vms Software

X86 Porting Update

Camiel Vanderhoeven

November 30, 2016

vms Software

X86 Porting Update

This information contains forward looking statements and is provided solely for your convenience. While the information herein is based on our current best estimates, such information is subject to change without notice.

Camiel Vanderhoeven (1977)

Currently (2015 -)

- Software Engineer at VMS Software, Inc.
- X86 Architecture and C++ Expert
- Working mainly on the x86 port (and on Java 8)

Previously

- Architect and developer of the Avanti and FreeAXP emulators, and of the Open-Source ES40 emulator
- OpenVMS experience as a contractor in government, banking, automotive, healthcare, utility, transportation, weather prediction, steel, and nuclear industry

Personal

- Married, three kids
- Collecting old hardware (www.vaxbarn.com)
- Tinkering with Electronics and FPGAs
- Wine
- Twitter: @iamcamiel

Porting Play Book (The Plan)

Chapter 1 – Executable Images

- **Definition**: Register Mapping, Calling Standard extensions
- **Creation:** Compilers, Assembler
- **Action:** LIBRARIAN, LINKER, INSTALL, Image Activator
- **Analysis:** SDA, DEBUG/XDELTA, ANALYZE IMAGE, ANALYZE OBJECT

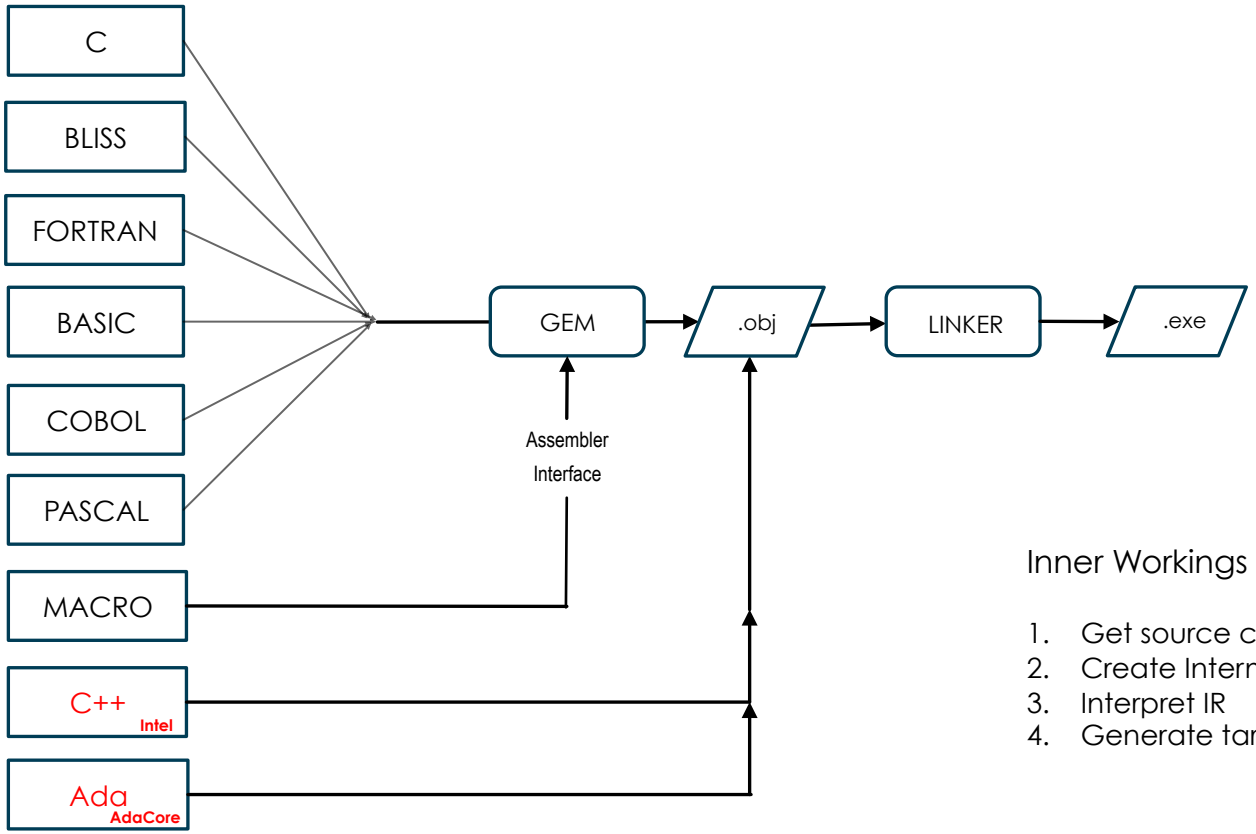
Chapter 2 – Architecture-Specific Needs (a.k.a. “The 5%”)

