Project3 Interactive OS and Process Management

中国科学院大学
陈灿宇
2018.11.25

1 Shell 设计

(1) shell 实现过程中遇到的问题和得到的经验（如果有的话可以写下来，不是必需项）

- 遇到的第一个问题是开始的时候想当然的以为应该以 '\n' 为依据来判断结尾，结果始终无法打印出结果，后来区分了一下换行 ('\n') 和回车 ('\r') 之后，才能打印出字符。
- 第二个问题是由于 lab2 的 schedule() 函数存在漏洞，在进程切换的时候会陷入死循环，改掉这个 BUG 之后 shell 就基本正常了。
- 我在最初设想 shell 的时候考虑的比较复杂，希望能解析不同的命令，同时具有较强的可扩展性，但由于时间有限，后来我还是采用了较为简单的方案，设计了一个较为精巧的结构体，通过指针移动，将命令存入到 buffer 中，然后从 buffer 中进行读取并进行解析。
- 滚屏的实现其实较为简单，因为在 screen_scroll() 和 screen_write_ch() 函数中已经考虑到了滚屏，如果只使用用户态的函数 printf 进行打印，避免使用 printk，就可以实现滚屏的效果。
- 我还设计了一个可以将 enum 类型名直接打印出来的函数，可以用于 ps 命令的解析进程状态。

```
typedef struct InputBuffer {
    char *buffer;
    int32_t buffer_length;
    int32_t input_length;
    int pointer;
} InputBuffer_t;
char Buffer[INPUT_BUFFER_MAX_LENGTH];
InputBuffer_t inputBuffer;

#define ENUM_TYPE_CASE(x)   case x: return(#x);
static inline const char *status_type_to_string(task_status_t status)
{
    switch (status)
    {
        ENUM_TYPE_CASE(TASK_BLOCKED)
        ENUM_TYPE_CASE(TASK_RUNNING)
        ENUM_TYPE_CASE(TASK_READY)
        ENUM_TYPE_CASE(TASK_EXITED)
        ENUM_TYPE_CASE(TASK_CREATED)
        ENUM_TYPE_CASE(TASK_SLEEPING)
    }
    return "Invalid Status";
}
```

2　spawn, kill 和 wait 内核实现的设计

（1）spawn 的处理过程，如何生成进程 ID

有一个全局变量 pid, 每有一个新 task 来时, 就 pid++ 赋给这个新 task，从而生成进程 ID。spawn 的处理过程是：

1. 查找 pcb 数组, 找到一个 status == TASK_EXITED 的 pcb 块, 若找不到则 spawn 直接返回。(init_pcb() 的时候已经将 pcb 数组中的所有 pcb 初始化为 TASK_EXITED 状态)
2. 将该 pcb 分配给新的 task，并根据 task_info 进行初始化
3. pid++ 产生新的 task 的 pid

（2）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。

- 针对同步原语的处理较为简单，采用了直接将进程杀死的方法。
- kill 的过程是先遍历 PCB 数组，找到对应 PID 的进程，如果进程状态是 TASK_EXITED，则直接进行进程切换; 如果是 TASK_BLOCKED，则遍历所有的锁，找到该进程阻塞的队列，然后将该进程从锁的等待队列中删去; 如果是 TASK_SLEEPING，则将该进程从 sleeping 队列中删去; 如果是 TASK_READY 或 TASK_CREATED，则将该进程从 ready 队列中删去。
- 然后遍历 PCB 中持有的锁的指针数组，将该进程持有的锁全部释放。
- 清空该进程的等待队列。
- 最后如果要杀死的进程是自己，则启动调度。

```
typedef struct pcb {
...
    mutex_lock_t *lock[LOCK_MAX_NUM];
    uint32_t lock_num;
    queue_t waiting_queue;
...
} pcb_t;
```

```
void do_kill(int n)
{
    int i = 0;
    while(pcb[i].pid != n){
        i++;
        if(i >= NUM_MAX_TASK){
            return;
        }
    }
    if(pcb[i].status != TASK_EXITED){
        if(pcb[i].status == TASK_READY){
            queue_remove(&ready_queue, &pcb[i]);
```

```
13          }
14          else if(pcb[i].status == TASK_BLOCKED){
15              int k = 0;
16              for(;k<MAX_LOCK_NUM_TOTAL;k++){
17                  if(check_in_queue(&(Lock[k]->mutex_lock_queue), &pcb[i])){
18                      queue_remove(&(Lock[k]->mutex_lock_queue), &pcb[i]);
19                  }
20              }
21          }
22          else if(pcb[i].status == TASK_SLEEPING){
23              queue_remove(&sleeping_queue, &pcb[i]);
24          }
25          else if(pcb[i].status == TASK_CREATED){
26              queue_remove(&ready_queue, &pcb[i]);
27          }
28          pcb[i].status = TASK_EXITED;
29          int j = 0;
30          for(;j<LOCK_MAX_NUM;j++){
31              if(pcb[i].lock[j] != 0){
32                  do_mutex_lock_release(pcb[i].lock[j]);
33              }
34          }
35          clear_waiting_queue(&(pcb[i].waiting_queue));
36      }
37      if(current_running->pid == n){
38          do_scheduler();
39      }
40  }
```

