8

Memory Management

Main memory ranks high among the important resources that an operating system manages because it is essential for program execution. The operating system keeps track of the location and size of available free space, allocating it on demand, and recovering it when processes complete. In large computer systems, where the demand for memory at a given time often exceeds the total available, the system must multiplex real memory among processes waiting to use it. Memory multiplexing can take the form of *swapping*, where entire processes are written to secondary store when they are not using the CPU, or *paging*. A paged system divides each program into small fixed size pieces called *pages*, keeping all but the most recently referenced pages on secondary storage. Programmers do not worry about swapping or paging because the system performs these activities in a way that is transparent to the running programs.

Often, paged systems do more than multiplex the real memory among processes – they supply each process with its own, independent address space. These so-called *virtual* address spaces can be larger than the real memory on the machine because the paging system keeps the virtual image on disk, moving only the small subset of pages being referenced to main memory. When a process references memory location *i* in its address space, the hardware consults memory mapping tables to determine if the page on which the reference lies currently resides in main memory. If not, the system suspends the process and loads the page, writing some other page back to secondary storage if necessary to make room for the new one. Finally, when the page has been loaded, process execution resumes.

To be efficient, memory management (especially paging) requires hardware support. If a memory management mechanism is designed well, the operating system can use it to partition memory in such a way that the hardware prevents one process from reading or writing memory allocated to another process. Such protection is essential in environments where processes are not friendly, or where security is important; it is convenient in almost any environment because it helps detect programming errors.

8.1 Memory Management On The 8088

Unfortunately, the 8088 hardware poorly manages multiple address spaces and cannot protect processes from one another. As a consequence, the PC-Xinu system and all processes occupy portions of the same text and data address spaces which are arranged as shown in Figure 8.1:

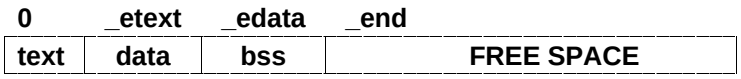

Figure 8.1 Storage layout when PC-Xinu begins

The program text occupies the lowest part of memory, followed by the global variables in the *data* segment. At system startup, PC-Xinu initializes global variable *maxaddr* to the address of the highest memory location addressable in the data segment, so the initial size of the free space can be determined in a C program.

Maintaining processes using a single data space is certainly not ideal, but it does have some advantages. For one thing, processes can pass pointers among themselves and the operating system easily, because the interpretation of an address does not depend on the process context. The ability to share data is another advantage: because the processes share global variables, they can exchange large amounts of data without copying it. Finally, having just one address space for data makes the memory management routines much simpler to implement than those found in other systems.

8.2 Dynamic Memory Requirements In PC-Xinu

PC-Xinu requires program text and all global data to remain resident in main memory at all times. However, program text and global data account for only part of the space required by an executing process. Each process also needs space for a stack to hold procedure frames and local variables, as well as so-called *heap* space for other dynamically allocated variables. PC-Xinu allocates stack memory from the lowest addresses in free space, producing a run-time allocation like that shown in Figure 8.2.

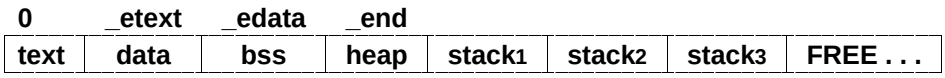

Figure 8.2 Storage layout during execution

This chapter explores the procedures and data structures that manage the free memory, allocate space for stack and heap storage, and keep track of storage that has been released. At this level, free space is treated as an exhaustible resource – the system simply passes it out as long as requests can be satisfied. A process that cannot obtain

memory must decide for itself whether to try again. Exhaustive allocation works only when processes cooperate to keep from consuming all free memory. (Chapter 15 explores a set of high level memory management routines that prevent exhaustion by partitioning memory and blocking requesting processes until memory becomes available.)

