

第1章 xv6 安装使用

xv6 是一个教学操作系统,它是对 Dennis Ritchies 和 Ken Thompson 的 UNIX Version 6 (v6) 的一个实现,但并不严格遵循 v6 的结构和风格。xv6 用 ANSI C 实现并运行在 x86 多核系统上,麻省理工大学的网站 (<http://pdos.csail.mit.edu/6.828/2011/xv6.html>) 上有 xv6 来龙去脉的详细介绍。由于不同版本略有不同,本书使用的是 rev 9。

在我们后面的操作中,将使用 x86 仿真器来观察 xv6 的运行过程,因此我们先安装 QEMU (当然也可以用 bochs),然后再安装 xv6 代码。需要注意的是 xv6 所生成的代码是 32 位代码。

1.1. 运行于 QEMU 的 xv6

我们先以 CentOS7 64 位 Linux 为例说明 xv6 的安装过程。后面再介绍 Ubuntu18 64 位 Linux 上安装 xv6 的过程。两种操作系统环境中,都是以 QEMU 仿真系统来运行 xv6 的。QEMU 是一套由法布里斯贝拉 (Fabrice Bellard) 所编写的处理器仿真软件,是在 GNU/Linux 平台上使用广泛的 GPL 开源软件。xv6 操作系统可以运行于该仿真系统上。

1.1.1. CentOS7+QEMU+xv6

如果读者想在 CentOS7 环境中安装 QEMU 仿真器来运行 xv6 操作系统,那么可以按照以下的步骤来操作。我们假设 CentOS7 系统中已经有 GCC 等基本开发环境。

安装 QEMU

比较简单的方式是使用 `yum install qemu` 命令完成 QEMU 的安装,将自动检测软件依赖关系并下载所需其他软件后进行安装。由于 CentOS7 默认的 yum 源出于稳定性考虑,其中的软件版本都比较滞后,很可能找不到 QEMU。此时可以先执行 `yum -y install epel-release` 命令,安装 Fedora 社区打造的 EPEL (Extra Packages for Enterprise Linux) 扩展源。

安装 xv6

xv6 源码可以从多个地方获得,例如从 <https://github.com/mit-pdos/xv6-public> 网站下载 xv6 的源码。也可以在 shell 中执行 `git clone git://pdos.csail.mit.edu/xv6/xv6.git` 或 `git clone https://github.com/mit-pdos/xv6-public.git` 命令下载 xv6 源码。还可以用 `wget` 从 yale 大学下载,具体命令为 `wget http://zoo.cs.yale.edu/classes/cs422/2011/xv6-rev4.tar.gz`。

我们是用浏览器在 github 网站 <https://github.com/mit-pdos/xv6-public/releases> 下载 rev9 版本的 xv6 源码包 `xv6-public-xv6-rev9.tar.gz`,然后用 `tar -zxvf xv6-public-xv6-rev9.tar.gz` 解压缩,最后进入源代码目录中执行 `make` 即可进行编译,如果没有提示错误则已完成安装。

如果有编译错误,请查看具体问题并解决。

在安装目录下执行 `make qemu` 则可以启动 QEMU 仿真环境并运行 xv6 操作系统, 如图 1-1 所示。当单击 QEMU 仿真窗口后被该窗口捕获, 呈现出系统无法响应鼠标操作的现象。如果需要从仿真窗口脱离, 需要用“`Alt+Ctrl`”组合键把鼠标从仿真窗口中解脱出来。如果使用 `make qemu-nox` 启动的, 则可以在仿真窗口中先按下“`Ctrl+a`”松开后再按“`x`”则结束仿真。

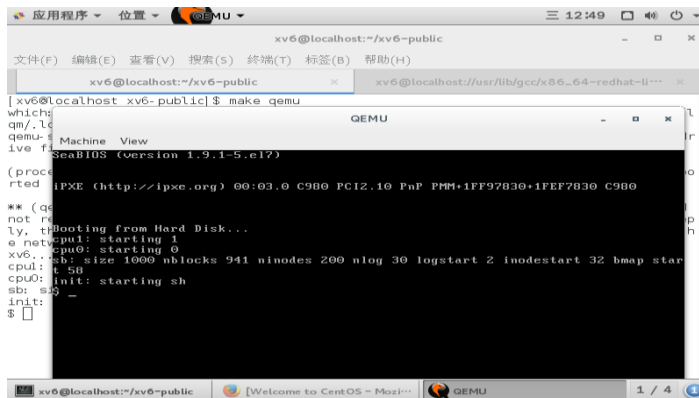


图 1-1 CentOS7+QEMU+xv6 运行示意图

批注 [E1]: 图不清楚。

1.1.2. Ubuntu18+QEMU+xv6

本节我们使用 Ubuntu 18.04.01 的 ISO 镜像 (ubuntu-18.04.1-desktop-amd64) 来安装开发环境的操作系统, 安装期间选择不更新其他软件包也不安装第三方软件。Ubuntu 安装结束后, 输入用户名和密码后即可完成登陆。如果要修改 root 账户的密码, 此时可以执行 `sudo passwd root` 然后输入新密码就可以了。如果需要使用终端, 则在桌面空白处按下 `Ctrl+ALT+T` 可以弹出终端。使用 `Ctrl+ALT+F1~F6` 可以在多个终端之间切换。

下面展示如何在 Ubuntu 上安装 xv6 及相关软件。

QEMU 安装

前面提到过, QEMU 是一套由法布里斯贝拉 (Fabrice Bellard) 所编写的处理器仿真软件, 是在 GNU/Linux 平台上使用广泛的 GPL 开源软件。以管理员身份, 在终端窗口直接执行 `apt-get install qemu` 命令即可完成安装。

xv6 安装

在 github 网站 <https://github.com/mit-pdos/xv6-public/releases> 下载 rev9 版本的 xv6 源码包 `xv6-public-xv6-rev9.tar.gz`, 然后用 `tar -zxvf xv6-public-xv6-rev9.tar.gz` 解压缩, 最后进入源代码目录中执行 `make` 即可完成编译。如果系统还没有安装 `make` 则需要用 `apt install make`。类似地, 如果 `make` 是提示还未安装 `gcc` 则需要执行 `apt install gcc` 完成相应的安装。

