RM Implementation Challenges:
Static vs Dynamic Scheduling and Pitfalls

ECEN 5653/4653
Lecture 6
Overview

Digital Media Storage and Networking (Next Week)
- Fundamentals of Storage and Networking
- Linux Kernel Drivers
- Scalable RAID Storage Systems
- Advanced Erasure Codes for Data Protection

Back to Real-Time Scheduling (Tonight)
- Thread Scheduling and Resource Synchronization
  - Deadlock and Unbounded Priority Inversion
  - Real-Time Resource Management
  - RT Shared Resource Management
- RM vs EDF
  - Dynamic Feasibility Tests
  - Disadvantages of Fixed Priority RM for Soft Real-Time
  - Point by Point Comparison of EDF Advantages (In Buttazzo Paper)
- Relative and Absolute Time
Broadcast, On-Demand and Viral Videos …

- Difference Between Viral Video (Download Program Streams) and True Transport Streaming for On-Demand

- An Excuse to Watch some YouTube: Peter Sellers Viral Video on Broadcast
  - Pink Panther Strikes Again – Insane Inspector Dreyfus
    - National Broadcast Disruption – Could it Happen?
    - Preventing Pirated Content
    - CAS, DRM, Watermarking, Fingerprinting, Authentication
    - Digital Content Revolution
  - More Gratuitous Viral Video Just for fun…
    - Pink Panther Strikes Again – Dentist Scene
    - Pink Panther Strikes Again – Dog Scene
    - Peter Sellers – Dr. Strangelove

- True On-Demand Real-Time Streaming Requires Broadcast-like Quality, but Time-Shifted Content Library

- Broadcast is Coordinated within a Leap-Second to UTC/GPS time
  - Allows for Ad-insertion
  - Clean cuts between content
  - Accurate guide data
  - News feeds

© Sam Siewert
DVB - Broadcast Content

- Broadcast Must be Coordinated on the Networks
  - Within a Leap-second, to base date, with real-time streaming and clear cuts between programming
  - Often employs SDI and HD-SDI
    - Synchronous Digital Interface – SMPTE 259M, 270 Mbit/s, 360 Mbit/s, 143 Mbit/s, and 177 Mbit/s, 480i, 576i (NTSC, PAL)
    - High Definition SDI – SMPTE 292M, 1.485 Gbit/s, and 1.485/1.001 Gbit/s, 720p, 1080i
    - Dual Link HD-SDI and 3G-SDI – SMPTE 372M, SMPTE 424M, 2.970 Gbit/s, and 2.970/1.001 Gbit/s, 1080p
  - DVB Transport over ATSC (8VSB) and Digital Cable (QAM) - Not Internet Yet
  - Future of Broadcast? – Will everything go on-demand, time-shifted and location shifted?

- We need Real-time Networking, Storage, and Computing systems

- What are the fundamentals?
  - Scheduling: Rate-Montonic, Dynamic QoS
  - Transport: Isochronal networks, Constant Bit-rate transport
  - Encoding: Bit-rate control and grooming on channels (carriers)
Rate Monotonic or EDF?

- Liu & Layland Rate Montonic - Fixed Priority Policy
  - Feasibility Test - Sufficient RM LUB
  - Most Hard RT Systems Have Static Services Sets

\[ U = \sum_{i=1}^{m} \left( \frac{1}{C_i/T_i} \right) \leq m \left( 2^m - 1 \right), U \lim m \to \infty = \ln(2) \approx 0.69 \]

- EDF (Deadline Driven) – Dynamic Priority Policy
  - Sufficient Feasibility Test Based on Hyper-Period (Product of All Periods)

\[ \forall \text{tasks} \in \{1...m\}, T_{\text{hyperperiod}} = 0...(T_1T_2...T_{m-1}T_m) \]

\[ (T_1T_2...T_{m-1}T_m) = \frac{(T_1T_2...T_{m-1}T_m)}{T_1} C_1 + ... + \frac{(T_1T_2...T_{m-1}T_m)}{T_m} C_m \]

\[ \therefore \sum_{i=1}^{m} \left( \frac{C_i}{T_i} \right) \leq 1 \]
Notes on Dynamic Admission

