

Dynamic Memory Allocation: Advanced Concepts

15-213/14-513/15-513: Introduction to Computer Systems
14th Lecture, March 2, 2023

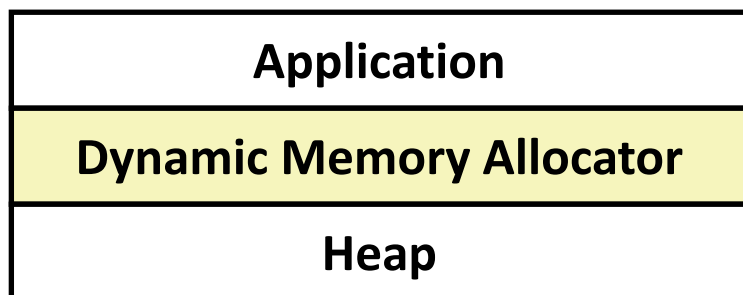
Instructors:

Dave Andersen (15-213)

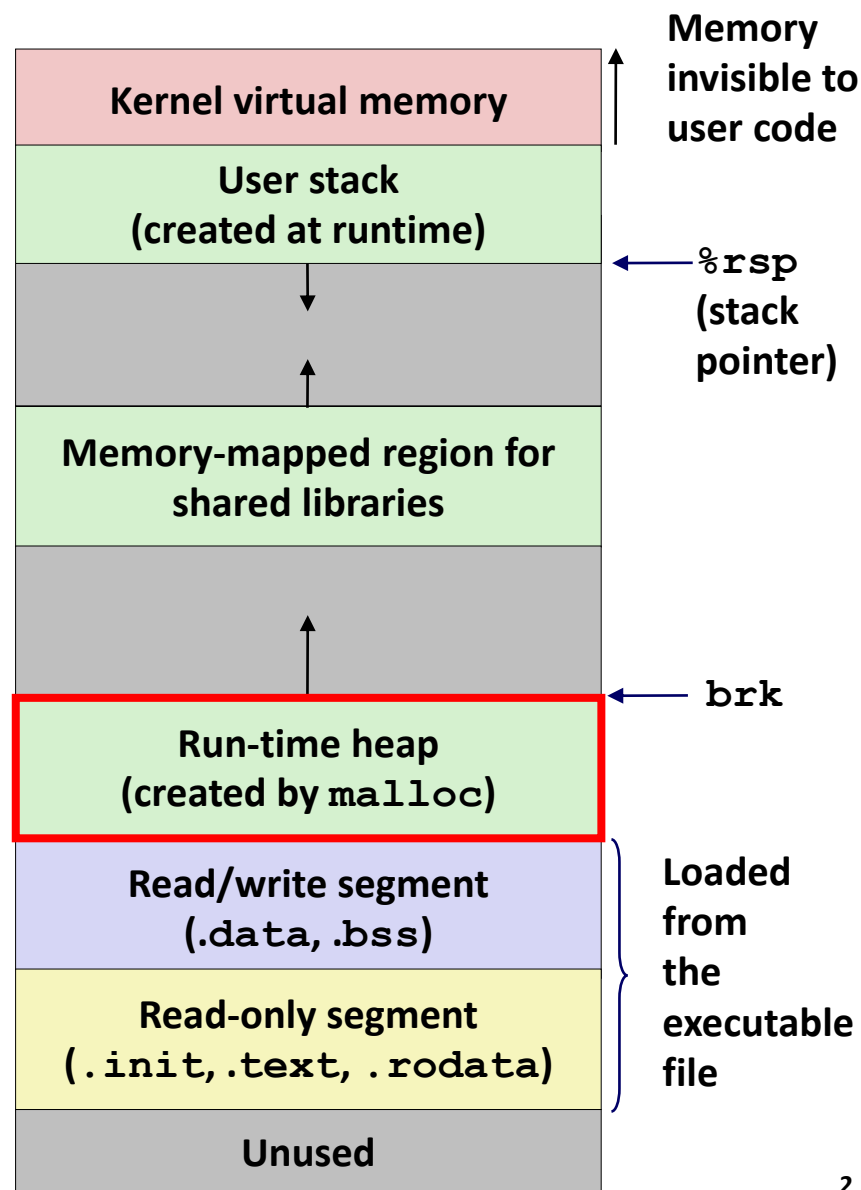
Zack Weinberg (15-213)

Brian Railing (15-513)

Review: Dynamic Memory Allocation

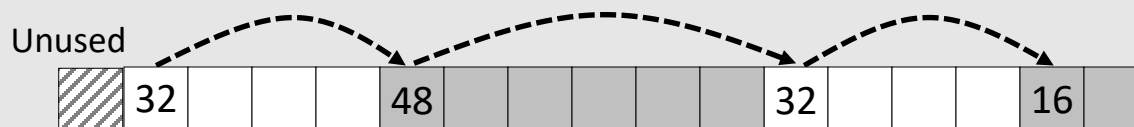


- Programmers use *dynamic memory allocators* (such as `malloc`) to acquire virtual memory (VM) at runtime
 - For data structures whose size is only known at runtime
- Dynamic memory allocators manage an area of process VM known as the *heap*



