Memory Allocation

Computer Operating Systems, Fall 2023

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Andy Jiang	Haoyun Qin	Kevin Bernat	Ryoma Harris
Audrey Yang	Jason hom	Leon Hertzberg	Shyam Mehta
August Fu	Jeff Yang	Maxi Liu	Tina Kokoshvili
Daniel Da	Jerry Wang	Ria Sharma	Zhiyan Lu
Ernest Ng	Jinghao Zhang	Rohan Verma	

Poll Everywhere

pollev.com/tqm

- What is/was ur favourite CIS course (do not include this one)

Administrivia

- Project 1 is out now
  - Project is due 11:59 pm on Wed, Oct 11 **(1 week from tomorrow)**
    late deadline 11:59 pm on Sun, Oct 15

- For project 1 full submission, please do a group submission on gradescope (one of you submits but you add your partner to the submission)

- Recitation today on process groups, terminal control and waitpid

Poll Everywhere

pollev.com/tqm

- Any questions, comments or concerns from last lecture?

Lecture Outline

- **Heap & Stack**
- Fragmentation & Allocation Strategies
- Buddy Algorithm
- Slab Algorithm

Stack & Heap

- Hopefully you are familiar with the stack and the heap,
  - Quick refresher now though

- Stack:
  - Where local variables & information for local functions are stored (return address, etc).
  - Grows whenever you call a function. pushes a “stack frame” for each function call.

- Heap:
  - Dynamically allocated data stored here. Usually done when data needs to exist beyond the scope it is allocated in, or the size is not known at compile time

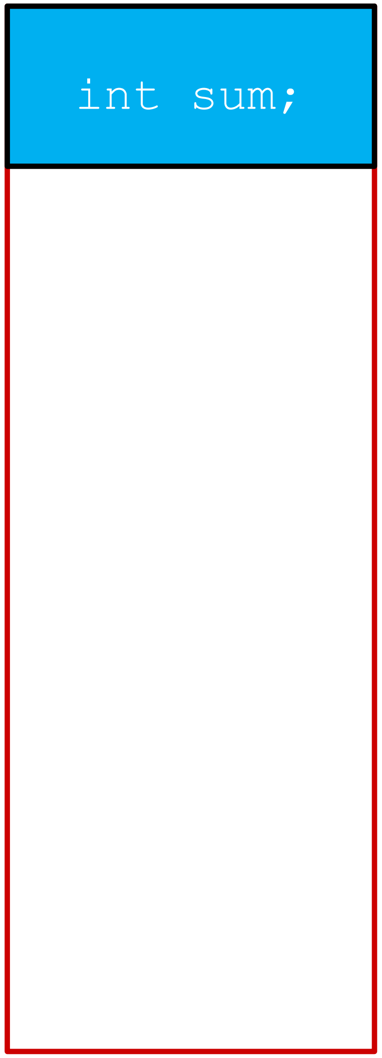
Stack Example:

Stack frame for main is created when CPU starts executing it

```c
#include <stdio.h>
#include <stdlib.h>

int sum(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

int main() {
  int sum = sum(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```

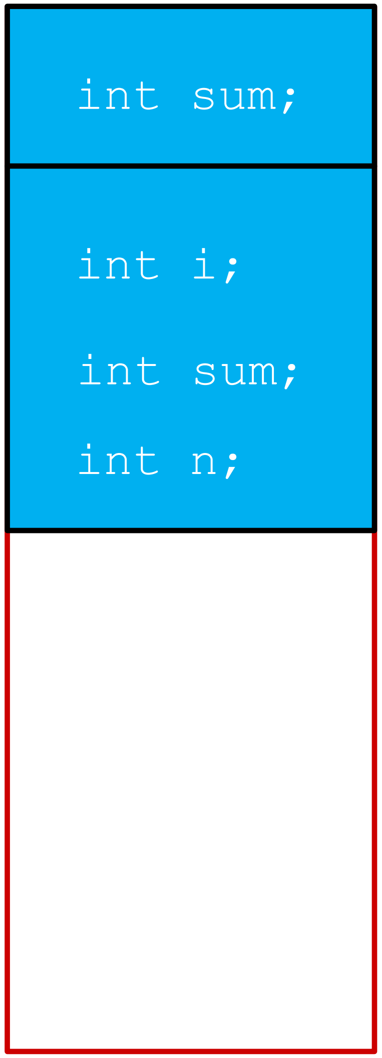


Stack Example:

```
#include <stdio.h>
#include <stdlib.h>

int sum(int n) {
   int sum = 0;
   for (int i = 0; i < n; i++) {
      sum += i;
   }
   return sum;
}

int main() {
   int sum = sum(3);
   printf("sum: %d\n", sum);
   return EXIT_SUCCESS;
}
```



Stack frame for `main()`

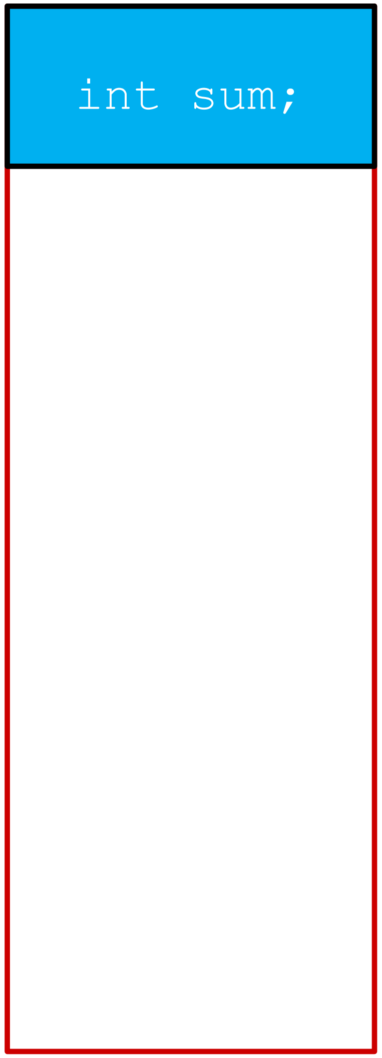
Stack frame for `sum()`

Stack Example 1:

```c
#include <stdio.h>
#include <stdlib.h>

int sum(int n) {
   int sum = 0;
   for (int i = 0; i < n; i++) {
      sum += i;
   }
   return sum;
}

int main() {
   int sum = sum(3);
   printf("sum: %d\n", sum);
   return EXIT_SUCCESS;
}
```



Stack frame for `main()`

`sum()`'s stack frame goes away after **`sum()`** returns.

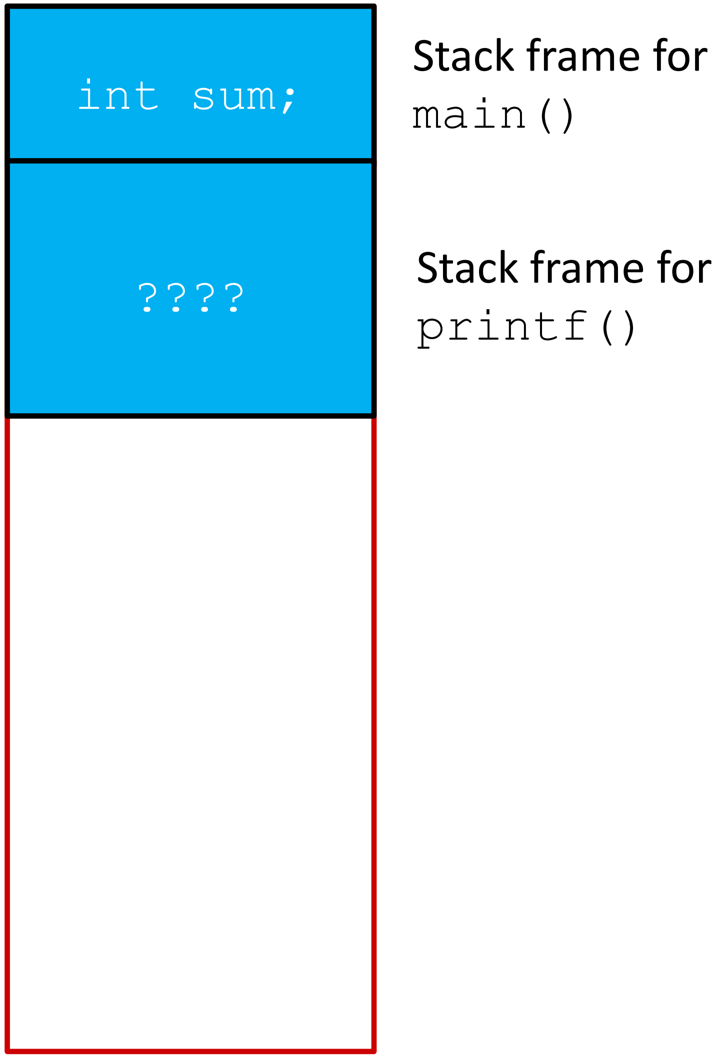
`main()`'s stack frame is now top of the stack and we keep executing **`main()`**

Stack Example:

```
#include <stdio.h>
#include <stdlib.h>

int sum(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

int main() {
  int sum = sum(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```



Stack

- Grows, but has a static max size
  - Can find the default size limit with the command `ulimit –all` (May be a different command in different shells and/or linux versions. Works in bash on Ubuntu though)
  - Can also be found at runtime with `getrlimit(3)`

- Max Size of a stack can be changed
  - at run time with `setrlimit(3)`
  - At compilation time for some systems (not on Linux it seems)
  - (or at the creation of a thread, more on threads next lecture)

The Heap

- The Heap is a large pool of available memory to use for Dynamic allocation
- This pool of memory is kept track of with a small data structure indicating which portions have been allocated, and which portions are currently available.
- **`malloc`**:
  - searches for a large enough unused block of memory
  - marks the memory as allocated.
  - Returns a pointer to the beginning of that memory
- **`free`**:
  - Takes in a pointer to a previously allocated address
  - Marks the memory as free to use.

