OpenVMS Technical Journal

V6 - June 2005

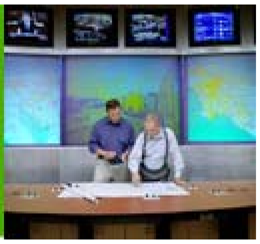


Table of Contents

Porting OpenVMS to HP Integrity Servers

Porting OpenVMS to HP Integrity Servers

Clair Grant, OpenVMS Base Operating System Technical Leader

Distinguished Technologist, Hewlett-Packard Company

Overview

This article describes the 3.5 years of work that went into porting OpenVMS to Integrity Servers. A little history, the rationale behind the major decisions, and some technical details are combined to present the engineering that resulted in OpenVMS I64 V8.2. First there is a chronology of the major events and decisions of the project. The concluding sections provide details of some of the technology that we found the most interesting and challenging and how they came together to produce the preliminary and final releases.

The Early Days

Decision

During the first half of 2001, a group of senior COMPAQ engineers completed an evaluation of the Alpha processor. Alpha had held a performance advantage over Intel processors for several years. However, the difference was getting smaller as time progressed. The evaluation team concluded that even if all planned Alpha projects completed on schedule the two processor families would be performance equals in approximately 2005 and Alpha would almost certainly fall behind in the future.

The economics of future Alpha development were simply no longer feasible. The work on the EV7 that was in progress would continue, but that would be the end of Alpha processor development and the associated hardware platforms would be the end of the Alpha line. All EV8 processor development was terminated.

Announcement

On Monday June 25, 2001, senior COMPAQ management publicly announced that Alpha development would be terminated and that OpenVMS would be ported to the Intel Itanium architecture. The message to customers was clear: in the future there will be faster systems at a lower price than if the current Alpha plans were continued. They also said when it would happen:

“We will deliver a production quality release of OpenVMS in 2004.”

This was a shocking announcement, not only for the public but also for OpenVMS Development!

The promise to application providers was that they would “Recompile, Relink, and Go.” A popular variation of this became, “You are a qual away” from running on HP Integrity. This was an extremely bold message, since we had not even begun to research the details of what needed to be done. However, it became an extremely powerful measuring stick for implementation decisions made throughout the project.

Another aspect of this huge undertaking was that the existing Alpha OpenVMS roadmap would remain unchanged. The importance of this should not be underestimated. It was critical to our customers. Plus, a large number of engineering resources (engineers, build team, release team, business management, qualification and verification, documentation) were already committed to the next OpenVMS release and that could not change.

NOTE: Within just a few days of the announcement I received many mail messages saying, in one form or another, “You people are crazy! VMS will never run on a processor with only two privilege modes!” I attempted to reassure them that “Itanium is indeed a four-mode processor, just like VAX, and VMS will be just fine.”

Research

June 25, 2001 was not only the day of 'the announcement," it was also the day we started working on the project. First, it was communicated to our customers and application providers what we were about to do and what it would mean to them. We also met with engineering and field organizations that would be affected by this new direction. Finally, we formed the initial team of engineers to do the research that would be the input for the plans and schedules.

Read the Manuals

The Intel websites have a great deal of relevant information: white papers, current hardware offerings, future plans, etc. We started reading the Intel IA-64 Architecture Software Developer's Manual, a 4-volume set that would be, and still is, our primary source of information for the architecture (available from http://www.csee.umbc.edu/help/architecture/). The Advanced Configuration and Power Interface (ACPI) specification from Intel was also critical. But we knew this would not be sufficient; we needed to talk with those who architected and implemented the system.

Question: When is a word not a word?

Answer: When you put the Intel Itanium terminology and Alpha terminology side by side.

A byte is 8 bits in both architectures, but that is where the commonality in data sizes ends. The VAX/Alpha "word" is 16 bits long, whereas the Intel "word" is 32 bits. (See Appendix A - How to Make Your Head Hurt for the complete comparison.) Once we got beyond the "communication gap," we wondered how we would document such a fundamental difference. Finally we decided that we would continue with our long-standing tradition of documenting only OpenVMS, and ignore the terminology difference completely. It was the easy way out, but every alternative we considered led to an ugly documentation strategy that did not make anything clearer.

Talk with Engineers

Some members of the COMPAQ compiler group had Intel experience as a result of previous projects. The GEM backend used by the OpenVMS compilers was already "Itanium-savvy." We sought the advice of the compiler developers as to how we should get started. They would soon play a critical role, because compiling the code would be our first step in the porting process.

Just as we were starting to gain a little "knowledge momentum" and realizing where some major decisions might be needed, the next "announcement" occurred: COMPAQ and HP were proposing to merge. This was announced just days before the annual COMPAQ Enterprise Technology Symposium, so the entire tenor of that event changed over night. Little did we know at the time that it would take eight months for the merger would be completed. It was not until May 7, 2002 that OpenVMS become part of Hewlett-Packard.

Historical Footnote: The first public engineering presentation of OpenVMS on the Intel Itanium Processor Family was at the annual COMPAQ Enterprise Technology Symposium in Anaheim, CA. The presentation was scheduled for Tuesday morning – September 11, 2001. However, the world stopped that morning and watched in horror at the tragic events at New York City's World Trade Center, western Pennsylvania, and the Pentagon in Washington D.C. The entire COMPAQ conference resorted to a make-shift schedule for the rest of the week.

Part of the "First Announcement" was that Tru64 UNIX would also be ported. Some of the Tru64 engineers had Intel experience from other projects. Since we were all in this together, we jointly invited a small group of Intel architects to Nashua in mid-September of 2001 for a 3-day seminar covering the instruction set, operating system interfaces, the boot path, and ACPI. It was a real eye-opener! The manuals had brought us a long way with the basics but the face-to-face conversations made it clear that we still had a long way to go. Fortunately, we now had engineers to talk to whenever we had questions.

HP had codeveloped the Itanium architecture with Intel so all of HP's systems were moving in that direction and we were already porting OpenVMS to the desired platform. Immediately, on Tuesday

May 7th, we were called by HP firmware engineers offering to help us in any way, and we also started conversations with the HP-UX engineers. Another May 7th event was the joining of the COMPAQ and HP networks and e-mail. Access to documents and engineers was just a few key strokes away. It was gratifying, as well as tremendously satisfying to realize that in a matter of hours we had access to large development organizations who were eager to share their knowledge with us. After all, they had already overcome many of the hurdles we were about to encounter. Their knowledge would prove invaluable.

The book ia-64 linux kernel by David Mosberger and Stephane Eranian was extremely helpful. In the spring of 2002, OpenVMS engineers attended a seminar in Marlboro, MA, presented by Mr. Mosberger. An excellent reference for anyone wanting to learn about the Intel architecture, the book is available from http://www.lia64.org/book/.

Big Decisions

Very early in the project, we had to make some big decisions that had significant technical implications for porting the code. In some cases, these decisions would be visible to our customers. The following sections describe many of those decisions, but always there was the overriding theme that we stated at the beginning: this was not going to be a "bug-for-bug" compatible port of the existing Alpha code. We would make decisions, even drastic ones, if they would make a better OpenVMS in the future. As you will see, we pushed the limits a few times, but the quality of the product and the needs of our customers was our paramount concern.

1. Implement the Intel Calling Standard (and More)

This was a completely unanticipated outcome. After lengthy evaluation we stunned everyone, including ourselves, when we decided to implement a variant of the Intel Calling Standard. In addition, we decided to use the more universally known Executable and Linking Format (ELF) and Debugging with Attributed Record Formats (DWARF) standards rather than our unique OpenVMS standards.

The calling standard decision was based on a combination of considerations. For the benefits of performance and shared software technology, we tried to use the Intel conventions as much as possible, straying from them only where necessary in order to preserve user-visible characteristics that are essential for multilanguage compatibility and interoperability, features that came from the original VAX design. We knew that creating such a hybrid design was a monumental decision, but over the next couple of years we came to realize that even "monumental" was a gross understatement!

The decision to use the Intel calling standard required us to create the Calling Standard Committee, which consisted of compiler, run-time, and OS experts. Their job was to systematically evaluate every detail of the OpenVMS Calling Standard for VAX and Alpha systems and work out what should be preserved, what had to change, and how best to support the "Recompile, Relink and Go" message. This decision had to be communicated early to Intel compiler developers because the OpenVMS C++ and FORTRAN compilers would be coming from Intel.

2. ELF/DWARF File Format

Selecting the ELF/DWARF file format caused enormous shock waves. The compilers, the linker, the librarian, the debugger, the dump analyzer, and the image activator (just to name a few) would have completely different formats of input and/or output. But we wanted to make OpenVMS more receptive to the world of porting applications and analysis tools. The system would be more accessible to applications because it would have more well-known internal formats and more developers would understand them.

One of the goals of the project was to make OpenVMS more portable. Who knows, maybe someone will be doing this again! The calling standard, ELF, and DWARF decisions doubled the length of the port of OpenVMS, not just in the coding time but also in the conceptualization and the debugging of completely new implementations. It was hard work, but OpenVMS is a better system for it.

3. No Compiler for Alpha Assembler Code

Half of OpenVMS is written in MACRO-32, so we needed a compiler. On Alpha we have the AMACRO compiler, so we created the IMACRO compiler on Integrity. As for Alpha assembler code, we concluded that there isn't much of that around, especially in the application world, and there probably would not be any more written. We could make it easy to rewrite most Alpha assembler by providing the right compiler built-ins for instructions and the right library routines for internal calling standard knowledge, like stack unwinding. These seemed to be the two reasons most implementers wrote Alpha assembler code.

The Integrity C compiler would not be supporting the asm construct. We validated our theory by rewriting all of our existing asm code in C; that is, we never replaced a C asm that contained Alpha assembler with Intel assembler code. This was a very good sign and, although it was not a silver bullet for all applications, it confirmed our "no compiler" decision.

4. Binary Translator for Alpha Images

On Announcement Day we had said that there would be no binary translator, as there is for porting from VAX to Alpha. The binary translator is a very large and difficult piece of code. This decision had implications for numerous components of the system. Components of the base operating system, some RTLs, and even the Alpha firmware have specific code just for the binary translator. Not creating a translator would have saved us a lot of work, but it became clear that it was necessary for the customer offering. We contracted this work to Software Resources International (SRI) , since they specialize in this type of application and already worked with us on other projects.

5. Ada – Yes; PL/I —No

OpenVMS is written almost entirely in MACRO-32, BLISS, and C, and we knew that a little Intel assembler would be necessary. We could go a long way in the port with nothing else. However, we do have a little Ada cade, and it is popular in a few OpenVMS customer segments. We decided to contract with Ada Core Technologies to provide an OpenVMS I64 Ada compiler.

But we decided against a PL/I compiler because there is very little PL/I in the OpenVMS application world and PL/I-to-C translators are available. (In the end, we did rewrite the PL/I part of MONITOR in C! The result is common code for Alpha and Integrity.)

6. Default Floating Point Format is IEEE

The Floating Point format was a gigantic decision that came with far more ramifications than we originally anticipated. On Alpha, VAX floating point formats (commonly known as F/D/G) and IEEE are all implemented in the hardware. In other words, there is no performance advantage to convert to IEEE. For OpenVMS on Integrity, the VAX floating point formats are emulated in software —only IEEE is in the hardware. Depending on the application, the performance difference can be significant.

We decided to make IEEE the OpenVMS default on Integrity. If you compile without a format qualifier on Alpha, you get VAX floating points. On Integrity, the same compile command will produce IEEE floating points. Although this affects some calculations in the highest degrees of precision, we could not justify the performance hit from defaulting to emulation. For those applications where the slower software emulation is an acceptable solution, the OpenVMS Integrity compilers support F/D/G by using the /FLOAT switch.

7. Math Library

The math runtime library is one of the most visible performance-critical components of the system. On Alpha, it is written in C specifically for the Alpha architecture. Recompiling it on Integrity was not acceptable. A viable alternative appeared to be the Intel Itanium math library, which is optimized for the Integrity architecture. We decided to get the math library from Intel. Part of the announcement included sharing compiler technology with Intel, and this was one of the first cases where we reaped the benefits of it. The drawbacks were minor, and vastly overshadowed by the performance improvements, and we were better prepared for future changes.

8. Expanding the Kernel Process Model

On Alpha, a number of OpenVMS system components, such as RMS, the XQP, and DECnet, managed multiple threads of execution on their own; that is, they each knew the details of a context swap and implemented the pieces their individual environments needed in order to suspend and resume execution contexts. Due to different "context" details, these implementations would have to be rewritten for Integrity. We knew some customers had also made similar design decisions. These changes are difficult on any system, but they are particularly challenging on Integrity. Considering the number of components we would need to modify, this would be a daunting task. Likewise, it would be a huge task for customers porting to Integrity. Therefore, we decided to enhance an internal mechanism called "kernel processes" to be callable from any mode and modify all of the OpenVMS components with private threading to call the enhanced routines. Writing the extremely complicated code once and then converting the callers was a much better approach and it would increase the maintainability of the code immensely. An additional benefit would be common code for Alpha and Integrity. We have a paper with examples (google EXE$KP_START) about converting to the new routines (highly recommended) rather than porting their Alpha code.

9. Licensing

We were concerned about being a new HP offering. Licensing OpenVMS was totally different from any of the company's existing products. This would not be necessary if we were willing to make a rather large change. For Integrity (but not Alpha) we decided to adopt the licensing model of HP-UX, in which you purchase a level of operating environment (OE). This would require significant work on our part but it was the right strategy to pursue.

Technology Overview

The vast majority of code in a large and complex operating system like OpenVMS has no dependency on the platform or CPU architectures. For example, device drivers, network protocols, clusters, lock manager, and the file system required no recoding to run on Integrity. However, we had some very big architectural challenges and there was a great deal of work to do in a few concentrated areas of the system.

The next few subsections describe the components of the base operating system that made up the bulk of our work. The components fall into one of the following categories:

1. Those that are analogous to Alpha but different in detail.
2. Those that are unique to Integrity.

No Alpha Console

On Alpha, the console is part of the firmware package. It plays an integral role in booting the system and in device discovery for OpenVMS. The Intel Itanium architecture boot path is by necessity a generic and minimal environment, since its goal is to accommodate the basic needs of a number of operating systems without being specific to any. Therefore, the boot path would be completely new code up to the point of starting SYSBOOT. On Integrity, OpenVMS itself provides many of the runtime services that are provided by the firmware on Alpha.

No Alpha Privileged Architecture Library (PAL)

The PAL is another part of the Alpha firmware package. While the Alpha architecture is operating system agnostic, it allows an operating system to provide its own specialized PAL. The OpenVMS PAL contains code for the VAX queue instructions, the VAX special registers, VAX IPL and privilege mode change management, interrupt handling, knowledge of PTE format for translation buffer misses, and alignment faults. Thus, code written specifically to use these features on a VAX can run unchanged on an Alpha. Again, OpenVMS itself would need to provide these functions on Integrity. Replacing the Alpha PAL was especially challenging.

Different Processor Primitives

The Itanium register set is very different from Alpha, and vastly bigger. The Register Stack Engine is a mechanism to help the operating system manage this large stack environment. It required totally new code in OpenVMS.

The load-locked/store-conditional instruction pairs on Alpha guarantee atomic access to data in a multiprocessor environment. Like VAX, Itanium has individual instructions that guarantee atomicity.

The new register set changes the way a process's context is saved. Therefore, any code that did its own "threading" would need serious revision.

The Alpha PAL does a lot of interrupt processing before notifying the operating system of the interrupt. One of the biggest parts of the porting project was the Software Interrupt Services (SWIS), Integrity-specific code that completely controls interrupt processing and its associated mode checking and management.

Calling Standard Impact

The Intel Calling Standard's register conventions and stack usage are very different from those of the Alpha Calling Standard. To avoid modifying many source code modules due to the register differences, especially modules written in MACRO-32 and BLISS, we defined a mapping of Alpha's R0-R31 to Itanium's R0-R31. The compilers automatically make the conversion, based on a switch. However, programmers still need to know the register mappings when debugging. This mapping found its way to scraps of paper taped to our workstations, from there to a PowerPoint slide, and finally onto a mouse pad, which is still an invaluable reference. (See Appendix B - OpenVMS Alpha to Itanium Register Mapping for the entire register set.)

Integrity exception processing uses "PC mapping" rather than the "frame pointer" scheme we are familiar with on VAX and Alpha. Furthermore, there are two stacks to be coordinated: an integer register stack managed by the hardware as well as the memory stack managed by the generated code. On VAX and Alpha, programmers "walk the stack" in order to find information. The Integrity memory stack has very little information of this type. To find a caller's saved registers, programmers use the current PC to index into the "unwind tables," which point to the saved registers. The unwind data created by the compilers is loaded into the unwind tables during image activation. Use of the unwind tables is the foundation of exception handling.

NOTE: We made the LIBRTL calling standard routines usable from all privilege modes so there would be a single OpenVMS source for this knowledge.

It seemed to take forever to get the base operating system and the LIBRTL unwind interfaces correct. For months we received a steady stream of exception handling problem reports. Even after the code functioned correctly, it was reworked a couple of times to improve its performance. As we expected, this was a very difficult part of the project.

ELF/DWARF Impact

The decision to use the ELF/DWARF internal format resulted in a lot of new code and also made our progress difficult to plan because the areas most impacted were our development tools. Creative debugging ideas were at a premium, to say the least.

The Plan

During the latter months of 2001 we assembled the pieces of the development plan and plotted a preliminary schedule. This was necessary in order to understand the early dependencies. It was not a long-range plan, but it was just enough to get everyone moving and to determine whether we had a grasp of reality.

Common Code Base for Alpha and Integrity

The first implementation decision was to have a common code base for Alpha and Integrity. (VAX and Alpha have different code bases. We were not going to make that mistake again!) In practice, we added Integrity support to the existing Alpha sources. There were three kinds of work:

- Creating a few Integrity-specific modules
- Conditionalizing code in existing modules
- Rewriting some existing routines so that they could be executed on both platforms

Having a common code base was going to make this a very different project from that of porting from VAX to Alpha, as you will see.

External Help

Remember that we promised that "the OpenVMS Alpha schedule would remain unchanged." We needed to do the porting work while the Alpha development work continued. Some of the new Integrity work could only be done by a specific set of OpenVMS engineers. But a much, much larger set of work would be closer to our "Recompile, Relink, and Go" direction.

EDS has supported many of the OpenVMS utilities for years and the engineers are familiar with a lot of our code, build procedures, and integration testing. Therefore, we contracted EDS to port many of the OpenVMS utilities and layered products, including some they did not already support. The EDS engineers who ported and tested a large body of OpenVMS code were critical to our ability to meet our schedules.

A number of contractors also played a vital role in meeting our early schedule demands. With their prior OpenVMS internals experience, they contributed mightily right from their first day on Integrity-specific code.

Project Dependencies and Schedule

Simply put, the direction was to write, compile, link, and execute. From the earliest stages of planning, we knew that we would have BLISS and C compilers first, and then the Intel assembler (IAS) and IMACRO a few months later; but the LINKER was far in the future. A lot of code was written and compiled before we were able to link and debug any of it. This had many ramifications in the months to follow. The LINKER would undergo as much change as any single component in OpenVMS, and much of what we did for the next year, even after we released V8.0, was dictated by the evolving capabilities of the LINKER.

The System Code Debugger (SCD) was also a distant promise. SCD is akin to DEBUG except that it is used for kernel and driver debugging. Unfortunately we would have only XDELTA and print statements for debugging for a very long time. We did not know how much this would affect our schedule. What we did know was that, without SCD, everything would take longer.

The most difficult challenge was to make a sufficiently functioning OpenVMS available to the myriad layered product engineering groups, such as DECthreads, JAVA, DECnet, TCP/IP, and DECwindows, so that they could get going with their porting work.

VAX-to-Alpha as Reference

Many engineers working on the project also worked on the VAX-to-Alpha port a dozen years ago. We tried to translate as much of that experience as we could into useful insights for the current work. There were certainly some similarities but there were also major differences. Comparisons to the VAX-to-Alpha port are inevitable. The striking difference at the technical level is what I call "broad and shallow vs. narrow and deep."

The VAX-to-Alpha port consisted of two major challenges:

1. OpenVMS had never been ported before. How would we compile all of that MACRO-32 code and make it work in a reasonable timeframe?

2. OpenVMS was written specifically for the 32-bit VAX architecture. What would it take to make it run on a 64-bit architecture? The Alpha architecture was the least of our worries, since each Alpha system has OpenVMS-specific PALcode, which makes the system look very VAX-like. In fact, the entire project was called EVAX, or "extended VAX."

This was the "broad and shallow" project; tons of code needed to be changed, but for the vast majority it wasn't rocket science.

The Alpha-to-Integrity port had one gigantic challenge. Integrity doesn't look anything like a VAX and some concepts and code at the very core of the operating systems would be completely new. Meanwhile, the vast majority of the system would require little work, if any. This was the "narrow and deep" project — it would take far fewer engineers but the time required to get from "no bits to ship" was about the same as the VAX-to-Alpha port.

Evaluation Releases

We decided to distribute one internal baselevel to a few select groups and then release two external evaluation releases before the official Field Test. The internal baselevel would be provided to the compiler groups so that they could verify that their generated code was working correctly.

The first evaluation release (E8.0) was to be a "cross tools" environment, where developers could compile and link on Alpha and run the resulting image on Integrity. It would consist of a limited set of compilers, and a large part of the operating system itself would be in place. There were restrictions, but E8.0 would give our work a little exposure and would allow a few adventurous 3rd party developers to kick the tires.

The second evaluation release (E8.1) would be released 6 months later and would provide a "native" environment where code could compile, link, and run on the Integrity system. We would add more partners as the system became more complete. We might even add another hardware platform at this stage.

These two evaluation releases were for Integrity only. It was not until the official Field Test that both Alpha and Integrity customers would see the new version of OpenVMS. (It would be the largest Field Test in OpenVMS history, with 48 Alpha installations and 54 Integrity installations.)

Development Begins

It is impossible to pinpoint a precise day on which development began. During the next few months bits and pieces of work were happening somewhat independently as the design team began to understand what they needed. At the same time, procedures were established, defining the development environment. By the end of 2001, momentum grew as a small group of engineers became connected and started moving in unison. The first release stream code change was made on January 15, 2002.

We were flattered by the confidence shown in us by The Inquirer when its headline of 1/2/2002 proclaimed:

Compaq boots VMS on Itanic

Cool. Even though this was still 13 days before our first code checkin, we enjoyed the publicity anyway!

Compilers

In the beginning, we were limited to the cross tools environment. The compilers ran on Alpha and generated code for Integrity. It would remain like that for a long time —a running OpenVMS on Integrity was at least a year away. Once again, some creative thinking proved valuable.

We would get to a running system faster if the build compilers generated reasonable code right away. The C compiler engineers built prototype Integrity Linux compilers to test the generated code and ELF/DWARF generation on Integrity Linux systems. Much of the Linux work in the compilers had already been developed for the Alpha Linux compilers that we were selling. Starting work on the Integrity Linux compilers in September 2001, the compiler engineers used their existing compilers to tested about 95% of the code generation and much of the ELF/DWARF generation that was needed for the first system boot.

An important step in early compiler development was determining which built-ins OpenVMS would need. The compiler teams looked at some existing Intel compilers and started with those built-ins, adding a few more as the operating system needs became clearer. We needed far fewer built-ins for MACRO-32 and BLISS, because new code that was specific to Integrity would be written in C.

OpenVMS has its fair share of C ASMs on Alpha. They allow:

- Referencing R26 as the caller's return address register
- Programming atomicity with LOAD-LOCKED / STORE-CONDITIONAL instructions

Both the Alpha and Integrity C compilers defined built-ins __RETURN_ADDRESS and __CMP_SWAP_QUAD, so we eliminated the ASMs and produced common code.

MACRO-32 code posed special challenges. It was easy in the VAX-to-Alpha port because the two calling standards are more or less the same – we designed them that way. We were now faced with a very different calling standard. One of the most dramatic differences is that the Intel standard defines only four preserved general registers (R4-R7), whereas Alpha defines fourteen (R2-R15). This meant that many assumptions in the MACRO-32 code were not valid for Integrity. IMACRO (the MACRO-32 compiler on Integrity) recognizes this difference and, unless specifically flagged by a new compiler directive, saves/restores registers that may be at risk. We also recoded these types of operations when convenient, especially if they were performance sensitive.

CALLs to routines that return data in registers other than R0 or R1 must also be flagged with the new directive so IMACRO will not overwrite the returned value. We devised a way for the LINKER to provide diagnostic warnings when register assumptions could not be validated. This was painful in the beginning but helped us immensely as time went on. We then defined all the linkages used by OpenVMS in a new file $IA64_LINKAGES, which is inserted into each MACRO-32 source module using ARCH_DEFS.MAR.

Another MACRO-32 problem was the hand coding of a CALL; that is, moving arguments into R16 through R21, putting the number of arguments in R25, and then JSBing to a standard CALL routine — truly evil! All these occurrences had to be changed due to the different calling standard, so we recoded them to CALLs. If the world were perfect everything would be a standard CALL, but that was not exactly on anyone's mind during the first 12 years of OpenVMS development. In a way, we had made it "too easy in VAX-to-Alpha" (a phrase we've uttered numerous times during the Alpha-to-Integrity porting project).

BLISS global registers were an unexpected challenge. A BLISS global register is one defined to have a common use for all modules in an image, such as RMS's GLOBAL REGISTER that contains the FAB. This doesn't fit into any Intel register convention. BLISS code generation was enhanced to recognize such registers with possible "live" values and protects them.

Many years ago there was a monster movie called The Thing. We encountered one of its descendents – Not a Thing (known as NaT). NaT is the 65th bit of a register. When set, it says that the register contents are invalid. It is used as part of speculative access. Unless you use a special instruction, you can only save a register to memory if the NaT bit is 0 (the register is not NaT). In other words, speculative access (no change) is allowed but a real change is not.

This was a new concept for IMACRO and it took awhile to work out the kinks. The operating system struggled too, because correctly preserving the NaT state when switching context turned out to be nontrivial. At last the monster was slain.

Learning Itanium Assembly Language

We had a goal of writing as little assembler code as possible but we knew there would be some. How do we get started without an assembler? It would be a few months before IAS was ported to OpenVMS; this was due to timing and schedules rather than degree of difficulty. Eventually we took the Open Source IAS assembler and added our limited ELF extensions, a relatively small job. But in the meantime we got an Itanium1 ProLiant DL590/64 and installed Linux. Reading the manuals is one thing but we really learned about 3-instruction bundles, predicates, and unwind directives by writing C code, compiling it, and studying the generated code on our Linux system.

In fact, the following were first compiled and debugged with GNU C and the assembler on Linux:

- SWIS
- EXCEPTION
- The Interruption Vector Table (IVT)
- The Translation Lookaside Buffer (TLB) miss handler

Compiling, linking, and debugging this crucial code depended on Linux tools running on Itanium hardware for some time. This allowed basic OpenVMS functions to be ready when needed by IPB and SYSBOOT.

When we originally made the decision to go with ELF/DWARF we didn't realize how much it would help us during early development. The theory of why it made sense for applications also made sense for OpenVMS.

Replacing the Alpha PALcode

There were three primary areas of work:

- Interrupts, IPL, and privilege mode management done with little OpenVMS knowledge
- Direct PAL routine calls
- Memory management

For speed and simplicity, we defined a system service for each PAL call by creating the appropriate header and library files for C, IMACRO, and BLISS. This got us going quickly and also made it easy for later performance improvements.

SWIS is at the heart of what makes OpenVMS work on Integrity. Everything in the system that requires a check for, or manipulation of, IPL and privilege mode is controlled by SWIS. The data in the SWIS log became the mainstay of debugging the most serious problems we encountered.

Emulation of the interlocked VAX queue instructions was trouble from Day 1. There are many of them and they are complicated. OpenVMS doesn't work at all without many of these instructions.

The memory management changes consisted of virtual-to-physical address translation, alignment fault handling, and PROBE emulation. We devised a scheme for maintaining Alpha virtual address translation and address space layout with GH regions using the VHPT, a basic Itanium construct.

Porting Guidelines

We wrote a short document containing coding practices. Remember that we were adding Integrity support to the Alpha source code while V7.3-2 for Alpha was in full development. The code itself was not the only thing that needed work. The development environment required serious enhancement because we would be building for Alpha and Integrity from common sources, using common building procedures, on the same cluster. Porting guidelines were a must for all facets of the project. As this small document matured it was used by other development groups, too.

You might expect "binary thinking" from computer programmers but we realized very early that some of our existing code base and build environment had a very definite "it's either VAX or Alpha" approach when differences needed consideration. This was perfectly understandable given where we were when we ported from VAX to Alpha; however, it was short-sighted and would cause much work later. We were not going to make that mistake again.

One of our first warnings to programmers was, "watch out for those binary conditionals" and make sure they turn into something as general as possible for the future. All conditionals had to be meticulously checked and many of them needed modification. This was as true for DCL command procedures as it was for C, BLISS, etc. This was so important that as each conditional was checked the engineer added the comment "Verified for IA64 port" even if no change was needed. Here are a few examples:

- IF VAX (…..) ELSE (…..) ENDIF
- IF ALPHA (…..) ELSE (…..) ENDIF
- IF VAX (…..) ENDIF followed by IF ALPHA (…..) ENDIF

It is easy to see that all these instructions have problems when a third alternative arrives. The first two examples result in either the VAX or Alpha code being used on Integrity. This produces a compile or link error on Integrity and the programmer has to figure out the necessary adjustment. But the third is particularly troublesome because it produces, without error, no code at all on Integrity — not desirable! We definitely underestimated the amount of work that needed to be done to our existing conditionalized code. In order to maintain readable, common sources we frequently buried the conditionals in a macro.

Boot Sequence Takes Shape

In the fall of 2001 we bought our first HP i2000s. We were still part of COMPAQ but the proposed merger with HP had just recently been announced and we figured the i2000 would give us a head start in our future company.

Very shortly we reached the point of needing regular, coordinated meetings for those designing and writing code in the boot path. The first "Debug Meeting" was held on November 14, 2001, and this meeting continued multiple times a week for almost two years. They were not status meetings; they were for asking dumb questions, showing our ignorance, and especially learning from one another – and they were usually somewhat entertaining. The Debug Meetings were for developers only —no managers allowed, 30-45 minutes and get back to work. They were for down-in-the-trenches bits and bytes discussion of what was and was not working and how to keep moving forward.

In the spring of 2002 we acquired some Intel "white boxes," generic development starters, engineering samples that are loaned to companies creating products based on Intel processors. These white boxes nearly doubled our systems. In addition, we were learning different experimentation techniques to assist us in our design and development.

For this discussion, the boot sequence is defined as the code needed to get from issuing the boot command to having loaded all of the necessary execlets (the set of images that constitute the core of OpenVMS). The code of interest is in:

- VMS_LOADER.EFI
- IPB.EXE
- SYSBOOT.EXE
- EXEC_INIT.EXE

Before describing the code we should review the generic Intel-defined boot path structure:

1. There must be a File Allocation Table (FAT) partition on the boot device. OpenVMS does not have disk "partitions" in the PC sense. The OpenVMS FAT partition is a container file; that is, a regular OpenVMS file with special contents and format on the system disk.

2. One of the files in the partition is the operating system's loader. This file is created by each operating system development group wanting to boot on an Intel Itanium architecture system.

As you can tell from the .EFI extension, the first code to execute is not an OpenVMS image. It is an Extensible Firmware Interface (EFI) application written by OpenVMS Engineering that prepares the system and then loads and starts IPB.EXE. IPB is the Itanium Primary Bootstrap, which is analogous to APB.EXE on Alpha and VMB.EXE on VAX. In addition to the boot sequence logic, the loader contains a primitive file system so that it can find, load, and start an OpenVMS image before the normal mechanisms are available. Thus, even the very first piece of code executed has to know about the inner construction of an ELF image.

The loader is completely new code for Integrity. One way to look at its function is that it gets things in place so that IPB.EXE can function much the same as its counterpart on Alpha. VMS_LOADER.EFI does many of the same things that the firmware does on Alpha, such as creating the configuration tree and the HWRPB. In turn, IPB does as much as possible to make things the same on Integrity as on Alpha for SYSBOOT.EXE. Once SYSBOOT.EXE starts, things look nearly identical to Alpha in terms of any external interfaces (for example, stopping to examine or change system parameters). SYSBOOT.EXE creates a number of data structures, loads a few execlets, and finally transfers control to EXEC_INIT.EXE.

At this point, most of the platform-specific code in the boot sequence is complete. EXEC_INIT.EXE loads the rest of the execlets (typically a couple of dozen), executes their initialization routines, and transfers control to the SWAPPER process.

Another project theme was "don't be different" without a very compelling reason. This challenged us in a number of areas but none more so than ACPI with its relationship to the boot path. One manifestation of "don't be different" was if other HP Integrity operating systems did something strictly by the Intel specifications, then OpenVMS should too. We went out of our way not to require special treatment from the architecture or the firmware designers or implementers. Booting the system is one of those areas where operating systems can vary greatly. Specifically, most operating systems consist of one monolithic file that gets loaded. Knowing that any relevant code is already running, ACPI then provides all the configuration data. This is clearly not the OpenVMS model you have seen in the previous paragraphs.

Primitive Debugging

Before we had the luxury of a debugger or the ability to take a crash dump, various forms of print statements were the only means of tracking down problems. We suffered with print statements for awhile, still making progress. Then one of those silver bullets materialized – the instruction decoder. The callable instruction decoder and instruction format structure definitions are the foundation for all of the debugging tools, like DELTA, XDELTA, DEBUG, SDA, SCD, and PCA. In addition, the format definitions are used by the base operating system during alignment fault handling. When XDELTA became functional, breakpoints and single stepping instructions emerged to increase our productivity. While most of the underlying debugging code was completely new for Integrity, there was one thing that was "old." Setting the PSR.ss bit is just like the Tbit on VAX, and much simpler than the way it's done on Alpha.

Being able to take a crash dump is a momentous event in a project like this and our first one happened quite unintentionally. While stepping through some code, a breakpoint was incorrectly positioned. A ";P" caused an ACCVIO and, lo and behold, a crash dump was executed perfectly! The code had been recently compiled and linked and was just waiting to be exercised. It was one of unusual moments where seeing a crash happen is a joyous occasion.

Early "Builds"

During the first half of 2002 a significant amount of code was written and the compilers began to take shape. A few OpenVMS source code facilities were in the early stages of building for Integrity. It was way too soon to think about doing a full Integrity build, so we concentrated on the few pieces we needed to continue making progress. In the early days we built only those few images and then copied them to an OpenVMS Alpha system disk, using that to test the early stages of booting. This was a little cumbersome but it required only a few engineers and it moved us along quite well. One of the nice things about using industry standard components was that whenever we needed a few new disks we just went over to CompUSA and picked up a few — a very new concept for OpenVMS development.

At this point in the project we did not yet have the LIBRARIAN, which is at the heart of how we build most of OpenVMS. We changed the build process so that it does not create the big object library and delete the .OBJs, and then we changed all of the link procedures to include the .OBJs directly rather extracting them from the library.

Our LINKER capabilities were so limited at the time that OpenVMS was forced to run in a dramatically unusual way. The LINKER could link a single image, but it could not yet link execlets that call from one to another through the base image. Everyone who was around OpenVMS 20 years ago (prior to V5.0) will remember the monolithic SYS.EXE. That was not quite what we needed now but a very, very large SYSBOOT.EXE did fill the bill. During the very early days of debugging the growing boot path, all the code that we needed that normally resides in the execlets was linked into SYSBOOT.EXE. We didn't need this throwback solution for very long, but every single day counted and it kept us moving.

The first general Integrity build was announced on June 3, 2002. It was really the starting point for everyone who was not working on the boot path to see the result of compiling their code. (Loosely translated, this meant discovering just how well MACRO-32 code survived the new IMACRO compiler.)

Now part of HP, we received our first internal shipment of rx2600s in July of 2002. The project was starting to look almost normal — regular builds, regular meetings, a growing number of development systems, and some code that worked. With the i2000s, the white boxes, and now the rx2600s, every developer doing significant work had a system for exclusive use. As the number of implementers grew, so did our need for more and better procedures.

Since we had no networking capability on Integrity yet, we needed a way to quickly and easily move files from our development cluster to the Integrity systems. We took an HP disk system 2100 (a "pizza box" or "pancake") and connected it to a small OpenVMS Alpha system and six Integrity systems, replicating this setup numerous times. This is a completely unsupported configuration, of course, but with careful mounting and dismounting it works. We would build on the development cluster and copy the resulting kit to the "server" Alphas. Individual developers did system disk builds on their designated disk, dismounted it, and booted their Integrity system.

It was about this time that we were starting to realize how much larger the image size on Integrity was, due to relocation data, unwind data, and the number of instructions. Added to that was the traceback information that we had decided to create, to help with debugging in the field. The result is that OpenVMS images on Integrity (application software as well as the operating system images) are on the order of three times larger than the corresponding image on Alpha.

Boot Contest

The "First Boot" is always a milestone in a project like this. From the technology standpoint, however, it is only a minor proof point that the final goal might be attainable. So much under-the-hood cheating is going on that it borders on fantasy. For the most part, people understand this. Nevertheless, given the building anticipation, it is still extremely exciting.

The first boot of OpenVMS on Integrity became a public contest. Predict exactly when it would happen, and win a prize! (This just added to the pressure when we were struggling to get anything at all to work, let alone do something as identifiable as a system boot.)

Every contest must have rules. Here was the official definition of "first boot:"

1. System must boot "MIN."
2. Must be able to login as SYSTEM.
3. $ DIRECTORY returns the correct list of files.

There was no requirement for security, network, clustering, SMP, batch, editor, or anything else not absolutely necessary.

In estimating the expected date for the project plan, we looked back to the VAX-to-Alpha port, when we booted OpenVMS six months after we had reasonably reliable, linked images. Back then we were working on hardware that was in development. For the Alpha-to-Integrity first boot, given that we were using Integrity systems that were shipping HP products, we reasoned that it might take half as long, so we established the goal of three months, which translated into February 13, 2003. Diligent engineering plans notwithstanding, someone announced to the world that we were trying to boot by the end of 2002!

