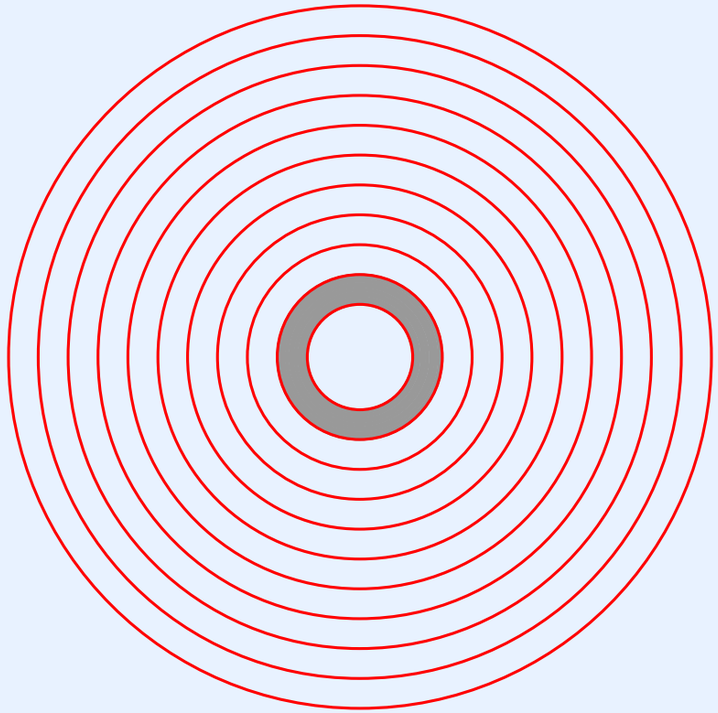
Module VII

Low-Level Memory Management

Location Low-Level Of Memory Management In The Hierarchy



Apparent Impossibility Of A Hierarchical OS Design

Apparent Impossibility Of A Hierarchical OS Design

- Process manager uses memory manager to allocate space for a process

Apparent Impossibility Of A Hierarchical OS Design

- Process manager uses memory manager to allocate space for a process
- Memory manager uses device manager to page or swap to disk

Apparent Impossibility Of A Hierarchical OS Design

- Process manager uses memory manager to allocate space for a process
- Memory manager uses device manager to page or swap to disk
- Device manager uses process manager to block and restart processes when they request I/O

Apparent Impossibility Of A Hierarchical OS Design

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

Two Types Of Memory Management

- Low-level memory manager
  - Manages memory within the kernel address space
  - Allocates 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 address space
  - Positioned in the hierarchy above device manager
  - Divides memory into abstract resources

Conceptual Uses Of A Low-Level Memory Manager

- Allocation of stack space for a process
  - Performed by the process manager
  - Primitives needed to allocate and free stack space
- Allocation of heap storage
  - Performed by device manager (buffers) and other system facilities
  - Primitives needed to allocate and free heap space

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

Possible Allocation Strategies

- Stack and heap
  - Allocated from same free area
  - Allocated from separate free areas
- Single free list within an area
  - First-fit
  - Best-fit
  - Circular list with roving pointer
- Multiple free lists within an area
  - By exact size (static / dynamic)
  - By range

Possible Allocation Strategies
(continued)

- Hierarchical data structure (tree)
  - Binary size allocation
  - Other
- FIFO cache with above methods

Practical Considerations

- Sharing
  - Stack cannot be shared
  - Multiple processes may share access to a heap
- Persistence
  - Stack object is associated with one process
  - Heap object may persist longer than the process that created it
- Stack objects tend to be uniform size

Xinu Low-Level Allocation

- One free area
- Single free list used for both heap and stack allocation
  - Ordered by increasing address
  - Singly-linked
  - Initialized at system startup to contain *all* free memory
- Allocation policy
  - Heap: first-fit
  - Stack: last-fit
  - Results in two conceptual pools

Result Of Xinu Allocation Policy

- First-fit allocates heap from lowest free address
- Last-fit allocates stack from highest free memory
- Uniform stack object size means higher probability of reuse and lower probability of fragmentation

