# 15

# *High-Level Memory Management and Message Passing*

This chapter describes the design of high-level buffer memory management and message passing facilities. The motivation for buffer management comes from the disk driver software that uses these facilities to allocate fixed-size buffers, to pass them among processes freely, and to release them. To prevent the software from exhausting memory resources, the high level memory manager must be able to limit the amount of memory used for a given function. It must coordinate processes by blocking them until their requests can be satisfied.

Network driver software, which is not considered in this book, requires message-passing primitives for passing data among the layers efficiently. The high-level message passing facility described in this chapter provides buffered message exchange through named rendezvous points.

Instead of building these primitives directly into the driver software, we have chosen to design general-purpose routines that can be used in a variety of ways. One advantage of designing general-purpose primitives is that they become available to user programs. Another advantage is that they can be implemented and tested independent of the specific routines that use them. In the case of the primitives described in this chapter, testing can be carried out even before the driver software has been designed. While we have said little about testing up to this point, it should be obvious that systems as complicated as PC-Xinu must be built and tested in small pieces; designs that attempt to integrate functions into large subsystems are doomed to failure.

## 15.1 Self-Initializing Modules

Lower layers of the operating system, like the process manager, must be present whenever a program runs. Higher layers like the disk software need not be included, however, unless a program uses them. In PC-Xinu, optional software is stored in libraries, from which the linker selects the routines that are referenced when it constructs a memory image. It is convenient to think of a set of related library routines as a *module* that implements operations on some abstract data object. For example, one can imagine a set of procedures *push*, *pop*, and *makestack* that push an element onto a stack, pop an element from a stack, and create a stack.

In C, procedures that form a module can be combined into a single source file along with the static data upon which they operate. When a program references one of the procedures in the file, the entire file is loaded. (Alternatively, the procedures can reside in separate files and use shared external data structures.)

This chapter describes two modules – one that implements a mechanism for data exchange (called *ports*), and one that implements a mechanism for buffer storage allocation (called *buffer pools*). Before describing the routines themselves, we address the issue of how to initialize the static data associated with a module.

### 15.1.1 Specifying Initialization

The designer could arrange for the operating system to initialize each module when it first starts. This solution has three drawbacks. First, if the system initialization procedure explicitly references an object, the linker would include that object (and all other objects defined in the same file) in the memory image. Thus, every procedure and variable from every library module would be present in memory – something that is clearly undesirable on a small machine. Second, adding a module to a library would involve changing the operating system initialization procedure. Third, users could not have their modules initialized automatically because they may not be able to change the system initialization routine.

Forcing the programmers to initialize modules explicitly is equally undesirable. They would have to know which primitives go with each module and remember to change the way their programs performed initialization when they add or remove calls to library procedures. Ideally, the routines that comprise a module should be *self-initializing*; they should perform initialization automatically (and exactly once) as they are called.

### 15.1.2 Automatic Initialization

In most environments, self-initialization is not difficult. Modules include a Boolean variable that is set to *false* when the program is loaded. Each procedure in the module checks the module's Boolean variable when called, performing initialization if it is zero. The initialization code assigns the Boolean variable the value *true*, so subsequent uses of the module can proceed without calling the initialization procedure.

Linker-defined Boolean variables do not suffice for self-initialization in environments where the system can be restarted without being reloaded. There are two ways to overcome the problem: the system could be changed to set all static memory to zero at startup, or a new operating system primitive could be added that tests whether something has been initialized independent of the values in static memory. We have chosen the latter option; the next section describes such a primitive.

## 15.2 Memory Marking

This section introduces a technique called *memory marking* that enables modules to initialize themselves. Memory marking does not perform initialization; it is merely a service supplied by the system that reliably determines whether a memory location has been ''marked'' since the system started. The essential idea is this: the system maintains a set *S* of ''marked'' memory locations. When the system starts running, it sets *S* to empty. As execution proceeds, processes call system procedure *mark*(*k*) to add memory location *k* to set *S*; they call Boolean function *unmarked*(*k*) to test whether the location of integer *k* is currently in set *S*.

The implementation of memory marking is surprising because the operations are extremely efficient. Initializing *S* to empty at startup requires one instruction. Testing whether a location has been marked or marking a location, each require only a few instructions; their execution cost does not depend on the number of locations that have been marked. The trick required to achieve this efficiency comes from a clever use of pointers – the ''marked'' location contains an index that lets the system verify whether it is in *S* without searching the entire set.

## 15.3 Implementation Of Memory Marking

The memory marking routines maintain a set of addresses of ''marked'' locations in array *marks*. Each marked location contains the integer index in *marks* corresponding to that location's mark. To test whether the location of integer *L* has been marked, *unmarked* checks whether *marks*[*L*] contains the address of *L*. The code for predicate *unmarked* is found in file *mark.h*.

```
/* mark.h - unmarked */

#ifdef  MEMMARK

#ifndef MAXMARK
#define MAXMARK 20              /* maximum number of marked locations   */
#endif

extern  int     *(marks[]);
extern  int     nmarks;
extern  int     mkmutex;
typedef int     MARKER[1];      /* by declaring it to be an array, the  */
                                /* name provides an address so forgotten*/
                                /* &'s don't become a problem           */

#define unmarked(L)             (L[0]<0 || L[0]>=nmarks || marks[L[0]]!=L)

#endif /* def MEMMARK */
```

Note that *unmarked* is defined as a macro but could easily have been written as a subroutine that returns a value. It is important to understand the difference between a macro and a procedure in this context. Subroutine calls always result in longer execution time, since there is substantial overhead for passing parameters, calling, and returning; but the code for a subroutine is included only once in the load module. Macros, on the other hand, are expanded at compile time and do not incur the same run-time overhead as subroutines; they do, however, result in more code generated at each macro call. It is considered good practice to use macros where execution speed, rather than code space, is an important issue, as is the case with memory marking.

