EE 312

Day 5:

Dynamic Memory Allocation

Harsh Reality: Memory Matters!

Memory is not unlimited!

- It must be allocated and managed
- Many applications are memory dominated
  - *Especially those based on complex, graph algorithms*

Memory referencing bugs especially pernicious

- Effects are distant in both time and space

Memory performance is not uniform

- Cache and virtual memory effects can greatly affect program performance
- Adapting program to characteristics of memory system can lead to major speed improvements

Dynamic Memory Management

- There are two broad classes of memory management schemes:
- Explicit memory management
  - Application code responsible for both explicitly **allocating** and **freeing** memory.
  - Example: `malloc()` and free()
- Implicit memory management
  - Application code can allocate memory, but **does not free memory explicitly**
  - Rather, rely on **garbage collection** to “clean up” memory objects no longer in use
  - Used in languages like Java and Python
- Advantages and disadvantages of each?

Dynamic Memory Management

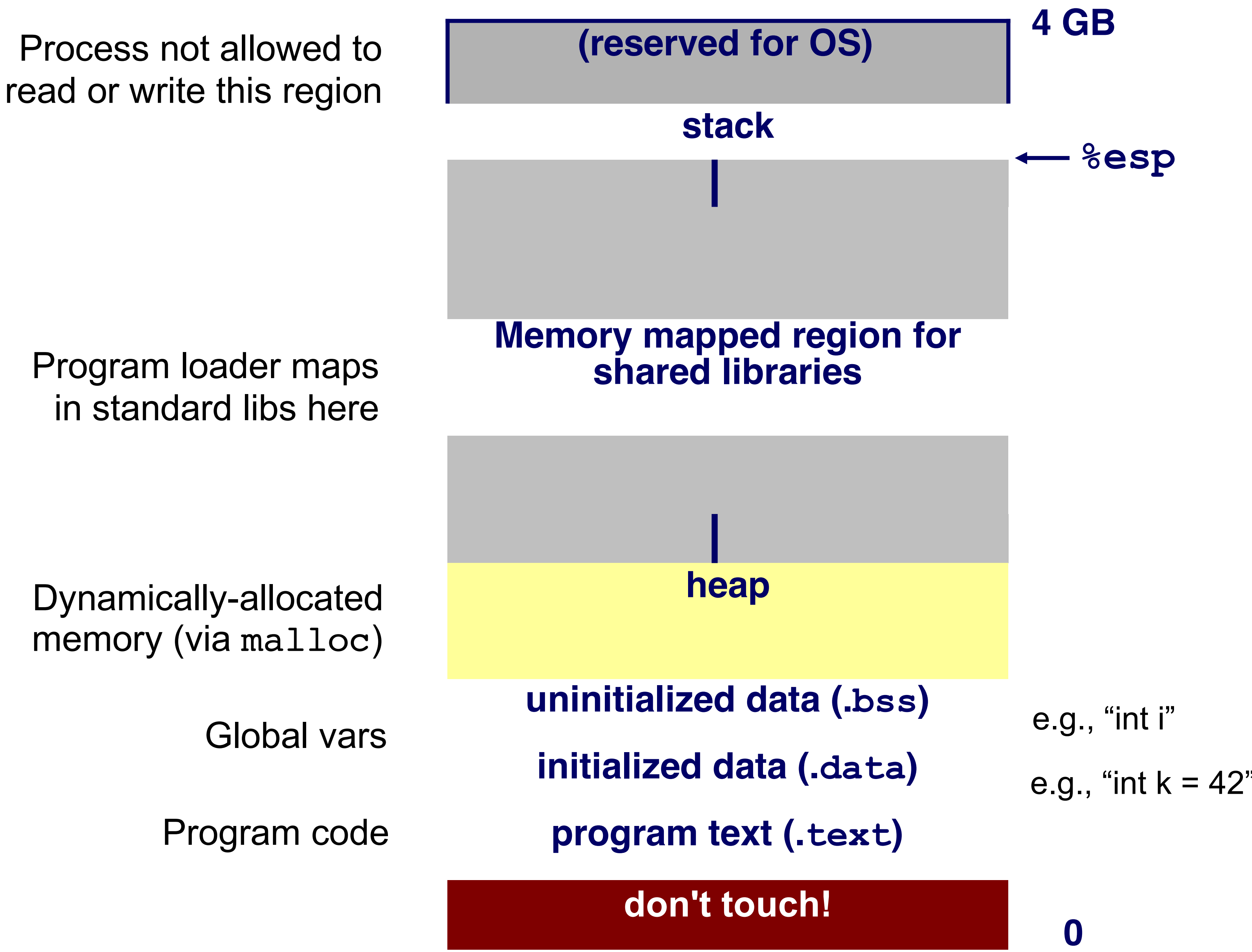
- There are two broad classes of memory management schemes:

- Explicit memory management
  - Application code responsible for both explicitly **allocating** and **freeing** memory.
  - Example: `malloc()` and free()

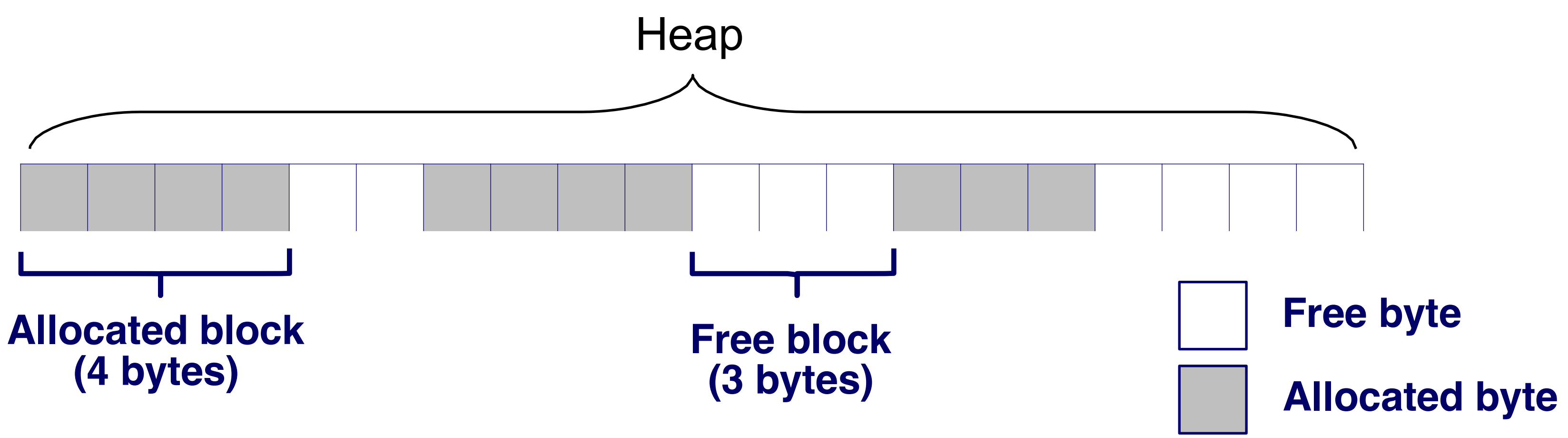
- Implicit memory management
  - Application code can allocate memory, but **does not free memory explicitly**
  - Rather, rely on **garbage collection** to “clean up” memory objects no longer in use
  - Used in languages like Java and Python

- Advantages and disadvantages of each?
  - Explicit management: Application has control over everything, possibly faster
  - But, application can seriously screw things up
    - *Attempt to access a freed block*
    - *Freeing same block multiple times*
    - *Forgetting to free blocks (**memory leak**)*

A process's view of memory



The heap



The **heap** is the region of a program's memory used for dynamic allocation.

Program can allocate and free blocks of memory within the heap.

Heap starts off with a fixed size (say, a few MB).

The heap can grow in size, but never shrinks!

- Program can grow the heap if it is too small to handle an allocation request.
- On UNIX, the sbrk() system call is used to expand the size of the heap.
  - *Why doesn't it make sense to shrink the heap?*

Malloc Package

`#include <stdlib.h>`

`void *malloc(size_t size)`

- If successful:
  - *Returns a pointer to a memory block of at least `size` bytes*
  - *If `size == 0`, returns NULL*
- If unsuccessful: returns NULL.

