

master ▾

MIT6.S081 / lec01-introduction-and-examples /  
1.5-some-systemcalls.md

Go to file

...



huihongxiao GitBook: [master] 30 p...



Latest commit bc9c619 on Apr 24

History

1 contributor

60 lines (32 sloc) | 6.76 KB



Raw

Blame



## 1.5 read, write, exit系统调用

接下来，我将讨论对于应用程序来说，系统调用长成什么样。因为系统调用是操作系统提供的服务的接口，所以系统调用长什么样，应用程序期望从系统调用得到什么返回，系统调用是怎么工作的，这些还是挺重要的。你会在第一个lab中使用我们在这里介绍的系统调用，并且在后续的lab中，扩展并提升这些系统调用的内部实现。

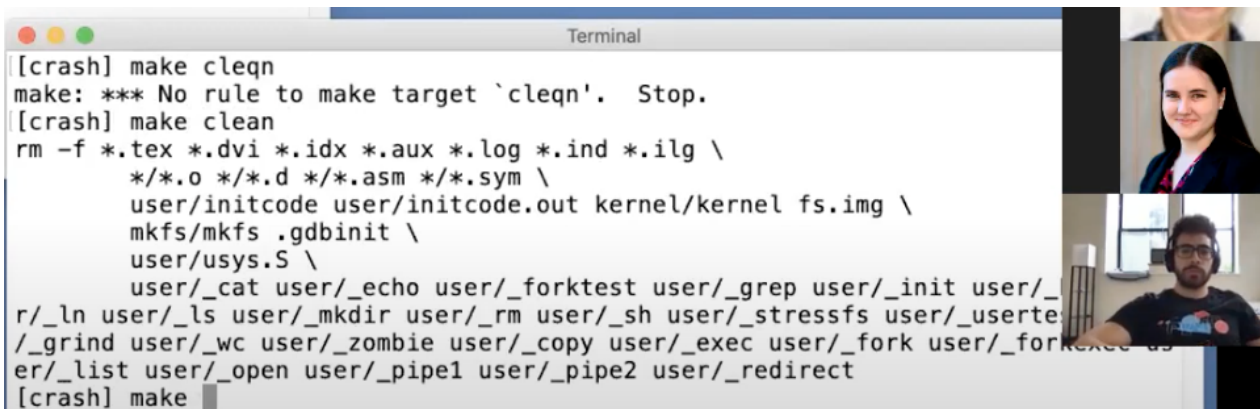
我接下来会展示一些简单的例子，这些例子中会执行系统调用，并且我会在XV6中运行这些例子。XV6是一个简化的类似Unix的操作系统，而Unix是一个老的操作系统，但是同时也是很多现代操作系统的基础，例如Linux，OSX。所以Unix使用的非常广泛。而作为我们教学用的操作系统，XV6就要简单的多。它是受Unix启发创造的，有着相同的文件结构，但是却要比任何真实的Unix操作系统都要简单的多。因为它足够简单，所以你们极有可能在几周内很直观的读完所有的代码，同时也把相应的书也看完，这样你们就能理解XV6内部发生的一切事情了。

XV6运行在一个RISC-V微处理器上，而RISC-V是MIT6.004课程讲解的处理器，所以你们很多人可能已经知道了RISC-V指令集。理论上，你可以在一个RISC-V计算机上运行XV6，已经有人这么做了。但是我们会在一个QEMU模拟器上运行XV6。

我这里会写下来，我们的操作系统是XV6，它运行在RISC-V微处理器上，当然不只是RISC-V微处理器，我们假设有一定数量的其他硬件存在，例如内存，磁盘和一个console接口，这样我们才能跟操作系统进行交互。但是实际上，XV6运行在QEMU模拟器之上。这样你们都能在没有特定硬件的前提下，运行XV6。

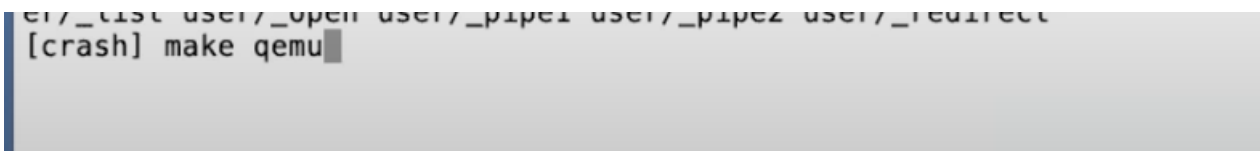
xv6  
RISC-V  
QEMU

接下来，我会展示一下代码。首先，我会在我的笔记本上设置好XV6。首先输入make qemu，你会发现你在实验中会经常用到这个命令。这个命令会编译XV6，而XV6是用C语言写的。我首先执行一下make clean，这样你们就能看到完整的编译过程。