Free Lists

- One way that malloc can be implemented is by maintaining an implicit list of the space available and space allocated.
- Before each chunk of allocated/free memory, we'll also have this metadata:

```
// this is simplified
// not what malloc really does
struct alloc_info {
  alloc_info* prev;
  alloc_info* next;
  bool allocated;
  size_t size;
};
```

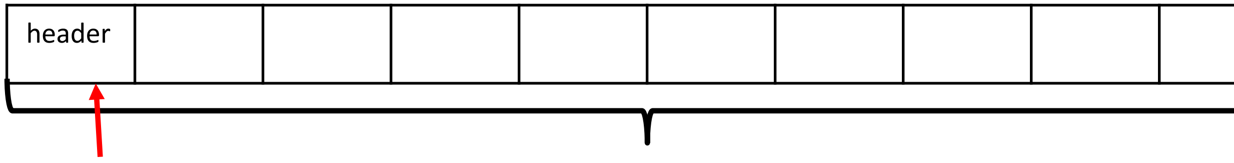
Dynamic Memory Example

```
#include <stdlib.h>

int main() {
  char* ptr = malloc(4*sizeof(char));
  char* ptr2 = malloc(6*sizeof(int));
  ...               // do stuff with ptr
  free(ptr);
  free(ptr2);
}
```

This diagram is not to scale

- free_list ->



```
{
 NULL,
 NULL,
 false,
 1024
}
```

The metadata is at the beginning of the chunk of memory

Dynamic Memory Example

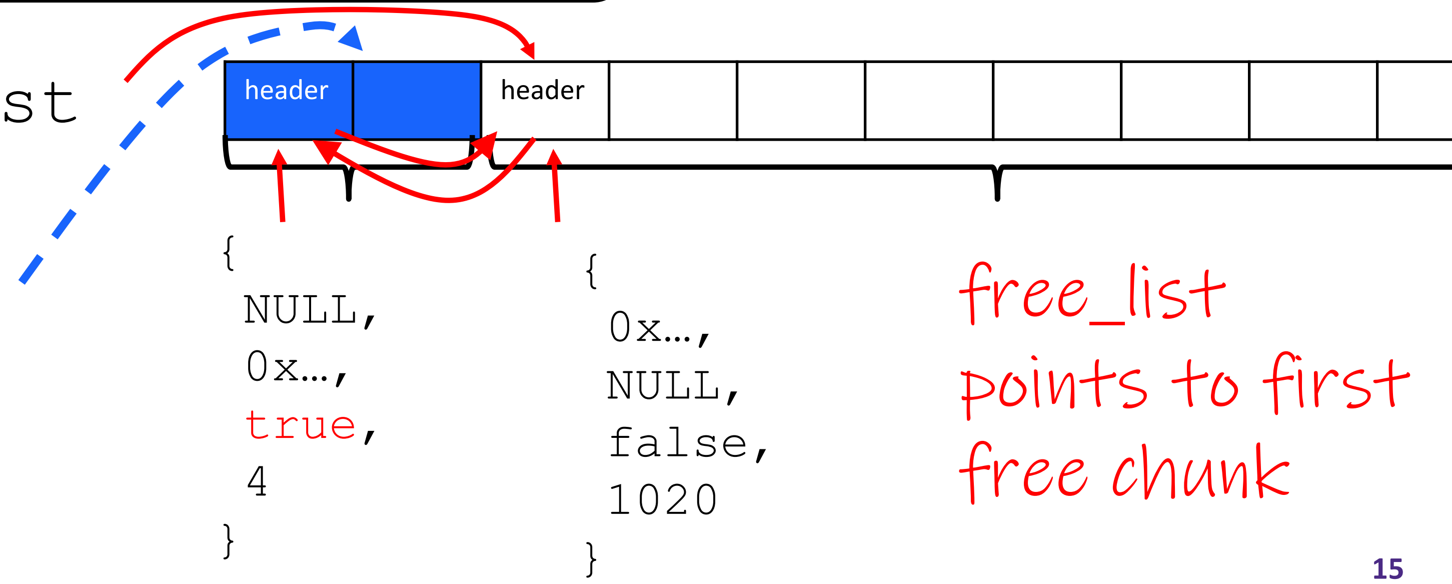
Free chunks can be split to allocate blocks of specific size

```
#include <stdlib.h>

int main() {
  char* ptr = malloc(4*sizeof(char));
  char* ptr2 = malloc(6*sizeof(int));
  ...             // do stuff with ptr
  free(ptr);
  free(ptr2);
}
```

Malloc gets a pointer to just after the metadata

- free_list



```
{
 NULL,
 0x…,
 true,
 4
}
```

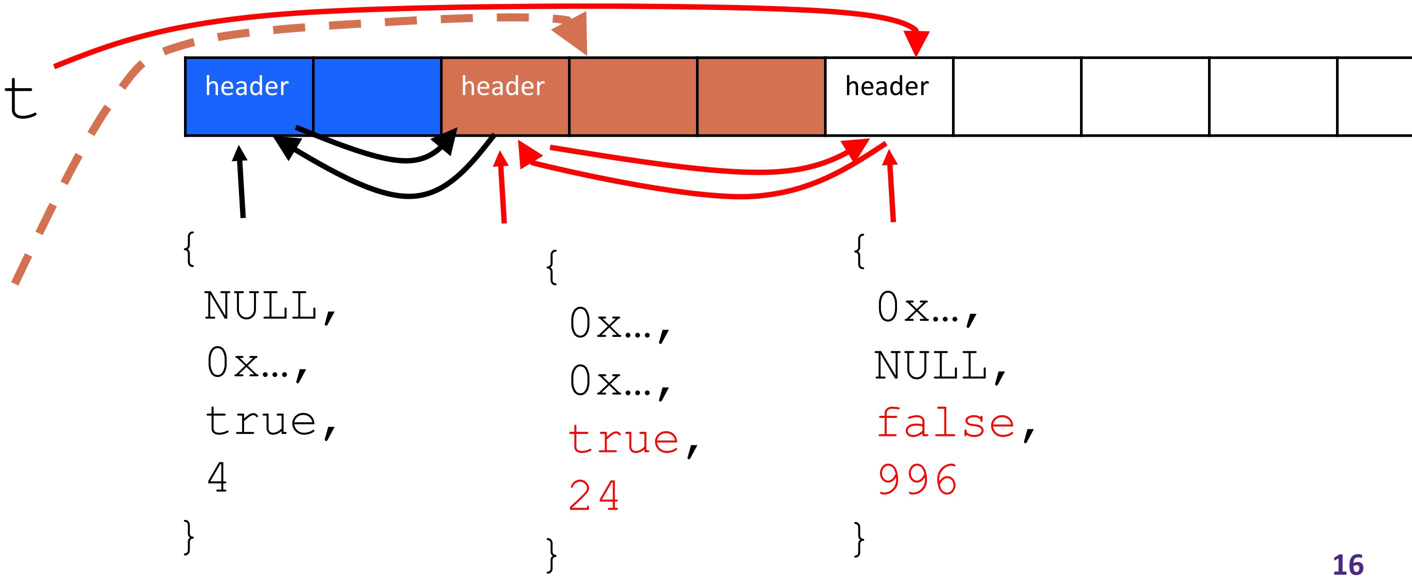
```
{
 0x…,
 NULL,
 false,
 1020
}
```

free_list points to first free chunk

Dynamic Memory Example

```
#include <stdlib.h>

int main() {
  char* ptr = malloc(4*sizeof(char));
  char* ptr2 = malloc(6*sizeof(int));
  ...               // do stuff with ptr
  free(ptr);
  free(ptr2);
}
```