（3）wait 的处理过程

如果等待的 PID 不是 EXITED 状态，将 current_running 的状态设为 TASK_BLOCKED, 加入对应的进程的 waiting 队列中, 并启动调度。

(4) 设计或实现过程中遇到的问题和得到的经验（如果有的话可以写下来，不是必需项）

- kill 的实现过程是相对较为复杂，我在 PCB 中增加了一个数组 lock 记录该进程持有的锁的，所以如果 kill 的时候进程处于 blocked 状态，则必须先将该进程从它阻塞的那个进程中去除，再把它所持有的锁全部释放，避免被 kill 的进程进入 ready 队列。

3　同步原语设计

（1）条件变量数据结构和相关实现

数据结构为 1 个等待队列:

- init 时初始化等待队列;
- wait 时将 current running 加入等待队列, 释放锁并启动调度;
- signal 时将等待队列中的 1 个 PCB 加入 ready 队列;
- broadcast 时将等待队列中的所有 PCB 加入 ready 队列。

```
typedef struct condition{
    queue_t waiting_queue;
} condition_t;
```

（2）信号量数据结构和相关实现

数据结构为 1 个信号量值和 1 个等待队列:

- init 时初始化信号量值和等待队列;
- up 时判断等待队列是否为空, 若为空则将信号量值加 1, 否则将等待队列中的 1 个 PCB 加入 ready 队列;
- down 时判断信号量值是否为 0, 若为 0 则将 current_running 加入等待队列并启动调度, 否则将信号量值减 1。

```
typedef struct semaphore{
    int sem_value;
    queue_t waiting_queue;
} semaphore_t;
```

（3）屏障数据结构和相关实现

数据结构为 1 个能到达屏障的最大数量、1 个当前未到达屏障的数量和 1 个等待队列:

- init 时初始化能到达屏障的最大数量、当前未到达屏障的数量和等待队列;
- wait 时判断当前未到达屏障的数量是否为 1, 若为 1 则将等待队列中的所有 PCB 加入 ready 队列, 否则将当前未到达屏障的数量减 1, 并将 current_running 加入等待队列。

```
typedef struct barrier{
    int max_num;
    int not_arrive_num;
    queue_t waiting_queue;
} barrier_t;
```

4 mailbox 设计

(1) mailbox 的数据结构以及主要成员变量的含义

mailbox 的数据结构由三部分构成:

- 首先由长度为 MAX_MESSAGE_LENGTH 字节的数组构成数据类型 message;
- 然后由名称字符串 name、使用计数 count、消息组指针 ptr、信号量 send、信号量 recv 和消息组 msg 构成 mailbox 类型;
- 然后由 mailbox 构成 mboxs 数组

```
#define MAX_MBOX_LENGTH 32
#define MAX_MESSAGE_LENGTH 32
#define MBOX_NAME_LENGTH 32
#define MAX_NUM_BOX 32

typedef struct message{
    char value[MAX_MESSAGE_LENGTH];
} message_t;

typedef struct mailbox{
    char name[MBOX_NAME_LENGTH];
    int count;
    int ptr;
    mutex_lock_t lock;
    semaphore_t send, recv;
    message_t msg[MAX_MBOX_LENGTH];
} mailbox_t;

static mailbox_t mboxs[MAX_NUM_BOX];
```

