15-213

“The course that gives CMU its Zip!”

Memory Management I: Dynamic Storage Allocation March 2, 2000

Topics

- Explicit memory allocation
- Data structures
- Mechanisms

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

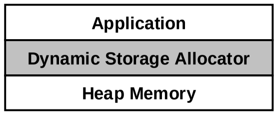
Memory referencing bugs especially pernicious

- 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

Dynamic Storage Allocation



Explicit vs. Implicit Storage Allocator

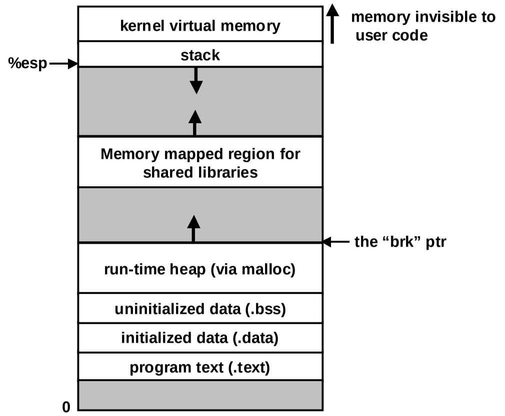
- Explicit: application allocates and frees space
  - E.g., malloc and free in C
- Implicit: application allocates, but does not free space
  - E.g. garbage collection in Java, ML or Lisp

Allocation

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

Will discuss explicit storage allocation today

Process memory image


kernel virtual memory
memory invisible to user code
stack
%esp
Memory mapped region for shared libraries
the “brk” ptr
run-time heap (via malloc)
uninitialized data (.bss)
initialized data (.data)
program text (.text)
0

Malloc package

`void *malloc(int size)`

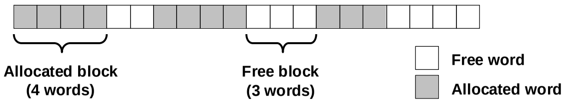
- **if successful:**
  - returns a pointer to a memory block of at least **`size`** bytes
  - if `size==0`, returns NULL
- **if unsuccessful:** returns NULL

`void free(void *p)`

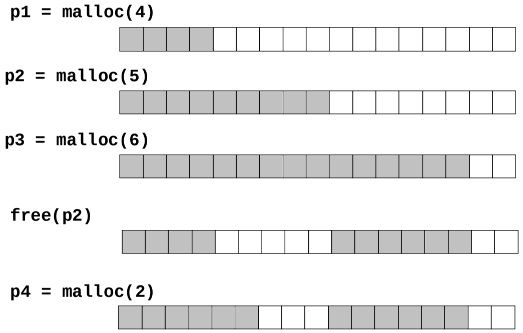
- returns the block pointed at by **`p`** to pool of available memory
- **`p`** must come from a previous call to **`malloc`**().

Assumptions made in this lecture

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



Allocation example



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

Goals of good malloc/free

Primary goals

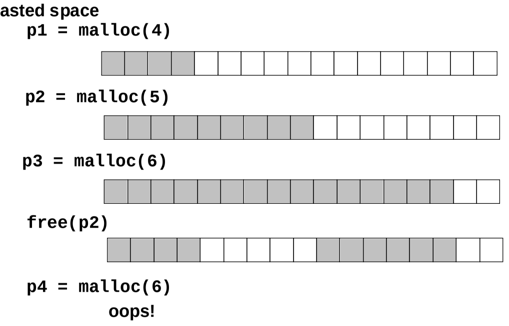
- **Good time performance for `malloc` and `free`**
  - Ideally should take constant time (not always possible)
  - Should certainly not take linear time in the number of blocks
- **Good space usage**
  - User allocated structures should be large fraction of operating-system allocated pages
  - Need to avoid 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

Fragmentation

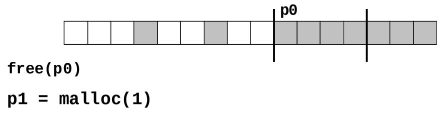
Tendency for free blocks to become smaller over time leading to wasted space



No general solution assuming we cannot move blocks
We will consider several heuristics

Implementation issues

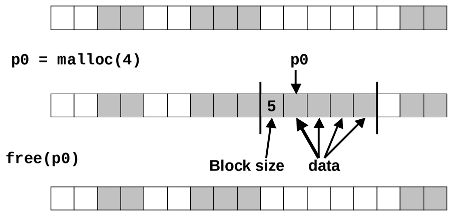
- How do we know how much memory to free just given a pointer?
- How do we keep track of the free blocks?
- What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?
- How do we pick a block to use for allocation -- many might fit?
- How do we reinsert freed block?



