

Lab3: Page tables

2351882 王小萌

Tongji University, 2025 Summer

代码仓库链接: <https://github.com/wmxmw/Tongji-University-xv6-labs-2021.git>

1.Speed up system calls

1.1 实验目的

1.2 实验步骤

1.3 实验中遇到的问题和解决办法

1.4 实验心得

2.Print a page table

2.1 实验目的

2.2 实验步骤

2.3 实验中遇到的问题和解决办法

2.4 实验心得

3 Detecting which pages has been accessed

3.1 实验目的

3.2 实验步骤

3.3 实验中遇到的问题和解决办法

3.4 实验心得

4 实验检验得分

1 Speed up system calls

1.1 实验目的

某些操作系统（例如 Linux）通过在用户空间和内核之间共享一个只读区域的数据来加速某些系统调用。这消除了在执行这些系统调用时进入内核的需求。为了学习如何将映射插入页表，第一个任务是在 xv6 中为 getpid() 系统调用实现此优化。

1.2 实验步骤

1. 要修改 kernel/proc.c 中的 allocproc() 函数，并在分配 trapframe 后面添加分配 usyscall 页面的部分。

```
105     struct file *ofile[NOFILE]; // Open files
106     struct inode *cwd;           // Current directory
107     char name[16];               // Process name (debugging)
108     struct usyscall *usyscall;   // usyscall page address
109 };
110
```

```

29
30 //完成共享页得分配与初始化
31 if((p->usyscall = (struct usyscall *)kalloc()) == 0){
32     freeproc(p);
33     release(&p->lock);
34     return 0;
35 }
36 p->usyscall->pid = p->pid;
37

```

2. 在 memorylayout.h 将当前进程的 pid 存入 usyscall 页面的开始处:

```

73 #ifndef LAB_PGIBL
74 #define USYSCALL (TRAPFRAME - PGSIZE)
75
76 struct usyscall {
77     int pid; // Process ID
78 };
79 #endif
80

```

3. 在 kernel/proc.c 文件中的 proc_pagetable() 函数中利用 mappages() 函数来实现。我们需要在这个函数中添加一个 usyscall 页面的映射。

```

95 }
96 if(mappages(pagetable, USYSCALL, PGSIZE,
97     (uint64)(p->usyscall), PTE_U | PTE_R) < 0){
98     uvmfree(pagetable, 0);
99     return 0;
100 }
101 return pagetable;

```

4. 还需要在必要的时候释放页面。终止进程时，需要在 kernel/proc.c 的 freeproc() 函数中，将我们分配的 usyscall 和 trapframe 页面做相同处理，添加相关代码。

```

161 if(p->trapframe)
162     kfree((void*)p->trapframe);
163 p->trapframe = 0;
164 if(p->usyscall)
165     kfree((void*)p->usyscall);
166 p->usyscall = 0;
167 if(p->pagetable)
168     proc_freepagetable(p->pagetable, p->sz);
169 p->pagetable = 0;

```

5. 在 kernel/proc.c 的 proc_freepagetable() 函数中释放我们之前建立的虚拟地址到物理地址的映射:

```

220 proc_freepagetable(pagetable_t pagetable, uint64 sz)
221 {
222     uvmunmap(pagetable, TRAMPOLINE, 1, 0);
223     uvmunmap(pagetable, TRAPFRAME, 1, 0);
224     uvmunmap(pagetable, USYSCALL, 1, 0);
225     uvmfree(pagetable, sz);
226 }

```

6. 编译运行，运行 pgtbltest，结果如下：

```

xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgtbltest: pgaccess_test failed: incorrect access bits set, pid=3
$

```

7. 单项评分：

```

vale@Puppyyoo:~/xv6-labs-2021$ ./grade-lab-pgtbl ugetpid
make: 'kernel/kernel' is up to date.
== Test pgtbltest == (0.0s)
== Test pgtbltest: ugetpid ==
pgtbltest: ugetpid: OK
vale@Puppyyoo:~/xv6-labs-2021$

```

1.3 实验中遇到的问题和解决办法

问题：对已经释放或未正确初始化的页表项进行操作，导致异常。

解决办法：仔细检查页表的分配和释放逻辑，确保在分配页表时正确初始化各个页表项。最终发现问题 出在 proc_freepagetable() 函数中，没有正确处理 usyscall 页面的释放。要在处理完页面的释放后再释放进程，顺序关键。

```

161     if(p->trapframe)
162     | kfree((void*)p->trapframe);
163     p->trapframe = 0;
164     if(p->usyscall)
165     | kfree((void*)p->usyscall);
166     p->usyscall = 0;
167     if(p->pagetable)
168     | proc_freepagetable(p->pagetable, p->sz);
169     p->pagetable = 0;

