
# 实验二：系统调用

《操作系统》课程实验

# 目录

# 实验目的

- 了解xv6系统调用的工作原理。
- 熟悉xv6通过系统调用给用户程序提供服务的机制。

# 实验任务 | 任务1：trace系统调用

**请切换到syscall分支，实现两个系统调用：**

① ***trace***系统调用 *int trace(int mask)*

- mask：每一位对应一个系统调用，位的比特值指示是否需要追踪对应的系统调用。
  - 如调用trace(1 << SYS_fork)，则代表只追踪系统调用fork。
  - 如调用trace(0xffffffff)，则代表追踪所有系统调用。
- 返回值：正常执行返回0，异常返回-1。

kernel > C syscall.h

```
1  // System call numbers
2  #define SYS_fork    1
3  #define SYS_exit    2
4  #define SYS_wait    3
5  #define SYS_pipe    4
6  #define SYS_read    5
7  #define SYS_kill    6
8  #define SYS_exec    7
9  #define SYS_fstat   8
10 #define SYS_chdir   9
11 #define SYS_dup     10
12 #define SYS_getpid  11
13 #define SYS_sbrk    12
14 #define SYS_sleep   13
15 #define SYS_uptime  14
16 #define SYS_open    15
17 #define SYS_write   16
18 #define SYS_mknod   17
19 #define SYS_unlink  18
20 #define SYS_link    19
21 #define SYS_mkdir   20
22 #define SYS_close   21
```

# 实验原理 | 任务1：trace系统调用

## ① *trace*系统调用 *int trace(int mask)*

- 内核每处理完一次系统调用后，即系统调用返回前，若mask指示了该系统调用，则打印对应信息。打印格式： *PID: sys_$name(arg0) -return_value*
- *Trace* 用户程序：通过 *trace* 系统调用设置需要跟踪的系统调用，然后启动另一个程序，显示该程序对指定系统调用的情况。

例1

```
$ trace 32 grep hello README
3: sys_read(3) -> 1023
3: sys_read(3) -> 966
3: sys_read(3) -> 70
3: sys_read(3) -> 0
$
```

例2

```
$ trace 2147483647 grep hello README
4: sys_trace(2147483647) -> 0
4: sys_exec(12240) -> 3
4: sys_open(12240) -> 3
4: sys_read(3) -> 1023
4: sys_read(3) -> 966
4: sys_read(3) -> 70
4: sys_read(3) -> 0
4: sys_close(3) -> 0
$
```

例3

```
$ trace 2 usertests forkforkfork
usertests starting
test forkforkfork: 407: syscall fork -> 408
408: sys_fork(-1) -> 409
409: sys_fork(-1) -> 410
410: sys_fork(-1) -> 411
409: sys_fork(-1) -> 412
410: sys_fork(-1) -> 413
409: sys_fork(-1) -> 414
411: sys_fork(-1) -> 415
...
$
```

## ② *sysinfo*系统调用 *int sysinfo(struct sysinfo*)*

- 功能：用于收集xv6运行的一些系统信息
- 参数：结构体 sysinfo的指针
  - freemem：当前剩余的**内存字节数**
  - nproc：**状态为UNUSED**的进程个数
  - freefd：当前进程可用文件描述符的数量，即**尚未使用**的文件描述符数量

```
1  struct sysinfo {
2    uint64 freemem;   // amount of free memory (bytes)
3    uint64 nproc;     // number of process
4    uint64 freefd;    // number of free file descriptor
5  };
```

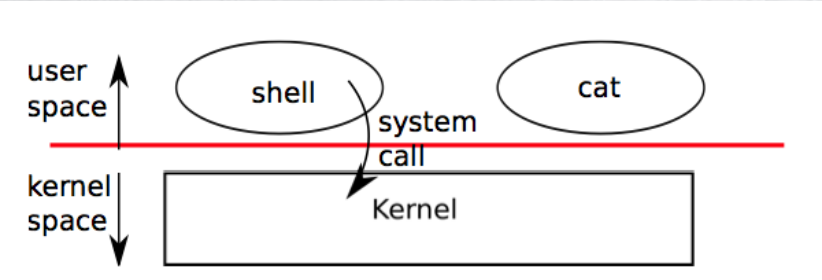
当完成上述的两个实验后，在命令行输入make grade进行测试。

如果通过测试，会显示如下内容：

```
== Test trace 32 grep ==
$ make qemu-gdb
trace 32 grep: OK (5.7s)
== Test trace all grep ==
$ make qemu-gdb
trace all grep: OK (1.0s)
== Test trace nothing ==
$ make qemu-gdb
trace nothing: OK (0.9s)
== Test trace children ==
$ make qemu-gdb
trace children: OK (16.8s)
== Test sysinfotest ==
$ make qemu-gdb
sysinfotest: OK (3.1s)
== Test time ==
time: OK
Score: 35/35
```

