# outlineL10-w5TR-student

Tuesday, October 4, 2022 1:08 PM

outlineL10-w5TR-student

## CS 354 - Machine Organization & Programming
## Tuesday Oct 4th, and Thursday Oct 6th, 2022

**Midterm Exam - Thurs, Oct 6th, 7:30 - 9:30 pm**

| If your Lecture number is | and the first letter of your family name is , | then, your assigned exam room is: |
|---|---|---|
| 001 | A-K | B130 Van Vleck |
| 001 | L-Z | B102 Van Vleck |
| 002 | A-K | S413 Chemistry |
| 002 | L-Z | S429 Chemistry |

- **UW ID required. Students without UW ID must wait until other students are checked in**
- **#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 7th

**Homework hw2:** Due on Monday Oct 3rd (solution available Wed morning)

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

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

#include <error.h>

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

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

bc C std allocator is well imp, safe + portable

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

use one or other → not both!

## C's Heap Allocator & `stdlib.h`

**What?** **`stdlib.h`** contains a collection of ~25 commonly used C functions (strtol better)

- conversion: atoi (convert string → int)
- execution flow: exit/abort
- math: abs
- searching: bsearch
- sorting: qsort
- rand num: rand srand (seeded rand)
- AND ↓

### C's Heap Allocator Functions

unsigned int → don't put negatives in

```
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)
```

← use on P2B

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

safe, but expensive

```
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 // realloc

fails if large contig block isn't available

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

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

- no error checking

✵ *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 the process VAS, used for dynamically alloc memory
  virtual address space

*dynamically allocated memory*: memory requested while program is running to satisfy newly known memory needs

- A collection of various-sized memory blocks that are managed by allocator (p3)

*block*: contiguous chunk of memory that contains

Blocks have 2 parts

*payload*: part of memory block that is usable by process.

*overhead*: part of block that is used by allocator to manage heap.

*allocator*: code that allocates and frees mem block

**Two Allocator Approaches**

1. Implicit: Java / Py
- "new" operator implicitly determines size needed
- gc: garbage collector determines unused bytes and frees them

2. Explicit: C
- malloc must be explicitly told size (# byte) needed
- free must be explicitly called to free malloc'd data

## Allocator Design

### Two Goals

1. maximize *throughput* # malloc and free requests handled (ops/sec)

Higher is better

free → O(1)

malloc → O(n) where N = no. of heap blocks

2. maximize *memory utilization* % of memory used for payload

= mem requested / heap allocated (payload + overhead)

Trade Off: Incrementing one => decrease in other

### Requirements

→ List the requirements of a heap allocator.

1. alloc. requests use heap space.
2. provide immediate response
3. must handle arbitrary seq. of requests
4. must not move/change prev. allocated blocks
5. must follow memory alignment requirements ← Improve performance!

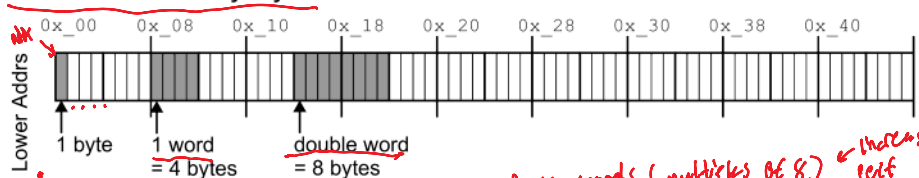
### Design Considerations

- free blocks organization
- placement policy
- splitting free blocks
- coalescing free blocks

# Simple View of Heap

0/2 address

## Rotated Linear Memory Layout

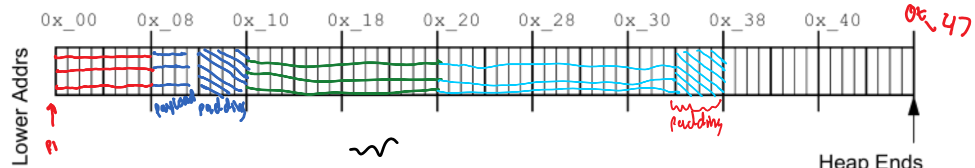



increases perf

*double word alignment*: 1) block sizes must be double-words (multiples of 8)

2) payload addr must be mult of 8 also

## Run 1: Simple View of Heap Allocation




→ Update the diagram to show the following heap allocations:

pay ADDR

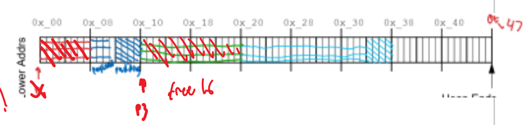
```
1) p1 = malloc(2 * sizeof(int));
2) p2 = malloc(3 * sizeof(char));
3) p3 = malloc(4 * sizeof(int));
4) p4 = malloc(5 * sizeof(int));
```

1) 8+0 = 8 block size — 0x_00
2) 3 + 5 (overhead) = 8 — 0x_08 — pad 0x_0B-0F
3) 16+0 = 16 — 0x_10
4) 20+4=24 — 0x_20 — 0x_34-37

→ What happens with the following heap operations:

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

Alloc FAILS!




*External Fragmentation*: when there's enough heap memory, but it's divided into too small blocks to satisfy the requests.

*Internal Fragmentation*: when heap memory in a block is used for overhead and not payload

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

memory safety errors?

## Free Block Organization

✵ *The simple view of the allocator has* No WAY TO DETERMINE size/status of each block

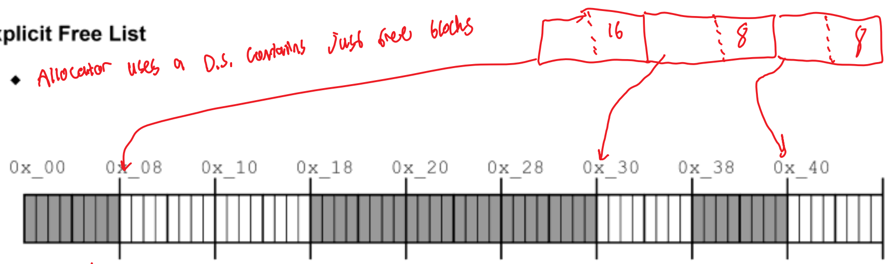
<u>*size*</u> # of bytes in each block (payload + overhead)

<u>*status*</u> whether allocated or free (1-bit)

e.g. free block list (a simple struct)

### Explicit Free List

- Allocator uses a D.S. contains just free blocks




freeblock list

code: Only need to track size of each block

− space: potential to request more memory to store free list

\+ time: usually faster! (we only search freeblocks vs. allocated blocks)

### Implicit Free List

- Allocator uses heap blocks for DS.

code: Must track size and status of each block

\+ space: no extra space, use free space in heap

− time: more time to skip over alloc'd blocks

## Implicit Free List

- The first word (n bytes) of each block is a header

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


- The header stores size + 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 8 0000 — 1000 24 000 — 1000
16 0000 — 1000 use bit 0 as a-bit
32
48
alloc Bit a-bit

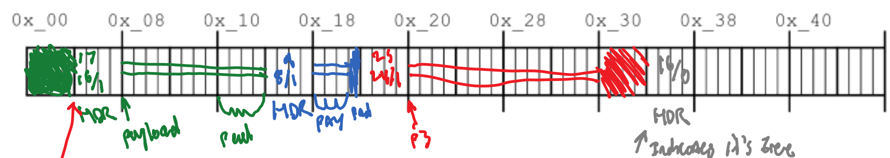
→ What integer value will the header have for a block that is:

allocated and 8 bytes in size? 8+1=9 heap block size 8 & allocated

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

allocated and 64 bytes in size? 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));
```

