Lazy Data Movement in Pluton OS

Tock World 8 – Redmond, WA
9/5/2025
Hussain Miyaziwala

Goal: Design HILs that don’t require static buffers

```
enum State {
    Start {
        aes_arb: &'static MutexRef<Aes>,

        plaintext: &'static mut [u8],
        ciphertext: &'static mut [u8],
        key: &'static mut [u8],
        iv: &'static mut [u8],
        gcm_tag: &'static mut [u8],
        gcm_aad: &'static mut [u8],
    },
```

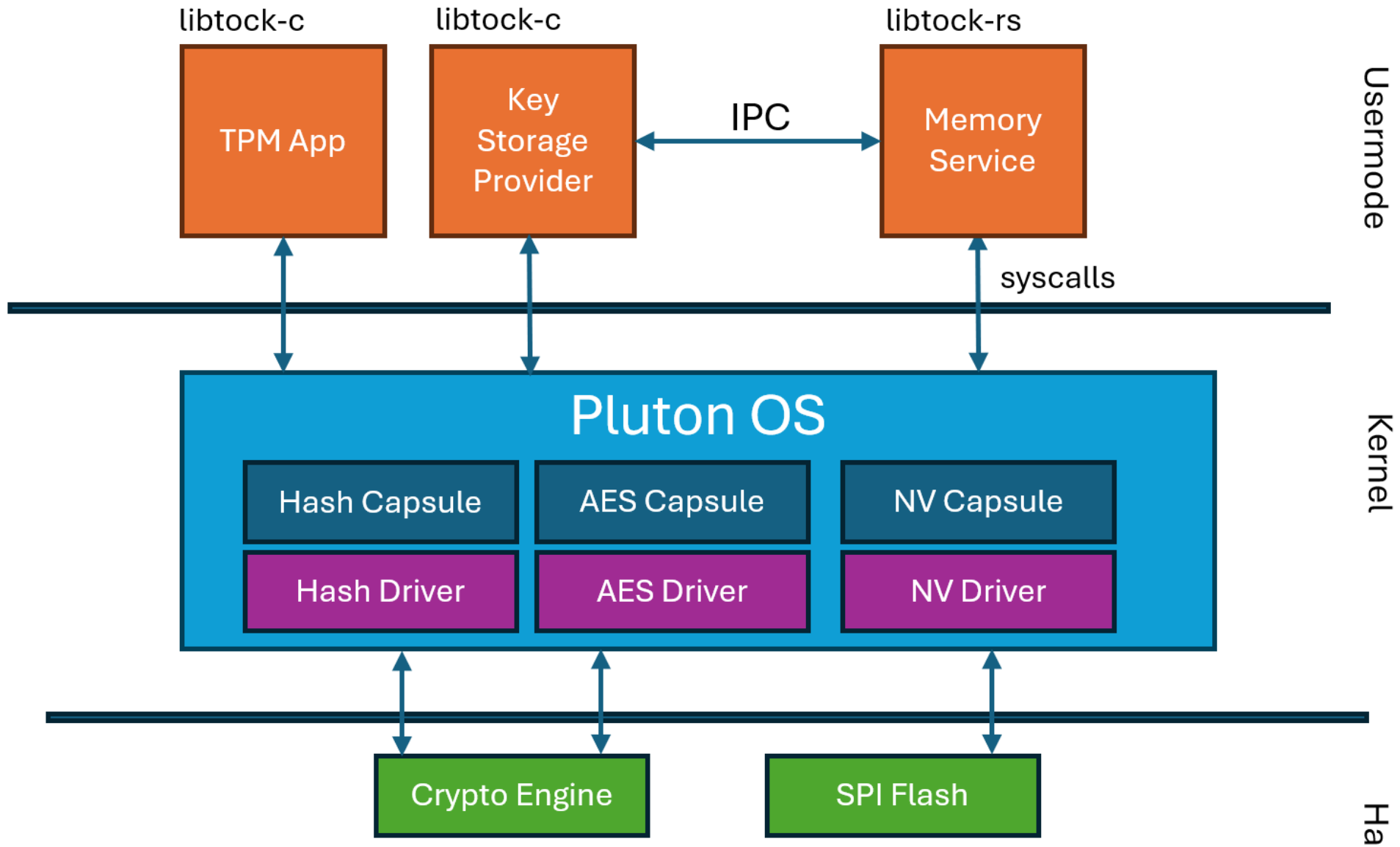
```
enum State {
    Start {
        aes_arb: &'static MutexRef<Aes>,
    },
```

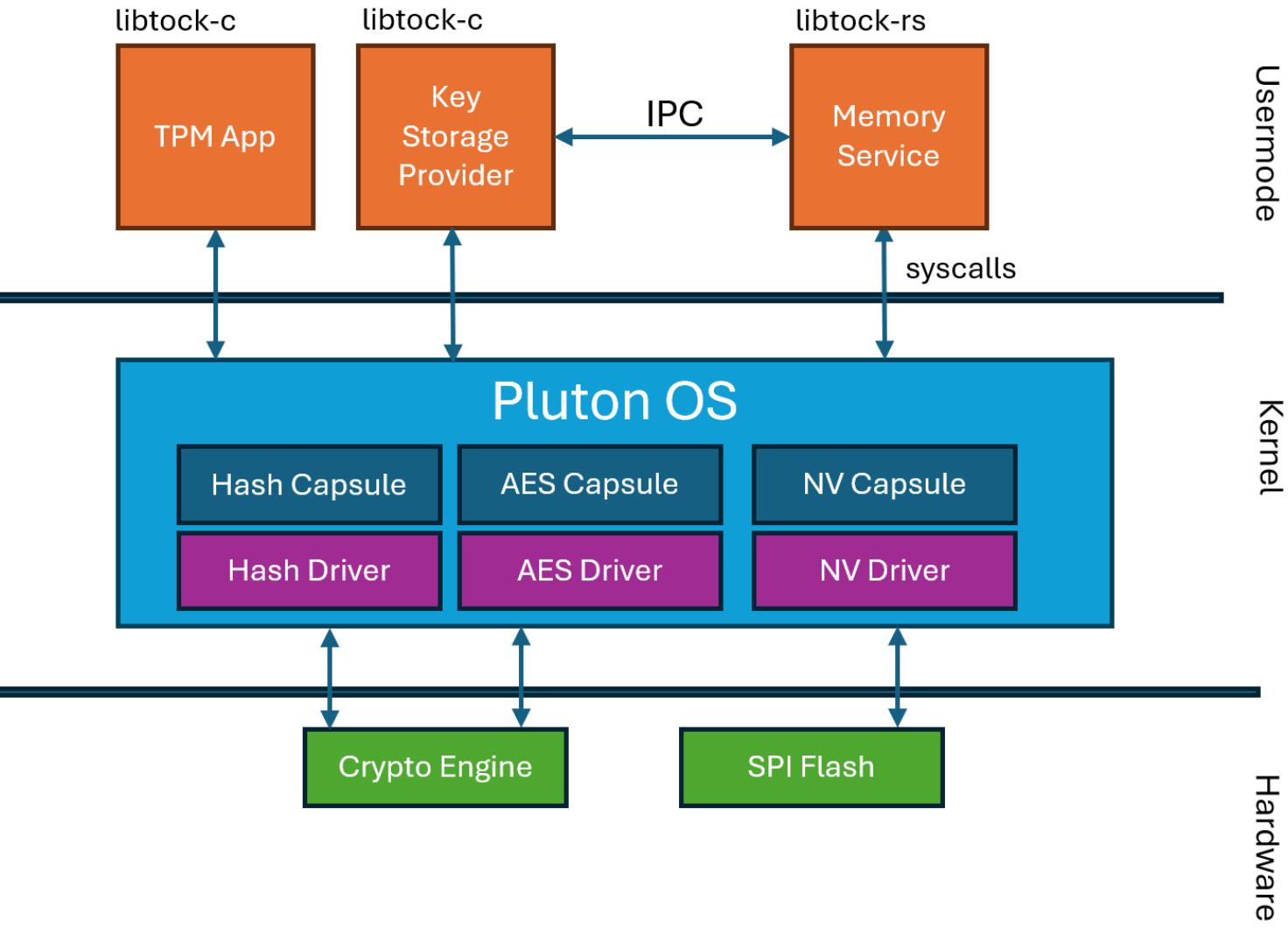
Hello!

