November 2017

Technical Update: VAFS and the Port to x86-64

vms Software

Camiel Vanderhoeven

About Me

Currently (2015 -)

- Software Engineer at VMS Software, Inc.
- X86 Architecture and C++ Expert
- Working on the x86 port

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 production, and nuclear industry

Personal

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

VMS Advanced File System

History of VAFS

- Started by DEC engineers in Edinburgh, Scotland in 1996
  They previously did Spiralog
- Designed to run on multiple operating systems (VMS, Windows NT)
- Moved to VMS Engineering (Nashua, NH) in 1998
- Developed on and off until 2004
- Restarted by VSI in 2016

Need for a new file system

- Volume size limited to 2TB
- Performance
- Number of files on disk and in a directory is limited

ODS-2/5 Limitations

- 32 bit VBN & LBN
- 512 byte block dependency
- Sequential directory format
  - Square law delete performance
- "Careful write" update strategy
  - Deferred write requires a log for safety
- Bitmap based allocation
  - Linear solution to an exponential problem
- Code entropy

Storage Scale

- 32 bit LBN = 2TB
- >2TB hard drives have been available for a while
- >2TB logical volumes have been possible for a long time
- **Any** solution requires an on disk structure change

Storage Scale – Market Demands (2004)

- Mormon church genealogical database
  - Projected 50PB several years ago
- Medical imaging
  - 1 digitized X-ray = 1GB
  - 1 CAT scan = 100-200GB
- Russian Customs
  - 120TB database, 1TB / week log file
  - Planned video archive requires 2PB

File System Performance

- Typical Unix file system is 10x faster than VMS for open/close/create/delete
- Deferred write (both user data & FS metadata)
- Write-ahead logging in current file systems
- Shorter code stack – no RMS/XQP layering
- Simpler file naming semantics (no logical names)
- No shared-everything cluster model
  - Distributed locking
  - Thrashing updates

Benefits of VAFS

Performance

- Write behind caching
- Metadata writes to sequential log

  “Metadata” being (in ODS-2/5 terms) INDEXF.SYS, *.DIR, QUOTA.SYS, ACLs

Benefits of VAFS

Extensibility

- Small number of basic concepts used as building blocks (List Pages, Streams, Trees)

Benefits of VAFS

Maintainability

- Small number of basic concepts used as building blocks (List Pages, Streams, Trees)
- Written in C (no MACRO, no BLISS)

vms

Benefits of VAFS

Scalability

- Large disk support (64-bit LBNs)
- More files on volume
- More files in a directory
- Space allocation performance improvement
- Recovery time after crash (MOUNT /REBUILD)

VAFS vs. ODS-2/5: Similarities

DCL utilities (COPY, DELETE, EDIT, MOUNT, INIT, etc…)

User-visible interfaces and upper-layer data structures

- FCB’s
- WCB’s
- ACP-QIO Interface
- XFC
- ACL’s
- Disk quotas
- File ID’s
- RMS
- File sizes limited to 1TB (RMS 32-bit limitation)
- Host-Based Volume Shadowing

VAFS vs. ODS-2/5: Differences

- On-disk structure for metadata is completely different!

VAFS vs. ODS-2/5: Differences

- On-disk structure for metadata is completely different!
- All metadata writes bounce through a recovery log before being written to destination LBN’s

VAFS vs. ODS-2/5: Differences

- On-disk structure for metadata is completely different!
- All metadata writes bounce through a recovery log before being written to destination LBN's
- File deletion: VAFS moves "deleted" files to [SYSDELETE], then deleted in background. Allows deletion of large files to be broken up into smaller atomic transactions.

VAFS vs. ODS-2/5: Differences

- On-disk structure for metadata is completely different!
- All metadata writes bounce through a recovery log before being written to destination LBN’s
- File deletion: VAFS moves “deleted” files to [SYSDELETE], then deleted in background. Allows deletion of large files to be broken up into smaller atomic transactions.
- [SYSHIDDEN]: All files must be in a directory

VAFS vs. ODS-2/5: Differences

- On-disk structure for metadata is completely different!
- All metadata writes bounce through a recovery log before being written to destination LBN's
- File deletion: VAFS moves "deleted" files to [SYSDELETE], then deleted in background. Allows deletion of large files to be broken up into smaller atomic transactions.
- [SYSHIDDEN]: All files must be in a directory
- File structure metadata is organized and stored outside of file system itself (no INDEXF.SYS, QUOTA.SYS, etc. …)

VAFS vs. ODS-2/5: Differences

- On-disk structure for metadata is completely different!
- All metadata writes bounce through a recovery log before being written to destination LBN’s
- File deletion: VAFS moves “deleted” files to [SYSDELETE], then deleted in background. Allows deletion of large files to be broken up into smaller atomic transactions.
- [SYSHIDDEN]: All files must be in a directory
- File structure metadata is organized and stored outside of file system itself (no INDEXF.SYS, QUOTA.SYS, etc. …)
- VAFS uses “disk pages” of 2048 bytes as unit of operation (may be increased to 4096)

VAFS vs. ODS-2/5: Differences

- On-disk structure for metadata is completely different!
- All metadata writes bounce through a recovery log before being written to destination LBN’s
- File deletion: VAFS moves “deleted” files to [SYSDELETE], then deleted in background. Allows deletion of large files to be broken up into smaller atomic transactions.
- [SYSHIDDEN]: All files must be in a directory
- File structure metadata is organized and stored outside of file system itself (no INDEXF.SYS, QUOTA.SYS, etc. …)
- VAFS uses “disk pages” of 2048 bytes as unit of operation (may be increased to 4096)
- No volume sets, bad block handling, geometry sensitivity, placed allocation

VAFS vs. ODS-2/5: Differences

- On-disk structure for metadata is completely different!
- All metadata writes bounce through a recovery log before being written to destination LBN's
- File deletion: VAFS moves "deleted" files to [SYSDELETE], then deleted in background. Allows deletion of large files to be broken up into smaller atomic transactions.
- [SYSHIDDEN]: All files must be in a directory
- File structure metadata is organized and stored outside of file system itself (no INDEXF.SYS, QUOTA.SYS, etc. …)
- VAFS uses "disk pages" of 2048 bytes as unit of operation (may be increased to 4096)
- No volume sets, bad block handling, geometry sensitivity, placed allocation
- Cannot be a system disk on IA64 or Alpha (yes on X86)

