

CSE 31

Computer Organization

Lecture 9 – C Memory Management (cont.)

Announcement

▶ Labs

- Lab 3 due this week (with 7 days grace period after due date)
 - Demo is REQUIRED to receive full credit
- Lab 4 out this week
 - Due at 11:59pm on the same day of your next lab
 - You must demo your submission to your TA within 14 days

▶ Reading assignment

- Reading 03 (zyBooks 3.1 - 3.7, 3.9) due 11-OCT
 - Complete Participation Activities in each section to receive grade towards Participation
 - IMPORTANT: Make sure to submit score to CatCourses by using the link provided on CatCourses

Announcement

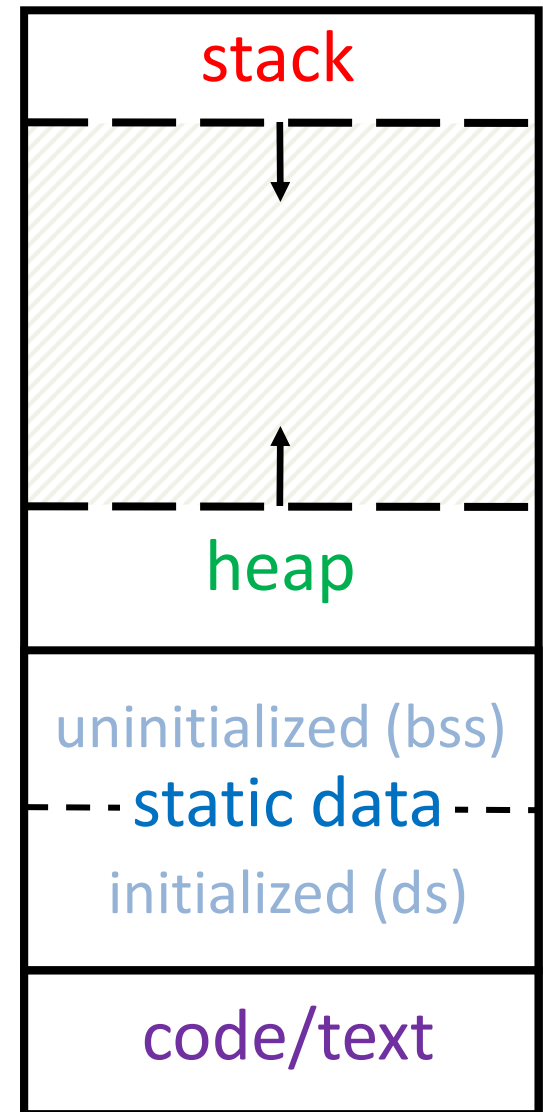
- ▶ Homework assignment
 - Homework 02 (zyBooks 2.1 - 2.9) due 04-OCT
 - Complete *Challenge Activities* in each section to receive grade towards Homework
 - IMPORTANT: Make sure to submit score to CatCourses by using the link provided on CatCourses

Normal C Memory Management (review)

- ▶ A program's **address space** contains 4 regions:
 - **stack**: local variables, grows downward
 - **heap**: space requested for pointers via `malloc()` ; resizes dynamically, grows upward
 - **static data**: Initialized/uninitialized static and global variables
 - **code/text**: loaded when program starts, does not change

For now, OS somehow prevents accesses between stack and heap (gray hash lines). Wait for virtual memory

$\sim FFFF\ FFFF_{hex}$



$\sim 0_{hex}$

K&R Malloc/Free Implementation

(review)

- ▶ Each block of memory is preceded by a header that has two fields:
 - **size** of the block
 - a **pointer to the next** block
- ▶ All **free blocks** are kept in a circular linked list, the pointer field is unused in an allocated block
- ▶ `malloc()` searches the free list for a block that is big enough. If none is found, more memory is requested from the operating system. If what it gets can't satisfy the request, it fails.
- ▶ `free()` checks if the blocks adjacent to the freed block are also free
 - If so, adjacent free blocks are merged (**coalesced**) into a single, larger free block
 - Otherwise, the freed block is just added to the free list

Choosing a block in `malloc()`

- ▶ If there are multiple free blocks of memory that are big enough for some request, how do we choose which one to use?
 - **best-fit**: choose the smallest block that is big enough for the request
 - **first-fit**: choose the first block we see that is big enough
 - **next-fit**: like first-fit but remember where we finished searching and resume searching from there

Tradeoffs of allocation policies

- ▶ **Best-fit:** Tries to limit fragmentation but at the cost of time (must examine all free blocks for each malloc).
 - Leaves lots of small blocks (why?)
- ▶ **First-fit:** Quicker than best-fit (why?) but potentially more fragmentation.
 - Tends to concentrate small blocks at the beginning of the free list (why?)
- ▶ **Next-fit:** Does not concentrate small blocks at front like first-fit, should be faster as a result.