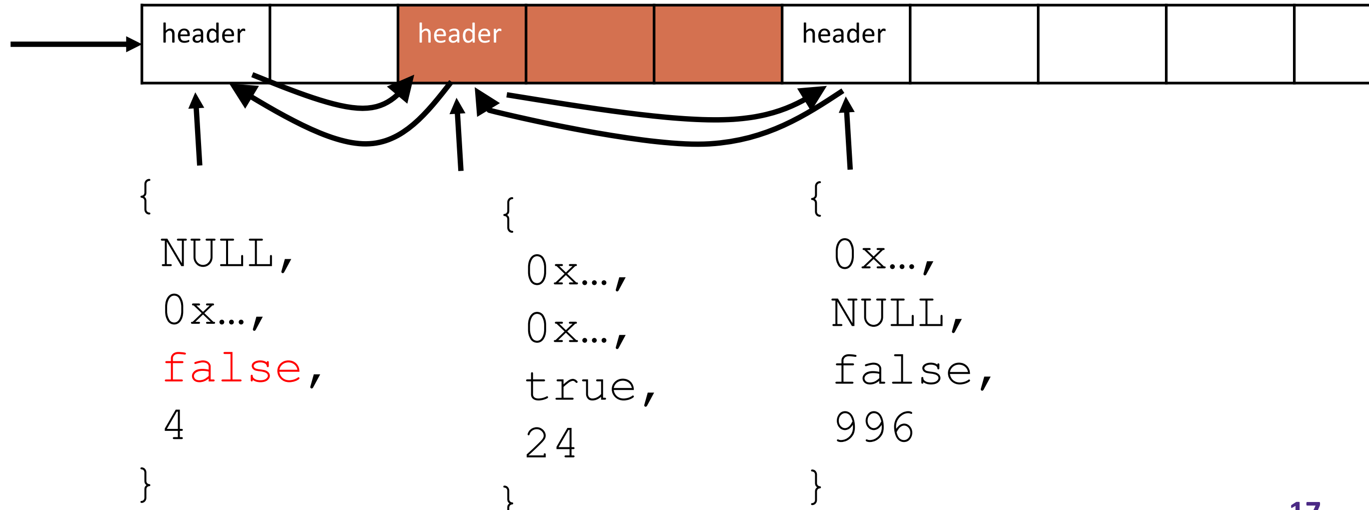


Dynamic Memory Example

```
#include <stdlib.h>

int main() {
  char* ptr = malloc(4*sizeof(char));
  char* ptr2 = malloc(6*sizeof(int));
  ...             // do stuff with ptr
  free(ptr);
  free(ptr2);
}
```

- free_list

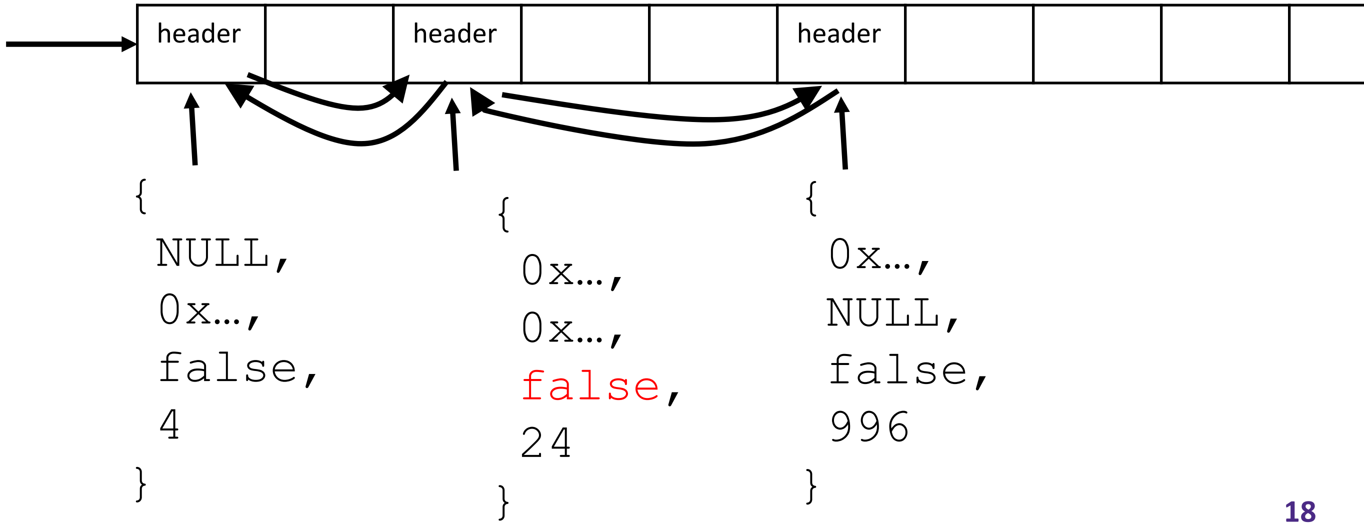


Dynamic Memory Example

```
#include <stdlib.h>

int main() {
  char* ptr = malloc(4*sizeof(char));
  char* ptr2 = malloc(6*sizeof(int));
  ...               // do stuff with ptr
  free(ptr);
  free(ptr2);
}
```

- free_list



Dynamic Memory Example

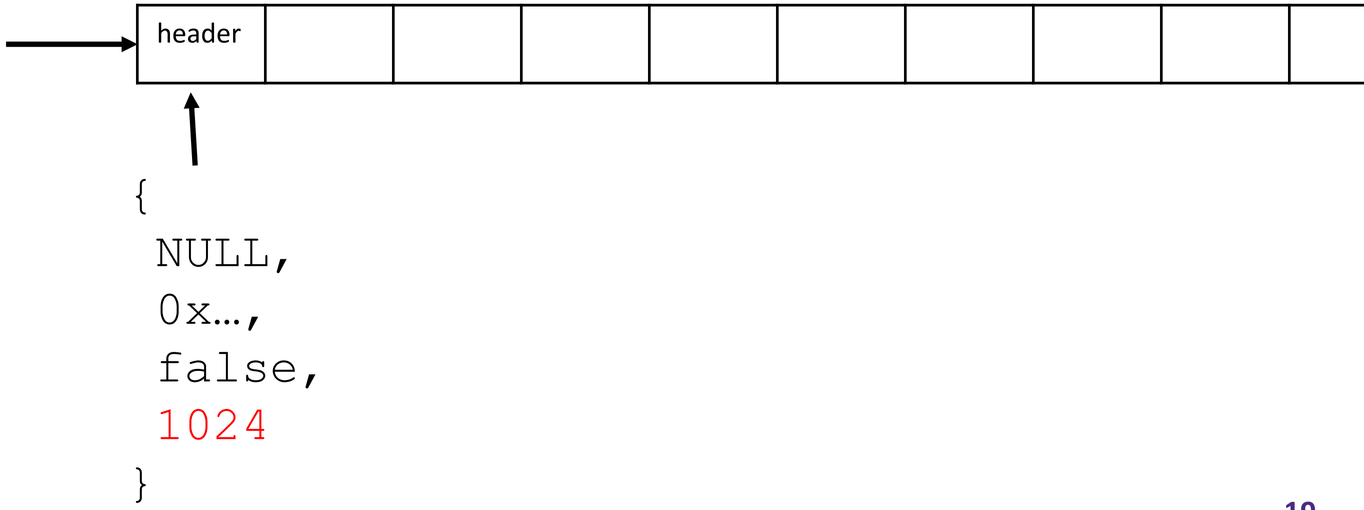
```
#include <stdlib.h>

int main() {
  char* ptr = malloc(4*sizeof(char));
  char* ptr2 = malloc(6*sizeof(int));
  ...              // do stuff with ptr
  free(ptr);
  free(ptr2);
}
```

Once a block has been freed, we can try to "coalesce" it with their neighbors

The first free couldn't be coalesced, only neighbor was allocated

- free_list



```
{
 NULL,
 0x…,
 false,
 1024
}
```

Heap

- malloc() and `free()` **are not system calls**, they are implemented as part of the C std library
  - `malloc()` and `free()` will sometimes internally invoke system calls to expand the heap if needed
  - Instead, these functions just manipulate memory already given to the process, marking some as free and some as allocated

- brk() and sbrk()
  - Used to grow/shrink the data segment of memory

- mmap(), munmap()
  - creates / or destroys a mapping in virtual address space

Lecture Outline

- Heap & Stack
- **Fragmentation & Allocation Strategies**
- Buddy Algorithm
- Slab Algorithm

Fragmentation

- Fragmentation: when storage is used inefficiently, which can hurt performance and ability to allocate things.

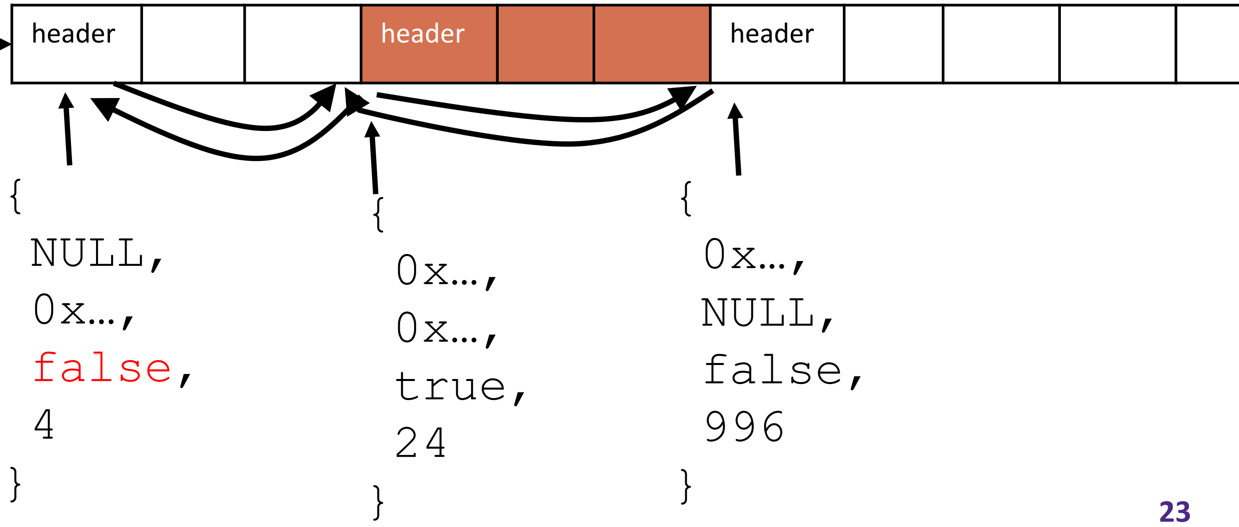
  Specifically, when there is something that prevents "unused" memory from otherwise being used

