CS 354 - Machine Organization & Programming
Tuesday Feb 20th, and Thursday Feb 22nd, 2024

Midterm Exam - Thurs, Feb 22nd, 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**

A05 submit copy of e1_cheatsheet.pdf to your activities directory

PM BYOL: Exam Review

Project p2B: Due on or before Friday, Feb 23rd

Homework hw2: Due on Monday 2/19 (solution available Wed morning)

This Week: Linux: Proceses and Address Spaces 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 POSIX API functions

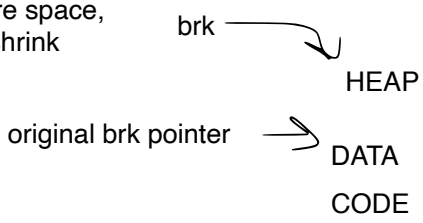
Posix API (Portable OS Interface) standard for maintaining compatibility among Unix OS's

POSIX API functions

DIY Heap via Posix Calls

brk shifts up if need more space, usually only grows not shrink

brk points to the end of the program




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

Sets the top of heap to the specified address `addr`.
Returns 0 if successful, else -1 and sets `errno`.

not common to use this, hard to tell exactly what addr

```
void *sbrk(intptr_t incr)  //intptr_t is sizeof long for ptr addr
```

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

figure out how much space we need, and shifts brk up accordingly

`errno`

```
#include <errno.h>
printf("Error: %s\n", strerror(errno));
```

errno overwritten if got new error
errno field only one error at a time

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

  because C libraries are efficient and well-tested

- ***Caveat: Using both malloc/calloc/realloc and break functions above*** *results in undefined program behavior.*

  UB

C's Heap Allocator & `stdlib.h`

dynamically allocated

What? **`stdlib.h`** contains a collection of ~25 commonly used functions

- conversion utilities: atoi and string to long
- execution flow: skip right to exit eg exit(1), skip flow and exit program
- math functions: absolute, floor, ceiling
- search function: linear search and binary search
- sorting functions: qsort (quick sort)
- random number: rand

C's Heap Allocator Functions

all return void pointer and makes space on heap

```
void *malloc(size_t size)
```

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

```
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. args are number of items then size of each item

```
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. pointer to heap we already have

if size is smaller than current -> no problem
make block bigger if can without interfering with other blocks,
if cant it will call malloc if cant make it bigger,
return NULL if malloc fails too

```
void free(void *ptr)
```

Frees the heap memory pointed to by `ptr`. If `ptr` is `NULL` then does nothing.

❋ ***For CS 354, if malloc/calloc/realloc returns NULL***
just exit the program with an appropriate error message.

Meet the Heap

What? The heap is segment of Virtual Address Space used for dynamically allocated memory

- *dynamically allocated memory*: requested while program is running

- a collection of various sized memory blocks

  block: a contiguous chunk of memory that contains payload, overhead, and allocator

  payload: part requested and available to user

  overhead: part used by allocator to manage (keep track) of allocation

  allocator: the code that allocates and frees

Two Allocator Approaches

1. Implicit: JAVA Approach
   - "new" operator implicitly computing bytes needed
   - gc "Garbage Collector" —implicitly keeps track of what to free, when to free memory

2. Explicit: C Approach
   - malloc - must be explicitly told the number of bytes
   - free - must be explicitly called to free up the memory

Allocator Design

Two Goals request for bytes in a certain order, just going to allocate the space

1. maximize ***throughput*** number of malloc + frees that be handled per time interval

   higher is better malloc -- more likely to be O(N) where N is the number of blocks

   free -- more likely to be O(1)

2. maximize ***memory utilization*** % of memory used, that is used for payload

   memory requested (bytes) / heap that has been allocated (bytes)

Trade Off: increasing one, decreases the other

Requirements

→ List the requirements of a heap allocator.

