Project4 Synchronization Primitives and IPC 设计文档

中国科学院大学
[王苑铮]
[2017.11.29]

1. do_spawn, do_kill 和 do_wait 设计

（1） do_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。

Do_spawn 的过程：

1.关中断，

2.查找“ramdisk”找到指定 filename 的 task。如果没有这个名字的文件，spawn 直接 return。

3.查找 pcb 数组，找到一个 status == EXIT 的 pcb 块，若找不到则 spawn 直接返回。

4.++pid 产生 task 的 pid

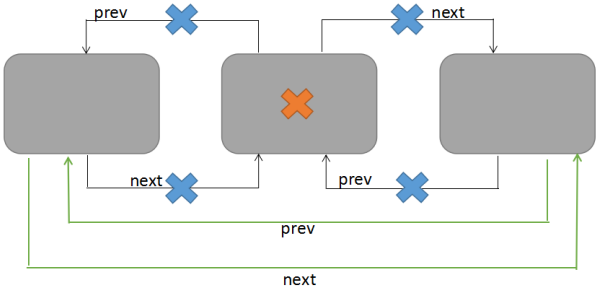
5.用 pid 和从 ramdisk 中找到的 task 以及找到的空闲 pcb 块，initialize 为这个 task 的 pcb

6.开中断，spawn 返回 task 的 pid

（2） do_kill 的处理过程。如果有做 bonus，请在此说明在 kill task 时如何处理锁

do_kill 过程：

1. 检查 pcb 数组找到 kill 的 pid 对应的 pcb 块。如果没有这个 pid 对应的块，或者找到的这个块已经 exit，则 do_kill 直接 return
2. 若找到 pcb 后发现是 current_running,则将其 pcb 状态标为 exit，释放它的 wait 队列中的 task（状态标为 ready，入 ready 队列），把这个 task 从它所在的 queue 中清除，并释放锁（把锁的 wait queue 中的 task 释放掉，与释放 task 的 wait 队列类似），调用 scheduler 调度出下一个 task
3. 若找到 pcb 发现不是 current_running,被 kill 的 task 状态标为 exit,释放 task 的 wait 队列,释放锁。若被 kill 的 task 不是 current_running，则其一定在且仅在一个队列中（ready，sleep，某个 task 的 wait_qeueu，某个同步元语的 wit_queue）。队列是双向循环链表，所以只需按如下修改指针，就可以把这个 task 从它所在的队列中去掉

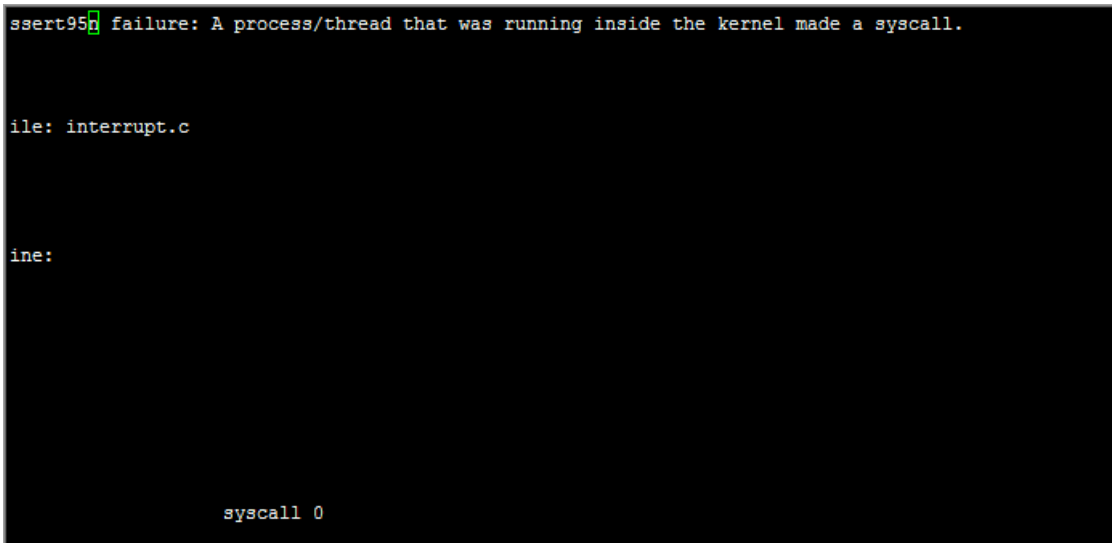


（3） do_wait 的处理过程

1. 检查 pcb 数组。如果 wait 的 task 不存在或者 wait 的是自己，则 do_wait 直接 return
2. Current_running->status = BLOCK，入对应的要 wait 的 task 的 wait_queue(这是 pcb 结构体的一部分。当 wait 的 task 被 kill 时，wait 队列中所有的 task 便被释放回到 ready)
3. 调 scheduler 换下一个 task

