Computer Science

Georgia Institute of Technology

CS 3210 Spring 2020

Quiz I

Many problems are open-ended questions so be mindful of time spent for each question. To receive credit, you must answer the question as precisely yet concisely as possible. You have 70 minutes to complete this quiz.

Write your name and GT ID number on the top-left corner of this cover sheet AND at the bottom of each page of this booklet.

Some questions may be harder than others. Read them all through first and attack them in the order that allows you to make the most progress (i.e., tackling problems with bonus points later). **If you find a question ambiguous, be sure to write down any assumptions you make.** Be neat. If we can't understand your answer, we can't give you credit!

Minor syntax errors in your Rust code is acceptable.

THIS IS AN OPEN BOOK, OPEN LAPTOP EXAM
(NO INTERNET ALLOWED)

Please do not write in the boxes below.

I (xx/30)	II (xx/30 + 5)	III (xx/40 + 5)	IV (xx/10)	Total (xx/125)

You only need 100 out of 120 points.

I Rustling [30 points]

Please **circle none, one, or more than one** answers. If the provided answer is wrong, your score will be **deducted** by **2** points on each problem in this section. Answer only when you are confident. If you don't see any correct answer, describe your reasons *concisely*.

1. [5 points]: Type/Size. What are the sizes of variables, `v0`, `v1`, `v2`, `v3`, and `v4` in the *64-bit* architecture (8-byte pointer)?

```rust
let v0 = String::from("Hello World!");
let v1 = vec![
    String::from("Hello"),
    String::from(" "),
    String::from("World!"),
];
let v2 = vec!["Hello", " ", "World!"];
let v3 = ["Hello", " ", "World!"];
let v4 = &v0;
```

For this problem, each correct answer is worth **1** point, but your score will be deducted by **1** point if the answer is wrong. Point will not be given or deducted if the answer is empty.

Answer (size of v0): ________________
Answer (size of v1): ________________
Answer (size of v2): ________________
Answer (size of v3): ________________
Answer (size of v4): ________________

Name:

2. [5 points]: Slice. Which of the following can fill in the blank in this code to print "hello" without compile error?

```
fn main(){
    let s1 = String::from("hello world");
    let s2 = "hello world";

    println!("{}", _______________ );
}
```

A. `s1`

B. `&s1`

C. `s2`

D. `&s2`

E. `&s1[0..5]`

F. `&s2[0..5]`

G. `s1[0..5]`

H. `s2[0..5]`

I. `s1.as_str()[0..5]`

J. `s2.split(' ').next().unwrap()`

3. [5 points]: Overflow. Assume that `x = i64::MIN`. How does this code print out in both a *debug* and a *release* mode?

```rust
let x: i64 = ...;

if (x < 0) {
    if (x < -x) {
        println!("1");
    } else {
        println!("2");
    }
} else {
    println!("3");
}
```

A. (1, panic)

B. (1, 1)

C. (3, 3)

D. (panic, 2)

E. (panic, 1)

F. Undefined behavior

4. [3 points]: Trait. Select all function definitions that return a trait object?

```rust
trait Summary {
    fn summary(&self) -> String;
}

// A.
fn new_summary(option: u32) -> Summary { ... }
// B.
fn new_summary<T: Summary>(option: u32) -> T { ... }
// C.
fn new_summary(option: u32) -> &impl Summary { ... }
// D.
fn new_summary(option: u32) -> &dyn Summary { ... }
// E.
fn new_summary(option: u32) -> Box<dyn Summary> { ... }
```

Answer: ______________________________

Name:

5. [4 points]: Lifetime. Select all `insert_field` function signatures that compile.

```rust
struct TextField<'a>(&'a str);
struct InputForm<'a>(Vec<TextField<'a>>);

fn insert_field ______________________________ {
    in_form.0.push(TextField(txt));
}

fn main(){
    let mut in_form = InputForm(Vec::new());
    insert_field(&mut in_form, "First Name");
    insert_field(&mut in_form, "Second Name");
    ...
}

// A.
fn insert_field(in_form: &mut InputForm, txt: &str) { ... }
// B.
fn insert_field<'a, 'b: 'a>(in_form: &mut InputForm<'a>, txt: &'b str) { ... }
// C.
fn insert_field<'a, 'b>(in_form: &mut InputForm<'a>, txt: &'b str) { ... }
// D.
fn insert_field<'c>(in_form: &mut InputForm<'c>, txt: &'c str) { ... }
```

Answer: ______________________________

Name:

6. [3 points]: Dereference. What is the output of the following code:

```rust
struct Stack();
trait ID { fn who_am_i(&self); }

impl ID for Stack {
    fn who_am_i(&self) { print!("A"); }
}
impl ID for &Stack {
    fn who_am_i(&self){ print!("B"); }
}

fn main(){
    let x = Stack();
    x.who_am_i();
    let y = &x;
    y.who_am_i();
    let z = &y;
    z.who_am_i();
}
```

A. Compilation error.

B. Compiles but panics at runtime.

C. **Printing:** ______________________________

7. [5 points]: Unsafety. Select all possible outputs of the following program:

```rust
use std::mem;

fn always_returns_true(x: u8) -> bool {
    x < 150 || x > 120
}
fn main() {
    let x: u8 = unsafe { mem::uninitialized() };
    println!("{}", always_returns_true(x));
}
```

A. true

B. false

C. none (the program crashes/panics)

D. Random garbage byte

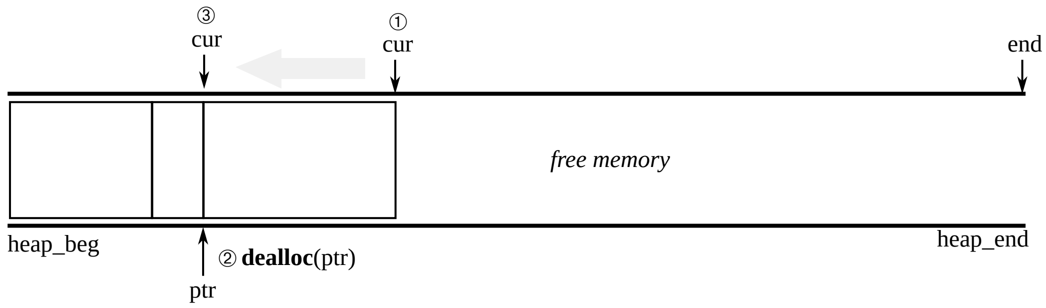
E. Compilation error

Name:

II Heap Allocators [30 points]

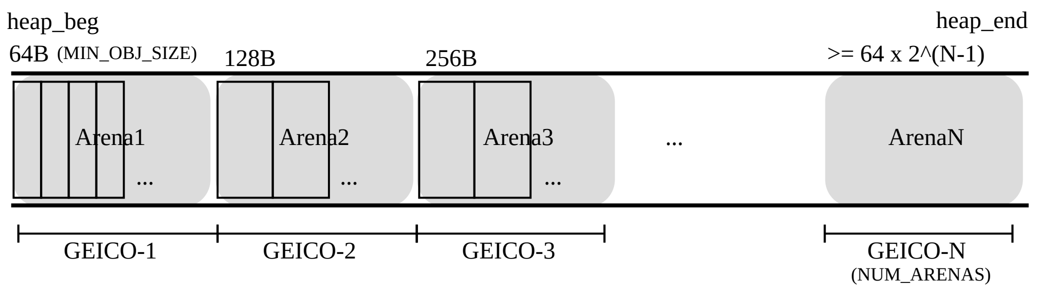
8. [10 points]: GEICO Allocator. The Bump Allocator in Lab3 shamelessly leaks the memory when `dealloc()` is invoked. However, we observed an interesting pattern: when an object is allocated, it's likely to be deallocated immediately after. It means, there is no other `alloc()`s invoked in between. To take advantage of this usage pattern, we'd like to push the about-to-be-freed chunk *back* to the unallocated heap region, restoring the heap from the last bump.

