# Algorithms for Dynamic Memory Management (236780)

## Lecture 8

Lecturer: Erez Petrank

# Topic last week

- Snapshot & Sliding Views

# Topics this week (extended)

- Allocations
- Parallel
- Emery's talk

# Allocation Techniques

# Allocation techniques

- Fragmentation
- Some basic notions and techniques
- Doug Lea's allocator
- Boehm's allocator
- Allocation caches (IBM)
- Hoard [Berger, McKinley, Blumofe, Wilson 00]

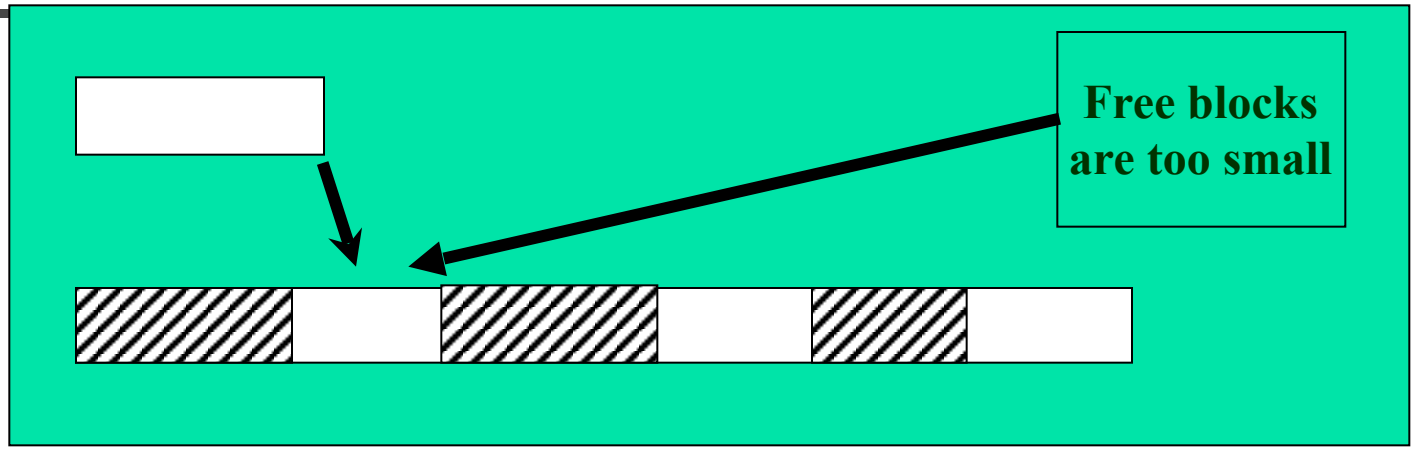
# Allocator is measured by

- Speed of allocation
- Fragmentation
- (Speed of reclamation)
- Cache-conscious placement.