1. User uses allocs and allocs use heap
2. provide immediate response
3. must handle arbitrary sequence — first come first serve handle request, arbitrary — dont know what sequence will be
4. must not move or change previous allocation
5. must follow memory alignment requirements

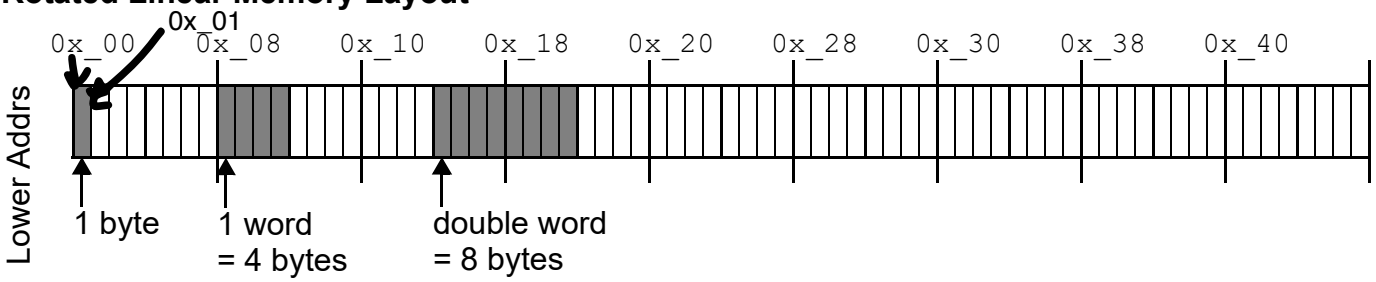
Design Considerations

- "free block" organisation — cant change which blocks empty, but can keep track
- placement policy — First Fit, Next Fit, Best Fit — no VIPs but can make requests in different areas
- "splitting" free blocks — isnt free as need memory to keep track
  but probably worth it to improve memory utilization
- "coalescing" free blocks — satisfy larger requests

Simple View of Heap

DOES NOT WORK because no tracking information

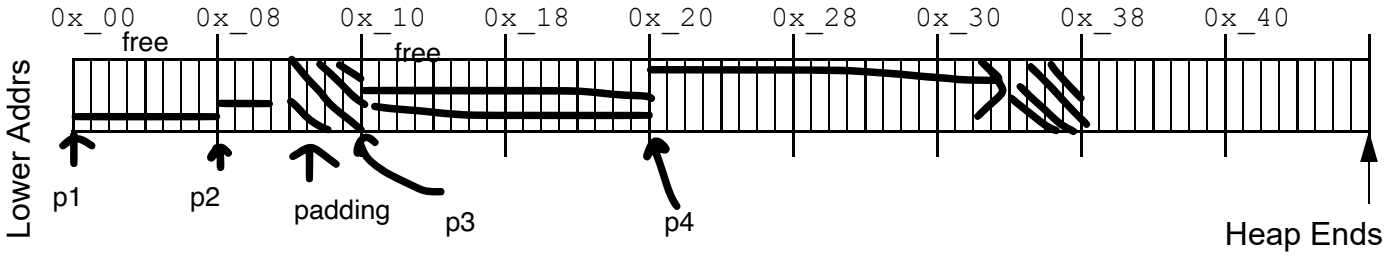
Rotated Linear Memory Layout



double word alignment: 1) Block sizes must be multiples of 8

2) Payload address must be multiple of 8

Run 1: Simple View of Heap Allocation



→ 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));
```

8 bytes

3 + 5

// 16

// 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));
```

// 24

ALLOC FAIL, not enough space on heap, returns NULL
though total free space is 8 + 16 + 20

External Fragmentation:

enough free heap memory divided into blocks that are each too small
happens because sequence of calls

Internal Fragmentation:

when memory used in the block is used for overhead

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

wastes space — very small compared to other objects, and have min requirements for size here

