# 实验报告

## Lab 2

# 1.实验过程

## Part A: `Trace`

先按照要求，与lab1类似地修改`Makefile`。

```
UPROGS=\
        $U/_cat\
        $U/_echo\
        $U/_forktest\
        $U/_grep\
        $U/_init\
        $U/_kill\
        $U/_ln\
        $U/_ls\
        $U/_mkdir\
        $U/_rm\
        $U/_sh\
        $U/_stressfs\
        $U/_usertests\
        $U/_grind\
        $U/_wc\
        $U/_zombie\
        $U/_trace\
```

我们先看给定的函数`user/trace.c`，第17行:

```
    if (trace(atoi(argv[1])) < 0)
```

可以看到其调用了函数trace，这就是我们需要添加的**系统调用**。
为了添加新的系统调用，我们首先需要知道在xv6中，系统调用是以怎样的路径实现的。

## A.1 系统调用原理

在上面的`trace.c`中，我们调用了`trace`函数。那么这个函数的声明和定义是怎样的呢?

在`user/user.h`中，可以看到这里有各种系统调用的声明，使得用户程序能够**显式调用**这些函数。在这里我们也需要为需要的`trace`函数添加声明。

```
// system calls
...
//add new declarations
int trace(int);
```

`user/usys.pl`用于生成`usys.S`汇编代码，其作用为提供系统调用的入口。同样在这里添加一个entry。

```
# Generate usys.S, the stubs for syscalls.
print "# generated by usys.pl - do not edit\n";

print "#include \"kernel/syscall.h\"\n";

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

...
# add new entrys
entry("trace");
```

这里我们关注sub entry这段代码。因为pl文件代表perl语言，sub在perl中代表一段子程序(函数)，这段子程序的作用为在`usys.S`中写入这段汇编代码。而这段汇编代码的作用为:将传入的系统调用名(shift)作为$name，声明其为.global的，其表示$name作为一个全局符号，这样**在调用用户函数时能将其识别，并转入执行下面的代码，从而完成了从用户函数到系统调用的转换**。将SYS_${name}的低6位写入a7寄存器。ecall指令是一条**陷阱指令(trap)**，用于从用户态转换到内核态。ret是一条"伪指令"，作为函数的结尾。

注意最开头的一句`#include \"kernel/syscall.h\"\n`,即usys.S文件包含了`kernel/syscall.h`。查

看`kernel/syscall.h`就能发现，该头文件是用于将SYS_${name}替换成具体的值的。相应的这里我们也需要为新系统调用`sys_trace`添加定义。

```
// System call numbers
...
// add new definition
#define SYS_trace  22
```

根据xv6 book 4.3:

> The `ecall` instruction traps into the kernel and causes `uservec`, `usertrap`, and then `syscall` to execute, as we saw above.

`uservec`位于`kernel/trampoline.S`。该程序段将用户态的页表转换为内核态的页表。

`usertrap`位于`kernel/trap.c`，该函数是一个中断处理函数，并判断是否执行了来自用户的系统调用，如果是，则调用`syscall`函数。执行完毕后，执行同文件中的`usertrapret`函数，用于从内核态回到用户态。（调用`userret`）

`trampoline.S`中的`userret`段用于将内核页表转换为用户页表，并返回用户态。

现在我们把目光转向kernel/`syscall.c`。该文件是执行系统调用的主体。

```
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;
  }
}
```

`syscall`函数是系统调用的主函数。其从进程的 **陷阱帧(trapframe)** 的a7寄存器获取之前我们存入的系统调用名对应的数字(如`SYS_trace`对应21)，并存入`num`变量。当获取的`num`正常时，使用相应的系

