Memory Allocation Overview

Witawas Srisa-an

CSCE496/896: Embedded Systems Design and Implementation

(Getting Ready for the Final Assignment-part 1)

Motivation

- It has been said in 1998 that …
  - Java spends [Arm98]
    - 20% in dynamic memory management
    - 19% in synchronization
    - 60% in interpretation
    - 1% in native method
  - That was almost 6 years ago, nowadays
    - Much faster collector
    - JIT instead of interpretation
    - Thin-lock for lightweight synchronization
  - How did they do it?

Dynamic Memory Management

- Agenda
  - The basic information about DMM
  - Various Dynamic Memory Management Algorithms [WJN 95]
  - Cost of dynamic memory management in O-O languages [DDZ 93, QSSC 02]
  - Memory management for servers
  - Memory management for embedded systems

Why DMM?

- Major source of expense
  - 38% in C++
  - 20% in Java
  - X% in .NET
- Non-real-time
- May not be thread-safe
- Fragmentation

Dynamic Memory Management (DMM)

- Heap versus stack allocation
  - Stack allocation is used for data where the lifespan is the same as the method
    - Objects created when entering a method and destroyed when the method return
  - Heap allocation is used when data can outlive the method that created them
    - An object is created in the heap; the reference to it is stored in the stack

DMM

- In C++, programmers can choose to have their objects created on either stack or in the heap
- In Java and .Net, all objects are created in the heap

DMM

- Let's do a simple exercise
  - Available from: ~witty/share/csce496/lecture1/example1.c
  - Spend a few minutes to look at the code in groups of 3
    - What do you expect to see at print statement 1 – 6 if select is 0? What if select is 1?
  - Let's execute the code
    - Issue a.out 0
    - Issue a.out 1

DMM

- Notice that malloc is used to allocate objects
  - Is the result as expected?
  - Explain print statement 2 and 3
  - What about statement 3 and 4?
  - Did we create garbage? How?

DMM

- Uncollected garbage results in memory leaks
- From our exercise, when the application exits, do leaks go away?

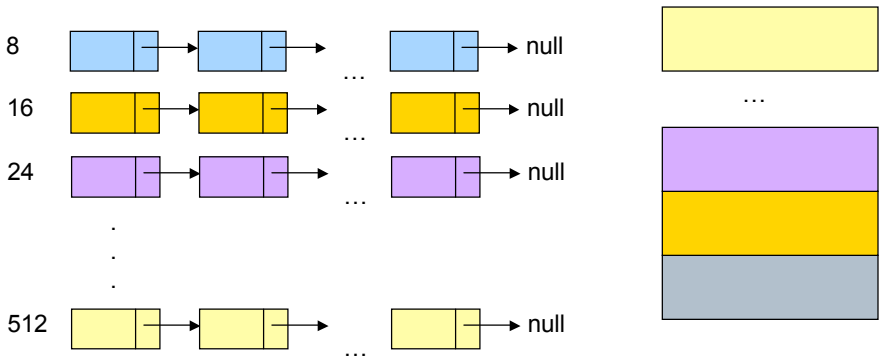
DMM Analogy

- Assuming you've just bought 1000 acres and plan to develop the land
  - Define each lot as half acres
  - Assume you have sold some lots
  - If somebody drop you off in the middle of this undeveloped land, can you tell whether the lot you are standing in has been sold?
    - What if you have built a road and assigned address to each lot?

DMM Analogy

- Analogy continue
  - What if you have paper record of every address at your office, can you find out whether the lot has been sold?
- Dynamic memory management is the same, it is a matter of bookkeeping
  - How fast can you create just the right size?
  - How do you organized the free space?
  - How soon can you look for a free space of certain size?
  - How soon can you free the space?
  - How soon can you merge the adjacent freed neighbors back together?

DMM



Link-list structures to represent heap status

One common approach for heap management

Memory Allocator

``It has been proven that for any possible allocation algorithm, there will always be the possibility that some application program will allocate and deallocate blocks in some fashion that defeats the allocator's strategy."

Paul R. Wilson *et al.*

Memory Allocator Should …

- Maximize compatibility
  - Easily linked to all sorts of application
  - POSIX compliance
- Maximize portability
  - Doesn't rely on system specific features
  - Provide optional support for useful features on some systems
- Minimize space usage
  - Efficiently implemented

Memory Allocator Should …

- Minimize time
  - As fast as possible in average case
- Maximize Tunability
  - User should be able to fine tune the allocator
- Maximize locality
  - Allocate chunks that are used together near each others
- Maximize debugging feature
- Minimize anomalies
  - Perform well across wide range of applications

What Must an Allocator Do?

