操作系统课程设计

齐划一 201605130105

同组者：无

一、主要内容

1. 探究“协处理器模拟”
2. 探究“块设备驱动程序”

二、协处理器模拟

2.1 概述

为了控制成本，Intel 80386 芯片中并没有 FPU 模块，而是由 80387 芯片单独提供浮点运算功能，所谓“协处理器”。当不存在 80387 芯片时，CPU 在执行浮点运算指令时，会直接触发 7 号中断“设备不存在”，拒绝计算。此时，操作系统可以选择直接显示错误信息（Linux 0.11），也可以选择用软件模拟 FPU 模块，将计算结果放置在恰当的寄存器中，并继续执行进程（Linux 0.12）。

从 80486 开始，FPU 集成在 CPU 内部。因此在 X86 指令集的处理器中，此问题不再存在。

但这并不意味着研究此问题是没有价值的。相当一部分 MIPS 和 ARM 指令集的处理器不集成 FPU，如以联发科公司的 MT762x 系列为代表的 MIPS 芯片，广泛集成在路由器中——显然 FPU 对于路由器来说是无用的——，而硬件发烧友对路由器进行“软件改造”时，往往会在 Linux 内核中加入浮点运算模拟功能，以便支持更多的应用程序，而其原理与 Linux 0.12 中的协处理器模拟基本一致。

另外，正是因为 FPU 的缺失对 X86 而言只是“昙花一现”，这为研究此机制带来了非常大的麻烦。研究过程并不是一帆风顺，最后以失败告终。

2.2 理想的计划

1. 模拟一个 80386 的环境，它不具有 80387 芯片，且会在执行浮点运算指令时，触发中断“设备不存在”。
2. 在恰当位置设置断点，输出中断前后的寄存器的内容，以及内核代码的执行流程。
3. 根据输出的各种信息，对其进行可视化展示。

2.3 模拟旧时环境

2.3.1 虚拟机的选择

市面上流行的虚拟机软件有 Virtual Box、VMware Workstation、QEMU、Bochs （读作 Box）等。其中，不利用处理器的“硬件虚拟化”功能，完全基于软件方式进行二进制模拟的主要是 QEMU 和 Bochs。显然，我们无法控制处理器的“硬件虚拟化”，只能采用后者。而 Bochs 之于 QEMU 而言，虽然更小众，但胜于文档清晰、源代码简洁，甚至直接提供了 disable-fpu 的编译选项（后面会提到）。因此，选择 Bochs 是更合理的。

2.3.2 编译 Bochs

Bochs 的最新版本是 2.6.9。从 https://sourceforge.net/p/forge/documentation/svn/ 上可以获取任意一个版本的 Bochs 的源代码。

在 Ubuntu 16.04 x86 上编译 Bochs。首先下载必要的库。

```
apt-get install libx11-dev libgtk2.0-dev build-essential
```

编译选项中提供了 disable-fpu 选项。于是，设置此选项，并开始编译。

```
./configure --enable-fpu=no --enable-static --enable-shared=no
make -j8
```

这里 `make -j8` 的意思是使用 8 线程编译，以加快速度。但是，如果遇到编译错误，输出信息会变得一团乱，此时应使用单线程编译。

编译报错，提示需要将 cpu-level 从默认的 6 改为最低的 3。按提示修改。

```
./configure --enable-fpu=no --enable-cpu-level=3 --enable-static --enable-shared=no
make
```

编译报错，在某行代码中，某个与“ZMM”有关的类未定义。查阅资料知这与 AVX512 指令集有关。由于本项目与 AVX 八杆子打不着，所以将相关代码直接注释掉。修改 `cpu/proc_ctrl.cc` 文件 311 行附近：

```
void BX_CPP_AttrRegparmN(1) BX_CPU_C::CLZERO(bxInstruction_c *i)
{
  bx_address eaddr = RAX & ~BX_CONST64(CACHE_LINE_SIZE-1) & i->asize_mask();

/*
  BxPackedZmmRegister zmmzero; // zmm is always made available even if EVEX is not
compiled in
  zmmzero.clear();
  for (unsigned n=0; n<CACHE_LINE_SIZE; n += 64) {
    write_virtual_zmmword(i->seg(), eaddr+n, &zmmzero);
  }
*/
}
```