(2) mailbox 设计

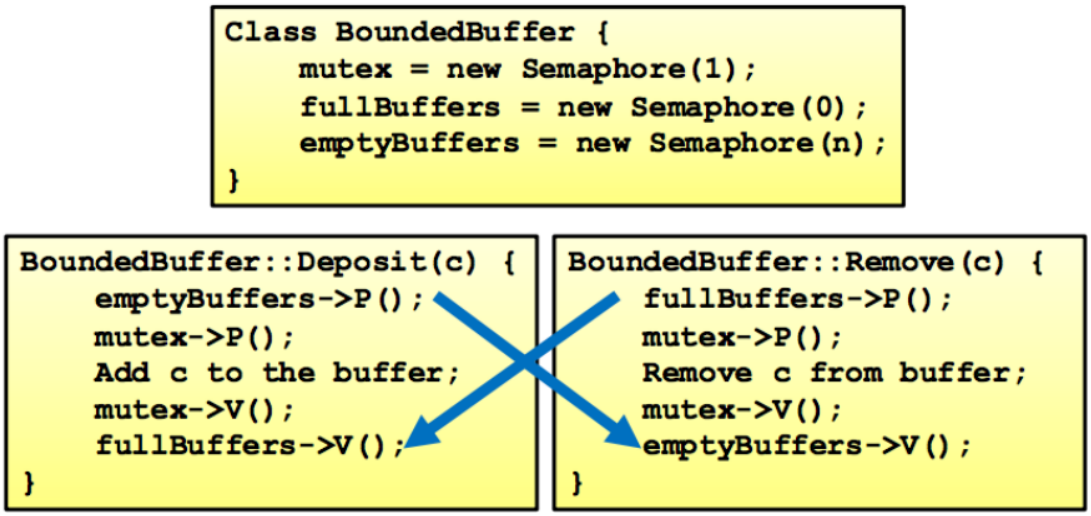
- init 时遍历 mboxs, 将所有 mailbox 的 name 设为空字符串,count 设为 0;
- open 时遍历 mboxs 寻找 name 与指定名称相同的 mailbox 并返回其编号; 若找不到则遍历 mboxs 寻找 count 为 0 的 mailbox,count 加 1,ptr 设为 0, 初始化信号量 send 为 MAX_MBOX_LENGTH, 初始化信号量 recv 为 0, 并返回其指针;
- close 时将指定编号 mailbox 的 count 减 1
- send 时 down 信号量 send, 获取锁, 复制消息到消息组, 移动消息组指针, 释放锁, 并 up 信号量 recv;
- recv 时 down 信号量 recv, 获取锁, 移动消息组指针, 从消息组复制消息, 释放锁, 并 up 信号量 send。

（3）你在 mailbox 设计中如何处理 producer-consumer 问题，你的实现中是否有多 producer 或多 consumer，如果有，你是如何处理的

生产者-消费者问题是多个进程共享有界缓冲区的问题, 一部分进程写缓冲区, 一部分进程读缓冲区, 不允许当缓冲区满时继续写或当缓冲区空时继续读。

本实现中使用 2 个信号量来控制是否允许读和写, 写缓冲区时视为申请写资源、释放读资源; 读缓冲区时视为申请读资源、释放写资源。当可用资源为 0 时申请不成功, 进程会被阻塞, 直到资源可用时被唤醒, 继续执行。

在三国的测试程序中曹操可以当作消费者，孙权和刘备既可以当作消费者也可以当作生产者，所以是存在多 producer 和多 consumer 的。通过在 mailbox 中添加两个信号量 send 和 recv 来分别表示写缓冲区和读缓冲区即可解决该问题。recv 初始值为 0, send 初始值为信箱大小。通过生产者、消费者的 PV 操作和锁对临界区的保护, 即可解决生产者-消费者问题。



（4）设计或实现过程中遇到的问题和得到的经验（如果有的话可以写下来，不是必需项）

- 注意这里的 semaphore_down 和 mutex_lock_acquire 的位置一定不能颠倒，否则会造成死锁!

```
1  void mbox_recv(mailbox_t *mailbox, void *msg, int msg_length)
2  {
3      semaphore_down(&(mailbox->recv));
4      mutex_lock_acquire(&(mailbox->lock));
5      memcpy(msg, &(mailbox->msg[mailbox->ptr]), msg_length);
6      mailbox->ptr = (mailbox->ptr + MAX_MBOX_LENGTH - 1) % MAX_MBOX_LENGTH;
7      mutex_lock_release(&(mailbox->lock));
8      semaphore_up(&(mailbox->send));
9  }
```