`void free(void *p)`

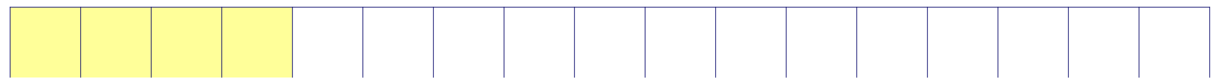
- Returns the block pointed at by `p` to pool of available memory
- `p` must come from a previous call to `malloc` or `realloc`.

`void *realloc(void *p, size_t size)`

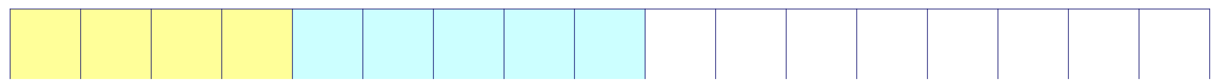
- Changes size of block `p` and returns pointer to new block.
- Contents of new block unchanged up to min of old and new size.

Allocation Examples

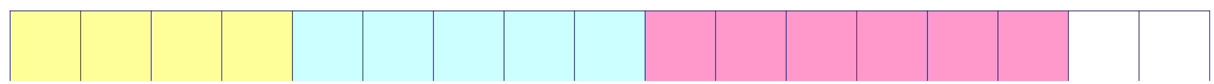
```
p1 = malloc(4)
```



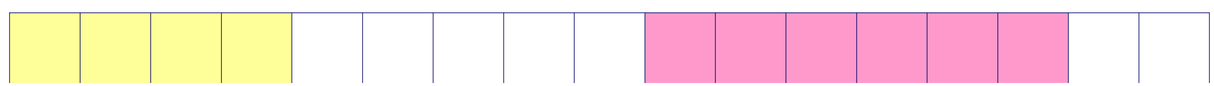
```
p2 = malloc(5)
```



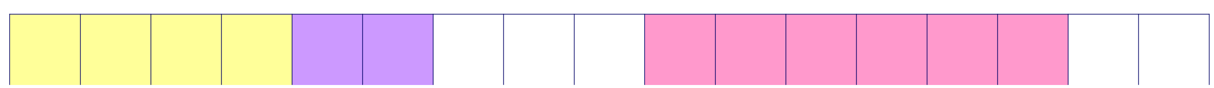
```
p3 = malloc(6)
```



```
free(p2)
```



```
p4 = malloc(2)
```



Constraints

Application code is allowed to....

- Can issue arbitrary sequence of allocation and free requests
- Free requests must correspond to an allocated block

The memory management code must obey the following constraints:

- Can't control number or size of requested blocks
- Must respond immediately to all allocation requests
  - *i.e., can't reorder or buffer requests*
- Must allocate blocks from free memory
  - *i.e., can only place allocated blocks in free memory*
- Must align blocks so they satisfy all alignment requirements
  - *8 byte alignment for GNU malloc (`libc` malloc) on Linux boxes*
- Can only manipulate and modify free memory
- Can't move the allocated blocks once they are allocated
  - *i.e., compaction is not allowed*

Performance Goals: Allocation overhead

Want our memory allocator to be fast!

- Minimize the overhead of both allocation and deallocation operations.

One useful metric is **throughput**:

- Given a series of allocate or free requests
- Maximize the number of completed requests per unit time

Example:

- 5,000 `malloc` calls and 5,000 `free` calls in 10 seconds
- Throughput is 1,000 operations/second.

Note that a fast allocator may not be efficient in terms of memory utilization.

- Faster allocators tend to be "sloppier"
- To do the best job of space utilization, operations must take more time.
- Trick is to balance these two conflicting goals.

Performance Goals: Memory Utilization

Allocators rarely do a perfect job of managing memory.

- Usually there is some "waste" involved in the process.

Examples of waste...

