

# 第一次操作系统实验报告

---

## 作者

191850036 冯亚林 软件学院

## 目录

### 第一次操作系统实验报告

作者

目录

#### 1.实验目标

- 1.1 安装Linux虚拟机
  - I.南大开源镜像站网址:
  - II.amd64:
- 1.2 安装 GNU Make 和 GCC
  - I.apl和apt-get有什么区别?
  - II.GNU是什么
  - III.GCC是什么
  - IV.GNU Make是什么
- 1.3 解决虚拟机和宿主机间的文件传输问题
- 1.4 安装 NASM
- 1.5 安装 虚拟机bochs 和 bochs-sdl库
  - 1.5.1 安装**
  - 1.5.2 源码安装的配置问题详解博客
  - 1.5.3 bochs安装特点
- 1.6 课程实验教学目标
- 1.7 Hello, OS
- 1.8 大数运算要求

#### 2.大数运算实现笔记

- 2.0 IDE——SASM
- 2.1 分析
- 2.2 输入与输出: 系统调用 或 调用C函数 printf
  - 2.2.1 系统调用实现read
  - 2.2.2 系统调用实现print
  - 2.2.3 调用C函数**
- 2.3 64位和32位的不同点
  - 2.3.1 系统调用号
  - 2.3.2 寄存器/寄存器个数
  - 2.3.3 64位系统下没有pusha/popa
  - 2.3.4 指针/解引用
- 2.4 64位系统下的寄存器约定
  - 2.4.1 传参
- 2.5 主文件/主函数
- 2.6 输入处理 Locate
- 2.7 加减法实现 MyAdd
- 2.8 乘法实现 MyTimes
- 2.9 测试结果

## 1.实验目标

---

## 1.1 安装Linux虚拟机

我在南大开源镜像站里下载了ubuntu镜像：ubuntu-18.04.6-desktop-amd64.iso文件，然后在VMware里创建了64位的、ubuntu18.04操作系统的虚拟机；其中虚拟机配置如下：



附：

I.南大开源镜像站网址：

<http://mirror.nju.edu.cn/>

II.amd64：

**x86-64**是“64位的x86指令集”的简写。该指令集由AMD设计，1999年由AMD发布，2000年发布了完整标准，后来由AMD、Intel、VIA分别实现。这个AMD版本的64位指令集**向前兼容**32位的x86指令集。苹果、RPM包管理、Arch Linux称之为“**x86-64**”或“**x86\_64**”，甲骨文和微软称之为“**x64**”，BSD和其他Linux发行版称之为“**amd64**”。

Intel在AMD 64发布之前就发布了IA-64架构，用于Itanium处理器家族，**不兼容x86指令集**，后来甲骨文、Redhat、微软都宣布停止支持IA-64。（Intel先搞出来了64位的指令集架构，但是不兼容32位的指令集系统，所以市场反响非常差劲）

Intel后来推出了Intel 64，与AMD64基本相同但有细节上的区别。虽然跟AMD64基本就一样，但是Intel一直在营销上把AMD贬为自家产品的仿制品，不承认使用了AMD的技术，于是Intel给AMD64另起了个名字叫“EM64T”，并且在文档里对其指令集的起源只字不提。媒体讽刺此事，给EM64T起了个外号叫“iAMD64”。再后来Intel就干脆把EM64T改名叫Intel 64了。

所以是AMD 64成为了x86-64的实际标准，怎么说都行。

作者：中型世界主义路灯

链接：<https://www.zhihu.com/question/28194051/answer/1714710341>

来源：知乎

## 1.2 安装 GNU Make 和 GCC

make的安装非常简单

终端输入：

```
sudo apt install make
```

但何不用更好的方式？

以下方式可以同时安装gcc, g++, make等;

```
sudo apt update 检查更新
```

```
sudo apt install build-essential
```

```
sudo apt install manpages-dev
```

```
gcc --version 检验gcc是否安装成功
```

### I.apt和apt-get有什么区别？

通过 apt 命令，用户可以在同一地方集中得到所有必要的工具，apt 的主要目的是提供一种以「让终端用户满意」的方式来处理 Linux 软件包的有效方式。

apt 具有更精简但足够的命令选项，而且参数选项的组织方式更为有效。除此之外，它默认启用的几个特性对最终用户也非常有帮助。例如，可以在使用 apt 命令安装或删除程序时看到进度条。

不仅广大 Linux 发行商都在推荐 apt，它还提供了 Linux 包管理的必要选项。

最重要的是，apt 命令选项更少更易记，因此也更易用，所以没理由继续坚持 apt-get。

**总结：apt比apt-get更好，包管理更简单，而且软件发行商更推荐。**

### II.GNU是什么

Unix 系统被发明之后，大家用的很爽。但是后来开始收费和商业闭源了。一个叫 Richard Stallman 的大叔觉得很不爽，于是发起 GNU 计划，模仿 Unix 的界面和使用方式，从头做一个开源的版本。然后他自己做了编辑器 Emacs 和编译器 GCC。

