

Last Time: Virtual Memory

- **Main idea: View DRAM as cache for disk**
 - Fully associative cache
 - “Cache block size” is one page, usually 4 KB
 - Write- back
 - Cache hit is handled by the CPU
 - Cache miss is a page fault: handled by the OS
 - Miss rate must be very low
- **Page tables**
 - Support finding page contents either in DRAM or on disk
 - Can be large (solution?)
 - Can be slow (solution?)
- **The point**
 - “Soft” failure in out-of-memory condition
 - Programmer sees flat address space, paged nature of memory is hidden by the OS

Today: Dynamic Memory Allocation

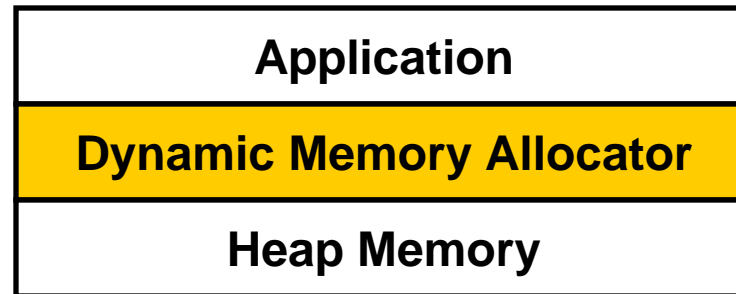
- **Topics**
 - Simple *explicit* allocators
 - Data structures
 - Mechanisms
 - Policies

Harsh Reality #3

- *Memory Matters*

- **Memory is not unbounded**
 - It must be allocated and managed
 - Many applications are memory dominated
 - » Especially those based on complex, graph algorithms
 - CAD, theorem proving, model checking,
- **Memory referencing bugs are especially evil**
 - Effects are distant in both time and space
 - » hence these bugs are particularly pernicious
- **Memory performance is not uniform**
 - Cache and virtual memory effects can greatly affect program performance
 - Adapting program to the characteristics of memory system can lead to major speed improvements

