

# 本科生实验报告

实验课程: 操作系统原理实验

实验名称: 中断

专业名称: 计算机科学与技术

学生姓名: 数据删除

学生学号: 数据删除

实验成绩:

报告时间: 2025年4月3日

## 实验要求

### Assignment 1 混合编程的基本思路

复现Example 1，结合具体的代码说明C代码调用汇编函数的语法和汇编代码调用C函数的语法。例如，结合代码说明global、extern关键字的作用，为什么C++的函数前需要加上extern "C"等， 结果截图并说说你是怎么做的。同时，学习make的使用，并用make来构建Example 1，结果截图并说说你是怎么做的。

### Assignment 2 使用C/C++来编写内核

复现Example 2，使用gdb或其他debug工具在进入保护模式的4个重要步骤上设置断点，并结合代码、寄存器的内容等来分析这4个步骤，最后附上结果截图。gdb的使用可以参考appendix的“debug with gdb and qemu”部份。

### Assignment 3 中断的处理

复现Example 3，你可以更改Example中默认的中断处理函数为你编写的函数，然后触发之，结果截图并说说你是怎么做的。

### Assignment 4 时钟中断

复现Example 4，仿照Example中使用C语言来实现时钟中断的例子，利用C/C++、 InterruptManager、STDIO和你自己封装的类来实现你的时钟中断处理过程，结果截图并说说你是怎么做的。注意，不可以使用纯汇编的方式来实现。(例如，通过时钟中断，你可以在屏幕的第一行实现一个跑马灯。跑马灯显示自己学号和英文名，即类似于LED屏幕显示的效果。)

## 实验过程

### Assignment 1

这个任务用于学习如何进行混合编程，要点在于：

#### 使用global关键字暴露全局符号

global关键字用于在汇编语言中声明一个符号（变量、函数等）为全局可见的。这意味着，其他编译单元（例如 C/C++ 代码）可以访问这个符号。

在asm\_func.asm中，global function\_from\_asm声明了function\_from\_asm函数为全局可见。因此，main.cpp中的extern "C" void function\_from\_asm(); 才能正确地引用到这个汇编函数。

#### 使用extern关键字声明外部符号

extern关键字用于声明一个符号（变量、函数等）在其他编译单元中定义。它告诉编译器，这个符号的定义不在当前文件中，而是在其他文件中。

在asm\_func.asm中，extern function\_from\_C和extern function\_from\_CPP声明了function\_from\_C和function\_from\_CPP函数在其他文件中定义（分别是 c\_func.c和cpp\_func.cpp）。

在main.cpp中，extern "C" void function\_from\_asm();声明了 function\_from\_asm 函数在其他文件中定义（asm\_func.asm）。

#### C++中extern关键字的使用

extern "C" 用于告诉 C++ 编译器，按照 C 语言的规则来处理声明的函数或变量。这主要是因为 C++ 和 C 在函数命名方式上有所不同。C++ 编译器为了支持函数重载等特性，会对函数名进行编码，生成一个更复杂的名字。而 C 编译器则通常直接使用函数名。

### Assignment 2

我使用vscode+clangd为C++程序提供代码高亮服务，为了正确生成compile\_commands.json，使用xmake而不是make进行编译。

观察到输出hello world的逻辑在asm\_util.asm内的asm\_hello\_world函数中，因此将其内部逻辑替换即可改为输出学号。

### Assignment 3

中断处理函数在InterruptManager::initialize由setInterruptDescriptor导入IDT，只需修改其中的参数即可指定自定义的中断处理函数。

在asm\_util.asm内新增asm\_hello\_world函数并实现输出hello world逻辑，然后修改InterruptManager::initialize即可。

### Assignment 4

中断处理程序在interrupt.cpp/c\_time\_interrupt\_handler()中，只需对其进行修改即可实现自己的中断服务。

考虑实现学号和英文名的跑马灯，将其转换为字符串常量后存放于全局，并使用一个全局下标变量表示当前高亮的字符下标。每次中断程序触发时，清屏并逐个输出字符串，迭代到高亮字符时使用一个不同的颜色实现跑马灯效果，然后将下标变量指向下一个应该高亮的位置。

