Systems Programming

Dynamic Memory Allocation: Basic Concepts

Textbook coverage:
Ch 9.9: Dynamic Memory Allocation

Byoungyoung Lee
Seoul National University
byoungyoung@snu.ac.kr
https://lifeasageek.github.io

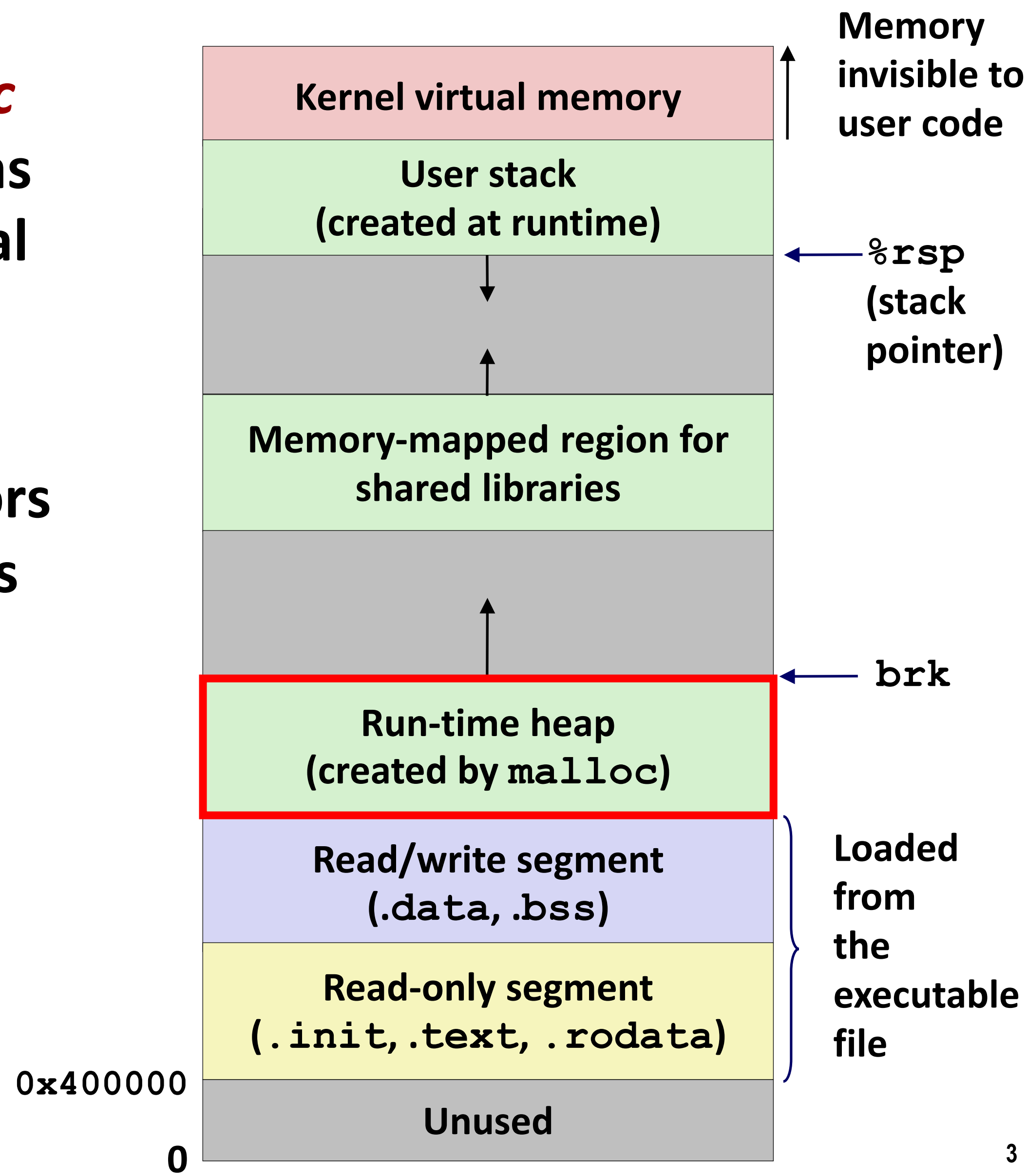
Lecture slides are prepared based on materials provided by CSAPP authors.

