8 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 different 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**: entire processes are written to secondary (disk) storage when they are not 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.

Memory Management

- 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's 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 constitutes an especially important aspect for security.

8.1 Memory Management on the 11/2

- The whole program resides in **main memory**!
- Storage layout **when Xinu begins**:

↓ 0	↓`_etext`	↓`_edata`	↓`_end`
text	**data**	**bss**	**free space** ⟵ · · · **stack**

8.2 Dynamic Memory Requirements in Xinu

- Storage layout **during execution**:

↓ 0	↓`_etext`	↓`_edata`	↓`_end`				
text	**data**	**bss**	**heap**	**free**	**stack** # 3	**stack** # 2	**stack** # 1

8.3 Low-Level Memory Management Procedure

- System calls `getstk` and `freestk` are used by `create` and `kill`:
  - in `create`: ...`saddr = getstk(ssize)`... – see page 75
  - in `kill`: ...`freestk(pptr->pbase, pptr->pstklen)`... – see page 77
- `getstk` **obtains a block of memory** and returns its highest address, at which the stack of the created process initially starts.
- `create` records **size and location** of the allocated space in the process table entry of a process – later `kill` returns this stack space to **free memory** space by calling `freestk` which has two arguments: highest **address** of block being returned and its **size**.
- XINU guarantees that the stack space allocated to a process is **released** at process exit.

Low-Level Memory Management Procedure

- For the **heap no deallocation procedures** are built into Xinu– deallocation must be done on a process-by-process basis:

  A process must release storage that it allocates from the heap before it exit by itself (i.e., this doesn't happen automatically).

- **Allocation** on the heap is done by procedure `getmem`, **deallocation** on the heap is done by procedure `freemem`. Both system calls reference blocks by their **lowest address**.

- `freestk` is implemented using `freemem`:

```
#define freestk(p,len) freemem((unsigned)(p) - (unsigned)(roundew(len)) \
                                  + (unsigned)sizeof(int), roundew(len))
```

  This converts counting an address from the top to counting from its bottom address:

...	1001	1000	999	998	997	996	...
		↑	length 4		↑		

8.4 The Location of Allocated Storage

- **Separating** heap from the stack storage works best when a **single process** executes: if (single) **stack overflow** occurs, the process runs into the free space located between stack and heap:

heap	**free space** $\longleftarrow$	**stack**

- A problem arises when more than one process executes: **Stack overflow** that causes **corruption of data**... an exercise: **suggest a solution**!

8.5 Implementation Of Xinu Memory Management

- The free memory `freemem` is organized as follows: **Blocks of free memory** are **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 `*mnext` to the next free block** and the **size `mlen` of the current block**:

```
struct  mblock  {
        struct  mblock  *mnext;
        unsigned mlen;
        };
```

- `mnext` points to the next block or contains `NULL`, `mlen` contains the length of the current block in bytes **including the two-word-header**! So only blocks of **two words or more** can be linked into the free list, i.e., of four bytes or more, making the number of bytes a multiple of four.
- Record `memlist` has the same form as all other blocks on the free list: it's of type `struct mblock`!

