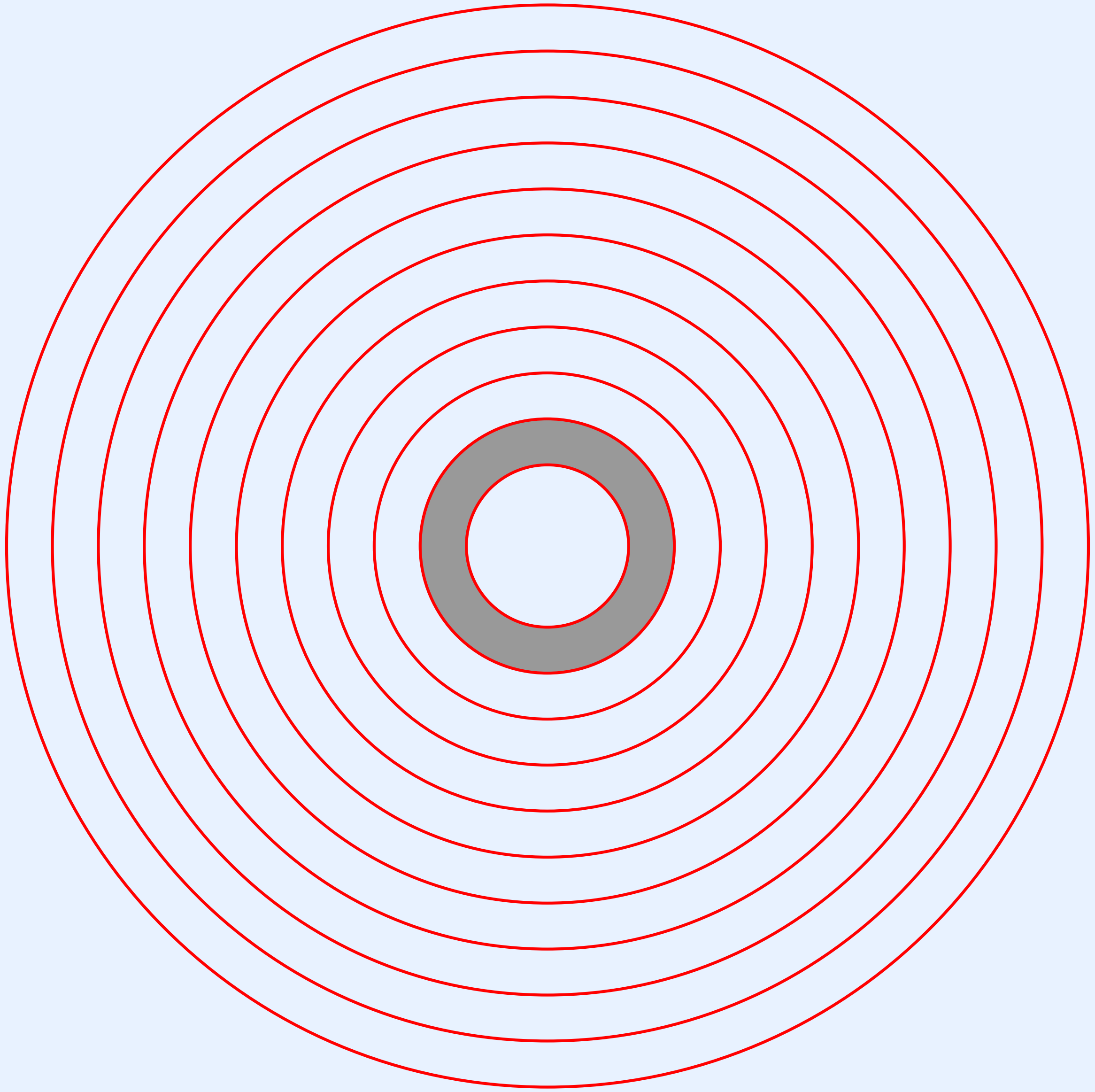
Module IX

Low-Level Memory Management

Low-Level

Memory Management

Location Of Low-Level Memory Management In The Hierarchy



The Apparent Impossibility Of A Hierarchical OS Design

- A process manager uses the memory manager to allocate space for a process
- A memory manager uses the device manager to page or swap to disk
- A device manager uses the process manager to block and restart processes when they request I/O
- Solution: divide the memory manager into two parts

