2

Under The Hood: The System Call

In this note, we'll peak under the hood of one simple and neat OS called **xv6** [CK+08]. The xv6 kernel is a port of an old UNIX version 6 from PDP-11 (the machine it was originally built to run on) to a modern system, Intel x86, and the idea to make it a source of study stems from Lions' famous commentary on Unix SV6 [L76]. The xv6 kernel does is compact and concise, and thus represents a great way to understand many of the fundamentals that underly operating systems without the code deluge that often accompanies such a pursuit.

ASIDE: UNDER THE HOOD

In these "under the hood" sections, we'll be looking at what goes on inside an operating system. The actual code and all of that jazz. Hopefully it will help deconstruct the mythic beast that is the OS and transform it in your mind into what it actually is: just software.

We'll specifically trace what happens in the code in order to understand a **system call**. System calls allow the operating system to run code on the behalf of user requests but in a protected manner, both by jumping into the kernel (in a very specific and restricted way) and also by simultaneously raising the privilege

1

level of the hardware, so that the OS can perform certain restricted operations.

2.1 System Call Overview

Before delving into the details, we first provide an overview of the entire process. The problem we are trying to solve is simple: how can we build a system such that the OS is allowed access to all of the resources of the machine (including access to special instructions, to physical memory, and to any devices) while user programs are only able to do so in a restricted manner?

g on a machine, with access to all resources. A typical user program, in contrast, should have reduced

The way we achieve this goal is with hardware support. The hardware must explicitly have a notion of privilege built into it, and thus be able to distinguish when the OS is running versus typical user applications.

XXX – not done – XXX – should do as more general discussion of CONCEPTS behind a system call?

2.2 Getting Into The Kernel: A Trap

The first step in a system call begins at user-level with an application. The application that wishes to make a system call (such as `read()`) calls the relevant library routine. However, all the library version of the system call does is to place the proper arguments in relevant registers and issue some kind of **trap** instruction, as we see in an expanded version of `usys.S` [1] (Figure 2.2).

---

[1]Some macros are used to define these functions so as to make life easier for the kernel developer; the example shows the macro expanded to the actual assembly code
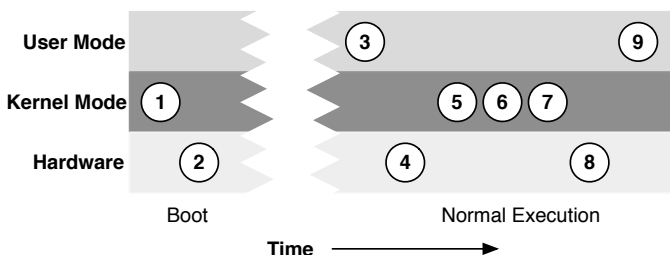
Figure 2.1: System Call Overview

```
.globl read;
read:
  movl $6, %eax;
  int $48;
  ret
```

Figure 2.2: File: usys.S

Here we can see that the read() library function actually doesn't do much at all; it moves the value 5 into the register %eax and issues the x86 trap instruction which is confusingly called **int** (short for "interrupt"[2]). The value in %eax is going to be used by the kernel to **vector** to the right system call, i.e., it determines which system call is being invoked. The int instruction takes one argument (here it is 48), which tells the hardware which trap type this is. In xv6, trap 48 is used to handle system calls. Any other arguments which are passed to the system call are passed on the stack.

---

[2]The x86 architecture has been called an architecture only a mother could love by Dave Patterson, a well-known computer architect. Among other things Intel got wrong, the names they use for well-known concepts tend to be slightly strange. Using "int" for what is a trap is just one such example.

2.3   Kernel Side: Trap Tables

Once the `int` instruction is executed, the hardware takes over and does a bunch of work on behalf of the caller. One important thing the hardware does is to raise the **privilege level** of the CPU to kernel mode; on x86 this is usually means moving from a **CPL (Current Privilege Level)** of 3 (the level at which user applications run) to CPL 0 (in which the kernel runs). Yes, there are a couple of in-between privilege levels, but most systems do not make use of these.

The second important thing the hardware does is to transfer control to the **trap vectors** of the system. To enable the hardware to know what code to run when a particular trap occurs, the OS, when booting, must make sure to inform the hardware of the location of the code to run when such traps take place. This is done in `main.c` as follows:

```
// FILE: main.c
int
main(void)
{
   ...
   tvinit(); // trap vectors initialized here
   ...
}
```

The routine `tvinit()` is the relevant one here. Peeking inside of it, we see:

```
// FILE: trap.c
void tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);

  // this is the line we care about...
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
}
```

