东莞理工学院

操作系统课程设计报告

院　　系：计算机学院

班　　级：14 软卓

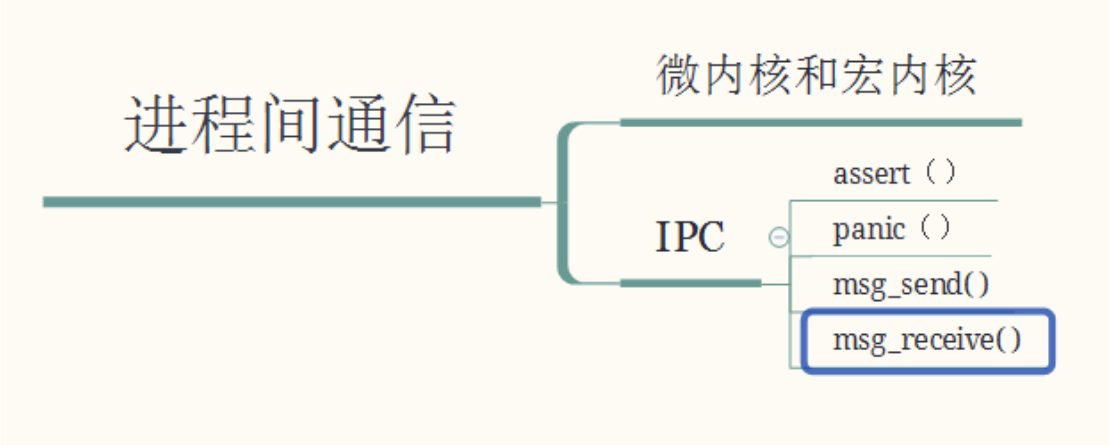
姓　　名：赖键锋

学　　号：201441402130

指导老师：李伟

日　　期：2016.6 - 2016.7

一、 相关说明



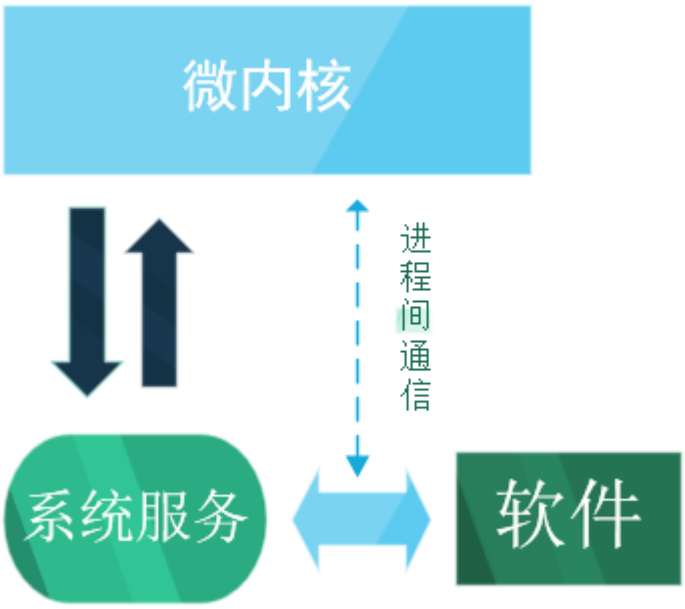
二、 相关知识的记录和说明

1. 宏内核 是将所有服务功能集成于一身,使用时直接调用. 宏内核的系统有 Unix,Linux,etc.

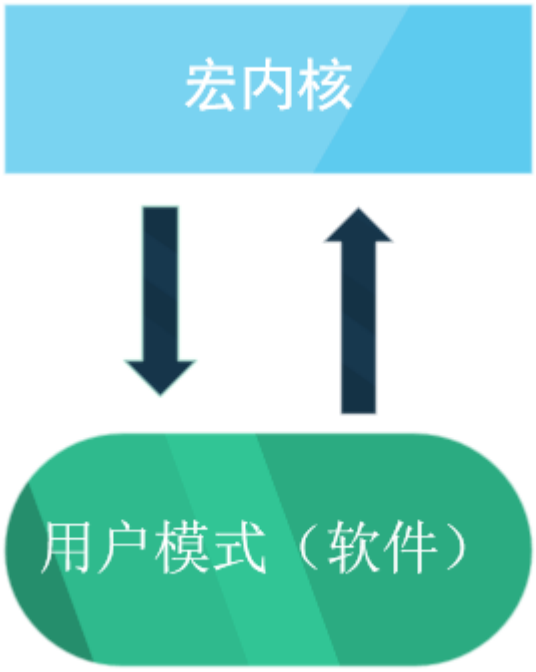
   微内核 是将各种服务功能放到内核之外,自身仅仅是一个消息中转战,用于各种功能间的通讯. 微内核的系统有 WindowNT,Minix,Mach ,etc.

