Project 2 Simple Kernel (Part 2) 设计文档

马月骁

2019K8009915025

一、时钟中断与系统调用

1. 例外处理流程

例外分为中断与异常，包含主动触发与被动触发两种情形。不论例外触发的原因或具体类型，其处理流程基本一致（在本次实验中仅涉及时钟中断与系统调用）：

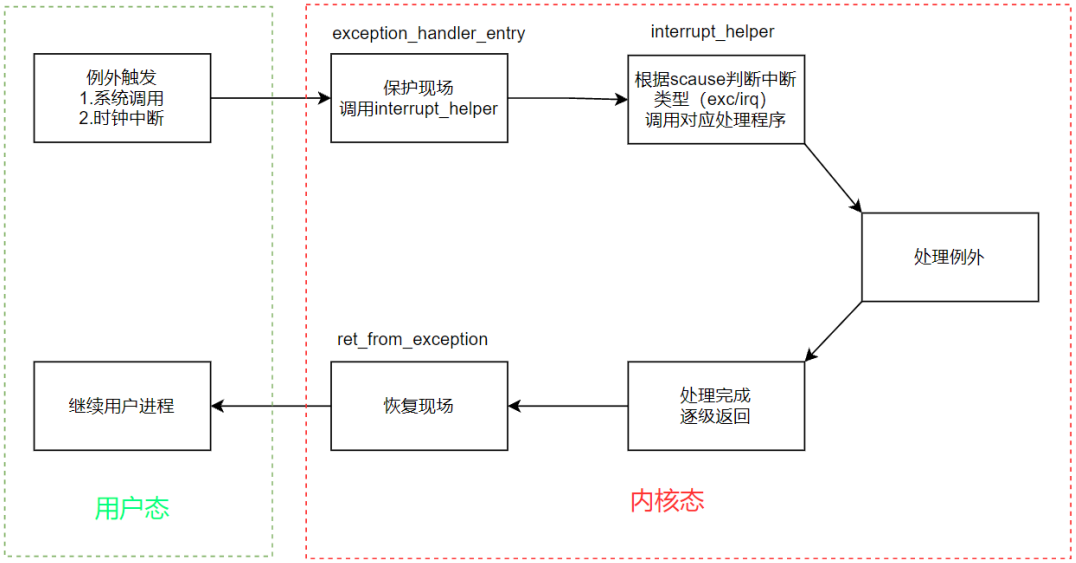

图 1 例外处理流程

（1）首先，当例外触发后，处理器自动跳转至 stvec 中所存的地址，即 exception_handler_entry 的地址。同时，sepc 中记录当前 pc 的值。

（2）exception_handler_entry：保护现场（将通用寄存器与 sepc、scause、sstatus、sbadaddr 存入栈），然后调用 interrupt_helper 进行中断类型的判断和分类处理。

（3）interrupt_helper：根据 scause 寄存器标记的例外原因，去不同的例外处理函数表（irq_table 或 exc_table）中调用相应的处理函数。

其中 scause 寄存器第 63 位（最高位）记录了该例外类型，62 至 0 位记录中断类型编号。

（4）ret_from_exception：恢复现场，通过 sret 指令返回用户态。

值得注意的是，在运行用户态进程之前需要通过 ret_from_exception 进入用户态。

2. 系统调用

（1）系统调用中用户态与内核态的通信

a）系统调用用户态函数接口

用户态进程通过调用系统调用接口函数，触发系统调用：

```
void sys_sleep(uint32_t time)
{
    // TODO:
    invoke_syscall(SYSCALL_SLEEP, time, IGNORE, IGNORE);
}

void sys_write(char *buff)
{
    // TODO:
    invoke_syscall(SYSCALL_WRITE, (uintptr_t)buff, IGNORE, IGNORE);
}

void sys_reflush()
{
    // TODO:
    invoke_syscall(SYSCALL_REFLUSH, IGNORE, IGNORE, IGNORE);
}
```

图 2 部分接口函数

b）传参，陷入内核态

通过 invoke_syscall 函数，完成参数传递，通过 ecall 陷入内核态。值得注意的是，根据 risc-v ABI 约定，系统调用号应当存于 a7 寄存器

```
ENTRY(invoke_syscall)
  /* TODO: */
  mv a7, a0
  mv a0, a1
  mv a1, a2
  mv a2, a3
  ecall
  jr ra
ENDPROC(invoke_syscall)
```

图 3 invoke_syscall

c）返回值的传递

handle_syscall 函数从例外栈帧中获得寄存器传来的参数，提供给内核态的处理函数，并且将返回值写入例外栈帧的 a0 寄存器中。这样在恢复现场时，返回值自然就被恢复到 a0 通用寄存器中。

