



Lecture 21: Dynamic memory allocation

Computer Architecture and
Systems Programming
(252-0061-00)

Timothy Roscoe
Herbstsemester 2012



Last time

What is a device?

- Registers, Interrupts, DMA
 - Example: NS16550 UART
- PCI (Peripheral Component Interconnect)

We'll return to this next week, and look at more complex devices.

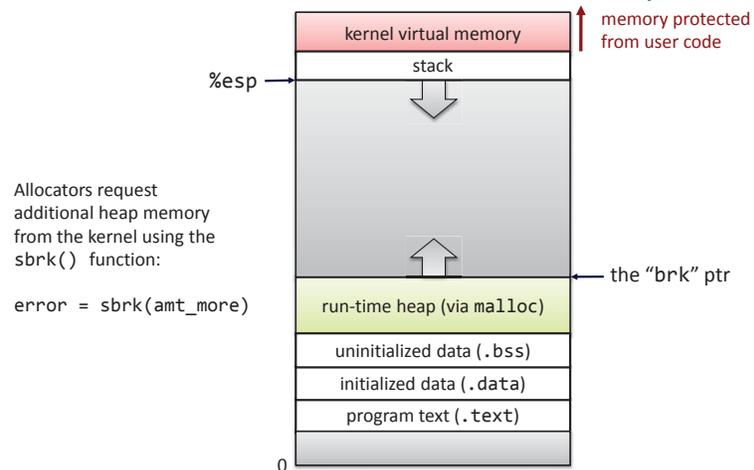


Today: Dynamic memory allocation

- How does malloc() work?
 - Principles apply to many other allocators
- Problem definition
 - What are we trying to achieve?
 - The issue of fragmentation
- Implementation 1: Implicit lists
 - How it works
 - Constant-time coalescing
 - Boundary tags
- Next time: better implementations



Process Memory Image



Why Dynamic memory allocation?

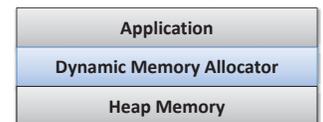
- It's very simple:

Sizes of needed data structures may only be known at runtime



Dynamic Memory Allocation

- Memory allocator?
 - VM hardware and kernel allocate pages
 - Application objects are typically smaller
 - Allocator manages objects within pages
- Explicit vs. Implicit Memory Allocator
 - **Explicit:** application allocates and frees space
 - In C: malloc() and free()
 - **Implicit:** application allocates, but does not free space
 - In Java, ML, Lisp: garbage collection
- Allocation
 - A memory allocator doles out memory blocks to application
 - A “block” is a contiguous range of bytes
 - of any size, in this context
- **Today:** simple explicit memory allocation



Malloc Package



- `#include <stdlib.h>`
- `void *malloc(size_t size)`
 - Successful:
 - Returns a pointer to a memory block of at least `size` bytes (typically) aligned to 8-byte boundary
 - If `size == 0`, returns `NULL`
 - Unsuccessful: returns `NULL (0)` and sets `errno`
- `void free(void *p)`
 - Returns the block pointed at by `p` to pool of available memory
 - `p` must come from a previous call to `malloc()` or `realloc()`
- `void *realloc(void *p, size_t size)`
 - Changes size of block `p` and returns pointer to new block
 - Contents of new block unchanged up to min of old and new size
 - Old block has been `free()`'d (logically, if new != old)

Malloc Example



```
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);
    }
    for (i=0; i<n; i++) p[i] = i;

    /* add m bytes to end of p block */
    if ((p = (int *)realloc(p, (n+m) * sizeof(int))) == NULL) {
        perror("realloc");
        exit(0);
    }
    for (i=n; i < n+m; i++) p[i] = i;

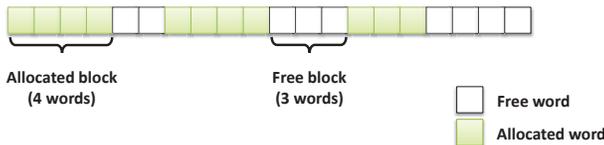
    /* print new array */
    for (i=0; i<n+m; i++)
        printf("%d\n", p[i]);

    free(p); /* return p to available memory pool */
}
```

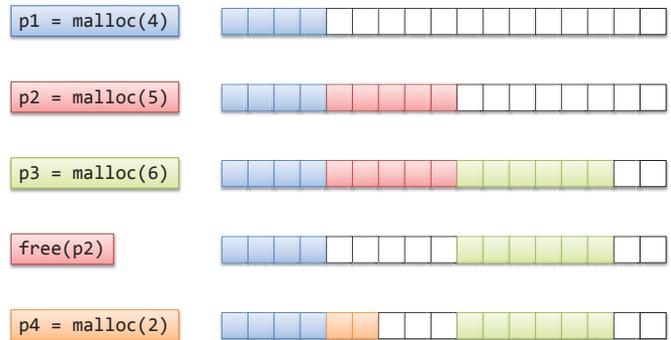
Assumptions we make (in this lecture)



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



Allocation Example



Constraints



- Applications
 - Can issue arbitrary sequence of `malloc()` and `free()` requests
 - `free()` requests must be to a `malloc()`'d block
- Allocators
 - Can't control number or size of allocated blocks
 - Must respond immediately to `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 alignment for GNU `malloc (libc malloc)` on Linux boxes
 - Can manipulate and modify only free memory
 - Can't move the allocated blocks once they are `malloc()`'d
 - *i.e.*, compaction is not allowed

Performance Goal: Throughput



- Given some sequence of `malloc` and `free` requests:
 - $R_0, R_1, \dots, R_k, \dots, R_{n-1}$
- Goals: maximize throughput and peak memory utilization
 - These goals are often conflicting
- Throughput:
 - Number of completed requests per unit time
 - Example:
 - 5,000 `malloc()` calls and 5,000 `free()` calls in 10 seconds
 - Throughput is 1,000 operations/second
 - **How to do `malloc()` and `free()` in $O(1)$? What's the problem?**

Performance Goal: Peak Memory Utilization



- Given some sequence of `malloc` and `free` requests:
 - $R_0, R_1, \dots, R_k, \dots, R_{n-1}$
- 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
 - all `malloc()`'d stuff minus all `free()`'d stuff
- Def:** Current heap size = H_k
 - Assume H_k is monotonically nondecreasing
 - reminder: it grows when allocator uses `sbrk()`
- Def:** Peak memory utilization after k requests
 - $U_k = (\max_{i < k} P_i) / H_k$

Fragmentation

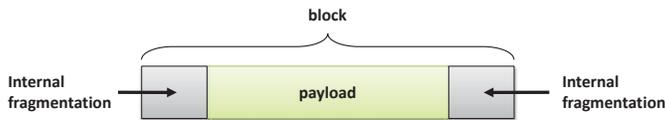


- Poor memory utilization caused by **fragmentation**
 - internal** fragmentation
 - external** fragmentation

Internal Fragmentation



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

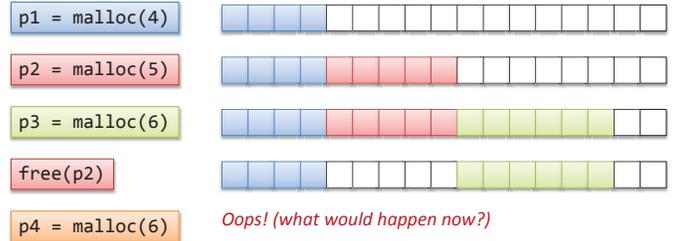


- Caused by
 - overhead of maintaining heap data structures
 - padding for alignment purposes
 - explicit policy decisions (e.g., to return a big block to satisfy a small request)
- Depends only on the pattern of **previous** requests
 - thus, easy to measure

External Fragmentation



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



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

Implementation Issues

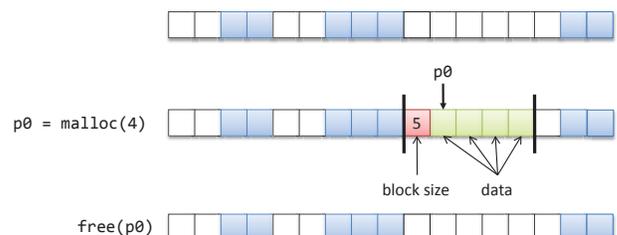


- How to know how much memory is being `free()`'d when it is given only a pointer (and no length)?
- How to keep track of the free blocks?
- What to do with extra space when allocating a block that is smaller than the free block it is placed in?
- How to pick a block to use for allocation—many might fit?
- How to reinsert a freed block into the heap?

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



Keeping Track of Free Blocks



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



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

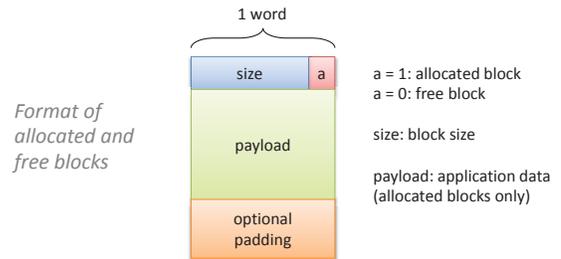


- Method 3: *Segregated free list*
 - Different free lists for different size classes
- Method 4: *Blocks sorted by size*
 - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

Implicit List



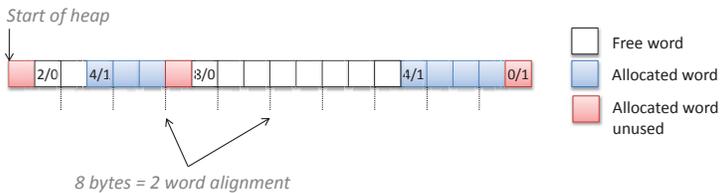
- For each block we need: length, is-allocated?
 - Could store this information in two words: wasteful!
- Standard trick
 - If blocks are aligned, some low-order address bits are always 0
 - Instead of storing an always-0 bit, use it as a allocated/free flag
 - When reading size word, must mask out this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



Example



Sequence of blocks in heap: 2/0, 4/1, 8/0, 4/1



- 8-byte alignment
 - May require initial unused word
 - Causes some internal fragmentation
- One word (0/1) to mark end of list
- Here: block size in words for simplicity

Implicit List: Finding a Free Block



- First fit:*
 - Search list from beginning, choose *first* free block that fits: (*Cost?*)

```

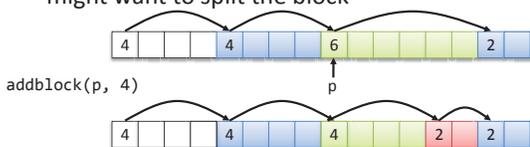
p = start;
while ((p < end) && // not passed end
      ((*p & 1) || // already allocated
       (*p <= len))) // too small
  p = p + (*p & -2); // goto next block (word addressed)
    
```

 - 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 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 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



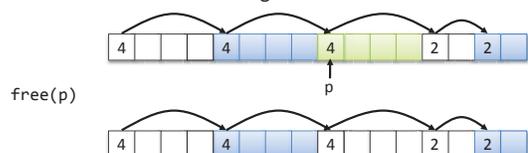
```

void addblock(ptr p, int len) {
  int newsize = ((len + 1) >> 1) << 1; // round up to even
  int oldsize = *p & -2; // mask out low bit
  *p = newsize | 1; // set new length
  if (newsize < oldsize)
    *(p+newsize) = oldsize - newsize; // set length in remaining
  // part of block
}
    
```

Implicit List: Freeing a Block



- Simplest implementation:
 - Need only clear the "allocated" flag
 - But can lead to "false fragmentation"



malloc(5) *Oops!*

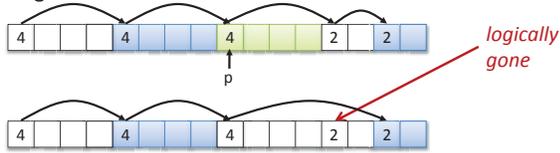
There is enough free space, but the allocator won't be able to find it

Implicit List: Coalescing



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

– Coalescing with next block:



```
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
}
```

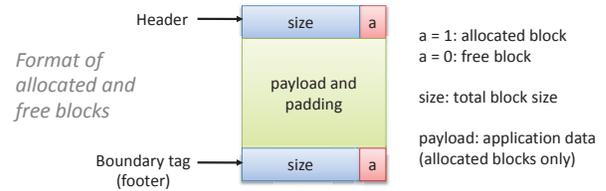
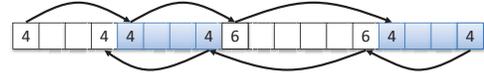
– But how do we coalesce with *previous* block?

Implicit List: Bidirectional Coalescing

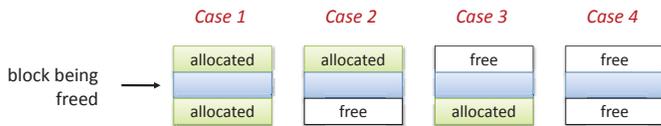


- Boundary tags* [Knuth73]

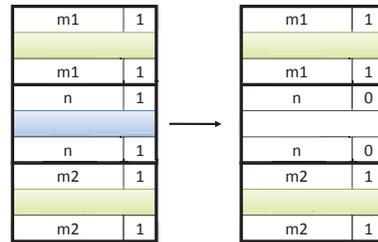
- Replicate size/allocated word at “bottom” (end) of free blocks
- Allows us to traverse the “list” backwards, but requires extra space
- Important and general technique!



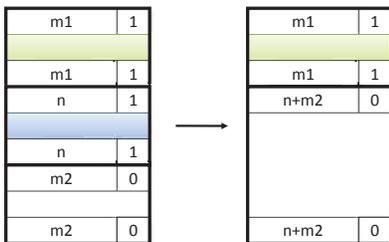
Constant Time Coalescing



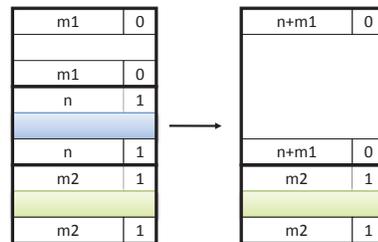
Constant Time Coalescing Case 1



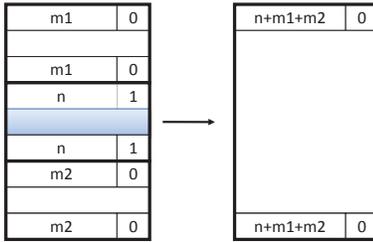
Constant Time Coalescing Case 2



Constant Time Coalescing Case 3



Constant Time Coalescing Case 4



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 (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