《操作系统》

实验五报告

目录

1 实验代码分析

1.1 实验目的

本实验和实验四的目的在于深刻理解中断的原理和机制，掌握 CPU 访问中断控制器的方法，掌握 Arm 体系结构的中断机制和规范，实现时钟中断服务和部分异常处理等。

1.2 了解 Arm 的中断系统

补充知识：

1.2.1 快速中断 FIQ 与普通中断 IRQ

上一个实验中简单提及过，这里进行一个详细介绍它们之间的区别。

1.优先级和处理速度：快速中断通常用于处理**紧急且时间敏感**的中断请求，如*硬件计数器溢出*、*实时数据传输* 等。在某些架构如 ARM 中，FIQ 提供更多的寄存器来保存状态信息，使得中断处理过程更快，减少上下文切换的时间。而普通中断则用于处理**相对不那么紧急或者非实时性的**中断请求，如*键盘输入*、*网络数据包接收* 等。其优先级低于快中断，多个 IRQ 可以排队等待处理，不会打断正在进行的 FIQ 处理。

2.用途和场景：快速中断主要用于处理对时间要求比较紧急的中断请求，如高速数据传输及通道处理中。而普通中断则更多地用于通用中断处理，在硬件产生中断信号后自动进入。

3.寄存器使用：快中断有许多（R8~R14）自己的专用寄存器，发生中断时，使用自己的寄存器就避免了保存和恢复某些寄存器。如果异常中断处理程序中使用它自己的物理寄存器之外的其他寄存器，异常中断处理程序必须保存和恢复这些寄存器。而普通中断在处理过程中可能需要保存和恢复更多的寄存器状态。

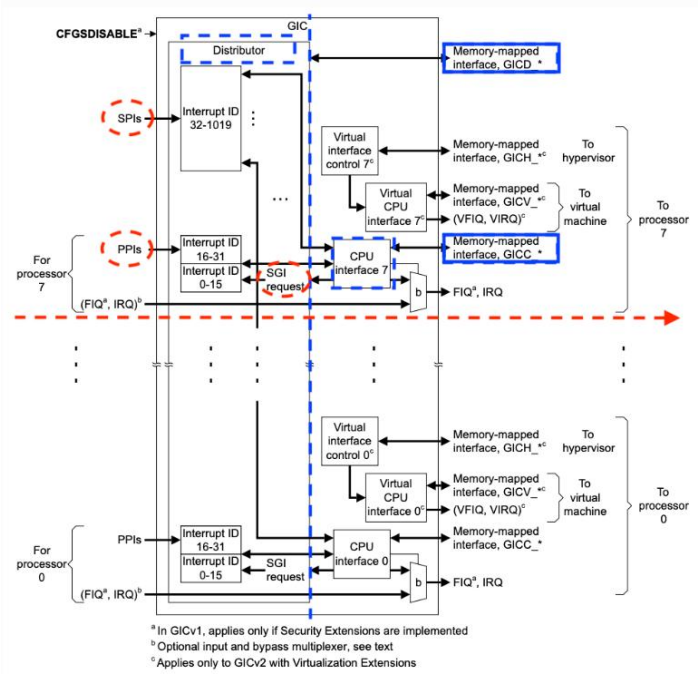
1.2.2 ARM core

Core（核心）在计算机科学中通常指的是处理器（CPU）的核心或内核。具体来说，一个 CPU 可以有一个或多个 Core，每个 Core 都可以独立地执行指令和进行计算。这种多核设计可以提高处理器的并行处理能力，使其能够同时处理多个任务或线程，从而提高整体性

能。

在 ARM 架构中，ARM core 是基于 ARM architecture 开发出来的 IP core，它是介于 architecture 和最终的 CPU（MCU）之间的中间产品。ARM core 有多种类型，如 ARM7、ARM9、ARM Cortex M3、ARM Cortex A57 等，这些 core 被广泛应用于各种嵌入式系统和移动设备中。

1.2.3 软中断、私有外设中断和共享外设中断



文中定义如下：

1.SPI（Shared Peripheral Interrupt）：

定义：共享外设中断，来源于外设，但可以通过 Distributor 分发给特定的 core。

中断编号：32-1019。

特点：当**多个外设设备共享同一个中断线**时，它们会生成共享外设中断。这种中断机制允许在硬件资源有限的情况下，多个设备能够共享中断处理能力。私有外设中断通常具有唯一的优先级和响应机制。所有核共享 SPI，即一个 SPI 中断可以被多个 core 接收和处理，但具体由哪个 core 处理取决于 Distributor 的分发策略和中断处理程序的设置。