（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）

测试 spawn 时，出了如下这个错误。我想在 spawn 中打印一些信息，但是调用 printf 又会出 lpppppppppppp，于是这个 spawn 里的行为我就没法了解了。后来小伙伴帮忙看了一下，发现是我的 ramdisk 函数用的不对。我用的是 ramdisk_find, 而成功的小伙伴用的是 ramdisk_find_File（都是已经提供的函数）。换成这个函数就可以了。这个 bug 卡了很久。



2. 同步原语设计

（1） 条件变量、信号量和屏障的含义，及其所实现的各自数据结构的包含内容

1.Condition：

```
/* TODO */
typedef struct condition{
    node_t wait_queue;
} condition_t;
```

void condition_wait(lock_t * m, condition_t * c)：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 condition_signal(condition_t * c);唤醒 condition 中的一个 task 放入 ready

void condition_broadcast(condition_t * c);把 condition 中所有 task 都唤醒放入 ready

2.semaphore：

```
typedef struct semaphore{
    int value;
    node_t wait_queue;
} semaphore_t;
```

void semaphore_down(semaphore_t * s); 将 semaphore 的 value-1，如果 value<0 就把当前执行 down 的 task 放入 semaphore 的 wait_queu 等待

void semaphore_up(semaphore_t * s);将 semaphore 的 value+1，并释放 semaphore 的 wait_queue 的一个 task。

注：我的实现中，semaphore 的 value 是可以为负数的。Down 第一步是—value，up 第一步是++value。只要有数量相同的 up 和 down，这种设计一样可以保证最后 value 回到初始的值

3.barrier

```
/* TODO */
typedef struct barrier{
    int max_members;
    int current_members;
    node_t wait_queue;
} barrier_t;
```

void barrier_wait(barrier_t * b);如果进入 barrier 的 task 数量小于预设的 max_members 数量，则 barrier 中的所有 task 都等待

如果 barrier 中 task 的数量与 max_member 相同，则释放 barrier 中所有 task 入 ready

（2） 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）

我的 barrier 中用到的这个 unblock_all（）是我自己定义的，开始是在 scheduler.c 里实现的，并将其定义写在 scheduler.h 里让 sync.c include”scheduler.h”,但是测试时 barrier 只能加载出两个 task，另外两个 task 永远加载不出来，barrier 永远在等。换用 testall 用例测试，其他元语都能执行，只有 barrier 无法执行

后来我把 unblock_all（）从 scheduler.c 挪到 sync.c 里，就可以运行了。

这个错误让我很抓狂，完全意料不到。

```
// TODO: Block until all n threads have called barrier_wait
void barrier_wait(barrier_t * b){
    enter_critical();
        ++b->current_members;
        if(b->current_members == b->max_members){
            unblock_all(&b->wait_queue);
            b->current_members = 0;
        }
        else{
            block(&b->wait_queue);
        }
    leave_critical();
}
```

3. mailbox 设计

（1） mailbox 的数据结构以及主要成员变量的含义

```
typedef struct
{
  /* TODO */
  char message[MAX_MESSAGE_LENGTH];
} Message;

typedef struct
{
  /* TODO */
    semaphore_t used;
    semaphore_t not_used;
    semaphore_t mutex;

    char name[MBOX_NAME_LENGTH+1];
    Message box[MAX_MBOX_LENGTH];

    int user_number;

    //node_t producer_wait_queue;
    //node_t costumer_wait_queue;

    int write_index;
    int read_index;

} MessageBox;
```

