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 硬件模拟器，比如 QEMU、BOCHS、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 等。在分析源代码上，可以采用 Scitools 提供的 understand 软件（跨平台），windows 环境上的 source insight 软件，或者基于 emacs+ctags，vim+ctags 等，都可以比较方便在在一堆文件中查找变量、函数定义、调用/访问关系等。软件开发的版本管理可以采用 GIT、SVN 等。比较文件和目录的不同可发现不同实验中的差异性和进行文件合并操作，可使用 meld、kdiff3、UltraCompare 等软件。调试（deubg）实验有助于发现设计中的错误，可采用 gdb（配合 qemu）等调试工具软件。并可整个实验的运行环境和开发环境既可以在 Linux 或 Windows 中使用。推荐使用 Linux 环境。

那我们准备如何一步一步来实现 ucore 呢？根据一个操作系统的设计实现过程，我们可以有如下的实验步骤：

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：

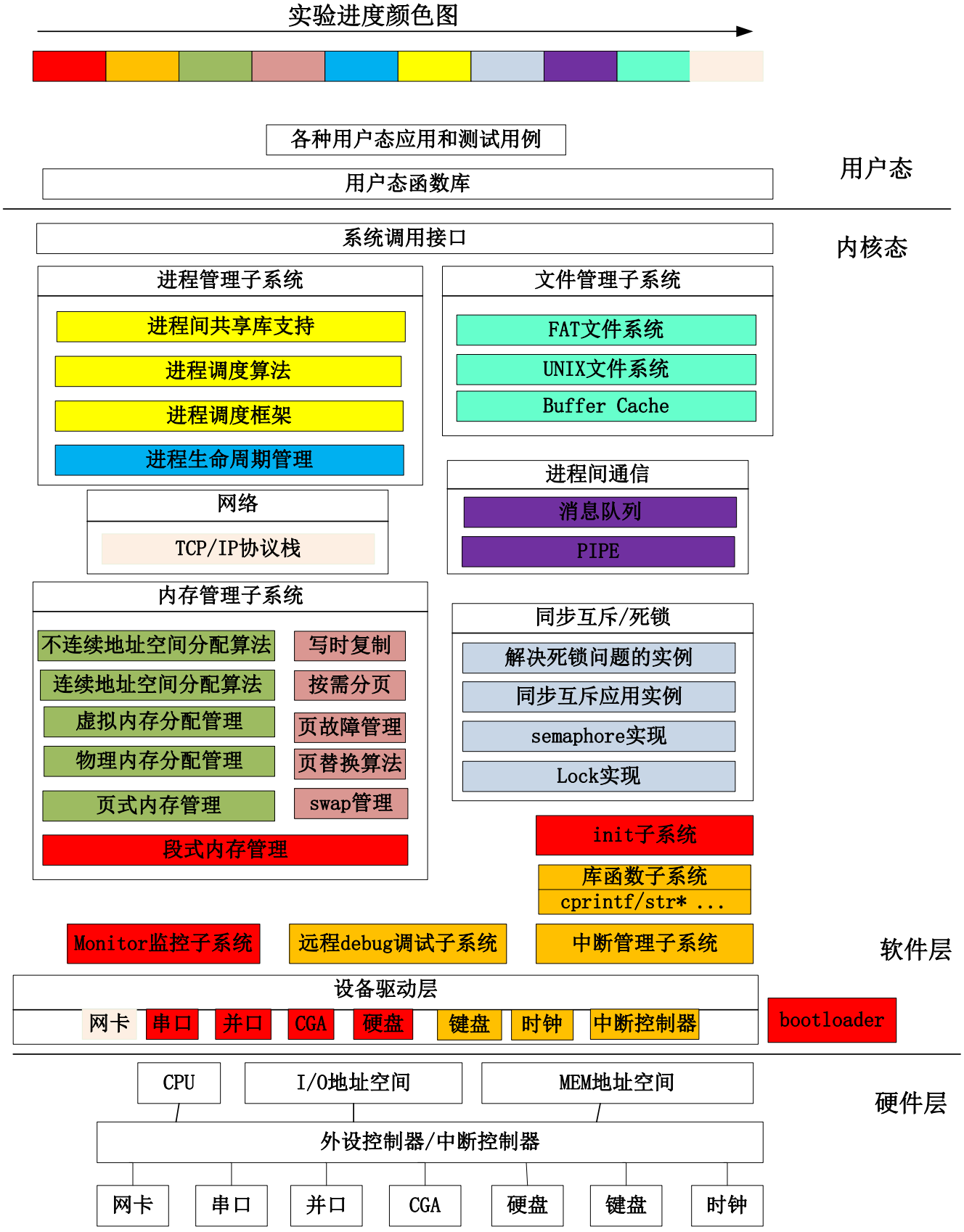


图 1 ucore 系统结构图

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 设置实验环境

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 github 网站（https://github.com/chyyuu/mooc_os_lab）可下载我们提供的 lab1~lab8 实验软件和参考答案代码，大致经过如下过程就可以完成使用。

