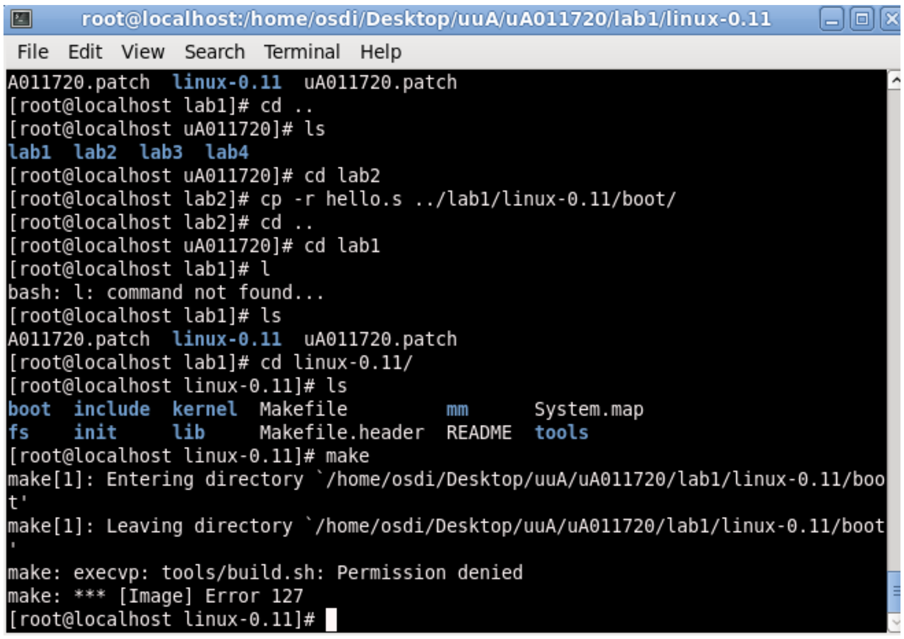
Operating System Design and Implement
OSDI 2015
Mid-Term Report

學號: A011720

姓名:李政廷

Lab 1. Establish Lab Environment


root@localhost:/home/osdi/Desktop/uuA/uA011720/lab1/linux-0.11
File Edit View Search Terminal Help

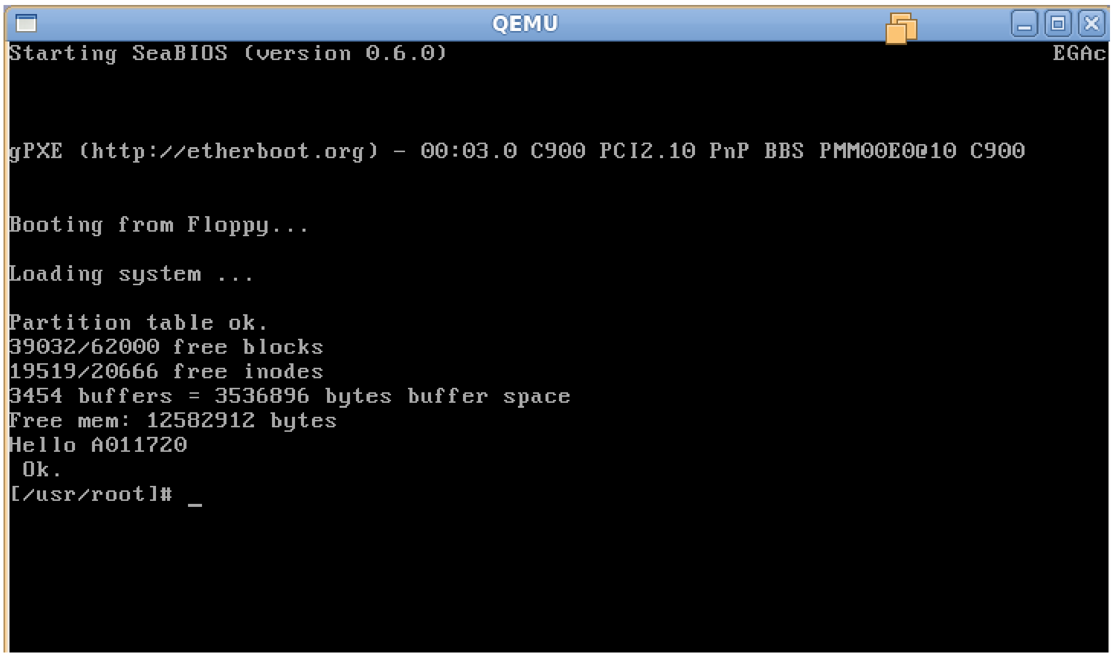
```
A011720.patch  linux-0.11  uA011720.patch
[root@localhost lab1]# cd ..
[root@localhost uA011720]# ls
lab1  lab2  lab3  lab4
[root@localhost uA011720]# cd lab2
[root@localhost lab2]# cp -r hello.s ../lab1/linux-0.11/boot/
[root@localhost lab2]# cd ..
[root@localhost uA011720]# cd lab1
[root@localhost lab1]# l
bash: l: command not found...
[root@localhost lab1]# ls
A011720.patch  linux-0.11  uA011720.patch
[root@localhost lab1]# cd linux-0.11/
[root@localhost linux-0.11]# ls
boot  include  kernel  Makefile         mm      System.map
fs    init     lib     Makefile.header  README  tools
[root@localhost linux-0.11]# make
make[1]: Entering directory `/home/osdi/Desktop/uuA/uA011720/lab1/linux-0.11/boo
t'
make[1]: Leaving directory `/home/osdi/Desktop/uuA/uA011720/lab1/linux-0.11/boot
'
make: execvp: tools/build.sh: Permission denied
make: *** [Image] Error 127
[root@localhost linux-0.11]# 
```

