lab4 混合编程与中断

1、实验要求

1、了解C代码是如何通过预编译、编译、汇编和链接生成最终的可执行文件。

2、为了更加有条理地管理我们操作系统的代码，使用一种C/C++项目管理方案。

3、学习C和汇编混合编程方法，即如何在C代码中调用汇编代码编写的函数和如何在汇编代码中调用使用C编写的函数。

4、学习在保护模式下的中断处理机制和可编程中断部件8259A芯片。

5、通过编写实时钟中断处理函数来将本章的所有内容串联起来。

6、为后面的二级分页机制和多线程/进程打下基础。

2、实验过程+实验结果+关键代码

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。

`c_func.c`

```
#include <stdio.h>

void function_from_C() {
    printf("This is a function from C.\n");
}
```

在 C++ 中，如果一个函数是按照 C 语言的方式编写的，那么在函数声明或定义前需要添加 `extern "C"`。

原因：C++ 采用了一种名为命名重载的技术，即在函数名前面添加了一些字符，以便区分不同的函数，使得 C++ 代码无法直接调用 C 语言的函数。为了解决这个问题，C++ 提供了 `extern "C"` 语法来告诉编译器，这个函数要使用 C 语言的命名约定，而不是 C++ 的命名重载，这样编译器就不会在函数名前面添加额外的字符，从而使 C++ 代码可以直接调用 C 语言的函数。

例如，`cpp_func.cpp`中的一个函数声明，使用了 `extern "C"`：

```
extern "C" void function_from_CPP();
```

该声明告诉编译器，`function_from_CPP` 函数应该按照 C 语言的约定进行命名，而不是 C++ 的命名重载方式。当 C++ 代码调用 `function_from_CPP` 函数时，编译器将不会对函数名进行额外的处理。

`cpp_func.cpp`

```cpp
#include <iostream>

extern "C" void function_from_CPP() {
    std::cout << "This is a function from C++." << std::endl;
}
```

在汇编语言中，`global`用于定义全局变量，而`extern`用于声明在其他文件中定义的变量或函数

具体而言，`global`使得该变量或函数能够被其它模块访问和使用，并且可以在当前模块中定义和使用

而`extern`表明该变量或者函数已经在其它模块中定义过，而当前模块只是引用而非定义

`asm_utils.asm`

- `global function_from_asm`：定义`function_from_asm`为全局可见，并且可以被其他模块引用和调用。
- `extern function_from_C`：声明`function_from_C`是一个外部函数，需要从其他目标文件中获取其定义。
- `extern function_from_CPP`：声明`function_from_CPP`是一个外部函数，需要从其他目标文件中获取其定义。

```asm
[bits 32]
global function_from_asm
extern function_from_C
extern function_from_CPP

function_from_asm:
    call function_from_C
    call function_from_CPP
    ret
```

`main.cpp`

```cpp
#include <iostream>

extern "C" void function_from_asm();

int main() {
    std::cout << "Call function from assembly." << std::endl;
    function_from_asm();
    std::cout << "Done." << std::endl;
}
```

`./main.out`是x86的指令，在arm架构下不能直接运行

需要修改`Makefile`文件，用`cpio`打包`initramfs`，`qemu`启动内核，并加载`initramfs`，即可看到`main`的输出

```
TARTGET = main
ASM_FILE = $(wildcard *.asm)
```

```
build:
  @rm -rf *.o
  @nasm -f elf32 $(ASM_FILE)
  @i686-linux-gnu-gcc -m32 -g -static -c c_func.c
  @i686-linux-gnu-g++ -m32 -g -static -c cpp_func.cpp
  @i686-linux-gnu-g++ -m32 -g -static -c main.cpp
  @i686-linux-gnu-g++ -m32 -g -static -o $(TARTGET) main.o c_func.o
cpp_func.o asm_utils.o
  @echo main | cpio -o --format=newc > hwinitramfs
run:
  @qemu-system-i386 -kernel ~/lab1/linux-5.10.172/arch/x86/boot/bzImage -
initrd hwinitramfs -append "console=ttyS0 rdinit=main" -nographic
clean:
  @rm -rf *.o
```

```
[    1.605453] Freeing unused kernel image (initmem) memory: 680K
[    1.609100] Write protecting kernel text and read-only data: 15604k
[    1.609334] Run main as init process
[    1.613034] process '/main' started with executable stack
Call function from assembly.
This is a function from C.
This is a function from C++.
Done.
[    1.639749] Kernel panic - not syncing: Attempted to kill init! exitcode=0x00
[    1.640041] CPU: 0 PID: 1 Comm: main Not tainted 5.10.172 #1
[    1.640160] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.164
[    1.640414] Call Trace:
[    1.641055]  dump_stack+0x54/0x68
[    1.641143]  panic+0xaf/0x270
[    1.641188]  do_exit.cold+0x52/0xcd
[    1.641271]  do_group_exit+0x2a/0x90
[    1.641323]  __ia32_sys_exit_group+0x10/0x10
[    1.641411]  __do_fast_syscall_32+0x45/0x80
[    1.641464]  do_fast_syscall_32+0x29/0x60
[    1.641548]  do_SYSENTER_32+0x15/0x20
[    1.641598]  entry_SYSENTER_32+0x9f/0xf2
[    1.641782] EIP: 0xb7fc4549
[    1.641981] Code: 03 74 c0 01 10 05 03 74 b8 01 10 06 03 74 b4 01 10 07 03 76
[    1.642331] EAX: ffffffda EBX: 00000000 ECX: 000000fc EDX: 00000001
[    1.642437] ESI: ffffffdc EDI: 09372380 EBP: 08221840 ESP: bf9ab420
[    1.642541] DS: 007b ES: 007b FS: 0000 GS: 0033 SS: 007b EFLAGS: 00000212
[    1.642985] Kernel Offset: disabled
[    1.643214] ---[ end Kernel panic - not syncing: Attempted to kill init! exi-
```

