# Project 3 Interactive OS and Process Management 设计文档

中国科学院大学

张磊

2019 年 11 月 9 日

## 1. Shell 设计

问题：

（1） 开始设计的 shell 的时候，以为\n 就是回车键，结果无法识别出命令，后来上网查阅之后才知道，mips 机器回车是\r\n，在这里卡了很长时间；

（2） 在对命令进行解析时，由于一开始知识比较简单地几个命令（kill，wait，exit），几乎没有参数，所以没有考虑到参数的问题，就直接用字符串比对的方法解析执行命令，后来做到 exec 的时候发现命令后边有参数，这样做不合理，于是又改为将输入命令拆分为主命令和参数两个部分，在进行命令解析和执行的时候又进一步根据参数个数细分；

## 2. spawn, kill 和 wait 内核实现的设计

（1）spawn 的实现：

```
275  void do_spawn(task_info_t *task)
276  {
277      int i = 0;
278      while(pcb_valid[i] && i < NUM_MAX_TASK)
279          i++;
280
281      if(i == NUM_MAX_TASK)
282      {
283          my_printf("There is no space for more process\n");
284          return ;
285      }
286
287      pcb_valid[i] = 1;
```

```
289     pcb[i].pid = process_id++;
290     pcb[i].type = task->type;
291     pcb[i].status = TASK_READY;
292
293     pcb[i].prev = NULL;
294     pcb[i].next = NULL;
295
296     pcb[i].kernel_context.regs[31] = (uint32_t)&init_handle;
297     pcb[i].user_context.regs[31] = task->entry_point;
298
299     pcb[i].kernel_context.cp0_epc = task->entry_point;
300     pcb[i].user_context.cp0_epc = task->entry_point;
301
302     pcb[i].kernel_context.cp0_status = 0x10008003;
303     pcb[i].user_context.cp0_status = 0x10008003;
304
305     pcb[i].kernel_context.regs[29] = pcb[i].kernel_stack_top;
306     pcb[i].user_context.regs[29] = pcb[i].user_stack_top;
307
308     pcb[i].priority = 1;
309     priority_queue_push(&pcb[i]);
```

Spawn 的实现其实就是 P2 的 init_pcb，只不过这次我们把它拆开来，根据任务序号依次初始化；

（2） kill 的实现：

```
312   void do_kill(pid_t pid)
313   {
314       recycle(pid);
315   }
```

Kill 的实现直接由于在很多函数里都会用到，所以直接用 recycle 函数封装起来，方便使用；

```
244   void recycle(pid_t pid)
245   {
246       int i;
247       for(i = 0; i < NUM_MAX_TASK; i++)
248       {
249           if(pcb[i].pid == pid)
250               break;
251       }
252
253       if(i == NUM_MAX_TASK)
254       {
255           my_printf("Wrong pid\n");
256           return ;
257       }
```

根据输入的 Pid 在 pcb_table 里查找，如果找到对应进程则回收其资源；

```
259     if(&pcb[i] == lock1.wait_queue.head)
260     {
261         do_mutex_lock_release(&lock1);
262     }
263
264     if(&pcb[i] == lock2.wait_queue.head)
265     {
266         do_mutex_lock_release(&lock2);
267     }
```

其次是判断，该要被回收的进程是否占用着锁，如果占用着，则释放锁；

```
269     do_unblock_all(&(pcb[i].wait_queue));
270
271     pcb[i].status = TASK_EXITED;
272     pcb_valid[i] = 0;
273 }
```

最后再将所有因为该进程被阻塞的进程唤醒，并将状态设置为 TASK_EXITED 状态，并将指示对应 pcb 时候有效的信号置为 0，完成资源的回收；

（3） wait 的实现：

```
326 void do_wait(pid_t pid)
327 {
328     int i;
329     for(i = 0; i < NUM_MAX_TASK; i++)
330     {
331         if(pcb[i].pid == pid)
332             break;
333     }
334
335     if(i == NUM_MAX_TASK)
336     {
337         my_printf("Waiting Task Does Not exist\n");
338         return ;
339     }
340
341     current_running->status = TASK_WAIT;
342     queue_push(&(pcb[i].wait_queue),current_running);
343
344     do_scheduler();
345 }
```

