Memory Allocation (fin)

Computer Operating Systems, Summer 2025

Instructors: Joel Ramirez Travis McGaha

TAs: Ash Fujiyama Sid Sannapareddy Maya Huizar

Poll Everywhere

pollev.com/tqm

- How is PennOS Going? Any Questions about Memory Allocation?

Administrivia

- Final Exam; July 31st
  - Same format as the Midterm!
  - Last course day is July 23rd
  - Final Exam Review Thursday, July 24th
- PennOS Due Friday, July 25th
- Final Grades due August 11th
  - So even though the course is “over”, there’s still wiggle room at the end.

Administrivia

- Some notes:
  - DO NOT mmap the entire File System. Only mmap the Allocation Table, the rest of the file system needs to be handled with lseek/write.
    - Do not keep the contents of the file in memory, it should be stored in the file
    - If your PennFat is killed with kill -9, your file contents should still be saved in disk
  - Advice for using gdb to debug
    - **`handle SIGUSR1 noprint nostop`**
      Makes it so that gdb doesn't report every time SIGUSR1 goes and interrupts you
  - (more on next slide)

Administrivia

- Some notes:
  - Reminder, you instead of just doing:

```
lseek(FAT_FD, offset, SEEK_SET);
write(FAT_FD, contents, size);
```

    you may need to do:

```
lseek(FAT_FD, offset, SEEK_SET);
write(FAT_FD, contents, size);
lseek(FAT_FD, offset, SEEK_SET);
write(FAT_FD, contents, size);
```

  - With the description of `setitimer()`, it just says that sigalarm is delivered to the process, not necessarily the calling thread. To make sure siglaram goes to the scheduler, you may want to make it so that all threads (spthread or otherwise) that aren't the scheduler call something like: **`pthread_sigmask(SIG_BLOCK, SIGALARM)`**
    - Which will block SIGALARM in that thread.

Administrivia

- If you are having issues with the scheduler not running you can try running
  - `strace -e 'trace=!all' ./bin/pennos`
  - You may have to install strace: `sudo apt install strace`
  - This will print out every time a signal is sent to your pennos
  - (Usual fix is the pthread_sigmask thing on the previous slide)

Lecture Outline

- **Garbage collection**
- Arena Allocation
- Buddy Algorithm
- Slab/Slub Allocator

Memory Leaks

- The most common Memory Pitfall
- What happens if we allocate something, but don't delete it?
  - That block of memory cannot be reallocated, even if we don't use it anymore, until it is **`delete`**`-d`

- Garbage Collection
  - Automatically "frees" anything once the program has lost all references to it
  - Affects performance, but avoid memory leaks
  - Java and other "high level" languages

- RAII (Resource Acquisition Is Initialization)
  - C++ and Rust have this, it is VERY GOOD

Garbage Collection

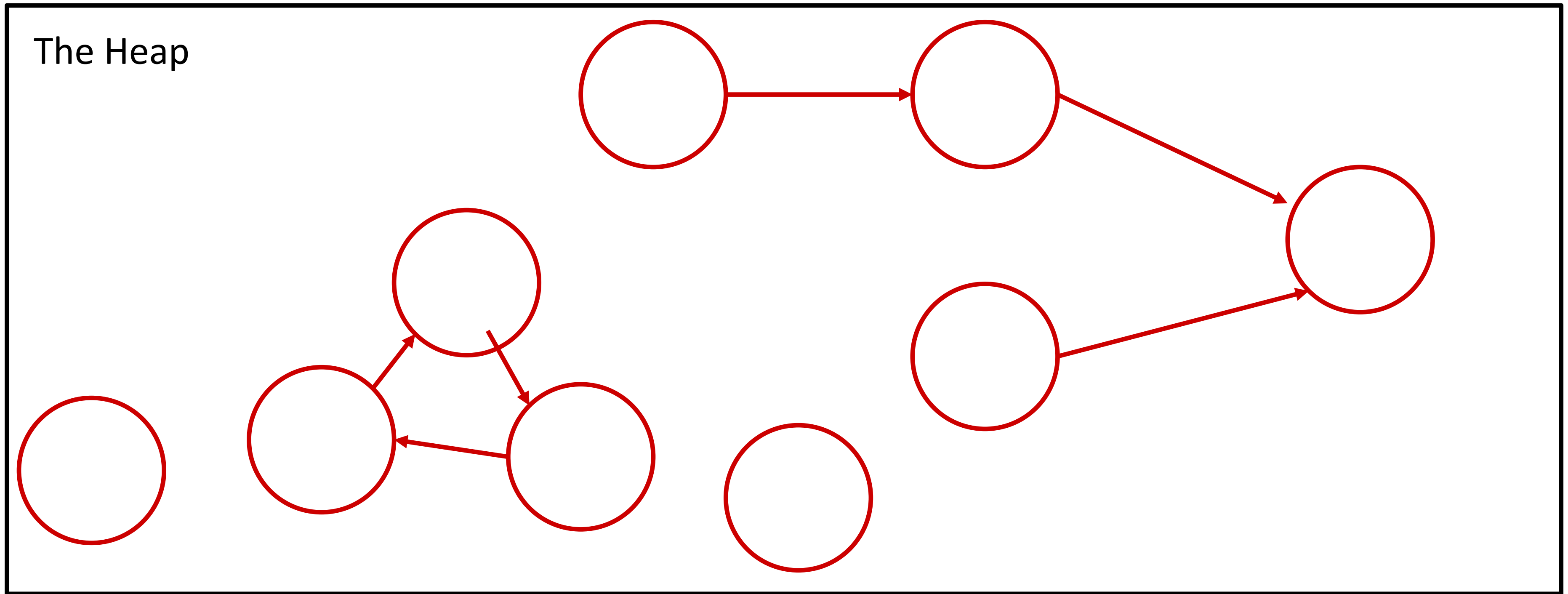
- When memory is automatically deallocated for us, so we do not need to explicitly free memory

- Very common in higher level languages:
  - Java, C#, Python, Javascript, Ruby, Lisp, Erlang, Racket, Haskell, Scala, Dart, etc.

- Big difference between these languages and languages like C / C++ / Rust:
  - Many of these languages are not run directly on your hardware.
  - Java (for example) runs on the JVM (Java Virtual Machine) which then runs on your computer
  - Garbage collection requires some help from the “runtime” environment” and/or the compiler to keep track of pointers, memory allocations etc and decide when to free them

Garbage Collection

- With the aid of the runtime and compiler we can keep track of all memory allocation and represent it as a directed graph
  - Each allocation is a node in the graph
  - Each pointer is an edge in the graph
  - If an object contains a pointer to another object we draw an edge from that node to the other.