5　关键函数功能

请列出上述各项功能设计里，你觉得关键的函数或代码块，及其作用

较为复杂的由以下三个代码块：

(1) do_spawn()

```
1  void do_spawn(task_info_t *task_info)
2  {
3      int i = 0;
4      while(pcb[i].status != TASK_EXITED){
5          i++;
6          if(i >= NUM_MAX_TASK){
7              //ERROR handle
8              return;
9          }
10     }
11
12     queue_init(&(pcb[i].waiting_queue));
13
14     bzero(&(pcb[i].kernel_context), sizeof(pcb[i].kernel_context));
15     bzero(&(pcb[i].user_context  ), sizeof(pcb[i].user_context  ));
16     int j = 0;
17     for(; j < LOCK_MAX_NUM; j++){
18         pcb[i].lock[j] = NULL;
19     }
20     pcb[i].kernel_context.regs[29] = STACK_TOP;
21     pcb[i].user_context.regs[29] = STACK_TOP + STACK_SIZE;
22     pcb[i].kernel_context.regs[30] = STACK_TOP;
23     pcb[i].user_context.regs[30] = STACK_TOP + STACK_SIZE;
24     pcb[i].kernel_stack_top = STACK_TOP;
25     pcb[i].user_stack_top = STACK_TOP + STACK_SIZE;
26     STACK_TOP += STACK_SIZE*2;
27     if(STACK_TOP > STACK_MAX)
28     {
29         //ERROR handle
30     }
31
32     pcb[i].prev = NULL;
33     pcb[i].next = NULL;
34     pcb[i].pid = PID;
35     pcb[i].type = task_info->type;
36     pcb[i].status = TASK_CREATED;
37     pcb[i].cursor_x = 0;
38     pcb[i].cursor_y = 0;
39
40     pcb[i].entry_point = task_info->entry_point;
41
```

```
    pcb[i].kernel_context.regs[31] = (uint32_t)first_entry;

    pcb[i].kernel_context.cp0_status = CP0_STATUS_INIT;
    pcb[i].user_context.cp0_status = CP0_STATUS_INIT;

    pcb[i].kernel_context.cp0_epc = pcb[i].entry_point;
    pcb[i].user_context.cp0_epc = pcb[i].entry_point;
    //cp0_epc add 4 automatically when encountering interrupt

    pcb[i].mode = USER_MODE;
    pcb[i].priority = INITIAL_PRIORITY;
    pcb[i].wait_time = 0;
    pcb[i].sleeping_deadline = 0;
    pcb[i].lock_num = 0;

    PID++;
    queue_push(&ready_queue,&pcb[i]);
}
```

(2) do__mutex__lock__acquire() do__mutex__lock__release()

```
void do_mutex_lock_acquire(mutex_lock_t *lock)
{
    while(lock->status == LOCKED){
        do_block(&(lock->mutex_lock_queue));
    }
    lock->status = LOCKED;
    int i = 0;
    while(current_running->lock[i] != NULL){
        i++;
        if(i >= LOCK_MAX_NUM){
            //ERROR handle
            return;
        }
    }
    current_running->lock[i] = lock;
    current_running->lock_num++;
}

void do_mutex_lock_release(mutex_lock_t *lock)
{
    if(lock->status == LOCKED){
        lock->status = UNLOCKED;

        pcb_t *head = lock->mutex_lock_queue.head;
        while(head != NULL){
            int i = 0;
            while(head->lock[i] != lock){
```

```
                i++;
                if(i >= LOCK_MAX_NUM){
                    //ERROR handle
                    break;
                }
                head->lock[i] = NULL;
                head->lock_num--;
            }
            head = head->next;
        }

        do_unblock_all(&(lock->mutex_lock_queue));
    }
}
```

(3) do_semaphore_up() do_semaphore_down()

```
void do_semaphore_up(semaphore_t *s)
{
    if(!queue_is_empty(&(s->waiting_queue))){
        pcb_t *pcb = queue_dequeue(&(s->waiting_queue));
        pcb->status = TASK_READY;
        queue_push(&ready_queue, pcb);
    }
    else{
        s->sem_value++;
    }
}

void do_semaphore_down(semaphore_t *s)
{
    if(s->sem_value == 0){
        current_running->status = TASK_BLOCKED;
        queue_push(&(s->waiting_queue), current_running);
        do_scheduler();
    }
    else{
        s->sem_value--;
    }
}
```