Computer Science

Georgia Institute of Technology

CS 3210 Spring 2020

Quiz II

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 24 hours to complete this quiz.

Please submit through Canvas following the submission guideline

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 COLLABORATION ALLOWED)

Please do not write in the boxes below.

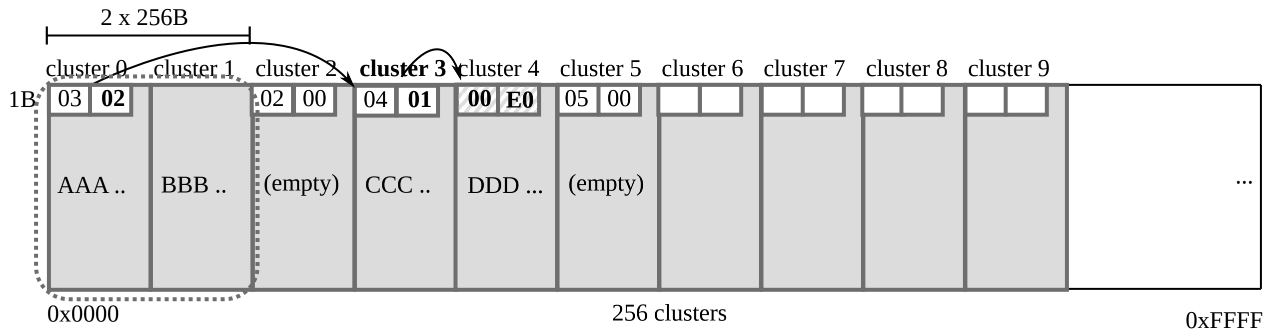
I (xx/30)	II (xx/30)	III (xx/30)	IV (xx/30)	V (xx/10)	Total (xx/130)

You only need 100 out of 130 points.

I TAF2: In-memory Filesystem for Tiny Allocation [30 points]

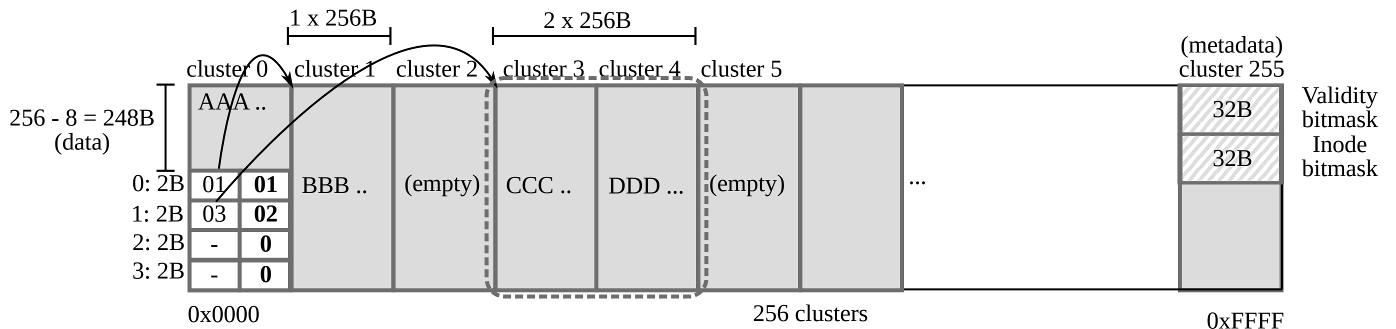
In Quiz I, we designed an in-memory filesystem, called TAF, that aims to provide Tiny Allocation Table (TAT) for small devices. It means TAF uses a byte addressable DRAM as storage, so no persistence guaranteed. To make it simple, we only consider the design spaces of TAF for 64 KB (2^{16}) DRAM.

One of the design that aims to maximize the fast speed of sequential reads or writes on DRAM is by *inline*-ing metadata (i.e., TAT-like) as part of the data (e.g., first two bytes of each cluster). More specifically, we invented a scheme to record the usage of clusters in sequence. For example, cluster 0 and 1 can be merged as one large cluster—the second byte in the cluster 0 indicates the number of clusters in the same group.



However, this scheme has two non-trivial problems: 1) *seeking* (i.e., reading a part of file) requires loading of *all* clusters of the file (i.e., required to chase the linked list); and 2) *allocating* a new block also requires scanning of the *entire* clusters in the worst case (i.e., checking the metadata of each cluster).

To solve these problems, we adopted inode-like structure in TAF2: each *leading* cluster (i.e., the first cluster of each entry like a file) contains the metadata of the entire file (i.e., which clusters belong to itself and which offset each cluster locates for the entry). In particular, the end of the leading cluster contains 4 tuples of a cluster offset (1 byte) and the number of clusters (1 byte), so total 8 bytes. The remaining 248 bytes in the leading cluster are used to store the data of the entry. The rest of the data will be stored in sequence from the cluster recorded in the metadata (see the figure below). To avoid the scanning on allocation, TAF2 keeps track of two system-wide bitmaps to indicate the validity of cluster (used or free) and the leading cluster (so called inode bitmap). For example, the validity bitmap mark the cluster 0/1/3/4 used and the inode bitmap mark the cluster 0 as a leading cluster in the figure below.



