

# 实验十二：用进程实现服务的微内核

## 实验十二：用进程实现服务的微内核

- 1 实验目的
- 2 实验概述
- 3 本实验完成的功能
- 4 实现原理
  - 4.1 基本原理
  - 4.2 消息API的实现
  - 4.3 进程控制块对进程间通信的支持-消息与队列
  - 4.4 系统内核对进程间通信的支持-调度与阻塞
    - 4.5 msg\_send函数具体实现
    - 4.6 msg\_recv函数具体实现
      - 4.6.1 具体思路说明
      - 4.6.2 代码实现
- 5 情景分析
  - 5.1 任务进程的初始化-后台运行
    - 5.1.1 初始化任务进程的进程控制块
    - 5.1.2 任务进程的说明
  - 5.2 user\_get\_ticks函数的调用
- 6 实现中出现的问题
  - 6.1 内核内是否允许抢占？
  - 6.2 还有很多很多问题
- 7 测试过程
  - 7.1 测试代码
  - 7.2 测试结果
- 8 实验感想

## 1 实验目的

1. 实现发送消息与接收消息的系统调用
2. 使用发送消息与接收消息，将一些系统调用修改为基于进程间通信来实现。

## 2 实验概述

在我打算实现硬盘驱动以便读取文件的时候，我发现硬盘的操作是异步的，我往硬盘中写命令数据后，硬盘通过中断的方式来通知我完成了操作。这一种事件驱动的方式在我们的操作系统中，如果不使用进程间通信的方式来实现，会麻烦很多（来源于看linux-0.11源码和orange源码对比的感想）。因此，我打算先参考orange的实现，完成进程间通信功能。以后在增加功能的时候，就可以很简单的通过增加一个任务进程的方式来实现。

这一种将操作系统的一些功能转移给各个不同的任务进程的实现，是将来实现简洁而优雅的微内核的基础。

## 3 本实验完成的功能

1. 将0x80号中断设置为系统调用入口，并新增了sendrec系统调用，用于发送或者接受消息。
2. 在用户端，将sendrec系统调用使用send\_rec函数再包装，以使用户库函数的使用
3. 在内核，sendrec系统调用会转移到sys\_sendrec函数中，该函数会根据参数的不同，分别调用msg\_send,或msg\_receive函数发送或接受消息。
4. 基于sendrec系统调用，我实现了两个简单的功能user\_get\_ticks， user\_get\_pid，用来测试该系统调用的正确性。

## 4 实现原理

进程间通信的实现一般会有两种类型：同步和异步。异步的实现虽然没有死锁的烦恼，但是实现的难度更高。以学习研究目的，我采取了一种相对比较简单同步的实现。这也就是说，一个进程发送消息后，一定会被阻塞住，直到对方接受了消息。

### 4.1 基本原理

在发送消息与接收消息的实现中，基本原理为内存复制。也就是说，发送消息的进程将消息准备好，然后接收消息的进程将消息复制过来，这就完成了一次消息传递。这一部分的代码说明可见下面：

```
// 发送消息的一方, pid=3
/* 准备消息内容 */
message_t send_msg;
msg_reset(&send_msg);
send_msg.type = GET_TICKS;
/* 将消息的地址传递到内核中, 发送到指定pid=1的进程 */
msg_send_recv(SEND, 1, &send_msg);
/*****
// 接受消息一方, pid=1, 接受pid=3的进程的信息
message_t recv_msg;
msg_reset(&recv_msg);
/* 接受消息 */
msg_send_recv(RECEIVE, 3, &recv_msg)
```

受益于同步的进程间通信的实现，对于两个发送与接受消息的进程之间，不需要维护任何消息缓冲区。（先忽略地址权限的问题）消息的传递，只需要依赖在msg\_send\_recv函数内部修改指定地址处的内容即可。

以上代码同时说明了我实现的进程间通信的API，关于该API的底层实现以及一些关键问题的解决，下面会做进一步的解释。

### 4.2 消息API的实现

以下代码并不是实现进程间通信的核心，不过作为一个对系统调用的包装，在这里还是做一下适当的说明。

在调用msg\_send\_recv函数发送或者接受消息的时候，该函数会根据function的值调用sendrec系统调用。这里需要说明的是，为了简化上层的API，在这里的参数中，除了send和receive，我还设置了BOTH参数，该参数可以先发送消息再接受消息，因此上层API可以只调用一个带有BOTH参数的msg\_send\_recv函数，就可以完成一整个进程间通信的过程。