- External Fragmentation: when free memory is spread out over small portions that cannot be coalesced into a bigger block that can be used for allocation

External Fragmentation Example

```
#include <stdlib.h>

int main() {
  char* ptr = malloc(4*sizeof(char));
  char* ptr2 = malloc(6*sizeof(int));
  ...            // do stuff with ptr
  free(ptr);
  ptr = malloc(2*sizeof(char));
  ...
}
```

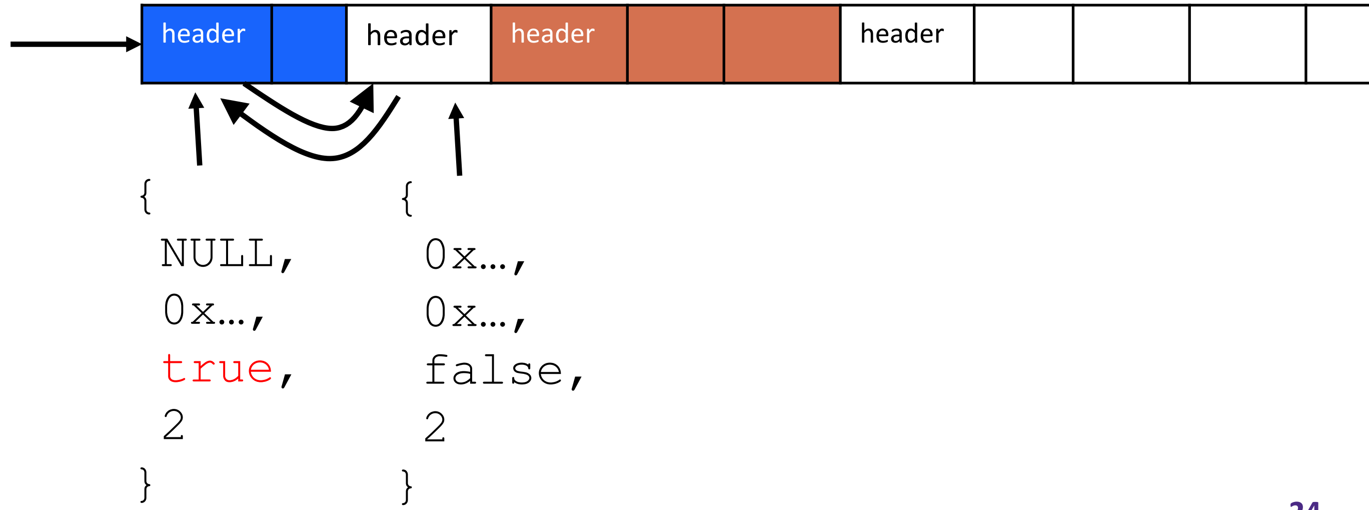


External Fragmentation Example

```c
#include <stdlib.h>

int main() {
  char* ptr = malloc(4*sizeof(char));
  char* ptr2 = malloc(6*sizeof(int));
  ...            // do stuff with ptr
  free(ptr);
  ptr = malloc(2*sizeof(char));
  ...
}
```

- free_list



```
{              {
 NULL,          0x…,
 0x…,           0x…,
 true,          false,
 2              2
}              }
```

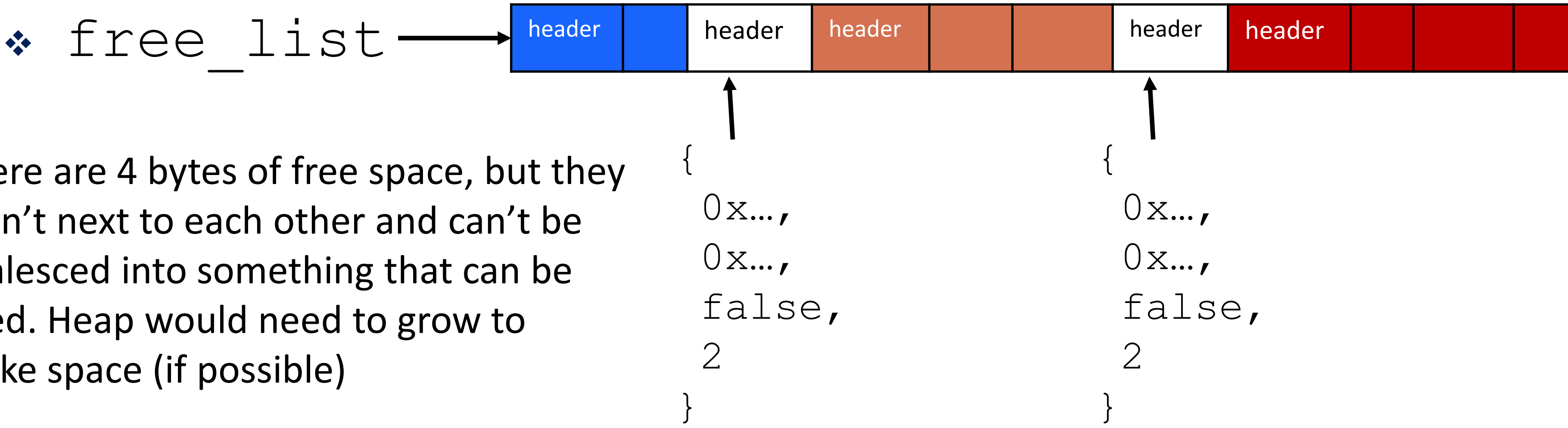
External Fragmentation Example

```
#include <stdlib.h>

int main() {
  char* ptr = malloc(4*sizeof(char));
  char* ptr2 = malloc(6*sizeof(int));
  ...              // do stuff with ptr
  free(ptr);
  ptr = malloc(2*sizeof(char));
  ...
}
```

After some more series of allocations and frees (not shown), we get this:

Let's say **`malloc(4)`** gets called (trying to allocate 4 bytes) what happens?



There are 4 bytes of free space, but they aren't next to each other and can't be coalesced into something that can be used. Heap would need to grow to make space (if possible)

Internal Fragmentation

- Internal Fragmentation: When more space is allocated for something than is actually used. This fragmentation happens “internally” within an allocated portion, instead of “external” to one.

- What if someone calls **malloc(4096 * sizeof(char*))** and only uses the first `char*`?
  - Can be thought of internal fragmentation, not the allocator's fault though (in this use case)
- Sometimes we call **malloc()** and more space is allocated than needed.
  - if we allocate for 7 bytes, 8 may actually be allocated. Computer may want addresses to be aligned to a multiple of a power of 2

First Fit

- There may be multiple free blocks that can be chosen for allocation.

- The allocation policy we used in our examples is **First Fit**: find the first block of memory that is big enough
  - Start at the front of the free list, iterate till we find something big enough
  - Usually the simplest to implement

Best Fit

- **Best Fit**: another approach where instead you look for the portion of memory that is the “best” or “tightest” fit

- If allocating for 4 bytes of memory, search for the smallest block that is >= 4 bytes.

Worst Fit

- **Worst Fit**: another approach where instead you look for the portion of memory that is the "worst" fit (opposite of best fit)

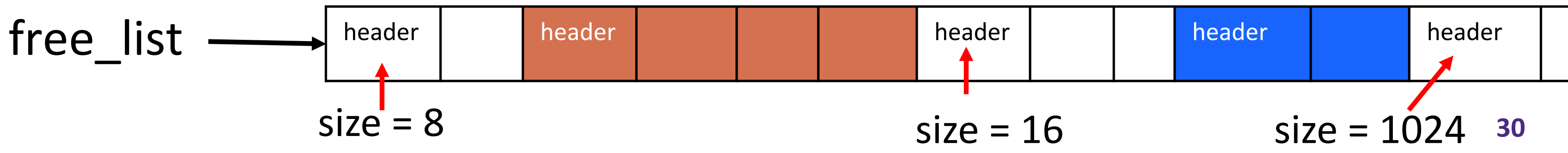
- If allocating for 4 bytes of memory, search for the **largest** block that is >= 4 bytes.

Poll Everywhere

pollev.com/tqm

- What is the approximate runtime of the algorithms? (e.g. O(N log(N))). What is the best/worst case?
  - First Fit
  - Best Fit
  - Worst Fit

- Lets say we call `malloc(4 bytes)`. Which block is allocated in this example if we choose:
  - First Fit
  - Best fit
  - Worst fit



Poll Everywhere

pollev.com/tqm

- It turns out that over long periods of time, worst fit can work better than best fit. Why is this the case?

