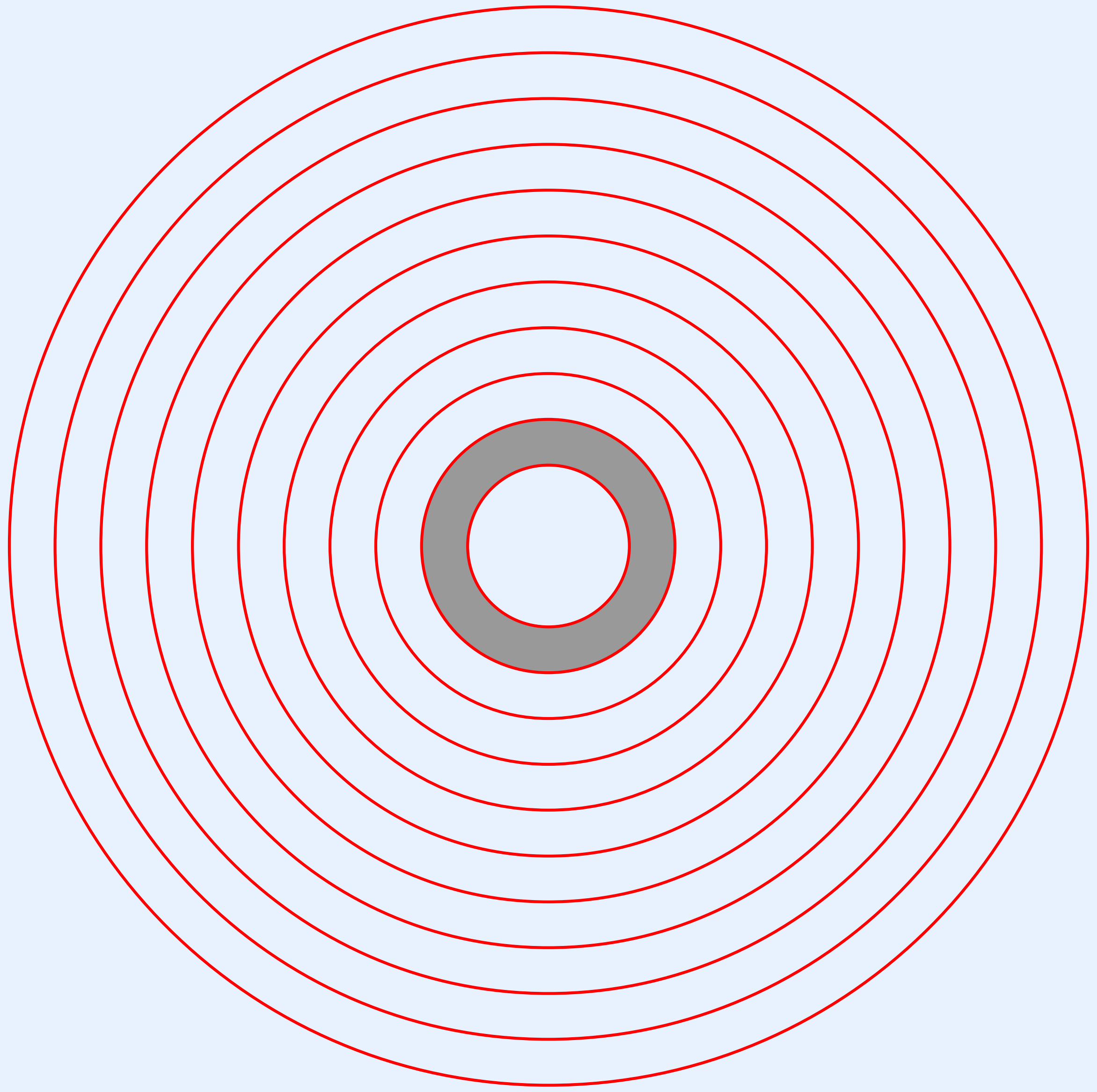
Low-Level
Memory Management

Location Of Low-Level Memory Management In The Hierarchy



The Apparent Impossibility Of A Hierarchical OS Design

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

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 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 application processes; *getstk* / *freestk* are restricted to the OS

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 exists
  - 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

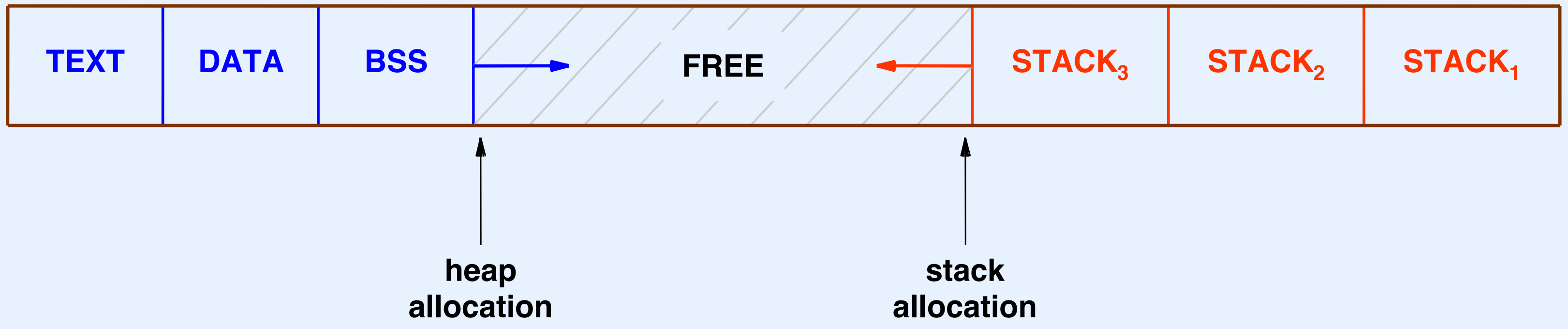
Memory Fragmentation

- 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 (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 allocates 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 process stack
    - For I / O buffers
    - For pointers
  - Free memory blocks are kept on free list
  - One cannot allocate / free individual bytes
- Solution: choose a minimum granularity and round all requests to the minimum

Example Code To Round Memory Requests

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

/*----------------------------------------------------------------------
 * 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) )

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   */
```

- 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

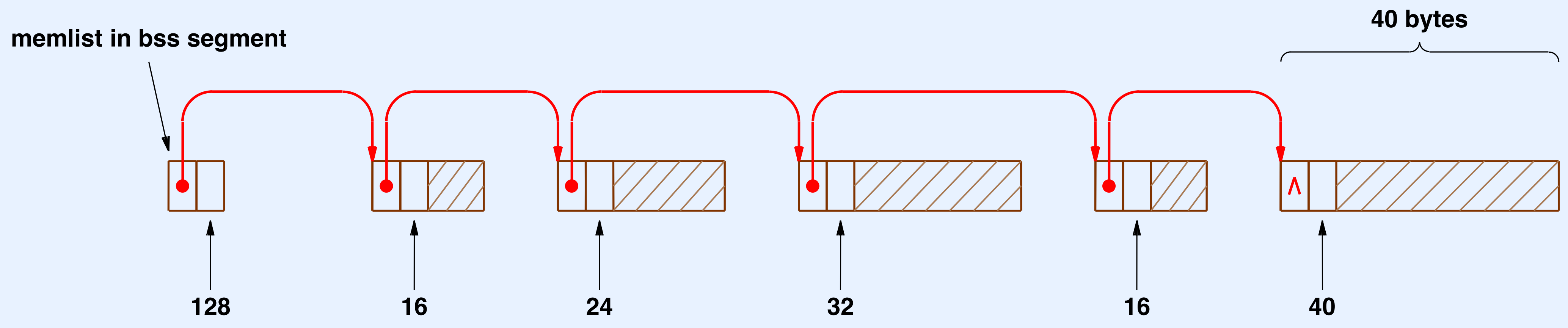
- Employs a well-known trick: to link together a list of free blocks, place all pointers *in the blocks themselves*
- Each block on the list contains
  - A pointer to the next block
  - An integer giving the size of the block
- A fixed location (*memlist* contains a pointer to the first block on the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

Declarations For The Free List

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

/*----------------------------------------------------------------------
 * 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) )

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
- 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*
- The length in *memlist* counts total free memory bytes

Allocation Technique