编译成功。

尝试启动 Bochs。失败，无任何屏幕输出。

2.3.3 原因猜测

1. 因为绝大多数人不会关掉 FPU ，因此经历了这么多次版本迭代后， Bochs 的 disable-fpu 选项变成了摆设，已经与源代码起了冲突。
2. Bochs 自带的主板 BIOS 或显卡 BIOS 至少其中一个依赖 FPU。

经历过接下来的尝试之后，终于发现，两条原因都存在。

2.3.4 更换 Bochs 版本、更换 BIOS

Bochs 的版本基本上可以划分为两类：2.0版本之前（不含，下称为 1.X），与2.0版本之后（含，下称为 2.X）。

虽然前者的源代码仍然可以获取到，但是，编译过程中需要从特定的站点下载一些依赖文件，而这些文件早已不知去向。1.X 版本的 Bochs 名存实亡，已经无法成功编译。

从 Bochs 官方网址可以下载到两种 BIOS，一种称为 “latest”（最新版），一种称为“legacy”（过时的）。

在 disable-fpu 后的 Bochs 2.X 下的结果如下表：

BIOS 类型	能否亮机	能否进入 Linux 0.11	能否进入 Linux 0.12
latest	否	/	/
legacy	是	否	否

2.3.5 退路

原计划中的第一条已经无法实现。退路如下：

1. 不使用浮点运算相关的指令，而是选择其他未定义的、能引起中断的指令。
2. 尝试换用其他虚拟机软件。
3. 换用 ARM 或 MIPS 指令集。 综合考虑成本，只有第一条退路是可行的。

2.4 修正后的计划

1. 模拟一个正常的 X86 的环境。这里采用了陈宇翔同学的虚拟环境。
2. 程序调用 0x0F 0x0B 这个未定义的指令，期待处理器触发中断“指令不存在”。
3. 修改 Linux 内核代码，使其对“指令不存在”的情况进行特殊处理。
4. 在恰当位置设置断点，输出中断前后的寄存器的内容，以及内核代码的执行流程。
5. 根据输出的各种信息，对其进行可视化展示。

