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)

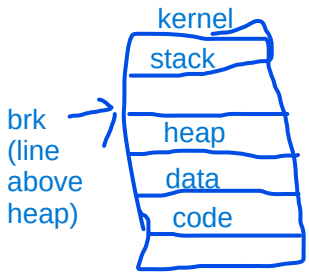
Last Week: Standard & String I/O in `stdio.h` File I/O in `stdio.h` Copying Text Files Meet Globals and Static Locals	C's Abstract Memory Model Where Do I Live? Three Faces of Memory Virtual Address Space Linux: Processes and Address Spaces

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 functions

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 break

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

`void *sbrk(intptr_t incr)` set break

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 some O.S. function to communicate system error

set by OS functions to communicate a specific error

```
#include <error.h>

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

- *For most applications, it's best to use malloc/calloc/realloc/free*
  because the C standard library is 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 C functions

- conversion
- execution flow: alert, exit
- math
- searching
- sorting
- random number

C's Heap Allocator Functions

type of int that can't be negative so it is a valid size (in bytes)

***pointer to void, not void

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

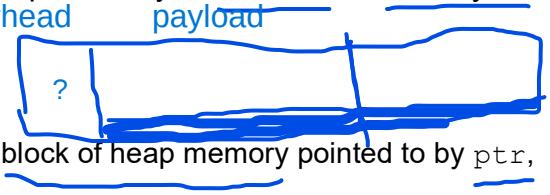
void *

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.

overhead payload




```
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.

only works (increasing size) if there is more empty room directly after memory block

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

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

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

void cannot communicate error
undefined error if things don't work

❄ *For CS 354, if malloc/calloc/realloc returns NULL*
just exit the program with an appropriate error message.
exit(1)

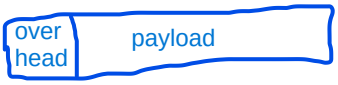
Meet the Heap

What? The heap is

- a segment of process's virtual address space used for dynamically allocated memory

  dynamically allocated memory: memory that is requested while program is running

- a collection of various sized memory blocks
  managed by allocator (code)

  block: contiguous chunk of memory 

  payload: part of block that is useable by process

  overhead: part of block used by allocator to manage heap

  allocator: code that allocates and frees blocks

Two Allocator Approaches

1. Implicit: Java
   - "new" operator - implicitly determines size of block
   - Garbage collector - implicitly determines unused blocks and frees them

2. Explicit: C's approach
   - malloc must be explicitly told # of bytes
   - free must be explicitly called to free up unused bytes

Allocator Design

Two Goals

1. maximize ***throughput*** : number of mallocs and frees
   more throughput is good
   free = O(1) constant
   malloc = O(N) where N is # of heap blocks

2. maximize ***memory utilization*** : percent of memory use for payload
   memory requested / mem allocated
   higher is better

Trade Off: typically increasing one decreases the other

Requirements

→ List the requirements of a heap allocator.

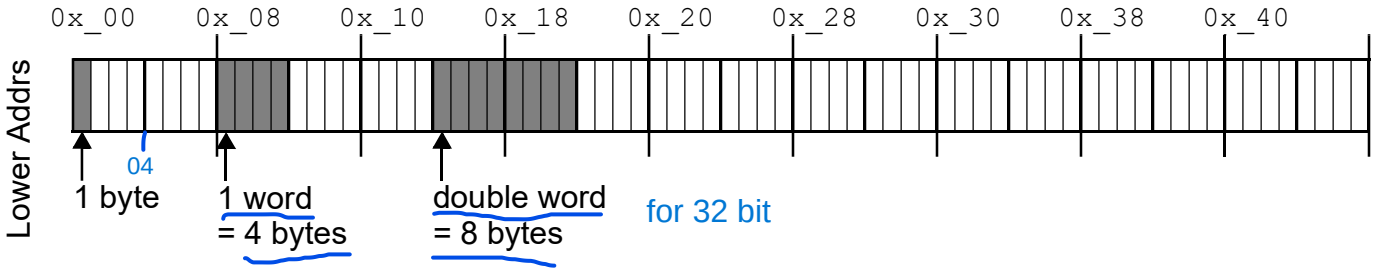
1. allocs use heap
2. provide "immediate" response
3. must handle arbitrary sequence of requests
4. must not move or change previous requests
5. must follow memory alignment requirements

Design Considerations

- "free blocks" organization
- placement policy
- splitting free blocks, for better utilization
- coalescing free blocks to create larger free block, better utilization

