Introduction to System z Architecture

Jens Freimann – jfrei@linux.vnet.ibm.com
KVM on System z
IBM Linux Technology Center
IBM Deutschland Research & Development GmbH

2013-10-21
About me

- At IBM for the last 5 ½ years
- 4 years in System z I/O Firmware
- 1 ½ years working on QEMU and KVM
About us
KVM/QEMU on S390 team in Germany

Christian Bornträger  Cornelia Huck  Heinz Graalfs  Thomas Huth
Dominik Dingel  Jens Freimann  Michael Müller
Introduction to System z

- System z Architecture
  - Hardware
  - Memory
  - Interrupts

- Lessons learned
Hardware
17 years later...
Hardware today

- Internal Batteries (optional)
- Power Supplies
- Support Elements
- I/O cage Carried Forward
- PCIe I/O drawer
- Flexible Service Processor (FSP) controller cards
- Processor Books with Memory, HCA and PCIe-Fanout cards
- InfiniBand and PCIe I/O Interconnects
- N+1 Water Cooling Units
Processor book
MCM, PU, core

@5.5 GHz
MCM, PU, core

@5.5 GHz
No need for liquid nitrogen :)

![Image of liquid nitrogen experiment](image-url)
Introduction to System z

• System z Architecture
  - Hardware
  - Memory
  - Interrupts

• Lessons learned
Memory

• System z is a Big-Endian machine
• Storage in z/Architecture means Memory(!)
  – zEC12: 3.75 TB max

Adressing

• Types of addresses:
  • **Virtual**: Translated by dynamic address translation (DAT) to real addresses
  • **Real**: Translated to absolute addresses using the prefix register
  • **Absolute**: After applying the prefix register
  • **Logical**: The address seen by the program (this can either be a virtual or a real address))
  • **Physical**: translated to absolute addresses by the Config Array
## Memory Address types

<table>
<thead>
<tr>
<th>absolute address</th>
<th>real address</th>
<th>virtual address</th>
</tr>
</thead>
</table>

- Apply prefix
- DAT tables
- Register
Storage keys

- No equivalent in x86
- One of four storage-protection mechanisms defined in z/Architecture
- Storage (memory) protection mechanism
  - Key-controlled protection
  - Associated with each 4K-byte block of real storage.
  - Program runs with storage key set in PSW

<table>
<thead>
<tr>
<th>ACC</th>
<th>F</th>
<th>R</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

- ACC = access-control bits
- F = fetch-protection bit
- R = reference bit
- C = change bit
Storage keys

Program-Status Word

Bits 0-31 of Instruction Address

Bits 32-63 of Instruction Address
Storage keys (cont.)

→ How to do migration of Storage keys efficiently? They are a separate entity besides memory, needs to be tracked

Model storage keys as device?
• Would provide hook to trigger migration
Keep it in a separate MemoryRegion?
• Subregion of RAM?
• More similar to real System
Introduction to System z

- System z Architecture
  - Hardware
  - Memory
  - Interrupts

- Lessons learned
Interrupts
prerequisite: Prefixing

- No equivalent on x86
- Map range of real addresses 0-8191 to a different block in absolute storage for each CPU

- Each CPU is assigned a private memory area of 8 KB, called prefix area
  - contains data critical to system operation, e.g. interrupt processing
  - other names: fixed storage locations, low core
Interrupts
prerequisite: Prefixing
Interrupts

• There are six classes of interrupts:
  – Supervisor call
  – Program
  – Machine check
  – External
  – Input/output
  – Restart

• Each class is associated with a pair of old/new PSWs in the assigned storage locations
Interrupt Action
(example I/O interrupt)

1. subchannel status pending, generate I/O IRQ

Current PSW

(like Program counter + status register)

* registers are saved/restored by software (OS)

Lowcore

<table>
<thead>
<tr>
<th>...</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x170</td>
<td>I/O old PSW</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x1F0</td>
<td>I/O new PSW</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Interrupt Action
(example I/O interrupt)

1. subchannel status pending, generate I/O IRQ

2. Store current PSW into lowcore field
   I/O old PSW

* registers are saved/restored by software (OS)
Interrupt Action
(example I/O interrupt)

