CS 162 Summer 2022 Solutions

Final Exam

INSTRUCTIONS

Please do not open this exam until instructed to do so.

Do not discuss exam questions for at least 24 hours after the exam ends, as some students may be taking the exam at a different time.

For questions with **circular bubbles**, you should select exactly *one* choice.

○ You must choose either this option

○ Or this one, but not both!

For questions with **square checkboxes**, you may select *multiple* choices.

☐ You could select this choice.

☐ You could select this one too!

Preliminaries

This is a proctored, closed-book exam. You are allowed 2 pages of notes (both sides). You may not use a calculator. You have 110 minutes to complete as much of the exam as possible. Make sure to read all the questions first, as some are substantially more time-consuming.

If there is something about the questions you believe is open to interpretation, please ask us about it.

We will overlook minor syntax errors when grading coding questions. **There is a reference sheet at the end of the exam that you may find helpful.**

(a) Name

(b) Student ID

(c) Please read the following honor code: "I understand this is a closed-book exam. I promise the answers I give on this exam are my own. I understand that I am allowed to use two 8.5x11, double-sided, handwritten cheat-sheets of my own making, but otherwise promise not to consult other people, physical resources (e.g. textbooks), or the internet in constructing my answers."

Write your full name below to acknowledge that you've read and agreed to this statement.

1. (16 points) True/False

Please explain your answer in **TWO SENTENCES OR LESS**. Longer explanations may get no credit. Answers without an explanation **GET NO CREDIT**.

(a) (2 points)

i. In Linux, a user program will terminate if it page faults.

○ True.

[✓ ● False.]

ii. Explain.

Stack growth and paging out to disk are counterexamples.

(b) (2 points)

i. Regardless of the distribution of job arrival times and job servicing times, queuing delay will grow unboundedly as utilization approaches 1.

○ True.

[✓ ● False.]

ii. Explain.

Consider constant arrival and service rates.

(c) (2 points)

i. Increasing the amount of physical memory while keeping page size constant will always decrease the number of page faults (assuming we use the same demand paging policy).

○ True.

[✓ False.]

ii. Explain.

FIFO and Belady's anomaly.

(d) (2 points)

i. A program in an unsafe state will deadlock (or crash before doing so).

○ True.

[✓ False.]

ii. Explain.

Programs don't necessarily acquire all their resources at once.

(e) (2 points)

i. FAT is slow when reading from a random location in a large file.

[✓ True.]

False.

ii. Explain.

FAT has linear traversal time as it uses a linked list implementation of files.

(f) (2 points)

i. Using a buffer cache is an example of demand paging.

True.

[✓ False.]

ii. Explain.

The buffer cache is not the same as demand paging. The buffer cache caches disk blocks in memory; demand paging saves pages of memory to disk.

(g) (2 points)

i. The function $f(x) = x + 1$ is idempotent.

○ True.

[✓ False.]

ii. Explain.

f(f(x)) != f(x)

(h) (2 points)

i. Compared to single-level page tables, multi-level page tables use less memory for sparse address spaces.

[✓ True.]

○ False.

ii. Explain.

You don't need to allocate every table when the address space is sparse.

2. (16 points) Multiple Choice

Choose ALL that apply.

(a) (2 pt) Which of the following are approximation(s) to MIN?

- ■ Second-chance list
- ■ Clock
- ☐ FIFO
- ■ LRU
- ☐ None of the above.

LRU approximates MIN, and clock/second-chance approximate LRU.

(b) (2 pt) Which of the following are true about I/O?

- ☐ The top half of a device driver polls the bottom half of the device driver.
- ☐ Memory-mapped I/O is an example of direct memory access.
- ☐ In port-mapped I/O, each device port is mapped to a unique location on disk.
- ☐ In programmed I/O, the CPU programs an external controller to do I/O while the CPU can work on other things.
- ■ None of the above.

None of the options are correct. A: Top half and bottom half refer to interrupt-driven I/O; polling is not used. B: Memory-mapped I/O actively involves the CPU; in DMA, an external device writes data to memory independently, then interrupts the CPU when the data is ready. C: Port-mapped I/O does not require involvement of the disk. D: Programmed I/O involves special CPU instructions (eg. in and out), not an external I/O controller.

