实验零：操作系统实验准备

1.实验目的：

- 了解操作系统开发实验环境
- 熟悉命令行方式的编译、调试工程
- 掌握基于硬件模拟器的调试技术
- 熟悉 C 语言编程和指针的概念
- 了解 X86 汇编语言

2.准备知识：

2.1 了解 OS 实验

写一个操作系统难吗？别被现在上百万行的 Linux 和 Windows 操作系统吓倒。当年 Thompson 乘他老婆带着小孩度假留他一人在家时，写了 UNIX；当年 Linus 还是一个 21 岁大学生时完成了 Linux 雏形。站在这些巨人的肩膀上，我们能否也尝试一下做“巨人”的滋味呢？

MIT 的 Frans Kaashoek 等在 2006 年参考 PDP-11 上的 UNIX Version 6 写了一个可在 X86 上跑的操作系统 xv6（基于 MIT License），用于学生学习操作系统。我们可以站在他们的肩膀上，基于 xv6 的设计，尝试着一步一步完成一个从“空空如也”到“五脏俱全”的“麻雀”操作系统—ucore，此“麻雀”包含虚存管理、进程管理、处理器调度、同步互斥、进程间通信、文件系统等主要内核功能，总的内核代码量（C+asm）不会超过 5K 行。充分体现了“小而全”的指导思想。

ucore 的运行环境可以是真实的 X86 计算机，不过考虑到调试和开发的方便，我们可采用 X86 模拟器，比如 QEMU、BOCHS 等，或 X86 虚拟运行环境，比如 VirtualBox、VMware Player 等。ucore 的开发环境主要是 GCC 中的 gcc、gas、ld 和 MAKE 等工具，也可采用集成了这些工具的 IDE 开发环境 Eclipse-CDT。运行环境和开发环境既可以在 Linux 或 Windows 中使用。

那我们准备如何一步一步实现 ucore 呢？安装一个操作系统的开发过程，我们可以有如下的开发步骤：

1) 启动操作系统的 bootloader，用于了解操作系统启动前的状态和要做的准备工作，了解运行操作系统的硬件支持，操作系统如何加载到内存中，理解两类中断--“外设中断”，“陷阱中断”等；
2) 物理内存管理子系统，用于理解 x86 分段/分页模式，了解操作系统如何管理物理内存；
3) 虚拟内存管理子系统，通过页表机制和换入换出（swap）机制，以及中断-“故障中断”、缺页故障处理等，实现基于页的内存替换算法；
4) 内核线程子系统，用于了解如何创建相对与用户进程更加简单的内核态线程，如果对内核线程进行动态管理等；
5) 用户进程管理子系统，用于了解用户态进程创建、执行、切换和结束的动态管理过程，了解在用户态通过系统调用得到内核态的内核服务的过程；
6) 处理器调度子系统，用于理解操作系统的调度过程和调度算法；
7) 同步互斥与进程间通信子系统，了解进程间如何进行信息交换和共享，并了解同步互斥的具体实现以及对系统性能的影响，研究死锁产生的原因，以及如何避免死锁；
8) 文件系统，了解文件系统的具体实现，与进程管理等的关系，了解缓存对操作系统 IO 访问的性能改进，了解虚拟文件系统（VFS）、buffer cache 和 disk driver 之间的关系。

其中每个开发步骤都是建立在上一个步骤之上的，就像搭积木，从一个一个小木块，最终搭出来一个小房子。在搭房子的过程中，完成从理解操作系统原理到实践操作系统设计与实现的探索过程。这个房子最终的建筑架构和建设进度如下图所示：

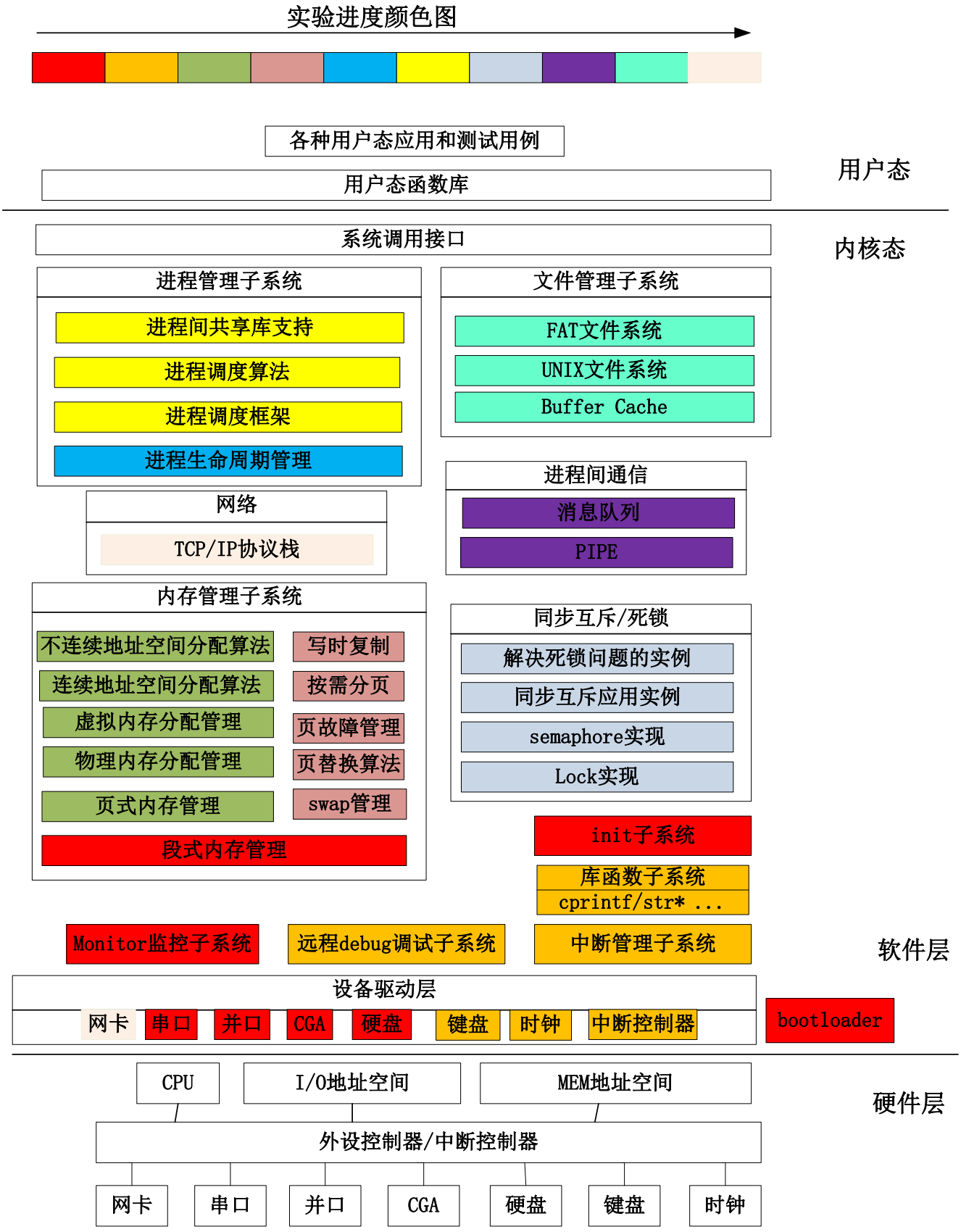


图 1 ucore 系统结构图

如果完成上诉实验后还想做更大的挑战，那么可以参加 ucore 的研发项目，我们可以完成 ucore 的网络协议栈，增加图形系统，在 ARM 嵌入式系统上运行，支持虚拟化功能等。这些项目已经有同学参与，欢迎有兴趣的同学加入！

2.2 熟悉实验环境

我们参考了 MIT 的 xv6、Harvard 的 OS161 和 Linux 等设计了 ucore OS 实验，所有 OS 实验需在 Linux 下运行。对于经验不足的同学，推荐参考通过虚拟机使用 Linux 实验环境一节用虚拟机方式进行试验。

2.2.1开发 OS 实验的简单步骤

在我们提供的 lab1~lab8 实验软件包中，大致经过如下过程就可以完成使用。

1. 解压软件包 例如执行：tar jxf lab1.tar.bz2
2. 进入各个 OS 实验工程目录 例如： cd lab1
3. 根据实验要求阅读源码并修改代码（用各种文本编辑器）
4. 并编译源码 例如执行：make
5. 如编译不过则返回步骤 3
6. 如编译通过则测试是否基本正确，例如执行：make grade
7. 如果实现基本正确（即看到步骤 6 的输出存在不是 OK 的情况）则返回步骤 3
8. 如果实现基本正确（即看到步骤 6 的输出都是 OK）则生成实验提交软件包，例如执行：make handin
9. 把生成的使用提交软件包和实验报告上传/email 给助教和老师。

