Tutorial 06: Multi-Tasking

Our Goal

In this tutorial, there will be several tasks running in parallel in Skelix, and each task use its own LDT, we are going to implement preemptively task switching.

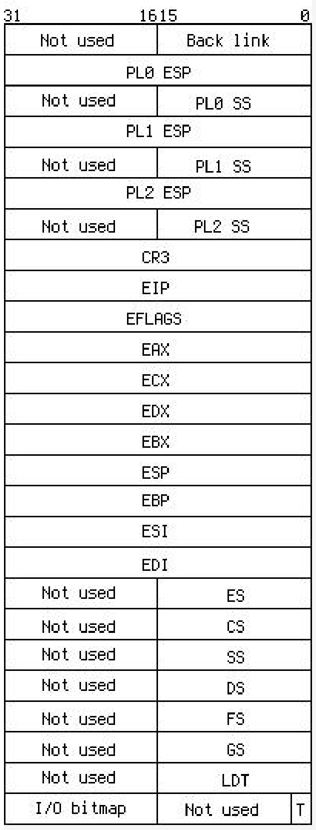
Download source code

TSS

At first let's clarify one opinion, running multiple tasks concurrently on one single processor(one core) is not going to happen, in reality, the processor switches tasks very fast, say 100 times per second, so it looks like several processes are running at the same time.

The 386 processor has hardware support for multitasking, but certainly it also can be accomplished manually, here is a program I wrote before, it switches the execution of several code snippets. We all know there is only one set of registers in every single CPU, and they are used by all processes, so when a running processes is stopped and before it is switched to another process, all the informations about this process which are registers and some other information relate to them should be stored, because another process is going to use them. This information called "context" in general, and the 386 processor use the task state segments (TSS) to store this information, which is at least 104 bytes long, and a TSS descriptor refers to it. TSS descriptors only appear in the GDT, that means a user task can not switch to any specific process by itself via LDT. When we switch processes in the hardware way, the processor stores all informations in the TSS automatically.

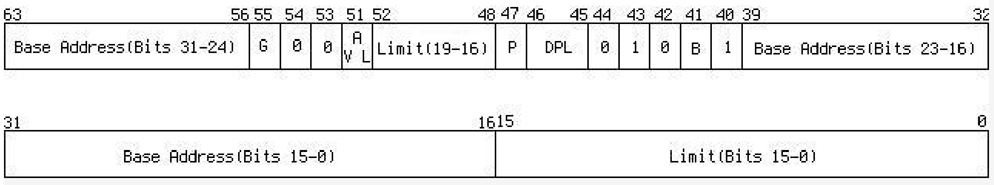
The TSS defines the state of the execution environment for a task. Now let's take a look at the TSS structure,



31 16	15 0	
Not used	Back link	
PL0 ESP		
Not used	PL0 SS	
PL1 ESP		
Not used	PL1 SS	
PL2 ESP		
Not used	PL2 SS	
CR3		
EIP		
EFLAGS		
EAX		
ECX		
EDX		
EBX		
ESP		
EBP		
ESI		
EDI		
Not used	ES	
Not used	CS	
Not used	SS	
Not used	DS	
Not used	FS	
Not used	GS	
Not used	LDT	
I/O bitmap	Not used	T

It is a big table, but it is not that scary if you look into it. For those "not used" fields, they are reserved by Intel. The 386 processor supports nested task, that means when task 1 switch to task 2, then the "Back link" field of task 2 is set to the selector of task 1 and the NT(Nested Task) bit in its `EFLAGS` is set as well, so when task 2 returns, the processor knows which task is going to take control. We know that the 386 processor supports 4 different privileges, and TSS allows each privilege uses different stacks when task switching, so there are 4 different stacks for one process can be used. We are going to talk about `CR3` register in following tutorials. We are going to talk about LDT in a short while. About I/O bitmap an T bit, we don't care about them at this moment.

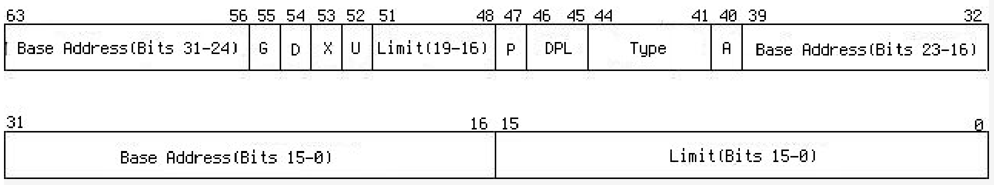
Like other descriptors in GDT, the TSS also need a descriptor in GDT refer to it, here is its format,



63 56	55	54	53	51 52	48	47	46 45	44	43	42	41	40	39 32
Base Address(Bits 31-24)	G	0	0	AVL	Limit(19-16)	P	DPL	0	1	0	B	1	Base Address(Bits 23-16)

31 16	15 0
Base Address(Bits 15-0)	Limit(Bits 15-0)

