配一个虚拟地址位于 USYSCALL 的页，然后这个页的开头保存一个 usyscall 结构体，结构体中存放这个进程的 pid

- 创建每个进程时，在 USYSCALL 处映射一个只读页面（在 memlayout.h 中定义的 VA ）。
- 在此页面的开头，存储一个 struct usyscall （也在 memlayout.h 中定义），并初始化它以存储当前进程的 PID 。
- 对于本实验， ugetpid() 已在用户空间端提供，并将自动使用 USYSCALL 映射。
- 要求运行 pgtbltest 时 ugetpid 测试用例通过。

实验思路

在 xv6 中，结构体 proc 本身存储在内核空间中，如果用户空间需要访问进程控制块（struct proc）中的内容（例如 pid），通常需要通过系统调用来实现。
而用户空间可以通过页表访问自己的虚拟地址空间中的物理内存，因此，考虑把 pid 直接放到物理内存中来加速访问。

- **struct proc**

  上文中给出了该结构体的代码部分，其中 `struct trapframe *trapframe;` 用于存储指向当前进程的陷阱帧（trap frame）的指针，**每个进程的陷阱帧通常会被分配一个单独的页面。**
  从而确保在发生中断、异常或系统调用时，处理器状态（包括寄存器值、程序计数器等）有一个专门的地方进行保存和恢复。
  因此，我们可以仿照其来实现目标页面的分配。
  首先，已有定义在 `memlayout.h` 的结构体：

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

  所以我们只需在 `struct proc` 中加入字段：
  `struct usyscall *usyscall;  // 指向用户态系统调用结构的指针`

- **proc.c / allocproc**

  在创建一个进程时，通常会对 struct proc 结构体进行初始化。
  - 系统中有一个 struct proc 数组，存储了系统中所有进程的信息，用于在系统运行时创建、销毁和管理进程。
  - **涉及的重要函数**
    `kalloc()`：
    负责从操作系统的内存池中分配一块固定大小的内存块，并返回指向该内存块的指针。分配的内存块通常是物理页面（或者页帧），大小通常是操作系统的页面大小（在 xv6 中通常是 4KB）的整数倍。

`void *memmove(void *dest, const void *src, size_t n)`：
将源地址的数据复制到目标地址中

- **对 trapframe 的操作**

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

这段代码给陷阱帧分配了一个物理页面，并返回指向该内存块的指针。如果分配失败，就调用 `freeproc()` 清理和回收进程结构体及其相关资源（如内存、页表等）。

- **添加代码段**
只需仿照上面的代码来分配 usyscall 的页面：

```
if((p->usyscall = (struct usyscall *)kalloc()) == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
}
```

除此之外还需要将 pid 的值存储到该物理页中，采用 `memmove()` 函数实现：

```
memmove(p->usyscall,&p->pid,4);
```

因为 pid 是 int 型的，所以复制字节数为 4

- **proc.c / proc_pagetable**

该函数用于为一个进程创建一个用户页表，并映射必要的页面到该页表中；每个进程都有自己的页表，用于将虚拟地址映射到物理地址。

- **涉及的重要函数**
`uvmcreate()`：
用于创建一个空的页表并返回其指针
`mappages(pagetable_t pagetable, uint64 va, uint64 size, uint64 pa, int perm)`：
主要用于在页表中建立虚拟地址到物理地址的映射关系，以便于进程在访问这些虚拟地址时能够正确地访问对应的物理内存。
  - pagetable_t pagetable：页表指针，表示要操作的页表。
  - uint64 va：虚拟地址的起始地址。
  - uint64 size：需要映射的内存大小。
  - uint64 pa：物理地址的起始地址。
  - int perm：权限位，包括读、写、执行权限。

`uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)`：
修改进程的页表，删除指定的虚拟地址范围的映射，并释放对应的物理页。
`uvmfree()`：
释放页表及其引用的所有物理内存页面的函数。

- **函数执行流程**

创建完页表后，开始建立各页面的虚拟地址和物理地址间的映射关系。

```
// 映射跳板代码（用于系统调用返回）到最高的用户虚拟地址。
// 只有内核态使用它，在进入/离开用户空间时使用，因此没有 PTE_U 标志。
if(mappages(pagetable, TRAMPOLINE, PGSIZE,
            (uint64)trampoline, PTE_R | PTE_X) < 0){
    uvmfree(pagetable, 0);
    return 0;
}

// 映射 trapframe 到 TRAMPOLINE 下方，用于 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;
}
```

