内存管理

我们知道，在 ANSI C 中可以用 malloc()和 free()两个函数动态地分配内存和释放内存。但是，在嵌入式实时操作系统中，多次这样做会把原来很大的一块连续内存区域，逐渐地分割成许多非常小而且彼此又不相邻的内存区域，也就是内存碎片。由于这些碎片的大量存在，使得程序到后来连非常小的内存也分配不到。在 4.02 节的任务堆栈中，我们讲到过用 malloc()函数来分配堆栈时，曾经讨论过内存碎片的问题。另外，由于内存管理算法的原因，malloc()和 free()函数执行时间是不确定的。

在 μC/OS-II 中，操作系统把连续的大块内存按分区来管理。每个分区中包含有整数个大小相同的内存块，如同图 F7.1。利用这种机制，μC/OS-II 对 malloc()和 free()函数进行了改进，使得它们可以分配和释放固定大小的内存块。这样一来，malloc()和 free()函数的执行时间也是固定的了。

如图 F7.2，在一个系统中可以有多个内存分区。这样，用户的应用程序就可以从不同的内存分区中得到不同大小的内存块。但是，特定的内存块在释放时必须重新放回它以前所属于的内存分区。显然，采用这样的内存管理算法，上面的内存碎片问题就得到了解决。

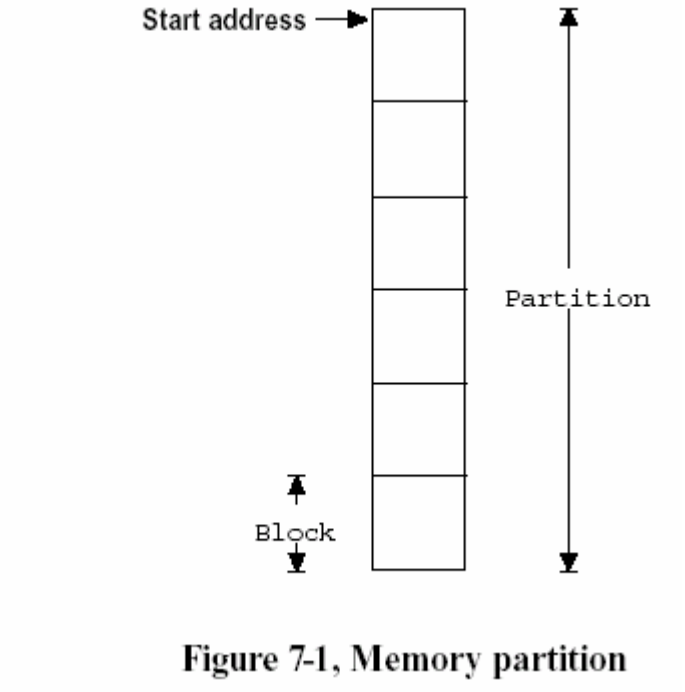

Figure 7-1, Memory partition

图 F7.1 内存分区——Figure 7.1

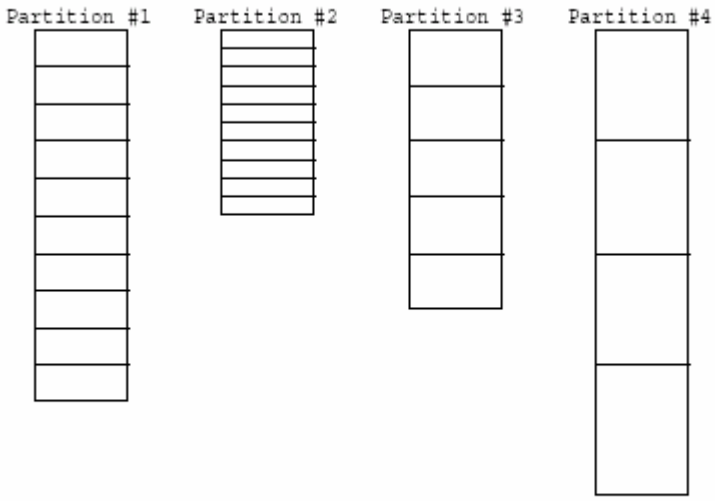

Figure 7-2, Multiple memory partitions.

