15-213

"The course that gives CMU its Zip!"

Memory Management II: Dynamic Storage Allocation Mar 7, 2000

Topics

- **Segregated free lists**
  - Buddy system
- **Garbage collection**
  - Mark and Sweep
  - Copying
  - Reference counting

Basic allocator mechanisms

Sequential fits (implicit or explicit single free list)

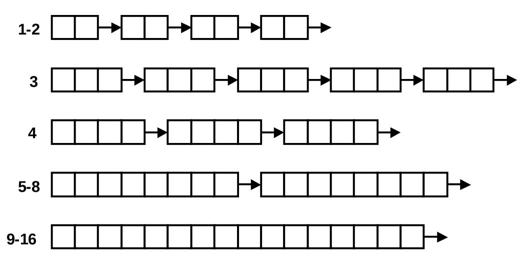
- **best fit, first fit, or next fit placement**
- **various splitting and coalescing options**
  - splitting thresholds
  - immediate or deferred coalescing

Segregated free lists

- **simple segregated storage -- separate heap for each size class**
- **segregated fits -- separate linked list for each size class**
  - buddy systems

Segregate Storage

Each size "class" has its own collection of blocks



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

Simple segregated storage

Separate heap and free list for each size class

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**
- **If page is empty, return the page for use by another size (optional)**

Tradeoffs:

- **fast, but can fragment badly**

Segregated fits

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)

Tradeoffs

- faster search than sequential fits (i.e., log time for power of two size classes)
- controls fragmentation of simple segregated storage
- coalescing can increase search times
  - deferred coalescing can help

Buddy systems

Special case of segregated fits.

- all blocks are power of two sizes

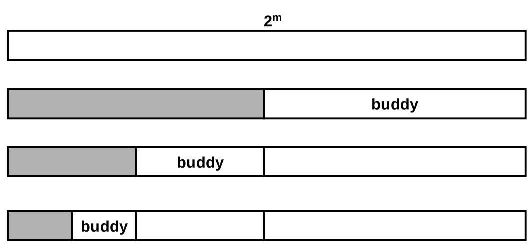
Basic idea:

- Heap is 2^m words
- Maintain separate free lists of each size 2^k, 0 <= k <= m.
- Requested block sizes are rounded up to nearest power of 2.
- Originally, one free block of size 2^m.

Buddy systems (cont)

To allocate a block of size 2^k:

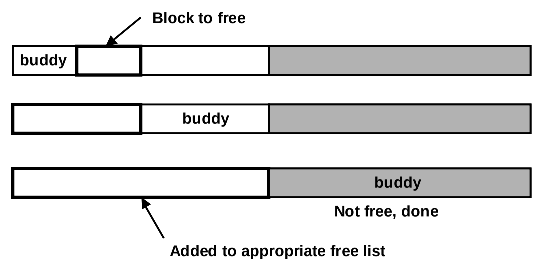
- Find first available block of size 2^j s.t. k <= j <= m.
- if j == k then done.
- otherwise recursively split block until j == k.
- Each remaining half is called a “buddy” and is placed on the appropriate free list



Buddy systems (cont)

To free a block of size 2^k

- continue coalescing with buddies while the buddies are free



Buddy systems (cont)

Key fact about buddy systems:

- **given the address and size of a block, it is easy to compute the address of its buddy**
- **e.g., block of size 32 with address `xxx...x00000` has buddy `xxx...x10000`**

Tradeoffs:

- **fast search and coalesce**
- **subject to internal fragmentation**

Internal fragmentation

Internal fragmentation is wasted space inside allocated blocks:

- **minimum block size larger than requested amount**
  - e.g., due to minimum free block size, free list overhead
- **policy decision not to split blocks**
  - e.g., buddy system
  - Much easier to define and measure than external fragmentation.

