**实验二 system calls**

**2.1 实验目的**

1. 进一步了解系统调用

2. 通过阅读xv6-book了解用户态、核心态相关知识并理解为什么这么设计

3. 尝试修改kernel中的系统调用并理解其运行过程

**2.2 实验内容**

2.2.1 System call tracing（moderate）

在本作业将添加一个系统调用跟踪功能，该功能可能在以后调试实验室时对您有所帮助。您将创建一个新的跟踪系统调用来控制跟踪。它应该有一个参数，一个整数“mask”，其位指定要跟踪的系统调用。例如，为了跟踪fork系统调用，程序调用trace（1<<SYS\_fork），其中SYS\_fork是kernel/syscall.h中的syscall编号。如果在mask中设置了系统调用的编号，则必须修改xv6内核，以便在每个系统调用即将返回时打印出一行。该行应包含进程id、系统调用的名称和返回值；您不需要打印系统调用参数。跟踪系统调用应启用对调用它的进程及其随后派生的任何子进程的跟踪，但不应影响其他进程。

2.2.2 Sysinfo（moderate）

在此作业中，您将添加一个系统调用sysinfo，该调用收集有关正在运行的系统的信息。系统调用接受一个参数：指向结构sysinfo的指针（请参阅kernel/sysinfo.h）。内核应填写此结构的字段：freemem字段应设置为可用内存的字节数，nproc字段应设置为状态未使用的进程数。我们提供了一个测试程序sysinfotest；如果它打印“sysinfotest:OK”，则通过此项目。

2.2.3 submit

提交实验。

**2.3 实验过程**

2.3.1 system call tracing

（1）为了实现trace功能需要修改proc.h中的porc结构体，添加mask变量表示每个进程的mask，添加后如下所示

// Per-process state

struct proc {

  struct spinlock lock;

  // p->lock must be held when using these:

  enum procstate state;        // Process state

  struct proc \*parent;         // Parent process

  void \*chan;                  // If non-zero, sleeping on chan

  int killed;                  // If non-zero, have been killed

  int xstate;                  // Exit status to be returned to parent's wait

  int pid;                     // Process ID

  // these are private to the process, so p->lock need not be held.

  uint64 kstack;               // Virtual address of kernel stack

  uint64 sz;                   // Size of process memory (bytes)

  pagetable\_t pagetable;       // User page table

  struct trapframe \*trapframe; // data page for trampoline.S

  struct context context;      // swtch() here to run process

  struct file \*ofile[NOFILE];  // Open files

  struct inode \*cwd;           // Current directory

  char name[16];               // Process name (debugging)

  int mask;

};

（2）编写trace函数，参照sysproc.c中其他函数进行编写，给进程对应的mask变量赋值。

uint64

sys\_trace(void)

{

  if(argint(0,&mask)<0)

    return -1;

  myproc()->mask=mask;

  return 0;

}

（3）在如下行中调用了syscalls这一数组，所以在此数组中对应地加入trace相关的内容。

p->trapframe->a0 = syscalls[num]();

修改后数组如下所示

static uint64 (\*syscalls[])(void) = {

[SYS\_fork]    sys\_fork,

[SYS\_exit]    sys\_exit,

[SYS\_wait]    sys\_wait,

[SYS\_pipe]    sys\_pipe,

[SYS\_read]    sys\_read,

[SYS\_kill]    sys\_kill,

[SYS\_exec]    sys\_exec,

[SYS\_fstat]   sys\_fstat,

[SYS\_chdir]   sys\_chdir,

[SYS\_dup]     sys\_dup,

[SYS\_getpid]  sys\_getpid,

[SYS\_sbrk]    sys\_sbrk,

[SYS\_sleep]   sys\_sleep,

[SYS\_uptime]  sys\_uptime,

[SYS\_open]    sys\_open,

[SYS\_write]   sys\_write,

[SYS\_mknod]   sys\_mknod,

[SYS\_unlink]  sys\_unlink,

[SYS\_link]    sys\_link,

[SYS\_mkdir]   sys\_mkdir,

[SYS\_close]   sys\_close,

[SYS\_trace]   sys\_trace,

};

为保证代码的一致性，在syscall.h中添加如下定义

#define SYS\_trace  21

（4）为输出系统调用的名称，我们使用一个新的字符串数组来进行输出，完成后的syscall函数如下所示。

void

syscall(void)

{

  char const \*syscall\_names[] = {"fork", "exit", "wait", "pipe", "read",

  "kill", "exec", "fstat", "chdir", "dup", "getpid", "sbrk", "sleep",

  "uptime", "open", "write", "mknod", "unlink", "link", "mkdir","close","trace"};

  int num;

  struct proc \*p = myproc();

  num = p->trapframe->a7;

  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {

    p->trapframe->a0 = syscalls[num]();

    if((p->mask)&(1<<num)){

      printf("%d: syscall %s -> %d\n", p->pid, syscall\_names[num - 1], p->trapframe->a0);