原先给出的代码中给出了两个页面的映射关系建立，若映射失败则解除虚拟地址与物理地址之间的映射，释放所有物理页并删除页表。

- **添加代码段**

我们只需要在上述代码段的下方仿照其过程创建 usyscall 物理地址与虚拟地址间的映射关系，并处理好映射失败后的释放即可。注意！若分配失败应该对前面映射的两个页面都进行释放。
xv6 已经在 `memlayout.h` 中定义了 USYSCALL 页面的虚拟地址:
`#define USYSCALL (TRAPFRAME - PGSIZE)`，这代表在虚拟地址中，USYSCALL 是 TRAPFRAME 前面的一个页面。
另外，要设置标志位使得该页面具有读权限（PTE_R）并且可以被用户模式（PTE_U）下的进程访问。

```
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;
}
```

- **疑问**
  我一开始以为标志位都是 0 或者 1，所以好奇为什么通过按位或可以判断各标志位的设置情况，通过查看定义，发现它们的设置如下：

  ```
  #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) // 1 -> user can access
  ```

  这样就能保证按位或的时候能有区分。

- **proc.c / freeproc**

  当一个进程终止或失败时，需要释放分配给它的所有资源，包括内存、页表和其他内核资源。 freeproc 用于释放与进程相关的资源。

  - **涉及的重要函数**
    `kfree()` ：
    用于释放先前通过 kalloc 分配的内存。
  - **对其他页的操作**

    ```
    if(p->trapframe)
        kfree((void*)p->trapframe);
    p->trapframe = 0;
    ```

  - **添加代码段**
    所以我们只需仿照上方释放陷阱帧页面的代码来释放 usyscall 页面即可。

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

- **proc.c / proc_freepagetable**

  用于释放进程页表及其所有用户内存,遍历并释放页表中的每一个用户页面，最后释放整个页表。

  - **对其他页的操作**

    ```
    uvmunmap(pagetable, TRAMPOLINE, 1, 0);
    uvmunmap(pagetable, TRAPFRAME, 1, 0);
    ```

  - **添加代码段**

    ```
    uvmunmap(pagetable,USYSCALL,1,0);
    ```

运行结果

```
init: starting sh
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
```

作业二

递归遍历页表，碰到有效的就遍历进下一层页表，打印页表内容

- 定义一个名为 vmprint() 的函数。它应该使用 pagetable_t 参数，并以下面描述的格式打印该页表。

```
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
.. .. ..509: pte 0x0000000021fdd813 pa 0x0000000087f76000
.. .. ..510: pte 0x0000000021fddc07 pa 0x0000000087f77000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
```

- 在 exec.c 中 return argc 之前插入 if(p->pid==1) vmprint(p->pagetable)，以打印第一个进程的页表，并通过 make Grade 的 PTE 打印输出测试

实验思路

输出的内容

首先了解一下要打印的内容，我们要打印的是各级页表及其内容。

- **总体结构**
  - 页表起始地址:
    `page table 0x0000000087f6e000`
    这个地址指向的是页表结构在内存中的位置。
  - 页表项 (PTE):
    - `..0: pte 0x0000000021fda801 pa 0x0000000087f6a000`
      每个条目前的 .. 表示这是一个层级结构。
      0: 表示这是页表中的第 0 项。
      pte 0x0000000021fda801 是页表项的内容。
      pa 0x0000000087f6a000 是页表项指向的物理地址。

- **逐层解释**
  - 第一层 (L1) 页表项
    - `..0: pte 0x0000000021fda801 pa 0x0000000087f6a000`
      这是一级页表中的第 0 项，pte 的值是 0x21fda801。
      该页表项指向一个物理地址 0x87f6a000，这是下一级 (L2) 页表的位置。
  - 第二层 (L2) 页表项
    - `.. ..0: pte 0x0000000021fda401 pa 0x0000000087f69000`
      这是二级页表中的第 0 项，pte 的值是 0x21fda401。
      该页表项指向物理地址 0x87f69000，这是下一级 (L3) 页表的位置。
  - 第三层 (L3) 页表项
    - `.. .. ..0: pte 0x0000000021fdac1f pa 0x0000000087f6b000`
      第 0 项 pte 值为 0x21fdac1f，指向物理地址 0x87f6b000。

