# Project #2 - xv6 System Call

## 1. Intro to Kernel Hacking

To develop a better sense of how an operating system works, you will also do a few projects *inside* a real OS kernel. The kernel we'll be using is a port of the original Unix (version 6) and is runnable on modern x86 processors. It was developed at MIT and is a small and relatively understandable OS and thus an excellent focus for simple projects.

This first project is just a warmup, and thus relatively light on work. The goal of the project is simple: to add a system call to xv6. Your system call, **getreadcount()**, simply returns how many times that the **read()** system call has been called by user processes since the time that the kernel was booted.

## 2. Background

Proferssor Remzi Arpaci-Dusseau from Wisconsin-Madison made a great introduction video to the xv6 kernel. You can find it here.

It will walk you through how the Makefile works as well as the general structure of the kernel. You can skip through some of the parts that are not specific to xv6. This will be very helpful aid as you read through the next couple of sections.

More information about xv6, including a very useful book written by the MIT folks who built xv6, is available here. Do note, however, that we use a slightly older version of xv6 (for various pedagogical reasons), and thus the book may not match our code base exactly.

### xv6 System Call Background

To be able to implement this project, you'll have to understand a little bit about how xv6 implements system calls. As you recall from the OS book, a system call is a protected transfer of control from an application (running in *user mode*) to the OS (running in *kernelmode*).

The general approach, which we refer to as *limited direct execution* (*LDE*), enables the kernel to maintain control of the machine while generally letting user applications run efficiently and without kernel intervention. 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 level of the hardware, so that the OS can perform certain restricted operations.

## 3. 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?

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.

## 4. 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** (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).

```
.globl read;
read:
  movl $6, %eax;
  int $64;
  ret
```

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*).

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 64), which tells the hardware which trap type this is. In xv6, trap 64 is used to handle system calls. Any other arguments which are passed to the system call are passed on the stack.

# 5. 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:

```
int
mainc(void)
{
   ...
   tvinit(); // trap vectors initialized here
   ...
}
```

FILE: **main.c**

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

```
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");
}
```

FILE: **trap.c**

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):

```
// 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;                  \
}
```

FILE: **mmu.h**

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 manuals (particularly Chapters 3a and 3b) for more information.

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**, which calls **idtinit** in **trap.c**, which calls **lidt()** in the include file **x86.h**:

```
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 instruction is called to tell the hardware where to find the *interrupt 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

executing 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;
};
```

File: **x86.h**

# 6. 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**.

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 **vector64()** as found in **vectors.S** (which is automatically generated by the script **vectors.pl**).

```
.globl vector64
vector64:
  pushl $64
  jmp alltraps
```

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.

```
# 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
```

File: **trapasm.S**

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.

# 7. 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 interrupts 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 64 as defined somewhat arbitrarily in **traps.h**), which then handles the system call, as you see here:

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

FILE: **trap.c**

The code isn't too complicated. It checks if the current process (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.

```
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;
  }
}
```

File: **syscall.c**

## 8. Vectoring to The System Call

Once we finally get to the **syscall()** routine in **syscall.c**, not much work is left to do (see above). 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.

## 9. 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.

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

File: **trapasm.S**

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** instruction 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 the **usys.S** code) is run again, which just uses a normal return-from-procedure-call instruction (**ret**) in order to return to the caller.

## 10. 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.

## 11. Your System Call

Your new system call should look have the following return codes and parameters:

```
int getreadcount(void)
```

Your system call returns the value of a counter (perhaps called **readcount** or something like that) which is incremented every time any process calls the **read()** system call. That's it!

# 12. Tips

Watch Professor Remzi's video (https://www.youtube.com/watch?v=vR6z2QGcoo8)– it contains a detailed walk-through of all the things you need to know to unpack xv6, build it, and modify it to make this project successful.

One good way to start hacking inside a large code base is to find something similar to what you want to do and to carefully copy/modify that. Here, you should find some other system call, like **getpid()** (or any other simple call). Copy it in all the ways you think are needed, and then modify it to do what you need.

Most of the time will be spent on understanding the code. There shouldn't be a whole lot of code added.

You can add *printf* statements at appropriate places to debug the code.

Using gdb (the debugger) may be helpful in understanding code, doing code traces, and is helpful for later projects too. Get familiar with this fine tool!

- **Running Tests:**

Running tests for your system call is easy. Just do the following from inside the initial-xv6 directory:

```
prompt> ./test-getreadcounts.sh
```

If you implemented things correctly, you should get some notification that the tests passed. If not …
The tests assume that xv6 source code is found in the src/ subdirectory. If it's not there, the script will complain.
The test script does a one-time clean build of your xv6 source code using a newly generated makefile called Makefile.test. You can use this when debugging (assuming you ever make mistakes, that is), e.g.:

```
prompt> cd src/
prompt> make -f Makefile.test qemu-nox
```

You can suppress the repeated building of xv6 in the tests with the -s flag. This should make repeated testing faster:

```
prompt> ./test-getreadcounts.sh -s
```

The other usual testing flags are also available. See the testing README for details.

# Appendix – Environment Setup

Starting from this project, it is recommended for you to work on linux in your local machine.

**Follow the steps if you don't have linux:**

- Install Virtual Box for either Windows or Mac at https://www.virtualbox.org/
- Download Ubuntu 18.04 from https://ubuntu.com/download/desktop
- Create a new VM (30G-50G should be enough for the virtual machine disk size)
- Attach the Ubuntu image to the VM
- Start the VM and install ubuntu

**Install necessary packages:**

- For xv6, install qemu, expect, build-essentials, and gawk using this command

```
sudo apt-get install build-essential gawk qemu expect
```

# Appendix – Test options

The run-tests.sh script is called by various testers to do the work of testing. Each test is actually fairly simple: it is a comparison of standard output and standard error, as per the program specification.

In any given program specification directory, there exists a specific tests/ directory which holds the expected return code, standard output, and standard error in files called n.rc, n.out, and n.err (respectively) for each test n. The testing framework just starts at 1 and keeps incrementing tests until it can't find any more or encounters a failure. Thus, adding new tests is easy; just add the relevant files to the tests directory at the lowest available number.

The files needed to describe a test number n are:

- n.rc: The return code the program should return (usually 0 or 1)
- n.out: The standard output expected from the test
- n.err: The standard error expected from the test
- n.run: How to run the test (which arguments it needs, etc.)
- n.desc: A short text description of the test
- n.pre (optional): Code to run before the test, to set something up
- n.post (optional): Code to run after the test, to clean something up

There is also a single file called pre which gets run once at the beginning of testing; this is often used to do a more complex build of a code base, for example. To prevent repeated time-wasting pre-test activity, suppress this with the -s flag (as described below).

In most cases, a wrapper script is used to call ***run-tests.sh*** to do the necessary work.

The options for run-tests.sh include:

- -h (the help message)
- -v (verbose: print what each test is doing)
- -t n (run only test n)
- -c (continue even after a test fails)
- -d (run tests not from tests/ directory but from this directory instead)
- -s (suppress running the one-time set of commands in pre file)