The Two Types Of Memory Management

- Low-level memory manager
  - Manages memory within the kernel address space
  - Used to allocate address spaces for processes
  - Treats memory as a single, exhaustible resource
  - Positioned in the hierarchy below process manager
- High-level memory manager
  - Manages pages within a process’s virtual address space
  - Positioned in the hierarchy above the device manager
  - Divides memory into abstract resources

Conceptual Uses Of A Low-Level Memory Manager

- Allocate stack space for a process
  - Performed by the process manager when a process is created
  - The memory manager must include functions to allocate and free stacks
- Allocation of heap storage
  - Performed by the device manager (buffers) and other system facilities
  - The memory manager must include functions to allocate and free heap space

The Xinu Low-Level Memory Manager

- Two functions control allocation of stack storage

```
addr = getstk(numbytes);

freestk(addr, numbytes);
```

- Two functions control allocation of heap storage

```
addr = getmem(numbytes);

freemem(addr, numbytes);
```

- Memory is allocated until none remains
- Only *getmem*/*freemem* are intended for use by Xinu application processes; *getstk*/*freestk* are restricted to the OS

A Principle Regarding Memory Allocation

- If an operating system dynamically allocates memory to service a system call
  - The OS becomes vulnerable to malicious code
  - The system may run out of memory and malfunction
- General principle:

> **Whenever possible, avoid dynamic memory allocation inside the operating system by using static data structures to hold operating system variables; never allow an application to force the system to allocate memory.**

Well-Known Memory Allocation Strategies

- Stack and heap can be
  - Allocated from the same free area
  - Allocated from separate free areas
- The memory manager can use a single free list and follow a paradigm of
  - First-fit
  - Best-fit
  - The free list can be circular with a roving pointer
- The memory manager can maintain multiple free lists
  - By exact size (static / dynamic)
  - By range

Well-Known Memory Allocation Strategies
(continued)

- The free list can be kept in a hierarchical data structure (e.g., a tree)
  - Binary sizes of nodes can be used
  - Other sequences of sizes are also possible (e.g., Fibonacci)
- To handle repeated requests for the same size blocks, a cache can be combined with any of the above methods

Practical Considerations

- Sharing
  - A stack can never be shared
  - Multiple processes may share access to a given block allocated from the heap
- Persistence
  - A stack is associated with one process, and is freed when the process exits
  - An item allocated from a heap may persist longer than the process that created it
- Stacks tend to be one size, but heap requests vary in size
- Fragmentation can occur