此时, 在 `xv6` 目录下执行 `make qemu` 即可启动仿真运行, 如图 1-2 所示。当单击 QEMU 仿真窗口时被该窗口捕获, 呈现出系统无法响应外部鼠标操作的现象。如果需要从仿真窗口

脱离，需要用“Alt+Ctrl”组合键把鼠标从仿真窗口中解脱出来。如果使用 `make qemu-nox` 启动的，则可以在仿真窗口中先按下“Ctrl+a”松开后再按“x”则结束仿真。

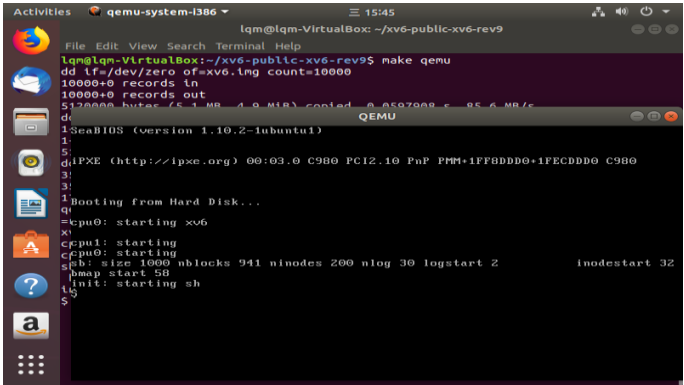


图 1-2 Ubuntu18+QEMU+xv6 运行示意图

批注 [E2]: 正文中没有引用。
已引用

1.2. 调试观察

在学习 `xv6` 代码的过程中，我们需要不断地用 `gdb` 调试器观察代码的行为，因此读者需要有一点 `gdb` 常用命令的知识。如果读者已经学习过本系列的《Linux GNU C 程序观察》一书，已具备足够的背景知识。否则，可能需要自行了解 `gdb` 调试器基本命令的使用。

1.2.1. xv6 shell 命令

在 `xv6` 源码目录下执行 `make qemu` 将启动 QEMU 仿真器并运行 `xv6` 操作系统，此时还将弹出一个独立的窗口用于显示 `xv6` 的输出，如图 1-3 所示。如果执行 `make qemu-nox` 将不会另外弹出窗口，而是在原来的 `shell` 文本窗口显示 `xv6` 的输出。

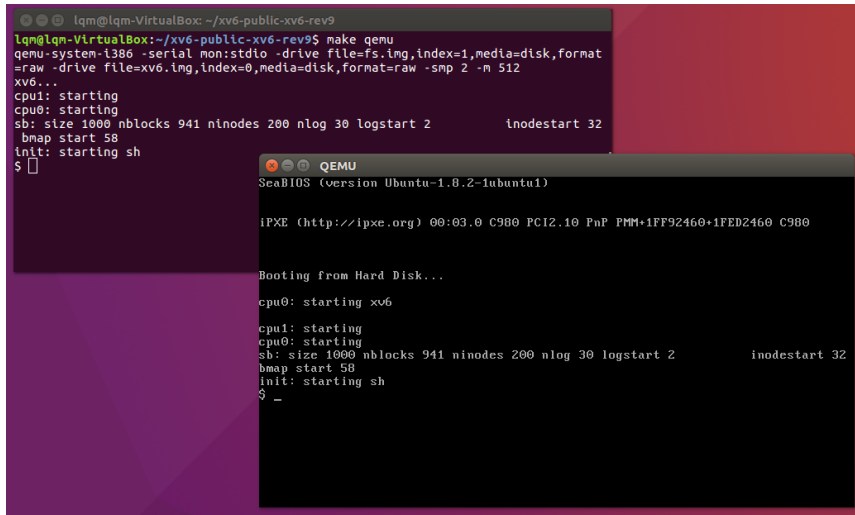


图 1-3 make qemu 启动 QEMU 仿真器运行 xv6

再次提醒，如果在 QEMU 仿真窗口单击鼠标，那么鼠标输入会被捕获而且无法移出仿真窗口，容易误以为系统无法操作。如果需要解除鼠标捕获，需要按下 **Ctrl+Alt** 组合键即可。

此时在 xv6 shell（或 QEMU 仿真窗口）中执行 **ls** 命令，可以看到 xv6 文件系统中所有的可执行文件，如屏显 1-1 所示。

屏显 1-1 ls 列出所有磁盘文件

```
$ ls
.          1 1 512
..         1 1 512
README    2 2 2191
cat        2 3 13236
echo       2 4 12428
forktest   2 5 8136
grep       2 6 15176
init       2 7 13016
kill       2 8 12468
ln         2 9 12376
ls         2 10 14592
mkdir      2 11 12492
rm         2 12 12468
sh         2 13 23108
stressfs   2 14 13148
usertests  2 15 55568
wc         2 16 14004
zombie     2 17 12200
console    3 20 0
$
```

读者可以尝试用 **cat README** 查看 README 内容，或者 **echo** 显示某一行消息等。甚至还可以尝试其他 Linux 常用功能，例如用管道将两个命令连接执行 **cat README | grep xv6**。xv6 对目录操作的几个命令和 Linux 的命令差不多：**mkdir**、**rm** 和内部命令 **cd** 等。

如果在 shell 中按下 Ctrl+P 则会显示当前运行的进程信息，如屏显 1-2 所示。此时有两个进程，进程号为 1 的是 init 进程，当前处于 sleep 阻塞状态；编号为 2 的进程是 sh 进程，当前也是处于 sleep 阻塞状态。后面我们会看到，上面的信息实际上是通过内核函数 procdump() 打印出来的。