(c) (2 pt) Which of the following are true about storage devices?

- ■ Sequential reads are faster than random reads in HDDs.
- ☐ Constantly writing and erasing the same memory location will wear out HDDs faster than SSDs.
- ☐ SSDs are generally cheaper than HDDs (for the same amount of storage).
- ☐ Like RAM, storage devices are byte-addressed.
- ☐ None of the above.

A: Sequential reads on HDDs are usually faster than random reads because we only have to wait for the disk to spin under the head. B: SSD pages wear out when they are erased; HDDs do not wear out as quickly. C: SSDs are generally more expensive. D: Storage devices are addressed in units of sectors/pages/blocks.

(d) **(2 pt)** Which of the following are true about cache misses?

- [✓] Coherence misses can occur when multiple processors share a single cache.
- [✓] Conflict misses can be reduced by reducing the number of index bits.
- [ ] Capacity misses grow linearly with the size of the cache.
- [ ] Compulsory misses grow linearly with the size of the cache.
- [ ] None of the above.

A: In a multiprocessor system, actions by one processor can invalidate the cache entries for another processor, possibly resulting in a coherence miss. B: A cache with a higher associativity can reduce the number of conflict misses. C: Making a cache larger should decrease the number of capacity misses. D: Compulsory misses occur when a cache line is loaded for the first time; increasing the size of the cache has no effect on the number of compulsory misses.

(e) **(2 pt)** Which of the following are true about this Rust program?

```
fn main() {
    let s = "General".to_string();
    let s = "General Kenobi".to_string();
    kenobi(s);
    println!("{}", s);
}

fn kenobi(s: String) {
    if s == "General Kenobi" {
                println!("Hello there");
    }
}
```

- [ ] The program reuses `s` without declaring `s` as mutable, which is not allowed.
- [ ] `main` will error at RUNTIME because it tries to use a variable it does not own.
- [✓] The program will compile if we delete the print statement in `main`.
- [✓] The program will compile if we replace `kenobi(s: String)` and `kenobi(s)` with `kenobi(s: &str)` and `kenobi(&s)`, respectively.
- [ ] None of the above.

A: The second `s` is a new variable due to the `let` declaration. It is not a reassignment of the original `s`. B: This program will result in compile-time, not run-time, errors. C: This solves the issue; `s` would no longer be used after being moved into `kenobi`. D: Passing a reference to `s` also fixes the problem, as ownership of `s` is not moved into `kenobi`.

(f) (2 pt) Which of the following are true about deadlocks?

- ■ Deadlocks can be provably avoided by defining a global resource acquisition order.
- ■ Deadlocks can ONLY occur if there is a circular wait for resources.
- ☐ Strict priority schedulers use priority donation to prevent deadlocks.
- ■ A program in a safe state can eventually deadlock.
- ☐ None of the above.

A: Circular wait is a necessary condition for deadlock. B: Circular wait is a necessary condition for deadlock. C: Priority donation prevents starvation of a high-priority thread; it does not prevent deadlock. D: A safe state means that there is a non-blocking order of threads that allows all threads to complete. Deadlock can still occur starting from a safe state (eg. if the non-blocking ordering is not followed).

(g) (2 pt) Which of the following are true about file numbers (inumbers)?

- ■ A directory is a file containing file name and file number mappings.
- ■ In FAT, the file number is the index of the first block of a file in the file allocation table.
- ■ A file number uniquely identifies and locates a file on disk.
- ■ In FFS, the file number is the index of an inode in the inode array.
- ☐ None of the above.

All are correct; see lectures 19-21.

(h) (2 pt) Which of the following are optimizations introduced by Berkeley FFS?

- ☐ Introducing skip-lanes to the freelist for faster deallocation of used blocks.
- ☐ Using variable-sized extents to accommodate files too large to be stored in an inode-based format.
- ■ A first-free allocation of new file blocks which yields sequential blocks for large files.
- ■ Putting a directory and its file into common block groups.
- ☐ None of the above.

See lecture 20.

3. (15 points) Short Answer

Please answer in **THREE SENTENCES OR LESS**. Longer explanations may get no credit.

(a) (3 pt) Explain why the FAT file system does not support hard links.