- Software Developer at Microsoft
- Joined out of college in 2022
- Pluton Firmware Team
- Seattle -> Atlanta

Microsoft Pluton

- Secure crypto-processor built into some AMD and Intel CPUs
- Solves various existing HW security problems in industry
  - physical attacks
  - inconsistent updates
  - supply chain risks
- A platform for TPM 2.0 and future security products

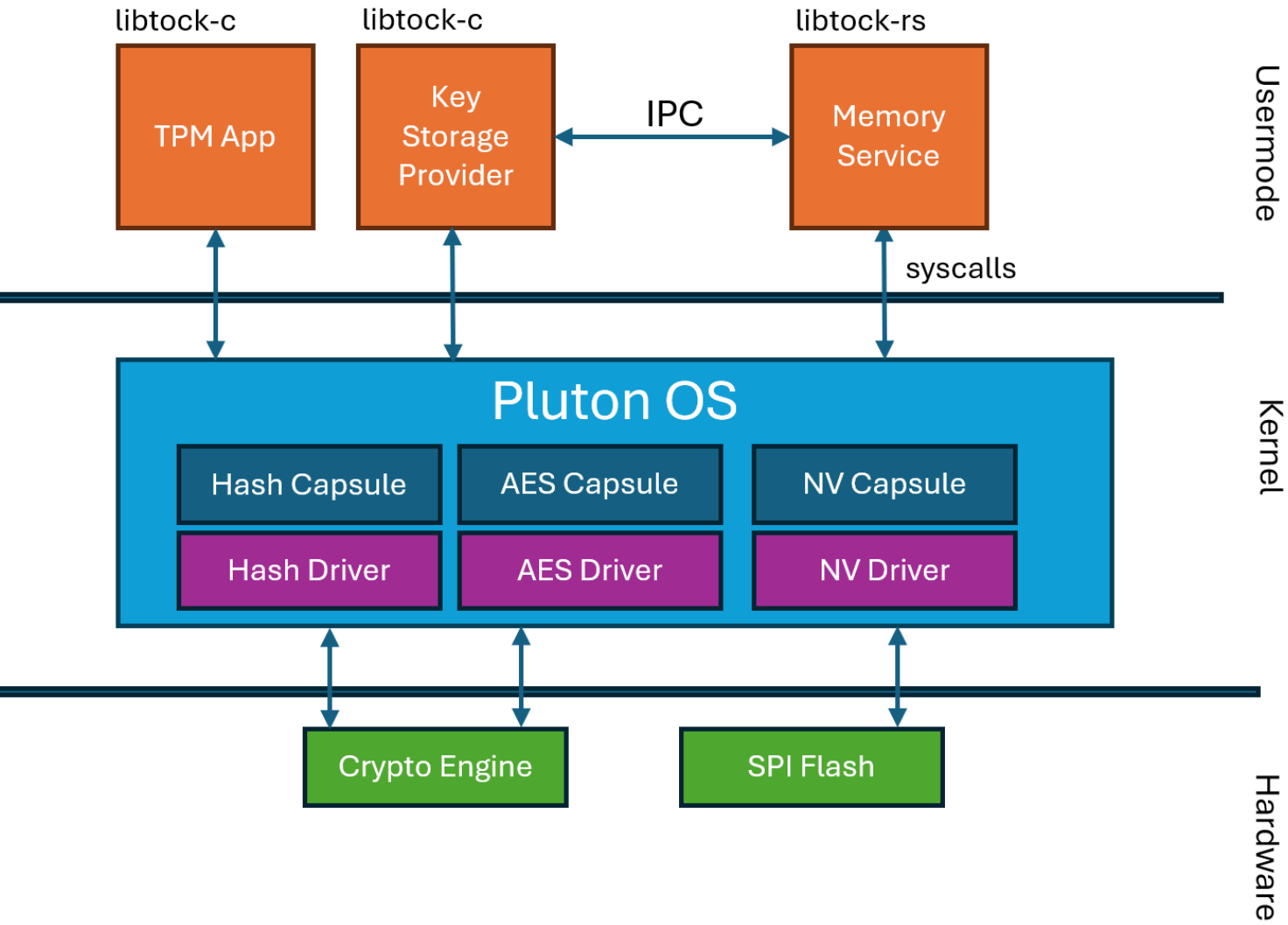

libtock-c
libtock-c
libtock-rs
TPM App
Key Storage Provider
IPC
Memory Service
Usermode
syscalls
Pluton OS
Hash Capsule
AES Capsule
NV Capsule
Hash Driver
AES Driver
NV Driver
Kernel
Crypto Engine
SPI Flash
Hardware



Business logic:

- TPM 2.0 App
- NV Memory Service
- Future products

Often not originally designed for Pluton OS

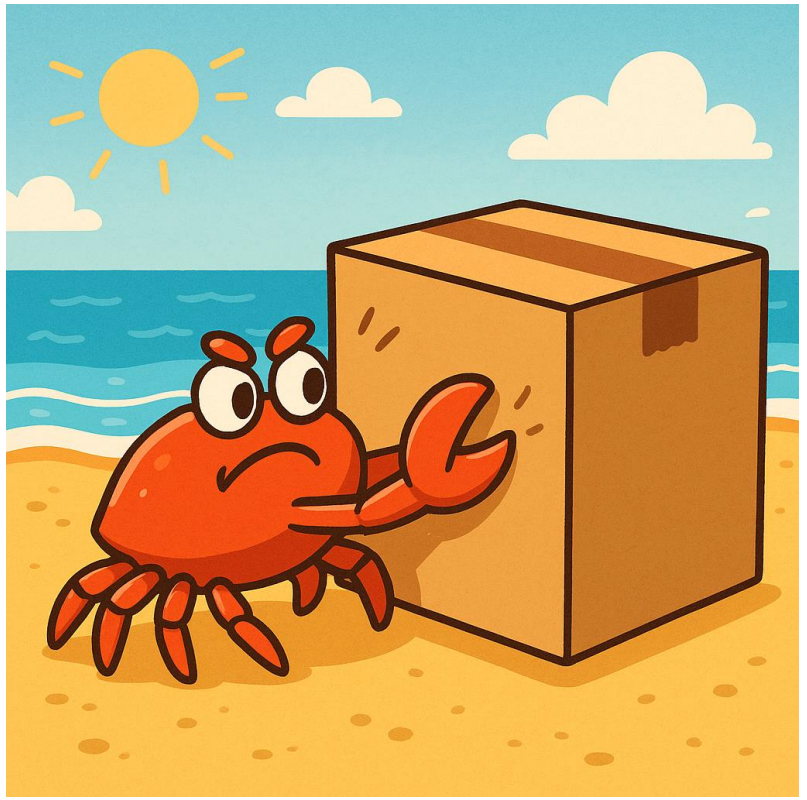


Capsules / Drivers:

- SHA Hash
- SHA HMAC
- KDF
- AES
- RSA
- ECC
- HW Keys
- DRAM Bridge
- Nor Flash
- RPMC
- SVN

Agenda

- Pluton Constraints
- Review of Tock OS's Data Movement Pattern
- Proposing Push/Pull
- Case Study: AES
- Open Questions / Future Work



Crab determined to push payload

Setting the Stage (1)

This is a crypto heavy project – we process a lot of data

- RSA Key Generation and Signing – 3K bit BigNums
- Hashing and Encrypting Entire TPM State – 16Kb
- Reading from NV - 4Kb
- Funneling x509 certificates (UART) – 5Kb

Setting the Stage (2)

Extremely limited memory – we choose space over latency

- Pluton platforms have roughly 500 KB of memory (RAM + Flash)
- Kernel and drivers consumes ~150KB
- TPM consumes ~300KB

- Leaves us about 50KB for other payloads…

Crab unable to fit in home

Setting the Stage (3)

Multiple kernel clients for a single (usually crypto) driver

Ex: Hash driver is mutexed and shared by:

- The hash syscall handler capsule
- Storage Capsule (encrypt + hash)
- KDF Capsule
- Self Test Capsule

Setting the Stage (4)

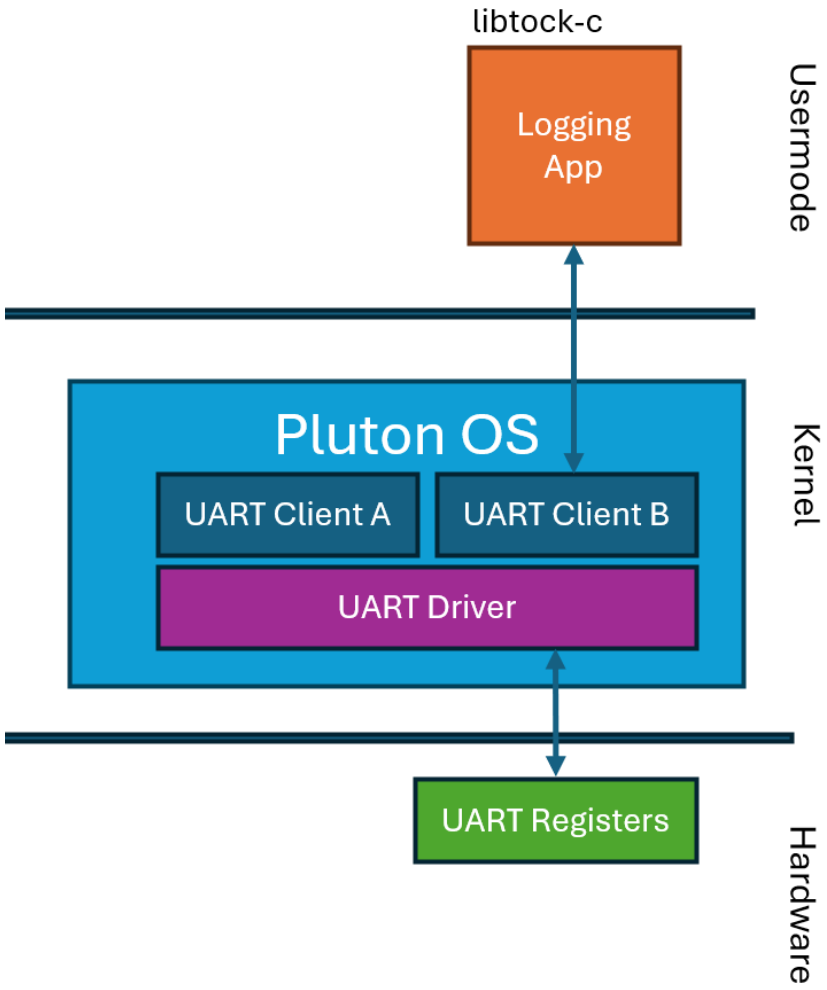
Very different interfaces across platforms due to physical ASIC differences

Need to design for:

- Chip with a 4KB mailbox interface
- Chip with a 256-byte buffer interface
- Chip that requires DMA

Crabs celebrating their differences

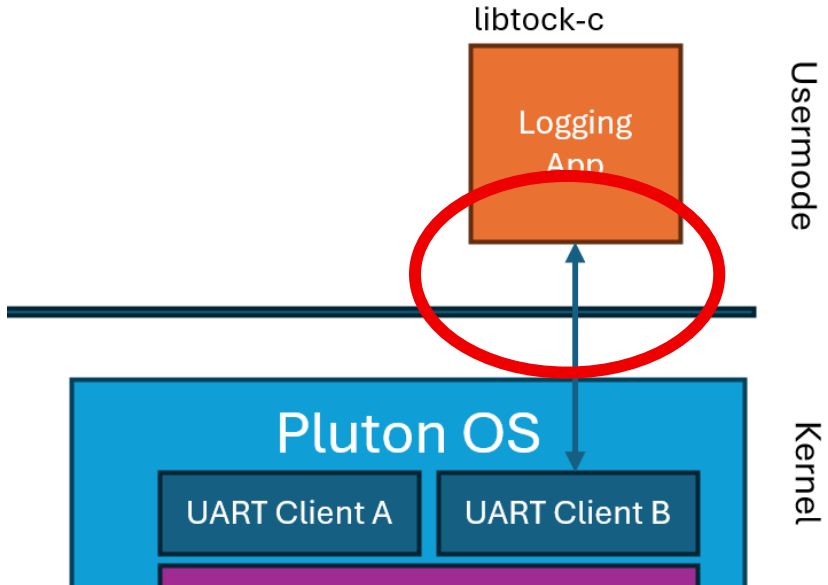
Our Simplified UART Scenario



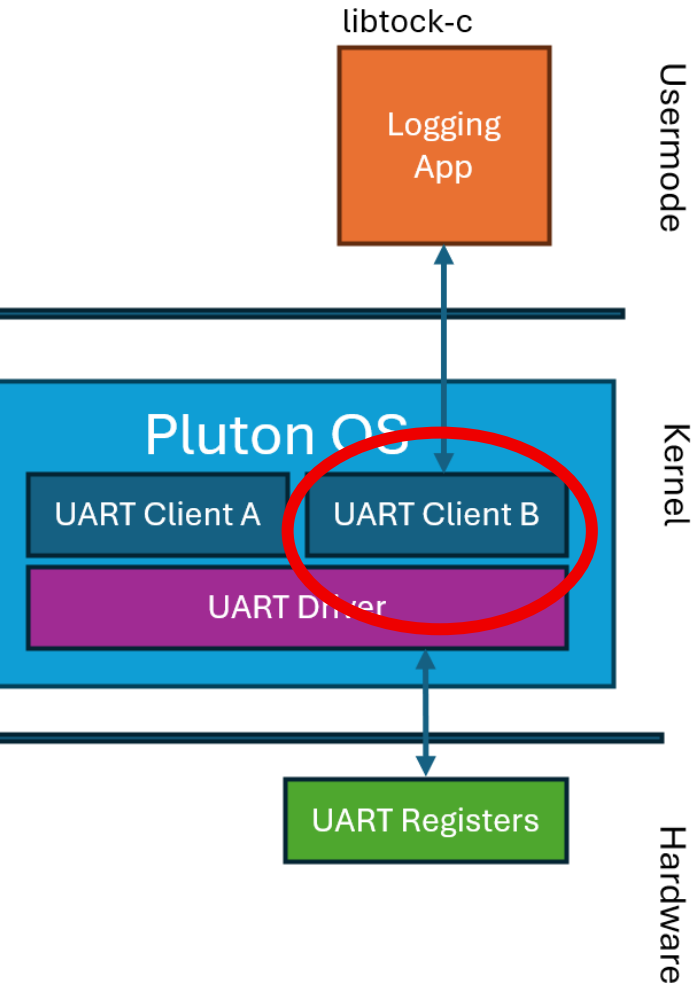
- Some logging app that wants to write to UART
- Two UART clients
  - Syscall handler
  - Some other kernel component
- One UART driver that handles HW interaction

