Dynamic Allocation

Dynamic Memory Allocation

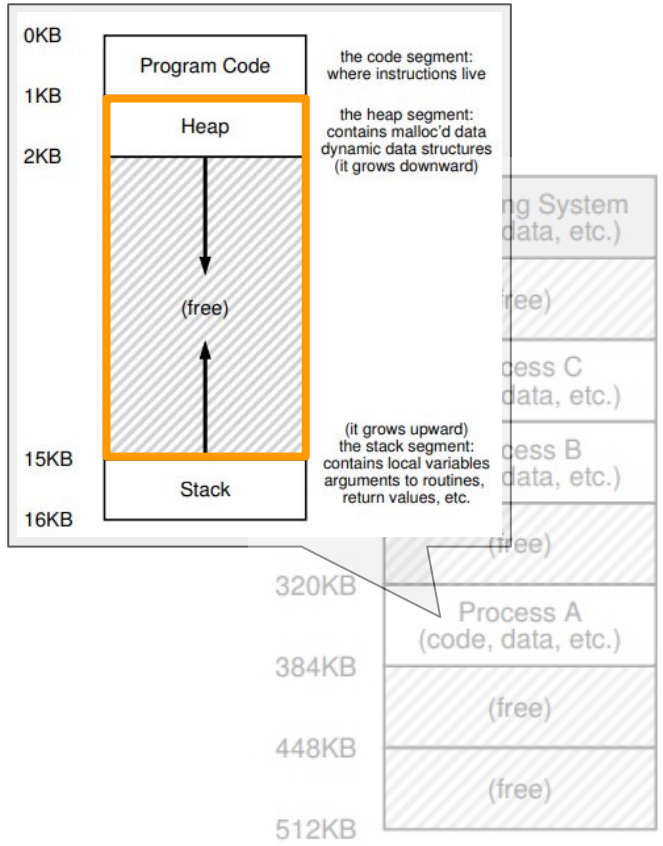
- Programmers use Dynamic Memory Allocation (malloc is an example) to request virtual memory during run-time.
  - We do this at runtime for structures, for which we cannot allocate at compile-time (e.g. a linked list)
    - Other languages like C++ or Java use ‘new’ to allocate memory
    - (Scripting languages do this behind the scenes for us)

```
1  #include <stdlib.h>

2  int main(){

3      int* array = (int*)malloc(500);

4      for(int i= 0; i < 500; ++i){
5          array[i] = i;
6      }

7      free(array);

8      return 0;
9  }
```

Note: We could allocate an array of ‘500’ integers at compile-time, this is just an example.

Dynamic Memory Allocation

- Dynamic memory allocators manage virtual memory in what is known as the *heap*
  - Each process has a heap segment of memory that can resize.
  - The heap consists of variable sized blocks
    - Each block can be marked as either:
      - allocated
      - free



Explicit vs Implicit Memory Allocation

- Explicit Allocator
  - The application(program) allocates and frees space
    - This is what we do in C with malloc and free
    - (or equivalently in C++ with *new* and *delete*)
- Implicit Memory Allocation
  - The application allocates, but does not free space
    - A **garbage collector** instead frees the memory for us.
    - e.g. The Java programming language has different garbage collectors to help us

Our focus in this class (since we use C) is to look at an explicit allocator

Malloc | Found in #stdlib.h or #malloc.h

- void* malloc(size_t size)
  - On Success
    - Returns a pointer to a memory block of at least size bytes
      - For x86 this memory is aligned to 8-byte boundaries
      - For x86-64 this memory is aligned to 16-byte boundaries
    - A size of 0 returns NULL
  - Unsuccessful allocation: returns NULL(0)
- void free(void *p)
  - Returns the block pointed at by p to pool of available memory
  - p must come from previous call to malloc (or realloc)
    - Note: Never use free with **delete** (used in C++), these are different allocators!