Not All Fantasy

Many fundamental pieces of the system must be working in order to achieve even a minimalist first boot. Here are some:

- IMACRO-32, BLISS, and C compilers
- LINKER
- Device discovery
- Memory layout
- Device driver
- Platform support
- Interrupts
- Clock ticks
- Memory management
- Lowering/raising IPL
- AST delivery
- MOUNT
- File system
- RMS
- DCL
- Image activator

Within these parameters, cheating was allowed, even encouraged. Exception handling, error returns, security, accounting – who cares, as long as the DIRECTORY listing is correct.

The official time of the first boot of OpenVMS on Integrity was January 31, 2003 at 3:31 PM EST on an HP i2000 (Itanium1) system. This prompted hugs, handshakes, and congratulatory words. It was a huge boost to our confidence and a gigantic sense of relief! The verified Boot Contest directory output and the winners of the contest are in Appendix D – Boot Contest is Official: January 31, 2003 at 3:31 PM EST.

NOTE: After listing the files we didn't return to the $ prompt. The rules of the contest specifically left that out (in case anyone is getting picky). Why? The last event in processing the DIRECTORY command is to return "file not found." Since the system was not even close to handling signals or errors, it stopped at a breakpoint after printing out the last file specification!

The complete list of files used in the first boot is in Appendix E - Boot Contest Code.

Internal Baselevel – March 28, 2003

OpenVMS booted on an HP rx2600 (Itanium2) system on March 17, 2003; this was the platform that most of our early users would have. Less than two weeks later on March 28, 2003 we released our first internal baselevel to the compiler groups. We were definitely rolling. Having a system we were confident enough to give to other groups expecting to do their own development and testing was a gigantic step. Soon we had DECnet and TCP/IP, making it easier to share development systems.

We first booted an Integrity system into a cluster with Alpha and VAX systems on May 15, 2003. The first public clustered demo was May 19, 2003 at the HP-Interex Conference in Amsterdam. See Appendix F - The Cluster T-Shirt for the $SHOW CLUSTER command output from our development lab cluster. (The cluster contained a VAX but it was edited out of the picture due to early uncertainty concerning our support statement for VAX and Integrity in the same cluster.)

There was still a long way to go, but booting networked, clustered systems confirmed our feeling that a completely functioning OpenVMS on Integrity was possible. When we started, many people were extremely skeptical that we would ever match the quality and performance of OpenVMS on Alpha. Others said it would be impossible to make it work at all. It should be noted that, 12 years earlier, there was the same skepticism (05%) about porting from VAX to Alpha. The 1990 nay-sayers had more credibility, since it was the very first attempt at porting OpenVMS and yes, it was indeed questionable as to whether it could be done. This time we never had any question about being able to do the port. The only question was how well the system would perform.

OpenVMS V8.0 – June 30, 2003

The internal code name for V8.0 was MAKO. The development environment used cross tools. We compiled and linked on Alpha and moved the images to an Integrity system. The goal of this first evaluation release was to give some of our 3rd party providers the opportunity to get an early start. We hoped that they would be ready when our first general release shipped. Also, it would give us some limited exposure in an environment other than our own. Things were starting to get serious.

The cross tools were:

- Compilers – BLISS32, BLISS64, C, IMACRO, FORTRAN
- IAS assembler
- LINKER
- CRFSHR
- LIBRARIAN
- CHECKSUM
- CDU
- ANALYZE_OBJ

OpenVMS Engineering created a special program called Fast Track and provided dedicated support for 20 selected partners, albeit with a very limited OpenVMS environment, while we learned just how things were starting to shape up.

A Technical Exchange in Nashua on July 8-9, 2003 with engineers from HP firmware and operating system groups, in particular one of the HP team members who co-developed the Itanium architecture with Intel, was enlightening. Although we had a functioning system, it was still one of those "the more you know, the more you realize you don't know" experiences.

Now more OpenVMS engineers began to get involved with the compiling and linking of various components of the system. We held a series of lunch-time seminars open to the entire engineering community. The engineers who had been doing the initial development work presented details of the new Integrity-specific areas of the system.

What's in a Name?

In 1998 at a very public forum in Los Angeles, OpenVMS Business Management said, "There will never be a V8.0 of OpenVMS." It would be V7.*x* forever, implying that we would be causing as little perturbation as possible in the system for the rest of its lifetime. At the time, there was a different mindset regarding the future of OpenVMS.

Times had changed. In 2003 there was widespread debate within OpenVMS as to which OpenVMS Integrity release would be dubbed V8.0. "V8" became the local catch-phrase, and large quantities of a certain juice maker's product mysteriously appeared in conspicuous places.

OpenVMS V8.1 – December 18, 2003

The internal code name for V8.1 was JAWS and it came from a native development environment. We compiled and linked on the Integrity system. The goal of this second evaluation release was twofold:

- Increase the number of partners.
- Establish a complete Integrity environment.

Compiling and linking was now supported on the Integrity systems – no more cross tools and transferring files. In fact, one of the early tests of this environment was building the tools themselves natively (for example, a compiler compiling itself).

Whole Program Floating Point Mode

Floating Point...floating point...floating point! Floating point works very differently on Alpha and Itanium. Source code modules that have been compiled with different floating point qualifiers (control, rounding, and precision), as well as the objects, get linked into a single image. On Alpha, floating point preferences are part of the instruction stream generated by the compilers. On Integrity, they are defined by the contents of the Floating Point Status Register (FPSR) at the time of instruction execution.

On Alpha, you get exactly what you declared for each module with no decisions or assumptions needed along the way. On Integrity, the floating point preferences are noted in each object file and the LINKER stores in the image the floating point preferences of the module with the MAIN entry point. After an image is loaded into memory but before any instructions are executed, the FPSR is set with the values established by the LINKER. As you can see, there is no way for OpenVMS to magically reproduce the exact Alpha behavior for this mixed-mode case. For an application that is sensitive to the individual module declarations, there are system services, namely SYS$IEEE_SET_*, for dynamic changes as the program executes.

Converting from VAX F/D/G to IEEE

We had to add more library routines to help applications convert from VAX floating point types to IEEE. Many routines already existed but now we were actively encouraging application writers to convert and we discovered that there were missing functions. Who would do this unplanned work at this very late date? We enlisted some engineers at the Customer Support Center and they ably produced the desired results.

Locking the Working Set in Memory

Privileged processes, which enter kernel mode and raise IPL, lock their code and data in the working set so that the pagefault handler doesn't get invoked at too high an IPL for I/O. The system service SYS$LKWSET accepts a range of addresses and locks the appropriate pages in memory. For compatibility, we changed SYS$LKWSET to lock the entire image in memory when an address in the input range is found to be in the code image section.

Only the code and data need to be locked on VAX, but Alpha locks code, data, and linkage data – a little messier but doable. Because of the new ELF image format on Integrity, programmers have to lock code, data, short data, and linkage data, which is nearly impossible to figure out at the application level. We highly recommend that applications use LIB$LOCK_IMAGE to lock the entire image in memory; it is easier to use than multiple calls to SYS$LKWSET. This new $LIB routine is available on both Alpha and Integrity, so you can maintain common code. The entire image is locked, so the process may require larger working set quotas. In OpenVMS, we switched many of our privileged processes to use LIB$LOCK_IMAGE and increased the SYSTEM account's working set quota.

OpenVMS V8.2

Superdome Demo

There is nothing like a little publicity. The Superdome 4-OS (HP-UX, Linux, OpenVMS, Windows) demo at the HP Analysts Meeting in Westboro, MA. on February 13, 2004 drew lots of attention. The 4-cell system was configured as 4 nPars (hard partitions), each running a different operating system.

V8.2 does not support cell-based systems, which presented some new problems to overcome. Since this was a demo (some cheating allowed) we hard-coded a few I/O-related addresses in the drivers rather than devise a complete solution. We even had to boot OpenVMS in cell 0, but these quick, temporary measures brought a high payback.

50-bit Physical Addressing

A lightening bolt struck at the wrong time. Everything was in place for the first official Integrity Field Test in mid-summer 2004, but when we started testing with the mx2 CPUs for the rx4640 we had a revelation: the mx2 cache brought with it the need to use addresses beyond the OpenVMS-supported 43-bit maximum. One option was to not support the 8-CPU version of the rx4640 and postpone the necessary memory management work until the next release, but we decided to do the work in V8.2 because additional new platforms might come along before then. The data structure changes and source code changes would affect any privileged code that worked directly with PFNs, for example DECnet and TCP/IP, and we wanted to get all known, mandatory privileged code changes into the first Integrity release.

A design was created, a project plan written, and five engineers were drafted on very short notice to make the OpenVMS changes:

1. Expand the PFN field in the PTE from 32 bits to 40 bits.
2. Expand all other PFN fields in data structures, global cells, and routine interfaces to 64 bits.

These changes were extensive and for Integrity only; however, common source code was maintained using macros in which the platform differences were buried. This work disrupted the weekly builds for awhile and the affected layered products (who had thought they were already done) now had difficult work to do – not a lot, but in sensitive areas. As the base operating system work concluded, the 50-Bit Physical Addressing Programming Cookbook helped developers complete the extra work we had foisted on them at the last minute and everyone survived.

Rewriting the VAX Queue Instruction Emulation

Replacing the Alpha PAL calls with OpenVMS system services worked well in most cases. Early in the project we created the "promote page" as our mechanism for employing Integrity's Execute Privileged Code (EPC) instruction. We used this mechanism for entering kernel mode before transferring to SWIS to switch stacks, save registers, and call the system service dispatcher.

EPC, without SWIS, switches privilege modes without switching stacks, saving registers, or dealing with IPLs, ASTs, etc. This is exactly what we needed for the performance-critical queue instruction emulation. However, without SWIS to set up stacks, there are many restrictions. The code had to be written carefully in assembly language, but we were able, after a few misadventures, to create exactly the code we needed.

Checking access to possibly misaligned queue elements in threaded process environments was difficult. We programmed exception handlers to catch queue elements that the caller's privilege mode cannot access. We also designed a clever use of the TAK, PROBER, and PROBEW instructions with interrupts disabled to quickly determine if the code could proceed to atomically execute the queue instruction. After a few false starts, we finally had a good solution. EPC gave us just what we needed. A few other PAL calls have been changed to use this method.

The Final V8.2 Changes

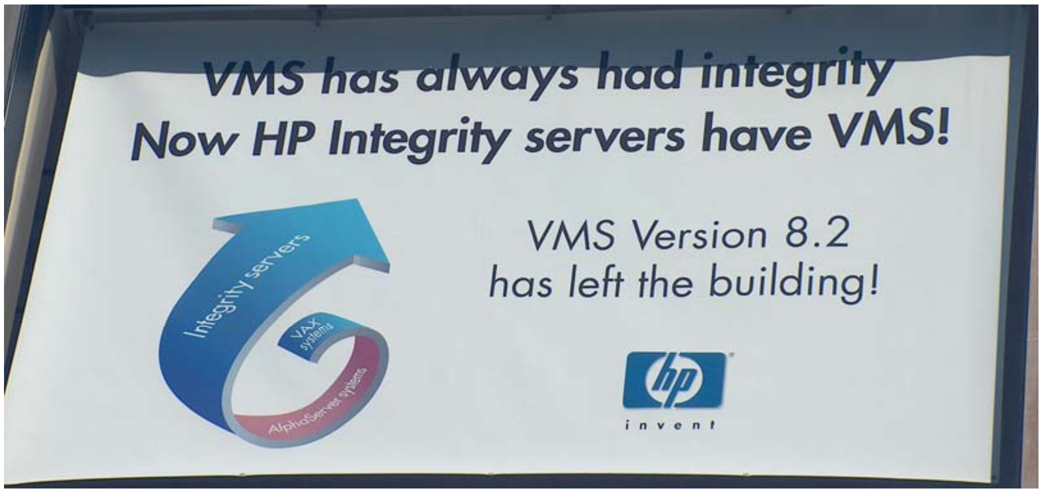
Just before Christmas 2004 the irrepressible NaT consumption fault made another appearance. It was found and fixed and we went home for the holidays thinking that we were done. The test clusters would run over the holiday break and we would claim victory when we returned. Unfortunately, the results were not satisfactory. In fact, during the holidays a few of the engineers noticed a couple of problems.

Floating Point...Floating Point...Floating Point!

There are 128 floating point registers on Integrity systems. It would be desirable to not save/restore all these registers unnecessarily when context is changed. To that end they are architecturally treated as two sets, F0-F31 (low) and F32-F127 (high). Status bits, maintained by the processor for each set, indicate whether a change has been made to a register in the set. In many cases, the high floating point register set is not used and OpenVMS takes advantage of the "modified" bit indicators to eliminate unnecessary saves/restores. There was a case involving AST delivery where we didn't have it quite right and it took us a very long time to find it.

Then we encountered another floating point problem! Floating point register corruption was happening when correctable memory errors were reported. Not only was this problem difficult to find, but it was involved with our internal OpenVMS fiat about floating point usage. OpenVMS and HP-UX have an agreement with the compiler writers that if code is compiled using /INSTRUCTION_SET=NOFLOAT, only registers F6-F11 can be used in the generated code. The operating system is compiled this way in order to limit the saves and restores at interrupt level. The register corruption happened at the same time as the memory errors. Firmware test engineering provided a test program that inserts correctable memory errors. We quickly ported it to OpenVMS, increased the error polling rate, then diagnosed and fixed the problem. The OpenVMS error logger was calling firmware, and not appropriately preserving the floating point registers.

The last code change in the V8.2 release was made on January 10, 2005 and the final test runs were started. They ran with no problems. We had finally made it! The OpenVMS Engineering Readiness Review was held on January 13, 2005 and it was official – the final DVD was sent to manufacturing.



Performance Evaluation

Now that we had a production-quality system, how well did it perform? That is a very open-ended question so we decided to concentrate on how comparable Alpha and Integrity hardware platforms performed with a selected set of CPU, memory, and I/O tests. For example, we compared an Alpha DS25 with an Integrity rx2600 and an Alpha ES45 with an Integrity rx4640.

In general, these Alpha and Integrity systems perform about the same. In some tests, one is slightly better than the other, and different tests produce results that are reversed, but in most cases the differences are small. When "the announcement" occurred way back in 2001 it was stated that the performance leadership crossover from Alpha leadership to Integrity would be roughly 2005. That turned out to be a very accurate prediction.

Integrity systems have a wealth of performance counters accessible from software. OpenVMS tools have incorporated the ability to gather the data and relate appropriate information to instruction pointers in routines. This helped in tracking down and improving some initially poorly performing areas, for example needless stack unwinding (very expensive) during certain calling sequences of an RTL. Another example was an inefficient OTS data-moving routine.

One universal finding was that everyone had to eliminate unaligned data. An unaligned data reference on Alpha is expensive, but it can be two orders of magnitude worse on Integrity. So be sure to naturally align those data cells!

As we gain more experience on these new platforms there will certainly be on-going operating system and compiler performance improvements, as well as CPU and platform speed-ups. This will be true for the rx1600, rx2600, and rx4640 (supported by V8.2), and even more so for the larger systems supported by future releases of OpenVMS.

Summary

Three and a half years of work resulted in OpenVMS I64 on Integrity servers. We tried to make moving from Alpha to Integrity as easy as possible for the application providers. We know there will be an occasional exception but for the vast majority of applications we achieved our goal. We may have caused minor inconveniences when it was not strictly necessary for Integrity, but part of the goal was to make OpenVMS a better system in 2005 than it was in 2001, and that's what we did.

A project of this magnitude happens only a few times during the life of a product. This is the second time for OpenVMS and it is still moving forward.

Acknowledgements

Porting OpenVMS encompassed the work of more people and groups than can be adequately recognized here. The entire OpenVMS community participated in this enormous undertaking.

Everyone was dedicated to the highest standards in producing a quality product worthy of our customers.

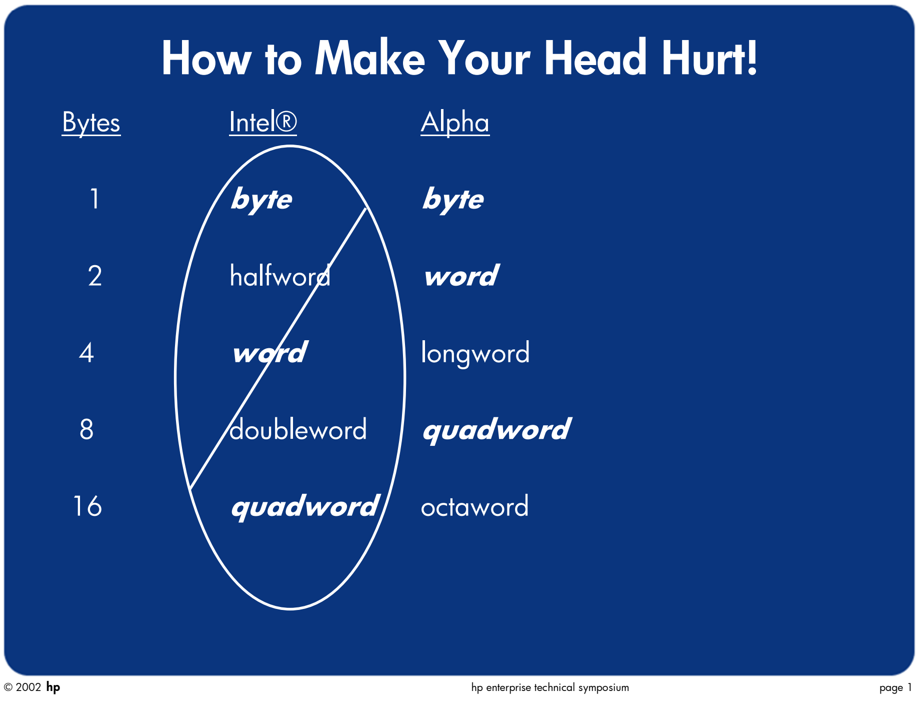
This port to Integrity was the result of the efforts of many very talented people, but the work done by those who contributed to OpenVMS in previous years, even previous decades, also contributed greatly. OpenVMS is a great system to work with in terms of concept and implementation, and we are indebted to all of those who came before us.

A special "thank you" goes to the reviewers of this article – Burns Fisher, Karen Noel, John Reagan, Ron Brender, Jeff Nelson, Josh Cope, Gaitan D'Antoni, Sue Lewis, Ken Moreau, and Randy Barth.

For more information

Send questions and comments about this article to clair.grant@hp.com.

Appendix A - How to Make Your Head Hurt

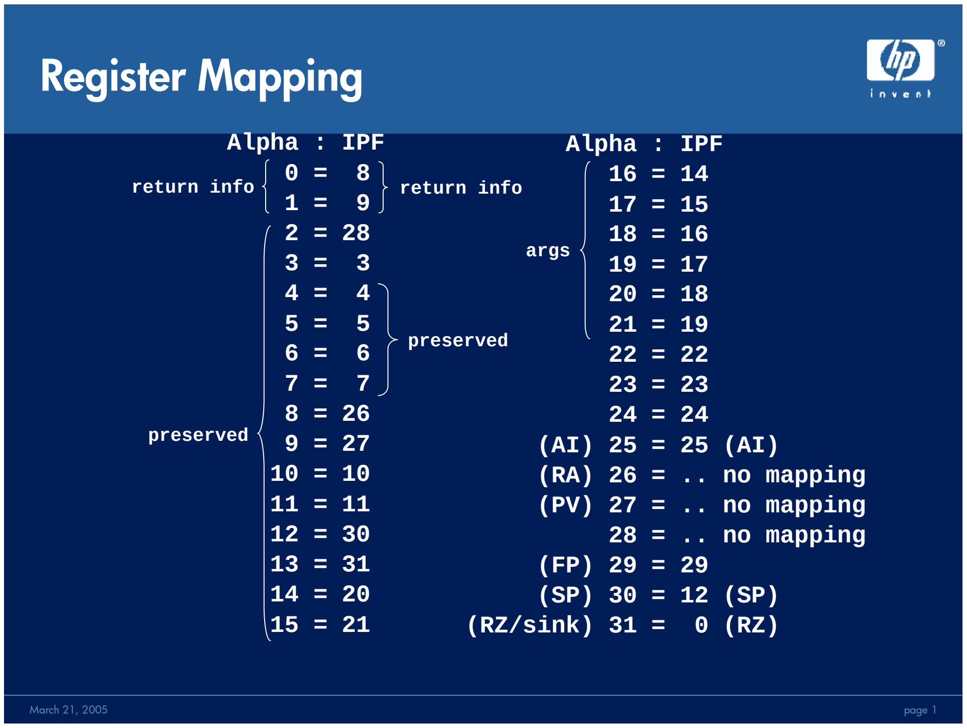


How to Make Your Head Hurt!

Bytes	Intel®	Alpha
1	***byte***	***byte***
2	halfword	***word***
4	***word***	longword
8	doubleword	***quadword***
16	***quadword***	octaword

© 2002 **hp** hp enterprise technical symposium page 1

Appendix B - OpenVMS Alpha to Itanium Register Mapping



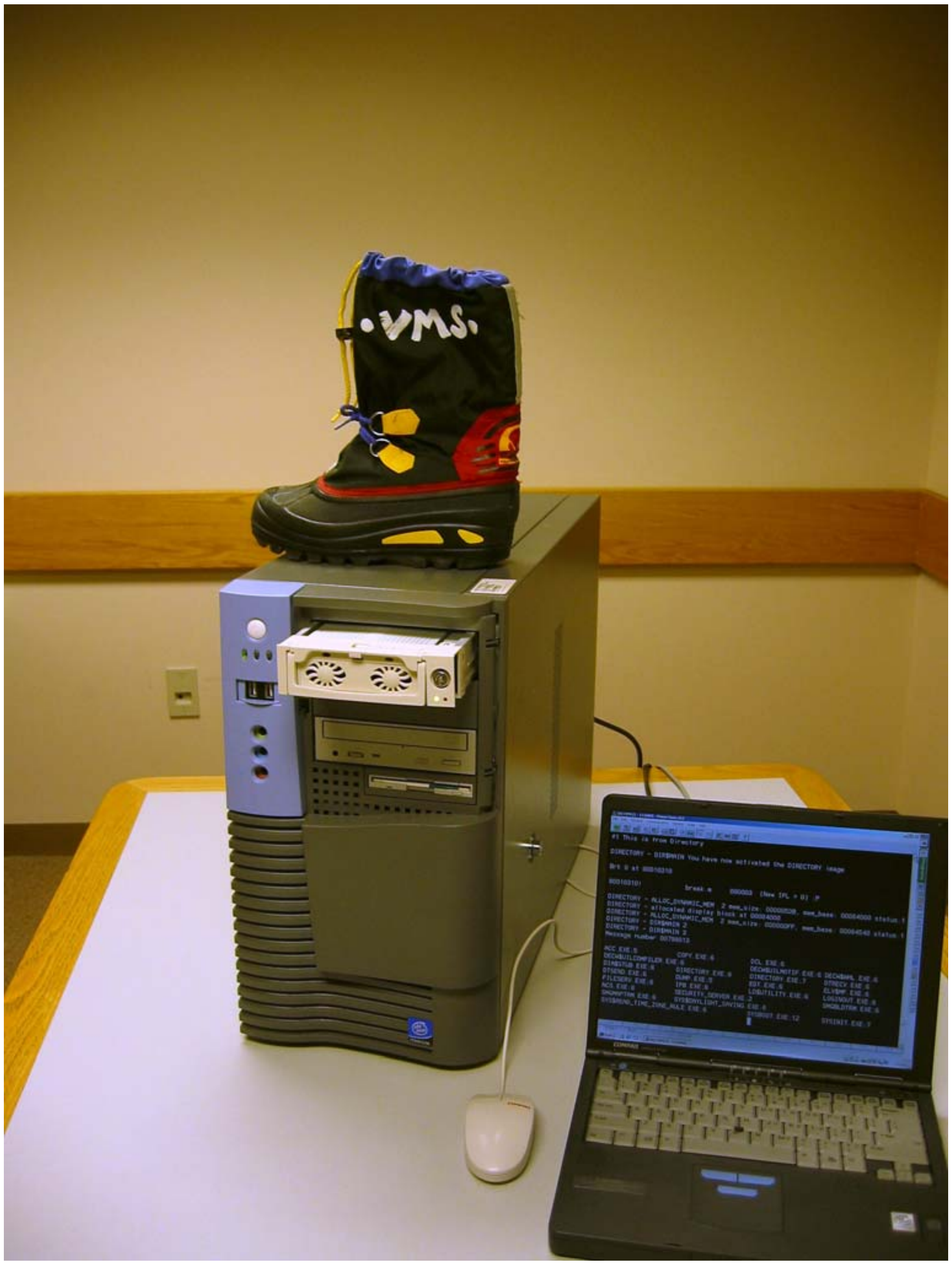
Register Mapping

	Alpha	:	IPF	
return info	0	=	8	return info
return info	1	=	9	return info
preserved	2	=	28	
preserved	3	=	3	
preserved	4	=	4	preserved
preserved	5	=	5	preserved
preserved	6	=	6	preserved
preserved	7	=	7	preserved
preserved	8	=	26	
preserved	9	=	27	
preserved	10	=	10	
preserved	11	=	11	
preserved	12	=	30	
preserved	13	=	31	
preserved	14	=	20	
preserved	15	=	21	

	Alpha	:	IPF	
args	16	=	14	
args	17	=	15	
args	18	=	16	
args	19	=	17	
args	20	=	18	
args	21	=	19	
	22	=	22	
	23	=	23	
	24	=	24	
(AI)	25	=	25	(AI)
(RA)	26	=	..	no mapping
(PV)	27	=	..	no mapping
	28	=	..	no mapping
(FP)	29	=	29	
(SP)	30	=	12	(SP)
(RZ/sink)	31	=	0	(RZ)

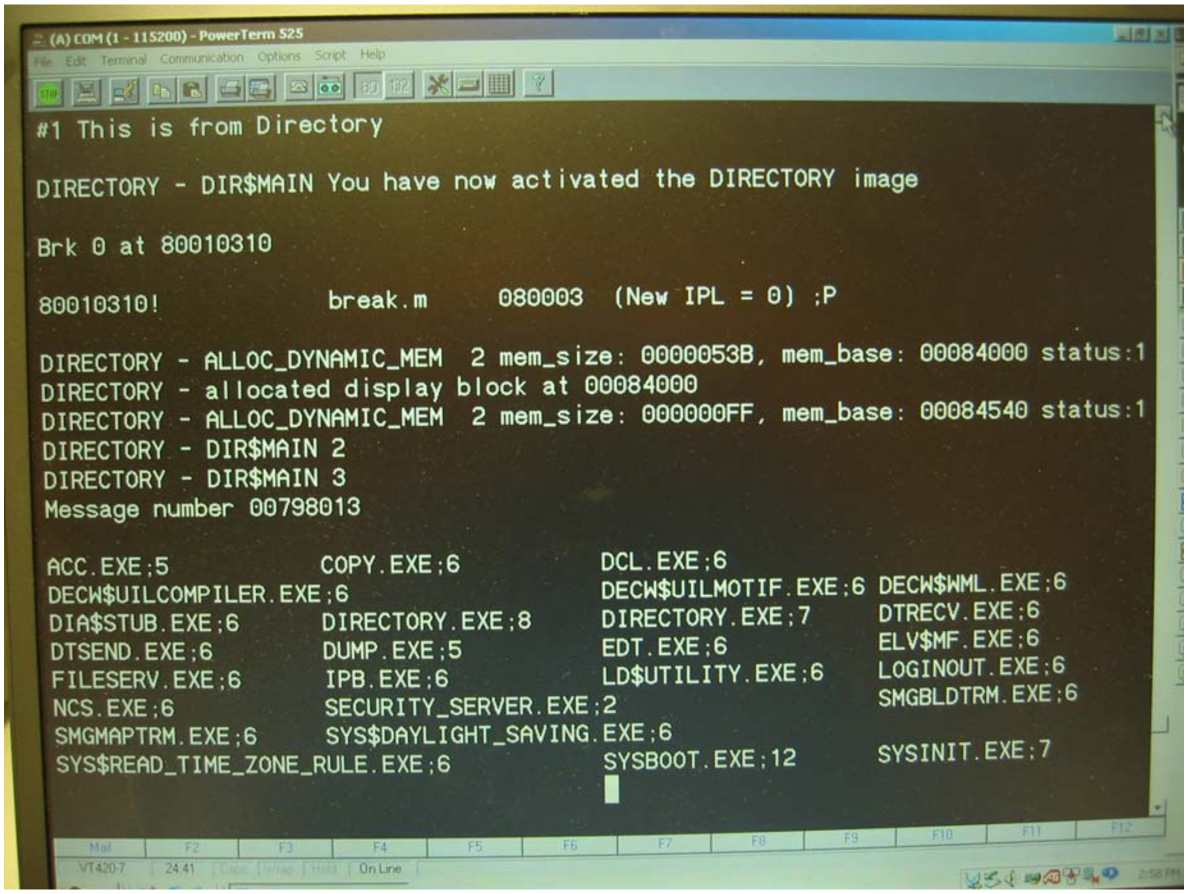
March 21, 2005 page 1

Appendix C - First Boot on HP i2000



```
#1 This is from Directory

DIRECTORY - DIR$MAIN You have now activated the DIRECTORY image

Brk 0 at [illegible]

[illegible]
                    break.m     [illegible]  (New IPL = 0) .P
DIRECTORY - ALLOC_DYNAMIC_MEM  2 mem_size: [illegible], mem_base: [illegible] status: 1
DIRECTORY - allocated display block at [illegible]
DIRECTORY - ALLOC_DYNAMIC_MEM  2 mem_size: [illegible], mem_base: [illegible] status: 1
DIRECTORY - DIR$MAIN 2
DIRECTORY - DIR$MAIN 3
Message number [illegible]

ACC.EXE;5
DECW$UILCOMPILER.EXE;6   COPY.EXE;6
DIR$STUB.EXE;6                          DCL.EXE;6
DTSEND.EXE;6         DIRECTORY.EXE;6    DECW$UILMOTIF.EXE;6 DECW$AWL.EXE;6
FILESERV.EXE;6       DUMP.EXE;5         DIRECTORY.EXE;7     DTRECV.EXE;6
NCS.EXE;6            IPB.EXE;6          EDT.EXE;6           ELV$MF.EXE;6
SMGMAPTRN.EXE;6      SECURITY_SERVER.EXE;2 LOGUTILITY.EXE;6 LOGINOUT.EXE;6
SYS$SEND_TIME_ZONE_RULE.EXE;6 SYS$DAYLIGHT_SAVING.EXE;6     SMGBLDTRM.EXE;6
                                        SYSBOOT.EXE;12      SYSINIT.EXE;7
```

Appendix D – Boot Contest is Official: January 31, 2003 at 3:31 PM EST


(A) COM (1 - 115200) - PowerTerm 525
File Edit Terminal Communication Options Script Help
VT420-7
24.41
On Line

```
#1 This is from Directory

DIRECTORY - DIR$MAIN You have now activated the DIRECTORY image

Brk 0 at 80010310

80010310!          break.m     080003  (New IPL = 0) ;P

DIRECTORY - ALLOC_DYNAMIC_MEM  2 mem_size: 0000053B, mem_base: 00084000 status:1
DIRECTORY - allocated display block at 00084000
DIRECTORY - ALLOC_DYNAMIC_MEM  2 mem_size: 000000FF, mem_base: 00084540 status:1
DIRECTORY - DIR$MAIN 2
DIRECTORY - DIR$MAIN 3
Message number 00798013

ACC.EXE;5             COPY.EXE;6              DCL.EXE;6
DECW$UILCOMPILER.EXE;6                        DECW$UILMOTIF.EXE;6 DECW$WML.EXE;6
DIA$STUB.EXE;6        DIRECTORY.EXE;8         DIRECTORY.EXE;7     DTRECV.EXE;6
DTSEND.EXE;6          DUMP.EXE;5              EDT.EXE;6           ELV$MF.EXE;6
FILESERV.EXE;6        IPB.EXE;6               LD$UTILITY.EXE;6    LOGINOUT.EXE;6
NCS.EXE;6             SECURITY_SERVER.EXE;2                       SMGBLDTRM.EXE;6
SMGMAPTRM.EXE;6       SYS$DAYLIGHT_SAVING.EXE;6
SYS$READ_TIME_ZONE_RULE.EXE;6                 SYSBOOT.EXE;12      SYSINIT.EXE;7
```

Winners of the Boot Contest shirts were:

- Luciana Silva
- Gabriel Sterk
- Randy Loch
- Geoff Bryant
- Kurt Huddleston
- James Wilkinson
- Lee Mah
- Alan Frisbie
- Hans Vlems

Appendix E - Boot Contest Code

These files contain the 1361 modules that completed the first boot of OpenVMS on Integrity.

vms_loader.efi
ipb.exe
sysboot.exe
sys$public_vectors.exe
sys$base_image.exe
sys$cpu_routines_4001.exe
sys$srbtdriver.exe
sys$opdriver.exe

exception.exe
io_routines.exe
sysdevice.exe
process_management.exe
sys$vm.exe
shell8k.exe

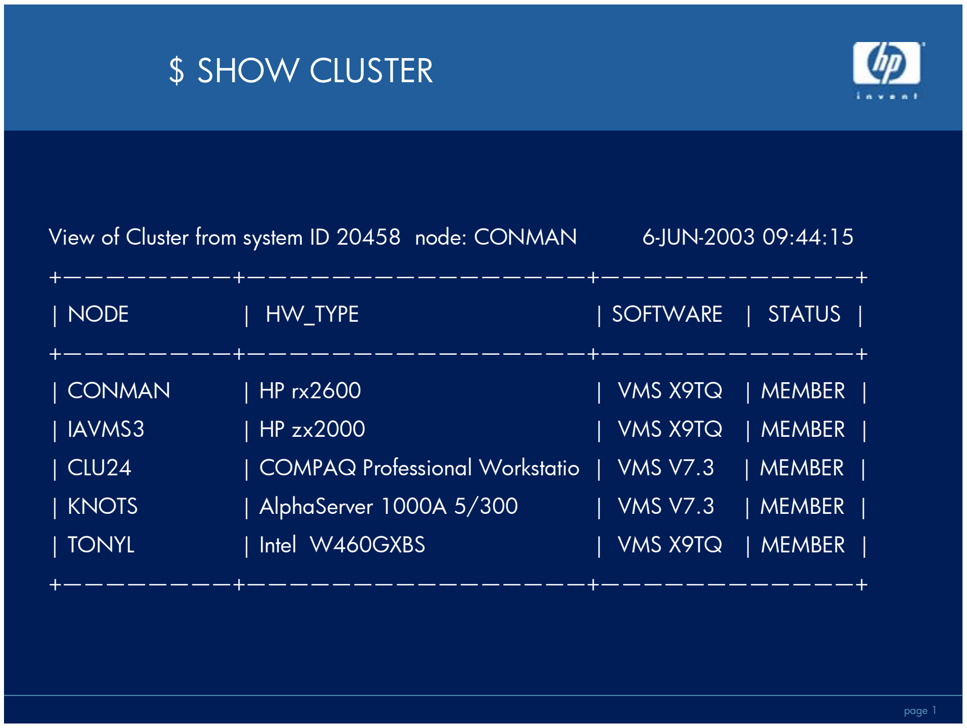
sysinit.exe
libots.exe
librtl.exe
decc$shr.exe
cma$tis_shr.exe
image_management.exe
locking.exe
sysldr_dyn.exe
security.exe
vms_extension.exe
f11bxqp.exe
rms.exe
loginout.exe

sys$dqbtdriver.exe
system_primitives_min.exe
system_synchronization_uni.exe
errorlog.exe
exec_init.exe
system_debug.exe
ia64vmssys.par

logical_names.exe
sys$ttdriver.exe
sys$srdriver.exe
sys$dqdriver.exe
vms$system_images.dat
sys$config.dat

dcl.exe
dcltables.exe
directory.exe
util$share.exe
directmsg
sysmsg

Appendix F - The Cluster T-Shirt



$ SHOW CLUSTER

```
View of Cluster from system ID 20458  node: CONMAN         6-JUN-2003 09:44:15
+---------+-----------------+-------------+
| NODE      |  HW_TYPE                      | SOFTWARE   |  STATUS  |
+---------+-----------------+-------------+
| CONMAN    | HP rx2600                     |  VMS X9TQ  | MEMBER  |
| IAVMS3    | HP zx2000                     |  VMS X9TQ  | MEMBER  |
| CLU24     | COMPAQ Professional Workstatio |  VMS V7.3  | MEMBER  |
| KNOTS     | AlphaServer 1000A 5/300       |  VMS V7.3  | MEMBER  |
| TONYL     | Intel  W460GXBS               |  VMS X9TQ  | MEMBER  |
+---------+-----------------+-------------+
```

page 1

Appendix G - Evaluation Release V8.0

Fatal Bugchecks on OpenVMS Alpha and OpenVMS I64 Systems

Fatal Bugchecks on OpenVMS Alpha and OpenVMS I64 Systems

Ruth Goldenberg and Richard Bishop, OpenVMS Engineering

Overview

When OpenVMS code detects an internal inconsistency, such as a corrupted data structure or an unexpected exception, it generates a bugcheck. If the inconsistency is not severe enough to prevent continued system operation, the bugcheck generated is nonfatal and merely results in an error log entry.

If the error is serious enough to jeopardize system operation and data integrity, OpenVMS code generates a fatal bugcheck. This generally results in aborting normal system operation, recording the contents of memory to a dump file for later analysis, and rebooting the system.

This article describes how fatal bugchecks are handled on OpenVMS Alpha and OpenVMS I64, how the dump file is found, how memory is written to the dump file, and how the dump file is organized.

Initial Bugcheck Handling

On Alpha, a bugcheck is generated by placing a bugcheck value in R16 and executing a `CALL_PAL BUGCHK` instruction. On I64 a bugcheck is generated by placing a value in R17 and executing a `BREAK BREAK$C_SYS_BUGCHECK` instruction. This article refers to these simply as the `BUGCHECK` instruction. In both cases, the bugcheck value identifies the type and severity of the bugcheck and determines whether a crash should be followed by a cold or warm reboot. If the bugcheck is fatal, bit 2 of the value is 1; otherwise it is 0. If the bugcheck is fatal and a cold reboot is requested, bit 0 of the value is 1; otherwise, it is 0. Bit 1 is not used and is always 0.