- Off-Line for Fixed Set of Services
  - Transition from Off-Line to On-Line RT Services
  - Service Set Can’t Change – No Dynamic Admission

- On-Line Admission of Services
  - Feasibility Test Must be Run On-Line to Re-Test Feasibility of New Services Before They Go On-Line
  - Feasibility Test Itself is a Service that Must Be Admitted by Off-Line Test or Run on Dedicated Hardware
Hard RT and RM Pessimism

- Hard RT Pessimism Requires Resource Margin for Safety
  - $C = WCET$
    - Longest Path for Code
    - Path is Executed Least Efficiently
    - WCET Is Worse than 3-Sigma Deviation From Norm (Rarely Happens)
  - $T = $Worst Case Inter-arrival Rate (Period Transform Required)
  - Critical Instant - Assumes that Service Requests May All Come At Once
  - RM LUB Sufficient Test
    - RM LUB is Based Upon Service Periods That Are Not Necessarily Harmonic, So Partial Interference is Over-Accounted When They Are
    - Feasible Service Sets May Be Rejected, Although Unsafe Will Never Be Admitted
    - Can Use Alternative Scheduling Point or Completion Test RM N&S Feasibility Tests
Pitfalls Applying RM Theory

Problem: Period is not Constant

- Service Releases have Period Jitter, are Sporadic, or A-periodic
- **Possible Solution:** Period Transform
  - Assume Service Period Based on Worst Case Inter-arrival Rate (Highest Known Frequency of Requests)
  - When Frequency is Lower than Worst Case, Margin Available for Slack Stealers
  - May not Handle Truly A-periodic Services (Worst Case Frequency Unknown)

Problem: C is not Deterministic

- Service Execution has Jitter due to Processor Architecture and Data Driven Algorithms
- **Possible Solution:** WCET
  - Assume Worst Case Execution Time
    - Longest Path Possible for Algorithms
    - Worst Efficiency Possible Executing that Path (e.g. all cache accesses miss)
Pitfalls Applying RM Theory

Problem: RM Policy Does Not Encode Importance
- The Longest Period Service May not Be the Service We Want to Fail in an Overload Situation
- **Possible Solution**: Period Transform
  - Assume High Importance Service with Long Period has a Shorter Period than it Actually Does to Artificially Increase Priority
    - Increases System CPU Resource Margin
    - Overload Failures Result in Less Important Services Missing Deadlines

Problem: Deadline Does Not Equal Period
- Deadline may be Less than Period when Response I/O Latency is Accounted for (e.g. if Service Response is Transported on a Network)
- Deadline may be Greater than Period if Service Release Processing is Allowed to Overlap (e.g. 2 or more service requests within T)
- **Possible Solution**: Deadline Monotonic Policy and Feasibility Test
RM Necessary and Sufficient Feasibility

- RM LUB is Only Sufficient
  - Order N
  - Sufficient, but Not Necessary and Sufficient
  - A Sufficient Test Will Never Incorrectly Pass an Infeasible Service Set, But Will Fail Some Feasible Ones
  - An N&S Test is Exact – Will Not Pass Infeasible Sets Nor Will it Fail a Feasible Set

- N&S Feasibility Tests (Order N^3) for N Services
  - Based Upon Lehoczky, Shah, Ding Theorem
    - If Services Can Be Scheduled over Longest Period, Then Set is Feasible
    - Best To Look over LCM (Least Common Multiple) of All Periods
  - N&S Test Concept: Iterate Over Subsets of Services for Each Longest Period for All Services
  - Encodes Process of Hand Drawing a Service Timing Diagram With Preemption According to RM Policy
    - Scheduling Point (Lehoczky, Shah, and Ding)
    - Completion Test
  - See RTECS Text for Details on Algorithms

\[
U = \sum_{i=1}^{m} \frac{1}{(C_i / T_i)} \leq m \left(2^m - 1\right)
\]
Pitfalls Applying RM Theory