General GDT descriptor,



63 56	55	54	53	52	51 48	47	46 45	44 41	40	39 32
Base Address(Bits 31-24)	G	D	X	U	Limit(19-16)	P	DPL	Type	A	Base Address(Bits 23-16)

31 16	15 0
Base Address(Bits 15-0)	Limit(Bits 15-0)

We can compare it to a general segment descriptor, we can see the D(data or code segment) and X(not used) bits are set to zero, the AVL bit is available to users. The type is 010B, the B(bit 41) is the busy bit of this TSS descriptor, because one process can only own one TSS, so once it is executing, the B bit of its descriptor is set to indicate its TSS can not be used any more just in case of the reentrant. A(access) is set to 1, but the TSS can neither be read nor written by processes even A bit is set to writable. Other field of the descriptor have the same meaning compares to normal data and code segments, just one thing we have to know, the limit field of the descriptor must can present a memory space that at least 104 bytes long which is the minimum length of TSS. There is another thing I have to mention, in reality most OS use software context switching because hardware context switching just save the registers we mentioned above, but not include `FPU`, `MMX`, `SSE` etc. Since we are not going to use them in Skelix, so it is safe to use TSS to do environment saving.

In this tutorial, DPL fields of TSS descriptors are going to be set to zero, so that only the kernel has the right to perform task switching. Just like `GDTR` and `IDTR`, TSS descriptors also has its own specific register for task switching, called `TR`, it can be loaded to this register by instruction `LTR`. And the selector refers to TSS descriptor in GDT can not be load into any other segment registers, or an exception will occur.

There are several ways to perform task switching, and we are going to use a far jump to a TSS descriptor in GDT to achieve it. All those boring stuffs I mentioned above serve for this purpose. If you want to find out alternate ways to do it, you'd better read some code from wherever possible, because there are really only few books on this topic.

During task switching, the processor first stores the context of current task in the current TSS, then loads the task register of the new task, then loads the context of the new task from the new TSS with the new task's descriptor in GDT, finally executes the new task.

Let's take a look at the code, this is the TSS structure we are going to use,

06/include/task.h

```
struct TSS_STRUCT {
        unsigned int back_link;
        unsigned int esp0, ss0;
        unsigned int esp1, ss1;
        unsigned int esp2, ss2;
        unsigned int cr3;
        unsigned int eip;
        unsigned int eflags;
        unsigned int eax,ecx,edx,ebx;
        unsigned int esp, ebp;
        unsigned int esi, edi;
        unsigned int es, cs, ss, ds, fs, gs;
        unsigned int ldt;
        unsigned int trace_bitmap;
};
```

It is exactly the same as the TSS structure I mentioned above, but because this structure is going to be used by the processor, so it has to be 104 bytes long and no paddings in it, if you use IA-64 or other arches or other compilers, check documents youself.

For a working OS, those information are not enough, so anothing structure `TASK_STRUCT` is used to wrap TSS structure.

```
#define TS_RUNNING      0
#define TS_RUNABLE      1
#define TS_STOPPED      2

struct TASK_STRUCT {
        struct TSS_STRUCT tss;
        unsigned long long tss_entry;
        unsigned long long ldt[2];
        unsigned long long ldt_entry;
        int state;
        int priority;
        struct TASK_STRUCT *next;
};

#define DEFAULT_LDT_CODE        0x00cffa000000ffffULL
#define DEFAULT_LDT_DATA        0x00cff2000000ffffULL
```

```
#define INITIAL_PRIO            200
```

This is all we need at this moment, those `TS_*` stuff defines all states that a process can be at. The `tss_entry` in `TASK_STRUCT`defines the descriptor of the tss field in this structure, I am going to explain it in a short while. Two `*ldt*` fields will be explained later on. `state` field store the current state of this process, which is one of those `TS_*`. `priority` indicates the sequence of the execution of processes in system, and the new task will be given a initial priority `INITIAL_PRIO`. All tasks in Skelix are managed as a link, the `next` field defines the next task in the link.

Now let's look at an example, this is the `TASK_STRUCT` structure for task 0, that is the first task in system, when kernel finishes all initialization, it becomes task 0.

```
static unsigned long TASK0_STACK[256] = {0xf};
```

This is the stack used as the CPL0 stack for task 0. Whitout that 0xF, I encountered a problem during my compiling, if it is just initialized like `static unsigned long TASK0_STACK[256];` then this memory area just vanishes (Shen Feng explains, without 0xF, `TASK0_STACK[256]` would be an zero-initialized static data, and it would be in `.bss` section, and the object file only record its length and start address etc.. That's why a nonzero value has to be used to keep it in `.data` section.)

```
struct TASK_STRUCT TASK0 = {
        /* tss */
        {       /* back_link */
                0,
                /* esp0                                    ss0 */
                (unsigned)&TASK0_STACK+sizeof TASK0_STACK, DATA_SEL,
```

