|  |
| --- |
| NJUCS |
| 汇编语言实验讲义 |
| V1.0 |

编写组

2017-7-16

目录

[第 1 章 Linux 简介 3](#_bookmark0)

* 1. [Linux 起源 3](#_bookmark1)
  2. [Linux 安装 5](#_bookmark2)
     1. [虚拟机安装 5](#_bookmark3)
     2. [Linux 系统安装 7](#_bookmark4)
  3. [Linux 配置 11](#_bookmark5)
  4. [Linux 常用命令和工具 14](#_bookmark6)
     1. [查看手册命令 14](#_bookmark7)
     2. [文件和文件夹操作命令 15](#_bookmark8)
     3. [显示文件内容命令 18](#_bookmark9)
     4. [查看更改权限命令 19](#_bookmark10)
     5. [查看进程和杀死进程命令 20](#_bookmark11)

[第 2 章 Linux 下编程调试工具 21](#_bookmark12)

* 1. [源代码版本控制工具 git 21](#_bookmark13)
  2. [文本编辑器 vi/vim 25](#_bookmark14)
  3. [通用编译和链接工具 gcc 28](#_bookmark15)
  4. [汇编编译工具 as 30](#_bookmark16)
  5. [符号调试器 gdb 32](#_bookmark17)
  6. [维护工具 make 33](#_bookmark18)
  7. [反汇编工具 objdump 35](#_bookmark19)

[第 3 章 实验内容 39](#_bookmark20)

[实验 1：Linux 编程基础实验 39](#_bookmark21)

[实验 2 数据表示和运算实验 43](#_bookmark22)

[实验 3 数据分支控制实验 48](#_bookmark23)

[实验 4 复杂结构实验 52](#_bookmark24)

[实验 5 存分配、缓冲区溢出实验 59](#_bookmark25)

[综合实验 二进制炸弹实验 62](#_bookmark26)

# 第1章 Linux 简介



Linux 官方吉祥物 Tux

## Linux 起源

Linux 是一种自由和开放源代码的类 UNIX 操作系统。该操作系统的内核由林纳斯·托瓦兹（Linus Torvalds）在 1991 年 10 月 5 日首次发布。在加上用户空间的应用程序之后， 成为 Linux 操作系统。Linux 也是自由软件和开放源代码软件发展中最著名的例子。只要遵循 GNU 通用公共许可证，任何个人和机构都可以自由地使用 Linux 的所有底层源代码，也可以自由地修改和再发布。大多数 Linux 系统还提供 GUI 的 X Window 之类的程序。

Linux 严格来说是单指操作系统的内核，因操作系统中包含了许多用户图形接口和其他实用工具。如今 Linux 常用来指基于 Linux 的完整操作系统，内核则改以 Linux 内核称之。由于这些支持用户空间的系统工具和库主要由理查德·斯托曼于 1983 年发起的 GNU 计划提供，自由软件基金会提议将其组合系统命名为 GNU/Linux，但 Linux 不属于 GNU 计划， 这个名称并没有得到一致认同。

Linux 最初是作为支持英特尔 x86 架构的个人电脑的一个自由操作系统。目前 Linux 已经被移植到更多的计算机硬件平台，远远超出其他任何操作系统。Linux 可以运行在服务器和其他大型平台之上，如大型主机和超级计算机。世界上 500 个最快的超级计算机 90％以上运行 Linux 发行版或变种，包括最快的前 10 名超级电脑运行的都是基于 Linux 内核的操作系统。Linux 也广泛应用在嵌入式系统上，如手机（Mobile Phone）、平板电脑（Tablet）、路由器（Router）、电视（TV）和电子游戏机等。在移动设备上广泛使用的 Android 操作系统就是创建在 Linux 内核之上。

UNIX 操作系统，是美国 AT&T 公司贝尔实验室于 1969 年完成的操作系统。最早由肯·汤普逊（Ken Thompson），丹尼斯·里奇（Dennis Ritchie），道格拉斯·麦克罗伊

（Douglas McIlroy），和乔伊·欧桑纳于 1969 年在 AT&T 贝尔实验室开发。于 1971 年首次发布，最初是完全用汇编语言编写，这在当时是一种普遍的做法。后来，在 1973 年用一个重要的开拓性的方法，Unix 被丹尼斯·里奇用编程语言 C（内核和 I/O 例外）重新编写。高级语言编写的操作系统具有更佳的兼容性，能更容易地移植到不同的计算机平台。

1983 年，理查德·马修·斯托曼（Richard M. Stallman）创立 GNU 计划。目标是为了发展一个完全自由的类 Unix 操作系统。1985 年，理查德·马修·斯托曼发起自由软件基金会并且在 1989 年撰写 GPL。1990 年代早期，GNU 开始大量的产生或收集各种系统所必备的组件，像是——库、编译器、调试工具、文本编辑器、网页服务器，以及一个 Unix 的

用户界面（Unix shell）——但是像一些底层环境，如硬件驱动、守护进程运行内核仍然不完整。

通常情况下，Linux 被打包成供个人计算机和服务器使用的 Linux 发行版，一些流行的主流 Linux 发布版，包括 Debian（及其派生版本 Ubuntu、Linux Mint）、Fedora（及其相关版本 Red Hat Enterprise Linux、CentOS）和 openSUSE 等。Linux 发行版包含 Linux 内核和支撑内核的实用程序和库，通常还带有大量可以满足各类需求的应用程序。个人计算机使用的 Linux 发行版通常包含 X Window 和一个相应的桌面环境，如 GNOME 或 KDE。桌面 Linux 操作系统常用的应用程序，包括 Firefox 网页浏览器、LibreOffice 办公软件、GIMP 图像处理工具等。由于 Linux 是自由软件，任何人都可以创建一个匹配自己需求的 Linux 发行版。本讲义后文提到的 Linux 默认指代 Linux 的 Ubuntu 发行版。

#### Linux 系统架构

1、Linux 分区

Linux 分区的形式一般是 Swap 分区、根分区/和其他用户按自己需求定义的分区，如果用户不进行自定义，则其他所有分区统一作为子目录挂载在根目录/下。

其中 Swap 分区的作用类似于 Windows 操作系统中的虚拟内存。

2、 Linux 文件类型

普通文件：同 Windows 中的文件类似；目录文件：目录即文件；链接文件；设备文件：字符设备文件/块设备文件；管道；堆栈；套接字。

目录文件与索引节点关系

Linux 系统中每个文件赋予惟一数字，此数值称为索引节点。索引节点存储在索引节点表中，该表在磁盘格式化时被分配。Linux 通过查找从根目录开始的一个目录链来找到系统中的任何文件。Linux 通过目录链实现对整个文件系统的操作。

3、Linux 文件系统

ext2、ext3、ext4 等：默认的文件系统。采用日志式的管理机制，使文件系统具有很强的快速恢复功能。

Swap 文件系统：是 Linux 作为交换分区使用的。

vFAT 文件系统：Linux 中把 Dos 采用的 FAT 文件系统都称为 vFAT 文件系统。

NFS 文件系统：是指网络文件系统。

ISO9660 文件系统：光盘所使用的文件系统。内存文件系统：proc、sys、ramdisk。

嵌入式文件系统：cramfs、jffs、yaffs 等。

4、Linux 目录结构

/bin 该目录中存放 Linux 的常用命令（一般是可执行程序或链接）

/boot 引导目录

/dev 该目录包含了 Linux 系统中使用的所有外部设备，它实际上是访问这些外部设备的端口，你可以访问这些外部设备，与访问一个文件或一个目录没有区别

/sbin 该目录用来存放系统管理员的系统管理程序

/usr 用户应用程序和文件都存放在该目录下

/etc 该目录存放了系统管理时要用到的各种配置文件和子目录，例如网络配置文件、文件系统等

/home 用户的主目录

/lib 该目录用来存放系统动态连接共享库，几乎所有的应用程序都会用到该目录下的共享库

/tmp 用来存放不同程序执行时产生的临时文件

/lost+found 该目录在大多数情况下都是空的，但当突然停电、或者非正常关机后，有些文件就临时存放在这里

/mnt 该目录在一般情况下也是空的，插上 U 盘之后你的 U 盘一般挂在这儿

/proc 可以在该目录下获取系统信息，这些信息是在内存中由系统自己产生的

/root 超级用户的主目录

/sys sys 文件系统

/pro cproc 文件系统

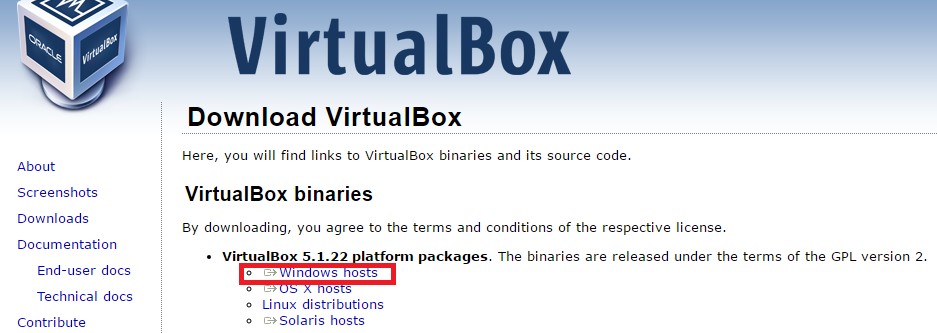
## Linux 安装

为了安全及让操作更便捷，我们在虚拟机上安装 Linux 系统（如果喜欢，可以自己研究如何在自己的电脑上安装和使用 Linux 系统）。

虚拟机我们选择使用 VirtualBox。

## 虚拟机安装

#### 1、虚拟机下载

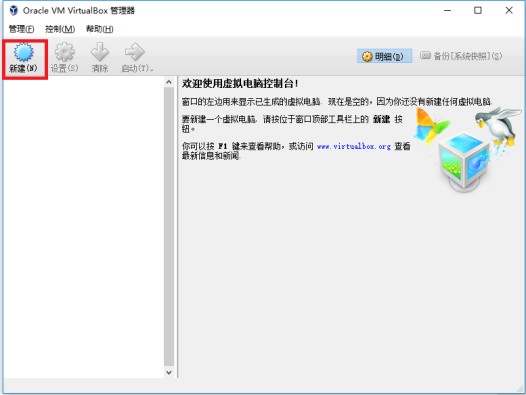
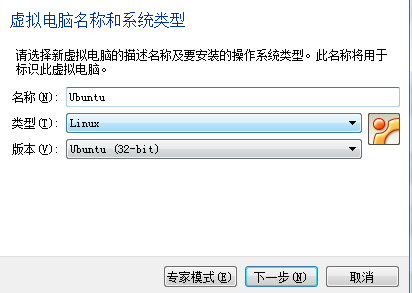
<https://www.virtualbox.org/wiki/Downloads>

打开链接，点击红框位置下载虚拟机安装包

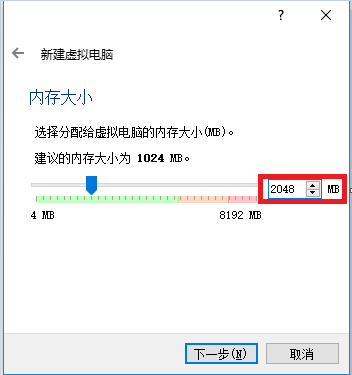


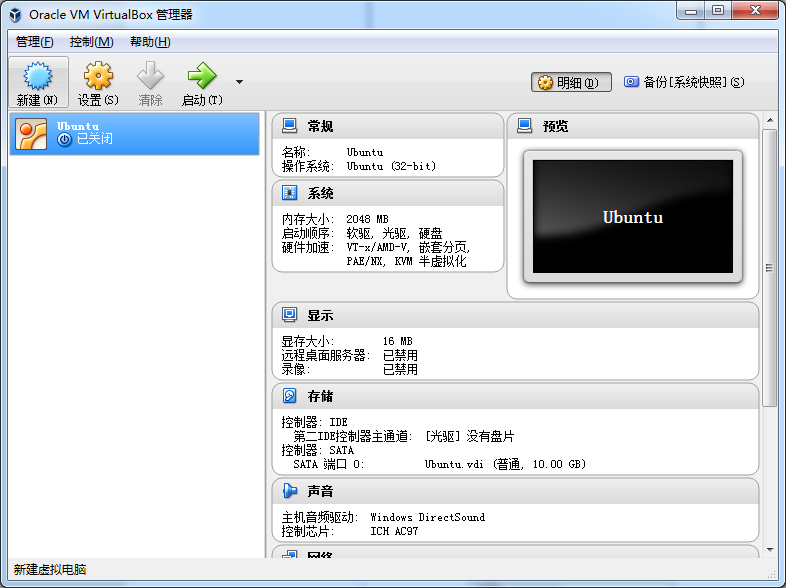
#### 2、虚拟机安装：双击安装，安装过程跳出任何警告和弹窗都选择确定选项。

**3、虚拟机创建**

由于实验要求，**请选择 32 位！**



建议修改内存大小为 2048MB。**后续步骤全点继续**，使用默认设置即可。然后一路点击默认选项直至 Linux 虚拟机创建完成。

## Linux 系统安装

#### 1、Linux 下载

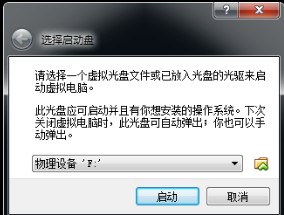
<http://cn.ubuntu.com/download/>

选择并下载 **32 位的系统**。（实际上，64 位系统可以支持 32 位程序，而 32 位系统不可

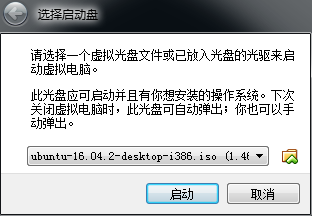
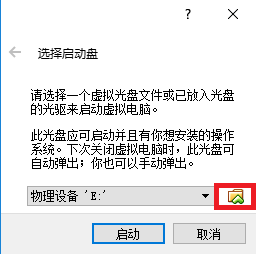
以支持 64 位程序，然而由于本门课实验要求，**我们下载 32 位系统**）

理论上系统版本不影响实验过程和结果，以防万一可以确保系统版本相同（amd64 代表 64 位系统，i386 代表 32 位系统，图中是 64 位系统的安装文件）。

