CS 33

Storage Allocation

All of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook "Computer Systems: A Programmer's Perspective," 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O'Hallaron in Fall 2010. These slides are indicated "Supplied by CMU" in the notes section of the slides.

Memory-Related Perils and Pitfalls

- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks
- Failing to free blocks

Supplied by CMU.

Dereferencing Bad Pointers

- The classic `scanf` bug

```
int val;
...
scanf("%d", val);
```


Supplied by CMU.

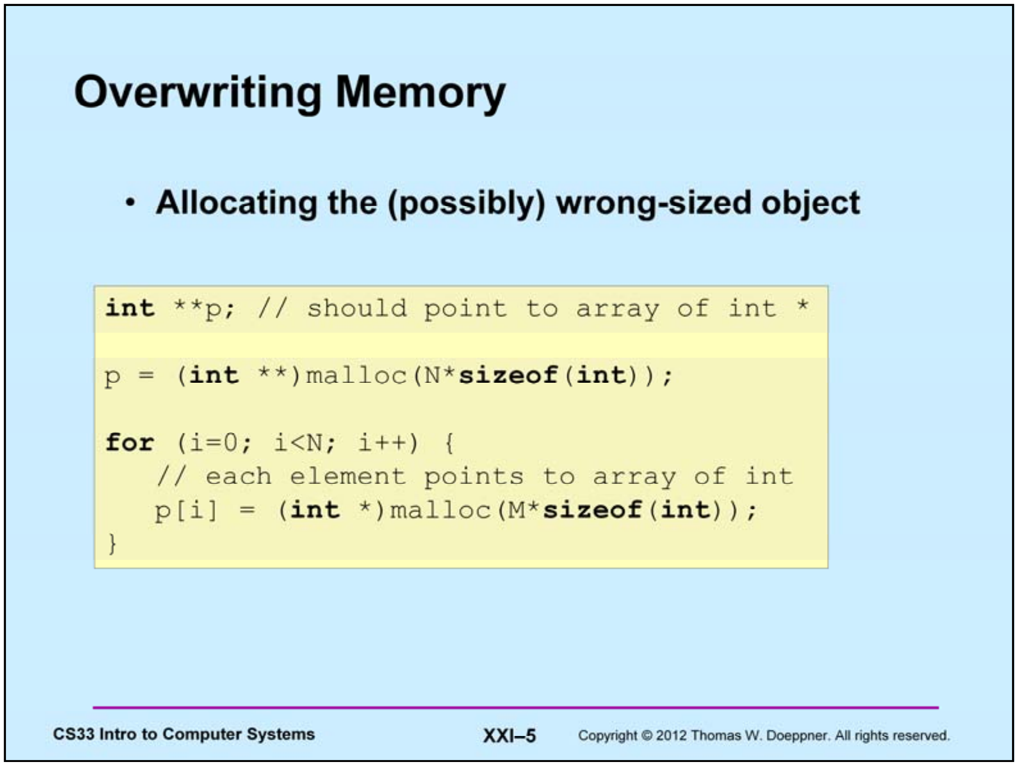
Reading Uninitialized Memory

- Assuming that heap data is initialized to zero

```
/* return y = Ax */
int *matvec(int **A, int *x) {
   int *y = malloc(N*sizeof(int));
   int i, j;

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         y[i] += A[i][j]*x[j];
   return y;
}
```


Supplied by CMU.



Supplied by CMU.

The problem here is that the storage allocated for p is of size N*sizeof(int), when it should be N*sizeof(int *) — on a 64-bit machine, p won't have been assigned enough storage.

Overwriting Memory

- Off-by-one error

```
int **p;

p = malloc(N*sizeof(int *));

for (i=0; i<=N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```


Supplied by CMU.

Overwriting Memory

- **Not checking the max string size**

```
char s[8];
int i;

gets(s);  /* reads "123456789" from stdin */
```

- **Basis for classic buffer overflow attacks**

Supplied by CMU.

Overwriting Memory

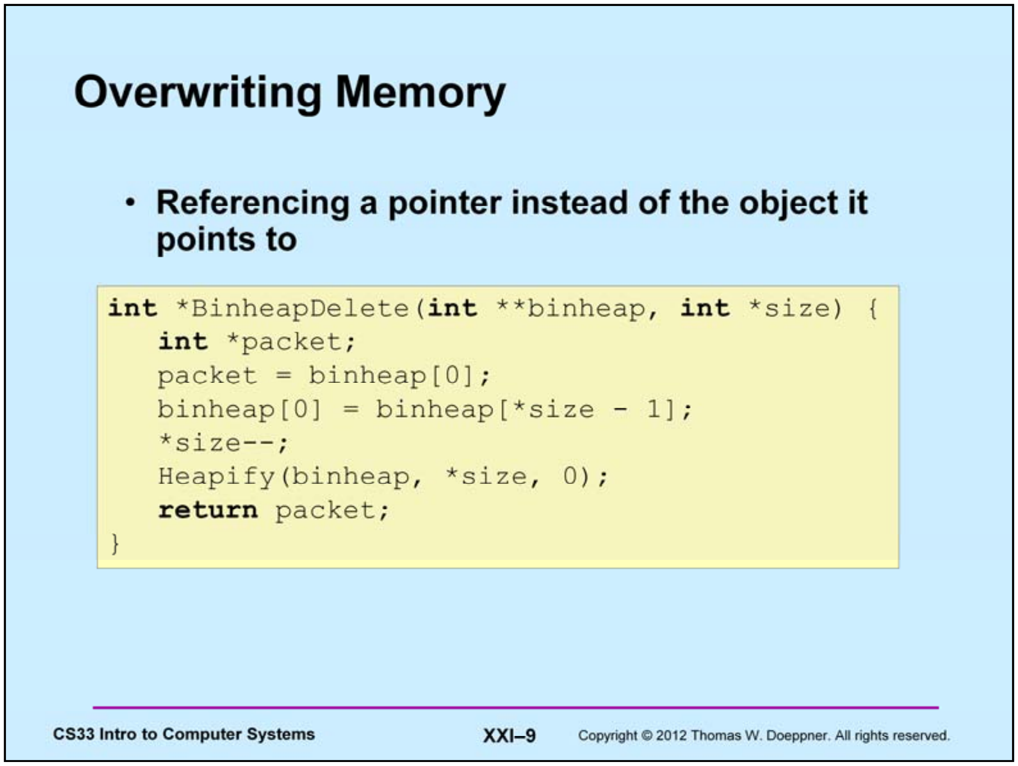
- **Misunderstanding pointer arithmetic**

```
int *search(int *p, int val) {

   while (*p && *p != val)
      p += sizeof(int);

   return p;
}
```


Supplied by CMU.



Supplied by CMU.

It should be (*size)--.

Referencing Nonexistent Variables

- Forgetting that local variables disappear when a function returns

```
int *foo () {
   int val;

   return &val;
}
```


Supplied by CMU.

Freeing Blocks Multiple Times

- **Nasty!**

```
x = (int *)malloc(N*sizeof(int));
        <manipulate x>
free(x);

y = (int *)malloc(M*sizeof(int));
        <manipulate y>
free(x);
```


Supplied by CMU.

Referencing Freed Blocks

- **Evil!**

```
x = (int *)malloc(N*sizeof(int));
  <manipulate x>
free(x);
   ...
y = (int *)malloc(M*sizeof(int));
for (i=0; i<M; i++)
   y[i] = x[i]++;
```


Supplied by CMU.

Failing to Free Blocks (Memory Leaks)

- **Slow, long-term killer!**

```
foo() {
    int *x = malloc(N*sizeof(int));
    ...
    return;
}
```


Supplied by CMU.

Failing to Free Blocks (Memory Leaks)

- **Freeing only part of a data structure**