# 实验原理 | 系统调用

- 系统调用：操作系统为用户态进程与硬件设备进行交互提供了一组接口
  - 进程管理：复制创建进程 fork、退出进程 exit、执行进程 exec 等
  - 进程间通信：管道 pipe
  - 虚存管理：改变进程内存空间大小 sbrk
  - 文件I/O操作：读 read、写 write、打开 open、关闭 close 等
  - 文件系统：改变当前目录 chdir、创建新目录 mkdir、创建设备文件 mknod等




user > C user.h

```c
struct stat;
struct rtcdate;

// 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(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);
```

# 实验原理 | 系统调用 (以write为例)

user/user.h
xx系统调用函数

用户态

内核态

## user/user.h

user > C user.h

```
1   struct stat;
2   struct rtcdate;
3
4   // system calls
5   int fork(void);
6   int exit(int) __attribute__((noreturn));
7   int wait(int*);
8   int pipe(int*);
9   int write(int, const void*, int);
10  int read(int, void*, int);
11  int close(int);
12  int kill(int);
```

# 实验原理 | 系统调用（以write为例）

user/user.h
xx系统调用函数

usys.S
将系统调用号给寄存器a7，执行ecall指令触发软中断，陷入内核态

用户态

内核态

## user/usys.pl

```
9   sub entry {
10      my $name = shift;
11      print ".global $name\n";
12      print "${name}:\n";
13      print " li a7, SYS_${name}\n";
14      print " ecall\n";
15      print " ret\n";
16  }
17
18  entry("fork");
19  entry("exit");
20  entry("wait");
21  entry("pipe");
22  entry("read");
23  entry("write");
24  entry("close");
```

## user/usys.S

```
28    .global write
29    write:
30     li a7, SYS_write
31     ecall
32     ret
```

# 实验原理 | 系统调用（以write为例）

kernel/trampoline.S

```
12  .globl trampoline
13  trampoline:
14  .align 4
15  .globl uservec
16  uservec:
17          #
18          # trap.c sets stvec to
```

```
77          ld t1, 0(a0)
78          csrw satp, t1
79          sfence.vma zero, zero
80
81          # a0 is no longer val
82          # table does not spec
83
84          # jump to usertrap(),
85          jr t0
```

user/user.h
xx系统调用函数

usys.S
将系统调用号给寄存器a7，执行ecall指令触发软中断，陷入内核态

Kernel/trampoline.S
Uservec段
上下文切换（PCB）
保存user进程数据
恢复kernel寄存器数据

用户态

内核态

# 实验原理 | 系统调用（以write为例）

## kernel/trap.c

```
36  void
37  usertrap(void)
38  {
39    int which_dev = 0;
```

```
53    if(r_scause() == 8){
54      // system call
55
56      if(p->killed)
57        exit(-1);
58
59      // sepc points to the ecall instruction,
60      // but we want to return to the next instruction.
61      p->trapframe->epc += 4;
62
63      // an interrupt will change sstatus &c registers,
64      // so don't enable until done with those registers.
65      intr_on();
66
67      syscall();
68    } else if((which_dev = devintr()) != 0){
69      // ok
70    } else {
71      printf("usertrap(): unexpected scause %p pid=%d\n",
72      printf("            sepc=%p stval=%p\n", r_sepc(),
73      p->killed = 1;
74    }
75
76    if(p->killed)
77      exit(-1);
78
79    // give up the CPU if this is a timer interrupt.
80    if(which_dev == 2)
81      yield();
82
83    usertrapret();
84  }
```

