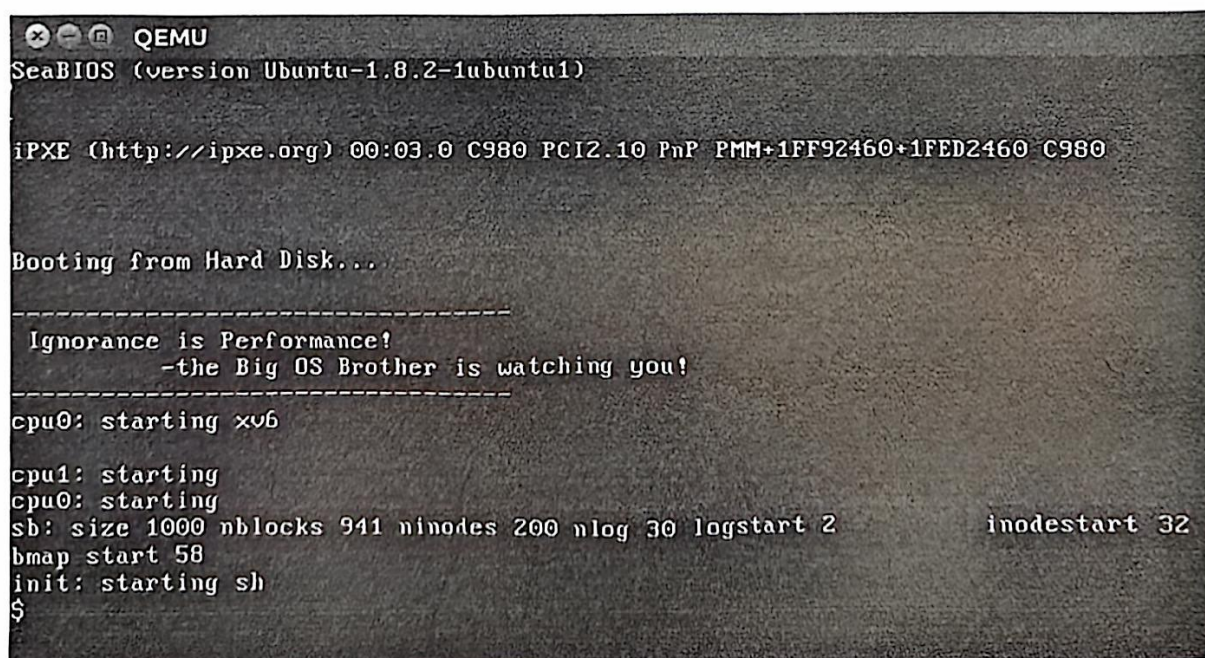


第 2 章

入门实验

在深入学习和修改内核代码之前,我们先来学习一些外围的编程操作。先学会如何添加新的应用程序,如何增加新的系统调用,之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/1> 下载。

首先我们先来实施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,这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,需要先了解 xv6 磁盘文件系统是如何生成的,然后才能编写出现在 xv6 磁盘文件系统中的应用程序。

2.1.1 磁盘映像的生成

现在先来了解 xv6 磁盘文件系统上的可执行文件是怎么生成的,其包括两个步骤:

- (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 中创建磁盘文件系统的相关部分。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

我们先在 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,如代码 2-3 所示。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 号是标准输入文件,1 号是标准错误输出文件,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. }
```



之后我们修改 Makefile 中的 UPROGS 变量,添加一个 `_my-app`。然后执行 `make`,此时可以看到输出的 `_my-app` 文件,如屏显 2-1 所示。其中第 2 行 `gcc` 编译命令是由默认规则触发的,生成 `my-app.o`。第 3、4、5 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 6 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`。

屏显 2-1 编译 `my-app.c`

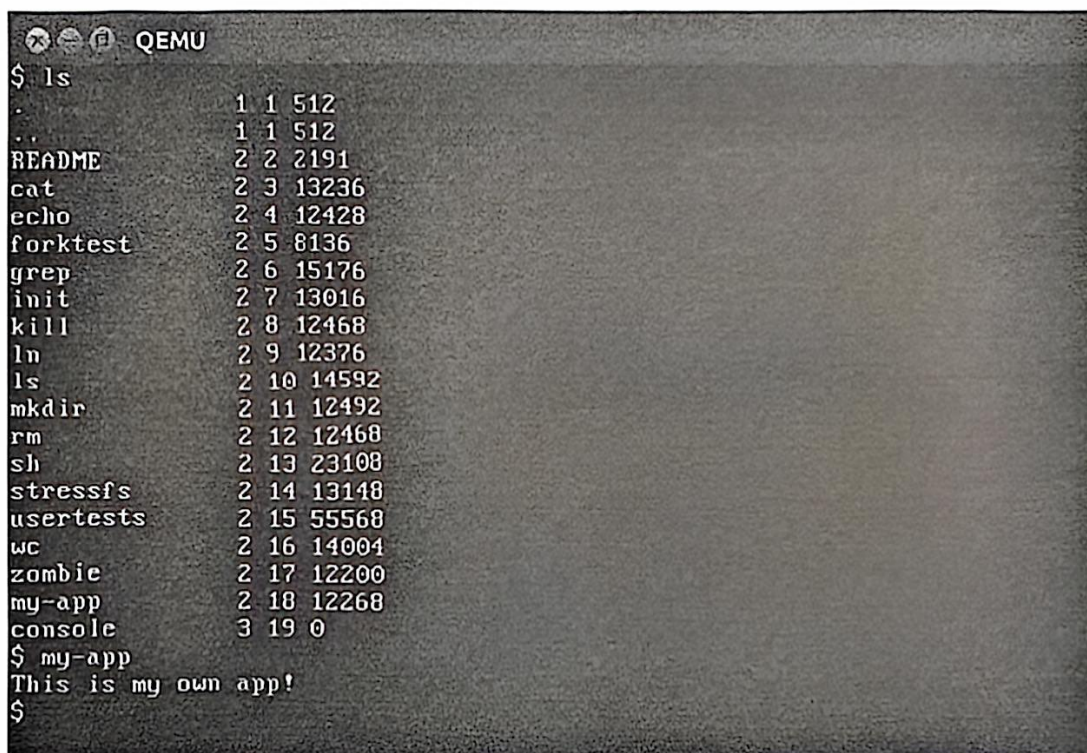
```
1. lqm@lqm-VirtualBox:~/xv6-public-xv6-rev9$ make
2. gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb
   -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -c -
   o my-app.o my-app.c
3. ld -m elf_i386 -N -e main -Ttext 0 -o my-app my-app.o ulib.o usys.o printf.
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