2. 宏内核与微内核的区别:

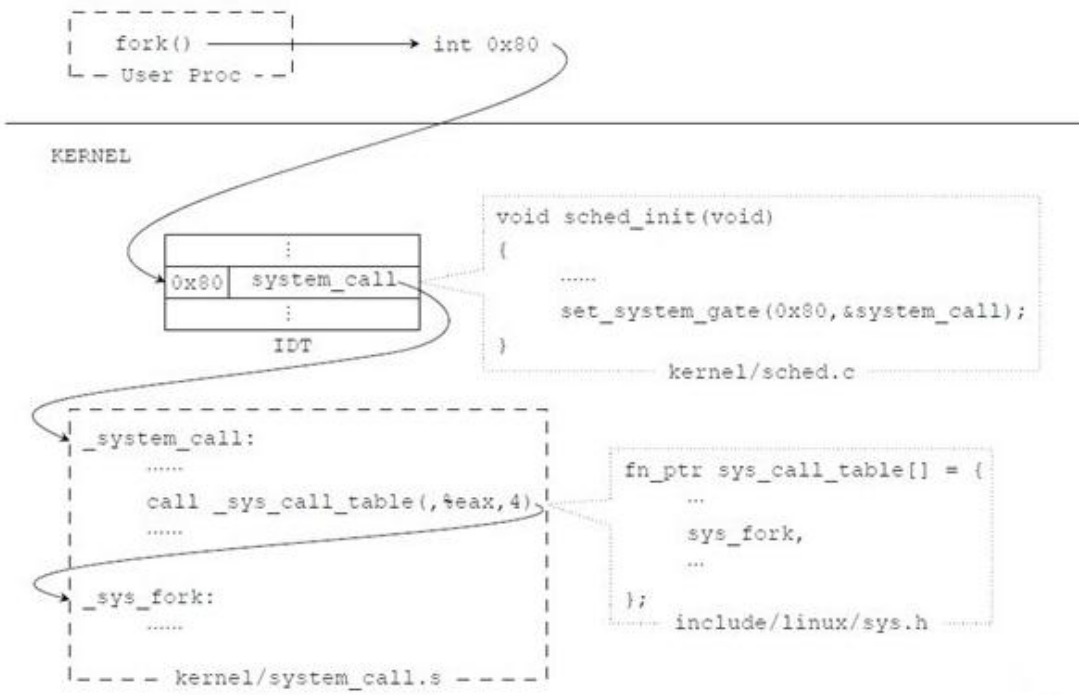
   微内核：信息中转站，自身完成很少功能，主要是传递一个模块对另一个模块的功能请求;从理论上来看,微内核的思想更好些，微内核把系统分为各个小的功能块，降低了设计难度，系统的维护与修改也容易，但通信带来的效率损失是个问题。



宏内核：大主管，把内存管理，文件管理等等全部接管；宏内核的功能块之间的耦合度太高造成修改与维护的代价太高，但 Linux 目前还不算太复杂；宏内核因为是直接调用，所以效率是比较高的。