user/user.h
xx系统调用函数

↓

usys.S
将系统调用号给寄存器a7，执行ecall指令触发软中断，陷入内核态

↓

Kernel/trampoline.S
Uservec段
上下文切换（PCB）
保存user进程数据
恢复kernel寄存器数据

↓

usertrap()
中断处理函数
检查是否为来自用户的系统调用

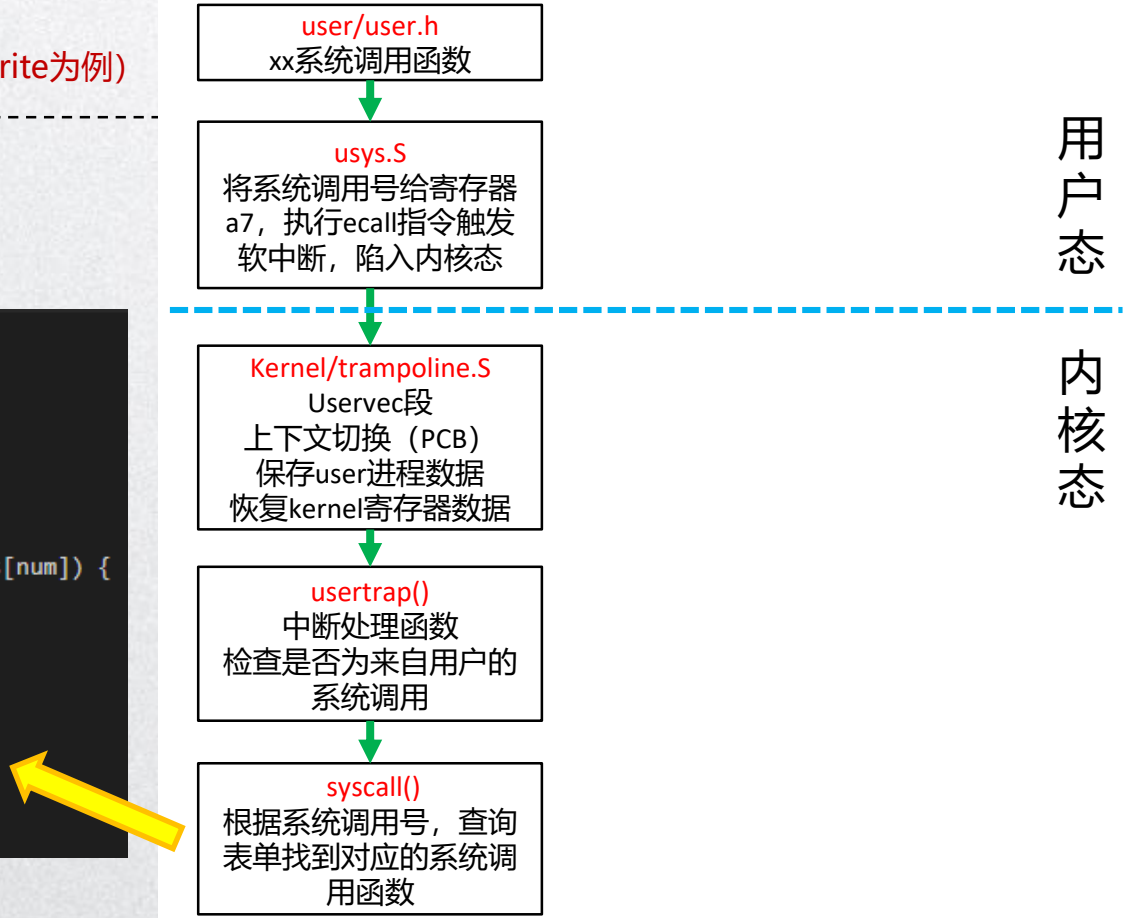
用户态

内核态

# 实验原理 | 系统调用（以write为例）

## kernel/syscall.c

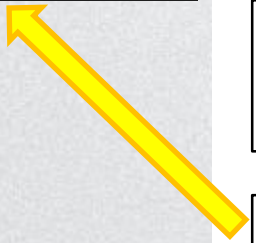
```
132  void
133  syscall(void)
134  {
135    int num;
136    struct proc *p = myproc();
137
138    num = p->trapframe->a7;
139    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
140      p->trapframe->a0 = syscalls[num]();
141    } else {
142      printf("%d %s: unknown sys call %d\n",
143              p->pid, p->name, num);
144      p->trapframe->a0 = -1;
145    }
146  }
147
```

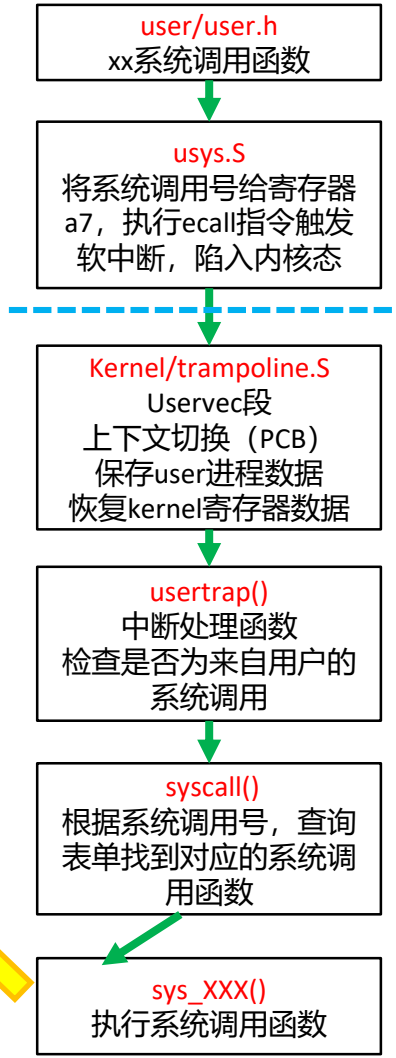


# 实验原理 | 系统调用（以write为例）

kernel/sysfile.c

```
81  uint64
82  sys_write(void)
83  {
84    struct file *f;
85    int n;
86    uint64 p;
87
88    if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argaddr(1, &p) < 0)
89      return -1;
90
91    return filewrite(f, p, n);
92  }
```





