Lab 3

Part A: Speed up system calls

在 proc.h 中 proc 结构部分添加 usyscall 结构体指针*usyscall。先在 allocproc()中模仿 trapframe 为 usyscall 分配空间，如果分配失败则释放进程 p 以及锁，并直接返回 0。分配完后根据提示将当前进程的 pid 存储至 usyscall 结构体指针中的 pid 部分。代码如下：

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

在 freeproc()部分同样模仿 trapframe 释放 usyscall，代码如下。

```
if(p->usyscall)
    kfree((void*) p->usyscall);
  p->usyscall = 0;
```

随后，在 proc_pagetable()部分完成页到 USYSCALL 的映射。模仿先前 TRAMPOLINE 与 TRAPFRAME 部分，运用 mappage()函数实现映射，并对返回值做判断：若返回值小于零，表示映射失败，因此需要通过 uvmunmap()函数解除先前定义的映射关系，释放页表，并立即返回 0。其中，mappage()函数的最后一个 int 变量值设置为 PTE_R | PTE_U，表示允许读，以及允许用户访问。完整代码如下：

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

另外，在紧随其后的 proc_freepagetable()部分同样模仿 TRAMPOLINE 与 TRAPFRAME 解除 USYSCALL 映射。

完成以上全部工作后尝试运行，发现 xv6 报出以下错误信息：

```
$ pgtbltest
ugetpid_test starting
usertrap(): unexpected scause 0x0000000000000005 pid=4
            sepc=0x000000000000049a stval=0x0000003fffffd000
$ QEMU: Terminated
```

检查代码后发现分配空间时 usyscall 的分配发生在页表的分配之后，因此调用 proc_pagetable()函数时无法实施 USYSCALL 的映射。修改后可正常运行得到结果，

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

```
p->usyscall->pid = p->pid;

// An empty user page table.
p->pagetable = proc_pagetable(p);
......
```

Part B: Print a page table

根据提示在 exec.c 中加入相应语句。vmprint 部分采取深度优先遍历的方式。对每个 page table entry，用 pte & PTE_V 判断该页表项有效位是否为 1，若有效则通过 PTE2PA 宏来获取该页表项指向的物理地址，并按照提示规则进行打印。随后，由于指向下一级页表的页表项在读、写以及可执行位上均为零，因此可以通过 pte & (PTE_R | PTE_W | PTE_X)进行判断；若该页表项指向下一级页表，则将等级值减一并递归调用函数 vmpwalk()。完成 vmpwalk 的编写后，在打印部分主函数 vmprint 中调用 vmpwalk，并将初始等级设置为 2。代码如下。

```
void vmpwalk(pagetable_t pagetable, int level){
  for (int i = 0; i < 512; i++){
    pte_t pte = pagetable[i];
    if (pte & PTE_V){
      pagetable_t p_addr = (pagetable_t)PTE2PA(pte);
      switch (level){
        case 2: printf(".."); break;
        case 1: printf(".. .."); break;
        case 0: printf(".. .. .."); break;
      }
      printf("%d:pte %p pa %p\n", i, pte, p_addr);
      if ((pte & (PTE_R | PTE_W | PTE_X)) == 0) vmpwalk(p_addr, level-1);
    }
  }
}

void vmprint(pagetable_t pagetable){
  printf("page table %p\n", pagetable);
  vmpwalk(pagetable, 2);
}
```

Part C: Detect which pages have been accessed

首先在 sysproc.c 中的 sys_pgaccess()记录用户态传入的参数。其中，0 号、1 号、2 号寄存器中分别存储初始地址、页数和结果存储地址。对于地址，用 argaddr()函数获得参数；对于页数一类的整型变量则使用 argint()。获得参数后，在该函数内调用 pgaccess()并返回 pgaccess 的返回值。另外，在 riscv.h 部分需添加 PTE_A 的宏定义，定义值根据 xv6-book 设置为 1L<<6。这意味着每个页表项的第六位表示该页是否被访问过。代码如下：

```
#ifdef LAB_PGTBL
int
```

```
sys_pgaccess(void)
{
  // lab pgtbl: your code here.
  uint64 inipg_addr;
  int pg_num;
  uint64 store_addr;
  argaddr(0, &inipg_addr);
  argint(1, &pg_num);
  argaddr(2, &store_addr);
  return pgaccess(inipg_addr, pg_num, store_addr);
}
#endif
```

