Towards a Resilient Operating System for Wireless Sensor Networks

Hyoseung Kim    Hojung Cha
Yonsei University, Korea

2006. 6. 1.

Hyoseung Kim
hskim@cs.yonsei.ac.kr
Motivation (1)

- **Problems: Application errors on sensor networks**

  - Diverse hardware devices
  - Cooperate with large number of nodes
  - Unattended operation for long-time

  - Kernel data corruption
  - Kernel code execution
  - Hardware control faults

  **Kernel fail & sensor node crash**
  
  - No way to recover from system failure
  - Crashed sensor node is useless in the field

  **System safety like general purpose OS needed!**
Motivation (2)

- **Popular sensor nodes: Telos / Mica**
  - No MMU, privileged mode, and exceptions

- **Current sensor network systems**
  - No system safety mechanism

- **TinyOS**
  - MoteIV Telos Rev. B
    - TI MSP430 8Mhz
    - 10Kb RAM, 60Kb ROM

- **SOS**
  - CrossBow Mica2, MicaZ
    - ATmega128L 8Mhz
    - 4Kb RAM, 128Kb ROM

- **MANTIS**

- **Maté**
  - Virtual machine
Towards Resilient Sensor Networks

- **Ensure system safety at the operating system level**
  - **Applications**: we cannot write the code safely all the time
  - **Sensor node hardware**: does not easily detect application errors

- **Objectives**
  - Provide error-safe mechanism on resource-constrained sensor devices
  - Do not require any hardware supports
  - Do not require users to learn new programming language semantics
Previous Work

- **SFI (Software Fault Isolation)**
  - Requires MMU for segmentation and stack safety
  - Designed for fixed-length instruction architecture

- **Proof-carrying Code**
  - The automatic policy generator does not exist

- **Programming language approaches**
  - Cyclone, Control-C, Cuckoo
  - Users should be aware of the different usages of pointer/array
  - Requires hardware supports for stack safety

RETOS Architecture

Resilient, Expandable, and Threaded Operating System
RETOS Architecture

Resilient, Expandable, and Threaded Operating System

Code (Flash Rom)
- Applications
- Static/dynamic code checking
- Modules

User area
- User stack
- User data

Kernel area
- Module data
- Single kernel stack
- Kernel Data

User space
- Static kernel
dynamic kernel

static kernel
dynamic kernel

Scheduler
- Policy
- Priority

Variable timer
Erred apps manager
System call

Module Manager
Function Table
**RETOS Architecture**

**Resilient, Expandable, and Threaded Operating System**

- **Code (Flash Rom)**
  - Applications
  - Modules
  - Multi-threads
- **Data (RAM)**
  - User stack
  - User data
  - Module data
  - Single kernel stack
  - Kernel Data

**Scheduler**
- Policy
- Priority

**Variables**
- Timer
- Erred apps manager
- System call
Ensuring System Safety

- **Principle**
  - Dual Mode operation & static/dynamic code checking
Dual Mode Operation

- **Why dual-mode is needed?**
  - Static/dynamic code checking evaluates if the application modifies data or issues code in its allocated data
  - Preemption would invoke problems

- *(ex)* _Thread_A_ is preempted by _Thread_B_

  - Code checking considers it legal
  - _Thread_B_ has access rights to this area
  - Kernel/user separation is required
Static/Dynamic Code Checking

- Use data and code area within application itself
- Restrict direct hardware resource manipulation

---

```
#include <lo.h>
#include <syscall.h>
#include <pthread.h>

... 

// main thread */
int main ()
{
    while (1) {
        void (*func)(void);
        /* radio packet recv */
        radio_recv(PORT, &pkt);
        func = foo_func[pkt->id];

        /* do something*/
        func();
    }
}
```

```
main:
/* prologue: frame size - 44 */
.L_FrameSize_main=0x2c
.L_FrameOffset_main=0x2e
mov #(_stack-44), r1
/* prologue end (size-2) */
.L29:
/* do something */
call &func
mov #1100(6), r15
call syscall0
jmp .L5
br #_stop_progExec_
/* epilogue: frame size=44 */
add #44, r1
br #_stop_progExec_
/* epilogue end (size=4) */
/* function main size 160 (162) */
.LFE6:
.size main,.Lfe6-main__
```

---

```
mov #1792, r1
mov #6, r15
push r15
mov #0x0608,r15
cmp 12(r15),r1
jnc $+8
cmp 14(r15),r1
jnc $+6
call #4488
```
Implementation

• **RETOS implemented on TI MSP430 F1611**
  – 8MHz core clock, 10Kb RAM, 48Kb Flash ROM
  – Current version: 0.92 (May 24 2006)

• **Peripheral supports**
  – 2.4Ghz RF Module (Chipcon 2420)
  – Ultrasound, Humidity, and Light sensors

• **Applications: MPT (Multi-Party Object Tracking)**
Functionality Test (1)

- **Classify safety domain into four parts:**
  - Stack / Data safety → Stack and data area
  - Code safety → Control flow
  - Hardware safety → Immediate hardware control

### Hardware safety

- Disable interrupt
  - `DINT();`
  - Detected by **Static check**

- Flash ROM writing
  - (memory mapped registers)
  - `FCTL1 = FWKEY+WRT;`
  - Detected by **Static check**

### Stack safety

- General/Mutual recursive call
  - `void foo() { foo(); }`
  - Detected by **Dynamic check**
# Functionality Test (2)