Problem: Service Requires Resources In Addition to CPU
  – A Service may Block

  – Blocking on a Resource When by RM Policy the Service Should beExecuting Causes Priority Inversion - I.e. a Service With LowerPriority Runs in Place of the Service Which is Expected to Run

  – Possible Solution: Eliminate or Model Worst Case Blocking Time in Response Time (I.e. Inversion Duration is Bounded)
    - CPU is Yielded While Blocking, so Blocking Time Creates Margin, but Must Be Accounted for in Response Time
    - Blocking Could Be Eliminated by Polling in Some Circumstances if Additional Resource is Optional for Progress
Service Response Timeline
(With Intermediate Blocking)

Response Time = Time_{Actuation} – Time_{Sensed}
(From Event to Response)
Blocking

- Blocking is Evil!

- Caused by Need for Shared Resource that is Unavailable Despite Availability of CPU

- Ideally Eliminate Potential for Blocking During Service Execution

- If Elimination Impossible, Bounded Blocking OK
  - Deterministic Upper Bound on Blocking Time
  - Lump Bounded Blocking Time Into Response Timeline
  - Creates Slack

- Unbounded Blocking is Possible and the Real Problem
Resource Deadlock (Circular Wait)

- A is holding X and would like Y
- B is holding Y and would like X
- How is this resolved?
- A and B could Block Indefinitely
- Each could release X or Y and try again?
  - Can Result in Livelock
  - They Release, A grabs X, B grabs Y, Deadlock, Detection, Release, A grabs X, B grabs Y …
  - Random Back-Off Times (Possible Solution)
- Circular Wait Can Evolve over Complex Sets of Tasks and Resources (Hard to Detect or Prevent)
- This is Unbounded Blocking
- Detection Most Often with Watch-Dog and Sanity Monitors
Deadlock NPTL Demo

Guaranteed Deadlock (default)

[root@localhost example-sync]# ./deadlock
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
Thread 2 spawned
rsrcaCmnt=0, rsrbcCnt=0
will try to join CS threads unless they deadlock
THREAD 1 grabbing resources
THREAD 2 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got B, trying for A
<Ctrl-C>
[root@localhost example-sync]#
Deadlock NPTL Demo

Guaranteed Safe

[root@localhost example-sync]# ./deadlock safe
Creating thread 1
Thread 1 spawned
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Creating thread 2
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=1
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 1 got B, trying for A
THREAD 2 got B and A
THREAD 2 done
All done
[root@localhost example-sync]#
Shared Mutually Exclusive Access Resource

- Shared Mutex Resources Require Protection from Unintentional Non-Mutex Access
- E.g. Shared Memory State that Can’t Be Updated Atomically
  - Position[] = {x, y, z}
  - To Update Position Requires More Than One Write Instruction
  - What Happens on Interrupt Between Update of X and Y?
  - What if That Interrupt Releases a Service that Reads Position?
  - Data Corruption!
Shared Mutually Exclusive Access Resource

Figure 1. Priority inversion

- semTake(semCS);
- H blocked here
- Use_shared_resource();
- L preempted here by M
- semGive(semCS);
- unrelated_code();
- M running here

H has priority > M and M has priority > L
H and L are blocked as long as M continues to run
By priority policy H should run except it can't obtain
mutual exclusion resource

Priority inversion and deadlock can kill "five nines"
Both priority inversion and deadlock in software often require recovery through a hardware watchdog timer and a reboot. Rebooting means loss of service time. A "five nines" system (one with 99.999% uptime) can't be out of service for more than five minutes per year, so rebooting to recover directly affects a system's ability to meet this availability measure. Quicker recovery methods can help, but avoiding the necessary conditions for deadlock and priority inversion is a much better design approach.