图 F7.2 多个内存分区——Figure 7.2

内存控制块

为了便于内存的管理，在 μC/OS-II 中使用内存控制块（memory control blocks）的数据结构来跟踪每一个内存分区，系统中的每个内存分区都有它自己的内存控制块。程序清单 L7.1 是内存控制块的定义。

程序清单 L7.1 内存控制块的数据结构

```
typedef struct {
    void   *OSMemAddr;
    void   *OSMemFreeList;
    INT32U  OSMemBlkSize;
    INT32U  OSMemNBlks;
    INT32U  OSMemNFree;
} OS_MEM;
```

.OSMemAddr 是指向内存分区起始地址的指针。它在建立内存分区[见 7.1 节，建立一个内存分区，OSMemCreate()]时被初始化，在此之后就不能更改了。
.OSMemFreeList 是指向下一个空闲内存控制块或者下一个空闲的内存块的指针，具体含义要根据该内存分区是否已经建立来决定[见 7.1 节]。
.OSMemBlkSize 是内存分区中内存块的大小，是用户建立该内存分区时指定的[见 7.1 节]。
.OSMemNBlks 是内存分区中总的内存块数量，也是用户建立该内存分区时指定的[见 7.1 节]。
.OSMemNFree 是内存分区中当前可以得空闲内存块数量。
如果要在 μC/OS-II 中使用内存管理，需要在 OS_CFG.H 文件中将开关量 OS_MEM_EN 设置为 1。这样 μC/OS-II 在启动时就会对内存管理器进行初始化[由 OSInit()调用 OSMemInit()实现]。该初始化主要建立一个图 F7.3 所示的内存控制块链表，其中的常数 OS_MAX_MEM_PART（见文件 OS_CFG.H）定义了最大的内存分区数，该常数值至少应为 2。

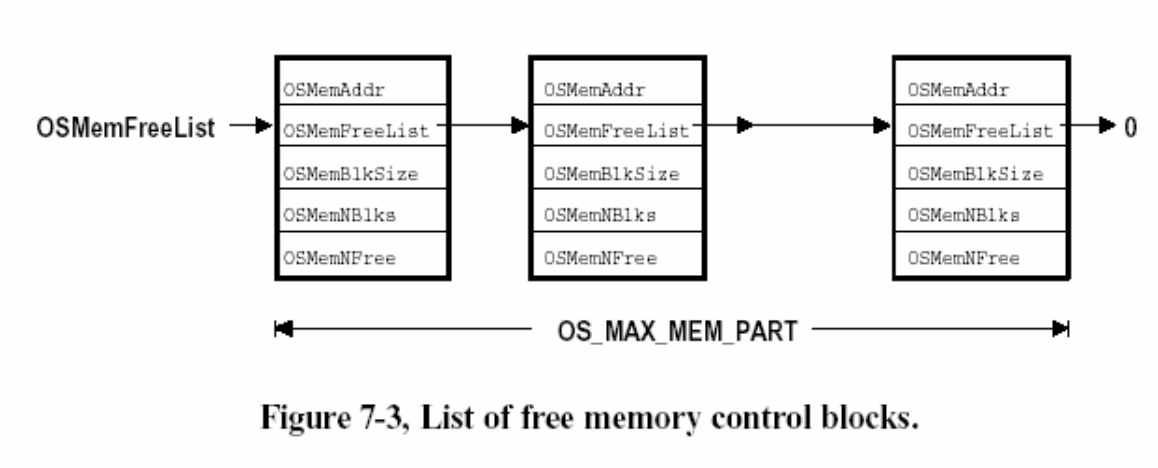

Figure 7-3, List of free memory control blocks.

