

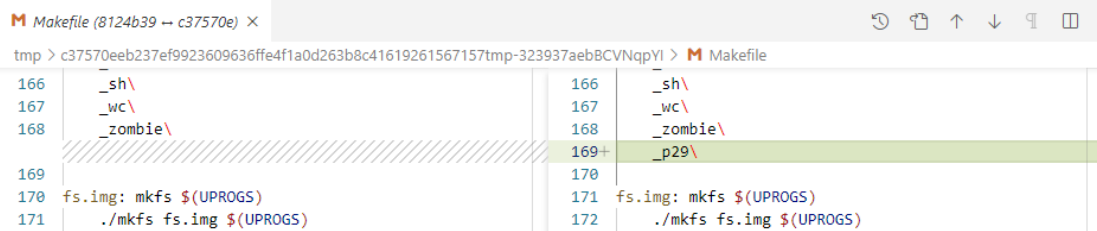
第 0 次操作系统实验报告

161810129 董世晨

一、 添加用户命令

由于实验要求添加的命令和已有的echo命令很类似，我全局搜索了echo命令出现的地方，对应地添加了p29命令。具体改动和起到的作用如下所示：

1. Makefile



```
tmp > c37570eeb237ef9923609636ffe4f1a0d263b8c41619261567157tmp-323937aeb8CVNqpYI > M Makefile
166 | _sh\
167 | _wc\
168 | _zombie\
169+| _p29\
170 | fs.img: mkfs $(UPROGS)
171 | ./mkfs fs.img $(UPROGS)
```

在Makefile的UPROGS变量中添加_p29，让mkfs生成p29命令。

2. p29.c

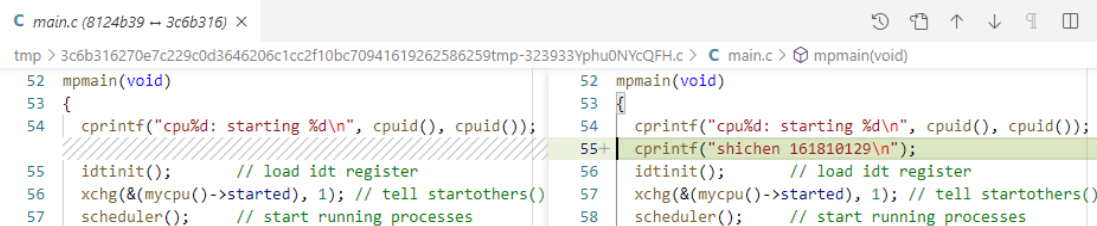
添加p29.c文件，模仿echo.c，根据main函数的argc判断是否要提前输出'\n'，输出空格分隔的argv中的字符串。

编译后成功运行，运行结果如下图所示：

```
$ p29
OS Lab 161810129:
$ p29 Hello, Clubie!
OS Lab 161810129: Hello, Clubie!
$
```

二、 添加内核输出语句

根据实验报告的指示，我全局搜索了cpu%d: starting %d，在void mpmain(void)函数中找到了该cprintf输出语句，并在其后添加了cprintf("shichen 161810129\n");



```
C main.c (8124b39 -> 3c6b316) X
tmp > 3c6b316270e7c229cd3646206c1cc2f10bc70941619262586259tmp-323933Yphu0NYcQFH.c > C main.c > mpmain(void)
52 | mpmain(void)
53 | {
54 |     cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
55+|     cprintf("shichen 161810129\n");
56 |     idtinit(); // load idt register
57 |     xchg(&(mycpu()->started), 1); // tell startothers()
58 |     scheduler(); // start running processes
```

观察到，前一个实验任务中，p29命令运行在用户空间，输出使用了printf函数，而这个实验任务中，mpmain函数运行在内核空间，输出使用了cprintf函数。

我进而详细阅读了这两个输出函数的源码。cprintf函数内部使用的是consputc函数来输出每一个字符，而consputc函数又调用了uartputc函数和cgaputc函数，这两个函数都是使用out指令和in指令来直接操作控制台显示字符和移动光标的。printf函数和cprintf大相径庭，它使用putc函数来输出每一个字符，而putc又调用了write这个系统调用，这是因为printf函数运行在用户空间，无法直接使用out和in这两个内核指令，所以

