北京大学

PEKING UNIVERSITY

XV6 中断机制源码阅读报告

组长：	罗登 2001210364
组员 1：	毕廷竹 2001210186
组员 2：	周旭敏 2001210723
组员 3：	罗旭坤 2001210368

基本问题回答

1. 什么是用户态和内核态？两者有何区别？什么是中断和系统调用？两者之间有何区别？计算机在运行时，是如何确定当前处于用户态还是内核态的？

- **用户态和内核态**

用户态和内核态是程序运行时 CPU 的两种状态，分别执行两种不同性质的程序：用户自编程序和操作系统内核程序。

CPU 中存在两类寄存器，分别是用户可见寄存器以及控制和状态寄存器。出于安全的考虑，访问和修改控制和状态寄存器的指令不应该被用户自编程序随意执行。于是操作系统将 CPU 状态划分为用户态和内核态，处于内核态时 CPU 能够执行设计控制和状态寄存器的特权指令，而处于用户态时 CPU 则不能执行特权指令。

x86 系统中 CPU 存在四种状态（特权级），分别是 R0-R3。R0 相当于内核态，R3 相当于用户态。

- **中断和系统调用**

中断，指来自 CPU 执行指令以外的事件的发生，它们通常与当前 CPU 运行的程序无关，引入的目的是为了支持 CPU 和设备之间的并行操作。

系统调用，是用户在编程时可以调用的操作系统功能，是操作系统提供给用户程序使 CPU 状态从用户态陷入内核态的唯一接口，它是异常的一种。

中断是与当前运行程序无关的事件，部分中断（可屏蔽中断）可以被操作系统通过关中断操作屏蔽。系统调用源自 CPU 执行指令内部的事件，是程序通过陷入指令主动使 CPU 进入内核态，无法被屏蔽。

- **CPU 状态的确定**

计算机在运行时，可以根据程序状态字寄存器 PSW 中相应的位，判断 CPU 当前处于用户态还是内核态。如图 1 所示，X86 架构中的程序状态字寄存器为 EFLAGS，其中 IOPL 这两位表示了 CPU 当前的特权级。当 IOPL 为 00 时表示 CPU 处于内核态，当 IOPL 为 11 时表示 CPU 处于用户态。

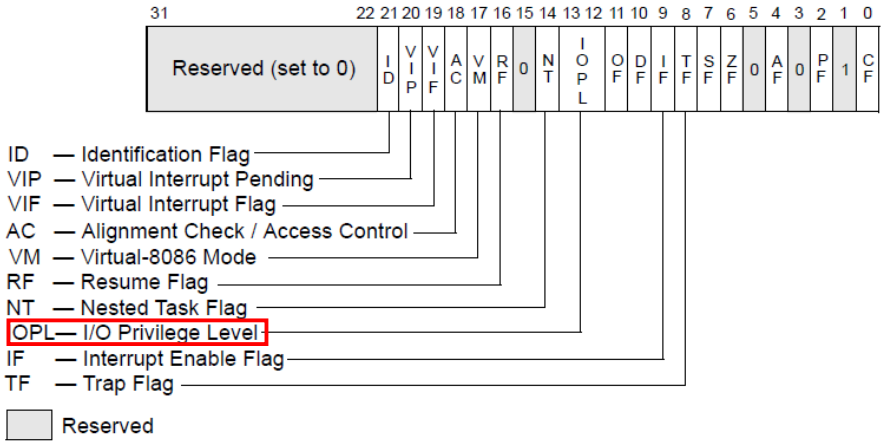


图1　EFLAGS 寄存器各个位置功能

2. 计算机开始运行阶段就有中断吗？XV6 的中断管理是如何初始化的？XV6 是如何实现内核态到用户态的转变的？XV6 的硬件中断是如何开关的？实际的计算机里，中断有哪几位？

- **BIOS 支持中断**

在计算机启动时，执行存放在非易失存储器中的 BIOS。BIOS 作为一个小型操作系统，为了初始化硬件设备，可能设置了自己的中断处理程序，所以计算机在开始运行阶段便存在 BIOS 支持的中断。

- **XV6 中断管理初始化**

XV6 的中断管理初始化主要包括以下步骤：

1) 计算机启动时运行 BIOS，BIOS 中存在一些自身支持的中断。
2) BIOS 将控制权交给从引导扇区加载的代码，即 bootloader（包含 bootasm.S 和 bootmain.c）。进入 bootasm.S 后第一条执行的指令便是屏蔽中断（如图 2 所示），原因是：当前 BIOS 已经失去了控制权，并且此时操作系统还未定义各种中断和初始化中断处理程序，故无法响应各种中断事件。

```
1  #include "asm.h"
2  #include "memlayout.h"
3  #include "mmu.h"
4
5  # Start the first CPU: switch to 32-bit protected mode, jump into C.
6  # The BIOS loads this code from the first sector of the hard disk into
7  # memory at physical address 0x7c00 and starts executing in real mode
8  # with %cs=0 %ip=7c00.
9
10 .code16                       # Assemble for 16-bit mode
11 .globl start
12 start:
13   cli                         # BIOS enabled interrupts; disable
14
```