Assignment 2 使用C/C++来编写内核

复现Example 2，在进入`setup_kernel`函数后，将输出 Hello World 改为输出你的学号，结果截图并说说你是怎么做的。

`setup.cpp`

```
#include "asm_utils.h"
extern "C" void setup_kernel()
{
    asm_my_function();//在进入setup_kernel后，将输出Hello World的asm_hello_world函数改为输出学号的asm_my_function函数
    while(1)
    {
    }
}
```

在bootloader中加载操作系统内核到地址0x20000，然后跳转到0x20000。内核接管控制权后，输出学号和英文名

`boot.inc`

```
; ___________kernel_________
KERNEL_START_SECTOR equ 6
KERNEL_SECTOR_COUNT equ 200
KERNEL_START_ADDRESS equ 0x20000
```

`entry.asm`定义内核进入点

链接阶段将`entry.asm`的代码放在内核代码的最开始部份，使得bootloader在执行跳转到`0x20000`后，即内核代码的起始指令，执行的第一条指令是`jmp setup_kernel`。在`jmp`指令执行后，跳转到使用C++编写的函数`setup_kernel`，之后可以使用C++来写内核。

```
extern setup_kernel
enter_kernel:
    jmp setup_kernel
```

`asm_utils.asm`实现函数`asm_my_function`，输出学号姓名

```
[bits 32]
global asm_my_function
asm_my_function:
    push eax
    xor eax, eax
    mov ah, 0x03 ;青色
    mov al, '2'
    mov [gs:2 * 0], ax
    mov al, '1'
    mov [gs:2 * 1], ax
    mov al, '3'
    mov [gs:2 * 2], ax
    mov al, '0'
    mov [gs:2 * 3], ax
    mov al, '7'
    mov [gs:2 * 4], ax
    mov al, '0'
    mov [gs:2 * 5], ax
    mov al, '3'
```

```
    mov [gs:2 * 6], ax
    mov al, '5'
    mov [gs:2 * 7], ax
    mov al, 'D'
    mov [gs:2 * 8], ax
    mov al, 'X'
    mov [gs:2 * 9], ax
    mov al, 'Y'
    mov [gs:2 * 10], ax
    pop eax
    ret
```

在文件`asm_utils.h`中声明所有的汇编函数，就不用单独地使用`extern`来声明，只需要`#include "asm_utils.h"`即可

```
#ifndef ASM_UTILS_H
#define ASM_UTILS_H

extern "C" void asm_my_function();

#endif
```

关于`Makefile`文件

1、文件路径

```
SRCDIR = ../src
RUNDIR = ../run
BUILDDIR = build
INCLUDE_PATH = ../include
```

2、编译MBR、bootloader，`-I`参数指定了头文件路径，`-f`指定了生成的文件格式是二进制的文件

```
ASM_COMPILER = nasm
mbr.bin : $(SRCDIR)/boot/mbr.asm
  $(ASM_COMPILER) -o mbr.bin -f bin -I$(INCLUDE_PATH)/ $(SRCDIR)/boot/mbr.asm

bootloader.bin : $(SRCDIR)/boot/bootloader.asm
  $(ASM_COMPILER) -o bootloader.bin -f bin -I$(INCLUDE_PATH)/
$(SRCDIR)/boot/bootloader.asm
```

3、编译内核的代码，将所有的代码（C/C++，汇编代码）都同一编译成可重定位文件，然后再链接成一个可执行文件。

编译`src/boot/entry.asm`和`src/utils/asm_utils.asm`

```
entry.obj : $(SRCDIR)/boot/entry.asm
  $(ASM_COMPILER) -o entry.obj -f elf32 $(SRCDIR)/boot/entry.asm

asm_utils.o : $(SRCDIR)/utils/asm_utils.asm
  $(ASM_COMPILER) -o asm_utils.o -f elf32 $(SRCDIR)/utils/asm_utils.asm
```

4、编译cpp文件`setup.cpp`

```
CXX_COMPLIER = i686-linux-gnu-g++
CXX_COMPLIER_FLAGS = -g -Wall -march=i386 -m32 -nostdlib -fno-builtin -
ffreestanding -fno-pic
$(CXX_OBJ):
  $(CXX_COMPLIER) $(CXX_COMPLIER_FLAGS) -I$(INCLUDE_PATH) -c $(CXX_SOURCE)
```

参数分析：

- `-O0`不开启编译优化。
- `-Wall`显示所有编译器警告信息
- `-march=i386`生成i386处理器下的`.o`文件格式。
- `-m32`生成32位的二进制文件。
- `-nostdlib -fno-builtin -ffreestanding -fno-pic`不要包含C的任何标准库。
- `-g`向生成的文件中加入debug信息供gdb使用。
- `-I`指定了代码需要的头文件的目录。
- `-c`生成可重定位文件。

5、链接生成的可重定位文件为两个文件：只包含代码的文件`kernel.bin`和可执行文件`kernel.o`

```
LINKER = i686-linux-gnu-ld
kernel.bin : kernel.o
  i686-linux-gnu-objcopy -O binary kernel.o kernel.bin

kernel.o : entry.obj $(OBJ)
  $(LINKER) -o kernel.o -melf_i386 -N entry.obj $(OBJ) -e enter_kernel -Ttext
0x00020000
```