Today

- **Basic concepts**
- Implicit free lists

Dynamic Memory Allocation

- Programmers use ***dynamic memory allocators*** (such as `malloc`) to acquire virtual memory (VM) at runtime
- Dynamic memory allocators manage an area of process VM known as the ***heap***



Dynamic Memory Allocation

- **Allocator maintains heap as a set of *blocks*, which are either *allocated* or *free***
- **Types of allocators**
  - ***Explicit allocator*:** application allocates and frees space
    - e.g., **`malloc`** and **`free`** in C
    - e.g., **new** and **delete** operators in C++
  - ***Implicit allocator:*** allocation is explicit, but free is implicit
    - e.g., **Smart pointers** in C++
    - e.g., **garbage collection** in Java

- **Will discuss simple explicit memory allocation today**

The `malloc` Package

`void *malloc(size_t size)`

- Successful:
  - Returns a pointer to a memory block of at least **`size`** bytes aligned to a 16-byte boundary (on x86-64)
  - If **`size == 0`**, returns NULL
- Unsuccessful: returns NULL and sets **`errno`**

`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`**, **`calloc`**, or **`realloc`**

Other functions

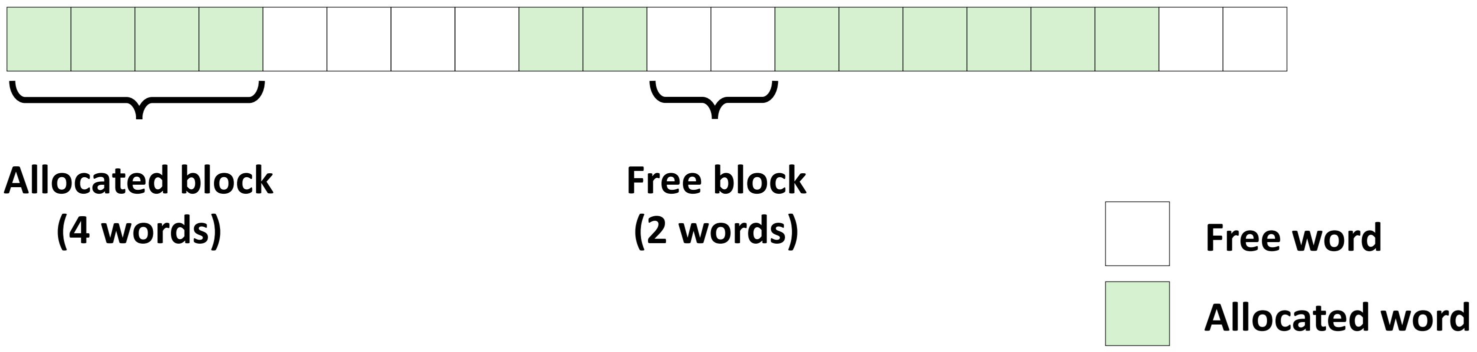
- **`calloc`**: Version of **`malloc`** that initializes allocated block to zero
- **`realloc`**: Changes the size of a previously allocated block
- **`sbrk`**: Used internally by allocators to grow or shrink the heap

malloc Example

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

void foo(long n) {
    long i, *p;

    /* Allocate a block of n longs */
    p = (long *) malloc(n * sizeof(long));
    if (p == NULL) {
        perror("malloc");
        exit(0);
    }

    /* Initialize allocated block */
    for (i=0; i<n; i++)
       p[i] = i;
    /* Do something with p */
    . . .
    /* Return allocated block to the heap */
    free(p);
}
```

Visualization Conventions

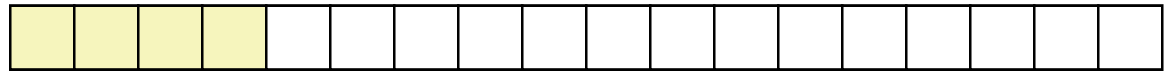
- Show 8-byte words as squares
- Allocations are double-word aligned



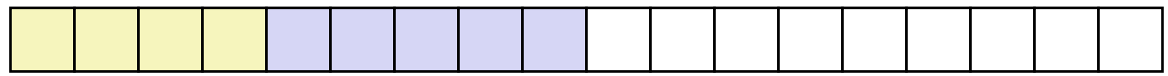
Allocation Example (Conceptual)

```
#define SIZ sizeof(size_t)
```

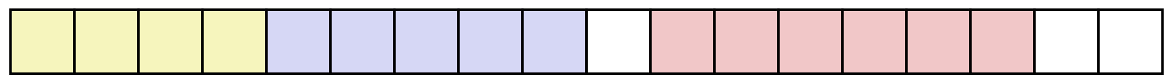
```
p1 = malloc(4*SIZ)
```



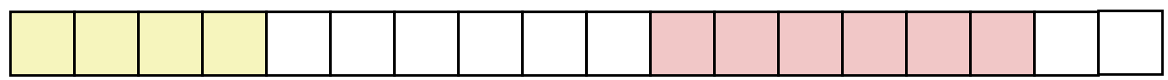
```
p2 = malloc(5*SIZ)
```



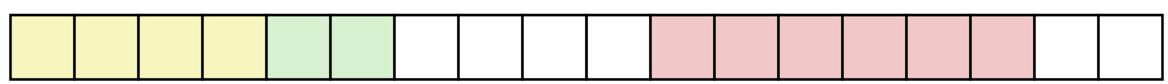
```
p3 = malloc(6*SIZ)
```