图 F7.3 空闲内存控制块链表——Figure 7.3

建立一个内存分区，OSMemCreate()

在使用一个内存分区之前，必须先建立该内存分区。这个操作可以通过调用 OSMemCreate()函数来完成。程序清单 L7.2 说明了如何建立一个含有 100 个内存块、每个内存块 32 字节的内存分区。

程序清单 L7.2 建立一个内存分区

```
OS_MEM *CommTxBuf;
INT8U    CommTxPart[100][32];


void main (void)
{
    INT8U err;


    OSInit();
    .
    .
    CommTxBuf = OSMemCreate(CommTxPart, 100, 32, &err);
    .
    .
    OSStart();
}
```

程序清单 L7.3 是 OSMemCreate()函数的源代码。该函数共有 4 个参数：内存分区的起始地址、分区内的内存块总块数、每个内存块的字节数和一个指向错误信息代码的指针。如果 OSMemCreate()操作失败，它将返回一个 NULL 指针。否则，它将返回一个指向内存控制块的指针。对内存管理的其它操作，象 OSMemGet()，OSMemPut()，OSMemQuery()函数等，都要通过该指针进行。

每个内存分区必须含有至少两个内存块[L7.3(1)]，每个内存块至少为一个指针的大小，因为同一分区中的所有空闲内存块是由指针串联起来的[L7.3(2)]。接着，OSMemCreate()从系统中的空闲内存控制块中取得一个内存控制块[L7.3(3)]，该内存控制块包含相应内存分区的运行信息。OSMemCreate()必须在有空闲内存控制块可用的情况下才能建立一个内存分区[L7.3(4)]。在上述条件均得到满足时，所要建立的内存分区内的所有内存块被链接成一个单向的链表[L7.3(5)]。然后，在对应的内存控制块中填写相应的信息[L7.3(6)]。完成上述各动作后，OSMemCreate()返回指向该内存块的指针。该指针在以后对内存块的操作中使用[L7.3(6)]。

程序清单 L7.3 OSMemCreate()

```
OS_MEM *OSMemCreate (void *addr, INT32U nblks, INT32U blksize, INT8U *err)
{
    OS_MEM  *pmem;
    INT8U   *pblk;
    void   **plink;
    INT32U   i;



    if (nblks < 2) {                                                    (1)
        *err = OS_MEM_INVALID_BLKS;
        return ((OS_MEM *)0);
    }
    if (blksize < sizeof(void *)) {                                     (2)
        *err = OS_MEM_INVALID_SIZE;
        return ((OS_MEM *)0);
    }
    OS_ENTER_CRITICAL();
    pmem = OSMemFreeList;                                               (3)
    if (OSMemFreeList != (OS_MEM *)0) {
        OSMemFreeList = (OS_MEM *)OSMemFreeList->OSMemFreeList;
    }
    OS_EXIT_CRITICAL();
    if (pmem == (OS_MEM *)0) {                                          (4)
        *err = OS_MEM_INVALID_PART;
        return ((OS_MEM *)0);
    }
    plink = (void **)addr;                                              (5)
    pblk  = (INT8U *)addr + blksize;
    for (i = 0; i < (nblks - 1); i++) {
        *plink = (void *)pblk;
        plink  = (void **)pblk;
```

```
        pblk   = pblk + blksize;
    }
    *plink = (void *)0;
    OS_ENTER_CRITICAL();
    pmem->OSMemAddr     = addr;                                    (6)
    pmem->OSMemFreeList = addr;
    pmem->OSMemNFree    = nblks;
    pmem->OSMemNBlks    = nblks;
    pmem->OSMemBlkSize  = blksize;
    OS_EXIT_CRITICAL();
    *err   = OS_NO_ERR;
    return (pmem);                                                 (7)
}
```

图 F7.4 是 OSMemCreate()函数完成后，内存控制块及对应的内存分区和分区内的内存块之间的关系。在程序运行期间，经过多次的内存分配和释放后，同一分区内的各内存块之间的链接顺序会发生很大的变化。

分配一个内存块，OSMemGet()