使用`i686-linux-gnu-objcopy`来生成二进制文件`kernel.bin`：`objcopy`会从`kernel.o`中取出代码段，即可执行指令在ELF文件中的区域，放到`kernel.bin`中。`kernel.bin`从头到尾都是代码对应的机器指令，不再是`ELF`格式的。此时，我们将其加载到内存后，跳转执行即可。

参数分析：

- `-m`指定模拟器为i386`-melf_i386`
- `-N`不要进行页对齐
- `-Ttext`指定标号的起始地址0x00020000
- `-e`指定程序进入点`enter_kernel`
- `--oformat`指定输出文件格式

6、使用dd命令将`mbr.bin bootloader.bin kernel.bin`写入硬盘

```
build : mbr.bin bootloader.bin kernel.bin kernel.o
  dd if=mbr.bin of=$(RUNDIR)/hd.img bs=512 count=1 seek=0 conv=notrunc
  dd if=bootloader.bin of=$(RUNDIR)/hd.img bs=512 count=5 seek=1 conv=notrunc
  dd if=kernel.bin of=$(RUNDIR)/hd.img bs=512 count=145 seek=6 conv=notrunc
```

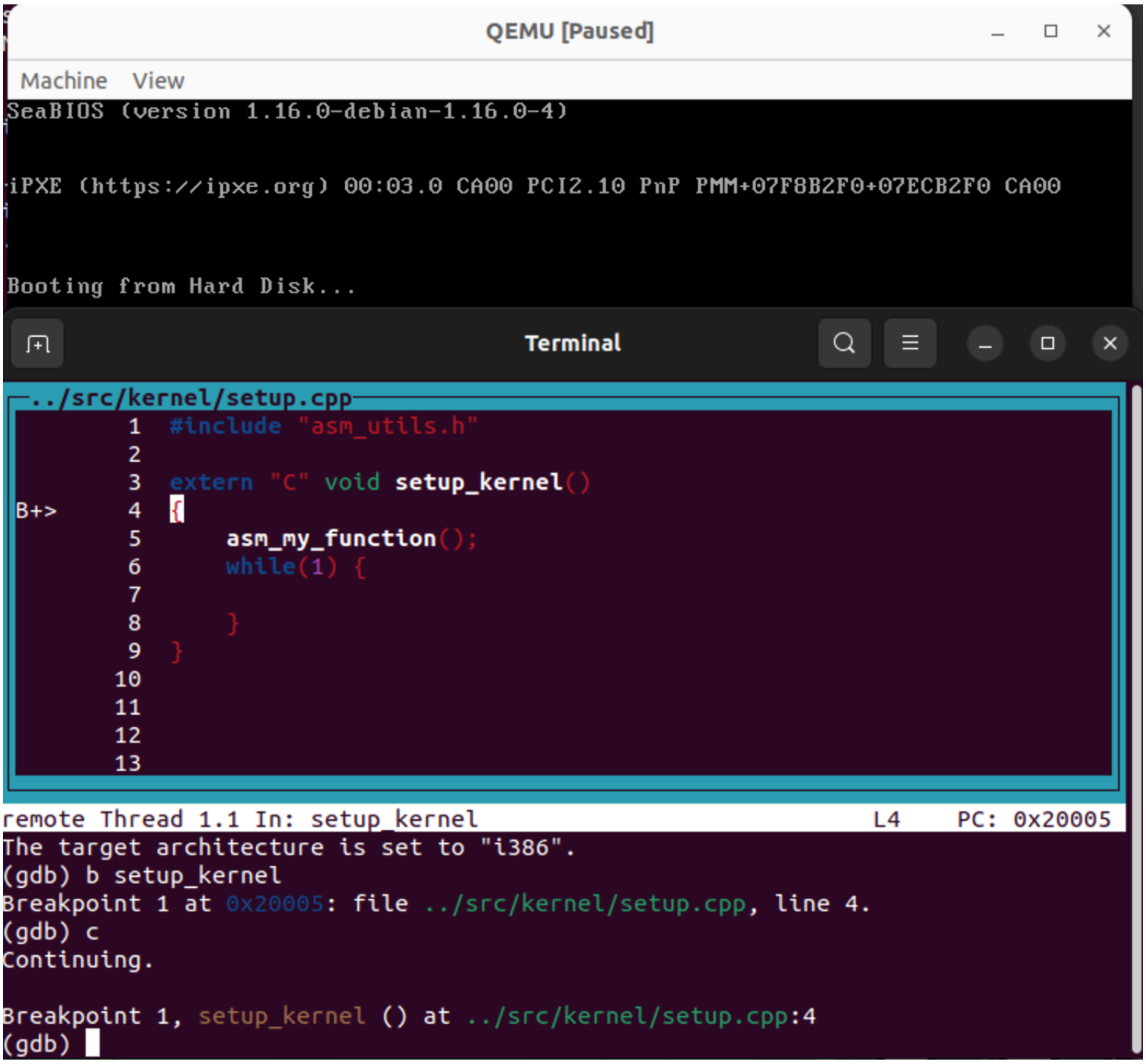
7、启动qemu

```
run:
	qemu-system-i386 -hda $(RUNDIR)/hd.img -serial null -parallel stdio -no-reboot
```

8、debug

出现无法set architecture i386的错误，需要将gdb改为gdb-multiarch

```
debug:
	qemu-system-i386 -S -s -parallel stdio -hda $(RUNDIR)/hd.img -serial null&
	@sleep 1
	gnome-terminal -e "gdb-multiarch -q -tui -x $(RUNDIR)/gdbinit"
```



QEMU

Machine View

21307035DXYrsion 1.16.0-debian-1.16.0-4)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B2F0+07ECB2F0 CA00

Booting from Hard Disk...