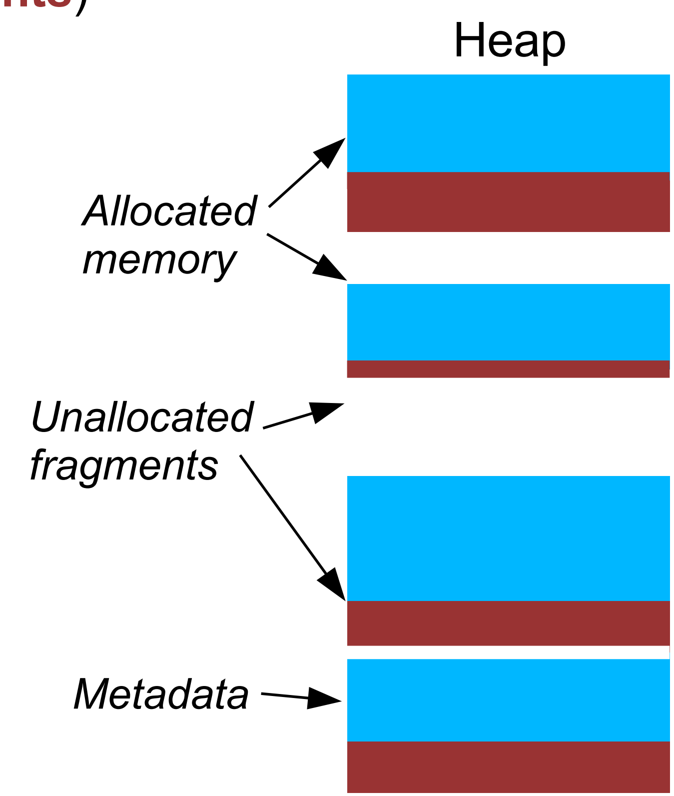
- Extra metadata or internal structures used by the allocator itself (example: Keeping track of where free memory is located)
- Chunks of heap memory that are unallocated (**fragments**)

We define **memory utilization** as...

- The **total amount of memory allocated to the application** divided by the total **heap size**

Ideally, we'd like utilization to be to 100%

- In practice this is not possible, but would be good to get close.



Conflicting performance goals

Note that good throughput and good utilization are difficult to achieve simultaneously.

A fast allocator may not be efficient in terms of memory utilization.

- Faster allocators tend to be “sloppier” with their memory usage.

Likewise, a space-efficient allocator may not be very fast

- To keep track of memory waste (i.e., tracking fragments), the allocation operations generally take longer to run.

Trick is to balance these two conflicting goals.

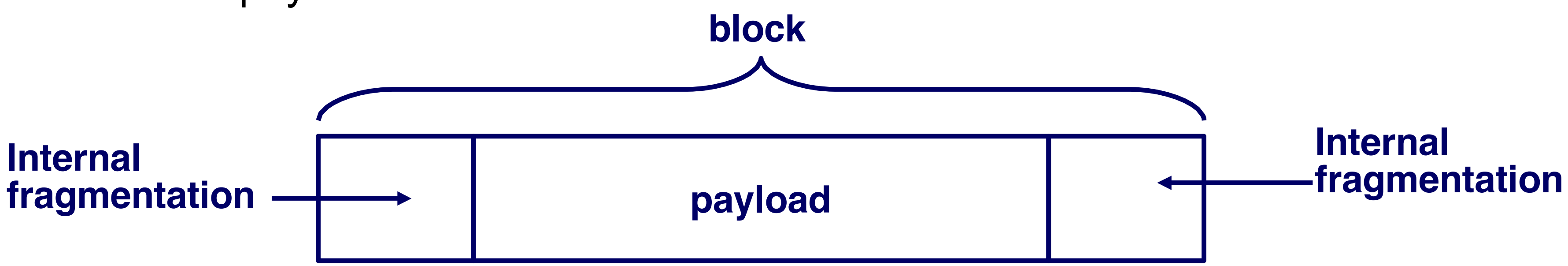
Internal Fragmentation

Poor memory utilization caused by **fragmentation.**

- Comes in two forms: **internal** and **external** fragmentation

Internal fragmentation

- **Internal fragmentation** is the difference between the block size and the payload size.