GNU 是一个计划或者叫运动。在这个旗帜下成立了 FSF，起草了 GPL 协议等。

接下来大家纷纷在 GNU 计划下做了很多的工作和项目，基本实现了当初的计划。包括核心的 gcc 和 glibc。但是 GNU 系统缺少操作系统内核。原定的内核叫 HURD，一直完不成。同时 BSD（一种 UNIX 发行版）陷入版权纠纷，x86 平台开发暂停。然后一个叫 Linus 的同学为了在 PC 上运行 Unix，在 Minix 的启发下，开发了 Linux。注意，Linux 只是一个系统内核，系统启动之后使用的仍然是 gcc 和 bash 等软件。Linus 在发布 Linux 的时候选择了 GPL，因此符合 GNU 的宗旨。

最后，大家突然发现，这玩意不正好是 GNU 计划缺的么。于是合在一起打包发布叫 GNU / Linux。然后大家念着念着省掉了前面部分，变成了 Linux 系统。实际上 Debian，RedHat 等 Linux 发行版中内核只占了很小一部分容量。

作者：匿名用户

链接：<https://www.zhihu.com/question/319783573/answer/656033035>

来源：知乎

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

### III.GCC是什么

gcc全称是GNU Compiler Collection，它是一个能编译多种语言的编译器。最开始gcc是作为C语言的编译器（GNU C Compiler），现在除了C语言，还支持C++、java、Pascal等语言。gcc支持多种硬件平台。gcc是GNU计划的核心套件，采取GPL协议。

### IV.GNU Make是什么

GNU make是一个项目包的高效管理和构建工具，可以方便地编译、链接多个源代码文件，自动决定哪些源文件需要重新编译（即所谓的高效构建项目）；它也可以用来卸载项目包；

官网：<https://www.gnu.org/software/make/>

## 1.3 解决虚拟机和宿主机间的文件传输问题

文件传输可以通过共享文件夹、FTP等方式实现；

但VMware上可以拖动文件进行文件传输（vmware tools里的功能），所以.....真香！

既然在本机上可以编辑代码，那就可以不用装Emacs这种编辑器了，嘿嘿。=>事实证明，我想多了

最后我选择了在虚拟机里安装轻量级的编辑器SASM；

网址：<http://dman95.github.io/SASM/english.html>

别的不说，光是一个色彩鲜艳又好看的界面就足以给人写下去的动力。

## 1.4 安装 NASM

笔记：

**Netwide Assembler**（简称 **NASM**）是一款基于Intel x86架构的汇编与反汇编软件。它可以用来编写16位（8086、80286等）、32位（IA-32）和64位（x86\_64）的程序。

它使用简化BSD协议，开源。

### 方法一:命令行

**sudo apt install nasm**

终端输入命令：`nasm -version`.输出版本信息就说明安装成功

**nasm --version**

### 方法二：源码压缩包（比较麻烦）

官网下载.tar.gz源码压缩包，传入虚拟机中解压，然后在解压文件夹下官方提供的 INSTALL 文件指导下安装;

具体来说就是：

a.提前装make和一个C编译器;

b.进入文件目录，执行以下文件：

```
sh configure
```

c.在终端的安装文件夹下输入

```
make 或者 make everything （包含make+文档）
```

d.最后输入

```
sudo make install/install_rdf/install everything
```

e.输入

```
nasm --version 检验安装是否成功
```

## 1.5 安装 虚拟机bochs 和 bochs-sdl库

在Linux虚拟机上继续安装ISA级虚拟机bochs;（虚拟机上安装虚拟机，套娃了属于是）

Bochs是x86硬件平台的开源模拟器。

它的特点是可以透过本机的指令集模拟目标指令集，是一种ISA级的虚拟机。

虚拟机级别：ISA级虚拟机（通过模拟指令集实现一定的虚拟化功能）、硬件级虚拟机(需要一个Hypervisor、VMM，比如VMware)、操作系统级虚拟机（操作系统中添加一个虚拟化层）、运行时库(wine)、应用程序级（JVM）

### 1.5.1 安装

#### 方法一：命令行安装

```
sudo apt install bochs
```

优点：简单方便;

缺点：书上说这样安装没有调试功能，实际好像是有的;

#### 方法二：下载源码压缩包

输入命令

```
tar zxvf bochs-2.7.tar.gz 解压
```

```
cd bochs-2.7 进入文件夹
```

```
./configure --enable-debugger --enable-disasm (参数就是打开调试功能)
```

输入这个以后，**报错**：

```
ERROR: X windows gui was selected, but X windows libraries were not found
```

