Tock Embedded OS Training

SOSP 2017

Please make sure you have completed all of the tutorial pre-requisites. If you prefer, you can download a virtual machine image with all the pre-requisites already installed.

https://github.com/helena-project/tock/tree/master/doc/courses/sosp/README.md

aka

https://goo.gl/s17fy8

Tock

A secure operating system for microcontrollers

- Kernel components in Rust
- Type-safe API for safe driver development
- Hardware isolated processes for application code

Microcontrollers

System-on-a-chip with integrated flash, SRAM, CPU and a bunch of hardware controllers.

Typically:

- Communication: UART, SPI, I2C, USB, CAN...
- External I/O: GPIO, external interrupt, ADC, DAC
- Timers: RTC, countdown timers

Maybe...

- Radio (Bluetooth, 15.4)
- Cryptographic accelerators
- Other specialized hardware...

Low Resource

- 10's of µA average power draw
- 10's of kBs of RAM
- Moderate clock speeds

Use cases

- Security applications (e.g. authentication keys)
- Sensor networks
- Programmable wearables
- PC/phone peripherals
- Home/industrial automation
- Flight control

Two types of components: capsules and processes

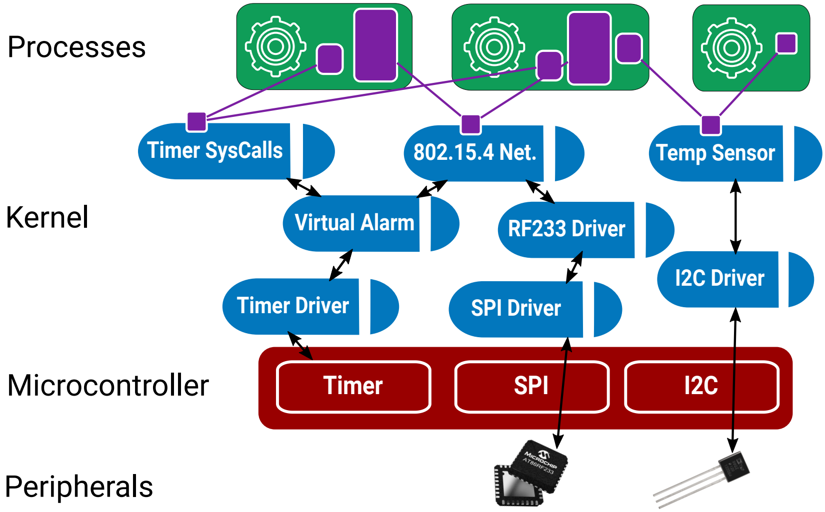

Figure 1

Two types of scheduling: cooperative and preemptive

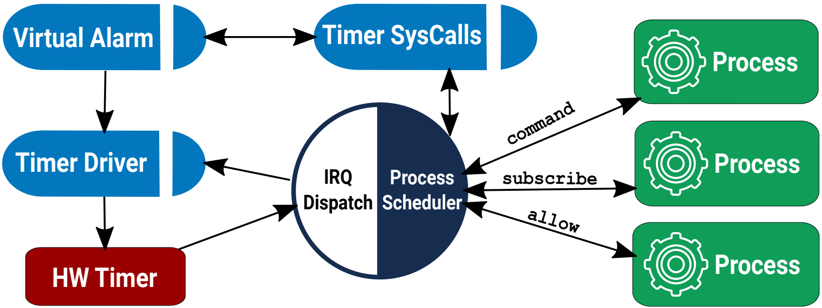

Figure 2

Agenda Today

1. Intro to hardware, tools and development environment
2. Write an end-to-end Bluetooth Low Energy environment sensing application
3. Add functionality to the Tock kernel

