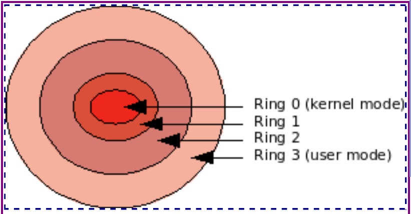
www.jamesmolloy.co.uk

Home » JamesM's kernel development tutorials

10. User mode (and syscalls)

Your kernel, at the moment, is running with the processor in "kernel mode", or "supervisor mode". Kernel mode makes available certain instructions that would usually be denied a user program - like being able to disable interrupts, or halt the processor.

Once you start running user programs, you'll want to make the jump from kernel mode to user mode, to restrict what instructions are available. You can also restrict read or write access to areas of memory. This is often used to 'hide' the kernel's code and data from user programs.



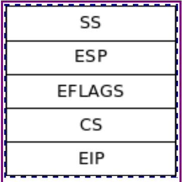
The x86's protection system.

10.1. Switching to user mode

The x86 is strange in that there is no direct way to switch to user mode. The only way one can reach user mode is to return from an exception that *began* in user mode. The only method of getting there in the first place is to set up the stack as if an exception in user mode had occurred, then executing an exception return instruction (IRET).

The IRET instruction expects, when executed, the stack to have the following contents (starting from the stack pointer - the lowermost address upwards):

- The instruction to continue execution at - the value of EIP.
- The code segment selector to change to.
- The value of the EFLAGS register to load.
- The stack pointer to load.
- The stack segment selector to change to.



SS
ESP
EFLAGS
CS
EIP

Stack prior to IRET.

The EIP, EFLAGS and ESP register values should be easy to work out, but the CS and SS values are slightly more difficult.

When we set up our GDT we set up 5 selectors - the NULL selector, a code segment selector for kernel mode, a data segment selector for kernel mode, a code segment selector for user mode, and a data segment selector for user mode.

They are all 8 bytes in size, so the selector indices are:

- 0x00: Null descriptor
- 0x08: Kernel code segment
- 0x10: Kernel data segment
- 0x18: User code segment
- 0x20: User data segment

We're currently using selectors 0x08 and 0x10 - for user mode we want to use selectors 0x18 and 0x20. However, it's not quite that straightforward. Because the selectors are all 8 bytes in size, the two least significant bits of the selector will always be zero. Intel use these two bits to represent the RPL - the *Requested Privilege Level*. These have currently been zero because we were operating in ring 0, but now that we want to move to ring three we must set them to '3'. If you wish to know more about the RPL and segmentation in general, you should read the intel manuals. There is far too much information for me to explain everything here.

So, this means that our code segment selector will be (0x18 | 0x3 = 0x1b), and our data segment selector will be (0x20 | 0x3 = 0x23).

10.1.1. task.c

This function should go in your task.c. We'll call it from main.c.

```
void switch_to_user_mode()
{
   // Set up a stack structure for switching to user mode.
   asm volatile("  \
     cli; \
     mov $0x23, %ax; \
     mov %ax, %ds; \
     mov %ax, %es; \
     mov %ax, %fs; \
     mov %ax, %gs; \
                   \
     mov %esp, %eax; \
     pushl $0x23; \
     pushl %eax; \
     pushf; \
     pushl $0x1B; \
     push $1f; \
     iret; \
   1: \
     ");
}
```

This code firstly disables interrupts, as we're working on a critical section of code. It then sets the ds, es, fs and gs segment selectors to our user mode data selector - 0x23.

Our aim is to return from the switch_to_user_mode() function in user mode, so to do that we need to not change the stack pointer. The next line saves the stack pointer in EAX, for reference later. We push our stack segment selector value (0x23), then push the value that we want the stack pointer to have after the IRET. This is the value of ESP before we changed anything on the stack (stored in EAX).

The pushf instruction pushes the current value of EFLAGS - we then push the CS selector value (0x1b).