搜索后发现：需要安装xorg-dev包

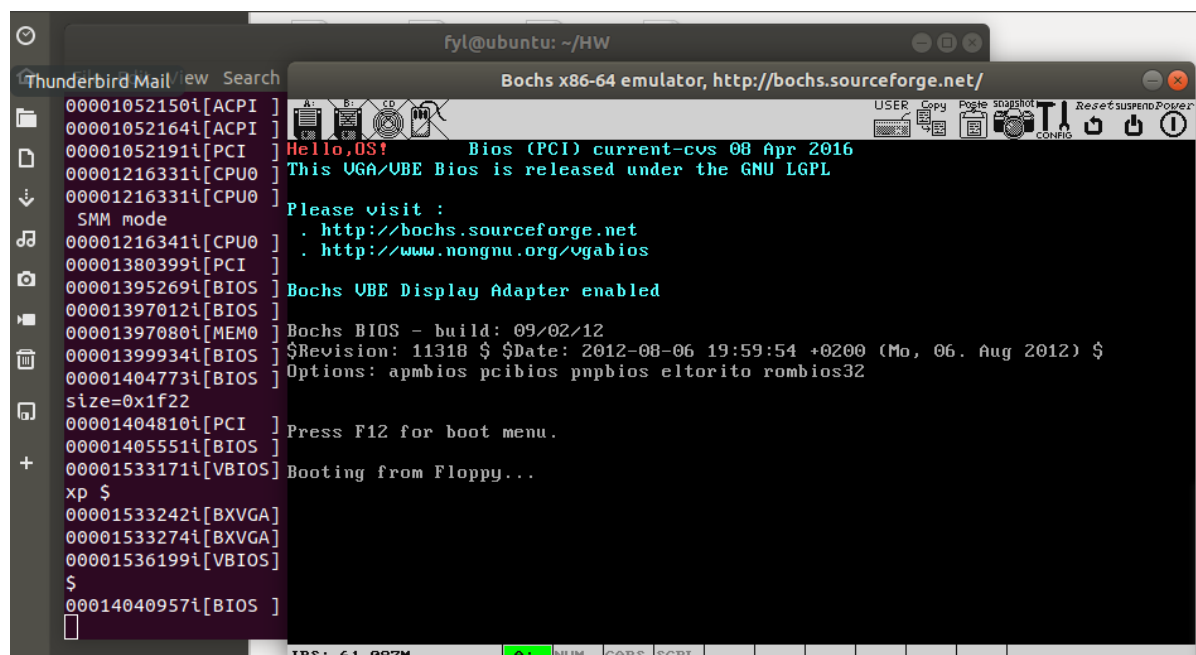
```
sudo make install
```

配置问题: <https://www.cnblogs.com/viviwind/archive/2012/12/21/2827581.html>

一开始想用源码安装，配置出问题以后感觉还是命令行安装更香，太省事了。而且，而且不同于《Orang's》书上的说法，我发现现在用apt命令安装的bochs一样有debugger;

x86汇编基础 x => x86-64汇编基础√

选择任意你喜欢的平台，参考 PPT 和《Orange's 一个操作系统的实现》，搭建 nasm + bochs 实验平台，在该实验平台上编写汇编文件 boot.asm 并用 bochs 执行，显示 Hello OS。请提交运行截图和代码。



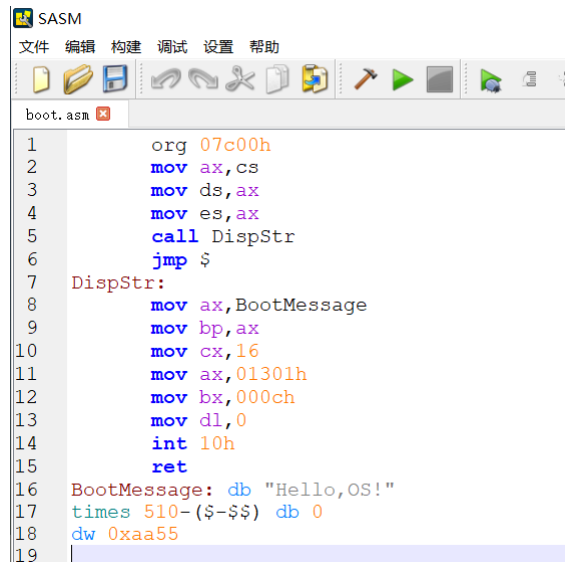
前面几步已经完成了nasm+bochs环境搭建。现在只需：

照着书或者PPT编辑汇编代码文件boot.asm => 用nasm汇编生成boot.bin =>

创建虚拟软盘映像a.img => 用dd命令将boot.bin写入a.img =>

添加bochs的配置文件bochsrc => 把bochsrc、a.img、boot.bin放在同一目录下 => 在终端输入bochs启动;

boot.asm文件：



```
1      org 07c00h
2      mov ax,cs
3      mov ds,ax
4      mov es,ax
5      call DispStr
6      jmp $
7  DispStr:
8      mov ax,BootMessage
9      mov bp,ax
10     mov cx,16
11     mov ax,01301h
12     mov bx,000ch
13     mov dl,0
14     int 10h
15     ret
16  BootMessage: db "Hello,OS!"
17     times 510-($-$$) db 0
18     dw 0xaa55
19
```

nasm编译命令：

## 基本格式

nasm -f <format> <filename> [-o <output>]