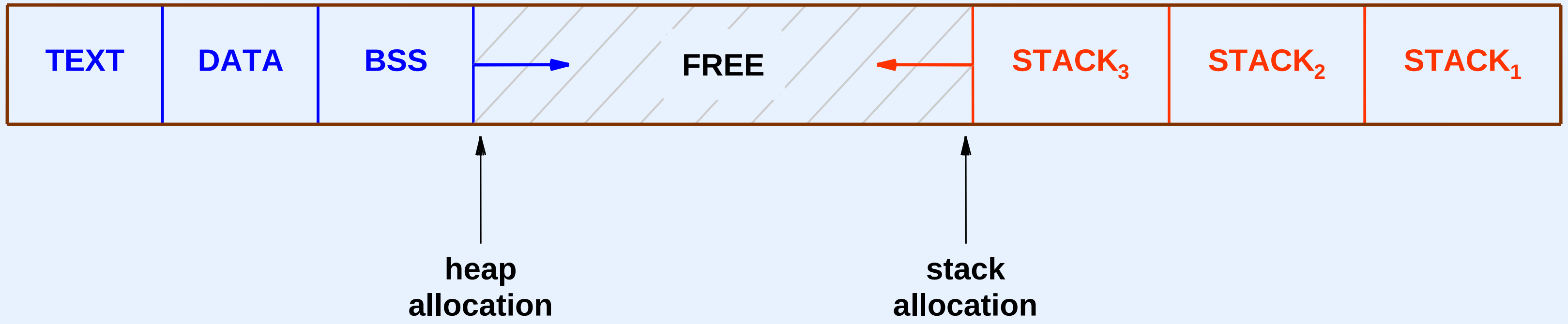
Memory Fragmentation

- Can occur if processes allocate and then free arbitrary-size blocks
- Symptom: after many requests to allocate and free blocks of memory, small blocks of allocated memory exist between blocks of free memory
- The problem: although much of the memory is free, each block on the free list is small
- Example
  - Assume a free memory consists of 1 Gigabyte total
  - A process allocates 1024 blocks of one Megabyte each (a total of 1 Gigabyte)
  - The process then frees every other block
  - Although 512 Megabytes of free memory are available, the largest free block is only 1 Megabyte

The Xinu Low-Level Allocation Scheme

- All free memory is treated as one resource
- A single free list is used for both heap and stack allocation
- The free list is
  - Ordered by increasing address
  - Singly-linked
  - Initialized at system startup to contain *all* free memory
- The Xinu allocation policies
  - Heap allocation uses the first-fit approach
  - Stack allocation uses the last-fit approach
  - The design results in two conceptual pools of memory

Consequence Of The Xinu Allocation Policy



- The first-fit policy means heap storage is allocated from lowest part of free memory
- The last-fit policy means stack storage is allocated from the highest part of free memory
- Note: because stacks tend to be uniform size, there is higher probability of reuse and lower probability of fragmentation

Protecting Against Stack Overflow

- Note that the stack for a process can grow downward into the stack for another
- Some memory management hardware supports protection
  - The memory for a process stack is assigned the process’s protection key
  - When a context switch occurs, the processor protection key is set
  - If a process overflows its stack, hardware will raise an exception
- If no hardware protection is available
  - Mark the top of each stack with a reserved value
  - Check the value when scheduling
  - The approach provides a little protection against overflow

Memory Allocation Granularity

- Facts
  - Memory is byte addressable
  - Some hardware requires alignment
    - For a process stack
    - For I / O buffers
    - For pointers
  - Free memory blocks are kept on free list
  - One cannot allocate / free an individual byte of memory efficiently
- Solution: choose a minimum granularity and round all requests to the minimum

Example Code To Round Memory Requests

```
/* excerpt from memory.h */
#define PAGE_SIZE       4096

/*----------------------------------------------------------------------
 * roundmb, truncmb - Round or truncate address to memory block size
 *----------------------------------------------------------------------
 */
#define roundmb(x)      (char *)( (7 + (uint32)(x)) & (~7) )
#define truncmb(x)      (char *)( ((uint32)(x)) & (~7) )
```

- Note the efficient implementation
  - The size of *memblk* is chosen to be a power of 2
  - The code implements rounding and truncation with bit manipulation

