

操作系统实验4

邓人嘉 21301032

一、实验步骤

1.1 时钟中断与计时器

- 实现timer子模块获取mtime值。

os /src/timer.rs:

```
@edb8f2fa7b92/mnt/os/src x + v
1 //os /src/timer.rs
2 use riscv::register::time;
3
4 pub fn get_time() -> usize {
5     time::read()
6 }
```

- 在sbi子模块实现设置mtimecmp的值，并在timer子模块进行封装。

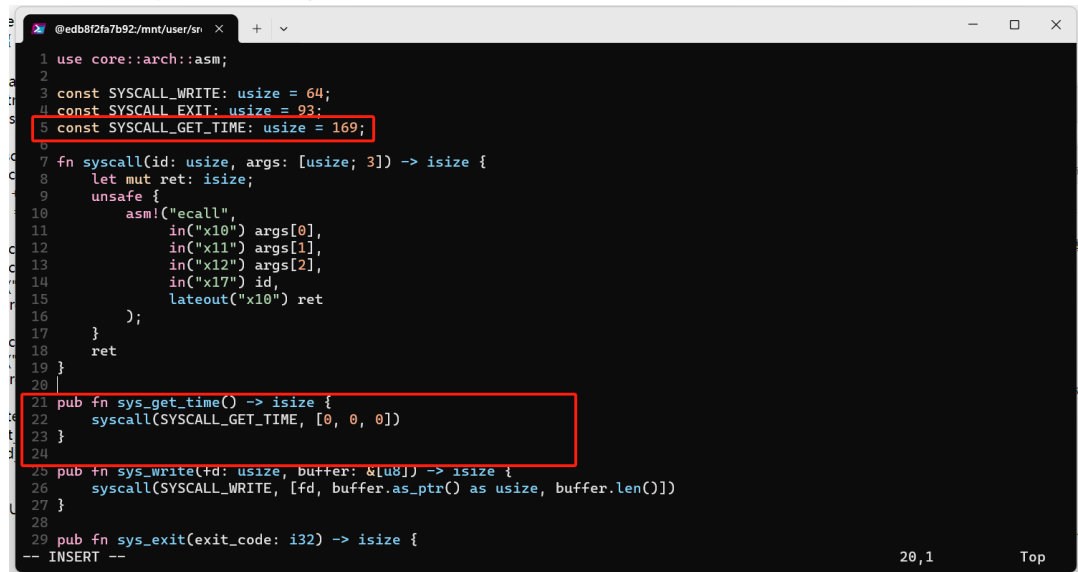
os/src/sbi.rs:

```
@edb8f2fa7b92/mnt/os/src x + v
13 const SBI_SHUTDOWN: usize = 8;
14 const SBI_SET_TIMER: usize = 0;
15 #[inline(always)]
16 fn sbi_call(which: usize, arg0: usize, arg1: usize, arg2: usize) -> usize {
17     let mut ret;
18     unsafe {
19         asm!("ecall",
20             in("x10") arg0,
21             in("x11") arg1,
22             in("x12") arg2,
23             in("x17") which,
24             lateout("x10") ret
25         );
26     }
27     ret
28 }
29
30 pub fn set_timer(timer: usize) {
31     sbi_call(SBI_SET_TIMER, timer, 0, 0);
32 }
33
34
35 pub fn console_putchar(c: usize) {
36     sbi_call(SBI_CONSOLE_PUTCHAR, c, 0, 0);
37 }
38
39 pub fn console_getchar() -> usize {
40     sbi_call(SBI_CONSOLE_GETCHAR, 0, 0, 0)
41 }
-- INSERT --
33,1 70%
```

os /src/timer.rs:

