Processes

Abhilash Jindal

Agenda

- Vocabulary (OSTEP Ch. 4)
  - What is a process? System calls? Scheduler? Address space?
- Memory management (OSTEP Ch. 13-17)
  - How to manage and isolate memory? What are memory APIs? How are they implemented?
- Processes in action (xv6 Ch. 3: system calls, x86 protection, trap handlers)
  - Process control block, user stack<>kernel stack, sys call handling
- Scheduling (xv6 Ch5: context switching, OSTEP Ch. 6-9)
  - Response time, throughput, fairness

Process is a running program

- Load program from disk to memory
  - Exactly how we loaded OS
- Give control to the process. Jump cs, eip

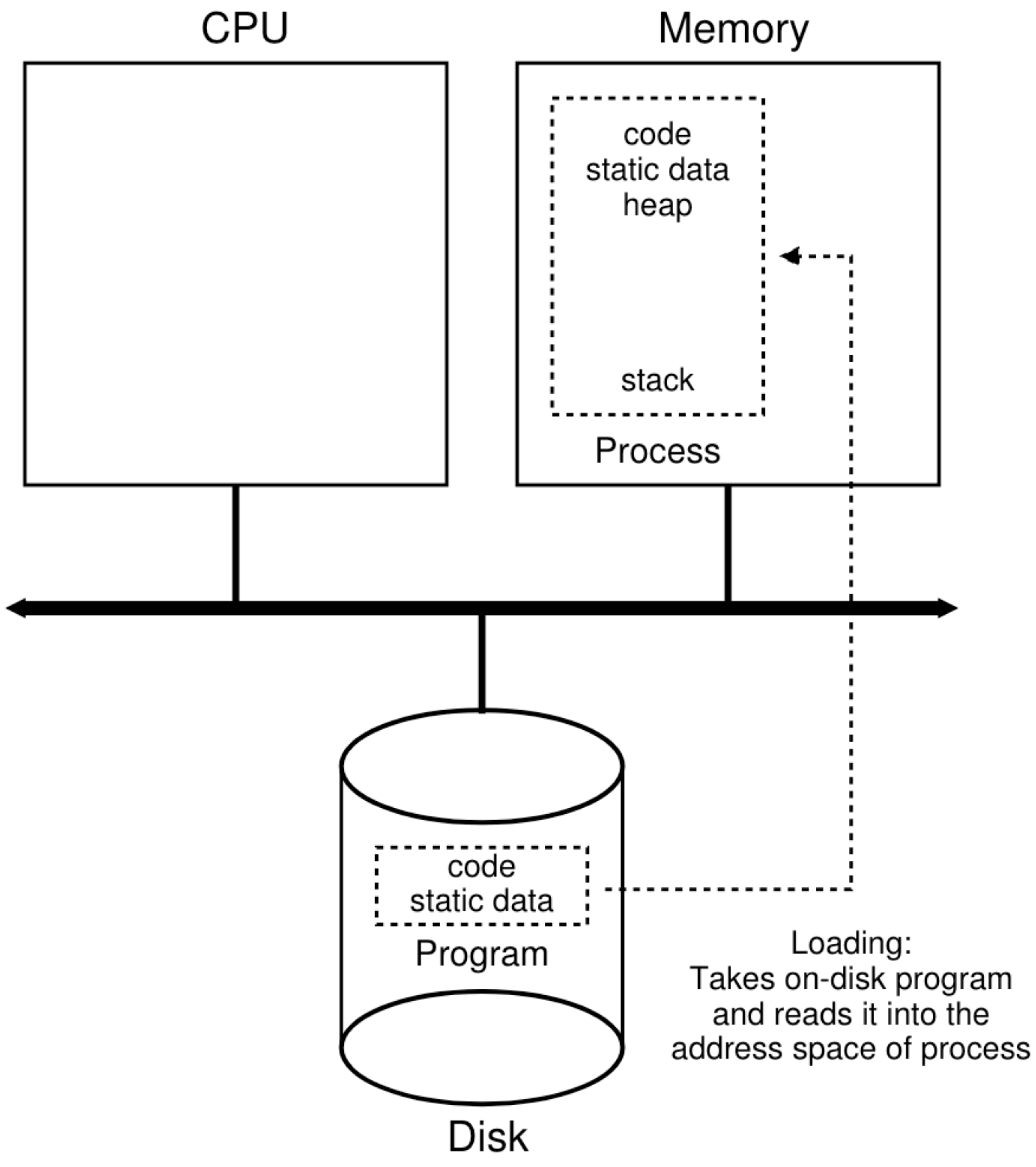


Figure 4.1: **Loading: From Program To Process**

Processes can ask OS to do work for them

System calls

```
$ strace cat /tmp/foo
…
openat(AT_FDCWD, "/tmp/foo", O_RDONLY)  = 3
read(3, "hi\n", 131072)                 = 3
write(1, "hi\n", 3)                     = 3
…
```

OS maintains process states

Scheduler switches between processes

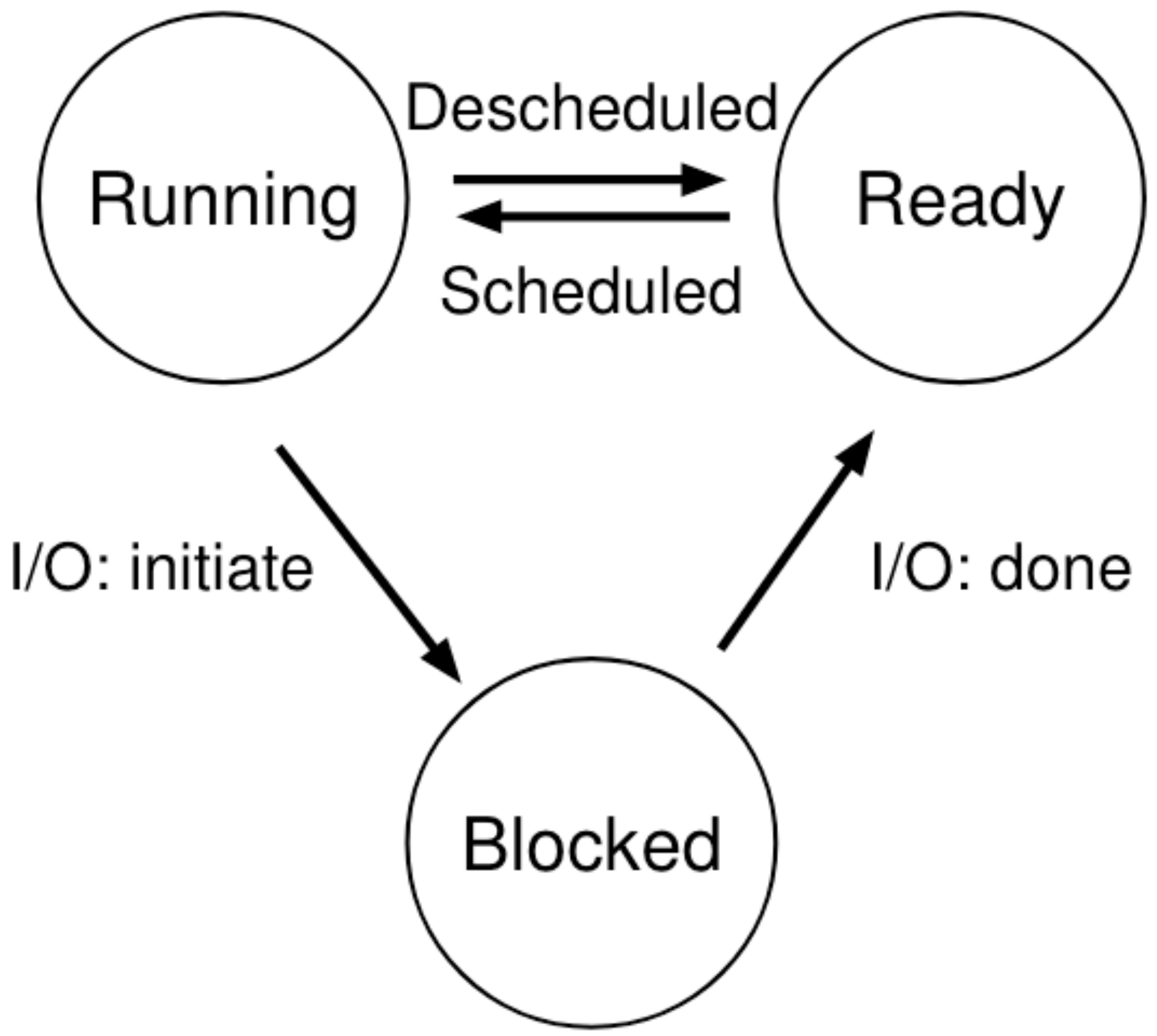
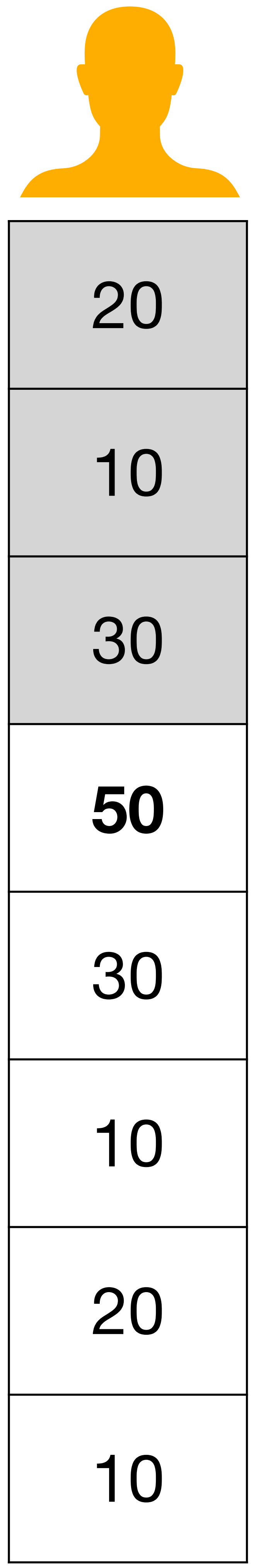