Before make file, we should have the privilege of the build.sh.
……]#chmod 777 tools/build.sh

After following the steps of hangout and the privilege is released, I could make the Image successfully.
Then I use the command “qemu -m 16M -boot a -fda Image -hda ../osdi.img” to boot my Image. Note: the osdi.img is received from teaching assistant, instead produced from our system.
As shown in below is the debugging flow, and the tool is GDB.
GDB command is “qemu -m 16M -boot a -fda Image -hda ../osdi.img -s -S -serial stdio”.
Note: If –s –S is not added in the command line, the debugger can not stop the system to debug.

1. First, the system is stuck in panic function, due to the uncompleted panic function. (After the panic function is commented, the system can run in next step)
2. Second, the system cannot run the init function, due to the false of if(!fork()).
3. So, we find there is no NR_TASKS of find_empty_process function in fork.c
4. Finally, we assign the number of NR_TASKS as 64, so as the system can boot normally.

Booting Successful

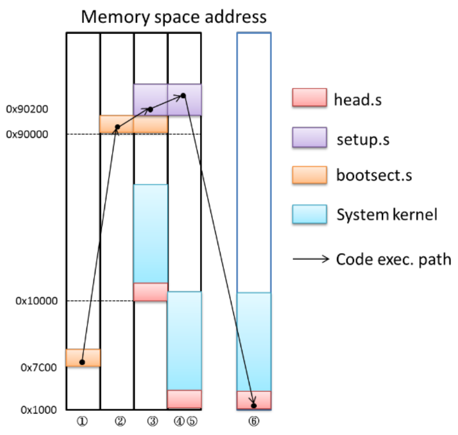


```
Starting SeaBIOS (version 0.6.0)                                                 EGAc


gPXE (http://etherboot.org) - 00:03.0 C900 PCI2.10 PnP BBS PMM00E0@10 C900


Booting from Floppy...

Loading system ...

Partition table ok.
39032/62000 free blocks
19519/20666 free inodes
3454 buffers = 3536896 bytes buffer space
Free mem: 12582912 bytes
Hello A011720
 Ok.
[/usr/root]# _
```

Conclusion:

In this lab, I have experience to use the GDB tool to debug the program. Before using the GDB tool, I have to know the program flow and guess which function will be executed after the system booting. Then I set the breakpoint in which function that I want to observe. After breakpoint is set, I press "c" to continue the program until it stops in the line of breakpoint. Then "n" will be pressed to observe the program that is executed in next step. The "l" will list the code that is running.

