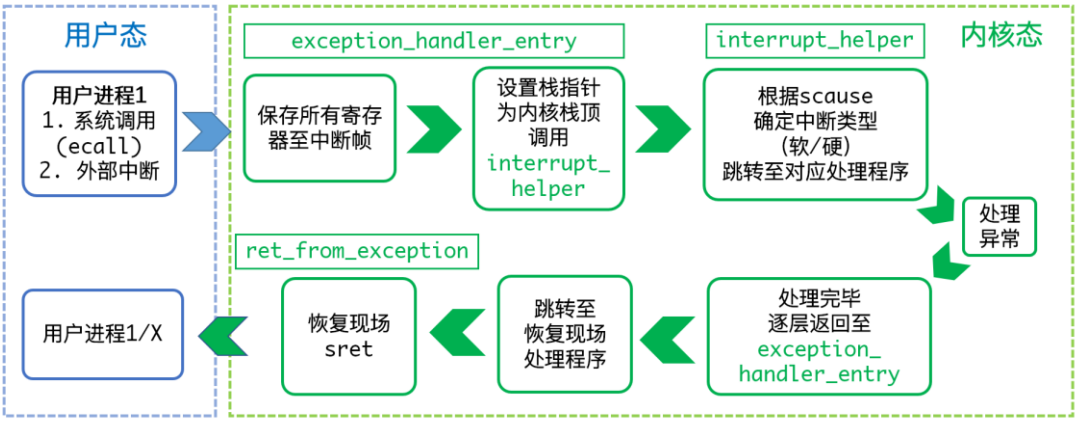
Project2 A Simple Kernel 设计文档（Part II，C-core）

中国科学院大学
王嵩岳
2020 年 10 月 30 日

1. 时钟中断、系统调用与 blocking sleep 设计流程

（1） 中断的一般处理流程

中断一般分为内部中断（异常）和外部中断。它们触发的方式不同，但大致的处理流程类似。具体流程如下图所示：



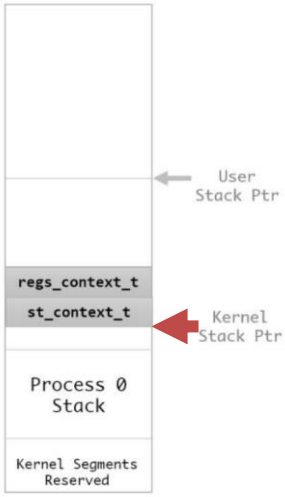
图表 1 中断处理的一般流程

a. 首先，当中断触发后，CPU 会自动跳转至 stvec（CSR 寄存器，中断入口向量地址）所存的地址，即 exception_handler_entry 的地址。

b. Exception_handler_entry 函数的主要作用是：保存上下文（所有通用寄存器，以及 sstatus, scause, sepc, sbadaddr 四个特权级寄存器[1]）至中断帧，然后进一步调用 interrupt_helper 进行中断类型的判断和分类处理。

这里需要特别注意：由于 interrupt_helper 并不是叶子函数，且运行在内核态，所以保存现场之后，一定要把栈指针指向内核栈栈顶，让接下来的内核函数使用内核栈，否则将会覆盖上下文数据。

[1] 保存特权级寄存器的本意是可以支持嵌套中断，但是本实验并不涉及。



图表 2 保存现场后，栈指针应指向内核栈栈顶

c. Interrupt_helper 的主要功能是：根据 scause 寄存器标记的中断原因，去不同的中断处理函数表中调用相应的处理函数。

Interrupt_helper 需要参数：中断帧的地址（也就是 sp），stval 和 scause。这里我们根据 RV64 ABI 约定，在汇编中传参并调用之。

```
ld a0, PCB_KERNEL_SP(tp)
addi a0, a0, SWITCH_TO_SIZE
csrr a1, CSR_STVAL
csrr a2, CSR_SCAUSE
call interrupt_helper
```