统调用函数，并将返回值存在trapframe的a0寄存器。

在`syscall.c`中还有两个部分，一个是**声明系统调用函数原型**部分，另一个是**将系统调用号转换为系统调用函数的函数指针**部分。

```
// Prototypes for the functions that handle system calls.
...
// add new prototypes
extern uint64 sys_trace(void);

// An array mapping syscall numbers from syscall.h
// to the function that handles the system call.
static uint64 (*syscalls[])(void) = {
...
// add new function pointers
[SYS_trace]   sys_trace,
};
```

首先我们声明新系统调用函数，由于这些函数都是**`extern`**的，所以可以从外界引用。然后在`syscalls`函数指针数组中添加从`SYS_trace`(由于`syscall.c`包含了`syscall.h`，`SYS_trace`被转换为22)到函数`sys_trace`的函数指针。

最后，我们需要实际实现`sys_trace`函数，这需要在`sysproc.c`中添加。

## A.2 `sys_trace`函数实现

根据实验要求,`sys_trace`函数应当接收一个int类型参数mask,**该参数各位置上的1代表哪些系统调用需要跟踪(如0x0000 0003,对应3号系统调用sys_wait)**。当系统调用返回时，应**输出一行标识包含其pid、系统调用名和返回值**。

### A.2.1 `sys_trace`函数：获取参数

`sys_trace`函数需要接收一个参数，但作为系统调用函数，其接收参数都是void。要获取参数需要通过`syscall.c`中提供的函数`argint`、`argaddr`和`argstr`，它们将an位置寄存器的值取出并放到传入的参数中，具体实现于`argraw`函数中。

对于`sys_trace`函数，其只有一个参数，从用户态传入的参数被依次放入a0、a1...寄存器。因此只需要

```
int m;
argint(0,&m);
```

就能将参数`mask`放入到`m`中。

### A.2.2 修改proc结构

根据实验指示，我们需要修改位于`kernel/proc.h`中的`proc`结构的定义，添加一个相应的变量记录`mask`，指定该进程需要跟踪的系统调用。

```
// Per-process state
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)
  int mask;
};
```

### A.2.3 `sys_trace`函数

我们只需要将获取的`mask`传入到`proc`变量中即可。这里`myproc()`代表当前进程，其定义可见`kernel/proc.c`。

```
uint64
sys_trace(void)
{
  int m;

  argint(0,&m);
  myproc()->mask=m;
  return 0;
}
```

### A.2.4 修改 `fork()` 函数

`fork()` 函数位于 `kernel/proc.c` 中。当调用 `fork()` 函数时，当前进程会创建一个子进程。我们需要让**生成的子进程继承父进程的 `mask` 值**，表示子进程应与父进程跟踪同样的系统调用。

```
// Create a new process, copying the parent.
// Sets up child kernel stack to return as if from fork() system call.
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *p = myproc();
  ...
  // copy mask from parent process.
  np->mask=p->mask;

  return pid;
}
```

### A.2.5 修改 `syscall()` 函数

根据实验要求，我们需要修改 `syscall()` 函数以输出跟踪结果。这里需要添加一个系统调用名的数组作为输出的系统调用名的来源，因为 `proc` 变量中存储的 `name` 变量保存的是进程名(如 `grep` 命令对应的进程名就是 `grep` )而非系统调用名。因此需要添加这样的数组。

这里注意系统调用名从1开始，所以下标0的位置设为空字符串。

```
//notice that name start from 1
static char* syscall_names[23]={"",
  "fork","exit","wait","pipe","read",
  "kill","exec","fstat","chdir","dup",
  "getpid","sbrk","sleep","uptime","open",
  "write","mknod","unlink","link","mkdir",
  "close","trace"
  };
```

然后是具体输出部分。根据`mask`的语义，我们让其与`1<<num`位与，这样只要`num`对应的系统调用在`mask`追踪内容中，就输出相应的语句。

```
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]();
    if(p->mask & (1<<num))
        printf("%d:syscall %s -> %d\n",p->pid,syscall_names[num],p->trapframe->a0);
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

## A.3 实验结果

1. 执行`trace 32 grep hello README`的结果如图:

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ trace 32 grep hello README
3:syscall read -> 1023
3:syscall read -> 961
3:syscall read -> 321
3:syscall read -> 0
$
```