应用程序可以调用 OSMemGet()函数从已经建立的内存分区中申请一个内存块。该函数的唯一参数是指向特定内存分区的指针，该指针在建立内存分区时，由 OSMemCreate()函数返回。显然，应用程序必须知道内存块的大小，并且在使用时不能超过该容量。例如，如果一个内存分区内的内存块为 32 字节，那么，应用程序最多只能使用该内存块中的 32 字节。当应用程序不再使用这个内存块后，必须及时把它释放，重新放入相应的内存分区中[见 7.03 节，释放一个内存块，OSMemPut()]。

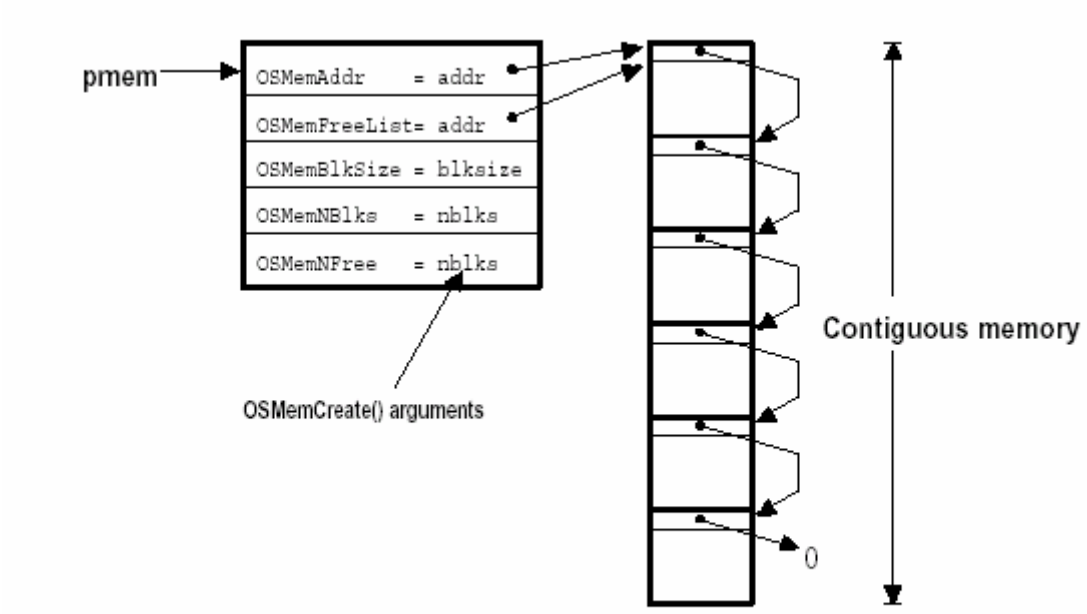

Figure 7-4, OSMemCreate()

图 F7.4 OSMemCreate()——Figure 7.4

程序清单 L7.4 是 OSMemGet()函数的源代码。参数中的指针 pmem 指向用户希望从其中分配内存块的内存分区[L7.4(1)]。OSMemGet()首先检查内存分区中是否有空闲的内存块[L7.4(2)]。如果有，从空闲内存块链表中删除第一个内存块[L7.4(3)]，并对空闲内存块链表作相应的修改 [L7.4(4)]。这包括将链表头指针后移一个元素和空闲内存块数减 1[L7.4(5)]。最后，返回指向被分配内存块的指针[L7.4(6)]。

程序清单 L7.4 OSMemGet()

```
void *OSMemGet (OS_MEM *pmem, INT8U *err)                              (1)
{
    void    *pblk;



    OS_ENTER_CRITICAL();
    if (pmem->OSMemNFree > 0) {                                        (2)
        pblk                 = pmem->OSMemFreeList;                    (3)
        pmem->OSMemFreeList = *(void **)pblk;                          (4)
        pmem->OSMemNFree--;                                            (5)
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
        return (pblk);                                                 (6)
    } else {
        OS_EXIT_CRITICAL();
        *err = OS_MEM_NO_FREE_BLKS;
        return ((void *)0);
    }
}
```

值得注意的是，用户可以在中断服务子程序中调用 OSMemGet()，因为在暂时没有内存块可用的情况下，OSMemGet()不会等待，而是马上返回 NULL 指针。

释放一个内存块，OSMemPut()

