Porting Linux to the IBM zSeries platform

Dr. Ulrich Weigand <uweigand@de.ibm.com>
Linux on zSeries Development, IBM Lab Böblingen
Agenda

- **Linux on zSeries**
  - Platform overview
  - The Linux port

- **zSeries Architecture**
  - Processor
  - Memory
  - I/O Subsystem

- **Virtualization issues**
  - Why virtualization?
  - Memory management
  - Timer ticks
zSeries Hardware
zSeries Historical Overview

- **1964: System/360**
  - First system to define *architecture* common across implementation
  - 24-bit address space
- **1971: System/370**
  - Virtual storage, multi-processor support
- **1981: System/370 XA (Extended Architecture)**
  - 31-bit address space, channel subsystem, interpretive execution
- **1988: Enterprise System Architecture/370**
  - Multiple address spaces, LPAR
- **1990: Enterprise System Architecture/390**
  - Enhanced instruction set, IEEE floating point
- **1994: Parallel Sysplex**
  - Coupling facility, 32-way clusters
- **2000: z/Architecture**
  - 64-bit arithmetic and address space
  - HiperSockets, SCSI/FCP
zSeries Architecture Overview

- **Core features**
  - CISC using 600+ instructions
  - Channel-based I/O subsystem
  - Efficient virtualization capabilities

- **Designed for RAS**
  - Two units operated in parallel, checkpoint-restart on miscompare
  - Hot-spare CPU take-over without OS involvement
  - Concurrent firmware update without downtime

- **Typical usage**
  - Transaction systems, database servers, ...
  - Operating systems: z/OS, VSE/ESA, TPF, z/VM, Linux
  - Linux brings 'modern' applications to the platform!
Linux on zSeries System Components

- Linux Kernel
  - glibc
  - gcc
  - grep
  - mount
  - cvs
  - gdb
  - ls

- GNU - essentials
- Unix - tools
- Applications
  - SAP R/3
  - DB2 UDB
  - Samba
  - Apache

- Applications
  - Apache
  - Samba
  - DB2 UDB
  - SAP R/3
Linux on zSeries System Structure
## Linux on zSeries Code Contributions

<table>
<thead>
<tr>
<th></th>
<th>&quot;May 2002&quot; stream</th>
<th>SuSE SLES-8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>z-relevant</td>
</tr>
<tr>
<td>kernel 2.4.19</td>
<td>4,600,000</td>
<td>1,100,000</td>
</tr>
<tr>
<td>binutils 2.12.90.0.15</td>
<td>1,800,000</td>
<td>750,000</td>
</tr>
<tr>
<td>gcc 3.2</td>
<td>3,700,000</td>
<td>2,000,000</td>
</tr>
<tr>
<td>glibc 2.2.5</td>
<td>2,000,000</td>
<td>1,700,000</td>
</tr>
<tr>
<td>gdb 5.2.1</td>
<td>2,000,000</td>
<td>1,100,000</td>
</tr>
</tbody>
</table>

|                | Total             | z-relevant       | z-specific      |
| kernel 2.4.19  | 6,000,000         | 1,400,000        | 160,000         | 11.4%   |
| binutils 2.12.90.0.15 | 1,800,000     | 750,000          | 10,000          | 1.3%    |
| gcc 3.2        | 3,700,000         | 2,600,000        | 15,000          | 0.6%    |
| glibc 2.2.5    | 2,100,000         | 1,800,000        | 13,000          | 0.7%    |
| gdb 5.2.1      | 1,500,000         | 1,100,000        | 10,000          | 0.9%    |
Linux on zSeries Development Process

Open Source Community

IBM Linux on zSeries development

developerWorks web site

Linux Distribution Partners

Customers
z900 Processor and Memory

6 x 1 GB/s

Memory
(up to 2 x 16 GB)

Level 2 Cache (16 MB)

6 x 1 GB/s

Memory
(up to 2 x 16 GB)

Level 2 Cache (16 MB)
z900 Multi-Chip Module (MCM) - 20 PUs
zSeries Processor Architecture

- **Architecture modes**
  - ESA/390 (32-bit) vs. z/Architecture (64-bit)
  - S/390 machines (1990..1999) support only ESA/390
  - zSeries machines (2000..) operate in either mode

- **Addressing modes**
  - 24-bit, 31-bit, 64-bit (z/Architecture only)
  - Support for mixed addressing mode operation

- **Register file**
  - Program Status Word (instruction address, condition code, ...)
  - 16 general purpose registers (32-bit / 64-bit)
  - 16 floating point registers (64-bit) + floating point control word
  - 16 access registers (32-bit)
  - 16 control registers (32-bit / 64-bit)
zSeries Processor Architecture (cont.)