1. 通过 git 下载软件包
2. 进入各个 OS 实验工程目录 例如： cd lab1
3. 根据实验要求指导书阅读源码并修改代码
4. 编译源码，如编译不过则返回步骤 3
5. 如编译通过则测试是否基本正确
6. 如果实现有问题，则返回步骤 3 执行 make grade
7. 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
8. 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。

其他运行调试命令：

1) make qemu: 让 OS 实验在 qemu 的图形界面上运行，且让模拟并口输出在 shell 上
2) make qemu-mon: 让 OS 实验在 qemu 上运行，且让 qemu monitor 运行在 shell 上
3) make qemu-nox: 让 OS 实验在无图形界面的 qemu 上运行，且让模拟串口输出在 shell 上
4) make debug: 让 OS 实验在 qemu 的图形界面上运行，且可以通过 gdb 调试
5) make debug: 让 OS 实验在无图形界面的 qemu 上运行，且可以通过 gdb 调试

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 下载），然后在网上下载一个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 VirtualBox 虚拟硬盘文件(mooc-os-2014.tar.xz，包含一个虚拟磁盘镜像文件和两个配置描述文件，下载此文件的网址址见 https://github.com/chyyuu/ mooc_os_lab 下的 README 中的描述)。用好压软件(有 windows 版本，可到 http://www.haozip.com 下载。一般软件解压不了 xz 格式的压缩文件）先解压到 C 盘的 vms 目录下即：

```
\mooc-os\mooc-os.vbox
\mooc-os\mooc-os.vbox-prev
\mooc-os\mooc-os.vdi
```

解压后这三个文件所占用的硬盘空间为 5GB 左右。在 VirtualBox 中加载 mooc-os.vbox，就可以启动并运行 Linux 实验环境了。如果 VirtualBox 无法执行此 vbox 文件，则需要在 VirtualBox 中增加一个新的虚拟电脑，并在配置此虚拟电脑过程中，注意选择 Linux，Ubuntu 64 bit，且选择 mooc-os.vdi 作为已有的磁盘镜像。启动将直接到桌面，用户名是 moocos，口令是“空格键”，即当系统提示输入口令时，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 ucore_lab 目录下。可以通过如下命令获得整个实验的代码：

```
$ git clone https://github.com/chyyuu/ mooc_os_lab.git
```

并可通过如下命令获得以后更新后的代码：

```
$ git pull
```

当然，你需要了解一下 git 的基本使用方法，这可以通过网络获得很多这方面的信息。

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。

命令模式的基本结构和概念

Ubuntu 是图形界面友好和易操作的 linux 发行版，但有时只需执行几条简单的指令就可以完成繁琐的鼠标点击才能完成的操作。linux 的命令行操作模式功能可以实现你需要的所有操作。简单的说，命令行就是基于字符命令的用户界面，也被称为文本操作模式。绝大多数情况下， 用户通过输入一行或多行命令直接与计算机互动，来实现对计算机的操作。

如何进入命令模式

假设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 程序，从而可以在此软件界面中进行命令行操作。
打开 gnome-terminal 程序后你首先可能会注意到类似下面的界面：

```
moocos-> ls
file1.txt    file2.txt file3.txt tools
```

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。

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。

常用指令

(1)**查询文件列表**：(ls)

```
moocos-> ls
file1.txt    file2.txt file3.txt tools
```

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)

```
moocos-> pwd
/home/moocos
```

(3)进入其他目录：(cd)

```
moocos-> pwd
/home/moocos
```

```
moocos-> cd /root/
moocos-> pwd
/root
```

上面例子中，当前目录原来是/home/moocos,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)

```
moocos-> 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

```
moocos-> 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

```
moocos-> cp file1.txt file1_copy.txt
moocos-> cat file1_copy.txt
Roses are red.
Violets are blue,
and you have the bird-flue!
```

(7)移动文件：mv

```
moocos-> ls
file1.txt
file2.txt
moocos-> mv file1.txt new_file.txt
moocos-> 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；

```
moocos-> mv -v file1.txt new_file.txt
`file1.txt' -> `new_file.txt'
```

(8)建立一个空文本文件：touch

```
moocos-> ls
file1.txt
moocos-> touch tempfile.txt
moocos-> ls
file1.txt
tempfile.txt
```

(9)建立一个目录：mkdir

```
moocos-> ls
file1.txt
tempfile.txt
moocos-> mkdir test_dir
moocos-> ls
file1.txt
tempfile.txt
```

```
test_dir
```

(10)删除文件/目录：rm

```
moocos-> ls -p
file1.txt
tempfile.txt
test_dir/
moocos-> rm -i tempfile.txt
rm: remove regular empty file `test.txt'? y
moocos-> ls -p
file1.txt
test_dir/
moocos-> rm test_dir
rm: cannot remove `test_dir': Is a directory
moocos-> rm -R test_dir
moocos-> 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，让一i 成为默认参数：

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

(11)查询当前进程：ps

```
moocos-> ps
PID TTY          TIME CMD
21071 pts/1    00:00:00 bash
22378 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，你可以访问网络得到更加详细的 Linux 命令介绍。

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：

```
moocos-> ls >file4.txt
```

```
moocos-> cat file4.txt
file1.txt    file2.txt file3.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:

```
moocos-> ls >> file4.txt
moocos-> cat file4.txt
file1.txt    file2.txt file3.txt
file1.txt    file2.txt file3.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:

```
moocos-> cat > file5.txt
a3.txt
a2.txt
file2.txt
file1.txt
 <Ctrl-D>   //这表示敲入 Ctrl+D 键
moocos-> sort < file5.txt
a2.txt
a3.txt
file1.txt
file2.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> //这表示敲入 Ctrl+Z 键
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, ...