Marking requires the system to add a new element to the *marks* array and to place the appropriate index in the ‘‘marked’’ location. The code is found in file *mark.c*.

```
/* mark.c - _mkinit, mark */

#include <conf.h>
#include <kernel.h>
#include <mark.h>

#ifdef MEMMARK
int             *marks[MAXMARK];
int             nmarks;
int             mkmutex;

/*------------------------------------------------------------------------
```

```
 *  _mkinit  --  initialize memory marking; called once at system startup
 *------------------------------------------------------------------------
 */

_mkinit()
{
        mkmutex = screate(1);
        nmarks = 0;
}

/*------------------------------------------------------------------------
 *  mark  --  mark a location if it has not been marked
 *------------------------------------------------------------------------
 */

mark(loc)
int loc[];
{
        if ( unmarked(loc) == 0 )
                return(0);
        if (nmarks>=MAXMARK)
                return(SYSERR);
        wait(mkmutex);
        marks[ loc[0] = nmarks++ ] = loc;
        signal(mkmutex);
        return(OK);
}
#endif
```

The buffer pool routines, found in the next section, show how modules use these memory marking primitives for self-initialization.

## 15.4 Partitioned Space Allocation

Routines *getmem* and *freemem*, that were described in Chapter 8, constitute the basic memory manager. They place no limit on the amount that a given process can allocate, nor do they attempt to divide free space ‘‘fairly’’ – they merely honor requests on a first-come-first-served basis until no free memory remains. Once free memory has been exhausted, these routines reject further requests without waiting for memory to be released. Such global allocation strategies are relatively efficient, but because they force all processes to contend for the same memory they permit deprivation, a situation in which a process or processes cannot obtain memory because it has been consumed.

There are a number of situations illustrating the inadequacy of global memory allocation. For example, what should a process do if a request for memory fails? It is unreasonable for a process to terminate itself if this should occur, but busy-waiting for available space is equally unreasonable. In another situation, a process may allocate all available memory for some processing activity, only to find it impossible to write the results to disk because there is no space left to allocate a disk request buffer.

Global memory allocation schemes also do not work well for communication software because the time required to process messages is often longer than the time required to read them, and exhaustive allocation can lead to disaster. Consider, for example, the situation where a receiver process repeatedly obtains a buffer, reads data into it from the network, and then enqueues the buffer for processing. As incoming messages pile up waiting to be processed, the process reading input keeps allocating space for new messages from the free memory. In the worst case, it will use up all the available space. One might expect that the process consuming messages would eventually release space, allowing the reader to continue, but this may not happen. The process that consumes incoming messages must also allocate space to perform such tasks as reassembling long messages, or forwarding copies of messages to other machines. If no space remains, the system can deadlock with the consumer process waiting for space it needs to handle existing messages, and the reader process waiting for space it needs to read more messages.

To prevent deadlocks caused by exhaustive memory allocation schemes, higher-level memory management must be designed to partition free memory and control the allocation and deallocation independently within each partition. By limiting the amount of memory that processes use for a particular function, the system can guarantee that excessive requests will not lead to global deprivation. Furthermore, the system can assume that memory allocated for a particular function will always be returned, so it can arrange to suspend processes until their memory request can be satisfied, eliminating the overhead introduced by busy waiting. The mechanism we have chosen to handle these tasks is a *buffer pool* manager.

## 15.5 Buffer Pools

Each buffer pool consists of a fixed number of memory blocks, where all blocks in a given pool are the same length. The term *buffer* was chosen to reflect the intended use in I/O routines and communication software.

The memory space for a particular set of buffers is allocated all at once, when the pool is created. Each pool is identified by an integer *pool identifier* through which processes refer to it. After initialization, a process can request a buffer from a pool (by giving its identifier) or release a buffer back to a pool. There is no need to specify a buffer length in these requests because the size of memory blocks allocated to the pool is fixed when the pool is created.

The buffer pool mechanism differs from the low-level memory manager in another way: processes that request buffers will block until one is available. As usual, semaphores are used to control the resource. A process requesting a buffer from a pool *wait*s

on that pool's semaphore; the call returns immediately if buffers remain in the pool. If no buffers remain, the process blocks. Eventually, when another process returns a buffer to a pool, it signals the pool's semaphore, allowing a blocked process to obtain the buffer and resume execution.