屏显 1-2 用 Ctrl+P 查看运行进程信息

```
$
$ 1 sleep init 80103e4f 80103ee9 80104789 801057a9 8010559b
2 sleep sh 80103e13 801002a2 80100f5c 80104a82 80104789 801057a9 8010559b
```

每个进程后面的数字，是调用栈中关于函数调用的返回地址。如果读者对它们感兴趣，可以用 addr2line 工具找出它们各自属于什么文件的哪一行。例如我们执行 addr2line -e kernel 80103e13，可以知道地址 0x80103e13 对应于 kernel 内核代码 proc.c 文件的第 388 行，如屏显 1-3 所示。查看 proc.c 的 388 可以知道它位于 sleep() 函数内。

屏显 1-3 用 addr2line 查看地址 80103e13 在 kernel 源代码中的位置

```
lqm@lqm-VirtualBox:~/xv6-public-xv6-rev9$ addr2line -e kernel 80103e13
/home/lqm/xv6-public-xv6-rev9/proc.c:388
lqm@lqm-VirtualBox:~/xv6-public-xv6-rev9$
```

将调用栈的所有地址逐个检查一遍，就可以还原该进程阻塞前的函数调用嵌套情况。例如上面的 sh 进程是通过系统调用进入到内核的，具体过程包括 alltraps -> trap -> syscall -> sys_read() -> readi() -> consleread() -> sleep()，如图 1-4 所示。

```
[trapasm.S:24] alltraps
->[trap.c:44] trap()
->[syscall.c:133] syscall()
->[sysfile.c:75] sys_read()
->[file.c:106] readi()
->[console.c:243] consleread()
->[proc.c:388] sleep()
```

图 1-4 sh 进程进入阻塞的过程示意图

1.2.2. QEMU+gdb 调试

用 GDB 调试 QEMU 时可以将调试目标分为两种，一种是用 GDB 调试由 QEMU 启动的虚拟机，即远程调试虚拟机系统内核，可以从虚拟机的 bootloader 开始调试虚拟机启动过程，另一种是调试 QEMU 本身的代码而不是虚拟机要运行的代码。我们这里需要调试的是 xv6 代码，而不是 QEMU 仿真器的代码。

当用 gdb 调试时涉及三个窗口，如图 1-5 所示。一个是启动 QEMU 的 shell 窗口（左上角），一个 QEMU 虚拟机的输出窗口（左下角），一个是运行 gdb 的 shell 窗口（右上角）。下面将详细讨论这三个窗口中运行的命令。

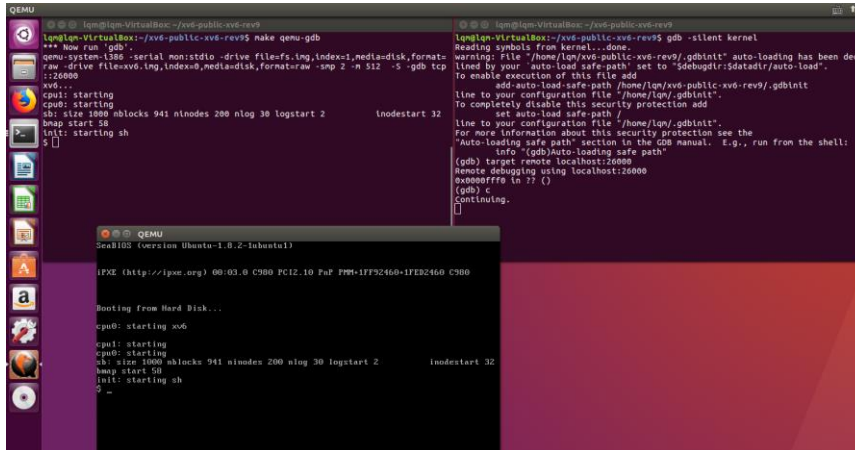


图 1-5 用 gdb 调试运行于 QEMU 环境的 xv6

首先，我们在一个终端 shell 中进入 xv6 源码目录并执行 `make qemu-gdb` 启动调试模式，实际上执行了调试服务器 `gdbserver` 的角色，等待 `gdb` 客户端的接入，如屏显 1-4 所示。此时将会弹出一个 QEMU 的窗口，但是窗口标题显示“stopped”状态，表示被调试（Traced）而未运行。

屏显 1-4 启动 QEMU 调试模式

```
lqm@lqm-VirtualBox:~/xv6-public-xv6-rev9$ make qemu-gdb
dd if=/dev/zero of=xv6.img count=10000
记录了 10000+0 的读入
记录了 10000+0 的写出
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0402051 s, 127 MB/s
dd if=bootblock of=xv6.img conv=notrunc
记录了 1+0 的读入
记录了 1+0 的写出
512 bytes copied, 0.0271599 s, 18.9 kB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
记录了 333+1 的读入
记录了 333+1 的写出
170532 bytes (171 kB, 167 KiB) copied, 0.0142105 s, 12.0 MB/s
sed 's/localhost:1234/localhost:26000/' < .gdbinit.tpl > .gdbinit
*** Now run `gdb`.
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive
file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512 -S -gdb tcp::26000
main-loop: WARNING: I/O thread spun for 1000 iterations
```

从屏显 1-4 可以看出调试服务器 `gdbserver` 在 TCP 端口 26000 上监听，因此我们在另外一个终端上启动 `gdb` 并连接到该主机的 26000 上。执行 `gdb -silent kernel` 启动 `gdb` 对 xv6 内核 `kernel` 的调试，然后在 `gdb` 命令提示符下执行 `target remote localhost:26000` 连接到 xv6 目标系统上，如屏显 1-5 所示。注意其中的被调试对象（`kernel` 文件）要在当前目录，或者用路径指出其位置。