The next statement is a little special, and can confuse some people who are not used to AS syntax. we push the value of $1f onto the stack. $1f means "the address of the next label '1:', searching forward". Read the GNU AS manual for more information, but numeric symbols are treated differently by it - you can have as many definitions of "1:", "2:" etc as you like.

After this we execute our IRET, and hopefully we should now be executing code at the "1:" line with the same stack, in user mode.

10.1.2. Something to watch out for

You may notice that we disabled interrupts before starting the mode switch. A problem now occurs - how do we re-enable interrupts? You'll find that executing *sti* in user mode will cause a general protection fault, however if we enable interrupts before we do our IRET, we may be interrupted at a bad time.

A solution presents itself if you know how the *sti* and *cli* instructions work - they just set the 'IF' flag in EFLAGS. Wikipedia tells us that the IF flag has a mask of 0x200, so what you *could* do, is insert these lines just after the 'pushf' in the asm above:

```
pop %eax ; Get EFLAGS back into EAX. The only way to read EFLAGS is to pushf then pop.
or %eax, $0x200 ; Set the IF flag.
push %eax ; Push the new EFLAGS value back onto the stack.
```

This solution means that interrupts get reenabled atomically as IRET is executing - perfectly safe.

10.2. System calls

Code running in user mode cannot run any code which is located in or accesses a supervisor-only area of memory (see the page table entry flags) or any code which uses privileged instructions such as *hlt*. Most kernels therefore provide an interface by which common functions can be executed. A call to the kernel through this interface is called a "system call".

The historical, easy, and still widely used way to implement system calls on x86 is to use software interrupts. The user program will set up one register to indicate which system function it would like to execute, then set up parameters in others. It would then execute a software interrupt to a specific vector - linux uses 0x80. The software interrupt causes a mode change to ring 0 - the kernel will have a handler for this interrupt vector, and dispatch the system call appropriately.

One thing that is important to note is that the kernel, when executing interrupt handling code, requires a valid stack to work with. If it doesn't have one, the processor will double fault (and then eventually triple fault because the double fault handler needs a valid stack too!). This would obviously be a very easy way for a malicious user to bring down your system, so it is normal practice to, on mode change from ring 3 to ring 0, switch to a new stack designed solely for use by the kernel, and which is guaranteed to be valid.

Obviously, if you want your kernel to be preemptible (i.e. you want to be able to task switch while executing code inside the kernel) you'll need one of these kernel stacks per task, or you'll end up overwriting one task's data when executing another task!

10.2.1. The task state segment

The X86 architecture has support for hardware-assisted task switching by way of a list of Task State Segments (TSS). In this tutorial set we have (like BSD, linux and most x86 operating systems) decided against using it and opted instead for a software based solution. The main reason for this is that hardware task switching is actually not much faster than software, and software task switching is far more portable between platforms.

With that said, the way the x86 architecture is designed we have no choice but to use at least one TSS. This is because when a program in user mode (ring 3) executes a system call (software interrupt) the processor automatically looks in the current TSS and sets the stack segment (SS) and stack pointer (ESP) to what it finds in the SS0 and ESP0 fields ('0' because it's switching to ring 0) - in essence this switches from the user's stack to your kernel stack.

Normal practice when implementing software task switching is just to have one TSS, and update the ESP0 field of it whenever a task switch takes place - this is the minimum work neccessary to allow system calls to work properly.

10.2.1.1. descriptor_tables.h

We'll need to add a TSS entry structure into the descriptor_tables header file:

```
// A struct describing a Task State Segment.
struct tss_entry_struct
{
   u32int prev_tss;   // The previous TSS - if we used hardware task switching this would form a linked list.
   u32int esp0;       // The stack pointer to load when we change to kernel mode.
   u32int ss0;        // The stack segment to load when we change to kernel mode.
   u32int esp1;       // Unused...
   u32int ss1;
   u32int esp2;
   u32int ss2;
   u32int cr3;
   u32int eip;
   u32int eflags;
   u32int eax;
   u32int ecx;
   u32int edx;
   u32int ebx;
   u32int esp;
   u32int ebp;
   u32int esi;
   u32int edi;
   u32int es;         // The value to load into ES when we change to kernel mode.
   u32int cs;         // The value to load into CS when we change to kernel mode.
   u32int ss;         // The value to load into SS when we change to kernel mode.
   u32int ds;         // The value to load into DS when we change to kernel mode.
   u32int fs;         // The value to load into FS when we change to kernel mode.
   u32int gs;         // The value to load into GS when we change to kernel mode.
   u32int ldt;        // Unused...
   u16int trap;
   u16int iomap_base;
} __attribute__((packed));

typedef struct tss_entry_struct tss_entry_t;
```

