目 录

1 进入内核前的准备 3
1.1 跳入保护模式 . . . 3
1.2 开启分页机制 . . . 4
1.3 重新放置内核 . . . 7
1.3.1 program header table . . . 8
1.4 重新放置内核 . . . 9
1.5 进入内核 . . . 12

2 内核雏形 13
2.1 定义相关的数据结构和数据类型 . . . 13
2.2 C 与汇编程序的相互调用 . . . 14
2.2.1 在汇编程序中调用 C 函数 . . . 14
2.2.2 在 C 程序中调用汇编函数 . . . 15
2.3 移动 GDT 到内核 . . . 16
2.3.1 memcpy 函数 . . . 16
2.3.2 移动 GDT . . . 17
2.4 显示字符串的函数 disp_str() . . . 18

3 kvm 环境的搭建 20
3.1 搭建硬件环境 . . . 20
3.1.1 检查处理器是否支持 VT 技术 . . . 20
3.1.2 设置 BIOS . . . 20
3.2 安装 KVM . . . 20
3.2.1 下载 KVM 源代码 . . . 20
3.2.2 配置 KVM . . . 21
3.2.3 编译 KVM . . . 22
3.2.4 安装 KVM . . . 23
3.2.5 加载 kvm 和 kvm_intel 模块 . . . 23
3.3 安装 qemu-kvm . . . 23
3.3.1 下载 qemu-kvm 源代码 . . . 23
3.3.2 配置 qemu-kvm . . . 24

3.3.3 编译 qemu-kvm . . . 24
3.3.4 安装 qemu-kvm . . . 24
3.4 安装客户机 . . . 24
3.4.1 创建镜像文件 . . . 24
3.4.2 安装客户机 . . . 25
3.4.3 查看客户机 . . . 25
3.5 启动 KVM 客户机 . . . 25

4 CPU 配置 27
4.1 -smp 参数项 . . . 27
4.2 查看 cpu 配置 . . . 27
4.2.1 在客户机中查看 cpu 信息 . . . 27
4.2.2 使用 qemu 监控客户机 cpu 信息 . . . 28
4.3 -cpu 参数项 . . . 28
4.4 vCPU 的绑定 . . . 29
4.4.1 隔离宿主机 CPU . . . 29
4.4.2 绑定客户机 vCPU . . . 30

5 内存配置 32
5.1 -m 参数项 . . . 32
5.2 查看内存信息 . . . 32
5.3 EPT 扩展页表 . . . 32
5.4 -mem-path 参数项 . . . 33

1 进入内核前的准备

1.1 跳入保护模式

一个操作系统从开机到开始运行，需要经历“引导，加载内核进入内存，跳入保护模式，开始执行内核”。这句话我在上一次的学习报告的开头说过，上一次学习报告完成了前两个部分：引导，加载内核进入内存。我在前几次的学习报告中已经讨论过保护模式的进入，接下来的代码更多的是走一下流程，感受一下如何在 Loader 中跳入保护模式。

我们定义了两个段描述符，一个是可执行段，一个是可读写段。这两个段的大小都是 4GB，基址为 0，所以之后主要靠段内的偏移来进行程序的转移。在感受代码前，我们应该意识到，Loader 模块的物理基址是 BaseOfLoader*10h+OffsetOfLoader。所以在 Loader 文件中的标号的实际物理地址都是该物理基址加上标号在文件中的偏移。

```
1   BaseOfLoader equ 09000h
2   OffsetOfLoader equ 0100h
3   ; BaseOfLoader是段基址，所以实际物理地址需要左移4位
4   BaseOfLoaderPhyAddr equ BaseOfLoader * 10h + OffsetOfLoader
5
6   LABEL_GDT:   Descriptor 0, 0, 0
7   LABEL_DESC_FLAT_C:   Descriptor 0, 0fffffh, DA_CR | DA_32 | DA_LIMIT_4K
8   LABEL_DESC_FLAT_RW:  Descriptor 0, 0fffffh, DA_DRW | DA_32 | DA_LIMIT_4K
9
10  GdtLen equ $ - LABEL_GDT
11  GdtPtr dw GdtLen - 1
12         dd BaseOfLoaderPhyAddr + LABEL_GDT
13
14  SelectorFlatC equ LABEL_DESC_FLAT_C - LABEL_GDT
15  SelectorFlatRW equ LABEL_DESC_FLAT_RW - LABEL_GDT
16  ; ...
17  LABEL_FILE_LOADED:
18      ; ...
19      lgdt [GdtPtr]
20      cli
21
22      in al, 92h
23      or al, 00000010b
24      out 92h, al
25
26      mov eax, cr0
27      or eax, 1
28      mov cr0, eax
29
30      jmp dword SelectorFlatC:(BaseOfLoaderPhyAddr+LABEL_PM_START)
31
32  [SECTION .s32]
33  ALIGN 32
34  [BITS 32]
35  LABEL_PM_START:
36      $
```

1.2 开启分页机制

既然我们进入了保护模式，就能开启分页机制了。这里开启分页机制的思路和前几次学习报告中的思路是一样的，所以在代码中不会有很多注释。

在此重温一下 ARDS 数据结构：

偏移	名称	意义
0	BaseAddrLow	基地址的低 32 位
4	BaseAddrHigh	基地址的高 32 位
8	LengthLow	长度的低 32 位
12	LengthHigh	长度的高 32 位
16	Type	这个地址范围的地址类型