如上，找到要等待的进程，修改当前进程状态为 TASK_WAIT，然后将当前进程 Push 到对应进程的 pcb 中的 wait_queue 里，对等待中的进程不做其他处理，即不需要额外保存等待进程的 PCB，直接用原来的 PCB 就好；

## 3. 同步原语设计

（1）条件变量的实现：

```
 7   typedef struct condition
 8   {
 9       queue_t cond_queue;
10   } condition_t;
```

用一个队列来保存被阻塞的进程；

```
 4   void do_condition_init(condition_t *condition)
 5   {
 6       queue_init(&condition->cond_queue);
 7   }
```

条件变量的初始化，其实就是初始化一个队列；

```
 9   void do_condition_wait(mutex_lock_t *lock, condition_t *condition)
10   {
11       do_mutex_lock_release(lock);
12       do_block(&condition->cond_queue);
13       do_mutex_lock_acquire(lock);
14   }
```

将当前进程等待时，需要先让当前进程释放掌握的资源（LOCK），否则可能会引起死锁，然后将当前进程阻塞；当这个进程被唤醒后将会继续跟着 13 行执行，即先请求锁，若成功则可以继续执行，否则会被阻塞到对应的锁队列中；

```
16   void do_condition_signal(condition_t *condition)
17   {
18       do_unblock_one(&condition->cond_queue);
19   }
20
21   void do_condition_broadcast(condition_t *condition)
22   {
23       do_unblock_all(&condition->cond_queue);
24   }
```

Signal 和 broadcast 其实就是释放一个或者全部 wait_queue 里被阻塞的进程；

（3） 信号量的实现：

```
 7   typedef struct semaphore
 8   {
 9       int sem;
10       queue_t sem_queue;
11   } semaphore_t;
```

信号量的实现比条件变量多了一个指示资源数量的整型变量；

```
 5   void do_semaphore_init(semaphore_t *s, int val)
 6   {
 7       if(val >= 0)
 8       {
 9           s->sem = val;
10           queue_init(&(s->sem_queue));
11       }
12       else
13       {
14           my_printf("value error\n");
15       }
16   }
```

信号量的初始化即将信号量中对应的 sem 值修改为要初始化的目标值 value，然后初始化 sem 队列，由于信号量必须是非负值，所以，当判断出 val 小于 0 会报错；

```
18   void do_semaphore_up(semaphore_t *s)
19   {
20       if(!queue_is_empty(&s->sem_queue))
21       {
22           pcb_t * item = (pcb_t *)priority_queue_dequeue(&s->sem_queue);
23           item->status = TASK_READY;
24           priority_queue_push(&ready_queue, (void *)item);
25       }
26       else
27       {
28           s->sem++;
29       }
30   }
```

Up 操作，如果等待队列不空，则唤醒其中的一个进程，否则，将 sem 的值加 1；

```
32   void do_semaphore_down(semaphore_t *s)
33   {
34       if(s->sem)
35       {
36           s->sem--;
37       }
38       else
39       {
40           do_block(&s->sem_queue);
41       }
42   }
```

Down 操作，如果 sem 值大于 0 则将 sem 减 1，否则表示资源不够用，将当前进程阻塞到等待队列中；

（4） 屏障的实现：

```
 6   typedef struct barrier
 7   {
 8       uint32_t barrier_goal;
 9       uint32_t barrier_waiting_number;
10       queue_t  barrier_queue;
11   } barrier_t;
```

屏障结构包括一个设定的需要达到的目标进程数量 goal，以及已经在等待的进程数 waiting_number，和一个被屏障阻塞的队列；

```
4   void do_barrier_init(barrier_t *barrier, int goal)
5   {
6       barrier->barrier_goal = goal;
7       barrier->barrier_waiting_number = 0;
8       queue_init(&barrier->barrier_queue);
9   }
```

屏障的初始化即将屏障中对应的目标值设定为 goal，将正在等待的进程数初始化为 0，最后初始化等待队列；

```
11  void do_barrier_wait(barrier_t *barrier)
12  {
13      barrier->barrier_waiting_number++;
14      if(barrier->barrier_waiting_number < barrier->barrier_goal)
15      {
16          do_block(&barrier->barrier_queue);
17      }
18      else
19      {
20          do_unblock_all(&barrier->barrier_queue);
21
22          do_scheduler();
23      }
24  }
```