Figure 4.2: **Process: State Transitions**

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$ now done

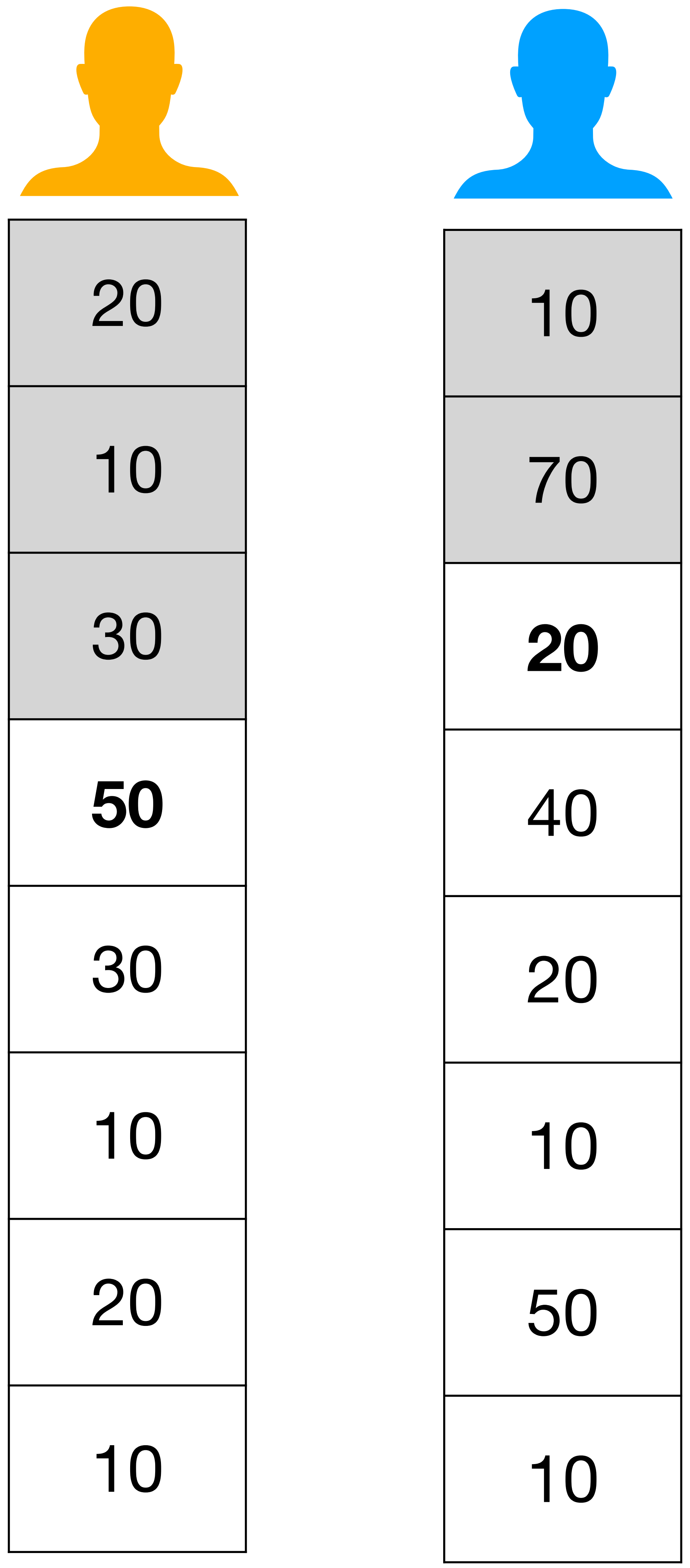
Figure 4.4: **Tracing Process State: CPU and I/O**

Calculator analogy: Computing long sum



- 2 0 = (move pointer to 10)
- \+ 1 0 = (move pointer to 30)
- \+ 3 0 = (move pointer to 50)
- \+ 5 0 = (move pointer to 30)
- \+ 3 0 = (move pointer to 10)
- \+ 1 0 = (move pointer to 20)
- \+ 2 0 = (move pointer to 10)

Sharing the calculator



- Steps to share the calculator:
    - 20 + 10 = 30 + 30 = 60
    - Write 60 in notebook, remember that we were done till 30, give calculator
    - 10 + 70 = 80
    - Write 80 in notebook, remember that we were done till 70, give the calculator back

