同濟大學
TONGJI UNIVERSITY

MIT 6.S081 实验报告

Lab 3

2151767 薛树建

日期：2023 年 7 月 20 日

目录

Lab3: page tables

1. Speed up system calls (easy)

1) 实验目的

在 linux 操作系统中，可以通过共享只读的页来加速系统调用。这样的话，内核在执行某些系统调用的时候就可以直接在用户态进行(大多数系统调用可能都是仅仅去访问内核态的一些数据)，而不必去进行上下文切换。

在本实验中，要求我们通过上述的方式来加速系统调用 getpid 系统调用。大致思路就是在进程 proc 结构体中添加一个变量 usyscall 的指针（其记录这本进程的进程号，因而我们此次实验加速的系统调用便是 getpid，其作用时获取当前进程的 id)。同时我们需要在该进程的虚拟地址空间中开辟一个部分(USYSCALL)用来映射物理页，然后为进程块 proc 中的 usyscall 指针的分配一个物理页上，将该物理页映射到 USYSCALL 的虚拟地址上，然后用户态就可以直接访问 USYSCALL 来获取进程 id 了。

2) 实验步骤

1. 首先实验已经给我们定义 USYSCALL 的值与需要用到的结构体 usyscall

```
#define USYSCALL (TRAPFRAME - PGSIZE)

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

2. 接着我们需要在进程块中加入一个该结构体的指针，用它来记录该进程的进程号，同时在 allocproc 函数中对其进行初始化，并为其分配一个物理页。

```
  char name[16];               // Process name (debugging)
  struct usyscall *usersyscall;   // 记录进程号的变量
};
```

```
  p->context.ra = (uint64)forkret;
  p->context.sp = p->kstack + PGSIZE;
  p->usersyscall->pid=p->pid;
  return p;
}
```

```
//分配一个页
if((p->usersyscall=(struct usyscall*)kalloc())==0){
  freeproc(p);
  release(&p->lock);
  return 0;
}
```

3. 接下来修改 proc_pagetable 函数来为将该物理页映射到 USYSCALL 上，如果映射失败，则应该解除映射并释放页表。

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

4. 最后需要注意在 freeproc 与 proc_freepagetable 中将分配的物理页（物理内存）与建立的映射给释放掉。

```
if(p->usersyscall)//释放usersyscall
  kfree((void*)p->usersyscall);
p->usersyscall = 0;
```

```
void
proc_freepagetable(pagetable_t pagetable, uint64
{
  uvmunmap(pagetable, TRAMPOLINE, 1, 0);
  uvmunmap(pagetable, TRAPFRAME, 1, 0);

  uvmunmap(pagetable, USYSCALL, 1, 0);

