Readme

@Liu_Yuanchi TJU-CS-OS(~MIT 6.828 xv6 LAB3) 2022.11
系个人整理，用于实验报告，如有错误请至github提交issue或致邮斧正

XV6源码结构

1.Perl说明

Perl语言文件即.pl文件，作用是作为跳板脚本，生成usys.S(汇编)，定义了每个 system call 的用户态跳板函数

```
my $name = shift;
    print ".global $name\n";
    print "${name}:\n";
    print " li a7, SYS_${name}\n";
    print " ecall\n";
    print " ret\n";
```

> li a7, SYS_${name}
> ecall
> ret

此段代码的作用是将系统调用编号存入 a7 (陷入) 调用syscall ,注：CPU提权是靠ecall指令，由硬件实现（qemu模拟），xv6在32位保护模式下运行

2.调用流程

由于需要实现内核态与用户态的隔离，所以封装操作很多，调用也极其繁琐

```
user/user.h -> user/usys.S
                  |_ kernel/syscall.c -> kernel/sysproc.c
```

大致实现流程是用户态的 trace() 借由usys.S的ecall调用内核态中的 syscall() ; syscall()再查表，执行具体的内核代码

而在XV6中文文档中有如下一段(第三章)

> 系统调用的实现（例如，sysproc.c 和 sysfile.c）仅仅是封装而已：他们用 argint，argptr 和 argstr 来解析参数，然后调用真正的实现。在第二章，sys_exec 利用这些函数来获取参数。

简单来说，由于特权模式的不同，用户态不能直接传递参数或者地址指针到内核态，而是需要中断（陷入）帧，检查函数等工具向内核请求系统调用 具体实现和原理可参见中文手册第二章和第三章，这里不再赘述

下面是传递系统调用的线程结构体

```
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  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

  // wait_lock must be held when using this:
  struct proc *parent;         // Parent process

  // 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)
};
```

3.具体文件作用

1.修改用户接口代码
user.h的作用：记录系统函数，添加函数；
usys.pl:添加函数入口

2.修改内核

syscall.h:定义系统调用编号
syscall.h 定义系统调用编号（宏）
syscall.c 定义系统调用名称（char*）和外部声明
user.h 用户态函数定义
usys.S 实现
sysproc.c 系统调用函数实现

XV6源码实现

TASK-1 核心函数：

kernel/syscall.c > syscall(void)

```
void syscall(void)
{
  int num;
  struct proc *p = myproc();

  num = p->trapframe->a7;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    // Use num to lookup the system call function for num, call it,
    // and store its return value in p->trapframe->a0
    p->trapframe->a0 = syscalls[num]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

- [ ] 的键值对（C语法糖） 来找到系统函数调用的编号和名称，所以syscalls[num]( )即是num所指代的系统函数
  对于 p->trapframe->a0 这是CPU陷入的指定页帧，具体实现和地址转换和内存结构有关；

这里首先顺着修改中断页帧修改线程标识结构体，使之能够记录所要追踪的系统调用；

```
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)
  uint64 syscall_trace;
}
```

新添加 *uint64 syscall_trace* 的用于标识追踪哪些 system call 的 mask

然后在内核中添加系统级函数 sys_trace() 将参数整合进所监视的进程中

kernel/sysproc.c

```
uint64 sys_trace(void)
{
    int mask;
    if(argint(0, &mask) < 0)
        return -1;
    myproc()->syscall_trace = mask;
    return 0;
}
```

> 附录：小函数*argint*
> 作用是提取第n个寄存器的信息，并用int型指针指向它
>
> ```
> int
> argint(int n, int *ip)
> {
>  *ip = argraw(n);
>  return 0;
> }
> ```

接下来需要修改头文件，添加系统函数编号
kernel/syscall.h

```
#define SYS_trace  22
```

并异声明内核函数，并在函数键值对（映射）中添加Trace,同时新声明一个系统调用与其名称（字符串形式）的映射关系，也用键值对的数据结构即可

kernel/syscall.c

```
extern uint64 sys_trace(void);
[SYS_trace]   sys_trace,
```

```
const char *syscall_names[] = {
[SYS_fork]    "fork",
[SYS_exit]    "exit",
[SYS_wait]    "wait",
[SYS_pipe]    "pipe",
[SYS_read]    "read",
[SYS_kill]    "kill",
[SYS_exec]    "exec",
```

```
[SYS_fstat]   "fstat",
[SYS_chdir]   "chdir",
[SYS_dup]     "dup",
[SYS_getpid]  "getpid",
[SYS_sbrk]    "sbrk",
[SYS_sleep]   "sleep",
[SYS_uptime]  "uptime",
[SYS_open]    "open",
[SYS_write]   "write",
[SYS_mknod]   "mknod",
[SYS_unlink]  "unlink",
[SYS_link]    "link",
[SYS_mkdir]   "mkdir",
[SYS_close]   "close",
[SYS_trace]   "trace",
};
```

加入用户态到内核态的跳板函数

usys.pl

```
entry("trace");
```

在用户态下的头文件加入声明,使用户态程序可以找到函数入口

user/user.h

```
int trace(int);
```

然后在初始化线程描述结构体时同时初始化trace的mask

kernel/proc.c > allocproc()

```
 p->syscall_trace = 0;
```

然后在fork函数中添加对mask的继承，这样才能有效监视

kernel/proc.c > fork

