CSE 31
Computer Organization

Lecture 9 – C Memory Management (cont.)

Announcements

- 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 (with 7 days grace period after due date)
    - » You must demo your submission to your TA within 14 days from posting of lab
    - » Demo is REQUIRED to receive full credit
- Reading assignments
  - Reading 02 (zyBooks 2.1 – 2.9) due 22-FEB
    - » 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
- Homework assignment
  - Homework 01 (zyBooks 1.1 – 1.5) due 22-FEB
    - » 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

Announcements

- Project 01
  - Due 17-MAR
  - Can work in teams of 2 students
    - » Each team member must identify teammate in “Comments…” text-box at the submission page
    - » If working in teams, each student must submit code (can be the same as teammate) and demo individually
    - » Grade can vary among teammates depending on demo
  - Demo required for project grade
    - » No partial credit for submission without demo
  - No grace period
    - » Must complete submission and demo by due date.

The Heap (Dynamic memory)

- Large pool of memory, not allocated in contiguous order
  - back-to-back requests for heap memory could result in blocks very far apart
  - where Java/C++ ***new*** command allocates memory

- In C, specify number of bytes of memory explicitly to allocate item

```
int *ptr;
ptr = (int *) malloc(10*sizeof(int));
/* malloc returns type (void *),
so need to cast to right type */
```

  - `malloc()`: Allocates raw, uninitialized memory from heap

Memory Management

- How do we manage memory?
  - Code/Text, Static
    - » Simple
    - » They never grow or shrink
  - Stack
    - » Simple
    - » Stack frames are created and destroyed in last-in, first-out (LIFO) order
  - Heap
    - » Tricky
    - » Memory can be allocated / deallocated at any time

Heap Management Requirements

- Want `malloc()` and `free()` to run quickly.

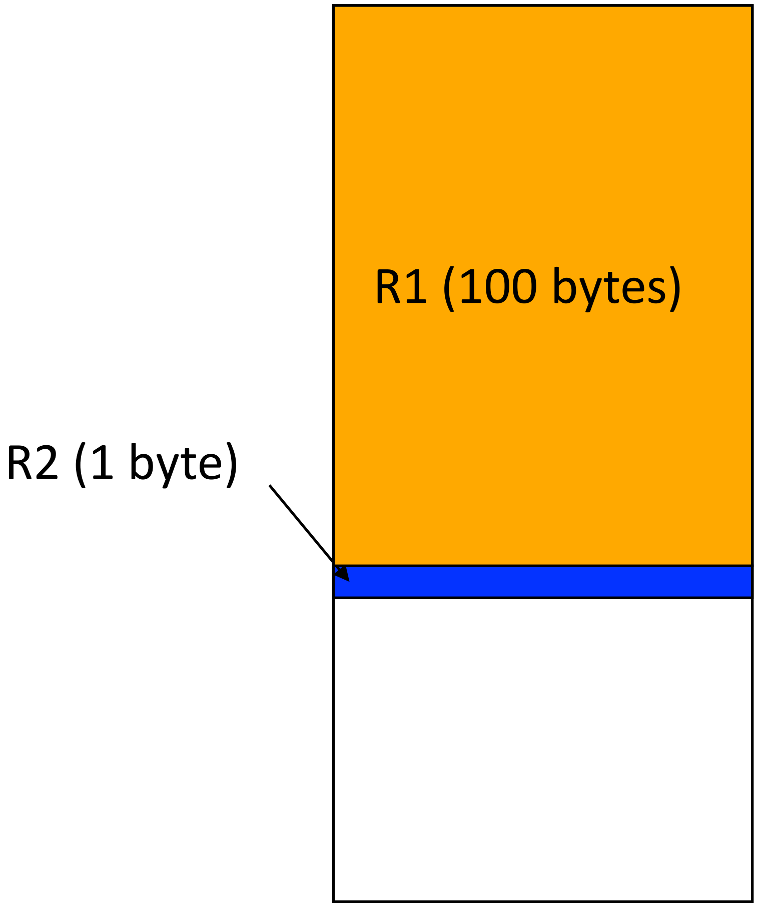
- Want minimal memory overhead

- Want to avoid fragmentation*
  - When most of our free memory is in many small chunks
  - In this case, we might have many free bytes but not be able to satisfy a large request since the free bytes are not contiguous in memory.