10.2.1.2. descriptor_tables.c

We'll also need code to initialise the TSS. The TSS is actually stored as a pointer inside the GDT, so we'll need another GDT entry too.

```
// Lets us access our ASM functions from our C code.
...
extern void tss_flush();

// Internal function prototypes.
```

```
...
static void write_tss(s32int,u16int,u32int);
...

tss_entry_t tss_entry;

static void init_gdt()
{
   gdt_ptr.limit = (sizeof(gdt_entry_t) * 6) - 1;
   gdt_ptr.base  = (u32int)&gdt_entries;

   gdt_set_gate(0, 0, 0, 0, 0);                // Null segment
   gdt_set_gate(1, 0, 0xFFFFFFFF, 0x9A, 0xCF); // Code segment
   gdt_set_gate(2, 0, 0xFFFFFFFF, 0x92, 0xCF); // Data segment
   gdt_set_gate(3, 0, 0xFFFFFFFF, 0xFA, 0xCF); // User mode code segment
   gdt_set_gate(4, 0, 0xFFFFFFFF, 0xF2, 0xCF); // User mode data segment
   write_tss(5, 0x10, 0x0);

   gdt_flush((u32int)&gdt_ptr);
   tss_flush();
}

// Initialise our task state segment structure.
static void write_tss(s32int num, u16int ss0, u32int esp0)
{
   // Firstly, let's compute the base and limit of our entry into the GDT.
   u32int base = (u32int) &tss_entry;
   u32int limit = base + sizeof(tss_entry);

   // Now, add our TSS descriptor's address to the GDT.
   gdt_set_gate(num, base, limit, 0xE9, 0x00);

   // Ensure the descriptor is initially zero.
   memset(&tss_entry, 0, sizeof(tss_entry));

   tss_entry.ss0  = ss0;  // Set the kernel stack segment.
   tss_entry.esp0 = esp0; // Set the kernel stack pointer.

   // Here we set the cs, ss, ds, es, fs and gs entries in the TSS. These specify what
   // segments should be loaded when the processor switches to kernel mode. Therefore
   // they are just our normal kernel code/data segments - 0x08 and 0x10 respectively,
   // but with the last two bits set, making 0x0b and 0x13. The setting of these bits
   // sets the RPL (requested privilege level) to 3, meaning that this TSS can be used
   // to switch to kernel mode from ring 3.
   tss_entry.cs   = 0x0b;
   tss_entry.ss = tss_entry.ds = tss_entry.es = tss_entry.fs = tss_entry.gs = 0x13;
}
```

Well define tss_flush in a second. We'll also need a function to update the TSS entry when we change tasks, so it holds the address of the correct kernel stack;

```
void set_kernel_stack(u32int stack)
{
   tss_entry.esp0 = stack;
}
```

10.2.1.3. gdt.s

Here we define our tss_flush function. In it, we tell the processor where to find our TSS within the GDT.

```
[GLOBAL tss_flush]    ; Allows our C code to call tss_flush().
tss_flush:
   mov ax, 0x2B      ; Load the index of our TSS structure - The index is
                     ; 0x28, as it is the 5th selector and each is 8 bytes
                     ; long, but we set the bottom two bits (making 0x2B)
                     ; so that it has an RPL of 3, not zero.
   ltr ax            ; Load 0x2B into the task state register.
   ret
```

Notice that we have to specify an RPL, just like when we switched to user mode.