2.PPI（Private Peripheral Interrupt）：

定义：私有外设中断，来源于外设，但只对指定的 core 有效。

中断编号：16-31。

特点：这种中断是由**特定的、私有的外设设备产生的**。例如，一个设备可能有一个私有的定时器或看门狗定时器，当这些设备需要 CPU 的注意时，它们会生成私有外设中断。这些中断是由特定的硬件设备生成的，并在硬件层面上进行处理。当设备需要 CPU 的注意时，它会通过中断控制器发送中断信号到 CPU，然后 CPU 会跳转到相应的中断处理程序来执行。每个 core 都有自己的 PPI，中断信号只会发送给指定的 core。这种中断类型通常用于 CPU core 的私有 timer 等。

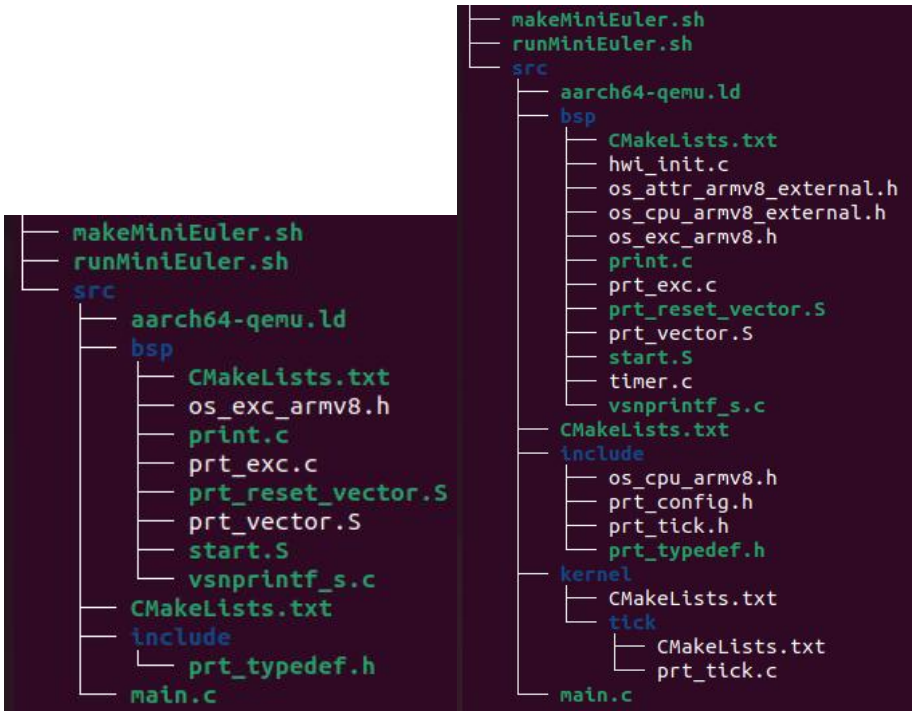
3.SGI（Software Generated Interrupt）：

定义：软中断，由软件产生的中断，用于给其他的 core 发送中断信号。

中断编号：0-15。

特点：这是一种**由内核机制触发的事件引起的**中断，它通常**用于在软件层面上模拟硬件中断的效果，实现异步执行的效果**。软中断通常与硬中断服务程序对内核的中断相对应，它可以帮助应用程序间接地调用系统 API 函数，实现应用程序的动态加载。软中断不是非常紧急，并且可能比较耗时，因此它们通常不会在中断服务上下文中立即执行，而是由系统根据需要安排运行时机。SGI 不是由硬件触发的，而是软件通过写中断控制器的某个寄存器触发的。它通常用于多核之间的消息通信或实现核间同步等功能。

1.3 代码结构



对应 lab4 的结构来看，在 bsp 硬件相关文件下，新增了 hwi_init.c，prt_tick.c hwi_init.c，os_attr_armv8_external.h，os_cpu_armv8_external.h，timer.c 等文件，新增一个 kernel 文件夹。且 prt_exc.c，prt_vector.c，main.c 等多文件均有修改。