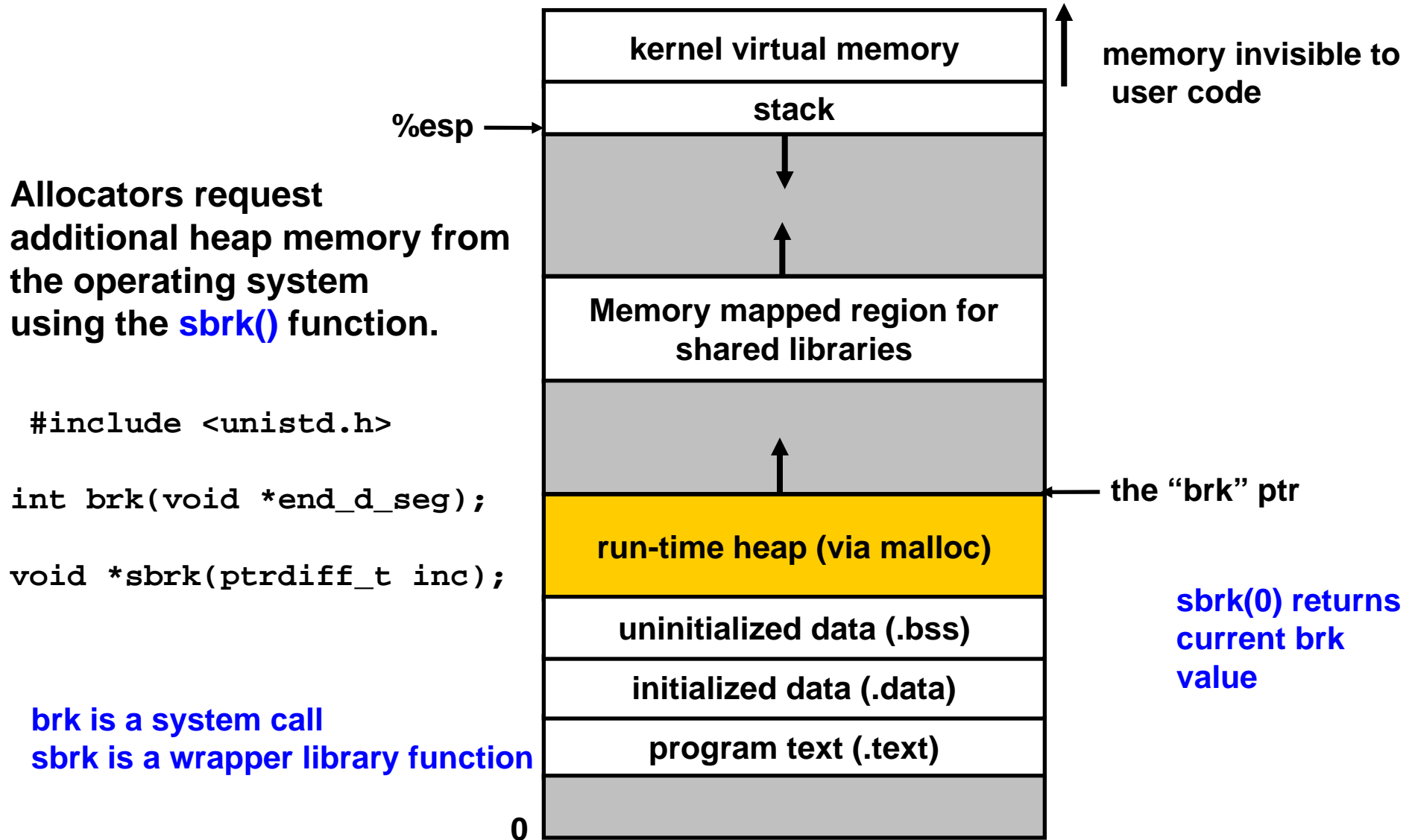
Dynamic Memory Allocation



The API

- **Explicit vs. Implicit Memory Allocator**
 - **Explicit:** application allocates and frees space
 - » E.g., `malloc` and `free` in C
 - **Implicit:** application allocates when a dynamic structure is created
 - » but does not free space
 - » free space is automatically *garbage collected* in Java, ML, or Lisp
- **Allocation**
 - In both cases the memory allocator provides an abstraction of memory as a set of blocks
 - Doles out free memory blocks to application
- **Today: start with simple explicit memory allocation**

Process memory image



Malloc package

- `#include <stdlib.h>`
- `void *malloc(size_t size)`
 - **if successful:**
 - » returns a pointer to a memory block of at least `size` bytes, aligned to 8-byte boundary (in IA32 LINUX).
 - » 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` to `size` bytes and returns ptr to new block.
 - contents of new block unchanged up to min of old and new size.

Malloc example

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

    /* allocate a block of n ints */
    if ((p = (int *) malloc(n * sizeof(int))) == NULL) {
        perror("malloc");      /* print malloc FAILURE to stdio */
        exit(0);
    }
    for (i=0; i<n; i++)        /* otherwise use the new space */
        p[i] = i;

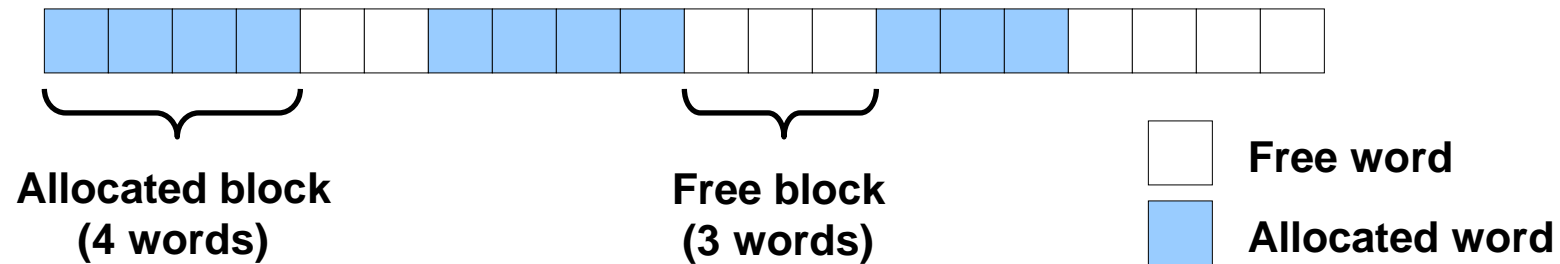
    /* add m bytes to end of p block */
    if ((p = (int *) realloc(p, (n+m) * sizeof(int))) == NULL) {
        perror("realloc");
        exit(0);
    }
    for (i=n; i < n+m; i++) /* use m more int's */
        p[i] = i;

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

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

Assumptions

- Assumptions made in this lecture
 - memory is word addressed (each word can hold a pointer)

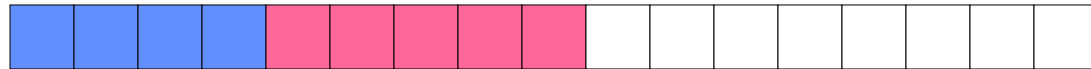


Allocation examples

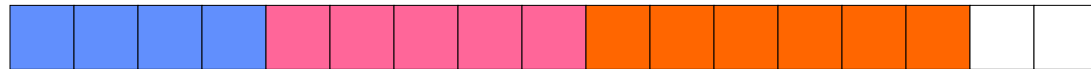
`p1 = malloc(4)`



`p2 = malloc(5)`



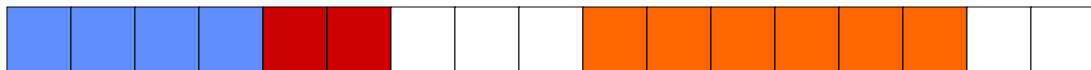
`p3 = malloc(6)`



`free(p2)`



`p4 = malloc(2)`



Constraints

- **Applications:**

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

- **Allocators**

- Can't control number or size of allocated 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
 - » usually 8 byte alignment
- Can only manipulate and modify free memory
 - » **CAN NOT** move allocated blocks since application level pointer would be invalidated silently.
 - » → compaction is not allowed

Goals of good malloc/free

- **Primary goals**

- **Good time performance for `malloc` and `free`**
 - » Ideally should take constant time (not always possible)
 - » Should certainly not take linear time in the number of blocks
- **Good space utilization**
 - » User allocated structures should be large fraction of the heap.
 - » want to minimize “fragmentation”.