10.2.2. The system call interface

We're going to create a syscall interface similar to Linux's, in that it uses interrupt vector 0x80. Our defined interrupt handlers don't currently reach that high, so we'll have to add another - a "ISR_NOERRCODE 128" in interrupt.s, and an extra idt_set_gate in descriptor_tables.c (and of course an extra function prototype in descriptor_tables.h).

10.2.2.1. syscall.h

Initially, we just need to give an interface for starting the syscall interface...

```
// syscall.h -- Defines the interface for and structures relating to the syscall dispatch system.
// Written for JamesM's kernel development tutorials.

#ifndef SYSCALL_H
#define SYSCALL_H

#include "common.h"

void initialise_syscalls();

#endif
```

10.2.2.2. syscall.c

... and then implement it. As mentioned previously, the normal way to dispatch syscalls is to have one register contain a number which indexes a table of functions. the given function is then executed.

For the moment, we just have three functions which can be called via syscall - the three monitor output functions. This will enable us to check whether our code works easier, by allowing text output in user mode.

```
// syscall.c -- Defines the implementation of a system call system.
// Written for JamesM's kernel development tutorials.

#include "syscall.h"
#include "isr.h"

#include "monitor.h"

static void syscall_handler(registers_t *regs);

static void *syscalls[3] =
{
   &monitor_write,
   &monitor_write_hex,
   &monitor_write_dec,
};
u32int num_syscalls = 3;

void initialise_syscalls()
{
   // Register our syscall handler.
   register_interrupt_handler (0x80, &syscall_handler);
}

void syscall_handler(registers_t *regs)
{
   // Firstly, check if the requested syscall number is valid.
   // The syscall number is found in EAX.
   if (regs->eax >= num_syscalls)
       return;

   // Get the required syscall location.
   void *location = syscalls[regs->eax];

   // We don't know how many parameters the function wants, so we just
   // push them all onto the stack in the correct order. The function will
   // use all the parameters it wants, and we can pop them all back off afterwards.
   int ret;
   asm volatile (" \
     push %1; \
     push %2; \
     push %3; \
     push %4; \
     push %5; \
     call *%6; \
     pop %%ebx; \
     pop %%ebx; \
     pop %%ebx; \
     pop %%ebx; \
     pop %%ebx; \
   " : "=a" (ret) : "r" (regs->edi), "r" (regs->esi), "r" (regs->edx), "r" (regs->ecx), "r" (regs->ebx), "r" (location));
   regs->eax = ret;
}
```

So here we have a table of the addresses of our syscall functions. The initialise_syscalls function merely adds the syscall_handler function as an interrupt handler for interrupt 0x80.

The syscall_handler function checks that the given function index is valid, then gets the address of the function to call, and then pushes all the parameters we were given onto the stack, call the function, and pop all the parameters back off the stack.

As is customary it also puts the return value of the function call in EAX, when the interrupt returns.

10.2.3. Helper macros

So a syscall from user mode would look something like this:

```
mov eax, call>
mov ebx,
mov ecx,
mov edx,
mov esi,
mov edi,
int 0x80 ; execute syscall
         ; return value of syscall is in EAX.
```

This is, however, a little unwieldy. We can simplify this by creating some helper macros to define stub functions that contain inline assembler that actually does the syscall;

In syscall.h

```
#define DECL_SYSCALL0(fn) int syscall_##fn();
#define DECL_SYSCALL1(fn,p1) int syscall_##fn(p1);
#define DECL_SYSCALL2(fn,p1,p2) int syscall_##fn(p1,p2);
#define DECL_SYSCALL3(fn,p1,p2,p3) int syscall_##fn(p1,p2,p3);
#define DECL_SYSCALL4(fn,p1,p2,p3,p4) int syscall_##fn(p1,p2,p3,p4);
#define DECL_SYSCALL5(fn,p1,p2,p3,p4,p5) int syscall_##fn(p1,p2,p3,p4,p5);

#define DEFN_SYSCALL0(fn, num) \
int syscall_##fn() \
{ \
 int a; \
 asm volatile("int $0x80" : "=a" (a) : "0" (num)); \
 return a; \
}

#define DEFN_SYSCALL1(fn, num, P1) \
int syscall_##fn(P1 p1) \
{ \
 int a; \
 asm volatile("int $0x80" : "=a" (a) : "0" (num), "b" ((int)p1)); \
 return a; \
}

#define DEFN_SYSCALL2(fn, num, P1, P2) \
int syscall_##fn(P1 p1, P2 p2) \
{ \
 int a; \
 asm volatile("int $0x80" : "=a" (a) : "0" (num), "b" ((int)p1), "c" ((int)p2)); \
 return a; \
}

...
```