8.3 Low-Level Memory Management Procedures

Two procedures, *getmem* and *freemem*, obtain and release process stack space. Recall that *create* used procedure *getmem* to allocate stack space when forming a new process. *Getstk* obtains a block of memory and returns its address. *Create* records the size and location of the allocated space in the process table entry. Later, when the process terminates, *kill* calls procedure *freemem* to return the process' stack space to the free list again. *Freemem* expects as arguments the address of the block being returned and the size. *Getmem* and *freemem* also obtain and release blocks of free space for other system purposes.

Because only *create* and *kill* allocate and free process stacks, PC-Xinu guarantees that the stack space allocated to a process will be released at process exit. Unfortunately, the system cannot guarantee that other space allocated by *getmem* will be released, because it does not record such allocations on a process by process basis. Thus, the burden of returning heap space is left to the user program:

> *A process must release storage that it allocates from the heap before it exits.*

Of course, returning allocated space does not guarantee that the heap will never be exhausted. The demand could still exceed the available space, or the free space could be fragmented into small, discontiguous pieces. But releasing space avoids needless exhaustion.

8.4 The Location Of Allocated Storage

It is desirable to have stack space and heap storage allocated from opposite ends of the available space. Because the hardware grows stacks downward, such an allocation technique is ideal for a single user process because all remaining free space separates the heap and stack. If the stack overflows accidentally, it runs into free space located between the stack and the heap, instead of data. When more than one process executes concurrently, the situation is not as pleasant. Stack overflow in one process corrupts data in the stack of another because the system allocates stacks contiguously. In fact, stack overflow is one of the most common problems in PC-Xinu. The exercises in Chapter 4 suggested one solution – place an uncommon value at the base of each process stack, and have *resched* check both the current process stack size and the value at the base of the stack before the process resumes.

8.5 The Implementation Of PC-Xinu Memory Management

PC-Xinu keeps blocks of free memory linked together on a list with global variable *memlist* pointing to the first free block. An important invariant maintained by all these procedures is:

Blocks on the free list are ordered by increasing address.

While on the free list, each block contains, in the first two words, a pointer to the next block and the size of the current block. The record *memlist* is also declared to have the same form as all blocks on the free list. File *mem.h* contains the pertinent definitions in C:

```
/* mem.h - roundew, truncew, getstk, freestk */

/*------------------------------------------------------------------------
 * roundew, truncew - round or truncate address to next even word
 *------------------------------------------------------------------------
 */
#define roundew(x)      ( (3 + (WORD)(x)) & (~3) )
#define truncew(x)      ( ((WORD)(x)) & (~3) )

#define getstk(n)       getmem(n)
#define freestk(b,s)    freemem(b,s)

struct  mblock  {
        struct  mblock  *mnext;
        word    mlen;
        };

#define end     endaddr                 /* avoid C library conflict     */

extern  struct  mblock  memlist;        /* head of free memory list     */
extern  char    *maxaddr;               /* max memory address           */
extern  char    *end;                   /* address beyond loaded memory */

#define MMAX    65024                   /* maximum memory size          */
#define MBLK    512                     /* block size for global alloc  */
#define MMIN    8192                    /* minimum Xinu allocation      */
#define MDOS    1024                    /* save something for MS-DOS    */

extern  char    *getmem();
extern  int     freemem();
```

Structure *mblock* gives the shape of each node on the free list. Field *mnext* always points to the next block (or contains *NULL*), and field *mlen* gives the length of the current block in bytes, including the two-word header.

File *mem.h* introduces two in-line functions, *roundew* and *truncew*. Because only blocks of two words (4 bytes) or more can be linked onto the free list, PC-Xinu refuses to allocate or free smaller quantities of memory. To be sure that requests specify correct amounts of memory, the memory management routines use *roundew* or *truncew* as needed to round or truncate to an even number of words, by making the number of bytes a multiple of four.