Please implement this algorithm in `dealloc()`.



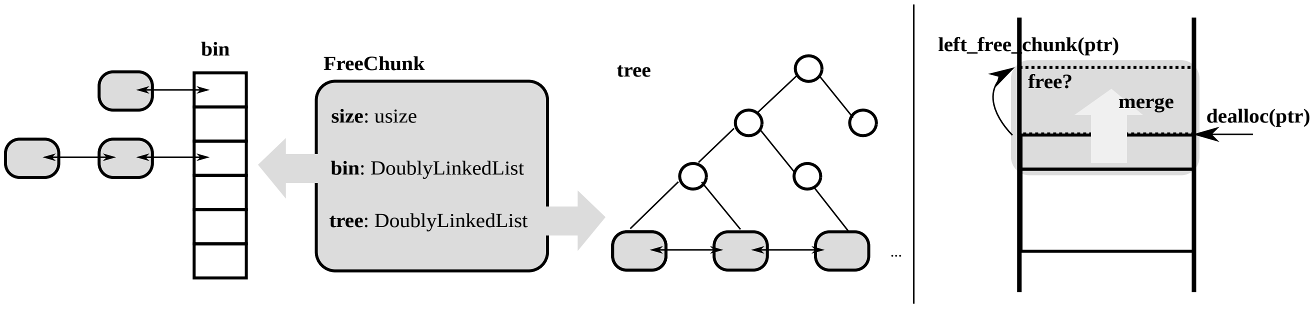
```rust
#[derive(Default)]
pub struct GEICOAllocator {
    cur: usize,
    end: usize,
}

impl GEICOAllocator {
    pub fn new(beg: usize, end: usize) -> GEICOAllocator { ... }
    unsafe fn alloc(&mut self, layout: Layout) -> *mut u8 { .. }
    unsafe fn dealloc(&mut self, ptr: *mut u8, layout: Layout) {














    }
}
```

9. [10 points]: Ensemble Allocator. The GEICO allocator fails to restore free chunks if there exist any new allocation in between. To alleviate this problem, we introduce the *Ensemble Allocator* that divides an available heap space into multiple sub-regions, called *Arena*, and applies the GEICO allocator on *each arena*. Each arena will only be used for the *same size* chunks to increase the chance of merging, with incrementally increasing chunk sizes for each arena: i.e., Arena 0 will have 64-B chunks, Arena 1 will have 128-B chunks, etc. To make it simple, it's okay to just panic when one Arena is out of space. However, **[5 points]** bonus will be given if such a case is correctly handled without panicking!



```rust
// Please use GEICOAllocator in the previous page

const MIN_OBJ_SIZE: usize = 64;
const NUM_ARENAS: usize = 10;

#[derive(Default)]
pub struct EnsembleAllocator {
    arenas: [GEICOAllocator; NUM_ARENAS]
}

impl EnsembleAllocator {
    pub fn new(beg: usize, end: usize) -> EnsembleAllocator {
        let size = (end - beg) / NUM_ARENAS;
        assert!(size > 1*MB);

        let mut alloc: EnsembleAllocator = Default::default();
        let mut cur = beg;
        for n in 0..NUM_ARENAS {
            alloc.arenas[n] = GEICOAllocator::new(cur, cur + size);
            cur += size;
        }
        alloc
    }
    unsafe fn alloc(&mut self, layout: Layout) -> *mut u8 {



















    }

    unsafe fn dealloc(&mut self, ptr: *mut u8, layout: Layout) {

















      }







}
```