- Round up the request to a multiple of memory blocks
- 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

When Searching The Free List

- Use two pointers that point to two successive nodes on the list
- An invariant is used during the search
  - Pointer *curr* points to a node on the free list (or *NULL*)
  - Pointer *prev* points to the previous node (or *memlist*)
- The invariant is established initially by making *prev* point to *memblk* and making *curr* point to the item to which *memblk* 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)

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 list, combine the new and existing block into one larger block
- Note: the code must check for coalescing with the preceding block, the following block, or 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)

```
        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

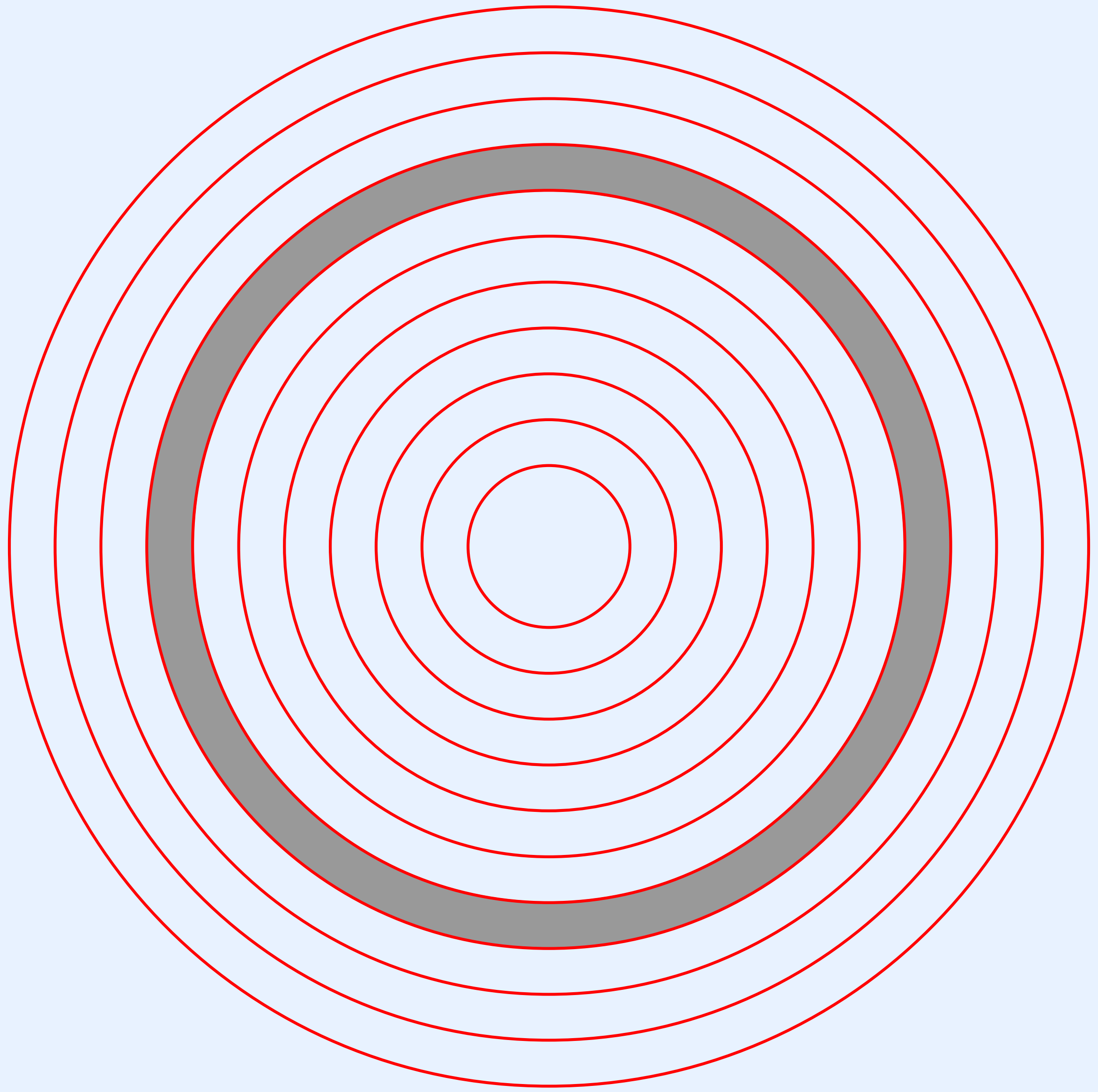
```
/* excerpt from memory.h */

/*----------------------------------------------------------------------
 *  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
- Technique: convert address from the highest address in block being freed to the lowest address in the block, and call *freemem*

Module VI

High-Level Memory Management

Location Of High-Level Memory Management In The Hierarchy



Our Approach To Memory Management (Review)

- Divide the memory manager into two pieces
- Low-level piece
  - A basic facility
  - Provides functions for stack and heap allocation
  - Treats memory as exhaustible resource
- High-level piece
  - Accommodates other memory uses
  - Assumes both operating system modules and sets of applications need dynamic memory allocation
  - Prevents exhaustion

Motivation For Memory Partitioning

- Competition exists for kernel memory
- Many subsystems in the operaing system
  - Allocate blocks of memory
  - Have needs that change dynamically
- Examples
  - The disk subsystem allocates buffers for disk blocks
  - The network subsystem allocates packet buffers
- Interaction among subsystems can be subtle and complex

Managing Memory Demands

- Overall goals can conflict
  - Protect information