图2　bootasm.S 关中断操作

3) bootasm.S 调用 bootmain.c 中的 bootmain()方法，从磁盘第二个扇区开头读入内核程序（entry.S），之后跳转到 main.c 的 main()方法，main()中调用了各种系统初始化方法。
4) 如图 3 所示，main()方法中与中断管理初始化相关的方法及其作用如下（具体逻辑将在“源代码阅读”部分提及）：
   - ✧ lapicinit()：初始化局部高级可编程中断控制器（Local APIC）。
   - ✧ picinit()：禁用 8259A 中断控制器。
   - ✧ ioapicinit()：初始化全局高级可编程中断控制器（I/O APIC）。
   - ✧ consoleinit()：初始化控制台相关中断。
   - ✧ uartinit()：设置通用异步接收器/发送器（UART）中断。
   - ✧ tvinit()：初始化中断描述符表（IDT）。
   - ✧ mpmain()：将 IDT 基址写入描述符表寄存器（IDTR），开启中断。
   - ✧ start_others()：主 CPU 通过初始化号的 Local APIC 发送中断信号使其他 CPU 开始初始化操作，其中包括各自中断管理的初始化。

```
17  int
18  main(void)
19  {
20    kinit1(end, P2V(4*1024*1024)); // phys page allocator
21    kvmalloc();      // kernel page table
22    mpinit();        // detect other processors
23    lapicinit();     // interrupt controller
24    seginit();       // segment descriptors
25    picinit();       // disable pic
26    ioapicinit();    // another interrupt controller
27    consoleinit();   // console hardware
28    uartinit();      // serial port
29    pinit();         // process table
30    tvinit();        // trap vectors
31    binit();         // buffer cache
32    fileinit();      // file table
33    ideinit();       // disk
34    startothers();   // start other processors
35    kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
36    userinit();      // first user process
37    mpmain();        // finish this processor's setup
38  }
```

图3　mian.c 文件中的 main()方法

- **内核态到用户态**

此处将 XV6 中断管理初始化后第一个用户进程的创建和运行为例说明 XV6 如何实现 CPU 从内核态到用户态的转变。XV6 系统在完成资源及中断管理的初始化后，在 main()中调用 uerinit()初始化第一个用户态进程的信息。userinit()设置第一个进程的 trap frame, 其中 cs、ds、es、ss 中相应特权级位均设为 DLP_USER（用户模式）。在执行完 forkret 和 trapret 后，系统已将 trap frame 中的内容推送到相应的寄存器。此时，进程的用户代码运行在用户态下，只能使用带有 PTE_U 设置的页，而且无法修改像 %cr3 这样的敏感的硬件寄存器（allocuvm()方法中设置标志位为 PTE_U 表示允许用户代码访问的内存，以限制用户态下 CPU 可访问的内存地址）。

- **XV6 开关中断**

XV6 中可以通过汇编指令 cli 关中断，sti 开中断。在硬件实现上，XV6 通过设置 EFLAGS 寄存器中的 IF 位来控制中断的开关，1 表示开中断，0 表示关中断。

- **中断的分类**

中断（这里理解为课上的事件）可以分为外中断和内中断。外中断是外部设备产生的事件，包括：I/O 中断，时钟中断和硬件故障等。内中断是程序运行过程中产生的事件，包括系统调用、页错误、保护性异常，断点指令和其他程序性异常（如算数溢出）。

3. 什么是中断描述符，中断描述符表？在 XV6 里是用什么数据结构表示的？

- **中断描述符与中断描述符表**

在实模式下，操作系统使用中断向量（4 字节）存放中断处理程序的入口地址。但是，在保护模式下，4 字节的表项无法满足要求（至少 2 字节的段描述符和 4 字节的偏移量），于是使用 8 字节的表项，改称中断描述符。中断描述符表（Interrupt Descriptor Table，IDT）将每个异常分别与他们的处理过程联系起来，每个表项均为一个中断描述符。

- **XV6 中的实现**

在 XV6 系统中，中断描述符使用 mmu.h 文件中的 gatedesc 结构表示。如图 4 所示，中断描述符共 64 位，其中包括 32 位的中断处理程序偏移地址、16 的代码段选择符、2 位的门特权级和 4 位的门类型等。

```
147  // Gate descriptors for interrupts and traps
148  struct gatedesc {
149    uint off_15_0 : 16;   // low 16 bits of offset in segment
150    uint cs : 16;         // code segment selector
151    uint args : 5;        // # args, 0 for interrupt/trap gates
152    uint rsv1 : 3;        // reserved(should be zero I guess)
153    uint type : 4;        // type(STS_{IG32,TG32})
154    uint s : 1;           // must be 0 (system)
155    uint dpl : 2;         // descriptor(meaning new) privilege level
156    uint p : 1;           // Present
157    uint off_31_16 : 16;  // high bits of offset in segment
158  };
```

图 4　XV6 中断描述符定义

在 XV6 系统中，中断描述符表在 trap.c 中有所定义，是一个 gatedesc 数组，大小为 256，如图 5 所示。

```
11  // Interrupt descriptor table (shared by all CPUs).
12  struct gatedesc idt[256];
13  extern uint vectors[];  // in vectors.S: array of 256 entry pointers
14  struct spinlock tickslock;
15  uint ticks;
```

图 5　XV6 中断描述符表定义

4. 请以某一个中断（如除零、页错误等）为例，详细描述 XV6 一次中断的处理过程。包括：涉及哪些文件的代码？如何跳转？内核态，用户态如何变化？涉及哪些数据结构等等。