该条指令的语义为，执行命令`grep hello README`(其语义为在README文件中搜索hello，具体见`user/grep.c`)，并追踪系统调用sys_read($2^5 = 32$)。

2. 执行`trace 2147483647 grep hello README`的结果如图:

```
$ trace 2147483647 grep hello README
7:syscall trace -> 0
7:syscall exec -> 3
7:syscall open -> 3
7:syscall read -> 1023
7:syscall read -> 961
7:syscall read -> 321
7:syscall read -> 0
7:syscall close -> 0
$
```

该条指令执行的内容仍为`grep hello README`，只是`mask`为2147483647($2^{31} - 1$，即0x7FFFFFFF),表示**追踪所有系统调用**。可以看到，除了`read`外还有系统调用`trace`（本身)、`exec`、`open`和`close`。

3. 执行`grep hello README`的结果如图:

```
$ grep hello README
$
```

因为实际上在`README`中没有hello这段文本，所以什么都得不到。

4. 执行`trace 2 usertests forkforkfork`的结果如图:

```
$ trace 2 usertests forkforkfork
usertests starting
3:syscall fork -> 4
test forkforkfork: 3:syscall fork -> 5
5:syscall fork -> 6
6:syscall fork -> 7
6:syscall fork -> 8
7:syscall fork -> 9
6:syscall fork -> 10
7:syscall fork -> 11
8:syscall fork -> 12
7:syscall fork -> 13
7:syscall fork -> 14
9:syscall fork -> 15
7:syscall fork -> 16
6:syscall fork -> 17
7:syscall fork -> 18
7:syscall fork -> 19
7:syscall fork -> 20
8:syscall fork -> 21
7:syscall fork -> 22
10:syscall fork -> 23
7:syscall fork -> 24
9:syscall fork -> 25
7:syscall fork -> 26
6:syscall fork -> 27
15:syscall fork -> 28
6:syscall fork -> 29
7:syscall fork -> 30
6:syscall fork -> 31
7:syscall fork -> 32
6:syscall fork -> 33
7:syscall fork -> 35
8:syscall fork -> 34
6:syscall fork -> 36
8:syscall fork -> 37
6:syscall fork -> 38
11:syscall fork -> 39
6:syscall fork -> 40
12:syscall fork -> 41
31:syscall fork -> 42
7:syscall fork -> 43
6:syscall fork -> 44
7:syscall fork -> 45
6:syscall fork -> 46
34:syscall fork -> 47
25:syscall fork -> 48
26:syscall fork -> 49
25:syscall fork -> 50
25:syscall fork -> 51
26:syscall fork -> 52
25:syscall fork -> 53
25:syscall fork -> 54
34:syscall fork -> 55
10:syscall fork -> 56
15:syscall fork -> 57
10:syscall fork -> 58
16:syscall fork -> 59
10:syscall fork -> 60
16:syscall fork -> 61
18:syscall fork -> 62
33:syscall fork -> 63
18:syscall fork -> 64
33:syscall fork -> 65
36:syscall fork -> -1
13:syscall fork -> -1
33:syscall fork -> -1
OK
3:syscall fork -> 66
ALL TESTS PASSED
$
```

该条指令追踪`sys_exit`系统调用,执行命令`usertests forkforkfork`，即调用`user/usertests.c`中的`forkforkfork`函数。该函数会一直`fork`，直到无法`fork`为止（达到最大进程数`NPROC=64`,详见`kernel/param.h`）。`fork`的返回值最大达到65后就返回-1了，表示不能创建进程了。最终返回66，相应进程数为66-3+1=64，这与实验结果吻合。

