The Solaris Operating System on x86 Platforms

Crashdump Analysis
Operating System Internals

Table of Contents

1.Foreword..5
1.1.History of this document...5
1.2.About modifying this document...7

2.Introduction to x86 architectures...9
2.1.History and Evolution of the x86 architecture..9
2.2.Characteristics of x86..12
2.3.Marketeering – Naming the architecture...19

3.Assembly Language on x86 platforms...21
3.1.Generic Introduction to Assembly language..21
3.2.Assembly language on x86 platforms..25
3.3.x86 assembly on UNIX systems – calling conventions, ABI...........................32
3.4.Case study: Comparing x86 and SPARC assembly languages.......................42
3.5.The role of the stack..46
3.6.Odd things about the x86 instruction set...50
3.7.Examples of compiler-generated code on x86 platforms.............................51
3.8.Accessing data structures..65
3.9.Compiler help for debugging AMD64 code..73

4.Memory and Privilege Management on x86...77
4.1.The x86 protected mode – privilege management.....................................78
4.2.Traps, Interrupts, System Calls, Contexts...89
4.3.Virtual Memory Management on x86..91
4.4.Advanced System Programming Techniques on x86..................................101

5.Interrupt handling, Device Autoconfiguration.......................................103
5.1.Interrupt Handling and Interrupt Priority Management............................103
5.2.APIC and IOAPIC features...104

6.Solaris/x86 architecture...111
6.1.Kernel and user mode...112
6.2.Entering the Solaris/x86 kernel...113
6.3.Solaris/x86 VM architecture – x86 HAT layer...118
6.4.Virtual Memory Layout..121
6.5.Context switching..122
6.6.Supporting Multiple CPUs..124
6.7.isaexec – Creating 32/64bit-specific applications.....................................125

7.Solaris/x86 Crashdump Analysis..127
7.1.Debugging tools for core-/crashdump analysis...127
7.2.Troubleshooting system hangs on Solaris/x86...129
7.3.32bit kernel crashdump analysis – a well-known example...........................132
7.4.64bit kernel crashdump analysis – well known example.............................150
7.5.Another 64bit crashdump analysis example...163
7.6.AMD64 ABI – Backtracing without framepointers.....................................176

7.7.Examples on application coredump analysis on Solaris/x86....177
7.8.Advanced Debugging Topics....178

8.Lab Exercises....179
8.1.Introduction to x86 assembly language....179
8.2.Stacks and Stacktracing....183

9.References....185

10.License....187

1.Foreword

1.1.History of this document

This document didn't start out from nowhere, but neither has it originally been intended for publication in book form. But then, sometimes history takes unexpected paths ...

Shortly after Sun had revised the ill-begotten idea of “phasing out” Solaris for x86 platforms and started to ramp up a hardware product line with Intel CPUs in it, I was approached by the Service division within Sun about where they could get an introductory course about how to perform low-level troubleshooting – crashdump analysis – on the x86 platform. Information and trainings about troubleshooting on this level on SPARC platforms are widely available – starting with the famous “Panic!” book all the way to extensive classes offered by Sun Educational Services to participants both internal and external to Sun. That notwithstanding, we soon found out that no internal training about the low-level guts of Solaris/x86 did exist. Development engineers were usually both capable and encouraged to find out about the x86 platform on their own, and users outside of the engineering space were few and far between. So this project started as a slide set for teaching engineers who were familiar with SPARC assembly, Solaris Internals and some Crashdump Analysis the fundamentals of x86 assembly and Solaris on x86 platforms, strongly focusing on “what's similar” and “what's different” between the low-level Solaris kernel on SPARC and x86 platforms.

I was to a large degree surprised by the amount of interest this material generated internally, so it grew, as time allowed, into a multi-day internal course on Solaris/x86 internals and crashdump analysis. For a while, I came to spend a significant amount of time teaching this never-official “class” ...

Then came the work on Solaris 10 and the AMD64 port. The new “64bit x86” platform support brought changes in the ABI with it that severely surprised even experienced “x86 old-timers” and required a large amount of addition to the existing material, which at that time had grown into a braindump of semi-related slides. Revamping the Solaris hardware interface layer for both 32bit and 64bit on x86/AMD64 as well as the addition of new features like Dtrace or the Linux Application Environment made further modifications necessary.

In the end, StarOffice's limited ability to deal with presentations of 200+ slides eventually made it inevitable to drop the till-then adapted method of “add a slide as a new question comes up”.

Would I have to make the same choice again I'd probably have opted to install myself a TeX system, but I decided to give StarOffice another chance and turn this material into something closer to a book. How I regret not having used TeX to start with ... that'll teach me !

Over the course of the AMD64 port of Solaris this grew into essentially the current form, and when people started using the 64bit port internally a large amount of new questions and typical problems came up which I attempted to address. To say it upfront, while the assembly language on AMD64 will be immediately familiar to people who know about “classical” x86, the calling conventions used in 64bit machine code on AMD64 are so much different that in many aspects crashdump analysis on Solaris/AMD64 is closer to Solaris/SPARC than it is to Solaris/x86. But then it isn't ... well, I'm disgressing, go read the book.

Then the OpenSolaris project came. Initially, I had planned to publish this on launch day, but for many reasons this didn't work out at that time. So here it is – several months delayed, no longer completely covering the state of our internal and external (OpenSolaris) development releases. But it's finally reviewed, the crashdump analysis example dumps are made available, the StarOffice document has been cleaned up to only rely on freely available fonts + graphics.

Which means that *you* – yes, look into the mirror – are now supposed to work with this material, *and on it*. The whole document including all illustrations are now made available in editable form.

Please read the license attached to the end of the document.

- Yes, you can make modifications to this document.
- Yes, you can redistribute copies of this document in any form you see fit – you're in fact encouraged to do so.
- Yes, you're encouraged to contribute corrections or additions.

For all else legalese, see the appendix.

Enjoy – and never forget:

Don't panic !

(Shall I say green is my favourite color ?)

If you wish to contact the author, please send Email to:

Frank.Hofmann@sun.com

At this point in time, I cannot even start listing the number of people that have made this document possible. Given that it didn't start as a book project I've kept a lousy bibliography.

I'd like to both thank every unnamed contributor as well as excuse myself for not naming you.

Using the words of Isaac Newton:

"If I have seen further it is by Standing on the shoulder of giants."

You know who you are.

1.2.About modifying this document

StarOffice8 is used to edit this document, but (Beta) versions of OpenOffice 2.x should be able to access it as well.

The document uses the OpenSource *DejaVu* fonts which are a derivative of BitStream Vera. The difference between these two is that the DejaVu font family contains full **bold**/*italic*/***bolditalic***/condensed typefaces for Sans, Serif and Monospaced, while the original Bitstream Vera fonts only supply the full typeface set for Sans. Installing the DejaVu fonts is therefore a prerequisite to being able to edit this document and recreate the output as-is.

These fonts are available from http://dejavu.sourceforge.net

Other fonts than DejaVu should not be used. To simplify this, switch the StarOffice stylist tools to only show “Applied Styles”, and don't use any but these.

If you wish to contribute back changes/additions in plain text that's more than welcome. If you modify the StarOffice document itself, allow simple merge back by enabling the change recording facility in StarOffice. See the help functionality, on “Changes”.

Note that StarOffice's master text facility is somewhat dumb – it records full pathnames (instead of relative locations) for the subdocuments. When you open **`book.odm`** in StarOffice8, the Navigator will show you the list of subdocuments. Use the right mousebutton to request the context menu, and choose “Edit Link” to change the pathnames of the subdocuments to refer to the location where you unpacked the file set.

The same is true for embedding graphics. Not even the documented functionality (“link” to the illustrations instead of instantiate a copy for the document) is working. So be aware when you change some file under **`figures/`**, you might need to delete and reinsert it in the main document ...

I'll keep a pointer to the current version (StarOffice for editing / PDF for reading and printing) of this document on my blog:

http://blogs.sun.com/ambiguous/

And finally: These instructions should be better ...

2.Introduction to x86 architectures

2.1.History and Evolution of the x86 architecture

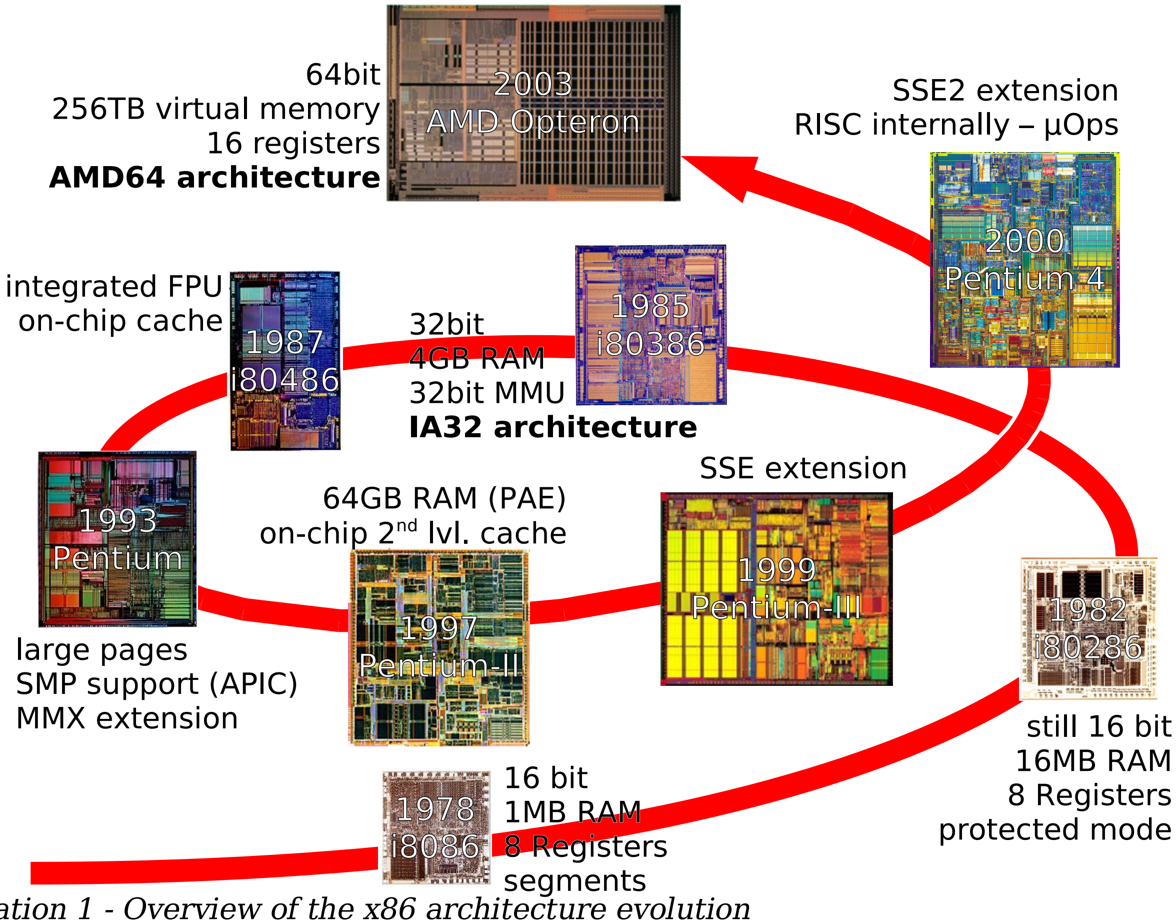


Illustration 1 - Overview of the x86 architecture evolution

The main driving force in development of the x86 processor family has always been to enhance existing functionality in such a way that full binary-level compatibility with previous x86 processors can be maintained. How important this is to Intel is best described in Intel's own words:

> ***One of the most important achievements of the IA-32 architecture is that the object code programs created for these processors starting in 1978 still execute on the latest processors in the IA-32 architecture family.***

Among all CPU architectures still available in current machines, only the IBM3xx mainframe architecture (first introduced in 1964 with the IBM360, still available in the IBM zSeries mainframes) has a longer history of unbroken binary backward compatibility. All current “x86-compatible” CPUs still support and implement the full feature set of the original member of the x86 family, the Intel 8086 CPU which was introduced in 1978.

This means: Executable programs from code originally written for the 8086 will run unmodified on any recent x86-compatible CPU such as Intel's Pentium-IV or AMD's Opteron processor. Yes, MSDOS 1.0 is quite likely to run on the very latest and greatest "PC-compatible", provided you can still find some single-sided 360kB 5¼" floppy drive which would allow you to boot it on that shiny new AMD Opteron workstation.

Backward compatibility of the x86 processor family goes way beyond what most other CPU architectures (including SPARC) have to offer. Sun Microsystem's Solaris/SPARC binary compatibility guarantee only ensures that *applications* (not operating systems or other low-level code) written on and for previous OS/hardware will continue to run on recent OS/hardware combinations, but it does not claim that old versions of the Solaris Operating Environment will run on processors that were yet unreleased at the time a specific release shipped. This is different on x86. New versions of x86 CPUs from whatever vendor run older operating systems just fine. Incompatibilities if at all rise from the lack of device driver support for newer integrated peripherals, but not from the newer CPU's inability to function like its predecessors.

Since introduction of the Intel i80386 in 1985 (!), most features of the x86 architecture have remained remarkably constant. SMP support (via APIC) and support for more than 4GB physical memory (via PAE) was added in the Pentium respectively to the PentiumPro processors; after that, only instruction set extensions (MMX, SSE) were added but no externally-visible changes were done to other core subsystems of x86.

From the point of view of Solaris/x86, it was never necessary therefore to have more than one kernel, **`/platform/i86pc/kernel/unix`**, for supporting the operating system on x86 processors. Put this in context and compare it with Solaris in SPARC: For the various SPARC generations (maximum number of architectures concurrently supported in Solaris 2.6: sun4, sun4c, sun4d, sun4m, sun4u, sun4u1), each time separate platform support was required. Even today, Solaris 9 delivers ten (!) different kernels for the various SPARC platforms, while Solaris 9 for x86 still has only one.

This strict insistence on binary compatibility with its predecessors obviously has disadvantages as well. The way how the i80386 introduced 32bit support in some areas looks illogical and counterintuitive, especially when comparing it with 32bit architectures that were designed for 32bit from their very beginnings. Some examples of this will be given later.

After releasing the i80386 32bit processor, Intel decided to keep future versions of x86-compatible (“IA32” in Intel's terms) CPUs on 32bit. Each generation became faster and added functionality, but the limitation to 32bit remained. In the early 1990s, this did not seem a problem because the major markets for x86 at that time (Microsoft DOS and Windows) were 16bit only anyway, and Intel's evolutionary path to 64bit had been layed out in the agreement with HP to co-develop a new 64bit architecture: IA64, then dubbed “Merced”, is today found in the Intel Itanium processors.

But IA64 has nothing to do with x86. The instruction sets have nothing in common and existing programs or operating systems written for 32bit x86 processors cannot run on machines with IA64/Itanium processors in it. The Itanium, though produced by Intel, is a genetic child of HP's PA-RISC architecture, but only a distant relative to Intel's own x86/IA32.

In addition to that, Intel and HP were late at delivering the IA64 CPU – very late.

So late that back in 2000, AMD stepped in and decided to extend the old x86 architecture another time – to 64bit. AMD had, with varying success, been building x86-compatible processors since the early 1980s and saw Intel's de-facto termination of x86 as a chance to extend its own market reach. The AMD64 (64bit x86) architecture was done in a way very similar to how Intel had done the i80386, and processors based on AMD64 (much unlike Itanium/IA64) are, in good old x86 tradition, fully binary backward compatible. Of course, actually using the new 64bit operating mode requires porting operating system and applications (like using 32bit on the i80386 did require at the time). But even when running a 64bit operating system does AMD64 provide a sandboxed 32bit environment to run existing applications in (again, like the i80386 which allowed the same for 16bit programs running on a 32bit OS). Therefore the AMD64 architecture offers much better investment protection than IA64 – which will *not* run existing 32bit operating systems or applications.

By the time the AMD Opteron 64bit processor became available, the Itanium, on the market for three years then, had seen very little adoption – while users and software vendors kept pushing ever harder on Intel to follow AMD's lead and provide 64bit capabilities in their x86 processor line as well. Intel resisted this for several years in order not to jeopardize the market for their Itanium processors but eventually gave in and cloned AMD64. For obvious reasons Intel doesn't call their 64bit-capable x86 processors “AMD64-compatible” but uses the term EM64T (*Enhanced Memory 64bit Technology*) for the architecture and IA32e for the 64bit instruction set extension. Intel CPUs with EM64T are compatible to AMD64 – which Intel confirms in the FAQ for the 64bit Extension Technology.

http://www.intel.com/technology/64bitextensions/faq.htm notes that:

> ***Q9****:* Is it possible to write software that will run on Intel's processors with Intel® EM64T, and AMD's 64-bit capable processors?
>
> ***A9****: Yes, in most cases. Even though the hardware microarchitecture for each company's processor is different, the operating system and software ported to one processor will likely run on the other processor due to the close similarity of the instruction set architectures.*

How the future of x86 will look remains to be seen. But the x86 architecture, with more than 25 years of age, has far surpassed the success of all other (non-embedded) processor architectures ever developed. With 64bit extensions that have rejuvenated x86, and x86-compatible processors with 64bit capabilities becoming commonplace now, this is unlikely to change in the near future.

2.2.Characteristics of x86

There are two factors responsible for the main characteristics of the machine instruction set for what is commonly termed “x86 architecture”:

- The long history of x86 has left its mark on the instruction set.
  x86 machine code carries a huge legacy of (mis-)features from the time when the architecture was still 16bit only, and in parts even from pre-x86 8bit days (in the form of limited compatibility with the Intel 8008).
- The need to introduce new capabilities without breaking binary compatibility has lead to a lot of instruction set extensions that are optional, and whose presence needs to be detected by applications / operating systems that want to make use of them. In addition, x86 never was a vendor-locked-in architecture, even though Intel's decisions have dominated its evolution. Both operating systems and application code for x86 therefore needs to expend some efforts on determining which CPU by what vendor it runs on, and what instruction set extensions this CPU provides before it can make use of optimized code.
  This is fortunately much improved by AMD64 which establishes a new “64bit x86 baseline”.

In addition to that, x86 CPUs use the so-called *little endian* way of ordering data in memory. Endianness becomes very relevant once data needs to be exchanged between systems of differing architecture.

2.2.1.CISC and RISC

Back in the early days of CPU design in the 1970s and early 1980s, manufacturing technology did not allow for anything close to the complexity we have today. CPU designers then had to make tradeoffs, mostly between a feature-rich assembly language, but few registers and generally lower instruction throughput, and a feature-poor assembly language with many registers and faster execution for the simple instructions that there were.

The x86 architecture is the classical example of a so-called CISC processor. The term *CISC* stands for *Complex Instruction Set Computer*, and is used to describe a processor whose instruction set offers single, dedicated CPU instructions for possibly very involved tasks. Philosophically, the ultimate design goal for a CISC processor is to achieve a 1:1 match between CPU instructions and instructions in a high-level programming language.

CISC is almost a requirement for CPUs which maintain full backward compatibility such as the x86 family. Adding functionality to an existing architecture always means adding instructions and complexity. A pure evolutionary CPU development as Intel has done it therefore almost necessitates a CISC architecture.

All in all, Intel's latest instruction set reference needs two volumes and more than 1000 pages to describe all x86 instructions supported by the latest x86 CPUs by Intel. For comparison - the sparcv9 architecture reference manual only has 106 pages describing all sparcv9 assembly instructions.

Given the focus on instruction functionality vs. versatility, CISC architectures tend to have features like:

- many special-purpose instructions.
  An example on x86 would be two separate instructions for comparison – the generic **CMP** instruction and the **TEST** instruction which will only check for equality or zeroness.

- the ability to modify a memory location directly, without the need to load its contents into a register first.
  This is done to offset the lack of registers – the idea is that if destination or source of an operation can be memory, less registers are needed.
- instructions with varying length.
  This is both due to the fact that CISC architectures usually allow to embed (large) constants into the instruction, and because feature additions over time have required the introduction of longer opcodes (instruction encodings).
  Another consequence of this is that there are few gaps (undefined or illegal opcodes) in the instruction set. As we will see, to an x86 CPU random data makes up for a decodeable instruction stream !
- few general-purpose registers.
  Historically there had to be a tradeoff between using the space on the CPU die to provide more registers or more-capable instructions. CISC CPU designers chose to do the latter, and it often proved difficult to extend the register set even after manufacturing technologies would have allowed for it. The x86 architecture lived with only eight registers, until AMD designing the 64bit mode finally took the chance and extended the register set to 16.

The x86 architecture is the single major remaining CISC architecture out there today. Most other CPU architectures on the market today, whether SPARC, PowerPC, ARM or (to a degree) even IA64, have gone the other way – RISC.

SPARC assembly source	***binary machine code***	***disassembler output***
`func:`	`section     .text`	`section     .text`
`    tst   %i0`	`  0:  80 90 00 18`	`     tst  %i0`
`    orcc  %g0, %i0, %g0`	`  4:  80 90 00 18`	`     tst  %i0`
`    set   1234, %i0`	`  8:  b0 10 24 d2`	`     mov  0x4d2, %i0`
`    or    %g0, 1234, %i0`	`  c:  b0 10 24 d2`	`     mov  0x4d2, %i0`
`    cmp   %i0, %i1`	` 10:  80 a6 00 19`	`     cmp  %i0, %i1`
`    subcc %i0, %i1, %g0`	` 14:  80 a6 00 19`	`     cmp  %i0, %i1`
`    clr   %i0`	` 18:  b0 10 00 00`	`     clr  %i0`
`    or    %g0, %g0, %i0`	` 1c:  b0 10 00 00`	`     clr  %i0`
`    mov   %i1, %i0`	` 20:  b0 10 00 19`	`     mov  %i1, %i0`
`    or    %g0, %i1, %i0`	` 24:  b0 10 00 19`	`     mov  %i1, %i0`
`.size func,.-func`		

Illustration 1 - machine code example on RISC, synthetic instructions

SPARC and all its incarnations are a classical example of *RISC* (*Reduced Instruction Set Computer*), and share many generic features with other RISC architectures:

- Lots and lots of CPU registers are available. For example, SPARC provides at least 32 general-purpose registers (internally hundreds, via register windows).
- To modify data in memory, one must load it into a register, modify the register contents and store the register back into memory. This is called a *load-store architecture*.
- RISC instructions usually have a fixed instruction size. All SPARC instructions, for example, are 32bit. RISC Instruction sets are rather *designed* than *evolved*.
- Instructions often are multi-purpose. A RISC CPU, for example, may not have separate instructions for subtracting values, comparing values or testing values for zero – instead, typically, “SUB” will be used but the result (apart from condition bits) be ignored. See the SPARC assembly code example above.
- Instructions tend to be simple. If a RISC CPU offers complex instructions at all, they

are usually completed by help of the operating system - instructions leading to complex system activity will trap and require software help to finish.

Unlike CISC, the focus for RISC is on raw execution power - the more instructions per unit of time a CPU can process the faster it will be in the end. Executing a dozen simple instructions as fast as theoretically possible often proves to provide better throughput than executing a single, slow instruction to achieve the same effect. RISC originally was invented to allow for simpler CPU designs running at higher clock speed.

RISC pays for this by often requiring more instructions to achieve an equivalent result as CISC gets with just one or two instructions:

x86 assembly	***binary code***	***SPARC assembly***	***binary code***
`movq`	`48 b8`	`sethi %hi(0x12345400), %o1`	`13 04 8d 15`
` $0x123456789abcdef0,`	` f0 de bc 9a`	`xor   %o1, -0x279, %o1`	`92 1a 7d 87`
` %rax`	` 78 56 34 12`	`sethi %hi(0x65432000), %o0`	`11 19 50 c8`
`addq %rax,var`	`48 01 04 25`	`xor   %o0, -0x110, %o0`	`90 1a 3e f0`
	` XX XX XX XX`	`sllx  %o1, 32, %o1`	`93 2a 70 20`
		`xor   %o1, %o0, %o0`	`90 1a 40 08`
		`sethi %hi(var), %o1`	`13 0X XX XX`
		`or    %o1, %lo(var), %o1`	`92 12 6X XX`
		`ldx   [%o1], %o2`	`d4 5a 40 00`
		`addc  %o0, %o2, %o2`	`94 42 00 0a`
		`stx   %o2, [%o1]`	`d4 72 40 00`

*Illustration 2 - RISC & CISC: Adding a 64bit constant to a global variable “***`var`***”*

Today, most arguments in the CISC vs. RISC debate have become obsoleted by technical progress.

Since the introduction of Intel's Pentium-IV and AMD's Athlon, modern x86 processors internally "recompile" x86 instructions into RISC instruction sets. Intel calls this µ-ops, while AMD uses the term ROPs (RISC ops) openly. These RISC execution engines in x86 CPUs are not exposed to the user - the step of decoding/compiling x86 instructions into the underlying micro-ops is done by an additional layer of hardware in the instruction decoder part of these CPUs.

Likewise, RISC CPUs over time have added complex instructions such as hardware multiply/divide which had to be done purely in software in early RISC designs. Additionally, instruction set extensions like the Visual Instruction Set (VIS) on UltraSPARC or AltiVec on PowerPC allow for DSP-like (SIMD) functionality just like MMX/SSE do on x86.

So what is a modern x86 CPU then ? CISC or RISC ?

The answer is:

Both. It is a CISC CPU, but to perform best, one has to program it like a RISC CPU.

For example, AMD in their *Software Optimization Guide for AMD Athlon64 and AMD Opteron Processors* explains it like this:

> *The AMD64 instruction set is complex; instructions have variable-length encodings and many perform multiple primitive operations. AMD Athlon 64 and AMD Opteron processors do not execute these complex instructions directly, but, instead, decode them internally into simpler fixed-length instructions called* ***macro-ops****. Processor schedulers subsequently break down macro-ops into sequences of even simpler instructions called* ***micro-ops****, each of which specifies a single primitive operation.*

and a little later:

> *Instructions are classified according to how they are decoded by the processor. There are three types of instructions:*
>
> | ***Instruction Type*** | ***Description*** |
> |---|---|
> | *DirectPath Single* | *A relatively common instruction that the processor decodes directly into one macro-op in hardware.* |
> | *DirectPath Double* | *A relatively common instruction that the processor decodes directly into two macroops in hardware.* |
> | *VectorPath* | *A sophisticated or less common instruction that the processor decodes into one or more [...] macro-ops [...]* |
>
> *.*

and finally:

> *Use DirectPath instructions rather than VectorPath instructions.*

In short:

Bypass the CISC runtime translation layer to get best performance out of the underlaying RISC execution engine.

Similar notes can be found in the respective manuals for Intel's Pentium IV CPU family and later.

2.2.2.Endianness

The x86 CPU family is traditionally *Little Endian*. What does this mean ?

The topic of how bytes that form multi-byte (or, for that matter, multi-bit) entities should be ordered in the past used to have almost religious traits. This is the reason why the technical term for memory byte ordering, *Endianness*, was taken from *Gulliver's Travels* by Jonathan Swift and refers to the holy war between the two empires of Lilliput and Blefuscu about the question which *end* eggs are to be opened at first.

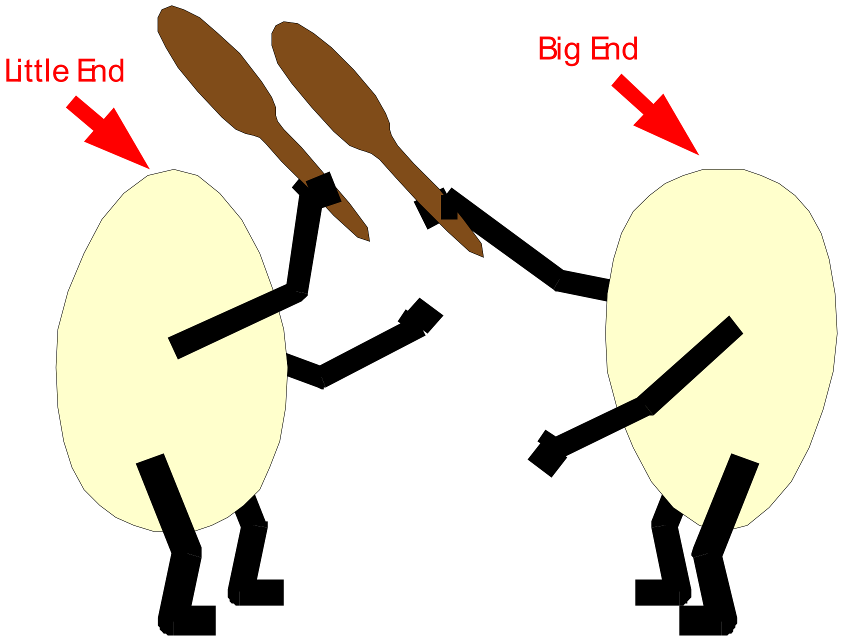


Illustration 4 - On the origin of the term "Endianness"

The original reference which coined the term seems to be a posting by David Conen in his famous essay *"On holy wars and a plea for peace"*, which dates from the 1st of April 1980 and became a classic on that subject after it was published by the IEEE computing magazine in 1981. The article is also known under the reference number IEN-137.

Data ordering in little endian mode	***Data ordering in big endian mode***

`utsname+0x303?s`
`utsname+0x303:   snv_24`
`> utsname+303?J`
`utsname+0x303:   736e765f32340000`
`> utsname+303?2X`
`utsname+0x303:   32340000    736e765f`
`> utsname+303?4x`
`utsname+0x303:     0   3234   765f   736e`
`> utsname+303?8B`
`utsname+0x303:    0 0 34 32 5f 76 6e 73`
	`utsname+0x303?s`
	`utsname+0x303:   snv_24`
	`> utsname+0x303?J`
	`utsname+0x303:   736e765f32340000`
	`> utsname+0x303?XX`
	`utsname+0x303:   736e765f 32340000`
	`> utsname+0x303?4x`
	`utsname+0x303:   736e 765f 3234 0`
	`> utsname+0x303?8B`
	`utsname+0x303:   73 6e 76 5f 32 34   0   0`

When a processor accesses a multi-byte data type (i.e. C types **`short`**, **`int`**, **`long`**, **`long long`**) from memory in a single operation, it will make an implicit assumption what comes first – the *most significant byte* (MSB) or the *least significant byte* (LSB). These terms are used interchangeably with Endianness,

- LSB (least significant byte first)   : Little Endian

- MSB (most significant byte first) : Big Endian

As there is endianness on byte level, there's also endianness on bit level, i.e. regarding the ordering of bits within a byte. But this poses less problems than byte ordering, because apart from serial protocols little data exchange is done on bit-level, and fortunately mixed-endian CPUs that used little endian for bits and big endian for bytes or vice versa (yuck – like ancient greek written in a mode called “βουστροφεδον”, “like the ox plows” - one line from left-to-right, and the next right-to-left) are no longer on the market. Today, big-endian CPUs use big endian for both bit and byte ordering, and likewise little-endian CPUs.

To a CPU, reading numbers from memory, aka ordering bytes within a word, is like reading a text to humans – words are made up from characters, and you read them from left-to-right – unless, of course, you're reading Arabic or Hebrew texts, or traditional chinese, where you read them from right-to-left. There is no inherent advantage or disadvantage to do it either way, and what's supposed to be the correct way of doing it depends on the CPU/language you use. But a consequence is that what feels natural to one seems very odd to the other.

So accessing data the big-endian way is like reading left-to-right, while little-endian is like reading right-to-left and therefore may look odd. But if the output is formatted, it becomes clear again:

Little Endian: Right-aligned pointers	***Big Endian: Left-aligned pointers***
`utsname+101/J`	`utsname+101/J`
`utsname+0x101:  6361626863746168`	`utsname+0x101:  6361626863746168`
`> utsname+101/X`	`> utsname+101/X`
`utsname+0x101:          63746168`	`utsname+0x101:  63616268`
`> utsname+101/x`	`> utsname+101/x`
`utsname+0x101:              6168`	`utsname+0x101:  6361`
`> utsname+101/B`	`> utsname+101/B`
`utsname+0x101:                68`	`utsname+0x101:  63`

The difference in endianness between e.g. x86 (little endian) and SPARC (big endian) becomes relevant as soon as data is exchanged between two machines of differing endianness. Even within the same system this can happen, in the case the CPU and a peripheral device use different endianness but share memory. Whenever file contents, shared memory or network packets are exchanged between two parties that use different endianness, a common storage format must be agreed on, or a method to swap endianness must be found.

Examples how to deal with endianness are:

Network Byte Ordering.

On creating network packet contents, the sender is supposed to use the host-to-network interfaces, **`htonl()`** etc., to convert data to the network byte ordering, while the receiver shall use the corresponding network-to-host functions, **`ntohl()`** etc., to decode network data into its native format.

Network byte ordering is big-endian, but it is unportable to program based on that assumption. It's also unnecessary – on big-endian machines, the interfaces for host/network byteorder conversion will do nothing – they'll simply pass through their input. Compiler optimizers eliminate these calls on big-endian systems.

See manpage **`byteorder(3SOCKET)`**.

Remote Procedure Calls (RPC).

Passing RPC arguments between two systems requires an endian-agnostic data representation. This is called XDR (exchangeable data representation), and a library is supplied that programs can use to convert a huge variety of basic data types into XDR representation. XDR is a generalization of network byte ordering to arbitrary data types. See manpage **xdr(3NSL).**

DMA Memory Access by device drivers.

Peripheral devices and the main CPU in a machine may access memory with differing endianness. When writing a device driver for such a device, the programmer therefore needs an interface to specify to the host operating system that a given device is big- or little-endian. Depending on whether device and host use the same or a different byte ordering, data to be transferred to or from that device must be converted into the proper byte order. Under Solaris, the DDI interface set provides routines to request byte swapping to be done by the framework. See manpages:
ddi_device_acc_attr(9S), **ddi_dma_mem_alloc(9S)** and **ddi_dma_sync(9S)**.

2.3.Marketeering – Naming the architecture

Illustration 5 - Pieter Brügel, Tower of Babel

The number of trademarked and non-trademarked terms applied to “x86 CPUs” and software that runs on the “x86 platform” is legend, and marketing departments everywhere keep adding to it.

Naming the architecture is truly the Babel of the computing industry.

- The term “x86” and derivatives of that is generic (not trademarked), and commonly used to describe all architectures (by various vendors) that were in one way or the other “derived” from the original Intel 8086 microprocessor, and have a high degree of compatibility with Intel CPUs.
- The same applies to “PC compatible” - though that includes more than just a CPU that is “x86 compatible”. The original IBM PC/AT (trademarked terms, again) had, in addition to the i8086 CPU, a set of standard hardware/peripherals whose presence can be assumed on “compatibles”. Later, Microsoft, Intel and other hardware vendors devised updated “PC XX” standards to list a set of hardware/bus interfaces available by default on “modern” systems.
- Intel uses the trademarked terms “Intel Architecture”, and more specifically “32bit Intel Architecture” (IA32). Intel always had more than one CPU architecture in their portfolio (e.g. today the Itanium/IA64, in the past the i860 RISC, even before that the i437) so “Intel Architecture” alone doesn't mean anything technically. IA32, on the other hand, is the term applied to the instruction set/feature set of Intel CPUs whose ancestor is the 8086 – in short, IA32 is “x86 by Intel”.
- The CPU names i8086, i80286, i80386, i80486, Pentium, ... are Intel trademarks.
- UNIX platforms have traditionally shortened these to i86, i286, i386 (with and without the leading 'i'). “386” is particularly frequent as the first 32bit version. “386” and variants thereof is found all over:

```
$ file ls
ls: ELF 32-bit LSB executable 80386 Version 1,
 dynamically linked, stripped
```

```
$ uname -a
SunOS hatchback 5.10.1 onnv-work i86pc i386 i86pc
```

```
$ isainfo
amd64 i386
```

We also find it, for example, in the ELF format architecture name:

```
 <sys/elf.h>: #define EM_386       3    /* Intel 80386 */
```

It's also present as the conditional-compile definition for 32bit x86:

```
<sys/isa_defs.h>:
[ ... ]
/*
 * The feature test macro __i386 is generic for all processors implementing
 * the Intel 386 instruction set or a superset of it. Specifically, this
 * includes all members of the 386, 486, and Pentium family of processors.
 */
#elif defined(__i386) || defined(i386)
/*
 * Make sure that the ANSI-C "politically correct" symbol is defined.
 */
#if !defined(__i386)
#define __i386
#endif
[ ... ]
```

Intel itself never used the terms “i586”, “i686” (with or without the 'i') or similar, but other CPU vendors (like AMD or Cyrix) did, and e.g. the GNU gcc compiler recognizes **-m586** and similar as hint to optimize code for post-486 processors.

The confusion about names doesn't get better with the extension to 64bit.

- The 64bit extension was created and first specified by AMD. AMD called this “x86_64” during development (and the term is still used as the architecture name on Linux), and “AMD64” on release.
  In fact, both are found as the ELF architecture name:

  ```
  <sys/elf.h>:
  #define EM_AMD64        62        /* AMDs x86-64 architecture */
  #define EM_X86_64       EM_AMD64  /* (compatibility) */
  ```

  AMD64 applies to the instruction set (x86 including 64bit extensions).

- The AMD Opteron and Athlon64 are CPUs by AMD implementing the AMD64 architecture.

- Intel for obvious reasons does *not* use the term “AMD64”. Since “IA64” is already given to the (x86-incompatible) Itanium architecture, Intel has created two new names of its own instead:

  - EM64T (Extended Memory 64bit technology)

  - IA32e

  The first is applied to processors by Intel that are “AMD64 compatible”, while the second (which is very uncommon) is used in Intel's architecture reference manual to describe the 64bit x86 instruction set (extension).

- Microsoft and Sun, for example, chose to use the term “x64” when talking about the 64bit x86 architecture, resp. their operating systems supporting it. In that context, “x86” means 32bit-x86, while “x64” means 64bit-x86.

In this document, the term “x86” is used wherever possible, with a specific note “32bit”, “32bit mode”, “64bit” etc. as appropriate.

3.Assembly Language on x86 platforms

From “The Tao of Programming”:

The Tao gave birth to machine language.
Machine language gave birth to the assembler.

The assembler gave birth to the compiler.
Now there are ten thousand languages.

3.1.Generic Introduction to Assembly language

Programming languages, however they are structured, tend to implement a common set of minimum functionality. Programming languages usually have features like:

- *instructions*, i.e. operations to modify and query "state"
- "state" (*operands*/*variables*/*data*) that instructions operate on
- modularization (the ability to substructure both program and data into smaller reusable units of execution/access, termed *functions*/*structures*)

Assembly language of course supplies all of these. The purpose of this section is to explain how constructs used in x86 assembly language implement these basic building blocks. Since this manual is not supposed to replace introductory tutorials on either programming in general nor machine-level programming as such, no attempt will be made to explain things like "what is an instruction", "what is an expression". Minimum familiarity with programming is assumed.

To understand assembly language programs (or disassembled compiled code), look at the above list of language building blocks again in more detail.

3.1.1.Instructions

Assembly language uses *mnemonics* (human-readable transcript of the actual binary machine code) for instructions. The following classes are usually supplied:

1. arithmetic/logical instructions. Anything that actually modifies data (aka performs an operation) falls under this category. Examples are addition, multiplication, and other numerical operations.
2. comparisons and conditionals to query state and change the flow of execution depending on that state. A typical example would be a "check if lower than" or a "branch if equal" instruction.
3. Load/Store operations for data transfer
4. function subroutine support, aka call/ret instructions (instruction transfer)

How readable the assembly language for a specific processor is depends somewhat on the choice of the CPU vendor how to name the instructions.

Intel for the x86 CPU family has used plain english terms (or at worst simple abbreviations) for assembly instruction names. A typical example would be the name of the instruction that calculates the sum of two operands: "**`ADD`**". At worst, an abbreviation as "**`MOVNTQA`**" (Move non-temporal quadword aligned) can occur, but in most cases x86 assembly instruction names are descriptive.

3.1.2.Operands, Variables and Data

To understand the concepts used in assembly language for accessing data, we have to examine more closely what data can be. More precisely, what the *scope* (visibility) of a particular item is.

