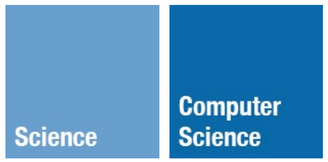
Garbage Collection



CS 351: Systems Programming
Michael Lee <lee@iit.edu>

ILLINOIS INSTITUTE OF TECHNOLOGY
College of Science

Q: can we *automate* the free-ing of memory on the heap?

Q: *why* would we want to?

A: - one less menial, distracting task for programmer

- help prevent common memory bugs:
  - memory leaks
  - dangling pointers
  - double frees
- help combat fragmentation

Q: *how*?

System must:

1. determine when a block of memory is no longer in use
2. free it! (we already know how to do this)

Two general approaches:

1. *reference counting*
2. *tracing*

1. *reference counting* = track # of references (in user code) to every block of memory

- when refcount → 0, free it

Reference counts can be maintained by user or system

- *manual* reference counting: *user*-maintained
- *automatic* reference counting: *system*-maintained

case study: *ObjC*

refcount adjusting methods:

- `new`: create obj with refcount = 1
- `retain`: increment refcount
- `release`: decrement refcount

case study: *ObjC*

other rules:

- ok to call methods on `nil` (no effect)
- when refcount drops to 0, object's `dealloc` is called before it is automatically freed

e.g., manual reference counting (ObjC)

```
@implementation Foo {  // definition of class Foo
    Widget *_myWidget; // instance var declaration (inited to nil)
}
- (void)setWidget:(Widget *)w { // setter method
    [_myWidget release];        // no longer need old obj; refcount--
    _myWidget = [w retain];     // take ownership of new obj; refcount++
}
- (void)dealloc {               // called when Foo's refcount = 0
    [_myWidget release];        // release hold of obj in ivar; refcount--
}
@end
```

```
Widget *w = [Widget new]; // allocate Widget obj; refcount = 1
Foo *f = [Foo new];       // allocate Foo obj; refcount = 1
[f setWidget:w];          // f now (also) owns w; refcount on w = 2
[w release];              // don't need w anymore; refcount on w = 1
...

[f release];              // done with f; refcount on f = 0 (dealloc'd)
                          // refcount on w also -> 0 (dealloc'd)
```

why is this better than malloc/free? (it seems like more work!)

much improved *granularity* of memory management

- a single free will not universally release an object (what if someone still needs it?)
- each object is responsible for its own data (instead of thinking for everyone)

surprisingly, it can also help simplify management of nested/ linked structures:

```
@implementation LinkedList {
    LLNode *_head;
}
- (void)add:(id)obj {
    LLNode *n = [LLNode new];
    [n setObj:obj];
    [n setNext:_head];
    [_head release];
    _head = n;
}

- (void)clear {
    [_head release];
    _head = nil;
}

- (void)dealloc {
    [_head release];
}
@end
```

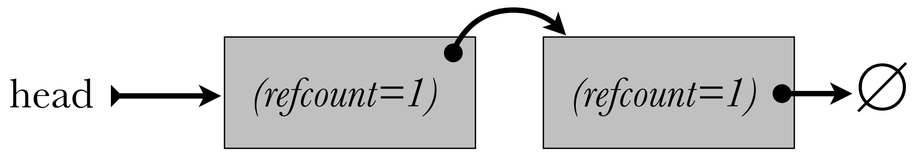
```
@implementation LLNode {
    id _obj;
    LLNode *_next;
}
- (void)setObj:(id)obj {
    [_obj release];
    _obj = [obj retain];
}

- (void)setNext:(LLNode *)next {
    [_next release];
    _next = [next retain];
}

- (void)dealloc {
    [_obj release];
    [_next release];
}
@end
```

```objc
@implementation LinkedList {
    LLNode *_head;
}
- (void)add:(id)obj {
    LLNode *n = [LLNode new];
    [n setObj:obj];
    [n setNext:_head];
    [_head release];
    _head = n;
}

- (void)clear {
    [_head release];
    _head = nil;
}

- (void)dealloc {
    [_head release];
}
@end
```


```objc
@implementation LLNode {
    id _obj;
    LLNode *_next;
}
- (void)setObj:(id)obj {
    [_obj release];
    _obj = [obj retain];
}

- (void)setNext:(LLNode *)next {
    [_next release];
    _next = [next retain];
}

- (void)dealloc {
    [_obj release];
    [_next release];
}
@end
```

