TinyOS and nesC

- TinyOS: OS for wireless sensor networks.
- nesC: programming language for TinyOS.
Mica2 Mote

- **Processor**
  - 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!
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
Traditional OS

- Multi-threaded
- Preemptive scheduling
- Threads:
  - ready to run;
  - executing on the CPU;
  - waiting for data.
Pros and Cons 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

Chenyang Lu
Example: 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

- Process 1
- Process 2
- Memory
- PC
- Registers
- CPU
# 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
Typical Application

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

application

routing

Messaging Layer

Radio Packet

Radio Byte (MAC)

byte

RFM

bit

sensing application

ADC

photo

Temp

i2c

clocks

Routing Layer

HW

SW

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 event/command through the component graph.
  - Immediate execution of function calls
  - e.g., get bit out of radio hw before it gets lost.
- **Post tasks for deferred computations.**
  - e.g., encoding.
- **Events preempt tasks to handle new interrupts.**
Sending a Message

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 $\rightarrow$ efficiency
  - Single stack $\rightarrow$ 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

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 approx 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
Issues

- No protection barrier between kernel and applications
- No preemptive real-time scheduling
  - Urgent task may wait for non-urgent ones
- Static linking
  - Cannot change parts of the code dynamically
- Virtual memory?
More

- Multi-threaded vs. event-driven architectures
  - Lack empirical comparison against existing OSes
  - A “standard” OS is more likely to be adopted by industry
  - Jury is still out…

- Alternative: Contiki
  - Multi-threaded OS

- Alternative: Native Java Virtual Machine
  - Java programming
  - Virtual machine provides protection
  - Example: Sun SPOT
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

```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

StdControl

SurgeP

ADC

Timer

SendMsg

Leds

StdControl

PhotoC

TimerC

MultihopC

Leds
Concurrency

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

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

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

<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>

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

- 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/