One possible way how data can be classified in a hierarchical way would be:

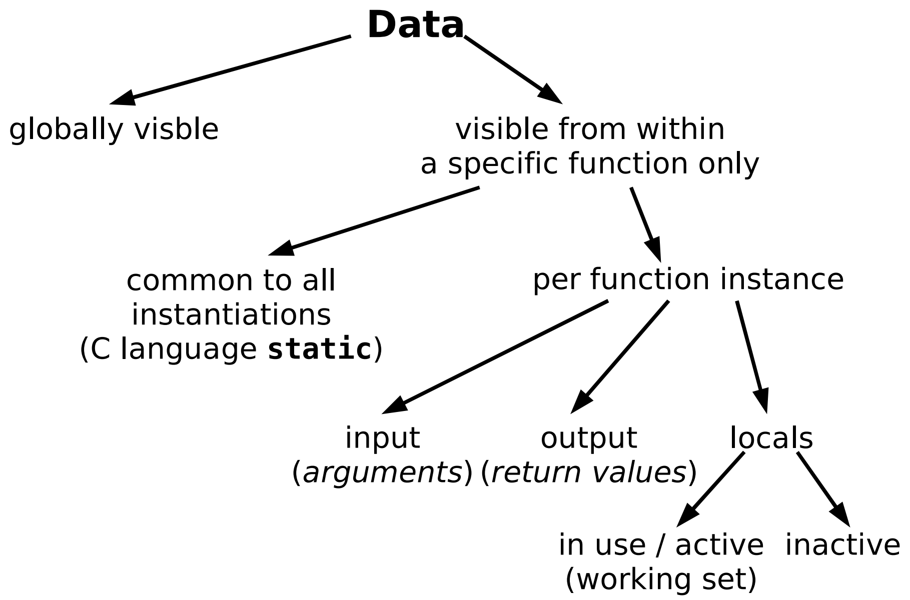

Data
globally visble
visible from within a specific function only
common to all instantiations (C language **static**)
per function instance
input (*arguments*)
output (*return values*)
locals
in use / active (working set)
inactive

Illustration 1 - Data Namespace based on scope of access

This is not the only possible subclassification of "data", of course, but the above scheme has the advantage that it maps very well to some of the concepts inherent to assembly language.

From the point of view of currently executing machine code, data can be considered to be “closer” and “further” away.

- Data that can be seen from any code within the current program is called *global*. Global data is persistent, it will continue to exist even if the specific piece of code that happened to be using it has been completed.
  - The C programming language knows a specific subtype of global data that is called **`static`**. Static data in C is not visible to every code from the current program but only to code from the same sourcefile, or to all instantiations (calls) of a given function. C static also is persistent.
- Any other data in use by the program is temporary and only lives as long as the current function is executing. Such data is recreated/reinitialized each time a given function is run, and different functions operate on different sets of data. This is generically called *local data*. It is usually subclassed further into:
  - Function input: *Arguments*
  - Function output: *return value(s)*
  - Other non-persistent data in use by the function: *local variables*
- Structured programming languages have finer-grained blocks of execution than functions. Consider, for example, a loop within a function. It uses data, though in most cases not all of the data that this function is operating on. Instead, it uses only a subset of that. This subset of currently-in-use data is called the *working set*.
  For optimal performance, a method is desired to access data from the working set in

as fast a way as possible.

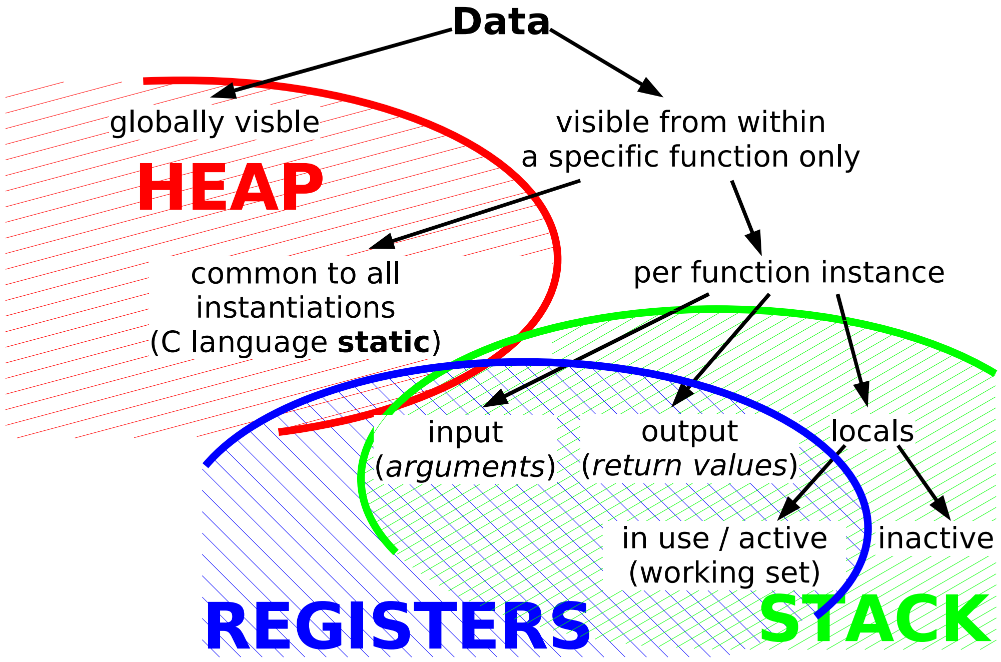


Illustration 2 - Machine-Language concepts for heap, stack and registers

In terms of machine-level architecture, data as classified above therefore falls into three big groups:

1. Global, persistent data. This is the *Heap*.
2. Temporary data which lives as long as the function that uses it is executing. This is usually called the *Stack*.
3. Data that makes up the current working set. Most CPUs provide fast-access temporary storage for such data – a set of *Registers*.

3.1.3.Registers, the Stack and the Heap

A high-level programming language often does not inherently know the concept of *memory*. Where data is stored or how it is accessed is up to the internal implementation of the language and not usually exposed to the programmer. Even intermediate-level languages like "C" that supply language features for specifying *data locality* (C keywords **`extern`**/**`static`**/**`auto`**/**`register`**, pointers) don't usually specify how these features are implemented, but refer to "the architecture" to supply the backend. Assembly language is different here. Due to the tight binding between hardware features and assembly language, the programmer here has to know about the details regarding where data is stored, resp. consider the optimal place where to put operands at any given time. This is where the above diagram comes in handy.

Assembly language at least knows the distinction between *persistent* and *temporary* data – the *heap* and the *stack*. There are machines out there (the Java Virtual Machine, or Forth, for example) which implement nothing else, but most current processors provide hardware support for putting a *working set* of data into fast temporary storage – a set of *registers*.

CPU Registers are kind of a "Level 0 Cache" (and the existance of registers as a fast-access temporary data storage far preceeds the existance of CPU caches) within the CPU, and used to hold variables that are either frequently queried or being modified as part of a computation. In many CPUs, arithmetic operations require the presence of the operands within registers. CPU registers, provided enough of them are available, will be the place where the *working set* of variables for the current function is found.

But even modern CPUs created/designed at a time when space on the CPU die is aplenty, don't offer unlimited number of registers. On the contrary, registers are usually a scarce resource. This is where the *stack* comes in again - to serve as a backing store for local variables. By giving each function its own dedicated piece of memory specific to this instantiation (i.e. different for e.g. two CPUs calling the same code), a so-called *stack frame*, the function can "swap" its working set between stack (or heap) and registers.

Registers and/or the stack frame also serve for data-passing between nested function calls. By letting the frames of calling and called function overlap, arguments can be passed between functions or values returned.

Data that is not specific to one instantiation of a function call but shared between all calls to this function (a C **`static`**), or all calls to all functions (a global variable) will not end up in the stack but in a well-defined location in memory that every code knows about. This memory location is often called the *data segment* of the program, or the *heap*.

3.2.Assembly language on x86 platforms

3.2.1.Registers

The general-purpose x86 register set has evolved from the 8bit i8008 processor's **AH**/**AL** accumulator model via the eight 16bit registers of the i8086 processor, and their extension (hence the register name prefix '**E**') to 32bit in the i80386 and 64bit in the AMD Opteron. All registers are global, and 16/8bit register names are only alias names for lower bits of the 32bit register. This is called *register aliasing*. In 32bit mode, x86 processors implement the following general-purpose registers:

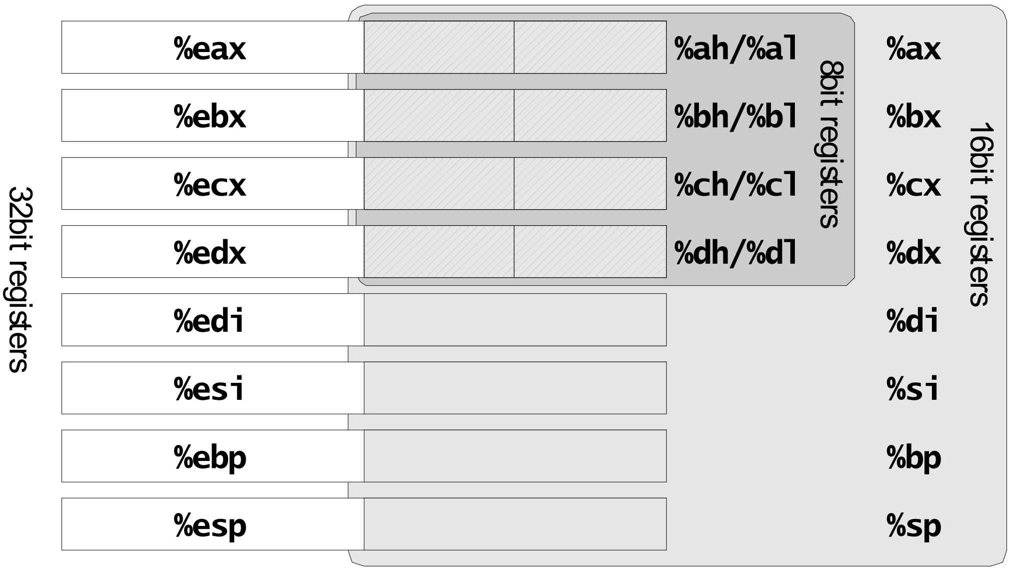


Illustration 3 - Register set (integer registers) on x86 architectures in 32bit mode

Overall, x86 CPUs in 32bit mode have only eight global, general-purpose registers. They are shared between 32/16/8bit access:

- 32bit registers : **EAX**, **EBX**, **ECX**, **EDX**, **ESI**, **EDI**, **EBP**, **ESP**
- 16bit registers : **AX**, **BX**, **CX**, **DX**, **SI**, **DI**, **BP**, **SP**
  These registers cover bit 0..15 of the corresponding 32bit registers.
- 8bit registers : **AL**, **BL**, **CL**, **DL**, and **AH**, **BH**, **CH**, **DH**,
  These registers cover bits 0...7 (**.L**) or bits 8..15 (**.H**) of registers **EAX** ... **EDX**.

Processors in the x86 family supply many more registers than that, but none of these are general-purpose. Instead, specific instructions are required to make use of those. Commonly-seen special registers in x86 include:

- The processor state register(s): **EFLAGS**, **CR0**...**CR8**.
- The program counter (instruction pointer) register: **EIP**.
- Floating point and vector registers: **ST0**..**ST8**, **MM0**..**MM8**, **XMM0**..**XMM8**

Peculiar to the architecture is the concept of segmentation, which also is controlled via a special set of registers:

- Descriptor Table registers: **GDTR**, **LDTR**, **IDTR**
- Segment registers: **CS**, **DS**, **ES**, **FS**, **GS**, **SS**

Modern x86 CPUs supply hundreds of registers, all of them special-purpose. They are called *machine-specific registers*, or MSR, and control specific features of the given CPU. Please refer to the processor manuals from the respective CPU vendors.

In 64bit mode (AMD64 and EM64T processors), the general-purpose register set is twice as large as before, and access to 16/8bit “subregisters” has been unified:

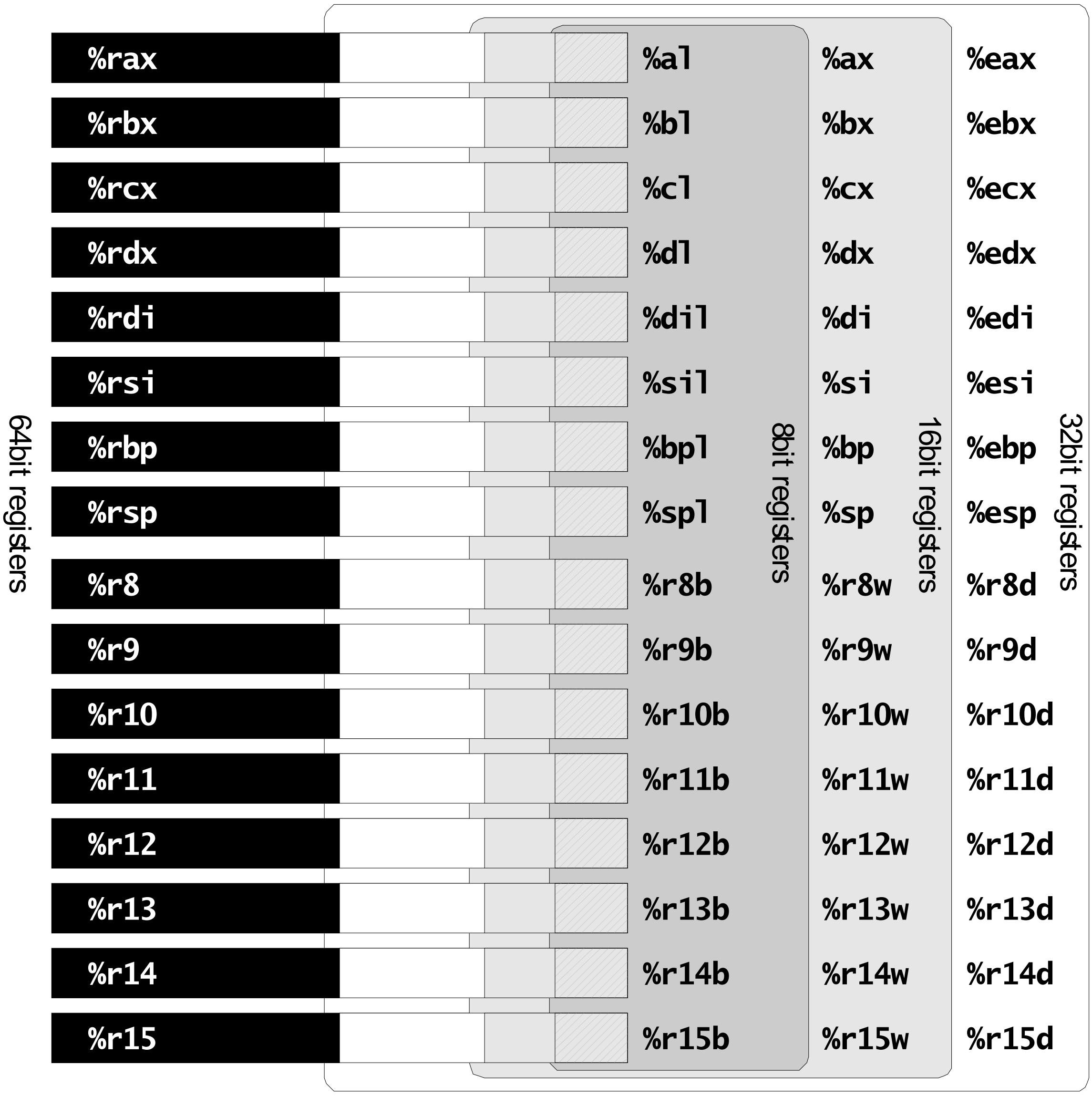


Illustration 4 - Register set (integer registers) on x86 architectures in 64bit mode

64bit mode retains register aliasing but makes it uniform. In addition to that, the number of general-purpose registers (and the number of `XMM` vector registers) has been doubled. In 64bit mode, the CPU provides:

- 16 64bit registers : **RAX**, **RBX**, **RCX**, **RDX**, **RDI**, **RSI**, **RBP**, **RSP** and **R8**..**R15**.
- 16 32bit registers : **EAX**, **EBX**, **ECX**, **EDX**, **EDI**, **ESI**, **EBP**, **ESP** and **R8D**..**R15D**. These registers map bits 0..31 of the corresponding 64bit register.
- 16 16bit registers : **AX**, **BX**, **CX**, **DX**, **DI**, **SI**, **BP**, **SP** and **R8W**..**R15W**. These registers map bits 0..15 of the corresponding 32/64bit register.
- 16 8bit registers : **AL**, **BL**, **CL**, **DL**, **DIL**, **SIL**, **BPL**, **SPL** and **R8B**..**R15B**.

These registers map bits 0..7 of the corresponding 16/32/64bit register.

In 64bit mode, the “highbyte” registers **AH**..**DH** are deprecated; they still are available but their use is no longer suggested for 64bit code.

The 64bit x86 register set is uniform – all registers can be used in the same way, i.e. all of them have 8/16/32bit “subregisters”. That doesn't mean all of them are equally efficient, though. The x86 instruction set has “optimized machine opcodes” for some arithmetic operations that put their result into **%eax**/**%rax**, for example. Likewise, the 64bit extensions encode the use of **%r8**..**%r15** via an additional byte in the instruction stream, so the use of the “classical” registers vs. the “new” registers creates more compact binary code. Please refer to the CPU vendors' optimization guidelines for instructions on how to optimally use the register set if you intend to write assembly code for 64bit x86 platforms manually.

64bit mode also has the **FLAGS** register (**RFLAGS**), and the 64bit program counter **RIP**, which is made explicitly available for PC-relative addressing, a feature not available in 32bit code.

Register aliasing requires rules that specify how the high bits of the 16/32/64bit register are handled if an instruction operates explicitly on a 32/16/8bit register:

- A 8bit operation on **.L** does not affect bits 8..31 (i.e. the upper bits in the **.X** and **E..** registers). Operating on **.H**, bits 0..7 and 16..31 are unaffected.
  Bits 32...63 of the 64bit **R..** register are cleared.
- A 16bit operation does not affect bits 16..31 (i.e. the upper bits in **E..**/**R..D**).
  Bits 32...63 of the 64bit **R..** register are cleared.
- A 32bit operation clears bits 32..63 of the 64bit **R...** register.

In other words, if operating in 64bit mode, all operations that are not explicitly 64bit will zero extend their result to 64bit. The advantage of doing this is in preserving the semantics of all existing 32bit operations. For example, a 32bit addition will overflow after 32bit and set status register bits to indicate this condition, instead of silently wrapping around to 64bit and preventing proper detection of the 32bit overflow.

3.2.2.Addressing Modes

Accessing memory is possible either:

- *Direct*, supplying an absolute 32/64bit value as address
- *Register indirect*, using the value contained in a register as address
- *Indirect with offset*, using the contents of a register as the base address and a (no larger than 32bit) constant as additional offset
- *Indirect with index and scale*, using a register as base address of an array, a second register as index into that array and a scale factor of 1, 2, 4 or 8 for that register to specify the size of the elements in the array.
- *Indirect with offset, index and scale*. Same as before, except that now the start address of the array will be the sum of base register and offset. This allows e.g. to efficiently access arrays that are themselves members of larger data structures.
- *instruction pointer relative with offset*. This is only available in 64bit mode and allows for efficient position-independent code.

As a summary, memory access on x86 systems is done by calculating the address implicitly using the following formula:

$$\text{memory location} = \mathbf{\texttt{offset}} + \begin{pmatrix} \texttt{\%rax} \\ \texttt{\%rbx} \\ \texttt{\%rcx} \\ \texttt{\%rdx} \\ \texttt{\%rsi} \\ \texttt{\%rdi} \\ \texttt{\%rbp} \\ \texttt{\%rsp} \\ \texttt{\%r8} \\ \vdots \\ \texttt{\%r15} \end{pmatrix} + \begin{pmatrix} 1 \\ 2 \\ 4 \\ 8 \end{pmatrix} \times \begin{pmatrix} \texttt{\%rax} \\ \texttt{\%rbx} \\ \texttt{\%rcx} \\ \texttt{\%rdx} \\ \texttt{\%rsi} \\ \texttt{\%rdi} \\ \texttt{\%rbp} \\ \texttt{\%rsp} \\ \texttt{\%r8} \\ \vdots \\ \texttt{\%r15} \end{pmatrix}$$

instruction-pointer-relative (64bit only):

$$\text{memory location} = \mathbf{\texttt{offset}} + \begin{pmatrix} \texttt{\%rip} \end{pmatrix}$$

Illustration 5 - Summary of x86 addressing modes

Any parts are optional. In 32bit mode, only the 32bit registers **%eax**...**%esp** can be used, of course.

The stack is special on x86, and the architecture has explicit support for accessing stack memory – via **PUSH**/**POP** instructions.

Pushing something onto the stack will decrement **%esp**/**%rsp** by the size of the operand and put the value of the operand into the memory location that **%esp**/**%rsp** points at then.

Popping something off the stack takes the value the **%esp**/**%rsp** points at, and then increments **%esp**/**%rsp** by the size of the operand.

3.2.3.x86 assembly syntax

On most processors, the assembly language (i.e. the human-readable mnemonics) has been created by the CPU vendor. This of course also applies to x86, but the story doesn't end there. On x86 systems, there are *two dialects* of assembly language:

- *Intel Syntax*
- *AT&T Syntax*

For non-UNIX assembly programmers, Intel's official assembly language syntax will be used. But on UNIX systems, the situation is traditionally reversed. After Intel released the 80386 processor with its 32bit capabilities, AT&T's UNIX System Laboratories were one of the first operating system vendors to create a 32bit operating system for it. Since there was no existing 32bit-x86 market in 1985, there were also no readily-available development toolchains (compiler, assembler, linker) for them to use, so this had to be written, in the case of the assembler from scratch. Legend has it that AT&T developers looked at Intel's assembly language specification and were horrified by the ambiguities in the syntax, and the strong dissimilarity of Intel's assembly language syntax to those of other CPUs that UNIX had been ported to before.

So AT&T devised their own assembly language syntax for x86 platforms, which is standard for x86 assembly on UNIX and UNIX-like systems.

On binary level, there's of course only one x86 machine language. AT&T and Intel Syntax are just a different way of making the machine language human-readable. As an analogy, consider writing the same chinese-language text with chinese characters and in the pinyin style using latin characters – striking differences, but yet it's chinese. Fortunately, the differences between AT&T syntax and Intel syntax are smaller than that.

Some simple examples illustrate differences between Intel and AT&T syntax very well:

Operation	***AT&T Syntax***	***Intel Syntax***
Move data from memory into a register	`movb  address, %ah`	`MOV   AH, [ address ]`
	`movw  address, %ax`	`MOV   AX, [ address ]`
	`movl  address, %eax`	`MOV   EAX, [ address ]`
	`movq  address, %rax`	`MOV   RAX, [ address ]`
More Addressing: Direct, Indirect, Indexed	`movl  address, %edi`	`MOV   EDI, [ address ]`
	`movb  -0x20(%ebp), %dl`	`MOV   DL, [ EBP – 0x20 ]`
	`movq  (%rax,%rcx), %r12`	`MOV   R12, [ RAX + RCX ]`
	`movl  (%edx,%esi,4), %edx`	`MOV   EDX, [ EDX + 4 * ESI ]`
	`movw  0x8(%ebp,%ecx), %si`	`MOV   SI, [ EBP + ECX + 0x8 ]`
	`movw  $0, 0x20(%ebx)`	`MOV WORD PTR [ EBX + 0x20 ], 0`
	`movl  %r12d, 12345(%rip)`	`MOV   [ RIP + 12345 ], R12D`
Use of constants	`addb  $10, %r15b`	`ADD   R15B, 10`
	`subl  $1234, -0x10(%rbp)`	`SUB INT PTR [ RBP – 0x10 ], 1234`
	`orl   $0x10110, %ebx`	`OR    EBX, 0x10110`
	`xorl  $0xffffffff, %ecx`	`XOR   ECX, 0xffffffff`
	`movq  $0x123456789abcdef0, %rax`	`MOV   RAX, 0x123456789abcdef0`
	`andl  $0xfffffff0, %esp`	`AND   ESP, 0xfffffff0`
Arithmetics	`xorl  %eax, %eax`	`XOR   EAX, EAX`
	`addl  %ecx, %edx`	`ADD   EDX, ECX`
	`andl  $0x1000, %esi`	`AND   ESI, 0x1000`
	`andl  $0x10, -0x30(%ebp)`	`AND INT PTR [ EBP – 0x30 ], 0x10`

Operation	***AT&T Syntax***	***Intel Syntax***

	`imulq %rax, %rbx, %rcx`
	`orl   $0x400, %r13d`
	`addb  $123, globalvar`
	
	`leal  (%eax,%eax,4), %eax`
	`leal  (%eax,%ebx), %ecx`
		`IMUL  RCX, RAX, RBX`
		`OR    R13D, 0x400`
		`ADD BYTE PTR`
		`      [ globalvar ], 123`
		`LEA   EAX, [ EAX + 4 * EAX ]`
		`LEA   ECX, [ EAX + EBX ]`

Control transfer	`call  funcX`
	`call  *%ebx`
	`ret`
	`iret`
	`lcall $0x27,0`
	`jae   func+0x123`
		`CALL  func`
		`CALL  [ EBX ]`
		`RET`
		`RET FAR`
		`CALL FAR 0, 0x27`
		`JAE   func+0x123`

Special Instructions	`cmpxchgl %eax, (%ecx)`
	`cmpxchgq %rdx, (%r15)`
	
	`repz`
	`scasb`
	
	`pushal`
	`movl  %xmm5, (%eax,%ebx)`
	
	`lock`
	`orl   $0x800, (%rax)`
		`CMPXCHG   [ ECX ], EAX`
		`CMPXCHG8B [ R15 ], RDX`
		
		`REPZ CASB`
		
		`PUSHAD`
		`MOVD  [ EAX + EBX ], XMM5`
		
		`LOCK OR INT PTR`
		`      [ RAX ], 0x800`

In general, AT&T syntax has been designed to remove ambiguities that are inherent to Intel syntax. The differences can be summed up as follows:

- AT&T prefixes register names with '**%**' to avoid ambiguities with names of variables. Intel reserves the names of registers. If a variable uses a name that the next Intel CPU uses for a register – you're out of luck.
- AT&T prefixes constants (whether numerical values or symbols whose address is to be taken as a constant) with '**$**'.
  Intel does not specifically mark constants. Again, ambiguities between register names and variable names possible.
- AT&T orders operands source first, destination second, i.e. a from-to ordering.
  Intel uses destination first, source second. Operations are “do to <>: .”.<>
- AT&T suffixes instruction names with **b**, **w**, **l** or **q** to specify the operand size.
  Intel derives the operand size implicitly from the name of the target register. In cases where the target is memory, the **... PTR** syntax extension is used.
- AT&T uses a format **offset(%base,%index,scale)** for address declarations.
  Intel puts the formula in square brackets, **[BASE + SCALE * INDEX + OFFSET]**
- AT&T in assembly sourcecode (not in disassembler output, though) puts *instruction prefixes* onto separate lines.
  Intel requires instruction prefixes to directly preceed the instruction they apply to.
- AT&T and Intel name a small set of instructions differently, in cases where the size suffix in AT&T syntax makes a new instruction name unnecessary, for string instructions, or far calls/returns.
- AT&T by convention uses lowercase, and variable names are case-sensitive.
  Intel syntax uses capital letters for everything.

Keep the differences between AT&T and Intel syntax in mind if you're reading Intel's or AMD's architecture reference manuals – these all use Intel syntax. Especially the different argument ordering can be confusing at times.

For porting assembly sourcecode written in Intel syntax to AT&T syntax, the “shortcut” via assembling it using an Intel-syntax-aware assembler, and disassembling it with one that outputs AT&T assembly is suggested.

As far as this document is concerned, AT&T syntax will be used. This is the x86 assembly language found in the Solaris sourcecode, and the kind of disassembler output one gets when using debugging tools on the Solaris operating system.

3.3.x86 assembly on UNIX systems – calling conventions, ABI

The x86 architecture supplies instructions for stack- and framepointer maintenance and **`call`**/**`ret`** instructions for performing function calls. But this is not sufficient. From architectural constraints, no rules exist for:

- how does the caller *pass arguments* to the called function ?
- how does the called function *return results* to the caller ?
- What happens with contents of the (global) registers when doing a **`call`** ?

In addition to that, x86 in hardware only knows basic C data types **`char`**, **`short`**, **`int`**, (**`long`**) **`long`**, **`float`** and **`double`**. What about compound data types – arrays and structures ? How are these laid out in memory ?

Compiled code should better agree on a common set of rules for passing arguments and returning values, for register usage and data structure layout, or else linking code from a library and a user-supplied program together is going to break big-time. This is why operating systems define *standard calling conventions* which all dynamically-linked code on this platform must obey. This is called the *Application Binary Interface*, or short ABI.

The ABI usually is a big document that defines much more than just the standard calling conventions for binary code. Details on functions in the standard libraries, software packaging and installation rules or lists of software programs that are considered an essential part of the system are also in the ABI.

For UNIX systems derived from AT&T UNIX System V R4, like Solaris, the relevant document is the System V ABI. It contains:

- A generic (platform-independent) part listing things common to all platforms UNIX has been ported to
- A *platform supplement* part, chapter 3, that details the abovementioned platform-specific calling conventions and data layout rules for a specific architecture. When one talks about “the i386 UNIX ABI” or “the x86-64 UNIX ABI”, what's meant is the platform-specific chapter 3 ABI supplement for the given architecture.

To illustrate the calling conventions on x86 systems, we'll investigate compiler-generated assembly code for a simple C language program:

```
#include <strings.h>

/*
 * A compound data structure consisting of several primitive types
 */
typedef struct {
      char s_c;
      unsigned short s_us;
      long s_l;
      int s_i;
      char s_name[256];
      struc_t *s_next;
} struc_t;

/*
 * C “constructor” for struc_t
 */
int init_struc(
    struc_t *s,
    char i_c,
    unsigned short i_us,
    int i_i,
    long i_l,
    char *i_name,
    struc_t *i_next)
{
      s->s_c = i_c;
      s->s_us = i_us;
      s->s_i = i_i;
      s->s_l = i_l;
      s->s_next = i_next;
      strncpy(s->s_name,
          i_name,
          sizeof(s->s_name));

      return (1);
}
```

It depends on compiler and optimization how the assembly will look like in the end; the code below was created using the Sun Workshop compiler, version 10, on a system running Solaris 10. The ABI is different for 32bit and 64bit binaries, as will be shown.

3.3.1.The i386 UNIX ABI – 32bit x86

First, the 32bit assembly code created from the above:

function offset	***binary opcode***	***assembly***		***C sourcecode***
`init_struc`	`55`	`pushl`	`%ebp`	`... init_struct(`
`init_struc+0x1`	`8b ec`	`movl`	`%esp,%ebp`	`...`
`init_struc+0x3`	`83 e4 f0`	`andl`	`$0xfffffff0,%esp`	`) {`
`init_struc+0x6`	`8a 45 0c`	`movb`	`0xc(%ebp),%al`	
`init_struc+0x9`	`8b 4d 08`	`movl`	`0x8(%ebp),%ecx`	
`init_struc+0xc`	`88 01`	`movb`	`%al,(%ecx)`	`s->s_c = i_c;`
`init_struc+0xe`	`66 8b 45 10`	`movw`	`0x10(%ebp),%ax`	
`init_struc+0x12`	`66 89 41 02`	`movw`	`%ax,0x2(%ecx)`	`s->s_us = i_us;`
`init_struc+0x16`	`8b 45 14`	`movl`	`0x14(%ebp),%eax`	
`init_struc+0x19`	`89 41 08`	`movl`	`%eax,0x8(%ecx)`	`s->s_i = i_i;`
`init_struc+0x1c`	`8b 45 18`	`movl`	`0x18(%ebp),%eax`	
`init_struc+0x1f`	`89 41 04`	`movl`	`%eax,0x4(%ecx)`	`s->s_l = i_l;`
`init_struc+0x22`	`8b 45 20`	`movl`	`0x20(%ebp),%eax`	
`init_struc+0x25`	`89 81 0c 01 00 00`	`movl`	`%eax,0x10c(%ecx)`	`s->s_next = i_next;`
`init_struc+0x2b`	`68 00 01 00 00`	`pushl`	`$0x100`	`sizeof(s->s_name)`
`init_struc+0x30`	`ff 75 1c`	`pushl`	`0x1c(%ebp)`	`i_name`
`init_struc+0x33`	`83 c1 0c`	`addl`	`$0xc,%ecx`	`&s->s_name`
`init_struc+0x36`	`51`	`pushl`	`%ecx`	
`init_struc+0x37`	`e8 fc ff ff ff`	`call`	`<strncpy>`	`strncpy(...);`
`init_struc+0x3c`	`b8 01 00 00 00`	`movl`	`$0x1,%eax`	`return (1);`
`init_struc+0x41`	`8b e5`	`movl`	`%ebp,%esp`	
`init_struc+0x43`	`5d`	`popl`	`%ebp`	
`init_struc+0x44`	`c3`	`ret`		`}`

The disassembler output is color-coded in the table above to highlight specific areas:

- ***Green*** shows *function prologue* and *function epilogue*.
- ***Red*** shows how **`init_struc()`** acccesses its own arguments.
- ***Blue*** shows access to different members of **`struc_t`**.

This simple example therefore suffices to demonstrate the rules in the i386 UNIX ABI supplement.

How are arguments passed to a function ?
Where does a function find its own arguments ?

> Look at the call to **`strncpy()`**. Its arguments are **`pushl`**'ed to the stack, in reverse order, i.e. argN pushed first, arg0 last – immediately before the **`call`**.
>
> Likewise, look at the way **`init_struc()`** accesses its own arguments. This is the code marked ***red***. Its arguments are found at:
>
> | | | |
> |---|---|---|
> | **`+0x8(%ebp)`** | **`struc_t *s`** | argument 1 |
> | **`+0xc(%ebp)`** | **`char i_c`** | argument 2 |
> | **`+0x10(%ebp)`** | **`unsigned short i_us`** | argument 3 |
> | **`+0x14(%ebp)`** | **`int i_i`** | argument 4 |
> | **`+0x18(%ebp)`** | **`long i_l`** | argument 5 |
> | **`+0x1c(%ebp)`** | **`char *i_name`** | argument 6 |
> | **`+0x20(%ebp)`** | **`struc_t *i_next`** | argument 7 |
>
> All arguments are passed on the stack.
> A function locates its arguments based on its framepointer.
> Arguments, even if smaller than 4 bytes in size, are 4-byte aligned on the stack.

How does a function return a value ?

Clear enough – the function epilogue places '1' into register **%eax**.

How is data in a compound (C struct) laid out, i.e. what padding if any is used ?

We can see from the instructions marked in ***blue***, i.e. those that store the values passed as arguments into **init_struc()** into the actual **struc_t**, that the data structure is laid out so that members can be accessed *aligned at a multiple of their size*. After the leading **char s_c**, one byte of padding makes sure that the following **unsigned short s_us** starts at a 2-byte aligned address, and so on.

This is not mandated by the x86 architecture – as we can see e.g. from the instruction pointers, x86 has no generic problem with misaligned memory access.

So far, no thorough explanation of the ***green*** stuff, the function prologue/epilogue, has been given. We notice that it saves/restores the caller's framepointer and initializes the one for **init_struc()**, which is needed to make argument access via **+...(%ebp)** possible, of course, but there's more to it.

The purpose of the prologue becomes clear when one looks at a significantly more complicated function. Let's take such an example from the Solaris kernel, in the form of the ufs filesystem implementation of **VOP_GETPAGE()**. The disassembly starts with:

```
ufs_getpage:        pushl  %ebp
ufs_getpage+0x1:    movl   %esp,%ebp
ufs_getpage+0x3:    subl   $0x58,%esp
ufs_getpage+0x6:    andl   $0xfffffff8,%esp
ufs_getpage+0x9:    pushl  %ebx
ufs_getpage+0xa:    pushl  %esi
ufs_getpage+0xb:    pushl  %edi
```

and the function epilogue looks like this:

```
ufs_getpage+0x861:  movl   -0x48(%ebp),%eax
ufs_getpage+0x864:  popl   %edi
ufs_getpage+0x865:  popl   %esi
ufs_getpage+0x866:  popl   %ebx
ufs_getpage+0x867:  movl   %ebp,%esp
ufs_getpage+0x869:  popl   %ebp
ufs_getpage+0x86a:  ret
```

We see that in addition to stack reservation and framepointer initialization, the prologue also saves registers **%ebx**, **%esi** and **%edi** to the stack, while the function epilogue restores them before returning to the caller.

The complicated function obviously needs to do this because there is a rule that says “the value in these registers may not to change during **call**”. Which is precisely what the ABI does – it splits the x86 register set into:

- *Registers that belong to the caller* (nonvolatile registers). These are preserved during function calls, and the called function, if it uses them, has the obligation to restore their previous values before returning. The nonvolatile registers are **%esp**/**%ebp** (obviously) and **%ebx**, **%esi**, **%edi**.
- *Scratch registers*. These change their values during function calls – they belong to the called function, which is free to use them for whatever it wants (exception: it must put its return value into **%eax**). The scratch registers are **%eax**, **%ecx**, **%edx**.

This is more complicated than simpler rules like “all registers scratch” or “all registers preserved”. So what is the advantage of splitting the register set into *caller-owned* and *callee-owned* registers ?

It becomes clear if you consider what happens in simple functions that have no need to use all registers. There are two possibilities:

a. Complicated function calling a simple function.
   In this case, a rule that says “all registers scratch” (the caller cannot rely on any register contents after **`call`**) would be counterproductive. The simple function may well be able to do its task without overwriting registers of the caller.

b. Simple function calling a complicated function.
   This is the other extreme. A rule “called function must preserve every register” would now unneededly require the complicated function to save and later restore all registers – although its caller hasn't even used all of them.

The designers of the i386 UNIX ABI therefore considered it to be beneficial to go the middle way – simple functions can get away using only the scratch registers, while complicated functions will never have to save/restore more than the nonvolatile (local) registers.

3.3.2.The AMD64 UNIX ABI – 64bit x86

The 64bit UNIX ABI supplement for AMD64 has been created when the 64bit-x86 Linux port was done. There are significant differences wrt. to function calling and argument passing between 32bit x86 and 64bit x86, because the 64bit ABI attempts to exploit the new possibilities,

- 16 general-purpose registers
- fast SSE2 floating point available by default, including 16 XMM registers

to speed up function calling.

Under 64bit-x86, ***arguments are passed in registers***. Argument passing therefore becomes complicated – for floating point arguments, *very complicated*.

The binary code for the small example turns into the following 64bit machine code:

function offset	***binary opcode***	***assembly***	***C sourcecode***
`init_struc`	`55`	`pushq %rbp`	`int init_struc(`
`init_struc+0x1`	`48 8b ec`	`movq %rsp,%rbp`	`...) {`
`init_struc+0x4`	`40 88 37`	`movb %sil,(%rdi)`	`s->s_c = i_c;`
`init_struc+0x7`	`66 89 57 02`	`movw %dx,0x2(%rdi)`	`s->s_us = i_us;`
`init_struc+0xb`	`89 4f 10`	`movl %ecx,0x10(%rdi)`	`s->s_i = i_i;`
`init_struc+0xe`	`4c 89 47 08`	`movq %r8,0x8(%rdi)`	`s->s_l = i_l;`
`init_struc+0x12`	`4c 8b 45 10`	`movq 0x10(%rbp),%r8`	`i_next;`
`init_struc+0x16`	`4c 89 87 18 01 00 00`	`movq %r8,0x118(%rdi)`	`s->s_next = ...;`
`init_struc+0x1d`	`48 83 c7 14`	`addq $0x14,%rdi`	`&s->s_name`
`init_struc+0x21`	`49 8b f1`	`movq %r9,%rsi`	`i_name`
`init_struc+0x24`	`48 c7 c2 00 01 00 00`	`movq $0x100,%rdx`	`sizeof(s->s_name)`
`init_struc+0x2b`	`33 c0`	`xorl %eax,%eax`	
`init_struc+0x2d`	`e8 00 00 00 00`	`call <strncpy>`	
`init_struc+0x32`	`b8 01 00 00 00`	`movl $0x1,%eax`	`return(1);`
`init_struc+0x37`	`48 8b e5`	`movq %rbp,%rsp`	
`init_struc+0x3a`	`5d`	`popq %rbp`	
`init_struc+0x3b`	`c3`	`ret`	`}`

The same color coding as before is used:

- ***Green*** shows *function prologue* and *function epilogue*.
- ***Red*** shows how **`init_struc()`** acccesses its own arguments.
- ***Blue*** shows access to different members of **`struc_t`**.