- **Some other goals**

- **Good locality properties**
 - » structures allocated close in time should be close in space
 - » “similar” objects should be allocated close in space
- **Robust**
 - » can check that `free(p1)` is on a valid allocated object `p1`
 - » can check that memory references are to allocated space

Performance goals

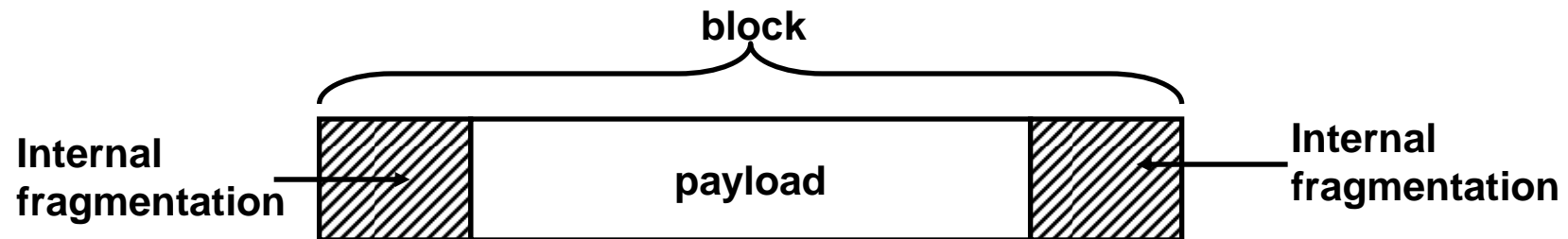
- Given some sequence of malloc and free requests:
 - $R_0, R_1, \dots, R_k, \dots, R_{n-1}$
- **Throughput:**
 - *want to process maximum requests for unit time*
 - *requests = mallocs + frees*
- **Utilization**
 - *% of heap that is allocated*
- **Peak Utilization**
 - *max utilization for the request sequence*
- **Want to maximize throughput and peak memory utilization.**
 - *these goals are often conflicting*

Performance goals: peak memory utilization

- Given some sequence of malloc and free requests:
 - $R_0, R_1, \dots, R_k, \dots, R_{n-1}$
- **Def: aggregate payload P_k :**
 - `malloc(p)` results in a block with a *payload* of p bytes..
 - After request R_k has completed, the *aggregate payload* P_k is the sum of currently allocated payloads.
- **Def: current heap size is denoted by H_k**
 - Note that H_k is monotonically nondecreasing
 - » not a strict requirement but it makes thinking about things more simple
- **Def: peak memory utilization:**
 - After k requests, *peak memory utilization* is:
 - » $U_k = (\max_{i < k} P_i) / H_k$

Internal Fragmentation

- **Poor memory utilization caused by *fragmentation*.**
 - Comes in two forms: internal and external fragmentation
- **Internal fragmentation**
 - For some block, 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 explicit policy decisions (e.g., not to split the block).
- Depends only on the pattern of previous requests, and thus is easy to measure.

External fragmentation

Occurs when there is enough aggregate heap memory, but no single free block is large enough → heap size needs to grow

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



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



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



```
free(p2)
```

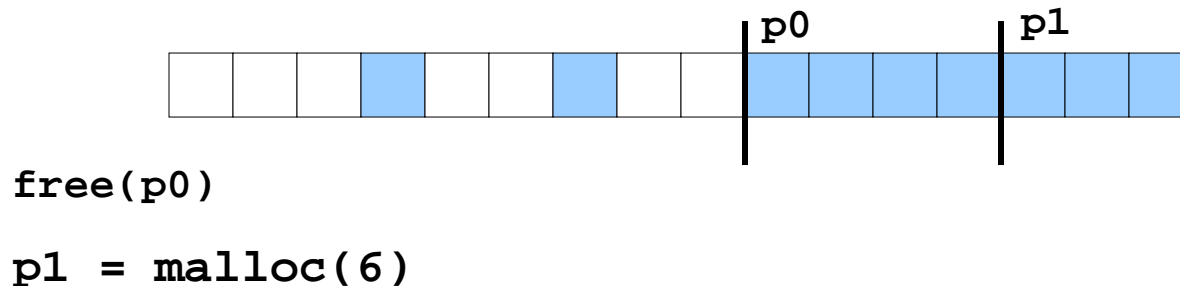


```
p4 = malloc(6) oops!
```

External fragmentation depends on the pattern of future requests, and thus is difficult to measure.