Make `esp0` points to the "bottom" of the stack. `DATA_SEL` and `CODE_SEL` appears in next few lines are defined at**`06/include/kernel.h`**, they are selectors of data and code segments in GDT.

```
                /* esp1 ss1 esp2 ss2 */
                0, 0, 0, 0,
                /* cr3 */
                0,
                /* eip eflags */
                0, 0,
                /* eax ecx edx ebx */
                0, 0, 0, 0,
                /* esp ebp */
                0, 0,
                /* esi edi */
                0, 0,
                /* es          cs             ds */
                USER_DATA_SEL, USER_CODE_SEL, USER_DATA_SEL,
                /* ss          fs             gs */
                USER_DATA_SEL, USER_DATA_SEL, USER_DATA_SEL,
                /* ldt */
                0x20,
                /* trace_bitmap */
                0x00000000},
                /* tss_entry */
                0,
                /* idt[2] */
                {DEFAULT_LDT_CODE, DEFAULT_LDT_DATA},
                /* idt_entry */
                0,
                /* state */
                TS_RUNNING,
                /* priority */
                INITIAL_PRIO,
                /* next */
                0,
};
```

Now we have the TSS, we are going to create a TSS descriptor, remember there are two reserved place in GDT.

06/bootsect.s

```
gdt:
        .quad    0x0000000000000000 # null descriptor
        .quad    0x00cf9a000000ffff # cs
        .quad    0x00cf92000000ffff # ds
        .quad    0x0000000000000000 # reserved for tss
        .quad    0x0000000000000000 # reserved for ldt
```

The forth entry(0x3) are reserved for TSS of current running task, so a macro `CURR_TASK_TSS = 3` has been defined as an index refers to this position in GDT. We are going to let the current task uses this place, once it gives up the control, it saves its descriptor in its `tss_entry` of `TASK_STRUCT`. When a new task takes over the control, it loads its own TSS descriptor from its`TASK_STRUCT` to this place. In this way, we can allow unlimited tasks works in system. Because there is a length limit about GDT, it can only have 8096 descriptors, actually, Linux has the limit of tasks for quite long time, I don't know why it has this limit, because eliminating this limit seems to be quite simple.

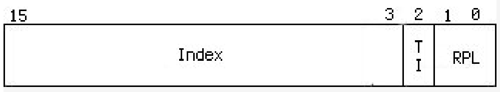
```
unsigned long long
set_tss(unsigned long long tss) {
        unsigned long long tss_entry = 0x0080890000000067ULL;
        tss_entry |= ((tss)<<16) & 0xffffff0000ULL;
        tss_entry |= ((tss)<<32) & 0xff00000000000000ULL;
        return gdt[CURR_TASK_TSS] = tss_entry;
}
```

`set_tss` generates the TSS descriptor and put it into GDT, we can see it set the DPL of descriptor to 0, so only kernel can use this descriptor.

```
unsigned long long
get_tss(void) {
        return gdt[CURR_TASK_TSS];
}
```

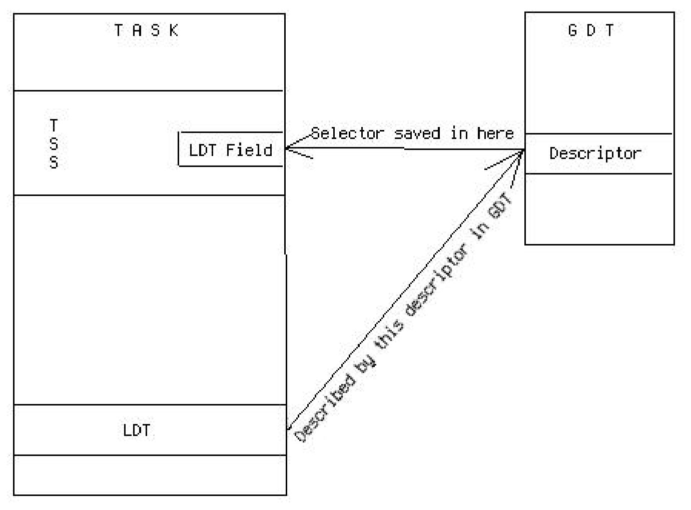
LDT

It has been a while since we use GDT, LDT should be fairly easy at this point. LDT is just like GDT, but it is local, every task can have its own LDT, we are going to let every task in Skelix has two descriptors in LDT, the first one is for the code segment, and the second one is for data and stack segments. Now let's go over the selector format,



we are going to let all tasks work at privilege level 3, that means RPL=11b, and TI=1 indicates this selector refers to a LDT. Unlike GDT, the first descriptor of LDT can be used by users, so the selector refers to code segment will be 0x7 , and the selector refers to data and stack segments will be 0xF.

The LDT field in TSS is the selector which refers to an descriptor in GDT, this descriptor describes the LDT. I know it sounds confusing, so let's make it clear, first we let every task owns a LDT, and this LDT resides in somewhere in memory(we do not concern about virtual memory at this stage), and we need a descriptor to describe this memory area which contains this LDT, and this descriptor will be put in GDT, and the selector refers to this descriptor will be put into the LDT field in TSS of this task.