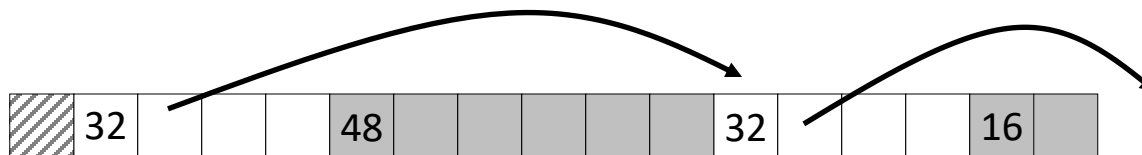
Review: Keeping Track of Free Blocks

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



Need to tag each block as allocated/free

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



Need space for 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

Review: Implicit Lists Summary

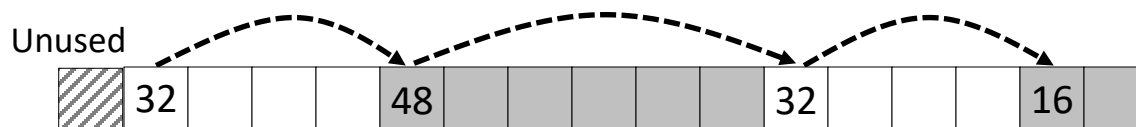
- **Implementation: very simple**
- **Allocate cost:**
 - linear time worst case
- **Free cost:**
 - constant time worst case
 - even with coalescing
- **Memory Overhead:**
 - Depends on placement policy
 - Strategies include first fit, next fit, and best fit
- **Not used in practice for `malloc/free` 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**

Today

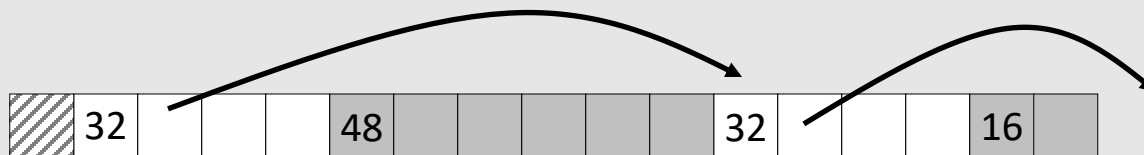
- **Explicit free lists**
- Segregated free lists
- Memory-related perils and pitfalls