本报告以除零异常为例，下面详细描述 XV6 发生一次除零异常后的处理过程。

开始时，CPU 处于用户态执行用户程序中的一条条指令。在某个指令周期内，CPU 执行指令发现除数为 0，产生异常。系统检查到发生除零异常（若为外部中断，中断信号由中断控制器产生；若为内部中断，中断信号由系统自身产生。此处的除零异常为内部中断），则产生除零异常对应的系统调用号，进行下一步的准备操作。

处理器从任务段描述符中加载内核的%esp 和%ss 以切换内核栈，将旧的%ss 和%esp 压入新栈，在内核栈中保存被中断程序的重要上下文环境，主要是%eip、%cs 和%eflags，对于一部分中断类型还可能压入错误字。然后，CPU 从相应 IDT 表项加载新的%eip 和%cs（%eip 的读取涉及到中断初始化中 tvinit()的逻辑，具体说明见“源代码阅读”部分）。

如图 6 所示，新%eip 的获取方式是：首先，通过 IDTR 得到 IDT 基址；第二，通过中断号得到 IDT 中对应项，得到段选择符和 32 位偏移；第三，通过 GDTR 得到 GDT 基址，使用段选择符作为索引得到段描述符；最后，将段描述符中的段基址和 IDT 中的偏移组合，得到中断程序入口地址。

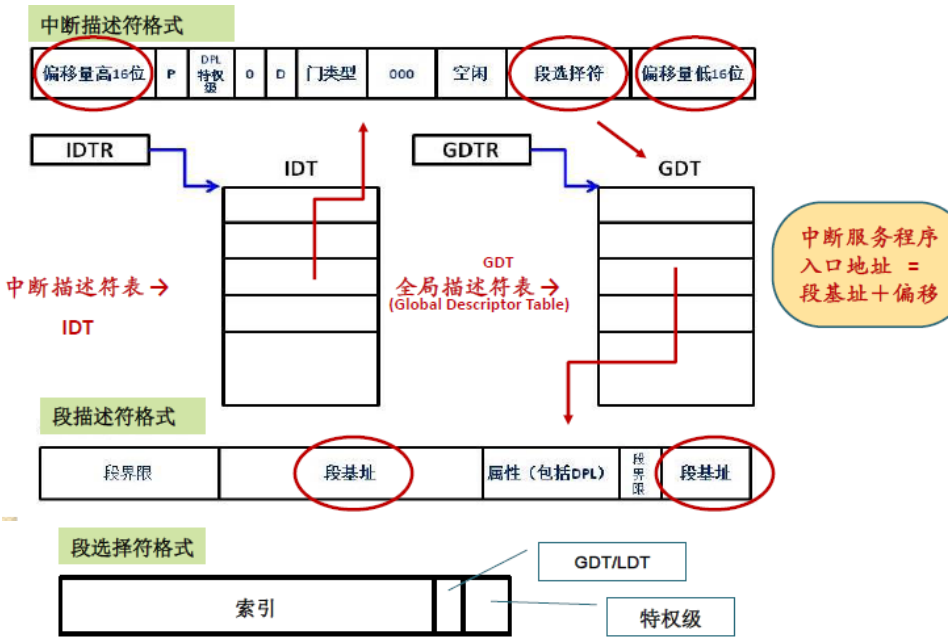
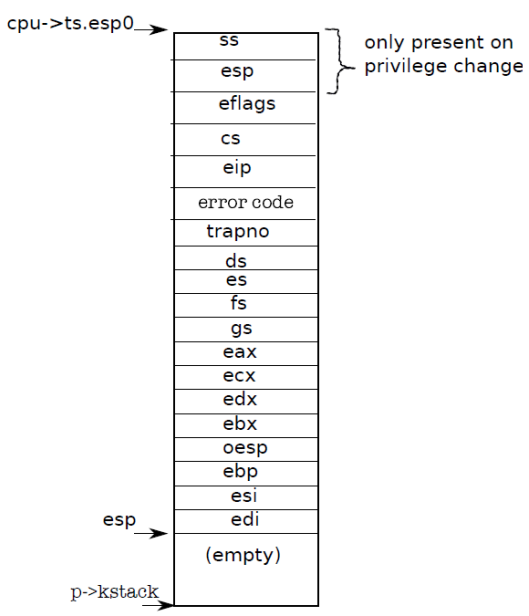


图6　通过中断描述符获取新 PC

CPU 按照%eip 中的指令地址从相应中断处理程序地址取指，执行压入错误字（若 CPU 没有压入错误字）和中断号操作，之后跳转到 trapasm.S 中的 alltraps。

alltraps 继续压入寄存器%ds、%es、%fs，%gs 和通用寄存器，此时栈中压入了一个 trap frame 结构（定义于 x86.h）。图 7 是此时内核栈与 trap frame 结构的对比，可以看出 XV6 系统保存在栈中的上下文环境结构与 trap frame 相同。之后系统继续设置数据段，压入栈指针，此时，CPU 完成了从用户态到内核态的转化。完成 CPU 状态的转换后，调用 trap.c 中的 trap() 函数，开始进行中断的相应处理。



7(a)　内核栈

```
151   struct trapframe {
152     // registers as pushed by pusha
153     uint edi;
154     uint esi;
155     uint ebp;
156     uint oesp;      // useless & ignored
157     uint ebx;
158     uint edx;
159     uint ecx;
160     uint eax;
161     // rest of trap frame
162     ushort gs;
163     ushort padding1;
164     ushort fs;
165     ushort padding2;
166     ushort es;
167     ushort padding3;
168     ushort ds;
169     ushort padding4;
170     uint trapno;
171     // below here defined by x86 hardware
172     uint err;
173     uint eip;
174     ushort cs;
175     ushort padding5;
176     uint eflags;
177     // below here only when crossing rings
178     uint esp;
179     ushort ss;
180     ushort padding6;
181   };
```

