IoT Operating Systems

Chenyang Lu
Critiques

- 1/2 page critiques of research papers
- Due at 10am on the class day (hard deadline)
- Email Corey <dairuixuan@wustl.edu> in plain txt
- Back-of-envelop notes - NOT whole essays
Critique #1 (IoT OS)

- Due on 9/24
Critique #2 (Wireless)

- Due on 10/15
Proposal Presentation

- In class on 10/1

- 7 min per group
  - 6-min talk + 1-min Q&A
  - 4 slides
  - Rehearse over Zoom
  - Turn on your video during your presentation

- Your elevator pitch!

- Email Corey your slides before class
Written Proposal

- One proposal/team, one page
  - Team members
  - Concise description of project
  - Responsibilities of each member
  - Equipment needed

- Written proposal due: **10/1, 11:59pm**
  - Email to Corey
Demo I

- In class on **10/27** and **10/29**.
- **15 min** per team.
- Must show something **real**.
- Send Corey a video before class as backup.
Demo II

- In class on **11/17** and **11/19**.

- **15 min** per team.

- Substantial progress → final demo.

- Send Corey a video before class as backup.
IoT OS

- **Linux**
- **Windows 10 IoT Core**: Windows 10 optimized for ARM and x86/x64.
- **Amazon FreeRTOS**: open-source FreeRTOS kernel + libraries to securely connect devices to AWS cloud services.
- **Arm Mbed**: open-source OS for Arm Cortex-M microcontrollers.
- **Contiki**: open-source, multi-threaded OS.

![IoT Operating Systems Chart](image-url)
Amazon FreeRTOS

IoT operating system for microcontrollers that extends the FreeRTOS kernel with libraries for security, connectivity, and updateability.

Microcontroller-based smart lighting

Amazon FreeRTOS Bluetooth Low Energy
Secure direct connection to mobile devices via Bluetooth Low Energy

AWS IoT Core
AWS IoT Device Management
AWS IoT Device Defender
AWS IoT Analytics
Direct connection to cloud services like AWS IoT Core

https://aws.amazon.com/freertos
Diverse Platforms

**TelosB**
- TI MSP430 microcontroller, 4/8 MHz, 8 bit
- Memory: 10KB data, 48 KB program
- IEEE 802.15.4 radio: max 250 Kbps

**Raspberry Pi 4**
- Quad core Cortex-A72 64-bit SoC, 1.5GHz
- 2GB-8GB SDRAM
- IEEE 802.11ac wireless, Bluetooth 5.0, BLE
- Gigabit Ethernet

Chenyang Lu
Epic Core

CC2420 radio
802.15.4
6LoWPAN/IPv6

2.5 x 2.5 cm

RAM 10 KB
Flash 48 KB
TI MSP430
Clock 4/8 MHz

I/O (some shared)
8 ADC (12 bit)
2 DAC (12 bit)
1 I2C
1 JTAG
1 1-Wire
2 SPI
2 UART

8 general, 8 interrupt, and 5 special pin connectors

16 MB Flash memory
Unique hardware ID

3 V

Typical sleep current 9μA at 3V, radio active ~20mA
TelosB

- Six major I/O devices
- Possible Concurrency
  - I²C, SPI, ADC
- Energy Management
  - Turn peripherals on only when needed
  - Turn off otherwise

**Diagram:**
- Humidity
- Temp
- MSP430
- ADC
- SPI
- Radio
- Flash
- Total Solar
- Photo Active
Severe constraints on power, size, and cost →

- slow microprocessor
- low-bandwidth radio
- limited memory
- limited hardware parallelism → CPU hit by many interrupts!
- manage sleep modes in hardware components
Software Challenges

- Small memory footprint
- Efficiency - power and processing
- Concurrency-intensive operations
- Diversity in applications & platform → efficient modularity
  - Support reconfigurable hardware and software
OS: Basic Functions

- OS controls resources:
  - who gets the CPU;
  - when I/O takes place;
  - how much memory is allocated;
  - power management

- Application programs run on top of OS services

- Challenge: manage multiple, concurrent tasks.
Example: Engine Control

Concurrent tasks
- spark control
- crankshaft sensing
- fuel/air mixture
- oxygen sensor

Chenyang Lu
A process is a unique execution of a program.
  - Several copies of a program may run simultaneously.