Lab 2: Kernel booting



1. BIOS load the bootsect from disk MBR to 0x7c00 memory address
2. boosect copy itself to 0x90000 memory address and jump to 0x90000.
3. boosect load setup from disk to 0x90200 memory address.
4. Get some system peripheral device parameters (video, root disk, keyboard,…,etc.) and jump to 0x90200.
5. Switch system into protected mode move kernel from 0x10000(64K) to 0x01000
6. Jump to 0x1000 and execute head.s for kernel boot

Before executing main function, the CPU must execute bootsect.s routine.
So the .equ INITSEG, 0x9000 command let the INITSEG be the reference address that jmp can jump to, if system boot.

Once the system power on, the command “ljmp $BOOTSEG, $_start” will jump to 0x07c0, then the system will call the command of “ljmp $INITSEG, $go” in _start.

Since we need to design the multi booting function, and let the user can select which one program, the selection method is implemented as below.

```
read_input:
    mov $0x0000, %ax
    int $0x16
    cmp $0x31, %al
    je   load_setup
    cmp $0x32, %al
    je   load_hello
    jmp read_input
```

Then we have to implement the “load_hello” program to execute.
Because loading the setup program relies on the INT 0x13 interrupt vector, which refers to the interrupt service routine in BIOS. So INT 0x13 is necessary to load our designed program, say, here is hello program.

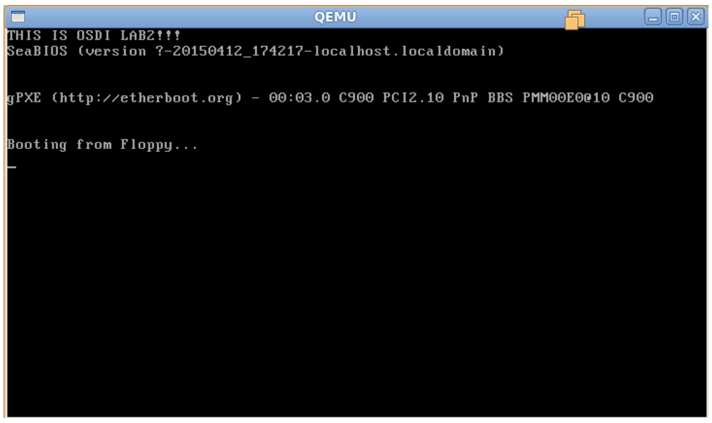
And the hello.s is placed in where we defined like below.

```
[ ! -f "$hello_img" ] && echo "there is no hello binary file there" && exit -1
dd if=$hello_img seek=1 bs=512 count=1 of=$IMAGE 2>&1 >/dev/null
```

AL = 1

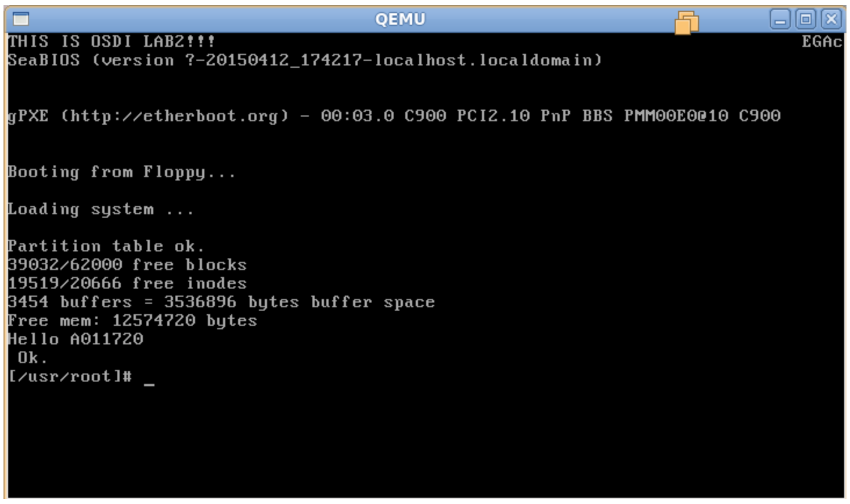
```
load_hello:
        mov $0x0000, %dx          # drive 0, head 0
        mov $0x0002, %cx          # sector 2, track 0
        mov $0x0200, %bx          # address = 512, in INITSEG 2*16**2
        .equ    AX, 0x0200+1
        mov     $AX, %ax          # service 2, nr of sectors
        int $0x13       # INT 13h AH=02h: Read Sectors From Drive, then clear flag
        jnc ok_hello              # ok – continue, since flag has been cleared
        mov $0x0000, %dx
        mov $0x0000, %ax          # reset the diskette
        int $0x13
        jmp load_hello
ok_hello:

    jmp  $SETUPSEG, $0
```