```
struct list {
   int val;
   struct list *next;
};

foo() {
   struct list *head = malloc(sizeof(struct list));
   head->val = 0;
   head->next = NULL;
   <create and manipulate the rest of the list>
    ...
   free(head);
   return;
}
```


Supplied by CMU.

Dealing With Memory Bugs

- **Conventional debugger (`gdb`)**
  - good for finding bad pointer dereferences
  - hard to detect the other memory bugs
- **Debugging `malloc` (UToronto CSRI `malloc`)**
  - wrapper around conventional `malloc`
  - detects memory bugs at `malloc` and `free` boundaries
    - » memory overwrites that corrupt heap structures
    - » some instances of freeing blocks multiple times
    - » memory leaks
  - cannot detect all memory bugs
    - » overwrites into the middle of allocated blocks
    - » freeing block twice that has been reallocated in the interim
    - » referencing freed blocks

Supplied by CMU.

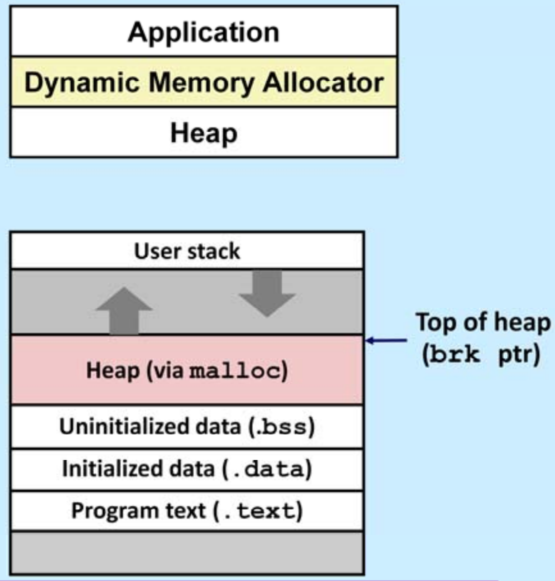
Dealing With Memory Bugs (cont.)

- **Some malloc implementations contain checking code**
  - **Linux glibc malloc:** `setenv MALLOC_CHECK_ 2`
  - **FreeBSD:** `setenv MALLOC_OPTIONS AJR`
- **Binary translator: valgrind (Linux), Purify**
  - **powerful debugging and analysis technique**
  - **rewrites text section of executable object file**
  - **can detect all errors as debugging** `malloc`
  - **can also check each individual reference at runtime**
    - » **bad pointers**
    - » **overwriting**
    - » **referencing outside of allocated block**

Supplied by CMU.

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***

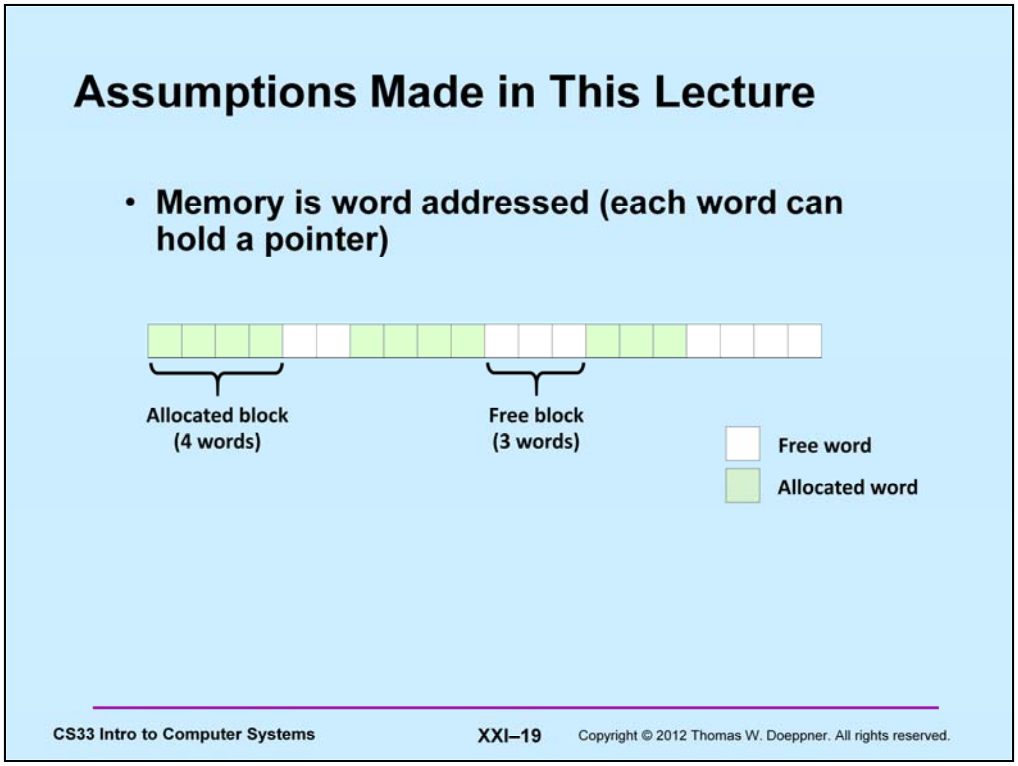


Supplied by CMU.

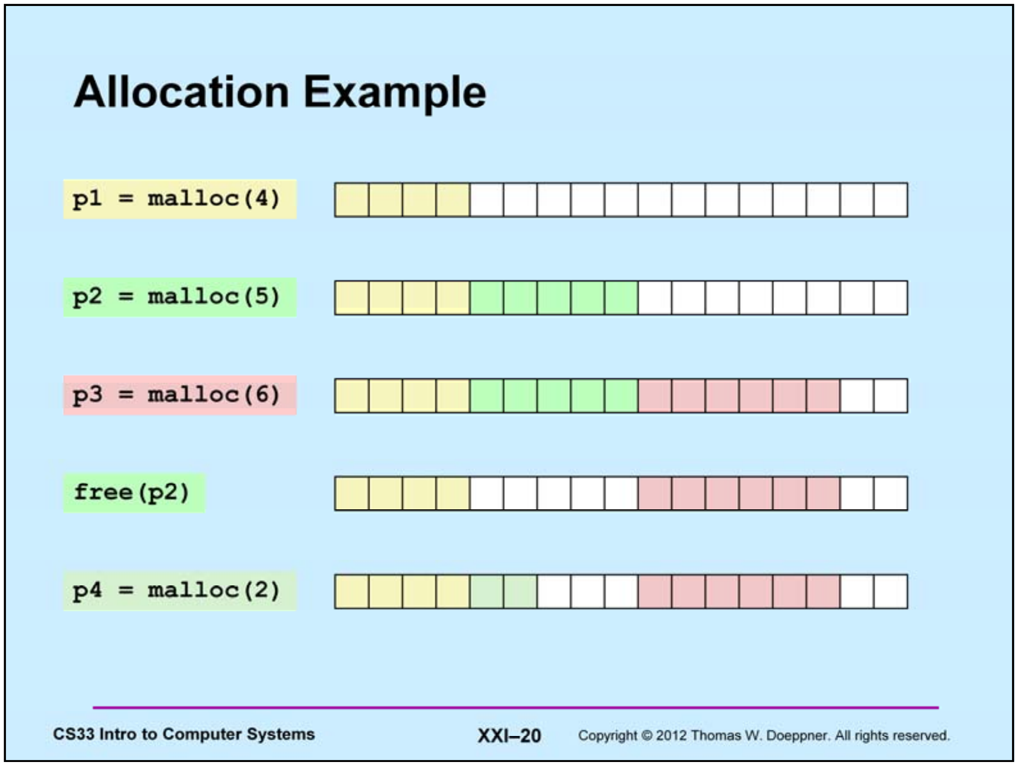
Dynamic Memory Allocation