图表 3 Exception_handler_entry 的传参

```
void interrupt_helper(regs_context_t *regs, uint64_t stval, uint64_t cause)
{
    // TODO interrupt handler.
    // call corresponding handler by the value of `cause`
    handler_t *table = (cause >> 63) ? irq_table : exc_table;
    uint64_t exc_code = cause & ~SCAUSE_IRQ_FLAG;
    table[exc_code](regs, stval, cause);
}
```

图表 4 Interrupt_helper 的处理

首先，可根据 scause 寄存器第 63 位（最高位）的值，来判断这个中断是硬中断还是软中断[2]，并根据 62:0 位获得中断类型号。根据中断类型和具体编号，我们可以调用起 irq_table 或 exc_table 中的对应的具体处理函数。

对本次实验而言，硬中断我们只有时钟中断（调用 handle_int 进而调用 reset_timer_irq），软中断只有系统调用（调用 handle_syscall）。

d. 接下来就是对应的内核态函数处理。处理完毕后，根据函数调用关系逐级返回到 interrupt_helper 的最后，然后返回 exception_helper_entry 中 call interrupt_helper 的下一条指令，即进行中断返回。

[2] 参见 RV64 特权级 ISA 手册或实验指导书。

```
    ld a0, PCB_KERNEL_SP(tp)
    addi a0, a0, SWITCH_TO_SIZE
    csrr a1, CSR_STVAL
    csrr a2, CSR_SCAUSE
    call interrupt_helper
    la ra, ret_from_exception
    jr ra
ENDPROC(exception_handler_entry)
```

图表 5 Exception_helper_entry 的后半部分

e. 中断返回程序 ret_from_exception 负责恢复现场，并且 sret 回到内核态。

（2） 时钟中断

时钟中断到来时，我们需要完成以下四件事：

- **刷新串口**：由于之前 screen_write 的数据还在串口缓冲区，我们需要定时刷一下串口以保证串口的正常实时输出。我们让每次时钟中断到来时刷串口。
- **检查睡眠进程是否到点**：我们需要定时查看在 sleep_queue 中的进程是否达到了睡眠时间。我们让每次时钟中断到来时检查。
- **设置下次时钟中断到来时间**：这里通过特权级操作接口 SBI 的函数设置定时。实际上是在设置 M 态的 mtimecmp 寄存器。当 mtimecmp 寄存器的值与 mtime 的值相等时，触发时钟中断。
- **切换进程**：由于我们的任务调度是基于时间片法的，所以当时钟中断到来时，我们进行一次进程的调度。

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

图表 6 时钟中断内核处理函数

（3） 系统调用

系统调用与外部中断一样，都属于中断事件，CPU 都要进行相应。在进行系统调用后，ecall 指令会让处理器陷入内核态，进行中断处理，处理完毕后中断返回，整个流程与外部中断基本一致。系统调用相对于外部中断的主要不同点在于：

- 触发方式不同。
- 从微结构来说：系统调用属于内部中断，是同步事件；外部中断是异步事件。
- 从过程来说：**系统调用需要参数和返回值的传递！涉及到用户态和内核态的通信。**

对于操作系统课程，我们重点关注**用户态和内核态的通信**。

a. **为用户进程提供 系统调用的用户态函数接口**

首先用户态的进程不可以直接调用内核态的函数，我们也不能让用户进程 C 程序“丑陋地”都通过内嵌汇编指令“ecall”的形式调用系统调用，因此我们需要向用户进程提供**系统调用的用户态函数接口**。

用户程序只需调用这些函数，即可触发系统调用。

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

void sys_scheduler()
{
    invoke_syscall(SYSCALL_SCHED, IGNORE, IGNORE, IGNORE);
}

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

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

图表 7 一些用户态函数接口

b. **触发系统调用、参数传递**

上面提到，系统调用可能需要参数。如 sleep 需要设置睡眠时间。那么用户接口 sys_sleep(uint32_t time)的参数应该如何传递呢？这里我们将触发系统调用的汇编指令打包，写成方法 invoke_syscall：