屏显 1-5 gdb 客户端接入

```
lqm@lqm-VirtualBox:~/xv6-public-xv6-rev9$ gdb -silent kernel
```

```

Reading symbols from kernel...done.
warning: File "/home/lqm/xv6-public-xv6-rev9/.gdbinit" auto-loading has been declined by your `auto-load
safe-path' set to "$debugdir:$datadir/auto-load".
To enable execution of this file add
    add-auto-load-safe-path /home/lqm/xv6-public-xv6-rev9/.gdbinit
line to your configuration file "/home/lqm/.gdbinit".
To completely disable this security protection add
    set auto-load safe-path /
line to your configuration file "/home/lqm/.gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
    info "(gdb)Auto-loading safe path"
(gdb) target remote localhost:26000
Remote debugging using localhost:26000
0x0000ffff in ?? ()
(gdb)

```

虽然我们将 `gdb` 连接到被调试的 `xv6` 上，但是屏显 1-5 提示了 `.gdbinit` 脚本没有执行。按照所提示的解决方法，将“`set auto-load safe-path /`”添加到用户主目录（本例子是 `/home/lqm`，读者按自己目录修改）下的 `.gdbinit` 文件中。此时再启动 `gdb-silent kernel`，不仅没有警告提示，而且无需执行 `target remote localhost:26000`——因为 `/home/lqm/xv6-public-xv6-rev9/.gdbinit` 初始化脚本已经为我们准备好了。

屏显 1-6 修正.gdbinit 后再次运行 gdb kernel 命令

```

lqm@lqm-VirtualBox:~/xv6-public-xv6-rev9$ gdb kernel -silent
Reading symbols from kernel...done.
+ target remote localhost:26000
The target architecture is assumed to be i8086
[f000:ffff] 0xfffff0: jmp $0x3630,$0xf000e05b
0x0000ffff in ?? ()
+ symbol-file kernel
(gdb)

```

此时读者用 `gdb` 的 `c` 命令，将开始执行 `xv6` 内核代码，并在 `xv6` 终端上看到系统启动并执行 `shell` 的界面，如屏显 1-7 所示。同时，在 QEMU 输出窗口中显示了相同的启动过程。

屏显 1-7 xv6 启动运行

```

xv6...
cpu1: starting
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2          inodestart 32 bmap start 58
init: starting sh
$

```

本书其余部分将使用 QEMU+gdb 的调试方式来观察 `xv6` 的运行。

1.2.3. 多核调试

我们重新执行 `make qemu-gdb` 启动仿真器，并在另一终端执行 `gdb kernel` 调试，观察两个处理器核并行执行的情况，如屏显 1-8 所示。首先用 `b main` 将断点设置在内核 `main()` 入口，然后用 `c` 命令运行到该处。接着用多个 `n` 命令，逐个执行直到 `startothers()`，最后用 `info threads` 查看线程信息。

屏显 1-8 用 gdb 运行 kernel 到 startothers()

```

(gdb) b main

```

```

Breakpoint 1 at 0x80102f20: file main.c, line 19.
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x80102f20 <main>:    lea    0x4(%esp),%ecx

Thread 1 hit Breakpoint 1, main () at main.c:19
19 {
(gdb) n
=> 0x80102f2f <main+15>:  sub    $0x8,%esp
main () at main.c:20
20    kinit1(end, P2V(4*1024*1024)); // phys page allocator
(gdb) n
=> 0x80102f41 <main+33>:  call   0x801067e0 <kvmalloc>
21    kvmalloc(); // kernel page table

...省略，以节省篇幅

(gdb) n
=> 0x80102f94 <main+116>: mov     0x801112c4,%ebx
35    if(!lisp)
(gdb)
=> 0x80102fa5 <main+133>: sub     $0x4,%esp
37    startothers(); // start other processors
(gdb) info threads
Id      Target Id      Frame
* 1      Thread 1 (CPU#0 [running]) main () at main.c:37
  2      Thread 2 (CPU#1 [halted]) 0x000fd412 in ?? ()
(gdb)

```

上面屏显 1-8 显示的 info threads 输出表明当前 cpu1 对应的线程 2 还是停机 halted 状态。相应地，此时 QEMU 仿真终端中显示 cpu0 启动，但未见 cpu1 的启动信息，如图 1-6 所示。

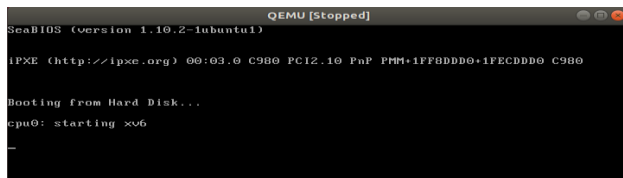


图 1-6 cpu0 启动而 cpu1 未启动时的 xv6

此时继续用 n 命令往下执行，当执行完 startothers() 之后，此时用 info threads 查看到 cpu1 对应的 thread2 已经退出 halted 状态并变为 running 状态，如屏显 1-9 所示。

屏显 1-9 执行 startothers() 启动 cpu1

```

...
(gdb) n
=> 0x80102fa5 <main+133>: sub     $0x4,%esp
37    startothers(); // start other processors
(gdb) i threads
Id      Target Id      Frame
* 1      Thread 1 (CPU#0 [running]) main () at main.c:38
  2      Thread 2 (CPU#1 [running]) getcallerpcs (pcs=0x801118ec <ptable+12>,
v=0x803befb0) at spinlock.c:77
(gdb)

```

相应地，QEMU 仿真窗口可以看到 cpu1 启动的信息，如图 1-7 所示。

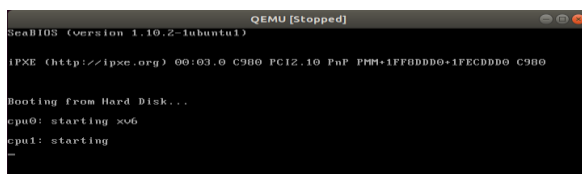


图 1-7 cpu0 和 cpu1 都已完成启动

如果执行 `thread 2` 命令，我们将调试器连接到线程 2 从而控制 `cpu1` 的运行。此时再用 `info threads` 命令查看，会发现线程 2 的前面会标注有“*”符号。