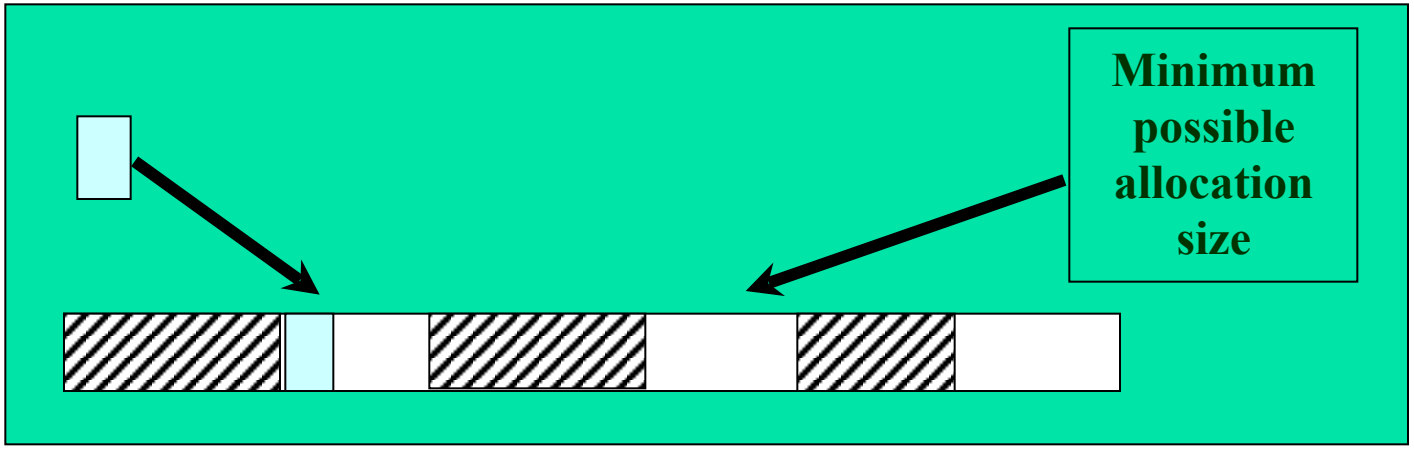
# Fragmentation

- Inability to reuse memory that is free
  - Severity determined by distribution of holes, and requests of the program.
  - Caused by reclamation or allocation policies (e.g., allow only certain sizes on one page).
- External fragmentation: holes outside the objects.
- Internal fragmentation: allocated area is larger than requested area.

External frag.




Internal frag.



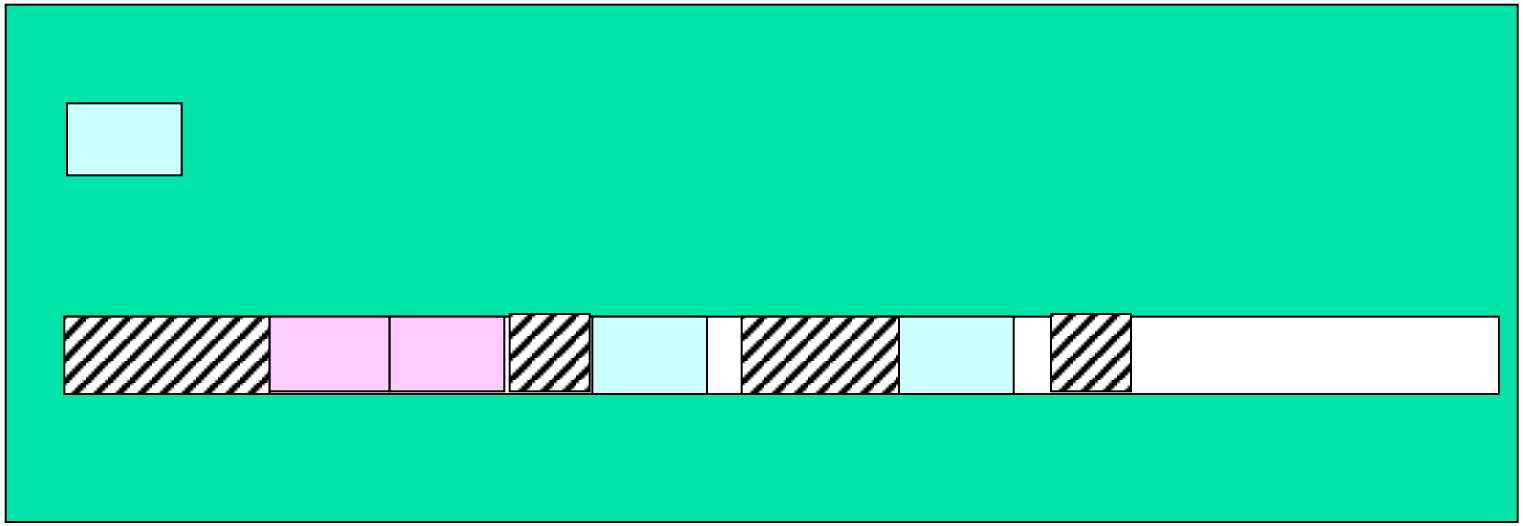
# Basic Notions and Techniques

- Various fits: best fit, first fit.
- Indexed fits
- Segregated free lists
- Boundary tags

# Best fit

- Allocate in the smallest free block that may satisfy the request.
- In practice,
  - Exploit most of the used “hole”.
  - remainder will be quite small and perhaps unusable.