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

图表 8 触发系统调用和参数传递

它的作用就是将用户态函数传的参数，根据系统调用规则重新摆放至对应的寄存器，然后触发系统调用。

c. **内核中参数和返回值的传递**

我们已经把系统调用的参数通过寄存器传给内核。触发系统调用后，首先保存现场，也就意味着参数值已经被保存至中断帧中。

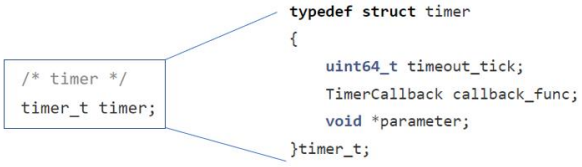
接下来，根据 scause 的值，会到达 handle_syscall 函数。它从中断帧中获得寄存器传来的参数，提供给内核态的处理函数，并且将返回值写入中断帧的 a0 寄存器中。这样在恢复现场时，返回值自然就被恢复到 a0 寄存器中。这样就完成了内核态向用户态的参数传递。

```
void handle_syscall(regs_context_t *regs, uint64_t interrupt, uint64_t cause)
{
    // syscall[fn](arg1, arg2, arg3)
    regs->sepc = regs->sepc + 4;
    regs->regs[10] = syscall[regs->regs[17]](regs->regs[10],
                                             regs->regs[11],
                                             regs->regs[12]);
}
```

图表 9 系统调用处理函数的参数传递和返回值传递

（4） 睡眠任务的阻塞与唤醒

睡眠进程需要记录时间（我的实现是记录它**到点的时间**，即 timeout_tick），睡眠到点后应调用的函数和这个函数的参数。因此我对 PCB 结构体中嵌入了一个 timer_t 类型的计时器。



图表 10 PCB 内嵌计时器

当进程调用 sys_sleep 触发睡眠系统调用后，进入内核态会调用 do_sleep 方法：首先将当前进程状态设置为阻塞态，然后初始化其 PCB 中内嵌的计时器并将其加入睡眠队列 sleep_queue，最后重新调度。

```
void do_sleep(uint32_t sleep_time)
{
    // TODO: sleep(seconds)
    // note: you can assume: 1 second = `timebase` ticks
    // 1. block the current_running
    current_running->status = TASK_BLOCKED;
    // 2. create a timer which calls `do_unblock` when timeout
    timer_create(&do_unblock, &current_running->list, sleep_time * time_base);
    // 3. reschedule because the current_running is blocked.
    do_scheduler();
}

void timer_create(TimerCallback func, void* parameter, uint64_t tick)
{
    disable_preempt();
    current_running->timer.timeout_tick = get_ticks() + tick;
    current_running->timer.callback_func = func;
    current_running->timer.parameter = parameter;
    enqueue_timer(&sleep_queue, current_running);
    enable_preempt();
}

void enqueue_timer(list_head* queue, pcb_t* item)
{
     list_add_tail(&item->list, queue);
}
```

图表 11 睡眠进程的初始化

每次时钟中断到来时，都会调用 timer_check 方法检查各任务是否到点。具体的方法是：遍历整个 sleep_queue，对每个 PCB 的计时器检查，若到点，则调用 callback（也就是 unblock 方法）重新将其加入就绪队列 ready_queue。

```
pcb_t *gethead_timer(list_head* queue)
{
    timer_t *tmp = list_entry(queue, pcb_t, list);
    return tmp;
}

void timer_check()
{
    disable_preempt();
    pcb_t *tmp;
    list_node_t *ptr = sleep_queue.next;
    while(!list_empty(&sleep_queue) && (ptr != &sleep_queue)){
        tmp = gethead_timer(ptr);
        if(get_ticks() >= tmp->timer.timeout_tick){
            tmp->timer.callback_func(tmp->timer.parameter);
          }
        ptr = ptr->next;
     }
    enable_preempt();
}
```