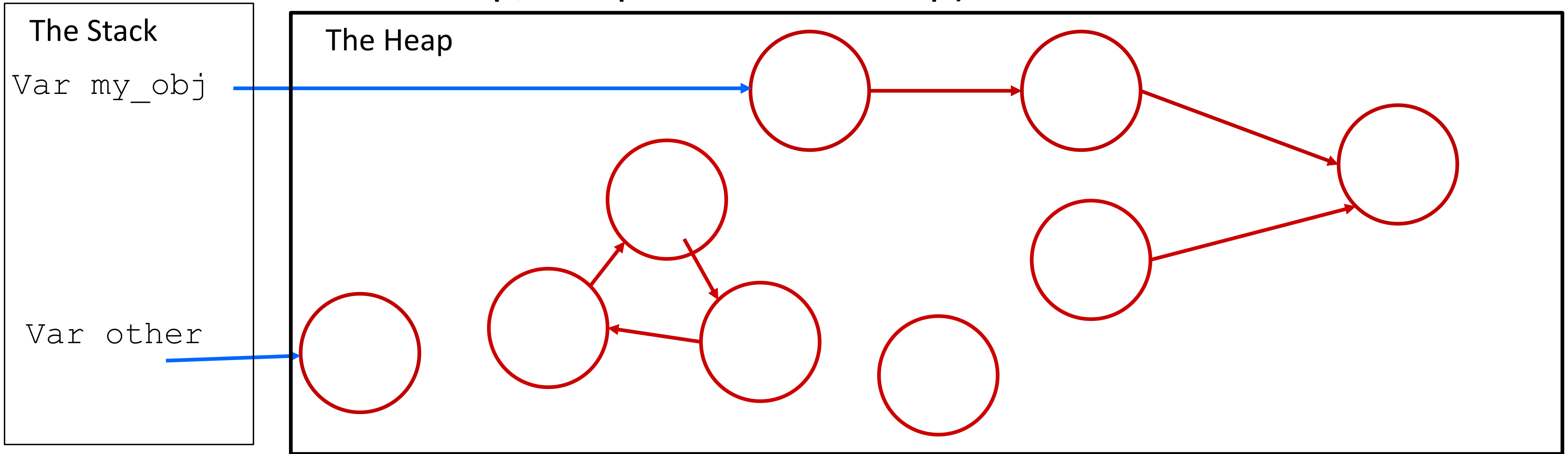


Garbage Collection

Nodes that are “reachable” from a root are safe
if it can’t be reached from a root, then it is garbage

- Each allocation is a node in the graph
- Each pointer is an edge in the graph
- If an object contains a pointer to another object we draw an edge from that node to the other.

❖ We also keep track of which pointers are held by local variables (pointers that are not on the heap, but point to the heap). These are “roots”

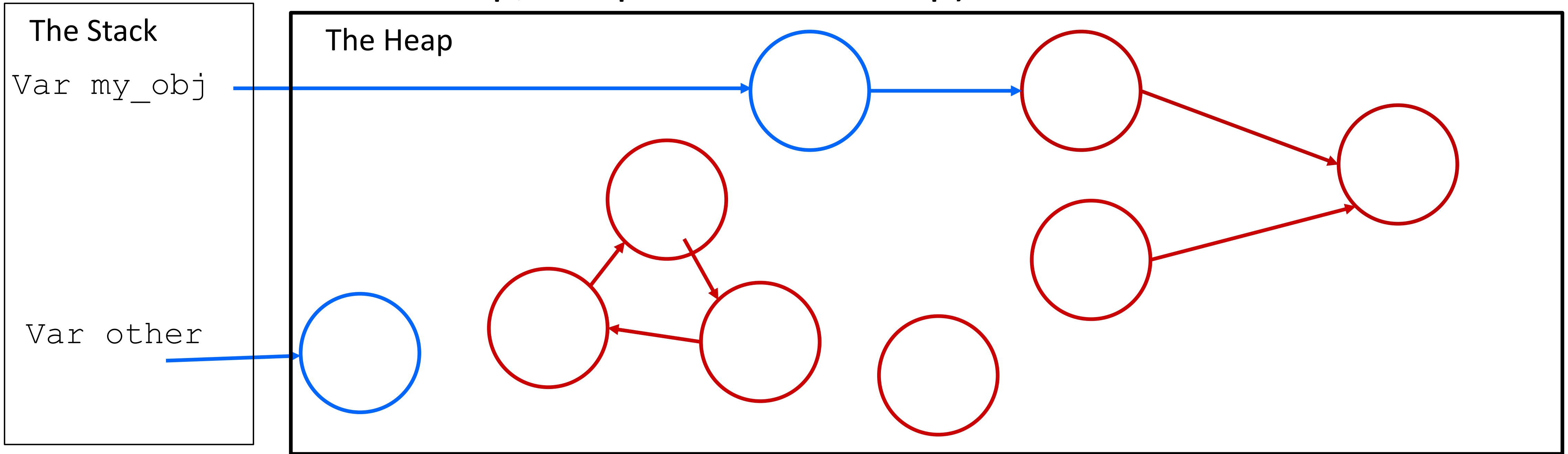


Garbage Collection

Nodes that are “reachable” from a root are safe
if it can’t be reached from a root, then it is garbage

- Each allocation is a node in the graph
- Each pointer is an edge in the graph
- If an object contains a pointer to another object we draw an edge from that node to the other.

❖ We also keep track of which pointers are held by local variables (pointers that are not on the heap, but point to the heap). These are “roots”

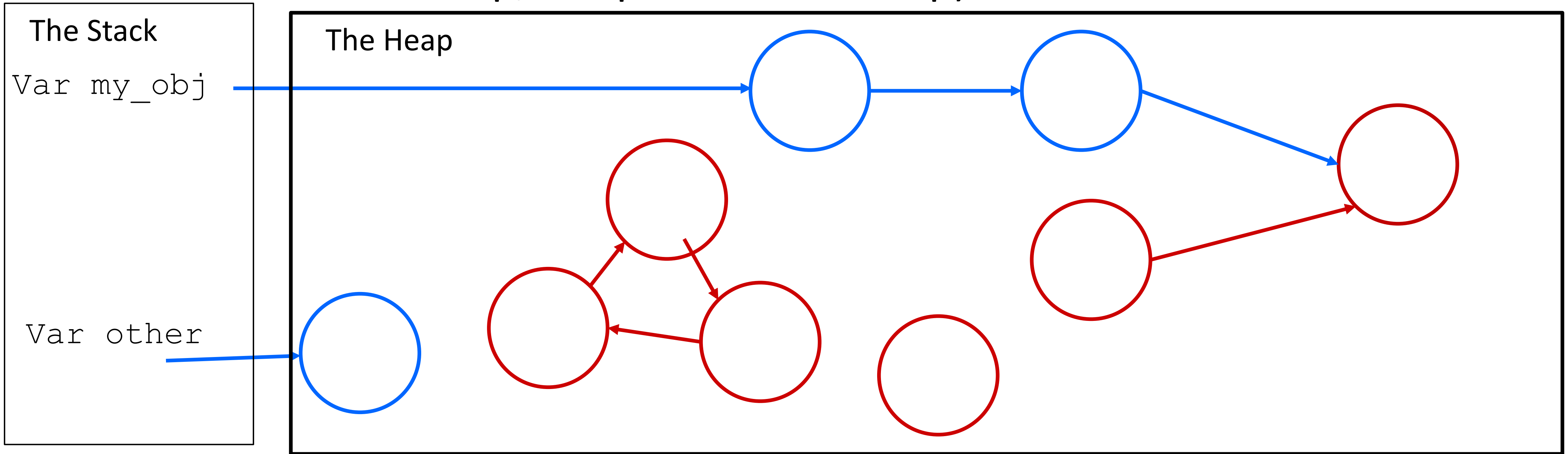


Garbage Collection

Nodes that are "reachable" from a root are safe
if it can't be reached from a root, then it is garbage

- Each allocation is a node in the graph
- Each pointer is an edge in the graph
- If an object contains a pointer to another object we draw an edge from that node to the other.