Implementation Of Xinu Memory Management

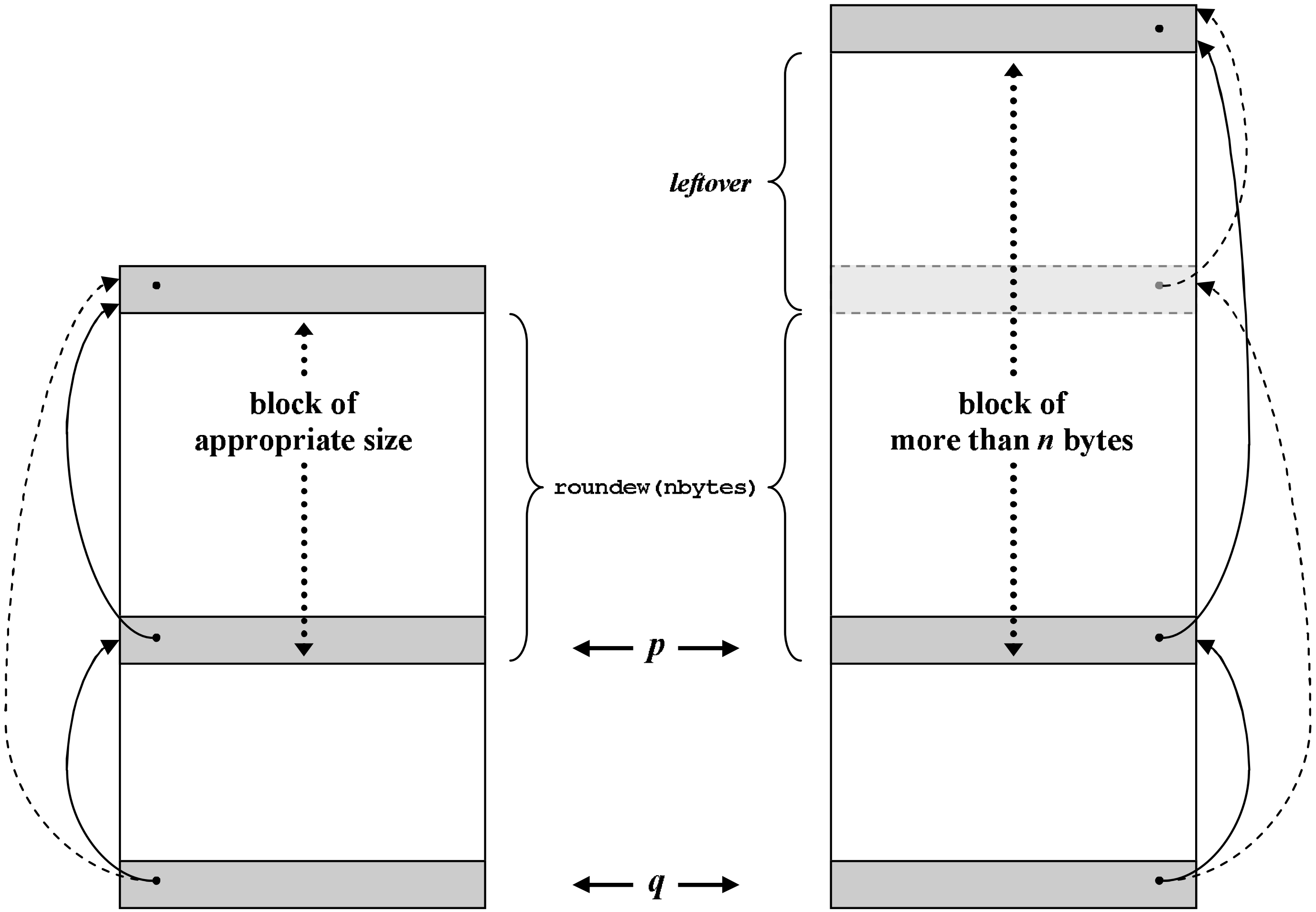
```
/* mem.h - freestk, roundew, truncew */

/*----------------------------------------------------------------------
 * 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 mlen;
        };
extern  struct  mblock  memlist;        /* head of free memory list     */
extern  int     *maxaddr;               /* max memory address           */
extern  int     etext;                  /* address beyond text segment  */
extern  int     edata;                  /* address beyond data segment  */
extern  int     end;                    /* address beyond loaded memory */
```

8.5.1 Allocating Heap Storage

- Procedure `*getmem(nbytes)` allocates heap storage returning the **lowest** integer address. `getmem` searches the memory list to find the **first block of memory large enough** to contain `roundew(nbytes)` bytes of free storage.
- Because the list of free blocks is **singly-linked**, **two pointers** p and q with q trailing behind p, are used to search for a block of appropriate size. When p points to a block of suitable size, q points to its **predecessor** on the list (possibly its head, `memlist`).