---

```

PUBLIC int msg_send_recv(int function, int src_dest, message_t* msg)
{
    int ret = 0;

    if (function == RECEIVE)
        com_memset(msg, 0, sizeof(message_t));

    switch (function) {
    case BOTH:
        ret = sendrec(SEND, src_dest, msg);
        if (ret == 0)
            ret = sendrec(RECEIVE, src_dest, msg);
        break;
    case SEND:
    case RECEIVE:
        ret = sendrec(function, src_dest, msg);
        break;
    default:
        assert((function == BOTH) ||
               (function == SEND) || (function == RECEIVE));
        break;
    }
    return ret;
}

```

这里，sendrec函数在kernel/syscall.asm中有下面的定义，该函数仅仅是传递相关参数后，调用int 0x80号中断，陷入内核调用系统调用。

```

sendrec:
    mov eax, _NR_sendrec
    mov ebx, [esp + 4] ; function
    mov ecx, [esp + 8] ; src_dest
    mov edx, [esp + 12] ; p_msg
    mov esi, [g_cur_proc]
    int INT_VECTOR_SYS_CALL
    ret

```

## 4.3 进程控制块对进程间通信的支持-消息与队列

为了实现进程间通信，每一个进程对应的进程控制块做了相应的调整，增加了一些成员变量。

进程控制块定义在include/proc/process.h中，代码如下

```

/* 这就是PCB啦，进程控制块 */
typedef struct s_proc{
    proc_regs_t regs;
    uint32_t pid;
    uint32_t status; /* 0 */

```

```

char p_name[16];
void * kernel_stack;

/* 进程优先级 及时间片信息*/
uint32_t remain_ticks;          /* remained ticks */
uint32_t priority;

/* 进程的内存信息 */
mm_struct_t *mm, *active_mm;

/* 消息发送信息 */
uint32_t p_recvfrom;
uint32_t p_sendto;
uint32_t p_flags;
uint32_t has_int_msg;
struct s_proc * q_sending;
struct s_proc * next_sending;
/* 消息内容 */
message_t * p_msg;

}proc_task_struct_t;

```

在原有的进程控制块上，我新增加了以下几项，这几项的作用在下面的表格中列了出来

成员名称	作用
p_recvfrom	记录该进程在阻塞并且处于RECEIVING状态时，在等待哪一个进程发送？
p_sendto	记录该进程在阻塞并且处于SENDING状态时，在等待哪一个进程接受？
p_flags	该进程的状态，如RECEIVE，SENDING等
has_int_msg	是否有中断信息，指irq线带来的中断，这里并未实现
q_sending	记录发消息给该进程的对应的进程控制块的指针
next_sending	如果发消息给该进程的进程不止一个，这里将会建立一个进程的队列
p_msg	阻塞状态下，发送消息的内容或者接受消息的地址。

## 4.4 系统内核对进程间通信的支持-调度与阻塞

系统内核对进程间通信的支持，主要通过0x80号中断的系统调用来实现。

在产生80号中断后，内核函数的运行轨迹如下所示。

```

_interrupt_handler(regs)
-->>
sys_call(regs)
-->>
sys_sendrec(int, int, message_t*, proc_task_struct_t*)
-->>
    msg_send(proc_task_struct_t * current, int dest, message_t * m)
或 msg_recv(proc_task_struct_t * current, int sec, message_t * m)

```

其中对最后两个函数的作用，下面的表格做了详细的说明

函数名称	作用
	将message 从当前进程发送至pid==dest的目的进程中。
msg_send	若对方暂不接收，则阻塞直到对方接收 若对方处于阻塞状态并等待接受，则将消息复制过去并唤醒对方
	从pid==src的源进程中，获得消息，放到当前进程中的消息m处。
msg_recv	若暂时不能够获取，则阻塞直到对方发送消息。 若对方处于阻塞状态并等待发送成功，则将对方消息复制过来，并唤醒对方

以上的两个函数，在调用的时候可能会在不同的情况阻塞当前进程，或者唤醒目的进程。阻塞和唤醒的实现通过修改进程控制块的成员status即可实现。在调度算法中，会只调度status == \_PROC\_RUN的进程，因此通过修改status即可实现阻塞与唤醒。在阻塞进程后，还通过调用进程调度模块，修改当前进程指针g\_cur\_proc与当前进程内核栈指针g\_cur\_proc\_context\_stack，来保证在从中断退出后，一定会进入到其他的进程中，而不会返回原进程。这部分阻塞与唤醒的实现，可见kernel/sys\_call.c的代码：

```
/* 结束阻塞，唤醒该进程 */
PRIVATE void unblock(proc_task_struct_t* p)
{
    assert(p->p_flags == 0);
    p->status = _PROC_RUN;
}

/* 阻塞当前进程 */
PRIVATE void block(proc_task_struct_t* p)
{
    p->status = _PROC_SLEEP;
    assert(p->p_flags);
    proc_schedule(); // 该函数会找到下一个可执行的进程，并更新当前进程
}
```