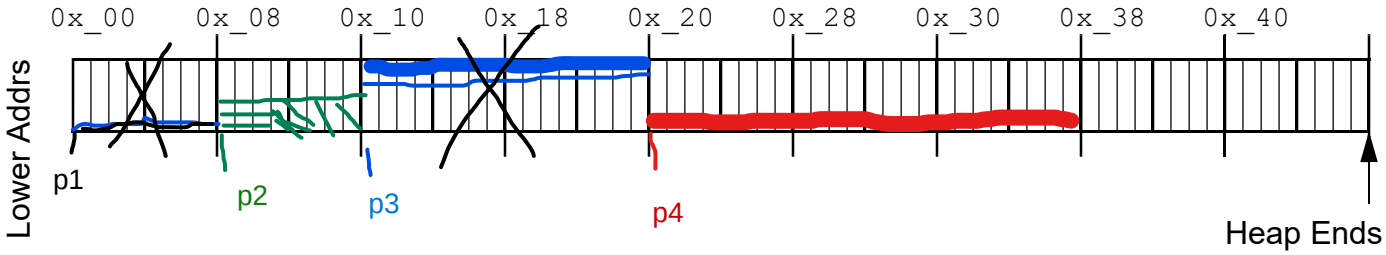
Simple View of Heap - NOT USED

Rotated Linear Memory Layout



double word alignment: 1.) block sizes must be multiples of 8 bytes
2.) payload address must be multiple of 8

Run 1: Simple View of Heap Allocation



→ Update the diagram to show the following heap allocations:

payload + padding = total

```
1) p1 = malloc(2 * sizeof(int));          8 + 0 = 8
2) p2 = malloc(3 * sizeof(char));         3 + 5 = 8
3) p3 = malloc(4 * sizeof(int));          16 + 0 = 16
4) p4 = malloc(5 * sizeof(int));          20 + 4 = 24
```

→ What happens with the following heap operations:

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

24 + 0 = 24 Alloc Fails
return false

External Fragmentation: when there is enough heap memory to satisfy the request, but is in blocks that are too small

Internal Fragmentation: when available heap memory is used for overhead

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

Free Block Organization

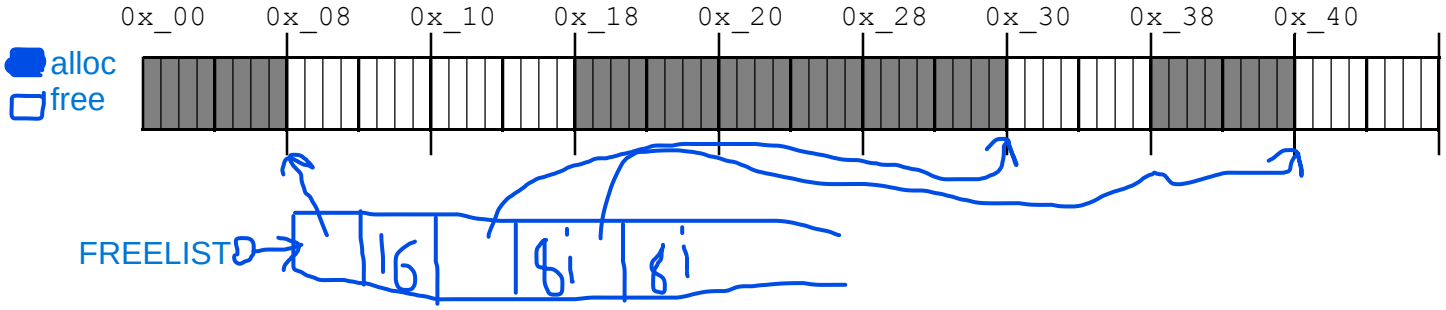
- *The simple view of the allocator has* no way of determining the size and status of each block

  size number of bytes in heap block, payload, padding, overhead for tracking

  status whether block is free or allocated

Explicit Free List

- allocator uses a data structure to keep track of the free blocks



code: only need size and start address of each block

− space: potentially requires more memory to store free list

\+ time: a bit faster to search only free blocks

Implicit Free List

- Allocator uses heap blocks as data structure

code: must track size + status of each block

\+ space: potentially less memory required

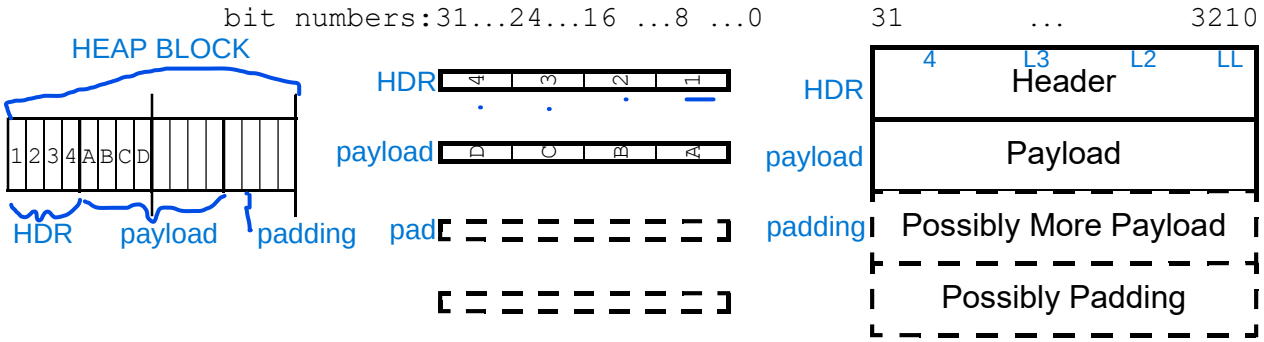
− time: takes more time to check each block and skip allocated ones

