os_lab3_report

21302010042

侯斌洋

（1）Speed up system calls (easy)

实验思路及代码：

memlayout.h中已经为我们定义了page的虚拟地址USYSCALL和要使用的结构usyscall

```
//memlayout.h
#ifdef LAB_PGTBL
#define USYSCALL (TRAPFRAME - PGSIZE)

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

在proc.h中添加usyscall的指针以存储pid。

```
//proc.h
struct usyscall *usc_pid;
```

之后在allocproc()函数中为该结构分配页，并保存pid。

```
//proc.c
if ((p->usc_pid = (struct usyscall *)kalloc()) == 0)
{
    freeproc(p);
    release(&p->lock);
    return 0;
}
p->usc_pid->pid = p->pid;
```

然后在proc_pagetable() 中以用户可读的状态将映射写入 pagetable 中，地址为USYSCALL

```
//proc.c
if (mappages(pagetable, USYSCALL, PGSIZE,(uint64)(p->usc_pid), PTE_R | PTE_U) < 0)
{
    uvmunmap(pagetable, TRAPFRAME, 1, 0);
    uvmfree(pagetable, 0);
    return 0;
}
```

在freeproc() 中释放该共享页

```
//proc.c
if (p->usc_pid)
    kfree((void *)p->usc_pid);
p->usc_pid = 0;
```

最后在proc_freepagetable() 中解除映射关系。

```
//proc.c
uvmunmap(pagetable, USYSCALL, 1, 0);
```

这样就可以通过共享页直接获取pid，减少了内核交叉，从而加速了getpid()系统调用。

实验过程：

代码：见上述，与思路一同给出（lab3_src中也有源文件）。

系统调用流程：ugetpid()代码如下：

```
//ulib.c
#ifdef LAB_PGTBL
int
ugetpid(void)
{
  struct usyscall *u = (struct usyscall *)USYSCALL;
  return u->pid;
}
#endif
```

可知该系统调用直接通过USYSCALL，即共享页面所在的虚拟地址，来获取usyscall的地址，然后通过usyscall直接读取pid并返回。因此该系统调用的速度很快。

实验效果： （lab3_result中也有实验截图）

pgtbltest：

```
hby@hby-ubuntu:~/Desktop/xv6-labs-2022$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -g

xv6 kernel is booting

hart 2 starting
hart 1 starting
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
init: starting sh
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgaccess_test: OK
pgtbltest: all tests succeeded
$
```

make grade:

```
make[1]: Leaving directory '/home/hby/Desktop/xv6-labs-2022'
== Test pgtbltest ==
$ make qemu-gdb
(3.0s)
== Test   pgtbltest: ugetpid ==
  pgtbltest: ugetpid: OK
== Test   pgtbltest: pgaccess ==
  pgtbltest: pgaccess: OK
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (0.6s)
```

(2) Print a page table (easy)

实验思路及代码：

首先在 exec.c 中 return argc 之前插入以下代码来打印第一个进程的页表

```
//exec
if (p->pid == 1)
    vmprint(p->pagetable);
```

然后在defs.h中定义vmprint()

```
//defs.h
void vmprint(pagetable_t pt);
```

并在vm.c中实现

```
//vm.c
void vmprint(pagetable_t pagetable)
{
  static int depth = 0;
  if (depth == 0)
    printf("page table %p\n", 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))
    {
      if (depth < 3)
      {
        depth++;
        // this PTE points to a lower-level page table.
        uint64 child = PTE2PA(pte);
        for (int j = 0; j < depth - 1; j++)
          printf(".. ");
        printf("..");
        printf("%d: pte %p pa %p\n", i, pte, PTE2PA(pte));
        vmprint((pagetable_t)child);
        depth--;
      }
    }
  }
}
```

以上代码主要参考 freewalk() 代码的实现，递归地向下寻找下一级页表直至到达最后一层。通过静态变量 depth 来记录当前递归的深度并以此为依据来构建树形输出。

实验过程：

代码：见上述，与思路一同给出。

系统调用流程：vmprint()主要为（3）中调试服务，只是递归地打印pagetable中的有效项，不存在复杂的系统调用。

实验效果：

```
#见（1）中结果
```

（3）Detect which pages have been accessed (hard)

实验思路及代码：

首先查找 RISC-V privileged architecture manual 得到 PTE_A 的值，并在riscv.h中定义

```
//riscv.h
#define PTE_A (1L << 6) // acess
```

然后在 sysproc.c 中实现该系统调用即可。

```c
//sysproc.c
#ifdef LAB_PGTBL
int sys_pgaccess(void)
{
  // lab pgtbl: your code here.
  uint64 start_va;
  int pg_num;
  uint64 user_addr_bits_mask;

  // get args
  argaddr(0, &start_va);
  argint(1, &pg_num);
  argaddr(2, &user_addr_bits_mask);

  // get pagetables
  struct proc *cur_proc = myproc();
  pagetable_t pgtb = cur_proc->pagetable;

  unsigned int bits_buf = 0;
  unsigned long bits_mask = 1;
  pte_t *pte;

  //vmprint(pgtb);

  for (uint64 i = 0; i < pg_num; i++)
  {
    pte = walk(pgtb, start_va + i * PGSIZE, 0);
    if ((*pte & PTE_A) != 0)
    {
      bits_buf += bits_mask;
    }
    // printf("pte : %p\n", *pte);
    // printf("bits_mask : %p\n", bits_mask);
    // printf("bits_buf  : %p\n", bits_buf);
    bits_mask = (bits_mask << 1);
    *pte = ((*pte) & (~PTE_A));
  }
  copyout(pgtb, user_addr_bits_mask, (char *)(&bits_buf), sizeof(user_addr_bits_mask));
  return 0;
}
#endif
```

sys_pgaccess() 思路如下：

首先获取系统调用的参数， start_va 为起始虚拟地址， pg_num 为要检测的page数量，user_addr_bits_mask 为按位存储的变量的用户地址，之后要通过 copyout 函数来将bits_buf(按位存储检测到 PTE_A 的 page)传到该用户地址。然后获取当前进程的pagetable，即变量pgtb。然后利用walk函数获取虚拟地址对应的 pte 并进行检测，若 pte 中 PTE_A 位为 1 则在 bits_buf 中记录该位已被访问过。遍历所有要检测的 page 即可获取完整的 bits_buf。 之后传到用户地址即可。

实验过程：

代码：见上述，与思路一同给出。

系统调用流程：根据riscv的架构，每次访问page都会设置 PTE_A ，在pgaccess_test()中通过下面的代码

```
buf[PGSIZE * 1] += 1;
buf[PGSIZE * 2] += 1;
buf[PGSIZE * 30] += 1;
```

访问了buf中的三个page，这三个 page 对应的 pte 的 PTE_A 位就置为 1，之后进行系统调用 sys_pgaccess 时通过检测 pte 对应的位就可以构造出 bits_buf，并通过 copyout 函数将值写入 abits ，从而使用户也可以知道哪些 page 被访问过。

实验效果：

```
#见（1）中结果
```