EECS 571 “Principles of Real-Time Embedded Systems”

Lecture Note #10:
EMERALDS: A Small-Memory Real-Time Microkernel

ACM SOSP’99 paper by Zuberi et al.
Outline

- Motivation
- Overview of EMERALDS
- Minimizing Code Size
- Minimizing Execution Overheads
- Conclusions
Why Small memories, slow processors?

- Small-memory embedded systems used everywhere!
  - automobiles
  - home appliances
  - telecommunication devices, PDAs,…
  - factory automation and avionics

- Massive volumes (10K-10M units per annum)
  - Saving even a few dollars per unit is important:
    - cheap, low-end processors (Motorola 68K, Hitachi SH-2)
    - max. 32-64 KB SRAM, often on-chip
    - low-cost networks, e.g., Controller Area Network (CAN)
RTOS for Small-Memory Embedded Systems

- Despite restrictions, must perform increasingly complex functions
- General-purpose RTOSs (VxWorks, pSOS, QNX) too large or inefficient
- Some vendors provide smaller RTOSs (pSOS Select, RTXC, Nucleus) by carefully handcrafting code to get efficiency
RTOS Requirements for Small-Memory Embedded Systems

- Code size ~ 20 kB
- Must provide all basic OS services: process management, IPC, task synchronization, scheduling, I/O
- All aspects must be re-engineered to suit small-memory embedded systems:
  - API
  - IPC, synchronization, and other OS mechanisms
  - Task scheduling
  - Networking
EMERALDS Architecture

- Extensible Microkernel for Embedded ReAL-time Distributed Systems
Minimizing Kernel Size

- Location of resources known
  - allocation of threads on nodes
  - compile-time allocation of mailboxes => no naming services

- Memory-resident applications:
  - no disks or file systems

- Simple messages
  - e.g., sensor readings, actuator commands
  - often can directly interact with network device driver
Reducing Kernel Execution Overhead

- **Task Scheduling**: EDF/RM can "consume" 10-15% of CPU
- **Task Synchronization**: semaphore operations incur context switch overheads
- **Intertask Communication**: often exchange 1000’s of short messages, especially if OO is used
Real-Time Scheduling

- Problems with cyclic time-slice schedulers
  - Poor aperiodic response time
  - Long schedules

- Problems with common priority-driven schedulers
  - EDF: High run-time overheads
  - RM: High schedulability overheads
Scheduler Overheads

- **Run-time Overheads**: Execution time of scheduler
  - RM: static priorities, low run-time overheads
  - EDF: high run-time overheads

- **Schedulability Overhead**: $1 - U^*$
  - $U^*$ is ideal utilization attainable, assuming no run-time overheads
  - EDF: $U^* = 1$ (no schedulability overhead)
  - RM: $U^* > 0.69$ with avg. 0.88

- **Total Overhead**: Sum of these overheads
  - Combined static/dynamic (CSD) scheduler finds a balance between RM and EDF
Example of RM schedulability issue

<table>
<thead>
<tr>
<th>Task</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>P (ms)</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>20</td>
<td>30</td>
<td>50</td>
<td>100</td>
<td>130</td>
</tr>
<tr>
<td>c (ms)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
</tr>
</tbody>
</table>

$U = 0.88$; EDF schedulable, but not under RM

$T_5$ misses deadline
CSD maintains two task queues:
- Dynamic Priority (DP) scheduled by EDF
- Fixed Priority (FP) scheduled by RM

Given workload \( \{T_i : i = 1, 2, ..., n \} \) sorted by RM-priority
- Let \( r \) be smallest index such that \( T_{r+1} - T_n \) are RM-schedulable
- \( T_1 - T_r \) are in DP queue
- \( T_{r+1} - T_n \) are in FP queue
- DP is given priority over FP queue
CSD Overhead

- CSD has near zero schedulability overhead
  - Most EDF schedulable task sets can work under CSD
- Run-time overheads lower than EDF
  - \( r \)-long vs. \( n \)-long DP queue
  - FP tasks incur only RM-like overhead
- Reducing CSD overhead further
  - **split** DP queue into multiple queues
  - **shorter queues** for dynamic scheduling
  - need careful allocation, since schedulability overhead incurred **between** DP queues
CSD Performance

- Comparison of CSD-\(x\), EDF, and RM
  - 20-40\% lower overhead than EDF for 20-30 tasks
  - CSD-\(x\) improves performance, but diminishing returns
Concurrency control among tasks
May cause a large number of context switches
Typical scenario: Tx>T2 > T1 and T1 is holding lock

unblock T2
context switch C1
T2 calls acquire_sem()
priority inheritance
(bump-up T1 to T2’s)
block T2
context switch C2
T1 calls release_sem()
undo T1 priority inheritance
unblock T2
context switch C3
Eliminating Context Switch

- For each `acquire_sem(S)` call:
  - pass `S` as an extra parameter to blocking call
  - if `S` unavailable at end of call, stay blocked
  - unblock when `S` is released
  - `acquire_sem(S)` succeeds without blocking
For DP tasks, change one variable, since they are in unsorted queue

For FP tasks, must remove $T_1$ from queue and reinsert according to new priority assignment

- Solution: switch positions of $T_1$ and $T_2$
- Avoids parsing queue
- Since $T_2$ is blocked, can be put anywhere as position holder to remember $T_1$’s original position
New Semaphore Scheme Performance

- DP tasks - fewer context switches
- FP tasks - optimized PI steps
Tasks in embedded systems may need to exchange thousands of short messages per second, e.g., OO.