当用户应用程序不再使用一个内存块时，必须及时地把它释放并放回到相应的内存分区中。这个操作由 OSMemPut()函数完成。必须注意的是，OSMemPut()并不知道一个内存块是属于哪个内存分区的。例如，用户任务从一个包含 32 字节内存块的分区中分配了一个内存块，用完后，把它返还给了一个包含 120 字节内存块的内存分区。当用户应用程序下一次申请 120 字节分区中的一个内存块时，它会只得到 32 字节的可用空间，其它 88 字节属于其它的任务，这就有可能使系统崩溃。

程序清单 L7.5 是 OSMemPut()函数的源代码。它的第一个参数 pmem 是指向内存控制块的指针，也即内存块属于的内存分区[L7.5(1)]。OSMemPut()首先检查内存分区是否已满[L7.5(2)]。如果已满，说明系统在分配和释放内存时出现了错误。如果未满，要释放的内存

块被插入到该分区的空闲内存块链表中[L7.5(3)]。最后，将分区中空闲内存块总数加1[L7.5(4)]。

程序清单 L7.5 OSMemPut()

```
INT8U OSMemPut (OS_MEM  *pmem, void *pblk)                              (1)
{
    OS_ENTER_CRITICAL();
    if (pmem->OSMemNFree >= pmem->OSMemNBlks) {                         (2)
        OS_EXIT_CRITICAL();
        return (OS_MEM_FULL);
    }
    *(void **)pblk      = pmem->OSMemFreeList;                          (3)
    pmem->OSMemFreeList = pblk;
    pmem->OSMemNFree++;                                                 (4)
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

查询一个内存分区的状态，OSMemQuery()

在 μC/OS-II 中，可以使用 OSMemQuery()函数来查询一个特定内存分区的有关消息。通过该函数可以知道特定内存分区中内存块的大小、可用内存块数和正在使用的内存块数等信息。所有这些信息都放在一个叫 OS_MEM_DATA 的数据结构中，如程序清单 L7.6。

程序清单 L7.6 OS_MEM_DATA数据结构

```
typedef struct {
    void  *OSAddr;     /* 指向内存分区首地址的指针 */
    void  *OSFreeList; /* 指向空闲内存块链表首地址的指针 */
    INT32U OSBlkSize;  /* 每个内存块所含的字节数 */
    INT32U OSNBlks;    /* 内存分区总的内存块数 */
    INT32U OSNFree;    /* 空闲内存块总数 */
    INT32U OSNUsed;    /* 正在使用的内存块总数 */
} OS_MEM_DATA;
```

程序清单 L7.7 是 OSMemQuery()函数的源代码，它将指定内存分区的信息复制到 OS_MEM_DATA 定义的变量的对应域中。在此之前，代码首先禁止了外部中断，防止复制过程中某些变量值被修改[L7.7(1)]。由于正在使用的内存块数是由 OS_MEM_DATA 中的局部变量计算得到的，所以，可以放在（critical section 中断屏蔽）的外面。

程序清单 L7.7 OSMemQuery()