- 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

Supplied by CMU.



Supplied by CMU.



Supplied by CMU.

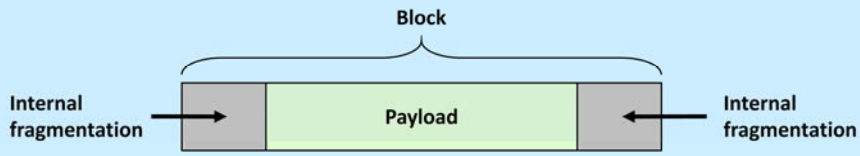
Constraints

- **Applications**
  - can issue arbitrary sequence of `malloc` and `free` requests
  - `free` request 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

Supplied by CMU.

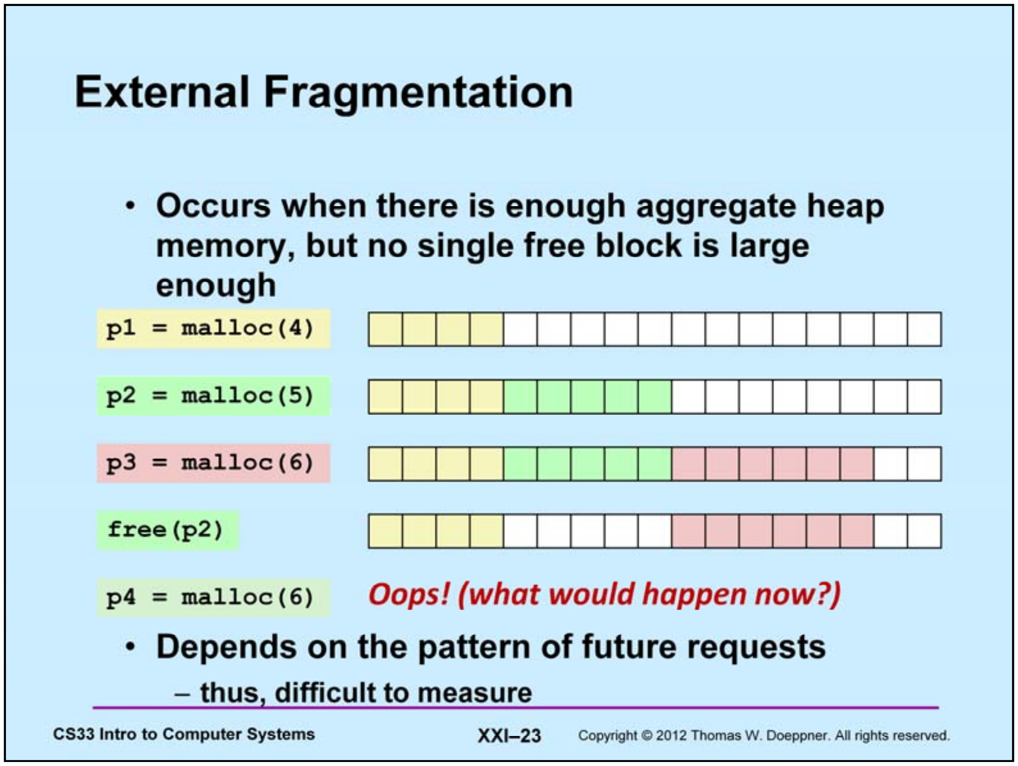
Internal Fragmentation

- 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

Supplied by CMU.



Supplied by CMU.

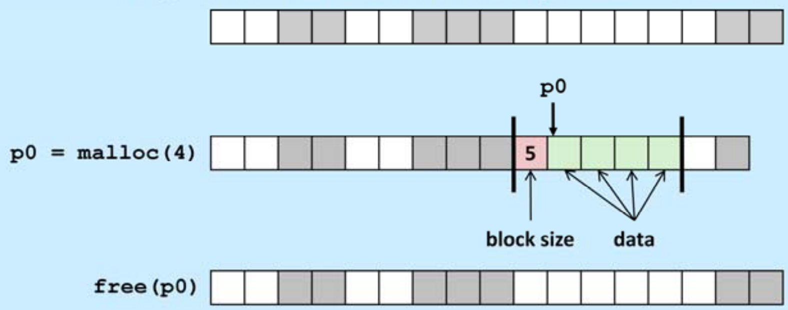
Implementation Issues

- 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?

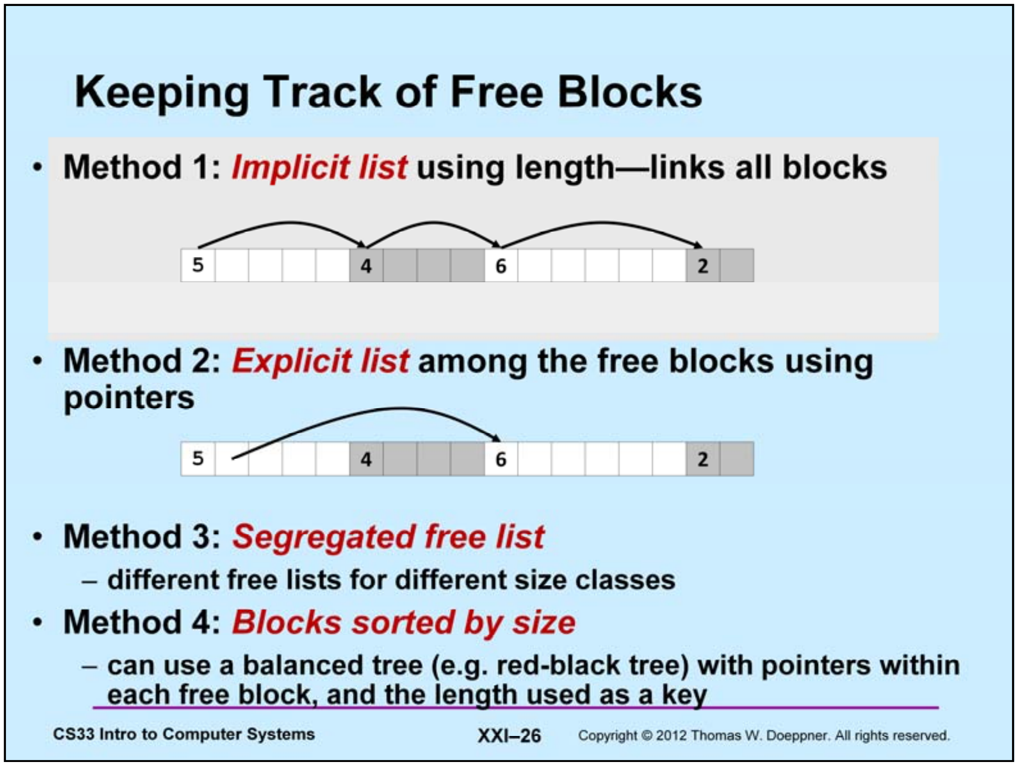
Supplied by CMU.

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



Supplied by CMU.

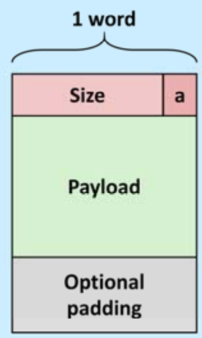


Supplied by CMU.

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, mask out this bit

Format of allocated and free blocks

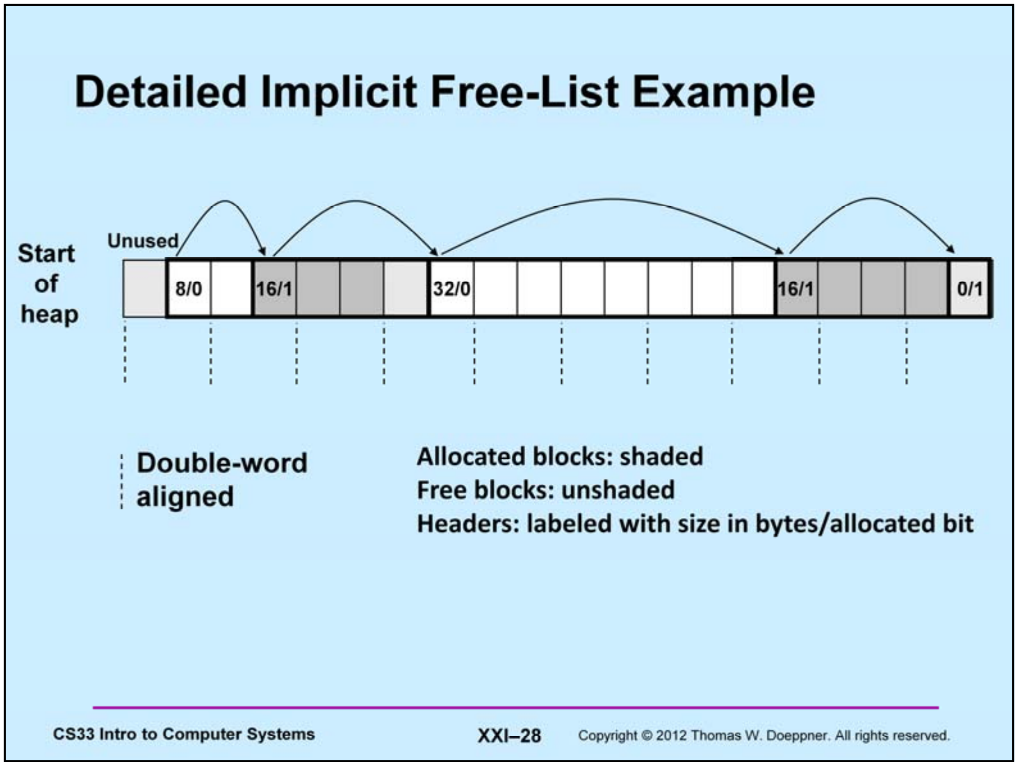


a = 1: Allocated block
a = 0: Free block

Size: block size

Payload: application data (allocated blocks only)

Supplied by CMU.



Supplied by CMU.

Implicit List: Finding a Free Block

- ***First fit:***
  - search list from beginning, choose ***first*** free block that fits:

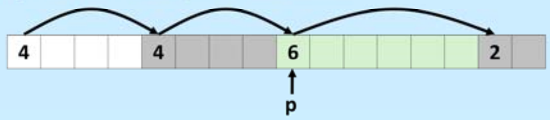
```
p = start;
while ((p < end) &&       // not passed end
       ((*p & 1) ||       // already allocated
       (*p <= len)))      // too small
  p = p + (*p & -2);      // goto next block (word addressed)
```

  - 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

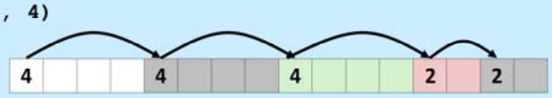
Supplied by CMU.

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**

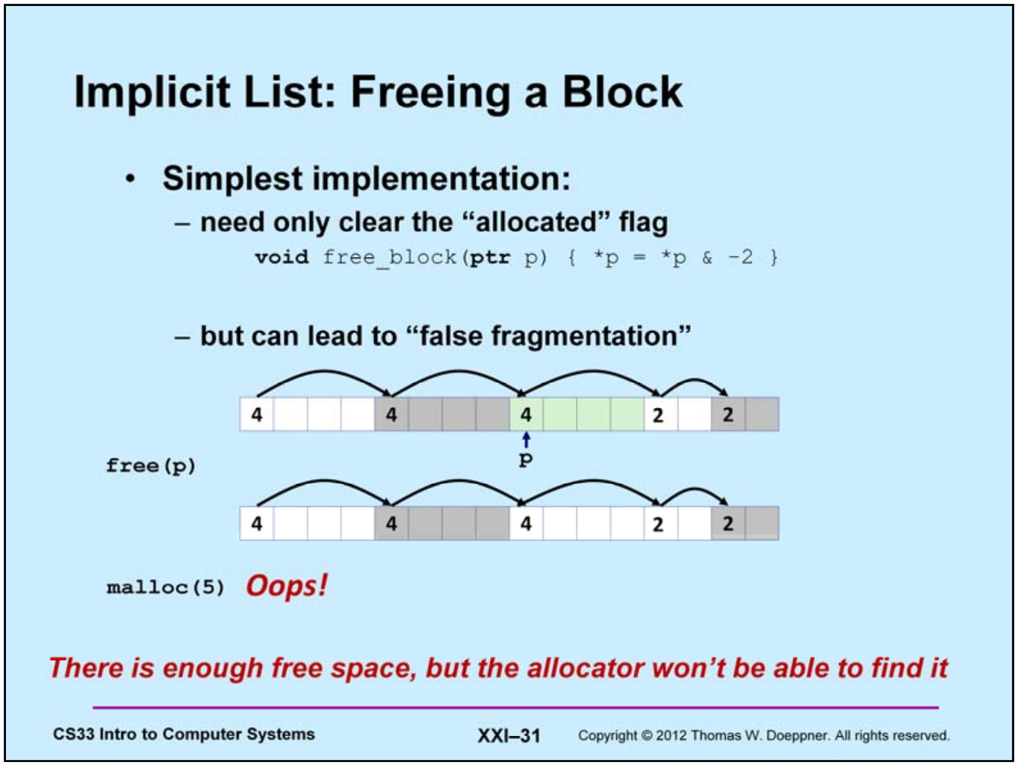


addblock(p, 4)




