实验四（PA4）中断与 I/O

实验简介(请认真阅读以下内容，若有违反，后果自负)

预计平均耗时/代码量: 30 小时/300 行

实验内容: 本次实验的阶段性安排如下:

- 阶段 1: 在 NEMU 中运行使用 printf()输出的 Hello 程序
- 阶段 2: 移植打字小游戏
- 阶段 3: 移植仙剑奇侠传

提交说明: 见这里

注意事项: 实验部分内容需要 cmov 指令的支持，但 cmov 指令在 i386 手册中没有涉及。提供的 PA3 答案中已经实现了所有的 cmov 指令，请大家根据 nemu 运行时提示的 opcode 信息，将对应的 cmov 指令填入的操作码表即可。

评分依据: 代码实现占 70%，实验报告占 30%。代码实现中完成阶段 1 占 50%，阶段 2 占 40%，阶段 3 占 10%。

进行本实验前，请在工程目录下执行以下命令进行分支整理，否则将影响成绩:

```
git commit --allow-empty -am "before starting PA4"
git checkout master
git merge pa3
git checkout -b pa4
```

普通阅读，无需提交。

必做任务，需要提交

思考题，需要提交

实验简介，请仔细阅读。

满足合理的需求

在之前的 PA 中，用户进程都只能安分守己地在运行在 NEMU 上，除了"计算"之外，什么都做不了。但像"输出一句话"这种合理的需求，内核必须想办法满足它。举一个银行的例子，如果银行连最基本的取款业务都不能办理，是没有客户愿意光顾它的。但同时银行也不能允许客户亲自到金库里取款，而是需要客户按照规定的手续来办理取款业务。同样地，操作系统并不允许用户进程直接操作显示器硬件进行输出，否则恶意程序就很容易往显示器中写入恶意数据，让屏幕保持黑屏，影响其它进程的使用。因此，用户进程想输出一句话，也要经过一定的合法手续，这一合法手续就是系统调用。

我们到银行办理业务的时候，需要告诉工作人员要办理什么业务，账号是什么，交易金额是多少，这无非是希望工作人员知道我们具体想做什么。用户进程执行系统调用的时候也是类似的情况，要通过一种方法描述自己的需求，然后告诉操作系统内核。用来描述需求最方便的手段就是使用通用寄存器了，用户进程将系统调用的参数依次放入各个寄存器中(第 1 个参数放在 %eax 中,第二个参数放在%ebx 中...)。为了让内核注意到用户进程提交的申请，系统调用通常都会触发一个异常，然后陷入内核。这个异常和非法操作产生的异常不同，内核能够识别它是由系统调用产生的。在 GNU/Linux 中，这个异常通过 int $0x80 指令触发。

我们可以在 GNU/Linux 下编写一个程序，手工触发一次 write 系统调用：

```
const char str[] = "Hello, world!\n";

int main() {
        asm volatile (  "movl $4, %eax;"        // system call ID, 4 = SYS_write
                        "movl $1, %ebx;"        // file descriptor, 1 = stdout
                        "movl $str, %ecx;"      // buffer address
                        "movl $14, %edx;"       // length
                        "int $0x80");

        return 0;
}
```

用户进程执行上述代码，就相当于告诉内核：帮我把从 str 开始的 14 字节写到 1 号文件中去。其中"写到 1 号文件中去"的功能相当于输出到屏幕上。

虽然操作系统需要为用户进程服务，但这并不意味着操作系统需要把所有信息都暴露给用户程序。有些信息是用户进程没有必要知道的，也永远不应该知道，例如 GDT，页表。因此，通常不存在一个系统调用用来获取 GDT 这些操作系统私有的信息。

事实上，你平时使用的 printf，cout 这些库函数和库类，对字符串进行格式化之后，最终也是通过系统调用进行输出。这些都是"系统调用封装成库函数"的

例子。系统调用本身对操作系统的各种资源进行了抽象，但为了给上层的程序员提供更好的接口(beautiful interface)，库函数会再次对部分系统调用再次进行抽象。例如 fopen 这个库函数用于创建并打开一个新文件，或者打开一个已有的文件，在 GNU/Linux 中，它封装了 open 系统调用。另一方面，系统调用依赖于具体的操作系统，因此库函数的封装也提高了程序的可移植性，在 windows 中，fopen 封装了 CreateFile 系统调用，如果在代码中直接使用 CreateFile 系统调用,把代码放到 GNU/Linux 下编译就会产生链接错误，即使链接成功，程序运行的时候也会产生运行时错误。

并不是所有的库函数都封装了系统调用，例如 strcpy 这类字符串处理函数就不需要使用系统调用. 从某种程度上来说，库函数的抽象确实方便了程序员，使得他们不必关心系统调用的细节。

穿越时空的旅程

异常是指 CPU 在执行过程中检测到的不正常事件，例如除数为零，无效指令，缺页等。IA-32 还向软件提供 int 指令，让软件可以手动产生异常，因此上文提到的系统调用也算是一种异常。那触发异常之后都发生了些什么呢？我们先来对这一场神秘的时空之旅作一些简单的描述。

为了方便叙述，我们称触发异常之前用户进程的状态为 A。触发异常之后，CPU 将会陷入内核，跳转到操作系统事先设置好的异常处理代码，处理结束之后再恢复 A 的执行。可以看到，A 的执行流程被打断了，为了以后能够完美地恢复到被打断时的状态,CPU 在处理异常之前应该先把 A 的状态保存起来，等到异常处理结束之后，根据之前保存的信息把计算机恢复到被打断之前的状态。

哪些内容表征了 A 的状态？在 IA-32 中，首先当然是 EIP(instruction pointer)了，它指示了 A 在被打断的时候正在执行的指令(或者下一条指令)；然后就是 EFLAGS(各种标志位)和 CS(代码段, CPL)。由于一些特殊的原因，这三个寄存器的内容必须由硬件来保存。此外，通用寄存器(GPR, general propose register)的值对 A 来说还是有意义的，而进行异常处理的时候又难免会使用到寄存器。但硬件并不负责保存它们，因此需要操作系统来保存它们的值。

> 思考题 1：异常和函数调用
>
> 我们知道进行函数调用的时候也需要保存调用者的状态：返回地址，而进行异常处理之前却要保存更多的信息. 尝试对比它们，并思考两者保存信息不同是什么原因造成的。

要将这些信息保存到哪里去呢？一个合适的地方就是进程的堆栈。触发异常时，硬件会自动将 EFLAGS, CS, EIP 三个寄存器的值保存到堆栈上。此外, IA-32 提供了 pusha / popa 指令，用于把通用寄存器的值压入/弹出堆栈，但你需要注

意压入的顺序，更多信息请查阅 i386 手册。

等到异常处理结束之后，CPU 将会根据堆栈上保存的信息恢复 A 的状态，最后执行 iret 指令。iret 指令用于从中断或异常处理代码中返回，它将栈顶的三个元素来依次解释成 EIP，CS，EFLAGS，并恢复它们。这样用户进程就可以从 A 开始继续运行了，在它看来，这次时空之旅就好像没有发生过一样。

神奇的传送门

我们在上文提到，用户进程触发异常之后将会陷入内核。"陷入内核"究竟是怎么样的一个过程？具体要陷入到什么地方去？要回答这些问题，我们首先要认识 IA-32 中断机制。

在 IA-32 中，异常事件的入口地址是通过门描述符(Gate Descriptor)来指示的. 门描述符有 3 种：

- 中断门(Interrupt Gate)
- 陷阱门(Trap Gate)
- 任务门(Task Gate)

