C Programming Language:
Memory Management: malloc/free, first/next/best fit

Math 230

Assembly Language Programming
(Computer Organization)

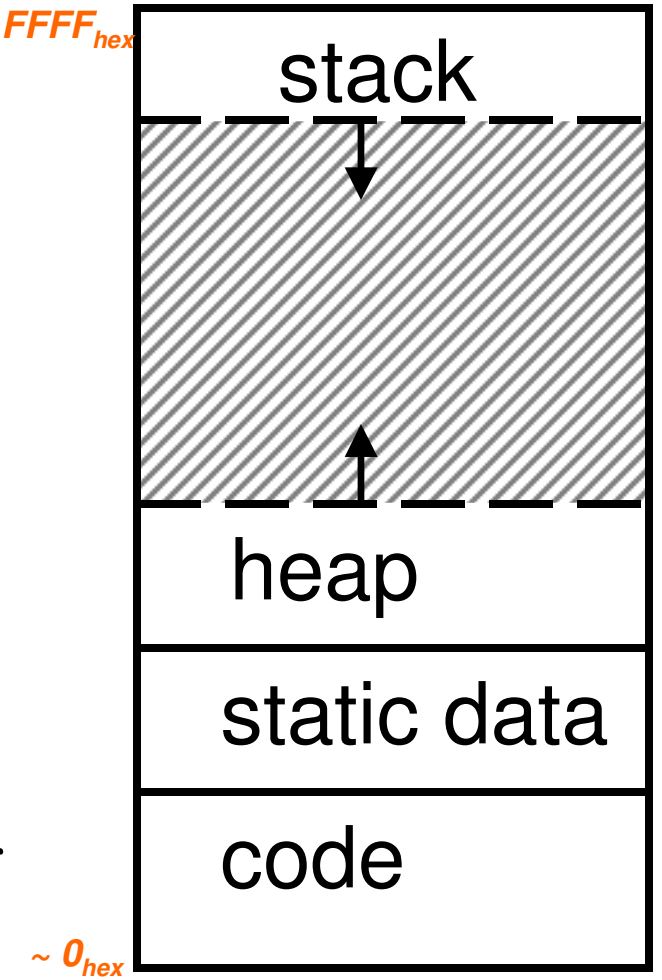
Thu. Feb 7, 2008

Overview

- malloc
  - under the hood
  - first-fit, best-fit, next-fit, buddy system

Recall: Normal C Memory Management

- 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: variables declared outside main, does not grow or shrink
  - code: loaded when program starts, does not change



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

slide from DGarcia CS61C UC Berkeley

Recall: Memory Management

- How do we manage memory?
- Code, Static storage are easy:
  they never grow or shrink
- Stack space is also easy:
  stack frames are created and destroyed in last-in, first-out (LIFO) order
- Managing the heap is tricky:
  memory can be allocated / deallocated at any time

