# Lab3 实验报告


- 于凡奇 18307130182


## 1 习题一

请简要描述一下在你实现的操作系统中，中断时 CPU 进行了哪些操作。

在EL0发生中断后，CPU硬件：

- 将当前PC值存入ELR_EL1
- 将中断前的PSTATE存入SPSR_EL1
- 将中断原因信息存入ESR_EL1
- 将PSTATE中的DAIF全部置位，屏蔽所有中断
- 将栈指针从SP_EL0换为SP_EL1，并将SPSEL_EL1置位
- 根据某些状态寄存器判断中断类型以及是否需要切换Exception Level
- 根据对应的中断向量表陷入到对应的中断处理（汇编）代码中

CPU执行操作系统定义的中断处理代码：

- 将x0~x31和ELR_EL1, SPSR_EL1等寄存器保存至trapframe中
- 跳转至真正的中断处理程序（C语言编写）
- 识别中断产生原因，处理中断，解除中断状态
- 返回至汇编中断处理程序
- 将trapframe恢复至各寄存器中
- eret: 恢复PC和PSTATE寄存器的值

中断处理完成，处理器回到产生中断的位置恢复执行。

## 2 习题二

请在 `inc/trap.h` 中设计你自己的 trap frame，并简要说明为什么这么设计。

我在trapframe中保存了x0~x30全部31个数字寄存器，加上spsr_el1, elr_el1和sp寄存器。

其中x0~x15都是caller-saved寄存器，在调用trap前显然需要保存。其余数字寄存器中固然有些是callee-saved，但在中断的语境下却不能保证其能够被正确保存与恢复，因此最好还是将他们全部保存为妙，这样还可以方便中断处理程序判断中断前的情况。

原本需要保存sp_el0寄存器，但我们的内核实际上一直在使用sp_el1，也就是中断进入内核态后的sp，再加上sp_el1不能直接在el1访问，此处只能保存sp寄存器。这一行为也就是保存了中断前的栈指针，方便中断程序判断情况。

若在中断处理程序中再次触发异常或中断，则spsr_el1和elr_el1会存入新的异常状态值和返回地址，因此需要在进入trap前将它们提前保存。同时，中断处理程序也可能需要用到它们的值，保存在trapframe中更方便使用。

# 3 习题三

请补全 `kern/trapasm.S` 中的代码，完成 trap frame 的构建、恢复。

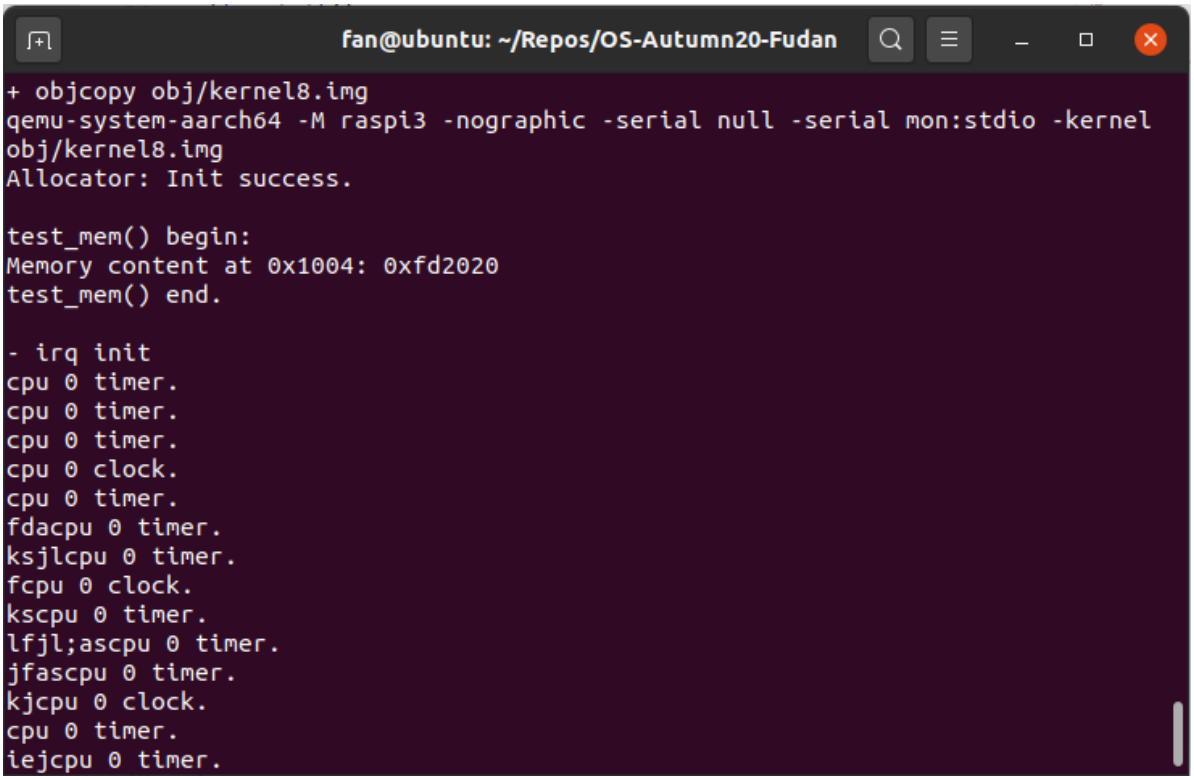
```
/* Build trap frame */
sub     sp, sp, #16
str     x30, [sp]
mov     x30, sp
str     x30, [sp, #8]

stp     x28, x29, [sp, #-16]!
...
stp     x0, x1, [sp, #-16]!

mrs     x10, elr_el1
mrs     x9, spsr_el1
stp     x9, x10, [sp, #-16]!

/* Call trap(struct *trapframe) */
mov     x0, sp
bl      trap
```

构建trap frame时，只要把需保存的寄存器依次压栈即可，注意顺序与struct trapframe中的定义一致。注意sp, el_er1和spsr_el1这些非通用寄存器不允许直接访存，需要通过其他通用寄存器中转。压栈后trap frame整个位于栈顶，故只需将sp作为参数传给trap函数。跳转时需要使用bl命令，使用b命令则处理器无法从trap函数返回。

恢复trap frame的操作与构建时完全类似只是存取顺序相反，故这里省略了具体代码。有一点不同是最后需要eret。

# 4 测试中断

按照lab3.md中的说明手动在内核开启中断后，运行make qemu有如下结果：




此次提交的版本也开启了中断。