The pool data structures consist of a table that contains a semaphore and a linked list of buffers for each pool. Pertinent declarations can be found in file bufpool.h:

```
/* bufpool.h */

#ifndef NBPOOLS
#define NBPOOLS 5                       /* Maximum number of pools      */
#endif

#ifndef BPMAXB
#define BPMAXB  512                     /* Maximum buffer length        */
#endif

#define BPMINB  2                       /* Minimum buffer length        */

#ifndef BPMAXN
#define BPMAXN  100                     /* Maximum buffers in any pool  */
#endif

struct  bpool   {                       /* Description of a single pool */
        int     bpsize;                 /* size of buffers in this pool */
        int     *bpnext;                /* pointer to next free buffer  */
        int     bpsem;                  /* semaphore that counts buffers*/
        };                              /*  currently in THIS pool      */

extern  struct  bpool bptab[];          /* Buffer pool table            */
extern  int     nbpools;                /* current number of pools      */
extern  char    *getbuf();

#ifdef  MEMMARK
extern  MARKER  bpmark;
#endif
```

Structure *bpool* defines the contents of entries in the buffer pool table, *bptab*. The buffers for a given pool are linked into a list, with head *bpnext*. Semaphore *bpsem* controls allocation from the pool, and integer *bpsize* gives the length of buffers in the pool.

Processes call procedure *getbuf* to obtain a buffer, passing the pool identifier as an argument. *Getbuf* works as expected, waiting until a buffer is available and then unlinking it from the list:

```
/* getbuf.c - getbuf */

#include <conf.h>
#include <kernel.h>
#include <mark.h>
#include <bufpool.h>

/*------------------------------------------------------------------------
 *  getbuf  --  get a buffer from a preestablished buffer pool
 *------------------------------------------------------------------------
 */
char *getbuf(poolid)
int poolid;
{
        int     ps;
        int     *buf;

        disable(ps);
        if ( poolid<0 || poolid >= nbpools
#ifdef  MEMMARK
                || unmarked(bpmark)
#endif
                ) {
                restore(ps);
                return(NULL);
        }
        wait( bptab[poolid].bpsem );
        buf = bptab[poolid].bpnext;
        bptab[poolid].bpnext = (int *) *buf;
        *buf++ = poolid;
        restore(ps);
        return((char *)buf);
}
```

The conditional code at the beginning of *getbuf* contains an *if* statement that determines whether the buffer pool routines have been initialized and returns *SYSERR* if they have not. The statement is valid only if memory marking is available on the system, so it has been made into conditional code that will be compiled only if constant *MEMMARK* is defined.

Observant readers may have noticed that *getbuf* does not return the address of the beginning of the physical buffer to its caller. Instead, it stores the pool id in the first integer location and returns the address just beyond the id. A user need not worry that the first location in the buffer holds the pool id – the initialization routine actually allocates

extra space in each buffer to hold the id when it creates the pool. As we will see, the procedure *freebuf* uses this pool id when it returns the buffer to free storage, making it unnecessary to specify the pool id when returning buffers. The technique of identifying a buffer automatically turns out to be useful when buffers are returned by processes other than the one that allocated them.

## 15.6 Returning Buffers To The Buffer Pool

Procedure *freebuf* returns a buffer to the correct pool given its address. The code is located in file freebuf.c:

```
/* freebuf.c - freebuf */

#include <conf.h>
#include <kernel.h>
#include <mark.h>
#include <bufpool.h>

/*------------------------------------------------------------------------
 *  freebuf  --  free a buffer that was allocated from a pool by getbuf
 *------------------------------------------------------------------------
 */
int freebuf(buf)
char *buf;
{
        int     ps;
        int     poolid;
        int     *bf;

        if ( buf == NULL )
                return(SYSERR);
        disable(ps);
        bf = (int *) buf;
        poolid = *(--bf);
        if ( poolid<0 || poolid>=nbpools
#ifdef  MEMMARK
                || unmarked(bpmark)
#endif
                ) {
                restore(ps);
                return(SYSERR);
        }
        *bf = (int)bptab[poolid].bpnext;
        bptab[poolid].bpnext = bf;
        signal(bptab[poolid].bpsem);
        restore(ps);
        return(OK);
}
```

*Freebuf* obtains the pool id that *getbuf* stored in the block when it was allocated, links the buffer back into the appropriate pool, and signals the pool semaphore *bpsem*, allowing a process to use the buffer.

## 15.7 Creating A Buffer Pool

Procedure *mkpool* creates a new buffer pool and returns its id. Like most other identifiers in PC-Xinu, the pool id is merely an index into the global buffer pool table. *Mkpool* computes the size of memory required and calls *getmem* to allocate it. It then divides up the memory into buffers and links them together into a free list. Enough space is allocated for each buffer so the integer pool id can be stored in the buffer along with user data. After the free list has been formed, *mkpool* returns its id to the caller:

```
/* mkpool.c - mkpool */

#include <conf.h>
#include <kernel.h>
#include <mark.h>
#include <mem.h>
#include <bufpool.h>

/*------------------------------------------------------------------------
 *  mkpool  --  allocates buffers for a buffer pool
 *              and links them together
 *------------------------------------------------------------------------
 */

int mkpool(bufsiz, numbufs)
int     bufsiz, numbufs;
{
        int     ps;
        int     poolid;
        char    *where;


        disable(ps);
#ifdef MEMMARK
        if ( unmarked(bpmark) )
                poolinit();
#endif
        bufsiz = (bufsiz + 1) & ~1;     /* round up             */
        if ( bufsiz<BPMINB ||
             bufsiz>BPMAXB ||
             numbufs<1 ||
             numbufs>BPMAXN ||
             nbpools >= NBPOOLS ||
             (where=getmem((bufsiz+sizeof(int))*numbufs)) == NULL ) {
                restore(ps);
```

```
                return(SYSERR);
        }
        poolid = nbpools++;
        bptab[poolid].bpnext = (int *)where;
        bptab[poolid].bpsize = bufsiz;
        bptab[poolid].bpsem  = screate(numbufs);
        bufsiz += sizeof(int);
        for ( numbufs-- ; numbufs>0 ; numbufs--, where += bufsiz )
                *( (int *) where) = bufsiz+(int)where;
        *( (int *) where) = (int) NULL;
        restore(ps);
        return(poolid);
}
```