  - Share information
- Extremes
  - Xinu has much sharing and almost no protection
  - The original Unix™ had much protection and almost no sharing

The Concept Of Subsystem Isolation

- An OS designer desires
  - Predictable behavior
  - Provable assertions (e.g., “network traffic will never deprive the disk driver of buffers”)
- The reality
  - Subsystems are designed independently; there is no global policy or guarantee about their memory use
  - If one subsystem allocates memory excessively, others can be deprived
- Conclusions
  - We must not treat memory as a single, global resource
  - We need a way to isolate subsystems from one another

Providing Abstract Memory Resources

Assertion: to be able to make guarantees about subsystem behavior, one must partition memory into abstract resources with each resource dedicated to one subsystem.

A Few Examples Of Abstract Resources

- Disk buffers
- Network buffers
- Message buffers
- Inter-process communication buffers (e.g., Unix pipes)
- Note that
  - Each subsystem should operate safely and independently
  - An operating system designer may choose to define finer granularity separations
    - A separate set of buffers for each network interface (Wi-Fi and Ethernet)
    - A separate set of buffers for each disk

The Xinu High-Level Memory Manager

- Partitions memory into groups of *buffer pools*
- Each pool is created once and persists until the system shuts down
- At pool creation, the caller specifies the
  - Size of buffers in the pool
  - Number of buffers in the pool
- Once a pool has been created, buffer allocation and release is dynamic
- The system provides a completely synchronous interface

Xinu Buffer Pool Functions

poolinit	–	Initialize the entire mechanism
mkbufpool	–	Create a pool
getbuf	–	Allocate buffer from a pool
freebuf	–	Return buffer to a pool

- Memory for a pool is allocated by *mkbufpool* when the pool is formed

Although the buffer pool system allows callers to allocate a buffer from a pool and later release the buffer back to the pool, the pool itself cannot be deallocated, which means that the memory occupied by the pool can never be released.

