15-213 Recitation
Malloc Lab (Checkpoint)

Your TAs

Friday, October 11th

Reminders

- `cachelab` was due ***yesterday***.
- `malloclab` was released yesterday:
  - Checkpoint: ***October 29th***
  - Final: ***November 5th***
- *Bootcamp 5: Post-Checkpoint Malloc* will be in-person, and is happening on ***October 27th***.

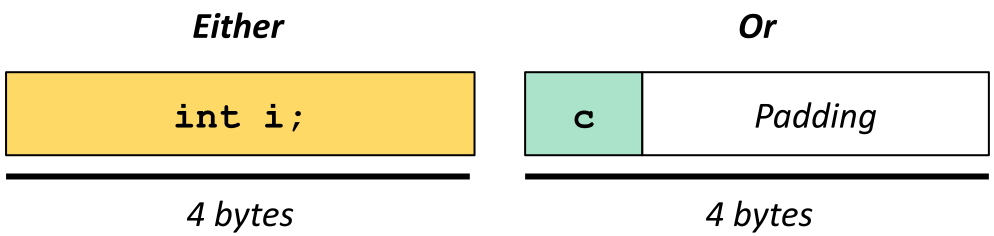
Agenda

- Review: Programming in C
- `malloc` concepts
- Optimizations
  - Explicit Lists
  - Seglists
- Strategy Guide
  - Debugging
  - Suggested Roadmap

Review: Programming in C

Programming in C: Unions

```
union temp {
   int i;
   char c;
};
```



- Store potentially different data types in the same region of memory.
- Specifies multiple ways to interpret data at the same memory location.

Programming in C: Zero-Length Arrays

```
typedef uint64_t word_t;
```

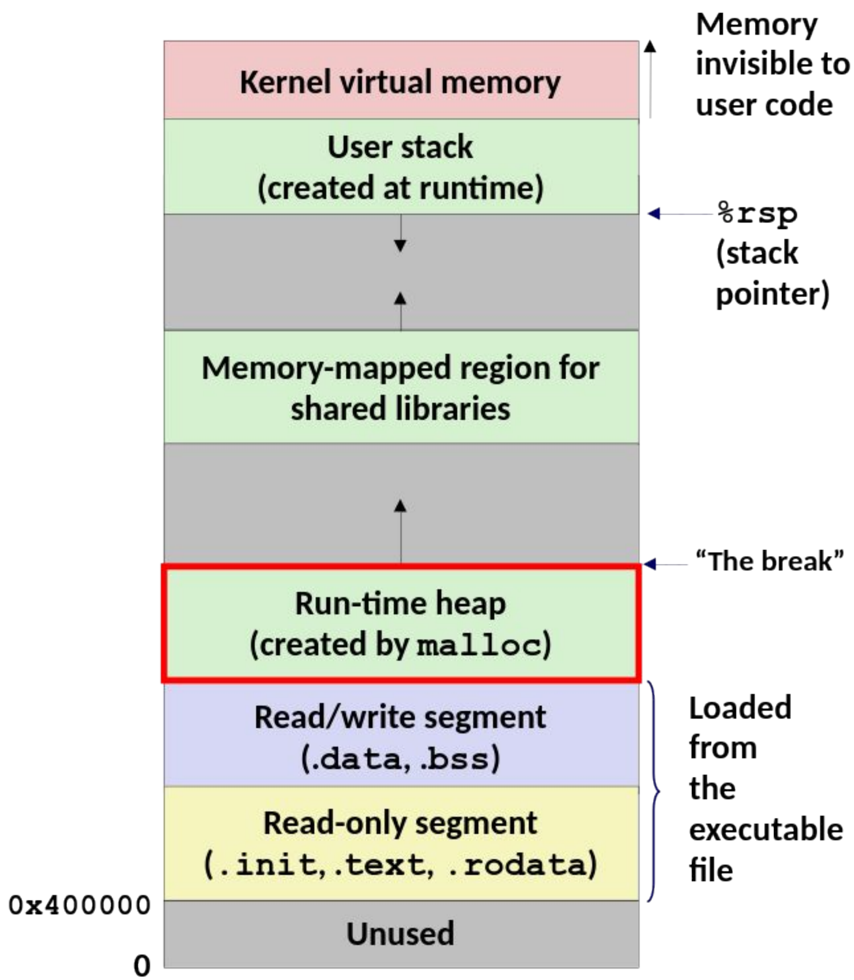
```
typedef struct block
{
    word_t header;
    unsigned char payload[0];          // Zero length array
} block_t;
```

- Allowed in GNU C as an extension.
- A zero-length array must be the last element in a struct.
- **`sizeof(payload)`** always returns 0
- But, the payload itself can have variable length

`malloc` Concepts

What does `malloc` do?