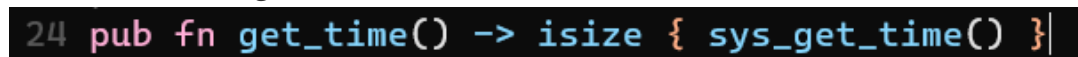
1.2 修改应用程序

- 增加get_time系统调用
 - 在user/src/syscall.rs增加get_time系统调用



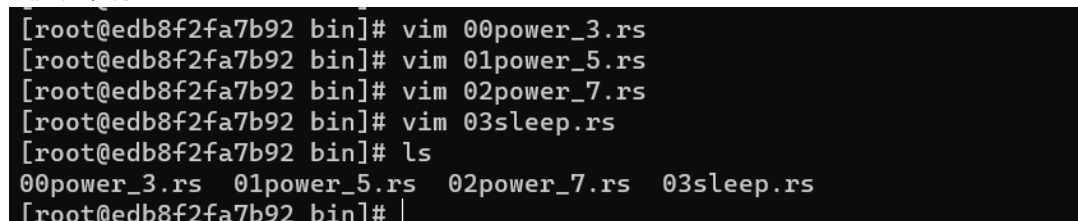
```
1 use core::arch::asm;
2
3 const SYSCALL_WRITE: usize = 64;
4 const SYSCALL_EXIT: usize = 93;
5 const SYSCALL_GET_TIME: usize = 169;
6
7 fn syscall(id: usize, args: [usize; 3]) -> isize {
8     let mut ret: isize;
9     unsafe {
10         asm!("syscall",
11             in("x10") args[0],
12             in("x11") args[1],
13             in("x12") args[2],
14             in("x17") id,
15             lateout("x10") ret
16         );
17     }
18     ret
19 }
20
21 pub fn sys_get_time() -> isize {
22     syscall(SYSCALL_GET_TIME, [0, 0, 0])
23 }
24
25 pub fn sys_write(fd: usize, buffer: &[u8]) -> isize {
26     syscall(SYSCALL_WRITE, [fd, buffer.as_ptr() as usize, buffer.len()])
27 }
28
29 pub fn sys_exit(exit_code: i32) -> isize {
30     -- INSERT --
31 }
```

- 在user/src/lib.rs增加get_time用户库的封装



```
24 pub fn get_time() -> isize { sys_get_time() }
```

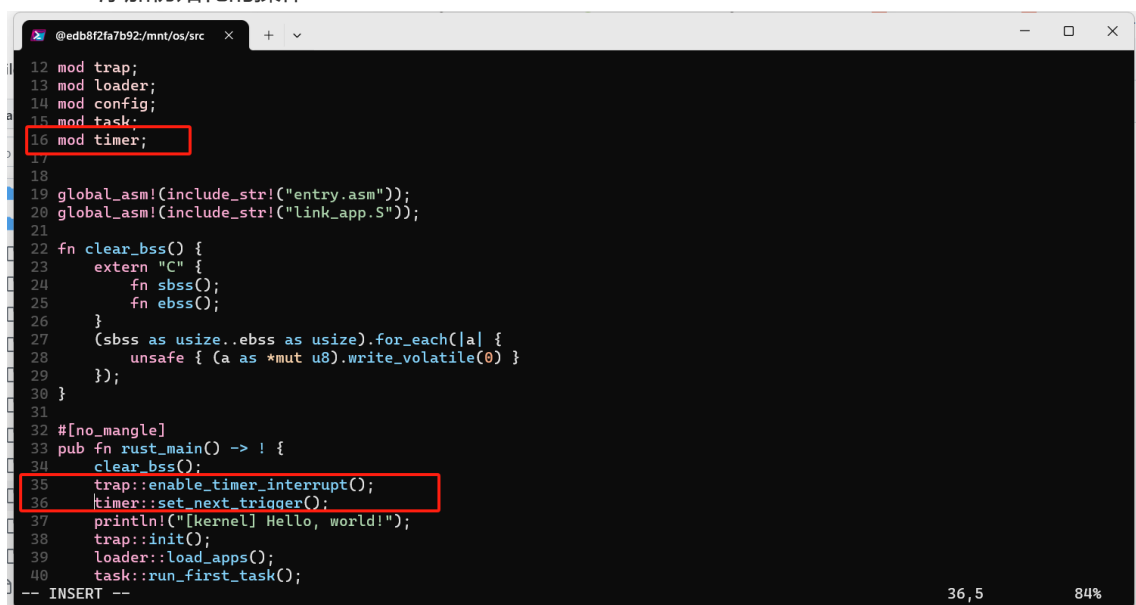
- 实现新的测试应用
 - 按照实验手册分别实现00power_3.rs, 01power_5.rs, 02power_7.rs以及03sleep.rs四个测试应用程序。



```
[root@edb8f2fa7b92 bin]# vim 00power_3.rs
[root@edb8f2fa7b92 bin]# vim 01power_5.rs
[root@edb8f2fa7b92 bin]# vim 02power_7.rs
[root@edb8f2fa7b92 bin]# vim 03sleep.rs
[root@edb8f2fa7b92 bin]# ls
00power_3.rs 01power_5.rs 02power_7.rs 03sleep.rs
[root@edb8f2fa7b92 bin]#
```

