实验报告

Lab 3

pagetable

姓名：张明瑞

班级：2020 级信息安全

学号：20307130247

一、 实验

Part A：Speed up system calls

(一)实验步骤

1. 在结构 proc 中保存结构 usyscall：

```
85 struct proc {
86   struct spinlock lock;
87
88   // p->lock must be held when using these:
89   enum procstate state;        // Process state
90   void *chan;                  // If non-zero, sleeping on chan
91   int killed;                  // If non-zero, have been killed
92   int xstate;                  // Exit status to be returned to parent's wait
93   int pid;                     // Process ID
94
95   // wait_lock must be held when using this:
96   struct proc *parent;         // Parent process
97
98   // these are private to the process, so p->lock need not be held.
99   struct usyscall *usyscall;   // Speed up user syscall and avoid switching to kernel
```

2. 将 USYSCALL 映射到页表中：

(1)在函数 allocproc()中为 usyscall 分配内存：

```
// Allocate a shared usyscall page.
if((p->usyscall = (struct usyscall *)kalloc()) == 0){
  freeproc(p);
  release(&p->lock);
  return 0;
}
// Copy p->pid to usyscall->pid.
p->usyscall->pid = p->pid;
```

(2)在函数 proc_pagetable()中建立 USYSCALL 与页表的映射：

```
219   // map the usyscall page just below the trapframe page.
220   if(mappages(pagetable, USYSCALL, PGSIZE,
221               (uint64)(p->usyscall), PTE_R | PTE_U)){
222     uvmunmap(pagetable, TRAMPOLINE, 1, 0);
223     uvmunmap(pagetable, TRAPFRAME, 1, 0);
224     uvmfree(pagetable, 0);
225     return 0;
226   }
```

需要注意 USYSCALL 还在内核空间，于是设置页属性时需要设置 PTE_R | PTE_U，表示用户可访问但只有读权限。

3. 释放内存、取消映射：

(1)在函数 freeproc()中释放内存：

```
166 static void
167 freeproc(struct proc *p)
168 {
169   if(p->trapframe)
170     kfree((void*)p->trapframe);
171   p->trapframe = 0;
172   if(p->pagetable)
173     proc_freepagetable(p->pagetable, p->sz);
174   p->pagetable = 0;
175   if(p->usyscall)
176     kfree((void*)p->usyscall);
177   p->usyscall = 0;
```

(2)在函数 proc_freepagetable()中释放内存并取消映射：

```
233 void
234 proc_freepagetable(pagetable_t pagetable, uint64 sz)
235 {
236   uvmunmap(pagetable, TRAMPOLINE, 1, 0);
237   uvmunmap(pagetable, TRAPFRAME, 1, 0);
238   uvmunmap(pagetable, USYSCALL, 1, 0);
239   uvmfree(pagetable, sz);
240 }
```

(二)运行结果

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

Part B：Print a page table

(一)实验步骤

1. 在 kernel/vm.c 中增加函数 vmprint()：

```
33 void vmprint(pagetable_t pagetable){
34   printf("page table %p\n", pagetable);
35   vmprintRecurrence(pagetable, 0);
36 }
```

我准备使用递归的方式来遍历各级页表，但受限于实验对 vmprint()的参数要求，于是使用另一个函数来进行递归。

2. 在 kernel/vm.c 中增加函数 vmprintRecurrence()来进行递归：

```
18 char *prefix[4] = {"..", ".. ..", ".. .. .."};
19
20 void vmprintRecurrence(pagetable_t p, int level){
21   if(level == 3)
22     return;
23   for(int i = 0; i < 512; ++i){
24     pte_t pte = p[i];
25     if((pte & PTE_V)){
26       uint64 child = PTE2PA(pte);
27       printf("%s%d: pte %p pa %p\n", prefix[level], i, pte, child);
28       vmprintRecurrence((pagetable_t)child, level + 1);
29     }
30   }
31 }
```

该函数采取深度优先搜索的算法，遍历各级页表，并按照实验所要求的格式打印它们的内容。在函数外定义了一个字符串数组，以便打印各行内容的前缀。

3. 在 kernel/exe.c 中插入语句调用函数 vmprint()：

```
131   if(p->pid == 1)
132     vmprint(p->pagetable);
133   return argc; // this ends up in a0, the first argument to main(argc, argv)
```

