华东师范大学数据学院上机实践报告

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：**李芳

上机实践名称：实验三 页表　　**学号：**10214602404　　**上机实践日期：**2024.05.09

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

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`

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

三、使用环境

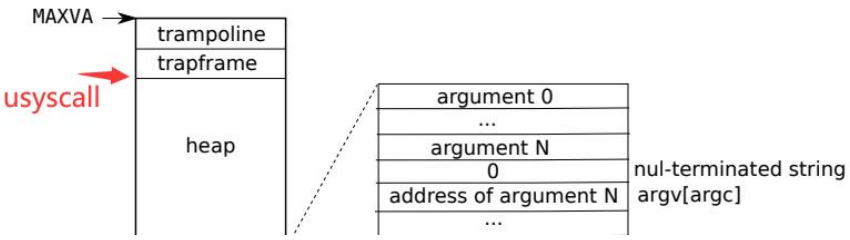
vscode

四、实验过程及结果

1. 作业一

由题干可知，这道题要给系统调用函数 ugetpid()进行提速，也就是给每个进程的单独内存空间里添加一个 USYSCALL 页面，里面存放进去 pid。这样，当 ugetpid()用到 pid 时，会在用户态直接使用 USYSCALL 页面进行直接调用，就不用再切换到内核态，节省时间。

- 首先阅读 kernel/memlayout.h 代码，可以推测出，USYSCALL 页面位于 TRAPFRAME 和 HEAP 之间，如下：



而且，在 kernel/memlayout.h 代码中 usyscall 结构体中，也能看到 int pid; 的语句。

- 要实现添加 USYSCALL 页面，需要完成两步：分配内存空间+页面映射
  - 首先，在 proc.h 文件的 proc 结构体中添加 usyscall 指针变量，它会指向共享页的物理地址

```
105   struct file *ofile[NOFILE];  // Open files
106   struct inode *cwd;           // Current directory
107   char name[16];               // Process name (debugging)
108
109   struct usyscall *usyscall;   //存储共享页物理地址
110 };
```

  - 然后，修改 proc.c 中关于进程和映射的函数：
    - 在 allocproc()函数中，分配 usyscall 物理页并初始化。假设分配失败，就释放进程资源和进程锁；分配成功时，将 pid 复制 usyscall 指针处，这样就可以在用户态直接使用 pid。

```
kernel / C proc.c / ⓕ allocproc(void)
107  allocproc(void)
124
125  found:                  // 找到情况下，进行编辑
126    p->pid = allocpid(); // 设置进程pid以及状态
127    p->state = USED;
128
129    // Allocate a trapframe page.
130    if ((p->trapframe = (struct trapframe *)kalloc()) == 0)
131    {
132      freeproc(p); // 出错，就释放进程内存，返回0
133      release(&p->lock);
134      return 0;
135    }
136
137    //为新进程分配用户系统调用区域，进程可以通过系统调用访问自己的PID
138    if ((p->usyscall = (struct usyscall *)kalloc()) == 0)
139    { //尝试为当前进程分配内存才存储usyscall(用户系统调用区域)
140      freeproc(p);  //分配失败，释放之前该进程的资源和锁
141      release(&p->lock);
142      return 0;
143    }
144    memmove(p->usyscall, &p->pid, sizeof(int)); // 分配成功，将当前进程的PID复制到
145                                                //新分配的用户系统调用区域中
146    //
147
```

    - 当在 allocproc 中进行完内存分配后，会调用映射函数 proc_pagetable()完成内存映射。所以接着修改该函数：

可以模仿前面关于 TRAPOLINE、TRAPFRAME 的映射代码。要注意的是：因为 USYSCALL 允许用户进行 read 操作，所以 mappages 最后参数使用 PTE_R | PTE_U。映射失败时，先将前面映射好的 TRAPOLINE、TRAPFRAME 页面解除映射，再将整个进程的空间内存空