Regardless of the type of bugcheck, executing the instruction causes an exception. The `BUGCHECK` instruction results in changing access mode to kernel mode and passing control to code that services the exception. In this article, this service routine and the routines it calls are referred to as BUGCHECK code.

OpenVMS BUGCHECK code is implemented in several parts. Initially, the exception is dispatched to platform-specific code. On I64 platforms, BUGCHECK code flushes all stacked registers to their memory backing store. On both platforms, BUGCHECK code ensures that interrupt priority level (IPL) is at least 3. This prevents rescheduling and resumption on a different CPU.

BUGCHECK code tests whether this is a recursive bugcheck, that is, whether it has already saved bugcheck context in a nonpaged pool per-CPU structure called the per-CPU database. If so, it refrains from overwriting the description of the original bugcheck. If this is not a recursive bugcheck, BUGCHECK code records information such as the following in the per-CPU database:

- Access mode stack pointers and, on I64 platforms, access mode register stack pointers
- Page table base registers
- Address space numbers
- AST summary and enable information
- Software interrupt summary information
- Process unique value, which identifies a user thread
- Processor status at bugcheck, including IPL and access mode
- Number identifying the specific bugcheck
- Process and system cycle counters

- Address of the stack structure describing the BUGCHECK exception: on Alpha, the exception stack frame, or, on I64, the INTSTK structure built by software interrupt services (SWIS) code

BUGCHECK code decides whether subsequent processing can occur in the current hardware context or whether it needs to switch contexts. If this is an outer mode bugcheck (user or supervisor), no switch is necessary. If this is an inner mode fatal bugcheck, BUGCHECK code switches to its own context, which has a larger kernel mode stack.

On Alpha platforms, BUGCHECK code saves on the current stack the integer registers not already saved in the exception frame. If the process is performing floating-point calculations, it saves all the floating-point registers on the stack. On certain Alpha platforms, it also triggers the on-chip logic analyzer to dump any program counter (PC) data and to stop logging more.

On I64 platforms, the general registers, branch registers, predicate registers, application registers, and minimal floating point registers (F6 through F11) have already been saved in the INTSTK structure. BUGCHECK code saves the rest of the low bank of floating point registers (F0 through F31) and, if the process is using the high bank, saves F32 through F127.

BUGCHECK code's subsequent actions vary, depending on the access mode in which the bugcheck occurred and the severity of the bugcheck. The following sections describe the BUGCHECK code actions.

Bugchecks from User and Supervisor Modes

The OpenVMS operating system itself generates few bugchecks from user or supervisor mode. It provides the mechanism for use by other software. When a bugcheck is generated from user or supervisor mode code running in a process with BUGCHK privilege, BUGCHECK code writes an error log message.

The SYSGEN parameter BUGCHECKFATAL has no effect on bugchecks generated from user or supervisor mode. Only bit 2 of the bugcheck value determines whether a given bugcheck is fatal. Fatal user and supervisor mode bugchecks affect only the current process.

If the bugcheck is fatal, BUGCHECK code returns to the access mode of the bugcheck and requests the Exit ($EXIT) system service. It specifies the value SS$_BUGCHECK as the final image status. What happens as a result of this service request depends on whether the process is executing a single image (without a command language interpreter, CLI, to establish a supervisor mode exit handler) or is an interactive or batch job.

- If the process is executing a single image, a fatal bugcheck from user or supervisor mode typically results in process deletion.

- If the process has a CLI, a fatal bugcheck generated from an interactive or batch job typically causes the currently executing image to exit and control to be passed to the CLI through its supervisor mode exit handler. The CLI prompts for the next command.

In either case, the only difference between fatal user and supervisor mode bugchecks is that user mode exit handlers are not called when a fatal bugcheck is generated from supervisor mode.

If the bugcheck is not fatal, BUGCHECK code restarts Alpha on-chip logic analyzer data collection (if it was active), restores saved registers, and dismisses the exception. Execution continues with the instruction following the `BUGCHECK` instruction.

Bugchecks from Executive and Kernel Modes

Various OpenVMS components generate bugchecks from executive and kernel modes. If an executive or kernel mode bugcheck value is not fatal and the SYSGEN parameters BUGCHECKFATAL and SYSTEM_CHECK are both zero, BUGCHECK code proceeds as it does for nonfatal bugchecks for the outer two access modes. It writes an error log message, restores the saved registers, and dismisses the exception, passing control back to the instruction following the `BUGCHECK` instruction.

Typically, execution continues with no further effects. However, the routine that detected the error and generated the bugcheck can take further action. One example of such a routine is the last chance handler for executive mode exceptions. It generates the nonfatal bugcheck SSRVEXCEPT (unexpected system service exception). On the presumption that process data structures are inconsistent, it then requests the $EXIT system service. Exiting from executive mode results in process deletion. Another example is the Record Management Services (RMS) routine that generates the nonfatal bugcheck RMSBUG. On the presumption that process RMS data structures are inconsistent, it deletes the process by requesting the Delete Process ($DELPRC) system service.

If BUGCHECKFATAL is 1 or SYSTEM_CHECK is nonzero, any executive or kernel mode bugcheck is treated as fatal, independent of bit 2 of the bugcheck value. By default, BUGCHECKFATAL and SYSTEM_CHECK are 0, which means that a nonfatal inner access mode bugcheck does not cause the system to crash. BUGCHECK performs fatal bugcheck processing under any of the following circumstances:

- BUGCHECKFATAL = 1
- SYSTEM_CHECK ≠ 0
- The bugcheck is fatal.

In the case of a fatal bugcheck, the most important function of BUGCHECK code is to record the contents of memory and of the error log buffers. BUGCHECK code does not use standard I/O mechanisms to write this data because they may be affected by the system inconsistency that triggered the fatal bugcheck. Instead, it performs I/O through the bootstrap system device driver, the one used during system initialization. After recording memory contents, BUGCHECK halts the system to prevent any further system operations in case they might lead to data corruption.

After the system reboots, during system initialization, the error log buffers are copied to nonpaged pool for processing by the ERRFMT process. The dump file can be examined with the System Dump Analyzer (SDA) to determine the cause of the crash.

Fatal Bugcheck Processing on a Uniprocessor System

In processing a fatal bugcheck, BUGCHECK code takes the following steps:

1. On an I64 system, BUGCHECK turns off SWIS tracing and disables virtual hash page table walking (a hardware mechanism to improve performance of handling translation buffer misses). On an Alpha system, BUGCHECK turns off performance monitoring. On both platform types, BUGCHECK raises IPL to 31 to disable all interrupts.

2. BUGCHECK tests whether there have been four or more recursive bugchecks. If so, it displays on the console an error message, information about the most recent exception, and the stack. It then

reboots or shuts the system down, depending on the value of SYSGEN parameter BUGREBOOT.

3. If all of the following conditions are met, BUGCHECK displays on the console the location of crash-related information and enters XDELTA:

   - There have been no recursive bugchecks.
   - BUGREBOOT is zero.
   - The bugcheck has not been triggered by shutting the system down or running OPCCRASH.
   - The system has been booted with XDELTA or the remote debugger.

   Entering XDELTA enables a human operator to examine the state of the operating system through XDELTA commands and request a crash dump at will.

4. Based on the value of the BUGREBOOT parameter and bit 0 of the bugcheck value, BUGCHECK selects one of the following halt action values: perform a cold bootstrap, perform a warm bootstrap, or remain halted. It stores the value in the hardware restart parameter block (HWRPB) per-CPU slot.

5. If this system is a Galaxy node, BUGCHECK notifies other sharing set members that this node is going down unless there have been three or more recursive bugchecks. This test allows for the possibility that attempting to notify the other members caused some of the bugchecks.

6. BUGCHECK shuts down any system communication services (SCS) circuits unless there have been three or more recursive bugchecks. This test allows for the possibility that attempting to shut down SCS circuits caused some of the bugchecks.

7. BUGCHECK shuts down all adapters and initializes the adapter that connects the system device circuits unless there have been three or more recursive bugchecks. This test allows for the possibility that attempting to shut down adapters caused some of the bugchecks.

8. BUGCHECK stores the value of the bugcheck code in the per-CPU database.

9. If there have been no recursive bugchecks, BUGCHECK begins to write information about the bugcheck to the console terminal: it announces that the system is crashing. If this is the second nested bugcheck, BUGCHECK writes a message saying that this is a recursive bugcheck and displays information about the most recent exception.

10. BUGCHECK validates the checksum of the boot control block, the data structure containing the locations of the error log and dump files. If the checksum is bad, no dump can be written.

11. If the checksum is good and the system disk is shadowed, BUGCHECK determines the unit number of the master disk in the shadow set. If bit 2 in DUMPSTYLE is set, BUGCHECK also selects the first valid dump device from the DUMP_DEV environment variable, as described later in this article.

12. BUGCHECK writes information about the bugcheck to the console terminal. This information can include the bugcheck message, addresses of loaded executive images, current process name, current image name, privileges of the current security persona, contents of registers, and contents of stacks relevant to the crash.

    How much information is written depends on the value of bit 1 in SYSGEN parameter DUMPSTYLE. The default value of 0 inhibits most output. The console output is written before the dump file and should not be interrupted by halting the processor from the console terminal. Such an interruption prevents the dump file from being written.

13. BUGCHECK writes the dump header, trap information blocks, the bugcheck error log entry, and the error log buffers to SYS$SPECIFIC:[SYSEXE]SYS$ERRLOG.DMP. For simplicity, the layout within SYS$ERRLOG.DMP matches the layout within the dump file. After writing the error log buffers, BUGCHECK rewrites the dump header to indicate that the error log buffers have been written.

    If the system disk is a member of a shadow set, BUGCHECK writes the same set of information to every member of the shadow set. This ensures that regardless of which member is master when the system reboots, system initialization code can process the error log buffers from the system disk before the complete shadow set is mounted and made consistent.

14. BUGCHECK determines whether a dump is to be written and, if so, what kind of dump, based on the following criteria:

    - If the SYSGEN parameter DUMPBUG is 0, no dump is written. Its default value is 1.
    - If neither SYS$SPECIFIC:[SYSEXE]SYSDUMP.DMP nor PAGEFILE.SYS exists on the system disk or a disk specified by the DUMP_DEV environmental parameter, no dump is written.
    - If, during system initialization, resources cannot be allocated for BUGCHECK's use, no dump can be written. BUGCHECK initialization code attempts to allocate system page table entries to map I/O requests, its memory stack, and, on an I64 platform, its register stack. It needs additional page table entries if the dump is to be compressed.
    - If this is an operator-requested shutdown generated through the system shutdown command procedure, no dump is written.
    - If bit 0 in DUMPSTYLE is set, memory is dumped selectively; otherwise, a physical memory dump (also known as a full dump) is written. The default value of this parameter specifies a selective dump.
    - If bit 3 in DUMPSTYLE is set, the memory contents in the dump will be compressed. BUGCHECK needs physical memory to serve as compression buffers. It borrows memory from the resident code granularity section after copying the pages' current contents to a temporary location in the dump file. If insufficient pages are available, an uncompressed dump is written. The default value of this parameter specifies a compressed dump.

15. If no dump is to be written, BUGCHECK continues with step 23.

16. If any type of dump is to be written, BUGCHECK switches, if necessary, to the disk containing the dump file. This could be needed if BUGCHECK had been writing to shadow set members' SYS$ERRLOG.DMP or if it is writing a "dump off system disk" (DOSD). It then writes the dump header, trap information blocks, bugcheck error log entry, and the error log buffers to the dump file. Then it rewrites the dump header with a status indicating that the dump contains the error log allocation buffers.

17. If the system disk is shadowed, BUGCHECK outputs a message to the console terminal indicating which member it is dumping to and whether that member is the master unit.

18. BUGCHECK outputs a message saying that it is starting to dump memory and specifying whether the dump is physical or selective and whether it is compressed.

19. If the dump is compressed, BUGCHECK zeroes the blocks in the dump that will contain the compression map.

20. If the dump is physical, BUGCHECK writes a map of memory to the dump file so that SDA can associate blocks of the file with physical pages. (Memory is likely to be discontiguous and may not start at page 0.) It then writes physical memory to the dump file, starting with the lowest page and, if requested, compressing as it goes. A later section in this article describes in detail the layout of a physical memory dump and the compression algorithm.

21. If the dump is selective, BUGCHECK writes selected virtual address spaces to the dump, compressing as it goes if compression was requested. Each selected virtual address space is represented by a logical memory block (LMB).

    Not all virtual addresses in the range spanned by an LMB are necessarily included in it. Because a virtual page not in memory cannot be written to the dump file, it represents a hole in the virtual address space. An LMB with holes in its address space contains a **hole table**, which lists the pages of address space not present in the dump.

    The general sequence in writing an LMB is as follows:

    a. BUGCHECK writes an LMB header in the next block of the dump.

    b. BUGCHECK scans the page tables that describe the address space to be dumped, looking for invalid pages that are not transition pages. It writes an entry in a hole table for each such sequence of pages found. It writes the hole table to the next block (or blocks) of the dump. It rewrites the LMB header and the dump header to reflect the presence of the hole table in the dump.

    c. BUGCHECK scans the hole table, filling in its allocated system page table entries with information from each valid or transition page table entry that it found. That is, it double-maps those pages so that it can write virtually noncontiguous pages in one I/O request.

    d. When BUGCHECK has written all the valid and transition pages in a particular LMB to the dump file, it rewrites the block containing the LMB header with correct information about the number of holes in the address space and the number of data blocks (valid and transition pages) in the LMB.

    e. It rewrites the dump header to increase the chances that the dump can be analyzed even if the dump is incomplete.

    If BUGCHECK reaches the end of a file sized for selective dumps before it reaches the end of the LMB list, it rewrites the descriptor of the current LMB with the hole count and actual number of data blocks written. It then rewrites the dump header, filling in status information such as the number of I/O errors encountered while writing the dump file, the number of process LMBs written, and so on.

    In writing a selective dump, BUGCHECK must defend against the possibility that whatever error led to the bugcheck might also have corrupted the data structures necessary to write virtual address space. BUGCHECK replaces the page fault and access violation exception service routines with its own routines to prevent recursive bugchecks if either of those errors occurs. It also performs consistency checks on certain key data structures. For example, it checks whether an address presumed to be that of a process header is syntactically correct; that is, the address must be within known address boundaries and at an integral number of process headers from the beginning of the address range.

22. If a dump was written, BUGCHECK rewrites the dump header so that information such as the error count and number of blocks of memory in the dump reflects what was actually written to the dump file.

23. If SYSGEN parameter BUGREBOOT is 0, BUGCHECK writes a message on the console terminal. As a last step, it halts the system. In response, the console subsystem gains control and acts on the halt action value stored in the per-CPU slot of the hardware restart parameter block (HWRPB) during step 4. Typically, BUGCHECK performs a warm bootstrap.

Fatal Bugcheck Processing on a Symmetric Multiprocessing (SMP) System

When one CPU member of an SMP system incurs a fatal bugcheck, all members crash; the executive takes the conservative approach that an inconsistency severe enough for operations on one CPU to cease is likely to be systemwide. All members of the active set participate in fatal bugcheck processing.

The CPU that first incurs a fatal bugcheck (known as the CRASH CPU) drives the crash. It informs the other active CPUs that a bugcheck sequence has been initiated and takes a number of steps to ensure that a consistent system state can be saved. In response, the other active CPUs crash with the fatal bugcheck CPUEXIT. The primary CPU performs most of the remaining fatal bugcheck processing.

The following figure shows the sequence of some of the steps in fatal bugcheck processing as they might occur concurrently on the CRASH CPU (which is not pictured as the primary processor), the primary processor, and the other active set members. Steps shown in different columns but on the same line do not necessarily execute at the same time on all CPUs. The numbers in the figure correspond to the steps described after the figure, not all of which are represented in the figure.

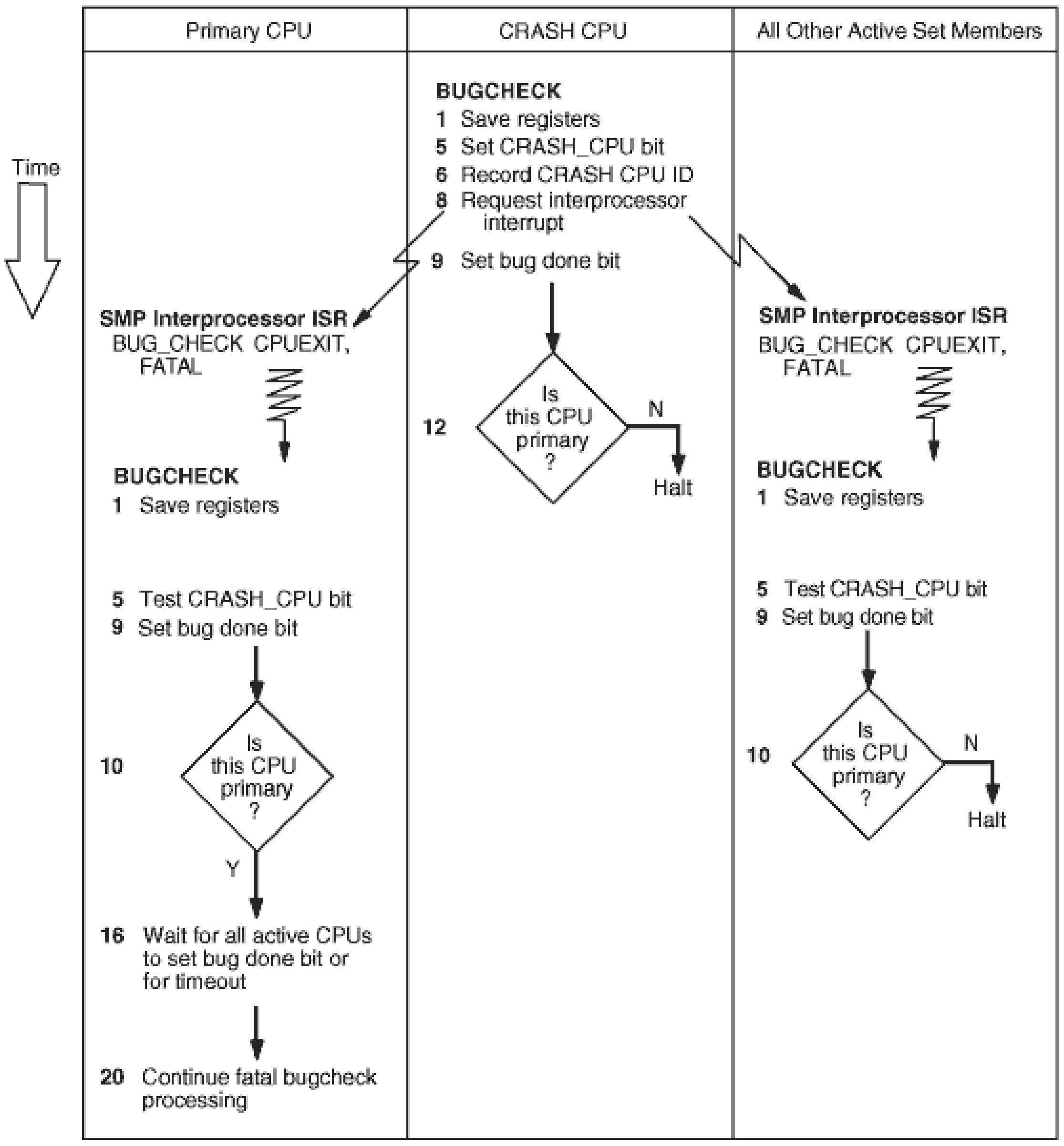


Figure 1 Steps in Fatal Bugcheck Processing on an SMP System

The following steps, which correlate to numbers in the figure, focus on SMP-specific processing and omit most steps common to uniprocessor processing.

1. BUGCHECK initially runs on the CRASH CPU and subsequently on the other SMP members. It records information in the per-CPU database.

2. The primary processor switches to BUGCHECK's hardware context; the secondary processors switch to the system hardware context.

3. BUGCHECK saves registers in the per-CPU database.

4. BUGCHECK tests whether it is running on a member of the active set. If not (a pathological and unlikely case), it proceeds to step 9 rather than taking any steps that might interfere with SMP operations.

5. If BUGCHECK is running on a member of the active set, it tests and sets the bit SMP$V_CRASH_CPU in SMP$GL_FLAGS. Only the first CPU to crash sets this bit and thus becomes the CRASH CPU.

   If the bit is already set, BUGCHECK continues with step 9. Use of the bit prevents confusion during concurrent independent crashes.

6. BUGCHECK records the ID of the CPU on which it is running in SMP$GL_BUGCHKCP as the CRASH CPU.

7. BUGCHECK acquires the CPU mutex to prevent any other processors from joining the active set.

8. BUGCHECK requests an interprocessor interrupt of every other member of the active set, specifying bugcheck as the work request type. The interprocessor interrupt service routine (ISR) generates the fatal bugcheck CPUEXIT.

9. BUGCHECK then sets its CPU ID bit in SMP$GL_BUG_DONE to indicate that it has saved its context.

10. BUGCHECK compares its CPU ID to that in SMP$GL_PRIMID to determine whether it is executing on the primary processor. If it is not, it halts. If it is in the failover set in a Galaxy system, it will be reassigned to another partition.

11. This and the following steps execute only on the primary processor because it is the only member guaranteed access to the console terminal.

    BUGCHECK sets its CPU ID in SMP$GL_OVERRIDE, adding itself to the override set. As a member of the override set, its spinlock acquisitions and releases are not subject to the usual checks.

12. BUGCHECK turns off the sanity timer to prevent sanity timer timeouts during the bugcheck.

13. On a Galaxy system, BUGCHECK reassigns any CPUs that were already stopped at the time of the bugcheck.

14. BUGCHECK waits up to a maximum of 10 seconds plus the value of SYSGEN parameter SMP_SPINWAIT (whose default value is 10 seconds) for all active members to save their context. Under normal circumstances, much of this wait does not occur. However, if one member restarts following a halt, it can take the member a significant time to complete that process and respond to the interprocessor interrupt requesting bugcheck processing. If 10 seconds passes before all members are done, BUGCHECK proceeds.

15. On a Galaxy system, BUGCHECK reassigns previously active secondary CPUs that have already saved their context.

16. BUGCHECK again waits, up to a maximum of 10 seconds plus SMP_SPINWAIT, for all active members to save their context. This wait ensures that the CRASH CPU has saved its context.

17. Still running on the primary CPU, BUGCHECK tests whether the CRASH CPU has saved its register context. If not, it uses its own per-CPU database in the following steps.

18. BUGCHECK uses the bugcheck code in the CRASH CPU's per-CPU database to select the bugcheck message text. This field is initialized to BUG$_CPUCEASED in case a problem on the CRASH CPU prevents it from recording the real bugcheck code.

19. BUGCHECK writes bugcheck information from the CRASH CPU's per-CPU database to the console terminal.

20. Running on the primary CPU, BUGCHECK then continues execution at step 13 in the earlier section titled “Fatal Bugcheck Processing on a Uniprocessor System”.

The System Dump File

System initialization code locates and opens the error log dump file SYS$SPECIFIC:[SYSEXE]SYS$ERRLOG.DMP and the system dump file SYS$SPECIFIC:[SYSEXE]SYSDUMP.DMP. If the error log dump file is not found, the error log buffers are not dumped when the system crashes or shuts down, and therefore cannot be recovered by system initialization code during the subsequent reboot.

Absence of a system dump file on the system disk does not necessarily mean a system dump cannot be written. If SYS$SPECIFIC:[SYSEXE]PAGEFILE.SYS exists, the executive writes the system dump there instead. Subsequent analysis of a dump written to the page file requires that the SYSGEN parameter SAVEDUMP be 1.

If the “dump off system disk” (DOSD) bit of the SYSGEN parameter DUMPSTYLE is set, BUGCHECK locates and opens SYSDUMP.DMP on an alternate disk. If such a dump file is found, it is used instead of a file on the system disk (SYSDUMP.DMP or PAGEFILE.SYS). If such a file is not found, the system disk dump file is used. If no dump file is found on either disk, then no dump is written. DOSD is discussed in more detail at the end of this article.

There are two types of dump:

- A **full** or **physical** dump is a dump of physical memory from the lowest numbered physical page to the highest.

- A **selective** dump is a dump of the virtual memory in use at the time of the system crash.

Both physical and selective dumps are divided into several sections, as shown in the following figure and descriptive text.

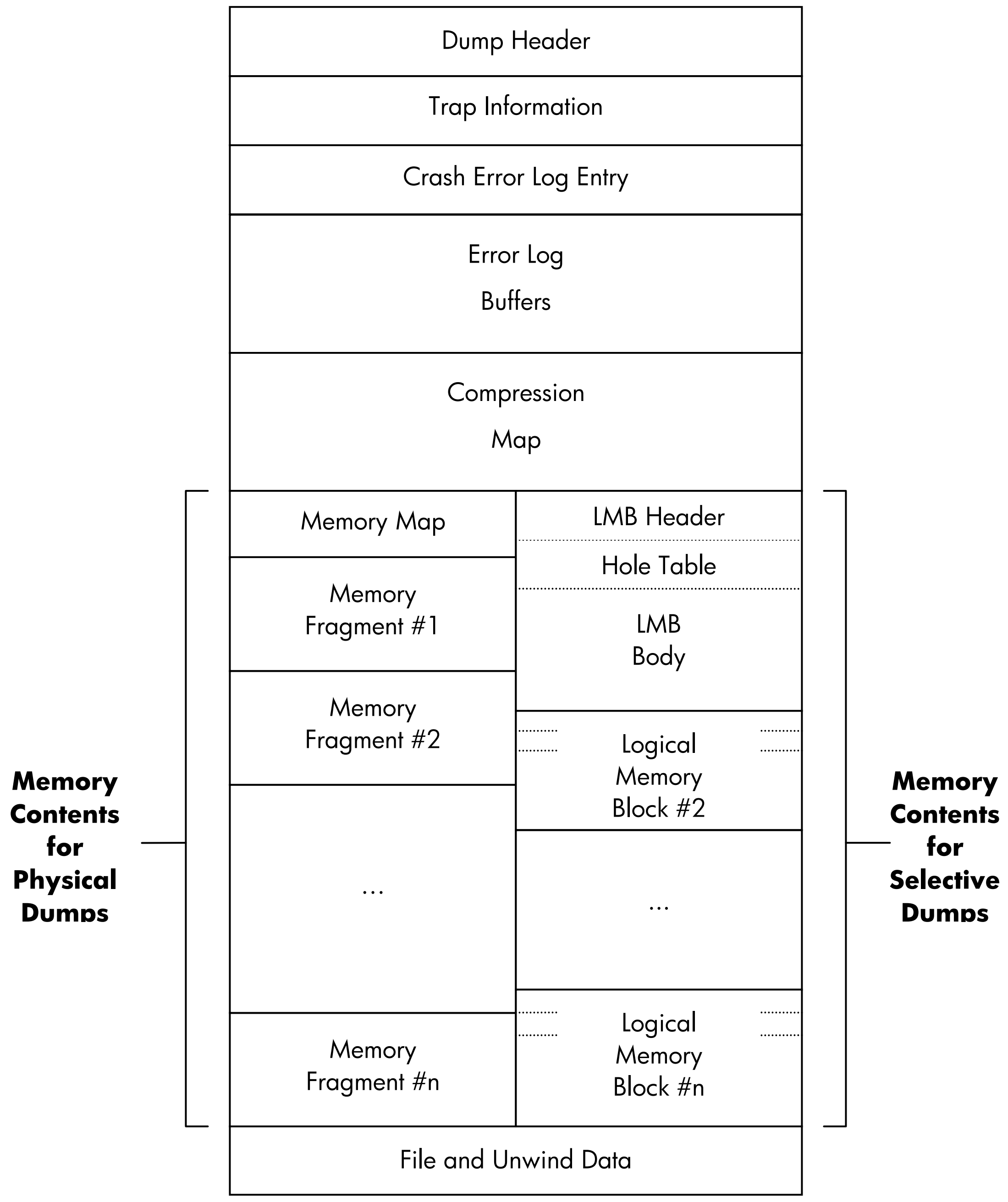


Figure 2 Layout of Physical and Selective Dumps

- Dump Header. This consists of two blocks of information describing the dump, including data such as the date and time of the system crash, the bugcheck type, the size of the dump, and so on.

- Trap Information. In case the BUGCHECK code itself incurs a trap such as an ACCVIO while it is writing the dump, blocks are set aside for the data from the trap to be saved. The saved data consists of the exception frame plus some additional fields. Currently, one block is required on Alpha, and two blocks on I64.

- Crash Error Log Entry. This is the error log entry that describes the system crash. Two blocks are used on both Alpha and I64.

- Error Log Buffers. These are the error log entries that were recorded during the period immediately preceding the system crash and have not yet been written to the error log file ERRLOG.SYS. The number of blocks required for these buffers is calculated by multiplying the SYSGEN parameters ERRORLOGBUFF_S2 and ERLBUFFERPAG_S2. (When the system reboots after a crash, SYSINIT copies the Error Log Buffers and the Crash Error Log Entry into nonpaged pool. Later, the ERRFMT process records them in ERRLOG.SYS).

- Compression Map. If a compressed dump is being written, space is allocated in the dump for the compression map. The format of this map is discussed later in this article. The BUGCHECK code calculates the worst-case size for the map – the number of blocks that would be needed if no compression were possible and if the dump file were completely filled.

- Memory Contents. The actual contents of memory is dumped next. The layout of this section depends on the type of dump (physical or selective) being written. The two layouts are described in more detail later in this article.

- File and Unwind Data. Starting in the OpenVMS release that follows Version 8.2, additional data is appended to the dump after the system reboots. This data is appended when the dump is copied by SDA during system startup. The appended data comprises:

    - Data to translate file identification numbers (FIDs) to file names
    - Unwind tables for images activated by any processes (OpenVMS I64 only)

  The file and unwind data section is described in more detail later in this article.

Physical Dumps

In a physical dump, all the memory in the system is written to the dump file, with two exceptions:

- If the SYSGEN parameter PHYSICAL_MEMORY has been set to less than the size of the system's memory, any ranges of pages marked "Available" are not dumped.

- In the case of a system crash following a MACHINECHECK due to a double-bit error in memory, the page with the error is not dumped.

In a physical dump, memory is handled in fragments. A **fragment** is a contiguous range of pages with common attributes (for example, console-owned, OpenVMS-owned, or Galaxy shared memory). In an OpenVMS Galaxy system, usually the only fragments dumped are those owned by the member that has crashed, plus shared memory. The one exception to this is the page range containing the Galaxy Configuration Tree (GCT). The GCT, which needs to be included in the dump, is normally located in the local memory of member zero. If a system crash occurs in another member, an additional fragment must be created by BUGCHECK code to describe the page range of the GCT.

The memory portion of a physical dump begins with the **memory map**. The memory map describes the memory fragments (physically contiguous ranges of pages) in the system. Up to 16 memory fragments are described by each block of the memory map. The remainder of the dump has the following contents, in this order:

- A memory fragment for the GCT (if BUGCHECK code created such a fragment)

- All console-owned pages
- All OpenVMS-owned pages
- Any Galaxy shared memory pages

Console-owned pages are written before OpenVMS-owned pages to ensure that the level 1 page table page created by the console is written early. On many Alpha platforms, this page table is in a high-numbered page, and if pages were written in ascending order, the page would often fail to be written if the dump file was even slightly too small.

An **uncompressed** or **raw** physical dump requires a dump file large enough for all of memory, plus space for the headers, error log buffers, memory map, and so on. If the dump file is too small, it is quite likely that no analysis will be possible. A **compressed** physical dump requires a dump file about half this size, but the exact size depends on the contents of memory. The memory map is not compressed. Dump compression is described later in this article.

Selective Dumps

In a selective dump, only memory in use at the time of the system crash is written to the dump file, and it is ordered by its virtual memory address. System memory is dumped first, followed by the memory of processes and global sections, in units called **Logical Memory Blocks** (LMBs). An LMB consists of a 1-block header, followed by a hole table (one or more blocks in length), followed by the actual memory contents. The hole table describes the ranges of virtual addresses not written for the section within the LMB.

The complete set of possible LMB types, and their order in a selective dump, is as follows:

1. PT Space (the page table for all of system space (S0/S1 and S2)).

2. S0/S1 space (32-bit system space)

3. S2 space (64-bit system space)

4. RAD-specific data (virtual pages in S0/S1 and S2 space that map to different physical pages in systems that support Resource Affinity Domains). There can be multiple RAD-specific LMBs in a dump.

5. Key Process page tables (one for each key process). An explanation of key processes follows later in this article.

6. Key Process memory (one for each key process). For each process, the page table LMB and memory LMB are written together as a pair.

7. Key Global pages (any global pages in the working set of at least one key process at the time of the system crash).

8. Non-key Process page tables (one for each non-key process)

9. Non-key Process memory (one for each non-key process)

10. Remaining Global pages (any global pages in the working set of a process at the time of the system crash that were not included in the "Key Global pages" LMB).

In contrast to physical dumps, it is impossible to determine exactly the size needed for an uncompressed or raw selective dump. For analysis to be possible, the dump file must be large enough to contain all system space plus the current process on the crash CPU. Best results are obtained if the dump file is large enough to accommodate all key processes and key global pages, as discussed in the next section. A compressed selective dump requires a dump file about half this size, but, as with physical dumps, the exact size depends on the contents of memory. The LMB headers and hole table blocks are not compressed. Dump compression is described later in this article.

Incidentally, **process dumps** use a format similar to selective dumps. Process dumps contain two LMBs – one for all process address space (both page tables and memory), and one for all system space memory that is pertinent to the process being dumped. Process dumps do not contain error logs or space reserved for trap information. Instead, process dumps include blocks following the dump header that contain an invocation context block and other data. This information provides a snapshot of the processor registers at the time of the failure.

Key Processes

When a dump is analyzed, the necessary information is usually found in the address space of the process that was current at the time of the system crash, or in the address space of a process that contains important system data structures. For this reason, certain processes, known as **key** processes, are dumped first in a dump. These processes, and the order in which they are dumped, are as follows:

- The current process on the CPU where the system crash was initiated
- The SWAPPER process
- Current processes on any CPUs that did not respond to the request from the crash CPU to perform a `CPUEXIT BUGCHECK` instruction
- Current processes on any other CPUs
- Processes registered as priority processes by the SYSMAN DUMP_PRIORITY ADD command
- Processes designated by Hewlett-Packard to be key processes. This list of processes is built into the BUGCHECK code and comprises the following:
    MSCPmount
    AUDIT_SERVER
    NETACP
    NET$ACP
    REMACP
    LES$ACP
    SHADOW_SERVER
- Processes in any miscellaneous wait state: for example, RWAST, MUTEX, WTBYT, and so on.

SYSMAN DUMP_PRIORITY ADD commands can be entered directly by the system manager or can be included in the installation procedures of layered products and third-party products. Careful use of SYSMAN DUMP_PRIORITY ADD commands to include all key processes ensures that a selective dump includes useful data first. This prevents the dump file being filled with the memory contents of processes that have no relevance to the system crash.

Dump Compression

Dump compression was introduced in OpenVMS Alpha Version 7.0 and has been enhanced since then, most recently in Version 7.3-1. It does not use an established compression algorithm, but one designed to be most effective with common OpenVMS memory patterns, and which can be used by the BUGCHECK code.

Following the dump header and error logs, a compressed dump contains a compression map. This is a sequence of quadwords, each describing up to 128 blocks of the compressed dump. (In a selective dump, additional quadwords describe the LMB headers and the hole tables for each LMB. These quadwords can easily be identified because bit 63, DMP$V_NOCOMP, is set).

The data in each quadword includes the final raw VBN represented by this compression map entry, the number of compressed blocks written for this entry, and flags such as DMP$V_NOCOMP. When SDA is reading the dump file, it scans the compression map looking for the entry containing the raw VBN it wants. Having counted the compressed blocks written for preceding entries, SDA can then read the exact set of blocks containing the raw VBN of interest and decompress just those blocks to get to the raw data.

The BUGCHECK code attempts to compress four blocks of raw data at a time using the algorithms described below. The compressed data is appended to a buffer until the maximum size of a compressed section (127 blocks) is reached.

The BUGCHECK code looks for the following byte patterns:

- Repeated sequences of a single value (DMP$K_REPEAT)
- An ascending sequence of values (DMP$K_INCREMENT)
- A descending sequence of values (DMP$K_DECREMENT)
- A sequence containing a limited set of values (up to 2, 4, 8, 16, 32, or 64) (DMP$K_REENCODE_n, n=1-6)
- A sequence where a small number of values (1-7) is dominant (DMP$K_BITMAP_n, n=1,3,7)
- A sequence where no compression is possible (DMP$K_NOCOMP)

For each byte pattern, the compressed data has a fixed header of three bytes. The first byte contains the pattern type (DMP$K_xxx) and some flag bits (to be described later). The second and third bytes contain a word count of the number of raw bytes represented in the compressed section.

For DMP$K_REPEAT, DMP$K_INCREMENT, and DMP$K_DECREMENT, a fourth byte contains the first character of the sequence. A minimum sequence of 16 bytes is required for the pattern to be recognized. If successive complete DMP$K_REPEAT sections are found for the same byte value, these are merged to a maximum raw size of 64 blocks.

For DMP$K_NOCOMP, the sequence of raw byte values follows the header. If successive complete DMP$K_NOCOMP sections are found, these are merged, also to a maximum raw size of 64 blocks. If two successive 64-block DMP$K_NOCOMP sections occur, the 3-byte headers are dropped, the entire 128 blocks are written to the dump file, and the DMP$V_UNCOMP bit is set in the compression map entry. Without this optimization, the compressed dump would end up being larger than the raw dump if memory patterns are such that little or no compression is possible.