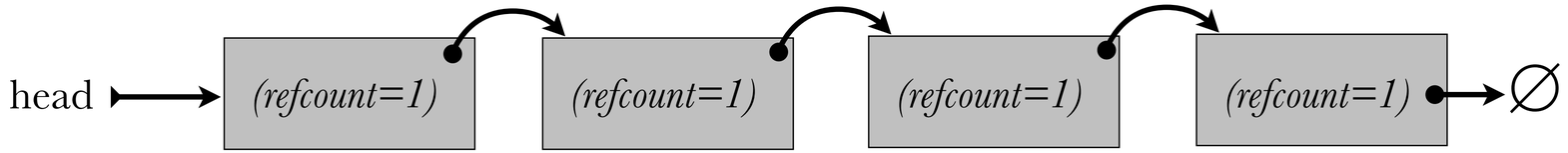
```objc
@implementation LinkedList {
    LLNode *_head;
}
- (void)add:(id)obj {
    LLNode *n = [LLNode new];
    [n setObj:obj];
    [n setNext:_head];
    [_head release];
    _head = n;
}

- (void)clear {
    [_head release];
    _head = nil;
}

- (void)dealloc {
    [_head release];
}
@end
```

```objc
@implementation LLNode {
    id _obj;
    LLNode *_next;
}
- (void)setObj:(id)obj {
    [_obj release];
    _obj = [obj retain];
}

- (void)setNext:(LLNode *)next {
    [_next release];
    _next = [next retain];
}

- (void)dealloc {
    [_obj release];
    [_next release];
}
@end
```



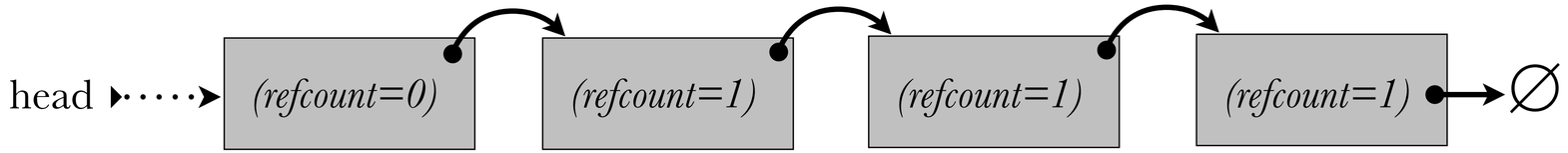
```
@implementation LinkedList {
    LLNode *_head;
}
- (void)add:(id)obj {
    LLNode *n = [LLNode new];
    [n setObj:obj];
    [n setNext:_head];
    [_head release];
    _head = n;
}

- (void)clear {
    [_head release];
    _head = nil;
}

- (void)dealloc {
    [_head release];
}
@end
```

```
@implementation LLNode {
    id _obj;
    LLNode *_next;
}
- (void)setObj:(id)obj {
    [_obj release];
    _obj = [obj retain];
}

- (void)setNext:(LLNode *)next {
    [_next release];
    _next = [next retain];
}

- (void)dealloc {
    [_obj release];
    [_next release];
}
@end
```



