Project 3 Interactive OS and Process Management 设计文档

中国科学院大学
贾志杰
2020 年 11 月 14 日

1. Shell 设计

（1）shell 实现过程中遇到的问题和得到的经验

①从键盘输入到串口的回车不是'\n'=10，而是 13，因此判定输入回车时需要判定 input=13。

②虚拟屏幕只有 30 行，如果光标 y 超过这个范围会造成打印混乱，需要在光标 y 大于 30 时使用滚屏。滚屏功能已经在 screen.c 中实现，需要封装为系统调用。

③为了使 shell 更加人性化，我们可以在解析命令时忽略其中前导和重复的空格。

④可以在用户输入未知命令或已知命令参数不匹配时给出报错提示。

2. kill 和 wait 内核实现的设计

（1）kill 处理过程中如何处理锁，是否有处理同步原语，如果有处理，请说明。

在 pcb 中增加一个 lock 队列，用于记录进程持有的锁的 id。在杀死或主动结束进程时，检测其持有 lock 队列，释放持有的全部锁。

对于同步原语导致的阻塞，在杀死或主动结束进程时，也需要检测各同步原语的阻塞队列。当然需要指出的是，对于同步原语的阻塞，杀死进程不一定能释放其他进程，比如生产者-消费者中的条件变量，即使杀死生产者进程，由于没有产品可供消费，消费者还是会被继续阻塞。

（2）wait 实现时，等待的进程的 PCB 用什么结构保存？

在 pcb 中增加一个 block_task 队列，用于记录因为等待该进程而被阻塞的进程的 pid。当进程等待其他进程时，它会被阻塞并将它的 pid 写入它要等的那个进程的阻塞队列。

（3）设计或实现过程中遇到的问题和得到的经验

在杀死进程时，与 exit 相比，还需要增加检查就绪队列和全局阻塞队列的步骤。因为对于 exit，current_running 就是要结束的进程，它处于 RUNNING 态，必然不在就绪队列和全局阻塞队列中。但 kill 的进程可能正在就绪队列和全局阻塞队列中，如果不把它从这些队列中移除，那么之后可能再次被调度运行，并且因为那时它的资源已经被释放，会导致出错。

3. 同步原语设计

（1）信号量和屏障实现的各自数据结构的包含内容

①条件变量

```
typedef struct condition{
    pid_t block_task[NUM_MAX_TASK];
```

```
    uint32_t block_task_count;
} condition_t;
```

条件变量结构体包含一个阻塞队列和计数指针，用于存放被条件变量阻塞的进程的 pid。

条件变量主要包括 wait、signal、broadcast 操作。

其中 wait 操作首先释放锁，然后阻塞当前进程。之后当解除阻塞后，获取锁进入临界区。代码如下所示。

```
void do_condition_wait(lock_id_t lock_id, condition_t *condition){
    /* [1] release the lock
    do_mutex_lock_release(lock_id);
    /* [2] do scheduler */
    do_block(condition->block_task, &(condition->block_task_count));
    /* [3] acquire lock */
    do_mutex_lock_acquire(lock_id);
}
```

broadcast 操作唤醒所有被条件变量阻塞的进程，只需调用 do_unblock_all 函数即可。

```
void do_condition_broadcast(condition_t *condition){
    do_unblock_all(condition->block_task,
    &(condition->block_task_count));
}
```

signal 只唤醒一个进程，与 broadcast 类似，调用之前实现的 do_unblock_one 即可，这里不再赘述。

②屏障

```
typedef struct barrier{
    int goal;
    int now;
    pid_t block_task[NUM_MAX_TASK];
    uint32_t block_task_count;
} barrier_t;
```

屏蔽结构体包含目标数量 goal、当前达到数量 now、以及一个阻塞队列和计数指针。

屏障主要包括 init、wait 操作。

init 操作只需根据提供的目标数量将屏障的 goal 初始化，并将 now 清零即可。

wait 操作首先把 now 数量加 1，然后判断是否达到目标数量 goal，如果没有达到就将该进程阻塞，如果达到目标就唤醒所有被屏障阻塞的进程，并将 now 清零。

```
void do_barrier_wait(barrier_t *barrier){
    (barrier->now)++;
    if(barrier->now < barrier->goal){
        do_block(barrier->block_task, &(barrier->block_task_count));
    }
    else{
        barrier->now = 0;
        do_unblock_all(barrier->block_task,
        &(barrier->block_task_count));
    }
```

```
    }
```