(二)运行结果

```
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 0x0000000021fdd013 pa 0x0000000087f74000
.. .. ..510: pte 0x0000000021fdcc07 pa 0x0000000087f73000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
```

Part C：Detect which pages have been accessed

(一)实验步骤

1.在 kernel/risc.h 中添加宏定义 PTE_A:

```
346 #define PTE_A (1L << 6)
```

2.在 kernel/sysproc.c 中实现函数 sys_pgaccess():

```
74 int sys_pgaccess(void)
75 {
76     uint64 addr;
77     int len;
78     uint64 bitmask;
79     argaddr(0, &addr);
80     argint(1, &len);
81     argaddr(2, &bitmask);
82     struct proc* p = myproc();
83     uint64 result = vm_pgaccess(p->pagetable, addr, len);
84     if (copyout(p->pagetable, bitmask, (char*)&result, sizeof(result)) < 0)
85         return -1;
86     return 0;
87 }
```

3.在 vm.c 中实现 vm_pgaccess()函数，作为对 sys_pgaccess()的补充:

```
461 uint64 vm_pgaccess(pagetable_t pagetable, uint64 addr, int len)
462 {
463     if (addr >= MAXVA)
464         panic("walk");
465     uint64 mask = 0;
466     for (int i = 0; i < len; i++, addr += PGSIZE)
467     {
468         pte_t* pte = 0;
469         pagetable_t p = pagetable;
470         for (int level = 2; level >= 0; level--)
471         {
472             pte = &p[PX(level, addr)];
473             if (*pte & PTE_V)
474                 p = (pagetable_t)PTE2PA(*pte);
475         }
476         if (*pte & PTE_V && *pte & PTE_A)
477         {
478             mask |= (1L << i);
479             *pte &= ~(PTE_A);
480         }
481     }
482     return mask;
483 }
```

(二)运行结果

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

二、 回答问题

1.

在 Part A 中，我们将进程的 pid 保存到 USYSCALL 中，借此避免用户态与内核态间切换的开销。

或许 fork()、read()、write()等可以建立映射的函数都可以使用此方法加速。

2.

虚拟内存可以用来缓解物理内存的不足。

在计算机本身运行内存不足时，可以将部分硬盘空间暂且作为虚拟内存使用，这样一来，计算机就有相对充足的资源去同时运行大量进程，这有助于提高其工作效率。

3.

使用多级页表本质上是增加索引，使用多级页表后可以将页表分为目录与下级页表，在目录中写入后续页表的索引，就可以将它们离散存储在内存中，可利用较为分散的内存空间资源；如果采用单级页表，则各页表需要连续空间以配合索引，不能利用碎片内存资源，比较浪费空间，这就像数组与指针的区别。

4.
make qemu 后输入 pgtbltest 指令，即在用户态下运行 pgtbltest.c 的可执行文件，观察其中函数 pgaccess_test()代码：

```
51 void
52 pgaccess_test()
53 {
54   char *buf;
55   unsigned int abits;
56   printf("pgaccess_test starting\n");
57   testname = "pgaccess_test";
58   buf = malloc(32 * PGSIZE);
59   if (pgaccess(buf, 32, &abits) < 0)
60     err("pgaccess failed");
61   buf[PGSIZE * 1] += 1;
62   buf[PGSIZE * 2] += 1;
63   buf[PGSIZE * 30] += 1;
64   if (pgaccess(buf, 32, &abits) < 0)
65     err("pgaccess failed");
66   if (abits != ((1 << 1) | (1 << 2) | (1 << 30)))
67     err("incorrect access bits set");
68   free(buf);
69   printf("pgaccess_test: OK\n");
70 }
```

可以发现它会将 buf、32、abits 传入我们填写的函数 pgaccess()中，而函数 pgaccess()的主要功能是检查以 buf 为起始地址，连续 32 页，是否有被访问过，即 PTE_A 的页属性是否为 1，并将结果通过 abits 返回。
而在函数 pgaccess()中，关键步骤为获取检查的虚拟页的 pte 以及将数据从内核态传回用户态，这分别通过使用 walk()和 copyout()函数实现。

三、实验感想

本次实验收获很多，尤其是在 Part C 中了解各函数的调用方法，这样从微观局部入手逐步了解宏观系统的方式让人受益匪浅。
另外调试 bug 的过程也让人学到不少，虽然目前已能通过测试，但或许还能做一些我暂未想到的优化，比如如何减少循环嵌套、如何减少递归重复次数等等。