另外，可以通过”make qemu”让 OS 实验工程在 qemu 上运行；可以通过”make debug”或“make debug-nox “命令实现通过 gdb 远程调试 OS 实验工程。

2.2.2通过虚拟机使用 Linux 实验环境（推荐：最容易的实验环境安装方法）

这是最简单的一种通过虚拟机方式使用 Linux 并完成 OS 各个实验的方法，不需要安装 Linux 操作系统和各种实验所需开发软件。首先安装 VirtualBox 虚拟机软件（有 windows 版本和其他 OS 版本，可到 http://www.virtualbox.org/wiki/Downloads 下载），然后在 OS FTP 服务器上下载一个老师已经安装好各种所需编辑/开发/调试/运行软件的 Linux 实验环境的文件(即一个虚拟磁盘镜像文件，lab4student2011.7z)。用 7zip 软件(有 windows 版本和其他 OS 版本，可到 http://www.7-zip.org/download.html 下载)解压 lab4student2011.7z 后为 ubuntu10.10.vdi，大小大约为 3GB，在 VirtualBox 中加载这个虚拟磁盘文件，就可以启动并运行 Linux 实验环境了。
启动到提示输入用户名时，请输入
student
当提示输入口令时，只需简单敲一个空格键和回车键即可。然后就进入到开发环境中了。实验内容位于 labs 目录下。

10.

2.2.3安装使用 Linux 实验环境（适合希望自己安装 Linux 系统的同学）

这里我们主要以 Ubuntu Linux 10.10（32 bit）作为整个实验的系统软件环境。首先我们需要安装 Ubuntu Linux 10.10。

2.2.3.1WUBI 方式安装（最容易的 Linux 安装方法）

WUBI 是一个专门针对 Windows 用户的 UBUNTU Linux 安装工具，你需要做的只是点击几下鼠标而已。不需要改变分区设置，不需要启动文件，不需要 Live CD。WUBI 让你如同 Windows 操作系统里的其他软件一样安装卸载 Ubuntu，如果你从来没有安装过 UBUNTU Linux，Wubi 很适合你第一次安装 UBUNTU Linux。具体方法如下：

(1)去 OS course ftp 或官方网站 http://releases.ubuntu.com/10.10/ubuntu-10.10-desktop-i386.iso 下载

了一个 ubuntu-10.10-desktop-i386 的 ISO 文件。

(2)通过 winrar 等工具将下载来的 ISO 文件中的 wubi.exe 解压出来，放在任意一个分区的根目录下。这里推荐预留了一个至少大小为８G 的 NTFS 分区，单击 wubi.exe 安装文件，这时会弹出以下

对话框：

(3)设置好分区将要安装的分区，语言，分配的系统大小，用户名和密码（务必记住）之后，点击“安装”，这时如果你的机器已经联网了，会自动从镜像网站上下载 ISO 文件。这里采用绕过 wubi 下载镜像 ISO 的方法安装 ubuntu 10.10，会节省大量时间。避免下载 ISO 文件的这一步非常关键。在进行这一步之前请将网线断开，然后将提前下载来的 ubuntu- 10.10-desktop-i386.iso 文件拷贝至 wubi 所创建的 ubuntu 目录下的 install 文件夹中，重新运行 wubi.exe。这次再也不会提示下载 ISO 文件了。几秒钟后，wubi 就会提示你重新启动系统。注意，此时 ubuntu 并没有安装在硬盘上，必须重新启动才开始进行 ubuntu 10.10 的安装。如下图所示：

(4)点击”完成”按钮，选择重启计算机。计算机重启后，在启动选项中选择 ubuntu，出现”press 'ESC' to … ” 时，不用理会，这时我们熟悉的 ubuntu 滚动条出现在屏幕上。此时，才正式开始安装 ubuntu 10.10 至硬盘分区某一目录下。接下来我们什么也不用做，只需等待。当提示正式安装完成后，重新启动计算机系统，可以发现在启动选项中有"ubuntu"和“windows”。你可以根据你的情况进行选择。

2.2.3.2 使用 Linux

在实验过程中，我们需要了解基于命令行方式的编译、调试、运行操作系统的实验方法。为此，需要了解基本的 Linux 命令行使用。

下面的内容来源于:

http://wiki.ubuntu.org.cn/index.php?title=%E5%91%BD%E4%BB%A4%E8%A1%8C%E6%8C%87%E5%8D%97&variant=zh-cn

可以说 Ubuntu 是当前图形界面最为友好和易操作的 linux 发行版，但还是有很多时候，只需执行几条简单的指令就可以完成繁琐的鼠标点击所完成的操作，从而节约大量的时间和精力，linux 的命令操作模式功能可以实现你需要的所有操作。简单的说，命令行就是基于成行的命令的用户界面。您也可称其为一个**文本化指令序列处理器。**绝大多数情况下， 用户通过输入一行命令（尽管可以不止一行）直接与计算机互动， 所触发的行为基于当前处理器的语法。命令操作模式是一个很简洁的操作界面，它通过输入一条一条的指命（有些情况下可能是成组的命令）来实现对计算机的操作，通常它也被称为“文本操作模式”

2.2.3.2.1如何进入命令模式

假设你正在使用默认的图形界面为 GNOME 的任意版本 Ubuntu Linux。点击 GNOME 菜单—>附件->终端，就可以启动名为 gnome-terminal 的程序，它就是 GNOME 随机的终端模拟器。

2.2.3.2.2 命令模式的基本结构和概念

打开命令终端后你首先可能会注意到类似下面的界面:

```
chy@laptop:~ $
```

你所看到的这些被称为命令终端提示符，它表示计算机已就绪，正在等待着用户输入操作指令。以我的屏幕画面为例，“**chy**"是当前所登录的用户名，“**laptop”**是这台计算机的主机名，“**~**”表示当前目录。此时输入任何指令按回车之后该指令将会提交到计算机运行，比如你可以输入命令：ls 再按下回车:

```
ls [ENTER]
```

注意: [ENTER]是指输入完 ls 后按下回车键，而不是叫你输入这个单词，ls 这个命令将会列出你当前所在目录里的所有文件和子目录列表。

2.2.3.2.3基本用法

下面介绍 bash shell 程序的基本使用方法，它是 ubuntu 缺省的外壳程序。

2.2.3.2.3.1常用指令

(1)**查询文件列表：** (ls)

```
dud@shadowplay:~ $ ls
file1.txt
```

```
file2.pdf
file3.mp3
file1.pdf
another_file.txt
Yet-Another_file.txt
file-with_other-NAME.TXT
```

ls 命令默认状态下将按首字母升序列出你当前文件夹下面的所有内容，但这样直接运行所得到的信息也是比较少的，通常它可以结合以下这些参数运行以查询更多的信息：

- **ls /** 将列出根目录'/'下的文件清单.如果给定一个参数，则命令行会把该参数当作命令行的工作目录。换句话说，命令行不再以当前目录为工作目录。
- **ls -l** 将给你列出一个更详细的文件清单.
- **ls -a** 将列出包括隐藏文件(以.开头的文件)在内的所有文件.
- **ls -h** 将以 KB/MB/GB 的形式给出文件大小,而不是以纯粹的 Bytes.

(2)查询当前所在目录：(pwd)

```
chy@laptop:~ $ pwd
/home/dud
```

(3)进入其他目录：(cd)

```
dud@shadowplay:~ $ pwd
/home/dud
dud@shadowplay:~ $ cd /root/
dud@shadowplay:~ $ pwd
/root
```

上面例子中，当前目录原来是/home/dud,执行 cd /root/之后再运行 pwd 可以发现，当前目录已经改为/root 了。

(4)在屏幕上输出字符： (echo)

```
dud@shadowplay:~ $ echo "Hello World"
Hello World
```

这是一个很有用的命令，它可以在屏幕上输入你指定的参数(""号中的内容)，当然这里举的这个例子中它没有多大的实际意义，但随着你对 LINUX 指令的不断深入，就会发现它的价值所在。

(5)显示文件内容：(cat)

```
dud@shadowplay:~ $ cat file1.txt
Roses are red.
Violets are blue,
and you have the bird-flue!
```

也可以使用 less 或 more 来显示比较大的文本文件内容。

(6)复制文件： (cp)

```
dud@shadowplay:~ $ cp file1.txt file1_copy.txt
dud@shadowplay:~ $ cat file1_copy.txt
Roses are red.
Violets are blue,
and you have the bird-flue!
```

(7)移动文件： (mv)

```
dud@shadowplay:~ $ ls
file1.txt
file2.txt
dud@shadowplay:~ $ mv file1.txt new_file.txt
dud@shadowplay:~ $ ls
```

```
file2.txt
new_file.txt
```