需要使用系统调用来间接地输出。

另一方面，`cprintf`函数只能向控制台输出，而`printf`函数可以通过它的第一个参数`fd`来控制输出到哪里，当`fd`为1时表示标准输出。

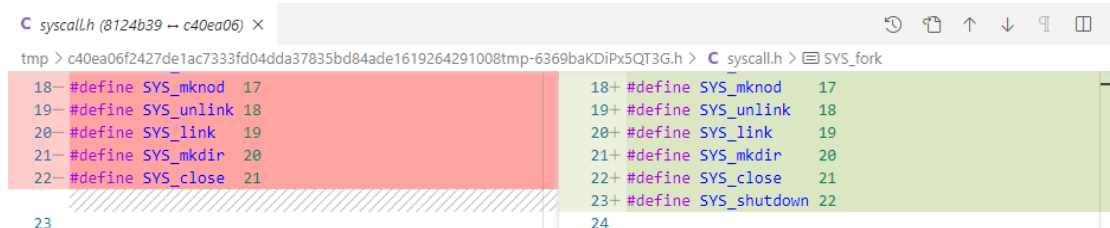
编译后，运行结果如下图所示：

```
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xvfb...
cpu0: starting 0
shichen 161810129
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$
```

三、 添加系统调用

我参考了`write`这个系统调用，模仿地添加了`shutdown`系统调用。具体的改动和起到的作用如下所示：

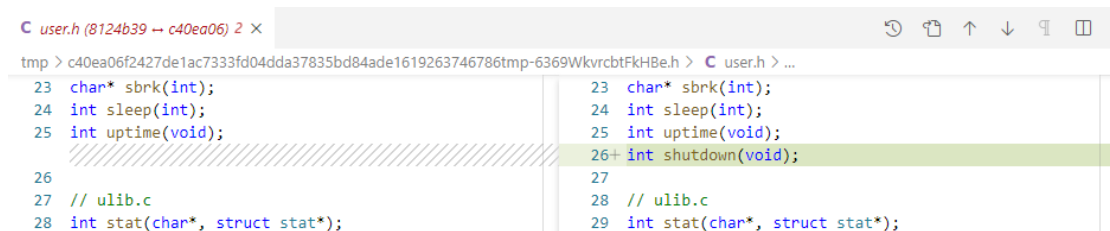
1. syscall.h



```
C syscall.h (8124b39 → c40ea06) X
tmp > c40ea06f2427de1ac7333fd04dda37835bd84ade1619264291008tmp-6369baKDIPx5QT3G.h > C syscall.h > SYS_fork
18- #define SYS_mknod 17
19- #define SYS_unlink 18
20- #define SYS_link 19
21- #define SYS_mkdir 20
22- #define SYS_close 21
23
18+ #define SYS_mknod 17
19+ #define SYS_unlink 18
20+ #define SYS_link 19
21+ #define SYS_mkdir 20
22+ #define SYS_close 21
23+ #define SYS_shutdown 22
24
```

在`syscall.h`中注册`shutdown`系统调用对应的系统调用号为22。

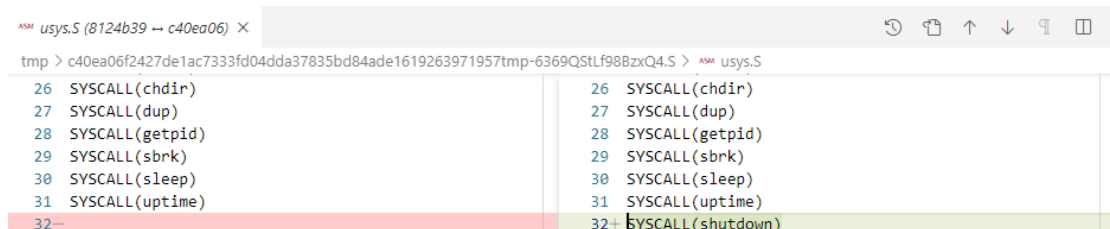
2. user.h



```
C user.h (8124b39 → c40ea06) 2 X
tmp > c40ea06f2427de1ac7333fd04dda37835bd84ade1619263746786tmp-6369WkvrcbtFkHBe.h > C user.h > ...
23 char* sbrk(int);
24 int sleep(int);
25 int uptime(void);
26
27 // ulib.c
28 int stat(char*, struct stat*);
23 char* sbrk(int);
24 int sleep(int);
25 int uptime(void);
26+ int shutdown(void);
27
28 // ulib.c
29 int stat(char*, struct stat*);
```