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。

获得软件包

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 [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。

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。

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”。

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：

```
cmoocos->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。

2.2.2.1 实验中可能使用的软件

编辑器

(1) 在配置好的实验环境中，已经安装了 scitools 提供的 understand 软件（试用版），用来分析 ucore OS 源码很方便。
(2) 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，对于编辑一些小文件比较便捷。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。
(3) Vim 编辑器：Vim 是一款极方便的文本编辑软件，适合老手。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。

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。

提示：习惯 GUI 方式的同学，可采用图形界面的 understand、source insight 等软件。

diff & patch

在比较不同目录中文件的差异性时，推荐使用有图形界面的 meld，在系统中没有提前安装，可通过如下命令来安装：

```
sudo apt-get install meld
```

如果有提问，选 “Y”即可。

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。

提示：习惯 GUI 方式的同学，可采用图形界面的 meld、kdiff3、UltraCompare 等软件。

2.3 了解编程开发调试的基本工具

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。

2.3.1 gcc 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
```

2.3.1.1 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’。

2.3.1.2 AT&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]

2.3.1.3 GCC 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。

2.3.1.4 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
r	使用任何可用的通用寄存器
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，冒号“：”要保留，当“memory”存在时，三个冒号都要保留，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。
2. 使用 r 指示编译器从 eax, ebx, ecx, edx, esi, edi 分配寄存器。
3. 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。
4. "="是标示输出寄存器，必须这样用。
5. 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。

6. 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。

2.3.2 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。

2.3.2.1.1makefile 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。

2.3.3 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。

用 gdb 查看源代码可以用 list 命令，但是这个不够灵活。可以使用"layout src"命令，或者按 Ctrl-X 再按 A，就会出现一个窗口可以查看源代码。也可以用使用-tui 参数，这样进入 gdb 里面后就能直接打开代码查看窗口。其他代码窗口相关命令：

```
info win       显示窗口的大小
layout next  切换到下一个布局模式
layout prev  切换到上一个布局模式
layout src     只显示源代码
layout asm     只显示汇编代码
layout split  显示源代码和汇编代码
layout regs     增加寄存器内容显示
focus cmd/src/asm/regs/next/prev  切换当前窗口
refresh         刷新所有窗口
tui reg next  显示下一组寄存器
tui reg system  显示系统寄存器
update          更新源代码窗口和当前执行点
winheight name +/- line  调整 name 窗口的高度
tabset nchar  设置 tab 为 nchar 个字符
```

2.3.4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

…

2.4基于硬件模拟器实现源码级调试

2.4.1安装硬件模拟器 QEMU

2.4.1.1Linux 运行环境

QEMU 用于模拟一台 x86 计算机，让 ucore 能够运行在 QEMU 上。可直接使用 ubuntu 中提供的 qemu，只需执行如下命令即可。

sudo apt-get install qemu-system-x86

在本实验中，使用的是 qemu-system-i386 这个软件来模拟一个 Intel 80386 计算机。

2.4.2使用硬件模拟器 QEMU

2.4.2.1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          //让 ucore 在 qemu 模拟的 x86 硬件环境中执行
```

或

```
qemu -S -s -hda ucore.img -monitor stdio    //用于与 gdb 配合进行源码调试
```

2.4.2.2 常用调试命令

(1)qemu 中 monitor 的常用命令：

help	查看 qemu 帮助，显示所有支持的命令。
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。
info 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。
stop	暂停虚拟机
cont	继续执行虚拟机
quit	退出 qemu

其他具体的命令格式以及说明，参见 qemu help 命令帮助。

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
```

2.4.3 基于 qemu 内建模式调试 ucore

调试举例：调试 lab1，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: ~/lab1$ 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
```

更简单办法是在 labX 中，运行 make debug

2.4.4 结合 gdb 和 qemu 源码级调试 ucore

2.4.4.1编译可调试的目标文件

为了使得编译出来的代码是能够被 gdb 这样的调试器调试，我们需要在使用 gcc 编译源文件的时候添加参数：-g –gdb 。这样编译出来的目标文件中才会包含可以用于调试器进行调试的相关符号信息。

2.4.4.2ucore 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：

```
chy@laptop: ~/lab1$   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。

2.4.4.3 使用远程调试

为了与 qemu 配合进行源代码级别的调试，需要先让 qemu 进入等待 gdb 调试器的接入并且还不能让 qemu 中的 CPU 执行，因此启动 qemu 的时候，我们需要使用参数-S –s 这两个参数来做到这一点。在使用了前面提到的参数启动 qemu 之后，qemu 中的 CPU 并不会马上开始执行，这时我们启动 gdb，然后在 gdb 命令行界面下，使用下面的命令连接到 qemu：

