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*用户测试程序**：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

user > C trace.c > main(int, char * [])

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int
main(int argc, char *argv[])
{
  int i;
  char *nargv[MAXARG];

  if(argc < 3 || (argv[1][0] < '0' || argv[1][0] > '9')){
    fprintf(2, "Usage: %s mask command\n", argv[0]);
    exit(1);
  }

  if (trace(atoi(argv[1])) < 0) {
    fprintf(2, "%s: trace failed\n", argv[0]);
    exit(1);
  }

  for(i = 2; i < argc && i < MAXARG; i++){
    nargv[i-2] = argv[i];
  }
  exec(nargv[0], nargv);
  exit(0);
}
```

user/trace.c
用户测试程序

执行trace系统调用

执行exec系统调用

实验原理 | 任务1：trace系统调用

① *trace*系统调用 *int trace(int mask)*

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
$
```

1. 在 **第一个例子** 中，`trace 32 grep hello README`，其中，trace表示我们希望执行用户态应用程序trace（见user/trace.c），后面则是trace应用程序附带的入参：
   - `32` 是"1 << SYS_read"，表示只追踪系统调用read；
   - `grep` 是trace应用程序中通过"exec"启动的另一个程序（见 user/grep.c）；
   - `hello README` 则是grep程序的入参；
   - 该命令的作用是使用grep程序查找README文件中匹配"hello"的行，并将其所使用到的read系统调用的信息打印出来，打印的格式为：`PID: sys_read(read系统调用的arg0) -> read系统调用的return_value`。

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
$
```

2. 在 **第二个例子** 中，trace也是启动了`grep`程序，同时追踪所有的系统调用其中`2147583647`是`31`位bit全置一的十进制整型。可以看出，打的第一信息就是系统调用trace，其第一个参数即命令行中输入2147583647。

实验原理 | 任务1：trace系统调用

① *trace*系统调用 *int trace(int mask)*

例3

```
/* 例子3，手动输入:grep hello README */

$ grep hello README
$
```

3. 在 **第三个例子** 中，启动了 `grep` 程序，但是没有使用trace，所以什么trace都不会出现。

例4

```
$ 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
...
$
```

4. 在 **第四个例子** 中，trace启动了 `usertests` 程序中 `forkforkfork` （见user/usertests.c），跟踪系统调用了fork，每次fork后代都会打印对的进程id。

   - 该例中的fork实际上并没有参数，方便起见，你可以直接打印用于传该参数的寄存器的值，它可能是任意值。
   - forkforkfork 会一直不停的fork子进程，直到进程数超过 `NPROC`，其定义见kernel/param.h。
   - usertests是实验提供的用于测试xv6的系统调用，详见user/usertests.c。

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  };
```