该部分实现的重要结论：

该系统调用能够保证，调用中断到从中断回来恢复现场后（中间可能被阻塞了多个时钟中断），信息一定发送/接受成功。

### 4.5 msg\_send函数具体实现

该函数需要下列参数

参数名称	current	dest	m
参数作用	调用该函数的进程（当前进程）	目的进程pid	指向需要发送的消息的指针

在发送消息函数msg\_send的具体实现中，该函数的总体编写思路如下。

- 1. 查看发送目的进程状态
  - 1. 如果目的进程正好处于等待该进程信息的状态
    - 1. 得到目的进程中的用于接受信息的message\_t结构的地址
    - 2. 使用com\_memncpy函数将消息内容复制过去。
    - 3. 修改目的进程状态，目的进程此时不被阻塞
  - 2. 如果目的进程没有在等待

1. 将该进程放到目的进程的接收队列中。
2. 调用block函数，阻塞自己，将状态置为\_PROC\_SLEEP。

实现该过程的代码可见kernel/sys\_call.c，如下所示。

```
PRIVATE int msg_send(proc_task_struct_t * current, int dest, message_t * m)
{
    proc_task_struct_t * sender = current;
    proc_task_struct_t * p_dest = g_pcb_table + dest; /* proc dest */

    /* check for deadlock here */
    if (deadlock(proc2pid(sender), dest)) {
        panic(">>DEADLOCK<< %s->%s", sender->p_name, p_dest->p_name);
    }

    if ((p_dest->p_flags & RECEIVING) && /* dest is waiting for the msg */
        (p_dest->p_recvfrom == proc2pid(sender) ||
         p_dest->p_recvfrom == ANY)) {

        com_memncpy(va2la(dest, p_dest->p_msg),
                    va2la(proc2pid(sender), m),
                    sizeof(message_t));
        p_dest->p_msg = 0;
        p_dest->p_flags &= ~RECEIVING; /* dest has received the msg */
        p_dest->p_recvfrom = NO_TASK;
        unblock(p_dest);
    }
    else { /* dest is not waiting for the msg */
        sender->p_flags |= SENDING;
        sender->p_sendto = dest;
        sender->p_msg = m;

        /* append to the sending queue */
        proc_task_struct_t * p;
        if (p_dest->q_sending) {
            p = p_dest->q_sending;
            while (p->next_sending)
                p = p->next_sending;
            p->next_sending = sender;
        }
        else {
            p_dest->q_sending = sender;
        }
        sender->next_sending = 0;

        block(sender);
    }

    return 0;
}
```

## 4.6 msg\_recv函数具体实现

该函数需要下列参数

参数名称	<b>current</b>	<b>src</b>	<b>m</b>
参数作用	调用该函数的进程（当前进程）	源进程pid	指向存放接受的消息的指针

在接受消息函数msg\_recv的具体实现中，该函数的总体编写思路是：先看目的进程是否已经准备好，如果准备好了就把信息复制过来，并且唤醒被阻塞的进程，如果没有复制过来就阻塞自己等待消息被发送到自己处。以下是代码的详细思路说明以及相应的代码实现，代码可见kernel/sys\_call.c

### 4.6.1 具体思路说明

- 查看源进程的状态（源进程是否准备好了消息）
  - 该步骤需要确定两个量

1. 变量名	类型	作用
p_from	proc_task_struct_t	确定src对应的源进程的进程控制块对应的指针 确定src对应源进程是否准备好了消息
copyok	int	如果准备好了，则copyok==1 不然就为0
  - 如果对源没有特定的要求，进程控制块中的q\_sending又不为空，则说明已经有进程将消息准备好了准备发过来。则可以设置copyok等于1，并且p\_from就是进程控制块宏的p\_from成员。
  - 如果对源有特定的要求，则遍历队列寻找队列中等待被接收的进程是否有特定的进程
    - 如果有，则同样设置p\_from为找到的进程控制块的对应指针，设置copyok为1
    - 如果找不到，则copyok为0
- 根据源进程的状态，下面有两种行为
  - 如果源进程已经准备好了消息，那就使用com\_memncpy函数将消息复制过来。并且接触源进程的阻塞
  - 如果源进程没有准备好消息，则阻塞自身，并设置自身为等待该进程，等待消息被发送过来。