Keeping Track of Free Blocks

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



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



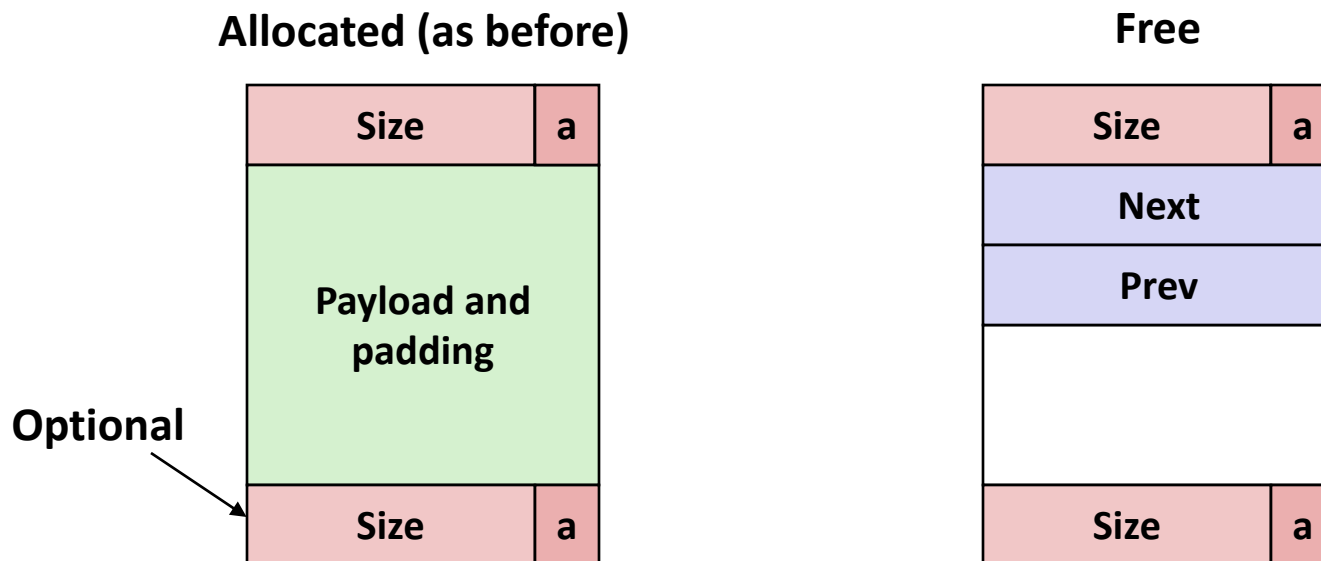
- Method 3: *Segregated free list*

- 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

Explicit Free Lists



■ Maintain list(s) of *free* blocks, not *all* blocks

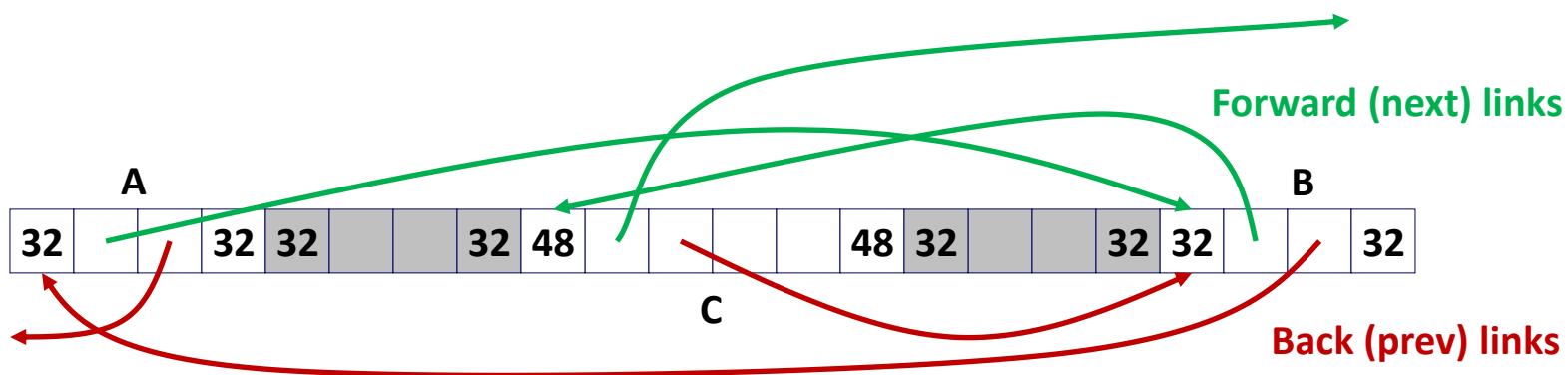
- Luckily we track only free blocks, so we can use payload area
- The “next” free block could be anywhere
 - So we need to store forward/back pointers, not just sizes
- Still need boundary tags for coalescing
 - To find adjacent blocks according to memory order

Explicit Free Lists

- Logically:



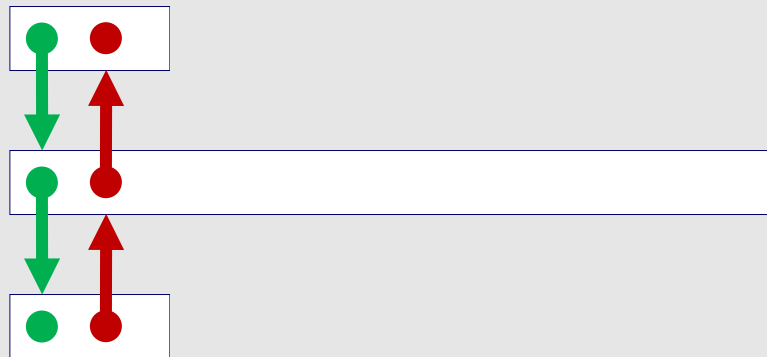
- Physically: blocks can be in any order



Allocating From Explicit Free Lists

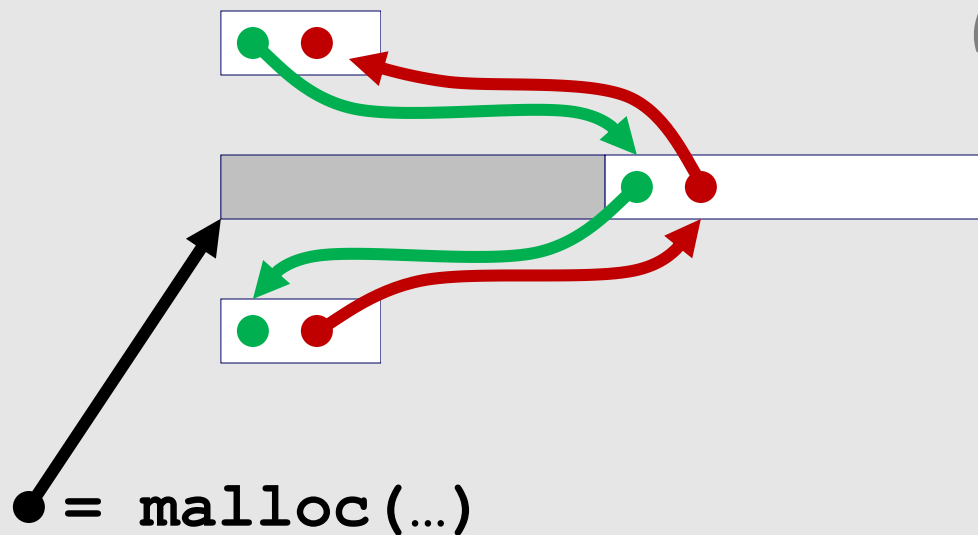
conceptual graphic

Before



After

(with splitting)



