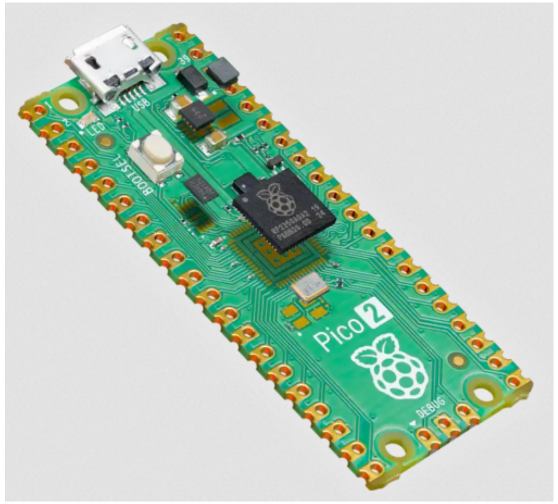
Tock Goes *Multicore*

Gongqi Huang, Princeton 🐯

Multicore MCUs are Useful

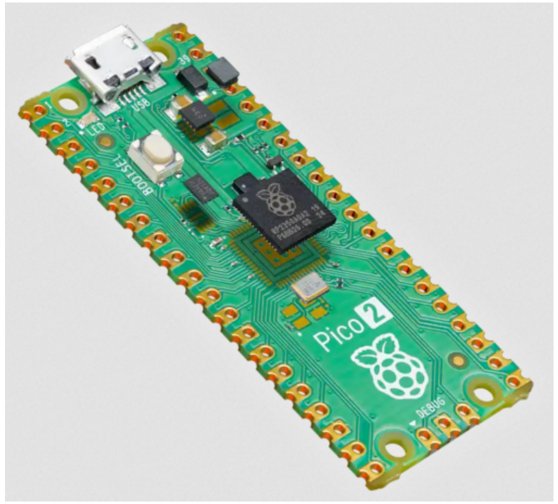


Raspberry Pi Pico 2[1]

[1]https://www.raspberrypi.com/products/raspberry-pi-pico-2

Multicore MCUs are Useful

- Dedicating a CPU core for a specific task

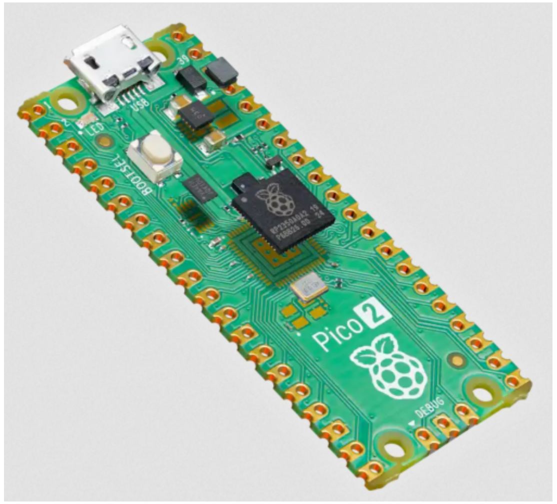


Raspberry Pi Pico 2[1]

[1]https://www.raspberrypi.com/products/raspberry-pi-pico-2

Multicore MCUs are Useful

- Dedicating a CPU core for a specific task
  - Performance isolation
    - Radio/BLE stack

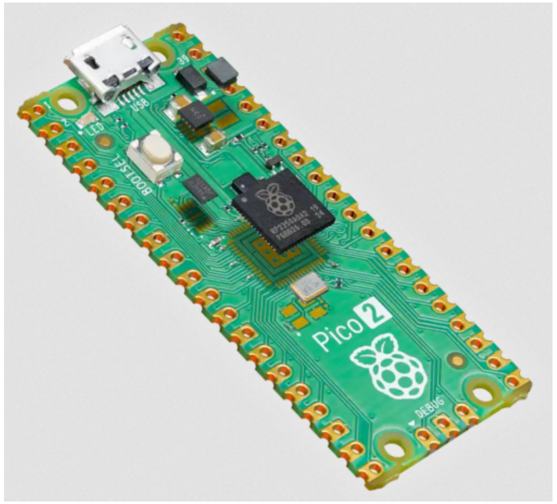


Raspberry Pi Pico 2[1]