```

1.4 实验心得

在本次实验中，通过实现将 usyscall 页面映射到用户空间并优化 getpid() 系统调

用，我们深入了解了如何在 xv6 操作系统中处理虚拟地址和物理地址的映射。这不仅提升了系统调用的效率，还加深了我们对操作系统内核机制的理解。

2 Print a page table

2.1 实验目的

1. 编写一个函数来打印页表的内容。定义一个名为 `vmprint()` 的函数。它应该接收一个 `pagetable_t` 参数，并按照描述的格式打印该页表。
2. 在 `exec.c` 中，在 `return argc` 之前插入 `if(p->pid==1) vmprint(p->pagetable)` 语句，以打印第一个进程的页表，不输出无效的 PTE。

2.2 实验步骤

1. 在 `kernel/vm.c` 中编写函数 `vmprint()`。可以参考 `freewalk()` 函数的实现。

```
436 // print page tables lab3-1
437 void vmprint(pagetable_t pagetable) {
438     printf("page table %p\n", pagetable);
439     // range top page dir
440     const int PAGE_SIZE = 512;
441     // 遍历最高级页目录
442     for (int i = 0; i < PAGE_SIZE; ++i) {
443         pte_t top_pte = pagetable[i];
444         if (top_pte & PTE_V) {
445             printf("..%d: pte %p pa %p\n", i, top_pte, PTE2PA(top_pte));
446             // this PTE points to a lower-level page table.
447             pagetable_t mid_table = (pagetable_t) PTE2PA(top_pte);
448             // 遍历中间级页目录
449             for (int j = 0; j < PAGE_SIZE; ++j) {
450                 pte_t mid_pte = mid_table[j];
451                 if (mid_pte & PTE_V) {
452                     printf(".. ..%d: pte %p pa %p\n",
453                           j, mid_pte, PTE2PA(mid_pte));
454                     pagetable_t bot_table = (pagetable_t) PTE2PA(mid_pte);
455                     // 遍历最低级页目录
456                     for (int k = 0; k < PAGE_SIZE; ++k) {
457                         pte_t bot_pte = bot_table[k];
458                         if (bot_pte & PTE_V) {
459                             printf(".. .. ..%d: pte %p pa %p\n", k, bot_pte, PTE2PA(bot_pte));
460                         }
461                     }
462                 }
463             }
464         }
465     }
466 }
```

2. 在 `kernel/defs.h` 文件中添加函数声明

```
171 int      copyin(pagetable_t, char *, uint64, uint64);
172 int      copyinstr(pagetable_t, char *, uint64, uint64);
173 void     vmprint(pagetable_t pagetable);
```

3. 在 `kernel/exec.c` 的 `exec` 函数的 `return argc` 前插入 `if(p->pid==1) vmprint(p->pagetable)`。

```
18     if(p->pid==1)
19         vmprint(p->pagetable);
20 }
```

4. 终端执行 make qemu 编译运行 xv6。命令结果：

```
xv6 kernel is booting
hart 2 starting
hart 1 starting
page table 0x0000000087f6a000
..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
$ QEMU: Terminated
```

5. 实验单项评分

```
vale@Puppyyoo: ~/xv6-labs-2021$ ./grade-lab-pgtbl pte
make: 'kernel/kernel' is up to date.
== Test pte printout == pte printout: OK (1.2s)
(Old xv6.out.ptepprint failure log removed)
```

2.3 实验中遇到的问题和解决办法

问题：格式化输出的问题。

解决办法：结合控制台调试，使用 %p 格式化符正确地打印了 64 位的十六进制 PTE（页目录表）和物理地址。

2.4 实验心得

通过输出的页表信息，可以直观地看到页表条目的层次结构和对应的物理地址。这对于理解和调试分页机制有很大帮助。

3 Detecting which pages have been accessed

3.1 实验目的

有些垃圾回收器（一种自动内存管理形式）可以通过获取哪些页面被访问过（读或写）的信息来获益。在本实验的这一部分中，将为 xv6 添加一个新功能，通过检查 RISC-V 页表中的访问位来检测并报告这些信息到用户空间。RISC-V 硬件页步行器在解决 TLB 未命中时会在 PTE 中标记这些位。实验的最终目的是实现 pgaccess() 系统调用，它报告哪些页面被访问过。

