Project 3 Interactive OS and Process Management 设计文档

中国科学院大学

吴俊亮

2020 年 11 月 14 日

1. Shell 设计

(1) shell 实现过程中遇到的问题和解决方法

从 screen.c 中我们可以知道屏幕的高度是 50 个字符，屏幕的宽度是 80 个字符。但是显示器大小会影响 qemu 和 minicom 中屏幕的高度，实测屏幕高度只有 35-45 个字符。为了保证正确性，我在实际实现中只使用了屏幕的第 1-30 行，其中 1-14 行用于测试程序输出，15-30 行用于命令行输入。

(2) shell 支持的命令

命令	参数	说明
ps	无	显示所有进程的 pid，状态，MASK 和运行在哪个核上
reset	无	清空下半屏（命令行）
clear	无	清空上半屏（用户输出）
exec	tasknum	将第 tasknum 个任务启动
kill	pid	杀死进程号为 pid 的进程
taskset	mask tasknum -p mask pid	将第 tasknum 个任务启动并设置 MASK 设置进程号为 pid 的进程的 MASK

2. kill 和 wait 内核实现的设计

(1) kill 的实现和锁的处理

do_kill()函数中主要要完成如下工作：①将等待队列中的进程解除阻塞；②释放获得的 binsem 锁；③释放用户栈；④将进程从就绪队列或阻塞队列中删除；⑤删除在 timer 队列中该进程的 timer；⑥进入 ZOMBIE 状态，等待父进程或者 init 进程回收 PCB 和内核栈。

为了在 kill 时能够释放 binsem 锁，我们需要在 PCB 中记录当前进程获得锁的数量和锁

的 ID，在进程获取锁时 binsem_num++，在进程释放锁时 binsem_num--。在 do_kill()中释放锁的代码如下所示，注意释放锁后若有进程要获取锁，还要将它的 PCB 也做相应的修改。

```
    // release binsem mutex
    for(int i = 0; i < killed_pcb->binsem_num; i++){
        int binsem_id = killed_pcb->binsem_id[i];
        binsem_nodes[binsem_id].sem++;
        if(binsem_nodes[binsem_id].sem <= 0){
            list_node_t * unblocked_pcb_list = binsem_nodes[binsem_id].
block_queue.next;
            pcb_t *unblocked_pcb = list_entry(unblocked_pcb_list, pcb_t
, list);
            do_unblock(unblocked_pcb_list);
            unblocked_pcb->binsem_id[unblocked_pcb->binsem_num] = binse
m_id;
            unblocked_pcb->binsem_num++;
        }
    }
```

(2) wait 的实现

进程 waitpid 等待另一个进程时（不妨称“另一个进程”为子进程），若子进程已经退出，那么子进程为 ZOMBIE 状态，此时回收它的内核栈和 PCB 即可；若子进程未退出，则将自己阻塞在子进程的等待队列中，子进程退出后会将等待队列内的进程解除阻塞，此时我们再回到 do_waitpid()中回收它的内核栈和 PCB。考虑到父进程被子进程释放后会直接回到用户态，这样就无法为子进程回收内核栈和 PCB，为解决这个问题用了一个比较 tricky 的方法，如果子进程未退出，则将 sepc 减 4，这样父进程被释放时就会再次 ecall，回到 do_waitpid()函数中为子进程回收内核栈和 PCB。相关代码如下。

```
    if(child_pcb->status == TASK_ZOMBIE){
        // release pcb
        child_pcb->status = TASK_EXITED;
        child_pcb->pid = -1;
        // release kernel stack
        freePage(child_pcb->kernel_stack_base, 1);
        return 1;
    }
    else{
```

```
        // if child task has not exited, call do_waitpid again when par
ent task is unblocked
        // trick: return pid to keep a0 unchanged
        regs->sepc = regs->sepc - 4;
        do_block(&current_running[cpu_id]->list, &child_pcb->wait_list)
;
        scheduler();
        return pid;
    }
```

3. 同步原语设计

(1) 信号量的实现

信号量的实现利用了用户态的原子操作和 futex 锁。信号量定义为一个 int 型变量，记录阻塞的进程的数量。进程实际上阻塞在一个 futex 锁上，futex 锁的 ID 是信号量的地址。signal 和 broadcast 利用之前实现的 futex_wakeup 系统调用可以实现。相关代码如下。

```
typedef atomic_int mthread_cond_t;

int mthread_cond_wait(mthread_cond_t *cond, int binsem_id)
{
    fetch_add(cond, 1);
    sys_binsem_op(binsem_id, BINSEM_OP_UNLOCK);
    sys_futex_wait((unsigned long*)cond);
    sys_binsem_op(binsem_id, BINSEM_OP_LOCK);
    return 1;
}
int mthread_cond_signal(mthread_cond_t *cond)
{
    sys_futex_wakeup((unsigned long*)cond, 1);
    fetch_sub(cond, 1);
    return 1;
}
int mthread_cond_broadcast(mthread_cond_t *cond)
{
    sys_futex_wakeup((unsigned long*)cond, *cond);
    atomic_exchange(cond, 0);
    return 1;
```