1.4 GICv2 初始化

```
17        intc@8000000 {
18                phandle = <0x8002>;
19                reg = <0x00 0x8000000 0x00 0x10000 0x00 0x8010000 0x00 0x10000>;
20                compatible = "arm,cortex-a15-gic";
21                ranges;
22                #size-cells = <0x02>;
23                #address-cells = <0x02>;
24                interrupt-controller;
25                #interrupt-cells = <0x03>;
26
27                v2m@8020000 {
28                        phandle = <0x8003>;
29                        reg = <0x00 0x8020000 0x00 0x1000>;
30                        msi-controller;
31                        compatible = "arm,gic-v2m-frame";
32                };
33        };
```

```
12        timer {
13                interrupts = <0x01 0x0d 0x104 0x01 0x0e 0x104 0x01 0x0b 0x104 0x01 0x0a 0x104>;
14                always-on;
15                compatible = "arm,armv8-timer\0arm,armv7-timer";
16        };
```

reg 指明寄存器映射到内存的位置，#interrupt-cells 指明 interrupts 包括 3 个 cells。以 timer 设备为例，其中包括 4 个中断。

在 hwi_init.c 中 OsHwiInit 函数实现 GIC 的初始化，此外还提供了其他函数实现开关

指定中断、设置中断属性、确认中断和标记中断完成等功能。

需要参照 OsGicIntSetPriority 等函数实现 OsGicEnableInt 和 OsGicClearInt 函数。

OsGicInitCpuInterface：

初始化 GIC 的 distributor（分发器）和 CPU interface（CPU 接口）。首先禁用 distributor 和 CPU interface，以防止在配置过程中发生不期望的中断（GICD_CTLR, GICD_CTLR_DISABLE）。配置 GIC 的 CPU 接口，如设置优先级掩码（PMR）和二进制点寄存器（BPR）。启用 distributor 和 CPU interface，使 GIC 开始正常工作（GICC_CTLR, GICC_CTLR_ENABLE）。

OsGicDisableInt：

禁用指定 ID 的中断。

OsGicEnableInt：

使能指定 ID 的中断。

OsGicClearInt：

清除指定 ID 的中断。这通常是为了告诉 GIC 该中断已经被处理，从而防止中断再次被触发。通过写入 GIC 的 Interrupt Clear-Pending Registers（清除-挂起寄存器）来实现。

OsGicIntSetPriority：

设置指定 ID 的中断的优先级。中断优先级决定了 CPU 处理中断的顺序。通过写入 GIC 的 Interrupt Priority Registers（中断优先级寄存器）来实现。

OsGicIntSetConfig：

设置指定 ID 的中断的配置。这可以包括设置中断是边缘触发还是电平触发，是否支持组中断等。通过写入 GIC 的 Interrupt Configuration Registers（中断配置寄存器）来实现。

OsGicIntAcknowledge：

这个函数用于中断确认。

当中断发生时，GIC 会标记一个或多个中断为“挂起”（Pending）状态，并通知 CPU 有中断需要处理。CPU 在响应中断之前，需要执行中断确认操作。中断确认的主要作用包括：1.清除挂起状态：通过读取 GICC_IAR（Interrupt Acknowledge Register，中断确认寄存器），CPU 告诉 GIC 它已经开始处理这个中断。这个读取操作会清除该中断在 GIC 中的挂起状态。2.获取中断 ID：读取 GICC_IAR 寄存器时，GIC 会返回当前优先级最高的挂起中

断的 ID。这样，CPU 就可以知道它应该处理哪个中断。3.准备中断处理：一旦 CPU 知道了要处理的中断 ID，它就可以查找与该中断 ID 关联的中断处理程序（Interrupt Service Routine, ISR），并开始执行相应的中断处理代码。4. 防止中断丢失：通过确认中断，CPU 告诉 GIC 它已经开始处理该中断，从而避免了在中断处理过程中 GIC 再次发送相同的中断信号。

OsGicIntClear：

这个函数用于标记中断已经完成处理，并清除相应的中断位。当一个中断被处理后，需要通知 GIC 这个中断已经处理完毕，从而 GIC 可以从其挂起状态列表中移除这个中断。这通常是通过写入 GICC_EOIR（End of Interrupt Register，中断结束寄存器）来实现的。