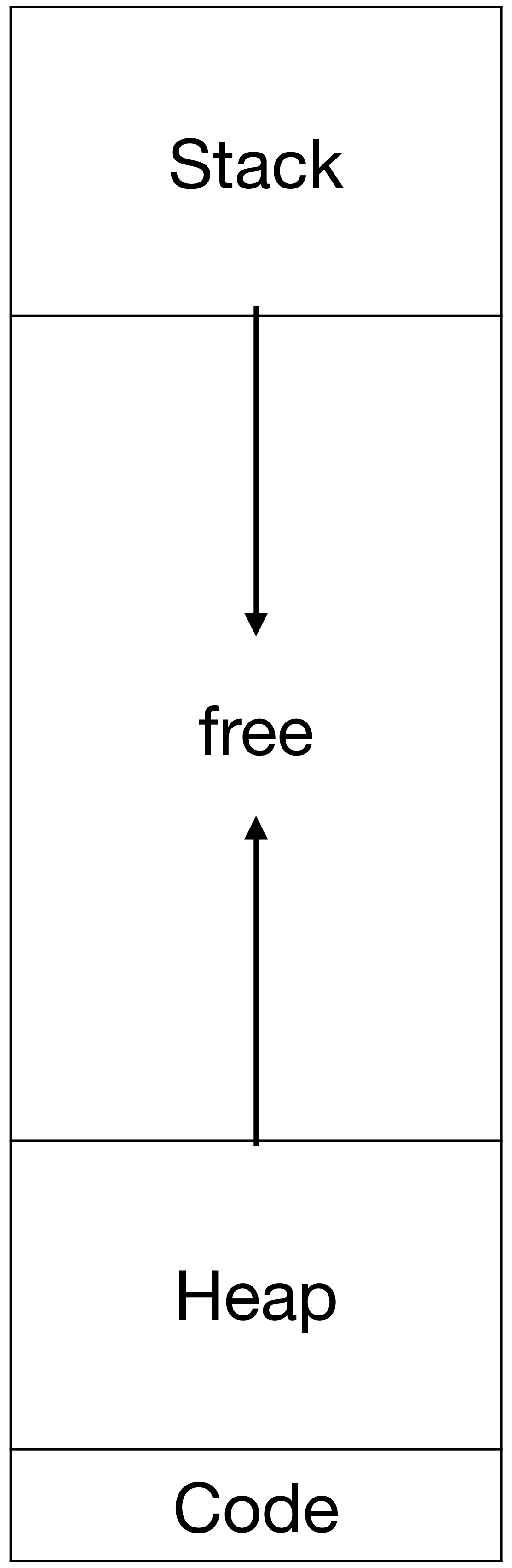
Memory isolation and management

OSTEP Ch. 13-17

Abhilash Jindal

Process Address Space

Code, Heap, Stack



Process address space

Function calling in action

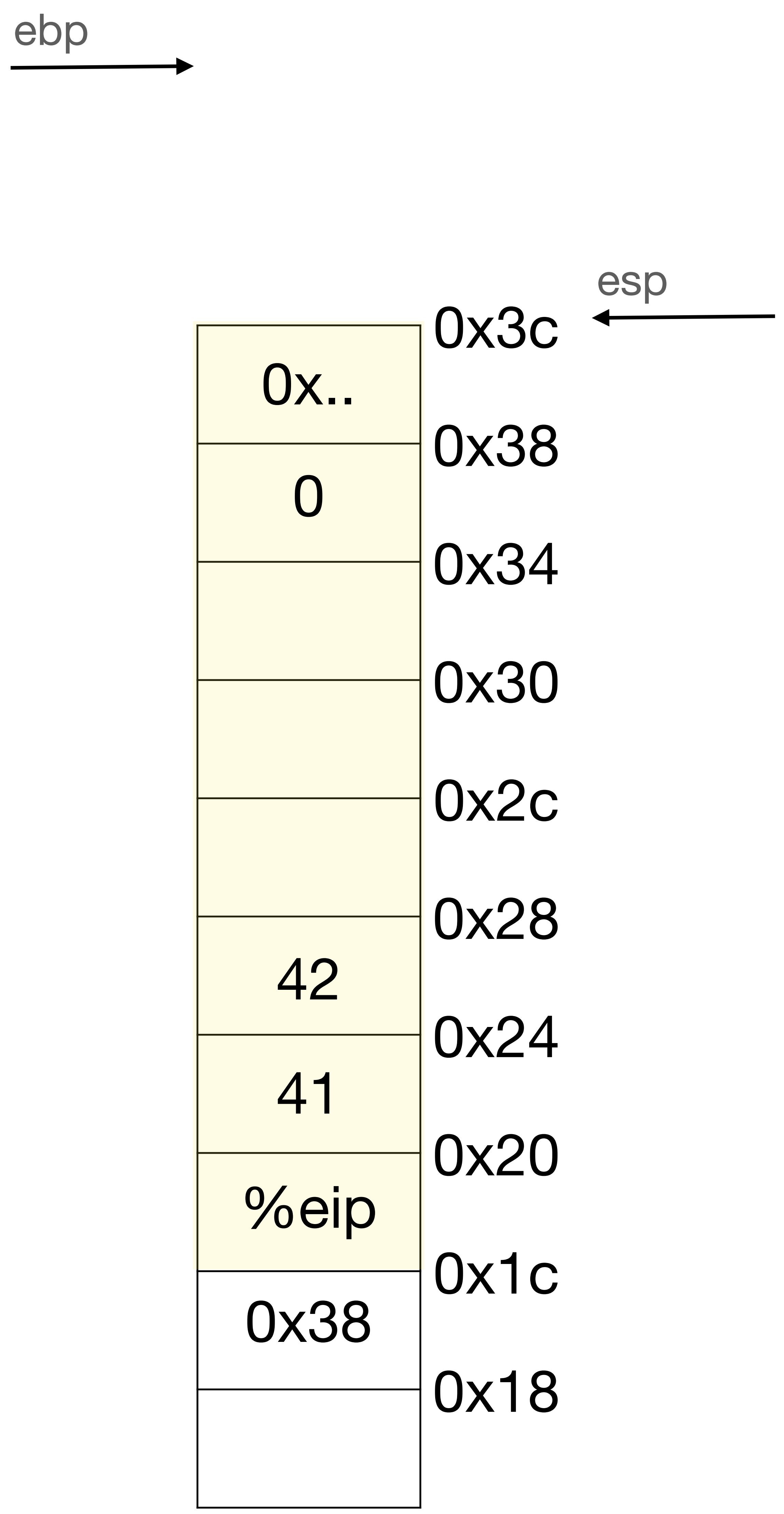
Stack

```
02.s

_foo:
  pushl %ebp                  Save caller’s base pointer
  movl  %esp, %ebp            ebp = esp
  movl  8(%ebp), %eax         eax = *(ebp + 8)
  addl  12(%ebp), %eax        eax = eax + *(ebp + 12)
  popl  %ebp                  Restore caller’s base pointer
  retl                        change eip to return address

  .globl  _main                            ## -- Begin function main
  .p2align  4, 0x90
_main:
  pushl %ebp                  Save caller’s base pointer
  movl  %esp, %ebp            ebp = esp
  subl  $24, %esp             esp = esp - 0x18
  movl  $0, -4(%ebp)          *(ebp-4)=0
  movl  $41, (%esp)           *(esp) = 41
  movl  $42, 4(%esp)          *(esp+4) = 42
  calll _foo                  Push current eip on to stack, jump to foo
  addl  $24, %esp             esp = esp + 24 (Restore caller’s esp)
  popl  %ebp                  Restore caller’s ebp
  retl
```

eip

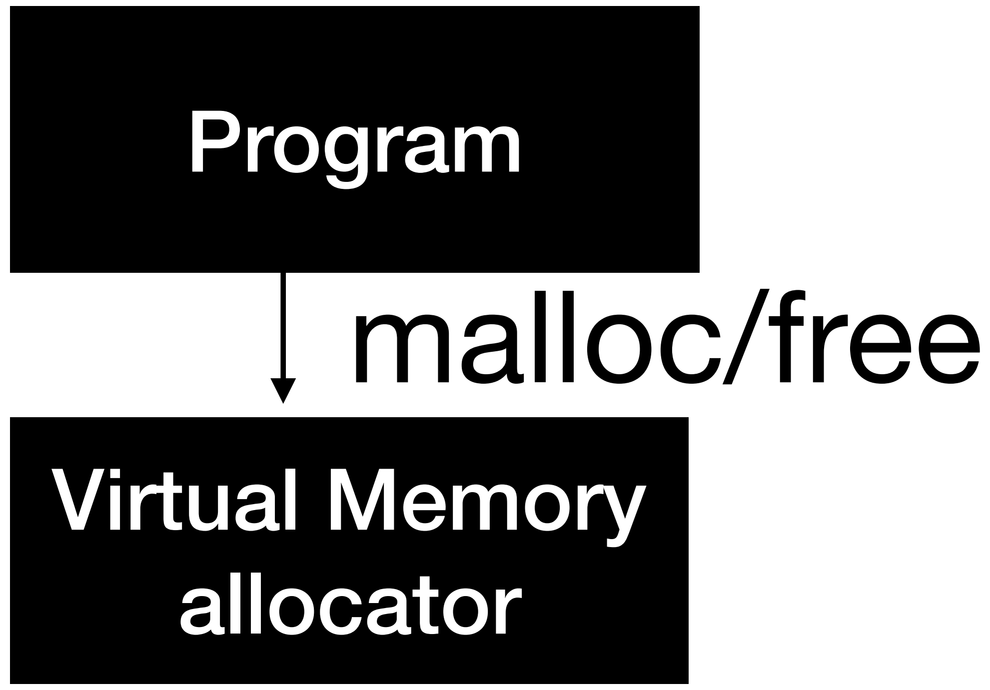


Memory APIs and bugs

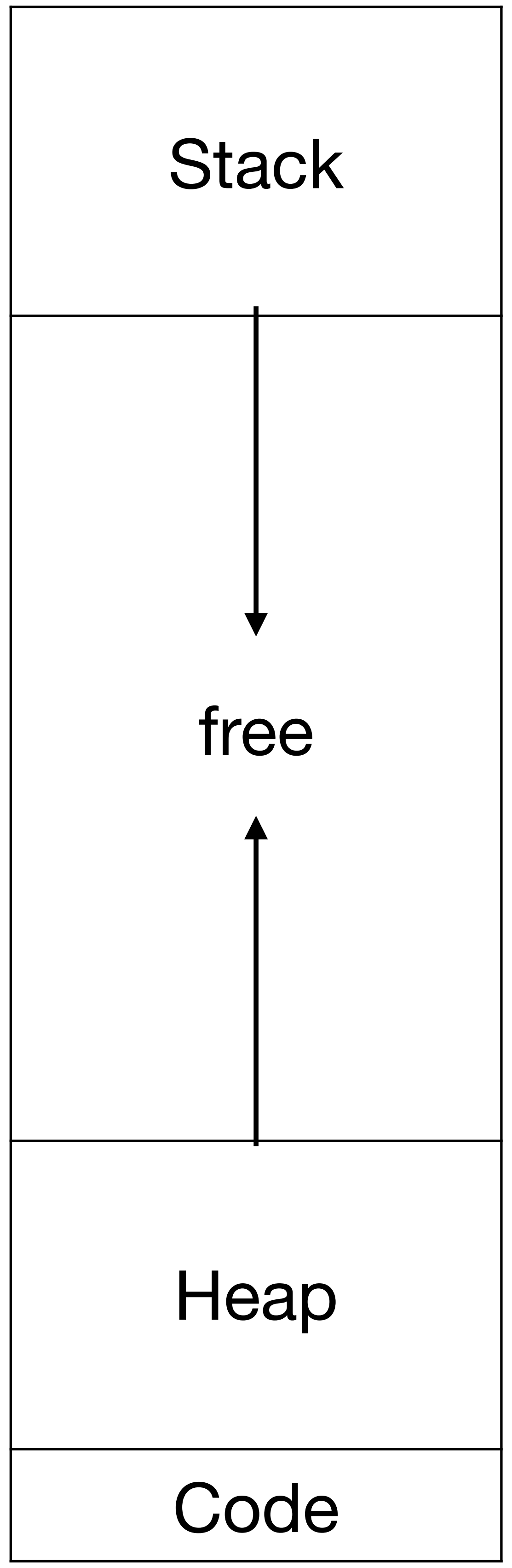
- malloc, free. va.c.
  - malloc is for dynamic allocation. Size is not known at compile time. Slower than stack allocations. Need to find free space.
- Null pointer dereference. null.c
- Memory leak. leak.c
- Buffer overflow. overflow.c
- Use after free. useafterfree.c
- Invalid free. invalidfree.c
- Double free. doublefree.c
- Uninitialised read. uninitread.c