# Part B: `Sysinfo`

按照与Part A完全相同的步骤做好铺垫(修改`Makefile`、`user/user.h`、`user/usys.pl`、`kernel/syscall.h`、`kernel/syscall.c`)。注意在`user/user.h`中声明函数时要先声明结构体sysinfo。

```
UPROGS=\
        $U/_cat\
        $U/_echo\
        $U/_forktest\
        $U/_grep\
        $U/_init\
        $U/_kill\
        $U/_ln\
        $U/_ls\
        $U/_mkdir\
        $U/_rm\
        $U/_sh\
        $U/_stressfs\
        $U/_usertests\
        $U/_grind\
        $U/_wc\
        $U/_zombie\
        $U/_trace\
        $U/_sysinfotest\
```

```
// 1.user/user.h
struct sysinfo;

// system calls
...
// add new declarations
int trace(int);
int sysinfo(struct sysinfo*);
// 2. kernel/syscall.h
...
// add new definitions
#define SYS_trace  22
#define SYS_sysinfo  23
// 3. kernel/syscall.c
// Prototypes for the functions that handle system calls.
...
// add new prototypes
extern uint64 sys_trace(void);
extern uint64 sys_sysinfo(void);

// An array mapping syscall numbers from syscall.h
// to the function that handles the system call.
static uint64 (*syscalls[])(void) = {
...
// add new function pointers
[SYS_trace]   sys_trace,
[SYS_sysinfo]     sys_sysinfo,
};

 //notice that name start from 1
static char* syscall_names[24]={"",
  "fork","exit","wait","pipe","read",
  "kill","exec","fstat","chdir","dup",
  "getpid","sbrk","sleep","uptime","open",
  "write","mknod","unlink","link","mkdir",
  "close","trace","sysinfo"
  };
```

```
# 4.user/usys.pl
# add new entrys
entry("trace");
entry("sysinfo");
```

## B.1 实现 `sys_sysinfo` 函数

按照实验要求，`sysinfo` 函数接收一个指向sysinfo结构(见 `kernel/sysinfo.h` )的指针，并将该结构体的内容补充完整。

```
struct sysinfo {
  uint64 freemem;   // amount of free memory (bytes)
  uint64 nproc;     // number of process
};
```

需要补充的信息为空闲内存字节数和状态不为 `UNUSED` 的进程数。与 `sys_trace` 一样，我们同样需要在 `sysproc.c` 中实现它。

### B.1.1 `sys_sysinfo` 函数主体

类似 `sys_trace` 函数的实现，首先需要获取传入的参数 `sysinfo*` 。利用 `argaddr` 函数就能做到这点。并且需要创建一个 `sysinfo` 类型的变量 `info` 暂时保存结果。

```
uint64
sys_sysinfo(void)
{
    struct sysinfo info;
    uint64 ip;
    argaddr(0,&ip);
}
```

接下来需要分别实现**计算剩余内存字节的函数**和**计算活跃进程数的函数**。

### B.1.2 计算剩余内存字节: `leftMemory()` 函数

按照实验要求，需要在 `kernel/kalloc.c` 中添加相应的新函数。
在 `kalloc.c` 头部，我们能找到两个结构体的定义:

```
struct run {
  struct run *next;
};

struct {
  struct spinlock lock;
  struct run *freelist;
} kmem;
```

`run`结构体就是普通的链表，`kmem`结构体含有一个`spinlock`和一个相应的链表`freelist`，代表相应的空闲内存链表，每个节点对应一页。因此，如果需要知道剩余内存数，只需要遍历链表得到链表节点数即可。

相应函数的写法可以参考`kalloc.c`中的`kalloc`函数，相应代码如下:

```
uint64 leftMemory(void)
{
  struct run *r;

  acquire(&kmem.lock);
  r = kmem.freelist;
  uint64 num=0;
  //the length of the list
  while(r)
  {
    ++num;
    r=r->next;
  }
  release(&kmem.lock);

  return num*PGSIZE;
}
```

