Real-Time Operating Systems: Principles and a Case Study

Kang G. Shin
Real-Time Computing Laboratory
EECS Department
University of Michigan
URL: http://www.eecs.umich.edu/~kgshin

Outline

- Generic Aspects of RTOSs
  - Requirements
  - Classification
  - Approaches
- Case Study: a Small Memory RTOS, EMERALDS
  - Motivation
  - Overview of EMERALDS
  - Minimizing Code Size
  - Minimizing Execution Overheads
- Conclusions

Real-Time Operating Systems

- Four main functions
  - Process management and synchronization
  - Memory management
  - IPC
  - I/O
- Must also support predictability and real-time constraints

Classification of RTOSs

- Small proprietary (homegrown and commercial) kernels
- RT extensions to UNIX and others
- Research RT kernels

Proprietary Kernels

Small and fast commercial RTOSs: QNX, pSOS, VxWorks, Nucleus, ERCOS, EMERALDS, Windows CE,...

- Fast context switch and interrupt response
- Small in size
- No virtual memory and can lock code & data in memory
- Multitasking and IPC via mailboxes, events, signals, and semaphores

Proprietary Kernels (cont’d)

- How to support real-time constraints
  - Bounded primitive exec time
  - real-time clock
  - priority scheduling
  - special alarms and timeouts
- Standardization via POSIX RT extensions

RT Extensions

RT-UNIX,RT-LINUX, RT-MACH, RT-POSIX

- Slower, less predictable, but more functions and better development envs.
- RT-POSIX: timers, priority scheduling, rt files, semaphores, IPC, async event notification, process
- mem locking, threads, async and sync I/O.
- Problems: coarse timers, system interface and implementation,long interrupt latency, FIFO queues,no locking pages in memory, no predictable IPC

Research RTOSs

- Support rt sched algorithms and timing analysis
- RT sync primitives, e.g., priority ceiling.
- Predictability over avg perf
- Support for fault-tolerance and I/O
- Examples: Spring, Mars, HARTOS, MARUTI, ARTS, CHAOS, EMERALDS

Small memories, slow processors

- Small-memory embedded systems used everywhere:
  - automobiles
  - factory automation and avionics
  - home appliances
  - telecommunication devices, PDAs,…
- Massive volumes (10K-10M units) ⇒ Saving even a few dollars per unit 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

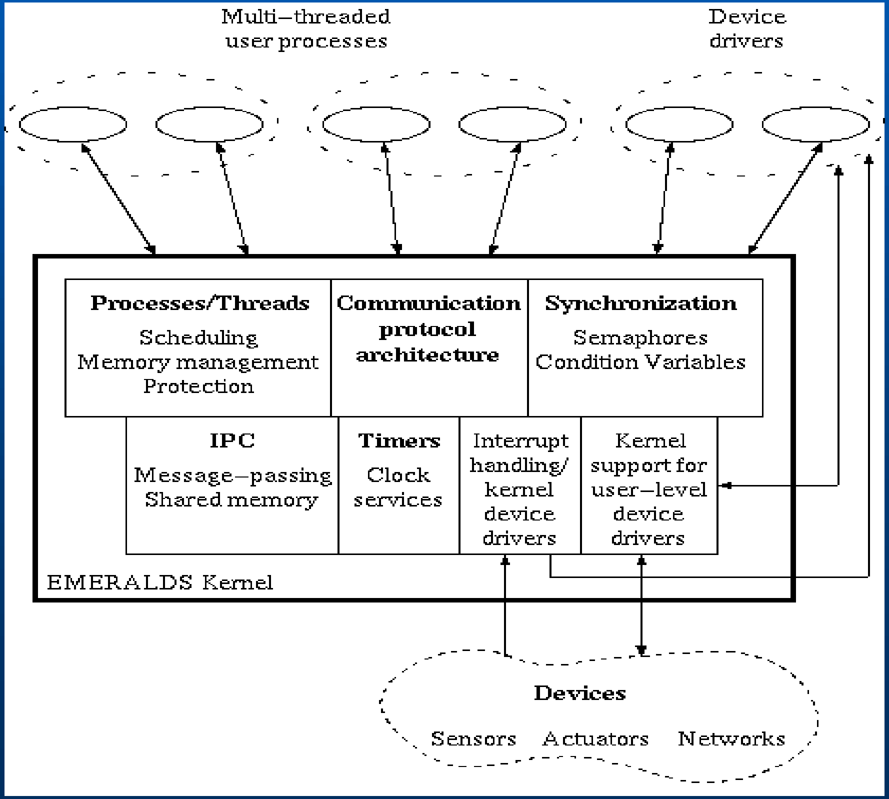
- Despite restrictions, must perform increasingly complex functions
- *General-purpose* RTOSs (VxWorks, pSOS, QNX) too large or inefficient
- Some vendors provide smaller RTOSs (pSOS Select, RTXC, Nucleus) by carefully *handcrafting* code to get efficiency

