lab syscall 实验报告

lab 实现

exercise 0

这部分直接看system call tracing就好了

exercise system call tracing

这个实验要求实现一个给用户态的系统调用函数`trace`对当前进程及其子进程对应的系统调用进行追踪和打印，根据xv6 book和实验提示按照用户态进行系统调用的流程即可完成实验。

第一步在`user/user.h`中添加`int trace(int)`作为给用户态程序进行系统调用时调用的函数。

第二步创建`trace`由用户态切换至内核态的入口，xv6已经了生成入口函数的脚本，只需要在`user/usys.pl`中加入`entry("trace")`脚本就会生成对应的函数（在`user/usys.S`中）。

第三步创建在内核态运行的系统调用函数，在`kernel/syscall.h`中增加`#define SYS_trace 22`，在`kernel/sysproc.c`中增加`uint 64 sys_trace(void)`函数。

```
uint sys_trace(void){
  int n;
  if(argint(0, &n) < 0){
    return -1;
  }
  struct proc* p = myproc();

  acquire(&p->lock);
  myproc()->mask = n;
  release(&p->lock);

  return 0;
}
```

当前已经处于内核态，利用`argint`从进程的trapframe中获取保存在寄存器中的参数，修改进程对应的mask参数（标记追踪的系统调用函数），因为需要对进程的数据进行修改，所以这里使用了锁。（不使用锁应该也没问题？）

第四步在调用`syscall`函数时根据进程的mask决定是否要打印当前的系统调用及其返回结果。

```
    p->trapframe->a0 = syscalls[num]();
    if((1 << num) & p->mask){
      printf("%d: syscall %s -> %d\n", p->pid, syscalls_name[num], p->trapframe-
>a0);
    }
```

返回结果存储在进程的trapframe中，另外这里建立了一个字符串数组`syscalls_name`用于打印函数名。

最后一步是在第一个进程创建时初始化mask并且在`fork()`函数中用父进程的mask给子进程的mask赋值。

经过上述几步该实验就完成了。

exercise Sysinfo

这个实验要求实现一个系统调用函数返回记录了当前空闲内存和不再UNUSED状态的进程数。关于增加新的系统调用的部分和前一个实验是完全一致的，不一样的地方在于内核态中系统调用函数的具体实现。

在`kernel\kalloc.c`和`kernel\proc.c`中增加计数用的函数。

```
uint64 count_freemem(void){
  uint64 free_pages = 0;
  // acquire(&kmem.lock);
  for(struct run* r = kmem.freelist; r != (void*)0; r = r->next){
    ++free_pages;
  }
  // release(&kmem.lock);

  return free_pages * PGSIZE;
}

uint64 count_not_unused_proc(void){
  struct proc* p;
  uint64 unused_proc = 0;
  for(p = proc; p < &proc[NPROC]; ++p){
    if(p->state != UNUSED){
      ++unused_proc;
    }
  }

  return unused_proc;
}
```

仿照类似的函数进行遍历即可计数，因为只有读操作所以我没有使用lock。

具体的系统调用函数如下

```
uint64 sys_sysinfo(void){
  struct sysinfo info;
  info.freemem = count_freemem();
  info.nproc = count_not_unused_proc();

  uint64 uinfo;
  if(argaddr(0, &uinfo) < 0){
    return -1;
  }

  if(copyout(myproc()->pagetable, uinfo, (char *)&info, sizeof(struct sysinfo))
< 0){
    return -1;
  }

  return 0;
}
```

关键点是利用`argaddr`从寄存器中获取在用户地址空间中sysinfo的地址，再利用`copyout`函数将当前存储有具体数值的info复制到用户态中sysinfo。`copyout`利用进程的pagetable能通过用户态的虚拟地址找到物理地址，然后将指定物理地址的值复制到其中。

```
== Test trace 32 grep ==
$ make qemu-gdb
trace 32 grep: OK (8.5s)
== Test trace all grep ==
$ make qemu-gdb
trace all grep: OK (2.6s)
== Test trace nothing ==
$ make qemu-gdb
trace nothing: OK (2.0s)
== Test trace children ==
$ make qemu-gdb
trace children: OK (29.3s)
== Test sysinfotest ==
$ make qemu-gdb
sysinfotest: OK (5.4s)
    (Old xv6.out.sysinfotest failure log removed)
== Test time ==
time: OK
Score: 35/35
```

完成两个实验后的grade如上。

lab 总结

lab syscall总体上比lab util要简单，并没有一些比较tricky的实现，但要成功完成这个lab需要我们对xv6的系统调用流程特别熟悉，通过阅读源代码和xv6 book理清了xv6系统调用的过程：用户程序直接调用提供给用户态的系统调用函数，该函数通过一段RISC-V代码调用ecall陷入内核，陷入内核时用户程序的context保存在进程trapframe中，内核的系统调用函数通过进程的trapframe获取参数，通过pagetable读写用户程序的内存，执行完操作后将返回值保存在进程的trapframe中再从内核态返回用户态。明白了这一过程后只需要按照流程在关键处添加代码即可完成实验。