

15-213

Dynamic Memory Allocation

April 9, 2002

Topics

- Simple explicit allocators
- Data structures
- Mechanisms
- Policies
- Reading: 10.9
- Problems: 10.15 and 10.16

class21.ppt

Harsh Reality #3

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

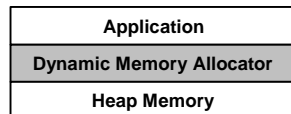
- Cache and virtual memory effects can greatly affect program performance
- Adapting program to characteristics of memory system can lead to major speed improvements

class21.ppt

– 2 –

15-213 S'02(Based on CS 213 F'01)

Dynamic Memory Allocation



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

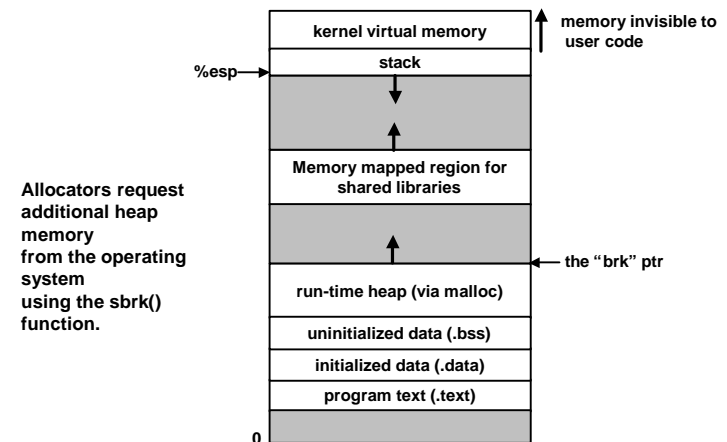
Will discuss simple explicit memory allocation today

class21.ppt

– 3 –

15-213 S'02(Based on CS 213 F'01)

Process memory image



Allocators request additional heap memory from the operating system using the sbrk() function.

class21.ppt

– 4 –

15-213 S'02(Based on CS 213 F'01)

Malloc package

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

```
void *malloc(size_t size)
```

- **if successful:**
 - returns a pointer to a memory block of at least `size` bytes, aligned to 8-byte boundary.
 - if `size==0`, returns NULL
- **if unsuccessful:** returns NULL

```
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 ptr to new block.
- contents of new block unchanged up to min of old and new size.

class21.ppt

– 5 –

15-213 S'02(Based on CS 213 F'01)

Malloc example

```
void foo(int n, int m) {
    int i, *p;

    /* allocate a block of n ints */
    if ((p = (int *) malloc(n * sizeof(int))) == 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 */
}
```

class21.ppt

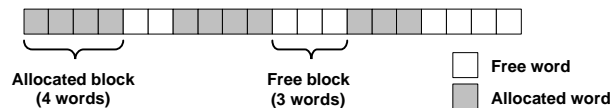
– 6 –

15-213 S'02(Based on CS 213 F'01)

Assumptions

Assumptions made in this lecture

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

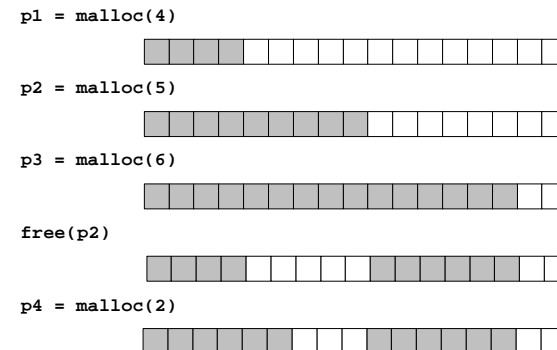


class21.ppt

– 7 –

15-213 S'02(Based on CS 213 F'01)

Allocation examples



class21.ppt

– 8 –

15-213 S'02(Based on CS 213 F'01)

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
 - usually 8 byte alignment
- Can only manipulate and modify free memory
- Can't move the allocated blocks once they are allocated
 - i.e., compaction is not allowed

class21.ppt

– 9 –

15-213 S'02(Based on CS 213 F'01)

Goals of good malloc/free

Primary goals

- Good time performance for `malloc` and `free`
 - Ideally should take constant time (not always possible)
 - Should certainly not take linear time in the number of blocks
- Good space utilization
 - User allocated structures should be large fraction of the heap.
 - want to minimize “fragmentation”.

Some other goals

- Good locality properties
 - structures allocated close in time should be close in space
 - “similar” objects should be allocated close in space
- Robust
 - can check that `free(p1)` is on a valid allocated object `p1`
 - can check that memory references are to allocated space

class21.ppt

– 10 –

15-213 S'02(Based on CS 213 F'01)

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.

class21.ppt

– 11 –

15-213 S'02(Based on CS 213 F'01)

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

- Note that H_k is monotonically nondecreasing

Def: peak memory utilization:

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

class21.ppt

– 12 –

15-213 S'02(Based on CS 213 F'01)

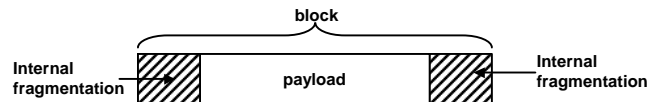
Internal Fragmentation

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., not to split the block).
- Depends only on the pattern of previous requests, and thus is easy to measure.

class21.ppt

- 13 -

15-213 S'02(Based on CS 213 F'01)

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!
```

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

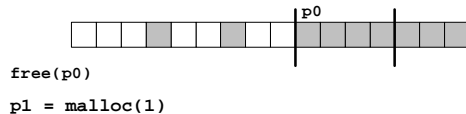
class21.ppt

- 14 -

15-213 S'02(Based on CS 213 F'01)

Implementation issues

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



class21.ppt

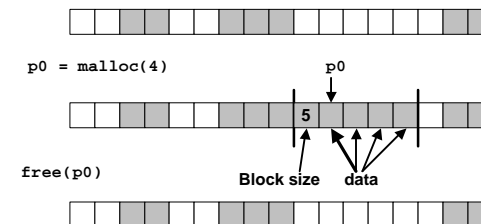
- 15 -

15-213 S'02(Based on CS 213 F'01)

Knowing how much to free

Standard method

- keep the length of a structure in the word preceding the structure
 - This word is often called the *header field* or *header*
- requires an extra word for every allocated structure



class21.ppt

- 16 -

15-213 S'02(Based on CS 213 F'01)

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 lists
 - 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

class21.ppt

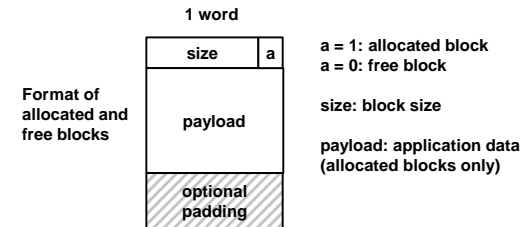
- 17 -

15-213 S'02(Based on CS 213 F'01)

Method 1: implicit list

Need to identify whether each block is free or allocated

- Can use extra bit
- Bit can be put in the same word as the size if block sizes are always multiples of two (mask out low order bit when reading size).



class21.ppt

- 18 -

15-213 S'02(Based on CS 213 F'01)

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
```

- 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 from location of end of previous search
- Research suggests that fragmentation is worse

Best fit:

- Search the list, choose the free block with the closest size that fits
- Keeps fragments small --- usually helps fragmentation
- Will typically run slower than first-fit

class21.ppt

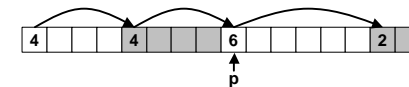
- 19 -

15-213 S'02(Based on CS 213 F'01)

Implicit list: allocating in a 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);
```



class21.ppt

- 20 -

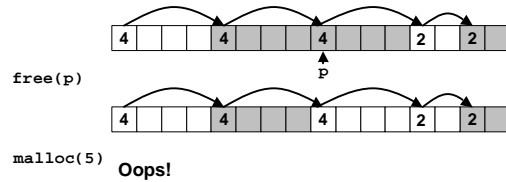
15-213 S'02(Based on CS 213 F'01)

Implicit list: freeing a block

Simplest implementation:

- Only need to clear 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

class21.ppt

- 21 -

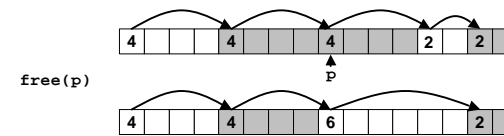
15-213 S'02(Based on CS 213 F'01)

Implicit list: coalescing

Join with next and/or previous block 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?

class21.ppt

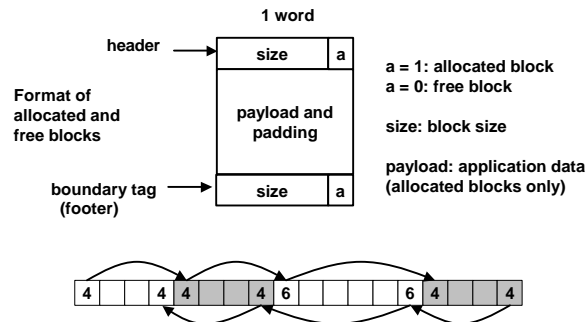
- 22 -

15-213 S'02(Based on CS 213 F'01)

Implicit list: bidirectional

Boundary tags [Knuth73]

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

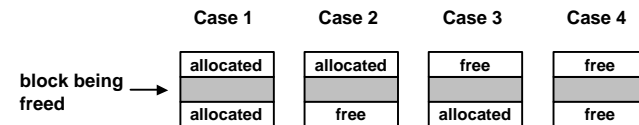


class21.ppt

- 23 -

15-213 S'02(Based on CS 213 F'01)

Constant time coalescing

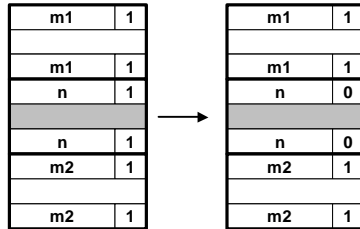


class21.ppt

- 24 -

15-213 S'02(Based on CS 213 F'01)

Constant time coalescing (case 1)

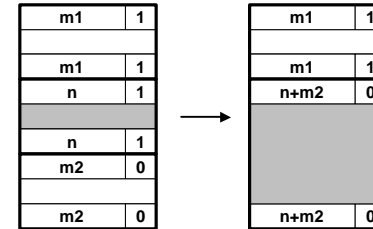


class21.ppt

- 25 -

15-213 S'02(Based on CS 213 F'01)

Constant time coalescing (case 2)

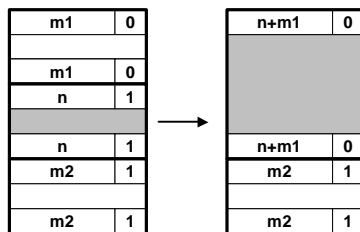


class21.ppt

- 26 -

15-213 S'02(Based on CS 213 F'01)

Constant time coalescing (case 3)

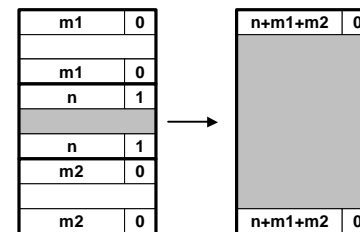


class21.ppt

- 27 -

15-213 S'02(Based on CS 213 F'01)

Constant time coalescing (case 4)



class21.ppt

- 28 -

15-213 S'02(Based on CS 213 F'01)

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 the 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 adjacent blocks 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.

class21.ppt

– 29 –

15-213 S'02(Based on CS 213 F'01)

Implicit lists: Summary

- **Implementation:** very simple
- **Allocate:** linear time worst case
- **Free:** 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 allocate. Used in many special purpose applications.

However, the concepts of splitting and boundary tag coalescing are general to *all* allocators.

class21.ppt

– 30 –

15-213 S'02(Based on CS 213 F'01)