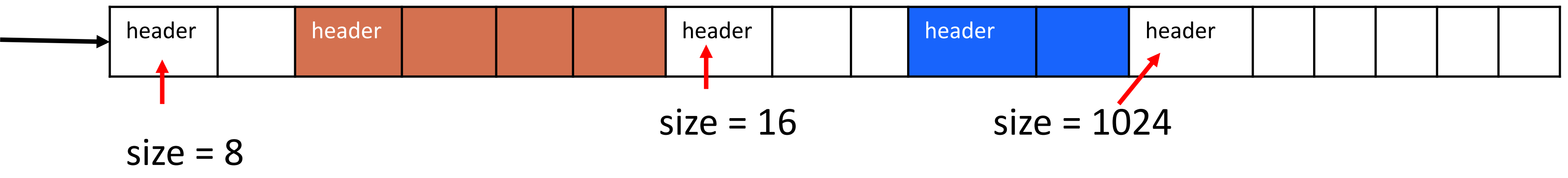
Poll Everywhere

pollev.com/tqm

- It turns out that over long periods of time, worst fit can work better than best fit. Why is this the case?

- Less small “leftover” fragments, fragments are bigger and easier to reuse

- In the previous example, if we allocate for size 6…



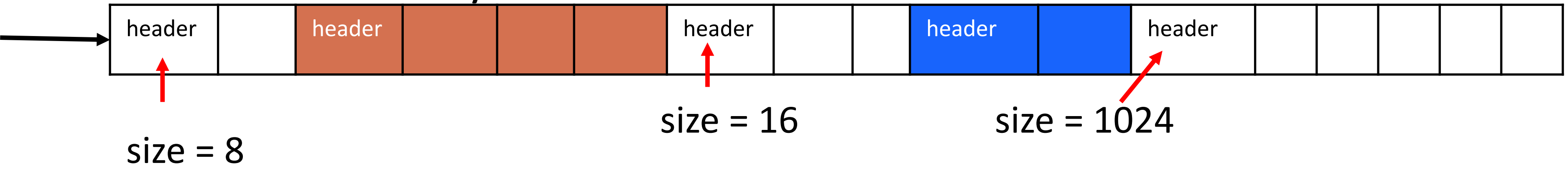
Poll Everywhere

pollev.com/tqm

- It turns out that over long periods of time, worst fit can work better than best fit. Why is this the case?

- Less small "leftover" fragments, fragments are bigger and easier to reuse

- In the previous example, if we allocate for size 6…
  - Best fit would allocate the size 8 free chunk leaving a size 2 chunk that is unlikely to be usable



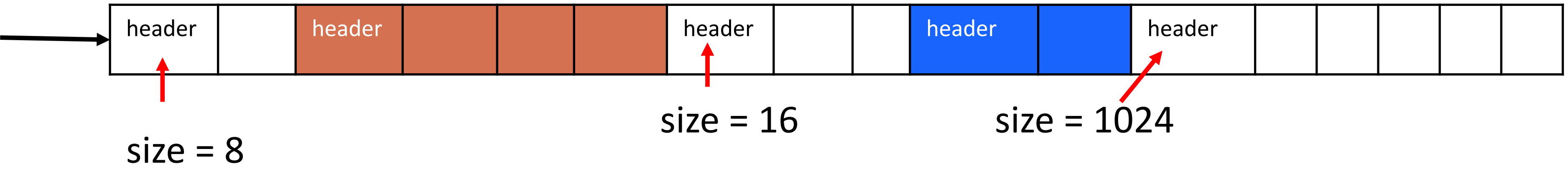
Poll Everywhere

pollev.com/tqm

- It turns out that over long periods of time, worst fit can work better than best fit. Why is this the case?

- Less small "leftover" fragments, fragments are bigger and easier to reuse

- In the previous example, if we allocate for size 6…
  - Worst fit would use 1024, splitting it into 6 and 1018. 8 chunk is still usable and 1018 is still usable.



Lecture Outline

- Heap & Stack
- Fragmentation & Allocation Strategies
- **Buddy Algorithm**
- Slab Algorithm

Buddy Algorithm

- Keeps in mind that there is some “maximum” amount of memory and divides memory into partitions that are powers of 2.
  - Power of 2 allows for compact allocation tracking and makes coalescing memory quick.
  - Usually with the smallest unit being 1 page, 4096 bytes.

- Modified implementation of the buddy system is one of many things used by the Linux kernel and the others (like a version of malloc called `jemalloc`)

Buddy Algorithm walkthrough

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2^4 pages															

- We start with the full pool of memory, in this example, 2^4 pages (usually a higher cap than this, this is for example)

- What happens if someone asks to allocate 1 page?

Buddy Algorithm walkthrough

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2^3 pages								**2^3 pages**							

- What happens if someone asks to allocate 1 page?
  - Split page chunks into half until we have enough

Buddy Algorithm walkthrough

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2^2 pages				**2^2 pages**				2^3 pages							

- What happens if someone asks to allocate 1 page?
  - Split page chunks into half until we have enough

Buddy Algorithm walkthrough

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2^1 pages		**2^1 pages**		2^2 pages				2^3 pages							

- What happens if someone asks to allocate 1 page?
  - Split page chunks into half until we have enough

Buddy Algorithm walkthrough

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	**1**	2^1 pages		2^2 pages				2^3 pages							

- What happens if someone asks to allocate 1 page?
  - Split page chunks into half until we have enough

Buddy Algorithm walkthrough

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2^1 pages		2^2 pages				2^3 pages							
A															

- What happens if someone asks to allocate 1 page?
  - Split page chunks into half until we have enough

- Can mark the one page as being used by allocation **A**

Buddy Algorithm walkthrough

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2^1 pages		2^2 pages				2^3 pages							
A															

- Now someone requests 2 pages, what happens?

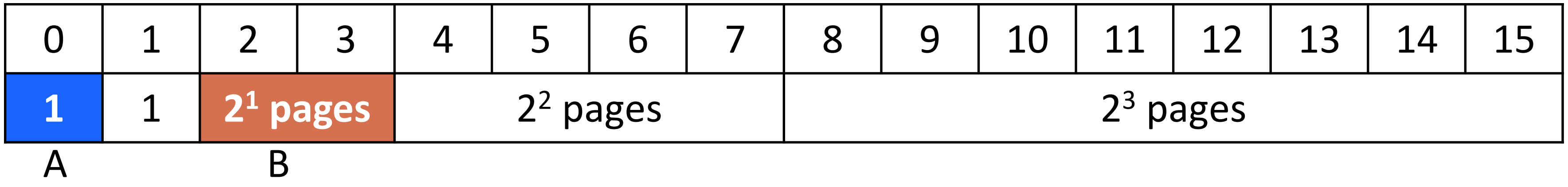
Buddy Algorithm walkthrough

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	**2^1 pages**		2^2 pages				2^3 pages							
A		B													

- Now someone requests 2 pages, what happens?

- We can claim the 2^1-page chunk and mark it as being used by allocation **B**

Buddy Algorithm walkthrough



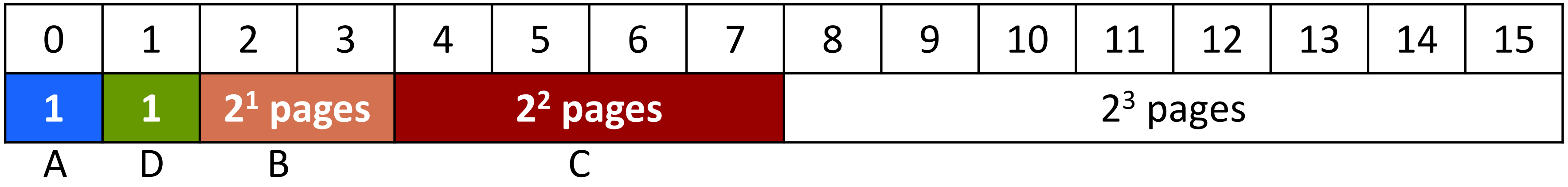
- Now someone requests 3 pages, what happens?

Buddy Algorithm walkthrough

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	**2^1 pages**		**2^2 pages**				2^3 pages							
A		B		C											

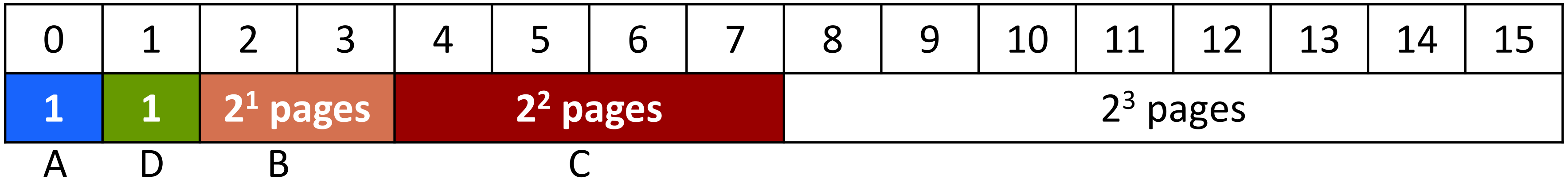
- Now someone requests 3 pages, what happens?
- Buddy ONLY deals with powers of 2, this gets rounded up to 2^2 pages (4 pages)
- We can claim the 2^2-page chunk and mark it as being used by allocation **C**

Buddy Algorithm walkthrough



- Last allocation: someone allocates 1 page, what happens?
- We can claim the 1-page chunk and mark it as being used by allocation **D**