Memory Protection And Stack Overflow

- If memory management hardware supports protection
  - Assign a unique protection key to each process stack
  - Set the processor protection key during context switch
  - Hardware will raise an exception if a process attempts to access another process's stack
- If no hardware protection is available
  - Mark the top of each stack with a reserved value
  - Check the value when scheduling
  - Provides partial protection against overflow

Memory Allocation Granularity

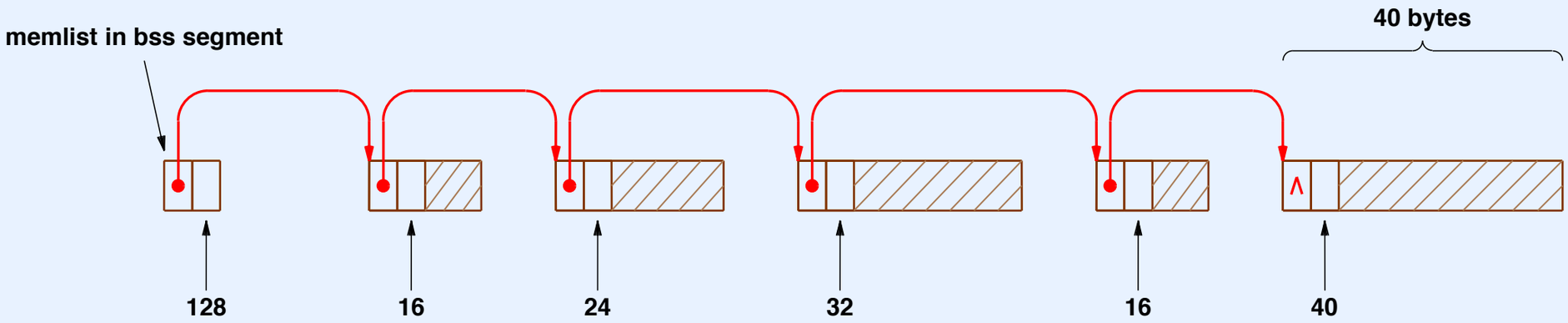
- Facts
  - Memory is byte addressable
  - Some hardware requires alignment
    * For process stack
    * For I / O buffers
    * For pointers
  - Free memory blocks are kept on free list
  - One cannot allocate / free individual bytes
- Solution: choose a minimum granularity and round all requests

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
  - Bit manipulation is used for rounding and truncation

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
- Invariant used during search:
  - *curr* points to a node on the free list (or *NULL*)
  - *prev* points to the previous node (or *memlist*)

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;
}
```

Deallocation Technique

- Round size to a multiple of memory blocks
- 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
  - Coalesce with the previous block if new block is contiguous
  - Add the new block to the free list
- Coalesce with the next block, if the result of the above is adjacent with the next block

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

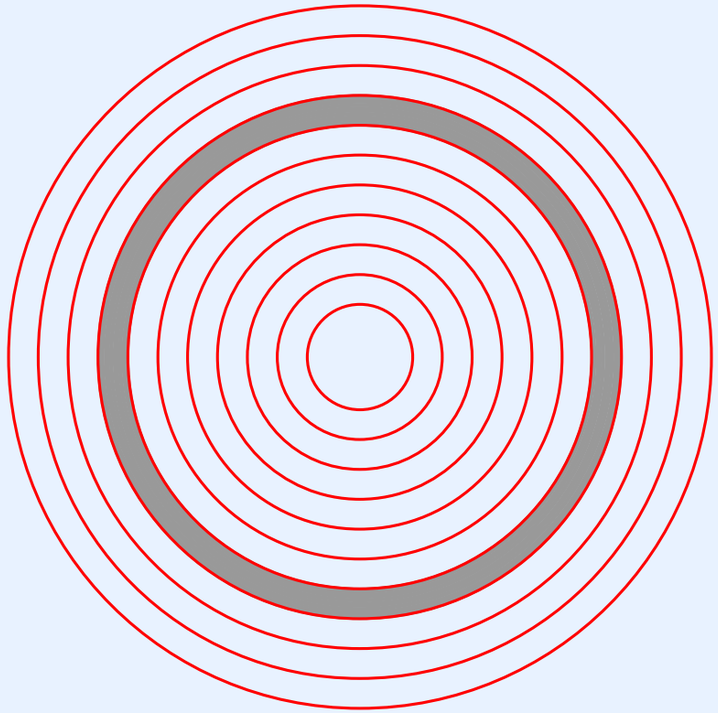
```
/* excerpt from memory.h */