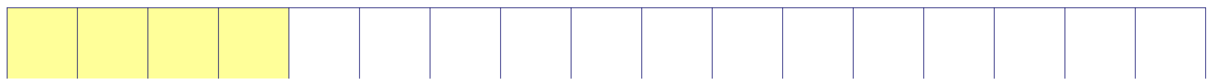


- Caused by overhead of maintaining heap data structures, padding for alignment purposes, or the policy used by the memory allocator
- Example: Say the allocator always “rounds up” to next highest power of 2 when allocating blocks.
  - *So malloc(1025) will actually allocate 2048 bytes of heap space!*

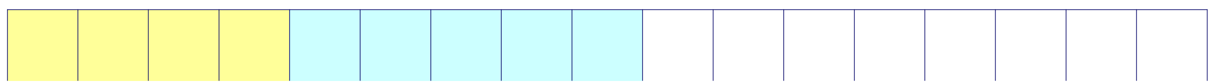
External Fragmentation

Occurs when there is enough aggregate heap memory, but no single free block is large enough to satisfy a given request.

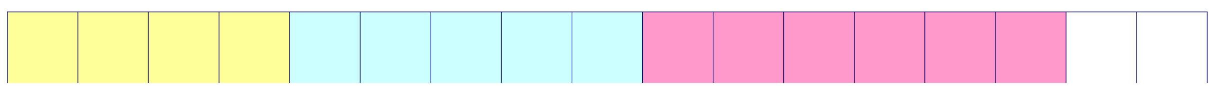
```
p1 = malloc(4)
```



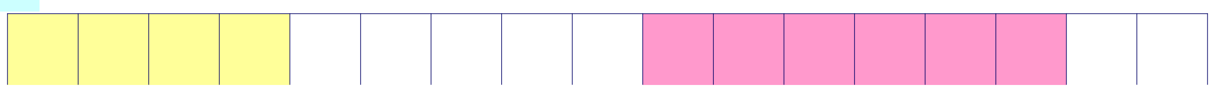
```
p2 = malloc(5)
```



```
p3 = malloc(6)
```



```
free(p2)
```



```
p4 = malloc(6)
```

oops! - no free block large enough.

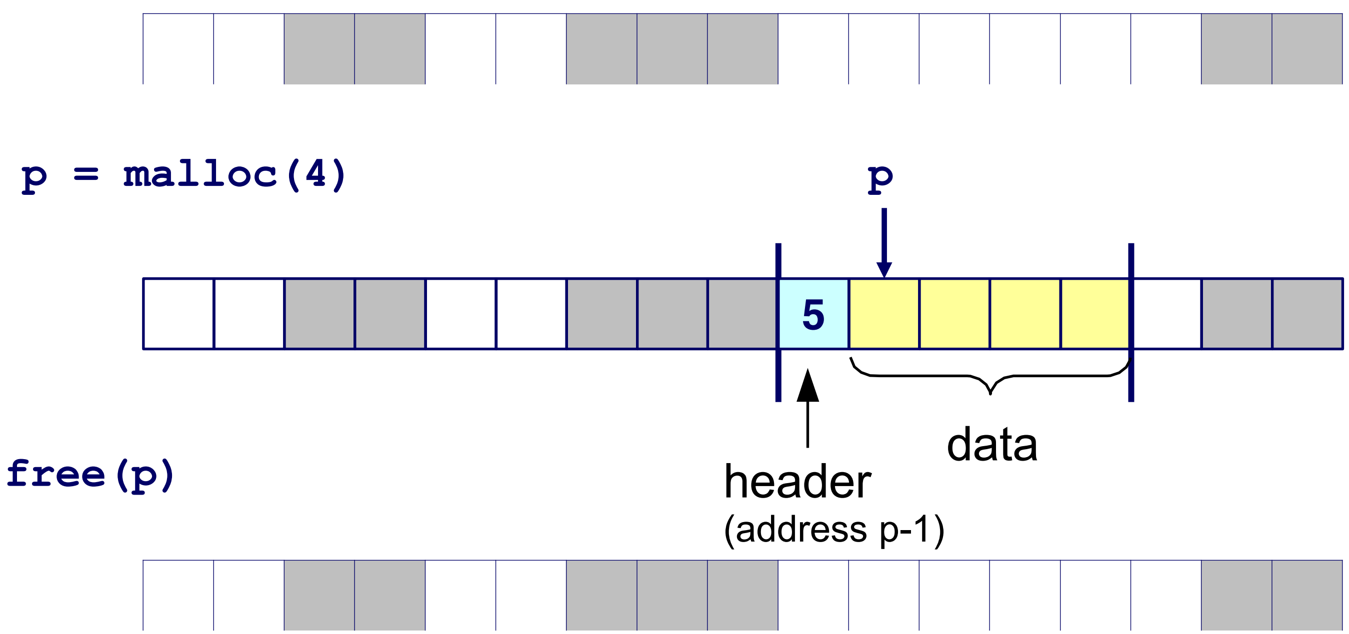
Implementation Issues

- How do we know how much memory to free just given a pointer?
- How do we keep track of the free blocks?
- What do we do with the extra space when allocating a memory block that is smaller than the free block it is placed in?
- How do we pick which free block to use for allocation?