After setup the booting program, we can boot the system ad shown in below.



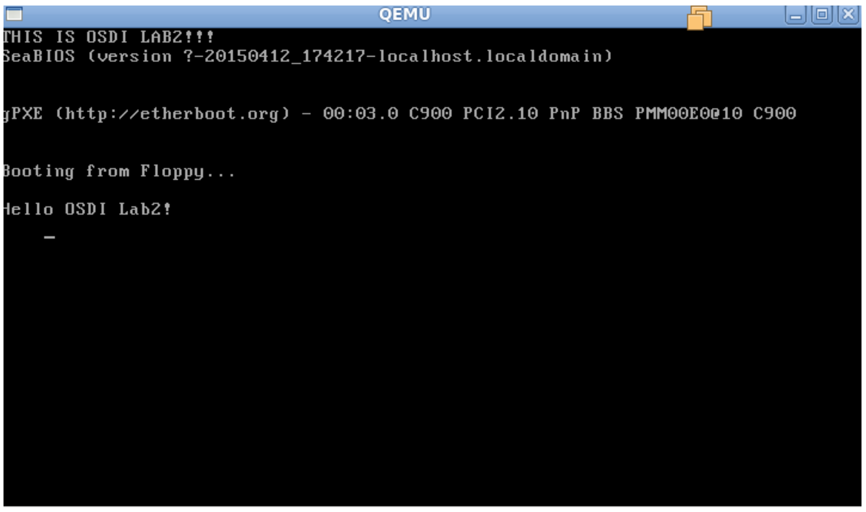
```
THIS IS OSDI LAB2!!!
SeaBIOS (version ?-20150412_174217-localhost.localdomain)


gPXE (http://etherboot.org) - 00:03.0 C900 PCI2.10 PnP BBS PMM00E0@10 C900


Booting from Floppy...
_
```

If we Press “1”



```
THIS IS OSDI LAB2!!!                                                         EGAc
SeaBIOS (version ?-20150412_174217-localhost.localdomain)


gPXE (http://etherboot.org) - 00:03.0 C900 PCI2.10 PnP BBS PMM00E0@10 C900


Booting from Floppy...

Loading system ...

Partition table ok.
39032/62000 free blocks
19519/20666 free inodes
3454 buffers = 3536896 bytes buffer space
Free mem: 12574720 bytes
Hello A011720
 Ok.
[/usr/root]# _
```

If we press “2”



```
THIS IS OSDI LAB2!!!
SeaBIOS (version ?-20150412_174217-localhost.localdomain)


gPXE (http://etherboot.org) - 00:03.0 C900 PCI2.10 PnP BBS PMM00E0@10 C900


Booting from Floppy...

Hello OSDI Lab2!
    _
```

Conclusion:

In this lab, I learned how to run the program that I designed and called in bootsect.s. Since we want to make the system support more than one OS sometimes, the multi booting offers more than one choice for user.

Lab 3: X86 I/O System and Interrupt

In order to build my own OS, I need to offer some basic I/O system to user.
Here, we offer two fundamental interrupts to build the simple system.
One is keyboard interrupt, the other is timer.