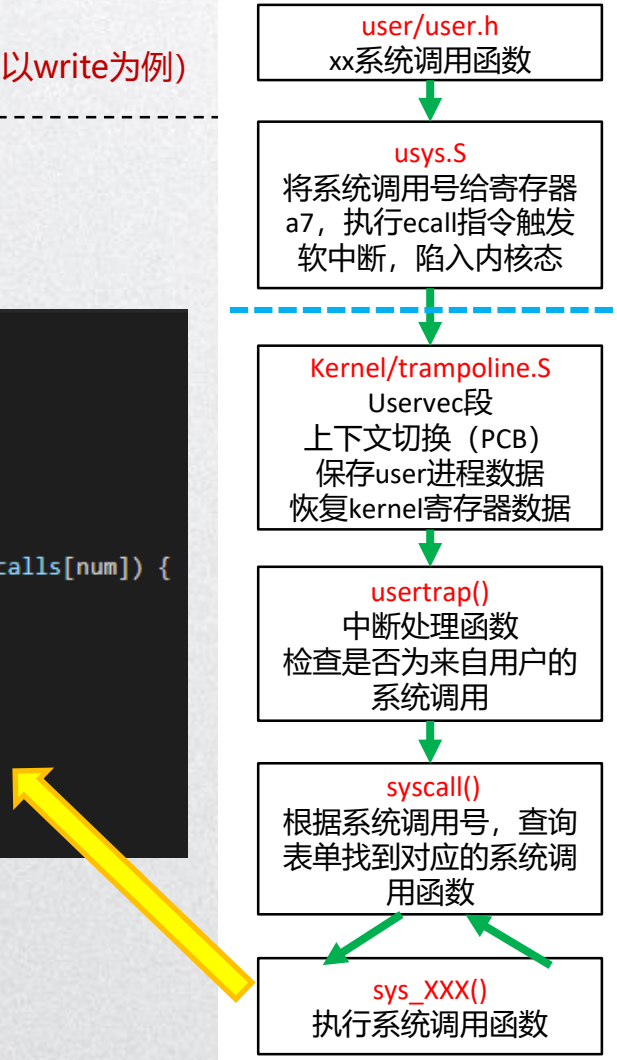

用户态

内核态

# 实验原理 | 系统调用（以write为例）

kernel/syscall.c

```
132  void
133  syscall(void)
134  {
135    int num;
136    struct proc *p = myproc();
137
138    num = p->trapframe->a7;
139    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
140      p->trapframe->a0 = syscalls[num]();
141    } else {
142      printf("%d %s: unknown sys call %d\n",
143              p->pid, p->name, num);
144      p->trapframe->a0 = -1;
145    }
146  }
147
```



# 实验原理 | 系统调用（以write为例）

kernel/trap.c

```
36  void
37  usertrap(void)
38  {
39    int which_dev = 0;
```

```
53    if(r_scause() == 8){
54      // system call
55
56      if(p->killed)
57        exit(-1);
58
59      // sepc points to the ecall instruction,
60      // but we want to return to the next instruction.
61      p->trapframe->epc += 4;
62
63      // an interrupt will change sstatus &c registers,
64      // so don't enable until done with those registers.
65      intr_on();
66
67      syscall();
68    } else if((which_dev = devintr()) != 0){
69      // ok
70    } else {
71      printf("usertrap(): unexpected scause %p pid=%d\n",
72      printf("            sepc=%p stval=%p\n", r_sepc(),
73      p->killed = 1;
74    }
75
76    if(p->killed)
77      exit(-1);
78
79    // give up the CPU if this is a timer interrupt.
80    if(which_dev == 2)
81      yield();
82
83    usertrapret();
84  }
```