在`user.h`中声明`shutdown`函数。它在`usys.S`中用汇编实现。

3. usys.S



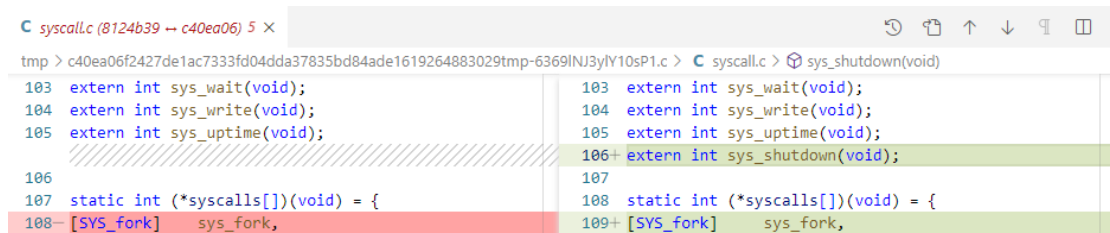
```
ASM usys.S (8124b39 → c40ea06) X
tmp > c40ea06f2427de1ac7333fd04dda37835bd84ade1619263971957tmp-6369QStLf98BzxQ4.S > ASM usys.S
26 SYSCALL(chdir)
27 SYSCALL(dup)
28 SYSCALL(getpid)
29 SYSCALL(sbrk)
30 SYSCALL(sleep)
31 SYSCALL(uptime)
32-
26 SYSCALL(chdir)
27 SYSCALL(dup)
28 SYSCALL(getpid)
29 SYSCALL(sbrk)
30 SYSCALL(sleep)
31 SYSCALL(uptime)
32+ SYSCALL(shutdown)
```

在`usys.S`中添加`SYSCALL(shutdown)`，也就实现了之前在`user.h`中声明的`shutdown`函数。该`SYSCALL(shutdown)`宏展开后为：

```
.globl shutdown;
shutdown:
    movl $SYS_shutdown, %eax;
    int $T_SYSCALL;
    ret
```

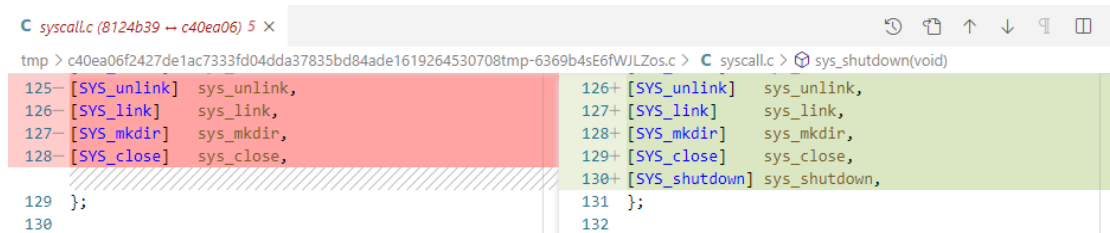
可以看到，该宏定义了shutdown这个全局符号，将eax置为SYS_shutdown，并用int指令触发中断。

4. syscall.c



```
C syscall.c (8124b39 ↔ c40ea06) 5 ×
tmp > c40ea06f2427de1ac7333fd04dda37835bd84ade1619264883029tmp-63691NJ3yY10sP1.c > C syscall.c > sys_shutdown(void)
103 extern int sys_wait(void);
104 extern int sys_write(void);
105 extern int sys_uptime(void);
106 //////////////////////////////////////////////////
107 static int (*syscalls[])(void) = {
108- [SYS_fork] sys_fork,
106+ extern int sys_shutdown(void);
107
108 static int (*syscalls[])(void) = {
109+ [SYS_fork] sys_fork,
```