三个 semaphore：
used：已经使用的消息槽数量。
Not_used：空闲的消息槽数量。
Mutex：对消息槽区域的互斥访问

Name：mailbox 的名字
Box：里面存储了 message
User_member：使用这个 mailbox 的 task 数量
Write_index：生产者可以写到 box 的这个角标往后的位置
Read_index：消费者可以读 box 这个角标往后的位置

因为 semaphore 中本身就有 wait queue，所以不需给 mailbox 额外的 wait queue

（2） producer-consumer 问题是指什么？你在 mailbox 设计中如何处理该问题？
producer-consumer 问题：有一个有限的缓冲区，一直只能允许一个 task 进行读或者写操作。
producer 要往里面写，如果缓冲区满了 producer 就要先阻塞，直到不满才能继

续写。Consumer 要从里面读，如果缓冲区空了就要先阻塞，直到不空才能继续读。

```
void do_mbox_recv(mbox_t mbox, void *msg, int nbytes)
{
  (void)mbox;
  (void)msg;
  (void)nbytes;
  /* TODO */

  semaphore_down(&MessageBoxen[mbox].used);
  semaphore_down(&MessageBoxen[mbox].mutex);

  int read_index = MessageBoxen[mbox].read_index;
  int i;
  for(i = 0; i < nbytes && i < MAX_MESSAGE_LENGTH; i++)
      *((char *)msg + i) = MessageBoxen[mbox].box[read_index].message[i];
  MessageBoxen[mbox].read_index = (MessageBoxen[mbox].read_index + 1) % MAX_MBOX_LENGTH;

  semaphore_up(&MessageBoxen[mbox].mutex);
  semaphore_up(&MessageBoxen[mbox].not_used);

}
```

```
void do_mbox_send(mbox_t mbox, void *msg, int nbytes)
{
  (void)mbox;
  (void)msg;
  (void)nbytes;

  /* TODO */
  semaphore_down(&MessageBoxen[mbox].not_used);
  semaphore_down(&MessageBoxen[mbox].mutex);

  int write_index = MessageBoxen[mbox].write_index;
  int i;
  for(i = 0; i < nbytes && i < MAX_MESSAGE_LENGTH; i++)
      MessageBoxen[mbox].box[write_index].message[i] = *((char *)msg + i);
  MessageBoxen[mbox].write_index = (MessageBoxen[mbox].write_index + 1) % MAX_MBOX_LENGTH;

  semaphore_up(&MessageBoxen[mbox].mutex);
  semaphore_up(&MessageBoxen[mbox].used);

}
```

以 send 为例，如果 mailbox 已满（not_used==0），down 操作导致 producer 被挂起，直到 consumer 读完后 up not_used 将 producer 唤醒。

Producer 被唤醒或者本身邮箱就没满，但是此时正有其他 task 在操作缓冲区（它们先 down 了 mutex），则 producer down mutex 会被挂起。直到另一个 task 结束使用缓冲区，对 mutex 发出 up，producer 可以执行缓冲区。

Producer 操作完缓冲区，对 used 发出 up，这时如果有因为 mailbox 空而被阻塞的 consumer，就会被唤醒

（3）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）

4. 关键函数功能

do_spawn：从 ramdisk 里读出一个“文件”作为 task

do_kill：杀死一个 task

do_wait：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

void condition_wait(lock_t * m, condition_t * c)：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 condition_signal(condition_t * c);唤醒 condition 中的一个 task 放入 ready

void condition_broadcast(condition_t * c);把 condition 中所有 task 都唤醒放入 ready

void semaphore_down(semaphore_t * s); 将 semaphore 的 value-1，如果 value<0 就把当前执行 down 的 task 放入 semaphore 的 wait_queu 等待

void semaphore_up(semaphore_t * s);将 semaphore 的 value+1，并释放 semaphore 的 wait_queue 的一个 task。

void barrier_wait(barrier_t * b);如果进入 barrier 的 task 数量小于预设的 max_members 数量，则 barrier 中的所有 task 都等待

mbox_t do_mbox_open(const char *name); 打开一个邮箱

void do_mbox_close(mbox_t mbox); 关闭一个邮箱

void do_mbox_send(mbox_t mbox, void *msg, int nbytes); 向邮箱中写 message

void do_mbox_recv(mbox_t mbox, void *msg, int nbytes); 从邮箱中读 message

参考文献

[1] [单击此处键入参考文献内容]