FAT cannot reference count because file metadata is stored in the parent directory.

(b) (3 pt) What is the relationship between `struct file` and `struct inode` in PintOS?

A `struct file` contains a `struct inode`. There can be many `struct file`'s referencing the same `struct inode`.

(c) (3 pt) Is port-mapped I/O or DMA preferable for storage devices? Explain.

DMA does not require CPU cycles (eg. polling), so it is preferable for disk accesses, which can potentially take millions of CPU cycles.

(d) **(3 pt)** Under what conditions can we overlap cache and TLB lookups?

Cache lookups can be overlapped when the page offset in the virtual address contains the cache index. See lecture 15, slide 26.

(e) **(3 pt)** Oh no! Darth Vader broke one of his disks and lost his Death Star plans. Luckily, Darth Vader was using RAID 5 and still has the other 4 disks. The contents of these disks are

```
Disk 1: 100010
Disk 2: 010110
Disk 3: 110100
Disk 4: 011101
```

What were the contents of Disk 5? Explain how you got your answer.

The contents of disk 5 are 011101. This can be calculated by taking the bitwise XOR of all the data on disks 1, 2, 3, and 4.

4. (12 points) Potpourri

(a) (4 pt) Suppose we have a hard drive with the following specifications:

- An average seek time of 10 ms.
- A rotational speed of 6000 revolutions per minute.
- A controller that can transfer data at a rate of 40 MB/s.

What is the expected throughput of the hard drive when reading a 4 KB sector from a random location on disk? You may ignore queueing delay. Please leave your answer as a fraction (KB/ms) in simplest form. We may give partial credit if you show your work.

The total time is the seek time plus the rotation time plus the transfer time. Since the disk runs at 6000 RPM, a single revolution takes 10ms. So the average rotation time is 5ms. Transferring 4KB at a rate of 40 MB/s takes 0.1ms.
Thus, the total time is 10ms + 5ms + 0.1ms = 15.1ms. So the expected throughput is 4 KB/15.1 ms.

(b) (4 pt) Anakin Skywalker has a computer that

- Has a single-level page table.
- Has a TLB.
- Has a single cache.

Furthermore, suppose

- Each main memory access takes 100ns.
- Each TLB access takes 10ns.
- The cache has a 30ns lookup time.
- Cache and TLB lookups do not overlap.
- Both the TLB and the cache have a 90% hit rate.

What is the expected time to read from a location in memory? Please leave your answer in ns. We may give partial credit if you show your work.

The average cache lookup time is C = 0.9 * 30ns + 0.1(30ns + 100ns) = 40ns. The AMAT is 0.9 * (10ns + C) + 0.1 * (10ns + 2C) = 54ns.

(c) **(4 pt)** Suppose we have resources A, B, C and threads T1, T2, T3, T4. The total number of each resource as well as the current/max allocations for each thread are as follows:

Total Allocation:

A	B	C
5	6	7

Current Allocation:

T\R	A	B	C
T1	1	2	0
T2	0	1	1
T3	2	0	3
T4	1	1	1

Max Allocation:

T\R	A	B	C
T1	2	4	3
T2	3	2	5
T3	3	4	3
T4	2	3	3

Is this system in a safe state? If so, provide a non-blocking ordering of threads. If not, explain why.

This is a safe state. There are 2 possible orderings: T4, T1, T2, T3, or T4, T1, T3, T2.

5. (10 points) Page Table Lookup

For online exam takers: you may find it useful to read this problem in a separate PDF instead of scrolling back and forth.

Consider a multi-level memory management scheme with the following format for virtual addresses:

```
+-----------------------------------------------+
| Virtual Page #1 | Virtual Page #2 |   Offset   |
|     (10 bits)   |     (10 bits)   |  (12 bits) |
+-----------------------------------------------+
```

Virtual addresses are translated into physical addresses of the following form:

```
+--------------------------------+
| Physical Page # |   Offset     |
|    (20 bits)    |  (12 bits)   |
+--------------------------------+
```

Page table entries (PTE) are 32 bits in the following form:

```
+-----------------------------------------------------------------+
| Physical Page # |    OS    | 0 | 0 | D | A | N | WT | U | W | V |
|    (20 bits)    | (3 bits) |   |   |   |   |   |    |   |   |   |
+-----------------------------------------------------------------+
```

where `D, A, N, WT, U, W, V` are status bits. In this problem, `V = 1` means the PTE is valid. Otherwise `V = 0`. You do NOT need to worry about any of the other status bits.