user/user.h
xx系统调用函数

usys.S
将系统调用号给寄存器a7，执行ecall指令触发软中断，陷入内核态

用户态

Kernel/trampoline.S
Uservec段
上下文切换（PCB）
保存user进程数据
恢复kernel寄存器数据

usertrap()
中断处理函数
检查是否为来自用户的系统调用

syscall()
根据系统调用号，查询表单找到对应的系统调用函数

sys_XXX()
执行系统调用函数

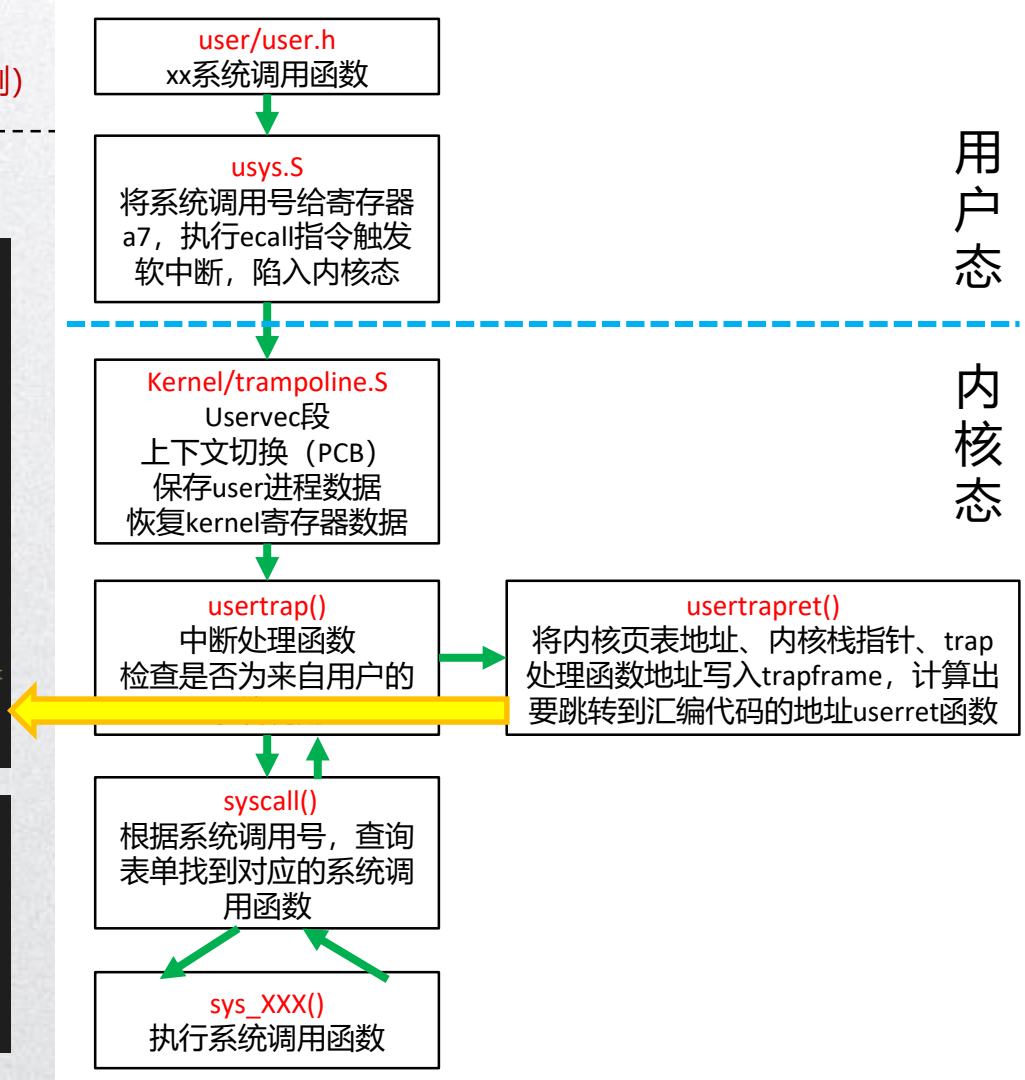
内核态

# 实验原理 | 系统调用（以write为例）

## kernel/trap.c

```
89  void
90  usertrapret(void)
91  {
92    struct proc *p = myproc();
93
94    // we're about to switch the destination of traps from
95    // kerneltrap() to usertrap(), so turn off interrupts until
96    // we're back in user space, where usertrap() is correct.
97    intr_off();
98
99    // send syscalls, interrupts, and exceptions to trampoline.S
100   w_stvec(TRAMPOLINE + (uservec - trampoline));
101
102   // set up trapframe values that uservec will need when
103   // the process next re-enters the kernel.
104   p->trapframe->kernel_satp = r_satp();         // kernel page table
105   p->trapframe->kernel_sp = p->kstack + PGSIZE; // process's kernel stack
106   p->trapframe->kernel_trap = (uint64)usertrap;
107   p->trapframe->kernel_hartid = r_tp();         // hartid for cpuid()
108
```

```
122   uint64 satp = MAKE_SATP(p->pagetable);
123
124   // jump to trampoline.S at the top of memory, which
125   // switches to the user page table, restores user registers,
126   // and switches to user mode with sret.
127   uint64 fn = TRAMPOLINE + (userret - trampoline);
128   ((void (*)(uint64,uint64))fn)(TRAPFRAME, satp);
129 }
```