Terminal

../src/kernel/setup.cpp

```
        1  #include "asm_utils.h"
        2
        3  extern "C" void setup_kernel()
B+>     4  {
        5      asm_my_function();
        6      while(1) {
        7
        8      }
        9  }
       10
       11
       12
       13
       14
       15
       16
       17
       18
```

remote Thread 1.1 In: setup_kernel

```
The target architecture is set to "i386".
(gdb) b setup_kernel
Breakpoint 1 at 0x20005: file ../src/kernel/setup.cpp, line 4.
(gdb) c
Continuing.

Breakpoint 1, setup_kernel () at ../src/kernel/setup.cpp:4
(gdb) c
Continuing.
```

dengxy@dengxy-Parallels-ARM-Virtual-Machine: ~/lab4/5/b...

```
dengxy@dengxy-Parallels-ARM-Virtual-Machine:~/lab4/5/build$ make run
qemu-system-i386 -hda ../run/hd.img -serial null -parallel stdio -no-reboot
WARNING: Image format was not specified for '../run/hd.img' and probing guessed
raw.
         Automatically detecting the format is dangerous for raw images, write o
perations on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.
```

QEMU

Machine View

21307035DXYrsion 1.16.0-debian-1.16.0-4)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B2F0+07ECB2F0 CA00

Booting from Hard Disk...

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。

初始化中断描述符表IDT：

- 确定IDT的地址。
- 定义中断默认处理函数。
- 初始化256个中断描述符。

在`interrupt.h`中定义一个类`InterruptManager`中断管理器

类的实现在`interrupt.cpp`

```
#ifndef INTERRUPT_H
#define INTERRUPT_H

#include "os_type.h"

class InterruptManager
{
private:
    // IDT起始地址
    uint32 *IDT;

public:
    InterruptManager();
    // 初始化
    void initialize();
    // 设置中断描述符
    // index    第index个描述符，index=0, 1, ..., 255
    // address 中断处理程序的起始地址
    // DPL      中断描述符的特权级
    void setInterruptDescriptor(uint32 index, uint32 address, byte DPL);
};

#endif
```

初始化IDT

```
void InterruptManager::initialize()
{
    // 初始化IDT
    // 先设置IDTR，然后再初始化256个中断描述符
    IDT = (uint32 *)IDT_START_ADDRESS;
    asm_lidt(IDT_START_ADDRESS, 256 * 8 - 1);
    for (uint i = 0; i < 256; ++i)
    {
        setInterruptDescriptor(i, (uint32)asm_interrupt_empty_handler, 0);
    }
}
```

确定IDTR的32位基地址是`0x8880`，表界限是`2047`

> 不妨将IDT设定在地址`0x8880`处，即`IDT_START_ADDRESS=0x8880`
>
> ```
> #define IDT_START_ADDRESS 0x8880
> ```
>
> 为了使CPU能够找到IDT中的中断处理函数，需要将IDT的信息放置到寄存器IDTR中。当中断发生时，CPU会自动到IDTR中找到IDT的地址，然后根据中断向量号在IDT找到对应的中断描述符，最后跳转到中断描述符对应的函数中进行处理
>
> 有256个中断描述符，每个中断描述符的大小均为8字节，因此: 表界限=8*256−1=2047

初始化IDTR

使用指令`lidt`

```
lidt [tag]
```

将以`tag`为起始地址的48字节放入到寄存器IDTR中。我们需要在C代码中初始化IDT，而C语言的语法并未提供`lidt`语句，因此需要在汇编代码中实现能够将IDT的信息放入到IDTR的函数`asm_lidt`，

`asm_utils.asm`

```
; void asm_lidt(uint32 start, uint16 limit)
asm_lidt:
    push ebp
    mov ebp, esp
    push eax

    mov eax, [ebp + 4 * 3]
    mov [ASM_IDTR], ax
    mov eax, [ebp + 4 * 2]
    mov [ASM_IDTR + 2], eax
    lidt [ASM_IDTR]

    pop eax
    pop ebp
    ret
```

```
ASM_IDTR dw 0
         dd 0
```

插入256个默认的中断处理描述符到IDT中

对于中断描述符的几个定值

- P=1表示存在。
- D=1表示32位代码。
- DPL=0表示特权级0.
- 代码段选择子等于bootloader中的代码段选择子，也就是寻址4GB空间的代码段选择子。

不同的中断描述符的差别只在于中断处理程序在目标代码段中的偏移。在平坦模式下，也就是段起始地址从内存地址0开始，长度为4GB。此时函数名就是中断处理程序在目标代码段中的偏移。

在函数`InterruptManager::setInterruptDescriptor`中设置段描述符

```
// 设置中断描述符
// index   第index个描述符，index=0, 1, ..., 255
// address 中断处理程序的起始地址
// DPL     中断描述符的特权级
void InterruptManager::setInterruptDescriptor(uint32 index, uint32 address,
byte DPL)
{
    IDT[index * 2] = (CODE_SELECTOR << 16) | (address & 0xffff);
    IDT[index * 2 + 1] = (address & 0xffff0000) | (0x1 << 15) | (DPL << 13) |
(0xe << 8);
}
```

`IDT`是中断描述符表的起始地址指针，实际上可以认为中断描述符表就是一个数组。

在`InterruptManager`中，将变量`IDT`视作是一个`uint32`类型的数组。由于每个中断描述符的大小是两个`uint32`，因此第`index`个中断描述符是`IDT[2 * index],IDT[2 * index + 1]`。