Memory allocator

Works with virtual memory



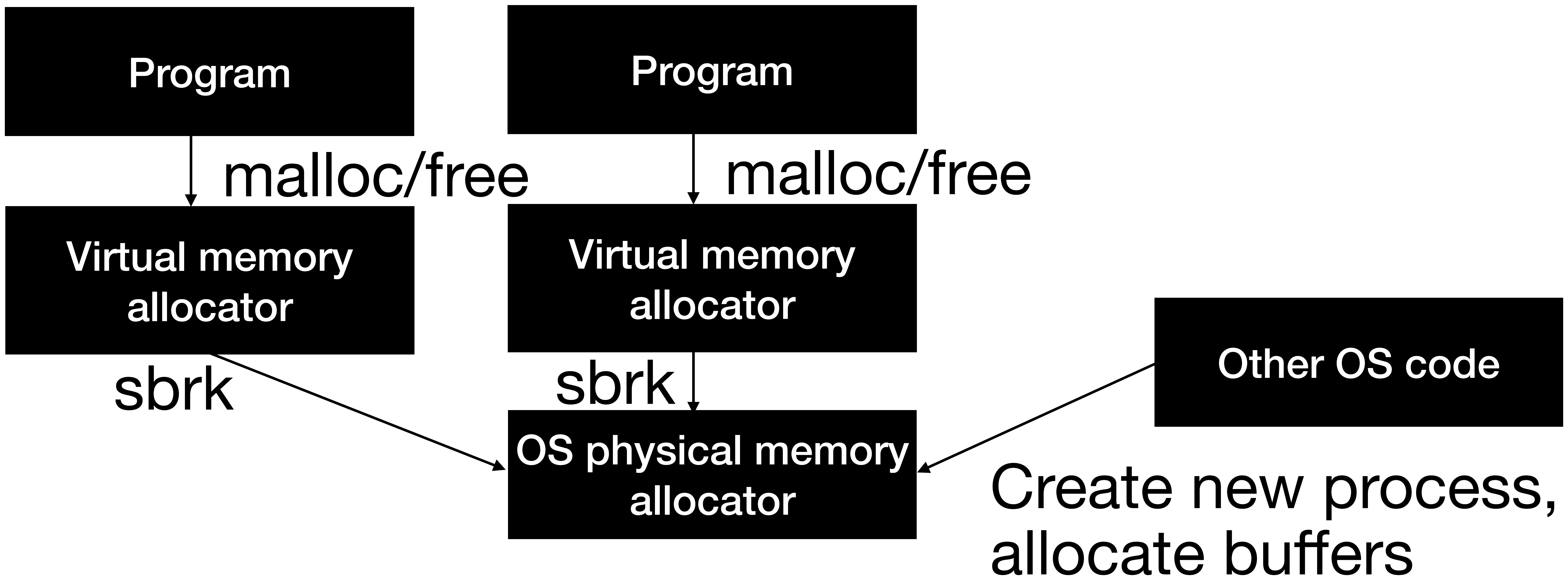
Manages heap memory

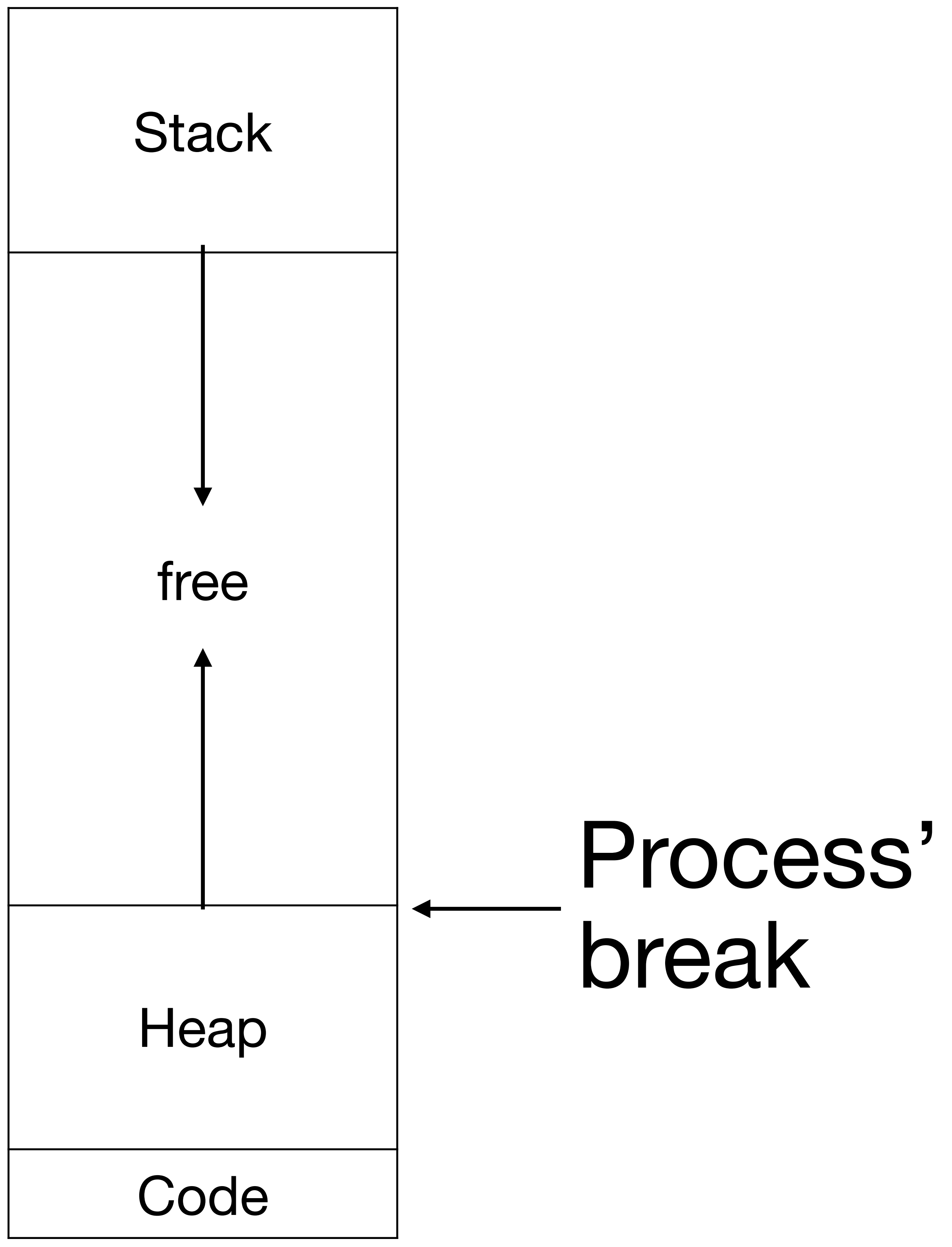


Process address space

OS memory allocator

- sbrk(int *increment*) increments process' break. *increment* can be negative.





Process address space

Memory allocation

```
ptr=malloc(size_t size);

free(ptr);
```

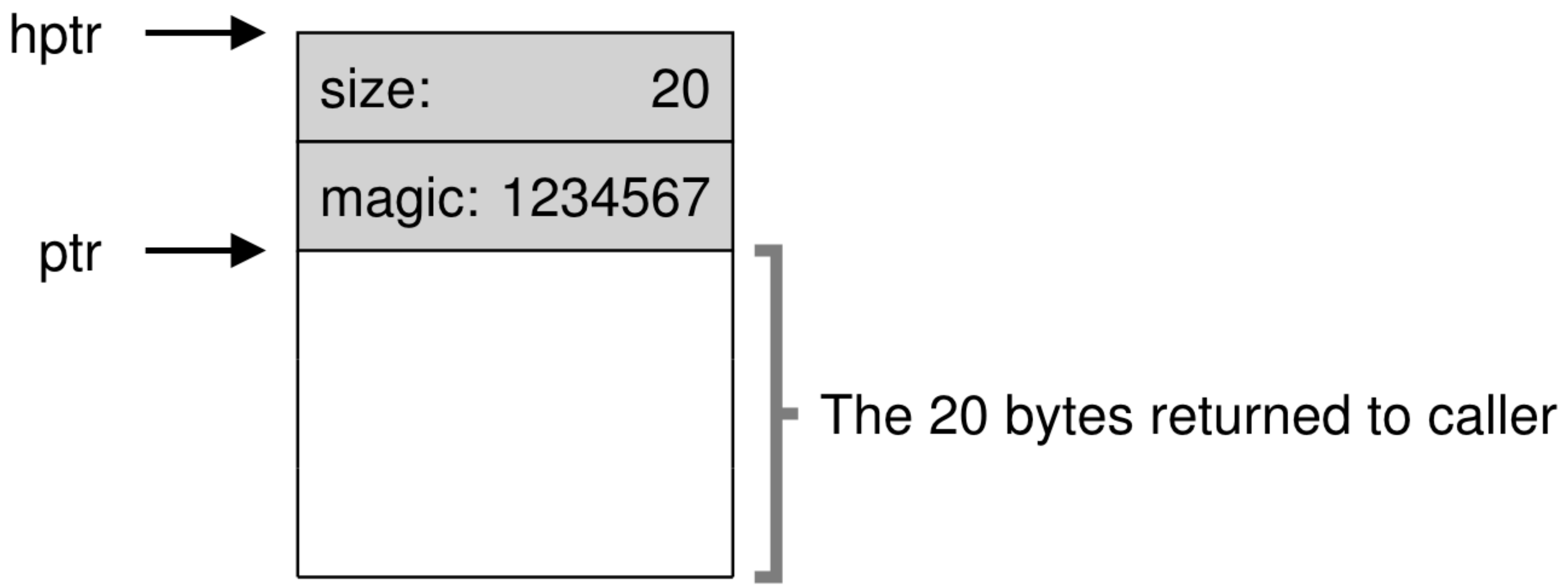
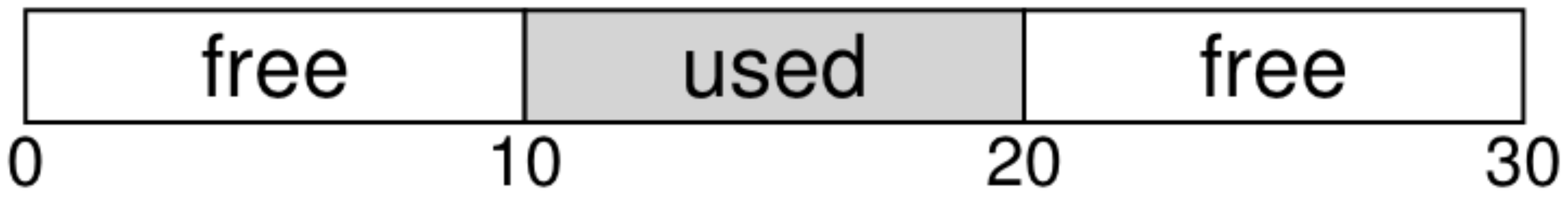


Figure 17.2: **Specific Contents Of The Header**

Stack allocations are faster than heap allocations. No need to find space.