2.5 可执行文件的制作

2.5.1 a.out 格式与 ELF 格式

尝试直接从 32 位 Linux 系统静态编译可执行文件，并转移到 Linux 0.11 中。失败。

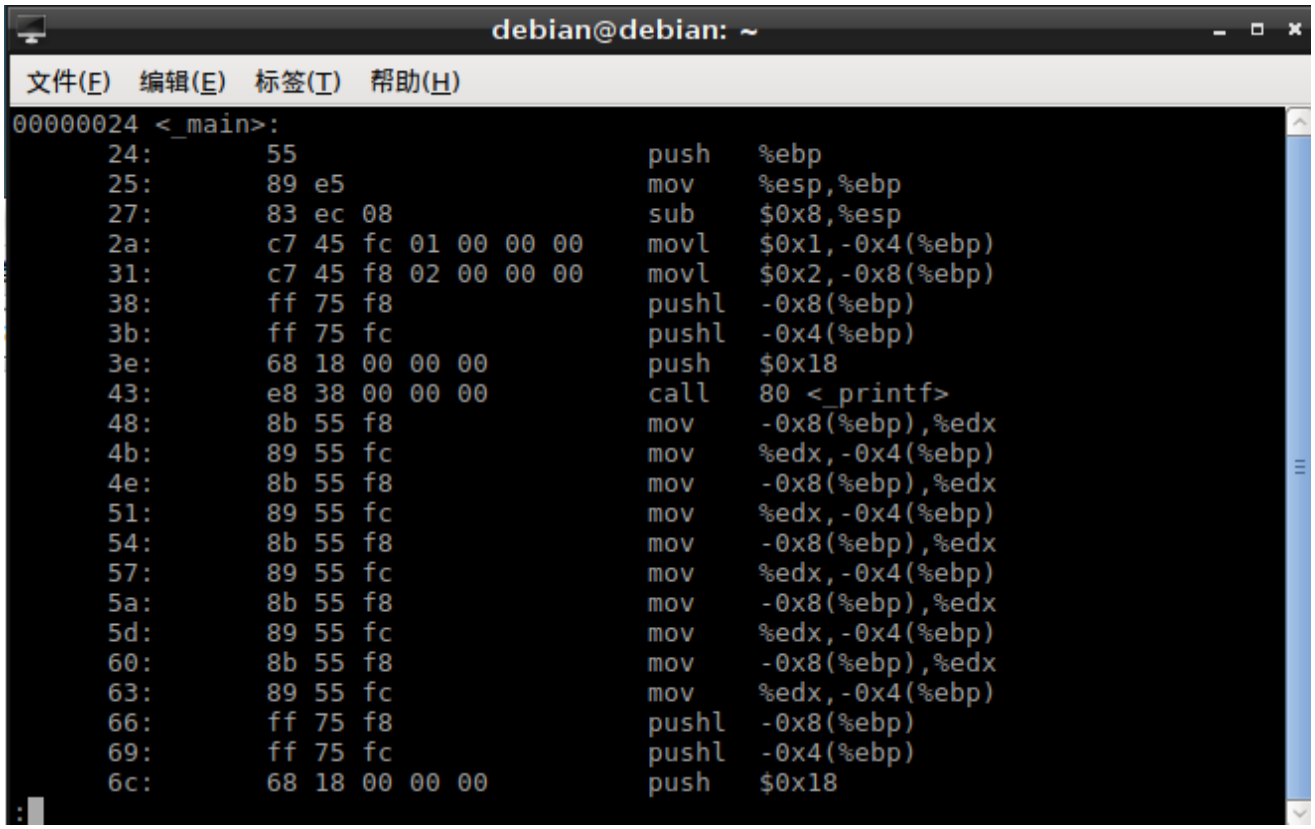
查阅资料得知，现在的 Linux 系统使用 ELF 作为可执行文件的格式，而早期的 Linux 系统采用 a.out 格式作为可执行文件格式。两者完全不同。

