Aislamiento, acceso al reloj y comunicaciones en aplicaciones MaRTE OS sobre Linux
MaRTE OS

• Main features
  ▫ follows the Minimal Real-Time POSIX.13 subset
  ▫ single address space shared by kernel and application
  ▫ support for concurrent Ada and C applications
    • implements *Ada Real-Time Systems Annex*

• Supported architectures

<table>
<thead>
<tr>
<th>x86</th>
<th>ARM</th>
<th>Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Bare-machine  &lt;br&gt; • XtratuM</td>
<td>• Raspberry pi  &lt;br&gt; • STM32  &lt;br&gt; • Lego EV3 (WiP)</td>
<td>• linux  &lt;br&gt; • linux_lib</td>
</tr>
</tbody>
</table>
MaRTE OS: *linux_lib* architecture

- MaRTE OS behaves as a pthread library for Linux
  - concurrency is provided at library level (not by using Linux threads)
  - no admin privileges required to assign priorities to threads
- Applications are executed as a standard Linux user process
- *Hardware Abstract Layer* (HAL) is based on Linux system calls

Original image from Mario Aldea in “MaRTE OS: Overview and Linux Version”
Motivation

• Online teaching during Covid-19 pandemic
  ▫ easy to install and use for students at home
• Research platform for EDF systems
  ▫ global-clock EDF
    • scheduling deadlines can be referred to the release time of tasks allocated in a different node
    • global clock is required
      ▫ clock synchronization protocol and network card drivers implemented in Linux
Problem #1: predictability in Linux

• **Issue to address:** the design of the Linux kernel favors throughput over determinism
  ▫ interferences from other users’ workload
  ▫ interferences from the Linux kernel

• **Proposed approach:**
  ▫ core partitioning for real-time tasks
    • *isolcpus* and *cpuset* facilities
    • tickless or *adaptative* tick mode
    • offload interrupts and workload to *non-real-time* cores
  ▫ Linux settings for preemptible kernel
    • *PREEMPT_RT* patch or *PREEMPT* low-latency kernel
Execution framework

- **OTHER APPS.**
  - IEEE 802.1AS PROTOCOL
  - PTP STACK IMPLEMENTATION & TOOLS

- **ISOLATED CORE**
  - REAL-TIME APP.
  - MARTE OS

- **LINUX SERVICES**

- **MULTICORE PROCESSOR**
  - CORE A
  - CORE B
  - ... CORE Z

- **SYSTEM CLOCK**
- **KERNEL**
- **NETWORK**
Tests platform

- Hardware: quad-core 1.9 GHz
- Software
  - kernel v4.4.256-rt214
  - each experiment is executed with SCHED_FIFO scheduling and maximum priority
  - core 2 isolated for real-time applications
    - isolcpu=2
    - nohz_full=2
    - shielding with cset
    - other options also enabled
Problem #1: experimental results (1/4)

- Monitoring IRQs with workload (1 hour)
  - workload added by `stress-ng`
    - 3 threads add synthetic workload (~25% system load per thread)
    - 2 threads add I/O operations
    - 1 thread adds timer interrupts
  - user load in the isolated core keeps at 0%

<table>
<thead>
<tr>
<th>IRQ</th>
<th>CPU0</th>
<th>CPU1</th>
<th>CPU2</th>
<th>CPU3</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O miscellaneous</td>
<td>807 333</td>
<td>1 198</td>
<td>0</td>
<td>3 566</td>
</tr>
<tr>
<td>Local timer interrupts</td>
<td>50 061 503</td>
<td>664 673 816</td>
<td>56 927</td>
<td>30 294 281</td>
</tr>
<tr>
<td>Rescheduling interrupts</td>
<td>173 638</td>
<td>28 420</td>
<td>1 247 011</td>
<td>428 492</td>
</tr>
<tr>
<td>Function call interrupts</td>
<td>665</td>
<td>622</td>
<td>9 261</td>
<td>668</td>
</tr>
<tr>
<td>TLB shootdowns</td>
<td>271</td>
<td>316</td>
<td>200</td>
<td>454</td>
</tr>
</tbody>
</table>
Problem #1: experimental results (2/4)