Figure 1 shows a scenario that can cause an unbounded inversion. Here we have a high-priority service (H), a middle-priority service (M), and a low-priority service (L). The necessary condition for priority inversion are:

- Three or more services in the system sharing a CPU
- Two services with priority H and L accessing a shared resource. The resource must have mutually exclusive access to avoid data corruption or contention.
- pri(H) > pri(M1) > pri(M2) > ... > pri(Mn) > pri(L). Thus, all M services with priorities between H and L are not involved in the shared resource, but can preempt L.
Priority Inversion

Problem: Service Using Shared Resource May Suffer Unbounded Priority Inversion

- Mutex Protection of a Resource May Result in Unbounded Inversion

- **3 Necessary Conditions for Unbounded Inversion**
  - Three or More Services With Unique Priority in the System - High, Medium, Low Priority Sets of Services
  - At Least Two Services of Different Priority Share a Resource with Mutex Protection - One or More High and One or More Low Involved
  - One or More Services Not Involved in the Mutex Has Priority Between the Two Involved in the Mutex

- What Happens?
  - Low Priority Service Enters Mutex and High Priority Blocks on Mutex
  - The Medium Priority Services Not Involved in the Mutex Can Interfere with the Low Priority Service for An Indeterminate Amount of Time

- **Possible Solution:** Priority Inheritance or Priority Ceiling
Priority Inversion NPTL Demo

First, 3 Threads with No CS

[root@localhost example-sync]# ./pthread3ok 10000
interference time = 10000 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Creating thread 0
Creating thread 1
High prio 1 thread spawned at 1203339106 sec, 128498 nsec
Creating thread 2
Middle prio 2 thread spawned at 1203339106 sec, 128549 nsec
Creating thread 3
Low prio 3 thread spawned at 1203339106 sec, 128604 nsec
**** 1 idle stopping at 1203339106 sec, 168299 nsec
**** 2 idle stopping at 1203339106 sec, 172718 nsec
Start services thread spawned
will join service threads
**** 3 idle stopping at 1203339106 sec, 188757 nsec
LOW PRIOR done
MID PRIOR done
HIGH PRIOR done
START SERVICE done
All done
[root@localhost example-sync]#
Priority Inversion NPTL Demo

3 Threads Prio H,L in CS and Prio M Not

[root@localhost example-sync]# ./pthread3 10000
interference time = 10000 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Creating thread 0
Creating thread 3
Low prio 3 thread spawned at 1203339142 sec, 688239 nsec
Start services thread spawned
will join service threads
Creating thread 2
Middle prio 2 thread spawned at 1203339143 sec, 688496 nsec
Creating thread 1, CScnt=1
High prio 1 thread spawned at 1203339143 sec, 688567 nsec
**** 2 idle NO SEM stopping at 1203339143 sec, 726988 nsec
**** 3 idle stopping at 1203339144 sec, 856797 nsec
LOW PRIO done
MID PRIO done
**** 1 idle stopping at 1203339146 sec, 968860 nsec
HIGH PRIO done
START SERVICE done
All done
[root@localhost example-sync]#
Priority Inversion

- Well Understood, Appreciated, and Many Solutions for Unbounded Inversion in RTOS Community
- Linux Was a Different Story at First…
  - Early Claims Made that This Could Never Happen in Linux (Perhaps true before NPTL and POSIX SCHED_FIFO)
  - Debates On the Web
  - Summary of Linux Status
  - Linux Robust Mutex / FUSYN Project
  - Carrier Grade Linux
Priority Inheritance

When Higher Priority Task is Blocked on Mutex and Lower Priority Task is in Mutex,
Higher Prio Loans Its Prio to the Lower for Scope of Mutex

Can Chain
- Even Higher Prio Task Also Blocks and Again Loans Even Higher Prio
- As More Block More Temporary Prio Transfers Occur
- All Prios Must Ultimately Be Restored
- What is the Limit of Chaining?

What Happens if Mutexes are Nested?
Priority Ceiling Protocol

- Precise Version of Priority Inheritance
- System Ceiling for ALL Lockers Known by OS
- **CS Entry Precondition**: Holders of System Ceiling or Higher Can Access CS (Safe From Unbounded Inversion)
- Thread in CS will Receive Blocked Task Priority - Inheritance
Priority Ceiling Emulation Protocol

“Highest Locker”

Instead of Chaining, Simply Set Prio of Task in Mutex to Highest Immediately When There is an Inversion

Could be highest Prio in the System
  – May Over-amplify
  – Simple to Implement

More Precisely Can Be highest Prio of Those Tasks Actually Involved in Mutex
RM vs EDF