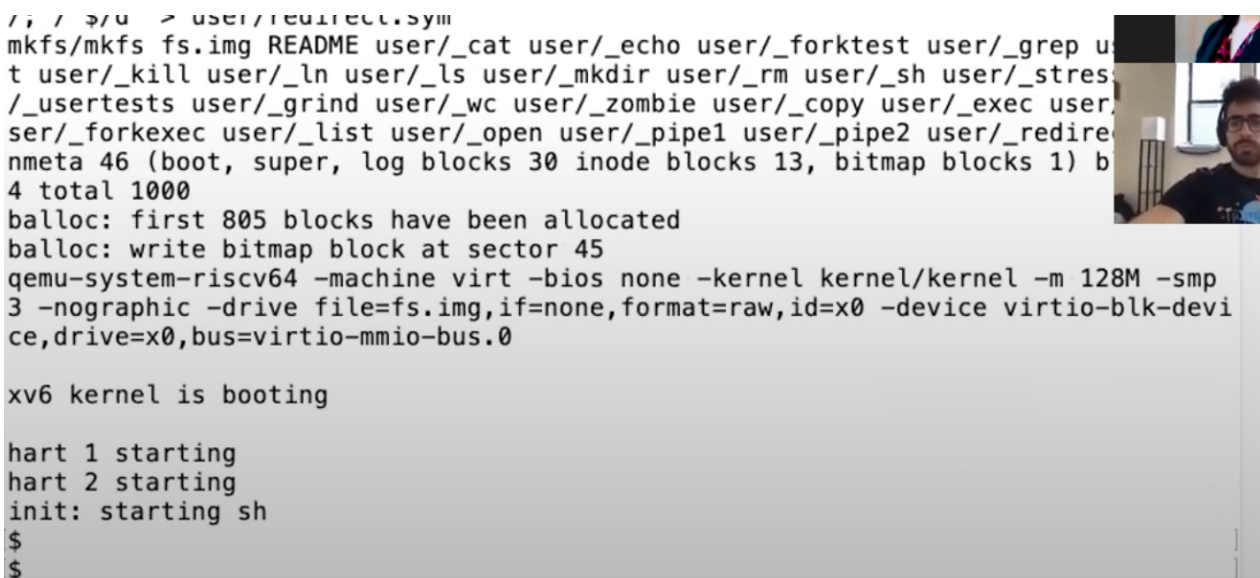
```
[crash] make cleqn
make: *** No rule to make target `cleqn'. Stop.
[crash] make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
    /*.o /*.d /*.asm /*.sym \
    user/initcode user/initcode.out kernel/kernel fs.img \
    mkfs/mkfs .gdbinit \
    user/usys.S \
    user/_cat user/_echo user/_forktest user/_grep user/_init user/_
r/_ln user/_ls user/_mkdir user/_rm user/_sh user/_stressfs user/_userte
/_grind user/_wc user/_zombie user/_copy user/_exec user/_fork user/_forkexec us
er/_list user/_open user/_pipe1 user/_pipe2 user/_redirect
[crash] make
```

之后我输入make qemu，这条指令会编译并构建xv6内核和所有的用户进程，并将它们运行在QEMU模拟器下。



```
er/_list user/_open user/_pipe1 user/_pipe2 user/_redirect
[crash] make qemu
```

