Project 3 Interactive OS and Process Management 设计文档

中国科学院大学
[薛峰]
[2018/11/14]

1. Shell 设计

（1）shell 实现过程中遇到的问题和得到的经验

1、一开始不知道怎么从串口输入字符，之后一步一步试出来如何使用串口。

2、刚开始不知道如何从解析一个字符串，即不知道一个字符串什么时候结束。后来想到当检测到回车的时候，便可知道用户当前命令输入结束。

3、将 ASCII 码中回车和换行符弄混，导致输入命令之后无法解析。

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。

如何生成进程 ID：

有一个全局变量 pid，初始化为 1（main 函数的 pid 为 0），没每当调用 do_spawn 时，就将 pid 赋给这个 task，随后将 pid++。

Spawn 处理过程：

1、寻找空闲的 pcb 块，其中判断标准为检查相应的 pcb 的 status 域是否为 TASK_EXITED，若 status 域为 TASK_EXITED，则该 pcb 域是空闲的；否则不空闲。

2、为这个空闲的 pcb 块进行初始化。即寄存器初始化，队列初始化，状态初始化等。

3、将该 pcb 块 push 进准备队列。

（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。

对于要 kill 的进程，需要释放进程所占有的锁，将锁的状态更改成 unblock，并且需要将该进程从 ready_queue，block_queue 等队列中拿出。

并且如果该进程持有 mailbox，需要放弃该 mailbox。

（3）wait 的处理过程。

Spawn 处理过程：

1、根据 pid 找到对应的 pcb；

2、将 current_running 的状态改为 TASK_BLOCKED；

3、将 current_running 的 pcb 放到第一步中找到的 pcb 的 wait_queue 中；

4、进程切换

（4）设计或实现过程中遇到的问题和得到的经验。

最大的困难在于 kill 的实现，需要释放很多东西，并且还需要找到该 pcb 所在的队列，因此需要对 pcb 添加一些域，这样使得其他操作（例如申请锁，需要在 pcb 中指明该进程持有哪些锁）变得更加复杂，需要考虑更多的东西。

并且在实现的过程中犯了些低级的错误，比如在核心态调用了用户态的函数，函数生命的大括号写成了（），并且这些错误并不难找到，尤其是括号那个低级 bug 浪费了很长时间。

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：

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

block_queue 的作用：当对某条件变量进行 wait 操作时，就将该任务 push 进该条件变量的 block_queue。

do_condition_wait(mutex_lock_t *lock, condition_t *condition)：current_running 把自己 push 进 condition 的 block_queue，等待被唤醒。入队列前要先释放锁 lock，防止发生死锁。并且唤醒之后需要重新申请锁。

do_condition_signal(condition_t *condition)：将 condition 中的一个任务放入 ready_queue。

do_condition_broadcast(condition_t *condition)：将 condition 中所有任务放入 ready_queue。

信号量：

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

其中，value 表示资源的个数，block_queue 为信号量的锁，当 up 或 down 操作不成功时，需要将当前的任务 push 进这个队列。

do_semaphore_up(semaphore_t *s)：将 value+1，并释放 semaphore 的 wait_queue 的一个 task。

do_semaphore_down(semaphore_t *s)：如果 value=0 就把当前的任务放入 s 的 block_queue 中，知道 value>0，随后将 value-1。

屏障：

```
typedef struct barrier
{
  int total_num;
  int current_num;

  queue_t block_queue;
} barrier_t;
```

其中，total_num 表示总共需要的进程数量，current_num 表示当前到达屏障的进程数，block_queue 用于存放阻塞在该屏障的进程。

do_barrier_wait(barrier_t *barrier)：如果当前等待的进程等于 total_num，则释放所有 block_queue 中的进程；否则，将 current_running 阻塞。

（2）设计或实现过程中遇到的问题和得到的经验

这一实验最大的收获在于对课上学习的同步原语概念的理解更加深刻了。当时上课的时候只是学了概念，知道他们可以实现数据的互斥访问，但是具体是如何实现的却不太了解。

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[MAX_NAME_LENGTH];
  char buffer [MAX_BUFFER_LENGTH];
  int user_num;

  int num_item;
  int read_index;
  int write_index;