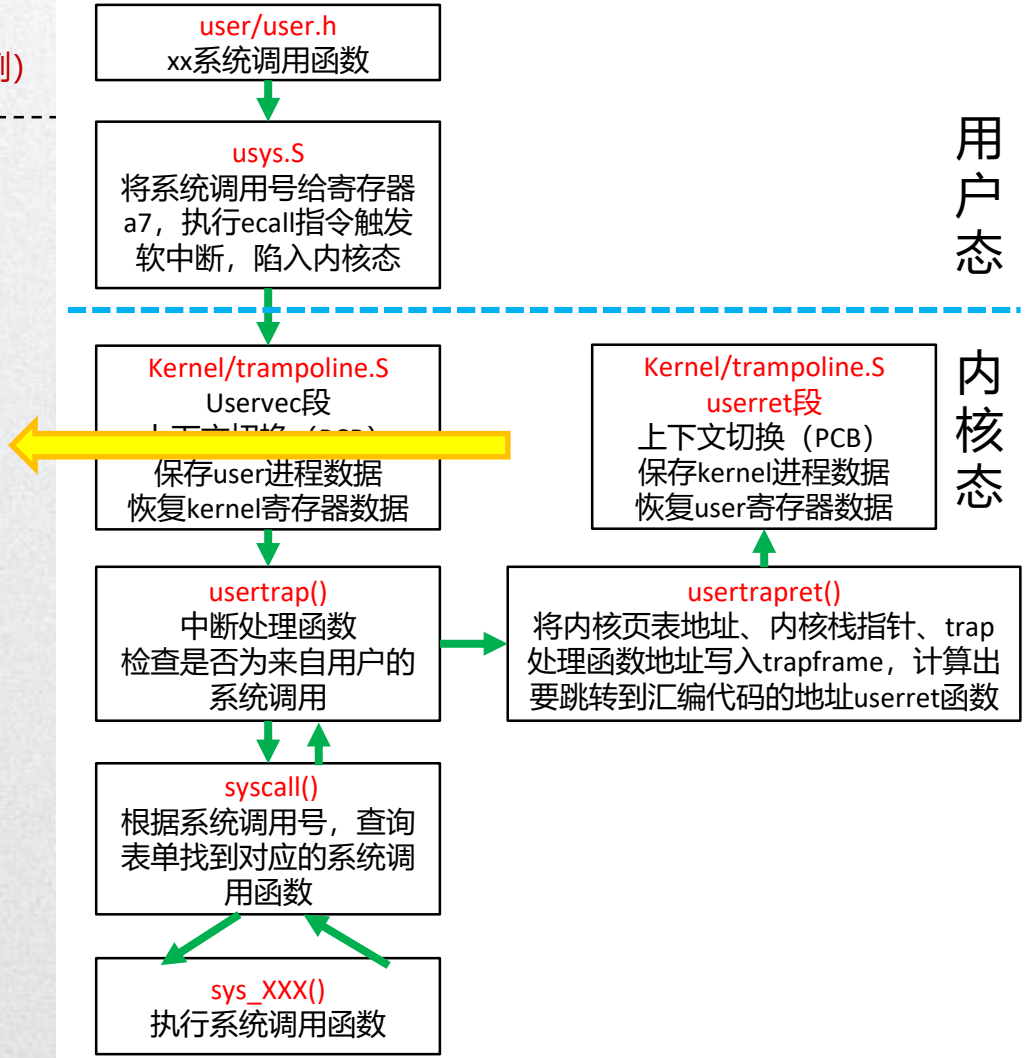


# 实验原理 | 系统调用（以write为例）

## kernel/trampoline.S

```
87  .globl userret
88  userret:
89          # userret(TRAPFRAME, pagetable)
90          # switch from kernel to user.
91          # usertrapret() calls here.
92          # a0: TRAPFRAME, in user page table.
93          # a1: user page table, for satp.
```

```
135
136         # restore user a0, and save TRAPFRAME in sscratch
137         csrrw a0, sscratch, a0
138
139         # return to user mode and user pc.
140         # usertrapret() set up sstatus and sepc.
141         sret
```

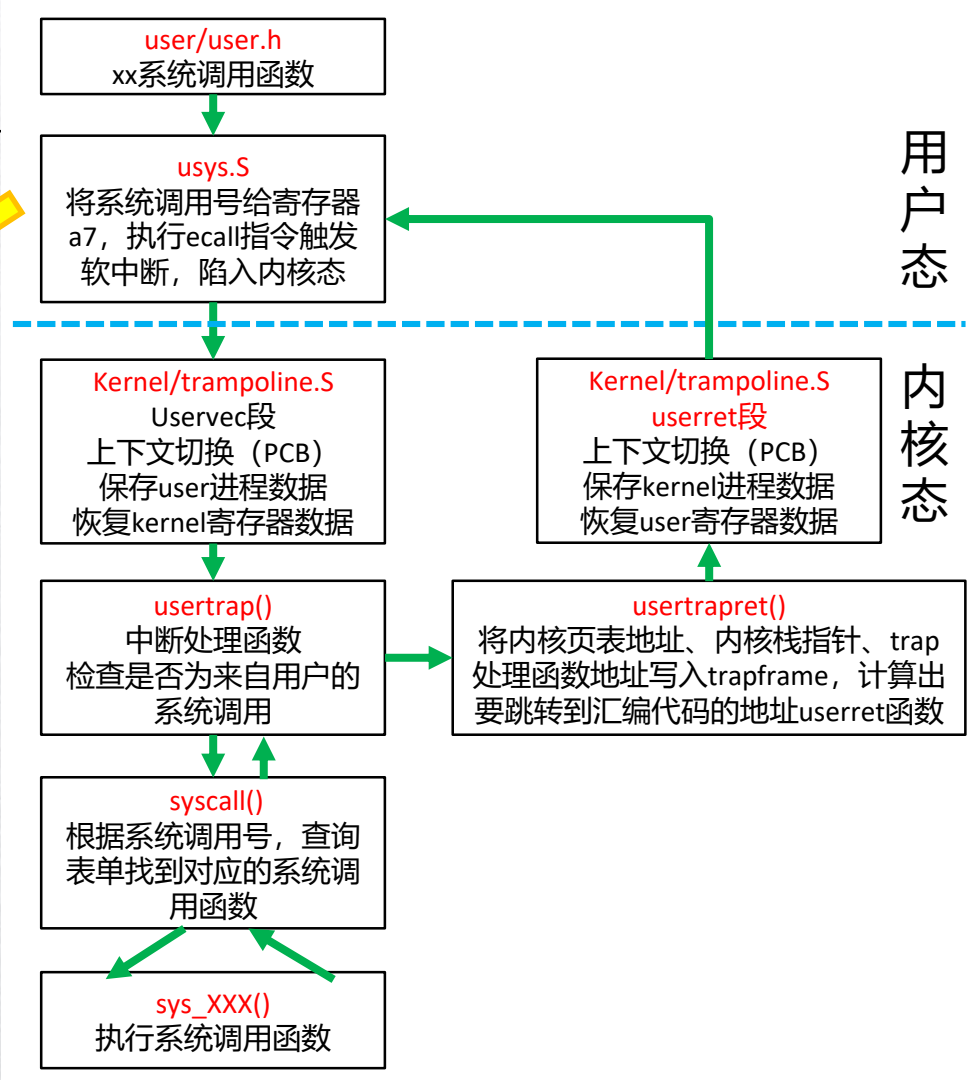


# 实验原理 | 系统调用（以write为例）

user/usys.S

```
28   .global write
29   write:
30    li a7, SYS_write
31    ecall
32    ret
```