注意：在命令操作时系统基本上不会给你什么提示，当然，绝大多数的命令可以通过加上一个参数—v 来要求系统给出执行命令的反馈信息；

```
dud@shadowplay:~ $ mv -v file1.txt new_file.txt
`file1.txt' -> `new_file.txt'
```

(8)建立一个空文本文件： (touch)

```
dud@shadowplay:~ $ ls
file1.txt
dud@shadowplay:~ $ touch tempfile.txt
dud@shadowplay:~ $ ls
file1.txt
tempfile.txt
```

(9)建立一个目录：(mkdir)

```
dud@shadowplay:~ $ ls
file1.txt
tempfile.txt
dud@shadowplay:~ $ mkdir test_dir
dud@shadowplay:~ $ ls
file1.txt
tempfile.txt
test_dir
```

(10)删除文件/目录：(rm)

```
dud@shadowplay:~ $ ls -p
file1.txt
tempfile.txt
test_dir/
dud@shadowplay:~ $ rm -i tempfile.txt
rm: remove regular empty file `test.txt'? y
dud@shadowplay:~ $ ls -p
file1.txt
test_dir/
dud@shadowplay:~ $ rm test_dir
rm: cannot remove `test_dir': Is a directory
dud@shadowplay:~ $ rm -R test_dir
dud@shadowplay:~ $ ls -p
file1.txt
```

在上面的操作：首先我们通过 ls 命令查询可知当前目下有两个文件和一个文件夹；

- 你可以用参数 -p 来让系统显示某一项的类型，比如是文件/文件夹/快捷链接等等；
- 接下来我们用 rm -i 尝试删除文件，-i 参数是让系统在执行删除操作前输出一条确认提示；i(interactive)也就是交互性的意思；
- 当我们尝试用上面的命令去删除一个文件夹时会得到错误的提示，因为删除文件夹必须使用-R(recursive,循环）参数

特别提示：在使用命令操作时，系统假设你很明确自己在做什么，它不会给你太多的提示，比如你执行 rm -Rf /，它将会删除你硬盘上所有的东西，并且不会给你任何提示，所以，尽量在使用命令时加上-i 的参数，以让系统在执行前进行一次确认，防止你干一些蠢事。如 果你觉得每次都要输入-i 太麻烦，你可以执行以下的命令，让—i 成为默认参数：

```
alias rm='rm -i'
```

(11)查询当前进程：(ps)

```
dud@shadowplay:~ $ ps
PID TTY          TIME CMD
11278 pts/1    00:00:00 bash
24448 pts/1    00:00:00 ps
```

这条命令会例出你所启动的所有进程；

- ps -a 可以例出系统当前运行的所有进程，包括由其他用户启动的进程；
- ps auxww 是一条相当人性化的命令，它会例出除一些很特殊进程以外的所有进程，并会以一个高可读的形式显示结果，每一个进程都会有较为详细的解释；

基本命令的介绍就到此为止，你可以参阅以下地址得到更加详细的命令指南：

- HTML - Bash command reference
- PDF - Unix commands: A Quick Reference Card

2.2.3.2.3.2控制流程

(1)输入/输出

input 用来读取你通过键盘（或其他标准输入设备）输入的信息，output 用于在屏幕（或其他标准输出设备）上输出你指定的输出内容.另外还有一些标准的出错提示也是通过这个命令来实现的。通常在遇到操作错误时，系统会自动调用这个命令来输出标准错误提示；

我们能重定向命令中产生的输入和输出流的位置 。

(2)重定向

如果你想把命令产生的输出流指向一个文件而不是（默认的）终端，你可以使用如下的语句：

```
dud@shadowplay:~ $ ls > file4.txt
dud@shadowplay:~ $ cat file4.txt
file1.txt
file2.pdf
file3.mp3
file1.pdf
another_file.txt
Yet-Another_file.txt
file-with_other-NAME.TXT
file4.txt
```

以上例子将创建文件 file4.txt 如果 file4.txt 不存在的话。**注意：**如果 file4.txt 已经存在，那么上面的命令将复盖文件的内容。如果你想将内容添加到已存在的文件内容的最后，那你可以用下面这个语句：

- **command >> filename**

示例:

```
dud@shadowplay:~ $ ls >> file4.txt
dud@shadowplay:~ $ cat file4.txt
file1.txt
file2.pdf
file3.mp3
file1.pdf
another_file.txt
Yet-Another_file.txt
file-with_other-NAME.TXT
file4.txt
file1.txt
file2.pdf
file3.mp3
file1.pdf
another_file.txt
Yet-Another_file.txt
file-with_other-NAME.TXT
```

```
file4.txt
```

在这个例子中，你会发现原有的文件中添加了新的内容。接下来我们会见到另一种重定向方式：我们将把一个文件的内容作为将要执行的命令的输入。以下是这个语句：

- **command < filename**

示例:

```
dud@shadowplay:~ $ sort < file4.txt
another_file.txt
another_file.txt
file1.txt
file1.txt
file2.pdf
file2.pdf
file3.mp3
file3.mp3
file4.txt
file4.txt
file-with_other-NAME.TXT
file-with_other-NAME.TXT
Yet-Another_file.txt
Yet-Another_file.txt
```

(3)**管道**

Linux 的强大之处在于它能把几个简单的命令联合成为复杂的功能，通过键盘上的管道符号'|' 完成。现在，我们来排序上面的"grep"命令：

```
grep -i command < myfile | sort > result.text
```

搜索 myfile 中的命令，将输出分类并写入分类文件到 result.text 。 有时候用 ls 列出很多命令的时候很不方便 这时“｜”就充分利用到了 ls -l | less 慢慢看吧.

(4)后台进程

CLI 不是系统的串行接口。您可以在执行其他命令时给出系统命令。要启动一个进程到后台，追加一个“&”到命令后面。

```
sleep 60 &
ls
```

睡眠命令在后台运行，您依然可以与计算机交互。除了不同步启动命令以外，最好把 '&' 理解成 ';'。如果您有一个命令将占用很多时间，您想把它放入后台运行，也很简单。只要在命令运行时按下 *ctrl-z*，它就会停止。然后键入 bg 使其转入后台。fg 命令可使其转回前台。

```
sleep 60
<ctrl-z>
bg
fg
```

最后，您可以使用 *ctrl-c 来杀死一个前台进程。*

环境变量

特殊变量。PATH, PS1, ...

2.2.3.2.3.3不显示中文

可通过执行如下命令避免显示乱码中文。在一个 shell 中，执行：

export LANG=””

这样在这个 shell 中，output 信息缺省时英文。

2.2.3.3获得软件包

2.2.3.3.1命令行获取软件包

Ubuntu 下可以使用 apt-get 命令，apt-get 是一条 Linux 命令行命令，适用于 deb 包管理式的操作系统，主要用于自动从互联网软件库中搜索、安装、升级以及卸载软件或者操作系统。一般需要 root 执行权限，所以一般跟随 sudo 命令，如：

```
sudo apt-get install gcc-3.4 [ENTER]
```

常见的以及常用的 apt 命令有：

apt-get install <package>

下载 <package> 以及所依赖的软件包，同时进行软件包的安装或者升级。

apt-get remove <package>

移除 <package> 以及所有依赖的软件包。

apt-cache search <pattern>

搜索满足 <pattern> 的软件包。

apt-cache show/showpkg <package>

显示软件包 <package> 的完整描述。

例如：

```
chy@laptop:~ $apt-cache gcc-4.3
gcc-4.3 - The GNU C compiler
gcc-4.3-base - The GNU Compiler Collection (base package)
gcc-4.3-doc - Documentation for the GNU compilers (gcc, gobjc, g++)
gcc-4.3-multilib - The GNU C compiler (multilib files)
gcc-4.3-source - Source of the GNU Compiler Collection
gcc-4.3-locales - The GNU C compiler (native language support files)
chy@laptop:~ $
```

2.2.3.3.2图形界面软件包获取

新立得软件包管理器，是 Ubuntu 下面管理软件包得图形界面程序，相当于命令行中得 apt 命令。进入方法可以是

```
菜单栏 > 系统管理 > 新立得软件包管理器 (System>Administration>Synaptic Package Manager)
```

使用更新管理器可以通过标记选择适当的软件包进行更新操作。

2.2.3.3.3配置升级源

Ubuntu 的软件包获取依赖升级源，可以通过修改 “/etc/apt/sources.list” 文件来修改升级源（需要 root 权限）；或者修改新立得软件包管理器中 “设置 > 软件库”。