❖ We also keep track of which pointers are held by local variables (pointers that are not on the heap, but point to the heap). These are "roots"

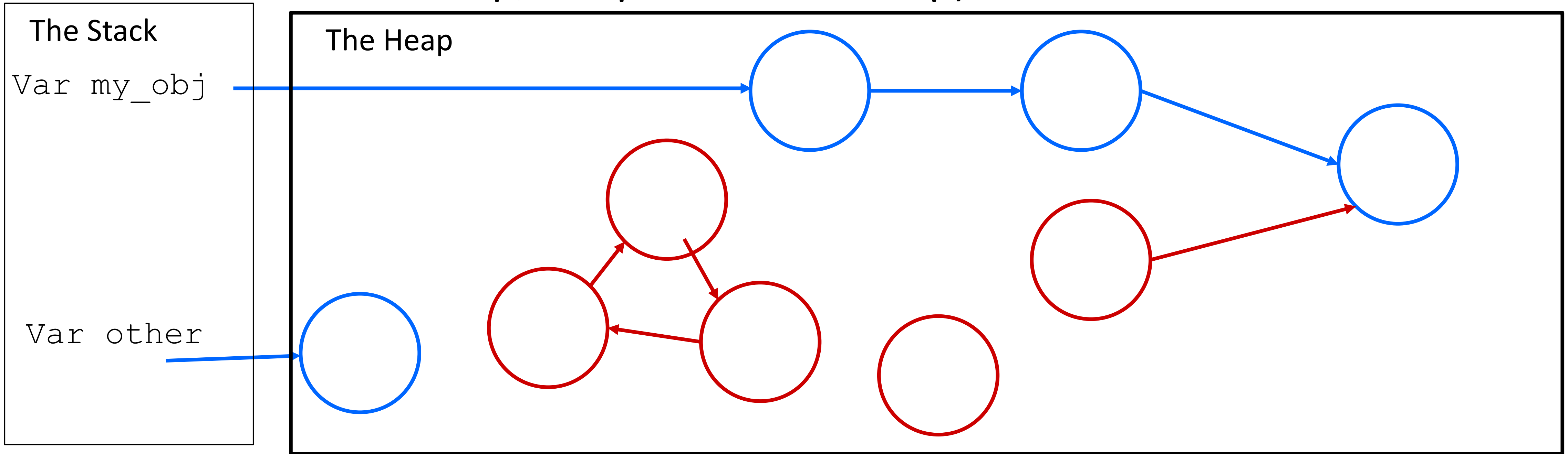


Garbage Collection

Nodes that are “reachable” from a root are safe
if it can’t be reached from a root, then it is garbage

- Each allocation is a node in the graph
- Each pointer is an edge in the graph
- If an object contains a pointer to another object we draw an edge from that node to the other.

❖ We also keep track of which pointers are held by local variables (pointers that are not on the heap, but point to the heap). These are “roots”



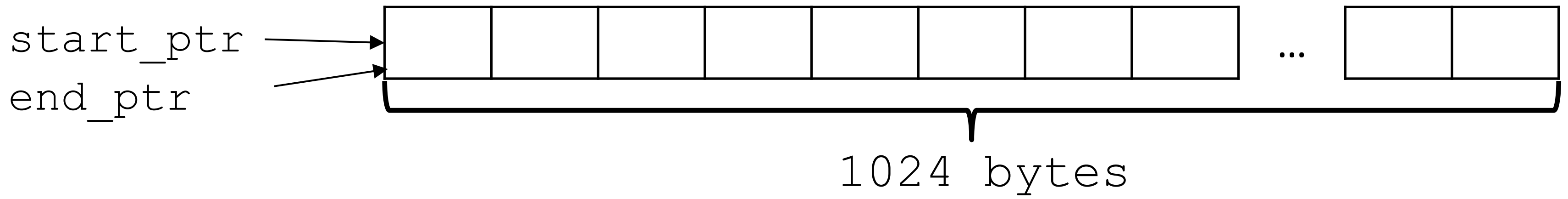
Lecture Outline

- Garbage collection
- **Arena Allocation**
- Buddy Algorithm
- Slab/Slub Allocator

Arena Allocator

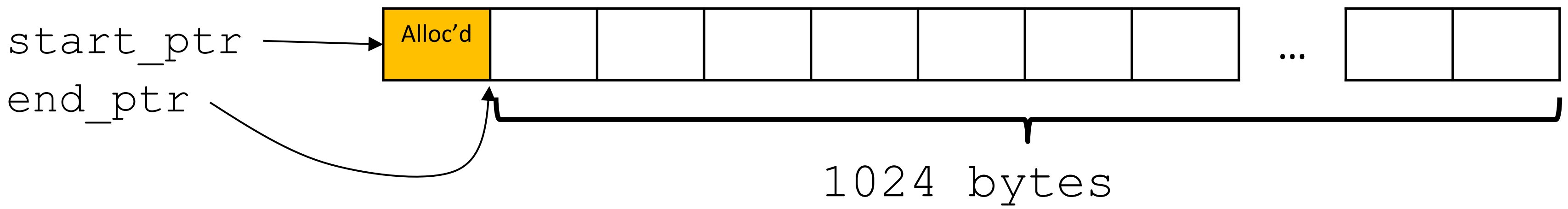
- In some instances, we want to allocate a lot of items and limit those allocations to one scope. We call our allocator a “arena allocator”. It allocates things within the same "arena"/region/pool of the same scope

- For example, Consider we start with:



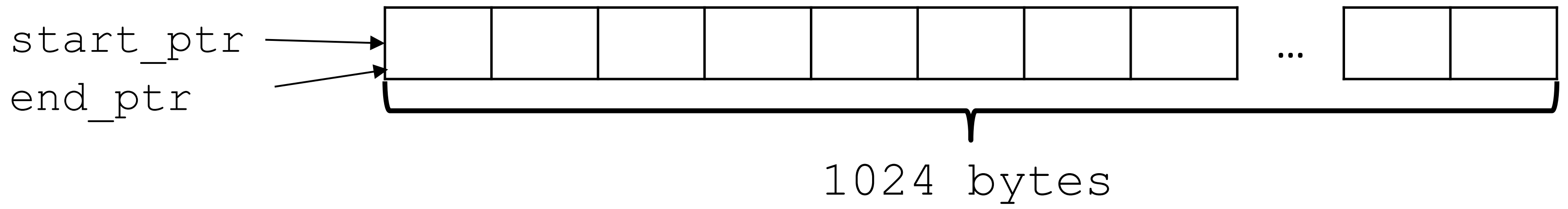
  - Note that there is a little bit more metadata than just these two pointers

- Then we allocate 4 bytes



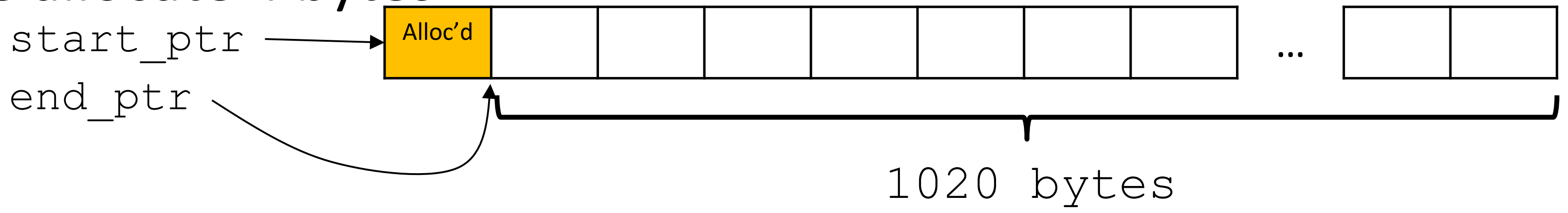
Arena Allocator: Alloc

- For example, Consider we start with:

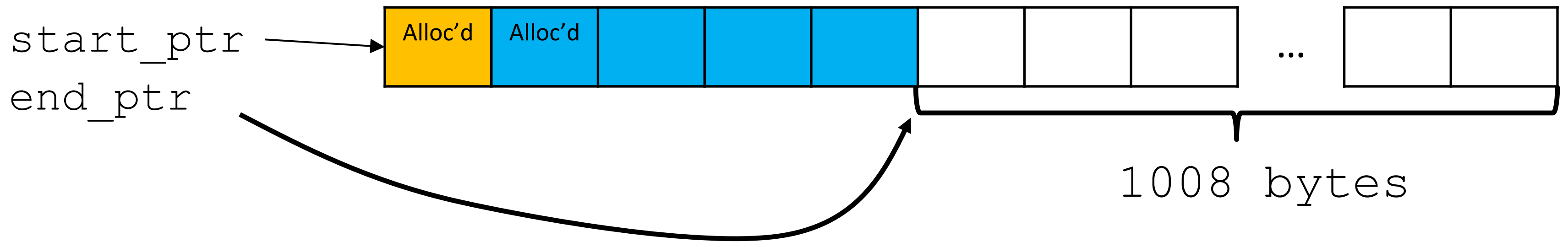


  - Note that there is no metadata, just these two pointers

- Then we allocate 4 bytes



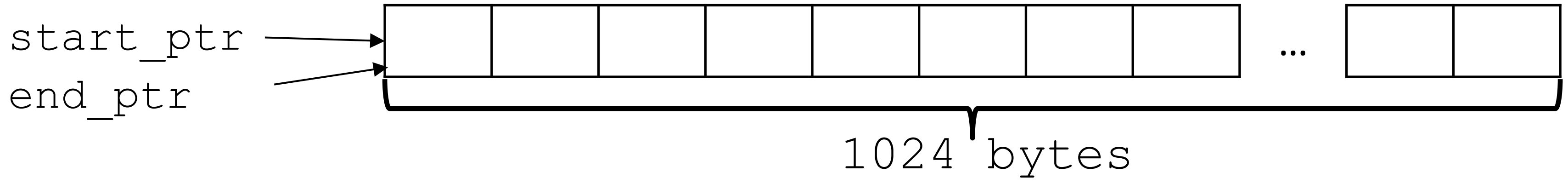
- Then we allocate 16 bytes



*Alignment could be a thing that affects how we allocate things, but we are leaving that out

Arena Allocator: Free

- Once we are done with our temporaries, we free the all allocations, and we can then use it again as if "fresh"



  - Looks the same as when we started!
- **That is the API**
- Example usage:

```
temp_allocator t_alloc = init_allocator();
for (many iterations) {
  int *ptr = allocate(t_alloc, 4 bytes);
  image *img = allocate(t_alloc, 1024 bytes);
  // a bunch of other allocations local
  // to this scope
  clear_allocs(t_alloc);
}
```

  - Instead of being scoped to a function, an arena allocator can also be scoped to an "object" or the lifetime of some "task"

Arena Allocator: Growing

- This simple arena allocator we are showing can also be called a “bump allocator” since to allocate we just “bump” the pointer

- All the memory for an arena allocator is allocated before hand, typically there is a good guess for the memory that a given scope will need, so we can just allocate that many pages or bytes

- If we want to handle growing an arena allocator, it may handle multiple “arenas” and simply allocate a new arena whenever one is requested.
  - Can allocate new pages by using mmap() to create “anonymous” mappings (anonymous = pages aren’t mapped to a file)

Poll Everywhere

pollev.com/tqm

- How fast is our arena allocator at allocating things on average? At freeing things?

- What does the internal and external fragmentation look like with our arena allocator?

- Why can’t we use this as a replacement for malloc maintaining lists of allocated & freed memory?

Poll Everywhere

pollev.com/tqm

- How fast is our allocator at allocating things on average? At freeing things?

  Very Fast, constant time for each

- What does the internal and external fragmentation look like with our allocator?

  Minimal/none for both ☺ Since we know how big each allocation is, we can allocate the exact size requested (no internal) and chunk our memory so that there is minimal space between each allocated chunk

- Why can't we use this as a replacement for malloc maintaining lists of allocated & freed memory?

  Malloc manages things that are freed individually that may be allocated for varying lengths of time. This allocator assumes everything can be allocated together.

Lecture Outline

- Garbage collection
- Arena Allocation
- **Buddy Algorithm**
- Slab/Slub Allocator

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`)
  - Linux Kernel uses the buddy algorithm to allocate physical pages to the kernel

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?

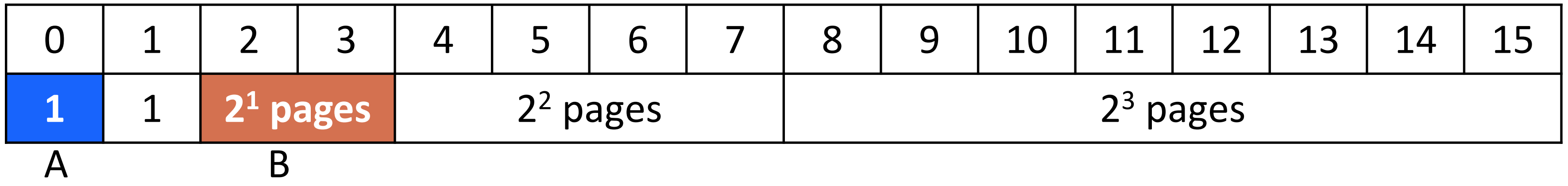
Buddy Algorithm walkthrough

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?

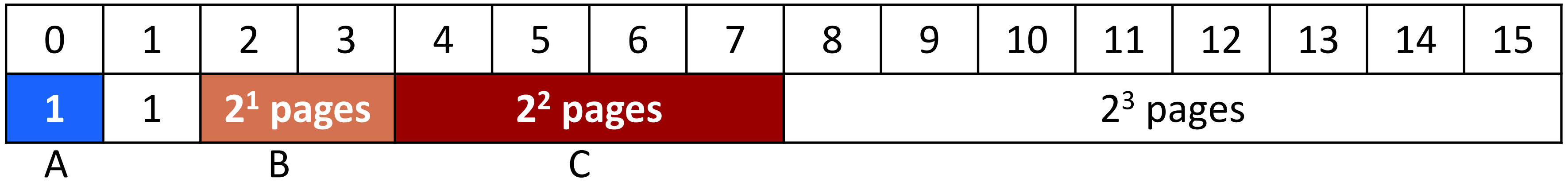
- We can claim the 2^1-page chunk and mark it as being used by allocation **B**

Buddy Algorithm walkthrough



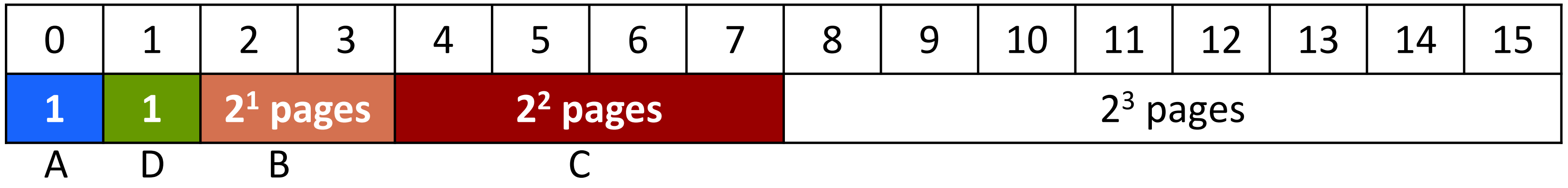
- Now someone requests 3 pages, what happens?