```
free(p2)
```



```
p4 = malloc(2*SIZ)
```



Constraints

- **Applications**
  - Can issue arbitrary sequence of **`malloc`** and **`free`** requests
  - **`free`** request must be to a **`malloc`**'d block

- **Requirements**
  - **`malloc`** requests should be served as first-come, first-served
    - *i.e.*, allocator may reorder, but it should not have side-effects
  - Must align blocks so they satisfy all alignment requirements
    - e.g., 16-byte (x86-64) alignment on 64-bit systems
  - Must allocate blocks from free memory
    - Can manipulate and modify only the free memory
    - Q. What would happen if the allocator modifies the allocated memory?
  - Can't move the allocated blocks once they are **`malloc`**'d
    - Q. What would happen if the allocator moves the blocks around?

Performance Goal

- **Goals: maximize throughput and peak memory utilization**
  - These goals are often conflicting

- **Throughput**
  - 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

Performance Goal: Memory Overhead

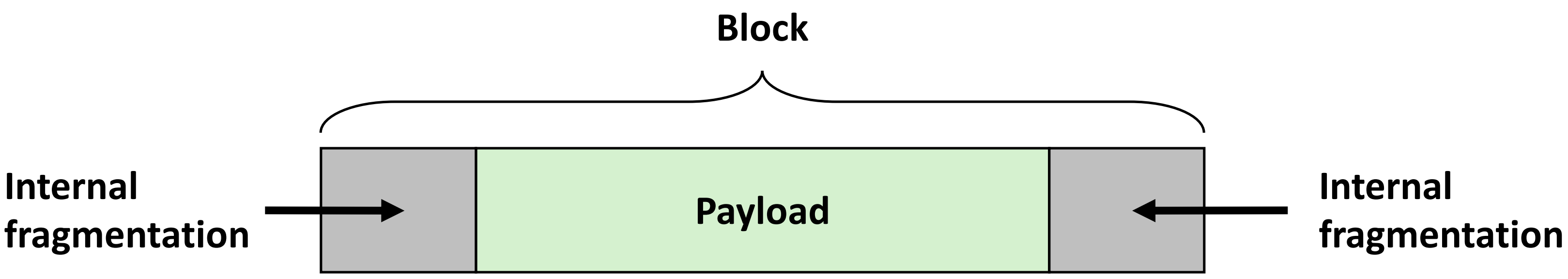
- **Given some sequence of `malloc` and `free` requests:**
  - $R_1, \ldots, R_k, \ldots, R_{n-1}$
- ***After k requests we have:***
- ***Def: Aggregate payload P_k***
  - **`malloc(p)`** results in a block with a ***payload*** of **`p`** bytes
  - The ***aggregate payload*** P_k is the sum of currently allocated blocks
  - The ***peak aggregate payload*** $\max_{i \le k} P_i$ is the maximum aggregate blocks at any point in the sequence up to request
- ***Def: Current heap size H_k***
  - Assume heap only *grows* when allocator uses **`sbrk`**, never shrinks
- ***Def: Memory Overhead, O_k***
  - Peak memory utilization after k requests
  - $O_k = (\max_{i \le k} P_i / H_k)$

Fragmentation

- **Poor memory utilization caused by *fragmentation***
  - ***Internal*** fragmentation
  - ***External*** fragmentation

Internal Fragmentation

- **For a given block, *internal fragmentation* occurs if payload is smaller than block size**



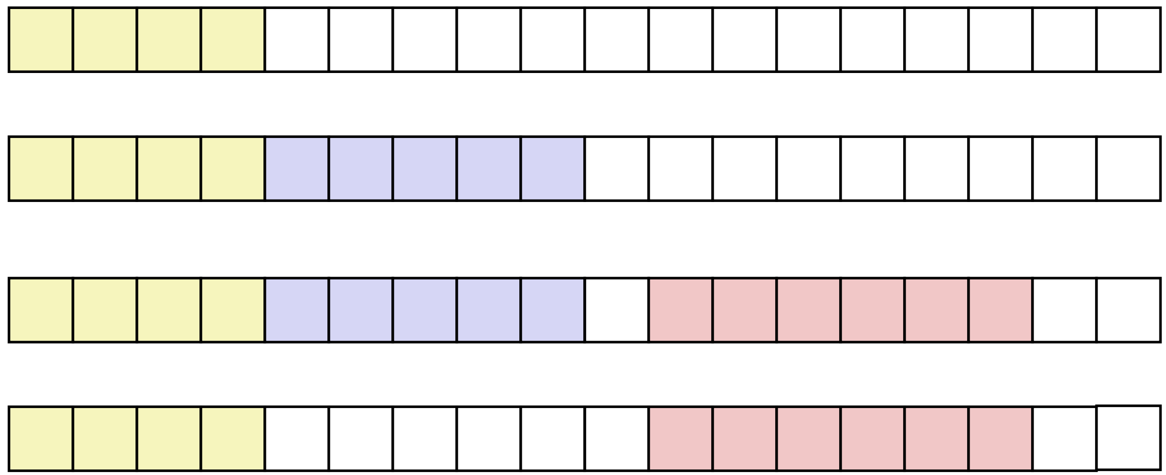
- **Caused by**
  - Overhead of maintaining heap data structures
  - Padding payload for alignment purposes

External Fragmentation