## 关键代码

### Assignment 2

xmake配置。首先是全局配置，定义了3个rule，分别负责编译和链接asm elf（entry、util）、asm bin（mbr、bootloader）和kernel导出到磁盘：

set\_xmakever("2.9.8")

add\_rules("mode.debug","mode.release")

set\_languages("c11","cxx20")

rule("asm.elf",function()

    set\_extensions(".asm")

    on\_build\_file(function (target, sourcefile, opt)

        local objfile = target:objectfile(sourcefile)

        os.mkdir(path.directory(objfile))

        local includes=""

        for \_,v in ipairs(target:get("includedirs")) do

            includes=includes.."-I"..v

        end

        os.runv("nasm", {"-f", "elf32", includes, "-o", objfile, sourcefile})

        table.insert(target:objectfiles(),objfile)

    end)

    on\_link(function(target)

        local objfiles = target:objectfiles()

        local outputfile = target:targetdir().."/"..target:filename()

        os.mkdir(target:targetdir())

        os.runv("ld", {"-m", "elf\_i386","-e","entry\_kernel","-Ttext","0x00020000", "-o", os.projectdir().."/run/kernel.o", unpack(objfiles)})

        os.runv("ld", {"-m", "elf\_i386","-e","entry\_kernel","-Ttext","0x00020000","--oformat","binary", "-o", outputfile, unpack(objfiles)})

    end)

end)

rule("asm.bin",function()

    set\_extensions(".asm")

    on\_build\_file(function (target, sourcefile, opt)

        os.mkdir(target:targetdir())

        local includes=""

        for \_,v in ipairs(target:get("includedirs")) do

            includes=includes.."-I"..v

        end

        os.runv("nasm", {"-f", "bin", includes, "-o", target:targetdir().."/"..target:filename(), sourcefile})

    end)

    on\_link(function(target)

    end)

end)

rule("kernel.build",function()

    before\_build(function(target)

        os.runv("mkdir",{"-p","run"})

    end)

    after\_build(function(target)

        mbr = target:dep(target:name()..".mbr")

        bootloader = target:dep(target:name()..".bootloader")

        kernel = target:dep(target:name()..".kernel")

        local mbr\_file = path.join(mbr:targetdir(),mbr:filename())

        local bootloader\_file = path.join(bootloader:targetdir(), bootloader:filename())

        local kernel\_file = path.join(kernel:targetdir(), kernel:filename())

        local hd\_file = path.join(os.projectdir(), "run", "hd.img")

        os.runv("dd", {

            "if=" .. mbr\_file, "of=" .. hd\_file, "bs=512", "count=1", "seek=0",

            "conv=notrunc"

        })

        os.runv("dd", {

            "if=" .. bootloader\_file, "of=" .. hd\_file, "bs=512", "count=5",

            "seek=1", "conv=notrunc"

        })

        os.runv("dd", {

            "if=" .. kernel\_file, "of=" .. hd\_file, "bs=512", "count=200", "seek=6",

            "conv=notrunc"

        })

    end)

end)

includes("\*\*/xmake.lua")

具体项目的xmake.lua配置：

add\_includedirs("include")

target("lab4-2.mbr", function()

    add\_rules("asm.bin")

    add\_files("src/boot/mbr.asm")

    set\_filename("mbr.bin")

end)

target("lab4-2.bootloader", function()

    add\_rules("asm.bin")

    add\_files("src/boot/bootloader.asm")

    set\_filename("bootloader.bin")

end)

target("lab4-2.kernel", function()

    add\_rules("asm.elf")

    add\_files("src/boot/entry.asm", "src/kernel/\*.cpp", "src/utils/\*.asm")

    set\_filename("kernel.bin")

    add\_cxxflags("-march=i386", "-m32", "-nostdlib", "-fno-builtin", "-ffreestanding", "-fno-pic")

end)

target("lab4-2", function()

    set\_kind("phony")

    add\_deps("lab4-2.mbr", "lab4-2.bootloader", "lab4-2.kernel")

    add\_rules("kernel.build")

end)