### 4.6.2 代码实现

```
PRIVATE int msg_receive(proc_task_struct_t * current, int src, message_t * m)
{
    proc_task_struct_t * p_who_wanna_recv = current;
    proc_task_struct_t * p_from = 0; /* from which the message will be fetched */
    proc_task_struct_t * prev = 0;
    int copyok = 0;

    /* 确定p_from, 在队列中找到一个用于接受消息的进程指针 */
    if (src == ANY) {
        if (p_who_wanna_recv->q_sending) {
            p_from = p_who_wanna_recv->q_sending;
        }
    }
    else {
        p_from = &g_pcb_table[src];
    }
}
```

```

if ((p_from->p_flags & SENDING) &&
    (p_from->p_sendto == proc2pid(p_who_wanna_recv))) {
    /* Perfect, src is sending a message to
     * p_who_wanna_recv.
     */
    copyok = 1;

    proc_task_struct_t * p = p_who_wanna_recv->q_sending;
    /* 遍历队列，找到发送消息的源进程 */
    while (p) {
        assert(p_from->p_flags & SENDING);
        /* 如果p就是我要收的源进程，就不用找了 */
        if (proc2pid(p) == src) { /* if p is the one */
            p_from = p;
            break;
        }
        prev = p;
        p = p->next_sending;
    }
}

/* 标注了copyok，意味着已经找到了p_from，确定了源进程，并且源进程已经准备好了消息，可以将消息复制过来 */
if (copyok) {
    /* 维护待接收信息的进程队列 */
    /* 如果要接受的进程位于队列头，则由于队列头需要从队列中除去，需要做一些处理 */
    if (p_from == p_who_wanna_recv->q_sending) { /* the 1st one */
        p_who_wanna_recv->q_sending = p_from->next_sending;
        p_from->next_sending = 0;
    }
    /* 如果不是队列头，则源进程的前一项和后一项接在一次 */
    else {
        prev->next_sending = p_from->next_sending;
        p_from->next_sending = 0;
    }

    /* copy the message */
    com_memncpy(va2la(proc2pid(p_who_wanna_recv), m),
                va2la(proc2pid(p_from), p_from->p_msg),
                sizeof(message_t));

    p_from->p_msg = 0;
    p_from->p_sendto = NO_TASK;
    p_from->p_flags &= ~SENDING;
    unblock(p_from);
}

/* 从已有的接收消息等待队列中，找不到自己想要的信息 */
else { /* nobody's sending any msg */
    p_who_wanna_recv->p_flags |= RECEIVING;
    p_who_wanna_recv->p_msg = m;
}

```



```

    if (src == ANY)
        p_who_wanna_recv->p_recvfrom = ANY;
    else
        p_who_wanna_recv->p_recvfrom = proc2pid(p_from);

    /* block住, 然后退出系统调用后时钟中断会使该进程被阻塞 */
    block(p_who_wanna_recv);
}
return 0;
}

```

## 5 情景分析

下面挑两个具体的场景梳理一遍以上代码的运行流程。

### 5.1 任务进程的初始化-后台运行

#### 5.1.1 初始化任务进程的进程控制块

在该操作系统刚开机时，操作系统会进行各项初始化工作，其中包括后台任务的初始化。

这些后台任务实际上是支撑操作系统的一个个内核进程。每一个内核进程负责操作系统的其中一项具体事务，在本场景中，初始化的是一个普通系统任务，该任务进程会处理一些简单的消息，并且返回相应的结果。

在kernel/main.c中，调用了task\_init()函数，该函数的定义可见kernel/sys\_tasks.c中

```

// TASK_SYS 为 常量1
PUBLIC void task_init(){
    proc_init_a_task(TASK_SYS, "sys_call", TASK_SYS, (void*)task_sys_call, (proc_regs_t*)0x20000, 2);
}

```

在这里，将pid为1的任务进程相关信息在进程控制块中初始化好了，相关的函数proc\_init\_a\_task定义在kernel/proc/process.c中，用于在内核的进程控制块数组中写入一个进程的相关信息，并设置状态为\_PROG\_RUN，之后当进程正式启动的时候，该任务进程便会以优先级为2的登记被调度程序调度。