The third descriptor of GDT is for current running task's TSS, and the forth descriptor is quite clear, it is for current task's LDT. So theirs selector will be 0x18 and 0x20 respectively. Like functions relate to TSS, we also have,

06/task.c

```
unsigned long long
set_ldt(unsigned long long ldt) {
        unsigned long long ldt_entry = 0x008082000000000fULL;
        ldt_entry |= ((ldt)<<16) & 0xffffff0000ULL;
        ldt_entry |= ((ldt)<<32) & 0xff00000000000000ULL;
        return gdt[CURR_TASK_LDT] = ldt_entry;
}
```

It is similar to GDT entries except for the DPL is 3 instead 0.

```

unsigned long long
get_tss(void) {
        return gdt[CURR_TASK_TSS];
}
```

At this moment we will let all tasks have the same LDT content, it means they share the same memory space, that is not a good idea, but I will give them separate memory space by using virtual memory, it will be introduced in later tutorial.

06/include/task.h

```
#define DEFAULT_LDT_CODE        0x00cffa000000ffffULL
#define DEFAULT_LDT_DATA        0x00cff2000000ffffULL
```

These are the LDT descriptors in tasks' LDT, they are almost the same as in GDT, except for the DPL field is set to 3.

I mentioned above, all tasks are linked as a single direction link structure, there are two important pointers, one is the `next` field in`TASK0`, it is the head of the whole link, and a `current` pointer is used for pointing to the current running task.

Creating and Scheduling

Before any tasks start, we define,

06/task.c

```
struct TASK_STRUCT *current = &TASK0;
```

Then we take a look at how a new task is generated,

06/init.c

```
static void

new_task(struct TASK_STRUCT *task, unsigned int eip,

                unsigned int stack0, unsigned int stack3) {
```

`new_task` takes four arguments, the first one is the `TASK_STRUCT` of new task, the second one is the start entry of the program, the `eip` field in TSS will be assigned to this value, so the processor can know where to start this program, the `esp0` and `esp3`arguments are for the `ESP` pointer in privilege level 0 and 3, because their selectors have fixed value, 0x10 an 0xf, ss0 and ss are not needed.

```
    memcpy(task, &TASK0, sizeof(struct TASK_STRUCT));

    task->tss.esp0 = stack0;

    task->tss.eip = eip;

    task->tss.eflags = 0x3202;

    task->tss.esp = stack3;



    task->priority = INITIAL_PRIO;



    task->state = TS_STOPPED;
```

`TASK0` are used as a template, we just modify some fields to suits our needs. We change new task state to `TS_STOPPED` before the task link is modified correctly. Otherwise, the link can be broken if another `new_task` is invoked at the same time.

```
    task->next = current->next;

    current->next = task;

    task->state = TS_RUNABLE;

}
```

Puts new task in task link.

```
extern void task1_run(void);

extern void task2_run(void);
```

They are the entries of new tasks, will be introduced in a short while.

```
static long task1_stack0[1024] = {0xf, };

static long task1_stack3[1024] = {0xf, };

static long task2_stack0[1024] = {0xf, };

static long task2_stack3[1024] = {0xf, };
```

Because we didn't have the memory management done in this tutorial, so we just set up several arrays as task stacks.

```
void

init(void) {
    char wheel[] = {'\\', '|', '/', '-'};
    int i = 0;
    struct TASK_STRUCT task1;
    struct TASK_STRUCT task2;

    idt_install();
    pic_install();
    kb_install();
    timer_install(1000);
    set_tss((unsigned long long)&TASK0.tss);
    set_ldt((unsigned long long)&TASK0.ldt);
```

Loads the `TASK0`'s LDT and TSS descriptors to GDT.

```
    __asm__ ("ltrw    %%ax\n\t"::"a"(TSS_SEL));
    __asm__ ("lldt    %%ax\n\t"::"a"(LDT_SEL));
```

Like you might guess, there are two new registers for TSS and LDT like GDT and IDT, called `TR` and `LDTR`, they are loaded by `ltr`and `lldt`. Before we trying to do any task manipulation, we have to load valid values to `TR` and `LDTR`, or you will get an general protection exception.

```
    sti();
    new_task(&task1,
            (unsigned int)task1_run,
            (unsigned int)task1_stack0+sizeof task1_stack0,
            (unsigned int)task1_stack3+sizeof task1_stack3);
    new_task(&task2,
            (unsigned int)task2_run,
            (unsigned int)task2_stack0+sizeof task2_stack0,
            (unsigned int)task2_stack3+sizeof task2_stack3);
```

Creates two new tasks, please note that, before creating any new tasks, the interrupt had been enabled.

```
    __asm__ ("movl %%esp,%%eax\n\t" \
             "pushl %%ecx\n\t" \
             "pushl %%eax\n\t" \
             "pushfl\n\t" \
             "pushl %%ebx\n\t" \
             "pushl $1f\n\t" \
             "iret\n" \
             "1:\tmovw %%cx,%%ds\n\t" \
             "movw %%cx,%%es\n\t" \
             "movw %%cx,%%fs\n\t" \
             "movw %%cx,%%gs" \
             ::"b"(USER_CODE_SEL),"c"(USER_DATA_SEL));
```