修改后的asm\_util.asm：

asm\_hello\_world:

    pushad

    mov  ecx, student\_id\_end - student\_id

    mov  ebx, 0

    mov  esi, student\_id

    mov  ah,  0x3

    output\_student\_id:

        mov  al,           [esi]

        mov  word[gs:ebx], ax

        add  ebx,          2

        inc  esi

        loop output\_student\_id

    popad

    ret

student\_id db '00000000'

student\_id\_end:

### Assignment 3

在asm\_utils.asm加入自定义中断处理函数：

asm\_hello\_world:

    pushad

    mov  ecx, helloworld\_end - helloworld

    mov  ebx, 0

    mov  esi, helloworld

    mov  ah,  0x3

    output\_helloworld:

        mov  al,           [esi]

        mov  word[gs:ebx], ax

        add  ebx,          2

        inc  esi

        loop output\_helloworld

    popad

    ret

helloworld db 'hello world'

helloworld\_end:

该函数会打印一个hello world。

然后在interrupt.cpp中向IDT传入该函数的地址：

void InterruptManager::initialize() {

    // 初始化IDT

    IDT = (uint32\*)IDT\_START\_ADDRESS;

    asm\_lidt(IDT\_START\_ADDRESS, 256 \* 8 - 1);

    for (uint i = 0; i < 256; ++i) {

        // 修改为自定义中断处理函数

        setInterruptDescriptor(i, (uint32)asm\_hello\_world, 0);

    }

}

### Assignment 4

修改c\_time\_interrupt\_handler为自定义流水灯程序：

char message[] = "00000000 Zhangsan";

int index = 0;

// 中断处理函数

extern "C" void c\_time\_interrupt\_handler() {

    // 清空屏幕

    for (int i = 0; i < 80; ++i) {

        stdio.print(0, i, ' ', 0x07);

    }

    if (message[index] == 0) index = 0;

    stdio.moveCursor(0);

    for (int i = 0; message[i]; i++) {

        if (i == index) {

            stdio.print(message[i], 0x03);

        }

        else {

            stdio.print(message[i], 0x07);