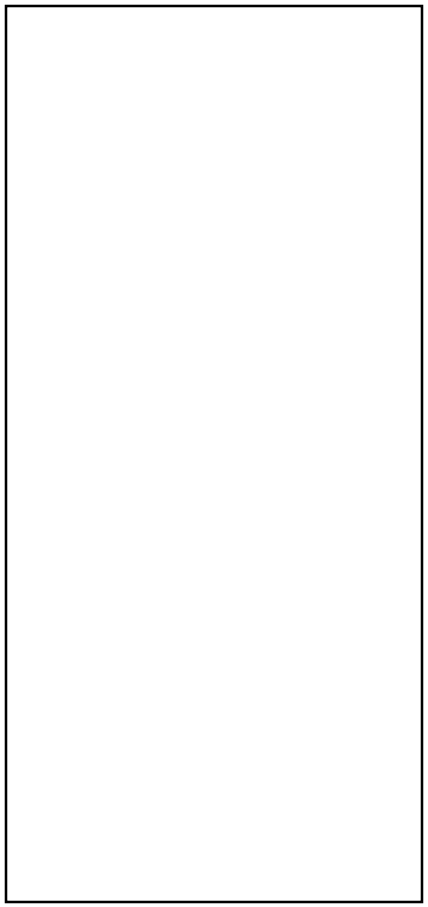
Heap Management Requirements

- 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 fragmention*

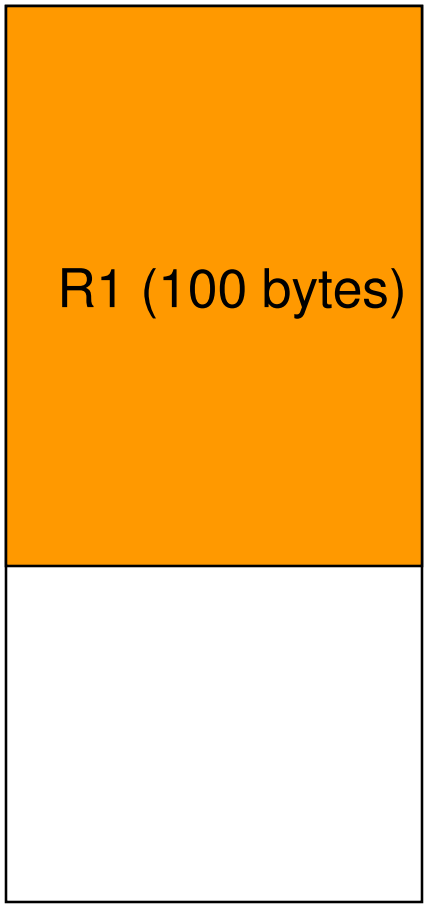
Heap Management

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



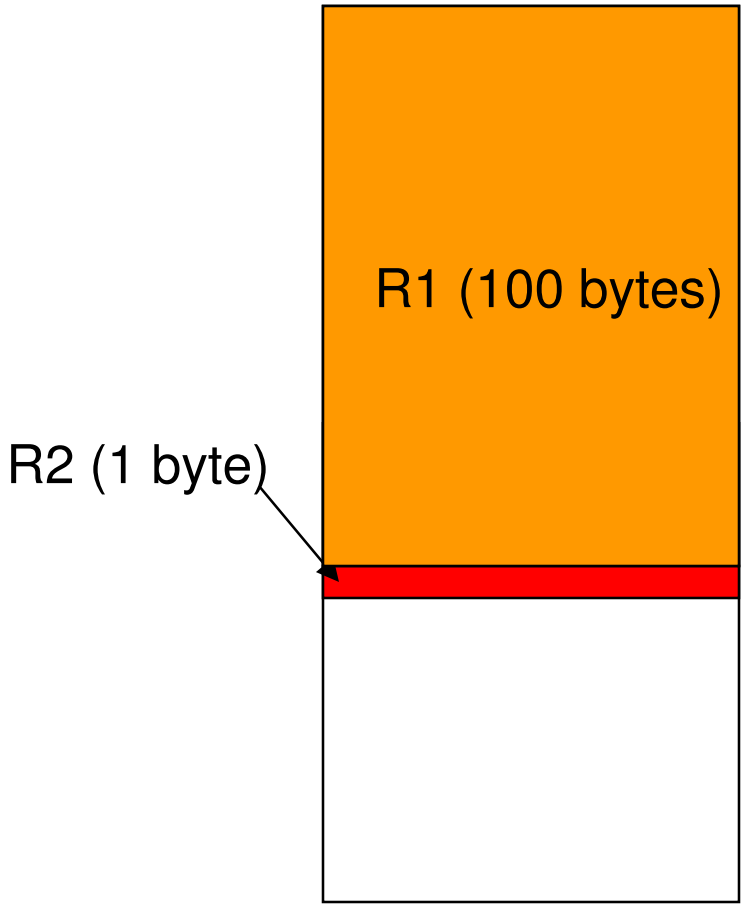