Other C Memory functions

C Library

- calloc
  - Similar to malloc, but initializes the allocated block to zero.
- realloc
  - Changes size of previously allocated block
- free
  - Reclaims memory allocated by malloc, calloc, or realloc.

System Calls

- sbrk
  - Use internally by allocators to grow or shrink the heap
  - This will be handy for implementing our own memory allocator!
- mmap
  - Creates a new mapping in virtual address space of calling process.

Malloc example

- Here is another annotated example of what is going on when we malloc properly.

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

int main(){

        // Allocate a block with 50 ints
        int* data = (int*)malloc(50*sizeof(int));

        // If our allocation fails, we get NULL back
        if(data==NULL){
                exit(0);
        }

        // Initialize allocated block
        int i;
        for(i=0; i < 50; ++i){
                data[i]=i;
        }

        // Return allocated block of size '50' to the heap
        free(data);

        return 0;
}
```

Malloc Performance

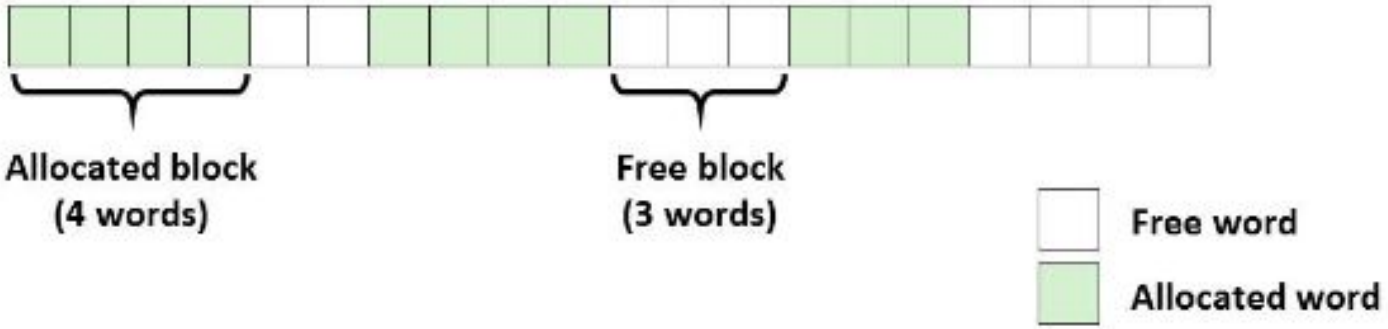
- Why is allocation expensive?
  ?

Malloc Performance

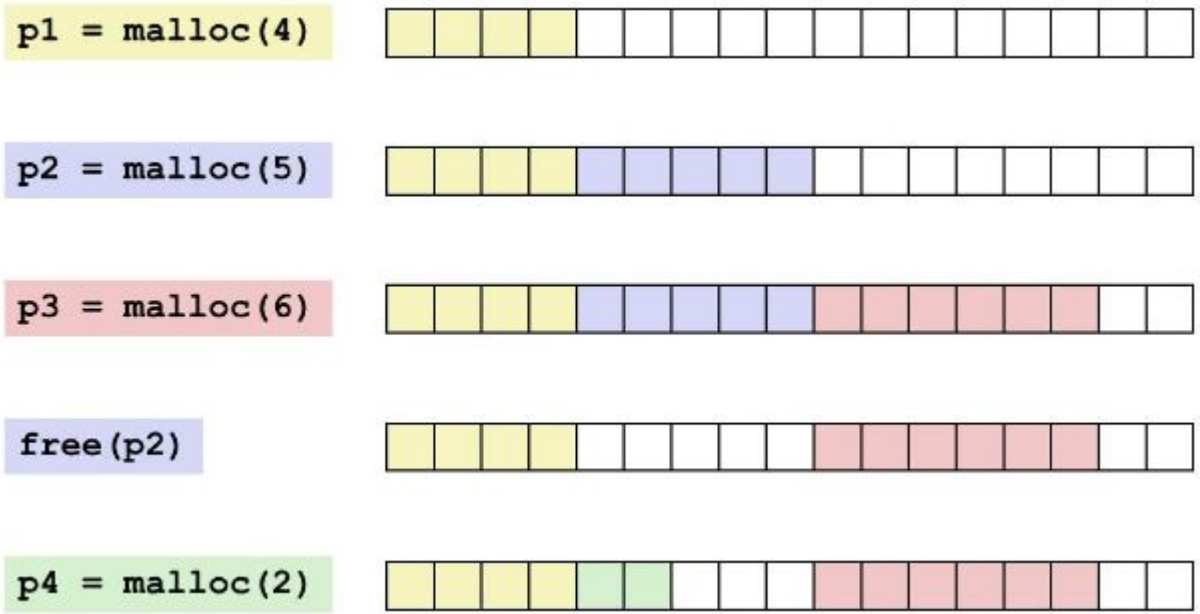
- Why is allocation expensive?
- A few answers:
  - One is that we have to find (perform some search) a block of memory that ‘fits’ in our heap.
    - If it does not fit then we have to make a system call to sbrk to extend our heap and find a block that fits
    - Anytime we have to call sbrk, that is a syscall
      - syscalls are expensive
      - They involve context switching (saving our registers state, moving to kernel mode, then swapping back)

Some ways to think about memory.

- You have some assortment of blocks
- Some are allocated, and some are free
  - (Assume 1 block here is 1 word or 32 bytes)



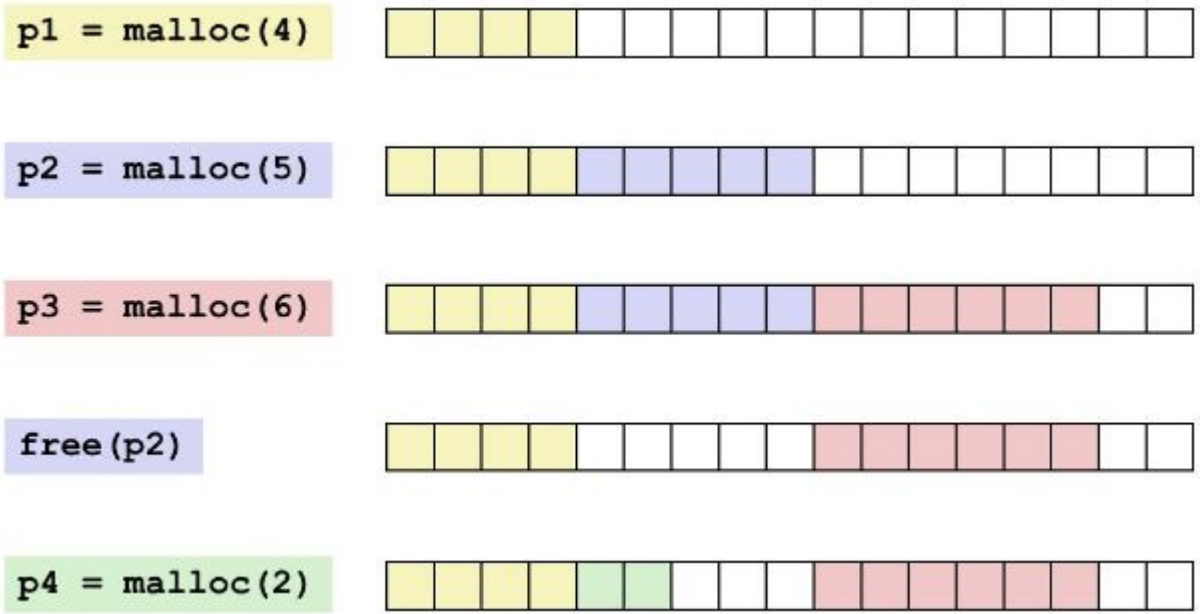
Allocation example visualization



Constraints (i.e. What makes memory allocation interesting?)