DMP$K_REENCODE_n sections contain the same 3-byte header, followed by bytes containing the values that occur in the sequence (from 2**(n-1)+1 to 2**n such values). So that SDA can determine the end of the set of values, if the full set (2**n) is not used, the last value is always hex FF, even if it does not occur in the sequence. To understand how re-encode compression works, consider a sequence of bytes that contains only the hex values 17, 31, 65, A4, and E9. This can be compressed as a DMP$K_REENCODE_3 section, using binary 000 for every 17, binary 001 for every 31, 010 for 65, 011 for A4, and 100 for E9. The five raw values, followed by hex FF, are written to the dump, followed by as many 3-bit groups as there were bytes in the raw data. If a sequence of 100 bytes containing just these five values was found, its compressed size would be 3+6+(100*3+7)/8 =

47 bytes. (That's three bytes of fixed header, six bytes of raw values including the hex FF terminator, and 100 3-bit groups for the actual re-encoded values, rounded up to the next complete byte.)

DMP$K_BITMAP_n sections contain the 3-byte header, followed by bytes containing the dominant values that occur in the sequence (exactly n such values). There is no need to have a filler value, as do DMP$K_REENCODE_n sections, because any sequence of bytes that contains fewer than seven distinct values is more efficiently compressed using re-encoding. Following the set of dominant values is a set of 1-bit, 2-bit, or 3-bit groups, with one group for each byte in the raw data. The value in each group is the offset in the set of dominant bytes unless it is the maximum value for the bit group (1, 3, or 7). In that case, the value indicates that some other less common (latent) value occurs in the raw data. These latent values follow the bit groups in the same order they occur in the raw data. The number of bytes required to contain the compressed data using bitmap compression depends on the repeat counts for each dominant value.

The BUGCHECK code looks first for repeat, increment, or decrement sequences, temporarily ignoring intervening bytes. Having identified such a sequence, it then scans the ignored bytes, counting occurrences of all possible values. Using the collected statistics, it decides if a re-encoded compression is possible or if a bitmap compression would produce a better result. If no approach will compress the data, it becomes a DMP$K_NOCOMP section. The result is written to the dump, followed by any previously-identified repeat, increment, or decrement section.

But compression does not end here. In many situations, the compressed data is itself compressible, so the BUGCHECK code attempts further compression, up to four times. While it is rare to perform the maximum number of compression passes, two or three passes are common, especially with certain patterns that occur in page tables and the PFN database. Flag bits in the first byte of the 3-byte header indicate to SDA when it has completely decompressed a section.

File and Unwind Data

Beginning with the OpenVMS release that follows Version 8.2, additional data is appended to the dump after the system reboots. However, the actual dump file cannot be modified. For example, when a dump is written to a page file, any unused space in the dump file is made available for paging as soon as the system reboots. Therefore, the additional data is appended to the copy when the dump is copied by SDA. The appended data comprises the following:

1. File identification to file name translation data. Collection of this data allows SDA to display file names when analyzing dumps for the following commands:
   ```
   SHOW PROCESS /CHANNELS
   SHOW GLOBAL_SECTION_TABLE
   SHOW MEMORY /FILES
   ```

2. Unwind tables for images activated by any processes. OpenVMS I64 uses unwind tables to describe when and where registers have been saved during the execution of a procedure. The unwind tables for an activated image are in pageable sections. As a result, the tables and the data structures that describe them have often been unavailable in system dumps. Their absence affects the SHOW CALL command, preventing a complete analysis of all call frames if any frame is for a PC in an activated image. Collection of this data allows SDA to access the data it needs to display such call frames.

   The SHOW UNWIND *address* command also uses the collected data if the given address is for a PC in an activated image. However, the SHOW PROCESS /UNWIND [=ALL] command can still fail because it also relies on the data structures that describe the in-memory unwind tables; those cannot be reconstructed after the crash.

It is worth noting that because the unwind tables for loaded executive images are always resident, the SHOW UNWIND [/ALL] command is never affected. Likewise, SHOW CALL can always display call frames whose PC is in a loaded executive image.

By default, a copy of an original dump, as written at the time of the system crash, includes collection data. You can execute COPY /NOCOLLECT to override this. Conversely, a copy of a dump previously copied by SDA without collection (COPY /NOCOLLECT) does not include collection data, but COPY /COLLECT can be used to override this. Copying a dump that already contains an appended collection will always include that collection.

For all file and unwind data to be collected successfully, all disks that were mounted at the time of the system crash should be remounted and accessible to the process running SDA. SDA is usually invoked early during startup to save the contents of the dump, for example by defining the logical name CLUE$SITE_PROC to point to an SDA command procedure that includes a COPY command. If some disks are not mounted until a batch job is run, some file or unwind data might not be collected successfully. To avoid this, use the COPY/NOCOLLECT command in the CLUE$SITE_PROC command procedure; then, once all disks have been mounted, execute an additional COPY/COLLECT command to save file and/or unwind data.

Dump Off System Disk (DOSD)

On many systems, disk space can be at a premium, especially on a cluster-common system disk. For this reason, OpenVMS on both Alpha and I64 provides the ability to write system dumps to a dump file on a disk other than the system disk. To use this feature, the system manager must perform the following setup steps:

1. Set the “dump off system disk” (DOSD) bit in the SYSGEN parameter DUMPSTYLE. This is bit 2 (value 4).

2. Put the name of the disk to be used in the DUMP_DEV environment variable. On Alpha, this is done with a SET DUMP_DEV command at the console prompt. On I64, it is done using the command procedure SYS$MANAGER:BOOT_OPTIONS.COM. If there are multiple paths to the dump disk, all paths should be included in DUMP_DEV. The disk cannot be a member of a shadow set or part of a volume set.

3. On the desired dump disk, create the file [SYSn.SYSEXE]SYSDUMP.DMP using the SYSGEN CREATE /SIZE=size command. The file must be in the same system root that is used to boot the system. Using SYSGEN to create the dump file ensures that, even if the file is not contiguous, it will not be so badly fragmented that an extension header is required to record its location on the disk.

When the system crashes, if the BUGCHECK code finds the DOSD bit set in DUMPSTYLE, it attempts to access each entry in DUMP_DEV in turn and tries to locate the dump file, using the same primitive file system that is available to SYSBOOT when the system is booted. Once the BUGCHECK code finds the dump file, it writes the dump in exactly the same way as it does for a system disk dump file.

The error log dump file is always located on the system disk because SYSINIT must access it when the system reboots. The BUGCHECK code uses DUMP_DEV to locate alternate paths to the system disk if it failed over to a different path while OpenVMS was running. For this reason, if there is more than one path to the system disk, DUMP_DEV should include all paths. If the system disk is a shadow set, the BUGCHECK code attempts to write the contents of the error log buffers to all members of the shadow set. Therefore, DUMP_DEV should include all paths to all members of the system disk shadow set.

Unfortunately, on Alpha there is a limit to the number of paths that can be included in DUMP_DEV because the console provides only a single fixed-length buffer for the list. On some Alpha platforms, only a single device can be specified; on others, there is room for 4 to10 devices, depending on their type. As a result, the system manager must determine which devices to include. At a minimum, all paths to the dump device should be included if DOSD is being used. As many paths to the system disk as possible should then be added; in the case of a shadow set, start with the member that is usually the master – the member used in BOOTDEF_DEV.

On I64, the limit on the number of paths is 99, so there should be no difficulty in defining all paths to the dump disk and all paths to all members of the system disk shadow set.

For more information

For more information about the creation and analysis of system crash dumps, refer to the following OpenVMS manuals:

HP OpenVMS System Analysis Tools Manual
http://h71000.www7.hp.com/doc/82FINAL/6549/6549PRO.HTML

HP OpenVMS System Manager's Manual, Volume 2: Tuning, Monitoring, and Complex Systems
Refer to the chapter titled "Managing Page, Swap, and Dump Files."
http://h71000.www7.hp.com/doc/82FINAL/aa-pv5nj-tk/aa-pv5nj-tk.HTMl

OpenVMS Technical Journal V6

Disk Partitioning on OpenVMS: LDdriver's Secrets

Disk Partitioning on OpenVMS: LDdriver's Secrets

Jur van der Burg, HP OpenVMS Sustaining Engineering

Overview

LDdriver is a device driver that runs under the OpenVMS operating system to allow creation of virtual disks. This article describes many ways to use this driver that are not widely known.

LDdriver was developed a long time ago (around 1985) and has been improved and extended many times since then. It started as freeware (it still is) and was integrated into OpenVMS in V7.3-1.

History

LDdriver was originally written for VAX/VMS around 1985 by Adrie Sweep, a software engineer in the Netherlands. The initial version worked well, but it was not flexible in configuration and usability. Some time later the driver was changed to use so-called cloned devices to provide a more flexible way to manage the virtual disks. Over the years a number of functions have been added, increasing LDdriver's versatility, and LD has been ported to Alpha. LD has been a part of OpenVMS since VMS V7.3-1, and it is used by CDRECORD. Beginning in V8.2, LD is fully integrated. The procedures described in this article are valid for the latest version of LD (V8.0), which will be available as a separate kit and will also be integrated into the next major OpenVMS release (V8.2-1).

What is LDdriver?

As its name implies, LDdriver is a logical disk driver that allows you to use a file on any type of hard disk as a disk. For example, if you have a file called LDDISK:[VMS]DISK.DAT, you can use that file as a disk by entering the command LD CONNECT LDDISK:[VMS]DISK.DAT LDA1:. After that you have a device LDA1: on the system which you can use as a disk.

The LD system startup procedure is SYS$STARTUP:LD$STARTUP. You can put the startup command the SYS$SYSTEM:SYSTARTUP_VMS.COM file. An optional startup parameter allows you to increase the number of LD devices by specifying the controller letter to use for LD. If the controller letter is not specified, then LDA will be used (allowing up to 9999 LD devices), but if you would like an additional controller, specify B as the first parameter to get LDB.

LDdriver operates in three different modes:

- FILE mode, which allows any arbitrary file to be used as disk.
- LBN mode, which allows you to specify blocks on a disk.
- REPLACE mode, which allows access to the disk.

These modes are explained in the following subsections. If you need any help, just remember the command LD HELP. Every command is described in detail, and a couple of examples are provided as well.

FILE Mode

The most widely-used mode of operation is FILE mode. It allows an arbitrary file to be used as a disk. The only restriction on the file itself is that it must fit on a real disk; the size may be as big as the physical disk can handle. In the past, the file needed to be contiguous, but that restriction has been removed.

The setup is very simple:

1. Create a file on a physical disk.
2. Connect the file to a logical disk.
3. Use it!
4. After use, disconnect it.

For example:

```
$ ld create lddisk:[vms]disk.dsk/size=10000
$ ld connect lddisk:[vms]disk.dsk lda1:
$ ld show lda1:
%LD-I-CONNECTED, Connected $7$LDA1: to $7$DRA6:[VMS]DISK.DSK;1
```

This creates a new disk on the system (LDA1:), which can do anything a normal disk can do:

```
$ initialize lda1: lddisk
$ mount/system lda1: lddisk
%MOUNT-I-MOUNTED, LDDISK mounted on _$7$LDA1: (LDDRVR)
$ dismount lda1:
$ ld disconnect lda1:
```

This setup allows for very flexible use of disk space. Suppose you want to create an OpenVMS disk that you want to burn on a CD. All you need to do is mount the LD device, copy any files on it the way you want, dismount the disk, disconnect the file from the LD device and burn the file on a CD in any way you like. This can also be done on a PC with a variety of PC software.

Another use of LD is sharing the LD devices across different nodes in a cluster. Notice that LD devices are not MSCP-served (for a number of reasons). If sharing is to be used, the physical disk where LD's container file resides needs to be visible on other cluster members. For example:

```
$ ld connect lddisk:[vms]disk.dsk lda1:/share/log
%LD-I-CONNECTED, Connected $7$LDA1: to $7$DRA6:[VMS]DISK.DSK;1 (Shared)
```

If the same command is given on another node in the cluster, the resulting LD device can be mounted on that node as well. For this to work, the driver imposes several checks.

First, the device name has to match, as well as the allocation class, unit number, and controller letter. The device must also connect to exactly the same file, and with the same sharing options. The maximum disk size and the device geometry must be the same as well.

If one of these prerequisites is not met, a specific message will follow. Suppose a device is connected on one node, and when you try to connect it on a second node you see the following error:

```
$ ld connect lddisk:[vms]disk.dsk lda1:
%LD-F-FILEINUSE, File incompatible connected to other LD disk in cluster
-LD-F-ALLOCLASS, Allocation class mismatch
```

This means that you have an allocation class mismatch. This occurs when two nodes in a cluster have a different allocation class. Remember, the default allocation class of an LD device will be the allocation class of the node. You can change this at connect time by specifying the correct allocation class. However, you can only do this if no other LD devices for the same controller are connected.

If you look at the other node where the file was first connected, you see this:

```
$ ld show/all
%LD-I-CONNECTED, Connected _$12$LDA1: to $7$DRA6:[VMS]DISK.DSK;1 (Shared)
```

By specifying the correct allocation class, you can do what you want:

```
$ ld connect lddisk:[vms]disk.dsk lda1:/alloclass=12
%LD-F-DEVICEINUSE, Device incompatible connected to other LD disk in
cluster
-LD-F-NOSHARE, No sharing specified for file on this node
```

Oops! That does not work, either! Remember that everything has to be the same. You have to specify that you want to share the file, or the driver will reject our request. The following command will work:

```
$ ld connect lddisk:[vms]disk.dsk lda1:/alloclass=12/share/log
%LD-I-CONNECTED, Connected $12$LDA1: to $7$DRA6:[VMS]DISK.DSK;1 (Shared)
```

You can use different LD devices with different allocation classes on one node. In this case, you need a new controller letter, which can be created by invoking LD's startup procedure and specifying the controller letter as the command parameter, as follows:

```
$ @sys$startup:ld$startup b
$ ld create disk2
$ ld connect disk2 ldb5
$ ld show/all
%LD-I-CONNECTED, Connected _$12$LDA1: to $7$DRA6:[VMS]DISK.DSK;1 (Shared)
%LD-I-CONNECTED, Connected _$7$LDB5: to $7$DRA6:[VMS]DISK2.DSK;1
```

Connect, Create and Other Options

Several options can be specified at connect time to influence the appearance of the logical disk. For example, if a file was created with a size of 10000 blocks and you only want to use the first 5000, you can do so by specifying /MAXBLOCK=5000 at connect time. You can also influence the geometry by specifying /CYLINDERS, /TRACKS, and /SECTORS. Although of limited use with modern disks, these qualifiers allow you to configure a disk to exactly mimic another disk. The geometry can be copied from an already mounted disk with /CLONE, which can also be used to create a container file that mimics a real disk, as follows:

```
$ ld create/clone=$8$dua14: lddisk:[vms]cloned.dsk
$ ld connect cloned lda2/log
%LD-I-CONNECTED, Connected $12$LDA2: to $7$DRA6:[VMS]CLONED.DSK;1
```

This procedure will create a logical disk LDA2 with exactly the same properties as device $8$DUA14:. Notice that the geometry information is stored in the container file, so a subsequent disconnect/connect will restore the same disk parameters. The geometry information can be overridden by specifying /NOAUTOGEOMETRY, in which case the geometry will be calculated by the driver.

The geometry information will also be saved in the container file if /SAVE was specified during connect.

LD devices can be write-protected using the LD PROTECT command. You can make this protection permanent by adding /PERMANENT, which will store the write-protect status in the container file. This way it will be possible, for example, to emulate a CDROM; on a subsequent CONNECT command, the write-protect status will be restored.

Normally, a file is connected to an LD device by explicitly specifying which device you want to get. If you don't do this, the driver will assign a unit number and inform you about it. If you specify /SYMBOL as well, you will get the assigned unit number in a DCL symbol:

```
$ ld connect disk.dsk/symbol
%LD-I-UNIT, Allocated device is $12$LDA22:
$ show symbol ld_unit
  LD_UNIT = "22"
```

This procedure is useful in command procedures, where the actual device name is unimportant. The symbol makes it easy to reference such a device.

LBN Mode

In LBN (Logical Block Number) mode, a physical disk can be accessed in several parts, as specified by a LBN range. Look at it as partitioning a disk without any file structure on it. For example:

```
$ ld connect $1$dga1: lda1:/lbn=(start=0,count=1000)/log
%LD-I-CONNECTED, Connected $7$LDA1: to _$1$DGA1: (LBN Mapping: Start=0
End=999)
$ ld connect $1$dga1: lda2:/lbn=(start=1000,end=2000)/log
%LD-I-CONNECTED, Connected $7$LDA2: to _$1$DGA1: (LBN Mapping: Start=1000
End=2000)
```

This procedure uses two fragments of device $1$DGA1 for devices $7$LDA1 and $7$LDA2. The range of LBNs can be specified in either of the following ways:

- A starting LBN with a block count
- A starting LBN and an ending LBN

The driver will check for an overlap of LBNs, so any attempt to use a range already in use will result in the following error:

```
$ ld connect $1$dga1: lda3:/lbn=(start=600,end=2000)/log
%LD-F-DEVICEINUSE, Device incompatible connected to other LD disk in
cluster
-LD-F-RANGEINUSE, LBN range already in use
```

This way of partitioning a disk is the most efficient way to use the physical device, since the overhead of unused blocks is zero. The physical device can be divided in as many parts as you need, as long as the system's resources will allow them.

Another way to use this is to map a range of blocks from a foreign volume, for example a UNIX file system. Use your imagination!

As with FILE mode, these LD devices can be shared in a cluster:

```
$ ld connect $1$dga1: lda3:/lbn=(start=5000,count=5000)/log/share
%LD-I-CONNECTED, Connected $7$LDA3: to _$1$DGA1: (LBN Mapping: Start=5000
End=9999) (Shared)
```

The same restrictions apply as with FILE sharing: the device name, unit number, controller letter, and allocation class must match the remote node. The range and physical device have to match as well, of course. Notice that the range of LBNs in use is checked on all the nodes in the cluster that have an interest in the physical device, so if any block in the specified range is already in use, an error will be generated. This is really a tricky thing to check; for more information about this check, see Internals.

To be able to use LBN connect, the physical device must not be in use anywhere in the cluster. The driver will enforce a check on this, and if the check fails an error will follow. After the first LD device is connected with LBN, the physical device will not be available for any other use in the cluster; that is, a mount on any node will fail until the last LD device disconnects. For example:

```
$ mount/over=id $1$dga1:
%MOUNT-I-OPRQST, device already allocated to another user
%MOUNT-I-OPRQST, device _$1$DGA1: (LDDRVR) is not available for mounting.
```

If any failure to use the physical device occurs on any node, make sure that all LD devices are disconnected from this device. The driver will go to great lengths to protect the user against any errors.

REPLACE Mode

REPLACE mode is not a form of partitioning, but a way to create access to the physical device using LDdriver. You will understand how to use this when you read about I/O Tracing and Watchpoints, which describe how to perform a real-time trace of all I/O requests.

The following example creates a device ($7$LDA1) that will direct all I/O to the physical device $1$DGA1:

```
$ ld connect $1$dga1: lda1:/replace/log
%LD-I-CONNECTED, Connected $7$LDA1: to _$1$DGA1: (Replaced)
```

Replaced devices can also be shared:

```
$ ld connect $1$dga1: lda1:/replace/log/share
%LD-I-CONNECTED, Connected $7$LDA1: to _$1$DGA1: (Replaced) (Shared)
```

The same restrictions apply for sharing as with FILE and LBN mode: the device name, unit number, controller letter, and allocation class must match. The physical device will be made unavailable by means of a lock.

I/O Tracing

One of the most powerful and little known features of LDdriver is I/O tracing. For any mode that an LD device operates, a real-time trace of all I/O activity can be created. This is a simple example:

```
$ ld connect disk.dsk lda1
$ ld trace lda1
$ mount lda1: testdisk
%MOUNT-I-MOUNTED, TESTDISK mounted on _$7$LDA1: (LDDRVR)
$ ld show/trace lda1
         I/O trace for device $7$LDA1:
    26-APR-2005 22:18:36.28 on node LDDRVR::

Start Time  Elaps   Pid        Lbn     Bytes  Iosb    Function
--------------------------------------------------------------
22:18:32.65 00.00 09C00227          0      0 NORMAL  PACKACK|INHERLOG
22:18:32.65 00.01 09C00227          1    512 NORMAL  READPBLK
22:18:32.66 00.00 09C00227       1034    512 NORMAL  READPBLK
22:18:32.67 00.00 09C00227       5007    512 NORMAL  READPBLK
22:18:32.67 00.00 09C00227       5008    512 NORMAL  READPBLK
22:18:32.67 00.00 09C00227       5002    512 NORMAL  READPBLK
22:18:32.67 00.00 09C00227       5002    512 NORMAL  WRITEPBLK
22:18:32.68 00.00 09C00227       5002    512 NORMAL  WRITEPBLK
22:18:32.69 00.00 09C00227       5003   1536 NORMAL  READPBLK
22:18:32.69 00.00 09C00227       5006    512 NORMAL  READPBLK
22:18:32.69 00.00 09C00227       5010    512 NORMAL  READPBLK|EXFUNC
22:18:32.69 00.00 09C00227       5000    512 NORMAL  READPBLK|EXFUNC
22:18:32.69 00.00 09C00227          0      0 NORMAL
PACKACK|BYPASS_VALID_CHK
22:18:32.69 00.00 09C00227       5002    512 NORMAL  READPBLK|EXFUNC
22:18:32.70 00.00 09C00227       5016    512 NORMAL  READPBLK|EXFUNC
22:18:32.70 00.00 09C00227       5023   1024 NORMAL  READPBLK
22:18:32.70 00.00 09C00227       5016    512 NORMAL
WRITEPBLK|EXFUNC|DATACHECK
```

The default display shows a timestamp, the elapsed time of the request, and so forth. Various qualifiers allow you to modify the display, such as /IOSB=LONGHEX to show the real contents of the IOSB, and /FUNCTION=HEX to show the function code without any translation.

Another powerful function is to trace FDT (Function Decision Table) calls to the driver. These always happen but are not easy to see. These requests happen in the first part of calling a driver and are used, for example, to validate parameters. But the I/O completion can occur directly from these FDT routines, so that the I/O request is normally not noticed. FDT tracing is not implemented on the VAX version of LDdriver.

The timing can be measured not only by the normal timestamps, but also by means of the SCC (System Cycle Counter), which is a hardware register in the processor architecture that can be used for accurate timing measurements. This counter is only available on Alpha and IA64, so accurate tracing is not implemented on the VAX version of LDdriver. Accurate tracing must be enabled at the time that the trace buffer is activated (using the LD TRACE command); it is not the default because activating this option may impose a small performance hit. The SCC counter is specific for each processor; therefore, in a multiprocessor system, the driver may need to reschedule the completion of an I/O request to the same processor where the request was initiated to get an accurate measurement.

If you enable FDT and accurate tracing, you get this:

```
$ ld notrace lda1
$ ld trace/fdt/accurate lda1
$ dir lda1:[000000]/out=nl:
$ ld show/trace/fdt/accurate lda1
         I/O trace for device $7$LDA1:
    26-APR-2005 22:36:35.69 on node LDDRVR::

Start Time  Elaps  uSecs   Pid        Lbn     Bytes  Iosb    Function
```

```
----------------------------------------------------------------------
22:36:29.10 00.00      0 09C00227          0      0 FDT     ACCESS
22:36:29.10 00.00      0 09C00227          0      0 FDT
ACCESS|ACCESS|EXFUNC
22:36:29.10 00.00      0 09C00227          0      0 FDT
READVBLK|EXFUNC
22:36:29.10 00.00    422 00000000       5000    512 NORMAL
READPBLK|EXFUNC
22:36:29.10 00.00      0 09C00227          0      0 FDT
DEACCESS|EXFUNC
```

Notice that the driver is called more times to do the FDT work. The real I/O request had an elapsed time of 422 microseconds; you can measure the timing with much finer granularity this way.

Of course, all trace data can be saved to disk, either as ASCII or as binary data. You need binary data to process the trace data later. You can specify a block size, which will force LD to create a new version of the output file once the number of blocks have been reached. Together with a version limit, this provides continuous tracing without having to worry about filling up a disk, while still capturing a predictable amount of data.

The trace data can also be viewed in real time with LD SHOW/TRACE/CONTINUOUS. This command reads the trace data from the driver and resets the trace buffer after reading it. As soon as there is new data available, the driver notifies LD driver so that you can get it. The viewer can be stopped either by Control-C or by the LD TRACE/STOP LDA1 command issued from another terminal.

The default size of the trace buffer is 512 entries, but this can be as big as non-paged pool allows. The amount of bytes taken for the buffer is charged against the byte count quota of the process issuing the command.

Watchpoints

Watchpoints provide another tool that can help you debug various pieces of software. A watchpoint is a logical block on disk that will generate a special action as soon as it is hit by an I/O request. Another form of watchpoints is VIRTUAL watchpoints - virtual block numbers inside a file on disk. You can choose any of the following actions to occur when a watchpoint is hit:

- Error

  A watchpoint with the Error characteristic will return a predetermined error (specified by the user). If you want to simulate an error on a disk, you can do this:

  ```
  $ ld watch lda1 123/action=error
  $ ld show/watch lda1
  Index LBN     Action     Function         Error return code
  ---------------------------------------------------------------------
   1      123  Error     READPBLK             02A4 (BUGCHECK)
  ```

  As soon as logical block 123 is hit by an IO$_READPBLK request, the I/O will be terminated with an SS$_BUGCHECK error:

  ```
  $ dump lda1:/block=start=123
  %DUMP-E-READERR, error reading LDA1:
  -SYSTEM-F-BUGCHECK, internal consistency failure
  ```

  The function code to trigger the watchpoint is by default an IO$_READPBLK. You can specify any function with /FUNCTION=ALL, or for a unique function by specifying, for example, /FUNCTION=CODE=13.

- Suspend

  The SUSPEND action means that a request will be suspended until it is released by an LD command. If you ever wonder where a request came from, you can let it run into a watchpoint, and use SDA to see what is going on. Multiple suspends will be queued, and can be released by a simple command.

```
$ ld watch lda1 10/action=suspend
$ ld show/watch lda1
Index LBN     Action     Function         Error return code
-------------------------------------------------------------------
 1       10  Suspend   READPBLK
```

```
$ spawn/nowait/input=nl: dump lda1:/block=(count=1,start=10)
%DCL-S-SPAWNED, process SYSTEM_50 spawned
$ spawn/nowait/input=nl: dump lda1:/block=(count=1,start=10)
%DCL-S-SPAWNED, process SYSTEM_9 spawned
$ ld show/watch lda1
Index LBN     Action     Function         Error return code
-------------------------------------------------------------------
 1       10  Suspend   READPBLK
             Suspended process: 09C00247
             Suspended process: 09C00248
```

The I/O requests are suspended by the driver and the processes are simply waiting for them to be completed. This can be done with one command:

```
$ ld watch/resume lda1
```

These I/Os will then complete, and the processes will finish.

- OPCOM

  The OPCOM action allows an OPCOM message to be created when a watchpoint is hit. The message includes the image name, process id, device name, I/O function code, and logical block number:

```
$ reply/enable
$ ld watch lda1 1/action=opcom
$ ld show/watch lda1
Index LBN     Action     Function         Error return code
-------------------------------------------------------------------
 1        1  Opcom     READPBLK
$ dump lda1:/block=(count=1,start=1)
%%%%%%%%%%%  OPCOM  26-APR-2005 23:15:30.23  %%%%%%%%%%%
Message from user SYSTEM on LDDRVR
***** LDdriver detected LBN watchpoint access *****
PID:      09C00227
Image:    DCL
Device:   $7$LDA1: (LDDRVR)
Function: 000C
LBN:      1
```

  The OPCOM type may be combined with any other type of watchpoint.

- Crash

  As its name implies, this action will cause the system to crash when the selected watchpoint is hit. It is a "big hammer" approach for troubleshooting, but it can be handy at times. For example:

```
$ ld watch lda1 1/action=crash
%LD-F-DETECTEDERR, Detected fatal error
-SYSTEM-F-NOCMKRNL, operation requires CMKRNL privilege
```

  This example shows what happens when you do not have the CMKRNL privilege. As crashing a multiuser system is very serious business, a high privilege level is required for this operation to succeed. The driver checks the issuing process for this privilege, and if it is not there, the above error will result.

```
$ set proc/privilege=cmkrnl
$ ld watch lda1 1/action=crash
$ ld show/watch lda1
Index LBN     Action     Function         Error return code
--------------------------------------------------------------------
 1        1  Crash     READPBLK
$ dump lda1:/block=start=1
```

The privilege is enabled, but there is no response here. Fortunately, you can see something on the console:

```
**** OpenVMS Alpha Operating System V8.2     - BUGCHECK ****
** Bugcheck code = 000008F4: RSVD_LP, Reserved for layered product use
** Crash CPU: 00    Primary CPU: 00    Active CPUs: 00000001
** Current Process =   SysDamager
** Current PSB ID = 00000001
** Image Name = $10$DKA600:[SYS0.SYSCOMMON.][SYSEXE]DUMP.EXE;1

**** Starting selective memory dump at 26-APR-2005 23:29...
```

Note that multiple watchpoints can be created using one command:

```
$ ld watch/action=error=%x84/function=write lda1 1,2,3,4,5
$ ld show/watch lda1
Index LBN     Action     Function         Error return code
--------------------------------------------------------------------
 1        1  Error     WRITEPBLK           0084 (DEVOFFLINE)
 2        2  Error     WRITEPBLK           0084 (DEVOFFLINE)
 3        3  Error     WRITEPBLK           0084 (DEVOFFLINE)
 4        4  Error     WRITEPBLK           0084 (DEVOFFLINE)
 5        5  Error     WRITEPBLK           0084 (DEVOFFLINE)
```

A useful watchpoints is when a device is put into mount verification:

```
$ ld watch lda1 1/action=error=%x84/function=code=%x0808
$ ld watch lda1 10/action=error=%x84/function=read
$ ld show/watch lda1
Index LBN     Action     Function         Error return code
--------------------------------------------------------------------
 1           Error     PACKACK|INHERLOG     0084 (DEVOFFLINE)
 2       10  Error     READPBLK            0084 (DEVOFFLINE)
$ spawn/nowait/input=nl: dump lda1:/block=(count=1,start=10)
%DCL-S-SPAWNED, process SYSTEM_253 spawned
$
%%%%%%%%%%%  OPCOM  26-APR-2005 23:44:41.37  %%%%%%%%%%%
Device $7$LDA1: (LDDRVR) is offline.
Mount verification is in progress.


$ ld nowatch lda1
$
%%%%%%%%%%%  OPCOM  26-APR-2005 23:44:50.62  %%%%%%%%%%%
Mount verification has completed for device $7$LDA1: (LDDRVR)
```

```
VIRTUAL watchpoints are watchpoints for a given virtual block in a file:
```

```
$ copy sys$system:copy.exe lda1:[000000]
$ ld watch/file=lda1:[000000]copy.exe lda1: 10
$ ld show/watch lda1
Index LBN     Action     Function         Error return code
--------------------------------------------------------------------
             $7$LDA1:[000000]COPY.EXE;1
```

```
 1       10  Error     READPBLK             02A4 (BUGCHECK)
$ dump lda1:[000000]copy.exe/block=(count=1,st=10)
%DUMP-E-READERR, error reading LDA1:[000000]COPY.EXE;1
-SYSTEM-F-BUGCHECK, internal consistency failure
```

As long as a virtual watchpoint is set, it will be impossible to dismount the disk containing the watchpoint:

```
$ dismount lda1:
%DISM-W-CANNOTDMT, LDA1: cannot be dismounted
%DISM-W-USERFILES, 1 user file open on volume
$ ld nowatch lda1/index=1
$ dismount lda1
```

Callable Interface

All the LD commands are also available from a user application program by issuing QIO requests to LDdriver. To make this possible, a C header file (LDDEF.H) that contains all the necessary definitions is included in the LD driver kit. As of OpenVMS version V8.2-1, this definition file will be included in the system library, so an external file is not needed anymore.

The command LD HELP Driver_functions provides the details you need to implement the callable interface.

Internals

At first sight, operation of the driver seems quite straightforward, and in fact, it is. Once a drive is connected to a file there is not much more to do than getting the request and passing it in a modified form to the physical disk driver. Of course, getting the mapping right and splitting the request into multiple segments if the container file is not contiguous is some extra work, but it's not that difficult. For LBN mode we simply needed to add an offset to the block in question, and make sure we didn't step outside the capacity of the disk.

The tricky part is to protect innocent users from shooting themselves in the foot. For example, on one node in a cluster you connect file FILE.DSK to device LDA1 with the intent to share it. Then on another node you connect the same file to LDA2. If the driver allowed this, it could be a recipe for a disaster, so strict rules are in place for sharing container files. This is all coordinated by using the fork level interface of the distributed lock manager.

For every connected file, two locks are taken: one for the file (which includes the volume lock name and the file identification) and one for the LD device itself. This makes it possible to verify all possible combinations for the correct behavior. The lock value blocks of the involved locks are used extensively to pass parameters between the various nodes to check everything.

One extreme case was found during development of LBN mode, where we needed to determine whether a given range of LBNs was already in use. How could we do this using locks? We needed to be able to ask another node if it is using a range, but there may be dozens of ranges already there, and that will not fit in a lock value block that is used for the communication. A way around it might have been to create a server process that could do the check, but it we knew it would be nicer if we could do this without it. Such a process would mean more things to check for, the error handling would be more complicated, and it would take additional resources as well.

Our solution still uses locks. By using one lock and putting the range of blocks we intend to use in the lock value block we can trigger a blocking AST routine on any node that is already using the proposed device. This blocking AST routine can then do the check, and set a go/nogo bit in the lock value block as a return signal. If even just one node objects to the range that we proposed, connecting a drive will fail. Very careful design was required in order to use the right lock modes to prevent loops by repeating blocking AST routine calls. Finally a working model was created that is efficient, does not need an external process and is contained completely within the driver.

For LBN and REPLACE mode, a lock is used to prevent the physical device from being used by anything other than LDdriver. Since connects and disconnects may occur in any order and on multiple nodes, the driver will attempt to take out the lock itself. If that fails, it will queue the lock so that if another user disconnects, any LD device that was connected to this physical device will still maintain a lock somewhere in the cluster. For example, if we are connecting on one node and we are the first one, we will get the lock. Then when we connect to the same device on another node the lock will not be granted, and the request will be stalled. If we then disconnect on the first node the lock will be granted, so that we continue to be protected against other uses than LDdriver.

Conclusion

LDdriver has come a long way. After many hours of midnight puzzling and thinking, a lot of functions have been added, thanks to a number of people in the outside world who have given us great suggestions.

As development continued we have been able to keep the VAX version of the driver pretty much in synch with the Alpha/IA64 version. It only lacks accurate- and FDT tracing.

By allowing a great deal of source code transparency between Alpha and I64, there was no need for a special port to IA64. In fact, there is not even one conditional in the driver to make a distinction between the architectures. This shows what a marvelous job the people in OpenVMS engineering have done with the I64 port.

For more information

If you have questions after reading this article, or if you have any suggestions, we are always interested in possible improvements. If you happen to find a bug, just let us know and we will do just about anything to fix it.

When the V8.0 LD kit is released, it will include examples as well as source code and will be posted to the OpenVMS freeware archive at http://www.hp.com/go/openvms/freeware.

The author can be contacted at lddriver@hp.com.

Threaded Tests in the OpenVMS Cluster Test Manager (CTM)

Threaded Tests in the OpenVMS Cluster Test Manager (CTM)

Richard Stammers

Overview

Testing extremely complex systems, such as large OpenVMS clusters involving SMP systems, is a race against time. It is daunting, in fact, when you view this testing as a "combination search" of all the various states that the cluster and the systems can be in. Moreover many of the hardware and software components of such systems are, by their nature, autonomous; they work together and co-operate heterarchically rather than having some "super system" or "super process" that controls their operations. As a result, most combinations of states are theoretically possible and hence must be tested. As Murphy would have it, "Anything that can go wrong will!" In practice, we have to cheat and connive to accomplish all the required testing in a reasonable time. We have to force the systems into all the nasty conditions that otherwise occur so infrequently that otherwise would be missed in the finite amount of time available for testing. There are limits to the extent that we can "cheat and connive." The components under test cannot be changed, and the tests have to be valid in the context of OpenVMS. Threads provide an effective way to contrive tests that explore various combinations of system states, and are essential to properly testing SMP systems. This article discusses in general terms how threaded tests are designed and used, showing specific examples of threaded tests that are employed in CTM, the OpenVMS Cluster Test Manager.

Introduction

Threaded tests in CTM are organized and designed so that multiple instances of routines used in the test can operate concurrently. During testing, it is important that we do as much as possible with the time that is available. As with all types of applications, threads can be used to provide a significant improvement in performance, especially on multiprocessor systems, where threads can run concurrently on separate processors. This parallelism also allows multiple execution threads to run concurrently in the context of the test, thus allowing us to verify that execution threads running on different processors are synchronized and maintain coherency.

Writing a test is a little different from writing other types of application programs. A good application programmer uses his knowledge of the system to make the application run as quickly and efficiently as possible. Tests are often written with exactly the opposite intention. Problems or defects show up much more quickly when the system is deliberately stressed and forced into difficult situations, so while application programmers use their knowledge of a system to avoid these situations, a person writing a test must approach things with exactly the opposite intention, designing the test so that nasty situations occur as frequently as possible.

CTM tests the OpenVMS operating system, including hardware and software components, at the application level. And because the threaded tests in CTM use it, they also test the multithreading runtime library. This is an important consideration.

No matter how zealous the tester is to try and break the system, the tests always have to abide by the rules! Moreover there are human factors to be considered. Tests are written and performed to ensure the quality of the final product; finding a problem or a defect is only the first step towards fixing it.

In some ways, the multithreaded runtime library is like a “mini operating system;” the results, logs, and traces from a threaded test are often complex and voluminous. The person writing the tests must guard against undue complexity and zeal must be tempered with the practical consideration of how the test will be used to pin-point and remedy any problems and defects that are found.

One final point to be considered when designing any test, particularly tests that make use of threads, is that a test that fails to find a problem or a defect, or that detects spurious problems that do not in fact exist, is worse than useless. On the one hand, time and effort is expended in writing and running the test, only to provide a false sense of security that everything is alright. On the other hand, that time and effort would be wasted in tracking down a “problem” that doesn’t exist. The design of any test should verify beyond any reasonable doubt that the test itself is working properly.