Implicit Free List

(4 bytes)

- *The first word of each block* as a HEADER

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



- *The header stores* both the size and status as a single integer (32 bits)

→ Since the block size is a multiple of 8, what value will the last three header bits always have? 0

8: 1000
16: 10000
24: 11000
32: 100000

1001 = 8 + 1 8 bytes in block 1-alloc'd

100000 = 32 + 0 32 0-free

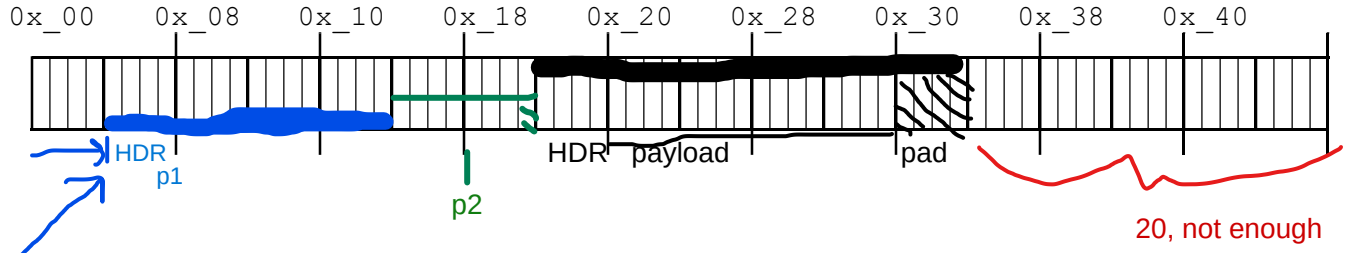
→ What integer value will the header have for a block that is:

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

free and 32 bytes in size? 32 32 + 0 = 32

allocated and 64 bytes in size? 65 64 + 1 = 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));
```

8 (payload) + 4 (header) + 4 (padding) = 16 (heap block size)

3 + 4 + 1 = 8

16 + 4 + 4 = 24

20 ++ 4 + 0 = 24 (ALLOC FAILS)

start of heap

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

ptr + curr_block_size

Placement Policies

What? *Placement Policies* are algorithms used to search heap for free blocks

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 is big enough
  fail if end mark is reached

  \+ mem util: likely to choose block close to desired size

  \- thruput: requires lots of skips to get to large blocks

- **Next Fit** (NF): start from most recently allocated block
  stop at first free block that is big enough
  fail if we reach the start block, ... after wrap-around

  \- mem util: must choose next block that fits, may be too large

  \+ thruput: faster than first fit, no need to skip through many small blocks for allocation

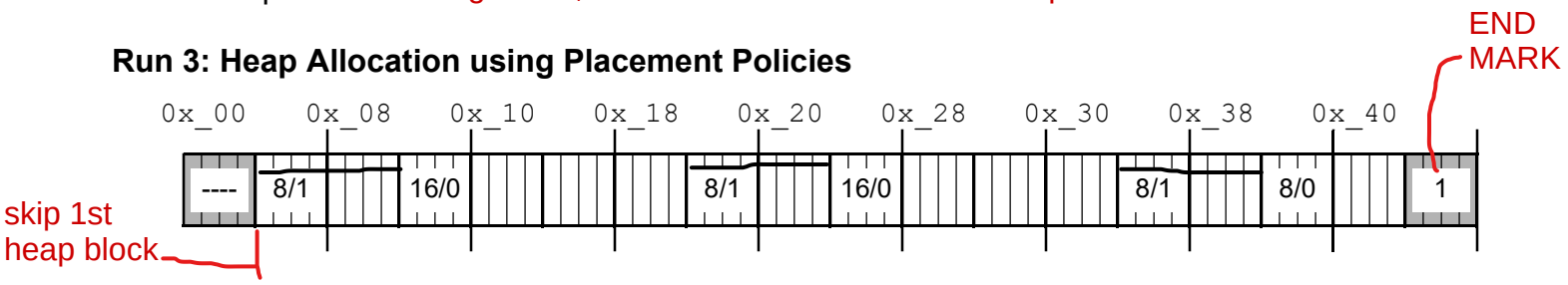
- **Best Fit** (BF): start from beginning of heap
  stop at END MARK, choose free block closest to desired size and big enough
  or stop early if exact fit free block is found
  fail if no blocks are big enough

→

  \+ mem util: closest to best size

  \- thruput: slowest in general, because we must search all heap blocks

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));          block + HDR = 4 + 4 = 8   //FF? 0x_10   BF? 0x_40
ptr = malloc(10 * sizeof(char));    10 + 4 = + 2 = 16         //FF? 0x_10   BF? 0x_10
```

→ 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? 0x_10   0x_34? 0x_40
ptr = malloc(3 * sizeof(int));         //0x_1C? 0x_28   0x_34? 0x_10
```