屏障的等待需要判断正在等待的进程数，如果加上当前进程（将 waiting_number 加 1），等待进程数达到了设定的目标等待进程数则将所有被阻塞的进程释放，否则将当前进程阻塞；

## 4. mailbox 设计

（1）数据结构即变量定义：

```
 6   #define MAX_MSG_SZ 100
 7
 8   typedef struct mailbox
 9   {
10       mutex_lock_t mutex;
11       condition_t empty;
12       condition_t full;
13
14       char name[20];
15       char msg[MAX_MSG_SZ];
16       int front;
17       int rear;
18       int left_space;
19       int user;
20
21   } mailbox_t;
```

结构体中需要包含一个互斥锁，用来保证同一时间段内只有一个进程能够访问共享数据；此外还需要两个条件变量 full 和 empty 来存放由于 buff 变得 full 和 empty 被阻塞的进程；另外还要有个信箱的名字，存储消息的 msg 数组，front 和 rear 指针用来指示消息的开头和结尾，left_space 指示剩余空间，user 指示正在使用当前信箱的人数；

```
11  void mbox_init()
12  {
13      int i;
14      mutex_lock_init(&mboxs_mutex);
15      for(i = 0; i < MAX_NUM_BOX; i++)
16      {
17          mutex_lock_init(&mboxs[i].mutex);
18          condition_init(&mboxs[i].empty);
19          condition_init(&mboxs[i].full);
20
21          mboxs[i].name[0] = '\0';
22          mboxs[i].front = 0;
23          mboxs[i].rear = 0;
24          mboxs[i].left_space = MAX_MSG_SZ;
25          mboxs[i].user = 0;
26      }
27  }
```

信箱的初始化即将对应的变量进行初始化即可；

```
30  mailbox_t *mbox_open(char *name)
31  {
32      int i;
33      mutex_lock_acquire(&mboxs_mutex);
34      for(i = 0; i < MAX_NUM_BOX; i++)
35      {
36          if(!strcmp(mboxs[i].name, name))
37          {
38              mboxs[i].user++;
39              mutex_lock_release(&mboxs_mutex);
40              return &mboxs[i];
41          }
42      }
```

```
44      for(i = 0; i < MAX_NUM_BOX; i++)
45      {
46          if(mboxs[i].name[0] == '\0')
47          {
48              strcpy(mboxs[i].name, name);
49              mboxs[i].user++;
50              mutex_lock_release(&mboxs_mutex);
51              return &mboxs[i];
52          }
53      }
54
55      printf("No more left mailboxs\n");
56  }
```

打开信箱由于需要修改信箱中的内容，所以需要在进行修改之前加锁，然后通过一个循环，比对信箱的名字，判断是否有已经存在的信箱，如果有则直接返回，否则，选择一个空的信箱创建，然后返回信箱；

```
59  void mbox_close(mailbox_t *mailbox)
60  {
61      mutex_lock_acquire(&mboxs_mutex);
62      --mailbox->user;
63      if(mailbox->user <= 0)
64      {
65          mutex_lock_init(&mailbox->mutex);
66          condition_init(&mailbox->empty);
67          condition_init(&mailbox->full);
68
69          mailbox->name[0] = '\0';
70          mailbox->front = 0;
71          mailbox->rear = 0;
72          mailbox->left_space = MAX_MSG_SZ;
73          mailbox->user = 0;
74      }
75      mutex_lock_release(&mboxs_mutex);
76  }
```

信箱的关闭，同样需要先加锁，然后判断如果当前正在使用信箱的人数为 0，即没有人在使用信箱，则将信箱初始化；

```
78  void mbox_send(mailbox_t *mailbox, void *msg, int msg_length)
79  {
80      mutex_lock_acquire(&mailbox->mutex);
81      while(mailbox->left_space < msg_length)
82          condition_wait(&mailbox->mutex, &mailbox->full);
83
84      if(MAX_MSG_SZ - mailbox->rear < msg_length)
85      {
86          memcpy((uint8_t *)(mailbox->msg + mailbox->rear), (uint8_t *)msg, MAX_MSG_SZ - mailbox->rear);
87          mailbox->rear = (msg_length + mailbox->rear) % MAX_MSG_SZ;
88          memcpy((uint8_t *)mailbox->msg, (uint8_t *)(msg + msg_length - mailbox->rear), mailbox->rear);
89      }
90      else
91      {
92          memcpy((uint8_t *)(mailbox->msg + mailbox->rear), (uint8_t *)msg, msg_length);
93          mailbox->rear += msg_length;
94      }
95
96      mailbox->left_space -= msg_length;
97      condition_broadcast(&mailbox->empty);
98      mutex_lock_release(&mailbox->mutex);
99  }
```

