8 Xinu Memory Managment

- The operating system keeps track of location and size of the available free space, allocating it on demand, and recovering it when processes complete.
- In large operating system deference from Xinu, the demand for memory by processes may exceed the total amount of core/primary memory available, hence the system must multiplex real memory among processes waiting for use it.
- memory multiplexing can take form of :
  - **swapping** or
  - **paging**

swapping, paging (1)

- Swapping: entire processes are written to secondary (disk) storage when they are using the CPU.
- Paging: each program is divided into small fixed-size pieces called **pages**. The system keeps only the most recently referenced pages in primary storage (i.e., main memory).
- Programmers don't notice swapping and paging because the system hides it by appropriate hardware.

swapping, paging (2)

- In addition to pages, paged system supply each process with its own independent address space
- This **virtual** address space can be larger than real memory because the paging system keeps the virtual image on disk, moving only the small subset of referenced pages to main memory
- When a process references location i in its address space, the hardware consults memory mapping tables to see whether the page with location i resides in main memory or not
- If not, the operating system suspends the process, loads the requested page from secondary to primary memory, making room for it, when needed, and when it is there, resumes execution of the process
- Memory management requires hardware support. When done well
  - the operating system can portion memory in such a way that hardware prevents a process from reading or writing memory allocated to another process
  - This supply an especially important aspect for security

8.1 Memory Management on the 11/2

- This chapter describes procedures and data structures managing the free memory that allocate space for stack and heap storage, and keep track of released storage.
- Free space is treaded as an **exhaustable** resource hande out as long as it is available and quests can be satisfied
- A process which can not obtain memory must decide for itself when to try again
- Exhaustable allocation worths only processes cooperate to keep from consuming all free memory.

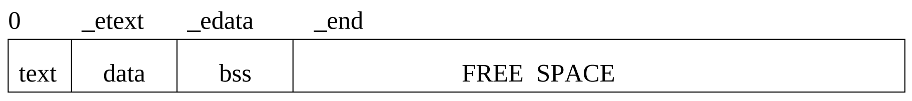

Figure 8.1: Storage layout when Xinu begins

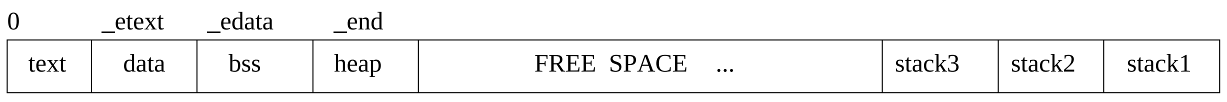

Figure 8.2: Storage layout during execution

8.2 Low-Level Memory Management Procedure

- *getstk, freestk*: used by *create* and *kill*:
  - `create : saddr = getsth(ssize) ... pg. 75`
  - `kill : freestk(pptr->pbase, pptr->pstklen) ... pg. 77`
- *getstk* obtains a block of memory and returns its highest address, at which the stack of the created process initially starts. It has tow args.:
  - highest address of block being returned
  - its size
- *create* records size and location of the allocated space in the process table entry of a process
- *kill* returns this stack space to free memory space by calling *freestk* .

Xinu guarantees that the stack space allocated to a process is released at process exit.

Heap (de)allocation, stack (de)allocation

- For the heap no deallocation procedures are built into Xinu

  - deallocation must be done on a process-by-process basic

  A process must release storage that it allocates from the heap befor it exit by itself. (This dosen't happen automatically)

- Allocation on the heap is done by procedure *getmem*
- Deallocation on the heap is done by procedure *freemem*
  - *getmem. freemem* reference blocks by their lowest address
  - *freestk* is implemented using *freemem*

8.3 Location Of Allocated Storage

- Separating heap from the stack storage works best when a singe process executes :
  - if (single) stack overflow occurs, the process runs into the free space located between stack and heap
- A problem arises when more than one process executes
  - Stack overflow that causes corruption of data ... an exercise suggest a solution.

