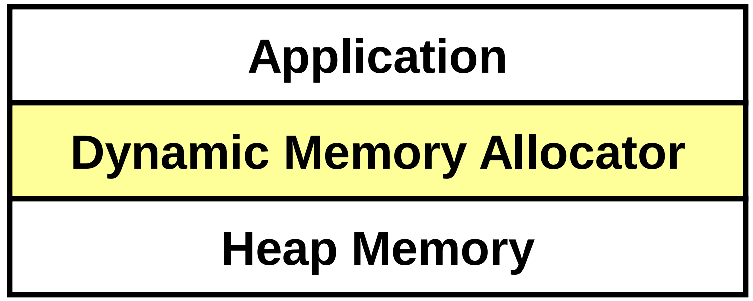
Dynamic Memory Allocation

Harsh Reality

- *Memory Matters*

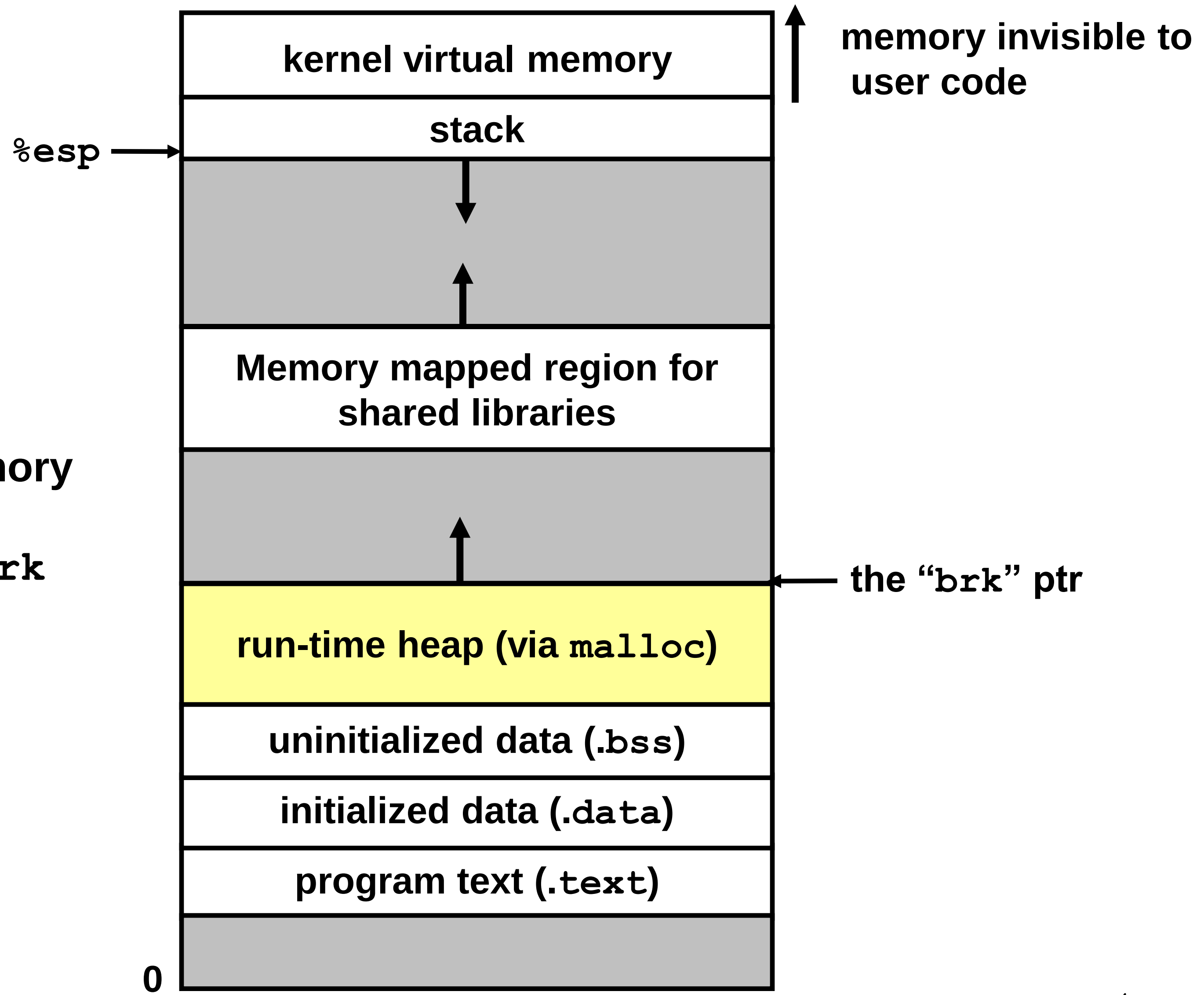
- Memory is not unbounded (Statically reserving the maximum amount of global memory is NOT good!)
  - It must be allocated and managed
  - Many applications are memory dominated
    - Especially those based on complex, graph algorithms
- Memory referencing bugs especially pernicious
  - Effects are distant 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 Memory Allocation



- Dynamic Memory Allocator allocates a memory block only when necessary
- 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

Process Memory Image



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

Malloc Package

- `#include <stdlib.h>`
- `void *malloc(size_t size)`
  - If successful:
    - Returns a pointer to a memory block of at least `size` bytes, (typically) aligned to 8-byte boundary.
    - If `size == 0`, returns NULL
  - If unsuccessful: returns NULL (0) and sets `errno`.

- `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`.

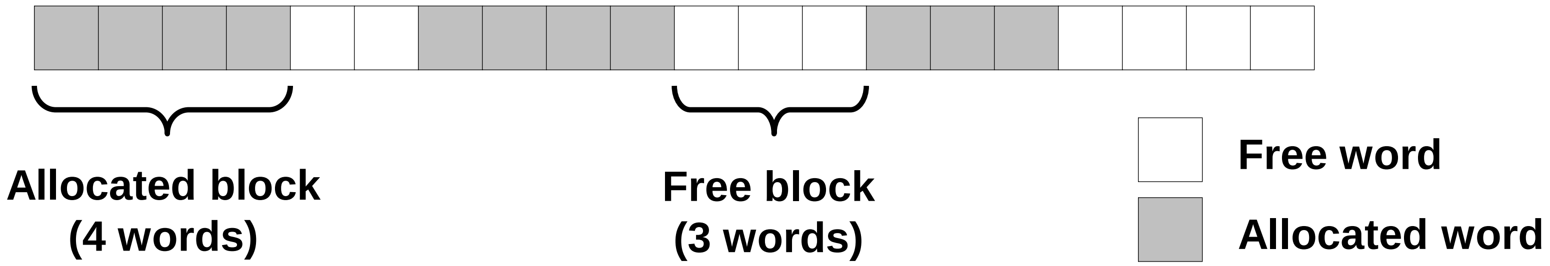
- `void *realloc(void *p, size_t size)`
  - Changes size of block `p` and returns pointer to new block.
  - Contents of new block unchanged up to min of old and new size.

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 */
}
```

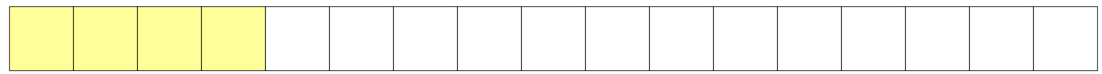
Assumptions

- Assumptions made in this lecture
  - Memory is word addressed (each word can hold a pointer)

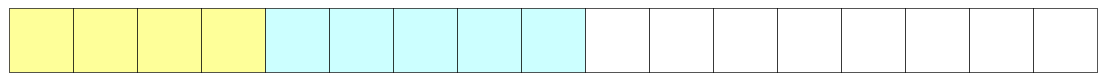


Allocation Examples

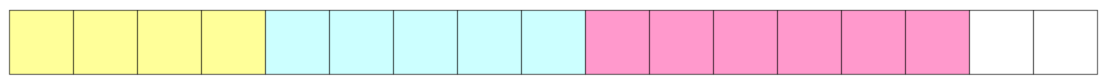
```
p1 = malloc(4)
```



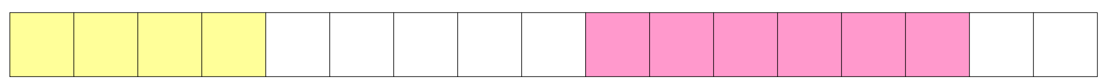
```
p2 = malloc(5)
```



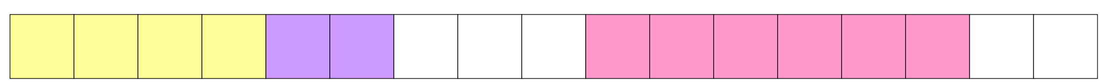
```
p3 = malloc(6)
```