Now the kernel has became task `TASK0`, by a long return it loaded correct `EIP` and `CS` and `SS` and `ESP` from `TASK0` structure, the stack looks like,

```
+--------------------+

| LDT stack selector |

+--------------------+

|        ESP         |

+--------------------+

|       EFLAGS       |

+--------------------+

| LDT code selector  |

+--------------------+

|        EIP         |

+--------------------+
```

then we load data segment selectors from LDT.

```
    for (;;) {

        __asm__ ("movb    %%al,    0xb8000+160*24"::"a"(wheel[i]));

        if (i == sizeof wheel)

            i = 0;

        else

            ++i;

    }

}
```

Now all tasks are running at privilege level 3, so all other parts that have privilege level higher that that can not be accessed, it gives you some idea of protection. But in Skelix, I am not going to separate the kernel space from the whole memory, the memory protection will be achieved by the memory mapping of virtual memory. In later tutorial, the user tasks can invoke system functions via system calls.

Now we have three tasks, then which part will do the switching? Well, as you may guess, we have the timer interrupt in a certain interval, so it is the good and clearly choice. We have to change the code in timer interrupt like this,

06/timer.c

```
void do_timer(void) {

        struct TASK_STRUCT *v = &TASK0;

        int x, y;

        ++timer_ticks;

        get_cursor(&x, &y);

        set_cursor(71, 24);

        kprintf(KPL_DUMP, "%x", timer_ticks);

        set_cursor(x, y);

        outb(0x20, 0x20);

        cli();

        for (; v; v=v->next) {
```

Traversing the task link, to change priorities of all tasks.

```
                if (v->state == TS_RUNNING) {

                        if ((v->priority+=30) <= 0)

                                v->priority = 0xffffffff;
                } else
                        v->priority -= 10;
        }
```

Well, the priority of all tasks will be changed during timer interrupt, the running task get higher numerical priority, that is the lower priority. And the waiting tasks get higher priority.

```
        if (! (timer_ticks%1))
                scheduler();
```

We re-schedule all tasks in two timer ticking.

```
        sti();
}
```

Let's check out the `scheduler`,

```
void scheduler(void) {
        struct TASK_STRUCT *v = &TASK0, *tmp = 0;
        int cp = current->priority;

        for (; v; v = v->next) {
                if ((v->state==TS_RUNABLE) && (cp>v->priority)) {
                        tmp = v;
                        cp = v->priority;
                }
        }
```

Traversing the task link, to find the task which has the highest priority to run, that is the smallest number.

```
        if (tmp && (tmp!=current)) {
                current->tss_entry = get_tss();
                current->ldt_entry = get_ldt();
```

We save the TSS and LDT descriptors from GDT to `TASK_STRUCT`, because some state bits in those descriptors might have been modified by the processor.

```
                  tmp->tss_entry = set_tss((unsigned long long)((unsigned int)&tmp->tss));
                  tmp->ldt_entry = set_ldt((unsigned long long)((unsigned int)&tmp->ldt));
                current->state = TS_RUNABLE;
                tmp->state = TS_RUNNING;
                current = tmp;
```

Sets TSS and LDT descriptors of the task which is going to run to the GDT, and changes theirs `state` fields.

```
                __asm__ __volatile__("ljmp      $" TSS_SEL_STR ",       $0\n\t");
        }
}
```

Skelix uses a far jump instruction to perform the task switching, the segment part is the TSS selector, and the offset part is simply ignored by the processor. The `TSS_SEL_STR` is defined as "$0x18", the quotation marks are included, so C can take all those string fragments into one string.

A & B

At last, let's check out the code of task1_run and task2_run, their entries are written in assembly instead C, because I do not want to handle the stack change in C function calls. These two tasks actually call other C functions to do the real job.

```
task1_run:
        call    do_task1
        jmp        task1_run
task2_run:
        call    do_task2
        jmp        task2_run
```

At first, let's check if tasks work in privilege 3 properly, so we call kprintf to print the current CS that task is using, then let it go to busy idle or it will keep printing it on the screen, and screen scrolls up too fast to see any result.