    }

    }

  } else {

    printf("%d %s: unknown sys call %d\n",

            p->pid, p->name, num);

    p->trapframe->a0 = -1;

  }

}

（5）在user.h以及usys.pl中进行修改。

在user.h中添加

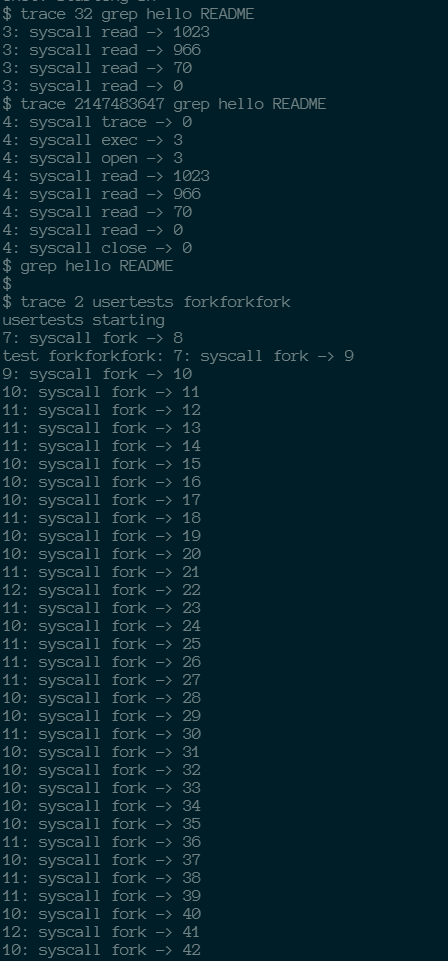
int trace(int);

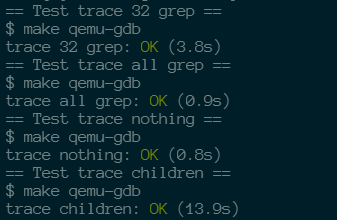
在usys.pl中添加

entry("trace");

使代码可以运行

（6）运行trace





2.3.2 sysinfo

（1）sysinfo为获得当前系统的可用进程数和可用内存。

（2）为获得空闲的物理内存即在kalloc.c中添加freemem函数统计可用内存。因为物理内存是采用链表存储当前空闲内存块，所以统计内存时遍历链表并且统计长度就可以得到空闲的内存。（注：需要乘以页面大小）

uint64

freemem(void)

{

  struct run \*r;

  uint64 freepage = 0;

  acquire(&kmem.lock);

  r = kmem.freelist;

  while (r)

  {

    freepage += 1;

    r = r->next;

  }

  release(&kmem.lock);

  return (freepage << 12);

}

（3）为获得可用进程数，只需要遍历所有进程，即proc数组，并查看其state是否为UNUSED即可统计完成，代码写于proc.c

uint64

unusedproc(void)

{

  struct proc \*p;

  uint64 unused = 0;

  for(p = proc; p < &proc[NPROC]; p++)

  {

    if(p->state != UNUSED) {

      unused++;

    }

  }

  return unused;

}

（4）在有以上两个函数后可以在sysproc.c中进行sys\_sysinfo的实现，用freemem和unusedproc统计未使用的进程和内存，最后使用copyout方法进行复制，将内容复制回用户空间。

uint64

sys\_sysinfo(void)

{

  uint64 addr;

  if(argaddr(0, &addr) < 0)

    return -1;

  struct proc \*p = myproc();

  struct sysinfo info;

  info.freemem = freemem();

  info.nproc = unusedproc();

  if(copyout(p->pagetable, addr, (char \*)&info, sizeof(info)) < 0)

      return -1;

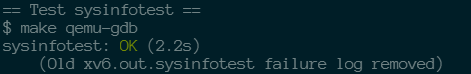
  return 0;

}

（5）在两个统计函数实现后需要在defs.h中进行声明才能在sysproc.c中使用，同时需要按照如trace一样的方式对其他函数进行声明。

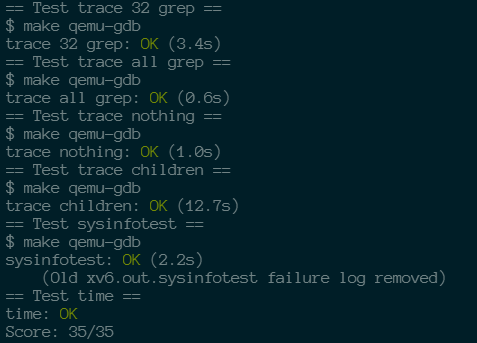
（6）运行sysinfo





**2.4 实验结果**

通过make grade对所有实验内容进行测试，结果如下图所示。



**2.5 实验小结**

本次实验更深入的对系统调用进行了解和学习，在编写代码时不仅仅只需要修改一个文件，而是在多个文件中协同进行修改实现最终代码的运行。同时还需要阅读更多kernel中的代码并且了解其中的函数来学习如何编写实验的内容。

本次实验让我更加了解xv6的工作原理并且通过实验对操作系统的结构和概念掌握的更加深刻了。