/*------------------------------------------------------------------------
 *  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 VIII

High-level Memory Management

Location Of High-level Memory Management In The Hierarchy



Our Approach To Memory Management (Review)

- Divide 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
  - Disk subsystem allocates buffers for disk blocks
  - Network subsystem allocates packet buffers
- Interaction among subsystems can be subtle and complex

Managing Memory

- Conflicting philosophies and tradeoffs
  - Protecting information
  - Sharing information
- Extreme examples
  - Xinu has much sharing; little protection
  - Original Unix™ had much protection; little 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, anothers can e deprived
- Conclusions
  - We must not treat memory as a single, global resource
  - We need a way to isolate subsystems

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 Memory Resources

- Disk buffers
- Network buffers
- Message storage
- Inter-process communication buffers (e.g., Unix pipes)

Notes:

* Each subsystem should operate safely and independently
* An OS may defines separate sets of buffers for each network interface (e.g., Wi-Fi and Ethernet)

Xinu High-level Memory Manager

- Partitions memory into set of *buffer pools*
- Each pool is created once and persists until system shuts down
- At pool creation, we fix 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
  - Uses a synchronous interface

Virtual Memory

Definition Of Virtual Memory

- Abstraction of physical memory
- Provides separation from underlying hardware details
- Primarily applied to applications (user processes)
- Allows applications to run independent of
  - Physical memory size
  - Position in physical memory
- Many mechanisms have been proposed and used

General Approach

- Heavyweight process
  - Lives in an isolated address space
  - All addresses are *virtual*
- Operating system
  - Establishes policies
  - Provides support for virtual address space creation
  - Configures the hardware
- Underlying hardware
  - Dynamically translates from virtual address to physical address
  - Provides support to help OS make policy decisions

Virtual Address Space

- Can be smaller than physical memory
  - A 32-bit computer with more than 2^{32} bytes (four GB) of memory
- Can be larger than the physical memory
  - 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 items to main memory as needed
  - Write items back to disk to create space in memory for other items
- Questions
  - How much of a process's address space should reside in memory?
  - When should items be loaded into memory?
  - When should items be written back to disk?

General Approaches

- *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

General Approaches
(continued)

- *Segmentation with paging*
  - Divide an image into large segments
  - Further subdivide segments into fixed-size pages
- Note: simple paging has emerged as the most popular

