University of Arizona

CSc 340

Foundations of Computer Systems

Christian Collberg
April 12, 2001

Heap Management

Dynamic Memory Allocation

- Suppose we want to write a program that keeps track of a company's employees. The information for each employee will be kept in a structure:
  1. a structure is created when an employee is hired, and
  2. deleted when the employee is fired or quits.

  How should the structures be allocated and freed?

Slide 13–1

Static allocation

- Use the `.word` directive. Memory is allocated before the program runs.
  - Requires allocating the maximum number of employee structures.
  - Not all may be used, wasting space.
  - Increasing the maximum requires reassembling the program.
- This is not what we want.

Slide 13–2

Stack allocation

- Use a stack, similar to stack frames for subroutines.
  - A stack supports push and pop.
  - A stack is a First-In-First-Out data structure.
  - Employees must quit or be fired in the reverse order in which they were hired.
- This is not what we want.

Slide 13–3

Dynamic Heap Allocation

- We need a fully dynamic memory allocation strategy. Employees can quit or be fired in any order relative to their hiring.
- The data structure used to provide general memory allocation such as this is called a heap. It's not the same type of heap you learned about (or will learn about) in a data structures course (those heaps were used to implement a priority queue). Our heap is simply a data structure for keeping track of memory.

Slide 13–4

Run-time Meory Organization

- The memory of a running program is divided into three segments:

  text: the program instructions

  stack: stack frames for subroutines

  heap: program data, both static and dynamic.

- The stack grows downward from the highest address.

Slide 13–5

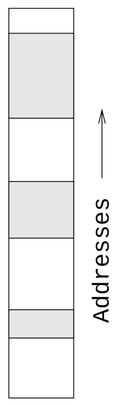
Run-time Meory Organization. . .

- The heap grows upward from the top of the text. Initially the heap contains the static data from the program, and one big free region to hold dynamic data. Once this space is used up, the operating system is invoked to increase the size of the heap by moving the boundary up towards the stack.
- Each program has its own heap and stack.
- When the stack and heap collide your program runs out of memory.

Slide 13–6

Heap Organization

- The heap starts with one big free region, but after the program has been running for a while it will probably be divided into some regions that are being used by the program, and some that are not (employees have been hired and fired). In the following, shaded regions of memory are in-use (they contain data that are being used by the program). Blank regions are not in-use (free).



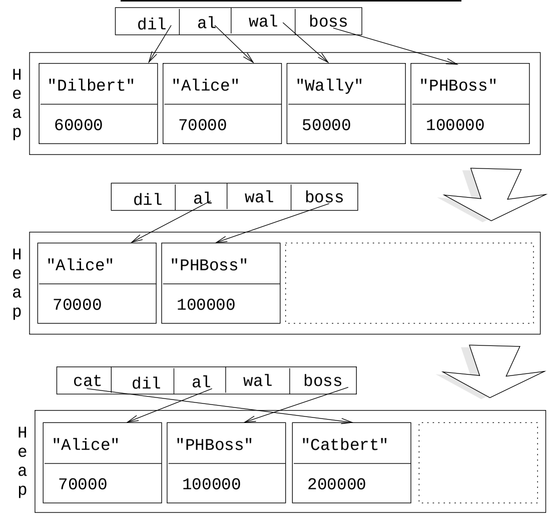
Slide 13–7

Java Example

```
class Employee {String name; int salary;}
class Database {
  public static void main () {
    Employee dil = new Employee("Dilbert",60000);
    Employee al = new Employee("Alice",70000);
    Employee wal = new Employee("Wally",50000);
    Employee boss = new Employee("PHBoss",100000);
    wal = null; dil = null;
    Employee cat = new Employee("Catbert",200000);
  } }
```

Slide 13–8

Java Example...