比如：

nasm -f elf myfile.asm ; 将myfile.asm 编译成 elf文件格式的 myfile.o文件，elf 是linux的可执行程序格式

## 常用编译命令

-f

指定要编译的格式，  
windows 用 -f win32 或者 win64  
linux 用 -f elf  
裸机用 -f bin

-o

指定编译后的文件的名称 否则会有默认的文件名称

-i

%include指定要查找的目录，最好是能在 NASMENV 环境变量里设置固定的路径，省的每次设置

-d

宏定义，编译的时候添加宏定义，这样就可以在编译的时候决定代码的内容了

-l (小写的L)

主要是查看汇编结果，调试之前对照用的，看哪里那个地址是不是写错了什么的

-e

这个也是调试的时候用的，主要是查看宏展开的等是否正确

```
nasm boot.asm -o boot.bin          #使用nasm汇编boot.asm生成“操作系统”的二进制代码

bximage      #输入以后开始“问答”
fd           #默认好像是hadr disk，需要输入fd
1.44         #默认也是1.44M，直接回车都行
a.img        #默认也叫a.img，直接回车都行

dd if=boot.bin of=a.img bs=512 count=1 conv=notrunc  #将boot.bin文件内容写入软盘中

#新建文本文件bochsrc并输入：
```

- 配置文件bochsrc：
  - display\_library: bochs使用的GUI库，在Ubuntu下面是sdl
  - meps: 虚拟机内存大小 (MB)
  - floppy: 虚拟机外设，软盘为a.img文件
  - boot: 虚拟机启动方式，从软盘启动

```
meps:32
display_library: sdl
floppya: 1_44=a.img, status=inserted
boot: floppy
```

```
#终端启动bochs
bochs
#没有显示Hello,OS!可能是因为处于debug模式，在终端输入c
c
#显示 Hello,OS!
```

## 1.8 大数运算要求

参考寻址方式和指令系统 PPT，熟悉汇编指令，用汇编语言（NASM）实现大整数（超过 64位）的加法和乘法。

### 1.2 汇编语言实践

参考寻址方式和指令系统 PPT，熟悉汇编指令，用汇编语言（NASM）实现大整数（超过 64位）的加法和乘法。

#### 1.2.1 输入输出格式

- 输入输出为两个整数  $x, y$  以空格分割，以回车结束。
- 输出为两行，第一行是两个整数的和，第二行是两个整数的乘积
- 程序使用标准输入（键盘）和标准输出（屏幕）

#### 1.2.3 要求

- 基本得分：实现  $0 \leq x, y \leq 10^{20}$  的全部情况
- 附加得分：实现  $-10^{20} \leq x, y \leq 10^{20}$  的情况



来来回回写了几次，不想写笔记了，但又很重要。前面几天磨磨唧唧的，写的文档和注释比代码多了几倍，最后一天进入状态了，一天写了几百行，注释和文档倒是一点儿都不想写了。

## 2.大数运算实现笔记

### 2.0 IDE——SASM

写nasm已经很折磨了，总不至于还要用记事本写吧.....

SASM是一个不错的工具。

### 2.1 分析

首先，我装的虚拟机是ubuntu18.04，是一个64位的linux系统。

**64位系统和32位系统的架构区别非常大**（包括寄存器位数、个数，某些命令，以及系统调用的命令和约定号！），下面会专门有一些笔记。

其次，现在大数运算要求实现从  $-10^{20}$  到  $10^{20}$  的加减乘运算。

通过取对数计算，发现：

$$10^{20} > 2^{64}$$

$$10^{20} < 2^{71}$$

$$10^{40} < 2^{134}=2^{128+6}$$

因此对于2个输入、1个加减法的结果、1个乘法的结果，我在数据区(.data)和声明区(.bss)准备了这样的空间：

数据区：

显然 tips 和 tipsLen 只是为了输出提示信息准备的变量；isNeg1 和 isNeg2 用于代表第一个数和第二个数的符号位；换行符 nextLine是为了打印换行而准备的，0AH是换行的ascii码；

```
26 ;-----分割线-----;
27 ;-----数据区-----;
28 section .data
29     tips: db "please input 2 numbers seperated by space",0Ah
30     tipsLen: equ $ - tips
31     ;第一个数和第二个数是否是负数,值为0代表正数,1代表负数,默认为正
32     isNeg1:db 0
33     isNeg2:db 0
34     ;换行符
35     nextLine:db 0AH
```

声明区：

因为是大数，最好用字符数组的形式，1个字节存大数的1位；

len1 和len2是第一个和第二个数的位数（也就是数组的长度），resb 代表预留1个字节的空间；1个字节无符号可以表示0~255，1个字节表示位数绰绰有余；

inputBuff 是输入缓冲区，50个字节绝对够用了；inputLen 是输入的字节数；

第一个数 firstPtr 和第二个数 secondPtr 是指向数组起始地址的指针 pointer；使用时一定要记得“解引用”，即用QWORD[pointer]来获得指向地址的内容；