(gdb)　target remote 127.0.0.1:1234

然后输入 c（也就是 continue）命令之后，qemu 会继续执行下去，但是 gdb 由于不知道任何符号信息，并且也没有下断点，是不能进行源码级的调试的。为了让 gdb 获知符号信息，需要指定调试目标文件，gdb 中使用 file 命令：

(gdb)　file obj/kernel/kernel.elf

之后 gdb 就会载入这个文件中的符号信息了。

通过 gdb 可以对 ucore 代码进行调试，以 lab1 中调试 memset 函数为例：

(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: ~/lab1$ qemu -S -hda ./bin/ucore.img    -s
```

窗口二

```
chy@laptop: ~/lab1$ gdb ./bin/kernel
(gdb) target remote:1234
Remote debugging using :1234
0x0000fff0 in ?? ()
(gdb) file obj/kernel/kernel.elf
(gdb) break memset
Breakpoint 1 at 0x100d9f: file libs/string.c, line 54.
(gdb) run
Starting program:
/home/chenyu/oscourse/develop/ucore/lab1/bin/kernel

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

更简单办法是在 labX 中，运行 make debug

2.4.4.4 使用 gdb 配置文件

在上面可以看到，为了进行源码级调试，需要输入较多的东西，很麻烦。为了方便，可以将这些命令存在脚本中，并让 gdb 在启动的时候自动载入。

我们可以创建文件 gdbinit，并输入下面的内容：

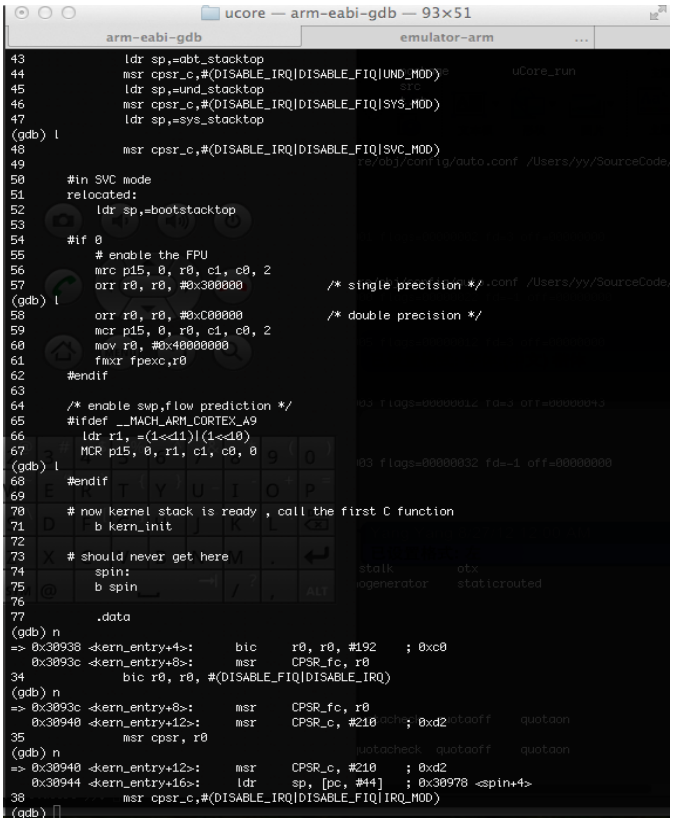
```
target remote 127.0.0.1:1234
file obj/kernel/kernel.elf
```

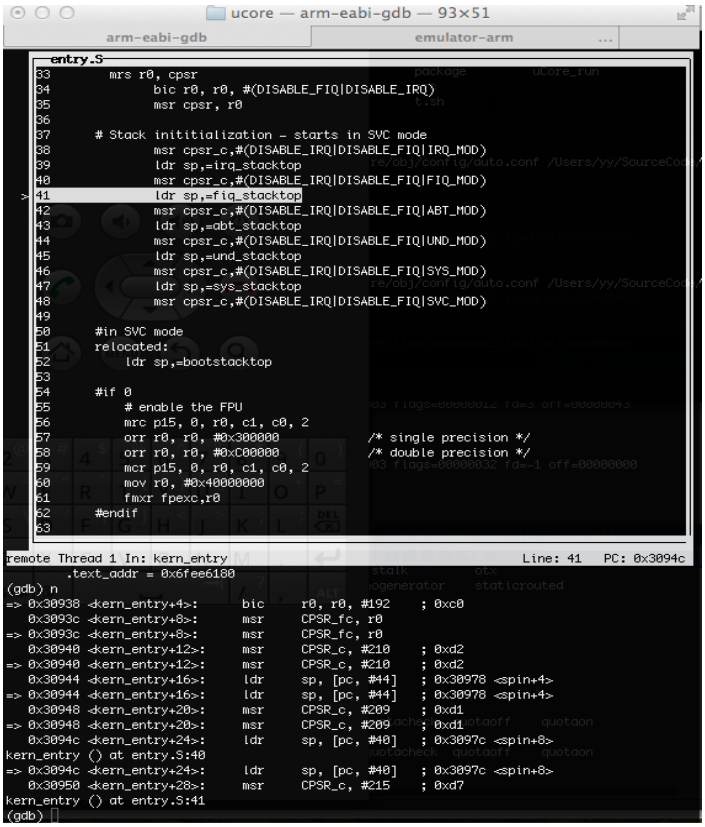
为了让 gdb 在启动时执行这些命令，使用下面的命令启动 gdb：

```
$ gdb –x gdbinit
```