Traditional Approach To Identifying A Buffer

- Most systems use the address of lowest byte in the buffer as the buffer address
- Doing so means
  - Each buffer is guaranteed to have a unique ID
  - A buffer can be identified by a single pointer
- The scheme
  - Works well in C
  - Is convenient for programmers

Consequences Of Using A Single Pointer As An ID

- Consider function *freebuf*
  - It must return a buffer to the correct pool
  - It takes the buffer identifier as argument
- Information about buffer pools must be kept in a table
- Given a buffer, *freebuf* needs to find the pool from which the buffer was allocated

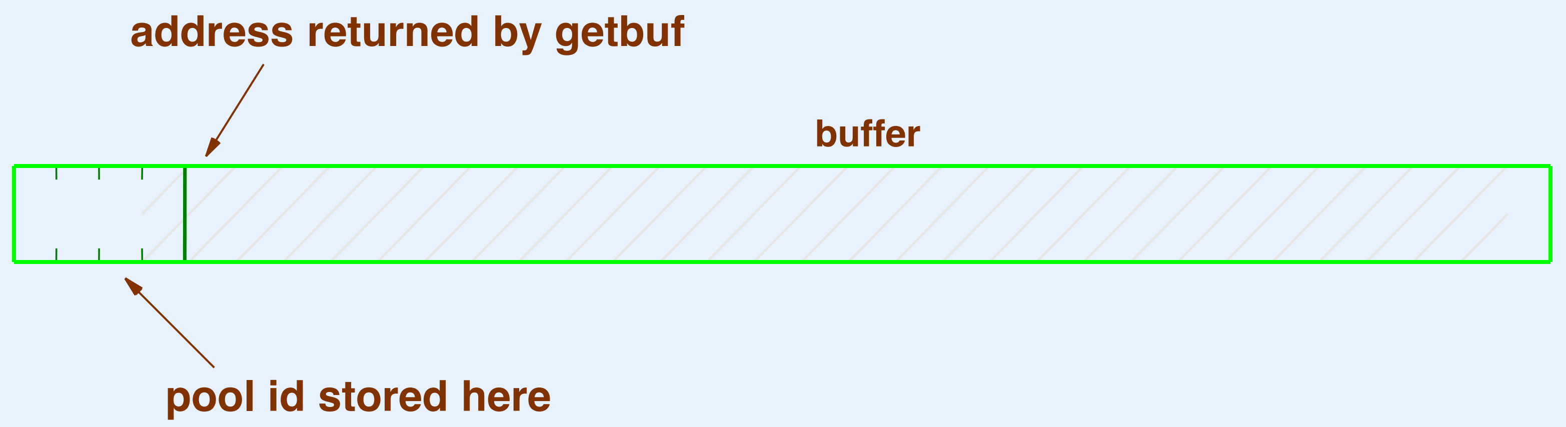
Finding The Pool To Which A Buffer Belongs

- Possibilities
  - Search the table of buffer pools to find the correct pool
  - Use an external data structure to map a buffer address to the correct pool (e.g., keep a list of allocated buffers and the pool to which each belongs)
- An alternative
  - Have *getbuf* pass the caller two values: a pool ID and a buffer address
  - Have *freebuf* take two arguments: a pool ID and a buffer address
- Unfortunately, using two arguments is inconvenient for programmers

Solving The Single Pointer Problem

- Xinu uses a clever trick to avoid passing two values
  - Use the address of the lowest usable byte as a buffer identifier
  - Store a pool ID along with each buffer, but hide it from the user
- The implementation
  - When allocating a buffer, allocate enough extra bytes to hold the pool ID
  - Store the pool ID in the extra bytes
  - Place the extra bytes *before* the buffer
  - Return a pointer to the buffer, not the extra bytes
- A process can a use buffer without knowing that the extra bytes exist

Illustration Of A Pool ID Stored With A Buffer



- Xinu allocates four bytes more than the user specifies
- Conceptually, the additional bytes precede the buffer, and are used to store the ID of the buffer pool
- *Getbuf* returns a single pointer to the data area of the buffer (beyond the extra bytes)
- *Freebuf* expects the same pointer that *getbuf* returns to a caller
- The pool ID is transparent to applications using the buffer pool

Potential Downsides Of The Xinu Scheme

- Some device hardware requires a buffer to start on a page boundary, but adding four bytes to the size may ruin alignment
- If the pool id is accidentally overwritten, the buffer will either be returned to the wrong pool or an error will occur because the pool ID is invalid

Buffer Pool Operations

- Create a pool (*mkpool*)
  - Compute the size of a buffer with extra bytes added to hold a pool ID
  - Use *getmem* to allocate memory for all the buffers that will be in the pool
  - Form a singly-linked list of the buffers (storing links in the buffers themselves)
  - Allocate a semaphore to count buffers (*getbuf*)
- Allocate a buffer from a pool
  - Use the pool ID argument, and locate the correct buffer pool