其中，Type 的取值有三种类型：

1	表示这个内存段是一段可以被 OS 使用的 RAM
2	表示这个地址段正在被使用或者被系统保留，所以一定不可以被 OS 使用。
其他	表示这个内存段被保留，留作以后使用，不可以被 OS 使用。

```
1   LABEL_START:
2       mov ax, cs
3       mov ds, ax
4       mov es, ax
5       mov ss, ax
6       mov sp, BaseOfStack
7
8       ; int 15h，将ARDS结构体读入es:di所指的位置
9       ; ebx用于寻找下一个地址范围描述符结构ARDS。首次调用int 15h，ebx的值应该设为
            0
10      mov ebx, 0
11      mov di, _MemChkBuf
12  .MemChkLoop:
13      mov eax, 0E820h
14      ; ARDS共20字节，所以通过ecx指定int 15h向es:di读入20字节的内容
15      mov ecx, 20
16      mov edx, 0534D4150h
17      int 15h
18      ; 当CF=1时，表示发生错误。
19      jc .MemChkFail
20      ; 让es:di指向缓冲区的下一个20字节
21      add di, 20
22      inc dword [_dwMCRNumber]
23      ; 如果ebx的值为0，代表着当前的ARDS是最后一个地址范围描述符。
```

```
24          cmp ebx, 0
25          jne .MemChkLoop
26          jmp .MemChkOk
27      .MemChkFail:
28          mov dword [_dwMCRNumber], 0
29      .MemChkOk:
30          ; ...
31
32      [SECTION .s32]
33      ALIGN
34      [BITS 32]
35      LABEL_PM_START:
36          mov ax, SelectorFlatRW
37          mov ds, ax
38          mov es, ax
39          mov fs, ax
40          mov ss ,ax
41          mov esp, TopOfStack
42
43          call DispMemInfo
44          call SetupPaging
45
46          jmp $
47
48      DispMemInfo:
49          push esi
50          push edi
51          push ecx
52          ; ds:si指向存放ARDS的数据缓冲区
53          mov esi, MemChkBuf
54          ; ecx存放着ARDS的数目
55          mov ecx, [dwMCRNumber]
56      .loop:
57          ; ARDSStruct用于存放数据缓冲区当前指向的ARDS结构
58          ; edi指向ARDS结构体的成员
59          mov edx, 5
60          mov edi, ARDSStruct
61      .1:
62          mov eax, dword [esi]
63          ; 将eax中存放的数据放入es:edi中，edi自动加4
64          stosd
65          ; 用于指向数据缓冲区的下一个ARDS结构
66          add esi, 4
67          dec edx
68          cmp edx, 0
69          jnz .1
70          cmp dword [dwType], 1
71          jne .2
72          ; 这里只需要使用低32位的基地址和长度，因为高32位没有用到
73          ; 基地址加长度的最大值就是内存空间的大小
74          mov eax, [dwBaseAddrLow]
75          add eax, [dwLengthLow]
76          cmp eax, [dwMemSize]
77          jb .2
78          mov [dwMemSize], eax
79      .2:
```

```
80          ; 使用loop指令会检查ecx是否为0，并将ecx减一
81          loop .loop
82
83          pop ecx
84          pop edi
85          pop esi
86          ret
87
88      SetupPaging:
89          xor edx, edx
90          mov eax, [dwMemSize]
91          ; 一个页表对应4k*1024的内存大小
92          mov ebx, 400000h
93          div ebx
94          mov ecx, eax
95          ; 检查edx是否为0。test指令将执行and操作，不过不会保存执行的结果
96          test edx, edx
97          jz .no_remainder
98          ; 余数不为0的话，页表数目加一
99          inc ecx
100     .no_remainder:
101         push ecx
102
103         mov ax, SelectorFlatRW
104         mov es, ax
105         mov edi, PageDirBase
106         xor eax, eax
107         mov eax, PageTblBase | PG_P | PG_USU | PG_RWW
108     .1:
109         ; 将eax中的内容存入es:eax中
110         stosd
111         add eax, 4096
112         loop .1
113
114         pop eax
115         mov ebx, 1024
116         mul ebx
117         ; 需要初始化的页帧数目为1024*页表数
118         mov ecx, eax
119         mov edi, PageTblBase
120         xor eax, eax
121         mov eax, PG_P | PG_USU | PG_RWW
122     .2:
123         stosd
124         add eax, 4096
125         loop .2
126
127         ; 将页目录基址存入cr3中
128         mov eax, PageDirBase
129         mov cr3, eax
130         ; 将cr0的PG位置一
131         mov eax, cr0
132         or eax, 80000000h
133         mov cr0, eax
134         jmp short .3
135     .3:
```

```
136         nop
137         ret
138
139     [SECTION .data1]
140     ALIGN 32
141
142     LABEL_DATA:
143     _szMemChkTitle: db "BaseAddrL BaseAddrH LengthLow LengthHigh   Type", 0Ah, 0
144     _szRAMSize: db "RAM size:", 0
145     _szReturn: db 0Ah, 0
146
147     ; 用于存放ARDS结构的数量
148     _dwMCRNumber: dd 0
149     ; 用于存放可用内存的大小
150     _dwMemSize: dd 0
151     ; 代表了ARDS结构
152     _ARDSStruct:
153         _dwBaseAddrLow: dd 0
154         _dwBaseAddrHigh: dd 0
155         _dwLengthLow: dd 0
156         _dwLengthHigh: dd 0
157         _dwType: dd 0
158     _MemChkBuf: times 256 db 0
159
160     szMemChkTitle equ BaseOfLoaderPhyAddr + _szMemChkTitle
161     szRAMSize equ BaseOfLoaderPhyAddr + _szRAMSize
162     szReturn equ BaseOfLoaderPhyAddr + _szReturn
163     ; 代表了_dwMCRNumber的地址
164     dwMCRNumber equ BaseOfLoaderPhyAddr + _dwMCRNumber
165     ; 代表了_dwMemSize的地址
166     dwMemSize equ BaseOfLoaderPhyAddr + _dwMemSize
167     ; 代表了ARDS结构各成员的地址
168     ARDSStruct equ BaseOfLoaderPhyAddr + _ARDSStruct
169         dwBaseAddrLow equ BaseOfLoaderPhyAddr + _dwBaseAddrLow
170         dwBaseAddrHigh equ BaseOfLoaderPhyAddr + _dwBaseAddrHigh
171         dwLengthLow equ BaseOfLoaderPhyAddr + _dwLengthLow
172         dwLengthHigh equ BaseOfLoaderPhyAddr + _dwLengthHigh
173         dwType equ BaseOfLoaderPhyAddr + _dwType
174     MemChkBuf equ BaseOfLoaderPhyAddr + _MemChkBuf
175
176     PageDirBase equ 100000h
177     PageTblBase equ 101000h
178
179     StackSpace: times 1024 db 0
180     TopOfStack equ BaseOfLoaderPhyAddr + $
```

1.3 重新放置内核

我们在此需要根据内核的 program header table 的信息进行内存复制。上一次学习报告中 ELF 文件格式的内容虽然很多，但是太晦涩难懂了。在此现在复习一遍 program header table。

1.3.1 program header table

program header 的定义如下：

```
typdef struct
{
    ELf32_Word p_type;
    ELf32_Off p_offset;
    ELf32_Addr p_vaddr;
    ELf32_Addr p_paddr;
    ELf32_Word p_filesz;
    ELf32_Word p_memsz;
    ELf32_Word p_flags;
    ELf32_Word p_align;
}ELf32_Phdr;
```

program header 描述的是系统准备程序运行所需的一个段的信息。结构体中各成员的意义如下：

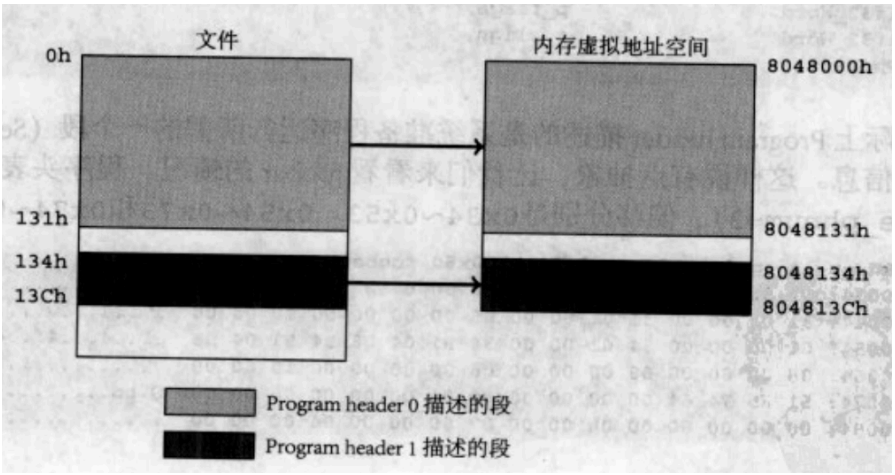
p_type	当前 program header 所描述的段的类型
p_offset	段的第一个字节在文件中的偏移
p_vaddr	段的第一个字节在内存中的虚拟地址
p_paddr	在物理地址定位相关的系统中，此项是为物理地址保留的
p_filesz	段在文件中的长度
p_memsz	段在内存中的长度
p_flags	与段相关的标志
p_align	根据此项值来确定段在文件以及内存中如何对齐