编译需要花费一定的时间。

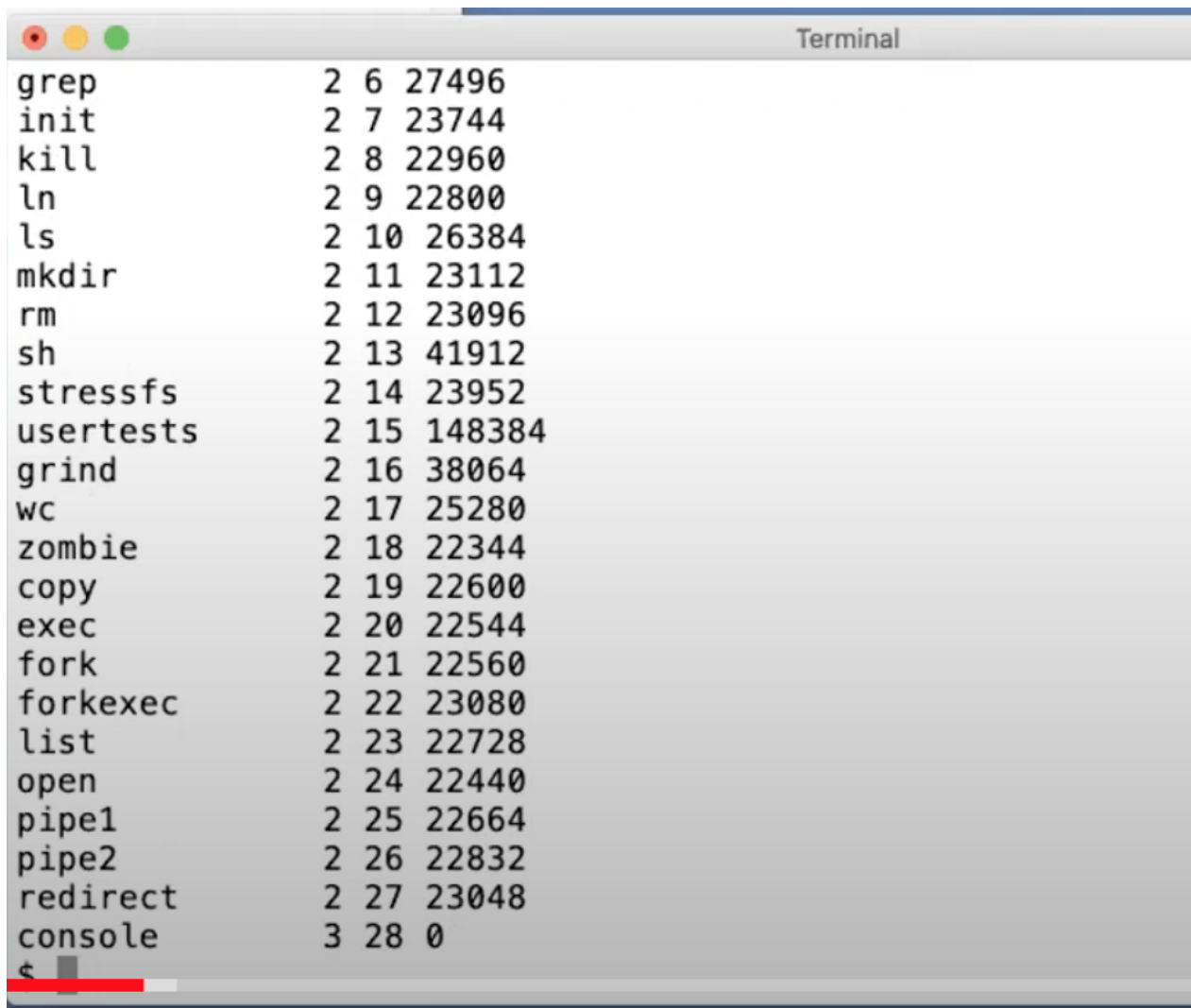


```
// / 3/0 > user/_redirect.sym
mkfs/mkfs fs.img README user/_cat user/_echo user/_forktest user/_grep u
t user/_kill user/_ln user/_ls user/_mkdir user/_rm user/_sh user/_stres
/_usertests user/_grind user/_wc user/_zombie user/_copy user/_exec user
ser/_forkexec user/_list user/_open user/_pipe1 user/_pipe2 user/_redire
nmeta 46 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 1) b
4 total 1000
ballocc: first 805 blocks have been allocated
ballocc: write bitmap block at sector 45
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-devi
ce,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$
$
```

现在xv6系统已经起来并运行了。\$表示Shell，这是参照Unix上Shell的命令行接口。如果你用过Athena工作站，它的Shell与这里的非常像。XV6本身很小，并且自带了一小部分的工具程序，例如ls。我这里运行ls，它会输出xv6中的所有文件，这里只有20多个。