Quiz – Pros and Cons of fits

- 1) **first-fit** results in many **small blocks** at the beginning of the free list
- 2) **next-fit** is **slower than first-fit**, since it takes longer in steady state to find a match
- 3) **best-fit** leaves lots of tiny blocks

	1	2	3
a)	F	F	T
b)	F	T	T
c)	T	F	F
d)	T	F	T
e)	T	T	T

Quiz – Pros and Cons of fits

- 1) **first-fit** results in many **small blocks** at the beginning of the free list
- 2) **next-fit** is **slower than first-fit**, since it takes longer in steady state to find a match
- 3) **best-fit** leaves lots of tiny blocks

	1	2	3
a)	F	F	T
b)	F	T	T
c)	T	F	F
d)	T	F	T
e)	T	T	T

Summary

- ▶ C has 3 pools of memory
 - Static storage: global and static variable storage, basically permanent, entire program run
 - The Stack: local variable storage, parameters, return address
 - The Heap (dynamic storage): `malloc()` grabs space from here, `free()` returns it.
- ▶ `malloc()` handles free space with freelist. Three different ways to find free space when given a request:
 - **First fit** (find first one that's free)
 - **Next fit** (same as first, but remembers where left off)
 - **Best fit** (finds most “snug” free space)

Slab Allocator

- ▶ A different approach to memory management (used in GNU `libc`)
- ▶ Divide blocks in to “large” and “small” by picking an arbitrary threshold size (say 128kB). Blocks larger than this threshold are managed with a **freelist** (as before).
- ▶ For small blocks, allocate blocks in sizes that are powers of 2
 - e.g., if program wants to allocate 20 bytes, actually give it 32 bytes


Slab Allocator

- ▶ Bookkeeping for small blocks is relatively easy
 - Use a **bitmap** for each range of blocks of the same size
- ▶ Allocating is easy and fast
 - Compute the size of the block to allocate and find a free bit in the corresponding bitmap.
- ▶ Freeing is also easy and fast
 - Figure out which slab the address belongs to and clear the corresponding bit.

Slab Allocator

16 byte blocks: 

32 byte blocks: 

64 byte blocks: 

16 byte block bitmap: 11011000

32 byte block bitmap: 0111

64 byte block bitmap: 00

Slab Allocator Tradeoffs

- ▶ Extremely fast for small blocks.
- ▶ Slower for large blocks
 - But presumably the program will take more time to do something with a large block, so the overhead is not as critical.
- ▶ Minimal space overhead
- ▶ No external fragmentation (as we defined it before)
 - For small blocks, but still have wasted space!

Internal vs. External Fragmentation

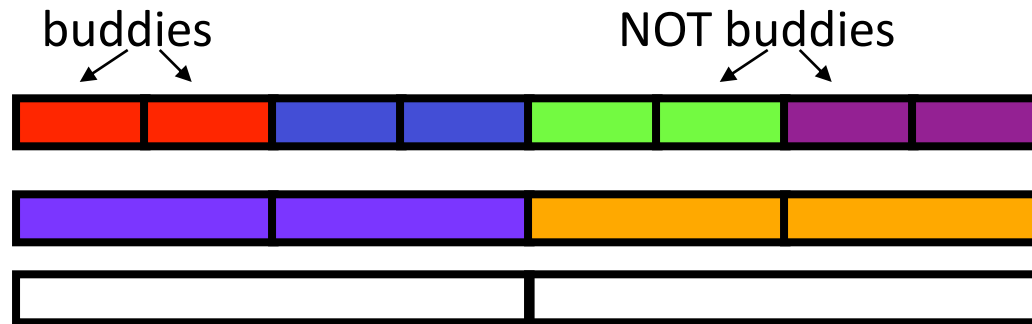
- ▶ With the slab allocator, difference between requested size and next power of 2 is wasted
 - e.g., if program wants to allocate 20 bytes and we give it a 32 byte block, 12 bytes are unused.
- ▶ We also refer to this as fragmentation, but call it **internal fragmentation** since the wasted space is actually within an allocated block.
- ▶ **External fragmentation**: wasted space between allocated blocks.