2.5.2 反编译、二进制编辑

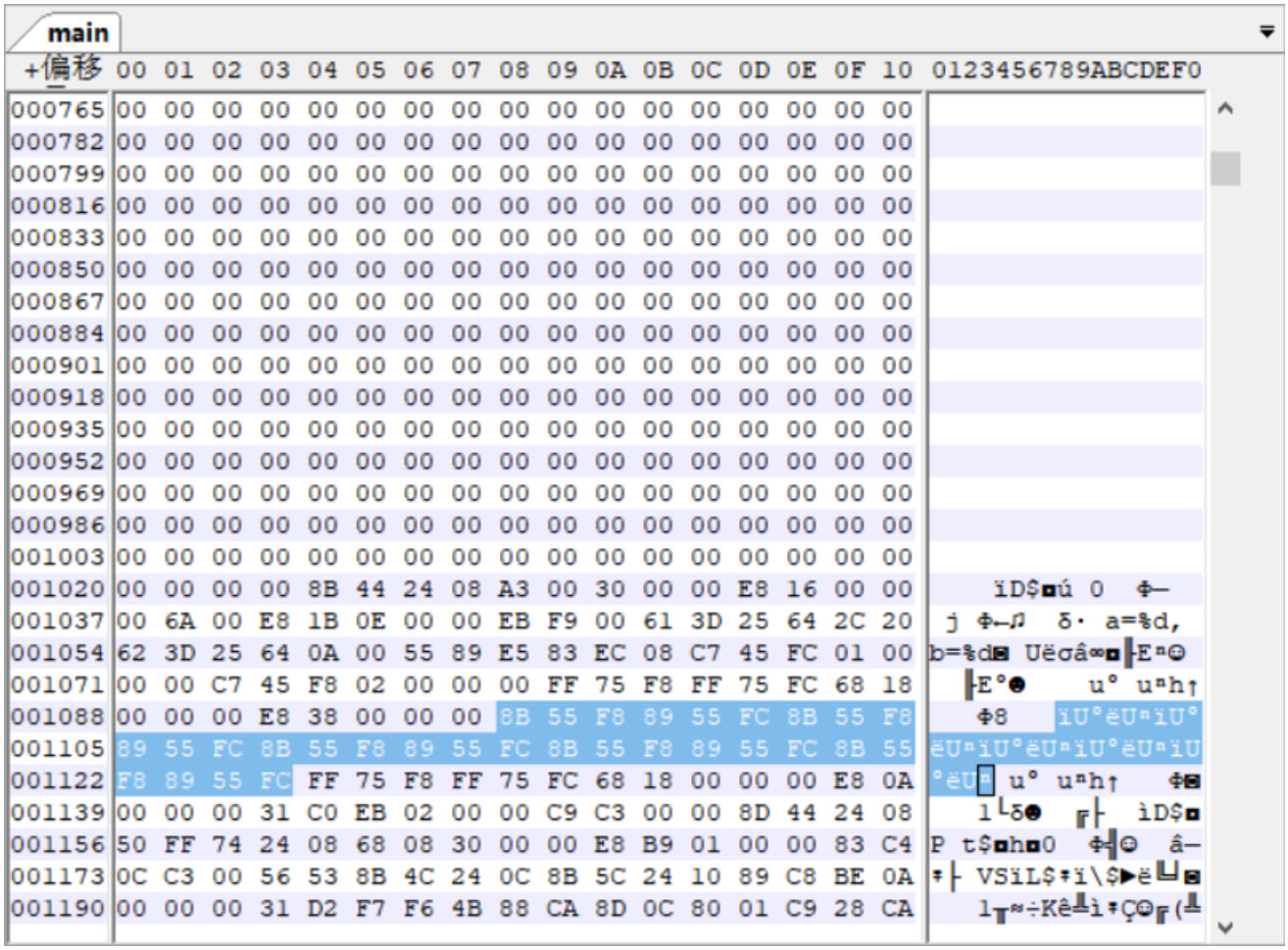
在 Linux 0.11 中，编译一个 main.c 文件，内容如下。

```c
#include <stdio.h>
int main(){
    int a=1;
    int b=2;
    printf("a=%d, b=%d\n",a,b);
    a=b;
    a=b;
    a=b;
    a=b;
    a=b;
    printf("a=%d, b=%d\n",a,b);
    return 0;
}
```

将编译好的可执行文件拷贝到虚拟机外。使用 `objdump` 工具对其反编译，从中得到了 `a=b;` 这句所对应的 6 个字节。



使用 `wxHexEditor` 工具编辑文件，将 `a=b` 对应的字节用 0x90 填充。0x90 就是 X86 中的 NOP，表示"什么也不做"。



接着，在 0x90 之中，选择一个合适的位置，用 0x0F 0x0B 替换之。

2.6 执行带有自定义指令的程序

2.6.1 将可执行文件复制回虚拟机

由于 Linux 0.11 的权限管理部分并不完善，如果直接将文件复制回去，则无法对此文件进行读写、删除、执行、更改权限等任何操作。如图所示。

```
[/usr/root]# chmod root:root main
chmod: invalid mode
[/usr/root]# ls -l
total 44
-rwx--x--x   1 root     root         20590 Nov 27 02:27 a.out
drwxr-xr-x   9 root     root           144 Nov 26 18:22 examples
-rw-------   1 1000     232          20590 Nov 26 18:29 main
-rw-r--r--   1 root     root           152 Nov 26 18:22 main.c
[/usr/root]# chown root:root main
chown: main: Not owner
[/usr/root]#
```