Memory allocator



Fragmented heap over time

- Assumptions
  - Do not apriori know allocation size and order
  - Cannot move memory once it is allocated. Program might have the pointer to it.
- Goals
  - Quickly satisfy variable-sized memory allocation requests. How to track free memory?
  - Minimize fragmentation

Memory (de)allocation patterns

- Small mallocs can be frequent. Large mallocs are usually infrequent.
  - After malloc, program will initialise the memory area.
- “Clustered deaths”: Objects allocated together die together.

Free list splitting and coalescing

```
ptr = malloc(100)

sptr = malloc(100)

optr = malloc(100)

free(sptr)

free(ptr)

free(optr)
```

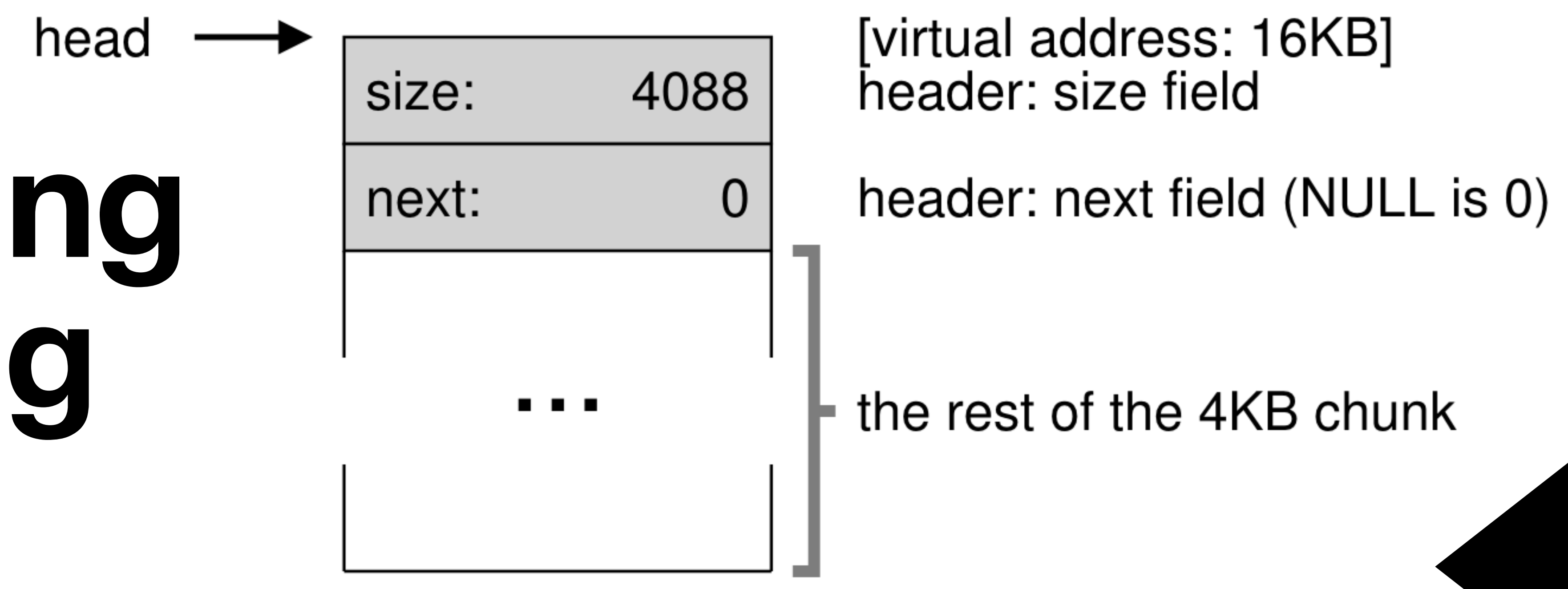


Figure 17.3: **A Heap With One Free Chunk**

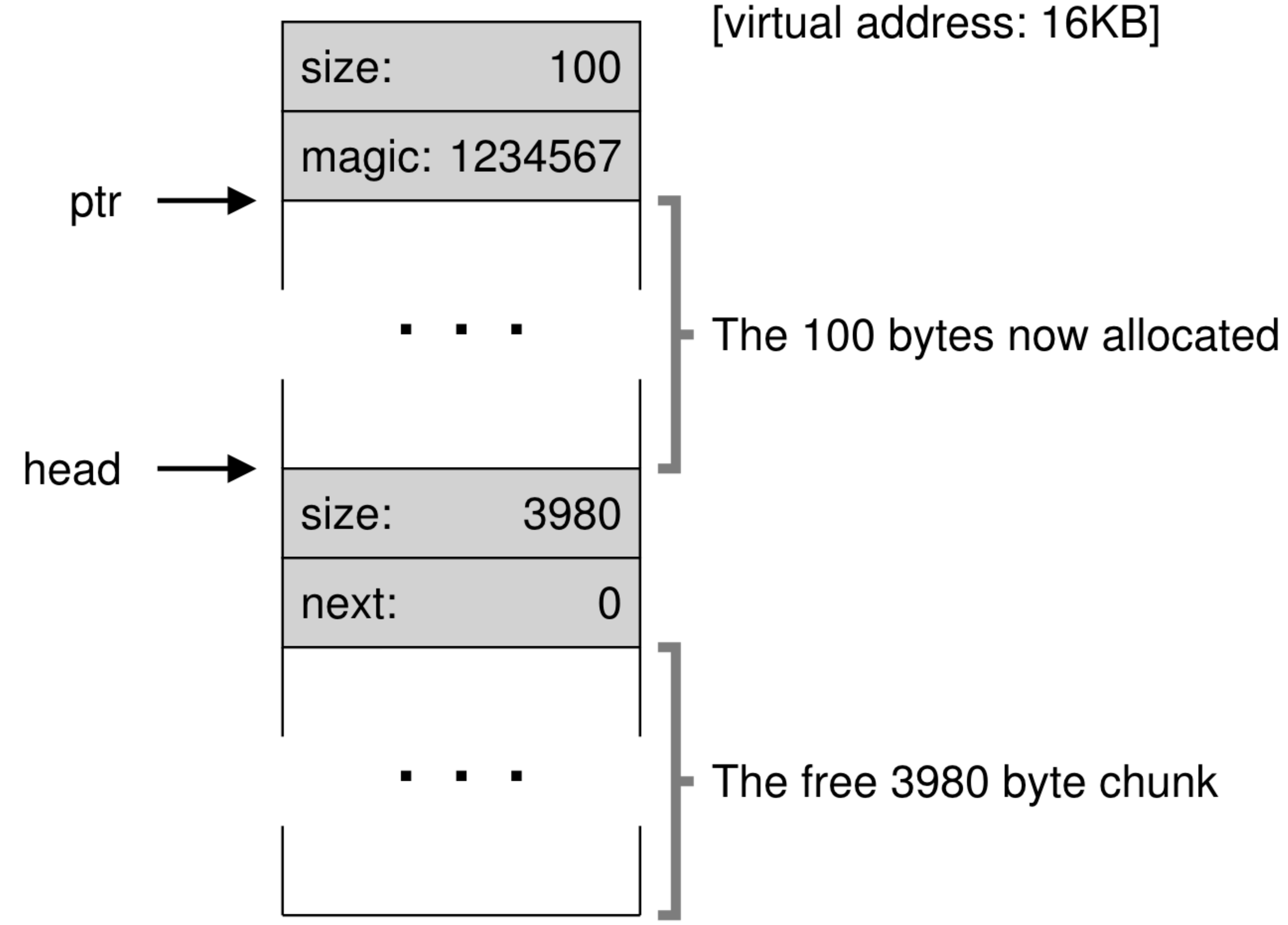


Figure 17.4: **A Heap: After One Allocation**

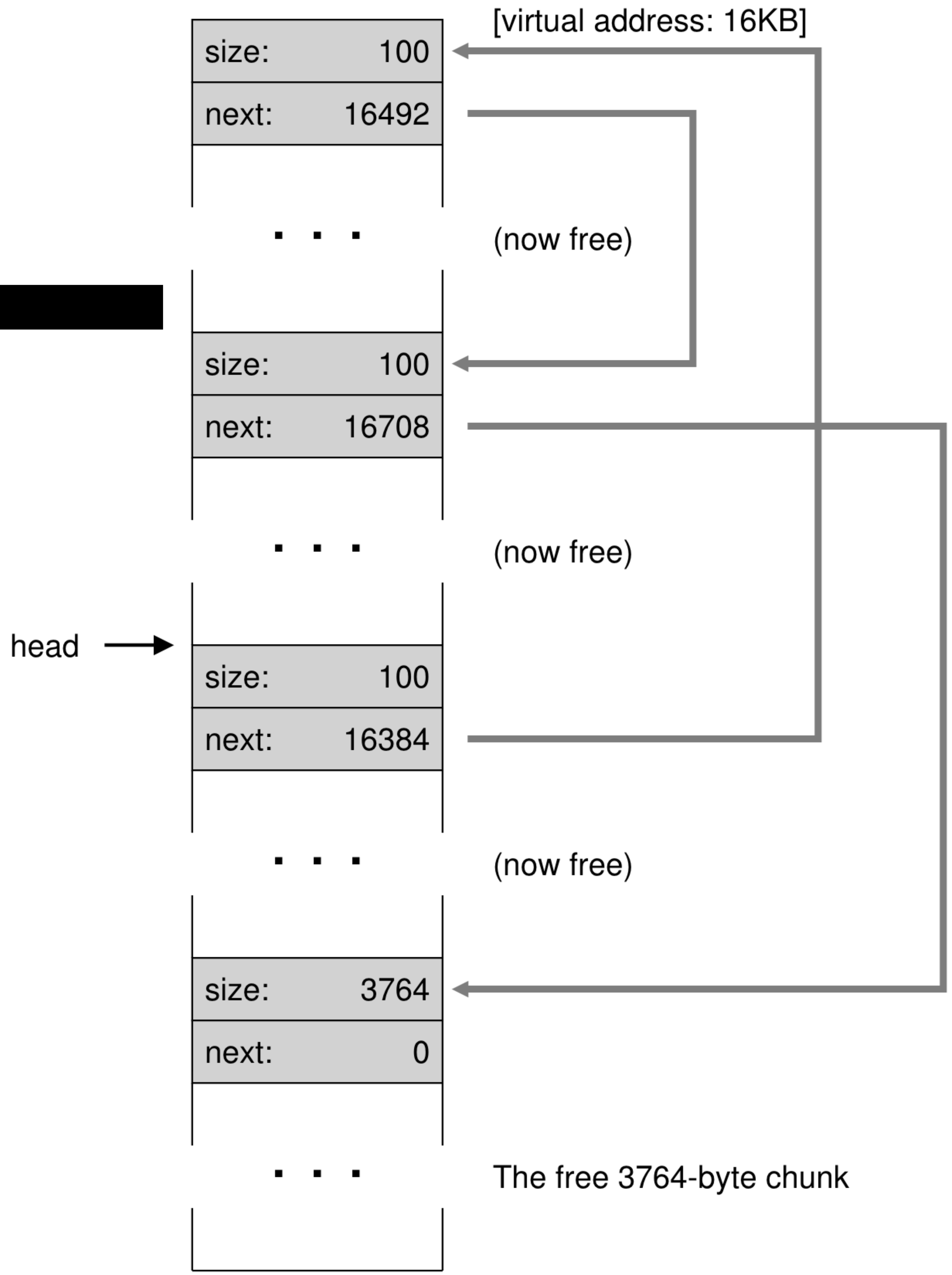
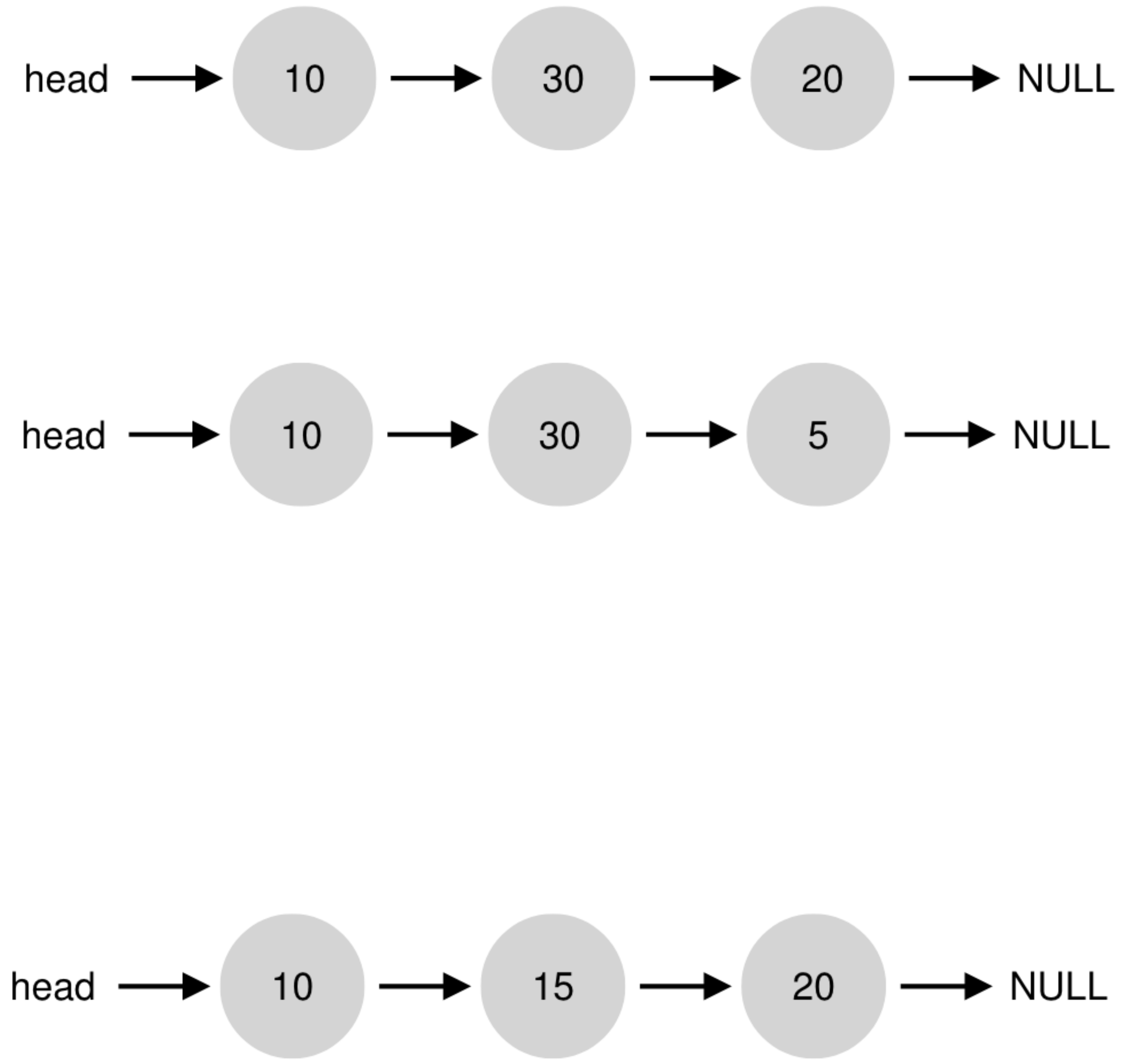


Figure 17.7: **A Non-Coalesced Free List**

Which block to allocate?

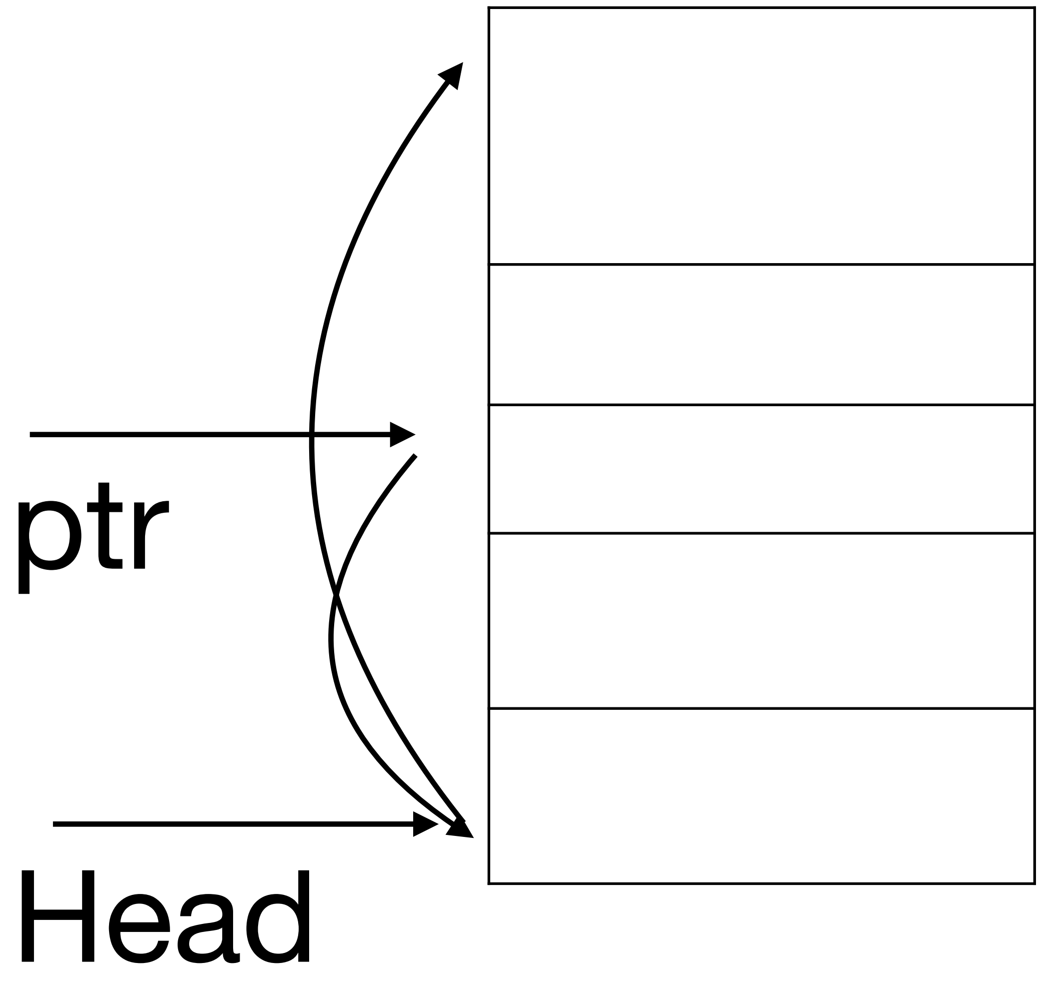
Example: malloc(15)