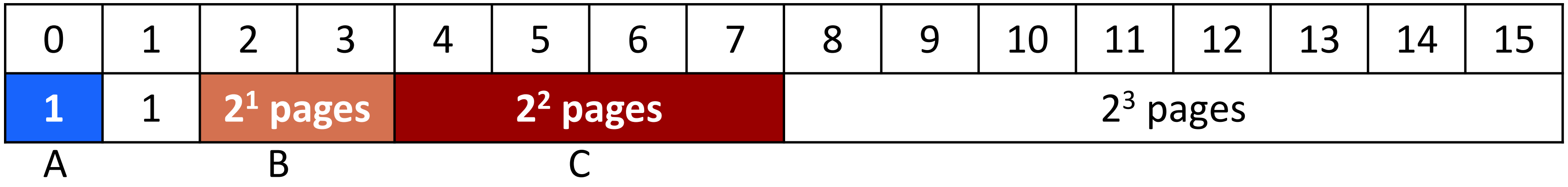
Buddy Algorithm walkthrough



- Let’s walk through the freeing process

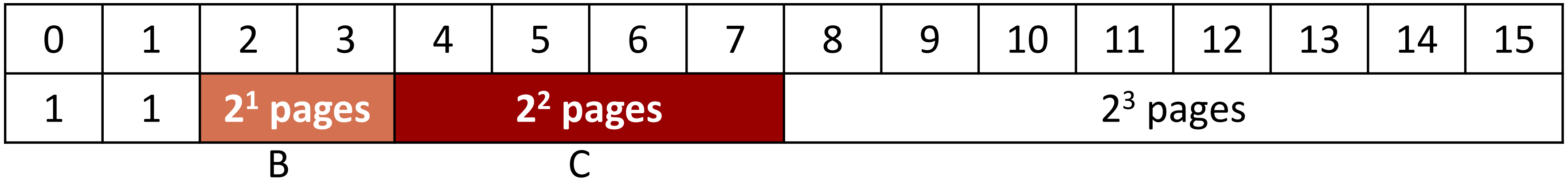
- First, allocation D is done and frees its page

Buddy Algorithm walkthrough



- Let’s walk through the freeing process
- First, allocation D is done and frees its page
- To free the page, we just mark it as no longer being allocated. Nothing we can coalesce (yet)

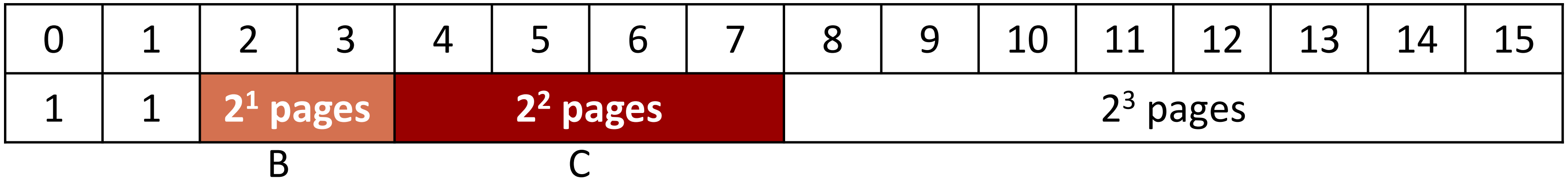
Buddy Algorithm walkthrough



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	**2^1 pages**		**2^2 pages**				2^3 pages							
		B		C											

- Let’s walk through the freeing process
- Second, allocation A is done and frees its page
- To start, we just mark it as no longer being allocated.

Buddy Algorithm walkthrough

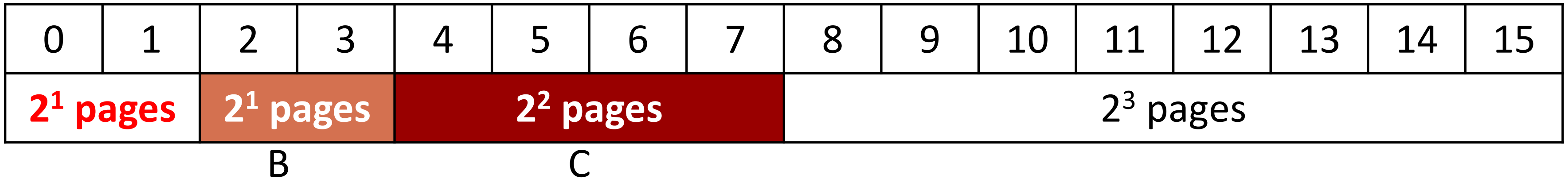


- Let’s walk through the freeing process

- Second, allocation A is done and frees its page

- To start, we just mark it as no longer being allocated.
- Then we can coalesce!
- Each “chunk” has a “buddy”, the buddy being the its “twin” created while spitting chunks in half.
- If both buddies are free, they can be combined ☺

Buddy Algorithm walkthrough

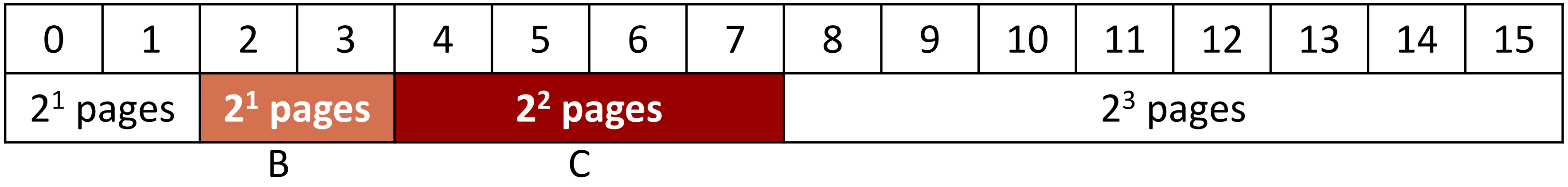


- Let's walk through the freeing process

- Second, allocation A is done and frees its page

- To start, we just mark it as no longer being allocated.
- Then we can coalesce!
- Each "chunk" has a "buddy", the buddy being the its "twin" created while spitting chunks in half.
- If both buddies are free, they can be combined ☺

Buddy Algorithm walkthrough



- Let’s walk through the freeing process
- Third, allocation C is done and frees its pages

Buddy Algorithm walkthrough

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2^1 pages		**2^1 pages**		2^2 pages				2^3 pages							
		B													

- Let’s walk through the freeing process
- Third, allocation C is done and frees its pages
- Can’t coalesce since its buddy is not completely free

Buddy Algorithm walkthrough

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2^1 pages		2^1 pages		2^2 pages				2^3 pages							

- Let’s walk through the freeing process
- lastly, allocation B is done and frees its pages

Buddy Algorithm walkthrough

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2^2 pages				2^2 pages				2^3 pages							

- Let’s walk through the freeing process
- lastly, allocation B is done and frees its pages
- Its buddy is free so we can coalesce!

Buddy Algorithm walkthrough

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2^3 pages								2^3 pages							

- Let's walk through the freeing process
- lastly, allocation B is done and frees its pages
- Its buddy is free so we can coalesce!
- The newly coalesced chunk can be further coalesced!

