第 1 次操作系统实验报告

161810129 董世晨

一、 带参数的系统调用

1. 改动的内容

在shutdown.c中从命令行参数读入参数，将字符串转换成数字，调用shutdown系统调用，并补充必要的错误提示。由于xv6原有的atoi函数不提供校验功能，这里我不得不自己重写一个。

```
#include "types.h"
#include "stat.h"
#include "user.h"

extern int shutdown();

int
main(int argc, char *argv[])
{
  shutdown();
  exit();
}
```

```
#include "types.h"
#include "stat.h"
#include "user.h"

extern int shutdown(int a);

void print_usage_and_exit()
{
  printf(1, "usage:\n    shutdown <code>    Shutdown with leave code\n");
  exit();
}

int
main(int argc, char *argv[])
{
  if (argc != 2)
    print_usage_and_exit();
  // Sadly `atoi` defined in `ulib.c` does not validate the input string,
  // I need to implement my own version.
  int a = 0;
  for (char *p = argv[1]; *p; ++p)
  {
    if (*p < '0' || *p > '9')
      print_usage_and_exit();
    a = a * 10 + (*p - '0');
  }
  shutdown(a);
  exit();
}
```

通过模仿sleep，在sysproc.c中的sys_shutdown函数里，使用argint函数获取第一个传入的参数，保存到a中，进行输出。最后使用outw指令关机。

```

int
sys_shutdown(void){

  outw(0x604, 0x2000);
  return 0;
}
```

```

int
sys_shutdown(void){
  int a;
  if (argint(0, &a) < 0)
    return -1;
  cprintf("Leaving with code %d.\n", a);
  outw(0x604, 0x2000);
  return 0;
}
```

2. 运行结果

```
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodes
init: starting sh
$ shutdown 161810129
Leaving with code 161810129.
shichen@VM-199-186-ubuntu:~/project$
```

3. 回答问题

Question 1：esp+4+4*n中第一个4是什么意思？

Answer 1：esp~esp+4中保存了系统调用的返回地址，由于xv6是32位系统，地址长度为32位，即4字节。

Question 2：esp+4+4*n中第二个4是什么意思？

Answer 2：在xv6中，定义int为32位，即4字节，不仅如此，所有系统调用的参数都是4字节宽的，因此可以很方便地表示第n个参数的地址。（注：C标准规定int至少为16位，但

在xv6和大部分系统中，int为32位。[1](Cpp Reference)）

Question 3：为什么不是esp+4*n？

Answer 3：因为esp~esp+4中保存了系统调用的返回地址，从esp+4开始才是参数所在的地址。

Question 4：为什么不是esp-4-4*n？

Answer 4：因为xv6的栈是从上往下压栈的，函数参数位于高地址处。

```
                +----------------------------+
0x104FF4    +->| Parameter 2 for routine 2  | \  <-- Routine 1's EBP
            |   +----------------------------+  |
0x104FF0    |   | Parameter 1 for routine 2  |  |
            |   +----------------------------+  |
0x104FEC    |   | Return address, routine 1  |  |
            |   +----------------------------+  |
```