2.2.3.4查找帮助文件

Ubuntu 下提供 man 命令以完成帮助手册得查询。man 是 manual 的缩写，通过 man 命令可以对 Linux 下常用命令、安装软件、以及 C 语言常用函数等进行查询，获得相关帮助。例如：

```
chy@laptop:~  $man printf
PRINTF(1)                  BSD General Commands Manual                  PRINTF(1)

NAME
     printf -- formatted output

SYNOPSIS
     printf format [arguments ...]

DESCRIPTION
     The printf utility formats and prints its arguments, after the first, under control of the format. The format is a character string
which contains three types of objects: plain characters, which are simply copied to standard output, character escape sequences
which are converted and copied to the standard output, and format specifications, each of which causes ...
```

```
...
The characters and their meanings are as follows:
        \e      Write an <escape> character.
        \a      Write a <bell> character.
...
```

通常可能会用到的帮助文件例如：

```
gcc-doc cpp-doc glibc-doc
```

上述帮助文件可以通过 apt-get 命令或者软件包管理器获得。获得以后可以通过 man 命令进行命令或者参数查询。

2.2.3.5 实验中可能使用的软件

2.2.3.5.1 编辑器

(1) Ubuntu 下自带的编辑器可以作为代码编辑的工具。例如 gedit 是 gnome 桌面环境下兼容 UTF-8 的文本编辑器。它十分的简单易用，有良好的语法高亮，对中文支持很好。通常可以通过双击或者命令行打开目标文件进行编辑。

(2) Vim 编辑器：Vim 是一款极方便的文本编辑软件，是 UNIX 下的同类型软件 VI 的改进版本。Vim 经常被看作是“专门为程序员打造的文本编辑器”，功能强大且方便使用，便于进行程序开发。

Ubuntu 下默认安装的 vi 版本较低，功能较弱，建议在系统内安装或者升级到最新版本的 Vim。

1.关于 Vim 的常用命令以及使用，可以通过网络进行查找，例如 百度百科 相关词条。

2.配置文件：Vim 的使用需要配置文件进行设置，例如：

```
set nocompatible
set encoding=utf-8
set fileencodings=utf-8,chinese
set tabstop=4
set cindent shiftwidth=4
set backspace=indent,eol,start
autocmd Filetype c set omnifunc=ccomplete#Complete
autocmd Filetype cpp set omnifunc=cppcomplete#Complete
set incsearch
set number
set display=lastline
set ignorecase
syntax on
set nobackup
set ruler
set showcmd
set smartindent
set hlsearch
set cmdheight=1
set laststatus=2
set shortmess=atI
set formatoptions=tcrqn
set autoindent
```

可以将上述配置文件保存到：

```
~/.vimrc
```

注意：.vimrc 默认情况下隐藏不可见，可以在命令行中通过 “ls -a” 命令进行查看。如果 '~' 目录下不存在该文件，可以手动创建。修改该文件以后，重启 Vim 可以使配置生效。

2.2.3.5.2 exuberant-ctags：

exuberant-ctags 可以为程序语言对象生成索引，其结果能够被一个文本编辑器或者其他工具简捷迅速的定位。支持的编辑器有 Vim、Emacs 等。

实验中，可以使用命令：

```
ctags -h=.h.c.S -R
```

等类似命令对工程文件建立索引。

默认的生成文件为 tags (可以通过 -f 来指定)，在相同路径下使用 Vim 可以使用改索引文件，例如：使用“ctrl＋]”可以跳转到相应的声明或者定义处，使用“ctrl＋t”返回（查询堆栈）等。

2.2.3.5.3 diff & patch

diff 为 Linux 命令，用于比较文本或者文件夹差异，可以通过 man 来查询其功能以及参数的使用。使用 patch 命令可以对文件或者文件夹应用修改。

例如实验中可能会在 projb 中应用前一个实验 proja 中对文件进行的修改，可以使用如下命令：

```
diff -r -u -P proja_original proja_mine > diff.patch
cd projb
patch -p1 -u < ../diff.patch
```

注意：proja_original 指 proja 的源文件，即未经修改的源码包，proja_mine 是修改后的代码包。第一条命令是递归的比较文件夹差异，并将结果重定向输出到 diff.patch 文件中；第三条命令是将 proja 的修改应用到 projb 文件夹中的代码中。

2.2.4安装硬件模拟器 QEMU

2.2.4.1Linux 运行环境

QEMU 用于模拟一台 x86 计算机，让 ucore 能够运行在 QEMU 上。为了能够正确的编译和安装 qemu，需要使用最新版本的 qemu（或者 os ftp 服务器上提供的 qemu 源码：qemu-0.13.0.tar.gz）。目前 qemu 能够支持最新的 gcc-4.x 编译器。

例如：在 Ubuntu 10.10 系统中，默认得版本是 gcc-4.4.x (可以通过 gcc -v 或者 gcc --version 进行查看)。

此外，编译 qemu 还会用到的库文件有 **libsdl1.2-dev** 等。编译安装命令如下：

```
configure --target-list="i386-softmmu"
make
make install
```

qemu 执行程序将缺省安装到 /usr/local/bin 目录下。

2.2.4.2Linux 环境下的安装过程

2.2.4.2.1获得并应用修改

获得 qemu 的安装包以后，对其进行解压缩(如果格式无法识别，请下载相应的解压缩软件)。

例如 qemu.tar.gz/qemu.tar.bz2 文件，在命令行中可以使用：

```
tar zxvf qemu.tar.gz
或者  tar jxvf qemu.tar.bz2
```

对 qemu 应用修改：如果实验中使用的 qemu 需要打 patch，应用过程如下所示：

```
chy@laptop:~ $ls
qemu.patch      qemu
chy@laptop:~ $cd qemu
chy@laptop:~ $patch -p1 -u < ../qemu.patch
```

2.2.4.2.2配置、编译和安装

编译以及安装 qemu 前需要使用 <qemu>(表示 qemu 解压缩路径)下面的 configure 脚本生成相应的配置文件等。而 configure 脚本有较多的参数可供选择，可以通过如下命令进行查看：

```
configure --help
```

实验中可能会用到的命令例如：

```
configure --target-list="i386-softmmu"
make
sudo make install
```

注意：版本小于 0.10.0 的 qemu 仅支持 gcc-3.x 版本编译器。但 0.10.x 以上版本的 qemu 已经支持用 gcc-4.x 编译器了。

如果使用的是默认的安装路径，那么在 “/usr/local/bin” 下面即可看到安装结果：

```
qemu qemu-img qemu-nbd
```

2.2.5硬件模拟器 QEMU 介绍

2.2.5.1运行参数

如果 qemu 使用的是默认 /usr/local/bin 安装路径，则在命令行中可以直接使用 qemu 命令运行程序。qemu 运行可以有多参数，格式如：

```
qemu [options] [disk_image]
```

其中 disk_image 即硬盘镜像文件。

部分参数说明：

`-hda file'  `-hdb file'`-hdc file'`-hdd file'

使用 file 作为硬盘 0、1、2、3 镜像。

`-fda file' `-fdb file'

使用 file 作为软盘镜像，可以使用 /dev/fd0 作为 file 来使用主机软盘。

`-cdrom file'

使用 file 作为光盘镜像，可以使用 /dev/cdrom 作为 file 来使用主机 cd-rom。

`-boot [a|c|d]'

从软盘(a)、光盘(c)、硬盘启动(d)，默认硬盘启动。

`-snapshot'

写入临时文件而不写回磁盘镜像，可以使用 C-a s 来强制写回。

`-m megs'

设置虚拟内存为 msg M 字节，默认为 128M 字节。

`-smp n'

设置为有 n 个 CPU 的 SMP 系统。以 PC 为目标机，最多支持 255 个 CPU。

`-nographic'

禁止使用图形输出。

其他：

可用的主机设备 dev 例如：

vc

虚拟终端。

null

空设备

/dev/XXX

使用主机的 tty。

file: filename

将输出写入到文件 filename 中。

stdio

标准输入/输出。

pipe：pipename

命令管道 pipename。

等。

使用 dev 设备的命令如：

`-serial dev'

重定向虚拟串口到主机设备 dev 中。

`-parallel dev'

重定向虚拟并口到主机设备 dev 中。

`-monitor dev'

重定向 monitor 到主机设备 dev 中。

其他参数：

`-s'

等待 gdb 连接到端口 1234。

`-p port'

改变 gdb 连接端口到 port。

`-S'

在启动时不启动 CPU， 需要在 monitor 中输入 'c'，才能让 qemu 继续模拟工作。

`-d'

输出日志到 qemu.log 文件。

其他参数说明可以参考：http://bellard.org/qemu/qemu-doc.html#SEC15 。其他 qemu 的安装和使用的说明可以参考 http://bellard.org/qemu/user-doc.html。

或者在命令行收入 qemu (没有参数) 显示帮助。

在实验中，例如 lab1，可能用到的命令如：

```
qemu -hda ucore.img -parallel stdio
```

或

```
qemu -S -s -hda ucore.img -monitor stdio
```

2.2.6 ucore 代码编译

(1)编译过程：在解压缩后的 ucore 源码包中使用 make 命令即可。例如 lab1 中的 proj1 中：

```
chy@laptop: ~/proj1$  make
```