[1] https://www.raspberrypi.com/products/raspberry-pi-pico-2

Multicore MCUs are Useful

- Dedicating a CPU core for a specific task
  - Performance isolation
    - Radio/BLE stack
  - Security
    - Close μ-arch side channels

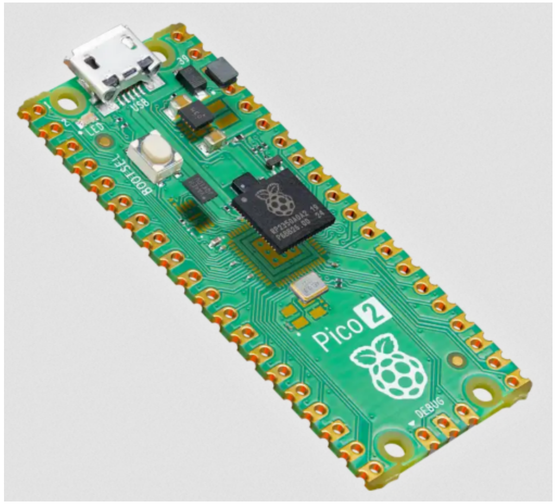


Raspberry Pi Pico 2[1]

[1]https://www.raspberrypi.com/products/raspberry-pi-pico-2

Multicore MCUs are Useful

- Dedicating a CPU core for a specific task
  - Performance isolation
    - Radio/BLE stack
  - Security
    - Close µ-arch side channels
- Utilizing multiple CPU cores for a specific task

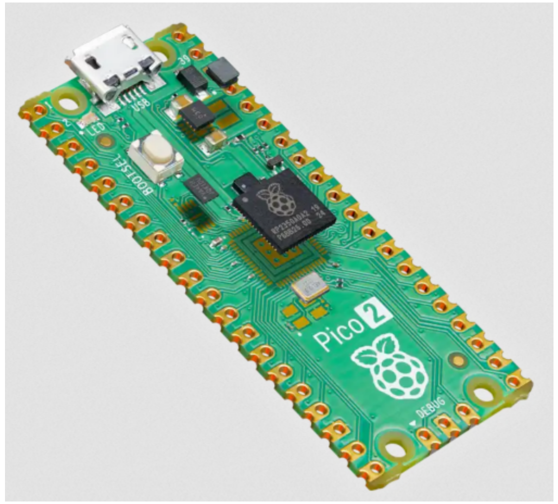


Raspberry Pi Pico 2[1]

[1]https://www.raspberrypi.com/products/raspberry-pi-pico-2

Multicore MCUs are Useful

- Dedicating a CPU core for a specific task
  - Performance isolation
    - Radio/BLE stack
  - Security
    - Close µ-arch side channels
- Utilizing multiple CPU cores for a specific task
  - Performance boost with hardware parallelism



Raspberry Pi Pico 2[1]

[1]https://www.raspberrypi.com/products/raspberry-pi-pico-2

Multicore MCUs Have Various Architectures

Multicore MCUs Have Various Architectures

- Homogeneous (RP2350) or heterogenous (CC26x2) CPU cores
  - Different ISAs

Multicore MCUs Have Various Architectures

- Homogeneous (RP2350) or heterogenous (CC26x2) CPU cores
  - Different ISAs
- Sharing all memory (RP2350) or part of memory (nRF5340)
  - A radio core can have a *private* memory region

Multicore MCUs Have Various Architectures

- Homogeneous (RP2350) or heterogenous (CC26x2) CPU cores
  - Different ISAs
- Sharing all memory (RP2350) or part of memory (nRF5340)
  - A radio core can have a *private* memory region
- Sharing all (RP2350) or part of peripherals (nRF5340)

Tock is Not Multi-Core Ready

Tock is Not Multi-Core Ready

- Tock is a single-core OS

Tock is Not Multi-Core Ready

- Tock is a single-core OS
- The single-core assumption manifests in many of Tock’s design

Tock is Not Multi-Core Ready

- Tock is a single-core OS
- The single-core assumption manifests in many of Tock's design
  - Use of interior mutability
  - Single-threaded asynchronous drivers
  - …