Knowing how much to free

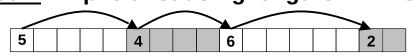
Standard method

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

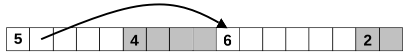


Keeping track of free blocks

- ***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

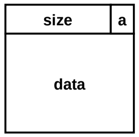
Method 1: implicit list

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).**

1 word

Format of allocated and free blocks



a = 1: allocated block
a = 0: free block

size: block size

data: application data (allocated blocks only)

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
- Does a better job of spreading out the free blocks

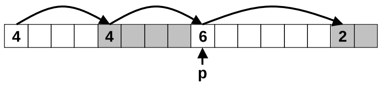
Best fit:

- 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

Implicit list: allocating in a free block

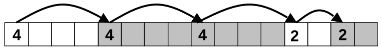
Allocating in a free block - *splitting*

- **Since allocated space might be smaller than free space, we need to split the block**



```
void addblock(ptr p, int l) {
  int newsize = ((l + 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
```

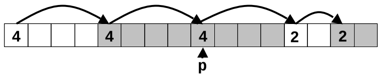
```
addblock(p,2);
```



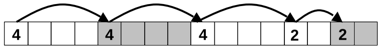
Implicit list: freeing a block

Simplest implementation:

- **Only need to clear allocated flag**
  `void free_block(ptr p) { *p= *p & -2}`
- **But can lead to “false fragmentation”**



`free(p)`



`malloc(5)` **Oops!**

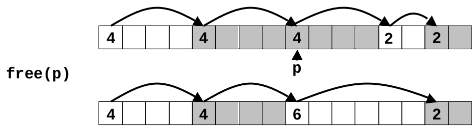
There is enough free space, but the allocator won’t be able to find it

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
```

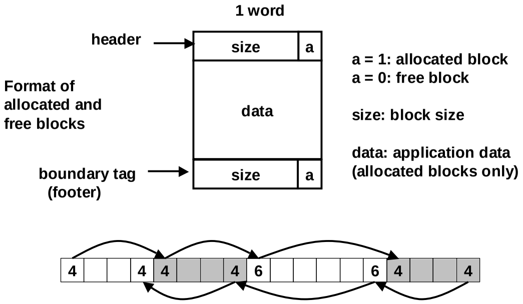


- But how do we coalesce with previous block?

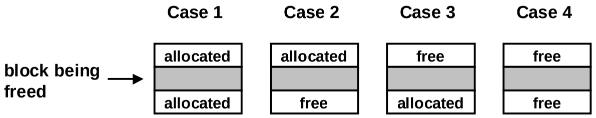
Implicit list: bidirectional

Boundary tags [Knuth73]

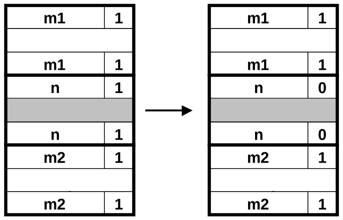
- replicate size/allocated word at bottom of free blocks
- Allows us to traverse the “list” backwards, but requires extra space



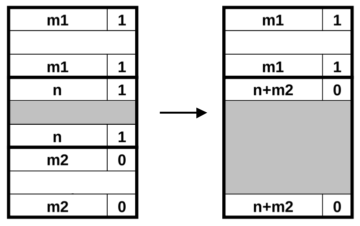
Constant time coalescing



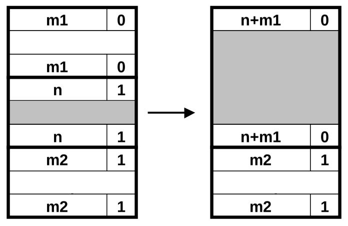
Constant time coalescing (case 1)



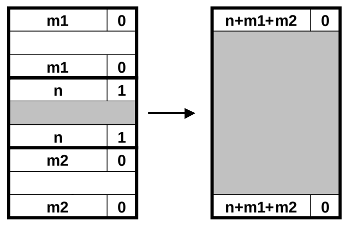
Constant time coalescing (case 2)



Constant time coalescing (case 3)



Constant time coalescing (case 4)



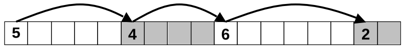
Implicit lists: Summary

- **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

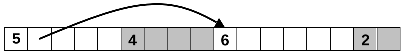
Not used in practice for malloc/free because of linear time allocate. Used in many special purpose applications.

Keeping track of free blocks

- ***Method 1***: implicit list using lengths -- links all blocks

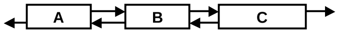


- ***Method 2***: explicit list among the free blocks using pointers within the free blocks



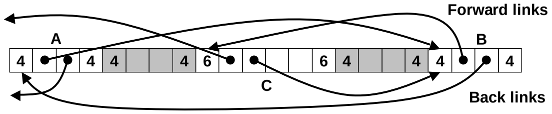
- ***Method 3***: segregated free lists
  - Different free lists for different size classes
- ***Method 4***: blocks sorted by size
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

Linked list of free blocks



Use data space for link pointers

- Typically doubly linked
- Still need header and footer for coalescing



- It is important to realize that links are not necessarily in the same order as the blocks

Linked list of free blocks

Allocation

- Splice block out of the free list
- Split the block
- If remaining space, put space back onto the free list

Free

- Determine if coalescing with neighboring block
  - If not coalescing, add block to free list
  - If coalescing with next block, need to splice next block out of the free list, and add self into it
  - If coalescing with previous block, only need to modify lengths of previous block
  - If coalescing with both previous and next, then need to splice the next block out of the free list (but not add self)

Linked list of free blocks

Comparison to implicit list:

- Allocate is linear time in number of free blocks instead of total blocks -- much faster allocates 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 (4 words needed for each block)

Main 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

For more information

D. Knuth, “The Art of Computer Programming, Second Edition”, Addison Wesley, 1973

- the classic reference on dynamic storage allocation

Wilson et al, “Dynamic Storage Allocation: A Survey and Critical Review”, Proc. 1995 Int’l Workshop on Memory Management, Kinross, Scotland, Sept, 1995.

- comprehensive survey
- $classdir/doc/dsa.ps