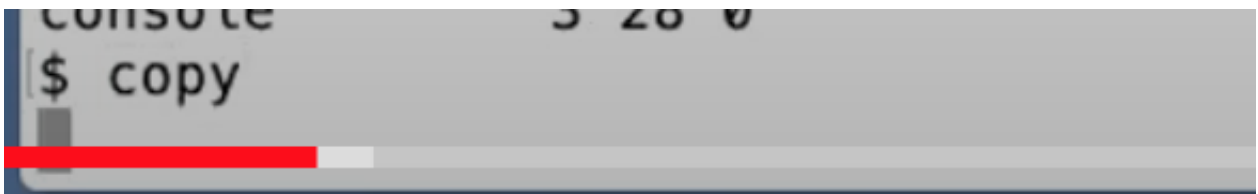
```
Terminal
grep      2  6  27496
init      2  7  23744
kill      2  8  22960
ln        2  9  22800
ls        2 10  26384
mkdir     2 11  23112
rm        2 12  23096
sh        2 13  41912
stressfs  2 14  23952
usertests 2 15 148384
grind     2 16  38064
wc        2 17  25280
zombie    2 18  22344
copy      2 19  22600
exec      2 20  22544
fork      2 21  22560
forkexec  2 22  23080
list      2 23  22728
open      2 24  22440
pipe1     2 25  22664
pipe2     2 26  22832
redirect  2 27  23048
console   3 28   0
$
```

可以看到，这里还有grep，kill，mkdir和rm，或许你们对这些程序很熟悉，因为它们在Unix中也存在。

我向你们展示的第一个系统调用是一个叫做copy的程序。

```
Terminal
[crash] cat -n copy.c
 1
 2 // copy.c: copy input to output.
 3
 4 #include "kernel/types.h"
 5 #include "user/user.h"
 6
 7 int
 8 main()
 9 {
10     char buf[64];
11
12     while(1){
13         int n = read(0, buf, sizeof(buf));
14         if(n <= 0)
15             break;
16         write(1, buf, n);
17     }
18
19     exit(0);
20 }
[crash]
```

它的源代码只有不到一页。你们这里看到的是一个程序，它从第8行的main开始，这是C程序的风格。它在第12行进入到一个循环中，在循环中，它会在第13行从输入读取一些数据，并在第16行，将数据写入到输出。如果我在XV6中运行这个copy程序，



它会等待输入。我随便输入一些字符，程序会读取我输入的字符，并将相同的字符输出给我。



所以这是一个非常简单的程序。如你所看到的，这个程序是用C语言写的，如果你不懂C语言，那最好还是去读一本标准的[C编程语言](#)。这个程序里面执行了3个系统调用，分别是read，write和exit。

如果你看第13行的read，它接收3个参数：

- 第一个参数是文件描述符，指向一个之前打开的文件。Shell会确保默认情况下，当一个程序启动时，文件描述符0连接到console的输入，文件描述符1连接到了console的输出。所以我可以通过这个程序看到console打印我的输入。当然，这里的程序会预期文件描述符已经被Shell打开并设置好。这里的0，1文件描述符是非常普遍的Unix风格，许多的Unix系统都会从文件描述符0读取数据，然后向文件描述符1写入数据。
- read的第二个参数是指向某段内存的指针，程序可以通过指针对应的地址读取

内存中的数据，这里的指针就是代码中的buf参数。在代码第10行，程序在栈里面申请了64字节的内存，并将指针保存在buf中，这样read可以将数据保存在这64字节中。

- read的第三个参数是代码想读取的最大长度，sizeof(buf)表示，最多读取64字节的数据，所以这里的read最多只能从连接到文件描述符0的设备，也就是console中，读取64字节的数据。

read的返回值可能是读到的字节数，在上面的截图中也就是6（xyzzzy加上结束符）。read可能从一个文件读数据，如果到达了文件的结尾没有更多的内容了，read会返回0。如果出现了一些错误，比如文件描述符不存在，read或许会返回-1。在后面的很多例子中，比如第16行，我都没有通过检查系统调用的返回来判断系统调用是否出错，但是你应该比我更加小心，你应该清楚系统调用通常是通过返回-1来表示错误，你应该检查所有系统调用的返回值以确保没有错误。

如果你想知道所有的系统调用的参数和返回值是什么，在XV6书籍的第二章有一个表格。

学生提问：如果read的第三个参数设置成1 + sizeof(buf)会怎样？

Robert教授：如果第三个参数是65字节，操作系统会拷贝65个字节到你提供的内存中（第二个参数）。但是如果栈中的第65个字节有一些其他数据，那么这些数据会被覆盖，这里是个bug，或许会导致你的代码崩溃，或者一些异常的行为。所以，作为一个程序员，你必须要小心。C语言很容易写出一些编译器能通过的，但是最后运行时出错的代码。虽然很糟糕，但是现实就是这样。

有一件事情需要注意的事，这里的copy程序，或者说read，write系统调用，它们并不关心读写的数据格式，它们就是单纯的读写，而copy程序会按照8bit的字节流处理数据，你怎么解析它们，完全是用应用程序决定的。所以应用程序可能会解析这里的数据为C语言程序，但是操作系统只会认为这里的数据是按照8bit的字节流。