```
INT8U OSMemQuery (OS_MEM *pmem, OS_MEM_DATA *pdata)
{
```

```
    OS_ENTER_CRITICAL();
    pdata->OSAddr     = pmem->OSMemAddr;                                    (1)
    pdata->OSFreeList = pmem->OSMemFreeList;
    pdata->OSBlkSize  = pmem->OSMemBlkSize;
    pdata->OSNBlks    = pmem->OSMemNBlks;
    pdata->OSNFree    = pmem->OSMemNFree;
    OS_EXIT_CRITICAL();
    pdata->OSNUsed    = pdata->OSNBlks - pdata->OSNFree;                    (2)
    return (OS_NO_ERR);
}
```

Using Memory Partitions

图 F7.5 是一个演示如何使用 μC/OS-II 中的动态分配内存功能，以及利用它进行消息传递[见第 6 章]的例子。程序清单 L7.8 是这个例子中两个任务的示意代码，其中一些重要代码的标号和图 F7.5 中括号内用数字标识的动作是相对应的。

第一个任务读取并检查模拟输入量的值（如气压、温度、电压等），如果其超过了一定的阈值，就向第二个任务发送一个消息。该消息中含有时间信息、出错的通道号和错误代码等可以想象的任何可能的信息。

错误处理程序是该例子的中心。任何任务、中断服务子程序都可以向该任务发送出错消息。错误处理程序则负责在显示设备上显示出错信息，在磁盘上登记出错记录，或者启动另一个任务对错误进行纠正等。

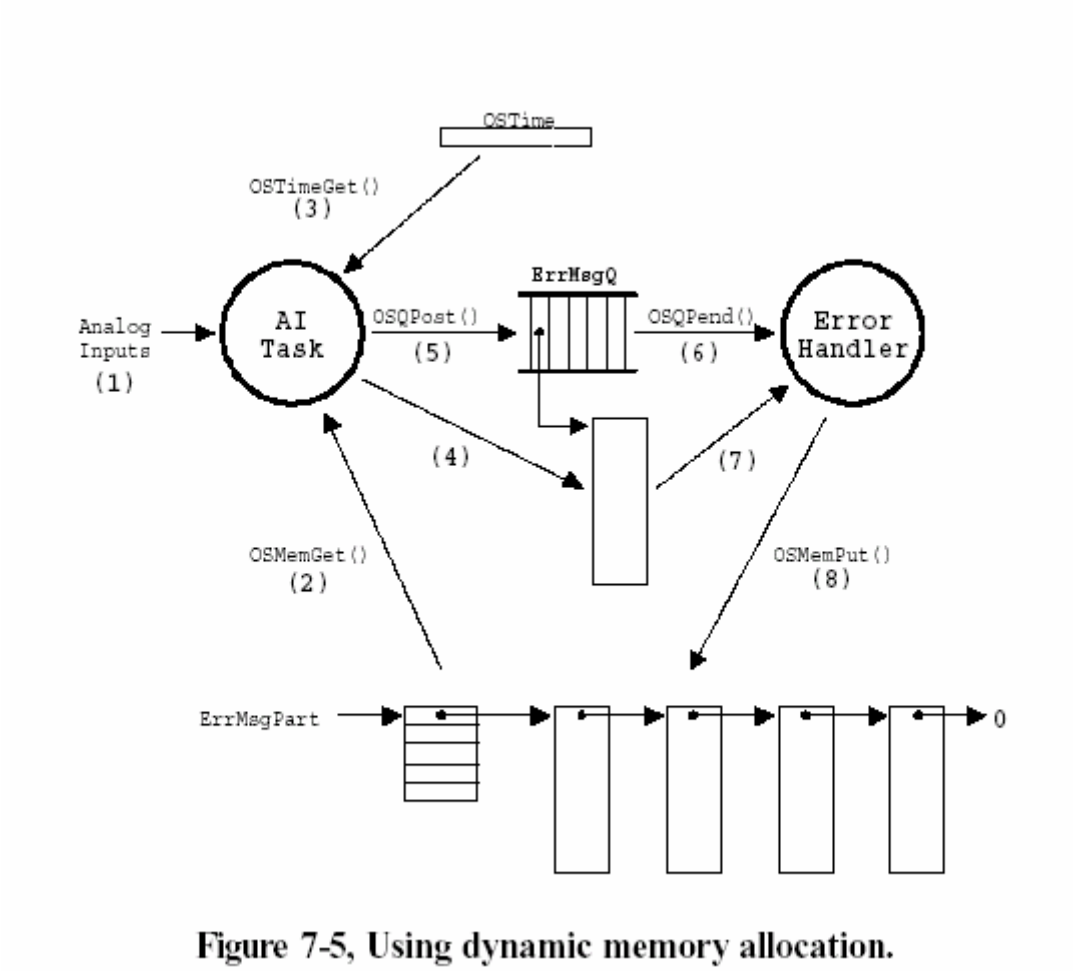

Figure 7-5, Using dynamic memory allocation.

图 F7.5 使用动态内存分配——Figure 7.5

程序清单 L7.8 内存分配的例子——扫描模拟量的输入和报告出错