定义一个默认的中断处理函数是`asm_interrupt_empty_handler`

`asm_utils.asm`

`asm_interrupt_empty_handler`首先关中断，然后输出提示字符串，最后做死循环。

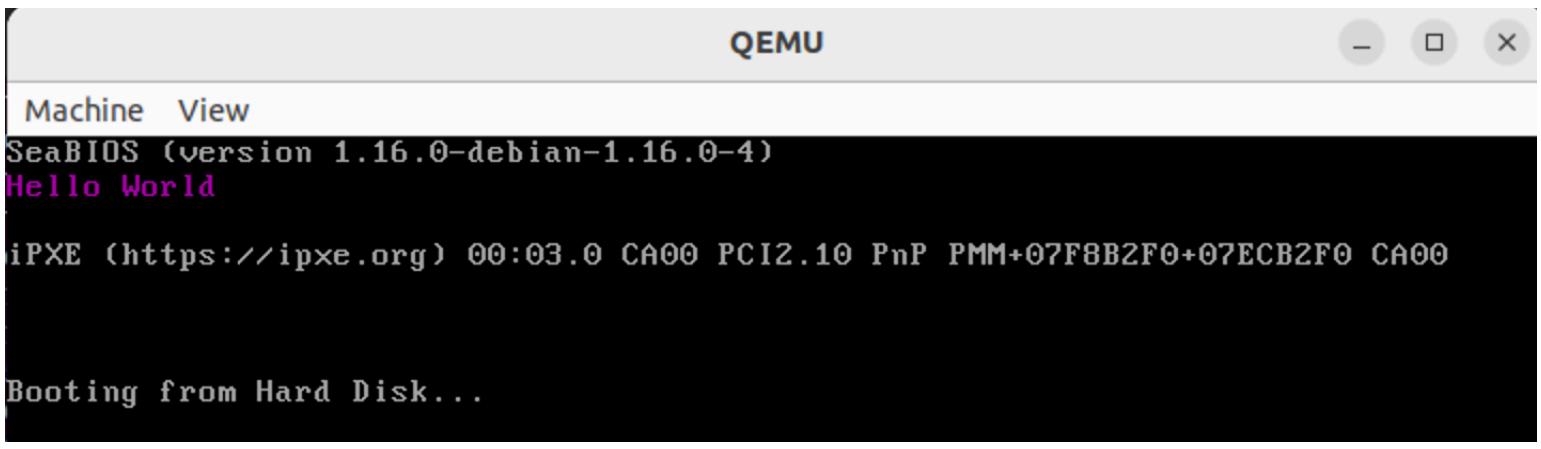
```
ASM_UNHANDLED_INTERRUPT_INFO db 'assignment 3:Unhandled interrupt happened,
halt...'
                             db 0

; void asm_unhandled_interrupt()
asm_unhandled_interrupt:
    cli
    mov esi, ASM_UNHANDLED_INTERRUPT_INFO
    xor ebx, ebx
    mov ah, 0x03
```

```
.output_information:
    cmp byte[esi], 0
    je .end
    mov al, byte[esi]
    mov word[gs:bx], ax
    inc esi
    add ebx, 2
    jmp .output_information
.end:
    jmp $
```

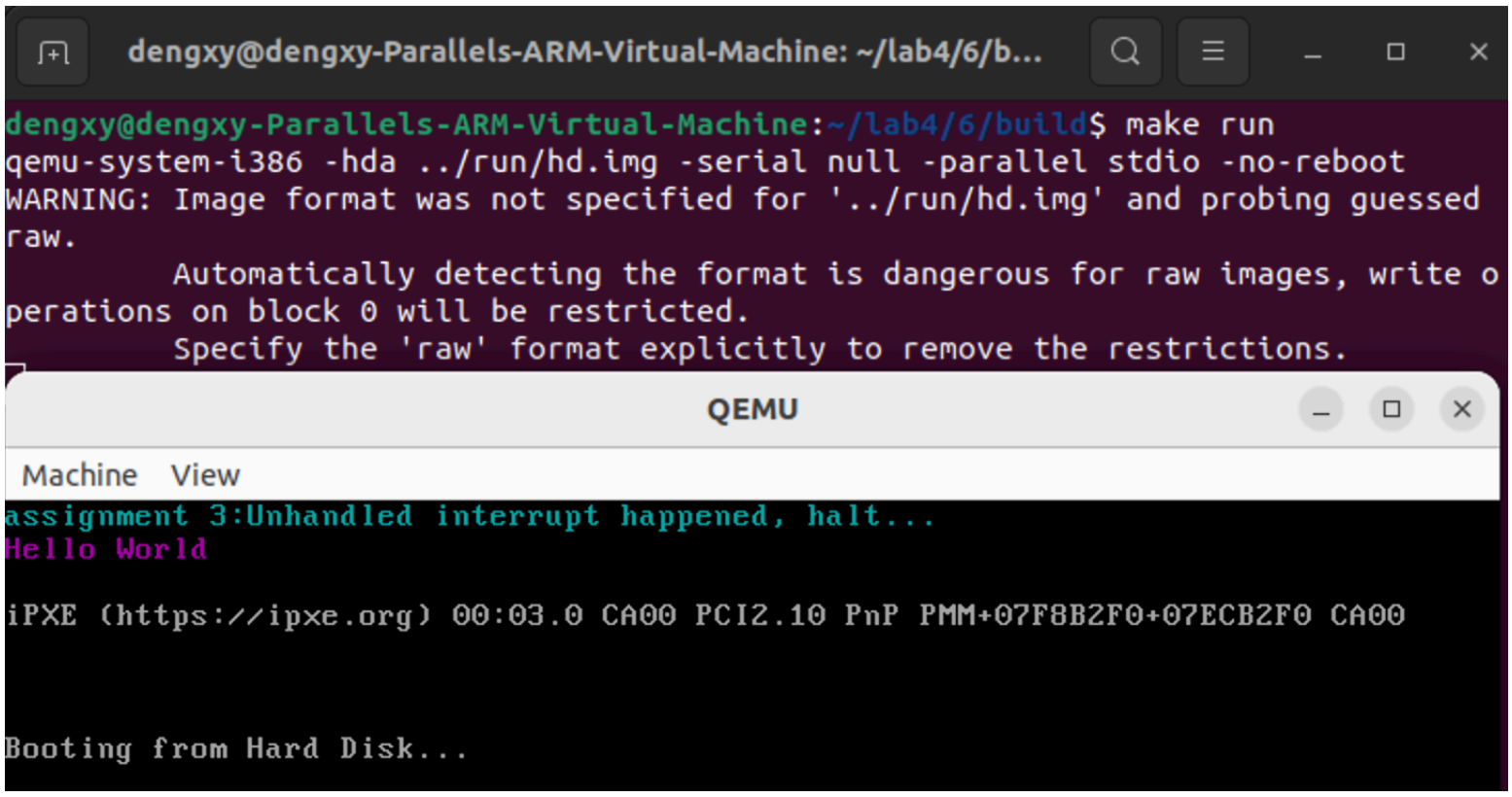
修改`setup_kernel.cpp`，先在（1,0）的位置开始输出hello world，再触发中断

```
#include "asm_utils.h"
#include "interrupt.h"

// 中断管理器
InterruptManager interruptManager;

extern "C" void setup_kernel()
{
    asm_hello_world();
    // 中断处理部件
    interruptManager.initialize();
    // 尝试触发除0错误
    int a = 1 / 0;
    // 死循环
    asm_halt();
}
```