如果觉得这个命令太长，可以将这个命令存入一个文件中，当作脚本来执行。

另外，如果直接使用上面的命令，那么得到的界面是一个纯命令行的界面，不够直观，就像下图这样：





如果想获得上面右图那样的效果，只需要再加上参数-tui 就行了，比如：

```
gdb -tui –x gdbinit
```

2.4.4.5 设定调试目标架构

在调试的时候，我们也许需要调试不是 i386 保护模式的代码，比如 8086 实模式的代码，我们需要设定当前使用的架构：

```
(gdb) set arch i8086
```

这个方法在调试不同架构或者说不同模式的代码时还是有点用处的。

2.5 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。

2.5.1Intel 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，支持内存分段机制和分页机制（简称段机制和页机制），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。

2.5.2Intel 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。地址是访问内存空间的索引。**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**是在操作系统的虚存管理之下每个运行的应用程序能访问的地址空间。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

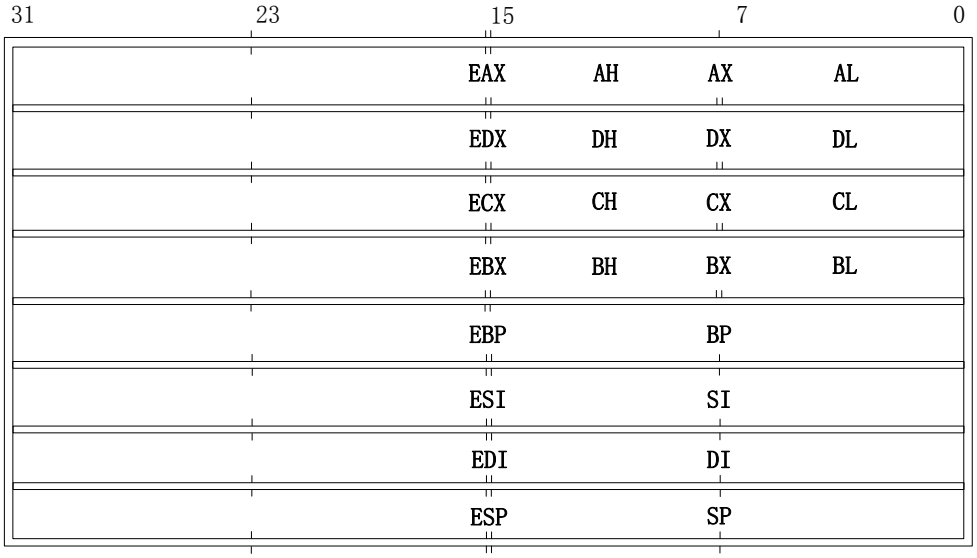
2.5.3Intel 80386 寄存器

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

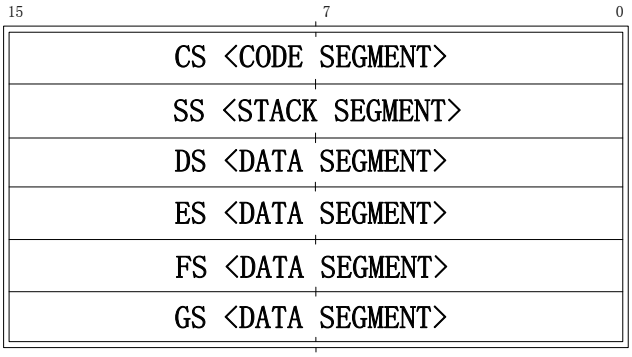
通用寄存器



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：

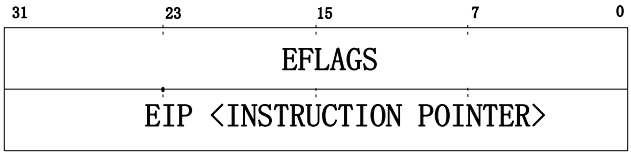
CS：代码段(Code Segment)
DS：数据段(Data Segment)
ES：附加数据段(Extra Segment)
SS：堆栈段(Stack Segment)
FS：附加段
GS：附加段