program header 描述的是一个段在文件中的位置、大小以及它被放进内存后所在的位置和大小。

举一个例子，假设一个程序的 object 文件中有 2 个 program header，取值如下：

名称	program header 0	program header 1
p_type	0x1	0x1
p_offset	0x0	0x134
p_vaddr	0x8048000	0x8049134
p_paddr	0x8048000	0x8049134
p_filesz	0x131	0x8
p_memsz	0x131	0x8
p_flags	0x5	0x6
p_align	0x1000	0x1000

那么这个程序在文件中的位置、大小以及它被放进内存后所在的位置和大小如下：



1.4 重新放置内核

由 ld 生成的可执行文件中 p_vaddr 的值总是一个类似于 0x8048XXX 的值。我们不能让编译器来决定内核加载到什么地方，有两种解决的方法：

- 一是通过修改页表让 0x8048XXX 映射到较低的地址。
- 另一种方法是通过修改 ld 的选项让它生成的可执行代码中 p_vaddr 的值变小。

第二种方法如下所示：

```
1  nasm -f elf -o kernel.o kernel.asm
2  ld -s -Ttext 0x30400 -o kernel.bin kernel.o
```

这样子就将程序的入口地址变为 0x30400，ELF header 的信息会位于 0x30400 之前。ELF header 的内容如下所示：

e_ident	...	
e_type	2H	可执行文件
e_machine	3H	80386
e_version	1H	
e_entry	30400H	入口地址
e_phoff	34H	program header table 在文件中的偏移量
e_shoff	448H	section header table 在文件中的偏移量
e_flags	0H	
e_ehsize	34H	ELF header 大小

e_phentsize	20H	每一个 program header 大小 20H 字节
e_phnum	1H	program header table 中只有一个条目
e_shentsize	28H	每一个 section header 大小 28H 字节
e_shnum	4H	section header table 有 4 个条目
e_shstndx	3H	包含节名称的字符串表是第 3 个节

实际上，我们需要根据 program header table 的信息对内核在内存中的位置进行整理。上面通过 ld 调整程序的入口地址，只是为了让可执行代码中的 p_vaddr 的值变小。在实际操作过程中，我们通过 program header 得知段在文件中的偏移 p_offset，得知段的大小 p_filesz，然后将它移动到段在内存的虚拟地址 p_vaddr。如下所示：

```
1    memcpy(p_vaddr, BaseOfLoaderPhyAddr + p_offset, p_filesz);
```

如果 program header 的内容如下所示：

p_type	1H	PT_LOAD
p_offset	0H	段的第一个字节在文件中的偏移
p_vaddr	30000H	段的第一个字节在内存中的虚拟地址
p_paddr	30000H	
p_filesz	40DH	段在文件中的长度
p_memsz	40DH	段在内存中的长度
p_flags	5H	
p_align	1000H	

这样的话，我门内核放置函数的语句如下：

```
1    memcpy(3000h, 9000h + 0, 40Dh);
```

memcpy 函数在第五次学习报告中有提到过，这里将它的代码实现复习一下：

```
1    ; 该函数使用堆栈读入输入参数，使用eax作为返回参数
2    ; 函数有三个参数，分别是dest、source和size
3    MemCpy:
4        push ebp
5        mov ebp, esp
6
7        push esi
8        push edi
9        push ecx
10       ; 目的地址单元
11       mov edi, [ebp + 8]
12       ; 源数据地址单元
13       mov esi, [ebp + 12]
```

```
        ; 源数据内容大小
        mov ecx, [ebp + 16]
    .1:
        cmp ecx, 0
        jz .2
        ; 将源数据的1字节读入al中
        mov al, [ds:esi]
        ; 让esi指向下一字节
        inc esi
        ; 将al的值读入目的地址单元中
        mov byte [es:edi], al
        ; 让esi指向下一字节
        inc edi

        dec ecx
        jmp .1
    .2:
        ; 让eax存放目的地址
        mov eax, [ebp + 8]

        pop ecx
        pop edi
        pop esi
        mov esp, ebp
        pop ebp
        ret
```

在这里我总结一下整理内核的步骤：

- 根据 ELF header 得到 e_phnum、e_phoff。利用这两个值就可以遍历 program header table 中所有的 program header。
- 根据 program header 得到 p_vaddr、p_offset 和 p_filesz。将这三个参数从后往前压栈，调用 MemCpy 函数。

这样一来，就能把内核的各个段安排到内存中合适的位置，代码如下：

```
        call InitKernel
        ; ...
    InitKernel:
        xor esi, esi
        ; 将e_phnum移入cx中
        mov cx, word [BaseOfKernelFilePhyAddr+2Ch]
        ; movzx指令将源操作数取出来,然后置于目的操作数,目的操作数其余位用0填充
        movzx ecx, cx
        ; 将program header table在文件中的偏移量放到esi中
        mov esi, [BaseOfKernelFilePhyAddr+1Ch]
        ; 让esi指向program header table
        add esi, BaseOfKernelFilePhyAddr
    .Begin:
        ; 将program header成员p_type的值移入eax
        mov eax, [esi + 0]
        ; 检查p_type是否是PT_NULL
        cmp eax, 0
```

```
18          ; 如果是，就避开这个program header
19          jz .NoAction
20          ; 将p_filesz的值压栈，也就是将段在文件中的长度压栈
21          ; ，作为函数所需的源数据大小
22          push dword [esi + 010h]
23          ; 将p_offset的值存放eax，也就是将段在文件中的偏移存入eax
24          mov eax, [esi + 04h]
25          ; 加上内核实际物理地址，得到段在文件中的实际物理地址
26          add eax, BaseOfKernelFilePhyAddr
27          ; 将段的实际物理地址压栈，作为函数所需的源数据地址
28          push eax
29          ; 将段在内存的虚拟地址压栈，作为函数所需的目的地址
30          push dword [esi + 08h]
31          ; 调用MemCpy函数
32          call MemCpy
33          ; 清除堆栈中的参数
34          add esp, 12
35      .NoAction:
36          ; 一个program header有32个字节
37          ; 让esi指向下一个program header
38          add esi, 020h
39          ; 将ecx的值减一
40          dec ecx
41          jnz .Begin
42
43          ret
```