## 15.8 Initializing The Buffer Pool Table

Procedure *poolinit* initializes the buffer pool table, *bptab*. The code, found in file *poolinit.c*, uses conditional compilation to select one of two versions. If memory marking is available, *poolinit* and the other buffer pool routines use it to self-initialize. If memory marking is not available, the user must call *poolinit* explicitly.

Refer to the conditional code in *mkpool* to see how it calls *poolinit*. When compiled to use memory marking, *mkpool* performs an extra test at the beginning: it invokes *unmarked* to test whether the module's memory marker, *bpmark*, has been marked. If it has not, then *mkpool* needs to initialize the module, so it calls *poolinit* and marks *bpmark*. When *mkpool* is compiled without memory marking, the user must call *poolinit* explicitly before creating any buffer pools.

```
/* poolinit.c - poolinit */

#include <conf.h>
#include <kernel.h>
#include <mark.h>
#include <bufpool.h>

struct bpool bptab[NBPOOLS];
int    nbpools;

#ifdef MEMMARK
MARKER bpmark;
#endif

/*------------------------------------------------------------------------
 *  poolinit  --  initialize the buffer pool
```

```
 *------------------------------------------------------------------------
 */

SYSCALL poolinit()
{
#ifdef  MEMMARK
        int     status;

        if ( (status=mark(bpmark)) == SYSERR )
                panic("poolinit - memory marking error");
        if ( status == 0 )              /* already marked               */
                return(OK);
#endif
        nbpools = 0;
        return(OK);
}
```

## 15.9 Communication Ports

Communication ports are rendezvous points through which processes exchange messages. They differ from process-to-process message passing, described in Chapter 7. Because ports allow multiple outstanding messages, any process can receive a message from a port, and processes accessing ports are blocked until requests can be satisfied. Each port, which holds a fixed set of (one-word) messages, consists of a finite length message queue. Processes producing messages send them to the port with primitive *psend* (messages are deposited in FIFO order). The sending process can continue to execute after depositing its message as long as space remains in the port. If no space remains in the port, however, the sending process is blocked until messages have been removed and space becomes available.

Processes invoke primitive *preceive* to remove the next message from a port. Like *psend*, *preceive* operates synchronously, blocking the caller until a message is available.

## 15.10 The Implementation Of Ports

Each port consists of a queue to hold messages and two semaphores. One of the semaphores controls producers, blocking any process that attempts to add messages to a full port (i.e., one in which the current count of messages fills its quota). The other semaphore controls consumers, blocking any process that attempts to remove a message from an empty port.

Because ports are created at run-time, it is impossible to know the total count of items that will be enqueued at all ports at any given time. Although each message is small (one word), the total space required for port queues must be limited to prevent the

port procedures from using all the free space. To guarantee a limit on the total space used, the port procedures allocate a fixed number *MAXMSGS* of queue nodes and share them among all ports. Initially, these nodes are linked into a free list given by variable *ptfree*. Procedure *psend* takes a node from the free list and adds it to the queue at a specified port when sending a message; procedure *preceive* returns a node to the free list after the message has been received.

In file *ports.h*, structure *pt* defines the contents of an entry in the port table, and structure *ptnode* defines the contents of a message node. Most of the fields in *ptnode* are expected. We will comment on the sequence field *ptseq* in structure *pt* later.

```
/* ports.h - isbadport */

#define NPORTS          30              /* maximum number of ports      */
#define MAXMSGS         100             /* maximum messages on all ports*/
#define PTFREE          1               /* port is free                 */
#define PTLIMBO         2               /* port is being deleted/reset  */
#define PTALLOC         3               /* port is allocated            */
#define PTEMPTY         -1              /* initial semaphore entries    */

struct  ptnode  {                       /* node on list of message ptrs */
        int     ptmsg;                  /* a one-word message           */
        struct ptnode   *ptnext;        /* address of next node on list */
};

struct  pt      {                       /* entry in the port table      */
        char    ptstate;                /* port state (FREE/LIMBO/ALLOC)*/
        int     ptssem;                 /* sender semaphore             */
        int     ptrsem;                 /* receiver semaphore           */
        int     ptmaxcnt;               /* max messages to be queued    */
        int     ptct;                   /* no. of messages in queue     */
        int     ptseq;                  /* sequence changed at creation */
        struct  ptnode  *pthead;        /* list of message pointers     */
        struct  ptnode  *pttail;        /* tail of message list         */
};

extern  struct  ptnode  *ptfree;        /* list of free nodes           */
extern  struct  pt      ports[];        /* port table                   */
extern  int     ptnextp;                /* next port to examine when    */
                                        /*   looking for a free one     */

#ifdef  MEMMARK
extern  MARKER  ptmark;
#endif

#define isbadport(portid)       ( (portid)<0 || (portid)>=NPORTS )
```