First, we have to add two interrupt handler that is shown in below in the trap_entry.s.

```
TRAPHANDLER_NOEC(handler_kbd, IRQ_OFFSET+IRQ_KBD);
TRAPHANDLER_NOEC(handler_timer, IRQ_OFFSET+IRQ_TIMER);
```

Then register the handler in the trap.c.

```
extern void handler_kbd();
extern void handler_timer();
SETGATE(idt[IRQ_OFFSET+IRQ_KBD ],0,GD_KT,handler_kbd,0);
SETGATE(idt[IRQ_OFFSET+IRQ_TIMER],0,GD_KT,handler_timer,0);
```

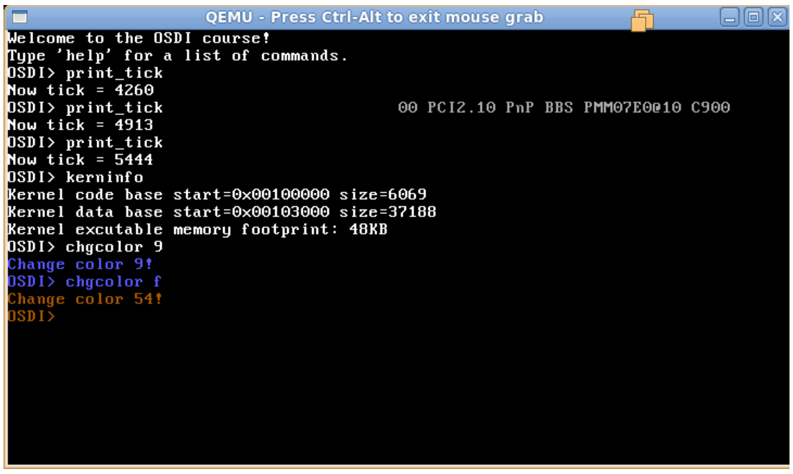
And we have to write a switch to select which interrupt is enabled by following the comment.

```
    /* Lab3: Check the trap number and call the interrupt handler. */
    switch (tf->tf_trapno){
        case IRQ_OFFSET+IRQ_KBD:
                 kbd_intr();
                     break;
        case IRQ_OFFSET+IRQ_TIMER:
                 timer_handler();
                     break;
    }
```

The most important of all, we need to push some information into stack such as program counter (CS:EIP), stack point(SS:ESP) and error flag (EFLAGS) before trap. And the code is shown in below.

```
pushal;          #Push EAX, ECX, EDX, EBX
# override es and ds with GD_KD
push    %ds;
push    %es;
mov $GD_KD, %ax;        #load Global Descriptor Table into %ax
mov %ax, %ds;
mov %ax, %es;
push %esp; # pointer to trapframe
call default_trap_handler;
add $4, %eax; # pop the trapframe pointer
popl %es; # pop es
popl %ds; # pop ds
popal
```

Then the result is shown in below.
The change text color is produced by employ the function of "int chg_color(int argc, char **argv);"



```
Welcome to the OSDI course!
Type 'help' for a list of commands.
OSDI> print_tick
Now tick = 4260
OSDI> print_tick                        00 PCI2.10 PnP BBS PMM07E0@10 C900
Now tick = 4913
OSDI> print_tick
Now tick = 5444
OSDI> kerninfo
Kernel code base start=0x00100000 size=6069
Kernel data base start=0x00103000 size=37188
Kernel excutable memory footprint: 48KB
OSDI> chgcolor 9
Change color 9!
OSDI> chgcolor f
Change color 54!
OSDI>
```

Conclusion:

In this lab, I learned how to register the interrupt to let the user can input the information by keyboard. And store the system state, before the trap is used.

Lab 4: Minimal Kernel

In order to let our OS can be more complete, and we want to make the our OS can deal numbers of task and improve the throughput of the system.