3. Linux 的 fork 调用：



Linux 0.01 的 fork 系统调用

Minix 的 fork 调用，user_proc 和 MM 都是通过 sys_call，而 sys_call 最终把真正的处理分配给其他处理函数做。

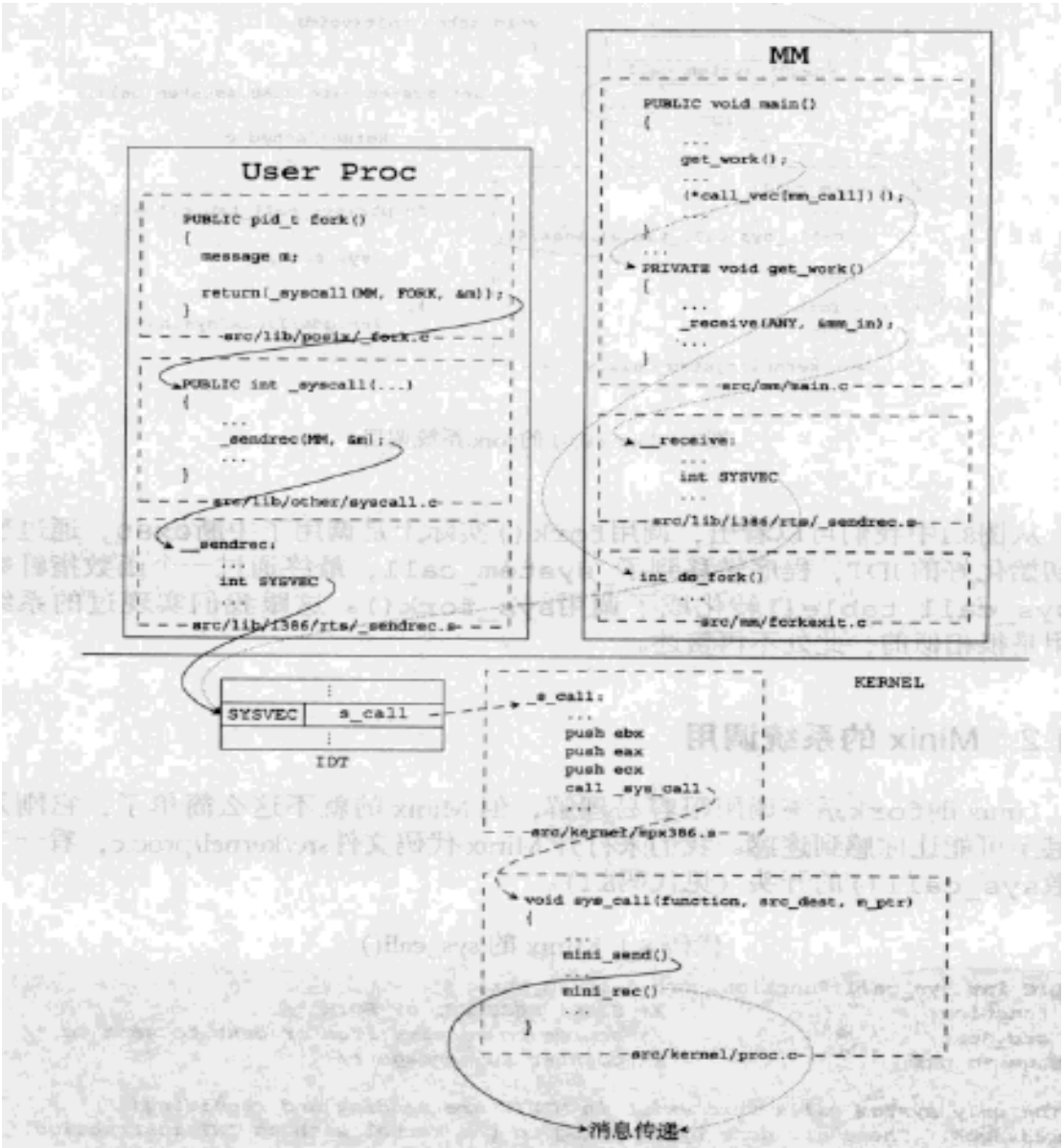


图 8.2 Minix 的 fork“系统调用”

4. IPC：进程间通信：

同步 IPC：同步好比走路，当左脚迈出后，会等待你的右脚迈出去，不然你的左脚只能等待（一般人不会连续两次迈左脚）。

异步 IPC： 异步则不会相互等待。

5. Assert()和 panic()是断言函数

- Assert（）函数：这是个宏调用，最开始没看懂!!!

```
#ifndef _ORANGES_CONST_H_
#define _ORANGES_CONST_H_

/* the assert macro */
#define ASSERT
#ifdef ASSERT
void assertion_failure(char *exp, char *file, char *base_file, int line);

#define assert(exp)  if (exp) ; \
        else assertion_failure(#exp, __FILE__, __BASE_FILE__, __LINE__)

#else
#define assert(exp)
#endif
```

宏调用assert(exp)中，如果exp为真，则空操作，exp为假，则调用assertion_failure()将错误的位置信息打印出来

```
#define assert(exp)   if (exp) ; ₩
        else assertion_failure(#exp, __FILE__, __BASE_FILE__, __LINE__)
```

注解：宏调用 assert(exp)会被 宏替换 为 if-else 语句，

如果 exp 为真，则为空操作；

如果 exp 为假，则调用 assertion_failure()将错误的位置信息打印出来。

```
PUBLIC void assertion_failure(char *exp, char *file, char *base_file, int line)
{
    printl("%c  assert(%s) failed: file: %s, base_file: %s, ln%d",
           MAG_CH_ASSERT,
           exp, file, base_file, line);

    spin("assertion_failure()");

    /* should never arrive here */
        __asm__ __volatile__("ud2");
}
```

- Panic()处于 ring1 或 ring0 层，当发生 panic 严重错误时，直接叫停整个系统。

```
159   PUBLIC void panic(const char *fmt, ...)
160   {
161       int i;
162       char buf[256];
163
164       /* 4 is the size of fmt in the stack */
165       va_list arg = (va_list)((char*)&fmt + 4);
166
167       i = vsprintf(buf, fmt, arg);
168
169       printl("%c !!panic!! %s", MAG_CH_PANIC, buf);
170
171       /* should never arrive here */
172       __asm__ __volatile__("ud2");
173   }
174
```

6. Sys_Sendrec():通过 assert 确保不是处于 ring0 等条件，如果是 Send 消息，则转发给 msg_send（）处理，如果是 Receive 消息，则转发给 msg_receive（）处理。

```
66    PUBLIC int sys_sendrec(int function, int src_dest, MESSAGE* m, struct proc* p)
67    {
68        assert(k_reenter == 0); /* make sure we are not in ring0 */
69        assert((src_dest >= 0 && src_dest < NR_TASKS + NR_PROCS) ||
70               src_dest == ANY ||
71               src_dest == INTERRUPT);
72        int ret = 0;
73        int caller = proc2pid(p);
74        MESSAGE* mla = (MESSAGE*)va2la(caller, m);
75        mla->source = caller;
76        assert(mla->source != src_dest);
77
78        if (function == SEND) {
79            ret = msg_send(p, src_dest, m);
80            if (ret != 0)
81                return ret;
82        }else if (function == RECEIVE) {
83            ret = msg_receive(p, src_dest, m);
84            if (ret != 0)
85                return ret;
86        }else {
87            panic("{sys_sendrec} invalid function: "
88                  "%d (SEND:%d, RECEIVE:%d).", function, SEND, RECEIVE);
89        }
90        return 0;
91    }
```

7. 几个用到的函数

Ldt_seg_linear()：给出指定进程和第 idx 个 ldt，返回该 ldt 的段基址。

```
146  PUBLIC int ldt_seg_linear(struct proc* p, int idx)
147  {
148      struct descriptor * d = &p->ldts[idx];
149
150      return d->base_high << 24 | d->base_mid << 16 | d->base_low;
151  }
```