- You may not know how software will malloc or free
  - i.e. Patterns are not always predictable
- You can only free a malloc’d block
- Typically we try to keep our blocks aligned
  - x86 - 8-byte blocks
  - x86-64 - 16 byte blocks
- We are often trying to maximize speed and memory efficiency
  - i.e. when we malloc we want it to be fast
  - i.e. When we malloc, we do not want to constantly allocate new memory
    - (which would be the fastest thing to do)
  - We might measure this using a throughput measure
    - (i.e. how many mallocs and frees can I do in 10 seconds)

Ideally, we have no empty blocks ever

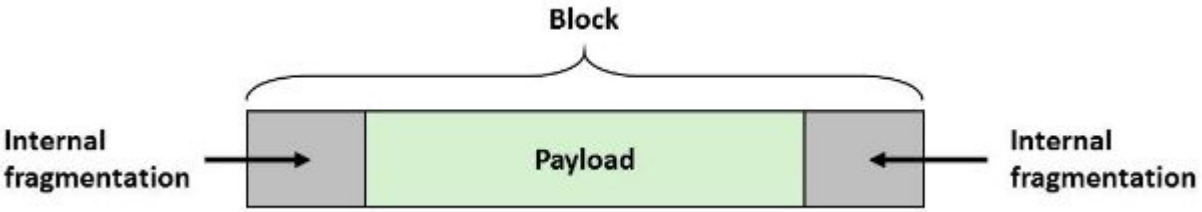


Poor memory utilization

- How we talk about this more formally
  a. Internal Fragmentation
  b. External Fragmentation

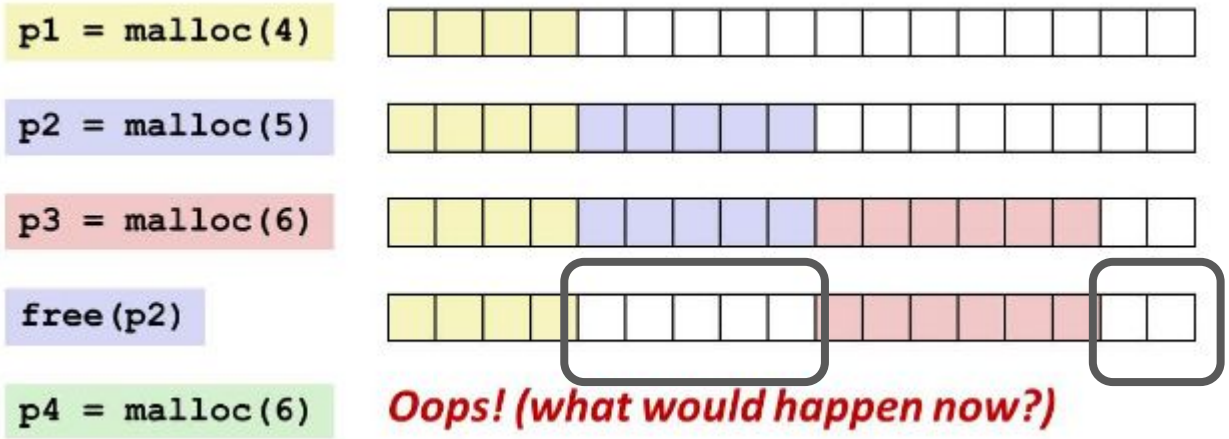
Internal Fragmentation

- e.g. we allocate structures smaller than our block size of 32 bytes (i.e. 1 char at a time)



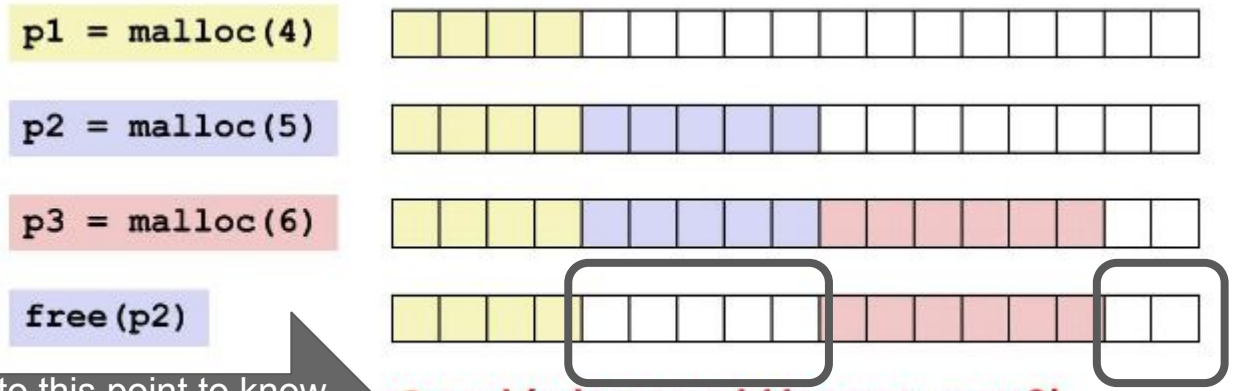