如果我们希望各个线程都受 `gdb` 控制而执行，而不是现在只控制其中一个线程，那么我们设置 `set scheduler-locking on`，反之设置为 `off`。我们可以控制调试命令施加到指定的线程上，具体命令的形式说明如下：

<code>thread apply ID1 ID2 command</code>	让指定的线程 (ID1/ID2...) 执行 GDB 命令 <code>command</code>
<code>thread apply all command</code>	让所有被调试线程执行 GDB 命令 <code>command</code> 。

1.3. 小结

本章完成了 `xv6` 实验系统的建立，分别给出了 `CentOS7` 和 `Ubuntu18` 上的 `QEMU` 安装和 `xv6` 的安装过程。出于方便考虑，`QEMU` 的宿主操作系统 `CentOS7` 和 `Ubuntu18` 可以安装在 `VirtualBox` 或其他虚拟机上。然后简单展示了 `xv6 shell` 中执行磁盘上的外部命令，如何用 `gdb` 调试 `xv6` 代码的运行，也包括对 `QEMU` 仿真的多核环境下不同执行流的跟踪。

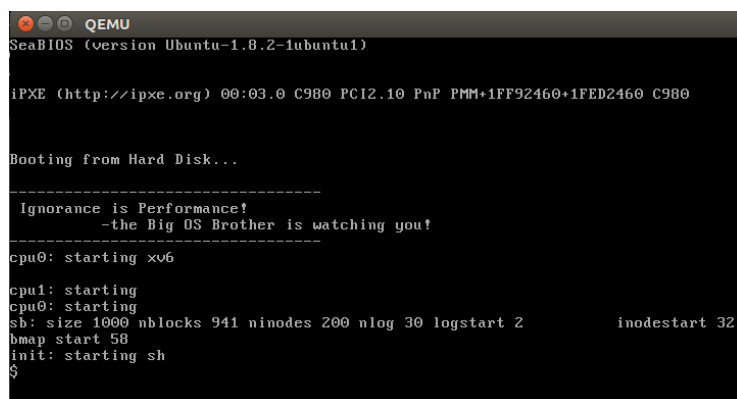
练习

1. 请修改 `Makefile` 中的“`CPUS :=2`”，将 `CPU` 数量给为 3 或 4，重行生成 `xv6` 的内核。注意观察启动时所打印的信息变化，然后用 `gdb` 将各个处理器所执行的代码分别停在 `scheduler()` 的不同指令处（或你感兴趣的其他代码处），并用 `info threads` 查看线程的数量是否与预期的一致。
2. 执行 `make qemu-nox` 运行 `xv6`，看看此时的显示方式和原来用的 `make qemu` 有什么不同。

第2章 入门实验

在深入学习和修改内核代码之前，我们先来在一些外围的编程操作。学会如何添加新的应用程序、如何增加新的系统调用，以后才能做一些更加复杂的实验修改。本章实验的完整代码可以从 <https://github.com/luoszu/xv6-exp/tree/master/code-examples/l1> 下载。

首先我们先来一个最简单的热身动作，修改 xv6 启动时的提示信息。打开 main.c 程序，将其中的 `cprintf()` 函数打印的启动提示信息“cpu0: starting xv6”修改成你所希望的样子，例如将它修改成图 2-1 所示的样子——标志着自己开始动手修改 xv6 操作系统了。



```
QEMU
SeaBIOS (version Ubuntu-1.8.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03:0 C980 PCI2.10 PnP PMM+1FF92460+1FED2460 C980

Booting from Hard Disk...

-----
Ignorance is Performance!
-the Big OS Brother is watching you!
-----
cpu0: starting xv6
cpu1: starting
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2      inodestart 32
bmap start 58
init: starting sh
$
```

图 2-1 修改后的启动提示信息

2.1. 新增可执行程序

为了给 xv6 添加新的可执行文件，需要了解 xv6 磁盘文件系统是如何生成的，然后才能编写应用程序并出现在 xv6 的磁盘文件系统中。

2.1.1. 磁盘映像的生成

现在我们先来了解 xv6 磁盘文件系统上的可执行文件是怎么生成的，包括两步骤：

- (1) 生成各个应用程序。
- (2) 将应用程序构成文件系统映像。

首先在 Makefile 中有一个默认规则，那就是所有的 *.c 文件都需要通过默认的编译命令生成 *.o 文件。另外在 Makefile 中有一个规则用于指出可执行文件的生成，如代码 2-1 所示。该模式规则说明，对于所有的 %.o 文件将结合 \$(ULIB) 一起生成可执行文件_%，比如说 ls.o（即依赖文件 \$^）将链接生成 _ls（即输出目标 \$@）。-Ttext 0 用于指出代码存放在 0 地址开始的地方，

-e main 参数指出以 main 函数代码作为运行时的第一条指令，-N 参数用于指出 data 节与 text 节都是可读可写、不需要在页边界对齐。

代码 2-1 Makefile 中可执行文件的生成规则

```
1.  ULIB = ulib.o usys.o printf.o umalloc.o
2.
3.  _%: %.o $(ULIB)
4.      $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^
5.      $(OBJDUMP) -S $@ > $*.asm
6.      $(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $*.sym
```

代码 2-2 是 Makefile 中有关创建磁盘文件系统的部分。其中变量 UPROGS 包含了所有相关的可执行文件名。磁盘文件系统 fs.img 目标依赖于 UPROGS 变量，并且将它们和 README 文件一起通过 mkfs 程序转换成文件系统映像 fs.img。