Free Block Organization

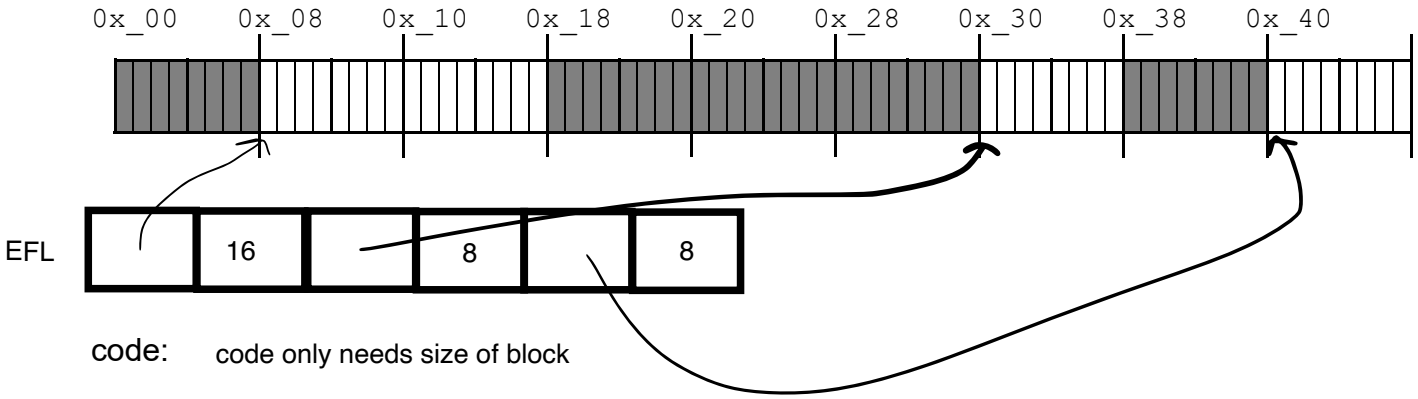
❋ *The simple view of the allocator has* no way to determine size and status of block

size number of bytes in block

status whether block is allocated or freed

Explicit Free List EFL can be external or internal

- allocator maintains data structure with free blocks



code: code only needs size of block

-ve space: potentially more memory request for data structs

+ve time: a bit faster to search only free block for size

Implicit Free List just look at all blocks instead of keeping list of free blocks

- allocator uses heap blocks as data structure

code: must track size and status of each block

+ve space: potentially less memory

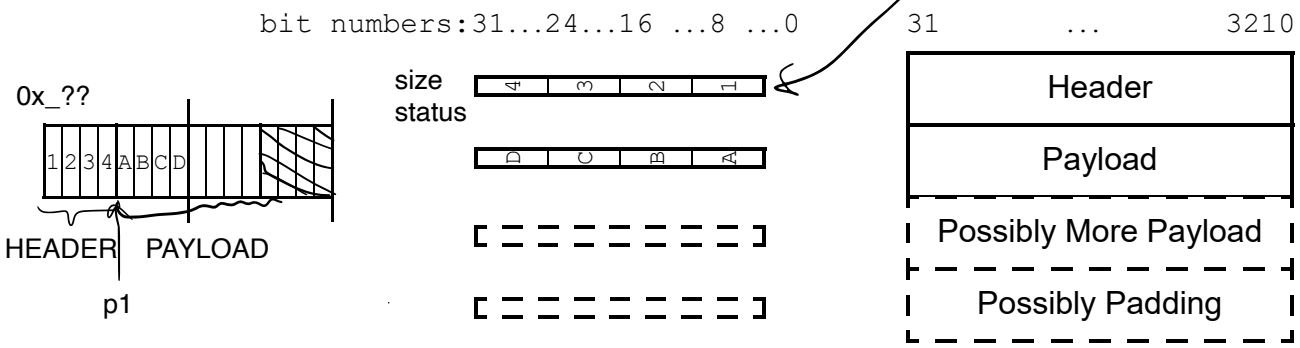
-ve time: more time to search alloc'd and free blocks

