|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
| 北京交通大学软件学院  **《操作系统》课程**  **实验报告** | | |

|  |
| --- |
| **姓名: 张鑫成** |
| **学号: 20271055** |

目录

[1 实验目的 2](#_Toc11355)

[2 实验过程设计 3](#_Toc19994)

[3 源代码及注释 3](#_Toc30679)

[4 运行结果与分析 5](#_Toc17763)

[5 实验总结 6](#_Toc21866)

# 1 实验目的

（说明：写出本次实验的主要实验目的与原因。）

实现操作系统对中断的处理和对应用程序的抢占，设计实现更加公平和高效交互的抢占式操作系统。

# 2 实验过程设计

（说明：写出本次实验的主要实验流程或个人实验的实施过程。）

1. 实现时间中断与计时器。
2. 实现抢占式调度
3. 使用时间片轮转算法(RR, Round-Robin) 来对应用进行调度。

# 3 源代码及注释

（说明：写出本次实验使用的命令或编写的源代码。请在下列表格中说明源代码的文件名和代码功能概述或者命令名和该命令的主要作用。要求针对核心功能代码进行注释。）

表 3-1 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/timer.rs |
| **主要功能:** | 取得当前 mtime 计数器的值。  统计一个应用的运行时长 |
| **源代码:**  //! RISC-V timer-related functionality  use crate::config::CLOCK\_FREQ;  use crate::sbi::set\_timer;  use riscv::register::time;  const TICKS\_PER\_SEC: usize = 100;  const MSEC\_PER\_SEC: usize = 1000;  /// read the `mtime` register  pub fn get\_time() -> usize {  time::read()  }  /// get current time in milliseconds  pub fn get\_time\_ms() -> usize {  time::read() / (CLOCK\_FREQ / MSEC\_PER\_SEC)  }  /// set the next timer interrupt  pub fn set\_next\_trigger() {  set\_timer(get\_time() + CLOCK\_FREQ / TICKS\_PER\_SEC);  } | |

表 3-2 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/sbi.rs |
| **主要功能:** | 调用接口间接实现计时器的控制 |
| **源代码:**  //! SBI call wrappers  use core::arch::asm;  const SBI\_SET\_TIMER: usize = 0;  const SBI\_CONSOLE\_PUTCHAR: usize = 1;  #[cfg(feature = "board\_k210")]  const SBI\_SHUTDOWN: usize = 8;  // const SBI\_CONSOLE\_GETCHAR: usize = 2;  // const SBI\_CLEAR\_IPI: usize = 3;  // const SBI\_SEND\_IPI: usize = 4;  // const SBI\_REMOTE\_FENCE\_I: usize = 5;  // const SBI\_REMOTE\_SFENCE\_VMA: usize = 6;  // const SBI\_REMOTE\_SFENCE\_VMA\_ASID: usize = 7;  /// handle SBI call with `which` SBI\_id and other arguments  #[inline(always)]  fn sbi\_call(which: usize, arg0: usize, arg1: usize, arg2: usize) -> usize {  let mut ret;  unsafe {  asm!(  "li x16, 0",  "ecall",  inlateout("x10") arg0 => ret,  in("x11") arg1,  in("x12") arg2,  in("x17") which,  ); | |

表 3-3 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/trap/mod.rs |
| **主要功能:** | 新增一个条件分支跳转，暂停当前应用并切换到下一个 |
| **源代码:**  match scause.cause() {  Trap::Interrupt(Interrupt::SupervisorTimer) => {  set\_next\_trigger();  suspend\_current\_and\_run\_next();  }}  use riscv::register::sie;  pub fn enable\_timer\_interrupt() {  unsafe { sie::set\_stimer(); }} | |

表 3-4 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/main.rs |
| **主要功能:** | 初始化设置 |
| **源代码:**  #[no\_mangle]pub fn rust\_main() -> ! {  clear\_bss();  println!("[kernel] Hello, world!");  trap::init();  loader::load\_apps();  trap::enable\_timer\_interrupt();  timer::set\_next\_trigger();  task::run\_first\_task();  panic!("Unreachable in rust\_main!");} | |