### 15.10.1 Ports Initialization

Because initialization routines are designed after basic operations have been implemented, we have been discussing them after other routines. In the case of ports, we will discuss initialization first, because it may make the remaining routines easier to understand. File *pinit.c* contains the code to initialize ports and the declaration of the port table as well. Global variable *ptnextp* gives the index in array *ports* at which to start when searching for a free port. Initialization consists of marking each port free and forming the linked list of free message nodes. *Pinit* first allocates a block of memory using *getmem* and then moves through it, linking the individual message nodes together.

```
/* pinit.c - pinit */

#include <conf.h>
#include <kernel.h>
#include <mem.h>
#include <mark.h>
#include <ports.h>

#ifdef  MEMMARK
MARKER  ptmark;
#endif

struct  ptnode  *ptfree;        /* list of free queued nodes */
struct  pt      ports[NPORTS];
int             ptnextp;

/*------------------------------------------------------------------------
 *  pinit  --  initialize all of the ports
 *------------------------------------------------------------------------
 */

SYSCALL pinit()
{
        int             i;
        struct  ptnode  *next, *prev;
        int maxmsgs;
#ifdef  MEMMARK
        int status;

        if ( (status=mark(ptmark)) == SYSERR )
                panic("pinit - memory marking error");
        if ( status == 0 )
                return(OK);             /* already marked               */
#endif
        maxmsgs = MAXMSGS;
        if ((ptfree=(struct ptnode *)getmem(maxmsgs*sizeof(struct ptnode)))
                == (struct ptnode *)NULL ) {
#ifdef MEMMARK
                panic("pinit - insufficient memory");
#else
                return(SYSERR);
#endif
        }
        for (i=0 ; i<NPORTS ; i++)
```

```
                ports[i].ptstate = PTFREE;
        ptnextp = NPORTS - 1;

        /*  link up free list of message pointer nodes  */

        for ( prev=next=ptfree ; --maxmsgs > 0 ; prev=next )
                prev->ptnext = ++next;
        prev->ptnext = (struct ptnode *)NULL;
        return(OK);
}
```

The call to *panic* also deserves comment because this is its first occurrence. If there is insufficient space for creating a pool of message nodes, the system will be unable to provide any message passing services through ports. Port creation and use will be prohibited. *Panic* is designed for just such situations; it prints the specified error message and halts processing. (The exercises suggest alternative ways of handling the problem.)

Observe that with ports, all the memory required for message nodes is obtained using *getmem* at initialization time. Contrast this with buffer pools described earlier in this chapter. There, *getmem* is called only when a pool is created, not at initialization time.

### 15.10.2 Port Creation

Port creation consists of allocating an entry in the port table from among those that are free; procedure *pcreate* contains the code. After *pinit* establishes the list of free message nodes and fills in the port table, procedure *pcreate* creates a port and returns its table index to serve as a port identifier. It takes as an argument the maximum count of outstanding messages that the port should allow, permitting the caller to determine how many messages can be enqueued at a port before the sender blocks. Note that, like the buffer pool routines, the port procedures are self-initializing if memory marking is available.

```
/* pcreate.c - pcreate */

#include <conf.h>
#include <kernel.h>
#include <mark.h>
#include <ports.h>

/*------------------------------------------------------------------------
 *  pcreate  --  create a port that allows "count" outstanding messages
 *------------------------------------------------------------------------
 */

SYSCALL pcreate(count)
int count;
{
        int     ps;
        int     i, p;
        struct  pt      *ptptr;
        int     mkval;

        if ( count < 0 ||  count > MAXMSGS )
                return(SYSERR);
        disable(ps);
#ifdef  MEMMARK
        if ( unmarked(ptmark) )
                pinit();
#endif
        for (i=0 ; i<NPORTS ; i++) {
                if ( (p=ptnextp--) <= 0 )
                        ptnextp = NPORTS - 1;
                if ( (ptptr= &ports[p])->ptstate == PTFREE ) {
                        ptptr->ptstate = PTALLOC;
                        ptptr->ptssem  = screate(count);
                        ptptr->ptrsem  = screate(0);
                        ptptr->pthead  = ptptr->pttail
                                       = (struct ptnode *)NULL;
                        ptptr->ptseq++;
                        ptptr->ptmaxcnt = count;
                        ptptr->ptct = 0;
                        restore(ps);
                        return(p);
                }
        }
        restore(ps);
        return(SYSERR);
}
```

The basic operations on ports, sending and receiving messages, are handled by routines *psend* and *preceive*. They each require the caller to specify the port on which the operation is to be performed by passing the port identifier as an argument.