Implicit Memory Management Garbage collector

***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**

Need to make certain assumptions about pointers

- **Memory manager can distinguish pointers from non-pointers**
- **All pointers point to the start of a block**
- **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")

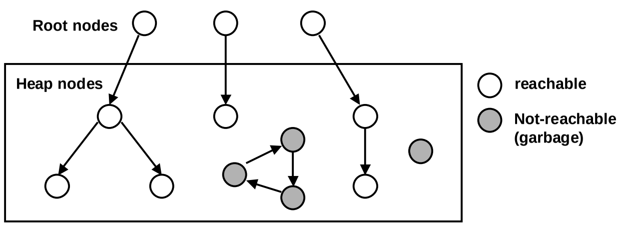
Reference counting (Collins, 1960)
- Does not move blocks

Copying collection (Minsky, 1963)
- Moves blocks

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 <u>cleared</u>
- `read(b,i)`: read location `i` of block `b into register`
- `write(b,i,v)`: write `v` into location `i` of block `b`

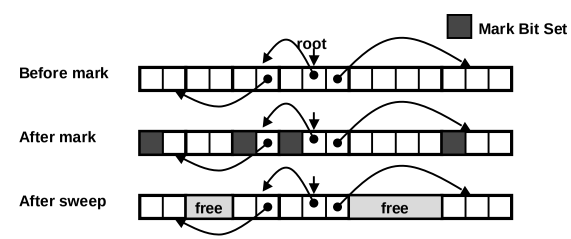
Each block will have a header word
- addressed as `b[-1]`, for a block `b`
- Used for different purposes in different collectors

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 blccks and **free** blocks that are **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;
}
```

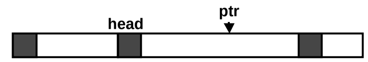
Sweep using lengths to find next block

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


Mark and sweep in C

A C Conservative Collector

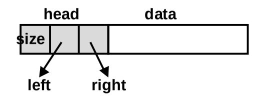
- **`Is_ptr()` can determines if a word is a pointer by checking if it points to an allocated block of memory.**
- **But, in C pointers can point to the middle of a block.**



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 head (use two additional words)

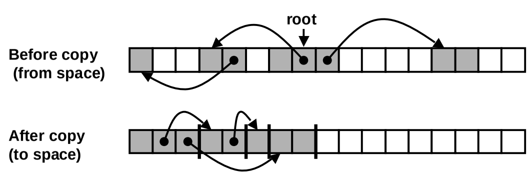


Copying collection

Keep two equal-sized spaces, from-space and to-space

Repeat until application finishes

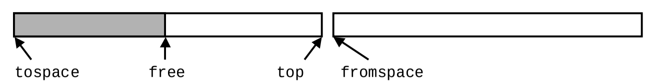
- **Application allocates in one space contiguously until space is full.**
- **Stop application and copy all reachable blocks to contiguous locations in the other space.**
- **Flip the roles of the two spaces and restart application.**



Copy does not necessarily keep the order of the blocks

Has the effect or removing all fragments

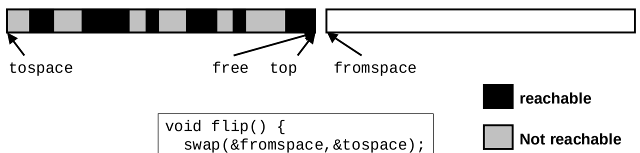
Copying collection (new)




```
ptr new (int n) {
   if (free+n+1 > top) flip();
   newblock = free;
   free += (n+1);
   for (i=0; i < n+1; i++)
      newblock[i] = 0;
   return newblock+1;
}
```

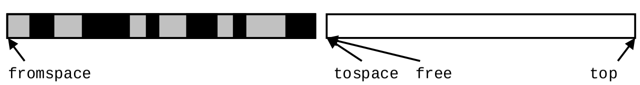
- **All new blocks are allocated in `tospace`, one after the other**
- **An extra word is allocated for the header**
- **The Garbage-Collector starts (flips), when we reach `top`**

Copying collection (flip)

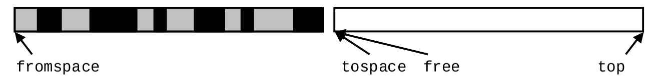



```
void flip() {
  swap(&fromspace,&tospace);
  top = tospace + size;
  free = tospace;
  for (r in roots)
    r = copy(r);
}
```

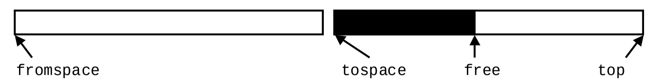
After the first three lines of flip (before the copy).



Copying collection (copy)