It's surprising to see that the 64bit code is actually more compact than the 32bit version. This is due to the changed argument passing conventions – there is no need to explicitly load arguments from the stack into registers – they're already there.

First, what has not changed ?

- There's still a function pro- and epilogue which initializes/restores the framepointer for the function. Its role will also be to save/restore the nonvolatile registers.
- The return value is still the same register – now 64bit of course, **`%rax`**.
- Members of compound data structures are still being aligned at a multiple of their size. Since AMD64 is LP64, **`long`** is now 8 bytes and the **`s_i`** and following members therefore are shifted backwards.

But the only access to **`+...(%rbp)`** that we see is for argument 7. Args 1..6 are passed in registers instead:

`%rdi`	**`struc_t *s`**	argument 1

%sil	**char i_c**	argument 2
%dx	**unsigned short i_us**	argument 3
%ecx	**int i_i**	argument 4
%r8	**long i_l**	argument 5
%r9	**char *i_name**	argument 6
+0x10(%rbp)	**struc_t *i_next**	argument 7

This makes it difficult to retrieve arguments from call stacks – they will only be part of the call stack if the called function itself decides to preserve them. If not, since all registers are global, the arguments to a given function will be lost as soon as this function makes another call. Besides, before doing that **%rax** is cleared – which again is something that wasn't done in 32bit.

Like on 32bit x86, the 64bit ABI also specifies how registers are shared between caller and callee. Looking at prologue:

```
ufs_getpage:         pushq  %rbp
ufs_getpage+0x1:     movq   %rsp,%rbp
ufs_getpage+0x4:     pushq  %r15
ufs_getpage+0x6:     pushq  %r14
ufs_getpage+0x8:     pushq  %r13
ufs_getpage+0xa:     movq   %rsi,%r13
ufs_getpage+0xd:     pushq  %r12
ufs_getpage+0xf:     pushq  %rbx
ufs_getpage+0x10:    subq   $0x98,%rsp
```

and epilogue:

```
ufs_getpage+0x6b0:   movl   -0x78(%rbp),%eax
ufs_getpage+0x6b3:   addq   $0x98,%rsp
ufs_getpage+0x6ba:   popq   %rbx
ufs_getpage+0x6bb:   popq   %r12
ufs_getpage+0x6bd:   popq   %r13
ufs_getpage+0x6bf:   popq   %r14
ufs_getpage+0x6c1:   popq   %r15
ufs_getpage+0x6c3:   leave
ufs_getpage+0x6c4:   ret
```

of a complicated function again, we find how the register set is used in 64bit x86 on UNIX systems:

- **%rsp** and **%rbp** are used for the *stack*- and *framepointer*. Their value is preserved over function calls (obviously). If optimized code eliminates the use of a framepointer, **%rbp** becomes a n*onvolatile register* – as in 32bit.
- **%rdi**, **%rsi**, **%rdx**, **%rcx**, **%r8** and **%r9** (in that order) are *arguments*.
  A function that uses less than six arguments can use them as *scratch registers*.
- **%rbx** and **%r12** through to **%r15** are *nonvolatile*. They belong to the caller and retain their values when doing a function call. It's the called function's responsibility to save and restore them if it needs them.
- **%rax**, **%r10** and **%r11** are *scratch registers*. **%rax** contains a *return value*.

3.3.3.The AMD64 UNIX ABI and the register lifecycle

The big problem for post-mortem analysis on 64bit x86 is the argument passing conventions: If arguments are passed in global registers, then deeply-nested function call sequences will always have re-used the argument registers of some earlier callers in the sequence. Stacktraces in debuggers on 64bit x86 therefore usually do not display arguments. But the way registers are being used:

- **`%rbx`** and **`%r12`**..**`%r15`** are nonvolatile (*local*) – every user of these has the obligation to save/restore them for its caller
- **`%rsp`**/**`%rbp`** as stack- and framepointer
- **`%rdi`**, **`%rsi`**, **`%rdx`**, **`%rxd`**, **`%r8`** and **`%r9`** as argument registers – being re-used when the next function call is made,
- **`%rax`**, **`%r10`** and **`%r11`** as scratch registers

enforces a certain code style that all compiler-generated assembly code uses.

On entry, a function:

1. saves **`%rbp`** of its caller,
2. initializes its own **`%rbp`**,
3. allocates stackspace by explicitly subtracting a value from **`%rsp`**,
4. saves (to the stack) those of the nonvolatile registers **`%rbx`**, **`%r12`**..**`%r15`** that it plans to use,
5. moves those of its argument registers (**`rdi`**, **`%rsi`**, **`%rdx`**, **`%rxd`**, **`%r8`** and **`%r9`**) that it wants to use even after making function calls of its own into permanent places:
    - into one of the now-available nonvolatile registers (i.e. into its *local registers*)
    - into “hidden” local variables (i.e. into its stack).

On exit from the function, steps 1..4 are reversed, in order to restore both stack and nonvolatile registers for the caller.

This code that sets up (on entry) and tears down (before return) a stackframe for a function is called function *prologue* / *epilogue*. The x86 architecture does not supply a single CPU instruction that could be used to implement all of the above (SPARC does all of this implicitly via register window switches), so a sequence of simple instructions is used.

For performance, the generated code will only perform the necessary operations. All of the above is optional in case the given function doesn't require:

- all of the nonvolatile registers (it'll only save/restore those that it needs)
- stack space (it won't allocate stackspace then)
- a framepointer (it won't save/initialize/restore it then)

In addition to that, neither the AMD64 nor the i386 UNIX ABI specify the exact steps how prologue/epilogue code is supposed to implement the “tasklist” above. The instructions chosen (**`MOV`** vs. **`PUSH`**, for example), the sequence in which e.g. registers are saved, or whether registers are saved before or after stack allocation is done all depends on the compiler's choice and is subject to compiler optimization.

This leads to a multitude of different code sequences for prologue/epilogue.

Particularly the GNU C compiler is creating many different variants, as the following examples for function prologue code, all created by gcc, show:

“Dense Prologue”	***Mixing prologue and function code***	***Using `MOV` instead of `PUSH`***
`pushq  %rbp`	`pushq  %rbp`	`pushq  %rbp`
`movq   %rsp,%rbp`	`movq   %rsp,%rbp`	`movq   %rsp,%rbp`
`pushq  %r15`	`pushq  %r15`	`subq   $0x30,%rsp`
`pushq  %r14`	`movl   %r9d,%r15d`	`movq   %r13,-0x18(%rbp)`
`pushq  %r13`	`pushq  %r14`	`movq   %r14,-0x10(%rbp)`
`pushq  %r12`	`xorl   %r14d,%r14d`	`movq   %rsi,%r13`
`pushq  %rbx`	`pushq  %r13`	`movq   %rbx,-0x28(%rbp)`
`subq   $0xd8,%rsp`	`movq   %rsi,%r13`	`movq   %r12,-0x20(%rbp)`
`movq   %rdi,-0x40(%rbp)`	`pushq  %r12`	`movq   %rdi,%r14`
`movq   %rsi,-0x48(%rbp)`	`movl   %edx,%r12d`	`movq   %r15,-0x8(%rbp)`
`movl   %edx,-0x4c(%rbp)`	`pushq  %rbx`	*[...]*
`movq   %rcx,-0x58(%rbp)`	`subq   $0x28,%rsp`	
[...]	`cmpl   $0x1000,%r8d`	
	`movl   %edi,-0x3c(%rbp)`	
	`movq   %rcx,-0x48(%rbp)`	
	[...]	

The Sun Workshop 10 compiler, as of today, always creates a strict, dense prologue similar to the first gcc variant shown above. Yet, it's distinctly different in two things:

1. Workshop-compiled code allocates stackspace *before* saving the nonvolatile registers,
2. Workshop-compiled code saves nonvolatile registers in a different order than gcc.

Unlike gcc, Sun Workshop cc *never* uses **`MOV`** instructions to save the nonvolatile registers, and it also doesn't intersperse code from “further down” in the function with the prologue. Workshop-created prologue code is always similar to something like:

Typical 64bit prologue code created by Sun Workshop 10

```
pushq  %rbp
movq   %rsp,%rbp
subq   $0x8,%rsp
pushq  %r12
pushq  %r13
pushq  %r14
movq   %rsi,%r14
movq   %rdx,%r12
movq   %rcx,%r13
[ ... ]
```

There's similar variation in function epilogue code created by gcc, while, again, Sun Workshop 10 cc always uses a compact sequence of **`popq`** instructions.

It's left as an exercise to the reader to familiarize oneself with different styles for function epilogues.

This *register lifecycle*, with associated explicit function prologues and epilogues that implement it, is due to the register set being global (shared by everybody) and the resulting need to manage register use in a cooperative way. It's achieved by moving registers to the stack before using them as “locals”, and matching that restoring their values before returning from the function.

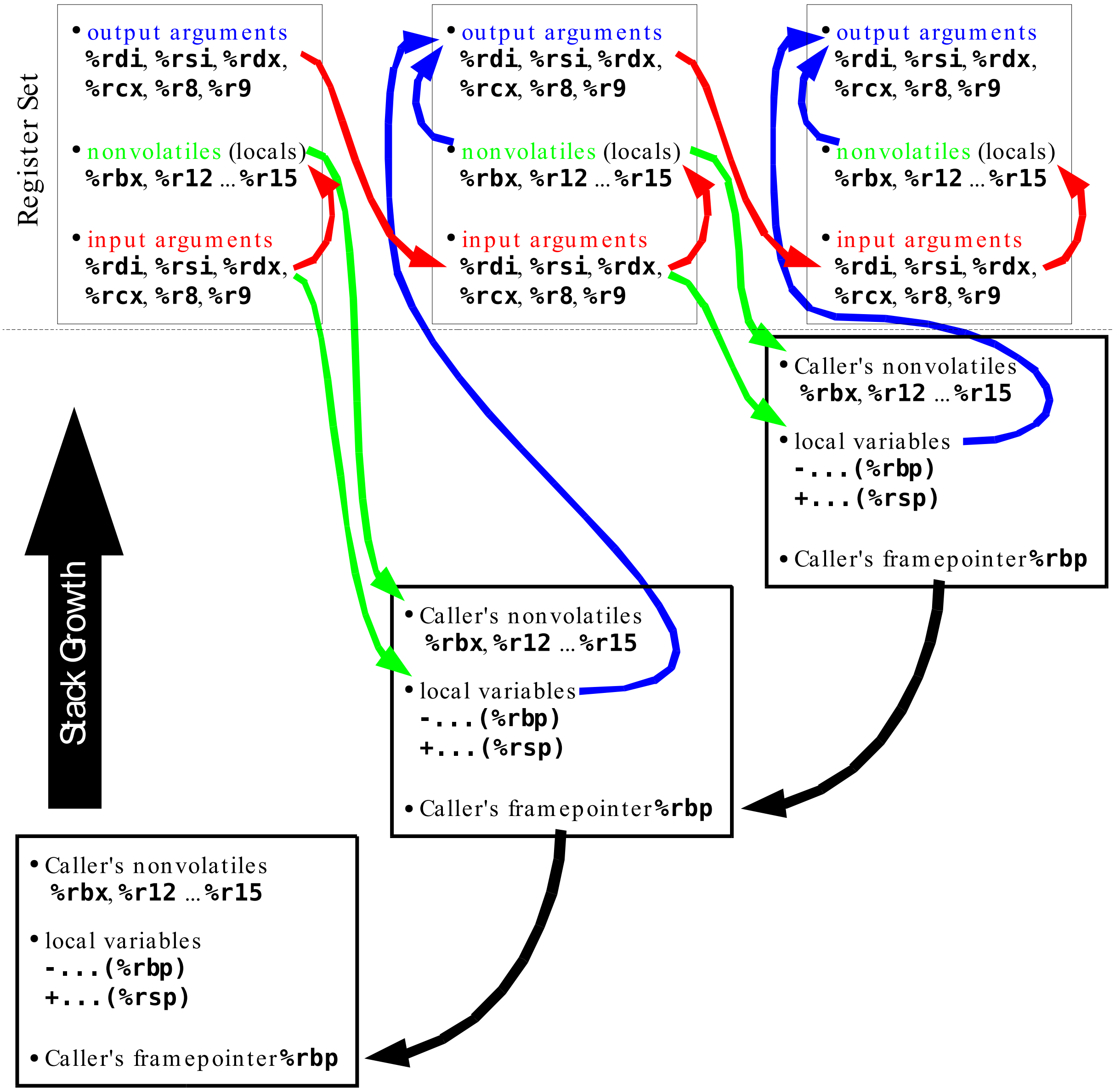


Illustration 6 - Register lifecycle – sharing registers, function calling and the stack

Asking “where did a specific register go to” (after entering a function) or “where did a specific register come from” (before calling another function) therefore allows us to track arguments even if register-based calling conventions are in effect, like on AMD64.

3.4.Case study: Comparing x86 and SPARC assembly languages

The two platforms currently supported by the Solaris operating system are SPARC and x86. As already shown, SPARC is a RISC architecture while x86 is CISC. This alone doesn't sum up all of the differences between the two assembly languages, of course:

Feature	***How x86 does it***	***How SPARC does it***
instruction length	varies from 1..16 bytes. The instruction pointer *usually* is misaligned.	constant 4 bytes. Misaligned program counters cause traps.
general-purpose registers	all registers are global. eight in 32bit, sixteen in 64bit. ABI determines register usage.	32 registers, 8 each are global/input/local/output. Register windows with the idea of output/input overlap are inherent in SPARC architecture.
dedicated /dev/null register	None. It's unnecessary. x86 has plenty of special-case instructions that make discarding a result unnecessary, and the ability to embed zero as a constant into any instruction.	Register **`%g0`** discards writes and always reads as zero. Frequently used if the result of a calculation is unnecessary and only the effect on condition codes is relevant, i.e. in synthetic instructions like **`cmp`**.
stack- and framepointer	The architecture mandates the use of register **`%esp`**/**`%rsp`** as stackpointer, and suggests using **`%ebp`**/**`%rbp`** as framepointer. Many instructions implicitly operate on the stack.	The register windowing mechanism would support the use of any pair **`%i.`**/**`%o.`** as stack/framepointer. By convention, **`%i6`**/**`%o6`** are used for frame/stackpointer.
arithmetic/logical instructions	x86 arithmetic / binary logical instructions are *destructive* – the second source operand will be overwritten with the result. x86 implements the C language **`a += b`** style.	SPARC arithmetic / binary logical instructions are usually *nondestructive*. All instructions take three registers – two for the source operands and one for the destination operand. In C: **`a = b + c`**.
memory operands	x86 allows one memory operand per (arithmetic) instruction. Either source or destination can be memory. Instructions modifying memory without involving a register are possible.	SPARC can only modify data in a register. To modify a word in memory, a *load-store cycle* (load it from memory to a register, change it in the register, store it back) is required.

implicit register usage	Many instructions implicitly modify certain registers:
	stack/framepointer changed by:
	`call`/**`ret`**
	`push`/**`pop`**
	`enter`/**`leave`**
	string instructions:
		SPARC instructions require an explicit destination register.
		In cases where this is not shown, the instruction is *synthetic* and the destination is explicit from that, usually **`%g0`** if the result is irrelevant. Notable exception is **`call`**/**`ret`**, where

Feature	*How x86 does it*	*How SPARC does it*
	take source/destination from the string source/dest registers **%esi**/**%edi** (32bit), **%rsi**/**%rdi** (64bit) counted loops, repeat prefixes: **%ecx**/**%rcx** holds counter value 64bit arithmetics in 32bit mode: **%edx**:**%edx** hold 64bit value.	**%o7**/**%i7** are used to hold the return address.
instruction-embedded constants	For direct addressing and initialization (**movq** instruction), constants up to 64bit are possible. All other instructions allow using 32bit constants directly.	Instruction-embedded constants cannot be larger than 13 bit (including the sign). For initialization, the **sethi** instruction allows to put a 19bit value into bits 31..13 of a register. Larger constants are constructed in a sequence of **sethi**/**or**.
atomic instructions	x86 knows *explicit atomicity* via the **lock** instruction prefix. Every instruction that allows a memory location as target operand can be made atomic.	SPARC has a dedicated set of instructions that are *implicitly atomic:* **cas**, **ldstub**, or **xchg**. Others need to be "emulated" using these.
illegal/undefined instructions	Generically, none. Random data translates into "meaningful" instruction sequences for x86. x86 knows "the" undefined instruction, **ud2**, a reserved opcode that triggers a trap.	Lots of. Wide ranges for the opcode bits in the 32bit instruction are not assigned a specific meaning (**illtrap** and **undef**) and cause traps.
Addressing	x86 addressing modes: Direct (64bit absolute address) Indirect (address in a register) Indirect with 8/16/32bit offset Indexed (address + scale*index) and combinations of those. In 64bit mode, it can also do: position-relative (PC plus offset)	SPARC addressing modes: Indirect Indirect with index Indirect with 13bit offset On SPARC, addressing is always indirect – an address must be in a register before **ld**/**st**. Indirect without (!) index is done using **%g0** as index ...

The total number of instructions on x86, given that it's CISC, is of course much higher than on SPARC, being a classical RISC architecture.

Apart from these architectural differences between the assembly languages, there are differences to the ABI wrt. to how function calling works – which, from the point of view of post-mortem analysis, mostly affect layout and contents of the stack. Comparing properties of the stack on x86 and SPARC:

stack detail	***How x86 does it***	***How SPARC does it***
function arguments	32bit x86 passes all arguments on the stack. A debugger can always retrieve them. 64bit x86 puts the first six args to **`%rdi`**, **`%rsi`**, **`%rdx`**, **`%rcx`**, **`%r8`**, **`%r9`** and spills arguments after that into the stack. args 1..6 can only be retrieved from the stack if the called function saves them to the stack explicitly.	SPARC puts the first six args into **`%o0`**..**`%o5`**, which after the **`save`** done by the called function are then found in **`%i0`**..**`%i5`**. A register window flush at a later time will write these into the stack and a debugger can retrieve them from there, but an input register could've been reused. Arguments past the sixth spill onto the stack, like 64bit x86.
return addresses	The x86 **`call`** instruction pushes the address of the instruction following it (i.e. the actual return address) onto the stack. The x86 **`ret`** instruction pops off the return address and jumps there.	The SPARC **`call`** instruction puts its own (!) address into **`%i7`**. The SPARC **`ret`** instruction is synthetic – it will evaluate to **`jmp [%o7+8]`** (skipping the delay slot of **`call`**). Using a **`restore`** as delay slot for **`ret`** resets the register window for the caller. Return addresses are written to the stack when a register window flush occurs.

saved framepointers	x86 requires explicit stack frame maintenance, both on entry:
	`pushl %ebp`
	`movl  %esp,%ebp`
	`subl  ...,%esp`
	or, as “abbreviation” for these:
	`enter ...`
	and on exit from a function:
	`movl  %ebp,%esp`
	`popl  %ebp`
	or, again as a single instruction:
	`leave`
	The use of a framepointer is optional – it can be optimized away.
	`call` writes the return address to the stack before the called function saves the framepointer.
	The stack therefore has pairs of return address/framepointer.
		SPARC gives framepointer maintenance for free via the register windowing mechanism. A function on entry reserves stackspace via:
		`save  %i6,...,%o6`
		and thereby also performs the window switch (new **`%i6`** = old **`%o6`**, and new **`%o6`** = old **`%o6`** minus frame size). **`save`** is equivalent to the explicit stack-/framepointer dance on x86.
		To undo the window switch and restore the caller's stack- and framepointer, a function will call **`restore`** on return.
		Stack-/Framepointer **`%i6`**/**`%o6`** are flushed on window spill.
		Return addresses are in **`%i7`**/**`%o7`** and the stack therefore has

stack detail	*How x86 does it*	*How SPARC does it*
		pairs framepointer/return address.
stackspace usage	x86 stacks are *dense*. Functions that can do all their work using their arguments and the available scratch registers will not allocate stackspace at all. A stack sequence where two return address/framepointer pairs follow each other without anything in-between is common.	SPARC stacks are *sparse*. Stackframes on SPARC are at least **`MINFRAME`** size – to provide backing store for the 16 new registers 'allocated' by the register window switch. Even a function that doesn't use all of its **`%i./%l./%o.`** registers must allocate backing store for them. The minimum distance between framepointer/return address pairs therefore is **`MINFRAME`**.
stack bias	x86 stack- and framepointers are not biased, neither in 32bit nor in 64bit mode. A x86 CPU has other means of detecting the operating mode than deriving that from whether the stackpointer is misaligned or not. See chapter 3.	SPARC 64bit stackpointers are *biased* – offset by **`0x7ff`**, and therefore always misaligned. The SPARC kernel uses this for two purposes: 1. To distinguish between 32bit and 64bit applications in trap and system call handlers. 2. Due to the address offset size limitation, **`±0x7ff`**, biased stacks allow 64bit code on SPARC to access stackframes twice the size without explicit offset calculations.

3.5.The role of the stack

On RISC architectures with huge numbers of registers, assembly-level programmer and/or engineers doing low-level troubleshooting and crash-/coredump analysis often can ignore the stack – everything interesting is in registers, the stack is purely backing store and the debugging tools “just display” arguments or local variables for a given function in the backtrace. This is especially true on SPARC, where the register window architecture makes this approach comfortable and easy – the debugger finds me the register window and I will then only look at the registers of a specific function.

This is not so on x86 platforms. The scarcity of registers forces functions to allocate stack space for local variables. It has already been shown that at least in 32bit mode, even argument passing is done via the stack. In 64bit, where arguments are passed in registers, they may *indirectly* appear on the stack – the called function often has to save them into “hidden locals” on its stackframe to preserve them for the case that it needs the arg registers for doing another **call** itself.

All this makes explicit stack accesses the most-frequent operation x86 machine code will do. Statistics illustrate this. The following shellscript:

```
#!/bin/ksh
functions=$(
echo “::nm -t func -f name” | mdb -k | sort -u | awk '{ if (NR > 1) print }'
)
for fnc in $functions; do
      asmcode=$(echo “$fnc::dis” | mdb -k)
      linestotal=$(echo “$asmcode” | wc -l)
      stackaccess$(echo “$asmcode” | egrep '[re][bs]p|push|pop' | wc -l)
      echo “$fnc $linestotal $stackaccess” |
            awk '{ printf(“%s %d %d %2.2f\n”, $1, $2, $3, 100*$3/$2) }'
done 2>/dev/null
```

allows to create a table what percentage of the instructions in Solaris kernel functions is an explicit stack access. This gives:

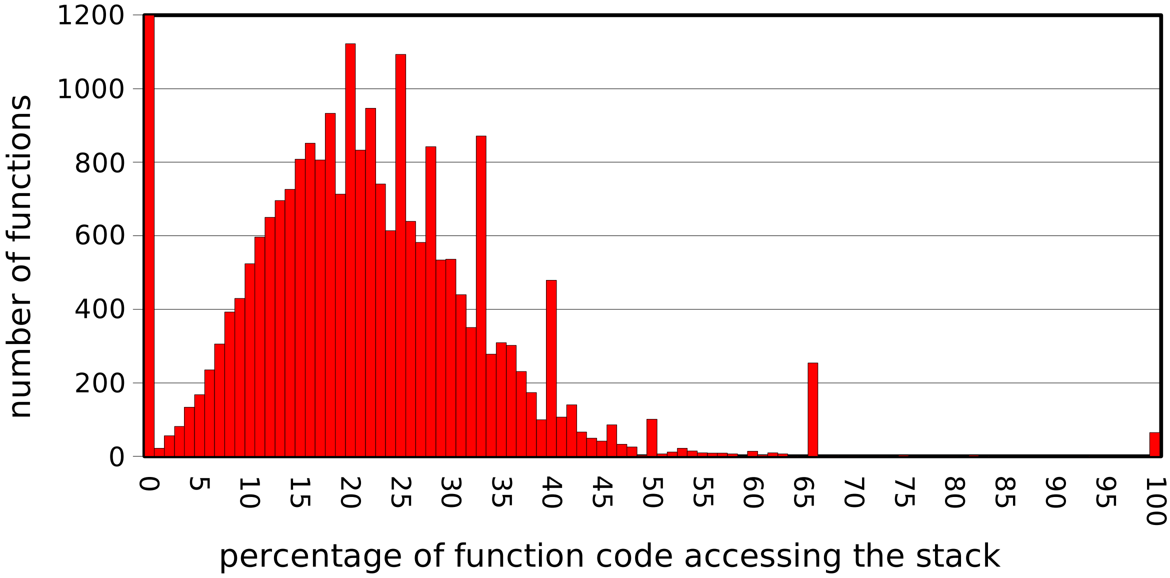


Illustration 7 - stack access by ~25000 functions in Solaris 10/amd64

The result, averaged over every function in Solaris 10/x86 64bit, is a whopping 23% – wow. Almost every fourth instruction on average is a stack access !

For a 32bit kernel, due to the fact that argument passing there is done on the stack as well, we expect even higher figures. Verifying this is left as an exercise to the reader !

Working with x86 assembly code, and even more so doing post-mortem debugging on x86 platforms therefore requires a thorough understanding of the stack. We have to become, literally, able to identify each and every value in a given stack to reconstruct what has been happening in this function call sequence.

3.5.1.Stack basics

Stack access can be *implicit* or *explicit* in x86.

Implicit stack access is done by instructions that modify the stackpointer and/or contents of the stack without requiring stack- or framepointer as argument – typically **PUSH** or **POP**. This is the classical stack – a *LIFO* (last in first out) array.

A C language type declaration for using a stack would look like this:

```
void *stackpointer[];
#define PUSH(value) *(--stackpointer) = (void *)value
#define POP(value) value = *(stackpointer++)
```

The stack by convention *grows downward*. Subtracting something from the stackpointer therefore means stackspace is being allocated, and vice versa adding to the stackpointer frees stackspace. **PUSH** and **POP** do that implicitly – allocate/free one word of stackspace and put/get the argument (from) there.

x86 has **pushl** (32bit)/**pushq** (64bit) and the corresponding **popl**/**popq** instructions, it implements a stack as shown above in hardware. **push**/**pop** are extremely common operations. In addition to these basics, x86 has several important instructions which push/pop values implicitly and do something with it. Such instructions also come in pairs, like **push** and **pop**.

The first pair is **call** and **ret**. They are used to perform function calling.

- **call** transfers execution *and* saves information for the called function where to return to once its work is done. The *return address* (address of the instruction following **call**) is put – on the stack. **call** therefore is equivalent to (nonexistant) pseudocode:

  ```
  PUSH <INSTRUCTION POINTER + sizeof(call instrunction)>
  GOTO <CALL TARGET>
  ```

- **ret** of course is the opposite – the called function uses this when it's done, in order to resume the caller. Given how **call** works on x86, pseudocode for **ret** must be:

  ```
  POP  <RETURN ADDRESS>
  GOTO <RETURN ADDRESS>
  ```

The other pair, **enter**/**leave**, is also related to code modularization/function calling. To explain what exactly these do, we need to go into the details of modularization again – how does nested function calling work, and where do function put their data ?

It was already mentioned that the stack can be used by functions in low-level code to hold transient data – arguments, local variables and return values. Since for obvious reasons two functions don't share local variables, that means every function in a calling sequence must get its own dedicated piece of stackspace – the *stack frame* of this function. It is convenient to let stackframes of caller/callee overlap – for passing data between two functions, if required. SPARC register windows with output/local/input registers use this model for registers, but in a more rigid version (fixed size frames) than “generic stacks”. Stacks therefore generically look like this:

With a stackpointer-only model, the running function only knows the *end of its stack* –

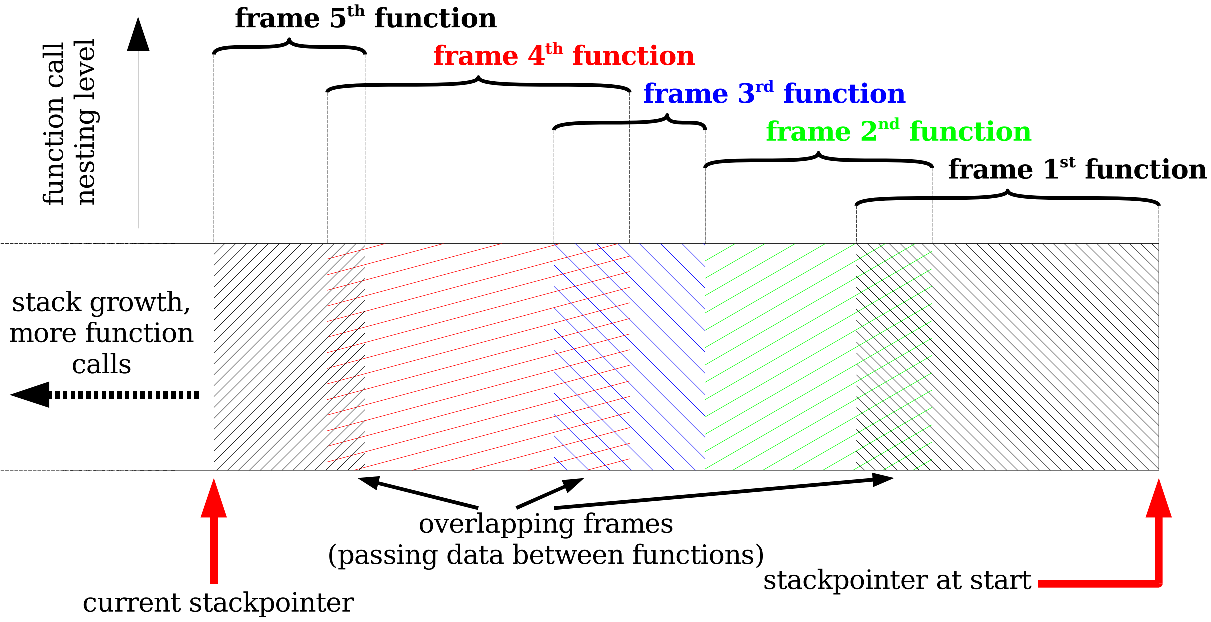


Illustration 8 - stack layout

but has no direct knowledge of where its stackframe starts. It can go the hard way and count (or let the compiler count during code generation) how many items it has on the stack at a particular point – or it keeps a record of what the stackpointer was at entry to the function, by recording the *start of its stackframe*. This is called *framepointer*. Stack- and framepointer specify where a stackframe starts and where it ends.

- The ***framepointer*** locates the start of the last stackframe.
- The ***stackpointer*** records the location of the last item on the stack.

Since a framepointer is nice to have, x86 provides a pair of instructions **`enter`**/**`leave`** that functions may use on entry/exit to allocate/free stackspace and switch framepointer and stackpointer.

- **`enter`** is equivalent to the following sequence of simple instructions (32bit shown):

  ```
  pushl %ebp        save previous framepointer
  movl  %esp,%ebp   new framepointer: stackpointer at entry
  subl  $...,%esp   allocate stackspace.
  ```

  It's uncommon to see compiler-generated code that uses **`enter`** – both AMD's and Intel's Optimization Guidelines deprecate this instruction in favour of the simple sequence shown.

- **`leave`** is equivalent to the following sequence of simple instructions (64bit shown):

  ```
  movq  %rbp,%rsp   free stackspace
  popq  %rbp        restore previous framepointer
  ```

  It depends on the compiler whether it uses **`leave`** or the simple sequence – both are commonly found.

These are the common x86 instructions that implicitly access the stack. But stack- and framepointer both are registers (**`%esp`**/**`%rsp`**: stackpointer, **`%ebp`**/**`%rbp`**: framepointer, in 32/64bit mode), and therefore usable with any of the x86 addressing modes to locate data on the stack. *Explicit* stack access via stack- or framepointer-indirect addressing therefore has the advantage of locating any item on the stack directly – not just the topmost one as **`PUSH`**/**`POP`**.

3.5.2.Stack contents in detail

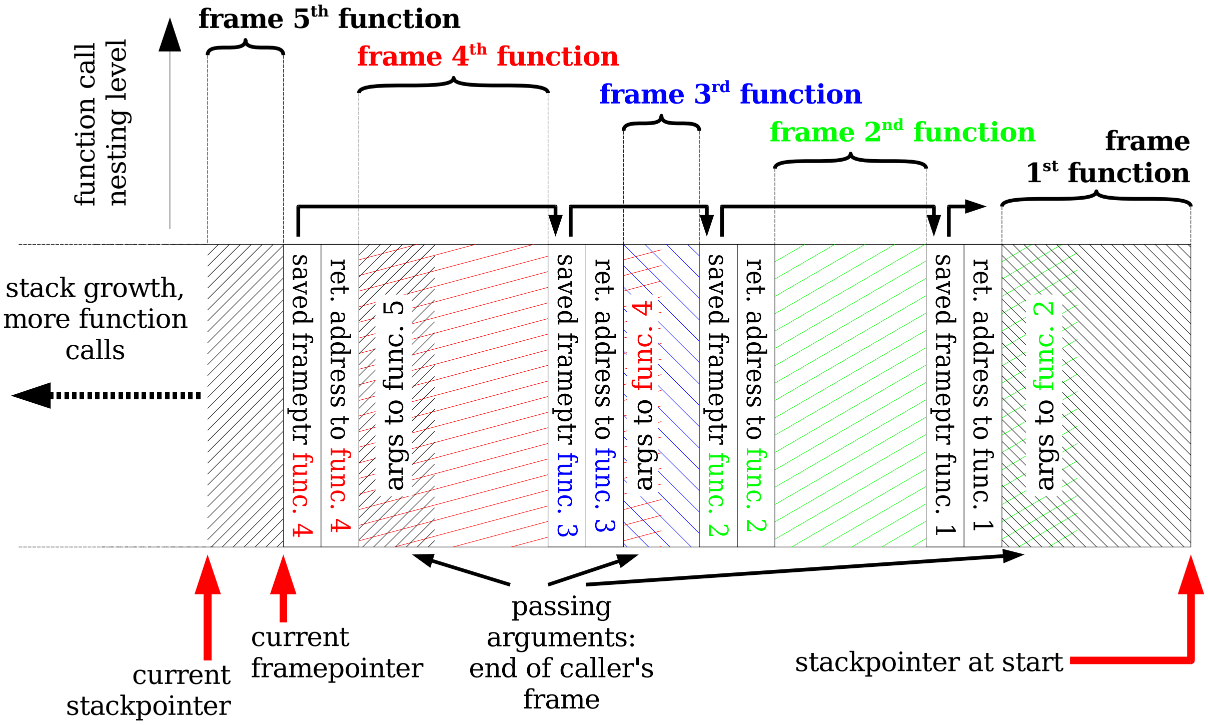


Illustration 9 - x86 stack layout details

Matching this to real stack data from core- and crashdumps on Solaris/x86 will be explained in all details in chapter 6.

3.5.3.Advanced: Buffer overflow exploits

Not sure – shall I include this ?

3.6. Odd things about the x86 instruction set

Talk about things like **`LEA`**, **`REP`**, **`LOOP`**, ... ? Floating point and MMX/SSE ?

3.7.Examples of compiler-generated code on x86 platforms

In order to give the reader a better impression what to expect when looking at disassembled program code, this section will provide examples of compiler-generated assembly from typical C language constructs:

- function calling, argument passing, argument access
- **`if (...) { ... } else { ... }`** statements
- **`switch { ... }`** statements
- **`for (...)`** loops
- accessing data structures and arrays.

We will also check how the binary code for data access changes if the compiler is instructed to create *position-independent code*.

Example sourcecode and resulting compiled code will be given for 32bit and 64bit. GNU gcc and Sun Workshop 10 compilers are used to illustrate compiler differences.

3.7.1.x86 instructions used by Sun Workshop & GNU C Compilers

Overall, compiled code on x86, whether 32bit or 64bit, uses far fewer instructions than one could expect given the ~1000 different ones that the instruction set provides. As already indicated before, efficient programming on x86 means using the chip like it were RISC, and restrict oneself to a set of “fastpath instructions”. But what are those ?

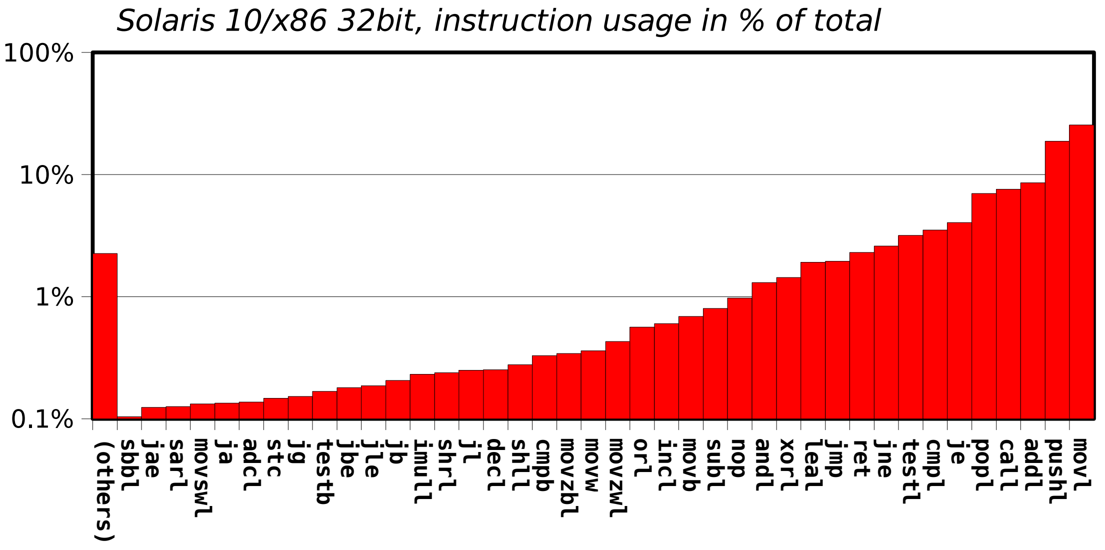


Illustration 10 - 40 most-used instructions in the 32bit Solaris/x86 kernel

This diagram has been created by iterating over the symbol table of a 32bit Solaris/x86 crashdump and disassembling every function. In total, 1689120 instructions were found and then binned by instruction name – the graph shows the result.

Since the Solaris 10 kernel is a pretty large piece of code (albeit integer only), we can assume with a good level of confidence that the distribution of how frequent certain instructions are being used here is representative of what we can expect in *any* large compiler-generated binary.

Let's look at the same histogram as shown before, but now generated from a 64bit

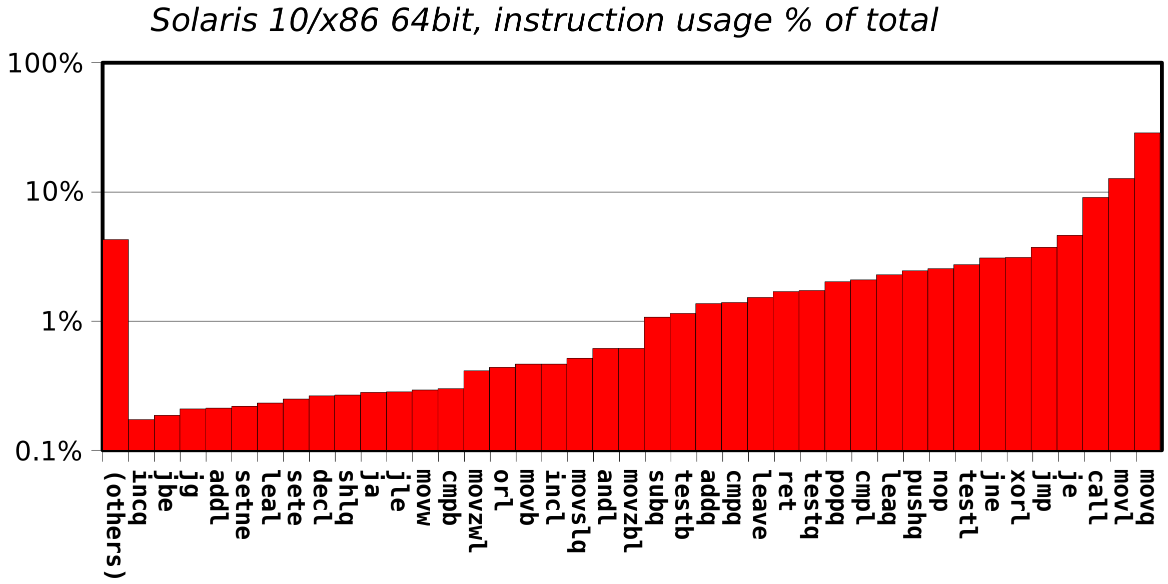


Illustration 11 - 40 most-used instructions in the 64bit Solaris/x86 kernel

This not only differs in 32bit vs. 64bit, but also shows how different compilers can change the set of instructions used in compiler-generated binary code – the 64bit Solaris 10 kernel is compiled using gcc 3.4.3.