External Fragmentation

- We have enough blocks (i.e. we don’t need to extend the heap), but they are not contiguous



External Fragmentation

- We have enough blocks (i.e. we don't need to extend the heap), but they are not contiguous



It is hard up to this point to know fragmentation will occur. We cannot always predict the future.

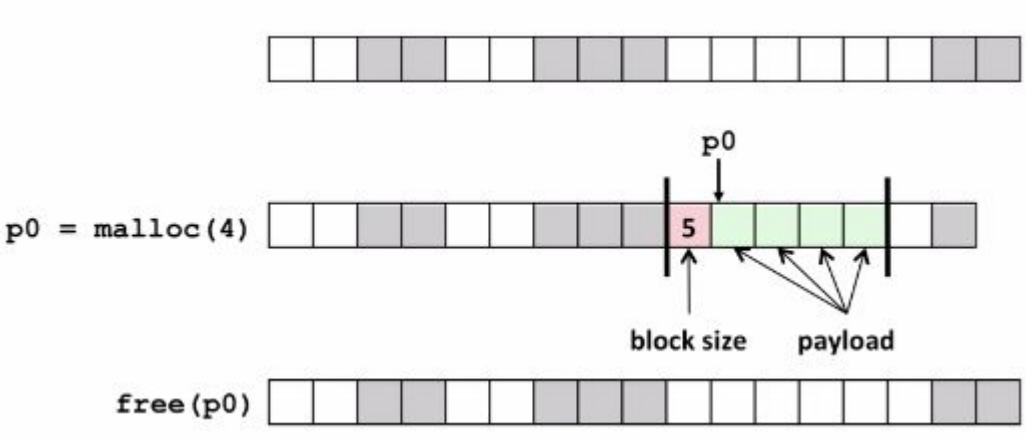
Implementation of an Allocator

- How do we know how much memory to free in a given pointer?
- How do we keep track of free blocks?
- What do we do with extra space when allocating structures smaller than the free block?
- How do we pick a block to use in allocation?
- How do we reinsert freed blocks?

How much do we free?

- Standard method
  - Keep the length of a block in the preceding block
    - (Often called the header)
  - This requires one extra word for every allocated block.

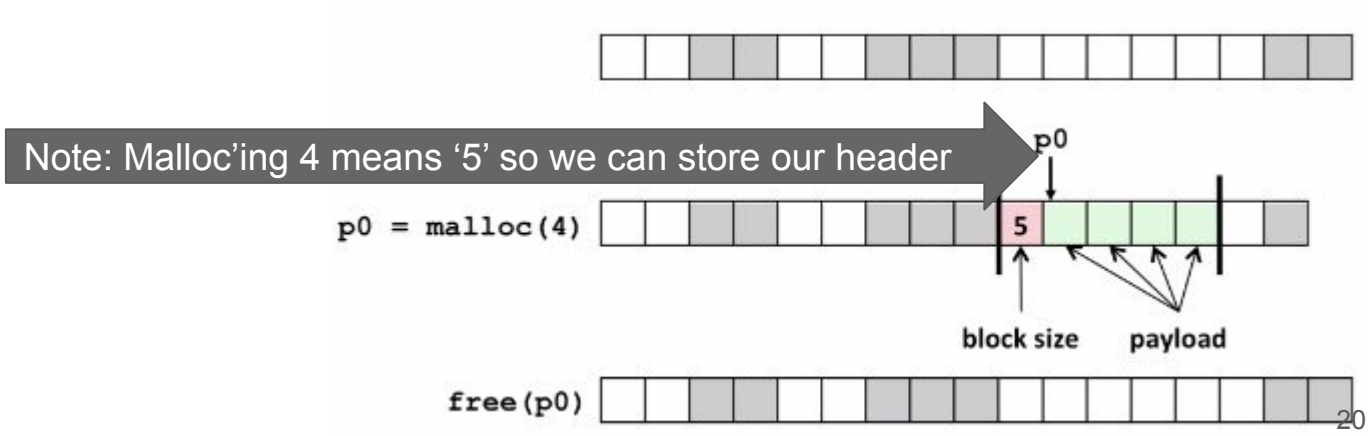
```
typedef long Align;  /* for alignment to long boundary */

