Project4 Synchronization Primitives and IPC 设计文档

中国科学院大学
李云志
2017.11.28

1. do_spawn, do_kill 和 do_wait 设计

(1) do_spwan 的处理过程，如何生成进程 ID
   a) 通过文件查找函数从伪文件系统中找到所需要进程的相关信息。
   b) 寻找空闲的 PCB 块将进程的信息存入该 PCB 块中，同时该 PCB 块的索引 i 记为该进程的 ID
   c) 将改进程加入到 ready_queue 中

(2) do_kill 的处理过程。Kill task 时如何处理锁
   a) 将进程的 PCB 块释放，即将其 status 更改为 EXITED
   b) 从所有队列中寻找该进程，并将该进程移除
   c) 唤醒所有正在等待给进程结束的任务，该动作通过条件变量完成。

   Kill task 时如何处理锁？

   i. 建立一个队列存储所有被用户进程使用到的锁，同时锁的结构体里边记录其持有者的 PID
   ii. 当执行一个 kill 操作时，去锁队列中检查所有锁的持有者是否与被 Kill 的进程相同，如果相同则释放这个锁。

(3) do_wait 的处理过程
   a) 将 current_running 加入到要等待进程的等待队列中，该动作通过条件变量完成。

值得注意的是，do_kill 与 do_wait 都是通过条件变量完成的。每一个 PCB 块对应一个条件变量，wait 动作与 kill 动作都通过 pid 索引，进一步找到相关条件变量从而完成相应动作。

2. 同步原语设计

(1) 条件变量、信号量和屏障的含义，及其数据结构包含的内容
   a) 条件变量即为通过一个变量来完成等待某个事件发生的动作，在此次实验中，条件变量即对应一个等待队列。

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

   b) 信号量数据结构包含一个可用资源数以及一个等待队列。其含义是如果当前存在可用资源即进行资源的分配，否则将相应任务加入到等待队列当中。

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

c) 屏障数据结构包含三个要素：屏障目标数、当前等待数、等待队列。其含义是当一个任务执行 barrier_wait 时，如果等待任务总数到达屏障数时，及释放所有的等待任务、否则也将其加入到等待队列当中。

```
typedef struct barrier{
    unsigned int total;
    unsigned int now;
    node_t wait_queue;
} barrier_t;
```

3. mailbox 设计

(1) mailbox 的数据结构以及主要的成员变量的含义

mailbox 数据结构如下：

```
typedef struct
{
  char name[MBOX_NAME_LENGTH];
  Message messages[MAX_MBOX_LENGTH];
  unsigned int use_count;
  semaphore_t full;
  semaphore_t empty;
  lock_t mutex;
  int send;
  int receive;
} MessageBox;
```

说明：

其中一条信息是一个长度最大为 MAX_MESSAGE_LENGTH 的字符串，信箱一共可以存储 MAX_MBOX_LENGTH 个信息。

通过两个信号量 full 与 empty 来解决生产者与消费者问题。

互斥锁 mutex 来防止生产者与消费者共同访问信箱。

Send 与 receive 用来维护一个 FIFO。

(2) producer-consumer 问题是指什么？如何处理该问题？

生产者与消费者问题指的是生产者不停的往有界缓冲区存入信息，消费者从有界缓冲区中取出信息。需要解决的问题是当缓冲区已满时的生产者行为以及缓冲区已空时的消费者的行为。

本次实验中采用信号量来解决该问题。当缓冲区已满则使生产者休眠，当有新的消费请求时即可将其唤醒。当缓冲区已空时的消费者行为同理。

4. 关键函数功能

条件变量相关：

```
void condition_wait(lock_t * m, condition_t * c){
    enter_critical();
    lock_release_helper(m);
    ASSERT(disable_count);
    block(&c->wait_queue);
    lock_acquire_helper(m);
    leave_critical();
}//条件变量等待操作
```

```
void condition_signal(condition_t * c){
    enter_critical();
    ASSERT(disable_count);
    unblock_one(&c->wait_queue);
    leave_critical();
}//条件变量信号
```

```
void condition_broadcast(condition_t * c){
    enter_critical();
    ASSERT(disable_count);
    unblock_all(&c->wait_queue);
    leave_critical();
}//broadcast 操作
```

信号量相关：

```
void semaphore_up(semaphore_t * s){
  enter_critical();
  if(!is_empty(&s->wait_queue))
  unblock_one(&s->wait_queue);
  else
  s->value++;
  leave_critical();
}
```

```
void semaphore_down(semaphore_t * s){
  enter_critical();
  if(s->value<1)
    block(&s->wait_queue);
  else
    s->value--;
  leave_critical();
}
```

屏障相关：

```
void barrier_wait(barrier_t * b){
  enter_critical();
  if(b->now+1>=b->total)
  {
    b->now=0;
    unblock_all(&b->wait_queue);
  }
  else
  {
    b->now++;
    block(&b->wait_queue);
  }
  leave_critical();
}
```

Mailbox 相关：

```
typedef struct
{
  char name[MBOX_NAME_LENGTH];
  Message messages[MAX_MBOX_LENGTH];
  unsigned int use_count;
  semaphore_t full;
  semaphore_t empty;
  lock_t mutex;
  int send;
  int receive;
} MessageBox; //结构体定义
```

```
mbox_t do_mbox_open(const char *name)
{
  (void)name;
  enter_critical();
  int i=0;
  for(i=0;i<MAX_MBOXEN;i++) //寻找相同名字的 mailbox
  {
    if(same_string(name,MessageBoxen[i].name))
   {
     MessageBoxen[i].use_count++;
     leave_critical();
     return i;
   }
  }
  for(i=0;i<MAX_MBOXEN;i++)
  //如果没有名字相同的，则开一个新的 mailbox
  {
    if(MessageBoxen[i].use_count==0)
    {
      int length = strlen(name);
      bcopy(name,MessageBoxen[i].name,length*sizeof(char));
      MessageBoxen[i].use_count++;
      leave_critical();
      return i;
    }
  }
  leave_critical();
  return -1;
```

```
void do_mbox_send(mbox_t mbox, void *msg, int nbytes)
{
  (void)mbox;
  (void)msg;
  (void)nbytes;
  semaphore_down(&MessageBoxen[mbox].empty);
  bcopy(msg,&MessageBoxen[mbox].messages[MessageBoxen[mbox].send],nbytes*sizeof(char));
  MessageBoxen[mbox].send = (MessageBoxen[mbox].send+ 1)%MAX_MESSAGE_LENGTH;
  semaphore_up(&MessageBoxen[mbox].full);
}
```

Bonus 在 kill 中添加的部分：

```
node_t* ptr = lock_queue.next;
  while(ptr != &lock_queue) {
    if(((lock_t*) ptr)->held_task && ((pcb_t*)((lock_t*) ptr)->held_task)->pid == pid) {
      leave_critical();
      lock_release((lock_t*) ptr);
      enter_critical();
      ptr->prev->next = ptr->next;
      ptr->next->prev = ptr->prev;
      }
      ptr = ptr->next;
}//寻找持有者与被 kill 进程 PID 相同的锁，并将其释放
```

参考文献

[1] 无