Dynamic Memory Allocation

Computer Systems Organization (Spring 2017)
CSCI-UA 201, Section 3

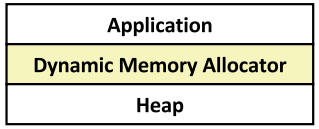
Instructor: Joanna Klukowska

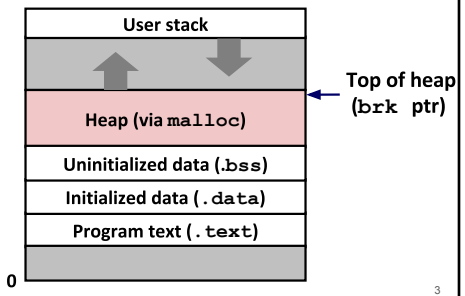
Slides adapted from
Randal E. Bryant and David R. O'Hallaron (CMU)
Mohamed Zahran (NYU)

Basic Concepts

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.**





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

The `malloc` Package

```
#include <stdlib.h>
```

`void *malloc(size_t size)`

- Successful:
  - Returns a pointer to a memory block of **at least size bytes** aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
  - If size == 0, returns NULL
- Unsuccessful: returns NULL (0) and sets errno

`void free(void *p)`

- Returns the block pointed at by p to the pool of available memory
- p must come from a previous call to `malloc, calloc` or `realloc`

Other functions

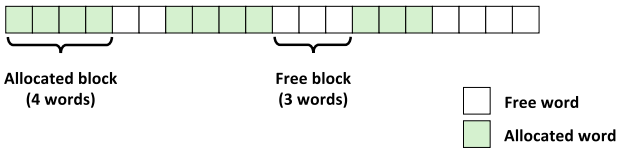
- **`calloc:`** Version of malloc that initializes allocated block to zero.
- **`realloc:`** Changes the size of a previously allocated block.
- **`sbrk:`** Used internally by allocators to grow or shrink the heap

`malloc` Example

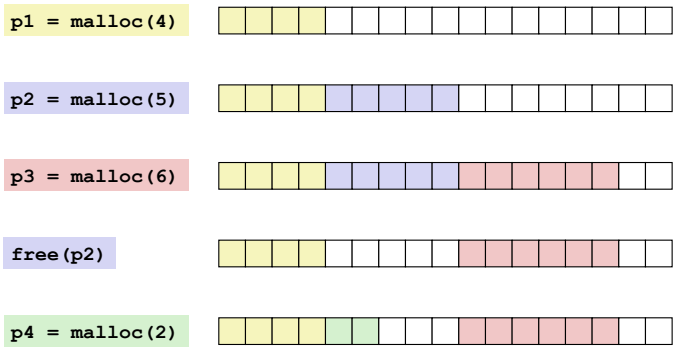
```
#include <stdio.h>
#include <stdlib.h>

void foo(int n) {
    int i, *p;

    /* Allocate a block of n ints */
    p = (int *) malloc(n * sizeof(int));
    if (p == NULL) {
        perror("malloc");
        exit(0);
    }

    /* Initialize allocated block */
    for (i=0; i<n; i++)
       p[i] = i;


    /* Return allocated block to the heap */
    free(p);
}
```


Assumptions Made in This Lecture

- Memory is word addressed.
- Words are int-sized.



Allocation Example



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 (x86) or 16-byte (x86-64) alignment 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

Performance Goal: Throughput

- **Given some sequence of `malloc` and `free` requests:**
  - $R_0, R_1, ..., R_k, ..., R_{n-1}$
- **Goals: 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 Goal: Peak Memory Utilization

- **Given some sequence of `malloc` and `free` requests:**
  - $R_0, R_1, ..., R_k, ..., R_{n-1}$
- ***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 H_k***
  - Assume H_k is monotonically nondecreasing
    - i.e., heap only grows when allocator uses `sbrk`
- ***Def: Peak memory utilization after k+1 requests***
  - $U_k = (max_{i<=k} P_i) / H_k$

