Getting to Ring 3

From OSDev Wiki

This page is a work in progress and may thus be incomplete. Its content may be changed in the near future.

As fun as making a kernel is, eventually we have to get outside the kernel into userspace. This involves getting from ring 0 to ring 3. I am sure all of us wish we could just make a GDT entry and - poof - ring 3 works, but Intel wants us to pull our hair out at least a little with their Task State Segment. So, in order to get to ring 3 we must do the following:

- Get 2 new GDT entries (at least) configured for ring 3.
  - These entries are needed for the user's code and data segments (one each)
- Set up a barebones TSS with an ESP0 stack.
  - When an interrupt(be it fault, IRQ, or software interrupt) happens while the CPU is in user mode the CPU needs to know where the kernel stack is located, this location is stored in the ESP0 entry of the TSS.
- Set up an IDT entry for ring 3 system call interrupts (optional, actually).
  - System calls are the way user code requests the kernel to do IO and process management. For more information see System Calls

Contents

Requirements

I'm not going to go through making a whole kernel that can get to ring 3. I will assume you have a decent and usable ring 0 GDT and IDT, along with being able to handle IRQs properly. I also assume you will be multitasking, and so will cover switching ring 3>ring 0(switch task)>ring 3.

GDT

This is my GDT struct I will be using(it's split into bit-fields)

```
struct gdt_entry_bits
```

```
{
        unsigned int limit_low:16;
        unsigned int base_low : 24;
     //attribute byte split into bitfields
        unsigned int accessed :1;
        unsigned int read_write :1; //readable for code, writable for dat
        unsigned int conforming_expand_down :1; //conforming for code, ex
        unsigned int code :1; //1 for code, 0 for data
        unsigned int always_1 :1; //should be 1 for everything but TSS ar
        unsigned int DPL :2; //priveledge level
        unsigned int present :1;
     //and now into granularity
        unsigned int limit_high :4;
        unsigned int available :1;
        unsigned int always_0 :1; //should always be 0
        unsigned int big :1; //32bit opcodes for code, uint32_t stack for
        unsigned int gran :1; //1 to use 4k page addressing, 0 for byte a
        unsigned int base_high :8;
} __packed; //or __attribute__((packed))
```

We will be doing a simple setup, and I will assume you will later implement paging in your OS, so we will use only 2 ring 3 segments both with base of 0 and limit of 0xFFFFFFFF so we will setup our two GDT segments like this:

```
//....insert your ring 0 segments here or whatever
gdt_entry_bits *code;
gdt_entry_bits *data;
//I assume your ring 0 segments are in gdt[1] and gdt[2] (0 is null segme
code=(void*)&gdt[3]; //gdt is a static array of gdt_entry_bits or equival
data=(void*)&gdt[4];
code->limit_low=0xFFFF;
code->base_low=0;
code->accessed=0;
code->read_write=1; //make it readable for code segments
code->conforming=0; //don't worry about this..
code->code=1; //this is to signal its a code segment
code->always_1=1;
code->DPL=3; //set it to ring 3
code->present=1;
code->limit_high=0xF;
code->available=1;
code->always_0=0;
code->big=1; //signal it's 32 bits
code->gran=1; //use 4k page addressing
code->base_high=0;
*data=*code; //copy it all over, cause most of it is the same
data->code=0; //signal it's not code; so it's data.

install_tss(&gdt[5]); //we'll implement this function later...
```

```
//...go on to install GDT segments and such
//after those are installed we'll tell the CPU where our TSS is:
flush_tss(); //implement this later
```

Ok, so now we have our two user mode segments. Now technically, we can get to user mode right now with these two segments. The problem is we can't get back to ring 0 for system calls or faults or even IRQs. That is where the TSS comes in.

The TSS

The TSS can be used for multitasking, though it is recommended to use software multitasking for these reasons:

- Software task switching is faster(usually)
- When you port your OS to a different CPU, it won't have the TSS, so you'll have to implement software task switching anyway
- x86 64bit mode does not allow you to use the TSS for task switching.

