# LwIP 在 ucOS 上的移植

经过几天调试除掉几个 bug 以后,ucos+lwip 在我的 44b0+8019 开发板上终于跑得比较稳定了.一只觉得 lwip 是一个不错的开放源码的 tcp/ip 协议栈,想把自己对 lwip 的移植和理解写出来.但是由于最近比较忙,lwip 的移植也是利用业余时间做的,今天写好了第一部分(lwip 的 process model)先贴上来,如果大家有兴趣我再接着往下写.另外我的移植参看了 skyeye 扬晔大侠的代码,大家可以去看看扬晔大侠的 lwip 在 ucos 上移植的文章和代码.

lwip 应用心得

lwIP 是瑞士计算机科学院（Swedish Institute of Computer Science）的 Adam Dunkels 等开发的一套用于嵌入式系统的开放源代码 TCP/IP 协议栈。Lwip 既可以移植到操作系统上,又可以
在无操作系统的情况下独立运行.

LwIP 的特性如下：

(1) 支持多网络接口下的 IP 转发
(2) 支持 ICMP 协议
(3) 包括实验性扩展的的 UDP（用户数据报协议）
(4) 包括阻塞控制，RTT 估算和快速恢复和快速转发的 TCP（传输控制协议）
(5) 提供专门的内部回调接口（Raw API）用于提高应用程序性能
(6) 可选择的 Berkeley 接口 API（多线程情况下）
(7) 在最新的版本中支持 ppp
(8) 新版本中增加了的 IP fragment 的支持.
(9) 支持 DHCP 协议,动态分配 ip 地址.

现在网上最新的版本是 V0.6.4

1.lwip 的进程模型(process model)

tcp/ip 协议栈的 process model 一般有几种方式.

1.tcp/ip 协议的每一层是一个单独进程.链路层是一个进程,ip 层是一个进程,tcp 层是一个进程.这样的好处是网络协
议的每一层都非常清晰,代码的调试和理解都非常容易.但是最大的坏处数据跨层传递时会引起上下文切换(context switch).