8.5.1 Allocating Storage

Procedure *getmem* allocates storage. The code for *getmem* appears below in file *getmem.c*. It uses *roundew* to round up the memory request and then searches the memory list to find the first block of memory large enough to satisfy the request. Because the list of free blocks is singly-linked, *getmem* uses two pointers, *p* and *q*, to search it. When *p* points to a block of suitable size, *q* points to its predecessor on the list (possibly the head, *memlist*). If the size of a free block exactly matches the size of the request, *getmem* merely deletes the block from the free list and returns its address. If the size of the free block is greater than the size requested, *getmem* partitions off a piece of size *nbytes* and links the remainder back on the free list. It returns the address of the allocated piece to the caller.

When a block on the free list must be divided, variable *leftover* points to the piece that must be left on the free list. Computing such an address is conceptually simple: the leftover piece lies *nbytes* beyond the beginning of the block. However, adding *nbytes* to pointer *p* does not produce the desired result because C performs pointer arithmetic. To force C to use integer arithmetic instead of pointer arithmetic, *p* is changed to a *char* pointer using a cast (i.e., ‘‘(*char* *)*p*’’), and then the result is changed back into a *struct mblock* pointer with another cast.

```
/* getmem.c - getmem */

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

/*------------------------------------------------------------------------
 *  getmem  --  allocate heap storage, returning lowest integer address
 *------------------------------------------------------------------------
 */
char *getmem(nbytes)
word nbytes;
{
        int     ps;
        struct  mblock  *p, *q, *leftover;

        disable(ps);
        if ( nbytes==0 ) {
                restore(ps);
                return( NULL );
        }
        nbytes = roundew(nbytes);
        for ( q=&memlist, p=q->mnext ;
                (char *)p != NULL ;
                q=p, p=p->mnext )
                if ( p->mlen == nbytes) {
                        q->mnext = p->mnext;
                        restore(ps);
                        return( (char *) p );
                } else if ( p->mlen > nbytes ) {
                        leftover = (struct mblock *)((char *)p + nbytes);
                        q->mnext = leftover;
                        leftover->mnext = p->mnext;
                        leftover->mlen = p->mlen - nbytes;
                        restore(ps);
                        return( (char *) p );
                }
        restore(ps);
        return( NULL );
}
```

8.5.2 Releasing Storage

Processes return previously allocated memory to the list of free blocks when they finish using it so it can be given out again. System call *freemem* returns a block of storage by inserting it in the proper location of the free list and coalescing it with any adjacent free blocks. As in *getmem*, pointers *p* and *q* run down the list of free blocks. Because the list is kept in order by block address, *freemem* stops searching as soon as the address of the block to be returned lies between *p* and *q*.

Special cases complicate the code that links the returned block into the free list. The new block may lie adjacent to free blocks above or below, or both. When these cases arise, *freemem* groups adjacent free blocks together to form larger blocks. (Failure to do so would eventually *fragment* the free list into small pieces.)

A pitfall can be avoided by remembering that the new block may be adjacent to free blocks on both sides. As shown in file *freemem.c*, the code always checks to see whether the new block is adjacent to the block following, even if it coalesces the new block with the previous one:

```
/* freemem.c - freemem */

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

/*------------------------------------------------------------------------
 *  freemem  --  free a memory block, returning it to memlist
 *------------------------------------------------------------------------
 */
SYSCALL freemem(block, size)
char *block;
word size;
{
        int     ps;
        struct  mblock  *p, *q;
        char    *top;

        size = roundew(size);
        block = (char *) truncew( (word)block );
        if ( size==0 || block > maxaddr || (maxaddr-block) < size ||
                block < end )
                return(SYSERR);
        disable(ps);
        (char *)q = NULL;
        for( p=memlist.mnext ;
                (char *)p != NULL && (char *)p < block ;
                q=p, p=p->mnext )
                        ;
        if ( (char *)q != NULL && (top=(char *)q+q->mlen) > block
                    || (char *)p != NULL && (block+size) > (char *)p ) {
                restore(ps);
                return(SYSERR);
        }
        if ( (char *)q != NULL && top == block )
                q->mlen += size;
        else {
                ((struct mblock *)block)->mlen = size;
                ((struct mblock *)block)->mnext = p;
                memlist.mnext = (struct mblock *)block;
                (char *)q = block;
        }
        /* note that q != NULL here */
        if ( (char *)p != NULL
```

```
                && ((char *)q + q->mlen) == (char *)p) {
                q->mlen += p->mlen;
                q->mnext = p->mnext;
        }
        restore(ps);
        return(OK);
}
```