10. [10 points]: BinTree Allocator The Bin Allocator in Lab3 suffers from external fragmentation, in which an `alloc()` will fail even if there *is* indeed a free space larger than the requested size. To resolve this issue, next to the bin data structure, we are providing an additional data structure (**tree**) that links two consecutive free chunks in the freed region. Notably, it includes the **size** information that is required to check the physical proximity between the chunks. Meaning that, the tree data structure can be used for a quick, and sorted searching of freed chunks based on their addresses. While the bins are used to link chunks that are within the bins' size range. Use both data structures and their API to write the `dealloc()` function such that it consolidate adjacent free chunks, and inserts them back to both data structures appropriately.



```rust
// Assume that there is a global allocator of 'Allocator' type
pub struct Allocator {
    // These data structures are provided.
    // Use APIs defined in 'BinTreeAllocator' to interact with them.
    bin: Bin,
    tree: Tree,
    ...
}

#[derive(Default)]
pub struct DoublyLinkedList {
    left: *mut usize,
    right: *mut usize,
}

#[derive(Default)]
pub struct FreeChunk {
    size: usize,
    // You don't have to read from or write to these fields; Use the APIs below.
    bin: DoublyLinkedList,
    tree: DoublyLinkedList,
}

impl BinTreeAllocator {
    // Use these APIs to add/del chunks to/from bin and tree data structures.
    fn add_to_bin(&mut self, chunk: *mut FreeChunk) {}
    fn add_to_tree(&mut self, chunk: *mut FreeChunk) {}
    fn del_from_bin(&mut self, chunk: *mut FreeChunk) {}
    fn del_from_tree(&mut self, chunk: *mut FreeChunk) {}

    // Given an address, it returns the physically closest left/right free chunks
    // using the tree data structure. 'ptr::null()' is returned
    // if there is no nearest free chunk
    // (e.g., leftmost or rightmost free chunk is provided)
    fn left_free_chunk(&self, chunk: *const FreeChunk) -> *mut FreeChunk {}
    fn right_free_chunk(&self, chunk: *const FreeChunk) -> *mut FreeChunk {}
}
```

Name:

```
    unsafe fn dealloc(&mut self, ptr: *mut u8, layout: Layout) {
        let mut size = layout.size();

        let mut chunk = ptr as *mut FreeChunk;
        chunk.write(FreeChunk { size: size, ..Default::default() });













































    }
}
```

Name:

III TAF: In-memory Filesystem for Tiny Allocation [40 points]

We will design an in-memory filesystem that aims to provide Tiny Allocation Table (TAT). It means we use a byte addressable DRAM as our storage backend, so of course, no persistence guaranteed. At the same time, our filesystem is only for tiny devices (e.g., IoT devices, sensors) with a limited capability. More specifically, we'd like to explore a design space of TAF for 64 KB (2^{16}) DRAM that *maximizes* the memory utilization.

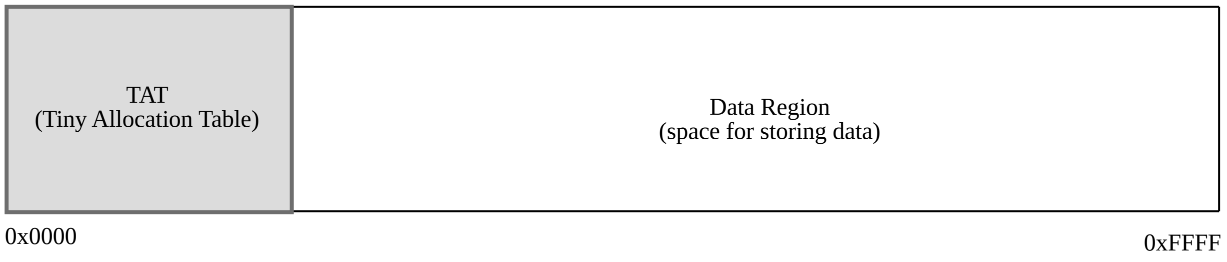


Figure 1: A layout structure for TAF on the 64 KB DRAM.

11. [5 points]: To reduce the wasted space in the data region, we use **8-byte** as a unit to store and load to the data region (as one *cluster size* in FAT). To simplify TAT's design, we use **2-byte** entry to address the data region.