1.5 进入内核

进入内核的代码很简单，相当于开始指向内核的代码，只要向内核跳转即可，代码如下：

```
1       ; 030400h是内核的程序入口
2       jmp SelectorFlatC:030400h
```

这样操作系统就真正地开始执行内核了。“引导，加载内核进入内存，跳入保护模式，开始执行内核”这四部分内容到此我们都完成了。

2 内核雏形

2.1 定义相关的数据结构和数据类型

为了让内核的代码更清楚，在本节将定义一些相关的数据结构和数据类型。

我定义了宏 PUBLIC 和 PRIVATE 来区分文件中的普通函数和静态函数，用于更好地区分 static 函数和可以被其他文件引用的函数，

```
#ifndef _CONST_H_
#define _CONST_H_

#define PUBLIC
#define PRIVATE static

#define GDT_SIZE 128

#endif
```

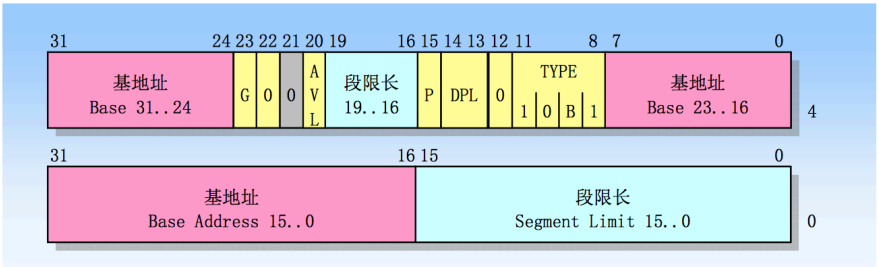
为了在处理段描述符或其他数据结构时，能清楚每个成员类型的长度，我在 type.h 中定义了 u8、u16 和 u32 等类型，分别代表 8 位、16 位和 32 位的数据类型。

```
#ifndef _TYPE_H
#define _TYPE_H

typedef unsigned int u32;
typedef unsigned short u16;
typedef unsigned char u8;

#endif
```

同时我用 C 语言定义了段描述符结构体。因为段描述符是第三次学习报告的内容，时间隔了比较久，所以在此再复习一下段描述符。先看一下它的格式：



对它的各个位解释如下：

0-15 位	段限长的第一部分，16 位
16-31 位	基地址的第一部分，16 位

32-39 位	基地址的第二部分，8 位
40-47 位	段描述符的属性的第一部分，8 位
48-55 位	8 位，低 4 位是段限长的第二部分，高 4 位是属性的第二部分
56-63 位	基地址的第三部分，8 位

```
#ifndef _PROTECT_H_
#define _PROTECT_H_

typedef struct s_descriptor
{
    u16 limit_low;
    u16 base_low;
    u8 base_mid;
    u8 attr1;
    u8 limit_high_attr2;
    u8 base_high;
}DESCRIPTOR;

#endif
```

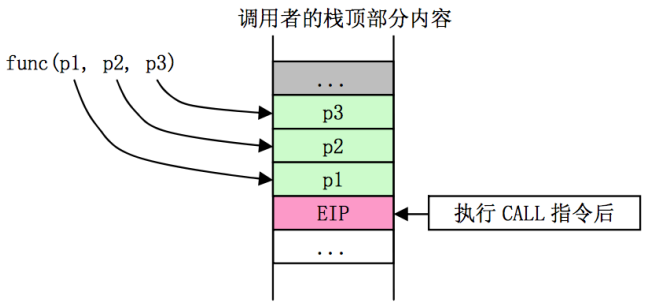
2.2 C 与汇编程序的相互调用

因为之后经常会涉及到 C 语言和汇编程序的相互调用，所以学习这个是很有必要的。

2.2.1 在汇编程序中调用 C 函数

在汇编程序调用一个 C 函数时，程序首先需要按照逆向顺序把函数参数压入栈中，即函数最右的一个参数最先压入栈，函数最左的一个参数最后压入栈，然后执行 call 指令。在调用函数返回后，程序需要将先前压入栈中的函数参数清除掉。

汇编中调用 C 函数的时候没有对参数的压栈有严格的约束。首先见下图这个例子：



如果我们没有专门为调用函数 func() 压入参数就直接调用它的话，那么 func() 函数仍然会把存放 EIP 位置以上的栈中其他内容作为自己的参数使用。

2.2.2 在 C 程序中调用汇编函数

在 C 程序中调用汇编函数时，类似于汇编程序调用 C 函数，程序也是将函数的最右参数最先压入栈，将函数的最左参数最后压入栈。

下面是一个例子：

```
# 导出_myadd函数
.global _myadd

.text
# int myadd(int a, int b, int* res);
# eax存放着返回值
_myadd:
    pushl %ebp
    movl %esp, %ebp
    # 4(%ebp)地址单元存放着EIP
    # 8(%ebp)地址单元存放着第一个参数
    # 12(%ebp)地址单元存放着第二个参数
    # 16(%ebp)地址单元存放着第三个参数
    movl 8(%ebp), %eax
    movl 12(%ebp), %edx
    xorl %ecx, %ecx
    addl %eax, %edx
    jo 1f
    movl 16(%ebp), %eax
    movl %edx, (%eax)
    incl %ecx
1:
    movl %ecx, %eax
    movl %ebp, %esp
    popl %ebp
    ret
```

```
int myadd(int a, int b, int* res);

int main()
{
    int a = 5;
    int b = 10;
    int c;
    myadd(a, b, &c);
    return 0;
}
```

2.3 移动 GDT 到内核

之前我们在 Loader 模块内定义了 GDT 表，随后就开始执行内核。为了继续利用 Loader 中 GDT 表的信息，应该将 GDT 表复制到内核中。

目前我们 gdt 寄存器中存放着 Loader 模块中 GDT 表的基地址以及 GDT 表的长度。而我们在内核中重新定义一个 GDT 表，也是有一个地址，于是就可以使用 memcpy 函数将 Loader 中的 GDT 表的信息复制到内核中的 GDT 表中。随后，我们应该更新 gdt 寄存器中的信息。

2.3.1 memcpy 函数

在学习 Loader 模块或分页机制的时候，我们都有接触过 MemCpy 函数。这里在 linux 汇编下再来感受一下，并且体验如何在 C 程序中调用这个函数。代码如下：

```
1   [SECTION .text]
2   global memcpy
3   ; void* memcpy(void* es:dest, void* ds:src, int iSize);
4   ; eax存放着返回值
5   memcpy:
6       push ebp
7       ; 让ebp指向栈顶
8       mov ebp, esp
9       push esi
10      push edi
11      push ecx
12
13      ; ebp+4存放着EIP
14      ; ebp+8存放着dest
15      ; ebp+12存放着src
16      ; ebp+16存放着iSize
17      mov edi, [ebp + 8]
18      mov esi, [ebp + 12]
19      mov ecx, [ebp + 16]
20  .1:
21      cmp ecx, 0
22      jz .2
23
24      mov al, [ds:esi]
25      inc esi
26
27      mov byte [es:edi], al
28      inc edi
29
30      dec ecx
31      jmp .1
32  .2:
33      ; 让eax存放着目的地址，作为返回值
34      mov eax, [ebp + 8]
35
36      pop ecx
37      pop edi
```

```
38          pop esi
39          mov esp, ebp
40          pop ebp
41
42          ret
```