```
AnalogInputTask()
{
    for (;;) {
        for (所有的模拟量都有输入) {
            读入模拟量输入值;                                    (1)
            if (模拟量超过阈值) {
                得到一个内存块;                                  (2)
                得到当前系统时间 (以时钟节拍为单位);             (3)
                将下列各项存入内存块:                            (4)
                    系统时间 (时间戳);
                    超过阈值的通道号;
                    错误代码;
                    错误等级;
                    等.
                向错误队列发送错误消息;                          (5)
```

```
                    (一个指向包含上述各项的内存块的指针)
                }
            }
            延时任务,直到要再次对模拟量进行采样时为止;
        }
    }



ErrorHandlerTask()
{
    for (;;) {
        等待错误队列的消息;                                          (6)
            (得到指向包含有关错误数据的内存块的指针)
        读入消息,并根据消息的内容执行相应的操作;                    (7)
        将内存块放回到相应的内存分区中;                              (8)
    }
}
```

等待一个内存块

有时候，在内存分区暂时没有可用的空闲内存块的情况下，让一个申请内存块的任务等待也是有用的。但是，μC/OS-II 本身在内存管理上并不支持这项功能。如果确实需要，则可以通过为特定内存分区增加信号量的方法，实现这种功能（见 6.05 节，信号量）。应用程序为了申请分配内存块，首先要得到一个相应的信号量，然后才能调用 OSMemGet()函数。整个过程见程序清单 L7.9。

程序代码首先定义了程序中使用到的各个变量[L7.9(1)]。该例中，直接使用数字定义了各个变量的大小，实际应用中，建议将这些数字定义成常数。在系统复位时，μC/OS-II 调用 OSInit()进行系统初始化[L7.9(2)]，然后用内存分区中总的内存块数来初始化一个信号量[L7.9(3)]，紧接着建立内存分区[L7.9(4)]和相应的要访问该分区的任务[L7.9(5)]。当然，到此为止，我们对如何增加其它的任务也已经很清楚了。显然，如果系统中只有一个任务使用动态内存块，就没有必要使用信号量了。这种情况不需要保证内存资源的互斥。事实上，除非我们要实现多任务共享内存，否则连内存分区都不需要。多任务执行从 OSStart()开始[L7.9(6)]。当一个任务运行时，只有在信号量有效时[L7.9(7)]，才有可能得到内存块[L7.9(8)]。一旦信号量有效了，就可以申请内存块并使用它，而没有必要对 OSSemPend()返回的错误代码进行检查。因为在这里，只有当一个内存块被其它任务释放并放回到内存分区后，μC/OS-II 才会返回到该任务去执行。同理，对 OSMemGet()返回的错误代码也无需做进一步的检查（一个任务能得以继续执行，则内存分区中至少有一个内存块是可用的）。当一个任务不再使用某内存块时，只需简单地将它释放并返还到内存分区[L7.9(9)]，并发送该信号量[L7.9(10)]。

程序清单 L7.9 等待从一个内存分区中分配内存块

```
OS_EVENT  *SemaphorePtr;                                             (1)
```

```
OS_MEM    *PartitionPtr;
INT8U      Partition[100][32];
OS_STK     TaskStk[1000];


void main (void)
{
    INT8U err;

    OSInit();                                                    (2)
    .
    .
    SemaphorePtr = OSSemCreate(100);                             (3)
    PartitionPtr = OSMemCreate(Partition, 100, 32, &err);        (4)
    .
    OSTaskCreate(Task, (void *)0, &TaskStk[999], &err);          (5)
    .
    OSStart();                                                   (6)
}
void Task (void *pdata)
{
    INT8U  err;
    INT8U *pblock;


    for (;;) {
        OSSemPend(SemaphorePtr, 0, &err);                        (7)
        pblock = OSMemGet(PartitionPtr, &err);                   (8)
        .
        .  /* 使用内存块 */
        .
        OSMemPut(PartitionPtr, pblock);                          (9)
        OSSemPost(SemaphorePtr);                                (10)
    }
}
```