CS 354 - Machine Organization & Programming
Tuesday Feb 21, and Thursday Feb 23, 2023

Midterm Exam - Thurs, February 23th, 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, February 24th

Homework hw2: Due on Monday February 20st (solution available Wed morning)

Last Week: Standard & String I/O in `stdio.h` File I/O in `stdio.h` Copying Text Files Meet Globals and Static Locals	C's Abstract Memory Model Where Do I Live? Three Faces of Memory Virtual Address Space Linux: Processes and Address Spaces

This Week: Linux: Processes 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: The Heap & Dynamic Memory Allocators 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 (Portable OS Interface) standard for maintaining compatibility among Unix OS's

DIY Heap via Posix Calls

brk "program break" - pointer 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`.

```
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`

set by OS functions to communicate a specific error

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

❋ ***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 C functions

-
-
-
-
-
-

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

-

  dynamically allocated memory:

-

  block:

  payload:

  overhead:

  allocator:

Two Allocator Approaches

1. Implicit:
   -
   -
2. Explicit:
   -
   -

Allocator Design

Two Goals

1. maximize *throughput*

2. maximize *memory utilization*

Trade Off:

Requirements

→ List the requirements of a heap allocator.

1.

2.

3.

4.

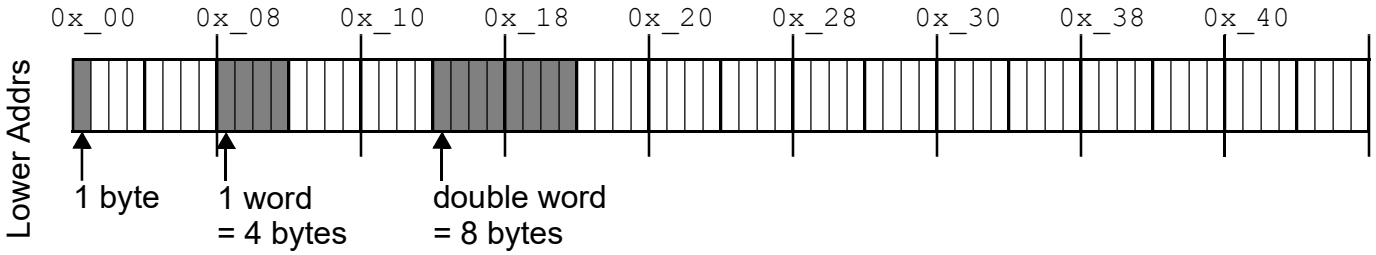
5.

Design Considerations

-
-
-
-

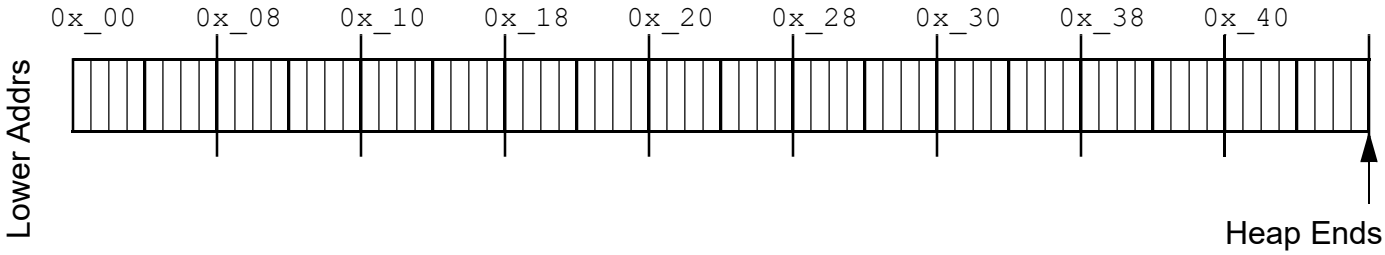
Simple View of Heap

Rotated Linear Memory Layout



double word alignment:

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));
```

→ What happens with the following heap operations:

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

External Fragmentation:

Internal Fragmentation:

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

Free Block Organization

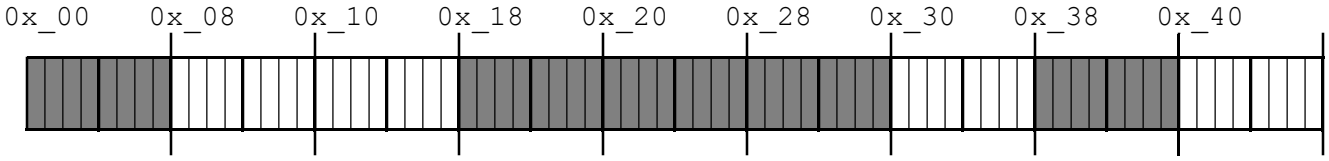
❋ *The simple view of the allocator has*

size

status

Explicit Free List

◆



code:

space:

time:

Implicit Free List

◆

code:

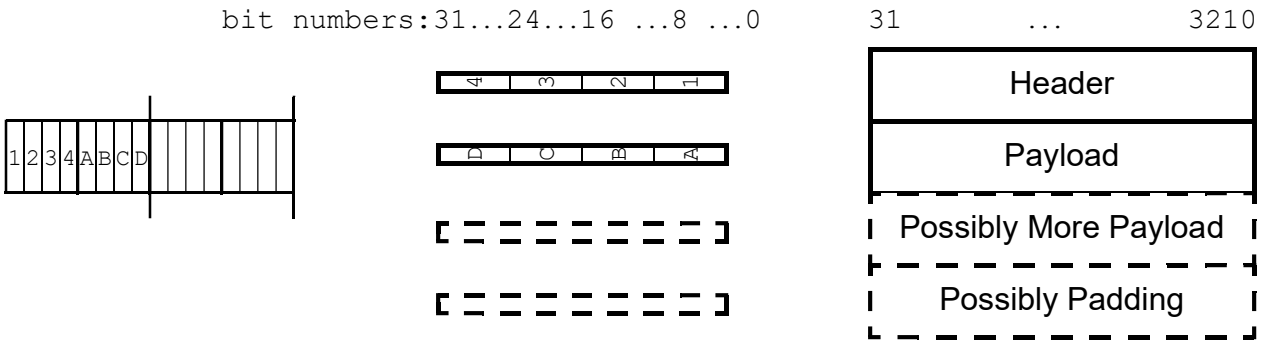
space:

time:

Implicit Free List

❋ *The first word of each block*

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?

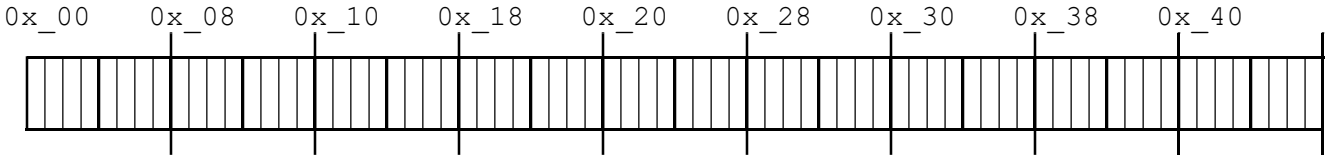
→ What integer value will the header have for a block that is:

allocated and 8 bytes in size?

free and 32 bytes in size?

allocated and 64 bytes in size?

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));
```

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

Placement Policies

What? *Placement Policies* are

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

- **First Fit** (FF): start from
  stop at
  fail if

  mem util:

  thruput:

- **Next Fit** (NF): start from
  stop at
  fail if

  mem util:

  thruput:

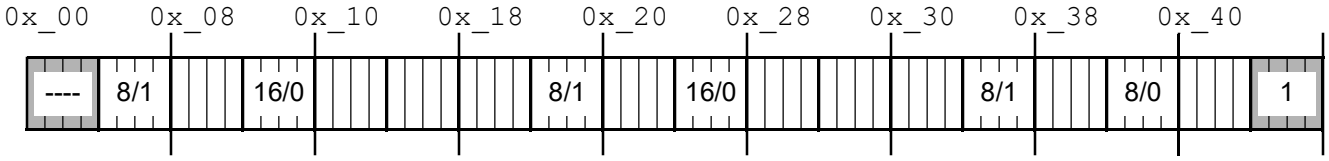
- **Best Fit** (BF): start from
  stop at
  or stop early
  fail if

→

mem util:

thruput:

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));            //FF?            BF?

ptr = malloc(10 * sizeof(char));      //FF?            BF?
```

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

```
ptr = malloc(sizeof(char));         //0x_04?           0x_34?

ptr = malloc(3 * sizeof(int));      //0x_1C?           0x_34?
```