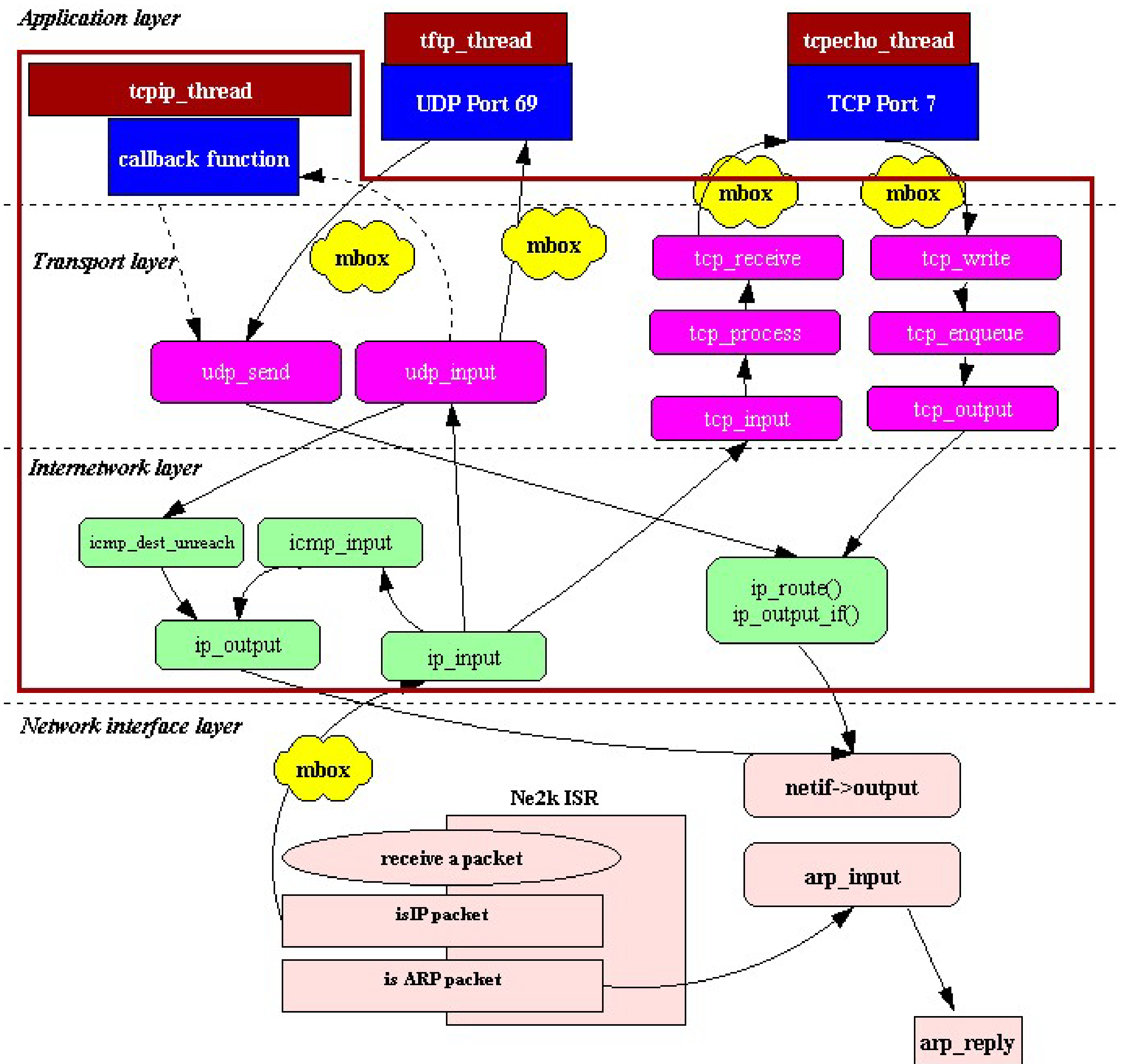
对于接收一个 TCP segment 要引起 3 次 context switch(从网卡驱动程序到链路层进程,从链路层进程到 ip 层进程,从 ip 层进程
到 TCP 进程).通常对于操作系统来说,任务切换是要浪费时间的.过频的 context swich 是不可取的.

2.另外一种方式是 TCP/IP 协议栈在操作系统内核当中.应用程序通过操作系统的系统调用(system call)和协议栈来进行通讯.

这样 TCP/IP 的协议栈就限定于特定的操作系统内核了.如 windows 就是这种方式.

3.lwip 的 process model:所有 tcp/ip 协议栈都在一个进程当中,这样 tcp/ip 协议栈就和操作系统内核分开了.而应用层程序既可以
是单独的进程也可以驻留在 tcp/ip 进程中.如果应用程序是单独的进程可以通过操作系统的邮箱,消息队列等和 tcp/ip 进程进行通讯.

如果应用层程序驻留 tcp/ip 进程中,那应用层程序就利用内部回调函数口(Raw API)和 tcp/ip 协议栈通讯.对于 ucos 来说进程就是一个系统任务.lwip 的 process model 请参看下图.在图中可以看到整个 tcp/ip 协议栈都在同一个任务(tcpip_thread)中.应用层程序既可以是独立的任务(如图中的 tftp_thread,tcpecho_thread),也可以在 tcpip_thread 中(如图左上角)中利用内部回调函数口(Raw API)和 tcp/ip 协议栈通讯




2 Port Lwip to uCos

在这个项目中我用的硬件平台是s3c44b0x+rtl8019.ucos在44b0上的移植在网上有很多大侠非常详尽的讲解和移植代码.我就不敢罗嗦了.需要说明的一点是 lwip 会为每个网络连接动态分配一些信号量(semaphone)和消息队列(Message Queue),当连接断开时会删掉这些 semaphone 和 Queue而 Ucos-2.0 不支持 semaphone 和 Queue 的删除,所以要选择一些较高版本的 ucos.我用的是 ucos-2.51.