```
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
```

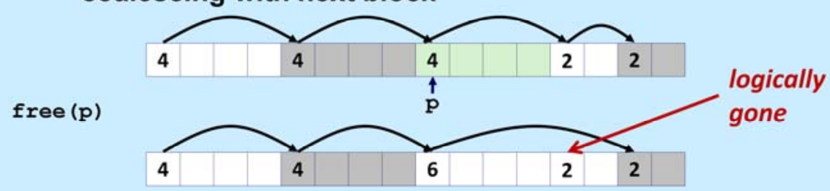

Supplied by CMU.



Supplied by CMU.

Implicit List: Coalescing

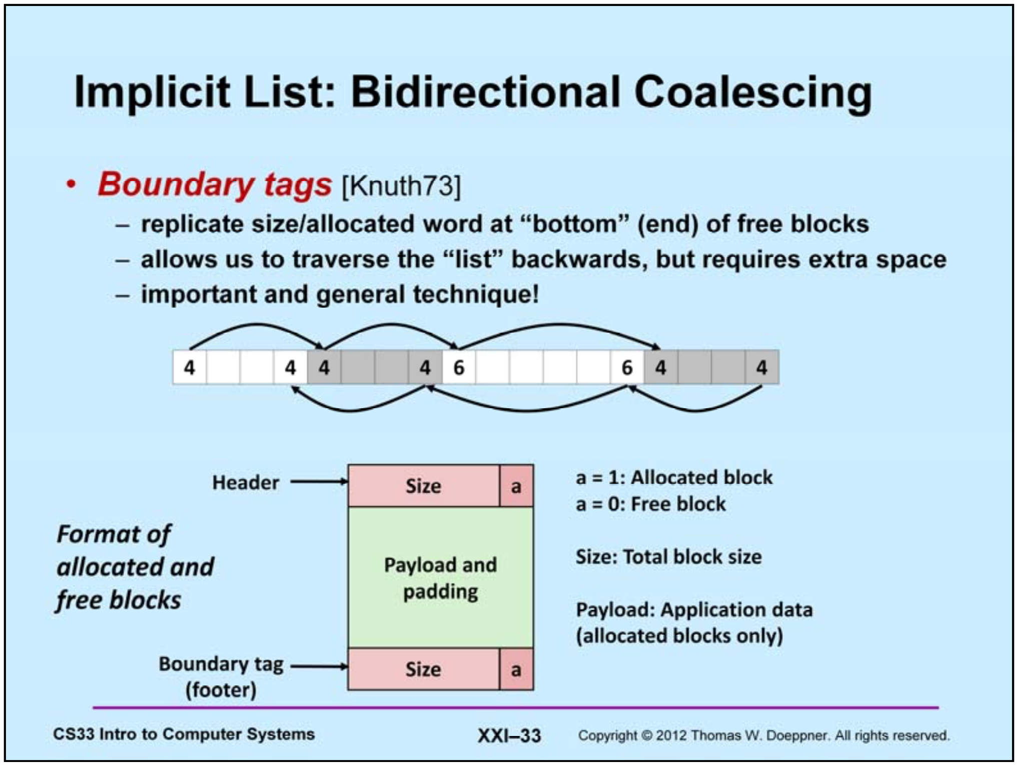
- **Join *(coalesce)* with next/previous blocks, if they are free**
  - **coalescing with next block**



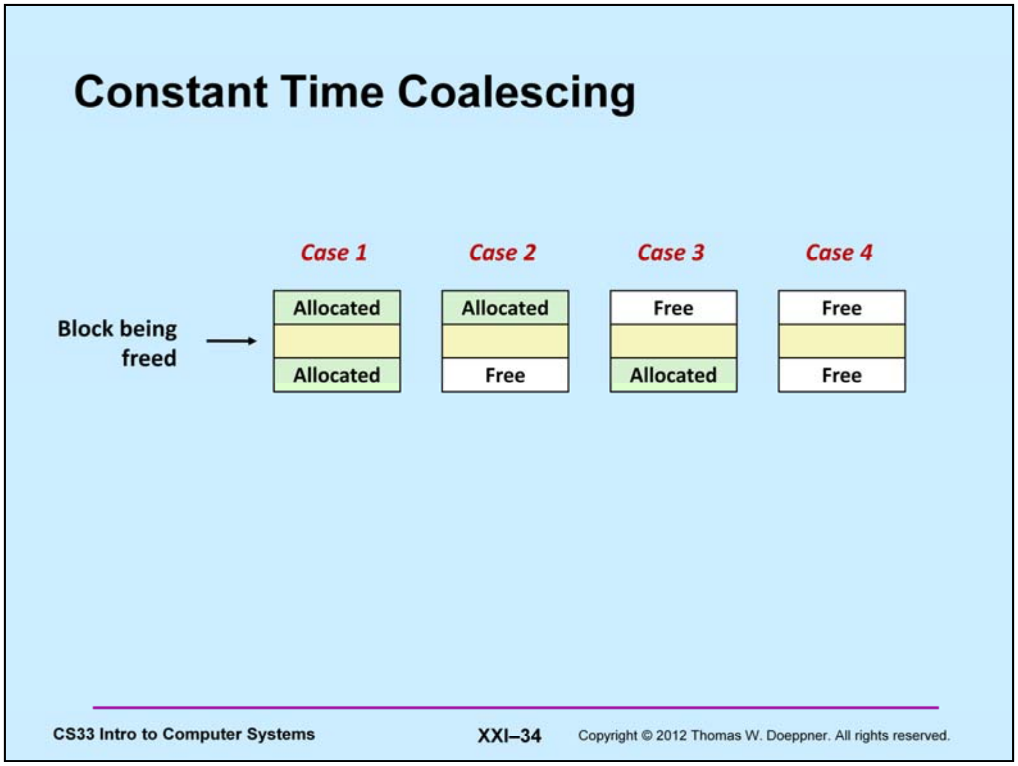