关于中断/异常会在下文作进一步的解释，而任务门在实验中不会用到。中断门和陷阱门的结构如下：

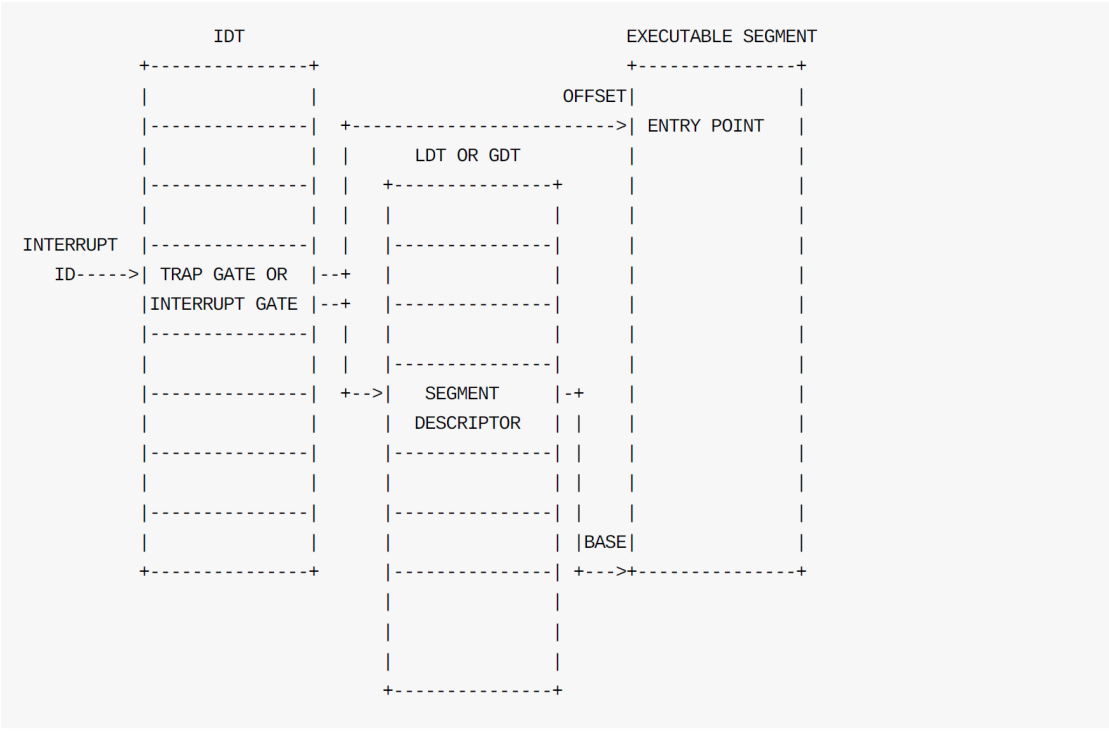
```
                            80386 INTERRUPT GATE
 31                23                15                7               0
+-----------------+-----------------+---+---+---------+-----+-----------+
|           OFFSET 31..16           | P |DPL|0 1 1 1 0|0 0 0|(NOT USED) |4
+-----------------------------------+---+---+---------+-----+-----------+
|             SELECTOR              |           OFFSET 15..0            |0
+-----------------+-----------------+-----------------+-----------------+

                            80386 TRAP GATE
 31                23                15                7               0
+-----------------+-----------------+---+---+---------+-----+-----------+
|          OFFSET 31..16            | P |DPL|0 1 1 1 1|0 0 0|(NOT USED) |4
+-----------------------------------+---+---+---------+-----+-----------+
|             SELECTOR              |           OFFSET 15..0            |0
+-----------------+-----------------+-----------------+-----------------+
```

由于 IA-32 分段机制的存在，我们必须通过段和段内偏移来表示跳转入口。因此在中断门和陷阱门中，selector 域用于指示目标段的段描述符，offset 域用于指示跳转目标在段内的偏移。这样，如果能找到一个门描述符，就可以根据门描述符中的信息计算出跳转目标了。

和分段机制类似，为了方便管理各个门描述符，IA-32 把内存中的某一段数据专门解释成一个数组，叫 IDT(Interrupt DescriptorTable，中断描述符表)，数

组的一个元素就是一个门描述符。为了找到一个门描述符，我们还需要一个索引. 对于 CPU 异常来说，这个索引由 CPU 内部产生(例如缺页异常为 14 号异常)，或者由 int 指令给出(例如 int $0x80)。最后，为了找到 IDT，IA-32 中使用 IDTR 寄存器来存放 IDT 的首地址和长度。操作系统需要事先把 IDT 准备好，然后通过一条特殊的指令把 IDT 的首地址和长度装载到 IDTR 中，IA-32 中断处理机制就可以正常工作了。

现在是万事俱备，等到异常的东风一刮, CPU 就会按照设定好的 IDT 跳转到目标地址：



1. 依次将 EFLAGS, CS, EIP 寄存器的值压入堆栈
2. 从 IDTR 中读出 IDT 的首地址
3. 根据异常(中断)号在 IDT 中进行索引，找到一个门描述符
4. 把门描述符中的 selector 域装入 CS 寄存器
5. 根据 CS 寄存器中的段选择符，在 GDT 或 LDT 中进行索引，找到一个段描述符，并把这个段的一些信息加载到 CS 寄存器的描述符 cache 中
6. 在段描述符中读出段的基地址，和门描述符中的 offset 相加，得出入口地址
7. 跳转到入口地址

需要注意的是，这些工作都是硬件自动完成的，不需要程序员编写指令来完成相应的内容。事实上，这只是一个大概的过程，在真实的计算机上还会遇到很

多特殊情况，在这里我们就不深究了。i386 手册中还记录了处理器对中断号和异常号的分配情况，并列出了各种异常的详细解释，需要了解的话可以进行查阅。

特殊的原因
我们在上一小节中提到："由于一些特殊的原因，这三个寄存器的内容必须由硬件来保存"。究竟是什么特殊的原因，使得 EFLAGS, CS, EIP 三个寄存器的值必须由硬件来保存？尝试结合 IA-32 中断机制思考这个问题。

在计算机和谐社会中，大部分神奇的传送门都不能让用户进程随意使用，否则恶意程序就可以通过 int 指令欺骗操作系统。例如恶意程序执行 int $0x2 来谎报电源掉电，扰乱其它进程的正常运行。因此执行 int 指令需要进行特权级检查，门描述符中的 DPL 域将会参与到特权级检查的过程中，具体的检查规则我们就不展开讨论了，需要了解的时候请查阅 i386 手册。

时空之旅大揭秘

让我们通过 testcase/hello-inline-asm.c 这个具体的例子，亲自体验一趟神秘的旅程. 我们会跟随 hello-inline-asm 程序一同探索这一时空之旅，当我们从时空之旅结束归来，hello-inline-asm 程序也已经将信息传达出去，我们的第一个任务也就完成了。

添加传送门

在踏上旅程之前，你还需要在 NEMU 中实现 IA-32 中断机制，只有这样才能敲开神奇的传送门。

一方面，你需要在 kernel 中加入相关的代码，你只需要在 kernel/include/common.h 中定义宏 IA32_INTR，然后重新编译 kernel 就可以了。重新编译后, kernel 会在 init_cond()函数中多进行两项初始化工作：

- 重新设置 GDT。
- 设置 IDT，具体来说就是填写 IDT 中每一个门描述符，设置完毕后通过 lidt 指令装载 IDTR。其它的工作和之前一样，没有变化.

另一方面，你需要在 NEMU 中添加中断机制的功能，以便让上述代码成功执行。具体的，你需要：

- 添加 IDTR 寄存器和 lidt 指令，注意 IDTR 中存放的 IDT 首地址是线性地址。
- 添加如下的 raise_intr()函数（可以新建一个 intr.c 的文件来定义该函数）来模拟 IA-32 中断机制的处理过程：

```
#include <setjmp.h>
extern jmp_buf jbuf;

void raise_intr(uint8_t NO) {
        /* TODO: Trigger an interrupt/exception with ``NO''.
         * That is, use ``NO'' to index the IDT.
         */


        /* Jump back to cpu_exec() */
        longjmp(jbuf, 1);
}
```

其中 longjmp()的功能相当于跨越函数的 goto 语句，执行 longjmp()会跳转到 cpu_exec()中 setjmp()的下一条指令，这样从 raise_intr() 中"返回"之后就会

马上继续执行 cpu.eip 所指向的指令，也就是异常处理入口指令。

- 添加 int, iret, cli, pusha, popa 等指令。关于 int 和 iret 指令，实验中不涉及特权级的切换，查阅 i386 手册的时候你不需要关心和特权级切换相关的内容。要注意的是
  - push imm8 指令需要对立即数进行**符号扩展**，这一点在 i386 手册中并没有明确说明。
  - 执行 int 指令后保存的 EIP 指向的是 int 指令的下一条指令，这有点像函数调用，具体细节可以查阅 i386 手册。
  - 你需要在 int 指令的 helper 函数中调用 raise_intr()，而不要把 IA-32 中断处理的代码放在 int 指令的 helper 函数中实现，因为在后面我们会再次用到 raise_intr()函数。

必做任务 1：实现 IA-32 中断机制

你需要在 NEMU 中添加 IA-32 中断机制的支持，如有疑问，请查阅 i386 手册。实现成功后在 NEMU 中运行 hello-inline-asm.c 程序，你会看到在 kernel/src/irq/irq_handle.c 中的 irq_handle()函数中触发了 BAD TRAP。这说明用户进程已经成功通过 IA-32 中断机制陷入内核，你将会在下面的任务中修复"触发 BAD TRAP"的问题。

神奇的 longjmp

你可能会认为没有必要在 raise_intr()函数中使用 longjmp，直接一步一步返回到 cpu_exec()中就可以了。但如果需要现处理器异常的识别和处理，longjmp 是最好的选择。假设你打算在页级转换过程中，当检测到页表项的 present 位为 0 时,通过 raise_intr(14)抛出缺页异常，如果不使用 longjmp，你将如何让代码从 raise_intr()返回到 cpu_exec()？如果还需要实现无效指令异常的处理，你又会怎么办？