Share buffers and send command

```
allow_ro_return_t aro = allow_readonly(UART_DRIVER_NUM, ALLOW_TX_BUFFER_ID,
                                       (const void *)tx, (uint32_t)len);

command_return_t cr = command(UART_DRIVER_NUM, CMD_TRANSMIT, (int)len, 0);
```

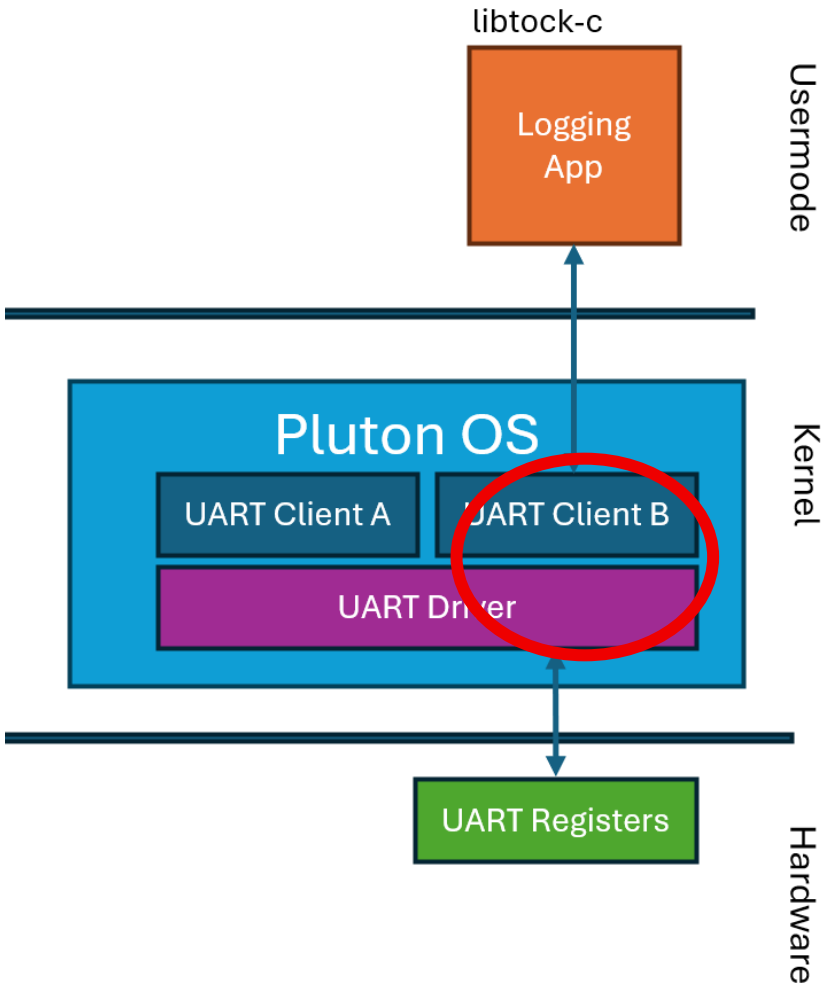


Capsule copies app data into static buffer



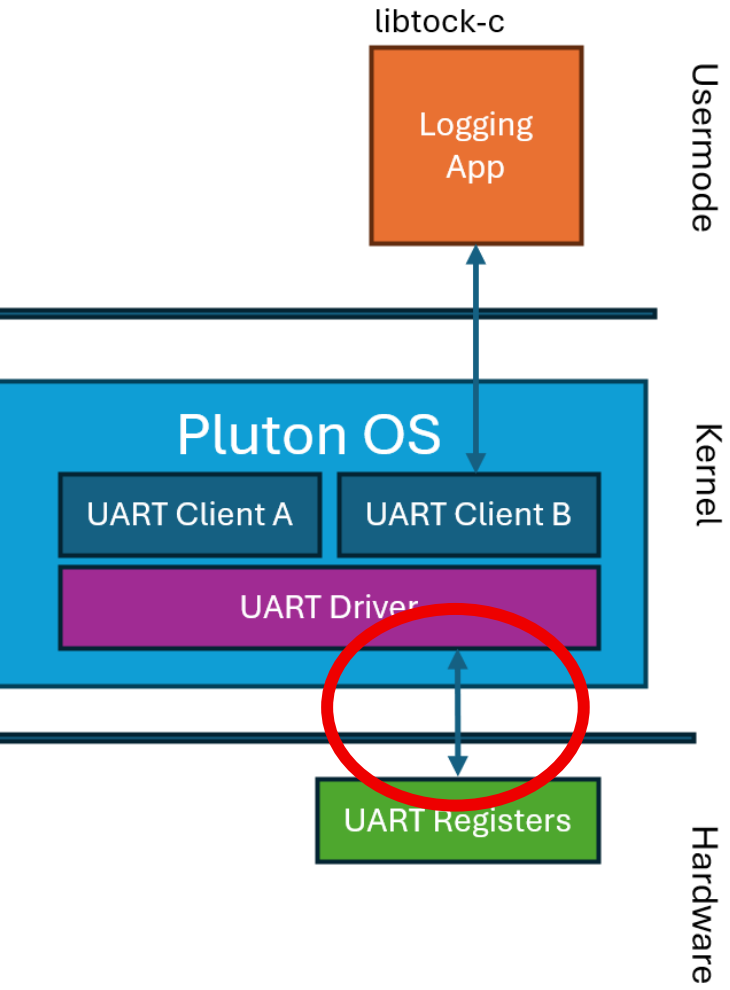

```rust
match self.state.replace(State::Processing) {
    // 1) Get static buffer from capsule state
    State::PullTx { tx_buffer } => {
        self.apps
            .enter(pid, |_app, kernel_data| {

                // 2) Get app's read-only buffer
                let buffer_src: ReadOnlyProcessBuffer =
                    kernel_data.get_readonly_processbuffer(ro_allow::TX)?;

                buffer_src.enter(|src| {
                    // Check if the static buffer is large enough
                    // for the data to be copied
                    if tx_buffer.len() < src.len() {
                        return Err(ErrorCode::SIZE);
                    }

                    // 3) Copy data from app's buffer to static buffer
                    src.copy_to_slice(&mut tx_buffer[..src.len()]);

                    // Return number of bytes copied
                    Ok(src.len())
                })
            })?;
```

