目 录

1 实现 IPC

实现框架如下：

1. 增加 sendrec 系统调用，这里的系统调用供进程调用，最终根据类型决定调用 msg_receive() 还是 msg_send() 函数。
2. 新定义 MESSAGE 结构体和扩展 proc 结构体，并且声明一个 phys_copy() 函数，用于进程之间复制消息体。
3. 实现 msg_send() 函数，用于将 sender 进程的消息体内容传递给目标进程。
4. 实现 msg_receive() 函数，用于接收来自其它进程的消息。

1.1 增加 sendrec 系统调用

两个函数体分别如下：

```
sendrec:
    mov eax, _NR_sendrec
    mov ebx, [esp + 4]
    mov ecx, [esp + 8]
    mov edx, [esp + 12]
    int INT_VECTOR_SYS_CALL
    ret

// ==============================

PUBLIC int sys_sendrec(int function, int src_dest, MESSAGE* m, struct proc* p)
{
    assert(k_reenter == 0);
    assert((src_dest >= 0 && src_dest <= NR_TASKS + NR_PROCS) || src_dest ==
        ANY || src_dest == INTERRUPT);

    int ret = 0;
    int caller = proc2pid(p);
    MESSAGE* mla = (MESSAGE*)va2la(caller, m);
    mla->source = caller;

    assert(mla->source != src_dest);

    if(function == SEND)
    {
        ret = msg_send(p, src_dest, m);
        if(ret != 0)
            return ret;
    }
    else if(function == RECEIVE)
    {
        ret = msg_receive(p, src_dest, m);
        if(ret != 0)
```

```
33                  return ret;
34          }
35          else
36          {
37              panic("{sys_sendrec} invalid function: %d (SEND:%d RECEIVE:%d)",
                    function, SEND, RECEIVE);
38          }
39      }
```

我们这里还用了 send_recv() 函数去封装 sendrec() 系统调用：

```
1   PUBLIC int send_recv(int function, int src_dest, MESSAGE* msg)
2   {
3       int ret = 0;
4       if(function == RECEIVE)
5           memset(msg, 0, sizeof(MESSAGE));
6
7       switch(function)
8       {
9           case BOTH:
10              ret = sendrec(SEND, src_dest, msg);
11              if(ret == 0)
12                  ret = sendrec(RECEIVE, src_dest, msg);
13              case SEND:
14              case RECEIVE:
15                  ret = sendrec(function, src_dest, msg);
16                  break;
17              default:
18                  assert((function == BOTH) || (function == SEND) || (function ==
                        RECEIVE));
19                  break;
20      }
21      return ret;
22  }
```

1.2 为消息的接收与发送做准备

1.2.1 扩展进程结构体

定义 MESSAGE 结构体：

```
1   struct mess1
2   {
3       int m1i1;
4       int m1i2;
5       int m1i3;
6       int m1i4;
7   };
8
9   struct mess2
10  {
11      void* m2p1;
```

```
        void* m2p2;
        void* m2p3;
        void* m2p4;
    };

    struct mess3
    {
        int m3i1;
        int m3i2;
        int m3i3;
        int m3i4;
        u64 m3l1;
        u64 m3l2;
        void*   m3p1;
        void*   m3p2;
    };

    #define RETVAL      u.m3.m3i1

    typedef struct
    {
        int source;
        int type;
        union {
            struct mess1 m1;
            struct mess2 m2;
            struct mess3 m3;
        } u;
    } MESSAGE;
```

接收方和发送方斗维护着一个消息结构体，发送方的结构体携带了消息内容，而接收方是空的。

为了使进程可以通信，我们要在进程体结构中增加几个成员：

```
    struct proc
    {
        struct stackframe regs;

        u16 ldt_sel;
        struct descriptor ldts[LDT_SIZE];

        int ticks;
        int priority;

        u32 pid;
        char name[16];

        int nr_tty;

        // 以下是扩展的成员
        int p_flags;
        /*
           用于表明进程的状态
           0，表示进程正在运行或准备运行
```

```
21              SENDING，进程处于发送消息的状态，消息还未送达，进程被阻塞
22              RECEIVING。进程处于接收消息的状态，消息还未收到，进程被阻塞
23          */
24
25          MESSAGE * p_msg; // 指向消息体
26
27          int p_recvfrom;
28          /*
29              记录进程想要从谁那里接收消息
30          */
31
32          int p_sendto;
33          /*
34              记录进程想要发送消息给谁
35          */
36
37          int has_int_msg;
38          /*
39               系统是否正在等待一个中断发生
40          */
41
42          struct proc * q_sending;
43          /*
44               向进程发送消息的进程队列中，q_sending指向第一个试图发送消息的进程
45          */
46
47          struct proc * next_sending;
48          /*
49               向进程发送消息的进程队列中，进程如果处在这个队列中，next_sending指向下
                 一个进程
50          */
51      }
```