你可以通过

```
man setjmp
```

```
man longjmp
```

查阅 setjmp 和 longjmp 的相关信息。思考一下，如果让你实现 setjmp 和 longjmp，你将如何实现？

思考题 2：重新设置 GDT

为什么要重新设置 GDT？尝试把 init_cond()函数中的 init_segment()注释掉，编译后重新运行，你会发现运行出错，你知道为什么吗？

曲折的旅途

我们的第一次时空之旅被迫止于半路之中，看来旅途并非我们想象中的那么顺利。为了找到问题的原因，我们需要仔细推敲途中的每一处细节。

用户进程执行 int $0x80 之后，CPU 将会保存现场，查阅 kenrel 设置好的 IDT，跳转到入口函数 vecsys()，压入错误码和异常号#irq，跳转到 asm_do_irq。在 asm_do_irq 中，代码将会把用户进程的通用寄存器保存到堆栈上，这些寄存器的内容连同之前保存的错误码，#irq，以及硬件保存的 EFLAGS，CS，EIP 形成了 **trap frame(陷阱帧)**的数据结构，它记录了用户进程陷入内核时的状态，注意到 trap frame 是在堆栈上构造的. 保存好用户进程的信息之后, kernel 就可以随意使用通用寄存器了. 接下来代码将会把当前的%esp 压栈，并调用 C 函数 irq_handle()。

思考题 3：诡异的代码

do_irq.S 中有一行 pushl %esp 的代码，乍看之下其行为十分诡异，你能结合前后的代码理解它的行为吗？Hint：不用想太多，其实都是你学过的知识。

必做任务 2：重新组织 TrapFrame 结构体

框架代码在 irq_handle()函数的开始处设置了 panic()，你的任务是理解 trap frame 形成的过程，然后重新组织 kernel/include/irq.h 中定义的 TrapFrame 结构体的成员，使得这些成员声明的顺序和 kernel/src/irq/do_irq.S 中构造的 trap frame 保持一致。实现正确之后，去掉上述的 panic()，irq_handle() 以及后续代码就可以正确地使用 trap frame 了。重新运行 hello-inline-asm 程序，你会看到在 kernel/src/syscall/do_syscall.c 中的 do_syscall()函数中触发了 BAD TRAP。

irq_handle()将会根据#irq 确定异常事件的类型，从而进行不同的处理。用户进程执行的 int $0x80 会被识别成系统调用请求，于是 kernel 会调用 do_syscall()对相应的请求进行处理。由于用户进程的所有现场信息都已经保存在 trap frame 中了，kernel 很容易获取它们，do_syscall()将根据用户进程之前

设置好的系统调用号和系统调用参数进行处理。我们找到了第二次触发 BAD TRAP 的原因：kernel 并没有实现 SYS_write 系统调用的处理。为了再次踏上旅程，你需要在 do_syscall()中添加 SYS_write 的系统调用。

添加一个系统调用比你想象中要简单，所有信息都已经准备好了。根据 write 系统调用的函数声明(参考 man 2 write)，在 do_syscall()中识别出系统调用号是 SYS_write 之后，检查 fd 的值，如果 fd 是 1 或 2(分别代表 stdout 和 stderr),则将 buf 为首地址的 len 字节输出到屏幕上，你需要思考如何从 trap frame 中获取这些信息。

现在又回到了最根本的问题了:要怎么才能把内容输出到屏幕上?在真实的计算机中，这些内容最终是由设备来负责输出的，但 NEMU 中还没有添加设备，所以我们还是先借助 NEMU 的功能来完成输出吧。我们使用以下内联汇编来"陷入"到 NEMU 中：

```
asm volatile (".byte 0xd6" : : "a"(2), "c"(buf), "d"(len));
```

这正是我们在 NEMU 中手工加入的 nemu_trap 指令。接下来修改 nemu/cpu/src/exec/special/special.c 中 nemu_trap()的 helper 函数，如果%eax 为 2，则输出%ecx 为首地址的%edx 字节的内容。在 NEMU 中输出就是一件易如反掌的事情了，不过需要注意，%ecx 表示的地址是**虚拟地址**，聪明的你知道应该怎么做了吧。

回到 kernel 中，假设执行完上述的 nemu_trap 指令后，"输出"成功了，处理系统调用的最后一件事就是设置系统调用的返回值。GNU/Linux 约定系统调用的返回值存放在%eax 中，所以我们只需要修改 tf->eax 的值就可以了。至于 write 系统调用的返回值是什么，请查阅 man 2 write。

系统调用处理结束后，代码将会一路返回到 do_irq.S 的 asm_do_irq()中.接下来的事情就是恢复用户进程的现场，kernel 将根据之前保存的 trap frame 中的内容，恢复用户进程的通用寄存器(注意 trap frame 中的%eax 已经被设置成系统调用的返回值了),并直接弹出一些不再需要的信息，最后通过 iret 指令恢复用户进程的 EIP，CS，EFLAGS。用户进程可以通过%eax 寄存器获得系统调用的返回值，进而得知系统调用执行的结果。

这样，一次系统调用就执行完了，时空之旅圆满结束。

必做任务 3：实现系统调用

你需要在 kernel 中添加 SYS_write 系统调用，让 hello-inline-asm 程序成功输出"Hello, world!"的信息。

实现 SYS_write 系统调用之后，我们已经为"使用 printf()"扫除了最大的障碍了，因为 printf()进行字符串格式化之后，最终会通过 write 系统调用进行输出. 至于格式化的功能, uclibc 已经为我们实现好了。

我们这次的任务是在 NEMU 中执行 testcase/hello.c 程序。SYS_brk 系统调用用于调整用户进程堆区的大小，kernel 中已经实现了这个系统调用了。hello 程序编译通过之后，你就可以在 NEMU 中运行它了。

必做任务 4：在 NEMU 中输出“Hello world”
在 NEMU 中运行 testcase/hello.c 程序.

温馨提示
PA4 阶段 1 到此结束。

天外有天的世界

我们之前让 NEMU 伸出上帝之手，用户进程才能输出一句话，怎么说都有点作弊的嫌疑。在真实的计算机中，输入输出都是通过 I/O 设备来完成的。

设备的工作原理其实没什么神秘的。你只要向设备控制器发送一些有意义的信号，设备就会按照这些信号的含义来工作。让一些信号来指导设备如何工作，这不就像"程序的指令指导 CPU 如何工作"一样吗？恰恰就是这样！设备也有自己的状态寄存器(相当于 CPU 的寄存器)，也有自己的功能部件(相当于 CPU 的运算器)。当然不同的设备有不同的功能部件，例如键盘控制器有一个把按键的模拟信号转换成扫描码的部件，而 VGA 控制器则有一个把像素颜色信息转换成显示器模拟信号的部件。如果把控制设备工作的信号称为"设备指令"，设备控制器的工作就是负责接收设备指令，并进行译码和执行... 你已经知道 CPU 的工作方式，这一切对你来说都太熟悉了。唯一让你觉得神秘的，就要数设备功能部件中的模/数转换，数/模转换等各种有趣的实现。遗憾的是，我们的课程体系并没有为我们提供实践的机会，因此它们成为了一种神秘的存在。

巴别之塔

我们希望计算机能够控制设备，让设备做我们想要做的事情，这一重任毫无悬念地落到了 CPU 身上。CPU 除了进行运算之外，还需要与设备协作来完成不同的任务。要控制设备工作，就需要向设备发送设备指令。接下来的问题是, CPU 怎么区分不同的设备?具体要怎么向一个设备发送设备指令?