Heap Management

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



Heap Management

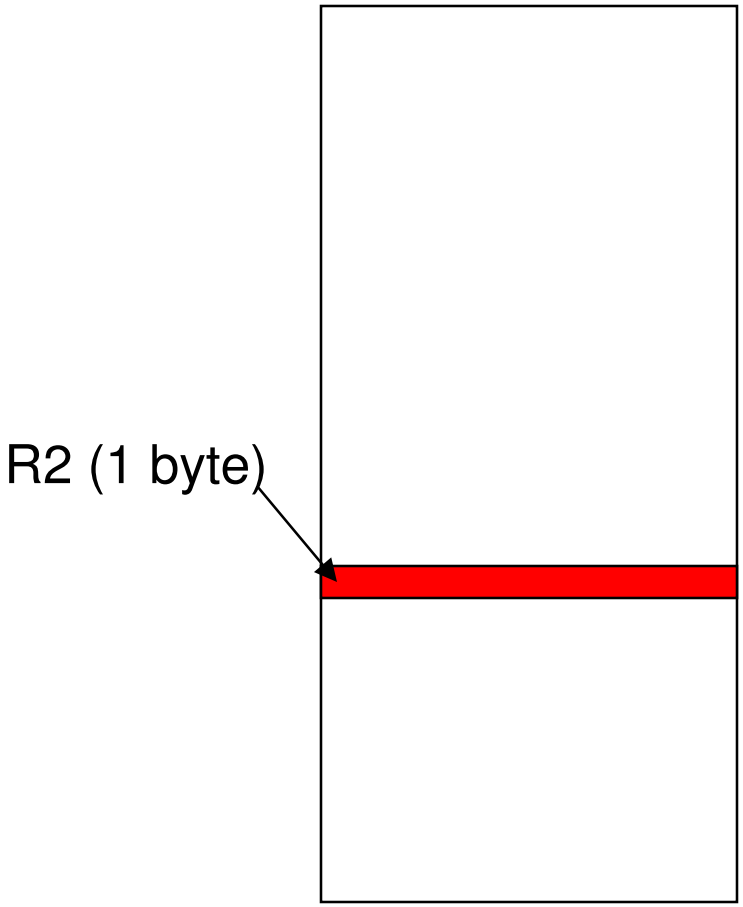
- An example
  - Request R1 for 100 bytes
  - Request R2 for 1 byte
  - Memory from R1 is freed
  - Request R3 for 50 bytes