### 15.10.3 Sending To Ports

Procedure *psend* adds a message to those that are waiting at the port. It waits for space in the port, enqueues the message given by its argument, signals the receiver semaphore to indicate another message is available, and returns.

```
/* psend.c - psend */

#include <conf.h>
#include <kernel.h>
#include <mark.h>
#include <ports.h>

/*------------------------------------------------------------------------
 *  psend  --  send a message to a port by enqueuing it
 *------------------------------------------------------------------------
 */

SYSCALL psend(portid, msg)
int portid;
int msg;
{
        int     ps;
        struct  pt      *ptptr;
        int     seq;
        struct  ptnode  *freenode;

        disable(ps);
        if ( isbadport(portid) ||
#ifdef  MEMMARK
                unmarked(ptmark) ||
#endif
                (ptptr=&ports[portid])->ptstate != PTALLOC ) {
                restore(ps);
                return(SYSERR);
        }

        /* wait for space and verify that the port is still allocated */

        seq = ptptr->ptseq;
        ptptr->ptct++;
        if (wait(ptptr->ptssem) == SYSERR
                || ptptr->ptstate != PTALLOC
                || ptptr->ptseq != seq ) {
                        restore(ps);
                        return(SYSERR);
                }
        if ( ptfree == (struct ptnode *)NULL ) {
                kprintf("psend - out of nodes");
                xdone();
```

```
        }
        freenode = ptfree;
        ptfree = freenode->ptnext;
        freenode->ptnext = (struct ptnode *)NULL;
        freenode->ptmsg = msg;
        if ( ptptr->pttail == (struct ptnode *)NULL )   /* empty queue */
                ptptr->pttail = ptptr->pthead = freenode;
        else {
                (ptptr->pttail)->ptnext = freenode;
                ptptr->pttail = freenode;
        }
        signal(ptptr->ptrsem);
        restore(ps);
        return(OK);
}
```

The initial code in *psend* merely verifies that the argument is valid. What happens next is similar to the implementation of windows. *Psend* waits on the "sender" semaphore. The call returns immediately if the port is not full, but it delays if the number of messages already enqueued equals the maximum allowed. *Psend* records the value of *ptseq* before the call to *wait*, and then verifies that it remained the same after the call. Similar to closing and opening windows, ports may be deleted (and even recreated) while processes remain enqueued waiting to send to them. When this happens, the port sequence number changes and the waiting processes are resumed. So waiting processes must verify that the *wait* did not terminate because the port was deleted. If it did, and the sequence number changed, *psend* reports an error to its caller.

*Psend* enqueues messages in FIFO order. It relies on *pttail* to point to the last node if the queue is nonempty, and it leaves *pttail* pointing to the new node after adding the node to the list. It signals semaphore *ptrsem* once the new message has been added to the queue, allowing a receiver to consume the message. The invariant being maintained is:

> *Semaphore ptrsem has nonnegative count n if n messages are waiting in the port; it has negative count –n if n processes are waiting for messages.*

In our design, running out of message nodes is a catastrophe from which the system cannot recover. It means that the arbitrary limit on message nodes, set to prevent ports from using all the free memory, is insufficient. Perhaps the programs using ports are operating incorrectly. Perhaps, through no fault of the user, the system cannot honor a valid request; there is no way to know. Under such circumstances, it is often better to announce failure and stop rather than attempt to go on. Consequently, *psend* calls *panic* in case there are no more message nodes available.

### 15.10.4 Receiving From Ports

Procedure *preceive* implements the basic consumer operation. It removes a message from a port and returns it to the caller.

```
/* preceive.c - preceive */

#include <conf.h>
#include <kernel.h>
#include <mark.h>
#include <ports.h>

/*------------------------------------------------------------------------
 *  preceive  --  receive a message from a port, blocking if port empty
 *------------------------------------------------------------------------
 */

SYSCALL preceive(portid)
int portid;
{
        int     ps;
        struct  pt      *ptptr;
        int     seq;
        int     msg;
        struct  ptnode  *nxtnode;

        disable(ps);
        if ( isbadport(portid) ||
#ifdef MEMMARK
                unmarked(ptmark) ||
#endif
                (ptptr=&ports[portid])->ptstate != PTALLOC ) {
                restore(ps);
                return(SYSERR);
        }

        /* wait for a msg. and verify that the port is still allocated */

        seq = ptptr->ptseq;
        if ( wait(ptptr->ptrsem) == SYSERR
            || ptptr->ptstate  != PTALLOC
            || ptptr->ptseq != seq ) {
                restore(ps);
                return(SYSERR);
```

```
        }

        /* dequeue the first message that is waiting in the port */

        nxtnode = ptptr->pthead;
        msg = nxtnode->ptmsg;
        if (ptptr->pthead == ptptr->pttail)     /* delete the last item */
                ptptr->pthead = ptptr->pttail = (struct ptnode *)NULL;
        else
                ptptr->pthead = nxtnode->ptnext;
        nxtnode->ptnext = ptfree;               /* return to free list */
        ptfree = nxtnode;
        ptptr->ptct--;
        signal(ptptr->ptrsem);
        restore(ps);
        return(msg);
}
```

*Preceive* waits until a message is available, verifies that the port was not deleted, and dequeues the message node. It records the message in local variable *msg* before returning the message node to the free list.