```
  ......

  safestrcpy(np->name, p->name, sizeof(p->name));
  np->syscall_trace = p->syscall_trace;
  pid = np->pid;
```

```
    ......
```

最后处理 syscall() 函数，由于所有的系统调用经由syscall()唤起。通过系统调用编号，获取系统调用处理函数的指针，调用并将返回值存到用户进程的 a0 寄存器中，（加一个检测）如果当前进程设置了对该编号系统调用的 trace，则打出 pid、系统调用名称和返回值。

kernel/syscall.c > syscall

```
if(num > 0 && num < NELEM(syscalls) && syscalls[num]) // 如果系统调用编号有效
{
    p->trapframe->a0 = syscalls[num]();
    if((p->syscall_trace >> num) & 1)
    {
      printf("%d: syscall %s -> %d\n",p->pid, syscall_names[num], p->trapframe-
>a0);
    }
}
```

编译执行详见测试部分

TASK-2 核心函数：

def.h是内核级函数文件：包含 kalloc 等内存操作函数
添加调用的过程与任务1类似

添加入口

usys.pl

```
entry("sysinfo");
```

添加调用码

syscall.h

```
#define SYS_sysinfo 23
```

添加声明

user.h

```
int sysinfo(struct sysinfo*);
```

kernel/defs.h > 'kalloc.c'

```
uint64          count_free_mem(void);
```

在 kalloc.c 中添加计算空闲内存的函数：

```
uint64
count_free_mem(void) // added for counting free memory in bytes (lab2)
{
  //先锁内存管理结构，防止计算过程中出现冲突
  acquire(&kmem.lock);

  // 统计空闲页数，乘上页大小 PGSIZE 就是空闲的内存字节数
  uint64 mem_bytes = 0;
  struct run *r = kmem.freelist;
  while(r){
    mem_bytes += PGSIZE;
    r = r->next;
  }

  release(&kmem.lock);

  return mem_bytes;
}
```

syscall.c

```
#include "sysinfo.h"
```

xv6中对内存操作有如下的描述

> 函数 kfree（2815）首先将被释放内存的每一字节设为 1。这使得访问已被释放内存的代码所读到的不是原有数据，而是垃圾数据；这样做能让这种错误的代码尽早崩溃。接下来 kfree 把 v 转换为一个指向结构体 struct run 的指针，在 r->next 中保存原有空闲链表的表头，然后将当前的空闲链表设置为 r。kalloc 移除并返回空闲链表的表头。

这里 xv6 采用的是空闲链表法。

xv6 中，空闲内存页的记录方式是，将空虚内存页本身直接用作链表节点，形成一个空闲页链表，每次需要分配，就把链表根部对应的页分配出去。每次需要回收，就把这个页作为新的根节点，把原来的 freelist 链表接到后面。注意这里是直接使用空闲页本身作为链表节点，所以不需要使用额外空间来存储空闲页链表，在 kalloc() 里也可以看到，分配内存的最后一个阶段，是直接将 freelist 的根节点地址（物理地址）返回出去了：

```
void *
kalloc(void)
```

```
{
  struct run *r;

  acquire(&kmem.lock);
  r = kmem.freelist; // 获得空闲页链表的根节点
  if(r)
    kmem.freelist = r->next;
  release(&kmem.lock);

  if(r)
    memset((char*)r, 5, PGSIZE); // fill with junk
  return (void*)r; // 把空闲页链表的根节点返回出去，作为内存页使用（长度是 4096）
}
```

对于内存进程数的获取也是类似的

proc.c

```
uint64
count_process(void) { // added function for counting used process slots (lab2)
  uint64 cnt = 0;
  for(struct proc *p = proc; p < &proc[NPROC]; p++) {
    // acquire(&p->lock);
    // 不需要锁进程 proc 结构，因为只需要读取进程列表，不需要写
    if(p->state != UNUSED) { // 不是 UNUSED 的进程位，就是已经分配的
        cnt++;
    }
  }
  return cnt;
}
```

最后实现系统调用即可

sysproc.c

```
uint64
sys_sysinfo(void)
{
  // 从用户态读入一个指针，作为存放 sysinfo 结构的缓冲区
  uint64 addr;
  if(argaddr(0, &addr) < 0)
    return -1;

  struct sysinfo sinfo;
  sinfo.freemem = count_free_mem(); // kalloc.c
  sinfo.nproc = count_process(); // proc.c

  // 使用 copyout，结合当前进程的页表，获得进程传进来的指针（逻辑地址）对应的物理地址
  // 然后将 &sinfo 中的数据复制到该指针所指位置，供用户进程使用。
  if(copyout(myproc()->pagetable, addr, (char *)&sinfo, sizeof(sinfo)) < 0)
```

```
      return -1;
    return 0;
  }
```

内核编译与运行

1.修改makefile

> Add $U/_trace to UPROGS in Makefile
> Add $U/_trace to UPROGS in Makefile

makefile语法此次不再赘述
make qemu 在 qemu 模拟器运行即可
按操作指南测试如下命令（已写为shell脚本附在文件夹里）

> trace 32 grep hello README
> trace 2147483647 grep hello README
> grep hello README
> trace 2 usertests forkforkfork
> sysinfotest

运行结果详见/asset