```
free(p2)
```



```
p4 = malloc(2)
```



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”.

Performance Goals: Throughput

- Given some sequence of malloc and free requests:
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$

- Want to maximize throughput and peak memory utilization.
  - These goals are often conflicting

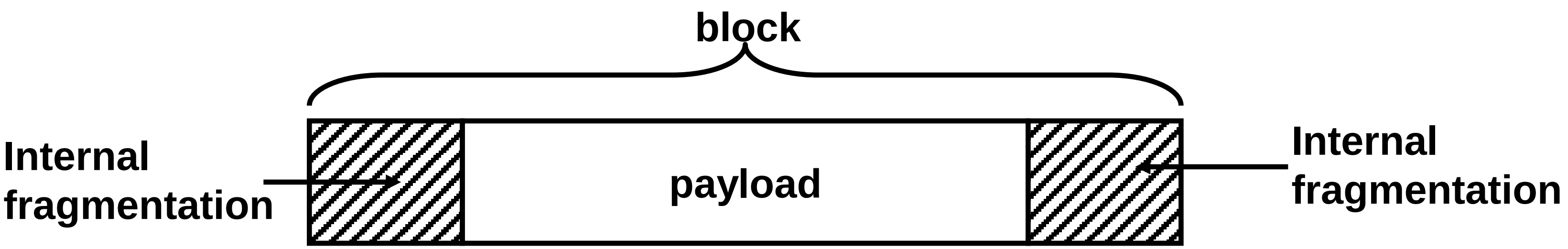
- Throughput:
  - Number of completed requests per unit time
  - Example:
    - 5,000 malloc calls and 5,000 free calls in 10 seconds
    - Throughput is 1,000 operations/second.

Performance Goals: Peak Memory Utilization

- Given some sequence of malloc and free requests:
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$
- *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*
  - Assume that H_k is monotonically nondecreasing
- *Def: Peak memory utilization:*
  - After k requests, *peak memory utilization* is:
    - $U_k = (\max_{i<k} P_i) \; / \; H_k$

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.

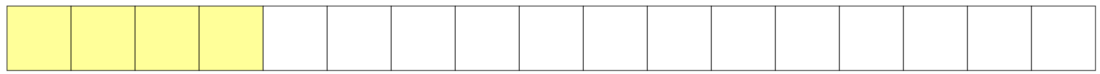


  - Caused by overhead of maintaining heap data structures, padding for alignment purposes, or explicit policy decisions (e.g., not to split the block).
  - Depends only on the pattern of *previous* requests, and thus is easy to measure.

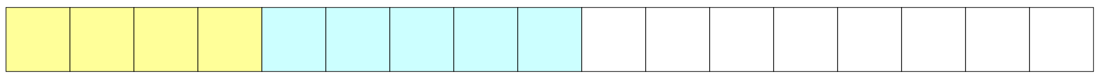
External Fragmentation

Occurs when there is enough aggregate heap memory, but no single free block is large enough

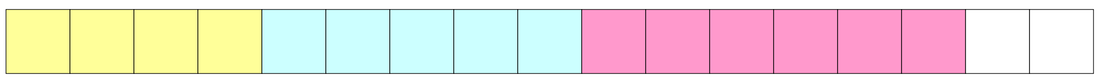
```
p1 = malloc(4)
```



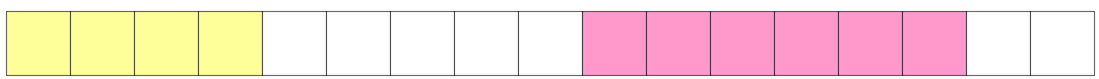
```
p2 = malloc(5)
```



```
p3 = malloc(6)
```



```
free(p2)
```



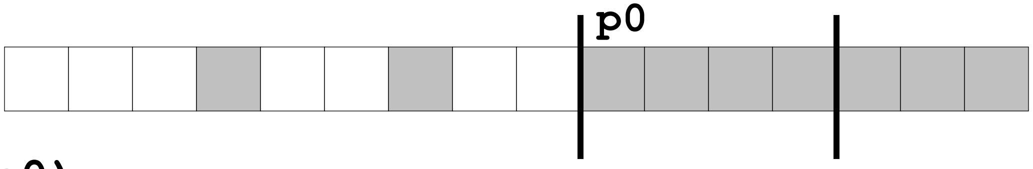
```
p4 = malloc(6)
```

oops!

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

Implementation Issues

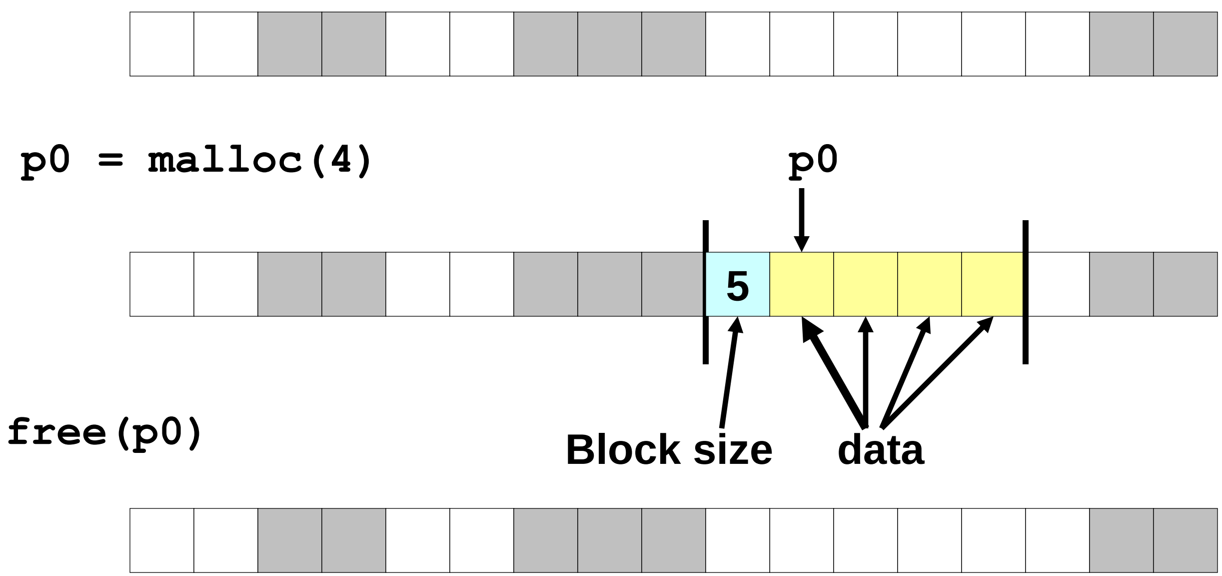
- How do we know how much memory to free just given a pointer? (free(p)?)
- How do we keep track of the free blocks?
- What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in? (Splitting or not?)
- How do we pick a block to use for allocation -- many might fit? (Placement issue)
- How do we reinsert freed block? (Merge or not?)