A newly-initialized VAFS disk

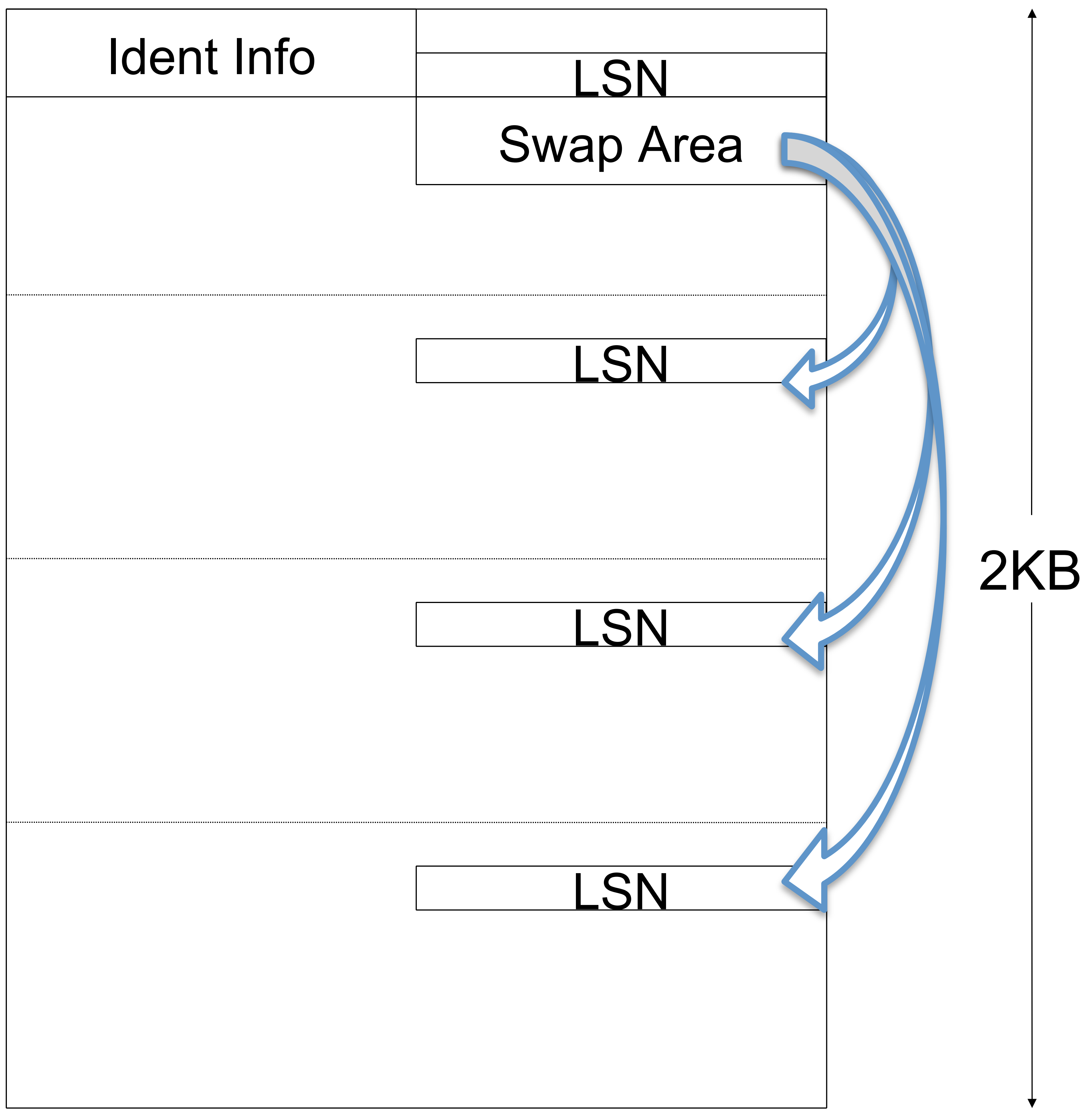
```
Directory $1$DGA220:[0,0]

000000.DIR;1            0/0  31-JUL-2017 11:45:03.40  (RWED,RWED,RE,E)
SYSDELETE.DIR;1         0/0  31-JUL-2017 11:45:03.31  (RWED,RWED,RWED,RWED)
SYSHIDDEN.DIR;1         0/0  31-JUL-2017 11:45:03.40  (RWE,RWE,RE,)
SYSQUOTA.DIR;1          0/0  31-JUL-2017 11:45:03.31  (RWED,RWED,RWED,RWED)
SYSRECOVERY.DIR;1       0/0  31-JUL-2017 11:45:03.31  (RWED,RWED,RWED,RWED)
```

Note the lack of ODS-2/5 style metadata files

VAFS: How it works

- VAFS is log-based, not log-structured (Spiralog)
- All file system metadata writes are first written to a transaction log before moved to destination LBNs
- Metadata encapsulated in building block data structures like
  - List Pages
  - Streams
  - Trees
  - Key-list value pairs

Disk Page

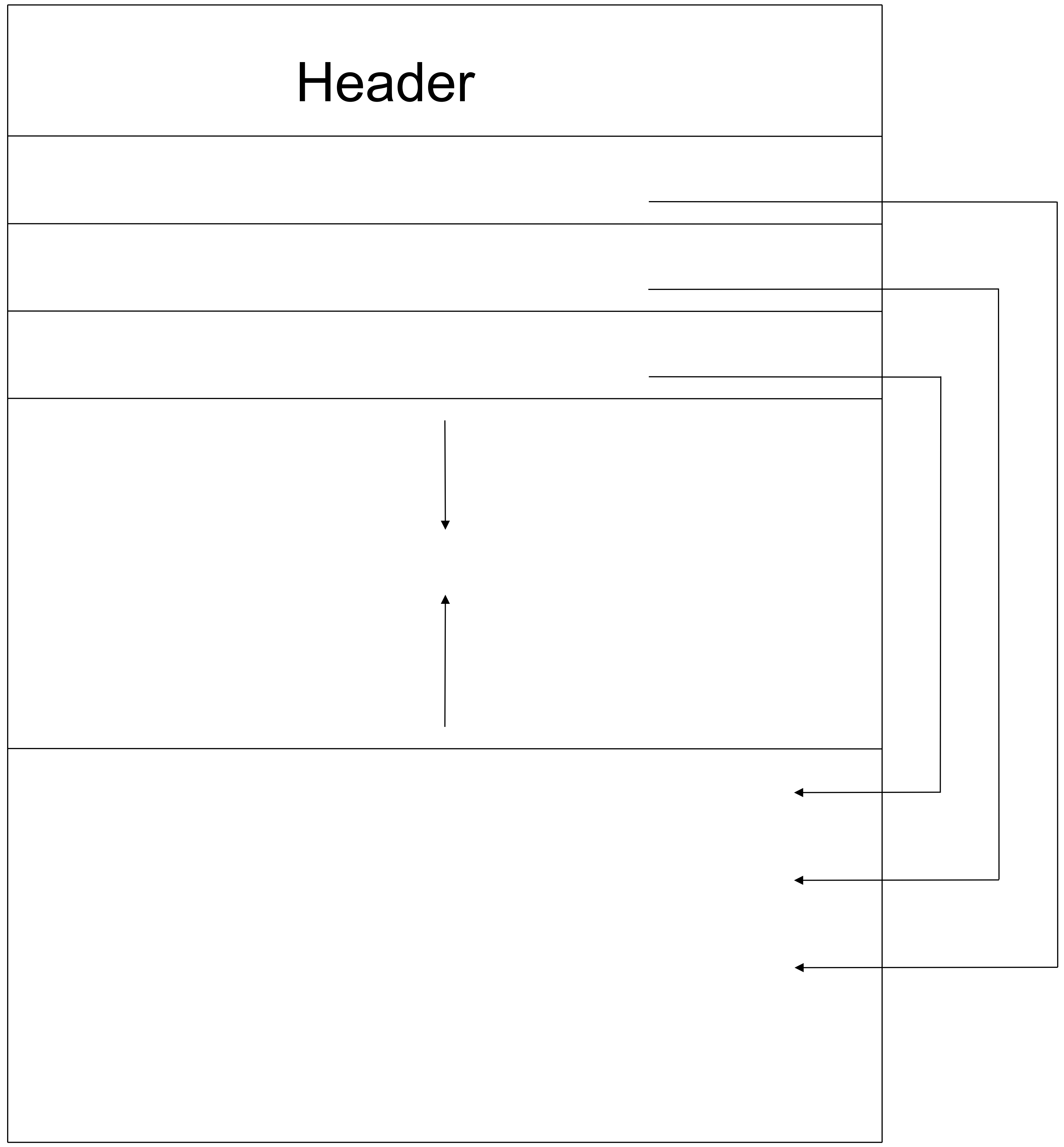


List Pages

- Ordered array of key-length value items. Most VAFS metadata is stored in LIST PAGES
- LIST PAGES have SLOTS which contain STREAMS
- Aggregated into TREES; leaf pages store the actual data
- Located by index entry in a parent list page
- Examples of LIST PAGES as TREES
  - Attributes (ODS-2/5 file header == VAFS tree)
  - Directories
  - Extent maps