Implicit Free List

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

size multiple of 8,
all of last 3 bits 0

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



❋ *The header stores* both size and status as a single integer

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

8 0000 1000
16 0001 0000
32 0010 0000
64 0100 0000

last 3 bits does not tell size

8 + 1 = 9 or 8/1 alloc'd

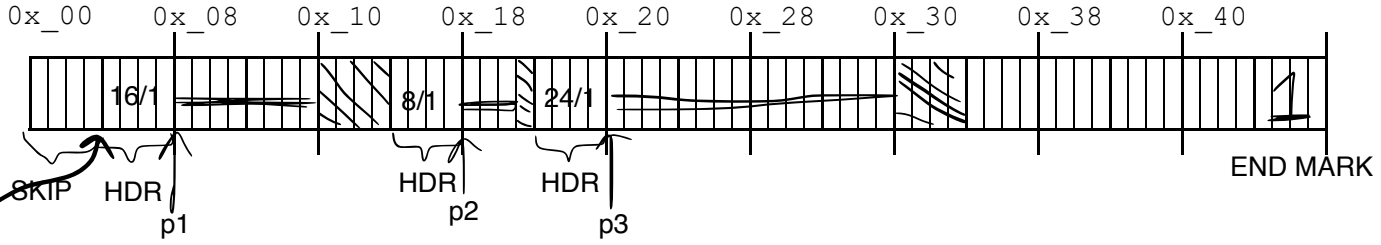
16/0 free

a-bit = alloc'd if 1

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

allocated and 8 bytes in size?	8/1	0000 1001 = 9
free and 32 bytes in size?	32/0	0010 0000 = 32
allocated and 64 bytes in size?	64/1	0100 0001 = 65

Run 2: Heap Allocation with Block Headers



* payload pointer must be at multiple of 8

→ Update the diagram to show the following heap allocations:

```
1) p1 = malloc(2 * sizeof(int));
```
2 * 4 = 8, then 8 + 4(HDR) = 12 then (since not mult of 8) + 4 (PAD) = 16

```
2) p2 = malloc(3 * sizeof(char));
```
(3 + 4) + 1 = 8

```
3) p3 = malloc(4 * sizeof(int));
```
(16 + 4) + 4 = 24

```
4) p4 = malloc(5 * sizeof(int));
```
(20 + 4) + 0 = 24 (no space)

Fblock

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

add 16 (the size) (void *) Fblock + "curr_size"

Placement Policies

What? *Placement Policies* are algorithms used to determine which free block is used

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 free block that is big enough
  fail if reach END MARK

+ve mem util: likely to choose desired size

-ve thruput: must step through many allocated blocks

- **Next Fit** (NF): start from block most recently allocated
  stop at first free block that is big enough
  fail if reach block started with (since wrap from end to start)

-ve mem util: not as good, may choose large block for small request

+ve thruput: faster, get to skip many previously alloc'd blocks

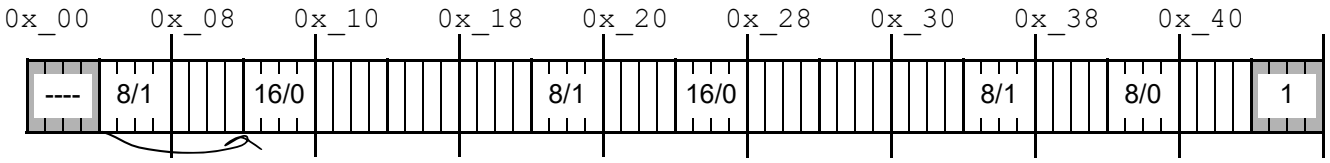
- **Best Fit** (BF): start from beginning of heap
  stop at END MARK and choose best fit if found on the way
  or stop early exact size match is found
  fail if no block found is big enough

→

+ve mem util: closest to best size for each request

-ve thruput: slowest in general

Run 3: Heap Allocation using Placement Policies



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

`ptr = malloc(sizeof(int));` (4 + 4) + 0 = 8 `//FF?` payload at 0x10 `BF?` payload ay 0x40

`ptr = malloc(10 * sizeof(char));` (10 + 4) + 2 = 16 `//FF?` payload at 0x10 `BF?` payload at 0x10

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

`ptr = malloc(sizeof(char));` (1 + 4) + 3 = 8 `//0x_04?` payload at 0x10 `0x_34?` payload at 0x40

`ptr = malloc(3 * sizeof(int));` (12 + 4) + 0 = 16 `//0x_1C?` payload at 0x28 `0x_34?` payload at 0x10