OsHwiInit：

这个函数用于初始化硬件中断。它调用了 OsGicInitCpuInterface 函数来初始化 GIC 的 CPU 接口，以确保 CPU 可以正确地处理来自 GIC 的中断。

1.5 使能时钟中断

1.5.1 新建 src/include/prt_config.h

在其中定义 OS_TICK_PER_SECOND 宏，配置关于时间间隔（tick）的参数。OS_TICK_PER_SECOND 被定义为 1000，意味着系统每秒钟产生 1000 个 tick。

1.5.2 新建 src/include/os_cpu_armv8.h

CURRENT_EL_2、CURRENT_EL_1、CURRENT_EL_0：

定义了 ARMv8 架构中的当前异常级别（Current Exception Level, EL）。

DAIF_DBG_BIT、DAIF_ABT_BIT、DAIF_IRQ_BIT、DAIF_FIQ_BIT：

定义了 ARMv8 架构中的中断屏蔽位（DAIF，Debug, Abort, IRQ, FIQ）。可以通过修改这些位来控制哪些类型的中断或异常是屏蔽的（即不会触发）。

PRT_DSB()、PRT_DMB()、PRT_ISB()：

定义了内嵌汇编指令，用于控制 ARMv8 CPU 的内存访问顺序。

DSB（Data Synchronization Barrier）指令确保所有在 DSB 前的写操作在 DSB 后的读操作之前完成。它通常用于确保数据一致性。

DMB（Data Memory Barrier）指令与 DSB 类似，但它只确保内存访问的顺序，而不保

证写操作的完成。它用于控制数据内存访问的顺序。

ISB（Instruction Synchronization Barrier）指令确保在 ISB 之前的指令在 ISB 之后的指令之前完成。它用于确保指令执行的顺序。

1.5.3 新建 src/bsp/timer.c 文件

对定时器和对应的中断进行配置。

中断：配置为电平触发、优先级为 0、清除中断、启用中断。

定时器：使用内嵌汇编来读取时钟频率、计算定时器周期、禁用并清除计时器、启用计时器（是为了更新计时器周期）、清除 IRQ 中断屏蔽允许 IRQ 中断被触发。

1.6 时钟中断处理

将 prt_vector.S 中的 EXC_HANDLE 5 OsExcDispatch 改为 EXC_HANDLE 5 OsHwiDispatcher，表明我们将对 IRQ 类型的异常（即中断）使用 OsHwiDispatcher 处理。并增加 OsHwiDispatcher 处理代码。

在 prt_exc.c 中加入头文件引用：os_attr_armv8_external.h ，os_cpu_armv8.h。加入 OsHwiDispatch 处理（在 OsHwiDispatcher 调用）。

新建 src/bsp/os_attr_armv8_external.h 头文件。

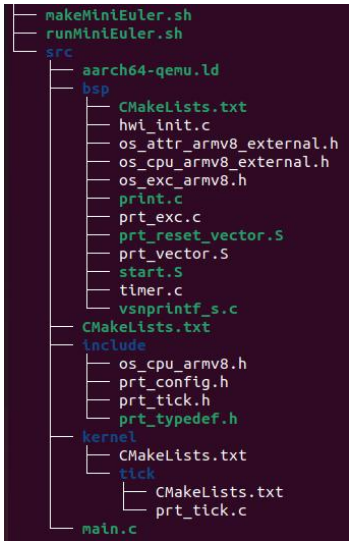
新建 src/kernel/tick/prt_tick.c 文件，提供 OsTickDispatcher 时钟中断处理函数。

将 OsIntLock 和 OsIntRestore 函数放入 prt_exc.c 中，用于关中断和开中断。

修改后向量表将 IRQ 类型的异常（中断）指向 OsHwiDispatcher 函数处理，它调用了 OsHwiDispatch 函数，OsHwiDispatch 函数调用 OsHwiHandleActive 函数，OsHwiHandleActive 函数调用 OsTickDispatcher()函数，最终由 OsTickDispatcher()进行具体的中断处理。