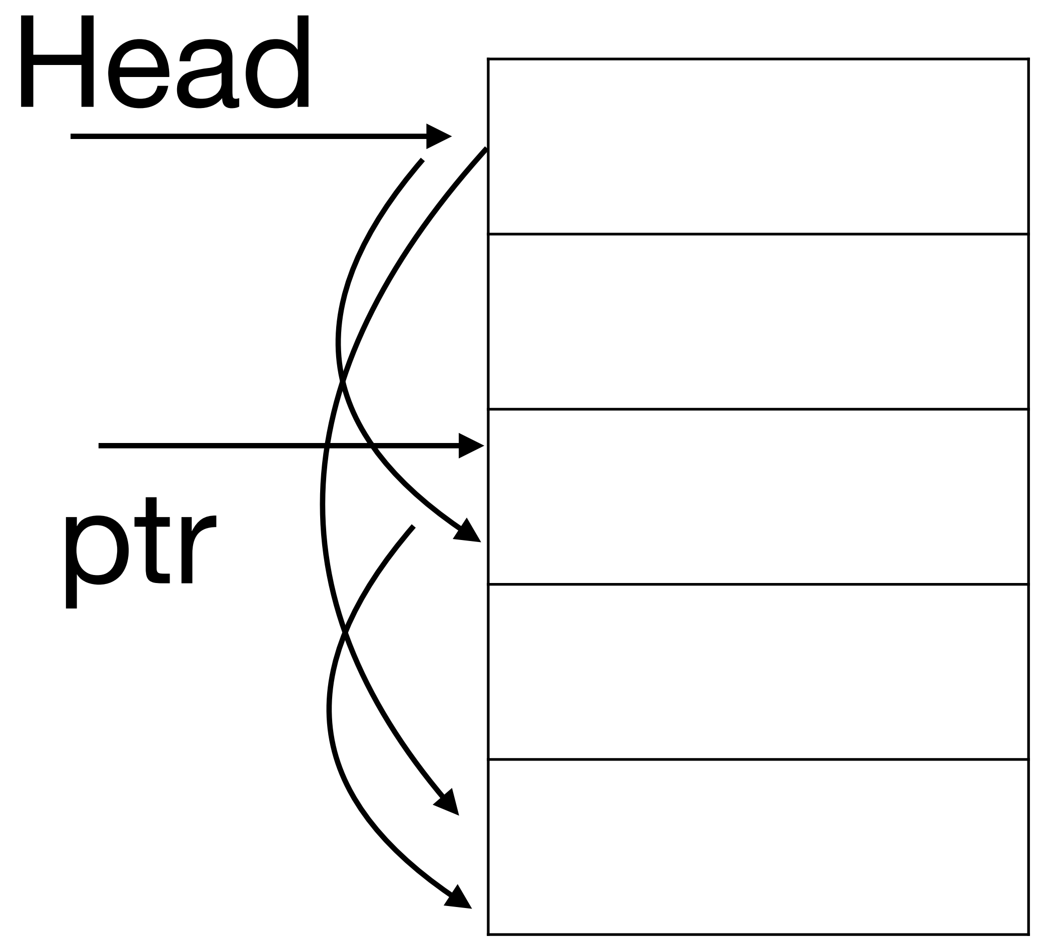


- Best fit
  - Slow. need to search the whole list
- First fit
  - Faster. (xv6: umalloc.c)
- Fragmentation
  - Example: malloc(25)

In which order to maintain lists?

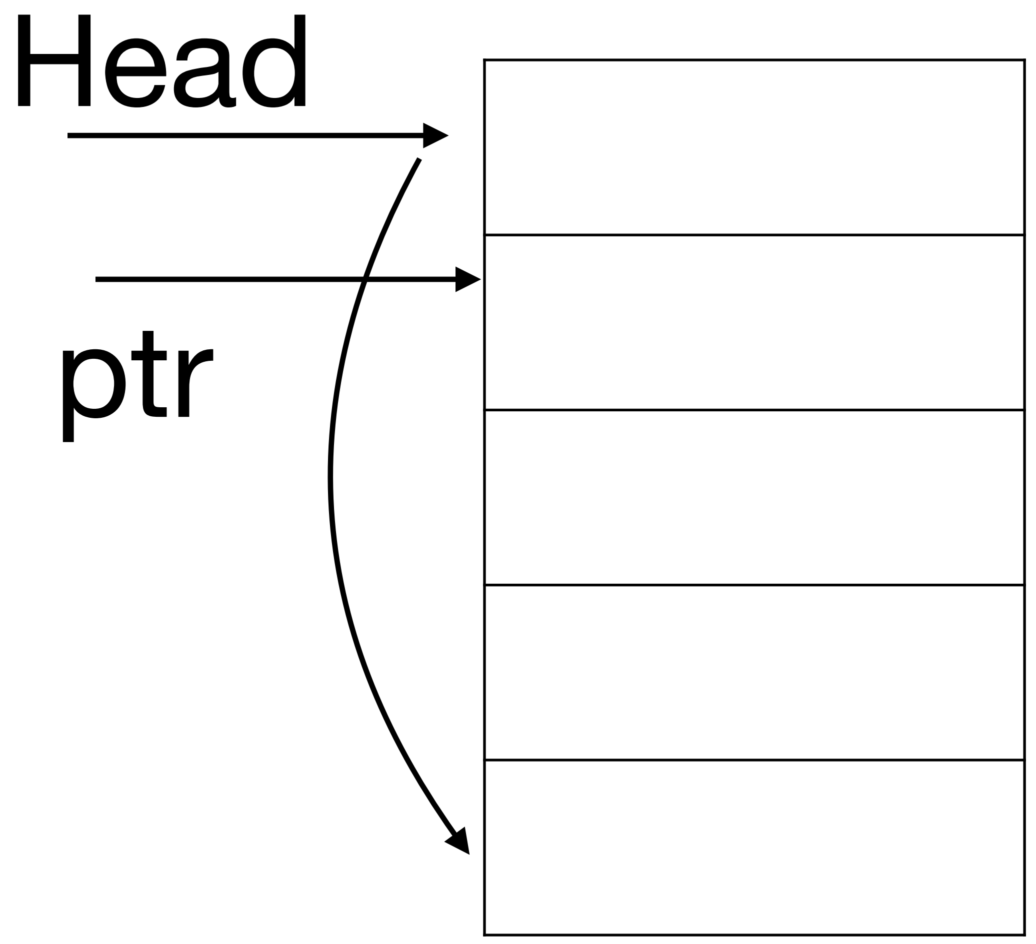
- (De)allocation order
- Address order
  - Slow frees: need to traverse the free list
  - (xv6: umalloc.c)
  - Address order, first fit will allocate back-to-back allocations contiguously.
    - Due to “clustered deaths”, we may get better chances of coalescing





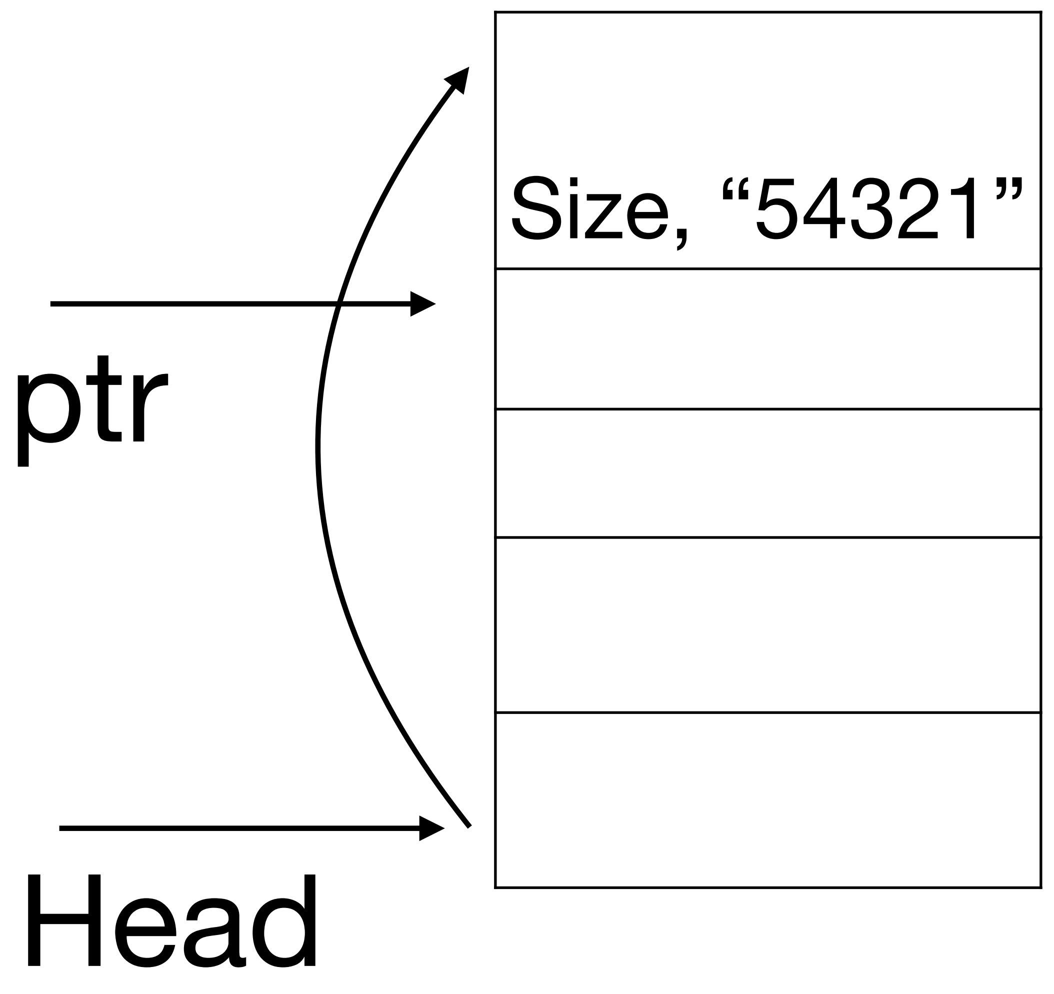
How to do coalescing?