        }

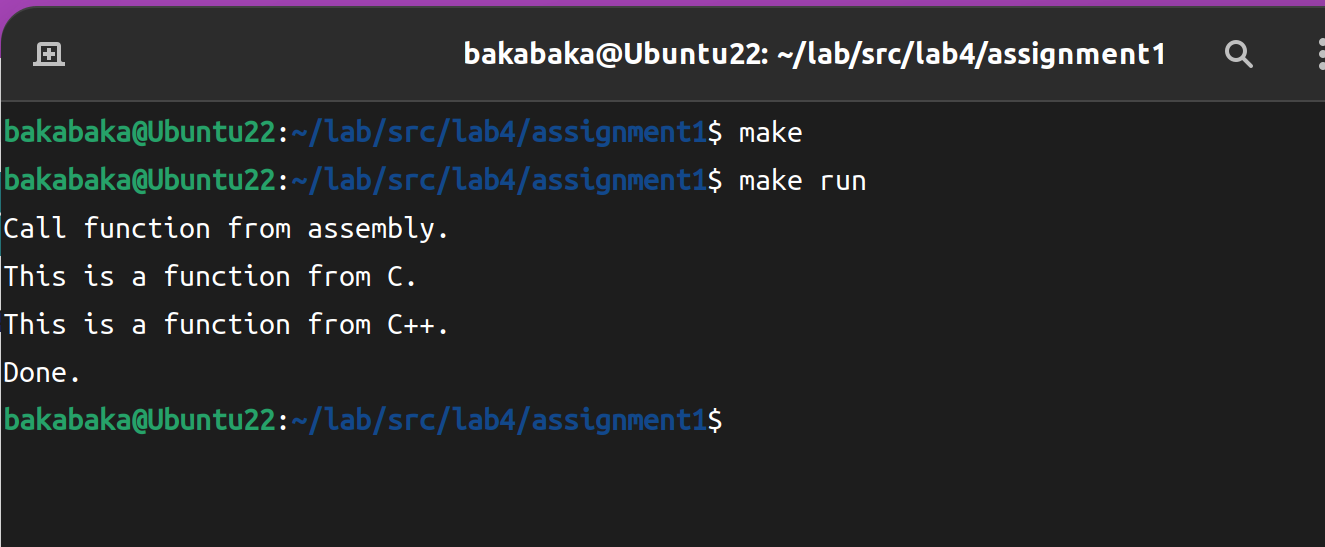
    }

    index++;

}

## 实验结果

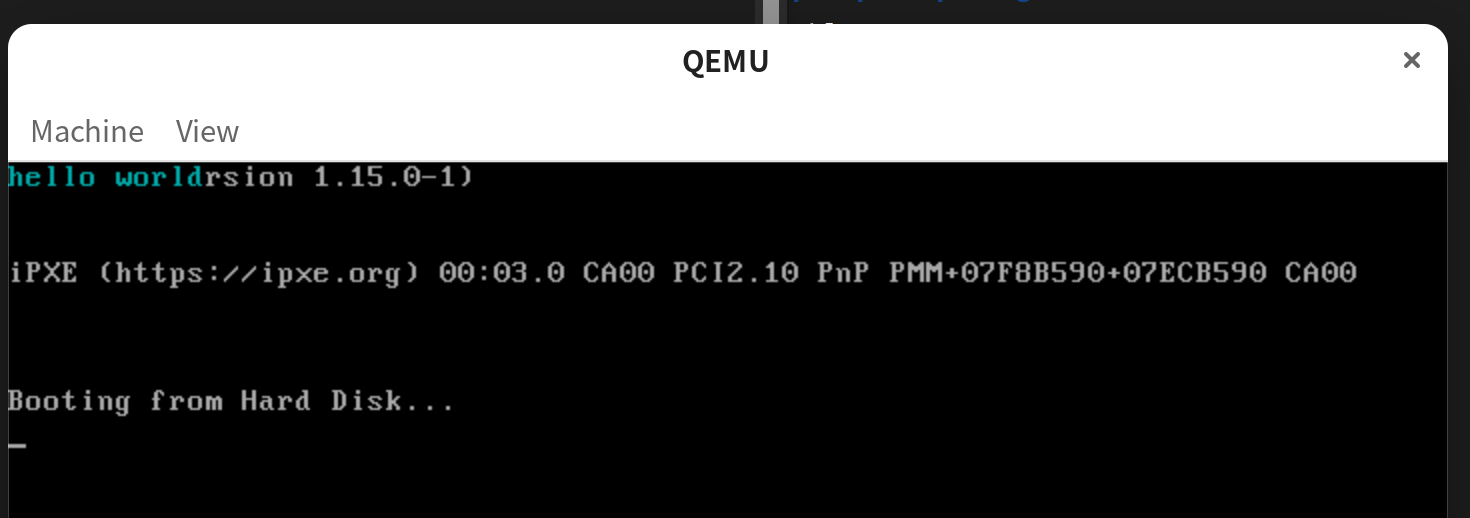
### Assignment 1



### Assignment 2

数据删除。

### Assignment 3



### Assignment 4

假装能看到流水灯效果。

数据删除。

## 总结

通过本次实验，我深入理解了保护模式下的中断处理机制及其完整实现流程。实验从混合编程基础开始，逐步构建了IDT的初始化框架，最终实现了时钟中断的捕获与处理。本次实验为我后续学习进程调度、系统调用等高级特性奠定了坚实基础。