Translating virtual addresses to physical addresses typically occurs in hardware. However, Kenobi decides to write a kernel function that takes a 32-bit virtual address and returns the corresponding 32-bit physical address. Help Kenobi implement his `lookup` function by filling in the code below. You can ONLY call functions that are provided below.

Hint: << and >> operators may be helpful.

```
#include <inttypes.h>
#include <stdbool.h>

/*
 * Kernel page fault handler.
*/
void page_fault_handler(uint32_t vaddr) {
    ...
}

/*
 * Gets the 32-bit word at the 32-bit physical address PADDR.
*/
uint32_t get_word(uint32_t paddr) {
    ...
}

/*
 * ZEROS OUT bits of INPUT from start index to end index, including START but not END.
 * Index 0 is the least significant bit. Index 31 is the most significant bit.
 * 0 <= start < end <= 32 must be true or zero will panic.
*/
uint32_t zero(uint32_t input, int start, int end) {
    ...
}
```

```
/*
 * Page address translation in the kernel.
 * VADDR is the 32-bit virtual addr to be translated.
 * PT is the 32-bit physical addr of the base page table. It is page-aligned.
 * This function returns the 32-bit physical addr corresponding to VADDR.
 * If VADDR is not mapped, call page_fault_handler(vaddr) and return 0 immediately.
 * (In other words, if lookup returns 0, then either VADDR was not mapped
 * or VADDR maps to 0x00000000.)
*/
uint32_t lookup(uint32_t vaddr, uint32_t pt) {
    uint32_t vpn[2];
    uint32_t ptentry;
    bool valid;

    ________________[A (x3)]__________________

    for (int i = 0; i < 2; i++) {
        ________________[B (x8)]__________________
    }
    ________________[C (x1)]__________________
}
```

(a) (2 pt) [A] (max 3 lines)

```
vpn[0] = vaddr >> 22;
vpn[1] = zero(vaddr >> 12, 22, 32);
```

(b) (6 pt) [B] (max 8 lines)

```
ptentry = get_word(pt + 4*vpn[i]);
if (pt_entry & 1 == 0) {
    page_fault_handler(vaddr);
    return 0;
}
pt = zero(pt_entry, 0, 12);
```

(c) (2 pt) [C] (max 1 line)

```
return pt + zero(vaddr, 12, 32);
```

6. (21 points) Journaling Key-Value Store

For online exam takers: you may find it useful to read this problem in a separate PDF instead of scrolling back and forth.

In this problem, we will implement a reliable key-value store.

The key-value store will support two operations:

- `PUT(list(k, v))`: Stores an ordered **list** of key-value pairs into the key-value store in a single transaction. Duplicate keys may be present.
- `GET(k)`: Gets the value corresponding to a single key.

`PUT` operations act like transactions; they must be **atomic**. That is, suppose a client executes this `PUT`:

```
x = 5
y = 7
x = 2
```

No clients should see the temporary assignment of `x` to `5`. A `PUT` operation is considered committed if and when the `KvStore::put` method returns `Ok(())`.

The key-value store must be **reliable**: data from *committed* `PUT` operations must be retained even if the key-value store crashes and restarts. Data written to disk (eg. the log) will be retained between crashes and is loaded at startup.

To achieve reliability, we will use **journaling**. We will maintain a durable log containing a record of operations applied to the key-value store. A log entry is represented by the `LogEntry` struct.

You are given an implementation of a log for a key-value store; its API is shown in the code below. You are also given some helper functions for interacting with the filesystem, but you should NOT need to use them in the sections of code you write.

For performance reasons, there are some restrictions on when and how you can acquire locks:

- You may never acquire the `log` lock at the same time as the `data` lock, except in `checkpoint`.
- You may never acquire the `log` lock in write mode, except in `checkpoint`.
- Your implementation must not be prone to deadlock.

Remember that calling `drop` on a lock guard causes the lock to be released immediately.

Fill in the implementations of `get` and `checkpoint`.

