Our TAs

Mo Sha
- gowustl@gmail.com
- TinyOS tutorial
- Help students with projects.
- Manage motes.
- Grade projects.
- Office Hour: Tue/Thu 5:30-6.
- Bryan 502A

Yong Fu
- fuy@cs.wustl.edu
- Grade critiques.
- Office Hour: by appointment.
- Bryan 502D

Top 11 Technologies of the Decade

1. Smartphones
2. Social Networking
3. Voice over IP
4. LED Lighting
5. Multicore CPUs
6. Cloud Computing
7. Drone Aircraft
8. Planetary Rovers
9. Flexible AC Transmission
10. Digital Photography
11. Class-D Audio

TinyOS and nesC

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

Hardware Evolution

- Miniature devices manufactured economically
  - Microprocessors
  - Sensors/actuators
  - Wireless chips

Hardware Constraints

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

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!
Software Challenges

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

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

Context Switch

<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>pOSEKsys</td>
<td></td>
<td></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>eXtreme</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chorus OS</td>
<td>&gt;100K</td>
<td>Pentium -&gt; SH4</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>

Existing Embedded OS

- QNX context switch ~ 2400 cycles on u86
- 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

Example: Surge

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 → 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

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

Chenyang Lu

Power Breakdown...

<table>
<thead>
<tr>
<th>Components</th>
<th>Active</th>
<th>Idle</th>
<th>Sleep</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>5 mA</td>
<td>2 mA</td>
<td>0 μA</td>
</tr>
<tr>
<td>Radio</td>
<td>7 mA (TX)</td>
<td>4.5 mA (RX)</td>
<td>5 μA</td>
</tr>
<tr>
<td>EE-Prom</td>
<td>3 mA</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LED's</td>
<td>4 mA</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Photo Diode</td>
<td>200 μA</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Temperature</td>
<td>200 μA</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- 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.

Chenyang Lu

Space Breakdown...

Code size for ad hoc networking application

- Interrupts: 144 Bytes code
- Message Dispatch: 200 Bytes code
- Initialization: 200 Bytes code
- C-Runtime: 250 Bytes code
- Light Sensor: 150 Bytes code
- Clock: 100 Bytes code
- Scheduler: 144 Bytes code
- Led Control: 100 Bytes code
- Packet Layer: 100 Bytes code
- Routing Application: 100 Bytes code
- Radio Byte Encoder: 100 Bytes code
- Totals: 3430 Bytes code
- 226 Bytes data

Chenyang Lu

Disadvantages

- Lack preemptive real-time scheduling
  - Urgent task may wait for non-urgent ones
- Lack flexibility
  - Static linking only
  - Cannot change parts of the code dynamically
- Lack virtual memory

Chenyang Lu
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: 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

- Implementation
  - module: C behavior
  - configuration: select & wire
- Interfaces
  - provides interface
  - uses interface

```
module TimerP {
  provides {
    interface StdControl;
    interface Timer;
  }
  uses interface Clock;
  ...
}
```

Interface

```
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

```
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

```
configuration TimerC {
  provides {
    interface StdControl;
    interface Timer;
  }
  implementation {
    components TimerP, HWClock;
    StdControl = TimerP.StdControl;
    Timer = Timer.P.Timer;
    TimerP.Clock -> HWClock.Clock;
  }
```

```
Module
```

```
Configuration
```
Example: Surge

A 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;
  }
}

Concurrency

- Race condition: concurrent interrupts/tasks update shared variables.
- Asynchronous code (AC): reachable from at least one interrupt handler.
- Synchronous code (SC): reachable from tasks only.
- Any update of a shared variable from AC is a potential race condition.

Atomic Sections

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

Prevent Race

module SurgeP { ... }
implementation {
  bool busy;
  norace uint16_t sensorReading;
  async event result_t Timer.fired() {
    bool localBusy;
    atomic { localBusy = busy; }
    if (!localBusy) {
      busy = TRUE;
      call ADC.getData();
      return SUCCESS;
    }
}

necS Compiler

- Race-free invariant: Any update to 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>84910</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

### Issues

- No dynamic memory allocation
  - Bound memory footprint
  - Allow offline footprint analysis
  - But
  - How to size buffer when data size varies dynamically?
- Restriction: no “long-running” code in
  - Command/event handlers
  - Atomic sections

### 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

### Reading

  - Purchase the book online
  - Download the first half of the published version for free.
  - [http://www.tinyos.net](http://www.tinyos.net)
 Proposal

- One proposal/team, 1-2 pages
  - Team members
  - Concise description of project
  - Responsibilities of each member
  - Specific equipment needed

- Written proposal due: 9/22, 11:59pm
  - Email to Mo and CC me
  - Subject: [CSE 5205] Proposal: Project Name

- Proposal presentation: 9/22, in class