2.1 Lwip 的操作系统封装层(operating system.emulation layer)

Lwip 为了适应不同的操作系统,在代码中没有使用和某一个操作系统相关的系统调用和数据结构.而是

在 lwip 和操作系统之间增加了一个操作系统封装层.操作系统封装层为操作系统服务(定时,进程同步,消息传递)提供了一个统一的接口.在 lwip 中进程同步使用 semaphone 和消息传递采用"mbox"(其实在 ucos 的实现中我们使用的是 Message Queue 来实现 lwip 中的"mbox",下面大家可以看到这一点)
Operating system emulation layer 的原代码在…/lwip/src/core/sys.c 中.而和具体的操作系统相关的代码在../lwip/src/arch/sys_arch.c 中.
操作系统封装层的主要函数如下:

```
void sys_init(void)//系统初始化
sys_thread_t sys_thread_new(void (* function)(void *arg), void *arg,int prio)//创建一个新进程
 sys_mbox_t sys_mbox_new(void)//创建一个邮箱
 void  sys_mbox_free(sys_mbox_t mbox)//释放并删除一个邮箱
 void  sys_mbox_post(sys_mbox_t mbox, void *data) //发送一个消息到邮箱
 void sys_mbox_fetch(sys_mbox_t mbox, void **msg)//等待邮箱中的消息
 sys_sem_t sys_sem_new(u8_t count)//创建一个信号量
void sys_sem_free(sys_sem_t sem)//释放并删除一个信号量
void sys_sem_signal(sys_sem_t sem)//发送一个信号量
void sys_sem_wait(sys_sem_t sem)//等待一个信号量
 void sys_timeout(u32_t msecs, sys_timeout_handler h, void *arg)//设置一个超时事件
 void sys_untimeout(sys_timeout_handler h, void *arg)//删除一个超时事件
 …
```

关于操作系统封装层的信息可以阅读 lwip 的 doc 目录下面的 sys_arch.txt.文件.

## 2.2 Lwip 在 ucos 上的移植.

### 2.2.1 系统初始化

sys_int 必须在 tcpip 协议栈任务 tcpip_thread 创建前被调用.

```
 #define MAX_QUEUES        20
#define MAX_QUEUE_ENTRIES   20
typedef struct {
      OS_EVENT*   pQ;//ucos 中指向事件控制块的指针
      void*  pvQEntries[MAX_QUEUE_ENTRIES];//消息队列
//MAX_QUEUE_ENTRIES 消息队列中最多消息数
} TQ_DESCR, *PQ_DESCR;
typedef PQ_DESCR  sys_mbox_t;//可见 lwip 中的 mbox 其实是 ucos 的消息队列
static char pcQueueMemoryPool[MAX_QUEUES * sizeof(TQ_DESCR) ];
  void sys_init(void)
{
   u8_t i;
   s8_t   ucErr;
   pQueueMem = OSMemCreate( (void*)pcQueueMemoryPool, MAX_QUEUES, sizeof(TQ_DESCR), &ucErr );//为消息队列创建内存分区
   //init lwip task prio offset
   curr_prio_offset = 0;
   //init lwip_timeouts for every lwip task
   //初始化 lwip 定时事件表,具体实现参考下面章节
   for(i=0;i<LWIP_TASK_MAX;i++){
      lwip_timeouts[i].next = NULL;
```

```
    }
}
```

2.2.2 创建一个和 tcp/ip 相关新进程:

lwip 中的进程就是 ucos 中的任务,创建一个新进程的代码如下:

```
#define LWIP_STK_SIZE      10*1024//和 tcp/ip 相关任务的堆栈大小.可以根据情况自
//己设置,44b0 开发板上有 8M 的 sdram,所以设大
//一点也没有关系:)
//max number of lwip tasks
#define LWIP_TASK_MAX    5 //和 tcp/ip 相关的任务最多数目
//first prio of lwip tasks
#define LWIP_START_PRIO   5 //和 tcp/ip 相关任务的起始优先级,在本例中优先级可
//以从(5-9).注意 tcpip_thread 在所有 tcp/ip 相关进程中//应该是优先级最高的.在本例中就是优先级 5
//如果用户需要创建和 tcp/ip 无关任务,如 uart 任务等,
//不要使用 5-9 的优先级
  OS_STK LWIP_TASK_STK[LWIP_TASK_MAX][LWIP_STK_SIZE];//和 tcp/ip 相关进程
//的堆栈区
   u8_t curr_prio_offset ;
   sys_thread_t sys_thread_new(void (* function)(void *arg), void *arg,int prio)
{
 if(curr_prio_offset < LWIP_TASK_MAX){
  OSTaskCreate(function,(void*)0x1111, &LWIP_TASK_STK[curr_prio_offset][LWIP_STK_SIZE-1],
LWIP_START_PRIO+curr_prio_offset );
  curr_prio_offset++;
  return 1;
 } else {
  // PRINT(" lwip task prio out of range ! error! ");
 }
}
```