1. [5 points]: Please answer the questions below:

A. [2 points] How many bytes of data can be stored in the cluster chain starting from cluster 0 in the figure above?

B. [3 points] What's the "theoretical" maximum file size this scheme can store (i.e., not limited to the 64 KB limitation)?

2. [10 points]: Let's implement TAF2! We have provided an interface for `TAF2BlockDevice` that can read from and write to a cluster given a cluster id.

Please define `Cluster` and implement `read_data()`.

```
1  impl TAF2BlockDevice {
2      /// Read cluster number `cluster_id` and return
3      ///
4      /// # Errors
5      ///
6      /// Returns an error if seeking or reading from `self` fails.
7      fn read_cluster(&mut self, cluster_id: u64) -> io::Result<Cluster> { ... }
8  }
9
10 #[derive(Copy, Clone)]
11 enum Cluster {
12     // FIXME: Define Cluster
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32 }
```

```rust
struct TAF2 {
    device: TAF2BlockDevice,

    // bitmasks for clusters (loaded from the last cluster)
    bitmask_valid: [u8; 32],
    bitmask_inode: [u8; 32],
}

impl TAF2 {
    pub fn new(mut device: TAF2BlockDevice) -> TAF2 { ... }

    pub fn read_data(
        &mut self,
        cluster_id: u8,
        offset: usize,
        buf: &mut [u8],
    ) -> io::Result<usize> {
        // FIXME: implement read_data()




























    }
}
```

3. [10 points]: The current design works great at a glance and correctly addresses two problems we spotted in the original TAF's design. However, as we use the file system longer, we found that certain files in TAF2 are not allowed for *appending* new data.

A. **[4 points]** What's the worst case use of a leading cluster in terms of the storage capacity of each entry (i.e., why we can't append more data)? and why such scenarios likely happen when using TAF2 longer?

B. **[6 points]** Could you propose a new design that can address this problem? Is your design not suffering from the seeking and allocation problems? Please be very specific (i.e., giving us an example)!

4. [5 points]: As TAF2 works great under DRAM, we have ported it to a real storage backend, say SSD. Alas, we found that TAF2 is not robust against a system crash—if a power failure occurs when updating an OS, it may make the underlying OS unbootable!

Your answer should be concise and clear.

A. [1 point] Could you think of an example of such scenario? Why is it happening in TAF2?

B. [2 points] Could you suggest any design to make TAF2 robust against the crash? (i.e., not corrupting important data in the first place)

C. [2 points] Could you also think of any invariants that we can use to restore TAF2 as part of the recovery process (used in fsck)?

Name:

II VM Application: Demand Paging / Lazy Loading [30 points]

Demand paging or lazy loading is a popular methodology to reduce the latency of application launch as well as to avoid unnecessary page allocation for unused memory. The key idea is to mark all pages invalid when loading a program from a disk and to incrementally load the pages whenever the process first touches. At the moment the process accesses a page, the kernel will get an exception (i.e., data/code abort or page fault) as the page is marked invalid. However, the page fault handler in the kernel can load the corresponding page from the disk, map the page, and resume the process, making the process proceed as if nothing had happened.

How would you implement demand paging in your RustOS? Provide us how to modify your RustOS implementation (i.e., based on the code after Lab4) in the code snippet below. Please think carefully about the questions below. You don't have to answer them in text, but address them in your implementation.

A. How would you mark all pages of a process invalid?

B. How to distinguish normal data/code aborts from the ones relevant to demand paging?

C. How to distinguish lazy loading of text (code) section and demand paging for stack area?

D. How would you know which file to access on data/code abort? How would you know which location of the file should be accessed?

E. How can you prevent the stack from growing indefinitely?

F. Does your design still respect the original goals (two aforementioned)?

G. Is your code readable and concise?

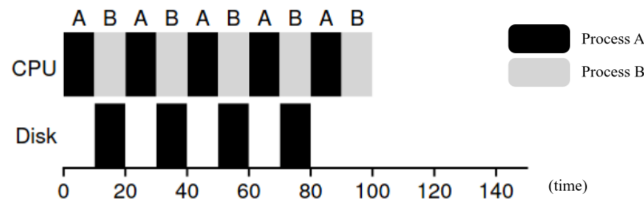
```
1  /* lib/fat32/src/traits/fs.rs */
2  // Assume this API has been implemented by the underlying file system
3  pub trait FileSystem: Sized {
4      /// Given a path and a page index, returns the contents in the offset range
5      /// `[page_idx * PAGE_SIZE, (page_idx + 1) * PAGE_SIZE)` of the file.
6      /// The length of the returned vector can be smaller than the page size when
7      /// reading the last page of the file.
8      ///
9      /// Returns `Err` if the page index is out of bound or any other file I/O
10     /// operation fails.
11     fn read_page_from_file<P: AsRef<Path>>(self, path: P, page_idx: usize)
12         -> io::Result<Vec<u8>> { ... }
13     ...
14 }
```

```
// start your implementation

/* kern/src/process/traps.rs */
pub extern "C" fn handle_exception(info: Info, esr: u32, tf: &mut TrapFrame) {
    match info.kind {
        Kind::Synchronous => {
            let syndrome = Syndrome::from(esr);
            match syndrome {



            }
        }
    }
...
}
```

Name:

III Blocking I/O and Scheduling [30 points]

We will implement a `read()` system call to load a certain region of a file to the userspace. It has two design goals: 1) an invocation of `read()` should *block* the execution of the userspace, making it easy to implement userspace programs (so called, blocking I/O); 2) instead of blocking the entire kernel, it should however *schedule* other processes until the data is completely loaded from the storage (SD card) and ready for use, like the figure below.



