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

# 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。

# 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。

1. **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。

1. **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。

# 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。

1. **typedef** **struct** condition
2. {
3. **int** num\_waiting;
4. queue\_t wait\_queue;
5. } 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。

1. **typedef** **struct** semaphore
2. {
3. **int** sem;
4. queue\_t block\_queue;
5. } 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。

1. **typedef** **struct** barrier
2. {
3. uint32\_t barrier\_num;
4. uint32\_t waiting\_num;
5. queue\_t block\_queue;
6. } barrier\_t;

# mailbox设计

* 1. **mailbox的数据结构以及主要成员变量的含义**

1. **typedef** **struct** mailbox
2. {
3. **char** name[30];
4. uint8\_t msg[MSG\_MAX\_SIZE];
5. **int** msg\_head, msg\_tail;
6. **int** used\_size;
7. **int** cited;
8. condition\_t full;
9. condition\_t empty;
10. mutex\_lock\_t mutex;
11. } 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。

* 1. **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。

# 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. }
16. **if**(!queue\_is\_empty(&lock->block\_queue)){
17. do\_unblock\_one(&lock->block\_queue);
18. }
19. lock->status=UNLOCKED;
20. do\_scheduler();
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