Part 1: Hardware, tools and development environment

Hail

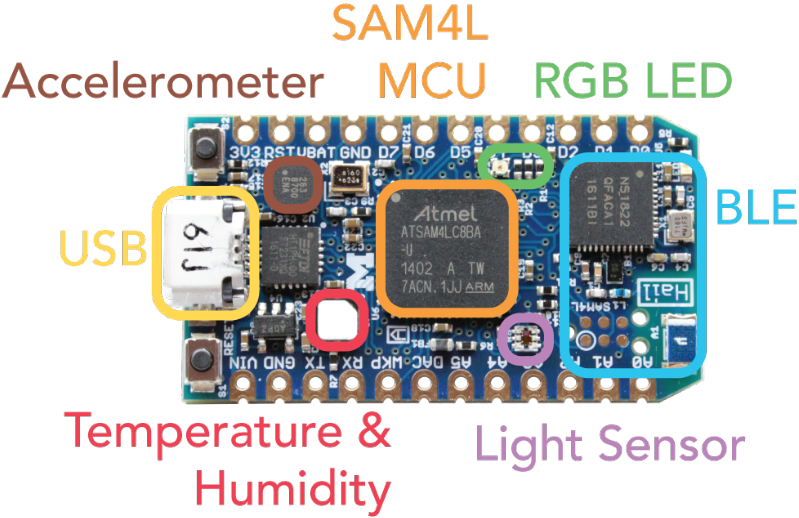

Figure 3

Binaries on-board

Bootloader

Kernel

Processes

Tools

- `make` (just instrumenting `xargo`)
- Rust (nightly for `asm!`, compiling `core`, etc)
- `xargo` to automate compiling base libraries
- `arm-none-eabi` GCC/LD to link binaries
- `tockloader` to interact with Hail and the bootloader

Tools: tockloader

Write a binary to a particular address in flash

```
$ tockloader flash --address 0x1000 \
    target/thumbv7em-none-eabi/release/hail.bin
```

Program a process in Tock Binary Format[1]:

```
$ tockloader install myapp.tab
```

Restart the board and connect to the debug console:

```
$ tockloader listen
```

[1] TBFs are relocatable process binaries prefixed with headers like the package name. .tab is a tarball of TBFs for different architectures as well as a metadata file for tockloader.

Check your understanding

1. What kinds of binaries exist on a Tock board? Hint: There are three, and only two can be programmed using `tockloader`.
2. Can you point to the chip on the Hail that runs the Tock kernel? How about the processes?
3. What steps would you follow to program a processes onto Hail? What about to replace the kernel?

Hands-on: Set-up development environment

1. Compile and flash the kernel
2. (Optional) Familiarize yourself with `tockloader` commands
   - `uninstall`
   - `list`
   - `erase-apps`
3. (Optional) Add some other apps from the repo, like `blink` and `sensors`

- Head to `<http://bit.ly/2lniNt6>` to get started!
- (github.com/helena-project/tock/blob/master/doc/courses/sosp/environment.md)

Part 2: User space

System calls

Call	Target	Description
command	Capsule	Invoke an operation on a capsule
allow	Capsule	Share memory with a capsule
subscribe	Capsule	Register an upcall
memop	Core	Modify memory break
yield	Core	Block until next upcall is ready

C System Calls: command & allow

```
int command(uint32_t driver, uint32_t command,
            int arg1, int arg2);

int allow(uint32_t driver, uint32_t allow, void* ptr,
          size_t size);
```

C System Calls: subscribe

```c
typedef void (subscribe_cb)(int, int, int,
                            void* userdata);

int subscribe(uint32_t driver, uint32_t subscribe,
              subscribe_cb cb, void* userdata);
```

C System Calls: `yield` & `yield_for`

```c
void yield(void);

void yield_for(bool *cond) {
  while (!*cond) {
    yield();
  }
}
```

Example: printing to the debug console

```c
static void putstr_cb(int _x, int _y, int _z, void* ud) {
  putstr_data_t* data = (putstr_data_t*)ud;
  data->done = true;
}

int putnstr(const char *str, size_t len) {
  putstr_data_t data;
  data.buf = str;
  data.done = false;

  allow(DRIVER_NUM_CONSOLE, 1, str, len);
  subscribe(DRIVER_NUM_CONSOLE, 1, putstr_cb, &data);
  command(DRIVER_NUM_CONSOLE, 1, len, 0);
  yield_for(&data.done);
  return ret;
}
```