Knowing how much to free

Standard method

- Keep the length of a block in a **header** preceding the block.
- Requires an extra word for every allocated block



Keeping Track of Free Blocks

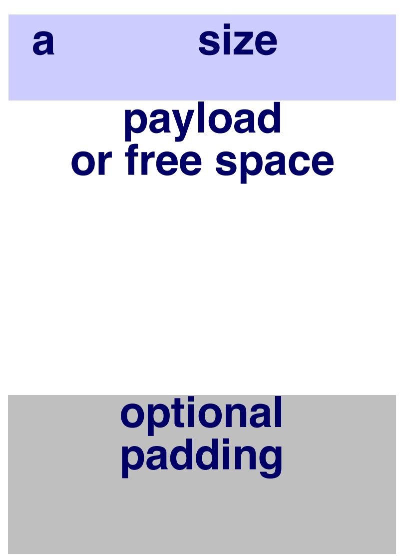
- One of the biggest jobs of an allocator is knowing where the free memory is.
- The allocator's approach to this problem affects...
  - Throughput – time to complete a malloc() or free()
  - Space utilization – amount of extra metadata used to track location of free memory.
- There are many approaches to free space management.
  - Today, we will talk about one: **Implicit free lists.**

Implicit free list

Idea: Each block contains a **header** with some extra information.

- **Allocated bit** indicates whether block is allocated or free.
- Size field indicates entire size of block (including the header)
- Trick: Allocation bit is just the **high-order bit** of the size word

- For this lecture, let's assume the header size is 1 byte.
  - Makes the pictures that I'll show later on easier to understand.
  - This means the block size is only 7 bits, so max. block size is 127 bytes (2^7-1).
  - Clearly a real implementation would want to use a larger header (e.g., 4 bytes).