# Best Fit Illustrated



# Best Fit (Cont.)

- Naïve implementation: search the whole free-list (linear complexity, unacceptable).
- Common implementations: use balanced binary trees, or keep a list of available blocks for each possible allocation size ("indexed fits" or "segregated fits").
- Note difference between policy and implementation (Wilson et al.)
- Best fit has low fragmentation with typical benchmarks.

# First Fit

- Allocate in the first sufficiently large free block.
  - Typically address-ordered,
  - can be "free-list ordered", or other.
- Search, split found block, put remainder on free-list.

# First Fit behavior

- Lots of small blocks near the beginning of the list.
- These "splinters" (שבבים, רסיסים) increase fragmentation and may increase search times.
- But: normally maintains very low fragmentation.

# Indexed Fits

- Use an indexing data structure to obtain efficient searching of a desired policy.
- Examples:
  - Best fit with a balanced tree.
  - Buddy systems
  - Bitmap fits – use a bitmap to search for free space…

# Modern Allocators

# Using Header Fields

- Most allocators use a hidden "header" field within each allocated area to store useful information like:
  - Size of the block, whether the block is in use, its class object, locking info, hash info, garbage collection info (reference count, mark-bit) etc.

# Coalescing via Boundary Tags

- For coalescing efficiency, allocated areas may also contain a "footer" field, with size of block.
- When a block is freed, examine footer of preceding block and header of following block for coalescing.
- Space saver: use footer only if object is free. Use a bit in the next object header to indicate if it is.

# Segregated Free Lists

- Typical structure: an array of free lists.
  - Each list holds free chunks of a particular size.
  - A freed chunk is pushed to the appropriate free list by the collector.
  - Allocation uses appropriate list.
  - A pool of free blocks is kept aside.
- Preferred implementation for approximate best fit.

# Segregated Free Lists Variations

- Each free list has a range of sizes for allocation.
  - Search for a chunk in the list using best fit, first fit, or next fit. (Typically --- first fit.)
- Number of lists.
  - A small number of lists may increase internal fragmentation or allocation time.
  - A large number of lists may increase space overhead.

# Block-Oriented Segregated Free Lists

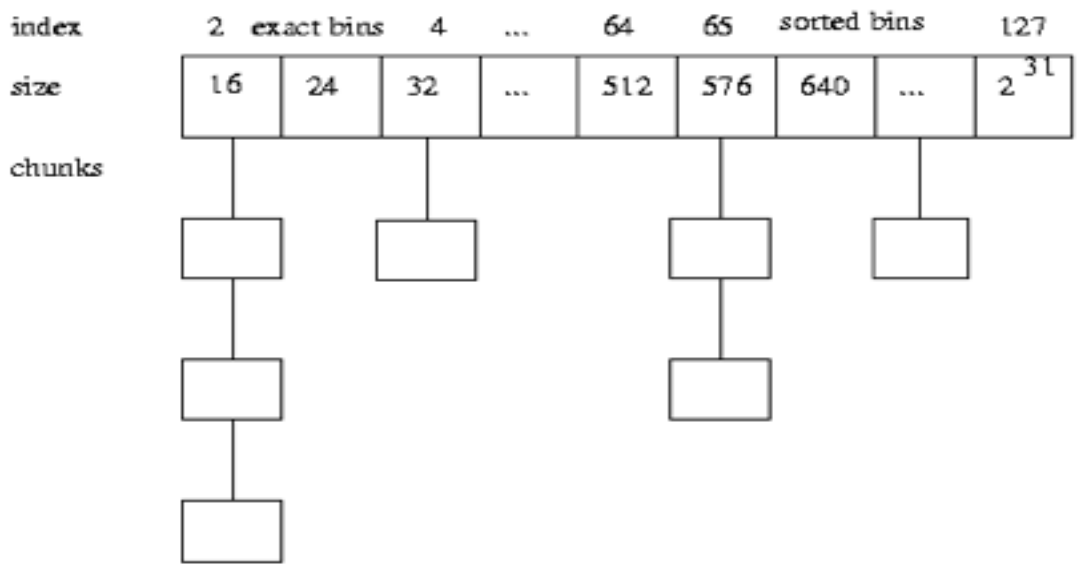
- Heap is partitioned to blocks (typically 4KB = page size).
- A block allocates only objects of same size.
- When a list is empty, a new block is obtained and split into same size objects.
- Reclamation:
  - Freed chunks are added to the appropriate list.
  - When a full block is reclaimed – it is returned to a pool and may later serve a different size.