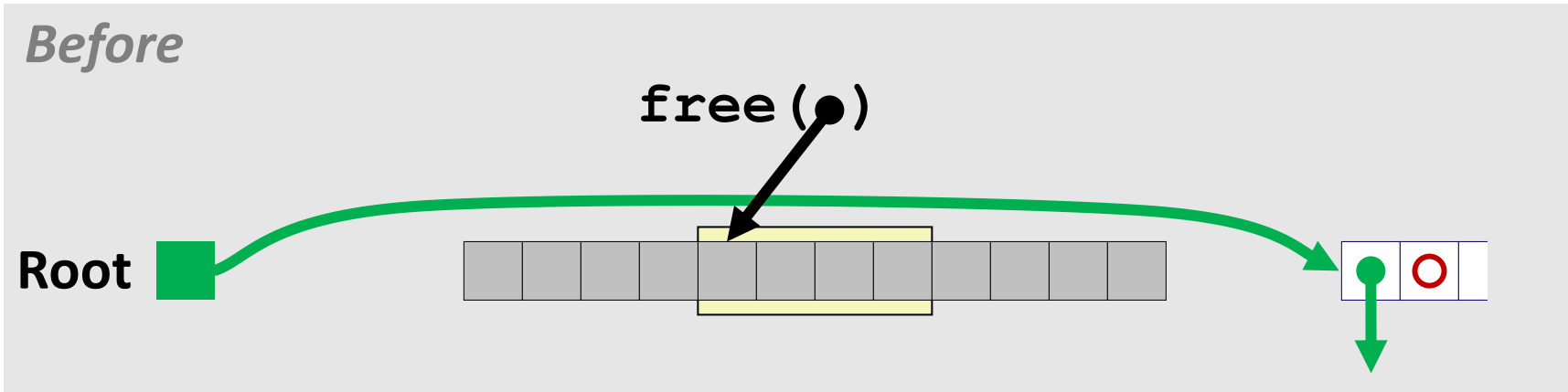
Freeing With Explicit Free Lists

- ***Insertion policy:*** Where in the free list do you put a newly freed block?
- **Unordered**
 - LIFO (last-in-first-out) policy
 - Insert freed block at the beginning of the free list
 - FIFO (first-in-first-out) policy
 - Insert freed block at the end 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/FIFO

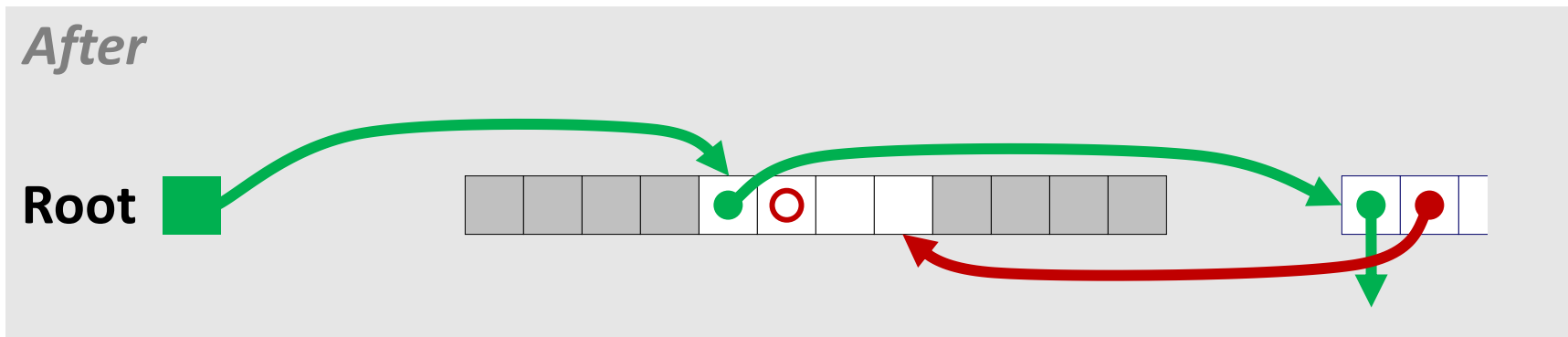
Freeing With a LIFO Policy (Case 1)



