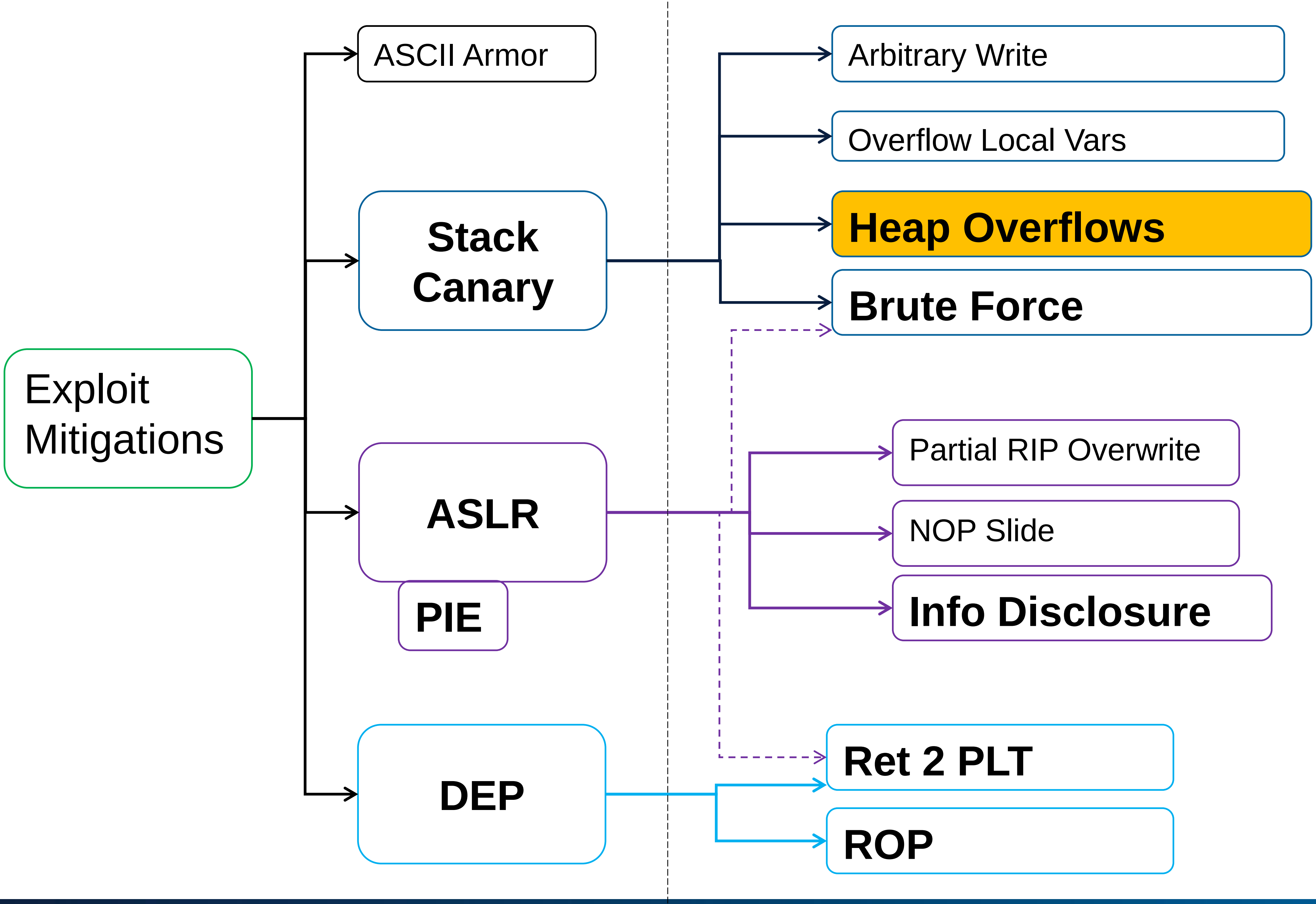
Defeat Exploit Mitigation Heap Intro

HEAP


Exploit Mitigations
ASCII Armor
Stack Canary
ASLR
PIE
DEP
Arbitrary Write
Overflow Local Vars
Heap Overflows
Brute Force
Partial RIP Overwrite
NOP Slide
Info Disclosure
Ret 2 PLT
ROP

Heap Exploitation

This slidedeck is not completely technically accurate

Should give an overview of heap exploitation concepts

Heap Introduction

What is a heap?

- malloc() allocations
- Fullfill allocating and deallocating of memory regions

Heap usage:

- Global variables (live longer than a function)
- Can be big (several kilobytes or even megabytes)

Reminder: Stack usage:

- Function-local variables
- Relatively small (usually <100 or <1000 bytes)

Heap Introduction

Heap:

- Dynamic memory (allocations at runtime)
- Objects, big buffers, structs, persistence, large things
- Slow, manually

Stack:

- Fixed memory allocations (known at compile time)
- Local variables, return addresses, function args
- Fast, automatic

Heap Introduction

Userspace/OS can implement his own memory allocator

- Linux: ptmalloc2 (previously dlmalloc)
- Samba: talloc
- FreeBSD and Firefox: jemalloc
- Google: tcmalloc
- Solaris: libumem

- Basically: mmap() a memory block and manage it
  - "Hey OS, give me 200mb of continuous memory. I will manage the details".

Heap Introduction

Heap in Linux

- Heap implementation is usually implemented in GLIBC
- Current Heap allocator implementation: ptmalloc2
  - Based on dlmalloc
  - From GLIBC 2.4 onwards

- Previous / Old:
  - Doug Lea’s memory allocator
  - Dlmalloc
  - Note: If you research heap exploits, check what allocator is assumed to be used

Heap introduction

malloc(): Get a memory region

free(): Release a memory region

We only cover manual allocations

- Not: Automatic garbage collection
- (Garbage collection is just an automatic free() by using reference counting)

Heap Interface

How does heap work?

void *ptr;

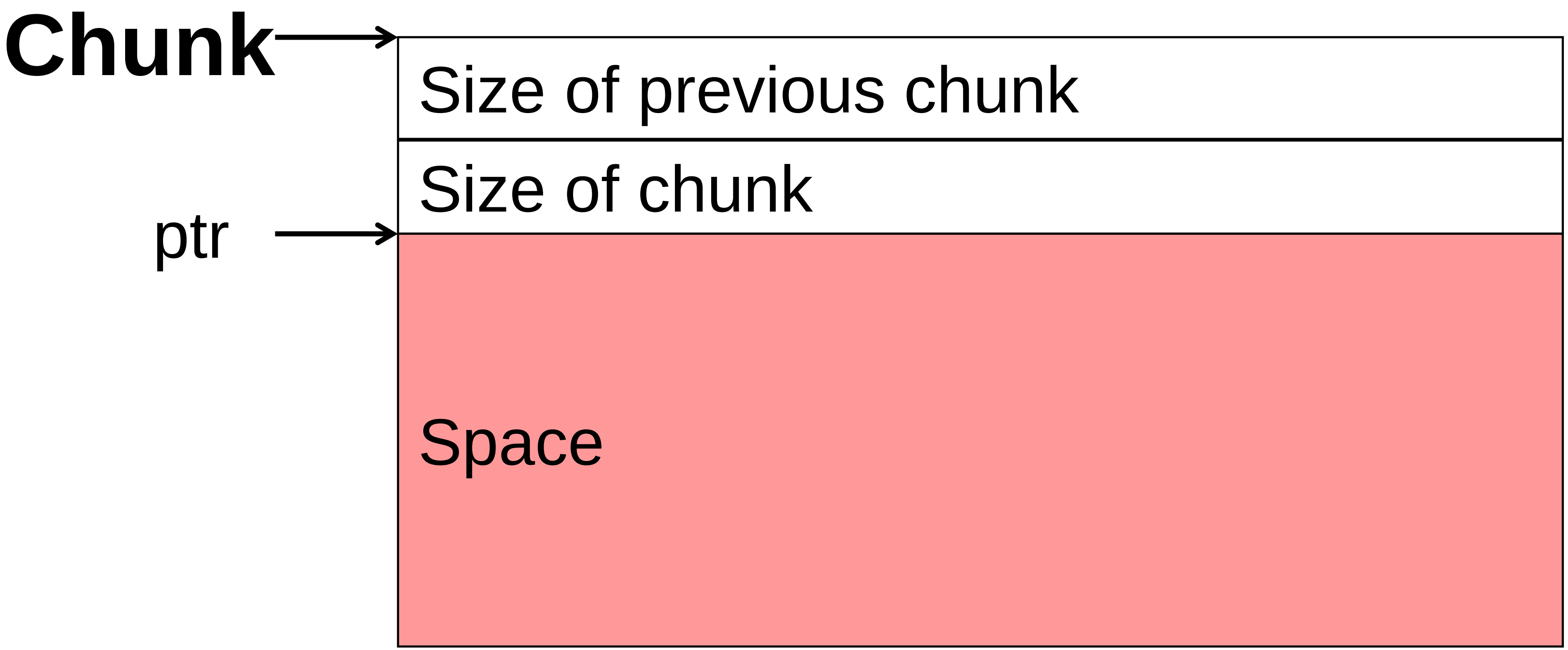
ptr = malloc(len)

- Allocated “len” size memory block
- Returns a pointer to this memory block

free(ptr)

- Tells the memory allocator that the memory block can now be re-used
- Note: ptr is NOT NULL after a free()

Heap Interface



Heap

What is a heap allocator doing?

- Allocate big memory **pages** from the OS
- Manage this **pages**

- Split the **pages** into smaller **chunks**
- Make these **chunks** available to the program

