Dynamic Memory Allocation

CSCI 2021: Machine Architecture and Organization

Pen-Chung Yew
Department Computer Science and Engineering
University of Minnesota

Outlines

- Basic concepts
- Implicit free lists
- Explicit free lists

Dynamic Memory Allocation

- Programmers use dynamic memory allocators (such as malloc) to acquire VM at runtime.
  - For data structures whose size is only known at runtime.
- Dynamic memory allocators manage an area of process virtual memory known as heap.

Dynamic Memory Allocation

- Allocator maintains heap as collection of variable sized blocks, which are either allocated or free
- Types of allocators
  - Explicit allocator: application allocates and frees space
    - e.g., malloc and free in C
  - Implicit allocator: application allocates, but does not free space
    - e.g. garbage collection in Java, ML, and Lisp

The malloc Package

```c
#include <stdlib.h>
void *malloc(size_t size)

#include <unistd.h>
void *sbrk(intptr_t incr);
```

- **malloc**
  - Returns a pointer to a memory block of at least `size` bytes (typically) aligned to 8-byte boundary
  - `size == 0`, returns NULL
  - **Unsuccessful**: returns NULL (0) and sets `errno`

```c
void free(void *p)
```

- frees the block pointed at by `p` to pool of available memory
- `p` must come from a previous call to malloc or realloc

Other functions

- calloc: Version of malloc that initializes allocated block to zero.
- realloc: Changes the size of a previously allocated block.
- sbrk: Used internally by allocators to grow or shrink the heap

malloc Example

```c
void foo(int n, int m)
{
    int i, *p);
    /* allocate a block of n ints */
    p = (int *)malloc(n * sizeof(int));
    if (p == NULL)
        perror("malloc");
        exit(0);
    }

    /* initialize allocated block */
    for (i=0; i<n; i++)
        p[i] = i;

    /* return p to the heap */
    free(p);
}
```
Assumptions Made Here

- Memory is word addressed (each word can hold a pointer)

![Allocated block (4 words)](image)

- Free block (3 words)
- Free word
- Allocated word

Allocation Example

<table>
<thead>
<tr>
<th>p1 = malloc(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2 = malloc(5)</td>
</tr>
<tr>
<td>p3 = malloc(6)</td>
</tr>
<tr>
<td>free(p2)</td>
</tr>
<tr>
<td>p4 = malloc(2)</td>
</tr>
</tbody>
</table>

Requirements in Designing Allocators

- Applications
  - Can issue arbitrary sequence of malloc and free requests
  - A free request must be to a malloc’d block

- Allocators’ Requirements
  - Can’t control number or size of allocated blocks
  - Must respond immediately to malloc requests
  - Must allocate blocks from free memory
    - i.e., can only place allocated blocks in free memory
  - Must align blocks to satisfy all alignment requirements
    - 8-byte alignment for GNU malloc (libc malloc) on Linux boxes
  - Can manipulate and modify only free memory
  - Can’t move allocated blocks once they are malloc’d
    - i.e., compaction is not allowed because other objects may have pointed to the allocated blocks

Performance Metric: Peak Memory Utilization

- Given some sequence of malloc and free requests:
  - \( R_0, R_1, \ldots, R_n \)

- Def: Aggregate payload \( P_k \)
  - malloc(p) results in a block with a payload of \( p \) bytes
  - After request \( R_k \) has completed, the aggregate payload \( P_k \) is the sum of currently allocated payloads

- Def: Current heap size \( H_k \)
  - Assume \( H_k \) is monotonically nondecreasing
    - i.e., heap only grows when allocator uses sbrk

- Def: Peak memory utilization after \( k \) requests
  - \( U_k = (\max_{i \leq k} P_i) / H_k \)

Performance Metric: Throughput

- Throughput:
  - Number of completed requests per unit time
    - 5,000 malloc calls and 5,000 free calls in 10 seconds
    - Throughput is 1,000 operations/second

Performance Goals

- Given some sequence of malloc and free requests:
  - \( R_0, R_1, \ldots, R_n \) -- sequence of allocate and free requests
  - \( P_0, P_1, \ldots, P_n \) -- aggregate payloads
    - \( H_0, H_1, \ldots, H_n \) -- resulting size (high water-mark) of heap