表 3-5 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | user/src/bin/03sleep.rs |
| **主要功能:** | 等待 3000ms 然后退出 |
| **源代码:**  #[no\_mangle]fn main() -> i32 {  let current\_timer = get\_time();  let wait\_for = current\_timer + 3000;  while get\_time() < wait\_for {  yield\_();  }  println!("Test sleep OK!");  0} | |

# 4 运行结果与分析

（说明：此部分请写出代码或命令运行结果及其分析。运行结果可以通过截图的方式提交,此外需要对截图进行分析与说明。）

代码成功使用时间片轮转算法运行，体现了抢占式调度。

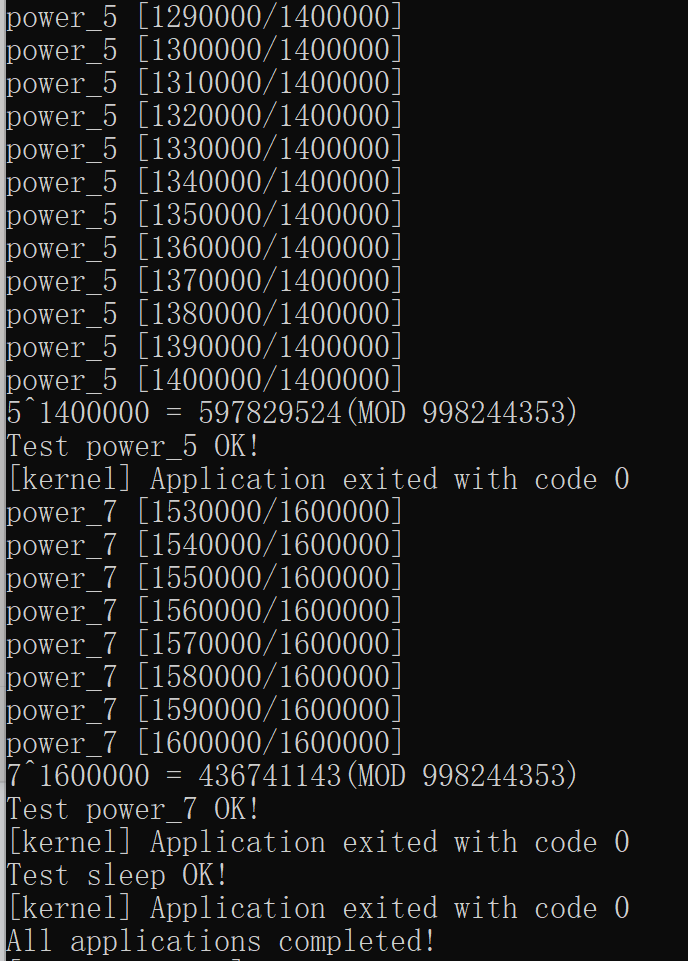


图 4.1运行结果截图

# 5 实验总结

（说明：此部分请写出针对本次实验的总结。也可以写出对于相同功能的不同实现方案，或是书写针对现有实验的改进方案等独创的且与本次实验有关的内容。）

初步学习了RISC-V 架构中的中断

了解了中断规则：

1. 如果中断的特权级低于 CPU 当前的特权级，则该中断会被屏蔽，不会被处理；
2. 如果中断的特权级高于与 CPU 当前的特权级或相同，则需要通过相应的 CSR 判断该中断是否会被屏蔽。

并且与之前学习的S、M特权级的相关内容相照应，大致了解了中断与特权级的关系

1. U 特权级的应用程序发出系统调用或产生错误异常都会跳转到 S 特权级的操作系统内核来处理；
2. S 特权级的时钟/软件/外部中断产生后，都会跳转到 S 特权级的操作系统内核来处理。

调用相关接口间接地实现了计时器的控制，巧妙化解了特权级不够的问题。

拥有时钟中断和计时器，之后设置分支跳转，当发现触发了一个 S 特权级时钟中断的时候，首先重新设置一个10ms的计时器，然后调用suspend\_current\_and\_run\_next函数暂停当前应用并切换到下一个。成功实现抢占式调度。