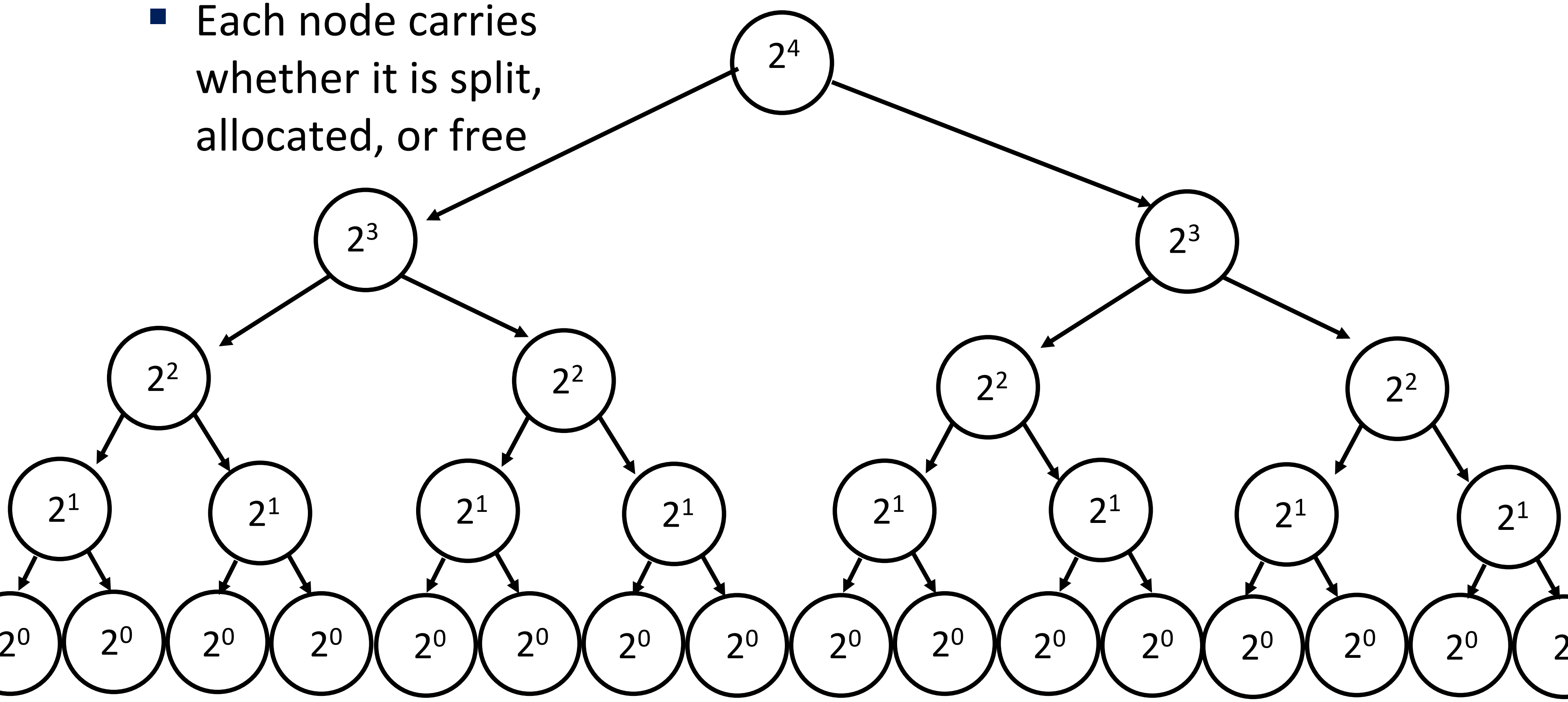
Buddy Algorithm walkthrough

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2^4 pages															

- Let’s walk through the freeing process
- lastly, allocation B is done and frees its pages
- Its buddy is free so we can coalesce!
- The newly coalesced chunk can be further coalesced!
- The newly coalesced chunk can be further coalesced!

Buddy Algorithm Implementation

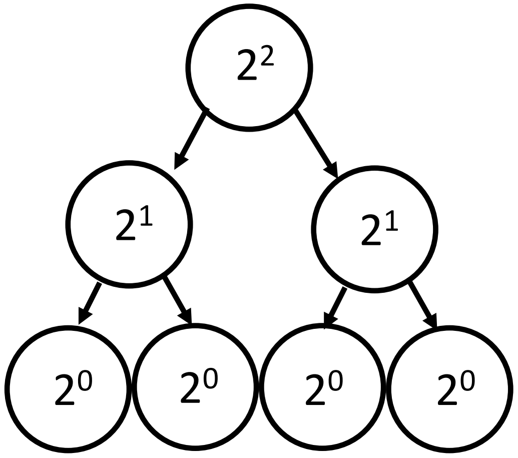
- Buddy Algorithm can be maintained with a binary search tree
  - Each node carries whether it is split, allocated, or free



Buddy Algorithm Implementation

- Since Buddy has a known max size, we can represent the tree in an array or bitmap. (example shows up to 2^2 for space)

2^2	2^1	2^1	2^0	2^0	2^0	2^0

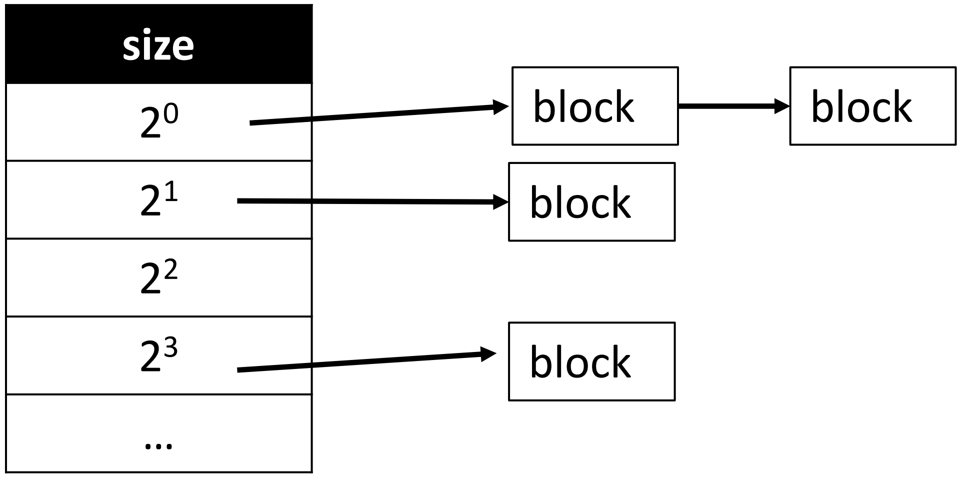


2^2			
2^1		2^1	
2^0	2^0	2^0	2^0

(alternate way to show the array, may make the connection between array and tree easier to see).
Indexes go Left -> Right, top to bottom

Buddy Algorithm Implementation

- The tree (array representation) is useful for coalescing, but we can make algorithm faster by keeping track of several free lists, roughly one list per size
  - Quicker lookup for memory allocation



Poll Everywhere

pollev.com/tqm

- How does the fragmentation for the buddy algorithm look?

Buddy Algorithm

- A bit restrictive in the interface, must be a power of 2
  - Internal fragmentation can be a lot ☹
  - If someone needs 2^4 +1 pages, buddy algorithm will allocate 2^5 pages, 2^4 - 1 pages of fragmentation
- External fragmentation is generally kept pretty small
- Small allocations don’t really work for this

Lecture Outline

- Heap & Stack
- Fragmentation & Allocation Strategies
- Buddy Algorithm
- **Slab Algorithm**

Slab Allocator

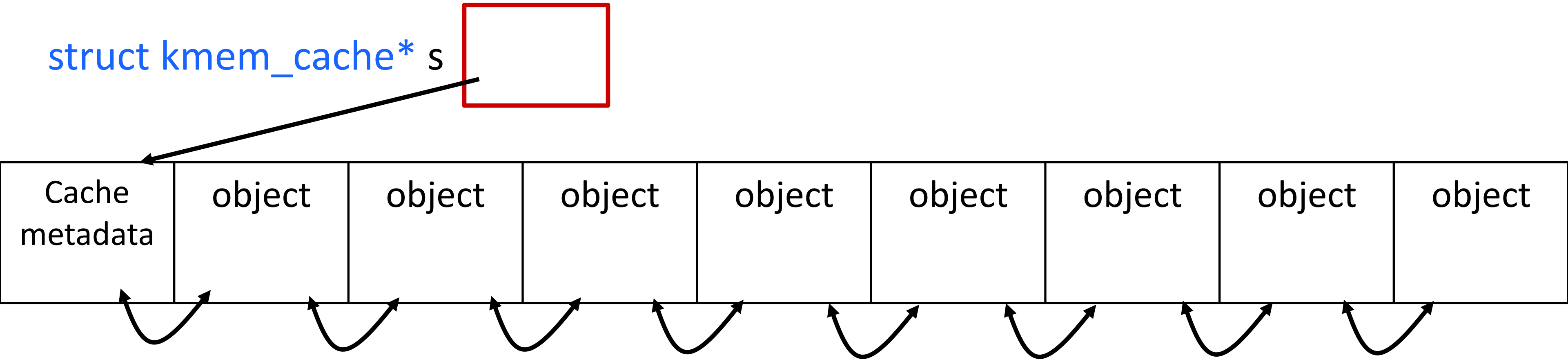
- What if we restrict the API to a ***single*** size that can be allocated or freed?

- First, you need to allocate the thing you will allocate from
  - When you create it, you specify a name and some other information
  - **The thing we care about is that you specify the size of the objects that the slab allocator will allocate from**

```
// Internal to the OS, you can't call it yourself
void * kmem_cache_alloc ( const char* name, unsigned int size,
                          unsigned int align, unsigned int flags);
```

Slab Allocator High Level

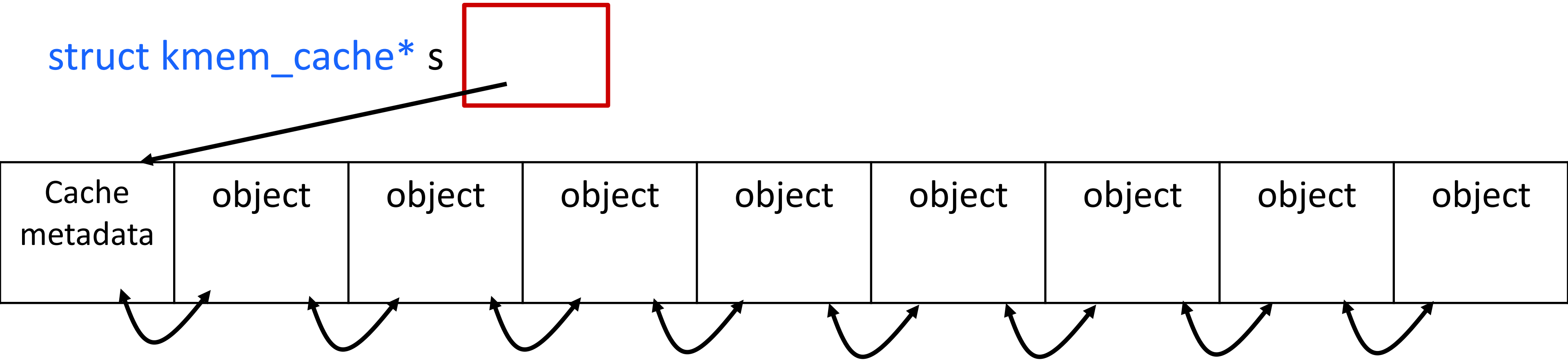
- In the context of a slab allocator
  - Object: the thing we want to allocate, some fixed size memory that we want to allocate
    NOT the same as a java object
  - Cache: a chunk of memory containing the "objects"



