CS 354 - Machine Organization & Programming
Tuesday Oct 3rd, and Thursday Oct 5th, 2023

Midterm Exam - Thurs, Oct 5th, 7:30 - 9:30 pm

You should have received email with your EXAM INFORMATION including:
DATE, TIME, ROOM, NAME, LECTURE NUMBER, and ID NUMBER,

- **UW ID required.** **Students without UW ID must wait until other students are checked in**
- **Copy or photo of Exam info email**
- **#2 pencils required**
- **closed book, no notes, no electronic devices (e.g., calculators, phones, watches)**
- **see "Midterm Exam 1" on course site Assignments for topics**

Project p2B: Due on or before Friday, Oct 6th

Homework hw2: Due on Monday Oct 2nd (solution available Wed morning)

This Week: Posix `brk` & `unistd.h` C's Heap Allocator & `stdlib.h` Meet the Heap Allocator Design Simple View of Heap	Free Block Organization Implicit Free List Placement Policies **MIDTERM EXAM 1**
Next Week: Dynamic Memory Allocator options Read for next week: B&O 9.9.7 Placing Allocated Blocks 9.9.8 Splitting Free Blocks 9.9.9 Getting Additional Heap Memory 9.9.10 Coalescing Free Blocks	9.9.11 Coalescing with Boundary Tags 9.9.12 Putting It Together: Implementing a Simple Allocator 9.9.13 Explicit Free Lists 9.9.14 Segregated Free Lists

Posix `brk` & `unistd.h`

What? `unistd.h` contains a collection of system call wrappers

Posix API (Portable OS Interface) std for maintaining compatibility

DIY Heap via Posix Calls

break
brk "program break" ptr to end of program
at top of heap

```
int brk(void *addr)
```

Sets the top of heap to the specified address `addr`.
Returns 0 if successful, else -1 and sets `errno`. initially clears new pages for security

\+ bigger
\- smaller

safer
```
void *sbrk(intptr_t incr)
```

Attempts to change the program's top of heap by `incr` bytes.
Returns the old brk if successful, else -1 and sets `errno`.

`errno` is set by O.S. functions to communicate error

#include <errno.h>

printf("Error %s\n", strerror(errno));

convert error # to string

✵ *For most applications, it's best to use malloc/calloc/realloc/free*

b/c the C std is efficient ∴ well tested

✵ ***Caveat: Using both malloc/calloc/realloc and break functions above***

is undefined

C's Heap Allocator & `stdlib.h`

What? **`stdlib.h`** contains a collection of ~ 25 common C func.

- conversion : atoi , strtol
- execution flow : abort , exit
- math : abs
- searching : bsearch
- sorting : qsort
- random numbers : random , srandom
  rand srand

, seed = 1 , seed

C's Heap Allocator Functions

`void *malloc(size_t size)` void *

Allocates and returns generic ptr to block of heap memory of `size` bytes, or returns `NULL` if allocation fails.

`void *calloc(size_t nItems, size_t size)`

Allocates, clears to 0, and returns a block of heap memory of `nItems * size` bytes, or returns `NULL` if allocation fails.

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

Reallocates to `size` bytes a previously allocated block of heap memory pointed to by `ptr`, or returns `NULL` if reallocation fails.

```
if (ptr == NULL) return malloc ( size)
else if (size == 0) { free (ptr) ; return NULL ; }
else // attempt reallocate
```

`void free(void *ptr)`

Frees the heap memory pointed to by `ptr`. If `ptr` is `NULL` then does nothing.
can't communicate an error

For CS 354, if malloc/calloc/realloc returns NULL , exit (1)

Meet the Heap

What? The heap is

- a segment of process VAS used for dynamically allocated memory

  <u>*dynamically allocated memory*</u>: is memory required while prog. is running

- a collection of various size memory block managed by the allocator

  <u>*block*</u>: contiguous chunk of memory

  <u>*payload*</u>: part useable by process

  <u>*overhead*</u>: part of block used by allocator to manage

  <u>*allocator*</u>: CODE that allocates and frees block

Two Allocator Approaches

1. Implicit: Java and Python
   - "new" operator - implicitly det. # of bytes needed
   - garbage collector - " " unused bytes and frees them

2. Explicit: C
   - malloc must be explicitly told how much memory
   - free " " " called to free the heap block

Allocator Design

Two Goals

1. maximize *throughput* # of malloc and free s handled (time)

   more is better

   free (O(1)) constant

   malloc (O(n)) where n = # heap block

2. maximize *memory utilization*

   memory requested / heap alloc

   more is better

Trade Off: increasing one decreases the other

Requirements

→ List the requirements of a heap allocator.

