**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

## DIY Heap via Posix Calls *brk*



int brk(void \*addr) points to end of the program



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



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.

**Errno #include <errno.h>**

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

C libs are efficient and well tested

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

## C’s Heap Allocator & stdlib.h

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

 Conversion Utilities

 Execution flow

 Math functions

 Searching

 Sorting

 Random number: rand

### C’s Heap Allocator Functions

void \*malloc(size\_t size)

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.

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 VAS used for dynamically allocated memory



*dynamically allocated memory*: is requested as your program is running

 A collection of varius sized memory blocks

*block*: a contiguous chunk of memory that contains:

*payload*: part req and available to user

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

*allocator*: the code that alloc + frees

### Two Allocator Approaches

1. Implicit: JAVA

 “new” operator implicitly computing bytes needed

 gc “Garbage Collector” -implicitly determines when to free memory

1. Explicit: C’s

 malloc – must be explicitly told the number of bytes

 free – must be explicitly called to free memory

**Allocator Design**

### Two Goals

1. maximize *throughput # of malloc + frees that can be handled in an amount of time*

higher is better || malloc O(N) N = # of blocks || free O(1)

1. maximize *memory utilization percent of memory used that is used for payload*

*mem req(bytes) / heap allocated(bytes)*

Trade Off: increase in one, leads to a decrease in the other

### Requirements

 List the requirements of a heap allocator.

