CSE 351: The Hardware/Software Interface

Section 9

Lab 5

Dynamic memory allocation

* In order to allocate memory that persists across function calls, one can use `malloc` in C to request heap space of a particular size
* Unlike with stack-allocated memory, `malloc`ed memory persists until it is explicitly returned to the C library with a call to `free`

malloc: behind the scenes

* As a process allocates memory through `malloc`, the C library makes requests to the operating system to increase the size of its data segment
  * This is accomplished via calls to `sbrk` (see `man 2 sbrk`), which changes the location of the “program break” denoting the end of the data segment
* When a process invokes `malloc`, the C library returns the address of an unused data block somewhere inside of the data segment

free: behind the scenes

- When a process `free`s a block of memory, that block is marked as available and can now be reused through subsequent calls to `malloc`
- To watch this happen in practice, try using GDB on a program that allocates and frees a block of memory using `malloc` and `free`. How do the bytes immediately preceding the block of memory change over time?

Lab 5

* Memory allocator: Implement custom versions of malloc and free called mm_malloc and mm_free
* Get experience with how dynamic memory allocation works
* Think critically about memory and pointers

Free list

* The primary data structure used in lab 5 is a free list. Entries in this list store information about how large they are and where the next and previous free entries are

```
struct BlockInfo {
  size_t sizeAndTags;
  struct BlockInfo* next;
  struct BlockInfo* prev;
};
```

Free list

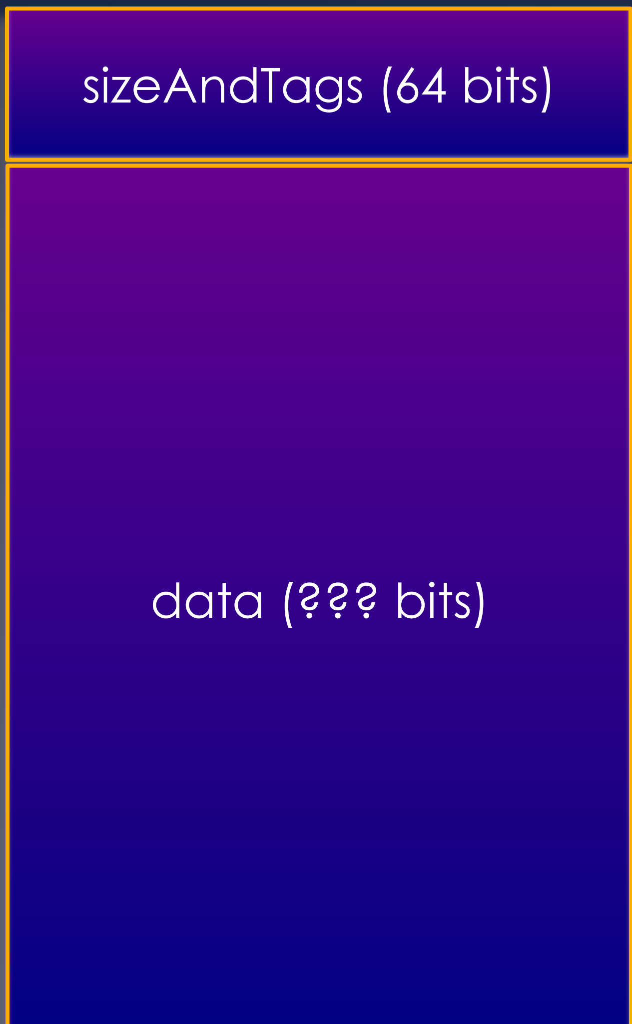
```
struct BlockInfo {
  size_t sizeAndTags;
  struct BlockInfo* next;
  struct BlockInfo* prev;
};
```

- `sizeAndTags`: The upper 61 bits store the total size of this block, the lowest bit indicates whether the block is used, and the second-lowest bit indicates whether the previous block is free. Only the upper 61 bits of the size are needed since block are 8-byte aligned
- `next` and `prev`: Pointers to the next and previous free blocks