1. allocs use the heap
2. provide immediate response
3. must handle arbitrary sequence of events
4. must not move or change prev. alloc block
5. must follow mem. alignment requirements

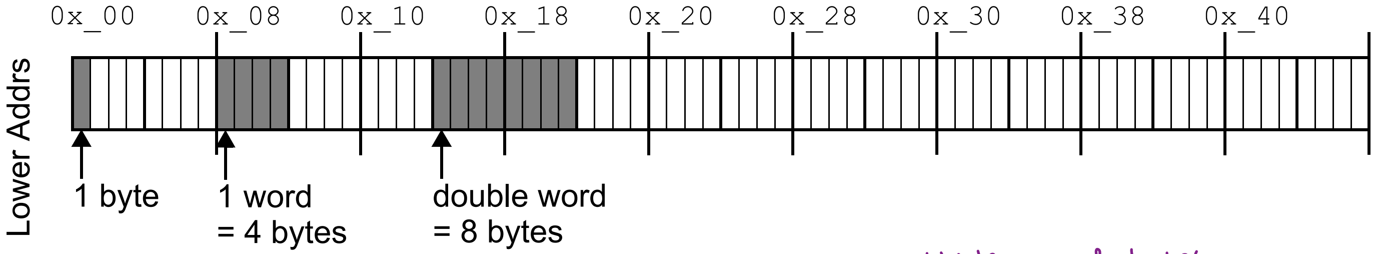
Design Considerations

- "free block" organize
- placement policy first fit, next fit, best fit FF, NF, BF
- "splitting" free blocks to create better fit
- "coalesce" adjacent free blocks

Simple View of Heap

Do NOT USE

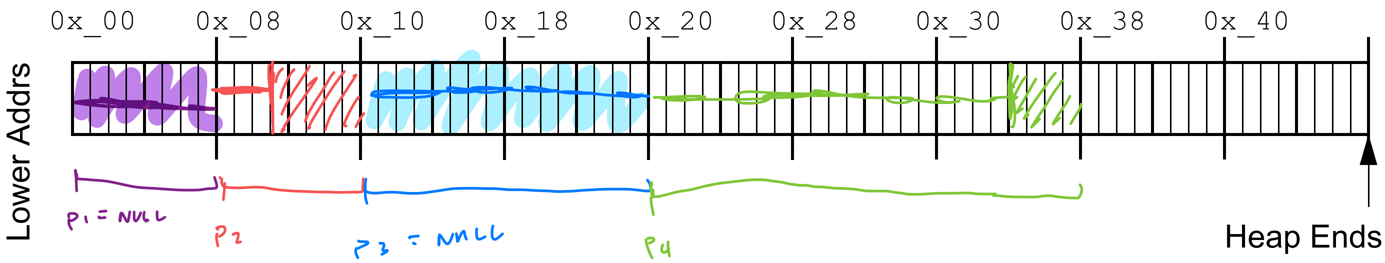
Rotated Linear Memory Layout



double word alignment: 1) block size must be multiple of 8 bytes

2) payload address must be double word aligned

Run 1: Simple View of Heap Allocation



→ Update the diagram to show the following heap allocations: payload addr

```
1) p1 = malloc(2 * sizeof(int));
2) p2 = malloc(3 * sizeof(char));
3) p3 = malloc(4 * sizeof(int));
4) p4 = malloc(5 * sizeof(int));
```

1) // 8 + 0 padding 0x_00

2) // 3 + 5 0x_08

3) // 16 + 0 0x_10

4) // 20 + 4

→ What happens with the following heap operations:

```
5) free(p1); p1 = NULL;
6) free(p3); p3 = NULL;
7) p5 = malloc(6 * sizeof(int));
```

7) // 24 + 0 alloc fails

External Fragmentation: when there is enough heap memory but divided into blocks that are too small

Internal Fragmentation: when a block is used for overhead (i.e "padding")

➢ Why does it make sense that Java doesn't allow primitives on the heap?