其中`PGSIZE`定义于`kernel/riscv.h`，值为4096，代表每页字节数。

因为`kalloc.c`引用了`defs.h`，所以还需要在`defs.h`中添加声明。

```
// kalloc.c
void*           kalloc(void);
void            kfree(void *);
void            kinit(void);
uint64      leftMemory(void);
```

### B.1.3 实现活跃进程计数: `procNum()` 函数

按要求在 `proc.c` 中添加新函数。
在 `proc.c` 开头能找到定义的进程数组 `proc[NPROC]`，其中NPROC=64。

```
struct proc proc[NPROC];
```

这个数组就是用于存储所有进程的。因此只需要遍历整个数组考察状态是否为 `UNUSED` 即可。
相应函数的写法可以参考同文件下的任意函数(如 `kill` 函数)。

```
uint64 procNum(void)
{
  struct proc *p;
  uint64 num=0;
  for(p=proc;p<&proc[NPROC];p++)
  {
    acquire(&p->lock);
    if(p->state!=UNUSED)
        ++num;
    release(&p->lock);
  }
  return num;
}
```

同样地需要添加声明于 `defs.h`。

```
// proc.c
...
uint64      procNum(void);
```

### B.1.4 利用 `copyout` 函数将结果传回用户态

根据提示，我们可以在 `kernel/sysfile.c` 找到 `sys_fstat` 函数。

```
uint64
sys_fstat(void)
{
  struct file *f;
  uint64 st; // user pointer to struct stat

  argaddr(1, &st);
  if(argfd(0, 0, &f) < 0)
    return -1;
  return filestat(f, st);
}
```

调用的 `filestat` 函数位于 `file.c` 中：

```
// Get metadata about file f.
// addr is a user virtual address, pointing to a struct stat.
int
filestat(struct file *f, uint64 addr)
{
  struct proc *p = myproc();
  struct stat st;

  if(f->type == FD_INODE || f->type == FD_DEVICE){
    ilock(f->ip);
    stati(f->ip, &st);
    iunlock(f->ip);
    if(copyout(p->pagetable, addr, (char *)&st, sizeof(st)) < 0)
      return -1;
    return 0;
  }
  return -1;
}
```

需要关注的语句为 `if(copyout(p->pagetable, addr, (char *)&st, sizeof(st)) < 0)` 。该函数的定义位于 `kernel/vm.c` 中。

```
// Copy from kernel to user.
// Copy len bytes from src to virtual address dstva in a given page table.
// Return 0 on success, -1 on error.
int
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
{
  uint64 n, va0, pa0;

  while(len > 0){
    va0 = PGROUNDDOWN(dstva);
    pa0 = walkaddr(pagetable, va0);
    if(pa0 == 0)
      return -1;
    n = PGSIZE - (dstva - va0);
    if(n > len)
      n = len;
    memmove((void *)(pa0 + (dstva - va0)), src, n);

    len -= n;
    src += n;
    dstva = va0 + PGSIZE;
  }
  return 0;
}
```

可以看到`copyout`函数的作用是将内容从内核复制到用户态，它接收的参数分别为:

> pagetable：进程页表
> dstva：目标地址
> src：源内容
> len：复制长度

因此在`sys_sysinfo`函数中，可以参考`filestat`函数中`copyout`的使用方式进行实现。我们需要将`info`变量传递到接收到的地址`ip`上。所以相应调用应为:`copyout(myproc()->pagetable,ip,(char*)info,sizeof(info))`。

### B.1.5 增加学号输出

&emsp按照要求，需要在`kernel/sysinfo.h`中添加一个字符串类型变量代表学号，并在每次运行`sys_sysinfo`函数时将其输出。

```
struct sysinfo {
  uint64 freemem;   // amount of free memory (bytes)
  uint64 nproc;     // number of process
};
char* studentID = "20307130231";
```