- Booting
- Interrupts, Exceptions
- Memory Management: protection types, access modes, address space, etc.
- Atomic Instructions
- Floating Point
- Special needs for code in assembler (e.g. VAX QUEUE instruction emulation)

Chapter 3 – Compiling and Linking Everything Else (a.k.a. “The 95%”)

- Large task but mostly mechanical
- Flush out any remaining ‘inter-routine linkage’ problems

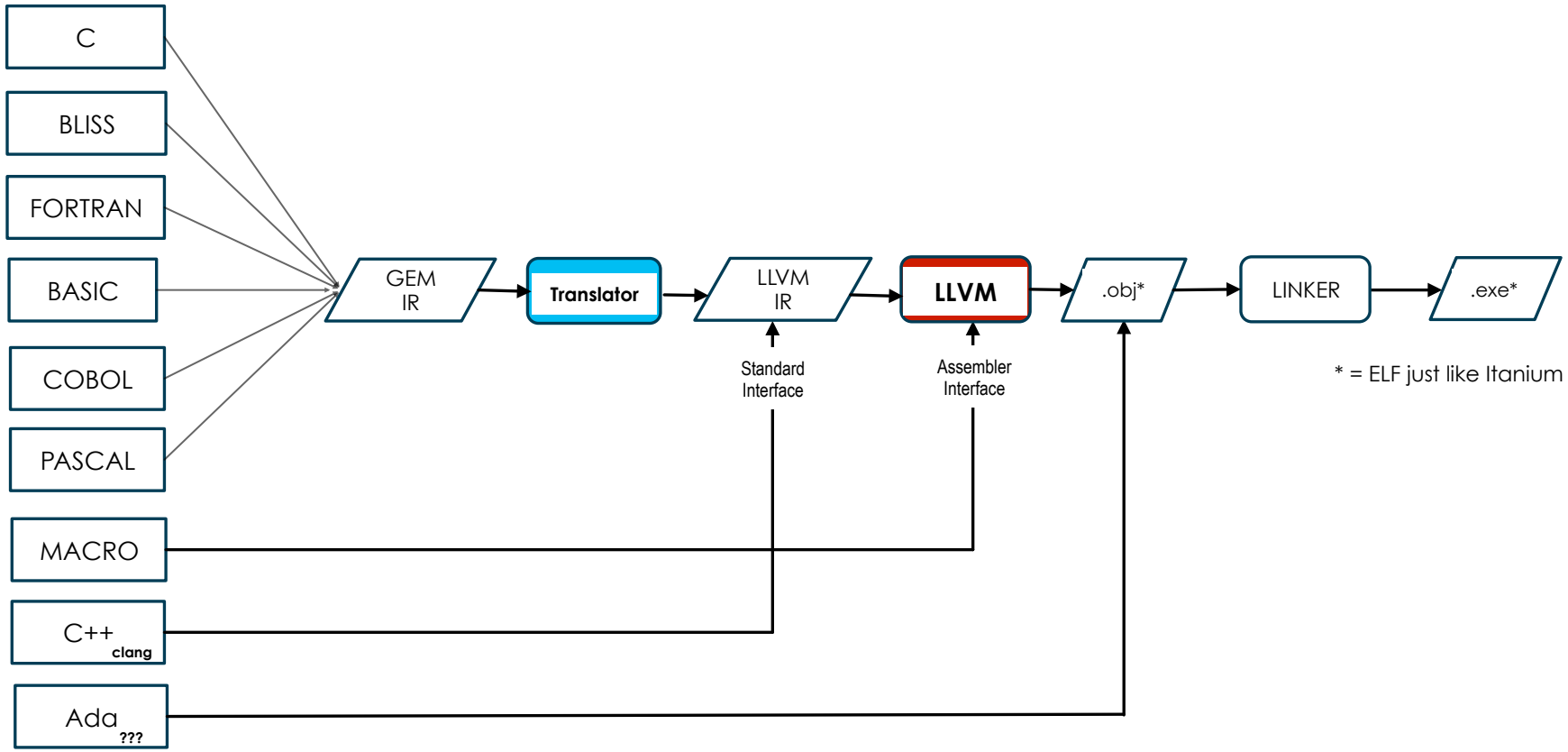
VMS Itanium Compilers and Image Building



Inner Workings of GEM

1. Get source code and command line directives
2. Create Intermediate representation (IR)
3. Interpret IR
4. Generate target object file