在 C 程序中调用 memcpy 函数如下所示：

```
1   void* memcpy(void* dest, void* src, int iSize);
2
3   int main()
4   {
5       int dest[10];
6       int src[10] = {1,2,3,4,5,6,7,8,9,10};
7       memcpy(dest, src, 10);
8       return 0;
9   }
```

2.3.2 移动 GDT

有了前面的铺垫和思路的介绍，将 Loader 模块中 GDT 的内容复制到内核中新的 GDT 表中的代码就清晰了很多。代码如下：

```
1   SELECTOR_KERNEL_CS equ 8
2   ; 引入函数
3   extern cstart
4   ; 引入全局变量
5   extern gdt_ptr
6
7   ; bss段用于放置未初始化的变量
8   [section .bss]
9   LABEL_STACK:
10  StackSpace resb 2*1024
11  StackTop:
12
13  ; text段用于放置代码
14  [section .text]
15
16  ; 导出_start
17  global _start
18  _start:
19      ; 让esp指向StackTop
20      mov esp, StackTop
21      ; 将全局描述符表寄存器的内容复制给gdt_ptr地址单元
22      sgdt [gdt_ptr]
23      ; 调用cstart函数
24      call cstart
25      ; 将gdt_ptr地址单元中的内容加载到全局描述符寄存器中
26      lgdt [gdt_ptr]
27
28      jmp SELECTOR_KERNEL_CS:csinit
29
30  csinit:
```

```
        push 0
        popfd
        hlt
```

```
#include"type.h"
#include"const.h"
#include"protect.h"

PUBLIC void* memcpy(void* pDst, void* pSrc, int iSize);
// gdt_ptr是48位，所以这里数组大小为6
PUBLIC u8 gdt_ptr[6];
// 在内核中新的GDT表可以存放128个段描述符
PUBLIC DESCRIPTOR gdt[GDT_SIZE];

PUBLIC void cstart()
{
    memcpy((void*)gdt, (void*)(*((u32*)(&gdt_ptr[2]))), *((u16*)(&gdt_ptr[0]))
        +1);
    // 指向gdt_ptr的低16位的地址
    u16* p_gdt_limit = (u16*)(&gdt_ptr[0]);
    // 指向gdt_ptr的高32位的地址
    u32* p_gdt_base = (u32*)(&gdt_ptr[2]);
    // 更新gdt_ptr中的GDT限长
    *p_gdt_limit = GDT_SIZE * sizeof(DESCRIPTOR) - 1;
    // 更新gdt_ptr中GDT表的基地址
    *p_gdt_base = (u32)gdt;
}
```

2.4 显示字符串的函数 disp_str()

```
[SECTION .data]
disp_pos dd 0

[SECTION .text]

; 导出disp_str()函数
global disp_str

; void disp_str(char* info);
disp_str:
    push ebp
    mov ebp, esp

    ; 让esi存放字符串的地址
    mov esi, [ebp + 8]
    mov edi, [disp_pos]
    mov ah, 0Fh
.1:
    ; 将ds:esi指向的地址的一个字节读入al
    lodsb
    ; 判断ds:edi是否指向字符串的尽头
    test al, al
```

```
        ; 如果是，就结束程序
        jz .2
        ; '\n'的ascii码是0Ah
        ; 判断是否读入回车键
        cmp al, 0Ah
        ; 如果不是，就跳到.3直接输出当前字符串
        jnz .3
        ; 如果是，就显示出回车键的效果
        ; 将eax的值压栈，保存eax的原值
        push eax
        ; edi指向要显示的下一个字符的位置，低8位为行号，8~15位为列号
        ; 修改edi的值，先让eax保存edi的值
        mov eax, edi
        ; 每行有160个字节
        mov bl, 160
        ; 得到当前的行号，存放在al中
        div bl
        ; 将列号清零
        and eax, 0FFh
        ; eax中的值加一，行号加一
        inc eax
        mov bl, 160
        ; 让eax指向下一行的头一个字节
        mul bl
        ; 修改edi的值结束，将eax中的值赋给edi
        mov edi, eax
        ; 恢复eax的原值
        pop eax
        jmp .1
    .3:
        mov [gs:edi], ax
        add edi, 2
        jmp .1
    .2:
        mov [disp_pos], edi

        pop ebp
        ret
```

C 程序调用 disp_str 函数的例子如下：

```
PUBLIC void disp_str(char* info);

int main()
{
    disp_str("hello world\n");
    return 0;
}
```

3 kvm 环境的搭建

首先声明，这个仅在 ubuntu16.04 下配置过，配置日期为 2016.12.16。

3.1 搭建硬件环境

在 x86_64 架构的 INTEL 处理器中，KVM 必需的硬件虚拟化扩展为 INTEL 的虚拟化技术 (INTEL VT)。首先处理器要在硬件上支持 VT 技术。只有在 BIOS 中将 VT 打开，才可以使用 KVM。

3.1.1 检查处理器是否支持 VT 技术

在 linux 系统中，可以通过/proc/cpuinfo 文件中的 CPU 特性标志来查看 CPU 是否支持 VT 技术。如果 CPU 支持 VT 技术，那么文件中的 flags 就包含”vmx”。

这里使用 grep 命令来查看/proc/cpuinfo 中是否包含”vmx”。首先介绍一下 grep 命令。

```
grep全称是Global Regular Expression Print，使用正则表达式搜索文本，并把匹配的行打印
    出来

grep [pattern] [file] 用于在file中查找符合pattern的文本，并打印出来

更多的grep信息，可以通过man grep查看。
```

使用如下命令就可以查看/proc/cpuinfo 中是否包含了”vmx”:

```
1    grep "vmx" /proc/cpuinfo
```

如果在/proc/cpuinfo 中包含了”vmx”，说明 CPU 支持 VT 技术，否则另外设置。

3.1.2 设置 BIOS

如果 CPU 目前不支持 VT 技术，就需要设置 BIOS 中相应的选项。

VT 的选项一般在 BIOS“Advanced”栏目的”CPU Configuration” 中，它由”Intel Virtualization Technology” 或”Intel VT” 标识。找到该标识后，将其设为 [Enabled] 就可以了。

3.2 安装 KVM

3.2.1 下载 KVM 源代码

使用如下命令即可下载 KVM 源代码:

```
1    git clone https://git.kernel.org/pub/scm/virt/kvm/kvm.git
```