```
void free_block(ptr p) {
    *p = *p & -2;           // clear allocated flag
    next = p + *p;          // find next block
    if ((*next & 1) == 0)
      *p = *p + *next;      // add to this block if
}                           //    not allocated
```

  - **but how do we coalesce with *previous* block?**

Supplied by CMU.

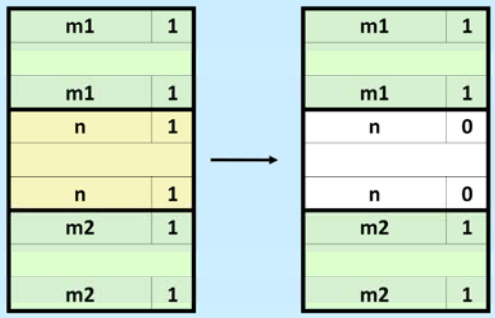


Supplied by CMU.

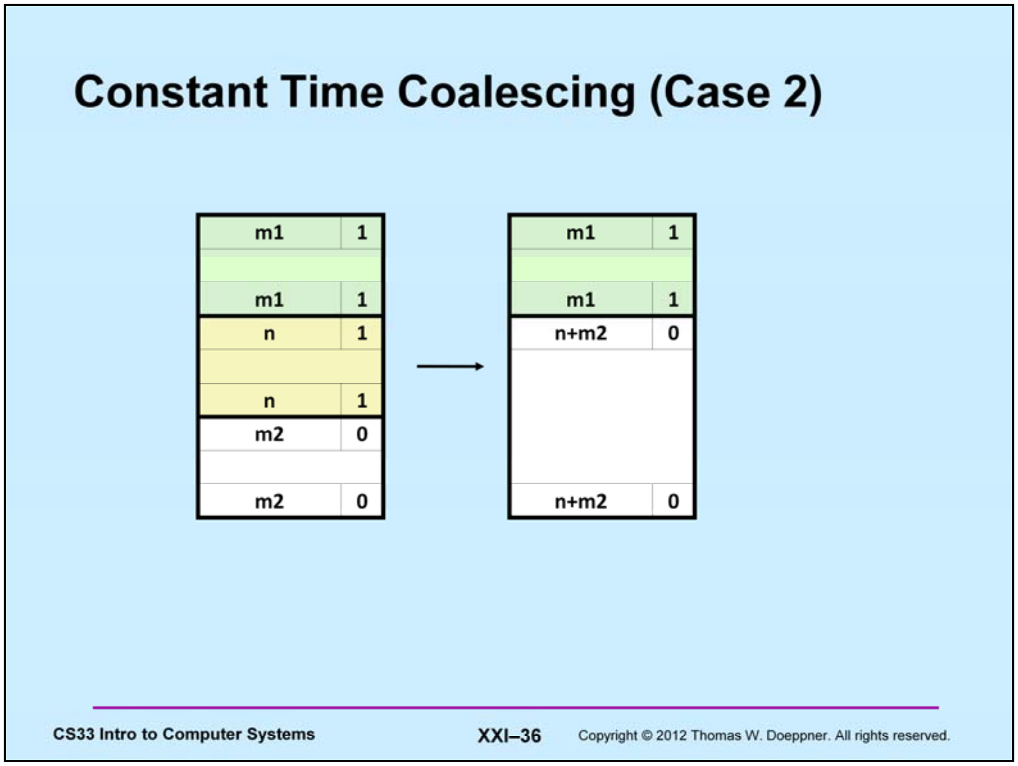


Supplied by CMU.

Constant Time Coalescing (Case 1)

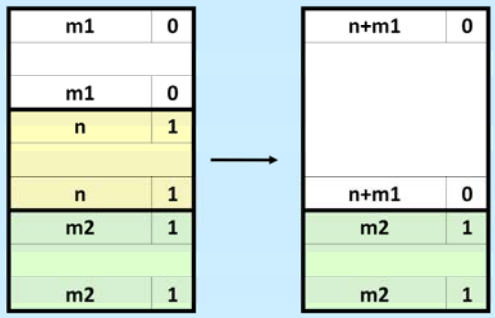


Supplied by CMU.



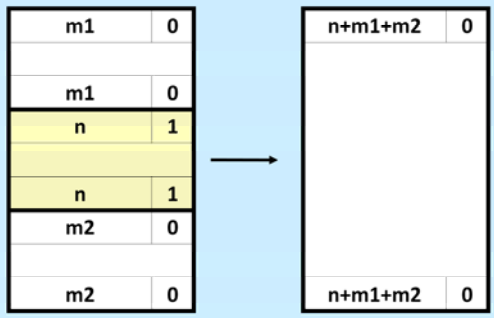
Supplied by CMU.

Constant Time Coalescing (Case 3)



Supplied by CMU.

Constant Time Coalescing (Case 4)



Supplied by CMU.

Disadvantages of Boundary Tags

- Internal fragmentation
- Can it be optimized?

Supplied by CMU.

Summary of Key Allocator Policies

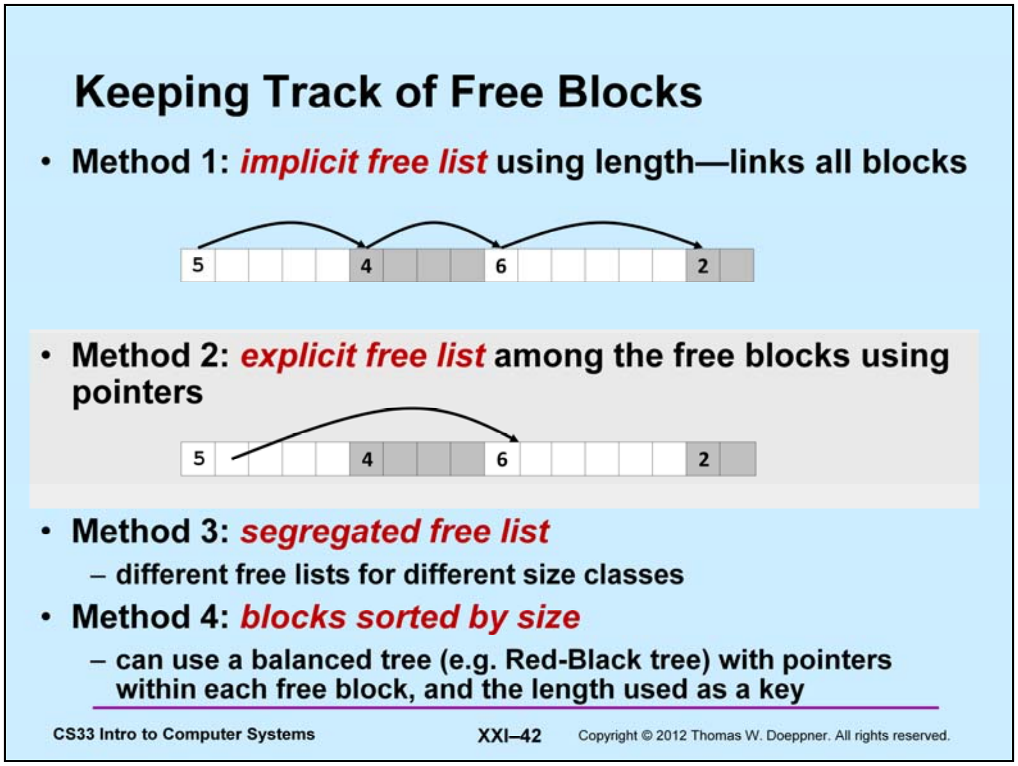
- **Placement policy:**
  - first-fit, next-fit, best-fit, etc.
  - trades off lower throughput for less fragmentation
  - *interesting observation*: segregated free lists (next lecture) approximate a best fit placement policy without having to search entire free list