当不触发除0错误时，只会执行`asm_hello_world`函数



尝试触发除0错误，可以发现`asm_unhandled_interrupt`正常工作，中断被触发



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。)

为中断控制器`InterruptManager`加入新的成员变量和函数

```
class InterruptManager
{
private:
    uint32 *IDT;                // IDT起始地址
    uint32 IRQ0_8259A_MASTER; // 主片中断起始向量号
    uint32 IRQ0_8259A_SLAVE;  // 从片中断起始向量号

public:
    InterruptManager();
    void initialize();
    // 设置中断描述符
    // index   第index个描述符，index=0, 1, ..., 255
    // address 中断处理程序的起始地址
    // DPL     中断描述符的特权级
    void setInterruptDescriptor(uint32 index, uint32 address, byte DPL);
    // 开启时钟中断
    void enableTimeInterrupt();
    // 禁止时钟中断
    void disableTimeInterrupt();
    // 设置时钟中断处理函数
    void setTimeInterrupt(void *handler);

private:
    // 初始化8259A芯片
```

```
    void initialize8259A();
};
```

对8259A芯片进行初始化

```
void InterruptManager::initialize8259A()
{
    // ICW 1
    asm_out_port(0x20, 0x11);
    asm_out_port(0xa0, 0x11);
    // ICW 2
    IRQ0_8259A_MASTER = 0x20;
    IRQ0_8259A_SLAVE = 0x28;
    asm_out_port(0x21, IRQ0_8259A_MASTER);
    asm_out_port(0xa1, IRQ0_8259A_SLAVE);
    // ICW 3
    asm_out_port(0x21, 4);
    asm_out_port(0xa1, 2);
    // ICW 4
    asm_out_port(0x21, 1);
    asm_out_port(0xa1, 1);
    // 由于未建立处理8259A中断的任何函数，因此在初始化的最后，需要屏蔽主片和从片的所有中断
    // OCW 1 屏蔽主片所有中断，但主片的IRQ2需要开启
    asm_out_port(0x21, 0xfb);
    // OCW 1 屏蔽从片所有中断
    asm_out_port(0xa1, 0xff);
}
```

`asm_out_port`是对`out`指令的封装

```
; void asm_out_port(uint16 port, uint8 value)
asm_out_port:
    push ebp
    mov ebp, esp

    push edx
    push eax

    mov edx, [ebp + 4 * 2] ; port
    mov eax, [ebp + 4 * 3] ; value
    out dx, al

    pop eax
    pop edx
    pop ebp
    ret
```

同理，`asm_in_port`是对`in`指令的封装

```
; void asm_in_port(uint16 port, uint8 *value)
```

```
asm_in_port:
    push ebp
    mov ebp, esp

    push edx
    push eax
    push ebx

    xor eax, eax
    mov edx, [ebp + 4 * 2] ; port
    mov ebx, [ebp + 4 * 3] ; *value

    in al, dx
    mov [ebx], al

    pop ebx
    pop eax
    pop edx
    pop ebp
    ret
```

处理时钟中断:主片的IRQ0中断

在计算机中，有一个称为8253的芯片，其能够以一定的频率来产生时钟中断。当其产生了时钟中断后，信号会被8259A截获，从而产生IRQ0中断。处理时钟中断并不需要了解8253芯片，只需要对8259A芯片产生的时钟中断进行处理即可，

步骤：

- 编写中断处理函数
- 设置主片IRQ0中断对应的中断描述符
- 开启时钟中断
- 开中断

编写一个处理屏幕输出的类`STDIO`

```
class STDIO
{
private:
    uint8 *screen;

public:
    STDIO();
    // 初始化函数
    void initialize();
    // 打印字符c，颜色color到位置(x,y)
    void print(uint x, uint y, uint8 c, uint8 color);
    // 打印字符c，颜色color到光标位置
    void print(uint8 c, uint8 color);
    // 打印字符c，颜色默认到光标位置
    void print(uint8 c);
```

```
    // 移动光标到一维位置
    void moveCursor(uint position);
    // 移动光标到二维位置
    void moveCursor(uint x, uint y);
    // 获取光标位置
    uint getCursor();

private:
    // 滚屏
    void rollUp();
};
```