06/init.c

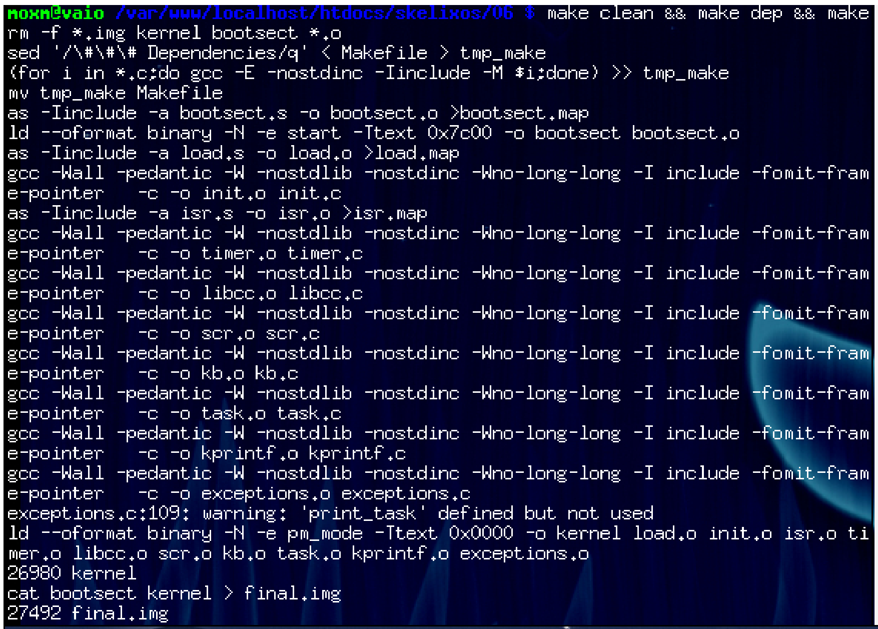
```
void
do_task1(void) {
    unsigned int cs;
    __asm__ ("movl    %%cs,    %%eax":"=a"(cs));
    kprintf(KPL_DUMP, "%x", cs);
    for (;;)
        ;
}

void
do_task2(void) {
    unsigned int cs;
    __asm__ ("movl    %%cs,    %%eax":"=a"(cs));
    kprintf(KPL_PANIC, "%x", cs);
    for (;;)
        ;
}
```

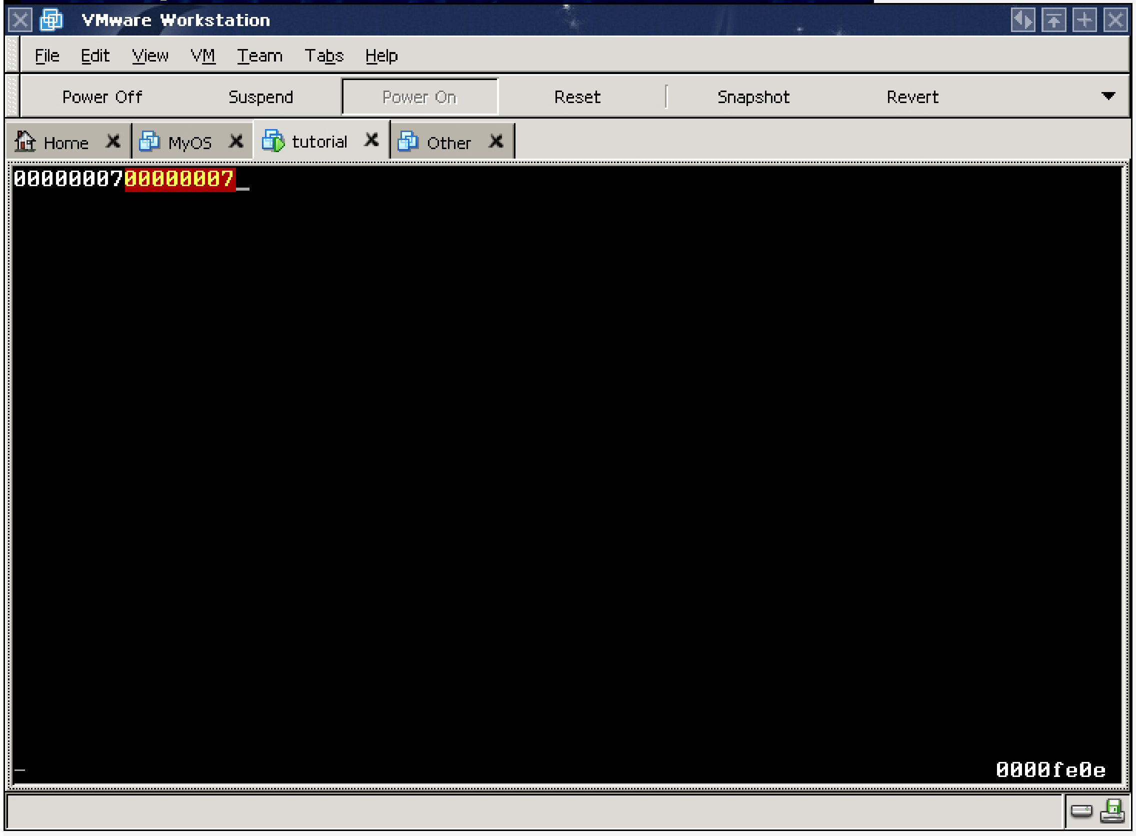
Don't forget to add new modules to KERNEL_OBJS in Makefile.