}

(2) 屏障的实现

屏障的实现同样利用了用户态的原子操作和 futex 锁，但考虑到原子操作仍不能满足互斥访问的需求，因此添加了 binsem 锁用于保护临界区的数据。在进程调用 mthread_barrier_wait 时首先获取锁，然后将到达的进程数量加一（barrier->reached，注意此变量是临界区数据），若全部进程都到达了，就释放进程；否则将进程阻塞在 futex 锁上，futex 锁的 ID 是屏障的地址。相关代码如下。

```
typedef struct mthread_barrier
{
    int count;
    atomic_int reached;
    int binsem_id;
} mthread_barrier_t;

void mthread_barrier_wait(mthread_barrier_t *barrier)
{
    sys_binsem_op(barrier->binsem_id, BINSEM_OP_LOCK);
    fetch_add(&barrier->reached, 1);
    if((barrier->reached) == (barrier->count)){
        atomic_exchange(&barrier->reached, 0);
        sys_binsem_op(barrier->binsem_id, BINSEM_OP_UNLOCK);
        sys_futex_wakeup((unsigned long*)barrier, barrier->count);

    }
    else{
        sys_binsem_op(barrier->binsem_id, BINSEM_OP_UNLOCK);
        sys_futex_wait((unsigned long*)barrier);
    }
}
```

4. mailbox 设计

（1） mailbox 的数据结构以及主要成员变量的含义

同步变量用于线程间同步操作，可以使用共享内存存储屏障或者信号量，但进程间通信不能借助共享内存，只能请内核帮忙。mailbox 主要实现在内核态，和 binsem 锁类似，用户

进程通过 mailbox 的名称获取一个 mailbox 的 ID，然后以该 ID 向 mailbox 发送或从 mailbox 接收数据。mailbox 结构体如下所示，其中的 id 既是返回用户进程的 ID，也是其对应的 message 结构体数组的下标，一个 mailbox 对应一个 message。message 中 wait_queue 表示被阻塞在该 mailbox 上的进程，opened 表示当前 mailbox 被多少进程访问，msg 是实际存储数据的数组，msg_len 表示当前存储的数据的长度。

```
typedef struct mailbox
{
    char name[100];
    int id;
} mailbox_t;

typedef struct message
{
    list_head wait_queue;
    int opned;
    char msg[MAX_MBOX_LENGTH];
    int msg_len;
} message_t;
```

（2） producer-consumer 问题的处理

生产者通过 mbox_send()向 mailbox 发送消息，该函数通过系统调用访问 message 结构体，对应的系统调用函数为 do_mbox_send()。若发送的消息不能被接收（消息长度加原有的数据长度超过了允许的数据长度的上限），则将该进程阻塞；若发送的消息被接收，则唤醒在阻塞队列上的消费者。相关代码如下。

```
int do_mbox_send(int mailbox_id, void *msg, int msg_length)
{
    message_t *cur_message = &message[mailbox_id];
    if(cur_message->msg_len + msg_length >= MAX_MBOX_LENGTH){
        do_block(&current_running[cpu_id]->list, &cur_message->wait_que
ue);
        scheduler();
        return 0;
    }
    else{
        kmemcpy(&(cur_message->msg[cur_message->msg_len]), msg, msg_len
gth);
```

```
        cur_message->msg_len += msg_length;
        while(!list_empty(&cur_message->wait_queue)){
            do_unblock(cur_message->wait_queue.next);
        }
        scheduler();
        return 1;
    }
}
```

消费者通过 mbox_recv()向 mailbox 请求接收消息，该函数通过系统调用访问 message 结构体，对应的系统调用函数为 do_mbox_recv()。若请求接收的消息长度大于已有数据长度，则将该进程阻塞；若成功接收了消息，则唤醒在阻塞队列上的生产者。

```
int do_mbox_recv(int mailbox_id, void *msg, int msg_length)
{
    message_t *cur_message = &message[mailbox_id];
    if(msg_length > cur_message->msg_len){
        do_block(&current_running[cpu_id]->list, &cur_message->wait_que
ue);
        scheduler();
        return 0;
    }
    else{
        cur_message->msg_len -= msg_length;
        kmemcpy(msg, &(cur_message->msg[cur_message->msg_len]), msg_len
gth);
        while(!list_empty(&cur_message->wait_queue)){
            do_unblock(cur_message->wait_queue.next);
        }
        scheduler();
        return 1;
    }
}
```

5. 双核使用设计

（1） 在启用双核时遇到的问题

在启动双核时遇到如下两个问题：①主核收到了核间中断；②例外发生时未跳转到相应

的例外处理函数。解决前一个问题的方法是设置软件中断例外的处理函数是 clear_ipi()，该函数将 SIP 寄存器中的软件中断位清零。后一个问题出现的原因是主核唤醒从核后，从核进行了清空 bss 段的操作，该操作导致被初始化的 PCB、例外跳转表等信息被抹掉，解决方法是从核启动后不进行清空 bss 段操作。

（2） 如何让不同的任务在不同的核上运行

打开两个核的时钟中断后，我们需要维护两个 current_running。一个核进入内核后只操作自己的 current_running。为了实现绑核，对原有的优先级调度进行修改，原有的 max_priority_node()函数功能是寻找最大优先级的 PCB，现在我们考虑 PCB 中的 MASK 变量，若 MASK 表明该进程不应该在当前核上运行，则跳过该 PCB。

```
pcb_t *node_pcb = list_entry(node, pcb_t, list);
mask_flag = (cpu_id == 0 && ((node_pcb->mask & 0x1) == 0x1))
          || (cpu_id == 1 && ((node_pcb->mask & 0x2) == 0x2));
if(!mask_flag){
    continue;
}
```

（3） 双核中如何保证同步原语和 kill 的正确性

因为之前设计同步原语时已经考虑了并发访问的问题，使用了原子操作和锁保护了临界区数据，所以同步原语在双核下也能正常工作。对于 kill 掉正在另一个核上跑的进程的情况，我们采用简化的方案，直接将该进程的状态标记为 TASK_KILLED，然后返回。被标记为 TASK_KILLED 的进程在下一次时钟中断时在 scheduler()函数中被实际杀死，并回收相关资源。