- **Instruction format**
  - 2-byte, 4-byte, or 6-byte instructions (2-byte aligned)
  - Pre-defined instruction formats

- **Example: OR instruction family (32-bit)**

```
OR  R1, R2   [RR]   '16' R1 R2
O   R1, D2(X2, B2)  [RX]   '56' R1 X2 B2 D2
OILL R1, I2      [RI]   'A5' R1 B' I2
OI  D1(B1), I2    [SI]   '96' I2 B1 D1
OC  D1(B1), D2(B2) [SS]   'D6' L B1 D1 B2 D2
```
Example "Hello World" Application

```
.section .rodata
.align 2
.LC0: .string "Hello, world!"
.text
.align 4
.globl main
main:
    stm %r13,%r15,52(%r15)
    bras %r13,.L3
.L2: .align 4
.LC1: .long .LC0
.LC2: .long puts
    .align 2
.L3:
    lr %r14,%r15
    ahi %r15,-96
    l %r1,.LC2-.L2(%r13)
    l %r2,.LC1-.L2(%r13)
    st %r14,0(%r15)
    basr %r14,%r1
    lhi %r2,0
    lm %r13,%r15,148(%r15)
    br %r14
```
Condition Code Handling

- **zSeries condition code**
  - Two-bit value (0..3) stored in the Program Status Word
  - Set by various instructions (arithmetic, logical, comparison, ...)
  - Used by conditional branch instructions
  - No globally fixed semantics of condition code values!

- **Branch condition mask**
  - Current CC value selects one of four mask bits
  - Branch taken if that bit is set
  - Mask 0 is 'branch never' (nop), mask 15 is unconditional branch
  - Every branch condition invertible!

<table>
<thead>
<tr>
<th>Condition code</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mask position value</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Condition Code Handling (cont.)

- Some regular CC examples

<table>
<thead>
<tr>
<th></th>
<th>CC 0</th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparison (signed)</td>
<td>Operands equal</td>
<td>First operand low</td>
<td>First operand high</td>
<td>Operands unordered</td>
</tr>
<tr>
<td>Comparison (unsigned)</td>
<td>Operands equal</td>
<td>First operand low</td>
<td>First operand high</td>
<td>n/a</td>
</tr>
<tr>
<td>Arithmetical operations</td>
<td>Result zero; no overflow</td>
<td>Result &lt; 0; no overflow</td>
<td>Result &gt; 0; no overflow</td>
<td>Overflow</td>
</tr>
<tr>
<td>Logical operations</td>
<td>Result zero; no carry</td>
<td>Result not zero; no carry</td>
<td>Result zero; carry</td>
<td>Result not zero; carry</td>
</tr>
<tr>
<td>Zero test</td>
<td>Result zero</td>
<td>Result not zero</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>
Condition Code Handling (cont.)

- **Irregular CC example: TEST UNDER MASK LOW**
  - Instruction: `TML R1, I2`
  - Compares low 16 bits of operand 1 bitwise with operand 2
  - Sets condition code according to the result
    - 0: Selected bits all zeros; or mask bit all zeros
    - 1: Selected bits mixed ones and zeros, and leftmost is zero
    - 2: Selected bits mixed ones and zeros, and leftmost is one
    - 3: Selected bits all ones

- **Usage example**
  - Source: `if ((flags & 0x80) && !(flags & 0x04)) { ... }`
  - Assembler: `tml %rl, 0x84 ; brc 2, ...
  - GCC 3.3 generates optimal code sequence
zSeries Program Status Word

- R: Program Event Recording Mask
- T: Dynamic Address Translation Mode
- IO: I/O Interruption Mask
- EX: External Interruption Mask
- Key: PSW Key (storage protection)
- M: Machine Check Mask
- W: Wait State
- P: Problem State
- AS: Address Space Control
- CC: Condition Code
- PM: Program Mask
- EA: Extended Addressing Mode
- BA: Basic Addressing Mode
zSeries Interruption Actions