The SETGATE() macro is the relevant code here. It is used to set the idt array to point to the proper code to execute when various traps and interrupts occur. For system calls, the single SETGATE() call (which comes after the loop) is the one we're interested in. Here is what the macro does (as well as the gate descriptor it sets):

```
// FILE: mmu.h
// Gate descriptors for interrupts and traps
struct gatedesc {
  uint off_15_0 : 16;   // low 16 bits of offset in segment
  uint cs : 16;         // code segment selector
  uint args : 5;        // # args, 0 for interrupt/trap gates
  uint rsv1 : 3;        // reserved(should be zero I guess)
  uint type : 4;        // type(STS_{TG,IG32,TG32})
  uint s : 1;           // must be 0 (system)
  uint dpl : 2;         // descriptor(meaning new) privilege level
  uint p : 1;           // Present
  uint off_31_16 : 16;  // high bits of offset in segment
};

// Set up a normal interrupt/trap gate descriptor.
// - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
//   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
// - sel: Code segment selector for interrupt/trap handler
// - off: Offset in code segment for interrupt/trap handler
// - dpl: Descriptor Privilege Level -
//        the privilege level required for software to invoke
//        this interrupt/trap gate explicitly using an int instruction.
#define SETGATE(gate, istrap, sel, off, d)               \
{                                                        \
  (gate).off_15_0 = (uint) (off) & 0xffff;               \
  (gate).cs = (sel);                                     \
  (gate).args = 0;                                       \
  (gate).rsv1 = 0;                                       \
  (gate).type = (istrap) ? STS_TG32 : STS_IG32;          \
  (gate).s = 0;                                          \
  (gate).dpl = (d);                                      \
  (gate).p = 1;                                          \
  (gate).off_31_16 = (uint) (off) >> 16;                 \
}
```

As you can see from the code, all the SETGATE() macros does is set the values of an in-memory data structure. Most important is the off parameter, which tells the hardware where

the trap handling code is. In the initialization code, the value
`vectors[T_SYSCALL]` is passed in; thus, whatever the `vectors`
array points to will be the code to run when a system call takes
place. There are other details (which are important too); consult
an x86 hardware architecture manual [I09] for more informa-
tion.

Note, however, that we still have not informed the hardware
of this information, but rather filled a data structure. The actual
hardware informing occurs a little later in the boot sequence; in
xv6, it happens in the routine `mpmain()` in the file `main.c`:

```
static void
mpmain(void)
{
  idtinit();
  ...

void
idtinit(void)
{
  lidt(idt, sizeof(idt));
}

static inline void
lidt(struct gatedesc *p, int size)
{
  volatile ushort pd[3];

  pd[0] = size-1;
  pd[1] = (uint)p;
  pd[2] = (uint)p >> 16;

  asm volatile("lidt (%0)" : : "r" (pd));
}
```

Here, you can see how (eventually) a single assembly in-
struction is called to tell the hardware where to find the **inter-
rupt descriptor table (IDT)** in memory. Note this is done in
`mpmain()` as each processor in the system must have such a
table (they all use the same one of course). Finally, after execut-
ing this instruction (which is only possible when the kernel is

running, in privileged mode), we are ready to think about what
happens when a user application invokes a system call.

```
struct trapframe {
  // registers as pushed by pusha
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;        // useless & ignored
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;

  // rest of trap frame
  ushort es;
  ushort padding1;
  ushort ds;
  ushort padding2;
  uint trapno;

  // below here defined by x86 hardware
  uint err;
  uint eip;
  ushort cs;
  ushort padding3;
  uint eflags;

  // below here only when crossing rings, such as from user to kernel
  uint esp;
  ushort ss;
  ushort padding4;
};
```

Figure 2.3: File: `x86.h`

2.4 From Low-level To The C Trap Handler

The OS has carefully set up its trap handlers, and thus we are
ready to see what happens on the OS side once an application
issues a system call via the `int` instruction. Before any code is
run, the hardware must perform a number of tasks. The first

thing it does are those tasks which are difficult/impossible for the software to do itself, including saving the current PC (IP or EIP in Intel terminology) onto the stack, as well as a number of other registers such as the `eflags` register (which contains the current status of the CPU while the program was running), stack pointer, and so forth. One can see what the hardware is expected to save by looking at the `trapframe` structure as defined in `x86.h` (Figure 2.3).

As you can see from the bottom of the trapframe structure, some pieces of the trap frame are filled in by the hardware (up to the `err` field); the rest will be saved by the OS. The first code OS that is run is `vector48()` (Figure 2.4) as found in `vectors.S` (which is automatically generated by `vectors.pl`).

```
.globl vector48
vector48:
  pushl $48
  jmp alltraps
```

Figure 2.4: File: `vectors.S` (generated by `vectors.pl`)

This code pushes the trap number onto the stack (filling in the `trapno` field of the trap frame) and then calls `alltraps()` to do most of the saving of context into the trap frame (Figure 2.5).

