Project4 Synchronization Primitives and IPC 设计文档

中国科学院大学
付琳晴
2017.11.29

1. do_spawn, do_kill 和 do_wait 设计

1.1 do_spawn 处理过程

Do_spawn 事实上和 scheduler 函数的作用类似，都是启动生成一个任务。与 scheduler 函数的区别是，scheduler 函数是从 ready 队列中取出一个任务，这个任务的 pcb 已经做了初始化。而 do_spawn 函数是每次从 file 队列中找到需要的文件，初始化其 pcb，并放入 ready 队列。利用 ramdisk_find_File 函数从 file 队列找到文件的 entry_point 和 task_type 信息。如图 1.

```
File *f;
f = ramdisk_find_File(filename);
ti.entry_point = f->process;
if (ti.entry_point == 0)
  return -1;
ti.task_type = f->task_type;
```

图 1. 利用 randisk_find_File 函数找到文件类型信息

任务的 PID 的赋值是通过定义一个全局变量 next_pid，每次初始化一个 pcb 就将 next_pid 加 1 并赋值给 pcb 的 pid 域。注意，next_pid 作为一个全局变量，它的加一操作需要在临界区内进行。

Pcb 的初始化通过 initialize_pcb 函数进行。对 initialize_pcb 函数的具体说明见下文。

1.2 do_kill 处理过程

Do_kill 分为两种情况，kill 自己和 kill 其他进程，但其实这两种没什么太大的区别，唯一的区别是如果 kill 了自己需要重新调度一个新的任务。Kill 一个任务时，要考虑到这个任务持有锁，且有其他进程在 wait 该进程，因此要 unblock 全部等这个任务的任务。我利用了提供的代码中 pcb 结构体中 wait_queue 域，用来保存等待该任务的其他任务，因此 kill 一个任务时，将其 wait_queue 域中的所有任务释放，都加入到 ready_queue 中。

此次实验做了 bonus，即考虑若 kill 一个任务，该任务持有锁的情况。需要将该任务持有的锁释放。因此在 pcb 结构体中新增加了一个域：lock 数组，数组类型为 bool_t，每一位对应一种锁，lock[i] = TRUE 表示该任务持有第 i 个锁，lock[i] = FALSE 表示该任务不持有第 i 个锁，并且在内核定义了一个锁数组，数组每一个元素是一种锁，其下标对应 lock 域的每一个下标。此实验中，我进行了简化，假设一共只有 32 种锁，那么数组大小是 32。每当 kill 一个任务时，在内核态调用 release_lock_all 函数，这个函数将扫描被 kill 任务的 pcb 的 lock 域，扫描一遍发现有 lock[i] = TRUE 的地方，即表示拥有第 i 种锁，则释放该锁。

Bonus 需要实现在进程上，进程是不能直接接触到内核函数的，因此需要利用 SYSCALL 接口，在 syslib.c 文件中，仿照已提供的代码，我新定义了三种进程 lock 系列函数，用来为

进程提供访问内核态 lock 系列函数的接口。如图 2. 相应的 syslib.h 文件中需要进行三个函数的声明。

```
91  int mylock_init(){
92    return invoke_syscall(SYSCALL_LOCK_INIT, IGNORE, IGNORE, IGNORE);
93  }
94
95  int mylock_acq(int i){
96    return invoke_syscall(SYSCALL_LOCK_ACQUIRE, i, IGNORE, IGNORE);
97  }
98
99  int mylock_real(int i){
100   return invoke_syscall(SYSCALL_LOCK_RELEASE, i, IGNORE, IGNORE);
101 }
```

图 2. 新定义的用户态函数

在 kernel.c 文件的 init_syscalls 函数中，相应的定义了三种 SYSCALL，提供系统调用接口，如图 3.

```
syscall[SYSCALL_LOCK_INIT] =  &my_lock_init;
syscall[SYSCALL_LOCK_ACQUIRE] =  &my_lock_acquire;
syscall[SYSCALL_LOCK_RELEASE] =  &my_lock_release;
```