```
#define SIZ sizeof(size_t)
```

- **Occurs when there is enough aggregate heap memory, but no free blocks are large enough**

```
p1 = malloc(4*SIZ)
p2 = malloc(5*SIZ)
p3 = malloc(6*SIZ)
free(p2)
p4 = malloc(7*SIZ)
```



Yikes! (what would happen now?)

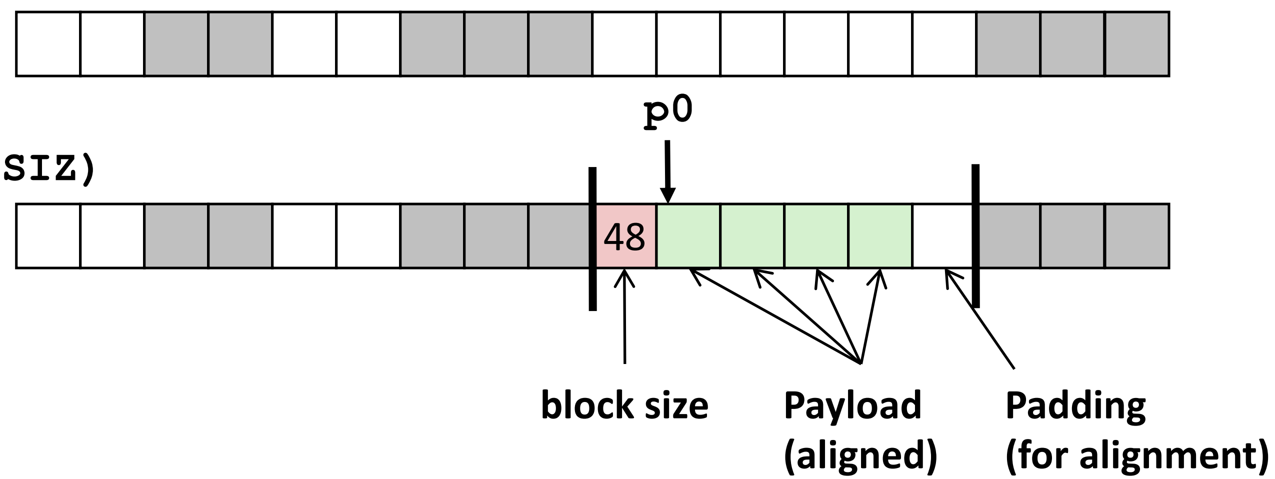
Implementation Issues

- How do we free a memory given just a pointer? You don't know the size of it.
- How do we keep track of all free (or available) blocks?
- How do we pick a block to return for allocation -- many might fit?
- How do we reuse a block that has been freed?

Knowing How Much to Free

- **Standard method**
  - Keep the length (in bytes) of a block in the word *preceding* the block.
    - This word is often called the ***header field*** or ***header***
  - Requires an extra word for every allocated block

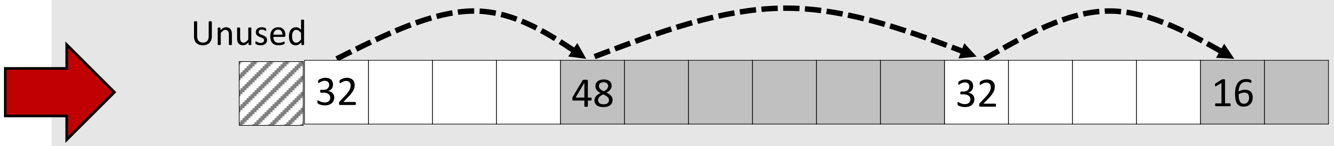
`p0 = malloc(4*SIZ)`



`free(p0)`

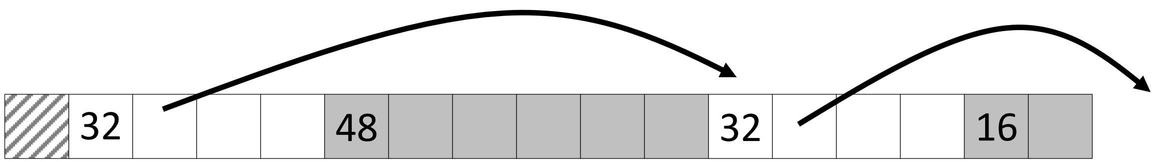
Keeping Track of Free Blocks

- **Method 1: *Implicit list* using length—links all blocks**



Need to tag each block as allocated/free

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



Need space for pointers

- **Method 3: *Segregated free list***
  - 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

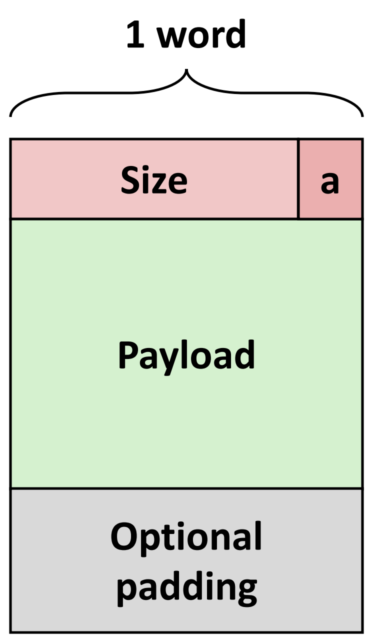
Today

- Basic concepts
- **Implicit free lists**

Method 1: Implicit Free List

- **For each block we need both size and allocation status**
  - Could store this information in two words: wasteful!
- **Standard trick**
  - When blocks are aligned, some low-order address bits are always 0
  - Instead of storing an always-0 bit, use it as an allocated/free flag
  - When reading the Size word, must mask out this bit