## 15.11 Other Operations On Ports

It is sometimes useful to determine the number of messages in a port, to delete a port, or to reset it. In the latter two cases, the system must dispose of waiting messages, return message nodes to the free list, and permit waiting processes to continue execution.

Counting the number of messages in a port appears to be trivial. The code is indeed trivial:

```
/* pcount.c - pcount */

#include <conf.h>
#include <kernel.h>
#include <mark.h>
#include <ports.h>

/*------------------------------------------------------------------------
 *  pcount  --  return the count of current messages in a port
 *------------------------------------------------------------------------
 */
SYSCALL pcount(portid)
int     portid;
{
        int     scnt;
        int     count;
        int     ps;
        struct  pt      *ptptr;

        disable(ps);
        if ( isbadport(portid) ||
#ifdef MEMMARK
                unmarked(ptmark) ||
#endif
                (ptptr= &ports[portid])->ptstate != PTALLOC ) {
                        restore(ps);
                        return(SYSERR);
        }
        count = ptptr->ptct;
        restore(ps);
        return(count);
}
```

Should the message count reflect the actual number of message nodes in the port queue, or should it take into account processes that are waiting to send or receive messages as well? A process waiting on the port's receiver semaphore *ptrsem* is a guaranteed producer of a message which will eventually end up on the port's message queue. If a process is interested in determining arriving messages in transit as well as actual messages in the queue, waiting producers should be taken into account in determining the count of messages. Conversely, a process waiting on the port's sender semaphore *ptrsem* is a guaranteed consumer of a message. If a process wishes to acknowledge that some messages are already ''spoken for,'' waiting consumers should be taken into account too. For these reasons, the *ptct* component of a port table entry is incremented at each call to *psend* – reflecting a potential production of a message- and is decremented at each call to

*preceive* – reflecting a potential consumption. (The exercises consider other ways of computing *ptct*.)

When deleting or resetting a port, how should the port mechanism dispose of waiting messages? It could choose to throw them away, or it might return them to the processes that sent them. Often, the user can describe a more meaningful disposition, so the design presented here allows the user to specify disposition. Procedures *pdelete* and *preset* perform port deletion and reset operations. Both take as an argument a function that will be called to dispose of each waiting message. The code is found in files *pdelete.c* and *preset.c*:

```
/* pdelete.c - pdelete */

#include <conf.h>
#include <kernel.h>
#include <mark.h>
#include <ports.h>

/*------------------------------------------------------------------------
 *  pdelete  --  delete a port, freeing waiting processes and messages
 *------------------------------------------------------------------------
 */

SYSCALL pdelete(portid, dispose)
int portid;
int (*dispose)();
{
        int     ps;
        struct  pt      *ptptr;

        disable(ps);
        if ( isbadport(portid) ||
#ifdef MEMMARK
                unmarked(ptmark) ||
#endif
                (ptptr=&ports[portid])->ptstate != PTALLOC ) {
                restore(ps);
                return(SYSERR);
        }
        _ptclear(ptptr, PTFREE, dispose);
        restore(ps);
        return(OK);
}
```

```
/* preset.c - preset */

#include <conf.h>
#include <kernel.h>
#include <mark.h>
#include <ports.h>

/*------------------------------------------------------------------------
 *  preset  --  reset a port, freeing waiting processes and messages
 *------------------------------------------------------------------------
 */

SYSCALL preset(portid, dispose)
int portid;
int (*dispose)();
{
        int     ps;
        struct  pt      *ptptr;

        disable(ps);
        if ( isbadport(portid) ||
#ifdef MEMMARK
                unmarked(ptmark) ||
#endif
                (ptptr=&ports[portid])->ptstate != PTALLOC ) {
                restore(ps);
                return(SYSERR);
        }
        _ptclear(ptptr, PTALLOC, dispose);
        restore(ps);
        return(OK);
}
```

Both *pdelete* and *preset* verify that their arguments are correct and then call *_ptclear* to perform the work of clearing messages and waiting processes.

*_Ptclear* places the port in a "limbo" state while clearing it. The limbo state guarantees that no other processes can use the port — procedures like *psend* and *preceive* will refuse to operate on a port that is not allocated, and *pcreate* will not allocate the port unless it is free. Thus, *_ptclear* can allow rescheduling while it clears the port.

Before declaring a port eligible for use again, *_ptclear* calls *dispose* repeatedly, passing it each waiting message. Finally, after all messages have been removed, *_ptclear* deletes or resets the semaphores as specified by its second argument. Before disposing of messages, *_ptclear* increments the port sequence number so that waiting processes can tell that the port has changed when they regain control of the CPU.

```
/* ptclear.c - _ptclear */

#include <conf.h>
#include <kernel.h>
#include <mark.h>
#include <ports.h>

/*------------------------------------------------------------------------
 *  _ptclear  --  used by pdelete and preset to clear a port
 *------------------------------------------------------------------------
 */
_ptclear(ptptr, newstate, dispose)
        struct  pt      *ptptr;
        int     newstate;
        int     (*dispose)();
{
        struct  ptnode  *p;


        /* put port in limbo until done freeing processes */

        ptptr->ptstate = PTLIMBO;
        ptptr->ptseq++;
        if ( (p=ptptr->pthead) != (struct ptnode *)NULL ) {
                for(; p != (struct ptnode *)NULL ; p=p->ptnext)
                        (*dispose)( p->ptmsg );
                (ptptr->pttail)->ptnext = ptfree;
                ptfree = ptptr->pthead;
        }
        if (newstate == PTALLOC) {
                ptptr->pttail = ptptr->pthead = (struct ptnode *)NULL;
                sreset(ptptr->ptssem, ptptr->ptmaxcnt);
                sreset(ptptr->ptrsem, 0);
                ptptr->ptct = 0;
        } else {
                sdelete(ptptr->ptssem);
                sdelete(ptptr->ptrsem);
        }
        ptptr->ptstate = newstate;
}
```