Va2la()：根据进程号 pid 获取进程的指针，再调用 ldt_seg_linear（）获取基地址，再加上虚拟地址就得到了　线性地址。

但是实在是找不出 INDEX_LDT_RW 这个索引是哪里来的!!!

```
164  PUBLIC void* va2la(int pid, void* va)
165  {
166      struct proc* p = &proc_table[pid];
167
168      u32 seg_base = ldt_seg_linear(p, INDEX_LDT_RW);
169      u32 la = seg_base + (u32)va;
170
171      if (pid < NR_TASKS + NR_PROCS) {
172          assert(la == (u32)va);
173      }
174
175      return (void*)la;
176  }
177
```

Block():在调用此函数前，p 的 flag 标志位必须已经被设置为 1，才调度其他进程。

```
203  PRIVATE void block(struct proc* p)
204  {
205      assert(p->p_flags);
206      schedule();
207  }
```

Unblock（）：调用此函数前，flag 必须已经被置为 0

```
218   PRIVATE void unblock(struct proc* p)
219   {
220       assert(p->p_flags == 0);
221   }
```

Deadlock（）：发生死锁时进程的状态：循环等待，进程间构成一个环

```
238   PRIVATE int deadlock(int src, int dest)
239   {
240       struct proc* p = proc_table + dest; //p设为链的最后一个进程
241       while (1) {
242           if (p->p_flags & SENDING) {     //如果p处于正在发送的状态，则可能发生了 死锁
243               if (p->p_sendto == src) {   //头尾相连，构成一个环，发生死锁
244                   /* print the chain */
245                   p = proc_table + dest;  //输出这个死锁环
246                   printl("=_=%s", p->name);
247                   do {
248                       assert(p->p_msg);
249                       p = proc_table + p->p_sendto;   //消息的接收对象，相当于指针next
250                       printl("->%s", p->name);
251                   } while (p != proc_table + src);
252                   printl("=_=");
253
254                   return 1;                   //return 1
255               }
256               p = proc_table + p->p_sendto;
257           }else {                             //没有处于正在发送的状态，则不是 死锁
258               break;
259           }
260       }
261       return 0;                               //return  0
262   }
263
```

8. 进程表添加的新成员

P_flags：用于标志进程的状态：

- 0 -- 正在运行或准备运行
- Sending -- 正在发送消息，消息未送达，被阻塞
- Receiving -- 正在接收消息，消息未到达，被阻塞

P_msg：指向消息体的指针

P_recvfrom：进程想要从哪个进程接收消息，就指向那个进程

P_sendto：要发送消息给哪个进程，就指向那个进程

has_int_msg：进程等待的中断发生后就置为 1，表中断已发生

q_sending：指向要发送消息给本进程的进程队列的第一个进程，进程队列由 next_sending 连接，next_sending 指向下一个发送消息的进程。

9. Msg_send()：IPC 的重点代码

phys_copy（）就是消息传递的最主要的函数，将消息从发送消息进程的消息存放区复制到接收消息进程的消息存放区；在复制的前后要做很多条件判断工作。

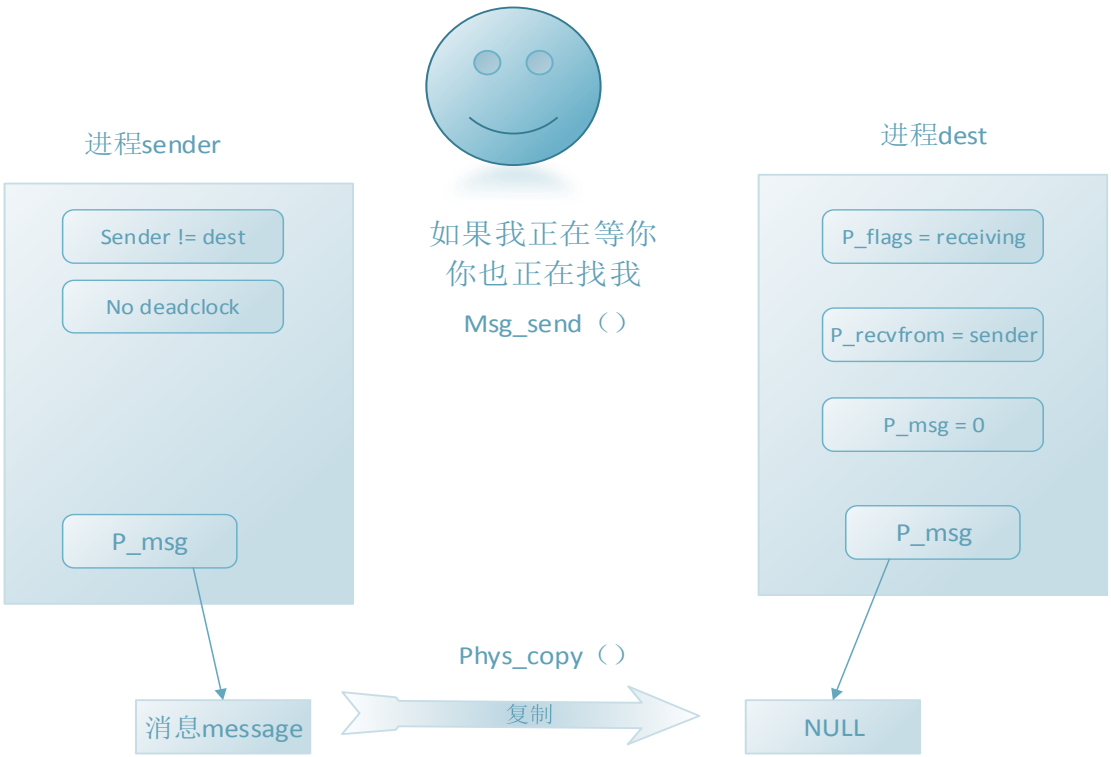
Phys_copy 是 memcpy 的宏，在 include/string.h 中定义：