1. alloc’s use heap

2. provide an immediate response

3. must handle arbitrary sequence

4. must not move or change prev alloc

5. must follow memory alignment requirements

### Design Considerations

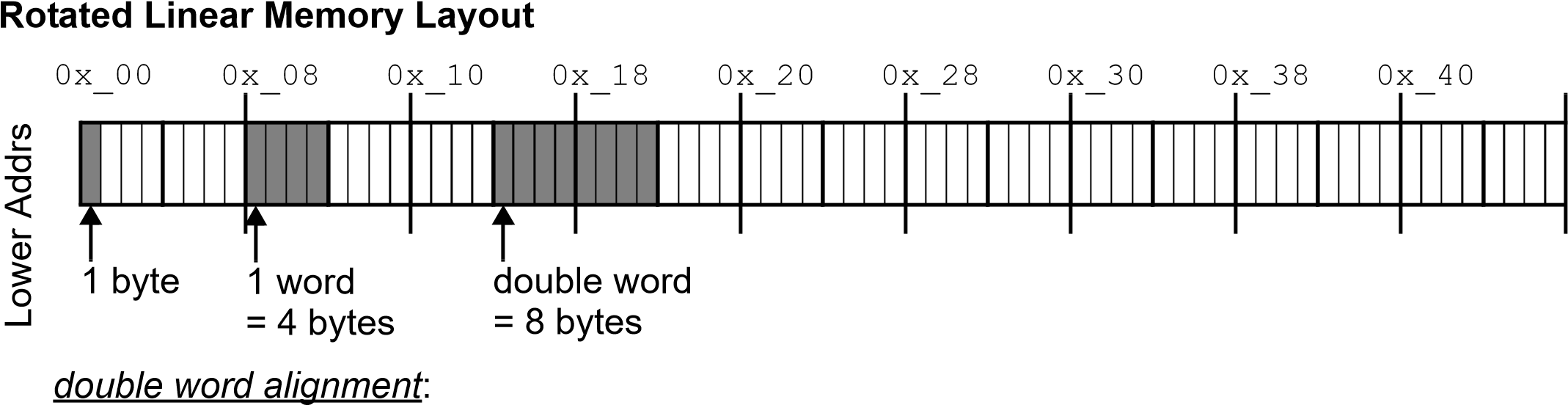
 “free block” organization

 placement policy: FF NF BF

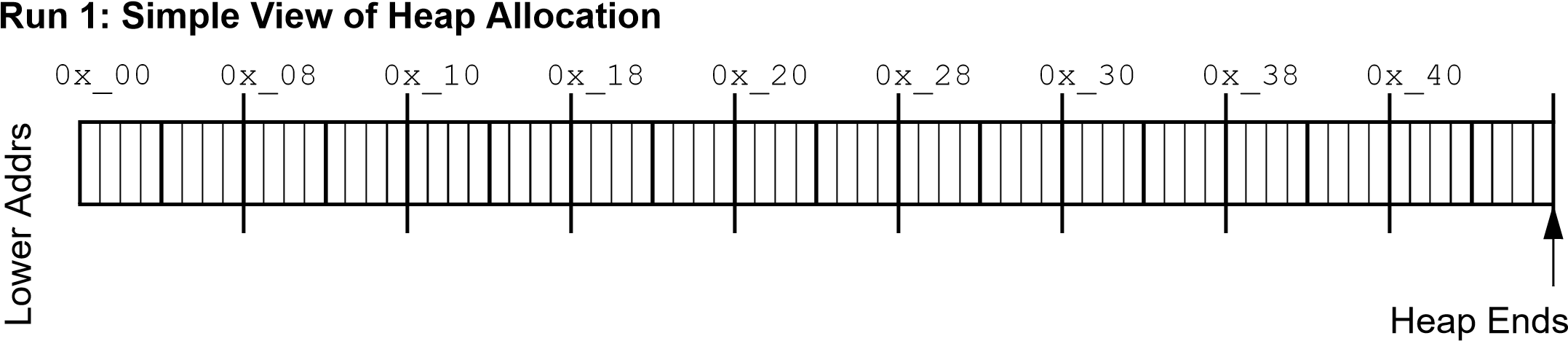
 “splitting” free blocks, to improve memory utilization

 “coalesing” freed blocks to satisfy larger requests

## Simple View of Heap DOES NOT WORK



1. block sizes must be multiples of 8 2) payload addr must be multiple of 8



Free

Free



* Update the diagram to show the following heap allocations:
  1. p1 = malloc(2 \* sizeof(int)); //8



* 1. p2 = malloc(3 \* sizeof(char)); // 3+5



* 1. p3 = malloc(4 \* sizeof(int));//16 + 0



* 1. p4 = malloc(5 \* sizeof(int));//20 +3



Padding is the +5 and +3 to make it multiple of 8

* What happens with the following heap operations:
  1. free(p1); p1 = NULL;
  2. free(p3); p3 = NULL;
  3. p5 = malloc(6 \* sizeof(int));//24 ALLOC FAIL not enough space

*External Fragmentation*:

*When there is enough heap memory but it is divided into blocks that are too small*

*Internal Fragmentation*: When memory in a block is used for overhead

 Why does it make sense that Java doesn’t allow primitives on the heap? VERY SMALL

## Free Block Organization

 *The simple view of the allocator has no way to determine size or status of block*

*Size Number of bytes in the block*

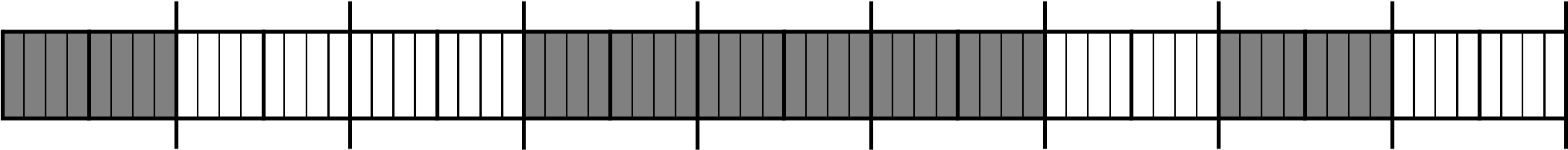
*Status*

*Whether block is allocated or free*

### Explicit Free List

 allocator maintains data structure with free blocks

0x\_00 0x\_08 0x\_10 0x\_18 0x\_20 0x\_28 0x\_30 0x\_38 0x\_40



code only needs size of block

E.F.L

16

8

8



code:

Potentially more mem. Req for data structure

space:

A bit faster to search only tree block for size

time:

### Implicit Free List

 Allocator uses heap blocks as its data structure

Must track size and status of each block

code: space: time:

More time required to search alloced and freed blocks

Potentially less memory req

## Implicit Free List

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

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

bit numbers:31...24...16 ...8 ...0 31 ... 3210

0x\_??