对第一个问题的回答涉及到 I/O 的寻址方式。一种 I/O 寻址方式是端口映射 I/O(port-mapped I/O), CPU 使用专门的 I/O 指令对设备进行访问，并为设备中允许 CPU 访问的寄存器逐一编号，这些编号叫端口号。有了端口号以后，在 I/O 指令中给出端口号，就知道要访问哪一个设备的哪一个寄存器了。市场上的计算机绝大多数都是 IBM PC 兼容机，IBM PC 兼容机对常见设备端口号的分配有专门的规定。设备中可能会有一些私有寄存器，它们是由设备控制器自己维护的，它们没有端口号, CPU 不能直接访问它们。

IA-32 提供了 in 和 out 指令用于访问设备，其中 in 指令用于将设备寄存器中的数据传输到 CPU 寄存器中，out 指令用于将 CPU 寄存器中的数据传送到设备寄存器中。一个例子是 kernel/src/irq/i8259.c 的代码，代码使用 out 指令与 Intel 8259 中断控制器进行通信，给控制器发送命令字。例如

```
movl $0x11, %eax
movb $0x20, %dl
outb %al, (%dx)
```

上述代码把数据 0x11 传送到 0x20 号端口所对应的设备寄存器中。你要注意区分 I/O 指令和设备指令，I/O 指令是 CPU 执行的，作用是对设备寄存器进行读写；而设备指令是设备来执行的，作用和设备相关，由设备来解释和执行。CPU 执行上述代码后，会将 0x11 这个数据传送到中断控制器的一个控制寄存器中，中断控制器接收到 0x11 后，把它解释成一条设备指令，发现是一条初始化指令，于是就会进入初始化状态；但对 CPU 来说，它并不关心 0x11 的含义，只会老老实实地把 0x11 传送到 0x20 号端口，至于设备接收到 0x11 之后会做什么，那就是设备自己的事情了。

另一种 I/O 寻址方式是内存映射 I/O(memory-mapped I/O)。这种寻址方式将一部分物理内存映射到 I/O 设备空间中，使得 CPU 可以通过普通的访存指令来访问设备。这种物理内存的映射对 CPU 是透明的，CPU 觉得自己是在访问内存，但实际上可能是访问了相应的 I/O 空间。这样以后，访问设备的灵活性就大大提高了。一个例子是物理地址区间[0xa0000, 0xc0000)，这段物理地址区间被映射到 VGA 内部的显存，读写这段物理地址区间就相当于对读写 VGA 显存的数据。例如

```
memset((void *)0xa0000, 0, SCR_SIZE);
```

思考题 4：不可缓存（uncachable）的内存区域

我们知道使用 cache 可以提高访问内存的速度，但是对于用作内存映射 I/O 的物理地址空间，使用 cache 却可能造成致命的错误。因此一般会将这部分物理地址空间设置为不可缓存，这样，每一次对它们的访问都不会经过 cache，而是老老实实地访问相应的"内存区域"。你知道为什么要这样做吗？

。

你已经见识过 IA-32 引入的保护机制对构造计算机和谐社会所做出的贡献了。IA-32 所倡导的和谐是全方面的，自然也不希望恶意程序随意访问设备. 针对端口映射 I/O，IA-32 规定了可以使用 I/O 指令的最低特权级，它用 EFLAGS 寄存器的 IOPL 位来表示. 如果当前进程的 CPL 在数值上大于 IOPL，使 in /out 指令将会导致 CPU 抛出异常，于是你在 GNU/Linux 中又看到了熟悉的段错误了。

理解 volatile 关键字

也许你从来都没听说过 C 语言中有 volatile 关键字，但它从 C 语言诞生开始就一直存在。该关键字的作用十分特别，它的作用是避免编译器对相应代码进行优化。你应该动手体会一下 volatile 的作用，在 GNU/Linux 下编写以下代码：

```
void fun() {
    volatile unsigned char *p = (void *)0x8049000;
    *p = 0;
    while(*p != 0xff);
    *p = 0x33;
    *p = 0x34;
    *p = 0x86;
}
```

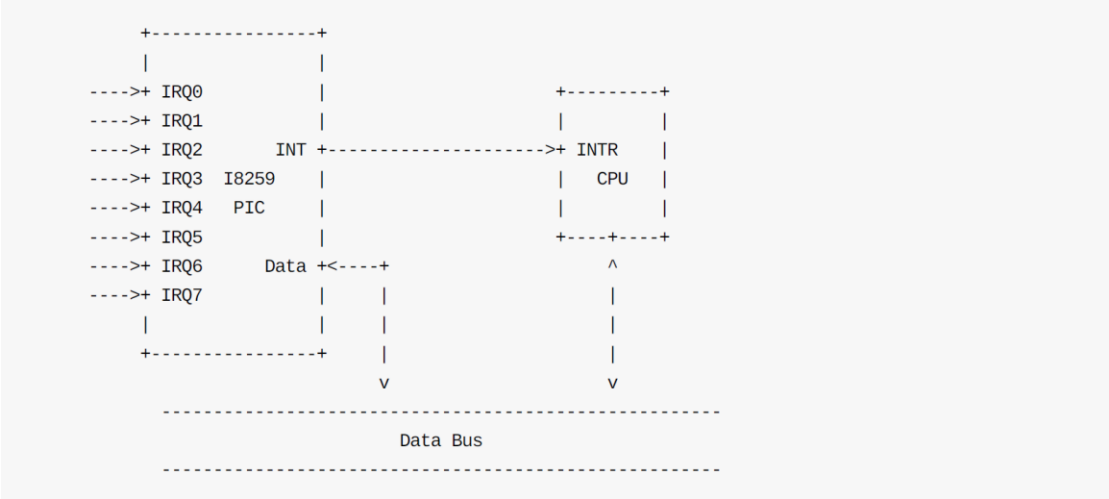
然后使用-O2 编译代码。尝试去掉代码中的 volatile 关键字，重新使用-O2 编译，并对比去掉 volatile 前后反汇编结果的不同。

来自外部的声音

让 CPU 一直监视设备的工作可不是明智的选择。以磁盘为例，磁盘进行一次读写需要花费大约 5 毫秒的时间，但对于一个 2GHz 的 CPU 来说，它需要花费 10,000,000 个周期来等待磁盘操作的完成。这对 CPU 来说无疑是巨大的浪费，因此我们迫切需要一种汇报机制：在磁盘读写期间，CPU 可以继续执行与磁盘无关的代码；磁盘读写结束后，主动向 CPU 汇报，这时 CPU 才继续执行与磁盘相关的代码。这样的汇报机制就是硬件中断。硬件中断的实质是一个信号，当设备有事件需要通知 CPU 的时候，就会发出中断信号.这个信号最终会传到 CPU 中，引起 CPU 的注意。

第一个问题就是中断信号是怎么传到 CPU 中的。支持中断机制的设备控制器都有一个中断引脚，这个引脚会和 CPU 的 INTR 引脚相连，当设备需要发出中断请求的时候，它只要将中断引脚置为高电平，中断信号就会一直传到 CPU 的 INTR 引脚中。但计算机上通常有多个设备，而 CPU 引脚是在制造的时候就固定了，因而在 CPU 端为每一个设备中断分配一个引脚的做法是不现实的。

为了更好地管理各种设备的中断请求，IBM PC 兼容机中都会带有 Intel 8259 PIC(Programmable Interrupt Controller，可编程中断控制器)。中断控制器最主要的作用就是充当设备中断信号的多路复用器，即在多个设备中断信号中选择其中一个信号，然后转发给 CPU。



如上图所示, i8259 有 8 个不同的中断请求引脚 IRQ0-IRQ7，可以识别 8 种不同的硬件中断来源。在 i8259 内部还有一些中断屏蔽逻辑和判优逻辑，当多个中断请求引脚都有中断请求到来时，i8259 会根据当前的设置选择一个未被屏蔽的，优先级最高的中断请求，根据该中断请求的所在的引脚生成一个中断号，将中断号发送到数据总线上，并将 INT 引脚置为高电平，通知 CPU 中断请求到来。

上面描述的是中断控制器最简单的工作过程，在真实的机器中还有很多细节问题，例如 i8259 的级联，多个设备共享同一个 IRQ 引脚等，这里就不详细展开了，更多的信息可以查看 i8259 手册，或者在互联网上搜索相关信息。

第二个问题是 CPU 如何响应到来的中断请求。CPU 每次执行完一条指令的时候，都会看看 INTR 引脚，看是否有设备的中断请求到来。一个例外的情况就是 CPU 处于关中断状态。在 x86 中，如果 EFLAGS 中的 IF 位为 0，则 CPU 处于关中断状态，此时即使 INTR 引脚为高电平, CPU 也不会响应中断。CPU 的关中断状态和中断控制器是独立的，中断控制器只负责转发设备的中断请求，最终 CPU 是否响应中断还需要由 CPU 的状态决定。

如果中断到来的时候, CPU 没有处在关中断状态，它就要马上响应到来的中断请求。我们刚才提到中断控制器会生成一个中断号, IA-32 将会保存中断现场，然后根据这个中断号在 IDT 中进行索引，找到并跳转到入口地址，进行一些和设备相关的处理，这个过程和之前提到的异常处理十分相似。一般来说，中断处理会在 IDT 中找到中断门描述符，异常处理会在 IDT 中找到陷阱门描述符。它们的唯一区别就是，穿过中断门的时候，EFLAGS 中的 IF 位将会被清零，达到屏蔽其它外部中断的目的；而穿过陷阱门的时候, IF 位将保持不变。

对 CPU 来说，设备的中断请求何时到来是不可预测的，在处理一个中断请求的时候到来了另一个中断请求也是有可能的。如果希望支持中断嵌套 -- 即在进行优先级低的中断处理的过程中，响应另一个优先级高的中断 -- 那么堆栈将是保存中断现场信息的唯一选择。如果选择把现场信息保存在一个固定的地方，