执行**ret**就结束了用户看到的系统调用，返回至用户程序。



# 实验实现 | 具体流程

## ① *trace*系统调用 *int trace(int mask)*

- 本实验的**内容及要求与MIT xv6 lab2不一样**，请大家以指导书为准。
  - **Step1**：在做实验前，需切换到 syscall 分支进行实现:
    - 同步上游仓库：https://hitsz-lab.gitee.io/os-labs-2021/tools/#31
- **用户部分：**
  - **Step2**：在Makefile中给 UPROGS 加 $U/_trace
  - **Step3**：在 user/user.h 加入函数声明
  - **Step4**：在 user/usys.pl 加入用户系统调用名称

# 实验实现 | 具体流程

## ① *trace*系统调用 *int trace(int mask)*

- **内核部分：**
  - **Step5**：在 kernel/syscall.h 加入系统调用号SYS_trace
  - **Step6**：在 kernel/sysproc.c 中添加 sys_trace()
  - **Step7**：在 kernel/syscall.c 中加入对应的系统调用分发逻辑
  - **Step8**：开始实现 sys_trace() 对应的逻辑，同时该系统调用还需要修改其他函数的逻辑。

## ① *trace*系统调用 *int trace(int mask)*

- **提示：**
  - 进程启动 trace 后，如果 fork，子进程也应该开启 trace，并且继承父进程的 mask。（这需要注意修改 kernel/proc.c 中 fork() 的代码）
  - 可以在 PCB （struct proc）中添加成员 int mask，这样我们可以记住trace告知进程的 mask 。PCB定义于 kernel/proc.h
  - 为了让每个系统调用都可以输出信息，我们应该在 kernel/syscall.c 中的 syscall() 添加相应逻辑。
  - 其他的系统调用实现可以参考，详见 kernel/sysproc.c。

# 实验实现 | 具体流程

## ② *sysinfo*系统调用 *int sysinfo(struct sysinfo*)*

- **用户部分：**
  - **Step1**：在 Makefile 中给 UPROGS 加 $U/_sysinfotest
  - **Step2**：在 user/user.h 加入函数声明
  - **Step3**：在 user/usys.pl 加入用户系统调用名称

```
1  struct sysinfo {
2    uint64 freemem;   // amount of free memory (bytes)
3    uint64 nproc;     // number of process
4    uint64 freefd;    // number of free file descriptor
5  };
```

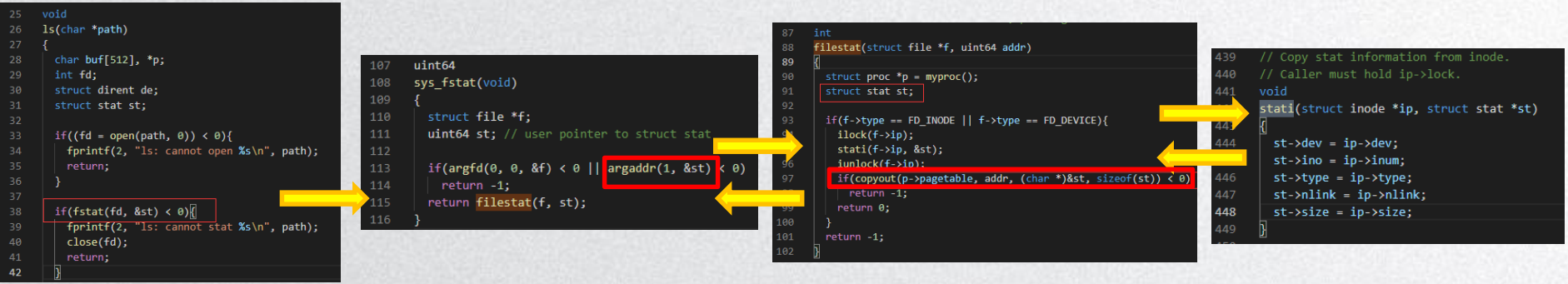
- `freemem`：当前剩余的内存 **字节** 数
- `nproc`：**状态为UNUSED** 的进程个数
- `freefd`：当前进程可用文件描述符的数量，即 **尚未使用** 的文件描述符数量

```
/* 你需要在user/user.h添加如下定义 */
struct sysinfo;  // 需要预先声明结构体，参考fstat的参数stat
int sysinfo(struct sysinfo *);
```

# 实验实现 | 实现提示

## ② *sysinfo*系统调用 *int sysinfo(struct sysinfo*)*

- **提示：**
  - sysinfo 需要在内核地址空间中填写结构体，然后将其复制到用户地址空间。可以参考fstat()(user/ls.c) 、sys_fstat() (kernel/sysfile.c) 、filestat()(kernel/file.c) 中通过 copyout() 函数对该过程的实现。