间释放掉，也就是 free 掉程序所处的 pagetable。

```
kernel > C proc.c > ...
197  proc_pagetable(struct proc *p)
214    return 0;
215  }
216
217  // map the trapframe just below TRAMPOLINE, for trampoline.S.
218  if (mappages(pagetable, TRAPFRAME, PGSIZE,
219               (uint64)(p->trapframe), PTE_R | PTE_W) < 0)
220  {
221    uvmunmap(pagetable, TRAMPOLINE, 1, 0);
222    uvmfree(pagetable, 0);
223    return 0;
224  }
225
226  // 内存映射失败时，进行清理工作，取消映射，恢复页面
227  if (mappages(pagetable, USYSCALL, PGSIZE,
228               (uint64)(p->usyscall), PTE_R | PTE_U) < 0)
229  { //将USYSCALL初始的大小为PGSIZE的地址空间映射到页表中，权限为可读可写，映射失败时
230    uvmunmap(pagetable, TRAMPOLINE, 1, 0);
231    uvmunmap(pagetable, TRAPFRAME, 1, 0);
232    uvmfree(pagetable, 0);
233    return 0;
234  }
235  //
236
237  return pagetable;
238  }
```

- 当 allocproc 调用完映射函数 proc_pagetable()完成内存映射后，假设分配映射失败，就会调用 freeproc()函数来释放进程资源。所以接着修改该函数：

还是可以类比 trapframe 的操作，先保证 usyscall 的物理页内存被释放掉，然后再把 p->usyscall 也一并释放掉。

```
169  static void
170  freeproc(struct proc *p)
171  {
172    //
173    if (p->usyscall) // 释放p->usyscall对应的物理页内存
174      kfree((void *)p->usyscall);
175    p->usyscall = 0;
176    //
177
```

- freeproc 释放进程页表的内存空间时，会调用 proc_freepagetable()函数来释放页表中对应页表项，所以最后再修改该函数：

类比函数中对 TRAPOLINE 和 TRAPFRAME 中解除映射的代码，调用 uvmunmap 函数把对应 USYSCALL 区域取消映射，并释放对应的物理页面。

```
// 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); // 释放页表中对应的页表项
  //
  uvmfree(pagetable, sz);
}
```

2. 作业二

由题干可知，这道题要打印 xv6 系统第一个进程的页表的所有内容，也就是需要编写一个 vmprint()函数。当启动 xv6 时，第一个进程完成 exec()初始化后，调用 vmprint()函数输出下面这个结构类型的页表信息：

```
page table 0x0000000087f6e000 ← 先打印页表指针指向的地址
 ..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
```

要先打印标红的第一行信息，就需要找到 exec.c 文件中的 exec()函数，在各种初始化操作后，调用 vmprint()函数前，打印第一个进程第一张页表的物理地址：

kernel > C exec.c > exec(char *, char **)

```
12  int exec(char *path, char **argv)
112 {
113   // Commit to the user image.
114   oldpagetable = p->pagetable;
115   p->pagetable = pagetable;
116   p->sz = sz;
117   p->trapframe->epc = elf.entry; // initial program counter = main
118   p->trapframe->sp = sp;         // initial stack pointer
119   proc_freepagetable(oldpagetable, oldsz);
120
121   //
122   if (p->pid == 1)
123   { //打印第一个进程的页表
124     printf("page table %p\n", p->pagetable);
125     vmprint(p->pagetable, 1);
126   }
127   //
128
129   return argc; // this ends up in a0, the first argument to main(argc, argv)
130
131 bad:
```

- 因为 exec()函数调用了用来打印页表信息的 vmprint()函数，所以在 vm.c 文件中添加该函数：

- hints 中提到了可以从 freewalk 函数中获取提示，根据 freewalk 函数可知：

先使用 for 循环来遍历一张页表中的 2^9=512 个页表项。在循环中，检查每条页表项是否指向另一个页表：(pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0），即页表项是否有效以及页表项是否不可读、不可写、不可执行。如果当前页表项指向另一个页表，就**使用 PTE2PA 宏来获取该页表项指向的物理地址，根据该物理地址递归调用 freewalk()函数**，然后清除该页表项；如果页表项有效但无效，就抛出 panic 异常。最后，所有页表项处理完毕后，使用 kfree 释放整个页表占用的内存。