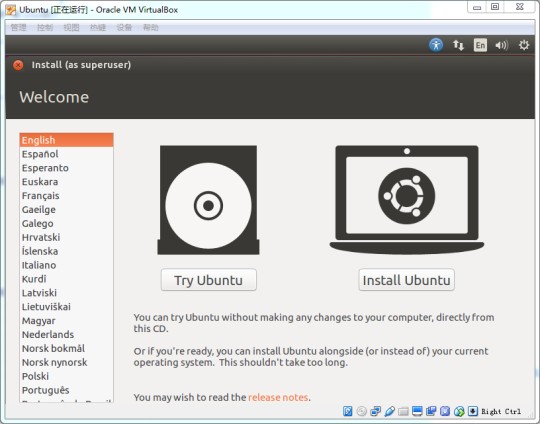
#### 2、Linux 安装



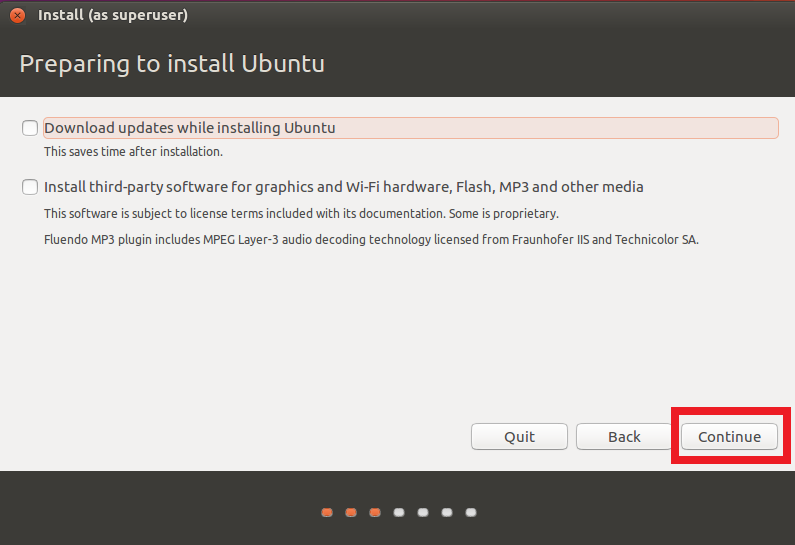
启动刚才创建的 Linux 虚拟机。



选择刚才下载的 **32 位安装文件**后点击启动。



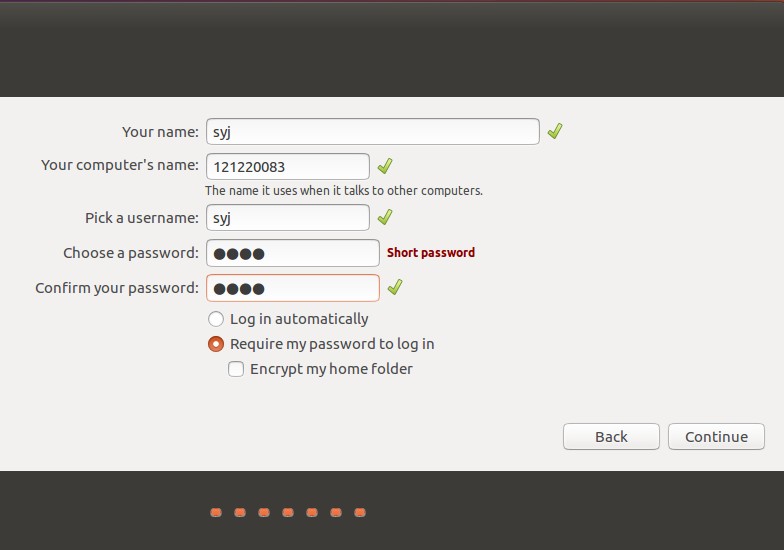
可以选择安装中文系统。**当然更推荐英文系统。**



都不需要。**直接继续。**

#### 仅本次实验推荐使用这个选项，自己使用请！千！万！别！选！ 仅本次实验推荐使用这个选项，自己使用请！千！万！别！选！ 仅本次实验推荐使用这个选项，自己使用请！千！万！别！选！

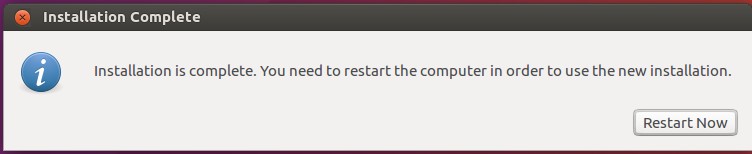
（有兴趣的同学可以咨询同学|助教|网上搜索如何安全地安装 Linux 系统） **之后一路点继续。**



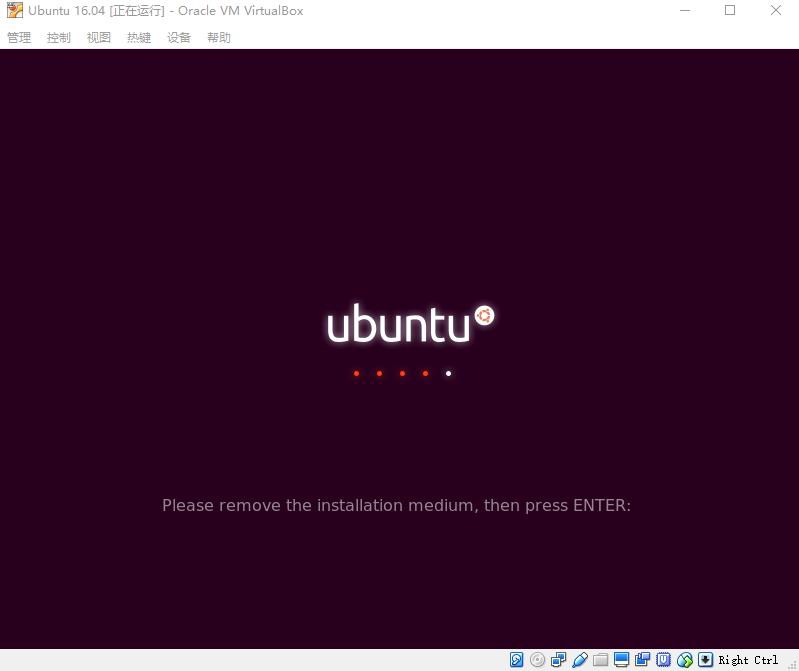
推荐这样设置用户名（姓名缩写）和计算机名（学号）。并自行设置密码（**推荐简单密码例如 1234**，并且千万不要忘记）。



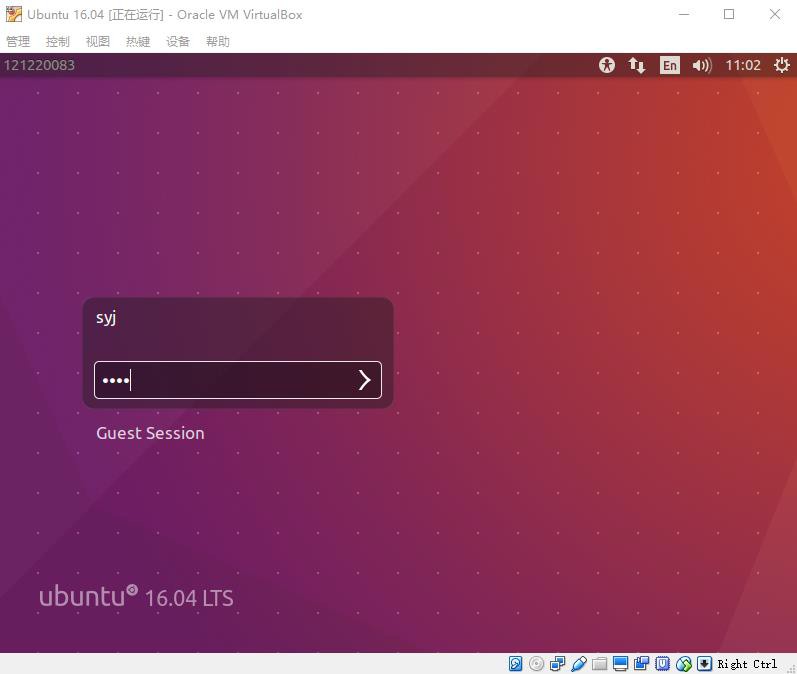
#### 在这个安装界面，单击进度条会在下方出现黑色命令行和右上角 skip，可以点 skip 跳过某些没什么用的下载加快安装。然后等待安装完成。



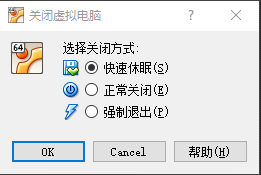
安装完成后需要重启。



#### 重启如果出现这个界面，按回车。



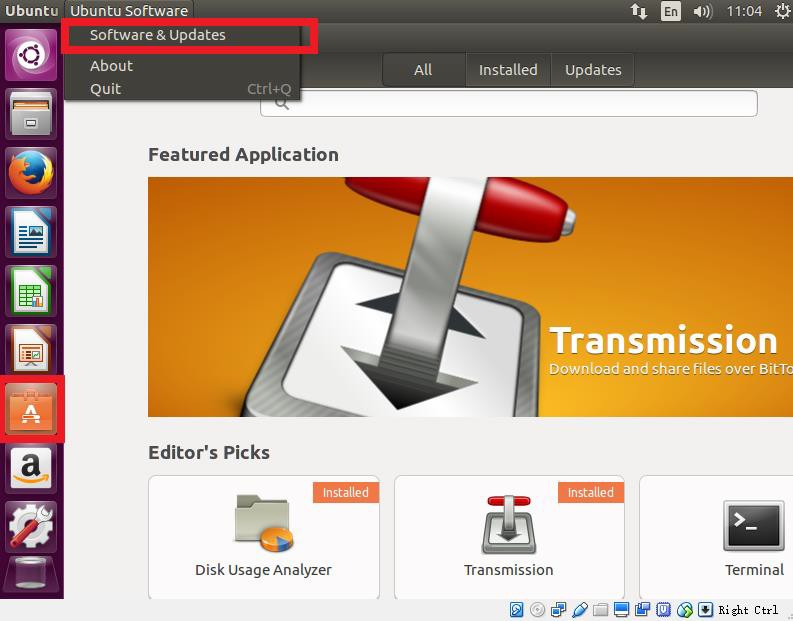
安装完毕，输入密码就可以进入系统了。**对了，当需要关机时，推荐直接点虚拟机窗口的 X 并选择快速休眠，这样可以让你下次进入系统快一点。**



## Linux 配置

Ubuntu 发行版使用的是类似应用商店的软件发布模式。通过修改软件源可以提高下载速度（例如中国的服务器肯定比外国的快，教育网的大多比非教育网的快）。

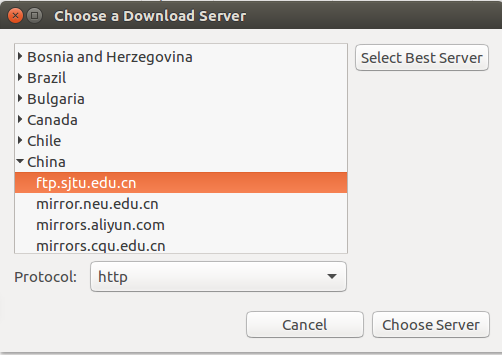
#### 1、设置软件源（使软件下载更快）



**打开左侧应用中心，鼠标移动到上方任务栏，点击新出现的标题名（第二个标题名） 后点击 Software & Updates。**

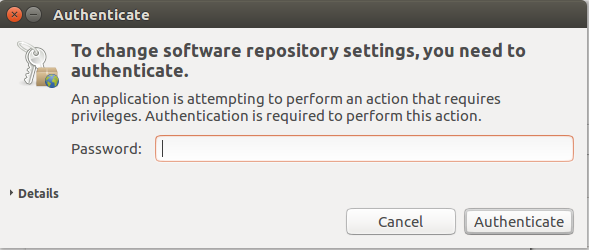


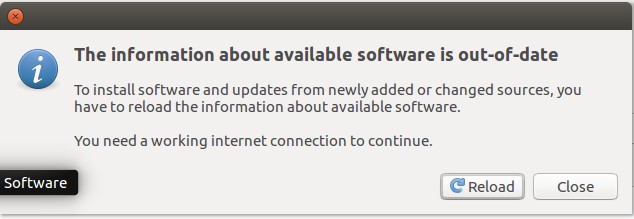
点击进入服务器选择界面。



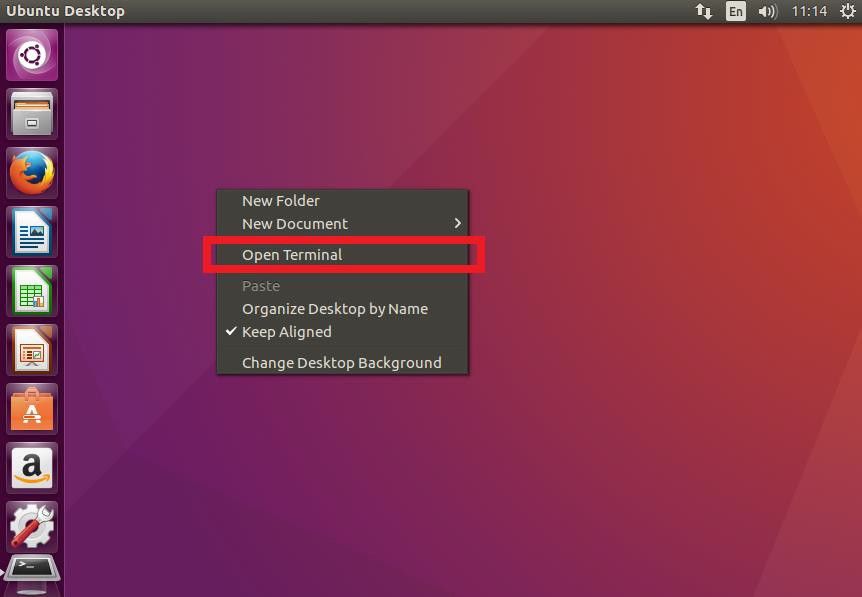
据不靠谱测试，图中上交的服务器会比较快（阿里云的也不错）。然后点 Choose

Server。当然你也可以 Select Best Server（大概率测出来是上交的）。

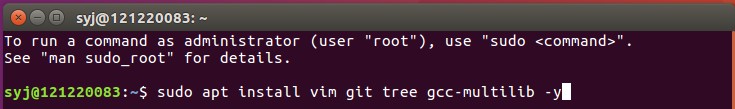
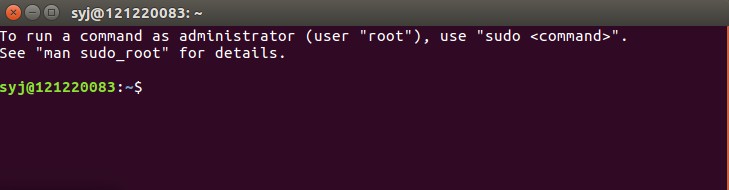


