EECS 571 “Principles of Real-Time Embedded Systems”

Lecture Note #10: More on Scheduling and Introduction of Real-Time OS

Kang G. Shin
EECS Department
University of Michigan
Mode Changes

- Changes in mission phase or processor failures
- Easy to deal with if no critical section is involved
- A task may be added if
  - Addition preserves RM-schedulability
  - Raise ceilings before addition if it increases priority ceilings of any semaphore.
- Rules of changing ceiling: indivisible action when a semaphore is unlocked.
Quick Recovery from Failure

- Useful for static schedules: sufficient reserve capacity and fast failure-response mechanism.

- Incorporate *ghost* clones into schedule which can be activated when a processor on which a *primary* clone was scheduled crashes.

- 2 types of tasks affected by a processor failure
  - Running at time of failure \( \Rightarrow \) FEC
  - Scheduled to run in future on the failed processor

- Assume there is a non-fault-tolerant algorithm for given assignment and scheduling
Fault-Tolerant Scheduling

- If up to $f$ processor failures are to be withstood, each task must have $f$ ghost clones.

- A primary and a ghost clone of the same task cannot be allocated to the same processor.

- A ghost schedule and a primary schedule are said to form a *feasible pair* if deadlines continue to be met even if primaries are shifted right by the time needed to execute ghosts.
More on FT Scheduling

- Ghosts are conditionally transparent. They can overlap primary clones or other ghost clones in the schedule, subject to certain correctness restrictions.

- When a ghost clone is activated, the schedule is moved appropriately.

- A ghost clone is never moved.

Kang Shin (kgshin@eecs.umich.edu)
Correctness Restrictions

- Primary clones moved by a ghost activation should still meet their deadlines.

- Two ghosts may overlap so long as at most one of the two ghosts is activated.
Real-Time Operating Systems

• 4 main functions:
  o Process management and synchronization
  o Memory management
  o IPC
  o I/O

• Must support predictability and real-time constraints

• 3 types of RTOS:
  o small proprietary (homegrown and commercial) kernels
  o RT extensions to UNIX and others
  o research kernels
Proprietary Kernels

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

- fast context switch and fast interrupt response
- small in size
- No virtual memory and can lock code & data in memory
- Multitasking and IPC via mailboxes, events, signals, and semaphores
- How to support real-time constraints
  - Bounded primitive exec time
  - real-time clock
  - priority scheduling
  - special alarms and timeouts
- Standardization via POSIX RT extensions

Kang Shin (kgshin@eecs.umich.edu)
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 performance
- Support for fault-tolerance and I/O
- Examples: Spring, Mars, HARTOS, MARUTI, ARTS, CHAOS, EMERALDS
RT-Mach: Predictable Task Execution

- Tasks = RT-Mach threads

- Bounded blocking delays

- Real-time scheduling of threads
  - Hard periodic: $p_i$, worst-case exec time $e_i$, deadline $d_i$.
  - Hard aperiodic: $a_i$, $e_i$, $d_i$.
  - Soft periodic or aperiodic: abort times can be specified.
Thread Scheduling

Scheduling Policies:
- Mach: RR, FP
- RT-Mach: RM, RM/DS, RM/SS, RM/PS

Each thread can pick its own policy and go in the corresponding queue.

Capacity Reserves:
- CPU cycles required by hard RT tasks first reserved
- Remaining cycles used for soft RT tasks

Kang Shin (kgshin@eecs.umich.edu)
Thread Synchronization

- Priority inversion and solutions

- Implementation
  - Scenario: One thread in CS and many others waiting (in queue) to get in
  - Issues: Allow preemption in CS or not? Which thread to pick next from the queue?

- Preemption in CS: a thread inside CS may be non-preemptable, preemptable, and restartable
Synchronization Policies

- Kernelized Monitor (KM): non-preemptable mode
- Basic priority (BP): preemptable mode and priority scheduling
- Basic Priority Inheritance (BPI): BP + Priority Inheritance
- Priority Ceiling Protocol (PCP)
  - Priority ceiling of lock = priority of highest-priority thread that may lock this lock
  - Those threads execute which are associated with lock with highest-priority ceiling
  - avoids deadlock
- Restartable Critical Section (RCS): Restartable mode
Schedulability Analysis

**KM:** Like EDF, except threads preemptable only when *not* in CS:

\[ \sum_{j=1}^{n} \frac{e_j + e_s}{p_j} \leq 1 \]

**Priority Inheritance:** Like RM, except lower priority thread can block higher-priority threads

\[ \forall i \quad \frac{B_i}{p_i} + \sum_{j=1}^{i} \frac{e_j}{p_j} \leq i \left(2^{1/i} - 1\right) \]
Which Policy is Best?

- KM: Good only if very short CS
- BP: Not good — priority inversion
- PCP: Use if possibility of deadlock
- RCS: Use if high-priority thread cannot wait for CS and Restart costs are low.
- BPI: Simple and fast
  Use if no deadlock conditions and high-priority threads can wait for CS