#### 5.1.2 任务进程的说明

任务初始化时，该任务将task\_sys\_call这个函数设置为了一个进程，该函数定义可见kernel/sys\_tasks.c中，代码如下。

该函数设置为了一个无限循环，运行的逻辑其实很简单，不断地尝试接收消息，一旦接收到一个消息就根据消息的类型做出相应的处理，因此，我通过此任务进程，处理我在调用某些函数时发出的消息，并且给源进程返回相应的消息。

```

PRIVATE void task_sys_call(){
    message_t msg;
    while (1) {
        msg_send_recv(RECEIVE, ANY, &msg);
        int src = msg.source;
    }
}

```

```

switch (msg.type) {
case GET_TICKS:
    msg.RETVAL = g_ticks;
    msg_send_recv(SEND, src, &msg);
    break;
case GET_PID:
    msg.RETVAL = g_pcb_table[msg.source].pid;
    msg_send_recv(SEND, src, &msg);
    break;
default: {}
    panic("unknown msg type");
    break;
}
}
}

```

## 5.2 user\_get\_ticks函数的调用

在调用该函数的时候，中间有比较多的进程的状态转换，同时由于函数较多，单独讲解每个函数并不容易说明一整个基于消息机制的系统接口的实现。因此下面以函数调用链的方式来说明从调用user\_get\_ticks函数到最终获得当前系统时钟的值的過程。

< 当前用户进程 pid : 5 >

```

user_get_ticks()
-> msg_send_recv(SEND)
-> sendrec()
-> [!]0x80号中断，保存现场!
-> _interrupt_handler(regs)
-> sys_call(regs)
-> sys_sendrec(...)
-> msg_send(...)
-> block()
-> proc_schedule()注意该函数修改了g_cur_proc与对应上下文指针
-> block()
-> msg_send(...)
-> sys_sendrec(...)
-> sys_call(regs)
-> _interrupt_handler(regs)
-> [!]从中断回来，恢复现场，被调度到任务进程中

```

< task\_sys\_call pid : 1 >

```

task_sys_call()
-> msg_send_recv(RECEIVE)
-> sendrec()
-> [!]0x80号中断，保存现场!
-> _interrupt_handler(regs)
-> sys_call(regs)

```

```

-> sys_sendrec(...)
-> msg_recv(...)
-> unblock() 注意：发现有消息被发送，于是接收消息，并解除阻塞
-> msg_recv(...)
-> sys_sendrec(...)
-> sys_call(regs)
-> _interrupt_handler(regs)
-> [!]从中断回来，恢复现场，调度的结果不确定，这里不妨假设时间片没有用完
-> sendrec()
-> msg_send_recv(RECEIVE)
-> msg_send_recv(SEND) 接收到GET_TICKS类型的消息，任务进程做出的反应是通过消息发送ticks。
-> sendrec()
-> [!]0x80号中断，保存现场！
-> _interrupt_handler(regs)
-> sys_call(regs)
-> sys_sendrec(...)
-> msg_send(...)
-> block() 目标进程并没有在等待消息，于是阻塞自身
-> msg_send(...)
-> sys_sendrec(...)
-> sys_call(regs)
-> _interrupt_handler(regs)
-> [!]从中断回来，假设被调度到用户进程中

< 当前用户进程 pid : 5 >
-> [!]被调度回来，恢复现场。
-> sendrec()
-> msg_send_recv(SEND)
-> msg_send_recv(RECEIVE)
-> sendrec()
-> [!]0x80号中断，保存现场！
-> _interrupt_handler(regs)
-> sys_call(regs)
-> sys_sendrec(...)
-> msg_recv(...)
-> unblock() 从任务进程获得消息，并解除任务进程阻塞
-> msg_recv(...)
-> sys_sendrec(...)
-> sys_call(regs)
-> _interrupt_handler(regs)
-> [!]从中断回来，恢复现场，调度的结果不确定，这里不妨假设时间片没有用完
-> sendrec()
-> msg_send_recv(RECEIVE)
-> user_get_ticks() 最终从msg.u.m3i1中获得返回值，返回到用户进程中，打印出来。

```

## 6 实现中出现的问题

### 6.1 内核内是否允许抢占？

在实现的时候，我对orange的代码一知半解，之前一直对block函数的理解不够透彻，因此并不清楚阻塞是如何实现的。下面是block函数的代码。

```
PRIVATE void block(proc_task_struct_t* p)
{
    p->status = _PROC_SLEEP;
    assert(p->p_flags);
    proc_schedule();
    /* 自旋锁 */
    // while (p->status != _PROC_RUN);
}
```