List Page

Attribute
Value
Pairs



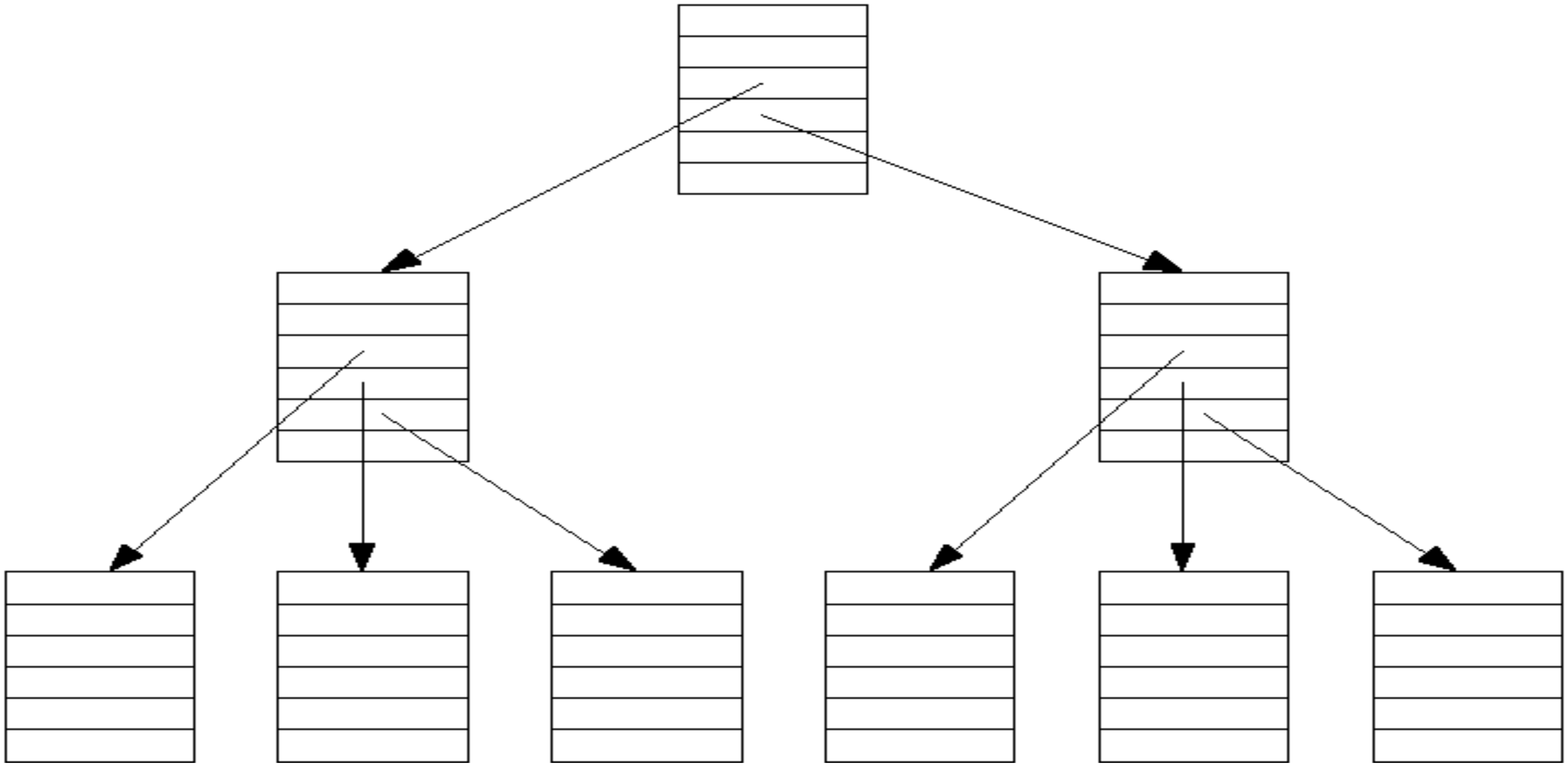
Fixed
Size Key
Prefix

Key
Remainder
& Value

Streams

- Direct: stored in a List Page as an attribute value in a SLOT
- Mapped: stored in List Pages via an extent tree. Root of the tree is an attribute value
- Examples of streams
  - Index file
  - Storage Bitmap
  - FID bitmap
  - Recovery log

Tree



Examples of Trees

- Directories
- Storage Bitmap Index
- FID Bitmap Index
- Storage Allocation Cache
- FID allocation Cache

Directory

- Special file type
- Directory content is a special file attribute, stored as a tree
- Directory entry
  - Key = file name, normalized Unicode + case flags
  - Value = file ID

Bitmap

- Used to allocate file IDs and free blocks
- Organized in page-size segments
- Extensible tree structure

How do we make sense of this stuff?

$ DUMP/XFS is the answer (without it, we'd be doomed!)

How do we make sense of this stuff?

$ DUMP/XFS is the answer (without it, we’d be doomed!)

Thanks, Andy!

VAFS: Let’s get started

$ **INIT <device name> /STRUCTURE = 6 <label>**

- Writes an ODS-2/5-compatible home block with a tiny bit of ODS-6 info
- Does not write much of the file system infrastructure

$ **MOUNT <device name> <label>**

- “First Mount” of a VAFS volume does most of the initialization
- Key structures include Home Page, Recovery Log, storage bitmap

VAFS Home Page

```
$ DUMP/XFS /BLOCK= (START:320,COUNT:4) <device>

XFS Metadata Page
XFS page header

Page size (blocks):     4 used, 4 allocated
Page address:           LBN 320
Page state:             AllocSeq = 503, UpdateSeq = 30, LSN = 57
Parent file number:     5
Page log flags:         file lock

XFS list page header

Page type:              attributes
Page flags:             <none specified>
Structure version:      1/1 (major/minor)
List page size:         1984 bytes, 12 slots in use, 0 deleted
Free space (bytes):     48 free on top, 0 deleted
```

Index File Info

```
Formatted List Page Slots

List Page Slot 0, flags:  <none specified>
    Stream type:       unspecified
    8 byte key:        (1) - volume attributes  (#define XFS_ATTR_VOLUME   1       /* volume attributes */)

    208 byte value:
 00000000 00000000 00000000 00000800 00000800 00000200 E944A850 01020101 ....P¨Dé......................... 000000
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 000020
 00000000 00500000 00000000 00000000 00000000 00000000 00000000 00000000 ..........................P..... 000040
 00000000 00000040 00000000 00500000 00000000 00500000 00000000 00500000 ..P.......P.......P.....@....... 000060
 00B1ED7A E944CF60 00000000 00000D80 00000D80 00000000 00000000 00000040 @.......................`ÏDézí±. 000080
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 0000A0
                                     00000000 00000000 00000000 00000000 ................................ 0000C0

List Page Slot 1, flags:  mapped
    Stream type:       metadata
    8 byte key:        (2) - index file stream  (#define XFS_ATTR_INDEX   2       /* index file stream .. */)
    Formatted extent list on following page

List Page Slot 2, flags:  <none specified>
    Stream type:       unspecified
    8 byte key:        (3) - index file stream info (#define XFS_ATTR_INDEX_INFO  3 /* .. and stream attributes */)
    Allocated length:  131072 (0000000000020000) bytes (256 blocks)
    Data length:       131072 (0000000000020000) bytes (256 blocks)
    Highest written:   0 (0000000000000000) bytes (0 blocks)
```

Port to X86-64

vms Software

Agenda

- **Previous VSI Boot Camps**
  - 2014: Dusted off the “Porting Play Book”
  - 2015: Described the basic plan and a few details
  - 2016: Added more plan details and described the beginnings of implementation
- **Today**
  - Focus on implementation progress
  - What was/is difficult?
  - Work progress and what remains

Boot Contest

- **What**
  - Boot OpenVMS
  - Login
  - Use DIR command to get a directory listing
- **Details**
  - To participate, send email to Sue Skonetski and fill in a survey
  - Guidance: Q1 2018