Capsule invokes UART driver and passes ownership of the static TX buffer



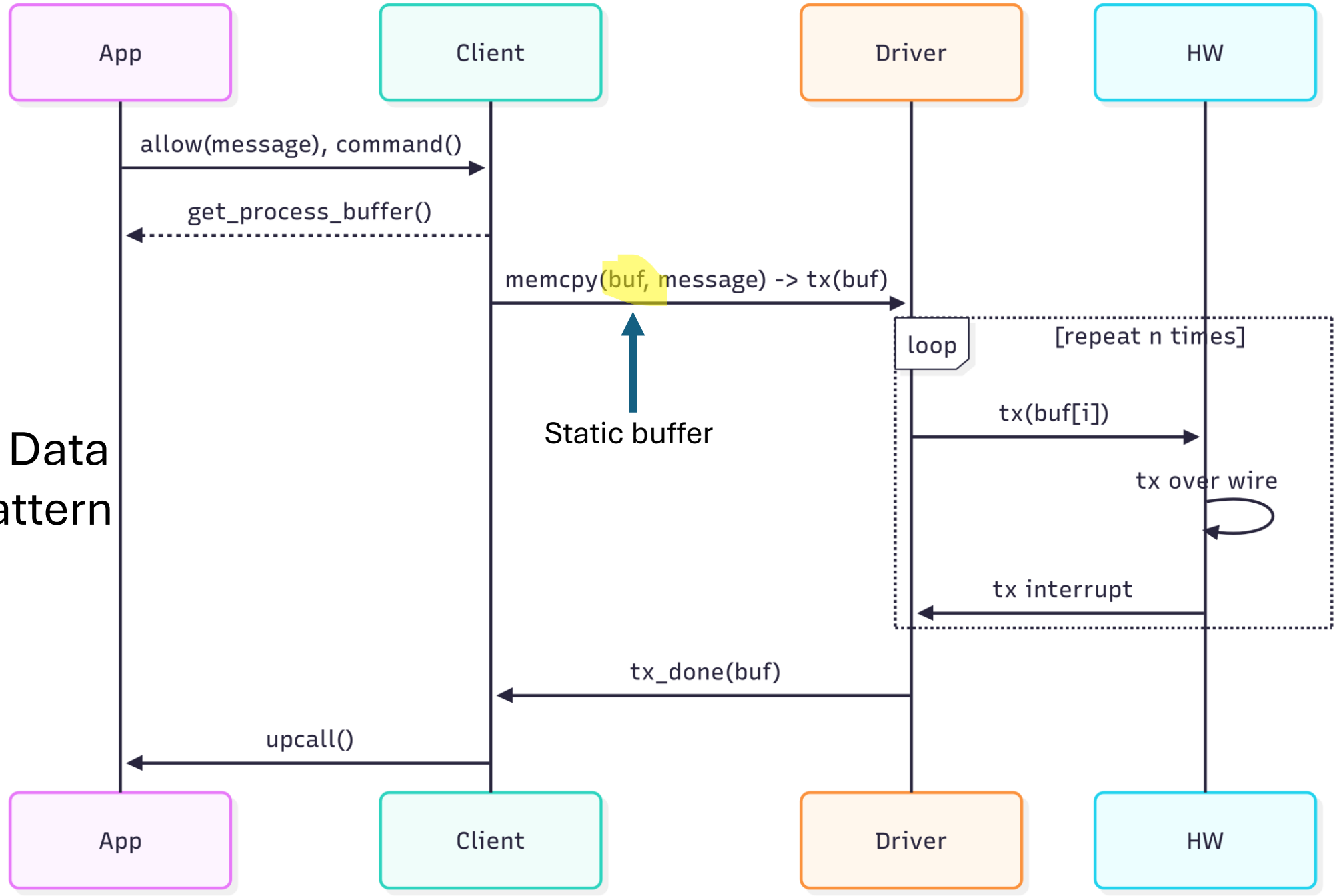

```
uart_driver.transmit(tx_buffer, tx_length)
```

Driver writes data to register byte at a time, N times



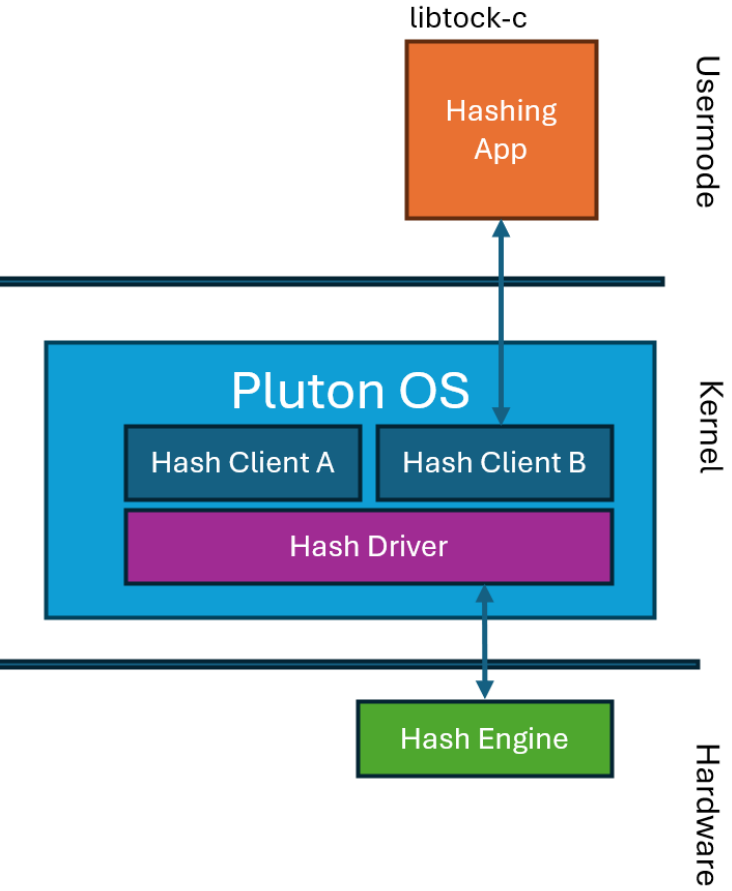

```
pub fn handle_tx_interrupt() {
    match self.state.replace(State::Processing) {
        State::WriteTx { tx_buffer, cursor, msg_len } => {

            if (cursor == msg_len) {
                self.state.replace(State::Idle);
                self.client.transmit_done(tx_buffer);
                return;
            }

            self.UART_REG.write(tx_buffer[cursor]);

            self.state.replace(
               State::WriteTx{ tx_buffer, cursor + 1, msg_len });
        }
    }
}
```

Current Tock Data Movement Pattern



What do we not like about this?