Since we will be using the software multitasking approach, the TSS will contain a lot of junk we don't need. Here is the structure of the TSS

```
// A struct describing a Task State Segment.
struct tss_entry_struct
{
   uint32_t prev_tss;   // The previous TSS - if we used hardware task sw
   uint32_t esp0;       // The stack pointer to load when we change to ke
   uint32_t ss0;        // The stack segment to load when we change to ke
   uint32_t esp1;       // everything below here is unusued now..
   uint32_t ss1;
   uint32_t esp2;
   uint32_t ss2;
   uint32_t cr3;
   uint32_t eip;
   uint32_t eflags;
   uint32_t eax;
   uint32_t ecx;
   uint32_t edx;
   uint32_t ebx;
   uint32_t esp;
   uint32_t ebp;
   uint32_t esi;
   uint32_t edi;
   uint32_t es;
   uint32_t cs;
   uint32_t ss;
   uint32_t ds;
   uint32_t fs;
   uint32_t gs;
```

```
   uint32_t ldt;
   uint16_t trap;
   uint16_t iomap_base;
} __packed;

typedef struct tss_entry_struct tss_entry_t;
```

As you can see.. a lot of wasted crap you don't need. But, intel demands it be used, so... Basically what we want to do to setup this TSS structure is give it an initial esp0 stack and setup the segments to point to our kernel segments, and really that's it.. so we can do something like this:

```
/**Ok, this is going to be hackish, but we will salvage the gdt_entry_bit
So some of these names of the fields will actually be different.. maybe I
tss_entry_t tss_entry;

void write_tss(gdt_entry_bits *g)
{
   // Firstly, let's compute the base and limit of our entry into the GDT
   uint32_t base = (uint32_t) &tss_entry;
   uint32_t limit = base + sizeof(tss_entry);

   // Now, add our TSS descriptor's address to the GDT.
   g->base_low=base&0xFFFFFF; //isolate bottom 24 bits
   g->accessed=1; //This indicates it's a TSS and not a LDT. This is a ch
   g->read_write=0; //This indicates if the TSS is busy or not. 0 for not
   g->conforming_expand_down=0; //always 0 for TSS
   g->code=1; //For TSS this is 1 for 32bit usage, or 0 for 16bit.
   g->always_1=0; //indicate it is a TSS
   g->DPL=3; //same meaning
   g->present=1; //same meaning
   g->limit_high=(limit&0xF0000)>>16; //isolate top nibble
   g->available=0;
   g->always_0=0; //same thing
   g->big=0; //should leave zero according to manuals. No effect
   g->gran=0; //so that our computed GDT limit is in bytes, not pages
   g->base_high=(base&0xFF000000)>>24; //isolate top byte.

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

   tss_entry.ss0  = REPLACE_KERNEL_DATA_SEGMENT;  // Set the kernel stack
   tss_entry.esp0 = REPLACE_KERNEL_STACK_ADDRESS; // Set the kernel stack
   //note that CS is loaded from the IDT entry and should be the regular
}

void set_kernel_stack(uint32_t stack) //this will update the ESP0 stack u
{
   tss_entry.esp0 = stack;
}
```

Now, I know you may spend a while looking at that atrocious code..but I do believe it works. Oh, and here is our flush_tss function: (in yasm/nasm syntax)

```
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 byte
                     ; 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
```

Ok, so now we are just about ready to do some ring 3 fun stuff!!

Entering Ring 3

Ok, the x86 is really a tricky CPU. The only way to get to ring 3 is to fool the processor into thinking it was already in ring 3 to start with. We effectively do this using an iret. I'll give you a simple example on how to execute something as ring 3:(yasm/nasm syntax)

```
GLOBAL _jump_usermode ;you may need to remove this _ to work right..
EXTERN _test_user_function
_jump_usermode:
     mov ax,0x23
     mov ds,ax
     mov es,ax
     mov fs,ax
     mov gs,ax ;we don't need to worry about SS. it's handled by iret

     mov eax,esp
     push 0x23 ;user data segment with bottom 2 bits set for ring 3
     push eax ;push our current stack just for the heck of it
     pushf
     push 0x1B; ;user data segment with bottom 2 bits set for ring 3
     push _test_user_function ;may need to remove the _ for this to work
     iret
;end
```

Now then, this will call the C function test_user_function and it will be operating in user mode! There is no easy way of getting back to ring 0(excluding IRQs) except for by setting up a task switching system, which you really should have in place to properly appreciate ring 3 in the first place.. But if you would

like to test things out in user mode, just have the test_user_function execute a cli or other privileged instruction and you'll be pleased by a GPF. I won't give you source examples on implementing this into your task switching system, as these vary a lot by operating system.

Multitasking considerations

There are a lot of subtle things with user mode and task switching that you may not realize at first. First: Whenever a system call interrupt happens, the first thing that happens is the CPU changes to ESP0 stack. Then, it will push all the system information. So when you enter the interrupt handler, your working off of the ESP0 stack. This could become a problem with 2 ring 3 tasks going if all you do is merely push context info and change esp. Think about it. you will change the esp, which is the esp0 stack, to the other tasks esp, which is the same esp0 stack. So, what you must do is change the ESP0 stack(along with the interrupt pushed ESP stack) on each task switch, or you'll end up overwriting yourself.