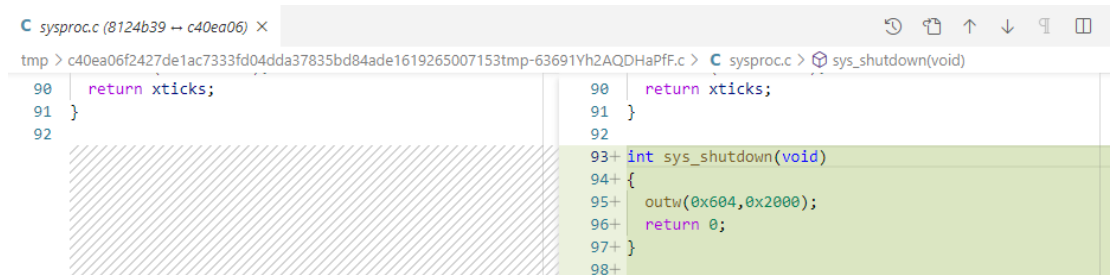
在syscall.c中声明外部函数sys_shutdown，该函数在sysproc.c中实现。



```
C syscall.c (8124b39 ↔ c40ea06) 5 ×
tmp > c40ea06f2427de1ac7333fd04dda37835bd84ade1619264530708tmp-6369b4sE6fWJLZos.c > C syscall.c > sys_shutdown(void)
125- [SYS_unlink] sys_unlink,
126- [SYS_link] sys_link,
127- [SYS_mkdir] sys_mkdir,
128- [SYS_close] sys_close,
129 };
130
126+ [SYS_unlink] sys_unlink,
127+ [SYS_link] sys_link,
128+ [SYS_mkdir] sys_mkdir,
129+ [SYS_close] sys_close,
130+ [SYS_shutdown] sys_shutdown,
131 };
132
```

在syscall.c中的syscalls函数指针数组中注册sys_shutdown函数。这里使用了C99标准引入的指派符列表初始化 (https://zh.cppreference.com/w/c/language/array_initialization)，可以优雅地将系统调用号和函数指针对应起来。

5. sysproc.c



```
C sysproc.c (8124b39 ↔ c40ea06) ×
tmp > c40ea06f2427de1ac7333fd04dda37835bd84ade1619265007153tmp-63691Yh2AQDHaPf.c > C sysproc.c > sys_shutdown(void)
90 return xticks;
91 }
92
93+ int sys_shutdown(void)
94+ {
95+     outw(0x604, 0x2000);
96+     return 0;
97+ }
98+
```






经查阅资料，xv6的系统调用的实现都出现在sysproc.c和sysfile.c这两个文件中，前者实现关于进程的系统调用，后者实现关于文件的系统调用。我认为shutdown系统调用更与进程相关，故将其放在sysproc.c文件的最后。该函数简单地调用outw函数实现关机操作。

编译后，运行结果如下图所示：








```
xv6...
cpu0: starting 0
shichen 161810129
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ shutdown
shichen@VM-199-186-ubuntu:~/project$ █
```

四、 其他

为了更好地记录我在xv6项目上做的每个改动，我使用了git版本管理工具。

● 使make submit的diff命令忽略.gitignore指定的文件 Cluble on 4/23/2021, 9:35:07 PM	da8ccdf  Soft Hard + Tag + Branch + More
● Project 0: 添加shutdown系统调用 Cluble on 4/21/2021, 7:06:58 PM	c40ea06  Soft Hard + Tag + Branch + More
● Project 0: 添加内核输出语句 Cluble on 4/21/2021, 6:41:34 PM	3e8b316  Soft Hard + Tag + Branch + More
● Project 0: 添加p29用户命令 Cluble on 4/21/2021, 5:29:58 PM	c37570e  Soft Hard + Tag + Branch + More
● 提交XV6原始代码 Cluble on 4/21/2021, 5:29:23 PM	8124e39  Soft Hard + Tag + Branch + More

但是，简历git仓库之后，使用make submit提交时，diff指令会生成很多.git文件夹中的差异内容，大幅增加了patch文件体积，为了解决这个问题，我修改了submit目标中的diff命令，使其能够读取.gitignore文件，从而忽略掉所有不被git管理的文件。

<div>  Makefile (c40ea06 ← da8ccdf) </div> <div> tmp > da8ccdf54a7708ef73c02827e7158617d3958f921619265519964tmp-6369cDo0milOyyqZ > Makefile </div> <div> <pre> 221 submitdir: 222 mkdir -p \${sfolder} 223 224 submit: clean submitdir 225- diff -uNr /home/proj0-base . > \${sfolder}/proj0.patch; [\$ 226- 227 </pre> </div>	<div>       </div> <div> <pre> 221 submitdir: 222 mkdir -p \${sfolder} 223 224 submit: clean submitdir 225- diff -uNrX .gitignore /home/proj0-base . > \${sfolder}/proj0.p 226 </pre> </div>
--	--