1. subchannel status pending, generate I/O IRQ

2. Store current PSW into lowcore field I/O old PSW

3. Load I/O new PSW (irq handler)

* registers are saved/restored by software (OS)

Lowcore

<table>
<thead>
<tr>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x170</td>
<td>I/O old PSW</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x1F0</td>
<td>I/O new PSW</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Interrupt Action
(example I/O interrupt)

1. subchannel status pending, generate I/O IRQ

2. Store current PSW into lowcore field
   I/O old PSW

3. load I/O new PSW (irq handler)

4. run I/O irq handler
   Use I/O interrupt information in lowcore

* registers are saved/restored by software (OS)

---

Current PSW

(like Program counter + status register)

Lowcore

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x170</td>
<td>I/O old PSW</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x1F0</td>
<td>I/O new PSW</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Interrupt Action
(example I/O interrupt)

1. subchannel status pending, generate I/O IRQ

2. Store current PSW into lowcore field
   I/O old PSW

3. load I/O new PSW (irq handler)

4. run I/O irq handler
   Use I/O interrupt information in lowcore

5. restore old PSW

* registers are saved/restored by software (OS)

Current PSW

(like Program counter + status register)

Lowcore

<table>
<thead>
<tr>
<th>...</th>
<th>I/O old PSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x170</td>
<td>...</td>
</tr>
<tr>
<td>0x1F0</td>
<td>I/O new PSW</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
## PSWs in the Assigned Storage Locations

<table>
<thead>
<tr>
<th>Real addresses</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120 - 0x012F</td>
<td>Restart old PSW</td>
</tr>
<tr>
<td>0x130 - 0x013F</td>
<td>External old PSW</td>
</tr>
<tr>
<td>0x140 - 0x014F</td>
<td>Supervisor-call old PSW</td>
</tr>
<tr>
<td>0x150 - 0x015F</td>
<td>Program old PSW</td>
</tr>
<tr>
<td>0x160 - 0x016F</td>
<td>Machine-check old PSW</td>
</tr>
<tr>
<td>0x170 - 0x017F</td>
<td>I/O old PSW</td>
</tr>
<tr>
<td>0x1A0 - 0x01AF</td>
<td>Restart new PSW</td>
</tr>
<tr>
<td>0x1B0 - 0x01BF</td>
<td>External new PSW</td>
</tr>
<tr>
<td>0x1C0 - 0x1CF</td>
<td>Supervisor-call new PSW</td>
</tr>
<tr>
<td>0x1D0 - 0x1DF</td>
<td>Program new PSW</td>
</tr>
<tr>
<td>0x1E0 - 0x01EF</td>
<td>Machine-check new PSW</td>
</tr>
<tr>
<td>0x1F0 - 0x01FF</td>
<td>I/O new PSW</td>
</tr>
</tbody>
</table>
**Interrupt Masking**

- In z/Architecture masking is done via bits in PSW and in Control registers.

### Program-Status Word

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R</td>
</tr>
<tr>
<td>1</td>
<td>E</td>
</tr>
<tr>
<td>2</td>
<td>PSW key</td>
</tr>
<tr>
<td>3</td>
<td>M</td>
</tr>
<tr>
<td>4</td>
<td>W</td>
</tr>
<tr>
<td>5</td>
<td>P</td>
</tr>
<tr>
<td>6</td>
<td>AS</td>
</tr>
<tr>
<td>7</td>
<td>CC</td>
</tr>
<tr>
<td>8</td>
<td>Program mask</td>
</tr>
<tr>
<td>9</td>
<td>RI</td>
</tr>
<tr>
<td>10</td>
<td>RI</td>
</tr>
<tr>
<td>11</td>
<td>RI</td>
</tr>
<tr>
<td>12</td>
<td>RI</td>
</tr>
<tr>
<td>13</td>
<td>RI</td>
</tr>
<tr>
<td>14</td>
<td>RI</td>
</tr>
<tr>
<td>15</td>
<td>RI</td>
</tr>
<tr>
<td>16</td>
<td>RI</td>
</tr>
<tr>
<td>17</td>
<td>RI</td>
</tr>
<tr>
<td>18</td>
<td>RI</td>
</tr>
<tr>
<td>19</td>
<td>RI</td>
</tr>
<tr>
<td>20</td>
<td>RI</td>
</tr>
<tr>
<td>21</td>
<td>RI</td>
</tr>
<tr>
<td>22</td>
<td>RI</td>
</tr>
<tr>
<td>23</td>
<td>RI</td>
</tr>
<tr>
<td>24</td>
<td>RI</td>
</tr>
<tr>
<td>25</td>
<td>RI</td>
</tr>
<tr>
<td>26</td>
<td>RI</td>
</tr>
<tr>
<td>27</td>
<td>RI</td>
</tr>
<tr>
<td>28</td>
<td>RI</td>
</tr>
<tr>
<td>29</td>
<td>RI</td>
</tr>
<tr>
<td>30</td>
<td>RI</td>
</tr>
<tr>
<td>31</td>
<td>RI</td>
</tr>
</tbody>
</table>

