

15-213

"The course that gives CMU its Zip!"

## Dynamic Memory Allocation I

March 25, 2008

### Topics

- Simple explicit allocators
  - Data structures
  - Mechanisms
  - Policies

class18.ppt

## Harsh Reality

### Memory Matters

#### Memory is not unbounded

- It must be allocated and managed
- Many applications are memory dominated
  - Especially those based on complex, graph algorithms

#### Memory referencing bugs especially pernicious

- Effects are distant in both time and space

#### Memory performance is not uniform

- Cache and virtual memory effects can greatly affect program performance
- Adapting program to characteristics of memory system can lead to major speed improvements

2

15-213, S08

## Dynamic Memory Allocation

### Memory Allocator?

- VM hardware and kernel allocate *pages*
- Application objects are typically smaller
- Allocator manages objects within pages
  - 4K page can hold ~64 64-byte objects

Application
Dynamic Memory Allocator
Heap Memory

### Explicit vs. Implicit Memory Allocator

- Explicit: application allocates and frees space
  - E.g., `malloc()` and `free()` in C
- Implicit: application allocates, but does not free space
  - E.g. garbage collection in Java, ML or Lisp

### Allocation

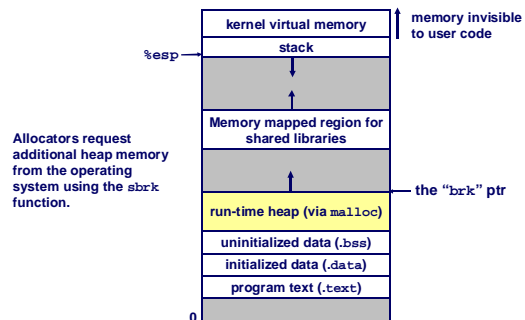
- In both cases the memory allocator provides an abstraction of memory as a set of blocks
- Doles out free memory blocks to application

Will discuss simple explicit memory allocation today

3

15-213, S08

## Process Memory Image



4

15-213, S08

## Malloc Package

```
#include <stdlib.h>

void *malloc(size_t size)

• If successful:
  • Returns a pointer to a memory block of at least size bytes, (typically)
    aligned to 8-byte boundary.
  • If size == 0, returns NULL
• If 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)
```

5

15-213, S'08

## 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 */
}
```

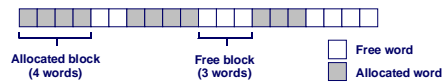
6

15-213, S'08

## Assumptions

### Assumptions made in this lecture

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



7

15-213, S'08

## Allocation Examples



8

15-213, S'08

## Constraints

### Applications:

- Can issue arbitrary sequence of allocation and free requests
- Free requests must correspond to an allocated block

### Allocators

- Can't control number or size of allocated blocks
- Must respond immediately to all allocation 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 allocated
  - i.e., compaction is not allowed

9

15-213, S'08

## Performance Goals: Throughput

Given some sequence of `malloc` and `free` requests:

- $R_0, R_1, \dots, R_k, \dots, R_{n-1}$

Want to 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.

10

15-213, S'08

## Performance Goals: 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.

**Def: Current heap size is denoted by  $H_k$**

- Assume that  $H_k$  is monotonically nondecreasing

**Def: Peak memory utilization:**

- After  $k$  requests, *peak memory utilization* is:
  - $U_k = (\max_{i \leq k} P_i) / H_k$

11

15-213, S'08

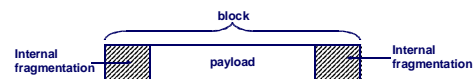
## Internal Fragmentation

Poor memory utilization caused by *fragmentation*.

- Comes in two forms: *internal* and *external* fragmentation

### Internal fragmentation

- For some block, *internal fragmentation* is the difference between the block size and the payload size.



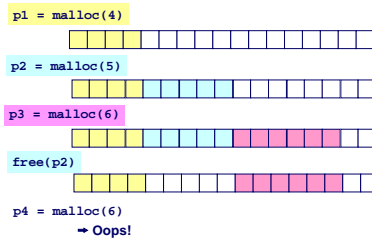
- Caused by overhead of maintaining heap data structures, padding for alignment purposes, or explicit policy decisions (e.g., to return a big block to satisfy a small request).
- Depends only on the pattern of *previous* requests, and thus is easy to measure.

12

15-213, S'08

## External Fragmentation

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



External fragmentation depends on the pattern of *future* requests, and thus is difficult to measure.

13

15-213, S'08

## Implementation Issues

How do we know how much memory is being freed when we are given only a pointer (no length)?

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 a freed block into the heap?

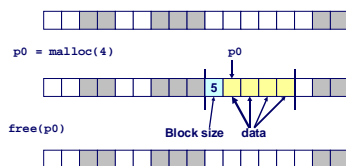
14

15-213, S'08

## 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



15

15-213, S'08

## Keeping Track of Free Blocks

### Method 1: Implicit list using lengths -- links all blocks



### Method 2: Explicit list among the free blocks using pointers within the free blocks



### 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

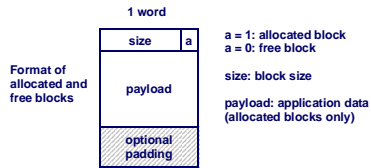
16

15-213, S'08

## Method 1: 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



17

15-213, S'08

## Implicit List: Finding a Free Block

**First fit:**

- Search list from beginning, choose first free block that fits

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

- 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

18

15-213, S'08

## Bit Fields

How to represent the Header:

Masks and bitwise operators

