Lab2: Fun with system calls

Sajad Meisami

Topics for today

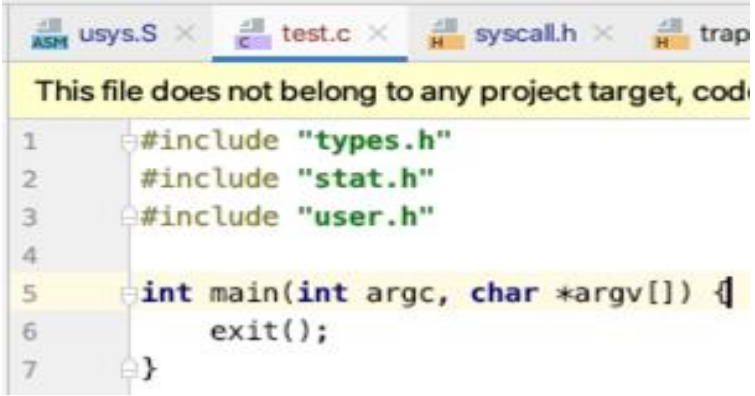
- How to add a functional system call?
- Assignments & hints:
  - Change the exit syscall signature to void exit(int status);
  - Update the wait syscall to int wait(int *status);
  - Add a waitpid syscall: int waitpid(int pid, int *status, int options);
  - Write an example program to illustrate your waitpid works.

[1]: XV6 book: https://pdos.csail.mit.edu/6.828/2014/xv6/book-rev8.pdf
[2]: XV6 syscall explained:
https://medium.com/@flag_seeker/xv6-system-calls-how-it-works-c541408f21ff

How does syscall exit() work?

Step 1: call from user space

- **test.c**
  - user program to make a syscall
- **user.h**
  - declaration of syscall in user level
- **usys.S**
  - macro definition which in effect as a function
  - move immediate value of *SYS_exit* (defined in syscall.h) into register *%eax*;
  - issue interrupt 64 (reserved for all system calls, see traps.h)
  - return




```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
```

```
#define SYSCALL(getpid) \
  .globl getpid; \
  getpid: \
    movl $SYS_getpid, %eax; \
    int $T_SYSCALL; \
    ret
```

How does syscall exit() work?

Step2: transfer to kernel mode

- raise privilege level of CPU to kernel mode
- transfer control to **trap vectors** (initialized in tvinit())
- setup trapframe (see definition in x86.h)
  - **vector.S**
    - push 0 and 64 to stack
    - call alltraps() in trapasm.S
  - **trapasm.S**
    - alltraps() finish trapframe
    - call trap() in trap.c
- trap() function
  - set trapframe and call syscall() defined in syscall.c

vectors.S | usys.S | tes

```
316   jmp alltraps
317 .globl vector64
318 vector64:
319   pushl $0
320   pushl $64
321   jmp alltraps
```

trap.c | vectors.S | usys.S | test.c

```
36  void
37  trap(struct trapframe *tf)
38  {
39    if(tf->trapno == T_SYSCALL){
40      if(myproc()->killed)
41        exit();
42      myproc()->tf = tf;
43      syscall();
44      if(myproc()->killed)
45        exit();
46      return;
47    }
```

How does syscall exit() work?

```
108     static int (*syscalls[])(void) = {
109     [SYS_fork]    sys_fork,
110     [SYS_exit]    sys_exit,
111     [SYS_wait]    sys_wait,
112     [SYS_pipe]    sys_pipe,
113     [SYS_read]    sys_read,
```