- `get` returns the value corresponding to the given `key` as of the last *committed* transaction. We consider the last transaction to be the one with the highest transaction ID.
- `checkpoint` *safely* deletes all log entries corresponding to committed transactions. A crash during `checkpoint` must not result in data loss. Log entries for uncommitted transactions should be kept in the log.

Your key-value store should provide correct results regardless of if or when a crash occurs. Functions marked **atomic** in the skeleton code can be assumed to complete successfully, or not complete at all. In other words, a crash will not cause an atomic operation to only partially complete. (Non-atomic operations, on the other hand, may be partially complete if a crash occurs.)

```
//! A durable key-value store that uses a write-ahead log.

use anyhow::Result;
use std::{
    collections::HashMap,
    sync::Arc,
};
use tokio::sync::RwLock;

type TransactionId = u64;
```

```
struct KvStore {
    log: Arc<RwLock<Log>>,
    data: Arc<RwLock<HashMap<String, String>>>,
}

struct Log {
    // Hidden
}

struct LogEntry {
    tx_id: TransactionId,
    record: LogRecord,
}

#[derive(Eq, PartialEq)]
enum LogRecord {
    BeginTx,
    Put(String, String),
    CommitTx,
}

impl Log {
    /// Load a log from `path`, or create a new one if `path` does not exist or is corrupted.
    /// NOT atomic.
    async fn new(path: &str) -> Result<Self> { ... }

    /// Begins a transaction, returning the transaction ID. ATOMIC.
    async fn begin(&self) -> Result<TransactionId> { ... }

    /// Records a SINGLE key-value pair. NOT atomic.
    async fn put(&self, tx_id: TransactionId, k: &str, v: &str) -> Result<()> { ... }

    /// Marks the given transaction as committed. ATOMIC.
    async fn commit(&self, tx_id: TransactionId) -> Result<()> { ... }

    /// Returns a list of all log entries.
    async fn entries(&self) -> Result<Vec<LogEntry>> { ... }

    /// Returns true if the given transaction was committed.
    async fn committed(&self, tx_id: TransactionId) -> bool { ... }

    /// Deletes all log entries, and writes the given entries in their place. NOT atomic.
    /// If a crash occurs during this function, the log will become corrupted.
    /// Subsequent calls to `Log::new` will provide an empty log.
    async fn set(&mut self, entries: Vec<LogEntry>) -> Result<()> { ... }
}

/// Saves `data` at `path`. NOT atomic.
async fn persist(path: &str, data: &HashMap<String, String>) -> Result<()> { ... }
/// Loads a saved HashMap from `path`. NOT atomic.
async fn load(path: &str) -> Result<HashMap<String, String>> { ... }
/// Renames file `src` to `dst`. ATOMIC.
async fn rename(src: &str, dst: &str) -> Result<()> { ... }
/// Deletes a file saved at `path`. ATOMIC.
```

```
async fn remove(path: &str) -> Result<()> { ... }


const KV_LOG: &str = "kv.log";
const KV_DATA: &str = "kv.dat";
const TMP_DATA: &str = "tmp.dat";

impl KvStore {
    async fn init() -> Result<Self> {
        Ok(Self {
            log: Arc::new(RwLock::new(Log::new(KV_LOG).await?)),
            data: Arc::new(RwLock::new(HashMap::new())),
        })
    }

    async fn put(&self, entries: Vec<(String, String)>) -> Result<()> {
        let log = self.log.read().await;

        let tx_id = log.begin().await?;
        for (k, v) in entries {
            log.put(tx_id, &k, &v).await?;
        }

        log.commit(tx_id).await?;
        Ok(())
    }

    async fn get(&self, key: String) -> Result<Option<String>> {
        let log = self.log.read().await;
        let entries = log.entries().await?;

        ________________[A (x25)]__________________
    }

    async fn checkpoint(&self) -> Result<()> {
        let mut data = self.data.write().await;
        let mut log = self.log.write().await;

        let entries = log.entries().await?;
        let mut keep = Vec::new();

        ________________[B (x20)]__________________

        persist(TMP_DATA, &data).await?;
        rename(TMP_DATA, KV_DATA).await?;
        let _ = remove(TMP_DATA).await;

        log.set(keep).await?;

        Ok(())
    }
}
```

Note: When grading, we will overlook Rust errors that are easily fixable. You may find the Rust reference sheet and the Rust examples sheet helpful.