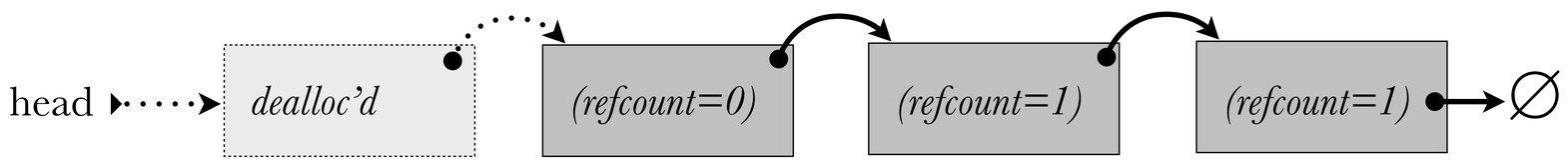
```objc
@implementation LinkedList {
    LLNode *_head;
}
- (void)add:(id)obj {
    LLNode *n = [LLNode new];
    [n setObj:obj];
    [n setNext:_head];
    [_head release];
    _head = n;
}

- (void)clear {
    [_head release];
    _head = nil;
}

- (void)dealloc {
    [_head release];
}
@end
```

```objc
@implementation LLNode {
    id _obj;
    LLNode *_next;
}
- (void)setObj:(id)obj {
    [_obj release];
    _obj = [obj retain];
}

- (void)setNext:(LLNode *)next {
    [_next release];
    _next = [next retain];
}

- (void)dealloc {
    [_obj release];
    [_next release];
}
@end
```



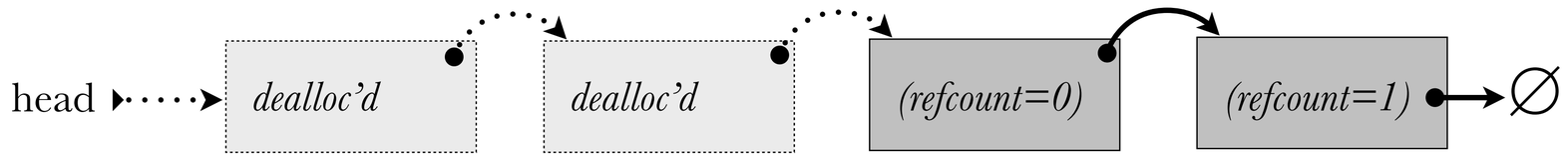
```
@implementation LinkedList {
    LLNode *_head;
}
- (void)add:(id)obj {
    LLNode *n = [LLNode new];
    [n setObj:obj];
    [n setNext:_head];
    [_head release];
    _head = n;
}

- (void)clear {
    [_head release];
    _head = nil;
}

- (void)dealloc {
    [_head release];
}
@end
```

```
@implementation LLNode {
    id _obj;
    LLNode *_next;
}
- (void)setObj:(id)obj {
    [_obj release];
    _obj = [obj retain];
}

- (void)setNext:(LLNode *)next {
    [_next release];
    _next = [next retain];
}

- (void)dealloc {
    [_obj release];
    [_next release];
}
@end
```



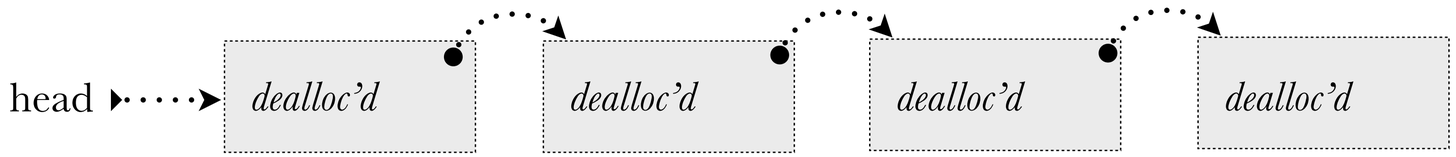
```
@implementation LinkedList {
    LLNode *_head;
}
- (void)add:(id)obj {
    LLNode *n = [LLNode new];
    [n setObj:obj];
    [n setNext:_head];
    [_head release];
    _head = n;
}

- (void)clear {
    [_head release];
    _head = nil;
}

- (void)dealloc {
    [_head release];
}
@end
```

```
@implementation LLNode {
    id _obj;
    LLNode *_next;
}
- (void)setObj:(id)obj {
    [_obj release];
    _obj = [obj retain];
}

- (void)setNext:(LLNode *)next {
    [_next release];
    _next = [next retain];
}

- (void)dealloc {
    [_obj release];
    [_next release];
}
@end
```