Allocating Heap Storage

- Two cases may occur while searching:



Allocating Heap Storage

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

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

/*------------------------------------------------------------------------
 * getmem  --  allocate heap storage, returning lowest integer address
 *------------------------------------------------------------------------
 */
SYSCALL *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 );
}
```

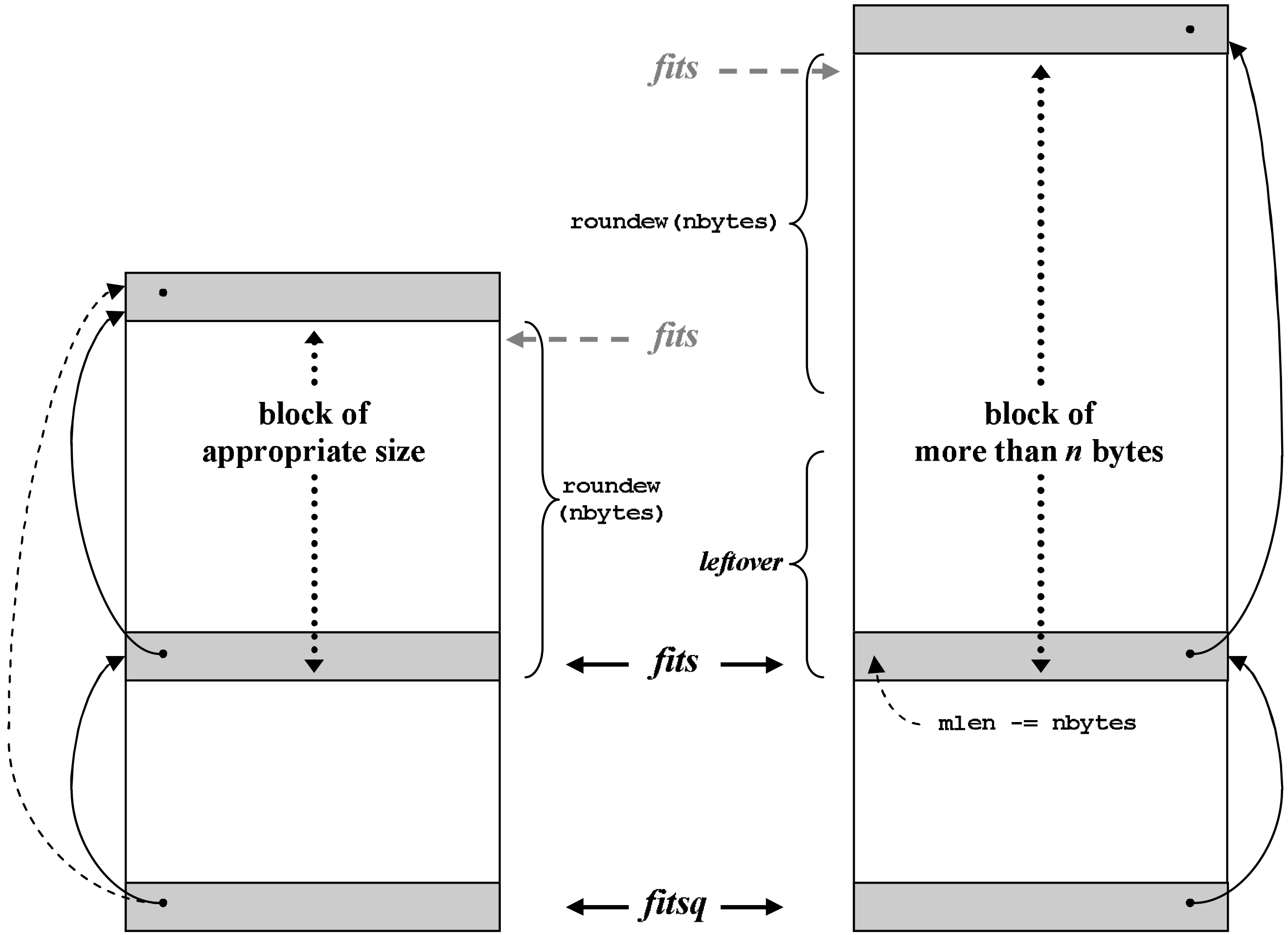
8.5.2 Allocating Stack Storage

- `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 last free block that satisfied the request, and `fitsq` points to its predecessor.
- `*getstk(nbytes)` makes full use of C's powerful `for`-statement, see page 382 for its semantics – `for(S1; exp; S2) { S3 }` corresponds in PASCAL with

