# LAB5 Report 0116320 陳威伯

2.1. About the locker Why we don’t need to unlock() before env\_pop\_tf() and the OS still works fine?

Because env\_pop\_tf() will retain the original EFLAGS when it execute IRET.  
STI command will set IF to 0. Howerver, env\_pop\_tf() recover IF value to 1 when execute IRET

2.2. Lock all kernel section Uncomment (1) and check whether it still works fine. If it works fine, please explain why it works. If not, please debug it by gdb, and explain why it crashes.

No, it doesn’t work.  
Without lock, system call may be interrupted.  
So, it only print out former children, later system call is interrupted and cancelled.

2.3. Lock partial kernel section Uncomment (2) lock, and comment (1) lock. Check the new result and explain why it behaves like that.   
It failed.  
Although (2) lock protect env\_pop\_tf(), but (2) lock doesn’t contain the system call entry, it acted like unlocked.

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. Hint: Each tasks shared same kernel stack, add lock before use this stack.

Lock the share variable in the following files   
systemcall.c

sys\_sleep

task.c

sys\_fork

sched.c

sched\_yield

And we have to check whether the interrupt is from user space or kernel space, modify the timer\_handler

If current timer interrupt is from kernel, it means the system call is interrupted by timer, we don’t call sched\_yield

if((tf->tf\_cs & 3) == 3)

sched\_yield();

else

env\_pop\_tf(tf);  
LAB1:

learn the basic usage of gdb, and the structure of the kernel code

get accustomed to kernel debugging environment  
LAB2:

Learn the boot procedure, what the boot loader do, and multi-boot implement  
LAB3:

Learn how to implement the keyboard and timer interrupt

Learn linker script knowledge to implement the kerninfo command  
LAB4:

Learn register the system call, and the details about system call register  
LAB5:

Learn how to lock critical section, and if system call is interrupted by other interrupt, how to deal with it.

Suggestion:

我認為關於每次LAB的說明可以更詳細一點, 不然對於第一次接觸kernel code的人常常會不知道從哪下手

其餘我認為每次的LAB都還蠻滿意的,覺得每次都學習到很多 m(\_ \_)m  
謝謝助教