因此，需要在复制时，在虚拟机之外修改文件的所有者和权限。

```
chown root:root /path/to/file
chmod 777 /path/to/file
```

2.6.2 正常内核下的执行结果

在未经修改的 Linux 0.11 下，执行此文件，得到了期望的报错结果。Linux 内核提示 invalid opcode，并给出了此时的上下文。

2.6.3 修改内核

修改 `kernel/asm.s` 和 `kernel/traps.c` 文件，以便让内核对“指令不存在”中断做出额外处理：若引发此中断时，所执行的指令是 0x0F 0x0B，则忽略错误并继续；否则，按老办法处理。

```
int do_invalid_op(long esp, long error_code)
{
    long * esp_ptr = (long *) esp;
    char first = 0xff & get_seg_byte(esp_ptr[1],(0+(char *)esp_ptr[0]));
    char second = 0xff & get_seg_byte(esp_ptr[1],(1+(char *)esp_ptr[0]));
    if (first==0x0f && second==0x0b) {
        return 1;
    }
    die("invalid operand",esp,error_code);
    return 0;
}
```

当然，为简单起见，先对所有的“指令不存在”中断都选择忽略。如果能够成功，那么刚才的逻辑也不难实现。结合 no_error_code 的代码，将 invalid_op 代码改为如下：

```
invalid_op:
    pushl $do_invalid_op
    xchgl %eax,(%esp)
    pushl %ebx
    pushl %ecx
    pushl %edx
    pushl %edi
    pushl %esi
    pushl %ebp
    push %ds
    push %es
    push %fs
    pushl $0        # "error code"
    lea 44(%esp),%edx
    pushl %edx
    movl $0x10,%edx
    mov %dx,%ds
    mov %dx,%es
    mov %dx,%fs
    call *%eax
    addl $8,%esp
    pop %fs
    pop %es
    pop %ds
    popl %ebp
    popl %esi
    popl %edi
    popl %edx
    popl %ecx
```

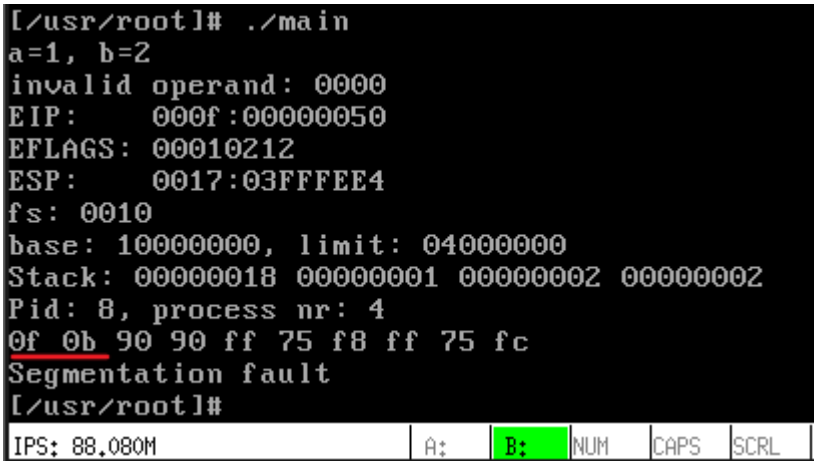
```
        popl %ebx
        popl %eax
        iret
```