```
free(p0)

p1 = malloc(1)
```

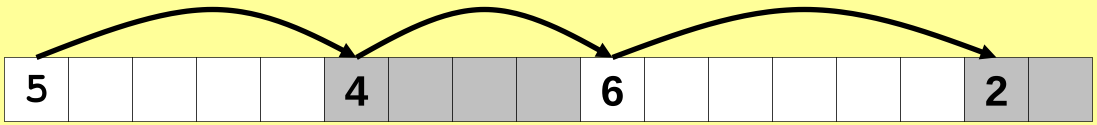
Knowing How Much to Free

- 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

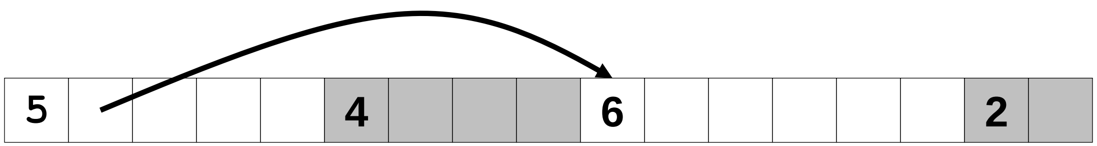


Keeping Track of Free Blocks

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



- *Method 2*: *Explicit list* among the free blocks using pointers within the free blocks (Fast search for a fitting free block)

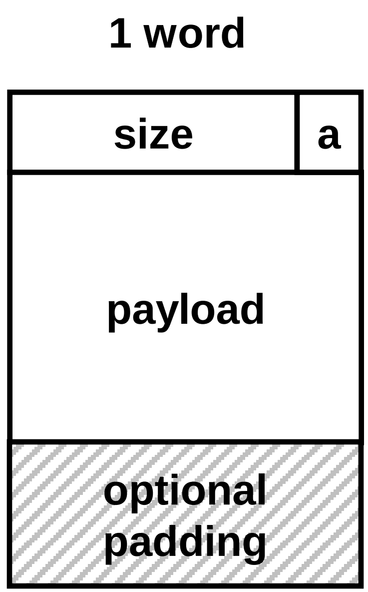


- *Method 3*: *Segregated free list*
  - Different free lists for different size classes (Faster search for fitting free block)
- *Method 4*: Blocks sorted by size
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

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).

Format of allocated and free blocks



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

size: block size

payload: application data (allocated blocks only)

Implicit List: Finding a Free Block

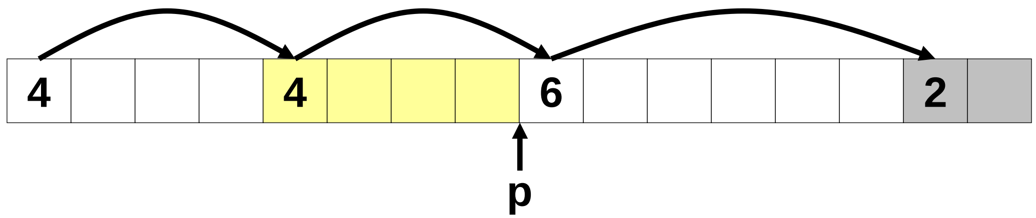
- *First fit:*
  - Search list from beginning, choose first free block that fits

```
p = start;
while ((*p & 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" (small free blocks) at beginning of list (Very likely search many small blocks from the beginning until find a large enough block later of the list)
- *Next fit:*
  - Like first-fit, but search list from location of end of previous search (Likely to find a fit in the remainder of the previous block)
  - 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 because of exhaustive search