union header {       /* block header: */
    struct {
        union header *ptr; /* next block if on free list */
        unsigned size;     /* size of this block */
    } s;
    Align x;         /* force alignment of blocks */
};

typedef union header Header;
```



How much do we free?

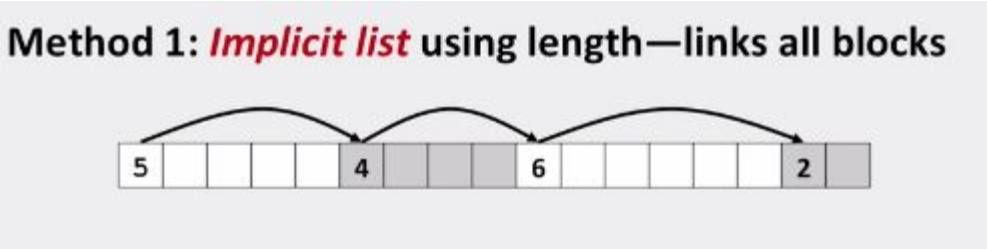
- Standard method
  - Keep the length of a block in the preceding block
    - (Often called the header)
  - This requires one extra word for every allocated block.

```
typedef long Align;  /* for alignment to long boundary */

union header {       /* block header: */
    struct {
        union header *ptr; /* next block if on free list */
        unsigned size;     /* size of this block */
    } s;
    Align x;         /* force alignment of blocks */
};