- Given a bunch of heap space, manage it effectively:
  1. Use heap space to organize blocks and information we store about blocks in a *structured way*.
  2. Using that structure, *decide where to allocate new blocks*.
  3. *Update structure correctly* when we allocate or free, *maintaining heap invariants*.
- …and do so in a way that maximizes throughput and utilization!



`malloc` Starter Code

```
static block_t *coalesce_block(block_t *block) {
    // TODO: delete or replace this comment once you're done.
    return block;
}
```

- Starter code: working implementation of implicit free list with boundary tags.
- However, it does not implement coalescing!
- Now it’s our turn! Let’s talk about what we need to do.

malloc Starter Code

```
[dalud@angelshark:~/.../15213/s17/malloclabcheckpoint-handout] $ ./mdriver -p
Found benchmark throughput 13090 for cpu type Intel(R)Xeon(R)CPUE5520@2.27GHz, benchmark checkpoint

Throughput targets: min=2618, max=11781, benchmark=13090
......................
Results for mm malloc:
  valid    util     ops    msecs    Kops  trace
   yes    78.4%      20    0.002    9632 ./traces/syn-array-short.rep
   yes    13.4%      20    0.001   25777 ./traces/syn-struct-short.rep
   yes    15.2%      20    0.001   24783 ./traces/syn-string-short.rep
   yes    73.1%      20    0.001   19277 ./traces/syn-mix-short.rep
   yes    16.0%      36    0.001   31192 ./traces/ngram-fox1.rep
   yes    73.6%     757    0.145    5237 ./traces/syn-mix-realloc.rep
 * yes    62.0%    5748    3.925    1464 ./traces/bdd-aa4.rep
 * yes    58.3%   87830 1682.766      52 ./traces/bdd-aa32.rep
 * yes    58.0%   41080  410.385     100 ./traces/bdd-ma4.rep
 * yes    58.1%  115380 4636.711      25 ./traces/bdd-nq7.rep
 * yes    56.6%   20547   26.677     770 ./traces/cbit-abs.rep
 * yes    55.8%   95276  675.303     141 ./traces/cbit-parity.rep
 * yes    58.0%   89623  611.511     147 ./traces/cbit-satadd.rep
 * yes    49.6%   50583  185.382     273 ./traces/cbit-xyz.rep
 * yes    40.6%   32540   76.919     423 ./traces/ngram-gulliver1.rep
 * yes    42.4%  127912 1284.959     100 ./traces/ngram-gulliver2.rep
 * yes    39.4%   67012  338.591     198 ./traces/ngram-moby1.rep
 * yes    38.6%   94828  701.305     135 ./traces/ngram-shake1.rep
 * yes    90.9%   80000 1455.891      55 ./traces/syn-array.rep
 * yes    88.0%   80000  915.167      87 ./traces/syn-mix.rep
 * yes    74.3%   80000  914.366      87 ./traces/syn-string.rep
 * yes    75.2%   80000  812.748      98 ./traces/syn-struct.rep
16 16     59.1% 1148359 14732.604     78

Average utilization = 59.1%. Average throughput = 78 Kops/sec
Checkpoint Perf index = 20.0 (util) + 0.0 (thru) = 20.0/100
```

Very **slow!**

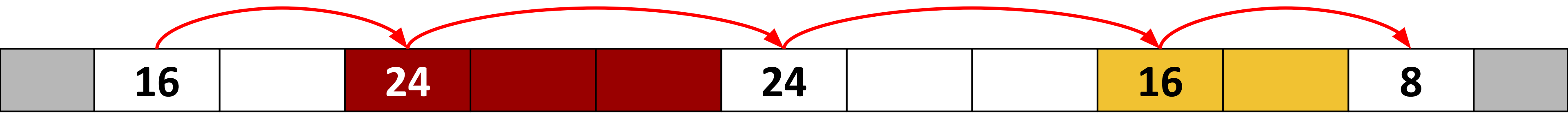
Getting Started on Checkpoint

- Based on the starter code, we've found two things we can improve on already!
- Implement Coalescing
  - See *"Malloc Basic"* lecture.
- Throw out implicit list for something *faster*.
  - Start with *explicit list*.
  - Work up to *segregated lists*!
- We'll talk about both today!