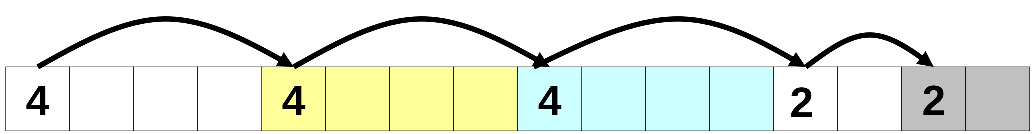
Implicit List: Allocating in Free Block

- Allocating in a free block – *splitting*
  - Since allocated space might be smaller than free space, we might want to split the block




```
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)`

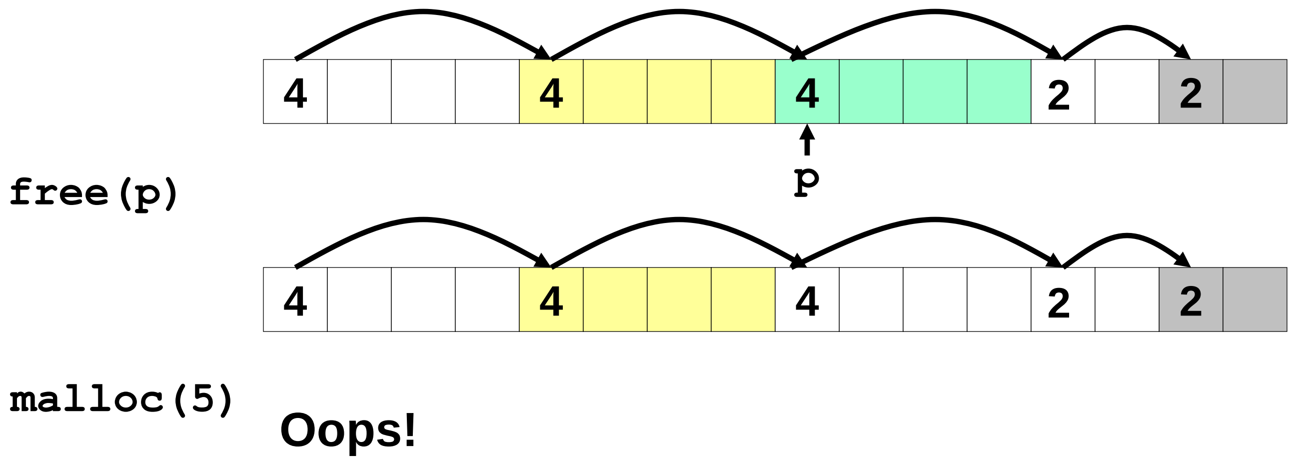


Implicit List: Freeing a Block

- Simplest implementation:
  - Only need to clear allocated flag

    ```
    void free_block(ptr p) { *p = *p & -2}
    ```

  - But can lead to "false fragmentation"

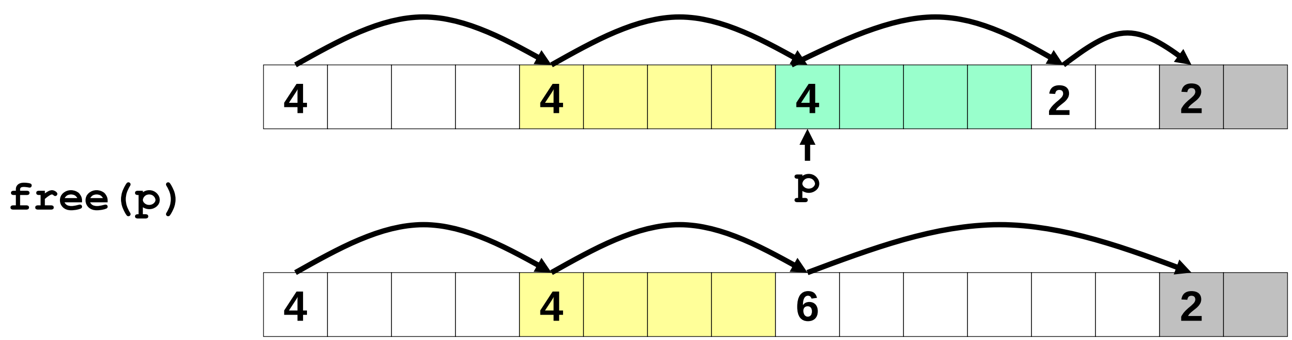


There is enough free space, but the allocator won't be able to find it

Implicit List: Coalescing

- Join (*coelesce*) with next and/or previous block if they are free
  - Coalescing with next block

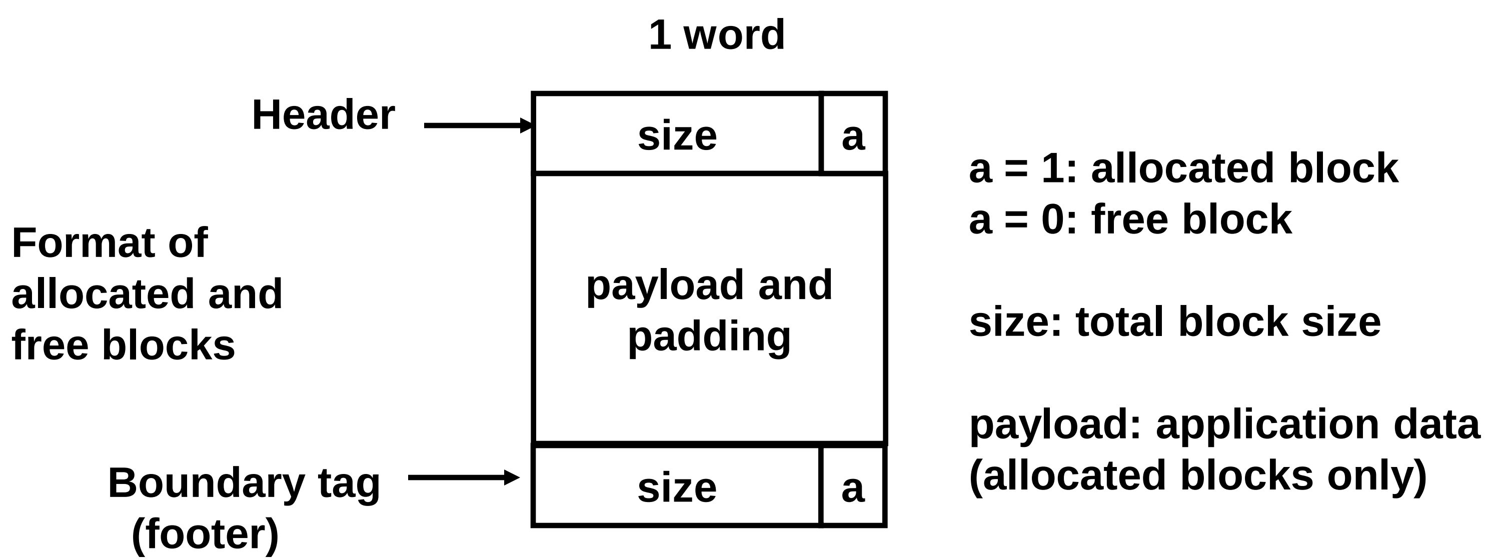
```
void free_block(ptr p) {
    *p = *p & -2;            // clear allocated flag
    next = p + *p;           // find next block
    if ((*next & 1) == 0)
      *p = *p + *next;      // add to this block if
}                           //    not allocated
```

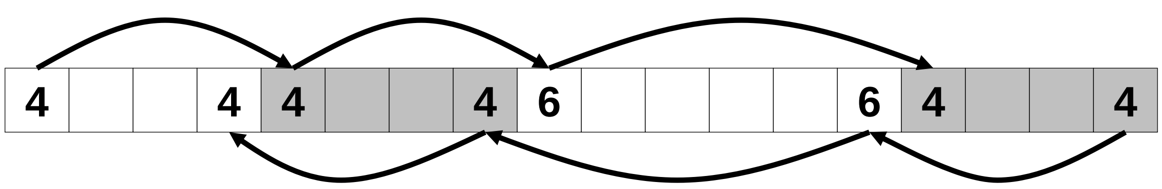


  - But how do we coalesce with previous block?

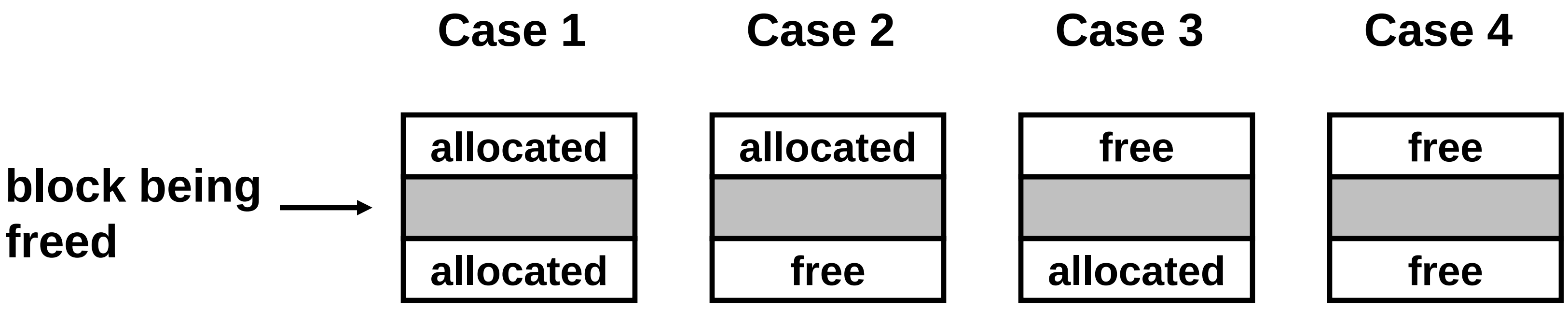
Implicit List: Bidirectional Coalescing

- *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!

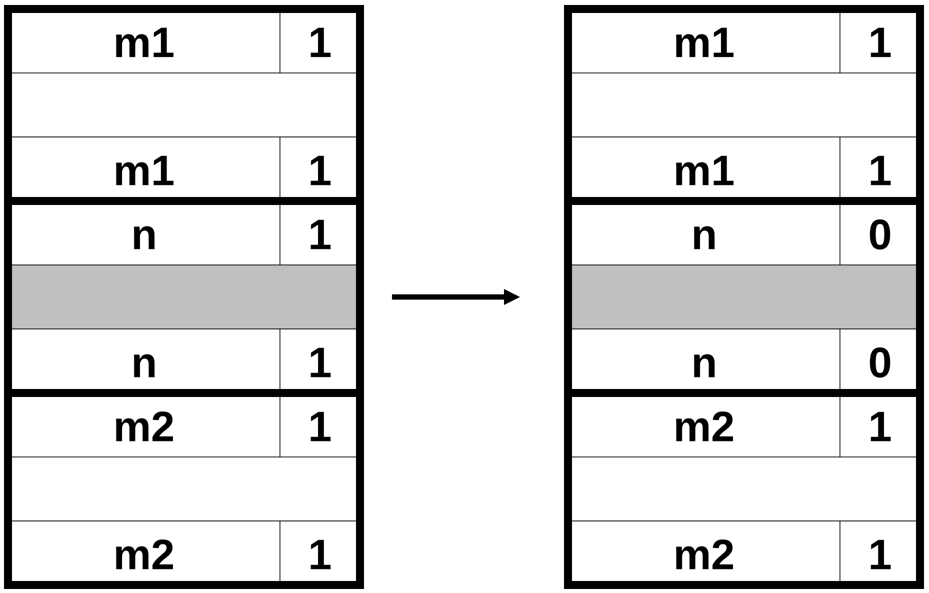




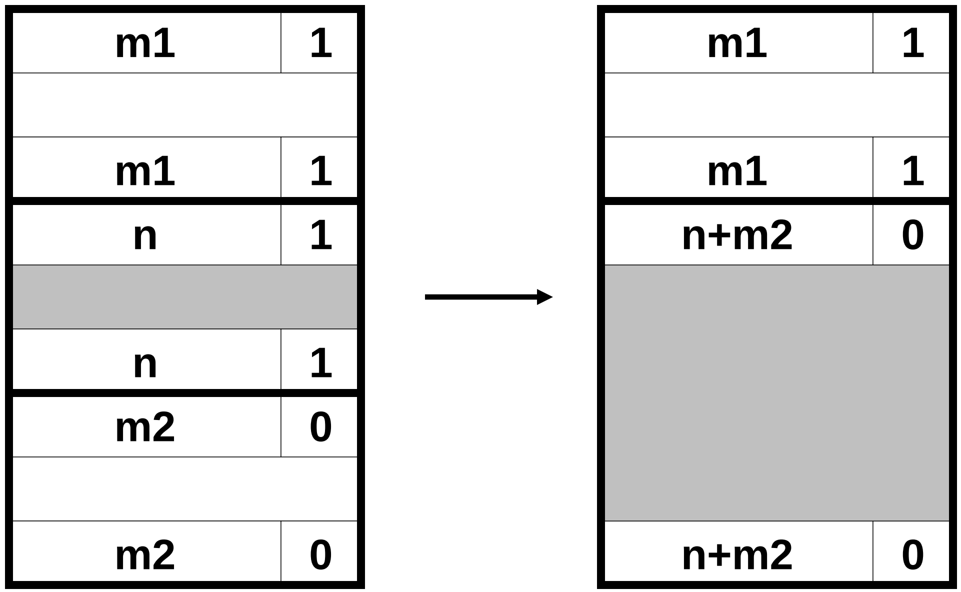
Constant Time Coalescing



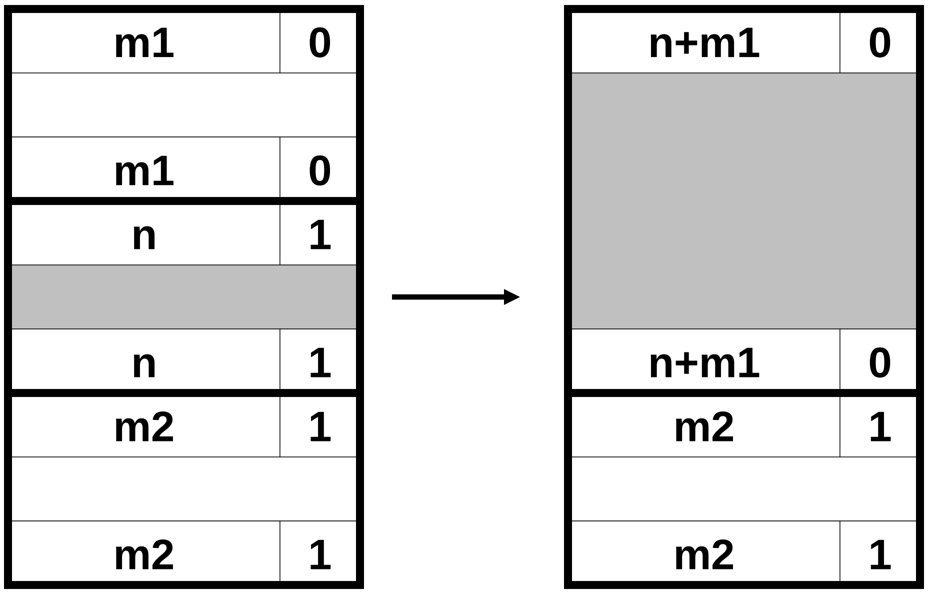
Constant Time Coalescing (Case 1)



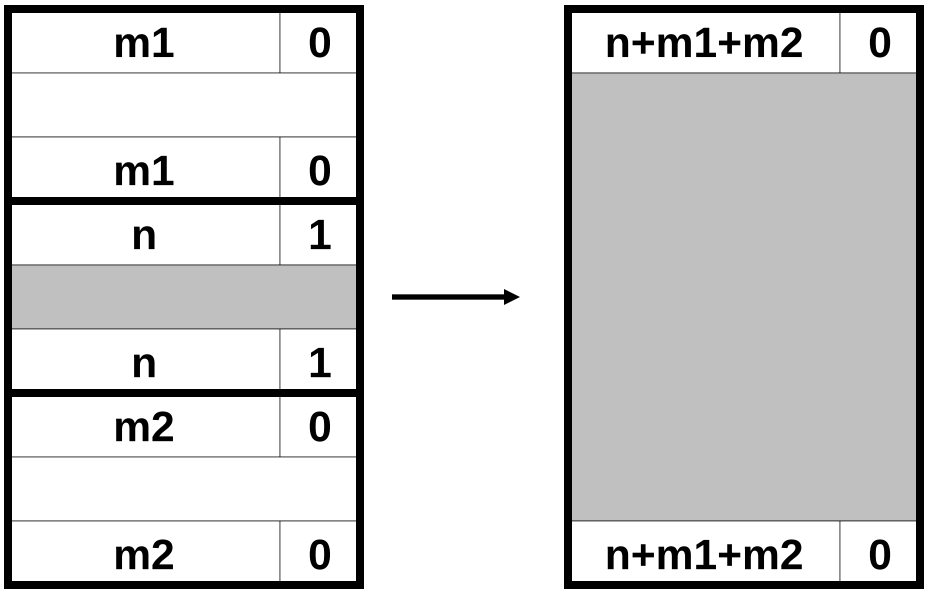
Constant Time Coalescing (Case 2)