(a) **(12 pt)** [A] (max 25 lines)

```
let mut last_tx = 0;
let mut value = None;
for record in records {
    if let LogEntry::Put(k, v) = record.entry {
        if k == key && log.committed(record.tx_id).await {
            if record.tx_id >= last_tx {
                last_tx = record.tx_id;
                value = Some(v);
            }
        }
    }
}
if value.is_some() {
    return Ok(value);
}
drop(log);
let data = self.data.read().await;
Ok(data.get(&key).cloned())
```

We first check the log to see if there is a committed transaction that modified the key we are looking for. We want `GET` to return the value saved by the last (ie. highest ID) transaction, so we create a temporary variable called `last_tx` to store the highest transaction ID we've seen so far that modifies the given key.

For each log record, we check if it is a PUT record, modifies the right key, is committed, and has a transaction ID greater than or equal to `last_tx`.

If no value is found in the log, we check the `data` `HashMap`.

(b) **(9 pt)** [B] (max 20 lines)

```
let mut last_tx = HashMap::new();
for record in records {
    if log.committed(record.tx_id).await {
        if let LogEntry::Put(k, v) = record.entry {
            if record.tx_id >= last_tx.get(&k).copied().unwrap_or(0) {
                last_tx.insert(k.clone(), record.tx_id);
                data.insert(k, v);
            }
        }
    } else {
        keep.push(record);
    }
}
```

We want to clear all log entries from committed transactions. So we loop through the log entries. If an entry is not committed, we save it in the `keep` list. For committed entries, we update the `data` `HashMap` only if the transaction is committed, the log entry is a `PUT` entry, and no later transaction set a value for the key in the log entry.

To see why the last condition is necessary, consider the following log (which is not prevented by the given implementation of `put`):

```
15: BEGIN TX
16: BEGIN TX
16: PUT x = 2
15: PUT x = 1
COMMIT 16
COMMIT 15
```

We want to see `x=2`, since this is the value of `x` from the transaction with the highest ID (16). However, the PUT record for transaction 15 appears later in the log than the PUT record for transaction 16.

7. Rust Reference Sheet

```
/********************************** Options *********************************/
pub enum Option<T> {
    None,
    Some(T),
}
impl<T> Option<T> {
    pub const fn is_some(&self) -> bool;
    pub const fn is_none(&self) -> bool;
    pub const fn as_ref(&self) -> Option<&T>;
    pub fn as_mut(&mut self) -> Option<&mut T>;
    pub fn cloned(self) -> Option<T> where T: Clone;
    pub fn unwrap(self) -> T;
    pub fn map<U, F>(self, f: F) -> Option<U> where F: FnOnce(T) -> U;
    pub fn take(&mut self) -> Option<T>;
}
/********************************** Results *********************************/
pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
impl<T, E> Result<T, E> {
    pub const fn is_ok(&self) -> bool;
    pub const fn is_err(&self) -> bool;
    pub fn ok(self) -> Option<T>;
    pub fn err(self) -> Option<E>;
    pub fn unwrap(self) -> T where E: Debug;
    pub fn map<U, F>(self, op: F) -> Result<U, E> where F: FnOnce(T) -> U;
    pub fn map_err<F, O>(self, op: O) -> Result<T, F> where O: FnOnce(E) -> F;
}
/********************************** Collections *********************************/
impl<T> Vec<T, Global> {
    pub const fn new() -> Vec<T, Global>;
}
impl<T, A> Vec<T, A> where A: Allocator {
    pub fn clear(&mut self);
    pub fn push(&mut self, value: T);
    pub fn pop(&mut self) -> Option<T>;
    pub fn insert(&mut self, index: usize, element: T);
    pub fn len(&self) -> usize;
    pub fn remove(&mut self, index: usize) -> T;
}
impl<K, V> HashMap<K, V, RandomState> {
    pub fn new() -> HashMap<K, V, RandomState>;
    pub fn clear(&mut self);
    pub fn contains_key<Q: ?Sized>(&self, k: &Q) -> bool where K: Borrow<Q>, Q: Hash + Eq;
    pub fn get<Q: ?Sized>(&self, k: &Q) -> Option<&V> where K: Borrow<Q>, Q: Hash + Eq;
    pub fn get_mut<Q: ?Sized>(&mut self, k: &Q) -> Option<&mut V> where K: Borrow<Q>, Q: Hash + Eq;
    pub fn insert(&mut self, k: K, v: V) -> Option<V>;
    pub fn len(&self) -> usize;
}
/********************************** Miscellaneous *********************************/
impl<T> Arc<T> {
    pub fn new(data: T) -> Arc<T>;
}
```

```
impl<T> Deref for Arc<T> where T: ?Sized { type Target = T; ... }
impl<T> Clone for Arc<T> where T: ?Sized;
impl<T: ?Sized> RwLock<T> {
    pub fn new(value: T) -> RwLock<T> where T: Sized;
    pub async fn read(&self) -> RwLockReadGuard<'_, T>;
    pub async fn write(&self) -> RwLockWriteGuard<'_, T>;
}
impl<T: ?Sized> Deref for RwLockReadGuard<'_, T> { type Target = T; ... }
impl<T: ?Sized> Deref for RwLockWriteGuard<'_, T> { type Target = T; ... }
impl<T: ?Sized> DerefMut for RwLockWriteGuard<'_, T> { ... }
pub fn drop<T>(_x: T);
println!("Hello, {}!", "world");
pub trait Clone {
    fn clone(&self) -> Self;
    fn clone_from(&mut self, source: &Self) { ... }
}
```

