Project 3 Interactive OS and Process Management 设计文档

中国科学院大学

蔡润泽

2019 年 11 月 11 日

1. Shell 设计

（1） shell 实现过程中遇到的问题和得到的经验

shell 在实现退格时，发现在 QEMU 里无法正常显示退格，在板上却可以正常显示退格。然后我发现 screen.c 里关于退格字符的 ascii 码里只有 8 而没有 0x7f。在此处加上 0x7f 后，退格可以正常显示。

2. spawn, 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**

该设计在 kill 一个进程时，会释放该进程所持有的锁。锁的数据结构中包括一个阻塞队列，用来存试图获得锁却未能获得于是被阻塞的任务。在 kill 释放锁时需要将相关阻塞的任务释放。对于同步原语的处理，kill 并没有直接影响同步原语，但相关对于锁和阻塞队列的操作会间接影响同步原语。

（2） wait 实现时，等待的进程的 PCB 用什么结构保存？

每个 PCB 会有一个 wait_queue 队列，用来存储因为等待该 PCB 结束而被阻塞的队列。在实现 wait 时，只需将当前 PCB 存到对应等待 PID 号码的 PCB 中的 wait_queue 即可。

（3） 设计或实现过程中遇到的问题和得到的经验

在实现 PCB 的 spawn、kill 和 exit 机制时，该设计利用了回收 PCB 块空间的办法。但是这样的设计会导致 PID 和 PCB 原本的数组顺序不相匹配。如果单纯的利用 PCB[pid] 来查找 PCB 块的话，就会导致错误。解决的方式是顺序搜索 PCB，直到找到一个 PCB->pid =pid 的 PCB 块。

3. 同步原语设计

（1） 条件变量、信号量和屏障实现的各自数据结构的包含内容

条件变量：包括一个记录正在等待的进程数目的整型变量以及一个等待队列。

```
typedef struct condition
{
    int num_waiting;
    queue_t wait_queue;
} condition_t;
```

信号量：包括一个记录信号量大小的整型变量和一个阻塞队列。

```
typedef struct semaphore
{
    int sem;
    queue_t block_queue;
} semaphore_t;
```

屏障：包括一个记录要求达到屏障进程数目的整型变量、已经达到屏障进程数目的整型变量以及一个阻塞队列。

```
typedef struct barrier
{
    uint32_t barrier_num;
    uint32_t waiting_num;
    queue_t block_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

```
typedef struct mailbox
{
    char name[30];
    uint8_t msg[MSG_MAX_SIZE];
    int msg_head, msg_tail;
    int used_size;
    int cited;
    condition_t full;
    condition_t empty;
    mutex_lock_t mutex;
} mailbox_t;
```

该数据结构主要包括一个信息名字符串、信箱内容字符串、信箱内容的头尾指针、已使用的大小、被引用的次数、两个和空、满相关的条件变量，以及一把控制互斥访问的互斥锁。

（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？

在处理生产者消费者模型时，该设计采用的是将条件变量作为同步原语的方式。

生产者为写者，当信箱空间不足时，其会在 empty 条件变量上等待，直到有足够的写空间。当写完毕时，生产者会唤醒阻塞在 full 条件变量上的消费者。

而消费者为读者，当信箱可读内容不足时，其会在 full 条件变量上等待，直到有足够的读内容。当读完毕时，消费者会唤醒阻塞在 empty 条件变量上的生产者。

该设计支持多 producer 或多 consumer。通过锁的实现，可以保证临界区只有一个进程进行访问。同时唤醒进程时，该设计采用的是 broadcast 形式，即唤醒所有相应条件变量的阻塞进程。

5. 关键函数功能

在释放锁时，原来的设计会导致直接运行时，两把因为抢锁失败的进程重新获得锁时，永远只有同一把先抢到释放的锁；而设置断点后的顺序执行，两个进程则都有机会能拿到锁，为了解决这个玄学 bug，本设计在释放锁的最后一步加上了 do_scheduler()，通过这一修改，两个进程就可以均有机会拿到刚释放的锁。实现代码如下：

```
1.  void do_mutex_lock_release(mutex_lock_t *lock)
2.  {
3.      int i;
4.      for(i = 0; i <= current_running->lock_top; i++){
5.          if(current_running->lock[i] == lock){
6.              int j = i+1;
7.              while(j <= current_running->lock_top){
8.                  current_running->lock[j-1] = current_running->lock[j];
9.                  j++;
10.             }
11.             --current_running->lock_top;
12.             break;
13.         }
14.     }
15.
16.     if(!queue_is_empty(&lock->block_queue)){
17.         do_unblock_one(&lock->block_queue);
18.     }
19.     lock->status=UNLOCKED;
20.     do_scheduler();
21.
22. }
```

参考文献

[1] 理论课同步原语相关 PPT
[2] 实验课 project_3_guid_book_MIPS
[3] 《现代操作系统》进程与线程章节