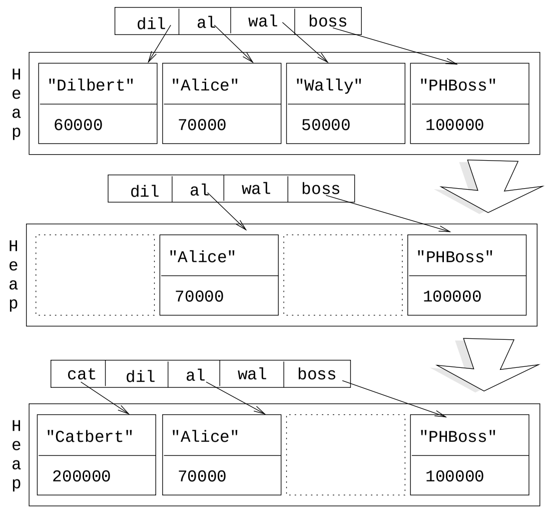
Slide 13–9

Pascal/C/...Example

```
program P;
  record Employee String name; int salary; end
begin
  Employee dil = new Employee("Dilbert",60000);
  Employee al = new Employee("Alice",70000);
  Employee wal = new Employee("Wally",50000);
  Employee boss = new Employee("PHBoss",100000);
  displose(wal); dispose(dil);
  Employee cat = new Employee("Catbert",200000);
end.
```

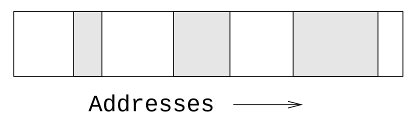
Slide 13–10

Pascal/C/...Example...



Slide 13–11

Heap Organization. . .

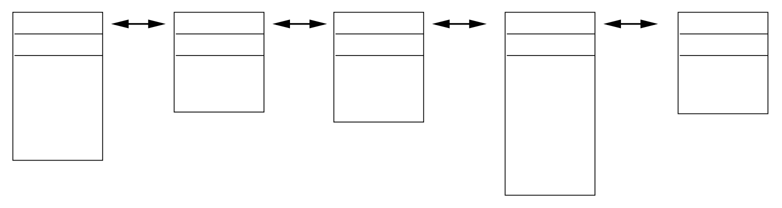


- When the OS increases the size of the heap, more free memory is added to the right of this picture.
- When the program wants memory from the heap, we must allocate some from one of the free regions. For each free region we need to know its starting address and its size.

Slide 13–12

Heap Organization. . .

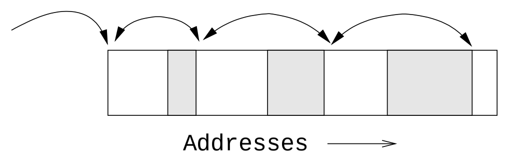
- Where should this information be kept? In an array? This has the same problems as allocating the employee structures themselves in an array. A linked-list called the free list is a better choice. Since the regions we are linking are free space, we can put the linked-list structures in the free regions.



Slide 13–13

Heap Organization. . .

- Think of each free region as a structure. Different regions are different sizes, so the structures have different sizes, but all begin with a standard header. The header contains the links for making the linked-list of free regions, and the size of the region.
- The previous picture of memory looks like the following with the free-list added.



Slide 13–14

Malloc

- Memory is allocated from the heap via

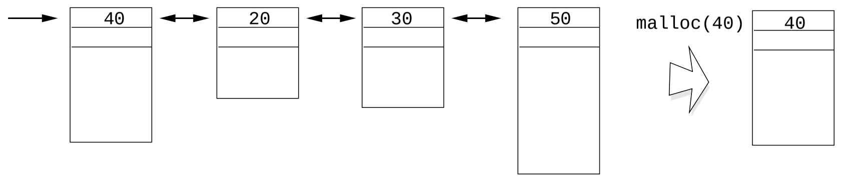
  `malloc(int size)`

  where `size` is the number of bytes needed. `malloc` returns the address of (a pointer to) a region of free memory of at least size bytes.
- `malloc` returns 0 (`NULL`) if there isn't a big enough free region to satisfy the request.

Slide 13–15

Malloc...

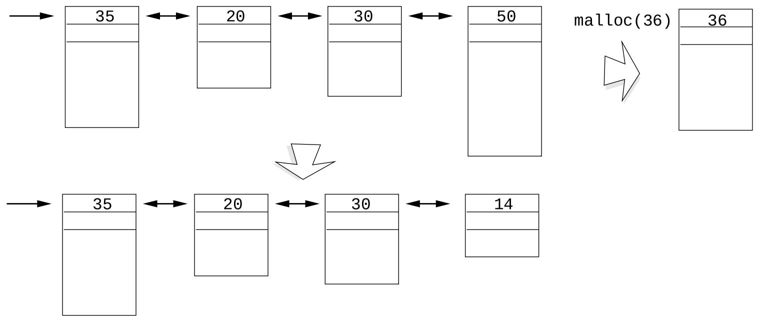
- `malloc` searches the free list for a free region that's big enough, removes it from the free list, and returns its address.