- 同理，vmprint()函数也可以使用遍历递归的思想：

首先，接受两个参数：指向页表的指针+当前页表级数；然后循环遍历一整页的所有页表项，循环中，先判断页表项标志位：如果页表项有效且指向另一页表项，就获取页表项指向的物理地址，打印每级对应的" .. "缩进，再打印当前页表项的索引、虚拟页地址、物理地址信息，接着根据页表项指向的物理地址递归调用下一级页表的 vmprint 函数；如果页表项有效但不指向下一页表项，就只打印，不递归调用。

kernel > C vm.c > copyinstr(pagetable_t, char *, uint64, uint64)

```
450 void vmprint(pagetable_t pagetable, int level)
451 {
452   for (int i = 0; i < 512; i++)
453   { //遍历页表项以打印
454     pte_t pte = pagetable[i];
455     if ((pte & PTE_V) && (pte & (PTE_R | PTE_W | PTE_X)) == 0)
456     { //检查当前页表项有效且无读、写、执行权限
457       uint64 child = PTE2PA(pte);  //当前页表项转换为对应物理地址
458       for (int j = 0; j < level; j++)  // 对应页表的级数，输出缩进
459       {
460         printf(" ..");
461       }
462       printf("%d: pte %p pa %p\n", i, pte, child);  //输出当前页表项的索引、内容、物理地址
463       vmprint((pagetable_t)child, level + 1);  //递归
464     }
465     else if (pte & PTE_V)
466     { //当前页表项有效且具有读、写、执行权限
467       uint64 child = PTE2PA(pte);
468       for (int j = 0; j < level; j++)
469       {
470         printf(" ..");
471       }
472       printf("%d: pte %p pa %p\n", i, pte, child);
473     }
474   }
475 }
```

- 最后，在 defs.h 文件中 vm.c 模块中声明新添加的 vmprint 函数：

kernel > C defs.h > ...

```
156  // vm.c
157  void            kvminit(void);
158  void            kvminithart(void);
159  void            kvmmap(pagetable_t, uint64, uint64, uint64, int);
160  int             mappages(pagetable_t, uint64, uint64, uint64, int);
161  pagetable_t     uvmcreate(void);
162  void            uvminit(pagetable_t, uchar *, uint);
163  uint64          uvmalloc(pagetable_t, uint64, uint64);
164  uint64          uvmdealloc(pagetable_t, uint64, uint64);
165  int             uvmcopy(pagetable_t, pagetable_t, uint64);
166  void            uvmfree(pagetable_t, uint64);
167  void            uvmunmap(pagetable_t, uint64, uint64, int);
168  void            uvmclear(pagetable_t, uint64);
169  uint64          walkaddr(pagetable_t, uint64);
170  int             copyout(pagetable_t, uint64, char *, uint64);
171  int             copyin(pagetable_t, char *, uint64, uint64);
172  int             copyinstr(pagetable_t, char *, uint64, uint64);
173
174  void            vmprint(pagetable_t,int); //加入vmprint函数原型
175
```

3. 实验结果截图

make 之后，运行 grade-lab-pgtbl 可执行文件进行实验任务测试，输出截图如下：

```
lifang@lifang-virtual-machine:~/xv6-labs-2021$ ./grade-lab-pgtbl ugetpid
make: “kernel/kernel”已是最新。
== Test pgtbltest == (7.7s)
== Test   pgtbltest: ugetpid ==
  pgtbltest: ugetpid: OK
lifang@lifang-virtual-machine:~/xv6-labs-2021$ ./grade-lab-pgtbl pte print
make: “kernel/kernel”已是最新。
== Test pte printout == pte printout: OK (2.3s)
```

