Part 1: trace close system call

Overview

There are three cases when control must be transferred from a user program to the kernel. Here we use `close` system call as an example to walk through an system call path in code after being called from a program to get a deep understand of control mechanism of the operating system.

User mode

1. Firstly, we could know xv6 provide what kind of system calls for the user from `user.h` and find the system call close() statement.

```
// system calls
int close(int);
...
```

2. From `traps.h`, we could find different kinds of traps. At this time, we just care about system call.

```
// x86 trap and interrupt constants.
// processor defined exceptions or interrupt vectors.
#define T_SYSCALL       64      // system call
```

3. Every time when the user program calls the system call, it will enter to `usys.S`, which is a macro.

```
#include "syscall.h"
#include "traps.h"
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
SYSCALL(close)
```

For the close() system call, it will be executed as the following:

```
.globl close;
close:
    movl $21, %eax;
    int 0x40;
    ret
```

The process pushed the arguments for an close call on the process's stack, and put the system call number in `%eax`. The system call numbers match the entries in the `syscalls` array, a table of function pointers. The `int` instruction switches the processor from user mode to kernel mode.

Kernel mode

When calling system call trap into OS, the processor enters an kernel mode. At first, we analysis details for `int` instruction; then explain what trap handler do, finally, we describe how kernel find the system call arguments which input from the user mode.

1. Details for `int` instruction

For the `int` instruction, the operation system should do at least three things:

a). switch stack: changing from user stack to kernel stack, it gets the base address of kernel stack from `ss0:esp0`;

b). push related registers to stack `ss, esp, eflag, cs, eip`;

c). jump to the trap handler

2. Trap handler

As we see, the last step is to jump to the trap handler. In this case, it will jump to `vector64`

```
# generated by vectors.pl - do not edit
# handlers
.globl alltraps
vector64:
  pushl $0
  pushl $64
  jmp alltraps
```

`vector64` push the `errono = 0` and `trapno = T_SYSCALL` into stack.

Then the program jump to `alltraps` to continue the next step.

```
#include "mmu.h"
# vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal
  # Set up data segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es
```

From this block, we could see `alltraps` continues to save processor registers. The result of this operation is that the kernel stack now contains a `struct trapframe` containing the processor registers at the time of the trap.

```
# Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp // make the esp points to trapframe
```

Once the segments are set properly, `alltraps` can call the C trap handler trap. It `pushl %esp`, which points at the trap frame it just constructed, onto the stack as an argument to trap.

`trap()` : check `tf->trapno == T_SYSCALL` then execute related logical code.

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
  ...
}
```

`syscall()`

For the system call, trap invokes `syscall`, which loads the system call number from the trap frame, and indexes into the system call tables. In this case, `%eax` contains the value `SYS_close`, and `syscall` will invoke the `SYS_close` entry of the system call table, which corresponds to invoking `sys_close`.

```
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](); // system call result
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

From this code, we can see `syscall` records the return value of the system call function in `%eax`. When the trap returns to user space, it will load the values from `cp->tf` into the machine registers.

`argfd()`

The helper function `argfd` retrieve the close system call argument. `argfd` uses `argint` t0 retrieve a file descriptor number, checks if it is valid file descriptor, and returns the corresponding `struct file`

```
// Fetch the nth word-sized system call argument as a file descriptor
// and return both the descriptor and the corresponding struct file.
```

```
static int
argfd(int n, int *pfd, struct file **pf)
{
  int fd;
  struct file *f;
  if(argint(n, &fd) < 0)
    return -1;
  if(fd < 0 || fd >= NOFILE || (f=myproc()->ofile[fd]) == 0)
    return -1;
  if(pfd)
    *pfd = fd;
  if(pf)
    *pf = f;
  return 0;
}
```

```
// Fetch the nth 32-bit system call argument.
int
argint(int n, int *ip)
{
  return fetchint((myproc()->tf->esp) + 4 + 4*n, ip); //%esp+4+4*n
}
```

So the close system call decodes the argument using `argint, argptr and argstr` and then call the really implementations. At here, the `argfd` checks it is an invalid file descriptor, it return -1. Right now, `syscall` prints an error and returns -1.

When the program executes here, xv6 prints error message and set `proc-killed` to remember to clean up the user process.