Poor Hardware Utilization

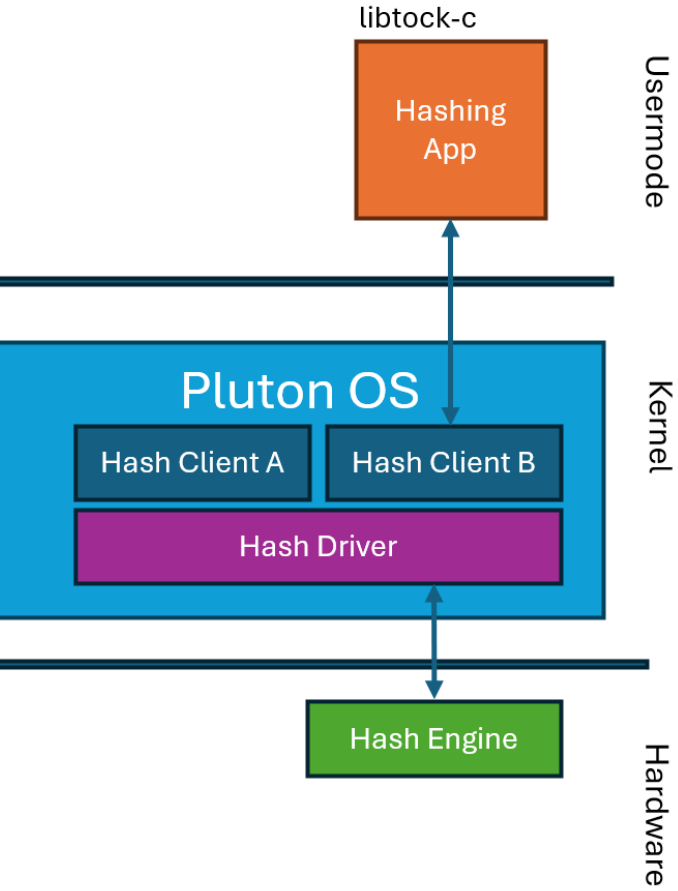


Ideally, each client's static buffer would be:

```
MIN(Data to operate on, amount HW can process)
```

- If we reduce operating data amount
  - ➢ More UM-Kernel roundtrips
- If we reduce HW processing amount
  - ➢ Poor performance, complex states

Static Buffer Pains



- Static buffers directly contribute to our base RAM consumption
  - **Multiplied per client!**
- Require being mocked for unit testing
- Increase memory fragmentation with alignment requirements
- Poor ergonomics…

Ergonomics – Complex States + Invariants

Consider our Storage Capsule that performs HMAC, SHA and AES:

```
enum State {
    Idle {
        plaintext: &'static mut [u8],
        ciphertext: &'static mut [u8],
        aes_key: &'static mut [u8],
        aes_iv: &'static mut [u8],
        hash_digest: &'static mut [u8],
        hash_ctx: &'static mut [u8],
        hash_partial: &'static mut [u8],
        hmac_digest: &'static mut [u8],
    },
```

```
    AesInProgress {
        hash_digest: &'static mut [u8],
        hash_ctx: &'static mut [u8],
        hash_partial: &'static mut [u8],
        hmac_digest: &'static mut [u8],
    },

    HashInProgress {
        ciphertext: &'static mut [u8],
        aes_key: &'static mut [u8],
        aes_iv: &'static mut [u8],
        hmac_digest: &'static mut [u8],
    },
```

Ergonomics – Complex States + Invariants

Consider our Storage Capsule that performs HMAC, SHA and AES:

```
pub struct StorageCapsule {
    apps: Grant<...>,
    state: RefCell<...>,
    plaintext: TakeCell<&'static mut [u8]>,
    ciphertext: TakeCell<&'static mut [u8]>,
    aes_key: TakeCell<&'static mut [u8]>,
    aes_iv: TakeCell<&'static mut [u8]>,
    hash_digest: TakeCell<&'static mut [u8]>,
    hash_ctx: TakeCell<&'static mut [u8]>,
    hash_partial: TakeCell<&'static mut [u8]>,
    hmac_digest: TakeCell<&'static mut [u8]>,
}
```

Ergonomics – Tracking Error States

Rule 4: Return Passed Buffers in Error Results

https://book.tockos.org/trd/trd3-hil-design

```
// Anti-pattern: caller cannot regain buf on an error
 fn send(&self, buf: &'static mut [u8]) -> Result<(), ErrorCode>;
```

```
fn send(&self, buf: &'static mut [u8]) -> Result<(), (ErrorCode, &'static mut [u8])>;
```

Proposal: Push Pull Pattern

General HIL Change

Current

```
trait Driver {
    fn operation(&self, input: &'static [u8], output: &'static mut [u8]) ->
                        Result<(), (ErrorCode, &'static [u8], &'static mut [u8])>;
}
trait Client {
    fn operation_done(&self, result: Result<(), ErrorCode>,
        input: &'static [u8], output: &'static mut [u8]);
}
```

General HIL Change

Current

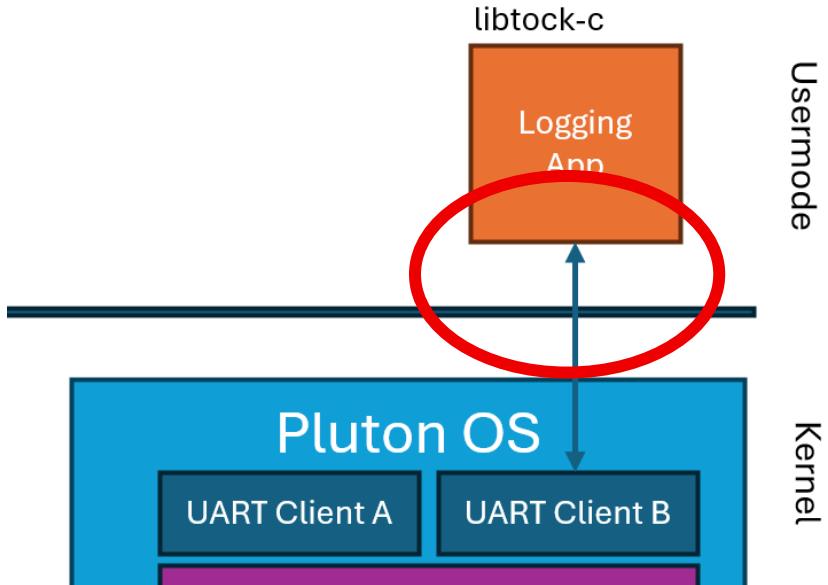
```
trait Driver {
    fn operation(&self, input: &'static [u8], output: &'static mut [u8]) ->
                        Result<(), (ErrorCode, &'static [u8], &'static mut [u8])>;
}
trait Client {
    fn operation_done(&self, result: Result<(), ErrorCode>,
        input: &'static [u8], output: &'static mut [u8]);
}
```