图 3. 新定义 SYSCALL

最后在内核定义三个 lock 函数，用来在内核实现 32 种锁的 initial、acquire 和 release. 用户调用这些函数时，只需要传参一个 int 类型，对应要得到/释放的锁的编号。内核得到该参数后，在锁的数组中找到该锁，进行相应操作。内核中使用的函数其实与已存在的 lock_acquire 函数类似，因此我就直接利用了函数调用。如图 4.

```
27  int my_lock_init(){
28    int i;
29    for (i = 0; i < 32; i ++){
30      lock_init(&kernel_lock[i]);
31    }
32    return 0;
33  }
34
35  int my_lock_acquire(int i){
36    if (i>=0 && i<= 32){
37      lock_acquire(&kernel_lock[i]);
38      current_running->lock[i] = TRUE;
39    }
40    return 0;
41  }
42
43  int my_lock_release(int i){
44    if (i >= 0 && i<= 32){
45      lock_release(&kernel_lock[i]);
46      current_running->lock[i] = FALSE;
47    }
48    return 0;
49  }
```

图 4. My_lock 系列内核函数

由此，便可以实现进程通过系统调用调用内核函数实现在 kill 任务时，释放该任务持有的锁。为了验证程序的正确性，我写了两个简单的进程进行测试。process1 先持有一个锁 l，然后 spawn 第二个 process，然后 yield。第二个 process 将 kill 第一个进程，并尝试 acquire 锁 l，若 kill 第一个进程时能成功释放锁，则将 acquire 成功，打印出成功字样。该程序经测试验证正确。

1.3 do_wait 处理过程

Do_wait 函数较简单，首先一个任务不能等待其自身，这要作为一个判断条件。然后根据传入的 pid 在 pcb 结构体中寻找到对应的 pcb，将当前进程放入到那个任务的等待队列中，然后调用 scheduler_entry 调度下一个任务。利用 pcb 结构体中的 wait_queue 域，我们很容易做到这一点。

1.4 遇到的问题和解决

（1）栈上界太小导致栈溢出

图 5. 问题 1

解决过程：更改了栈的上界为 0xa2000000 即可，不过后来上课听老师说可以回收栈，但是没有更改，及时回收栈应该是个更好的方案。

（2）初始化 PCB 时把所有任务的 task_type 全部设成 PROCESS，导致 assert(nested_count) 报错。

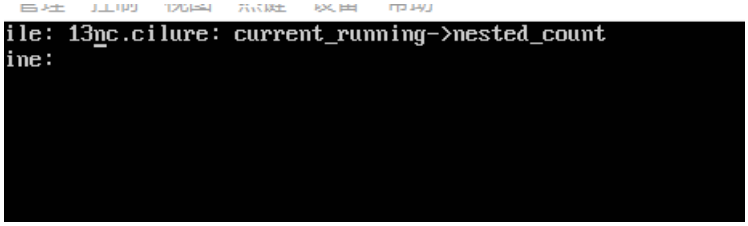

图 6. 问题 2

解决过程：经室友帮助定位到了 do_spawn 中，发现应该通过 ramdisk_find_File 返回文件的类型和 entry_point，但我用成了 ramdisk_find 函数，导致 task_type 赋值出错。

（3）自己写程序测试 bonus 内容时出现 EPPPPPPPP

解决过程：通过定位发现能够正确的执行 process1，但是在 process2 做 kill 操作时出现 EPPPPP，因此怀疑是 kill 函数里 leave_critical 多加了一个，查看 kernel.c 文件中 do_kill 函数，查看我在临界区做的操作，发现自己写的释放所有锁的函数，其实已经在临界区内，结果这个函数内部调用了 lock_release，这个函数是提供的，因此我没有注意到里面又一次 enter_critical，因此出错。发现错误之后，我将释放所有锁的函数放在了 enter_critical 之外，即运行正确。

2. 同步原语设计

请至少包含以下内容

（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

（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）

2.1 条件变量

条件变量结构体如图 7. 它的使用其实是与锁一起进行，当前进程企图 acquire 锁时，将被放入条件变量等待队列，等待锁被释放掉之后，且该任务又被调度的时候，将持有该锁。

```
22  typedef struct condition{
23      node_t wait_queue;
24  }condition_t;
25
```

图 7. 条件变量结构体

condition_signal 函数将条件变量等待队列的队头 Pop 出来，压入 ready_queue，将进程状态设为 ready；condition_broadcast 将条件变量等待队列的所有元素都 Pop 出来，压入 ready_queue.