Future VMS Compiler Strategy



- Continue with current GEM-based frontends
- Use open source LLVM for backend code generation
- *Create internal representation (IR) translator*
- LLVM targets x86, ARM, PowerPC, MIPS, SPARC, and more

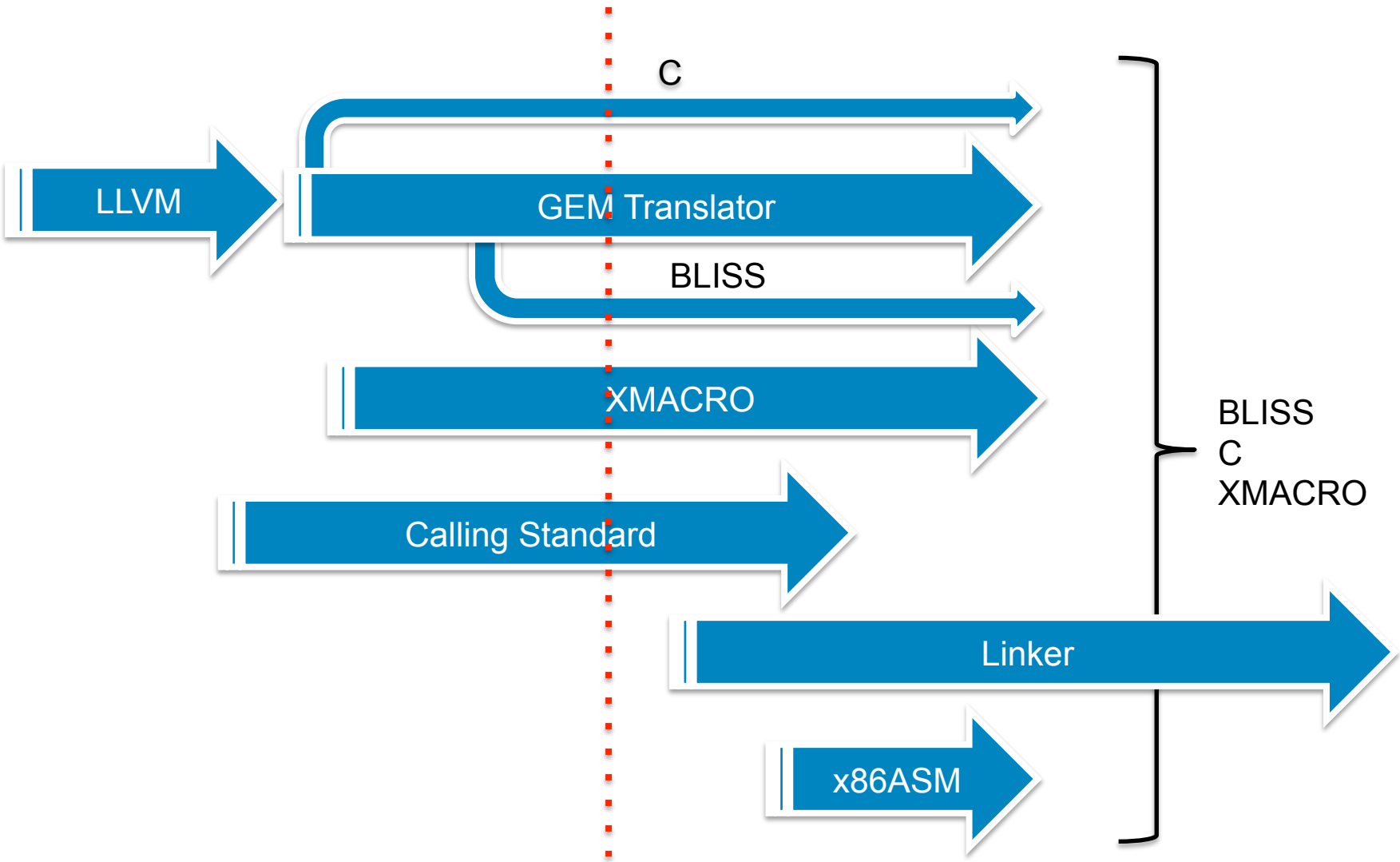
Code Generation

- GEM-to-LLVM (G2L) Converter
  - Started with C frontend
  - Focus on language constructs (and ignore builtins)
  - Converter continues to grow and improve
- C Compiler
  - DEC C Test Suite: 4000+ of 4200 compilation tests pass
  - Other tests:
    - 2049 compilation tests – 102 failures
    - 1336 run-time tests – 11 failures
  - Most failures due to things not yet implemented
  - For runtime tests
    - On VMS: LLVM outputs bitcode file
    - On linux: link with clang and run
  - Started compiling OS modules
  - Continuing to automate test analysis