对于和与积的结果，我都是用了4个东西(4元组)来表示它们：

存储结果的**字符数组**、指向**首地址的指针**、运算结果数的**位数**、**符号位**

```
37 ;-----分割线-----;
38 ;-----声明区-----;
39 section .bss
40 ;第一个数和第二个数的位数
41 len1:resb 1
42 len2:resb 1
43
44 ;10的20次方也就是21位数，1位数1个字节，50个字节的缓冲区够2个大数输入了
45 inputBuff: resb 50
46 inputLen:resb 1
47
48 ;第一个数和第二个数的起始地址
49 ;when use:QWORD[ptr]
50 firstPtr:resq 1
51 secondPtr:resq 1
52
53 ;2个21位的数相加，绝对值顶多22位数；值得注意的是sumPtr才是指向和数组首元素的指针
54 sumSpace:resb 22
55 sumPtr:resq 1
56 sumLen:resb 1 ;长度
57 sumSign:resb 1 ;符号
58
59 ;21位数乘以21位数，顶多42位；值得注意的是prodPtr才是指向乘积数组首元素的指针
60 prodSpace:resb 42
61 prodPtr:resq 1
62 prodLen:resb 1 ;长度
63 prodSign:resb 1 ;符号
```

## 2.2 输入与输出：系统调用 或 调用C函数 printf

△☆方法一：系统调用(我使用的这种)

### 2.2.1 系统调用实现read

首先搜索各种参考博客：

<https://www.cnblogs.com/tongye/p/9830006.html>

<https://blog.csdn.net/u013043103/article/details/108580611>

通过不断在csdn、知乎、stack overflow上搜索，我基本搞懂了64位系统下的系统调用逻辑；

然后查阅64位系统的系统调用表：

系统调用号	函数名	入口点	源代码
0	read	sys_read	<a href="#">fs/read_write.c</a>
1	write	sys_write	<a href="#">fs/read_write.c</a>
2	open	sys_open	<a href="#">fs/open.c</a>
3	close	sys_close	<a href="#">fs/open.c</a>
4	stat	sys_newstat	<a href="#">fs/stat.c</a>
5	fstat	sys_newfstat	<a href="#">fs/stat.c</a>
6	lstat	sys_newlstat	<a href="#">fs/stat.c</a>
7	poll	sys_poll	<a href="#">fs/select.c</a>
8	lseek	sys_lseek	<a href="#">fs/read_write.c</a>

系统调用read()用C语言函数形式写出来：

```
size_t read(int fildes,void *buf,size_t nbytes);
```

fildes 是文件描述符，**内核利用文件描述符来访问文件**，它是一个非负的整数，当打开现存文件或者新建一个文件时，都会返回一个文件描述符。有多少文件描述符取决于系统的配置情况，当一个程序开始运行时，它一般有 3 个已经打开的文件描述符：**标准输入 0；标准输出 1；标准错误 2。**

因为要使用read，从标准输入读取数据，第一个参数当然该默认为0；第二个参数是指向输入**缓冲区的首地址**，第三个参数是**最多读入的字节数**

代码实现：

```
;职责:读取键盘输入 read
;arg: (rax,rdi), rsi, rdx;前两个参数在函数内设置
;要保存输入的缓冲区起始地址，需要存到rsi里
;将最大读取的字节大小提前mov到rdx里
;ret: rax中存储总共输入位数
read:
    pusha                ; 保存现场
    mov rax,0             ; 64bit系统下sys_read的系统调用编号为0
    mov rdi,0             ; 文件句柄0对应输入流stdin
    syscall               ; 系统调用(64bit下可用)
    popa                  ; 恢复现场
    ret
```

首先注意这里的**pusha和popa也是个人实现的**；64位系统下的寄存器太多了，pusha和popa开销太大，系统没有设计它们，而寄存器较少的32位系统下是有的。

其次 mov rax,0 是因为64位系统下**系统调用号的传递约定由rax寄存器负责**，要使用read，就将read的系统调用号0传入rax中。

mov rdi,0 是因为文件句柄0代表从标准输入流 stdin 中读取数据；也就是size\_t read( int fildes,.....) 这个函数里的第一个参数

还要注意**64位系统下**最终进行系统调用，是使用命令 **syscall**，对比 **32位下使用的 int 80h**

使用read：

在上图的read里，我们设置好了系统调用号和输入流，也就是说，要使用read，我们现在还差输入缓冲区buf和输入最大字节nbytes2个参数没有设置；

```
size_t read(int fildes,void *buf,size_t nbytes);
```

rsi 寄存器约定负责传递第二个变量 void\* buff，即输入缓冲区的首地址；

rdx 寄存器约定负责传递第三个变量，即最多读取的字节数；

```
    ;读取输入
    mov rsi,inputBuff    ; 保存输入的变量,缓冲区
are mov rdx,50           ; 最大读取长度,最多读取50个字节
    call read
```