  - *Wait* on the semaphore associated with a pool, blocking until a buffer becomes available
  - Extract the buffer at the head of the list and return it to the caller

Buffer Pool Operations
(continued)

- Free (deallocate) a previously-allocated buffer (*freebuf*)
  - Extract the pool ID from the extra bytes that precede the buffer
  - Use the pool ID to locate the buffer pool
  - Insert the buffer at the head of the list for the pool
  - Signal the semaphore associated with the pool

Virtual Memory

Definition Of Virtual Memory

- An abstraction of physical memory
- It provides separation from underlying hardware
- Primarily used with applications (user processes)
- Provides an application with an address space that is independent of
  - Physical memory size
  - A position in physical memory
- Many mechanisms have been proposed and used

General Approach

- Typically used with a heavyweight process
  - The process appears to run in an isolated address space
  - All addresses are *virtual*, meaning that each process has an address space that starts at address zero
- The operating system
  - Establishes policies for memory use
  - Creates a separate virtual address space for each process
  - Configures the hardware as needed
- The underlying hardware
  - Dynamically translates from virtual addresses to physical addresses
  - Provides support to help the operating system make policy decisions

A Virtual Address Space

- Can be smaller than the physical memory
  - * Example: a 32-bit computer with more than 2^{32} bytes (four GB) of physical memory
- Can be larger than the physical memory
  - * Example: a 64-bit computer with less than 2^{64} bytes (16 million terabytes) of memory
- Historic note: on early computers, physical memory was larger. Then, virtual memory was larger until physical memory caught up. Now, 64-bit architectures mean virtual memory is once again larger than physical memory.

Multiplexing Virtual Address Spaces Onto Physical Memory

- General idea
  - Store a complete copy of each process's address space on secondary storage
  - Move pieces of the address space to main memory as needed
  - Write pieces back to disk to create space in memory for other pieces
- Questions
  - How much of a process's address space should reside in memory?
  - When should a particular piece be loaded into memory?
  - When should a piece be written back to disk?

Approaches That Have Been Used

- *Swapping*
  - Transfer an entire address space (complete process image)
- *Segmentation*
  - Divide the image into large segments
  - Transfer segments as needed
- *Paging*
  - Divide image into small, fixed-size pieces
  - Transfer individual pieces as needed

Approaches That Have Been Used
(continued)

- *Segmentation with paging*
  - Divide an image into large segments
  - Further subdivide segments into fixed-size pages

A Widely-Used Approach

- Paging has emerged as the most widely used approach for virtual memory