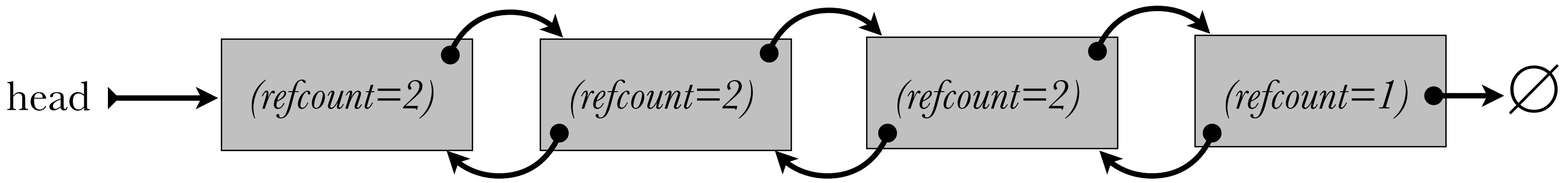


```objc
@implementation LinkedList {
    LLNode *_head;
}
- (void)add:(id)obj {
    LLNode *n = [LLNode new];
    [n setObj:obj];
    [n setNext:_head];
    [_head release];
    _head = n;
}

- (void)clear {
    [_head release];
    _head = nil;
}

- (void)dealloc {
    [_head release];
}
@end
```

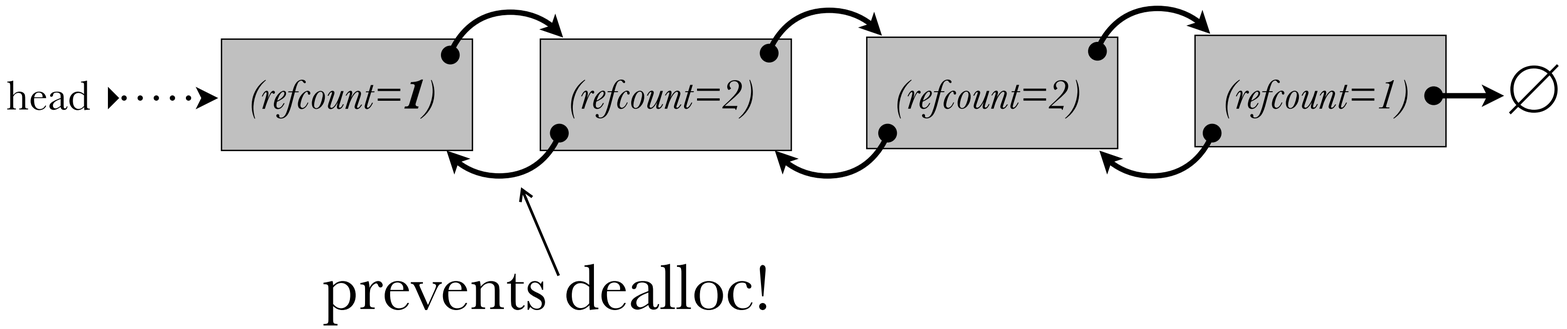
```objc
@implementation LLNode {
    id _obj;
    LLNode *_next;
}
- (void)setObj:(id)obj {
    [_obj release];
    _obj = [obj retain];
}

- (void)setNext:(LLNode *)next {
    [_next release];
    _next = [next retain];
}

- (void)dealloc {
    [_obj release];
    [_next release];
}
@end
```



but, a catch: beware *circular* references!



but, a catch: beware *circular* references!



typically establish conventions for non-retaining references;
aka. *weak* references

e.g., list:prior & tree:child links

rules for release/retain are quite simple:

- “strong” pointers: release old & retain new on assignment
  - also release if pointer goes out of scope
- “weak” pointers: never retain assigned objects

automatic reference counting requires that we designate strong & weak pointers

- retains & releases happen automatically

```
@implementation LLNode {
    __strong id _obj;
    __strong LLNode *_next;
}
@end

@implementation LinkedList {
    __strong LLNode *_head;
}
- (void)add:(id)obj {
    LLNode *node = [LLNode new];
    node.obj  = obj;    // obj is retained by node
    node.next = _head;  // _head is retained by node
    _head     = node;   // node is retained; old _head value is released
}

- (void)clear {
    _head = nil; // releases (previous) _head, causing chain of deallocs
}
@end
```

```objc
@implementation LLNode {
    __strong id _obj;
    __strong LLNode *_next;
    __weak LLNode *_prior; // note weak pointer; will not auto-retain
}
@end

@implementation LinkedList {
    __strong LLNode *_head;
}
- (void)add:(id)obj { // creates a circular doubly-linked list
    LLNode *node = [LLNode new];
    node.obj  = obj;
    if (_head == nil) {
        node.next = node.prior = node; // only next retains (not prior)
        _head = node;
    } else {
        node.next = _head;
        node.prior = _head.prior;
        _head.prior.next = _head.prior = node;
        _head = node;
    }
}

- (void)clear {
    _head.prior.next = nil; // take care that head is not in retain cycle
    _head = nil;            // so that this deallocs list (prior refs ok)
}
@end
```

NB: reference counting provides *predictable* deallocation

i.e., when refcount → 0, object is freed immediately

(contrast this with the next approach)

2. *tracing* = periodically determine what memory has become *unreachable*, and deallocate it

general approach:

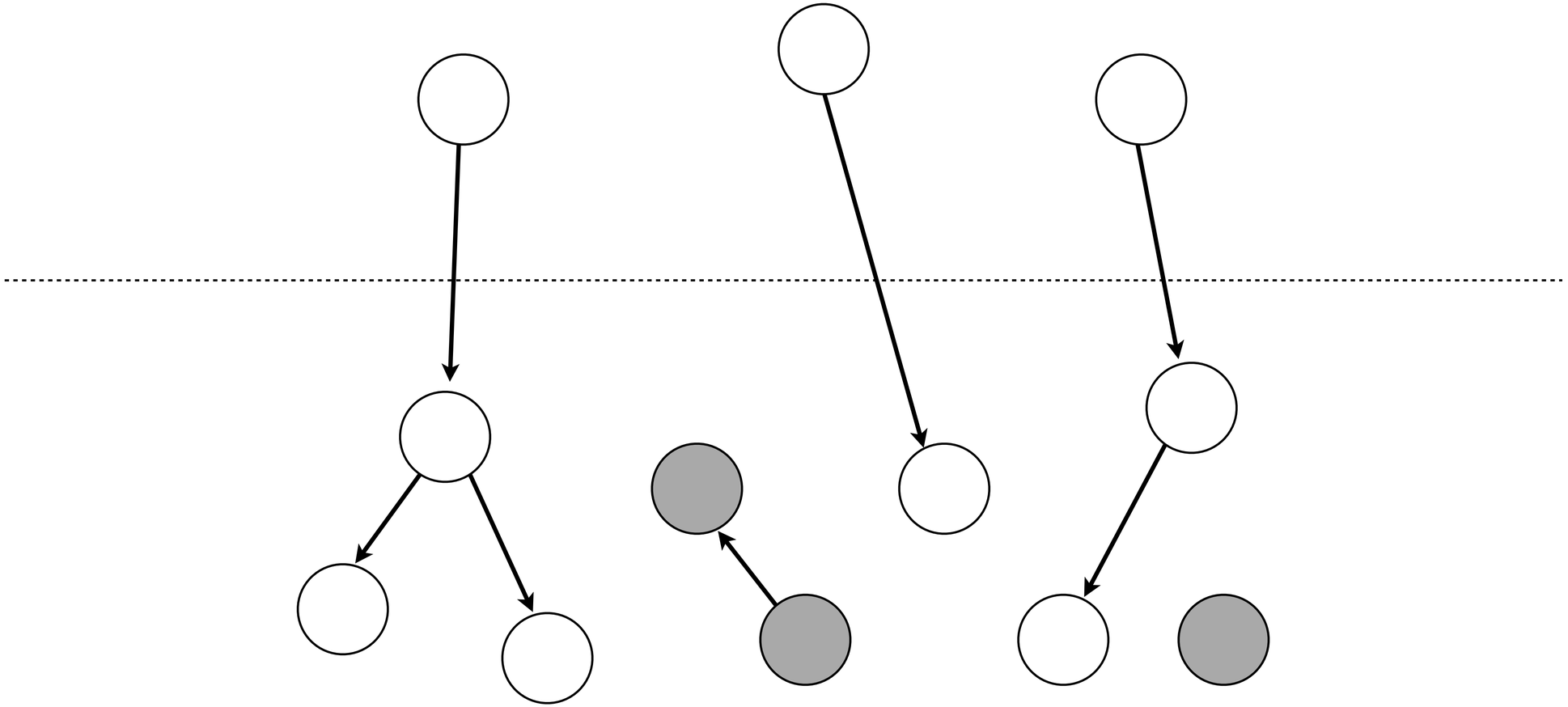
- treat memory as a *graph*
- identify *root nodes / pointers*
- (recursively) find all *reachable* memory