Free block format

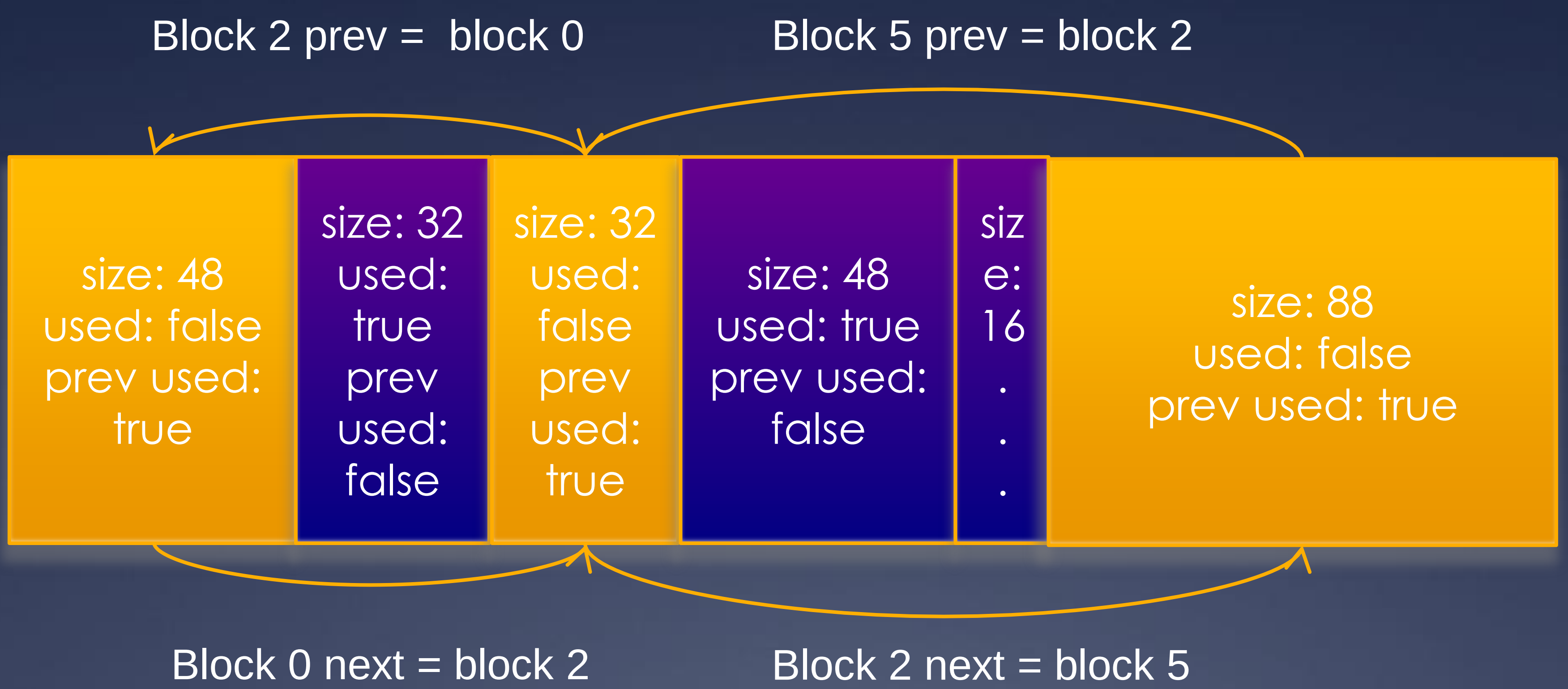
- Note that the size and tags are given at both the beginning and the end. What benefit does this provide?

Used block format



- Used blocks do not store prev and next pointers. What should happen when a used block is `mm_free`d?
- Data sections are always padded to an 8-byte boundary

Free list



Block 2 prev = block 0
Block 5 prev = block 2
size: 48 used: false prev used: true
size: 32 used: true prev used: false
size: 32 used: false prev used: true
size: 48 used: true prev used: false
siz e: 16 . . .
size: 88 used: false prev used: true
Block 0 next = block 2
Block 2 next = block 5

