Project4 Synchronization Primitives and IPC 设计文档

中国科学院大学
李静逸
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

设一个全局变量 next_pid，初始化为 1，然后将进程的 pid=next_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

首先在 pcb_t 的结构体里加入一个域 lock_t *l. 初始化为 NULL，

然后每次 task 获得锁，就将该锁存入 pcb.l 中，释放锁时让 pcb.l=NULL，

在 kill task 时检查要被杀的任务的 pcb 的 l 域，如果不为 NULL，就将 task 持有的锁释放，并将 task 的 pcb.l 赋值为 NULL。

（3）do_wait 的处理过程

首先在 pcb_t 的结构体里加入一个域 node_t node_queue，用来存 current_running 等待的 tasks。

然后扫描所有的 pcb，检查是否有对应的 pid 号，如果没有或者 pid==current_running->pid，return；如果有且 pid!=current_running->pid，就关中断，将 current_running 的状态设为 BLOCKED，并且将该对应 pid 号的 task 存入 current_running->node_queue.

最后调用 scheduler_entry().

（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）

我最开始在 do_wait 最后加了一句 leave_critical(),但上板时总是 LPPPPPPPPPP

后来发现 scheduler_entry()里已经开中断了，于是把 do_wait 里的 leave_critical()删去，就正常了。

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

条件变量：一个条件变量就是一个队列, 其中的线程正等待某个条件变为真。

信号量：信号量是一个同步对象，用于保持在 0 至指定最大值之间的一个计数值。当线程完成一次对该 semaphore 对象的等待时，该计数值减一；当线程完成一次对 semaphore 对象的释放时，计数值加一。当计数值为 0，则线程等待该 semaphore 对象不再能成功，会被阻塞到该 semaphore 对象变成 signaled 状态。semaphore 对象的计数值大于 0，为 signaled 状态；计数值等于 0，为 nonsignaled 状态.

屏障：同步屏障意味着任何线程/进程执行到此后必须等待，直到所有线程/进程都到达此点才可继续执行下文

条件变量的数据结构：

```
typedef struct condition{
    node_t wait_queue;
} condition_t;
```

信号量的数据结构：

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

屏障的数据结构：

```
typedef struct barrier{
    unsigned quorum;
    unsigned size;
    node_t wait_queue;
} barrier_t;
```

（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）

在同步原语的设计中没有遇到什么问题。

3. mailbox 设计

（1）mailbox 的数据结构以及主要成员变量的含义

Mailbox 的数据结构：

```
typedef struct
{
    char name[MBOX_NAME_LENGTH]; //信箱的名字
    Message box[MAX_MBOX_LENGTH]; //信箱里的消息
    uint32_t use_count; //访问该信箱的次数
    uint32_t msg_count;//信箱里有效消息的个数
    lock_t l;//锁，保证信箱的一致性
    node_t send_wait_queue;// 等待向该信箱写的进程
    node_t recv_wait_queue;// 等待读该信箱的进程
    int read_place;//写信箱的位置
    int write_place; //读信箱的位置
} MessageBox;
```

（2）producer-consumer 问题是指什么？你在 mailbox 设计中如何处理该问题？

Producer-consumer 问题指的是共享固定大小缓冲区的两个线程——即“生产者”和“消费者”——在实际运行时会发生的问题。生产者的主要作用是生成一定量的数据放到缓冲区中，然后重复此过程。与此同时，消费者也在缓冲区消耗这些数据。该问题的关键是要保证生产者不会在缓冲区满时加入数据，消费者也不会在缓冲区中空时消耗数据

（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）

最开始运行三国程序时，打印出初始信息后会出现内核态执行系统调用的错误，经仔细检查后发现是 do_mbox_open 的问题，找传入参数 name 的对应信箱时细节出了问题：如果没有名字为 name 的信箱，就应该新开一个空信箱，命名为 name，空信箱是通过 mailbox[i].name 的长度是否为 0 判断的，我判断成了传入参数 name 是否长度为 0，是一个很弱智的错误。

改掉这个错误后还是有错 LPPPPPPPPPPP，仔细检查了 mbox.c 没有发现问题，注释掉 test_sanguo 里的进程间通信操作，上板还是有问题，因此发现是 do_kill 和 do_wait 的问题。一步步的筛查，发现是在 do_wait 里 scheduler_entry 后面多加了 leave_critical(), scheduler_entry 里面已经有开中断操作了，所以不用在 do_wait 里开中断。