1.3 抢占式调度

- 在os/src/trap/mod.rs中实现抢占式调度。
按照手册修改代码即可，不具体截图了。
- main.rs添加初始化的操作



```
12 mod trap;
13 mod loader;
14 mod config;
15 mod task;
16 mod timer;
17
18
19 global_asm!(include_str!("entry.asm"));
20 global_asm!(include_str!("link_app.S"));
21
22 fn clear_bss() {
23     extern "C" {
24         fn sbss();
25         fn ebss();
26     }
27     (sbss as usize..ebss as usize).for_each(|a| {
28         unsafe { (a as *mut u8).write_volatile(0) }
29     });
30 }
31
32 #[no_mangle]
33 pub fn rust_main() -> ! {
34     clear_bss();
35     trap::enable_timer_interrupt();
36     timer::set_next_trigger();
37     println!("[kernel] Hello, world!");
38     trap::init();
39     loader::load_apps();
40     task::run_first_task();
41     -- INSERT --
42 }
```

1.4 运行结果

```
@edb8f2fa7b92/mnt/os x + v
[1] [root@edb8f2fa7b92 mnt]# cd os
[2] [root@edb8f2fa7b92 os]# make build
[3]     Compiling os v0.1.0 (/mnt/os)
[4]     Finished release [optimized] target(s) in 1.80s
[5] [root@edb8f2fa7b92 os]# make run
[6]     Compiling os v0.1.0 (/mnt/os)
[7]     Finished release [optimized] target(s) in 1.80s
[8] [rustsbi] RustSBI version 0.2.0-alpha.6
[9]
[10] [rustsbi] Implementation: RustSBI-QEMU Version 0.0.2
[11] [rustsbi-dtb] Hart count: cluster0 with 1 cores
[12] [rustsbi] misa: RV64ACDFIMSU
[13] [rustsbi] mideleg: ssoft, stimer, sext (0x222)
[14] [rustsbi] medeleg: ima, ia, bkpt, la, sa, uecall, ipage, lpage, spage (0xb1ab)
[15] [rustsbi] pmp0: 0x100000000 ..= 0x10001fff (rwx)
[16] [rustsbi] pmp1: 0x800000000 ..= 0x8fffffff (rwx)
[17] [rustsbi] pmp2: 0x0 ..= 0xfffffffffffff (---)
[18] qemu-system-riscv64: clint: invalid write: 00000004
[19] [rustsbi] enter supervisor 0x80200000
[20] [kernel] Hello, world!
[21] power_3 [10000/200000]
[22] power_3 [20000/200000]
[23] power_3 [30000/200000]
[24] power_3 [40000/200000]
[25]
[26] [rustsbi] Implementation: RustSBI-QEMU Version 0.0.2
[27] [rustsbi-dtb] Hart count: cluster0 with 1 cores
[28] [rustsbi] misa: RV64ACDFIMSU
[29] [rustsbi] mideleg: ssoft, stimer, sext (0x222)
[30] [rustsbi] medeleg: ima, ia, bkpt, la, sa, uecall, ipage, lpage, spage (0xb1ab)
[31] [rustsbi] pmp0: 0x100000000 ..= 0x10001fff (rwx)
[32] [rustsbi] pmp1: 0x800000000 ..= 0x8fffffff (rwx)
[33] [rustsbi] pmp2: 0x0 ..= 0xfffffffffffff (---)
[34] qemu-system-riscv64: clint: invalid write: 00000004
[35] [rustsbi] enter supervisor 0x80200000
[36] [kernel] Hello, world!
[37] power_3 [10000/200000]
[38] power_3 [20000/200000]
[39] power_3 [30000/200000]
[40] power_3 [40000/200000]
[41] power_3 [50000/200000]
[42] power_3 [60000/200000]
[43] power_3 [70000/200000]