Implementation issues

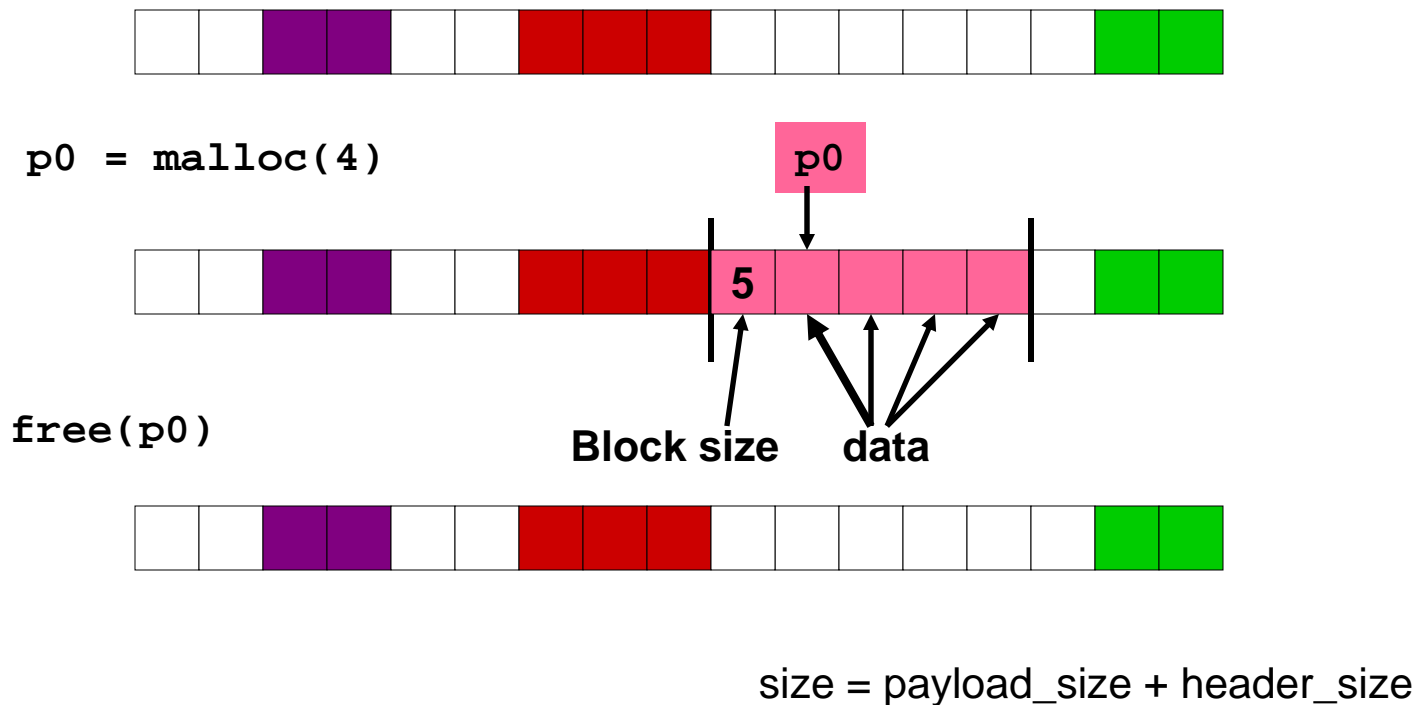
- How do we know how much memory to free just given a pointer?
- How do we keep track of the free blocks?
- What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?
- How do we pick a block to use for allocation -- many might fit?
- How do we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



Knowing how much to free

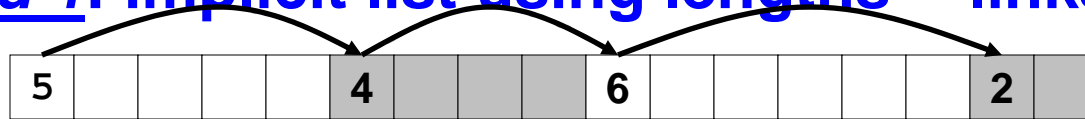
- **Standard method**

- keep the length of a structure in the word preceding the structure
 - » This word is often called the *header field* or *header*
- requires an extra word for every allocated structure

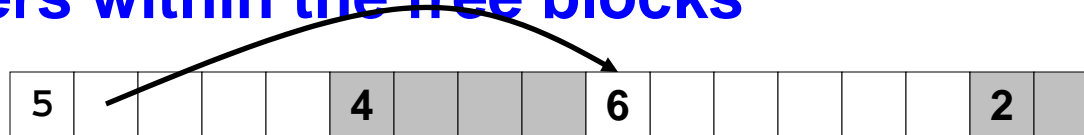


Keeping track of free blocks

- **Method 1**: implicit list using lengths -- links all blocks



- **Method 2**: explicit list among the free blocks using pointers within the free blocks



- **Method 3**: segregated free lists

- Different free lists for different size classes

- **Method 4**: blocks sorted by size

- Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

Beware

- **Terminology overloading**

- **allocators**

- » **cataloged by the application code perspective**

- **explicit** – code has instructions for *malloc* and *free*
 - **implicit** - alloc and free happen automatically
 - create some $n \times n$ array of ints and $n^2 * \text{sizeof}(\text{int})$ gets allocated
 - free happens as a result of garbage collection