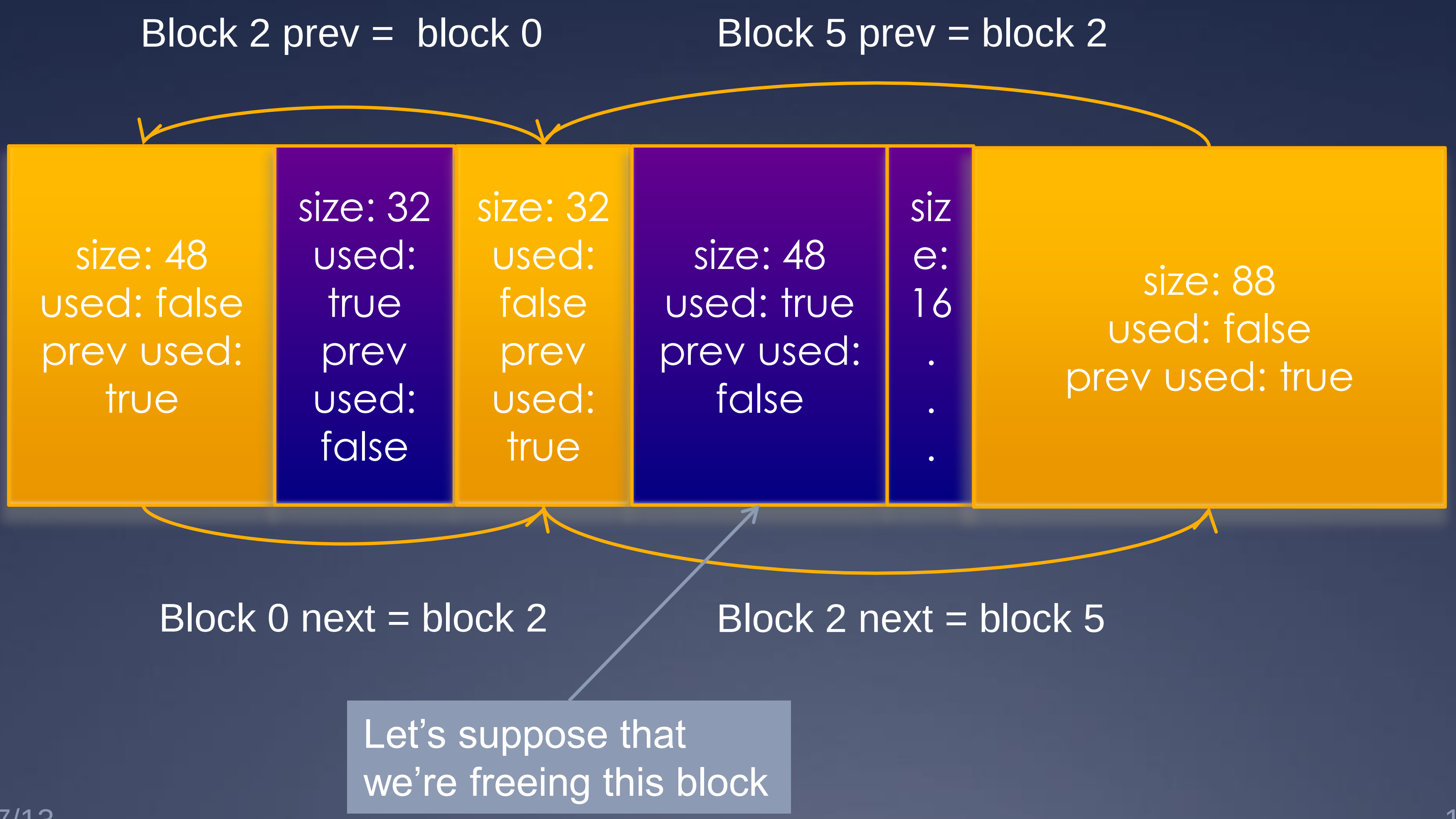
mm_malloc

- `mm_malloc` takes a single argument of how much memory to allocate
- `mm_malloc` scans through the free list, looking for a large enough unused block to fulfill the request
- If a large enough block is found, it is removed from the free list and marked as used
  - Otherwise, the program increases the size of the heap to make space for a new block to return