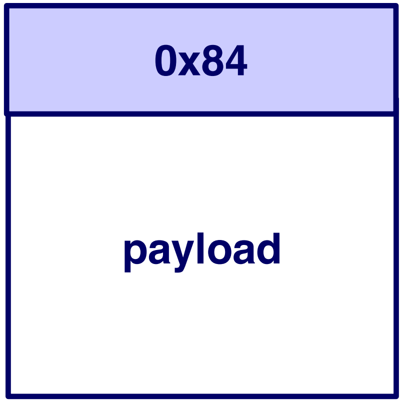


a = 1: block is allocated
a = 0: block is free

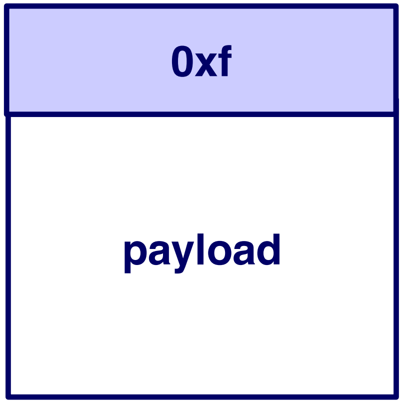
size: block size

payload: application data

Examples

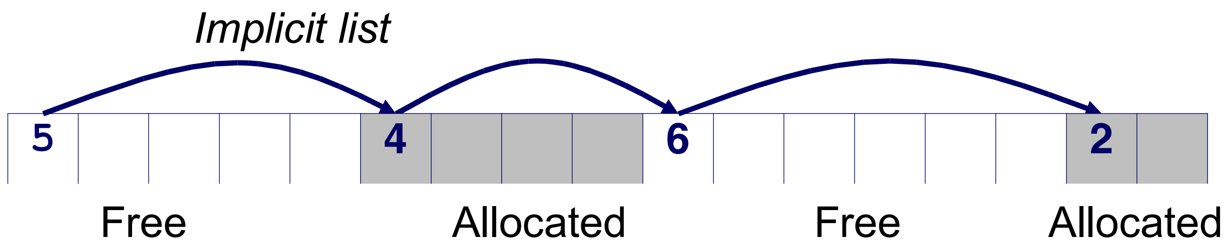


0x84 in binary: 1000 0100
allocated = 1
size = 0x4 = 4 bytes



0xf in binary: 0000 1111
allocated = 0
size = 0xf = 15 bytes

Implicit free list

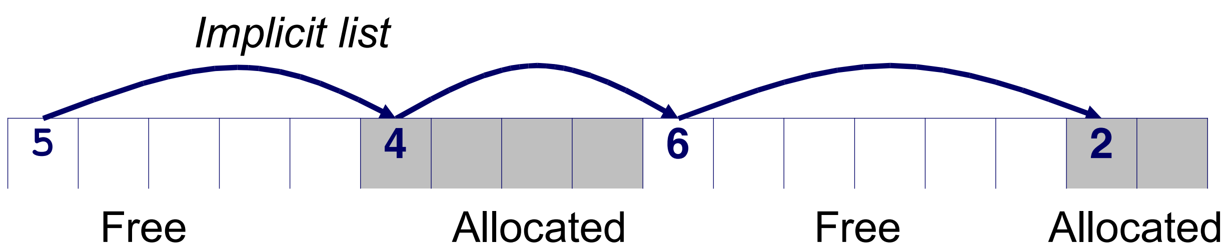


No **explicit** structure tracking location of free/allocated blocks.

- Rather, the size word (and allocated bit) in each block form an **implicit** “block list”

How do we find a free block in the heap?

Implicit free list



No **explicit** structure tracking location of free/allocated blocks.

- Rather, the size word (and allocated bit) in each block form an **implicit** “block list”

- How do we find a free block in the heap?

- Start scanning from the beginning of the heap.

- Traverse each block until (a) we find a free block and (b) the block is large enough to handle the request.

- This is called the **first fit** strategy.

Example

- If we need an array of n ints, then we can do
  - `int* A = malloc(n*sizeof(int));`

- A holds the address of the first element of this block of 4n bytes, and A can be used as an array. For example,
  - `if (A != NULL) for (i=0;i<n;i++) A[i] = 0;`

- will initialize all elements in the array to 0. We note that A[i] is the content at address (A+i). Therefore we can also write
  - `for (i=0;i<n;i++) *(A+i) = 0;`

- Recall that A points to the first byte in the block and A+i points to the address of the ith element in the list. That is `&A[i]`.
  We can also see the operator [] is equivalent to doing pointer arithmetic to obtain the content of the address.

calloc and realloc

- calloc and realloc are two functions that can be useful in dynamic memory management
  - `void *calloc(size_t nmemb, size_t size);`

- allocates memory for an array of **nmemb** elements each of **size** and returns a pointer to the allocated memory. Unlike malloc the memory is automatically set to zero.

  - `calloc(n, sizeof(int))`
- is equivalent to
  - `malloc(n*sizeof(int))`

Freeing memory

- `int n = 10;`
- `int A[10]; i=0;`
- `for (i=0; i < n ; i++) A[i] = rand();`
- `int* B = (int*)malloc(2*n);`
- `B = A;`

- What is wrong with the above code?

- `B = A;` is legal. Is `A = B;` legal?

- Freeing a block should be done only once. So assign null to pointers after you are done with them.

Malloc Example

```
void foo(int n, int m) {
  int i, *p;

  /* allocate a block of n ints */
   p = (int *)malloc(n * sizeof(int));
   if (p == NULL) {
    perror("malloc");
    exit(0);
  }
  for (i=0; i<n; i++) p[i] = i;

  /* add m bytes to end of p block */
  if ((p = (int *) realloc(p, (n+m) * sizeof(int))) == NULL) {
    perror("realloc");
    exit(0);
  }
  for (i=n; i < n+m; i++) p[i] = i;

  /* print new array */
  for (i=0; i<n+m; i++)
    printf("%d\n", p[i]);

  free(p); /* return p to available memory pool */
}
```