General Methodology

As described in an earlier OpenVMS Technical Journal article, CTM is not a test, but rather it is a framework which permits the control and running of multiple different tests in an OpenVMS cluster environment. To clarify this distinction, we refer to the framework in which the tests are run as the “test harness,” as distinct from the actual tests that are run by the harness under the direction of the test engineer.

The threaded tests in CTM run under a “gang scheduling” architecture (see Figure 1). Central to each of the tests is the main thread, which communicates with and is controlled by the test harness. It receives from the harness any parameters that control the test, validates them, and uses them to perform as much of the test initialization as possible prior to the start of the test. In fact, as much of the work inside CTM is done in the initialization stage as is possible (for example, setting up all the data buffers and data patterns the test will use, using alternating data patterns instead of poisoning buffers between data cycles, and so forth). When the actual test runs, it can run as quickly as possible. The main thread is also responsible for logging performance information and key events to the CTM TEL (Cluster Test Manager Test Event Log).

Having completed the initialization, the main thread then creates a number of client threads according to a parameter passed to it from the harness. The main thread controls each of the client threads using a “gang scheduling” method. Prior to each phase of the test, each of the client threads reports to the main thread that it is ready. When all the client threads have reported that they are ready, they are told at the same time to start that phase of the test. The main thread then waits for each client thread to report back that it has completed that phase of the test. When all of the client threads have reported that they completed a phase of the test, the main thread examines the results from each of the client threads. If there are any errors or inconsistencies, the test is stopped. If there are no problems, the main thread gang schedules all the client threads through the next phase of the test, and this process continues until either an error is detected or the main thread is told to stop the test by the CTM test harness.

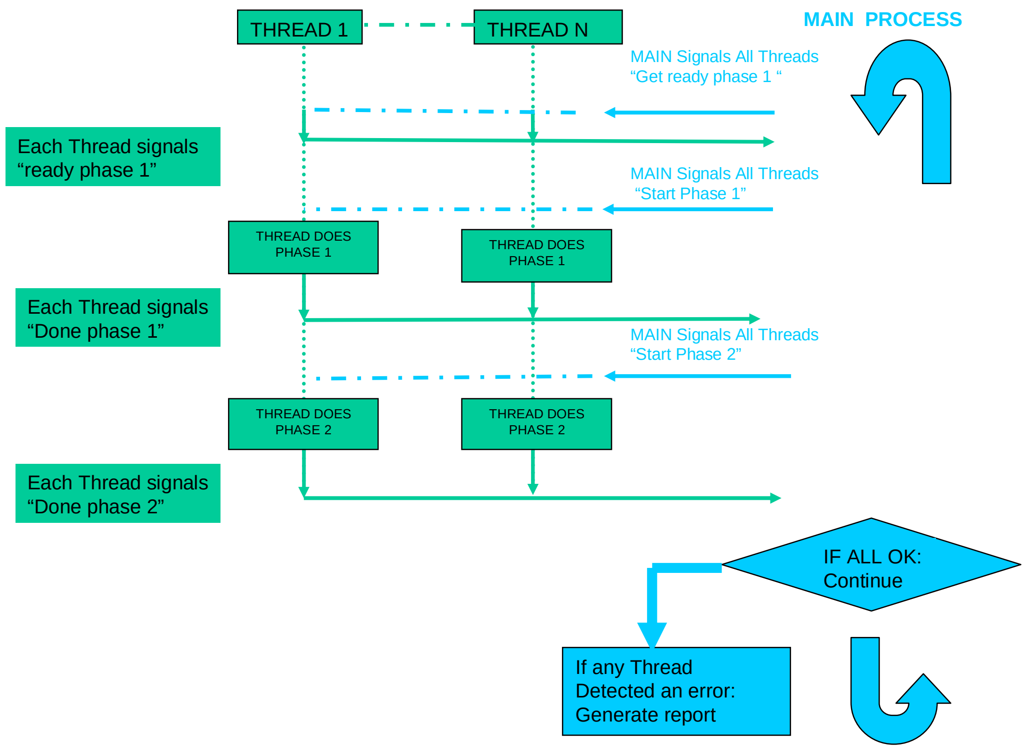

THREAD 1
THREAD N
MAIN PROCESS
MAIN Signals All Threads "Get ready phase 1 "
Each Thread signals "ready phase 1"
MAIN Signals All Threads "Start Phase 1"
THREAD DOES PHASE 1
THREAD DOES PHASE 1
Each Thread signals "Done phase 1"
MAIN Signals All Threads "Start Phase 2"
THREAD DOES PHASE 2
THREAD DOES PHASE 2
Each Thread signals "Done phase 2"
IF ALL OK: Continue
If any Thread Detected an error: Generate report

Figure 1 - Gang Scheduling

All of the threaded tests in CTM have only two phases, Phase I and Phase II, which are repeated indefinitely. The operations that are performed by each of the threads are the same, subject to specific differences, such as addresses of memory areas or names of files, which are set up for them and passed to them by the main thread. That is, in Phase 1 all the client threads do something, and in Phase II all the client threads check the results of what they did. Subject to the limitations of scheduling by the multithreaded runtime library, all the client threads do Phase 1 at the same time, and all the client threads do Phase II at the same time.

From a practical programming standpoint, this general methodology for the CTM threaded tests confers several benefits. Because the "gang scheduling – Phase 1 – Phase 2 methodology" is common to all the CTM threaded tests, a huge amount of code concerned with initializing and synchronizing the tests through these phases is common to all the tests. And because all the client threads are doing their work at the same time, we can contrive tests to make sure that memory coherency is maintained when multiple different threads are accessing the same areas of memory. In fact, all of the CTM threaded tests are designed like a framework within a framework. The tests themselves exist inside the framework of the CTM harness, and all the threaded tests are constructed inside a common "gang scheduling – Phase 1 – Phase II" methodology. In principle, a new test can be written by just changing the functionality performed in Phase 1 and Phase II. Of course, not all tests or ideas for testing lend themselves to this approach. This article only explains why and how this is done with CTM.

Existing Threaded Tests in CTM

The threaded tests in CTM were all designed and written for a particular purpose, but there is a considerable amount of commonality and even overlap between them because they were developed over time to meet testing needs as they arose. As the tests developed, we began to see how to write better, more generalized, and more powerful tests. Commonality and overlap is not a bad thing where testing is concerned. Despite all efforts to approach testing from a logical and analytical point of view, testing in such a complex environment can never be a totally exact science. Sometimes one test will find something that might be missed or might take much longer to find with another similar test. By far, the best general testing strategy is to employ as many and as varied a repertoire of tests as possible.

The strange names for some of the tests warrant some explanation. The names do in general convey something of what the test is doing, and are certainly easier to remember than for example CTM_T001, CTM_T002 etc. etc. would be. Also, because of their common framework a lot of cutting and pasting and text substitutions were used to write them and in this context it is useful to have names that don't frequently occur in code. If nothing else then names like BOZO and YOYO definitely meet these criteria.

CTM_BYTEM (Byte Mode Memory Coherency Exerciser)

CTM_BYTEM was an early test designed to test native byte mode access on Ev6 machines, and by implication to test emulation of byte mode access on machines that do not provide native byte mode access. The test is extremely simple but serves to illustrate how the CTM threaded test structure operates.

In this test, multiple client threads target a contiguous area of memory. The principle parameters for the test are the size of the memory area and the number of threads. If there are N such client threads (thread0, thread1,…threadN), then thread 0 operates on bytes at offsets 0, N, 2N, etc., inside the memory area. Thread1 operates on bytes at offsets 1, N+1, 2N+1, etc., inside the memory area, and so on for each thread. This way, each client thread targets a unique set of bytes within the memory area.

Each thread has a unique 8-bit pattern assigned to it, which is the ones complement of its thread number. (For example, the bit pattern for thread 1 is "11111110".) Prior to Phase 1, the main thread zeros the memory area. Then, in Phase 1, each of the client threads iterates through the memory area writing the bytes it operates on to its own specific bit pattern.

In Phase II, each of the client threads reads the bit patterns in the entire memory region to ensure that they are set to the value corresponding to the client thread which operates on them

This makes sure that native byte-mode operations work OK, but a simpler test would suffice for this. The use of threads expands the nature of the test to ensuring that, in the process of modifying only a byte, the other bytes in the quad word containing the byte are not affected, and also, in a multiprocessor system, coherency is maintained when one or more threads is attempting to modify a byte in the same quad word.

Under OpenVMS Version 7.0 and later, a two-level scheduling model is employed. Threads are scheduled to run on virtual processors the same way that OpenVMS schedules virtual processors to run on physical processors. Most thread scheduling takes place in user mode, which is much more efficient than switching contexts, which involves the operating system. The OpenVMS scheduler can schedule virtual processes onto separate processors of a multiprocessor machine. An upcall mechanism handles the cases where OpenVMS detects an event that affects the scheduling of a thread, calling up to the threads scheduler to notify it of the change in the status of a thread.

In the case of this test, if the system under test has N processors, then the recommended value of the parameter for the number of client threads is also N. If only the client threads are running, very simple arithmetic is required to figure out how often, on average, they try to modify a byte in the same quad word. Considering the number of processors, then adjusting the size of the memory area and the number of client threads provides a powerful test mechanism that we can use to force this situation to occur frequently.

This test structure also allows us to verify that the test is working as expected and that clients threads are in fact trying to access the same quad word with the expected frequency. The tests are written in C. Normally, the memory area being accessed by the threads would be compiled with the volatile qualifier to ensure that the memory was protected between the client threads. If the volatile qualifier is removed, then a number of corruptions should correspond to the number of times that two client threads were accessing a byte in the same quad word.

CTM_BOZO

CTM_BOZO uses threads to get as much testing work done as possible on a multiprocessor system. The purpose of the test is to have many threads performing memory tests (allocating and deallocating memory) and/or performing file I/O operations. Because the threads work in the context of a single process, they work more quickly than the corresponding number of processes; this allows them to test more allocation/deallocation sequences in the same amount of time.

As with the other threaded tests, the main thread communicates with the test harness, logs performance information, initializes data buffers and patterns, creates the client threads, and controls them using gang scheduling.

In Phase I, when the main thread instructs the client threads to start, they optionally allocate memory, write a data pattern to the memory and/or open a data file and write a data pattern to the file.

In Phase II, each of the client threads tests the results of the operations it performed in Phase I, either by verifying the data pattern in memory, and/or reading and verifying the data that was written to file. In Phase II, the client threads release the resources they were using in Phase I, thus assuring that resources are allocated and deallocated with each test iteration.

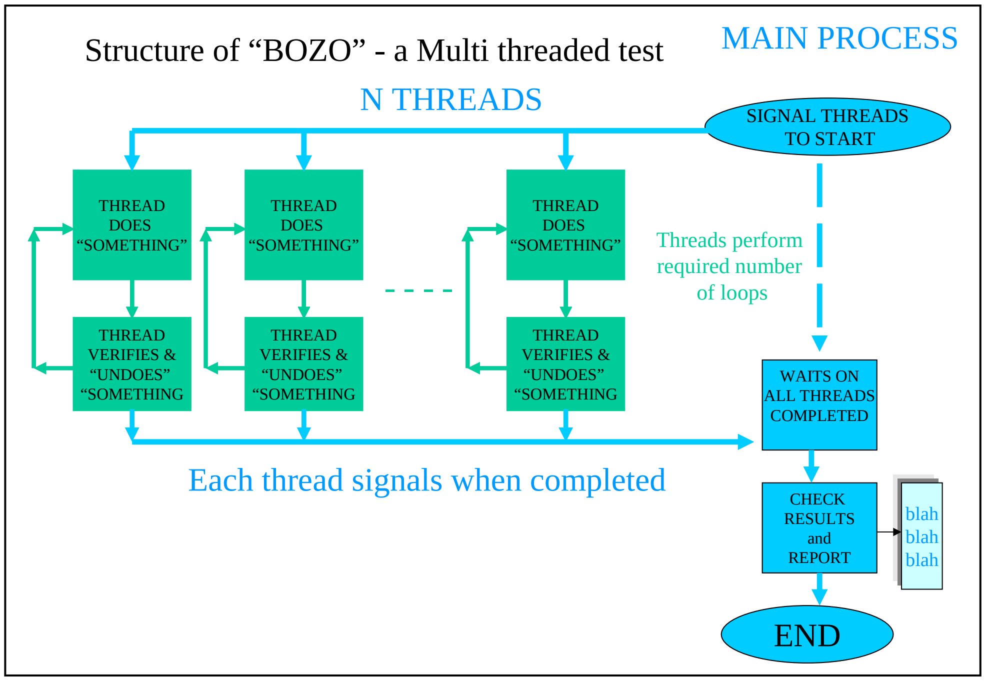

Structure of "BOZO" - a Multi threaded test
MAIN PROCESS
N THREADS
SIGNAL THREADS TO START
THREAD DOES "SOMETHING"
THREAD DOES "SOMETHING"
THREAD DOES "SOMETHING"
Threads perform required number of loops
THREAD VERIFIES & "UNDOES" "SOMETHING
THREAD VERIFIES & "UNDOES" "SOMETHING
THREAD VERIFIES & "UNDOES" "SOMETHING
WAITS ON ALL THREADS COMPLETED
Each thread signals when completed
CHECK RESULTS and REPORT
blah blah blah
END

Figure 2 - Structure of a Multithreaded Test

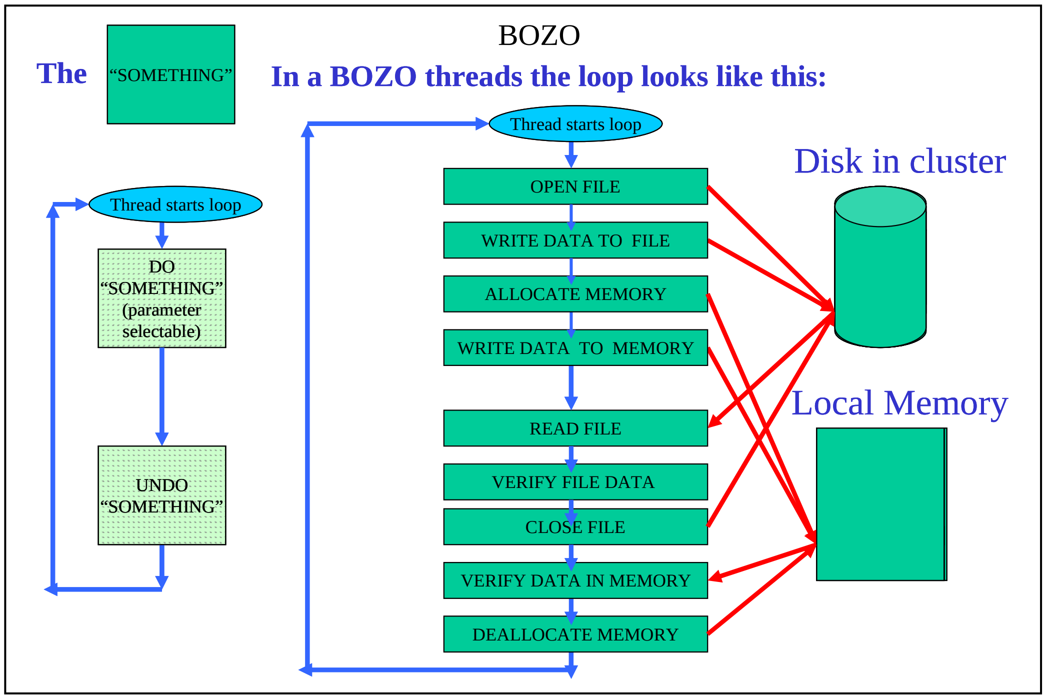


Figure 3 – The Loop

To permit the maximum utilization of the system, the SYNCH parameter specifies how many times the client thread should repeat the operations of Phase I and Phase II before it reports to the main thread that has completed.

The test parameters can specify any permutation of memory task to be performed (such as I/O only, memory only, I/O and memory, or neither I/O nor memory). The case where neither I/O nor memory operations are performed effectively tests the multithreading runtime library by causing a very large number of thread creates, joins, and terminations.

CTM_YOYO

CTM_YOYO is a threaded tool that was designed to stress threaded sequences and, in particular, to stress the thread creation, thread joins, thread termination, and upcall functionality.

The basic architecture of the test is the same as described under the General Methodology section above, except that the main thread, instead of just controlling a vector of client threads, actually controls a matrix of client threads. The main thread creates and synchronizes with the top level thread of what can be visualized as a series of columns of threads. The test uses recursive logic to test thread sequences. The top-level thread causes a series of threads to be created recursively underneath it. The number of columns of threads and the depth of recursion are specified by parameters.

As with the other threaded tests, the main thread communicates with the test harness, logs performance information, initializes data buffer and patterns, creates client threads, and controls them by gang scheduling. For speed and efficiency, the main thread creates all the data patterns for each of the client threads and the threads that they create, before the test starts.

The basic test unit in CTM_YOYO is a thread, and other than the top level thread (which communicates with the harness and reports back the results of all the threads created underneath it) and the lowest level thread which does not create a thread underneath it, the threads in CTM_YOYO operate as follows:

1. Thread starts a task that involves allocating resources.
2. Thread then recursively calls down to a copy of itself.
3. Thread waits for the completion of the recursive call.
4. .....

5. Recursive call completes.
6. Thread finishes the task it was doing.
7. Thread checks the results of what it did.
8. Thread releases the resources it was using.
9. Thread completes.

The name CTM_YOYO derives from the fact that the threads “yoyo” up and down through recursive calls. This is a very stressful on the mechanisms that are used to schedule the threads. It tests the up-calls and causes rapid context switching on the processors on which the threads are executing.

As mentioned above, a CTM_YOYO test is characterized by “WIDTH” and “DEPTH,” which are test parameters. The depth is the level that the threads go down to recursively. The width is the number of sequences of threads that are simultaneously yoyoing up and down. Effectively, a matrix of thread execution sequences is involved in a test, each characterized by (X,Y), where X is the width position of the thread within the matrix, and Y is the depth position of the thread within the matrix. The matrix of threads is controlled by a main program, which interfaces to CTM, schedules and synchronizes the threads within the matrix, tests the results they report back, and does the TEL logging of performance information.

Again, the tasks that the thread sequences perform are related to allocating and testing memory regions, and to simple I/O operations, and again any permutation of memory task and I/O task may be specified (such as I/O only, memory only, I/O and memory, or neither I/O nor memory). The case where no memory testing and no I/O testing is specified is not only valid but particularly useful for this test as it causes rapid and stressful sequence of thread creation, termination, and synchronization to occur.

Specifically, if memory is to be tested by CTM_YOYO then each thread will do the following:-

1. Allocate a memory region of the specified number of pages (8192 bytes).
2. Write a data pattern into this memory region.
3. Call down recursively and wait for completion on the recursive call.
4. .....
5. Read back and verify the data pattern from the memory region.
6. De-allocate the memory region.

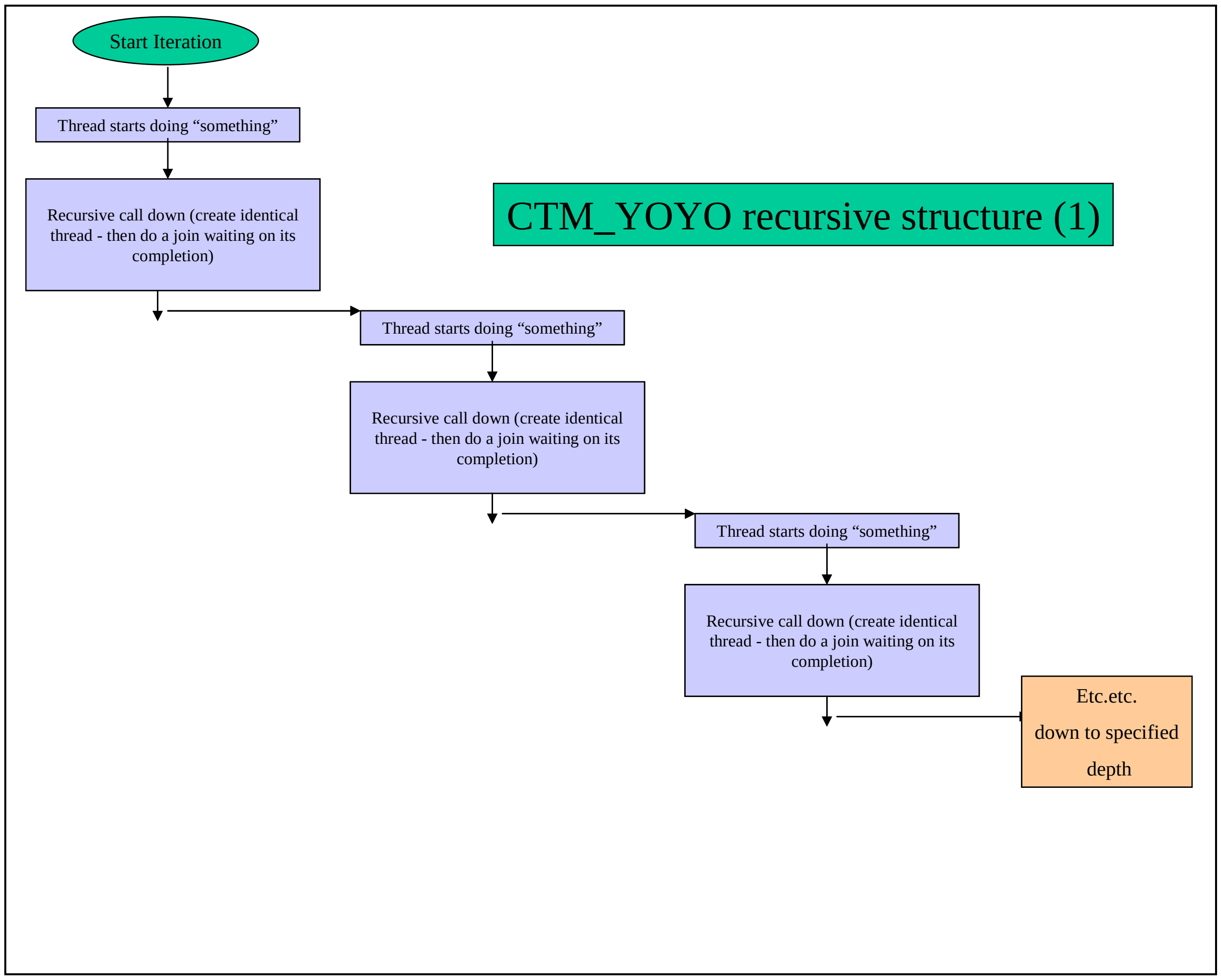

Start Iteration
Thread starts doing "something"
Recursive call down (create identical thread - then do a join waiting on its completion)
CTM_YOYO recursive structure (1)
Thread starts doing "something"
Recursive call down (create identical thread - then do a join waiting on its completion)
Thread starts doing "something"
Recursive call down (create identical thread - then do a join waiting on its completion)
Etc.etc. down to specified depth

Figure 4 – The Recursive Structure (1)

If I/O is to be tested by CTM_YOYO each thread will do the following:-

1. Create and open a data file.
2. Write a data pattern of the specified number of blocks (512 bytes) to this data file.
3. Call down recursively and wait for completion on the recursive call.
4. .....
5. Read back and verify the data pattern from the data file.
6. Delete the data file.

For a large test, there may be 16 or more columns of threads, each recursing down to a depth of 16, which potentially generates a huge amount of data. Tracing, logging, and generally interpreting this amount of data when a problem is detected poses quite a challenge. When a thread detects a problem or a corruption, it generates its own corruption report, numbered and identified according to its position in the matrix, before it terminates. In this case, it does not call down to the client thread below it but sets a status to indicate to the thread that created it that it failed. All failures are bubbled up to the top level thread, which reports the failure to the main thread. The main thread puts a summary of the failure into the test log and then terminates the test.

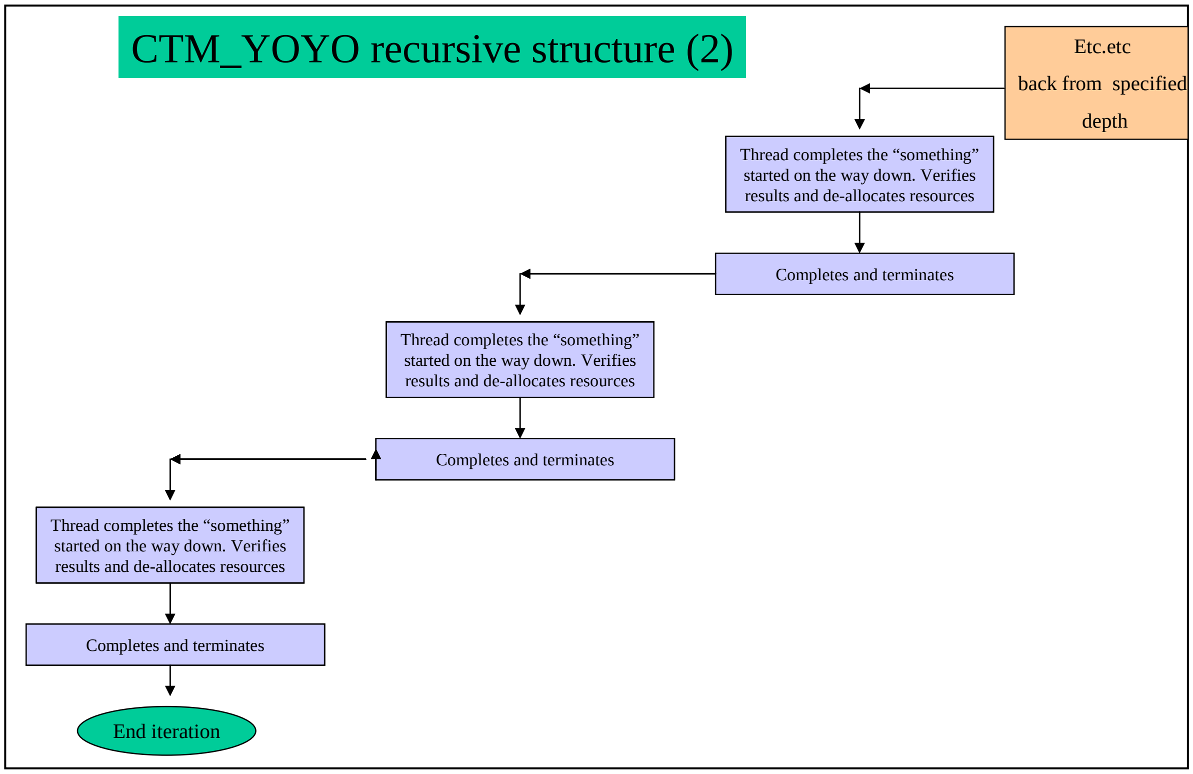

CTM_YOYO recursive structure (2)
Etc.etc back from specified depth
Thread completes the "something" started on the way down. Verifies results and de-allocates resources
Completes and terminates
Thread completes the "something" started on the way down. Verifies results and de-allocates resources
Completes and terminates
Thread completes the "something" started on the way down. Verifies results and de-allocates resources
Completes and terminates
End iteration

Figure 5 –The Recursive Structure (2)

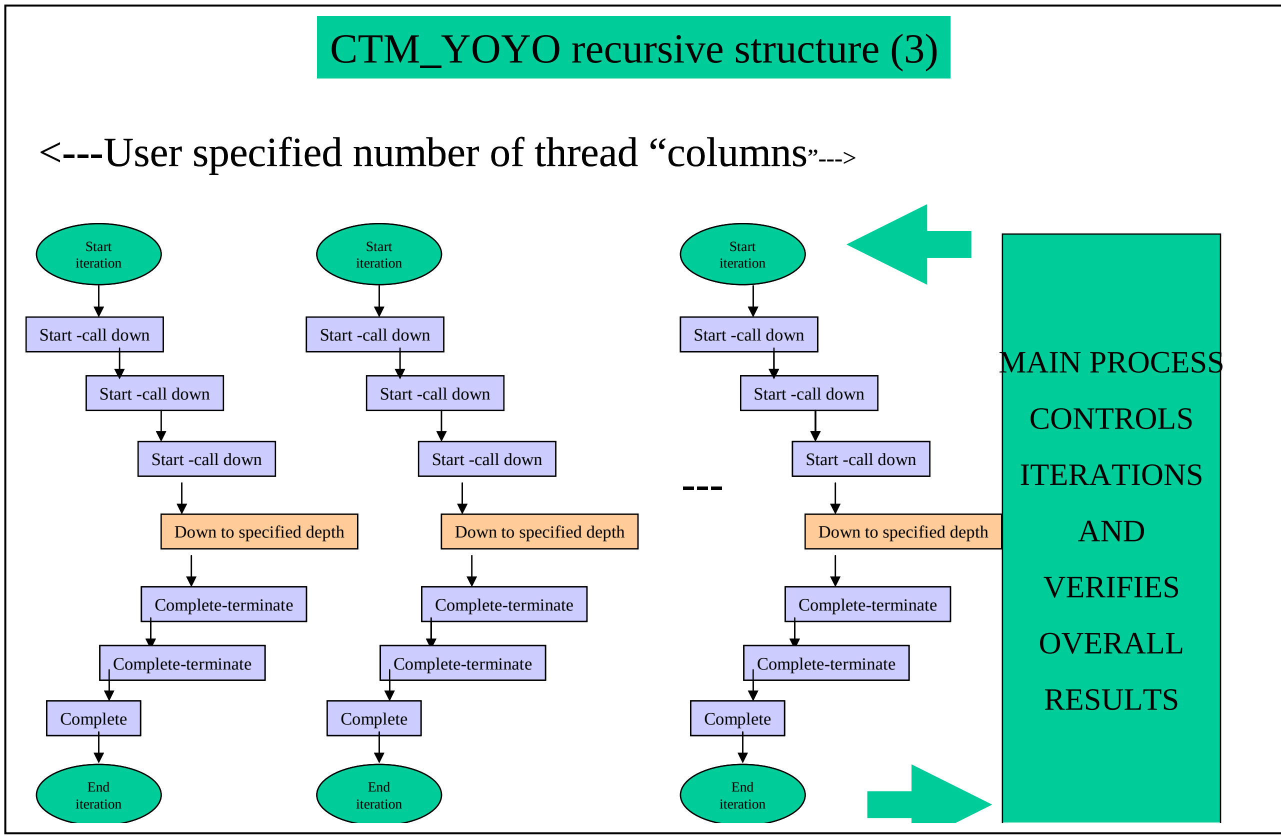


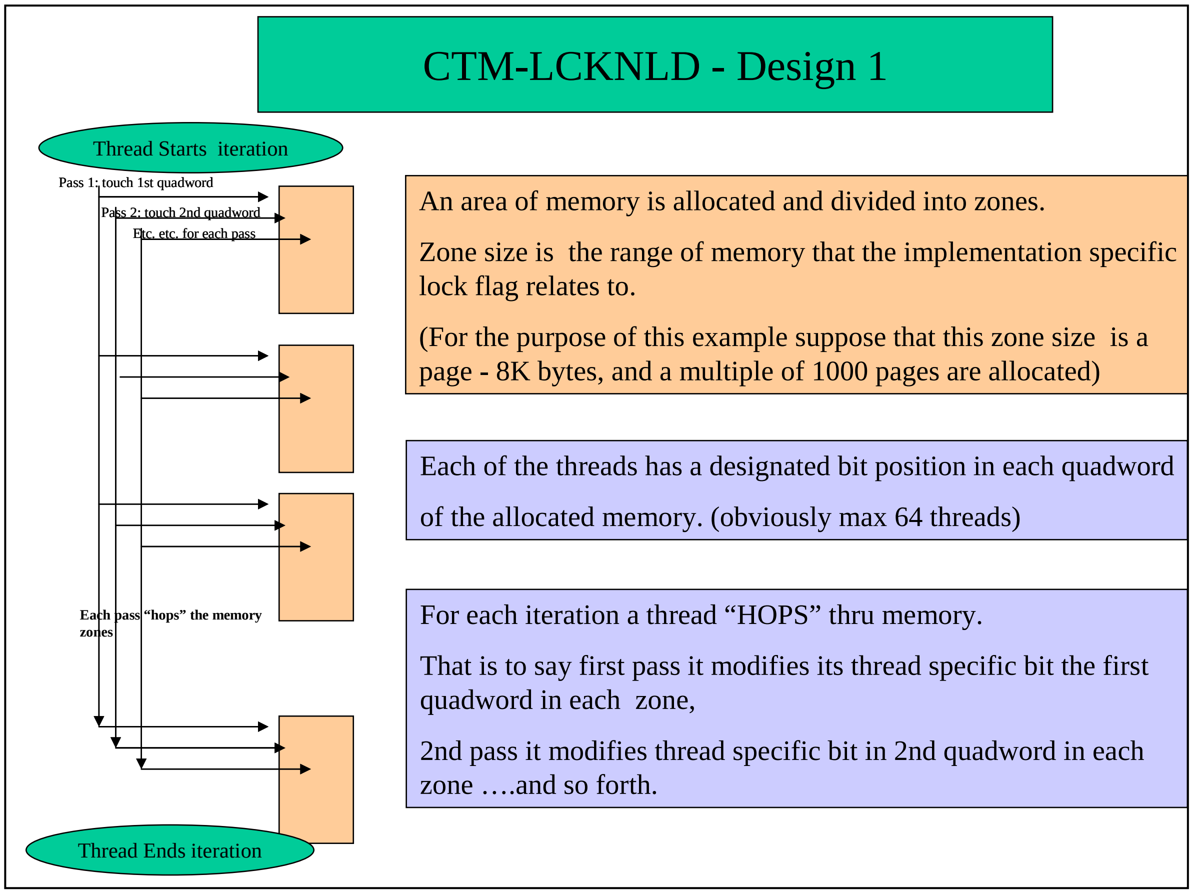
Figure 6 – The Recursive Structure (3)

CTM_LKNLD

The CTM_LKNLD test was specifically designed to test the load_lock/store_conditional sequence on multiprocessor Alpha systems. The test is designed for that Alpha architectures. This sequence was chosen for testing because of its importance for maintaining memory coherency on a multiprocessor system, and because rigorous testing in this area by implication tests other mechanisms that are involved in maintaining coherency on Alpha architectures.

To provide some context, a processor modifies the contents of physical memory by means of a LOAD/modify/STORE sequence. However in an SMP environment, where more than one processor has access to the memory location, this simple sequence will not suffice. For example, if there are two processors (A and B,) if A LOADS the memory location B LOADS the memory location, then B modifies the memory location and does a STORE, then A modifies the memory location and does a STORE, B's view of memory is now incorrect and something is going to get broken. The load locked/store conditional sequence provides a method whereby multiple CPUs can synchronize access.

The load locked/store conditional logic is the same for whatever granularity of memory is being accessed. CTM_LKNLD is based on quad word access (LDQ_L/STQ_C sequences). Roughly speaking, when a processor loads the memory with an LDQ_L instruction, it records the target address in a per-processor locked physical address register and sets a per-processor lock flag. If the per-processor lock flag is still set when it gets around to doing the store with a STQ_C instruction, then the store occurs; otherwise, the store does not occur. Whenever a processor successfully completes a store to within the locked address range of another processor, it clears the lock flag for that processor, causing that processor's subsequent store conditional to fail.


Thread Starts iteration
Pass 1: touch 1st quadword
Pass 2: touch 2nd quadword
Etc. etc. for each pass
Each pass "hops" the memory zones
Thread Ends iteration

CTM-LCKNLD - Design 1

An area of memory is allocated and divided into zones.

Zone size is the range of memory that the implementation specific lock flag relates to.

(For the purpose of this example suppose that this zone size is a page - 8K bytes, and a multiple of 1000 pages are allocated)

Each of the threads has a designated bit position in each quadword of the allocated memory. (obviously max 64 threads)

For each iteration a thread "HOPS" thru memory.

That is to say first pass it modifies its thread specific bit the first quadword in each zone,

2nd pass it modifies thread specific bit in 2nd quadword in each zone ….and so forth.

Figure 7 – CTM_LKNLD (1)

To use the example of processors A and B above:-

1. A does LDQ_L on address pa succeeds
2. B does LDQ_L on address pa succeeds.
3. A does STQ_C on address pa succeeds - clears B's lock flag.
4. B does STQ_C on address pa fails because B's lock flag is cleared.
5. B should now re-load pa and retry the sequence.

The basis of the test performed by CTM_LKNLD is similar to that used by CTM_BYTEM. The basic architecture is the same, with a main thread controlling a number of client threads by gang scheduling them through Phase 1 and Phase II. The result is a user-specified number of client threads concurrently looping through the same regions of memory. Each thread tries to modify a byte in a quad word with a data pattern specific to the thread. The threads access the quad words using LDQ_L/STQ_C sequences to access and store the modified quad words.

The data pattern that a thread writes to a byte within the quad word is the ones complement of the thread number. Hence, thread 0 writes binary 11111111, thread 1 writes binary 11111110, and so forth. All the memory regions are set to zero before a test iteration, so if a value of 0 is subsequently detected, it indicates that a thread did not write the pattern correctly.