Unfortunately, our existing block device doesn't provide an appropriate interface for asynchronous data loading from the storage. In this problem, we simply assume a new file system abstraction that is built on top of asynchronous interface named `IoFuture`. It's a complicated subject, but for the sake of discussion, you can consider `IoFuture`, here, is a way to *poll* the result:

- `Poll::Pending` if the future is not ready yet
- `Poll::Ready(val)` with the result val of this future if it finished successfully.

When waiting asynchronously, schedule out the process by switching its state to `State::Waiting` like we did in `sys_sleep()`.

```
1  struct IoFuture<T> { ... }
2
3  impl<T> Future for IoFuture<T> {
4      type Output = io::Result<T>;
5
6      // assume the poll can be safely invoked in any context
7      fn poll(&mut self) -> Poll<Self::Output> { ... }
8  }
9
10 impl<'a, HANDLE: VFatHandle> FileSystem for &'a HANDLE {
11     type File = File<HANDLE>;
12     type Dir = Dir<HANDLE>;
13     type Entry = Entry<HANDLE>;
14
15     fn open<P: AsRef<Path>>(self, path: P) -> IoFuture<Self::Entry> { ... }
16 }
17
18 impl<HANDLE: VFatHandle> io::Read for File<HANDLE> {
19     fn read(&mut self, buf: &mut [u8]) -> IoFuture<usize> { ... }
20 }
```

Name:

Please implement a `read` system call with the following type signatures. Feel free to use types, functions and interfaces in your RustOS.

```
1  pub fn sys_read<P: AsRef<Path>>(path: P,
2                                  offset: usize,
3                                  len: usize,
4                                  tf: &mut TrapFrame) {
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44 }
```

Name:

IV Designing Signal Handling Interface [30 points]

There is no interface provided for userspace processes to communicate each other in RustOS. In this problem, we will design a signal handling interface for RustOS. We have three design goals: 1) each process can send a signal (with `u64` data) to any process (via `sys_ping()`); 2) each process can register a signal handler (via `sys_sigaction()`); and 3) the kernel will check and invoke the signal handler right before making a context switch to the process having a pending signal.

```
1  pub struct Process {
2      ...
3      signal_handler_addr: Option<usize>,
4      signal_pending: bool,
5      signal_data: u64,
6  }
7
8  pub fn sys_ping(pid: process::Id, data: u64, tf: &mut TrapFrame) {
9      tf.xs[7] = OsError::ProcessNotFound as u64;
10
11     SCHEDULER.critical(|scheduler| {
12         scheduler
13             .find_process_pid(pid)
14             .map(|p| {
15                 p.signal_pending = true;
16                 p.signal_data = data;
17                 tf.xs[7] = OsError::Ok as u64;
18             });
19     });
20 }
21
22 pub fn sys_sigaction(signal_handler_addr: usize, tf: &mut TrapFrame) {
23     SCHEDULER.critical(|scheduler| {
24         let mut p = scheduler.find_process(tf);
25         if signal_handler_addr == 0 {
26             p.signal_handler_addr = None;
27         } else {
28             p.signal_handler_addr = Some(signal_handler_addr);
29         }
30     });
31     tf.xs[7] = OsError::Ok as u64;
32 }
```

Please think carefully about five questions below (no need to answer, but address them in your implementation):

A. How to modify the trap frame to invoke the signal handler (with the data) to process the pending signal?

B. Remember that simply manipulating the current trap frame would *destroy* the previous context that

Name:

the process is supposed to resume.

C. How/where to keep the previous trap frame?

D. What should happen when a signal handler in the userspace is done processing the signal? (hint: a new system call?)

E. Once the signal handler is done processing, how to correctly resume the execution of the target process?

```
impl Scheduler {
    fn switch_to(&mut self, tf: &mut TrapFrame) -> Option<Id> {





















    }
}
```

Name:

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52

Name:

V 3210: Design of Operating Systems

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

A. [1 point] Do you think you will continue to learn and use Rust?

B. [1 point] Are you going to check Lab5?

C. [2 points] How would you recommend RustOS to your friends?

D. [2 points] What is the best and worst aspect of 3210?

E. [2 points] Should we continue to offer one section of CS3210 as with RustOS, rather than a traditional OS class with C?

F. [2 points] Please leave any feedbacks and comments you might have.

P.S. We know for sure you will miss RustOS after this semester! Hope you learned "a lot" from CS3210, and best of luck for your journey and career.

VI Intentionally blank page

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

End of Quiz