Buddy System

- ▶ Yet another memory management technique (used in Linux kernel)
- ▶ Like GNU's "slab allocator", but only allocate blocks in sizes that are powers of 2 (internal fragmentation is possible)
- ▶ Keep separate free lists for each size
 - e.g., separate free lists for 16 byte, 32 byte, 64 byte blocks, etc.

Buddy System

- ▶ If no free block of size n is available, find a block of size $2n$ and split it in to two blocks of size n
- ▶ When a block of size n is freed, if its neighbor of size n is also free, combine the blocks into a single block of size $2n$
 - **Buddy** is a block in other half larger block



- ▶ Same speed advantages as slab allocator

Allocation Schemes

- ▶ So which memory management scheme (K&R, slab, buddy) is best?
 - There is no single best approach for every application.
 - Different applications have different allocation / deallocation patterns.
 - A scheme that works well for one application may work poorly for another application.

Automatic Memory Management

- ▶ Dynamically allocated memory is difficult to track
 - Why not track it **automatically**?
- ▶ If we can keep track of what memory is in use, we can reclaim everything else.
 - Unreachable memory is called **garbage**, the process of reclaiming it is called **garbage collection**.
- ▶ So how do we track what is in use?

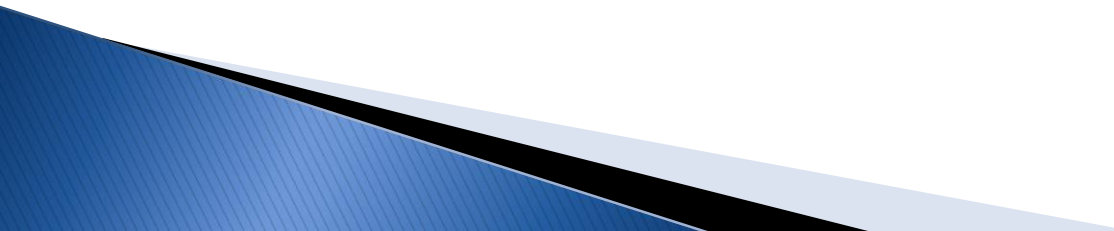
Tracking Memory Usage

- ▶ Techniques depend heavily on the programming language and rely on help from the compiler.
- ▶ Start with all pointers in global variables and local variables ([root set](#)).
- ▶ Recursively examine dynamically allocated objects we see a pointer to.
 - We can do this in **constant space** by reversing the pointers on the way down
- ▶ How do we recursively find pointers in dynamically allocated memory?

Tracking Memory Usage

- ▶ Again, it depends heavily on the programming language and compiler.
- ▶ Could have only a single type of dynamically allocated object in memory
 - E.g., simple Lisp/Scheme system with only `cons` cells
- ▶ Could use a **strongly typed** language (e.g., Java)
 - Don't allow conversion (casting) between arbitrary types.
 - C/C++ are not strongly typed.
- ▶ We will cover 3 schemes to collect garbage