4. 关键函数功能

请列出上述各项功能设计里，你觉得关键的函数或代码块，及其作用

```
298  static int do_kill(pid_t pid)
299  {
300      (void) pid;
301      /* TODO */
302      int num,i;
303      for(i=0;i<NUM_PCBS;++i){
304          if(pcb[i].pid==pid&&pcb[i].status!=EXITED){
305              num=i;
306              break;
307          }
308      }
309      if(i==NUM_PCBS)
310          return -1;
311      if(current_running->pid==pid)
312          return -1;
313      enter_critical();
314      dequeue(&pcb[num].node);
315      pcb[num].status=EXITED;
316      unblock_all(&pcb[num].wait_queue);
317      lock_release_helper((lock_t*)(pcb[num].l));
318      pcb[num].l=NULL;
319      leave_critical();
320      return -1;
321  }
```

```
323 static int do_wait(pid_t pid)//put the current_running into pcb
324 {
325   (void) pid;
326   /* TODO */
327   int num;
328   int i;
329   if(pid==current_running->pid){
330     return -1;
331   }
332   for(i=0;i<NUM_PCBS;++i){
333     if(pcb[i].pid==pid && pcb[i].status!=EXITED){
334         num=i;
335         break;
336     }
337   }
338   if(i==NUM_PCBS)
339     return -1;
340   enter_critical();
341   current_running->status=BLOCKED;
342   enqueue(&pcb[num].wait_queue,(node_t*)current_running);
343   scheduler_entry();
344   return 0;
345 }
```

```
110 void condition_init(condition_t * c){
111     queue_init(&c->wait_queue);
112 }
113
114 /* TODO: Release lock m and block the thread (enqueued on c).  When
115    re-acquire m */
116 /* error */
117 void condition_wait(lock_t * m, condition_t * c){
118     enter_critical();
119     lock_release_helper(m);
120     ASSERT(disable_count);
121     block( &c->wait_queue );
122     lock_acquire_helper(m);
123     leave_critical();
124
125 }
126
127 /* TODO: Unblock the first thread waiting on c, if it exists */
128 void condition_signal(condition_t * c){
129     enter_critical();
130     ASSERT(disable_count);
131     //may need to add
132     unblock_one(&c->wait_queue);
133     //may need to add
134     leave_critical();
135 }
```

```
137 /* TODO: Unblock all threads waiting on c */
138 void condition_broadcast(condition_t * c){
139     enter_critical();
140     ASSERT(disable_count);
141     unblock_all(&c->wait_queue);
142     leave_critical();
143 }
```

```
146 void semaphore_init(semaphore_t * s, int value){
147     s->value=value;
148     queue_init(&s->wait_queue);
149 }
150 
151 /* TODO: Increment the semaphore value atomically */
152 void semaphore_up(semaphore_t * s){
153     enter_critical();
154     if(s->value<1 && !is_empty(&s->wait_queue))
155         unblock_one(&s->wait_queue);
156     else
157         s->value++;
158     leave_critical();
159 }
160 
161 /* TODO: Block until the semaphore value is greater than zero a
162 void semaphore_down(semaphore_t * s){
163     enter_critical();
164     if(s->value<1)
165         block(&s->wait_queue);
166     else{
167         s->value--;
168     }
169     leave_critical();
170 }
```

```
174 void barrier_init(barrier_t * b, int n){
175     b->quorum=n;
176     b->size=0;
177     queue_init(&b->wait_queue);
178 }
179 
180 /* TODO: Block until all n threads have called barrier_wait 
181 void barrier_wait(barrier_t * b){
182     enter_critical();
183     if(b->size+1>=b->quorum){
184         b->size=0;
185         unblock_all(&b->wait_queue);
186     }
187     else{
188         b->size++;
189         block(&b->wait_queue);//put current_running into wai
190     }
191     leave_critical();
192 
193 }
```

```
79  mbox_t do_mbox_open(const char *name)//typedef int mbox_t
80  {
81    (void)name;
82      /* TODO */
83      int i,j;
84      for (i=0;i<MAX_MBOXEN;++i){
85          if(same_string(name,mail_box[i].name)){
86              current_running->mailbox[i]=TRUE;
87              mail_box[i].use_count++;
88              return i;
89          }
90      }
91      for(i=0;i<MAX_MBOXEN;++i){
92          j=strlen(name);
93          if(strlen(mail_box[i].name)==0){
94              bcopy(name,mail_box[i].name,j);
95              current_running->mailbox[i]=TRUE;
96              return i;
97          }
98      }
99  }
```

```
104  void do_mbox_close(mbox_t mbox)
105  {
106      (void)mbox;
107      /* TODO */
108      mail_box[mbox].use_count--;
109      current_running->mailbox[mbox]=FALSE;
110      if(mail_box[mbox].use_count==0)
111          init_mbox_one(mbox);
112  }
```

```
160 void do_mbox_send(mbox_t mbox, void *msg, int nbytes)
161 {
162     (void)mbox;
163     (void)msg;
164     (void)nbytes;
165      /* TODO */
166     int i;
167     pcb_t *p;
168     do_mbox_is_full(mbox);
169     lock_acquire(&mail_box[mbox].l);
170     for(i=0;i<nbytes && i<MAX_MESSAGE_LENGTH ;++i)
171         mail_box[mbox].box[mail_box[mbox].write_place].msg[i]=((char*)msg)[i];
172     mail_box[mbox].write_place++;
173     mail_box[mbox].write_place = mail_box[mbox].write_place % MAX_MBOX_LENGTH;
174     mail_box[mbox].msg_count++;
175     lock_release(&mail_box[mbox].l);
176
177     enter_critical();
178     ASSERT(disable_count);
179     if(!is_empty(&mail_box[mbox].recv_wait_queue)){
180         p=(pcb_t*)dequeue(&mail_box[mbox].recv_wait_queue);
181         unblock(p);
182     }
183     leave_critical();
184 }
```

```
199 void do_mbox_recv(mbox_t mbox, void *msg, int nbytes)
200 {
201     (void)mbox;
202     (void)msg;
203     (void)nbytes;
204   /* TODO */
205     int i;
206     pcb_t *p;
207     do_mbox_is_empty(mbox);
208 
209     lock_acquire(&mail_box[mbox].l);
210     for(i=0;i<nbytes && i<MAX_MESSAGE_LENGTH ;++i)
211         ((char*)msg)[i]=mail_box[mbox].box[mail_box[mbox].read_place].msg[i];
212     mail_box[mbox].read_place++;
213     mail_box[mbox].read_place = mail_box[mbox].read_place % MAX_MBOX_LENGTH;
214     mail_box[mbox].msg_count--;
215     lock_release(&mail_box[mbox].l);
216 
217     enter_critical();
218     ASSERT(disable_count);
219     if(!is_empty(&mail_box[mbox].send_wait_queue)){
220         p=(pcb_t*)dequeue(&mail_box[mbox].send_wait_queue);
221         unblock(p);
222     }
223     leave_critical();
224 }
```