输密码认证。然后关闭 Software & Updates 窗口，然后会自动跳出来下面的窗口：

#### 点击 Reload 等待软件源更新完成后关闭应用商店。

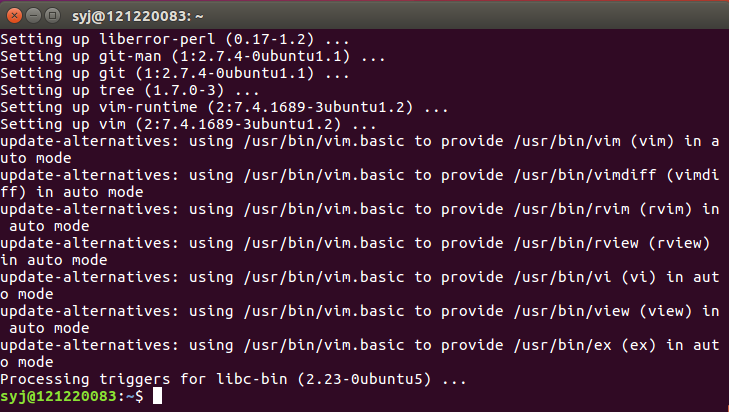
2、安装必要软件

桌面右键或者 **Ctrl+Alt+T 打开终端**。



使用键盘输入命令 s**udo apt install vim git tree gcc-multilib -y** 然后回车坐等安装完成。**系统会让你输入密码，密码是不可见的，只管自己输入然后回车就可以了。**

这条命令会安装三个工具，文本编辑器 vim，最流行的版本管理工具 git，以及助教推荐的 tree。此外还有工具包 gcc-multilib。



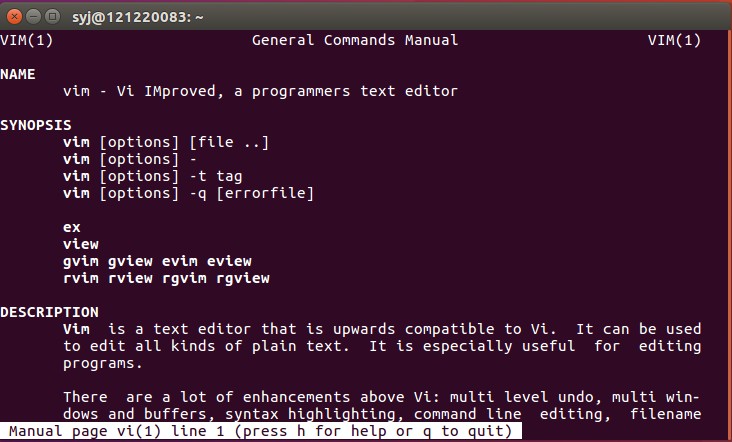
等待软件安装配置完毕。

## Linux 常用命令和工具

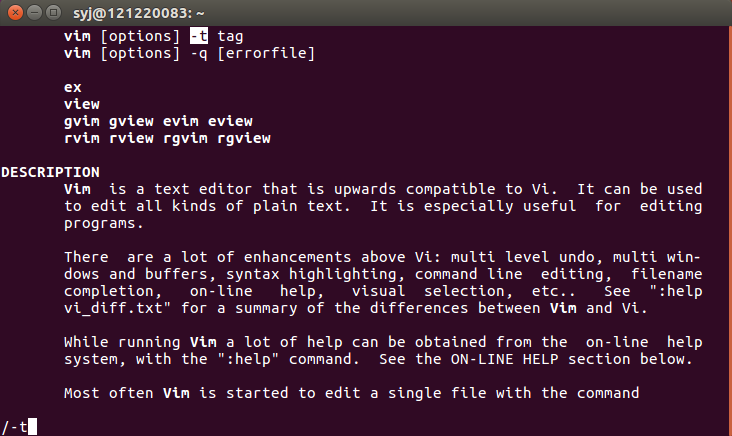
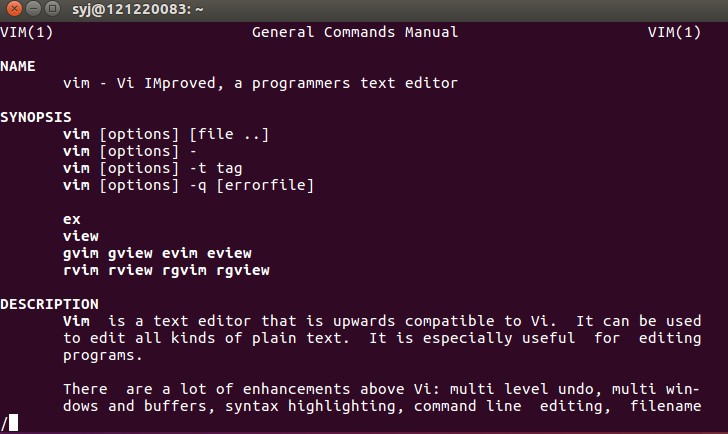
## 查看手册命令

**man <cmd>** 如果对某个命令有任何不明白的，都可以在终端敲入 man <命令名>，然后回车查看命令说明。例如：man vi





按键盘的上下键或者翻页键可以翻页，推荐使用关键词来查手册。 **使用关键词的方法是，在这个界面按除号，然后输入关键词后回车：**



#### 查找不一定正确，通过字母键 B 和 N 可以在搜索结果中上下跳转。按字母 Q 可以退出 man。

## 文件和文件夹操作命令

**常用的文件和文件夹操作命令有**：pwd, ls, mkdir, cd, touch, cp, rm, find, mv

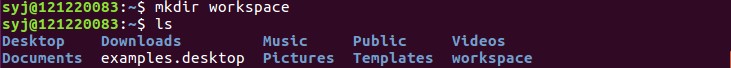
**1、pwd** 查看当前目录

**2、ls** 列出当前目录下的所有文件（不包括隐藏文件）；

ls -l 列出当前目录下的所有文件（不含隐藏文件）并显示查看权限；

ls -a 列出当前目录下的所有文件，包括隐藏文件（简化为 la）；

ll 列出当前目录下的所有文件，含隐藏文件并显示查看权限，Linux 系统下的隐藏文件泛指以点号开头的文件。

**3、mkdir** <dir name> 创建一个目录（文件夹）

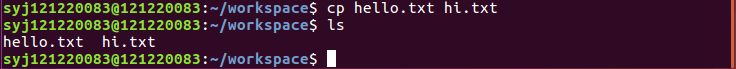
**4、cd** <dir> 进入某个目录，跟 windows 下相同



**5、touch** <file name> 创建一个文件

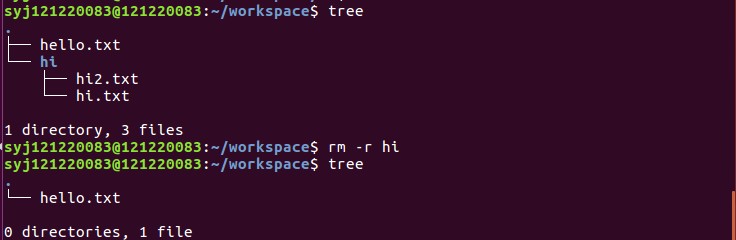


#### （请忽略下图中用户名不一样的问题，不影响命令说明）

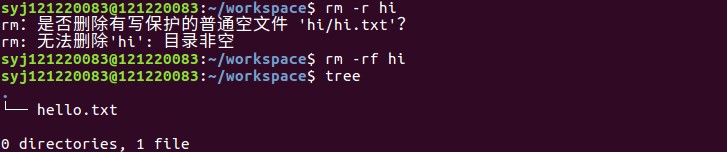
6、cp <filenameA> <filenameB> 将文件 A 复制为文件 B

**7、rm** <filename> 移除文件

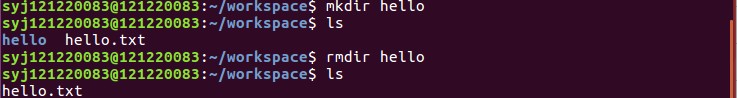


**rm** -r <dir name> 移除目录并递归删除目录内文件（如果有写保护文件，删除失败）

rm -rf <dir/file name> 移除写保护文件，并递归删除目录（包括写保护文件）



rmdir <dir name> 移除空白文件夹



**8、find** 命令一般用于查找文件，find [path] [options] [expression]

path 指定目录路径，系统从这里开始沿着目录树向下查找文件。它是一个路径列表， 相互用空格分离，如果不写 path，那么默认为当前目录。

主要参数

[options]参数：

－depth：使用深度级别的查找过程方式，在某层指定目录中优先查找文件内容。

－maxdepth levels：表示至多查找到开始目录的第 level 层子目录。level 是一个非负数，如果 level 是 0 的话表示仅在当前目录中查找。

－mindepth levels：表示至少查找到开始目录的第 level 层子目录。

－mount：不在其它文件系统（如 Msdos、Vfat 等）的目录和文件中查找。

－version：打印版本。

[expression]是匹配表达式，是 find 命令接受的表达式，find 命令的所有操作都是针对表达式的。它的参数非常多，这里只介绍一些常用的参数。

—name：支持统配符\*和?。

－atime n：搜索在过去 n 天读取过的文件。

－ctime n：搜索在过去 n 天修改过的文件。

－group grpoupname：搜索所有组为 grpoupname 的文件。

－user 用户名：搜索所有文件属主为用户名（ID 或名称）的文件。

－size n：搜索文件大小是 n 个 block 的文件。

－print：输出搜索结果，并且打印。

find 命令查找文件的几种方法：

（1）根据文件名查找

例如，我们想要查找一个文件名是 lilo.conf 的文件，可以使用如下命令：

find / －name lilo.conf

find 命令后的“/”表示搜索整个根目录。

（2）快速查找文件

根据文件名查找文件要花费相当长的一段时间，特别是大型 Linux 文件系统和大容量硬盘文件放在很深的子目录中时。如果我们知道了这个文件存放在某个目录中，那么只要在这个目录中往下寻找就能节省很多时间。比如 smb.conf 文件，从它的文件后缀“.conf”可以判断这是一个配置文件，那么它应该在/etc 目录内，此时可以使用下面命令：

find /etc －name smb.conf

这样，使用“快速查找文件”方式可以缩短时间。

（3）根据部分文件名查找方法

有时我们知道只某个文件包含有 abvd 这 4 个字，那么要查找系统中所有包含有这 4 个字符的文件可以输入下面命令：

find / －name '\*abvd\*'

输入这个命令以后，Linux 系统会将在/目录中查找所有的包含有 abvd 这 4 个字符的文件（其中\*是通配符），比如 abvdrmyz 等符合条件的文件都能显示出来。

（4）使用混合查找方式查找文件

find 命令可以使用混合查找的方法，例如，我们想在/etc 目录中查找大于 500000 字

节，并且在 24 小时内修改的某个文件，则可以使用-and (与)把两个查找参数链接起来组合成一个混合的查找方式。

find /etc -size +500000c -and -mtime +1

mv 命令

mv 命令用来为文件或目录改名，或者将文件由一个目录移入另一个目录中，它的使用权限是所有用户。该命令如同 Dos 命令中的 ren 和 move 的组合。

mv[options] 源文件或目录 目标文件或目录

[options]主要参数

－i：交互方式操作。如果 mv 操作将导致对已存在的目标文件的覆盖，此时系统询问是否重写，要求用户回答“y”或“n”，这样可以避免误覆盖文件。

－f：禁止交互操作。mv 操作要覆盖某个已有的目标文件时不给任何指示，指定此参数后 i 参数将不再起作用。

应用实例

（1）将/usr/cbu 中的所有文件移到当前目录（用“.”表示）中：

$ mv /usr/cbu/ \* .

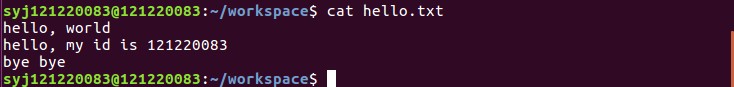
（2）将文件 cjh.txt 重命名为 wjz.txt：

$ mv cjh.txt wjz.txt

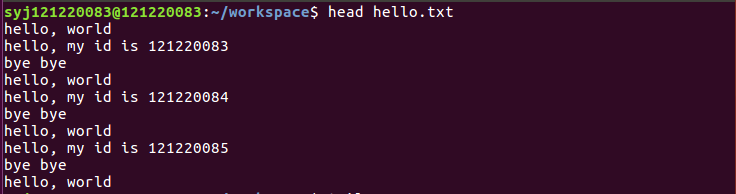
## 显示文件内容命令

常用显示文件内容的命令有：cat、head、tail、more、grep

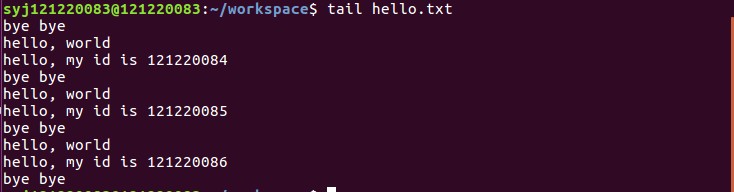
1、cat <file name> 在命令行里显示文件内容



2、head <file name> 在命令行里显示文件的前 10 行



3、tail <file name> 在命令行里显示文件的最后 10 行



4、more <file name> 在命令行里按页显示文件内容（使用 b 和空格键上下翻页， 按 q

退出）

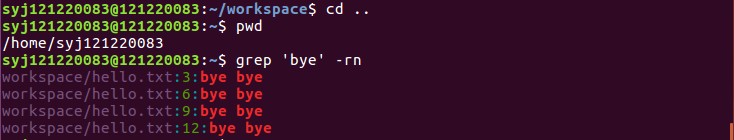
more +<number> <filename> 在命令行里按页显示文件第<number>行之后的内容

more -<number> <filename> 在命令行里按页现实文件内容，其中每一页包含<number>

行

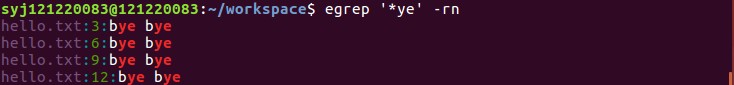
（跟在命令后的字符称为参数，一般命令内置参数是可以叠加使用的，例如上面的命令）

5、grep ‘<target>’ -rn 递归查找<target>出现在文件内的行数其中-r 代表递归查找，-n 代表同时显示行数



如果你了解正则表达式，那么可以使用 egrep 结合正则表达式进行更精准更强大的查找：

（下图，递归查找当前目录下，以 ye 结尾的单词，并显示行数）



Linux 支持 pipeline 式的命令，例如我们使用 wc 命令在 grep 的结果上进行计数：（两个命令之间使用竖线|，即按位或符号，进行分隔）



## 查看更改权限命令

#### 上文的 ls –l 命令输出的结果，每一列分别是这个意思：

总量 4：

#### 文件总大小为 4KB

下面每一列：

**权限/文件数/所属用户所在组/所属用户/文件大小（单位 B）/修改时间/文件名**其中第一列为文件权限：

第 1 位（一般只会见到前三项）：

“-”表示普通文件

“d”表示目录文件

“l”表示链接文件“c”表示字符设备

“b”表示块设备

“p”表示命名管道比如 FIFO 文件（First In First Out，先进先出） “f”表示堆栈文件比如 LIFO 文件（Last In First Out，后进先出）