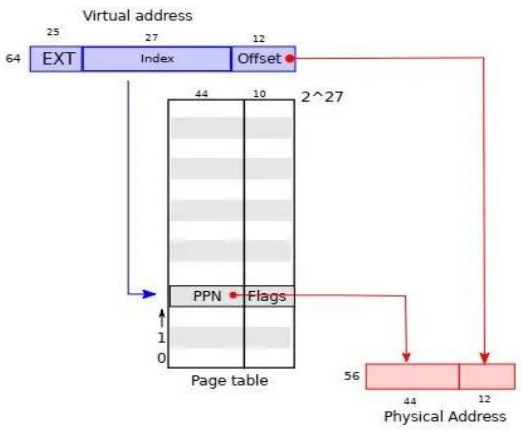
五、总结

1. 过程复盘

在 xv6 中，采用**分页**手段来构建虚拟内存，页表用来完成虚拟地址到物理地址的转换。

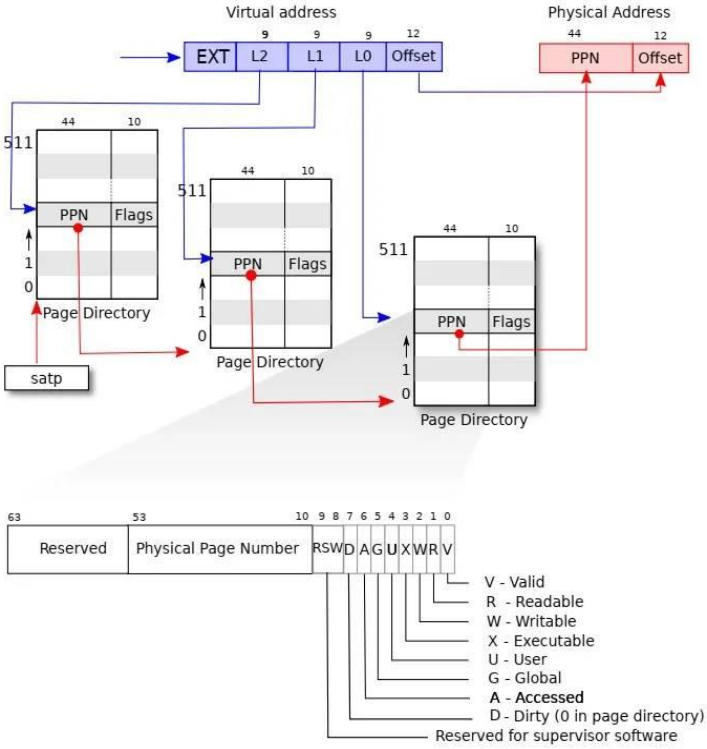
- RISC-V 指令：

用来处理 64 位的虚拟地址，将每个虚拟地址映射到对应物理地址，页表条目 PTE 用来保存其中的映射关系，下图是 RISC - V 的对应映射关系图：



因为虚拟地址的低 39 位被使用，高 25 位被保留，且使用的 39 位中的高 27 位作为 PTE 的索引，所以一张页表的容量是 2^27 个 PTE，剩下的 12 位作为页内偏移量，所以一个分页大小为 2^12B=4KB。每个 PTE 有 54 位，前 44 位作为 PPN，后 10 位作为标志位，使用 uint64 类型。虚拟地址转换成物理地址时：先拿出虚拟地址有效的 39 位，利用高 27 位索引 PTE，从 PTE 中得到 44 位 PPN，利用 PTE 的标志位检查权限后，将 PPN+偏移量作为物理地址，访问物理内存。