- **Splitting policy:**
  - when 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

Supplied by CMU.

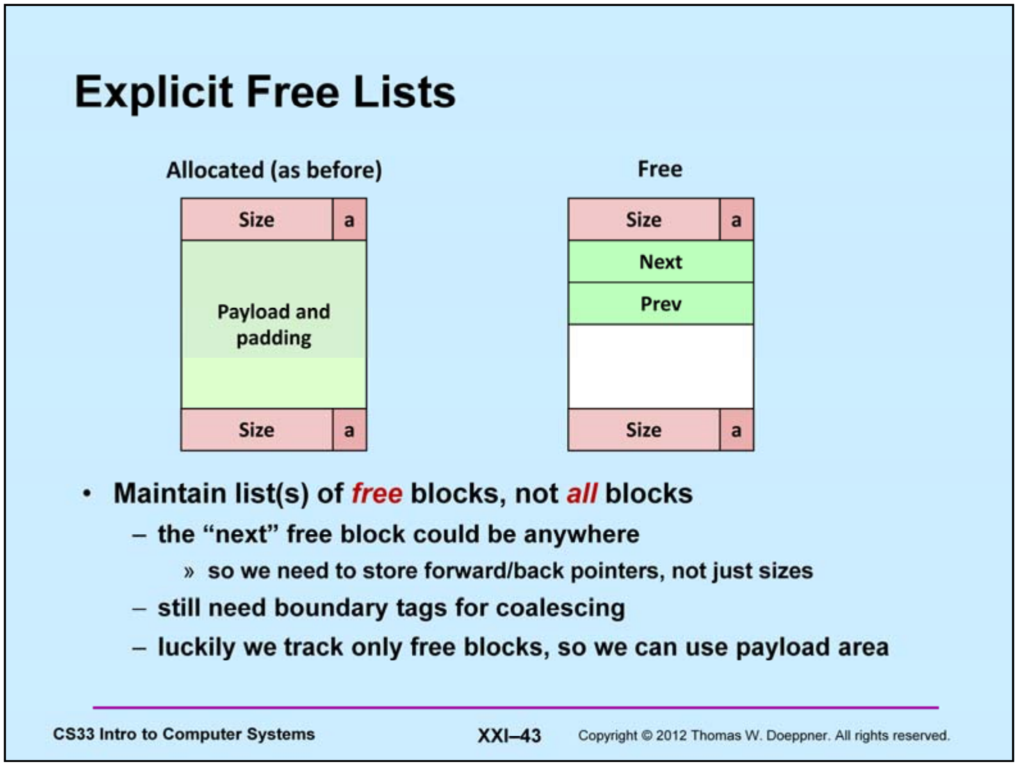
Implicit Lists: Summary

- **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**

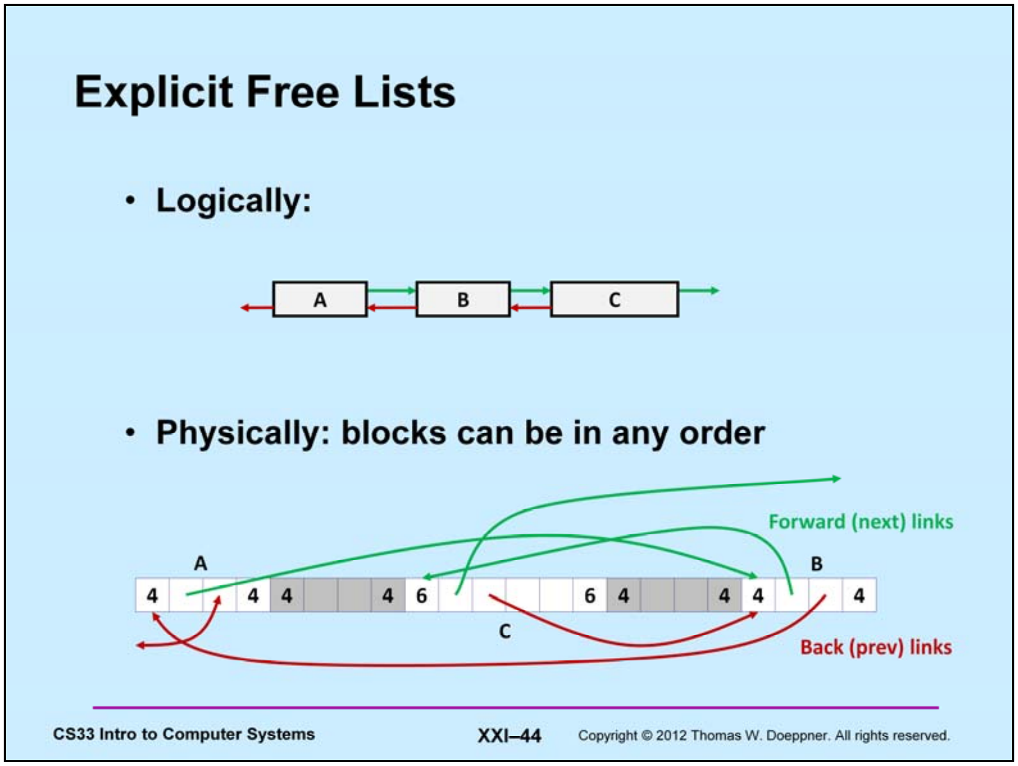
Supplied by CMU.



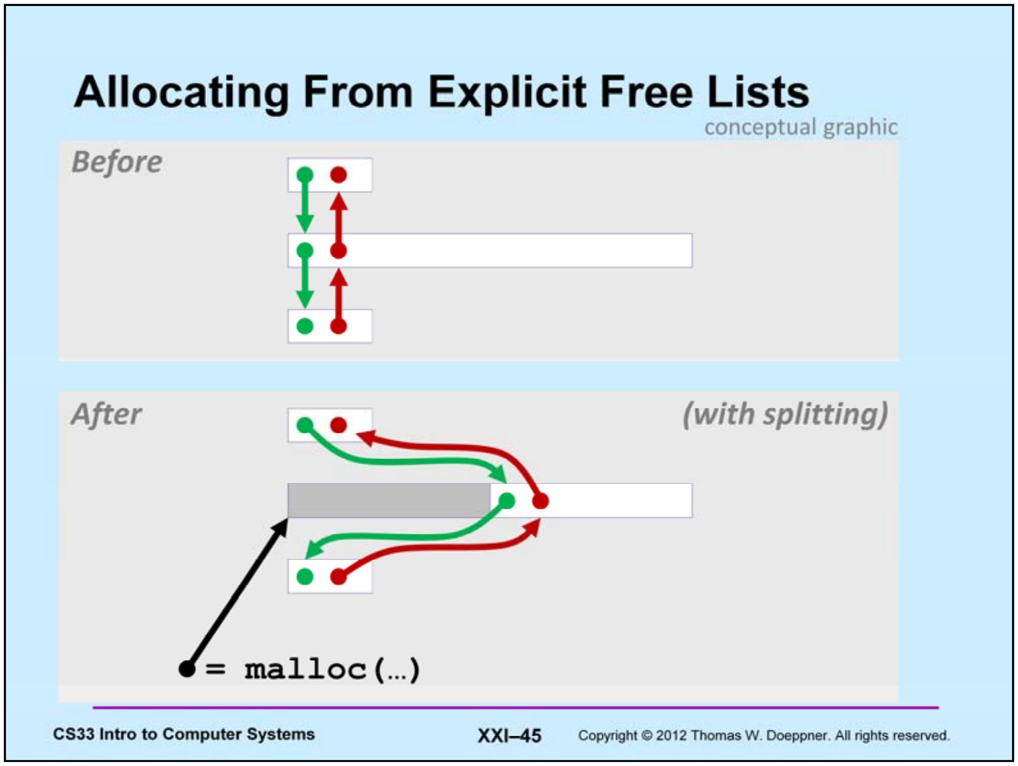
Supplied by CMU.



Supplied by CMU.



Supplied by CMU.



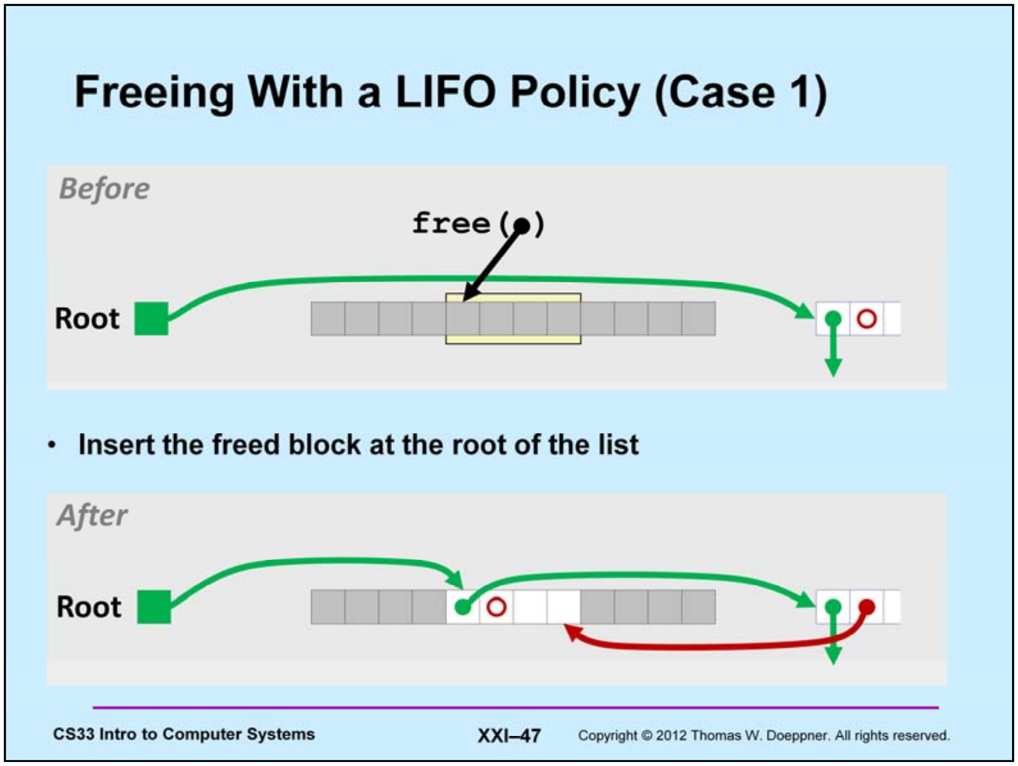
Supplied by CMU.

Freeing With Explicit Free Lists

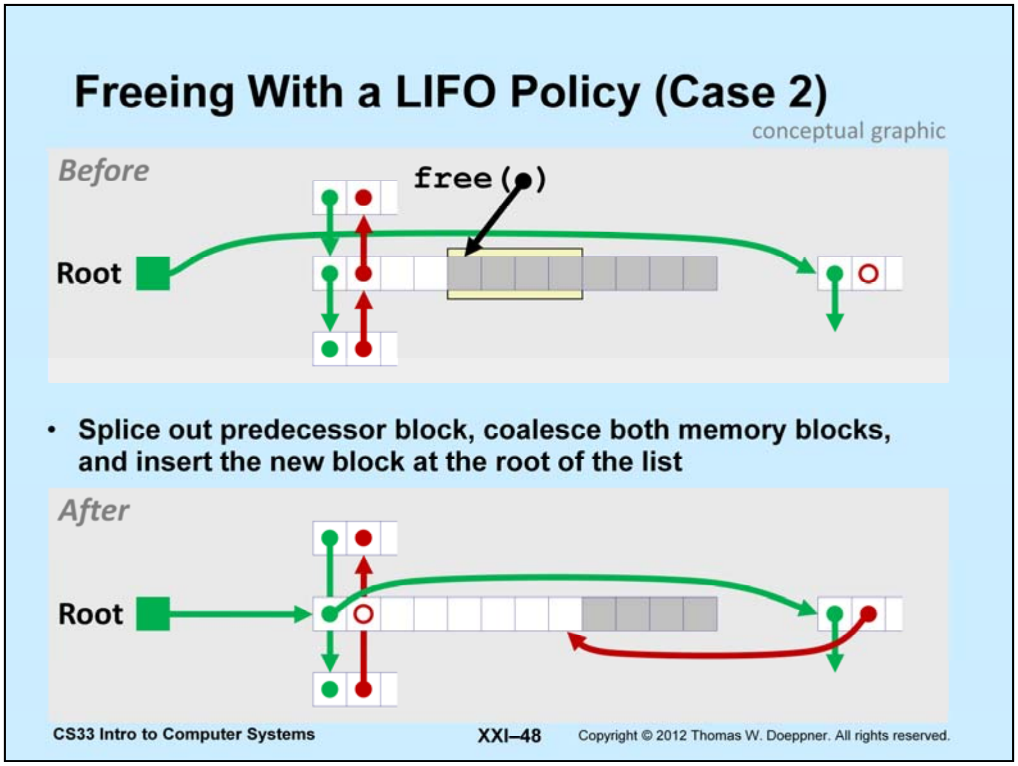
- ***Insertion policy*: where in the free list do you put a newly freed block?**
  - **LIFO (last-in-first-out) policy**
    - » **insert freed block at the beginning of the free list**
    - » ***pro:* simple and constant time**
    - » ***con:* studies suggest fragmentation is worse than address ordered**
  - **address-ordered policy**
    - » **Insert freed blocks so that free list blocks are always in address order:**

      addr(prev) < addr(curr) < addr(next)
    - » ***con:* requires search**