7(b)　trap frame

图7　内核栈结构与 trap frame 结构对比

trap()的参数便是之前在内核栈上建立好的 trap frame。系统首先根据 trapno 判断该中断是不是系统调用，再判断该中断是不是硬件中断。

由于本次发生的是除零异常，并不属于系统调用或硬件中断，于是将该进程的 killed 设为 1，根据判断调用 exit()做必须的退出处理，并调用 sched()进行新进程的调度。

当新进程被调度上 CPU，新进程的 trap frame 被推入各个寄存器。若此进程是用户进程，则 CPU 从内核态转化回用户态。

5. 请以系统调用 setrlimit（该系统调用的作用是设置资源使用限制）为例，叙述如何在 XV6 中实现一个系统调用。（提示：需要添加系统调用号、系统调用函数、用户接口等等）。

- **系统调用处理逻辑**

首先，考虑当发生一个系统调用时，XV6 的处理过程：在调用 trap.c 中的 trap()函数之前，硬件和操作系统的处理方式与除零异常时相同。在调用 trap()函数后：系统根据 trapno 判断该中断属于系统调用，于是调用 syscall.c 中的 syscall()函数。

系统从 trap frame 的 eax 中取出系统调用号，根据系统调用号在 syscalls 表中得到相应的系统调用函数地址，并开始执行。

与系统调用相关的文件有：

- ✧ syscall.h：定义系统调用号。
- ✧ syscall.c：定义 syscalls 表。
- ✧ sysproc.c：具体实现系统调用处理函数。
- ✧ user.h：定义系统调用处理相应的接口。
- ✧ usys.S：定义各种系统调用操作的宏定义。

- **实现 setrlimit**

在上述的系统调用相关文件中添加系统调用相应的系统调用号和调用函数等，具体的修改如下：

- ✧ 在 syscall.h 文件中添加系统调用号 22，将其作为 setrlimit 的系统调用号，如图 8 所示。

```
19  #define SYS_unlink 18
20  #define SYS_link   19
21  #define SYS_mkdir  20
22  #define SYS_close  21
23  // For setrlimit
24  #define SYS_setrlimit 22
```

图 8　syscall.h

- ✧ 在 syscall.c 文件的 syscalls 表中添加系统调用号与系统调用处理函数的映射。如图 9 所示。

```
107  static int (*syscalls[])(void) = {
108  [SYS_fork]    sys_fork,
109  [SYS_exit]    sys_exit,
110  [SYS_wait]    sys_wait,
111  [SYS_pipe]    sys_pipe,
112  [SYS_read]    sys_read,
113  [SYS_kill]    sys_kill,
114  [SYS_exec]    sys_exec,
115  [SYS_fstat]   sys_fstat,
116  [SYS_chdir]   sys_chdir,
117  [SYS_dup]     sys_dup,
118  [SYS_getpid]  sys_getpid,
119  [SYS_sbrk]    sys_sbrk,
120  [SYS_sleep]   sys_sleep,
121  [SYS_uptime]  sys_uptime,
122  [SYS_open]    sys_open,
123  [SYS_write]   sys_write,
124  [SYS_mknod]   sys_mknod,
125  [SYS_unlink]  sys_unlink,
126  [SYS_link]    sys_link,
127  [SYS_mkdir]   sys_mkdir,
128  [SYS_close]   sys_close,
129  [SYS_setrlimit] sys_setrlimit,
130  };
```

图9　syscall.c

✧　在 sysproc.c 文件中添加系统调用处理函数，如图 10 所示。

```
16  // For setrlimit
17  int
18  sys_setrlimit(void)
19  {
20    // TODO: 设置资源使用限制
21    return setrlimit();
22  }
23
```

图10　sysproc.c

✧　在 user.h 文件中添加系统调用处理相应接口的定义，并在合适的文件中加以实现，如图 11 所示。

✧　在 usys.S 文件中添加 setrlimit 系统调用的宏定义，如图 12 所示。

```
23  char* sbrk(int);
24  int sleep(int);
25  int uptime(void);
26  int setrlimit(void);
```

图11　user.h

```
4   #define SYSCALL(name) \
5     .globl name; \
6     name: \
7       movl $SYS_ ## name, %eax; \
8       int $T_SYSCALL; \
9       ret
10
11  SYSCALL(setrlimit)
12  SYSCALL(fork)
13  SYSCALL(exit)
```

图12　usys.S

源代码阅读

1. 启动部分：Bootloader：bootasm.S bootmain.c

计算机开始运行时先执行 BIOS，之后将控制权交给从引导扇区加载的代码，即 bootloader（包含 bootasm.S 和 bootmain.c）。bootasm.S 是由 16 位和 32 位汇编混合编写成的 XV6 引导加载器。此时 BIOS 已经失去了控制权，并且此时操作系统还未定义各种中断和初始化中断处理程序，故进入 bootasm.S 后第一条执行的指令便是屏蔽中断。

bootasm.S 调用 bootmain.c 中的 bootmain()方法，从磁盘第二个扇区开头读入内核程序（entry.S），之后跳转到 main.c 的 main()方法，其中调用了各种中断初始化方法。