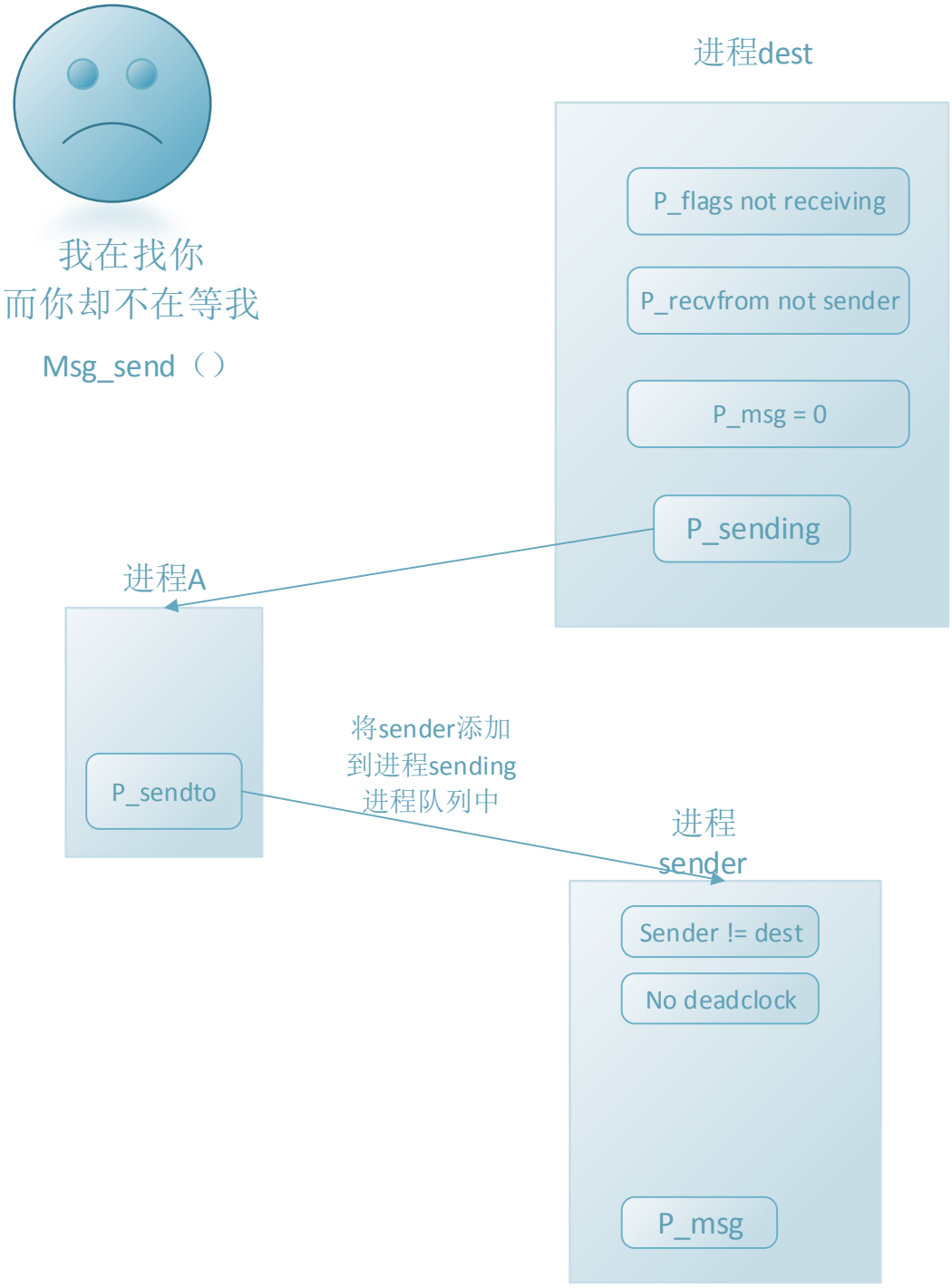
```
18    #define phys_copy   memcpy
19    #define phys_set    memset
20
```

而 memcpy 是内存赋值函数，将 src 的 size 字节复制到 dst 中（lib/string.asm）

```
; ------------------------------------------------------------------------
; void* memcpy(void* es:p_dst, void* ds:p_src, int size);
; ------------------------------------------------------------------------
memcpy:
    push    ebp
    mov ebp, esp

    push    esi
    push    edi
    push    ecx

    mov edi, [ebp + 8]  ; Destination
    mov esi, [ebp + 12] ; Source
    mov ecx, [ebp + 16] ; Counter
.1:
    cmp ecx, 0          ; 判断计数器
    jz  .2          ; 计数器为零时跳出

    mov al, [ds:esi]            ; ┓
    inc esi                 ; ┃
                            ; ┣ 逐字节移动
    mov byte [es:edi], al   ; ┃
    inc edi                 ; ┛

    dec ecx         ; 计数器减一
    jmp .1          ; 循环
.2:
    mov eax, [ebp + 8]  ; 返回值

    pop ecx
    pop edi
    pop esi
    mov esp, ebp
    pop ebp

    ret             ; 函数结束，返回
; memcpy 结束-------------------------------------------------------------
```