  condition_t not_empty;
  condition_t not_full;
} mailbox_t;
```

Name：该 mailbox 的名字；
Buffer：mailbox 中存放的数据；
User_num：目前正在使用这个 mailbox 的进程数，用于判断该 mailbox 是否需要被关闭；
Num_item：mailbox 的数据个数；
Read_index：读 buffer 时的指针；
Write_index：写 buffer 时的指针；
Not_empty：条件变量，用于等待 buffer 不空；
Not_full：条件变量，用于等待 buffer 不满。

（2）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，

producer 要往里面写，如果缓冲区满了 producer 就要阻塞到 not_full 中的队列中，直到被 consumer 唤醒；同理，consumer 要往里写，如果缓冲区空了，consumer 就要阻塞到 not_empty 的阻塞队列中，知道被 consumer 唤醒。

（3）设计或实现过程中遇到的问题和得到的经验

该部分的难度主要在 mailbox 结构体的设计上，需要简洁地定义 mailbox 并且能够支持 mailbox 的所有功能。另外该部分的实现是我更加理解了条件变量，感觉该部分是之前实验的综合。

5. 关键函数功能

1、do_wait 函数

功能：该函数将 current_running 的 pcb 阻塞，阻塞到对应 pcb 的队列当中，知道要等待的队列死亡才能被释放。

实现过程：为实现该函数需要在 pcb 结构体中添加一个 wait 队列，用于存放等待该进程的 pcb，在该进程死亡时（主动 exit 或者被 kill），需要释放该 wait 队列。该函数首先根据 pid 找到对应的 pcb，将 status 改为 BLOCKED，随后将该 pcb 放入对应的 block_queue 中，等待被唤醒，最后进行进程切换。

```
void do_wait(pid_t pid)
{
  int i;

  for(i = 0; i < NUM_MAX_TASK; i++)
  {
    if(pcb[i].pid == pid && pcb[i].status != TASK_EXITED)
    {
      current_running->status = TASK_BLOCKED;
      queue_push(&pcb[i].wait_queue, current_running);
      do_scheduler();
      return;
    }
  }
}
```

2、do_exit 函数

功能：挂起当前 task，等待一个 task，直到被等待的 task 被 kill，这个挂起的 task 被唤醒

实现过程：需要释放该进程占据的所有资源：wait 队列中的 task，占据的锁，和栈。对于栈的释放并不需要修改栈的数据，因为在为 pcb 分配栈的时候，每个 pcb 对应一个栈空间，因此根据 pcb 的状态便可判断该 pcb 是否可用，即该栈空间是否可用。

```
void do_exit()
{
  current_running->status = TASK_EXITED;

  // relaese it's wait queue
  do_unblock_all(&current_running->wait_queue);

  // relaese all locks the process has
  while(!queue_is_empty(&current_running->lock_queue))
  {
    mutex_lock_t* lock = ( (mutex_lock_t*)queue_dequeue(&current_running->lock_queue) );
    pcb_t *pcb;

    do_unblock_all(&lock->block_queue);
    lock->status = UNLOCKED;
  }

  do_scheduler();
}
```

3、do_kill 函数

功能：杀死一个 task

实现过程：与 do_exit 类似，不过需要注意的是，杀死的进程并不一定是 current_running，因此需要将该进程从某个队列中拿出，例如 ready_queue、sleep_queue 或 block_queue 中。

4、void condition_wait(lock_t * m, condition_t * c)函数

功能：current_running 把自己放入 condition 的 wait_queue 里等待 signal 唤醒它。并且入 wait 队前要先释放 current_running 的锁，防止发生死锁。

```
void do_condition_wait(mutex_lock_t *lock, condition_t *condition)
{
  do_mutex_lock_release(lock);
  do_block( &(condition->block_queue) );

  do_mutex_lock_acquire(lock);
}
```

5、void condition_signal(condition_t * c)函数

功能：唤醒 condition 中的一个 task 放入 ready。

```
void do_condition_signal(condition_t *condition)
{
  do_unblock_one( &(condition->block_queue) );
}
```

6、do_mbox_send 函数：

功能：向邮箱中写 message。

```
void do_mbox_send(mailbox_t *mailbox, void *msg, int msg_length)
{
  int i;
  do_mutex_lock_acquire(&mutex);

  for(i = 0; i < msg_length; i++)
  {
    while(mailbox->num_item == MAX_BUFFER_LENGTH)
      do_condition_wait(&mutex, &(mailbox->not_full));

    mailbox->buffer[mailbox->write_index++] = *((char *)msg + i);
    mailbox->write_index %= MAX_BUFFER_LENGTH;
    mailbox->num_item++;
  }

  do_mutex_lock_release(&mutex);

  // do_condition_signal(&(mailbox->not_empty));
  do_condition_broadcast(&(mailbox->not_empty));

}
```