root nodes?

- pointers in static memory
- pointers in active stack frames

Static & Local

"Root" space



Heap space

simple algorithm: “mark & sweep”

1. *mark*: recursively label all reachable blocks
2. *sweep*: free all allocated, unmarked (unreachable) blocks

```c
#define IS_ALLOCATED(p) (*(size_t *)(p) & 1)   // bit 0 = allocated?
#define IS_MARKED(p)    (*(size_t *)(p) & 2)   // bit 1 = marked?
#define BLK_SIZE(p)     (*(size_t *)(p) & ~3L)

void mark(void *p) {
    char *bp = find_block_header(p); // how do we do this efficiently?
    if (bp == NULL || !IS_ALLOCATED(bp) || IS_MARKED(bp)) {
        return;
    }
    *(size_t *)bp |= 2; // mark block

    // next, recursively mark all blocks reachable from this one

}

void sweep () {
    char *bp = heap_start();
    while (bp < heap_end()) {
        if (IS_MARKED(bp))
            *bp &= ~2L; // clear out mark bit
        else if (IS_ALLOCATED(bp))
            free(bp + SIZE_T_SIZE); // free unmarked, allocated block
        bp += BLK_SIZE(bp);
    }
}
```

```c
#define IS_ALLOCATED(p) (*(size_t *)(p) & 1)   // bit 0 = allocated?
#define IS_MARKED(p)    (*(size_t *)(p) & 2)   // bit 1 = marked?
#define BLK_SIZE(p)     (*(size_t *)(p) & ~3L)

void mark(void *p) {
    char *bp = find_block_header(p); // how do we do this efficiently?
    if (bp == NULL || !IS_ALLOCATED(bp) || IS_MARKED(bp)) {
        return;
    }
    *(size_t *)bp |= 2; // mark block
    for (int i=SIZE_T_SIZE; i<=BLK_SIZE(bp)-sizeof(void *); i++) {
        mark(*(void **)(bp + i)); // assume all words in payload can be pointers!
    }
}

void sweep () {
    char *bp = heap_start();
    while (bp < heap_end()) {
        if (IS_MARKED(bp))
            *bp &= ~2L; // clear out mark bit
        else if (IS_ALLOCATED(bp))
            free(bp + SIZE_T_SIZE); // free unmarked, allocated block
        bp += BLK_SIZE(bp);
    }
}
```

```
for (int i=SIZE_T_SIZE; i<=BLK_SIZE(bp)-sizeof(void *); i++) {
    mark(*(void **)(bp + i)); // assume all words in payload can be pointers!
}
```

need to do this because the user can store a pointer just about *anywhere* using C