* This is technically called *external fragmentation*

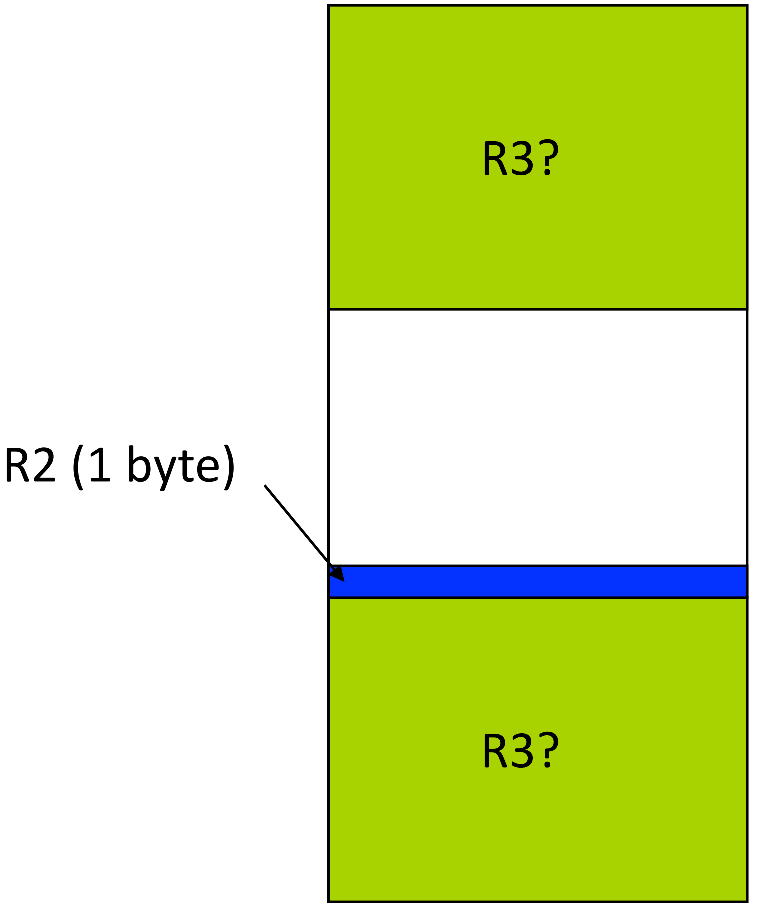
Heap Management

- An example
  - Request R1 for 100 bytes
  - Request R2 for 1 byte
  - Memory from R1 is freed



Heap Management

- An example
  - Request R1 for 100 bytes
  - Request R2 for 1 byte
  - Memory from R1 is freed
  - Request R3 for 50 bytes



K&R Malloc/Free Implementation

- From Section 8.7 of K&R
  - Code in the book uses some C language features we haven't discussed and is written in a very terse style, don't worry if you can't decipher the code

- 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

K&R Implementation

- `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

	123
a)	FFT
b)	FTT
c)	TFF
d)	TFT
e)	TTT

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

	123
a)	FFT
b)	FTT
c)	TFF
d)	**TFT**
e)	TTT

Summary

- C has 3 pools of memory
  - Static storage: global 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 a 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

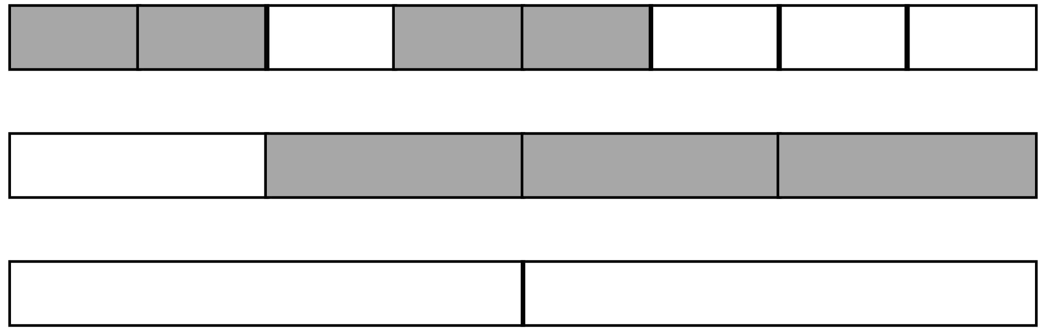
- Bookkeeping for small blocks is relatively easy
  - Use a bitmap for each range of blocks of the same size
- Allocating is easy and fast
  - Compute the size of the block to allocate and find a free bit in the corresponding bitmap.
- Freeing is also easy and fast
  - Figure out which slab the address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

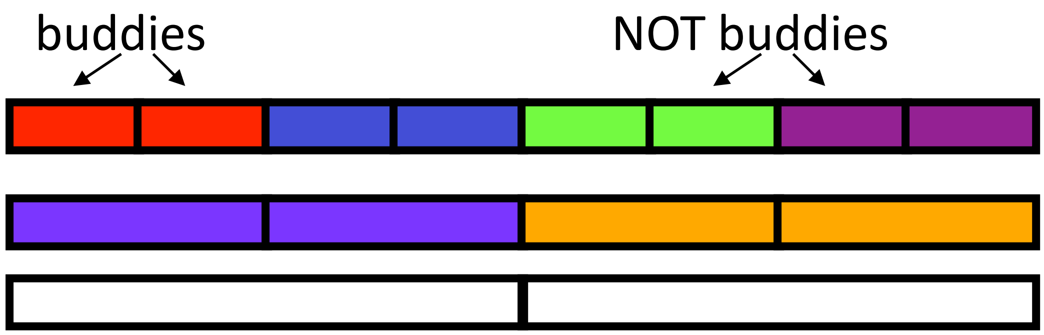
- Like GNU's "slab allocator", but only allocate blocks in sizes that are powers of 2 (internal fragmentation is possible)

- Initially entire memory space treated as single block whose size is power of 2
  - If **request size** > **0.5 * initial block size**, allocate entire block
  - Otherwise, split block in 2 companion buddies
  - If **request size** > **0.5 * buddy size**, allocate entire buddy
  - Otherwise, split one buddy in half again
  - Repeat till smallest block >= size of request is found

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 the other half of a larger block



- Same speed advantages as slab allocator

Buddy System

Buddy system at work, considering a 1024k (1-megabyte) initial block and the process requests as shown at the left of the table.

0 128k 256k 512k 1024k

start	1024k					
A=70K	A	128		256		512
B=35K	A	B	64	256		512
C=80K	A	B	64	C	128	512
A ends	128	B	64	C	128	512
D=60K	128	B	D	C	128	512
B ends	128	64	D	C	128	512
D ends	256			C	128	512
C ends	512					512
end	1024k					

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

- We will cover 3 schemes to collect garbage