Constant Time Coalescing (Case 3)



Constant Time Coalescing (Case 4)



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.

Implicit Lists: Summary

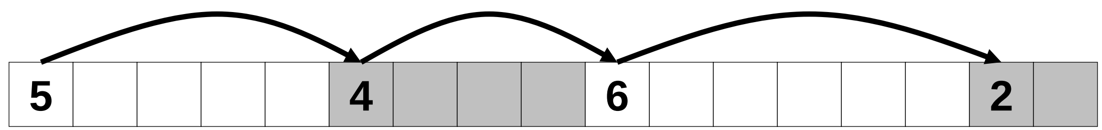
- Implementation: very simple
- Allocate: linear time worst case (in number of TOTAL blocks)
- Free: constant time worst case -- even with coalescing
- Memory usage: will depend on placement policy
  - First fit, next fit or best fit

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

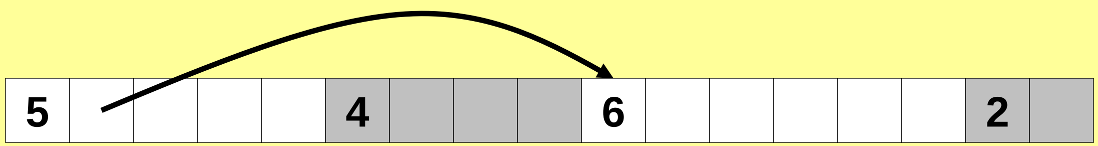
- However, the concepts of splitting and boundary tag coalescing are general to *all* allocators.

Keeping Track of Free Blocks

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

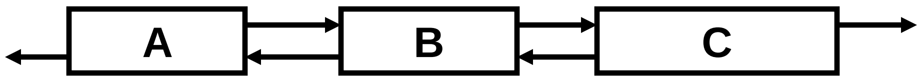


- *Method 2*: Explicit list among the free blocks using pointers within the free blocks

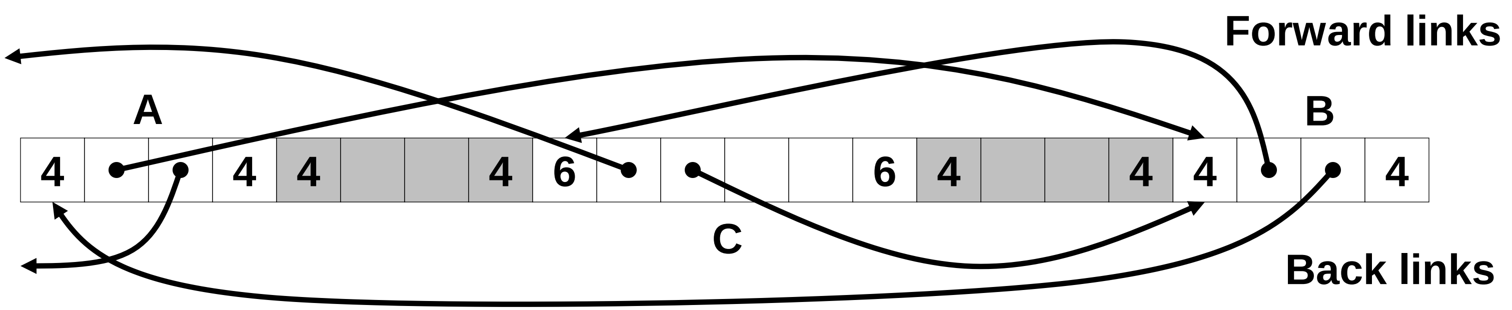


- *Method 3*: Segregated free lists
  - Different free lists for different size classes
- *Method 4*: Blocks sorted by size (not discussed)
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

Explicit Free Lists

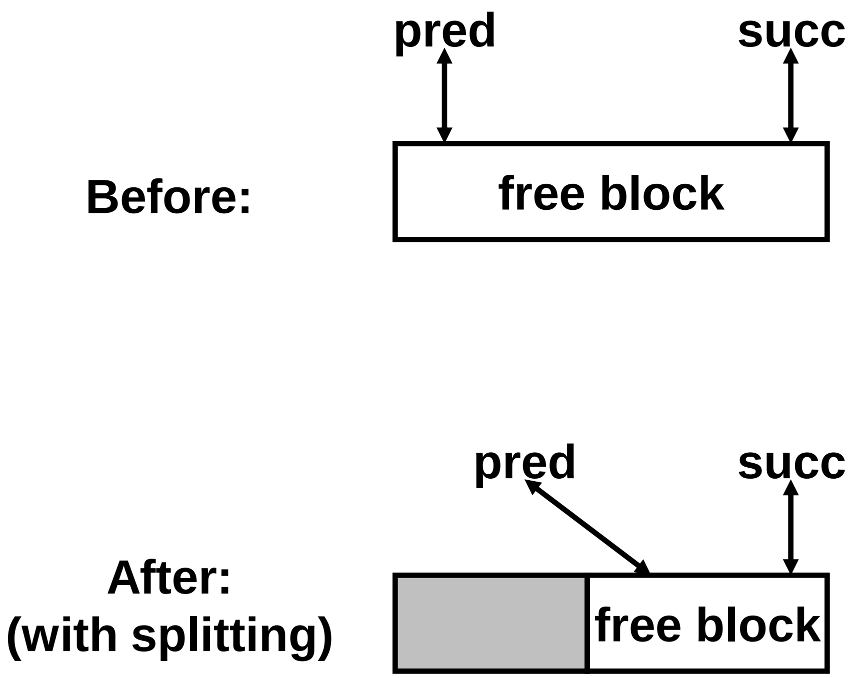


- Use data space for link pointers
  - Typically doubly linked
  - Still need boundary tags for coalescing



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

Allocating From Explicit Free Lists

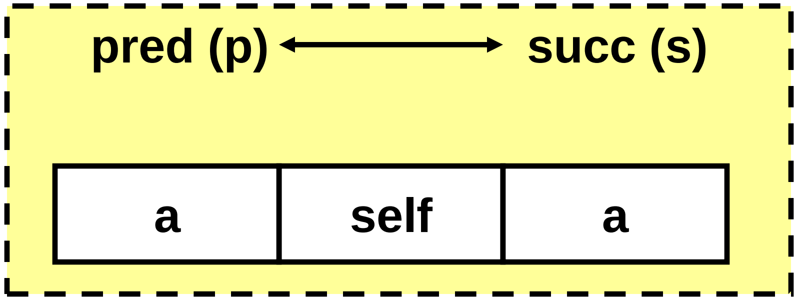


Freeing With Explicit Free Lists

- *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
      - i.e. addr(pred) < addr(curr) < addr(succ)
    - Con: requires search