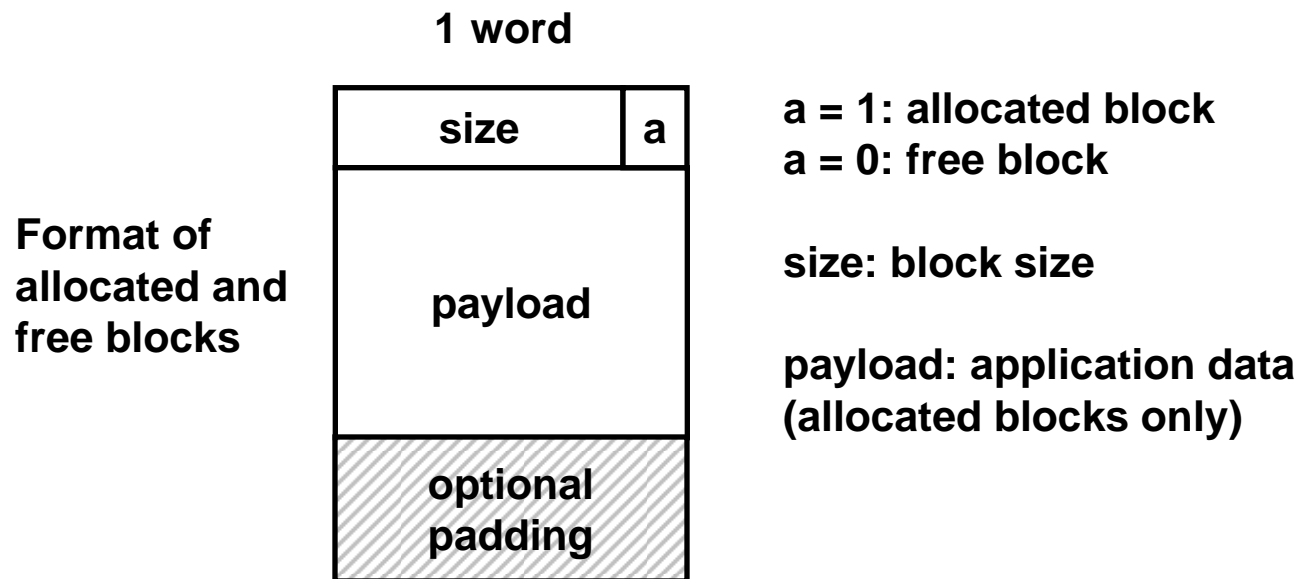
- **free lists**

- » **cataloged by the free list structure**

- **implicit** – next block found by size in the header
 - next block may be allocated rather than free
 - **explicit** – block contains a pointer to the next free block
 - size is there as well usually since you want to know if the new request will fit here
 - extra pointer overhead will increase internal fragmentation costs

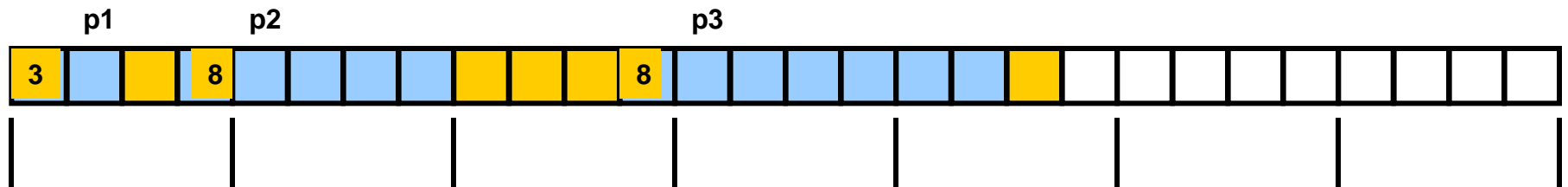
Method 1: implicit list

- Need to identify whether each block is free or allocated
 - Can use extra bit
 - » 8 byte alignment says low order 3-bits aren't needed for size
 - » hence there are 3 extras
 - Bit can be put in the same word as the size - if block sizes are always multiples of two (mask out low order bit when reading size).



Cache Friendly Padding

- **Heap is in memory – hence cache friendly alignment is a good idea**
 - note however that what we want to align is the payload
 - hence padding **may** need to be BOTH before and/or after
 - » basic strategy choice – this is just one example



- p1 ← malloc(1) – on cache line but pad for next
- p2 ← malloc(4) – even fit for 4 word line but pad for next
- p3 ← malloc(6) – split over 2 cache lines and pad for next

free 

payload 

pad or header 

Why can't you pad for yourself??

Implicit list: finding a free block

- **First fit:**

- Search list from beginning, choose first free block that fits

```
p = start;
while ((p < end) &&      \\ not passed end
      ~(*p & 1) &&      \\ not already allocated
      (*p <= len));    \\ too small
```

- Can take linear time in total number of blocks (allocated and free)

- **Next fit:**

- Like first-fit, but search list from location of end of previous search
- Research suggests that fragmentation is worse

- **Best fit:**

- Search the list, choose the free block with the closest size that fits
- Keeps fragments small --- usually helps fragmentation
- Will typically run slower than first-fit

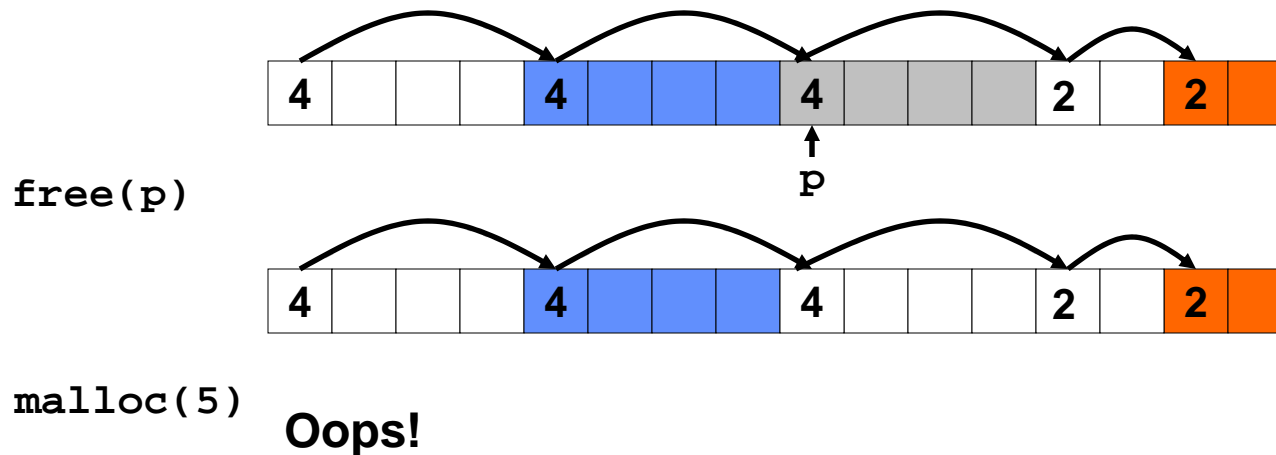
Implicit list: freeing a block

- Simplest implementation:

- Only need to clear allocated flag

```
void free_block(ptr p) { *p = *p & -2 }
```

- But can lead to “false fragmentation”

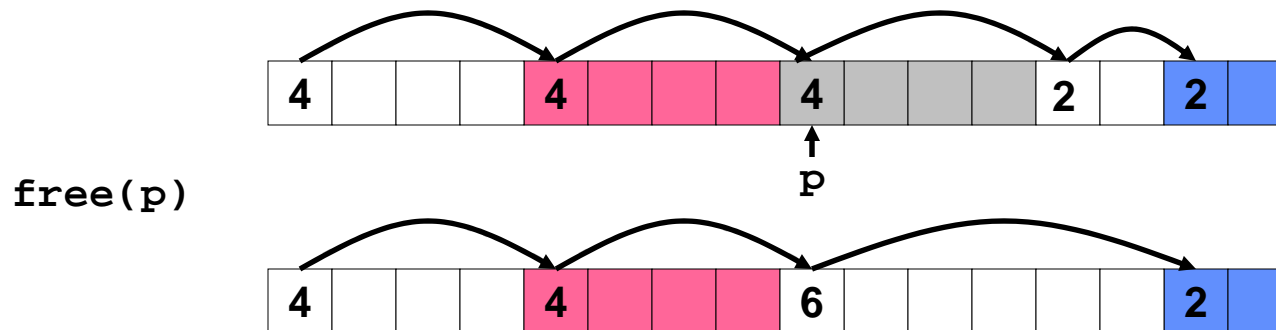


There is enough free space, but the allocator won't be able to find it

Implicit list: coalescing

- Join with next and/or previous block if they are free
 - Coalescing with next block

```
void free_block(ptr p) {  
    *p = *p & -2;           // clear allocated flag  
    next = p + *p;         // find next block  
    if ((*next & 1) == 0)  
        *p = *p + *next;   // add to this block if  
    }                       // not allocated
```

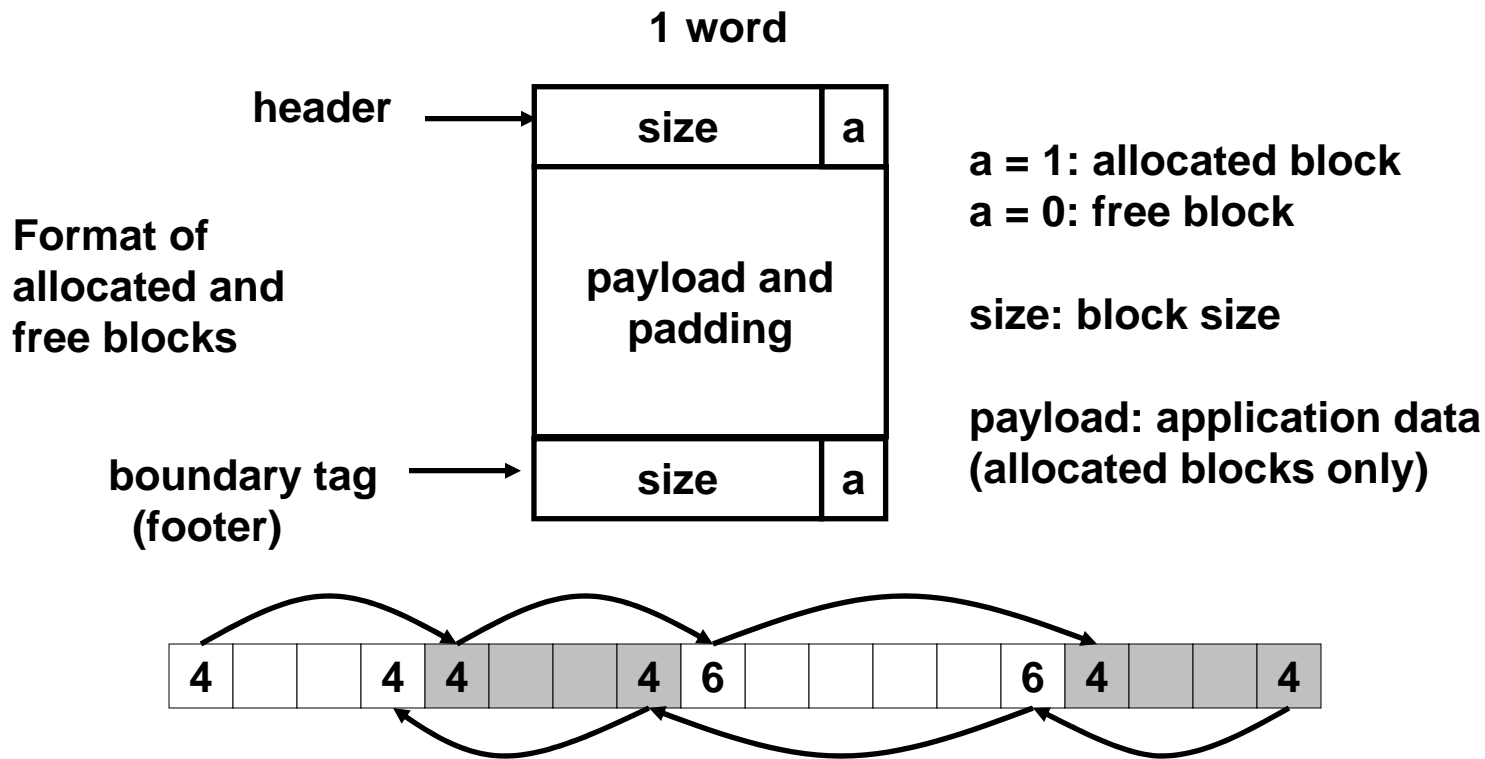


- But how do we coalesce with previous block?