RTOS Requirements for Small-Memory Embedded Systems

- Code size ~ 10 kB
- Must provide all basic OS services: 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, etc., so 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

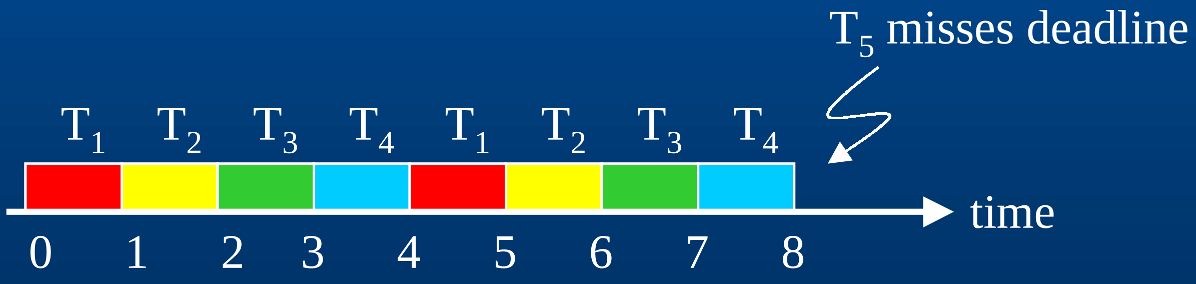
- Run-time Overheads: Execution time of scheduler
  - RM: static priorities, low overheads
  - EDF: high run-time overheads
- Schedulability Overhead: 1 - U^*
  - U^* is ideal utilization attainable, assuming no run-time overheads
  - EDF has $U^* = 1$ (no schedulability overhead)
  - RM has $U^* > 0.7$, avg. 0.88
- Total Overhead: Sum of these overheads
  - Combined static/dynamic (CSD) scheduler finds a balance between RM and EDF

Schedulability Overhead Illustration

- Example of RM schedulability issue

Task	1	2	3	4	5	6	7	8	9	10
P (ms)	4	5	6	7	8	20	30	50	100	130
c (ms)	1	1	1	1	0.5	0.5	0.5	0.5	0.5	0.5

- *U* = 0.88; EDF schedulable, but not under RM



Combined Static and Dynamic Scheduling

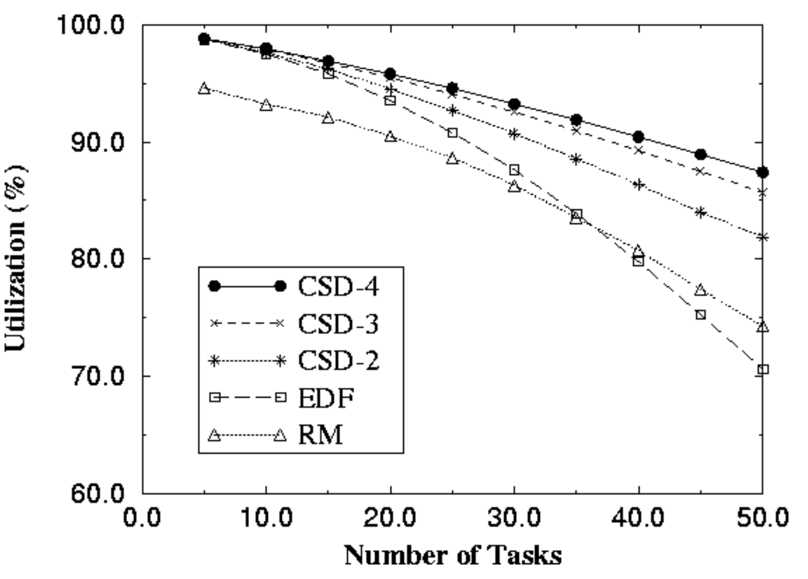
- CSD maintains two task queues:
  - Dynamic Priority (DP) scheduled by EDF
  - Fixed Priority (FP) scheduled by RM
- Given workload { T_i : $i = 1,2,\ldots,n$ } sorted by RM-priority
  - Let r be smallest index such that T_{r+1} - T_n are RM-schedulable
  - T_1 - T_r are in DP queue
  - T_{r+1} - T_n are in FP queue
  - DP has 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