Slide 13–16

Malloc...

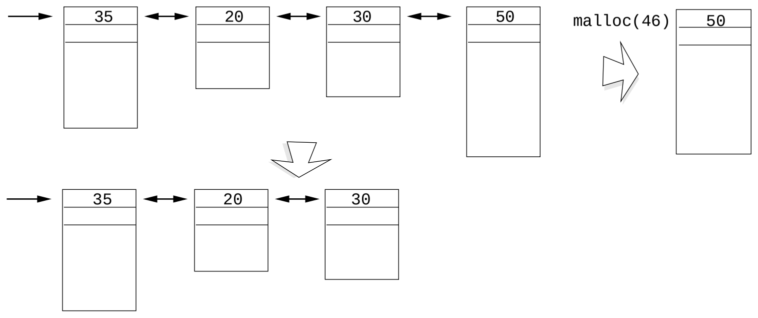
- If the region is bigger than requested, `malloc` takes what is needed from the end of the region and leaves the rest (the beginning) on the free list.



Slide 13–17

Malloc...

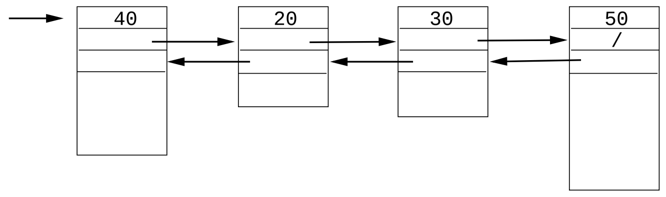
- If what is left is too small to hold a header `malloc` simply returns the entire region. The program that called `malloc` will get more memory then it asked for, but that doesn't matter.



Slide 13–18

Malloc...

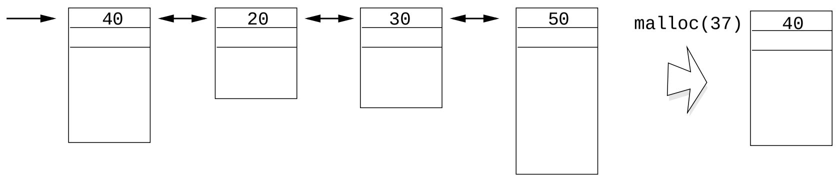
- A doubly-linked-list is often used to make insertion and deletion easier.



Slide 13–19

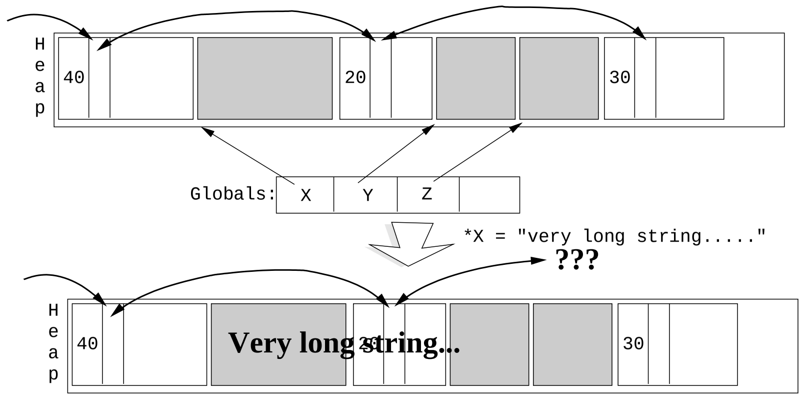
Malloc...

- `malloc` should return a word-aligned region that can store integers. The easiest way to do this is to start with the heap word-aligned, and always allocate memory in multiples of the word size.