lots of wasted space

Free Block Organization

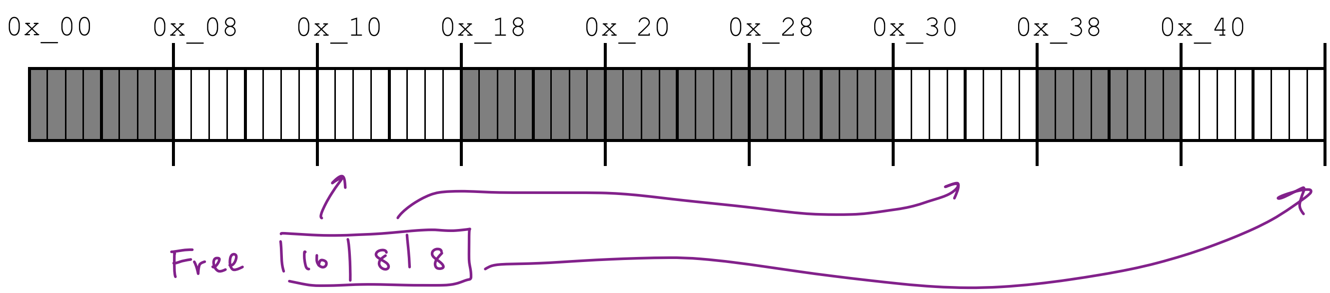
✵ *The simple view of the allocator has* no way to determine size and status of each block

<u>*size*</u> # of bytes in a block (payload + overhead)

<u>*status*</u> whether block is alloc'ed free

Explicit Free List

- keeps data structure w/ list of free block



code: only needs to track size

space: potentially more space required

time: a bit faster, only search free blocks

Implicit Free List

- use heap block to track size and status

code: must track size and status, and check each block

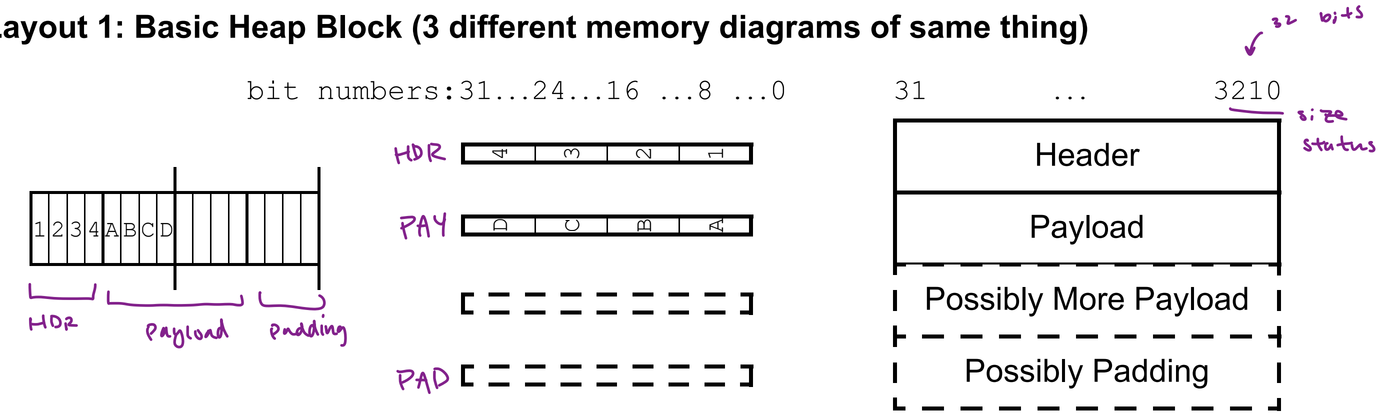
space: potentially less memory required

time: more time required to skip alloc'd blocks

Implicit Free List

❋ *The first word of each block* is a header

Layout 1: Basic Heap Block (3 different memory diagrams of same thing)



❋ *The header stores*

→ Since the block size is a multiple of 8, what value will the last three header bits always have?