# Segregated Free Lists
# Block Oriented

- Advantages
  - Shortened headers: (no need for size).
  - Efficient in time and space since typical programs use few sizes.
- Disadvantages:
  - External fragmentation.

# Algorithm used

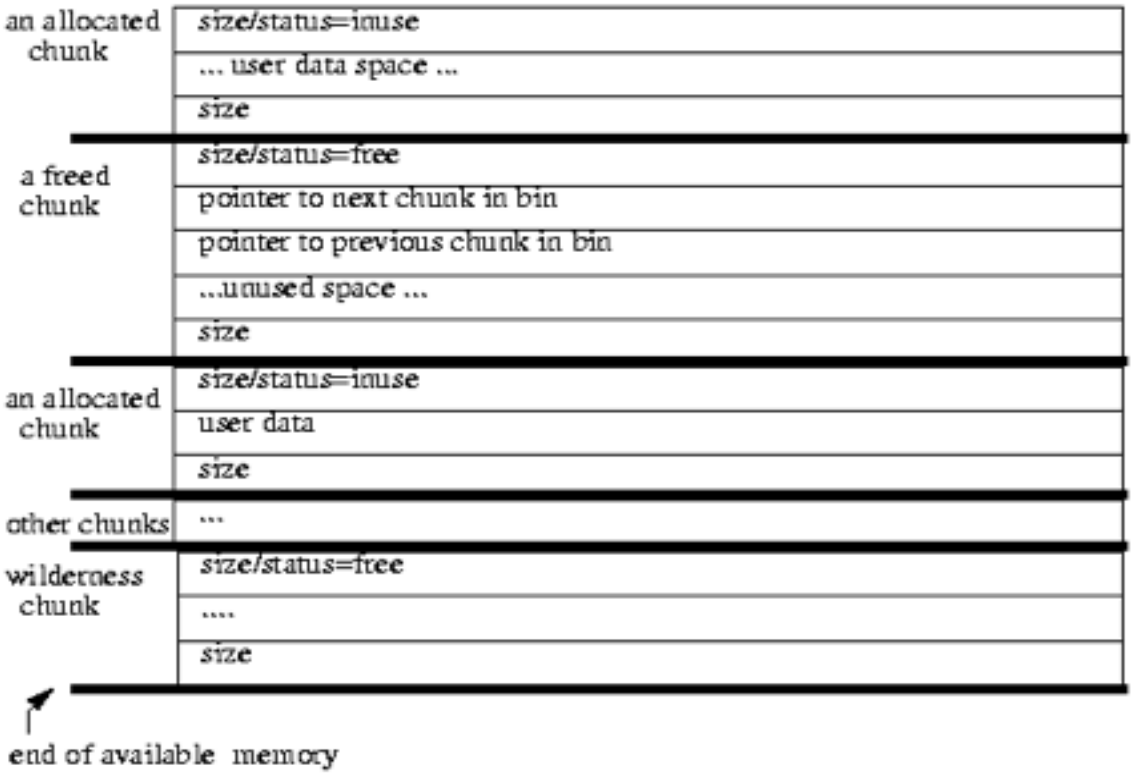
- **Segregated free lists** - Available chunks are maintained in bins, grouped by size.