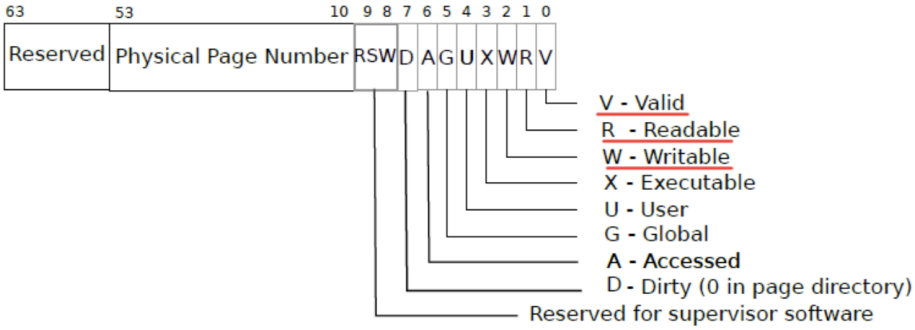
学习递归遍历页表

`freewalk()` 函数通过递归的方式逐层释放页表，所以可以先学习该函数再仿照其完成递归打印的功能。
以下是对函数的具体注释：

```
void freewalk(pagetable_t pagetable)
{
  // 一个页表中有 2^9 = 512 个页表项。
  for(int i = 0; i < 512; i++){
    // 获取当前索引 i 处的页表项。
    pte_t pte = pagetable[i];

    // 如果页表项有效且不指向物理页面，这意味着这是一个指向下级页表的 PTE
    if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
      // 将 PTE 转换为物理地址，表示下一级页表。
      uint64 child = PTE2PA(pte);
      // 递归调用 freewalk 函数释放下一级页表。
      freewalk((pagetable_t)child);
      // 递归释放完成后，将当前 PTE 清零。
      pagetable[i] = 0;
    } else if(pte & PTE_V){
      // 如果 PTE 有效但不是指向下级页表的，
      // 说明这是一个叶节点，触发一个 panic 错误。
      panic("freewalk: leaf");
    }
  }
  // 释放当前页表的内存。
  kfree((void*)pagetable);
}
```

- 页表项有效 : PTE_V 被设置
- 不指向一个物理页 : 没有 PTE_R、PTE_W 或 PTE_X 标志



实现流程

vm.c 是 xv6 操作系统的虚拟内存管理代码文件，负责管理虚拟内存和物理内存之间的映射，所以我们的打印函数应该放在该文件中。

- **vm.c / printwalk() & vmprint()**

```
void printwalk(pagetable_t pagetable, uint level) {
    char* prefix;

    // 根据level设置前缀，用于美化输出格式
    if (level == 2)
        prefix = "..";
    else if (level == 1)
        prefix = ".. ..";
    else
        prefix = ".. .. ..";

    // 遍历页表中的每一项
    for(int i = 0; i < 512; i++) { // 每个页表有512项
        pte_t pte = pagetable[i];

        // 检查页表项是否有效
        if(pte & PTE_V) { // 该页表项有效
            // 获取物理地址
            uint64 pa = PTE2PA(pte); // 将虚拟地址转换为物理地址

            // 打印当前页表项的信息
            printf("%s%d: pte %p pa %p\n", prefix, i, pte, pa);

            // 如果页表项不是叶子节点（即指向下一级页表）
            if((pte & (PTE_R|PTE_W|PTE_X)) == 0) { // 有下一级页表
                // 递归调用printwalk，处理下一级页表
                printwalk((pagetable_t)pa, level - 1);
            }
        }
    }
}
```

```
void vmprint(pagetable_t pagetable) {
    // 打印顶级页表的起始地址
    printf("page table %p\n", pagetable);

    // 调用printwalk函数，开始递归打印页表内容
    printwalk(pagetable, 2);
}
```

  - `pagetable_t` 是一个指向 64 位无符号整数数组的指针，每个 64 位无符号整数都是一个页表项（PTE）。

- xv6 使用三层页表结构：
  - 顶级页表（Level 2）：最顶层的页表，包含 512 个页表项。
  - 中间页表（Level 1）：中间层的页表，每个页表包含 512 个页表项。
  - 叶子页表（Level 0）：最底层的页表，每个页表包含 512 个页表项。

- **defs.h**

  该文件包含函数声明、宏定义以及一些通用的声明，为多个源文件提供了共享的接口和定义。
  在该文件中声明 vmprint() ,以便 exec.c 调用：

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

- **exec.c**

  该文件负责实现 exec 系统调用，exec 系统调用的作用是加载并执行一个新的可执行文件，从而替换当前进程的内存映像。这个过程会销毁当前进程的用户地址空间，并创建一个新的地址空间，其中加载了指定的可执行文件。这样，可以让当前进程执行一个新的程序。
  题目要求在 exec.c 中 `return argc` 之前插入 `if(p->pid==1) vmprint(p->pagetable)` ，以打印第一个进程的页表，照做即可。

运行结果

```
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (1.5s)
```

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
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
.. .. ..509: pte 0x0000000021fdd813 pa 0x0000000087f76000
.. .. ..510: pte 0x0000000021fddc07 pa 0x0000000087f77000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
```

总结

在完成这两个作业的过程中，我对 xv6 操作系统的内存管理机制有了更深入的了解。

- **作业一**
  - 深入理解了虚拟页面和物理页面的关系
  - 学会了如何在 xv6 中为进程分配特定的虚拟地址
  - 学会了建立虚拟地址和物理地址间的映射关系
- **作业二**
  - 进一步加深了对页表的理解
  - 学会了如何在 xv6 中实现对页表的操作和遍历
  - 了解了 exec 系统调用

通过这次实验，我加深了对操作系统和页表的理解。同时，实验反思与总结的过程也帮助我巩固了所学知识，并对下一步的学习和实践提供了方向和动力。