发生中断嵌套的时候，第一次中断保存的现场信息将会被优先级高的中断处理过程所覆盖，从而造成灾难性的后果。

如果没有中断的存在，计算机的运行就是完全确定的，根据当前的指令和计算机的状态，你完全可以推断出下一条指令执行后,甚至是执行 100 条指令后计算机的状态。正是中断的不可预测性，给计算机世界带来了不确定性的乐趣。而在分时多任务操作系统中，中断更是操作系统赖以生存的根基：只要中断的东风一刮，操作系统就会卷土重来，一个故意执行死循环的恶意程序就算有天大的本事，此时此刻也要被请出 CPU，从而让其它程序得到运行的机会；如果没有中断，一个陷入了死循环的程序将使操作系统万劫不复。但另一方面，中断的存在也会让操作系统在一些问题的处理上需要付出额外代价，最常见的问题就是保证某些操作的原子性：如果在一个原子操作进行到一半的时候到来了中断，数据的一致性状态将会被破坏，成为了潜伏在系统中的炸弹；而且由于中断到来是不可预测的，重现错误可能需要付出比修复错误更大的代价... 即使这样，中断对现代计算机作出的贡献是不可磨灭的，由中断撑起半边天的操作系统也将长久不衰。

加入最后的拼图

设备代码介绍

框架代码中已经提供了设备的代码，位于 nemu/src/device 目录下。代码中提供了两种 I/O 寻址方式，i8259 中断控制器和五种设备的模拟。为了简化实现，中断控制器和所有设备都是不可编程的，只实现了在 NEMU 中用到的功能。我们对代码稍作解释。

- nemu/src/device/io/port-io.c 是对端口 I/O 的模拟。其中 PIO_t 结构用于记录一个端口 I/O 映射的关系，设备会初始化时会调用 add_pio_map()函数来注册一个端口 I/O 映射关系，返回该映射关系的 I/O 空间首地址。pio_read()和 pio_write()是面向 CPU 的端口 I/O 读写接口。由于 NEMU 是单线程程序，因此只能串行模拟整个计算机系统的工作，每次进行 I/O 读写的时候,才会调用设备提供的回调函数(callback)，更新设备的状态。内存映射 I/O 的模拟和端口 I/O 的模拟比较相似，只是内存映射 I/O 的读写并不是面向 CPU 的，这一点会在下文进行说明。
- nemu/src/device/i8259.c 是对 Intel 8259 中断控制器的功能模拟。代码模拟了两块 i8259 芯片级联的情况，从片的 INT 引脚连接到主片的 IRQ2 引脚。i8259_raise_intr()是面向设备的接口，当设备需要发出硬件中断时，就会调用 i8259_raise_intr(),代码会根据当前的中断请求状态选择一个优先级最高的中断，并生成相应的中断号，把中断号记录在 intr_NO 变量中，然后把 CPU 的 INTR 引脚置为高电平，通知 CPU 有硬件中断到来。CPU 如果发现有硬件中断到来，可以通过 i8259_query_intr()查询当前优先级最高的中断号，并调用 i8259_ack_intr()向中断控制器确认收到中断信息，中断控制器收到 CPU 的确认后会更新中断请求的状态，清除刚才发送给 CPU 的中断请求。为了简化，中断控制器中没有实现中断屏蔽位的功能。
- nemu/src/device/timer.c 模拟了 i8253 计时器的功能。计时器的大部分功能都被简化，只保留了"发起时钟中断"的功能。
- nemu/src/device/keyboard.c 模拟了 i8042 通用设备接口芯片的功能。其大部分功能也被简化，只保留了键盘接口。i8042 初始化时会注册 0x60 处的端口作为数据寄存器，每当用户敲下/释放按键时，将会把键盘扫描码放入数据寄存器，然后发起键盘中断，CPU 收到中断后，可以通过端口 I/O 访问数据寄存器，获得键盘扫描码。
- nemu/src/device/serial.c 模拟了串口的功能。其大部分功能也被简化，只保留了数据寄存器和状态寄存器。串口初始化时会注册 0x3F8 处长度为 8 个字节的端口作为其寄存器，但代码中只模拟了其中的两个寄存器的功能，由于 NEMU 串行模拟计算机系统的工作，串口的状态寄存器可以一直处于

空闲状态；每当 CPU 往数据寄存器中写入数据时，串口会将数据传送到主机的标准输出。

- nemu/src/device/ide.c 模拟了磁盘的功能。磁盘初始化时会注册 0x1F0 处长度为 8 个字节的端口作为其寄存器，并把 NEMU 运行时传入的测试文件当做虚拟磁盘来使用。磁盘读写以扇区为单位，进行读写之前，磁盘驱动程序需要把读写的扇区号写入磁盘的控制寄存器，然后往磁盘的命令寄存器中写入读/写命令字。进行读操作时，驱动程序可以从磁盘的数据寄存器依次读出 512 个字节；进行写操作时，驱动程序需要向磁盘的数据寄存器依次写入 512 个字节。
- nemu/src/device/vga.c 模拟了 VGA 的功能。VGA 初始化时会注册了两个用于更新调色板的端口，并注册了从 0xa0000 开始的一段用于映射到 video memory 的物理内存。在 NEMU 中，video memory 是唯一使用内存映射 I/O 方式访问的 I/O 空间。代码只模拟了 320x200x8 的图形模式，一个像素占 8bit 的存储空间，因此在一幅图中最多能够同时使用 256 种颜色。
- nemu/src/device/vga-palette.c 定义了 VGA 的默认调色板. 现代的显示器一般都支持 24 位的颜色(R, G, B 各占 8 个 bit，共有 2^8*2^8*2^8 约 1600 万种颜色)，为了让屏幕显示不同的颜色成为可能，在 8 位颜色深度时会使用调色板的概念。调色板是一个颜色信息的数组，每一个元素占 4 个字节，分别代表 R(red)，G(green)，B(blue)，A(alpha)的值，其中 VGA 不使用 alpha 的信息.引入了调色板的概念之后，一个像素存储的就不再是颜色的信息，而是一个调色板的索引：具体来说，要得到一个像素的颜色信息，就要把它的值当作下标，在调色板这个数组中做下标运算，取出相应的颜色信息。因此，只要使用不同的调色板，就可以在不同的时刻使用不同的 256 种颜色了。如果你对 VGA 编程感兴趣，这里有一个名为 FreeVGA 的项目，里面提供了很多 VGA 的相关资料。
- nemu/src/device/sdl.c 中是和 SDL 库相关的代码, NEMU 使用 SDL 库来模拟计算机的标准输入输出。在 init_sdl()函数中会进行一些和 SDL 相关的初始化工作，包括创建窗口，设置默认调色板等。最后还会注册一个 100Hz 的定时器，每隔 0.01 秒就会调用一次 device_update()函数。device_update()函数主要进行一些设备的操作，包括发送 100Hz 的时钟中断，以 25Hz 的频率刷新屏幕，以及检测是否有按键按下/释放，若有，则发送键盘中断。需要说明的是，代码中注册的定时器是虚拟定时器,它只会在 NEMU 处于用户态的时候进行计时，如果 NEMU 在 ui_mainloop()中等待用户输入，定时器将不会计时；如果 NEMU 进行大量的输出，定时器的计时将会变得缓慢，因此除非你在进行调试，否则尽量避免大量输出的情况，从而影响定时器的工作。

如何检测多个键同时被按下

通常我们会利用键盘的扫描码判断按键信息：当按下一个键的时候，键盘控制器将会发送该键的通码(makecode)；当释放一个键的时候，键盘控制器将会发送该键的断码(break code)，其中断码的值为通码的值+0x80. 需要注意的是,断码仅在键被释放的时候才发送，因此你应该用"收到断码"来作为键被释放的检测条件，而不是用"没收到通码"作为检测条件。

在游戏中，很多时候需要判断玩家是否同时按下了多个键，例如 RPG 游戏中的八方向行走，格斗游戏中的组合招式等等. 根据键盘扫描码的特性，你知道这些功能是如何实现的吗？

提高键盘读写的效率

阅读 kernel/driver/ide/disk.c 中的代码，理解 kernel 读写磁盘的方式. 以读操作为例，你会发现磁盘中的每一个数据都先通过 in 指令读入到寄存器中，然后再通过 mov 指令把读到的数据放回内存；写操作也是类似的情况。思考一下，有什么办法能够提高磁盘读写的效率？

我们提供的代码是模块化的，为了让新加入的代码在 NEMU 中工作，你只需要在原来的代码上作少量改动：

- 在 nemu/include/common.h 中定义宏 HAS_DEVICE。
- 在 cpu 结构体(nemu/include/cpu/reg.h)中添加一个 bool 成员 INTR，这个成员用于表示是否有外部中断到来，它会在 nemu/src/device/i8259.c 中用到。
- 在 init_monitor()函数中加入初始化设备和 SDL 的代码：

```
init_device();
init_sdl();
```