代码 2-2 Makefile 中创建磁盘文件系统的部分

```
1.  UPROGS=\
2.      _cat\
3.      _echo\
4.      _forktest\
5.      _grep\
6.      _init\
7.      _kill\
8.      _ln\
9.      _ls\
10.     _mkdir\
11.     _rm\
12.     _sh\
13.     _stressfs\
14.     _usertests\
15.     _wc\
16.     _zombie\
17.
18.  fs.img: mkfs README $(UPROGS)
19.      ./mkfs fs.img README $(UPROGS)
```

2.1.2. 添加简单程序

因此我们在 xv6 源码目录下，编写一个程序作为我们为 xv6 增加的一个应用程序。其中 types.h、stat.h 和 user.h 都是本目录中的头文件。此处的 printf() 的第一个参数用于指出输出文件，例如 0 号标准输入文件、1 号是标准输出文件，2 号是出错文件。程序运行结果是打印一行信息 “This is my own app!\n”。

代码 2-3 my-app.c

```
1.  #include "types.h"
2.  #include "stat.h"
3.  #include "user.h"
4.
5.  int
6.  main(int argc, char *argv[])
7.  {
8.      printf(1, "This is my own app!\n");
9.      exit();
10. }
```

然后我们修改 Makefile 中的 UPROGS 变量，添加一个 _my-app。然后可以执行 make，此时可以看到输出的 _my-app 文件，如屏显 2-1 所示。其中第二行 gcc 编译命令是由默认规则触发

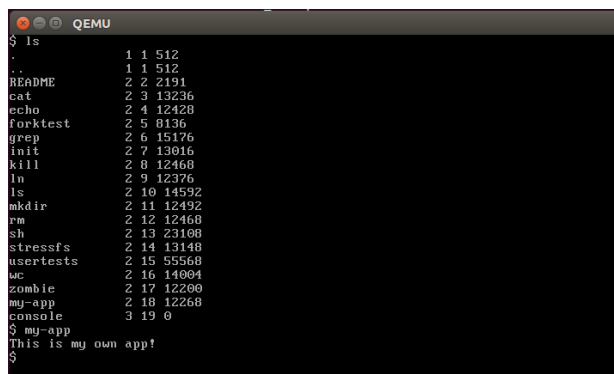
的，生成 `my-app.o`。第三、四、五行的命令是代码 2-1 的规则触发的，其中 `ld` 链接命令生成 `_my-app` 可执行文件。由于可执行文件发生了更新，因此触发了磁盘文件系统 `fs.img` 目标的规则——第六行显示的命令使用 `mkfs` 程序生成 `fs.img` 磁盘映像文件，其中可执行文件列表的最后一个就是我们刚生成的 `_my-app`。

最后用 `ll _my-app` 命令查看所生成的可执行文件，占用了 12268 字节的空间。

屏显 2-1 编译 my-app.c

1. `lqm@lqm-VirtualBox:~/xv6-public-xv6-rev9$ make`
2. `gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -c -o my-app.o my-app.c`
3. `ld -m elf_i386 -N -e main -Ttext 0 -o _my-app my-app.o ulib.o usys.o printf.o umalloc.o`
4. `objdump -S _my-app > my-app.asm`
5. `objdump -t _my-app | sed '1,/SYMBOL TABLE/d; s/./ /; / ^$/d' > my-app.sym`
6. `./mkfs fs.img README _cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh _stressfs _usertests _wc _zombie _my-app`
7. `nmeta 59 (boot, super, log blocks 30 inode blocks 26, bitmap blocks 1) blocks 941 total 1000`
8. `ballocc: first 590 blocks have been allocated`
9. `ballocc: write bitmap block at sector 58`
10. `dd if=/dev/zero of=xv6.img count=10000`
11. `记录了 10000+0 的读入`
12. `记录了 10000+0 的写出`
13. `5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0628553 s, 81.5 MB/s`
14. `dd if=bootblock of=xv6.img conv=notrunc`
15. `记录了 1+0 的读入`
16. `记录了 1+0 的写出`
17. `512 bytes copied, 0.000215976 s, 2.4 MB/s`
18. `dd if=kernel of=xv6.img seek=1 conv=notrunc`
19. `记录了 333+1 的读入`
20. `记录了 333+1 的写出`
21. `170532 bytes (171 kB, 167 KiB) copied, 0.000703954 s, 242 MB/s`
22. `lqm@lqm-VirtualBox:~/xv6-public-xv6-rev9$ ll _my-app`
23. `-rwxrwxr-x 1 lqm lqm 12268 3 月 7 19:42 _my-app*`
24. `lqm@lqm-VirtualBox:~/xv6-public-xv6-rev9$`

启动 `xv6` 系统后，执行 `my-app` 程序，正确地输出了我们期待的 “This is my own app!” 字符串，如图 2-2 所示。



```

QEMU
$ ls
.          1 1 512
..         1 1 512
README    2 2 2191
cat        2 3 13236
echo       2 4 12428
forktest   2 5 8136
grep       2 6 15176
init       2 7 13016
kill       2 8 12460
ln         2 9 12376
ls         2 10 14592
mkdir      2 11 12492
rm         2 12 12460
sh         2 13 23100
stressfs   2 14 13148
usertests  2 15 55560
wc         2 16 14004
zombie     2 17 12200
my-app     2 18 12268
console    3 19 0
$ my-app
This is my own app!
$

```

图 2-2 在 `xv6` 中运行新增的 `my-app` 程序

2.2. 新增系统调用

如果我们修改了 xv6 的代码，例如增加了调度优先级，那么就需要有设置优先级的系统调用，并且通过应用程序调用该系统调用进行优先级设置。因此我们需要学习如何增加新的系统调用，以及如何在应用程序中进行系统调用，后面才能验证 xv6 修改的功能。

比如，我们希望进程能知道自己所在的处理器编号，这可以通过一个新的系统调用来实现。下面我们先学习如何在应用程序中调用现成的系统调用，然后再学习如何实现上述的新的系统调用。