06/Makefile

```
KERNEL_OBJS= load.o init.o isr.o timer.o libcc.o scr.o kb.o task.o kprintf.o exceptions.o
```



```
moxn@vaio /var/www/localhost/htdocs/skelixos/06 $ make clean && make dep && make
rm -f *.img kernel bootsect *.o
sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
(for i in *.c;do gcc -E -nostdinc -Iinclude -M $i;done) >> tmp_make
mv tmp_make Makefile
as -Iinclude -a bootsect.s -o bootsect.o >bootsect.map
ld --oformat binary -N -e start -Ttext 0x7c00 -o bootsect bootsect.o
as -Iinclude -a load.s -o load.o >load.map
gcc -Wall -pedantic -W -nostdlib -nostdinc -Wno-long-long -I include -fomit-fram
e-pointer   -c -o init.o init.c
as -Iinclude -a isr.s -o isr.o >isr.map
gcc -Wall -pedantic -W -nostdlib -nostdinc -Wno-long-long -I include -fomit-fram
e-pointer   -c -o timer.o timer.c
gcc -Wall -pedantic -W -nostdlib -nostdinc -Wno-long-long -I include -fomit-fram
e-pointer   -c -o libcc.o libcc.c
gcc -Wall -pedantic -W -nostdlib -nostdinc -Wno-long-long -I include -fomit-fram
e-pointer   -c -o scr.o scr.c
gcc -Wall -pedantic -W -nostdlib -nostdinc -Wno-long-long -I include -fomit-fram
e-pointer   -c -o kb.o kb.c
gcc -Wall -pedantic -W -nostdlib -nostdinc -Wno-long-long -I include -fomit-fram
e-pointer   -c -o task.o task.c
gcc -Wall -pedantic -W -nostdlib -nostdinc -Wno-long-long -I include -fomit-fram
e-pointer   -c -o kprintf.o kprintf.c
gcc -Wall -pedantic -W -nostdlib -nostdinc -Wno-long-long -I include -fomit-fram
e-pointer   -c -o exceptions.o exceptions.c
exceptions.c:109: warning: 'print_task' defined but not used
ld --oformat binary -N -e pm_mode -Ttext 0x0000 -o kernel load.o init.o isr.o ti
mer.o libcc.o scr.o kb.o task.o kprintf.o exceptions.o
26980 kernel
cat bootsect kernel > final.img
27492 final.img
```


VMware Workstation
File Edit View VM Team Tabs Help
Power Off Suspend Power On Reset Snapshot Revert
Home MyOS tutorial Other

```
0000000700000007_

0000fe0e
```

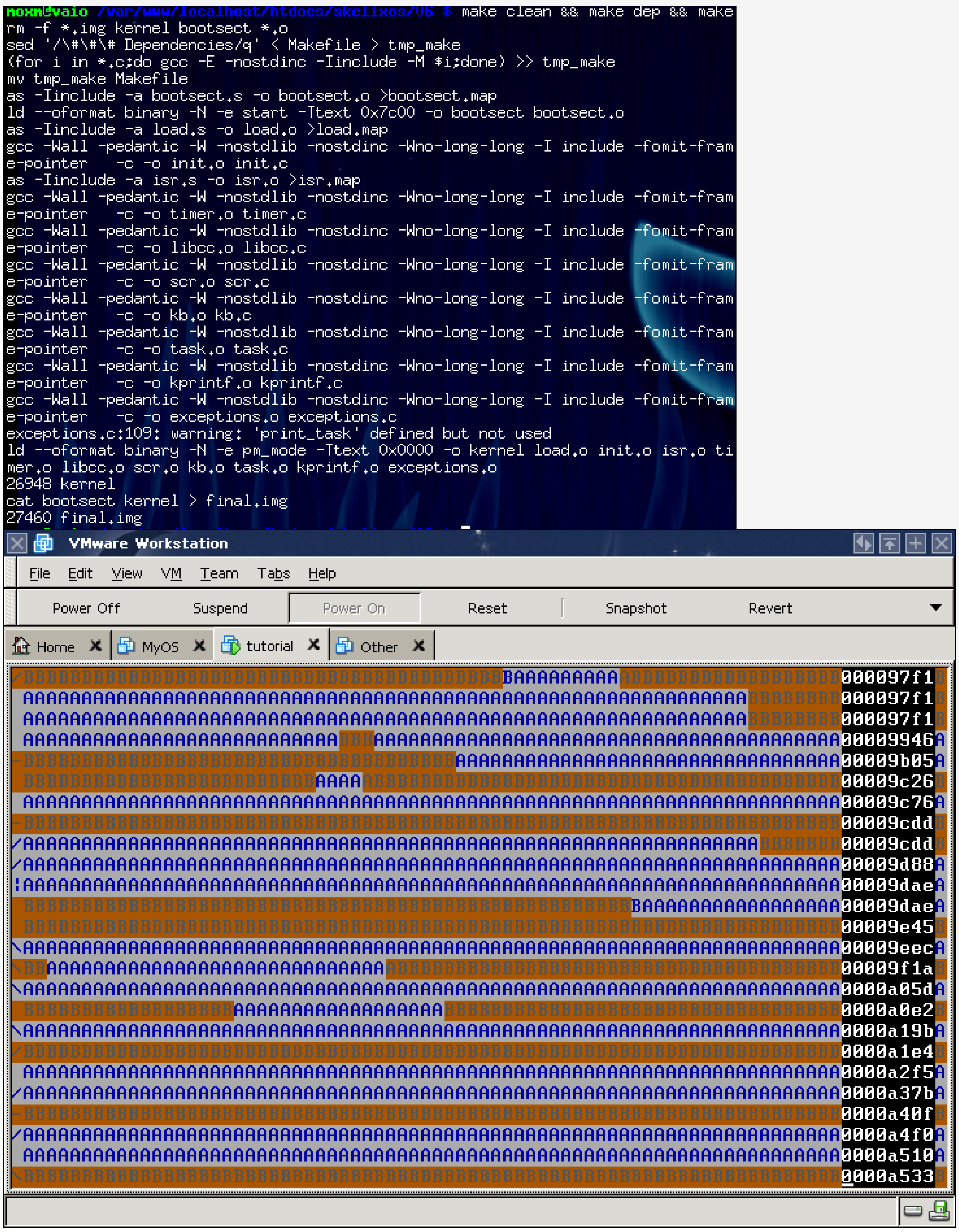
As we can see, first we are using LDT, and the `CS` has been set to 0x7 correctly, another thing is as we make the `kprintf` buffer global, so actually all tasks are sharing same buffer, so output of two tasks might be mixed together when another task starts printting before former task finishes.