The Xinu Free List

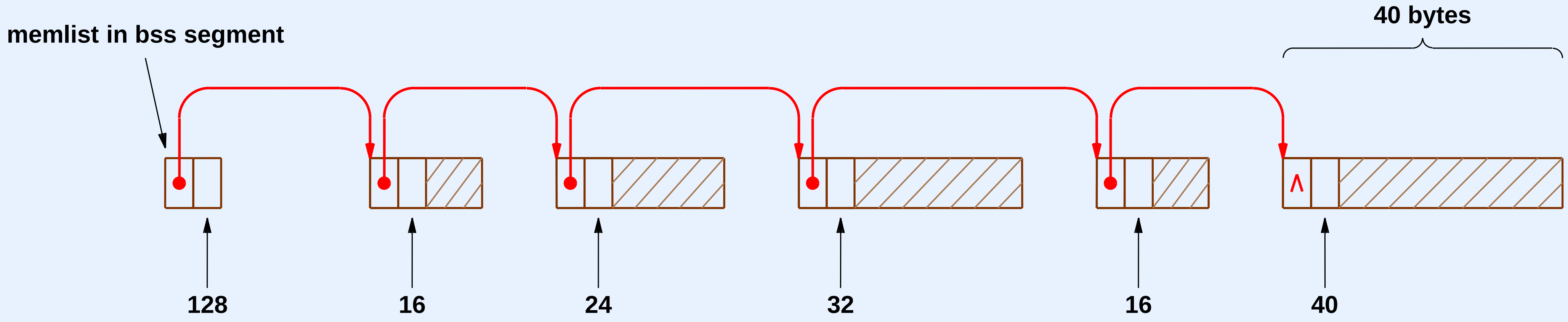
- Employs a well-known trick: to link together a list of free blocks, place all pointers *in the blocks themselves*
- Each block on the list contains
  - A pointer to the next block
  - An integer giving the size of the block
- A fixed location (variable *memlist*) contains a pointer to the first block on the list
- Look again at the definitions in memory.h

Declarations For The Free List

```
/* excerpt from memory.h */

struct  memblk  {                       /* See roundmb & truncmb        */
        struct  memblk  *mnext;         /* Ptr to next free memory blk  */
        uint32  mlength;                /* Size of blk (includes memblk)*/
        };
extern  struct  memblk  memlist;        /* Head of free memory list     */
extern  void    *minheap;               /* Start of heap                */
extern  void    *maxheap;               /* Highest valid heap address   */
```

- Struct *memblk* defines the two items stored in every block on the free list
  - A pointer to the next free block
  - The size of the current block
- Variable *memlist* is the head of the free list
- Making the head of the list have the same structure as other nodes reduces special cases in the code

Illustration Of Xinu Free List



- Free memory blocks are used to store list pointers
- Items on the list are ordered by increasing address
- All allocations rounded to size of struct *memblk*
- As the last node shows, the length includes the bytes used by the header
- The length in *memlist* counts total free memory bytes

Allocation Technique

- Round up the request to a multiple of sizeof(memblk)
- Walk the free memory list
- Choose either
  - First free block that is large enough (*getmem*)
  - Last free block that is large enough (*getstk*)
- If a free block is larger than the request, extract a piece for the request and leave the part that is left over on the free list
  - For *getmem*, allocate the lowest addresses in the block
  - For *getstk*, allocate the highest addresses in the block

When Searching The Free List

- Use two pointers that point to two successive nodes on the list
- An invariant controls the pointers during the search
  - Pointer *curr* points to a node on the free list (or *NULL*, if at the end of the list)
  - Pointer *prev* points to the previous node (or *memlist*, if at the beginning of the list)
- The invariant is established initially by making *prev* point to *memlist* and making *curr* point to the item to which *memlist* points
- The invariant must be maintained each time pointers move along the list

Xinu Getmem (Part 1)

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

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getmem  -  Allocate heap storage, returning lowest word address
 *------------------------------------------------------------------------
 */
char    *getmem(
          uint32        nbytes          /* Size of memory requested     */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  memblk  *prev, *curr, *leftover;

        mask = disable();
        if (nbytes == 0) {
                restore(mask);
                return (char *)SYSERR;
        }