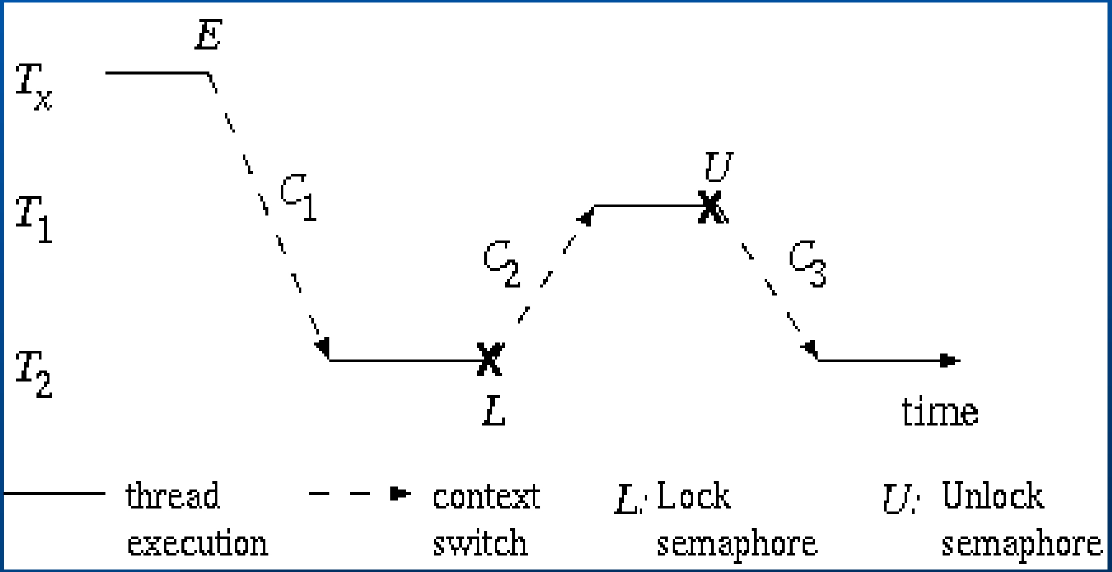
CSD Performance

- Comparison of CSD-*x*, EDF, and RM
  - 20-40% lower overhead than EDF for 20-30 tasks
  - CSD-*x* improves performance, but diminishing returns



Efficient Semaphores

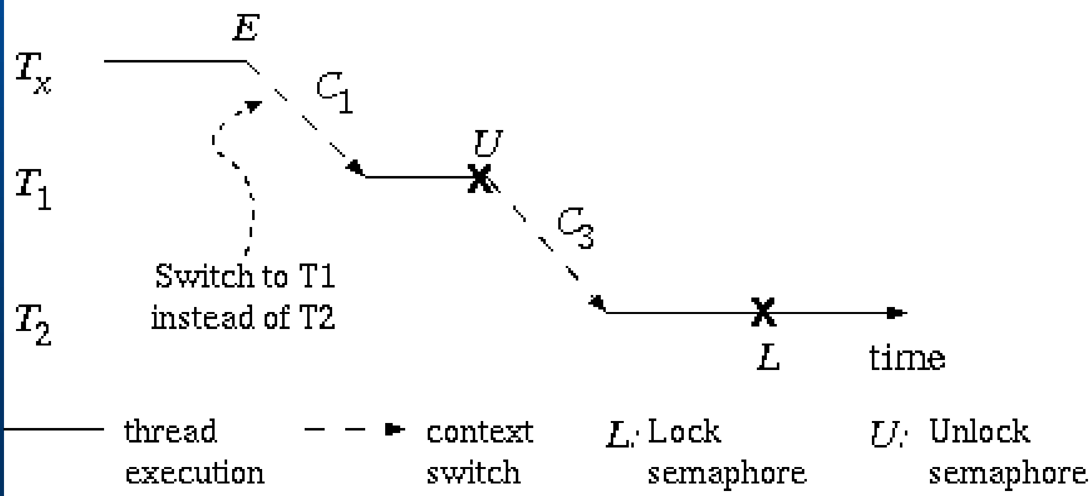
- Concurrency control among tasks
- May cause large number of context switches
- Typical scenario: T2 > Tx > T1 & T1 is holding lock



unblock T_2
context switch C_1
T_2 calls acquire_sem()
priority inheritance
(bump-up T_1)
block T_2
context switch C_2
T_1 calls release_sem()
undo T_1 priority
inheritance
unblock T_2
context switch C_3