What is the size of TAT? What percentage of 64 KB is occupied by TAT?

Name:

12. [5 points]: It turns out, although it reduces the wastes in the data region, TAF does not provide much of available data storage because of the storage for non-data regions like TAT. This time, we'd like to first dedicate **1 KB (1.6%)** for TAT and calculate a proper size of the unit for load/store. We still assume that TAT uses **2-byte** entries to address the data region.

What should be the proper cluster size?

13. [10 points]: We found that it's inefficient to use a **2-byte** entry to address the 64 KB space. This time, we'd like to use **1-byte** entries instead in TAT.

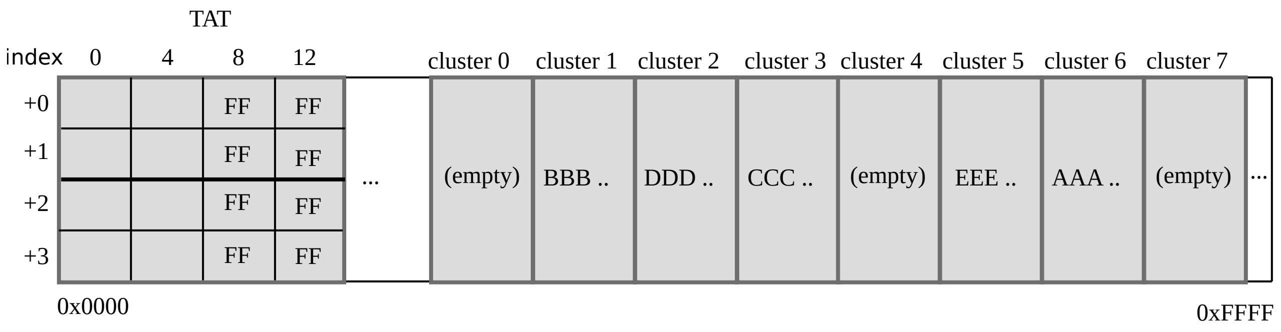
A. [3 points] What's the proper size of one cluster (load/store unit) ?

B. [3 points] What percentage of the storage is available for storing data?

C. [4 points] What's the biggest size file we can store in this TAF?

14. [10 points]: In the middle of storing a long file ("AAAA...BBBB...CCCC...DDDD...EEEE..."), the Rpi is unplugged and found that we couldn't read the file any more. When we dumped the filesystem image, it seems feasible to manually *restore* the TAT structure. In our TAF, a special cluster number, `0xFF` (1-byte) is used to indicate the end of cluster Note, we only have 255 (`0x00` to `0xFE`) clusters in TAF.

Please restore TAT in the figure below.