    - Pro: studies suggest fragmentation is better than LIFO

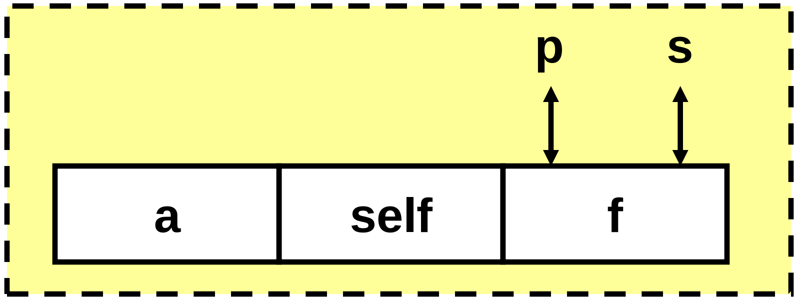
Freeing With a LIFO Policy

- Case 1: a-a-a
  - Insert self at beginning of free list

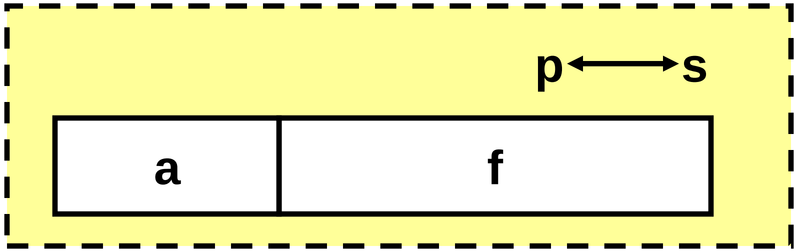


- Case 2: a-a-f
  - Splice out next, coalesce self and next, and add to beginning of free list
-

before:



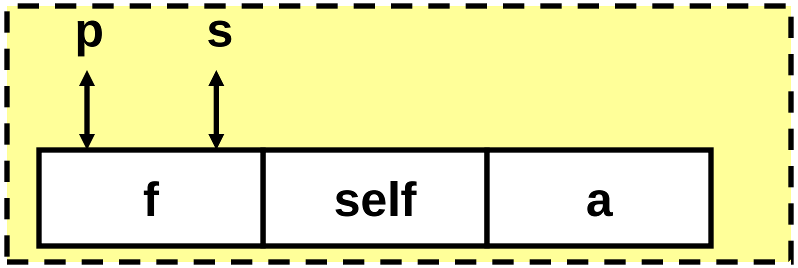
after:



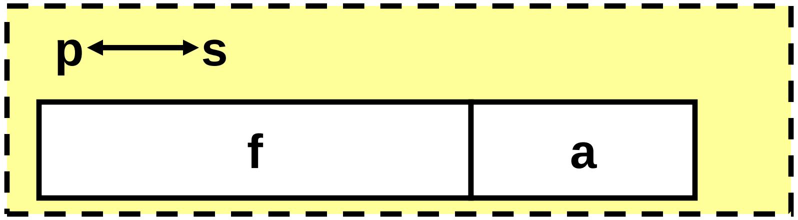
Freeing With a LIFO Policy (cont)

- Case 3: f-a-a
  - Splice out prev, coalesce with self, and add to beginning of free list

before:

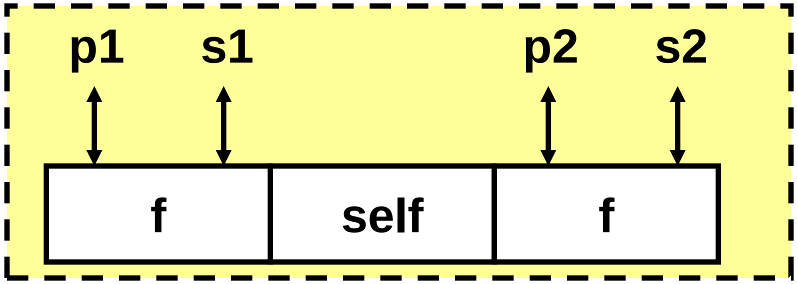


after:

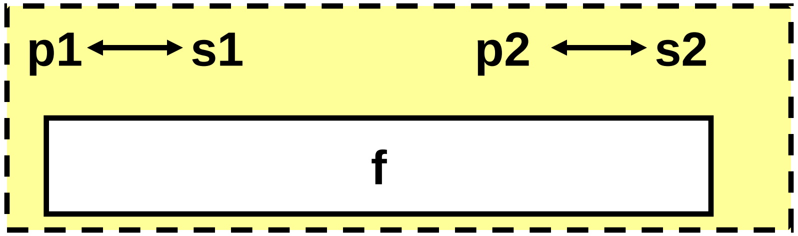


- Case 4: f-a-f
  - Splice out prev and next, coalesce with self, and add to beginning of list

before:



after:

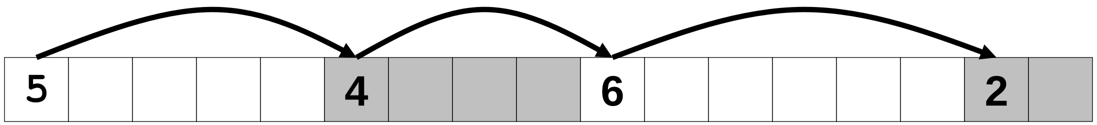


Explicit List Summary

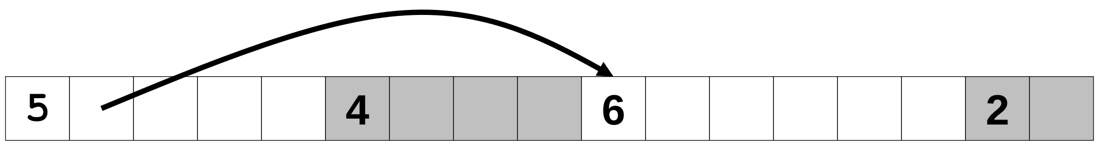
- Comparison to implicit list:
  - Allocate is linear time in number of FREE blocks instead of total blocks -- much faster allocates when most of the memory is full
  - Slightly more complicated allocate and free since needs to splice blocks in and out of the list
  - Some extra space for the links (2 extra words needed for each "free" 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

Keeping Track of Free Blocks

- <u>*Method 1*</u>: *Implicit list* using lengths -- links all blocks



- <u>*Method 2*</u>: *Explicit list* among the free blocks using pointers within the free blocks

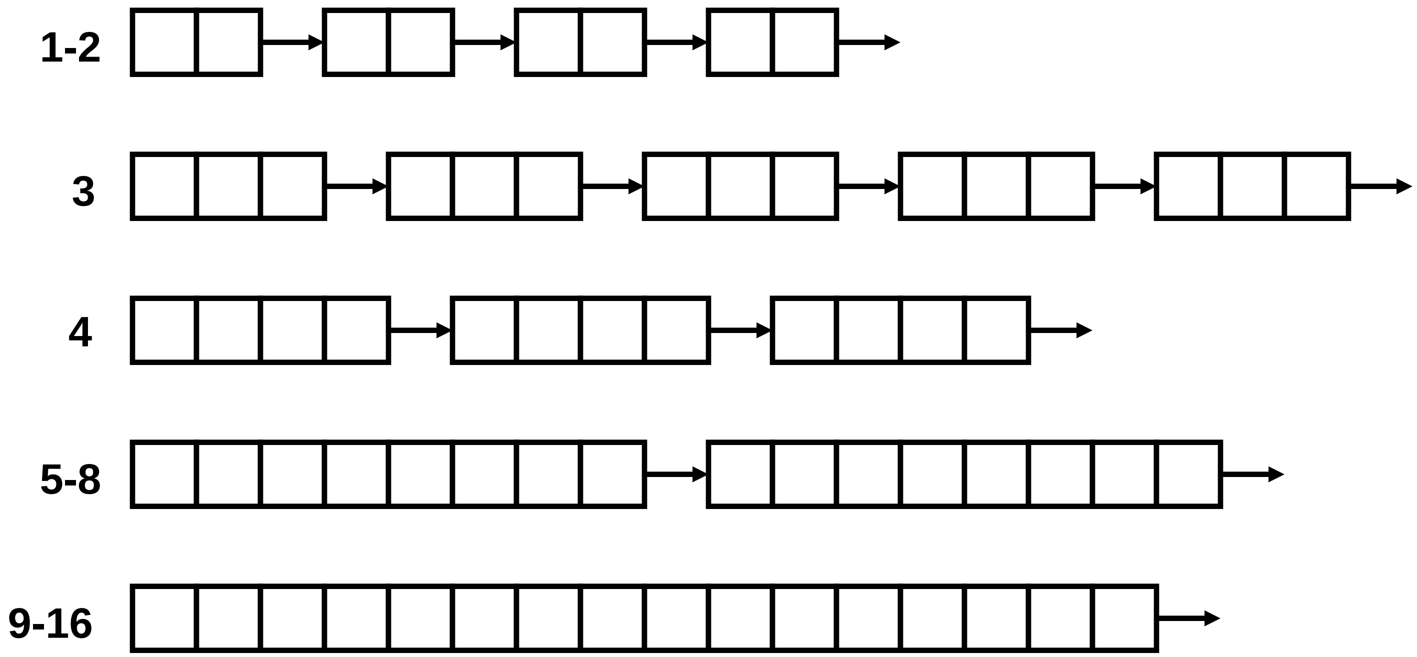


- <u>*Method 3*</u>: *Segregated free list*
  - Different free lists for different size classes

- <u>*Method 4*</u>: 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

Segregated Storage

- Each *size class* has its own collection of blocks



  - Often have separate size class for every small size (2,3,4,…)
  - For larger sizes typically have a size class for each power of 2

Simple Segregated Storage

- Separate heap and free list for each size class (each block in the same list has the same size)
- No splitting
- To allocate a block of size n:
  - If free list for size n is not empty,
    - allocate first block on list (note, list can be implicit or explicit)
  - If free list is empty,
    - get a new page
    - create new free list from all blocks in page
    - allocate first block on list
  - Constant time
- To free a block:
  - Add to free list

- Tradeoffs:
  - Fast, but can fragment badly

Segregated Fits (Improved Segregated Storage)

- 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
- To free a block:
  - Coalesce and place on appropriate list (optional)
- Performance
  - Faster search than sequential fits (i.e., log time for power of two size classes)
  - Controls fragmentation of simple segregated storage (Utilization performance is similar to Best Fit)
  - Coalescing can increase search times for free (free to which size class?)
    - Deferred coalescing can help

For More Info on Allocators

- Donald. 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

Implicit Memory Management: Garbage Collection

- *Garbage collection:* automatic reclamation of heap-allocated storage -- application never has to free