### B.1.6 sys_sysinfo函数完整实现

```
uint64
sys_sysinfo(void)
{
  struct sysinfo info;
  uint64 ip;
  argaddr(0,&ip);
  info.freemem=leftMemory();
  info.nproc=procNum();
  printf("my student number is %s\n",studentID);

  if(copyout(myproc()->pagetable, ip, (char *)&info, sizeof(info)) < 0)
      return -1;
  return 0;
}
```

## B.2 实验结果

执行`sysinfotest`的结果如图：

```
$ sysinfotest
sysinfotest: start
my student number is 20307130231
my student number is 20307130231
my student number is 20307130231
my student number is 20307130231
my student number is 20307130231
my student number is 20307130231
my student number is 20307130231
my student number is 20307130231
my student number is 20307130231
sysinfotest: OK
```

# 2.问题回答

(1)System calls Part A部分，简述一下`trace`全流程.

当在`qemu`中输入`trace`指令时，先通过`Makefile`的`UPROGS`部分找到相应的文件`user/trace.c`，然后从`trace.c`的`main`函数开始执行。

`main`函数中调用了`trace`函数，该函数在`user/user.h`中声明，并通过`user/usys.S`的`ecall`指令转入内核态执行。相应的系统调用号被存放于a7寄存器。

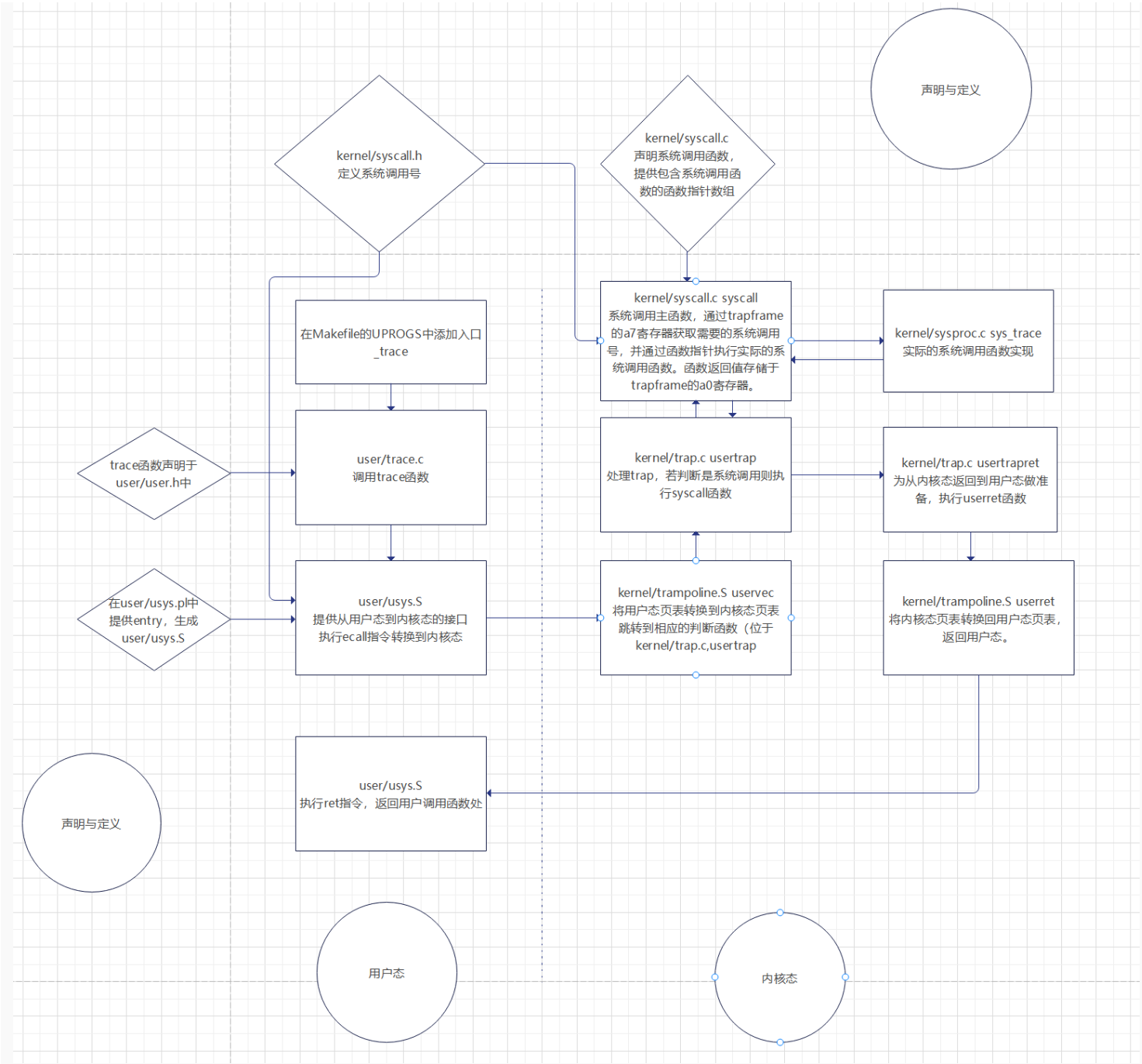
通过执行`uservec`和`usertrap`两个函数从用户态转换到内核态，并调用`syscall`函数。