Payload

HDR



A

D

C

B

Header

Payload

Possibly More Payload

Possibly Padding

1

2

3

4

D

C

B

A

3

1

2

4

 *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?A white paper with blue writing on it

  Description automatically generated
* What integer value will the header have for a block that is:

allocated and 8 bytes in size? 8/1 🡨 0001001 = 9

32/0 🡨 000100000 = 32

free and 32 bytes in size? allocated and 64 bytes in size?

64/0 🡨 001000001 = 65

### Run 2: Heap Allocation with Block Headers

0x\_00 0x\_08 0x\_10 0x\_18 0x\_20 0x\_28 0x\_30 0x\_38 0x\_40

End Mark

1



SKIP

HDR



* Update the diagram to show the following heap allocations: Size and pointer must be multiple of 8
  1. p1 = malloc(2 \* sizeof(int)); (8 + 4 (HDR)) +4(Pad) = 16 (Size)



* 1. p2 = malloc(3 \* sizeof(char)); (3 + 4) + 1=8



* 1. p3 = malloc(4 \* sizeof(int)); (16+4) + 4 = 24



* 1. p4 = malloc(5 \* sizeof(int)); (20 + 4) + 0 = 24 Alloc Fails



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

Pointer to first block = ptr to header

(void \*) Fblock + “curr\_size”

## Placement Policies

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

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

Beginning of the heap

* **First Fit** (FF): start from stop at

First free block big enough

Reach END mark

fail if

Likely to choose desired size

mem util:

Must step through many alloc blocks

thruput:

Block most recently allocd

* **Next Fit** (NF): start from stop at

First free block that’s big enough

Reach block started with

fail if

Not as good, may choose large block for small request

mem util:

Faster, skip prev alloc block

thruput:

Beginning of the heap

* **Best Fit** (BF): start from stop at

End Mark and choose best fit found on the way

If Exact size match is found

or stop early

No block is found that’s big enough

fail if

mem util:

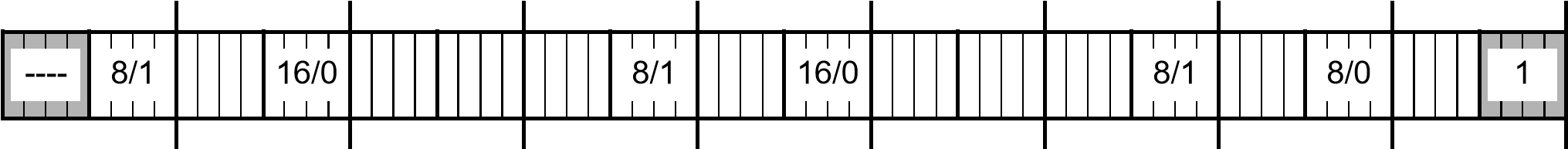
Slowest in general

Closest to best size for each req

thruput:

### Run 3: Heap Allocation using Placement Policies

0x\_00 0x\_08 0x\_10 0x\_18 0x\_20 0x\_28 0x\_30 0x\_38 0x\_40



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

ptr = malloc(sizeof(int)); 4(Payload) + 4(Header) = 8

//FF? 0x\_10 BF? 0x\_40

ptr = malloc(10 \* sizeof(char)); 10(Pay)+4(Hdr)+2(Pding) = 16

//FF? 0x\_10 BF? 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(pay)+4(hdr)+3(Pding)=8 |  |  |
| //0x\_04? 0x\_10  //0x\_34? 0x\_40 |  |  |
| ptr = malloc(3 \* sizeof(int));  12(pay)+4 = 16 |  |  |
| //0x\_1C? 0x\_28  //0x\_34? 0x\_10 |  |  |