进程间通信的实现要求进程能够主动阻塞自己，但由于该函数运行在内核态，我对于内核态内能否被抢占开始产生了疑惑。我原先的考虑是内核态内可以被抢占，因此开了时钟中断，并且在block函数内增加了自旋锁，这样子就可以实现block的状态。但是后来的结果是这样子在内核态随意抢占的话我的系统很可能会出问题（因为我之前一直是按照内核态不能够被抢占的思路来设计的）。

关于我这点疑问的产生，其实是因为我对进程的调度理解不够深入。进程的调度其实在运行proc\_schedule函数修改g\_cur\_proc，并且修改进程状态位status就已经能够使进程被阻塞了。有一点我之前没有意识到的是，中断返回恢复现场后，由于之前修改了全局的当前进程指针，一定会恢复到其他进程中，然后又由于之前的进程的状态已经被设置为\_PROC\_SLEEP状态了，因此已经实现了阻塞，不需要在内核态内通过自旋的方式来实现阻塞。

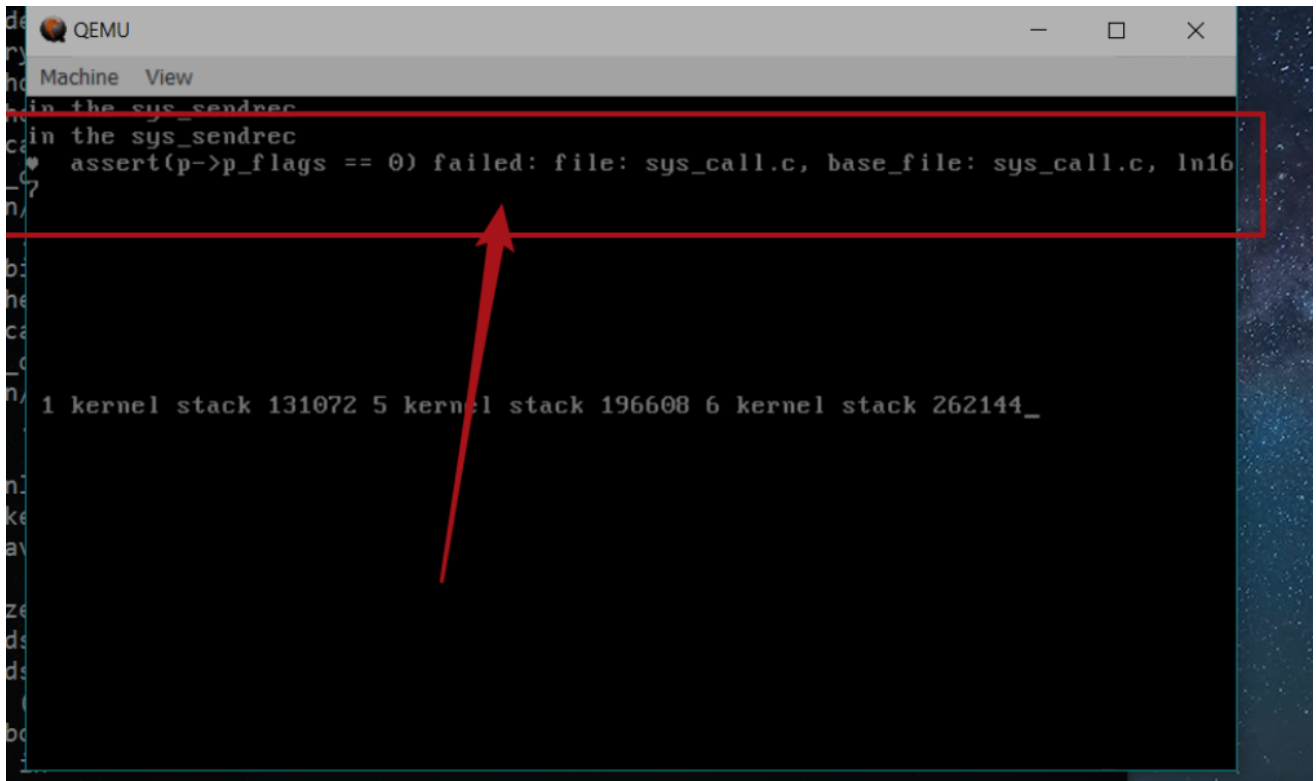
## 6.2 还有很多很多问题

有很多问题是在单步调试的过程中一点点摸出来的，同时，assert函数的实现，也给了我很大的帮助，关于这个assert函数的实现，我参考自orange对其相关的讲解，并且将对应的代码实现在了const.h和debug.h中。

assert函数的作用在于，如下面一段代码，对于解锁的代码，必须在flag为0的情况下才有可能调用该函数，但是如果代码的实现出现了差错，这里就可能在flag不为0的情况下调用了unlock函数。如何避免这种情况呢？assert的函数正是用在这样的场景中，可以将程序的错误发现在刚开始发生的地方，避免让错误扩大到最后难以调试。

```
PRIVATE void unlock(proc_task_struct_t* p)
{
    assert(p->p_flags == 0);
    p->status = _PROC_RUN;
}
```