转入内核态后，`kernel/syscall.c`的`syscall`函数从`trapframe`的a7寄存器获取系统调用号，并获取相应的函数指针以执行实际的系统调用函数`sys_trace`。

在`kernel/sysproc.c`中实际实现函数`sys_trace`。执行完后`syscall`函数将返回值置于`trapframe`的a0寄存器。

执行完毕后需要从内核态返回到用户态。通过`usertrapret`和`userret`函数返回到用户态，并通过`user/usys.S`的`ret`指令返回到调用函数的位置。

具体执行流程图如下:




(2) `kernel/syscall.h` 是干什么的，如何起作用的?

`kernel/syscall.h` 提供了系统调用号。通过在 `usys.pl` 、 `syscall.c` 中将相应的 `SYS_${name}` 转换为相应的系统调用号起作用。

（3）命令 “trace 32 grep hello README”中的trace字段是用户态下的还是实现的系统调用函数trace?

是用户态下的 `user/trace.c` 文件。通过(1)中描述的trace流程调用实际的 `sys_trace` 函数。

# 3. 实验问题与解决

主要问题是在对执行系统调用的流程刚开始不太明白，因为转换步骤本身很多也很繁杂。解决方法：仔细阅读了xv6 book的相关内容(2.6、4.3、4.4)，并根据其描述的代码，在github存放源实验文件的repository下搜索需要的函数，这大大提高了效率。通过阅读相应的函数的描述，加深对系统调用的理解。同时也在网上查了一些有关xv6系统是怎样执行系统调用的资料。

另一个问题在于这两个问题的具体实现上。因为以前并没有写过这种偏“底层”的C代码，所以写起来比较磕磕绊绊的。解决方法：通过pdf给的提示，模仿同文件下的函数写法，并仔细考察相关定义的结构体。

# 4. 实验感想

本实验做起来还是比较费时的，原因就在于对操作系统的系统调用不好理解上。不过我对操作系统的部分运转机制的了解也加深了很多。以后的实验中，随着对操作系统理解的加深与对这种实验方式的习惯，这样的现象应该会有所好转。

# 5. 参考资料


1. xv6实验的源代码仓库，帮助我能快速搜索需要的内容。
2. xv6-book-riscv-rec2，xv6的参考手册。
3. xv6系统调用实现，详细介绍了系统调用的实现，包括如何从用户态转换到内核态。
4. 如何用Vscode调试xv6，在Ubuntu中用Vscode调试xv6代码，给调试带来了很大的方便。