随后，在 proc.c 末尾定义函数 pgaccess()，利用传入参数中初始地址与页数对所有需要检查的页进行遍历，在每次遍历中通过当前页数与初始地址计算得到当前虚拟地址，并用 walk()函数获得相应的页表项地址。获得页表项后通过(*pte) & PTE_A 检查该页的访问情况，若被访问过则将变量 bitmask 的对应位的值改为一，并将该页的访问情况重置为零。最后，通过 copyout()将结果交还给用户态，并返回 copyout 的返回值。代码如下。

```
uint64
pgaccess(uint64 inipg_addr, int pg_num, uint64 store_addr){
  int max_num = 64;
  struct proc *p = myproc();
  pagetable_t pgtable = p->pagetable;
  int bitmask = 0;
  if (pg_num > max_num) return -1;

  for (int i = 0; i < pg_num; i++){
    uint64 virtual_addr = inipg_addr + i * PGSIZE;
    pte_t *pte = walk(pgtable, virtual_addr, 0);
    if((*pte) & PTE_A){
      bitmask |= (1 << i);
      (*pte) &= ~PTE_A;
    }
  }
  return copyout(pgtable, store_addr, (char *)&bitmask, sizeof(bitmask));
}
```

三个实验的最终结果如下：

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
page table 0x0000000087f72000
..0:pte 0x0000000021fdb801 pa 0x0000000087f6e000
.. ..0:pte 0x0000000021fdb401 pa 0x0000000087f6d000
.. .. ..0:pte 0x0000000021fdbc5f pa 0x0000000087f6f000
..255:pte 0x0000000021fdc401 pa 0x0000000087f71000
.. ..511:pte 0x0000000021fdc001 pa 0x0000000087f70000
.. .. ..509:pte 0x0000000021fdcc13 pa 0x0000000087f73000
.. .. ..510:pte 0x0000000021fdd0c7 pa 0x0000000087f74000
.. .. ..511:pte 0x0000000020001c4b pa 0x0000000080007000
init: starting sh
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgaccess_test: OK
pgtbltest: all tests succeeded
```

Questions

1. **在 Part A 加速系统调用部分，除了 getpid()系统调用函数，你还能想到哪些系统调用函数可以如此加速?**

   Part A 通过 paging 共享用户态和内核态的数据，因此凡是需要在用户态、内核态切换来获得数据的系统调用，亦即在系统调用函数中使用 myproc()者均可以通过这种方式加速，如 sys_sbrk 等。

2. **虚拟内存有什么用处？**

   虚拟内存的理念之一在于让用户可见的地址空间趋近于无限：通过虚拟地址到物理地址到映射关系以及替换规则的设置，虚拟内存可以让用户在同样有限的物理地址下使用远大与其的虚拟地址空间。另外，虚拟内存还能够解决用户空间必须连续分配的问题。运用虚拟内存，用户使用的地址逻辑上连续，但在实际的物理空间中可能散落在天涯海角。

3. **为什么现代操作系统采用多级页表？**

   单级页表容易造成页表过大，占用过多内存空间的问题。以 32 位地址空间为例，假设页大小为 16KB，则页内偏移为 14 位，页帧号为 18 位，故而总共需要 2^{18} 个页表项。如果每个页表项为 4Byte，则页表总共需占用 2^{20}Byte，即 1MB 空间。若采用多级页表，假设外页号为 8 位，内页号为 10 位，则总共占用空间为 $2^{10} + 2^{12}$ Byte，即 5KB，其远小于单级页表所占用的内存空间。

4. **简述 Part C 的 detect 流程。**

   进入 qemu 后，在用户态下运行 pgtbltest，运行至 pgaccess_test 部分将切换至内核态并调用系统函数 pgaccess。内核态首先经由 sysproc.c 将用户输入参数所在寄存器中的数值传入相应变量中并调用 proc.c 文件中定义的系统函数 pgaccess。pgaccess 通过循环遍历所有需要检查的页，通过当前页数及初始地址计算出当前虚拟地址并由此得到相

应的页表项，随后通过 PTE_A 检查该页表项是否被访问过，若被访问过，则将 bitmask 的对应位置为一，同时通过取否运算清空该页表项的访问状态。最后，通过 copyout 将结果交还用户态。

Thoughts

该实验分别实现了系统调用加速、页表打印以及页表项访问状态检查，其中包含页表映射、页表项信息查询等功能，加深了对于页表概念以及虚拟地址、物理地址间关系的理解。