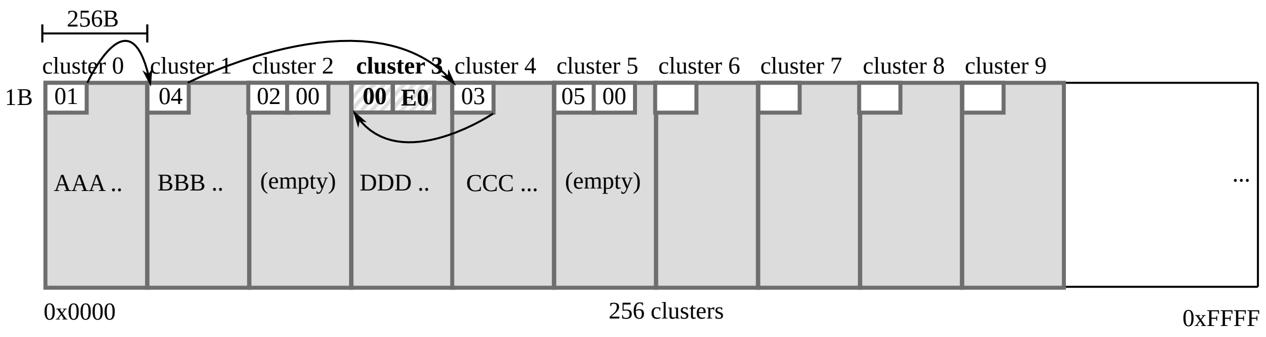


15. [10 points]: The current TAT design can only load/store data with a cluster unit (which is 256 B for this problem), failing to precisely indicate the valid data region (i.e., arbitrary file size). To overcome this problem, we have invented an *inlined-TAT* scheme that uses a first one-byte to point to the next cluster. As our one-byte address is not enough to represent the state of a cluster (i.e., all used to indicate 256 clusters in 64 KB), we introduce two rules: ❶ to indicate an empty/unused cluster, the *current* cluster number is stored (see cluster 2) in the first byte of the cluster, and *another byte* of `0x00` is stored right next to the first byte. ❷ to indicate the end of cluster, the *initial* cluster number is stored (see cluster 3) in the first byte of the cluster, and *another byte* which indicate the size of the data in the current cluster (`0xE0` in cluster 3) is stored right next to the first byte.

A. [2 points] What's the size of a file stored from cluster 0?

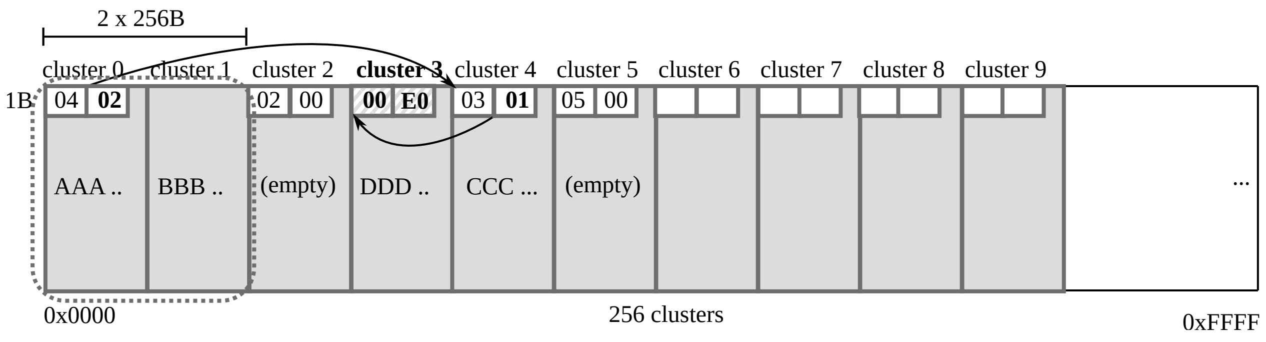
B. [2 points] Not just the size benefit, we found that TAF's *read latency* improves a bit compared to FAT (i.e., how does it quickly return the data). Why is that?

C. [6 points] However, it shows some glitches in writing (or appending) data to the file, because the size field in the last cluster should be removed and all data should be moved one byte forward. How would you address that?



Name: 16

D. [5 points (bonus)]: We found that DRAM (or any storage) performs faster when exhibiting sequential reads or writes. To exploit this behavior, we invented a scheme to record the usage of clusters in sequence. For example, cluster 0 and cluster 1 can be merged as one large cluster—the second byte in the cluster 0 indicates the number of clusters in the same group. Could you think of two disadvantages (or problems) of this scheme?



IV 3210: Design of Operating Systems

We'd like to hear your opinions about 3210, so please answer the following questions. (Any answer, except no answer, will receive full credit.)

16. [4 points]: Any feedback to TAs or to the instructor? What's your major concern on 3210? What should we do to improve them?

17. [2 points]: What is the best aspect of 3210?

18. [2 points]: What is the worst aspect of 3210?

19. [2 points]: Should we continue to offer one section of cs3210 as with RustOS, rather than a traditional OS class with C?

Name:

V Intentionally blank page

If you need extra space, feel free to use this page

End of Quiz

Name: