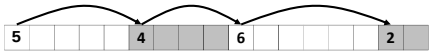
Dynamic Memory Allocation: Advanced Concepts

Kai Shen

1

Keeping Track of Free Blocks

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

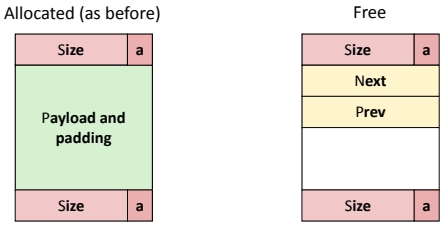


- Method 2: *Explicit list* among the free blocks using pointers

- 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

2

Explicit Free Lists

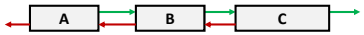


- Maintain list(s) of *free* blocks, not *all* blocks
  - The "next" free block could be anywhere
    - So we need to store forward/back pointers, not just sizes
    - Luckily we track only free blocks, the space usage is pretty much free
  - Still need boundary tags for coalescing

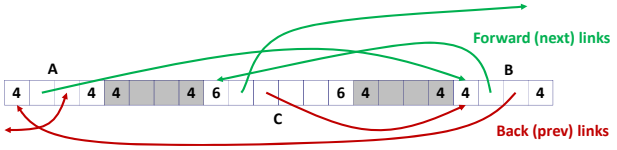
3

Explicit Free Lists

- Logically:



- Physically: blocks can be in any order



4

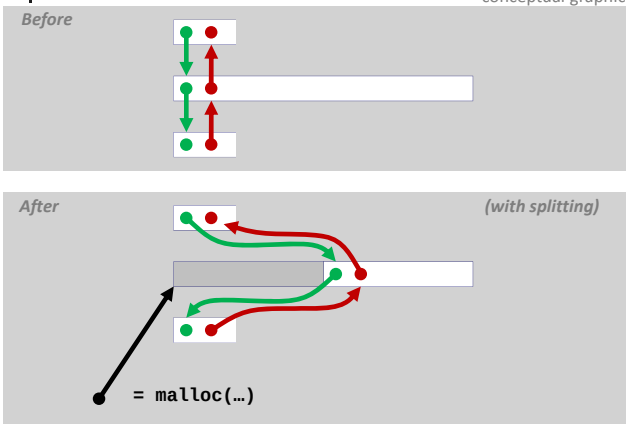
Allocating from Explicit Free Lists

- First fit
- Next fit
- Best fit

5

Allocating From Explicit Free Lists

conceptual graphic



6

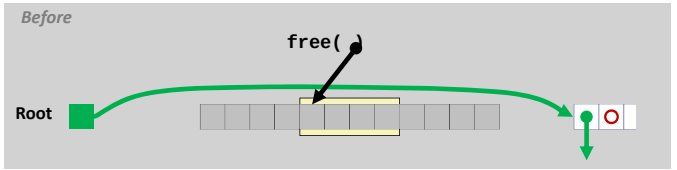
Freeing With Explicit Free Lists

- Where in the free list do you put a newly freed block?
  - An intuitive approach is to insert at the front of the free list
  - ***Pro:*** simple and constant time

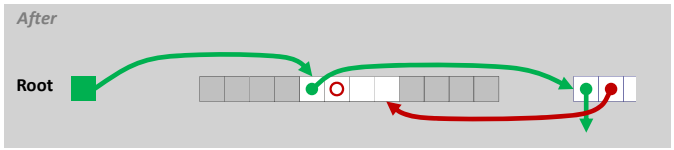
7

Freeing to Front

conceptual graphic



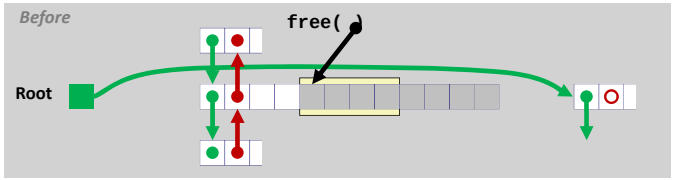
- Insert the freed block at the root of the list