Now let them do something fancier, we let two tasks keep displaying different letters,

06/init.c

```
void
do_task1(void) {
    print_c('A', BLUE, WHITE);
}

void
do_task2(void) {
    print_c('B', GRAY, BROWN);
}
```



```
moxn@vaio /var/www/localhost/htdocs/skelixos/06 $ make clean && make dep && make
rm -f *.img kernel bootsect *.o
sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
(for i in *.c;do gcc -E -nostdinc -Iinclude -M $i;done) >> tmp_make
mv tmp_make Makefile
as -Iinclude -a bootsect.s -o bootsect.o >bootsect.map
ld --oformat binary -N -e start -Ttext 0x7c00 -o bootsect bootsect.o
as -Iinclude -a load.s -o load.o >load.map
gcc -Wall -pedantic -W -nostdlib -nostdinc -Wno-long-long -I include -fomit-fram
e-pointer   -c -o init.o init.c
as -Iinclude -a isr.s -o isr.o >isr.map
gcc -Wall -pedantic -W -nostdlib -nostdinc -Wno-long-long -I include -fomit-fram
e-pointer   -c -o timer.o timer.c
gcc -Wall -pedantic -W -nostdlib -nostdinc -Wno-long-long -I include -fomit-fram
e-pointer   -c -o libcc.o libcc.c
gcc -Wall -pedantic -W -nostdlib -nostdinc -Wno-long-long -I include -fomit-fram
e-pointer   -c -o scr.o scr.c
gcc -Wall -pedantic -W -nostdlib -nostdinc -Wno-long-long -I include -fomit-fram
e-pointer   -c -o kb.o kb.c
gcc -Wall -pedantic -W -nostdlib -nostdinc -Wno-long-long -I include -fomit-fram
e-pointer   -c -o task.o task.c
gcc -Wall -pedantic -W -nostdlib -nostdinc -Wno-long-long -I include -fomit-fram
e-pointer   -c -o kprintf.o kprintf.c
gcc -Wall -pedantic -W -nostdlib -nostdinc -Wno-long-long -I include -fomit-fram
e-pointer   -c -o exceptions.o exceptions.c
exceptions.c:109: warning: 'print_task' defined but not used
ld --oformat binary -N -e pm_mode -Ttext 0x0000 -o kernel load.o init.o isr.o ti
mer.o libcc.o scr.o kb.o task.o kprintf.o exceptions.o
26948 kernel
cat bootsect kernel > final.img
27460 final.img
```

```
/BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAABBBBBBBBBBBBBBBBBBB000097f1B
 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBB000097f1B
 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBB000097f1B
 AAAAAAAAAAAAAAAAAAAAAAAAAAABBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA00009946A
-BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA00009b05A
 BBBBBBBBBBBBBBBBBBBBBBBBBAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB00009c26B
 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA00009c76A
-BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB00009cddB
/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBB00009cddB
/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA00009d88A
:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA00009daeA
 BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAA00009daeA
 BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB00009e45B
\AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA00009eecA
\BBAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB00009f1aB
\AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA0000a05dA
 BBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB0000a0e2B
\AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA0000a19bA
/BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB0000a1e4B
 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA0000a2f5A
/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA0000a37bA
-BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB0000a40fB
/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA0000a4f0A
 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA0000a510A
\BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB0000a533B
```

Prev	Tutorial 06: Multi-Tasking	Next