从代码中可以看出 tcpip_thread 应该是最先创建的.

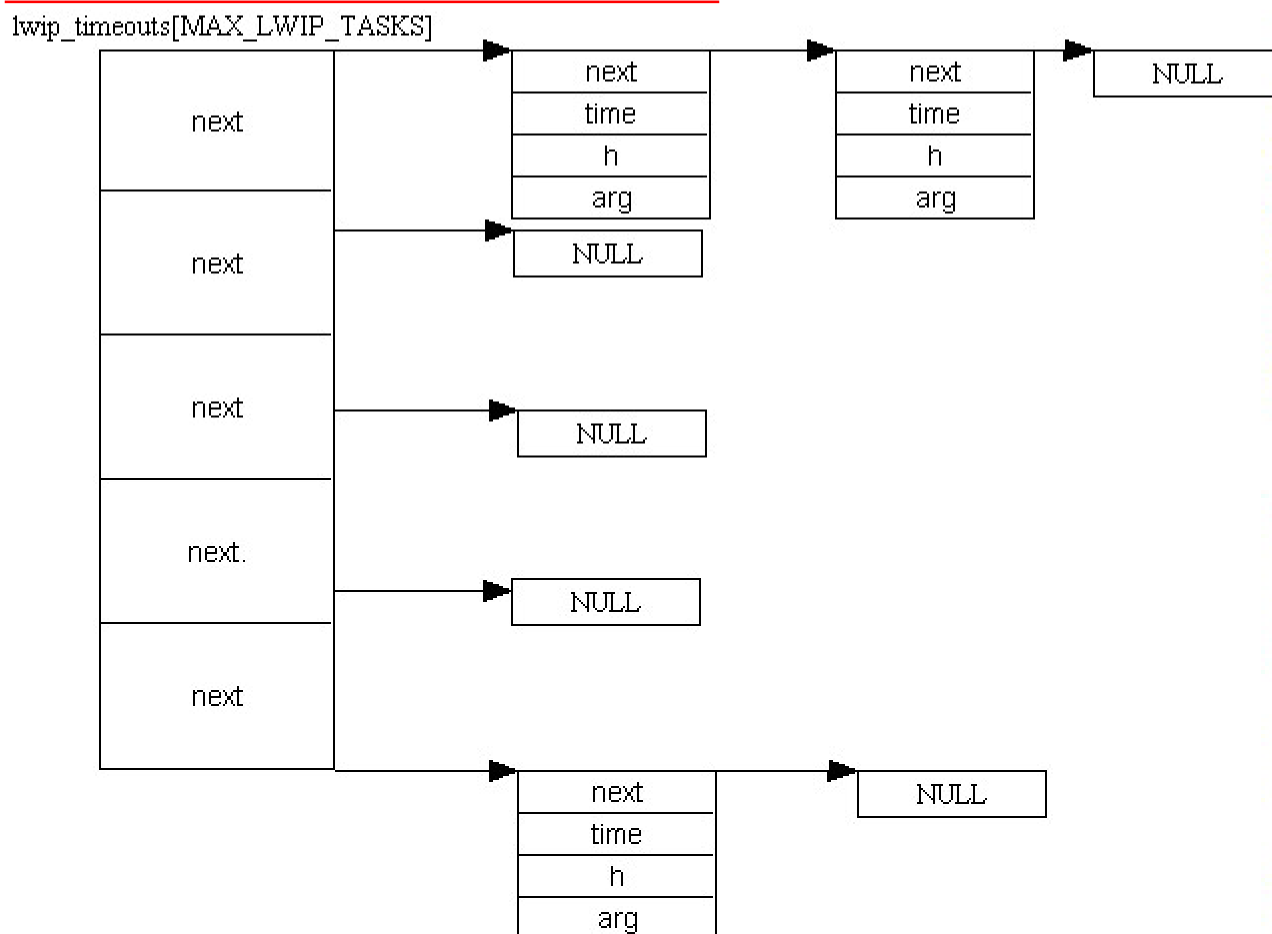
2.2.3 Lwip 中的定时事件

在 tcp/ip 协议中很多时候都要用到定时,定时的实现也是 tcp/ip 协议栈中一个重要的部分.lwip 中定时事件的数据结构如下.

```
struct sys_timeout {
 struct sys_timeout *next;//指向下一个定时结构
 u32_t time;//定时时间
 sys_timeout_handler h;//定时时间到后执行的函数
 void *arg;//定时时间到后执行函数的参数.
};
struct sys_timeouts {
 struct sys_timeout *next;
};
struct sys_timeouts lwip_timeouts[LWIP_TASK_MAX];
```

Lwip 中的定时事件表的结构如下图,每个和 tcp/ip 相关的任务的一系列定时事件组成一个单向链表.每个

链表的起始指针存在 lwip_timeouts 的对应表项中.




函数 sys_arch_timeouts 返回对应于当前任务的指向定时事件链表的起始指针.该指针存在 lwip_timeouts[MAX_LWIP_TASKS]中.

```
struct sys_timeouts null_timeouts;
struct sys_timeouts * sys_arch_timeouts(void)
{
 u8_t curr_prio;
 s16_t err,offset;
OS_TCB curr_task_pcb;
 null_timeouts.next = NULL;
 //获取当前任务的优先级
 err = OSTaskQuery(OS_PRIO_SELF,&curr_task_pcb);
 curr_prio = curr_task_pcb.OSTCBPrio;
 offset = curr_prio - LWIP_START_PRIO;
 //判断当前任务优先级是不是 tcp/ip 相关任务,优先级 5-9
 if(offset < 0 || offset >= LWIP_TASK_MAX)
 {
  return &null_timeouts;
 }
 return &lwip_timeouts[offset];
}
```

注意:杨晔大侠移植的代码在本函数有一个 bug.杨晔大侠的移植把上面函数中的
OS_TCB curr_task_tcb 定义成了全局变量,使本函数成为了一个不可重入函数.
我也是在进行如下测试时发现了这个 bug.
我的开发板上设置的 ip 地址是 192.168.1.95.
我在 windows 的 dos 窗口内运行
ping 192.168.1.95 –l 2000 –t,不间断用长度为 2000 的数据报进行 ping 测试,
同时使用 tftp 客户端软件给 192.168.1.95 下载一个十几兆程序,
同时再使用 telnet 连接 192.168.1.95 端口 7(echo 端口),往该端口写数测试 echo 功能.
在运行一段时间以后,开发板进入不再响应.我当时也是经过长时间的分析才发现是因为在低优先级任务运行 sys_arch_timeouts()时被高优先级任务打断改写了 curr_task_tcb 的值,从而使 sys_arch_timeouts 返回的指针错误,进而导致系统死锁.
函数 sys_timeout 给当前任务增加一个定时事件:

```
void sys_timeout(u32_t msecs, sys_timeout_handler h, void *arg)
{
  struct sys_timeouts *timeouts;
  struct sys_timeout *timeout, *t;
  timeout = memp_malloc(MEMP_SYS_TIMEOUT);//为定时事件分配内存