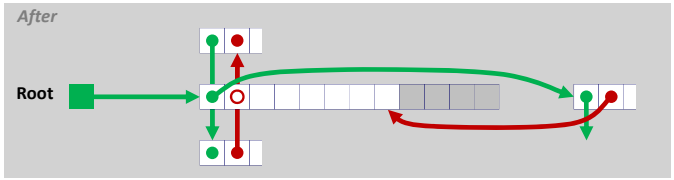
8

Freeing to Front (Coalescing)

conceptual graphic



- Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list



9

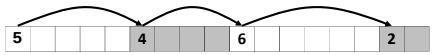
Explicit List Summary

- 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 allocation 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)
    - Is this a big deal?
- But linear allocation time is still bad!

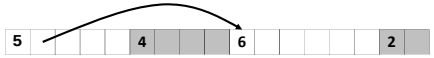
10

Keeping Track of Free Blocks

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



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

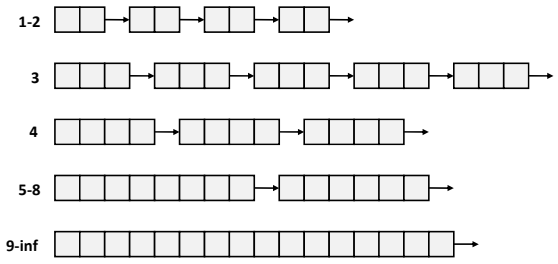


- Method 3: *Segregated free list*
  - Different free lists for different size classes
- Method 4: *Blocks sorted by size*
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

11

Segregated List Allocators

- Each *size class* of blocks has its own free list



12

Segregated List Allocator

- 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
- To free a block:
  - Coalesce and place on appropriate list (optional)
- Advantages of segregated list allocators
  - High speed
    - Linear on the number of lists
    - log time for power-of-two size classes
  - Good memory utilization
    - Approximates a best-fit search of entire heap.

13

Implicit Memory Management: Garbage Collection

- *Garbage collection:* automatic reclamation of heap-allocated storage—application never has to free
- How does the memory manager know when memory can be freed?
  - We cannot predict what is going to be used in the future with knowing future program execution and data input
  - But we can tell that certain heap objects cannot be accessed if there are no references to them

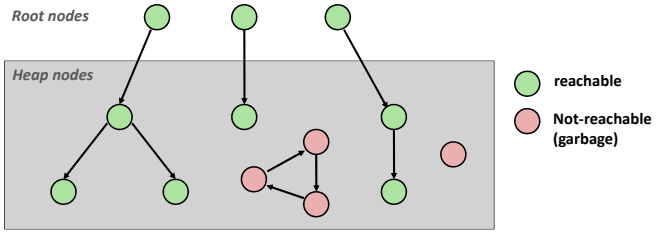
```
void foo() {
   String[] p = new String[2];
   return; /* p block is now garbage */
}
```

  - All heap accesses must be made through known "references"

14

Memory as a Graph

- We view memory as a directed graph
  - Each data object is a node in the graph
  - Each reference is an edge in the graph
  - Registers, global variables, locations on the stack are always reachable



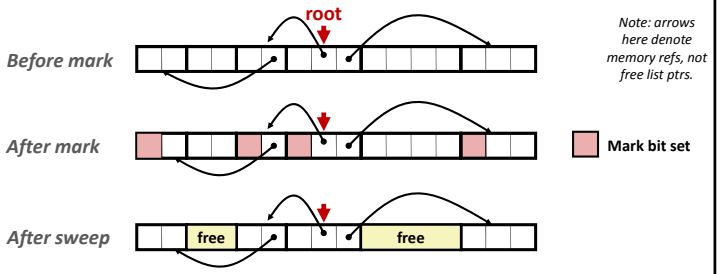
A node (object) is *reachable* if there is a path from any root to that node.

