Safely and Efficiently Multiprogramming a 64kB Computer


June 14th, 2017

A few months before SOSP
Emerging class of embedded applications are software platforms, rather than single purpose devices.
Embedded Software

- No isolation between components
- Deeply coupled components
- Static memory allocation to avoid unrecoverable runtime memory exhaustion
- Fixed concurrency at compile-time
Embedded Hardware

- Low-power budget—micro-amps average current consumption
- 64kB of RAM (to limit the sleep current)
- Memory Protection Unit—a limited hardware protection mechanism
Challenges

Software platforms must support multiple independent dynamically-loadable apps
-- New performance and safety concerns

- How to isolate components despite minimal hardware resources?
- How to replace individual components without restarting the whole system?
- *How to avoid fixed concurrency with limited memory*
Give up on isolation—write completely bug-free code
Common Solutions

- Give up on isolation—write completely bug-free code
- Whole system updates only
Common Solutions

- Give up on isolation—write completely bug-free code
- Whole system updates only
- Use *nix et al—forget about low power
Tock
Tock is a new operating system for low-power platforms that takes advantage of the limited hardware-protection mechanisms available on recent microcontrollers and the type-safety features of the Rust programming language to provide a multiprogramming environment:

- Isolation of software faults
- Efficient memory protection and management for dynamic application workloads
- Update/restart/remove individual (user-space) components independently
- Retains dependability requirements of long-running devices. -- Often run unattended
Tock Architecture

Processes
- Timer SysCalls
- 802.15.4 Net.
- Temp Sensor

Kernel
- Virtual Alarm
- RF233 Driver
- I2C Driver
- Timer Driver
- SPI Driver

Microcontroller
- Timer
- SPI
- I2C

Peripherals
- Microcontroller
- Temp Sensor
Capsules

- Capsules are components in the kernel
- Minimal runtime overhead:
  - Isolated “at compile-time” using the Rust language type/module system
  - Cooperatively scheduled
  - Can eliminate most isolation at compile-time

Capsules can...

- Violate real-time guarantees
- Panic (sort of... lets talk...)

But they cannot...

- Read arbitrary memory (secret encryption keys)
- Communicate with peripherals it’s not allowed to
Capsules

Stronger memory isolation than hardware protection?

```rust
struct DMAChannel {
    enabled: bool,
    buffer: &'static [u8],
}
```

Typing hardware register can constrain allowed values with very fine granularity.

--- Important because DMA hardware can manipulate data at any address
Processes

Can be unreliable since the system can respawn or kill processes without affecting other functionality.

- Hardware isolated concurrent executions of programs
- Written in any language (currently C, C++, Lua and Rust-ish)
- Total control over their memory, including dynamic heap allocation.
- Similar to processes in other systems.
  - Separate stacks allows preemptive execution
  - Memory isolated by the hardware
- Interact with kernel over a small but flexible system-call interface:
  - command, subscribe, allow
  - yield, memop

Non-blocking API
command: passes a word-size integer to a capsule
allow: passes data buffers from processes to capsules
subscribe: sets up a callback
What happens when the kernel requires dynamic resources to respond to a request from a process?

▶ We want to allow arbitrary apps so we don’t know concurrency requirements:
  ▶ How many timers will an application need?
  ▶ Will it use SPI, UART, USB, Bluetooth, etc? One socket? 1000 sockets?
▶ If the kernel allocates memory for requests dynamically, it may run out of resources.
TOSThreads has low memory efficiency. Static allocation costs 710-712 bytes per thread, of which at most 158 bytes (22%) can be in use at any time. These numbers do not include the thread stacks, each of which can be less than 100 bytes.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Kernel RAM</th>
<th>Syscall RAM</th>
<th>Max Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3506</td>
<td>712</td>
<td>158</td>
</tr>
<tr>
<td>2</td>
<td>4216</td>
<td>1422</td>
<td>316</td>
</tr>
<tr>
<td>3</td>
<td>4928</td>
<td>2134</td>
<td>474</td>
</tr>
</tbody>
</table>

TOSThreads, is a threads package for TinyOS that combines the ease of a threaded programming model with the efficiency of an event-based kernel.
Grants

Tock allows a process to “grant” to the kernel portions of its own memory, which the kernel can use to maintain state for process requests.

- Separate sections of kernel heap located in each process’s memory space.
- Grant allocations for one process do not affect kernel’s ability to allocate for another.
- Type-safe interface guarantees all grants for a process can be freed immediately if the process dies.

**Basic idea:** *kernel API ensures there are no long-lived pointers directly to grant-allocated memory.*
Grant Requirements

- Process cannot access grant allocated memory
  - We use an additional, dynamically determined MPU rule
- Ensure grant-allocated values unavailable to capsules once process dies through limited API:
  - Capsules pass a closure to the `enter` method
  - Memory in a grant region only accessible from within closure
  - Pointers to grant memory cannot escape the closure
  - Implications on kernel design: should avoid cross process data-structures
impl<T: Default> Grant {
    fn create() -> Grant<T>

    fn enter<F,R>(&self, proc_id: ProcId, func: F) -> Result<R, Error> where
        F: for<'b> FnOnce(&'b mut Owned<T>) -> R, R: Copy

    fn each<F>(&self, func: F) where
        F: for<'b> Fn(&'b mut Owned<T>)
}
Recall: TOSThreads requires 700 bytes statically allocated in the kernel for each additional thread. At most 22% can be used at any given time.

- Grants require *no additional per-thread memory* in the kernel
- Only useful memory is dynamically allocated in grants
- *Zero wasted* memory since it can re-use memory for non-concurrent operations.
Resource constraints continue to be a challenge for embedded system designers.

- Low-power, small form-factors and lower cost

- These limitations *should not* preclude software abstractions and protections common in general-purpose computers.

- Tock provides both dynamic operation and dependability in resource-constrained settings.

  - Best of all: flexible multiprogramming, isolation, system dependability

- Grants split the kernel heap across processes, allowing dynamic demands for kernel resources despite limited system memory

Buy a Hail! [https://tockos.org/hardware/hail](https://tockos.org/hardware/hail)