Goal #1: Run Tock on Multi-Core Platforms

- Utilize other CPU cores

Goal #1: Run Tock on Multi-Core Platforms

- Utilize other CPU cores
  - Performance

Goal #1: Run Tock on Multi-Core Platforms

- Utilize other CPU cores
  - Performance
  - Security
    - Capsules are fully trusted to maintain liveness of the system
    - Not necessary in a multi-core setting

Goal #2: Retain Tock's Existing Benefits

Goal #2: Retain Tock's Existing Benefits

- Avoiding deadlocks and contention
  - No mutex for synchronization

Goal #2: Retain Tock's Existing Benefits

- Avoiding deadlocks and contention
  - No mutex for synchronization
- Predictable resource utilization
  - No dynamic allocation

Goal #2: Retain Tock's Existing Benefits

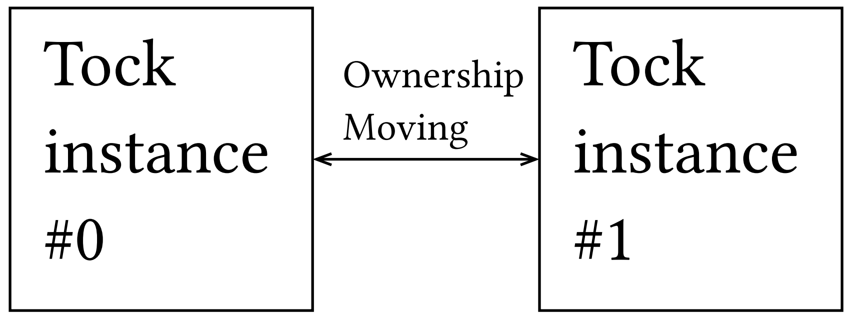
- Avoiding deadlocks and contention
  - No mutex for synchronization
- Predictable resource utilization
  - No dynamic allocation
- Isolation between process, capsules, and kernel
  - Maintain the soundness of Rust

Goal #2: Retain Tock's Existing Benefits

- Avoiding deadlocks and contention
  - No mutex for synchronization
- Predictable resource utilization
  - No dynamic allocation
- Isolation between process, capsules, and kernel
  - Maintain the soundness of Rust
- Easy-to-write device drivers
  - No concurrent state

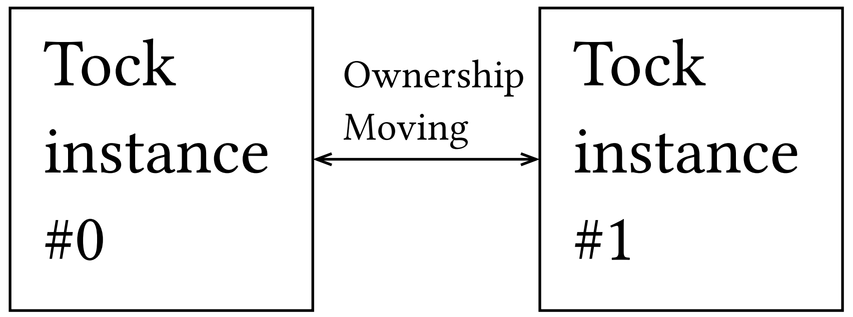
Multikernel Tock

Architecture



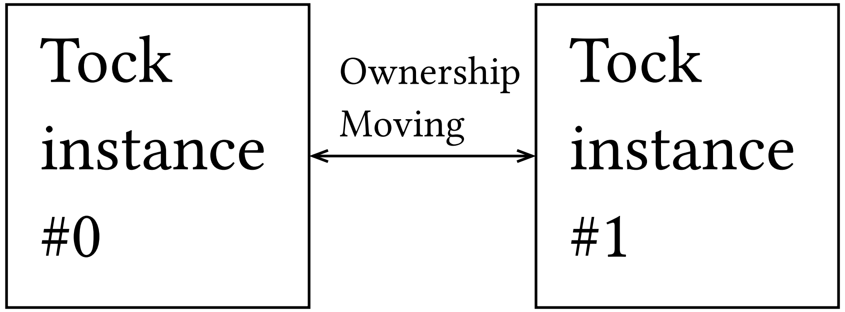
Architecture

- Each instance manages an *exclusive* set of peripherals
  - Retain *all* Tock's benefits