注意有call就该有ret：主程序这里用了call，封装的read里就该有 ret；

## 2.2.2 系统调用实现print

有了通过系统调用实现和使用read的经验，print的实现和使用就让人感到轻松；

C语言形式：

```
size_t write(int fildes,const void *buf,size_t nbytes);
```

实现：

实现和read几乎一模一样，只不过系统调用号变为了1，同时第一个参数文件描述符也应该是1；

（个人觉得这其实是64位系统设计的一个目的：read就用2个0，write/print就用2个1，好记、方便）

```
    ;-----封装函数区-----;
;职责:打印 print
;注意要输出的字符串地址,需要提前mov到rsi里
;要输出的字符串长度,需要提前mov到rdx里;
;有call就有ret
;ret: rax中存储输出的字节数
print:
    pusha                ; 保存现场
    mov rax,1            ; sys_write的系统调用编号为1
    mov rdi,1            ; 文件句柄1对应输出流stdout
    syscall              ; 系统调用(64bit下可用,对比32位下是int 80h)
    popa                 ; 恢复现场
    ret                  ; 返回
```

使用：

使用print时也和使用read几乎一模一样，在上图的print实现里，我们设置好了系统调用号和输出流，接下来还差输出的首地址 buf 和输出字节数nbytes2个参数没有设置：

rsi 寄存器约定负责传递第二个变量 void\* buff，即输出缓冲区的首地址；

rdx 寄存器约定负责传递第三个变量，即需要输出的字节数；

```

;提示
mov rsi,tips           ; 字符串地址
mov rdx,tipsLen        ; 字符串长度
call print

```

### 2.2.3 调用C函数

在 nasm 中，还可以调用C的库函数。

这需要使用 extern 关键字，后面跟函数名称，以便在连接时将 C函数copy-paste到源代码中。比如使用C的printf:

```
extern printf
```

然后调用printf:

```

;
mov rdi, fmt
mov rsi, rax
mov rax, 0 ;表示不使用xmm寄存器，xmm寄存器是128位的，也就是16字节长度
; printf('%ld',rsi)
call printf

```

我们知道printf的形式是： printf("format string %s",ptr1);

约定寄存器rdi 的内容是第一个参数 格式控制串，rsi是对应格式控制符的参数；

相关介绍博客：

[https://blog.csdn.net/qg\\_31917799/article/details/87930452](https://blog.csdn.net/qg_31917799/article/details/87930452)

<https://blog.csdn.net/sivolin/article/details/41895701>

## 2.3 64位和32位的不同点

可以参考nasm官网的参考手册：<https://nasm.us/docs.php>

### 2.3.1 系统调用号

64位系统下的系统调用号和32位系统下是不同的！

下图是64位的：

系统调用号	函数名	入口点	源代码
0	read	sys_read	fs/read_write.c
1	write	sys_write	fs/read_write.c
2	open	sys_open	fs/open.c
3	close	sys_close	fs/open.c
4	stat	sys_newstat	fs/stat.c
5	fstat	sys_newfstat	fs/stat.c
6	lstat	sys_newlstat	fs/stat.c
7	poll	sys_poll	fs/select.c
8	lseek	sys_lseek	fs/read_write.c

以下是32位的：

1	sys_exit
2	sys_fork
3	sys_read
4	sys_write
5	sys_open
6	sys_close

read、write都不同，差距还是挺大的；

2.3.2 寄存器/寄存器个数

```
AL/AH, CL/CH, DL/DH, BL/BH, SPL, BPL, SIL, DIL, R8B-R15B
AX, CX, DX, BX, SP, BP, SI, DI, R8W-R15W
EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, R8D-R15D
RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, R8-R15
```

上图为64位系统下的寄存器；可以通用的寄存器比32位的多了很多。  
而且这张图也部分体现了：x86\_64为什么可以兼容x86\_32，AL/AH—AX—EAX—RAX；

2.3.3 64位系统下没有pusha/popa

因为64位系统下的寄存器数量太多，如果都要pusha/popa，开销太大；

### 2.3.4 指针/解引用

解引用时要注意，64位系统下，寄存器是64位，64 bit= 8 Byte=1 QWORD; 如果在**内存某单元和寄存器之间相互赋值**，可能要注意：

[]之前可能使用QWORD；

比如：

这里要将 `rax` 中的 **第一个字符数组的首地址**传给 指针 `firstPtr` 时，就要注意是 将`QWORD[first]` 修改；

```
100      mov BYTE[isNeg1],dl
101      mov QWORD[firstPtr],rax
```