    - » ***pro:* studies suggest fragmentation is lower than LIFO**

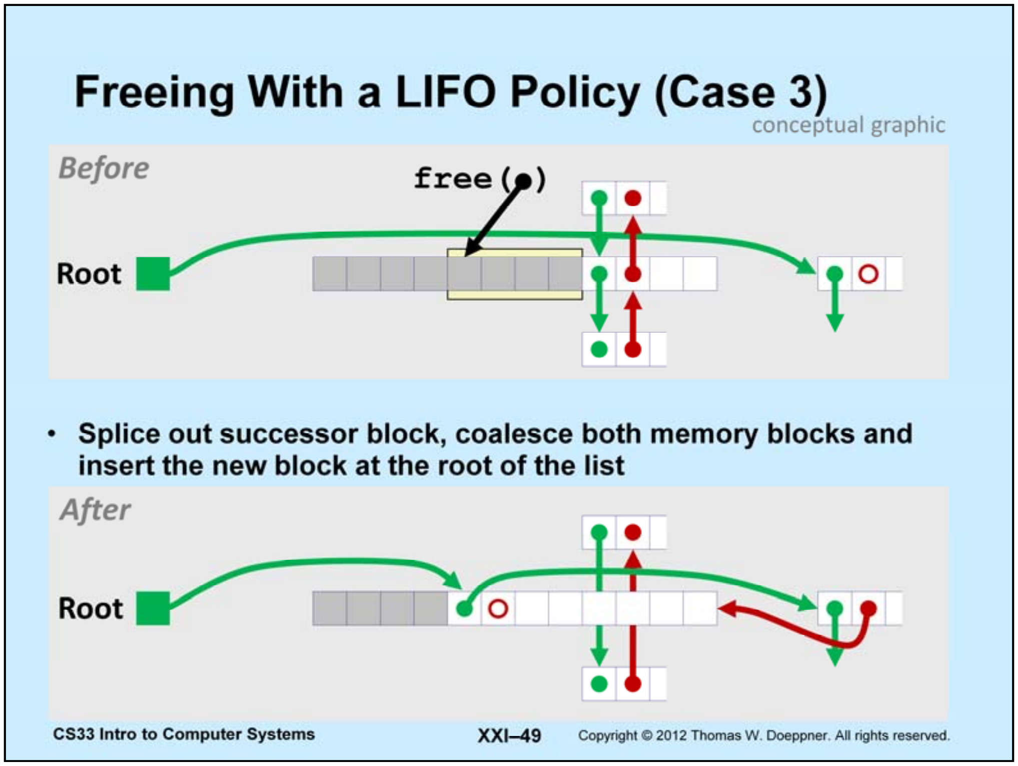
Supplied by CMU.



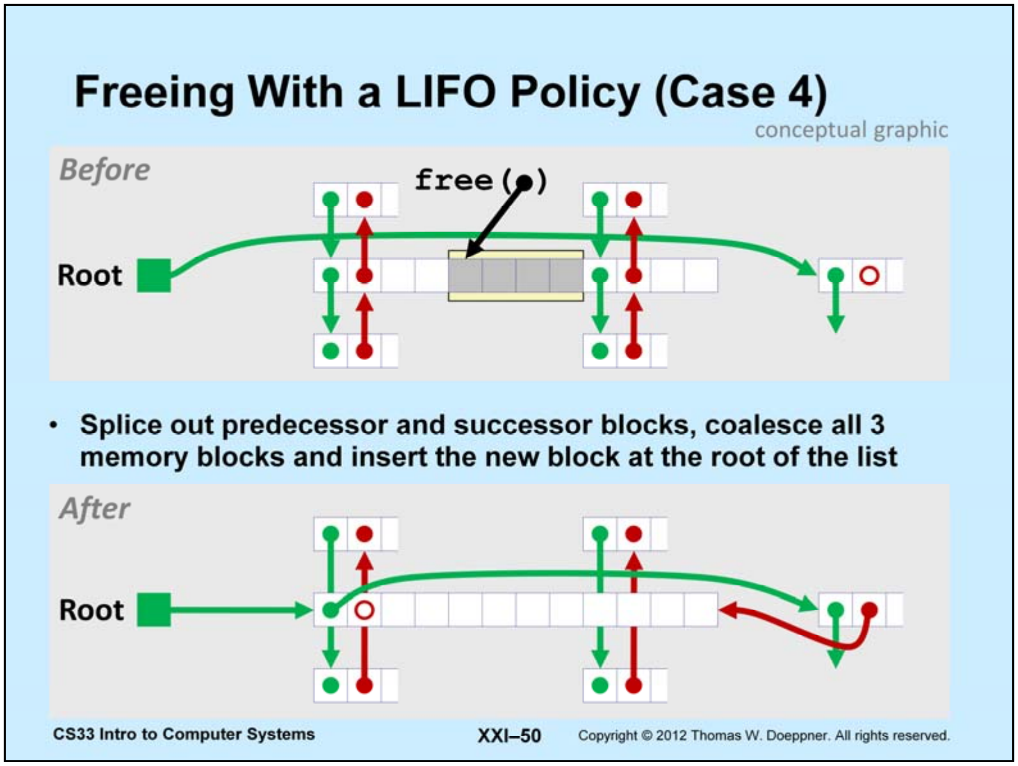
Supplied by CMU.



Supplied by CMU.



Supplied by CMU.



Supplied by CMU.

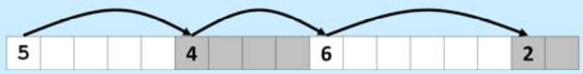
Explicit List Summary

- **Comparison to implicit list:**
  - allocate is linear time in number of ***free*** blocks instead of ***all*** blocks
    - » ***much faster*** 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 (2 extra words needed for each block)
    - » does this increase internal fragmentation?
- **Most common 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

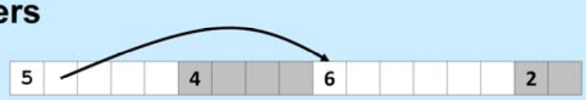
Supplied by CMU.

Keeping Track of Free Blocks

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

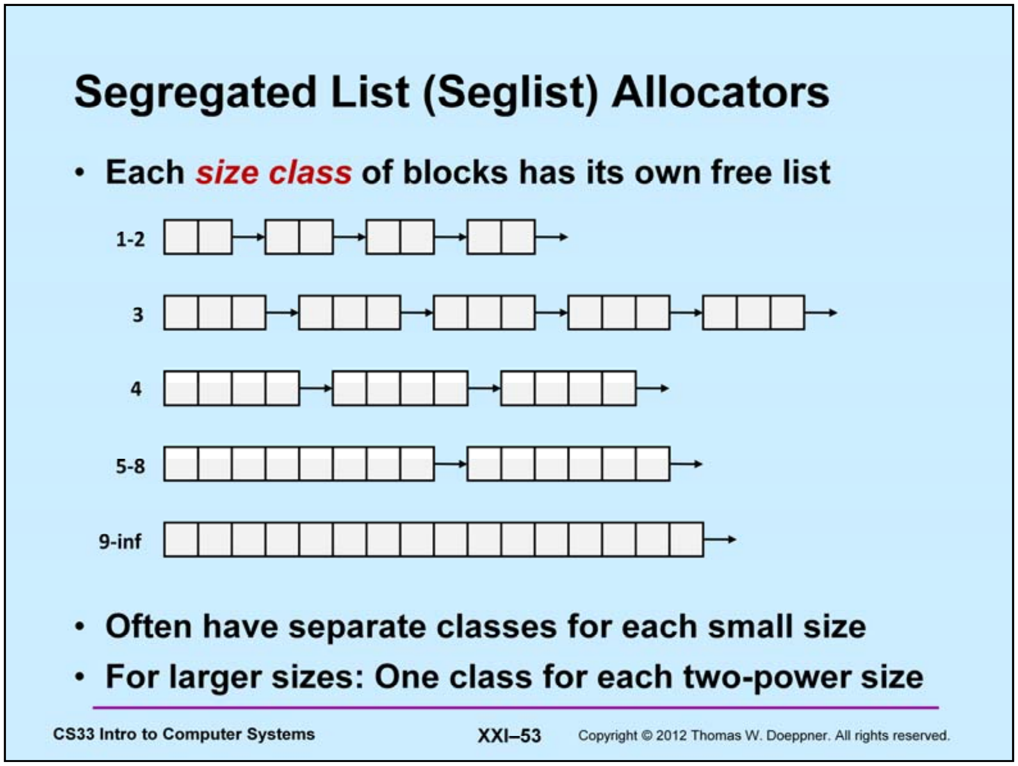


- Method 2: ***explicit list*** among the free blocks using pointers



- 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

Supplied by CMU.



Supplied by CMU.

Seglist Allocator

- Given an array of free lists, each one for some size class
- To allocate a block of size *n*:
  - search appropriate free list for block of size *m* > *n*
  - if an appropriate block is found:
    - split block and place fragment on appropriate list (optional)
  - if no block is found, try next larger class
  - repeat until block is found
- If no block is found:
  - request additional heap memory from OS (using `sbrk()`)
  - allocate block of *n* bytes from this new memory
  - place remainder as a single free block in largest size class

Supplied by CMU.

Seglist Allocator (cont.)

- **To free a block:**
  - coalesce and place on appropriate list (optional)
- **Advantages of seglist allocators**
  - higher throughput
    - » log time for power-of-two size classes
  - better memory utilization
    - » first-fit search of segregated free list approximates a best-fit search of entire heap.
    - » extreme case: giving each block its own size class is equivalent to best-fit

Supplied by CMU.

More Info on Allocators

- D. Knuth, "*The Art of Computer Programming*", 2nd 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
  - available from CS:APP student site (csapp.cs.cmu.edu)

Supplied by CMU.