1.7 构建系统处理

这个搞了很久。



include_directories()：这个命令告诉 CMake 在编译时应该在哪些目录中寻找头文件。

add_subdirectory()：这个命令告诉 CMake 在指定的子目录中查找另一个 CMakeLists.txt 文件，并处理其中的构建规则。这允许将项目分解为多个子目录，每个子目录都有自己的构建规则。例如，add_subdirectory(bsp)会告诉 CMake 在 bsp 子目录中查找 CMakeLists.txt 文件，并处理该文件中定义的任何目标（如库或可执行文件）、源文件、头文件目录等。

add_library()：这个命令添加一个库目标到构建系统中。命令的 OBJECT 类型是一个特殊的库类型，它允许将源文件（.c、.cpp 等）编译为对象文件（.o），但不将它们链接成一个库。这些对象文件可以在其他目标或库中被链接，用于在多个目标之间共享编译后的对象文件，提高编译效率。

list(APPEND obj ...) :这个命令将多个元素追加到名为 obj 的列表中。

我们将头文件 include 在总 CM 文件中包含进入编译目录 include_directories()，无需在 include 文件夹中使用 CM 文件；对于其他源文件，每个中都有一个 CM 文件。如果是底层目录，需要用 set 设置源文件列表并用 add_library 生成.o 目标文件并在总 CM 文件 list 中指定该目标，如果非底层目录，使用 add_subdirectory()在指定子目录中查找 CM 文件，并处理其中的构建规则。若在子目录中包含头文件，也需要被包含在 include_directories()命令中。

首先有一个总的 CMakeLists.txt 文件，名字为：src/CMakeLists.txt，包含一些基本的指令来指定源文件、库依赖、编译选项等。本实验需修改包含子目录 kernel 和编译目标 tick。

```
include_directories(
    ${CMAKE_CURRENT_SOURCE_DIR}/include   # 增加 src/include 目录
    ${CMAKE_CURRENT_SOURCE_DIR}/bsp
)

add_subdirectory(bsp)
add_subdirectory(kernel) # 增加 kernel 子目录

list(APPEND OBJS $<TARGET_OBJECTS:bsp> $<TARGET_OBJECTS:tick>) # 增加 $<TARGET_OBJECTS:tick> 目标
add_executable(${APP} main.c ${OBJS})
```

其余的文件夹 bsp，kernel 中均有自己的子目录 CMakeLists.txt 文件，分别添加该子目录下的源文件。如果子目录下还有更小的子目录，并且这些子目录也包含 CMakeLists.txt 文件，使用 add_subdirectory() 来包含它们。

src/bsp/ CMakeLists.txt：

```
set(SRCS start.S prt_reset_vector.S print.c vsnprintf_s.c prt_exc.c prt_vector.S os_exc_armv8.h  os_attr_armv8_external.h  os_cpu_armv8_external.h  hwi_init.c  timer.c )
add_library(bsp OBJECT ${SRCS})  # OBJECT类型只编译生成.o目标文件，但不实际链接成库
```

src/kernel/tick/CMakeLists.txt：

记得修改第二行中的目标名称；不能直接删除第二行，会报错无法找到目标 tick。

```
set(SRCS prt_tick.c )
add_library(tick OBJECT ${SRCS})  # OBJECT类型只编译生成.o目标文件，但不实际链接成库
```