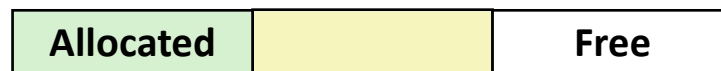
conceptual graphic



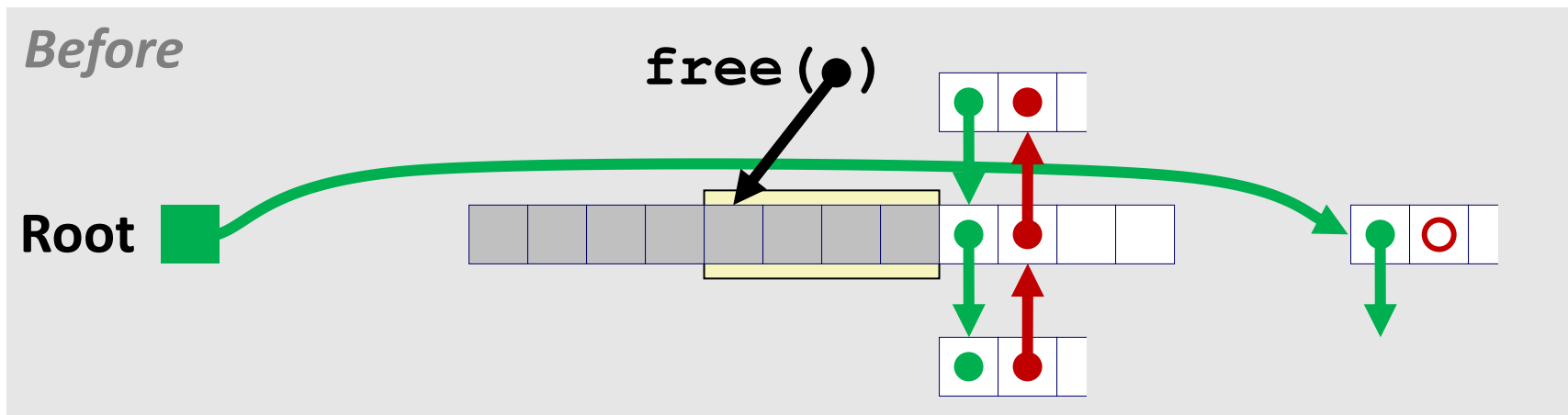
- Insert the freed block at the root of the list



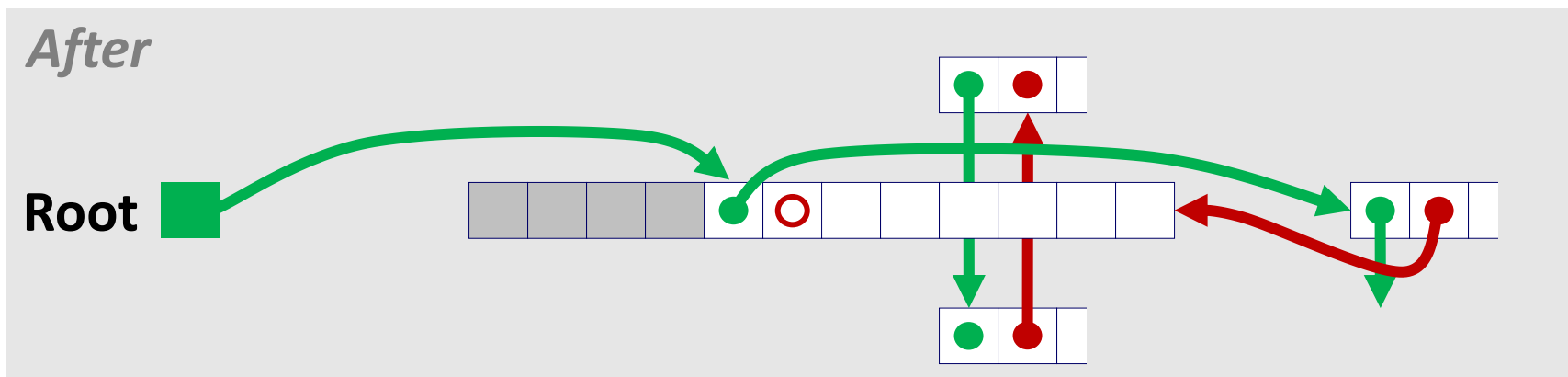
Freeing With a LIFO Policy (Case 2)