Non-reachable nodes are *garbage* (cannot be accessed by the application)

15

Mark and Sweep Collecting

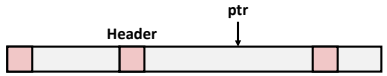
- Garbage Collection
  - Use extra ***mark bit*** in the head of each block
  - ***Mark:*** Start at roots and set mark bit on each reachable block
  - ***Sweep:*** Scan all blocks and free blocks that are not marked



16

Garbage Collection for C

- Problem for C garbage collection
  - References (pointers) are not strongly regulated
  - For instance, can point to middle of an allocated block



- So how to identify the block when pointer points to the middle?
  - Search all allocated blocks
  - Can use a balanced binary tree to keep track of all allocated blocks in address order
- Are we done?
  - C types can be cast back and forth
  - I save **ptr-1024** and add 1024 when using it

17

Memory-Related Perils and Pitfalls

- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Misunderstanding of pointer arithmetic
- Referencing nonexistent variables
- Referencing freed blocks
- Failing to free blocks

18

Dereferencing Bad Pointers

```
int val;

...

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

19

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;
}
```

20

Overwriting Memory

```
int **p;

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

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

- Off-by-one error

21

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

22

Misunderstanding of Pointer Arithmetic

```
int *search(int *p, int val) {
   while (*p && *p != val)
      p += sizeof(int);

   return p;
}
```

23

Referencing Nonexistent Variables

- Forgetting that local variables disappear when a function returns

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

   return &val;
}
```

24

Referencing Freed Blocks

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

25

Failing to Free Blocks (Memory Leaks)

- Slow, long-term killer!

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

26

Failing to Free Blocks (Memory Leaks)

- Freeing only part of a data structure

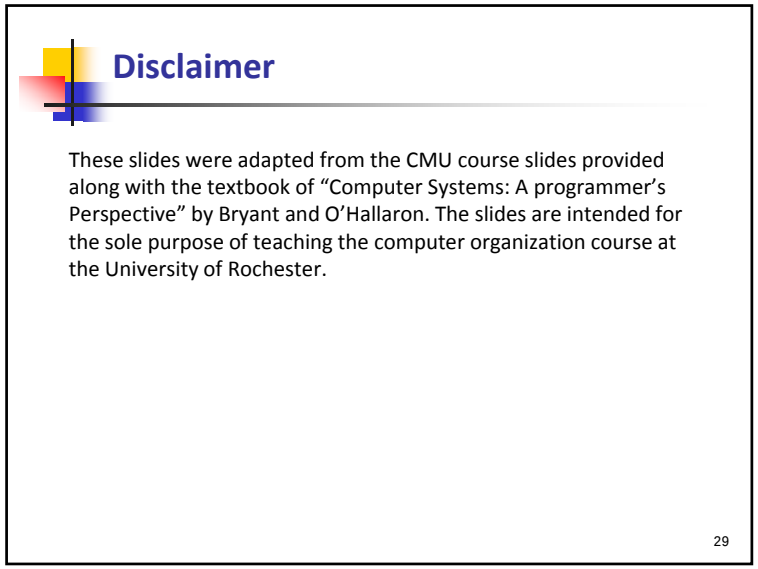
```
struct list {
   int val;
   struct list *next;
};

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;
}
```

27

Dealing With Memory Bugs

- Conventional debugger (**gdb**)
  - Not able to do much
- Debugging **`malloc`**
  - Wrapper around conventional **`malloc`**
  - Add canary values on allocation boundaries and do sanity checks sometimes
- Some malloc implementations contain checking code
  - Linux glibc malloc: **`setenv MALLOC_CHECK_ 2`**
- Binary translator: valgrind (Linux), Purify
  - Maintain memory map/status structure for each byte
  - Instrument data accesses to check for errors
- Garbage collection (Boehm-Weiser Conservative GC)
  - Let the system free blocks instead of the programmer.

28


Disclaimer
These slides were adapted from the CMU course slides provided along with the textbook of “Computer Systems: A programmer’s Perspective” by Bryant and O’Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.
29