## Data safety

<table>
<thead>
<tr>
<th>Issue</th>
<th>Code Example</th>
<th>Detection Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directly addressed pointers</td>
<td><code>int *tmp = 0x400; *tmp = 1;</code></td>
<td>Detected by <strong>Static check</strong></td>
</tr>
<tr>
<td>Illegal array indexing</td>
<td><code>/* array in heap area */ int array[10]; ... for(i = 10; i &gt; 0; i--) { array[i-100] = i; }</code></td>
<td>Detected by <strong>Dynamic check</strong></td>
</tr>
</tbody>
</table>

## Code safety

<table>
<thead>
<tr>
<th>Issue</th>
<th>Code Example</th>
<th>Detection Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directly addressed function call</td>
<td><code>void (*func)(void) = 0x1000; func();</code></td>
<td>Detected by <strong>Static check</strong></td>
</tr>
<tr>
<td>Corrupted return address (Buffer overflow)</td>
<td><code>void func() { int array[5], i; for(i = 0; i &lt; 10; i++) array[i] = 0; }</code></td>
<td>Detected by <strong>Dynamic check</strong></td>
</tr>
</tbody>
</table>
## Overhead Analysis (1)

- **Execution time running in user mode**
  - Programs, which require more memory access than complex arithmetic calculations, shows larger overhead

<table>
<thead>
<tr>
<th>Task</th>
<th>User Mode</th>
<th>System Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPT_mobile</td>
<td>386566</td>
<td>394624</td>
</tr>
<tr>
<td>MPT_backbone</td>
<td>40326</td>
<td>40508</td>
</tr>
<tr>
<td>sensing</td>
<td>1056</td>
<td>1096</td>
</tr>
<tr>
<td>r_recv</td>
<td>4704</td>
<td>5010</td>
</tr>
<tr>
<td>r_send</td>
<td>24815</td>
<td>26176</td>
</tr>
<tr>
<td>pingpong</td>
<td>1243</td>
<td>1347</td>
</tr>
<tr>
<td>surge</td>
<td>58723</td>
<td>62688</td>
</tr>
</tbody>
</table>

- **Computation based**
- **Memory access based**
Overhead Analysis (2)

- **Application code size comparison**
  - Code size increased, independent of application type
  - Applications are inherently small in size, separated from kernel

![Code size comparison diagram](image)
Overhead Analysis (3)

- **Dual mode operation**

<table>
<thead>
<tr>
<th>Event</th>
<th>Single mode (cycles)</th>
<th>Dual mode (cycles)</th>
<th>Overhead (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>system call (led toggle)</td>
<td>264</td>
<td>302</td>
<td>38</td>
</tr>
<tr>
<td>system call (radio packet send)</td>
<td>352</td>
<td>384</td>
<td>32</td>
</tr>
<tr>
<td>timer interrupt (invoked in kernel)</td>
<td>728</td>
<td>728</td>
<td>0</td>
</tr>
<tr>
<td>timer interrupt (invoked in user)</td>
<td>728</td>
<td>760</td>
<td>32</td>
</tr>
</tbody>
</table>

- **Summary**
  - Computational overhead & larger code size are inevitable to provide system safety in sensor node platform
  - However, they are considered as a fair trade-off compared to a system failure
Conclusions

- **Safety mechanism for wireless sensor networks**
  - Separate errant application from the kernel, “Kernel never die”
  - Automatically recover from serious errors
  - Useful in the real, large-scale sensor networks

- **Current status**
  - RETOS is being developed in our research group
  - Being ported to other processors (AVR)
  - “Application-Centric Networking Framework for Wireless Sensor Networks”
    - *Mobiquitous 2006, San Jose, July 17-21*
Questions
Watchdog Timer

- **Not easy to recognize and handle problems such as:**
  - Memory access beyond the application area
  - Immediate hardware control

- **Watchdog timer simply makes the system be restarted**
  - One errant application can stop all other applications
  - Disturbance of long-term operations of sensor applications
Multithread vs. Event-driven

- **Concurrency for computation based applications**
  - Run-to-completion vs. **preemptive time-sliced thread**

- **Poor software structure on event-driven model**
  - One conceptual function split into multiple functions
    - Loss of control structures / local variables

- **Simplified synchronization methods on thread model**
  - Like “atomic” operation in nesC

Limitations

- **Library codes**
  - We assume the libraries always operate safely
  - In real situation: If a user passes an invalid address to `memcpy()`?
  - Make wrapper functions that checks address parameters

- **Divide by zero**
  - Hardware multiplier/divider equipped processor: instruction level checking
  - Emulated: library level checking

- **Intentionally skips the code checking sequences**
  - User authentication on code updating would be required
## Sensor OS Comparison

<table>
<thead>
<tr>
<th></th>
<th>System Safety (kernel-user separation)</th>
<th>Light-weight dynamic reprogramming</th>
<th>Programming Model</th>
<th>Programming Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>TinyOS</td>
<td>No</td>
<td>No</td>
<td>Event-driven</td>
<td>nesC</td>
</tr>
<tr>
<td>SOS</td>
<td>No</td>
<td>Yes</td>
<td>Event-driven</td>
<td>Standard C</td>
</tr>
<tr>
<td>MANTIS</td>
<td>No</td>
<td>No</td>
<td>Preemptive Multithreading</td>
<td>Standard C</td>
</tr>
<tr>
<td>RETOS</td>
<td>Yes</td>
<td>Yes</td>
<td>Preemptive Multithreading</td>
<td>Standard C</td>
</tr>
</tbody>
</table>