Dynamic Memory Allocation, Part 1

Computer Systems Organization (Spring 2016)
CSCI-UA 201, Section 2

Instructor: Joanna Klukowska

Slides adapted from
Randal E. Bryant and David R. O'Hallaron (CMU)
Mohamed Zahran (NYU)

Dynamic Memory Allocation

- Programmers use **dynamic memory allocators** (such as malloc) to acquire VM at run time.
  - For data structures whose size is only known at runtime.

- Dynamic memory allocators manage an area of process virtual memory known as the **heap**.

```
[Diagram showing stack and heap organizations]
```

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

```
#include <stdlib.h>

void *malloc(size_t size)
{
  int i, *p;
  if (size == 0) { return NULL; }
  else { return (void *) malloc(size); }
}

void free(void *p)
{
  if (p != NULL) { free(p); }
}
```

Other functions

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

malloc Example

```
#include <stdio.h>
#include <stdlib.h>

void free(int *p)
{
  if (p != NULL) { free(p); }
}
```

Determine the block size as a multiple of 8 bytes

- Allocate a block of n integers
- If allocated, initialize to zeros
- Deallocate block

Determine the block size as a multiple of 8 bytes
Assumptions Made in This Lecture

- Memory is word addressed.
- Words are int-sized.

Allocation Example

- \( p_1 = \text{malloc}(4) \)
- \( p_2 = \text{malloc}(5) \)
- \( p_3 = \text{malloc}(6) \)
- \( \text{free}(p_2) \)
- \( p_4 = \text{malloc}(2) \)

Constraints

- Applications
  - Can issue arbitrary sequence of \text{malloc} and \text{free} requests
  - \text{free} request must be to \text{malloc}’d block

- Allocators
  - Can’t control number or size of allocated blocks
  - Must respond immediately to \text{malloc} requests
    - i.e., can’t reorder or buffer requests
  - Must allocate blocks from free memory
    - i.e., can only place allocated blocks in free memory
  - Must align blocks so they satisfy all alignment requirements
    - 8-byte (x86) or 16-byte (x86-64) alignment on Linux boxes
  - Can manipulate and modify only free memory
  - Can’t move the allocated blocks once they are \text{malloc}’d
    - i.e., compaction is not allowed

Performance Goal: Throughput

- Given some sequence of \text{malloc} and \text{free} requests:
  - \( R_0, R_1, ..., R_n \)

- Goals: maximize throughput and peak memory utilization
  - These goals are often conflicting

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

Performance Goal: Peak Memory Utilization

- Given some sequence of \text{malloc} and \text{free} requests:
  - \( R_0, R_1, ..., R_n \)

- Def: Aggregate payload \( P_k \)
  - \text{malloc} (p) results in a block with a payload of \( p \) bytes
  - After request \( R_i \) has completed, the aggregate payload \( P_i \) 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 \text{abz}

- Def: Peak memory utilization after \( k+1 \) requests
  - \( U_k = \left( \max_{p \geq k} P_p \right) / H_k \)

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

- Caused by
  - Overhead of maintaining heap data structures
  - Padding for alignment purposes
  - Explicit policy decisions (i.e., to return a big block to satisfy a small request)
  - Depends only on the pattern of previous requests
    - Thus, easy to measure

Implementation Issues

- How do we know how much memory to free given just a pointer?

- How do we keep track of the free blocks?

- What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?

- How do we pick a block to use for allocation — many might fit?

- How do we reinsert freed block?

External Fragmentation

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

- p1 = malloc(4)

- p2 = malloc(5)

- p3 = malloc(6)

- free(p2)

- p4 = malloc(6)  
  **Oops! (what would happen now?)**

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

Knowing How Much to Free

Standard method

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

- p0 = malloc(4)

- free(p0)

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 binary tree (e.g., Red-Black tree) with pointers within each free block, and the length used as a key
Method 1: 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 a allocated/free flag
  - When reading size word, must mask off this bit

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 “spillovers” 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 unhelpful 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 improves memory utilization
  - 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

Implicit List: Freeing a Block

Simplest implementation:

- Need only clear the “allocated” flag
  - void free_block(ptr p) { p = p + 4; -2 }
- But can lead to “false fragmentation”

```
 free(p)
 ^
 malloc(5)  Oops!
```

Implicit List: Coalescing

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

- Coalescing with next block

```
 free(p)
 ^
 logically gone
```

- 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!

![Diagram of implicit list with boundary tags]

**Format of allocated and free blocks**
- Header
- Payload and padding
- Size: Total block size
- Payload: Application data (allocated blocks only)

---

Constant Time Coalescing

**Case 1**

```
<table>
<thead>
<tr>
<th>m1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>1</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td>n</td>
<td>0</td>
</tr>
<tr>
<td>m2</td>
<td>1</td>
</tr>
<tr>
<td>m2</td>
<td>1</td>
</tr>
</tbody>
</table>
```

**Case 2**

```
<table>
<thead>
<tr>
<th>m1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>1</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td>n</td>
<td>0</td>
</tr>
<tr>
<td>m2</td>
<td>0</td>
</tr>
<tr>
<td>m2</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**Case 3**

```
<table>
<thead>
<tr>
<th>m1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>0</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td>m2</td>
<td>1</td>
</tr>
<tr>
<td>m2</td>
<td>1</td>
</tr>
</tbody>
</table>
```

**Case 4**

```
<table>
<thead>
<tr>
<th>m1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>0</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td>m2</td>
<td>0</td>
</tr>
<tr>
<td>m2</td>
<td>0</td>
</tr>
</tbody>
</table>
```

---

Block being freed

```
<table>
<thead>
<tr>
<th>Allocated</th>
<th>Allocated</th>
<th>Free</th>
<th>Free</th>
</tr>
</thead>
</table>
```

---

m+n+1 = 0
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 throughout for less fragmentation
  - Interesting observation: segregated free lists (next lecture) 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:
  - Will 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