# 2023/09/07

## XV6第二章

指出操作系统三大基本功能：多路复用，隔离和交互。并着重介绍了隔离(即使在本章开头说明了介绍内容为多路复用，隔离和交互，但我认为仅仅介绍了隔离)

### 为什么需要隔离性

最容易想到的情况是，应用程序自己会带上自己运行所需的库(以进行系统调用)，在应用程序分别在自己的CPU时间帧运行时，他们都通过自己的库直接与硬件交互。这样的结构在嵌入式系统中最为常见。

但这种方式是基于应用程序没有bug，即应用程序之间互相信任的基础上实施的。更多情况下，应用程序之间或者是应用程序内部都有或多或少的bug。如果应用程序能够直接和硬件资源交互，在应用程序出错时，它所暂时占用的公共硬件资源也会记入错误的内容，这显然会间接导致其他应用程序出错。

于是引入了进程间的隔离性的概念。即任何一个进程的崩溃，都对其他进程以及操作系统没有任何影响；并且操作系统能够及时清理掉出错的进程。

### 如何实现隔离性

为了实现这一点，操作系统引入了三个功能：资源抽象而来的服务（软件），模式切换（硬件），进程独立空间。

首先操作系统禁止了进程直接与敏感资源交互，转而提供封装后的调用资源的服务。例如write，read等。另一个新奇的例子是exec，exec将进程建立内存映像的工作交给了操作系统，于是进程便不能随意地在内存空间中圈下属于自己的空间。操作系统在内存紧张时甚至可以把进程部分数据存储在磁盘上。

RISC-V的模式切换提供了用户、机器和监督者模式。这限制了用户程序直接调用内核函数或执行内核指令（如启用和禁用中断）。用户想要调用内核函数时，CPU将切换到监督者模式并在内核指定的入口进入内核，并让内核有权限验证系统调用的参数，决定是否执行该操作。当清晰划分了用户模式和监督者模式后，一个新问题便出现了——哪些功能应该被放入监督者模式。这个问题的两个答案分别派生出了操作系统内核的两种风格：宏内核与微内核，在此不赘述。

进程独立的空间包含独立的CPU和独立的地址空间。独立地址空间显然由页表实现，独立的CPU则由分时实现。xv6还会为每个进程维护一个单独的proc结构体，储存进程的相关信息（如页表，状态，id等信息）。每个进程都有一个线程（实际中通常有多个）来执行指令，它可以被暂停也可以被恢复。线程拥有自己独立的线程栈来存储局部变量，函数返回地址等信息。同时，每个进程都有一个用户栈和一个内核栈，进程的线程交替在用户栈和内核栈中执行。

### XV6初始化和第一个进程

参见文档，过程详尽。

## Lab2 System calls

对于Lab2 Syscall中的实验，首先得清楚如果要添加一个新的调用，需要做哪些修改。

在用户模式中，系统调用必须在user.h中声明，其次需要在usys.pl文件中添加对应的入口:

user.h中的声明:

// system calls

int fork(void);

int exit(int) \_\_attribute\_\_((noreturn));

int wait(int\*);

int pipe(int\*);

int write(int, const void\*, int);

int read(int, void\*, int);

int close(int);

int kill(int);

int exec(const char\*, char\*\*);

int open(const char\*, int);

int mknod(const char\*, short, short);

int unlink(const char\*);

int fstat(int fd, struct stat\*);

int link(const char\*, const char\*);

int mkdir(const char\*);

int chdir(const char\*);

int dup(int);

int getpid(void);

char\* sbrk(int);

int sleep(int);

int uptime(void);

void trace(int);

void sysinfo(struct sysinfo\*);

usys.pl的入口：

entry("fork");

entry("exit");

entry("wait");

entry("pipe");

entry("read");

entry("write");

entry("close");

entry("kill");

entry("exec");

entry("open");

entry("mknod");

entry("unlink");

entry("fstat");

entry("link");

entry("mkdir");

entry("chdir");

entry("dup");

entry("getpid");

entry("sbrk");

entry("sleep");

entry("uptime");

其次，系统调用需要在内核中作为外函数声明,声明在syscall.c中:

// Prototypes for the functions that handle system calls.

extern uint64 sys\_fork(void);

extern uint64 sys\_exit(void);

extern uint64 sys\_wait(void);

extern uint64 sys\_pipe(void);

extern uint64 sys\_read(void);

extern uint64 sys\_kill(void);

extern uint64 sys\_exec(void);

extern uint64 sys\_fstat(void);

extern uint64 sys\_chdir(void);

extern uint64 sys\_dup(void);

extern uint64 sys\_getpid(void);

extern uint64 sys\_sbrk(void);

extern uint64 sys\_sleep(void);

extern uint64 sys\_uptime(void);

extern uint64 sys\_open(void);

extern uint64 sys\_write(void);

extern uint64 sys\_mknod(void);

extern uint64 sys\_unlink(void);

extern uint64 sys\_link(void);

extern uint64 sys\_mkdir(void);

extern uint64 sys\_close(void);

extern uint64 sys\_trace(void);

extern uint64 sys\_sysinfo(void);

在syscall.c中建立系统调用函数指针数组以便调用：

// An array mapping syscall numbers from syscall.h

// to the function that handles the system call.