Eliminating Context Switch

- For each `acquire_sem(`*`S`*`)` call:
  - pass *S* as 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



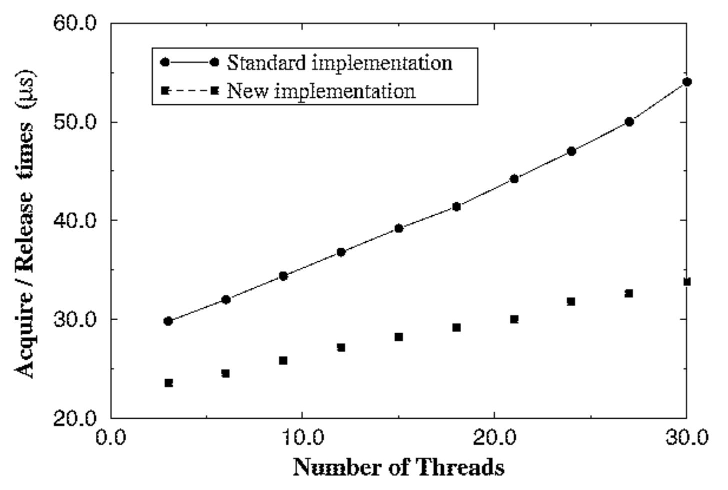
Optimize Priority Inheritance Steps

- 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 priority
  - 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

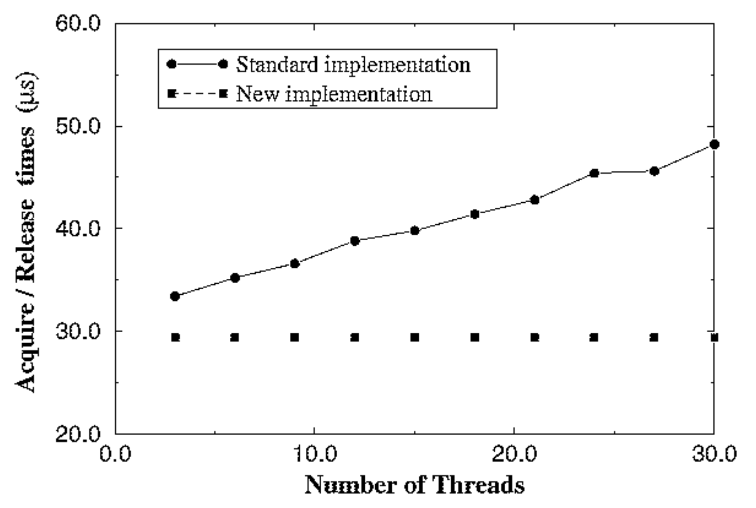
New Semaphore Scheme Performance

- DP tasks - fewer context switches
- FP tasks - reflects optimized PI steps

FP Tasks



DP Tasks



Message Passing

- Tasks in embedded systems may need to exchange thousands of short messages per second
- 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 forced to use global variables to exchange information
  - Not safe if access to global variable unregulated

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

State Messages

- 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)

	State Messages	Mailboxes
send (8 bytes)	2.4 us	16.0 us
receive (8 bytes)	2.0 us	7.6 us
receive_slow (8 bytes)	4.4 us	

Memory Protection

- Needed for fault-tolerance, isolating bugs
- Embedded tasks have small memory footprints
  - can use just 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 little more than for function calls

EMERALDS-OSEK

- 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 great constraints on operating system efficiency and size
- EMERALDS achieves good performance by re-designing 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

Current State and Future Directions

- Implemented on Motorola 68040
- Partial ports to 68332, PPC, and x86
- Investigating networking issues: devicenet, real-time Ethernet, UDP/IP
- OS-dependent and independent development tools
- Energy-Aware EMERALDS
  - extend to support energy saving hardware (DVS, sprint & halt)
  - 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 (to appear) – Exploitation of DVS

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