如果运行上面的那一段代码时flag不为0，运行时会有这样的提示，并且关闭中断，死循环，停止操作系统的运行。此时我们就可以通过查看堆栈信息，去调试和查错。



## 7 测试过程

### 7.1 测试代码

在test/text.c中可以看到本次测试的代码，本次测试初始化了2个用户进程，以及一个在task\_init函数中初始化好的任务进程，通过进程间通信机制，两个用户进程都可以从任务进程中获取他们需要的数据。

```
void main_test(){
    // _test1();
    // _test2();
    // _test3();
    // _test4();
    // _test5();
    // _test_get_ticks();
    _test_background_task();
    // _test_hd();
}

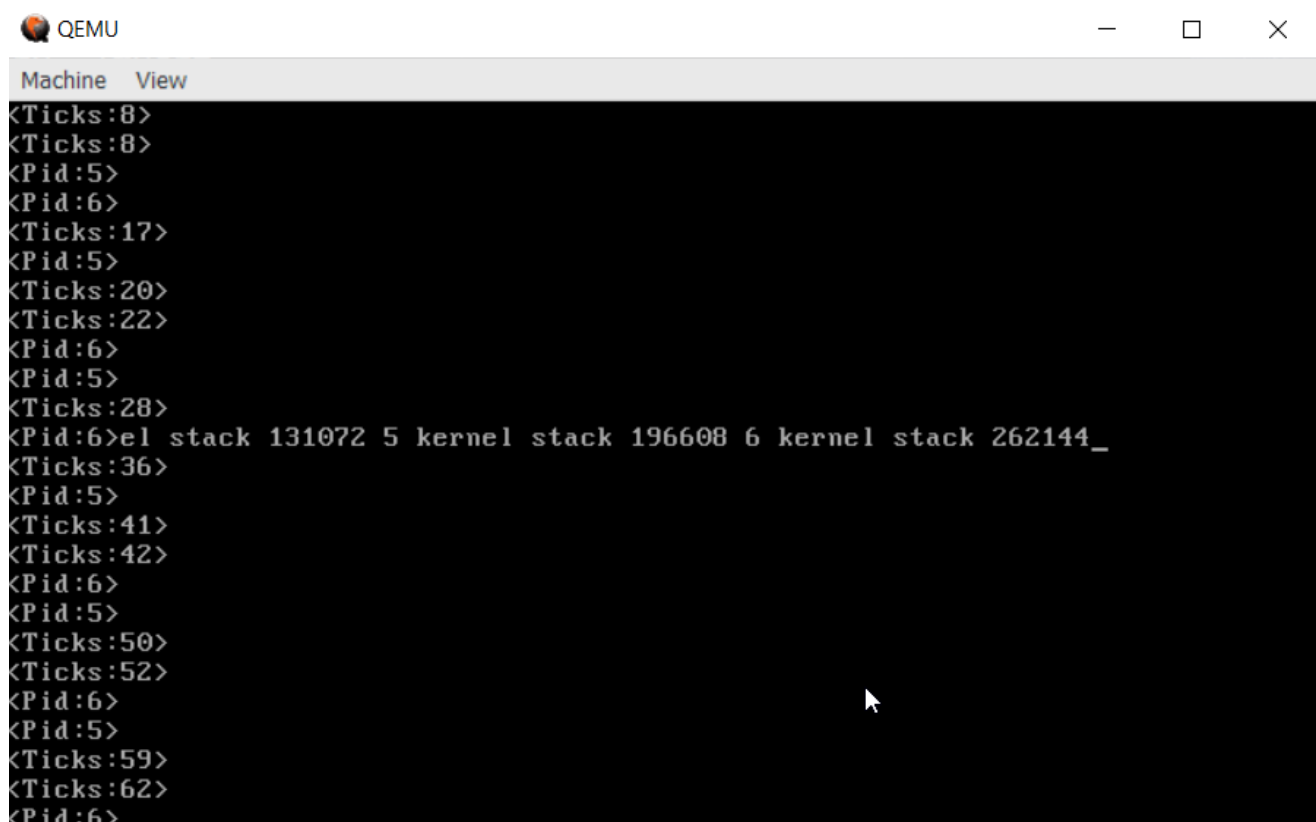
// 一个不断调用系统接口的进程
PRIVATE void _test_process_get_process(){
    while(1){
        for (int i = 0; i < 100000; i++){
            for (int j = 0; j < 1000; j++){
            }
            com_printk("<Ticks:%d>\n", user_get_ticks());
            com_printk("<Pid:%d>\n", user_get_pid());
        }
    }
}
```

```
// 另一个不断调度系统接口的进程
PRIVATE void _test_process_get_process_2(){
    while(1){
        for (int i = 0; i < 100000; i++){
            for (int j = 0; j < 1000; j++){
            }
            com_printk("<Ticks:%d>\n", user_get_ticks());
            com_printk("<Pid:%d>\n", user_get_pid());
        }
    }

PRIVATE void _test_background_task(){
    // 初始化进程, pid分别为5和6
    _init_a_process(5, "test_get_process", 5, _test_process_get_process, (proc_regs_t *)0x30000, 3);
    _init_a_process(6, "test_get_process", 6, _test_process_get_process, (proc_regs_t *)0x40000, 3);
    g_cur_proc = &g_pcb_table[5];
    g_cur_proc_context_stack = g_cur_proc->kernel_stack;
    // _basic_cli();
    // 开启,进入多进程
    _proc_restart();
}
```