Buddy Algorithm walkthrough



- 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**

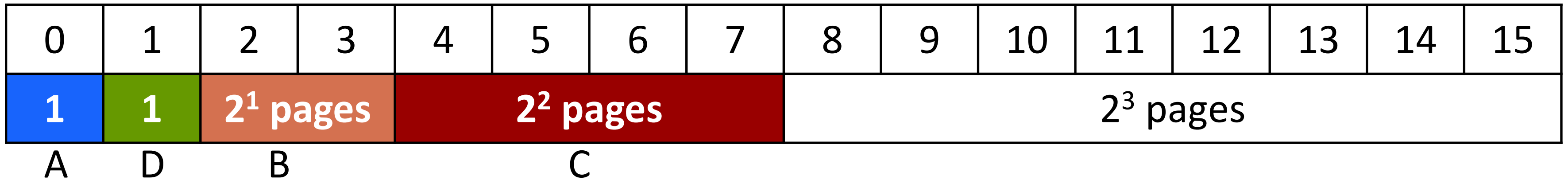
Buddy Algorithm walkthrough



- Last allocation: someone allocates 1 page, what happens?

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

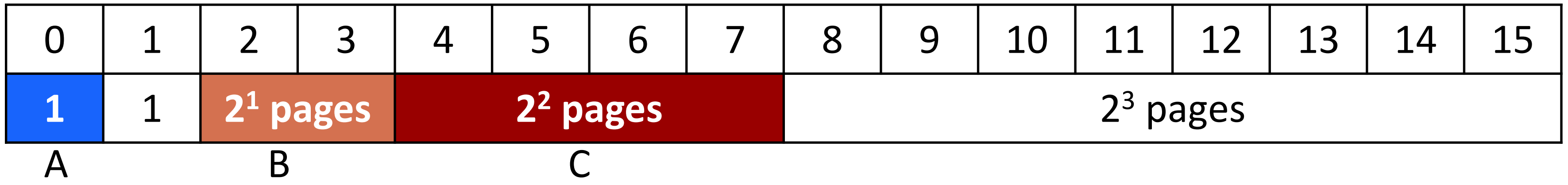
Buddy Algorithm walkthrough



- Let’s walk through the freeing process

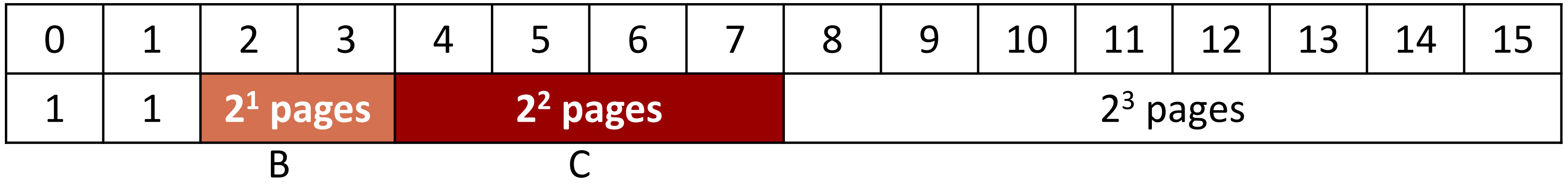
- First, allocation D is done and frees its page

Buddy Algorithm walkthrough



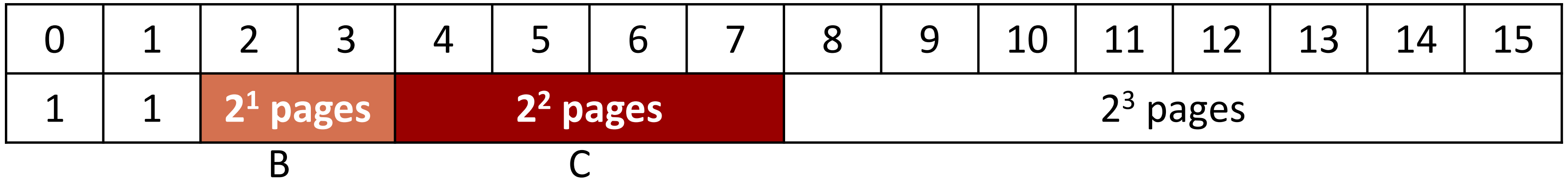
- Let’s walk through the freeing process
- First, allocation D is done and frees its page
- To free the page, we just mark it as no longer being allocated. Nothing we can coalesce (yet)

Buddy Algorithm walkthrough



- 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

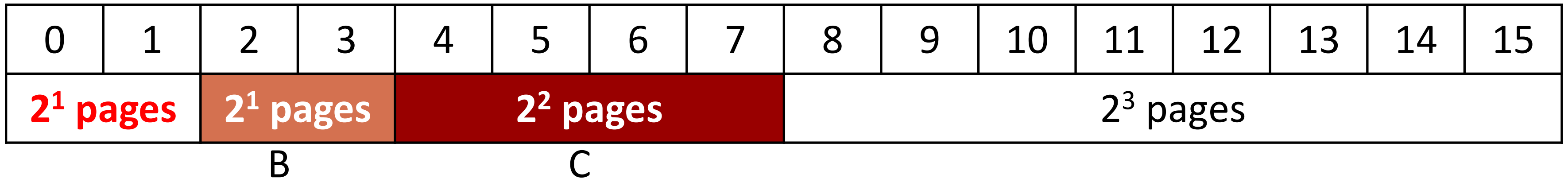


- Let’s walk through the freeing process

- 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

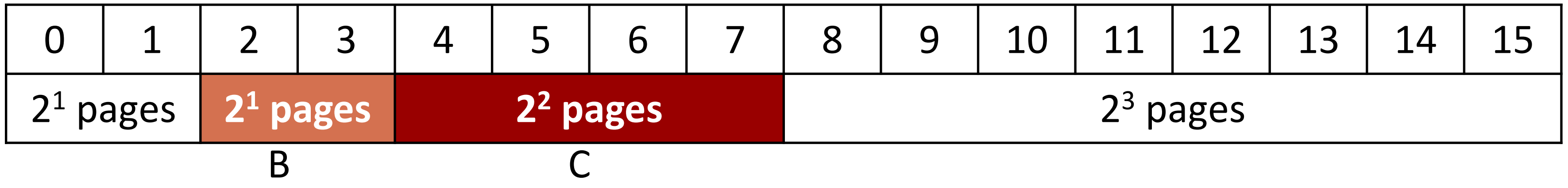


- Let’s walk through the freeing process

- 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 ☺

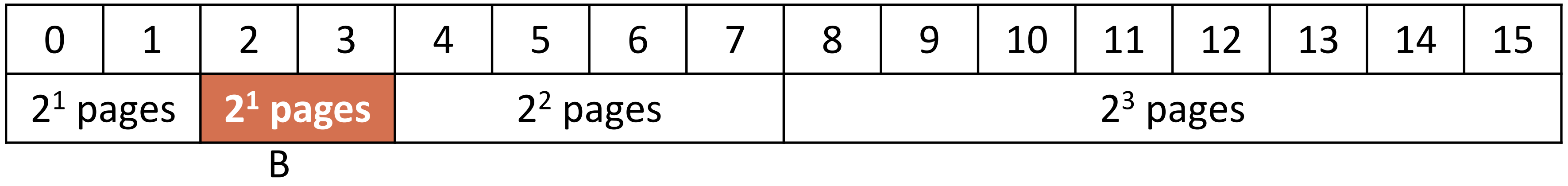
Buddy Algorithm walkthrough



- Let's walk through the freeing process

- Third, allocation C is done and frees its pages

Buddy Algorithm walkthrough



- 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!