A process has its own context.
  - Data in registers, Program Counter (PC), status.
  - Stored in Process Control Block (PCB)

Thread: lightweight process
  - Threads share memory space in a same process.

OS manages processes and threads.
Traditional OS

- Multi-threaded
- Preemptive scheduling
- Threads:
  - ready to run;
  - executing on the CPU;
  - waiting for data.
Preemptive Priority Scheduling

- Each process has a fixed priority (1 highest);
- $P_1$: priority 1; $P_2$: priority 2; $P_3$: priority 3.
Context Switch

Chenyang Lu
Limitations of Traditional OS

- Multi-threaded + preemptive scheduling
  - Preempted threads waste memory
  - Context switch overhead

- I/O
  - Blocking I/O: waste memory on blocked threads
  - Polling (busy-wait): waste CPU cycles and power
Existing Embedded OS

<table>
<thead>
<tr>
<th>Name</th>
<th>Code Size</th>
<th>Target CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>pOSEK</td>
<td>2K</td>
<td>Microcontrollers</td>
</tr>
<tr>
<td>pSOSSystem</td>
<td></td>
<td>PII-&gt;ARM Thumb</td>
</tr>
<tr>
<td>VxWorks</td>
<td>286K</td>
<td>Pentium -&gt; Strong ARM</td>
</tr>
<tr>
<td>QNX Nutrino</td>
<td>&gt;100K</td>
<td>Pentium II -&gt; NEC</td>
</tr>
<tr>
<td>QNX RealTime</td>
<td>100K</td>
<td>Pentium II -&gt; SH4</td>
</tr>
<tr>
<td>OS-9</td>
<td></td>
<td>Pentium -&gt; SH4</td>
</tr>
<tr>
<td>Chorus OS</td>
<td>10K</td>
<td>Pentium -&gt; Strong ARM</td>
</tr>
<tr>
<td>ARIEL</td>
<td>19K</td>
<td>SH2, ARM Thumb</td>
</tr>
<tr>
<td>Creem</td>
<td>560 bytes</td>
<td>ATMEL 8051</td>
</tr>
</tbody>
</table>

- QNX context switch = 2400 cycles on x86
- pOSEK context switch > 40 µs
- Creem -> no preemption

TinyOS Solutions

- **Efficient modularity**
  - Application = scheduler + graph of components
  - Compiled into one executable
  - Only needed components are compiled/loaded

- **Concurrency**: event-driven architecture

---

Modified from D. Culler et al., TinyOS boot camp presentation, Feb 2001
Example: Surge
Example: Mica2 Mote

- Microcontroller: 7.4 MHz, 8 bit
- Memory: 4KB data, 128 KB program
- Radio: max 38.4 Kbps
- Sensors: Light, temperature, acceleration, acoustic, magnetic…
- Power
  - <1 week on two AA batteries in active mode
  - >1 year battery life on sleep modes!
Example: Application

D. Culler et. Al., TinyOS boot camp presentation, Feb 2001

application

routing

Routing Layer

messaging

Messaging Layer

packet

Radio Packet

byte

Radio Byte (MAC)

bit

RFM

sensing application

photo

Temp

clocks

ADC

i2c

SW

HW

Chenyang Lu
Two-level Scheduling

- Events handle interrupts
  - Interrupts trigger lowest level events
  - Events can signal events, call commands, or post tasks
- Tasks perform deferred computations
- Interrupts preempt tasks and interrupts
Multiple Data Flows

- **Respond quickly:** sequence of events.commands through the component graph.
  - Immediate execution of function calls
  - e.g., get bit out of radio before it gets lost.

- **Post tasks for deferred computations.**
  - e.g., encoding.

- **Events preempt tasks to handle new interrupts.**
Timing diagram of event propagation
(step 0-6 takes about 95 microseconds total)
Scheduling

- Interrupts preempt tasks
  - Respond quickly
  - Event/command implemented as function calls

- Task cannot preempt tasks
  - Reduce context switch → efficiency
  - Single stack → low memory footprint
  - TinyOS 2 supports pluggable task scheduler (default: FIFO).

- Scheduler puts processor to sleep when
  - no event/command is running
  - task queue is empty
Space Breakdown...