段寄存器



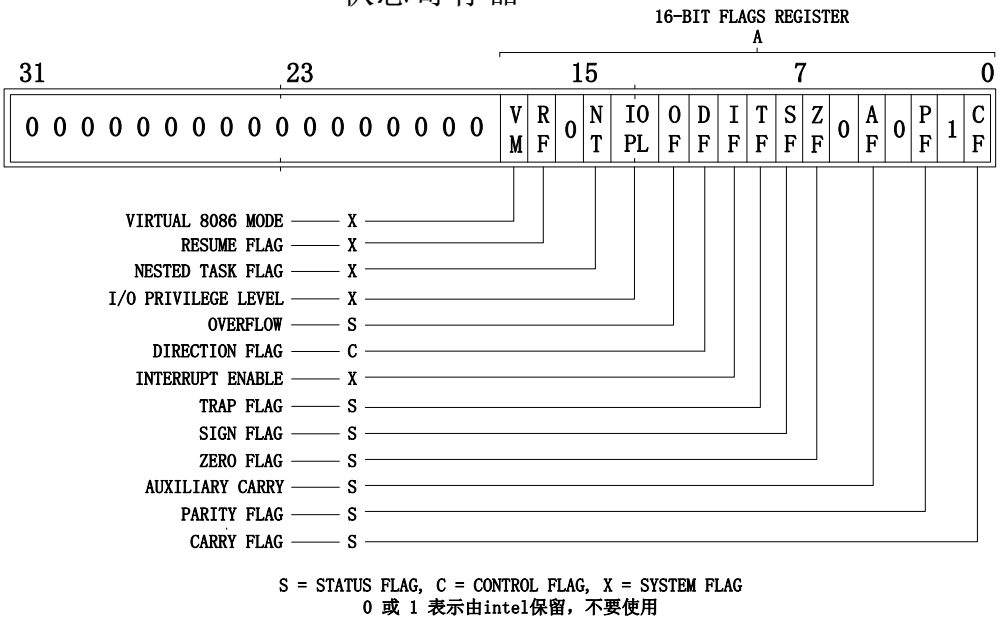
Instruction Pointer(指令指针寄存器)：EIP 的低 16 位就是 8086 的 IP，它存储的是下一条要执行指令的内存地址，在分段地址转换中，表示指令的段内偏移地址。

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。

2.6 了解 ucore 编程方法和通用数据结构

2.6.1面向对象编程方法

uCore 设计中采用了一定的面向对象编程方法。虽然 C 语言对面向对象编程并没有原生支持，

但没有原生支持并不等于我们不能用 C 语言写面向对象程序。需要注意，我们并不需要用 C 语言模拟出一个常见 C++ 编译器已经实现的对象模型。如果是这样，还不如直接采用 C++编程。

uCore 的面向对象编程方法，目前主要是采用了类似 C++的接口（interface）概念，即是让实现细节不同的某类内核子系统（比如物理内存分配器、调度器，文件系统等）有共同的操作方式，这样虽然内存子系统的实现千差万别，但它的访问接口是不变的。这样不同的内核子系统之间就可以灵活组合在一起，实现风格各异，功能不同的操作系统。接口在 C 语言中，表现为一组函数指针的集合。放在 C++ 中，即为虚表。接口设计的难点是如果找出各种内核子系统的共性访问/操作模式，从而可以根据访问模式提取出函数指针列表。

比如对于 uCore 内核中的物理内存管理子系统，首先通过分析内核中其他子系统可能对物理内存管理子系统，明确物理内存管理子系统的访问/操作模式，然后我们定义了 pmm_manager 数据结构（位于 lab2/kern/mm/pmm.h）如下：

```
// pmm_manager is a physical memory management class. A special pmm manager - XX
X_pmm_manager
// only needs to implement the methods in pmm_manager class, then XXX_pmm_manage
r can be used
// by ucore to manage the total physical memory space.
struct pmm_manager {
  // XXX_pmm_manager's name
  const char *name;
  // initialize internal description&management data structure
  // (free block list, number of free block) of XXX_pmm_manager
  void (*init)(void);
  // setup description&management data structcure according to
  // the initial free physical memory space
  void (*init_memmap)(struct Page *base, size_t n);
  // allocate >=n pages, depend on the allocation algorithm
  struct Page *(*alloc_pages)(size_t n);
  // free >=n pages with "base" addr of Page descriptor structures(memlayout.h)
  void (*free_pages)(struct Page *base, size_t n);
  // return the number of free pages
  size_t (*nr_free_pages)(void);
   // check the correctness of XXX_pmm_manager
  void (*check)(void);
};
```

这样基于此数据结构，我们可以实现不同连续内存分配算法的物理内存管理子系统，而这些物理内存管理子系统需要编写算法，把算法实现在此结构中定义的 init（初始化）、init_memmap（分析空闲物理内存并初始化管理）、alloc_pages（分配物理页）、free_pages（释放物理页）函数指针所对应的函数中。而其他内存子系统需要与物理内存管理子系统交互时，只需调用特定物理内存管理子系统所采用的 pmm_manager 数据结构变量中的函数指针即可