Slide 13–20

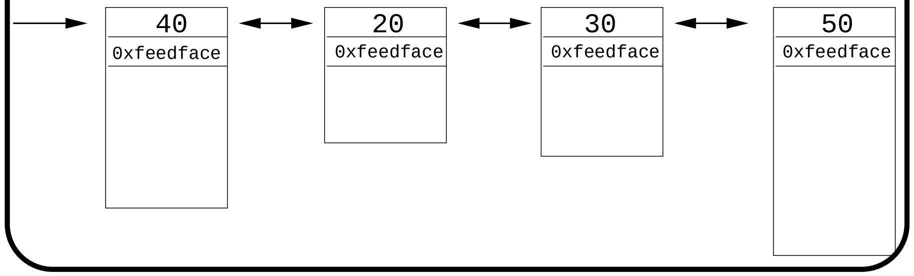
- What happens if the program asks for 50 bytes, but then writes 60 bytes to the region? The last 10 bytes overwrite the first 10 bytes of the next region. This will corrupt the free list if the next region is free (and probably crash the program if it is not).



Slide 13–21

Malloc...

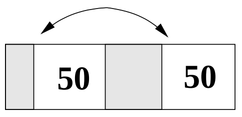
- For this reason it's a good idea to have a magic number in the free list node headers. This is a distinctive value that `malloc` checks when traversing the free list, and complains if the value changes (which indicates the list is corrupted). For example, put a field in the header whose value is always `0xfeedface`.



Slide 13–22

Fragmentation

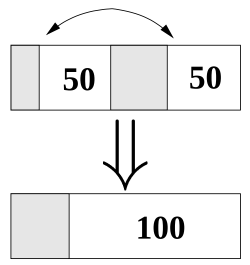
- There may be many regions big enough to satisfy a request. Which one should be used? First, why does it make a difference? Consider a heap that has two free regions, separated by allocated memory. Each region contains 50 bytes. `malloc` can't satisfy a request for 100 bytes, even though there is enough total free memory.



Slide 13–23

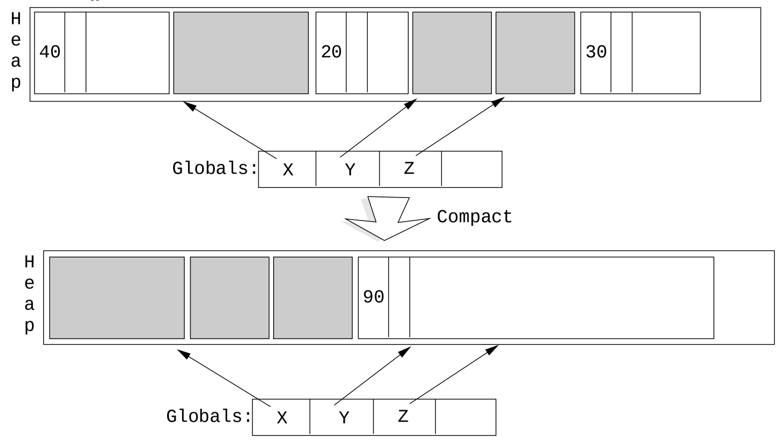
Fragmentation. . .

- Why not move the shaded region out of the way, by copying its contents to the beginning of the heap?



Slide 13–24

- This is called memory compaction, and doesn't work in general because it changes the address of memory previously returned by `malloc`. Most programs cannot handle this.



Slide 13–25

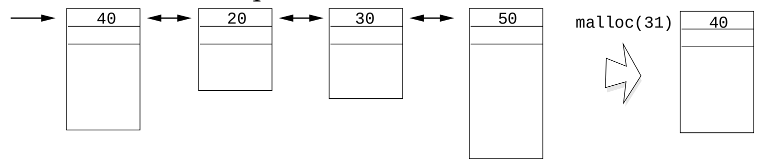
Fragmentation. . .

- Memory fragmentation is what occurs when the free space is in regions too small to be useful. We want our allocation scheme to avoid fragmentation, if possible.
- Note that there is no fragmentation with stack allocation because the free space is always in one big piece.
- Without compaction it is impossible to avoid fragmentation entirely.

Slide 13–26

Memory Allocation Schemes—Best-fit

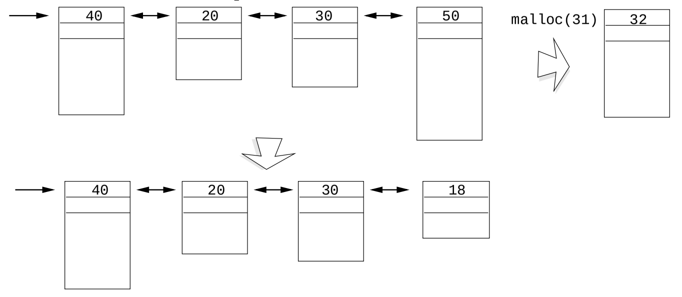
- `malloc` returns the smallest region that's big enough to satisfy the request.
- This favors small regions over large regions, leaving large regions for large requests.
- Unfortunately, it leaves lots of very tiny, useless regions because it always chooses the region whose size is closest to the request.



Slide 13–27

Memory Allocation Schemes—Worst-fit

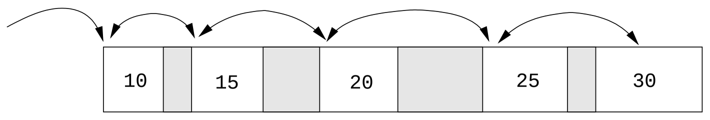
- `malloc` returns the biggest region that satisfies the request. This avoids the "lots of tiny regions" problem. The idea is that the amount "left over" is likely to be big enough to satisfy a subsequent request. This tends to produce free regions of about the same size.



Slide 13–28

Example I

- Example: suppose the free list has regions of size 10, 15, 20, 25, and 30 bytes, and `malloc` is called to allocate a region of size 12.

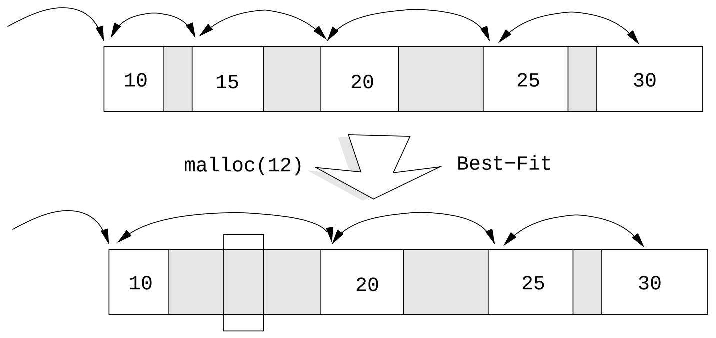


- Both of these algorithms require searching the entire free list, which can be expensive. They also cannot prevent fragmentation.

Slide 13–29

Example I — Best-Fit

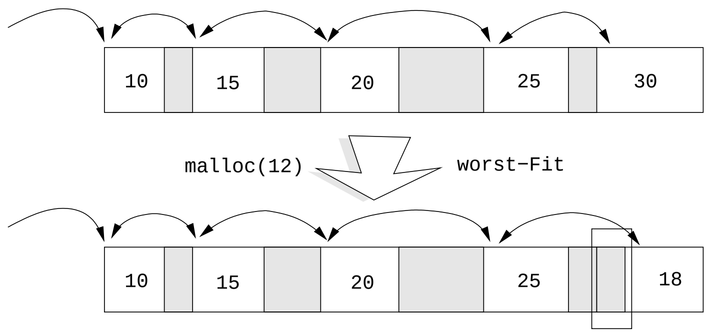
- Best-fit will allocate from the 15-byte region, leaving a region of 3 byte. Probably will return all 15 bytes as the 3-byte region is too small to hold a node structure.



Slide 13–30

Example I — Worst-Fit

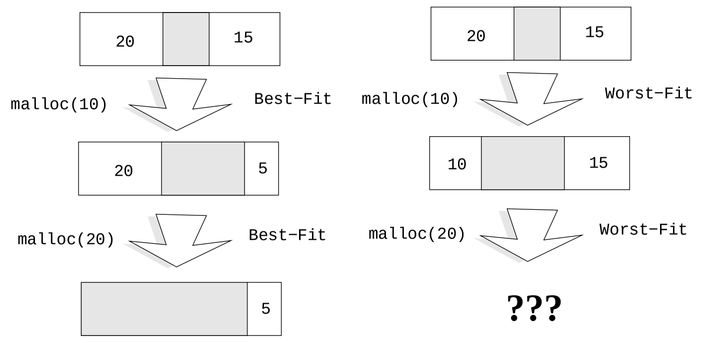
- Worst-fit will allocate from the 30-byte region, leaving a region of 18 bytes.