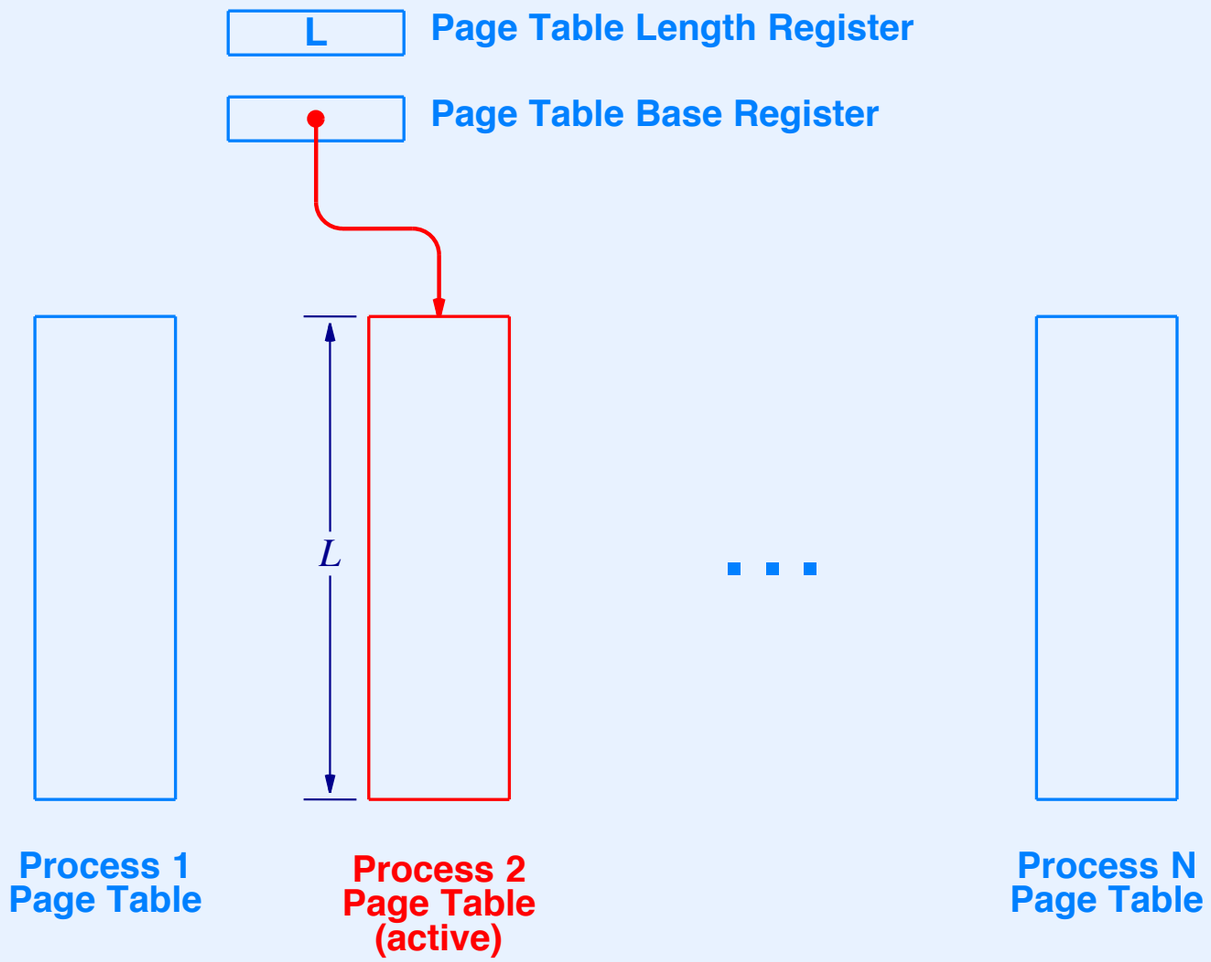
Hardware Support For Paging

- Page tables
  - One page table per process
  - Storage location depends on hardware
    * Kernel memory
    * MMU hardware (on some systems)
- Page table base register
  - Internal to the processor
  - Contains the address of current process's page table
  - Must be changed during a context switch

Hardware Support For Paging
(continued)

- Page table length register
  - Internal to the processor
  - Specifies the number of entries in the current page table
  - Determines the size of the virtual address space
  - Can be changed during context switch if the size of the virtual address space differs among processes