- **Types of Interruptions**
  - Restart Interruption (*once at boot*)
  - I/O Interruption (*associated with subchannel*)
  - External Interruption (*e.g. timer, inter-processor interrupt*)
  - Machine Check Interruption (*e.g. dynamic device reconfiguration*)
  - Program Interruption (*e.g. page fault, illegal instruction, ...*)
  - Supervisor Call Interruption (*Linux system call*)

- **Interruption Action**
  - Save interruption code to assigned storage location
  - Save old PSW to assigned storage location
  - Load new PSW from assigned storage location
    - Branch to interruption handler code
    - May change addressing mode / address translation mode
    - May enable/disable interruptions
    - May switch between supervisor state / problem state
zSeries Dynamic Address Translation: 31-bit

Segment Table Origin

STO

Virtual Address

Segment Index (11 bit)

SX

Page Index (8 bit)

PX

Byte Index (12 bit)

BX

Segment Table

PTO

Page Table

PFRA

Real Address

PFRA

Page Frame Real Address

© 2003 IBM Corporation
zSeries Dynamic Address Translation: 64-bit

Region Table Origin

Virtual Address

Region-1st Table

Region-2nd Table

Region-3rd Table

Segment Table

Page Table

Byte Index (12 bit)

Region-1st Index (11 bit)

Region-2nd Index (11 bit)

Region-3rd Index (11 bit)

Segment Index (11 bit)

Page Index (8 bit)

Segment

Page

Byte

Region 1st Table

Region 2nd Table

Region 3rd Table

Segment Table

Page Table

Virtual Address

Real Address

RFTO

RSTO

RTTO

STO

PTO

PFRA

PX

BX

© 2003 IBM Corporation
zSeries DAT: 64-bit Three-Level Translation

Region Table Origin

Virtual Address

Region-1st Table

Region-2nd Table

Region-3rd Table

Segment Table

Page Table

Byte Table

RTTO

0

0

RTX

SX

PX

BX

STO

PTO

PFRA

PFRA

BX
zSeries Address Translation Modes

- **Directly accessible address spaces**
  - Primary space: STO/RTO in Control Register 1
  - Secondary space: STO/RTO in Control Register 7
  - Home space: STO/RTO in Control Register 13
  - Access-register specified spaces

- **Access registers**
  - Base register used in memory access identifies access register
  - AR indirectly specifies STO/RTO via Access List Entry Token
  - Operating System manages ALETs and grants privilege
  - Special use: ALET 0 for primary space, ALET 1 for secondary space

- **Translation mode specified in PSW**
  - Primary space mode: use primary space
  - Secondary space mode: instructions in primary, data in secondary
  - Home space mode: use home space
  - Access register mode: instructions in primary, data AR-specified
Linux on zSeries: Use of Address Spaces

Primary space mode:
- `mvc 0(8,%r2),0(%r4)`
- `la 4,<source>`
- `sac 512`
- `mvc 0(8,%r2),0(%r4)`
- `sac 0`

Home space mode:
- `mvc 0(8,%r2),0(%r4)`
zSeries System Structure
zSeries Channel Subsystem

Subchannel Number:
Logical appearance of device to the OS
16-bit number assigned sequentially

Channel Path Identifier (CHPID):
 Assigned for each physical path
8-bit number defined in I/O configuration data set

Device Number:
 Used to identify a device to the operator
16-bit number defined in I/O configuration data set

A specific device (identified by the user via its device number) is accessed by the Operating System using its subchannel number; the channel subsystem manages one or more channel paths to one or more control units connected to the device.
## zSeries Subchannel Information Block