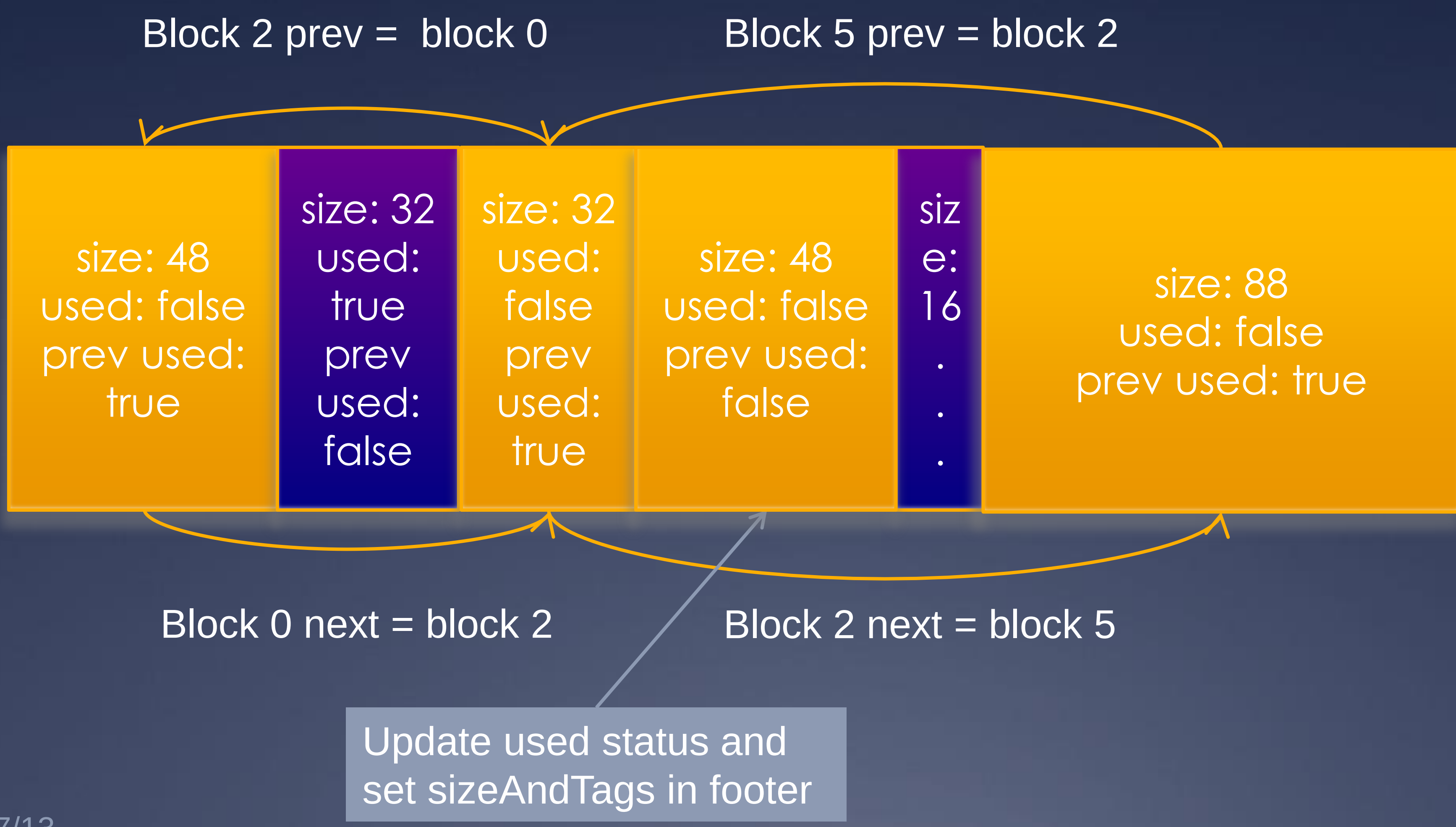
mm_free

- `mm_free` returns a now-unused block to the free list as the *head of the list*
  - Note that the “previous” and “next” blocks can actually be anywhere in memory relative to this one!
- If the blocks before or after the block in memory are also free, `mm_free` combines them into a single unused block
  - Why combine free blocks into larger ones?