src/kernel/CMakeLists.txt：

```
add_subdirectory(tick)
```

1.8 读取系统 Tick 值

新建 prt_tick.h，声明 Tick 相关的接口函数。

修改 main.c 文件，使用循环输出当前的时钟值，反应实验是否成功。

1.9 构建项目与执行结果

本实验逻辑为：

src/bsp/hwi_init.c 文件，初始化 GIC。

src/include/prt_config.h，定义宏，配置关于时间间隔（tick）的参数。

src/include/os_cpu_armv8.h，定义宏，定义当前异常级别、中断屏蔽位和内嵌汇编指令。

src/bsp/timer.c，对定时器和对应的中断进行配置。

修改 prt_vector.S 的异常向量表，并加入 OsHwiDispatcher 处理代码。

修改 prt_exc.c，添加头文件引用，增加实际的 OsHwiDispatch 代码，增加 OsIntLock 和

OsIntRestore 函数实现开关中断。

src/kernel/tick/prt_tick.c，提供 OsTickDispatcher 时钟中断处理函数。

prt_tick.h，声明 Tick 相关的接口函数。

新增头文件 src/bsp/os_cpu_armv8_external.h，src/bsp/os_attr_armv8_external.h。

修改构建系统和 main.c 函数。

本实验中只实现了时钟 tick，许多代码运用的解释需要见于实验六，中断。

实验结果：

```
guoruiling@guoruiling-virtual-machine:~/lab5$ sh makeMiniEuler.sh
mkdir: cannot create directory ‘build’: File exists
-- The C compiler identification is GNU 11.2.1
-- The ASM compiler identification is GNU
-- Found assembler: /home/guoruiling/aarch64-none-elf/gcc-arm-11.2-2022.02-x86_64-aarch64-none-
elf/bin/aarch64-none-elf-gcc
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /home/guoruiling/aarch64-none-elf/gcc-arm-11.2-2022.02-x86_64-
aarch64-none-elf/bin/aarch64-none-elf-gcc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/guoruiling/lab5/build
[  9%] Building C object kernel/tick/CMakeFiles/tick.dir/prt_tick.c.obj
[  9%] Built target tick
Scanning dependencies of target bsp
[ 18%] Building ASM object bsp/CMakeFiles/bsp.dir/start.S.obj
[ 27%] Building ASM object bsp/CMakeFiles/bsp.dir/prt_reset_vector.S.obj
[ 36%] Building C object bsp/CMakeFiles/bsp.dir/print.c.obj
[ 45%] Building C object bsp/CMakeFiles/bsp.dir/vsnprintf_s.c.obj
[ 54%] Building C object bsp/CMakeFiles/bsp.dir/prt_exc.c.obj
[ 63%] Building ASM object bsp/CMakeFiles/bsp.dir/prt_vector.S.obj
[ 72%] Building C object bsp/CMakeFiles/bsp.dir/hwi_init.c.obj
[ 81%] Building C object bsp/CMakeFiles/bsp.dir/timer.c.obj
[ 81%] Built target bsp
[ 90%] Building C object CMakeFiles/miniEuler.dir/main.c.obj
[100%] Linking C executable miniEuler
[100%] Built target miniEuler
```

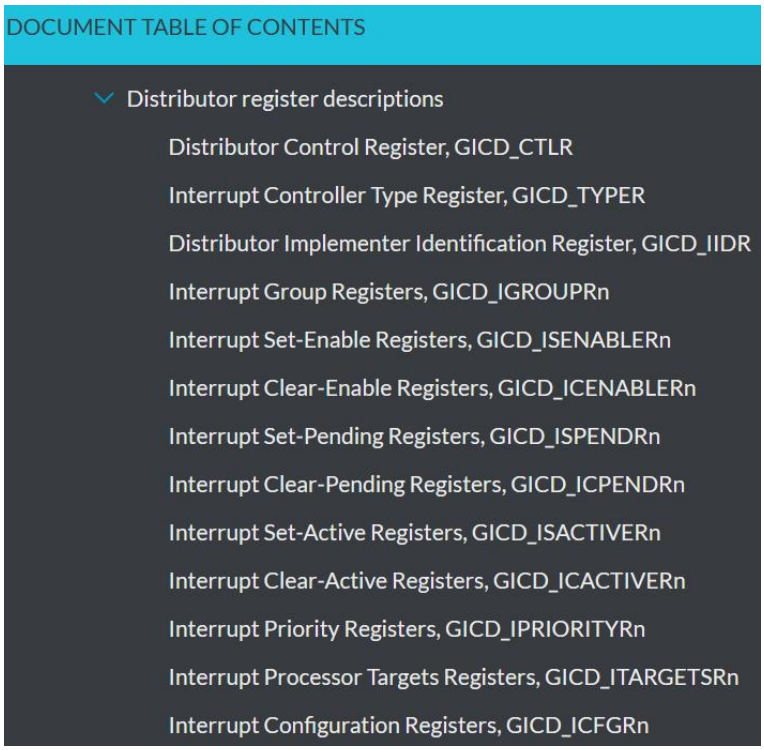
```
guoruiling@guoruiling-virtual-machine:~/lab5$ sh runMiniEuler.sh
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel bui
ld/miniEuler -s
------------ ------------    ------    -----------
 ************ ************   ********   ***********
----         ------------  -----------  ----    ---
************     ****     ****    **** *********
 ------------     ----     ------------ ---------
       *****     ****     ************ ****  ****
 ------------     ----     ----    ---- ----   ----
************     ****     ****    **** ****    ****
ctr-a h: print help of qemu emulator. ctr-a x: quit emulator.

[0] current tick: 2
[1] current tick: 125
[2] current tick: 189
[3] current tick: 270
[4] current tick: 348
[5] current tick: 397
[6] current tick: 437
[7] current tick: 502
[8] current tick: 544
[9] current tick: 673
```