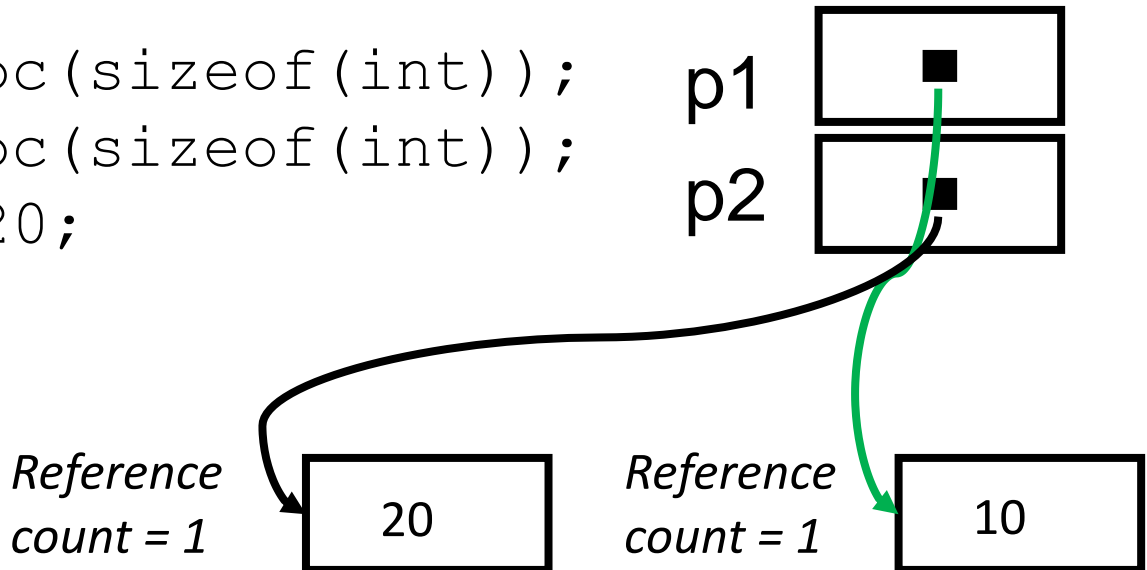
Scheme 1: Reference Counting

- ▶ For every chunk of dynamically allocated memory, keep a count of number of pointers that point to it.
 - ▶ When the count reaches 0, reclaim the memory.
 - ▶ Simple assignment statements can result in a lot of work, since may update reference counts of many items
- 

Reference Counting Example

- ▶ For every chunk of dynamically allocated memory, keep a count of number of pointers that point to it.
 - When the count reaches 0, reclaim.

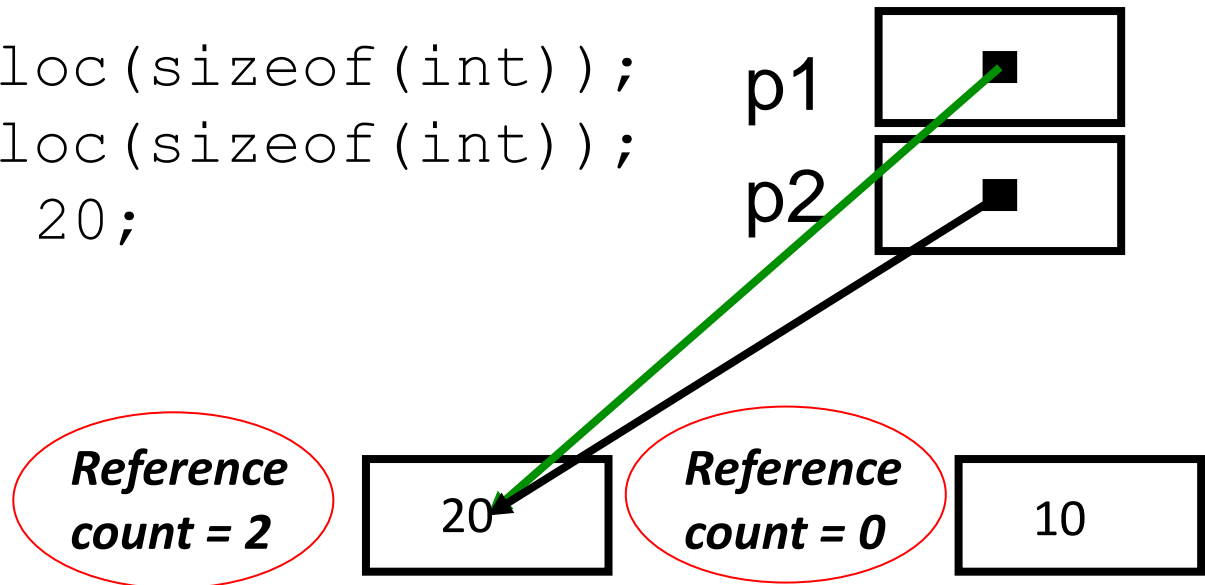
```
int *p1, *p2;  
p1 = (int *)malloc(sizeof(int));  
p2 = (int *)malloc(sizeof(int));  
*p1 = 10; *p2 = 20;
```



Reference Counting Example

- ▶ For every chunk of dynamically allocated memory, keep a count of number of pointers that point to it.
 - When the count reaches 0, reclaim.

```
int *p1, *p2;  
p1 = (int *)malloc(sizeof(int));  
p2 = (int *)malloc(sizeof(int));  
*p1 = 10; *p2 = 20;  
p1 = p2;
```



Reference Counting ($p1$, $p2$ are pointers)

$p1 = p2;$

- ▶ Increment reference count for $p2$
- ▶ If $p1$ held a valid value, decrement its reference count
- ▶ If the reference count for $p1$ is now 0, reclaim the storage it points to.
 - If the storage pointed to by $p1$ was pointing to other pointers, decrement all their reference counts, and so on...
- ▶ Must also decrement reference count when local variables cease to exist.