2.2.1. 系统调用示例

可用的系统调用都在**错误!未找到引用源。** user.h 中定义，我们在程序中直接使用即可。我们以获取进程号的 getpid() 系统调用为例，编写如代码 2-4 所示的 print-pid.c 代码。

代码 2-4 print-pid.c

```
1.  #include "types.h"
2.  #include "stat.h"
3.  #include "user.h"
4.
5.  int
6.  main(int argc, char *argv[])
7.  {
8.
9.      printf(1, "My PID is: %d\n", getpid());
10.     exit();
11. }
```

按照前面的方法修改 Makefile 并重新生成 xv6，启动后在 shell 中执行 print-pid 并成功打印进程号，如图 2-3 所示。

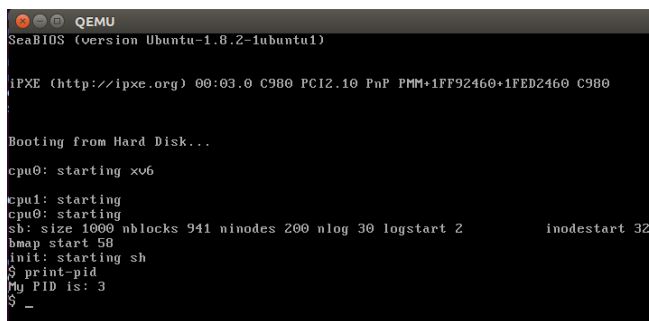


图 2-3 执行 print-pid 执行 getpid() 系统调用

2.2.2. 添加系统调用

由于系统调用涉及较多内容，分散在多个文件中——这包括系统调用号的分配、系统调用的分发代码（依据系统调用号）修改、系统调用功能的编码实现、用户库头文件修改等。另外还涉及验证用的样例程序——用于检验该系统调用的功能。

增加系统调用号

xv6 的系统调用都有一个唯一编号，定义在 `syscall.h` 中，如**错误!未找到引用源。**所示。我们可以在 `SYS_close` 的后面，新加入一行“`#define SYS_getcpuid 22`”即可，这里的编号 22 可以是其他值——只要不是前面使用过的就好。

增加用户态入口

为了让用户态代码能进行系统调用，需要提供用户态入口函数 `getcpuid()` 和相应的头文件。

■ 修改 `user.h`

为了让应用程序能调用用户态入口函数 `getcpuid()`，需要在**错误!未找到引用源。** `user.h` 中加入一行 函数原型声明“`int getcpuid(void);`”。该头文件应该被应用程序段源代码所使用，因为它声明了所有用户态函数的原型。除此之外所有标准 C 语言库的函数都不能使用，因为 `Makefile` 用参数“`-nostdinc`”禁止使用 Linux 系统的头文件，而且用“`-.l`”指出在当前目录中搜索头文件。也就是说 xv6 系统中，并没有实现标准的 C 语言库。

■ `usys.S` 中定义用户态入口

定义了 `getcpuid()` 原型之后，还需要实现 `getcpuid()` 函数。我们在**错误!未找到引用源。** `usys.S` 中加入一行“`SYSCALL(getcpuid)`”，例如可以插入到 `usys.s` 第 28 行后面。`SYSCALL` 是一个宏，定义于**错误!未找到引用源。** 第一行。`SYSCALL(getcpuid)` 将把“`getcpuid`”定义为函数入口，然后把 `SYS_getcpuid=22` 作为系统调用号保存到 `eax` 寄存器中，然后发出 `int` 指令进行系统调用“`int $T_SYSCALL`”。这样进入到系统调用公共入口后，以 `eax` 作为下标在系统调用表 `syscalls[]` 中找到需要执行的具体代码。

这里定义的 `getcpuid()` 函数，就是在需要执行系统调用时所调用的用户态函数，使得用户代码无需编写汇编指令来执行 `int` 指令。

修改 `syscall.c` 中的跳转表

在系统调用公共入口 `syscall()` 中，xv6 将根据系统调用号进行分发处理。负责分发处理的函数 `syscall()`（定义于**错误!未找到引用源。** `syscall.c`），分发依据是一个跳转表。我们需要这个修改跳转表，首先要在 `syscall.c` 第 102 行中的分发函数表 `syscalls[]` 中加入“`[SYS_getcpuid] sys_getcpuid,`”，也就是下标 22 对应的是 `sys_getcpuid()` 函数地址（后面我们会实现该函数）。其次，由于 `sys_getcpuid` 未声明，因此要在它前面（例如第 100 行后面的位置）加入一行“`extern int sys_getcpuid(void);`”用于指出该函数是外部符号。

前面提到：当用户发出 22 号系统调用是通过用户态函数 `getcpuid()` 完成的，其中系统调用号 22 是保存在 `eax` 的。因此 `syscall()` 系统调用入口代码可以通过 `proc->tf->eax` 获得该系统调用

号，并保存在 `num` 变量中，于是 `syscalls[num]` 就是 `syscalls[22]` 也就是 `sys_getcpuid()`。代码 2-5 是从 `syscall.c` 中截取的 `syscall()` 部分。

代码 2-5 系统调用分发代码 `syscall()`

```

1. void
2. syscall(void)
3. {
4.     int num;
5.
6.     num = proc->tf->eax;
7.     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
8.         proc->tf->eax = syscalls[num]();
9.     } else {
10.        cprintf("%d %s: unknown sys call %d\n",
11.            proc->pid, proc->name, num);
12.        proc->tf->eax = -1;
13.    }
14. }
```

实现 `sys_getcpuid()`

前面的工作使得用户可以用 `getcpuid()` 作为系统调用户态的入口，而且进入系统调用的分发例程 `syscall()` 中也能正确地转入到 `sys_getcpuid()` 函数里，但是我们还未实现 `sys_getcpuid()` 函数。如代码 2-6 所示，在 `sysproc.c` 中加入系统调用处理函数 `sys_getcpuid()`——注意和同名的用户态函数要区分开来。