2. 初始化模块：main.c entryother.S

main.c 中的 main()方法中调用了各种中断初始化方法，还调用了 startothers()发送 Local APIC 中断信号使其他 CPU 开始初始化。其他 Local APIC 之后发出中断信号给自身连接的 CPU，使他们运行 entryother.S 文件中的汇编指令进行初始化，然后跳转到 main.c 文件中的 mpenter()方法进行各自中断的初始化。mpenter()方法调用 lapicinit()进一步初始化相应的 Local APIC，调用 mpmain()初始化中断描述符表并告诉主 CPU 初始化完成，然后开中断，开始调度。

3. 中断与系统调用模块

本模块代码根据 main.c 文件中的 main()方法所调用的初始化方法涉及的文件进行讲解。

- **picinit()：picirq.c**

picinit()的作用是禁用 8259A 中断控制器，要想理解这个函数，需要先了解几个问题。

1) 为什么要禁用 8259A 中断控制器？

XV6 支持对称多处理器架构（Symmetric Multiprocessing，SMP）。传统的 i386 处理器采用 8259A 中断控制器。一般而言，8259A 的作用是提供多个外部中断源与单一 CPU 间的连接。如果在 SMP 结构中还是采用 8259A 中断控制器，就只能静态地把所有的外部中断源划分成若干组，分别把每一组都连接到一个 8259A，而 8259A 则与 CPU 一对一连接，这样就达不到动态分配中断请求的目的。Intel 为对称多处理器架构设计了一种更为通用的中断控制器，称为 APIC：考虑到“处理器间中断请求”的需要，每个 CPU 还要有本地的 APIC(Local APIC)，因为 CPU 常常要有目标地向系统中的其他 CPU 发出中断请求；另外，在 SMP 结构中还需要一个外部的、全局的 APIC(I/O APIC)。这样，APIC 在对处理器架构中的结构如图 13 所示，每个 CPU 都对应一个 Local APIC，还有一个全局的 I/O APIC 通过 bus 与 Local APIC 相连。

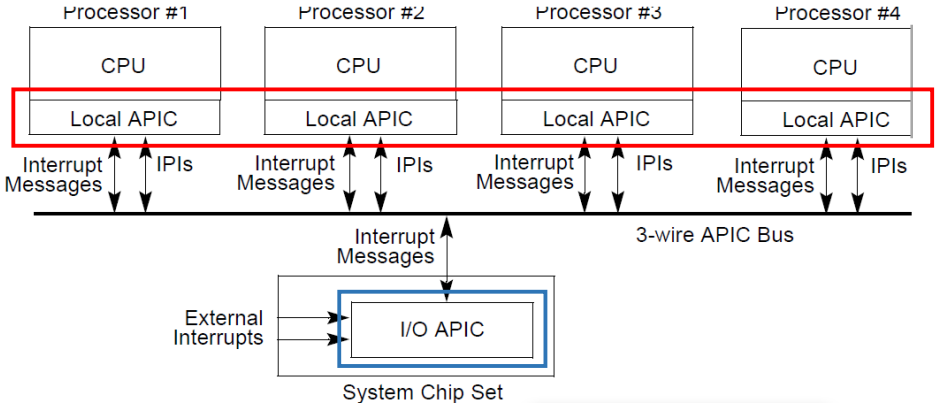


图 13　APIC（Local APIC & I/O APIC）

2) 如何禁用 8259A 中断控制器？

系统往地址线 A0=1 传送操作命令字 OCW1（如图 14 所示）对中断寄存器进行写操作（此时需要传递操作字的端口号主芯片为 0x21、从芯片为 0xA1）。若 Mi = 1，则屏蔽对应的中断请求级（共 8 个），将 OCW1 设为全 1 即可屏蔽 8259A 产生的所有中断。

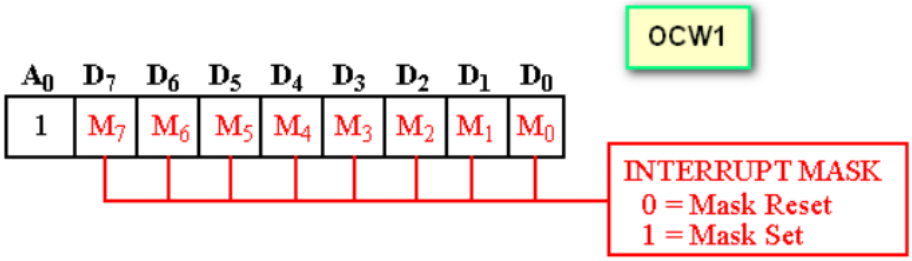


图 14　OCW1

3) XV6 使用什么指令访问 8259A 中断控制器？

操作系统访问中断控制器的方式有两种：MMIO（Memory-mapped I/O）和 PMIO（Port-mapped I/O）。

MMIO 使用相同的地址总线来处理内存和 I/O 设备，I/O 设备的内存和寄存器被映射到与之相关联的地址。当 CPU 访问某个内存地址时，它可能是物理内存，也可以是某个 I/O 设

备的内存。因此，用于访问内存的 CPU 指令也可来访问 I/O 设备。

PMIO 的思想则是内存和 I/O 设备有各自的地址空间。端口映射 I/O 通常使用一种特殊的 CPU 指令，专门执行 I/O 操作。在 Intel 的微处理器中，使用的指令是 IN 和 OUT。