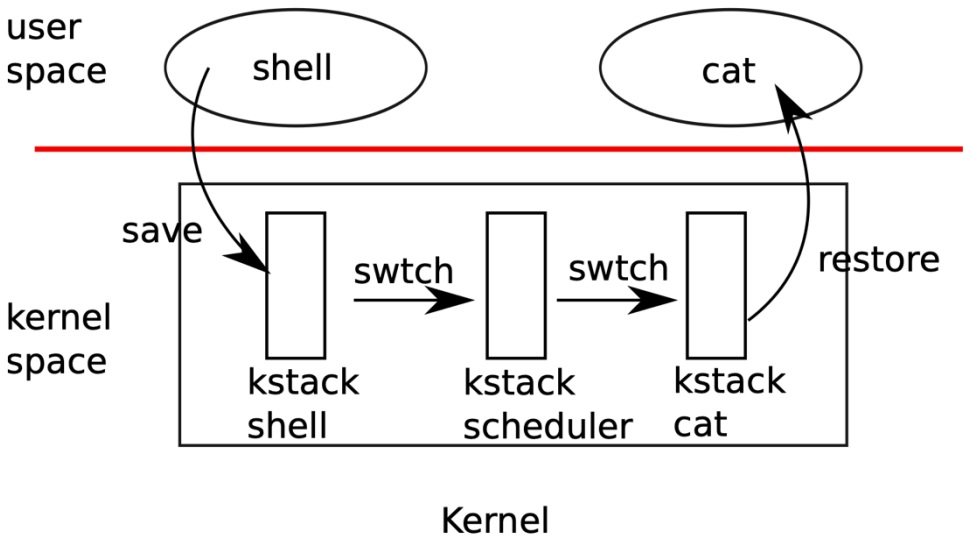
Question 5：为什么是esp+4+4*n，而不是ebp-4*n？

Answer 5：esp记录了栈顶位置，ebp记录了当前栈帧的底部。因为x86调用约定**函数参数逆序入栈**，即第一个参数在栈顶附近（靠近esp处），最后一个参数在当前栈帧的底部附近（靠近ebp处），使用ebp-4*n可以获取倒数第n个函数参数。[2](Wikipedia)

二、 理解进程切换

1. 进程切换

不是的，scheduler在切换进程时，需要进行两次上下文的切换。第一次发生在从旧进程切换到当前CPU的scheduler所在的内核进程，第二次发生在从scheduler所在的内核进程切换到新进程。



如上图所示，切换进程的完整流程是这样的：

1) 用户进程通过中断或系统调用陷入内核，从进程的用户栈进入进程的内核栈，并获得内核权限；
2) 通过swtch函数切换到当前CPU的scheduler函数，进入内核的栈；
3) scheduler选择下一个要执行的进程；
4) 通过swtch函数切换到下个进程的内核栈；
5) 通过trapret函数取消内核权限，回到用户进程，进入进程的用户栈。

2. 内核的栈 & 进程的内核栈 & 进程的用户栈

我首先展示三个栈一般所在的地址，方便后续对照。

内核的main函数运行在内核的栈上。在内核的main函数上打断点，观察main函数的寄存器状态，可见内核的栈在0x8010B000处。

```
(gdb) b main
Breakpoint 1 at 0x80102eb0: file main.c, line 19.
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x80102eb0 <main>:   lea    0x4(%esp),%ecx

Breakpoint 1, main () at main.c:19
19      {
(gdb) info reg
eax            0x80102eb0       -2146423120
ecx            0x0      0
edx            0x1f0    496
ebx            0x10074  65652
esp            0x8010b5c0       0x8010b5c0 <bcache>
ebp            0x7bf8   0x7bf8
esi            0x10074  65652
edi            0x0      0
eip            0x80102eb0       0x80102eb0 <main>
eflags         0x86     [ PF SF ]
cs             0x8      8
ss             0x10     16
ds             0x10     16
```

任意执行一个用户程序，如echo。echo的main函数在进程的用户栈上执行，在其上打断点观察各寄存器状态，可见此进程的用户栈在0x2000处。

```
(gdb) b main
Note: breakpoint 1 also set at pc 0x0.
Breakpoint 2 at 0x0: file echo.c, line 7.
(gdb) c
Continuing.
=> 0x0 <main>:  lea    0x4(%esp),%ecx

Breakpoint 1, main (argc=2, argv=0x2fe8) at echo.c:7
7       {
(gdb) info reg
eax            0x0      0
ecx            0x18a0   6304
edx            0xbfac   49068
ebx            0xbfa8   49064
esp            0x2fdc   0x2fdc
ebp            0x3fb8   0x3fb8
esi            0x0      0
edi            0x0      0
eip            0x0      0x0 <main>
eflags         0x206    [ PF IF ]
cs             0x1b     27
ss             0x23     35
ds             0x23     35
```

当echo进程触发write系统调用时，会陷入内核，进入进程的内核栈。在write函数上打断点，执行完int指令后，观察各寄存器状态，可见此进程的内核栈在0x8DFBE000处，和内核的栈不同。

```
(gdb) si
=> 0x2e7 <write+5>:     int    $0x40
0x000002e7      16      SYSCALL(write)
(gdb) si
=> 0x80105d99:  push   $0x40
0x80105d99 in ?? ()
(gdb) info reg
eax            0x10     16
ecx            0x2ff4   12276
edx            0x2fe8   12264
ebx            0x1      1
esp            0x8dfbefe8       0x8dfbefe8
ebp            0x2f98   0x2f98
esi            0x2ff5   12277
```