Example: free(ptr)



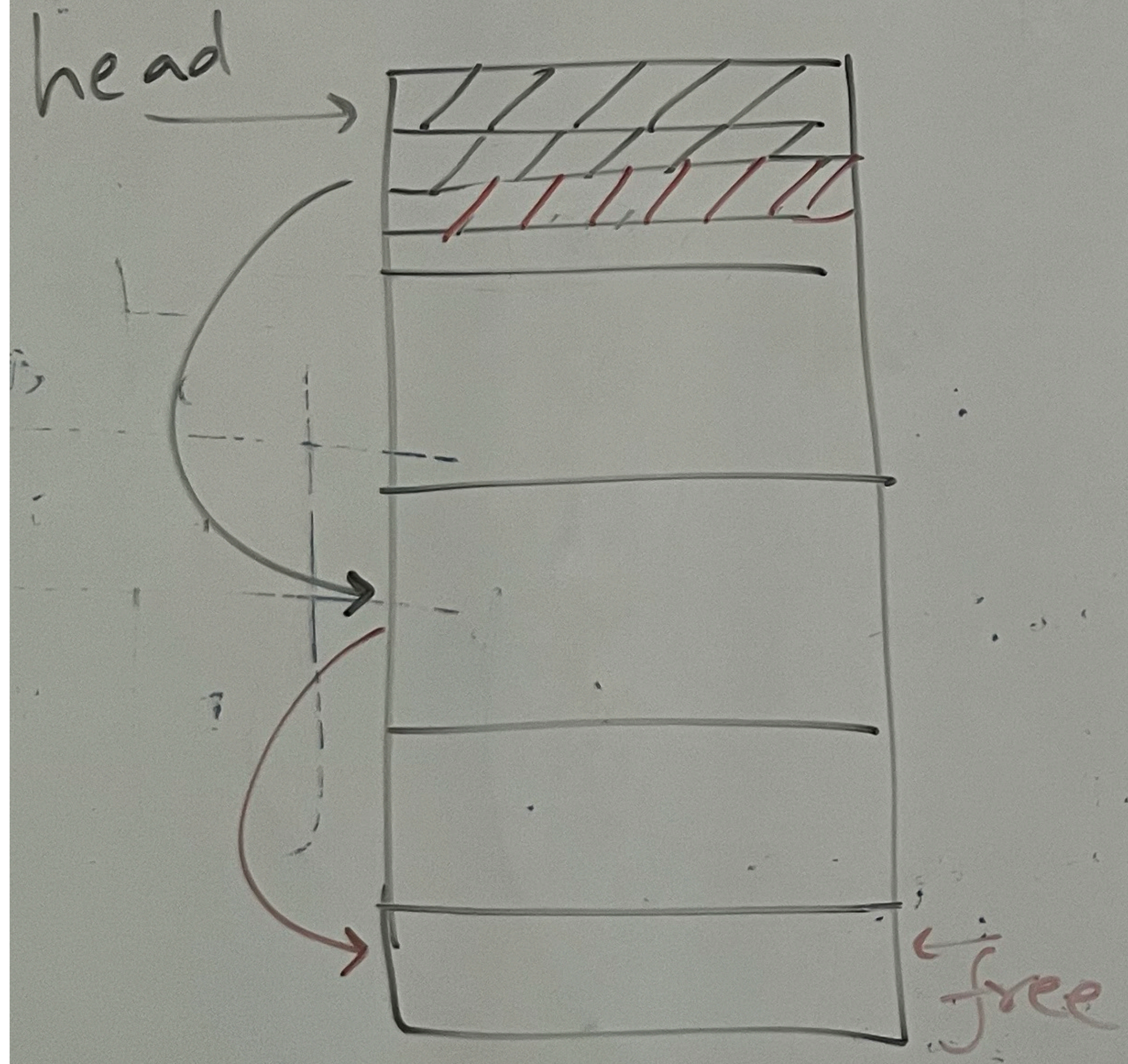
- Straightforward in address order since we are traversing the free list in address order

- In deallocation order: when an area is freed, check if the “boundary tag” is present in the footer above

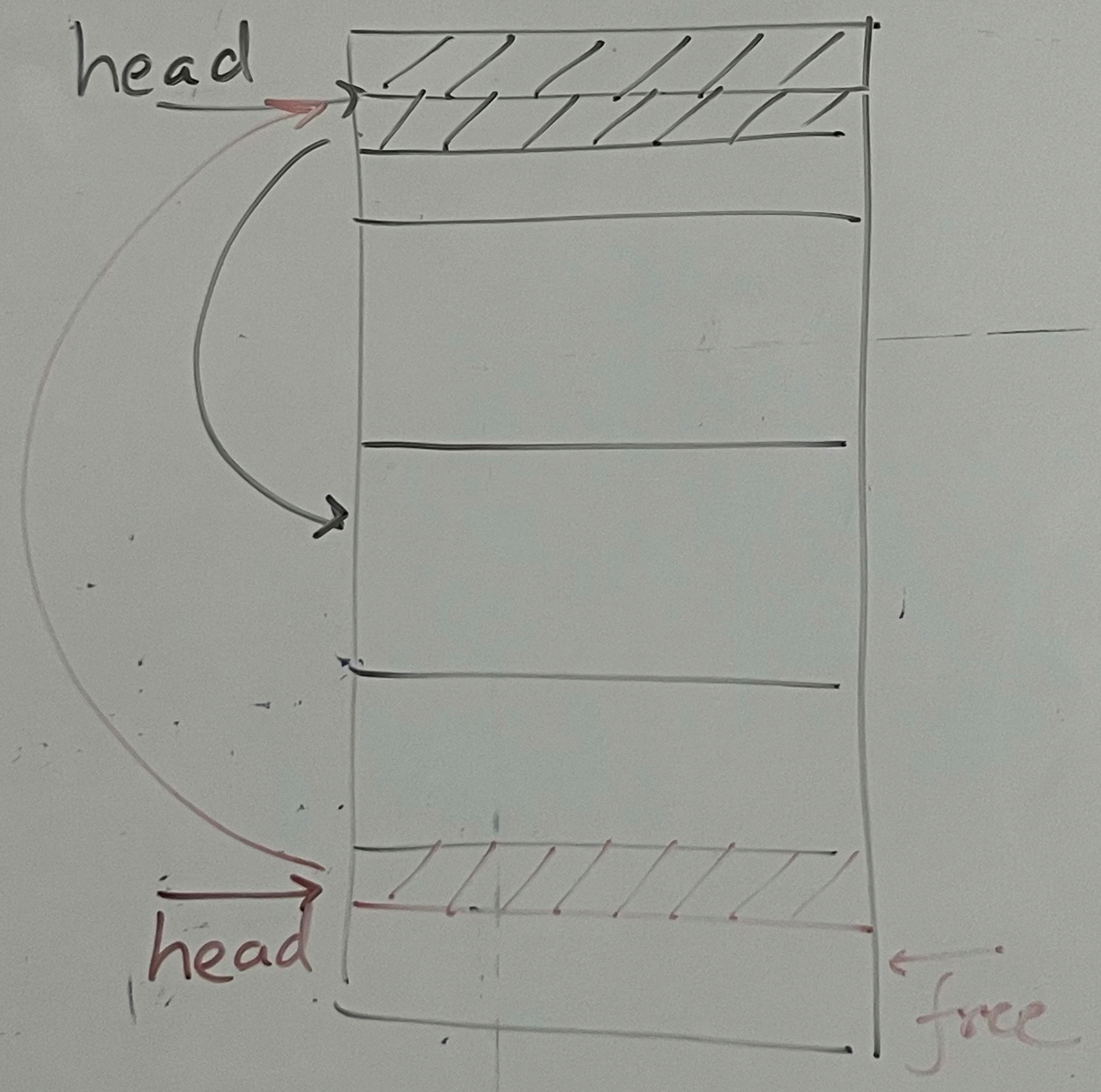


- First fit and address order work well in clustered deaths, simplify coalescing (no boundary tag), but cause fragmentation and traverse free list at free calls

First fit
address order



First fit
(de) allocation order



Boundary tag