正如同 C语言声明指针，必须指明 指针指向数据的类型 后才能使用：

这里的`firstPtr`是指向某个字节的首地址，`QWORD[]`意思是不能只用1个字节，要以`firstPtr`为首地址，将8个字节连起来使用；

如果是32位系统，寄存器是32位的，可能需要使用的就是`DWORD[]`了。

## 2.4 64位系统下的寄存器约定

根据 `nasm` 官网的手册：<https://nasm.us/docs.php>

在PDF文件133页

### 2.4.1 传参

对于从左到右的**整型** 参数，首先依次使用6个寄存器：

`RDI`, `RSI`, `RDX`, `RCX`, `R8`, and `R9`

更多的整形参数用 **栈** 传递。

`rax`、`r10`、`r11`会被**系统调用重置/摧毁**，在函数调用的**函数内部**是可用的，他们不需要保存；

整形返回值被先后传递在 **`rax`和`rdx`**中。

浮点数会使用SSE寄存器传递，80位的long double型例外；

浮点数会通过**`XMM0`到`XMM7`**这8个寄存器传递；返回值是在寄存器`XMM0`和`XMM1`中；

长整型用Stack传递，返回到`ST0`和`ST1`中；

所有SSE和x87寄存器会在系统调用中被重置/摧毁。

64位Unix上，`long`是64位的，会使用整型寄存器传递；

## 12.1 Register Names in 64-bit Mode

NASM uses the following names for general-purpose registers in 64-bit mode, for 8-, 16-, 32- and 64-bit references, respectively:

AL/AH, CL/CH, DL/DH, BL/BH, SPL, BPL, SIL, DIL, R8B-R15B  
AX, CX, DX, BX, SP, BP, SI, DI, R8W-R15W  
EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, R8D-R15D  
RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, R8-R15

This is consistent with the AMD documentation and most other assemblers. The Intel documentation, however, uses the names R8L-R15L for 8-bit references to the higher registers. It is possible to use those names by defining them as macros; similarly, if one wants to use numeric names for the low 8 registers, define them as macros. The standard macro package `altreg` (see section 6.1) can be used for this purpose.






## 2.5 主文件/主函数

主文件组织 `BigNum.asm`:

宏命令—数据定义区/声明区—代码区（主函数main）

文件组织:

Locate、MyAdd、MyTimes分别定义在同名.asm文件中

 <code>BigNum.asm</code>	2021/11/1 14:35	Assembler Source	8 KB
 <code>Locate.asm</code>	2021/10/24 22:31	Assembler Source	2 KB
 <code>MyAdd.asm</code>	2021/10/25 2:13	Assembler Source	5 KB
 <code>MyTimes.asm</code>	2021/10/25 2:13	Assembler Source	4 KB
 <code>UserIO.asm</code>	2021/11/1 14:42	Assembler Source	2 KB

main的主函数流程也非常简单:

输出提示信息—输入2个大数—对输入进行处理(使用Locate)—调用加法并输出结果(MyAdd)—调用乘法并输出结果(MyTimes)—结束

输入输出没什么好讲的; 在2.2里已经讲了UserIO的文件内容

```
66 section .text
67 ; 程序入口
68 global main
69 main:
70     mov rbp, rsp; for sscorrect debugging
71     ; 提示
72     mov rsi, tips                ; 字符串地址
73     mov rdx, tipsLen             ; 字符串长度
74     call print
75
76     ; -----小分割线-----;
77     ; -----读取输入-----;
78     ; ret: rax中存储总共输入位数
79     mov rsi, inputBuff           ; 保存输入的变量, 缓冲区
80     mov rdx, 50                  ; 最大读取长度, 最多读取50个字节
81     call read
82     mov BYTE[inputLen], al       ; 输入的总长度
```

## 2.6 输入处理 Locate

如注释; 从键盘输入的2个大数, 是以字符串的形式存在;

Locate函数通过循环, 设置好1个数的符号、(数字开始的)首地址;



```

;-----封装函数:Locate-----;
;Locate:主要目的是实现 split , 将输入的字符串分割为2个整数
;职责: 1.确定1个数的符号 2.确定1个数的位数/长度 3.确定数组的首地址
;args: rdi, 该数组的首地址(可能包含首个符号位)
;ret: rax存储该数去除符号位后的首地址;
; dl存储该数的符号
; rdi存储跳出循环的那个地址
;实现: 循环,一直做rdi++,遇到空格或者换行则返回,遇到+号、-号设置符号;遇到第一个数字就开始存储数字
Locate:
    cmp byte[rdi],20H ;ascii 20H空格
    jz return ;return
    cmp byte[rdi],0AH ;ascii 0AH换行
    jz return ;return

    cmp byte[rdi],2BH ;ascii 2BH +号
    jz setPos
    cmp byte[rdi],2DH ;ascii 2DH -号
    jz setNeg

    inc rdi
    jmp Locate ;loop

```

设置符号:

```

;setPos
;职责: 发现正号,设置dl为0(dl为存储符号位的部件)
; 因为发现了符号位,rax负责存储绝对值的首地址,故rax要加1
setPos:
    add rax,1
    inc rdi
    jmp Locate

;setNeg
;职责: 发现负号,设置dl为1(dl为存储符号位的部件)
; 因为发现了符号位,而rax负责存储绝对值的首地址,故rax要加1
setNeg:
    add rax,1
    mov dl,1
    inc rdi
    jmp Locate

return:
    ret