Code Generation (continued)

- As G2L matured, started building BLISS compiler
  - Adding BLISS requirements not already exposed by C
  - Generalizing some C-centric assumptions in G2L
  - Streamlining G2L's parsing of GEM IR
- BLISS Compiler: Compiling simple programs
- MACRO Compiler
  - XMACRO uses different LLVM interface than used by C and BLISS
  - LLVM integration complete
  - Preliminary x86 code generation

“First Boot” (with Cross Tools) Images



Analysis Tools: Laying the Foundation

- x86 Instruction Set Decoder completed
  - 640 opcodes in total
  - Test ‘byte streams’ created for each instruction; developed/verified on linux
  - Used by SDA, DELTA/XDELTA, DEBUG, SCD, ANALYZE/OBJECT

- Evaluate LIB$IPF_CALLING_STANDARD routines; used by “stack walkers” and others
  - Invocation Context (current, previous)
  - Registers
  - Unwind data

Architecture-Specific Needs

- Boot Path
- Dump Kernel
- Memory Management
- Use of Assembler

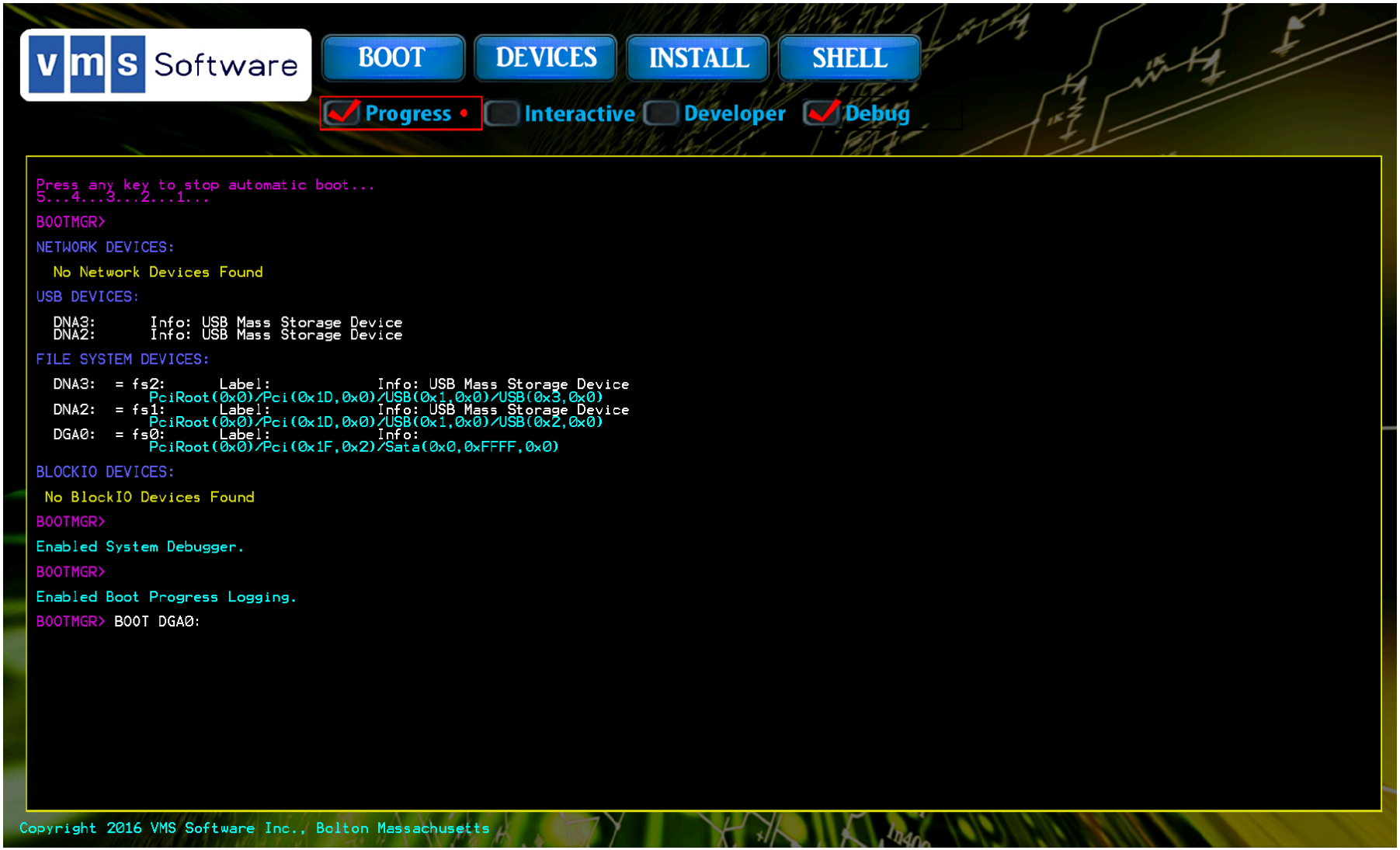
Boot Path

- Itanium: VMS_LOADER.EFI / IPB / SYSBOOT
- x86: VMS_BOOTMGR.EFI /  / SYSBOOT
- Goals:
  - Always boot from Memory Disk
  - Eliminate the need for boot drivers
  - Never touch the "primitive file system" again
  - Never write another xxBTDRIVER
- Results: Mission Accomplished!
- Now loading fewer files
- On-disk structure of the system disk is no longer a factor
- prior to loading the full file system
- Q: Without boot drivers, how do you write crash dumps?
  - A: The Dump Kernel

Dump Kernel

- A second, minimalist OS instance is loaded into memory during normal boot - but not booted
- Its memory allocation has a special tag
- As the system goes down:
  - The crashing primary kernel gathers the necessary information
  - BUGCHECK notifies the Boot Manager to boot the Dump Kernel
  - This is extremely fast since
    - the Dump Kernel is already in memory and
    - it only needs to boot as far as a specialized SYSINIT
  - The Dump Kernel writes the dump file using the run-time driver and initiates a shutdown

Sneak Preview: Graphical Bootmanager



Memory Management

- Review:
  - Two processor modes (kernel, user)
  - VMS fabricates two (executive, supervisor)
  - Four levels of page tables
  - VMS needs separate pages tables per mode
  - No PROBE instruction; look it up in page tables
  - Page sizes – 4KB, 2MB, 1GB
- SYSBOOT
  - Initial debugging: compile/link with Windows Visual Studio (just like the Boot Manager)