slide from DGarcia CS61C UC Berkeley

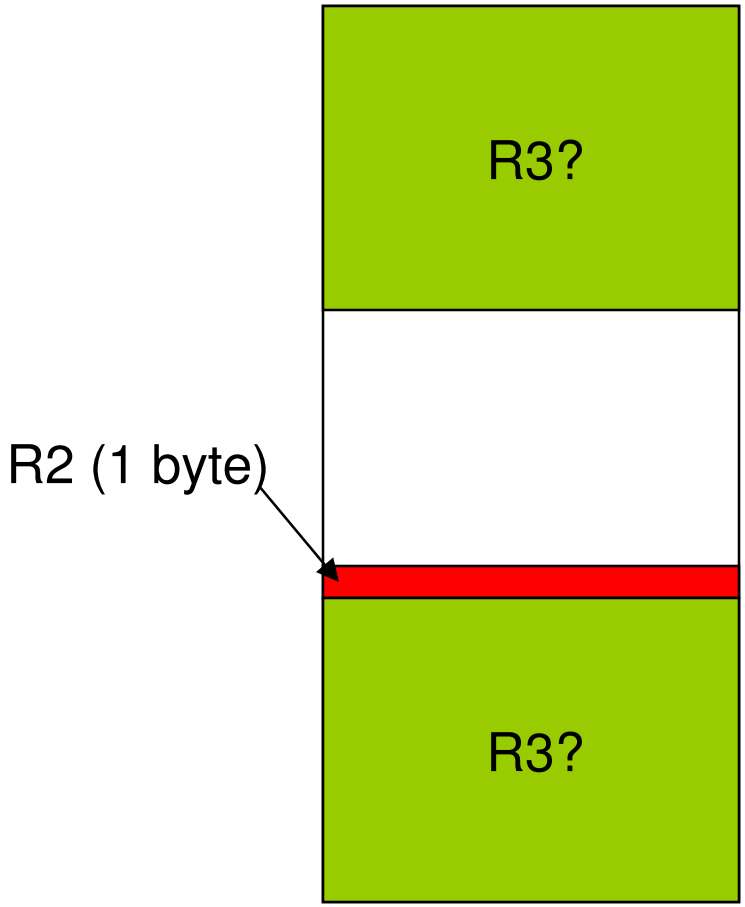
Heap Management

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



Heap Management

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



slide from DGarcia CS61C UC Berkeley

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 and
  a pointer to the next block
- All free blocks are kept in a linked list, the pointer field is unused in an allocated block

slide from DGarcia CS61C UC Berkeley

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

slide from DGarcia CS61C UC Berkeley

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

slide from DGarcia CS61C UC Berkeley

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.