8 16 24 0 0 0
1000 100000 11000 status allowed

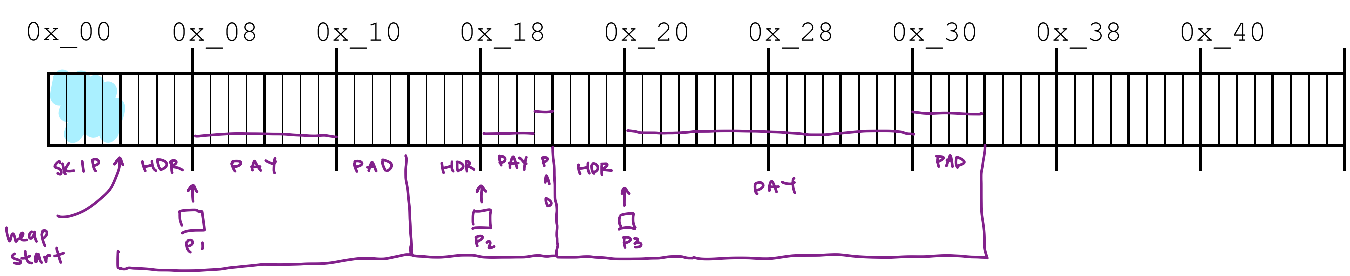
→ What integer value will the header have for a block that is:

allocated and 8 bytes in size? 1000 +1 = 1001 = 9

free and 32 bytes in size? 100000 = 32

allocated and 64 bytes in size? 1 000 000 +1 = 1000 001 = 65

Run 2: Heap Allocation with Block Headers



→ Update the diagram to show the following heap allocations:

```
1) p1 = malloc(2 * sizeof(int));
2) p2 = malloc(3 * sizeof(char));
3) p3 = malloc(4 * sizeof(int));
4) p4 = malloc(5 * sizeof(int));
```

1) 8 + 4 (HDR) = 12 +4 (PAD) = 16
2) 3+4 = 7 + 1 = 8
3) 16 +4 = 20 + 4 = 24
4) 20 +4 = 24 Alloc FAILS

→ Given a pointer to the first block in the heap, how is the next block found?

(void *) ptr (HDR *) + current. block. size

Placement Policies

What? *Placement Policies* are algorithms used to det. which free block

Assume the heap is pre-divided into various-sized free blocks ordered from smaller to larger.

- **First Fit** (FF): start from beginning of heap
  stop at first block that's big enough
  fail if reach END_MARK

  mem util: likely to choose block close to desired size

  thruput: must step through mem blocks to find larger size

- **Next Fit** (NF): start from block most recently alloc'd
  stop at first block big enough
  fail if reach first block checked (wrap around)

  mem util: may chose block that is too large

  thruput: faster , don't have to look at all blocks that were just alloc'd

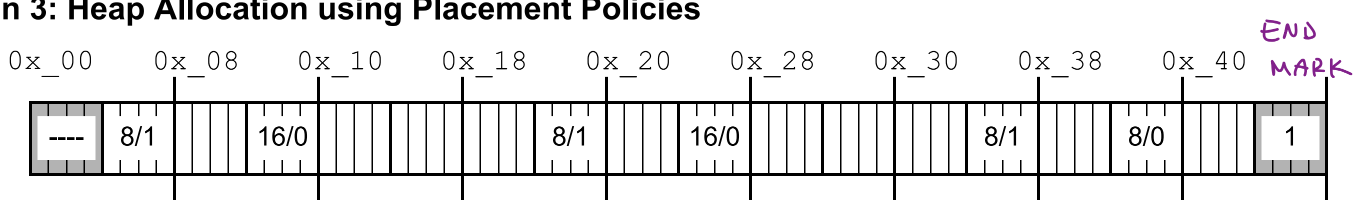
- **Best Fit** (BF): start from beginning of heap
  stop at end mark and choose best fit (closest to req. size)
  or stop early if exact match
  fail if no block is big enough

→

\+ \+ mem util: closest to best size

\- \- thruput: slowest in general, must search entire heap in worst case (usually)

Run 3: Heap Allocation using Placement Policies



→ Given the original heap above and the placement policy, what <u>address is `ptr` assigned</u>?

```
ptr = malloc(sizeof(int));          //FF?  0x__10     BF?  0x___40
ptr = malloc(10 * sizeof(char));    //FF?  0x__10     BF?  0x__10
```

→ Given the original heap above and the <u>address of block</u> most recently allocated, what <u>address is `ptr` assigned</u> using NF?

```
ptr = malloc(sizeof(char));       //0x_04?  0x__10     0x_34?  0x__40
ptr = malloc(3 * sizeof(int));    //0x_1C?  0x__28     0x_34?  0x__10
```