- (Almost) Best fit

# Doug Lea's Malloc

- **Boundary Tags** for coalescing & traversing starting from any chunk in any direction.



# Improving Locality

- Use the following modification:
  - If cannot find space in the desired bin, attempt to allocate from the space most recently used for split.
  - If that fails, switch to best fit (and record the new block found for future allocations).
- Resulting algorithm is almost best fit.
- Wilderness Preservation:
  - The "wilderness" is the free space at the end of the heap.
  - This chunk will be used only if no other smaller suitable chunk exists.
- http://gee.cs.oswego.edu/dl/html/malloc.html

# Boehm's allocator (and collector)

- http://www.hpl.hp.com/personal/Hans_Boehm/gc/
- Collector by Boehm-Demers-Wiser.
- May be used as a leak-detector
- Distributed with the GNU compiler
- Available for most standard PC and UNIX platforms, Win32, OS/2, and UNIX environments.
- The collector uses a mark-sweep algorithm. With incremental and generational support.

# Boehm's allocator (and collector)

- Memory split into blocks, typically block size = page size.
- Free blocks are maintained in a tree sorted by address.
- Two level allocator:
  - Large objects are allocated from tree of blocks.
  - Each block is dedicated to a single object size.
  - Small objects are allocated in blocks.
  - The allocator maintains separate free lists for each size of object.

# Parallel Allocation

# Allocation for SMP's

- Boehm's allocator and segregated free lists in general can be extended for a multiprocessor.
  - Typically, by maintaining segregated free lists per thread.
- Next:
  - Allocation caches (IBM).
  - (Immix - not this semester).
  - HOARD.

# Allocation caches

- Two goals:
  - Reduce contention on allocation.
  - Allow "bump-pointer" allocation for mark-sweep.
- Method: let each thread obtain a "local cache" using synchronization.
- After obtaining the local cache: allocate (small objects) from it locally with a bump pointer.

# Method with Mark-Sweep

- All available spaces are kept in a free-list, created by sweep.
- When a local cache is needed, the first large-enough space is taken via first-fit.
- There is a minimum and maximum size for a cache.
  - Minimum – because we do not want to switch caches too often. Switching involves synchronization.
  - Maximum – because we do not want one thread to use all free space as its own cache, starving the other threads.
- All small objects are allocated from the cache.

# The Free List

- If a free chunk on the list is too large, only a piece of it is taken for the cache, and the rest is left in the free list.
- Allocation of larger objects (say, more than half the minimum cache size) is done directly from the free-list via first-fit.
- A simple optimization: sweep does not put small spaces in the free list. All free chunks are large enough to serve as caches.

# Adding Bit-Wise Sweep

- Letting sweep ignore small chunks is fine with allocation caches.
- A major overhead of sweep is finding & freeing each chunk of 0's in the mark-bit table.
- Method: sweep only looks for zero words.
- When found – free the whole chunk of zeros.
- Advantage: looking for a zero word is fast.
- Disadvantage: inaccurate decision which size is minimal.

# Allocation Caches Properties

- Cache behavior: it is believed that allocating sequentially is very good for program locality.
- Local caches provide bump-pointer allocation to mark-sweep.
- It is fast and cache-friendly.
- Most allocations are executed locally in the local cache.
- Synchronization is seldom and thus contention is low.

# Hoard

- A Scalable Memory Allocator for Multithreaded Applications [Berger- McKinley-Blumofe-Wilson 01].
- Download from http://www.hoard.org .

# To be presented by Emery Berger