- Keep track of which parts of memory are in use and which are free
  - Minimize wasted space without undue time cost
    - Negligible time managing memory and waste negligible space
  - Cannot control the number or size of live blocks
  - Cannot compact memory

What Must an Allocator Do?

- Respond immediately to a request for space
  - Once a free block is chosen, it is inviolable until freed
- Thus, it is an on-line algorithm
  - Respond in strict sequence, immediately, and decisions are not irrevocable

Designing Issues

- Alignment – word alignment causes more memory usage but allow easy scanning and searching
- How much space should be sacrificed for speed
  - The fastest way to free is not to free at all
    - Is it acceptable?
- How much speed should be sacrificed for space
  - Best-fit everything
    - Is it acceptable?

Allocation Overhead

- Splitting and Coalescing are the two major overheads for *malloc*
  - Splitting -- break larger block into smaller chunks
  - Coalescing -- combine free neighboring blocks into a larger contiguous chunk
  - Should be avoid as much as possible

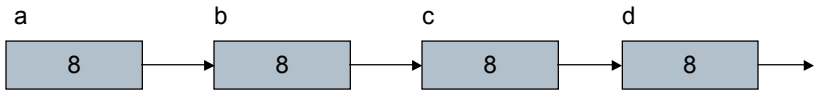
Allocation Holes

- Two types of fragmentation—internal and external
  - Internal because wasted memory is inside an allocated block rather than being recorded as a free block
  - External arises when free blocks of memory are available but can't be used to hold objects

Internal Fragmentation

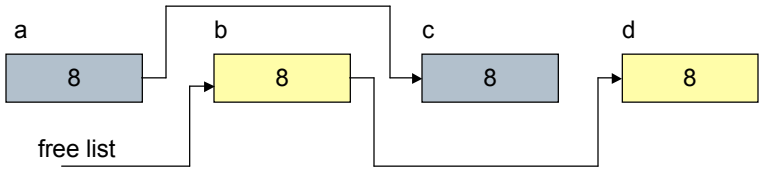
- Assuming you have decided to
  - Split a 4K page into fixed size chunk
  - Each chunk is 32 bytes (8 bytes headers, 24 bytes payload)
  - If a program request 18 bytes, the allocator will give a 32-byte chunk. Thus, 6 bytes are wasted.

External Fragmentation



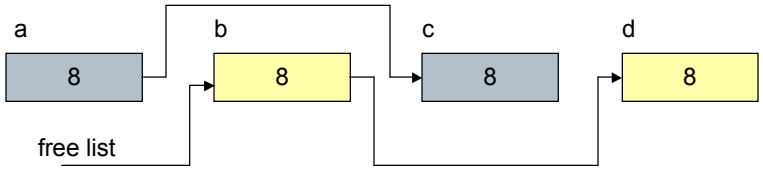
chunks of 8-byte payload are all occupied

External Fragmentation



b and d are freed

External Fragmentation



16 bytes is requested by the user's program
Cannot satisfy the request using b and d since they are not contiguous.

Fragmentation

- Can be disastrous
  - There are no algorithms for ensuring efficient memory usage (none is possible)
  - For any allocation algorithm, an application may force it into severe fragmentation
    - Plenty of proofs that any allocator will be bad for some applications
  - "There are regularities in program behavior that allocators exploit"

Strategy, Policy, and Mechanism

- **Strategy** considers regularities in program behaviors and determines a range of acceptable **policies** as to where to allocate blocks
- **Mechanism** is a set of algorithms and data structures they use.

Strategy and Policy

- Ideal strategy—"put block where they won't cause fragmentation later"
- Real strategies—heuristically approximate the ideal based on assume regularities
  - Doug Lea's—capitalize on reuse, classified blocks by bin for easy look up. Spend more space to gain speed for coalescing (header and footer)
- Policies determines exactly where in memory blocks will be allocated
  - Doug Lea's—prioritize (freebies, coalescing, splitting, system calls)

Policy consideration…

- Memory reuse
- When to coalesce, when to split
- What kind of fits
- What to do when the fitting is poor

Mechanisms

- Header fields and alignment
  - *free()* does not require size to be passed as argument; thus, size is often stored on the header
  - Bit to indicate whether a block is in use or not
  - Status of the neighbor may be recorded as well
  - Often use 1 or 2 machine words
    - Can be quite costly if the object size is small