8. Rust Examples

```
//! Examples of Rust syntax.
//! This code compiles, but doesn't do anything useful.
//! Running this code does not produce any panics.
use std::{collections::HashMap, sync::Arc};

use tokio::sync::RwLock;

struct Coordinate {
    x: i32,
    y: i32,
}

enum OperatingSystem {
    Mac,
    Windows,
    Linux,
    Other(String),
}

impl Coordinate {
    fn add(&mut self, other: Self) {
        self.x += other.x;
        self.y += other.y;
    }
}

#[tokio::main]
async fn main() {
    let mut x: i32 = 4;
    x += 1;

    let ptr = &mut x;
    *ptr += 1;

    assert_eq!(x, 6);

    let x = 123;
    let mut pt = Coordinate { x, y: 3 };
    assert_eq!(pt.x, 123);
    pt.add(Coordinate { x: 5, y: pt.y });

    assert_eq!(square(4), 16);

    let mut courses = vec!["161".to_string(), "162".to_string(), "164".to_string()];
    // Iterate by reference
    for course in courses.iter() {
        let tmp: &str = course;
        println!("Course: {}", tmp);
    }
    courses.push(String::from("169"));

    let mut map = HashMap::new();

    let mut first_num = None;
    // Transfer ownership of the entries in `courses`
    for course in courses {
        let num: i32 = course.parse().unwrap();
        if first_num.is_none() {
            first_num = Some(num);
        }
        map.insert(course, num);
    }

    match first_num {
        Some(num) => println!("First num: {}", num),
        None => println!("None"),
    }

    let cs161 = map.get("161").copied().unwrap_or(0);
    assert_eq!(cs161, 161);

    let my_os = OperatingSystem::Other(String::from("Redox"));
    if let OperatingSystem::Other(os) = my_os {
        map.insert(os, 162162);
    } else {
        println!("Someone has a common OS.");
    }

    let my_os = OperatingSystem::Mac;
    let name: &str = match my_os {
        OperatingSystem::Mac => "Mac",
        OperatingSystem::Linux => "Linux",
        _ => "Windows/Other",
    };

    let state = Arc::new(RwLock::new(Coordinate { x: 2, y: 5 }));

    // Multiple readers can immutably access the data at the same time
    let r1 = state.read().await;
    let r2 = state.read().await;

    println!("x = {}, y = {}", r2.x, r2.y);

    // Release the lock by dropping the guards
    drop(r1);
    drop(r2);

    let mut w1 = state.write().await;
    w1.x += 6;
    assert_eq!(w1.x, 8);
    drop(w1);
}

fn square(x: i32) -> i32 {
    x * x
}
```