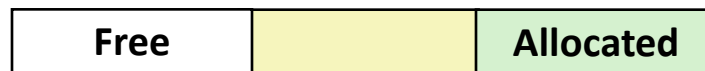
conceptual graphic



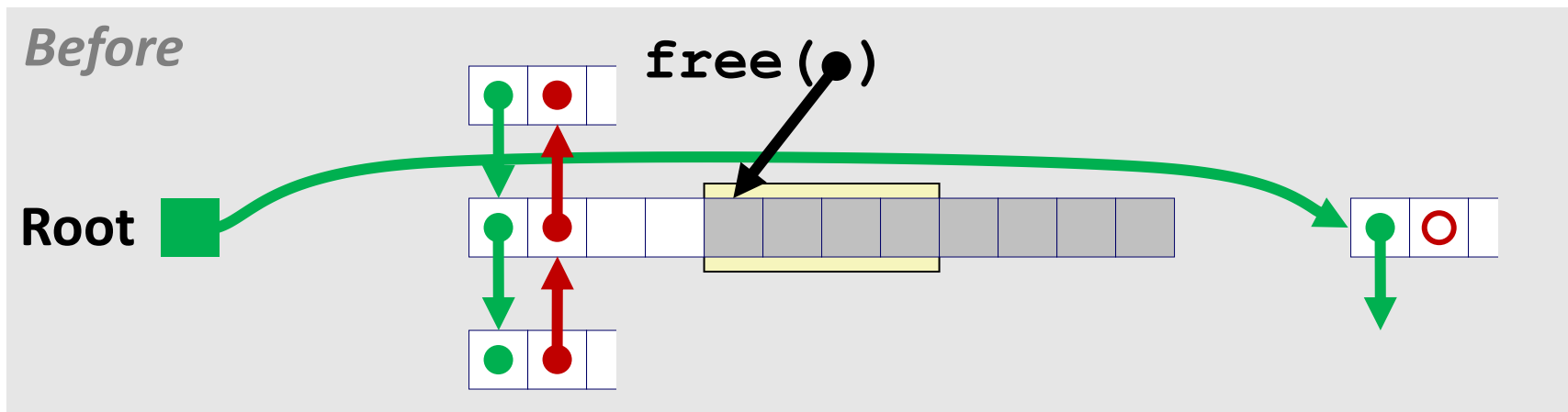
- Splice out adjacent successor block, coalesce both memory blocks, and insert the new block at the root of the list



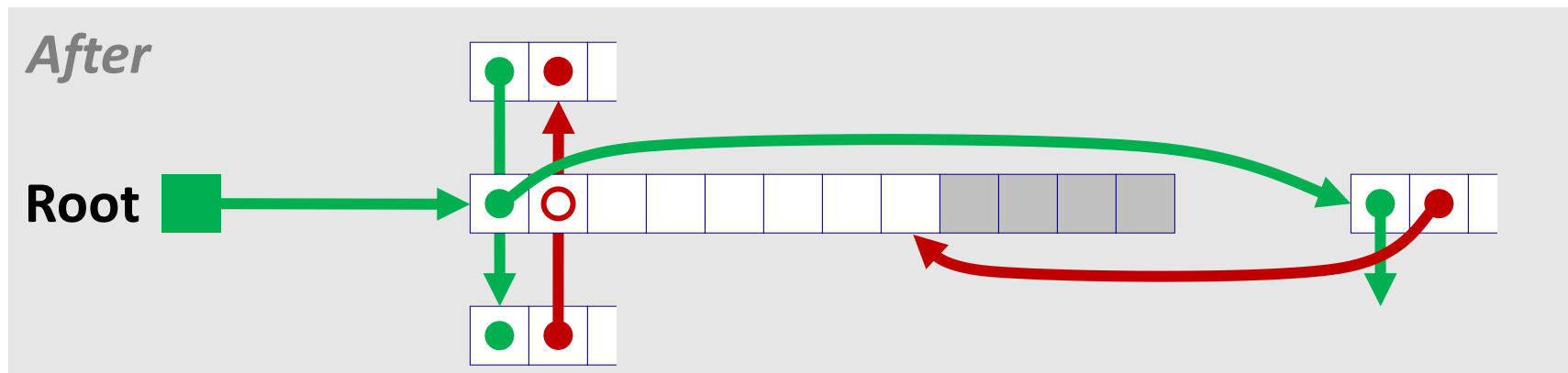
Freeing With a LIFO Policy (Case 3)