First of all, we need to finish the code of the task_create function in task.c.
And the code is shown in below

```
    ts->task_id = i;
    ts->state = TASK_RUNNABLE;    #Once the new task is created, it can not be
    execute immediately, so it is runnable
    if(cur_task) ts->parent_id = cur_task->task_id;
    else cur_task->task_id = 0;     #if the task is parent, then return the value of 0
    ts->remind_ticks = 100;
```

Then finish the sleep() and kill_self()
sleep()

```
        cur_task->state=TASK_SLEEP;
        cur_task->remind_ticks=a1;   #a1 is related to edx that is shown in below
        sched_yield();
```

```
    asm volatile("int %1\n"
        : "=a" (ret)
        : "i" (T_SYSCALL),
          "a" (num),
          "d" (a1),
          "c" (a2),
          "b" (a3),
          "D" (a4),
          "S" (a5)
        : "cc", "memory");
```

kill_self()

```
    case SYS_kill:
        /* Lab4 TODO: kill task. */
        sys_kill(a1);
        retVal = 0;
```

need to return the task of trapframe value of state in syscall_handler

```
ret = do_syscall(tf->tf_regs.reg_eax, tf->tf_regs.reg_edx, tf->tf_regs.reg_ecx,
tf->tf_regs.reg_ebx, tf->tf_regs.reg_edi,tf->tf_regs.reg_esi);
```

```
    /* Lab4 TODO: Register system call's trap handler (syscall_handler)*/
    register_handler(T_SYSCALL,syscall_handler,i_syscall, 1,3);
```

Since every time that task has limited executed time ticks, we have count the time tick to let every tasks has chance that is executed by cpu.

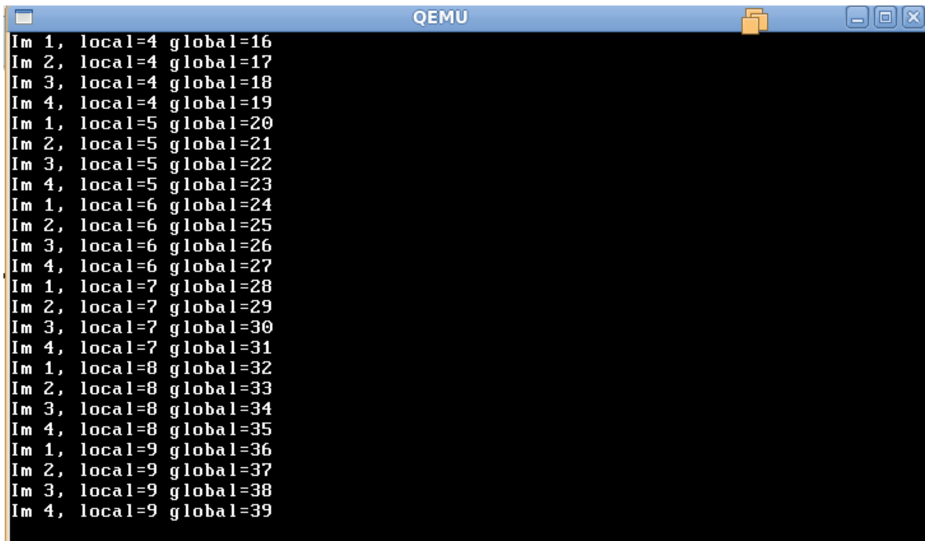
```
    if (cur_task != NULL)
    {
        /* Lab4 TODO: Check if tasks need wakeup.    */

        for(i=0;i<NR_TASKS;i++){
            if(tasks[i].state==TASK_SLEEP){
                tasks[i].remind_ticks--;
                if(tasks[i].remind_ticks<=0){
                    tasks[i].state = TASK_RUNNABLE;
                }
            }
        }
        /* Lab4 TODO: Check cur_task->remind_ticks, if remind_ticks <= 0 then
yield the task (call sched_yield() in sched.c)*/

        if(--cur_task->remind_ticks<=0)
                sched_yield();

    }
```

Finally, we use the round-robin to schedule the task.

```
    for(i=0;i<NR_TASKS;i++)
    {
        next_i = (i+cur_task->task_id)%NR_TASKS;
        if(tasks[next_i].state==TASK_RUNNABLE){
            tasks[next_i].remind_ticks = 100;
            tasks[next_i].state=TASK_RUNNING;