        nbytes = (uint32) roundmb(nbytes);      /* Use memblk multiples */
```

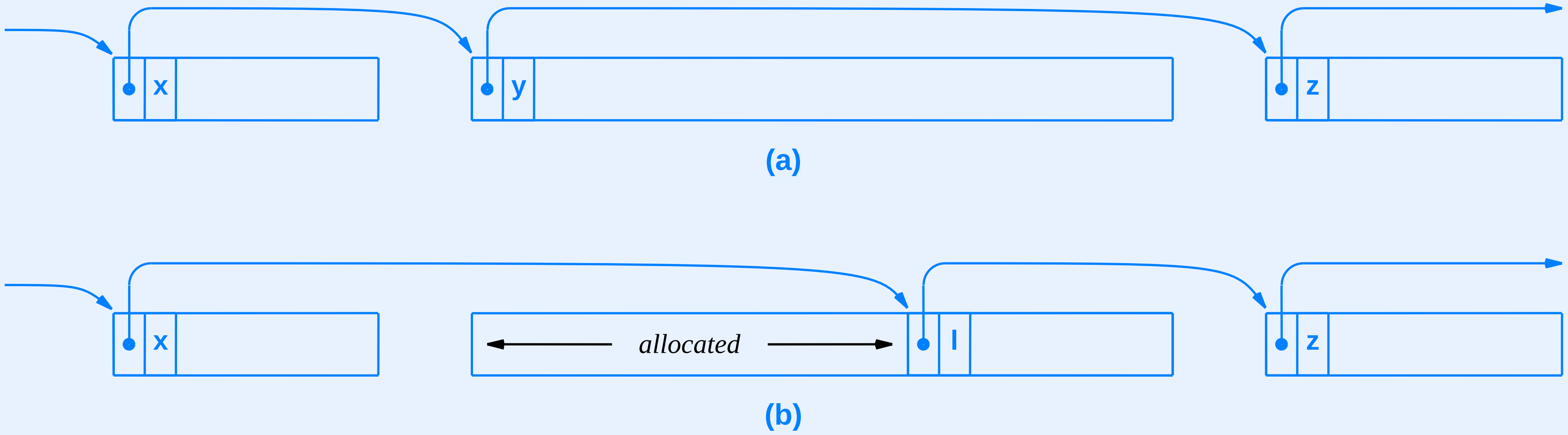
Xinu Getmem (Part 2)

```
        prev = &memlist;
        curr = memlist.mnext;
        while (curr != NULL) {                  /* Search free list     */

                if (curr->mlength == nbytes) {  /* Block is exact match */
                        prev->mnext = curr->mnext;
                        memlist.mlength -= nbytes;
                        restore(mask);
                        return (char *)(curr);