- Monitoring IRQs with workload (1 hour)
  - using the same workload than in the previous test
  - user load in the isolated core keeps at 0%
  - core 2 and core 3 disabled at boottime for Linux

<table>
<thead>
<tr>
<th>IRQ</th>
<th>CPU0</th>
<th>CPU1</th>
<th>CPU2</th>
<th>CPU3</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O miscellaneous</td>
<td>397 212</td>
<td>2 627</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Local timer interrupts</td>
<td>79 986 427</td>
<td>640 569 134</td>
<td>4 274</td>
<td>29 311 597</td>
</tr>
<tr>
<td>Rescheduling interrupts</td>
<td>66 832</td>
<td>33 063</td>
<td>1 137 134</td>
<td>173 616</td>
</tr>
<tr>
<td>Function call interrupts</td>
<td>588</td>
<td>552</td>
<td>2 210</td>
<td>143</td>
</tr>
<tr>
<td>TLB shootdowns</td>
<td>278</td>
<td>330</td>
<td>1</td>
<td>133</td>
</tr>
</tbody>
</table>
Problem #1: experimental results (3/4)

- Event-handling latency in core 2 using **cyclictest**
  - schedules timer events and compares the expected and actual wakeup time

![Diagram showing expected wakeup time, actual wakeup time, and latency over time]
Problem #1: experimental results (4/4)

- Event-handling latency in core 2 using `cyclictest`
  - using the same workload than in the previous test
Problem #2: clock overhead in \textit{linux\_lib} architecture

- **Issue to address:** overhead when reading the clock
  - calling \textit{clock\__gettime} is slower in MaRTE OS with \textit{linux\_lib}
    - frequent system calls can dominate overall performance
  - implementation details:
    - MaRTE OS with \textit{linux\_lib}: the \textit{clock\__gettime} function finishes in a Linux system call
    - Linux: the \textit{clock\__gettime} function is usually supported as vDSO (virtual dynamic shared object) to improve its performance
      - mechanism to export frequent \texttt{read-only} system calls to user space without a mode switch

- **Proposed approach:**
  - update \textit{linux\_lib} to use vDSO
  - minimize the number of kernel locks/unlocks in MaRTE
Problem #2: experimental results

• Overhead associated with the reading of the system clock

for 1 to 100000000:
    t1 = gettime(clock)
    t2 = gettime(clock)
    overhead = t2 – t1

• Executed 100,000,000 times
  ▫ *timing results are expressed in microseconds*

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Max</th>
<th>Avg</th>
<th>Min</th>
<th>Std Dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux RT</td>
<td>12.79</td>
<td>0.07</td>
<td>0.07</td>
<td>0.04</td>
</tr>
<tr>
<td>MaRTE OS over Linux</td>
<td>13.00</td>
<td>2.05</td>
<td>2.00</td>
<td>0.21</td>
</tr>
<tr>
<td>MaRTE OS over Linux with vDSO</td>
<td>9.80</td>
<td>0.19</td>
<td>0.18</td>
<td>0.06</td>
</tr>
</tbody>
</table>
Problem #3: Networking in *linux_lib* architecture

- **Issue to address:** the design of *linux_lib* does not favors the use of blocking network calls
  - when a thread blocks in a network call, the whole application blocks
    - e.g., waiting on a socket for incoming messages

- **Proposed approach:**
  - use a new communication layer (acting as a broker) and asynchronous I/O signals
    - user threads are blocked using MaRTe synchronization mechanisms
Problem #3: Proposed approach

- Sockets are configured to be non-blocking and asynchronous
- Linux raises a signal when new incoming messages are available for reading
- `epoll` syscall for monitoring multiple sockets
Conclusions

- Full isolation cannot be obtained
  - maximum interferences around tens of microseconds
- Clock overhead has been reduced
  - vDSO broadly available in current systems
- Networking functionality has been enhanced
  - *non-negligible* performance penalty compared to other architectures