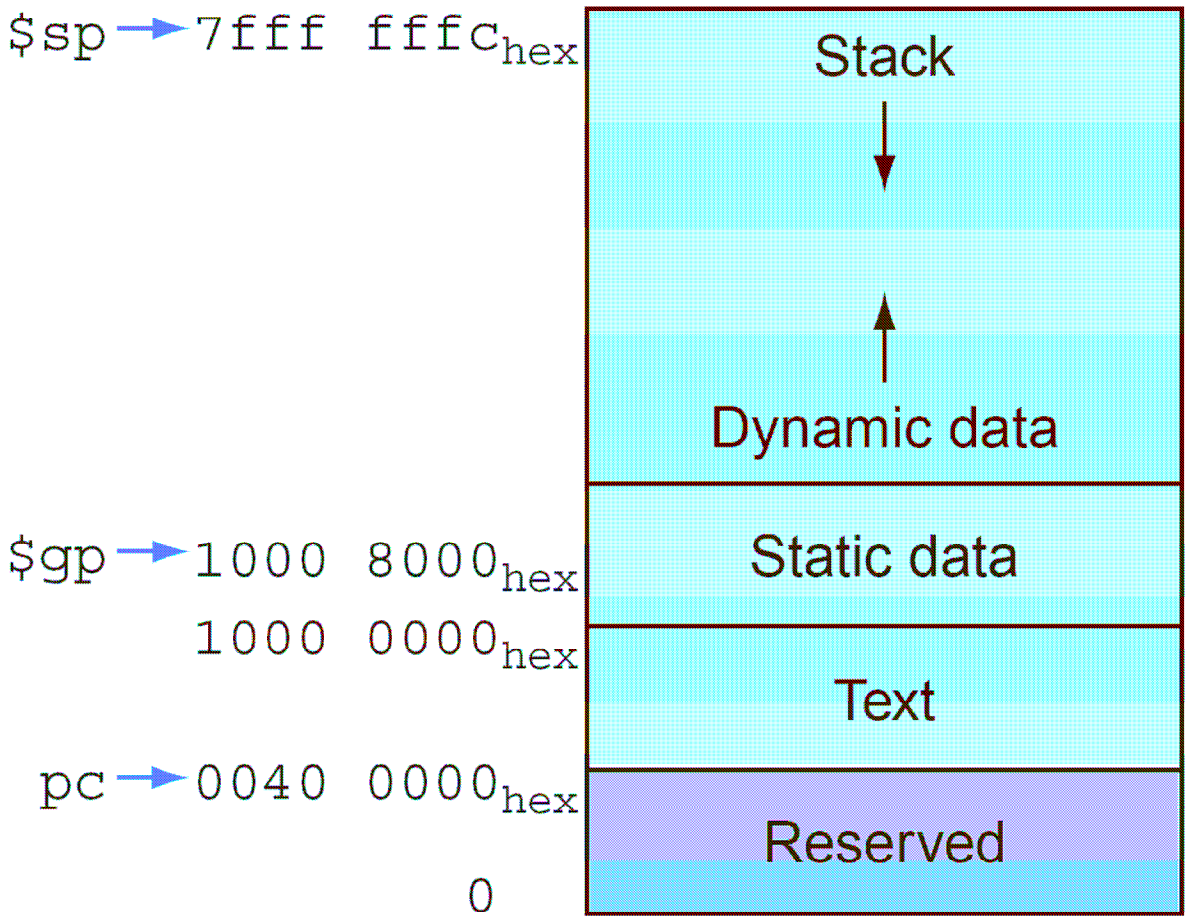
slide from DGarcia CS61C UC Berkeley

And in conclusion…

- 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)

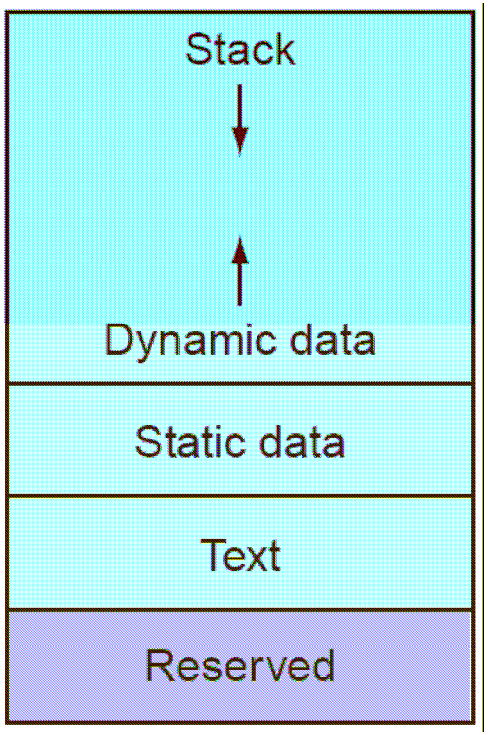
slide from DGarcia CS61C UC Berkeley

Memory Model



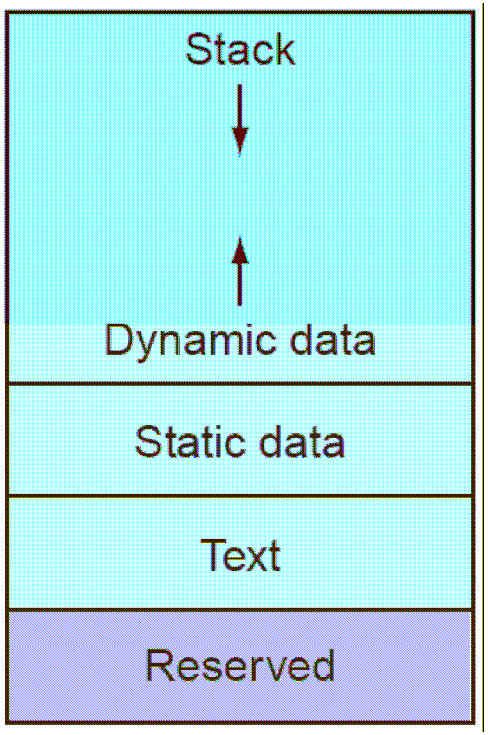
Memory Model

- Needed memory space
  - automatic variables local to procedures
    - automatically allocated upon function entry
  - Static variables
    - exists independent of any function; generally global in scope
  - dynamically allocated
    - requested and delivered as needed