## 15.12 Summary

High-level memory management and message passing primitives are needed to build disk driver and network software. The memory manager must prevent global exhaustion, and the message passing facility must provide buffered rendezvous points for message exchange. This chapter introduced two sets of primitives that satisfy these needs. Instead of building these primitives into the device software, we have chosen to design and build them separately. Such separation makes the primitives available to other system software and to user programs. It also makes them easier to test.

The high-level memory manager consists of buffer pool routines. Buffer pools permit memory to be partitioned, such that the number of buffers in a given pool is fixed. Once a pool has been created, processes can allocate and deallocate buffers from the pool without affecting other free memory. A process that attempts to allocate a buffer from an empty pool is blocked until another process returns a buffer to the pool.

The high level message passing mechanism, called communication ports, permits processes to exchange messages efficiently. Each port consists of a fixed length queue of messages. The basic operations *psend* and *preceive* deposit a message at the tail of the queue and extract a message from the head of the queue. Processes that attempt to receive from an empty port are blocked until a message arrives, and processes that attempt to send to a full port are blocked until space becomes available.

Both buffer pools and communication ports use another mechanism introduced in this chapter, namely memory marking. Memory marking provides an efficient way to determine whether a memory location has been ‘‘marked’’ since system startup or to ‘‘mark’’ a given location. As demonstrated by the buffer pool and port routines, memory marking can be used to make a set of procedures self-initializing.

## FOR FURTHER STUDY

Calingaert [1982] discusses allocation from a pool as well as alternative methods of recovering storage. Knuth [1968] describes in detail two such recovery methods: the ‘‘buddy system’’ and garbage collection. Memory marking is akin to the constant-time array initialization problem in Exercise 2.12 of Aho et. al. [1974].

## EXERCISES

**15.1** The chief advantage of combining procedures into a single file is that they can share static data structures. Consider, for example, the memory marking routines. Recode them so that *marks* is protected from access except through *mark* and *unmark*.

**15.2** Can you envision an alternative to the three state marking scheme described above?

**15.3** Design a new *getmem* that subsumes *getbuf*. Hint: allow the user to suballocate from a previously allocated block of memory.

**15.4** Explain how to modify *getbuf* so it does not allocate buffers until they are needed, even though it still limits the number of buffers that can be simultaneously allocated from a pool at a given time.

**15.5** Why is *freebuf* more efficient than *freemem*?

**15.6** The implementation of memory marking described here does not provide adequate mutual exclusion for two or more processes that attempt to access self-initializing routines concurrently. Redesign the primitives by introducing a third primitive, *marking*, so a process can declare that a cell is ''being marked'' during module initialization. Have concurrent calls to *mark* and *unmarked* block until the caller declares that the cell is ''marked.''

**15.7** Instead of putting the pool id at the beginning of the buffer, redesign the buffer pool mechanism to put the id at the end of the buffer. Explain why this may result in less efficiency.

**15.8** Consider the primitives *send—receive* and *psend—preceive*. Design a single message passing scheme that encompasses both.

**15.9** An important distinction is made between statically allocated and dynamically allocated resources. For example, *ports* are dynamically allocated while inter-process message slots are statically allocated. What is the key problem with dynamic allocation in a multi-process environment?

**15.10** Change the message node allocation scheme so that a semaphore controls nodes on the free list. Have *psend* wait for a free node if none exists. What potential problems, if any, does the new scheme introduce?

**15.11** Panic is used for conditions like internal inconsistency or potential deadlock. Often the conditions causing a panic are irreproducible, so their cause is difficult to pinpoint. Discuss what you might do to trace the cause of the panic in *psend*.

**15.12** As alternatives to the panic in *psend*, consider allocating more nodes or retrying the operation. What are the liabilities of each?

**15.13** Rewrite *psend* and *preceive* to return a special value when the port is deleted while they are waiting.

**15.14** Modify the routines in previous chapters that allocate, use, and delete objects so they use sequence numbers to detect deletion as the window and ports routines do.

**15.15** *Psend* and *preceive* cannot transmit a message with value equal to *SYSERR* because *preceive* cannot distinguish between a message with that value and an error. Redesign them to transmit any value.

**15.16** Why put a restriction on the number of messages per port?

**15.17** Rewrite the tty input code to use a buffer pool scheme for allocating small input buffers for all tty input operations, generalizing the fixed input buffer size in the *tty* structure.

**15.18** Show that the actual number of message nodes in the queue for a port is

$$ptmaxcnt\text{-}scount(ptssem)+scount(ptrsem)\text{-}ptct.$$

The variables above all come from structure *pt*.