```
S1; while exp <> 0 do begin S3; S2 end
```

Allocating Stack Storage

- Two cases may occur while searching:

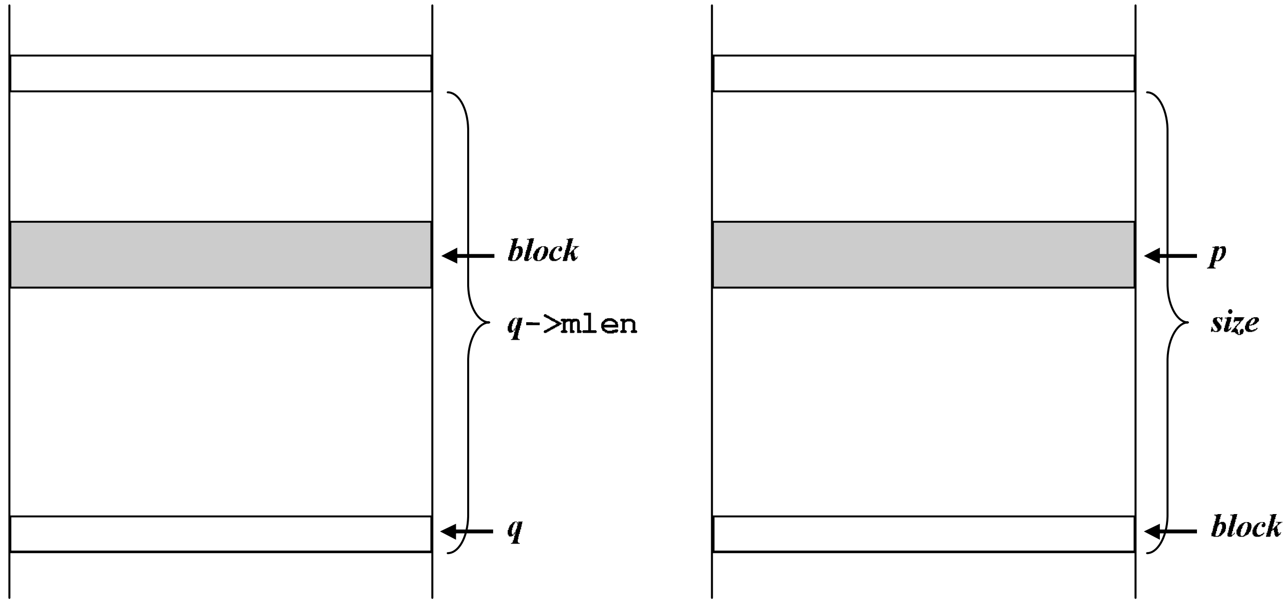


Allocating Heap Storage

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

        disable(ps);
        if (nbytes == 0) {
                restore(ps);
                return( (int *)SYSERR );
        }
        nbytes = (unsigned)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 == (len = fits->mlen) ) {
                fitsq->mnext = fits->mnext;
        } else {
                fits->mlen -= nbytes;
        }
        fits = ((unsigned)fits) + len - sizeof(int);
        restore(ps);
        return( (int *)fits );
}
```