```
void foo() {
   int *p = malloc(128);
   return; /* p block is now garbage */
}
```

- Common in functional languages, scripting languages, and modern object oriented languages:
  - Lisp, ML, Java, Perl, Mathematica,
- Variants (conservative garbage collectors) exist for C and C++
  - Cannot collect all garbage

Garbage Collection

- How does the memory manager know when memory can be freed?
  - In general we cannot know what is going to be used in the future since it depends on conditionals
  - But we can tell that certain blocks cannot be used if there are no pointers to them

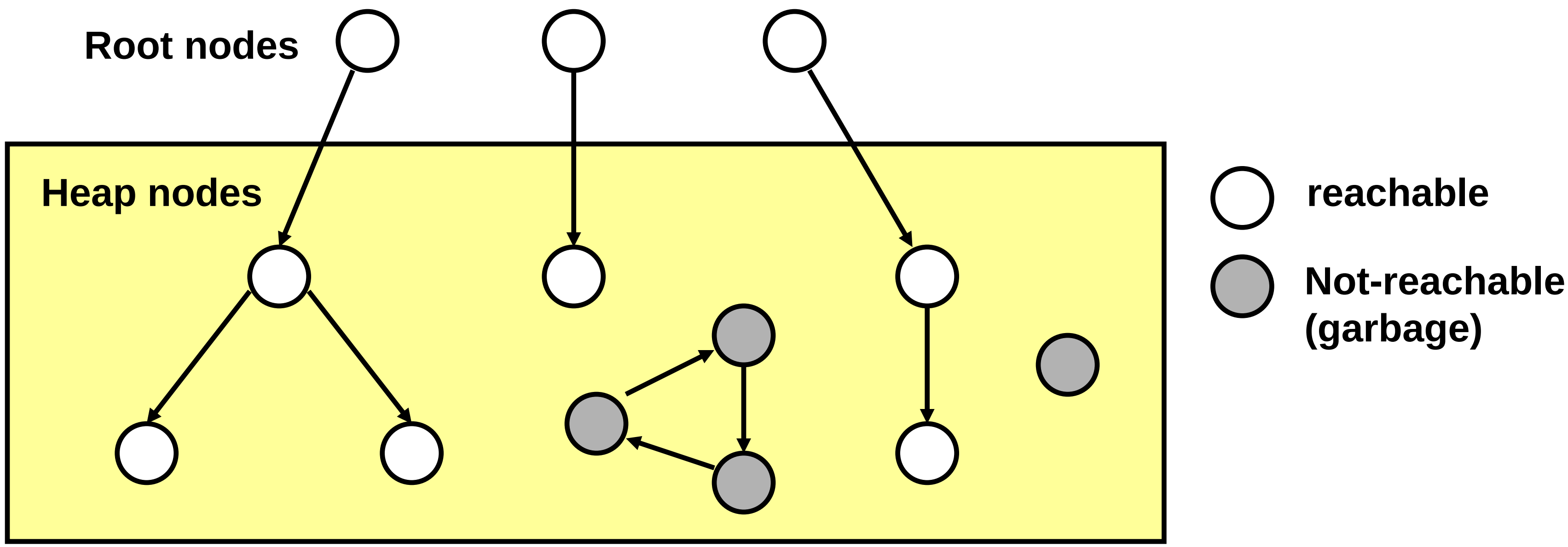
Classical GC algorithms

- Mark and sweep collection (McCarthy, 1960)
  - Does not move blocks (unless you also “compact”)
- Reference counting (Collins, 1960)
  - Does not move blocks (not discussed)
- Copying collection (Minsky, 1963)
  - Moves blocks (not discussed)

- For more information, see *Jones and Lin, “Garbage Collection: Algorithms for Automatic Dynamic Memory”, John Wiley & Sons, 1996.*

Memory as a Graph

- We view memory as a directed graph
  - Each block is a node in the graph
  - Each pointer is an edge in the graph
  - Locations not in the heap that contain pointers into the heap are called *root* nodes (e.g. registers, locations on the stack, global variables)



- A node (block) is *reachable* if there is a path from any root to that node.
- Non-reachable nodes are *garbage* (never needed by the application)

Assumptions For This Lecture

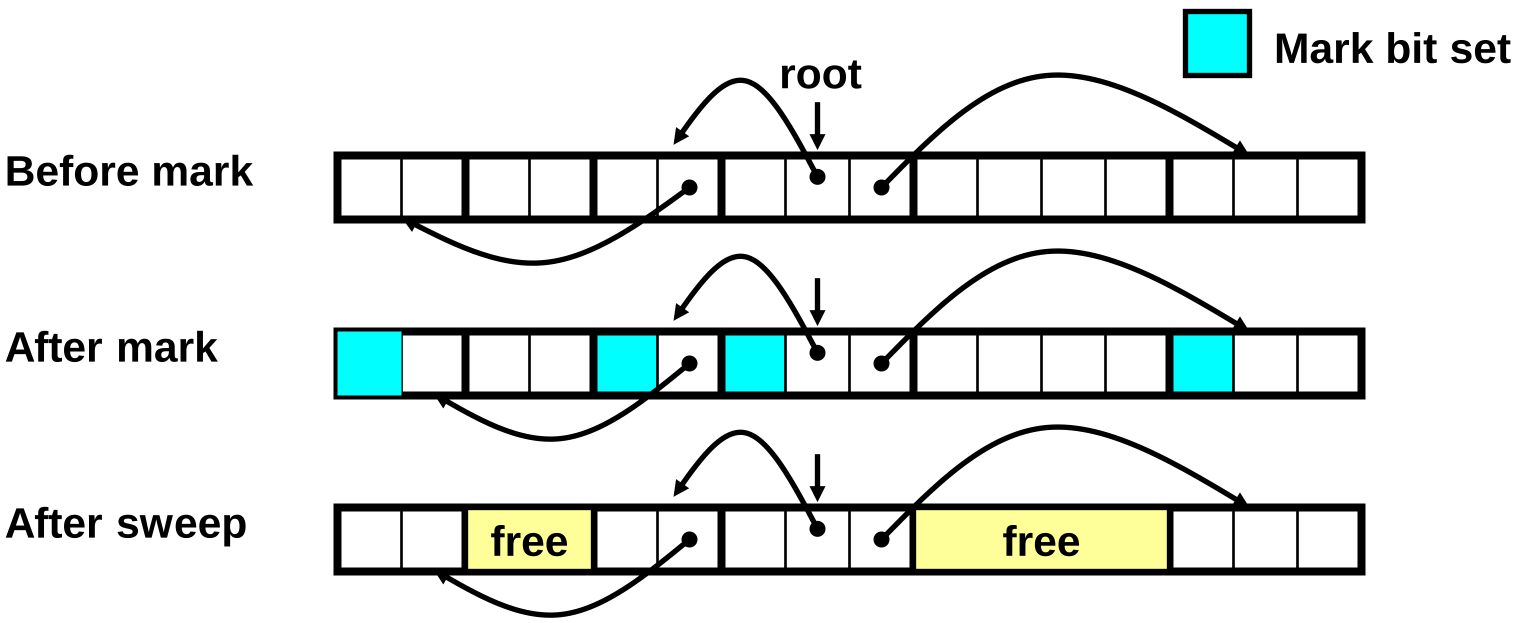
- Application
  - `new(n)`: returns pointer to new block with all locations cleared
  - `read(b,i)`: read location `i` of block `b` `into register`
  - `write(b,i,v)`: write `v` into location `i` of block `b`

- Each block will have a header word
  - addressed as `b[-1]`, for a block `b`
  - Used for different purposes in different collectors

- Instructions used by the Garbage Collector
  - `is_ptr(p)`: determines whether `p` is a pointer
  - `length(b)`: returns the length of block `b`, not including the header
  - `get_roots()`: returns all the roots

Mark and Sweep Collecting

- Can build on top of malloc/free package
  - Allocate using malloc until you "run out of space"
- When out of space:
  - Use extra *mark bit* in the head of each block
  - *Mark:* Start at roots and set mark bit on all reachable memory
  - *Sweep:* Scan all blocks and free blocks that are "allocated" but "not marked"



Mark and Sweep (cont.)

Mark using depth-first traversal of the memory graph

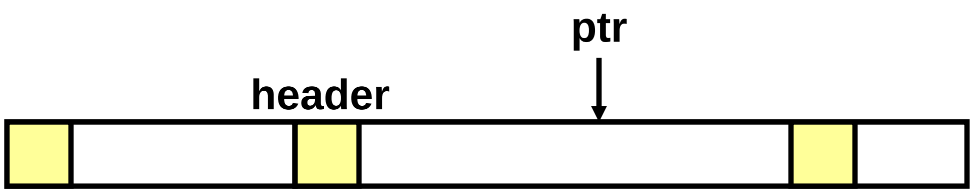
```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // do nothing if not pointer
   if (markBitSet(p)) return        // check if already marked
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)   // mark all children
     mark(p[i]);
   return;
}
```

Sweep using lengths to find next block

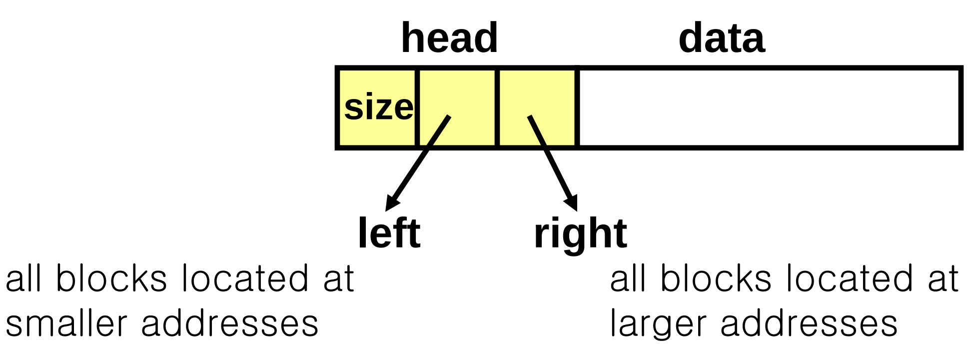
```
ptr sweep(ptr p, ptr end) {
   while (p < end) {
      if markBitSet(p)
         clearMarkBit();
      else if (allocateBitSet(p))
         free(p);
      p += length(p);
}
```

Conservative Mark and Sweep in C

- A conservative collector for C programs
  - `Is_ptr()` determines if a word is a pointer by checking if it points to an allocated block of memory.
  - But, in C pointers can point to the middle of a block.



- So how do we find the beginning of the block?
  - Can use balanced tree to keep track of all allocated blocks where the key is the location
  - Balanced tree pointers can be stored in header (use two additional words)



So, we can traverse the tree to see if the pointer is pointing a valid location of a allocated block

Memory-Related Bugs

- 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

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

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;
}
```

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));
}
```

should be sizeof(int *)

This is a problem if sizeof(int) != sizeof(int *)

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));
}
```

Overwriting Memory

- Not checking the max string size

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

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

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);
}
```

Intent is to decrement the integer value pointed by the pointer “size”

So, should be (*size)--

Overwriting Memory

- Misunderstanding pointer arithmetic

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

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

   return p;
}
```

should be
p++

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;
}
```

Don't make memory related bugs

- Deep understanding on the memory management mechanism will help!