Explicit Lists

Explicit Lists

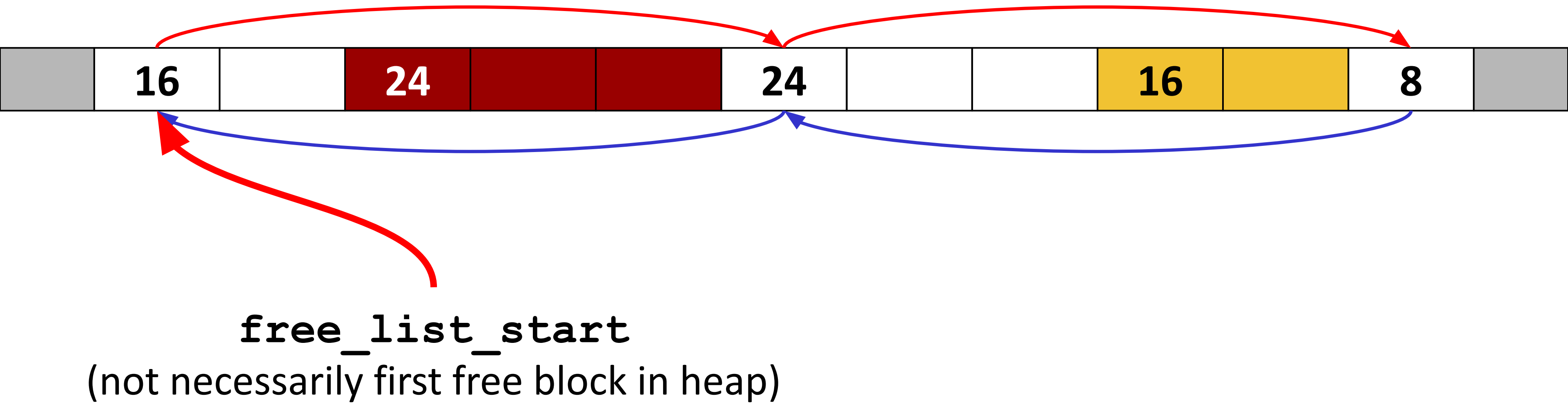
Implicit List



- Implicit List achieves poor throughput. Why?
- How do we find a free block for an allocation?
  - Which blocks are searched?
  - How could we do better?

Explicit Lists

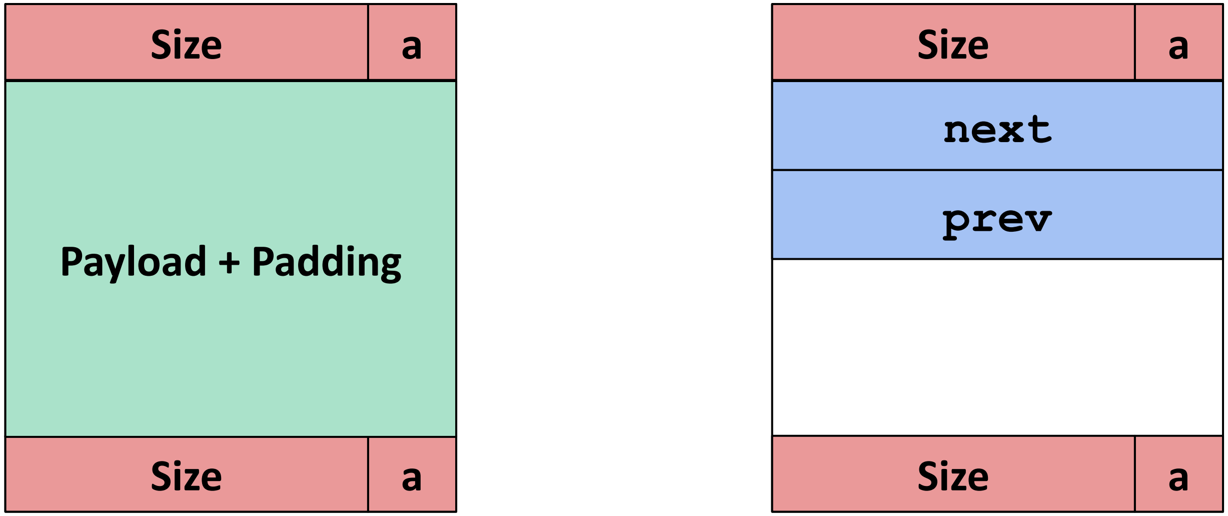
- We want to search only *free* blocks:



- Note: these forward/backward pointers require *space*
  - Where do we store them?