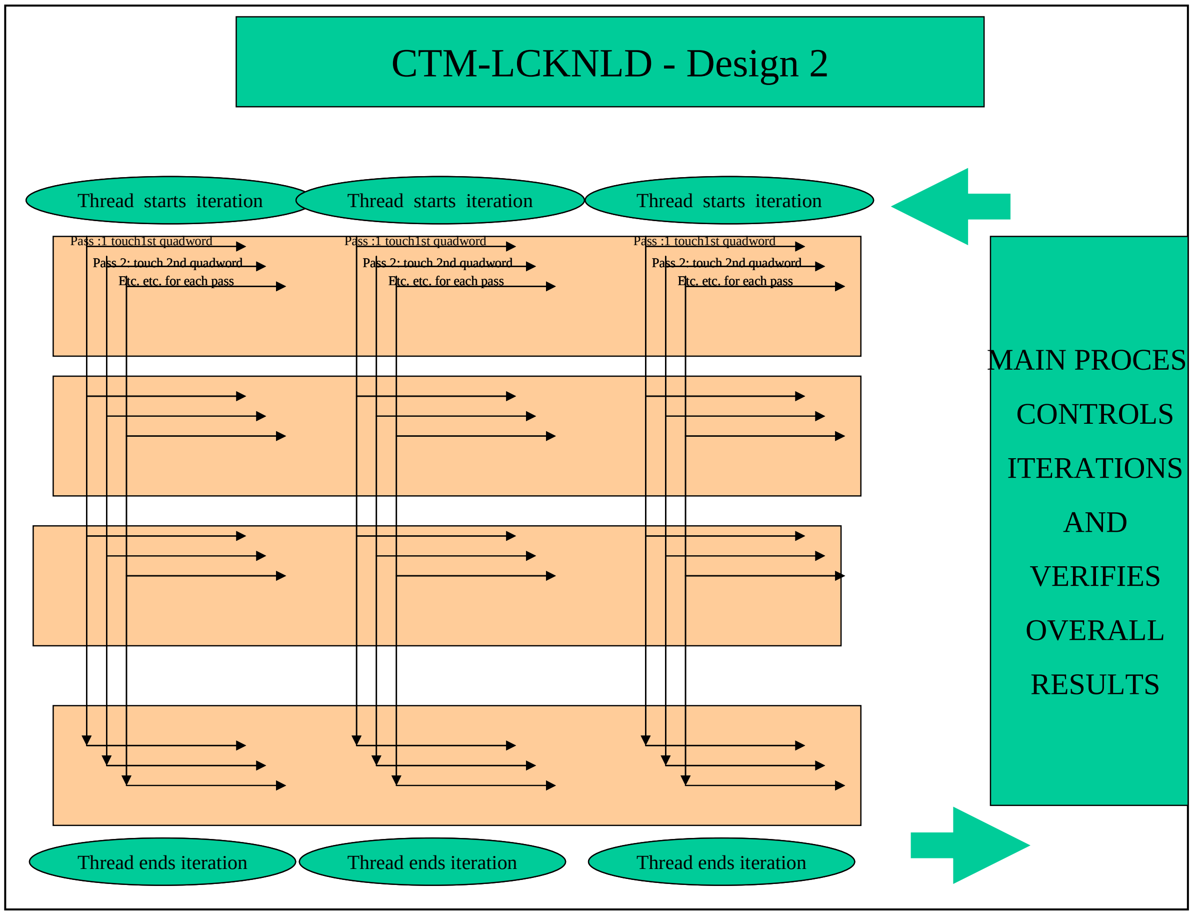


Figure 8 – CTM-LCKNLD (2)

Of course, the limits imposed by OpenVMS and the multithreaded runtime library make it impossible to control which thread will run on which CPU directly; however, it is a statistical certainty that there will be many times when different threads are simultaneously trying to access bytes in the same quad word, hence the LDQ_L /STQ_C sequence has to work correctly for the data patterns to be set correctly. By deliberately building the test to use LDQ_L/STQ_C sequences instead of regular Load and Store sequence, the test verifies that the number of reported corruptions corresponds to the approximate number of times two client threads try to access the same quad word. In the case of this test, which is significantly more complex, this technique was invaluable for ensuring the test was doing what was intended.

For historical reasons, the CTM_LKNLD test was extended beyond the basic test described above, by varying the conditions when the STQ_C occurs. For each thread a varying number of memory accesses are interposed between the LDQ_L and the STQ_C, so that when the STQ_C occurs, the test causes the TB (Translation Buffer) entry to correspond to the target address in different positions in the TB. This is achieved by having the threads hop through memory as follows:

1. First memory location in first memory region
2. First memory location in second memory region
3. ...
4. First memory location in last memory region
5. Second memory location in first memory region
6. Second memory location in first memory region
7. ...
8. Second memory location in last memory region
9. ...
10. ...
11. Last memory location in last memory region

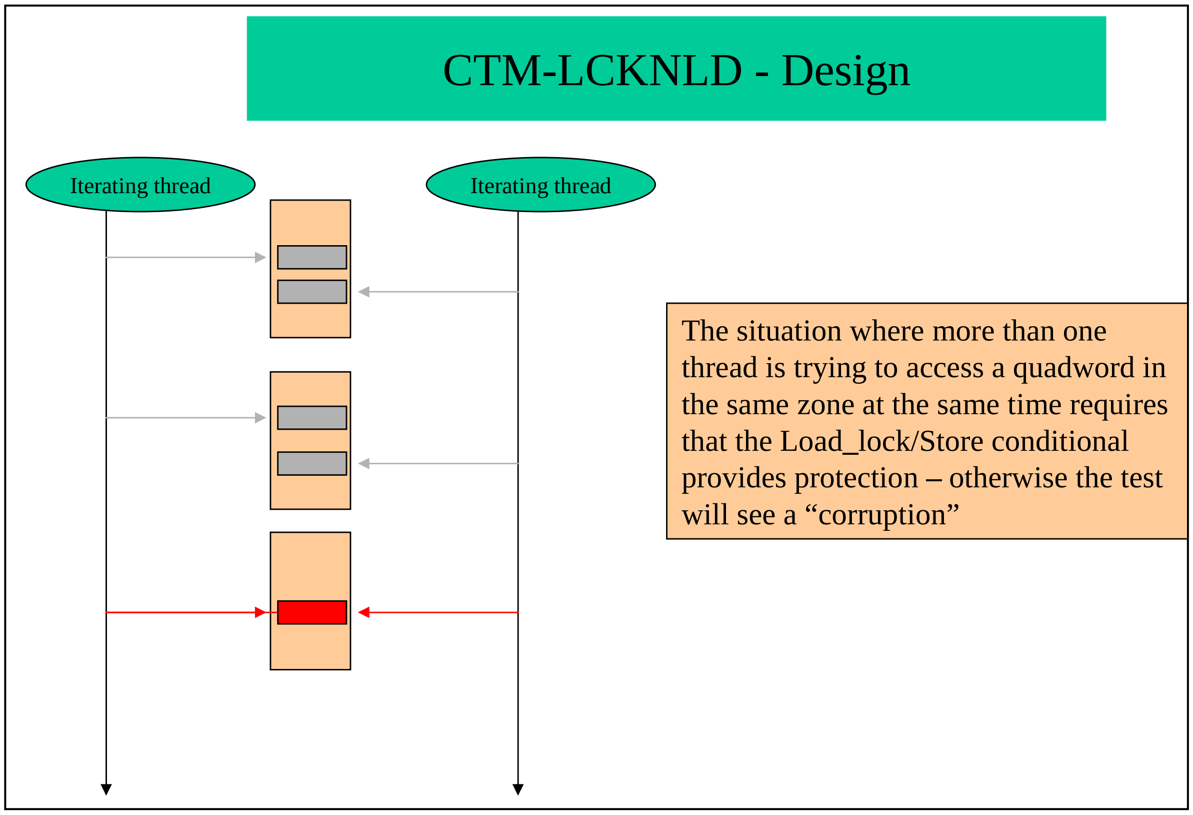

CTM-LCKNLD - Design
Iterating thread
Iterating thread
The situation where more than one thread is trying to access a quadword in the same zone at the same time requires that the Load_lock/Store conditional provides protection – otherwise the test will see a "corruption"

Figure 9 – CTM LCKNLD (3)

Two sets of memory regions are accessed alternately by the threads. For convenience, these are referred to as U regions and V regions. There are 128 U regions and the same number of V regions. Each memory region is the system-specific page size (currently set to 8192 bytes).

Each hop accesses a new page and results in a new entry in the TB. Having initiated an LDQ_L/STQ_C sequence with an LDQ_L, the thread then is then forced to access a varying number of memory locations in other memory regions before it completes the STQ_C. These are accessed with regular LDQ instructions; we consider them "noise," interposed between the LDQ_L and the STQ_C.

Because these noise memory accesses are to different pages, they have the effect of causing a new entry into the TB, and hence cause the TB entry for the target to be in a varying position in the TB when the STQ_C occurs. The number of noise accesses that are forced in this way varies from 0 to 128 entries plus one, with each iteration, so that eventually the thread completes the STQ_C with the target address entry in a variety of different positions within the TB.

On alternate iterations, each thread alternates as follows:

1. Uses U regions for LDQ_L/STQ_C (target) access.
2. Uses V regions for LDQ (noise) access to vary the position of the focus access in the TB.
3. Uses V regions for LDQ_L/STQ_C (target) access.
4. Uses U regions for LDQ (noise) access to vary the position of the focus access in the TB.

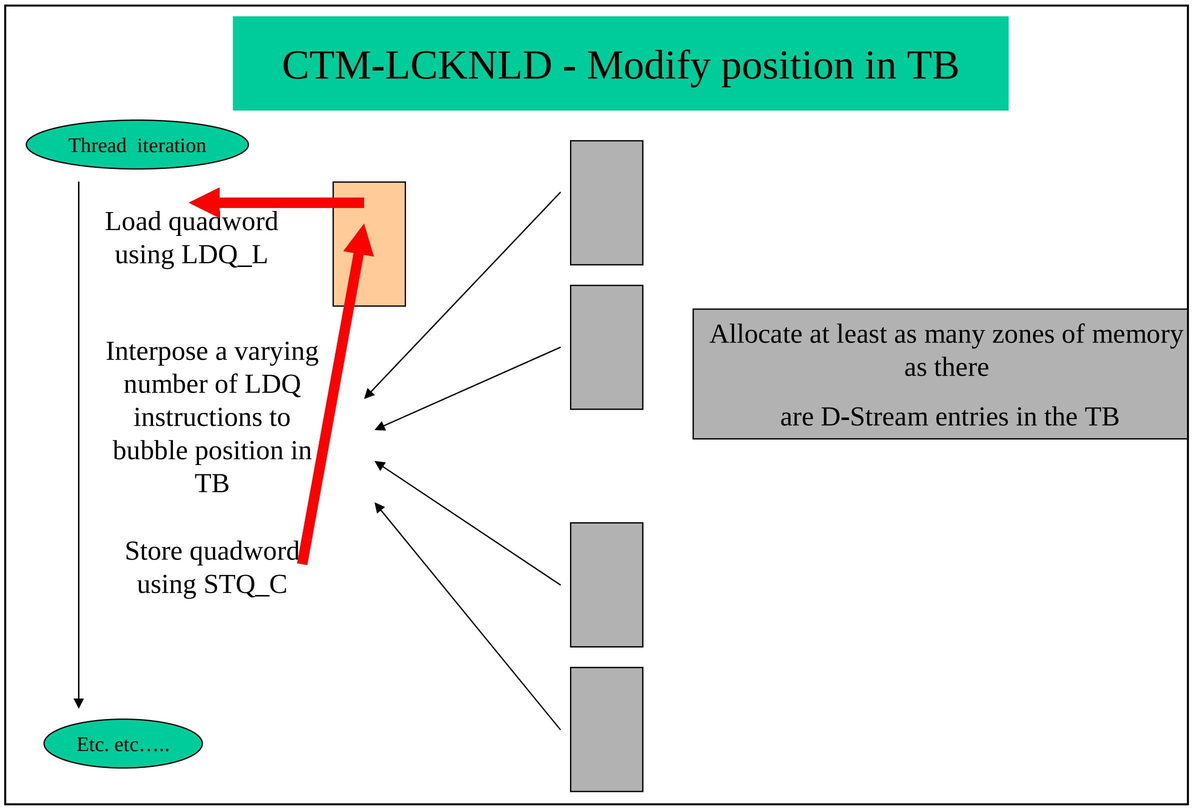

CTM-LCKNLD - Modify position in TB
Thread iteration
Load quadword using LDQ_L
Interpose a varying number of LDQ instructions to bubble position in TB
Store quadword using STQ_C
Etc. etc…..
Allocate at least as many zones of memory as there are D-Stream entries in the TB

Figure 10 – CTM_LCKNLD (4)

CTM_FAST_IO

CTM_FAST_IO was written to test FAST_IO functionality, a new feature in OpenVMS Version 7.0 aimed at dramatically improving I/O performance.

Fast I/O is for systems and applications that require highly optimized paths for high volume I/O, such as large database systems. Fast I/O comprises a set of system services that provide an alternative to QIOs. The greatest benefit of Fast I/O is that it locks and maps user buffers and user I/O status areas permanently in system space. For direct I/O, this avoids per-I/O buffer lockdown and unlocking; for buffered I/O, it avoids allocation and deallocation of separate system buffers, permitting complete Fast I/O operations at IPL 8 and reducing spinlock acquisitions.

Threads are not a particularly important aspect of the FAST_IO test, other than providing the capability for improved performance on multiprocessor systems. A description of the CTM_FAST_IO test is provided here for completeness.

As with the other CTM threaded tests, the CTM_FAST_IO test comprises a main thread, which controls a user-specified number of client threads Two write buffers are used by each client thread: a pattern0 buffer and a pattern1 buffer. In addition to all the usual work, the main thread presets data patterns into the write buffers. Data pattern1 is the ones complement of the pattern0 buffer. Pattern0 and Pattern1 are alternately written to and read from a data file by the threads, using Fast I/O operations. Because the patterns are complements, the data integrity of the I/O operations can be checked without the need to poison buffers between reads and writes. The main process allocates read and write buffers for each of the threads. The size of these data buffers is the size of the actual data files that will be used by the threads, and is determined by the BLOCKS parameter (expressed in 512-byte blocks). Prior to starting the threads, the main process locks down all the read and write buffers that the threads will use in buffer objects, using the SYC$CREATE_BUFOBJ_64 system service call. The buffer objects remain locked down for use by the threads for the entire duration of the test.

Having prepared and locked down the buffers that the threads will use, the main process then creates and synchronizes the client threads through the actual test, which conforms to the Phase I/Phase II structure of all the CTM threaded tests.

A thread performs each test iteration as follows.

In Phase I, the thread:

1. Deletes any previous versions of its data file.
2. Creates and opens a new data file using RMS.
3. Performs the setup for each of the subsequent file operations using the SYS$IO_SETUP system service.
4. Loops a user-specified number of times (specified in the SYNCH parameter), by doing the following Fast I/O operations on the data file.
5. Writes pattern0 data to the data file using the SYS$IO_PERFORMW system service.
6. Reads the data back from the data file using the SYS$IO_PERFORMW system service.
7. Verifies the data read back against the data written.

In Phase II, the thread:

1. Writes pattern1 data to the data file using the SYS$IO_PERFORMW system service.
2. Reads the data back from the data file using the SYS$IO_PERFORMW system service.
3. Verifies the data read back against the data written.
4. Closes its data file, and cleans up the Fast I/O operations using the SYS$IO_CLEANUP system service.
5. Reports the results (success or failure of the data verification) back to the main process, and waits to be started again by the main process.

Conclusion

This article presents only one basic design for threaded tests, namely multiple client threads gang scheduled by a main thread, with the actual test functionality performed by the client threads and formally divided into two separate phases (usually the test operation phase and the verification phase). Of course, such tests can be organized in many ways, and there is no intent to suggest this is the best way to do it. However, this approach has worked well for us in terms of reducing the effort required to write such tests and so far has not presented any limitations for the tests we have needed to write. To be of use, tests for the operating system need to be very solid and reliable, and the results of the testing must be readily understandable so that problems can be debugged. Writing tests inside this framework has definitely helped us meet these objectives.

Threaded tests provide an invaluable means for testing on multiprocessor systems, not only in so far as they improve the performance of the test and permit more work to be done in a given time, but, more importantly, they provide the means to directly test that the required functionality works properly when functionality is distributed across multiple threads running concurrently on separate processors. Properly constructed, such tests also provide the means to subject the systems under test to extreme stress and to manipulate the systems so that situations that could potentially cause problems are created much more frequently than would otherwise be the case. There is no doubt that such tests will become increasingly important in ensuring that the very high level of reliability that OpenVMS is famous for will continue into the future.

OpenVMS Technical Journal V6

Porting TDMS - The Gory Details

Porting TDMS - The Gory Details

Bob Sampson, Consulting Associate

Overview

Porting any application from OpenVMS VAX; first to OpenVMS Alpha, and then to OpenVMS I64 is a challenging project. As we ported TDMS, we encountered four tricky details that we have described here to help you in your own porting efforts.

Origin of TDMS Software

VAX TDMS is an older generation character-cell terminal forms package for transaction processing environments. It is a mature product that is available for OpenVMS VAX only.

The TDMS (Terminal Data Management System) for OpenVMS VAX software product (VAX TDMS) was designed for the implementation of interactive, forms-intensive applications.

Long ago, VAX TDMS was officially superseded by DECforms, which is recommended for any and all new character-cell terminal applications that require a forms-based user interface on OpenVMS.

The VAX TDMS Software Product Description (SPD) can be viewed at: http://h18000.www1.hp.com/info/SP2571/sp2571pf.pdf. Please note that any support or availability commitments in that document (whether expressed or implied) do not apply to ported TDMS software.

As a terminal subsystem, TDMS can reduce the application development and maintenance effort by replacing application program logic specific to terminal interactions with definitions that are external to the program.

Ported TDMS Software Services

My team offers ported TDMS software, in the form of licensing and support services, for the purpose of providing a simple migration path for TDMS applications (especially those running under the ACMS environment) from OpenVMS VAX to OpenVMS Alpha (and now to OpenVMS I64 as well).

The TDMS software was ported to OpenVMS Alpha in late 2001, and field tested in mid-2002. The TDMS software was further ported to OpenVMS I64 in late 2004.

As a part of the original Alpha TDMS porting effort, the hardcopy VAX TDMS documents (Forms Manual, Installation Guide, Pocket Guide, Reference Manual, and Request & Programming Manual) were converted to HTML, and are provided with every Alpha TDMS and I64 TDMS kit, along with the current release notes.

An evaluation license, a software kit, and limited remedial support are available to customers for sixty days at no charge.

Alpha TDMS licensing and support services have been offered for sale worldwide since early 2003. As of this writing, I64 TDMS is available only for evaluation.

Further information about the ported TDMS software (Alpha TDMS and I64 TDMS) is available at: http://h71000.www7.hp.com/commercial/tdms/tdms_services.html.

Porting Challenges

Four major porting challenges were encountered. Architectural differences among OpenVMS VAX, Alpha, and I64 were the leading cause, followed by limitations of the Bliss-32 language dialect. Mistakes were made on several occasions, introducing both subtle and not-so-subtle bugs, not all of which were immediately detected during regression testing.

- Differences in the calling standards were especially problematic, and accounted for two of the four challenges; see the first and third items below.

Using a transfer vector and fetching data from a known stack layout for VAX had to be replaced by using a bound procedure descriptor and specially-written Macro-64 transfer routine for Alpha, and by using a bound function descriptor, a new OpenVMS-provided transfer routine, and special linkage for I64.

An internal context-switching package had to be changed out to match the one used by ACMS on each of the architectures. On VAX and Alpha, this package had to be implemented in low-level assembly language, using detailed knowledge of the respective calling standards and stack layouts. On I64, the implementation was greatly simplified by making use of the new OpenVMS-provided user-mode KPS routines.

- Differences in the object file formats accounted for another of the four challenges; see the second item below.

The Alpha object file format is somewhat different from, yet almost completely analogous to, the VAX object file format. The I64 ELF object file format is driven by a cross-platform industry standard, so it is very different from its predecessors. However, the addition of OpenVMS-specific extensions makes the new format completely suitable for building ported software.

- Differences in 64-bit integer math implementations, combined with a lack of Bliss-32 language support for these multiply and divide operations, accounted for the last challenge; see the fourth item below.

The solution ultimately chosen was to provide these operations using routines written in a language that fully supports them; in this case, HP C.

1. Context Passing: Application Function Key (AFK) Asynchronous System Traps (ASTs)

VAX TDMS implements context passing for AFK ASTs by providing a transfer vector consisting of an entry register save mask and a jump instruction with address fix-up at the start of each context data structure. The AST routine then fetches the context from a known relative offset on the stack.

Over several months in late 2001, with some guidance from OpenVMS Engineering and the Calling Standard documentation, this context passing mechanism was ported to Alpha TDMS. The VAX transfer vector was replaced with an Alpha bound procedure descriptor at the start of each context data structure. A short Macro-64 transfer routine provides the context address (bound procedure descriptor address from R27) to the AST routine as the sixth actual argument in R21. The argument count (low byte of AI/R25) is also filled in with the value six. Names and attributes of code program sections are specified so that the linker places the transfer routine code immediately before the AST code, eliminating two instructions (load and jump). This results in an efficient implementation of AFK ASTs that is compatible with the Alpha calling standard. Please refer to section 2.2.2 of the release notes for further details.

Over about ten days in early 2003, with further guidance from OpenVMS Engineering, a similar mechanism was used for I64 TDMS, which was actually simpler to implement. The Alpha bound procedure descriptor was replaced by the I64 bound function descriptor. A transfer routine called OTS$JUMP_TO_BPV is already provided by OpenVMS I64, specifically for use with bound function descriptors. The context is passed to the AST in R9 by specifying special linkage.

Over a few days in late 2004, regression testing uncovered a problem with the bound function descriptor, which was corrected. Please refer to section 2.6.3 of the release notes for further details.

2. Object File Generation: Form Development Utility (FDU) Form Editor (FED) Memory-Resident Forms

VAX TDMS implements memory-resident forms for the FDU form editor. A utility program called FDVCDDOBJ is used to package each TDMS form from the Common Data Dictionary (CDD) into an OpenVMS VAX object file. The linker then arranges the forms into a linked list in the FDU program.

Over about one week in May 2001, FDVCDDOBJ was easily converted to generate object files in the OpenVMS Alpha format. The VAX and Alpha object formats are very similar.

Over about one month in April and May 2003, with some guidance from OpenVMS Engineering, FDVCDDOBJ was converted to generate object files in the OpenVMS I64 (ELF) format, which is very different, and considerably more complex. Also, ELF initially did not provide support for Link-Time Complex Expressions (LTCEs), and this support was not added until after the porting work on FDVCDDOBJ was completed.

The TDMS source code now compiles correctly for any of the three recognized target platforms: VAX, ALPHA, or IA64.

3. THD Component

VAX TDMS implements a simple form of "threading" (actually more like context switching using co-routines) with its THD component, which is very similar to (and very compatible with) the ACMS product's THD component on OpenVMS VAX.

From late May through late November 2001, an attempt was made (with reasonable success) to re-implement TDMS THD using POSIX threads, a.k.a. pthreads, formerly DECthreads. This also required extensive changes for thread-awareness. Not much effort was expended to investigate or test for potential thread-safety issues.

Very early during initial field test, this implementation had to be abandoned (or at least set aside), in order to provide interoperability with the ACMS product on OpenVMS Alpha. Please refer to section 3.23 of the release notes for further details.

Fortunately, ACMS Engineering kindly provided its ACMS THD OpenVMS Alpha source code. Over about one week in November 2001, this was incorporated into Alpha TDMS.

From mid-May through mid-November 2003, possible strategies were discussed with the ACMS and OpenVMS Engineering groups, for porting the THD component to OpenVMS I64. On a few occasions, attempts were made to perform some of this porting work.

As OpenVMS Engineering had advised, ACMS Engineering ported its ACMS THD to OpenVMS I64, using the newly-enhanced OpenVMS KPS routines, and again kindly provided the source code for I64 TDMS THD. Very little low-level machine language was required for this new implementation, aside from the OpenVMS KPS routines themselves.

This improves on the VAX and Alpha THD implementations, which require much more of their own Macro-32 and Macro-64 source code.

The I64 THD also appears to be highly portable to OpenVMS Alpha V8.2 or higher. In the event of any compelling need, it could most likely replace the existing Alpha THD.

Within one day in June 2004, the ACMS I64 THD was easily "dropped" into I64 TDMS.

4. 64-Bit Integer Math

The VAX TDMS source code was written almost entirely in Bliss-32, a language dialect that has no direct support for multiply and divide operations on 64-bit integers. OpenVMS date conversions make use of 64-bit integers. VAX TDMS performs divide and multiply operations on these by calling the undocumented COB$DIVQ_R8 and COB$MULQ_R8 COBOL language support run-time library routines with special JSB linkage.

These routines are not available for the use of native OpenVMS Alpha applications. The initial attempt during April 2001 to replace them with in-line Bliss-32 code did not work properly. Please refer to section 2.2.10 of the release notes for further details.

One day in January 2002, the DIVQ and MULQ replacement routines were implemented in C for Alpha, but unfortunately with coding errors that went undiscovered for about a year, even during intensive field testing.

One day in January 2003, in response to a problem report from a very alert HP employee and OpenVMS Ambassador (now an alumnus) in Ireland, these coding errors were finally corrected, along with two other software problems that had interfered with some of the date conversions. Please see section 2.2.14 of the release notes for further details.

Porting to I64 did not require any further changes to these routines.

Summary

We hope that that lessons we learned from porting TDMS software will save others some time and effort in the future.

For more information

Contact the author by e-mail: Robert.Sampson@hp.com.

OpenVMS Technical Journal V6

Protecting and Monitoring OpenVMS Systems

Protecting and Monitoring OpenVMS Systems

Michael Grinnell – Off-site Software Support Engineer

Gina Jones – Off-site Software Support Engineer

Overview

In the work environments of today, the risk of computer breaches from outside corporate or private firewalls are a constant threat. However, the threat to OpenVMS system security can also come from inside of these same firewalls - threats that can allow unwanted access to OpenVMS systems. Even though OpenVMS was named "Cool and Virtually Unhackable" at the 2001 DEFcon9 convention, lax security methods and irresponsible users can leave OpenVMS systems to become vulnerable to unauthorized access.

Fortunately, system administrators can enhance OpenVMS security by monitoring the integrity of critical system files, applications, and utilities without incurring costs for 3rd party products.

This article discusses two mechanisms that OpenVMS provides which can help administrators monitor file and utility access and prevent unauthorized access. OpenVMS Auditing and Access Control Entries (ACEs) are standard with OpenVMS installations. Using these utilities, system administrators can monitor and secure critical system files and resources, operations that are vital to maintaining a secure OpenVMS operating system.

User Access Security Issues in OpenVMS Environments

On computer systems there are basically two types of users: authorized and unauthorized. Authorized persons are given certain access to perform their jobs/tasks and with that access a certain amount of responsibility. On the other hand, unauthorized users are not intentionally given access to system resources and in general take no responsibility for their actions while accessing the system. Security breaches usually result from one of four types of user actions:

- User irresponsibility refers to situations where a user purposely or accidentally causes noticeable damage. A user accessing or copying a file or key to sell is an example of this type of irresponsibility.
- User probing refers to users who are authorized to access computer resources and exploit insufficiently protected parts of the system. Although some user's intentions may not be malicious in nature, theft of services is a crime. Users with more serious intent may seek confidential information, attempt embezzlement, or even destroy data by probing.
- User penetration refers to a situation where a user breaches the security controls to gain access to a system. Although OpenVMS is virtually "hack proof," inadequate security procedures can allow this most serious type of breach.
- Social engineering refers to intruders gaining access by deceiving legitimate users of the system. They may impersonate users by gaining access to their unsecured system/terminal passwords or by having a user perform actions that may compromise the security of the system.

Using Auditing to Monitor Access to Critical Files

Auditing can be described as recording security-relevant activities as they occur on the system and the review/analysis of the auditing log containing those activities.
Both successful and unsuccessful events can be recorded in the audit log file, SYS$COMMON:[SYSMGR]SECURITY.AUDIT$JOURNAL. In addition, a terminal can be designated as a security operator terminal where those same events can be displayed. Often, unsuccessful events are more useful in revealing possible security concerns than monitoring successful access.

The operating system audits the following security events by default and displays those events to the SECURITY.AUDIT$JOURNAL file or to an operator enabled terminal:

- Authorization database changes
- Intrusion attempts
- Login failures
- Use of DCL command SET AUDIT
- Events triggered by Audit or Alarm ACEs

Example of Audit Settings
The following command displays the security events currently enabled on an OpenVMS system.

```
$ show audit
System security alarms currently enabled for:
  ACL
  Authorization
  Audit:        illformed
  Breakin:      dialup,local,remote,network,detached
  Logfailure:   batch,dialup,local,remote,network,subprocess,detached

System security audits currently enabled for:
  ACL
  Authorization
  Audit:        illformed
  Breakin:      dialup,local,remote,network,detached
  Logfailure:   batch,dialup,local,remote,network,subprocess,detached
```

Additional events can be selected for auditing, such as volume mounts, elevated privilege use, and successfully logins, by using the SET AUDIT command.

Example of SET AUDIT Command
The SET AUDIT command enables auditing to capture unsuccessful and successful login information in audit file and on an OPCOM enabled terminal:

```
$ set audit/audit/alarm/enable=login=all
$ show audit
System security alarms currently enabled for:
  ACL
  Authorization
  Audit:        illformed
  Breakin:      dialup,local,remote,network,detached
  Login:        batch,dialup,local,remote,network,subprocess,detached
  Logfailure:   batch,dialup,local,remote,network,subprocess,detached

System security audits currently enabled for:
  ACL
  Authorization
  Audit:        illformed
  Breakin:      dialup,local,remote,network,detached
  Login:        batch,dialup,local,remote,network,subprocess,detached
  Logfailure:   batch,dialup,local,remote,network,subprocess,detached
```

Sample Auditing Outputs of Logins and Logfailures

Example of Unsuccessful Login

```
%%%%%%%%%%%  OPCOM   8-SEP-2004 12:38:16.07  %%%%%%%%%%%
Message from user AUDIT$SERVER on TIGRES
```

```
Security alarm (SECURITY) and security audit (SECURITY) on TIGRES, system
id: 1025
Auditable event:      Remote interactive login failure
Event time:           8-SEP-2004 12:38:16.06
PID:                  000000E0
Process name:         _TNA7:
Username:             SYSTEM
Terminal name:        TNA7:, _TNA7:, Host: 192.168.1.100
Port:                 3191
Remote node id:       1677830336
Remote node fullname: 192.168.1.100
Remote username:      TELNET_C0A80164
Status:               %LOGIN-F-INVPWD, invalid password
```

Example of Successful Login

```
%%%%%%%%%%%  OPCOM   8-SEP-2004 12:39:50.52  %%%%%%%%%%%
Message from user AUDIT$SERVER on TIGRES
Security alarm (SECURITY) and security audit (SECURITY) on TIGRES, system
id: 1025
Auditable event:        Remote interactive login
Event time:             8-SEP-2004 12:39:50.52
PID:                    000000E1
Process name:           _TNA8:
Username:               SYSTEM
Process owner:          [SYSTEM]
Terminal name:          _TNA8:, Host: 192.168.1.100 Port: 3192
Image name:             TIGRES$DKA0:[SYS0.SYSCOMMON.][SYSEXE]LOGINOUT.EXE
Remote node id:         1677830336
Remote node fullname:   192.168.1.100
Remote username:        TELNET_C0A80164
```

Using Access Control Entries to Limit Access to Files

UIC (User Identification Code) protection of files is often sufficient to deter users from accessing areas of the system they should not be accessing. However, there are times when UIC protection is not adequate. To provide individual customized access protection to files and objects, Access Control Entries (ACE) may be needed.

In addition to monitoring system events, such as login failures, administrators can use an ACE to monitor individual objects, such as files and print queues.

When users attempt to access a protected object, the OpenVMS operating system compares the user profile of the user process with the security profile of the object. The sequence of those checks is as follows:

1. Evaluate the Access Control List (ACL).
If the object has an ACL (Access Control List) the system attempts to match an ACE entry from that ACL list with the user's rights identifiers. If a match is found, the user is granted or denied access and further checking of the ACL ceases.
In the following example, the user TEST holds the identifier named LARRY, which is also present on the PROJECT.TXT file. In this example, the user TEST would be allowed full access to the file:

```
$ set default sys$system
$ run authorize
UAF> show test
Username: TEST
```

```
Identifier                        Value           Attributes
  LARRY                           %X8001003C

File with Identifier:
PROJECT.TXT;1      [SYSTEM]         (RWED,RWED,,)
(IDENTIFIER=LARRY,ACCESS=READ+WRITE+EXECUTE+DELETE)
```

Note: When you provide an ACE, do not forget to deny access via UIC protection; otherwise, you are defeating the purpose of the ACE.

2. Evaluate the protection code.

If the ACE does not grant access, the operating system then evaluates the protection code. The user is granted or denied access based on the UIC protection mask. The protection mask consists of the following categories: SYSTEM, OWNER, GROUP and WORLD. Within those categories, the granted access can be RWED (READ, WRITE, EXECUTE and DELETE) or any combination.

In the following example note the access on the following file:

```
LOGIN.COM;4          [SMITH]                         (RWED,RWED,RE,)
```

The SMITH account is the owner of the file. SYSTEM has RWED. The Owner of the file is SMITH, who also has RWED. The group has RE. The world has no access. Each access field is separated by a comma.

3. Checking for special privileges.

If access was not granted by an ACE or the UIC protection mask, privileges are then evaluated. Users with elevated system privileges may have the ability to access objects regardless of protection offered by an ACE or the UIC protection. The bypass privilege (BYPASS), group privilege (GRPPRV), read all privilege (READALL), or system privilege (SYSPRV) amplifies the holder's access to objects.

Note: Administrators must be careful when granting elevated privileges that may allow users access to critical objects. This can compromise system integrity.

Using Auditing to Monitor Queues and Logical Name Tables

For some object classes, such as queues and logical name tables, a user may gain access based on alternate privileges. For example, the queue object allows full access to all queues for users with the operator privilege (OPER), and the logical name table object allows access to the system table for users with the system name privilege (SYSNAM).

To monitor queues and logical name tables, you can set an alarm ACE on those objects.

Example of Queue Security Auditing

```
$ set audit/audit/alarm/enable=access=(success,failure)/class=queue
$ show audit
System security alarms currently enabled for:
  ACL
  Authorization
  Audit:         illformed
  Breakin:       dialup,local,remote,network,detached
  Logfailure:    batch,dialup,local,remote,network,subprocess,detached
  QUEUE access:
    Failure:     read,submit,manage,delete,control
    Success:     read,submit,manage,delete,control

System security audits currently enabled for:
  ACL
  Authorization
  Audit:          illformed
  Breakin:        dialup,local,remote,network,detached
  Logfailure:     batch,dialup,local,remote,network,subprocess,detached

QUEUE access:
    Failure:      read,submit,manage,delete,control
    Success:      read,submit,manage,delete,control

$ reply/enable=security
$ initialize/queue/batch  test_queue
$ submit/noprint sys$login:login.com

%%%%%%%%%%%  OPCOM  23-NOV-2004 13:19:01.05  %%%%%%%%%%%
Message from user AUDIT$SERVER on TIGRES
Security alarm (SECURITY) and security audit (SECURITY) on TIGRES, system
id: 10
25
Auditable event:   Object access
Event time:        23-NOV-2004 13:19:01.05
PID:               0000008D
Source PID:        000000D8
Username:          SYSTEM
Process owner:     [SYSTEM]
Object class name: QUEUE
Object name:       SYS$BATCH
Access requested:  READ
Status:            %SYSTEM-S-NORMAL, normal successful completion

%%%%%%%%%%%  OPCOM  23-NOV-2004 13:19:01.09  %%%%%%%%%%%
Message from user AUDIT$SERVER on TIGRES
Security alarm (SECURITY) and security audit (SECURITY) on TIGRES, system
id: 1025
Auditable event:   Object access
Event time:        23-NOV-2004 13:19:01.08
PID:               0000008D
Source PID:        000000D8
Username:          SYSTEM
Process owner:     [SYSTEM]
Object class name: QUEUE
Object name:       SYS$BATCH
Access requested:  SUBMIT
Status:            %SYSTEM-S-NORMAL, normal successful completion

Job LOGIN (queue SYS$BATCH, entry 634) started on SYS$BATCH

%%%%%%%%%%%  OPCOM  23-NOV-2004 13:23:52.86  %%%%%%%%%%%
Message from user AUDIT$SERVER on TIGRES
Security alarm (SECURITY) on TIGRES, system id: 1025
Auditable event:  Detached process login
Event time:       23-NOV-2004 13:23:52.85
PID:              000000E0
Username:         SYSTEM
Process owner:    [SYSTEM]
Image name:       TIGRES$DKA0:[SYS0.SYSCOMMON.][SYSEXE]LOGINOUT.EXE
```

Example of Logical Name Table Security Auditing

```
$ set audit/audit/alarm/enable=access=(success,failure)-
/class=logical_name_table
```

```
$ show audit
System security alarms currently enabled for:
  ACL
  Authorization
  Audit:          illformed
  Breakin:        dialup,local,remote,network,detached
  Logfailure:     batch,dialup,local,remote,network,subprocess,detached
  LOGICAL_NAME_TABLE access:
    Failure:      read,write,create,delete,control
    Success:      read,write,create,delete,control

System security audits currently enabled for:
  ACL
  Authorization
  Audit:         illformed
  Breakin:       dialup,local,remote,network,detached
  Logfailure:    batch,dialup,local,remote,network,subprocess,detached
  LOGICAL_NAME_TABLE access:
    Failure:     read,write,create,delete,control
    Success:     read,write,create,delete,control

$ show log/table=lnm$system_table

%%%%%%%%%%%  OPCOM  15-DEC-2004 08:54:25.55  %%%%%%%%%%%
Message from user AUDIT$SERVER on TIGRES
Security alarm (SECURITY) and security audit (SECURITY) on TIGRES, system
id: 10
25
Auditable event:    Object access
Event time:         15-DEC-2004 08:54:25.53
PID:                0000000C4
Process name:       SYSTEM
Username:           SYSTEM
Process owner:      [SYSTEM]
Terminal name:      TNA3:
Object class name:  LOGICAL_NAME_TABLE
Object name:        LNM$SYSTEM_TABLE
Access requested:   READ
Status:             %SYSTEM-S-NORMAL, normal successful completion

%%%%%%%%%%%  OPCOM  15-DEC-2004 08:54:25.55  %%%%%%%%%%%
Message from user AUDIT$SERVER on TIGRES
Security alarm (SECURITY) and security audit (SECURITY) on TIGRES, system
id: 10
25
Auditable event:   Object access
Event time:        15-DEC-2004 08:54:25.53
PID:               000000C4
Process name:      SYSTEM
Username:          SYSTEM
Process owner:     [SYSTEM]
Terminal name:     TNA3:
Object class name: LOGICAL_NAME_TABLE
Object name:       LNM$SYSTEM_TABLE
Access requested:  READ
Status:            %SYSTEM-S-NORMAL, normal successful completion
```

Using an Alarm ACE on Files

An alarm ACE can be used to monitor access to files and other objects.

