Linux for zSeries

Early Experiences with 64-bit Linux

Agenda

- z/Architecture Overview
- Linux implementation for z/Architecture
- ABI changes
- Building a kernel on a 31-bit system
- Building a file system from scratch
- Early experiences with ThinkBlue64

Linux for zSeries

z/Architecture Overview

z/Architecture Overview

- z/Architecture is the next step in the evolution from the System/360 to the System/370, S/370-XA, ESA/370, and ESA/390.
- z/Architecture includes all of the facilities of ESA/390 except for the asynchronous-pageout, asynchronous-data-mover, program-call-fast, and vector facilities.

z/Architecture Overview

- Four key features of z/Architecture include:
  - It is a full 64-bit architecture that provides for 24, 31 and 64-bit coexistence.
  - Intelligent Resource Director—Provides for an exclusive way to intelligently direct the processor and I/O resources to priority workloads running within the set of clustered LPARs.
  - HiperSockets—A statement of direction for z/Architecture that permits a TCP/IP network to be established between LPARs.
  - License Manager Enablement—The z/Architecture includes capabilities that enable IBM's License Manager to run on z/OS and z900. This capability, when combined with HiperSockets, creates an 'n-tier' environment for e-business applications within a z900.

z/Architecture Overview

- 64 bit PSW
  - Bit 12 – '0' specifies z/Architecture
- 64 bit control registers
- 16 IEEE/HFP registers
  - No need for software emulation

z/Architecture Overview

- 64 bit general registers
  - Can be operated upon as 64 or 32 bit entities

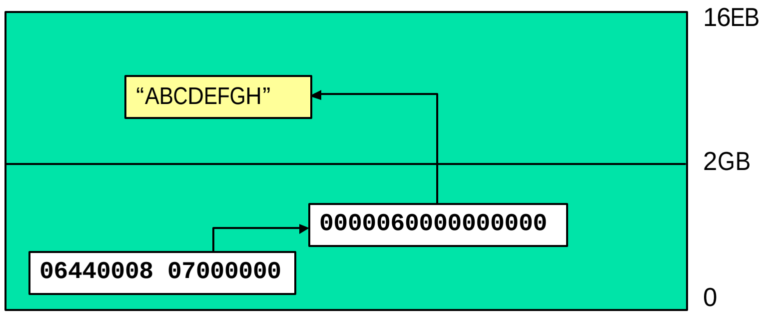
```
#include <stdio.h>
int main(int argc, char **argv)
{
 union {  long x;  int  y[2]; } longvar;

   longvar.x = -1;
   printf("%08X %08X %ld\n",longvar.y[0],longvar.y[1],longvar.x);
   __asm__  __volatile__ ("slr %0,%0" : "+d" (longvar.x) : : "cc");
   printf("%08X %08X %ld\n",longvar.y[0],longvar.y[1],longvar.x);
   __asm__ __volatile__ ("slgr %0,%0" : "+d" (longvar.x) : : "cc");
   printf("%08X %08X %ld\n",longvar.y[0],longvar.y[1],longvar.x);
}

FFFFFFFF FFFFFFFF -1
FFFFFFFF 00000000 -4294967296
00000000 00000000 0
```

z/Architecture Overview

- 64 bit addressing
  - 24 bit support
  - 31 bit support
  - Up to 3 levels of “Region Tables” to give:
    - 42, 53, 64 bit addressing
  - Use sam*xx* instruction to switch addressing modes
- New term:
  - >16MB = “above-the-line”
  - >2GB = “above-the-bar”

z/Architecture Overview

- 32 bit Access Registers
- CCWs still only use 31 bit address fields
  - IDAL used for “above-the-bar”



z/Architecture Overview

- Prefix page now 8KB
- LOTS of new instructions
  - 64 bit versions of 32 bit ops: LG (load) = L (load)
  - Instructions to manipulate 32 bit entities: LGFR
  - Some new compiler-friendly: RLL/RLLG; ALC/ALCG
  - Address mode related: SAM24/31/64; TAM
  - Unicode support: CUUTF; TRE
  - Enhanced relative branching: +/- 2GB branches

z/Architecture Overview

- New old/new PSW locations

```
EXT 1004  130 OLD 07060001 80000000 00000000 00015F1A
          1B0 NEW 04000001 80000000 00000000 00014D32
SVC 008E  140 OLD 0701C001 80000000 00000200 002618A6
          1C0 NEW 04000001 80000000 00000000 0001406C
PRG 0004  150 OLD 07004001 80000000 00000000 00087C7A
          1D0 NEW 04000001 80000000 00000000 00014AD6
MCH 0000  160 OLD 00000000 00000000 00000000 00000000
          1E0 NEW 04000001 80000000 00000000 00014DEA
I/O 0004  170 OLD 07060001 80000000 00000000 00015F1A
          1F0 NEW 04000001 80000000 00000000 00014C3A
```

z/Architecture Overview

- Implemented on:
  - z900 (aka Freeway) processors
  - Hercules
  - Flex/ES (or should be in the future)
- Supported by:
  - z/VM
  - OS/390
  - Linux for zSeries