```
#define SIZEMASK      (~0x7)
#define PACK(size, alloc) ((size) | (alloc))
#define GET_SIZE(p)    ((p)->size & SIZEMASK)
```

Bit Fields

```
struct {
    unsigned allocated:1;
    unsigned size:31;
} Header;
```

Check your K&R: structures are not necessarily packed

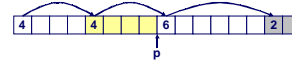
19

15-213, S'08

## 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; // add 1 and round up
    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
}
```

addblock(p, 2)



20

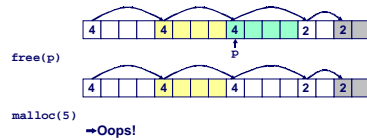
15-213, S'08

## Implicit List: Freeing a Block

### Simplest implementation:

- Need only clear the "allocated" flag  

```
void free_block(ptr p) { *p = *p & ~2; }
```
- But can lead to "false fragmentation"



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

21

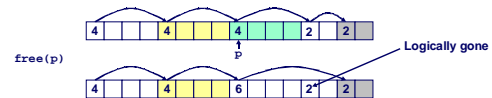
15-213, S'08

## Implicit List: Coalescing

Join (**coalesce**) with next and/or 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) // add to this block if not allocated
        *p = *p + *next;
}
```



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

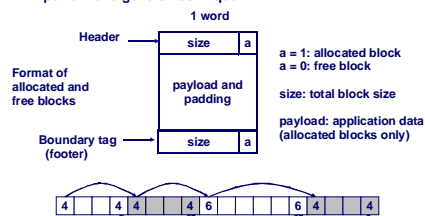
22

15-213, S'08

## Implicit List: Bidirectional Coalescing

### Boundary tags [Knuth73]

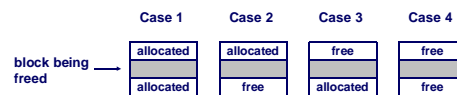
- Replicate size/allocated word at "bottom" (end) of free blocks
- Allows us to traverse the "list" backwards, but requires extra space
- Important and general technique!



23

15-213, S'08

## Constant Time Coalescing



24

15-213, S'08

### Constant Time Coalescing (Case 1)

m1	1
m1	1
n	1
n	1
m2	1
m2	1



m1	1
m1	1
n	0
n	0
m2	1
m2	1

25

15-213, S'08

### Constant Time Coalescing (Case 2)

m1	1
m1	1
n	1
n	1
m2	0
m2	0



m1	1
m1	1
n+m2	0
n+m2	0

26

15-213, S'08

### Constant Time Coalescing (Case 3)

m1	0
m1	0
n	1
n	1
m2	1
m2	1



n+m1	0
n+m1	0
m2	1
m2	1

27

15-213, S'08

### Constant Time Coalescing (Case 4)

m1	0
m1	0
n	1
n	1
m2	0
m2	0



n+m1+m2	0
n+m1+m2	0

28

15-213, S'08

## 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. e.g.,
  - Coalesce as you scan the free list for `malloc()`.
  - Coalesce when the amount of external fragmentation reaches some threshold.

29

15-213, S08

## 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.

30

15-213, S08