In the following example, an ACE on the project.txt file will display the access output to the enabled OPCOM terminal and will write that same information to the SECURITY.AUDIT$JOURNAL file**.**

Note: You must include success or failure or both (s+f) on the alarm ACE.

```
$ set security/acl=(alarm=security,access=r+w+e+d+s+f) project.txt
$ set security/acl=(audit=security,access=r+w+e+d+s+f) project.txt
$ directory/security project.txt
Directory DKA0:[000000.USERS]
PROJECT.TXT;1        [TEST]           (RWED,RWED,RE,)
(ALARM=SECURITY,ACCESS=READ+WRITE+EXECUTE+DELETE+SUCCESS+FAILURE)
(AUDIT=SECURITY,ACCESS=READ+WRITE+EXECUTE+DELETE+SUCCESS+FAILURE)
```

Enabling OPCOM Terminals to Monitor Security Alarms

From a privileged account, you can enable OPCOM to receive security messages using the following command:

```
$ reply/enable=security
```

The following is the output of an OPCOM enabled terminal display when an authorized user accesses the project.txt file.

```
$ type dka0:[users]project.txt
Security alarm (SECURITY) and security audit (SECURITY) on TIGRES, system
id: 1025
Auditable event:       Object access
Event time:            14-SEP-2004 11:15:47.24
PID:                   000000DF
Process name:          TEST
Username:              TEST
Process owner:         [TEST]
Terminal name:         TNA8:
Image name:            TIGRES$DKA0:[SYS0.SYSCOMMON.][SYSEXE]TYPE.EXE
Object class name:     FILE
File name:             TIGRES$DKA0:[USERS]PROJECT.TXT;1
File ID:               (8550,27,0)
Access requested:      READ
Sequence key:          0000C2CC
Status:                %SYSTEM-S-NORMAL, normal successful completion
```

Example of OPCOM Display

The following is the output of an OPCOM enabled terminal display when an unauthorized attempts to access the project.txt file.

```
$ type dka0:[users]project.txt
%TYPE-W-OPENIN, error opening DKA0:[USERS]PROJECT.TXT;1 as input -RMS-E-
PRV, insufficient privilege or file protection violation
Security alarm (SECURITY) and security audit (SECURITY) on TIGRES, system
id: 1025
Auditable event:        Object access
Event information:      file access request (IO$_ACCESS or IO$_CREATE)
Event time:             14-SEP-2004 11:17:17.74
PID:                    000000E0
Process name:           _TNA9:
Username:               GUEST
Process owner:          [GUEST]
Terminal name:          TNA9:
```

```
Image name:             TIGRES$DKA0:[SYS0.SYSCOMMON.][SYSEXE]TYPE.EXE
Object class name:      FILE
Object owner:           [TEST]
Object protection:      SYSTEM:RWED, OWNER:RWED, GROUP:RE, WORLD:
File name:              _TIGRES$DKA0:[USERS]PROJECT.TXT;1
File ID:                (8550,27,0)
Access requested:       READ
Sequence key:           0000C4EA
Status:                 %SYSTEM-F-NOPRIV, insufficient privilege or
object protection violation
```

Because AUDIT=SECURITY is enabled, you can also review the same information as above by analyzing the SECURITY.AUDIT$JOURNAL file from a privileged account.

Example of SECURITY.AUDIT$JOURNAL File
The following is an abbreviated display of the above alarms as they appear in the SECURITY.AUDIT$JOURNAL file.

```
$ set def sys$manager
$ analyze/audit/since=14-sep-2004:11:14
Date / Time            Type    Subtype    Node  Username    ID    Term
-----------------------------------------------------------------------
14-SEP-2004 11:15:47.24 ACCESS  OBJ_ACCESS  TIGRES TEST  000000DF TNA8:
14-SEP-2004 11:17:17.74 ACCESS  OBJ_ACCESS  TIGRES GUEST 000000E0 TNA9:
```

Note: Adding the /full qualifier to the above command provides the same output as seen in the OPCOM display above.

Removing ACEs, Audits, and Alarms from Objects

Use one of the following commands to remove the ACE from a file. The first command removes a single ACE. The second command removes both the audit ACE and the alarm ACE.

```
$ dir/sec project.txt
PROJECT.TXT;1        [TEST]                       (RWED,RWED,RE,)
(AUDIT=SECURITY,ACCESS=READ+WRITE+EXECUTE+DELETE+SUCCESS+FAILURE)
(ALARM=SECURITY,ACCESS=READ+WRITE+EXECUTE+DELETE+SUCCESS+FAILUR)
$ set security/acl=(alarm=security,access=r+w+e+d+s+f)/delete project.txt

$ dir/sec project.txt
PROJECT.TXT;1        [TEST]                       (RWED,RWED,RE,)
(AUDIT=SECURITY,ACCESS=READ+WRITE+EXECUTE+DELETE+SUCCESS+FAILURE)
$ set security/acl/delete=all project.txt

$ dir/sec project.txt
Directory DKA0:[000000.USERS]
PROJECT.TXT;1        [TEST]                       (RWED,RWED,RE,)
```

Example of Removing Security Alarm and Audit from Queue Object:

```
$ set audit/audit/alarm/disable=access=(success,failure)/class=queue
```

Example of Removing Security Alarm and Audit from Logical Name Table Object:

```
$ set audit/audit/alarm/disable=access=(success,failure)-
/class=logical_name_table
```

Summary

You can use the OpenVMS Auditing Utility and using Access Control Entries to monitor access to files and objects. There are other objects and system resources that can be monitored and restricted by using these two utilities. Please reference the OpenVMS Guide to System Security for additional information on using OpenVMS auditing and Access Control Lists (ACL) to help ensure the security of your OpenVMS system.

For more information

For more information about using these security methods, refer to the HP OpenVMS *Guide to System Security.*

Porting RPG: Moving a Compiler to the HP Integrity Server

Porting RPG: Moving a Compiler to the HP Integrity Server

Bruce Claremont, Software Migration and OpenVMS Consultant

Overview

This article covers MSI's experience in porting its Migration RPG compiler to the HP Integrity server, based on the industry-standard Intel® Itanium® 2 processor. The article covers the following subjects:

- Our software development methodology
- Our porting plan
- The porting process
- Product Q/A testing after the port was completed
- Summary and recommendations to others planning to port applications

Introduction

MSI is a small, privately held corporation that specializes in software migration and HP OpenVMS services. We also have a unique niche in the industry in that we provide the only RPG compiler available under OpenVMS, *Migration RPG*. This article covers our port of the compiler to OpenVMS running on the HP Integrity server line.

Software Development Methodology

This section briefly describes MSI's software development methodology. It's relevant to the porting process because our methodology greatly simplifies software porting.

MSI's software is developed using structured programming techniques. We stick to the rule of one entry and exit point for each routine, the only exception being routines that handle fatal errors. We strive to make routines as small and concise as possible and to reuse routines whenever possible.

Our code is well documented. In the case of Migration RPG, most of the documentation is embedded within the source code, with a summary document that describes the overall structure of the compiler and defines key internal layouts. Our standard for source code documentation is to provide enough descriptive text to allow a programmer unfamiliar with the programming language in use to understand the purpose and structure of each module.

Migration RPG is primarily written in HP Macro-32. It also contains a few modules developed in HP Fortran and HP C. The entire product comprises 120 source files containing about 320,000 lines of code.

I became lead software engineer on the product 20 years ago when its native environment was VAX/VMS. At that time, the product was both poorly documented and structured. It took about seven years to standardize and document the code while simultaneously adding new features and improving overall product quality. When MSI acquired the product in 1996, it was pretty much in full compliance with the standards described above. Because I am the founder of MSI, these standards continue to be enforced.

It is important to note that as Migration RPG was standardized and enhanced, a quality assurance suite of test programs was developed to test all aspects and features of the product suite. The basic rules of test suite development were:

- Every product feature has one or more test programs.
- Every bug fix generates one or more test programs.
- Every new feature generates one or more test programs.

The test suite is maintained under DEC Test Manager (DTM). It is highly structured and fully automated. It is critical to maintaining the quality of Migration RPG. I cannot overemphasize how important having the test suite was to a quick and successful port.

Porting Plan

MSI had two advantages as it prepared to port Migration RPG to the HP Integrity server:

- We specialize in software migrations so we have a great deal of experience in porting software applications.
- We had successfully ported Migration RPG from the VAX processor to the Alpha processor back in 1996.

We use the following steps to plan and implement a software port:

1. Inventory the application and remove any modules that are no longer in use. It makes no sense to port unused modules.
2. Develop test scripts, data, reports, and an acceptance test plan. Our existing DTM test suite took care of this requirement.
3. Develop a schedule and identify resources for the port. Because we were stepping into somewhat uncharted waters by being an early Integrity server adopter, setting a schedule was difficult. Our past software porting experience, specific experience porting Migration RPG to Alpha, and research into the tools available under Intel Itanium-based OpenVMS led us to estimate a 2- 3 week porting effort.
4. Train the personnel. In this instance, being an OpenVMS shop was advantageous. OpenVMS is OpenVMS, regardless of the underlying platform. Likewise, Fortran, C and DTM changed very little in moving to the Integrity server. Much to our surprise and pleasure, Macro-32 was ported to the Integrity server as well. Thus, personnel training requirements were minimal.
5. Walk before you run. When ever possible, we always port a small, self-contained application before moving on to a large one. This allows us to test our porting plan, the porting tools, and the target environment without a large investment or risk. We chose our EBCDIC to ASCII conversion product, CVTFILE, to conduct the initial test port.

Critical to the success of our port was the HP implementation of the following products on the HP Integrity server:

- OpenVMS operating system
- Macro-32 compiler
- DTM

C and Fortran are not listed as critical because the modules they support could have been re-written, if necessary. Thus, failure of the port of those components would have been an inconvenience, not a show stopper.

The Porting Process

I initiated the porting process by attending an HP/Intel Porting Forum. This gave me access to Integrity server equipment and HP engineers. It also had me working among my peers, which proved to be a rewarding experience in its own right.

My first step was to recreate our development environment on the Integrity server. I had brought along all of the setup procedures used on our Alpha development system to accomplish this. Configuring the Integrity server was no more difficult that configuring a new AlphaServer.

CVTFILE Test Case

I began with the test port of the CVTFILE utility. The utility is a straight-forward program written in Macro-32 that can convert data between ASCII and EBCDIC representation. It contained no compiler directives. It compiled and linked cleanly using its build procedure and passed the conversion tests associated with the product. It required no code modification to port successfully. From start to finish, the CVTFILE port was completed within an hour.

Encouraged by the ease of the CVTFILE port and the apparent maturity of the late beta releases of OpenVMS and its layered products, I was ready to tackle porting Migration RPG.

Migration RPG Port

Before discussing the Integrity server port of Migration RPG, a little background on the VAX to Alpha port is necessary. This information is especially pertinent if you are planning to port directly from a VAX system to the Integrity server.

Prior to being ported to the Integrity server, Migration RPG ran on both VAX and Alpha processors and was maintained via a common code base. In porting from the VAX to the Alpha, the only changes we needed to make to the Macro source code involved formatting of the external routine calls of modules in the shareable runtime image. We used compiler directives to accomplish this, as illustrated in Figures 1 and 2.

Figure 1: Original VAX Call

```
.ENTRY S3X_ACCEPT, ^M<R2,R3,R4,R5,R6,R7>
```

Figure 2: VAX/Alpha Common Code Call

```
.IF DEFINED VAX
        .ENTRY S3X_ACCEPT, ^M<R2,R3,R4,R5,R6,R7>
.ENDC

.IF DEFINED ALPHA
S3X$ACCEPT::    .CALL_ENTRY -
                  MAX_ARGS=4, -
                  HOME_ARGS=TRUE, -
                  PRESERVE=<R2,R3,R4,R5,R6,R7>
.ENDC
```

Because of the changes in external call structures and link vectors, the VAX to Alpha port necessitated creation of an Alpha-specific linker option file for the shareable image. This is one of only two places where our VAX and Alpha code sets are not identical. The other is the compile and link qualifiers used on the VAX and Alpha systems.

Obviously, being able to maintain a common code set across multiple platforms is desirable, so we hoped to accomplish this with the port to the Integrity server. Much to my relief, I found the Macro-32 compiler for the Integrity server to be fully compatible with the Alpha version of our Macro-32 code. Other than modifying the compiler directives to accommodate the Intel Itanium architecture, as shown in Figure 3, *no code changes were needed* to achieve clean compiles on all of our Macro modules. Likewise, our C modules only needed compiler directive modifications. Our Fortran modules compiled without modification.

Figure 3: VAX/Alpha/Integrity Common Code Call

```
.IF DEFINED VAX
        .ENTRY S3X_ACCEPT, ^M<R2,R3,R4,R5,R6,R7>
.ENDC

.IF NDF VAX
S3X$ACCEPT::    .CALL_ENTRY -
                  MAX_ARGS=4, -
                  HOME_ARGS=TRUE, -
                  PRESERVE=<R2,R3,R4,R5,R6,R7>
.ENDC
```

Because our code is standardized, modifying the compiler directives was a matter of running a couple of global find and replace operations via TPU. Likewise, our build procedures were equally easy to update. The Macro-32 compiler and linker on the Integrity server accept the same qualifiers as the Alpha. Modification to our build procedures involved changing a few IF statements to recognize the Intel Itanium architecture and default to the Alpha compile and link directives. Once the compiler directives and build procedures were updated, Migration RPG compiled and linked successfully.

From start to finish, achieving clean compiles and links of Migration RPG on the Integrity servers was accomplished in under 4 hours. This includes the time spent modifying source code and procedures to work correctly under the new architecture.

Q/A Testing

Testing began with a few manual tests conducted during the Porting Forum. Initial testing quickly revealed a problem with floating point numbers. Had I reviewed *Porting Applications from HP OpenVMS Alpha to HP OpenVMS Industry Standard 64 for Integrity Servers* porting guide more carefully, this would not have been an issue. The C compiler on Alpha systems defaults to G_FLOAT representation. C on Integrity server systems defaults to IEEE_FLOAT. The addition of the /G_FLOAT qualifier to our C compile command line resolved the issue.

After returning to the home office and taking delivery of our own Integrity server, serious testing using the DTM test suite was initiated. Initial testing revealed a couple of minor coding errors in our own code. These problems had never been detected by the VAX and Alpha-based test suites, but showed up when run on the Integrity server. Hence, the porting process served to improve our product.

Several tests failed during the initial pass because they report back file sizes in blocks. The same file on VAX, Alpha, and Integrity server will vary in size due to the difference in underlying architecture, so the DTM tests were catching the differences between files sizing on the Alpha and Integrity server system. Updating the DTM benchmark files on the Integrity server eliminated this issue.

Having addressed the minor issues, we ran the test suite again. Out of several hundred tests, we had only two failures. The errors proved to be a fault in the Macro-32 compiler. We conducted our port under a field test version of OpenVMS, so this was not unexpected. HP has addressed the issue and our Q/A test suite now runs cleanly.

During the testing phase, we also encountered problems with the symbolic debugger. The problems were encountered during the porting forum and workarounds were quickly provided by HP engineering. The problems have been addressed in the production release of OpenVMS Version 8.2.

During the Q/A test phase, we had no problems with the DTM software. This was a tremendous relief and had a significant impact on the speed of our port.

Summary

Our entire port of Migration RPG was completed with about 60 man-hours of effort. Preparation and planning consumed another 20 man-hours. Credit for the ease and speed of our port is twofold:

1. HP did their job well. The smoothness of the port speaks well to the care HP has taken in moving OpenVMS and its layered products to the Integrity server.
2. MSI did its job well. Being specialists in software migrations, we have developed our products with an eye towards portability and ease of maintenance. While Migration RPG is a sophisticated, complex application, its individual modules are very structured and as simplistic as possible. Our code is highly standardized, making changes that impact several modules easy to accomplish. Our build procedures are automated, allowing ease of execution and quick resolution of compile and link problems. Our test suite is large, comprehensive, and automated, permitting thorough and efficient testing. Finally, we documented our code well, greatly simplifying code review and error resolution.

Recommendations

The steps needed to prepare for a port are pretty simple:

1. ***Prepare:*** Clean up source code, resolve known problems, and eliminate obsolete modules. In short, clean house.

2. ***Plan:*** Schedule the process; allocate human, software, and hardware resources; define and prepare test material; and test it. Be realistic with the schedule and be prepared to be flexible should your port have difficulties.
3. ***Port:*** Conduct the port. Review HP porting guides and modify code as required. Update build procedures and source code to accommodate the new architecture. Compile and fix problems. Link and fix problems. Conduct initial tests.
4. ***Test:*** Execute the acceptance test plan. Fix problems and run the tests again. Repeat until all problems are resolved.
5. ***Celebrate:*** Once you've ported and successfully tested your code, throw a party. You'll deserve it.

Depending upon the state of your code, documentation, and application knowledge, the porting process can run the gauntlet from simple to arduous. Clean code, careful planning, and quality testing will go a long way towards making the process simple and successful. Good luck!

For more Information

More information about MSI products and services can be found at http://www.MigrationSpecialties.com.

OpenVMS Technical Journal V6

Using Python for OpenVMS Web Development

Using Python for OpenVMS Web Development

Author: Jean-Francois Pieronne

A few years ago, a large company whose data-processing architecture is built around OpenVMS were seeking a "Webification" solution for its applications. This company naturally wished to deploy this solution on OpenVMS.

A major criteria criterion was to offer the level of security and audit trail auditing equivalent to that offered by the classic character-cell based applications on OpenVMS using a VT terminal. It was therefore imperative to keep the authentication and security of OpenVMS.

Each user must run his applications in his own process just as in the case of a VT terminal session, only the interface has changed.

A system was needed that was sufficiently light in resource consumption so as to be able to instantiate each user, without an excessive consumption merely supporting such an infrastructure, and providing the required security, identification, and traceability.

Usually application servers like Tomcat or equivalent commercial offerings do not have this level of granularity because of their heavy resource usage.

For reasons of functionality, integration with OpenVMS, and performance, the WASD server package was selected.

Other criteria for the choice of the language for development included:

- Productivity in the creation and maintenance of the programs
- Wealth of libraries, existing tools
- Size of the installed base and the reputation of the language

A scripting language was preferable, due to the much higher productivity of developers in these types of languages than in other more traditional environments.

Finally, Python was selected as the principal language of development because, in conjunction with the WASD web server, it gave the best possibility of successfully deploying the required environment.

The Python Language

Python is a *portable, interpreted, interactive, object-oriented* programming language.

Python combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and strong dynamic typing. New built-in modules are easily written in C or C++. Python is also usable as an extension language for applications that need a programmable interface.

Python runs on most operating systems and for each has interfaces to many system calls and libraries.

The first public version was released in 1991. In 2001, Guido van Rossum, who is the author of Python, started a foundation named "Python Software Foundation," devoted to advancing open source technology related to the Python programming language.

The Python implementation is copyrighted but freely usable and distributable, even for commercial use.

Since its origin this language has met a growing success, its user-base doubling each year. Many books, articles and Internet sites are regularly devoted to it. Python is currently used by many companies such as Google, Yahoo, Redhat, Microsoft, Nokia, and others.

In the OpenVMS world, several companies use Python intensively; some of them have based their whole Web architecture on the WASD/Python combination.

One company that deploys a software system on hundreds of systems also uses Python intensively, including critical applications.

Here is an example of a script used to replace all the events of a character string in a whole series of HTML files contained in a directory:

```
import glob
for fn in glob.glob('./*.html'):
    res = open(fn).read().replace('old string', 'the new string')
    open(fn, 'w').write(res)
```

For more information and other demonstrations to refer to the site http://www.python.org.

Porting Python

A first port of Python 1.5.2 had been carried out, but for technical reasons and to ensure easier development in the future, it was decided to set out again from scratch.

Currently, this port is the result of collaborative work among several people, with the occasional participation of HP employees.

The current version of Python for OpenVMS is 2.3.5. New kits are periodically released, including bug fixes, or updates of the libraries contained in this port.

Since version 2.3.5, as the kits are now versioned, it is possible to have several versions of Python installed on the same system.

The Challenge

We had to ensure a port that is perennial and evolutionary, that facilitates the port of future versions, and is not a one-shot port.

This includes submission of the patches carried out specifically for OpenVMS for inclusion into the main stream Python codebase. This has been partially accomplished. However, there still exist some patches that are not included. Indeed, this is sometimes difficult. For example, it was necessary to explain how a file system like RMS works, because most people think that a file is limited to a flow of bytes.

We also had to ntegrate specific OpenVMS features, for example:

- Allow the calling of system services or various runtime library routines (SYSxxx, LIBxxx etc)
- Offer access to RMS files and not be limited to sequential files with organisation type of STREAMLF
- Utilize environments specific to OpenVMS such SGBD Oracle/Rdb or the WASD web server
- Allow the addition of modules or the evolution of existing modules without having to rebuild the whole interpreter.

- Allow the simplest possible port, under OpenVMS, of Python applications initially developed for other OS. For example, it was decided that for Python OpenVMS is seen as a posix system and that the routines for access to the file-system followed posix naming rules and not those of OpenVMS naming rules.
- Provide a form of installation integrated into OpenVMS.

The Choices We Made

The Python interpreter is built as a shareable image. The Python program itself is only one small program of 24 lines.

The extensions for OpenVMS are also built in the form of shareable images. Therefore, it is simple to upgrade a module or add new ones.

Moreover, as these libraries are dynamically activated, it is possible to deploy Python with all of the modules provided, on systems that do not have the specific environments installed (for example Oracle/Rdb or the Web server WASD).

When the modules use libraries that can also be used outside of Python, this functionality is provided in the form of separate kits.

Here is the list of the libraries for which there is a PCSI kit and a Python interface:

- Zlib
- LibBZ2
- LibJPEG
- LibPNG
- Freetype
- LibImaging
- LibGD
- GDChart
- LibXML2 Libxslt/Libexslt
- OPENSSL
- Swish-E

The OpenVMS Python kit and all the separate libraries are distributed in the form of PCSI kits and are therefore easily installable and upgradable.

It was also decided that the minimum supported version of OpenVMS would be V7.3 with the latest version of the CRTL. (There is a partial version for OpenVMS V7.2, but it is not maintained.)

It is strongly advisable to install Python on a ODS-5 disk. However, in the absence of a physical disk of this type, it is possible to use a virtual disk in a container file with LDdriver, the latter being included with OpenVMS starting from the V7.3-1. LDdriver for older versions of OpenVMS is available on the Freeware CD. (Editor's Note: Look for the article about the LDdriver in this issue of the OpenVMS Technical Journal.)

Some Examples of Integration with OpenVMS

Example 1

Display users whose accounts are not disusered and who have not logged on for 31 days or more:

```
import vms.user, vms.starlet, vms.uaidef
def fcmp(u1, u2):
    return cmp(u2.lastlogin_i, u1.lastlogin_i)
users = vms.user.all_users()
users = users.values()
# descending sort on last login interactive
users.sort(fcmp)
s,delta = vms.starlet.bintim('31 0:0:0.00')
s,curtim = vms.starlet.asctim()
s,minlogin = vms.starlet.bintim(curtim)
minlogin += delta
for user in users:
    if (not (user.flags & vms.uaidef.UAI_M_DISACNT) and
        0 < user.lastlogin_i < minlogin):
        print "%-33s %s" % (user.username,
                            vms.starlet.asctim(user.lastlogin_i)[1])
```

This gives the following result:

```
USER1                              8-APR-2005 13:38:43.51
FIELD                              8-APR-2005 13:16:48.21
DEMO                              27-SEP-2002 12:49:57.24
```

Example 2

Sample code to fetch information from queues, display all queue names and descriptions and for each queue all jobs name and id:

```
from vms.rtl.lib import getqui
from vms.quidef import *
from vms.jbcmsgdef import JBC__NOMOREQUE, JBC__NOSUCHJOB,\
                          JBC__NOMOREJOB
getqui(QUI__CANCEL_OPERATION)
while True:
    queue_name = ''
    try:
        s, v, queue_name = getqui(QUI__DISPLAY_QUEUE, QUI__QUEUE_NAME,
                                  None, '*')
    except VMSError, e:
        if e.errno  == JBC__NOMOREQUE:
            break
        else:
            raise
    s, v, queue_desc = getqui(QUI__DISPLAY_QUEUE,QUI__QUEUE_DESCRIPTION,
                              None, '*', QUI_M_SEARCH_FREEZE_CONTEXT)
    print 'Queue:', queue_name, '<', queue_desc, '>'
    while True:
        try:
            s, v, js = getqui(QUI__DISPLAY_JOB, QUI__JOB_STATUS,-1,None,
                              QUI_M_SEARCH_ALL_JOBS)
        except VMSError, e:
            if e.errno in (JBC__NOMOREJOB, JBC__NOSUCHJOB):
                break
            else:
                raise
        s, v, jn = getqui(QUI__DISPLAY_JOB, QUI__JOB_NAME, -1, None,
                          QUI_M_SEARCH_ALL_JOBS |
                                    QUI_M_SEARCH_FREEZE_CONTEXT)
        s, v, en = getqui(QUI__DISPLAY_JOB, QUI__ENTRY_NUMBER, -1, None,
                          QUI_M_SEARCH_ALL_JOBS |
                                    QUI_M_SEARCH_FREEZE_CONTEXT)
        print '    Job: %s (%s)' % (jn, en)
```

This gives the following result:

```
Queue: ASSP_QUEUE < queue for Anti-Spam SMTP Proxy Server >
    Job: LOGIN_ASSP (6)
Queue: LASER1 <  >
Queue: SCHEDULER <  >
    Job: IPCHECK (562)
    Job: CHECKAUDIT (292)
    Job: update_spamdb (530)
Queue: SETI$BATCH <  >
Queue: SYS$BATCH_NODE1 < Queue batch NODE1 >
Queue: SYS$BATCH_GENERIC < Queue batch generic >
Queue: TCPIP$SMTP_NODE1_00 <  >
```

Example 3

A small Rdb example which displays the user tables and views from the famous "mf_personnel" demonstration database. An iterator is used to read the lines.

```
import rdb
attach = rdb.statement("attach 'filename mf_personnel'")
commit = rdb.statement("commit work")
readonly = rdb.statement("set transaction read only")
curs = rdb.statement("""select rdb$relation_name from rdb$relations
where rdb$system_flag = ? order by rdb$relation_name""")
attach.execute()
print "users relations name:"
readonly.execute()
curs.execute(0)
for line in curs:
    print line[0]
commit.execute()
```

Which gives the following results:

```
users relations name:
CANDIDATES
COLLEGES
CURRENT_INFO
CURRENT_JOB
CURRENT_SALARY
DEGREES
DEPARTMENTS
EMPLOYEES
JOBS
JOB_HISTORY
RESUMES
SALARY_HISTORY
WORK_STATUS
```

For more information and other examples, refer to the site Python for OpenVMS.

Applications, Libraries, and Tools

It is generally easy to port a Python application to OpenVMS provided that it doesn't use platform specific libraries. For example, the porting of the application server Webware which represents around 40,000 lines of code required only the modification of a single line in the installation script. A PCSI installation kit for Webware will be shortly available.

Many other tools & libraries are installed and use the standard installation procedure without any modification for example the the Cheetah templating system which comprises around 12,000 lines of Python code or the PyChecker tool which is a Python source code checking tool and comprises approximately 5,000 lines of Python code.

A Web T4 viewer has been developed in a few days and comprises about only 900 lines of Python code. A demonstration is available from http://vmspython.dyndns.org/.

For More Information

Some interesting reading may be:

- The article from John K. Ousterhout creator of the TCL language: Scripting: Higher Level Programming for the 21st Century
- The article from Ill Venners: Use the Best Tool for the Job
- The comparative study written by Lutz Prechelt: An Empirical Comparaison of C, C++, Java, Perl, Python, Rexx, and Tcl.
- Artima.com's six-part interview with Python creator Guido van Rossum: http://www.artima.com/intv/guido.html
- The following presentations from Stefen Ferg:

  - The Business Case for Agile Languages
  - Python:a Powerful, Easy-to-Use, Open-Source Scripting Language
  - Python & Java: a Side-by-Side Comparison
  - More material can be accessed from http://www.ferg.org/python_presentations/index.html.

For more information about T4, read the following article by Steve Lieman a previous OpenVMS Technical Journal:

TimeLine-Driven Collaboration with "T4 & Friends": A Time-saving Approach to OpenVMS Performance

Case Studies Using EMC Legato NetWorker for OpenVMS Backups

Case Studies Using EMC Legato NetWorker for OpenVMS Backups

Siobhán Ellis, Senior Technical Consultant, ENSTOR, Sydney, Australia

Overview

EMC Legato NetWorker is the only product that incorporates OpenVMS into an enterprise-wide data protection solution providing direct backup to tape or disk as well as support for Oracle and Oracle Rdb. This article covers two real solutions that were delivered – one using EMC Symmetrix and CLARiiON Disk Libraries, and the other using host-based OpenVMS volume shadowing, HP storage, and an MSL5000 tape library.

Introduction

Backups are critical for any operating system and OpenVMS is no exception. There have been many solutions on the market, including offerings such as SLS and ABS from HP. Until recently, nearly all of these solutions concentrated on using BACKUP as the data mover. One of the products to break that paradigm is EMC Legato NetWorker.

By using its own data mover, NetWorker can write multiple streams of data in parallel to a single backup device. NetWorker also writes that data in a format that is operating system independent, which means that a tape written on an OpenVMS system can be read by a Windows system. NetWorker can often back up an entire OpenVMS system much faster than the traditional single stream method because the bottleneck often is not the tape device itself, but the process of getting the data off the disk fast enough to keep the tape device spooling. In one such case, NetWorker cut backup times in half without changing the hardware.

NetWorker is also extremely flexible. It provides the ability to share tape devices, tapes, and libraries among systems – even among operating systems – to maximize the use of resources.

This article describes two real backup solutions using EMC Legato NetWorker on OpenVMS.

Case Study 1: Backup of Data at Two Sites

The Scenario

The customer has two datacenters some distance apart. One site is the production site and the other is the disaster recovery site, which also houses development systems. The two sites are connected using TCP/IP and DECnet. There are about 90 OpenVMS servers spread across the two datacenters.

Data is stored on EMC Symmetrix disk arrays. Production data is replicated from Site 1 to Site 2 using EMC SRDF, an array-based duplication, over FCIP (fibre channel over IP) using CISCO MDS. All data, production, and development is in a RAID 1 set; this is locally copied using EMC TimeFinder to create BCVs (business continuation volumes), which is essentially a third copy of the data.

All OpenVMS systems have two fibre channel host bus adapters (HBAs) to provide dual paths to the data, and they are all in clusters. They were currently being backed up using a mixture of manual procedures and TAPESYS.

The Requirements

The customer wanted a solution for their OpenVMS systems only. Their UNIX and Windows systems were already covered by another backup solution from another vendor. Because of stability issues with the other backup product, they were not considering using it for their OpenVMS solution.

The customer needs were:

- A modern backup solution with the following features:

  - o Extensive reporting
  - o Administrative graphical user interface (GUI)
  - o Security
- Reduced backup times
- Ability to keep all backups for two months and occasionally keep a backup for 10 years
- High availability
- No operational impact on the clusters being backed up
- Multiple copies of the data without moving data over the corporate IP data network

The Solution

Figure 1 shows the solution that was designed for this customer's needs. The solutions chosen to address individual needs are discussed in the sections following the figure.

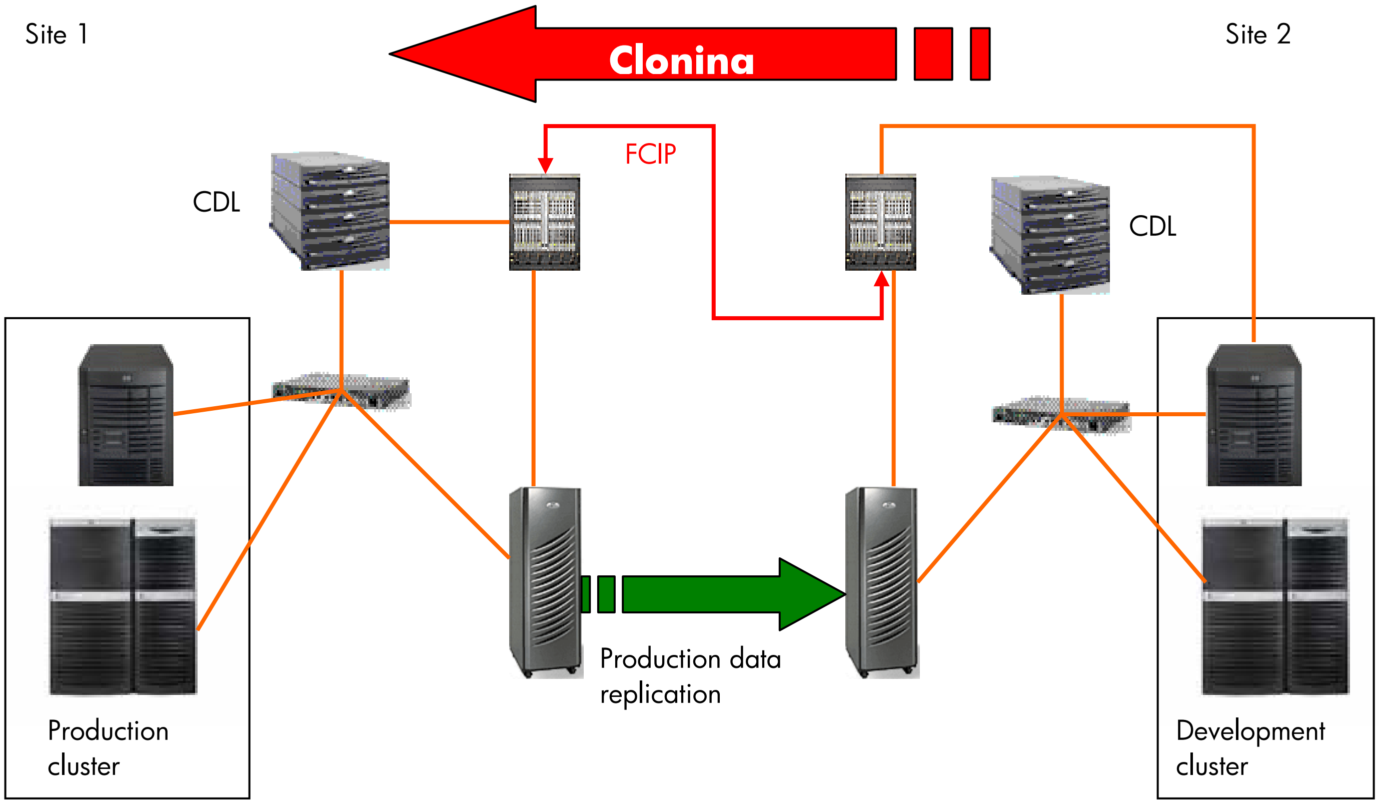


Figure 1 Production Cluster and Development Cluster Connections to SANs and CDLs

A Modern Backup Solution

EMC Legato NetWorker V7 was chosen because of its flexibility to deliver a solution that meets all the basic requirements. In conjunction with NetWorker Management Console (NMC), this solution provides a secure environment where the operations staff can be limited to performing only certain functions using a Java-based, web-start application. NMC also provides sophisticated and extensive reporting, and all actions taken on NetWorker are recorded.

Reduced Backup Times and Storage Capacity

It was determined that a disk-to-disk solution would provide the highest possible throughput for both restores and backups. However, with the amount of data required to be stored, EMC Symmetrix disk was too expensive to use as a backup device. The solution required lower cost disks such as serial ATA disks. EMC's mid-range disk solution is CLARiiON, which, in the spring of 2005, does not support OpenVMS. However, EMC had just launched the CLARiiON Disk Library (CDL), which was a perfect solution.

The CDL is a virtual tape library. It is essentially a collection of fibre-attached, serial ATA disks in a cabinet fronted by a high-availability pair of Linux servers that emulate a tape library using FalconStor VirtualTape Library (VTL) software. HP has just launched its equivalent, the StorageWorks 6000 Virtual Library System.

A CDL was implemented at each site with enough space to keep all data for two months. To ensure that the CDL is always available using fibre channel, two ports were dedicated to local data backup.

Each CDL was set up to emulate an ATL P3000 tape library, which is physically the same as an HP ESL9000, with 8 SDLT 320 drives. Site 1 also had a smaller library set up and was connected to the local MDS, which is described later.

High Availability

The NetWorker server, which holds the media database and client file indexes and controls the execution of backups, was installed on a Sun Solaris server at Site 1. To comply with the customer's standards, high availability was achieved by having a second server available at Site 1 and having the metadata stored on a partition on the Symmetrix; this data was then duplicated to the remote site where a third server was set up and ready to take over from Site 1 if necessary.

As already noted, the CDLs contain a pair of high availability Linux systems. All the disks are configured in RAID 5 sets, giving the CDLs a much better MTBF (mean time between failures) than can be provided by a real tape library.

No Operational Impact on the Clusters Being Backed Up

The customer had recently consolidated their OpenVMS systems, moving many servers onto fewer GS Series servers. This move had freed up a number of smaller AlphaServers, which could now be used as backup appliances in each cluster and as NetWorker storage nodes. An AlphaServer 4000 could push four data streams at once before saturating the CPU, at which point any more streams would degrade overall performance. Each storage node was connected to the local CDL using two fibre channel host-based adapters. At Site 2, one storage node was also connected to the local MDS so it could see the small virtual tape library at Site 1.

Multiple Copies of the Data Without Moving Data Over the Corporate IP Data Network

Because the production data is replicated at both sites, all that was required was to split the business continuation volumes (BCVs) simultaneously at both sites and back up the data locally. This was done using the preprocessing and postprocessing capabilities of NetWorker, which allow you to run command procedures before and after the backup on a particular client. The steps are as follows:

1. Preprocessing:
    a. Breaks each disk in the backup saveset out from the BCVs.
    b. Mounts each disk locally with the correct logical name, so it supersedes the SYSTEM logical.
2. The backup runs and backs up the mounted BCV disk.
3. When the backup is complete, postprocessing does the following:

   a. Dismounts the disks.
   b. Puts the disks back into the BCVs.