1.2.2 实现 phys_copy() 函数

这里的 phys_copy() 函数可以直接借助 memcpy() 函数来实现：

```
1   PUBLIC  void*   memcpy(void* p_dst, void* p_src, int size);
2
3   #define phys_copy   memcpy
```

为了使用 phys_copy() 函数，我们需要先把消息的地址转为线性地址，算法如下：

1. 根据上述进程结构体的定义，消息 MESSAGE 是 proc 结构体的一个成员，如果我们拥有一个 MESSAGE，那么它一定是附属于进程结构体的。那么这个消息的地址值一个是相对于进程结构体的偏离地址。
2. 现在我们拥有相对于进程结构体的偏离地址，只需要再求出进程结构体的线性地址，两者相加就是消息的线性地址。

```
// 每个进程都有自己的LDT
// 通过进程结构体中的LDT中的描述符可以得到相应段的基地址
PUBLIC int ldt_seg_linear(struct proc* p, int idx)
{
    struct descriptor* d = &p->ldts[idx];
    return d->base_high << 24 | d->base_mid << 16 | d->base_low;
}

PUBLIC void* va2la(int pid, void* va)
{
    struct proc* p = &proc_table[pid];

    u32 seg_base = ldt_seg_linear(p, INDEX_LDT_RW);
    u32 la = seg_base + u32(va);

    return (void*)la;
}
```

1.3 实现 msg_send() 函数

这个函数的算法如下：

1. 首先判断是否发生死锁。
2. 判断目标进程 dest 是否正在等待 sender 进程的消息。
3. 如果是，就把消息复制给目标进程，目标进程被解除阻塞，继续运行。如果不是，sender 进程被阻塞，并加入目标进程的发送队列中。

```
PRIVATE int msg_send(struct proc* current, int dest, MESSAGE* m)
{
    struct proc* sender = current;
    struct proc* p_dest = proc_table + dest;

    assert(proc2pid(sender) != dest);

    // 检测是否发生死锁
    if(deadlock(proc2pid(), dest))
    {
        panic(">>DEADLOCK<< %s->%s", sender->name, p_dest->name);
    }

    // 判断目标进程p_dest是否在等待sender进程的消息
    if((p_dest->p_flags & RECEIVING) && (p_dest->p_recvfrom == proc2pid(sender)
        ) || p_dest->p_recvfrom == ANY)
    {
        assert(p_dest->p_msg);
        assert(m);

        // 将消息复制给目标进程p_dest
```

```
21              phys_copy(va2la(dest, p_dest->p_msg), va2la(proc2pid(sender), m),
                    sizeof(MESSAGE));
22
23              // 恢复p_dest的状态
24              p_dest->p_msg = 0;
25              p_dest->p_flags &= ~RECEIVING;
26              p_dest->p_recvfrom = NO_TASK;
27
28              // 将目标进程解除阻塞
29              unblock(p_dest);
30
31              assert(p_dest->p_flags == 0);
32              assert(p_dest->p_msg == 0);
33              assert(p_dest->p_recvfrom == NO_TASK);
34              assert(p_dest->p_sendto == NO_TASK);
35              assert(sender->p_flags == 0);
36              assert(sender->p_msg == 0);
37              assert(sender->p_recvfrom == NO_TASK);
38              assert(sender->p_sendto == NO_TASK);
39          }
40          else // 目标进程没有在等待sender进程，sender进程被阻塞，加入到目标进程的发
                送队列中
41          {
42              sender->p_flags |= SENDING;
43              assert(sender->p_flags == SENDING);
44              sender->p_sendto = dest;
45              sender->p_msg = m;
46
47              struct proc* p;
48
49              // 将sender进程加入目标进程的发送队列中
50              if(p_dest->q_sending)
51              {
52                  p = p_dest->q_sending;
53                  while(p->next_sending)
54                      p = p->next_sending;
55                  p->next_sending = sender;
56              }
57              else
58              {
59                  p_dest->q_sending = sender;
60              }
61              sender->next_sending = 0;
62
63              // 阻塞sender进程
64              block(sender);
65
66              assert(sender->p_flags == SENDING);
67              assert(sender->p_msg != 0);
68              assert(sender->p_recvfrom == NO_TASK);
69              assert(sender->p_sendto == dest);
70          }
71          return 0;
72      }
```

1.3.1 实现一些辅助函数

在实现 msg_send() 函数的过程中，我们使用了 block() 和 unblock() 函数，用于阻塞和解锁进程。在这里可以借助进程调度来实现进程的阻塞与解锁：

```
PRIVATE void block(struct proc* p)
{
    assert(p->p_flags); // 首先判断进程的状态不为0，也就是不是runnable的状态
    schedule();
}

PRIVATE void unblock(struct proc* p)
{
    assert(p->p_flags == 0); // 判断进程为runnable的状态
}
```