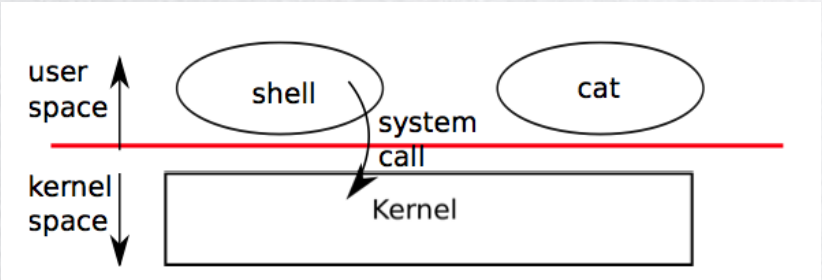
当完成上述的两个实验后，在命令行输入make grade进行测试。

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

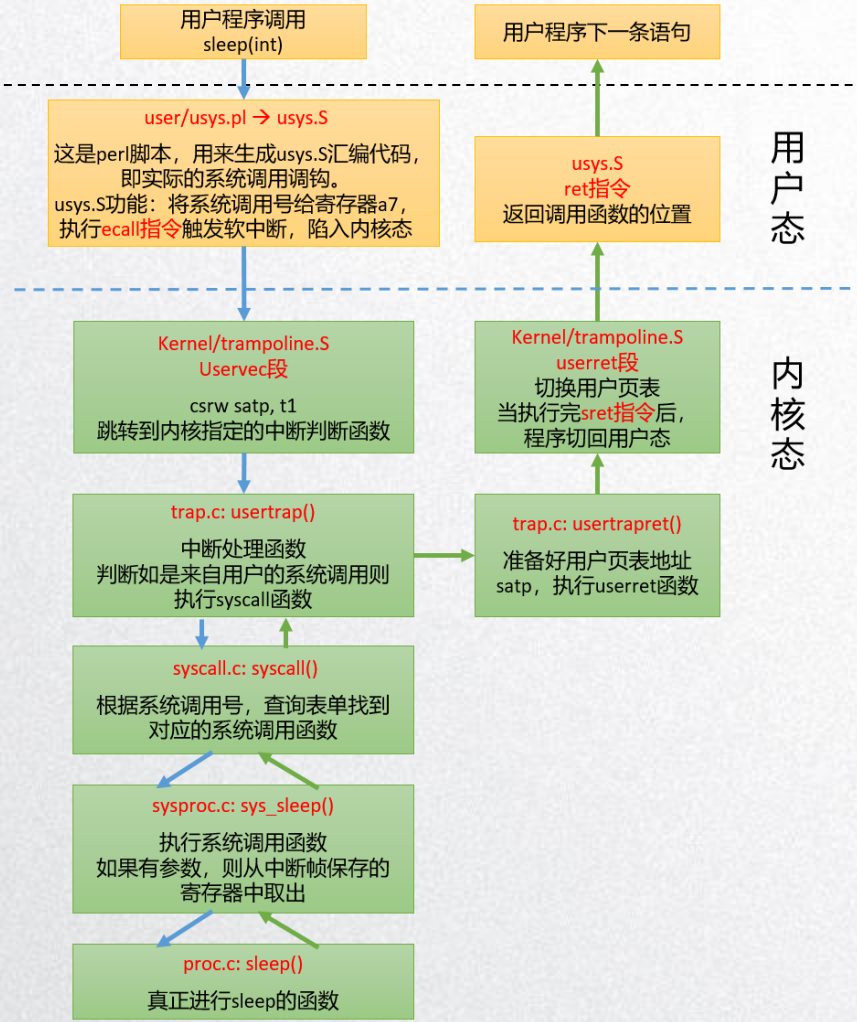


实验原理 | 系统调用

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。

1. 在做实验前，需保存上一个实验的更改。

- 使用git命令完成commit

3.3.1.1 Commit

```
$ git add 你要保存更改的文件们
# 或者，git add --all 以添加所有更改。
$ git commit -m "本次Commit你做了什么"
```

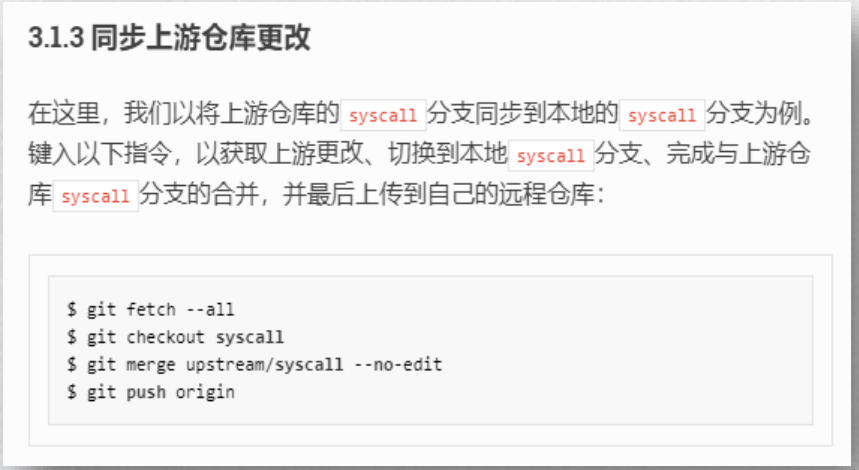
建议每次Commit都使用有效的Commit信息，以便于自己追踪代码进度。

- 或使用vscode图形化界面完成commit

3.3.2 使用VS Code内建的图形化界面完成操作

VS Code的版本控制界面在左侧从上往下数第二个按钮处；下方会显示当前所处的分支，点击可以切换当前所处分支；右侧的圆形双箭头则代表与远程环境同步。右侧Changes显示工作区中被修改的文件，而Staged则显示将被加入到下次Commit中的修改。如图，这个界面显示我们有两个被修改的文件：

2. 切换到 syscall 分支进行实现并同步上游仓库： https://hitsz-lab.gitee.io/os-labs-2021/tools/#31



3.1.3 同步上游仓库更改

在这里，我们以将上游仓库的 `syscall` 分支同步到本地的 `syscall` 分支为例。键入以下指令，以获取上游更改、切换到本地 `syscall` 分支、完成与上游仓库 `syscall` 分支的合并，并最后上传到自己的远程仓库：

```
$ git fetch --all
$ git checkout syscall
$ git merge upstream/syscall --no-edit
$ git push origin
```