两种情况：





注意，这里有一个很重要的点，就是它是如何实现进程切换的。

我们调用发送消息是通过系统中断来实现，在中断处理程序 sys_call 中的 save() 函数，会根据当前的 k_reenter 的值来分别将不同的东西压栈，而压入的内容，其实就是最终的进程切换程序 restart() ，并且最终在 sys_call 的最后一句 ret 来跳转到这个进程切换程序。

10. Msg_receive()（直接抄书里的）

假设有进程 B 想要接收消息（来自特定进程、中断或任意进程），那么过程将会是这样的：

1. B 准备一个空的消息结构体 M，用于接收消息。
2. B 通过系统调用 sendrec，最终调用 msg_receive。
3. 判断 B 是否有个来自硬件的消息（通过 has_int_msg 判断），如果是，并且 B 准备接收来自中断的消息或或准备接收任意消息，则马上准备一个消息给 B，并返回。
4. 如果 B 想接收来自任意进程的消息，则从自己的发送队列中选取第一个（如果队列非空的话），将其消息复制给 M。
5. 如果 B 向接收来自特定进程 A 的消息，则判断 A 是否正在等待向 B 发送消息，若是的话，将其消息复制给 M。
6. 如果此时没有任何进程发消息给 B，B 会被阻塞。

11. 增加消息机制后的进程调度

```
31  PUBLIC void schedule()
32  {
33      struct proc*    p;
34      int     greatest_ticks = 0;
35
36      while (!greatest_ticks) {
37          for (p = &FIRST_PROC; p <= &LAST_PROC; p++) {
38              if (p->p_flags == 0) {
39                  if (p->ticks > greatest_ticks) {
40                      greatest_ticks = p->ticks;
41                      p_proc_ready = p;
42                  }
43              }
44          }
45
46          if (!greatest_ticks)
47              for (p = &FIRST_PROC; p <= &LAST_PROC; p++)
48                  if (p->p_flags == 0)
49                      p->ticks = p->priority;
50      }
51  }
52
```

被阻塞的进程不会被调度

12. 用 IPC 替换系统调用 get_ticks（）

用户调用 get_ticks（）

```
112   PUBLIC int get_ticks()
113   {
114       MESSAGE msg;
115       reset_msg(&msg);
116       msg.type = GET_TICKS;
117
118       send_recv(BOTH, TASK_SYS, &msg);
119
120       return msg.RETVAL;
121   }
122
```

新建并清空消息体

传递的是消息体的引用

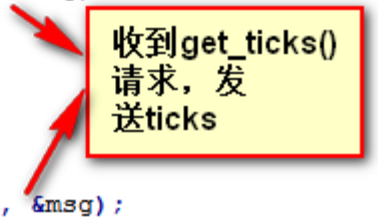
调用 send_recv（）

```
119   PUBLIC int send_recv(int function, int src_dest, MESSAGE* msg)
120   {
121       int ret = 0;
122       if (function == RECEIVE)
123           memset(msg, 0, sizeof(MESSAGE));
124       switch (function) {
125       case BOTH:
126           ret = sendrec(SEND, src_dest, msg);
127           if (ret == 0)
128               ret = sendrec(RECEIVE, src_dest, msg);
129           break;
130
131       case SEND:
132       case RECEIVE:
133           ret = sendrec(function, src_dest, msg);
134           break;
135
136       default:
137           assert((function == BOTH) ||
138                  (function == SEND) || (function == RECEIVE));
139           break;
140       }
141       return ret;
142   }
```