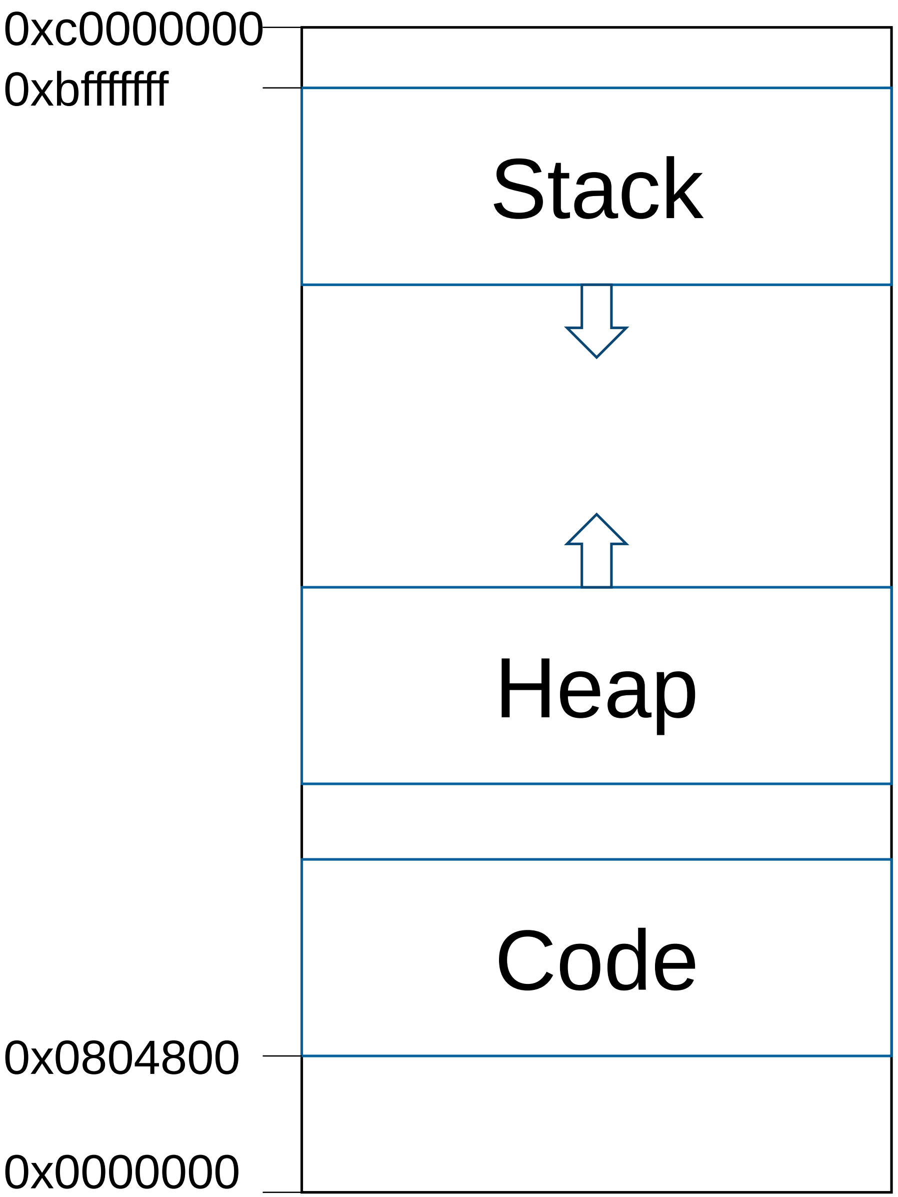
Heap – Simplified Example

Heap Introduction

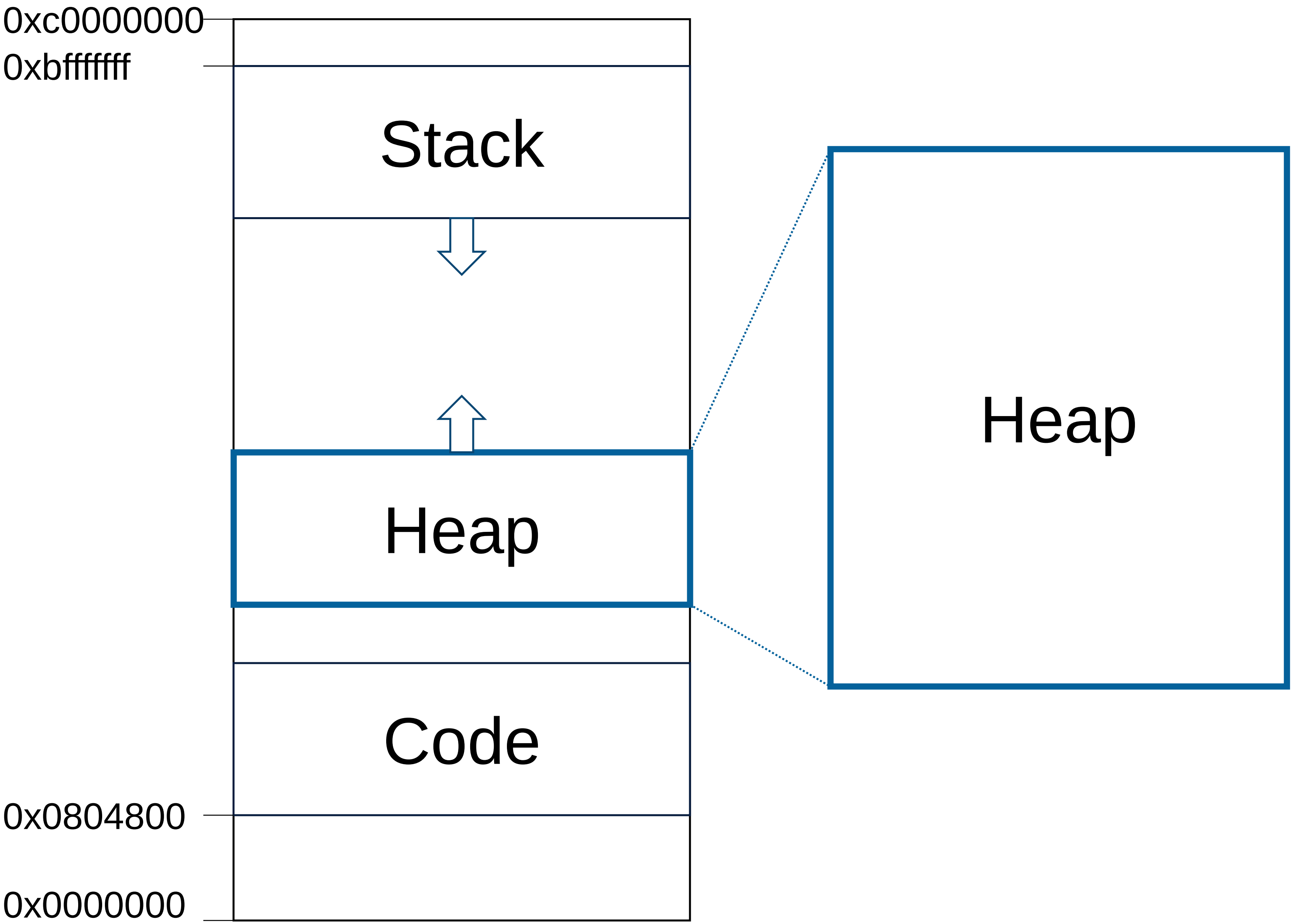
How is this implemented?

- The heap implementation gets a (big) block of flat/unstructured memory (**page** / pages)
- Partition the heap/page into **bin’s**
- A **bin** has **chunks** of the same size

Heap: Memory Layout



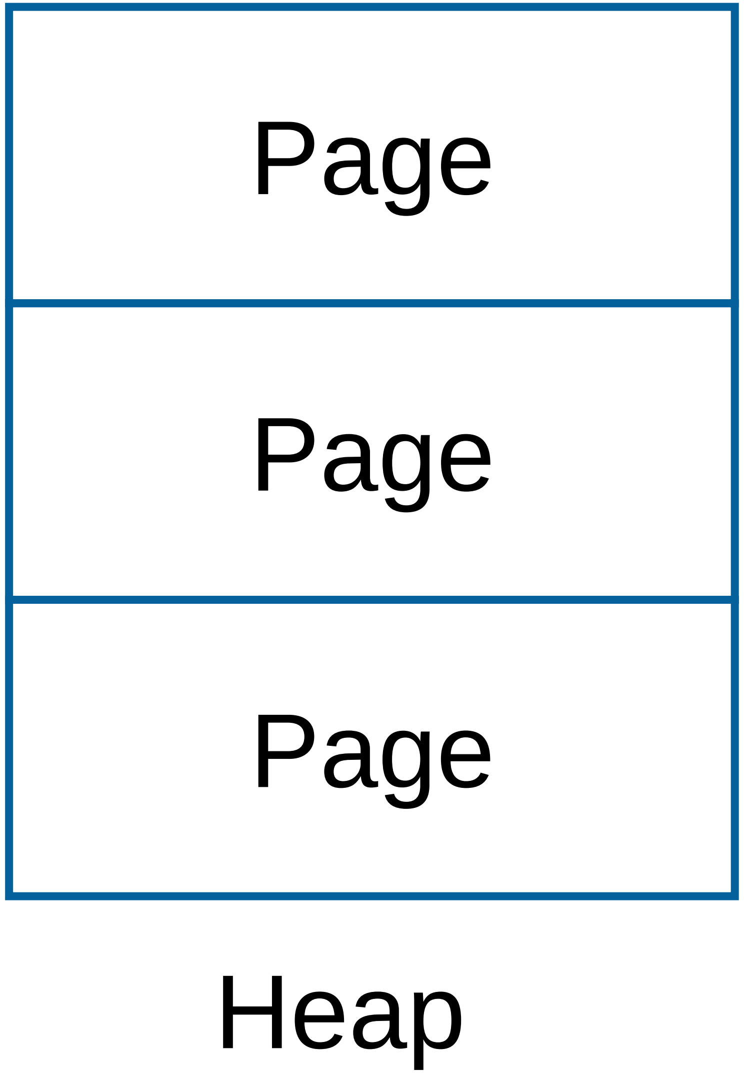
Heap: Memory Layout



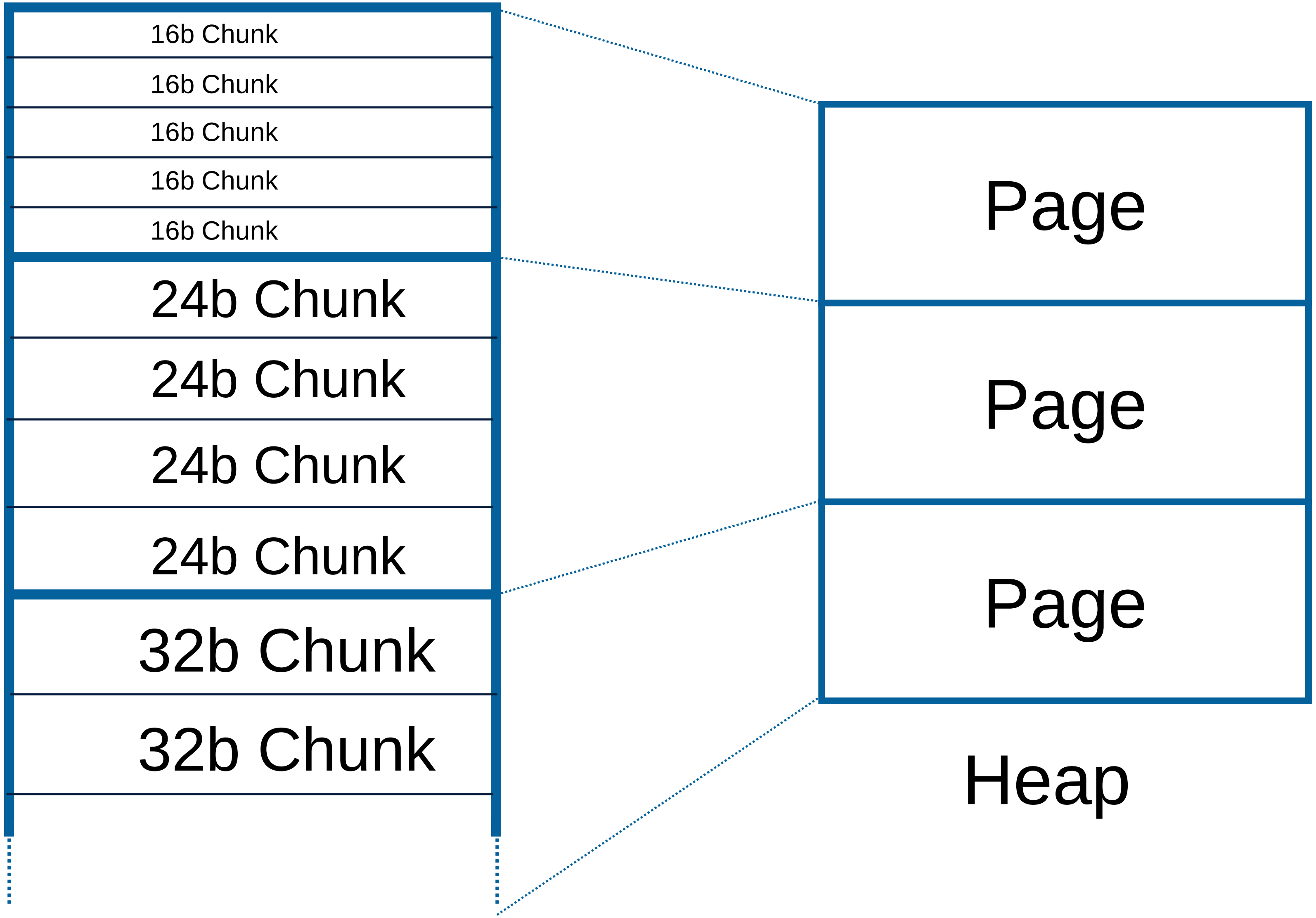
Heap: Memory Layout

Page:

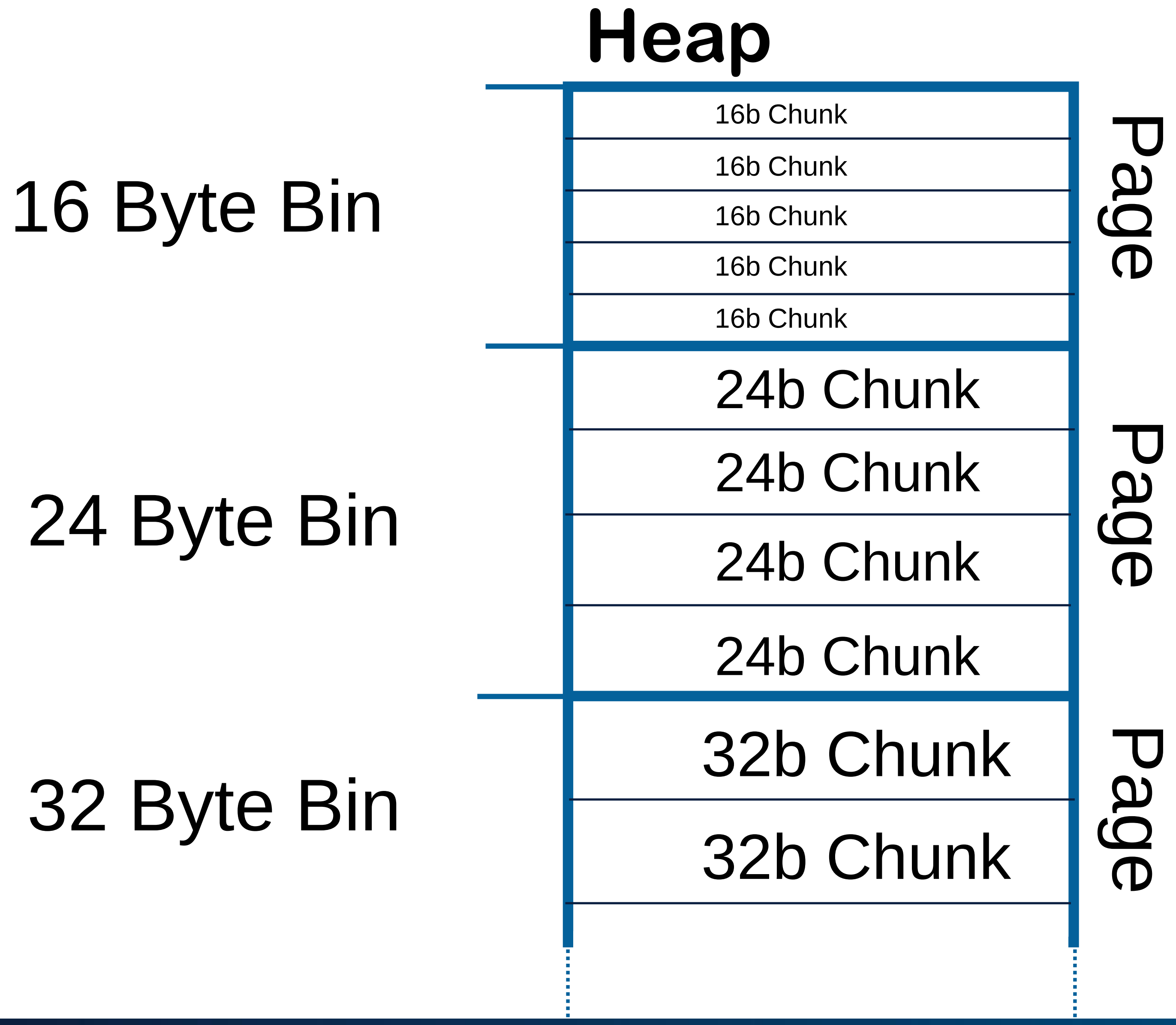
- A memory page
- Usually 4k
- Can also be 2 Megabytes or other
- Allocated via sbrk() or mmap()



Heap: Memory Layout

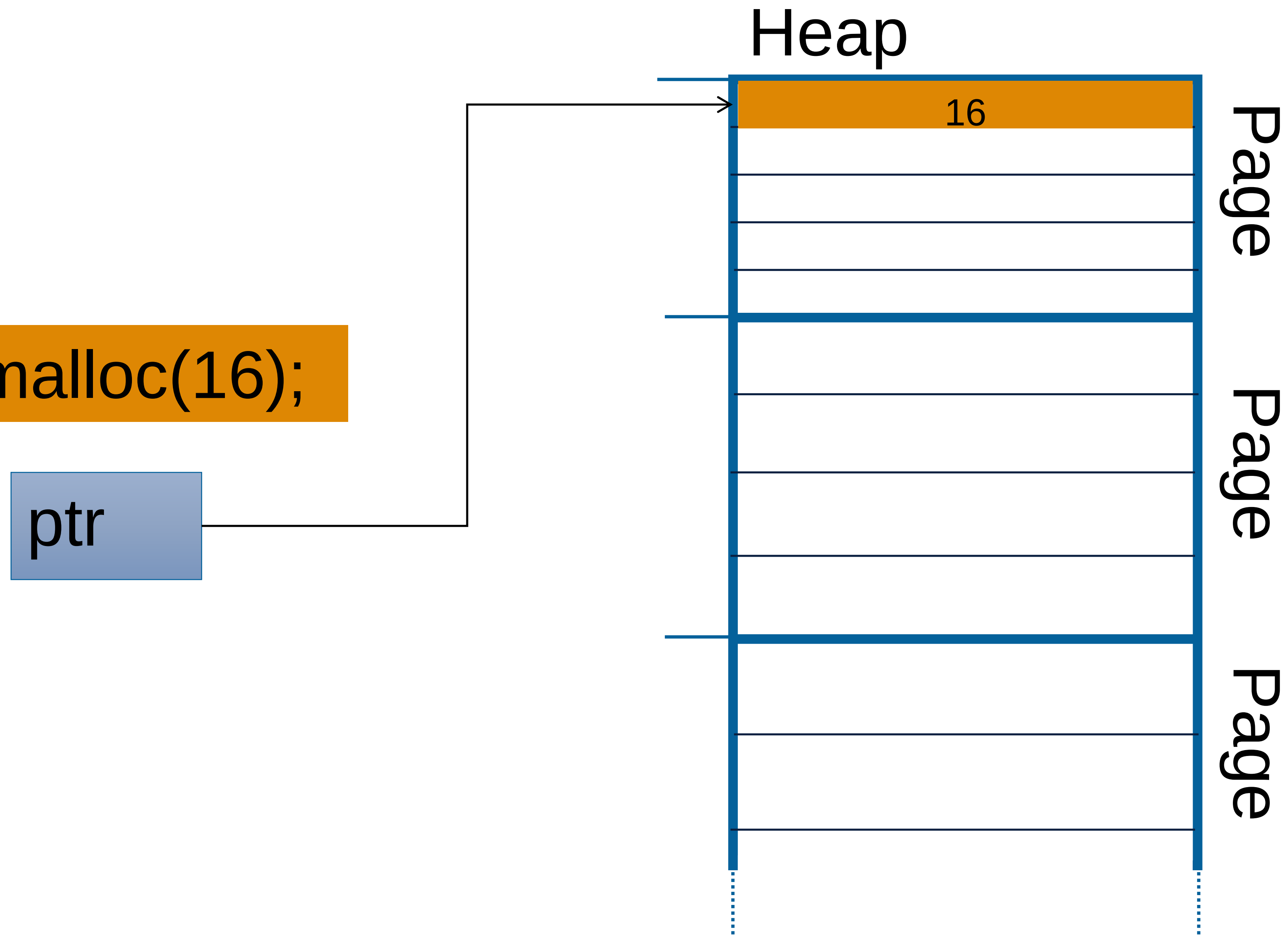


Heap: Oversimplified example



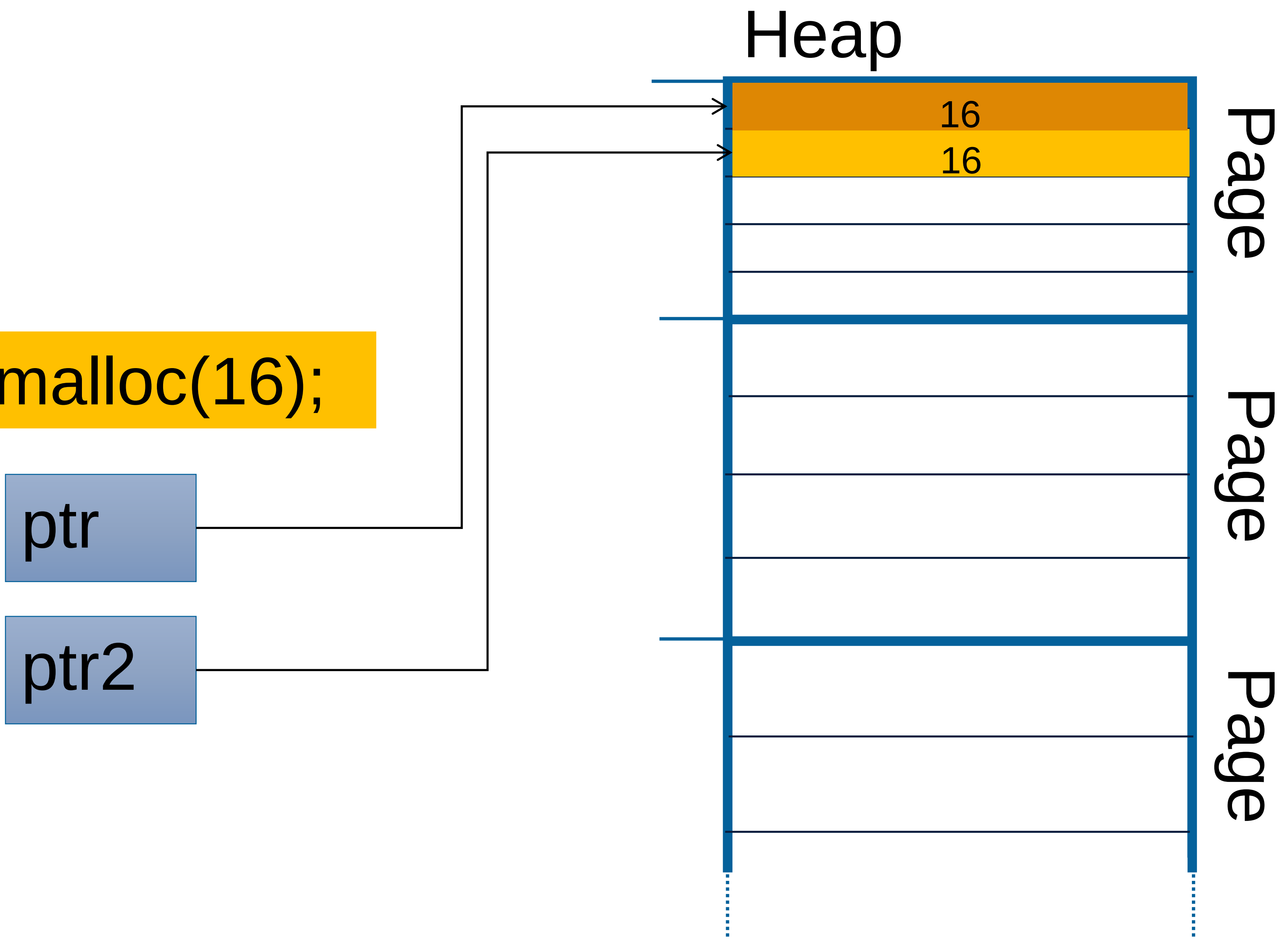
Heap: Oversimplified example

ptr = malloc(16);