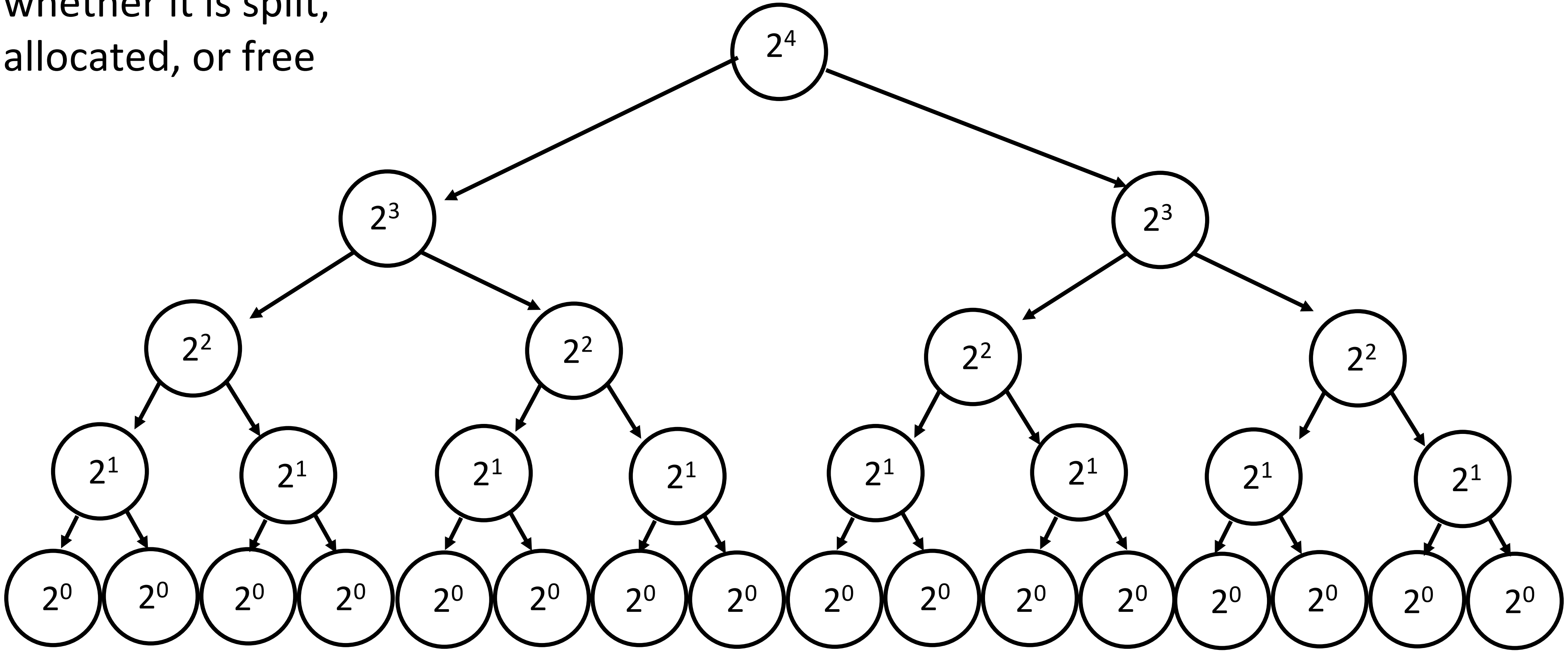
Buddy Algorithm walkthrough

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

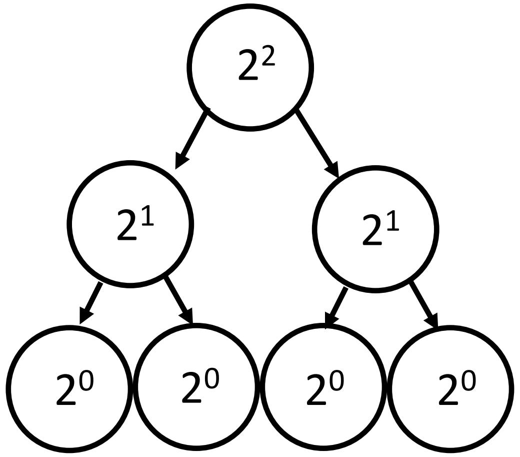
- Buddy Algorithm can be maintained with a binary search tree
  - Each node carries whether it is split, allocated, or free

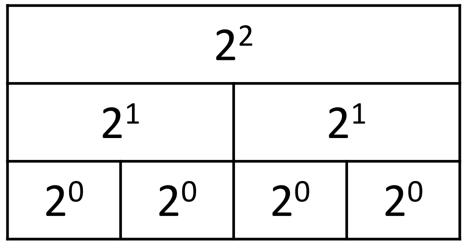


Buddy Algorithm Implementation

- 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 on the slide)

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

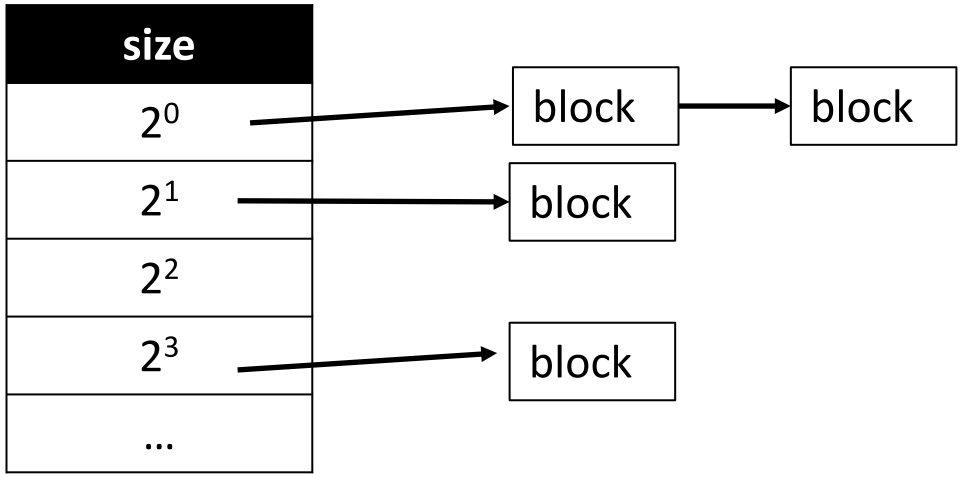




(alternate way to show the array, may make the connection between array and tree easier to see).
Indexes go Left -> Right, top to bottom

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
  - Coalescing is still fast since we can maintain a bitmap and easily find the location of a "buddy". If an allocation's "Buddy" is free it should be 2^k bytes before/after it.