Mechanisms

- Boundary tags
  - Each block has header and footer fields
    - Size and block status is recorded in the header and footer
    - Doug Lea's malloc follows this approach
    - Footer field can coexist with the payload
      - During allocation, footer information is not needed
      - When the block is free, we then need the size field so that it can be used to locate the header for coalescing
- Bit-maps

Common Algorithms

Sequential Fits

- Best fit
  - Strategy—minimize the amount of wasted space
    - Ensure small fragments
      - Can backfire if fragments are too small and unusable
  - Problem—Exhaustive search
    - Scale poorly (why?)
    - Worst-case performance is poor

Sequential Fits (cont.)

- First fit
  - Strategy—minimize search time by
    - Taking the first block that is big enough to satisfy the request
      - Splitting is commonly needed
  - Problem—Larger blocks at the beginning are split first resulting in splinters
    - Search time can grow large due to splinters
    - Scale poorly in system with large objects and many different-sized free blocks accumulate
  - Policy—where to put free blocks
    - Head of the free list
    - Address ordered (commonly used due to less fragmentation)

Sequential Fits (cont.)

- Next fit
  - Strategy—go one step beyond first fit to reduce average search time
    - Eliminate splinters by using roving pointers
  - Problem—roving pointer cycles through memory
    - Can result in bad cache locality
    - Can affect program locality

Segregate Free Lists

- In simplest form, uses an array of free lists
  - Strategy—give best fit without linear search
  - Policies
    - Simple segregated storage—one page contains only one size (used in Kaffe JVM [kaffe])
      - No header per object (why?)
      - Fast but can suffer from external fragmentation (no splitting or coalescing is allowed)

Segregated Free Lists (cont.)

- Policies (cont.)
  - Segregated fit—array of free lists. Each list contains free blocks within a size class
    - If there is not a free block in the size class list
      - Go to larger list and try to split
    - Smaller blocks are preferred over larger block (similar to best fit)
    - Coalescing can result in longer search time
      - e.g. coalesce then split again
    - Doug Lea's 2.5 uses this scheme for objects larger than 512 (what about objects smaller than 512 bytes)

Segregate Free Lists (cont.)

- Three general categories of segregate fits
  - Exact lists—conceptually a separate free list fore each possible block size
  - Strict size class with rounding—round up so that the requested size fits the predefined classes
  - Size classes with range lists—list can contain blocks of slightly different sizes

Buddy Systems

- Split the heap into two areas
  - Then those two areas are further split into smaller areas and so on
- For each allowable size, a separate free list is maintained as an array of free lists
- Coalecsing is peculiar
  - A free block may only be merged with its buddy (unique neighbor at the same level in the binary hierarchical division)

Buddy Systems (cont.)

- Strategy—fast allocation (segregated fit) and coalesing
- Problem—high external fragmentation (coarse grain class sizes)
  - Binary buddy
  - Fibonaci buddy
  - Weighted buddy
  - Double buddy

Doug Lea's Malloc

- Widely used in many systems
  - Linux
  - Many flavors of Unix
  - Sun's JDK, Kaffe's VM
- Arguably the most used *malloc* in the world

Basic Design

- Doug Lea uses delay coalescing
  - Freed objects are returned to a free list
  - New allocations start from here
    - Check the last 2 freed (or preallocated) objects
    - Check dirty bins
    - Going through the free list. If one fit pick it, other wise return to dirty bin
    - Look for the remainder of last split
    - Go through clean list in the bin
    - If exact size can't be found split

Reference

[Arm98] E. Armstrong. *HotSpot: A new breed of virtual machine*. JavaWorld: IDG's magazine for the Java community, March 1998. http://www.javaworld.com/javaworld/jw-03-1998/jw-03hotspot p.html.

[DDZ94] D. L. Detlefs, Al Dosser and B. Zorn, `Memory allocation costs in large C and C++ programs', Software Practice and Experience, 24(6), 527--542 (1994). workshop

[WJN95] P. R. Wilson et al. Dynamic storage allocation: A survey and critical review. Proc. Int'l on Memory Management. Kinross, Scotland, UK, Sep. 1995.

[Lea] Doug Lea, "A Memory Allocator" available from http://gee.cs.oswego.edu/dl/html/malloc.html

[Kernighan88] Brian W. Kernighan and Dennis M. Ritchie, "The C programming language", Prentice Hall Press, 1988.