typedef union header Header;
```



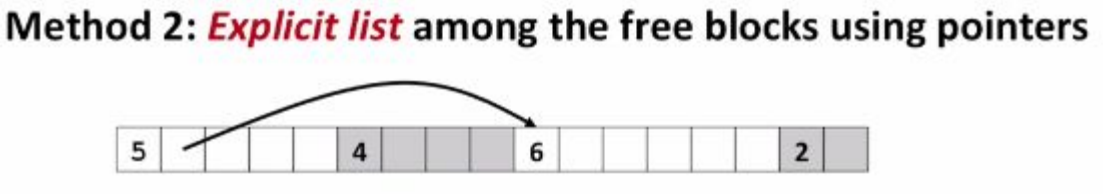
Keeping track of memory Strategy 1

- Keep track of everything we allocate
- If it is free, then we can reuse it.



Keeping track of memory Strategy 2

- Maintain another list that only points to free memory



Keeping track of memory Strategy 3 and Strategy 4

- Maintain a free list for different size classes
  - Perhaps all powers of 2
    - Or perhaps only allocate powers of 2
- Maintain a list that is sorted by size
  - Some balanced tree or binary tree structure

Implicit List: How to find a free block

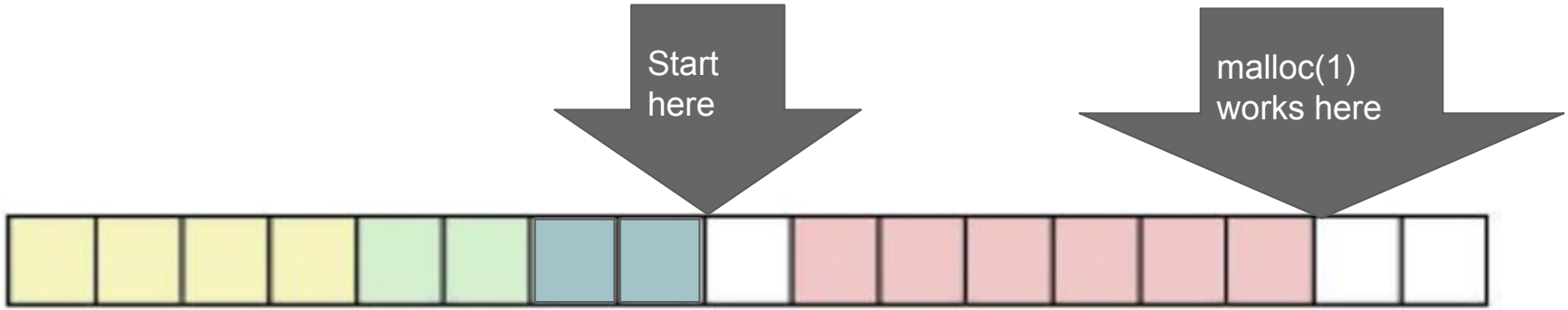
- First fit strategy (Perhaps for our homework :))
  - Search from beginning of list
    - Choose first free block that fits
  - Takes linear time (O (number of allocated and freed blocks)

Implicit List: How to find a free block

- First fit strategy (Perhaps for our homework :))
  - Search from beginning of list
    - Choose first free block that fits
  - Takes linear time (O (number of allocated and freed blocks)

Implicit List: How to find a free block

- Next-fit strategy
  - Search from where you left off from your previous search
    - Choose first free block that fits
  - Takes linear time (O (number of allocated and freed blocks)
    - May be better, avoids re-scanning unhelpful blocks if you are doing many similar allocations
    - Could make fragmentation worse though!

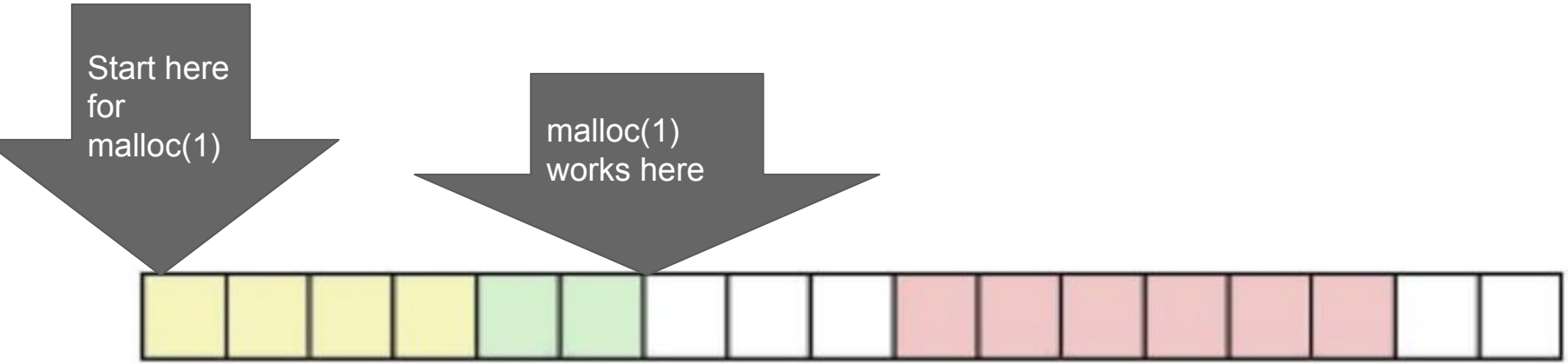