代码在模拟某些设备的功能时用到了 SDL 库，为了编译新加入的代码，你需要先安装 SDL 库：

```
sudo apt-get install libsdl-dev
```

安装成功后，修改 nemu/Makefile.part，把 SDL 库加入链接对象：

```
--- nemu/Makefile.part
+++ nemu/Makefile.part
@@ -4,1 +4,1 @@
-nemu_LDFLAGS := -lreadline
+nemu_LDFLAGS := -lreadline -lSDL
```

之后重新编译。

使用设备代码

上述代码只是提供了 I/O 寻址方式的接口，你还需要在 NEMU 中编写相应的代码来调用这些接口。具体的，你需要：

- 实现 in, out 指令，在它们的 helper 函数中分别调用 pio_read()和 pio_write()函数。
- 在 hwaddr_read()和 hwaddr_write()中加入对内存映射 I/O 的判断。通过 is_mmio()函数判断一个物理地址是否被映射到 I/O 空间，如果是，is_mmio()会返回映射号，否则返回-1。内存映射 I/O 的访问需要调用 mmio_read()或 mmio_write()，调用时需要提供映射号。如果不是内存映射 I/O 的访问，就访问 DRAM。

你还需要在 NEMU 中添加和硬件中断相关的代码：

- 在 cpu_exec ()中 for 循环的末尾添加轮询 INTR 引脚的代码，每次执行完一条指令就查看是否有硬件中断到来：

```
if(cpu.INTR & cpu.eflags.IF) {
        uint32_t intr_no = i8259_query_intr();
        i8259_ack_intr();
        raise_intr(intr_no);
}
```

- 添加 hlt 指令。这条指令十分特殊，执行这条指令后，CPU 直到硬件中断到来之前都不会执行下一条指令。实现的时候，只需要在相应的 helper 函数中通过一个循环不断查看 INTR 引脚，直到满足响应硬件中断的条件才退出循环。需要注意的是，如果在关中断状态下执行 hlt 指令，响应硬件中断的条件将永远得不到满足，CPU 将一直处于停止工作状态，永远无法执行下一条指令。

最后你需要在 kernel 中加入相关的代码，你只需要在 kernel/include/common.h 中定义宏 HAS_DEVICE，然后重新编译 kernel 就可以了。重新编译后，kernel 会在 init_cond()函数中多进行一些和设备相关的工作：

- 初始化 i8259。由于 NEMU 中的 i8259 模拟实现是不可编程的，因此它没有注册端口 I/O 的回调函数，故 kernel 中对 i8259 的初始化并没有实际效果。
- 初始化串口。如果你的 out 指令实现正确，初始化串口后，你就可以在 kernel 中使用 Log()进行输出了。同时 SYS_write 系统调用也不需要通过"陷入"NEMU 来输出了，修改 kernel 中 sys_write()的代码，通过 serial_printc()把 SYS_write 系统调用中 buf 的内容输出到串口。
- 初始化 IDE 驱动程序。kernel/src/driver/ide 中实现了 IDE 驱动程序。初始化工作包括：
  - 初始化 IDE 驱动程序的高速缓存。
  - 把 ide_writeback()函数加入到时钟中断的中断处理函数。每次时钟中断到达的时候，ide_writeback()将会被调用，它负责每经过 1 秒将高速缓存中的脏块写回磁盘，进行写数据的同步。
  - 把 ide_intr()函数加入到磁盘中断的中断处理函数. 每次磁盘中断到达的时候，ide_intr()将会被调用，它负责设 has_ide_intr 标志，记录磁盘中断的到来。
- 此时内核的底层初始化操作已经全部完成，可以打开中断。
- IDE 驱动程序封装了磁盘读写的功能，并向上层提供了 ide_read()和 ide_write()两个方便使用的接口来读写磁盘。端口 I/O 的功能实现正确后，我们已经可以使用"真正"的磁盘，而不需要使用 ramdisk 了。定义宏 HAS_DEVICE 后，kernel/src/elf/elf.c 中的一处代码会把磁盘开始的 4096 字节读入一个缓冲区中，这 4096 字节已经包含了 ELF 头部和 program header table 了。**你需要修改加载 loader 模块的代码**，从磁盘读入每一个 segment 的内容. 修改后，把 nemu/include/common.h 中定义的宏 USE_RAMDISK 注释掉，ramdisk 就可以退休了。
- 为用户进程创建 video memory 的虚拟地址空间。在 loader()函数中有一处代码会调用 create_video_mapping()函数（在 kernel/src/memory/vmem.c 中定义），为用户进程创建 video memory 的恒等映射，即把从 0xa0000 开始，长度为 320 * 200 字节的虚拟内存区间映射到从 0xa0000 开始，长度为 320 * 200 字节的物理内存区间。这是 PA3 中的一个选做任务，如果你之前没有实现的话，现在你需要面对它了。具体的，你需要定义一些页表（注意页表需要按页对齐，你可以参考

kernel/src/memory/kvm.c 中的相关内容)，然后填写相应的页目录项和页表项即可。注意你不能使用 mm_malloc()来实现 video memory 映射的创建，因为 mm_malloc()分配的物理页面都在 16MB 以上，而 video memory 位于 16MB 以内，故使用 mm_malloc()不能达到我们的目的。如果创建地址空间和内存映射 I/O 的实现都正确，你会看到屏幕上输出了一些测试时写入的颜色信息，同时 video_mapping_read_test()将会通过检查。

到此为止，随着设备这最后一块拼图的加入，NEMU 的基本功能都已经实现好了，最后我们通过往 NEMU 中移植两个游戏来测试实现的正确性。

移植打字小游戏

框架代码中的 game 目录下包含两款游戏，共用的部分存放在 game/src/common 目录下，游戏各自的逻辑分别存放在 game/src/typing 和 game/src/nemu-pal 中。可以通过修改 game/Makefile.part 中的 GAME 变量在两个游戏之间切换(需要重新编译)。

我们对打字小游戏的初始化部分进行一些说明：

- 程序入口是 lib-common/uclibc/lib/crt1.o 中的_start()函数。
- _start()函数会调用 lib-common/uclibc/lib/libc.a 中的__uClibc_main()函数，进行一系列运行时环境相关的初始化工作。其中会调用 ioctl()系统调用来检查 stdin，stdout，stderr 是否为字符设备.由于除此之外，在 NEMU 中运行的程序不会再调用 ioctl()，为了简单起见，我们只让内核实现了 ioctl()中我们目前需要的功能，而其它功能并未实现。
- 运行时环境初始化结束后，就会跳转到游戏入口 game/src/common/main.c 中的 main()函数。
- init_timer()函数用于设置 100Hz 的时钟频率，但由于 NEMU 中的时钟模拟实现是不可编程的，而且模拟实现的时钟的默认频率就是 100Hz，故此处的 init_timer()函数并没有实际作用。
- 如果 PA2 实现了调用 init_FLOAT_vfprintf()劫持 vfprintf()函数，就可以输出 FLOAT 类型变量帮助调试。
- 在游戏中，add_irq_handle()是一个人为添加的系统调用，其系统调用号是 0，用于注册一个中断处理函数。已经注册的中断处理函数会在相应中断到来的时候被内核调用，这样游戏代码就可以通过中断来控制游戏的逻辑了。但在真实的操作系统中，提供这样的系统调用是非常危险的：恶意程序可以注册一个陷入死循环的中断处理函数，由于操作系统处理中断的时候，处理器一般都处于关中断状态，若此时陷入了死循环，操作系统将彻底崩溃。
- 使用 Log()宏输出一句话。在游戏中，通过 Log()宏输出的信息都带有{game}的标签，方便和 kernel 中的 Log()宏输出区别开来。
- 进入游戏逻辑主循环. 整个游戏都在中断的驱动下运行。