  if (timeout == NULL) {
    return;
  }
  timeout->next = NULL;
  timeout->h = h;
  timeout->arg = arg;
  timeout->time = msecs;
  timeouts = sys_arch_timeouts();//返回当前任务定时事件链表起始指针
  if (timeouts->next == NULL) {//如果链表为空直接增加该定时事件
    timeouts->next = timeout;
    return;
  }
   //如果链表不为空,对定时事件进行排序.注意定时事件中的 time 存储的是本事件
//时间相对于前一事件的时间的差值
  if (timeouts->next->time > msecs) {
timeouts->next->time -= msecs;
    timeout->next = timeouts->next;
    timeouts->next = timeout;
  } else {
    for(t = timeouts->next; t != NULL; t = t->next) {
      timeout->time -= t->time;
      if (t->next == NULL ||
   t->next->time > timeout->time) {
  if (t->next != NULL) {
    t->next->time  -= timeout->time;
  }
  timeout->next = t->next;
```

```
  t->next = timeout;
  break;
      }
    }
  }
}
```
函数 sys_untimeout 从当前任务定时事件链表中删除一个定时事件

```
void sys_untimeout(sys_timeout_handler h, void *arg)
{
    struct sys_timeouts *timeouts;
    struct sys_timeout *prev_t, *t;
    timeouts = sys_arch_timeouts();//返回当前任务定时事件链表起始指针
    if (timeouts->next == NULL)//如果链表为空直接返回
        {
        return;
        }
    //查找对应定时事件并从链表中删除.
    for (t = timeouts->next, prev_t = NULL; t != NULL; prev_t = t, t = t->next)
    {
        if ((t->h == h) && (t->arg == arg))
        {
            /* We have a match */
            /* Unlink from previous in list */
            if (prev_t == NULL)
                timeouts->next = t->next;
            else
                prev_t->next = t->next;
            /* If not the last one, add time of this one back to next */
            if (t->next != NULL)
                t->next->time += t->time;
            memp_free(MEMP_SYS_TIMEOUT, t);
            return;
        }
    }
    return;
}
```