图表 12 遍历睡眠队列，检查 PCB 计时器是否到点

2. 基于优先级的调度器设计

（1）算法描述

我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，我们不再每次选取就绪队列的头结点 PCB 对应的进程/线程。而是综合考虑其优先级和加入就绪队列后的等待时间，根据二者加权计算分数，选择分数最高进程的调度。

$$score(x) = x[priority] * weight_{priority} + x[waitime] * weight_{waitime}$$

这样处理的好处有以下两点

- **保证优先级的有效性**：优先级作为参数之一，在等待时间相差不大时，优先级越高，越优先被调度。
- **保证优先级低的进程不被饿死**：当一个进程等待时间过长时，分数也会很高，被优先调度。

实际实验中，我采用经验得出最终的评分函数，稍后验证合理性：

```
uint16_t score(priority_t p, uint64_t time){
    return p * 5 + ((get_ticks() - time) / TIMER_INTERVAL);
}
```

图表 13 进程调度评分函数

（2） 实现细节

要实现上述策略，我们需要对 PCB 的数据结构进行修改。我们需要添加两项：**优先级**和**加入就绪队列的时间戳**。

```
    /* priority */
    priority_t priority;

    /* add tick */
    uint64_t add_tick;

} pcb_t;
```

图表 14 PCB 新增项

- **优先级**：我设置 priority_t 为枚举类型，包含四个优先级，其中，P4 最高，P1 最低。（当然，优先级个数可以动态可调）

```
typedef enum {P_1, P_2, P_3, P_4} priority_t;
```

图表 15 优先级定义

优先级的来源实际来自于 task。我们也需要对 task 结构体添加优先级项。

```
/* task information, used to init PCB */
typedef struct task_info
{
    ptr_t entry_point;
    task_type_t type;
    priority_t priority;
} task_info_t;
```

图表 16 更新 task 结构体

下面是我们的测试用例组，

```
struct task_info task2_14 = {(ptr_t)&priority_task1, USER_PROCESS, P_1};
struct task_info task2_15 = {(ptr_t)&priority_task2, USER_PROCESS, P_2};
struct task_info task2_16 = {(ptr_t)&priority_task3, USER_PROCESS, P_3};
struct task_info task2_17 = {(ptr_t)&priority_task4, USER_PROCESS, P_4};
struct task_info task2_18 = {(ptr_t)&do_sch_timer, USER_PROCESS, P_1};
struct task_info *priority_tasks[16] = {&task2_14, &task2_15, &task2_16, &task2_17, &task2_18};
int num_priority_tasks = 5;
```

图表 17 优先级测试用例组

可以看出，前四个任务对应被赋予不同的优先级。（第 5 个测试是在测上下文切换时间）。

- 加入等待队列的时间戳：在 do_scheduler()中记录。

```
64          if(current_running->status == TASK_RUNNING && current_running->pid != 0){
65              enqueue(&ready_queue, current_running);
66              current_running->status = TASK_READY;
67          }
68          next_running->status = TASK_RUNNING;
69          current_running->add_tick = get_ticks();
```

图表 18 记录加入就绪队列的时间

（3）验证合理性

上面提到，我们用四个优先级不同的进程进行测试。我们如下设计进程函数。

```c
void priority_task3(void)
{
    int i;
    int print_location = 4;

    for (i = 0;; i++)
    {
        sys_move_cursor(1, print_location);
        printf("> [TASK3] This task is to test priority P_3. (%d)", i);
    }
}
```

图表 19 优先级测试进程函数

优先级越高的进程，被分配执行的 CPU 时间应该越长，循环次数应该越多，i 值应该越大。因此四个进程打印的 i 值应该与优先级成正比。我们可以观察实际效果。

```
> [TASK1] This task is to test priority P_1. (382654)
> [TASK2] This task is to test priority P_2. (678795)
> [TASK3] This task is to test priority P_3. (1419459)
> [TASK4] This task is to test priority P_4. (1674037)
```