8259A 中断控制器的映射方式是 PMIO，故使用 IN 和 OUT 指令来进行对 8259A 中断控制器的访问。

picirq.c 中定义的两个端口 0x20 和 0xA0，分别是主 8259A 和从 8259A 的映射端口。picinit()方法使用 outb 往两个映射端口的 1 号地址线（即 0x21 和 0xA1）发送 8 位全 1 的 OCW1 命令字（0xFF），屏蔽了两个 8259A 中断控制器的所有中断信号。

- **lapicinit()：lapic.c mp.c mp.h**

如图 13 所示，Local APIC，每个 CPU 都对应一个，里面包括各种保存本地中断信息的寄存器（Local Vector Table）和定时器等。Local APIC 能够接收的中断来源有：本地连接 I/O 设备（LINT0 和 LINT1，连接系统芯片）、外部连接 I/O 设备、其他 CPU 发出的中断（IPIs）、APIC 时钟中断（timer）、APIC 内部错误引发的中断，性能计数器 Overflow 产生的中断和温度传感器产生的中断。

值得注意的是，Local APIC 使用的是 MMIO 的方式，Local Vector Table 会被映射到以 0xFEE00000H 为起始地址的虚拟内存空间中，且每一项为 4 字节。于是 XV6 系统能够基于基址通过数组访问的方式读写 Local Vector Table 中的每一项。

如图 15 所示，lapic.c 起始的大量宏定义便是 Local Vector Table 中相应表项的数组索引，而*lapic 则是在 mp.c 文件中 mpinit()方法从配置文件中读出的 Local Vector Table 基地址，即 0xFEE00000H（配置文件的描述结构定义与 mp.h 文件中）。

```
35  #define PCINT   (0x0340/4)   // Performance Counter LVT
36  #define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
37  #define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
38  #define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
39    #define MASKED     0x00010000   // Interrupt masked
40  #define TICR    (0x0380/4)   // Timer Initial Count
41  #define TCCR    (0x0390/4)   // Timer Current Count
42  #define TDCR    (0x03E0/4)   // Timer Divide Configuration
43
44  volatile uint *lapic;  // Initialized in mp.c
```

图 15　lapic.c

lapicinit()方法通过向 Local Vector Table 中相应项写入初始化信息以启动该 CPU 对应 Local APIC 的功能，并广播消息至其他 Local APIC，使它们进行简单的初始化，为之后的 startothers()方法中唤醒其他 CPU 做准备。

- **ioapicinit()：ioapic.c**

如图 13 所示，I/O APIC 是一个全局的 APIC，负责接收外部 I/O 设备发来的中断。I/O APIC 中包含的寄存器有：表示 I/O APIC 的物理名称的 I/O APIC ID（32 bits）、表示 I/O APIC 的版本信息的 I/O APIC Version（32 bits）、中断信息寄存器表 Redirection Table（每个表项项 64bits，称为 RTE，配置并记录一个 I/O 设备的中断信息）。图 14 是一种型号的 I/O APIC 示意图。

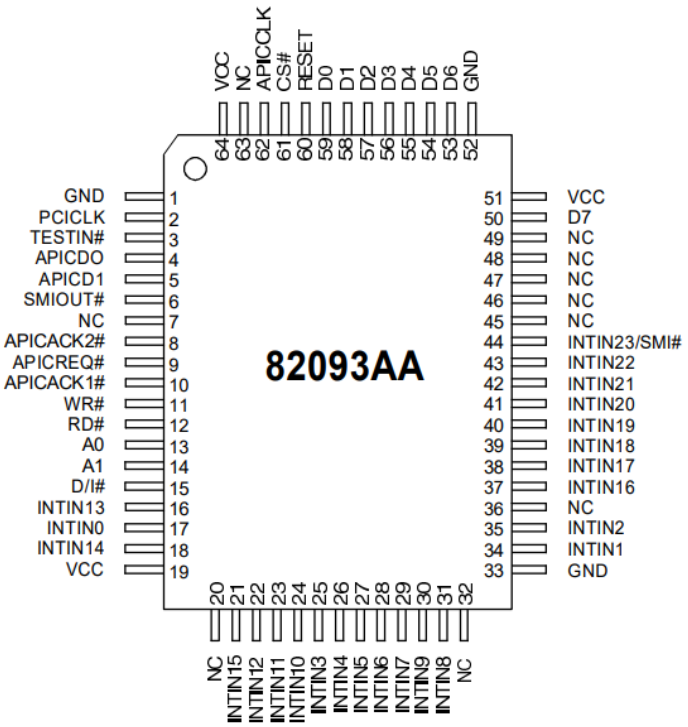


图16　I/O APIC

值得注意的是，I/O APIC Version 寄存器中 16-23 位为 Redirection Table 的条目数。其次，I/O APIC 使用的同样是 MMIO 的方式，但不同于 Local APIC 的映射方式，它只是将引脚 bus 默认映射到 0xFEC00000 为基地址的内存空间中。XV6 系统不能直接按照数组访问的方式访问 I/O APIC 中的寄存器，而是通过写一个 32 位的索引到特定 bus 对应的空间，有 I/O APIC 自身硬件逻辑返回（或写入）该索引对应的 32 位数据。另外，由于 RTE 每项为 64 位，故每项 RTE 对应两个索引，需要进行两次读（或写）才能读出（写入）RTE 的所有数据。最后，I/O APIC Version 寄存器的索引为 0x00，I/O APIC Version 寄存器的索引为 0x01，Redirection Table 的起始索引为 0x10。