8.6 Summary

The lowest level of the PC-Xinu memory manager maintains a linked list of all free storage, allocating storage on demand and adding it back to the free list when requested to do so. The free list is ordered by address. Memory is allocated from the lowest free memory addresses using the first-fit algorithm.

At this level, memory is considered an exhaustible resource, given out without constraint until none remains free. The low-level memory manager simply rejects requests that cannot be satisfied; there are no mechanisms to prevent processes from using all the free memory or to block processes until their requests can be satisfied. Higher layers of the memory manager that provide these mechanisms are discussed in Chapter 15.

FOR FURTHER STUDY

Memory management has received wide attention in the literature. The basic algorithm used here is called ‘‘first-fit’’ in Knuth [1968], where alternatives like ‘‘best-fit’’ and ‘‘buddy’’ are also considered. Comparisons of first-fit and best-fit can be found in the articles by Shore [1975] and Bays [1977].

Much of the research in memory management has centered on discovering and analyzing policies for paging and swapping on systems that support virtual address spaces. Peterson and Silbershatz [1983] devote an entire chapter to virtual memory. A key problem involves selecting which program or part of a program to write back onto secondary store when a new page must be brought into memory. Denning [1970, 1980] surveys virtual storage systems, describing the research. Madnick and Donovan [1974] describe commercial hardware that supports paging and describe its use. Other good descriptions of memory management can be found in Calingaert [1982], Habermann [1976], and Tsichritzis and Bernstein [1974].

EXERCISES

8.1 An early version of *getmem* had no provision for returning memory to the free list. Speculate about microcomputer applications: is *freemem* necessary?

8.2 Implement the smallest routines possible for memory allocation, assuming there is no need to return storage to a free list. How does the size of the new allocation routine compare to the size of *getmem*?

8.3 Allocating and deallocating blocks of varying size can *fragment* memory, leaving many small pieces on the free list. Investigate schemes other than ''first-fit'' for choosing a block of memory from the free list.

8.4 PC-Xinu applies the ''first-fit'' method from one end of the free list. Implement a way to even out the allocation of memory across the free memory area by beginning the current search where the last allocation left off (wrapping around memory if necessary).

8.5 PC-Xinu merely reports an error when no block of memory exists that can satisfy the request. Consider an alternative in which the calling process is merely delayed (say, by placing it on a queue) until a sufficiently large piece of memory has been returned to the free list. Explain how all processes might eventually be enqueued waiting for memory, even though no process requests more memory than exists.

8.6 Carefully consider a program that allocates a large block of memory, frees half of it, frees the other half, and then repeats the actions. Under what circumstances will the program fail? Do you consider this a problem?

8.7 Modify the memory management routines to allow allocation of arbitrarily small blocks of memory.

8.8 Investigate the buddy system for memory allocation. Would it be wise to use it in PC-Xinu?

8.9 Find out how the 80286 processor manages memory. Design software support routines to take advantage of the hardware.

8.10 Examine the Motorola 68000 and PDP-11 memory management schemes. What layers of software are needed at the lowest level of the system to implement demand paging?

8.11 Explore *segmentation* as an alternative to paging. What are its advantages? Does paging a segmented memory make sense?