- Goals: maximize peak memory utilization and throughput
  - These goals are often conflicting
  - Peak Memory Utilization
    - \( U_k = (\max_{i \leq k} P_i) / H_k \) -- peak memory utilization after \( R_k \)
  - Throughput:
    - Number of completed requests per unit time
Fragmentation

• Poor memory utilization caused by fragmentation
  • internal fragmentation
  • external fragmentation

Internal Fragmentation

• For a given block, internal fragmentation occurs if payload is smaller than block size

  \[\text{Block} \quad \text{Payload} \quad \text{Internal fragmentation} \]

• Caused by
  • Overhead of maintaining heap data structures
  • Padding for alignment purposes
  • Explicit policy decisions (e.g., to return a big block to satisfy a small request)

• Depends only on the pattern of previous requests
  • Thus, easy to measure

External Fragmentation

• Occurs when there is enough aggregate heap memory, but no single free block is large enough

  \[\text{Oops! (what would happen now?)}\]

• Depends on the pattern of future requests
  • Thus, difficult to measure

Implementation Issues

1. How do we know how much memory to free given a pointer? 
   • i.e. how do we know the size of a block to be freed?
2. How do we keep track of the free blocks?
3. What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in? 
   • i.e. do we want to leave it as is, or split it to create one extra free block?
4. How do we pick a block for allocation -- many might fit?
5. How do we reinsert freed block?

Knowing How Much to Free

• Standard method
  • Keep length of a block in the word preceding the block.
  • This word is often called header field or header
  • Requires an extra word for every allocated block

\[\text{p0 = malloc(4)} \quad \text{p1 = malloc(5)} \quad \text{p2 = malloc(6)} \quad \text{free(p2)} \quad \text{p3 = malloc(6)} \quad \text{p4 = malloc(6)}\]

Keeping Track of Free Blocks

• Method 1: implicit list using length—links all blocks

\[\text{Method 2: Explicit list among the free blocks using pointers}
\]

\[\text{Method 3: Segregated free list}
\]

• Different free lists for different size classes

\[\text{Method 4: Blocks sorted by size}
\]

• Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key
Outline

- Basic concepts
- Implicit free lists
- Explicit free lists

Implicit List

- For each block we need both size and allocation status
  - Could store this information in two words: wasteful!
- Standard trick
  - If blocks are aligned, some low-order address bits are always 0
  - Instead of storing an always-0 bit, use it as an allocated/free flag
  - When reading size word, must mask out this bit

Format of allocated and free blocks

<table>
<thead>
<tr>
<th>Size</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Payload</td>
<td></td>
</tr>
</tbody>
</table>

Optional padding

Payload: application data (allocated blocks only)

Implicit List: Finding a Free Block

- First fit:
  - Search list from beginning, choose first free block that fits:
    - Can take linear time in total number of blocks (allocated and free)
    - In practice it can cause "splinters" at beginning of list
- Next fit:
  - Like first fit, but search list starting where previous search finished
  - Should often be faster than first fit: avoids re-scanning unhealthy blocks
  - Some research suggests that fragmentation is worse
- Best fit:
  - Search the list, choose the best free block: fits, with fewest bytes left over
  - Keeps fragments small—usually helps fragmentation
  - Will typically run slower than first fit

Implicit List: Allocating in Free Block

- Allocating in a free block: splitting
  - Since allocated space might be smaller than free space, we might want to split the block

void addblock(ptr p, int len) {
    int newsize = ((len + 1) >> 1); // round up to even
    *p = newsize | 1;               // set allocation bit
    if (newsize < oldsize) {        // need splitting?
        *(p + newsize) = oldsize - newsize; // set length in remaining
    }
}

Implicit List: Freeing a Block

- Simplest implementation:
  - Need only clear the "allocated" flag
    - void free_block(ptr p) { *p = *p & -2 } #set allocation bit 0
  - But can lead to "false fragmentation"

void free_block(ptr p) { *p = *p & -2 } #set allocation bit 0

Implicit List: Allocating in Free Block

- Allocating in a free block: splitting
  - Since allocated space might be smaller than free space, we might want to split the block

void addblock(ptr p, int len) {
    int newsize = ((len + 1) >> 1); // round up to even
    *p = newsize | 1;               // set allocation bit
    if (newsize < oldsize) {        // need splitting?
        *(p + newsize) = oldsize - newsize; // set length in remaining
    }
}