3. CPU在执行scheduler()时运行在用户态还是内核态？运行在哪个栈上面？

scheduler函数运行在内核态，且运行在内核的栈上。从下图可以看到，scheduler所在的栈为0x8010B000，和之前看到的内核的栈地址一致，且CS寄存器后两位为00，表明CPU在内核态。

```
(gdb) bt
#0  0x801039c3 in scheduler () at proc.c:336
#1  0x80102e8f in mpmain () at main.c:57
#2  0x80102fcf in main () at main.c:37
(gdb) info reg
eax            0x80112d54       -2146357932
ecx            0x80112d54       -2146357932
edx            0x4      4
ebx            0x80112d54       -2146357932
esp            0x8010b550       0x8010b550 <stack+3984>
ebp            0x8010b578       0x8010b578 <stack+4024>
esi            0x80112780       -2146359424
edi            0x80112784       -2146359420
eip            0x801039c3       0x801039c3 <scheduler+51>
eflags         0x46     [ PF ZF ]
cs             0x8      8
ss             0x10     16
ds             0x10     16
es             0x10     16
```

4. 当你在命令行敲下shutdown时，系统会创建一个进程执行shutdown.c中的代码，当CPU执行以下三条指令时movl $SYS_shutdown,%eax; int $T_SYSCALL; ret时，CPU运行在用户态还是内核态？运行在哪个栈上面？

在shutdown函数上打断点，之后单指令执行，当shutdown运行int指令之前（还未执行int指令）时，观察寄存器状态，可见当前所在的栈为0x2000，是进程的用户栈，且CS寄存器后两位为11，表明CPU在用户态。

```
(gdb)
=> 0x387 <shutdown+5>:  int    $0x40
0x00000387      32      SYSCALL(shutdown)
(gdb) info reg
eax            0x16     22
ecx            0x1      1
edx            0x2ff1   12273
ebx            0xbfa8   49064
esp            0x2fac   0x2fac
ebp            0x2fc8   0x2fc8
esi            0x0      0
edi            0x0      0
eip            0x387    0x387 <shutdown+5>
eflags         0x216    [ PF AF IF ]
cs             0x1b     27
ss             0x23     35
ds             0x23     35
```

使用si命令执行int之后，再次观察寄存器状态，可见此时所在的栈为0x8DFBE000，

是进程的内核栈，且CS寄存器后两位为00，表明CPU在内核态。

```
(gdb) si
=> 0x80105d99:  push   $0x40
0x80105d99 in ?? ()
(gdb) symbol-file kernel
Load new symbol table from "kernel"? (y or n) y
Reading symbols from kernel...done.
(gdb) info reg
eax            0x16     22
ecx            0x1      1
edx            0x2ff1   12273
ebx            0xbfa8   49064
esp            0x8dfbefe8       0x8dfbefe8
ebp            0x2fc8   0x2fc8
esi            0x0      0
edi            0x0      0
eip            0x80105d99       0x80105d99 <vector64+2>
eflags         0x216    [ PF AF IF ]
cs             0x8      8
ss             0x10     16
ds             0x23     35
```

5. **在执行命令shutdown的过程中，当cpu执行到涉及特权指令的函数outw()时，CPU运行在用户态还是内核态？运行在哪个栈上面？**

加载kernel符号表，在outw函数上打断点，观察各寄存器状态，可见此时所在的栈为0x8DFBE000，是进程的内核栈，且CS寄存器的后两位为00，表明CPU在内核态。

```
(gdb) si
=> 0x8010566d <sys_shutdown+45>:        mov    $0x604,%edx
0x8010566d in outw (data=8192, port=1540) at x86.h:30
30        asm volatile("out %0,%1" : : "a" (data), "d" (port));
(gdb) info reg
eax            0x2000   8192
ecx            0x3d5    981
edx            0x0      0
ebx            0x80112e4c       -2146357684
esp            0x8dfbef40       0x8dfbef40
ebp            0x8dfbef68       0x8dfbef68
esi            0x0      0
edi            0x8dfbefb4       -1912868940
eip            0x8010566d       0x8010566d <sys_shutdown+45>
eflags         0x282    [ SF IF ]
cs             0x8      8
ss             0x10     16
ds             0x10     16
```