2.2.3 “mbox”的实现:

(1)mbox 的创建

```
     sys_mbox_t sys_mbox_new(void)
{
        u8_t       ucErr;
        PQ_DESCR    pQDesc;
//从消息队列内存分区中得到一个内存块
        pQDesc = OSMemGet( pQueueMem, &ucErr );
```

```
    if( ucErr == OS_NO_ERR ) {
       //创建一个消息队列
      pQDesc->pQ=OSQCreate(&(pQDesc->pvQEntries[0]), MAX_QUEUE_ENTRIES );
           if( pQDesc->pQ != NULL ) {
         return pQDesc;
      }
    }
   return SYS_MBOX_NULL;
}
```

(2)发一条消息给"mbox"

```
 const void * const pvNullPointer = 0xffffffff;
void sys_mbox_post(sys_mbox_t mbox, void *data)
{
   INT8U err;
   if( !data )
    data = (void*)&pvNullPointer;
  err= OSQPost( mbox->pQ, data);
}
```

在 ucos 中,如果 OSQPost (OS_EVENT *pevent, void *msg)中的 msg==NULL 会返回一条 OS_ERR_POST_NULL_PTR 错误.而在 lwip 中会调用 sys_mbox_post(mbox,NULL)发送一条空消息,我们在本函数中把 NULL 变成一个常量指针 0xffffffff.