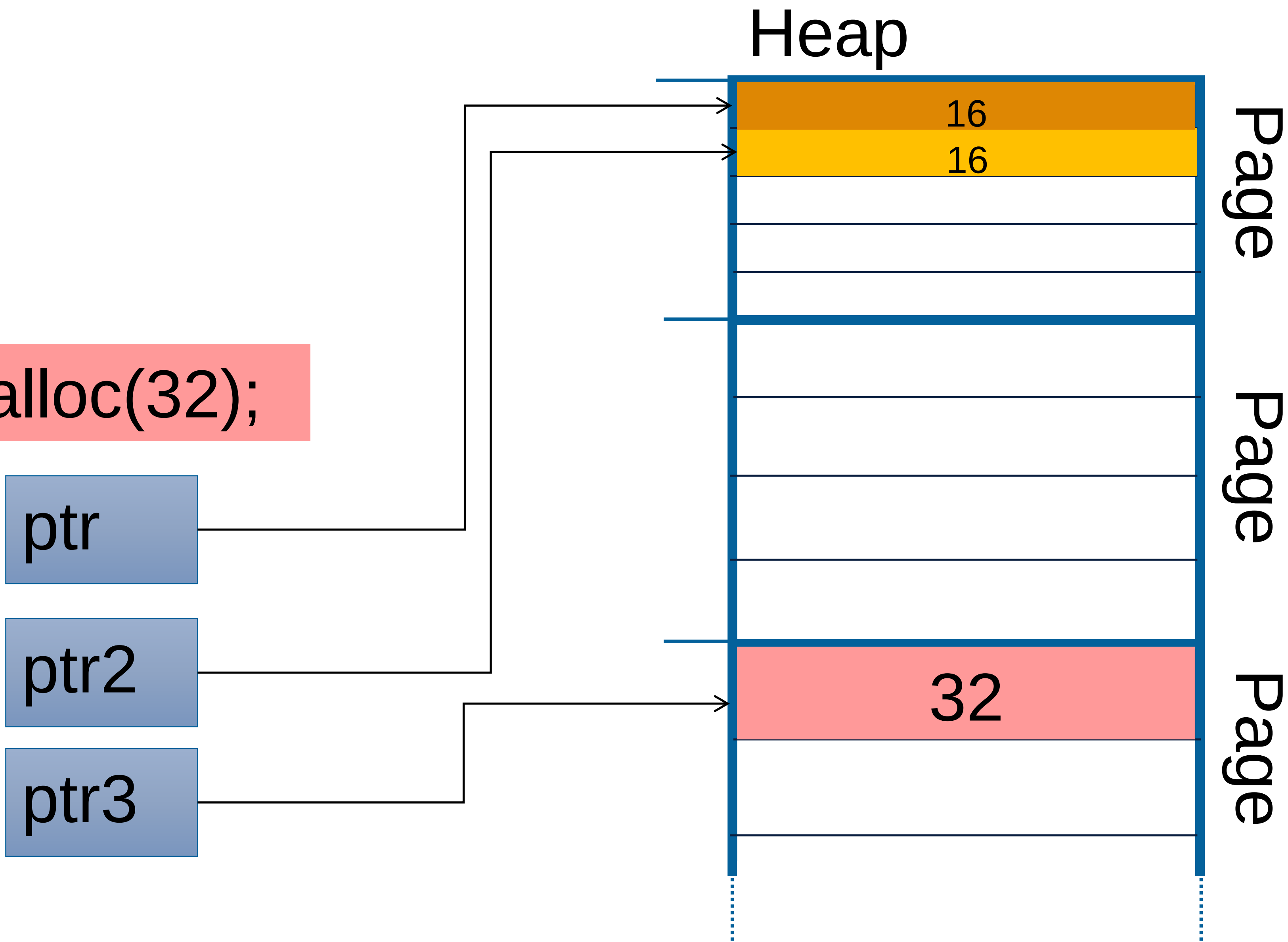
Heap: Oversimplified example

ptr2 = malloc(16);



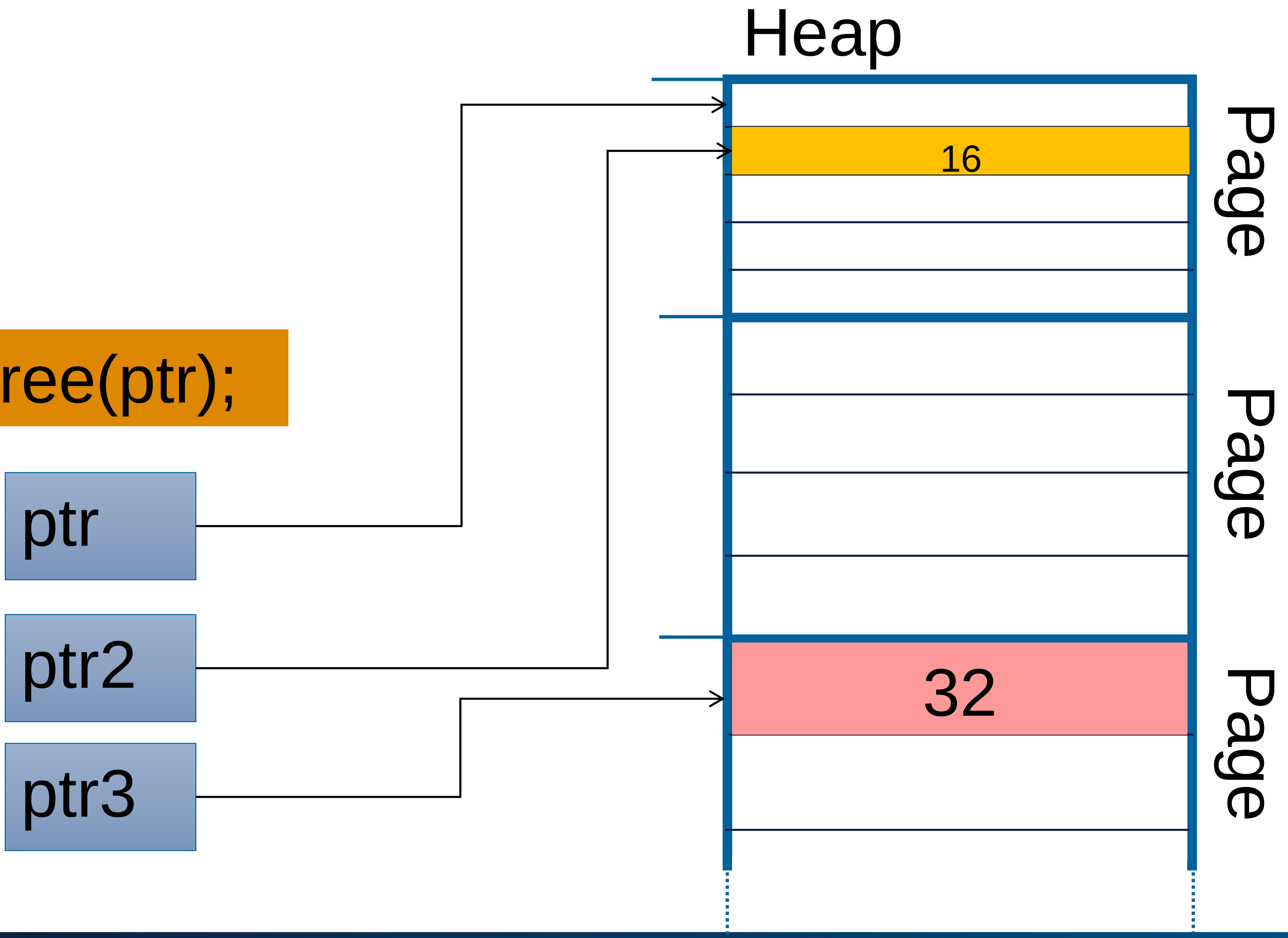
Heap: Oversimplified example

ptr3 = malloc(32);



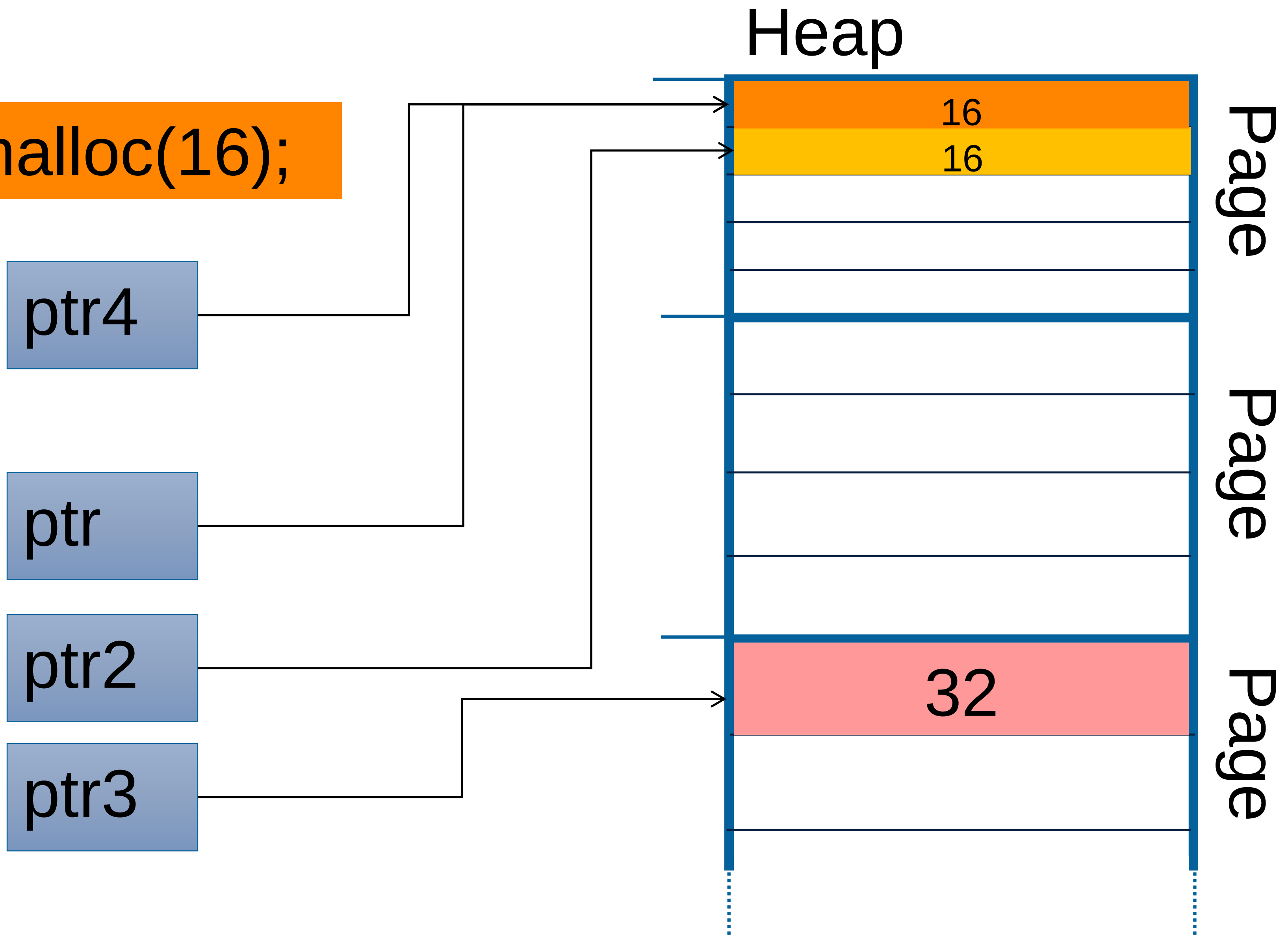
Heap: Oversimplified example

free(ptr);