### Control Registers

- **CR0**
  - Ext
  - Bits 48-59

- **CR6**
  - I/O
  - Bits 32-39

- **CR14**
  - M
  - Bits 36-39

- **CR1** Bit 57 Primary space-switch-event control
- **CR13** Bit 57 Home space-switch-event control

### Bits of Instruction Address

- **Bits 0-31 of Instruction Address**
  - Bits 64-95

- **Bits 32-63 of Instruction Address**
  - Bits 96-127
Interrupt Masking (cont.)

- There is no masking for
  - Supervisor calls (SVCs)
    - The whole purpose of the SUPERVISOR CALL instruction is to invoke the supervisor via the interrupt mechanism
  - Restart
    - SIGNAL PROCESSOR instruction, typically issued by the operating system during startup
    - Manual operation available from the support element (SE) intended to restart the operating system
  - Exigent machine checks
    - If PSW.13 is 0, the CPU check stops. An example of such a situation is instruction processing damage.
Updates regarding I/O

- Adapter interrupts
  - per Interruption Subclass (ISC)
  - Lightweight compared to classic I/O interrupts

<table>
<thead>
<tr>
<th>Classic I/O interrupts</th>
<th>Adapter interrupts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Get interrupt information from lowcore</td>
<td>1. Get interrupt information from lowcore</td>
</tr>
<tr>
<td>2. Test subchannel (tsch)</td>
<td>2. find indicator bit</td>
</tr>
<tr>
<td>3. find indicator bit</td>
<td></td>
</tr>
</tbody>
</table>
Updates regarding I/O

• Adapter interrupts
  – per Interruption Subclass (ISC)
  – Lightweight compared to classic I/O interrupts

Classic I/O interrupts
1. Get interrupt information from lowcore
2. Test subchannel (tsch)
3. find indicator bit

Adapter interrupts
1. Get interrupt information from lowcore
2. find indicator bit
Introduction to System z

- System z Architecture
  - Hardware
  - Memory
  - Interrupts

- Lessons learned
Lessons learned
A brief history of KVM on S390

- Built own userspace “kuli”, which was not an emulator but a small and simple driver for KVM

- KVM code was built to fit into kuli design

- Long pause in KVM on System z development

- Decision to go for QEMU as preferred userspace. Needed to adapt to QEMU “thinking”. Still learning...
Lessons learned

- Having only one single KVM exit reason turned out to be a bad idea
  - Need to sync everything all the time
  - X86 with multiple exit reasons has it easier
  - Introduced separate exit for “test subchannel”
Lessons learned (cont.)

- Worksplit between KVM and userspace caused us some headache
  - Example: reset/diag 308 where we reset some parts in KVM and others in QEMU
  - QEMUs “school of thinking” that all state is kept in userspace makes sense
Lessons learned (cont.)

- Keeping number of running CPUs in a global variable
  - When last CPU is stopped, shutdown guest
  - How to model this in a better way?
Thank you!

Thanks to Joachim von Buttlar for borrowing me some of his slides
This work represents the view of the author and does not necessarily represent the view of IBM.

IBM, IBM(logo), z/Architecture, zSeries, Enterprise Systems Architecture/390, ESA/390, Enterprise Systems Architecture/370, ESA/370 and System/360 are trademarks and/or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds.

Other company, product and service names may be trademarks or service marks of others.