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。

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 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
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 xv6 的代码, 例如增加调度优先级, 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。下面我们需要先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, 之后才能验证修改后 xv6 的功能。

如果进程希望知道自身所在的处理器编号, 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系统调用。



2.2.1 系统调用示例

可用的系统调用都在代码 7-8 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。

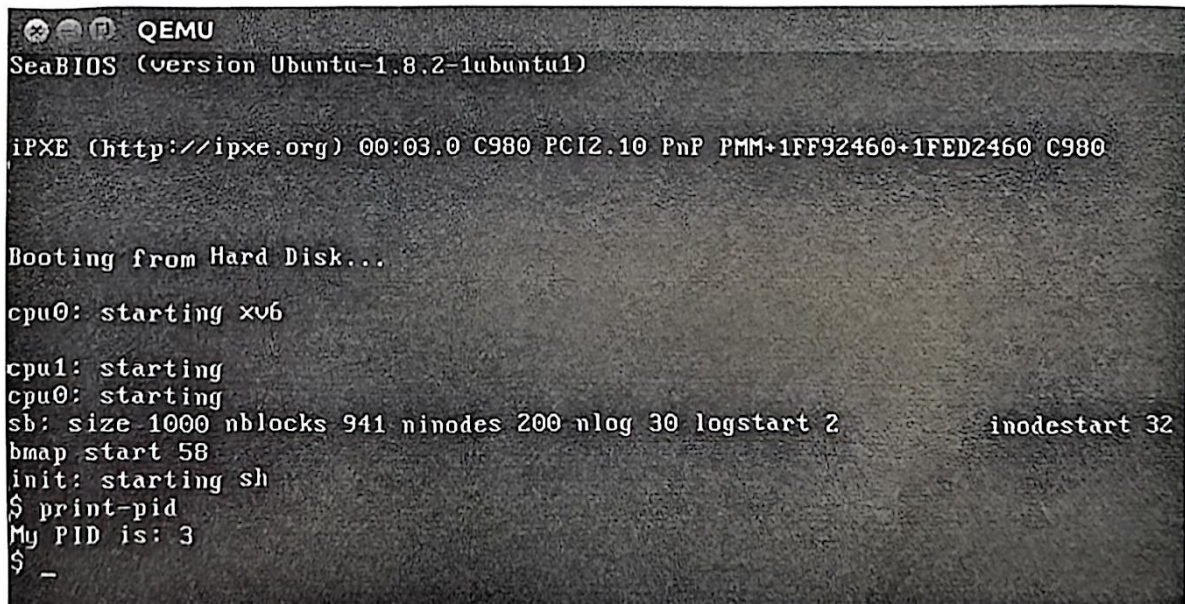


图 2-3 执行 `print-pid` 执行 `getpid()` 系统调用



2.2.2 添加系统调用

系统调用涉及内容较多且分散在多个文件之中,其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。

1. 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,如代码 7-5 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,只要该值不是前面使用过的就好。

2. 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()`,需要在代码 7-8 中加入一行函数原型声明“`int getcpuid(void);`”。该头文件会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“`-I.`”指出在当前目录中搜索头文件。也就是说 xv6 系统并没有实现标准的 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。我们在代码 7-9 中加入一行“`SYSCALL(getcpuid)`”,例如插入到 `usys.S` 第 28 行后面。`SYSCALL` 是一个宏,定义于代码 7-9 的第 4 行。`SYSCALL(getcpuid)` 经过宏展开,将“`getcpuid`”定义为入口函数名,然后把 `SYS_getcpuid=22` 作为系统调用号保存到 `eax` 寄存器中,最后发出 `int` 指令进行系统调用“`int $T_SYSCALL`”。这样,经过用户态函数 `getcpuid()`,借助 `int` 指令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[]` 中就可以找到需要执行的对应系统调用的具体代码。

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。

3. 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()` (定义于代码 7-7),分发依据是一个跳转表。我们需要修改这个跳



转表,首先要在代码 7-7 的第 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. }
```

4. 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()`,注意要与同名的用户态函数区分开来。

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

```
1. int
2. sys_getcpuid()
3. {
```



```

4.     return getcpuid ();
5. }

```

现在在 `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

下面, 我们验证应用程序是否能正常使用新增的系统调用。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对其进行调用。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编号,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,它可以创建多个进程并发运行,读者用之可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 调用将一共创建 4 个进程,如代码 2-9 所示。

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,其中第三次进程在一个 CPU 上运行,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 本章小结

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,感兴趣的读者还可以进一步完成各章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. 定义一个用于进程间共享的内核全局变量。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实现数值的共享。

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