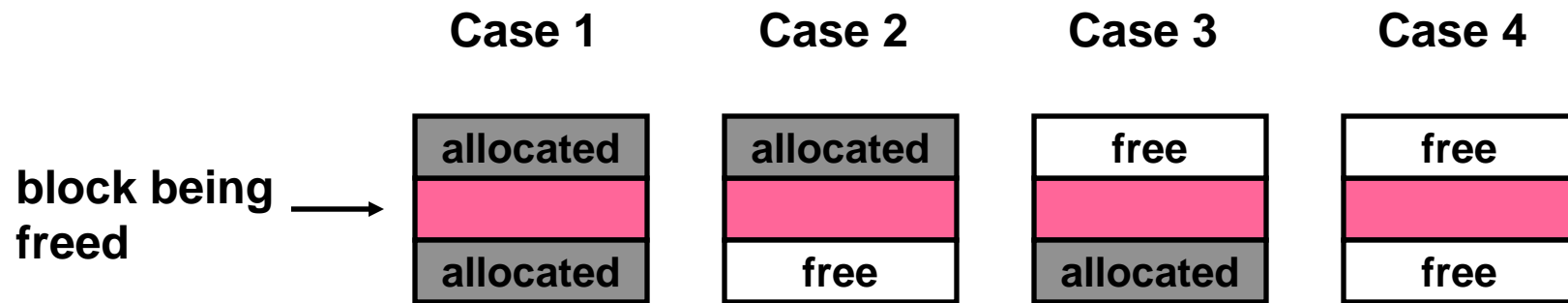
Implicit list: bidirectional

- **Boundary tags [Knuth73]**

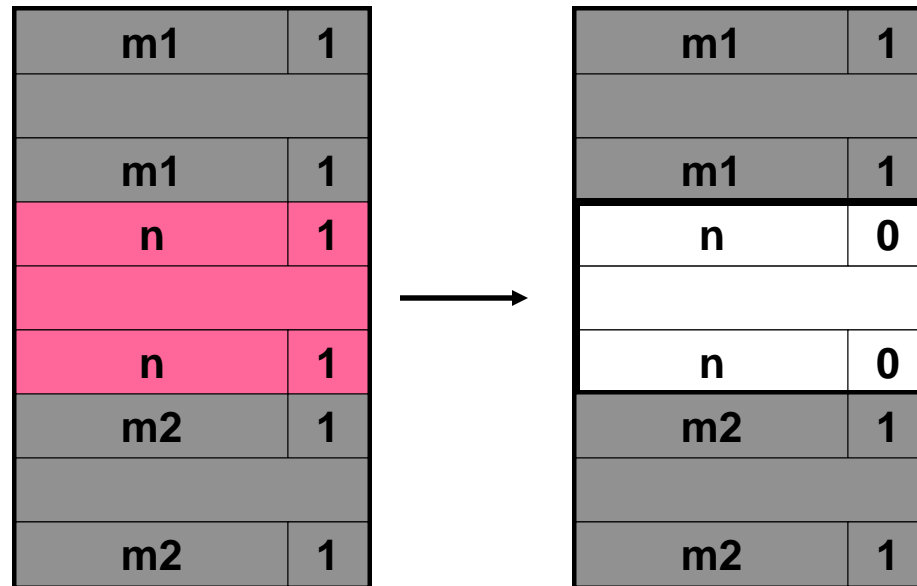
- replicate size/allocated word at bottom of blocks
- Allows us to traverse the “list” backwards, but requires extra space
- Important and general technique!



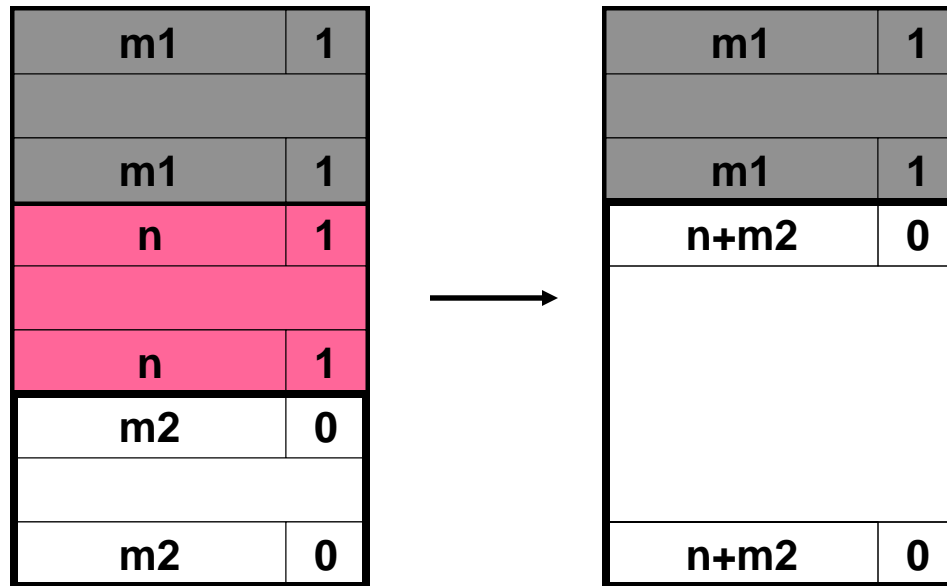
Constant time coalescing



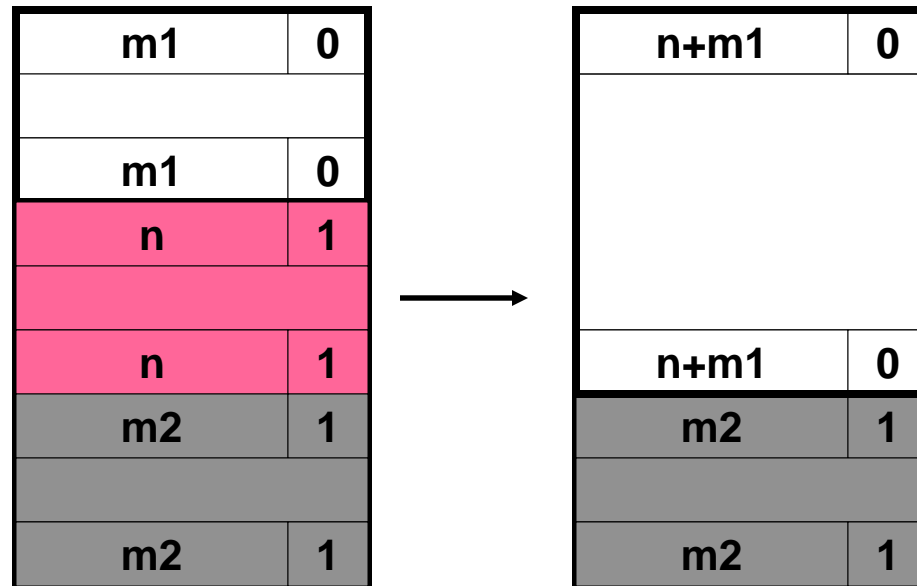
Constant time coalescing (case 1)



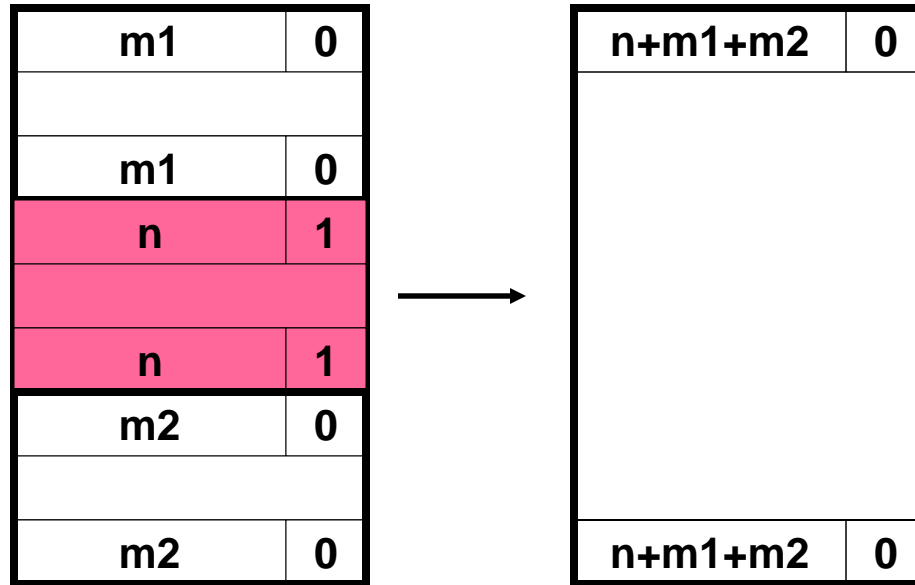
Constant time coalescing (case 2)



Constant time coalescing (case 3)



Constant time coalescing (case 4)



Summary of key allocator policies

- **Placement policy:**

- first fit, next fit, best fit, etc.
- trades off lower throughput for less fragmentation
 - » Interesting observation: segregated free lists (next lecture) approximate a best fit placement policy without having to search entire free list.

- **Splitting policy:**

- When do we go ahead and split free blocks?
- How much internal fragmentation are we willing to tolerate?

- **Coalescing policy:**

- immediate coalescing: coalesce adjacent blocks each time free is called
- Deferred coalescing: try to improve performance of free by deferring coalescing until needed. e.g.,
 - » coalesce as you scan the free list for malloc.
 - » coalesce when the amount of external fragmentation reaches some threshold.

Implicit lists: Summary

- **Implementation:** very simple
- **Allocate:** linear time worst case
- **Free:** constant time worst case -- even with coalescing
- **Memory usage:** will depend on placement policy
 - First fit, next fit or best fit
- **Not used in practice for malloc/free**
 - because of linear time allocate
 - finds use in other special purpose applications.
- **However, the concepts of splitting and boundary tag coalescing are general to *all* allocators.**