Illustration Of VM Hardware Registers



- Only one page table is active at given time

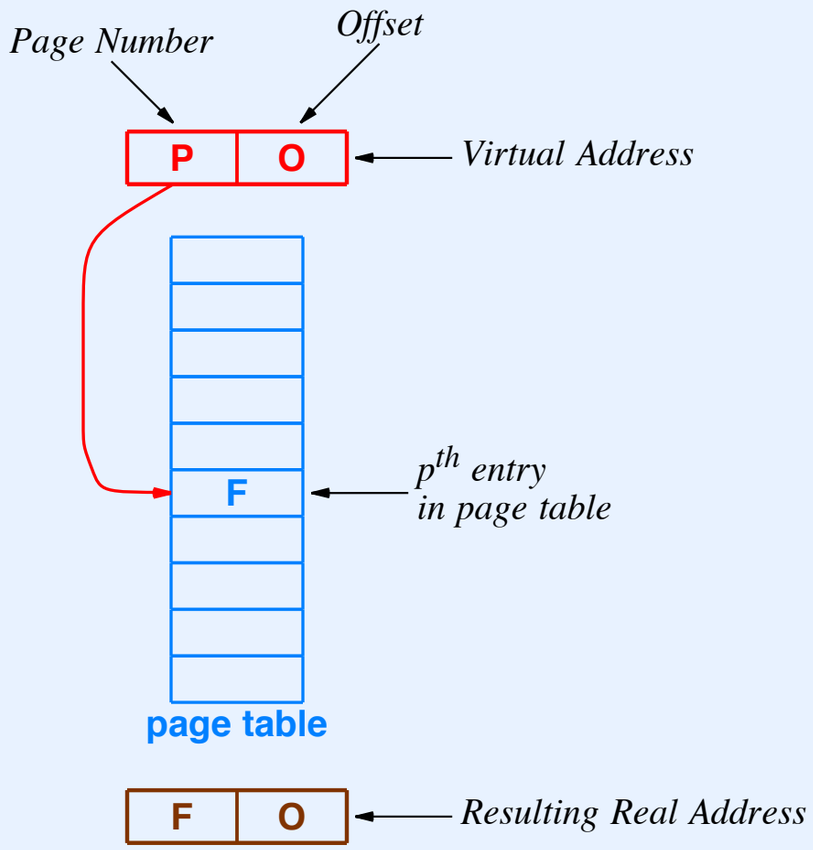
Address Translation

- Is performed by memory management hardware
- Must be applied to *every* memory reference
- Hardware translates from process's virtual address to corresponding physical memory address

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
  - The page size is 4 Kbytes
- Think of the physical memory area use 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



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

In Practice

- Some hardware offers separate page tables for text, data, and stack
  - Disadvantage: complexity
  - Advantage: independent policies
- The size of virtual space may be limited to physical memory size
- The kernel address space can also be virtual (but it hasn't worked well in practice)

Demand Paging

- Keep the process image on secondary storage
- Treat main memory as cache of recently-referenced pages
- Allocate space in the cache dynamically as needed
- Move pages between the secondary store and main memory on demand (when referenced)

The Importance Of Hardware Support For Virtual Memory

- Every memory reference must be translated from a virtual address to a physical address
  - Instructions must be translated as well as data (to accommodate branching)
  - Indirect addresses that are generated at runtime must be translated
- 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
  - Instruction fetch
  - Operand fetch
  - Indirect reference resolution
  - Memory copy instructions on CISC hardware
- Special-purpose hardware speeds page lookup
  - *Translation Look-aside Buffer* (*TLB*)
  - Implemented with T-CAM
  - TLB caches most recent address translations

In Practice
(continued)

- VM mappings change during context switch
  - Some hardware requires OS to flush TLB
  - Other hardware uses tags to distinguish among address spaces
    * Tag assigned by OS
    * Typically, a process ID
- Can page tables be paged?
  - On some hardware, yes
  - Extremely inefficient (impractical)

Bits That Record Page Status

- Part of the page table entry for each page
- Understood by hardware
- *Use Bit*
  - Set by the hardware whenever page referenced
  - Applies to all fetch and store operations
- *Modify Bit*
  - Set by the hardware when a *store* operation occurs
- *Presence Bit*
  - Set by the OS to indicate that the page currently *resident* in memory
  - Tested by the hardware when the page is referenced

Questions For OS Designers

- Which VM policies are most effective?
- Which pages from which 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?

A Critical Trade-off For Demand Paging

- Paging overhead and latency for 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
- What is the best choice?

Terminology

- *Reference string*: a list of page references emitted by a process
- *Resident set*: the subset of process's pages currently present in main memory
- *Page*: a fixed size piece of process's address space
- *Frame*: a "slot" in main memory exactly the size of a page
- *Dirty*: a page that has been modified since it was last written to secondary store
- *Page fault*: an error that occurs when a referenced page is not present

Page Replacement

- Hardware
  - Detects a page fault because a referenced page is not resident
  - Raises an exception
- Operating system
  - Receives the exception
  - Allocates a frame
  - Retrieves the needed page, and allows other processes to execute while page is being fetched
  - Allows the blocked process to continue once the page arrives

Frame Allocation

- A frame must be allocated when a page fault occurs
- Allocation is trivial if free (unfilled) frames exist
- Allocation is difficult if all frames are currently used
- A policy is needed because operating system must
  - Select one of the resident pages and save a copy on disk
  - Mark the page table to indicate the page is no longer resident
  - Load the needed page into the frame
  - Set the appropriate page table entry
  - Return from the page fault to restart the operation
- Question: which frame should be selected when all are in use?