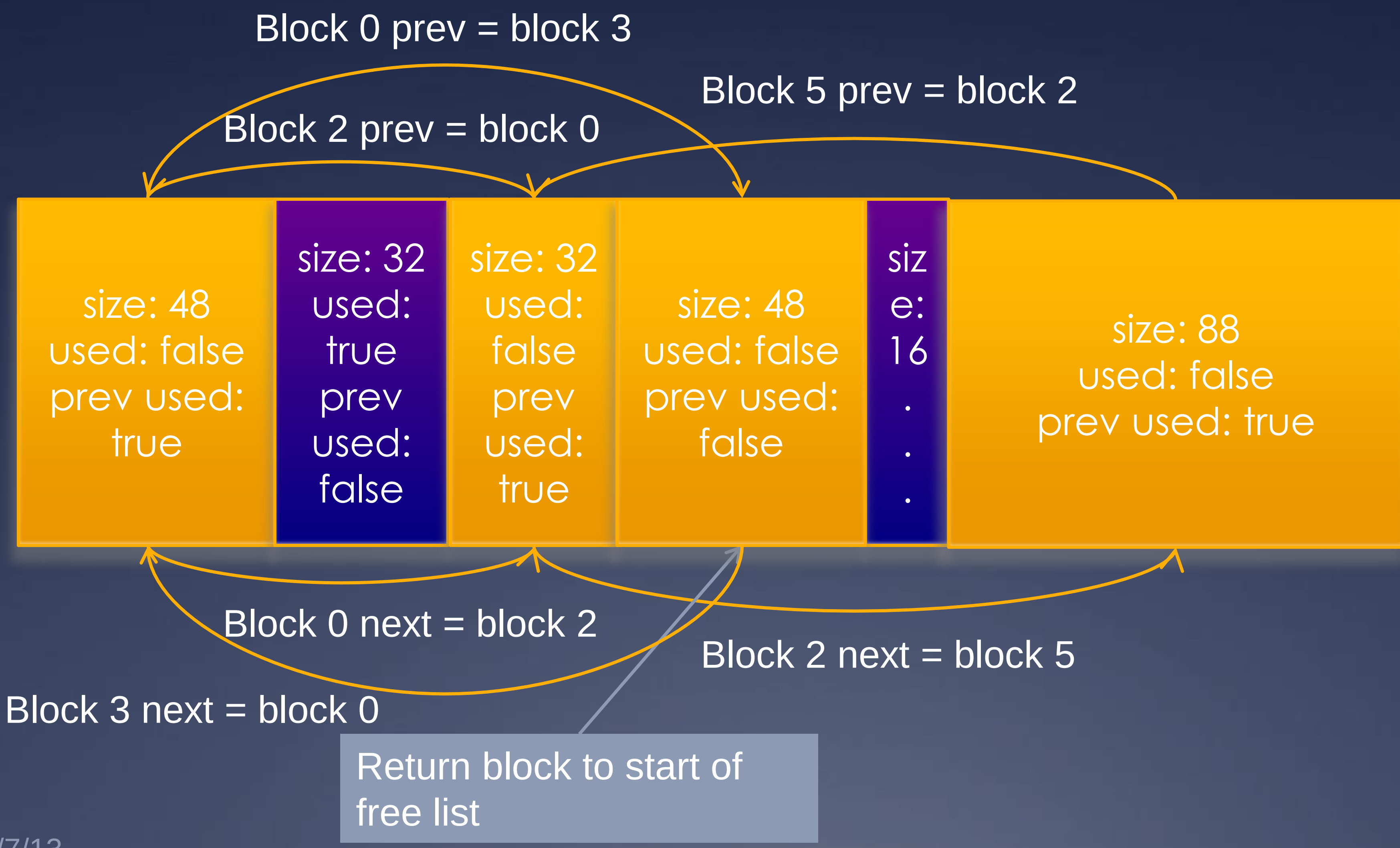
mm_free example



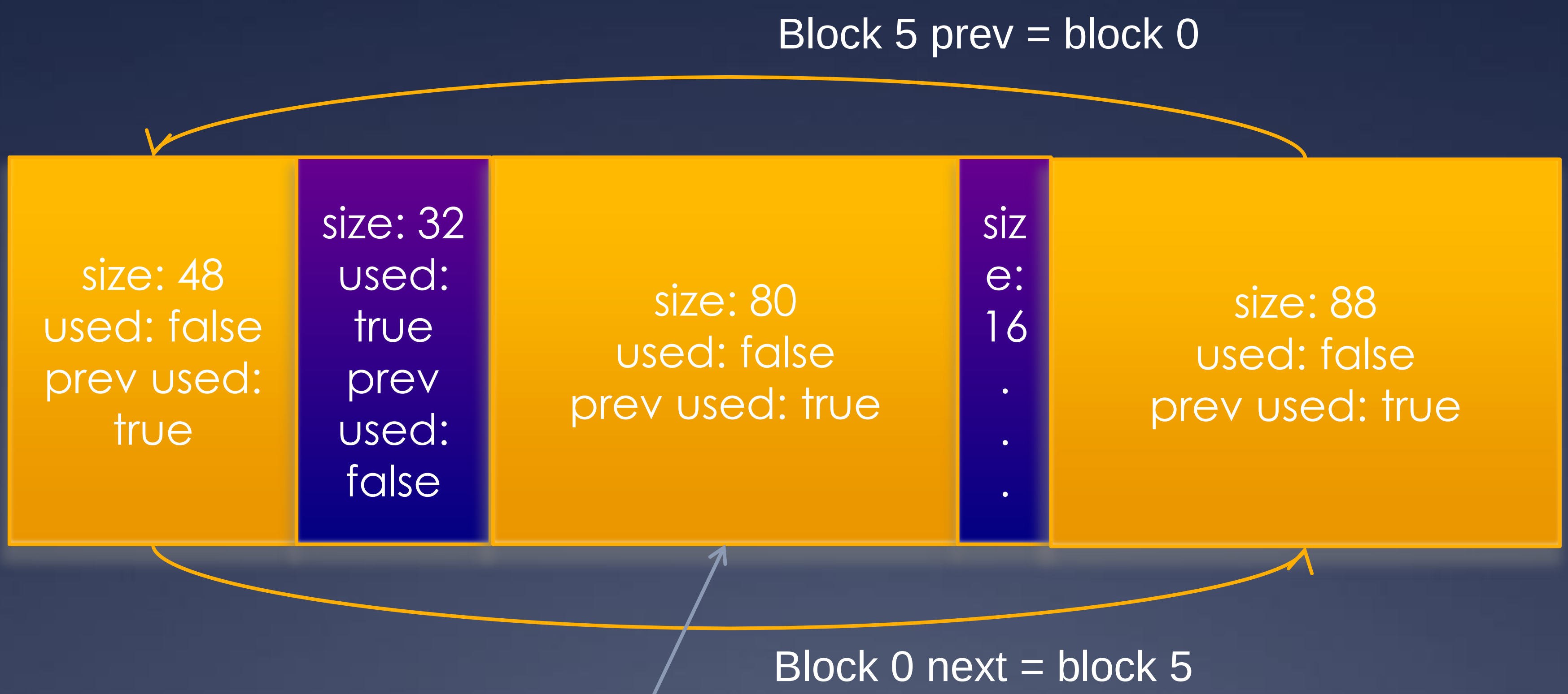
mm_free example



mm_free example

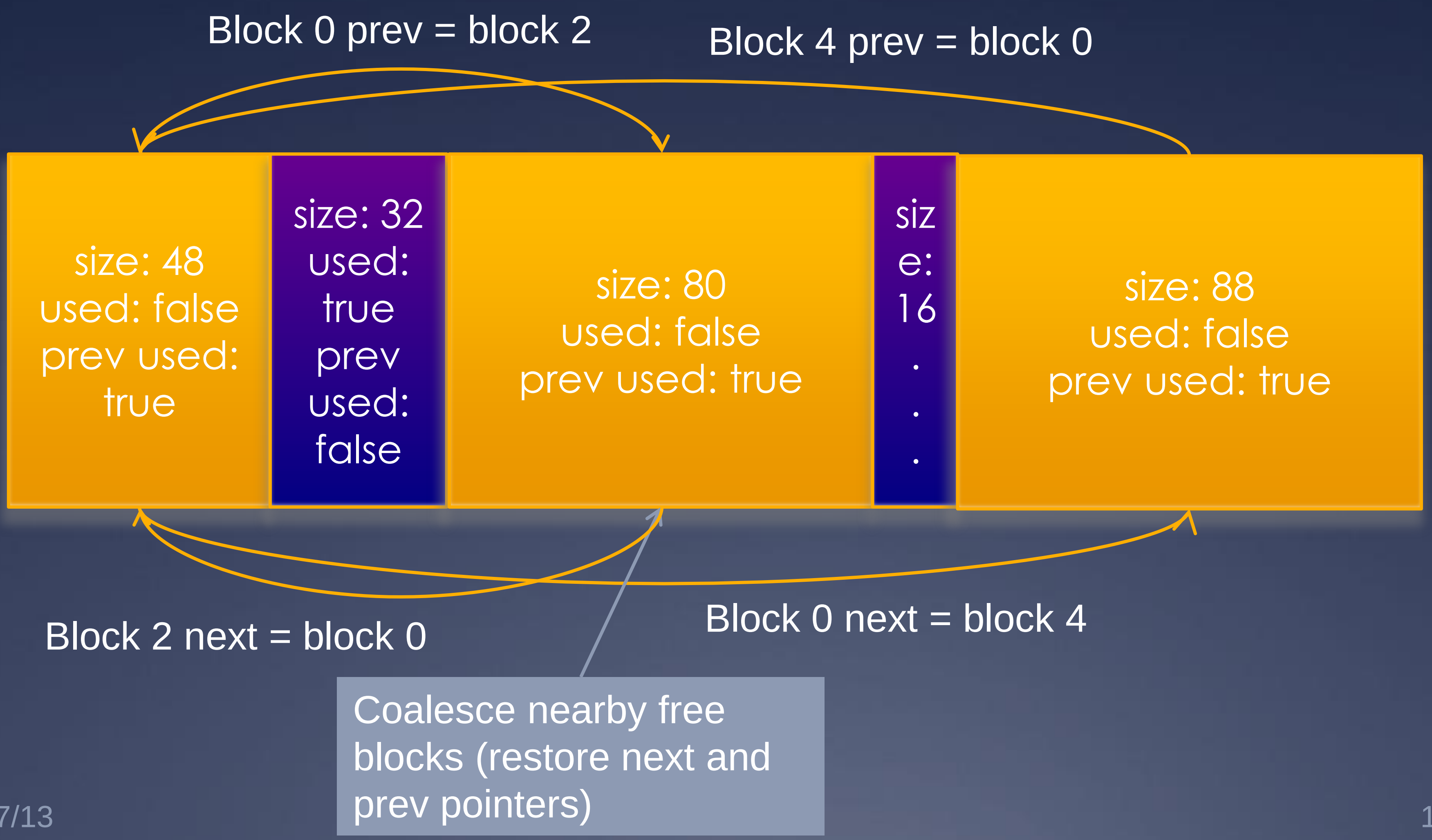


mm_free example

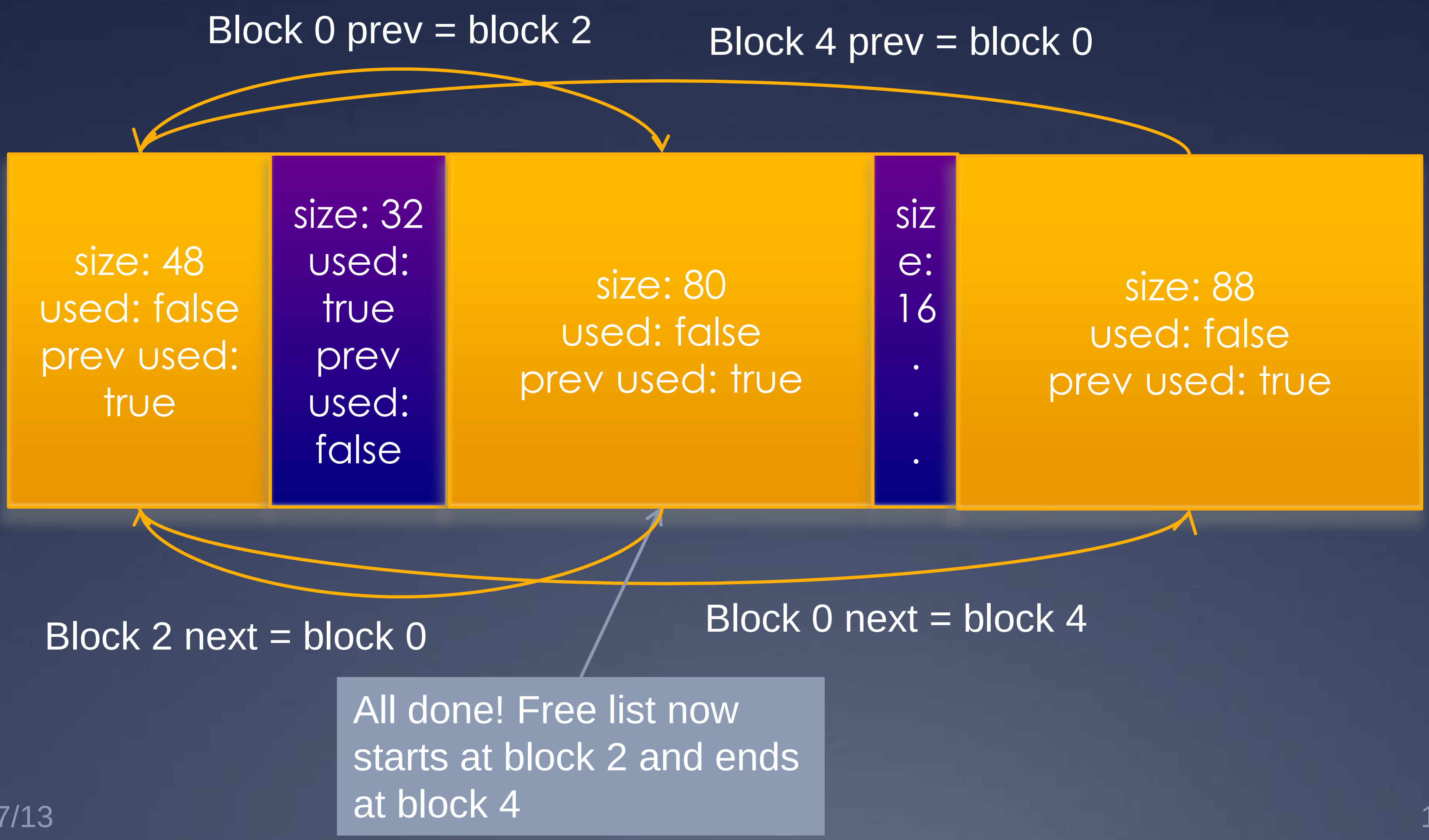


Coalesce nearby free blocks (intermediate step shown)

mm_free example



mm_free example



Words of advice

- The size portion of `sizeAndTags` can be accessed via the `SIZE()` macro. To assign the size, bitwise "or" in the existing tags and set the `sizeAndTags` field