- The reasons are
  - Choosing a reasonable page size (e.g., 4K bytes) reduces page faults for most applications
  - Efficient hardware is possible

Choosing a page size that is a power of two makes it possible to build extremely efficient address mapping hardware.

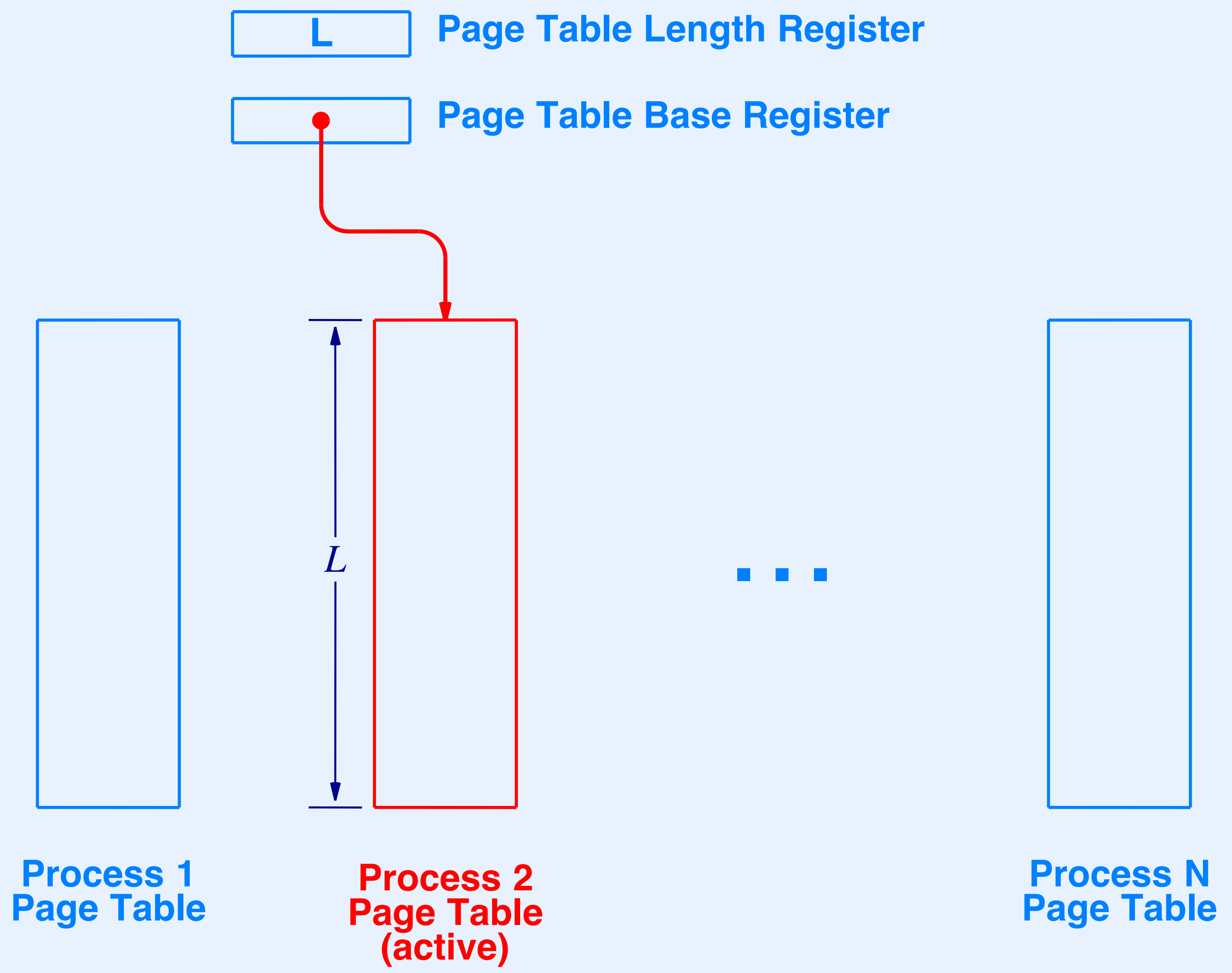
Hardware Support For Paging

- Page tables
  - The operating system allocates one page table per process
  - The location at which a page table is stored depends on the hardware
    - Kernel memory (typical)
    - *Memory Management Unit (MMU)* hardware (on some systems)
- A page table base register
  - Internal to the processor
  - Specifies the location of the page table currently being used (i.e., the page table for the current process)
  - Must be changed during a context switch

Hardware Support For Paging (continued)

- A page table length register
  - Internal to the processor
  - Specifies the number of entries in the current page table
  - Can be changed during context switch if the size of the virtual address space differs among processes
  - Can be used to limit the size of a process's virtual address space

Illustration Of VM Hardware Registers



- Only one page table is active at a given time (the page table for the current process)

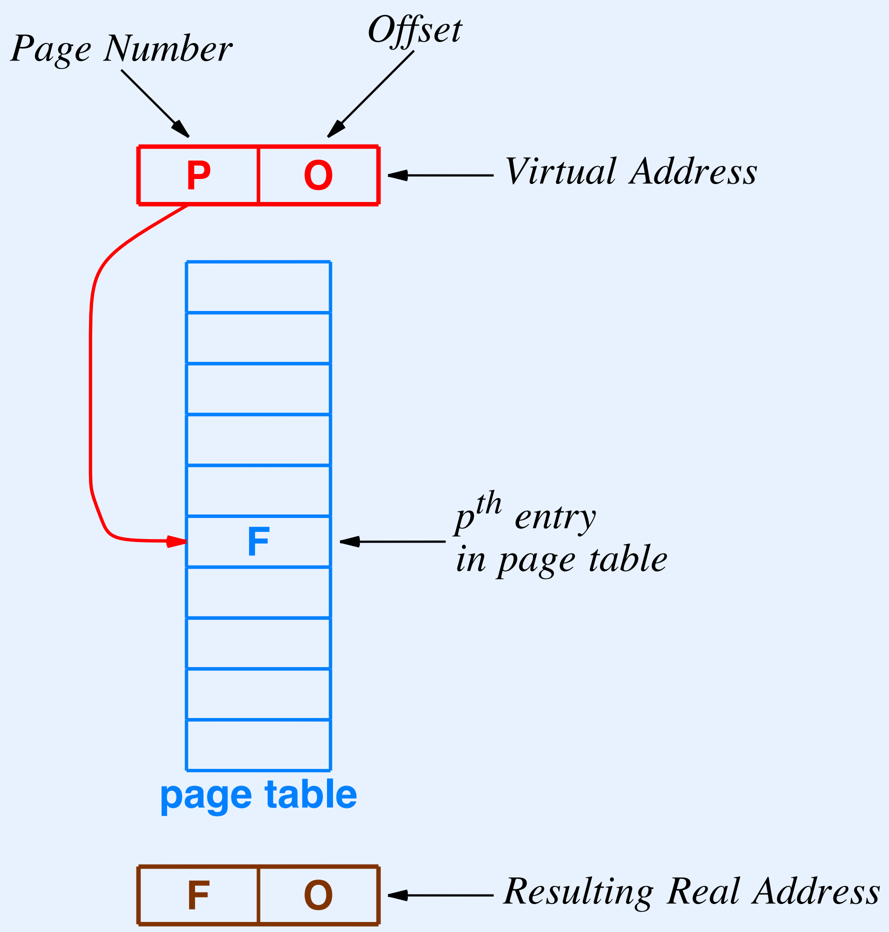
Address Translation

- A key part of virtual memory
- Refers to the translation from the virtual address a process uses to the corresponding physical memory address
- Is performed by memory management hardware
- Must occur on *every* memory reference
- A hardware unit performs the translation

Address Translation With Paging

- For now, we will assume
  - The operating system is not paged
  - The physical memory area beyond the operating system kernel is used for paging
  - Each page is 4 Kbytes (typical of current virtual memory hardware)
- Think of the physical memory area used for paging as a giant array of *frames*, where each frame can hold one page (i.e., a frame is 4K bytes)

Illustration Of Address Translation



- Each page table entry contains a physical frame address
- Choosing the page size to be a power of 2 eliminates division and modulus

In Practice

- The size of virtual space may be limited to physical memory size
- Some hardware offers separate page tables for text, data, and stack segments
  - The chief disadvantage: extra complexity
  - The advantage: the three can operate independently
- The kernel address space can also be virtual (but it hasn't worked well in practice)

Page Table Sizes And 32 and 64 Bit Computers

- For a 32-bit address space where each page is 4 Kbytes
  - There are 2^{20} page table entries of 4 bytes per entry
  - The total page table size for one process: 4 Mbytes
- For a 64-bit address space where each page is 4 Kbytes
  - There are 2^{52} page table entries of 4 bytes per entry
  - The total page table size for one process: 16,777,216 Gbytes!
