问题一：开关中断和 eret 的问题

很多同学在进入例外和退出例外时，对 CP0_STATUS 寄存器的设置不当，可能导致在中断返回指令（eret）之前就打开了中断，进而导致 eret 返回前就触发了其他中断，从而产生卡死或者其他错误。

《龙芯 2F 处理器用户手册》中对于 CP0_STATUS 寄存器已经有了详细的阐述，但有些同学可能对于一些内容存疑，因此我们从同学们编程中出现的一些问题出发，详细的说明一下 CP0_STATUS 寄存器的相关内容，以及在编程中需要注意的问题。

下图是 CP0_STATUS 寄存器的格式：

31 28	27	26	25	24	23	22	21	20	19 16	15 8	7 5	4 3	2	1	0
CU (cu3:cu0)	0	FR	0	NO-FDIV	NO-FSQR	BEV	0	SR	0	IM7-IM0	0	KSU	ERL	EXL	IE
4	1	1	1	1	1	1	1	1	4	8	3	2	1	1	1

图 5-11 Status 寄存器

对于例外的控制，主要是通过后 3 位进行的，它们分别是错误级（ERL）、例外级（EXL）、中断级（IE）。它们都占一位（1b），具体的功能阐述如下：

错误级（ERL）：当发生复位，软件复位，NMI 或 Cache 错误时处理器将重置此位。这一位通常情况下都是 0，只有发生了上述情况时，才会变成 1，但是在 Project2 中正常情况下是不会涉及上述例外的，因此，在本次实验这一位永远是 0，如果触发了该位置 1，则说明你的代码出现了问题。

例外级（EXL）：当一个不是由复位，软件复位或 Cache 错误引发的例外产生时，处理器将设置该位为 1。可以看出，当发生一个不会导致 ERL 置位的例外时，EXL 就会被置为 1，EXL 置 1 后相关的例外就不会触发了，EXL 置 0 相关例外才会触发。**产生例外时，EXL 会自动置 1，这是由处理器硬件完成的，不需要我们进行处理，例外返回时，EXL 的置 0 也是通过 eret 指令自动完成置 0 的，不需要我们手动的写寄存器这一位。**

注意 1：中断、系统调用的触发，都和这一位有关系，因此想要中断和系统调用触发正常，要确保这一位是 0。

注意 2：同学们在开关中断（CLI、STI）时，不要对这一位进行处理。例外触发时处理器会自动将这一位置为 1，这就相当于关闭了例外的响应，而在例外返回时，这一位的置 0 是通过 eret 指令自动完成的，不用通过手工写寄存器完成。

中断使能（IE）：这一位是用于中断处理的，当这一位置为 0 时，会禁用所有外部中断，当这一位置为 1 时，使能所有外部中断。比如时钟中断就是外部中断的一种，因此我们如果想响应时钟中断，务必要让这一位为 1。

那么有同学可能会问，EXL 不是就包含了中断的情况么？为什么还需要一位 IE 去控制呢？实际上，例外和中断是一个包含和被包含的关系，因此如果 EXL 置为 1，那么即使 IE 为 1，中断也是无法触发的。而如果 EXL 为 0，IE 为 0，那么时钟中断也是无法触发的。只有当 EXL 为 0，IE 为 1，此时时钟中断才会被正常的触发。

对于 IE 位，是需要我们进行手动控制的，因此我们需要在进入例外处理时，将这一位置为 0，退出例外时，将这一位置为 1。虽然刚才说了，进入例外时处理器已经将 EXL 自动置为 1 了，即使不置 IE 为 0 理论上也不会有问题。但是，本次实验要求大家对 IE 位进行置位处理。