## ② *sysinfo*系统调用 *int sysinfo(struct sysinfo*)*

➢ **提示：**

➢ 计算剩余的内存空间的函数代码，最好写在文件 kernel/kalloc.c 里。

kmem.freelist是一个保存了当前空闲内存块的链表，因此只需要统计这个链表的长度再乘以PGSIZE就可以得到空闲内存。

```
65  // Allocate one 4096-byte page of physical memory.
66  // Returns a pointer that the kernel can use.
67  // Returns 0 if the memory cannot be allocated.
68  void *
69  kalloc(void)
70  {
71    struct run *r;
72
73    acquire(&kmem.lock);
74    r = kmem.freelist;
75    if(r)
76      kmem.freelist = r->next;
77    release(&kmem.lock);
78
79    if(r)
80      memset((char*)r, 5, PGSIZE); // fill with junk
81    return (void*)r;
82  }
```

```
42  // Free the page of physical memory pointed at by v,
43  // which normally should have been returned by a
44  // call to kalloc().  (The exception is when
45  // initializing the allocator; see kinit above.)
46  void
47  kfree(void *pa)
48  {
49    struct run *r;
50
51    if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
52      panic("kfree");
53
54    // Fill with junk to catch dangling refs.
55    memset(pa, 1, PGSIZE);
56
57    r = (struct run*)pa;
58
59    acquire(&kmem.lock);
60    r->next = kmem.freelist;
61    kmem.freelist = r;
62    release(&kmem.lock);
63  }
```

# 实验实现 | 实现提示

## ② *sysinfo*系统调用 *int sysinfo(struct sysinfo*)*

- **提示：**
  - 计算空闲进程数量的函数代码，最好写在文件 kernel/proc.c 里。

proc->state字段保存了进程的当前状态，有UNUSED、SLEEPING、RUNNABLE、RUNNING、ZOMBIE五种状态

```
668  // Print a process listing to console.  For debugging.
669  // Runs when user types ^P on console.
670  // No lock to avoid wedging a stuck machine further.
671  void
672  procdump(void)
673  {
674    static char *states[] = {
675    [UNUSED]    "unused",
676    [SLEEPING]  "sleep ",
677    [RUNNABLE]  "runble",
678    [RUNNING]   "run   ",
679    [ZOMBIE]    "zombie"
680    };
681    struct proc *p;
682    char *state;
683
684    printf("\n");
685    for(p = proc; p < &proc[NPROC]; p++){
686      if(p->state == UNUSED)
687        continue;
688      if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
689        state = states[p->state];
690      else
691        state = "???";
692      printf("%d %s %s", p->pid, state, p->name);
693      printf("\n");
694    }
695  }
```

## ② *sysinfo*系统调用 *int sysinfo(struct sysinfo*)*

- **提示：**
  - 计算可用文件描述符数量的代码，最好写在文件 kernel/proc.c 里。
  - 文件描述符的值实际上就是 PCB 成员 ofile 的下标。

```
85  // Per-process state
86  struct proc {
87    struct spinlock lock;
88
89    // p->lock must be held when using these:
90    enum procstate state;        // Process state
91    struct proc *parent;         // Parent process
92    void *chan;                  // If non-zero, sleeping on chan
93    int killed;                  // If non-zero, have been killed
94    int xstate;                  // Exit status to be returned to parent's wait
95    int pid;                     // Process ID
96
97    // these are private to the process, so p->lock need not be held.
98    uint64 kstack;               // Virtual address of kernel stack
99    uint64 sz;                   // Size of process memory (bytes)
100   pagetable_t pagetable;       // User page table
101   struct trapframe *trapframe; // data page for trampoline.S
102   struct context context;      // swtch() here to run process
103   struct file *ofile[NOFILE];  // Open files
104   struct inode *cwd;           // Current directory
105   char name[16];               // Process name (debugging)
106 };
```

## ② *sysinfo*系统调用 *int sysinfo(struct sysinfo*)*

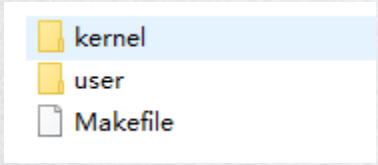
- **提示：**
  - 如何获取系统调用的参数?
    - argint 获取整数，参数n代表定位第n个参数
    - argptr 获取指针
    - argstr 获取字符串起始地址

## ② *sysinfo*系统调用 *int sysinfo(struct sysinfo*)*

- **提示：**
  - 在添加上述三个函数后，可以在 kernel/defs.h 中声明，以便在其他文件中调用这些函数。
  - 查阅《xv6 book》 chapter1 和 chapter2 中相关的内容。

# 实验提交

- 请务必注意查看作业提交平台，按时提交代码及报告
  - 实验二提交截止时间：下周二
  - 提交要求：
    - kernel文件夹，包括kernel目录下所有修改后的文件
    - user文件夹，包括user目录下所有修改后的文件
    - Makefile文件
    - 实验设计报告



# THANKS

同学们，
请开始实验吧！