                } else if (curr->mlength > nbytes) { /* Split big block */
                        leftover = (struct memblk *)((uint32) curr +
                                        nbytes);
                        prev->mnext = leftover;
                        leftover->mnext = curr->mnext;
                        leftover->mlength = curr->mlength - nbytes;
                        memlist.mlength -= nbytes;
                        restore(mask);
                        return (char *)(curr);
                } else {                        /* Move to next block   */
                        prev = curr;
                        curr = curr->mnext;
                }
        }
        restore(mask);
        return (char *)SYSERR;
}
```

Splitting A Block

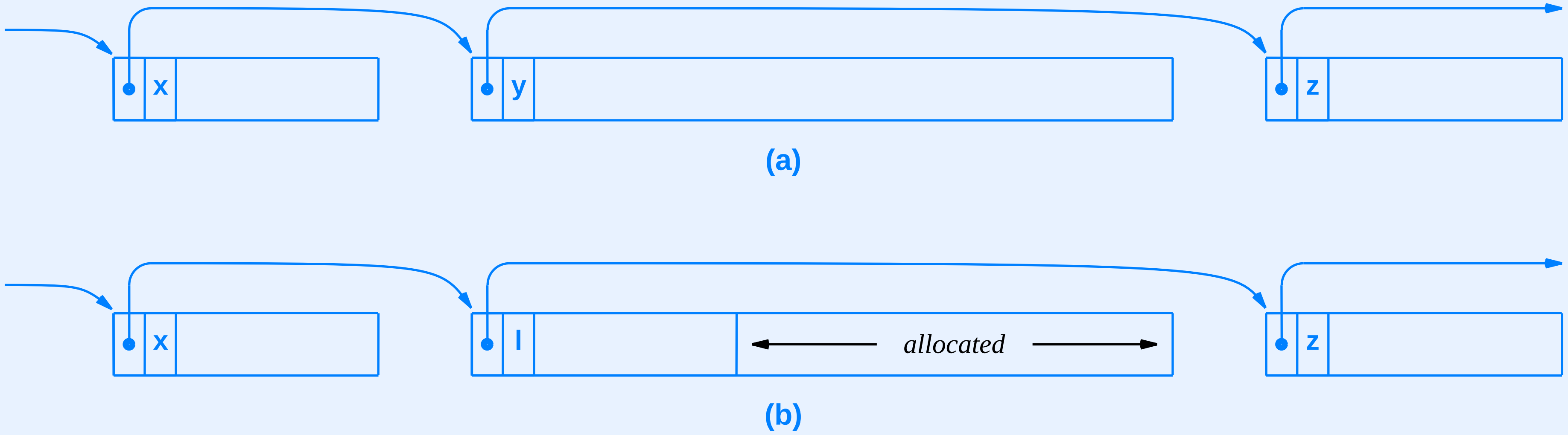
- Occurs when *getmem* chooses a block that is larger then the requested size
- *Getmem* performs three steps
  - Compute the address of the piece that will be left over (i.e., the right-hand side of the block)
  - Link the leftover piece into the free list
  - Return the original block to the caller
- Note: the address of the leftover piece is curr + nbytes (the addition must be performed using unsigned arithmetic because the high-order bit may be on)

Illustration Of How getmem Splits A Block



- Diagram (a) shows three nodes on a free list
- Diagram (b) shows the list after getmem has split the second block into an allocated piece and a piece that remains on the free list

Illustration Of How getstk Splits A Block



- Unlike getmem, getstk must allocate the highest memory addresses that satisfy a request.

- So, if it splits a block, getstk allocates the highest part of block and leaves the lower part of the block on the free list

Deallocation Technique

- Round up the specified size to a multiple of memory blocks (allows the user to specify the same value during deallocation that was used during allocation)
- Walk the free list, using *next* to point to a block on the free list, and *prev* to point to the previous block (or *memlist*)
- Stop when the address of the block being freed lies between *prev* and *next*
- Either: insert the block into the list or handle coalescing

Coalescing Blocks

- The term *coalescing* refers to the opposite of splitting
- Coalescing occurs when a block being freed is adjacent to an existing free block
- Technique: instead of adding the new block to the free list, combine the new and existing block into one larger block
- Important idea:

> *When adding a block to the free list, the memory manager must check to see whether the new block is only adjacent to the previous block, only adjacent to the next block, or adjacent to both.*

Xinu Freemem (Part 1)

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

#include <xinu.h>

/*------------------------------------------------------------------------
 *  freemem  -  Free a memory block, returning the block to the free list
 *------------------------------------------------------------------------
 */
syscall freemem(
          char          *blkaddr,       /* Pointer to memory block      */
          uint32        nbytes          /* Size of block in bytes       */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  memblk  *next, *prev, *block;
        uint32  top;

        mask = disable();
        if ((nbytes == 0) || ((uint32) blkaddr < (uint32) minheap)
                          || ((uint32) blkaddr > (uint32) maxheap)) {
                restore(mask);
                return SYSERR;
        }

        nbytes = (uint32) roundmb(nbytes);      /* Use memblk multiples */
        block = (struct memblk *)blkaddr;
```

Xinu Freemem (Part 2)

```
prev = &memlist;                        /* Walk along free list */
next = memlist.mnext;
while ((next != NULL) && (next < block)) {
        prev = next;
        next = next->mnext;
}

if (prev == &memlist) {         /* Compute top of previous block*/
        top = (uint32) NULL;
} else {
        top = (uint32) prev + prev->mlength;
}

/* Ensure new block does not overlap previous or next blocks    */

if (((prev != &memlist) && (uint32) block < top)
    || ((next != NULL)  && (uint32) block+nbytes>(uint32)next)) {
        restore(mask);
        return SYSERR;
}

memlist.mlength += nbytes;
```