上面是一级页表的对应解析，但 RISC - V 实际采用多级页表，一共有三级，是 task2 打印出来的类似树形的结构，具体映射如下图：



此时，虚拟地址低 39 位有效位的高 27 位平均分为 3 份，每份九位作为每级页表 PTE 的索引，所以查找过程要遍历三层，只有三层都命中，才能找到 PPN，通过 PPN+偏移量获取物理地址访问内存；有一级页表有不命中情况就会抛出缺页错误。

- PTE 标志位代表信息：

PTE_V：PTE 是否存在/有效。如果不存在，尝试引用该页时就会抛出缺页错误异常。

PTE_R：物理帧是否能被读。

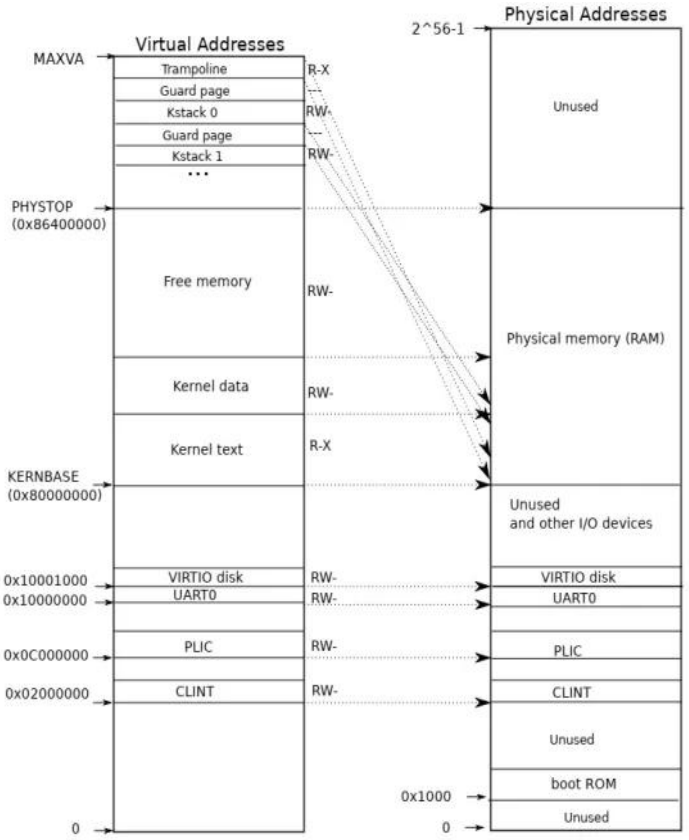
PTE_W：物理帧是否能被写。

PTE_X：物理帧是否能被 CPU 看待并转换成指令来执行。

PTE_U：物理帧在用户模式下是否能访问。如果没有置位，则该一页物理帧只能在监管者模式下被访问。

- 所有有关 RISC - V 相关的定义、常量、宏等在 kernel/riscv.h 文件中定义，该文件中的代码可以使 xv6 与 RISC - V 进行兼容，进行地址转换等功能。