The code in `alltraps()` pushes a few more segment registers (not described here, yet) onto the stack before pushing the remaining general purpose registers onto the trap frame via a `pushal` instruction. Then, the OS changes the descriptor segment and extra segment registers so that it can access its own (kernel) memory. Finally, the C trap handler is called.

2.5 The C Trap Handler

Once done with the low-level details of setting up the trap frame, the low-level assembly code calls up into a generic C trap handler called `trap()`, which is passed a pointer to the trap frame. This trap handler is called upon all types of inter-

```
  # vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushal

  # Set up data segments.
  movl $SEG_KDATA_SEL, %eax
  movw %ax,%ds
  movw %ax,%es

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp
```

Figure 2.5: File: `trapasm.S`

rupts and traps, and thus check the trap number field of the trap frame (`trapno`) to determine what to do. ==The first check is for the system call trap number (`T_SYSCALL`, or 48 as defined somewhat arbitrarily in `traps.h`), which then handles the system call,== as you see here:

```
// FILE: trap.c
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(cp->killed)
      exit();
    cp->tf = tf;
    syscall();
    if(cp->killed)
      exit();
    return;
  }
  ... // continues
}
```

The code isn't too complicated. It checks if the current pro-

cess (that made the system call) has been killed; if so, it simply exits and cleans up the process (and thus does not proceed with the system call). It then calls `syscall()` to actually perform the system call; more details on that below. Finally, it checks whether the process has been killed again before returning. Note that we'll follow the return path below in more detail.

2.6 Vectoring To The System Call

Once we finally get to the `syscall()` routine in `syscall.c`, not much work is left to do (Figure 2.6). The system call number has been passed to us in the register `%eax`, and now we unpack that number from the trap frame and use it to call the appropriate routine as defined in the system call table `syscalls[]`. Pretty much all operating systems have a table similar to this to define the various system calls they support. After carefully checking that the system call number is in bounds, the pointed-to routine is called to handle the call. For example, if the system call `read()` was called by the user, the routine `sys_read()` will be invoked here. The return value, you might note, is stored in `%eax` to pass back to the user.

2.7 The Return Path

The return path is pretty easy. First, the system call returns an integer value, which the code in `syscall()` grabs and places into the `eax` field of the trap frame. The code then returns into `trap()`, which simply returns into where it was called from in the assembly trap handler (Figure 2.7).

This return code doesn't do too much, just making sure to pop the relevant values off the stack to restore the context of the running process. Finally, one more special instruction is called: `iret`, or the **return-from-trap** instruction. This instruction is similar to a return from a procedure call, but simultaneously lowers the privilege level back to user mode and jumps back to the instruction immediately following the `int` instruc-

```
static int (*syscalls[])(void) = {
[SYS_chdir]   sys_chdir,
[SYS_close]   sys_close,
[SYS_dup]     sys_dup,
[SYS_exec]    sys_exec,
[SYS_exit]    sys_exit,
[SYS_fork]    sys_fork,
[SYS_fstat]   sys_fstat,
[SYS_getpid]  sys_getpid,
[SYS_kill]    sys_kill,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_mknod]   sys_mknod,
[SYS_open]    sys_open,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_unlink]  sys_unlink,
[SYS_wait]    sys_wait,
[SYS_write]   sys_write,
};

void
syscall(void)
{
  int num;

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

Figure 2.6: File: `syscall.c`

tion called to invoke the system call, restoring all the state that
has been saved into the trap frame. At this point, the user stub
for read() (as seen in Figure 2.2) is run again, which just uses

```
  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

Figure 2.7: File: trapasm.S

a normal return-from-procedure-call instruction (`ret`) in order to return to the caller.

2.8   Summary

We have seen the path in and out of the kernel on a system call. As you can tell, it is much more complex than a simple procedure call, and requires a careful protocol on behalf of the OS and hardware to ensure that application state is properly saved and restored on entry and return. As always, the concept is easy: with operating systems, the devil is always in the details.

References

[CK+08] "The xv6 Operating System"
Russ Cox, Frans Kaashoek, Robert Morris, Nickolai Zeldovich
From: http://pdos.csail.mit.edu/6.828/2008/index.html
xv6 was developed as a port of the original UNIX version 6 and represents a beautiful, clean, and simple way to understand a modern operating system. We'll try to peek inside it as often as possible to see some of the concepts we talk about in class in action.

[I09] "Intel 64 and IA-32 Architectures Software Developer's Manuals"
Intel, 2009
Available: http://www.intel.com/products/processor/manuals
In particular, pay attention to "Volume 3A: System Programming Guide Part 1" and "Volume 3B: System Programming Guide Part 2"

[L76] "Lions' Commentary on UNIX 6th Edition"
John Lions, 1976.
Info: http://en.wikipedia.org/wiki/Lions