  - Some code has been converted from BLISS to C in order to make early progress
  - Memory bit map constructed based on memory descriptors passed from boot manager
  - Currently implementing PFN database and page tables

Running in Two Processor Modes

- Review:
  - x86 has four modes (rings) 0, 1, 2, 3 but they do not provide the strict hierarchy of memory access protection expected by VMS
  - Example: Cannot allow kernel write and prevent exec write
  - VMS will run in two modes: kernel (0) and user (3)
  - Supervisor and Executive modes implemented in software
- 2-Mode Prototyping on Itanium Completed
- Results:
  - PROBE emulation does PTE lookup
  - Did not boot and run complete system
  - Did get far enough to verify transitions to/from Supervisor mode with correct access and mode information
- Proved the methodology to be sound
- Implementation details will differ, but not greatly, on x86

SWIS & Friends

- Many related architecture-specific details in one place: Software Interrupt Services (SWIS), Exception, AST Delivery…
- Hides the details from the rest of VMS
  - Many aspects of the Calling Standard
  - Entering a more privileged mode
  - Interrupt handling
    - Software Interrupts
    - ASTs
    - External Interrupts
  - Saved state
  - Exception frames
  - Context switching
  - System service calling

SWIS & Friends

- Conceptually architecture independent – the code is specific for each platform but it performs the same logical functions
- The largest single piece of concentrated assembler code apart from IMATHRTL
- Recently Completed:
  - Design for accessing per-CPU data
  - Basic design for mode changes (system services and interrupts)
- In progress:
  - Detailed design for system service calling
  - Detailed design for interrupt and exception handling

Virtual Machines

- Development/test environments:
  - KVM / CentOS
  - XEN
  - VirtualBox
  - VMware ESXi 6.0
  - Vmware Workstation 12
  - VMware Fusion
- Used so far for
  - Debugging VMS_LOADER.EFI
  - Prototyping and experimenting
- Paravirtualized storage drivers
  - Starting point: HPVM driver
- Paravirtualized network drivers

Various

- VAX/Alpha/IA64 conditionalized code
- Non-standard calling sequences
- Evaluated Ada code
  - Plan to rewrite ACME in C
  - Plan to rewrite Security_Server in C++

For more information, please contact us at:

RnD@vmssoftware.com

VMS Software, Inc. • 580 Main Street • Bolton MA 01740 • +1 978 451 0110