  - **Q. The fragmentation here is internal or external? What's the fragmentation ratio?**

Format of allocated and free blocks

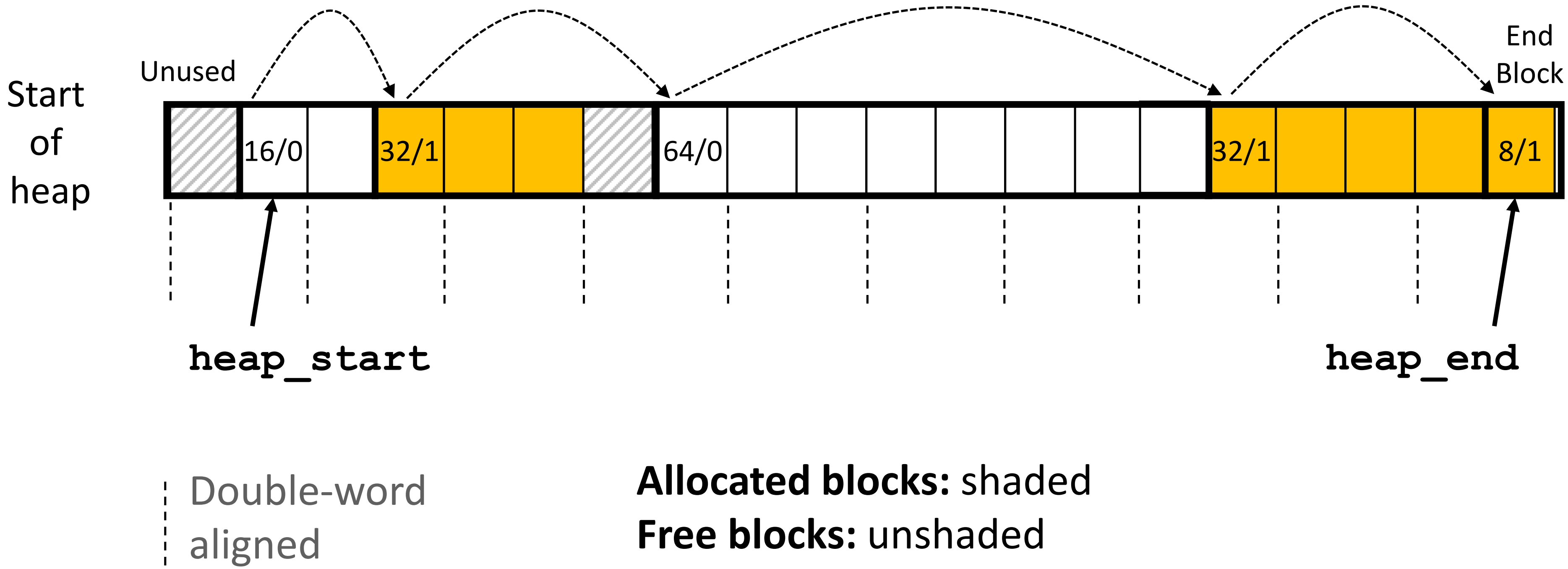


a = 1: Allocated block
a = 0: Free block

Size: total block size

Payload: application data (allocated blocks only)

Detailed Implicit Free List Example



Allocated blocks: shaded
Free blocks: unshaded
Headers: labeled with "size in words/allocated bit"

Note: Headers are at non-aligned positions
➔ Payloads are aligned

Q. Why should you align the memory?

- **#A1. Hardware requirements**
  - Direct memory access (DMA)

> 4.4. Allocating Aligned Memory
>
> When allocating host-side memories that are used to transfer data to and from the FPGA,the memory must be at least 64-byte aligned.
> Aligning the host-side memories allows direct memory access (DMA) transfers to occur to and from the FPGA and improves buffer transfer efficiency.

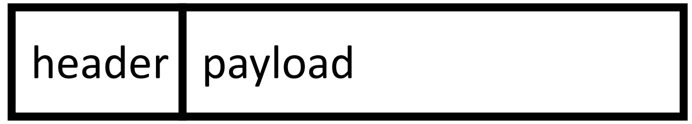
https://www.intel.com/content/www/us/en/docs/programmable/683521/21-4/allocating-aligned-memory.html

- **#A2. SIMD instructions only take the aligned memory**

> `_m256d _mm256_load_pd (double const * mem_addr)`
>
> Synopsis
>
> ```
> __m256d _mm256_load_pd (double const * mem_addr)
> #include "immintrin.h"
> Instruction: vmovapd ymm, m256
> CPUID Flags: AVX
> ```
>
> Description
>
> Load 256-bits (composed of 4 packed double-precision (64-bit) floating-point elements) from memory into dst. mem_addr must be aligned on a 32-byte boundary or a general-protection exception may be generated.

https://community.intel.com/t5/Intel-ISA-Extensions/SSE-and-AVX-behavior-with-aligned-unaligned-instructions/td-p/1170000

Implicit List: Data Structures



- **Block declaration**