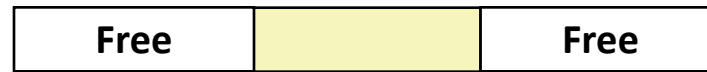
conceptual graphic



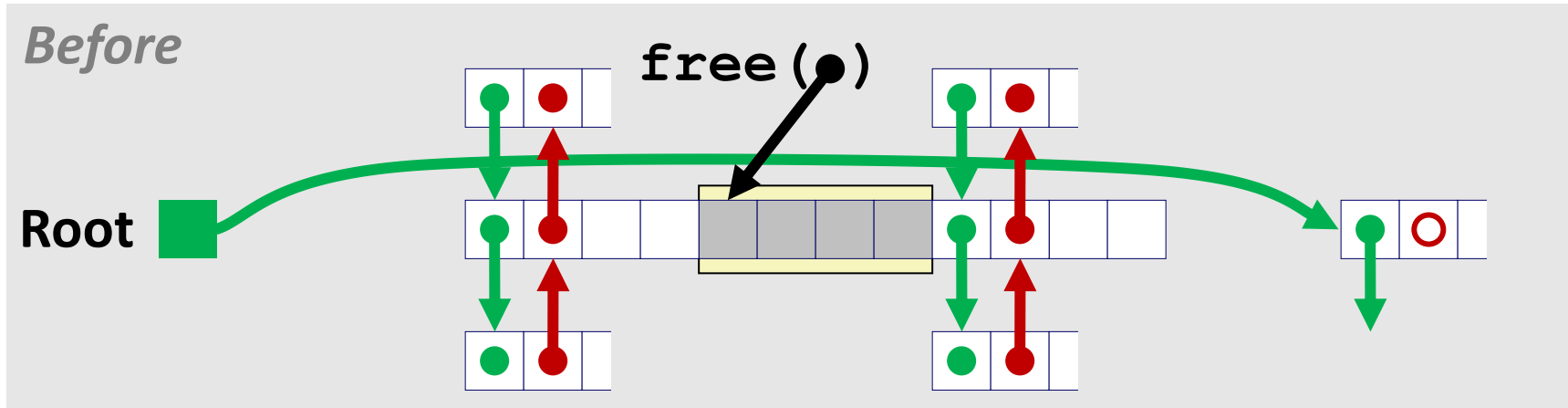
- Splice out adjacent predecessor block, coalesce both memory blocks, and insert the new block at the root of the list



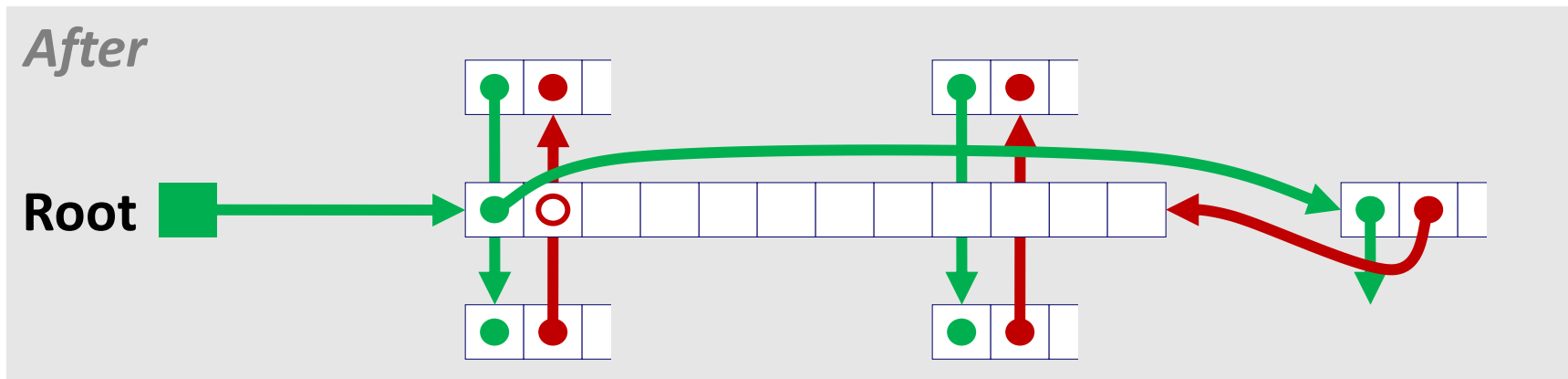
Freeing With a LIFO Policy (Case 4)



conceptual graphic



- Splice out adjacent predecessor and successor blocks, coalesce all 3 blocks, and insert the new block at the root of the list



Explicit List Summary

■ Comparison to implicit list:

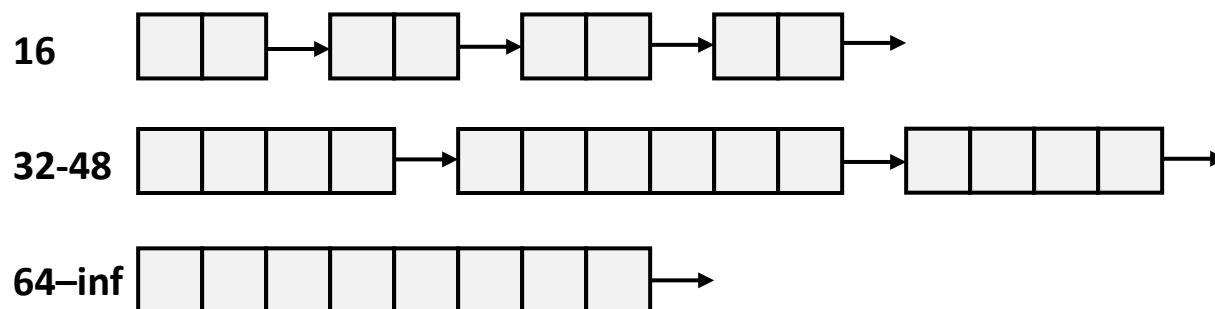
- Allocate is linear time in number of *free* blocks instead of *all* blocks
 - *Much faster* when most of the memory is full
- Slightly more complicated allocate and free
 - Need to splice blocks in and out of the list
- Some extra space for the links (2 extra words needed for each block)
 - Does this increase internal fragmentation?