## 7.2 测试结果

两个进程能够同时进行, 并且从任务进程中获取了相应的信息。



```
QEMU
Machine View
<Ticks:8>
<Ticks:8>
<Pid:5>
<Pid:6>
<Ticks:17>
<Pid:5>
<Ticks:20>
<Ticks:22>
<Pid:6>
<Pid:5>
<Ticks:28>
<Pid:6>el stack 131072 5 kernel stack 196608 6 kernel stack 262144_
<Ticks:36>
<Pid:5>
<Ticks:41>
<Ticks:42>
<Pid:6>
<Pid:5>
<Ticks:50>
<Ticks:52>
<Pid:6>
<Pid:5>
<Ticks:59>
<Ticks:62>
<Pid:6>
```

## 8 实验感想

本次实验做的是进程间通信。

为什么这一次实验我决定要做进程间通信呢？我之前看了很多关于linux的书，代码风格并没有特别大的问题，但是他们内核中各个子模块的那种强耦合关系，让我十分难以下手，我曾弄过硬盘驱动程序，做到一半没有坐下来（只做了读取硬盘信息的函数），我也写过内存管理和切换页表的相关代码（那些代码依然存在项目中，不过没有派上用场）但是还有一些依赖我没有处理好。我发现无论我如何组织代码，一个特别难解决的问题就是各子模块的强耦合。而这强耦合的结果，便是linux的代码虽然好，但是我用不了。

回过头来看orange的代码，看到后面orange走上了与linux截然不同的道路：微内核，便蛮有兴趣看了一下。对比一下如今主流的现代操作系统，window采用的是混合式内核（宏内核和微内核结合），linux则是纯正的宏内核。对于书中作者对微内核的评价，我感到十分认同。简洁，优雅，耦合度大大降低（虽然不可避免的会有耦合性）。思量再三，想到实现微内核中所必须的进程间通信，对之后其他模块的实现大有裨益，便铁下心来，干了这把硬骨头。

不过，话又说回来，如果没有任何参考，裸写进程间通信的话，我想我会在无数bug的地狱中无法翻身。orange关于进程间通信的实现于我而言，是一个极好的范例，模仿着orange的实现，我可以说是实现了一个和orange系统几乎一模一样的进程间通信的机制（暂时还没有写检测死锁的模块）。很多很好的设计思想，也是借鉴自orange系统的实现，我再结合着我自己的想法，还有我原来操作系统的实现，对代码做了适当的调整，调了蛮长时间的，终于写完了这一部分的实验，并且经过反复验证，暂时证实了自己代码能够正常地运作。

在实现进程间通信后，我实验的现象空间便大了许多，之后硬盘驱动也有希望了，读取文件也就有了希望，内存管理，页表也可以脱离出来使用一个单独的进程来进行管理了。一直想写一个fork()系统调用，我想之后很快，fork()也可以完成了，信号量的实现，也很快的，操作系统实验，虽然真的花了我很长时间，甚至可能有点耽误到期末的复习时间，还有其他科目的大项目，不过收获真的满满，可以说是最具性价比的一学分了😁~