对于 ERL、EXL、IE 的处理，为了让大家更好的理解，我们做了个图 1 供大家参考：

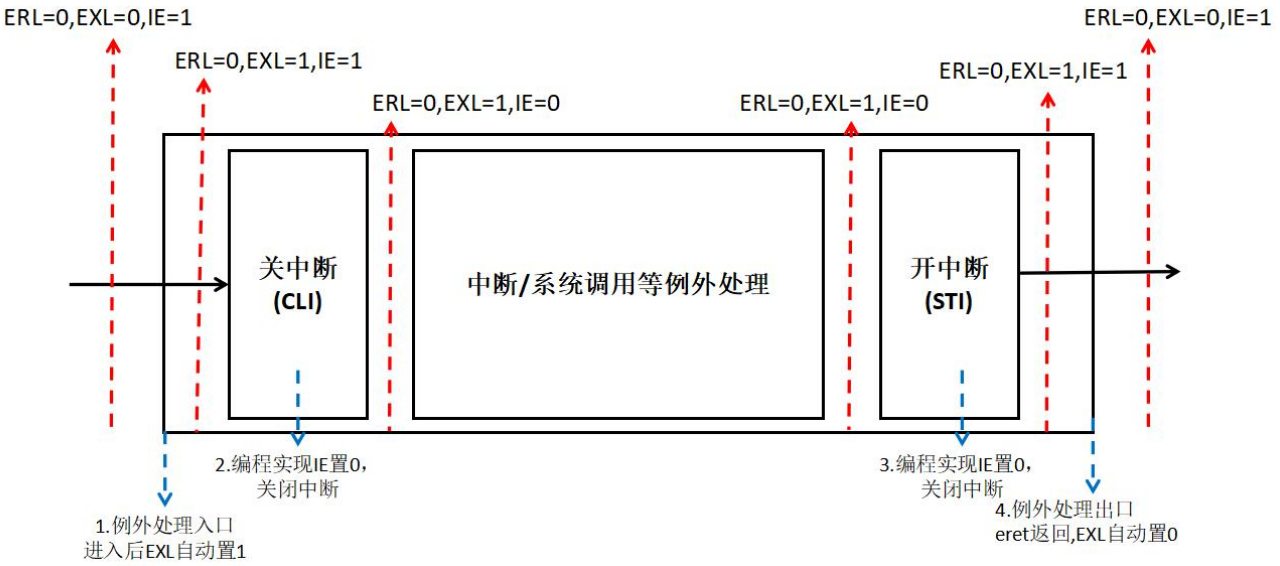

图 1 例外、中断开关与相应寄存器置位关系

问题二：屏幕出现满屏乱码

在任务 3 的时候，打印时出现屏幕乱码的问题可能是 init_screen 的时候没有对 screen 的 buffer 进行初始化，在 qemu 里这个 buff 也可能里面会有一些奇怪的内容，因此，需要在 init_screen 里调用 screen_clear 方法，可以解决这个问题，如图 2 所示。

```
void init_screen(void)
{
    vt100_hidden_cursor();
    vt100_clear();
    screen_clear();
}
```

图 2 init_screen 函数

问题三：用户态和内核态的上下文保存

在和同学们的交流以及基于 piazza 上的提问，我们发现很多同学对“进程从用户态到内核态进行任务切换，从内核态恢复再切换到用户态”这个过程还是比较模糊，以及有些同学在写代码的时候设计不合理，虽然可以打印出正确结果，但是这是由于我们的测试用例并不复杂，如果在后续任务中涉及到比较复杂任务时，还可能会导致问题。因此在这里我们给大家梳理一下。