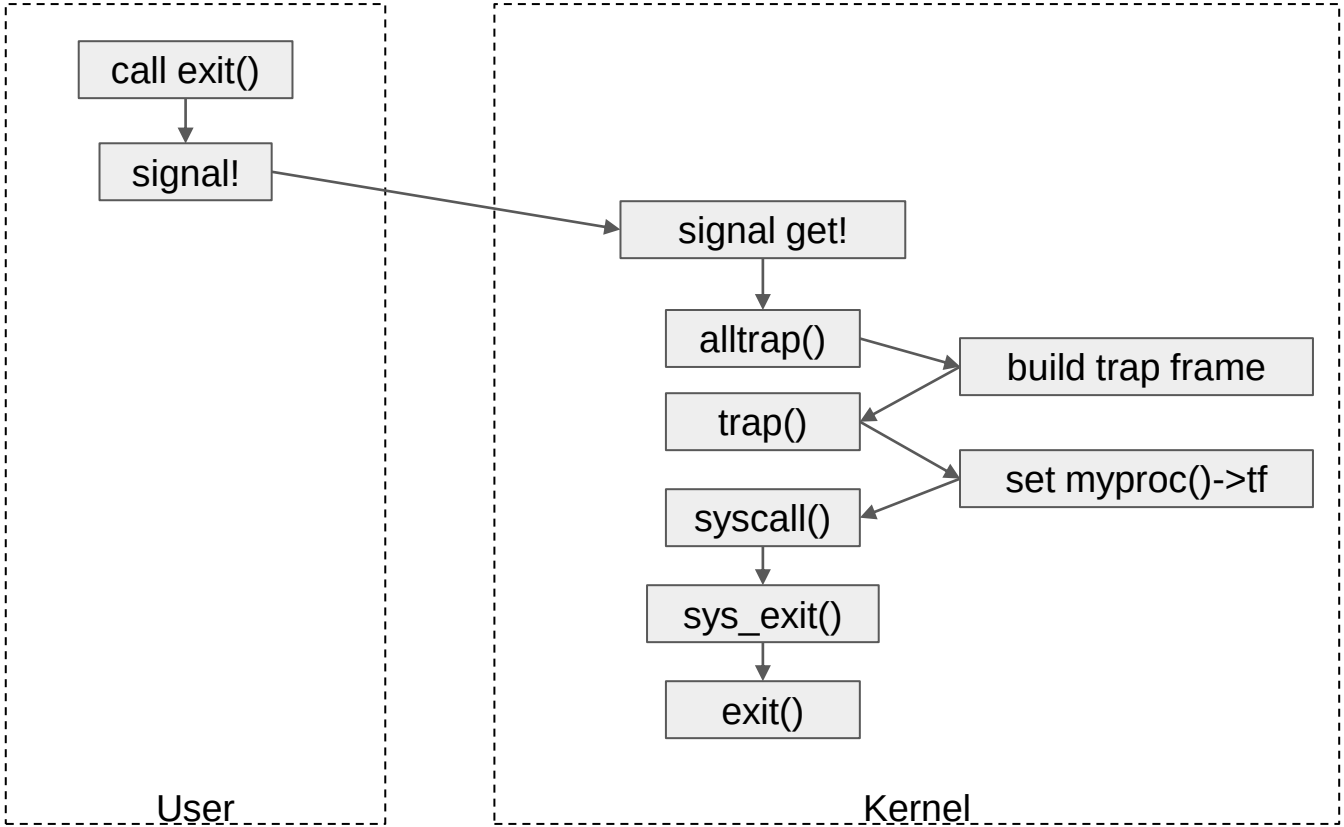
Step2: transfer to kernel mode

- **syscall() in syscall.c**
  - lookup array of function pointers;
    - int *function*(void);
  - call the sys_exit function and put return value in eax register.
- **sys_exit()**
  - implemented in sysproc.c
  - helper function exit():
    - defined in defs.h
    - implemented in proc.c

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

  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

How exit system call works



Step-by-step instructions

Step 1: user application

- Makefile

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _test\
```

- test.c

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

int main(int argc, char *argv[]) {
    //printf(1, "hello world\n");
    hello();
    exit();
}
```

- user.h

```c
char* sbrk(int);
int sleep(int);
int uptime(void);
int hello(void);
```

Step 2

- usys.S

```
28  SYSCALL(getpid)
29  SYSCALL(sbrk)
30  SYSCALL(sleep)
31  SYSCALL(uptime)
32  SYSCALL(hello)
33
```

- syscall.h

```
20  #define SYS_link   19
21  #define SYS_mkdir  20
22  #define SYS_close  21
23  #define SYS_hello  22
```

Step 3

- syscall.c

```
102  extern int sys_unlink(void);
103  extern int sys_wait(void);
104  extern int sys_write(void);
105  extern int sys_uptime(void);
106  extern int sys_hello(void);
```

```
126  [SYS_unlink]   sys_unlink,
127  [SYS_link]     sys_link,
128  [SYS_mkdir]    sys_mkdir,
129  [SYS_close]    sys_close,
130  [SYS_hello]    sys_hello,
131  };
```

Step 4

- sysproc.c

```
93  int
94  sys_hello(void) {
95      hello();
96      return 0;
97  }
98
```

- proc.c

```
536
537 void
538 hello(void) {
539     cprintf("\n\n Hello from your kernel space! \n\n");
540 }
541
```

- defs.h

```
118 void            sleep(void*, struct spinlock*);
119 void            userinit(void);
120 int             wait(void);
121 void            wakeup(void*);
122 void            yield(void);
123 void            hello(void);
```

Execute & check the result

$ make qemu-nox

(... xv6 boots ...)

$ test

```
$ test


 Hello from your kernel space!
```

Assignment a) change exit() signature to *void exit(int status)*

The exit system call must act as previously defined (i.e., terminate the current process) but it must also store the exit status of the terminated process in the corresponding structure.

- Modify proc struct (proc.h) to include a new field that saves an exit status for a terminated process. (e.g. int exitStatus;)
- You can either modify existing exit() system call in place or define a new system call; note that if you make modifications in place, make sure you modify all locations of exit() call in the codebase;
- Modify all relevant files correspondingly;

Assignment b) update wait() to *int wait(int *status)*

The wait system call must prevent the current process from execution until any of its child processes is terminated (if any exists) and return the terminated child exit status through the status argument.

- The goal is to get familiar with how to return a value from kernel space to user space;
- Return the terminated child proc’s exit status through the status pointer argument;
  - understand the current wait() system call, in terms of how to traverse the ptable to look up for child proc;
  - understand the proc structure;

Assignment c) add a waitpid() system call

This system call must act like wait system call with the following additional properties: The system call must wait for a process (not necessary a child process) with a pid that equals to one provided by the pid argument.

- Hint: This will be a modified version of the original wait() system call. You will need to traverse the ptable to find the proc with pid matches the given pid argument.

Assignment d) write a testing program

Write an example program to illustrate that your waitpid works. You have to modify the Makefile to add your testing program so that it can be executed from inside the shell once xv6 boots.

- In your report, show the printout of your testing program and illustrate why this printout proves the correctness of your implementation.

To help finish lab 2 tasks:

- check out existing syscalls with parameters and return value;
- chapter 0-3 of xv6 book (https://pdos.csail.mit.edu/6.828/2014/xv6/book-rev8.pdf)