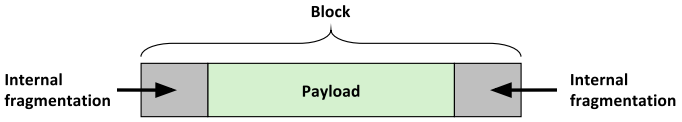
Fragmentation

Poor memory utilization caused by *fragmentation*

- ***internal*** fragmentation
- ***external*** fragmentation

Internal Fragmentation

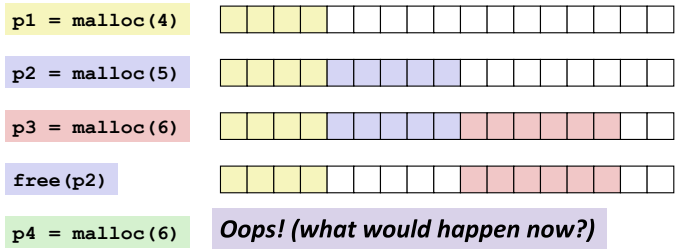
- For a given block, internal fragmentation occurs if payload is smaller than block size



- 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

External Fragmentation

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



- **Depends on the pattern of future requests**
  - Thus, difficult to measure

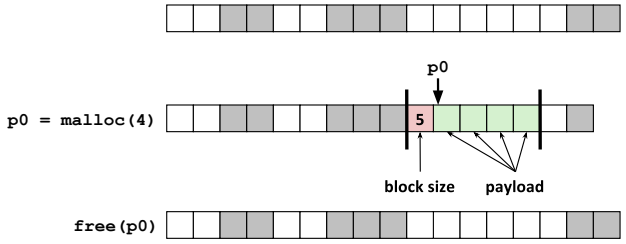
Implementation Issues

- **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?**

Knowing How Much to Free

Standard method

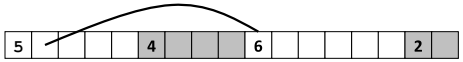
- Keep the length of a block in the word preceding the block.
  - This word is often called the ***header field*** or ***header***
- Requires an extra word for every allocated block



Keeping Track of Free Blocks

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

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



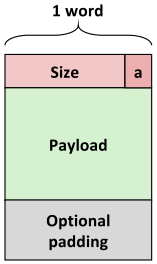
- Method 3: Segregated free list
  - Different free lists for different size classes
- Method 4: Blocks sorted by size
  - Can use a balanced binary tree with pointers within each free block, and the length used as a key

Implicit Free List

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, must mask out this bit

Format of allocated and free blocks

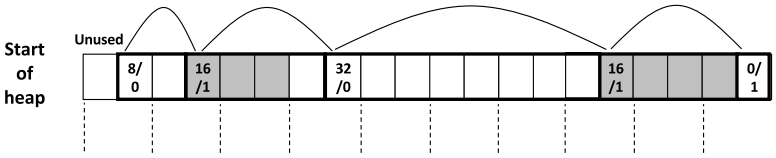


a = 1: Allocated block
a = 0: Free block

Size: block size

Payload: application data (allocated blocks only)

Detailed Implicit Free List Example



Double-word aligned

Allocated blocks: shaded
Free blocks: unshaded
Headers: labeled with size in bytes/allocated bit

*Assume 8-byte (2 word) align boundary.

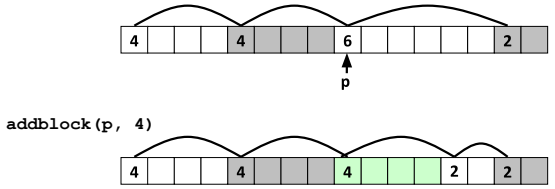
Implicit List: Finding a Free Block

- **First fit**:
  - Search list from beginning, choose **first** free block that fits
  - 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 improves memory utilization
  - Will typically run slower than first fit

Implicit List: Allocating in Free Block

Allocating in a free block: *splitting*

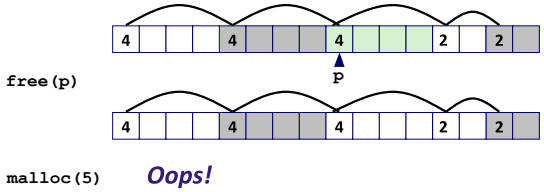
- Since allocated space might be smaller than free space, we might want to split the block



Implicit List: Freeing a Block

Simplest implementation:

- Need only clear the "allocated" flag
- Can lead to "false fragmentation"

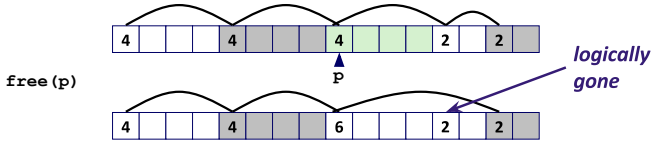


There is enough free space, but the allocator won't be able to find it
(since it sees a block of 4 and block of 2, not a block of 5).