```
ptr copy(ptr p) {
   if (!is_ptr(p)) return p;      // do nothing if not pointer
   if (p[-1] != 0) return p[-1];  // check if already forwaded
   new = free+1;                  // location for the copy
   p[-1] = new;                   // set the forwarding pointer
   new[-1] = 0;                   // clear forward in new copy
   free += length(p)+1;           // increment free pointer
   for (i=0; i < length(p); i++)  // copy all children and
     new[i] = copy(p[i]);         // update pointers to new loc
   return new; }
```



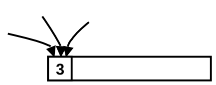
Reference counting

Basic algorithm

- Keeps count on each block of how many pointers point to the block
- When a count goes to zero, the block can be freed

Data structures

- Can be built on top of an existing explicit allocator
  - allocate(n), free(p)
- Add an additional header word for the "reference count"



- Keeping the count updated requires that the we modify every read and write (can be optimized out in some cases)

Reference counting

Set reference count to one when creating a new block

```
ptr new (int n) {
   newblock = allocate(n+1);
   newblock[0] = 1;
   return newblock+1;
}
```

When reading a value increment its reference counter

```
val read(ptr b, int i) {
  v = b[I];
  if (is_ptr(v)) v[-1]++;
  return v;
}
```

When writing decrement the old value and increment the new value

```
void write(ptr b, int i, val v) {
   decrement(b[I]);
   if (is_ptr(v)) v[-1]++;
   b[i] = v;
}
```

Reference counting

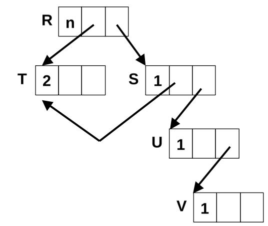
Decrement

- if counter decrements to zero then the block can be freed
- when freeing a block, the algorithm must decrement the counters of everything pointed to by the block -- this might in turn recursively free more blocks

```
void decrement(ptr p) {
   if (!is_ptr(p)) return;
   p[-1]--;
   if (p[-1] == 0) {
      for (i=0; i<length(p); i++)
         decrement(p[i]);
      free(p-1);
   }
}
```


Reference counting example

Initially

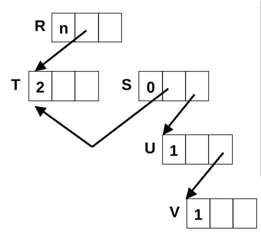


Now consider: `write(R,1,NULL)`

- This will execute a: `decrement(S)`

Reference counting example

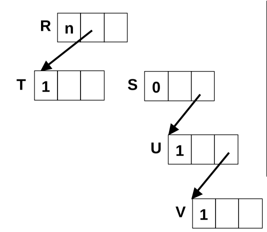
After counter on S is decremented




```
void decrement(ptr p) {
   if (!is_ptr(p)) return;
   p[-1]--;
   if (p[-1] == 0) {
      for (i=0; i<length(p); i++)
         decrement(p[i]);
      free(p-1);
   }
}
```


Reference counting example

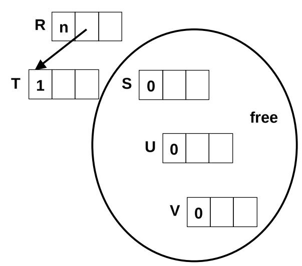
After: `decrement(S[0])`




```
void decrement(ptr p) {
   if (!is_ptr(p)) return;
   p[-1]--;
   if (p[-1] == 0) {
      for (i=0; i<length(p); i++)
         decrement(p[i]);
      free(p-1);
   }
}
```

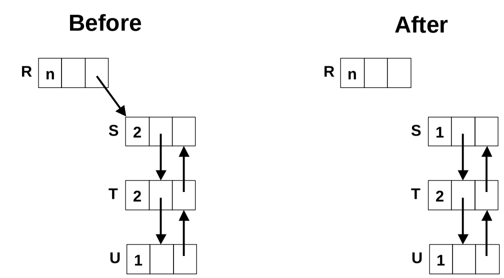
Reference counting example

After: `decrement(S[1])`



Reference counting cyclic data structures

`write(R,1,NULL)`



Garbage Collection Summary

Copying Collection

- **Pros:** prevents fragmentation, and allocation is **very** cheap
- **Cons:** requires twice the space (**from** and **to**), and stops allocation to collect

Mark and Sweep

- **Pros:** requires little extra memory (assuming low fragmentation) and does not move data
- **Cons:** allocation is somewhat slower, and all memory needs to be scanned when sweeping

Reference Counting

- **Pros:** requires little extra memory (assuming low fragmentation) and does not move data
- **Cons:** reads and writes are more expensive and difficult to deal with cyclic data structures

Some collectors use a combination (e.g. copying for small objects and reference counting for large objects)