4. mailbox 设计

（1）mailbox 的数据结构以及主要成员变量的含义

```
    typedef struct mailbox{
        char msg[MAX_MBOX_LENGTH];
        int msg_ptr;
        int open_num;
    } mailbox_t;
```

mailbox 结构体包括一个存放消息的 char 数组和指针，以及一个指示打开邮箱人数的变量 open_num。

每当一个进程使用 mbox_open 成功打开邮箱就将 open_num 加 1；当使用 mbox_close 时，将 open_num 减 1，同时检查 open_num 是否为 0，如果为零就关闭并回收该邮箱。

msg[]用于存放发送者发送的消息，并供接收者读取。msg_ptr 指示 msg[]中下一个待写入的字符位置。

（2）你在 mailbox 设计中如何处理 producer-consumer 问题，使用哪种同步原语进行并发访问保护？你的实现是否支持多 producer 或多 consumer，如果有，你是如何处理的？

使用管程，即一个锁加上条件变量进行保护。在发送方或接收方想要访问邮箱前，需要先申请一把锁，申请到锁后判定邮箱是否已满或已空，已满或已空则由条件变量负责阻塞，条件满足则访问邮箱 msg[]，访问完成后释放锁并唤醒阻塞的进程。以 send 操作为例：

```
    void do_mbox_send(mailbox_id_t boxid, void *msg, int msg_length){
        do_mutex_lock_acquire(alloc_lock);
        while(mbox_is_full(boxid, msg_length)){
            do_condition_wait(alloc_lock, &alloc_cond);
        }
       memcpy(&(mboxs[boxid].msg[mboxs[boxid].msg_ptr]),msg,msg_length);
        mboxs[boxid].msg_ptr += msg_length;
        do_mutex_lock_release(alloc_lock);
        do_condition_broadcast(&alloc_cond);
    }
```

邮箱支持多个接收方和多个发送方，但同时只允许一个进程访问邮箱内容。使用锁来保证，只有拿到锁的进程才能进入临界区访问邮箱内容。

5. 双核使用设计

（1）你在启用双核时遇到的问题有什么，如何解决的？

一开始不知道如何在内核中区分两个核，后来查手册得知可以通过 CP0-15 号寄存器读取当前核的 id。

（2）你是如何让不同的任务在不同的核上运行的？

使用两个就绪队列存放在两个核上运行的进程，current_running 也扩展为 2 项。同时 pcb 中增加一个 incore 表示进程可以处于哪个核运行，incore=0 表示该进程只能在核 0 上运行，

在调度时把它加入核 0 的就绪队列，incore=1 则只能在核 1 上运行。incore=2 则可以在任意核上运行，可以根据当前 core_id 选择加入哪个就绪队列。

（3）你在双核上如何保证同步原语和 kill 的正确性？

①kill

如果 kill 的对象不在另外一个核上 running，那么处理方式与之前相同。但如果 kill 的对象正在另外一个核上 running，那么就不能直接对其操作。

处理方式为，维护一个全局 kill[2]数组，这种情况下给对方核的 kill[core_id]置为 kill 对象的 pid。每次不管哪个核进入内核都检查这个自己的 kill 数组那项，如果不为 0 就处理这个 kill 命令。那么给对方核的 kill 数组项标记后，对方核就会在下次进入内核时 kill 掉目标进程。当然这个方法的 kill 会有一些延迟，不过最多一个时间片，总体上可以接受。

②同步原语

之前同步原语的操作基本都是放在内核中，并封装为系统调用，因此双核不会有影响。但需要注意一点，有些同步原语的实现中，进程会阻塞在内核中，这样当他被唤醒时就是直接在内核中运行，但是在载入下文后我们就直接释放锁，这样就会出现两个核同时进入内核，造成错误。因此我们在载入下文后，要先判断之后是位于内核还是用户态，如果是内核就不释放锁，而是等待其系统调用返回时再释放锁。

（4）设计或实现过程中遇到的问题和得到的经验

如双核同步原语那里所说，一开始没有考虑到阻塞在内核的情况，出现了两个核同时在内核跑的情况。解决方法为载入下文时判断之后是否处于内核态再释放锁。

6. 关键函数功能

（1）读取一个字符 getchar 函数

```
char getchar(void)
{
    char *tx_fifo_write = (char *)0xffffffffbfe00000;
    char *tx_fifo_state = (char *)0xffffffffbfe00005;
    while(1){
        if( (*tx_fifo_state) & 0x01 )
        return (*tx_fifo_write);
    }
}
```