Implicit List: Coalescing

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

- Coalescing with next block

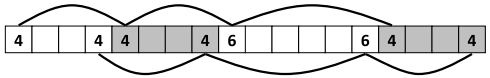


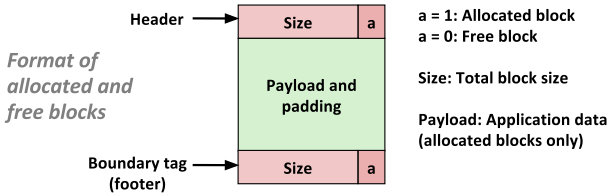
- But how do we coalesce with *previous* block?

Implicit List: Bidirectional Coalescing

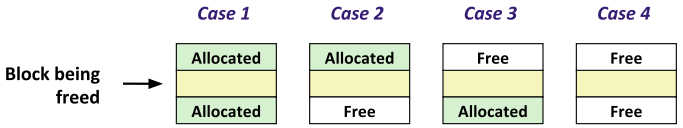
Boundary tags [Knuth73]

- Replicate size/allocated word at "bottom" (end) of free blocks
- Allows us to traverse the "list" backwards, but requires extra space
- Important and general technique!

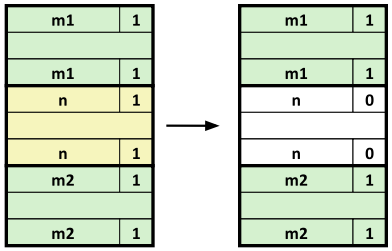




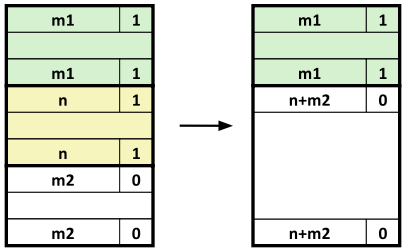
Constant Time Coalescing



Constant Time Coalescing (Case 1)



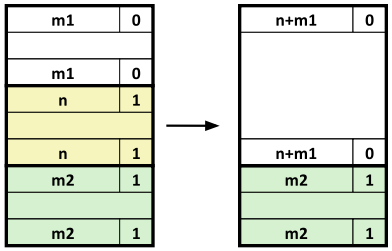
Constant Time Coalescing (Case 2)



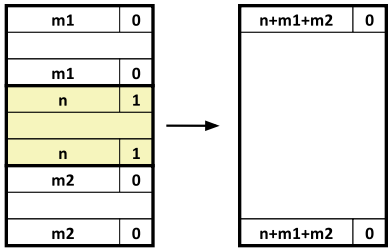
What do we do, if the next block is free as well?

- Not possible if we always coalesce.

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 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

Implicit Lists: Summary

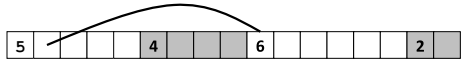
- Implementation: very simple
- Allocate cost:
  - linear time worst case
- Free cost:
  - constant time worst case
  - even with coalescing
- Memory usage:
  - will depend on placement policy
  - First-fit, next-fit or best-fit
- Not used in practice for malloc/free because of linear-time allocation
  - used in many special purpose applications
- However, the concepts of splitting and boundary tag coalescing are general to all allocators

Explicit Free List

Keeping Track of Free Blocks

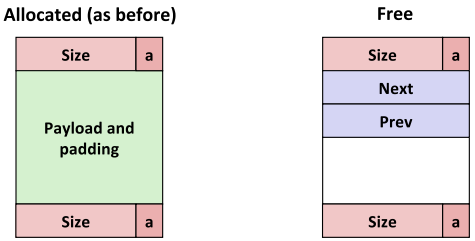
- Method 1: **Implicit list using length—links all blocks**

- 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 binary tree with pointers within each free block, and the length used as a key

Explicit Free Lists

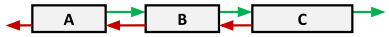


Maintain list(s) of *free* blocks, not *all* blocks

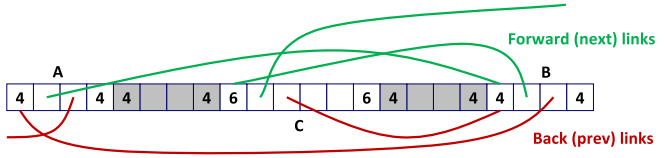
- The "next" free block could be anywhere
  - So we need to store forward/back pointers, not just sizes
- Still need boundary tags for coalescing
- Luckily we track only free blocks, so we can use payload area