生成目标文件为 ucore.img。

(2)保存修改：

使用 diff 命令对修改后的 ucore 代码和 ucore 源码进行比较，比较之前建议使用 make clean 命令清除不必要文件。(如果有 ctags 文件，需要手工清除。)

(3)应用修改：参见 patch 命令说明。

2.2.7 ucore 运行调试

(1)qemu 中 monitor 的常用命令：

help	查看 qemu 帮助，显示所有支持的命令。
q|quit|exit	退出 qemu。
stop	停止 qemu。
c|cont|continue	连续执行。
x /fmt addr xp /fmt addr	显示内存内容，其中 'x' 为虚地址，'xp' 为实地址。 参数 /fmt i 表示反汇编，缺省参数为前一次参数。

p|print'	计算表达式值并显示，例如 $reg 表示寄存器结果。
memsave addr size file pmemsave addr size file	将内存保存到文件，memsave 为虚地址，pmemsave 为实地址。
breakpoint 相关：	设置、查看以及删除 breakpoint，pc 执行到 breakpoint，qemu 停止。（暂时没有此功能）
watchpoint 相关：	设置、查看以及删除 watchpoint，当 watchpoint 地址内容被修改，停止。（暂时没有此功能）
s|step	单步一条指令，能够跳过断点执行。
r|registers	显示全部寄存器内容。
info 相关操作	查询 qemu 支持的关于系统状态信息的操作。

其他具体的命令格式以及说明，参见 qemu help 命令帮助。
注意：qemu 默认有‘singlestep arg’命令（arg 为 参数），该命令为设置单步标志命令。例如：'singlestep 0' 运行结果为禁止单步，'singlestep + 非 0' 结果为允许单步。在允许单步条件下，使用 cont 命令进行单步操作。如：

```
(qemu) xp /3i $pc
0xfffffff0: ljmp        $0xf000, $0xe05b
0xfffffff5: xor   %bh, (%bx, %si)
0xfffffff7: das
(qemu) singlestep 2
(qemu) cont
                0x000fe05b: xor %ax, %ax
```

step 命令为单步命令，即 qemu 执行一步，能够跳过 breakpoint 断点执行。如果此时使用 cont 命令，则 qemu 运行改为连续执行。

log 命令能够保存 qemu 模拟过程产生的信息（与 qemu 运行参数 `-d' 相同），具体参数可以参考命令帮助。产生的日志信息保存在 “/tmp/qemu.log” 中，例如使用 'log in_asm'命令以后，运行过程产生的的 qemu.log 文件为：

```
 1 ----------------
 2 IN:
 3 0xfffffff0:    ljmp     $0xf000,$0xe05b
 4
 5 ----------------
 6 IN:
 7 0x000fe05b:    xor      %ax,%ax
 8 0x000fe05d:    out      %al,$0xd
 9 0x000fe05f:    out      %al,$0xda
10 0x000fe061:    mov      $0xc0,%al
11 0x000fe063:    out      %al,$0xd6
12 0x000fe065:    mov      $0x0,%al
13 0x000fe067:    out      %al,$0xd4
```

2.2.7.1 qemu 本地调试 ucore

调试举例：调试 proj1，跟踪 bootmain 函数：

(1) 运行 qemu -S -hda ucore.img -monitor stdio
(2) 查看 bootblock.asm 得到 bootmain 函数地址为 0x7d60，并插入断点。
(3) 使用命令 c 连续执行到断点。
(4) 使用 xp 命令进行反汇编。
(5) 使用 s 命令进行单步执行。

运行结果如下：

```
chy@laptop: ~/proj1$ qemu -S -hda ucore.img -monitor stdio
(qemu) b 0x7d60
insert breakpoint 0x7d60 success!
(qemu) c
        working …
(qemu)
        break:
                0x00007d60:    push     %ebp
(qemu) xp /10i $pc
        0x00007d60:    push     %ebp
        0x00007d61:    mov      %esp,%ebp
        0x00007d63:    push     %esi
        0x00007d64:    push     %ebx
        0x00007d65:    sub      $0x4,%esp
        0x00007d68:    mov      0x7da8,%esi
        0x00007d6e:    mov      $0x0,%ebx
        0x00007d73:    movsbl (%esi,%ebx,1),%eax
        0x00007d77:    mov      %eax,(%esp,1)
        0x00007d7a:    call     0x7c6c
(qemu) step
                0x00007d61:    mov      %esp,%ebp
(qemu) step
                0x00007d63:    push     %esi
```

2.2.7.2 gdb 远地调试 ucore

通过 gdb 可以对 ucore 代码进行调试，以 lab3 中 main 函数中 memset 为例：

(1) 运行 qemu -S -s -hda ucore.img -monitor stdio

(2) 运行 gdb 并与 qemu 进行连接

(3) 设置断点并执行

(4) qemu 单步调试。

运行过程以及结果如下：

窗口一

```
chy@laptop: ~/proj3$ qemu -S -hda ./bin/ucore.img    -s
```

窗口二

```
chy@laptop: ~/proj3$ gdb ./bin/kernel
(gdb) target remote:1234
Remote debugging using :1234
0x0000fff0 in ?? ()
(gdb) list 20
15              const char *message = "(THU.CST) os is loading ...";
16              cprintf("[%d]%s\n", strlen(message), message);
17
18              while (1);
19      }
20        (gdb) break memset
(gdb) break memset
Breakpoint 1 at 0x100d9f: file libs/string.c, line 54.
(gdb) run
Starting                                                program:
/home/chenyu/oscourse/develop/ucore/lab1_boot_interrupt/proj3
/bin/kernel

Breakpoint 1, memset (s=0x1020fc, c=0 '\000', n=12) at
libs/string.c:54
54              return __memset(s, c, n);
(gdb)
```

2.3 了解处理器硬件

要想深入理解 ucore，就需要了解支撑 ucore 运行的硬件环境，即了解处理器体系结构（了解硬件对 ucore 带

来影响）和机器指令集（读懂 ucore 的汇编）。ucore 目前支持的硬件环境是基于 Intel 80386 以上的计算机系统。更多的硬件相关内容（比如保护模式等）将随着实现 ucore 的过程逐渐展开介绍。

2.3.1Intel 80386 运行模式

80386 有四种运行模式：实模式、保护模式、SMM 模式和虚拟 8086 模式。这里对涉及 ucore 的**实模式、保护模式**做一个简要介绍。

实模式：80386 加电启动后处于实模式运行状态，在这种状态下软件可访问的物理内存空间不能超过 1MB，且无法发挥 Intel 80386 以上级别的 32 位 CPU 的 4GB 内存管理能力。实模式将整个物理内存看成分段的区域，程序代码和数据位于不同区域，操作系统和用户程序并没有区别对待，而且每一个指针都是指向实际的物理地址。这样用户程序的一个指针如果指向了操作系统区域或其他用户程序区域，并修改了内容，那么其后果就很可能是灾难性的。

实模式：实模式是为了和 8086 处理器兼容而设置的。在实模式下，80386 处理器就相当于一个快速的 8086 处理器。80386 处理器被复位或加电的时候以实模式启动。这时候处理器中的各寄存器以实模式的初始化值工作。80386 处理器在实模式下的存储器寻址方式和 8086 是一样的，由段寄存器的内容乘以 16 当做基地址，加上段内的偏移地址形成最终的物理地址，这时候它的 32 位地址线只使用了低 20 位，即可访问 1MB 的物理地址空间。在实模式下，80386 处理器不能对内存进行页机制管理，所以指令寻址的地址就是内存中实际的物理地址。在实模式下，所有的段都是可以读、写和执行的。实模式下 80386 不支持优先级，所有的指令相当于工作在特权级（即优先级 0），所以它可以执行所有特权指令，包括读写控制寄存器 CR0 等。实模式下不支持硬件上的多任务切换。实模式下的中断处理方式和 8086 处理器相同，也用中断向量表来定位中断服务程序地址。中断向量表的结构也和 8086 处理器一样，每 4 个字节组成一个中断向量，其中包括两个字节的段地址和两个字节的偏移地址。

保护模式：实际上，80386 就是通过在实模式下初始化控制寄存器，GDTR，LDTR，IDTR 与 TR 等管理寄存器以及页表，然后再通过加载 CR0 使其中的保护模式使能位置位而进入保护模式的。当 80386 工作在保护模式下的时候，其所有的 32 根地址线都可供寻址，物理寻址空间高达 4GB。在保护模式下，支持内存分页机制，提供了对虚拟内存的良好支持。保护模式下 80386 支持多任务，还支持优先级机制，不同的程序可以运行在不同的优先级上。优先级一共分 0～3 4 个级别，操作系统运行在最高的优先级 0 上，应用程序则运行在比较低的级别上；配合良好的检查机制后，既可以在任务间实现数据的安全共享也可以很好地隔离各个任务。