（2）shell 回显和退格

```
while(1){
    input = getchar();
    if(input == '\b' || input == 127){
       if(buff_ptr > 0){
           cursor_x--;
           buff_ptr--;
           sys_move_cursor(cursor_x, cursor_y);
           printf(" ");
```

```
        }
    }
    else if(input == 13 || input == '\n'){
        buff[buff_ptr] = '\0';
        break;
    }
    else{
        buff[buff_ptr++] = input;
        sys_move_cursor(cursor_x, cursor_y);
        printf("%c", input);
        cursor_x++;
    }
}
```

（3）do_spawn 函数

```
pid_t do_spawn(task_info_t *task, int incore)
{
    pid_t pid = process_id++;
    if(incore == 2)
        set_pcb(pid, pcb, task, pid%2);
    else
        set_pcb(pid, pcb, task, incore);
    return pid;
}
```

（4）do_waitpid 函数

```
int do_waitpid(pid_t pid)
{
    pid_t proc_id = get_pcb(pid);
    if(proc_id == -1 || pcb[proc_id].status == TASK_EXITED){
        return -1;
    }
    current_running[core_id]->status = TASK_BLOCKED;
    current_running[core_id]->inqueue = BLOCK;
    pcb[proc_id].block_task[++(pcb[proc_id].block_task_count)] = current_running[co
re_id]->pid;
    do_scheduler();
    return 0;
}
```

（5）do_exit 函数

```
void do_exit(void)
{
    current_running[core_id]->status = TASK_EXITED;
```

```
    current_running[core_id]->inqueue = 0;
    current_running[core_id]->pid = -1;
    free_kernel_stack(current_running[core_id]->kernel_sp);
    free_user_stack(current_running[core_id]->user_sp);
    do_unblock_all(current_running[core_id]->block_task, &(current_running[core_id]
->block_task_count));
    while(current_running[core_id]->hold_lock_count){
        do_mutex_lock_release(current_running[core_id]->hold_lock[current_running[c
ore_id]->hold_lock_count]);
    }
    do_scheduler();
}
```

（6）do_kill 函数

```
int do_kill(pid_t pid)
{
    int i, j, find;
    pcb_t *one;
    pid_t proc_id = get_pcb(pid);

    if(pid <= 1)
        return 0;
    if(proc_id == -1 || pcb[proc_id].status == TASK_EXITED)
        return -1;
    if(core_id == 0 && current_running[1] == &pcb[proc_id]){
        kill[1] = pid;
        return pid;
    }
    if(core_id == 1 && current_running[0] == &pcb[proc_id]){
        kill[0] = pid;
        return pid;
    }
    if(pcb[proc_id].status = TASK_READY){
        find = 0;
        for(i=0; i<CORE_NUM; i++){
            for(j=0; j<NUM_MAX_PRIORITY; j++){
                if(!queue_is_empty(&ready_queue[i][j])){
                    one = (pcb_t*)(ready_queue[i][j].head);
                    while(one != NULL){
                        if(one == &pcb[proc_id]){
                            find = 1;
                            goto FIND_PROC;
                        }
                        one = (pcb_t*)(one->next);
```

```
                    }
                }
            }
        }
        FIND_PROC:
        if(find == 1){
            queue_remove(&ready_queue[i][j], &pcb[proc_id]);
        }
    }
    else if(pcb[proc_id].status = TASK_BLOCKED){
        find = 0;
        if(!queue_is_empty(&block_queue)){
            one = (pcb_t*)(block_queue.head);
            while(one != NULL){
                if(one == &pcb[proc_id]){
                    find = 1;
                    break;
                }
                one = (pcb_t*)(one->next);
            }
        }
        if(find == 1){
            queue_remove(&block_queue, &pcb[proc_id]);
        }
    }
    pcb[proc_id].status = TASK_EXITED;
    pcb[proc_id].inqueue = 0;
    pcb[proc_id].pid = -1;
    free_kernel_stack(pcb[proc_id].kernel_sp);
    free_user_stack(pcb[proc_id].user_sp);
    do_unblock_all(pcb[proc_id].block_task, &(pcb[proc_id].block_task_count));
    i = pcb[proc_id].hold_lock_count;
    while(i){
        lock_id_t lock_id = pcb[proc_id].hold_lock[i];
        do_unblock_all(lock_queue[lock_id].block_task, &(lock_queue[lock_id].block_
task_count));
        lock_queue[lock_id].status = UNLOCKED;
        lock_queue[lock_id].holder = 0;
        i--;
    }
    pcb[proc_id].hold_lock_count = 0;
    return pid;
}
```