Traditional IPC mechanisms (e.g., mailbox-based IPC) do not work well:
- high overheads
- no "broadcast" to send to multiple receivers

For efficiency, application writers are forced to use global variables to exchange information:
- unsafe if access to global variables is not regulated.
State Messages

- Uses single-writer, multiple-reader paradigm
- **Writer**-associated state message
  - “mailbox” (SMmailbox)
    - A new message overwrites previous message
    - Reads do not consume messages
    - Reads and writes are **non-blocking, synchronization-free**
- Read and write operations through **user-level** macros
  - Much less overhead than traditional mailboxes
  - A tool generates customized macros for each state message
Problem with global variables: a reader may read a half-written message as there is no synchronization

Solution: $N$-deep circular message buffer for each state message
- Pointer is updated atomically after write
- if writer has period 1 ms and reader 5 ms, then $N=6$ suffices

New Problem: $N$ may need to be in the 100’s
State Messages in EMERALDS

- Writers and “normal” readers use user-level macros
- Slow readers use atomic read system call
- $N$ depends only on faster readers (saves memory)

<table>
<thead>
<tr>
<th></th>
<th>State Messages</th>
<th>Mailboxes</th>
</tr>
</thead>
<tbody>
<tr>
<td>send (8 bytes)</td>
<td>2.4 us</td>
<td>16.0 us</td>
</tr>
<tr>
<td>receive (8 bytes)</td>
<td>2.0 us</td>
<td>7.6 us</td>
</tr>
<tr>
<td>receive_slow (8 bytes)</td>
<td>4.4 us</td>
<td></td>
</tr>
</tbody>
</table>
Memory Protection

- Needed for fault-tolerance, isolating SW bugs
- Embedded tasks have small memory footprints
  - use only 1 or 2 page tables from lowest level of hierarchy
  - use common upper-level tables to conserve kernel memory
- Map kernel into all task address spaces
  - Minimize user-kernel copying as task data and pointers accessible to kernel
  - Reduce system call overheads to a little more than for function calls
OSEK OS standard consists of:
- **API**: system call interface
- **Internal OS algorithms**: scheduling and semaphores

OSEK Communication standard (COMM) is based on CAN

Developed an OSEK-compliant version of EMERALDS for Hitachi SH-2 microprocessor
EMERALDS-OSEK (cont’d)

- **Features**
  - Optimized *context switching* for basic and extended tasks
  - Optimized *RAM usage*
- Developed OSEK-COMM over CAN for EMEMRALDS-OSEK
- Hitachi’s application development and evaluation: collision-avoidance and adaptive cruise control systems
Conclusions

- Small, low-cost embedded systems place stringent constraints on OS efficiency and size
- EMERALDS achieves good performance by redesigning basic services for such embedded systems
  - Scheduling overhead reduced 20-40%
  - Semaphore overheads reduced 15-25%
  - Messaging passing overheads 1/4 to 1/5 that of mailboxes
  - Complete code ~ 13 kB
Extensions

- Implemented on Motorola 68040
- Ported to 68332, PPC, x86, and strong ARM
- Also investigated networking issues: devicenet, wireless LANs, rt-ethernet, TCP and UDP/IP
- OS-dependent and independent development tools
- Energy-Aware EMERALDS
  - extend to support energy saving hardware (DVS, sprint & halt)
  - Energy-aware storage systems (memory and disks)
  - Energy-aware Quality of Service (EQoS)
  - Applications to info appliances and home networks
Related Publications

- RTAS ‘96 - original EMERALDS
- RTAS ‘97 - semaphore optimizations
- NOSSDAV ‘98 - protocol processing optimizations
- SAE ’99 - EMERALDS-OSEK
- SOSP ‘99 - EMERALDS with re-designed services
- RTSS’00 – Energy-aware CSD
- IEEE-TSE’00 –complete version with schedulability analysis
- SOSP’01- Exploitation of DVS
- ACM TECS (pending) - EQoS
- UNSENIX’03, PACS’04: power-aware memory
- SOSP’05: high-performance, low-power disk I/O
- USENIX’02 – totally non-blocking IPC

URL: http://kabru.eecs.umich.edu/rtos