图 17 中 ioapic 表示的是 I/O APIC 在 XV6 内存中映射的结构，reg 是寄存器的索引，data 是寄存器中的数据。ioapicread()通过将 ioapic 中的 reg 设为对应索引，使硬件返回数据至 data，便可实现读取。ioapicwrite()通过将 ioapic 中的 reg 设为对应索引，并将 data 更新，写硬件写入更新的数据。

Ioapicinit()的逻辑是：首先将 ioapic 指针指向 0xFEC00000；其次，调用 ioapicread()读取 I/O APIC Version 寄存器中的内容，取出 16-23 位得到 Redirection Table 的条目数；第三，调用 ioapicread()读取 I/O APIC ID，进行相应的比较判断；最后，根据第二步得到的 Redirection Table 条目数，逐项初始化 RTE（每项 RTE 需要分为高 32 位和低 32 位分别初始化）。

- **consoleinit()：console.c ioapic.c**

consoleinit()方法中调用了 ioapic.c 中的 ioapicenable()方法将控制台中断所对应的 RTE 项进行修改，使 I/O APIC 对控制台的中断信息转化为可接受状态。当系统开启中断后，CPU 便可以接收到控制台的中断信息（如果控制台发生中断事件）。Ioapicenable()中对 I/O APICRTE 表项的写入方式已经说明过，分别写入 RTE 的高 32 位和低 32 位。

```
27  // IO APIC MMIO structure: write reg, then read or write data.
28  struct ioapic {
29    uint reg;
30    uint pad[3];
31    uint data;
32  };
33  static uint
34  ioapicread(int reg)
35  {
36    // 看不懂
37    ioapic->reg = reg;
38    return ioapic->data;
39  }
40  static void
41  ioapicwrite(int reg, uint data)
42  {
43    ioapic->reg = reg;
44    ioapic->data = data;
45  }
```

图17　ioapic.c

- **uartinit()：uart.c**

UART 是通用异步接收器/发送器，负责执行与计算机的串行通信中的主要任务。该设备将传入的并行信息更改为可以在通信线路上发送的串行数据。UART 执行通信所需的所有任务，时序，奇偶校验等。

UART 使用 PMIO 的方式与 CPU 进行通信，在 XV6 中的起始端口号位 0x3f8。其内部寄存器所对应得端口偏移如图 18 所示，其通过 LRC line control（端口为 base+3）中得 DLAB 位决定自身暴露出哪组寄存器与 CPU 进行交互。uartinit()方法中系统通过 outb 命令不断切换 DLAB 位的值，对不同的寄存器写入初始化数据，使其能够开始与 CPU 进行异步通信。

UART register to port conversion table

	DLAB = 0		DLAB = 1	
I/O port	Read	Write	Read	Write
base	**RBR** receiver buffer	**THR** transmitter holding	**DLL** divisor latch LSB	
base + 1	**IER** interrupt enable	**IER** interrupt enable	**DLM** divisor latch MSB	
base + 2	**IIR** interrupt identification	**FCR** FIFO control	**IIR** interrupt identification	**FCR** FIFO control
base + 3		**LCR** line control		
base + 4		**MCR** modem control		
base + 5	**LSR** line status	– factory test	**LSR** line status	– factory test
base + 6	**MSR** modem status	– not used	**MSR** modem status	– not used
base + 7		**SCR** scratch		

图18　UART 寄存器端口偏移

- **tvinit()：trap.c mmu.h vectors.S vectors.pl**

tvinit()方法将每个中断描述符中 32 位为 vectors 数组中的对应项的偏移量分别填入中断描述符表，并将内核代码地址偏移放入 cs 位，并设置门特权级为内核特权级 0（系统调用对应的条目设置为用户特权级 3，使得用户可以通过该门描述符调用系统调用）。

值得注意的是，如图 19 所示，vectors 对应了 vectors.S 中声明的一个 vectors 地址，vectors.S 由 vectors.pl 脚本生成，里面包含了从 0 到 255 号中断的处理汇编程序，vectors 便是这 256 个汇编命令起始地址的数组。另外，从 vectors.pl 的生成逻辑中可以看出：所有的中断处理汇编程序都是在压入错误码（不是必须，可能在 CPU 已经压入）和中断码后，便跳转至 alltraps。

```
29  # sample output:
30  #   # handlers
31  #   .globl alltraps
32  #   .globl vector0
33  #   vector0:
34  #     pushl $0
35  #     pushl $0
36  #     jmp alltraps
37  #   ...
38  #
39  #   # vector table
40  #   .data
41  #   .globl vectors
42  #   vectors:
43  #     .long vector0
44  #     .long vector1
45  #     .long vector2
46  #   ...
```

图19　vectors.S 样例

另外，trap.c 中的 trap()方法是具体的中断处理程序，所有的中断处理最后都会汇集到这个方法。它根据中断号识别中断类型，根据不同的中断类型进行相应的操作。

- **mpmain(): main.c trap.c x86.h proc.c**

main.c 文件中的 mpmain()方法分别调用 idtinit()将中断描述符表的基地址写入 IDTR、调用 xchg 发送初始化完成信息、调用 scheduler()开中断使 CPU 开始接收中断信号。