```

```
                    if(cur_task->state=TASK_RUNNING)
                            cur_task->state=TASK_RUNNABLE;
                            cur_task = &tasks[next_i];
            env_pop_tf(&tasks[next_i].tf);

            }
    }
```

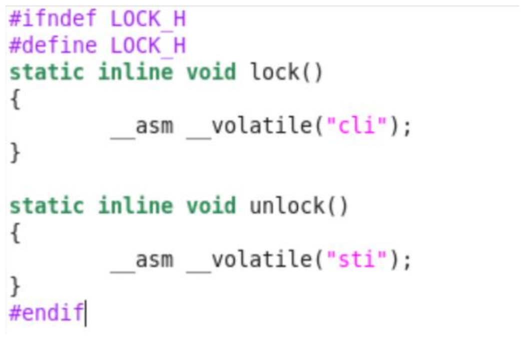
Then the result is shown in below.



```
Im 1,  local=4 global=16
Im 2,  local=4 global=17
Im 3,  local=4 global=18
Im 4,  local=4 global=19
Im 1,  local=5 global=20
Im 2,  local=5 global=21
Im 3,  local=5 global=22
Im 4,  local=5 global=23
Im 1,  local=6 global=24
Im 2,  local=6 global=25
Im 3,  local=6 global=26
Im 4,  local=6 global=27
Im 1,  local=7 global=28
Im 2,  local=7 global=29
Im 3,  local=7 global=30
Im 4,  local=7 global=31
Im 1,  local=8 global=32
Im 2,  local=8 global=33
Im 3,  local=8 global=34
Im 4,  local=8 global=35
Im 1,  local=9 global=36
Im 2,  local=9 global=37
Im 3,  local=9 global=38
Im 4,  local=9 global=39
```

Lab 5: Kernel Lock

In order to produce functions of lock() and unlock(), we need to define it by using the sti and cli that are from assembly code.
And the defined is shown in below.



```
#ifndef LOCK_H
#define LOCK_H
static inline void lock()
{
        __asm __volatile("cli");
}

static inline void unlock()
{
        __asm __volatile("sti");
}
#endif
```

In this lab, we can show that lock & unlock will protect the shared variable to be avoid polluting by others.
So, once we call the shared variable(cur_task->tf = *tf and tf = &(cur_task->tf);), we have to protect it by using lock(), then unlock() it, after using the shared variable.
Like below.

```
    lock();
    for(i=0;i<NR_TASKS;i++)
    {
        next_i = (i+cur_task->task_id)%NR_TASKS;
        if(tasks[next_i].state==TASK_RUNNABLE){
            tasks[next_i].remind_ticks = TIME_QUANT;
            tasks[next_i].state=TASK_RUNNING;

            if(cur_task->state==TASK_RUNNING)
                cur_task->state=TASK_RUNNABLE;
                cur_task = &tasks[next_i];
        env_pop_tf(&tasks[next_i].tf);

        }
    }
    unlock();
```

2. Questions

2.1. About the locker

Why we don’t need to unlock() before env_pop_tf() and the OS still works fine?

Ans: env_pop_tf() also can enable interrupts by setting the IF flag

2.2. Lock all kernel section

Uncomment (1) and check whether it still works fine.

Ans: It can works fine, because it will protect the shared variable(cur_task->tf = *tf; and tf = &(cur_task->tf);) to avoid changing by others.

2.3. Lock partial kernel section

Uncomment (2) lock, and comment (1) lock.

Ans: It cannot work fine, because it will not protect the shared ariable(cur_task->tf = *tf; and tf = &(cur_task->tf);) to avoid changing by others.

2.4. Lock shared kernel variable

In OS kernel, there are many variables which are shared by tasks (ex. Tasks[], *cur_task, etc). Due to the unpredictable interrupt/trap happens, these share data may be polluted. In this part, you need to find up the potential share data and add lock mechanism to ensure them are well protected. Besides, critical section will slow down our kernel, please modify your kernel code (system call and scheduler) to use as less critical section as possible.

Ans: lock all the potential used variable, say , in the “syscall.c”, “timer.c”, and “sched.c”