Today

- Explicit free lists
- **Segregated free lists**
- Memory-related perils and pitfalls

Segregated List (Seglist) Allocators

- Have several free lists, one for each *size class* of blocks



- Which blocks go in which size classes is a design decision
 - Can have major impact on both utilization and throughput
 - Common choices include:
 - One class for each small size (16, 32, 48, 64, ...)
 - At some point switch to powers of two: $[2^i + 1, 2^{i+1}]$
- The list for the *largest* blocks must have no upper limit
 - (well, 2^{64})

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 $m \geq n$ (i.e., first fit)
 - If an appropriate block is found:
 - Split block and place fragment on appropriate list
 - 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 appropriate size class.

Seglist Allocator (cont.)

- **To free a block:**
 - Coalesce and place on appropriate list

- **Advantages of seglist allocators vs. non-seglist allocators (both with first-fit)**
 - Higher throughput
 - log time for power-of-two size classes vs. linear time
 - 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.

More Info on Allocators

- **D. Knuth, *The Art of Computer Programming*, vol 1, 3rd edition, Addison Wesley, 1997**
 - 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)

Quiz

<https://canvas.cmu.edu/courses/33813/quizzes/97406>

Today

- Explicit free lists
- Segregated free lists
- **Memory-related perils and pitfalls**

Memory-Related Perils and Pitfalls

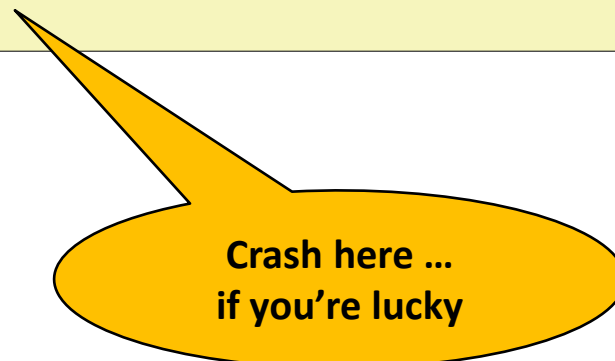
- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks
- Failing to free blocks

Dereferencing Bad Pointers

■ The classic scanf bug

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

```
case 'd': {  
    int *valp = va_arg(ap, int *);  
    *valp = (int)strtol(valbuf, &endp, 10);  
}
```



**Crash here ...
if you're lucky**

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;
}
```

- Can avoid by using `calloc`

Overwriting Memory

- Allocating the (possibly) wrong sized object

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

- Can you spot the bug?

Overwriting Memory

■ Off-by-one errors

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

```
char *p;  
  
p = malloc(strlen(s));  
strcpy(p, s);
```

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

Overwriting Memory

- Misunderstanding pointer arithmetic

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

Overwriting Memory

- Referencing a pointer instead of the object it points to

```
int *BinheapDelete(int **binheap, int *size) {
    int *packet;
    packet = binheap[0];
    binheap[0] = binheap[*size - 1];
    *size--;
    Heapify(binheap, *size, 0);
    return(packet);
}
```

- What gets decremented?
 - (See next slide)

Overwriting Memory

- Referencing a pointer instead of the object it points to

```
int *BinheapDelete(int **binheap, int *size) {
    int *packet;
    packet = binheap[0];
    binheap[0] = binheap[*size - 1];
    *size--;
    Heapify(binheap, *size, 0);
    return(packet);
}
```

- Same effect as

- `size--;`

- Rewrite as

- `(*size)--;`

Operators

```
( ) [ ] -> . ++ -- * / %
! ~ ++ -- + - & (type) sizeof
+ -
<< >>
< <= > >=
== !=
&
^
|
&&
||
?:
= += -= *= /= %= &= ^= != <<= >>=
,
```

Associativity

```
left to right
right to left
left to right
left to right
left to right
left to right
left to right
left to right
left to right
left to right
right to left
right to left
left to right
```

Referencing Nonexistent Variables

- Forgetting that local variables disappear when a function returns

```
int *foo () {  
    int val;  
  
    return &val;  
}
```


Freeing Blocks Multiple Times

- Nasty!

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

Referencing Freed Blocks

■ Evil!

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

Failing to Free Blocks (Memory Leaks)

- Slow, long-term killer!

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

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;
}
```

Dealing With Memory Bugs

■ Debugger: `gdb`

- Good for finding bad pointer dereferences
- Hard to detect the other memory bugs

■ Data structure consistency checker

- Runs silently, prints message only on error
- Use as a probe to zero in on error

■ Binary translator: `valgrind`

- Powerful debugging and analysis technique
- Rewrites text section of executable object file
- Checks each individual reference at runtime
 - Bad pointers, overwrites, refs outside of allocated block

■ `glibc malloc` contains checking code

- `setenv MALLOC_CHECK_ 3`