6. **为何在执行完swtch函数后，cpu没有像普通函数调用一样返回到scheduler函数中？**

因为swtch函数中有换栈操作，它通过movl %esp,(%eax)指令切换了栈，从而改变了最后的ret指令返回的地址，所以没有返回到scheduler中。

通过在swtch函数中连续执行bt和si命令，观察到执行movl %esp,(%eax)指令前后gdb输出的调用栈发生了根本性变化，如下图所示。这条指令将新的上下文的栈顶覆盖到当前的esp寄存器中，从而改变了上下文。

```
(gdb) si
=> 0x801046a9 <swtch+14>:       mov    %edx,%esp
22          movl %edx, %esp
(gdb) bt
#0  swtch () at swtch.S:22
#1  0x80112784 in cpus ()
#2  0x80112780 in ?? ()
#3  0x80102e8f in mpmain () at main.c:57
#4  0x80102fcf in main () at main.c:37
(gdb) si
=> 0x801046ab <swtch+16>:       pop    %edi
swtch () at swtch.S:25
25          popl %edi
(gdb) bt
#0  swtch () at swtch.S:25
#1  0x8010ffa0 in input ()
#2  0x00000200 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
```

三、 子进程优先的 fork

1. 改动的内容

在proc.c中新增一个全局变量fork_winner_setting用于标识是否启用子进程优先，该数据存储于内核的数据区，在一个进程中的改动会影响其他进程是否启用子进程优先。

```
12  struct proc proc[NPROC];
13 } ptable;
14
15 static struct proc *initproc;
16
17 int nextpid = 1;
```

```
12  struct proc proc[NPROC];
13 } ptable;
14
15+int fork_winner_setting = 0;
16+
17 static struct proc *initproc;
18
19 int nextpid = 1;
```

如果启用了子进程优先，在 fork 函数返回之前调用 yield 函数，yield 函数可以立刻主动交出当前时间片的剩余时间，把控制权返还给 scheduler，让 scheduler 函数重新选择一个新的进程来执行。

```
217  np->state = RUNNABLE;
218
219  release(&ptable.lock);
220
221  return pid;
222 }
```

```
219  np->state = RUNNABLE;
220
221  release(&ptable.lock);
222
223+  if (fork_winner_setting)
224+    yield();
225+
226  return pid;
227 }
```

在 sys_proc.c 中实现 fork_winner 系统调用，该函数通过 argint 获取传入的参数，经过有效性检验后存入 fork_winner_setting。

```
13  return fork();
14 }
15
16 int
17 sys_exit(void)
18 {
```

```
13  return fork();
14 }
15
16+// defined in `proc.c`
17+extern int fork_winner_setting;
18+int
19+sys_fork_winner(void)
20+{
21+  int n;
22+  if(argint(0, &n) < 0)
23+    return -1;
24+  if (n != 0 && n != 1)
25+    return -1;
26+  fork_winner_setting = n;
27+  return 0;
28+}
29+
30 int
31 sys_exit(void)
32 {
```

在 syscall.c 文件中还有一些新增系统调用所需的常规改动。

```
106 extern int sys_shutdown(void);
107
130 [SYS_shutdown] sys_shutdown,
131 };
```

```
106 extern int sys_shutdown(void);
107+extern int sys_fork_winner(void);
108
131 [SYS_shutdown] sys_shutdown,
132+[SYS_fork_winner] sys_fork_winner,
133 };
```

2. **为什么这样做可以保证子进程优先？**

yield 函数通过 sched 函数间接地调用了 swtch 函数进行上下文切换，切换到 scheduler 的上下文中。值得注意的是，每次通过这种方式回到 scheduler，并不是从头开始重新执行 scheduler，而是从上一次选择的进程往后继续扫描，符合时间片轮转调度算法的性质。这也保证了，**调用 yield 函数的进程再次被调度之前必定会尝试执行其他所有 runnable 的进程。**在父进程调用 yield 函数前，子进程已经被设置成 runnable，所以一定会先于父进程调度。

3. **执行结果**

```
$ forktest
Fork test
Set child as winner
Trial 0:  child parent! !
Trial 1:  child!  parent!
Trial 2:  child!  parent!
Trial 3:  child!  parent!
Trial 4:  child!  parent!
Trial 5:  child!  parent!
Trial 6:  child!  parent!
Trial 7:  child!  parent!
Trial 8:  child!  parent!
Trial 9:  child!  parent!

Set parent as winner
Trial 0:  parent!  child!
Trial 1:  parent!  child!
Trial 2:  parent!  child!
Trial 3:  parent!  child!
Trial 4:  parent!  child!
Trial 5:  parent!  child!
Trial 6:  parent!  child!
Trial 7:  parent!  child!
Trial 8:  parent!  child!
Trial 9:  parent!  child!
$
```

4. **为什么偶尔会顺序颠倒？**

因为优先执行的进程只来得及输出部分字符，时间片就用完了，立刻被时钟中断，scheduler调度另一个进程开始执行。