[44] power_3 [power_5 [10000/200000]
[45] power_5 [20000/200000]
[46] power_5 [30000/200000]
[47] power_5 [40000/200000]
[48] power_5 [50000/200000]
[49] power_5 [60000/200000]
[50] power_5 [70000/200000]
[51] power_5 [80000/200000]
[52] power_5 [90000/200000]
[53] power_5 [100000/200000]
[54] power_5 [110000/200000]
[55] power_5 [120000/200000]
[56]
[57] power_5 [130000/200000]
[58] power_5 [140000/200000]
[59] power_5 [150000/200000]
[60] power_5 [160000/200000]
[61] power_5 [170000/200000]
[62] power_5 [180000/200000]
[63] power_7 [10000/200000]
[64] power_7 [20000/200000]
[65] power_7 [30000/200000]
[66] power_7 [40000/200000]
[67] power_7 [50000/200000]
[68] power_7 [60000/200000]
[69] power_7 [70000/200000]
[70] power_7 [80000/200000]
[71] power_7 [90000/200000]
[72] power_7 [100000/200000]
[73] power_7 [110000/200000]
[74] power_7 [120000/200000]
[75] power_7 [130000/200000]
[76] power_7 [140000/200000]
[77] power_7 [150000/200000]
[78] power_7 [160000/80000/200000]
[79] power_3 [90000/200000]
[80] power_3 [100000/200000]
[81] power_3 [110000/200000]
[82] power_3 [120000/200000]
[83] power_3 [130000/200000]
[84] power_3 [140000/200000]
[85] power_3 [150000/200000]
[86] power_3 [160000/200000]
```

```
@edb8f2fa7b92/mnt/os x + v
power_3 [110000/200000]
power_3 [120000/200000]
power_3 [130000/200000]
power_3 [140000/200000]
power_3 [150000/200000]
power_3 [160000/200000]
power_3 [170000/200000]
power_3 [180000/200000]
power_3 [190000/200000]
power_3 [200000/200000]
3*200000 = 871008973
Test power_3 OK!
[kernel] Application exited with code 0
power_5 [200000]
power_7 [170000/200000]
power_7 [180000/200000]
power_7 [190000/200000]
power_7 [200000/200000]
7*200000 = 277895943
Test power_7 OK!
[kernel] Application exited with code 0
190000/200000
power_5 [200000/200000]
5*200000 = 670295496
Test power_5 OK!
[kernel] Application exited with code 0
Test sleep OK!
[kernel] Application exited with code 0
Panic! at src/task/mod.rs:98 All applications completed!
[root@edb8f2fa7b92 os]#
```

二、思考问题

2.1 分析分时多任务是如何实现的。

- 在timer.rs中实现mtime的获取，在sbi子模块设置mtimecmp的值，当mtime超过mtimecmp时需要触发一次时钟中断。
- 在trap子模块中使用set_stimer操作系统能够定时中断应用程序。在中断后调用suspend_current_and_run_next()来中断当前程序，并把资源分配给下一个应用程序。

2.2 分析抢占式调度是如何设计和实现的。

- 抢占式调度是基于时钟中断的，中断会周期性发起，当前正在运行的应用程序会被迫让出CPU权限。
- 中断发生时，操作系统会强制切换到下一个任务的上下文，具体是在trap子模块的suspend_current_and_run_next中执行的。

2.3 对比上个实验实现的协作式调度与本实验实现的抢占式调度。

- 协作式调度程序需要通过yield来主动让出cpu权限，而抢占式调度可以通过yield主动让出cpu权限，也可以被迫等待时钟中断，被强制让出cpu权限。
- 协作式调度可能存在一个程序一直堵塞后面的程序，而抢占式调度中即使当前程序是死循环也会因为时间到了而被迫让出cpu。

三、git截图

- git截图(<https://github.com/lovekdl/GardenerOS>)