  - Objects are in contiguous in memory, but still has links to keep track of which objects are free

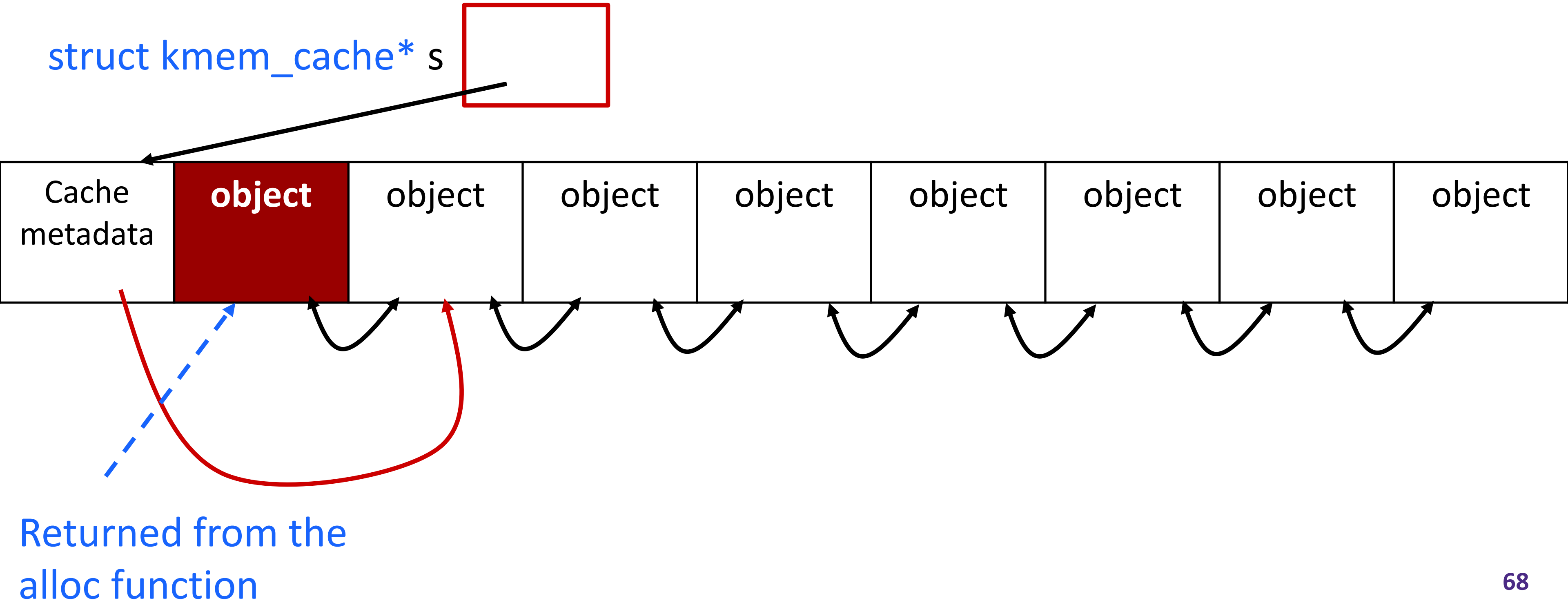
Slab Allocator High Level: Alloc

- When we allocate from the cache, we get a pointer to the first element that is free



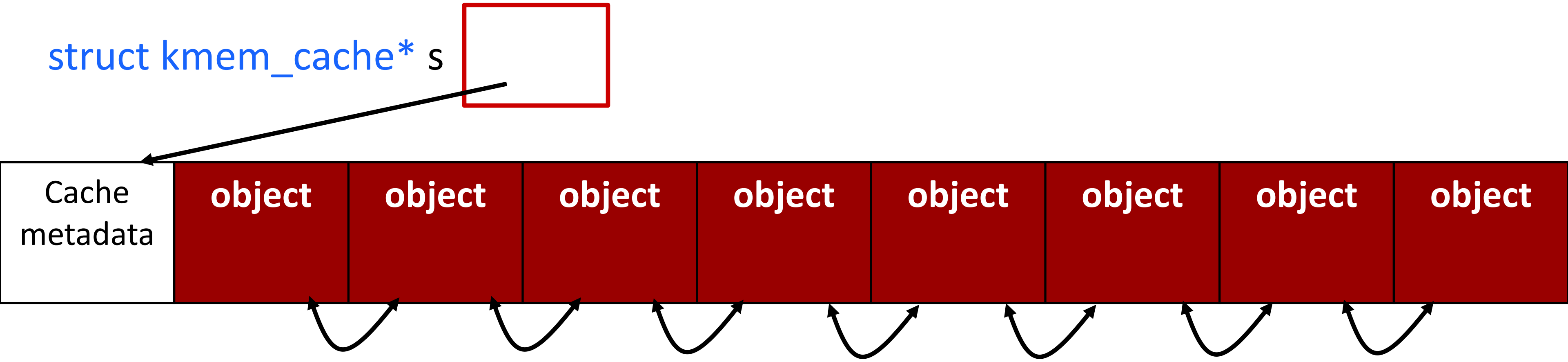
Slab Allocator High Level: Alloc

- When we allocate from the cache, we get a pointer to the first element that is free
  - Update the pointer to the next element once we alloc



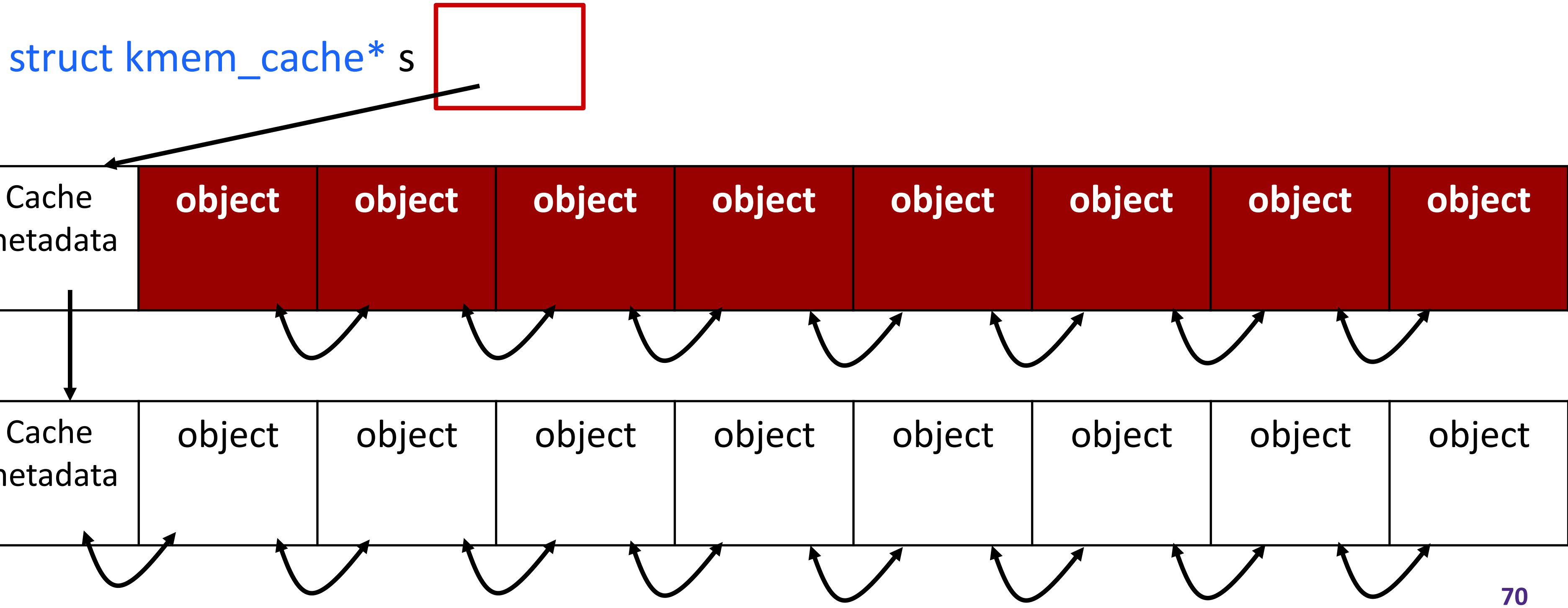
Slab Allocator High Level: Alloc

- What happens when we run out of objects?



Slab Allocator High Level: Alloc

- What happens when we run out of objects?
  - Allocate a new slab for the cache (allocate from buddy)
- Each contagious chunk of memory is one “slab”, with the slab usually being a size appropriate to ask from buddy



Discuss

- What is the runtime for slab?
- How does the fragmentation look?

Slab Allocator Analysis

- Slab allocator is very useful for minimizing overhead for allocating and freeing. A constant time algorithm

- Can be minimal internal and external fragmentation (gets more complicated when you account for alignment and buddy algo requirements)

Slab Allocator Usage

- Used on top of the buddy algorithm in the kernel.
  - This allows us to use the buddy algorithm still, but can quickly allocate smaller sized "objects" within the *slabs* of memory returned by the buddy algorithm

- General Memory allocators may use something like this, allocate many slabs of various sizes and try to mostly use those for allocation