- xv6 系统内核虚拟空间与实际物理地址空间之间的映射如下：

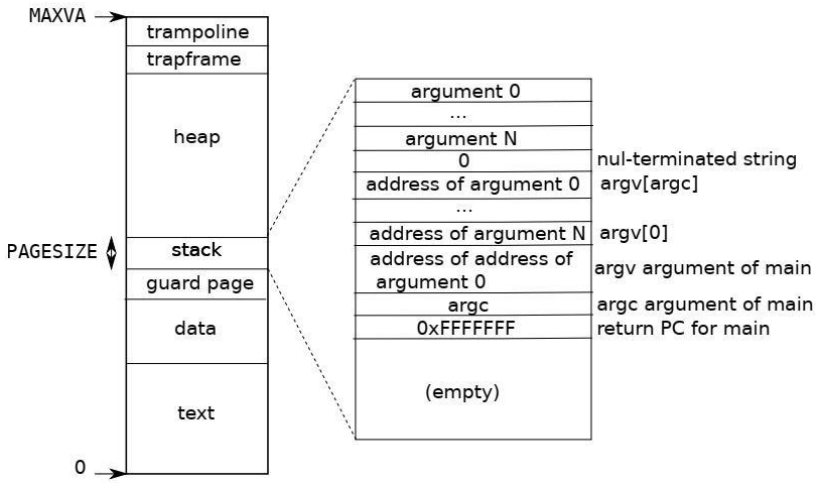


- xv6 内核在启动阶段时，页表被禁用。为了使 RISC-V 的分页硬件开始使用页表，内核要先将根页表的物理地址写入 satp 寄存器中。每个 CPU 都有自己的 satp 寄存器，CPU 在执行指令时，使用自己 satp 寄存器里指向的根页表，完成指令中虚拟地址的转换，所以不同 CPU 可以使用不同的页表，看到的进程虚拟内存视图是不同的。

- 在内核没使用页表时，虚拟内存采用直接映射方式转换为物理地址，也就是二者相同，相关的代码在 kernel/vm.c 中定义的 kvminit。在初始化内核页表并启用后，就要使用到上图左上部分，位于虚拟内存顶部，在 memlayout.h 中最先被定义，包括 TRAMPOLINE 以及内核栈，这两部分设置 PTE_R 和 PTE_X，代表可读且可直接被当作指令运行。

- Trampoline 页：如图所示被映射到虚拟地址空间的顶端，用户和内核都有这部分的映射，映射位置相同，会被映射两次，一次映射到虚拟空间最上面，一次是直接映射。内核可以将 trampoline 页映射到用户虚拟地址空间的顶端，所有进程都可以看到这一页。
- 内核栈：靠近 PHYSTOP 处开始往下分配，会被直接映射到内核的虚拟地址空间，之间会插入一些保护页（PTE_V 被设置为无效，防止内核栈溢出。

- 用户进程的虚拟地址空间如下：



每个用户进程都有自己的页表，对应不同的虚拟地址空间。用户进程的虚拟地址空间同样从 0 开始，一直到 MAXVA（2^38-1），这其中有多达 256GB 的空间。用户进程在运行时需要额外内存，就向内核内存分配器发出请求，kalloc 分配一些物理页，然后内核更新用户进程的页表，设置新的 PTE。进程与进程之间未使用的虚拟地址，在页表中将对应 PTE 标记为无效。

用户进程虚拟地址空间布局与内核相比，会有更多细节（尤其用户栈）。图中用户栈的初始内容是由系统调用 exec 产生的。初始用户栈包含各命令行参数的字符串以及指向各命令行参数的指针数组 argv[ ]（这样就可以调用 main 函数返回的其它参数）。在初始化用户栈完成后，用户程序返回、开始执行 main 函数。

保护页：用户栈之间的保护页和内核栈之间的保护页有不同：用户的有实际的物理帧对应，标志位为 RWV，没有 U；内核栈之间的保护页 PTE_V 无效，没有实际分配物理页。

- kernel/mamlayout.h：

定义 xv6 操作系统中的虚拟内存布局，包括内核和用户空间的分布，以及一些硬件设备在物理内存中的地址映射关系的代码在 kernel/mamlayout.h 文件中。在该文件中，BERNBASE 定义为

0x80000000，PHYSTOP 定义为 0x88000000，还包含了一些其他关键的数据结构和常量，描述内核和用户空间的布局以及内存映射的细节。

xv6 的 QEMU 模仿的物理内存空间从 KERNBASE 开始，至少到 PHYSTOP；一些 I/O 设备接口也要通过内存映射将设备寄存器映射到物理内存中，在 HERNBASE 下面。

- Kernel/main.c：

  操作系统的入口，其中的 main()函数是操作系统内核的入口函数，会在 CPU 上以监管模式跳转到 main 处执行，实现内核的初始化工作和多处理器同步，确保所有 CPU 在正确时机进行初始化，最后启动进程调度。