Implicit List: Freeing a Block

- Simplest implementation:
  - Need only clear the "allocated" flag
    - void free_block(ptr p) { *p = *p & -2 } #set allocation bit 0
  - But can lead to "false fragmentation"

void free_block(ptr p) { *p = *p & -2 } #set allocation bit 0

Implicit List: Finding a Free Block

- First fit:
  - Search list from beginning, choose first free block that fits:
    - Can take linear time in total number of blocks (allocated and free)
    - In practice it can cause "splinters" at beginning of list
- Next fit:
  - Like first fit, but search list starting where previous search finished
  - Should often be faster than first fit: avoids re-scanning unhealthy blocks
  - Some research suggests that fragmentation is worse
- Best fit:
  - Search the list, choose the best free block: fits, with fewest bytes left over
  - Keeps fragments small—usually helps fragmentation
  - Will typically run slower than first fit

void addblock(ptr p, int len) {
    int newsize = ((len + 1) >> 1); // round up to even
    *p = newsize | 1;               // set allocation bit
    if (newsize < oldsize) {        // need splitting?
        *(p + newsize) = oldsize - newsize; // set length in remaining
    }
}

Implicit List: Allocating in Free Block

- Allocating in a free block: splitting
  - Since allocated space might be smaller than free space, we might want to split the block

void addblock(ptr p, int len) {
    int newsize = ((len + 1) >> 1); // round up to even
    *p = newsize | 1;               // set allocation bit
    if (newsize < oldsize) {        // need splitting?
        *(p + newsize) = oldsize - newsize; // set length in remaining
    }
}

Implicit List: Freeing a Block

- Simplest implementation:
  - Need only clear the "allocated" flag
    - void free_block(ptr p) { *p = *p & -2 } #set allocation bit 0
  - But can lead to "false fragmentation"

void free_block(ptr p) { *p = *p & -2 } #set allocation bit 0

Implicit List: Allocating in Free Block

- Allocating in a free block: splitting
  - Since allocated space might be smaller than free space, we might want to split the block

void addblock(ptr p, int len) {
    int newsize = ((len + 1) >> 1); // round up to even
    *p = newsize | 1;               // set allocation bit
    if (newsize < oldsize) {        // need splitting?
        *(p + newsize) = oldsize - newsize; // set length in remaining
    }
}
Implicit List: Coalescing

- Join (coalesce) with next/previous blocks, if they are free
  - Coalescing with next block

```c
void free_block(ptr p) {
    p = p & -2; // clear allocated flag
    next = p + *p; // find next block
    if ((*next & 1) == 0)
        *p = *p + *next; // add to this block if not allocated
}
```

- But how do we coalesce with previous block?

Implicit List: Bidirectional Coalescing

- Boundary tags [Knuth73]
  - Replicate size/allocated word at "bottom" (end) of free blocks
  - Allows us to traverse the "list" backwards, but requires extra space
  - Important and general technique!

Constant Time Coalescing

<table>
<thead>
<tr>
<th>Case 1</th>
<th>Case 2</th>
<th>Case 3</th>
<th>Case 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1 = 1</td>
<td>m1 = 1</td>
<td>m1 = 1</td>
<td>m1 = 1</td>
</tr>
<tr>
<td>m2 = 1</td>
<td>m2 = 1</td>
<td>m2 = 1</td>
<td>m2 = 1</td>
</tr>
<tr>
<td>n = 1</td>
<td>n = 0</td>
<td>n = 1</td>
<td>n = 0</td>
</tr>
<tr>
<td>n = 0</td>
<td>n = 0</td>
<td>n = 0</td>
<td>n = 0</td>
</tr>
<tr>
<td>m2 = 0</td>
<td>m2 = 1</td>
<td>m2 = 1</td>
<td>m2 = 1</td>
</tr>
</tbody>
</table>

Constant Time Coalescing (Case 1)

Constant Time Coalescing (Case 2)

Constant Time Coalescing (Case 3)
Disadvantages of Boundary Tags

- Internal fragmentation
- Can it be optimized?
  - Which blocks need the footer tag?
  - What does that mean?