(3)从"mbox"中读取一条消息

```
#define SYS_ARCH_TIMEOUT 0xffffffff
void sys_mbox_fetch(sys_mbox_t mbox, void **msg)
{
 u32_t time;
 struct sys_timeouts *timeouts;
 struct sys_timeout *tmptimeout;
 sys_timeout_handler h;
 void *arg;
again:
 timeouts = sys_arch_timeouts();////返回当前任务定时事件链表起始指针
 if (!timeouts || !timeouts->next) {//如果定时事件链表为空
   sys_arch_mbox_fetch(mbox, msg, 0);//无超时等待消息
 } else {
   if (timeouts->next->time > 0) {
  //如果超时事件链表不为空,而且第一个超时事件的 time !=0
//带超时等待消息队列,超时时间等于超时事件链表中第一个超时事件的 time,
     time = sys_arch_mbox_fetch(mbox, msg, timeouts->next->time);
    //在后面分析中可以看到 sys_arch_mbox_fetch 调用了 ucos 中的 OSQPend 系统调
//用从消息队列中读取消息.
//如果"mbox"消息队列不为空,任务立刻返回,否则任务进入阻塞态.
//需要重点说明的是 sys_arch_mbox_fetch 的返回值 time:如果 sys_arch_mbox_fetch
//因为超时返回,time=SYS_ARCH_TIMEOUT,
```

```
//如果 sys_arch_mbox_fetch 因为收到消息而返回,
//time = 收到消息时刻的时间-执行 sys_arch_mbox_fetch 时刻的时间,单位是毫秒
//由于在 ucos 中任务调用 OSQPend 系统调用进入阻塞态,到收到消息重新开始执行
//这段时间没有记录下来,所以我们要简单修改 ucos 的源代码.(后面我们会看到).
    } else {
      //如果定时事件链表不为空,而且第一个定时事件的 time ==0,表示该事件的定时
//时间到
      time = SYS_ARCH_TIMEOUT;
    }
    if (time == SYS_ARCH_TIMEOUT) {
     //一个定时事件的定时时间到
      tmptimeout = timeouts->next;
      timeouts->next = tmptimeout->next;
      h = tmptimeout->h;
      arg = tmptimeout->arg;
      memp_free(MEMP_SYS_TIMEOUT, tmptimeout);
      //从内存中释放该定时事件,并执行该定时事件中的函数
      if (h != NULL) {
         h(arg);
      }
      //因为定时事件中的定时时间到或者是因为 sys_arch_mbo_fetch 超时到而执行到
//这里,返回本函数开头重新等待 mbox 的消息
      goto again;
    } else {
    //如果 sys_arch_mbox_fetch 无超时收到消息返回
//则刷新定时事件链表中定时事件的 time 值.
      if (time <= timeouts->next->time) {
  timeouts->next->time -= time;
      } else {
  timeouts->next->time = 0;
      }
    }

  }
}

u32_t  sys_arch_mbox_fetch(sys_mbox_t mbox, void **data, u32_t timeout)
{
    u32_t     ucErr;
    u16_t ucos_timeout;
  //在  lwip 中  ,timeout 的单位是 ms
  //  在 ucosII ,timeout  的单位是 timer tick
   ucos_timeout = 0;
  if(timeout != 0){
```

```
  ucos_timeout = (timeout )*( OS_TICKS_PER_SEC/1000);
  if(ucos_timeout < 1)
     ucos_timeout = 1;
  else if(ucos_timeout > 65535)
     ucos_timeout = 65535;
  }
  //如果 data!=NULL 就返回消息指针,
  if(data != NULL){
   *data = OSQPend( mbox->pQ, (u16_t)ucos_timeout, &ucErr );
  }else{
   OSQPend(mbox->pQ,(u16_t)ucos_timeout,&ucErr);
  }
//这里修改了 ucos 中的 OSQPend 系统调用,
//原来的 void  *OSQPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
// err 的返回值只有两种:收到消息就返回 OS_NO_ERR,超时则返回 OS_TIMEOUT
//这里先将 err 从 8 位数据改变成了 16 位数据  OSQPend(*pevent,timeout, INT16U *err)
//重新定义了 OS_TIMEOUT
//在 ucos 中原有#define OS_TIMEOUT 20
//改为  #define  OS_TIMEOUT  -1
//err 返回值的意义也改变了,如果超时返回 OS_TIMEOUT
//  如果收到消息,则返回 OSTCBCur->OSTCBDly 修改部分代码如下
//if (msg != (void *)0) { /* Did we get a message?  */
// OSTCBCur->OSTCBMsg = (void *)0;
// OSTCBCur->OSTCBStat     = OS_STAT_RDY;
// OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;
// *err = OSTCBCur->OSTCBDly;// zhangzs @2003.12.12
//    OS_EXIT_CRITICAL();
// return (msg);     /* Return message received */
//    }
//关于 ucos 的 OSTBCur->OSTCBDly 的含义请查阅 ucos 的书籍
  if( ucErr == OS_TIMEOUT ) {
      timeout = SYS_ARCH_TIMEOUT;
   } else {
     if(*data == (void*)&pvNullPointer )
     *data = NULL;
     //单位转换,从 ucos tick->ms
     timeout = (ucos_timeout -ucErr)*(1000/ OS_TICKS_PER_SEC);
   }
  return timeout;
}
```

semaphone 的实现和 mbox 类似,这里就不再重复了.

来源 21ic