“s”表示套接字

第 2，3，4 位代表文件拥有者权限（r/w/x 分别代表读/写/执行） 第 5，6，7 位代表当前用户组权限

第 8，9，10 位代表其他用户权限

chmod +x <file name> 给文件加上当前用户的执行权限

chmod 777 <filename> 给文件设置为所有用户均可读写执行（程序员害怕乘坐波音

777 的由来），因为一个 7 解析为二进制是 111，所以 777 就对应全 1，也就是-

rwxrwxrwx，即打开文件的所有权限。类似的大家可以算一下上面的 hello.txt 对应的权限数字是多少。

## 查看进程和杀死进程命令

#### 1、ps 命令

ps 命令用于报告当前系统的进程状态。可以搭配kill 指令随时中断、删除不必要的程序。

ps 命令是最基本同时也是非常强大的进程查看命令，使用该命令可以确定有哪些进程正在运行和运行的状态、进程是否结束、进程有没有僵死、哪些进程占用了过多的资源等等，总之大部分信息都是可以通过执行该命令得到的。

一般只会用到下面这个选项：

-A：显示所有程序。

**2、kill 命令:**用来中止一个进程。命令格式：

kill [ －s signal | －p ] [ －a ] pid ... kill －l [ signal ]

－s：指定发送的信号。

－p：模拟发送信号。

－l：指定信号的名称列表。

pid：要中止进程的 ID 号。

Signal：表示信号。

例如： kill -2 <pid>，让进程号为 pid 的进程正常退出。

kill -9 <pid>， 让进程号为 pid 的进程强制退出（一般配合 sudo 来用）。

# 第 2 章 Linux 下编程调试工具

* 1. **源代码版本控制工具** git

当我们在开发一个比较大的项目时，通常需要进行代码版本控制。现在最流行的版本控制工具就是 Git。

Git 的教程有很多，其中比较推荐看的是廖雪峰的教程**（请大家将教程中 Git 设置这一**

#### 节里的 name 设置为自己的学号 git config --global user.name “<学号>”），网址是

[http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000](http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/)

[/](http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/)，别名：两小时精通 Git**（请大家将教程中 Git 设置这一节里的 name 设置为自己的学号）**。

学会 Git 的使用方法**（请大家将教程中 Git 设置这一节里的 name 设置为自己的学号）** 之后，大家可以试着从~~世界上最大的程序员交友网站，~~GitHub 上，用下面的命令 clone 一份代码：

mkdir ~/workspace cd ~/workspace

git clone git@github.com:amix/vimrc.git

然后使用刚刚学会的 cp 命令，将 vimrc 项目里的文件复制到主目录（例如我的主目录是/home/syj，当 cd 命令后不加任何参数，敲回车后进入的就是主目录，主目录一般使用~ 符号表示，所以用 cd ~命令也可以进入主目录，用 ls ~命令可以列出主目录下的所有文件） 下。

cp ~/workspace/vimrc/vimrcs/basic.vim ~/.vimrc

（注意不要忘记点号和空格）



附部分常用的 Git 命令说明： **一、新建代码库**

# 在当前目录新建一个 Git 代码库

$ git init

# 新建一个目录，将其初始化为 Git 代码库

$ git init [project-name]

# 下载一个项目和它的整个代码历史

$ git clone [url]

#### 二、配置

Git 的设置文件为.gitconfig，它可以在用户主目录下（全局配置），也可以在项目目录下（项目配置）。

# 显示当前的 Git 配置

$ git config --list

# 编辑 Git 配置文件

$ git config -e [--global]

# 设置提交代码时的用户信息

$ git config [--global] user.name "[name]"

$ git config [--global] user.email "[email address]"

#### 三、增加/删除文件

# 添加指定文件到暂存区

$ git add [file1] [file2] ...

# 添加指定目录到暂存区，包括子目录

$ git add [dir]

# 添加当前目录的所有文件到暂存区

$ git add .

# 添加每个变化前，都会要求确认

# 对于同一个文件的多处变化，可以实现分次提交

$ git add -p

# 删除工作区文件，并且将这次删除放入暂存区

$ git rm [file1] [file2] ...

# 停止追踪指定文件，但该文件会保留在工作区

$ git rm --cached [file]

# 改名文件，并且将这个改名放入暂存区

$ git mv [file-original] [file-renamed]

#### 四、代码提交

# 提交暂存区到仓库区

$ git commit -m [message]

# 提交暂存区的指定文件到仓库区

$ git commit [file1] [file2] ... -m [message]

# 提交工作区自上次 commit 之后的变化，直接到仓库区

$ git commit -a

# 提交时显示所有 diff 信息

$ git commit -v

# 使用一次新的 commit，替代上一次提交

# 如果代码没有任何新变化，则用来改写上一次 commit 的提交信息

$ git commit --amend -m [message]

# 重做上一次 commit，并包括指定文件的新变化

$ git commit --amend [file1] [file2] ...

#### 五、分支

# 列出所有本地分支

$ git branch

# 列出所有远程分支

$ git branch -r

# 列出所有本地分支和远程分支

$ git branch -a

# 新建一个分支，但依然停留在当前分支

$ git branch [branch-name]

# 新建一个分支，并切换到该分支

$ git checkout -b [branch]

# 新建一个分支，指向指定 commit

$ git branch [branch] [commit]

# 新建一个分支，与指定的远程分支建立追踪关系

$ git branch --track [branch] [remote-branch]

# 切换到指定分支，并更新工作区

$ git checkout [branch-name]

# 切换到上一个分支

$ git checkout -

# 建立追踪关系，在现有分支与指定的远程分支之间

$ git branch --set-upstream [branch] [remote-branch]

# 合并指定分支到当前分支

$ git merge [branch]

# 选择一个 commit，合并进当前分支

$ git cherry-pick [commit]

# 删除分支

$ git branch -d [branch-name]

# 删除远程分支

$ git push origin --delete [branch-name]

$ git branch -dr [remote/branch]

#### 六、标签

# 列出所有 tag

$ git tag

# 新建一个 tag 在当前 commit

$ git tag [tag]

# 新建一个 tag 在指定 commit

$ git tag [tag] [commit]

# 删除本地 tag

$ git tag -d [tag]

# 删除远程 tag

$ git push origin :refs/tags/[tagName]

# 查看 tag 信息

$ git show [tag]

# 提交指定 tag

$ git push [remote] [tag]

# 提交所有 tag

$ git push [remote] --tags

# 新建一个分支，指向某个 tag

$ git checkout -b [branch] [tag]

#### 七、查看信息

# 显示有变更的文件

$ git status

# 显示当前分支的版本历史

$ git log

# 显示 commit 历史，以及每次 commit 发生变更的文件

$ git log --stat

# 搜索提交历史，根据关键词

$ git log -S [keyword]

# 显示某个 commit 之后的所有变动，每个 commit 占据一行

$ git log [tag] HEAD --pretty=format:%s

# 显示某个 commit 之后的所有变动，其"提交说明"必须符合搜索条件

$ git log [tag] HEAD --grep feature

# 显示某个文件的版本历史，包括文件改名

$ git log --follow [file]

$ git whatchanged [file]

# 显示指定文件相关的每一次 diff

$ git log -p [file]

# 显示过去 5 次提交

$ git log -5 --pretty --oneline

# 显示所有提交过的用户，按提交次数排序

$ git shortlog -sn

# 显示指定文件是什么人在什么时间修改过

$ git blame [file]

# 显示暂存区和工作区的差异

$ git diff

# 显示暂存区和上一个 commit 的差异

$ git diff --cached [file]

# 显示工作区与当前分支最新 commit 之间的差异

$ git diff HEAD

# 显示两次提交之间的差异

$ git diff [first-branch]...[second-branch]

# 显示今天你写了多少行代码

$ git diff --shortstat "@{0 day ago}"

# 显示某次提交的元数据和内容变化

$ git show [commit]

# 显示某次提交发生变化的文件

$ git show --name-only [commit]

# 显示某次提交时，某个文件的内容

$ git show [commit]:[filename]

# 显示当前分支的最近几次提交

$ git reflog

#### 八、远程同步

# 下载远程仓库的所有变动

$ git fetch [remote]

# 显示所有远程仓库

$ git remote -v

# 显示某个远程仓库的信息

$ git remote show [remote]

# 增加一个新的远程仓库，并命名

$ git remote add [shortname] [url]

# 取回远程仓库的变化，并与本地分支合并

$ git pull [remote] [branch]

# 上传本地指定分支到远程仓库

$ git push [remote] [branch]

# 强行推送当前分支到远程仓库，即使有冲突

$ git push [remote] --force

# 推送所有分支到远程仓库

$ git push [remote] --all

#### 九、撤销

# 恢复暂存区的指定文件到工作区

$ git checkout [file]

# 恢复某个 commit 的指定文件到暂存区和工作区

$ git checkout [commit] [file]

# 恢复暂存区的所有文件到工作区

$ git checkout .

# 重置暂存区的指定文件，与上一次 commit 保持一致，但工作区不变

$ git reset [file]

# 重置暂存区与工作区，与上一次 commit 保持一致

$ git reset --hard

# 重置当前分支的指针为指定 commit，同时重置暂存区，但工作区不变

$ git reset [commit]

# 重置当前分支的 HEAD 为指定 commit，同时重置暂存区和工作区，与指定 commit

一致

$ git reset --hard [commit]

# 重置当前 HEAD 为指定 commit，但保持暂存区和工作区不变

$ git reset --keep [commit]

# 新建一个 commit，用来撤销指定 commit

# 后者的所有变化都将被前者抵消，并且应用到当前分支

$ git revert [commit]

# 暂时将未提交的变化移除，稍后再移入

$ git stash

$ git stash pop

#### 十、其他

# 生成一个可供发布的压缩包

$ git archive

* 1. **文本编辑器** vi/vim

（一）vi/vim

Linux 世界几乎所有的配置文件都是以纯文本形式存在的，而在所有的 Linux 发行版系统上都有 vi 编辑器，因此利用简单的文字编辑软件就能够轻松地修改系统的各种配置了， 非常方便。vi 就是一种功能强大的文本编辑器，而 vim 则是高级版的 vi，不但可以用不同颜色显示文字内容，还能进行诸如 shell 脚本、C 语言程序编辑等功能，可以作为程序编辑器。

Vim 编辑器是习惯在 Linux 下编程的程序员中最流行的编辑器，如果嫌 Vim 使用不便

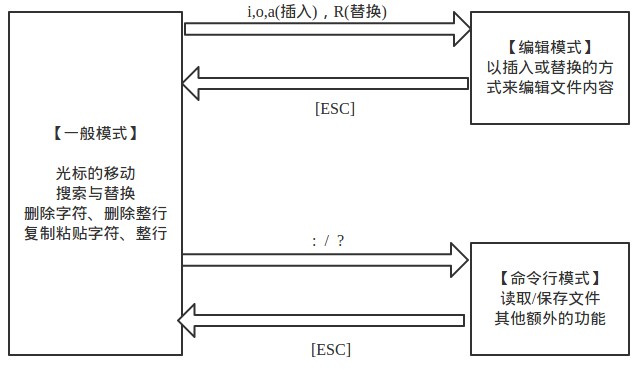
（主要是命令太多），那么也可以选用 gedit，gedit 是类似 windows 系统下的 notepad++的文本编辑器，图形界面，在远程连接时不能使用。助教推荐大家通过一个有趣的打字游戏来学习使用 Vim：<http://www.openvim.com/>

刚才我们把一个.vimrc 文件文件放到了主目录下，这其实是在对 vim 进行配置。我们可以把显示行号，设置 Tab 等于 4 个空格，设置使用空格来代替 Tab，显示行号等等的命令全都写在一个.vimrc 文件里，然后将这个文件放在主目录下，这样启动 vim 的时候，它就会自动读取用户的配置文件进行配置，而不需要用户每一次使用 vim 都先敲一堆配置命令了。

（二）为什么要学习 vi/vim？

首先所有的 Linux 发行版系统上都会默认内置 vi 编辑器，而不一定带有其他文本编辑器，非常通用；其次，很多软件的编辑接口都会默认调用 vi；第三，vi 具有程序编辑的能力；最后，vi 程序简单，编辑速度相当快速。

（三）vi 的三种模式及各个模式之间的转换关系



（四）一般模式常用操作

【h(或向左方向键)】 光标左移一个字符

【j(或向下方向键)】 光标下移一个字符

【k(或向上方向键)】 光标上移一个字符

【l(或向右方向键)】 光标右移一个字符

【[Ctrl] + f】 屏幕向下移动一页（相当于 Page Down 键）

【[Ctrl] + b】 屏幕向上移动一页（相当于 Page Up 键）

【[0]或[Home]】 光标移动到当前行的最前面

【[$]或[End]】 光标移动到当前行的末尾

【G】 光标移动到文件的最后一行（第一个字符处）

【nG】 n 为数字（下同），移动到当前文件中第 n 行

【gg】 移动到文件的第一行，相当于"1G"

【n[Enter]】 光标向下移动 n 行

【/word】 在文件中查找内容为 word 的字符串（向下查找）

【?word】 在文件中查找内容为 word 的字符串（向上查找）

【[n]】 表示重复查找动作，即查找下一个

【[N]】 反向查找下一个

【:n1,n2s/word1/word2/g】 n1、n2 为数字，在第 n1 行到第 n2 行之间查找 word1 字符串，并将其替换成 word2

【:1,$s/word1/word2/g】 从第一行（第 n 行同理）到最后一行查找 word1 注册，并将其替换成 word2

【:1,$s/word1/word2/gc】 功能同上，只不过每次替换时都会让用户确认

【x,X】 x 为向后删除一个字符，相当于[Delete]，X 为向前删除一个字符，相当于

[Backspace]