Lazy Pattern

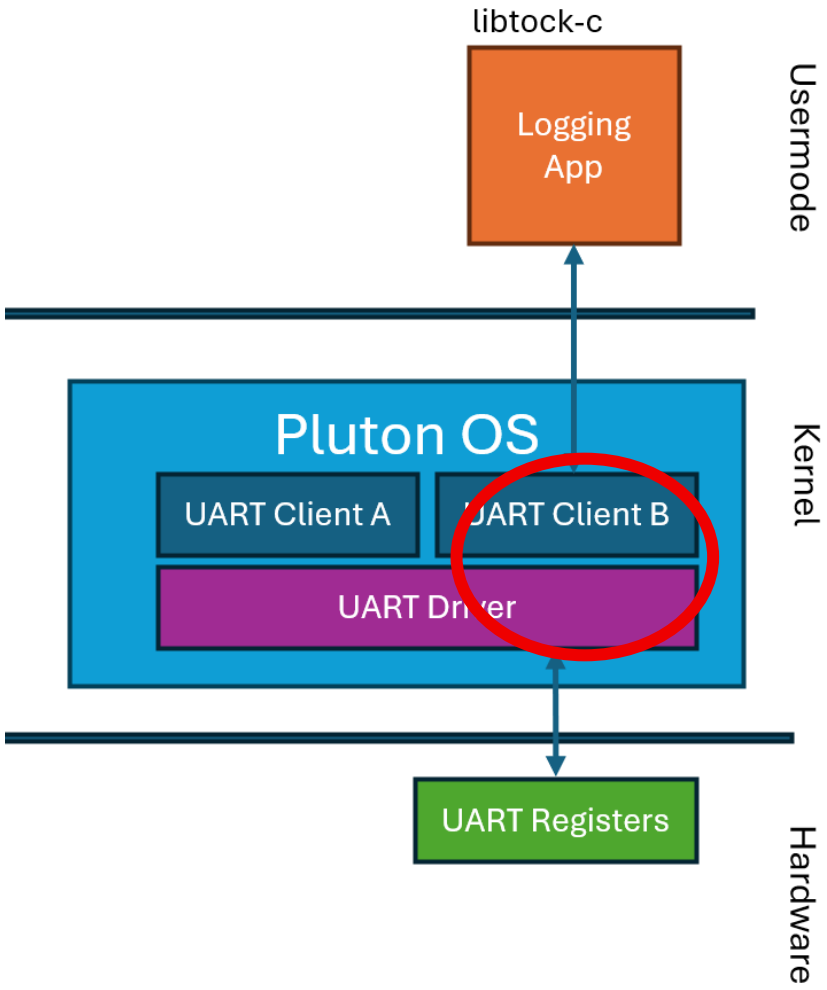
```
trait Driver {
    fn operation(&self) -> Result<(), ErrorCode>;
}

trait Client {
    fn pull_input(&self, buffer: &mut [u8], cursor: usize) -> Result<usize, ErrorCode>;
    fn push_output(&self, buffer: &[u8], cursor: usize) -> Result<usize, ErrorCode>;
    fn operation_done(&self, result: Result<(), ErrorCode>);
}
```

Share buffers and send command

```
allow_ro_return_t aro = allow_readonly(UART_DRIVER_NUM, ALLOW_TX_BUFFER_ID,
                                       (const void *)tx, (uint32_t)len);

command_return_t cr = command(UART_DRIVER_NUM, CMD_TRANSMIT, (int)len, 0);
```



Capsule invokes UART driver and passes ownership of the static TX buffer

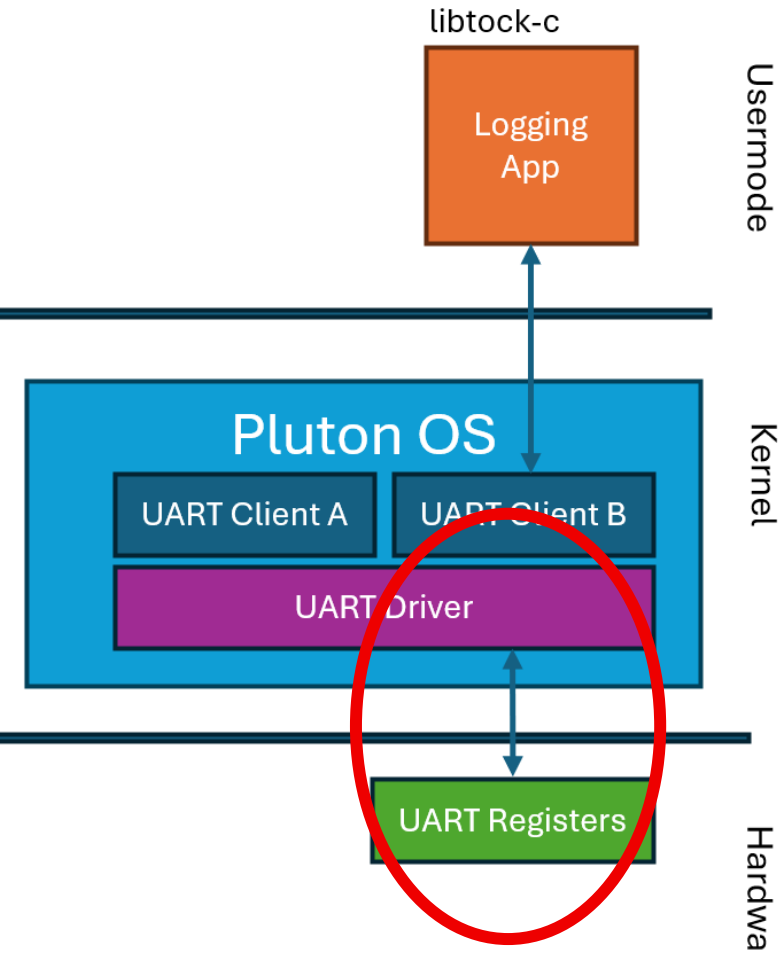



```
uart_driver.transmit(tx_length)
```

Client exposes a way to pull data

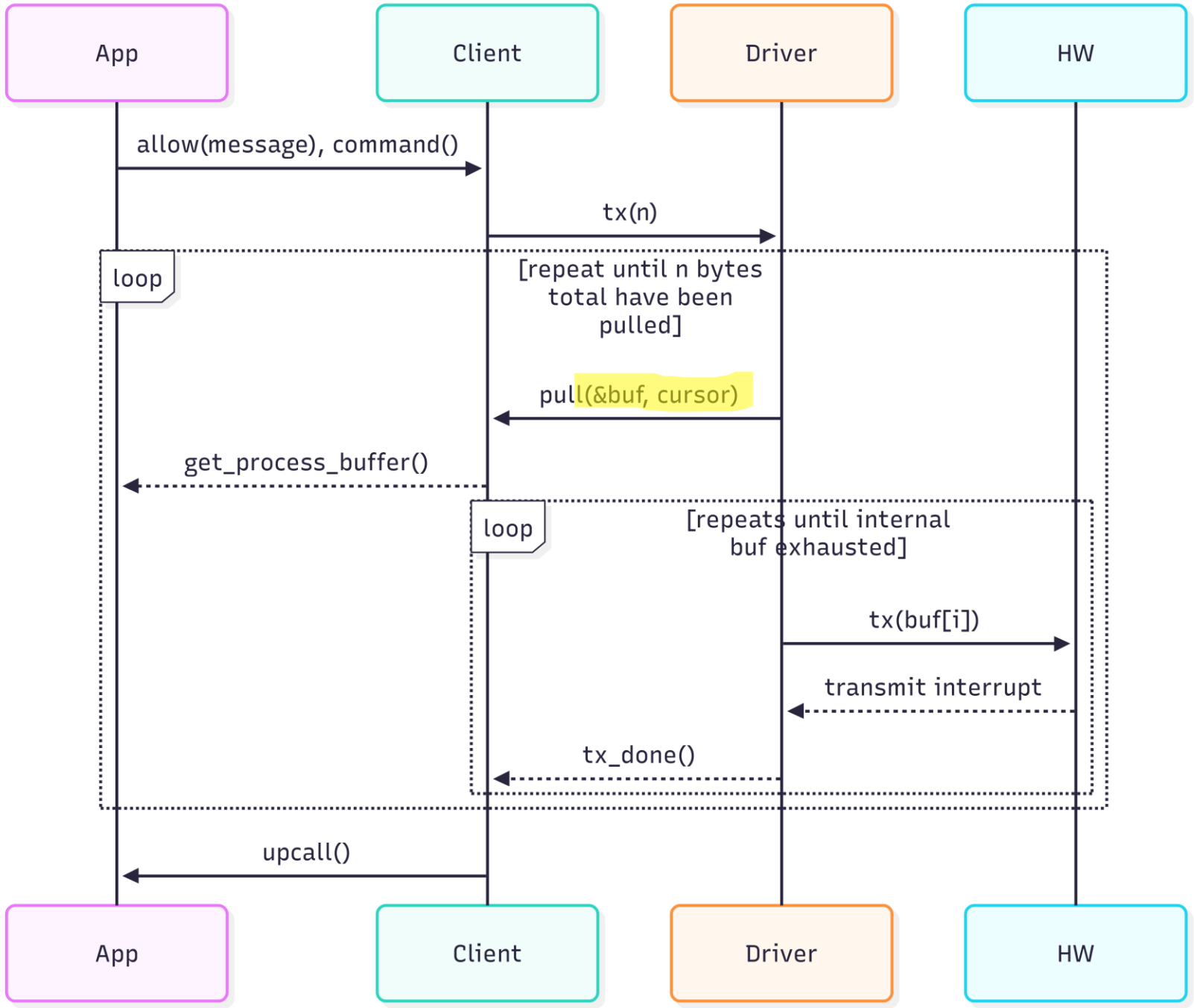
```rust
fn fill_input(&self, data: &mut [u8], offset: usize) -> Result<usize, ErrorCode> {
    // Check state machine to determine the PID of the app containing the source data
    let pid = match *self.state.borrow() {
        State::Tx { pid, .. } => pid,
        _ => return Err(ErrorCode::INVAL),
    };
    let mut data_len = 0;
    // Acquire data and context from the app
    self.apps.enter(pid, |_app, kernel_data| {
        // Grab the data from the app
        let data_src =
            kernel_data.get_readonly_processbuffer(ro_allow::TX)?;
        data_src.enter(|src| {
            data_len = core::cmp::min(data.len(), src.len() - offset);
            src[offset..offset + data_len].copy_to_slice(&mut data[..data_len]);
        })
    })??;
    Ok(data_len)
}
```