2.2 信号量

信号量结构体如图 8. 它与条件变量相比多了一个域 value。信号量经常用于生产者-消费者问题，初始化时将 valus 赋一个正值，当一个消费者要使用时，value 减一，生产者生产一个产品，value 加一。semaphore_up 和 semaphore_down 函数类似于操作系统理论课讲的 P 函数和 V 函数，semaphore_up 判断信号量是否是 0，如果是 0，且等待队列不为空，将信号量等待队列队头 Pop 出来，压入 ready_queue, 将进程状态设为 ready, 否则将信号量加 1. semaphore_down：判断信号量是否为 0，如果是，将当前进程压入条件变量等待队列，将进程状态设为 blocked。否则信号量减一。通过使用 enter_critical 函数保证对 value 的操作是原子操作。

```
27  typedef struct semaphore{
28      int value;
29      node_t wait_queue;
30  }semaphore_t;
31
```

图 8. 信号量结构体

2.3 屏障

barrier 结构体中存一个剩余任务数（整型），需要等待的总的任务数（整型）和一个等待队列，结构体如图 9. Barrier 的作用是使多个任务到达同一个状态点，再同时继续执行，若在其中，有任务还没有到该状态，其他任务将等待。具体实现为 barrier_wait 函数，判断剩余任务数是否为 0，如果是，将等待队列中的所有元素 pop 出来，并压入 ready_queue，然后将剩余任务数置为 n，否则将 barrier 的剩余任务数减 1，当前进程压入等待队列。

```
/* TODO */
typedef struct barrier{
    int size;
    int num_barrier;
    node_t wait_queue;
}barrier_t;
```

图 9. 屏障结构体

以上函数设计时，为了避免中断的影响，都需要在函数开始时关中断，结束时开中断。

3. mailbox 设计

3.1 mailbox

Mailbox 是一块信息数组，用来表示有界缓冲区，为了能正确实现多个生产者和消费者对其的同时读写，其数据结构里域比较多。结构体定义如图 10.

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

typedef struct
{
  /* TODO */
  char name[MBOX_NAME_LENGTH];
  Message box[MAX_MBOX_LENGTH];
  int use_num;
  lock_t l;
  node_t send_queue;
  node_t recv_queue;
  int write;
  int read;
  int msg_count;
} MessageBox;


static MessageBox MessageBoxen[MAX_MBOXEN];
```

图 10. Mailbox 结构体

定义一个长度为 MAX_MBOXEN 的缓冲区数组，数组的每一个元素是一种数据，数据名字存在 name 中，通信可以进行多次，因此通信内容存在一个 box 数组中，每个元素可存一条信息，一共可以存入 MAX_MBOX_LENGETH 条消息。其余域都是为正常读写缓冲区提供一些信息的域：write 和 read 域为信息写入 box 提供下标，有写入信息 write 加 1，有读出信息 read 减 1. Msg_count 域记录目前有多少条信息。定义两个队列 send_queue 和 recv_queue，分别用于等待消费者消费信息，和生产者生产出信息。

3.2 生产者-消费者问题

一般的生产者消费者问题首先需要用两个互斥锁来控制多个生产者间和多个消费者间的行为，防止不同进程同时对缓冲区内数据进行操作，其次，需要设置两个信号量：full_buf 和 empty_buf，用来表示当缓冲区已满或缓冲区为空。当缓冲区已满：生产者将被阻塞（压入 wait_queue）直到缓冲区有空位再被释放出来，当缓冲区为空，消费者将阻塞直到缓冲区

中有数据。

本次实验中，通过一个锁和 do_mbox_send 和 do_mbox_recv 函数实现。锁用来控制对缓冲区的读写操作是原子的，剩下两个函数用来控制缓冲区是否满或空，做出相应的阻塞操作。对于 do_mbox_send 函数，首先调用 do_mbox_is_full 判断 mbox 是否已满，若满就不能再送入消息了，将当前进程放入阻塞队列 send_queue。直到 mbox 有空出一个位置，才能继续送入消息。对于 do_mbox_recv 函数，首先调用 do_mbox_is_empty 判断 mbox 是否空，若空就没有消息可以取出，将当前进程放入阻塞队列 recv_queue。直到 mbox 有放入一条消息，才能继续取出消息。

4. 关键函数功能

4.1 do_kill

Do_kill 用来杀一个进程，可以是自己也可以是其他进程，由于杀的时候需要考虑该进程是否持有锁或被其他任务所等待，Kill 时需要将全部等待它的任务都释放出来。处理时要改变 pcb 信息，需要在临界区进行，代码如图 11.