System

Architecture-Specific Work

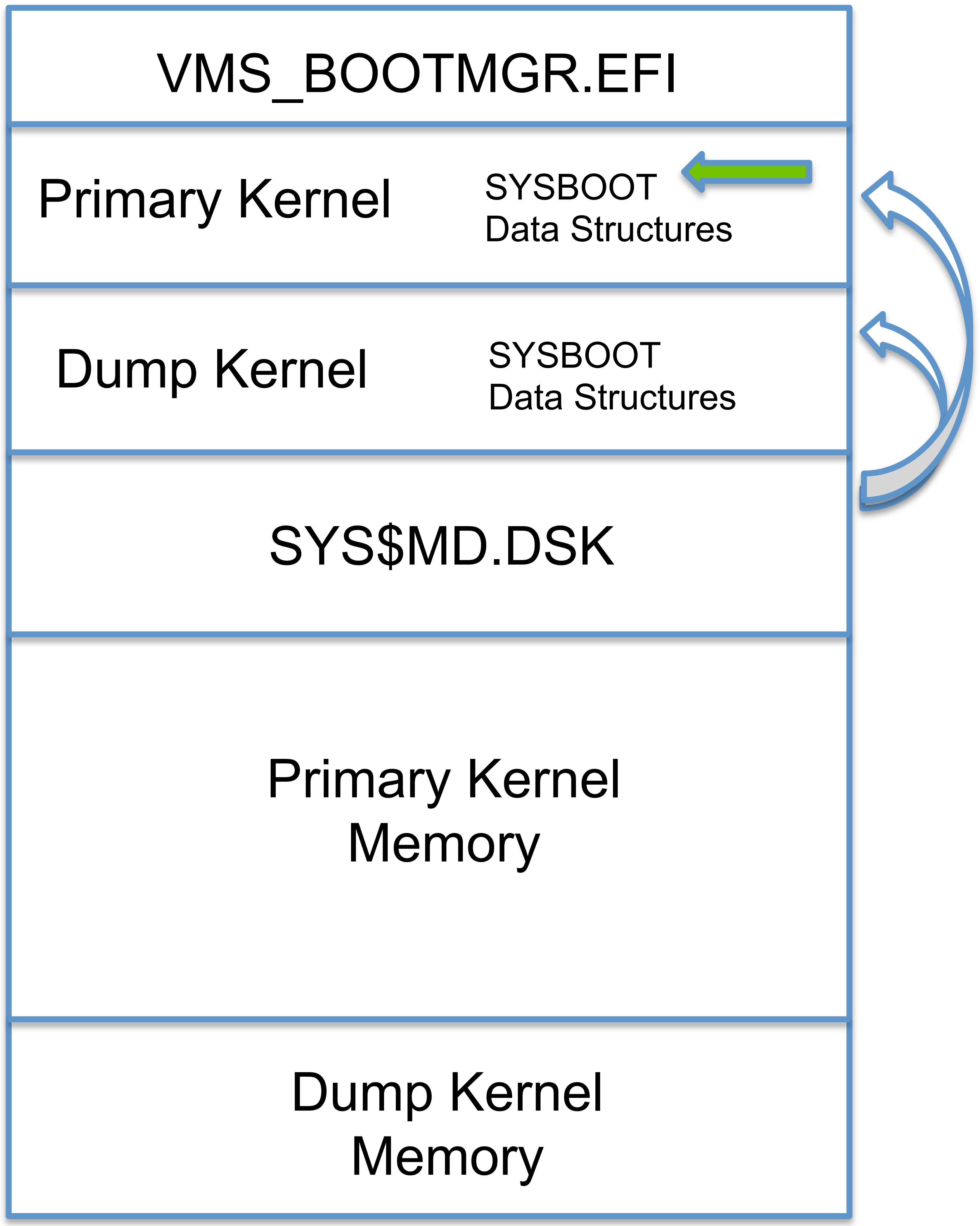
Boot Manager

- Select Console Mode
- Analyze Devices
- Auto-Action or Enter Command Loop
- Boot System via Memory Disk
- Primary Kernel
- Dump Kernel
- Enter Console Services

What is MemoryDisk?

- ODS-5 container file with a 3-partition disk image
- Built and maintained by OpenVMS utilities
- Contains kernel files with SYMLINKS to active system
- Shared by Primary Kernel and Dump Kernel
- Located on any accessible device, including network

Status: In use on multiple platforms.



BOOT DEVICES INSTALL SHELL

MESSAGES: PROGRESS SYSBOOT EXECINIT SYSINIT VERBOSE

BOOT MODES: DETAIL> XLDELTA> XDELTA> SYSBOOT> NETBOOT>

AUTOACTION: HALT

```
BOOT RELATED COMMANDS:

BOOT <device> <sysroot> <bootflags> <"Comment in quotes, max 64 characters">
     BOOT               - Boots with default device, system root and boot flags.
     BOOT DKA100        - Boots DKA100 with default system root and boot flags.
     BOOT DKA100 0      - Boots DKA100 with system root 0 and default boot flags.
     BOOT DKA100 2 20000 - Boots DKA100 with system root 2 and boot flags 0x20000.
     BOOT #3            - Boots the third option in the Boot Options List. See OPTIONS.
FLAGS <value>       - Show / Set <value> VMS Boot Flags. Expressed in hexidecimal.
ROOT <value>        - Show / Set <value> VMS System Root. Expressed in hexidecimal.
OPTIONS             - Displays the VMS Boot Options List showing the last ten unique boot commands.
                    - If the file: VMS_OPTS.TXT exists, it will be used as the option list.
AUTOACTION          - HALT, BOOT or RESTART. Automatic action to take when BootManager is invoked.
DEVICES             - Lists VMS Boot Devices and their UEFI File System equivalents.

MESSAGE RELATED COMMANDS:

PROGRESS            - Enables Boot Progress messages. NOPRO to disable.
SYSBOOT             - Enables SYSBOOT messages. NOSYSB to disable.
EXECINIT            - Enables EXECINIT messages. NOEXEC to disable.
SYSINIT             - Enables SYSINIT messages. NOSYSI to disable.
VERBOSE             - Enables Extended boot messages. NOVERB to disable.

MODE RELATED COMMANDS:

DETAIL              - Enables detailed BOOTMGR> conversation. NODET to disable.
XLDELTA             - Enables XLD> debugger and sets SYSBOOT breakpoint. NOXLD to disable.
XDELTA              - Enables loading of XDELTA debug execlet and initial breakpoint. NOXDE to disable.
SYSPROMPT           - Enables SYSBOOT> conversation. NOSYSP to disable.
NETBOOT             - Enables NETBOOT> conversation. NONET to disable.
DUMP                - Enables the VMS Crash Dump Kernel. NODUMP to disable.
DUMPDEVICE          - Sets or Shows the VMS Dump Device.
DUMPFLAGS <value>   - Show / Set <value> VMS Dump Kernel Boot Flags. Expressed in hexidecimal.

DIAGNOSTIC COMMANDS:

DEVELOPER           - Enables VSI Developer Mode. NODEVEL to disable. Function varies.
PCI                 - Show PCI Device list.
USB                 - Show USB Device list.
NETWORK             - Show NETWORK Device list.
APIC                - Show APIC (Interrupt Controllers) list.
SMBIOS              - Show SMBIOS (System Management) data.
GRAPHICS            - Enables Graphics diagnostics. NOGRAPH to disable.
MEMCHECK            - Enables Memory Config diagnostics. NOMEM to disable.
DEVCHECK            - Enables Device Config diagnostics. NODEV to disable.
KEYMAP              - Enables Keyboard Service diagnostics.
<PAGE>
```