### Path Management Control Word:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>8</th>
<th>16</th>
<th>24</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IPA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCW Address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Device Status</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subch. Status</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Count</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Subchannel Status Word:

```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- ISC: Interruption Subclass
- E: Enabled Flag
- LM: Limit Mode
- MM: Measurement Mode Enable
- D: Multipath Mode
- T: Timing Facility
- V: Device Number Valid
- LPM: Logical-Path Mask
- PNOM: Path Not Operational Mask
- LPUM: Last Path Used Mask
- PIM: Path Installed Mask
- PAM: Path Available Mask
- POM: Path Operational Mask
- MBI: Measurement Block Index
- S: Concurrent Sense
zSeries Subchannel Operations

- **How to perform an I/O operation**
  - Build a channel program to execute
  - Fill pointer to channel program into an operation request block
  - Start operation using START SUBCHANNEL instruction
  - Channel subsystem executes operation asynchronously
  - Completion triggers I/O interruption

- **Channel Program**
  - Sequence of Channel Command Words
  - Each CCW specifies Command Code and Data Address / Count
  - Control flow: command/data chaining, transfer-in-channel, suspend
  - Generic command codes: e.g. READ / WRITE
  - Device-specific command codes: e.g. LOCATE RECORD
zSeries Virtualization

Still 4 separate images, but...

NOW, Dynamic resource balancing achieved via LPAR management
zSeries Virtualization: z/VM Operating System

- CMS 'User'
- CMS 'User'
- CMS 'User'
- VSE/ESA Rel. X.Y
- Linux/390 2.2
- MVS/ESA
- OS/390

Control Program, CP

Hardware: Memory, Channels, DASDs, ...
zSeries Virtualization: Logical Partitioning (LPAR)

Hypervisor

Hardware: Memory, Channels, DASDs, ...

VM/ESA Rel. A.B
VSE/ESA Rel. B.C
Linux/390 2.2
Linux/390 2.4
OS/390 Rel N.M
OS/390 Rel. S.T
zSeries Virtualization: Hardware Support

- **Interpretive Execution**
  - START INTERPRETIVE EXECUTION instruction
  - Control block describes guest register file
  - Fine-grained interception control
  - Hardware support for up to two levels of SIE
  - Multi-processor guests supported

- **Processor architecture**
  - Wait state to avoid busy loops
  - CPU timer to access virtual CPU cycles

- **Memory**
  - Memory zone support for LPAR
  - Two-level hardware for DAT and TLB
Linux under VM: Pseudo Page Faults

- **Two-level dynamic address translation**
  - Linux DAT: Linux virtual address -> Linux 'real' address
  - VM DAT: Guest 'real' address -> Host real address

- **Two-level page fault handling**
  - Guest page fault
    - Linux page fault handler invoked
    - Initiates page-in operation from backing store
    - Suspends user process until page-in completed
    - Other user processes continue to run
  - Host page fault
    - VM page fault handler invoked
    - Initiates page-in operation from backing store
    - Suspends **guest** until page-in completed
    - **No other user processes can run**
Linux under VM: Pseudo Page Faults (cont.)

- **Solution: Pseudo Page Faults**
  - VM page fault handler invoked
    - Initiates page-in operation from backing store
    - Triggers guest 'pseudo page fault'
    - Linux pseudo page fault handler suspends user process
    - VM does not suspend the guest
  - On completion of page-in operation
    - VM calls guest pseudo page fault handler again
    - Linux handler wakes up blocked user process

- **Caveats**
  - Access to kernel pages
  - Access to user page from kernel code
Linux under VM: Idle Guest Overhead

- **Problem: timer tick every 10 ms**
  - consumes about 0.3% of one G5 CPU
  - VM considers guest always busy

- **Reasons for timer tick**
  - Increment internal clock ("jiffies")
  - Update wall clock
  - Process accounting (user/system time etc.) and scheduling
  - Execute scheduled events (timers, time slice expiry, ...)
Linux under VM: Timer Patch v1

- **Solution: "No more jiffies" timer patch**
  - Provided as kernel patch on developerWorks
  - Integrated e.g. in SLES-7 distribution

- **Eliminates "jiffies" and 100 Hz timer tick**
  - Internal clock computed on-the-fly from TOD clock
  - Uses clock comparator and CPU timer interrupts as needed for scheduled events (timers, time slices)
  - Process accounting done at system call entry/exit

- **Results**
  - Reduced CPU consumption
  - VM recognizes idle guests
Linux under VM: Timer Patch v2

- **Problem: System call overhead**
  - Timer patch v1 introduces overhead
  - Noticable on large, busy servers

- **Solution: Timer patch v2**
  - Available on developerWorks since May 2002
  - Integrated e.g. in SLES-8 distribution

- **How does it work?**
  - While CPU is busy: use 100 Hz tick as usual
  - While CPU is idle: stop 100 Hz tick
  - No system call overhead
  - VM still recognizes idle guests
Linux under VM: Timer Patch Results
Questions