```
291  static int do_kill(pid_t pid)
292  {
293    (void) pid;
294    /* TODO */
295    int num;
296    int i;
297    num = find_pid(pid);
298    if (num == -1) {
299      return -1;
300    }
301    if (pid == current_running->pid){
302      release_all_lock(pcb[num].lock);
303      enter_critical();
304      current_running->status = EXITED;
305      unblock_all(&(pcb[num].wait_queue));
306      mail_unblock(&pcb[num]);
307      scheduler_entry();
308      //leave_critical();
309    } else {
310      release_all_lock(pcb[num].lock);
311      enter_critical();
312      pcb[num].node.prev->next = pcb[num].node.next;
313      pcb[num].node.next->prev = pcb[num].node.prev;
314      pcb[num].status = EXITED;
315      unblock_all(&(pcb[num].wait_queue));
316      mail_unblock(&pcb[num]);
317      leave_critical();
318
319    }
320    return -1;
321  }
```

图 11. Do_kill 代码

4.2 initialize_pcb

由于在 pcb 中新添加了几个域，需要对提供的 initialize_pcb 函数做更改。域 wait_queue 需要初始化，pcb 中记录是否持有信息的数组也都需要初始化为 FALSE。Bonus 中完成的记录持有锁的信息的数组也需要初始化为 FALSE。代码如图 12.

```
static void initialize_pcb(pcb_t *p, pid_t pid, struct task_info *
{
    p->entry_point = ti->entry_point;
    p->pid = pid;
    p->task_type = ti->task_type;
    p->priority = 1;
    p->status = FIRST_TIME;

    int i;
    for(i = 0; i < MAX_MBOXEN; i++){
        p->mailbox[i] = FALSE;
    }
    for(i = 0; i < 32; i++){
        p->lock[i] = FALSE;
    }

    switch (ti->task_type) {
        case KERNEL_THREAD:
            p->kernel_tf.regs[29] = (uint32_t)stack_new();
            p->nested_count = 1;
            break;
        case PROCESS:
            p->kernel_tf.regs[29] = (uint32_t)stack_new();
            p->user_tf.regs[29] = (uint32_t)stack_new();
            p->nested_count = 0;
            break;
        default:
            ASSERT(FALSE);
    }
    p->kernel_tf.regs[31] = (uint32_t) first_entry;
    queue_init(&p->wait_queue);
}
```

图 12. Initialize_pcb 函数

4.3 do_mbox_send 和 do_mbox_recv

这两个函数是相对称的，一个往缓冲区写信息，一个读信息，因此函数具有极高的对称性。Do_mbox_send 需要先判断缓冲区是否满，而 do_mbox_recv 需要先判断缓冲区是否为空。对缓冲区的操作都需要得到锁后进行。写完信息后可释放所有等待缓冲区来信息的任务，读出一个信息也可释放所有等待往缓冲区放信息的任务。