2.3.2Intel 80386 内存架构

80386 是 32 位的处理器，即可以寻址的物理内存地址空间为 2^32=4G 字节。在理解操作系统的过程中，需要用到三个地址空间的概念。地址是访问地址空间的索引。**物理内存地址空间**是处理器提交到总线上用于访问计算机系统中的内存和外设的最终地址。一个计算机系统中只有一个物理地址空间。**线性地址空间**是每个运行的应用程序看到的地址空间，在操作系统的虚存管理之下，每个运行的应用程序都认为自己独享整个计算机系统的地址空间，这样可让多个运行的应用程序之间相互隔离。处理器负责把线性地址转换成物理地址。一个计算机系统中可以有多个线性地址空间（比如一个运行的程序就可以有一个私有的线性地址空间）。线性地址空间的大小与物理地址空间的大小没有必然的连续。**逻辑地址空间**是应用程序直接使用的地址空间。这是由于 80386 中无法禁用段机制，使得逻辑地址一直存在。比如如下 C 代码片段：

```
int boo=1;
int *foo=&a;
```

这里的 boo 是一个整型变量，foo 变量是一个指向 boo 地址的整型指针变量，foo 中储存的内容就是 boo 的逻辑地址。逻辑地址由一个 16 位的段寄存器和一个 32 位的偏移量构成。Foo 中放的就是 32 位的偏移量，而对应的段信息位于段寄存器中。

上述三种地址的关系如下：

- 分段机制启动、分页机制未启动：逻辑地址->***段机制处理***->线性地址=物理地址
- 分段机制和分页机制都启动：逻辑地址->***段机制处理***->线性地址->***页机制处理***->物理地址

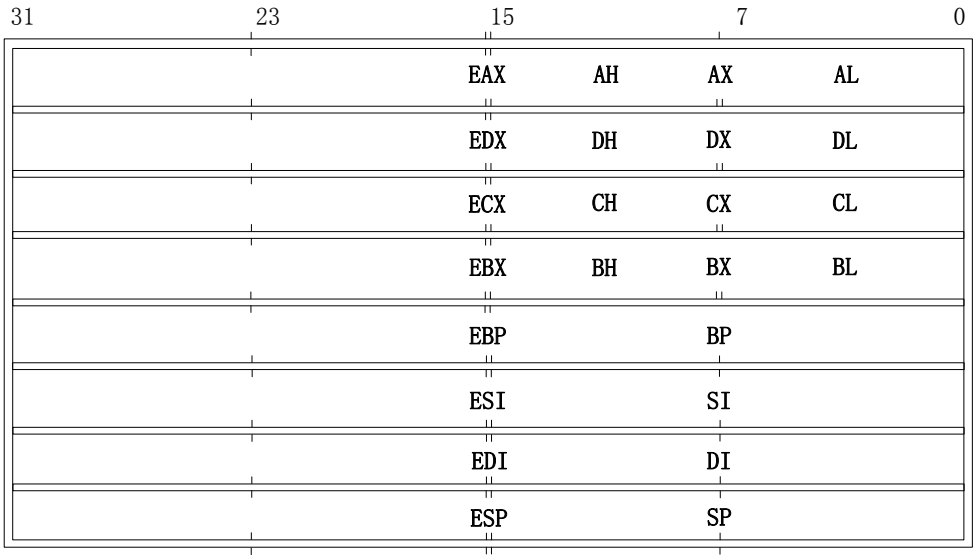
2.3.3Intel 80386 寄存器

80386 的寄存器可以分为 8 组：通用寄存器，段寄存器，指令指针寄存器，标志寄存器，系统地址寄存器，控制寄存器，调试寄存器，测试寄存器，它们的宽度都是 32 位。一般程序员看到的寄存器包括通用寄存器，段寄存器，指令指针寄存器，标志寄存器。

General Register(通用寄存器)：EAX/EBX/ECX/EDX/ESI/EDI/ESP/EBP 这些寄存器的低 16 位就是 8086 的 AX/BX/CX/DX/SI/DI/SP/BP，对于 AX,BX,CX,DX 这四个寄存器来讲,可以单独存取它们的高 8 位和低 8 位(AH,AL,BH,BL,CH,CL,DH,DL)。它们的含义如下:

EAX：累加器
EBX：基址寄存器
ECX：计数器
EDX：数据寄存器
ESI：源地址指针寄存器
EDI：目的地址指针寄存器
EBP：基址指针寄存器
ESP：堆栈指针寄存器

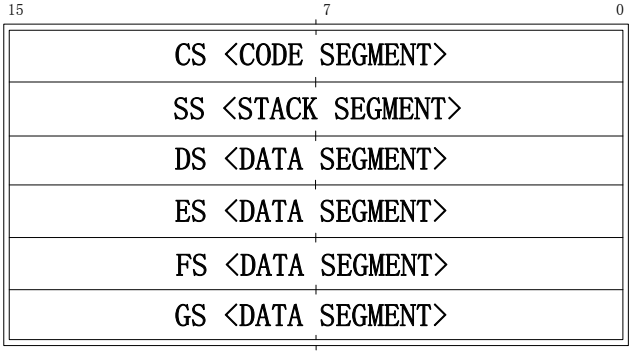
通用寄存器



Segment Register(段寄存器，也称 Segment Selector，段选择符，段选择子)：除了 8086 的 4 个段外(CS,DS,ES,SS)，80386 还增加了两个段 FS，GS,这些段寄存器都是 16 位的，它们的含义如下：

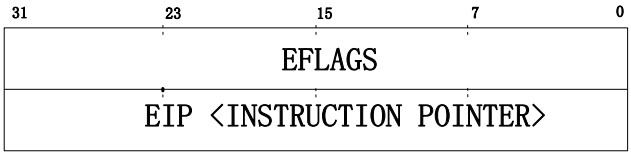
CS：代码段(Code Segment)
DS：数据段(Data Segment)
ES：附加数据段(Extra Segment)
SS：堆栈段(Stack Segment)
FS：附加段
GS 附加段

段寄存器



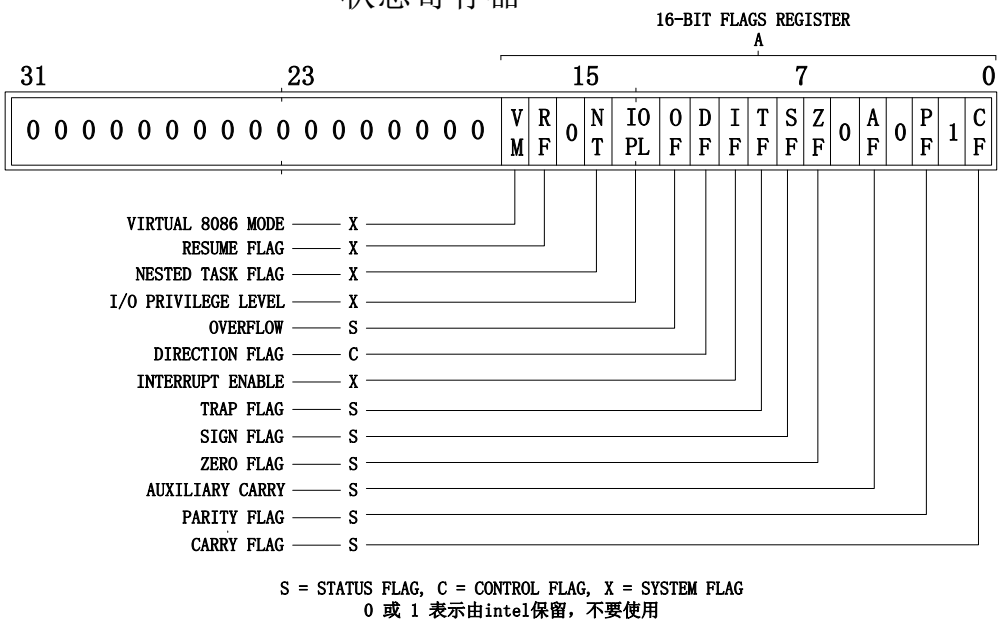
Instruction Pointer(指令指针寄存器)：EIP 的低 16 位就是 8086 的 IP，它存储的是下一条要执行指令的内存地址，在分段地址转换中，表示指令的段内偏移地址。

状态寄存器和指令寄存器



Flag Register(标志寄存器)：EFLAGS,和 8086 的 16 位标志寄存器相比，增加了 4 个控制位，这 20 位控制/标志位的位置如下图所示：