Slide 13–31

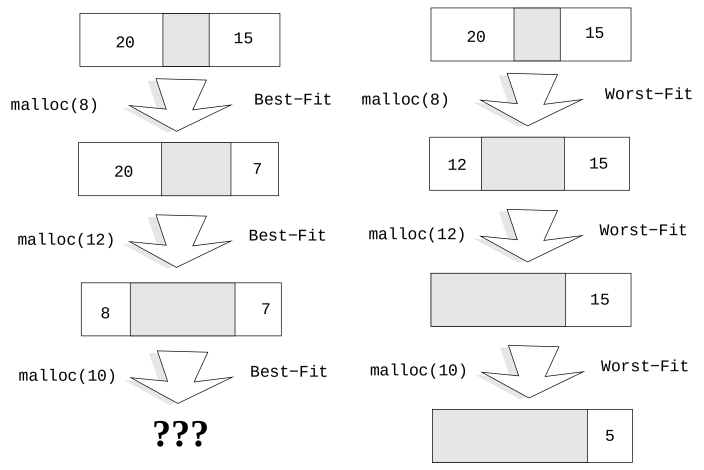
Example II

- Suppose the free list has two regions of size 20 and 15 and the allocations are 10 then 20: which algorithm wins?



Slide 13–32

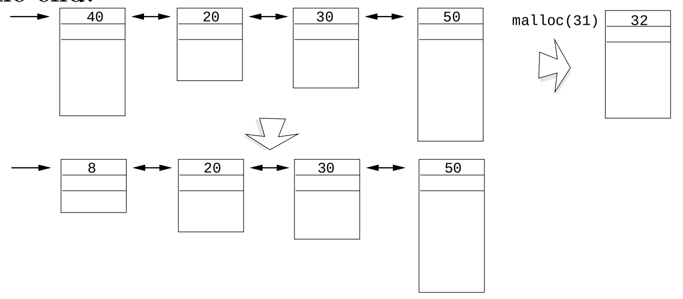
- Who wins if the allocations are 8, 12, then 10?



Slide 13–33

Memory Allocation Schemes—First-fit

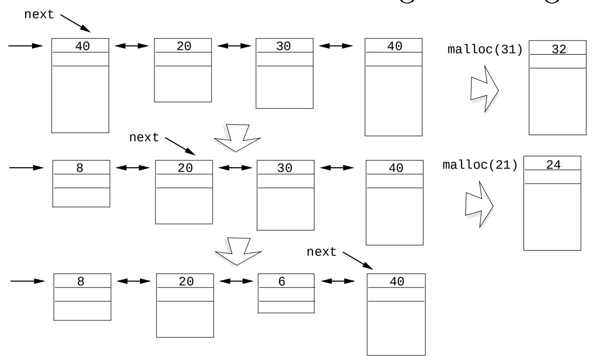
- To avoid searching the entire list First-fit allocates space from the first region on the free list that is big enough. It has the side-effect of leaving lots of small regions at the front of the list, while the larger are at the end.



Slide 13–34

Memory Allocation Schemes—Next-fit

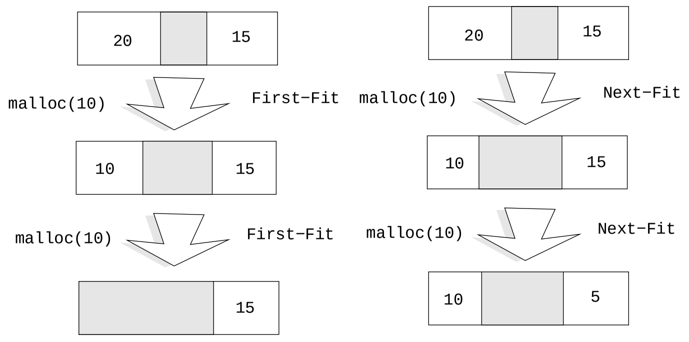
- Next-fit works like First-fit except that it starts searching at the region following the last region allocated. It tends to leave average-sized regions.