- The preceding block is the block before this one *sequentially in memory*, not necessarily the one that the `prev` pointer refers to
- A valid solution to this assignment is not very long, but getting it right is tricky

Words of advice

- Make use of the provided functions! There is already code for searching the free list for an empty block, inserting into it, removing from it, and coalescing free nodes
- See `searchFreeList`, `insertFreeBlock`, `removeFreeBlock`, and `coalesceFreeBlock` in `mm.c`

Words of advice

- If you want to test `mm_malloc` and `mm_free` with custom code, define a new Makefile rule:

```
malloc_test: malloc_test.o mm.o memlib.o
      $(CC) $(CFLAGS) -o malloc_test \
 malloc_test.o mm.o memlib.o
malloc_test.o: malloc_test.c mm.h memlib.h
```

- Before calling `mm_malloc` for the first time, you'll need to invoke `mem_init()` from `memlib.h` and then `mm_init()` from `mm.h`
- Use `make malloc_test` to build the executable

Example program

```
#include "memlib.h"
#include "mm.h"

int main(int argc, char* argv[]) {
  mem_init();
  mm_init();
  int* a = (int*) mm_malloc(sizeof(int));
  mm_free(a);
  return 0;
}
```

Demo time

- Let’s look at the provided code for the lab
- If there is time at the end, investigate how `malloc` and `free` allocate and free memory using GDB