vms Software
BOOT
DEVICES
INSTALL
SHELL
MESSAGES:
PROGRESS
SYSBOOT
EXECINIT
SYSINIT
VERBOSE
BOOT MODES:
DETAIL>
XLDELTA>
XDELTA>
SYSBOOT>
NETBOOT>
AUTOACTION: HALT
%VMS_BOOTMGR-I-REVISION: X9.0-0 Build 9 - Oct 9 2017
ENABLED: Progress messages.
ENABLED: Boot Manager interaction.
%VMS_BOOTMGR-I-DEVICE, Configuring System Devices...
+ 1 Network Devices (Protocol UNDI)
+ 3 File System Devices
+ 12 Block IO Devices
%VMS_BOOTMGR-I-DEVICE, Configuring Peripheral Devices...
Scanning PCI Bus Range: (00:04:1F:07)...
Added 11 additional PCI Devices discovered by bus scan.
Configured 14 PCI/e Devices.
Assigning VMS Device Names...
Assigning VMS Controller Letters...
Assigning VMS Unit Numbers...
Assigning VMS Network Devices...
Retrieving Device Information...
BOOTMGR DEVICE: DNA0 (fs0)
BOOTMGR> PAGE
ENABLED PAGE scrolling mode.
BOOTMGR> B
BOOT DESTINATION DEVICE: DNA0 (fs0) VMSUSBSTICK
DEFAULT BOOT COMMAND: BOOT DNA0 0 01000034
%VMS_BOOTMGR-W-MAIN, DISABLED Crash Dumps.
LOAD PATH:
PciRoot(0x0)/Pci(0x1D,0x0)/USB(0x1,0x0)/USB(0x4,0x0)/HD(1,GPT,40985391-9DF8-11E7-B56F-9C8E9935AD96,0x10CE0,0x3E800)
%VMS_BOOTMGR-I-MAIN, Allocating Kernel Memory...
ADDRESS SPACE ALLOCATION:
MAIN KERNEL SYSBOOT: PA Floor: 0x00400000, Ceiling: 0x006FFFFF, Size: 0x00300000 (3MB)
MAIN KERNEL HWRPB: PA Floor: 0x00800000, Ceiling: 0x008FFFFF, Size: 0x00100000 (1MB), Actual: 0x00058000
MEMORYDISK: PA Floor: 0x01400000, Ceiling: 0x113FFFFF, Size: 0x10000000 (256MB)
KERNEL_BASE_STRUCTURE: PA Floor: 0x00200000, Ceiling: 0x002FFFFF
+ MAIN KERNEL HWRPB PA: 0x00800000
<PAGE>
Copyright 2017 VMS Software Inc., Bolton Massachusetts, USA

```
%%%%%%% VSI OpenVMS (tm) x86-64 Operator Console %%%%%%%

Welcome to VSI OpenVMS
Parameter passed from the boot manager to SYSBOOT:
 HWRPB:         0x00000000.00800000 size 0x00000000.00058000

Key locations and sizes:
 Kernel Base:   0x00000000.00200000 size 0x00000000.00100000
 ConIoTable:    0x00000000.D93C5F18
 System Table:  0x00000000.D93A8F18
 SYSBOOT:       0x00000000.00400000 size 0x00000000.00300000
 Memory Disk:   0x00000000.01400000 size 0x00000000.10000000
SWRPB address 0x00415030
SWRPB flags address 0x00415048
Entering boo$sysboot_x86
Entering boo$init_swrpb
Leaving boo$init_swrpb
Entering boo$checkout_cpu
Leaving boo$checkout_cpu
%SYSBOOT-I-MEMDISKMOUNT, Boot memory disk mounted
%SYSBOOT-I-LOADPARAM, Loading parameter file X86_64VMSSYS.PAR
Entering bfs$open_file
Leaving bfs$open_file
Parameter file is 11264 bytes long (22 blocks)
boo$loadBootfile: loading paramter file
boo$usefile: Parameter file read in successfully
%SYSBOOT-I-LOADFILE, Loaded file [SYS0.SYSEXE]X86_64VMSSYS.PAR
%SYSBOOT-I-MEMDISKDISMOUNT, Boot memory disk dismounted
Entering boo$init_memalc
Entering boo$init_memory_variables
Leaving boo$init_memory_variables
Entering boo$calc_max_pfn
Best PXML memory ranges: 20200000 40003FFF 0 21FDFFFFF
minbitPFN 20200, maxbitPFN 40003, minPFN 0, maxPFN 21FDFF, memsize 1F9884
Leaving boo$calc_max_pfn
Entering boo$build_page_tables
MAXPHYADDR is 36 bits, Max linear address is 48 bits
Entering boo$find_free_pfns req_pages 1
Leaving boo$find_free_pfns
PT space base addr ffff800000000000
Leaving boo$build_page_tables
Entering boo$build_allocation_bitmap
Entering boo$find_free_pfns req_pages 4
Leaving boo$find_free_pfns
Entering boo$check_va
Leaving boo$check_va
%SYSBOOT-I-ALLOCMAPBLT, Allocation bitmap built
Leaving boo$build_allocation_bitmap and boo$init_memalc
Press Enter to continue
 Creating the PFN memory map
Entering boo$create_pfn_memory_map
Entering sort_syi_build_pfn_map
count 11FE00, phypgcnt 1F9884, mem_limit FFFFFFFFFFFFFF00
Leaving sort_syi_build_pfn_map
Leaving boo$create_pfn_memory_map
%SYSBOOT-I-PFNMAP, PFN memory map created
Creating the S0 space page tables
Entering boo$init_s0_space
Leaving boo$init_s0_space
S0 space page tables created
Remapping memory disk to S2 space
Entering boo$map_memorydisk
Memory disk pa = 0000000001400000, size = 10000000 bytes
<PAGE>
```

Always Boot from Memory Disk – Why?

- Why did we undertake this **large** and **complicated** project?
  - Increase maintainability - one boot method regardless of source device
  - Eliminate writing of OpenVMS boot drivers
  - Eliminate modifying (or replacing) primitive file system
- Other Factors
  - Take advantage of UEFI capabilities, especially I/O
  - This opportunity may never exist again

Status: 95+% done, only final details of booting into a cluster remain

Dump Kernel

- MemoryDisk dictated the need for a new way to handle crash dump
- User-mode program with some kernel-mode routines
- It “replaces” STARTUP.COM in the standard boot sequence
- Everything the Dump Kernel needs is in the MemoryDisk
- Writes raw/compressed full/selective dumps to system disk or DOSD

Status: We have debugged everything we can on Itanium and will do final verification work on x86 when enough of OpenVMS is running.

```
$ run crash_test

Initiating crash at 28-AUG-2017 13:00:52.98...

%IPB-I-DUMPBOOT, Booting the Crash Dump Kernel




**** OpenVMS IA64 Operating System XE60-T7Y - BUGCHECK ****

** Bugcheck code = 000003C4: SSRVEXCEPT, Unexpected system service exception
** Crash CPU: 00000000    Primary CPU: 00000000    Node Name: POTATO
** Highest CPU number:    00000003
** Active CPUs:           00000000.0000000F
** Current Process:       "SYSTEM"
** Current PSB ID:        00000001
** Image Name:            $1$DGA10:[SYS0.SYSCOMMON.][SYSMGR]CRASH_TEST.EXE;3
** Crash Time:            28-AUG-2017 13:00:52.98

** Dumping error logs to the system disk ($1$DGA10:)
** Error logs dumped to $1$DGA10:[SYS0.SYSEXE]SYS$ERRLOG.DMP
** Dumping memory to the system disk ($1$DGA10:)

**** Starting compressed selective memory dump at 28-AUG-2017 13:00:58.41 ****
...............................................................................
...............................................................................
..........
** System space, key processes, and key global pages have been dumped.
** Now dumping remaining processes and global pages...
..........
** Memory dumped to $1$DGA10:[SYS0.SYSEXE]SYSDUMP.DMP

**** Completed compressed selective memory dump at 28-AUG-2017 13:01:16.02 ****
** Time to initiate memory dump:            5.42
** Time to write memory dump:               17.61

**** Primary HALTED with code HWRPB_HALT$K_WARM_REBOOT
```

Displays time, then $CMKRNL & ACCVIO

New

Error log details

Starting timestamp

Ending timestamp

Statistics

Memory Management

- Challenges
  - OpenVMS-style page protections for kernel, exec, super, user
  - Designing for 4-level and 5-level paging
  - 2MB and 1GB pages
  - Change to traditional paging mechanism and access
- Status
  - SYSBOOT: done (compiled and linked in x-build)
    - Get memory descriptors from the boot manager
    - Set up paging mechanisms
  - Next up:
    - Create general page management routines
    - Fix code that manages pages on their own