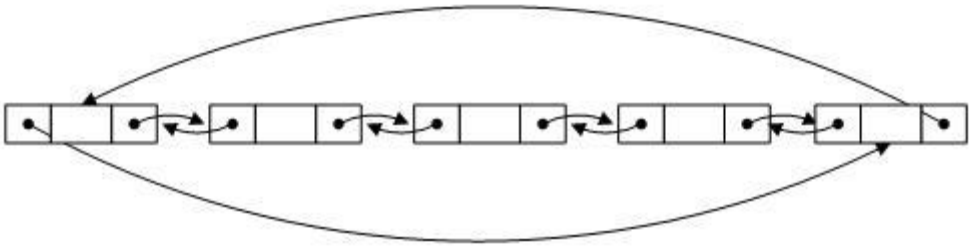
2.6.2通用数据结构

2.6.2.1双向循环链表

在“数据结构”课程中，如果创建某种数据结构的双循环链表，通常采用的办法是在这个数据结构的类型定义中有专门的成员变量 data，并且加入两个指向该类型的指针 next 和 prev。例如：

```
typedef struct foo {
      ElemType data;
      struct foo *prev;
      struct foo *next;
} foo_t;
```

双向循环链表的特点是尾节点的后继指向首节点，且从任意一个节点出发，沿两个方向的任何一个，都能找到链表中的任意一个节点的 data 数据。由双向循环列表形成的数据链如下所示：



双向循环链表

这种双向循环链表数据结构的一个潜在问题是，虽然链表的基本操作是一致的，但由于每种特定数据结构的类型不一致，需要为每种特定数据结构类型定义针对这个数据结构的特定链表插入、删除等各种操作，会导致代码冗余。

在 uCore 内核（从 lab2 开始）中使用了大量的双向循环链表结构来组织数据，包括空闲内存块列表、内存页链表、进程列表、设备链表、文件系统列表等的数据组织（在[labX/libs/list.h]实现），但其具体实现借鉴了 Linux 内核的双向循环链表实现，与“数据结构”课中的链表数据结构不太一样。下面将介绍这一数据结构的设计与操作函数。

uCore 的双向链表结构定义为：

```
struct list_entry {
    struct list_entry *prev, *next;
};
```

需要注意 uCore 内核的链表节点 list_entry 没有包含传统的 data 数据域，，而是在具体的数据结构中包含链表节点。以 lab2 中的空闲内存块列表为例，空闲块链表的头指针定义（位于 lab2/kern/mm/memlayout.h 中）为：

```
/* free_area_t - maintains a doubly linked list to record free (unused) pages */
typedef struct {
    list_entry_t free_list;         // the list header
    unsigned int nr_free;           // # of free pages in this free list
} free_area_t;
```

而每一个空闲块链表节点定义（位于 lab2/kern/mm/memlayout）为：

```
/* *
 * struct Page - Page descriptor structures. Each Page describes one
 * physical page. In kern/mm/pmm.h, you can find lots of useful functions
 * that convert Page to other data types, such as phyical address.
 * */
struct Page {
    atomic_t ref;          // page frame's reference counter
    ……
    list_entry_t page_link;         // free list link
};
```

这样以 free_area_t 结构的数据为双向循环链表的链表头指针，以 Page 结构的数据为双向循环链表的链表节点，就可以形成一个完整的双向循环链表，如下图所示：

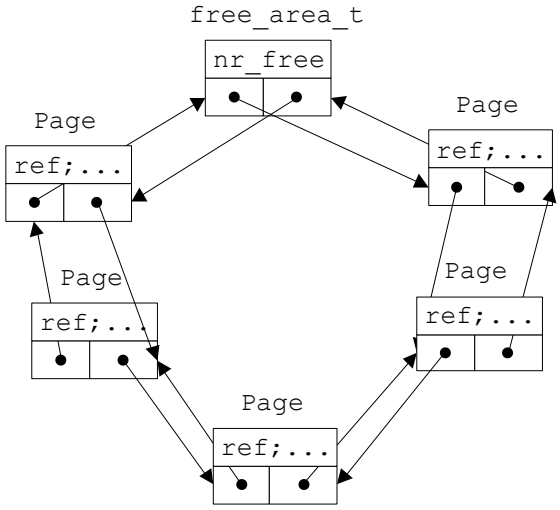


图 空闲块双向循环链表

从上图中我们可以看到，这种通用的双向循环链表结构避免了为每个特定数据结构类型定义针对这个数据结构的特定链表的麻烦，而可以让所有的特定数据结构共享通用的链表操作函数。在实现对空闲块链表的管理过程（参见 lab2/kern/mm/default_pmm.c）中，就大量使用了通用的链表插入，链表删除等操作函数。有关这些链表操作函数的定义如下。

1. 初始化

uCore 只定义了链表节点，并没有专门定义链表头，那么一个双向循环链表是如何建立起来的呢？让我们来看看 list_init 这个内联函数（inline funciton）：

```
static inline void
list_init(list_entry_t *elm) {
    elm->prev = elm->next = elm;
}
```

参看文件 default_pmm.c 的函数 default_init，当我们调用 list_init(&(free_area.free_list))时，就声明一个名为 free_area.free_list 的链表头时，它的 next、prev 指针都初始化为指向自己，这样，我们就有了一个表示空闲内存块链的空链表。而且我们可以用头指针的 next 是否指向自己来判断此链表是否为空，而这就是内联函数 list_empty 的实现。