图表 20 优先级测试效果

实际观察，可以看出，任意时刻，每个进程的循环执行次数与优先级正相关。另外，低优先级的进程的 i 值也在持续变化，说明低优先级的进程并没有被饿死，符合要求。

3. Context-switch 开销测量的设计思路

上下文切换的开销主要在 do_scheduler()。这里我们采用内核和用户进程协同测量的方法。

首先申请一个内核和用户都能访问的变量 sch_timer。为保证内核安全，不暴露内核地址，我们使用 kmalloc()在堆中给变量分配一个空间（这是堆中第一个分配的空间，故我们知道其地址是 0x5200_0000）。

Memory Layout

This time we just use direct memory mapping. Memory we used can be divided into parts shown below:

```
0x50200000      0x50500000                        0x52000000        0x600000000
     +--------------+---------------------------------+------------------+
     |    Kernel    |                                 |                  |
     |     Code     |        Process Stack zone       |       Heap       |
     |      and     |   each stack contains 4K space  |   For kmalloc()  |
     |     data     |                                 |                  |
     +--------------+---------------------------------+------------------+
```

图表 21 我自定义的内存布局（详见仓库/Project2-SimpleKernel/README.md）

在进入 do_scheduler()时，我们把当前时间戳记录在 sch_timer 中。

```c
void do_scheduler(void)
{
      *sch_timer = get_ticks();
```

图表 22 内核记录调度开始时间

当将要调起的是用户辅助测试进程时，辅助进程就会读取 sch_timer 的值，并用 get_ticks() 获得的当前时间戳减去 sch_timer 的值并打印。

```
void do_sch_timer(void)
{
    uint64_t *sch_timer = 0x52000000;
    sys_move_cursor(1, 1);
    printf("> Last do_scheduler costs %d ticks \n", get_ticks() - *sch_timer);
    while(1);
}
```

图表 23 调度开销用户辅助测试进程

这样我们就统计了一次调度（从 do_scheduler 开始到进入进程）的切换开销。

```
> Last do_scheduler costs 2384 ticks
> [TASK1] This task is to test priority P_1. (24676)
> [TASK2] This task is to test priority P_2. (33232)
> [TASK3] This task is to test priority P_3. (48951)
> [TASK4] This task is to test priority P_4. (78077)
```

图表 24 调度的开销

我们的时间片长度是 80000 个 tick，切换开销占比$\frac{2384}{80000} \times 100\% = 2\%$，可以看出切换开销相对于整个时间片，在合理范围内。（我认为超过 10%就意味着一次切换的代价过大）

4. 进程锁的设计与实现

在代码框架更新后，我们不需实现线程锁，而是实现进程锁。线程锁要求使用锁的线程共用同一个地址空间。而进程锁实现在内核中，进程可通过系统调用的方式获得锁和释放锁，不再有地址空间隔离的限制。

这里我们采用类 UNIX 的二元量的设计来设计锁。

（1） 锁的结构和基本操作

锁的结构如下：

```
typedef enum {
    BIN_LOCKED,
    BIN_UNLOCKED,
} binsem_status_t;

typedef struct binsem {
    uint8_t id;
    uint8_t key;
    binsem_status_t status;
    list_head block_queue;
} binsem_t;
```

图表 25 用户态锁的结构

每个锁有一个独特的 id，这个 id 后面用随机数的方式生成。Key 是这个锁的标志，用户可以通过 key 值申请或找到一把锁。Status 即二元状态，另外每个锁都有自己的阻塞队列。

（2） 用户态函数接口

须提供给用户以下两个函数：

```
int binsemget(int key);
```

此函数的作用是：用户提供一个 key，返回值为 key 对应的锁的 id，以方便后续访问。当没有锁的 key 值与用户提供的参数相等时，应新建一把锁，并返回其 id。

```
int binsemop(int binsem_id, int op);
```