Slide 13–35

Example III

- Suppose the free list has two regions of size 20 and 15, and the allocations are 10 then 10. Who wins?



Free

- The routine

  `free(void *address)`

  is used to release memory when it is no longer needed (e.g. an employee quits or is fired).
- The address parameter is a pointer to the region to be freed, and it must have previously been returned by `malloc`.
- In C the type `"void *"` means a pointer to an unspecified type (it can be a pointer to anything, much like a reference to an object of type `"Object"` in Java).

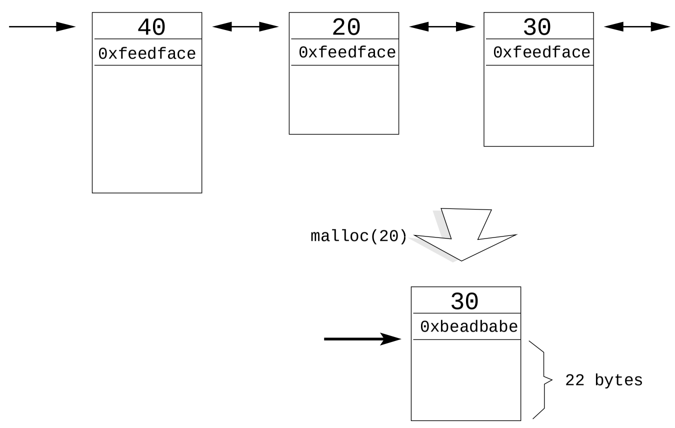
Free. . .

- `free` inserts the region into the free list. It must fill in the header to do so, but this introduces a problem. How does free determine the size of the region? It is only given its address, not its size. One solution is to store the size elsewhere, such as in an array.
- A simple solution is to put a hidden header on allocated regions. This header contains the size of the region, and is stored in memory just before the allocated region.
- I call it "hidden" because it is not seen by the program that called `malloc`; `malloc` returns the address of the first byte after the hidden header.

Free. . .

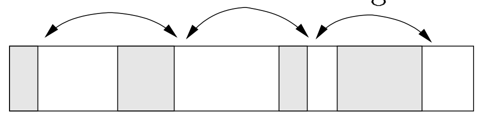
- `Free` subtracts the size of the hidden header from the address it is given to get the address of the hidden header.
  1. `malloc` must account for the hidden header by adding its size to the amount of memory requested.
  2. It's a good idea to have a (different) magic number in the hidden header.
  3. It's probably easiest to use the same header as the linked-list, although not the most space efficient.

Free. . .

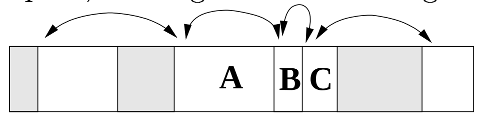


Slide 13–40

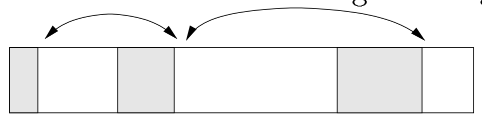
- There is a significant problem with this simple implementation of free. Consider the following memory status:



An employee quits, making an allocated region free:



Note that there are now three consecutive free regions (A,B,C), without any allocated regions between them. They should be coalesced into one large free region:



Slide 13–41

Free. . .

- Free needs to find the two regions on either side (A,C) of the region being freed (B), and if either is also free, merge it with the current region into a single region.
  1. How does free find adjacent regions? One option is to keep the free list sorted by address. When a region is freed insert it into the correct place in the list, then check to see if the free regions before and after it on the list are adjacent.

Slide 13–42

Free. . .

2. How are two regions coalesced? The one with the higher address is merged into the one with the lower address.
3. Increment the size of the lower region by the size of the higher region (including the size of the higher region's header).
4. Remove the higher region from the free list.

Slide 13–43

Putting it all together

- Each free region of memory contains a header with the following fields:
  1. Links for the doubly-linked free list
  2. Size of region (including header)
- Each allocated region of memory contains a hidden header:
  1. Size of region (including hidden header)

Slide 13–44

`Malloc`(size)

1. Add size of hidden header to size.
2. Round size up to a whole number of words.
3. Use Next-fit to search free list for first free region of at least size bytes.
4. Start search using pointer kept in global variable.
5. Check magic numbers as you go.
6. If there isn't a free region big enough, return 0 (NULL).
7. If free region isn't exactly size bytes, compute how much is left- over (excess).