Choosing A Frame

- Competition can be
  - Global: consider frames from all processes when choosing
  - Local: choose a frame within the same process that caused the page fault
- Possible selection policies
  - *Least Recently Used* (*LRU*)
  - *Least Frequently Used* (*LFU*)
  - *First In First Out* (*FIFO*)

Example Page Replacement Algorithms

- We will consider three examples
  - Belady's optimal page replacement algorithm
  - Global clock (second chance algorithm)
  - Working set algorithm

Belady's Optimal Algorithm

- Chooses a page that will be referenced farthest in the future
- Provably optimal
- Totally impractical (of theoretical interest only)
- Useful for comparison of other algorithms
- Corollary: increasing the physical memory size does not always decrease the rate of page faults (known as Belady's Anomaly).

Global Clock Algorithm

- Originated in the MULTICS operating system
- Allows processes to compete with one another (hence the term *global*)
- Relatively low overhead
- Widely implemented (the most popular practical method)

Global Clock Paradigm

- Clock algorithm is activated when a page fault occurs
- Searches frames in memory, and selects a frame to use
- Gives a frame containing a referenced page a “second chance” before reclaiming
- Gives a frame containing a modified page a “third chance” before reclaiming
- In the worst case: the clock sweeps through all frames twice before reclaiming one
- Does *not* require external data structure other than standard page table bits

Operation Of The Global Clock

- Uses a global pointer that picks up where it left off previously
  - Sweeps slowly through all frames in memory
  - Starts moving when a frame is needed
  - Stops moving once a frame has been selected
- During the sweep, the algorithm checks *Use* and *Modify* bits of each frame
- Reclaims the frame if *Use* / *Modify* bits are *(0,0)*
- Changes *(1,0)* into *(0,0)* and bypasses the frame
- Changes *(1,1)* into *(1,0)* and bypasses the frame
- The algorithm keeps a copy of the actual modified bit to know whether page is dirty

In Practice

- The global clock always reclaims a small set of frames when one is needed
- The reclaimed frames are cached for subsequent references
- Advantage: collecting multiple frames avoids running the clock frequently

Level Of Multitasking

- If too many processes are running
  - Each process will not have many frames
  - Paging activity will be high
  - Throughput will be low
- If too few processes are running, paging activity is low, but
  - The processor will not have a process to run while other processes wait for I/O
  - Throughput will be low

Load Balancing

- Refers to adjusting the number of processes that compete for frames
- Goal is to maximize throughput
- Works best in systems that have
  - A steady, predictable workload
  - A large set of available processes
- Is performed in cooperation with, and response to, the global clock
- Multitasking level (number of active processes) is
  - Decreased when page faults are frequent
  - Increased when paging level is low

Difficulties In Load Balancing

- Choosing thresholds
- Collecting measurements
- Anticipating paging activity
- It is easy to overreact

Working Set Algorithm

- More theory
- Finer level of assessment
- Definition

> **Let δ be the size of a window on a reference string measured in page references. A process's** working set **at time** i, w(i), **consists of the set of pages in the δ references immediately preceding time** i.

- Note: a *working set* is a set in the mathematical sense — it does not contain duplicates

Working Set Example For $\delta = 12$

3 2 5 3 8 3 27 3 6 27 2 11 5 1 2

w(1) = 3
w(2) = 3, 2
w(3) = 3, 2, 5
w(4) = 3, 2, 5
w(5) = 3, 2, 5, 8
w(6) = 3, 2, 5, 8
w(7) = 3, 2, 5, 8, 27
w(8) = 3, 2, 5, 8, 27
w(9) = 3, 2, 5, 8, 27, 6
w(10) = 3, 2, 5, 8, 27, 6
w(11) = 3, 2, 5, 8, 27, 6
w(12) = 3, 2, 5, 8, 27, 6, 11
w(13) = 3, 2, 5, 8, 27, 6, 11
w(14) = 3, 2, 5, 8, 27, 6, 11, 1
w(15) = 3, 2, 5, 8, 27, 6, 11, 1

Working Set Terminology

- *Working set size*: number of pages in the process's working set
- *Resident set size*: number of pages of process currently resident
- Note: both resident and working sets vary over time

Working Set Replacement Policy

Allocate to each process a resident set size equal to the process's working set size