  uvmfree(pagetable, sz);
}
```

5. 最后可以看到用户态可以访问呢 USYSCALL 来获取进程号

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

6. 测试通过。

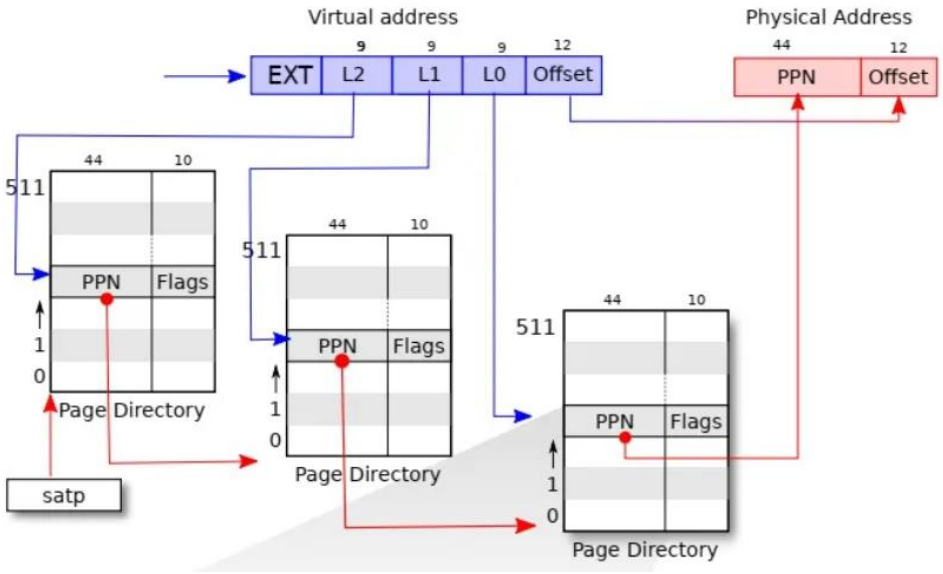
```
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgaccess_test: OK
pgtbltest: all tests succeeded
$
```

3) 实验中遇到的问题和解决方法

本实验主要关于如何去加速一个系统调用（本实验中为 getpid），实际上便是通过构建类似那种 trapframe 与 trampoline 那样的在内核与用户之间共享的页面，本实验中制定了 USYSCALL，其为 TRAPFRAME-PGSIZE，大小为一页。利用该页在进程初始化阶段将一些内核中的数据存放在此处，用户如果需要这些数据，便不需要进入内核态，而是直接通过 USYSCALL 来访问相应的物理内存。

本实验中遇到的主要问题是对页表与虚拟地址空间之间的联系分不清楚，通过阅读文档的已解决。

这时 xv6 的虚拟地址 va，页表与物理地址 pa 之间的关系图。



采用了三级页表结构，页表目录的起始地址存放在 satp 寄存器中，通过 va 中的 9 位偏移量寻址，得到之后在进入下一级页表，最后一级页表的 pte 项的内容先右移 10 位（去除 flags 部分），之后左移 12 位与 12 位 offset 相加得到物理地址（offset 12 位其实正好对应了 4k 大小的物理页）。

虚拟地址则对应一个进程，用户空间与内核空间位于上面，同时也有一些内核与用户共享的部分，如 trapframe 等。

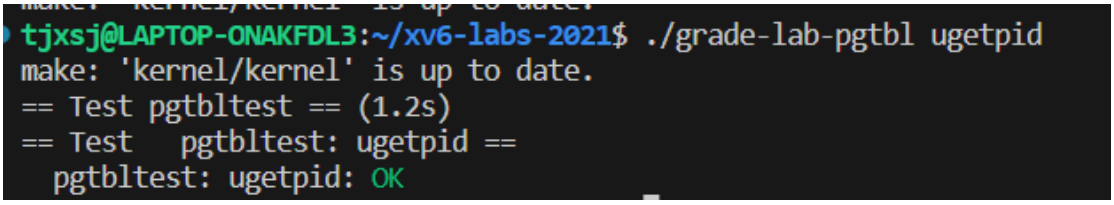
不过 xv6 中定义的 MAXVA 为(1<<38)，与上图中的 39 位虚拟地址相比少了一位，也就是实际虚拟地址空间并不能完全覆盖，不太清楚为什么会这样定义。

4) 实验心得

在完成页表实验的第一个实验之后，我本人对应进程与虚拟地址空间，还有虚拟地址与物理地址之后的关系有了更加一步的认知。

让我对操作系统理论课上老师讲的这一部分有了较为细致的理解。

评分如下：



2. Print a page table (easy)

1) 实验目的

本实验目的是让我们打印一个页表，定义一个名为 vmprint()的函数。它接收一个 pagetable_t 作为参数，并按照相应的格式打印该页表。

2) 实验步骤

根据 hints，我们可能需要去参考一下 freewalk 的实现

a) 首先查看 freewalk 函数的实现，采用了递归的方式去释放页表项。Pagetable_t 其实就是一个指针，其中储存的是地址。我们便可以仿照这种做法去递归打印页表

```
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);
}
```

b) 采用递归的方式来打印页表，通过 PTE_V 标志来查看该页表项是否有效，根据其页表项的等级来打印对应数量的'..'，pte 与 pa。如果不是最后一级页表，还需要去递归打印。

```
void vmprint2(pagetable_t pagetable,int level){//打印对应等级的页表,
  for(int i = 0; i < 512; i++){
    pte_t pte = pagetable[i];
    if(pte & PTE_V){//判断该条目是否有效
      uint64 child = PTE2PA(pte);//转换为物理地址
      for (int j = 0; j <=level; j++) {//根据页表的级别按格式打印
        if (j > 0) {
          printf(" ");
        }
        printf("..");
      }
      printf("%d: pte %p pa %p\n", i, pte, child);
      if (level < 2)
        vmprint2((pagetable_t)child, level+1);
    }
  }
```

c) 最后在写一个函数来调用该递归函数，同时打印一些提示信息。在 defs.h 中声明该函数。

```
void vmprint(pagetable_t pagetable){

  printf("page table %p\n", pagetable);//%p用来打印指针的值
  vmprint2(pagetable, 0);
}
```

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

d) 最后我们在 exec 中调用该函数，以便在启动 xv6 是可以打印第一个进程的页表

```
proc_freepagetable(oldpagetable, oldsz);
if(p->pid==1) vmprint(p->pagetable);//调用vmprint
```

e) 输出如下：