此函数的作用是：用户提供锁 id 和完成操作（获得锁、释放锁）。

这两个函数是用户态接口函数，它们须调用 invoke_syscall，陷入内核进行系统调用。

（3） **内核态处理函数**

内核中，我们可以开一个 binsem_t 的指针数组，用于指向我们创建的锁（这样节约空间）。每次新开一把锁时，我都通过 kmalloc 在堆中申请一块空间存储锁的信息，以实现资源的最节约化（锁空间的**动态申请分配**）

```
binsem_t *binsem_list[MAX_BINSEM];
```

我们需要两个处理函数：**获得锁 ID** 和**对锁操作（获得锁、释放锁）**的函数。它们被封装为系统调用，也就是（2）中用户态接口函数对应的内核处理函数。

a. **获得锁 ID**（./kernel/locking/lock.c 中的 kernel_binsemget，对应用户态的 binsemget 接口）

遍历整个指针数组，如果发现某个锁的 key 值恰为用户所求，则返回锁的 id，否则新建一把锁（这样保证了两个进程同时申请同一 key 时，不会建立两把具有相同 key 值的锁）。

通过 kmalloc 分配空间（堆中，动态分配）。

ID（下标）通过自实现时间伪随机发生器随机生成，保证安全性。

ID 和 key 值均不会暴露内核的地址，保证用户进程不会发现锁的地址而进行恶意篡改。

```
uint8_t id_gen()
{
    uint64_t random = get_ticks();
    uint64_t mask = 0xff;
    return (uint8_t)(random & mask);
}
```

图表 26 基于时间戳的伪随机数发生器

```
int kernel_binsemget(int key)
{
    uint8_t id;
    for(int i = 0; i < MAX_BINSEM; i++){
        if (binsem_list[i])
            if (binsem_list[i]->key == key){
                return binsem_list[i]->id;
            }
    }
    // Generate id
    for(id = id_gen(); binsem_list[id] != NULL; id = id_gen());
    // Alloc memory space
    binsem_t *new = (binsem_t *) kmalloc(sizeof(binsem_t));
    // Init
    new->id = id;
    new->key = key;
    new->status = BIN_UNLOCKED;
    init_list_head(&new->block_queue);
    // Add to list
    binsem_list[id] = new;
    return id;
}
```

图表 27 具体实现（前半部分是在寻找是否已经有此 key 值，后半部分是若找不到 key，则新建一把锁，分配 id 和空间）

b. 操作函数（./kernel/locking/lock.c 中的 kernel_binsemop，对应用户态的 binsemop 接口）

首先根据用户提供的 ID 找到对应的锁。对于请求锁操作，若锁被占用，则将当前进程阻塞（do_block），加入本锁的阻塞队列，否则拿到锁。

```
binsem_t *bin = binsem_list[binsem_id];
if (op == BINSEM_OP_LOCK){
    if(bin->status == BIN_LOCKED)
        do_block(current_running, &bin->block_queue);
    else
        bin->status = BIN_LOCKED;
```

图表 28 申请锁

对于释放锁，当阻塞队列为空时，可直接将状态记为 BIN_UNLOCK。否则，调用 do_unblock 方法唤起一个等待的进程。

```
} else if (op == BINSEM_OP_UNLOCK){
    if(list_empty(&bin->block_queue))
        bin->status = BIN_UNLOCKED;
    else{
        do_unblock(bin->block_queue.next);
    }
```

图表 29 释放锁

参考文献

[1] Raspberry Pi-OS Lesson: Scheduler (https://github.com/s-matyukevich/raspberry-pi-os/blob/master/docs/lesson04/rpi-os.md)

[2] 三十天自制操作系统 [日] 川和秀实

[3] ICS2019, Nanjing University (https://nju-projectn.github.io/ics-pa-gitbook/ics2019/)

[4] Xv6-public, MIT (https://github.com/mit-pdos/xv6-public)

[5] RISC-V Chinese Reader. CRVA, ICT (中国开放指令集联盟，中科院计算所)

[6] RISCV-Privileged-ISA Handbook, Berkeley