2 实验任务

2.1 作业 1

题目：实现 hwi_init.c 中缺失的 OsGicEnableInt 和 OsGicClearInt 函数。



查询阅读手册其 Chapter 4 Programmers’ Model 部分，可知有关于 GICD 和 GICC 寄存器的描述，以及如何使能 Distributor 和 CPU Interfaces 的方法。

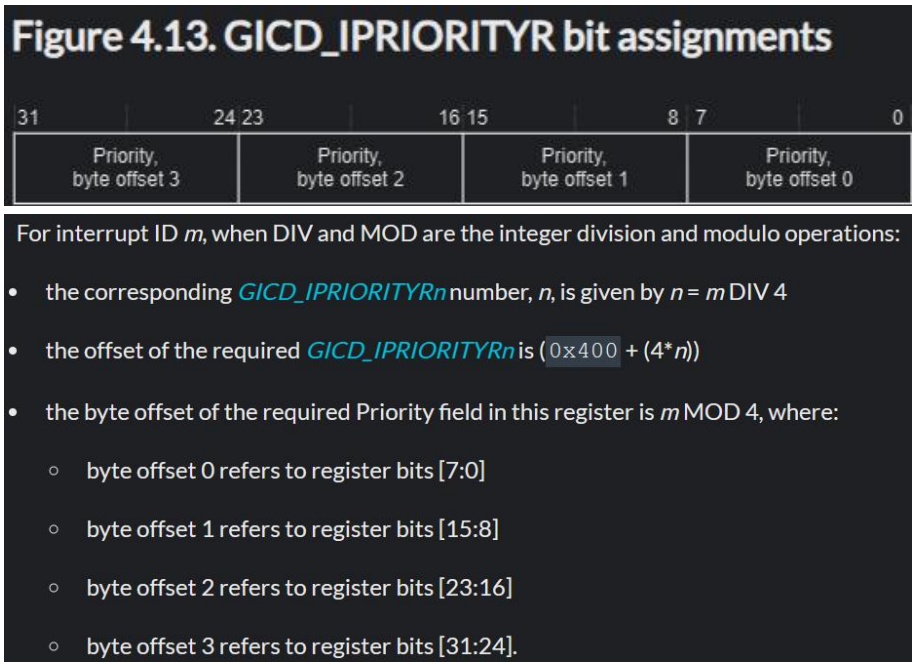
实验要求中提出需要参照 OsGicIntSetPriority 等函数实现 OsGicEnableInt 和 OsGicClearInt 函数，那么我们首先根据手册来解释一下 OsGicIntSetPriority 函数。

```
// 设置中断号为 interrupt 的中断的优先级为 priority
OS_SEC_L4_TEXT void OsGicIntSetPriority(uint32_t interrupt, uint32_t priority)
{
    uint32_t shift = (interrupt % GICD_IPRIORITY_SIZE) * GICD_IPRIORITY_BITS;
    volatile uint32_t* addr = ((volatile U32 *)(uintptr_t)(GICD_IPRIORITYn +
(interrupt / GICD_IPRIORITY_SIZE) * sizeof(U32))) ;
    uint32_t value = GIC_REG_READ(addr);
    value &= ~(0xff << shift); // 每个中断占 8 位，所以掩码为 0xFF115
    value |= priority << shift;
    GIC_REG_WRITE(addr, value);
}
```

首先要知道的是，在 ARM GIC（Generic Interrupt Controller）架构中，中断优先级寄存器（通常是 GICD_IPRIORITYn 系列寄存器）被组织成一系列的 32 位寄存器，每个寄存器可以存储多个中断的优先级。所以有：

interrupt / GICD_IPRIORITY_SIZE 计算中断所属的寄存器编号。

interrupt % GICD_IPRIORITY_SIZE 计算中断在其所属寄存器中的位置。




```
19  #define GICD IPRIORITYn          (GIC DIST BASE + 0x0400U)
```

```
28  #define GICD_IPRIORITY_SIZE          4
29  #define GICD_IPRIORITY_BITS          8
```