Code size for ad hoc networking application

- Interrupts
- Message Dispatch
- Initialization
- C-Runtime
- Light Sensor
- Clock
- Scheduler
- Led Control
- Messaging Layer
- Packet Layer
- Radio Interface
- Routing Application
- Radio Byte Encoder

Scheduler: 144 Bytes code
Totals: 3430 Bytes code
226 Bytes data

D. Culler et. Al., TinyOS boot camp presentation, Feb 2001
Lithium battery runs for 35 hours at peak load and years at minimum load!

- That’s three orders of magnitude difference!

A one byte transmission uses the same energy as approximately 11000 cycles of computation.
### Time Breakdown...

<table>
<thead>
<tr>
<th>Components</th>
<th>Packet reception work breakdown</th>
<th>CPU Utilization</th>
<th>Energy (nj/Bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM</td>
<td>0.05%</td>
<td>0.20%</td>
<td>0.33</td>
</tr>
<tr>
<td>Packet</td>
<td>1.12%</td>
<td>0.51%</td>
<td>7.58</td>
</tr>
<tr>
<td>Radio handler</td>
<td>26.87%</td>
<td>12.16%</td>
<td>182.38</td>
</tr>
<tr>
<td>Radio decode thread</td>
<td>5.48%</td>
<td>2.48%</td>
<td>37.2</td>
</tr>
<tr>
<td>RFM</td>
<td>66.48%</td>
<td>30.08%</td>
<td>451.17</td>
</tr>
<tr>
<td>Radio Reception</td>
<td>-</td>
<td>-</td>
<td>1350</td>
</tr>
<tr>
<td>Idle</td>
<td>-</td>
<td>54.75%</td>
<td>-</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>100.00%</td>
<td>100.00%</td>
<td>2028.66</td>
</tr>
</tbody>
</table>

- 50 cycle task overhead (6 byte copies)
- 10 cycle event overhead (1.25 byte copies)
Advantages

- **Small memory footprint**
  - Only needed components are complied/loaded
  - Single stack for tasks

- **Power efficiency**
  - Put CPU to sleep whenever the task queue is empty
  - TinyOS 2 (ICEM) provides power management for peripherals.

- **Efficient modularity**
  - Event/command interfaces between components
  - Event/command implemented as function calls

- **Concurrency-intensive operations**
  - Event/command + tasks
Critiques

- No protection barrier between kernel and applications
- No preemptive scheduling $\rightarrow$ a real-time task may wait for non-urgent ones
- Static linking $\rightarrow$ cannot change parts of the code dynamically
- Virtual memory?
nesC

- Programming language for TinyOS and applications
- Support TinyOS components

- Whole-program analysis at compile time
  - Improve robustness: detect race conditions
  - Optimization: function inlining

- Static language
  - No function pointer
  - No malloc
  - Call graph and variable access are known at compile time
Application

- **Interfaces**
  - provides interface
  - uses interface

- **Implementation**
  - module: C behavior
  - configuration: select & wire

---

Chenyang Lu
Module

- **Interfaces**
  - `provides` interface
  - `uses` interface

- **Implementation**
  - `module`: C behavior
  - `configuration`: select & wire

```c
module TimerP {
    provides {
        interface StdControl;
        interface Timer;
    }
    uses interface Clock;
    ...
}
```
interface Clock {
    command error_t setRate(char interval, char scale);
    event error_t fire();
}

interface Send {
    command error_t send(message_t *msg, uint16_t length);
    event error_t sendDone(message_t *msg, error_t success);
}

interface ADC {
    command error_t getData();
    event error_t dataReady(uint16_t data);
}

**Bidirectional** interface supports split-phase operation
module SurgeP {
    provides interface StdControl;
    uses interface ADC;
    uses interface Timer;
    uses interface Send;
}

implementation {
    bool busy;
    norace uint16_t sensorReading;
    async event result_t Timer.fired() {
        bool localBusy;
        atomic {
            localBusy = busy;
            busy = TRUE;
        }
        if (!localBusy)
            call ADC.getData();
        return SUCCESS;
    }
    async event result_t ADC.dataReady(uint16_t data) {
        sensorReading = data;
        post sendData();
        return SUCCESS;
    }
    ...
}
configuration TimerC {
  provides {
    interface StdControl;
    interface Timer;
  }
}

implementation {
  components TimerP, HWClock;

  StdControl = TimerP.StdControl;
  Timer = TimerP.Timer;

  TimerP.Clock -> HWClock.Clock;
}
Example: Surge

SurgeC

BootC

SurgeP

StdControl

ADC

Timer

SendMsg

Leds

StdControl

TimerC

MultihopC

LedsC

StdControl

PhotoC

StdControl

SendMsg

Leds

Send Msg
Concurrency

- Race condition: concurrent interrupts/tasks update shared variables.