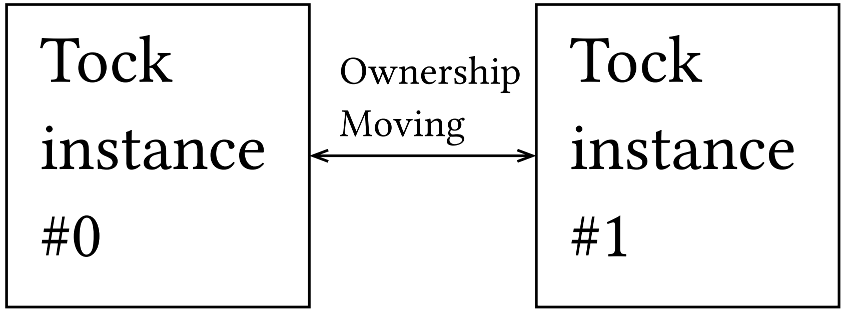
Architecture

- Each instance manages an *exclusive* set of peripherals
  - Retain *all* Tock's benefits
- Ownership moving enables



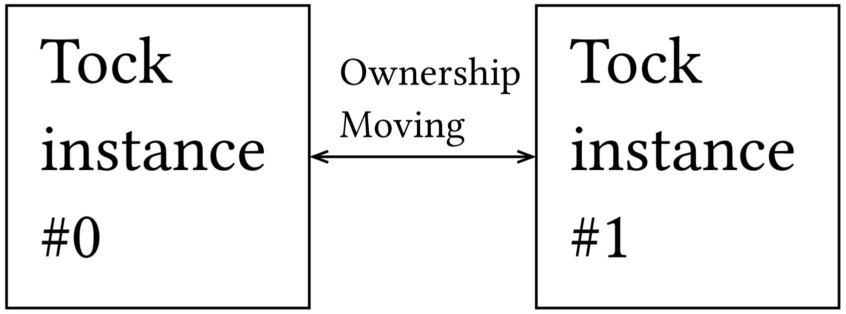
Architecture

- Each instance manages an *exclusive* set of peripherals
  - Retain *all* Tock's benefits
- Ownership moving enables
  - BYOB Communication
    - *Bring-Your-Own-Buffer*



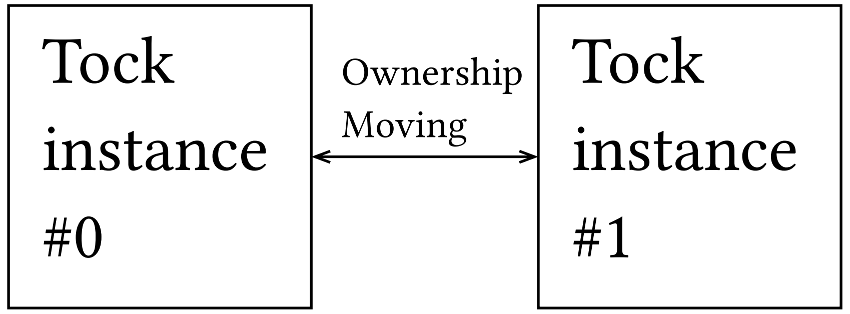
Architecture

- Each instance manages an *exclusive* set of peripherals
  - Retain *all* Tock's benefits
- Ownership moving enables
  - BYOB Communication
    - *Bring-Your-Own-Buffer*
  - Peripheral sharing w/ RPCs



Architecture

- Each instance manages an *exclusive* set of peripherals
  - Retain *all* Tock's benefits
- Ownership moving enables
  - BYOB Communication
    - *Bring-Your-Own-Buffer*
  - Peripheral sharing w/ RPCs
  - Raw peripheral sharing