小组问题讨论

1. 可屏蔽中断与不可屏蔽中断，异常呢？

- **中断**

中断可以被分为可屏蔽中断和不可屏蔽中断。cpu 一般设置两根中断请求输入线：可屏蔽中断请求 INTR(Interrupt Require)和不可屏蔽中断请求 NMI(Nonmaskable Interrupt)。可屏蔽中断是通过 CPU 的 INTR 引脚引入，当中断标志 IF＝1 时允许中断，当 IF=0 时禁止中断，

不可屏蔽中断是由 NMI 引脚引入，不受 IF 标志的影响。

由中断控制器管理的硬件中断属于可屏蔽的中断，主要包括由 8259A 中断控制器控制的外部中断、在 Local APIC 里产生的中断和在 I/O APIC 里产生的中断。对于这些中断源，可使用下面方法屏蔽：

1) 将 EFLAGS 寄存器中 IF 位置 0：当 IF 为 0 时，CPU 将不响应这些可屏蔽的中断。
2) 使用操作命令字 OCW1 在 8259A 中断控制器的 IMR(Interrupt Mask Register)寄存器里对 IRQ 相应的位置 1，将屏蔽对应等级的中断请求。
3) 将 Local APIC 的 Local Vector Table 对应项和 I/O APIC 中 Redirection Table 对应项的 Mask 位设为 1，可以屏蔽相应中断寄存器产生的中断信号。

- **异常**

对于异常：对异常的处理一般要依赖于当前程序的运行现场，异常不能屏蔽，一旦出现应立即处理。

2. 在系统初始化过程中，为什么在其他 CPU 未初始化 Local APIC 时主 CPU 可以通过 Local APIC 发送中断信号使其他 CPU 开始初始化？

如图 20 所示，在 main()方法中，主 CPU 在调用 startothers()发送 Local APIC 中断信号使其他 CPU 初始化前，调用了 lapicinit()对主 CPU 对应的 APIC 进行初始化。其中在末尾部分，系统向 Local APICI 的 ICR 寄存器写了一条广播信号，这条中断信号广播至其他 Local APIC，使它们进行较为简单的初始化，为后面接收中断信号唤醒 CPU 做准备。

```
88    // Ack any outstanding interrupts.
89    lapicw(EOI, 0);
90
91    // Send an Init Level De-Assert to synchronise arbitration ID's.
92    // ICR寄存器，用户发送IPI
93    lapicw(ICRHI, 0);
94    lapicw(ICRLO, BCAST | INIT | LEVEL);
95    while(lapic[ICRLO] & DELIVS)
96      ;
97
98    // Enable interrupts on the APIC (but not on the processor).
99    lapicw(TPR, 0);
100 }
```

图 20　iapicinit()广播 IPI

3. IA32 体系结构中门描述符总共有几种，XV6 使用了几种？

IA32 体系结构中门描述总共有四种，包括：任务门（Task Gate）、中断门（Interrupt Gate），陷阱门（Trap Gate）和调用门（Call Gate）。XV6 系统中只使用了两种，分别是用于系统调用的陷阱门和用于其他中断异常的中断门。

参考文献

[1] XV6 源码阅读-中断与系统调用[EB/OL]. 荒野之萍. 2019.06.09. https://icoty.github.io/2019/06/09/xv6-interrupt-systemcall/

[2] XV6 的中断和系统调用[EB/OL]. 苏畅. 2020.03.05. https://zhuanlan.zhihu.com/p/113365730

[3] LINT0 和 LINT1[EB/OL]. 李海伟 404. 2020.06.01. https://blog.csdn.net/GerryLee93/article/details/106475021/

[4] SMP 实现中的关键技术[EB/OL]. 白草黑尖 . 2010.04.18. https://www.cnblogs.com/sopc-mc/archive/2010/04/18/1714925.html

[5] CPU 三大架构 numa smp mpp[EB/OL]. 钟大发. 2017.02.23. https://www.jianshu.com/p/81233f3c2c14

[6] 详解 8259A[EB/OL]. 车子 chezi. 2018.03.04. https://blog.csdn.net/longintchar/article/details/79439466

[7] IO APIC - MIT-pdos[EB/OL]. MIT. 2012. https://pdos.csail.mit.edu/6.828/2012/readings/ia32/ioapic.pdf

[8] 浅谈内存映射 I/O(MMIO)与端口映射 I/O(PMIO)的区别[EB/OL]. veli. 2017.10.19. https://www.cnblogs.com/idorax/p/7691334.html

[9] Serial UART, an introduction[EB/OL]. Lammert Bies. 2019.11. https://www.lammertbies.nl/comm/info/serial-uart#LSR

[10] JOS lab4 Lapic 与 Intel 多核系统[EB/OL]. Tommylwp. 2016.03.27. https://blog.csdn.net/Poundssss/article/details/50989867?locationNum=1&fps=1

[11] xv6-Chinese[EB/OL]. Ranxian. 2020.03.31. https://github.com/ranxian/xv6-chinese

[12] Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1[EB/OL]. https://www.intel.cn/content/www/cn/zh/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html

[13] 北京大学操作系统高级课程运行环境和运行机制课件[EB/OL]. 陈向群. 2020.

[14] 可屏蔽中断和非屏蔽中断区别[EB/OL]. lidandan2016. 2016.12.02. https://blog.csdn.net/lidandan2016/article/details/53437273/

[15] 2020 年操作系统考研复习指导[M]. 王道论坛. 2020.

[16] 中断的屏蔽[EB/OL]. 李海伟_online. 2020.10.18. https://blog.csdn.net/GerryLee93/article/details/106477171/