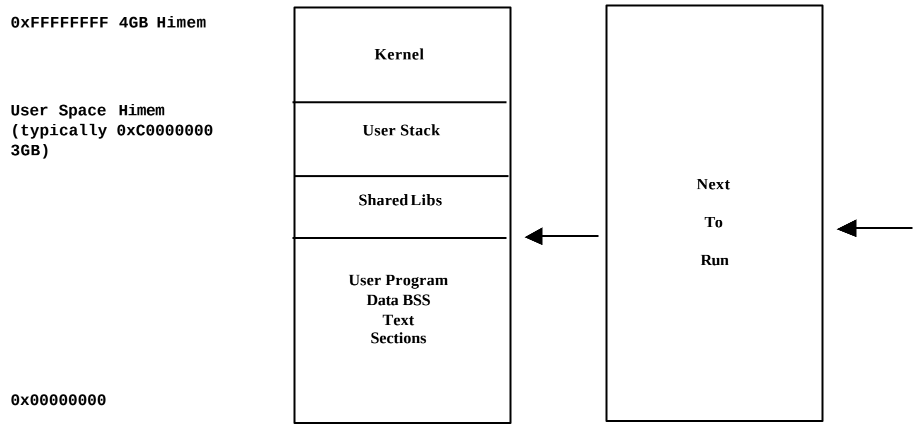
Linux for zSeries

Linux Implementation for z/Architecture

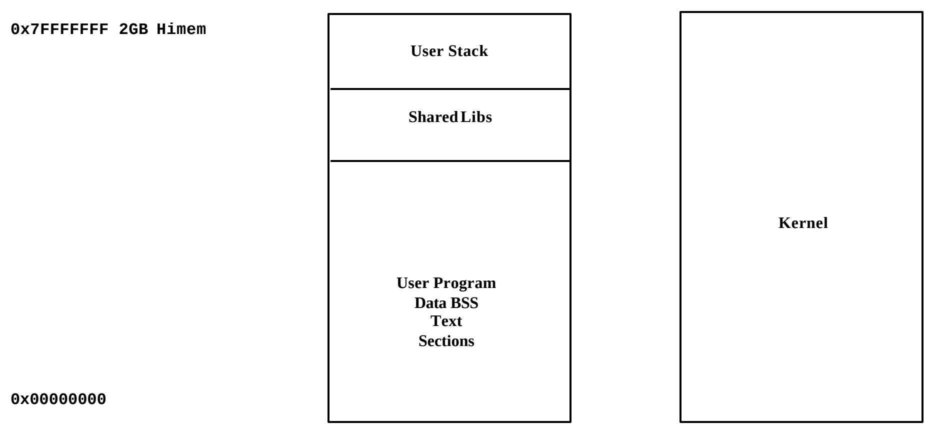
Linux for zSeries

- Based on 2.4 kernel
- Requires:
  - binutils
  - gcc
  - glibc
- Boots in 31 bit mode
- Switches to 64 bit mode fairly quickly

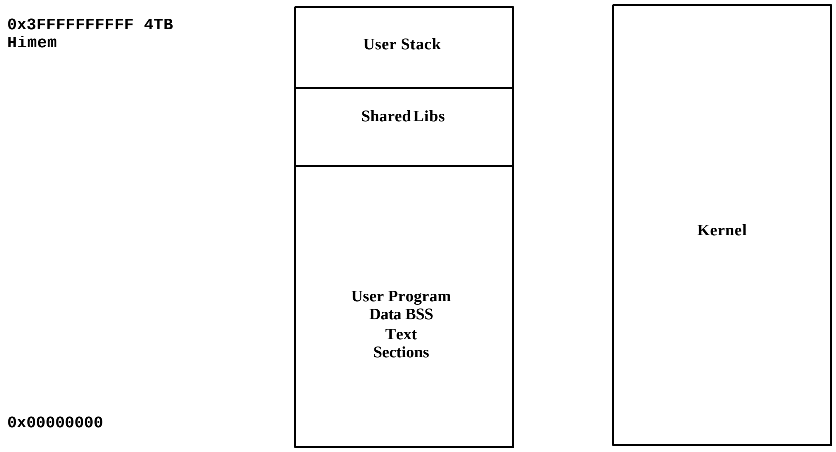
Linux – Intel Address Spaces



Linux – S/390 Address Spaces



Linux – zSeries Address Spaces



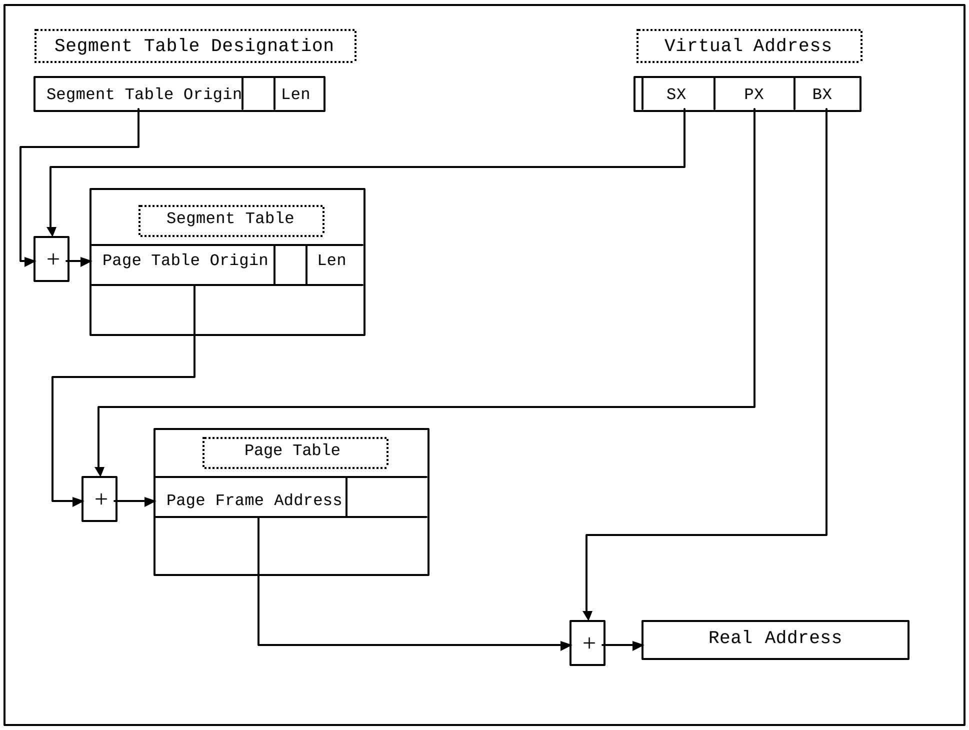
Linux for S/390 & zSeries

- A virtual address on S/390 is made up of 3 parts:

	Segment Index	Page Index	Byte Index
0	1	12	20 – 31

- On z/Architecture in Linux we currently make up an address from 4 parts:

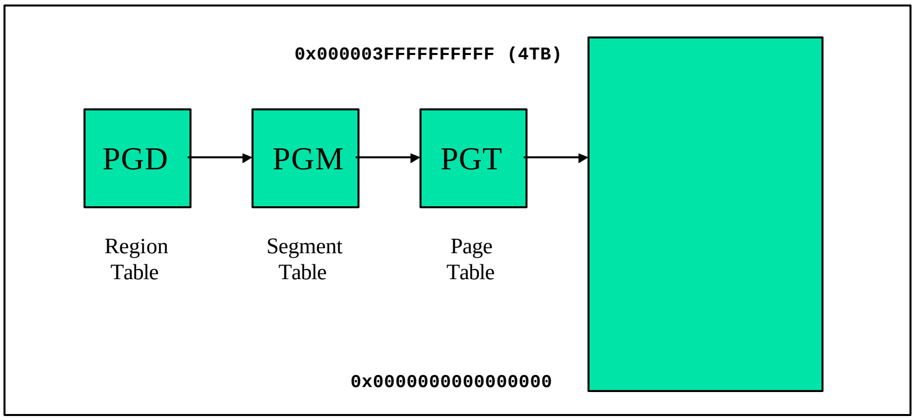
XXXXXXXXX	Region Index	Segment Index	Page Index	Byte Index
0	22	33	41	52 – 63



Linux for zSeries

- 64-bit
- 4TB address spaces
  - 1 Region Table
  - Segment Table
  - Page Table
- 31-bit compatibility mode
  - Existing apps will run
  - Provided they can find their libraries!

zArchitecture Address Spaces



Address Spaces

- Kernel runs in Primary Space mode
- User programs run in Home Space mode
- Copy to/from user just a MVC(L/E) in Access Register mode with AR set for kernel/user address spaces
- Compare this to some of the other elaborate schemes used

Address Space Usage

```
0000000080000000-0000000080008000 r-xp 0000000000000000 5e:01 207901 /bin/more
0000000080008000-0000000080009000 rw-p 0000000000007000 5e:01 207901 /bin/more
0000000080009000-000000008000d000 rwxp 0000000000000000 00:00 0
0000020000000000-000002000001b000 r-xp 0000000000000000 5e:01 223562 /lib/ld-2.2.2.so
000002000001b000-000002000001d000 rw-p 000000000001a000 5e:01 223562 /lib/ld-2.2.2.so
000002000001d000-000002000001f000 rw-p 0000000000000000 00:00 0
0000020000024000-0000020000028000 r-xp 0000000000000000 5e:01 223625 /lib/libtermcap.so.2.0.8
0000020000028000-0000020000029000 rw-p 0000000000003000 5e:01 223625 /lib/libtermcap.so.2.0.8
0000020000029000-0000020000170000 r-xp 0000000000000000 5e:01 223567 /lib/libc-2.2.2.so
0000020000170000-0000020000179000 rw-p 0000000000146000 5e:01 223567 /lib/libc-2.2.2.so
0000020000179000-000002000017f000 rw-p 0000000000000000 00:00 0
000003ffffffd000-0000040000000000 rwxp ffffffffffffe000 00:00 0
```

New Device Drivers

- Tape
  - 3490
  - Character and block
- 3270
  - Console
  - Standard terminal
- Cisco Routers

Device Drivers

- CCWs must live “below-the-bar”
- Kernel supports memory requests for under the bar storage
- Device drivers build CCW programs in this storage
- IDALs used to address “above-the-bar” storage

Linux for zSeries

ABI Changes

Application Binary Interface

- The Executable and Linkage Format Application Binary Interface (or ELF ABI), defines a system interface for compiled application programs. Its purpose is to establish a standard binary interface for application programs on LINUX for S/390 systems.

Application Binary Interface

- Defines (amongst other things):
  - Data formats
  - Byte layouts
  - Stack layouts
  - Process initialization
  - Register conventions
  - Routine linkage
  - Parameter passing
  - Returning results

Application Binary Interface

- Changes required for 64-bit support
  - Stack layouts
  - Routine prologues
  - Register conventions
  - Parameter passing
- Transparent for compiled applications
- Need to understand for such things as “FFI” or “JNI” or writing compilers

Stack Frame Layouts

Offset	Offset	Description
0	0	Back chain (a 0 here signifies end of back chain)
4	8	EOS (end of stack, not used on Linux for S390)
8	16	Glue used in other linkage formats
12	24	Glue used in other linkage formats
16	32	Scratch area
20	40	Scratch area
24-63	48-127	GPR register save area
64-79	128-159	FPR4 & FPR6 save area
96	160	Outgoing args (length x)
96+x	160+x	Possible stack alignment
96+x+y	160+x+y	alloca space of caller (if used)
96+x+y+z	160+x+y+z	alloca space of caller (if used)

31 Bit Co-existence

- ELF header indicates executable as:
  - S/390
  - 31 bit/64 bit
- Dynamic executables contain information regarding location of shared libraries
- ld.so.1 or ld.64 resolves information in elf header

31 Bit Co-existence

- Use **`ldd`** command to show what libraries your executable requires
- 31 bit apps cannot use 64 bit libraries
- LD_LIBRARY_PATH environment variable overrides internal specification of executable
- Can be set up globally or per application

Linux for zSeries

Building your first kernel

Building your first Kernel

- Create a cross-compile environment
  - **`binutils-2.10.0.8`** + experimental patches
  - **`gcc-2.95.2`** + experimental patches
  - **`glibc-2.2.2`** + experimental patches
- Download kernel from ftp.kernel.org
- Apply patches

Building your first Kernel

- Update Makefile:
  - `ARCH := s390x`
  - `CROSS_COMPILE=s390x-ibm-linux-`
- `make menuconfig`
- `make dep`
- `make`
- `make modules`
- `make modules_install`

Building your first Kernel

- Build silo:
  - `cd arch/s390x/tools/silo`
  - `make`
- Copy files to /boot:
  - `cp arch/s390x/boot/image /boot`
  - `cp arch/s390x/boot/*.boot /boot`
  - `cp System.map /boot`
- Run silo:
  - `cd /boot`
  - `<path-to-kernel-tools>/silo –f image –p parmfile –d /dev/dasdx`