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

- Garbage collection
- Arena Allocation
- Buddy Algorithm
- **Slab/Slub Allocator**

Slab Allocator*

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

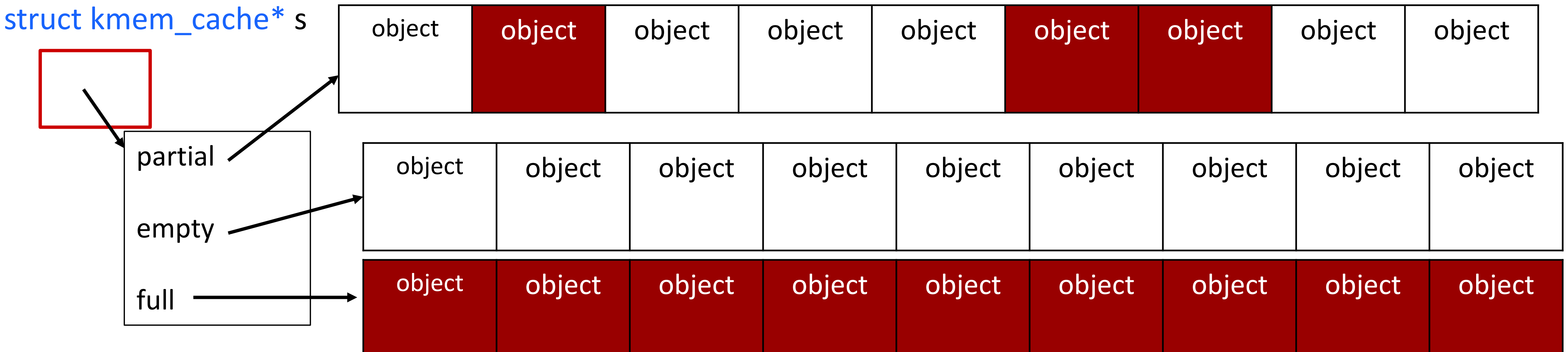
- First, you need to allocate the thing you will allocate from
  - When you create it, you specify a name and some other information
  - **The thing we care about is that you specify the size of the objects that the slab allocator will allocate from**

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

- We are simplifying this allocator a good bit

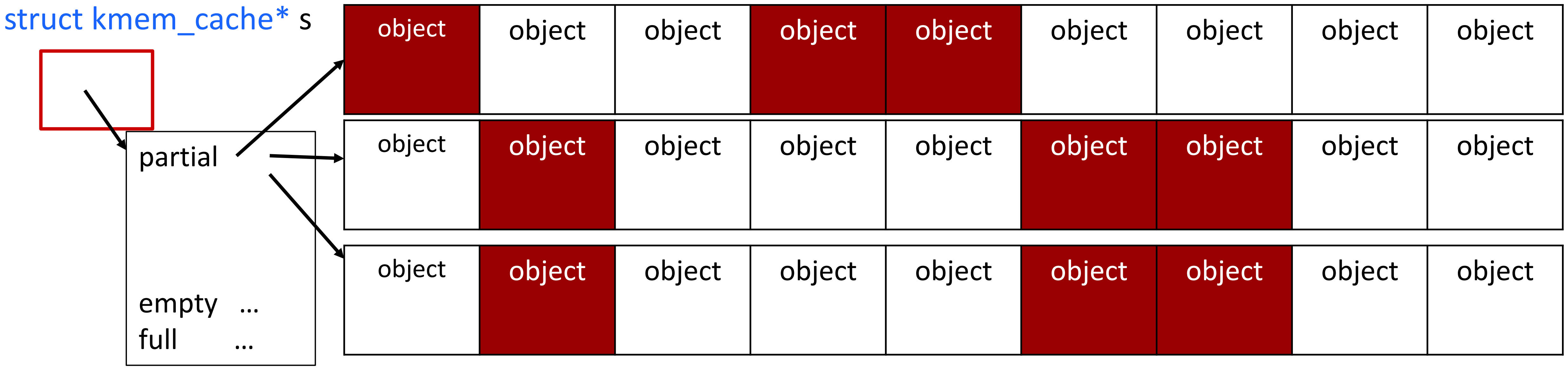
Slab Allocator High Level

- 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
  - slab: a chunk of memory containing the "objects"
  - A cache maintains lists of slabs noting which slabs are full/empty/partially in use



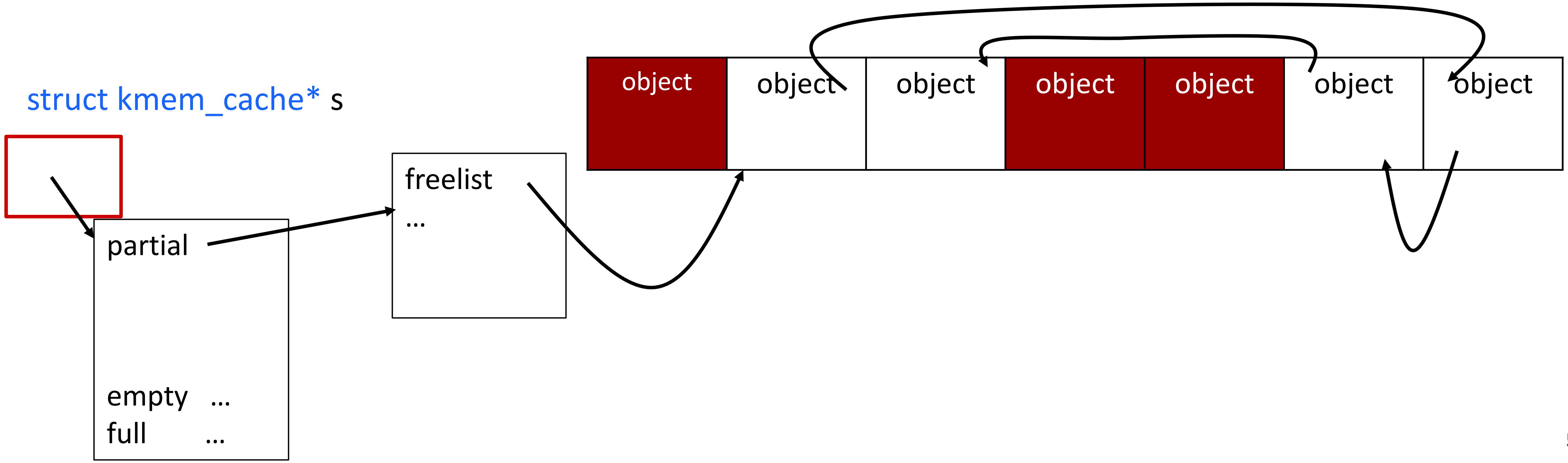
Slab Allocator High Level

- There can be multiple slabs that are partial/empty free


struct kmem_cache* s
partial
empty ...
full ...
object object object object object object object object object
object object object object object object object object object
object object object object object object object object object