Architecture

- Each instance manages an *exclusive* set of peripherals
  - Retain *a*
- Ownershi
  - BYOB C
    - *Bring*
  - Peripheral sharing w/ RPCs
  - Raw peripheral sharing

ck
stance

Runs on

- QEMU RISC-V Dual-Core Configuration
- Custom Dual-Core VexRiscv LiteX SoC

Teleporting Ownership with Care

```rust
pub trait Portal<'a, T: Send> {
…                                      …
    fn teleport(
        &self,
        traveler: &'static mut T,
    ) -> Result<(), (ErrorCode, &'static mut T)>; }
```

Teleporting Ownership with Care

1. Traveler must implement `Send`

```rust
pub trait Portal<'a, T: Send> {
                          …
    fn teleport(
        &self,
        traveler: &'static mut T,
    ) -> Result<(), (ErrorCode, &'static mut T)>; }
```

Teleporting Ownership with Care

1. Traveler must implement `Send`
2. `&mut T` is `Send` when `T: Send`

Rust

```rust
pub trait Portal<'a, T: Send> {
…
  fn teleport(
    &self,
    traveler: &'static mut T,
  ) -> Result<(), (ErrorCode, &'static mut T)>; }
```

Teleporting Ownership with Care

1. Traveler must implement `Send`
2. `&mut T` is `Send` when `T: Send`
3. A Tock instance is a `'static` “thread”

```rust
pub trait Portal<'a, T: Send> {
…                                    …
    fn teleport(
        &self,
        traveler: &'static mut T,
    ) -> Result<(), (ErrorCode, &'static mut T)>; }
```

Teleporting Ownership with Care

- Receiving the `traveler` back through callbacks

Teleporting Ownership with Care

- Receiving the `traveler` back through callbacks

```rust
pub trait Portal<'a, T: Send> {
    fn set_portal_client(&self, client: &'a dyn
    PortalClient<T>);
…                                          …
```

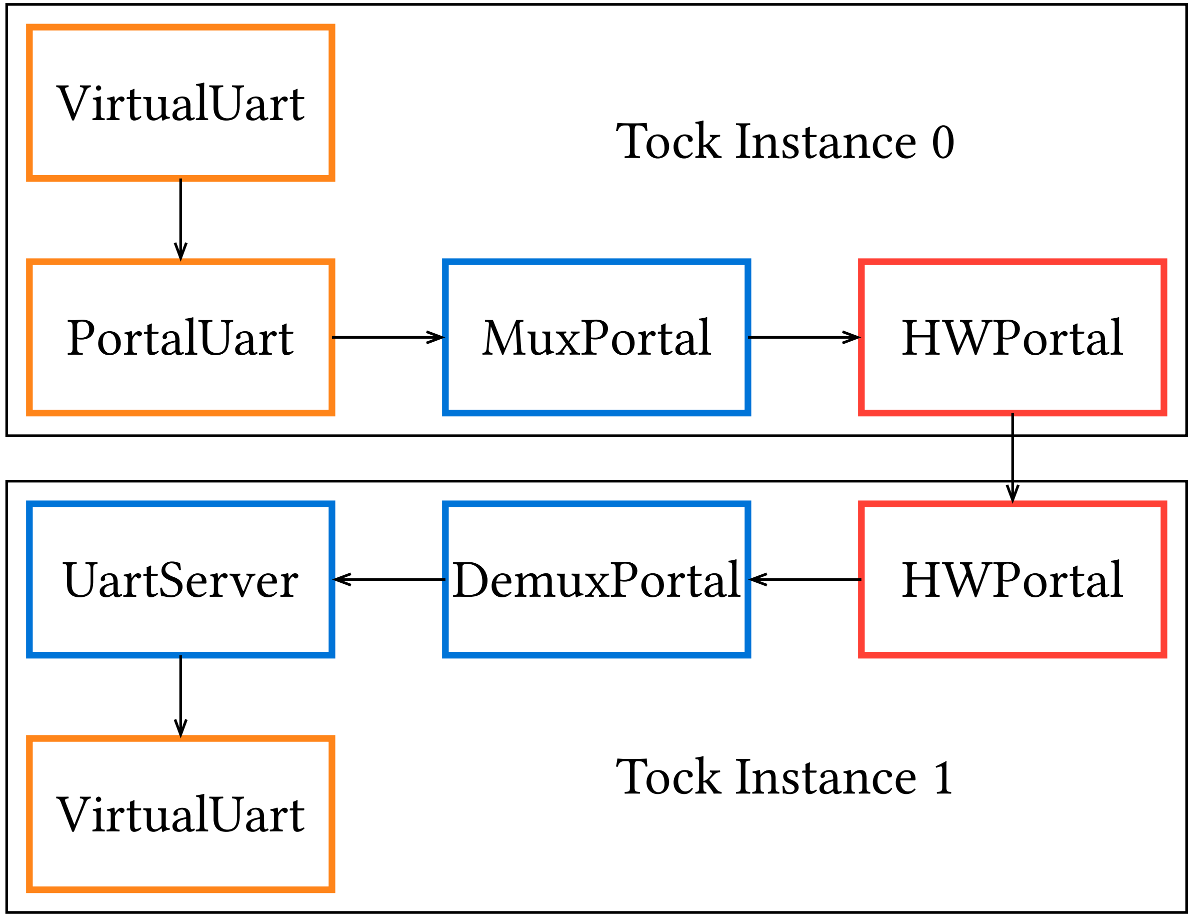
```rust
pub trait PortalClient<T: Send> {
    fn teleported(
        &self,
        traveler: &'static mut T,
        rcode: Result<(), ErrorCode>, ); }
```