Explicit Lists: Implementation

- Forward/backward pointers require *space*
- Free blocks are *free*!
  - Not in use by any application.
  - So our allocator can use their space to store its own data structures.



Allocated (as before)

Free

Explicit Lists: Performance

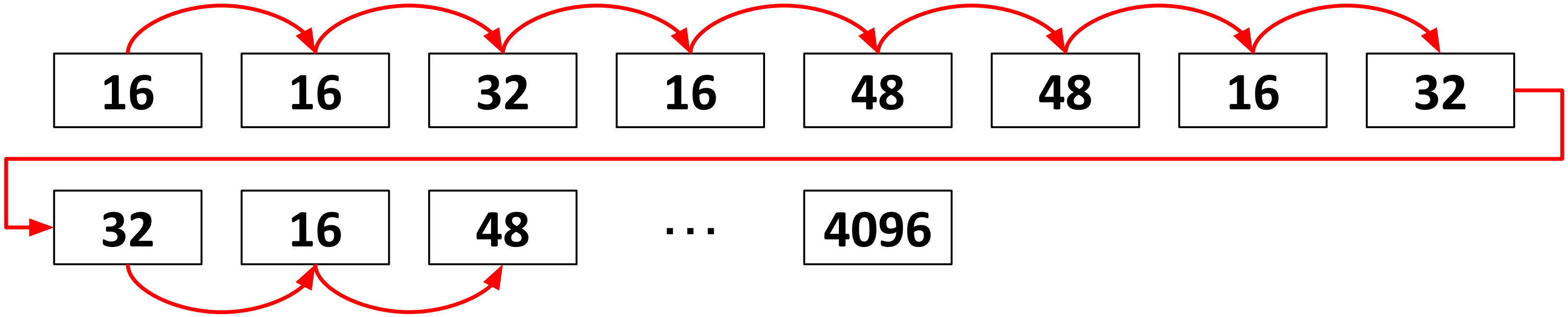
Optimization	Utilization	Throughput
Implicit List (Starter Code)	59%	10–100
Explicit Free List[a]	mid-50s	1000–2500
Segregated Free Lists	–	6000
Better Fit Algorithm	59%	Variable
Eliminating Footers in Allocated Blocks	+9%	–
Decreasing Block Size/Mini Blocks	+6%	–20%
Compressing Headers	+2%	–

[a] – utilization score assumes the allocation order is the same as implicit list - otherwise expect a minimum of 53% utilization.

- Way faster!
- But we can still do better… On to segregated lists!
  - Note: you'll need to understand explicit lists first.

Segregated Lists

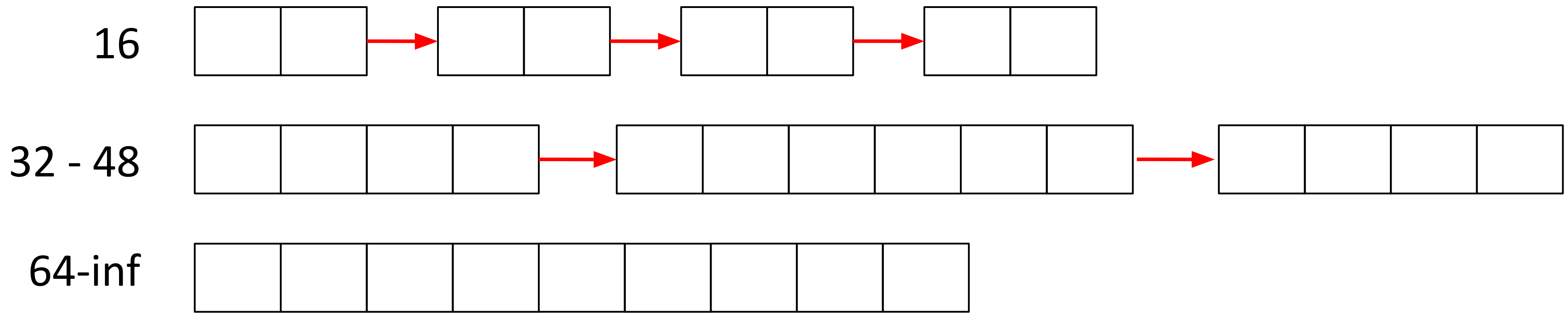
Segregated Lists



- With explicit lists, we only have to search free blocks.
- But, for a given request, we still have to search *all* free blocks.
  - What happens when **`malloc(4096)`** tries to find a free block?
- Can we do better?