1) 8 + header (4) = 12 + 4 (padding to get to 16) + 1 (alloc)

2) 4 + 3+1 = 8+1

3) (header) 4·4 = 16+8=24+1

4) header + 5·4 = 24 (no padding needed) + 1 = 25 ALLOC FAILS!

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

ptr + current block size will get you to next header

## Placement Policies

**What?** *Placement Policies* are algorithms used to search heap blocks 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
  stop at first free block that is good enough
  fail if reach "END MARK"

\+ mem util: likely to choose blocks close to desire size

− thruput: requires many steps to find a free block

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

— mem util: Not as good, may choose block that is too big

\+ thruput: Fast! Faster than FF, O(1)

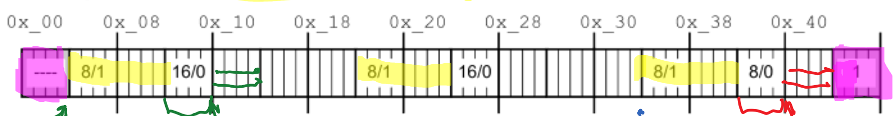
- **Best Fit (BF)**: start from Beginning
  stop at END MARK
  or stop early If block is exact size
  fail if no block big enough is found

→

\+ mem util: Best / closest to best

— thruput: AWFUL

**Run 3: Heap Allocation using Placement Policies**




Alloc BIT ptr HEAD PTR HEAD PTR Address After Header

→ 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?
```

`ptr = malloc(sizeof(int));` 4 + 4 = 8 //FF? 0x_10 BF? 0x_40

Head INT

`ptr = malloc(10 * sizeof(char));` 4 + 10 = 14 //FF? 0x_10 BF? 0x_10

Head f2 (pad) = 16

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

Rallocd

`ptr = malloc(sizeof(char));` 4+1+3=8 //0x_04? 0x_10 0x_34? 0x_40

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

4+12=16