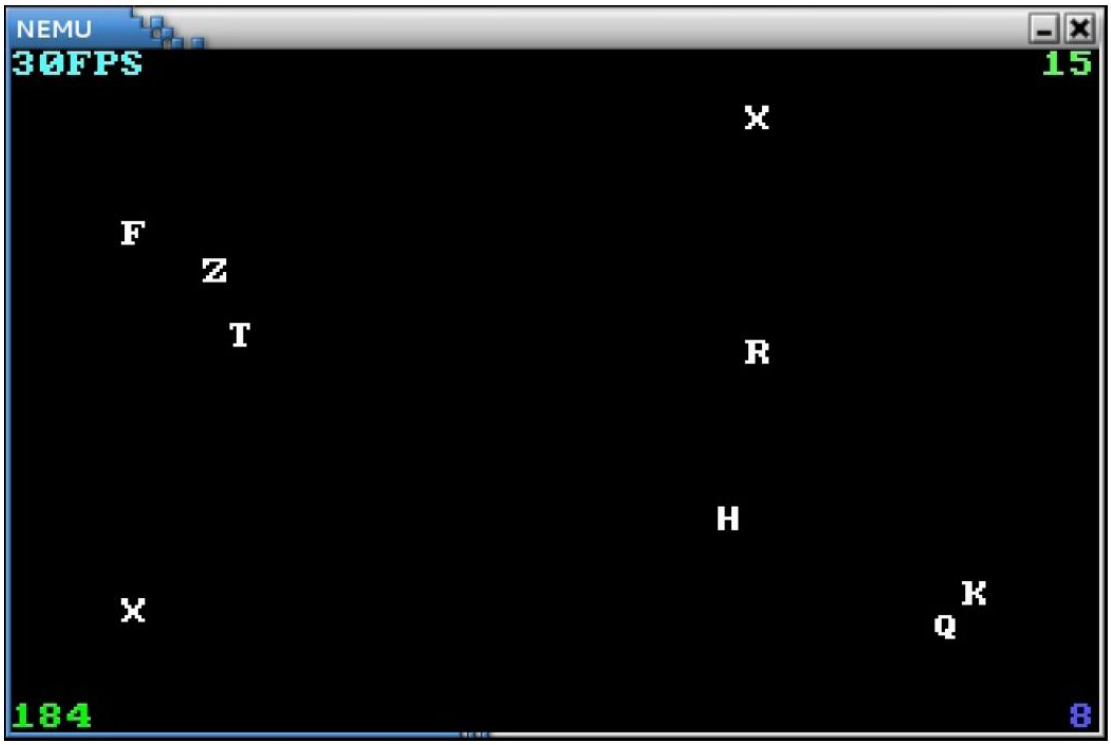
在工程目录下运行 make game 命令编译游戏，然后修改 Makefile 文件，把 USERPROG 变量设置为$(game_BIN)，让编译得到的可执行文件 game 作为 NEMU 的用户程序来运行。

如果你之前的实现正确，你将会看到打字游戏的画面，但你会发现按键后出现 system panic 的信息，这是因为 kernel 中没有对键盘中断进行响应，认为键

盘中断是一个非法的中断。

必做任务 5：相应键盘中断

你需要找出引起 system panic 的原因，然后根据你对中断响应过程的理解，在 kernel 中添加相应的代码来响应键盘中断。
添加成功后，按键后不再出现 system panic 的信息，但游戏却没有对按键进行响应，这是因为游戏并没有为键盘中断注册相应的中断处理函数，你还需要在游戏初始化的时候（game/src/common/main.c）为游戏注册键盘中断处理函数 keyboard_event()。
这些功能都实现之后，你会得到一款完整的打字游戏。



温馨提示

PA4 阶段 2 到此结束。

移植仙剑游戏

原版的仙剑奇侠传是针对 Windows 平台开发的，因此它并不能在 GNU/Linux 中运行(你知道为什么吗？)，也不能在 NEMU 中运行。网友 weimingzhi 开发了一款基于 SDL 库，跨平台的仙剑奇侠传，工程叫 SDLPAL.你可以通过 git clone 命令把 SDLPAL 克隆到本地，然后把仙剑奇侠传的数据文件（密码：yU9p）放在工程目录下，执行 make 编 SDLPAL，编译成功后就可以玩了。更多的信息请参考 SDLPAL 工程中的 README 说明。

把仙剑奇侠传移植到 NEMU 中的主要工作，就是把应用层之下提供给仙剑奇侠传的所有 API 重新实现一遍，因为这些 API 大多都依赖于操作系统提供的运行时环境，我们需要根据 NEMU 和 kernel 提供的运行时环境重写它们。主要包括以下四部分内容:

- C 标准库
- 浮点数
- SDL 库
- 文件系统

uclibc 已经提供了 C 标准库的功能，我们之前实现的 FLOAT 类型也已经解决了浮点数的问题，因此我们可以很简单地对这两部分内容进行移植，重点则落到了 SDL 库和文件系统的移植工作中。

关于浮点数有一点小小的补充。在 game/src/nemu-pal/battle/fight.c 的代码中有一处调用了 pow()函数，用于根据角色的敏捷度(身法)计算在半即时战

斗模式中角色的行动条变化量。在 SDLPAL 中，此处调用的 pow()函数计算的是 x^0.3，但 FLOAT 版本的通用 pow()函数实现相对麻烦，根据 KISS 法则，我们把此处调用修改成 x^(1/3)，lib-common/FLOAT.c 中实现的 pow()函数用来专门计算 x^(1/3)，采用的是 nth root algorithm。

我们把待移植的仙剑奇侠传称为 NEMU-PAL。NEMU-PAL 在 SDLPAL 的基础上经过少量修改得到，包括去掉了声音，修改了 game/src/nemu-pal/device/input.c 中和按键相关的处理，把我们关心的和 SDL 库的实现整理到 game/src/nemu-pal/hal 目录下，一些我们不必关心的实现则整理 game/src/nemu-pal/unused 目录下，同时还对浮点数用 binary scaling 进行了处理。

为了编译 NEMU-PAL，你需要修改 game/Makefile.part 中的 GAME 变量，从打字小游戏切换到 NEMU-PAL。然后把仙剑奇侠传的数据文件放在 game/src/nemu-pal/data 目录下，工程目录下执行 make game 即可。

下面来谈谈移植工作具体要做些什么。在这之前，请确保你已经理解打字小游戏的工作方式。

重写 SDL 库的 API

在 SDLPAL 中，SDL 库负责时钟，按键，显示和声音相关的处理. 由于在 NEMU 中没有模拟声卡的实现，NEMU-PAL 已经去掉了和声音相关的部分。其余三部分的内容被整理到 game/src/nemu-pal/hal 目录下，其中 HAL(Hardware Abstraction Layer)是硬件抽象层的意思，和硬件相关的功能将在 HAL 中被打包，提供给上层使用。

时钟相关

- SDL_GetTicks()用于返回用毫秒表示的当前时间(从运行游戏时开始计算)。
- SDL_Delay()用于延迟若干毫秒。
- jiffy 变量记录了时钟中断到来的次数，通过它可以实现上述和时钟相关的控制功能。

键盘相关

键盘通常都支持"重复按键"，即若一直按着某一个键不松开，键盘控制器将会不断发送该键的扫描码. 但是 SDLPAL(包括待移植的 NEMU-PAL)的游戏逻辑是在基于"非重复按键"的特性编写的，即若一直按着某一个键不松开，SDLPAL 只会收到一次该键的扫描码. 因此 HAL 需要把键盘的"重复按键"特性屏蔽起来，向上层提供"非重复按键"的特性.

- 实现这一抽象的方法是记录按键的状态. 你需要在键盘中断处理函数 keyboard_event()中编写相应代码，根据从键盘控制器得到的扫描码记录按键的状态.
- process_keys()函数会被 NEMU-PAL 轮询调用. 每次调用时，寻找一个刚刚按下或刚刚释放的按键，并调用相应的回调函数，然后改变该按键的状态. 若找到这样的按键，函数马上返回 true；若找不到，函数返回 false。注意返回之前需要打开中断。
- 代码中提供了数组的实现方式用于记录按键的状态，你也可以使用其它方式来实现上述抽象。

显示相关

SDL 中包含很多和显示相关的 API，为了重写它们，你首先需要了解它们的功能。通过 man 命令查阅以下内容：

- SDL_Surface
- SDL_Rect
- SDL_BlitSurface
- SDL_FillRect

在 game/src/nemu-pal/include/hal.h 中已经定义了相关的结构体，你需要阅读 man，了解相关成员的功能，然后实现 game/src/nemu-pal/hal/video.c 中相应函数的功能. 你可以忽略 man 中提到的"锁"等特性，我们并不打算在 NEMU-PAL 中实现这些特性。

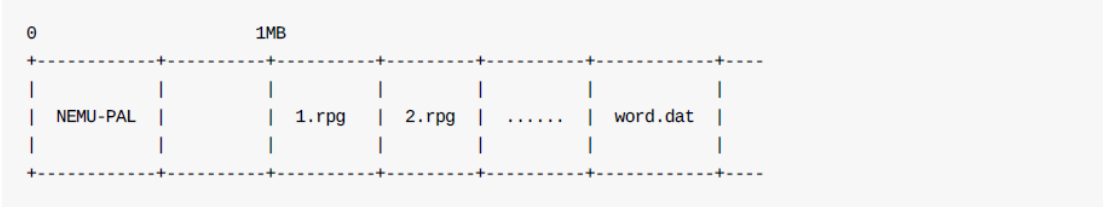
实现简易文件系统

对于大部分游戏来说，游戏用到的数据所占的空间比游戏逻辑本身还大，因此这些数据一般都存储在磁盘中。IDE 驱动程序已经为我们屏蔽了磁盘的物理特性，并提供了读写接口，使得我们可以很方便地访问磁盘某一个位置的数据。但为了易于上层使用,我们还需要提供一种更高级的抽象，那就是文件。