Xinu Freemem (Part 3)

```
        /* Either coalesce with previous block or add to free list */

        if (top == (uint32) block) {    /* Coalesce with previous block */
                prev->mlength += nbytes;
                block = prev;
        } else {                        /* Link into list as new node   */
                block->mnext = next;
                block->mlength = nbytes;
                prev->mnext = block;
        }

        /* Coalesce with next block if adjacent */

        if (((uint32) block + block->mlength) == (uint32) next) {
                block->mlength += next->mlength;
                block->mnext = next->mnext;
        }
        restore(mask);
        return OK;
}
```

Xinu Getstk (Part 1)

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

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getstk  -  Allocate stack memory, returning highest word address
 *------------------------------------------------------------------------
 */
char    *getstk(
          uint32        nbytes          /* Size of memory requested     */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  memblk  *prev, *curr;   /* Walk through memory list     */
        struct  memblk  *fits, *fitsprev; /* Record block that fits     */

        mask = disable();
        if (nbytes == 0) {
                restore(mask);
                return (char *)SYSERR;
        }

        nbytes = (uint32) roundmb(nbytes);      /* Use mblock multiples */

        prev = &memlist;
        curr = memlist.mnext;
        fits = NULL;
```

Xinu Getstk (Part 2)

```
        fitsprev = NULL;  /* Just to avoid a compiler warning */

        while (curr != NULL) {                  /* Scan entire list     */
                if (curr->mlength >= nbytes) {  /* Record block address */
                        fits = curr;            /*   when request fits  */
                        fitsprev = prev;
                }
                prev = curr;
                curr = curr->mnext;
        }

        if (fits == NULL) {                     /* No block was found   */
                restore(mask);
                return (char *)SYSERR;
        }
        if (nbytes == fits->mlength) {          /* Block is exact match */
                fitsprev->mnext = fits->mnext;
        } else {                                /* Remove top section   */
                fits->mlength -= nbytes;
                fits = (struct memblk *)((uint32)fits + fits->mlength);
        }
        memlist.mlength -= nbytes;
        restore(mask);
        return (char *)((uint32) fits + nbytes - sizeof(uint32));
}
```

Xinu Freestk

```
/* excerpt from  memory.h */

/*----------------------------------------------------------------------
 *  freestk  --  Free stack memory allocated by getstk
 *----------------------------------------------------------------------
 */
#define freestk(p,len)  freemem((char *)((uint32)(p)            \
                                - ((uint32)roundmb(len))        \
                                + (uint32)sizeof(uint32)),      \
                                (uint32)roundmb(len) )
```

- Implemented as an inline function for efficiency
- Technique
  - Convert address from the highest address in block being freed to the lowest address in the block
  - Call *freemem* with the converted address

A Note About Function Names

Although the current implementation uses the same underlying function to release heap and stack storage, having separate system calls for freestk *and* freemem *maintains the conceptual distinction and makes the system easier to change later.*

Summary

- To preserve a multi-level hierarchy, the memory manager is divided into two pieces
  - A low-level manager is used in kernel to allocate address spaces
  - A high-level manager is used to handle abstractions of virtual memory and paging within a process's address space
- The Xinu low-level manager offers two types of allocation
  - Memory for a process stack
  - Memory from the heap
- Stack requests tend to repeat the same size

Summary
(continued)

- The Xinu low-level memory manager
  - Places all free memory on a single list
  - Rounds all requests to multiples of *struct memblk*
  - Uses first-fit allocation for heap requests and last-fit allocation for stack requests
- Process creation and termination use the memory manager to allocate and free process stacks
- *Create* handcrafts an initial stack as if the top-level function had been called; the stack includes a return address given by constant *INITRET*

Questions?