【dd】 删除光标所在的一整行

【ndd】 删除光标所在的向下 n 行

【yy】 复制光标所在的那一行

【nyy】 复制光标所在的向下 n 行

【p,P】 p 为将已经复制的数据在光标下一行粘贴；P 为将已经复制的数据在光标上一行粘贴

【u】 撤消上一个操作

【[Ctrl] + r】 多次撤消

【.】 这是小数点键，重复上一个操作

（五）一般模式切换到编辑模式的操作

１、进入插入模式（6 个命令）

【i】 从目前光标所在处插入

【I】 从目前光标

【a】 从当前光标所在的下一个字符处开始插入

【A】 从光标所在行的最后一个字符处开始插入

【o】 英文小写字母 o，在目前光标所在行的下一行处插入新的一行并开始插入

【O】 英文大写字母 O，在目前光标所在行的上一行处插入新的一行并开始插入

2、进入替换模式（2 个命令）

【r】 只会替换光标所在的那一个字符一次

【R】 会一直替换光标所在字符，直到按下[ESC]键为止

【[ESC]】 退出编辑模式回到一般模式

（六）一般模式切换到命令行模式

【:w】 保存文件

【:w!】 若文件为只读，强制保存文件

【:q】 离开 vi

【:q!】 不保存强制离开 vi

【:wq】 保存后离开

【:wq!】 强制保存后离开

【:! command】 暂时离开 vi 到命令行下执行一个命令后的显示结果

【:set nu】 显示行号

【:set nonu】 取消显示行号

【:w newfile】 另存为

（七）文件恢复模式

【[O]pen Read-Only】 以只读方式打开文件

【[E]dit anyway】 用正常方式打开文件，不会载入暂存文件内容

【[R]ecover】 加载暂存文件内容

【[D]elete it】 用正常方式打开文件并删除暂存文件

【[Q]uit】 按下 q 就离开 vi，不进行其他操作

【[A]bort】 与 quit 功能类似

（八）块选择（一般模式下用）

【v,V】 v:将光标经过的地方反白选择；V：将光标经过的行反白选择

【[Ctrl] + v】 块选择，可用长方形的方式选择文本

【y】 将反白的地方复制到剪贴板

【d】 将反白的内容删除

（九）多文件编辑

【vim file1 file2】 同时打开两个文件

【:n】 编辑下一个文件

【:N】 编辑上一个文件

【:files】 列出当前用 vim 打开的所有文件

（十）多窗口功能

【:sp [filename]】 打开一个新窗口，显示新文件，若只输入:sp，则两窗口显示同一个文件

【[Ctrl] + w + j】 光标移动到下方窗口

【[Ctrl] + w + k】 光标移动到上方窗口

【[Ctrl] + w + q】 离开当前窗口

* 1. **通用编译和链接工具** gcc

大家学习 C++的时候有用过 g++来编译过程序吗？gcc 是差不多的工具。g++用来编译

C++程序，而 gcc 用来编译 C 程序。**基本用法：**

#### gcc hello.c –o hello

编译并链接 hello.c 文件，如果成功，将输出名为 hello 的二进制文件。

#### gcc –c hello.c

编译 hello.c 文件但不链接。输出同名的.o 文件。

（在 64 位系统下，加选项-m32 可以将程序编译为 32 位，不加选项则默认编译为 64

位。例如 gcc -m32 hello.c -o hello， gcc -m32 -c hello.c）

现在的 gcc 已经完美集成了链接工具 ld，因此下面的仅做说明用。

~~ld -m elf\_i386 -o hello --dynamic-linker /lib/ld-linker.so.2 /usr/lib32/crt1.o /usr/lib32/crti.o~~

~~/usr/lib32/ctrn.o hello.o -lc~~

~~链接工具使用示范，前面的选项-m elf\_i386 表示链接 32 位程序。~~

如果 64 位系统要链接 32 位程序，请使用 gcc 命令：

gcc -m32 -o hello hello.o

如果要链接 64 位程序，请使用 gcc 命令：

gcc -o hello hello.o

附部分常用 gcc 命令中文版说明： 语法

gcc(选项)(参数)

选项：

-o：指定生成的输出文件；

-E：仅执行编译预处理；

-S：将 C 代码转换为汇编代码；

-wall：显示警告信息；

#### -m32：可以将程序编译为 32 位

-c：仅执行编译操作，不进行连接操作。参数：C 源文件，指定 C 语言源代码文件。实例：

无选项编译链接 C 源程序 test.c： gcc test.c

将 test.c 预处理、汇编、编译并链接形成可执行文件。这里未指定输出文件，默认输

出为 a.out。

选项 -o

gcc test.c -o test

将 test.c 预处理、汇编、编译并链接形成可执行文件 test。-o 选项用来指定输出文件的文件名。

选项 -E

gcc -E test.c -o test.i

将 test.c 预处理输出 test.i 文件。选项 -S

gcc -S test.i

将预处理输出文件 test.i 汇编成 test.s 文件。选项 -c

gcc -c test.s

将汇编输出文件 test.s 编译输出 test.o 文件。无选项链接

gcc test.o -o test

将编译输出文件 test.o 链接成最终可执行文件 test。选项 -O

gcc -O1 test.c -o test

使用编译优化级别 1 编译程序。级别为 1~3，级别越大优化效果越好，但编译时间越长。

多源文件的编译方法

如果有多个源文件，基本上有两种编译方法： 假设有两个源文件为 test.c 和 testfun.c

多个文件一起编译

gcc testfun.c test.c -o test

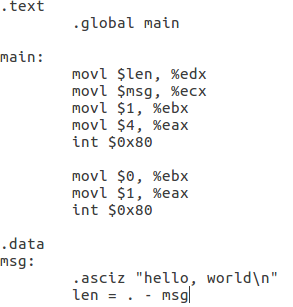
将 testfun.c 和 test.c 分别编译后链接成 test 可执行文件。分别编译各个源文件，之后对编译后输出的目标文件链接。gcc -c testfun.c #将 testfun.c 编译成 testfun.o

gcc -c test.c #将 test.c 编译成 test.o

gcc -o testfun.o test.o -o test #将 testfun.o 和 test.o 链接成 test

以上两种方法相比较，第一中方法编译时需要所有文件重新编译，而第二种方法可以只重新编译修改的文件，未修改的文件不用重新编译。

* 1. **汇编编译工具** as

先编写一个汇编程序：

然后使用 as 工具进行编译：

as -o hello.o hello.s

接着使用 gcc 进行链接：

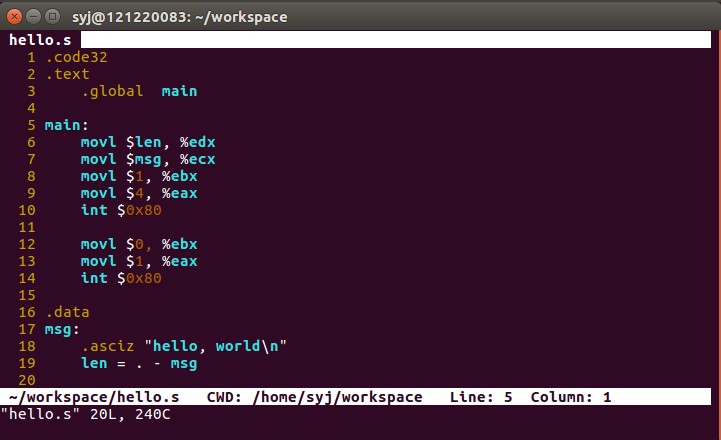
gcc -o hello hello.o

最后尝试运行：

./hello 

以下是 64 位系统运行 32 位程序的方法，**本门课使用 32 位系统**，以下选读

先写一个 32 位的汇编程序：



然后使用 as 工具进行编译：

as --32 -o hello.o hello.s

接着使用 gcc 进行链接：

gcc -m32 -o hello hello.o

最后尝试运行：



如果需要编写 64 位程序，只要在代码里去掉.code32，编译和链接时分别去掉--32 和-

m32 选项即可。

附 as 工具的中文命令指南：

as 命令是 GNU 组织推出的一款汇编语言编译器，它支持多种不同类型的处理器。语法

as(选项)(参数)

选项

-ac：忽略失败条件；

-ad：忽略调试指令；

-ah：包括高级源；

-al：包括装配；

-am：包括宏扩展；

-an：忽略形式处理；

-as：包括符号；

=file：设置列出文件的名字；

--alternate：以交互宏模式开始；

-f：跳过空白和注释预处理；

-g：产生调试信息；

-J：对于有符号溢出不显示警告信息；

-L：在符号表中保留本地符号；

-o：指定要生成的目标文件；

--statistics：打印汇编所用的最大空间和总时间。参数

汇编文件：指定要汇编的源文件。

* 1. **符号调试器** gdb

如果需要使用 gdb 来调试程序，需要在编译阶段加上选项-g 代表允许使用 gdb 进行调试。

gcc -g -o hello hello.c

五分钟学会 gdb 教程（推荐）： https:[//www](http://www.cs.umd.edu/%7Esrhuang/teaching/cmsc212/gdb-tutorial-handout.pdf).c[s.umd.edu/~srhuang/teaching/cmsc212/gdb-tutorial-handout.pdf](http://www.cs.umd.edu/%7Esrhuang/teaching/cmsc212/gdb-tutorial-handout.pdf) 十分钟学会 gdb 教程：

[https://www.cs.cmu.edu/~gilpin/tutorial/](https://www.cs.cmu.edu/%7Egilpin/tutorial/)

附中文版 gdb 部分命令说明：

file <文件名>

加载被调试的可执行程序文件。 因为一般都在被调试程序所在目录下执行 GDB，因而文本名不需要带路径。

(gdb) file gdb-sample r

Run 的简写，运行被调试的程序。 如果此前没有下过断点，则执行完整个程序；如果

有断点，则程序暂停在第一个可用断点处。

(gdb) r c

Continue 的简写，继续执行被调试程序，直至下一个断点或程序结束。

(gdb) c

b <行号>

b <函数名称> b \*<函数名称> b \*<代码地址> d [编号]

b: Breakpoint 的简写，设置断点。可以使用“行号”“函数名称”“执行地址”等方式指定断点位置。 其中在函数名称前面加“\*”符号表示将断点设置在“由编译器生成的

prolog 代码处”。如果不了解汇编，可以不予理会此用法。

d: Delete breakpoint 的简写，删除指定编号的某个断点，或删除所有断点。断点编号从 1 开始递增。

(gdb) b 8 (gdb) b main (gdb) b \*main

(gdb) b \*0x804835c

(gdb) d s, n

s: 执行一行源程序代码，如果此行代码中有函数调用，则进入该函数；

n: 执行一行源程序代码，此行代码中的函数调用也一并执行。

s 相当于其它调试器中的“Step Into (单步跟踪进入)”；

n 相当于其它调试器中的“Step Over (单步跟踪)”。

这两个命令必须在有源代码调试信息的情况下才可以使用（GCC 编译时使用“-g”参数）。

(gdb) s

(gdb) n si, ni

si 命令类似于 s 命令，ni 命令类似于 n 命令。所不同的是，这两个命令（si/ni）所针

对的是汇编指令，而 s/n 针对的是源代码。

(gdb) si (gdb) ni

p <变量名称>

Print 的简写，显示指定变量（临时变量或全局变量）的值。

(gdb) p i

(gdb) p nGlobalVar display ... undisplay <编号>

display，设置程序中断后欲显示的数据及其格式。 例如，如果希望每次程序中断后可

以看到即将被执行的下一条汇编指令，可以使用命令 “display /i $pc” 其中 $pc 代表当前汇编指令，/i 表示以十六进行显示。当需要关心汇编代码时，此命令相当有用。

undispaly，取消先前的 display 设置，编号从 1 开始递增。

(gdb) display /i $pc (gdb) undisplay 1

i

info 的简写，用于显示各类信息，详情请查阅“help i”。 (gdb) i r

q

Quit 的简写，退出 GDB 调试环境。

(gdb) q

help [命令名称]

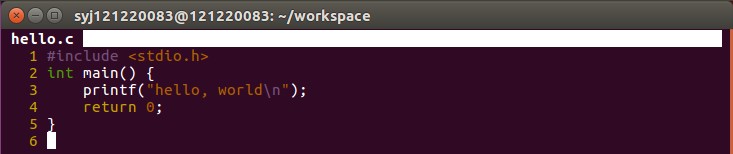
GDB 帮助命令，提供对 GDB 名种命令的解释说明。 如果指定了“命令名称”参数， 则显示该命令的详细说明；如果没有指定参数，则分类显示所有 GDB 命令，供用户进一步浏览和查询。维护工具 make

* 1. **维护工具** make

神器，可以大大提高非 IDE 环境下的编程效率。

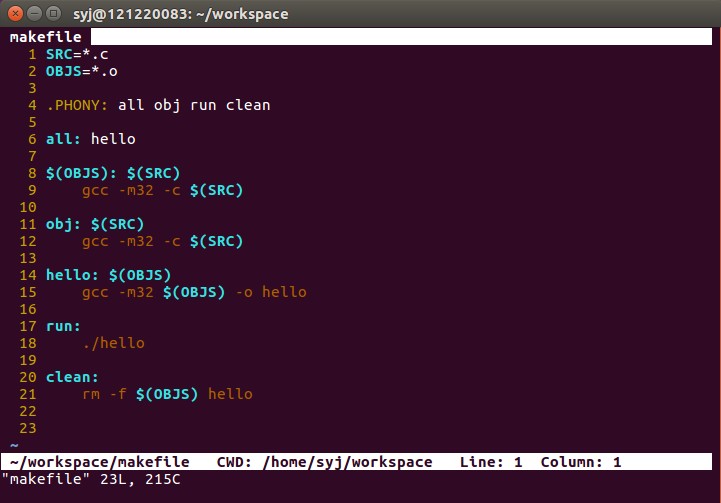
五分钟学会 makefile 教程：[http://www.cs.cmu.edu/~tom7/211/make.html](http://www.cs.cmu.edu/%7Etom7/211/make.html) 下面举个例子来说明它的用法：

准备工作：

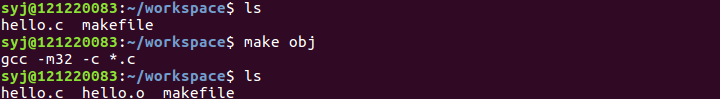
一个编译无误的 hello.c 文件

在这个文件同一个目录下创建一个文件 makefile

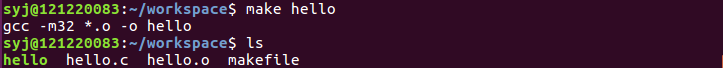


打开 makefile 文件然后输入下面内容：

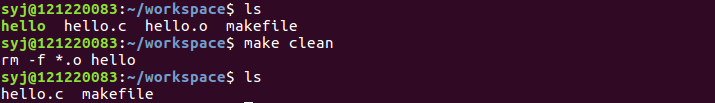
可以看到 obj 这条命令和$(OBJS)是完全重复的，放在这只是为了更便捷地进行编译。接下来看如何使用 makefile：

保存退出，现在我们编译（而不链接）只需要 make obj：

链接只需要 make hello：



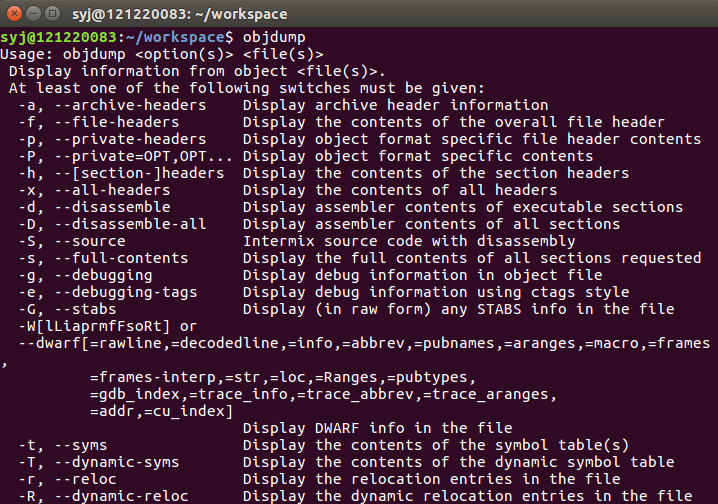
运行只需要 make run:

删除所有生成的文件只需要 make clean：

就本课程的实验来说，这个 makefile 例子里罗列出的功能已经足够使用了，当然

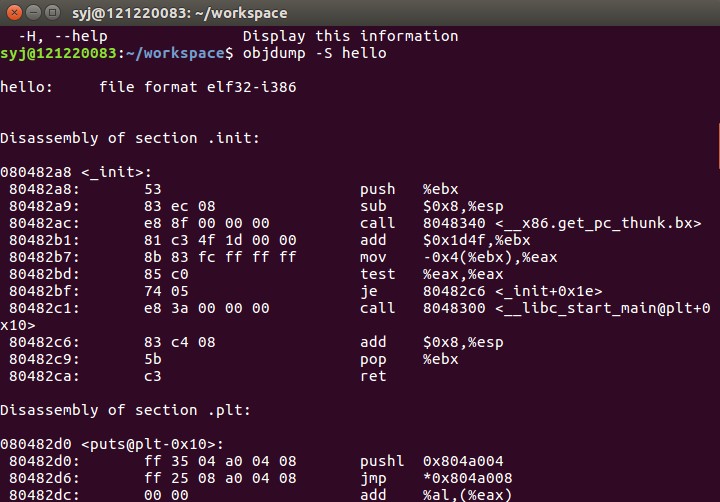
makefile 还有更高级用法，有兴趣的同学可以自己阅读文档或者直接上 GitHub 找相关代码。

* 1. **反汇编工具** objdump

命令行输入 objdump 就可以看到所有的参数。

我们可以通过-S（大写）参数查看一个二进制文件的汇编代码：

objdump -s hello



附 objdump 各选项说明：

--archive-headers

-a

显示档案库的成员信息,类似 ls -l 将 lib\*.a 的信息列出。

-b bfdname

--target=bfdname

指定目标码格式。这不是必须的，objdump 能自动识别许多格式，比如：

objdump -b oasys -m vax -h fu.o

显示 fu.o 的头部摘要信息，明确指出该文件是 Vax 系统下用 Oasys 编译器生成的目标文件。objdump -i 将给出这里可以指定的目标码格式列表。

-C

--demangle

将底层的符号名解码成用户级名字，除了去掉所开头的下划线之外，还使得 C++函数名以可理解的方式显示出来。

--debugging

-g

显示调试信息。企图解析保存在文件中的调试信息并以 C 语言的语法显示出来。仅仅支持某些类型的调试信息。有些其他的格式被 readelf -w 支持。

-e

--debugging-tags

类似-g 选项，但是生成的信息是和 ctags 工具相兼容的格式。

--disassemble

-d

从 objfile 中反汇编那些特定指令机器码的 section。

-D

--disassemble-all

与 -d 类似，但反汇编所有 section.

--prefix-addresses

反汇编的时候，显示每一行的完整地址。这是一种比较老的反汇编格式。

-EB

-EL

--endian={big|little}

指定目标文件的小端。这个项将影响反汇编出来的指令。在反汇编的文件没描述小端信息的时候用。例如 S-records.

-f

--file-headers

显示 objfile 中每个文件的整体头部摘要信息。

-h

--section-headers

--headers

显示目标文件各个 section 的头部摘要信息。

-H

--help

简短的帮助信息。

-i

--info

显示对于 -b 或者 -m 选项可用的架构和目标格式列表。

-j name

--section=name

仅仅显示指定名称为 name 的 section 的信息

-l

--line-numbers

用文件名和行号标注相应的目标代码，仅仅和-d、-D 或者-r 一起使用使用-ld 和使用-d 的区别不是很大，在源码级调试的时候有用，要求编译时使用了-g 之类的调试编译选项。

-m machine

--architecture=machine

指定反汇编目标文件时使用的架构，当待反汇编文件本身没描述架构信息的时候(比如

1. records)，这个选项很有用。可以用-i 选项列出这里能够指定的架构.

--reloc

-r

显示文件的重定位入口。如果和-d 或者-D 一起使用，重定位部分以反汇编后的格式显示出来。

--dynamic-reloc

-R

显示文件的动态重定位入口，仅仅对于动态目标文件意义，比如某些共享库。

-s

--full-contents

显示指定 section 的完整内容。默认所有的非空 section 都会被显示。

-S

--source

尽可能反汇编出源代码，尤其当编译的时候指定了-g 这种调试参数时，效果比较明显。隐含了-d 参数。

--show-raw-insn

反汇编的时候，显示每条汇编指令对应的机器码，如不指定--prefix-addresses，这将是缺省选项。

--no-show-raw-insn

反汇编时，不显示汇编指令的机器码，如不指定--prefix-addresses，这将是缺省选项。

--start-address=address

从指定地址开始显示数据，该选项影响-d、-r 和-s 选项的输出。

--stop-address=address

显示数据直到指定地址为止，该项影响-d、-r 和-s 选项的输出。

-t

--syms

显示文件的符号表入口。类似于 nm -s 提供的信息

-T

--dynamic-syms

显示文件的动态符号表入口，仅仅对动态目标文件意义，比如某些共享库。它显示的信息类似于 nm -D|--dynamic 显示的信息。

-V

--version

版本信息

--all-headers

-x

显示所可用的头信息，包括符号表、重定位入口。-x 等价于-a -f -h -r -t 同时指定。

-z

--disassemble-zeroes

一般反汇编输出将省略大块的零，该选项使得这些零块也被反汇编。

@file 可以将选项集中到一个文件中，然后使用这个@file 选项载入。

# 第 3 章 实验内容

## 实验 1：Linux 编程基础实验

一、实验目的

* 1. 学会自己安装 Linux 系统；
  2. 学会配置简单的 Linux 开发环境；
  3. 在 Linux 下完成简单编程练习并熟悉各种命令行工具的使用方法。二、实验内容

1. Linux 安装和配置
   1. 请分别按照 [Linux 安装](#_bookmark2)和 [Linux 配置](#_bookmark5)按步骤完成 VirtualBox 和 Linux 系统安装和环境配置。

要求 1：安装 32 位版本的 Ubuntu 并配置好所有工具。

* 1. 请参考 man 打印出 Vim/Git/GCC/AS/OBJDUMP/GDB 的版本然后截图贴在实验报告中。
  2. 写下你在安装过程中遇到的问题，并说明你是如何解决的（没有可不写）。

1. Linux 下的编程实践
   1. 注册并成为~~世界上最大的程序员交友网站，~~GitHub 的一员，然后通过上文的教程学会 Git 的用法。
   2. 使用文件管理器或者 mkdir 在主目录下创建工作目录 workspace。cd ~

mkdir workspace

* 1. 在 workspace 目录下创建目录 lab01，在 lab01 下创建目录 151220000（改成你自己的学号，后面的同理）。

cd workspace mkdir lab01 cd lab01

mkdir 151220000

* 1. 使用 git init 初始化~/workspace/lab01/151220000 为版本库，然后使用 git 记录你认为有意义的代码改动（推荐每实现一个功能进行一次提交）。

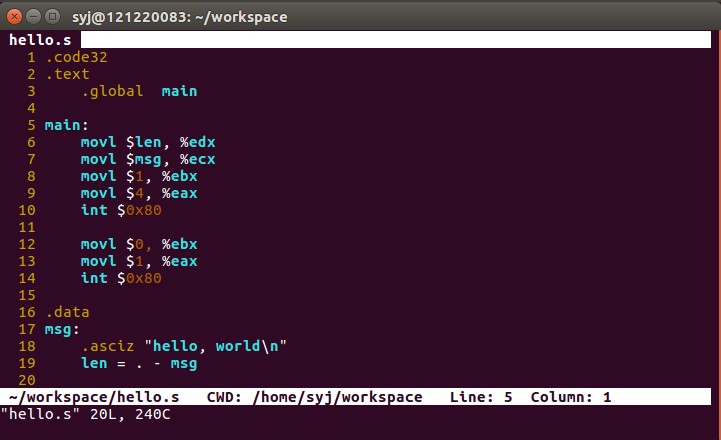
cd 151220000

git init

* 1. 学习使用.gitignore 将不需要进行版本追踪的文件（例如 pdf，word 等）从 Git

记录中排除。

* 1. 将工作目录切换为~/workspace/lab01/151220000，然后使用 Gedit 或 Vim 编辑器编写汇编语言的表白程序（可以直接修改下图中的代码让它打印出一个爱心），并保存为 heart.S（程序代码参考[**汇编编译工具 as**](#_bookmark16)，或下图）。



#### 要求：爱心形状如下，前后不要有多余的空格，爱心之后紧跟自己的学号，最后一行为空行（请严格遵守格式，助教的自动批改脚本非常不智能）。

touch heart.S # 创建文件

# 编辑 heart.S

as –o heart.o heart.S # 编译

gcc –o heart heart.o # 链接

./heart # 运行

TIPS：汇编的编译链接命令的详细介绍在教程[汇编编译工具 as](#_bookmark16) 中。

\* \*

\*\*\*\*\*\*\*

\*\*\*\*\*

\* 151220000

1. 熟悉工具：
   1. 使用 objdump 的-D 选项反汇编 heart.o 文件，找到你学号的位置并截图（提示：ASCII 码数字的 16 进制）。

objdump –D heart.o –o heart.s

#### 要求：在实验报告中贴出截图。

* 1. 通过修改~/.vimrc 文件，使你的 vim 支持显示行号。

set nu

#### 要求：请自行查找如何修改。

1. 数据的表示范围及不同类型的数据长度实验。

sqr.c

#include <stdio.h> int main()

{

int i,j; i=40000;

j=i\*i;

printf("The 40000\*40000 is %d\n", j); i=50000;

j=i\*i;

printf("The 50000\*50000 is %d\n", j); return 0;

}

# gcc –o sqr sqr.c

1. 将输出结果导出，说明发生这种现象的原因？
2. 寻找在该程序中保证结果正确的最大整数值？
3. 应如何修改程序，才能保证结果都正确？
4. 矩阵运算执行时间比较
5. 比较两个矩阵复制函数的执行时间。
6. 说明为什么会出现这个差别。代码：

matrix.c

#include <stdio.h>

#include <time.h>

void copyij(int src[2048][2048], int dst[2048][2048]){ int i, j;

for (i = 0; i < 2048; i++){ for (j = 0; j < 2048; j++){

dst[i][j] = src[i][j];

}

}

}

void copyji(int src[2048][2048], int dst[2048][2048]){ int i, j;

for (j = 0; j < 2048; j++){ for (i = 0; i < 2048; i++){

dst[i][j] = src[i][j];

}

}

}

int src[2048][2048]; int dstij[2048][2048]; int dstji[2048][2048]; int main(){

int t, m;

for (t = 0; t < 2048; t++){ for (m = 0; m < 2048; m++){

src[t][m] = t + m;

}

}

clock\_t startij, finishij, startji, finishji;

startij = clock(); copyij(src, dstij); finishij = clock();

double durationij = (double)(finishij - startij) / CLOCKS\_PER\_SEC;

printf("copyij %f s\n", durationij);

startji = clock(); copyji(src, dstji); finishji = clock();

double durationji = (double)(finishji - startji) / CLOCKS\_PER\_SEC;

printf("copyji %f s\n", durationji); return 0;

}

# gcc –o matrix matrix.c

#### 提交要求：

请提交一个压缩包 151220000.zip 到 ftp 的 lab01 文件夹（注意修改学号和压缩格式，如果需要覆盖第一次提交，请提交 151220000\_1.zip，依次类推）。压缩包内部应该是一个目录。

151220000.zip 解压后获得目录内容如下：

151220000

|----heart.S

|----sqr.c

|----matrix.c

|----report.pdf

或者 151220000\_1.zip 解压后获得内容与上面相同（注意，压缩文件内的文件夹名不需要加上\_1）。

## 实验 2 数据表示和运算实验

一、实验目的

* 1. 了解并学习计算机的数据表示方式，如无符号编码、补码编码和浮点数编码等，了解它们可表示的值的范围。
  2. 了解并学习计算机的算术运算方式，理解不同数据类型的运算属性。二、实验内容

1、在 32 位计算机中运行一个 C 语言程序，在该程序中出现了以下变量的初值，请在表格中填写它们对应的机器数（ 用十六进制表示）。在 gdb 里面使用 x/1xw 查看

int/unsigned/float 的机器数，使用 x/1xh 查看 short/unsigned short 的机器数，使用 x/1xb 查看 char 的机器数，使用 x/1xg 查看 double 的机器数：

（1）int x=-32768 （2）short y=522 （3）unsigned z=65530 （4）char c=’@’

（5）float a=-1. 1 （6）double b=10.5 （7）float u = 123456.789e4 （8）double v= 123456.789e4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 变量 | x | y | z | c |
| 机器数 | 0xffff8000 | 0x020a | 0x0000fffa | 0x40 |
| 变量 | a | b | u | v |
| 机器数 | 0xbf8ccccd | 0x4025000000000000 | 0x4e932c06 | 0x41d26580b4800000q |