- Conclusion: we cannot have complete page tables for a 64-bit address space

Paging In A 64-Bit System

- To reduce page table size, use multiple levels of page tables
  - The high-order bits of an address form an index into the top-level page table
  - The next bits form an index into the second-level page table (but only a few second-level page tables are defined)
- Key idea: only the lowest and highest pieces of the address space need to be mapped (heap and stack)
- The same technique can be applied to 32-bit address spaces to reduce page table size

The Concept Of Demand Paging

- Keep the entire memory image of each process on secondary storage
- Treat main memory as cache of recently-referenced pages
- Allocate space in memory for new pages dynamically (when needed)
- Copy a page from the secondary store to main memory on demand (when referenced)
- To make space for newly-referenced pages, move pages from the memory cache that have been changed but are not being referenced back to their place on secondary storage

The Importance Of Hardware Support For Virtual Memory

- Every memory reference must be translated from a virtual address to a physical address, including
  - The address of an instruction as well as data
  - Branch addresses computed as a *jump* instruction executes
  - Indirect addresses that are generated at runtime
- Hardware support is essential
  - For efficiency
  - For recovery if a fault occurs
  - To record which pages are being used

In Practice

- A single instruction may reference many pages!
  - To fetch the instruction
  - To fetch each operand
  - To follow indirect references
  - To store results
- On CISC hardware, a memory copy instruction can reference *multiple* pages
- The point: hardware support is needed to make references efficient

Hardware Support For Address Mapping

- Special-purpose hardware speeds page lookup and makes paging practical
- The hardware
  - Is called a *Translation Look-aside Buffer* (*TLB*)
  - Is implemented with T-CAM
- A TLB caches most recent address translations
- Good news: many applications tend to make repeated references to the same page (i.e., a high locality of reference), so a TLB works well
- Some VM mappings must change during a context switch (e.g., multiple processes may each have a virtual address 0, but the mapping differs)
- The point: the mappings in the TLB will not be valid for the new process

Managing The TLB

- When the operating system switches context
  - On some hardware, the operating system must flush the TLB to remove old entries
  - On other hardware, tags are used to distinguish among address spaces
- Using tags
  - A tag is assigned to each page by the OS (typically, the process ID)
  - The operating system tells the VM hardware the tag to use (i.e., the current process)
  - When placing a mapping in the TLB, the hardware appends the current tag to the address
  - When searching the TLB, the hardware appends the tag to the address
  - Advantage: the OS does not need to flush the TLB during a context switch

Can Page Tables Be Paged?

- On some hardware, yes
- The tables must be stored in memory
- The current page table must be locked into memory
- Paging page tables is so extremely inefficient that is it impractical

Bits That Record Page Status

- Each page table entry contains status bits that are understood by the hardware
- The *Use Bit*
  - Set by the hardware whenever the page is referenced
  - Applies to both *fetch* and *store* operations
- The *Modify Bit*
  - Set by the hardware when a *store* operation occurs
- The *Presence Bit*
  - Set by the operating system, to indicate that it has placed the page in memory (we say the page is *resident*)
  - Tested by the hardware when the page is referenced

Page Replacement

- The hardware
  - Generates a page fault when a referenced page is not resident
  - Raises an exception to inform the operating system
- The operating system
  - Receives the exception
  - Allocates a frame in physical memory
  - Retrieves the needed page (allowing other processes to execute while page is being fetched)
  - Once the page arrives, marks the page table entry to indicate the page is resident, and restarts the process that caused the page fault

Researchers Have Studied Many Aspects Of Paging

- Which replacement policies are most effective?
- Which pages from a given address spaces should be in memory at any time?
- Should some pages be locked in memory? (If so, which ones?)