（7）条件变量主要函数

```
void do_condition_wait(lock_id_t lock_id, condition_t *condition)
{
    do_mutex_lock_release(lock_id);
    do_block(condition->block_task, &(condition->block_task_count));
    do_mutex_lock_acquire(lock_id);
}
void do_condition_signal(condition_t *condition)
{
    pcb_t *unblock_one;
    if(condition->block_task_count){
        pid_t unblock_one_pid = get_pcb(condition->block_task[condition->block_task
_count]);
        if(unblock_one_pid > 0){
            unblock_one = &pcb[unblock_one_pid];
            do_unblock_one(unblock_one);
        }
        (condition->block_task_count)--;
    }
}
void do_condition_broadcast(condition_t *condition)
{
    do_unblock_all(condition->block_task, &(condition->block_task_count));
}
```

（8）屏障主要函数

```
void do_barrier_init(barrier_t *barrier, int goal)
{
    barrier->now = 0;
    barrier->goal = goal;
    barrier->block_task_count = 0;
}
void do_barrier_wait(barrier_t *barrier)
{
    (barrier->now)++;
    if(barrier->now < barrier->goal){
        do_block(barrier->block_task, &(barrier->block_task_count));
    }
    else{
        barrier->now = 0;
        do_unblock_all(barrier->block_task, &(barrier->block_task_count));
    }
}
```

（9）mailbox 主要函数

```
void do_mbox_send(mailbox_id_t boxid, void *msg, int msg_length)
{
    do_mutex_lock_acquire(alloc_lock);
    while(mbox_is_full(boxid, msg_length)){
        do_condition_wait(alloc_lock, &alloc_cond);
    }
    memcpy(&(mboxs[boxid].msg[mboxs[boxid].msg_ptr]), msg, msg_length);
    mboxs[boxid].msg_ptr += msg_length;
    do_mutex_lock_release(alloc_lock);
    do_condition_broadcast(&alloc_cond);
}
void do_mbox_recv(mailbox_id_t boxid, void *msg, int msg_length)
{
    do_mutex_lock_acquire(alloc_lock);
    while(mbox_is_empty(boxid, msg_length)){
        do_condition_wait(alloc_lock, &alloc_cond);
    }
    mboxs[boxid].msg_ptr -= msg_length;
    memcpy(msg, &(mboxs[boxid].msg[mboxs[boxid].msg_ptr]), msg_length);
    do_mutex_lock_release(alloc_lock);
    do_condition_broadcast(&alloc_cond);
}
```

（10）启动从核

```
void loongson3_boot_secondary(void)
{
    init_pcb_core1();
    uint64_t *Mailbox = (uint64_t *)0xffffffffbfe11120;
    Mailbox[1] = pcb[32].user_sp;
    Mailbox[2] = get_reg_gp();
    Mailbox[3] = 0;
    Mailbox[0] = (uint64_t)&smp_bootstrap;
}
```

（11）从核初始化

```
void smp_bootstrap(void)
{
    asm_start_core1();
    set_cp0_count(0);
    set_cp0_compare(TIMER_INTERVAL);
    enable_interrupt();
    while(1){
        ;
```

```
    }
}
```

（12）内核锁

获取锁：

```
.macro CORE_LOCK_ACQUIRE offset
    .set    noat
    dla    k1, core_lock
    Atomic_lock_acquire:
    ll     k0, 0(k1)
    bnez   k0, Atomic_lock_acquire
    daddi  k0, $0, 1
    sc     k0, 0(k1)
    beqz   k0, Atomic_lock_acquire
    sync
    .set    at
.endm
```

释放锁：（需要考虑之后是否是内核态再决定是否释放锁）

```
.macro CORE_LOCK_RELEASE offset
    .set    noat
    dla   k0, current_running
    dmfc0 k1, $15, 1
    andi  k1, k1, 0x3ff
    dsll  k1, k1, 3
    daddu k1, k1, k0
    ld    k0, 0(k1)
    ld    k1, 32(k0)
    bnez  k1, 1f
    sync
    dla    k0, core_lock
    sw     $0, 0(k0)
    1:
    nop
    .set    at
.endm
```

（13）绑核

```
int do_band(pid_t pid, int core)
{
    pid_t proc_id = get_pcb(pid);
    if(pid <= 1)
        return 0;
    if(proc_id == -1 || pcb[proc_id].status == TASK_EXITED)
        return -1;
```

```
    pcb[proc_id].incore = core;
    return pid;
}
```

参考文献

[1] MIPS® Architecture For Programmers Volume II-A: The MIPS64® Instruction Set Reference Manual, Revision 6.05, 2016

[2] 龙芯 GS264 处理器核用户手册, v1.0, 2018: 93-98, 101

[3] 现代操作系统(第 4 版)[M]. 北京: 机械工业出版社, 2017: 78-84