实验实现 | 具体流程

① *trace*系统调用 *int trace(int mask)*

- 记得在Makefile中给 `UPROGS` 加 `$U/_trace` 。
- 然后 `make qemu` 会发现无法编译 user/trace.c，这是因为我们没在用户态包装好 `trace()` ，因此我们需要按照 "实验原理" 中所述，慢慢添加一个系统调用的接口。
  - 在 `user/user.h` 加入函数定义；
  - 在 `user/usys.pl` 加入用户系统调用名称（也就是之前所说的entry）；
  - 在 `kernel/syscall.h` 加入系统调用号SYS_trace，以给trace做一个标识，该调用号的取值可自行决定。
- 然后启动xv6，在shell输入 `trace 32 grep hello README` ，会发xv6崩溃了，因为这条系统调用没有在内核中实现。

实验实现 | 具体流程

① *trace*系统调用 *int trace(int mask)*

- **内核部分：**
  - **Step1**：在 kernel/syscall.h 加入系统调用号 SYS_trace
  - **Step2**：在 kernel/sysproc.c 中添加 sys_trace()
  - **Step3**：在 kernel/syscall.c 中加入printf打印信息
  - **Step4**：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。

```
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,
};

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

  num = p->trapframe->a7;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    p->trapframe->a0 = syscalls[num]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

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：**

➢ 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 };
```

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。

```
25  void
26  ls(char *path)
27  {
28    char buf[512], *p;
29    int fd;
30    struct dirent de;
31    struct stat st;
32
33    if((fd = open(path, 0)) < 0){
34      fprintf(2, "ls: cannot open %s\n", path);
35      return;
36    }
37
38    if(fstat(fd, &st) < 0){
39      fprintf(2, "ls: cannot stat %s\n", path);
40      close(fd);
41      return;
42    }
```

```
107  uint64
108  sys_fstat(void)
109  {
110    struct file *f;
111    uint64 st; // user pointer to struct stat
112
113    if(argfd(0, 0, &f) < 0 || argaddr(1, &st) < 0)
114      return -1;
115    return filestat(f, st);
116  }
```

```
87  int
88  filestat(struct file *f, uint64 addr)
89  {
90    struct proc *p = myproc();
91    struct stat st;
92
93    if(f->type == FD_INODE || f->type == FD_DEVICE){
94      ilock(f->ip);
95      stati(f->ip, &st);
96      iunlock(f->ip);
97      if(copyout(p->pagetable, addr, (char *)&st, sizeof(st)) < 0)
98        return -1;
99      return 0;
100   }
101   return -1;
102 }
```

```
439  // Copy stat information from inode.
440  // Caller must hold ip->lock.
441  void
442  stati(struct inode *ip, struct stat *st)
443  {
444    st->dev = ip->dev;
445    st->ino = ip->inum;
446    st->type = ip->type;
447    st->nlink = ip->nlink;
448    st->size = ip->size;
449  }
```

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。

GDB调试演示视频

【1. VSCode调试xv6内核代码】

https://www.bilibili.com/video/BV1ZB4y1E7X5?share_source=copy_web&vd_source=a822dcda3537564ccdd0bb45aa0afe33

【2. VSCode调试xv6用户代码】

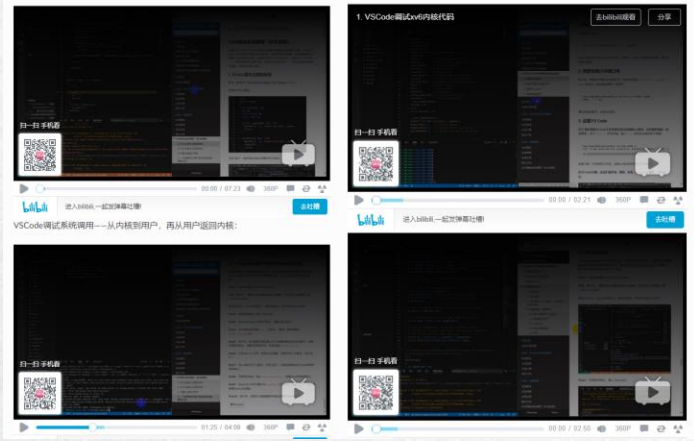
https://www.bilibili.com/video/BV1i14y1Y7ZZ?share_source=copy_web&vd_source=a822dcda3537564ccdd0bb45aa0afe33

【3. VSCode调试系统调用过程（包含pagetable和汇编）】

https://www.bilibili.com/video/BV12P411J7xq?share_source=copy_web&vd_source=a822dcda3537564ccdd0bb45aa0afe33

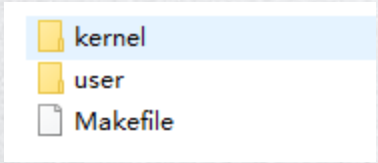
【4. VSCode调试系统调用——从内核到用户，再从用户返回内核】

https://www.bilibili.com/video/BV1ug411m7ir?share_source=copy_web&vd_source=a822dcda3537564ccdd0bb45aa0afe33vd_source=a822dcda3537564ccdd0bb45aa0afe33



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：下一次实验前
  - 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！