```
typedef uint64_t word_t;
```

```
typedef struct block
{
    word_t header;
    unsigned char payload[0];        // Zero length array
} block_t;
```

- **Getting payload from block pointer** // block_t *block

```
return (void *) (block->payload);
```

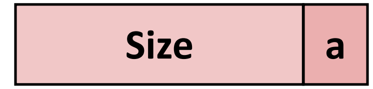
- **Getting header from payload** // p points to a payload

```
return (block_t *) ((unsigned char *) p
                    - offsetof(block_t, payload));
```

C function `offsetof(a,b)` returns offset of member (i.e., b) within struct (i.e., a):

#define offsetof(a,b) ((int)(&(((a*)(0))->b)))

Implicit List: Header access



- Getting allocated bit from header

```
return header & 0x1;
```

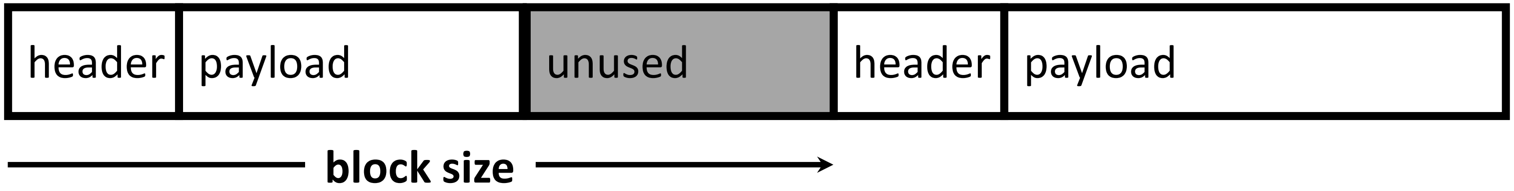
- Getting size from header

```
return (header >> 1) << 1;
```

- Initializing header `//block_t *block`

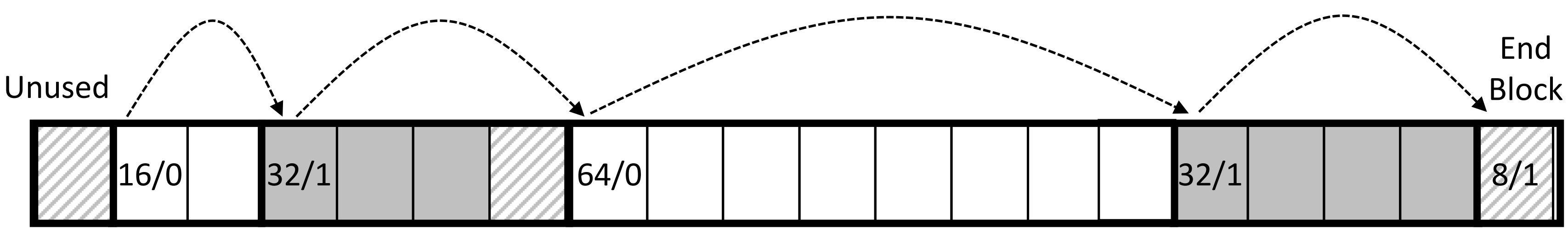
```
block->header = size | alloc;
```

Implicit List: Traversing list



- **Find next block**

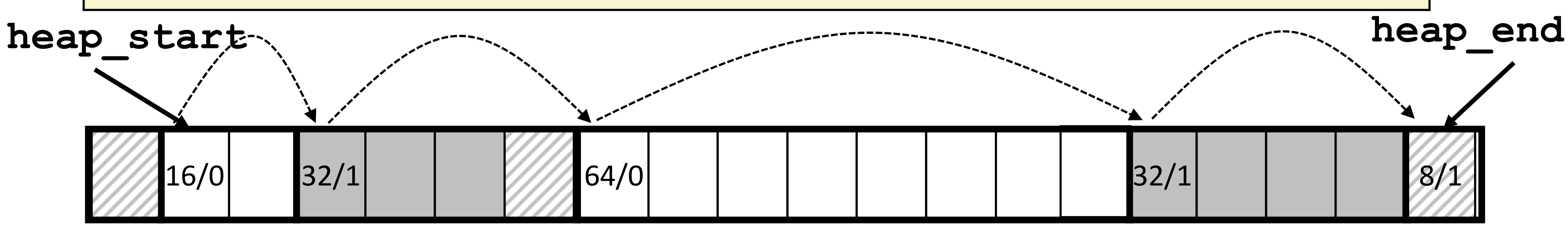
```
static block_t *find_next(block_t *block)
{
    return (block_t *) ((unsigned char *) block
                        + get_size(block));
}
```



Implicit List: Finding a Free Block

- ***First fit:***
  - Search list from beginning, choose ***first*** free block that fits:
  - Finding space for **asize** bytes (including header):