Inter Process Communication (IPC)

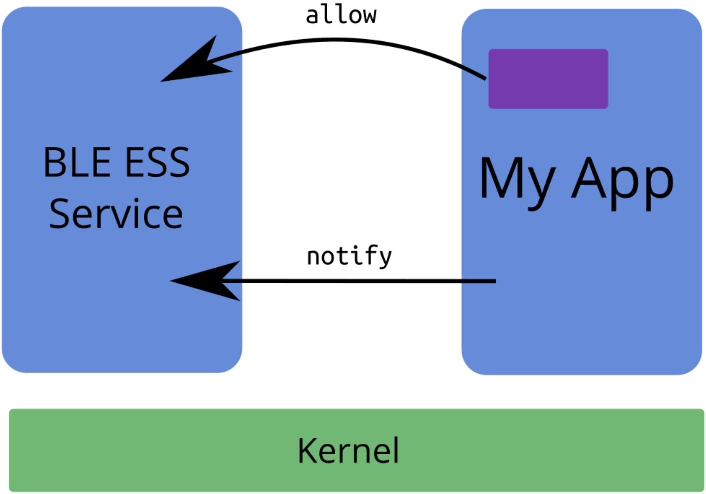

Figure 4

Tock Inter Process Communication Overview

Servers

- Register as an IPC service
- Call `notify` to trigger callback in connected client
- Receive a callback when a client calls `notify`

Clients

- Discover IPC services by application name
- Able to share a buffer with a connected service
- Call `notify` to trigger callback in connected service
- Receive a callback when service calls `notify`

Inter Process Communication API

```c
// discover IPC service by name
// returns error code or PID for service
int ipc_discover(const char* pkg_name);

// shares memory slice at address with IPC service
int ipc_share(int pid, void* base, int len);

// register for callback on server `notify`
int ipc_register_client_cb(int pid, subscribe_cb cb,
                           void* userdata);

// trigger callback in service
int ipc_notify_svc(int pid);

// trigger callback in a client
int ipc_notify_client(int pid);
```

Check your understanding

1. How does a process perform a blocking operation? Can you draw the flow of operations when a process calls `delay_ms(1000)`?
2. How would you write an IPC service to print to the console? Which functions would the client need to call?

Hands-on: Write a BLE environment sensing application

1. Get an application running on Hail
2. Print "Hello World" every second
3. Extend your app to sample on-board sensors
4. Extend your app to report through the `ble-env-sense` service

- Head to `<http://bit.ly/2hgpl8n>` to get started!
- (github.com/helena-project/tock/blob/master/doc/courses/sosp/application.md)

Part 3: The kernel

Trusted Computing Base (`unsafe` allowed)

- Hardware Abstraction Layer
- Board configuration
- Event & Process scheduler
- Rust `core` library
- Core Tock primitives

`kernel/`
`chips/`

Capsules (`unsafe` not allowed)

- Virtualization
- Peripheral drivers
- Communication protocols (IP, USB, etc)
- Application logic

`capsules/`

Constraints

Small isolation units

Breaking a monolithic component into smaller ones should have low/no cost

Avoid memory exhaustion in the kernel

No heap. Everything is allocated statically.

Low communication overhead

Communicating between components as cheap as an internal function call. Ideally inlined.