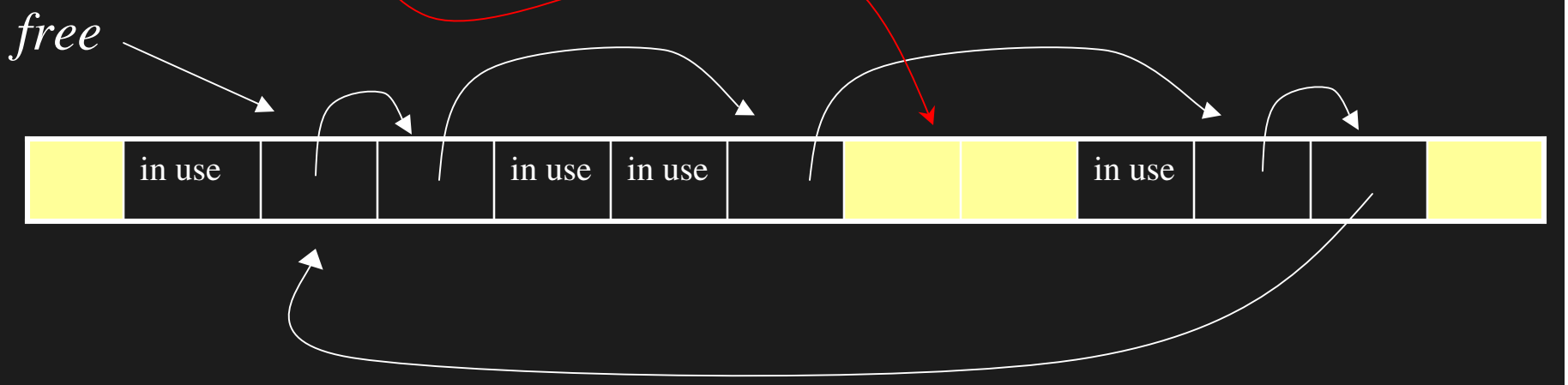
MIPS memory conventions

- stack
  - memory used from the top down
- heap
  - memory used from lower to higher, bottom up
- machine code
  - resides in memory even lower than the heap
  - traditionally called the “text” segment



Memory Mgmt Via malloc

- malloc manages memory requested by malloc
- other programs may have also requested their ***own space***, not managed by malloc
  - malloc may not have contiguous blocks
- malloc keeps track via linked lists