Drivers populate stack variable and write this to HW




```rust
pub fn handle_tx_interrupt() {
    match self.state.replace(State::Processing) {
        State::WriteTx { cursor, msg_len } => {

            // Check if the transmission is complete
            if (cursor == msg_len) {
                self.state.replace(State::Idle);
                self.client.transmit_done();
                return;
            }

            // Create a stack buffer to hold the TX byte
            let mut tx_buffer: [u8; 1] = [0; 1];

            // Pull the input byte from the client
            if let Err(e) =
                self.client.pull_input(&mut tx_buffer, cursor) {
                // Handle error
            }

            // Write the TX byte to the UART register
            self.UART_REG.write(tx_buffer[0]);

            // Update the state with the new cursor position
            self.state.replace(
                State::WriteTx { cursor: cursor + 1, msg_len });
        }
    }
}
```

Proposed Lazy Data Movement Pattern



Analysis

Instead of copying data into a client owned static buffer, we let drivers pull from grant / source data into their stack.

1. We replace large static buffers with stack buffers
   A. Greatly improved baseline RAM usage
   B. Ergonomics – Much simpler state machines, fewer TakeCells, fewer dropped buffers bugs
   C. Easier to unit test function inputs than mocking static buffers
2. No longer pay for static buffers per client
   A. Important given how Pluton shares crypto drivers across many clients

Analysis

Hardware drivers decide how much data they want to pull in at once and manage looping and cursors themselves.

1. Hardware drivers operate as efficiently as they can
    1. Eg: The 4KB mailbox system is no longer limited by the clients holding 512-byte buffers
2. Complexity is handled by the drivers, not clients

Discussion / Callouts

Case Study: AES Modes

Mode	IV	Tag	Counter	AAD
ECB	❌	❌	❌	❌
CTR	✅	❌	✅	❌
CBC	✅	❌	❌	❌
GCM	✅	✅	✅	✅

Case Study: AES Modes

```rust
pub trait Aes<K> {
    /// Perform AES operation.
    /// Implementations are expected to call the following:
    /// - [`AesClient::pull_input()`] to get the input data
    /// - [`AesClient::pull_key()`] to get the key
    /// - [`AesClient::push_output()`] to push the output data
    /// - [`AesClient::aes_done()`] to notify the client of the operation completion
    /// Implementations must also call the following if the mode requires it:
    /// | Mode        | Function Calls                                                                    |
    /// |-------------|-----------------------------------------------------------------------------------|
    /// | CBC         | [`AesClient::pull_iv()`], [`AesClient::push_iv()`]                                |
    /// | CTR         | [`AesClient::pull_iv()`], [`AesClient::pull_ctr()`], [`AesClient::push_iv()`], ... |
    /// | GCM         | [`AesClient::pull_iv()`], [`AesClient::pull_ctr()`], [`AesClient::pull_tag()`] ... |
    ///
    /// Notes:
    /// - For GCM Decryption, if the tag is not valid, the implementation will return [`ErrorCode::INVAL`]
    fn aes(&self, aes_mode: AesMode, aes_operation: AesOperation) -> Result<(), ErrorCode>;
```

Multiple uses of a driver within a component

This requires considering the current state within the push/pull callbacks.

```
fn pull_buffer(&self, buffer: &mut [u8], ro_allow_num: usize) -> Result<usize, ErrorCode> {
    let pid = match *self.state.borrow() {
        State::AesDone(pid) => pid,
        State::AesKHDone { pid, .. } => pid,
        _ => return Err(ErrorCode::RESERVE),
    };

    self.apps.enter(pid, |_app, kernel_data| {
        let buffer_src = kernel_data.get_readonly_processbuffer(ro_allow_num)?;
        buffer_src.enter(|src| {
            if buffer.len() < src.len() {
                    return Err(ErrorCode::SIZE);
            }
            src.copy_to_slice(&mut buffer[..src.len()]);
    ...
```

We don’t use cursors for small buffers

```
fn pull_ctr(&self, ctr: &mut [u8]) -> Result<usize, ErrorCode> {
    self.pull_buffer(ctr, ro_allow::CTR)
}

fn pull_iv(&self, iv: &mut [u8]) -> Result<usize, ErrorCode> {
    self.pull_buffer(iv, ro_allow::IV)
}

fn pull_key(&self, key: &mut [u8]) -> Result<AesKeySize, ErrorCode> {
    let size = self.pull_buffer(key, ro_allow::KEY)?;
    AesKeySize::try_from(size as u32).map_err(|_| ErrorCode::INVAL)
}
```

Callout: Naming

- We chose push/pull for ease of cognitive load

- Other contenders:
  - Read/write input/output
  - Transmit/receive
  - Domain specific names

Upstreaming / Next Steps

If we like the design:

- Can start up streaming our capsules
- Formalize design documentation
- Refactor existing capsules? Add “lazy streaming” alternative?

- SHA Hash
- SHA HMAC
- KDF
- AES
- ModExp
- ECC
- HW Keys
- Nor Flash
- RPMC
- SVN

Questions?

Contact: hmiyaziwala@microsoft.com

Notes from Tock World

- DMA
  - Consider an abstraction above each capsule to pick between static buffers for DMA and callbacks for others
- Reentrency
  - Callbacks may cause re-entering a grant or other capsule reentrency issues
  - Deep call stacks if we don’t DC
- Complexity ownership
  - Pluton aims to keep complexity within its drivers, may be against Tock’s general goals