Read Rate Monotonic vs EDF: Judgement Day by Giorgio Buttazzo
- Implementation Complexity
- Runtime Overhead
- Feasibility Analysis
- Permanent and Transient Overload Response
- Response Jitter and Latency
- RT Resource Sharing Issues
- Aperiodic Task Handling
Notes On Relative and Absolute Time

- POSIX / Linux Supports Both
  - Relative Time Based on Interval Timer Hardware
    - x86 PIT
  - Absolute Time Based on Battery-Backed Clock or NTP (Network Time Protocol)
    - RT Clock
      - Julian/Gregorian Date and TOD
      - Battery Backed or Network Time Protocol
POSIX Relative Timer Services

- Programmable Interval Timer (Hardware Device)
  - Base Frequency of Oscillator
  - Jiffy is basic counting frequency of PIT
  - On x86 Architecture
  - Core includes 0.42 μsec Jiffy
  - Approximately 2381 Jiffies per 1 millisecond Tick
  - sysClkRateSet(1000) for 1 millisecond Tick
  - Tick is defined as a sub-frequency of Jiffy
  - PIT Count-down/up Comparison

- Tick = N Jiffies
- Interrupt is Asserted at Tick
- PIT Count-down is reset
- Timeouts
- Knowledge accurate to +/- 1 Tick
- Want to ensure timeouts are at least as long as specified
- Assume TO will be at least value specified + 1 Tick
- Delays (sleep, nanosleep)
- Yield with alarm to wake up thread
- Wake-up implemented as binary semaphore or signal
Linux NTP and RT Clocks

- **Linux Time** - Secs Since Midnight Jan 1, 1970
- **ATSC Time** - Secs Since Midnight Jan 6, 1980
- Epoch offsets between Jan 1, 1970 and Jan 6, 1980 are 3657 days or 315964800 secs

```c
unsigned int secondsSinceUnixEpoch(unsigned int year, unsigned int day, unsigned int seconds)
{
    unsigned int dt;

    dt = seconds + // elapsed seconds
        (day*SECONDS_IN_DAY) + // seconds elapsed since start of year
        ((year-1970)*SECONDS_IN_NON_LEAP_YEAR) + // seconds since base date of Jan 1, 1970 at midnight
        ((year-1969)/4)*SECONDS_IN_DAY - // leap years are divisible by 4 and add one day
        ((year-1901)/100)*SECONDS_IN_DAY + // century years are divisible by 4, but are not leap years
        ((year-1900+199)/400)*SECONDS_IN_DAY; // every 4th century year is a leap year

    return dt;
}
```
/* Notes on NTP setup:
* Make sure /etc/sysconfig/clock has UTC=true and run the NTP daemon to update the Linux system time
* and then call secondsSinceUnixEpoch to adjust UTC time that is NTP-synchronized on this host. This assumes that
* /etc/localtime is set for the right timezone and daylight savings time options.
*
* Here's some notes on how to set up NTP daemon and /etc/localtime on a Linux host:
*  
*  1) Edit /etc/ntp.conf and replace the first two fedora pool servers with better time sources like NIST time-a.timefreq.bldrdoc.gov
*     (http://tf.nist.gov/service/time-servers.html).
*     The /etc/ntp.conf should look like:
*     server time-a.timefreq.bldrdoc.gov
*     server time-c.timefreq.bldrdoc.gov
*     server 2.fedora.pool.ntp.org
*  2) Start up NTP daemon with "service ntpd start" and verify that you have good time server with "ntpq -p".
*  3) Make sure NTPD is set up to just start up on reboot with "chkconfig --levels 2345 ntpd on"
*  4) Synch your system time as follows:
*     [root]# service ntpd stop
*     [root]# ntpdate -u time-b.timefreq
*     [root]# ntpdate -u time-b.timefreq
*     [root]# ntpdate -u time-b.timefreq
*     [root]# service ntpd start
*  5) Copy the right TZ and DST options file int
*     /etc/localtime from /usr/share/zoneinfo
*/