Everything you know about memory management is the same

50

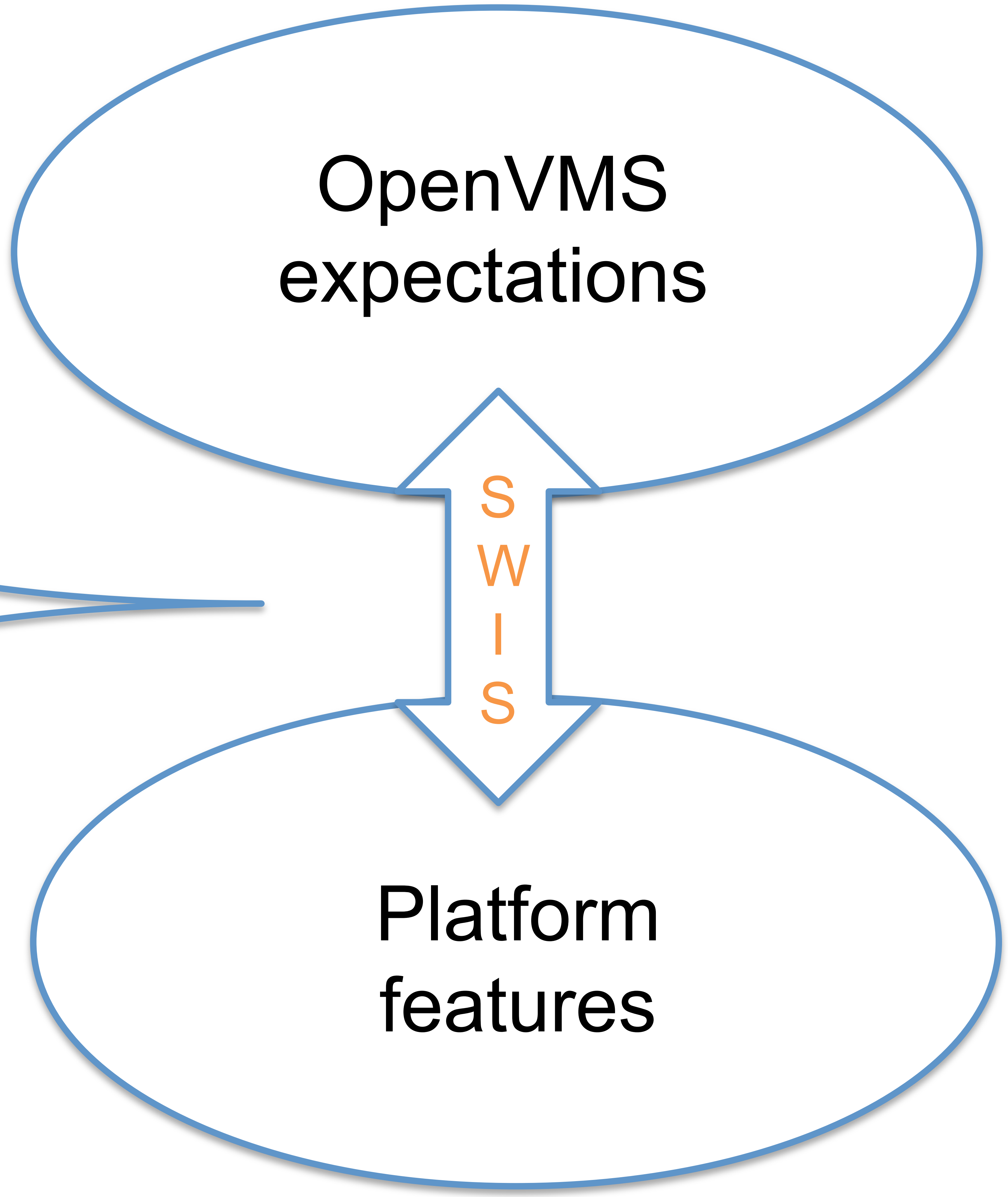
your unprivileged application knows

Almost everything ~~you know~~ about memory management is the same

51

Software Interrupt Services

- New Data Structures
- MTPR / MFPR
- Exceptions
- System Service Dispatching
- Interrupts
- ASTs
- Mode Switching
- Context Switching
- Performance Builds



OpenVMS Assumes Things…

- VAX/VMS was designed in tandem with the VAX hardware architecture.
- Where desirable, hardware features were added to satisfy the OS' needs.
- A lot of OS code was written to make use of these hardware features.

What are these Assumptions?

- 4 hardware privilege modes
- Each with different page protections
- And with their own stack
- 32 Interrupt Priority Levels
- 16 for Hardware Interrupts
- 16 for Software Interrupts
- Software Interrupts are triggered immediately when IPL falls below the associated IPL
- Asynchronous Software Trap (AST) associated with each mode, triggered immediately when IPL falls below ASTDEL (equally or less privileged mode)
- The hardware provides atomic instructions for queue operations
- The hardware provides a set of architecturally defined Internal Processor Registers (IPRs)

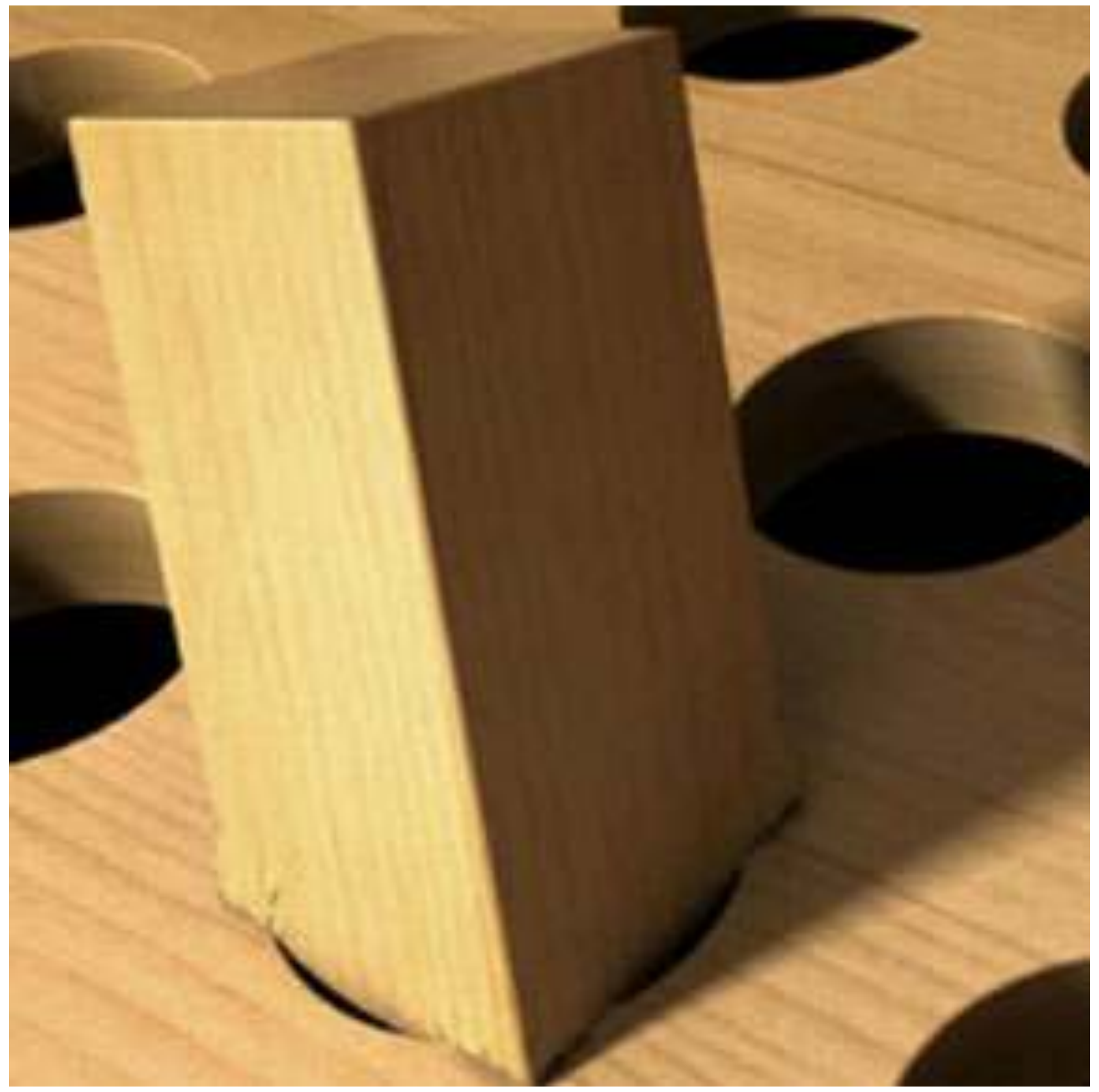
vms

How does Alpha meet these Assumptions?