Implicit List: How to find a free block

- Best-fit strategy
  - Scan for the block that fits best
    - i.e. fewest bytes left over
  - Keeps fragmentation small and improves memory utilization
  - Will typically run slower than first-fit (longer scan for optimal block)

Implicit List: How to find a free block

- Best-fit strategy
  - Scan for the block that fits best
    - i.e. fewest bytes left over
  - Keeps fragmentation small and improves memory utilization
  - Will typically run slower than first-fit (longer scan for optimal block)

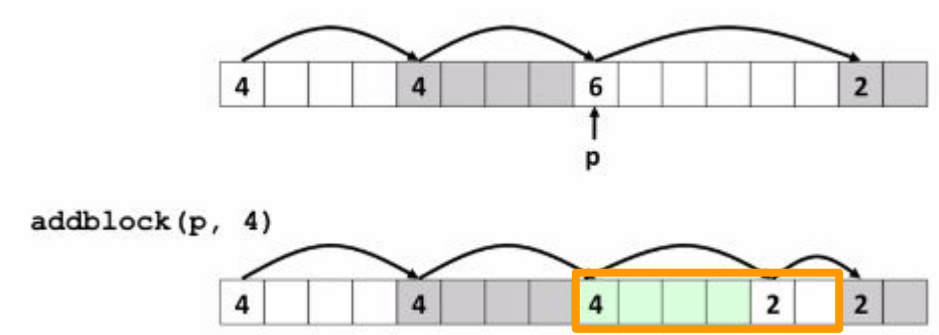


Implicit List: How to find a free block

- Best-fit strategy
  - Scan for the block that fits best
    - i.e. fewest bytes left over
  - Keeps fragmentation small and improves memory utilization
  - Will typically run slower than first-fit (longer scan for optimal block)

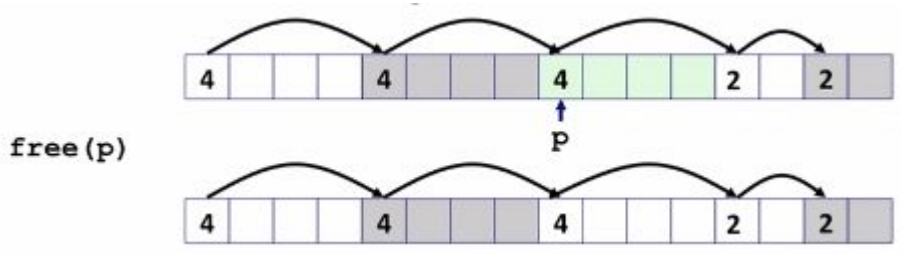
Implicit List: How allocate a free block?

- Depending on our strategy
  - We could allocate to the block found
    - i.e. we set a pointer to that block and size
    - We may want to also split that block
      - if there is room to do so (See example below)



Implicit List: How free block?

- Simply free a block by setting the free bit in the header
  - This will lead to fragmentation however!



Implicit List: How free block?

- Simply free a block by setting the free bit in the header
  - This will lead to fragmentation however!
    - It is suggested to coalesce these blocks (i.e. combine them)