首先，这个函数接受两个 uint32_t 类型的参数：interrupt（中断号）和 priority（优先级）。

shift：用于确定在 32 位寄存器中，该中断的优先级应该位于哪个位置（从最低位开始计算）。GICD_IPRIORITY_SIZE 表示寄存器中可以包含的中断数，在本实验中为 4。GICD_IPRIORITY_BITS 表示每个中断的优先级所占的位数，本实验中为 8。

addr：指向 GIC 中存储该中断优先级的寄存器地址。GICD_IPRIORITYn 是 GIC 中存储中断优先级的基地址。

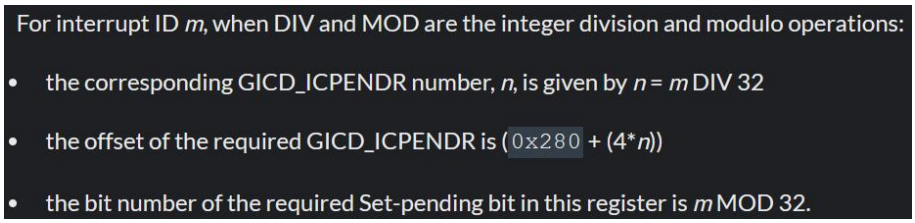
使用 GIC_REG_READ 宏（或函数）从指定的地址读取当前寄存器的值。使用 GIC_REG_READ 读出数据，修改优先级，使用 GIC_REG_WRITE 写入新值。

接下来我们来看 OsGicClearInt 的代码，该函数的实现已给出：

```
OS_SEC_L4_TEXT void OsGicClearInt(uint32_t interrupt)
```

```
{
    GIC_REG_WRITE(GICD_ICPENDRn + (interrupt / GICD_ICPENDR_SIZE)*sizeof(U32),
1 << (interrupt % GICD_ICPENDR_SIZE));
}
```

首先需要知道的是，GICD_ICPENDRs 为 GIC 支持的每个中断提供一个 Clear-pending 位。将 1 写入 Clear-pending bit 可清除相应外设中断的挂起状态。读取位可识别中断是否处于挂起状态。




```
16   #define GICD_ICPENDRn              (GIC_DIST_BASE + 0x0280U)

27   #define GICD_ICPENDR_SIZE                32
```

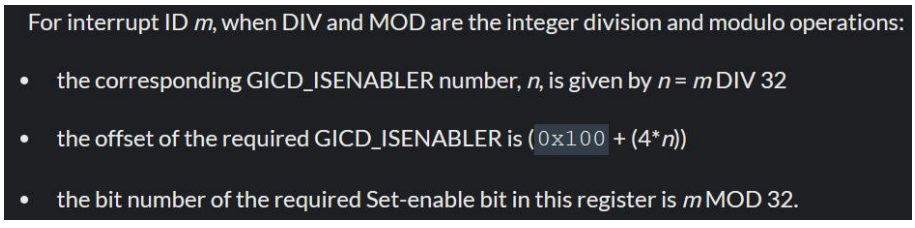
GICD_ICPENDRn + (interrupt / GICD_ICPENDR_SIZE)*sizeof(U32)：计算寄存器地址。

1 << (interrupt % GICD_ICPENDR_SIZE)：将 1 移至寄存器内的对应位，并写入。

接下来我们尝试写 OsGicEnableInt 函数：

首先需要知道的是，GICD_ISENABLERs 为 GIC 支持的每个中断提供一个 Set-enable 位。将 1 写入 Set-enable 位可以将相应的中断从分发器转发到 CPU 接口。读取位可识别中断是否已启用。

同样有：




```
13   #define GICD_ISENABLERn            (GIC_DIST_BASE + 0x0100U)

25   #define GICD_ISENABLER_SIZE              32
```

```
OS_SEC_L4_TEXT void OsGicEnableInt(U32 intId)
{
    GIC_REG_WRITE(GICD_ISENABLERn + (intId / GICD_ISENABLER_SIZE)*sizeof(U32),
1 << (intId % GICD_ISENABLER_SIZE));
}
```

模仿上个函数编写代码。