Because clients at both sites can split BCVs at the same time, duplicate backups are achieved, but are backed up directly to local CDLs.

The development data was a different matter because it is not replicated from Site 2 to Site 1. Again, the preprocessing and postprocessing capabilities were used to split the BCVs just as with the production data. However, after the backups are finished, NetWorker initiates an automatic clone operation to perform a copy of all the development savesets from the CDL in Site 2 to the small CDL in Site 1.

It took about three weeks to develop and test the preprocessing and postprocessing command procedures.

The Result

All the customer requirements were met, and the solution went into production. With no special tuning, one product system was backed up and cloned in the time it used to take to perform just the backup. This was done with a single clone stream running. It is possible to initiate multiple clone streams and cut cloning time by two thirds.

Case Study 2: Backup of a Cache Database as Part of a Heterogeneous Medical Information System

The Scenario

The customer has a number of healthcare information systems based on the Caché database on OpenVMS, and a number of web, file, and SQL servers running on Microsoft Windows. Storage for OpenVMS is connected using HSG80 controllers, and all disks are shadowed using host-based volume shadowing. Windows systems have direct-attached SCSI disks. OpenVMS was being backed up using scripts based on BACKUP, and Windows was backed up using a workgroup class backup tool.

The Requirements

Customer wanted a solution with the following capabilities:

- Works for both Windows and OpenVMS
- Does not require applications to stop running.
- Runs automatically so support staff can go home at a reasonable hour.

The Solution

A NetWorker server running on Windows with OpenVMS storage nodes filled the bill. Windows web servers, file servers, and SQL servers all are covered by the standard NetWorker and associated modules. Backup is done to an MSL5000 series library, which is shared between Windows and OpenVMS.

The only issue concerned the database because, unlike Oracle or Oracle Rdb, Caché does not have an API with which a backup tool can interface. However, Caché does have a command line that allows you to put the database into a suspend mode, and then later into a resume mode.

The new backup flow is as follows:

1. Start the Backup.
2. Use NetWorker preprocessing to do the following:
    a. Suspend the Caché database.
    b. Drop disks from the shadow sets.
    c. Mount the disks in the group with their correct logical names, thus overriding the system logicals.
    d. Resume the database.
3. Back up the disks on the system.
4. Use NetWorker postprocessing to minimerge the disks back into the shadow sets.
5. Automatically clone the data to ensure there are two copies, one for offsite and one for onsite.

The command procedures took about one week to write and test.

The Result

Backups across Windows and OpenVMS are running reliably and consistently. The applications do not shut down and support staff no longer needs to worry whether the data is being backed up.

Summary

OpenVMS is an excellent operating system for many reasons, some of which are its robustness and reliability. Consequently, system managers are often reluctant to use a backup utility that is not native to OpenVMS. While NetWorker performs no magic tricks, it does use the RMS APIs for OpenVMS, thus guaranteeing correct backup and restoration of data because it is using basic and legal OpenVMS system calls. Indeed, in tuning NetWorker to work best with OpenVMS, the NetWorker for OpenVMS engineering team worked closely with the RMS engineering team.

In these two scenarios, one using EMC storage and the other using HP storage and OpenVMS volume shadowing, NetWorker on OpenVMS was easily customized to meet the particular requirements of each customer to deliver a reliable data protection solution. These solutions were chosen as examples because of their complexity and the author's intimate involvement with the service delivery. There are many standard installations of NetWorker on OpenVMS where no customizations are performed at all. There are also other variants, particularly with the Caché database, where HP's EVA storage is used and snapshots are performed rather than splitting shadow sets.

With the addition of NetWorker support for Oracle and Oracle Rdb databases, the possibilities of using NetWorker on OpenVMS are boundless with minimal customization. Using NetWorker on OpenVMS, you can help to bring OpenVMS operations into the enterprise, reducing the total cost of ownership by sharing backup resources with other operating systems and reducing training needs because NetWorker works across many operating systems.

NetWorker is the most mature and best heterogeneous data protection tool available. Now that these capabilities are available on OpenVMS, the best operating system, the combination of NetWorker and OpenVMS makes an unbeatable backup solution.

For more information

See the ENSTOR web site at www.enstor.com.au.
Find out more about the EMC Legato NetWorker on the NetWorker product page.
Find out more about NetWorker on OpenVMS in the OpenVMS Journal V4 and OpenVMS Journal V1.

OpenVMS Technical Journal V6

Automatic Program Generation with MySQL and PHP

Automatic Program Generation with MySQL and PHP

Dick Munroe

Overview

It's all about leverage. During the heat of the last presidential election, I found myself between engagements and decided to go to Florida and work for the presidential campaign. After spending 3 days with another volunteer building PCs and installing networks for the Florida Democratic Party (FDP) headquarters and the north Florida offices, I found myself behind the keyboard building web applications with the rest of the IT department. I should qualify that. I *was* the IT department – at least the technical staff part of it. My responsibilities including designing, implementing, and deploying new web applications for everything from eCommerce (donations, and so forth) to volunteer and candidate management to results reporting to statistical analysis, whatever was needed; even better, it was all needed today (or tomorrow at the latest) and as inexpensively as possible.

This article is one part praise for the free software community; another part discussion of one of the techniques I used to get applications out the door fast enough to be useful in the time we had before the election.

The Computing Environment

What I did for the Florida Democratic Party was grind out web applications. At the state and local level of political parties, cost is everything and in the UNIX environment, the free software movement provides most generously. When I arrived, the principal web development tools were MySQL and PHP, which were used to deploy content from an Apache web server running on FreeBSD.

MySQL is an SQL compliant database that is capable of scaling smoothly from very small to very large data stores, and that supports transactions, queries over the net, and so forth. It's free, very well supported, and performs well. We used it in every web application deployed during my time with the FDP.

PHP (Personal Home Page) is yet another "swiss army knife" language for developing web pages. PHP version 4 is a 3rd generation programming language with object oriented extensions allowing inheritance, polymorphism, and introspection. PHP also has an extension interface and extensions (dynamically linked libraries) to the language have been written for everything from graphics to data base interfaces. The FDP used PHP 4. Since the election, PHP 5 has been released with significantly improved object-oriented capabilities. All the work discussed here is implemented in PHP 4 and easily ported to PHP 5.

A full discussion of Apache and FreeBSD is beyond the scope of this article. However nothing done by the FDP was specific to either Apache or FreeBSD. All that was needed was a server capable of running PHP and a platform that allowed integration of PHP with a web server and communication with a MySQL server. Technically speaking, the server running Apache didn't have to be capable of running MySQL but it was convenient that it did.

Each application at the FDP required the design of one or more tables in the FDP primary database. Database design wasn't all that difficult for the vast majority of applications that were on the table, but any tool is better than no tool. MySQL is not difficult to deal with when doing rapid, interactive database design, so initially that's what we did. Eventually we found and started using a terrific MySQL specific database design and maintenance tool called phpMyAdmin. All the database table examples and figures for this article were created using phpMyAdmin. If you're going to be responsible for any aspect of MySQL database administration, phpMyAdmin is a must.

The development workstations were all Windows boxes running Windows 2000. It is not my favorite platform but the FDP couldn't argue with the price of the hardware (all hardware was donated and working workstations were constructed from whatever parts could be salvaged) and had a site license for W2K. Fortunately there are free (or at least inexpensive) editors with language specific extensions for the Windows environment. I downloaded Cygwin (a UNIX CLI layer for Windows) and started using EMACS and other standard UNIX tools for development on the Windows PC I was using.

Application Development at the FDP

The first job I tackled was fixing up an application written by a couple of college student volunteers that did volunteer management.

[Present data to the user]
Get data from the user
Validate user data
Store/Update using MySQL

Table 1 - Structure of Volunteer Management Application

The structure of the volunteer management application is shown in Table 1. Each "layer" of the application was done in an ad hoc manner. Each page of the application basically duplicated whatever code was needed to interact with the user and the database, perform data validation, and so forth. An enormous amount of the code written for this application dealt with the interface to the MySQL database. Starting sessions, validating data, storing or or updating data, and closing sessions were all coded explicitly. No attempt had been made to factor out the database access details into a separate function library or class and the quantity of the code dedicated to dealing with the database obscured the details of the application under development, making it a lot harder to extend. Eventually I finished this job and moved on to the next.

Accepting Donations

The next application was for accepting donations. The details of the application aren't important; what is important is that it would be another web-enabled database application similar in kind, if not in detail, with the volunteer management application.

Given what I had seen in the volunteer management application, I wanted to come up with a more general view of the FDP's web-enabled applications and then use that view to drive the creation of tools that would make it easy to implement those applications. After some thought I realized that most of the applications used regularly by the FDP could be modeled as shown in Table 2.

[Query MySQL Database]
Display data to the user
Collect data from the user
Organize collected data
Validate collected data
Store/Update data in MySQL

Table 2 - FDP Application Architecture

Overall, an application should do the following:

- If data were being changed, one or more queries would be made to the appropriate database.
- Data (if any) would be displayed to the user.
- Data would be collected from the user.

- Data would be organized in structures, making it simple for validation and eventual storing into the appropriate database tables.
- Data would be validated and any additional interactions with the user would be done to correct any errors.
- Data would be stored or updated in the MySQL database.

Given that data was being stored in a MySQL database, I felt that the underlying data abstractions should closely model the relational storage model (tables), making the usual sorts of queries done by the applications easy, while not prohibiting the writing of substantially more complex queries. Ideally, most of the "what data needs to be read/written" from the database would be "automagically" figured out so that the applications could say things like "update the data" and the right things would happen.

Clearly the place to start was was with a decent data base abstraction layer – one that hid most of the tedious details of dealing with MySQL while not overly restricting access to MySQL's lower level features. This problem has been solved many times in the past and a quick session with Google turned up www.phpclasses.org, an enormously useful site if you're into PHP programming. www.phpclasses.org is dedicated solely to the collection and distribution of PHP class libraries. The code distributed by www.phpclasses.org comes from all over the world and varies in purpose from the sublime to the ridiculous and in quality from completely professional to totally amateur. I've saved myself a lot of time and my clients a lot of money by using things I've found from www.phpclasses.org either in whole or in part. If you're doing any serious PHP software development, you owe it to yourself to join www.phpclasses.org. (Yes, it is free, but donations are accepted.)

On www.phpclasses.org, I found the DB class. DB provided an object-oriented interface to MySQL. This solved my initial requirement for something a little higher level than the PHP interface to MySQL. It did not matter that the only database supported was MySQL. The FDP had standardized on a Linux, Apache, MySQL, PHP (aka "LAMP" – or LAMP-like, since the platform was actually FreeBSD) environment, so portability was not of immediate concern. One of the reasons that I chose the DB class was that, if necessary, the DB could easily be ported to support other or additional databases.

This solved the problem of the tedium of accessing the database but it didn't address the problem of a general database table-oriented data collection that could easily interact with any database.

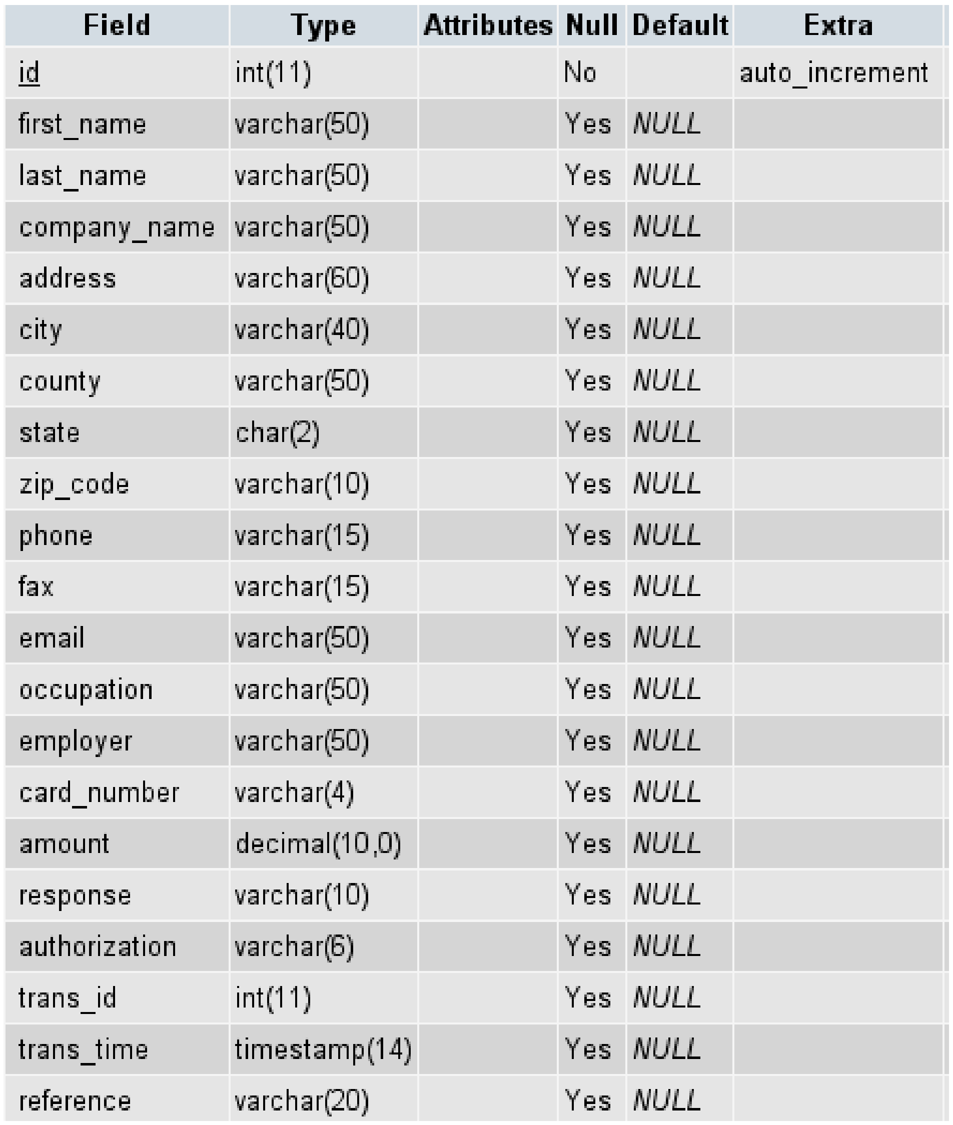
To solve this problem, I designed and wrote the SQLData class. This class evolved over time, but it was designed to:

- Associate an instance of the class with a specific table in a database.
- Be organized so that data to be stored to or fetched from the database can be manipulated by the name of the field.
- Keep track of the state of data in the instance so that minimal updates can be automatically performed.
- Provide iterators so that loops processing entire tables can be easily written.
- Provide hooks for structuring data as arrays or objects, and potentially arrays or objects stored in other tables

After we did the design of the table for the donations application, I dove into the coding process. The table design for the donations application is shown in Figure 1. As you can see, it is straightforward, with much of the complexity resulting from the eCommerce interface rather than anything fundamental in the nature of the problem of donating money.

Obviously enough, for every field defined in the table, you need a way to access, modify, and store the field's data. Even using SQLData, I had a lot of work to do. Each field needed a read/modify interface (plus additional ones that came in handy once I got into the details of actually doing the work, like controlling the "dirty" state of fields). This was a lot of typing, and it was error-prone too.

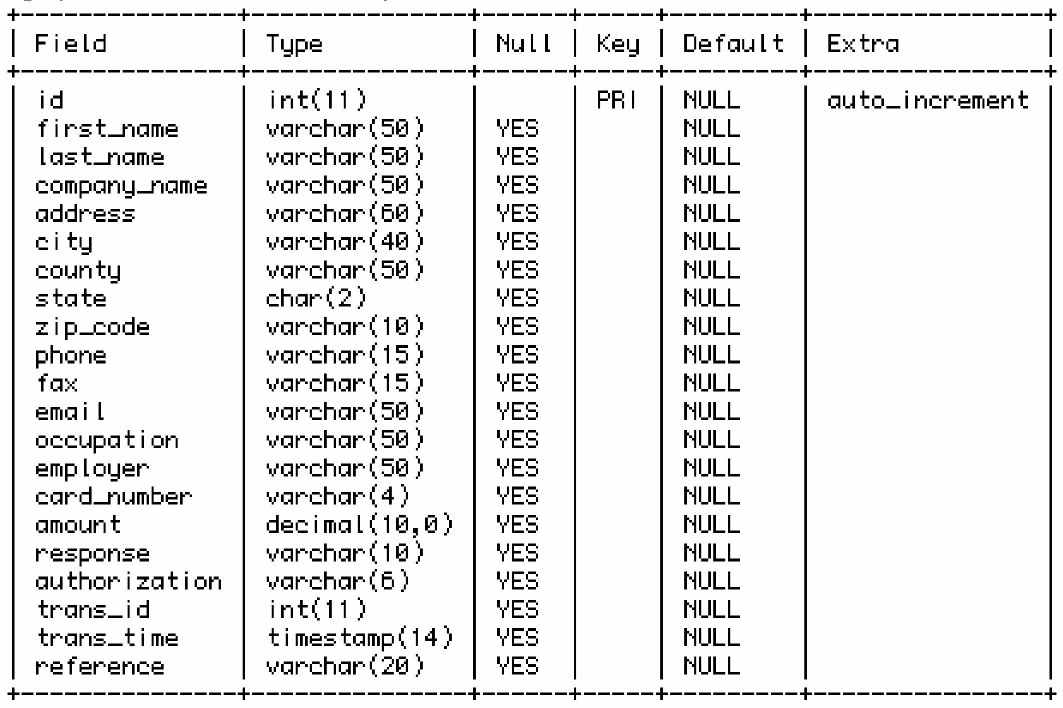
Much of the work was largely "cut and paste," but if that's all a human is doing, then maybe it is possible to get a computer to do it.



Field	Type	Attributes	Null	Default	Extra
id	int(11)		No		auto_increment
first_name	varchar(50)		Yes	*NULL*	
last_name	varchar(50)		Yes	*NULL*	
company_name	varchar(50)		Yes	*NULL*	
address	varchar(60)		Yes	*NULL*	
city	varchar(40)		Yes	*NULL*	
county	varchar(50)		Yes	*NULL*	
state	char(2)		Yes	*NULL*	
zip_code	varchar(10)		Yes	*NULL*	
phone	varchar(15)		Yes	*NULL*	
fax	varchar(15)		Yes	*NULL*	
email	varchar(50)		Yes	*NULL*	
occupation	varchar(50)		Yes	*NULL*	
employer	varchar(50)		Yes	*NULL*	
card_number	varchar(4)		Yes	*NULL*	
amount	decimal(10,0)		Yes	*NULL*	
response	varchar(10)		Yes	*NULL*	
authorization	varchar(6)		Yes	*NULL*	
trans_id	int(11)		Yes	*NULL*	
trans_time	timestamp(14)		Yes	*NULL*	
reference	varchar(20)		Yes	*NULL*	

Figure 1 - Donations Table Design

One of the greatest things about the architecture of SQL databases is that the metadata (the data describing the database and its content), is stored in an SQL database. Simple queries with results like those shown in Figure 2 make it easy to introspect (literally "look at oneself") the tables in the database and the contents of those tables. Given that the metadata is available to programs in general and to PHP in particular, it is relatively easy to write programs to process database table structures in a very general fashion. Table names, fields within tables, data types of fields, use of fields as keys, and so forth are all available for processing.



```
+---------------+---------------+------+-----+---------+----------------+
| Field         | Type          | Null | Key | Default | Extra          |
+---------------+---------------+------+-----+---------+----------------+
| id            | int(11)       |      | PRI | NULL    | auto_increment |
| first_name    | varchar(50)   | YES  |     | NULL    |                |
| last_name     | varchar(50)   | YES  |     | NULL    |                |
| company_name  | varchar(50)   | YES  |     | NULL    |                |
| address       | varchar(60)   | YES  |     | NULL    |                |
| city          | varchar(40)   | YES  |     | NULL    |                |
| county        | varchar(50)   | YES  |     | NULL    |                |
| state         | char(2)       | YES  |     | NULL    |                |
| zip_code      | varchar(10)   | YES  |     | NULL    |                |
| phone         | varchar(15)   | YES  |     | NULL    |                |
| fax           | varchar(15)   | YES  |     | NULL    |                |
| email         | varchar(50)   | YES  |     | NULL    |                |
| occupation    | varchar(50)   | YES  |     | NULL    |                |
| employer      | varchar(50)   | YES  |     | NULL    |                |
| card_number   | varchar(4)    | YES  |     | NULL    |                |
| amount        | decimal(10,0) | YES  |     | NULL    |                |
| response      | varchar(10)   | YES  |     | NULL    |                |
| authorization | varchar(6)    | YES  |     | NULL    |                |
| trans_id      | int(11)       | YES  |     | NULL    |                |
| trans_time    | timestamp(14) | YES  |     | NULL    |                |
| reference     | varchar(20)   | YES  |     | NULL    |                |
+---------------+---------------+------+-----+---------+----------------+
```

Figure 2 - Table Description Query

Once I realized that this data was available, I quickly designed and wrote a simple PHP program to generate classes from the metadata of a MySQL table. A day later, I had a program, buildClass.php, which reads the metadata of a MySQL table in a database and emits a class derived from SQLData that provides the framework for manipulating data within a single table of a database. Example 1 is a partial listing of the generated class included here for discussion of the generated classes.

```
<?php

//
// Class: Example
// Table: example
// Database: APG
//
// Generated by buildClass.php, written by Dick Munroe
(munroe@csworks.com)
//

include_once("SQLData/class.SQLData.php") ;                    // (1)
include_once("SDD/class.SDD.php") ;

class Example extends SQLData                                  // (2)
{

    //
    // Private (or constant) varibles.
    //

    var $m__tableName = 'example' ;                            // (3)

    //
    // Constructor
    //

    function Example($_dataBase,                               //(4)
                    $_host="localhost",
                    $_login="",
                    $_password="")
    {
        $this->SQLData($this->m__tableName, $_dataBase, $_host,
$_login, $_password) ;
    }

    //
    // Accessor Functions
    //

    function setId($theValue)                                  // (5)
    {
        $this->set('id', $theValue) ;
    }

    function getId()                                           // (6)
    {
        return $this->get('id') ;
    }

    function initId($theValue)                                 // (7)
    {
        $this->init('id', $theValue) ;
    }

    function un_setId()                                        // (8)
    {
      $this->un_set('id') ;
    }

    function is_setId()                                        // (9)
    {
      return $this->is_set('id') ;
    }

    //
    // Default update selector
    //

    function needUpdateSelector()                              // (10)
    {
        if (($this->is_setId()))
           return "where `id` = '" . $this->escape_string($this-
>getId()) .
                  "'" ;

        trigger_error("Internal Logic Error: No key data present",
                      E_USER_ERROR) ;
    }

    //
    // Insert function
    //

    function insert()                                          // (11)
    {
        $theReturnValue = parent::insert() ;
        if ($theReturnValue)
        {
            $this->initId($this->fetchLastInsertId()) ;
        }
        return $theReturnValue ;
    }

    //
    // Debugging Functions
    //

    function print_r()                                              // (12)
    {
      $sdd = new SDD() ;
      print($sdd->dump($this)) ;
    }
}

?>
```

Example 1- Automatically Generated Example Class

1. The various underlying components of the application generation, in particular the SQLData class from which all specific table classes are derived and the Structured Data Dumper class which is used. To make the code generated by the programs referred to here, SQLData and SDD must be installed in your PHP include path. See the PHP documentation for details.
2. Every table specific class is derived from SQLData, a class supporting generic table data storage and updating. Essentially, the table specific classes are convenience classes to make dealing with specific tables easier. Note that the first character of the table name is upper cased to make the class name.
3. This provides the binding between this class and the underlying MySQL table.
4. The constructor for the table-specific class. Since the table name is wired into the class, the remainder of the MySQL database access information must be provided when the class is instantiated. No data oriented constructors are provided, as most of the table-specific class data initialization occurs either as a side effect of accessing the table through the underlying SQLData interfaces or from interactions with a user through web forms.
5. set* member functions set the named field to a value and note that the value is now "dirty" and should be flushed to the database when the next update or insert operation is issued. A set* member function will be created for each field in the table metadata.
6. get* member functions get the named field value and return it to the caller. A get* member function will be created for each field in the table metadata.
7. init* member functions are the same as set* functions but the data is not set as "dirty" and will not be flushed to the database when the next update or insert operation is issued. An init* member function will be created for each field in the table metadata.
8. un_set* member functions delete data for a field from the underlying SQLData class. Once deleted, the field data is no longer considered in insert or update operations. An un_set* member function will be created for each field in the table metadata.
9. is_set* member functions are predicates returning true if the field has data in the underlying SQLData class. Data is stored for a field using either the set* or init* member functions. An is_set* member function will be created for each field in the table metadata.
10. The insert and update member functions (available using SQLData) both take an optional selector to indicate which record in the table should be modified. For properly designed tables with keys, it is generally possible to provide a set of default selectors, depending on which keys in the table currently have data associated with them, to be used when a selector is not

provided. The needUpdateSelector member function is overridden when possible to provide the default selectors.

11. For tables with indices that are auto_increment fields (see Figure 2), after an insert operation has succeeded the auto_increment fields are updated automatically by MySQL. The insert member function is overridden as necessary to make sure that the value of the auto_increment index is maintained when a new row is inserted into the table.
12. Last, but not least, debugging. The print_r member function dumps the content of the object (and its base objects) in a structured format that makes it easy (or at least possible) to see what's happening within the table specific object. If the execution environment is a web server, then the data is dumped in HTML format.

This simplified the job of developing the donations application enormously, and I began putting together the user interface that would use the mechanically generated table class.

Data Collection and Validation

Very shortly, it became clear that the user interface had a problem similar to that of the table classes; that is, there were one (or more) tables for which forms had to be generated. Furthermore, the data had to be syntactically and semantically verified, any errors correct by further interactions with the user, and the data stored in the appropriate tables in the MySQL database. Since I had a pattern for generating executable "stuff" from MySQL metadata, I decided to see what, if anything could be done to generate rough drafts of the forms necessary to collect data to be stored in the necessary table.

To keep the collected data from corrupting the tables (“garbage-in, garbage-out” applies here) the collected data would have to be verified. I partitioned the data verification into two distinct types:

- Syntactic
- Semantic

Verification could be done in two places, the client or the server. To improve responsiveness, syntactic validation using JavaScript (now ECMAScript) is done at the client (the web browser) and semantic validation at the server.

Examples of syntactic validation are:

- Zip codes must contain only digits and "-"s and can be either of Zip (01234) or Zip+4 (01234-5678) format.
- Telephone numbers have to be digits and must be 10 digits long.
- Required files must be non-blank.

Occasionally there would be one field that implied that others would no longer be optional and this sort of thing I defined to belong to syntactic verification.

I defined semantic validation to be “checking that data is meaningful in a given context.” Examples are:

- Is the county or state name real?
- Does the city exist?
- Is the credit card number valid?

Basically, semantic verification answers the question "does the data represent reality" in the context of the application. In many cases, semantic validation can be built in by restricting the input values to a particular range, such as forcing the user to select from a list of values like country or state names.

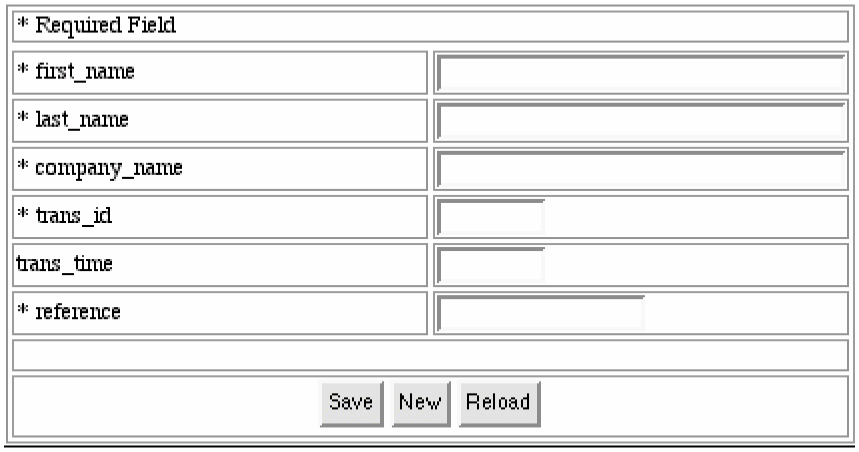
There were three additional pieces to be examined for automation opportunities:

- User Interface

- Syntactic Validation
- Semantic Validation

User Interface

As shown in Figure 2, the table metadata has the basics, field name, data type, size of data and whether data is required (not null). Given this information, it was easy to write another program much like the buildClass.php program to construct a simple user interface using HTML forms to display and capture data and to link that interface to the client-side syntactic validation framework and to the server-side semantic validation.



* Required Field	
* first_name	
* last_name	
* company_name	
* trans_id	
trans_time	
* reference	

Figure 3 - Generated User Interface

Figure 3 has been edited for space reasons and shows part of the user interface generated for the Example table.

It is important to remember that my goal for the FDP was not to produce a completely polished and fully functional web application solely from MySQL metadata. It was only to produce something that, with not much effort, could be turned into a "completely polished" and fully functional web application.

```
<link rel="stylesheet" type="text/css"
href="syntacticValidationFramework.css" />               (1)
<script type="text/javascript"
src="syntacticValidationFramework.js"></script>          (1)
<script type="text/javascript"
 src="form.Example.js"></script>                         (1)

<form name=data method=post action="process.Example.php"
 onSubmit="return validate(this) ;">                     (2)
<table name=dataTable id=dataTable border=1 cols=2>
        <tr>
            <td colspan=2>
                * Required Field
            </td>
        </tr>

        <tr id=errorRow>                                 (3)
        </tr>
        <tr>
            <td><span id="spanFirst_name" class="inline">*
                 first_name</span></td>
            <td>
<input name="first_name" id="first_name" type=text size=40
maxlength=50 required="first_name is required" validate="return
```

```
validateFirst_name(what)" value="">            </td>
               (4)
        </tr>
<tr>
            <td><span id="spanTrans_time"
                 class="inline">trans_time</span></td>
            <td>
<input name="trans_time" id="trans_time" type=text size=10
maxlength=10 required="" validate="return validateTrans_time(what)"
value="">            </td>
   (5)

        </tr>
<tr>
            <td align=center colspan=2>                  (6)
                <button name=save id=save type=submit>Save</button>
                <button name=new id=new type=button
onClick="window.location.search='?action=new'">New</button>
                <button name=reload id=reload type=button
onClick="window.location.search='?action=reload'">Reload</button>

            </td>
        </tr
    </table>
</form>
```

Example 2 - Generated HTML

A quick look at the HTML generated by buildForm.php (Example 2) is instructive.

1. These are the hooks to the "syntactic validation framework" discussed more fully in the next section. Unless buildSyntacticValidation.php has been run before buildForm.php the JavaScript components of the syntactic validation will not be included.
2. This is where the syntactic validation framework is actually invoked. When a Submit button is clicked, the onSubmit code is called. The form is not actually submitted unless or until all syntactic errors have been corrected. If buildSyntacticValidation.php has not been run, the onSubmit code is omitted and no syntactic validation will be done.
3. Another hook for the syntactic validation framework. This row is where the error information (if any) will be displayed by the framework.
4. This is a typical required form field. Validation for the field is provided by the validation attribute. Note the required attribute for required fields is non-null.
5. This is a typical optional form field. Validation is still required (the validation may always succeed, of course) and is done for consistency. Note that the required attribute is the null string for optional fields.
6. The action portion of the user interface. **Save** causes the captured data to be (optionally) syntactically validated and sent to the server for further processing. **New** clears the form and starts the data capture process over. **Reload** discards any data changes and starts the data capture process over.

Syntactic Validation

The syntactic validation requires JavaScript (now known as ECMAScript). Most modern browsers support JavaScript so this requirement isn't all that restrictive, only leaving out text-only browsers such as lynx or links. It relies on the Document Object Model (DOM), defined by the World Wide Web Consortium. Unfortunately, Microsoft's Internet Explorer is not particularly compliant with the DOM. It was possible to design an adequate web-browser independent framework providing primitives to handle collection and displaying of error data, validation of required fields, and a driver to validate a form upon submission.

Since my goal was to generate most, but not all, of a web-enabled application automatically, the validation hooks had to be associated with the individual fields rather than generated monolithically. Further, not every form would necessarily need syntactic validation.

This is handled by running (or not running) buildSyntacticValidation.php. This program generates the JavaScript routines to do the syntactic validation for each field in the form. If buildSyntacticValidation.php is not run, when the form is generated by buildForm.php no syntactic validation will occur at the time the form is submitted.

The syntactic validation framework is provided in a separate JavaScript file, syntacticValidationFramework.js. Understanding the details of the DOM that allow the framework to work is left as an exercise to the reader.

```
<form name=data method=post action="process.Example.php"
 onSubmit="return validate(this) ;">
```

Example 3 - Syntactic Validation Hook

Example 3 shows the hook between the user interface (form) and the syntactic validation framework. The onSubmit action is taken when a submit button is clicked. A pointer to the form object requiring validation is passed to the framework, along with the contents of the form used to actually determine what validation needs to be done. If the validation framework returns false, the form's data is not sent to the server.

```
<input
    name="first_name"                                     (1)
    id="first_name"                                       (2)
    type=text
    size=40
    maxlength=50
    required="first_name is required"                     (3)
    validate="return validateFirst_name(what)"            (4)
    value="">
```

Example 4 - Field Validation Details

The Document Object Model requires all attributes of a tag to be represented. This means that the page designer can put anything into an arbitrarily named attribute. Each field to be validated must have an unique id, an indication if the field is required, and a pointer to the routine to be used to validate the field's contents.

Example 4 shows a typical input tag with the hooks for the syntactic validation framework.

1. The name of the field received by the server during semantic validation and processing.
2. The unique id of the field. By convention it is always the same as the name field. The DOM interface is most easily used if each tag has an id by which the field can be found.
3. If the field in the database must have a value, the required field must be non-null and must contain the message to be presented to the user should the contents of the field be omitted.
4. Invocation of the validation framework routine for the field.

```
function validateFirst_name(what)
{
    //
    // Validate first_name field value
    //

    if (!isRequired(what))
    {
        return false ;
    }

    return true ;
}
```

Example 5 - Template Validation Routine

BuildSyntacticValidation.php generates a JavaScript routine identical in all but name for each field in the user interface. The validation framework calls the validation routines with a pointer to the field to be validated. Each validation routine must return either false (validation failed) or true (validation succeeded) and prepare any error text to be displayed at the end of validation.

The current syntactic validation framework can easily be modified to fit a variety of error reporting and interaction styles without modification to any of the generation code. Or something completely different can be written to meet site-specific requirements. After all, the generating code is in the public domain.

Semantic Validation

The data hits the database in semantic validation. Assuming that the form passes syntactic validation when the user presses the Save (Submit) button, the contents of the form are sent in HTTP format to the server. For the purposes of the FDP, semantic validation basically took the data from the form and put it in the databases.

The semantic validation code is generated by buildProcessForm.php. The semantic validation does not happen by default. As generated by buildProcessForm.php data was either inserted into or updated in the database and then control returned to the user for further work. Any semantic validation was considered to be custom code and to be written by hand. This is consistent with my goal of generating most but not all of the application.

The Complete Process

At this point, we had a set of tools that allowed the FDP applications to follow a very well defined data driven software development process. All of these tools were either free software, shareware, or easily developed in-house. The tools and process were:

1. phpMyAdmin	Design the database schema for the application. Database design was frequently a group interactive event, starting on a blackboard and quickly moving to a web browser or terminal. Only one database design only took more than an hour from discussion to completed design.
2. buildDatabaseConf.php	Generate the configuration file containing the information necessary to access the MySQL database.
3. buildClass.php	Create the PHP class to make it easier to manipulate the data. All the PHP code generated by the build* programs use the generated class.
4. buildSyntacticValidation	Create the individual field validation routines for the syntactic validation framework.
5. buildForm	Create the user interface.
6. buildProcessForm	Create the PHP code to store/update data in the

	database.
7. Apache, FreeBSD	Install the generated application on the development webserver.
8. Netscape, Internet Explorer, MySQL	Test the user interface/database interaction.
9. emacs, UltraEdit, Putty, WinSCP	Modify the generated application to full function and integrate with the production FDP web site.

Using this process, you can go from the completion of a database design to a prototype application in a matter of minutes.

Conclusion

The techniques associated with automatic program generation are well understood and widely used in many application generators on any number of platforms. By spending a few days writing tools, I made it much faster and easier to turn out finished applications. Experience showed that about 80% of the finished application can be automatically generated. Finished applications, fully integrated with the production FDP web site, are frequently produced in a day, with the vast majority completed in less than two days.

In all likelihood, similar productivity levels can be achieved elsewhere with site-specific versions of the tools described here.

All code developed for the FDP discussed in this paper is available from www.phpclasses.org. You will have to join www.phpclasses.org in order to download the code, but the membership is free. PHP is available from www.php.net and MySQL from www.mysql.com. These can be built and run on UNIX, Linux, and Windows platforms. Cygwin is available from www.cygwin.com if you want a UNIX-like environment for your Windows PC. If you're going to do any serious MySQL database development phpMyAdmin is a must and can be downloaded from www.phpMyAdmin.net. All are free software.

Acknowledgements

I'd like to thank Chris Sands, the director of IT for the Florida Democratic Party, for permission to publish this work. Many thanks to all the folks who supported the FDP during the 2004 campaign, lots of nights wouldn't have been possible without the free Diet Coke and junk food. I'd also like to thank Manuel Lemos, the creator of www.phpclasses.org. My job at the FDP would have been substantially more difficult if his site didn't exist. Last, but definitely not least, thanks to the many unnamed developers who have put together so much fine software and put it into the public domain and in particular to those folks who have contributed to PHP, MySQL, Apache, Cygwin, and FreeBSD. Without it the work we did in Florida during the last presidential campaign would have been impossible.

For more information

If you need more information about the work described here, contact Dick Munroe at munroe@csworks.com