3.2.2 配置 KVM

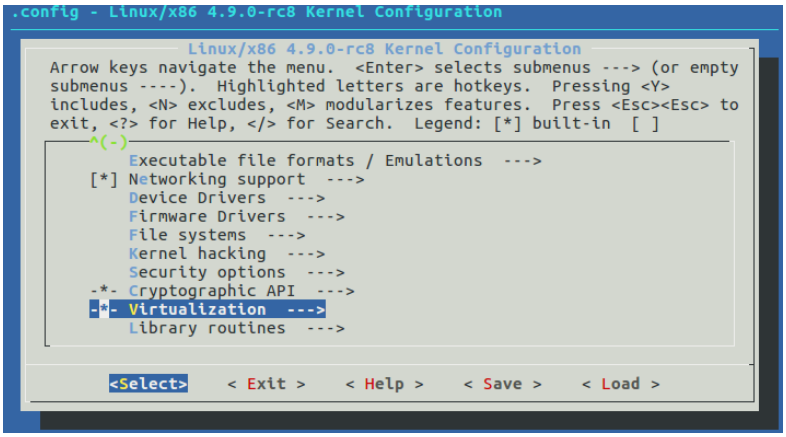
在此我们使用 make menuconfig 对 KVM 进行配置。首先安装 ncurses 库：

```
sudo apt-get install libncurses5-dev
```

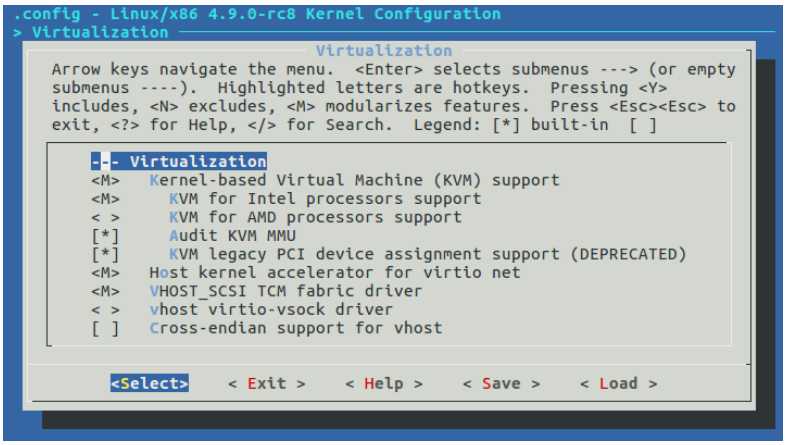
然后在 KVM 文件中打开 terminal，然后输入如下命令：

```
make menuconfig
```

然后选择”Virtualization”，如下所示：



进入以后，进行如下图的配置：



在配置完成后，将在 KVM 文件夹下生成.config 文件。如果想确保 KVM 相关的配置正确，可以检查.config 文件的相关配置项。查看命令如下：

```
vi .config
/CONFIG_HAVE_KVM
```

光标跳到对应的配置项后，查看是否如下面几个配置：

```
CONFIG_HAVE_KVM=y
CONFIG_HAVE_KVM_IROCHIP=y
CONFIG_HAVE_KVM_IRQFD=y
CONFIG_HAVE_KVM_IRQ_ROUTING=y
CONFIG_HAVE_KVM_ENENTFD=y
CONFIG_KVM_MMIO=y
CONFIG_KVM_ASYNC_PF=y
CONFIG_HAVE_KVM_MSI=y
CONFIG_HAVE_KVM_CPU_RELAX_INTERCEPT=y
CONFIG_KVM_VFIO=y
CONFIG_KVM_GENERIC_DIRTYLOG_READ_PROTECT=y
CONFIG_KVM_COMPAT=y
CONFIG_HAVE_KVM_IRQ_BYPASS=y
CONFIG_VIRTUALIZATION=y
CONFIG_KVM=m
CONFIG_KVM_INTEL=m
```

3.2.3 编译 KVM

KVM 的编译包括三个步骤：

- Kernel 的编译。
- bzImage 的编译。
- 内核模块的编译。

第一步是编译 kernel。因为 kernel 包含了 openssl 的库，所以在编译之前，需要先安装 openssl 和相关的库，命令如下：

```
sudo apt-get install openssl
sudo apt-get install libssl-dev
```

接下来，就可以直接开始编译 kernel，命令如下：

```
make vmlinux
```

第二步是编译 bzImage。命令如下：

```
make bzImage
```

第三步是编译内核的模块。命令如下：

```
make modules
```

3.2.4 安装 KVM

KVM 的安装包括两个步骤：

- module 的安装。
- kernel 与 initramfs 的安装。

首先安装 module，命令如下：

```
sudo make modules_install
```

然后安装 kernel 和 initramfs，命令如下：

```
sudo make install
```

最后重启系统，命令如下：

```
reboot
```

3.2.5 加载 kvm 和 kvm_intel 模块

使用如下命令即可加载 kvm 和 kvm_intel 模块：

```
modprobe kvm
modprobe kvm_intel
```

执行 lsmod 指令，会列出所有已载入系统的模块。然后使用 grep，可以查看是否存在 kvm 和 kvm_intel 模块，命令如下所示：

```
lsmod | grep kvm
```

如果 terminal 输出有 kvm 和 kvm_intel 有关的信息，就说明 kvm 和 kvm_intel 模块加载成功了。

3.3 安装 qemu-kvm

3.3.1 下载 qemu-kvm 源代码

使用如下命令即可下载 qemu-kvm 源代码：

```
git clone https://git.kernel.org/pub/scm/virt/kvm/qemu-kvm.git
```

3.3.2 配置 qemu-kvm

使用如下命令即可配置 qemu-kvm:

```
./configure
```

这里介绍一下./configure 命令:

```
./configure会根据当前系统环境和指定参数生成makefile文件，为下一步的编译做准备。
可以通过在configure后加上参数来对安装进行控制。
比如: ./configure --prefix=/usr
意思是将该软件安装在/usr下面，执行文件就会安装在/usr/bin
```

随后，为了不让编译器把警告当作错误处理，应该在 Makefile 中任意一行添加如下代码:

```
QEMU_CFLAGS += -w
```

3.3.3 编译 qemu-kvm

使用如下命令即可编译 qemu-kvm:

```
make
```

3.3.4 安装 qemu-kvm

使用如下命令即可安装 qemu-kvm:

```
sudo make install | tee make-install.log
```

这里介绍一下 tee 命令:

```
功能说明: 读取标准输入的数据，并将其内容输出成文件。
语法: tee [-ai][--help][--version][文件]
参数:
    -a 附加到既有文件的后面，而非覆盖它。
    -i 忽略中断信号。
    --help 在线帮助。
    --version 显示版本信息。
```

3.4 安装客户机

3.4.1 创建镜像文件

安装客户机之前，我们需要创建一个镜像文件来存储客户机中的系统和文件。镜像文件将作为客户机的硬盘，将客户机的操作系统安装在其中。

首先，使用如下命令行创建一个 8GB 大小的镜像文件 ubuntu1604.img：

```
dd if=/dev/zero of=ubuntu1604.img bs=1M count=8192
```

这里介绍一下 dd 命令：

```
功能说明：把指定的输入文件拷贝到指定的输出文件中。
语法：dd [选项]
参数：
    if=输入文件
    of=输出文件
    bs=bytes 同时设置读/写缓冲区的字节数
    count=blocks 只拷贝输入的blocks块
```