这里 unblock() 和 block() 的函数很简单，主要是利用 p_flags 这个状态，然后在进程调度函数中做一些手脚：

```
PUBLIC void schedule()
{
    struct proc* p;
    int greatest_ticks = 0;

    while(!greatest_ticks)
    {
        for(p = &FIRST_PROC; p <= &LAST_PROC; p++)
        {
            // 进程的p_flags只有为0，才可能分配到cpu
            if(p->p_flags == 0)
            {
                if(p->ticks > greatest_ticks)
                {
                    greatest_ticks = p->ticks;
                    p_proc_ready = p;
                }
            }
        }

        // 如果进程初始ticks都为0
        if(!greatest_ticks)
        {
            for(p = &FIRST_PROC; p <= &LAST_PROC; ++p)
            {
                // 只有进程的p_flags只有为0，才可能分配到ticks
                if(p->p_flags == 0)
                    p->ticks = p->priority;
            }
        }
    }
}
```

在实现消息传递机制中，我们还要谨防死锁。这里通过判断消息的发送是否构成一

个环来判断。如果构成一个环，则意味着发生死锁，比如 A 试图发消息给 B，同时 B 试图给 C，C 试图给 A 发消息，那么死锁就发生了。deadlock() 的实现如下：

```
PRIVATE int deadlock(int src, int dest)
{
    struct proc* p = proc_table + dest;
    while(1)
    {
        // 用于检测是否构成一个环
        if(p->p_flags & SENDING)
        {
            if(p->p_sendto == src)
            {
                // 如果发现可以构成一个环时，将这个进程环打印出来
                p = proc_table + dest;
                printl("=_=%s",p->name);
                do
                {
                    assert(p->msg);
                    p = proc_table + p->p_sendto;
                    printl("->%s", p->name);
                }while(p != proc_table + src);
                return 1;
            }
            p = proc_table + p->p_sendto;
        }
        else
            break;
    }
    return 0;
}
```