```

## 2.7 加减法实现 MyAdd

```

;-----我的大数加法-----;
;职责: 根据两个加数的符号位是否相同,决定执行绝对值相加还是绝对值相减
; 并在处理过程中,设置好和的位数、和的结果(以字符数组的形式)、字符数组的首地址
;args: rdi 第一个操作数的首地址; rsi 第二个操作数的首地址;
; dh 第一个操作数的符号; dl 第一个操作数的长度
; ch 第二个操作数的符号; cl 第二个操作数的长度
; rax 预留空间的尾地址
;ret: rax 数组的有效首地址
; r8b 和的符号; r9b 和的位数
; 设置好sum的每一位
;note: 过程中用b1(rbx)存储carrier,减法中还用r11和r12分别存储第一个和第二个操作数的首地址
;
;实现: 首先初始化,然后根据两个加数的符号位是否相同(异或后是否为0),决定执行绝对值相加还是绝对值相减
;
;绝对值相加:
; 如果是绝对值相加,先确定符号,然后做绝对值相加的循环: 1.第一个加数有没有遍历完? 2.第二个加数有没有遍历完?
; 3.用进位器暂存 Sum(i)=A(i)+B(i)+Carry(i); 4.暂存的Sum(i)>10? 大于10就将进位器设置为1;否则将进位器设置为0;个位存入这一位
; 5.让指针指向高一位数 6.进入下一次循环 7.假如两个数都遍历完,判断进位器是否为1,如果是1,还需要向前多设置1个1;最后返回
;绝对值相减:
; 绝对值相减的问题是无法确定|A|-|B|的符号,可以直接做|A|-|B|,如果|A|>|B|,那么做完减法后最终的进位器会是0;
; 如果|A|<|B|,那么做完后进位器值会是-1,可以设置符号为B,然后交换A\B,重新调用一次减法的循环,让|B|去做被减数
;
; 1.被减数有没有遍历完? 2.减数有没有遍历完?
; 3.用进位器暂存 Sum(i)=A(i)-B(i)+Carry(i);
; 4.暂存的Sum(i)>0? 大于0就将进位器设置为0,并保存值;否则将进位器设置为-1,向前借1位,然后这一位的数字是10+sum
; 5.让指针指向高一位数 6.进入下一次循环
; 7.假如两个数都遍历完,判断进位器是否为0,如果是0,可以设置好符号等后直接返回;如果是-1,设置符号,然后交换减数和被减数,重新做一次减法
ret_main:
    ret

```

具体的代码实现感觉没有必要贴上来;

有一个问题就是:  $193 + (-190) = 3$  但我存在内存中的值会是 003 ;输出时需要做些处理;

## 2.8 乘法实现 MyTimes

同样, 具体的代码实现感觉没有必要贴上来;

乘法的难点在于二重循环、mul指令和div指令、ascii码的偏移量

```
;-----我的大数乘法-----;
;职责: 在处理过程中,设置好积的位数、积的结果(以字符数组的形式)、字符数组的首地址
;args: rdi 第一个操作数的首地址; rsi 第二个操作数的首地址;
;      dh 第一个操作数的符号; dl 第一个操作数的长度
;      ch 第二个操作数的符号; cl 第二个操作数的长度
;      rax 预留空间的尾地址
;ret:   rax 数组的有效首地址
;      r8b 积的符号; r9b 积的位数
;      设置好prod的每一位
;note: 过程中用b1(rbx)存储carrier,还用r10、r11和r12分别存储预留空间的尾地址、第一个和第二个操作数的首地址
;      r13\14\15做tmp用
;      mul指令、div指令的使用需要留心
;      mul指令默认要用rax/eax和rdx/edx, 只有一个参数做乘数, 而且只能是 register; mul src
;      div指令也是只有一个register做除数, div src
;
;实现:
;      A x B  a1a2.....an * b1b2.....bm
;      1.首先初始化,并且设置符号位,负负得正,因为用0代表正,1代表负,设置两者乘积的符号,可以通过 xor 异或运算
;      2.将存储积的预留空间全部初始化为 真正的0值, 是0,不是'0' (这一步很重要)
;      3.进行双重循环: 外层我选的是b,内层选的是a
;          对bm和an,product(用carry暂存)=bm * an +carry+当前这一位的值;
;          product>=10? >=10,则当前这一位的新值=product%10,carry=product/10;否则,当前这一位的新值=product,carry=product/10
;          内层循环到下一步,判断 a有没有遍历完,没遍历完,就让a(n-1)替代a(n); a遍历完,让a重新指向个位,让bm向前到b(m-1)
;
;      (乘法的难度就在于设置这个双重循环,尤其循环的终止条件和每一步的递进要考虑清楚,还有在计算的时候一定要注意去除ascii码的偏移量)
;      4.双重循环做完以后,判断进位器carry>0? 大于0就要多进1位
;      5.返回之前,给每一位加上ascii码的偏移量030H,方便数字能正确显示;设置好后返回
ret_from_times:
```

## 2.9 测试结果

见big\_number文件夹下的TestCase