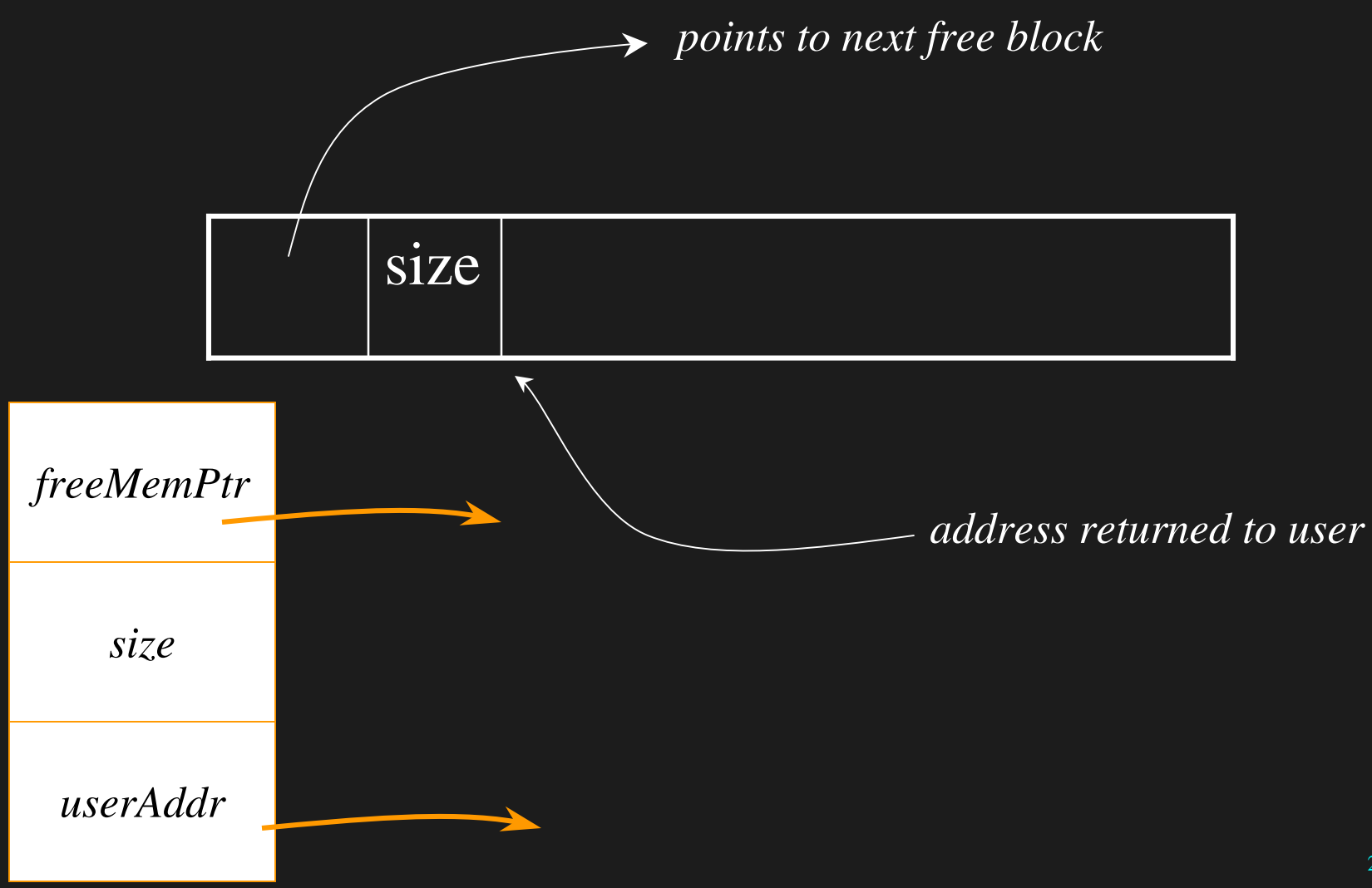
malloc

- memory mgmt algorithms
  - first fit
    - free list scanned/walked until big-enough block is found
  - best fit
    - looks for smallest block of memory to meet request
    - exact sized blocks are unlinked from the list and given to (claimed for) the user
    - big blocks are split and proper amount returned to user
      - rest remains on free list
    - if necessary, malloc will link in additional memory from the OS

freeing memory

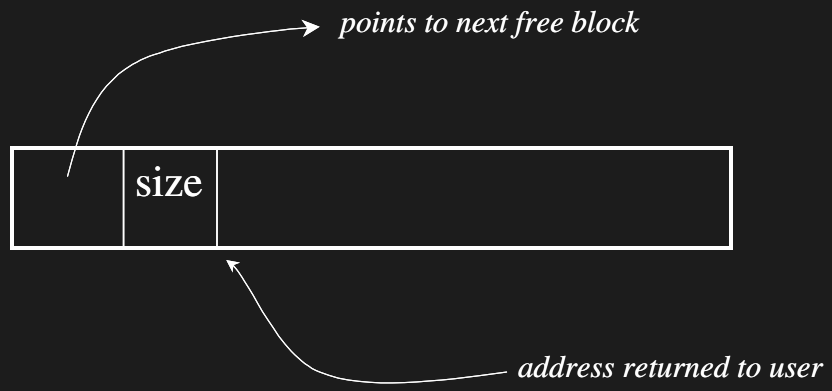
- when memory is freed, the list allocated must be returned to malloc's working list
- returned block may be "merged" (coalesced) with block next to it
  - helps with fragmentation

malloc blocks



malloc blocks II

- free blocks
  - pointer to next block in chain
  - size of block
  - available space follows
    - malloc'ed memory items are "tagged"



malloc blocks III

```
union header {
  struct {
     unsigned size;
     union header *ptr;
  } s;
};
typedef union header Header;
```

Summary