- How does a VM policy interact with other policies (e.g., scheduling?)
- Should high-priority processes/threads have guarantees about the number of resident pages?
- If a system supports libraries that are shared among many processes, which paging policy applies to a library?

A Critical Trade-off For Demand Paging

- Paging overhead and latency for a given process can be reduced by giving the process more physical memory (more frames)
- Processor utilization and overall throughput are increased by increasing level of multitasking (concurrent processes)
- Extremes
  - Paging is minimized when the current process has maximal memory
  - Throughput is maximized when all ready processes are resident
- Researchers considered the question, “What is the best tradeoff?”

Frame Allocation

- When a page fault occurs, the operating system must obtain a frame to hold the page
- If at least one frame is unused, the selection is trivial — select an unused frame
- If all frames are currently occupied by pages from various processes, the operating system must
  - Select one of the resident pages and save a copy on disk
  - Mark the page table entry to indicate that the page is no longer resident
  - Select the frame that has been vacated
  - Obtain the page that caused the page fault, and fill in the appropriate page table entry
- Question: which frame should be selected when all are in use?

Choosing A Frame

- Researchers have studied
  - Global competition: consider frames from all processes when choosing a frame
  - Local competition: choose one of the frames of the process that caused the page fault
- They have also studied various policies
  - *Least Recently Used* (*LRU*)
  - *Least Frequently Used* (*LFU*)
  - *First In First Out* (*FIFO*)
- In the end, a basic approach has been adopted: *global clock*

The Global Clock Algorithm

- Originated in the MULTICS operating system
- Allows all processes to compete with one another (hence the term *global*)
- Has relatively low overhead
- Has become the most popular practical method

Global Clock Paradigm

- The clock algorithm is activated when a page fault occurs
- It searches through all frames in memory, and selects a frame to use
- The term *clock* is used because the algorithm starts searching where it left off the last time
- A frame containing a referenced page is given a “second chance” before being reclaimed
- A frame containing a modified page is given a “third chance” before being reclaimed
- In the worst case: the clock sweeps through all frames twice before reclaiming one
- Advantage: the algorithm does *not* require any external data structure other than the standard page table bits

Operation Of The Global Clock

- The clock uses a global pointer that picks up where it left off previously
  - It sweeps through all frames in memory
  - It only starts moving when a frame is needed
  - It stops moving once a frame has been selected
- During the sweep, the algorithm checks *Use* and *Modify* bits of each frame
- It reclaims the frame if the *Use* / *Modify* bits are *(0,0)*
- It changes *(1,0)* into *(0,0)* and bypasses the frame
- It changes *(1,1)* into *(1,0)* and bypasses the frame
- The algorithm keeps a copy of the actual modified bit to know whether a page is dirty

In Practice

- A global clock is usually configured to reclaim a small set of frames when one is needed
- The reclaimed frames are cached for subsequent references
- Advantage: collecting multiple frames means the clock will run less frequently

A Problem With Paging: Thrashing

- Imagine a large set of processes each referencing their pages at random
- At first, free frames in memory can be used to hold pages
- Eventually, the frames in memory fill up, and each new reference causes a page fault, which results in
  - Choosing a frame (the clock algorithm runs)
  - Writing the existing page to secondary storage (disk I/O)
  - Fetching a new page from secondary storage (more disk I/O)
- The processor spends most of the time paging and waiting for I/O, so little computation can be performed
- We use the term *thrashing* to describe the situation
- Having a large memory on a computer helps avoid thrashing

The Importance/Unimportance Of Paging Algorithms

- Facts
  - At one time, page replacement algorithms were the primary research topic in operating systems
  - Sophisticated mathematical analysis was done to understand their behavior
  - By the 1990s, interest in page replacement algorithms faded
  - Now, almost no one uses complex replacement algorithms

The Importance/Unimportance Of Paging Algorithms

- Facts
  - At one time, page replacement algorithms were the primary research topic in operating systems
  - Sophisticated mathematical analysis was done to understand their behavior
  - By the 1990s, interest in page replacement algorithms faded
  - Now, almost no one uses complex replacement algorithms
- Why did the topic fade?
- Was the problem completely solved?

The Importance/Unimportance Of Paging Algorithms

- Facts
  - At one time, page replacement algorithms were the primary research topic in operating systems
  - Sophisticated mathematical analysis was done to understand their behavior
  - By the 1990s, interest in page replacement algorithms faded
  - Now, almost no one uses complex replacement algorithms
- Why did the topic fade?
- Was the problem completely solved?
- Answer: physical memories became so large that very few systems need to replace pages
- A computer scientist once quipped that paging only works if systems don't page

Summary

- We considered two forms of high-level memory management
- Inside the kernel
  - Define a set of abstract resources
  - Firewalling memory used by each subsystem prevents interference
  - The mechanism uses buffer pools
  - A buffer is referenced by single address
- Outside the kernel
  - Swapping, segmentation, and paging have been used

Summary
(continued)

- Demand paging is the most popular VM technology
  - It uses fixed size pages (typically 4K bytes)
  - A page is brought into memory when referenced
- The global clock algorithm is widely used for page replacement