Segregated Lists

- Segregated Lists: have *multiple* free lists, one for each size class.



- Size classes are up to you!
  - Remember: you may optimize for the traces as long as you don’t hardcode!

Segregated Lists: Performance

Optimization	Utilization	Throughput
Implicit List (Starter Code)	59%	10–100
Explicit Free List[a]	mid-50s	1000–2500
Segregated Free Lists	–	6000

- We have motivated seglists as a *throughput* optimization.
- What might they do for *utilization?*
  - If you're using "first fit"?
  - If you're using "best fit"?

Design Choices

Design Choices

- Though we'll recommend a strategy later, there are many ways to optimize your allocator.
- What kind of implementation to use?
  - Implicit list, explicit, segregated, binary tree, etc.
- What fit algorithm to use?
  - Best Fit?
  - First Fit? Next Fit?
  - Which is faster? Which gets better utilization?
- There are many different ways to get a full score!

Strategy Guide: Debugging

In a perfect world…

- Setting up blocks, metadata, lists, etc. (500 LoC)
- Finding and allocating the right blocks (500 LoC)
- Updating heap structure on frees (500 LoC)

=

```
[dalud@angelshark:~/.../15213/s17/malloclabcheckpoint-handout] $ ./mdriver
Found benchmark throughput 13056 for cpu type Intel(R)Xeon(R)CPUE5520@2.27

Throughput targets: min=6528, max=11750, benchmark=13056
......................
Results for mm malloc:
  valid    util      ops    msecs     Kops  trace
   yes    78.1%       20    0.004     5595 ./traces/syn-array-short.rep
   yes     3.2%       20    0.004     5273 ./traces/syn-struct-short.rep
 * yes    96.0%    80000   17.176     4658 ./traces/syn-array.rep
 * yes    93.2%    80000    6.154    12999 ./traces/syn-mix.rep
 * yes    86.4%    80000    3.717    21521 ./traces/syn-string.rep
 * yes    85.6%    80000    3.649    21924 ./traces/syn-struct.rep
16 16     74.2%  1148359   55.949    20525

Average utilization = 74.2%. Average throughput = 20525 Kops/sec
Perf index = 60.0 (util) + 40.0 (thru) = 100.0/100
```

In reality…

- Setting up blocks, metadata, lists, etc. (500 LoC)
- Finding and allocating the right blocks (500 LoC)
- Updating heap structure on frees (500 LoC)
- **+ Some bug hiding in those 1500 LoC…**

=

```
[dalud@angelshark:~/.../15213/s17/malloclabcheckpoint-handout] $ ./mdriver
Found benchmark throughput 13056 for cpu type Intel(R)Xeon(R)CPUE5520@2.27

Throughput targets: min=6528, max=11750, benchmark=13056
.....Segmentation fault
[dalud@angelshark:~/.../15213/s17/malloclabcheckpoint-handout] $
```

Debugging Strategies

- Use **`gdb`**!
- Write a heap checker!
  - Checks heap invariants
  - Call around major operations to make sure heap invariants aren't violated.
- Assertions (like 122!):
  - **`dbg_assert(...)`**

Common Errors

- ***Garbled Bytes***
  - This means you're overwriting data in an allocated block.
- ***Overlapping Payloads***
  - This means you have unique blocks whose payloads overlap in memory
- **`segfault`**!
  - This means something is accessing invalid memory.
- For all of the above, step through with **`gdb`** to see where things start to break!
  - Note: to run assert statements, you'll need to run **`./mdriver-dbg`** rather than **`./mdriver`**.

Using gdb: Breakpoints and Watchpoints

- *Breakpoints:*
  - `break coalesce_block`
  - `break mm.c:213`
  - `break find_fit if size == 24`
- *Watchpoints:*
  - `w block = 0x8000010`
  - `w *0x15213`
  - `rwatch <thing>` – stop on *reading* a memory location
  - `awatch <thing>` – stop on *any* access to the location

Using gdb: Inspecting Frames

```
(gdb) backtrace  #0  find_fit (...)
#1  mm_malloc (...)
#2  0x0000000000403352 in eval_mm_valid (...)  #3  run_tests (...)
#4  0x0000000000403c39 in main (...)
```

- **`backtrace`** - print call stack up until current function
- **`frame 1`**: switch to mm_malloc's stack frame
  - Can then inspect local variables.