Level Of Multitasking Under Working Set

- Level of multitasking
  - Is adjusted dynamically
  - Is a function of current working set sizes and memory size
- Idea: compare the sum of the working set sizes, W, and frames, F
  - Reduce multitasking when

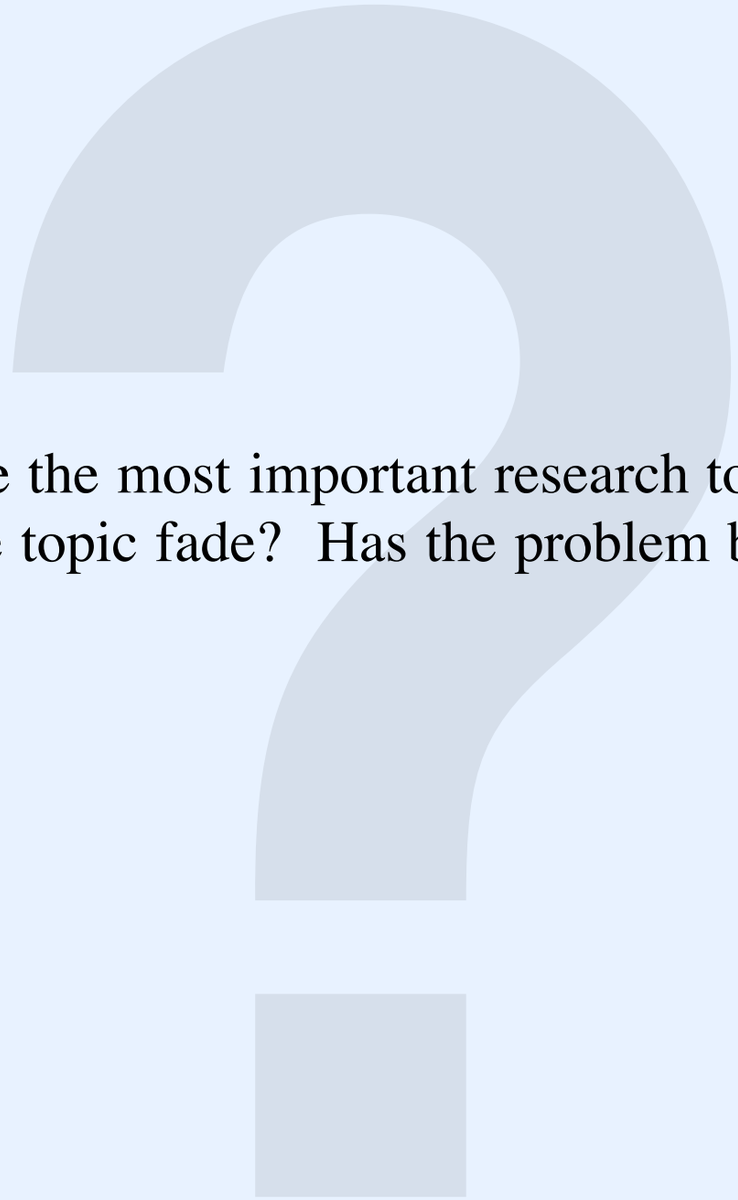
$$W > F$$

  - Increase multitasking when

$$W << F$$

Assessment Of Working Set Model

- Pros
  - Simulation shows it performs well
  - It uses paging performance as a way to balance load
- Cons
  - Need both a minimum and maximum bound on resident set size
  - Difficult (impossible) to capture the needed information
    * Working set must be recomputed on each memory reference
    * Even a hardware solution is inadequate



Virtual memory was once the most important research topic in operating systems. What changed? Why did the topic fade? Has the problem been solved completely?

Summary

- We considered two forms of high-level memory management
- Inside the kernel
  - Define a set of abstract resources
  - Firewalling memory used by each prevents interference
  - Mechanism uses buffer pools
  - Buffer referenced by single address
- Outside the kernel
  - Swapping, segmentation, or paging

Summary
(continued)

- Demand paging
  - Fixed size pages
  - Brought into memory when referenced
- Algorithms
  - Belady's algorithm (theoretical)
  - Global clock (widely used and practical)
- Working set (theoretical) relates paging to load balancing


Questions?