Explicit Free Lists

- **Logically:**

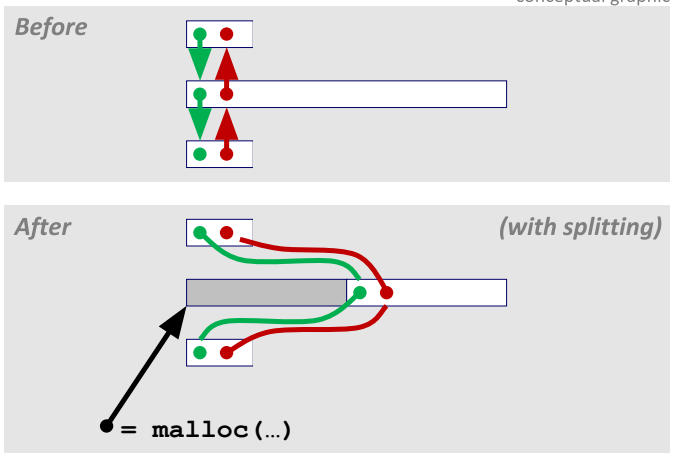


- **Physically: blocks can be in any order**



Allocating From Explicit Free Lists

conceptual graphic

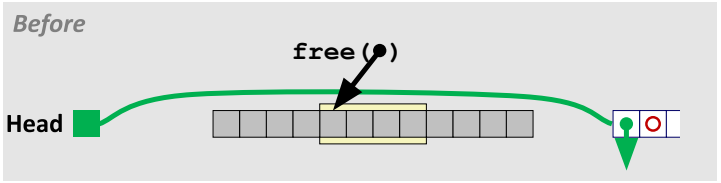


Freeing With Explicit Free Lists

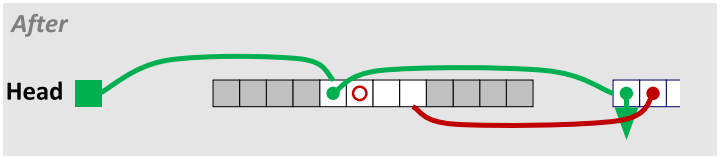
- ***Insertion policy*: Where in the free list do you put a newly freed block?**
- **LIFO (last-in-first-out) policy**
  - Insert freed block at the beginning of the free list
  - ***Pro:*** simple and constant time
  - ***Con:*** studies suggest fragmentation is worse than address ordered
- **Address-ordered 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

Freeing With a LIFO Policy (Case 1)

conceptual graphic

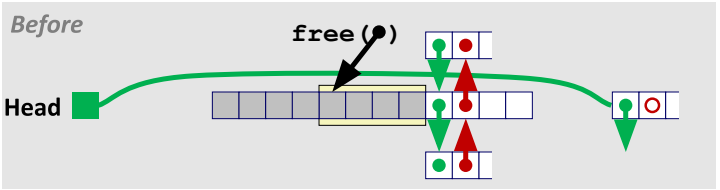


- **Insert the freed block at the root of the list**

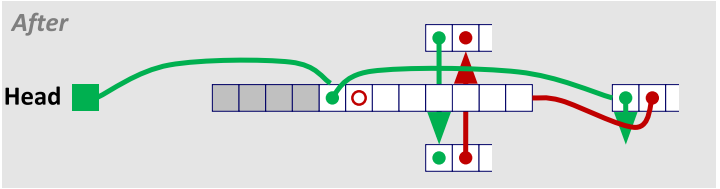


Freeing With a LIFO Policy (Case 2)

conceptual graphic

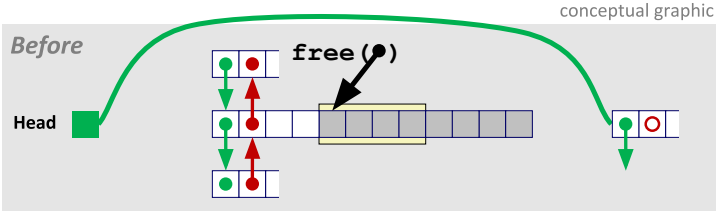


- **Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list**

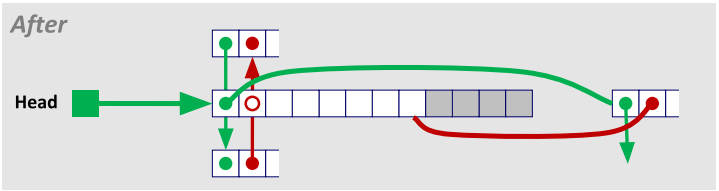


Freeing With a LIFO Policy (Case 3)