由于 do_invalid_op 函数不再让内核结束进程并显示错误消息，理论上，进程应当可以继续执行。

2.6.4 事与愿违

在执行到自定义指令时，进程卡住，无任何输出。推测，处理器在“指令不存在”后，并未按正常流程给 EIP 加一。因此，卡在了这里。

尝试继续修改内核代码，以此恢复正常的 EIP。但是结果往往是触发了某些保护机制，根据修改的地方不同，可能是 `Segmentation fault` 或者是 `General Protection`。如图。



因此，此条路基本上走不通。

2.7 总结

协处理器模拟是块“硬骨头”。要在 80386 环境下较劲，需要付出很大的代价，很可能无法顺利完成。

如果后来者对这方面感兴趣，我强烈建议在 MT762x 路由器下研究 MIPS 指令集的 FPU 模拟。截至目前（2018年12月底），MT7621 芯片的“新路由3”二手价不足一百元，而 MT7620 芯片的“优酷路由宝”二手价不足四十元，或者几百元的 MT7621 开发板。以上硬件配合基于 Linux 4.x 的 OpenWRT 系统，是理想的研究对象。毕竟，在 ARM 和 MIPS 指令集下，FPU 模拟是 Linux 内核的一部分，而不像 X86 一样只是昙花一现。

三、块设备驱动程序

3.1 概述

常见的块设备包括软盘、机械硬盘、固态硬盘、U 盘等。

块设备驱动程序有一个重要（但未来可能不再重要）的功能，就是对一系列的读写请求进行合理排序，以便尽量减小磁头的移动。

顺序读取自然是效率最高的……吗？

3.1.1 固态硬盘和其他的闪存盘

闪存，是从 ROM、EEPROM 等技术上逐渐发展而来的，主要有 NOR Flash 和 NAND Flash 两种主流闪存类型。闪存的存储单元的写入寿命有次数限制。因此为了尽可能延长寿命，主控芯片会把新的数据选择合适的位置存放。这个过程是对操作系统透明的，也就是说，操作系统访问的地址，并非闪存存储单元的实际地址。主控芯片负责对这两种地址之间进行转换。另外，连续的地址，与不连续的地址相比，闪存的读取速度并不会得到提升。因此，对于闪存来说，维护地址连续完全没有意义。

因此，在 Windows 7 及更高版本的 Windows 系统上，碎片整理功能被完全禁用（因为减少寿命，而且不会得到速度提升）。同时期的 Linux 内核也对固态硬盘做了足够的优化。

3.1.2 传统的机械硬盘

对于机械硬盘而言，地址空间的连续可以节约磁头寻址的时间，因此维护地址连续是有意义的。

3.1.3 带有 SMR 技术的机械硬盘

为了尽可能提高磁盘的密度，SMR 技术正逐渐从服务器领域渗透到了家用领域。它具有和固态硬盘类似的性质，即写入数据时会连带把周围的数据也清除。因此，SMR 机械硬盘在物理层面上不能和传统机械硬盘一样任意地寻址，它的写入也是有“代价”的，它的主控芯片也会进行与固态硬盘类似的地址转换。

因此，对于 SMR 技术的机械硬盘，维护地址连续也没有意义。令人痛心的是，我们无法通过磁盘碎片整理来使得数据在物理上连续，因此——除非刻意地像使用磁带那样去使用 SMR 技术的机械硬盘——读取速度可能会非常缓慢。

另外一点，由于 SMR 技术比较新，所以操作系统几乎没有对 SMR 技术做优化，依然将其按普通机械硬盘对待。这只会增加硬盘的负荷。

SMR 技术的机械硬盘，同时拥有了机械硬盘的缺点和固态硬盘的缺点，不适合一般用途使用。购买硬盘或电脑时需仔细甄别，尽量避免。

3.2 Linux 0.11 的电梯算法

这里只考虑传统的机械硬盘，即地址空间的连续可以显著减少磁头寻址时间。在这种情况下，内核的块设备驱动程序应该尽可能优化读写请求，使得地址尽可能递增顺序访问。

Linux 0.11 在 `kernel/blk_drv/ll_rw_blk.c` 的 `add_request` 中对此做了优化。当读写请求特别多时，内核会用电梯算法给请求排序。