1.4 实现 msg_receive() 函数

这个函数的算法如下：

1. 首先判断进程是否有一个来自硬件的消息，如果是，并且进程的消息源为 ANY 或 INTERRUPT，就准备一个消息给进程，并返回。
2. 如果进程的消息源为 ANY，就从自己的 q_sending 中选取一个消息源，将其该源进程的消息复制给进程。
3. 如果进程的消息源为特定进程 A，则先判断 A 是否在等待向自己发送消息。如果是，就把消息复制给进程。
4. 如果此时没有任何进程发消息给本进程，则该进程将被阻塞。

```
PRIVATE int msg_receive(struct proc* current, int src, MESSAGE* m)
{
```

```
3         struct proc* p_who_wanna_recv = current;
4         struct proc* p_from = 0;
5         struct proc* prev = 0;
6         int copyok = 0;
7
8         assert(proc2pid(p_who_wanna_recv) != src);
9
10        // 判断进程是否有一个来自硬件的消息，并且进程的消息源为ANY或INTERRUPT
11        if((p_who_wanna_recv->has_int_msg) && ((src == ANY) || (src == INTERRUPT)))
12        {
13            MESSAGE msg;
14            reset_msg(&msg);
15            msg.source = HARD_INT;
16            assert(m);
17            phys_copy(va2la(proc2pid(p_who_wanna_recv), m), &msg, sizeof(MESSAGE));
18
19            p_who_wanna_recv->has_int_msg = 0;
20
21            assert(p_who_wanna_recv->p_flags == 0);
22            assert(p_who_wanna_recv->p_msg == 0);
23            assert(p_who_wanna_recv->p_sendto == NO_TASK);
24            assert(p_who_wanna_recv->has_int_msg == 0);
25
26            return 0;
27        }
28
29        // 如果进程的消息源为ANY
30        if(src == ANY)
31        {
32            if(p_who_wanna_recv->q_sending)
33            {
34                // 从自己的q_sending中选取一个消息
35                p_from = p_who_wanna_recv->q_sending;
36                copyok = 1;
37
38                assert(p_who_wanna_recv->p_flags == 0);
39                assert(p_who_wanna_recv->p_msg == 0);
40                assert(p_who_wanna_recv->p_recvfrom == NO_TASK);
41                assert(p_who_wanna_recv->p_sendto == NO_TASK);
42                assert(p_who_wanna_recv->q_sending != 0);
43
44                assert(p_from->p_flags == SENDING);
45                assert(p_from->p_msg != 0);
46                assert(p_from->p_recvfrom == NO_TASK);
47                assert(p_from->p_sendto == proc(p_who_wanna_recv));
48            }
49        }
50        else // 如果进程的消息源为特定进程
51        {
52            p_from = &proc_table[src];
53
54            // 判断进程是否在发消息，并且目标为本进程
55            if((p_from->p_flags & SENDING) && (p_from->p_sendto == proc2pid(
                  p_who_wanna_recv)))
56            {
57                copyok = 1;
```

```
58
59                  struct proc* p = p_who_wanna_recv->q_sending;
60                  assert(p);
61
62                  // 该循环用于找到进程队列中源进程的前一个进程prev，用于维护进程队列
63                  while(p)
64                  {
65                      assert(p_from->p_flags & SENDING);
66                      if(proc2pid(p) == src)
67                      {
68                          p_from = p;
69                          break;
70                      }
71                      prev = p;
72                      p = p->next_sending;
73                  }
74              }
75
76              assert(p_who_wanna_recv->p_flags == 0);
77              assert(p_who_wanna_recv->p_msg == 0);
78              assert(p_who_wanna_recv->p_recvfrom == NO_TASK);
79              assert(p_who_wanna_recv->p_sendto == NO_TASK);
80              assert(p_who_wanna_recv->q_sending != 0);
81
82              assert(p_from->p_flags == SENDING);
83              assert(p_from->p_msg != 0);
84              assert(p_from->p_recvfrom == NO_TASK);
85              assert(p_from->p_sendto == proc2pid(p_who_wanna_recv));
86          }
87
88          // 如果有相应的消息源
89          if(copyok)
90          {
91              // 如果p_from就是进程队列的第一个进程
92              if(p_from == p_who_wanna_recv->q_sending)
93              {
94                  assert(prev == 0);
95                  // 更新进程队列
96                  p_who_wanna_recv->q_sending = p_from->next_sending;
97                  p_from->next_sending = 0;
98              }
99              else // 如果不是，同样是维护进程队列
100             {
101                 assert(prev);
102                 prev->next_sending = p_from->next_sending;
103                 p_from->next_sending = 0;
104             }
105
106             assert(m);
107             assert(p_from->p_msg);
108
109             // 将消息体复制给目标进程
110             phys_copy(va2la(proc2pid(p_who_wanna_recv), m), va2la(proc2pid(p_from),
                        p_from->p_msg), sizeof(MESSAGE));
111
112             p_from->p_msg = 0;
```

```
113             p_from->p_sendto = NO_TASK;
114             p_from->p_flags &= ~SENDING; // 将p_from的p_flags设为0
115 
116             // 解除对p_from的阻塞
117             unblock(p_from);
118         }
119         else
120         {
121             // 将p_who_wanna_recv的p_flags设为RECEIVING
122             p_who_wanna_recv->p_flags |= RECEIVING;
123             p_who_wanna_recv->p_msg = m;
124 
125             if(src == ANY)
126                 p_who_wanna_recv->p_recvfrom = ANY;
127             else
128                 p_who_wanna_recv->p_recvfrom = proc2pid(p_from);
129 
130             // 阻塞目标进程p_who_wanna_recv
131             block(p_who_wanna_recv);
132 
133             assert(p_who_wanna_recv->p_flags == RECEIVING);
134             assert(p_who_wanna_recv->p_msg != 0);
135             assert(p_who_wanna_recv->p_recvfrom != NO_TASK);
136             assert(p_who_wanna_recv->p_sendto == NO_TASK);
137             assert(p_who_wanna_recv->has_int_msg == 0);
138         }
139     }
```

2 使用 IPC 机制实现 get_ticks() 函数

这里的 get_ticks() 函数首先需要向某个系统任务发出请求 ticks 的值，随后等待该系统任务的响应。这样的行为我们定义为 BOTH，也就是发送一个消息，随后马上等待接收一个消息。实现如下：

```
1   PUBLIC int get_ticks()
2   {
3       MESSAGE msg;
4       memset(&msg, 0, sizeof(MESSAGE));
5       msg.type = GET_TICKS;
6       send_recv(BOTH, TASK_SYS, &msg);
7       return msg.RETVAL;
8   }
9 
10  PUBLIC void task_sys()
11  {
12      MESSAGE msg;
13      while(1)
14      {
15          // 等待其他进程的消息
16          send_recv(RECEIVE, ANY, &msg);
17          int src = msg.source;
```

```
18
19              switch(msg.type)
20              {
21                  case GET_TICKS:
22                      msg.RETVAL = ticks;
23                      // 向特定进程发送消息
24                      send_recv(SEND, src, &msg);
25                      break;
26                  default:
27                      panic("unknown msg type");
28                      break;
29              }
30          }
31      }
```