3.4.2 安装客户机

在联网的情况下，使用如下命令安装客户机：

```
qemu-system-x86_64 -m 2048 -smp 4 -boot order=cd -hda ubuntu1604.img -cdrom
    ubuntu-16.04.iso  -vnc 127.0.0.1:2
```

这里介绍一下 qemu-system-x86_64 的参数，如下所示：

```
-m 2048是给客户机分配2048MB内存
-smp 4是给客户机分配4个CPU
-boot order=cd是指定系统的启动顺序为光驱、硬盘
-hda ubuntu1604.img是分配给客户机的IDE硬盘
-cdrom ubuntu-16.04.iso是分配给客户机的光驱
 -vnc 127.0.0.1:2使用vnc方式显示客户机，端口为127.0.0.1:2
```

3.4.3 查看客户机

因为这里使用 vnc 方式显示客户机，所以我们需要先安装 vncserver 和 vncviewer，命令如下：

```
sudo apt-get install vncserver
sudo apt-get install vncviewer
```

然后使用 vncviewer 查看客户机，命令如下：

```
vncviewer 127.0.0.1:2
```

3.5 启动 KVM 客户机

安装好系统之后，就可以使用镜像文件来启动并登陆到自己安装的系统之中。在联网的情况下，使用如下命令即可启动一个 KVM 的客户机：

```
qemu-system-x86_64 -m 2048 -smp 4 -hda ubuntu1604.img -vnc 127.0.0.1:2
```

使用如下命令可以查看 KVM 客户机:

```
vncviewer 127.0.0.1:2
```

4 CPU 配置

4.1 -smp 参数项

qemu-system-x86_64 命令行中，”-smp” 参数可以用来配置客户机的 SMP 系统，具体参数如下：

```
qemu-system-x86_64 -smp n[,maxcpus=cpus][,cores=cores][,threads=threads][,
    sockets=sockets]
```

各个选项介绍如下：

n	用于设置客户机中使用的逻辑 PCU 数量
maxcpus	用于设置客户机中最大可能被使用的 CPU 数量
cores	用于设置每个 CPU socket 上的 core 数量
threads	用于设置每个 CPU core 上的线程数
sockets	用于设置客户机中看到的总的 CPU socket 数量

例子如下：

```
1 qemu-system-x86_64 -smp 4,maxcpus=8,sockets=2,cores=2,threads=2 ubuntu1604 -vnc
      127.0.0.1:2
```

4.2 查看 cpu 配置

4.2.1 在客户机中查看 cpu 信息

使用如下命令可以输出 cpu 当前的信息：

```
1 cat /proc/cpuinfo
```

这里介绍一下 cat 命令：

三大功能	1. 一次显示整个文件: cat filename 2. 从键盘创建一个文件: cat > filename 3. 将几个文件合并为一个文件: cat file1 file2 > file
参数	-n 或–number: 由 1 开始对所有输出的行数编号 -b 或–number-nonblank: 和-n 相似，只不过对于空白行不编号 -s 或–squeeze-blank: 当遇到有连续两行以上的空白行，就代换为一行的空白行

4.2.2 使用 qemu 监控客户机 cpu 信息

使用 qemu-system-x86_64 命令时，加上”-monitor stdio”，即可使用 monitor command 监控客户机使用情况，比如在联网情况下输入如下命令：

```
qemu-system-x86_64 ubuntu1604.img -vnc 127.0.0.1:2 -monitor stdio
```

此时，就开始 monitor command 来监控客户机。可以在 qemu monitor 中使用如下命令查询 cpu 状态：

```
info cpus
```

4.3 -cpu 参数项

qemu-system-x86_64 命令行中，”-cpu” 参数可以用来查看 qemu 所支持 cpu 模型，或者指定客户机的 CPU 模型。具体使用如下：

```
// 查看qemu所支持的cpu模型
qemu-system-x86_64 -cpu ?
// 指定客户机中的cpu模型
qemu-system-x86_64 -cpu cpu_model
```

qemu 支持的 cpu 模型如下所示：

```
pengsida@psd:~$ qemu-system-x86_64 -cpu ?

(process:3543): GLib-WARNING **: /build/glib2.0-7IO_Yw/glib2.0-2.48.1/./glib
/gmem.c:483: custom memory allocation vtable not supported
x86       Opteron_G4  AMD Opteron 62xx class CPU
x86       Opteron_G3  AMD Opteron 23xx (Gen 3 Class Opteron)
x86       Opteron_G2  AMD Opteron 22xx (Gen 2 Class Opteron)
x86       Opteron_G1  AMD Opteron 240 (Gen 1 Class Opteron)
x86      SandyBridge  Intel Xeon E312xx (Sandy Bridge)
x86          Westmere  Westmere E56xx/L56xx/X56xx (Nehalem-C)
x86           Nehalem  Intel Core i7 9xx (Nehalem Class Core i7)
x86            Penryn  Intel Core 2 Duo P9xxx (Penryn Class Core 2)
x86            Conroe  Intel Celeron_4x0 (Conroe/Merom Class Core 2)
x86              n270  Intel(R) Atom(TM) CPU N270   @ 1.60GHz
x86            athlon  QEMU Virtual CPU version 1.2.50
x86          pentium3
x86          pentium2
x86           pentium
x86               486
x86           coreduo  Genuine Intel(R) CPU           T2600  @ 2.16GHz
x86             kvm32  Common 32-bit KVM processor
x86            qemu32  QEMU Virtual CPU version 1.2.50
x86             kvm64  Common KVM processor
x86          core2duo  Intel(R) Core(TM)2 Duo CPU     T7700  @ 2.40GHz
x86            phenom  AMD Phenom(tm) 9550 Quad-Core Processor
x86            qemu64  QEMU Virtual CPU version 1.2.50
```

如果不加”cpu” 参数启动客户机时，采用”qemu64” 作为默认的 cpu 模型。

4.4 vCPU 的绑定

vCPU 就是客户机的虚拟 cpu，vCPU 相当于宿主机中一个普通的 qemu 线程。可以使用 taskset 工具将 vCPU 线程绑定到特定的 cpu 上执行。

在实际应用中，如果想要为客户提供客户机使用，并且要求不受宿主机中其他客户机的影响，就需要将 vCPU 绑定到特定的 cpu 上。步骤如下：

1. 启动宿主机时隔离出特定的 CPU 专门供一个客户机使用。
2. 启动客户机，将其 vCPU 绑定到宿主机特定的 CPU 上。

4.4.1 隔离宿主机 CPU

在 grub 文件中 Linux 内核启动的命令行加上”isolcpus” 参数，就可以实现 CPU 的隔离。这里介绍一下”isolcpus” 参数项：

功能	将相应的 CPU 从调度算法中隔离出来
参数选项	isolcpus= cpu_number[,cpu_number,...]