Example: Sharing UART Through RPCs

🟧 `impl Uart`

🟦 `impl Portal`

🟥 `impl Portal` with `unsafe`



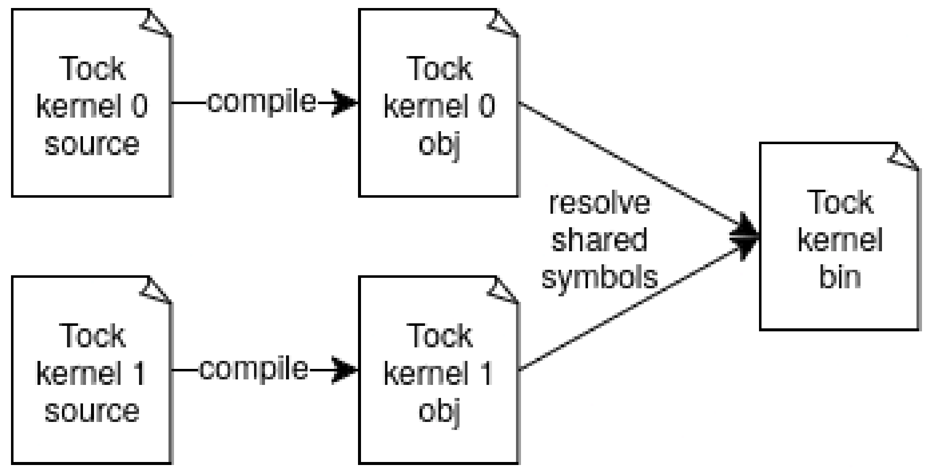
Sharing Raw Peripherals

- `Portal` permits transferring ownership of a raw device
  - E.g., Memory-mapped controller

Sharing Raw Peripherals

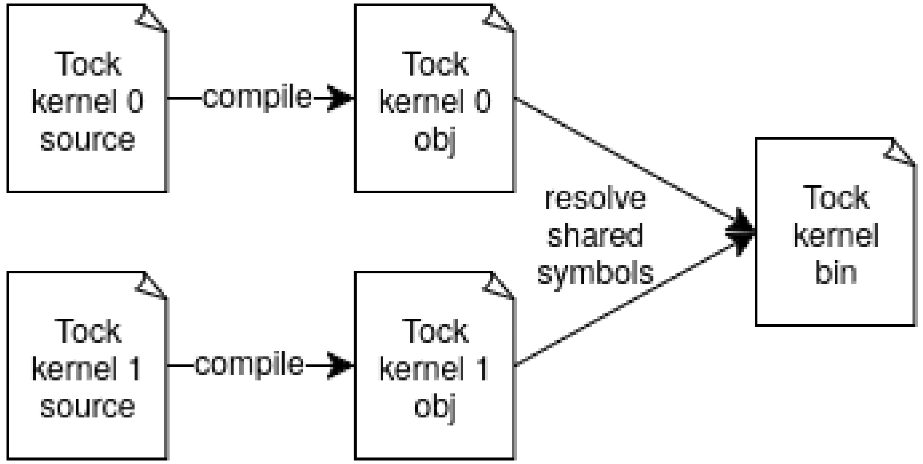
- `Portal` permits transferring ownership of a raw device
  - E.g., Memory-mapped controller
- Problem: *when* it is safe to transfer?
  - E.g., UART in the middle of a transmission
  - Currently unsupported :*(