运行下面的代码验证输出是否与 gdb 查看的结果一致：

### #include <stdio.h> int main() {

int x = -32768; short y = 522; unsigned z = 65530; char c = '@';

float a = -1.1; double b = 10.5;

float u = 123456.789e4; double v = 123456.789e4;

printf("++++++++++Machine value++++++++++++++++\n"); printf("x = 0x%x\n", x);

printf("y = 0x%hx\n", y); printf("z = 0x%x\n", z); printf("c = 0x%hhx\n", c);

printf("a = 0x%x\n", \*(unsigned \*)&a);

printf("b = 0x%llx\n",\*(unsigned long long \*)&b); printf("u = 0x%x\n", \*(unsigned \*)&u);

printf("v = 0x%llx\n",\*(unsigned long long \*)&v); printf("++++++++++Real value+++++++++++++++++++\n"); printf("x = %d\n", x);

printf("y = %hd\n", y); printf("z = %u\n", z);

printf("c = %c\n", c);

printf("a = %f\n", a);

printf("b = %f\n",b);

printf("u = %f\n", u);

printf("v = %f\n",v);

}

2、使用命令 gcc –ggdb swap.c –o swap 编译下面的 swap.c 代码，完成后面的实验

void xor\_swap(int \*x, int \*y){

\*y=\*x ^ \*y; /\* 第一步 \*/

\*x=\*x ^ \*y; /\* 第二步 \*/

\*y=\*x ^ \*y; /\* 第三步 \*/

}

int main() { int a = 1; int b = 2;

xor\_swap(&a, &b);

}

* + 1. 使用 gdb 命令查看程序变量的取值，填写下面两个表格：

|  |  |  |  |
| --- | --- | --- | --- |
| a 的存放地址(&a) | b 的存放地址(&b) | x 的存放地址(&x) | y 的存放地(&y) |
| 0xbffff0f4 | 0xbffff0f8 | 0xbffff0f4 | 0xbffff0f8 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 执行步数 | x 的值(机器值，  用十六进制) | y 的值(机器值，  用十六进制) | \*x 的值(程序中的  真值，用十进制) | \*y 的值(程序中的真  值，用十进制) |
| 第一步前 | 0x00000001 | 0x00000002 | 1 | 2 |
| 第一步后 | 0x00000001 | 0x00000003 | 1 | 3 |
| 第二步后 | 0x00000002 | 0x00000003 | 2 | 3 |
| 第三步后 | 0x00000002 | 0x00000001x | 2 | 1 |

* + 1. 运行下面的 reverse.c，并说明输出这种结果的原因，修改代码以得到正确的逆序数组

#include <stdio.h>

void xor\_swap(int \*x, int \*y) {

\*y=\*x ^ \*y; /\* 第一步 \*/

\*x=\*x ^ \*y; /\* 第二步 \*/

\*y=\*x ^ \*y; /\* 第三步 \*/

}

void reverse\_array(int a[], int len) {

int left, right=len-1;

for (left=0; left<=right; left++, right--)

xor\_swap(&a[left], &a[right]);

}

int main() {

int a[] = {1,2,3,4,5,6,7};

reverse\_array(a, 7);

int i;

for(i = 0; i < 7; ++i)

printf("%d ", a[i]);

printf("\n");

}

3、编译并运行下面的程序，使用 gdb 指令查看变量的取值，解释语句输出为 False 的原因并填写在表格中。

#include <stdio.h>

#include <limits.h> int main(){

int x = INT\_MAX; float xf = x; double xd = x;

printf("++++++++++True or False++++++++++++++++\n"); printf("x==(int)xd %s\n",x==(int)xd?"True":"False"); //语句一printf("x==(int)xf %s\n",x==(int)xf?"True":"False"); //语句二

float p1 = 3.141592653; float p2 = 3.141592654;

printf("p1!=p2 %s\n",p1!=p2?"True":"False");//语句三

float f = 1.0e20; vimble d = 1.0;

double result1 = d+(f-f); double result2 = (d+f)-f;

printf("result1==d %s\n",result1==d?"True":"False");//语句四

printf("result2==d %s\n",result2==d?"True":"False");//语句五

}

|  |  |  |
| --- | --- | --- |
|  | 输 出  True/False | 原因 |
| 语句一 | T |  |
| 语句二 | F |  |
| 语句三 | F |  |

|  |  |  |
| --- | --- | --- |
| 语句四 | T |  |
| 语句五 | F |  |

4、观察下面 data\_rep.c 程序的运行：

int main() {

char x = 0x66; char y = 0x39; char x\_bit\_not = ~x; char x\_not = !x;

char x\_bit\_and\_y = x & y; char x\_and\_y = x && y; char x\_bit\_or\_y = x | y; char x\_or\_y = x || y;

char x1 = 0xB0; char y1 = 0x8c;

char sum\_x1\_y1 = x1 + y1; char diff\_x1\_y1 = x1 - y1; char diff\_y1\_x1 = y1 - x1;

unsigned char x2 = 0xB0; unsigned char y2 = 0x8c;

unsigned char sum\_x2\_y2 = x2 + y2; unsigned char diff\_x2\_y2 = x2 - y2; unsigned char diff\_y2\_x2 = y2 - x2;

}

* + - 1. 使用命令 gdbtui data\_rep 进入 gdb 的 TUI 调试模式，之后分别输入命

令：layout asm 和 layout regs，再输入命令 start 启动程序，然后使用 si 命令进行单步运行。请在单步运行过程中完成下面的表格（如果 TUI 模式有问题就直接用

gdb，忽略掉前面的提示即可）：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 机器数  (十六进制) | 真值  (十进制) |  | 机器数  ( 十 六 进制) | 真值  ( 十 进  制) |
| x | 0x66 | 102 | y | 0x39 | 57 |
| ~x | 0x99 | -103 | !x | 0x00 | 0 |
| x & y | 0x20 | 32 | x && y | 0x01 | 1 |
| x | y | 0x7f | 127 | x || y | 0x01 | 1 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 机器数  (十六进制) | 真值  (十进制) | OF | SF | CF | AF |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| x1 | 0xb0 | -80 |  |  |  |  |
| y1 | 0x8c | -116 |  |  |  |  |
| x1+y1 | 0x3c | 60 |  |  |  |  |
| x1-y1 | 0x24 | 36 |  |  |  |  |
| y1-x1 | 0xdc | -36 |  |  |  |  |
| x2 | 0xb0 | 176 |  |  |  |  |
| y2 | 0x80 | 140 |  |  |  |  |
| x2+y2 | 0x3c | 60 |  |  |  |  |
| x2-y2 | 0x24 | 36 |  |  |  |  |
| y2-x2 | 0xdc | 220 |  |  |  |  |

* + - 1. 写出上面表格中每个标识位变化的原因，可直接在上表中注明。

#### 提交要求：

请提交一个压缩包 151220000.zip 到 ftp 的 lab02 文件夹（注意修改学号和压缩格式， 如果需要覆盖第一次提交，请提交 151220000\_1.zip，依次类推）。压缩包内部应该是一个目录。

151220000.zip 解压后获得目录内容如下：

151220000

|----reverse.c

|----report.pdf // report 中应包含实验中的所有表格

## 实验 3 数据分支控制实验

一、实验目的

了解数据分支流程。

识别汇编码中的分支跳转条件。并了解如何修改分支条件。二、实验内容

1、下面是 do\_loop 函数的定义，请自己编写一个 main 函数，要求如下：

1)在 main 函数中调用 do\_loop

2)main 函数能从 stdin 中接收输入参数并传递给 do\_loop

3)使用命令 gcc -O1 -ggdb do\_loop.c -o do\_loop 得到可执行程序 do\_loop

short do\_loop(short x, short y, short k) { do {

x\*=(y%k) ;

k--;

} while ((k>0) && (y>k)); return x;

}

实验内容如下：

（1） 输入参数 x=2, y = 4000, k=3 使用 gdb 进入 do\_loop 的第一次循环：

a 在执行指令 cltd 前 edx 的值是多少？ -1208248196

b 在刚执行完 cltd 后 edx 的值是多少？ 0

c 请回答为 cltd 指令的作用

d 在执行指令 idiv 后 edx 的值又变为了多少？请解释这种变化。 1

（2）使用输入 x=2，y=40000，k=3 重复（1）的内容

-1208248196

-1

0

Hints:

1. 使用 gdb do\_loop 进入 gdb

2. 使用 break do\_loop 设置 breakpoint

3. run

4. 使用 C-x C-a 组合键进入 TUI 模式

1. layout asm
2. si

7. 查看寄存器 edx 的值 print $edx

8.查看 do\_loop 汇编码 disassemble do\_loop

如果 TUI 模式有问题请直接时候用 gdb 查看内存及寄存器的值。

2、（简单）给定 if\_else.s 文件，完成如下要求：

修改 if\_else.s 中 if\_else 片段，只准修改分支条件，不需修改分支中的内容，达到如下要求。

A：输入 12 15 ，要求现在 if\_else 的返回值为 1 （原来返回值为 0）

B：输入学号后四位，（如学号后四位是 1234 则输入 12 34 ）要求输出结果为 2

（A、B 分别在不同文件中实现，并各自重新命名为 if\_else\_A.s 以及 if \_else\_B.s） Hints：

a：可以使用 gcc if\_else.s –o if\_else 将.s 文件生成可执行程序。可执行程序中会根据输入将结果输出到屏幕。（可具此判断修改后的.s 文件是否达到要求）

b：if\_else 片段如下：可修改语句已用红色标出

if\_else:

.LFB0:

.cfi\_startproc pushl %ebp

.cfi\_def\_cfa\_offset 8

.cfi\_offset 5, -8

movl %esp, %ebp

.cfi\_def\_cfa\_register 5 subl $16, %esp

cmpl $0, 8(%ebp) jle .L2

cmpl $29, 12(%ebp) jg .L2

movl $0, -4(%ebp) jmp .L3

.L2:

cmpl $0, 8(%ebp) jle .L4 cmpl $30, 12(%ebp) jle .L4

movl $1, -4(%ebp) jmp .L3

.L4:

movl $2, -4(%ebp)

.L3:

movl -4(%ebp), %eax leave

.cfi\_restore 5

.cfi\_def\_cfa 4, 4 ret

.cfi\_endproc

.LFE0:

.size if\_else, .-if\_else

.section .rodata

.LC0:

.string "%d %d"

.LC1:

.string "%d\n"

.text

.globl main

.type main, @function

3、给定一个 c 源程序 switch\_case.c，编译链接成可执行文件，并根据汇编码 switch\_case.s

完成如下要求：

1） 利用 gdb，分别输入参数： A: x = 2；

B: x = 6； C: x = 9； D: x = 3;

观察每个输入后 switch\_case 程序的返回值各是多少？

2）输入学号最后一位，观察 switch\_case 程序的返回值。

3）将如下函数填充完整（要求 case 内的值最终按照由小到大排列）：

int switch\_case(int x){ int result =0; switch(x){

case : result = x\* ; break;

case : result = x\* ; break;

case : result = x\* ; break;

default: result = ;

}

return result += ;

}

使用命令 objdump –d switch\_case > switch\_case.s 得到的函数 switch\_case 的汇编代码。并将源码填充完整。

Hints:

1.可以使用 gdb 运行观察每个寄存器的值，参考实验一的提示。

## 实验报告要求：

请提交一个压缩包 151220000.zip 到 ftp 的 lab03 文件夹（注意修改学号和压缩格式， 如果需要覆盖第一次提交，请提交 151220000\_1.zip，依次类推）。压缩包内部应该是一个目录。

151220000.zip 解压后获得目录内容如下：

151220000

|----if\_else\_A.s

|----if\_else\_B.s

|----report.pdf（包含三个实验内容的结果，细节可截图说明）

**实验** 4 **复杂结构实验**

一、实验目的

理解函数调用过程中堆栈的变化情况理解数组、链表在内存中的组织形式

理解 struct 和 union 结构数据在内存中的组织形式

二、实验内容

1、给定如下 array\_init.c 文件，使用命令 gcc -ggdb array\_init.c –o array\_init 编译代码， 使用命令 objdump –d array\_init > array\_init.s 反汇编二进制文件，分析反汇编后代码，并完成以下要求

#include <stdio.h>

#define M 2

#define N 10

void init(int a[N]){ int i;

char temp[N];

printf("input student id : \n"); fgets(temp,N,stdin); for(i=0;i<N;i++){

a[i]=temp[i]-'0';

}

}

void g(){

int a[N];

init(a);

}

void print(int b[M]){ int i; for(i=0;i<M;i++){

printf("%d ",b[i]);

}

printf("\n");

}

void f(){

int b[M];

print(b);

}

int main(){

g();

f();

return 0;

}

实验要求：

1. 查看函数 g 和 f 的反汇编代码，分别给出函数 g 和 f 中数组 a，b 在栈上的分布，在下图中给出 a[0]-a[9]以及 b[0]、b[1]位置。

函数 g 函数 f

|  |
| --- |
| old ebp |
|  |
|  |
| %gs(14) |
| b[1] |
| b[0] |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

|  |
| --- |
| old ebp |
|  |
|  |
| %gs(14) |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| a[1] |
| a[0] |

1. 运行程序，程序的输入为 9 位学号，观察输出。请详细解释为什么 b[0]和 b[1]是这两个值。说明使用未初始化的程序局部变量的危害。

6 -48

2 给定如下三维数组 A 的定义以及 store\_ele 函数，其中 R,S,T 是用#define 定义的常量。

又给定 3\_d\_array 这个可执行文件，在 3\_d\_array 的 main 函数中仅调用了一次 store\_ele 函数，使用命令 objdump –d 3\_d\_array > 3\_d\_array.s 反汇编二进制文件,观察 store\_ele 函数。

int A[R][S][T];

int store\_ele(int i,int j,int k,int dest){ A[i][j][k] = dest;

return sizeof(A);

}

1. push %ebp
2. mov %esp,%ebp

|  |  |  |
| --- | --- | --- |
| 3 | mov | 0xc(%ebp),%eax  **// %eax -> j** |
| 4 | mov | 0x8(%ebp),%ecx  **// %ecx -> i** |
| 5 | mov | %eax,%edx  **// %edx -> j** |
| 6 | lea | (%edx,%edx,1),%eax  **// %eax -> 2j** |
| 7 | mov | %eax,%edx  **// %edx -> 2j** |
| 8 | lea | 0x0(,%edx,8),%eax **// %eax -> 16j** |
| 9 | sub | %edx,%eax  **// %eax ->14j** |
| 10 | imul | $0xb6,%ecx,%edx **// %edx -> 182i** |
| 11 | add | %eax,%edx  **// %edx -> 14j + 182i** |
| 12 | mov | 0x10(%ebp),%eax  **// %eax -> k** |
| 13 | add | %eax,%edx  **// %edx -> 182i + 14j + k** |
| 14 | mov | 0x14(%ebp),%eax **// %eax -> dest** |
| 15 | mov | %eax,0x804a060(,%edx,4) |
| 16 | mov | $0x5c6c0,%eax |
| 17 | pop | %ebp |
| 18 | ret |  |