So we have a macro "DECL_SYSCALLX", which declares a stub function for a function *fn*, with *X* parameters, they being of type *p1..pn*.

The macro "DEFN_SYSCALLX" actually defines the stub function, which is just a piece of inline assembly. The *num* parameter is the index in the syscall function table to call.

So to define our monitor_* functions, we should declare them in syscall.h:

```
DECL_SYSCALL1(monitor_write, const char*)
DECL_SYSCALL1(monitor_write_hex, const char*)
DECL_SYSCALL1(monitor_write_dec, const char*)
```

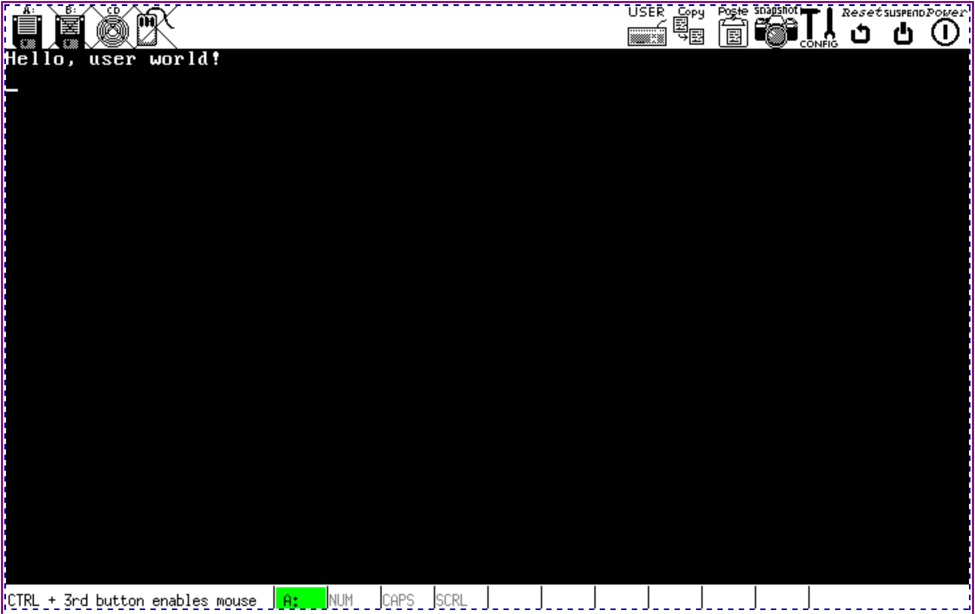
and define them in syscall.c:

```
DEFN_SYSCALL1(monitor_write, 0, const char*);
DEFN_SYSCALL1(monitor_write_hex, 1, const char*);
DEFN_SYSCALL1(monitor_write_dec, 2, const char*);
```

10.3. Testing

In main.c

```
// Start paging.
initialise_paging();

// Start multitasking.
initialise_tasking();

// Initialise the initial ramdisk, and set it as the filesystem root.
fs_root = initialise_initrd(initrd_location);

initialise_syscalls();

switch_to_user_mode();

syscall_monitor_write("Hello, user world!\n");

return 0;
```



```
Hello, user world!
_
```

Hello, user world!

With this test code in main.c, you should have a functional user mode and syscall interface, suitable for running untrusted user programs.

Full source code and image file is available here.

10.3.1. Possible problems

If you keep getting page faults when jumping to user mode, make sure that your kernel code/data is set to be user-accessible. When you actually load user programs you won't want this to be the case, however at the moment we merely jump back to the kernel and execute code in main(), so it needs to be accessible in user mode!