文件的本质就是字节序列，另外还由一些额外的属性构成。在这里，我们只讨论磁盘上的文件。这样，那些额外的属性就维护了文件到磁盘存储位置的映射。为了管理这些映射，同时向上层提供文件操作的接口，我们需要在 kernel 中实现一个文件系统。

不要被"文件系统"四个字吓到了，我们需要实现的文件系统并不是那么复杂，这得益于 NEMU-PAL 的一些特性：对于大部分数据文件来说，NEMU-PAL 只会读取它们，而不会对它们进行修改；唯一有可能进行文件写操作的，就只有保存游戏进度，但游戏存档的大小是固定的。因此我们得出了一个重要的结论：我们需要实现的文件系统中，所有文件的大小都是固定的。既然文件大小是固定的，我们自然也可以把每一个文件分别固定在磁盘中的某一个位置. 这些很好的特性大大降低了文件系统的实现难度，当然，真实的文件系统远远比这个简易文件系统复杂。

我们约定磁盘的最开始用于存放 NEMU-PAL 游戏程序，从 1MB 处开始一个挨着一个地存放数据文件：



kernel/src/fs/fs.c 中已经列出了所有数据文件的信息，包括文件名，文件大小和文件在磁盘上的位置。但若只有这些信息，文件系统还是不能表示文件在读写时的动态信息，例如读写位置的指针等。为此，文件系统需要为那些打开了的文件维护一些动态的信息：

```
typedef struct {
        bool opened;
        uint32_t offset;
} Fstate;
```

在这里，我们只需要维护打开状态 opened 和读写指针 offset 即可。由于这个简易文件系统中的文件数目是固定的，我们可以为每一个文件静态分配一个 Fstate 结构，因此我们只需要定义一个长度为 NR_FILES + 3 的 Fstate 结构数组即可. 这里的 3 包括 stdin, stdout, stderr 三个特殊的文件，磁盘中的第 k 个文件固定使用第 k + 3 个 Fstate 结构. 这样，我们就可以把 Fstate 结构在数组中的下标作为相应文件的文件描述符(fd, file descriptor)返回给用户层了。

有了 Fstate 结构之后，我们就可以实现以下的文件操作了：

```
int fs_open(const char *pathname, int flags);   /* 在我们的实现中可以忽略flags */
int fs_read(int fd, void *buf, int len);
int fs_write(int fd, void *buf, int len);
int fs_lseek(int fd, int offset, int whence);
int fs_close(int fd);
```

这些文件操作实际上是相应的系统调用在内核中的实现，你可以通过 man 查阅它们的功能，例如

```
man 2 open
```

其中 2 表示查阅和系统调用相关的 man page. 实现这些文件操作的时候注意以下几点：

- 由于简易文件系统中每一个文件都是固定的，不会产生新文件，因此 "fs_open()没有找到 pathname 所指示的文件"属于异常情况，你需要使用

assertion 终止程序运行。

- 使用 ide_read()和 ide_write()来进行文件的读写。
- 由于文件的大小是固定的，在实现 fs_read()和 fs_lseek()的时候，注意读写指针不要越过文件的边界。
- 除了写入 stdout 和 stderr 之外(即输出到串口)，其余对于 stdin，stdout 和 stderr 这三个特殊文件的操作可以直接忽略。

最后你还需要在 kernel 中编写相应的系统调用，来调用相应的文件操作，同时修改 game/src/common/lib/syscall.c 中的代码，为用户进程开放系统调用接口。

必做任务 6：把仙剑奇侠传移植到 NEMU

终于到了激动人心的时刻了：根据上述讲义内容，完成仙剑奇侠传的移植. 在我们提供的数据文件中包含一些游戏存档，可以读取迷宫中的存档，与怪物进行战斗，来测试实现的正确性。

不再神秘的秘籍

网上流传着一些关于仙剑奇侠传的秘技，其中的若干条秘技如下：

1. 很多人到了云姨那里都会去拿三次钱，其实拿一次就会让钱箱爆满！你拿了一次钱就去买剑把钱用到只剩一千多，然后去道士那里，先不要上楼，去掌柜那里买酒，多买几次你就会发现钱用不完了。

2. 不断使用乾坤一掷(钱必须多于五千文)用到财产低于五千文,钱会暴增到上限如此一来就有用不完的钱了。

3. 当李逍遥等级到达 99 级时，用 5～10 只金蚕王，经验点又跑出来了，而且升级所需经验会变回初期 5～10 级内的经验值，然后去打敌人或用金蚕王升级，可以学到灵儿的法术（从五气朝元开始）；升到 199 级后再用 5～10 只金蚕王，经验点再跑出来，所需升级经验也是很低，可以学到月如的法术（从一阳指开始）；到 299 级后再用 10～30 只金蚕王，经验点出来后继续升级，可学到阿奴的法术（从万蚁蚀象开始）。

假设这些上述这些秘技并非游戏制作人员的本意，请尝试解释这些秘技为什么能生效。

温馨提示

PA4 到此结束。

编写不朽的传奇

到此为止，我们已经把 ISA 层次中的指令系统，存储管理，中断/异常，I/O 这四个方面全部都讨论完了；仙剑奇侠传的移植工作也向你展示了 ISA 如何把软件硬件联系起来，从而支持一个游戏的运行。我们说"操作系统运行在 ISA 上"，其实是指操作系统的运行需要这四个方面的支持。至于操作系统如何在利用 ISA 的同时给上层应用程序提供支撑和服务，操作系统课程将会带你探索这个问题。ICS 已经为操作系统之旅扫清了障碍，准备好踏上新的旅程吧！

万变之宗 - 重新审视计算机

什么是计算机？为什么看似平淡无奇的一堆机械，竟然能够搭出如此缤纷多彩的计算机世界？那些酷炫的游戏画面，究竟和冷冰冰的电路有什么关系？

问题的答案其实已经蕴涵在上图中了，你能把"一堆沙子"(Physics，材料)和"玩游戏"(Application，应用)这两个看似毫不相关的概念联系起来吗？如果不能，你觉得还缺少些什么？

世界诞生的故事 - 终篇

感谢你帮助上帝创造了这个美妙的世界！同时也为自己编写了一段不朽的传奇！也希望你可以和我们分享成功的喜悦！^_^故事到这里就告一段落了，PA 也将要结束，但对计算机的探索并没有终点。如果你想知道这个美妙世界后来的样子，可以翻一翻 IA-32 手册。又或许，你可以用上帝赋予你的创造力，来改变这个美妙世界的轨迹，书写故事新的篇章。

任务自查表

序号	是否已完成
必做任务 1	
必做任务 2	
必做任务 3	
必做任务 4	
必做任务 5	
必做任务 6	

实验提交说明

提交地址

阶段性工程提交至自建 git 远程仓库

最终工程和实验报告提交智慧树平台

提交方式

1. 实验报告命名为“学号-PA4.pdf”，上传至智慧树平台；

2. 教师将通过脚本自动检查运行结果，因此除非实验手册特别说明，不要修改工程中的任何脚本，包括 Makefile、test.sh。

- 我们会清除中间结果，使用原来的编译选项重新编译(包括 -Wall 和 -Werror)，若编译不通过，本次实验你将得 0 分(编译错误是最容易排除的错误，我们有理由认为你没有认真对待实验)。

3. 学生务必使用 make submit 命令将整个工程打包，导出到本地机器，再上传到智慧树平台。

- make submit 命令会用你的学号来命名压缩包，然后修改压缩包的名称为“学号-PA4.zip”。此外，不要修改压缩包内工程根目录的命名。为了防止出现编码问题，压缩包中所有文件名都不要包含中文。

git 版本控制

请使用 git 管理你的项目，并合理地手动提交的 git 记录。请你不定期查看自己的 git log，检查是否与自己的开发过程相符。git log 是独立完成实验的最有力证据，完成了实验内容却缺少合理的 git log，会给抄袭判定提供最有力的证据。

如果出现 git head 损毁现象，说明你拷贝了他人的代码，也按抄袭处理。

实验报告内容

你必须在实验报告中描述以下内容：

- 实验进度。按照“任务自查表”格式在线制作表格并填写。缺少实验进度的描述，或者描述与实际情况不符，将被视为没有完成本次实验。

- 思考题

你可以自由选择报告的其它内容。你不必详细地描述实验过程，但我们鼓励你在报告中描述如下内容：

- 你遇到的问题和对这些问题的思考、解决办法
- 实验心得

如果你实在没有想法，你可以提交一份不包含任何想法的报告，我们不会强求。但请不要

- 大量粘贴讲义内容
- 大量粘贴代码和贴图，却没有相应的详细解释(让我们明显看出是凑字数的)

来让你的报告看起来十分丰富，编写和阅读这样的报告毫无任何意义，你也不会因此获得更多的分数，同时还可能带来扣分的可能。