向 grub 文件中添加”isolcpus” 参数的命令如下所示：

```
1  sudo vi /boot/grub/grub.cfg
2  /menuentry
```

然后在插入模式下，在 initrd 参数前一行写入：

```
1  isolcpus = cpu\_number1[,cpu\_number2,...]
```

如下图所示：

```
menuentry 'Ubuntu' --class ubuntu --class gnu-linux --class gnu --class os $menu
entry_id_option 'gnulinux-simple-84ca4c1b-ff58-4c6d-8808-45e3dc5f4e63' {
        recordfail
        load_video
        gfxmode $linux_gfx_mode
        insmod gzio
        if [ x$grub_platform = xxen ]; then insmod xzio; insmod lzopio; fi
        insmod part_gpt
        insmod ext2
        set root='hd0,gpt2'
        if [ x$feature_platform_search_hint = xy ]; then
          search --no-floppy --fs-uuid --set=root --hint-bios=hd0,gpt2 --hint-ef
i=hd0,gpt2 --hint-baremetal=ahci0,gpt2  84ca4c1b-ff58-4c6d-8808-45e3dc5f4e63
        else
          search --no-floppy --fs-uuid --set=root 84ca4c1b-ff58-4c6d-8808-45e3dc
5f4e63
        fi
        linux   /boot/vmlinuz-4.9.0-rc8+ root=UUID=84ca4c1b-ff58-4c6d-8808-45e3d
c5f4e63 ro  quiet splash $vt_handoff
        isolcpus = 2,3
        initrd  /boot/initrd.img-4.9.0-rc8+
}
-- 插入 --                                                    151,2         39%
```

重启电脑以后就将相应的 cpu 隔离出调度算法了。

使用如下命令可以查看 cpu 上执行的进程和线程总数，用于检查 CPU 是否成功被隔离。

```
ps -eLo psr | grep cpu\_number | wc -l
```

下面分别介绍命令中的 ps 和 wc：

ps	用于显示当前系统的进程信息的状态
参数项	-e: 用于显示所有进程 -L: 用于显示所有线程 -o: 用于以特定的格式输出信息,psr 指定输出分配给进程运行的处理器编号

wc	该命令统计给定文件中的字节数、字数、行数
参数项	-c: 统计字节数 -l: 统计行数 -w: 统计字数

假如成功隔离了 cpu2，就会看到在 cpu 上执行的进程和线程数非常少。

4.4.2 绑定客户机 vCPU

使用 taskset 命令就可以将 vCPU 绑定到特定的 CPU 上。taskset 命令的使用如下所示：

```
taskset -p mask pid
```

这里介绍一下 taskset 命令：

taskset	将进程绑定到特定的 CPU 上
参数项	-p: 将已经创建的进程绑定到 CPU 上 mask: 用于指定 CPU 的掩码，mask 第几位为 1 就代表第几号 CPU pid: 进程号，用于指定进程

比如，如果想把进程号为 3963 的进程绑定到 cpu2 和 cpu3 上，就使用如下命令：

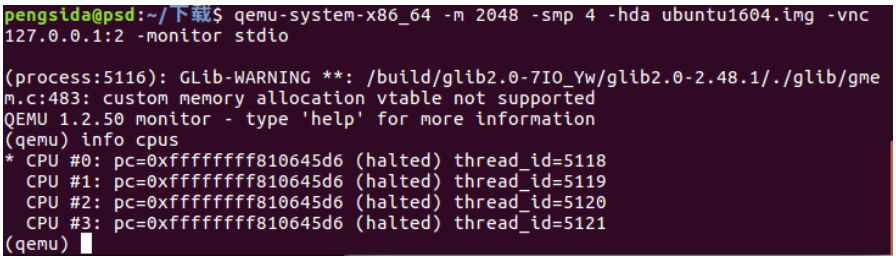
```
taskset -p 0x6 3963
```

0x6 二进制位 1100，代表 cpu2 和 cpu3。而 3963 指定了进程号为 3963 的进程。

如此一来，如果想把 vCPU 绑定到宿主机的 cpu 上，只要知道 vCPU 的进程号就行了。可以在 qemu monitor 中使用如下命令查询 vCPU 的进程号：

```
1    info cpus
```

如下所示：



可以看到，图中 vCPU 的进程号分别是 5118、5119、5120 和 5121。

5 内存配置

5.1 -m 参数项

-m megs	设置客户机的内存位 megsMB 大小 默认单位为 MB，加上”M” 或”G” 可以指定单位 不设置-m 参数，客户机内存默认为 128MB

5.2 查看内存信息

linux 下有两个命令可以用于查看内存信息。

第一个是 free -m，如下图所示：

```
psd@scholes:~$ free -m
              total        used        free      shared  buff/cache   available
Mem:           2000         560         771           7         667        1264
Swap:          2045           0        2045
```

第二个是 dmesg。不过因为 dmesg 存放着内核开机信息，信息量比较多，需要用 grep 命令来筛选，如下图所示：

```
psd@scholes:~$ dmesg | grep Memory
[    0.000000] Memory: 2009076K/2096752K available (8427K kernel code, 1285K rwd
ata, 3956K rodata, 1480K init, 1292K bss, 87676K reserved, 0K cma-reserved)
```

5.3 EPT 扩展页表

EPT 扩展页表是 Intel 的第二代硬件虚拟化技术，是针对内存管理单元的虚拟化扩展。

在 Linux 系统中，可以通过如下命令确定系统是否支持 EPT 功能：

```
grep ept /proc/cpuinfo
```

可以通过如下命令确定 KVM 是否打开了 EPT 功能：

```
cat /sys/module/kvm_intel/parameters/ept
```

在加载 kvm_intel 模块时，可以通过设置 ept 的值来打开 EPT。

```
modprobe kvm_intel ept=0 // ept代表关闭EPT功能
```

如果 kvm_intel 模块已经处于加载状态，则需要先写在这个模块，在重新加载时加入所需的参数设置。如下所示：

```
rmmod kvm_intel
modprobe kvm_intel ept=1
```

5.4 -mem-path 参数项

qemu-kvm 提供了”-mem-path” 参数项用于将 huge page 的特性应用到客户机上。

huge page 是大小超过 4KB 的内存页面，它可以让地址转换信息减少，节约页表所占用的内存数量，在整体上提升系统的性能。

可以使用如下命令查看系统中 huge page 的信息，如下所示：

```
cat /proc/meminfo | grep HugePages
```

可以通过以下几步让客户机使用 huge page：

(1) 在宿主机中挂载 hugetlbfs 文件系统，命令如下所示：

```
sudo mount -t hugetlbfs hugetlbfs /dev/hugepages
```

这里介绍一下 mount 命令：

标准格式	mount -t type device dir
功能	让内核将在 device 上的文件系统挂载到目录 dir 下，文件系统类型是 type
参数项	-t: 指定文件系统的类型

所以之前命令就是将 hugetlbfs 类型的文件系统挂载到/dev/hugepages 上。

(2) 设置 hugepage 的数量，命令如下所示：

```
sudo sysctl vm.nr_hugepages=num
```

(3) 启动客户机时使用”-mem-path” 参数让客户机使用 hugepage 的内存，如下所示：

```
qemu-system-x86_64 ubuntu1604.img -mem-path /dev/hugepages
```

上述过程的实际操作如下图所示：