Building Multikernel Tock



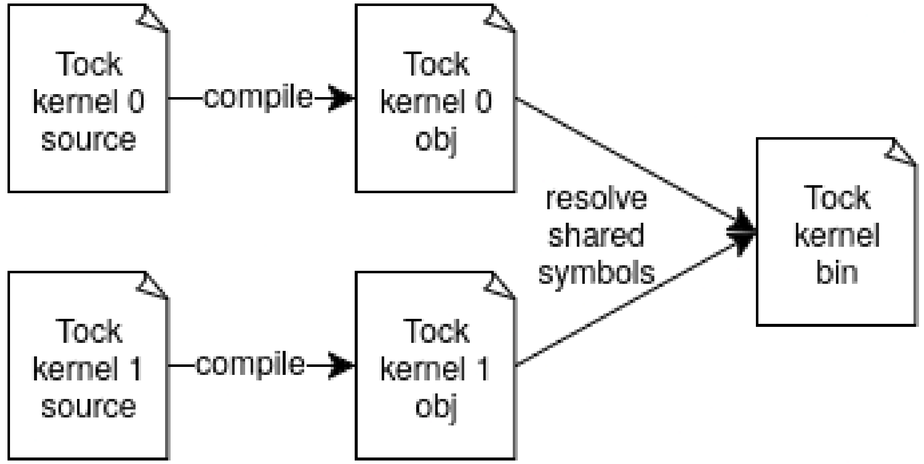
Building Multikernel Tock

- Build each kernel instance separately



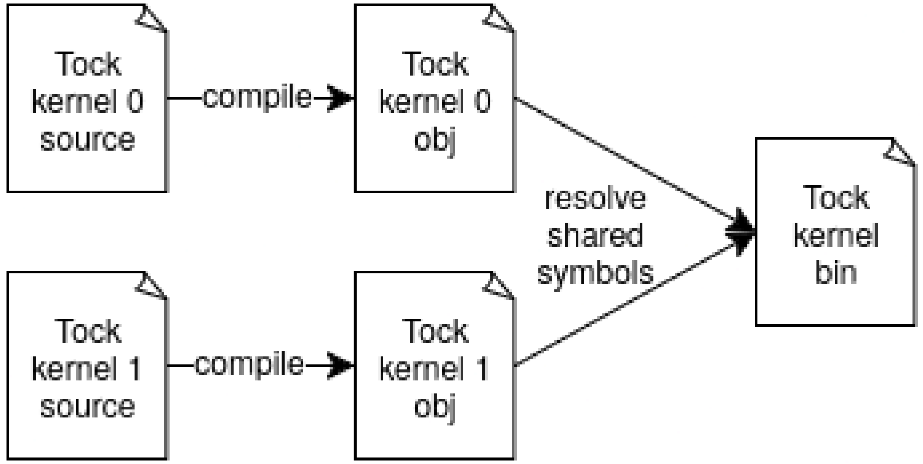
Building Multikernel Tock

- Build each kernel instance separately
- Resolved shared symbols
  - Hardware portals communicate through shared memory



Building Multikernel Tock

- Build each kernel instance separately
- Resolved shared symbols
  - Hardware portals communicate through shared memory
- Prepare the final image (board-dependent)



Booting

Booting

- Each kernel instance is responsible for initializing their own memory

Booting

- Each kernel instance is responsible for initializing their own memory
  - Instance 0 is responsible for the shared memory

Booting

- Each kernel instance is responsible for initializing their own memory
  - Instance 0 is responsible for the shared memory
- `Portals` are available *iff all instances are ready*
  - A (one and only) spin lock is used

Future Work

- Safely sharing physical devices
  - *When* it is OK to move a device?
- Process Migration?

Questions?