首先，对于指定的块设备而言，一个读写请求包含这几个关键的元素：

- 是读请求，还是写请求。
- 从哪个扇区开始读取。
- 一次读取几个扇区。

读写请求的其他元素与此算法无关，省略，相关内容定义在 `kernel/blk_drv/blk.h` 中。

首先，在 `kernel/blk_drv/blk.h` 中定义读写请求的大小关系。这个定义与考试排名时“若总成绩相同，则语文成绩高者名次更高”类似。

- 读请求比写请求更“小”。
- 如果同为读请求或同为写请求，设备号小者更小。（此条针对不同的块设备）
- 如果同为读请求或同为写请求，且设备号也相同，则扇区号小者更小。

接着，维护一个读写请求的链表。每当有新的读写请求加入时，根据算法加入链表的合适位置。如果设备已经处理完当前的请求，中断处理时会将链表的首元素移除，并让设备继续处理新的首元素。

3.3 数据的抓取和处理

3.3.1 设置断点

在 `kernel/blk_drv/ll_rw_blk.c` 的 `add_request` 中设置断点，以便抓取每个请求的数据，以及链表的情况。

```
static void add_request(struct blk_dev_struct * dev, struct request * req)
{
    struct request * tmp;

    req->next = NULL;
    cli();
    if (req->bh)
        req->bh->b_dirt = 0;
    if (!(tmp = dev->current_request)) {
        dev->current_request = req;
        sti();
        (dev->request_fn)();
        return;
    }
    //断点 1 所在位置
    for ( ; tmp->next ; tmp=tmp->next)
        if ((IN_ORDER(tmp,req) ||
            !IN_ORDER(tmp,tmp->next)) &&
            IN_ORDER(req,tmp->next))
            break;
    req->next=tmp->next;
    tmp->next=req;
    sti();
}
```

断点 1 的位置如图所示。不幸的是，断点 1 从未被触发。

3.3.2 原因猜测

硬盘的速度太快，一旦链表中有读写请求，会被立刻处理，导致链表最多有一个元素，永远形不成一条链。

3.3.3 半假半真

因此，将演示计划改为“半假半真”。

意思是，在函数的开头处设置断点，记录每一个读写请求的细节，但是，不记录链表的情况——因为最多有一个元素。取而代之的是，在演示时，手动控制硬盘控制器何时完成一个请求，手动控制下一个请求何时加入链表。这样，仍然可以将电梯算法进行展示。

3.3.4 数据的处理

利用陈宇翔同学的虚拟环境进行调试输出。数据略显杂乱，因此使用 Python 对信息进行处理，整理为 JSON 格式，供程序使用。相关数据和代码不在此处给出。

3.4 可视化程序

使用 C# 和 WPF 编写程序，以便展示读写请求、链表和算法的每个步骤。

值得一提的是，C# 的 `yield` 关键字支持函数执行到一般时，返回结果，而且保留相应的状态。等到下一次执行时，会从上次的状态开始继续执行。借助 C# 的这一特性，可以做到展示每次循环时的判断结果，使得算法更为直观。

相关代码不在此处给出。

界面如下。

“处理请求”和“添加请求”按钮，对应的就是手动控制的两个过程。用颜色来表示各种状态。

四、感想

1. 从整个探究过程可以看到，要想模拟出当年的运行状况，可能很难做，甚至可能做不到。比如说 80386 的 FPU 缺失的情况，以及块设备驱动程序中磁盘处理请求的速度。因此，在选题和实现时，应该对这些情况做出规避。
2. 探究过程中可能会有意想不到的问题，有些问题看似可以解决，实际上可能会很难，或者在有限时间内解决它“不划算”。应注意控制好时间和进度，必要时果断放弃。