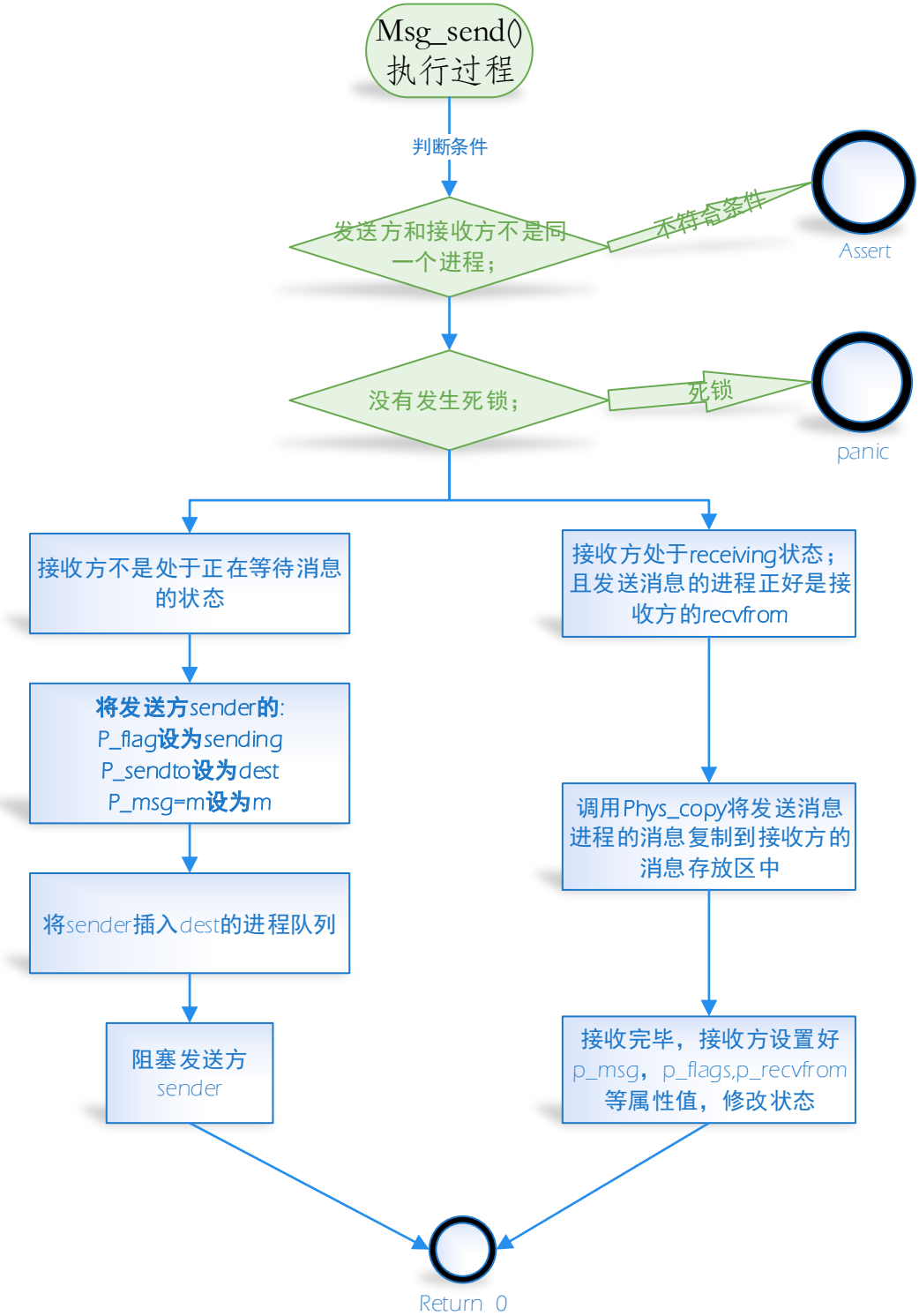
both:先发送，再接收

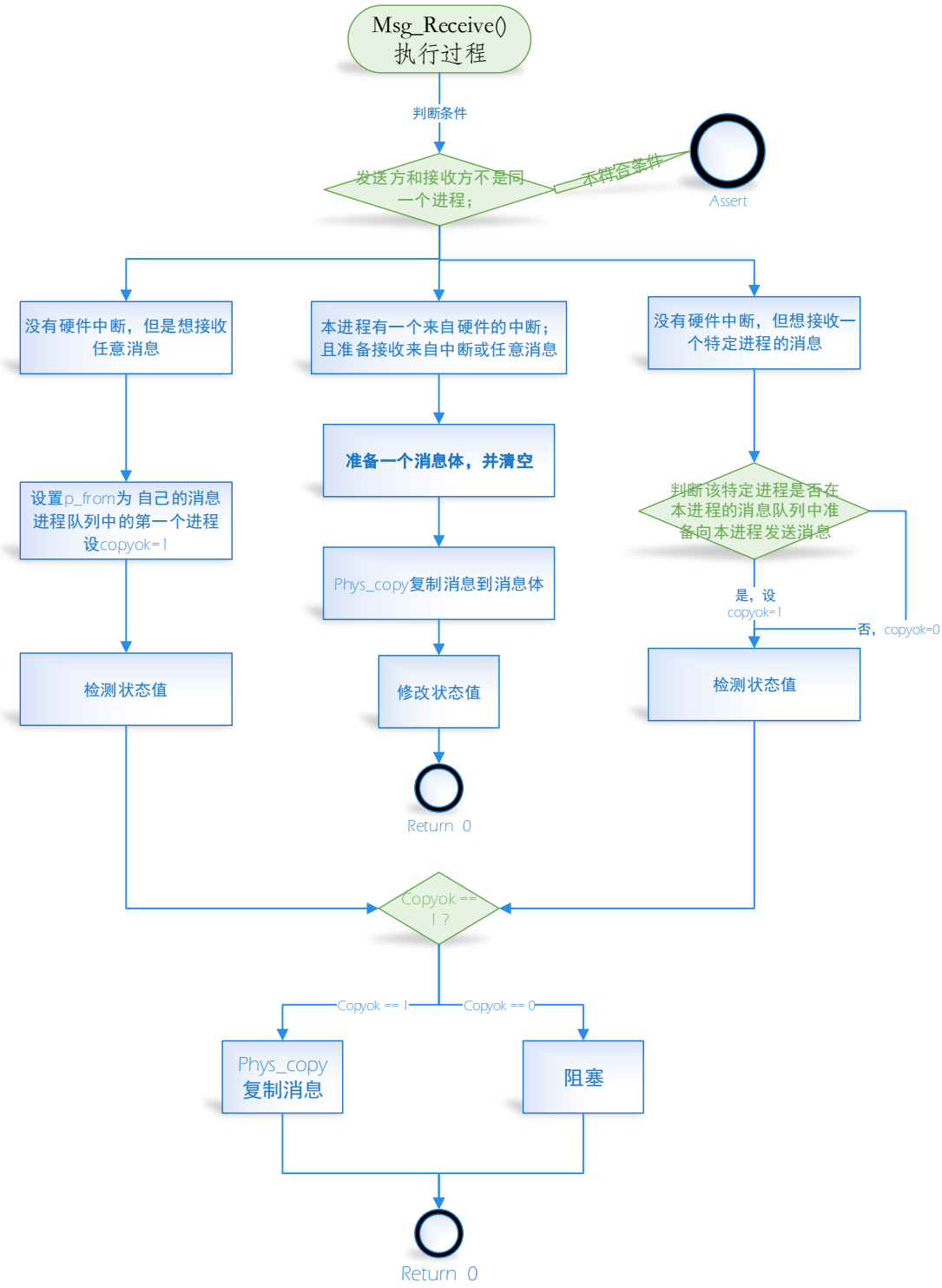
而 TASK_SYS 端：

```
29  PUBLIC void task_sys()
30  {
31      MESSAGE msg;
32      while (1) {
33          send_recv(RECEIVE, ANY, &msg);
34          int src = msg.source;
35
36          switch (msg.type) {
37          case GET_TICKS:
38              msg.RETVAL = ticks;
39              send_recv(SEND, src, &msg);
40              break;
41          default:
42              panic("unknown msg type");
43              break;
44          }
45      }
46  }
47
```