  初始化工作会调用 kinit()函数以及 kvminit()函数，两者分别位于 kalloc.c 和 vm.c 中，还有其他一系列初始化的函数，kvminithart()位于 vm.c 中，procinit()位于 proc.c 中等等。

- kernel/vm.c：
  - 存储实现内核和用户虚拟内存管理的代码。核心数据结构：pagetable_t，页表指针，uint64*类型，指向存放 RISC-V 跟页表的一页，可以是内核根页表或用户进程根页表。
  - 内核：
    - Kvminit()函数：刚才在 main.c 部分提到该函数，它将调用 kvmmake()函数进行创建内核页表。
    - Kvmmake()函数：负责进行内核页表的创建，并调用 kvmmap()函数进行映射建立。首先分配新一页物理帧装载内核根页表，再调用 kvmmap，建立直接映射，包括 I/O 设备、内核代码和数据、内核空闲内存段等，最后把 trampoline 映射到内核虚拟地址的最上面。到这完成了内核页表的初始化。
    - Kvmmap()函数：直接转交非常重要的 mappages()函数，来建立映射项。
    - Kvminithart()函数：进行内核页表装载，是上面提到的有关 satp 寄存器的操作。该函数将内核根页表的物理地址写入 satp 寄存器，CPU 进行地址转换之后，开始将下一条指令的虚拟地址映射到物理地址空间。

    在这个函数最后：sfence_vma();语句是 RISC-V 中提供的指令，用来刷新或者清空当前 CPU 的 TLB，每次 satp 中更新页表的时候，都会刷新 CPU 的 TLB。

- 用户：
  - Uvmalloc()函数：调用 kalloc 来分配物理内存，调用 mappages 来更新页表，PTE 的 5 个标志位都置 1。
  - Uvmdealloc()函数：回收已分配物理内存。
  - Uvmunmap()函数：使用 walk 找到相应的 PTE，并且调用 kfree 回收相应的物理帧。在 xv6 中，用户进程的页表不仅告诉了分页硬件如何转换虚拟地址，也记录着哪些物理帧被分配给该用户进程，因此在回收分配给用户的物理帧之前，应该先检查相关的 PTE 是否存在/有效。
- Mappages()函数：接收参数为页表，需要建立映射关系的 va 和 pa，映射的范围大小和 PTE 的权限。期间会调用 walk()函数进行最后一级页表的 PTE 查找（因为最后一页页表项一般直接指向物理地址），假设 PTE 没有被占用（有效位为 0 的页表项），就输入 pa 和 PTE 权限，更新该页表项设置有效位，建立起新的映射，重复这种操作，直到所有映射建立完成。
- Walk()函数：为输入的 va 找到对应的最后一级 PTE。它模仿 RISC-V 硬件的行为，使用软件实现，在页表还没有完成初始化之前，帮内核完成页表查找。
  - 根据最上面解释的 RISC-V 的操作原则：每次都通过 va 中的 9 位来索引某一级页表的相应 PTE。如果 PTE 指示我们，下一级页表不存在且 alloc 设置为 1，walk 就会请求为下一级页表分配新的一页，并且更新该 PTE。如此遍历，walk 最后返回的是最后一级页表的 PTE，且路径上经过的三级页表都一定已经被分配了物理帧，并被建立起来。
  - 如果传入的是新 va，相关的映射未被建立，那么 walk 只会建立第二级和第三级页表（根页表是已经存在的），为它们分配新的物理帧，walk 并没有为最后一级页表的 PTE 所指向的物理帧分配新的一页，假设这个映射是新的，通过 walk 返回的 PTE 是无效的（全 0），如果原来就有这个映射，那么 walk 就返回包含映射内容的 PTE。

- kernel/kalloc.c：

内核的内存分配，用来管理物理内存空间。基本结构体就是 kmem，设置了 freelist 以及保护它的自旋锁；struct run 串联每一页物理帧，被保存在每个空闲物理页内部。