`stdio.cpp`STDIO类的实现

初始化

```
void STDIO::initialize()
{
    screen = (uint8 *)0xb8000;
}
```

`print`向显存写入字符和颜色

```
void STDIO::print(uint x, uint y, uint8 c, uint8 color)
{
    if (x >= 25 || y >= 80)
    {
        return;
    }
    uint pos = x * 80 + y;
    screen[2 * pos] = c;
    screen[2 * pos + 1] = color;
}
void STDIO::print(uint8 c, uint8 color)
{
    uint cursor = getCursor();
    screen[2 * cursor] = c;
    screen[2 * cursor + 1] = color;
    cursor++;
    if (cursor == 25 * 80)
    {
        rollUp();
        cursor = 24 * 80;
    }
    moveCursor(cursor);
}
void STDIO::print(uint8 c)
{
    print(c, 0x07);
}
```

`cursor`光标处理

屏幕的像素为25*80，所以光标的位置从上到下，从左到右依次编号为0-1999，用16位表示。

与光标读写相关的端口为`0x3d4`和`0x3d5`，在对光标读写之前，我们需要向端口`0x3d4`写入数据，表明我们操作的是光标的低8位还是高8位。写入`0x0e`，表示操作的是高8位，写入`0x0f`表示操作的是低8位。如果我们需要需要读取光标，那么我们从`0x3d5`从读取数据；如果我们需要更改光标的位置，那么我们将光标的位置写入`0x3d5`。

```
void STDIO::moveCursor(uint position)
{
    if (position >= 80 * 25)
    {
        return;
    }
    uint8 temp;
    // 处理高8位
    temp = (position >> 8) & 0xff;
    asm_out_port(0x3d4, 0x0e);
    asm_out_port(0x3d5, temp);
    // 处理低8位
    temp = position & 0xff;
    asm_out_port(0x3d4, 0x0f);
    asm_out_port(0x3d5, temp);
}
uint STDIO::getCursor()
{
    uint pos;
    uint8 temp;
    pos = 0;
    temp = 0;
    // 处理高8位
    asm_out_port(0x3d4, 0x0e);
    asm_in_port(0x3d5, &temp);
    pos = ((uint)temp) << 8;
    // 处理低8位
    asm_out_port(0x3d4, 0x0f);
    asm_in_port(0x3d5, &temp);
    pos = pos | ((uint)temp);
    return pos;
}
```

`rollup`滚屏函数

如果我们需要向上滚屏，应当将光标放在`(24,0)`处

滚屏实际上就是将第2行的字符放到第1行，第3行的字符放到第2行，以此类推，最后第24行的字符放到了第23行，然后第24行清空，光标放在第24行的起始位置。

```
void STDIO::rollUp()
{
```

```
    uint length;
    length = 25 * 80;
    for (uint i = 80; i < length; ++i)
    {
        screen[2 * (i - 80)] = screen[2 * i];
        screen[2 * (i - 80) + 1] = screen[2 * i + 1];
    }

    for (uint i = 24 * 80; i < length; ++i)
    {
        screen[2 * i] = ' ';
        screen[2 * i + 1] = 0x07;
    }
}
```

编写中断处理函数

使用全局变量`times`来计算中断发生的次数，初始`times=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);
    }
    // 输出中断发生的次数
    ++times;
    char str[] = "interrupt happend: ";
    char number[10];
    int temp = times;

    // 将数字转换为字符串表示
    for(int i = 0; i < 10; ++i ) {
        if(temp) {
            number[i] = temp % 10 + '0';
        } else {
            number[i] = '0';
        }
        temp /= 10;
    }

    // 移动光标到(0,0)输出字符
    stdio.moveCursor(0);
    for(int i = 0; str[i]; ++i ) {
        stdio.print(str[i]);
    }
    // 输出中断发生的次数
    for( int i = 9; i > 0; --i ) {
```

```
        stdio.print(number[i]);
    }
}
```

设置时钟中断的中断描述符，即主片IRQ0中断对应的描述符

```
void InterruptManager::setTimeInterrupt(void *handler)
{
    setInterruptDescriptor(IRQ0_8259A_MASTER, (uint32)handler, 0);
}
```

开中断和关中断的函数

通过读取OCW1来得知8259A上的中断开启情况

修改8259A上的中断开启情况，需要先读取再写入对应的OCW1