- Alpha is a very clean RISC Architecture
- But OpenVMS was definitely in the Alpha Architecture designers' minds
- The 4 modes OpenVMS needs are part of the basic Alpha architecture
- PALcode, code supplied by firmware that has more privileges than even kernel mode, and which is uninterruptible, provides the flexibility to implement OS specific features
- IPLs, Software Interrupts and ASTs are implemented through a combination of hardware support and PALcode
- Atomic queue instructions are provided by PALcode
- PALcode also provides the mapping from IPRs as expected by OpenVMS to the hardware implementation's IPRs

So how about Itanium Hardware?

- Very different story, Itanium's design was finished before OpenVMS as an OS was considered
- Offers the 4 modes OpenVMS needs
- The TPR (Task Priority Register) provides an IPL-like mechanism for hardware interrupts only
- No compatible software interrupt mechanism or ASTs
- No atomic queue instructions
- No OpenVMS-compatible IPRs

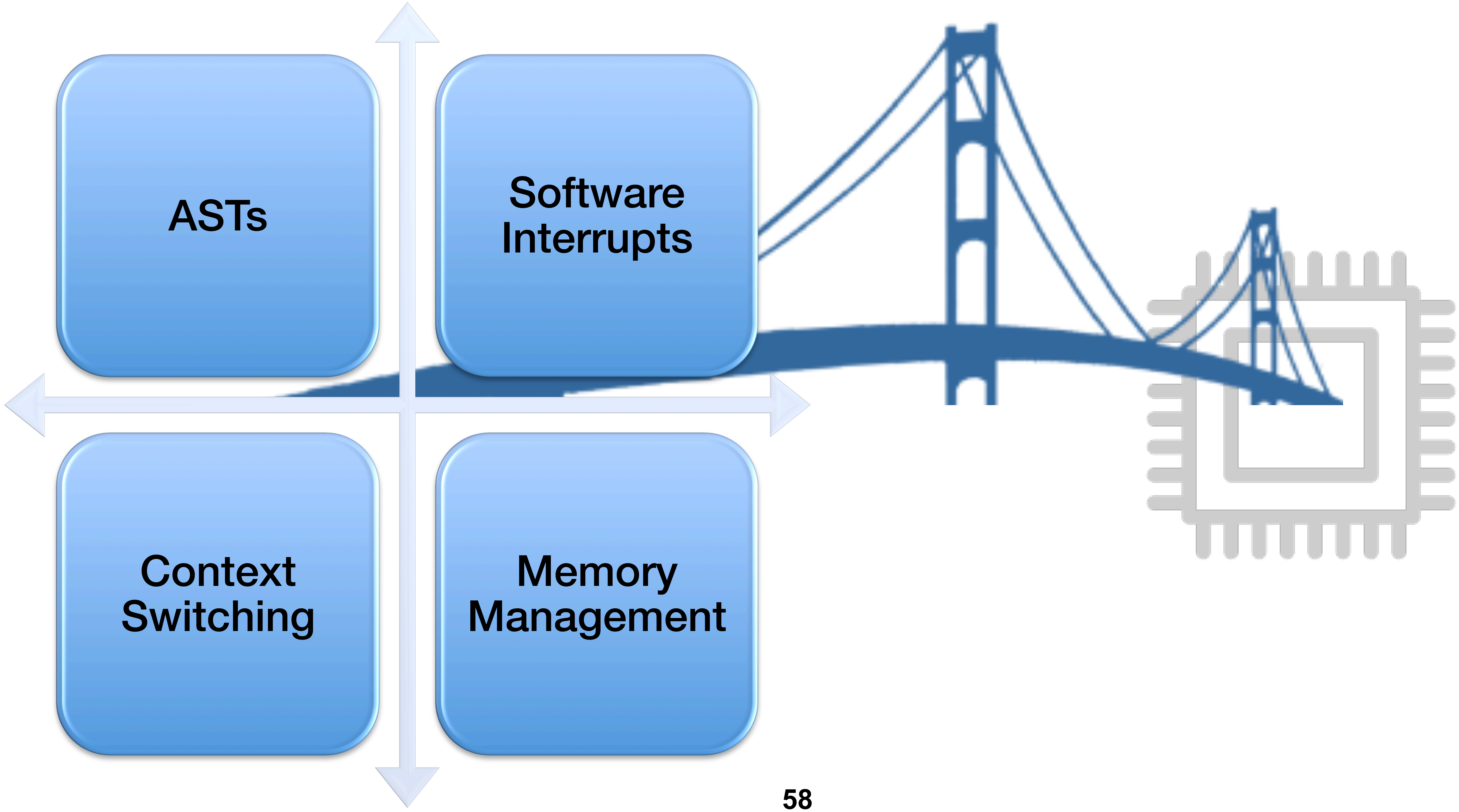


Hence, SWIS

- SWIS (Software Interrupt Services) is a piece of low-level OS code that is involved in mode changes.
- SWIS implements the software interrupt and AST support required by OpenVMS, using hardware support as available.
- Other code in the OS (with some special support from the SWIS code to ensure atomicity) provides atomic queue instructions
- A combination of code in SWIS and other code in the OS provides OpenVMS-compatible IPRs
- SWIS makes the Itanium CPU look more like a VAX to the rest of the OS

Bridge Function

SWIS bridges the gap between the assumptions made by the rest of the OS to the features supported by the hardware



SWIS on X86-64

- Because a similar mismatch exists between OpenVMS' assumptions and the hardware-provided features, SWIS will be ported to X86-64.
- Ported means mostly re-written here, as the provided features are very different between Itanium and X86-64.
- On X86-64, SWIS will have to do more, as the X86-64 architecture does not provide the 4 mode support OpenVMS needs.
- Because of this, SWIS on X86 will not only be active when transitioning from an inner mode to an outer mode, but also when transitioning from an outer mode to an inner mode.
- Also because of this, SWIS now needs to become involved in memory management (in a supporting role).
- There's good news too: the Itanium architecture has some features that are very complex to manage (think RSE), that are absent in X86-64.

Swis on X86-64

OpenVMS Expects:

- 4 Modes, different page protections, separate stacks
- 32 IPLs (16 h/w, 16 s/w)
- Software interrupts tied to IPLs
- Per-process, per-mode ASTs, delivered when below ASTDEL
- Atomic queue instructions
- VAX-like IPRs

X86-64 Offers:

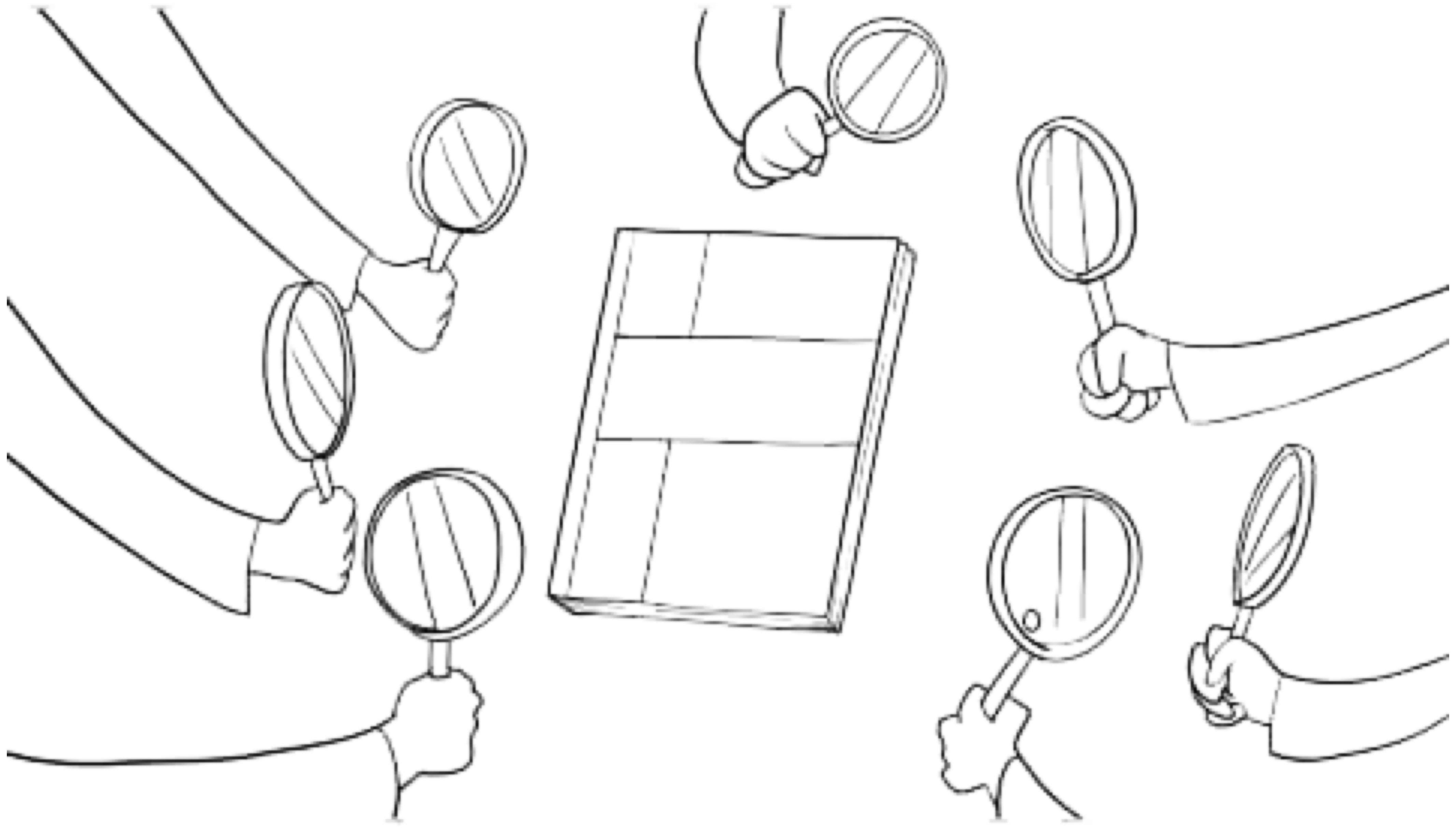
- 2 rings, different page protections, separate stacks
- 14 hardware TPR’s, mask off hardware interrupts in groups of 16
- Software interrupts unaffected by TPR’s. No IPL’s
- No AST-like concept at all
- No atomic queue instructions
- X86-64 IPRs

Design Phase

SWIS for X86-64 was designed over a period of 1.5 years (1 year part-time, 0.5 years full-time), in several phases:

- Basic design (not detailed enough to base implementation on)
- Detailed design for System Service dispatching
- Detailed design for Hardware Interrupt and Exception handling
- Detailed design for Software Interrupts and ASTs
- Detailed design for Processes and Kernel Threads

Design Review Phase



- Partial reviews as the design progressed
- In-depth 3-day review between myself and Burns Fisher
- This one turned up a design flaw that could have enabled unprivileged code to bring down the system
- Complete walk-through and review in one of our weekly X86-64 engineering meetings
- A lot of the content in this presentation is based on the slides I prepared for that walk-through

Implementation Phase

Implementation started in May 2017, broken down into different parts:

- Quick and Dirty Exception Handling for early code that needs something
- Data Structure Definitions
- VAX/Alpha IPRs
- Hardware Interrupts and Exceptions
- System Services
- Software Interrupts
- ASTs
- Initialization
- Processes and Scheduling

2 SYSTEM_PRIMITIVES execlet builds

- Compatibility build, works on any x86-64 CPU we support
- Performance builds, optimized for CPUs that have support for one or more of the following:

1. Address Space Numbers (PCIDs) in TLB
2. RDGSBASE instruction
3. XSAVES/XRSTORS instructions for saving/restoring extended (“floating point”) registers (MMX, SSE, AVX)

- Highest Performance build targets Intel processors made after 2013 (Ivy Bridge and beyond).

SWIS Data Structure

- One per CPU, stays with CPU over the lifetime of the system
- Only CPU-specific datastructure that can be found directly
- Has a different virtual address for each CPU
- Pointed to by GS segment register

Mode “Components”

- Processor ring (0 for K, 3 for ESU)
- Stack pointer
- Address Space Number
- Page Table Base
- Current mode as recorded in the SWIS data structure

- A mode is “canonical” when all the above are in agreement
- SWIS should be the only code that ever sees non-canonical modes

- We prototyped this on Itanium

Basics of Mode Switching

- Interrupt or SYSCALL instruction
1. Switches CS and SS to ring 0
2. Switches to the kernel-mode stack (interrupt only, not SYSCALL)
3. Disables interrupts
- Get fully into kernel mode (ASN, PTBR, stack, DS, ES)
- Going in? -> Build return frame on stack
- Going out? -> Deliver SwInts and ASTs as needed
- Get into destination mode (ASN, PTBR, stack, DS, ES)
- IRET or SYSRET instruction
1. Switches CS and SS to ring 3
2. Switches to the outer-mode stack (IRET only, not SYSRET)
3. Enables interrupts

XDELTA-lite (XLDELTA) Debugger

- **Wanted something, however primitive, as early as possible**
  - Started from scratch, written in C and a little assembler
  - Follows XDELTA syntax
  - Linked into SYSBOOT
- **Current Capabilities**
  - Set and proceed from breakpoints
  - Examine and deposit register
  - Examine and deposit memory location
  - Examine multiple instructions
  - Examine instruction and set breakpoint
  - List breakpoints
- **XDELTA vs. XLDELTA?**

Status: In use, may add additional capabilities

Objects & Images

Image Building and Execution

Calling Standard

- Started with AMD-64 runtime conventions
- Deviated only where absolutely necessary
- Problem #1
  - Standard assumes all within-the-image addressing can be done PC-relative
  - OpenVMS Image Activator may change relative distances between image sections
  - Solution: Attach a linkage table to each code segment and address all data through it
- Problem #2
  - Need to preserve 32b addressability when procedures are in P2 or S2
  - Solution: Create 32b-addressable stubs that forward calls to the procedures
- **Status**
  - Satisfies all current development needs
  - Remaining work: address unwinding, debugger, and translated code issues as they arise

Alpha-to-x86 Dynamic Binary Translator

- Directly execute an Alpha image on x86
- No restrictions in terms of compiler version or operating system version of source
- Does not support privileged code translation
- **Status: working prototype on x86 linux**
  - Using selected pieces of simh as a basis for emulation
  - Running simple Alpha images on x86 linux
  - Temporary code to emulate
    - OpenVMS loader and image activator
    - some OpenVMS library routines
  - BASIC, C, COBOL, FORTRAN, and PASCAL images have been translated
  - With no optimization work, performance is about equal to an Alpha ES47

Dynamic Binary Translator Flow

- First execution
  - Runs in emulation mode
  - Creates topology file
  - Quite slow
- Each subsequent execution
  - Reads topology file
  - Generates LLVM IR
  - Runs natively
  - Updates topology file, if needed

Dhrystone: microseconds/run

• Native	0.2
• Emulated	14.1
• Translated	0.2

Next Steps

- Synchronize topology updates (multiple users)
- Security of topology file
- Image activator integration
- Improve performance
- Translate a VESTed image – looks to be difficult

vms

Cross Build

- **Build on Itanium, target x86**
  - Builds done roughly weekly
  - Let the build tell us what we do not already know
  - Building everything
  - At some point will ignore components not needed for First Boot
- **Tools in place**
  - BLISS, C, XMACRO, assembler
  - Linker, Librarian, Analyze, SDL
- **Status**
  - Concentrating on INIT through ASSEM phases
  - Reducing “noise” with each iteration

What's Different ?

FAQ: What are the visible differences that will come with x86-64 OpenVMS?

- Applications: none that we know of now
- Interactive users and command procedures: none that we know of now
- System managers: new utility to update the MemoryDisk

Thank You

To learn more please contact us:

vmssoftware.com
info@vmssoftware.com
+1.978.451.0110