对于一个任务 A，切换到任务 B 时，它的执行逻辑应该如图 3 所示：

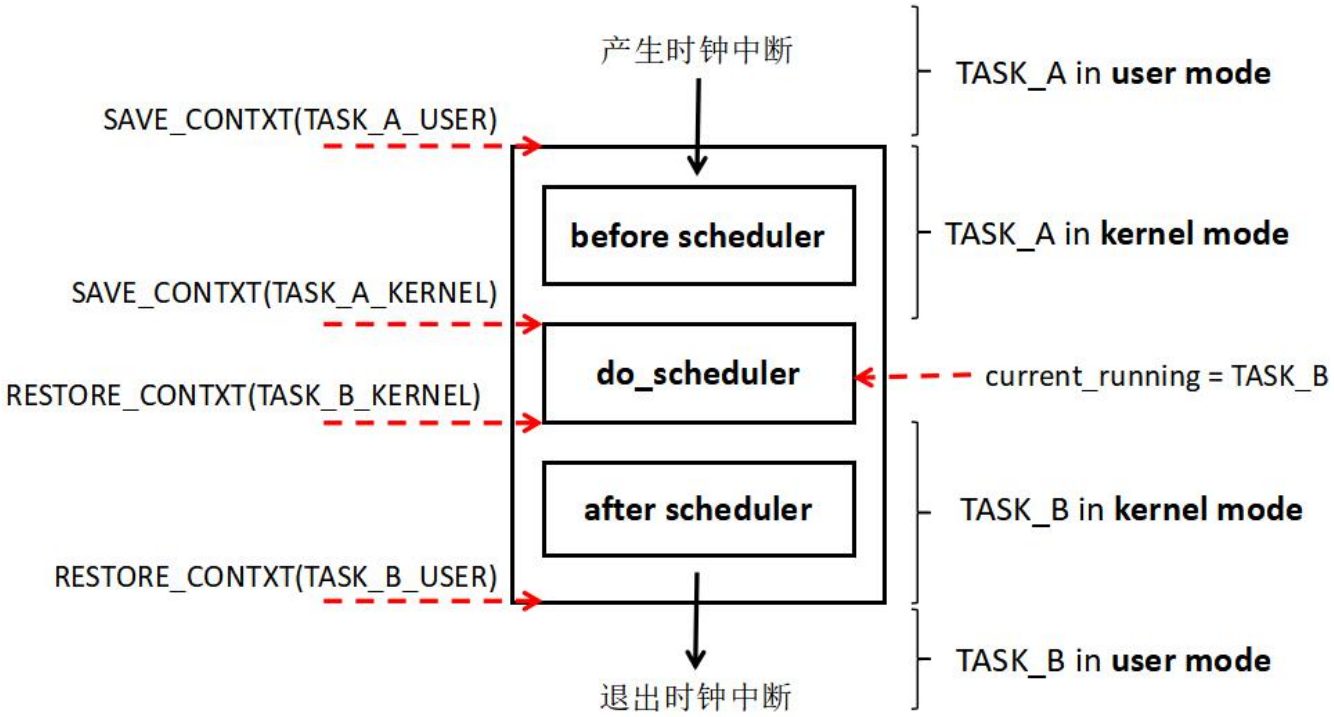

图 3 任务的 user context 和 kernel context

我们代码框架中给定的 PCB 有 kernel_context，user_context。这两个变量大家在进行时钟调度时，都是需要用到的。根据图 3 的逻辑：

（1）当任务 A 从用户态切换到内核态的时候，需要将用户态的现场保存在 user_context 中，也就是 SAVE_CONTEXT（TASK_A_USER）此时 current_running 依然是任务 A。

（2）在进行 do_scheduler 之前会进行多级例外处理，屏幕刷新或者其他的方法执行。直到执行到 do_scheduler，此时我们需要保存任务 A 的内核态现场，将此时寄存器的值保存在任务 A 的 kernel_context 中，也就是 SAVE_CONTEXT（TASK_A_KERNEL）然后进行调度。调度结束后，进行下一个任务 B 的内核态现场的恢复，也就是 RESTORE_CONTEXT（TASK_B_KERNLE），则切换到了任务 B 的内核态现场，此时 current_running 是任务 B。

（3）切换到了任务 B 的内核态之后，会继续执行任务 B 上次进入时钟中断被切走的内核态现场，直到恢复任务 B 的用户态现场，也就是 RESTORE_CONTEXT（TASK_B_USER），然后通过 eret 进入到任务 B 的用户态，至此，一次抢占式调度完成。

注意：很多同学没有使用 kernel_context 进行内核态现场的保存，而是在进行 do_scheduler 完成后，直接开中断，然后通过 ra 寄存器跳到了用户态，这种设计是不合理的。本着“从哪里进入，从哪里出去”的原则，任务在内核态被切换，就应该在内核态被还原，然后再恢复到用户态现场，最终通过 eret 跳回用户态。

那么，一个任务第一次的调度肯定不是从用户态通过中断进来的，又是怎么启动第一个任务的？关于这方面的实现，我们建议大家**认真考虑 init pcb 时对 kernel_context 相关寄存器初始值的设置，使其可以造成一种“我是从用户态进来的假象”**，从而进行调度时，依然可以通过 scheduler_after 此处的 RESTORE_CONTEXT 调回到用户态，开始第一次运行。

问题四：关于一些变量的初始化

不论大家用 qemu 还是开发板进行调试时，对于每个用到的变量，大家都要记得初始化，特别是在 qemu 里，如果不进行初始化，那么很可能这个变量的值并不会像平时编程那样，里面都是 0，而很有可能出现里面是奇怪的数的问题。

此外，有的同学在 createimage 时，只加载了代码段（text segment），而没有加载数据段（data segment），也会造成变量初始化值的缺失，因为变量的初始化值是保存在数据段内的。如果有出现变量初始值和预期不一样的问题时，同学可以再认真检查一下自己 createimage 编写时是否没有注意上述数据段的加载。

问题五：是否为进行初始化的内核线程分配一个 PCB

在 PCB 初始化时，有部分同学只对我们给出的测试任务进行了初始化，对进行初始化的内核线程没有分配 PCB，对于本次实验而言，这么做的确不会出错，但是，建议大家在实现里为进行初始化的内核线程分配一个 PCB（即 PID=0 的线程）。这样，即使没有测试任务，调度队列里仍然有一个内核线程在空转。

这么考虑的主要原因是在本次实验我们给出的测试用例都是 while(1)运行的，不会退出，但是在后续 project 中，我们需要实现任务的 exit 和 kill，此时可能出现我们所有的测试用例都退出的情况。如果这时调度队列中没有任务，那么就会出现问题。实际上在成熟的操作系统中，总会有这么一个内核线程，例如 Linux 系统中的 idle 进程

最后想说的话

本次实验想实现最终结果的正确打印的确不难，难就难在设计是否合理，很多同学即使设计不合理，最终也可以实现结果的正确输出。前面也说过，本次实验因为测试用例比较简单，因此即使大家的代码有问题，设计不合理，也不一定马上会产生问题。但在后续 Project 中，我们会实现更多的内核功能，比如同步原语、驱动、文件系统等。如果 P2 设计的不合理，那么很有可能对大家后续工作产生一定的影响，出现大家所说的“玄学 bug”。所以我们希望大家能够积极和助教老师们沟通，把这次任务做好，为后续的实验打下坚实的基础，**拒绝“玄学 bug”**。