Event-driven execution model

```
pub fn main<P, C>(platform: &P, chip: &mut C,
                  processes: &mut [Process]) {
    loop {
        chip.service_pending_interrupts();
        for (i, p) in processes.iter_mut().enumerate() {
            sched::do_process(platform, chip, process);
        }

        if !chip.has_pending_interrupts() {
            chip.prepare_for_sleep();
            support::wfi();
        }
    }
}
```

Event-driven execution model

```
fn service_pending_interrupts(&mut self) {
    while let Some(interrupt) = get_interrupt() {
        match interrupt {
            ASTALARM => ast::AST.handle_interrupt(),
            USART0 => usart::USART0.handle_interrupt(),
            USART1 => usart::USART1.handle_interrupt(),
            USART2 => usart::USART2.handle_interrupt(),
            ...
        }
    }
}
```

Event-driven execution model

```rust
impl Ast {
    pub fn handle_interrupt(&self) {
        self.clear_alarm();
        self.callback.get().map(|cb| { cb.fired(); });
    }
}
impl time::Client for MuxAlarm {
    fn fired(&self) {
        for cur in self.virtual_alarms.iter() {
            if cur.should_fire() {
                cur.armed.set(false);
                self.enabled.set(self.enabled.get() - 1);
                cur.fired();
            }
        }
    }
}
```

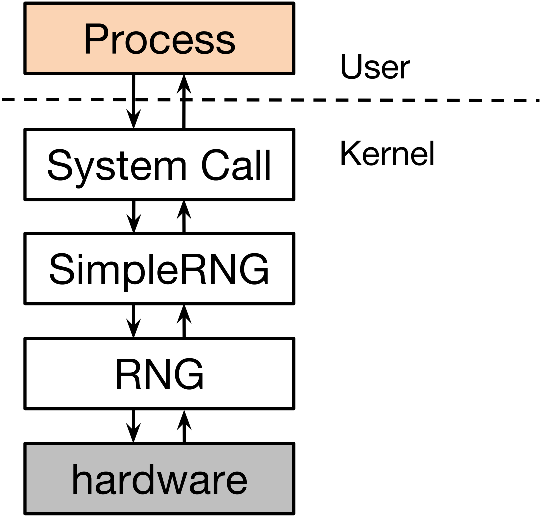

Figure 5: Capsules reference each other directly, assisting inlining

The mutable aliases problem

```
enum NumOrPointer {
  Num(u32),
  Pointer(&mut u32)
}

// n.b. will not compile
let external : &mut NumOrPointer;
match external {
  Pointer(internal) => {
    // This would violate safety and
    // write to memory at 0xdeadbeef
    *external = Num(0xdeadbeef);
    *internal = 12345;  // Kaboom
  },
  ...
}
```

Interior mutability to the rescue

Type	Copy-only	Mutual exclusion	Opt.	Mem Opt.
`Cell`	✓	✗	✓	✓
`VolatileCell`	✓	✗	✗	✓
`TakeCell`	✗	✓	✗	✓
`MapCell`	✗	✓	✓	✗

```
pub struct Fxos8700cq<`a> {
  i2c: &`a I2CDevice,
  state: Cell<State>,
  buffer: TakeCell<`static, [u8]>,
  callback:
    Cell<Option<&`a hil::ninedof::NineDofClient>>,
}

impl<`a> I2CClient for Fxos8700cq<`a> {
  fn cmd_complete(&self, buf: &`static mut [u8]) { ... }
}

impl<`a> hil::ninedof::NineDof for Fxos8700cq<`a> {
  fn read_accelerometer(&self) -> ReturnCode { ... }
}

pub trait NineDofClient {
  fn callback(&self, x: usize, y: usize, z: usize);
}
```

Check your understanding

1. What is a `VolatileCell`? Can you find some uses of `VolatileCell`, and do you understand why they are needed? Hint: look inside `chips/sam4l/src`.
2. What is a `TakeCell`? When is a `TakeCell` preferable to a standard `Cell`?

Hands-on: Write and add a capsule to the kernel

1. Read the Hail boot sequence in `boards/hail/src/main.rs`
2. Write a new capsule that prints "Hello World" to the debug console.
3. Extend your capsule to print "Hello World" every second
4. Extend your capsule to print light readings every second
5. Extra credit

- Head to <http://bit.ly/2zLoD9W> to get started!
- (github.com/helena-project/tock/blob/master/doc/courses/sosp/capsule.md)

Stay in touch!

https://www.tockos.org

https://github.com/helena-project/tock

tock-dev@googlegroups.com

#tock on Freenode