8.4 Implementation Of Xinu Memory Management

```
/* mem.h - freestk, roundew, truncew */
/*----------------------------------------------------------
 * roundew, truncew - round or truncate address to
 * next even word
 *----------------------------------------------------------
 */
#define roundew(x)   (int *)( (3 + (int)(x)) & (-3)
#define truncew(x)   (int *)( ((int)(x)) & (-3) )

/*----------------------------------------------------------
 * freestk -- free stack memory allocated by getstk
 *----------------------------------------------------------
 */
#define freestk(p, len)  freemem((unsigned)(p)
                                 - (unsigned)(roundew(len))
                                 + (unsigned)sizeof(int),
                                 roundew(len) )
struct mblock {
       struct  mblock *mnext;
       unsigned int   mlen;
       };
extern struct  mblock memlist;/* head of free memory list*/
extern int     *maxaddr;      /* max memory address      */
extern int     end;      /* address beyond loaded memory */
```

- The free memory FREEMEM is organized as follows :
  - Blocks of free memory linked together on a list with global variable MEMLIST pointing to the first free block.
- Blocks on this free list are ordered by increasing addresses
- While on the free list, each block contains in its first 2 words
  - a pointer to the next free block
  - size of the current block
- Only blocks of 2 words or more can be linked into the free list, i.e., of 4 bytes or more, making the number of bytes a multiple of 4

8.4.1 Allocating Heap Storage

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

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

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

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

- Procedure *getmem(nbytes)* allocates heap storage returning the lowest integer address
- *getmem* search the memory list to find block of memory large enough to contain *roundew(nbytes)* bytes of free storage
- Because the list of free blocks is singly linked, 2 pointers p and q, to search it.
  - When p points to a block of suitable size, q points to its predecessor on the list.

8.4.2 Allocating Stack Storage

- *getstk* must search the entire list of free blocks because the list is kept in ascending order by block address, and the desired block is the one with the highest address satisfying the request.
- **fits** and **fitsq** record the values of p and q each time a block satisfies the request.
  - When search is completed: fits points to the free block that last satisfied this request.
  - fits points to its predecessor
- **getstk(nbytes)* makes full use of C's powerful for statement. See pg. 382 for its semantics

```
/* getstk.c - getstk*/

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

/*------------------------------------------------------------
 * getstk -- allocate stack memory, returning address of
 * topmost int
 *------------------------------------------------------------
 */
int   *getstk(nbytes)
      unsigned  int nbytes;
{
      char    ps;
      struct  mblock  *p, *q; /* q follows p along memlist*/
      struct  mblock  *fits, *fitsq;
      int     len;

      disable(ps);
      if (nbytes == 0) {
             restore(ps);
             return( (int *)SYSERR );
      }
      nbytes = (unsigned int)roundew(nbytes);
```

```
        fits = NULL;
        q = &memlist;
        for (p = q->mnext ; p != NULL ; q = p, p = p->mnext)
               if ( p->mlen >= nbytes){
               fitsq = q;
                       fits = p;
          }
        if (fits == NULL){
                restore(ps);
                return( (int *)SYSERR );
        }
        if (nbytes == fits->mlen) {
                  fitsq->mnext = fits->mnext
                  len = nbytes;
        } else {
          len = fits->mlen;
                  fits->mlen -= nbytes;
        }
        fits = ((int)fits) + len - sizeof(int);
        *( (int *) fits ) = nbytes;
        restore(ps);
        return( (int *)fits );
}
```

8.4.3 releasing Storage

- System call *freemem* returns a block of storage by inserting it at the proper location in the free list and closing it with any adjacent free blocks.
- *freemem* stops searching as soon as address of the block to be returned lies between p and q.

```
SYSCALL freemem(block, size)
        struct mblock *block;
        unsigned size;
{
        char    ps;
        struct mblock *p, *q;
        unsigned top;

        if (size==0 || (unsigned)block>(unsigned)maxaddr
                    || ((unsigned)block)<((unsigned)&end))
               return(SYSERR);
        size = (unsigned)roundew(size);
        disable(ps);
        for( p=memlist.mnext,q= &memlist ; (char *)p!=NULL &&
                  p<block ; q=p,p=p->mnext )
                  ;
        if ((top=q->mlen+(unsigned)q)>(unsigned)block &&
                  q!= &memlist || (char *)p!=NULL &&
                  (size+(unsigned)block) > (unsigned)p) {
                restore(ps);
                return(SYSERR);}



        if ( q!= &memlist && top == (unsigned)block )
```

```
                q->mlen += size;
        else {
                block->mlen = size;
                block->mnext = p;
                q->mnext = block;
                q = block;
        }

        if ( (unsigned)( q->mlen + (unsigned)q ) == (unsigned)p)
                q->mlen += p->mlen;
                q->mnext = p->mnext;
        }
        restore(ps);
        return(OK);
}
```