```
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
init: starting sh
$
```

3) 实验中遇到的问题与解决方法

本实验在上一个实验的基础上还是比较容易完成，主要便是注意对 xv6 的多级页表机制，采用递归的方式去打印所有页表项。

4) 实验心得

通过本实验，我对于 xv6 操作系统的多级页表映射有了一定的了解，对于页表的作用与实现方式也有了一定的了解。

通过完成页表打印的实现，我个人的代码能力也得到了一定的提升。

本实验评分如下：

```
tjxsj@LAPTOP-ONAKFDL3:~/xv6-labs-2021$ ./grade-lab-pgtbl print
make: 'kernel/kernel' is up to date.
== Test pte printout == pte printout: OK (0.8s)
```

3. Detecting which pages have been accessed(hard)

1) 实验目的

垃圾回收器功能的实现则需要去得到该页面自上次检查之后是否被访问的信息，本实验目的是去实现一个系统调用来实现这种方式。

根据实验提示，可以利用 pte 中的 PTE_A 来查看该页表像是否被访问过

2) 实验步骤

a) 此系统调用的调用号已经一些相关的操作已经被实现好了我们只需要在 kernel/sysproc.c 添加 sys_pgaccess 的代码即可。该函数的参数有三个，分别是待检查的虚拟地址，待检查页数量与一个指向用户空间缓冲区的地址。我们最终需要向这个缓冲区中写入结果

b) 首先利用 argadd 与 argint 函数来获取相关参数

```
if (argaddr(0, &a) < 0)
  return -1;
if (argint(1, &num) < 0)
  return -1;
if (argaddr(2, &mask) < 0)
  return -1;
```

c) 由于我们需要按照页，一页一页的向后遍历，因而需要先利用 PGROUNDDOWN 函数来获取对齐 va 的虚拟地址。接着从该该地址向后遍历 num 个页。每次通过 walk 函数来获取其 pte 项，最后通过检查 pte 中的 PTE_A 来检查其是否已经被访问过，并记录结果。同时要注意将遍历过的 pte 的 PTE_A 项置 0.

```
uint64 start = PGROUNDDOWN(a);
for (int i = 0, va = start; i < num; i++, va += PGSIZE){
  pte = walk(myproc()->pagetable, va, 0);
  if (*pte & PTE_A) {
    result=result|1<<i;
    *pte = (*pte)&(~PTE_A);
  }
}
```

d) 我们还需要在 riscv.h 中声明 PTE_A，以方便后续函数调用

```
#define PTE_U (1L << 4) // 1 -> user can access
#define PTE_A (1L << 6)//accessed
```

e) 通过 copyout 函数将结果复制到用户缓冲区中

```
if (copyout(myproc()->pagetable, mask, (char*)&result, sizeof(result)) < 0) {
  return -1;
}
```

f) 结果如下：

```
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgaccess_test: OK
pgtbltest: all tests succeeded
```

3) 实验中遇到的问题与解决方法

本实验中遇到的主要问题是如何将结果表示成为 bitmap,在网上借鉴了相关方法。

先定义一个 uint64 的变量，每次循环找到 PTE_A 不为 0 的 pte 之后，将 1 左移 i 位并与该变量按位与，最终在复制到用户区时，取该变量的地址并将其转化为 char*指针作为参数传递。

4) 实验心得

通过本实验，我对于 xv6 操作系统的页表部分的 walk 函数（得到 va 对应的最后一级页表项）与 pte 页表项的 flags 部分有了更加深入的了解。同时也对 bitmap 有了进一步的了解。

本实验评分如下：

```
tjxsj@LAPTOP-ONAKFDL3:~/xv6-labs-2021$ ./grade-lab-pgtbl access
make: 'kernel/kernel' is up to date.
== Test pgtbltest == (1.6s)
== Test   pgtbltest: pgaccess ==
  pgtbltest: pgaccess: OK
```

总结

通过这次试验，我对于操作系统中的页表有了更加深入的了解，同时也包括相关的虚拟地址空间等部分，在平时上课时，感觉页表这个结构挺简单，但实际去看了代码之后才发现其复杂程度。

整体评分如下：

```
tjxsj@LAPTOP-ONAKFDL3:~/xv6-labs-2021$ ./grade-lab-pgtbl
make: 'kernel/kernel' is up to date.
== Test pgtbltest == (1.0s)
== Test   pgtbltest: ugetpid ==
  pgtbltest: ugetpid: OK
== Test   pgtbltest: pgaccess ==
  pgtbltest: pgaccess: OK
== Test pte printout == pte printout: OK (0.9s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test usertests == (94.2s)
== Test   usertests: all tests ==
  usertests: all tests: OK
== Test time ==
time: OK
Score: 46/46
```