Building your first Kernel

- Most things will work with your 31 bit executables and glibc-2.1.3
- Certain system interfaces have changed and are not supported by library routines
- Good enough to build everything you'll need

Linux for zSeries

Building a file system from scratch

Building a 64-bit file system

- Use first kernel as a base
- Get "Building LFS from Scratch" from: http://linuxfromscratch.org
- Get second disk and mount as **/root64**
- Build **ncurses** using cross-compiler
- Follow instructions in LFS document to populate **/root64**

Building a 64-bit file system

- Basically a two-stage process:
  - Starter pack:
    - Build statically linked versions of core packages (inc. gcc)
    - Create traditional file layouts (i.e. **/etc**, **/lib** etc.)
    - Create **/dev** nodes – need to manually define **/dev/dasdx**
    - Create **/etc** contents (e.g. **passwd**, **groups**)
    - **chroot** to **/root64**

Building a 64-bit file system

- Production pack:
  - Build new glibc-2.2.2
  - Rebuild core packages - this time dynamically linked
  - Create and tailor startup scripts
  - Copy **`/boot`** contents & **`/lib/modules`**
  - Update parameter file and run **`silo`**
  - Run **`depmod`**
  - Reboot off the /root64 disk

Linux for zSeries

ThinkBlue64 – ***Very*** Early Experiences

ThinkBlue64

- Redhat-like distribution
- Download from http://linux.zseries.org
- CDROM ISO image available
- 749 RPMS
- Starter system:
  - Kernel (tape or VM reader)
  - Initial RAMDISK (tape or VM reader)
  - Parameter file

ThinkBlue64

- Starting:
  - Mount CDROM on another Linux system:
    `mount –o loop ThinkBlue64-disc1.iso /mnt/cdrom`
  - Add **/mnt/cdrom** to **/etc/exports** and restart NFS server

```
# See exports(5) for a description.
# This file contains a list of directories exported to other computers
# It is used by rpc.nfsd and rpc.mountd.
/mnt/cdrom      10.20.45.7(rw;no_root_squash)
```

  - `/etc/rc.d/nfsserver restart`

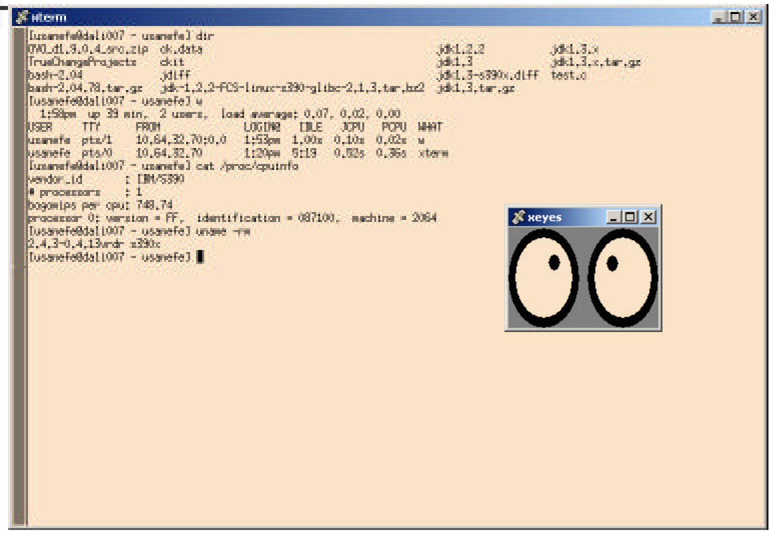
ThinkBlue64

- Upload starter components
- Punch to and boot from reader
- Answer questions:
  - IP connectivity
  - NFS server location
- Telnet to starter system
- Begin install of RPMS: **`./install`**

ThinkBlue64

- Three panels of questions:
  - Disks to use and mount points: No swap
  - NFS server containing RPMS
  - Repeat answers on IP addresses etc.
  - Install begins
- Install process runs **`zilo`**
- Now boot from disk

ThinkBlue64



ThinkBlue64

- Current work:
  - Bash2
    - Problem with signal handling: Union of pointer and int
  - Regina ported!!!
  - JDK 1.3
    - Porting invokeNative_s390.S
    - Instructions: **`sllg r1,r1,2`** versus **`sll r1,2`**
  - Assessing requirements & efforts for SAG products
    - Tamino currently runs on Linux for S/390
  - Samba 2.2 has just gone GA on other platforms

Questions