8.5.3 Releasing Storage

- System call `freemem` returns a block of storage by inserting it at the proper location in the **free list** and **coalescing it** with any **adjacent free blocks**.
- `freemem` stops searching as soon as address of the block to be returned lies between p and q.
- Two cases lead to `SYSERR`: a block starts **within free area** or block intersects "free area":

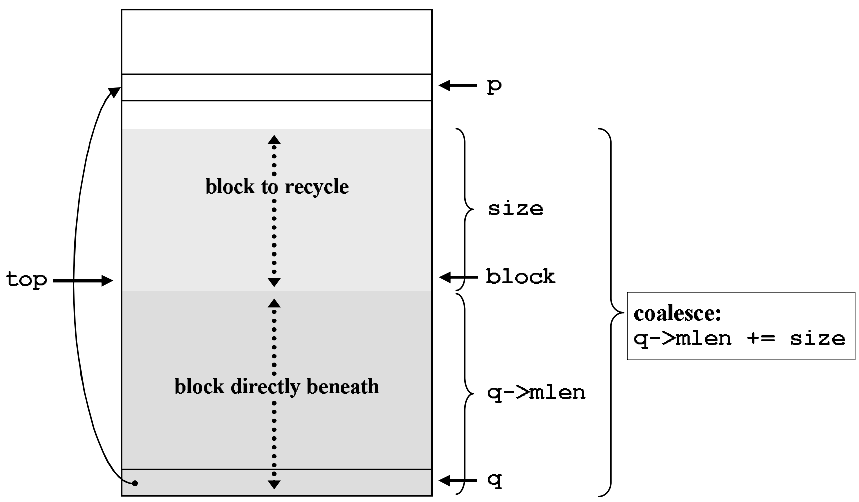


Releasing Storage

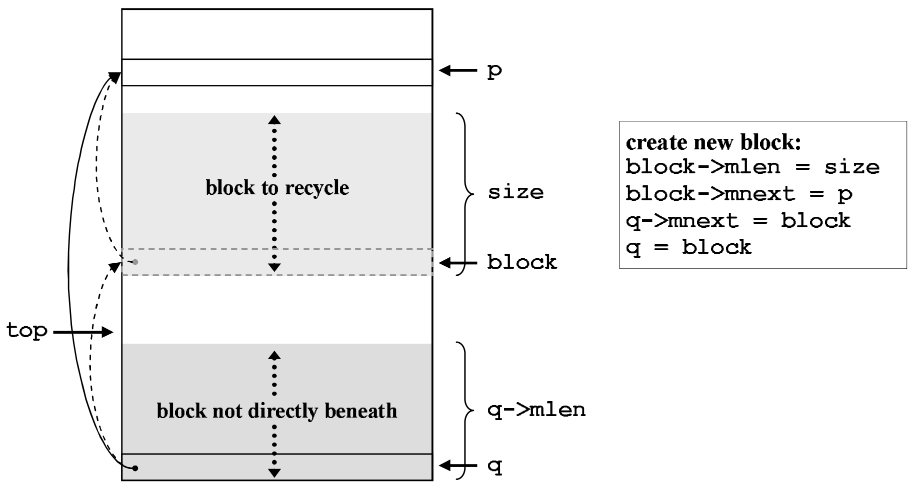
```
/*----------------------------------------------------------------------
 *  freemem  --  free a memory block, returning it to memlist
 *----------------------------------------------------------------------
 */
SYSCALL freemem(block, size)
        struct  mblock  *block;
        unsigned size;
{
        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);
}
```

Releasing Storage

1) a) if `(top==(unsigned)block)`: coalesce downwards



1) b) if `(top!=(unsigned)block)`: create new block



2) if `(q->mlen + (unsigned)q == (unsigned)p)`: coalesce upwards