代码 2-6 `sysproc.c` 中添加 `sys_getcpuid()`

```

1. int
2. sys_getcpuid()
3. {
4.     return getcpuid ();
5. }
```

然后在 `proc.c` 中实现内核态的 `getcpuid()` 函数，如代码 2-7 所示。

代码 2-7 `proc.c` 中添加 `getcpuid()`

```

1. int getcpuid()
2. {
3.     return cpunum();
4. }
```

为了让 `sysproc.c` 中的 `sys_getcpuid()` 能调用 `proc.c` 中的 `getcpuid()`，还需要在 `defs.h` 加入一行 “`int getcpuid(void);`”，用作内核态代码调用 `getcpuid()` 时的函数原型。`defs.h` 定义了 xv6 几乎所有的内核数据结构和函数，也被几乎所有内核代码所包含。

```

1. //PAGEBREAK: 16
2. // proc.c
3. void      exit(void);
4. int       fork(void);
5. int       growproc(int);
6. int       kill(int);
7. void      pinit(void);
8. void      procdump(void);
9. void      scheduler(void) __attribute__((noreturn));
10. void      sched(void);
11. void      sleep(void*, struct spinlock*);
12. void      userinit(void);
13. int       wait(void);
14. void      wakeup(void*);
15. void      yield(void);
16. int       getcpuid(void);
```

2.2.3. 验证新系统调用

最后，我们需要验证新增系统调用是否能被应用程序所正常使用。由于前面已经在 `user.h` 中声明了 `getcpuid()` 用户态函数原型，因此可以在应用程序中进行调用。编写如代码 2-8 所示的程序，打印本进程所在的 CPU 编号。参照前面 `my_app.c` 实验，完成其编译过程、加入到磁盘文件系统（记得要修改 `Makefile` 的 `UPROGS` 目标加上 `_pcpuid`）。

代码 2-8 `pcpuid.c`

```
1.  #include "types.h"
2.  #include "stat.h"
3.  #include "user.h"
4.
5.  int
6.  main(int argc, char *argv[])
7.  {
8.      printf(1, "My CPU id is :%d\n", getcpuid());
9.      exit();
10. }
```

执行 `pcpuid` 程序，将打印出本进程所在的处理器，如屏显 2-2 所示。

屏显 2-2 `pcpuid` 运行结果

```
$ pcpuid
cpu called from 801047c8 with interrupts enabled
My CPU id is :1
$
```

2.3. 观察调度过程

在本章结束之前，我们再编写一个程序，使得它可以创建多个进程并发运行，用于观察多进程分时运行的现象。我们将 `my-app.c` 稍作修改，调用两次 `fork()` 来创建（实际上是复制自己）新的进程，两次 `fork` 将一共创建 4 个进程。

代码 2-9 `my-app-fork.c`

```
1.  #include "types.h"
2.  #include "stat.h"
3.  #include "user.h"
4.
5.
6.  int
7.  main(int argc, char *argv[])
8.  {
9.      int a;
10.     printf(1, "This is my own app!\n");
11.     a=fork();
12.     a=fork();
13.     while(1)
14.         a++;
15.     exit();
16. }
```

按前面的 `my-app` 的方法，我们重新在磁盘文件系统中增加 `my-app-fork` 程序，运行后间断性地键入 `Ctrl+P` 用于显示当时的进程状态，如屏显 2-3 所示。在四次查看进程的状态中，发现第一次进程 3、6 在运行，第二次时进程 3、4 在运行，第三次只有进程 3 在运行，第四次有

进程 5、6 在运行。也就是说最多有两个进程能拥有 CPU，其中第三次进程 3 在一个 CPU 上跑，另一个 CPU 在运行 scheduler 执行流。

屏显 2-3 查看 my-app-fork 运行时的进程状态

```
$ my-app-fork
This is my own app!
1 sleep  init 80103e4f 80103ee9 80104789 801057a9 8010559b
2 sleep  sh 80103e4f 80103ee9 80104789 801057a9 8010559b
3 run    my-app-fork
4 runble my-app-fork
5 runble my-app-fork
6 run    my-app-fork

1 sleep  init 80103e4f 80103ee9 80104789 801057a9 8010559b
2 sleep  sh 80103e4f 80103ee9 80104789 801057a9 8010559b
3 run    my-app-fork
4 run    my-app-fork
5 runble my-app-fork
6 runble my-app-fork

1 sleep  init 80103e4f 80103ee9 80104789 801057a9 8010559b
2 sleep  sh 80103e4f 80103ee9 80104789 801057a9 8010559b
3 run    my-app-fork
4 runble my-app-fork
5 runble my-app-fork
6 runble my-app-fork

1 sleep  init 80103e4f 80103ee9 80104789 801057a9 8010559b
2 sleep  sh 80103e4f 80103ee9 80104789 801057a9 8010559b
3 runble my-app-fork
4 runble my-app-fork
5 run    my-app-fork
6 run    my-app-fork
```

2.4. 小结

在未阅读 xv6 内核代码之前，读者先完成两个 xv6 的入门小实验，可以近距离体验到内核修改所带来的成就感。除了本书组织的初级、中级和高级实验外，感兴趣的读者可以进一步完成各章的后面的练习或者直接学习美国大学的操作系统课程的实验内容（例如华盛顿大学的实验内容¹）。

练习

1. 请为 xv6 增加一个新的应用程序，读者自行选定其功能。
2. 定义一个内核全局变量，用于进程间共享的目的。设计并实现两个系统调用 `read_sh_var()` 和 `write_sh_var()` 用于读取和修改该全局变量的值。编写应用程序，检验是否能在进程间完成数值的共享。

¹ <https://courses.cs.washington.edu/courses/cse451/15au/index.html>