实验要求

（1）将数组地址计算扩展到三维，给出 A[i][j][k]地址的表达式。(A 的定义为 int A[R][S][T]，

sizeof(int)=4,起始地址设为 addr(A)) addr(A) +

（2）使用命令 gdb ./3\_d\_array 启动 gdb 调试。在 store\_ele 函数入口设置断点，以自己的

9 位学号为输入，运行程序。在 store\_ele 函数中，单步执行，并打印出每步汇编指令执行后寄存器 eax、ecx、edx 的值。上面给出了 store\_ele 函数的汇编指令及其指令编号，根据自己的实验结果填写每条指令运行后的结果。

（3）根据以上内容确定 R、S、T 的取值

|  |  |  |  |
| --- | --- | --- | --- |
|  | %eax | %ecx | %edx |
| 3 | 7 | 6 | 10 |
| 4 | 7 | 52 | 10 |
| 5 | 7 | 52 | 7 |
| 6 | 14 | 52 | 7 |
| 7 | 14 | 52 | 14 |
| 8 | 112 | 52 | 14 |
| 9 | 98 | 52 | 14 |
| 10 | 98 | 52 | 9464 |
| 11 | 98 | 52 | 9562 |
| 12 | 6 | 52 | 9562 |
| 13 | 6 | 52 | 9568 |
| 14 | 161220076 | 52 | 9568 |
| 15 | 161220076 | 52 | 9568 |
| 16 | 378560 | 52 | 9568 |

3、函数 recursion 是一个递归调用函数。其原函数存在缺失，试根据其汇编代码确定原函数， 保存为 recursion.c

int recursion (intx){ if( x<2 )

return 1;

else

}

Return recursion(x-1)+recursion(x-2);

00000000 <recursion>:

|  |  |  |  |
| --- | --- | --- | --- |
| 0: | 55 | push | %ebp |
| 1: | 89 e5 | mov | %esp,%ebp |
| 3: | 53 | push | %ebx |
| 4: | 83 ec 04 | sub | $0x4,%esp |
| 7: | 83 7d 08 02 | cmpl | $0x2,0x8(%ebp) |
| b: | 7f 07 | jg | 14 <recursion+0x14> |
| d: | b8 01 00 00 00 | mov | $0x1,%eax |
| 12: | eb 28 | jmp | 3c <recursion+0x3c> |
| 14: | 8b 45 08 | mov | 0x8(%ebp),%eax |
| 17: | 83 e8 01 | sub | $0x1,%eax |
| 1a: | 83 ec 0c | sub | $0xc,%esp |
| 1d: | 50 | push | %eax |
| 1e: | e8 fc ff ff ff | call | 1f <recursion+0x1f> |
| 23: | 83 c4 10 | add | $0x10,%esp |
| 26: | 89 c3 | mov | %eax,%ebx |
| 28: | 8b 45 08 | mov | 0x8(%ebp),%eax |
| 2b: | 83 e8 02 | sub | $0x2,%eax |
| 2e: | 83 ec 0c | sub | $0xc,%esp |
| 31: | 50 | push | %eax |
| 32: | e8 fc ff ff ff | call | 33 <recursion+0x33> |
| 37: | 83 c4 10 | add | $0x10,%esp |
| 3a: | 01 d8 | add | %ebx,%eax |
| 3c: | 8b 5d fc | mov | -0x4(%ebp),%ebx |
| 3f: | c9 | leave |  |
| 40: | c3 | ret |  |

4、给定以下结构定义struct ele{

union {

struct{

int\* p; int x;

}e1;

int y[3];

};

struct ele \*next;

};

实验要求

1. 确定下列字节的偏移量。

e1.p 0

e1.x 4

y 0

y[0] 0

y[1] 4

y[2] 8

next 12

（2）下面的过程（省略一些表达式）是对链表进行操作，链表是以上述结构作为元素的。现有 proc 函数主体的汇编码，查看汇编代码，并根据汇编代码补全 proc 函数中缺失的表达式，并保存为 proc.c。（不需要进行强制类型转换）

void proc(struct ele \*up){

up-> y[2] = \*(up-> e1.p )+up-> y[2] ;

}

00000000 <proc>:

|  |  |  |  |
| --- | --- | --- | --- |
| 0: | 55 | push | %ebp |
| 1: | 89 e5 | mov | %esp,%ebp |
| 3: | 8b 45 08 | mov | 0x8(%ebp),%eax // %eax->&up |
| 6: | 8b 40 0c | mov | 0xc(%eax),%eax // %eax->&next |
| 9: | 8b 55 08 | mov | 0x8(%ebp),%edx // %edx->&up |
| c: | 8b 12 | mov | (%edx),%edx // %edx->up |
| e: | 8b 0a | mov | (%edx),%ecx |
| 10: | 8b 55 08 | mov | 0x8(%ebp),%edx |
| 13: | 8b 52 08 | mov | 0x8(%edx),%edx // %edx->y[2] |
| 16: | 01 ca | add | %ecx,%edx |
| 18: | 89 10 | mov | %edx,(%eax) |
| 1a: | 90 | nop |  |

1b: 5d pop %ebp

1c: c3 ret

（3）有以下 main 函数，该 main 函数中声明了一数组和一链表并打印了每个元素的地址， 查看地址，并解释产生原因，体会数组与链表分别使用静态内存和动态内存的差异。

int main(){

struct ele a[5];

struct ele \* head, \* p; head=NULL;

for(int i=0;i<5;i++){

p=(struct ele \*)malloc(sizeof(struct ele)); p->next=NULL;

if(head==NULL){ head=p;

}else{

p->next=head; head=p;

}

for(int j=0;j<i;j++){ malloc(sizeof(int));

}

}

printf("array address:\n");

printf("%x\t%x\t%x\n",(unsigned int)&a[0],(unsigned int)&a[1],(unsigned int)&a[2]); printf("\nlist address:\n");

p=head; while(p!=NULL){

printf("%x\t",(unsigned int)p); p=p->next;

}

printf("\n");

}

## 实验报告要求：

请提交一个压缩包 151220000.zip 到 ftp 的 lab04 文件夹（注意修改学号和压缩格式， 如果需要覆盖第一次提交，请提交 151220000\_1.zip，依次类推）。压缩包内部应该是一个目录。

151220000.zip 解压后获得目录内容如下：

151220000

|----recursion.c

|----proc.c

|----report.pdf（包含四个实验内容的结果，细节可截图说明

## 实验 5 存分配、缓冲区溢出实验

**实 验 5.1：** 一、实验目的

1. 掌握基本的函数调用规则。
2. 掌握局部变量在函数栈上的存放原则。
3. 理解数组越界的危害。二、实验内容

1、 使用命令 gcc overflow1.c -fno-stack-protector –o overflow1 编译 overflow1.c，overflow1.c

代码如下：

#include <stdio.h>

#include <stdlib.h> void why\_here(void)

{

printf("why u r here?!\n"); exit(0);

}

int foo(){

int buff[1];

buff[2] = (int)why\_here; return 0;

}

int main(int argc, char \* argv[])

{

foo(); return 0;

}

三、实验要求

* 1. 运行程序 overflow1，观察出现的实验现象。
  2. 反汇编源程序，根据汇编代码回答下面的问题：

（1） 假设内存地址 0xbfffef30-0xbfffef33 保存 foo 函数的返回地址，请推断出数组 buf 的开始地址。0xbfffef1c

（2） 根据（1）的结果，推断出变量 buf[2]在内存中的开始地址。0xbfffef24

* 1. 根据 2 的分析结果，解释出现实验现象的原因。

#### 实验 5.2：

一、 实验目的

1. 掌握函数调用规则。
2. 理解标准输入函数 gets 的缺陷。
3. 掌握缓冲区溢出的基本原理。
4. 了解防止缓冲区溢出的防御方法。二、实验内容

1、使用命令gcc overflow2.c -fno-stack-protector –o overflow2 编译overflow2.c，overflow2.c

代码如下：

#include <stdio.h> void doit(void)

{

char buf[8];

gets(buf); printf("%s\n", buf);

}

int main(void)

{

printf("So... The End...\n"); doit();

printf("or... maybe not?\n"); return 0;

}

三、实验要求

1. 运行 overflow2，尝试输入任意个数的字符，如字符’a’，猜测需要连续输入几个字符“f”才能产生 segmentation fault？ 16
2. 反汇编源程序，验证 1 中的猜测。假设数组 buf 的开始地址为 0xbfffef30，输入 N 个字符’a’后出现 segmentation fault。查阅 gets 函数的资料，如 man。请写出输入 N+6 个字符后，内存地址范围[0xbfffef30, 0xbfffef30+N+5]的存放情况。都是a

3.根据 2 的假设，请写出未被污染前，地址范围 0xbfffef30+N 到 0xbfffef30+N+7 原始存放的数据。0xbfffef30+N~0xbfffef30+N+3 -> ebp 0xbfffef30+N+3~0xbfffef30+N+7 -> return addr

4.将源程序中的 gets 函数替换为 fgets 函数，然后重新实验。比较 fgets 函数和 gets 函数的异同，简要说明采用 fgets 函数为何无法完成上述攻击。

fgets() : 从文件结构体指针stream中读取数据，每次读取一行。读取的数据保存在buf指向的字符数组中，每次最多读取bufsize-1个字符（第bufsize个字符赋'\0'），**如果文件中的该行，不足bufsize个字符，则读完该行就结束。**

5. 查阅相关资料，简要阐述抵御缓冲区溢出的两种防御方法（栈随机化和栈破坏检测）的主要思想。

Hints：使用 gdb 查看栈的存储情况，跟踪程序执行过程中 ebp 寄存器的变化，以及观察程序 printf 的输出等，使用 gdb TUI 模式的 layout regs 命令可以实时监测各个寄存器的变化，在使用 gdb 前先把程序分析清楚，gdb 只是最后的救命稻草！

**实验 5.3：** 一、实验目的

1. 掌握不同机器平台不同类型整数的大小限制。
2. 理解整数溢出的危害。
3. 了解常用的抵御整数溢出的防御方法。二、实验内容

1、 使用命令 gcc –o overflow3 overflow3.c 编译 overflow3.c 文件。

#include <stdio.h>

#include <stdlib.h>

typedef unsigned int uint; int\* foo(uint len,int \*array)

{

int \*buf, i;

buf = malloc(len\*sizeof(int));

printf("malloc %lu bytes\n",len\*sizeof(int)); if(buf == NULL) {

return NULL;

}

printf("loop time: %d[0x%x]\n",len,len); for(i = 0; i < len; i++)

{

buf[i] = array[i];

}

return buf;

}

int main(int argc, char \*argv[]){ int array[5] = {1, 2, 3, 4, 5};

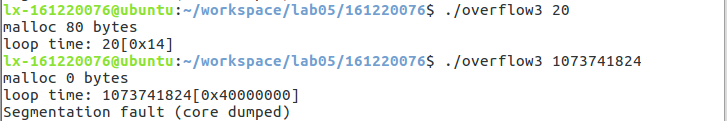
foo(atoi(argv[1]), array);

return 0;

}

三、实验要求

1. 分别以参数./ overflow3 20和./ overflow3 1073741824 运行程序，观察出现的现象。



1. 回答 32 位平台 size\_t 类型的范围，计算出两次分配内存大小的不同。

3. 解释出现的实验现象。

4. 分析现有 Linux 允许 malloc 0 个字节的合理性，回答这是操作系统内存设计还是 c

运行库设计的问题。

## 综合实验 二进制炸弹实验

一、实验目的

本实验通过要求你使用课程所学知识拆除一个“binary bombs”来增强对程序的机器级表示、汇编语言、调试器和逆向工程等方面原理与技能的掌握。 一个“binary bombs”（二进制炸弹，下文将简称为炸弹）是一个 Linux 可执行程序，包含了 6 个阶段（或层次、关卡）。炸弹运行的每个阶段要求你输入一个特定字符串，你的输入符合程序预期的输入，该阶段的炸弹就被拆除引信即解除了，否则炸弹“爆炸”打印输出 “BOOM!!!”。

二、实验内容

拆除尽可能多的炸弹层次，每个炸弹阶段考察了机器级程序语言的一个不同方面，难度逐级递增：

阶段 1：字符串比较阶段 2：循环

阶段 3：条件/分支

阶段 4：递归调用和栈阶段 5：指针

阶段 6：链表/指针/结构

另外还有一个隐藏阶段，只有当你在第 4 阶段的解后附加一特定字符串后才会出现。为完成二进制炸弹拆除任务，你需要使用 gdb 调试器和 objdump 来反汇编炸弹的可执行文件并单步跟踪调试每一阶段的机器代码，从中理解每一汇编语言代码的行为或作用，进而设法推断拆除炸弹所需的目标字符串。比如在每一阶段的开始代码前和引爆炸弹的函数前设置断点。

三、实验程序

你将在下面网站获得你的 bomb（<http://114.212.10.9/bomb>）

在这里你将看到一个二进制炸弹请求框，你可以填入你的学号然后按下下载按钮。这个服务器将返回给你的浏览器一个\*.tar文件的bomb，\*代表你的学号。

Tar文件包含2个文件：

Bomb:可执行的32位二进制bomb

Bomb.c:写有bomb的主程序的源文件

你的任务是去拆除炸弹，你可以用许多工具去帮助你拆除你的炸弹。最好的方法是去用你最喜欢的调试器去调试你的二进制程序，如 gdb。

Bomb 程序默认从 stdin 读入。你也可以通过文件输入，文件中的每一行包括一个输入

字符串。

四、提交要求

1、实验报告，实验报告内注明姓名学号，报告的文件名为自己的学号，在报告内要写明自己的分析过程，每个阶段都需要有一个详细的分析过程。

2、新建一个文本文件，该文件的第 n 行对应第 n 步的输入，文件命名成自己的学号， 不要带任何的后缀，尤其是不要有.txt 后缀，将该文件和实验报告一起上传到课程网站。

3、可以使用命令 vim 151220001 创建名为“151220001”的文本文件。

4、每一步的输入字符串以换行符结束，不要遗漏字符串中的任何一个标点以及空格。

五、实验结果

star

1 2 3 1 2 3

0 61

6 balabalaxiaomouxian(secret phrase)

1<1000

2 5 4 6 3 1

7