3.2 实验步骤

1. 在 kernel/riscv.h 文件中，定义以下页表条目（PTE）标志位：

```
342 #define PTE_V (1L << 0) // valid
343 #define PTE_R (1L << 1)
344 #define PTE_W (1L << 2)
345 #define PTE_X (1L << 3)
346 #define PTE_U (1L << 4) // 1 -> user can access
347 #define PTE_A (1L << 6) // accessed
348
349
```

2. 在 kernel/sysproc.c 文件中，添加以下代码以实现 sys_pgaccess 系统调用：
sys_pgaccess 接收三个参数，分别为：1. 起始虚拟地址； 2. 遍历页数目； 3. 用户存储返回结果的地址。因为其是系统调用，故参数的传递需要通过 argaddr、argint 来完成。

通过不断的 walk 来获取连续的 PTE，然后检查其 PTE_A 位，如果为 1 则记录在 mask 中，随后将 PTE_A 手动清 0。

最后，通过 copyout 将结果拷贝给用户即可。

```
78
79 #ifdef LAB_PGTBL
80
81 int sys_pgaccess(void) {
82     uint64 vaddr;
83     int num;
84     uint64 res_addr;
85     argaddr(0, &vaddr);
86     argint(1, &num);
87     argaddr(2, &res_addr);
88
89     struct proc *p = myproc();
90     pagetable_t pagetable = p->pagetable;
91     uint64 res = 0;
92
93     for(int i = 0; i < num; i++) {
94         pte_t *pte = walk(pagetable, vaddr + PGSIZE * i, 1);
95         if(*pte & PTE_A) {
96             *pte &= (~PTE_A);
97             res |= (1L << i);
98         }
99     }
100     copyout(pagetable, res_addr, (char*)&res, sizeof(uint64));
101     return 0;
102 }
103 #endif
104
```

3. defs.h 中添加 vmprint 及 walk 的函数声明：

```
171 int copyin(pagetable_t, char *, uint64, uint64);
172 int copyinstr(pagetable_t, char *, uint64, uint64);
173 void vmprint(pagetable_t pagetable);
174 pte_t* walk(pagetable_t pagetable, uint64 va, int alloc);
175
```

4. 保存后在终端里执行 make qemu 编译运行 xv6。运行 pgtbltest 结果：

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

3.3 实验中遇到的问题和解决办法

问题：程序陷入死锁或出现死循环。

解决办法：检查是否有未正确关闭的描述符，在合适的时机关闭读写端，避免导致进程阻塞或死锁的情况

3.4 实验心得


通过本实验，成功实现了一个基于 RISC-V 页表的页面访问检测系统调用 `pgaccess()`，并验证了其正确性。该系统调用能够有效地检测哪些页面被访问过，这对于垃圾回收器等自动内存管理系统的优化具有重要意义。实验过程中，通过解决页表遍历中的错误、正确处理用户空间缓冲区地址传递、以及确保清除 `PTE_A` 标志位，最终达成了实验目标。

4 实验检验得分

1. 在实验目录下创建 `time.txt`，填写完成实验时间数。

 `time.txt` 2025/7/10 0:01 文本文档 1 KB

2. 创建 `answers-pgtbl.txt` 文件,将程序运行结果填入。

 `answers-pgtbl.txt` 2025/7/10 15:44 文本文档 1 KB

3. 在终端中执行 `make grade`,得到 lab3 总分：

```
$ make qemu-gdb
(1.4s)
== Test pgtbltest: ugetpid ==
pgtbltest: ugetpid: OK
== Test pgtbltest: pgaccess ==
pgtbltest: pgaccess: OK
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (1.1s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test usertests ==
$ make qemu-gdb

(72.3s)
== Test usertests: all tests ==
usertests: all tests: OK
== Test time ==
time: OK
Score: 46/46
```

4. Lab3 代码提交

保存并提交到本地分支：

```
vale@Puppyyoo:~/xv6-labs-2021$ git add .
vale@Puppyyoo:~/xv6-labs-2021$ git commit -m"pgtbl finished"
On branch pgtbl
Your branch is ahead of 'origin/pgtbl' by 1 commit.
(use "git push" to publish your local commits)

nothing to commit, working tree clean
vale@Puppyyoo:~/xv6-labs-2021$
```