状态寄存器



相关的控制/标志位含义是：

CF(Carry Flag)：进位标志位；

PF(Parity Flag)：奇偶标志位；

AF(Assistant Flag)：辅助进位标志位；

ZF(Zero Flag)：零标志位；

SF(Singal Flag)：符号标志位；

IF(Interrupt Flag)：中断允许标志位,由 CLI，STI 两条指令来控制；设置 IF 使 CPU 可识别外部（可屏蔽）中断请求。复位 IF 则禁止中断。 IF 对不可屏蔽外部中断和故障中断的识别没有任何作用。

DF(Direction Flag)：向量标志位，由 CLD，STD 两条指令来控制；

OF(Overflow Flag)：溢出标志位；

IOPL(I/O Privilege Level)：I/O 特权级字段，它的宽度为 2 位,它指定了 I/O 指令的特权级。

如果当前的特权级别在数值上小于或等于 IOPL，那么 I/O 指令可执行。否则，将发生一个保护性故障中断。

NT(Nested Task)：控制中断返回指令 IRET，它宽度为 1 位。若 NT=0，则用堆栈中保存的值恢复 EFLAGS，CS 和 EIP 从而实现中断返回；若 NT=1，则通过任务切换实现中断返回。

2.4 熟悉编程

在 Ubuntu Linux 中的 C 语言编程主要基于 GNU C 的语法，通过 gcc 来编译并生成最终执行文件。GNU 汇编（assembler）采用的是 AT&T 汇编格式，Microsoft 汇编采用 Intel 格式。

2.4.1gcc 的基本用法

如果你还没装 gcc 编译环境或自己不确定装没装，不妨先执行 ：

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

2.4.1.1 编译简单的 C 程序

C 语言经典的入门例子是 **Hello World**，下面是一示例代码：

```
#include <stdio.h>
int
main(void)
{
    printf("Hello, world!\n");
    return 0;
}
```

我们假定该代码存为文件‘hello.c’。要用 **gcc** 编译该文件，使用下面的命令：

```
$ gcc -Wall hello.c -o hello
```

该命令将文件‘hello.c’中的代码编译为机器码并存储在可执行文件 ‘hello’中。机器码的文件名是通过 **-o** 选项指定的。该选项通常作为命令行中的最后一个参数。如果被省略，输出文件默认为 ‘a.out’。

注意到如果当前目录中与可执行文件重名的文件已经存在，它将被复盖。

选项 **-Wall** 开启编译器几乎所有常用的警告**——强烈建议你始终使用该选项**。编译器有很多其他的警告选项，但 **-Wall** 是最常用的。默认情况下 GCC 不会产生任何警告信息。当编写 C 或 C++ 程序时编译器警告非常有助于检测程序存在的问题。

本例中，编译器使用了 *-Wall* 选项而没产生任何警告，因为示例程序是完全合法的。

要运行该程序，输入可执行文件的路径如下：

```
$ ./hello
Hello, world!
```

这将可执行文件载入内存，并使 CPU 开始执行其包含的指令。 路径 **./** 指代当前目录，因此 **./hello** 载入并执行当前目录下的可执行文件 ‘hello’。

2.4.1.2AT&T 汇编基本语法

Ucore 中用到的是 AT&T 格式的汇编，与 Intel 格式的汇编有一些不同。二者语法上主要有以下几个不同：

* 寄存器命名原则
  AT&T: %eax　　Intel: eax
* 源/目的操作数顺序
  AT&T: movl %eax, %ebx　　Intel: mov ebx, eax
* 常数/立即数的格式
  AT&T: movl $_value, %ebx　　Intel: mov eax, _value
  把 value 的地址放入 eax 寄存器

AT&T: movl $0xd00d, %ebx　　Intel: mov ebx, 0xd00d

* 操作数长度标识

AT&T: movw %ax, %bx　　Intel: mov bx, ax

* 寻址方式

AT&T: immed32(basepointer, indexpointer, indexscale)

Intel: [basepointer + indexpointer ×indexscale + imm32)

如果操作系统工作于保护模式下，用的是 32 位线性地址，所以在计算地址时不用考虑 segment:offset 的问题。上式中的地址应为：

imm32 + basepointer + indexpointer ×indexscale

下面是一些例子：

- o 直接寻址

  AT&T: _boo ；_boo 是一个全局的 C 变量。注意加上$是表示地址引用，不加是表示值引用。对于局部变量，可以通过堆栈指针引用。

  Intel: [_boo]
- o 寄存器间接寻址

  AT&T: (%eax)

  Intel: [eax]
- o 变址寻址

  AT&T: _variable(%eax)

  Intel: [eax + _variable]

  AT&T: _array(,%eax, 4)

  Intel: [eax ×4 + _array]

  AT&T: _array(%ebx, %eax,8)

  Intel: [ebx + eax ×8 + _array]

2.4.1.3GCC 内联汇编

基本的 GCC 内联汇编很简单，一般是按照下面的格式：

```
asm("statements");
```

例如：

```
asm("nop"); asm("cli");
```

"asm" 和 "__asm__" 的含义是完全一样的。如果有多行汇编，则每一行都要加上 "\n\t"。其中的“\n” 是换行符，"\t” 是 tab 符，在每条命令的 结束加这两个符号，是为了让 gcc 把内联汇编代码翻译成一般的汇编代码时能够保证换行和留有一定的空格。例如：

```
asm( "pushl %eax\n\t"
     "movl $0,%eax\n\t"
     "popl %eax"
   );
```

实际上 gcc 在处理汇编时，是要把 asm(...)的内容"打印"到汇编文件中，所以格式控制字符是必要的。再例如：

```
asm("movl %eax, %ebx");
asm("xorl %ebx, %edx");
asm("movl $0, _boo);
```

在上面的例子中，由于我们在内联汇编中改变了 edx 和 ebx 的值，但是由于 gcc 的特殊的处理方法，即先形成汇编文件，再交给 GAS 去汇编，所以 GAS 并不知道我们已经改变了 edx 和 ebx 的值，如果程序的上下文需要 edx 或 ebx 作暂存，这样就会引起严重的后果。对于变量 _boo 也存在一样的问题。为了解决这个问题，就要用到扩展 GCC 内联汇编语法。

2.4.1.4扩展 GCC 内联汇编

使用扩展 GCC 内联汇编的例子如下：

```
#define read_cr0() ({ \
```

```
        unsigned int __dummy; \
        __asm__( \
                "movl %%cr0,%0\n\t" \
                :"=r" (__dummy)); \
        __dummy; \
})
```

它代表什么含义呢？这需要从其基本格式讲起。扩展 GCC 内联汇编的基本格式是：

```
__asm__ __volatile__ ("<asm routine>" : output : input : modify);
```

其中，__asm__ 表示汇编代码的开始，其后可以跟 __volatile__（这是可选项），其含义是避免“asm” 指令被删除、移动或组合，在执行代码时，如果不希望汇编语句被 gcc 优化而改变位置，就需要在 asm 符号后添加 volatile 关键词：asm volatile(...)；或者更详细地说明为：__asm__ __volatile__(...)；然后就是小括弧，括弧中的内容是具体的内联汇编指令代码。 "<asm routine>" 为汇编指令部分，例如，"movl %%cr0,%0\n\t"。数字前加前缀 “%“，如%1，%2 等表示使用寄存器的样板操作数。可以使用的操作数总数取决于具体 CPU 中通用寄存器的数 量，如 Intel 可以有 8 个。指令中有几个操作数，就说明有几个变量需要与寄存器结合，由 gcc 在编译时根据后面输出部分和输入部分的约束条件进行相应的 处理。由于这些样板操作数的前缀使用了”%“，因此，在用到具体的寄存器时就在前面加两个“%”，如%%cr0。输出部分（output），用以规定对输出变量（目标操作数）如何与寄存器结合的约束（constraint）,输出部分可以有多个约束，互相以逗号分开。每个约束以“＝”开头，接着用一个字母来表示操作数的类型，然后是关于变量结合的约束。例如，上例中：

```
:"=r" (__dummy)
```

“＝r”表示相应的目标操作数（指令部分的%0）可以使用任何一个通用寄存器，并且变量__dummy 存放在这个寄存器中，但如果是：

```
:“＝m”(__dummy)
```

“＝m”就表示相应的目标操作数是存放在内存单元__dummy 中。表示约束条件的字母很多，下表给出几个主要的约束字母及其含义：

字母	含义
m, v, o	内存单元
R	任何通用寄存器
Q	寄存器 eax, ebx, ecx,edx 之一
I, h	直接操作数
E, F	浮点数
G	任意
a, b, c, d	寄存器 eax/ax/al, ebx/bx/bl, ecx/cx/cl 或 edx/dx/dl
S, D	寄存器 esi 或 edi
I	常数（0～31）