```
static block_t *find_fit(size_t asize)
{
    block_t *block;
    for (block = heap_start; block != heap_end;
         block = find_next(block)) {
    {
      if (!(get_alloc(block))                 // check if free
          && (asize <= get_size(block)))  // check the size
         return block;
    }
    return NULL; // No fit found
}
```



Implicit List: Finding a Free Block

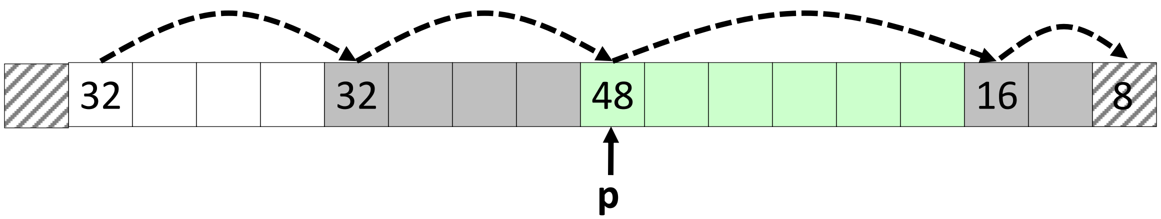
- ***First fit:***
  - Search list from beginning, choose ***first*** free block that fits:
  - Can take linear time in total number of blocks (allocated and free)

- ***Next fit:***
  - Similar to first fit, but search list starting where previous search finished
  - Should often be faster than first fit, assuming no/few frees were performed before
    - Avoids re-scanning unhelpful blocks

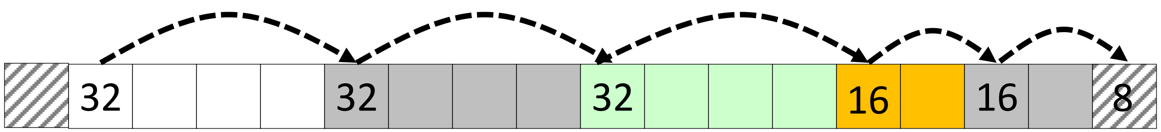
- ***Best fit:***
  - Search the list, choose the ***best*** free block: fits, with fewest bytes left over
  - Keeps (____) fragments small—usually improves memory utilization
  - Will typically run slower than first fit
  - Still a (____) algorithm. No guarantee of optimality
    - We never know the next alloc/free request

Implicit List: Allocating in Free Block

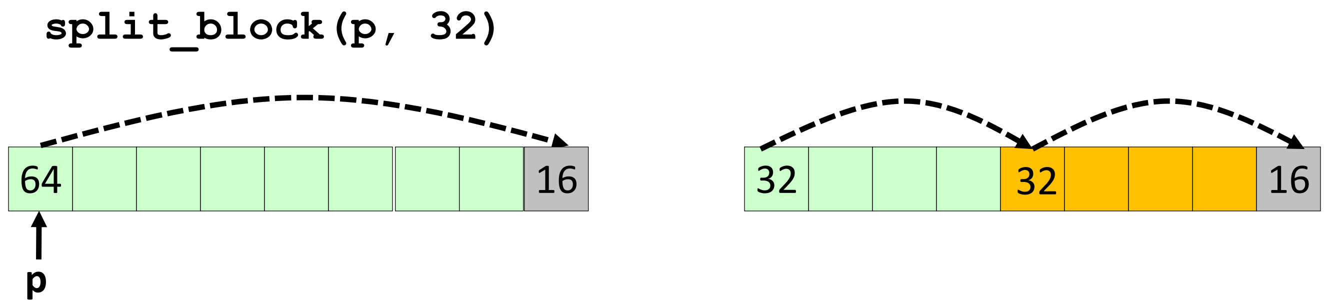
- **Allocating in a free block: *splitting***
  - Since allocated space might be smaller than free space, we might want to split the block



`split_block(p, 32)`



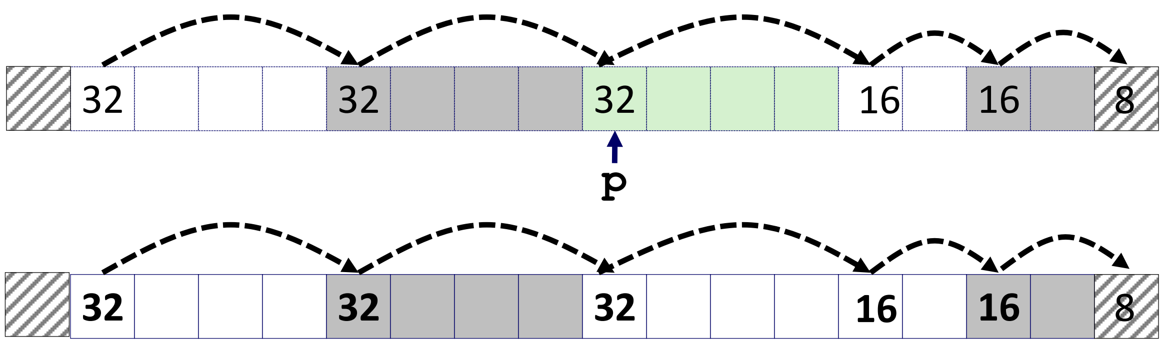
Implicit List: Splitting Free Block




```
// Warning: This code is incomplete

static void split_block(block_t *block, size_t asize){
    size_t block_size = get_size(block);

    if ((block_size - asize) >= min_block_size) {
        write_header(block, asize, true);
        block_t *block_next = find_next(block);
        write_header(block_next, block_size - asize, false);
}
```

Implicit List: Freeing a Block

- **Simplest implementation:**
  - Need only clear the "allocated" flag
  - But can lead to "false (_____) fragmentation"



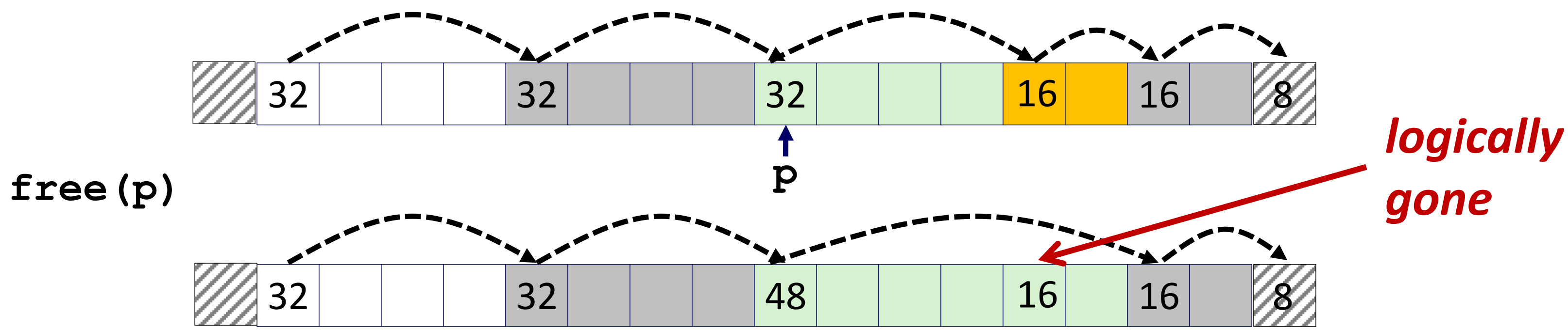
free(p)

malloc(5*SIZ) ***Yikes!***

There is enough contiguous free space (5*SIZ), but the allocator won't be able to find it

Implicit List: Coalescing

- **Join *(coalesce)* with next/previous blocks, if they are free**
  - Coalescing with next block



Summary of Key Allocator Policies

- **Placement policy:**
  - First-fit, next-fit, best-fit, etc.
  - Trades off: throughput Vs. 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 fragmentation are we willing to tolerate?

- **Coalescing policy:**
  - ***Immediate coalescing:*** coalesce each time **`free`** is called
  - ***Deferred coalescing:*** try to improve performance of **`free`** by deferring coalescing until needed.

Implicit Lists: Summary

- **Implementation: very simple**
- **Allocate cost:**
  - linear time worst case
- **Free cost:**
  - constant time worst case
  - even with coalescing
- **Memory Overhead**
  - will depend on placement policy
  - First-fit, next-fit or best-fit
- **Not used for modern allocators because of linear-time allocation**
  - used in many special purpose applications (e.g., embedded systems)
- **However, the concepts of splitting and coalescing are general to *all* allocators**