- but may encounter a lot of *false positives*
  - i.e., not a real pointer, but results in an allocated block being marked

this is a *conservative* GC implementation

in a system with *run time type information*, can reliably detect if a value is a pointer

enables *precise* garbage collection

another issue: *when* to run GC?

would like GC to periodically free up memory “in the background”, without interrupting user code

but if user mallocs while GC frees, we may have concurrent access to heap data structures …

```
void sweep () {
    char *bp = heap_start();
    while (bp < heap_end()) {
        if (IS_MARKED(bp))
            *bp &= ~2L; // clear out mark bit
        else if (IS_ALLOCATED(bp))
            free(bp + SIZE_T_SIZE); // free unmarked, allocated block
        bp += BLK_SIZE(bp);
    }
}
```

… and resulting race conditions! (e.g., consider a block being used for malloc that is currently being coalesced)

mark & sweep GC requires *freezing* the heap while running

- aka “stop-the-world” GC
- horrible performance implications!

other algorithms permit *on-the-fly* marking

e.g., *tri-color* marking

partition allocated blocks into 3 sets:

- white: unscanned
- black: reachable nodes that don't refer to white nodes
- grey: reachable nodes (but may refer to black/white nodes)

at outset:

- directly reachable blocks are grey
- all other nodes are white

periodically:

- scan all children of a grey object (making them black)
- move white blocks found to grey set

i.e., always white → grey → black

when grey set is empty; i.e., when we've scanned all reachable blocks, free any remaining white blocks

many *heuristics* exist to optimize GC

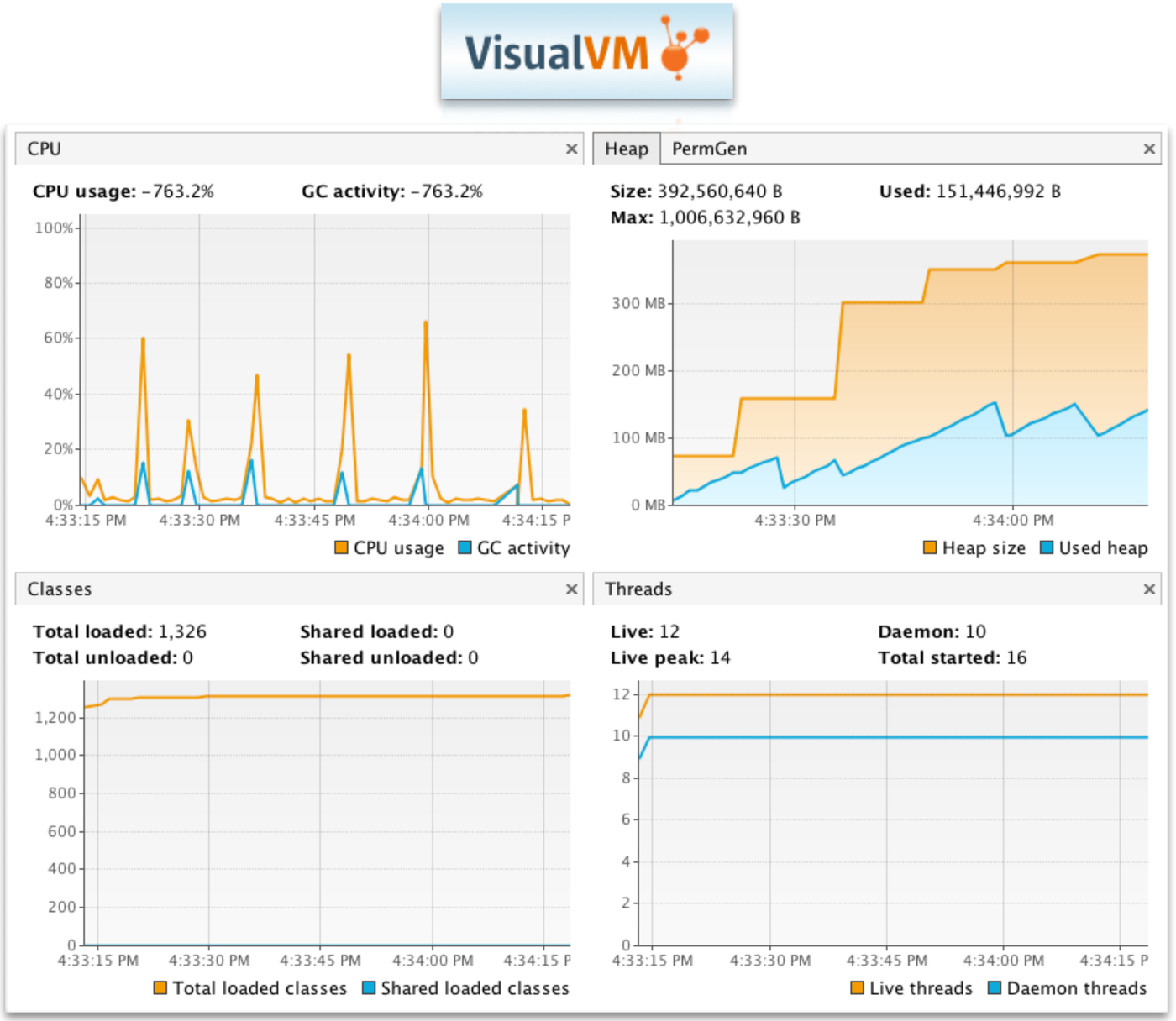
- *generational* GCs classify blocks by age and prioritize searching recent ones (good for handling "peaks")
- *copying* GCs allow for memory compaction to reduce fragmentation (requires opaque pointers)