输入部分（Input）：输入部分与输出部分相似，但没有“＝”。如果输入部分一个操作数所要求使用的寄存器，与前面输出部分某个约束所要求的是同一个寄存器，那就把对应操作数的编号（如“1”，“2”等）放在约束条件中，在后面的例子中，我们会看到这种情况。修改部分（modify）:这部分常常以“memory”为约束条件，以表示操作完成后内存中的内容已有改变，如果原来某个寄存器的内容来自内存，那么现在内存中这个单元的内容已经改变。 注意，指令部分为必选项，而输入部分、输出部分及修改部分为可选项，当输入部分存在，而输出部分不存在时，分号“：“要保留，当“memory”存在时，三个分号都要保留，例如

```
#define __cli() __asm__ __volatile__("cli": : :"memory")
```

下面是一个例子(为方便起见，我使用全局变量）：

```
int count=1;
int value=1;
int buf[10];
void main()
{
        asm(
                "cld nt"
                "rep nt"
                "stosl"
              :
              : "c" (count), "a" (value) , "D" (buf[0])
              : "%ecx","%edi"
              );
}
```

得到的主要汇编代码为：

```
movl count,%ecx
movl value,%eax
movl buf,%edi
#APP
cld
rep
stosl
#NO_APP
```

cld,rep,stos 就不用多解释了。这几条语句的功能是向 buf 中写上 count 个 value 值。冒号后的语句指明输入，输出和被改变的寄存器。通过冒号以后的语句，编译器就知道你的指令需要和改变哪些寄存器，从而可以优化寄存器的分配。其中符号"c"(count)指示要把 count 的值放入 ecx 寄存器。类似的还有：

```
a eax
b ebx
c ecx
d edx
S esi
D edi
I 常数值，(0 - 31)
q,r 动态分配的寄存器
g eax,ebx,ecx,edx 或内存变量
A 把 eax 和 edx 合成一个 64 位的寄存器(use long longs)
```

我们也可以让 gcc 自己选择合适的寄存器。如下面的例子：

```
asm("leal (%1,%1,4),%0"
        : "=r" (x)
        : "0" (x)
);
```

这段代码实现 5 ×x 的快速乘法。得到的主要汇编代码为：

```
movl x,%eax
#APP
leal (%eax,%eax,4),%eax
#NO_APP
movl %eax,x
```

几点说明：

1. 使用 q 指示编译器从 eax, ebx, ecx, edx 分配寄存器。

使用 r 指示编译器从 eax, ebx, ecx, edx, esi, edi 分配寄存器。

2. 我们不必把编译器分配的寄存器放入改变的寄存器列表，因为寄存器已经记住了它们。
3. "="是标示输出寄存器，必须这样用。
4. 数字%n 的用法：数字表示的寄存器是按照出现和从左到右的顺序映射到用"r"或"q"请求的寄

存器．如果我们要重用"r"或"q"请求的寄存器的话，就可以使用它们。

5. 如果强制使用固定的寄存器的话，如不用%1，而用 ebx，则：

```
asm("leal (%%ebx,%%ebx,4),%0"
    : "=r" (x)
    : "0" (x)
    );
```

注意要使用两个%,因为一个%的语法已经被%n 用掉了。

2.4.1.5 make 和 Makefile

GNU make(简称 make)是一种代码维护工具，在大中型项目中，它将根据程序各个模块的更新情况，自动的维护和生成目标代码。

make 命令执行时，需要一个 makefile （或 Makefile）文件，以告诉 make 命令需要怎么样的去编译和链接程序。首先，我们用一个示例来说明 makefile 的书写规则。以便给大家一个感兴认识。这个示例来源于 gnu 的 make 使用手册，在这个示例中，我们的工程有 8 个 c 文件，和 3 个头文件，我们要写一个 makefile 来告诉 make 命令如何编译和链接这几个文件。我们的规则是：

- 如果这个工程没有编译过，那么我们的所有 c 文件都要编译并被链接。
- 如果这个工程的某几个 c 文件被修改，那么我们只编译被修改的 c 文件，并链接目标程序。
- 如果这个工程的头文件被改变了，那么我们需要编译引用了这几个头文件的 c 文件，并链接目标程序。

只要我们的 makefile 写得够好，所有的这一切，我们只用一个 make 命令就可以完成，make 命令会自动智能地根据当前的文件修改的情况来确定哪些文件需要重编译，从而自己编译所需要的文件和链接目标程序。

2.4.1.5.1makefile 的规则

在讲述这个 makefile 之前，还是让我们先来粗略地看一看 makefile 的规则。

```
target ... : prerequisites ...
        command
        ...
        ...
```

target 也就是一个目标文件，可以是 object file，也可以是执行文件。还可以是一个标签（label）。prerequisites 就是，要生成那个 target 所需要的文件或是目标。command 也就是 make 需要执行的命令（任意的 shell 命令）。 这是一个文件的依赖关系，也就是说，target 这一个或多个的目标文件依赖于 prerequisites 中的文件，其生成规则定义在 command 中。说白一点就是说，prerequisites 中如果有一个以上的文件比 target 文件要新的话，command 所定义的命令就会被执行。这就是 makefile 的规则。也就是 makefile 中最核心的内容。

2.4.1.6 GDB 使用

gdb 是功能强大的调试程序，可完成如下的调试任务：

- ●设置断点
- ●监视程序变量的值
- ●程序的单步(step in/step over)执行
- ●显示/修改变量的值
- ●显示/修改寄存器
- ●查看程序的堆栈情况
- ●远程调试
- ●调试线程

在可以使用 gdb 调试程序之前，必须使用 -g 或 –ggdb 编译选项编译源文件。运行 gdb 调试程序时通常使用如下的命令：

```
gdb progname
```

在 gdb 提示符处键入 help，将列出命令的分类，主要的分类有：

- ●aliases：命令别名
- ●breakpoints：断点定义；
- ●data：数据查看；
- ●files：指定并查看文件；
- ●internals：维护命令；
- ●running：程序执行；
- ●stack：调用栈查看；
- ●status：状态查看；
- ●tracepoints：跟踪程序执行。

键入 help 后跟命令的分类名，可获得该类命令的详细清单。gdb 的常用命令如下表所示。

表 gdb 的常用命令

break FILENAME:NUM	在特定源文件特定行上设置断点
clear FILENAME:NUM	删除设置在特定源文件特定行上的断点
run	运行调试程序
step	单步执行调试程序，不会直接执行函数
next	单步执行调试程序，会直接执行函数
backtrace	显示所有的调用栈帧。该命令可用来显示函数的调用顺序
where	
continue	继续执行正在调试的程序
display EXPR	每次程序停止后显示表达式的值,表达式由程序定义的变量组成
file FILENAME	装载指定的可执行文件进行调试
help CMDNAME	显示指定调试命令的帮助信息
info break	显示当前断点列表，包括到达断点处的次数等
info files	显示被调试文件的详细信息
info func	显示被调试程序的所有函数名称
info prog	显示被调试程序的执行状态
info local	显示被调试程序当前函数中的局部变量信息
info var	显示被调试程序的所有全局和静态变量名称
kill	终止正在被调试的程序
list	显示被调试程序的源代码
quit	退出 gdb

下面以一个有错误的例子程序来介绍 gdb 的使用：

```
/*bugging.c*/
#include <stdio.h>
#include <stdlib.h>

static char buff [256];
static char* string;
int main ()
{
    printf ("Please input a string: ");
    gets (string);
    printf ("\nYour string is: %s\n", string);
}
```

上面这个程序非常简单，其目的是接受用户的输入，然后将用户的输入打印出来。该程序使用了一个未经过初始化的字符串地址 string，因此，编译并运行之后，将出现 Segment Fault 错误：

```
$ gcc -o bugging -g    bugging.c
$ ./bugging
Please input a string: asdf
Segmentation fault (core dumped)
```

为了查找该程序中出现的问题，我们利用 gdb，并按如下的步骤进行：

1)运行 “gdb bugging”，加载 bugging 可执行文件；
   $gdb bugging
2)执行装入的 bugging 命令；
   (gdb) run
3)使用 where 命令查看程序出错的地方；
   (gdb) where
4)利用 list 命令查看调用 gets 函数附近的代码；
   (gdb) list
5)在 gdb 中，我们在第 11 行处设置断点，看看是否是在第 11 行出错；
(gdb) break 11
6)程序重新运行到第 11 行处停止，这时程序正常，然后执行单步命令 next；
   (gdb) next
7)程序确实出错，能够导致 gets 函数出错的因素就是变量 string。重新执行测试程，用 print 命令查看 string 的值；
   (gdb) run
   (gdb) print string
   (gdb) $1=0x0
8)问题在于 string 指向的是一个无效指针，修改程序，在 10 行和 11 行之间增加一条语句 “string=buff;”，重新编译程序，然后继续运行，将看到正确的程序运行结果。

2.4.2进一步的相关内容

（请同学网上搜寻相关资料学习）
gcc tools 相关文档
版本管理软件（CVS、SVN、GIT 等）的使用
…