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\_sysinfo] sys\_sysinfo,

[SYS\_systrace] sys\_systrace,

};

其中，索引的宏定义在syscall.h中:

// System call numbers

#define SYS\_fork    1

#define SYS\_exit    2

#define SYS\_wait    3

#define SYS\_pipe    4

#define SYS\_read    5

#define SYS\_kill    6

#define SYS\_exec    7

#define SYS\_fstat   8

#define SYS\_chdir   9

#define SYS\_dup    10

#define SYS\_getpid 11

#define SYS\_sbrk   12

#define SYS\_sleep  13

#define SYS\_uptime 14

#define SYS\_open   15

#define SYS\_write  16

#define SYS\_mknod  17

#define SYS\_unlink 18

#define SYS\_link   19

#define SYS\_mkdir  20

#define SYS\_close  21

#define SYS\_trace  22

#define SYS\_sysinfo  23

而系统调用的具体实现则根据调用类型选择在sysfile.c或者sysproc.c中实现。使用频率高（被交叉使用）的系统调用可以先在proc.c中实现，再在sysfile.c或者sysproc.c中实现（即预处理数据后直接调用proc.c中的实现）。

编写完以上所有后，一个系统调用就建立了。

### trace

trace的实现比较简单，要点在于给proc结构体（储存进程所有状态的结构体）添加一个32位的掩码，每一位就对应一个系统调用的追踪启用与否。

在系统调用过程中，所有系统调用都会统一地在syscall.c中的syscall函数内执行。那么追踪系统调用在这里进行就可以了。

在对trace一个进程之后，trace便会把输入的掩码参数赋值给该进程的proc结构体。该进程的所有系统调用在syscall（void）中执行完毕后都会判断刚刚的系统调用是否被掩码所涵盖。如果是，那么就相应地print相关信息。

此外，为了追踪子进程，fork（）在产生子进程时，将复制父进程的掩码。

附代码如下：

syscall函数：

if((1<<num)&p->tracemask){

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

    }

fork函数:

np->tracemask=p->tracemask;

trace函数：

uint64

sys\_trace(void)

{

  struct proc\* p=myproc();

  argint(0,(int\*)&(p->tracemask));

  return 0;

}

### sysinfo

sysinfo就相当简单了，理解到proc的存储方式和内存分配器的原理即可

需要注意的是copyout函数的使用，作用是把内核空间的数据复制到指定的用户空间中。因为sysinfo所收集的信息都是在内核中保存的，所以这个转换是必须的。

copyout四个参数的含义分别为：该进程的用户-内核地址页表，用户空间目的地址，内核空间原地址，拷贝大小。

sysinfo代码如下：

uint64

sys\_sysinfo(void)

{

  uint64 sysinfo;

  struct proc \*p=myproc();

  struct sysinfo info;

  info.freemem=getfreemem();

  info.nproc=getatvproc();

  argaddr(0,&sysinfo);

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

    return -1;

  return 0;

}

getfreemem（）实现如下：

实际上就是数一遍freelist有多少个节点。跑出来的结果是133378048B(约合127MB)，也就是系统内核占用了不到1MB的大小。

int

getfreemem(void){

  struct run \*iter=kmem.freelist;

  int count=0;

  while(iter){

    iter=iter->next;

    count++;

  }

  return (uint64)count\*PGSIZE;

}

getatvproc实现如下：

实际就是遍历proc数组，把状态不为unused的进程计数。

int

getatvproc(){

  int count=0;

  for(struct proc \*p=proc;p<&proc[NPROC];p++){

    if(p->state!=UNUSED){

      count++;

    }

  }

  return count;

}

———————————————————————————————————————

### 总结：

lab1的syscall实验确实是不读源码是做不出来的，结合xv6第二章和第四章的部分内容，理解了进程的状态存储和基本的内存管理之后，两个实验的难度都不是很高。

在记录xv6第二章学习笔记的时候，因为它描述的是非常具象化的一个过程，一度导致我无从下笔……我认为这部分内容在后续章节应该会有刨根知底的讨论。