but question remains: *when* & *how frequently* to perform G.C.?

- invariably incurs overhead
- worse, results in *unpredictable* performance!

Example: Java GC behavior

```java
// Test bench for GC
public class TestGCThread extends Thread {
    public void run(){
        while(true){
            try {
                int delay = (int)Math.round(100 * Math.random());
                Thread.sleep(delay); // random sleep
            } catch(InterruptedException e){ }

            // create random number of objects
            int count = (int)Math.round(10000 * Math.random());
            for (int i=0; i < count; i++){
                new TestObject(); // immediately unreachable!
            }
        }
    }

    public static void main(String[] args){
        new TestGCThread().start();
    }
}

class TestObject {
    static long alloced = 0;
    static long freed   = 0;
    public TestObject () { alloced++; }
    public void finalize () { freed++; }
}
```


VisualVM
CPU
CPU usage: -763.2%
GC activity: -763.2%
100%
80%
60%
40%
20%
0%
4:33:15 PM
4:33:30 PM
4:33:45 PM
4:34:00 PM
4:34:15 P
CPU usage
GC activity
Heap
PermGen
Size: 392,560,640 B
Used: 151,446,992 B
Max: 1,006,632,960 B
300 MB
200 MB
100 MB
0 MB
4:33:30 PM
4:34:00 PM
Heap size
Used heap
Classes
Total loaded: 1,326
Shared loaded: 0
Total unloaded: 0
Shared unloaded: 0
1,200
1,000
800
600
400
200
0
4:33:15 PM
4:33:30 PM
4:33:45 PM
4:34:00 PM
4:34:15 P
Total loaded classes
Shared loaded classes
Threads
Live: 12
Daemon: 10
Live peak: 14
Total started: 16
12
10
8
6
4
2
0
4:33:15 PM
4:33:30 PM
4:33:45 PM
4:34:00 PM
4:34:15 P
Live threads
Daemon threads