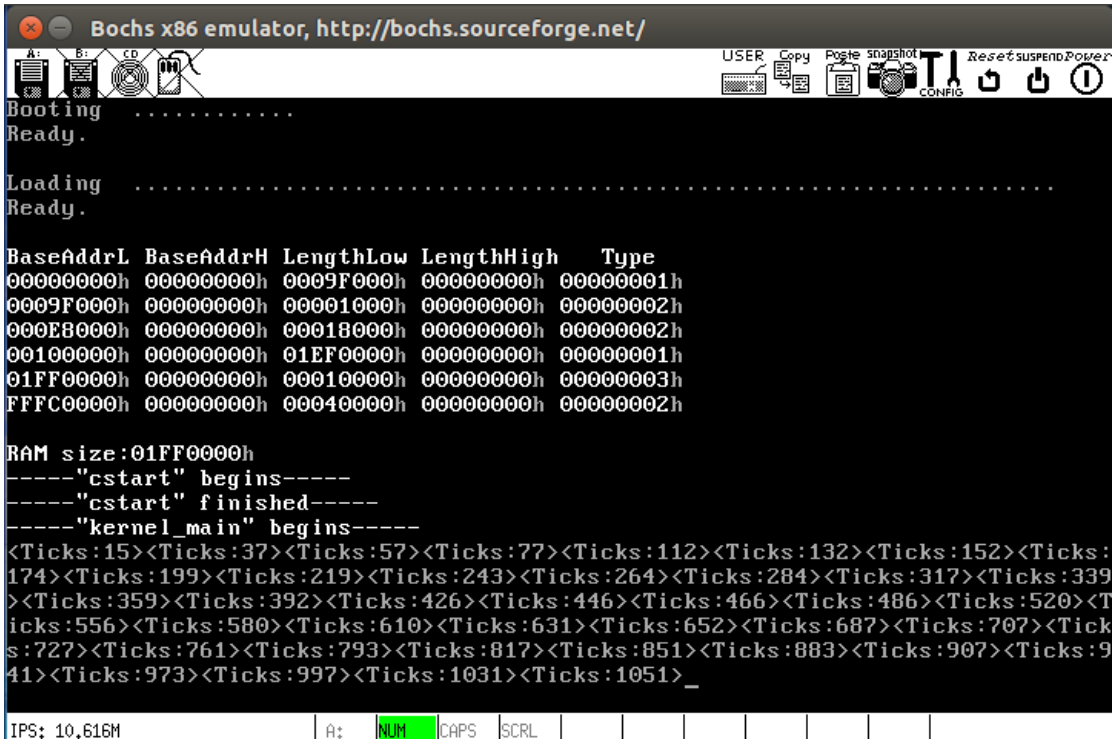


三、 程序关系图或流程图


Msg_send()
执行过程
判断条件
发送方和接收方不是同一个进程；
不符合条件
Assert
没有发生死锁；
死锁
panic
接收方不是处于正在等待消息的状态
将发送方sender的:
P_flag设为sending
P_sendto设为dest
P_msg=m设为m
将sender插入dest的进程队列
阻塞发送方
sender
接收方处于receiving状态；且发送消息的进程正好是接收方的recvfrom
调用Phys_copy将发送消息进程的消息复制到接收方的消息存放区中
接收完毕，接收方设置好p_msg，p_flags,p_recvfrom等属性值，修改状态
Return 0


Msg_Receive()
执行过程
判断条件
发送方和接收方不是同一个进程；
不符合条件
Assert
没有硬件中断，但是想接收任意消息
本进程有一个来自硬件的中断；且准备接收来自中断或任意消息
没有硬件中断，但想接收一个特定进程的消息
设置p_from为 自己的消息进程队列中的第一个进程
设copyok=1
准备一个消息体，并清空
判断该特定进程是否在本进程的消息队列中准备向本进程发送消息
Phys_copy复制消息到消息体
是，设copyok=1
否，copyok=0
检测状态值
修改状态值
检测状态值
Return 0
Copyok == 1 ?
Copyok == 1
Copyok == 0
Phys_copy
复制消息
阻塞
Return 0

四、 运行过程及理解



五、 重点知识总结