conceptual graphic

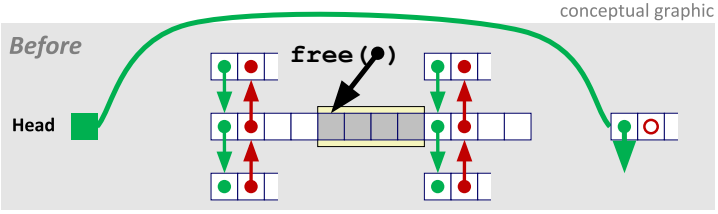


- Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list

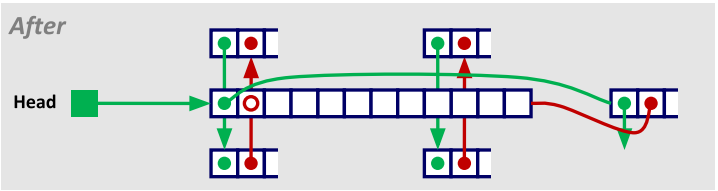


Freeing With a LIFO Policy (Case 4)

conceptual graphic



- Splice out predecessor and successor blocks, coalesce all 3 memory 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 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

Keeping Track of Free Blocks

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

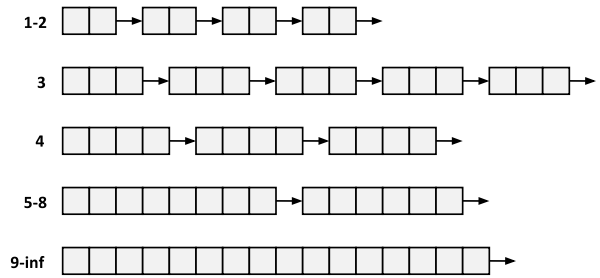
- Method 2: Explicit list among the free blocks using pointers

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

Segregated Free List

Segregated List (Seglist) Allocators

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



- **Often have separate classes for each small size**
- **For larger sizes: One class for each two-power size**

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.

Seglist Allocator (cont.)

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

Garbage Collection

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 many dynamic languages:**
  - Python, Ruby, Java, Perl, ML, Lisp, Mathematica
- **Variants ("conservative" garbage collectors) exist for C and C++**
  - However, cannot necessarily 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
- **Must make certain assumptions about pointers**
  - Memory manager can distinguish pointers from non-pointers (cannot do that in C)
  - All pointers point to the start of a block (not true in C)
  - Cannot hide pointers
    (e.g., by coercing them to an `int`, and then back again)

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)
- **Generational Collectors (Lieberman and Hewitt, 1983)**
  - Collection based on lifetimes
    - Most allocations become garbage very soon
    - So focus reclamation work on zones of memory recently allocated
- **For more information:**
  Jones and Lin, "*Garbage Collection: Algorithms for Automatic Dynamic Memory*", John Wiley & Sons, 1996.

Memory Related Bugs

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

C Pointer Declarations: Test Yourself!

`int *p`	p is a pointer to int
`int *p[13]`	p is an array[13] of pointer to int
`int *(p[13])`	p is an array[13] of pointer to int
`int **p`	p is a pointer to a pointer to an int
`int (*p)[13]`	p is a pointer to an array[13] of int
`int *f()`	f is a function returning a pointer to int
`int (*f)()`	f is a pointer to a function returning int
`int (*(*f())[13])()`	f is a function returning ptr to an array[13] of pointers to functions returning int
`int (*(*x[3])())[5]`	x is an array[3] of pointers to functions returning pointers to array[5] of ints

Source: K&R Sec 5.12

Dereferencing Bad Pointers

The classic `scanf` bug

```
int val;

...

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));
}
```


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

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


Referencing Nonexistent Variables

Forgetting that local variables disappear when a function returns

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

   return &val;
}
```


Freeing Blocks Multiple Times

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

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

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]++;
```


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`** (see the manual page for `mallopt`)