Summary of Key Allocator Policies

- Placement policy:
  - First-fit, next-fit, best-fit, etc.
  - Trades off lower throughput for less fragmentation
- Interesting observation: segregated free lists
  - Approximate a best fit placement policy without having to search entire free list
- Splitting policy:
  - When do we go ahead and split free blocks?
  - How much internal fragmentation are we willing to tolerate?
- Coalescing policy:
  - Immediate coalescing: coalesce each time free is called
  - Deferred coalescing: try to improve performance of free by deferring coalescing until needed. Examples:
    - Coalesce as you scan the free list for malloc
    - Coalesce when the amount of external fragmentation reaches some threshold

Implicit Lists: Summary

- Implementation: very simple
- Allocate cost:
  - Linear time worst case
- Free cost:
  - Constant time worst case even with coalescing
- Memory usage:
  - Depend on placement policy
  - First-fit, next-fit or best-fit
- Not used in practice for malloc/free because of linear-time allocation
  - Used in many special purpose applications
- However, the concepts of splitting and boundary tag coalescing are general to all allocators

Outlines

- Basic concepts
- Implicit free lists
- Explicit free lists

Keeping Track of Free Blocks

- Method 1: Implicit free list using length—links all blocks
- Method 2: Explicit free list among the free blocks using pointers
- Method 3: Segregated free list
  - Different free lists for different size classes
- Method 4: Blocks sorted by size
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key
Explicit Free Lists

• Maintain list(s) of free blocks, not all blocks
  • The "next" free block could be anywhere
  • So we need to store forward/back pointers, not just sizes
  • Still need boundary tags for coalescing
• We track only free blocks, so we can use payload area

Allocating From Explicit Free Lists

Before

After (with splitting)

= malloc( )

Freeing With Explicit Free Lists

• Insertion policy: Where in the free list do you put a newly freed block?
  • LIFO (last-in-first-out) policy
    • Insert freed block at the beginning of the free list
    • Pro: simple and constant time
    • Con: studies suggest fragmentation is worse than address ordered
  • Address-ordered policy
    • Insert freed blocks so that free list blocks are always in address order:
      addr(prev) < addr(curr) < addr(next)
    • Con: requires search
    • Pro: studies suggest fragmentation is lower than LIFO

Freeing With a LIFO Policy (Case 1)

Before

Root

After

Freeing With a LIFO Policy (Case 2)

Before

Root

After

• Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list
Freeing With a LIFO Policy (Case 3)

- Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list

Freeing With a LIFO Policy (Case 4)

- Splice out predecessor and successor blocks, coalesce all 3 memory blocks and insert the new block at the root of the list

Explicit List Summary

- Comparison to implicit list:
  - Allocate is linear time in number of free blocks instead of all blocks
  - Much faster when most of the memory is full
  - Slightly more complicated allocate and free since needs to splice blocks in and out of the list
  - Some extra space for the links (2 extra words needed for each block)
  - Does this increase internal fragmentation?

- Most common use of linked lists is in conjunction with segregated free lists
  - Keep multiple linked lists of different size classes, or possibly for different types of objects

Keeping Track of Free Blocks

- Method 1: Implicit list using length—links all blocks

- Method 2: Explicit list among the free blocks using pointers

- Method 3: Segregated free list
  - Different free lists for different size classes

- Method 4: Blocks sorted by size
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

Segregated List (Seglist) Allocators

- Each size class of blocks has its own free list

- Often have separate classes for each small size

- For larger sizes: One class for each two-power size

Seglist Allocator

- Given an array of free lists, each one for some size class

- To allocate a block of size \( n \):
  - Search appropriate free list for block of size \( m > n \)
  - If an appropriate block is found:
    - Split block and place fragment on appropriate list (optional)
    - If no block is found, try next larger class
    - Repeat until block is found

- If no block is found:
  - Request additional heap memory from OS (using \texttt{sbrk()} )
  - Allocate block of \( n \) bytes from this new memory
  - Place remainder as a single free block in largest size class.
Seglist Allocator (cont.)

- To free a block:
  - Coalesce and place on appropriate list (optional)

- Advantages of seglist allocators
  - Higher throughput
    - Log time for power-of-two size classes
  - Better memory utilization
    - First-fit search of segregated free list approximates a best-fit search of entire heap.
    - Extreme case: Giving each block its own size class is equivalent to best-fit.