- Kinit()函数：刚才提到的 main.c 会调用 kinit 完成内存分配器初始化。先初始化自旋锁，再根据

起始和结束地址调用 freerange()函数。

- Freerange()函数：对已分配的物理内存进行回收，对输入参数范围内的每一物理页都进行 kfree()，并把 free 掉的空物理页挂到 freelist 上，其中要 PGROUNDUP（）对齐地址。
- Kfree()函数：回收一页已经分配的物理帧。因为分配器 freelist 开始是空的，所以调用完 kfree 才会有空闲物理页进入空闲链表。操作时，先把被回收的物理页通过 memset()把所有字节置 1（原本拥有这页物理帧的进程再次读取它时判断为垃圾数据）。接着 pa 指针转换成 struct run* 类型，使用头插法插入 free-list。
- Kalloc()函数：进行一页物理页的分配，选取 freelist 的第一个空闲物理页分配。因为，调用 freerange 时传入的起始地址是 end，终止地址是 PHYSTOP，freerange 按照物理地址从小到大的顺序调用 kfree，kfree 使用头插法插入 free-list，证明链表的前面存的是物理地址大的空闲物理页，所以 kalloc 分配物理帧时，是按从大的物理地址开始往下分配的。

- Kernel/exec.c：
  - 实现系统调用 exec，存储在文件系统上，负责将新的用户程序装载进内存里并执行，以及新用户进程的虚拟地址空间的建立。这也是为什么 task2 在 exec 函数中进行 vmprint 函数调用的原因。
  - exec 通过路径名打开文件，然后尝试读取该文件的 ELFHeader。
    - 所以，第一步，exec()函数会先检查打开文件的 ELF 二进制文件；这之后，调用 proc.c 文件中存储的 proc_pagetable()函数，这个函数又调用 uvmcreate()函数，来创建一个空的用户页表，先在空闲页表添加 TRAMPOLINE 和 TRAPFRAME 映射（任务一，还要映射 SYSCALL）；
    - 然后，对每个程序段先调用 uvmalloc()函数分配物理帧，更新用户页表，再调用 loadseg 函数加载对应程序段到物理帧中；加载用户空间的 text 和 data。
    - 接着，分配初始化用户栈：分配两页物理帧：
      - 第一页是保护页：调用 uvmclear()函数将 PTE_U 设置为无效，用户空间下无法访问。
      - 第二页是用户栈：从栈顶开始，将命令行参数，还有一些其他参数压入用户栈。
    - 最后，这些初始化和加载工作完成后，就清除旧进程的内存映像，释放旧页表占用的物理内

存，准备使用新页表，最后返回后，该进程会执行新用户程序。

- Loadseg()函数：负责加载程序段到物理帧中，首先将传入的参数 va 传给 walkaddr()，walkaddr()又通过 walk()查找相关 PTE，将 va 转换为 pa，并返回 uvmalloc()分配的物理帧的物理地址；最后再调用 readi()，将程序段加载到物理内存中。

2. 总结

这次实验主要针对 xv6 操作系统中的页表相关代码进行操作，在助教讲解完 xv6 页表相关的基础知识后，我又自己巩固了页表的实现机制以及用户拷贝数据到内核态的方法。之后根据实验的一些 hints 完成了两个作业：

作业一的目标是加速 getpid() 系统调用，作业二则要求定义一个名为 vmprint() 的函数，用于递归遍历页表并打印页表内容。

通过本次实验，我深入理解了 xv6 内核中有关页表、虚拟内存、物理内存的代码，如 vm.c、kalloc.c、proc.c 等等文件具体函数、结构体的作用，以及 xv6 操作系统中用户空间虚拟内存及其映射、内核空间虚拟内存及其映射的详细细节以及两者的异同点。另外，通过 task1 与助教的答疑，我明白了这里能“加速”的具体原因是什么，本质就是减少用户空间和内核空间的相互切换，减少时间。

在未来的学习中，我将继续深入 xv6 操作系统的代码，每次都研究透其中的实现原理。