- Only interrupts cause preemption $\rightarrow$ concurrency
  - Asynchronous code (AC): reachable from at least one interrupt.
  - Synchronous code (SC): reachable from tasks only.

- Any update of a shared variable from AC is a potential race condition!
module SurgeP { ... }
implementation {
    bool busy;
    norace uint16_t sensorReading;
    async event result_t Timer.fired() {
        if (!busy) {
            busy = TRUE;
            call ADC.getData();
        }
        return SUCCESS;
    }
    task void sendData() { // send sensorReading
        adcPacket.data = sensorReading;
        call Send.send(&adcPacket, sizeof adcPacket.data);
        return SUCCESS;
    }
    async event result_t ADC.dataReady(uint16_t data) {
        sensorReading = data;
        post sendData();
        return SUCCESS;
    }
}
Atomic Sections

```c
atomic {
    <Statement list>
}
```

- Disable interrupt when atomic code is being executed
- But cannot disable interrupt for long!
  - No loop
  - No command/event
  - Function calls OK, but callee must meet restrictions too
module SurgeP { ... }
implementation {
    bool busy;
    norace uint16_t sensorReading;

    async event result_t Timer.fired() { 
        bool localBusy;
        atomic {
            localBusy = busy;
            busy = TRUE;
        }
        if (!localBusy) 
            call ADC.getData();
        return SUCCESS;
    }
}
nesC Compiler

- **Race-free invariant**: any update of a shared variable
  - is from SC only, or
  - occurs within an *atomic* section.

- Compiler returns error if the invariant is violated.

- **Fix**
  - Make access to shared variables *atomic*.
  - Move access to shared variables to *tasks*.
Results

- Tested on full TinyOS code, plus applications
  - 186 modules (121 modules, 65 configurations)
  - 20-69 modules/app, 35 average
  - 17 tasks, 75 events on average (per application) - lots of concurrency!

- Found 156 races: 103 real
  - About 6 per 1000 lines of code!

- Fixed races:
  - Add atomic sections
  - Post tasks (move code to task context)
Function Inlining

```c
int foo(a,b,c) { return a + b - c; }
z = foo(w,x,y);

⇒

z = w + x - y;
```

- Improve performance by eliminating function call overhead.
- May increase code size, but not always…
- Affect instruction cache behavior.
Optimization: Inlining

- Inlining improves performance and reduces code size.
- Why?

<table>
<thead>
<tr>
<th>App</th>
<th>Code size</th>
<th>Code reduction</th>
<th>Data size</th>
<th>CPU reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>inlined</td>
<td>noninlined</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Surge</td>
<td>14794</td>
<td>16984</td>
<td>12%</td>
<td>1188</td>
</tr>
<tr>
<td>Maté</td>
<td>25040</td>
<td>27458</td>
<td>9%</td>
<td>1710</td>
</tr>
<tr>
<td>TinyDB</td>
<td>64910</td>
<td>71724</td>
<td>10%</td>
<td>2894</td>
</tr>
</tbody>
</table>
Overhead for Function Calls

- **Caller**: call a function
  - Push return address to stack
  - Push parameters to stack
  - Jump to function

- **Callee**: receive a call
  - Pop parameters from stack

- **Callee**: return
  - Pop return address from stack
  - Push return value to stack
  - Jump back to caller

- **Caller**: return
  - Pop return value

*Overhead instructions for function calls!*
Principles Revisited

- Support TinyOS components
  - Interface, modules, configuration

- Whole-program analysis and optimization
  - Improve robustness: detect race conditions
  - Optimization: function inlining
  - More: memory footprint.

- Static language
  - No malloc, no function pointers
Critiques

- No dynamic memory allocation
  - Bound memory footprint
  - Allow offline footprint analysis
  - How to size buffer when data size varies dynamically?

- Restriction: no “long-running” code in
  - command/event handlers
  - atomic sections
Reading


  - Purchase the book online
  - Download the first half of the published version for free

- http://www.tinyos.net/