Writing a Heap Checker

- Heap checker: just a function that loops over your heap/data structures and makes sure *invariants* are satisfied.
  - Returns **`true`** *if and only if* heap is well-formed.
- Critical for debugging!
  - Update when your implementation changes.
- Worry about *correctness*, not efficiency.
  - But *do* avoid printing excessively.
- For Checkpoint, *you will be graded on the quality of your heap checker*.

Heap Invariants

- *Heap invariants* are things that should always be true about the heap/your data structures between calls to **`malloc/free`**.
- Can you come up with some invariants?
  - Block Level: what should be true about individual blocks?
  - List Level: what should be true about your free list(s)?
  - Heap Level: what should be true about your blocks in relation to the heap?

Heap Invariants: Block Level

Disclaimer: Non-Exhaustive

- Header and footer store size/allocation information. Do they match?
- Payload area is 16-byte aligned.
- Size is valid.
- No contiguous free blocks (unless you do deferred coalescing).

Heap Invariants: List Level

Disclaimer: Non-Exhaustive

- Assuming a doubly-linked explicit list:
  - `prev/next` pointers are consistent
  - No allocated blocks in free list
  - No cycles!
- Segregated lists:
  - Common bug: forgetting to move blocks between buckets when their sizes change.
  - Invariant: each segregated list contains only blocks in the appropriate size class.

Heap Invariants: Heap Level

Disclaimer: Non-Exhaustive

- All blocks are between heap boundaries.
- “Sentinel” Blocks store correct information.
  - “Dummy” footer (at the start of the heap) and “dummy” header (at the end of the heap) prevent accidental coalescing.

Strategy Guide: Suggested Roadmap

Suggested Roadmap

- First: read the write-up!
  - "Roadmap to Success" section

0. Start writing your heap checker!
1. Implement **`coalesce_block()`** *first*.
2. Implement an *explicit free list*.
3. Implement *segregated lists*!
4. Further optimizations (in this order)
   - Footer Removal in allocated blocks
   - Decrease minimum block size
   - Compress Headers (hard)

Checkpoint (items 0–3)

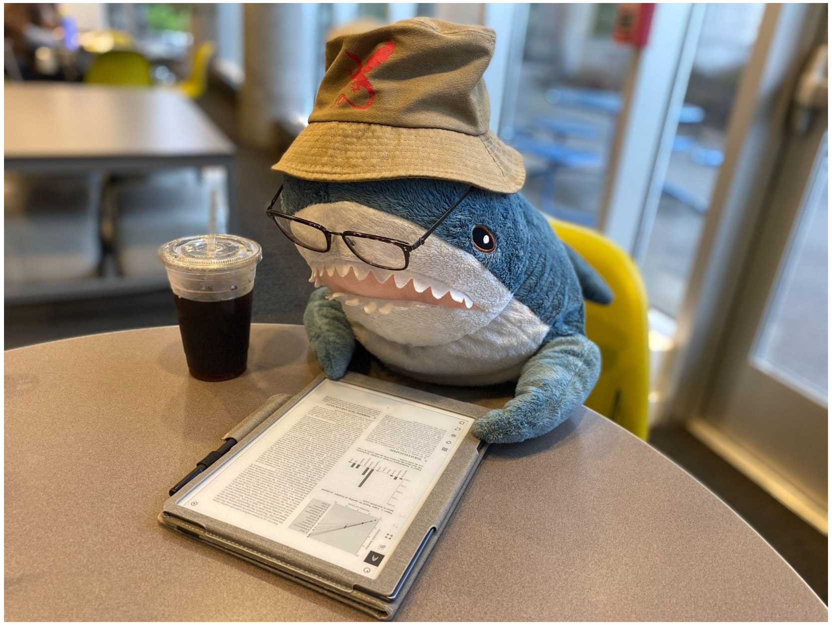
Final (item 4)

Note: Using `git`

- As we have seen:
  - This is a difficult lab.
  - You will experiment with different optimizations, with varying effects on performance and thus, your score.
- Make sure to regularly checkpoint your code with commits, and push it to GitHub!
  - Don’t want to lose your progress.
  - It will be helpful to include performance metrics in your commit messages.

Wrapping Up

- `malloc` due dates:
  - Checkpoint: ***October 29th***
  - Final: ***November 5th***
  - Start early!
- *Bootcamp 5: Post-Checkpoint Malloc*: ***October 27th***.
- `cachelab`: Watch your inbox for an email from your code review TA!
- Have a good Fall Break :-)



The End