进程锁的系统调用（bonus）：

```
19  typedef enum {
20      SYSCALL_YIELD=0,
21      SYSCALL_EXIT,
22      SYSCALL_GETPID,
23      SYSCALL_GETPRIORITY,
24      SYSCALL_SETPRIORITY,
25      SYSCALL_SLEEP,
26      SYSCALL_SHUTDOWN,
27      SYSCALL_WRITE_SERIAL,
28      SYSCALL_PRINT_CHAR,
29      SYSCALL_SPAWN,
30      SYSCALL_KILL, //10
31      SYSCALL_WAIT,
32      SYSCALL_MBOX_OPEN,
33      SYSCALL_MBOX_CLOSE,
34      SYSCALL_MBOX_SEND,
35      SYSCALL_MBOX_RECV,
36      SYSCALL_TIMER,
37      USR_LOCK_INIT,///////add
38      USR_LOCK_ACQUIRE,/////add
39      USR_LOCK_RELEASE,/////add
40      NUM_SYSCALLS,
41  } syscall_t;
```

```
11  void usr_lock_init(int lock_id){
12    invoke_syscall(USR_LOCK_INIT, lock_id, IGNORE, IGNORE);
13  }
14 
15  void usr_lock_acquire(int lock_id){
16    invoke_syscall(USR_LOCK_ACQUIRE, lock_id, IGNORE, IGNORE);
17  }
18 
19  void usr_lock_release(int lock_id){
20    invoke_syscall(USR_LOCK_RELEASE, lock_id, IGNORE, IGNORE);
21  }
```

```
196     syscall[USR_LOCK_INIT] = (int (*)()) &sys_lock_init;
197     syscall[USR_LOCK_ACQUIRE] = (int (*)()) &sys_lock_acquire;
198     syscall[USR_LOCK_RELEASE] = (int (*)()) &sys_lock_release;
```

```
79  void sys_lock_init(int lock_id){
80      lock_init(&usr_lock[lock_id]);
81  }
82
83  void sys_lock_acquire(int lock_id){
84      enter_critical();
85      lock_acquire_helper(&usr_lock[lock_id]);
86      current_running->l=&usr_lock[lock_id];
87      leave_critical();
88  }
89
90  void sys_lock_release(int lock_id){
91      enter_critical();
92      lock_release_helper(&usr_lock[lock_id]);
93      current_running->l=NULL;
94      leave_critical();
95  }
```

参考文献

无