Heap: Oversimplified example

ptr4 = malloc(16);



Heap - Recap

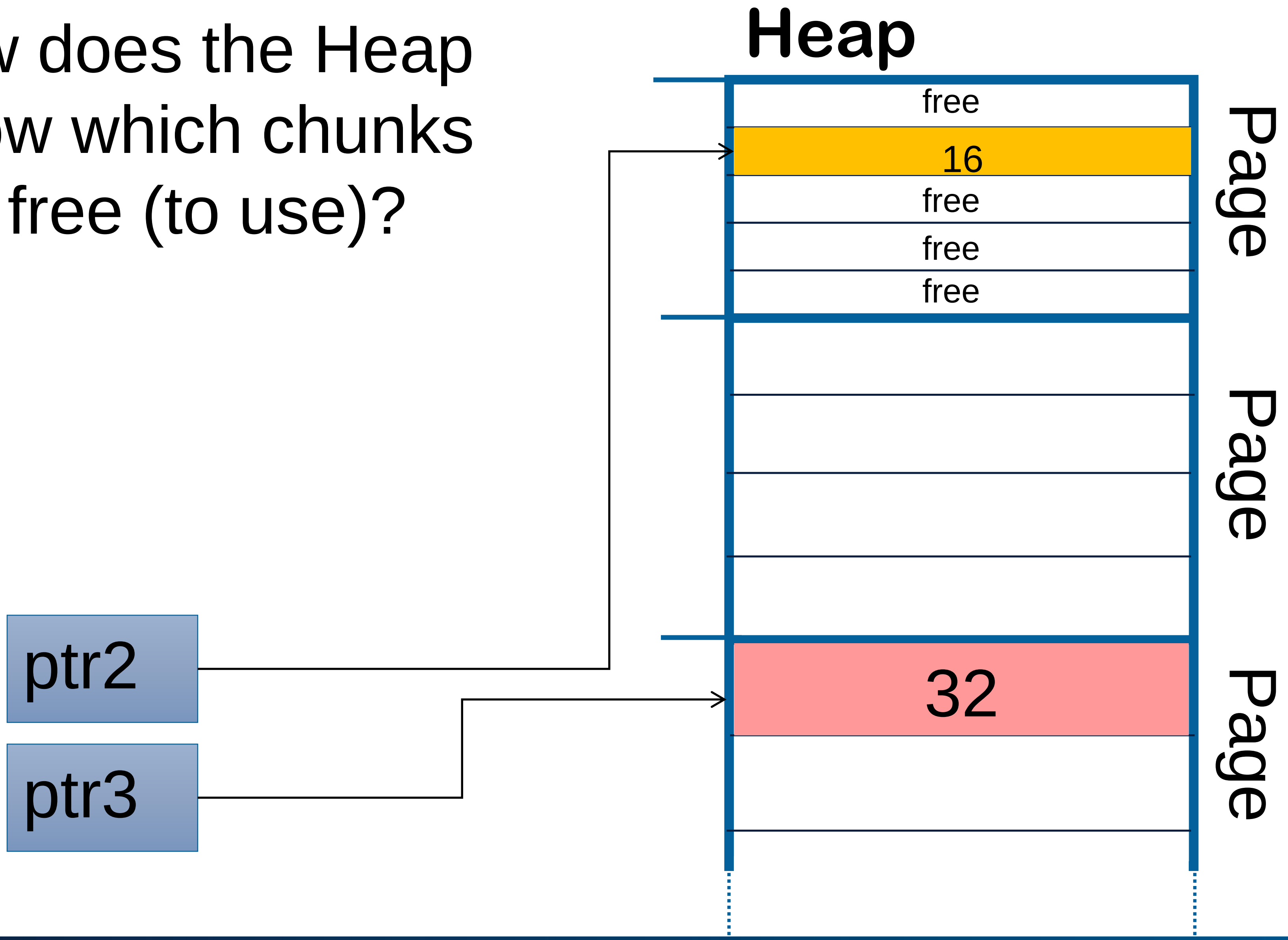
Recap:

- Heap divides big (4k) memory pages into smaller chunks
- Heap gives these chunks to the program on request
- A pointer to a heap allocation points to the data part (the chunk contains more metadata)

Heap Memory Management

Heap Memory Management

How does the Heap
Know which chunks
Are free (to use)?



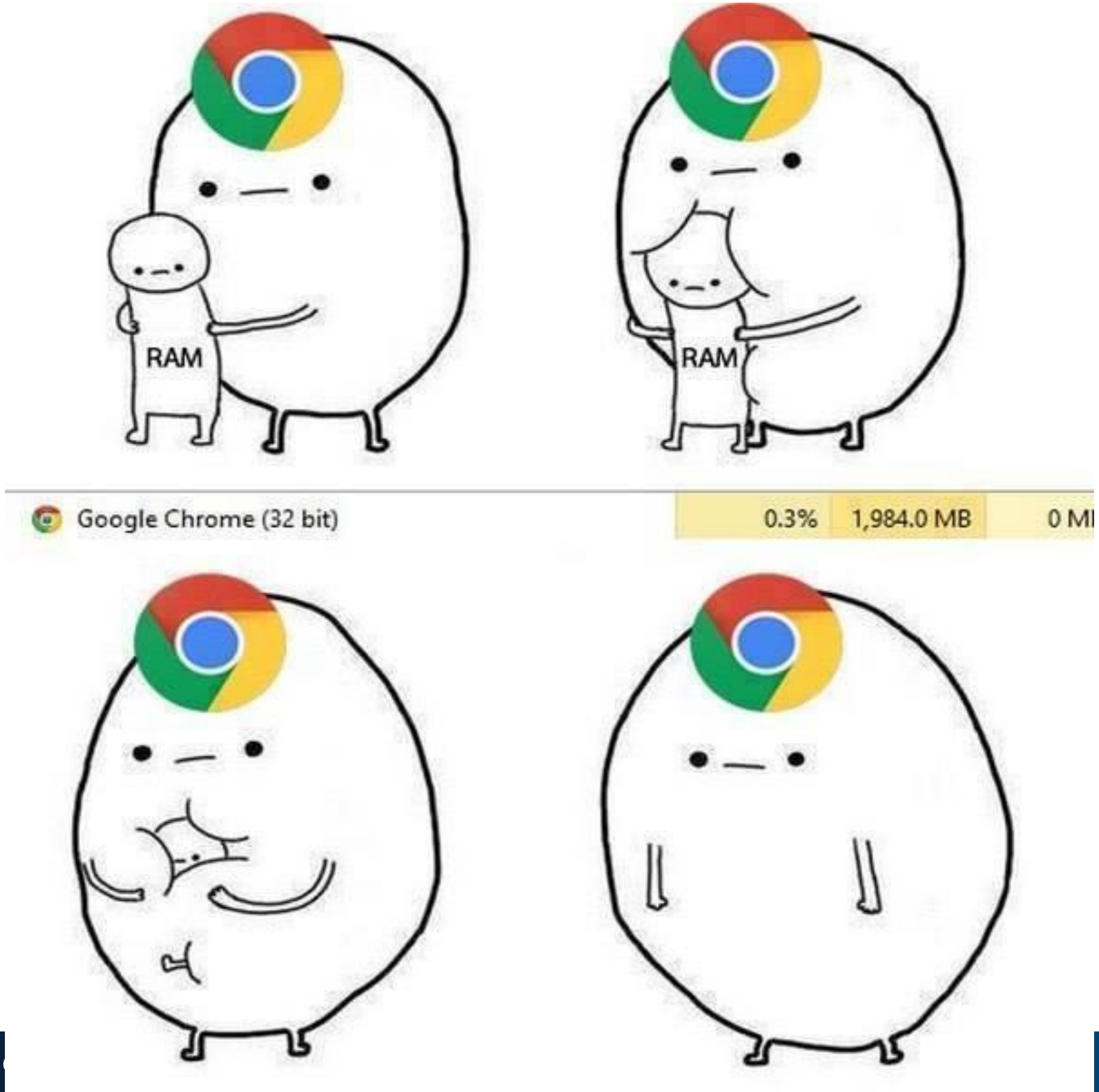
Heap Memory Management

Heap allocator requirements:

- Should be **quick** to fulfill malloc() and free()
- Should **not waste** memory by managing memory

- Also: No bugs, correct, low-fragmentation, etc.

Heap Memory Management



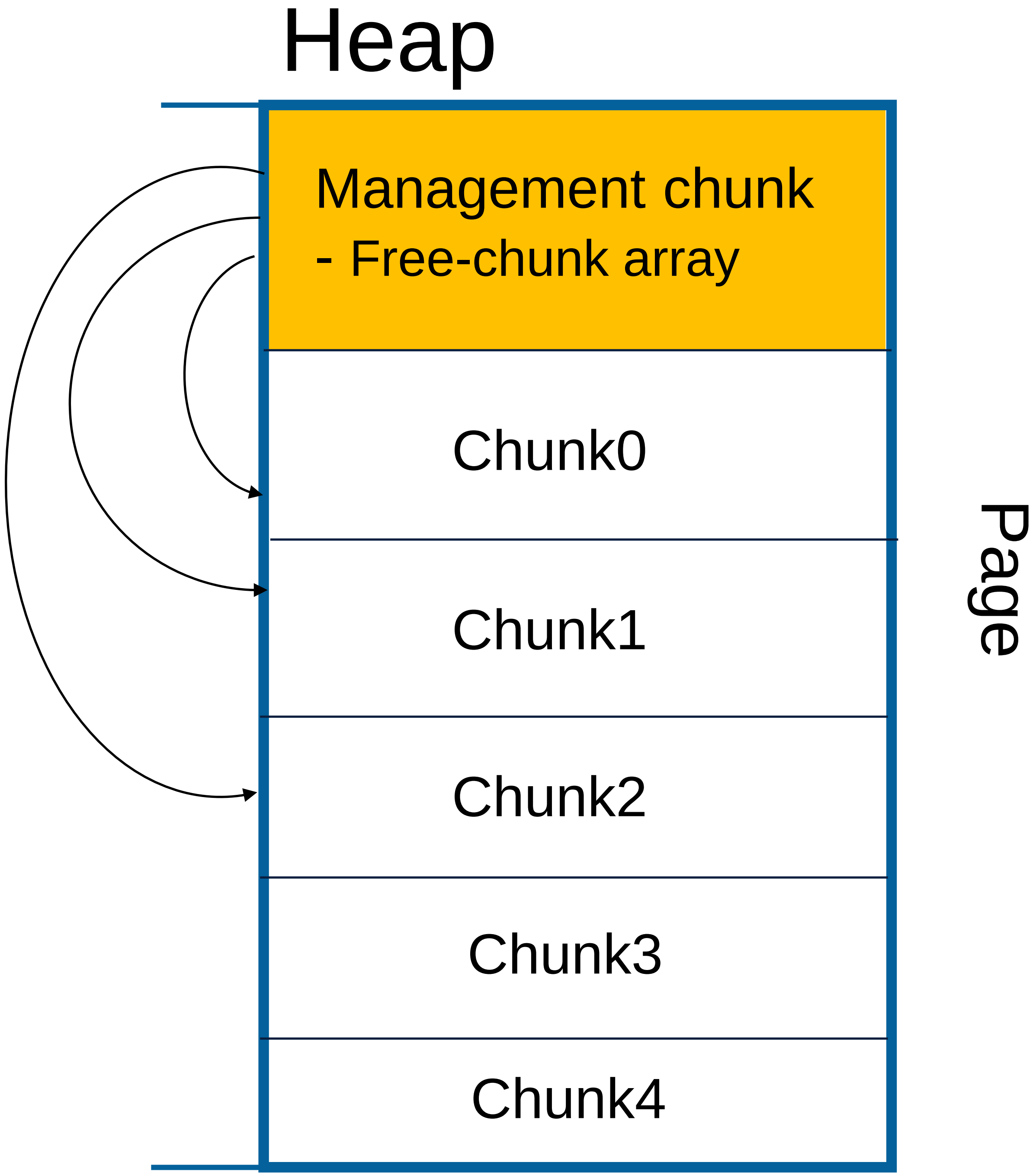
Heap Memory Management

One example of allocator:

PHP7 – emalloc

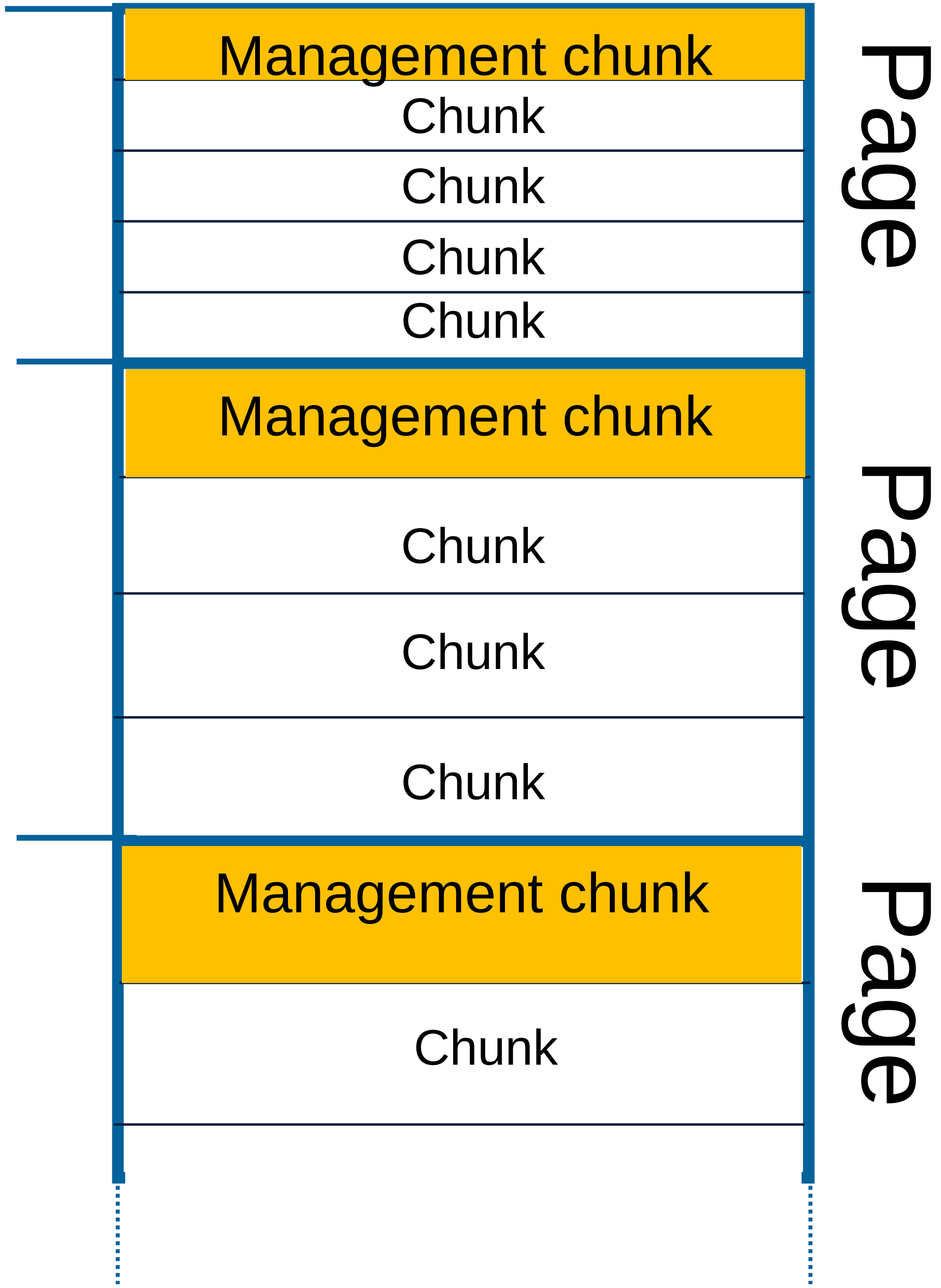
- First chunk has management information
- Management chunk describes other chunks
- Which are free, how big are they etc.

- *(ok, emalloc allocates chunks from the OS, divides them into pages - so the oposite naming convention. That’s a detail).*



Heap Memory Management

Heap could look like this:

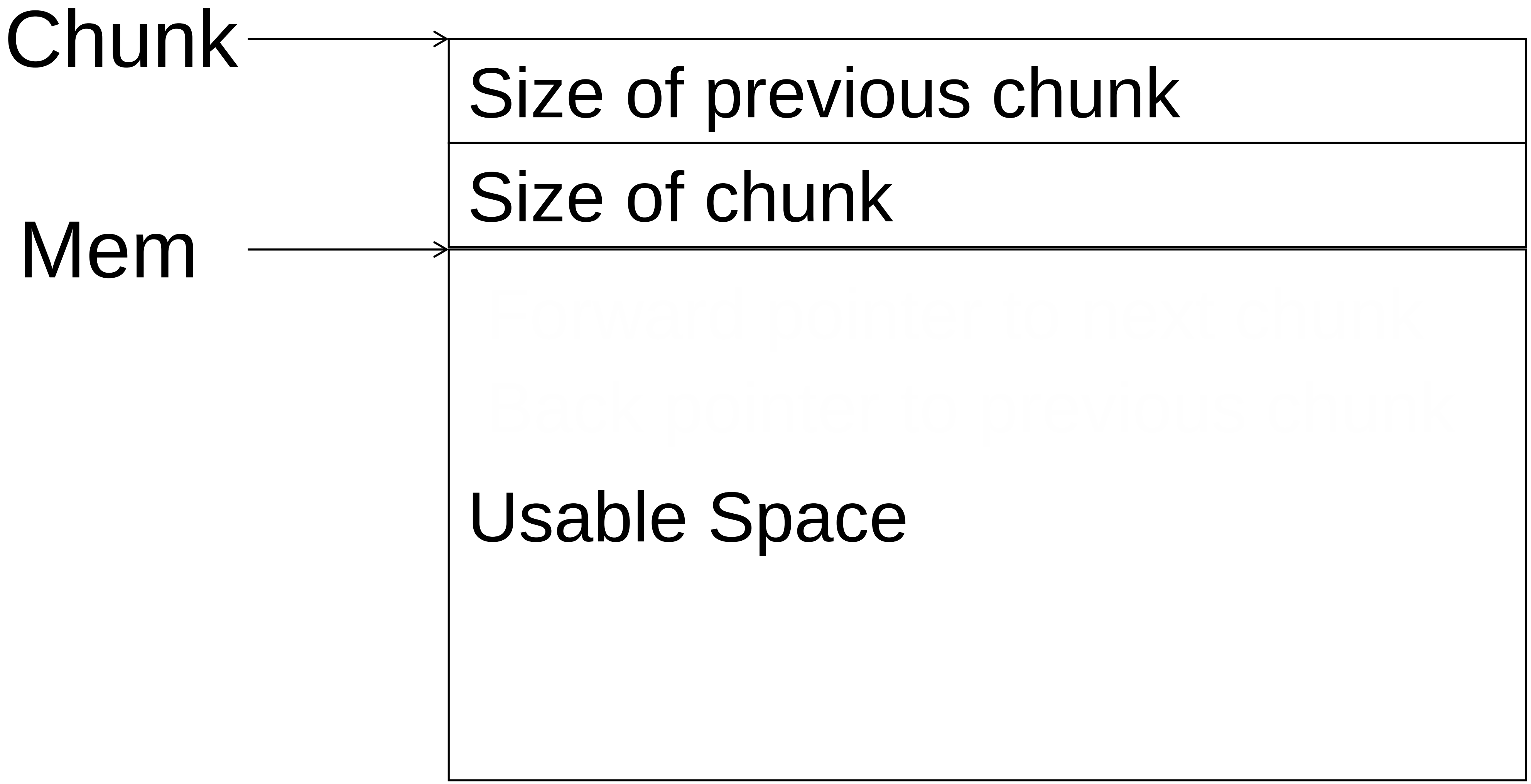


Heap Memory Management

But wait, there’s more!

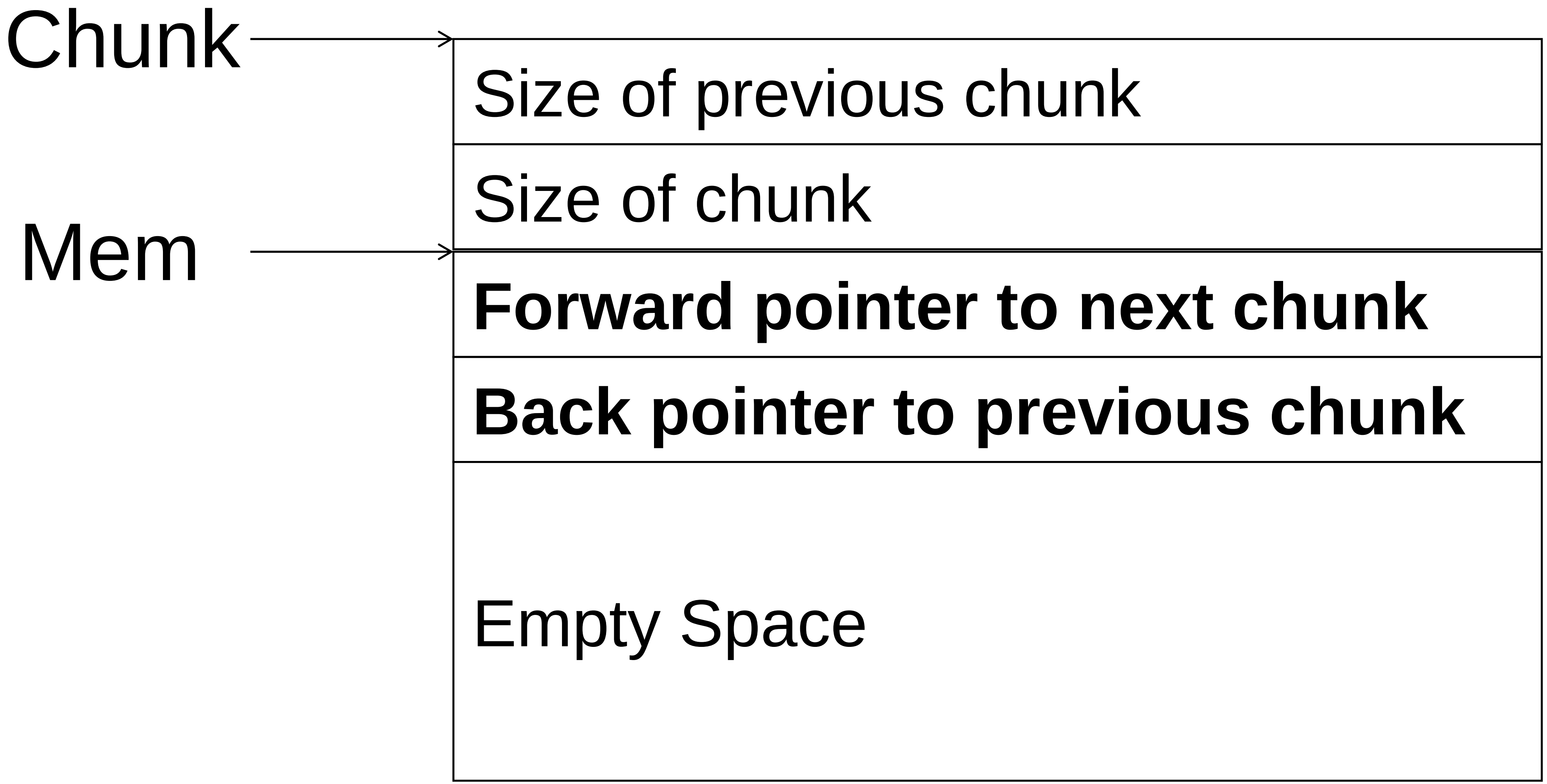
Heap Chunks

Ptmalloc2 **ALLOCATED** chunk:



Heap Chunks

Ptmalloc2 **FREE** chunk:



Chunk

Mem

Size of previous chunk

Size of chunk

Forward pointer to next chunk

Back pointer to previous chunk

Empty Space

Heap Chunks

- Free chunks contain heap-metadata in the usable space
- Free chunks organized in a linked list
- Adjenctant free chunks are merged in some allocators
  - This process is considerably useful for exploiting purposes

Heap attacks

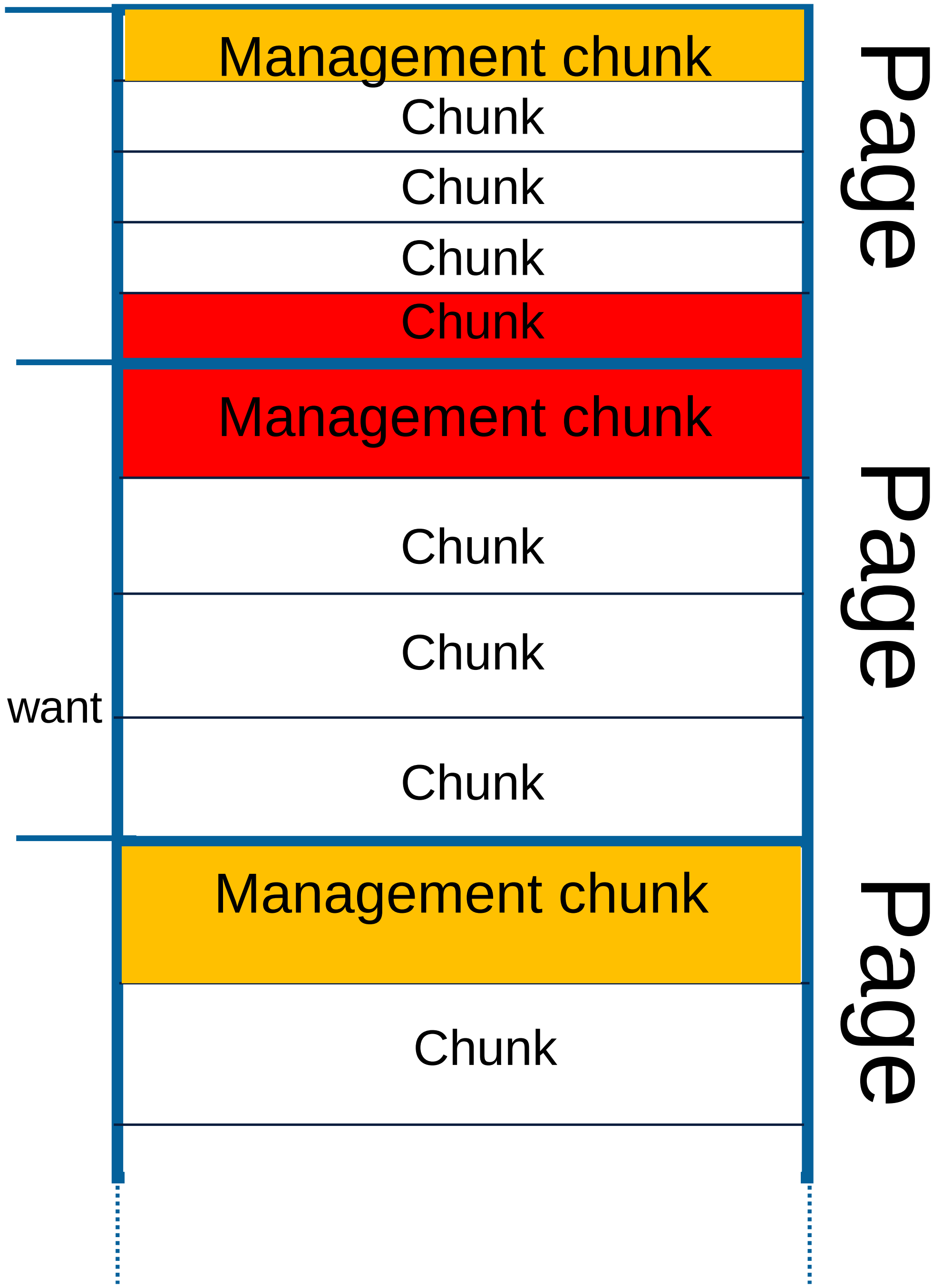
Heap Attacks: Buffer overflow

Heap attack:

Inter-chunk overflow with management chunk

Problem:

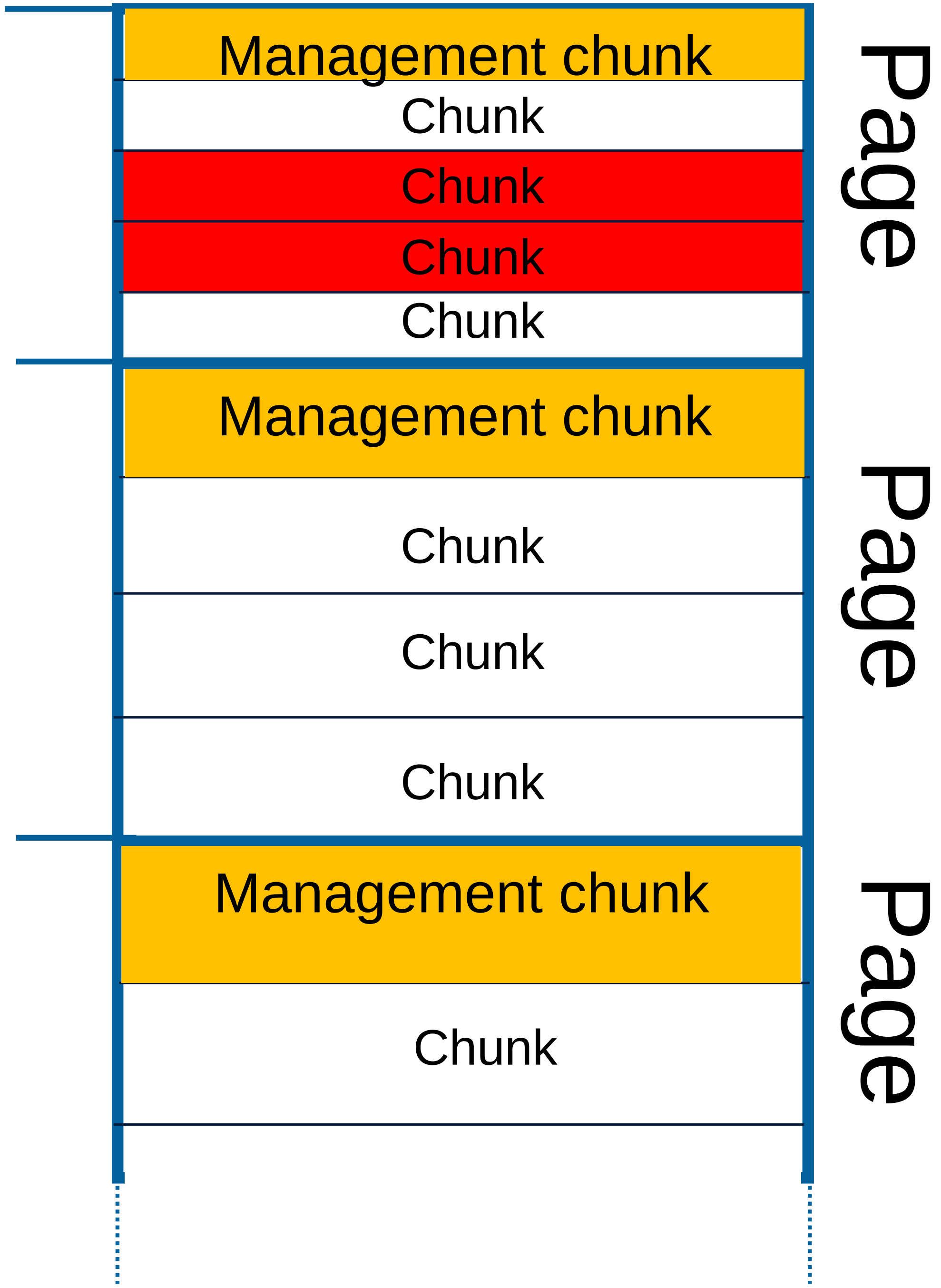
- In-band signaling (again)
- Can modify management data of heap allocator
- Therefore, can modify behavior of heap allocator
- Make the heap allocator write stuff we want to location we want
  - write-what-where
  - upon free



Heap Attacks: Buffer overflow

Heap attack:

Inter-chunk overflow



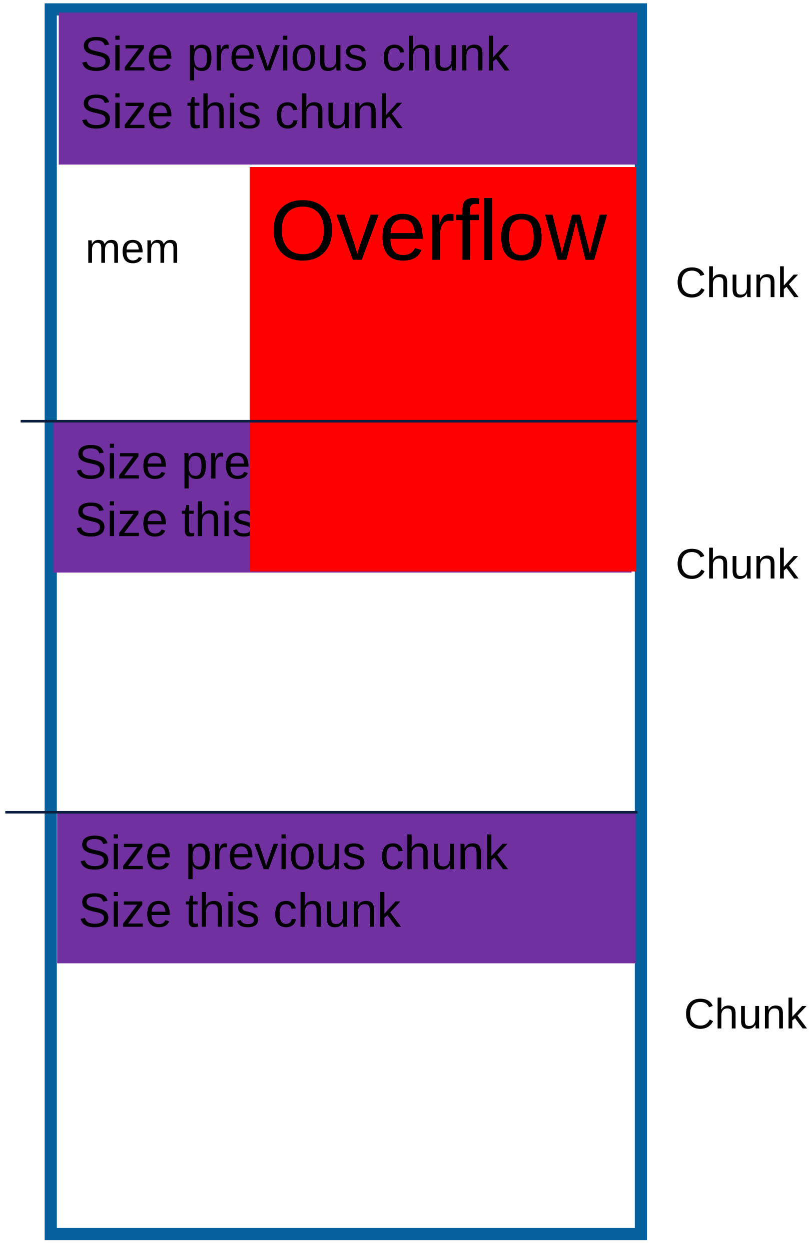
Heap Attacks: Buffer overflow

Heap attack:

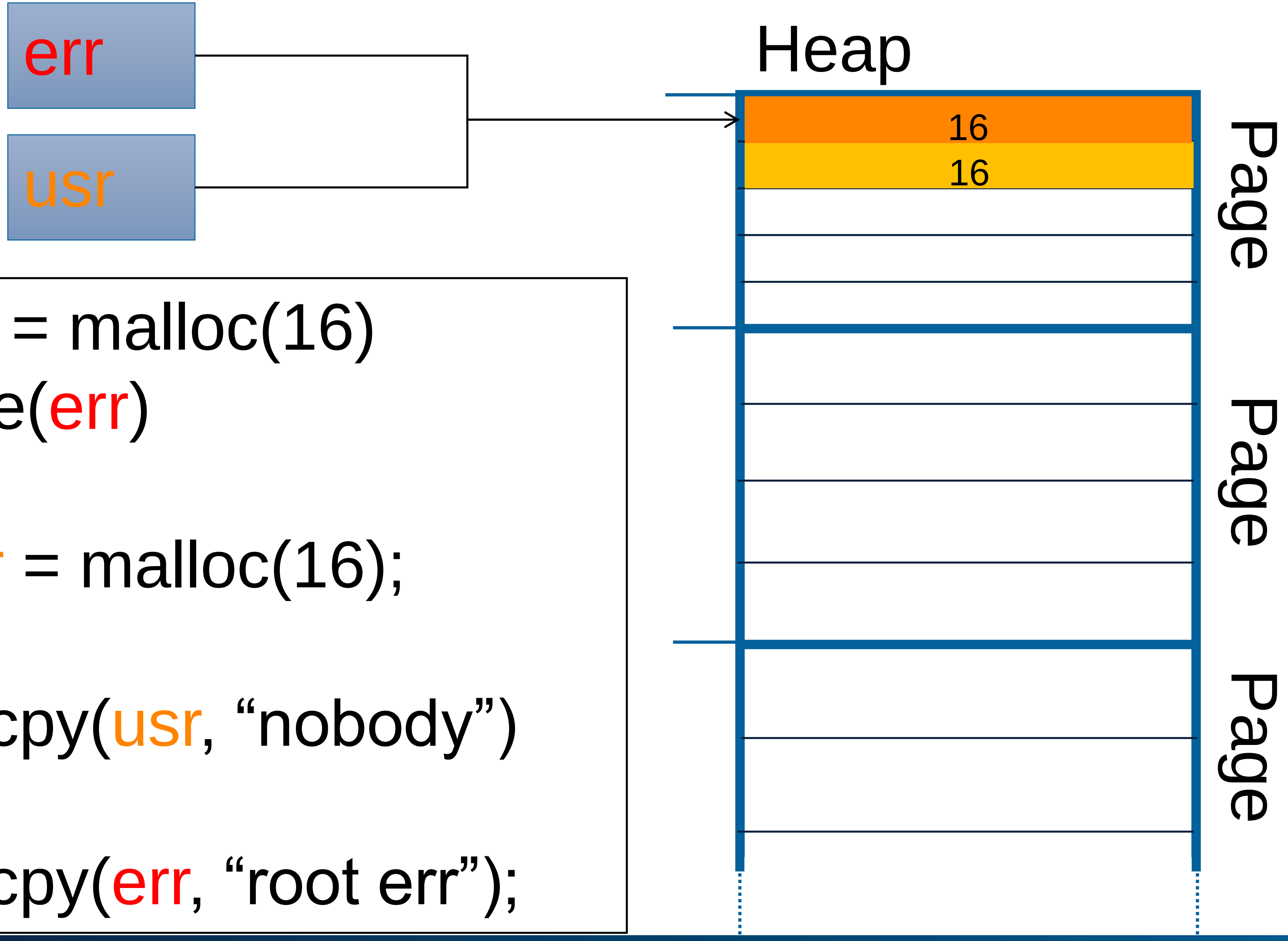
Inter-chunk overflow

Problem:

- In-band signaling (again)
- Can modify management data of heap allocator
- Therefore, can modify behavior of heap allocator
  - Create fake chunks
  - Ptmalloc2: Write what where upon free



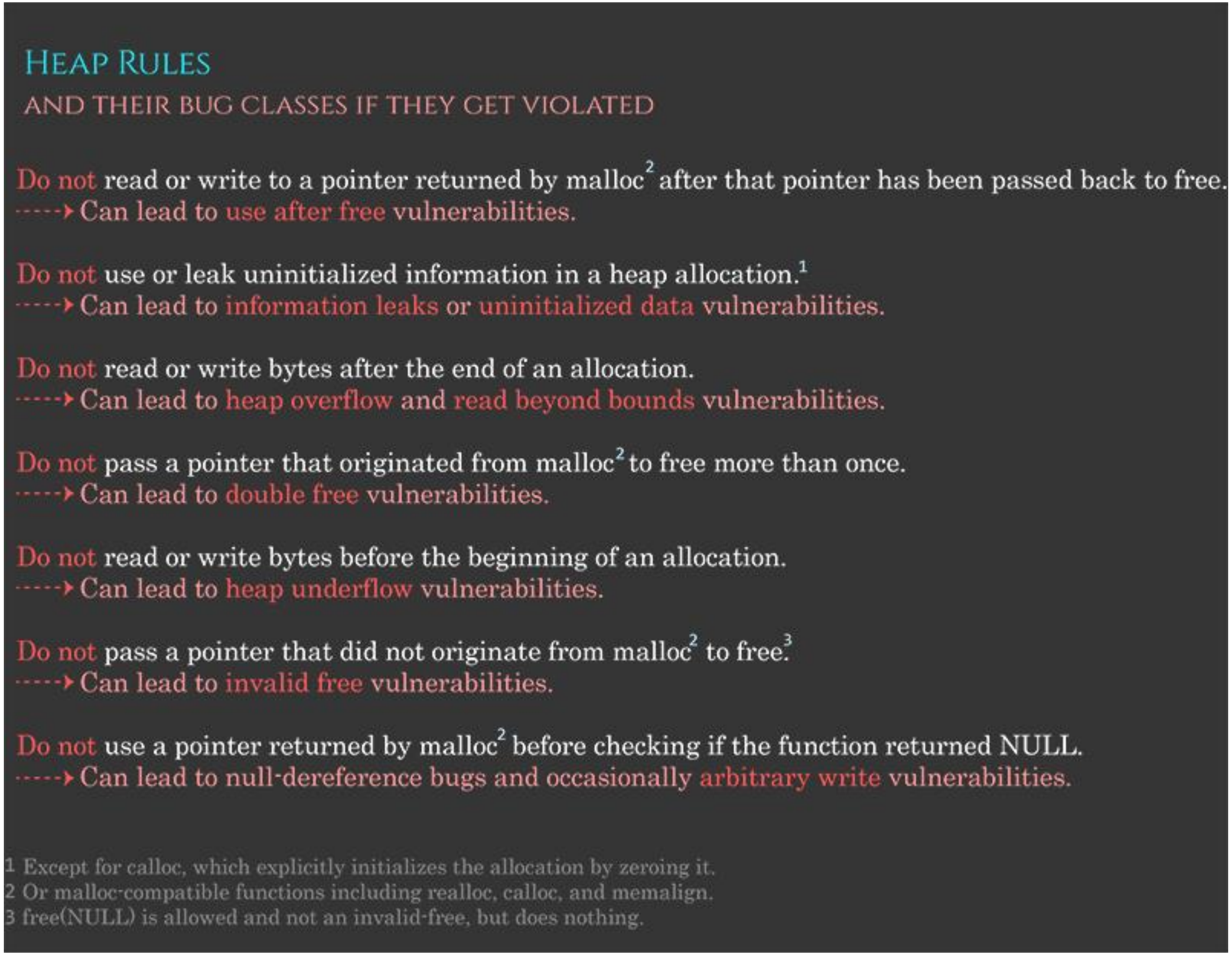
Heap Attacks: Use after free (UAF)



Use-after-free

```
err = malloc(16)
free(err)

usr = malloc(16);
…
strcpy(usr, “nobody”)
…
strcpy(err, “root err”);
```



Heap Rules

And their bug classes if they get violated

Do not read or write to a pointer returned by malloc[2] after that pointer has been passed back to free.
-----> Can lead to use after free vulnerabilities.

Do not use or leak uninitialized information in a heap allocation.[1]
-----> Can lead to information leaks or uninitialized data vulnerabilities.

Do not read or write bytes after the end of an allocation.
-----> Can lead to heap overflow and read beyond bounds vulnerabilities.

Do not pass a pointer that originated from malloc[2] to free more than once.
-----> Can lead to double free vulnerabilities.

Do not read or write bytes before the beginning of an allocation.
-----> Can lead to heap underflow vulnerabilities.

Do not pass a pointer that did not originate from malloc[2] to free.[3]
-----> Can lead to invalid free vulnerabilities.

Do not use a pointer returned by malloc[2] before checking if the function returned NULL.
-----> Can lead to null-dereference bugs and occasionally arbitrary write vulnerabilities.

[1] Except for calloc, which explicitly initializes the allocation by zeroing it.
[2] Or malloc-compatible functions including realloc, calloc, and memalign.
[3] free(NULL) is allowed and not an invalid-free, but does nothing.

https://azeria-labs.com/heap-exploitation-part-1-understanding-the-glibc-heap-implementation/

Heap Attacks

Recap:

- A buffer overflow on the heap can modify other buffers on the heap
- A buffer overflow on the heap can influence memory allocator management data structures (junks etc.)
  - and make it write some data to some memory address, in some cases

References

Resources:

- http://homes.soic.indiana.edu/yh33/Teaching/I433-2016/lec13-HeapAttacks.pdf
- http://www.pwntester.com/blog/2014/03/23/codegate-2k14-4stone-pwnable-300-write-up/