Send 函数，首先请求锁，然后判断剩余空间是否足够存放消息，如果够，则下一步，否则将当前进程阻塞住，等待剩余空间足够在继续执行；

在发送信息的存放信息阶段，运用循环队列存放消息，此时需要判断，如果尾指针到 buff 结尾的地方的空间大于消息长度，那么可以直接存放；否则，如果不够，那么需要分两次存储，一次填补 buff 的末尾，另一部分填补在 buff 开头的部分，并修改 rear 指针；

最后修改 left_space，将由于 buff 内容不够被阻塞的进程唤醒，释放锁退出；

```
101  void mbox_recv(mailbox_t *mailbox, void *msg, int msg_length)
102  {
103      mutex_lock_acquire(&mailbox->mutex);
104      while(MAX_MSG_SZ - mailbox->left_space < msg_length)
105          condition_wait(&mailbox->mutex, &mailbox->empty);
106
107      if(MAX_MSG_SZ - mailbox->front < msg_length)
108      {
109          memcpy((uint8_t *)msg, (uint8_t *)(mailbox->msg + mailbox->front), MAX_MSG_SZ - mailbox->front);
110          mailbox->front = (msg_length + mailbox->front) % MAX_MSG_SZ;
111          memcpy((uint8_t *)(msg + msg_length - mailbox->front), (uint8_t *)mailbox->msg, mailbox->front);
112      }
113      else
114      {
115          memcpy((uint8_t *)msg, (uint8_t *)(mailbox->msg + mailbox->front), msg_length);
116          mailbox->front += msg_length;
117      }
118
119      mailbox->left_space += msg_length;
120      condition_broadcast(&mailbox->full);
121      mutex_lock_release(&mailbox->mutex);
122  }
```

Recv 函数与 send 函数类似，首先请求锁，然后判断 buff 中的内容是否足够取出，如果足够则进行下一步，否则阻塞到 empty 队列中；

然后就是消息的取出操作，这里和消息的存放类似，需要分两种情况取出，不再赘述；

最后修改 lef_space，唤醒由于 buff 变得 full 而被阻塞住的进程，释放锁退出；

（2）在我的 mailbox 中，我采用了互斥锁和条件变量两种同步原语实现，类似于理论课上介绍的管程模型，并且支持多个生产者和多个消费者的情况，如何处理，详见上文；

## 5. 关键函数功能

（1） 在 mailbox 的设计中，发送和接收的判断设计：

```
103      mutex_lock_acquire(&mailbox->mutex);
104      while(MAX_MSG_SZ - mailbox->left_space < msg_length)
105          condition_wait(&mailbox->mutex, &mailbox->empty);
```

利用 while 循环，每次唤醒之后重新判断条件是否符合，如果符合则继续执行否则再次进入阻塞状态；

（2）shell 的设计里对命令的解析：

```
140  void analysis_cmd(char cmd[20], int *argc, char argv[10][20])
141  {
142      int i = 0;
143      int j = 0;
144      int k = 0;
145      while(cmd[i] != ' ' && cmd[i] != '\0')
146          i++;
147      if(cmd[i] == ' ')
148      {
149          cmd[i] = '\0';
150          i++;
151      }
152      *argc = 1;
```

```
154     while(cmd[i])
155     {
156         k = 0;
157         do{
158             argv[j][k++] = cmd[i++];
159         }while(cmd[i] != ' ');
160         argv[j][k] = '\0';
161         (*argc)++;
162         i++;
163         j++;
164     }
165 }
```

通过 cmd 里的空格符将命令分为主命令和参数分别存放到 cmd 和 argv 中，并计算出 cmd 中的参数个数，存放到 argc 中；