2. 插入

对链表的插入有两种操作，即在表头插入（list_add_after）或在表尾插入（list_add_before）。因为双向循环链表的链表头的 next、prev 分别指向链表中的第一个和最后一个节点，所以，list_add_after 和 list_add_before 的实现区别并不大，实际上 uCore 分别用__list_add(elm, listelm, listelm->next)和__list_add(elm, listelm->prev, listelm)来实现在表头插入和在表尾插入。而__list_add 的实现如下：

```
static inline void
__list_add(list_entry_t *elm, list_entry_t *prev, list_entry_t *next) {
    prev->next = next->prev = elm;
    elm->next = next;
    elm->prev = prev;
}
```

从上述实现可以看出在表头插入是插入在 listelm 之后，即插在链表的最前位置。而在表尾插入是插入在 listelm->prev 之后，即插在链表的最后位置。注：list_add 等于 list_add_after。

3. 删除

当需要删除空闲块链表中的 Page 结构的链表节点时，可调用内联函数 list_del，而 list_del 进一步调用了__list_del 来完成具体的删除操作。其实现为：

```
static inline void
list_del(list_entry_t *listelm) {
```

```
    __list_del(listelm->prev, listelm->next);
}

static inline void
__list_del(list_entry_t *prev, list_entry_t *next) {
    prev->next = next;
    next->prev = prev;
}
```

如果要确保被删除的节点 listelm 不再指向链表中的其他节点，这可以通过调用 list_init 函数来把 listelm 的 prev、next 指针分别自身，即将节点置为空链状态。这可以通过 list_del_init 函数来完成。

3. 访问链表节点所在的宿主数据结构

通过上面的描述可知，list_entry_t 通用双向循环链表中仅保存了某特定数据结构中链表节点成员变量的地址，那么如何通过这个链表节点成员变量访问到它的所有者（即某特定数据结构的变量）呢？Linux 为此提供了针对数据结构 XXX 的 le2XXX(le, member)的宏，其中 le，即 list entry 的简称，是指向数据结构 XXX 中 list_entry_t 成员变量的指针，也就是存储在双向循环链表中的节点地址值，member 则是 XXX 数据类型中包含的链表节点的成员变量。例如，我们要遍历访问空闲块链表中所有节点所在的基于 Page 数据结构的变量，则可以采用如下编程方式（基于 lab2/kern/mm/default_pmm.c）：

```
//free_area 是空闲块管理结构，free_area.free_list 是空闲块链表头
free_area_t free_area;
list_entry_t * le = &free_area.free_list;  //le 是空闲块链表头指针
while((le=list_next(le)) != &free_area.free_list) { //从第一个节点开始遍历
      struct Page *p = le2page(le, page_link); //获取节点所在基于 Page 数据结构的变量
      ……
}
```

le2page 宏（定义位于 lab2/kern/mm/memlayout.h）的使用相当简单：

```
// convert list entry to page
#define le2page(le, member)                 \
    to_struct((le), struct Page, member)
```

而相比之下，它的实现用到的 to_struct 宏和 offsetof 宏（定义位于 lab2/libs/defs.h）则有一些难懂：

```
/* Return the offset of 'member' relative to the beginning of a struct type */
#define offsetof(type, member)                                      \
    ((size_t)(&((type *)0)->member))

/* *
 * to_struct - get the struct from a ptr
 * @ptr:    a struct pointer of member
 * @type:   the type of the struct this is embedded in
 * @member: the name of the member within the struct
 * */
#define to_struct(ptr, type, member)                               \
    ((type *)((char *)(ptr) - offsetof(type, member)))
```

这里采用了一个利用 gcc 编译器技术的技巧，即先求得数据结构的成员变量在本宿主数据结构中的偏移量，然后根据成员变量的地址反过来得出属主数据结构的变量的地址。

我们首先来看 offsetof 宏，size_t 最终定义与 CPU 体系结构相关，本实验都采用 Intel X86-32 CPU，顾 szie_t 等价于 unsigned int。 ((type *)0)->member 的设计含义是什么？其实这是为了求得数据结构的成员变量在本宿主数据结构中的偏移量。为了达到这个目标，首先将 0 地址强制"转换"为 type 数据结构（比如 struct Page）的指针，再访问到 type 数据结构中的 member 成员（比如 page_link）的地址，即是 type 数据结构中 member 成员相对于数据结构变量的偏移量。在 offsetof 宏中，这个 member

成员的地址（即“`&((type *)0)->member)`”）实际上就是 type 数据结构中 member 成员相对于数据结构变量的偏移量。对于给定一个结构，offsetof(type,member)是一个常量，to_struct 宏正是利用这个不变的偏移量来求得链表数据项的变量地址。接下来再分析一下 to_struct 宏，可以发现 to_struct 宏中用到的 ptr 变量是链表节点的地址，把它减去 offsetof 宏所获得的数据结构内偏移量，即就得到了包含链表节点的属主数据结构的变量的地址。