```
void InterruptManager::enableTimeInterrupt()
{
    uint8 value;
    // 读入主片OCW
    asm_in_port(0x21, &value);
    // 开启主片时钟中断，置0开启
    value = value & 0xfe;
    asm_out_port(0x21, value);
}
void InterruptManager::disableTimeInterrupt()
{
    uint8 value;
    asm_in_port(0x21, &value);
    // 关闭时钟中断，置1关闭
    value = value | 0x01;
    asm_out_port(0x21, value);
}
```

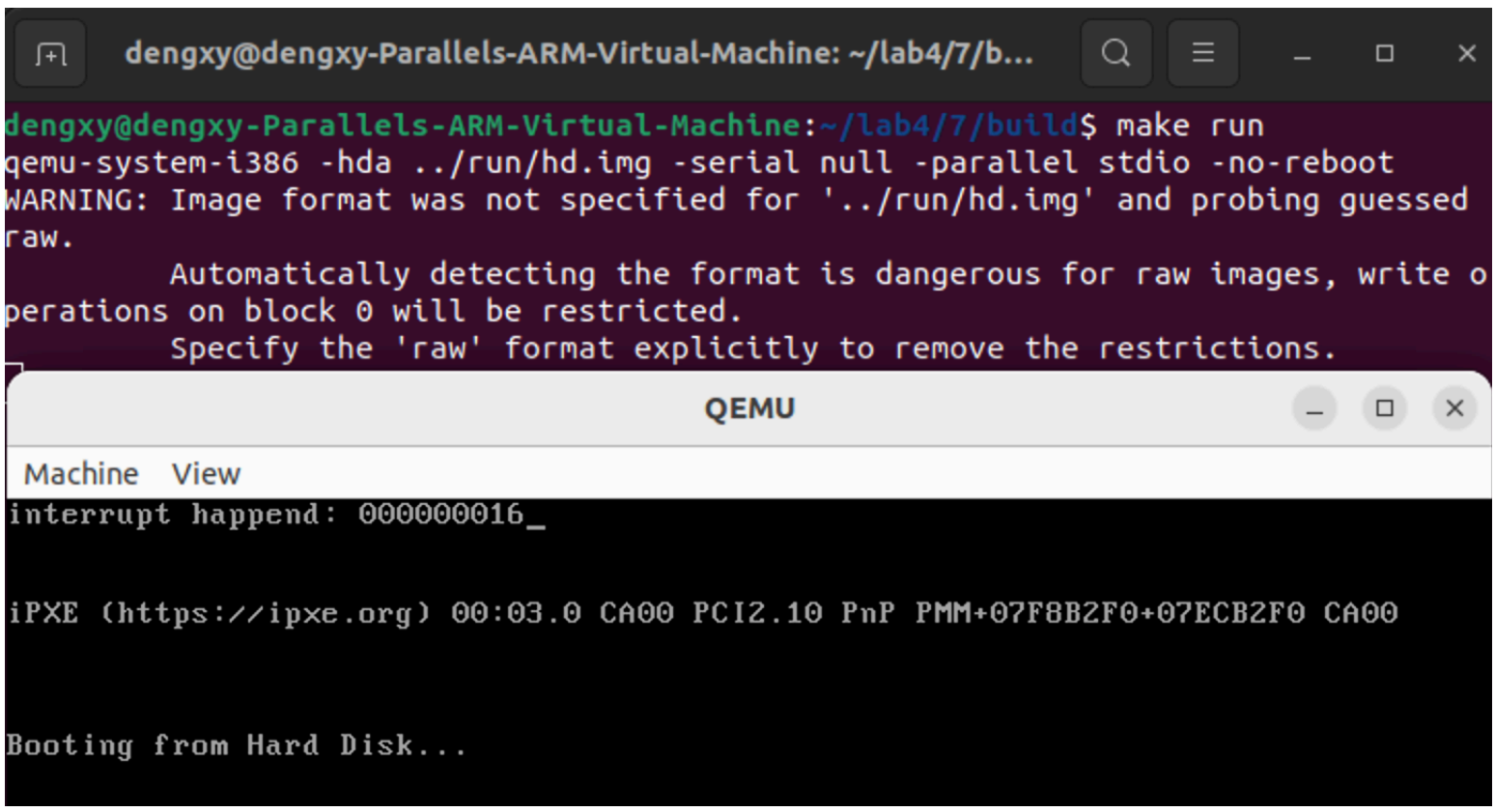
setup.cpp

```
// 屏幕IO处理器
// 定义STDIO的实例stdio
STDIO stdio;
// 中断管理器
InterruptManager interruptManager;
// 初始化内核的组件，然后开启时钟中断和开中断
extern "C" void setup_kernel()
{
    // 中断处理部件
    interruptManager.initialize();
    // 屏幕IO处理部件
    stdio.initialize();
    interruptManager.enableTimeInterrupt();
    interruptManager.setTimeInterrupt((void *)asm_time_interrupt_handler);
```

```
    asm_enable_interrupt();
    asm_halt();
}
```

开中断指令被封装在函数`asm_enable_interrupt`中

```
; void asm_enable_interrupt()
asm_enable_interrupt:
    sti
    ret
```



修改`interrupt.cpp`中的`c_time_interrupt_handler()`函数，在屏幕的第一行实现一个跑马灯来显示自己学号和英文名

```
// 输出跑马灯效果
    auto flag=0x01;
    char name[20]="21307035_DengXuying";
    if (times%3==0)
    {
        for(int i = 0; i<times%33;i++)
        {
            stdio.print(' ');
        }

        for(int i = 0; name[i]; i++)
        {
            stdio.print(name[i],flag+i%7);
        }
    }
    else if(times%3==1)
    {
        for (int i=0;i<(times-1)%33;i++)
        {
            stdio.print(' ');
```

```
        }
        for (int i=0;name[i];i++)
        {
            stdio.print(name[i],flag+i%7);
        }
    }
    else if(times%3==2)
    {
        for (int i=0;i<(times-2)%33;i++)
        {
            stdio.print(' ');
        }
        for (int i=0;name[i];i++)
        {
            stdio.print(name[i],flag+i%7);
        }
    }
}
```

```
interrupt happend: 000000011          21307035_DengXuying_

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B2F0+07ECB2F0 CA00


Booting from Hard Disk...
```

```
interrupt happend: 0000000921307035_DengXuying_

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B2F0+07ECB2F0 CA00


Booting from Hard Disk...
```

3、总结

- 在本次实验中，了解学习了C代码是如何通过预编译、编译、汇编和链接生成最终的可执行文件，并使用makefile来管理C/C++项目。
- 深入了解了如何在C代码中调用汇编代码编写的函数和如何在汇编代码中调用使用C编写的函数。
- 学习了保护模式下的中断处理机制和可编程中断部件8259A芯片，理解保护模式的中断处理机制和处理时钟中断，为后面的二级分页机制和多线程/进程打下基础。