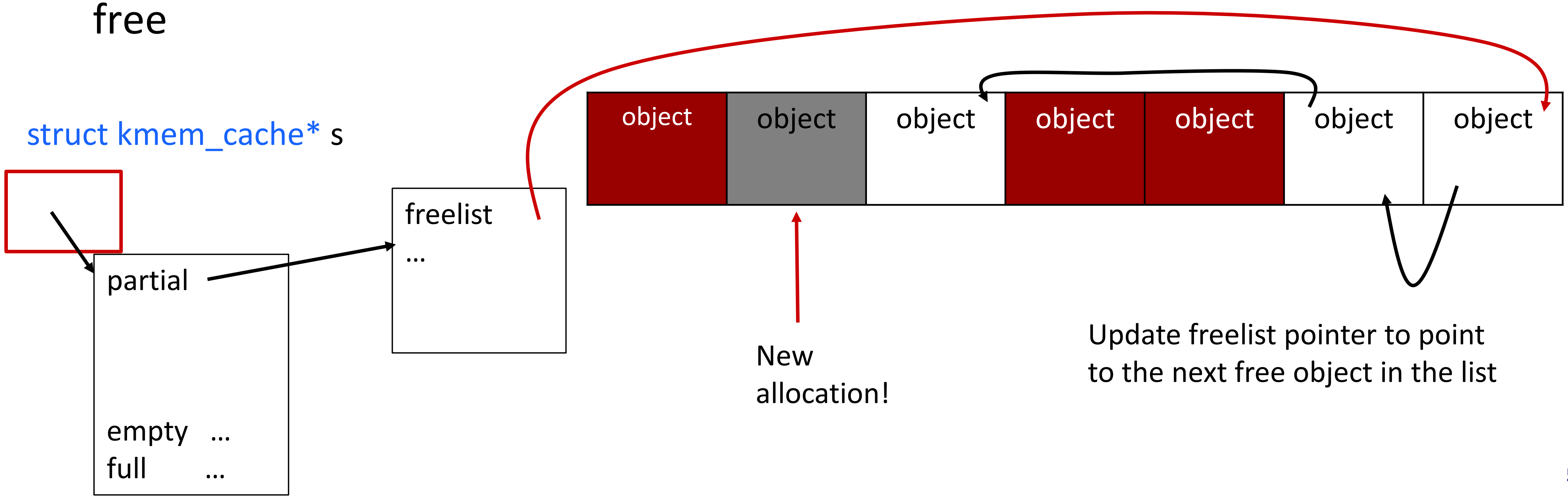
Slab Allocator High Level: Alloc

- Each slab maintains a pointer to an element that is free in the slab. (This pointer is stored in some metadata somewhere.)
- Each free object contains a pointer to the next free object in the slab
- When we allocate from the cache, we get a pointer to the first element that is free



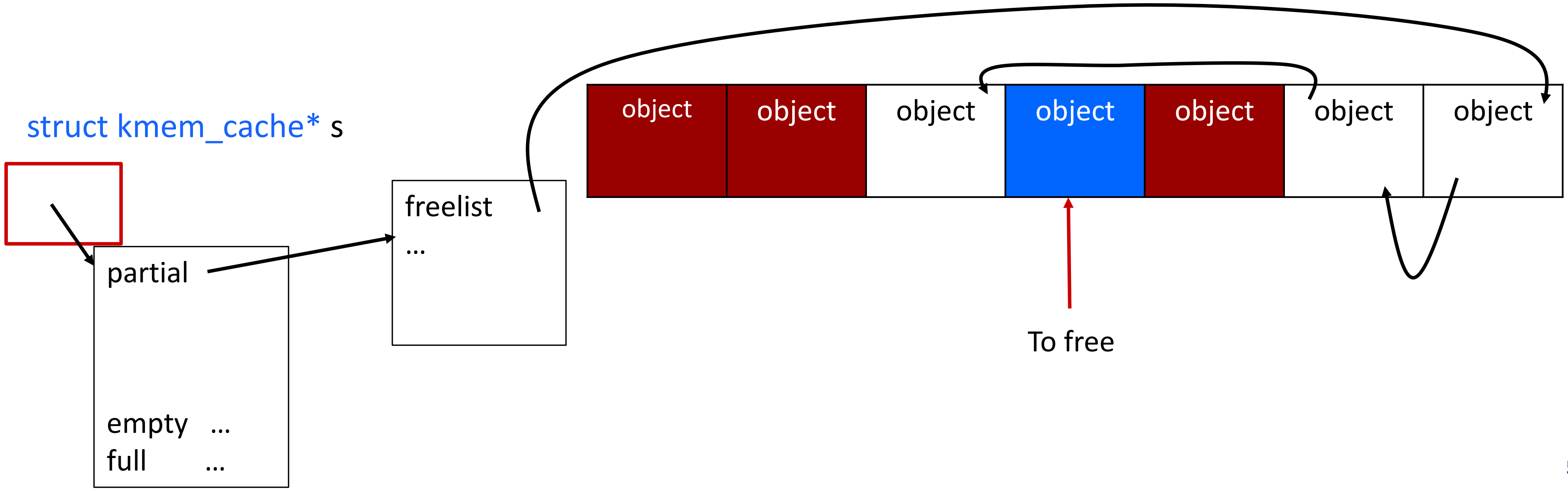
Slab Allocator High Level: Alloc

- Each slab maintains a pointer to an element that is free in the slab. (This pointer is stored in some metadata somewhere.)
- Each free object contains a pointer to the next free object in the slab
- When we allocate from the cache, we get a pointer to the first element that is free



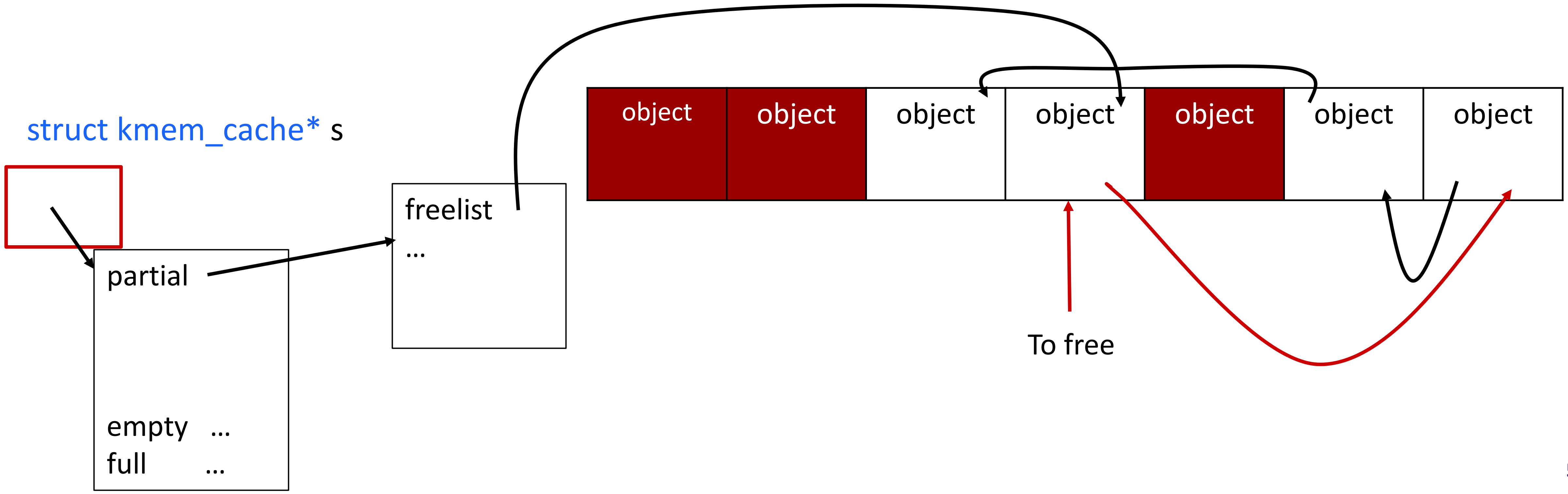
Slab Allocator High Level: Free

- When we want to free something, we are given the pointer to that object
  So we can do math on the address to calculate the page (and thus which slab it goes to)
- From there we can just "push it to the front of the free list"



Slab Allocator High Level: Free

- When we want to free something, we are given the pointer to that object So we can do math on the address to calculate the page (and thus which slab it goes to)
- From there we can just "push it to the front of the free list"



Poll Everywhere

pollev.com/tqm

- What is the runtime for slab?

- How does the fragmentation look?

Slab Allocator Analysis

- Slab allocator is pretty fast, O(1) ish, but can slow down when it needs to allocate a new slab

- Slab allocator is very useful for minimizing overhead for allocating and freeing.

- 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
  - The generic "kmalloc" (kernel malloc) is backed by the slab allocator.
    When it asks for N bytes it allocates from a slab that will best fit that allocation size.