（2）系统调用 sleep

sleep 系统调用流程大致如下：

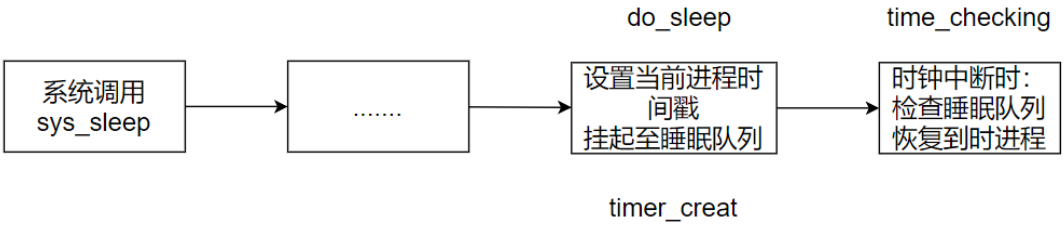

图 4 sleep 系统调用流程

具体函数内容可详参代码。

（3）系统调用 fork

a）系统调用用户态函数接口

由于未实现虚拟内存，故 fork 需要通过汇编函数实现接口函数：

```
ENTRY(sys_fork)
  li a0, 0
  li a1, 0
  li a2, 0
  li a7, 35 //syscall number of SYS_FORK
  ecall
  jr ra
ENDPROC(sys_fork)
```

图 5 fork 函数接口

b）子进程数据设置

子进程大部分数据（包含 PCB 信息以及栈中数据）均应当与父进程保持一致。

特别的，pid、状态、栈指针应当与父进程不同，同时内核栈中 a0，sp，fp，tp 寄存器也应更新：

```
np->pid = process_id;
np->status = TASK_READY;
np->kernel_sp = allocPage(1) + PAGE_SIZE;
np->user_sp = allocPage(1) + PAGE_SIZE;
```

```
pt_regs->regs[10] = 0;                  //a0 chlid process return 0

pt_regs->regs[2] += sp_distance;        //sp
pt_regs->regs[8] += sp_distance;        //s0
pt_regs->regs[4] = (reg_t)np;           //tp
```

图 6 子进程新信息

3. 时钟中断

时钟中断到来时，需要完成：

（1）刷新串口

（2）检查睡眠进程是否到点

（3）设置下次时钟中断到来时间

（4）切换进程

```
void reset_irq_timer()
{
    // TODO clock interrupt handler.
    // TODO: call following functions when task4
    screen_reflush();
    time_checking();
    // note: use sbi_set_timer
    sbi_set_timer(get_ticks() + TIMER_INTERVAL);
    //sbi_set_timer(0);
    // remember to reschedule
    do_scheduler();
}
```

图 7 时钟中断处理

二、抢占式调度与优先级调度

1. 抢占式调度

抢占式调度基于时钟中断实现，在每次时钟中断时进行调度（见时钟中断）。需要注意的时，在 main 函数中，应当将设置第一次时钟中断时间；并且 tset 中不再需要 sys_yield 实现进程切换。

2. 优先级调度

本实验中实现的调度策略是基于彩票算法的优先级调度策略。在调度时，综合考虑其优先级和加入就绪队列后的等待时间，根据二者加权计算 rank，选择 rank 最高进程的调度。

$$rank(x) = x[priority] * weightpriority + x[waitime] * weightwaitime$$

这种调度策略可以满足:

（1）优先级的有效性

（2）各个进程间的公平性

```c
uint32_t rank(pcb_t *pcb){
    uint32_t ticket = pcb->priority * 5 + ((get_ticks() - pcb->time_label) / TIMER_INTERVAL);
    return ticket;
}

pcb_t *prior_search_next(list_head *queue){
    pcb_t *next, *tmp;
    list_node_t *node = queue->next;
    next = list_entry(node, pcb_t, list);
    uint32_t ticket;
    uint32_t max_ticket = 0;
    while(node != queue){
        tmp = list_entry(node, pcb_t, list);
        if((ticket = rank(tmp)) > max_ticket){
            max_ticket = ticket;
            next = tmp;
        }
        node = node->next;
    }
    return next;
}
```

图 8 优先级调度评分与搜索函数

三、debug 时发现的问题

本次实验中出现的问题大多与进程切换时的栈指针出现错误有关，具体有：

1. 地址错误：sp（kernel）指针经过函数调用发生改变，导致进程切换时若使用当前 sp 作为基址会保存/恢复错误数据。
2. 数据覆盖：例外处理时将当前 sp（user）暂存至 PCB 中 kernel->user_sp 中，覆盖用户栈栈顶指针，导致 fork 时复制错误的父进程栈信息。

以上错误实际上均是由于初步设计时考虑不全面导致的（前置实验中未出错），在进行设计时留下记录，可以辅助后续功能添加后测试定位 bug。

对于栈指针的维护应当更加小心。