Slide 13–45

8. If excess is at least the size of a header plus one word:
   - Address of allocated region = address of free region + excess

   else
   - Remove free region from free list
   - Address of allocated region = address of free region
9. Put size of allocated region into hidden header at the start of the region.
10. Return address of first byte after hidden header.
11. Update starting search pointer to next region on free list.

Slide 13–46

`Free`(address)

1. Subtract size of hidden header from address to get hidden header's address.
2. Check magic number in hidden header.
3. Use size in hidden header to fill in free-list node header at start of region.
4. Insert free region into sorted free list. The pointer to the start of the free list is stored in a global variable.

Slide 13–47

`Free`(address)

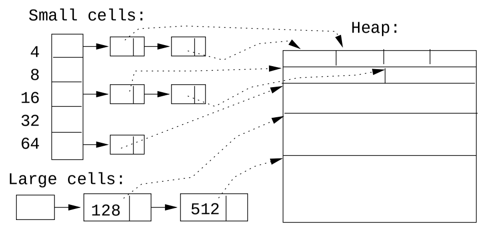
5. If previous region on free list is adjacent to new free region:
   (a) Coalesce by adding size of higher region (including header) to size of lower region.
   (b) Remove higher region from free list.
   (c) Also coalesce subsequent region on free list if it is adjacent.

Slide 13–48

More Algorithms

- There are many different memory allocation algorithms. Many optimize the amount of space used for the header.

  Binning: keeps separate lists for regions of the same size. Good if the program allocates and frees lots of regions of the same size (e.g. employee structures).



Slide 13–49

More Algorithms. . .

Bit map: if regions are of the same size (blocks), keep a bit map instead of a free list. One bit per block: 1 means allocated, 0 means free. One word of the bit map represents 32 blocks. Index of bit in bit map is the index of the region in the ”array” of regions.

Garbage collection: if a program does not have a pointer to a particular region of memory, that memory cannot be accessed by the program and can be automatically added to the free list. The programmer allocates memory, but never calls free. Examples: Lisp, Java

Slide 13–50

Garbage Collection

- Garbage collector must be able to find all regions in heap that are in- use, i.e. pointers to them exist.
- Pass 1: mark. Go through all static and automatic (stack) variables looking for pointers to the heap. Mark each region pointed to, and regions pointed to by pointers in those regions.

Slide 13–51

Garbage Collection. . .

- 1. Conservative collector: assume that any value that looks like a pointer to the heap is a pointer to the heap.
  2. Otherwise, compiler must tag pointers so they can be identified.
- Pass 2: sweep. Go though entire heap, add regions that aren't marked to free list.
- Garbage collection is expensive: 20% or more of all CPU time.
- C and assembly code require a conservative collector.

Slide 13–52

GC: Reference Counts I

- This is the simplest GC technique.
- An extra field is kept in each object containing a count of the number of pointers which point to the object.
- Each time a pointer is made to point to an object, that object's count has to be incremented.
- Similarly, every time a pointer no longer points to an object, that object's count has to be decremented.
- When we run out of dynamic memory we scan through the heap and put objects with a zero reference count back on the free-list.
- Maintaining the reference count is costly. Also, circular structures (circular linked lists, for example) will not be collected.

Slide 13–53

GC: Reference Counts II

- Every object records the number of pointers pointing to it.
- When a pointer changes, the corresponding object's reference count has to be updated.
- GC: reclaim objects with a zero count. Circular structures will not be reclaimed.

Slide 13–54

GC: Reference Counts III

——— `NEW(p)` is implemented as: ———

```
malloc(p); p↑.rc := 0;
```

——— `p↑.next:=q` is implemented as: ———

```
z := p↑.next;
if z ≠ nil then
   z↑.rc--;
   if z↑.rc = 0 then
      reclaim z↑
   endif;
endif;
p↑.next := q;
q↑.rc++;
```

- This code sequence has to be inserted by the compiler for *every* pointer assignment in the program. This is very expensive.

Slide 13–55