Some explanation of these graphs and of course the shown frequently-used instructions is in order.

First, we note in both 32bit and 64bit modes, **mov** (in all its sizes) is the by-far most-frequently used instruction. This has several reasons:

- **mov** is multi-purpose. It can:
  - *copy* a register into another,
  - *initialize* a register or memory with a constant,
  - do memory *loads and stores*.
- Our statistics for stack access have already shown that almost ¼th of all code in 64bit (and as mentioned even more in 32bit) does implicitly or explicitly use the stack. Not all of this is going to be **push**/**pop** – some significant amount will be explicit stack access via **mov**.

Considering the stack brings us to other important instructions. We note that on 32bit, **pushl** and **addl** are 2nd and 3rd place while their 64bit cousins **pushq**/**addq** are much less frequent. The reason for this again is the ABI difference – 32bit code passes arguments on the stack and uses **pushl** to do so. After returning from call, **addl** is (often) used to clean up the stack (i.e. pop off the arguments into "nowhere"). On 64bit, neither is necessary unless there are more than six arguments – so the remaining use for **pushq**/**popq** that now dominates their usage is the function prologue. The ratio of 64bit **popq** vs. **pushq** is much closer than the one for 32bit **pushl** vs. **popl**.

The next big contributors are of course **call**/**ret**/**leave**, and as indicated **push**/**pop** as part of function pro- and epilogues. The fact that the 32bit code doesn't seem to use **leave** highlights a compiler difference – as the disassembly examples later in this chapter will show, gcc tends to use **leave** in epilogues while Workshop cc uses the equivalent simple instruction sequence, and the 32bit code was compiled using Workshop cc, while the 64bit code is gcc-created.

Another big part of code is anything related to branching – **cmp** and **test** in all sizes, the different conditional branches **j..**, and **jmp** as unconditional goto. Not every code sequence can be streamlined, of course. Note that in 64bit mode, we more frequently use **cmpl** and **testl** than their 64bit cousins – which shows the code doesn't use 64bit operand size that often (except for pointers, of course), and proves AMD was right in making 32bit operand size the default even when running in 64bit mode. **cmpl/testl** use less code space and more efficient in execution than **cmpq/testq**.

Then, there's **xorl** – in both 32bit and 64bit. We see much more **xorl** operations than any other arithmetic/logical instruction. Why on earth should we xor things so often ? Well, we don't. But there's one peculiar **xorl** operation that's very handy, and that is to xor a register with itself – *zeroing it*. Due to zero-extension for 32bit operations, that even works in 64bit mode. The same instruction, **xorl %eax,%eax** zeros **%eax** in 32bit and **%rax** in 64bit mode. SPARC has **%g0**, x86 has **xorl**.

The next peculiar thing is the **lea** instruction (**leal** and **leaq**). **lea** stands for *load effective address*, but its name notwithstanding this is no memory access. **lea** does address calculation. It performs pointer arithmetics, if you like. The difference between

```
    leal +0x10(%eax,%ecx,8), %esi
    movl +0x10(%eax,%ecx,8),%esi
```

is the same as the difference between C code that does something like:

```
    varaddr = &mystruct->array[i];
    varcontents = mystruct->array[i];
```

In other words: **lea** is an arithmetic operation – it takes the address specification as a formula and calculates the result without actually attempting to dereference. **lea** can (and will) be used for general-purpose arithmetics, as long as the desired computation can be expressed within the bounds of what x86 addressing modes allow.

Following this is **nop** (and in 32bit code **andl**). Both are for optimization – while x86 CPUs don't require instruction (or stack-) pointer alignment, they may perform better in some situations if code or stack are aligned at e.g. multiples of 16. Both compilers are aware of this and therefore generate the necessary code.

What remains in the shown diagrams falls into two classes of instructions:

- *arithmetics* of all sorts. **add**, **adc** (add with carry), **sub**, **sbb** (subtract with borrow), **inc**, **dec**, **imul**, **or**, **and**, **shr/shl** resp. **sar/sal** (logical/arithmetical shifts) and more all fall into this category.
- *sign* and *zero extension*. This is what **movs[bwl][lq]** and **movz[bwl]l** are for. Remember that in 64bit, zero extension is done by default for 32bit operations – there's no need for a **movz[bwl]q** instruction because that's implicit. But of course sign extension to 64bit, **movs[bwl]q**, is required now and then,

To look up the documentation for the remaining instructions named in the graphs, **sete/setne** and **stc**, is left as an exercise to the reader, along with familiarizing oneself with the ~3% resp. ~5% of instructions that are classified as *others* above !

For the following examples, the given sourcecode was compiled using Sun Workshop 10 cc and GNU gcc 3.4.3 using the commands:

```
    cc -xO3 -xarch=amd64 -c ....c
    gcc -O2 -m64 -c ....c
```

Thin `fixedwidth`: additional switch to create 64bit code.
In all assembly listings, color and font coding is used:

- ***Green*** shows *function prologue* and *function epilogue*.
- ***Red*** shows when the function acccesses its own arguments.

- ***Blue*** shows function calls (and their arguments) made by the given code.
- *`thin italics`* designates branch targets within the function.

The codesamples shown are of course only a small fraction of “what is possible”. Their purpose is to illustrate both the general structure of compiler-generated output for 32bit and 64bit x86, and to demonstrate how different compiler output from two different code generators (those of gcc and Workshop) can become even in such simple cases.

3.7.2.`if()` … else statements

Sourcecode sample:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int rfile(char *filename, void *destbuf, off_t offset, size_t nbyte)
{
      int fd;

      if ((fd = open(filename, O_RDONLY, 0444)) < 0) {
            perror(“open failed”);
            return (-1);
      } else if (nbytes > 0) {
            if (pread(fd, destbuf, nbyte, offset) != nbyte) {
                  perror(“pread failed”);
                  return (-2);
            }
      } else {
            printf(“fooling me ? NULL read at %ull\n”, offset);
      }
      return (0);
}
```

The example was compiled adding **`` `getconf LFS_CFLAGS` ``** to compiler switches. This allows to demonstrate how 64bit values are passed as arguments in a 32bit program.

For the 32bit code, we find that the 64bit value (**`off_t`**) actually is passed as two separate 32bit values – in **`+0x10(%ebp)`** and **`+0x14(%ebp)`**, in this case.

Apart from that, the 32bit assembler code shows compiler differences.

Regarding optimizations, we find:

- The Sun Workshop compiler tends to inline function epilogues multiple time in order to avoid branches. Interesting to note that although it does this, the total amount of code is still less than the gcc output.
- gcc on the other hand puts arguments into nonvolatile registers in order to avoid having to re-fetch them from the stack.

Regarding how the stack is being used, we see:

- The Workshop compiler aligns the stackpointer on entry but not on **`call`**, where it removes exactly the number of args afterwards using **`addl`** as it **`pushl`**'ed before.
- gcc doesn't align the stackpointer on entry but when doing a function call, it always reserves a multiple of 16 bytes (**`0x10`**) for argument stackspace – even if the called function takes less than that.

x86 32bit binary, Sun Workshop cc			***x86 32bit binary, GNU gcc 3.4.3***		
rfile:	**pushl**	**%ebp**	**rfile:**	**pushl**	**%ebp**
rfile+0x1:	**movl**	**%esp,%ebp**	**rfile+0x1:**	**movl**	**%esp,%ebp**
rfile+0x3:	**andl**	**$0xfffffff0,%esp**	**rfile+0x3:**	**pushl**	**%edi**
rfile+0x6:	**pushl**	**%ebx**	**rfile+0x4:**	**pushl**	**%esi**
rfile+0x7:	**pushl**	**$0x124**	**rfile+0x5:**	**pushl**	**%ebx**
rfile+0xc:	**pushl**	**$0x0**	**rfile+0x6:**	**subl**	**$0x10,%esp**
rfile+0xe:	**pushl**	**0x8(%ebp)**	**rfile+0x9:**	**pushl**	**$0x124**
rfile+0x11:	**call**	**open64**	**rfile+0xe:**	**pushl**	**$0x0**
rfile+0x16:	**addl**	**$0xc,%esp**	**rfile+0x10:**	**pushl**	**0x8(%ebp)**
rfile+0x19:	**testl**	**%eax,%eax**	**rfile+0x13:**	**movl**	**0x10(%ebp),%esi**
rfile+0x1b:	**jl**	*<rfile+0x57>*	**rfile+0x16:**	**movl**	**0x14(%ebp),%edi**
rfile+0x1d:	**movl**	**0x18(%ebp),%ebx**	**rfile+0x19:**	**movl**	**0x18(%ebp),%ebx**
rfile+0x20:	**testl**	**%ebx,%ebx**	**rfile+0x1c:**	**call**	**open64**
rfile+0x22:	**jbe**	*<rfile+0x3d>*	**rfile+0x21:**	**addl**	**$0x10,%esp**
rfile+0x24:	**pushl**	**0x14(%ebp)**	**rfile+0x24:**	**testl**	**%eax,%eax**
rfile+0x27:	**pushl**	**0x10(%ebp)**	**rfile+0x26:**	**js**	*<rfile+0x75>*
rfile+0x2a:	**pushl**	**%ebx**	**rfile+0x28:**	**testl**	**%ebx,%ebx**
rfile+0x2b:	**pushl**	**0xc(%ebp)**	**rfile+0x2a:**	**je**	*<rfile+0x4c>*
rfile+0x2e:	**pushl**	**%eax**	**rfile+0x2c:**	**subl**	**$0xc,%esp**
rfile+0x2f:	**call**	**pread64**	**rfile+0x2f:**	**pushl**	**%edi**
rfile+0x34:	**addl**	**$0x14,%esp**	**rfile+0x30:**	**pushl**	**%esi**
rfile+0x37:	**cmpl**	**%ebx,%eax**	**rfile+0x31:**	**pushl**	**%ebx**
rfile+0x39:	**je**	*<rfile+0x50>*	**rfile+0x32:**	**pushl**	**0xc(%ebp)**
rfile+0x3b:	**jmp**	*<rfile+0x6e>*	**rfile+0x35:**	**pushl**	**%eax**
rfile+0x3d:	**pushl**	**0x14(%ebp)**	**rfile+0x36:**	**call**	**pread64**
rfile+0x40:	**pushl**	**0x10(%ebp)**	**rfile+0x3b:**	**addl**	**$0x20,%esp**
rfile+0x43:	**pushl**	**$printf_arg1**	**rfile+0x3e:**	**cmpl**	**%ebx,%eax**
rfile+0x48:	**call**	**printf**	**rfile+0x40:**	**jne**	*<rfile+0x5e>*
rfile+0x4d:	**addl**	**$0xc,%esp**	*rfile+0x42*:	**xorl**	**%eax,%eax**
rfile+0x50:	**xorl**	**%eax,%eax**	*rfile+0x44*:	**leal**	**-0xc(%ebp),%esp**
rfile+0x52:	**popl**	**%ebx**	**rfile+0x47:**	**popl**	**%ebx**
rfile+0x53:	**movl**	**%ebp,%esp**	**rfile+0x48:**	**popl**	**%esi**
rfile+0x55:	**popl**	**%ebp**	**rfile+0x49:**	**popl**	**%edi**
rfile+0x56:	**ret**		**rfile+0x4a:**	**leave**	
rfile+0x57:	**pushl**	**$perror_openfail**	**rfile+0x4b:**	**ret**	
rfile+0x5c:	**call**	**perror**	*rfile+0x4c*:	**pushl**	**%eax**
rfile+0x61:	**addl**	**$0x4,%esp**	**rfile+0x4d:**	**pushl**	**%edi**
rfile+0x64:	**movl**	**$-0x1,%eax**	**rfile+0x4e:**	**pushl**	**%esi**
rfile+0x69:	**popl**	**%ebx**	**rfile+0x4f:**	**pushl**	**$printf_arg1**
rfile+0x6a:	**movl**	**%ebp,%esp**	**rfile+0x54:**	**call**	**printf**
rfile+0x6c:	**popl**	**%ebp**	**rfile+0x59:**	**addl**	**$0x10,%esp**
rfile+0x6d:	**ret**		**rfile+0x5c:**	**jmp**	*<rfile+0x42>*
rfile+0x6e:	**pushl**	**$perror_preadfail**	*rfile+0x5e*:	**subl**	**$0xc,%esp**
rfile+0x73:	**call**	**perror**	**rfile+0x61:**	**pushl**	**$perror_preadfail**
rfile+0x78:	**addl**	**$0x4,%esp**	**rfile+0x66:**	**call**	**perror**
rfile+0x7b:	**movl**	**$-0x2,%eax**	**rfile+0x6b:**	**movl**	**$-0x2,%eax**
rfile+0x80:	**popl**	**%ebx**	**rfile+0x70:**	**addl**	**$0x10,%esp**
rfile+0x81:	**movl**	**%ebp,%esp**	**rfile+0x73:**	**jmp**	*<rfile+0x44>*
rfile+0x83:	**popl**	**%ebp**	*rfile+0x75*:	**subl**	**$0xc,%esp**
rfile+0x84:	**ret**		**rfile+0x78:**	**pushl**	**$perror_openfail**
			rfile+0x7d:	**call**	**perror**
			rfile+0x82:	**movl**	**$-0x1,%eax**
			rfile+0x87:	**addl**	**$0x10,%esp**
			rfile+0x8a:	**jmp**	*<rfile+0x44>*

x86 64bit binary, Sun Workshop cc			*x86 64bit binary, GNU gcc 3.4.3*		
rfile:	pushq	%rbp	rfile:	pushq	%rbp
rfile+0x1:	movq	%rsp,%rbp	rfile+0x1:	xorl	%eax,%eax
rfile+0x4:	subq	$0x8,%rsp	rfile+0x3:	movq	%rsp,%rbp
rfile+0x8:	pushq	%r12	rfile+0x6:	movq	%rbx,-0x18(%rbp)
rfile+0xa:	pushq	%r13	rfile+0xa:	movq	%r12,-0x10(%rbp)
rfile+0xc:	pushq	%r14	rfile+0xe:	movq	%rdx,%r12
rfile+0xe:	movq	%rsi,%r14	rfile+0x11:	movq	%r13,-0x8(%rbp)
rfile+0x11:	movq	%rdx,%r12	rfile+0x15:	movl	$0x124,%edx
rfile+0x14:	movq	%rcx,%r13	rfile+0x1a:	subq	$0x20,%rsp
rfile+0x17:	xorl	%esi,%esi	rfile+0x1e:	movq	%rsi,%r13
rfile+0x19:	movl	$0x124,%edx	rfile+0x21:	xorl	%esi,%esi
rfile+0x1e:	xorl	%eax,%eax	rfile+0x23:	movq	%rcx,%rbx
rfile+0x20:	call	open	rfile+0x26:	call	open
rfile+0x25:	testl	%eax,%eax	rfile+0x2b:	testl	%eax,%eax
rfile+0x27:	jl	*<rfile+0x65>*	rfile+0x2d:	js	*<rfile+0x82>*
rfile+0x29:	testq	%r13,%r13	rfile+0x2f:	testq	%rbx,%rbx
rfile+0x2c:	je	*<rfile+0x47>*	rfile+0x32:	je	*<rfile+0x60>*
rfile+0x2e:	movl	%eax,%edi	rfile+0x34:	movq	%r12,%rcx
rfile+0x30:	movq	%r14,%rsi	rfile+0x37:	movq	%rbx,%rdx
rfile+0x33:	movq	%r13,%rdx	rfile+0x3a:	movq	%r13,%rsi
rfile+0x36:	movq	%r12,%rcx	rfile+0x3d:	movl	%eax,%edi
rfile+0x39:	xorl	%eax,%eax	rfile+0x3f:	call	pread
rfile+0x3b:	call	pread	rfile+0x44:	cmpq	%rbx,%rax
rfile+0x40:	cmpq	%r13,%rax	rfile+0x47:	jne	*<rfile+0x71>*
rfile+0x43:	je	*<rfile+0x58>*	*rfile+0x49*:	xorl	%eax,%eax
rfile+0x45:	jmp	*<rfile+0x83>*	*rfile+0x4b*:	movq	-0x18(%rbp),%rbx
rfile+0x47:	leaq	printfarg(%rip),%rdi	rfile+0x4f:	movq	-0x10(%rbp),%r12
rfile+0x4e:	movq	%r12,%rsi	rfile+0x53:	movq	-0x8(%rbp),%r13
rfile+0x51:	xorl	%eax,%eax	rfile+0x57:	leave	
rfile+0x53:	call	printf	rfile+0x58:	ret	
rfile+0x58:	xorl	%eax,%eax	rfile+0x59:	nop	
rfile+0x5a:	popq	%r14	rfile+0x5d:	nop	
rfile+0x5c:	popq	%r13	*rfile+0x60*:	movq	%r12,%rsi
rfile+0x5e:	popq	%r12	rfile+0x63:	movl	$printf_arg,%edi
rfile+0x60:	movq	%rbp,%rsp	rfile+0x68:	xorl	%eax,%eax
rfile+0x63:	popq	%rbp	rfile+0x6a:	call	printf
rfile+0x64:	ret		rfile+0x6f:	jmp	*<rfile+0x49>*
rfile+0x65:	leaq	err_open(%rip),%rdi	*rfile+0x71*:	movl	$err_pread,%edi
rfile+0x6c:	xorl	%eax,%eax	rfile+0x76:	call	perror
rfile+0x6e:	call	perror	rfile+0x7b:	movl	$-0x2,%eax
rfile+0x73:	movl	$-0x1,%eax	rfile+0x80:	jmp	*<rfile+0x4b>*
rfile+0x78:	popq	%r14	*rfile+0x82*:	movl	$err_open,%edi
rfile+0x7a:	popq	%r13	rfile+0x87:	call	perror
rfile+0x7c:	popq	%r12	rfile+0x8c:	movl	$-0x1,%eax
rfile+0x7e:	movq	%rbp,%rsp	rfile+0x91:	jmp	*<rfile+0x4b>*
rfile+0x81:	popq	%rbp			
rfile+0x82:	ret				
rfile+0x83:	leaq	err_pread(%rip),%rdi			
rfile+0x8a:	xorl	%eax,%eax			
rfile+0x8c:	call	perror			
rfile+0x91:	movl	$-0x2,%eax			
rfile+0x96:	popq	%r14			
rfile+0x98:	popq	%r13			
rfile+0x9a:	popq	%r12			
rfile+0x9c:	movq	%rbp,%rsp			
rfile+0x9f:	popq	%rbp			
rfile+0xa0:	ret				

The 64bit code shows several interesting peculiarities beyond the ABI differences (args in registers, and zeroing **`%rax`** before calling a function) that need explanation.

Both Workshop cc and gcc for example seemingly fail to initialize the first argument for **`open()`** - neither writes to **`%rdi`**. This is of course correct – it'll pass-through the input value. The filename is not needed anymore after this, so it isn't saved anywhere.

That's not true for the remaining arguments to **`rfile()`**. They're still needed, so both cc and gcc decide to keep these in nonvolatile registers – but only gcc recognizes that using **`%rbx`** is more efficient than using **`%r12`**..**`%r15`**.

As with 32bit code, Workshop cc inlines the epilogue several times while gcc keeps a single copy. Also, as mentioned, gcc uses **`leave`** while Workshop cc uses the equivalent two-instruction sequence.

The Sun Workshop compiler shows that *position-independent code* on AMD64 is very efficiently doable via **`%rip`**-relative addressing. Sun Workshop cc defaults to that for loading the addresses of the strings passed as arguments to **`printf()`** and **`perror()`**.

gcc's prologue code here demonstrates the use of the so-called *stack redzone*. In gcc's terms, the redzone are the 128 bytes of unallocated stackspace immediately below the current value of the stackpointer. This can be accessed using byte offsets relative to the stack/framepointer, which is what gcc does here instead of using **`pushq`** instructions.

I.e. gcc *accesses* the stack *first*, and *allocates* it *later*.

It is debatable whether this is an optimization; in any case, this behaviour is ***highly incompatible*** with kernel code both in Linux and Solaris – the kernel is preemptive and running with interrupts enabled, and those events may use the current thread's stack below what the stackpointer is at the time the interrupt occurs. In both Linux and Solaris kernels, gcc's redzone usage must be disabled or data corruption (with a crash likely soon afterwards) will result if an interrupt occurs before the **`subl ..., %rsp`** has been done.

gcc provides a compile flag, **`-mno-red-zone`**, that ***must*** be used when compiling kernel code with gcc, both under Solaris and Linux. It makes sure gcc allocates stackspace before attempting to use it.

For application code, the use of the stack redzone is not a problem in either Solaris or Linux – the AMD64 UNIX ABI guarantees its availability *to applications*.

3.7.3. for() loop example

Sourcecode:

```
extern int get_a_num(void);

long loopit(int times)
{
      long tmp = 123;
      int i;

      for (i = 10; i < times; i++) {
            tmp += get_a_num();
            tmp -= get_a_num();
      }
      return (tmp);
}
```

Disassembler output:

```
x86 32bit binary, Sun Workshop cc        x86 32bit binary, GNU gcc 3.4.3
loopit:       pushl  %ebp                loopit:       pushl  %ebp
loopit+0x1:   movl   %esp,%ebp           loopit+0x1:   movl   %esp,%ebp
loopit+0x3:   andl   $0xfffffff0,%esp    loopit+0x3:   movl   0x8(%ebp),%eax
loopit+0x6:   pushl  %ebx                loopit+0x6:   pushl  %esi
loopit+0x7:   pushl  %esi                loopit+0x7:   cmpl   $0xa,%eax
loopit+0x8:   pushl  %edi                loopit+0xa:   pushl  %ebx
loopit+0x9:   movl   0x8(%ebp),%edi      loopit+0xb:   movl   $0x7b,%esi
loopit+0xc:   movl   $0x7b,%esi          loopit+0x10:  jle          <loopit+0x29>
loopit+0x11:  cmpl   $0xa,%edi           loopit+0x12:  leal   -0xa(%eax),%ebx
loopit+0x14:  jle          <loopit+0x2e> loopit+0x15:  leal   0x0(%esi),%esi
loopit+0x16:  movl   $0xa,%ebx           loopit+0x18:  call   get_a_num
loopit+0x1b:  call   get_a_num           loopit+0x1d:  addl   %eax,%esi
loopit+0x20:  addl   %eax,%esi           loopit+0x1f:  call   get_a_num
loopit+0x22:  call   get_a_num           loopit+0x24:  subl   %eax,%esi
loopit+0x27:  subl   %eax,%esi           loopit+0x26:  decl   %ebx
loopit+0x29:  incl   %ebx                loopit+0x27:  jne          <loopit+0x18>
loopit+0x2a:  cmpl   %edi,%ebx           loopit+0x29:  popl   %ebx
loopit+0x2c:  jl           <loopit+0x1b> loopit+0x2a:  movl   %esi,%eax
loopit+0x2e:  movl   %esi,%eax           loopit+0x2c:  popl   %esi
loopit+0x30:  popl   %edi                loopit+0x2d:  leave
loopit+0x31:  popl   %esi                loopit+0x2e:  ret
loopit+0x32:  popl   %ebx
loopit+0x33:  movl   %ebp,%esp
loopit+0x35:  popl   %ebp
loopit+0x36:  ret
```

Compiler differences are obvious here:

- The Workshop compiler uses three nonvolatile registers, **%edi**, **%esi** and **%ebx**, for the three variables **times**, **tmp** and **i**, and counts the loop forward, starting at 10.
- The GNU C compiler eliminates **i** as a variable and instead counts **(times-10)** downwards to zero. It therefore only uses **%esi** for **tmp** and **%ebx** for the inverted counter. It also interleaves the **pushl** operations in the prologue with instructions that don't access memory.

x86 64bit binary, Sun Workshop cc

```
loopit:       pushq  %rbp
loopit+0x1:   movq   %rsp,%rbp
loopit+0x4:   pushq  %rbx
loopit+0x5:   pushq  %r12
loopit+0x7:   pushq  %r13
loopit+0x9:   pushq  %r14
loopit+0xb:   movl   %edi,%r13d
loopit+0xe:   movq   $0x7b,%r14
loopit+0x15:  movl   $0xa,%ebx
loopit+0x1a:  cmpl   $0xa,%r13d
loopit+0x1e:  jle           <loopit+0x44>
loopit+0x20:  xorl   %eax,%eax
loopit+0x22:  call   get_a_num
loopit+0x27:  movslq %eax,%r12
loopit+0x2a:  addq   %r14,%r12
loopit+0x2d:  xorl   %eax,%eax
loopit+0x2f:  call   get_a_num
loopit+0x34:  movslq %eax,%r8
loopit+0x37:  movq   %r12,%r14
loopit+0x3a:  subq   %r8,%r14
loopit+0x3d:  incl   %ebx
loopit+0x3f:  cmpl   %r13d,%ebx
loopit+0x42:  jl            <loopit+0x20>
loopit+0x44:  movq   %r14,%rax
loopit+0x47:  popq   %r14
loopit+0x49:  popq   %r13
loopit+0x4b:  popq   %r12
loopit+0x4d:  popq   %rbx
loopit+0x4e:  movq   %rbp,%rsp
loopit+0x51:  popq   %rbp
loopit+0x52:  ret
```

x86 64bit binary, GNU gcc 3.4.3

```
loopit:       pushq  %rbp
loopit+0x1:   cmpl   $0xa,%edi
loopit+0x4:   movq   %rsp,%rbp
loopit+0x7:   pushq  %r12
loopit+0x9:   movl   $0x7b,%r12d
loopit+0xf:   pushq  %rbx
loopit+0x10:  jle           <loopit+0x2d>
loopit+0x12:  leal   -0xa(%rdi),%ebx
loopit+0x15:  call   get_a_num
loopit+0x1a:  cltq
loopit+0x1c:  addq   %rax,%r12
loopit+0x1f:  call   get_a_num
loopit+0x24:  cltq
loopit+0x26:  subq   %rax,%r12
loopit+0x29:  decl   %ebx
loopit+0x2b:  jne           <loopit+0x15>
loopit+0x2d:  popq   %rbx
loopit+0x2e:  movq   %r12,%rax
loopit+0x31:  popq   %r12
loopit+0x33:  leave
loopit+0x34:  ret
```

The 64bit code shows the ABI differences; both compilers do:

- access the (only) argument via **`%edi`** (not **`%rdi`** – the argument is type **`int`**)
- clear **`%rax`** (not **`%eax`** – remember 32bit operations zero-extend) before calling **`get_a_num()`** before making a function call. This is mandated by the 64bit ABI.
  - Workshop cc uses **`xorl`** for the purpose
  - gcc chooses **`cltq`**. (clear-to-quad, zeroes both **`%eax`** and **`%edx`**).

The gcc 64bit code looks almost identical to the 32bit code otherwise. But the Sun Workshop compiler has chosen to use four registers (not three as in 32bit), using two registers for the intermediate values of **`tmp`**.

3.7.4. switch() statements

Sourcecode:

```
#include <stdio.h>

extern void func_a(int);
extern void func_b(int);
extern void func_c(void);
extern void func_d(int);
extern void func_e(long);
extern void func_default(int);

void disp(long code)
{
      switch (code) {
      case 280:
            func_a(1);
            break;
      case 880:
            func_b(2);
            break;
      case 92:
            func_c();
            break;
      case 101:
            func_d(3);
            break;
      case 237:
            func_e(code);
            break;
      default:
            func_default(4);
            break;
      }
      printf("dispatch done for %ld\n", code);
}
```

This is the most-complicated example in this section.

There are multiple ways how compilers can create assembly code for such source. Sun Workshop cc and gcc actually aren't that dissimilar here, both, in 32bit as in 64bit, create sequences of **cmp/je** instructions followed by a direct **jmp** to the reach the end of the **switch {}** statement.

Depending on the exact structure of the **switch {}**, and the values **case:** check for, many optimizations are possible of course, and one cannot expect a generic, real-world sourcecode to result in assembly code as easy to read as that given on the following pages. This is *one* example of how assembly code for **switch {}** can look like – but reality will often be more complex.

x86 32bit binary, Sun Workshop cc

```
disp:      pushl  %ebp
disp+0x1:  movl   %esp,%ebp
disp+0x3:  andl   $0xfffffff0,%esp
disp+0x6:  pushl  %ebx
disp+0x7:  movl   0x8(%ebp),%ebx
disp+0xa:  cmpl   $0x5c,%ebx
disp+0xd:  je           <disp+0x67>
disp+0xf:  cmpl   $0x65,%ebx
disp+0x12: je           <disp+0x5b>
disp+0x14: cmpl   $0xed,%ebx
disp+0x1a: je           <disp+0x50>
disp+0x1c: cmpl   $0x118,%ebx
disp+0x22: je           <disp+0x44>
disp+0x24: cmpl   $0x370,%ebx
disp+0x2a: je           <disp+0x38>
disp+0x2c: pushl  $0x4
disp+0x2e: call   func_default
disp+0x33: addl   $0x4,%esp
disp+0x36: jmp          <disp+0x6c>
disp+0x38: pushl  $0x2
disp+0x3a: call   func_b
disp+0x3f: addl   $0x4,%esp
disp+0x42: jmp          <disp+0x6c>
disp+0x44: pushl  $0x1
disp+0x46: call   func_a
disp+0x4b: addl   $0x4,%esp
disp+0x4e: jmp          <disp+0x6c>
disp+0x50: pushl  %ebx
disp+0x51: call   func_e
disp+0x56: addl   $0x4,%esp
disp+0x59: jmp          <disp+0x6c>
disp+0x5b: pushl  $0x3
disp+0x5d: call   func_d
disp+0x62: addl   $0x4,%esp
disp+0x65: jmp          <disp+0x6c>
disp+0x67: call   func_c
disp+0x6c: pushl  %ebx
disp+0x6d: pushl  $printf_arg1
disp+0x72: call   printf
disp+0x77: addl   $0x8,%esp
disp+0x7a: popl   %ebx
disp+0x7b: movl   %ebp,%esp
disp+0x7d: popl   %ebp
disp+0x7e: ret
```

x86 32bit binary, GNU gcc 3.4.3

```
disp:      pushl  %ebp
disp+0x1:  movl   %esp,%ebp
disp+0x3:  pushl  %ebx
disp+0x4:  pushl  %eax
disp+0x5:  movl   0x8(%ebp),%ebx
disp+0x8:  cmpl   $0xed,%ebx
disp+0xe:  je           <disp+0x60>
disp+0x10: jle          <disp+0x48>
disp+0x12: cmpl   $0x118,%ebx
disp+0x18: je           <disp+0x6f>
disp+0x1a: cmpl   $0x370,%ebx
disp+0x20: je           <disp+0x82>
disp+0x22: subl   $0xc,%esp
disp+0x25: pushl  $0x4
disp+0x27: call   func_default
disp+0x2c: addl   $0x10,%esp
disp+0x2f: subl   $0x8,%esp
disp+0x32: pushl  %ebx
disp+0x33: pushl  $printf_arg1
disp+0x38: call   printf
disp+0x3d: addl   $0x10,%esp
disp+0x40: movl   -0x4(%ebp),%ebx
disp+0x43: leave
disp+0x44: ret
disp+0x45: leal   0x0(%esi),%esi
disp+0x48: cmpl   $0x5c,%ebx
disp+0x4b: je           <disp+0x7b>
disp+0x4d: cmpl   $0x65,%ebx
disp+0x50: jne          <disp+0x22>
disp+0x52: subl   $0xc,%esp
disp+0x55: pushl  $0x3
disp+0x57: call   func_d
disp+0x5c: jmp          <disp+0x2c>
disp+0x5e: movl   %esi,%esi
disp+0x60: subl   $0xc,%esp
disp+0x63: pushl  $0xed
disp+0x68: call   func_e
disp+0x6d: jmp          <disp+0x2c>
disp+0x6f: subl   $0xc,%esp
disp+0x72: pushl  $0x1
disp+0x74: call   func_a
disp+0x79: jmp          <disp+0x2c>
disp+0x7b: call   func_c
disp+0x80: jmp          <disp+0x2f>
disp+0x82: subl   $0xc,%esp
disp+0x85: pushl  $0x2
disp+0x87: call   func_b
disp+0x8c: jmp          <disp+0x2c>
```

Note that this code shows a difference in how Workshop cc and gcc pass arguments. While Workshop cc uses **`pushl/call/addl`** to put arguments onto the stack and remove exactly this amount from the stack again after the function call, GNU gcc allocates 16 bytes of stackspace for args even if the called function uses less than that.

x86 64bit binary, Sun Workshop cc	***x86 64bit binary, GNU gcc 3.4.3***
disp: pushq %rbp	disp: pushq %rbp
disp+0x1: movq %rsp,%rbp	disp+0x1: movq %rsp,%rbp
disp+0x4: subq $0x8,%rsp	disp+0x4: pushq %rbx
disp+0x8: pushq %r12	disp+0x5: movq %rdi,%rbx
disp+0xa: movq %rdi,%r12	disp+0x8: subq $0x8,%rsp
disp+0xd: cmpq $0x5c,%r12	disp+0xc: cmpq $0xed,%rdi
disp+0x11: je *<disp+0x78>*	disp+0x13: je *<disp+0x68>*
disp+0x13: cmpq $0x65,%r12	disp+0x15: jle *<disp+0x50>*
disp+0x17: je *<disp+0x6a>*	disp+0x17: cmpq $0x118,%rdi
disp+0x19: cmpq $0xed,%r12	disp+0x1e: je *<disp+0x87>*
disp+0x20: je *<disp+0x5e>*	disp+0x20: cmpq $0x370,%rdi
disp+0x22: cmpq $0x118,%r12	disp+0x27: je *<disp+0xa2>*
disp+0x29: je *<disp+0x50>*	*disp+0x29*: movl $0x4,%edi
disp+0x2b: cmpq $0x370,%r12	disp+0x2e: nop
disp+0x32: je *<disp+0x42>*	disp+0x30: call func_default
disp+0x34: movl $0x4,%edi	*disp+0x35*: addq $0x8,%rsp
disp+0x39: xorl %eax,%eax	disp+0x39: movq %rbx,%rsi
disp+0x3b: call func_default	disp+0x3c: movl $printf_arg1,%edi
disp+0x40: jmp *<disp+0x7f>*	disp+0x41: popq %rbx
disp+0x42: movl $0x2,%edi	disp+0x42: leave
disp+0x47: xorl %eax,%eax	disp+0x43: xorl %eax,%eax
disp+0x49: call func_b	disp+0x45: jmp printf
disp+0x4e: jmp *<disp+0x7f>*	disp+0x4a: nop
disp+0x50: movl $0x1,%edi	disp+0x4d: nop
disp+0x55: xorl %eax,%eax	*disp+0x50*: cmpq $0x5c,%rdi
disp+0x57: call func_a	disp+0x54: je *<disp+0x93>*
disp+0x5c: jmp *<disp+0x7f>*	disp+0x56: cmpq $0x65,%rdi
disp+0x5e: movq %r12,%rdi	disp+0x5a: jne *<disp+0x29>*
disp+0x61: xorl %eax,%eax	disp+0x5c: movl $0x3,%edi
disp+0x63: call func_e	disp+0x61: call func_d
disp+0x68: jmp *<disp+0x7f>*	disp+0x66: jmp *<disp+0x35>*
disp+0x6a: movl $0x3,%edi	*disp+0x68*: movl $0xed,%edi
disp+0x6f: xorl %eax,%eax	disp+0x6d: call func_e
disp+0x71: call func_d	disp+0x72: addq $0x8,%rsp
disp+0x76: jmp *<disp+0x7f>*	disp+0x76: movq %rbx,%rsi
disp+0x78: xorl %eax,%eax	disp+0x79: movl $printf_arg1,%edi
disp+0x7a: call func_c	disp+0x7e: popq %rbx
disp+0x7f: leaq printf_arg1(%rip),%rdi	disp+0x7f: leave
disp+0x86: movq %r12,%rsi	disp+0x80: xorl %eax,%eax
disp+0x89: xorl %eax,%eax	disp+0x82: jmp printf
disp+0x8b: call printf	*disp+0x87*: movl $0x1,%edi
disp+0x90: popq %r12	disp+0x8c: call func_a
disp+0x92: movq %rbp,%rsp	disp+0x91: jmp *<disp+0x35>*
disp+0x95: popq %rbp	*disp+0x93*: call func_c
disp+0x96: ret	disp+0x98: nop
	disp+0x9c: nop
	disp+0xa0: jmp *<disp+0x35>*
	disp+0xa2: movl $0x2,%edi
	disp+0xa7: call func_b
	disp+0xac: nop
	disp+0xb0: jmp *<disp+0x35>*

Both compilers generated similar code compared to their 32bit output. Note that Workshop cc by used **%rip**-relative (*position independent*) for loading the address of **printf()**'s format string, while gcc uses *tail-call optimization* and **jmp**'s to **printf()**.

3.8.Accessing data structures

The various addressing modes suppported by x86 are frequently used in compiled code. They match very well with the way the C programming language implements structures and arrays. The following section will demonstrate:

- how global and local variables differ, and where arguments fit in
- what consequences to the assembly output it has if the compiler is instructed to create position-independent code.

Variations of the following sourcecode will be used:

```
#include <string.h>

typedef struct st_s {
        char    s_c;
        long    s_l;
        short   s_s;
        int     s_i;
        char    s_name[23];
        struct st_s *s_nxt;
} sts_t;

/*
 * Variant A:
 * structure to be initialized is passed as an argument
 */
void initstruct(
    sts_t *initme,
    char i_c,
    long i_l,
    short i_s,
    int i_i,
    char *i_name
)
{
        initme->s_c = i_c + 1;
        initme->s_l = i_l + 2;
        initme->s_s = i_s + 3;
        initme->s_i = i_i + 4;
        strcpy(initme->s_name, i_name);
        initme->s_name[20] = 'A'
        initme->s_nxt = NULL;
}
```

3.8.1.Arguments, types, structure access 32/64bit ABI comparison

The above-shown sourcecode is designed to show ABI differences. Since arguments are used/typed in a way simple enough to identify, any change in 32/64bit wrt. to:

- argument access
- data structure member alignment
- sizes of data types
- function calling

become immediately obvious.

A different color coding than before is used:

- red are (input) arguments
- green are local variables (on the stack – local variables in registers aren't marked)
- blue are global variables
- *thin italics red* are output arguments, i.e. values passed to a called function

32bit binary, Sun Workshop cc ***64bit binary, Sun Workshop cc***

```
initstruct:      pushl  %ebp             initstruct:      pushq  %rbp
initstruct+0x1:  movl   %esp,%ebp        initstruct+0x1:  movq   %rsp,%rbp
initstruct+0x3:  andl   $0xfffffff0,%esp initstruct+0x4:  subq   $0x8,%rsp
initstruct+0x6:  pushl  %ebx             initstruct+0x8:  pushq  %r12
initstruct+0x7:  movsbl 0xc(%ebp),%eax   initstruct+0xa:  movq   %rdi,%r12
initstruct+0xb:  incl   %eax             initstruct+0xd:  movq   %rdx,%r10
initstruct+0xc:  movl   0x8(%ebp),%ebx   initstruct+0x10: movsbl %sil,%eax
initstruct+0xf:  movb   %al,(%ebx)       initstruct+0x14: incl   %eax
initstruct+0x11: movl   0x10(%ebp),%eax  initstruct+0x16: movb   %al,(%r12)
initstruct+0x14: addl   $0x2,%eax        initstruct+0x1a: addq   $0x2,%r10
initstruct+0x17: movl   %eax,0x4(%ebx)   initstruct+0x1e: movq   %r10,0x8(%r12)
initstruct+0x1a: movswl 0x14(%ebp),%eax  initstruct+0x23: movswl %cx,%eax
initstruct+0x1e: addl   $0x3,%eax        initstruct+0x26: addl   $0x3,%eax
initstruct+0x21: movw   %ax,0x8(%ebx)    initstruct+0x29: movw   %ax,0x10(%r12)
initstruct+0x25: movl   0x18(%ebp),%eax  initstruct+0x2f: addl   $0x4,%r8d
initstruct+0x28: addl   $0x4,%eax        initstruct+0x33: movl   %r8d,0x14(%r12)
initstruct+0x2b: movl   %eax,0xc(%ebx)   initstruct+0x38: movq   %r12,%rdi
initstruct+0x2e: pushl  0x1c(%ebp)       initstruct+0x3b: addq   $0x18,%rdi
initstruct+0x31: leal   0x10(%ebx),%eax  initstruct+0x3f: movq   %r9,%rsi
initstruct+0x34: pushl  %eax             initstruct+0x42: xorl   %eax,%eax
initstruct+0x35: call <strcpy>           initstruct+0x44: call <strcpy>
initstruct+0x3a: addl   $0x8,%esp        initstruct+0x49: movb   $0x41,0x2c(%r12)
initstruct+0x3d: movb   $0x41,0x24(%ebx) initstruct+0x4f: movq   $0x0,0x30(%r12)
initstruct+0x41: movl   $0x0,0x28(%ebx)  initstruct+0x58: popq   %r12
initstruct+0x48: popl   %ebx             initstruct+0x5a: movq   %rbp,%rsp
initstruct+0x49: movl   %ebp,%esp        initstruct+0x5d: popq   %rbp
initstruct+0x4b: popl   %ebp             initstruct+0x5e: ret
initstruct+0x4c: ret
```

Differences:

1. Size of basic data types has changed. arg3 (of type **long**) is 32bit in 32bit mode but 64bit in 64bit mode. Sizes of pointers also changed, of course. ILP32 vs. LP64.
2. Alignment of basic types changed. **long** and **char*** (**s_l** and **s_name**) are 4-Byte aligned in 32bit but 8-Byte aligned in 64bit mode. Access to **sts_t** looks like:

```
struct st_s {              32bit access                64bit access
    char s_c;              movb %al,     (%ebx)        movb %al,     (%r12)
    long s_l;              movl %eax, 0x4(%ebx)        movq %r10, 0x8(%r12)
    short s_s;             movw %ax,  0x8(%ebx)        movw %ax, 0x10(%r12)
    int s_i;               movl %eax, 0xc(%ebx)        movl %r8d,0x14(%r12)
    char s_name[23];       leal      0x10(%ebx),%eax   addq      $0x18,%rdi
    sts_t *s_nxt;          movl $0x0,0x28(%ebx)        movq $0x0,0x30(%r12)
}
```

3. Argument passing – input arguments all on the stack in 32bit mode, first six in registers in 64bit mode:

argument	***32bit access***	***64bit access***
#0: **sts_t *initme**	`movl    0x8(%ebp),%ebx`	`movq    %rdi,%r12`

argument	***32bit access***	***64bit access***
#1: **`char i_c`**	`movsbl  0xc(%ebp),%eax`	`movsbl %sil,%eax`
#2: **`long i_l`**	`movl   0x10(%ebp),%eax`	`movq   %rdx,%r10`
#3: **`short i_s`**	`movswl 0x14(%ebp),%eax`	`movswl %cx,%eax`
#4: **`int i_i`**	`movl   0x18(%ebp),%eax`	`movl   %r8d,0x14(%r12)`
#5: **`char *i_name`**	*`pushl`* `0x1c(%ebp)`	`movq   %r9,`*`%rsi`*

4. Argument passing – output arguments (to **`strcpy()`**) all on the stack in 32bit mode, in **`%rsi`**/**`%rdi`** in 64bit mode.
   Additionally, 64bit mode clears **`%eax`** before making a function call.

In short, apart from the changes in argument passing we note that data type sizes have changed. 32bit x86 is ILP32 (**`int`**, **`long`**, and *pointers* all being 32bit), while the 64bit mode is LP64 (**`long`** and *pointers* are 64bit). The size change goes in line with a change in alignment of data structures – **`long`** and *pointer* types are aligned at a multiple of four bytes in 32bit x86, while in 64bit mode they're both at a multiple of eight bytes.

3.8.2.Argument access vs. local variables

The following code modifies the previously-shown sample program so that **`init_struct()`** initializes a local instance of sts_t before copying the contents thereof into the passed-in address:

```
void initstruct(
    sts_t *initme,
    char i_c,
    long i_l,
    short i_s,
    int i_i,
    char *i_name
)
{
        sts_t localcopy;

        localcopy.s_c = i_c + 1;
        localcopy.s_l = i_l + 2;
        localcopy.s_s = i_s + 3;
        localcopy.s_i = i_i + 4;
        strcpy(localcopy.s_name, i_name);
        localcopy.s_name[20] = 'A';
        localcopy.s_nxt = NULL;
        memcpy(initme, &localcopy, sizeof(sts_t));
}
```

This allows us to see how access to local variables and arguments differs in compiled code. Let's compare the previously-shown code that initializes the members of **`*initme`** directly with the new version that initializes a local instance and copies that to **`*initme`** at the end.

First, 32bit code:

initializing via arg, cc, 32bit	***using local variable, cc, 32bit***
`initstruct:       pushl  %ebp`	`initstruct:       pushl  %ebp`
`initstruct+0x1:   movl   %esp,%ebp`	`initstruct+0x1:   movl   %esp,%ebp`
`initstruct+0x3:   andl   $0xfffffff0,%esp`	`initstruct+0x3:   subl   $0x30,%esp`
`initstruct+0x6:   pushl  %ebx`	`initstruct+0x6:   andl   $0xfffffff0,%esp`
`initstruct+0x7:   movsbl 0xc(%ebp),%eax`	`initstruct+0x9:   movsbl 0xc(%ebp),%eax`
`initstruct+0xb:   incl   %eax`	`initstruct+0xd:   incl   %eax`
`initstruct+0xc:   movl   0x8(%ebp),%ebx`	`initstruct+0xe:   movb   %al,0x4(%esp)`
`initstruct+0xf:   movb   %al,(%ebx)`	`initstruct+0x12:  movl   0x10(%ebp),%eax`
`initstruct+0x11:  movl   0x10(%ebp),%eax`	`initstruct+0x15:  addl   $0x2,%eax`
`initstruct+0x14:  addl   $0x2,%eax`	`initstruct+0x18:  movl   %eax,0x8(%esp)`
`initstruct+0x17:  movl   %eax,0x4(%ebx)`	`initstruct+0x1c:  movswl 0x14(%ebp),%eax`
`initstruct+0x1a:  movswl 0x14(%ebp),%eax`	`initstruct+0x20:  addl   $0x3,%eax`
`initstruct+0x1e:  addl   $0x3,%eax`	`initstruct+0x23:  movw   %ax,0xc(%esp)`
`initstruct+0x21:  movw   %ax,0x8(%ebx)`	`initstruct+0x28:  movl   0x18(%ebp),%eax`
`initstruct+0x25:  movl   0x18(%ebp),%eax`	`initstruct+0x2b:  addl   $0x4,%eax`
`initstruct+0x28:  addl   $0x4,%eax`	`initstruct+0x2e:  movl   %eax,0x10(%esp)`
`initstruct+0x2b:  movl   %eax,0xc(%ebx)`	`initstruct+0x32:  leal   0x14(%esp),%eax`
`initstruct+0x2e:  pushl  0x1c(%ebp)`	`initstruct+0x36:  pushl  0x1c(%ebp)`
`initstruct+0x31:  leal   0x10(%ebx),%eax`	`initstruct+0x39:  pushl  %eax`
`initstruct+0x34:  pushl  %eax`	`initstruct+0x3a:  call <strcpy>`
`initstruct+0x35:  call <strcpy>`	`initstruct+0x3f:  addl   $0x8,%esp`
`initstruct+0x3a:  addl   $0x8,%esp`	`initstruct+0x42:  movb   $0x41,0x28(%esp)`
`initstruct+0x3d:  movb   $0x41,0x24(%ebx)`	`initstruct+0x47:  movl   $0x0,0x2c(%esp)`
`initstruct+0x41:  movl   $0x0,0x28(%ebx)`	`initstruct+0x4f:  pushl  $0x2c`
`initstruct+0x48:  popl   %ebx`	`initstruct+0x51:  leal   0x8(%esp),%eax`
`initstruct+0x49:  movl   %ebp,%esp`	`initstruct+0x55:  pushl  %eax`
`initstruct+0x4b:  popl   %ebp`	`initstruct+0x56:  movl   0x8(%ebp),%eax`
`initstruct+0x4c:  ret`	`initstruct+0x59:  pushl  %eax`
	`initstruct+0x5a:  call <memcpy>`
	`initstruct+0x5f:  movl   %ebp,%esp`
	`initstruct+0x61:  popl   %ebp`
	`initstruct+0x62:  ret`

This shows that the workshop compiler, in this case, has chosen to access *arguments* via **`+...(%ebp)`**, i.e. relative to the *beginning* of the function's stackframe, while the second version shows it accesses *local variables* via **`+...(%esp)`**, relative to the *end* of the stackframe.

While arguments in 32bit mode, if there is a framepointer, will always be accessed via **`+...(%ebp)`**, it's common to see both **`+...(%esp)`** and **`-...(%ebp)`** for local variables.

The following comparison between 32bit cc/gcc output shows this clearly – the Sun Workshop compiler for the given example code has generated assembly which accesses local variables relative to the stackpointer, **`+...(%esp)`**, while gcc for the same sources creates assembly code that uses negative offsets relative to the framepointer, **`-...(%ebp)`** for the same.

No matter what – in 32bit code that uses framepointers, the following is always true:

- *Arguments*: before the start of the stackframe, and accessed via **`+8(%ebp)`** onwards.
- *Local variables*: Below **`%ebp`**, above **`%esp`**. Access via **`-...(%ebp)`** or **`+...(%esp)`**.

local variables, 32bit, Workshop cc	***local variables, 32bit, GNU gcc***
`initstruct:      pushl  %ebp`	`initstruct:      pushl %ebp`
`initstruct+0x1:  movl   %esp,%ebp`	`initstruct+0x1:  movl  %esp,%ebp`
`initstruct+0x3:  subl   $0x30,%esp`	`initstruct+0x3:  pushl %ebx`
`initstruct+0x6:  andl   $0xfffffff0,%esp`	`initstruct+0x4:  subl  $0x3c,%esp`
`initstruct+0x9:  movsbl 0xc(%ebp),%eax`	`initstruct+0x7:  movb  0xc(%ebp),%al`
`initstruct+0xd:  incl   %eax`	`initstruct+0xa:  incl  %eax`
`initstruct+0xe:  movb   %al,0x4(%esp)`	`initstruct+0xb:  movb  %al,-0x38(%ebp)`
`initstruct+0x12: movl   0x10(%ebp),%eax`	`initstruct+0xe:  movl  0x10(%ebp),%eax`
`initstruct+0x15: addl   $0x2,%eax`	`initstruct+0x11: addl  $0x2,%eax`
`initstruct+0x18: movl   %eax,0x8(%esp)`	`initstruct+0x14: movl  %eax,-0x34(%ebp)`
`initstruct+0x1c: movswl 0x14(%ebp),%eax`	`initstruct+0x17: movl  0x14(%ebp),%eax`
`initstruct+0x20: addl   $0x3,%eax`	`initstruct+0x1a: addl  $0x3,%eax`
`initstruct+0x23: movw   %ax,0xc(%esp)`	`initstruct+0x1d: movw  %ax,-0x30(%ebp)`
`initstruct+0x28: movl   0x18(%ebp),%eax`	`initstruct+0x21: movl  0x18(%ebp),%eax`
`initstruct+0x2b: addl   $0x4,%eax`	`initstruct+0x24: addl  $0x4,%eax`
`initstruct+0x2e: movl   %eax,0x10(%esp)`	`initstruct+0x27: pushl 0x1c(%ebp)`
`initstruct+0x32: leal   0x14(%esp),%eax`	`initstruct+0x2a: movl  %eax,-0x2c(%ebp)`
`initstruct+0x36: pushl  0x1c(%ebp)`	`initstruct+0x2d: leal  -0x28(%ebp),%eax`
`initstruct+0x39: pushl  %eax`	`initstruct+0x30: pushl %eax`
`initstruct+0x3a: call <strcpy>`	`initstruct+0x31: call <strcpy>`
`initstruct+0x3f: addl   $0x8,%esp`	`initstruct+0x36: addl  $0xc,%esp`
`initstruct+0x42: movb   $0x41,0x28(%esp)`	`initstruct+0x39: pushl $0x2c`
`initstruct+0x47: movl   $0x0,0x2c(%esp)`	`initstruct+0x3b: leal  -0x38(%ebp),%ebx`
`initstruct+0x4f: pushl  $0x2c`	`initstruct+0x3e: pushl %ebx`
`initstruct+0x51: leal   0x8(%esp),%eax`	`initstruct+0x3f: pushl 0x8(%ebp)`
`initstruct+0x55: pushl  %eax`	`initstruct+0x42: movb  $0x41,-0x14(%ebp)`
`initstruct+0x56: movl   0x8(%ebp),%eax`	`initstruct+0x46: movl  $0x0,-0x10(%ebp)`
`initstruct+0x59: pushl  %eax`	`initstruct+0x4d: call <memcpy>`
`initstruct+0x5a: call <memcpy>`	`initstruct+0x52: addl  $0x10,%esp`
`initstruct+0x5f: movl   %ebp,%esp`	`initstruct+0x55: movl  -0x4(%ebp),%ebx`
`initstruct+0x61: popl   %ebp`	`initstruct+0x58: leave`
`initstruct+0x62: ret`	`initstruct+0x59: ret`

The 64bit version doesn't show conceptual differences, what holds true for local variables in 32bit mode still applies to 64bit mode.

3.8.3.Global variables

Again changing the sourcecode slightly:

```
extern sts_t glob_st;

void initstruct(
    sts_t *i_nxt,
    char i_c,
    long i_l,
    short i_s,
    int i_i,
    char *i_name
)
{
        glob_st.s_c = i_c + 1;
        glob_st.s_l = i_l + 2;
        glob_st.s_s = i_s + 3;
        glob_st.s_i = i_i + 4;
        strcpy(glob_st.s_name, i_name);
        glob_st.s_name[20] = 'A';
        glob_st.s_nxt = i_nxt;
}
```

This example serves well to show the difference between local variables (on the stack) and global variables (on the heap). It also illustrates the difference between position-independent and regular code for accessing globals.

Sun Workshop cc, 32bit, non-PIC	***Sun Workshop cc, 32bit, -Kpic***
`initstruct:      pushl  %ebp`	`initstruct: pushl  %ebp`
`initstruct+0x1:  movl   %esp,%ebp`	`+0x1:  movl   %esp,%ebp`
`initstruct+0x3:  andl   $0xfffffff0,%esp`	`+0x3:  andl   $0xfffffff0,%esp`
`initstruct+0x6:  movsbl 0xc(%ebp),%eax`	`+0x6:  pushl  %ebx`
`initstruct+0xa:  incl   %eax`	`+0x7:  pushl  %esi`
`initstruct+0xb:  movb   %al,<g_s>`	`+0x8:  call <initstruct+0xd>`
`initstruct+0x10: movl   0x10(%ebp),%eax`	`+0xd:  popl   %ebx`
`initstruct+0x13: addl   $0x2,%eax`	`+0xe:  addl   $_GLOBAL_OFFSET_TABLE_+1, %ebx`
`initstruct+0x16: movl   %eax,<g_s+0x4>`	`+0x14: movsbl 0xc(%ebp),%eax`
`initstruct+0x1b: movswl 0x14(%ebp),%eax`	`+0x18: incl   %eax`
`initstruct+0x1f: addl   $0x3,%eax`	`+0x19: movl   g_s@GOT(%ebx),%esi`
`initstruct+0x22: movw   %ax,<g_s+0x8>`	`+0x1f: movb   %al,(%esi)`
`initstruct+0x28: movl   0x18(%ebp),%eax`	`+0x21: movl   0x10(%ebp),%eax`
`initstruct+0x2b: addl   $0x4,%eax`	`+0x24: addl   $0x2,%eax`
`initstruct+0x2e: movl   %eax,<g_s+0xc>`	`+0x27: movl   %eax,0x4(%esi)`
`initstruct+0x33: pushl  0x1c(%ebp)`	`+0x2a: movswl 0x14(%ebp),%eax`
`initstruct+0x36: pushl  $<g_s+0x10>`	`+0x2e: addl   $0x3,%eax`
`initstruct+0x3b: call <strcpy>`	`+0x31: movw   %ax,0x8(%esi)`
`initstruct+0x40: addl   $0x8,%esp`	`+0x35: movl   0x18(%ebp),%eax`
`initstruct+0x43: movb   $0x41,0x24`	`+0x38: addl   $0x4,%eax`
`initstruct+0x4a: movl   0x8(%ebp),%eax`	`+0x3b: movl   %eax,0xc(%esi)`
`initstruct+0x4d: movl   %eax,<g_s+0x28>`	`+0x3e: pushl  0x1c(%ebp)`
`initstruct+0x52: movl   %ebp,%esp`	`+0x41: leal   0x10(%esi),%eax`
`initstruct+0x54: popl   %ebp`	`+0x44: pushl  %eax`
`initstruct+0x55: ret`	`+0x45: call <strcpy>`
	`+0x4a: addl   $0x8,%esp`
	`+0x4d: movb   $0x41,0x24(%esi)`
	`+0x51: movl   0x8(%ebp),%eax`
	`+0x54: movl   %eax,0x28(%esi)`
	`+0x57: popl   %esi`

Sun Workshop cc, 32bit, non-PIC	***Sun Workshop cc, 32bit, -Kpic***
	`+0x58: popl   %ebx`
	`+0x59: movl   %ebp,%esp`
	`+0x5b: popl   %ebp`
	`+0x5c: ret`

Regular code (not explicitly position-independent) in 32bit x86 uses *direct addressing* (i.e. 32bit pointers) to access global variables.

The key point to understanding position-independent code in 32bit is the *Global Offset Table* (GOT). This is the memory location where PIC code stores global data structures. We see in the code:

1. Since PIC code can be put anywhere into memory without the dynamic linker/loader having to do relocation, a PIC function will need to:
   - determine its own load address. The shown code does this via:
     `call <initstruct+0xd>`
     `popl   %ebx`
     This "call yourself" is typical for position-independent code on architectures that don't allow the use of the program counter (**`%eip`** in 32bit x86) for relative addressing. The code simply reads out the current program counter.
   - add the offset between code location and GOT location to that. That's why it does:
     `addl   $_GLOBAL_OFFSET_TABLE_+1,%ebx`
     This code puts the base address of the Global Offset Table into **`%ebx`** – and it will be kept there over the lifetime of this function (at least).
2. Once the GOT location has been determined, addresses of global variables that are now within the GOT can be calculated relative to %ebx. We see in the example:
   `movl   g_s@GOT(%ebx),%esi`
   After that, the start address of the global data structure is known. Members within are updated relative to **`+...(%esi)`**.

Due to this overhead, and the need to keep the GOT address in **`%ebx`** (loosing one general-purpose register), PIC code in 32bit x86 is significantly slower than non-PIC.

64bit x86 on the other hand knows **`%rip`**-relative addressing. This simplifies PIC code so much that both gcc and Sun Workshop cc *will always create position-independent code* for 64bit x86, even without explicitly requesting it:

64bit, Sun Workshop cc	***64bit, GNU gcc***
`initstruct:       pushq  %rbp`	`initstruct:       pushq  %rbp`
`initstruct+0x1:   movq   %rsp,%rbp`	`initstruct+0x1:   incl   %esi`
`initstruct+0x4:   subq   $0x8,%rsp`	`initstruct+0x3:   addq   $0x2,%rdx`
`initstruct+0x8:   pushq  %r12`	`initstruct+0x7:   addl   $0x3,%ecx`
`initstruct+0xa:   movq   %rdi,%r12`	`initstruct+0xa:   addl   $0x4,%r8d`
`initstruct+0xd:   movq   %rdx,%r10`	`initstruct+0xe:   movq   %rsp,%rbp`
`initstruct+0x10:  movsbl %sil,%eax`	`initstruct+0x11:  pushq  %rbx`
`initstruct+0x14:  incl   %eax`	`initstruct+0x12:  movq   %rdi,%rbx`
`initstruct+0x16:  movb   %al,g_s(%rip)`	`initstruct+0x15:  movq   $g_s+0x18,%rdi`
`initstruct+0x1c:  addq   $0x2,%r10`	`initstruct+0x1c:  subq   $0x8,%rsp`
`initstruct+0x20:  movq   %r10,`	`initstruct+0x20:  movb   %sil,g_s(%rip)`
`                         g_s+0x8(%rip)`	`initstruct+0x27:  movq   %r9,%rsi`
`initstruct+0x27:  movswl %cx,%eax`	`initstruct+0x2a:  movq   %rdx,`
`initstruct+0x2a:  addl   $0x3,%eax`	`                         g_s+0x8(%rip)`
`initstruct+0x2d:  movw   %ax,`	`initstruct+0x31:  movw   %cx,`
`                         g_s+0x10(%rip)`	`                         g_s+0x10(%rip)`
`initstruct+0x34:  addl   $0x4,%r8d`	`initstruct+0x38:  movl   %r8d,`

64bit, Sun Workshop cc	*64bit, GNU gcc*
`initstruct+0x38: movl   %r8d,`	`                        g_s+0x14(%rip)`
`                        g_s+0x14(%rip)`	`initstruct+0x3f: call <strcpy>`
`initstruct+0x3f: leaq   g_s+0x18(%rip),`	`initstruct+0x44: movq   %rbx,`
`                        %rdi`	`                        g_s+0x30(%rip)`
`initstruct+0x46: movq   %r9,%rsi`	`initstruct+0x4b: movb   $0x41,`
`initstruct+0x49: xorl   %eax,%eax`	`                        g_s+0x2c(%rip)`
`initstruct+0x4b: call <strcpy>`	`initstruct+0x52: addq   $0x8,%rsp`
`initstruct+0x50: movb   $0x41,`	`initstruct+0x56: popq   %rbx`
`                        g_s+0x2c(%rip)`	`initstruct+0x57: leave`
`initstruct+0x57: movq   %r12,`	`initstruct+0x58: ret`
`                        g_s+0x30(%rip)`	
`initstruct+0x5e: popq   %r12`	
`initstruct+0x60: movq   %rbp,%rsp`	
`initstruct+0x63: popq   %rbp`	
`initstruct+0x64: ret`	

The Sun Workshop cc code is fully PIC, while gcc's code generator probably contains a small bug in this case. The instruction used by gcc:

```
initstruct+0x15: movq   $g_s+0x18,%rdi
```

which loads the first argument for strcpy() is NOT using **%rip**-relative addressing. The version the Sun Workshop compiler uses:

```
initstruct+0x3f: leaq   g_s+0x18(%rip),%rdi
```

achieves the same – but in the desirable position-independent way.

For Sun Workshop cc, (not) using **-kpic** makes no difference to the generated code in 64bit x86. For GNU gcc, **-fpic** resorts to the (bad) 32bit-style behaviour of referencing global data relative to **+...(%ebx)**, and only replaces the load of the GOT base address with **%rip**-relative addressing.

3.9.Compiler help for debugging AMD64 code

The biggest difference between AMD64 and i386 assembly code is the way arguments are passed.

- i386 passes arguments on the stack, in descending order.
  A function (or a debugger, post-mortem) can access its arguments relative to the framepointer, at **`4*arg`$_N$`(%ebp)`**.
- AMD64 passes the first six arguments in registers **`%rdi`**, **`%rsi`**, **`%rdx`**, **`%rcx`**, **`%r8`** and **`%r9`**. Only arguments past the sixth end up on the stack, at **`8*arg`$_{N-6}$`(%rbp)`**.
  It's up to the function where it keeps its arguments, i.e. whether it leaves them in the argument registers, moves them to nonvolatile registers, or puts them onto the stack.
  A debugger has no simple way of figuring out where in the stack the arguments finally end up (if at all).

Source-level debugging for applications of course solves this problem, since complete source-level debugging data contains the information about stackframe layout. But this is too heavyweight for many purposes, and inappropriate for kernel debugging.

The section on “Register Lifecycle” has already shown that argument passing in registers doesn't necessarily remove the need to have copies of these arguments in the stack *somewhere* during the “lifetime” of the function to which a given argument was passed to. In the vast majority of cases, anything but very simple wrapper functions will end up having their arguments in the stack, because:

1. The function uses its argument(s) again after having called another function itself and therefore needs to make them “permanent” - making function calls destroys the contents of the argument registers.
2. The function's active working set of variables is larger than the number of available registers, and it therefore moves the arguments out of the way in order to use the argument registers for other purposes.

So there often is need for AMD64 assembly code to *save* its arguments into the stack *somewhere*. And this can be taken advantage of.

In order to make this usable for a debugger to find function arguments automatically, we need to instruct the compiler to:

- make *somewhere* into a *defined place within the stackframe* of the function
- *always* perform argument saving at entry to the function.

In fact, Microsoft for the Windows/x64 ABI has specified this behaviour, by defining that the first four arguments are passed in registers, but the calling function also has to reserve stackspace for them. I.e. the Windows/x64 ABI specifies: “If you want to save your argument registers to the stack, save them *right here* - the caller has allocated this space for you...”

The UNIX ABI for AMD64 does not mandate such behaviour. Implementing it is therefore up to the compiler (which is not required to do it) – and the space where the arguments will be put in must be in the “local variables” section of the stackframe, not in the “arguments” section as on Windows/x64, because the prerequisite for the caller to reserve space for the arguments doesn't exist on UN*X/AMD64.

Recent UN*X compilers for x64 (Sun Workshop 10.1, and gcc via patches since August 2005) can be instructed to create function prologue code that saves argument registers into the stack, at the beginning of a function's stackframe.

The big problem one is faced with when debugging AMD64 code on assembly level,

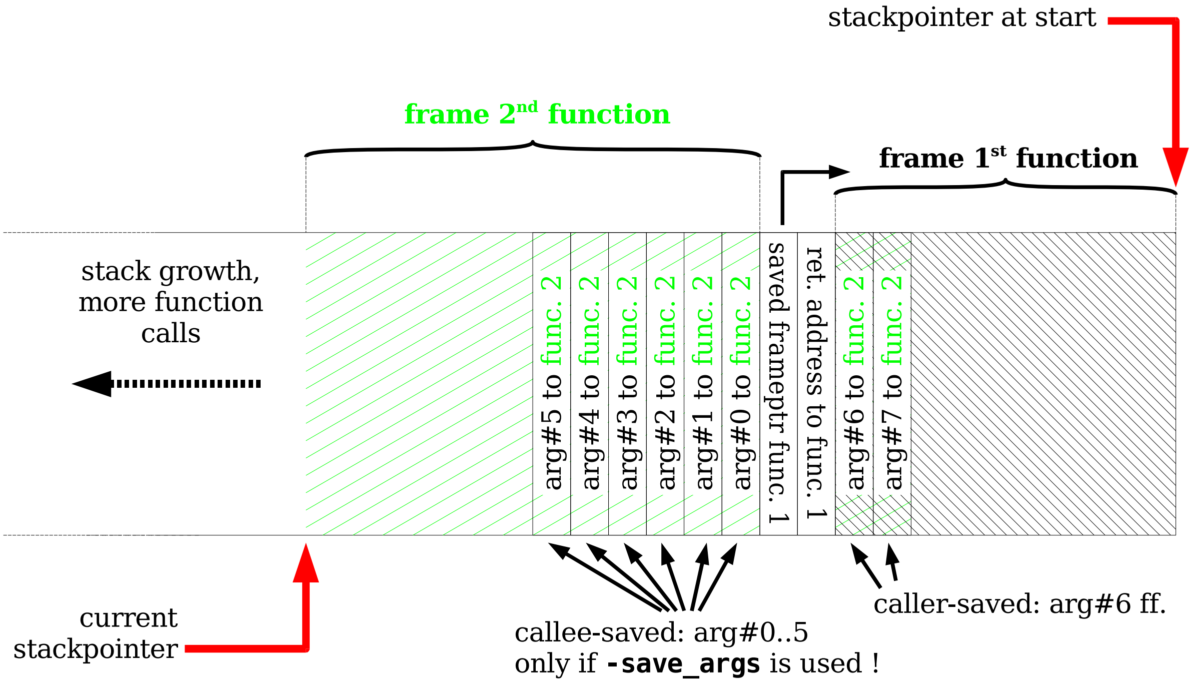


Illustration 12 - AMD64 Stack Layout, requesting the compiler to save args to the stack

namely finding arguments in stacktraces, becomes almost as trivial as with classical 32bit i386 code - when the **`-save_args`** compiler option (**`-msave_args`** on gcc) has been used.

All arguments are accessible at known locations relative to the framepointer for the function:

argument	***32bit / i386***	**`-save_args,`** ***64bit / amd64***	
first (0)	**`0x8(%ebp)`**	**`-0x8(%rbp)`**	**`%rdi`**
second (1)	**`0xc(%ebp)`**	**`-0x10(%rbp)`**	**`%rsi`**
third (2)	**`0x10(%ebp)`**	**`-0x18(%rbp)`**	**`%rdx`**
fourth (3)	**`0x14(%ebp)`**	**`-0x20(%rbp)`**	**`%rcx`**
fifth (4)	**`0x18(%ebp)`**	**`-0x28(%rbp)`**	**`%r8`**
sixth (5)	**`0x1c(%ebp)`**	**`-0x30(%rbp)`**	**`%r9`**
seventh (7)	**`0x20(%ebp)`**	**`0x10(%rbp)`**	
eigth (7)	**`0x24(%ebp)`**	**`0x18(%rbp)`**	
... (N)	*`0x4*(N+1)`***`(%ebp)`**	*`0x8*(N-5)`***`(%rbp)`**	

Compiling code with **`-save_args`** allows backtraces with arguments even from simple debuggers that don't do code flow tracing to locate the arguments.

3.9.1.Code samples with and without -save_args

As an illustration how this behaviour changes function prologues, let's look at an example.

amd64 binary, with **`-save_args`**	***amd64 binary, “classical”***
`hsfs_getpage:        pushq %rbp`	`hsfs_getpage:       pushq %rbp`
`hsfs_getpage+1:      movq %rsp,%rbp`	`hsfs_getpage+1:     movq %rsp,%rbp`
`hsfs_getpage+4:      subq $0x28,%rsp`	`hsfs_getpage+4:     subq $0x58,%rsp`
`hsfs_getpage+8:      pushq %rbx`	`hsfs_getpage+8:     pushq %rbx`
`hsfs_getpage+9:      pushq %r12`	`hsfs_getpage+9:     pushq %r12`
`hsfs_getpage+0xb:    pushq %r13`	`hsfs_getpage+0xb:   pushq %r13`
`hsfs_getpage+0xd:    pushq %r14`	`hsfs_getpage+0xd:   pushq %r14`
`hsfs_getpage+0xf:    pushq %r15`	`hsfs_getpage+0xf:   pushq %r15`
`hsfs_getpage+0x11:   movq %rdi,-0x8(%rbp)`	
`hsfs_getpage+0x15:   movq %rdi,%r14`	`hsfs_getpage+0x11: movq %rdi,%r14`
`hsfs_getpage+0x18:   movq %rsi,-0x10(%rbp)`	`hsfs_getpage+0x14: movq %rsi,%r13`
`hsfs_getpage+0x1c:   movq %rsi,%r13`	
`hsfs_getpage+0x1f:   movq %rdx,-0x18(%rbp)`	`hsfs_getpage+0x17: movq %rdx,%r12`
`hsfs_getpage+0x23:   movq %rdx,%r12`	`hsfs_getpage+0x1a: movq %rcx,%r15`
`hsfs_getpage+0x26:   movq %rcx,-0x20(%rbp)`	`hsfs_getpage+0x1d: movq %r8,-0x8(%rbp)`
`hsfs_getpage+0x2a:   movq %rcx,%r15`	
`hsfs_getpage+0x2d:   movq %r8,-0x28(%rbp)`	`hsfs_getpage+0x21: movq %r9,-0x18(%rbp)`
`hsfs_getpage+0x31:   movq %r8,-0x38(%rbp)`	`hsfs_getpage+0x25: movq 0x10(%r14),%rbx`
`hsfs_getpage+0x35:   movq %r9,-0x30(%rbp)`	`hsfs_getpage+0x29: movl 0x20(%rbp),%eax`
`hsfs_getpage+0x39:   movq %r9,-0x48(%rbp)`	*[...]*
`hsfs_getpage+0x3d:   movq 0x10(%r14),%rbx`	`hsfs_getpage+0xe3: movq 0x10(%rbp),%r8`
`hsfs_getpage+0x41:   movl 0x20(%rbp),%eax`	`hsfs_getpage+0xe7: movq 0x18(%rbp),%r9`
[...]	`hsfs_getpage+0xae: movq 0x28(%rbp),%r10`
`hsfs_getpage+0xfb:   movq 0x10(%rbp),%r8`	*[...]*
`hsfs_getpage+0xff:   movq 0x18(%rbp),%r9`	
`hsfs_getpage+0x103: movq 0x28(%rbp),%r10`	
[...]	

Code generated by the same compiler without and with the **`-save_args`** option only differs in

- the amount of space allocated on the stack
  (6 more words, 0x30 bytes for the six argument registers),
- the instructions that save the argument registers into the stack
- the **`%rbp`**-relative offsets used for local variables, which are shifted by **`0x30`**.

4.Memory and Privilege Management on x86

From “The Tao of Programming”:

> *Thus spake the master programmer:*
> *"When program is being tested, it is too late to make design changes."*

Advanced operating systems separate several concurrently running applications from each other, and keep the operating system kernel isolated from applications.

This puts two basic requirements on the CPU:

1. Configurable address spaces (“Contexts”). Two applications may not access the same memory unless an explicit “sharing” request was made. And likewise, applications may not usually access the operating system kernel's data, nor should unless deliberately requested the kernel modify memory in use by a given application.
   In short, modern operating systems require a MMU.
2. Privileged execution. Applications cannot be allowed to modify CPU state that is critical to the operating system. Only the kernel and its device drivers know how to access hardware, and only the kernel may grant memory/device access to applications via controlled interfaces. In other words, we need:
   - Privilege levels.
     The kernel will run in privileged mode and have access to all CPU instructions / all hardware including those that are crucial to system integrity.
     Applications, on the other hand, will run unprivileged and be prevented from “messing up” the system.
   - Privilege switching mechanisms.
     Controlled interfaces (system calls, exceptions, interrupts) are required to pass execution from unprivileged mode (applications) to privileged (kernel).

CPUs in the x86 family provide the abovementioned features if operating in the *Protected Mode*.

4.1.The x86 protected mode – privilege management

Intel was late to introduce a model for program execution at different privilege levels.

Not until 1982 when the 80286 was released did Intel-compatible CPUs know about a fully-functional model for stopping *user code* from executing arbitrary instructions or performing other critical actions that could compromise e.g. the integrity of an operating systems. The concept of *user/supervisor mode* that other architectures had for a long time already even back in 1982 has not been a designed part of the x86 architecture.

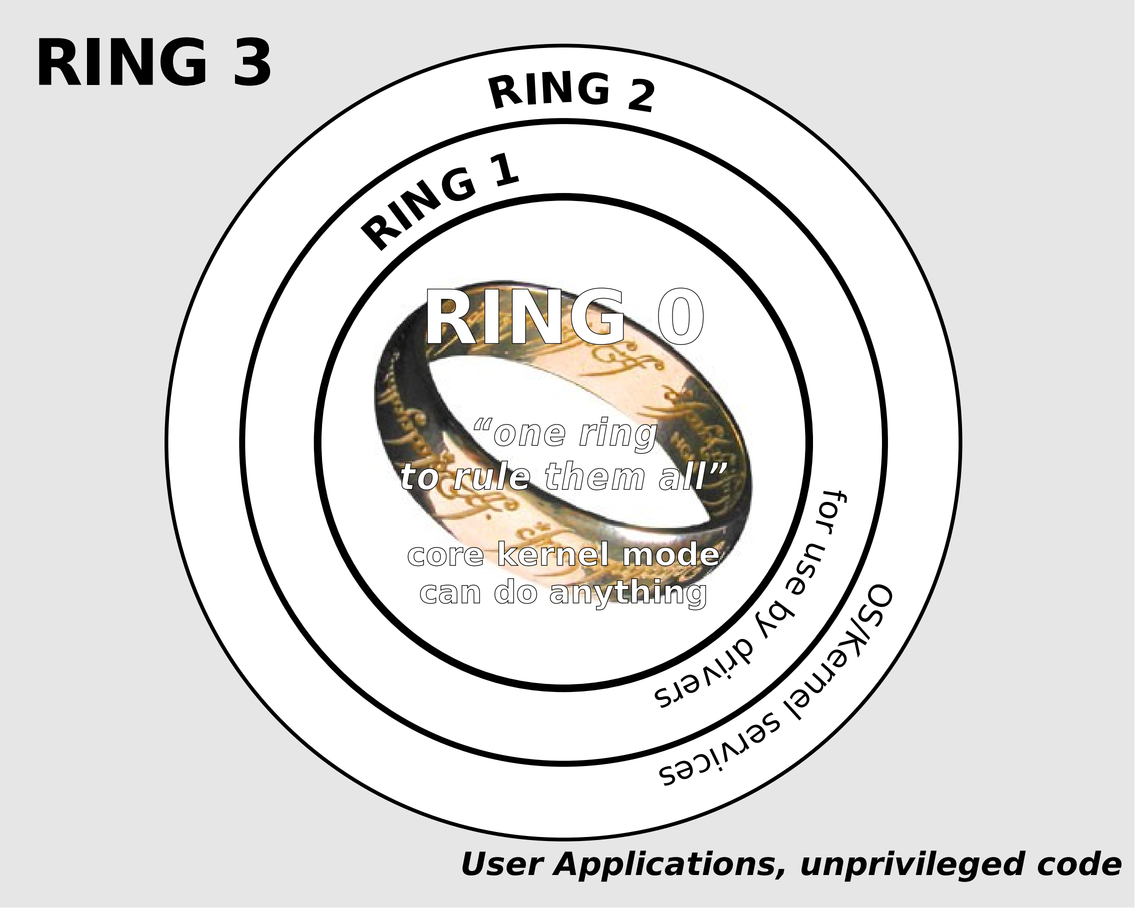


Illustration 1 - Protected Mode usage, as imagined by Intel

When Intel finally implemented privileges they chose a multi-layer model for it, allowing for two intermediate privilege levels between *user mode* and *fully-privileged supervisor mode*.

Intel calls this model *Protected Mode*. It theoretically allows for fine-grained control about who can execute instructions that modify critical system state, or modify memory/peripheral state whose consistency is crucial for the integrity of an operating system.

The Intel term for these execution modes is *Rings*. There are four:

- ***Ring 0***
  This mode allows unrestricted access to all CPU/hardware capabilities/resources.
- Ring 1
  This is meant to be used for drivers that can get by with unrestricted access to a

limited set of CPU/hardware capabilities/resources.

- Ring 2
  This is supposed to be used by services of the operating system which can be implemented e.g with the use of privileged instructions but without privileged access to hardware/memory on a large scale.
- ***Ring 3***
  Application programs that need strong isolation against each other will use ring 3.

Intel wasn't the first to introduce a fine-privileges execution model - a US military CPU design from the late 70s (MIL-STD-1750) has 16 (!) privilege levels. It never gained momentum. On paper, the idea of building *sandboxes* even for operating-system code has some benefits. But the actual hardware implementation for the US military's CPU was too complex/heavyweight for its time, while the stripped-down model by Intel reached market far too late. Intel introduced rings of execution at a time when most operating systems were already designed around hardware that only had the all-or-nothing user/supervisor model. That's one reason why all modern operating systems for x86 collapse ring 0..2 and use only two modes:

- ***Ring 0*** - operating system, drivers
- ***Ring 3*** - application programs

Additionally, what may well have contributed to the fact that the fine-grained privilege control of the Protected Mode was never really used to its full capabilities is the actual implementation of these features.

4.1.1.Evolution of privilege and memory management on x86

The 8086 16bit CPU had no concept of privilege whatsoever. Every code was allowed to use all instructions and modify CPU state that normally would be considered to “belong” to an operating system only.

But the 8086 had a rudimentary memory management concept. The CPU and all its registers, as well as direct address offsets, were 16bit only (64kB) but yet the processor could handle 1MB of physical memory.

Many 16bit CPUs had such capabilities – implemented via a mechanism usually called *bank switching*. One would program a memory controller register with “the bank of memory” (i.e. the 64kB window out of the larger physical addressing range) that 16bit memory accesses could map to.

This is inflexible if multiple 16bit chunks are concurrently in use. Consider copying memory: switch to bank A, move 16bit from mem to register, switch to bank B, move register to mem, switch to bank A, ...

Intel therefore provided *segmentation* and in the 8086. This does mean nothing else than subdividing the >16bit physical address space into an array of 16bit (64kB) banks which Intel decided to call *segments*, and making multiple of these accessible concurrently by the use of *segment registers*. In other words: The 8086's memory controller supported multiple banks/segments of memory at the same time.

Three of them use very common terms:

- code segment, **`%cs`** - instruction addresses (code locations).
- data segment, **`%ds`** - data addresses (direct or indirect via **`%ax`**/**`%bx`**/**`%cx`**/**`%dx`**)
- stack segment, **`%ss`** - the stack (base for **`%sp`**/**`%bp`**-relative addresses)

The fourth one was associated with specific instructions implementing many of the **mem.../str...** function family in hardware:

- string segment, **%es** - data addresses (indirect via string src/dest **%si**/**%di**)

In the 8086, the segment registers would simply contain the high 4 bit of the physical address, and the mapping between segment ID (contents of a segment register) and physical memory location of the 64bit segment was static 1:1.

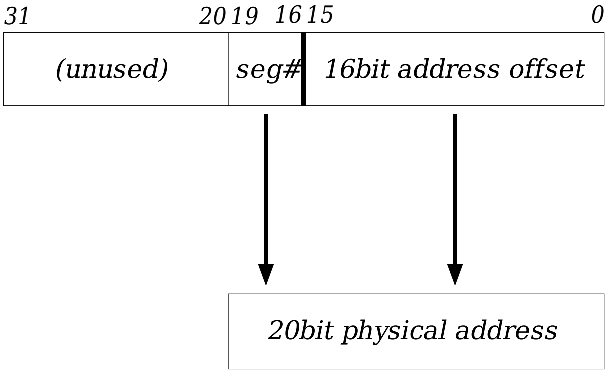


Illustration 2 - “memory management” on the 8086 processor

In addition to the above *implicit* use of the segment registers, the 8086 allowed explicit segment overrides. This is useful for segment-to-segment copy, which was then done using a method like:

```
movw  %es:(%dx),%ax
movw  %ax,(%dx)
```

which copies the value at offset **%dx** from the (non-default, therefore **%es:** prefix) string segment to the default data segment. As said, unlike other 16bit CPUs, no bank switching was needed.

The notation **segment:offset** is called *far pointer* by Intel.

The segmentation mechanism in the 8086 wasn't programmable, i.e. an operating system couldn't relocate segments. Of course, due to the lack of privileges neither was there a concept of write protection or privileged-only access.

Intel provided these capabilities in the 80286 via the *Protected Mode*. The protected mode:

- replaced the static segment numbers that were the contents of segment registers on the 8086 with so-called *segment selectors*.
- made the association between segments and physical memory programmable via mapping tables – the segment numbers now index a table, and the entry there provides the actual segment base address.
- allowed “requestors” (code using the segment register) to specify a privilege level for memory access through this segment (selector)
- allowed “requestors” to select which of the two translation tables (local or global) the CPU should use to get the physical location of the segment.

This is, in generic terms, of course a memory management unit with two contexts. Since the CPU was still 16bit and therefore no more than 64kB were accessible without using the segmentation mechanism, this seemed like a straightforward way of implementing virtual memory management capabilities.

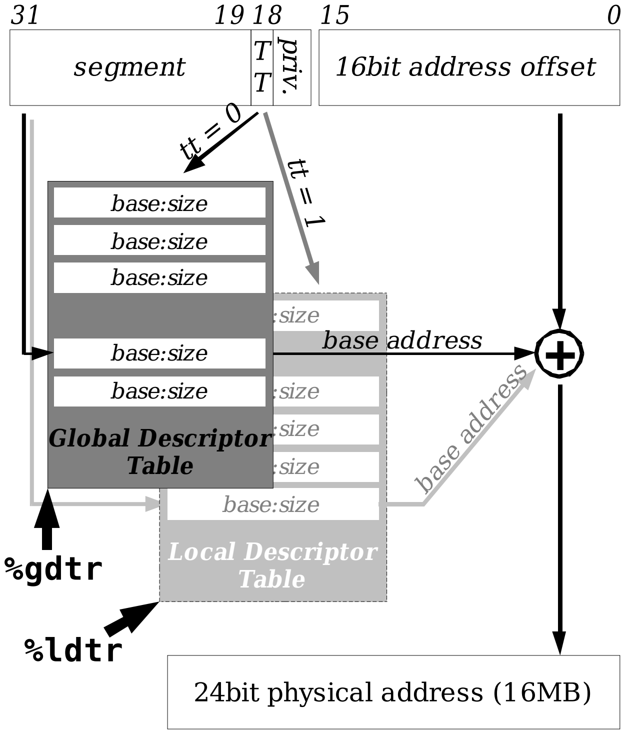


Illustration 3 - 16bit protected mode on the i80286

But unfortunately Intel tied CPU privilege management to these *Descriptor Tables*. These are not only used for programmable segmentation as memory management mechanism, but the descriptor table entries, called *segment descriptors*, are also used for privilege management. Segment descriptors:

- declare the segment privilege level (i.e. which ring this is accessible from)
- define access attributes (present, r/w or readonly, executable, ...)
- are *aliased* to data structures called *gate descriptors* that redirect execution between code segments (i.e. executable segments) of different privilege levels.

This leads to the odd situation that x86 descriptor tables contain two very different kinds of entries:

- *segment descriptors* (which provide base address, size and privilege level) for memory management *and* privilege declaration, and
- *gate descriptors* (which supply a **`call`** destination: **`%cs:function`**) for privilege switching.

A 16bit protected mode setup uses hundreds of segments – for the MMU functionality.

4.1.2.32bit Protected Mode

In 32bit mode, the memory management facilities of the protected mode turned inapplicable for two reasons:

- Registers, addresses and address offsets were now 32bit wide, there was no point anymore to force programmers into using and managing segment registers for accessing more than 16bit/64kB of continuous memory.
- The single-indirection table method that seemed still applicable for translating between “virtual” 16bit far pointers to physical addresses via the descriptor tables of course breaks down if the system can support 4GB physical memory. The way the

segment selector is laid out (two bits for privilege, the table type bit, and 13 bit for the table index) does only allow 8192 descriptor table entries, but at 64kB per segment one needs 65535 (64k) to cover the entire 32bit physical address space.

32bit programs therefore were supplied with a wholly separate 32bit MMU which will be described in section 3.3. Segmentation is *no longer used for address space separation,* the new (32bit) MMU takes over this task.

Segment descriptors in 32bit mode are supposed to be set up to allow *flat addressing*, where the "near" 32bit part of the 48bit *logical address* (i.e. the far pointer including the implicitly used segment register) maps 1:1 to the 32bits of the *virtual address*. This trick is accomplished by setting the base address to NULL and the segment size to 4GB in those segment descriptors.

The effect is to *bypass segmentation*:

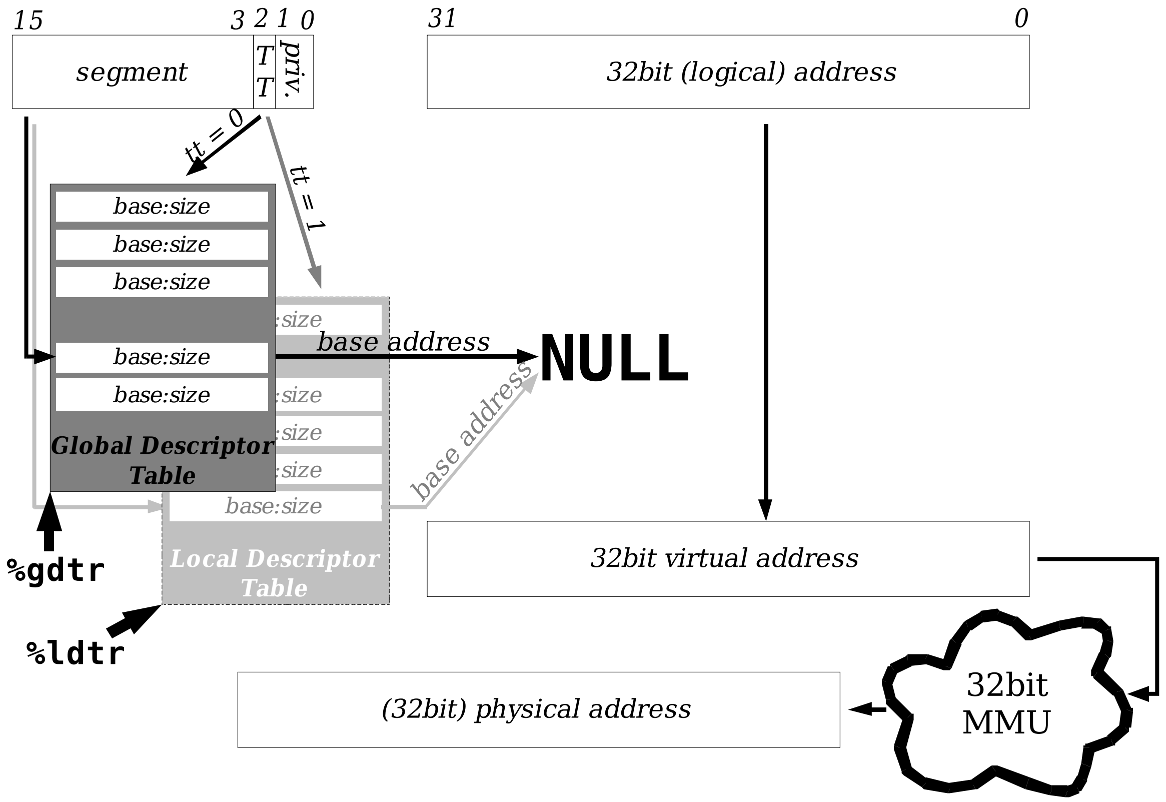


Illustration 4 - creating flat (unsegmented) 32bit address spaces on the 80386

The protected mode in 32bit is therefore reduced to:

1. Declare the privilege levels to use. Each ring to be used needs two segment descriptors:
    - one flat code segment descriptor, and
    - one flat data segment descriptor.

   All applications would share the same set of code/data segment descriptors. Since all these segments were flat (and therefore overlap), the segment registers contain, at all times, only one out of two configurations:

Segment Register	***value in kernel mode***	***value in user mode***
`%cs`	ring 0 code segment	ring 3 code segment
`%ds/%es/%ss`	ring 0 data segment	ring 3 data segment

2. Provide a privilege switching mechanism, i.e. at least one *system call gate*.

A typical 32bit protected mode setup therefore is very simple:

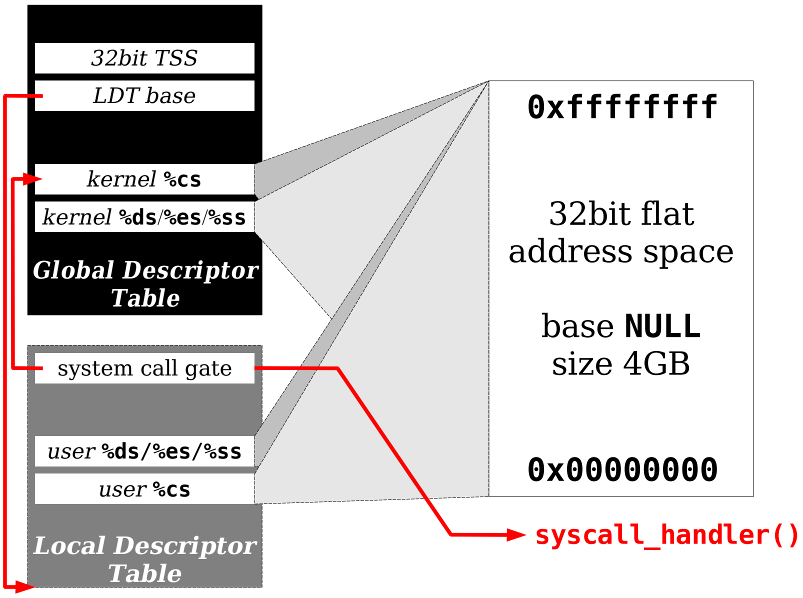


Illustration 5 - typical 32bit protected mode setup

Even using a Local Descriptor Table at all is optional – it's possible to put both user and kernel mode code/data segments and syscall gate into the Global Descriptor Table.

Note that on x86, *executability* is a property of the code segment. The classical 32bit x86 MMU does not have an attribute bit for “is this page executable ?” - only post-AMD64 MMUs know about a “NX” page attribute bit. A certain memory location is executable if there exists a code segment that covers it. Consequently, if we let code and data segments overlap (and there are very good reasons for this – do you want function addresses to be conceptually different from any other address ?), everything is executable.

On classic x86 platforms using the 32bit protected mode with flat segments, every address is executable !

4.1.3.System segments

The Protected mode uses two special segment descriptor types (system descriptors) for control purposes:

1. The *Logical Descriptor Table* in fact is a (non-flat) segment. **`%ldtr`** therefore doesn't contain a memory location (like **`%gdtr`** does), but a segment selector – the local descriptor table register is a segment register, with a special rule that it may only contain selectors whose **`TT`** bit is clear (i.e. which index the GDT).
   The original idea was to use the GDT for kernel segments and the LDT for those of applications, and operating systems keeping track of these user contexts by creating one LDT per application.
2. The *Task State Struct* (TSS) is another type of system descriptor.

TSS embodies the idea of a “context” in hardware.

In the limited way how most operating systems set up the protected mode, the role of the TSS is to supply the CPU with stackpointer locations for the various rings of privilege.

- The 32bit mode TSS is more complicated because it contains a backing store space for the general-purpose and segment register set as well as the set of stackpointers for the various privilege levels:

```
struct tss {
        uint16_t        tss_link;       /* 16-bit prior TSS selector */
        uint16_t        tss_rsvd0;      /* reserved, ignored */
        uint32_t        tss_esp0;
        uint16_t        tss_ss0;
        uint16_t        tss_rsvd1;      /* reserved, ignored */
        uint32_t        tss_esp1;
        uint16_t        tss_ss1;
        uint16_t        tss_rsvd2;      /* reserved, ignored */
        uint32_t        tss_esp2;
        uint16_t        tss_ss2;
        uint16_t        tss_rsvd3;      /* reserved, ignored */
        uint32_t        tss_cr3;
        uint32_t        tss_eip;
        uint32_t        tss_eflags;
        uint32_t        tss_eax;
        uint32_t        tss_ecx;
        uint32_t        tss_edx;
        uint32_t        tss_ebx;
        uint32_t        tss_esp;
        uint32_t        tss_ebp;
        uint32_t        tss_esi;
        uint32_t        tss_edi;
        uint16_t        tss_es;
        uint16_t        tss_rsvd4;      /* reserved, ignored */
        uint16_t        tss_cs;
        uint16_t        tss_rsvd5;      /* reserved, ignored */
        uint16_t        tss_ss;
        uint16_t        tss_rsvd6;      /* reserved, ignored */
        uint16_t        tss_ds;
        uint16_t        tss_rsvd7;      /* reserved, ignored */
        uint16_t        tss_fs;
        uint16_t        tss_rsvd8;      /* reserved, ignored */
        uint16_t        tss_gs;
        uint16_t        tss_rsvd9;      /* reserved, ignored */
        uint16_t        tss_ldt;
        uint16_t        tss_rsvd10;     /* reserved, ignored */
        uint16_t        tss_rsvd11;     /* reserved, ignored */
        uint16_t        tss_bitmapbase; /* io permission bitmap base
address */
};
```

%esp/%ss for all privileged rings

- The 64bit TSS is “reduced” to the simple role of providing stackpointers – one for each higher-privileged ring of execution, and a selectable table of *seven interrupt stackpointers*, the **IST[]**:

```
#pragma pack(4)
struct tss {
        uint32_t        tss_rsvd0;      /* reserved, ignored */
        uint64_t        tss_rsp0;       /* stack pointer CPL = 0 */
        uint64_t        tss_rsp1;       /* stack pointer CPL = 1 */
        uint64_t        tss_rsp2;       /* stack pointer CPL = 2 */
        uint64_t        tss_rsvd1;      /* reserved, ignored */
        uint64_t        tss_ist1;       /* Interrupt stack table 1 */
        uint64_t        tss_ist2;       /* Interrupt stack table 2 */
        uint64_t        tss_ist3;       /* Interrupt stack table 3 */
        uint64_t        tss_ist4;       /* Interrupt stack table 4 */
        uint64_t        tss_ist5;       /* Interrupt stack table 5 */
        uint64_t        tss_ist6;       /* Interrupt stack table 6 */
        uint64_t        tss_ist7;       /* Interrupt stack table 7 */
        uint64_t        tss_rsvd2;      /* reserved, ignored */
        uint16_t        tss_rsvd3;      /* reserved, ignored */
        uint16_t        tss_bitmapbase; /* io permission bitmap base
address */
};
#pragma pack()
```

Since in the 64bit mode, all implicitly-used segments (**%cs**/**%ds**/**%es**/**%ss**) are flat, it's unnecessary to provide values for **%ss** in any privilege level.

Intel originally introduced the TSS for *hardware task switching*. The current task is, like the **%ldtr**, a special segment register called *task register* (**%tr**). The selector in there indexes the GDT. An operating system could have multiple (one per process) TSS segments in the GDT, and "switch" between them by reloading **%tr**. Such a task switch would save the current state (registers) to the current TSS, and then reload that (i.e. all register/segment register contents) from the new TSS, making that current.

While a given task is running (i.e. a certain TSS being active), the TSS provides the CPU with the information *where to find kernel stackpointers* when doing a privilege switch.

Hardware task switching has proven troublesome over time:

- the TSS provides no means for saving/restoring floating point registers or other register extensions that were introduced in the x86 family post-80386.
- the TSS provides no means for an operating system to attach "OS state" to a task.
- hardware task switching is CISC at its worst – it's a single CPU instruction but executing this is horribly slow. It's in fact much slower than saving/restoring the register set manually using simple sequences of instructions.
- hardware task switching doesn't scale to large numbers of processes.
  Descriptor Tables have size limitations – the index part of a segment selector is only 13bit and the GDT therefore cannot be larger than 8192 entries. But TSS segments must be in the GDT, and a limit of ~few thousands of threads is below what x86 CPUs have been able to handle for some generations by now, even in 32bit.

This is why even Intel's manuals today discourage the use of hardware task switching. AMD in devising the 64bit extension therefore decided to limit the use of the TSS when running in 64bit mode to its remaining core purposes:

1. Provide stackpointers for the various privilege levels.
2. Provide a mechanism for interrupts to run on separate stacks.

Hardware task switching is no longer possible in 64bit mode – there's always one task only, and the operating system will need to change kernel stackpointers within that single TSS if it wishes to use e.g. per-thread kernel stacks.

4.1.4.Privilege switching

The x86 protected mode, very much unlike other CPU architectures, does not know any implicit privilege switching. There is no instruction at all, and no interrupt, trap or other event which will end up in privileged mode – unless the CPU was programmed for the specific event to redirect execution to a handler function running with higher privileges.

All privilege switching on x86 platforms happens through *gates*. Which means the "rings" model probably should better be drawn like this:

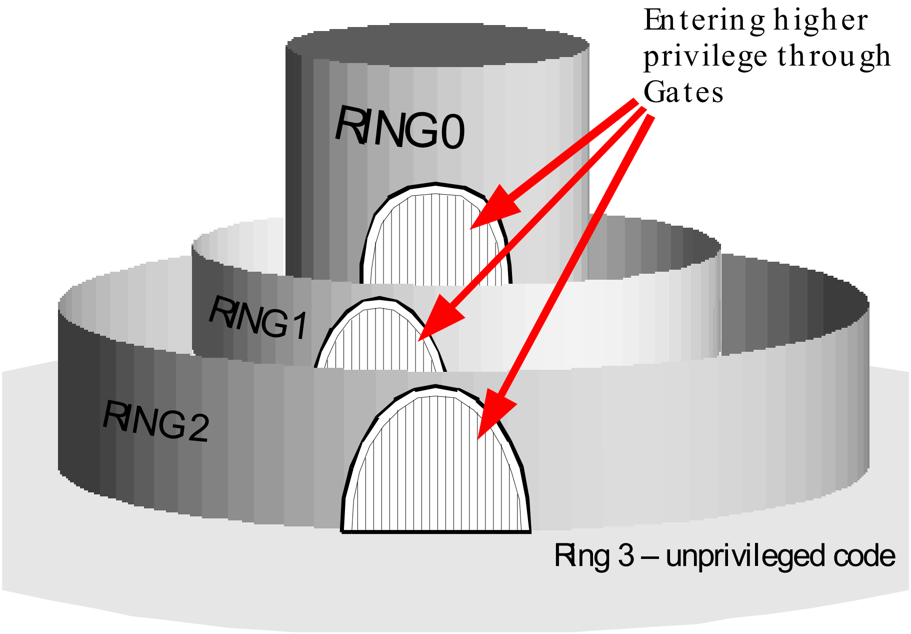

Entering higher privilege through Gates
RING0
RING1
RING2
Ring 3 – unprivileged code

Illustration 6 - privilege switching and gates

A gate is a descriptor (i.e. an entry of a descriptor table) which, instead of base address and size specifies a *gate handler address* and a *target code segment selector*.

Gates therefore:

- redirect execution to a specific location (the *gate handler*) in the target **`%cs`**.
- switch privileges if the *target* **`%cs`** privilege level is not equal to the *current* **`%cs`** privilege level.

Privilege switching is done by *calling a gate*. Gate calls can be:

- *explicit*, by using the far call instruction, **`lcall`**, and specifying the segment selector that indexes the desired gate in the GDT or LDT.
- *implicit*, via the *Interrupt Descriptor Table* (IDT). The IDT is special in the sense that it only may contain gate descriptors, and must have exactly 255 entries (one for each x86 interrupt number). All hardware exceptions, faults, traps and interrupts on x86 are routed via the IDT.

More details on privilege switching will be given in section 3.2.2.

4.1.5. 64bit Protected Mode

The 64bit protected mode is highly simplified. In fact, the already-described common practice of setting up the (32bit) protected mode as follows:

- All implicitly used segments (**%cs** and **%ds/%es/%ss**) are flat
- one code and one data segment for ring 0 – kernel
- one code and one data segment for ring 3 – usermode
- Segment regs **%ds/%es/%ss** are equal at all times.
- Both **%cs** and **%ds/%es/%ss** can only have one of two set of values: kernel/user.

is made *mandatory* in 64bit mode.

The 64bit protected mode doesn't care about descriptor base/size values as far as the corresponding segment selectors are in **%cs/%ds/%es/%ss**. These segments are implicitly flat, and the CPU will only use/check the privilege level bits – and the type of the segment. A typical 64bit x86 protected mode setup uses:

- a 64bit code and a data segment (one each) for ring 0, the kernel
- one data segment for ring 3, applications (*shared* by 32/64bit applications)
- one 64bit code segment for ring 3, used by 64bit applications
- one 32bit code segment for ring 3, used by 32bit applications (*compatibility mode*)

All these segments are *implicitly flat* – the 64bit x86 CPU ignores base/size values in these descriptors.

System call and trap handling changes slightly – see section 3.2.

4.1.6. Segment and Gate Descriptor Formats

4.1.7. The role of segment registers %fs and %gs

The x86 architecture suffers from the lack of general-purpose registers – as shown, there are only eight of them in 32bit mode, and 16 in 64bit mode, and their contents are shared between all functions in a program, and between the different levels of privileges (a privilege switch doesn't change any of the general-purpose registers except for **%esp/%rsp**).

But there often is need to keep some fixed reference, like a pointer to *thread-specific data*, in a location that's quickly accessible.

CPU architectures with many registers at their disposal usually specify in the ABI that one register is supposed to be set aside for this use; SPARC, for example, gives **%g7** on every CPU to hold the address of the current thread.

Doing that on x86 is bad – it'd slow down code significantly. Consider e.g. the i386 UNIX ABI, which already specifies fixed roles for **%esp/%ebp** (the stack-/framepointer) and **%ebx** (for the location of the global offset table in position-independent code). Taking **%ebx** is bad enough and, as shown in chapter 2, slows down position-independent code significantly. But taking yet another of the general-purpose registers away for thread-specific data is a very bad idea, not only because it'd reduce the number of registers available to only four, but also because it'd prevent running code from being able to use the full x86 instruction set. All remaining five registers (**%eax/%ecx/%edx/%esi/%edi**) are used implicitly in some contexts (**%ecx** is the counter register for **loop**/**rep**, **%eax**/**%edx** are preferred operand registers for 32 64 multiplication/division, and **%esi**/**%edi** are operand registers for string instructions) are implicitly used somewhere.

This means another solution is required. Intel had seen the need for this, and in fact provides a way out of the problem by supplying two *segment registers* that are not implicitly used for anything - **%fs** and **%gs**.

There are two related concepts that can be implemented using global segments:

1. thread-specific data / thread-local storage (TSD/TLS). This means a per-thread key is used to locate a piece of data that's global to a given thread and accessible under the same key from all functions within this thread.
   Different threads use *different keys* to locate “their” data. All descriptor tables (on all CPUs) will contain a common set of segment descriptors (one per key) that locate the various data sets.
2. CPU-local data in multiprocessor systems.
   The *same key* is used by code indifferent of what CPU it is running on but depending on that a different set of data is provided, by putting different segment descriptors into each CPU's descriptor table at the index specified by the common selector value.

4.2.Traps, Interrupts, System Calls, Contexts

4.2.1.The Interrupt Descriptor Table

As mentioned before, x86 CPUs do not know any instruction nor any other event that implicitly would switch the CPU from nonprivileged into privileged execution. Instead, all events that on other CPUs commonly involve a switch into supervisor mode :

- hardware interrupts
- traps and machine exceptions
- code execution errors (arithmetic faults, undefined/illegal opcodes, breakpoints)
- privilege violation attempts (executing privileged instructions / accessing privileged memory from unprivileged code)

are programmable on x86 – they are routed through the *Interrupt Descriptor Table*.

The IDT is different from GDT/LDT in that *it can only contain gate descriptors*. In addition to that, it always contains 255 entries – one for each interrupt vector known to the x86 CPU.

4.2.2.Privilege switches and stacks

4.2.3.Fast system call interfaces

“Classical” x86 system calls using a *call gate* in the LDT and the **`lcall`** instruction have a long latency due to the various descriptor table lookups that are needed:

- A segment lookup is performed to extract the LDT base address from the LDT segment descriptor in the GDT.
- A segment lookup is performed to extract the gate descriptor from the LDT
- A segment lookup is performed to extract the kernel code segment base address from the GDT.

Only then can execution be transferred into the kernel, and the handler be dispatched.

A faster method to perform a system call is using an *interrupt gate* in the IDT and an **`int`** instruction to issue the system call. Since the IDT is no segment, but located directly in memory via its base address in **`%idtr`**, this involves one less descriptor table lookup. Using **`int`** instead of **`lcall`** is therefore a preferable way how to perform system calls on x86 machines and yields lower latency syscalls.

But even that is still burdened with the overhead of segmentation and descriptor table lookup.

In a flat memory model as it is used in 32bit and 64bit protected mode, a *far pointer* **`kernel_code_segment:syscall_handler`** contains all of the information required to perform the privilege switch and call the kernel entry point. The target (kernel) code segment's privilege bits determine that a privilege switch is requested, and since the kernel code segment is flat no base address needs to be added to the handler, segmentation memory translation is a no-op. No descriptor table lookups at all are necessary to derive this.

What's needed therefore is a way to tell the CPU: For performing a syscall, call a specific predefined far pointer, i.e.:

- switch to the kernel code segment (and raise privileges as requested)
- run the kernel's system call handler given its address.

Both Intel and AMD independently introduced fast system call mechanisms in their x86 CPUs that allow this simple “switch to privileged mode and call that handler” approach – **`sysenter`** from Intel, and **`syscall`** by AMD.

The **`syscall`** instruction, if available (Intel CPUs only know it if they have the AMD64-compatible EM64T extension), is the preferrable solution because it automatically saves usermode return addresses and stackpointers on entry, and the corresponding **`sysret`** instruction can resume execution in userland after the system call from there directly. Intel's **`sysenter/sysexit`** instructions require the caller to pass return addresses and usermode stackpointers in registers, and the kernel must manually restore them before being able to issue **`sysexit`** to return from the system call.

Apart from these implementation differences, the actual mechanism to control fast system calls are similar between the two. As an example, the **`syscall/sysret`** method will be described here.

4.3.Virtual Memory Management on x86

The 8086 16bit CPU did not support any form of memory management – the mapping between the upper 16bits of a “logical” (`far`) address, i.e. the segment ID, and the upper 16bits of the 32bit (well – 20bit) physical address was static, 1:1. In addition, as mentioned before, the 8086 had no notion of privilege and operating systems could neither establish separate address spaces between nonprivileged user applications and privileged kernel code, nor prevent application code from executing instructions that would modify “critical” state.

With the 80286, Intel both introduced a mechanism for privilege management and made the mapping between segment IDs (the `far` part of an address, i.e. the upper 16bit) and physical address programmable. In the 80286 and following CPUs operating in 16bit mode, the *protected mode* allowed for privilege and address space separation by declaring (non-overlapping) user/kernel code and data segments. Since the association between segment IDs and their (physical) location in memory is fully programmable, the 16bit protected mode implemented a simple one-level MMU, with the GDT/LDT functioning as *translation table* for virtual/physical memory access. In other words: In 16bit mode, segmentation actually performs the role of the MMU, and logical addresses (`far` pointers) are virtual addresses.

4.3.1.The classical 32bit x86 MMU

The use of segmentation for virtual/physical translation may have been appropriate when x86 CPUs were 16bit only. But for 32bit mode, the reliance on `far` pointers, or explicitly segmented memory access, causes severe problems. Why should anybody want to fiddle with multiple segments in applications/operating systems if a single 32bit pointer can locate every byte of physical memory in a machine ?

In other words: If the *segment offset* alone (lower parts of a logical address, now a 32bit value) can address every piece of physical memory in a machine, why bother with multiple segments (and 48bit `far` pointers) at all ? What's needed for 32bit operation is a *flat address space* – unsegmented, with addresses starting at zero and ending at 4GB.

The 32bit Protected Mode allows to create such flat segments, which start at zero and cover all of the 32bit address space. But doing that reduces the 32bit protected mode to a vehicle for *supplying privileges only*. By using a pair of flat segments for application code and data (running in ring 3), and another such flat pair for kernel code and data (running in ring 0), both user and kernel code can run in a flat address space.

But clearly doing so removes memory management from the *Segmentation MMU*. The descriptor tables are now programmed in such a way that any memory translation capabilities via descriptor tables are *bypassed*. There's again a 1:1 mapping between *logical address* (i.e. `far` pointer) and the result of the segmentation translation step, and the unit-of-memory, the segment size, is 4GB.

In a flat 32bit mode, a logical address *cannot* be translated to a physical address directly. Logical and virtual addresses are *no longer equal*, and a memory granularity of 4GB is inappropriate. Result of the segmentation translation will be a virtual address now, and a new mechanism to translate this to the actual physical address is required.

This means for 32bit mode, the 80386 had to supply a new memory management unit as a 2nd stage of address translation – to convert from a 32bit virtual address to a 32bit physical address.

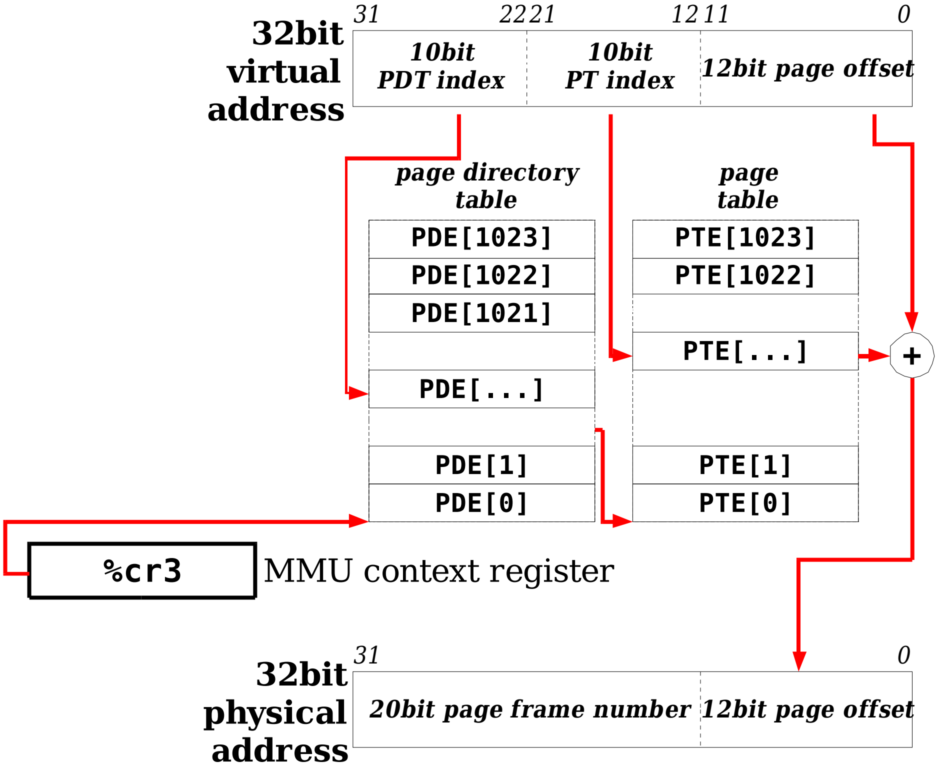


Illustration 7 - classical 32bit x86 MMU

The 32bit MMU performs address lookups using simple hashing. In C pseudo code, its operation can be expressed as:

```
register paddr_t  ***cr3;          /* pagedir base address in %cr3 */
paddr_t     tables[][];
#define     TABLESZ     (1 << 10)   /* 1024 */
#define     PAGESZ      (1 << 12)   /* 4096 */
#define     PAGEDIR_IDX(vaddr)      ((vaddr >> 22) & (TABLESZ – 1))
#define     PAGETBL_IDX(vaddr)      ((vaddr >> 12) & (TABLESZ – 1))
#define     PAGE_OFFSET(vaddr)      (vaddr & (PAGESZ – 1))
#define     VA_TO_PA(va)            tables[PAGEDIR_IDX(va)][PAGETBL_IDX(va)] \
                                          + PAGE_OFFSET(va)
```

In other words, the virtual address is split (hashed) into three parts:

- bits [31..22] supply the index into the level 1 table: The *page directory* (table). Entries in the page directory locate page tables in physical memory.
- bits [21..12] supply the index into the level 0 table: The *page table*. Entries in the page table locate the actual physical pages.
- bits [11..0] for the offset within the physical page.

Both page directory and page tables are sparse arrays of (no more than) 1024 32bit values. A specific index can therefore locate a physical page if the table contains a non-`NULL` *pagetable entry* that supplies sufficient permissions for the running code the access this page.

Otherwise, the MMU will cause a *pagefault* (`#PF` trap). This happens if:

- the pagetable entry is `NULL`, as indicator of unmapped memory
- the pagetable entry *present bit* has been cleared by the operating system to indicate e.g. a swapped-out page
- the running code executes at ring 3/0 but the *user/supervisor bit* in the pagetable

entry indicates that this page is supposed to be accessible from the “other” mode only.

The **#PF** trap has its own *pagefault address register* - **%cr2**. Virtual memory (i.e. *paging*) can easily be implemented via this mechanism. Whenever a pagefault occurs, the handler will find the virtual address that caused the fault in **%cr2**. It will explicitly perform the table lookup and inspect the pagetable entry at this position. If the entry doesn't exist, it can e.g. choose to create it (giving semantics of **MAP_ANON mmap**). If it exists but the page present bit is clear, it may decide to e.g. load the page from swap.

The illustration and the examples given so far already indicate that page directory and page table entries are not just 32bit physical addresses. And they need not be – they locate pages, and because the page size is a fixed 4kB a 20bit number, the *page frame number*, is sufficient to enumerate all 4kB pages on a machine that allows for 4GB of physical memory. Pagetable entries therefore contain the PFN – and attributes for that page. Some of them were already mentioned, the *page present* and the *user*/*supervisor* attributes. But there are more. **<vm/hat_pte.h>** on Solaris 10/x86 names them:

```
#define PT_VALID        (0x001) /* a valid translation is present */
#define PT_WRITABLE     (0x002) /* the page is writable */
#define PT_USER         (0x004) /* the page is accessible by user mode */
#define PT_WRITETHRU    (0x008) /* write back caching is disabled (non-PAT) */
#define PT_NOCACHE      (0x010) /* page is not cacheable (non-PAT) */
#define PT_REF          (0x020) /* page was referenced */
#define PT_MOD          (0x040) /* page was modified */
#define PT_PAGESIZE     (0x080) /* above level 0, indicates a large page */
#define PT_PAT_4K       (0x080) /* at level 0, used for write combining */
#define PT_GLOBAL       (0x100) /* the mapping is global */
#define PT_SOFTWARE     (0xe00) /* available for software */
```

The upper 20bits of a pagetable entry will of course contain the page frame number. PTEs therefore have the following format:

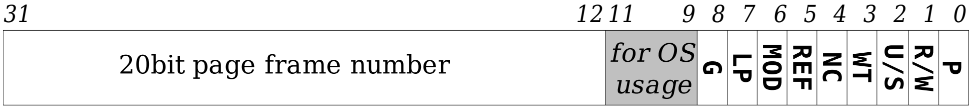


Illustration 8 - 32bit pagetable entry

As shown, there are no spare/reserved attribute bits left – all are used. This wasn't so in the 80386, which did not have global or large pages nor caching attributes. These MMU features, as usual with x86, have to be detected and activated before use, and **CPUID** is required to find out whether a given x86-compatible CPU has these features.

What's noticeably missing from the 32bit MMU is an attribute for page executability. “Classical” x86 has this in the segmentation MMU only, in form of code segments. This means that in flat address spaces, a page is executable if it is readable !

4.3.2.Physical Addressing Extension (PAE)

Ten years after the introduction of the 80386, x86-based systems had evolved far enough (and far beyond what Intel had anticipated) that the need for big server systems capable of accessing more than 4GB of physical memory became evident. Competing 32bit architectures of that time, like sun4d/32bit sun4u, PA-RISC 7xxx or MIPS 3xxx which were all used in server systems by various vendors, have had MMUs that allowed the operating system to hold several 32bit programs including all of their 4GB address space in system memory concurrently. This requires MMU translation modes that convert 32bit virtual addresses into more-than-32bit physical addresses.

The 32bit x86 MMU had no provisions for that. There are no spare/reserved bits in 32bit pagetable entries, so the page frame number couldn't simply be extended. In addition to that, the size of a pagetable had to be one physical page (4kB), so Intel neither could just double the size of pagetable entries.

Intel therefore had to:

- increase the size of a pagetable entry from 32bit to 64bit, using (some of) the new spare bits for a larger page frame number
- half the size of page table/page directory table from 1024 entries to 512 entries so that the total size of the table would stay 4kB.

But halving the table size to 512 of course means that the virtual address can no longer be split 10:10:12 – but 2:9:9:12, necessitating:

- the introduction of a *third translation table*.

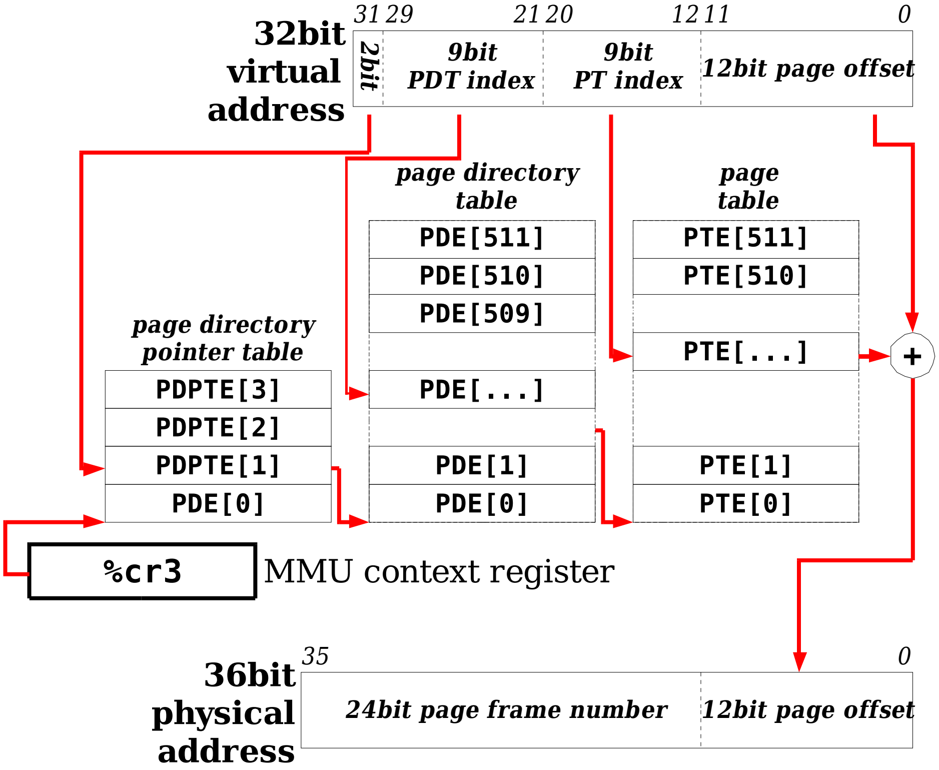


Illustration 9 - Physical Address Extension: allow 64GB memory in 32bit mode

This so-called *Page Directory Pointer Table* can, given that there are only two bits for its index, only contain four entries of course.

This MMU mode, called *Physical Addressing Extension* (PAE), was introduced with the PentiumPro/II CPUs and uses three levels of translation tables:

- the *page directory pointer table* (four entries only)
- the *page directory table* (which now has 512 entries)
- the *page table*.

The format of pagetable entries is twice the size as before, but uses the same format including all of the attribute bits with two exceptions:

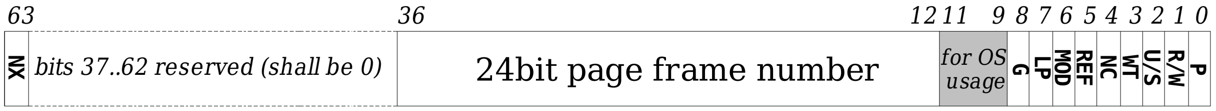


Illustration 10 - 64bit pagetable entry, 36bit PAE mode

1. The page frame number is now (at least) 24bits.
2. If the CPU supports it, bit 63 is a new page attribute “NX” - *not executable*.

The NX bit got introduced by AMD in the Opteron processors, but it's not 64bit specific. As usual with x86 extensions, the presence of the NX bit can be queried by the operating system using the **CPUID** instruction.

CPUs that have the NX bit the long-missing capability in (32bit) x86 to protect areas of memory from being executed. The feature is also called *Data Execution Protection* (DEP) by Microsoft, and *Enhanced Virus Protection* (EVP) by AMD Marketing – though the latter is a technically a misleading term, since the “only” thing NX protects against are simple-written stack overflow exploits.

4.3.3.The AMD64 64bit MMU

The major reason for going to 64bit, whatever CPU vendor, has always been to allow concurrent access to large virtual address spaces. This of course mandated a new MMU mode – both the classical 32bit x86 MMU and the PAE mode are only supporting 32bit virtual addresses.

AMD in designing the 64bit mode retained all existing x86 characteristics. It's therefore not surprising that the 64bit MMU mode is simply an extension of PAE mode to 64bit virtual addresses. PAE mode properties also found in the 64bit MMU are:

- pages are 2^{12} bytes, 4kB.
- Page directory entries can point to large pages of 2^{21} bytes, 2MB.
- pagetable entries use the PAE format (64bit wide, attribute bits identical to PAE)
- translation tables contain 2^9 (512) entries.

The big deficit of PAE mode, unbalanced translation tables because of the use of only 2 bits for the page directory pointer table index, of course is solved because 64bit virtual addresses supply enough bits to make that table, like all others, 512 entries. AMD additionally added a fourth table, simply called “*Page Map Level 4*”. The 64bit MMU therefore looks like this:

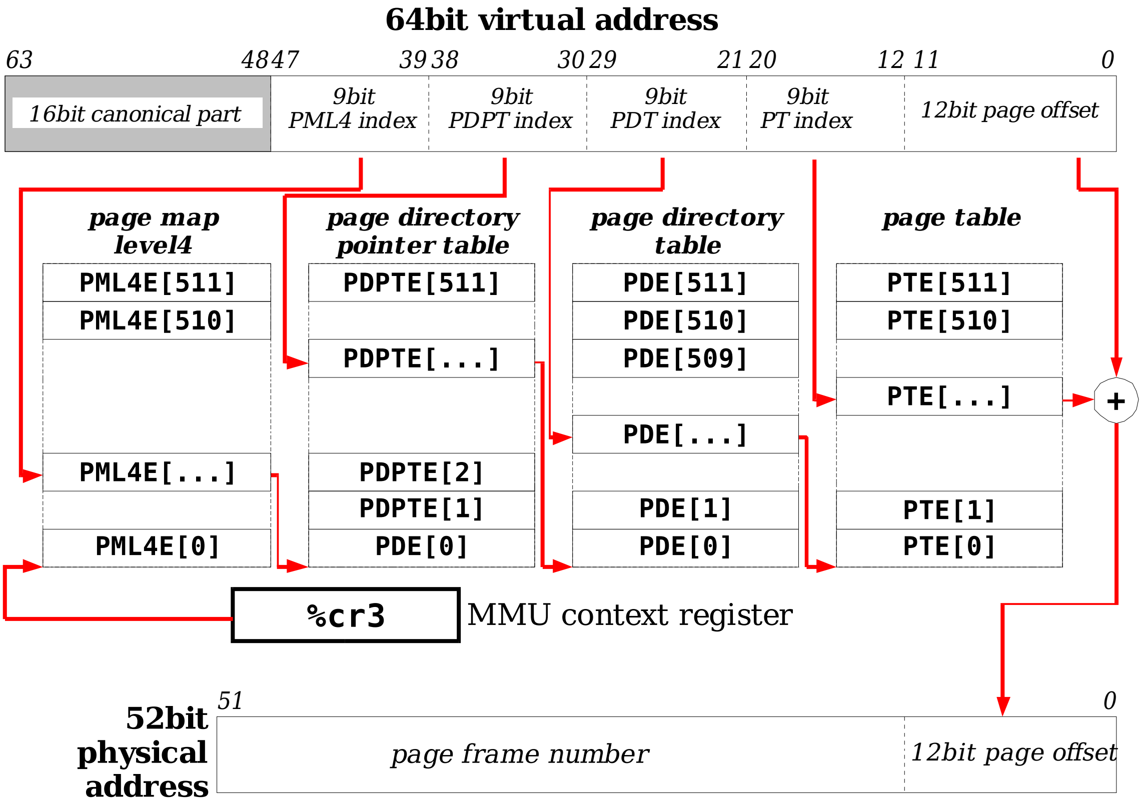


Illustration 11 - 64bit x86 MMU

Pagetable entries, as mentioned, use the PAE format – except, of course, the PFN which is no longer 24bit as in 32bit PAE mode, but 40bit now.

This allows the 64bit MMU to access 2^{52} bytes of memory, 4PB in total.

The 64bit MMU of course supplies the NX bit that AMD introduced with the Opteron and Athlon64 CPU series. 64bit pagetable entries use the following format:

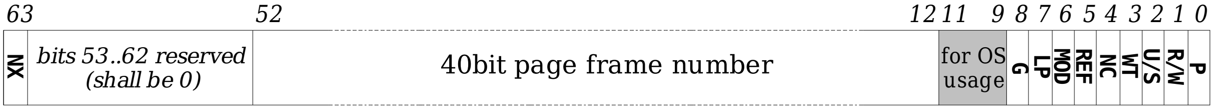


Illustration 12 - 64bit pagetable entry, 64bit mode

What needs explanation of course is the question what the virtual address bits that are not used for page offset and table indices are supposed to be. Simple arithmetics tells us that the MMU uses 12+9+9+9+9 = 48 bit for address translation only. What's the state of the upper 16 bits of a virtual address ?

The answer to that is that the MMU will only perform address translation if the upper 16bits are either *all zero* or *all one*, depending on the *state of bit 47*.

If we consider the 64bit virtual address space to be *unsigned* and to extend from 0 to 2^{64}-1, this splits the addressable virtual memory into *two ranges* of 2^{47} bytes (128TB) each, one at the *bottom* and one at the *top* of the virtual address space:

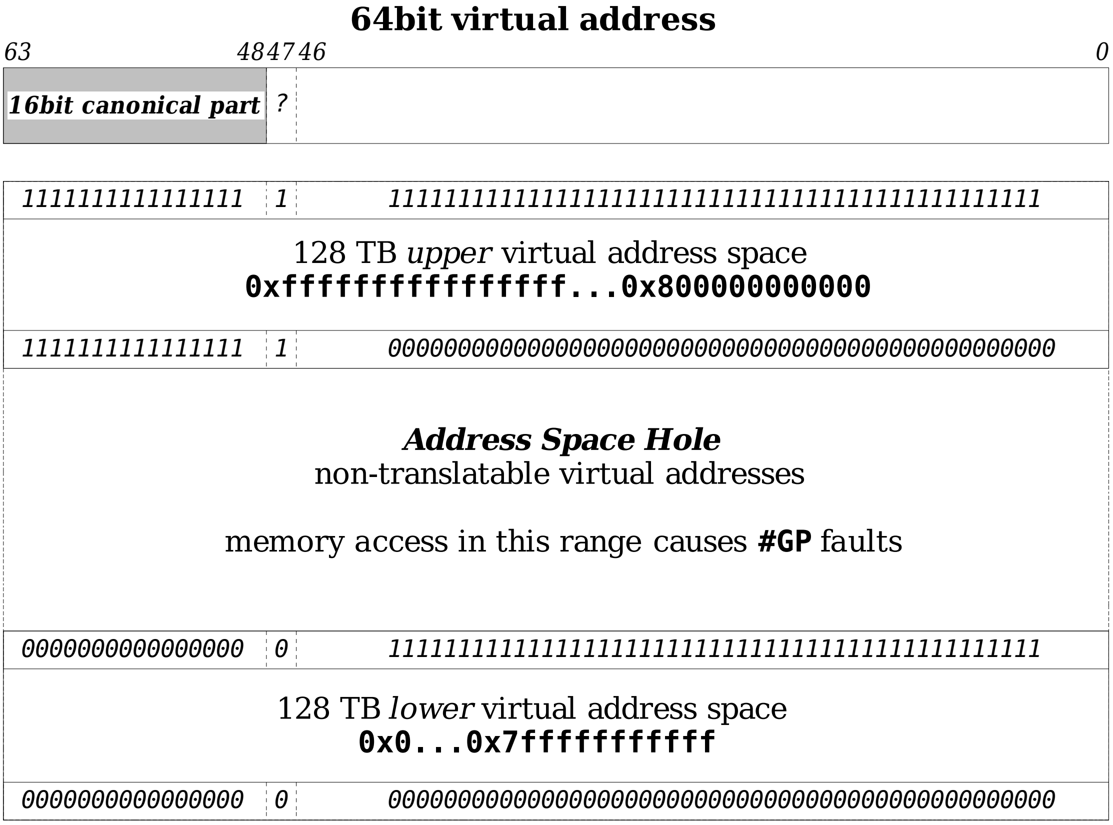


Illustration 13 - Address space hole in AMD64

The upper and lower address spaces are separated by an *address space hole*.

Translateable virtual addresses that match the condition of bits 63..47 inclusively are either all zero or all one, i.e. that the address is either within the low 128TB or the high 128TB of the virtual address space, are called *canonical addresses*. Virtual addresses in the hole are noncanonical and their use causes **#GP** faults.

Virtual address spaces with holes are not new in 64bit environments. For example, UltraSPARC-I and II also had an address space hole (but they were even more limited than AMD64 – with only 2x1TB of virtual address space).

There is a different way of understanding 64bit MMUs which use a canonical mode. Consider the virtual address to be a *signed 48bit value*. I.e. virtual addresses range from $-2^{47}...2^{47}-1$ bytes (i.e. ±128TB). The upper bits of the 64bit address are therefore derived by *sign extension*.

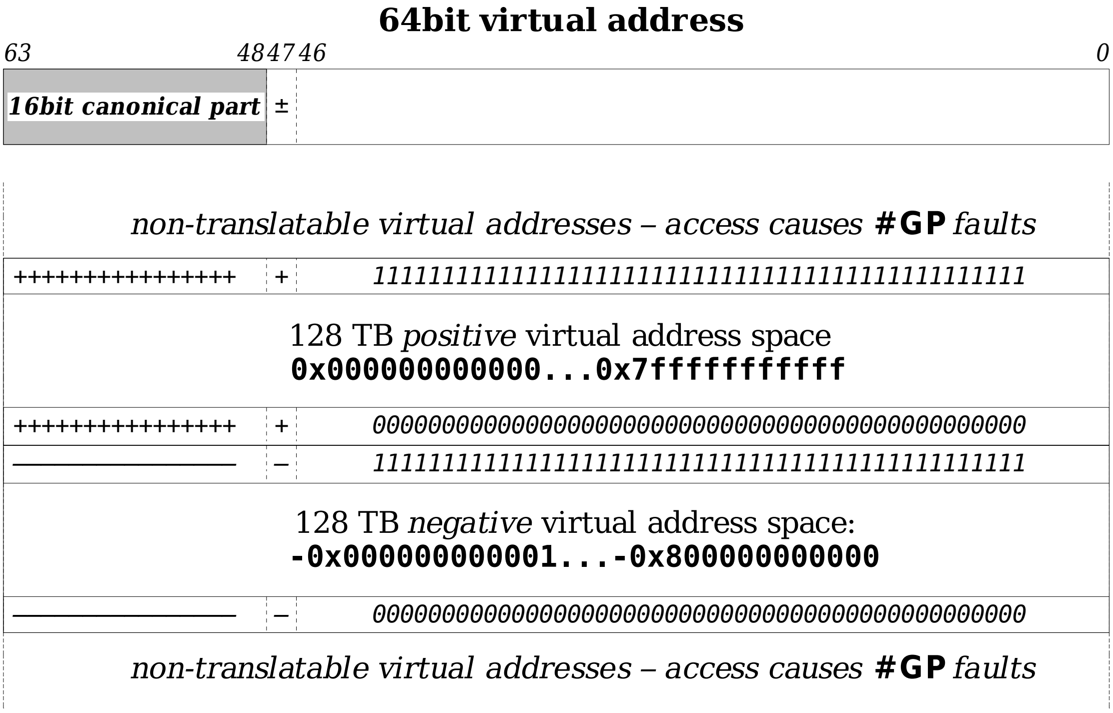


Illustration 14 - signed virtual addresses

In this signed representation, there is no address space hole. Allowed virtual addresses are continuous over a 256TB range, while virtual addresses outside of that range are undefined and cause **#GP** faults when used.

The terms *negative address range* and *upper address range* describe the same thing, and are often used interchangeably.

4.3.4. Large Pages

For accessing very large amounts of memory, 4kB pages are inconvenient and slow for obvious reasons:

- Having the operating system create e.g. 1 million pagetable entries to allocate a 4GB chunk of memory takes a while.
- Cached translations (TLB/*translation lookaside buffer* entries) will show heavy contention when so many translations need to be done all the time.

In order to speed up the use of large amounts of memory, *large pages* are being used. The common mechanism for creating a large page is to use a higher-level translation table entry as "*large PTE*" and make the size of large pages the sum of the sizes of all pages in the next lower-level table. In x86 terms, a page directory entry that has the *largepage attribute bit* set will not point to a pagetable, but directly to the physical location of the large page. Since pagetables contain 512 entries, the size of a large page will be 512x4kB, 2MB (if using the classical 32bit MMU mode: 1024x4kB, 4MB).

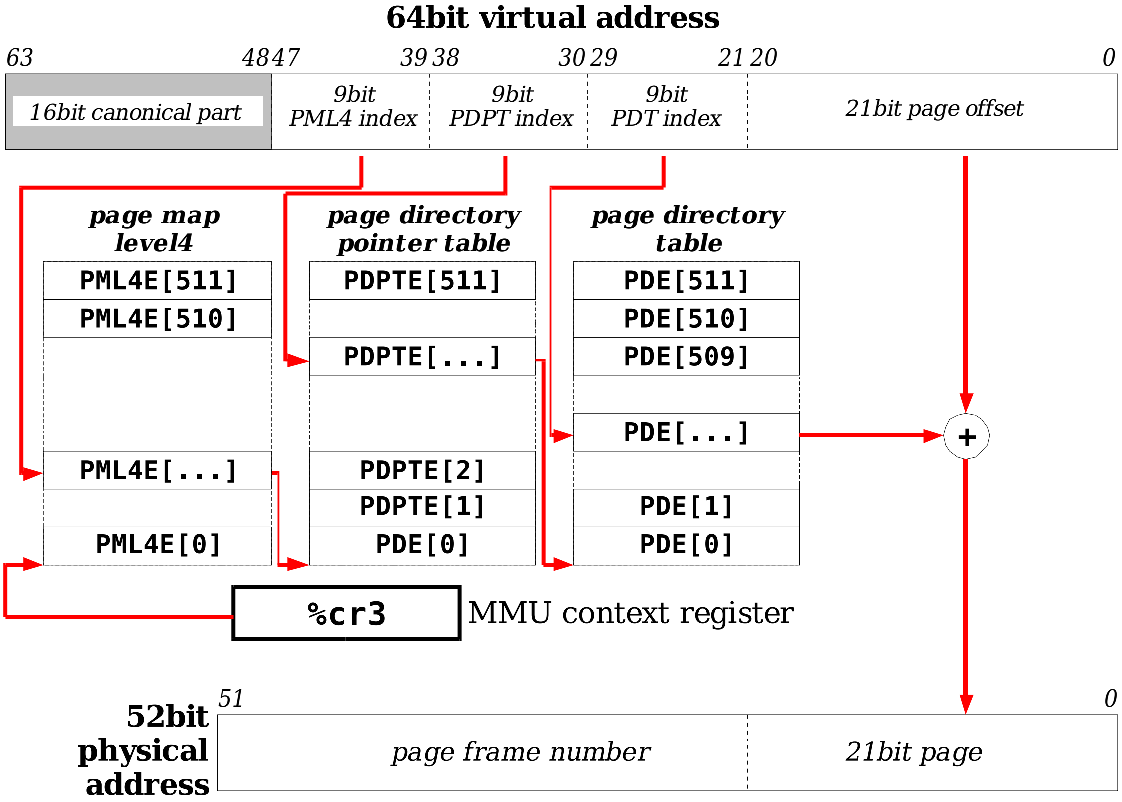


Illustration 15 - using large pages

The capability to support large pages, as all of the post-80386 features, again must be detected using a CPUID instruction, and selectively enabled. All 64bit-capable x86 CPUs support large pages, but not necessarily all 32bit "x86 compatibles".

At this time, there is no support for >2MB large pages (the next logical size would be 512x2MB, 1GB) in the x86 MMU. It's likely, though, that if this is ever introduced that it'll be done the same way – by making a page directory pointer table entry refer to a "*huge page*" of 1GB then, or even a "*giant page*" of 512GB, if ever the page map level 4 table may point directly to a page...

In any case, turning around the concept of large pages:
A **NULL** as table entry means:

- an unmapped area of 512GB if in the PML4 table
- an unmapped area of 1GB if in a page directory pointer table
- an unmapped area of 2MB if in a page directory table
- an unmapped page of 4kB if in a page table.

4.4.Advanced System Programming Techniques on x86

4.4.1.DMA using virtual addresses – IOMMU

4.4.2.Using the Protected Mode for Hardware Virtualization

Modern system architectures often allow to run multiple operating system instances on the same physical machine. Domaining / hardware partitioning / virtualization are the terms used to describe this capability. Supporting this requires

5.Interrupt handling, Device Autoconfiguration

5.1.Interrupt Handling and Interrupt Priority Management

x86 processors alone know only interrupt vectors (indices into the IDT) and a bit in the processor status register **`EFLAGS`**/**`RFLAGS`** that says “interrupts enabled”. The CPU knows no concept of interrupt priorities. To a x86 CPU, all 255 (resp. all user-available 223) interrupt vectors are equal. The processor has pins that connected peripheral devices can use to “cause interrupts”, but apart from “ignore all” (IE bit clear – after a **`cli`** instruction) and “accept all” (**`IE`** bit set, after a **`sti`** instruction) it hasn't the ability to selectively block interrupts, e.g. from a low-prio disk device if a handler for a high-prio network interrupt is just running. All of the following:

- Binding hardware interrupt sources to CPU interrupt vectors,
- classifying interrupts on peripheral devices into different priority classes, and
- selectively blocking out specific devices or specific interrupt priorities
- manage interrupt state (interrupt active/pending)

has, on x86 platforms, always been the task of a device called *Programmable Interrupt Controller* (PIC). The “classical” x86 PIC is the Intel i8259A.

5.2.APIC and IOAPIC features

5.2.1.Overview

In a multiprocessor environment, interrupt handling becomes significantly more complicated than before. The interrupt controller in SMP systems must provide all the capabilities of interrupt management and prioritization as the simple PIC, but in addition to that, a SMP-capable interrupt controller must be able to:

- *route interrupts*. It's highly undesirable on a multiprocessor system to bother all CPUs at once with handling a specific device interrupt.
- *support inter-processor interrupts*, i.e. a CPU itself being the interrupt source of another CPU.

With the introduction of the Pentium (P5) microprocessor architecture, Intel integrated SMP support on chip, with an interrupt arbitrator / coherency controller subsystem called the *Advanced Programmable Interrupt Controller*, APIC.

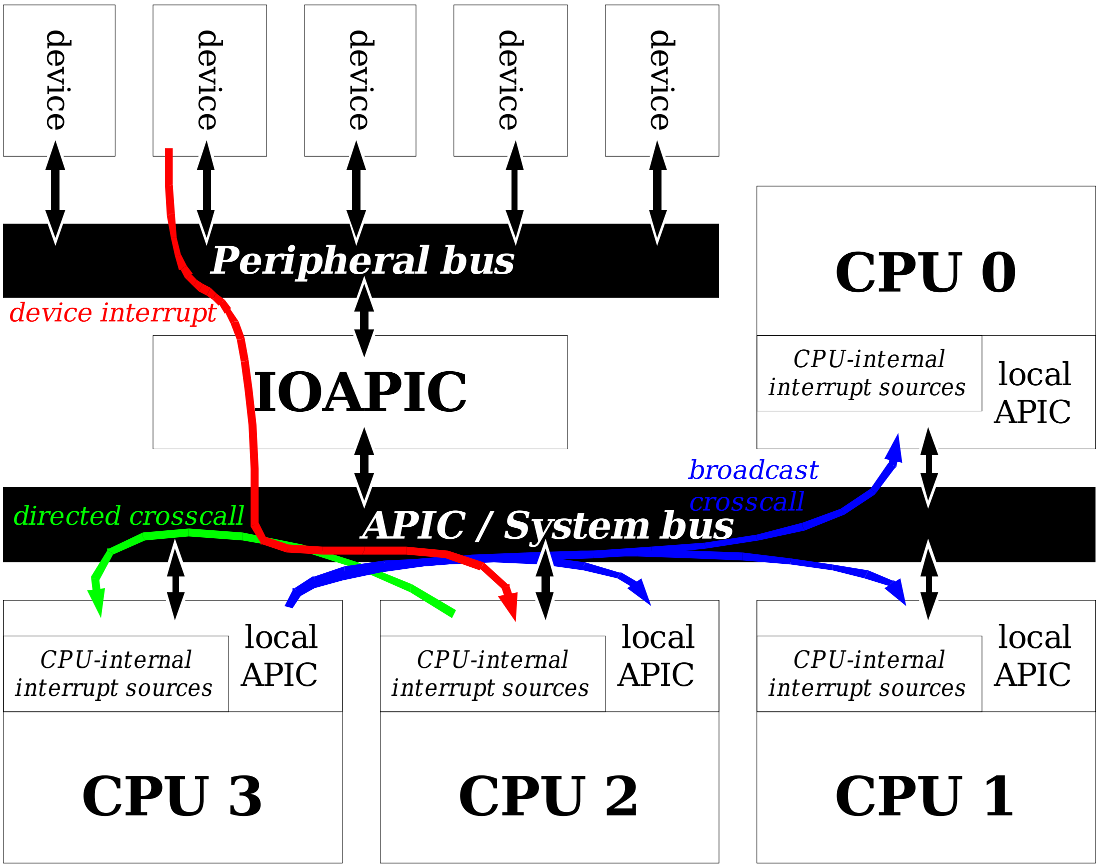


Illustration 1 - APIC functionality

Every modern x86 processor contains a *local APIC*, whose tasks are:

- Receive and dispatch local interrupts (from devices directly connected to CPU interrupt pins)
- Dispatch CPU-internal interrupts (APIC timer, temperature sensors, performance monitoring events)
- Receive and dispatch external interrupts (from the IOAPIC, a system/peripheral bus component that routes peripheral interrupts via the APIC protocols)
- Receive and dispatch inter-processor interrupts, IPIs (IA32 term for crosscall).

- Manage interrupt priorities.

The chipset will contain an external or I/O APIC, which is then used as a programmable dispatch facility for hardware interrupts to the various local APICs of the processors in the system.

The APIC routes external (IOAPIC) and local (see above) interrupt sources to IA32 interrupt numbers in a user-programmable way. APIC registers used for this purpose contain the IA32 int# to dispatch on event in their lowest 8 bits.

The APIC groups interrupt vector numbers (0..255) into 16 priority groups, with the priority of an interrupt given by int#/16. Priority 0 and 1 are highest (hardware exceptions), and can neither be created nor blocked by the APIC. Priorities 2..15 are available to APIC interrupts. The *Task Priority Register*, **TPR**, allows to block interrupts of low priority, while the read-only *Processor Priority Register*, **PPR**, reflects the current settings.

CPU-CPU communication is executed via *Inter-Processor Interrupt*, IPI, the x86 term for "crosscall". To generate IPIs, the Interrupt Control Register, ICR, is written. The ICR alone enables the programmer to:

- dispatch a programmable int# to a single other CPU (targeted IPI)
- broadcast a programmable int# to all CPUs, including/excluding the sending CPU

Additional APIC registers, the *Local Destination Register*, **LDR**, and the *Destination Format Register*, **DFR**, allow even for multicast IPIs (restricting broadcasts to a selected set of CPUs).

The APIC is programmed like a memory-mapped device; base address for the mapping is the machine-specific **IA32_APIC_BASE** register, any APIC registers are offset relatively to the base. To read or write an APIC register, simply access memory at the corresponding offset relative to the APIC base address.

The IOAPIC is a “doubly indirect” mapped device. It has two registers (**IOREGSEL** and **IOWIN**) that are accessed relative to the *Southbridge*'s **APIC_BASE**. (this is *not* the same as the CPUs **IA32_APIC_BASE** !) IOWIN maps to one of the actual IOAPIC registers depending on the value in **IOREGSEL**. I.e. to program an IOAPIC register, the register number is put into **IOREGSEL** first, and the actual IOAPIC register is then accessed via **IOWIN**.

5.2.2.APIC interrupt registers

APIC registers, whether IOAPIC (Intel 82093AA) or CPU-local APIC, use a common format for all registers associated with interrupt delivery:

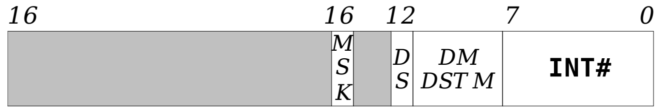


Illustration 2 - APIC Register format, for interrupt-dispatch related registers

The bits have the following meaning:

- 0..7: **INT#**
  the x86 interrupt vector to dispatch to the target CPU(s) on event
- 8..11, delivery mode / destination mode (only in registers for nonlocal delivery)
  decides which CPUs are targeted in broadcast/multicast routing modes
- 12: DS (delivery status)
  indicates whether an interrupt of this type is pending (queued) due to higher-

prioritized interrupt handlers running

- 16: MSK (mask)
  can be used to selectively block generation of interrupts from interrupt sources (devices) controlled by the given (IO)APIC register.
- 56..63: destination APIC ID (only in registers for non-local delivery, i.e. the IOAPIC redirection table and the local APIC's Interrupt Command Register) determines the interrupt target CPU set, together with the delivery / destination mode bits and two APIC control registers (logical destination register, destination format register).

The main APIC registers related to interrupt creation/dispatch are on Pentium-IV systems (others may not have all of the P-IV's **`LVT[]`** registers – again, ask via **`CPUID`**):

APIC base +...	***Register***	***Description***
`0x320`	**`LVT[0]`**	*timer register*. Program high-resolution timer interrupts here.

`0x380/90`
`0x3e0`
	`CCR`, **`ICR`**, **`DCR`**	*current count/initial count/divide configuration register*.
		Additional state for the APIC timer.

`0x330`	**`LVT[1]`**	*thermal monitor register.* Create interrupts on danger of overheating.
`0x340`	**`LVT[2]`**	*performance counter register*. Interrupts for profiling.

`0x350`
`0x360`
	`LVT[3]`
	`LVT[4]`
		local device 0 register
		local device 1 register

`0x370`	**`LVT[5]`**	*APIC error register*. Create interrupt if APIC encounters an error.
`0x280`	**`ESR`**	*Error status register.* Auxilliary (readonly) information on APIC errors.

`0x300`
`0x310`
	`ICR`	*Interrupt Command Register* (64bit)
		Create inter-processor interrupts.

`0xd0`
`0xe0`
	`LDR`, **`DFR`**	*Logical Destination / Destination Format Register*.
		Used to clarify destination CPUs for IPIs.

The IOAPIC uses a set of 24 registers called *I/O Redirection Table* that uses the mentioned format to dispatch events on the peripheral bus via the APIC mechanism. The **`IOREDIR[]`** registers use the described generic register format to select interrupt vector and target CPU.

5.2.3. Interrupt priorities, `%cr8`

The APIC (as well as the older i8257 PIC) uses a simple mapping between interrupt priorities and x86 interrupt vectors:

- Interrupt Priority : vector# / 16
- Priority subclass : vector# % 16

I.e. high-priority interrupts get high interrupt vector numbers.

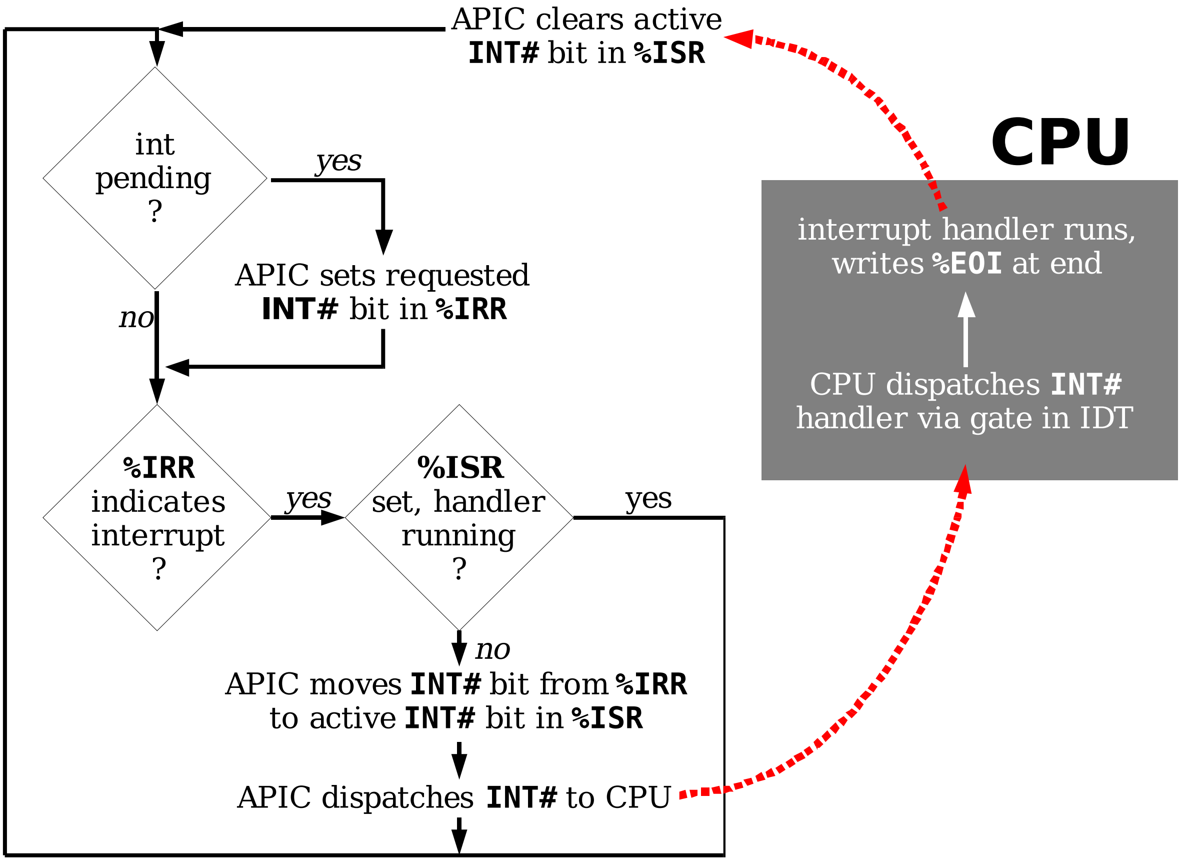
This mapping is not configurable; what the APIC allows to do is to block/queue interrupts not only based on the interrupt source & MSK bit, but also based on priority. For that purpose, the APIC provides a “priority register” which comes in two flavours:

Register Name	***Description***
Task Priority Register, **TPR**	Write-only register. Offset **0x80** Used to set the current IPL (Interrupt Privilege Level).
Processor Priority Register, **PPR**	Read-only register. Offset **0xa0** Used to query the current IPL.

Recent x86 CPUs map the APIC Interrupt Priority register directly to a new CPU control register. **%cr8**, if available, serves both purposes, to query and set the current IPL.

5.2.4. APIC interrupt processing flow

The APIC runs concurrently with and asynchronously to the CPU, in a state machine similar to the following:



APIC clears active **INT#** bit in **%ISR**
CPU
int pending ?
yes
no
APIC sets requested **INT#** bit in **%IRR**
interrupt handler runs, writes **%EOI** at end
CPU dispatches **INT#** handler via gate in IDT
%IRR indicates interrupt ?
yes
%ISR set, handler running ?
yes
no
APIC moves **INT#** bit from **%IRR** to active **INT#** bit in **%ISR**
APIC dispatches **INT#** to CPU

Illustration 3 - APIC interrupt processing workflow

Three additional APIC registers are involved with this flow:

- *Interrupt Request Register*, **%IRR**.
  The **%IRR** is a 256bit-wide bitmap – a pending but not-yet-dispatched interrupt will result in a bit being set in the **%IRR**.
  The APIC permanently monitors (active, i.e. not masked) interrupt sources and automatically manages the bitmap in **%IRR**.
  The register occupies 256bit of memory, offsets **0x200**..**0x270** to the base. Software can read this for debug purposes (but not write it).
- *In-Service Register*, **%ISR**.
  The APIC uses **%ISR** to indicate which interrupt is currently being serviced by the CPU. The **%ISR** contains the vector number. It's again a 256bit-wide register, but unlike the **%IRR** it can only contain exactly one bit set. The APIC automatically moves the highest-priority bit from **%IRR** to **%ISR** and dispatches the interrupt vector to the CPU if **%ISR** gets cleared.
  Again, 256bit of memory at offsets **0x100**..**0x170** map the **%ISR**. It's also readonly.
- *End-of-Interrupt Register*, **%EOI**.
  This register is writeonly and used as a trigger. The interrupt service routine running on the CPU writes to **%EOI** to indicate completion. Writing **%EOI** clears **%ISR** and causes the APIC to continue interrupt dispatch if there are interrupts queued via **%IRR**. The value written to **%EOI** is irrelevant – as said, it's a pure trigger.

6.Solaris/x86 architecture

From “The Tao of Programming”:

The warlord asked the programmer:
"Which is easier to design: an accounting package or an operating system?"

"An operating system," replied the programmer.

The warlord uttered an exclamation of disbelief.
"Surely an accounting package is trivial next to the complexity of an operating system," he said.

"Not so," said the programmer, "when designing an accounting package, the programmer operates as a mediator between people having different ideas:
how it must operate, how its reports must appear,
and how it must conform to the tax laws.
By contrast, an operating system is not limited by outside appearances.
When designing an operating system,
the programmer seeks the simplest harmony between machine and ideas.
This is why an operating system is easier to design."

The warlord of Wu nodded and smiled.
"That is all good and well, but which is easier to debug?"

The programmer made no reply.

To be covered here:

Solaris Internals

6.1.Kernel and user mode

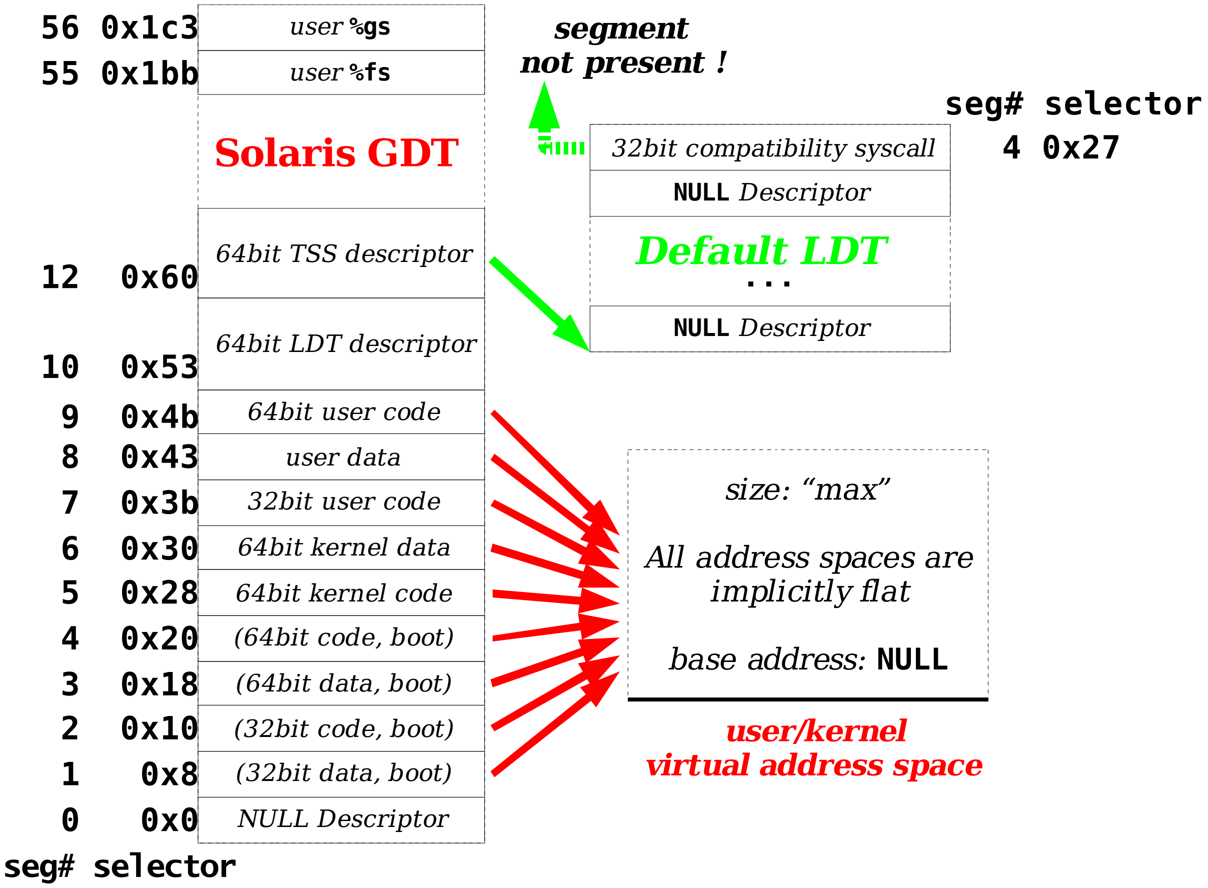


Illustration 1 - Solaris 10/x86 64bit protected mode

The way how Solaris/x86 sets up the 64bit protected mode is subject to the following constraints:

- The use of a bootloader and the use of BIOS services requires thunking interfaces between the (64bit) kernel and the 64/32bit parts of the bootloader that supply the bootops services. This is why the descriptor table contains a set of privileged code/data descriptors for calling into bootloader/BIOS during system startup.
- Using fast system call mechanisms (**`syscall`** and/or **`sysenter`**) mandates the shown ordering of code/data segments (i.e. the kernel data segment must directly follow the kernel code segment in the descriptor table, because this assumption is implicit in the way fast syscalls are set up).
-

6.2.Entering the Solaris/x86 kernel

Many hardware- and software-initiated events in Solaris require handling by the kernel. The following events enter the kernel via gates in the IDT:

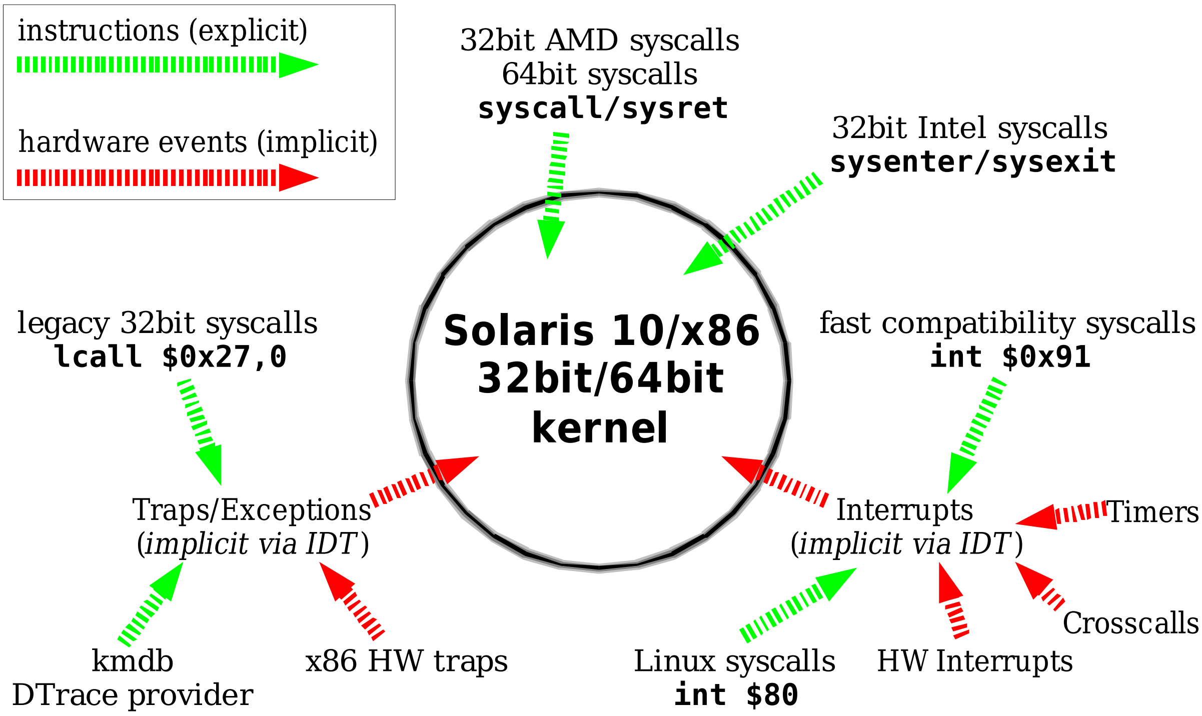


Illustration 2 - Entering the Solaris/10 kernel on x86 platforms

- All x86 interrupts, whether created by hardware (via APIC and/or IOAPIC) or via an explicit **int ...** instruction, call a handler in the kernel.
- All x86 hardware traps are redirected to the kernel.
- Debugging utilities (mdb/kmdb and other debuggers as well as DTrace) use explicitly trapping instructions (**ud2**, **int ...**) to control other programs.
- Legacy (classical 32bit) system calls use the **lcall $0x27,0** instruction to call a gate in the LDT.
  This requires using a call gate which has disadvantages: call gates don't save **FLAGS**, and neither can they disable interrupts before dispatch. Interrupt gates do both, but they must be in the IDT.
  In order to use an interrupt gate, the following trick is used: The “segment present” attribute in the LDT's syscall gate is cleared. On the attempt to call that gate, a **#NP** trap is raised and the legacy syscall is “doubly redirected” to the **#NP** trap handler from the IDT.
- The early adopter version of the Linux Application Environment (“Janus”) for Solaris provides a handler for classical Linux syscalls, which are done using the **int $80** instruction.

Solaris 10 uses fast system calls if the CPU model it's running on provides them. The fast system call mechanisms don't require gate descriptors. Instead, kernel **%cs**/**%ss** and **%rsp**/**%rip** (kernel stack pointer and system call handler address) are programmed using the machine-specific registers described in section 3.2.3.

The following system call mechanisms are supported:

- 32bit compatibility/legacy system calls via **lcall $0x27,0**.
  The 64bit kernel is redirects this to **#NP** by clearing the seg present bit.

- 32bit system calls via **syscall** instruction (on AMD CPUs that support it)
- 32bit system calls via **sysenter** instruction (on Intel CPUs that support it)
- 64bit system call via **syscall** instruction (all 64bit x86 CPUs have **syscall**)

Solaris 9/x86 and below do not use fast system calls.

The Interrupt Descriptor Table on Solaris 10/x86 contains the following entries in 64bit mode:

```
echo "idt0,0t256/an16+" | mdb -k | sed -e 's/:/::gate_desc/g' |
      mdb 24 | nawk '$1=="HANDLER" && f==1 {next} $1=="HANDLER" {f=1} {print}'
HANDLER                         SEL  DPL P TYP IST
div0trap                         28  0  + int  0
dbgtrap                          28  0  + int  0
nmiint                           28  0  + int  0
brktrap                          28  3  + int  0
ovflotrap                        28  3  + int  0
boundstrap                       28  0  + int  0
invoptrap                        28  0  + int  0
ndptrap                          28  0  + int  0
syserrtrap                       28  0  + int  1
resvtrap                         28  0  + int  0
invtsstrap                       28  0  + int  0
segnptrap                        28  0  + int  0
stktrap                          28  0  + int  0
gptrap                           28  0  + int  0
pftrap                           28  0  + int  0
resvtrap                         28  0  + int  0
ndperr                           28  0  + int  0
achktrap                         28  0  + int  0
mcetrap                          28  0  + int  0
xmtrap                           28  0  + int  0
invaltrap                        28  0  + int  0
[ ... ]
invaltrap                        28  0  + int  0
ivct32                           28  0  + int  0
[ ... ]
ivct125                          28  0  + int  0
dtrace_fasttrap                  28  3  + int  0
dtrace_ret                       28  3  + int  0
ivct128                          28  0  + int  0
[ ... ]
ivct209                          28  0  + int  0
fasttrap                         28  3  + int  0
ivct211                          28  0  + int  0
[ ... ]
ivct255                          28  0  + int  0
```

Hardware Exceptions/Traps

reserved vectors

Programmable Interrupts

As can be seen, all of these gates have the present bit set (“**+**”). Which as mentioned the LDT compatibility system call gate has not:

```
> ldt0_default+0x20::gate_desc
HANDLER                         SEL  DPL P TYP IST
sys_lcall32                      28  3      c  1
```

This means the compatibility syscall will cause a #NP trap and enter the kernel via **segnptrap()**. So the stub **sys_lcall32** can never be entered; if it yet would happen:

```
> sys_lcall32::dis
sys_lcall32:                    pushq  $0x0
sys_lcall32+2:                  pushq  %rbp
sys_lcall32+3:                  movq   %rsp,%rbp
```

```
sys_lcall32+6:                leaq   0x7(%rip),%rdi
sys_lcall32+0xd:              xorl   %eax,%eax
sys_lcall32+0xf:              call   +0x40ff1 <panic>
> sys_lcall32+0xd+7/S
0xfffffffffb8018e4:           sys_lcall32: shouldn't be here!
```

All other hardware exceptions have a similar structure:

- CPU dispatches the gate handler
- the gate handler may or may not do low-level work; in the end, it pushes a spoofed error code (if the gate itself didn't create a real one) and the exception number to the stack and calls **`cmntrap()`**.

An example:

```
> pftrap::dis
pftrap:                       pushq  $0xe
pftrap+2:                     jmp    -0x2b1b2 <_cmntrap>
> div0trap::dis
div0trap:                     pushq  $0x0
div0trap+2:                   pushq  $0x0
div0trap+4:                   jmp    -0x2a7c4 <_cmntrap>
```

The **`#PF`** trap entry has already been called with an error code on the stack, while the CPU doesn't push one for **`#DIV`** traps. This makes sure that the common trap handler starts with the same stack layout no matter how it is entered:

```
0xfffffe800074bfb8:           0xb            trap number
0xfffffe800074bfc0:           0x24           error code (may be spoofed)
0xfffffe800074bfc8:           0xbff0ff8c     %rip/%eip that caught the trap
0xfffffe800074bfd0:           0x3b           %cs at time of trap
0xfffffe800074bfd8:           0x10202        RFLAGS/EFLAGS before trap
0xfffffe800074bfe0:           0x8047b9c      %rsp/%esp at time of trap
0xfffffe800074bfe8:           0x43           %ss at time of trap
```

The bottom two values, as is the behaviour of gates on x86, are only present if:

- The CPU was operating in 64bit mode
- The CPU was operating in 32bit mode and the trap occurred while in usermode.

Remember, on 32bit x86 traps and interrupts which occurred when the CPU already was running in kernel do not record the kernel **`%ss/%esp`** – their values are implicitly known and the CPU “optimizes” gate dispatch by not recording these values.

Section 6.3 gives details what this means.

Interrupts (vectors 32..255) do something very similar:

```
> ivct32::dis
ivct32:                       pushq  $0x0
ivct32+2:                     pushq  $0x0
ivct32+4:                     jmp    -0x2a624 <_interrupt>
> ivct33::dis
ivct33:                       pushq  $0x0
ivct33+2:                     pushq  $0x1
ivct33+4:                     jmp    -0x2a634 <_interrupt>
> ivct255::dis
ivct255:                      pushq  $0x0
ivct255+2:                    pushq  $0xdf
ivct255+7:                    jmp    -0x2b417 <_interrupt>
```

which means that the same “input” stack layout is used for both **`_cmntrap()`** and **`_interrupt()`**., except that an interrupt number (starting at 0, ending at 0xdf = 233) replaces the trap number.

The reason for this behaviour is that the Solaris kernel creates a *common trapframe format* for all possible ways to enter the kernel.

Compare the 64bit code for **_cmntrap()** and **_interrupt()**:

entering kernel via _cmntrap()	***entering kernel via _interrupt()***

```
_cmntrap:      subq  $0xa8,%rsp               _interrupt:      subq  $0xa8,%rsp
_cmntrap+7:    movq  %r15,0x70(%rsp)          _interrupt+7:    movq  %r15,0x70(%rsp)
_cmntrap+0xc:  movq  %r14,0x68(%rsp)          _interrupt+0xc:  movq  %r14,0x68(%rsp)
_cmntrap+0x11: movq  %r13,0x60(%rsp)          _interrupt+0x11: movq  %r13,0x60(%rsp)
_cmntrap+0x16: movq  %r12,0x58(%rsp)          _interrupt+0x16: movq  %r12,0x58(%rsp)
_cmntrap+0x1b: movq  %r11,0x50(%rsp)          _interrupt+0x1b: movq  %r11,0x50(%rsp)
_cmntrap+0x20: movq  %r10,0x48(%rsp)          _interrupt+0x20: movq  %r10,0x48(%rsp)
_cmntrap+0x25: movq  %rbp,0x40(%rsp)          _interrupt+0x25: movq  %rbp,0x40(%rsp)
_cmntrap+0x2a: movq  %rbx,0x38(%rsp)          _interrupt+0x2a: movq  %rbx,0x38(%rsp)
_cmntrap+0x2f: movq  %rax,0x30(%rsp)          _interrupt+0x2f: movq  %rax,0x30(%rsp)
_cmntrap+0x34: movq  %r9,0x28(%rsp)           _interrupt+0x34: movq  %r9,0x28(%rsp)
_cmntrap+0x39: movq  %r8,0x20(%rsp)           _interrupt+0x39: movq  %r8,0x20(%rsp)
_cmntrap+0x3e: movq  %rcx,0x18(%rsp)          _interrupt+0x3e: movq  %rcx,0x18(%rsp)
_cmntrap+0x43: movq  %rdx,0x10(%rsp)          _interrupt+0x43: movq  %rdx,0x10(%rsp)
_cmntrap+0x48: movq  %rsi,0x8(%rsp)           _interrupt+0x48: movq  %rsi,0x8(%rsp)
_cmntrap+0x4d: movq  %rdi,0x0(%rsp)           _interrupt+0x4d: movq  %rdi,0x0(%rsp)
_cmntrap+0x52: xorl  %ecx,%ecx                _interrupt+0x52: xorl  %ecx,%ecx
_cmntrap+0x54: movw  %gs,%cx                  _interrupt+0x54: movw  %gs,%cx
_cmntrap+0x57: movq  %rcx,0xa0(%rsp)          _interrupt+0x57: movq  %rcx,0xa0(%rsp)
_cmntrap+0x5f: movw  %fs,%cx                  _interrupt+0x5f: movw  %fs,%cx
_cmntrap+0x62: movq  %rcx,0x98(%rsp)          _interrupt+0x62: movq  %rcx,0x98(%rsp)
_cmntrap+0x6a: movw  %es,%cx                  _interrupt+0x6a: movw  %es,%cx
_cmntrap+0x6d: movq  %rcx,0x90(%rsp)          _interrupt+0x6d: movq  %rcx,0x90(%rsp)
_cmntrap+0x75: movw  %ds,%cx                  _interrupt+0x75: movw  %ds,%cx
_cmntrap+0x78: movq  %rcx,0x88(%rsp)          _interrupt+0x78: movq  %rcx,0x88(%rsp)
[ ... ]                                       [ ... ]
_cmntrap+0xa4: cmpw  $0x28,0xc0(%rsp)         _interrupt+0xa4: cmpw  $0x28,0xc0(%rsp)
_cmntrap+0xad: je <_cmntrap+0xb2>             _interrupt+0xad: je <_interrupt+0xb2>
_cmntrap+0xaf: swapgs                         _interrupt+0xaf: swapgs
[ ... ]                                       [ ... ]
_cmntrap+0xd2: movq %rsp,%rbp                 _interrupt+0xb5: movq %rsp,%rbp
```

This is of course common code; on entry, the sequence of instructions does:

- write the general-purpose registers to the stack
- put segment registers **%ds/%es/%fs/%gs** on the stack (these have not yet been pushed by the CPU before dispatching the gate, like **%cs/%ss** were)
- (not shown) puts **%fsb** and **%gsb** to the stack
- determine whether the trap/interrupt happened in kernel (by looking at **%cs** from the trap frame).
  If not, call **swapgs** and (not shown) reinitialize the segment regs, completing the context switch to kernel mode.
- In the end, initialize the framepointer with the *trap frame address*.

In summary, all functions that enter the Solaris kernel write a common trap frame format. More examples on this will be given in chapter 6.

6.2.1.The double fault handler

The **#DF** handler (trap type **0x8**) is special. Double faults occur if the CPU attempts a privilege switch, but finds the kernel (ring 0) stackpointer invalid so that the attempt to write the trap return information (the bottom/HW part of the trapframe with the registers **%cs/%rip/RFLAGS/%ss/%rsp**) would cause a pagefault. Kernel stackpointer corruptions or kernel stack overflows cause double faults.

The gate that dispatches the #DF handler therefore must be special. Solaris uses:

- a *task gate* in 32bit mode. The hardware task switch “fixes” the kernel stackpointer by recreating it from **t_esp0** of the *double fault TSS*:
- the 64bit *Interrupt Stack Table* (**IST**) mechanism in 64bit mode. The 64bit TSS provides an array of seven alternate stackpointers that trap/interrupt gates can select from. The use of the **IST[]** forces a stack switch even if the trap/interrupt is not associated with a privilege switch, i.e. if it occurred in kernel.

```
> idt0+80::gate_desc
HANDLER                         SEL  DPL P TYP IST
syserrtrap                        28  0  + int  1
```

Both mechanisms ensure that the double fault handler starts with a known-good stack.

6.2.2.Fast system calls in Solaris Architecture-optimzed libc

Solaris 10 on x86 platforms detects the best available system call mechanism and actually allows entry into the kernel via one of the following methods:

- 64bit platforms are guarenteed to have the syscall/sysret instructions available. They're therefore used for system calls in 64bit mode.
- 32bit x86 CPUs by AMD (Athlon and upwards) have **syscall/sysret**.
  A specific 32bit libc that uses **syscall** is provided for these CPUs.
- 32bit x86 CPUs by Intel (Pentium-II and upwards) and other vendors have **sysenter/sysexit**. A specific 32bit libc that uses **sysenter** is provided for these.
- All 32bit x86 CPUs are guaranteed to allow the “classical” x86 UNIX system call mechanism via the **lcall $0x27,0** gate. This is supported as fallback.

6.3.Solaris/x86 VM architecture – x86 HAT layer

The role of the Solaris HAT (*hardware address translation*) layer is to provide architecture-dependent backend support for the virtual memory subsystem. On all architectures Solaris supports, the following functions declared in **`<vm/hat.h>`** will be called from the generic parts of the VM subsystem - they must be provided by the HAT. HAT functionality falls into one of the following categories:

1. operations on whole address spaces.
   Functions that allocate, free, swap in/out, duplicate per-as hat structures or query usage statistics per address space fall into that category.
2. operations on parts of address spaces (ranges of virtual addresses).
   Functions to map/unmap (termed load/unload) physical/device memory, functions that lock/unlock physical pages corresponding to a given virtual address range, query/modify VM attributes or pagesizes, retrieve physical addresses (PFNs) and share/unshare mappings between two HATs fall into this category.
3. operations on pages.
   Functions to query/modify attribute bits of a page, functions to translate and synchronize between VM attributes and architecture-dependent MMU attributes, and support for softlocking belong here.
4. HAT initialization and feature detection.
   The kernel during startup will call into these functions to initialize the architecture-dependent part of the VM subsystem, and to query for support of large pages, NUMA, (intimate) shared memory, and other features.

The HAT layer therefore decides how Solaris VM concepts (address spaces, segments, pages) are mapped to a given MMU architecture:

- Its initialization is responsible for creating the kernel address space, enumerate physical and device memory, and thereby also determines how the *kernel virtual address space* for a given architecture will look like.
- For a *user virtual address space*, the ABI for the architecture decides where code, data, mappings or stacks shall go, and of course the HAT's hooks for address space duplication/creation actually implement these rules when process contexts are created.

Given that context switching between processes and/or kernel also needs support from the HAT to “activate” a MMU context, the HAT will supply hooks to context switching for this. Since context switching is also architecture-dependent, this is not a standard (architecture-independent) HAT interface, but directly called from the context switching code for a specific architecture.

6.3.1.the 32bit x86 HAT before Solaris 10

6.3.2.post-S10 HAT

Both the port to the 64bit architecture and the need to support “modern” Solaris VM features on x86 platforms required a rewrite of the low-level VM support routines in Solaris 10. The result was a VM architecture that both scales to terabytes of physical memory and provides the following features which were not available on pre-S10 versions of Solaris/x86:

- unified code/binary – no more separate **`mmu32/mmu36`** modules to support the classical and PAE MMU modes. The x86 HAT is now integrated in **unix/genunix**.

- MMU context management is highly simplified. Instead of the threshold-based context generation and per-thread **%cr3**, the new HAT layer dynamically creates the MMU context register **%cr3**.
- support for ISM and DISM (flags **SHM_SHARE_MMU** and **SHM_PAGEABLE**)
- MPSS (multiple page sizes) support. The architecture allows 4kB and 2MB (4MB in non-PAE mode). This is accessible via the mechanisms mentioned in **mpss.so.1(1)**.
- MPO (NUMA, latency groups) support.
  The Hypertransport-based memory architecture of the AMD Opteron CPUs is non-uniform because access latency depends on the number of HT links and hops to the addressed memory bank. Users can access these features via **liblgrp(3LIB)**.
- support for **PROT_EXEC**. AMD Opteron CPUs know the "NX" page attribute bit to disable executability of a page of memory – which is something that classical x86 CPUs could not do due to "executability" being an attribute of the code segment – not the MMU page. Classical x86 therefore ignored **PROT_EXEC**. See chapter 3.
  On CPUs that support it, the S10 HAT will automatically use and enforce it.

  The support for **PROT_EXEC** can break legacy x86 applications that (being broken) assume every address to be executable even though **PROT_EXEC** wasn't requested for a mapping. Such applications should be fixed, but in cases where this is not possible the HAT honours a tunable in **/etc/system** to disable **PROT_EXEC** enforcement. Add "**set disable_NX=1**" in **/etc/system** and reboot.

  Additionally, the NX bit is only available if the MMU is operating in PAE mode. Since this is the only option in 64bit mode it can't cause problems there. But even in 32bit mode the S10 HAT will default to using PAE if it is available, in order to be able to support **PROT_EXEC**, while in S9 and below, PAE only was enabled if either requested explicitly or if the system had more then 4GB of physical memory. Incompatibilities e.g. with device drivers (or VMware – a well-known limitation of the latter is that it doesn't provide a guest OS with PAE support) may require to switch off PAE use in 32bit mode. This is possible by adding "**set force_pae_off=1**" to **/etc/system**.
- The x86 HAT supports *segkpm*, i.e. the ability of the Solaris kernel to create a complete in-kernel mapping for all of physical memory in order to speed up user/kernel data exchange and I/O.
  The support for segkpm is not 100% complete – *segmap* (the VM layer used to buffer filesystem I/O) is not yet using segkpm in the first-released version of S10.

Most operations on the HAT involve creating/locating/modifying pagetable entries (i.e. page attributes) for a given virtual/physical address (*forward* and *reverse* searches).

The x86 MMU itself is a multi-level hashtable (two levels for the classical 32bit MMU mode, three levels for PAE in 32bit mode and four levels for the 64bit MMU mode), so searching the physical page for a given virtual address could be done by walking the MMU hash directly (doing a forward search).

But this is both inefficient and insufficient:

- Walking the pagetable linkage requires up to five memory lookups and four hash calculations to get the table indices (assuming the 64bit mode with four tables). Given that in most cases, the higher-level tables (PML4, PDPT, PDT) will be sparse, it'd be desirable to have a faster way to look up the physical location for a virtual address.
- Pagetable addresses (page frame numbers) are physical – not virtual. In order to

search the pagetables directly, one would've to map them into virtual memory first.

- A reverse search (find the virtual address a given physical page is mapped to within a specific MMU context / address space) is not possible using the x86 pagetables.

The Solaris/x86 HAT solves these problems by introducing an abstraction layer on top of the physically-mapped page tables – the HTable mechanism.

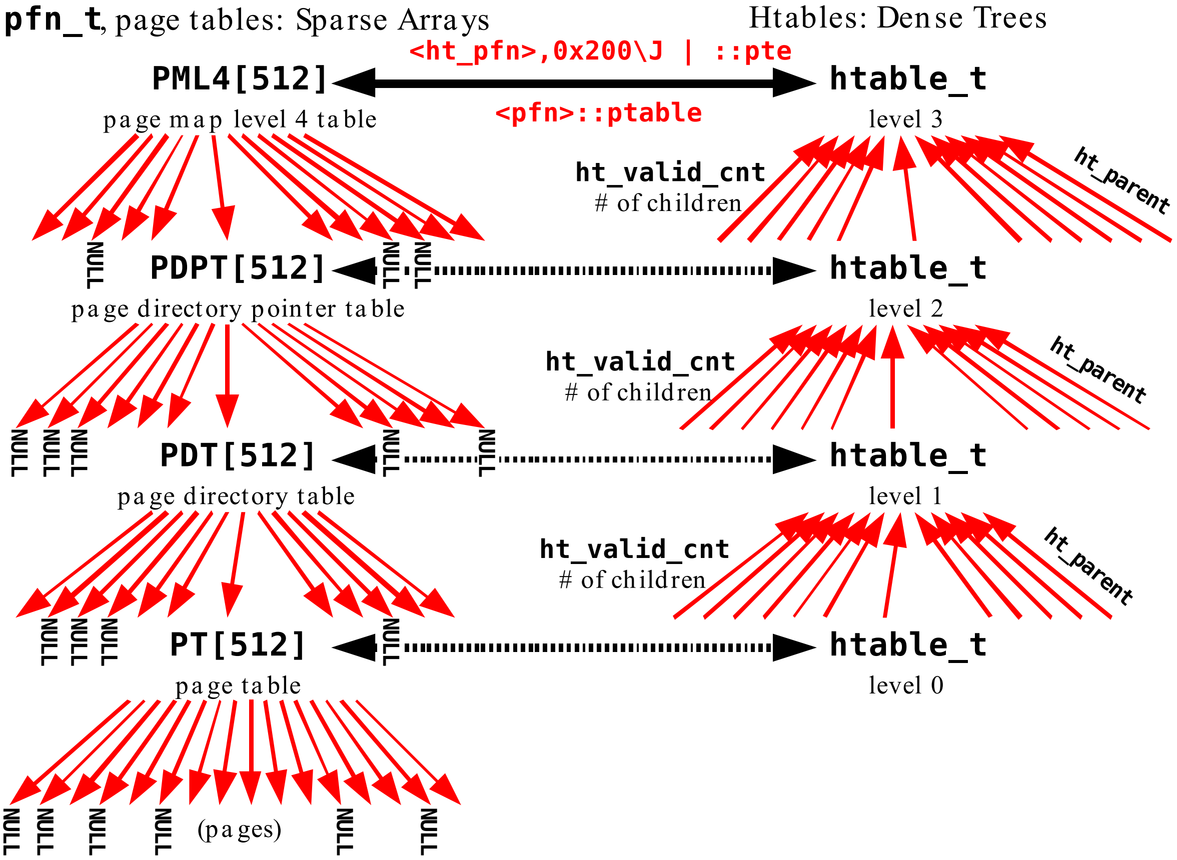


Illustration 3 - Entering the Solaris/10 kernel on x86 platforms

- Unlike MMU tables which are sparse, HTables are dense .
- There's one HTable for every non-NULL slot in the level 3/2/1 MMU tables.
- HTables count the number of mapped entries, i.e. the number of non-NULL PTEs in their associated page table.
- HTables allow reverse searches – a level 0 htable (corresponding to a MMU pagetable) has an **ht_parent** pointer at its level 1 htable, and so on.
- HTables are single-index hashed for fast mapping lookups.

HTables allow fast forward searches (VA PA) via the HAT's htable hash, and (slower) reverse searches (PA VA) via the **ht_parent** linkage.

6.4. Virtual Memory Layout

Both the 32bit and 64bit kernel on x86 share the address space between kernel and applications (i.e. a KERNELBASE parameter exists). For the 32bit virtual memory layout, nothing has changed in Solaris 10 vs. Solaris 9. If running on a 64bit kernel, though, 32bit applications are now able to use (almost) all of the 4GB virtual memory available to them.

The 64bit virtual memory layout (kernel/apps) looks like this:

```
> ::mappings ! tail -r
                           BASE             LIMIT             SIZE NAME
[ ARGSBASE     ffffffffffc00000 ................ ]
SEGDEBUGBASE   ffffffffff800000 ffffffffffc00000         400000 kdebugseg
KERNEL_TEXT    fffffffffe800000 ffffffffff0bc000         8bc000 ktextseg
valloc_base    fffffffffa200000 fffffffffe000000        3e00000 kvalloc
core_base      ffffffffc0000000 fffffffffa200000       3a200000 kvseg_core
segkmap_start  ffffffffbf000000 ffffffffc0000000        1000000 kmapseg
segkp_base     ffffffffb2800000 ffffffffbf000000        c800000 kpseg
kernelheap     fffffe8000000000 ffffffffb2800000    17fb2800000 kvseg
SEGKPM_BASE    fffffe0000000000 fffffe007feef000        7feef000 kpmseg
[ KERNELBASE   fffffd8000000000 fffffe0000000000 ]     user/kernel red zone
                                                       (unmapped)

> ffffffff850a0cf8::context; ::mappings ! tail -r
               fffffd7fffdfb000 fffffd7fffe00000           5000 [ anon ]
[  many many more libraries and anon segments ...  ]   mappings and thread stacks

                                            application stacks/mappings grow
                                            downwards towards VA hole

[              0000800000000000 ffff800000000000 ]     address space hole

                                 application heap grows upwards towards VA hole

(application heap)        48a000       aef000          665000 [ anon ]
(application static data) 479000       48a000           11000 /usr/bin/amd64/mdb
(application text)        400000       469000           69000 /usr/bin/amd64/mdb
[                              0       400000 ]        lower red zone (unmapped)
```

See **<sys/machparam.h>** for a detailed explanation. The main elements are:

- At the *bottom end* of the user 64bit address space is an *unmapped 2MB page*. This is specified by the AMD64 UNIX ABI – application text load address is 2MB, **0x400000**.
- Solaris has chosen to give the majority of the 64bit VA range to the application.
  - The user heap will start after the application text/data segments and grows upwards into the VA hole.
  - The user stacks and file mappings are created below KERNELBASE and grow down into the VA hole.

  Over all, a 64bit application on Solaris 10 can map up to ~250TB of memory.
- At the *top end* of the user address space is another *unmapped 2MB page*.
- The *kernel address space* begins at **KERNELBASE** – with an unmapped 'huge' page, i.e. a NULL PML4 entry, 512GB. Kernel segments extend above that. The kernel itself currently occupies four slots in the PML4 table, i.e. 2TB virtual memory.

6.5.Context switching

Taking a running thread off the CPU and resuming a previously sleeping one is done by the architecture-dependent function **`resume()`** in Solaris. On x86 platforms, this executes code as given in the following flow diagram:

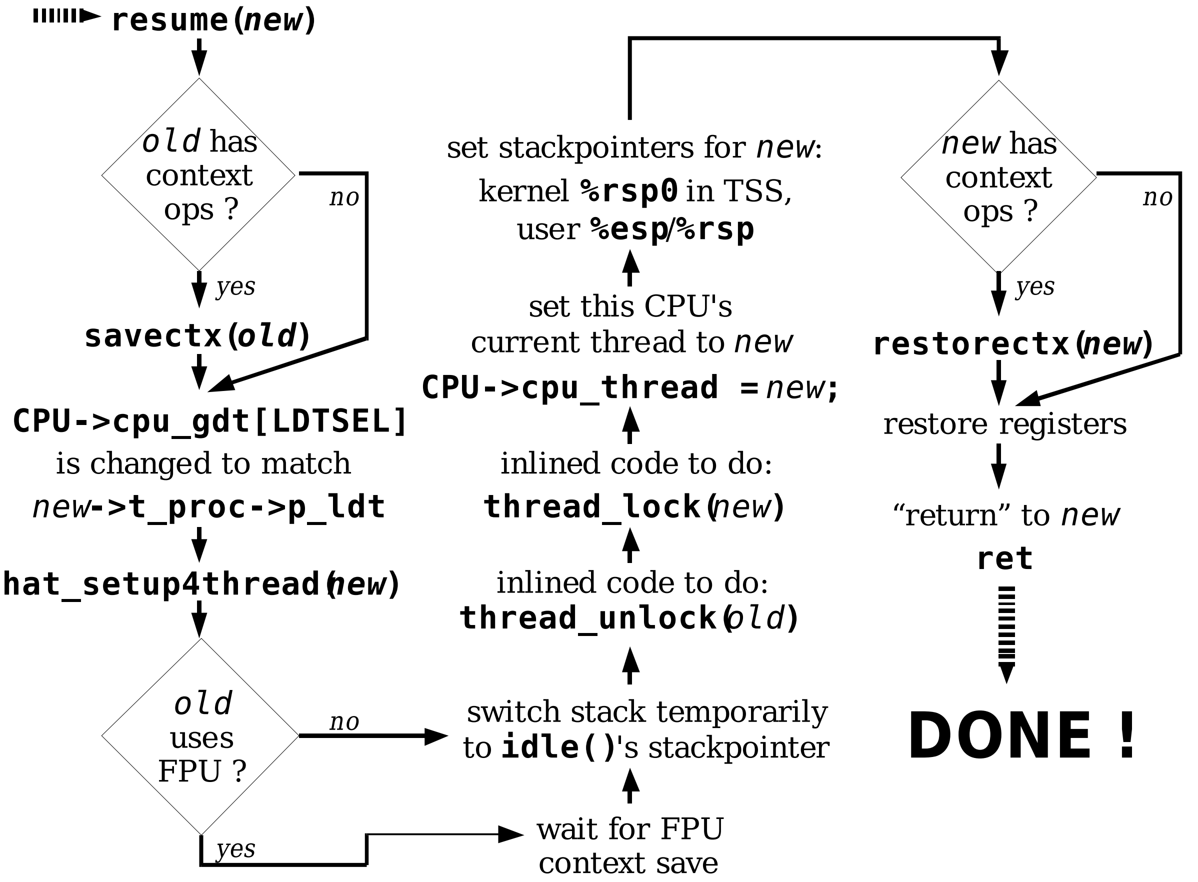


Illustration 4 - steps in context switching

The code is fully preemptive. When **`resume()`** is entered, the `old` onproc thread is already quiesced (its user registers have been saved), but the stack we're running on still belongs to `old`. Actions executed by **`resume()`** in sequence are:

1. If the `old` thread (i.e. the one currently marked **`T_ONPROC`** for this CPU) has context ops installed (`old`**`->t_ctx != NULL`**), the **`savectx()`** function from there will be called. If the thread hasn't an extended context this can be bypassed.
   A thread will have context ops e.g. if it used floating point registers. On first use of the FPU by a newly-started thread, a trap will occur and the trap handler installs context ops for the thread that save/restore the floating point context.

2. Because processes are free to create segments in their *Local Descriptor Table*, the kernel has to change the LDT segment descriptor in the CPU's Global Descriptor Table so that its base address matches that of the *new* thread's LDT.

3. The VM/MMU context (user mappings) is still that of the `old` thread. In order to initialize the MMU in preparation for running *new*, **`hat_setup4thread()`** is called. This creates and activates the MMU context register **`%cr3`** from *new*'s HAT.

4. Floating point context saves (**`fxsave`** instruction) on x86 platforms are asynchronous. If the context ops have kicked off a floating point context save, we'll now wait for that to finish (**`fwait`** instruction).
   There is no "**`waitctx()`**" context op ... so this is inlined.

5. In order to remove dependency on the *old* thread's kernel stackpointer, which is still in use by interrupts occurring during **resume()**, the stackpointer **%esp**/**%rsp** is *temporarily* changed to that of this CPU's *idle thread*.
6. All of the state of the *old* thread is now saved and it can be put off the CPU. **T_ONPROC** will be removed from its **t_flags**, and the thread lock can be released. For speed, the code to do this is inlined.
7. Threads currently executing on the CPU are locked, so **resume()** now performs, again inlined, a **thread_lock()** on *new*. Once done, it marks the thread **T_ONPROC**.
8. **curthread** aka **CPU->cpu_thread** still point to the *old* thread. This is now changed.
9. The *new* threads kernel stackpointer is read from **new->t_sp** and put into the CPU's TSS, as well as into **%esp**/**%rsp**. We're no longer running with **idle()**'s stack now, from here on interrupts occuring will use *new*'s stack.
10. The return address is put on the stack and execution of *new* (in kernel mode) is resumed via **ret** instruction. **resume()** is done.
11. Once the kernel stack is unwinded, system call and/or trap exit handling does **iret**/**sysret**/**sysexit** (whatever appropriate) to continue execution in user mode.

6.6.Supporting Multiple CPUs

6.6.1.Locking Primitives and Atomic Operations

6.6.2.SMP Interrupt Management and Crosscalls

6.6.3.NUMA

6.7.isaexec – Creating 32/64bit-specific applications

7.Solaris/x86 Crashdump Analysis

From “The Tao of Programming”:

Thus spake the master programmer:

"When you have learned to snatch the error code from the trap frame, it will be time for you to leave."

7.1.Debugging tools for core-/crashdump analysis

Solaris/x86, having been the ugly duckling for the longer part of its history, received much less attention by engineers doing debugging work on kernel-level than its SPARC brethren. The choice of debugging tools for Solaris/x86 therefore is much more limited than for Solaris/SPARC – unbundled (i.e. not distributed with Solaris itself) debugging tools used to be very uncommon, and still not all packages that see frequent use on SPARC have been ported to Solaris/x86. Most notably, the SolarisCAT (**`scat`**) kernel debugger that is very popular both inside and outside of Sun is not (yet) available under Solaris/x86.

Fortunately, the set of debugging utilities shipped with Solaris by default is available for x86 and SPARC architectures alike, with comparable functionality – which has grown significantly in Solaris 10 compared with what was available at the time of Solaris 8. Useful utilities for kernel debugging or core- and crashdump analysis under Solaris 10 include:

- The modular debugger, **`mdb`**.
  `mdb` is the main tool used for crashdump analysis on Solaris 10/x86. The default feature set of mdb is generic and identical on SPARC and x86 architectures. Since **`mdb`** is extensively documented in the standard Solaris manuals, the reader is referred to those for details on **`mdb`** usage.
  All examples given in the following sections will use **`mdb`**.
  There are special-purpose versions of **`mdb`** available:
  - The runtime kernel debugger, **`kmdb`**. This version runs in kernel context and allows things like kernel break- and watchpoints. **`kmdb`** is bundled with Solaris – in fact, enabling it at runtime can be done by calling **`mdb`** with the '**`-K`**' option.
  - *Enhanced mdb* – **`mdb+`**. This is a feature-extended mdb that is distributed internally at Sun as part of the *kenv debugging suite*.
    For those having to analyze crashdumps of Solaris 8/x86 or Solaris 9/x86 systems, the kenv package and **`mdb+`** have the advantage of supplying (most of) the functionality that is there in **`mdb`** on Solaris 10 but not on older bundled versions.
  - Automated Crash Tool – **`act`**. This package includes a loadable mdb module that extends **`mdb`** with several dcmds, among them a **`::findstack`** workalike that can display function arguments – even in 64bit mode. Examples later.
- The dynamic tracing framework, **`dtrace`**.
  While **`dtrace`** itself isn't targeted specifically at post-mortem analysis, it's of course available on Solaris/x86 – the feature set is generic, i.e. dtrace scripts will mostly be portable between Solaris/SPARC and Solaris/x86. **`mdb`** is able to extract **`dtrace`** records from crashdumps, in case the system paniced during tracing.
- The **`proc`** tools. Some of them (e.g. **`pmap`**) operate on crash- and/or coredumps while others are again targeted at runtime application analysis. Again, the ptools are generic and available for 32/64bit SPARC/x86.
- For application coredump analysis only, the **`dbx`** (Sun Workshop) and **`gdb`** (GNU)

debuggers are both available in 32bit and 64bit versions for Solaris 10/x86. People familiar with either won't need to re-learn when looking at application coredumps.

- The default disassembler, **`/usr/ccs/bin/dis`**. To inspect compiler-generated assembly code in binaries or dumps, **`dis`** can be a useful tool, but it isn't usable interactively and won't see much use in post-mortem analysis. Its biggest advantage is that it allows simple inspection of machine code along with the disassembler output, since by default it displays instruction binary code and mnemonic side by side.

This selection is biased towards crashdump analysis and kernel debugging and by no means complete. Please refer to material on generic low-level troubleshooting on Solaris for more choice ...

7.1.1.The Solaris/x86 boot debugger before Solaris 10

7.2.Troubleshooting system hangs on Solaris/x86

Machines based on x86 CPUs do not have the SPARC-style OBP interface that allows low-level system interaction from the “ok prompt”. This of course means that there is no equivalent on x86 platforms for either the **`<Stop+A>`** key combination to enter OBP interaction mode or the **`sync`** OBP command that forces a system crashdump to be written.

By default, Solaris/x86 machines have no capability for “break”ing the system. The machine ignores keyboard or serial line break sequences.

The presence of the kernel debugger, **`kadb`** (pre-Solaris 10), **`kmdb`** (Solaris 10 and later), is therefore a prerequisite.

7.2.1.Loading the kernel debugger – kadb/kmdb

The kernel debugger on Solaris/x86 can be automatically loaded *during boot* using the **`eeprom(1M)`** command:

- On Solaris 9 and below:
  `eeprom boot-file=kadb`
- On Solaris 10 and above:
  `eeprom boot-file=kmdb`
  `eeprom boot-args=-k` (suggested)

The disadvantage to this is that it requires a reboot in order to activate the kernel debugger. On Solaris 10, one of the enhancements in **`kmdb`** therefore is the ability to load the kernel debugger *at runtime*. Use:

```
mdb -K
```

(capital K) as root user to load **`kmdb`** at any time. The system will drop to the kmdb prompt and can be continued, **`:c`**, to resume execution.

Once the debugger is loaded via either of the above ways, it will catch **`<F1+A>`** (if a graphics console is used – there is no **`<Stop>`** key on PC keyboards), resp. serial breaks (if a serial console is used).
Also, the alternate break sequence, **`kdb -a,`** can then be set.

Note that on x86 systems, few graphics drivers support switching to textmode when the user presses **`<F1+A>`** to enter the kernel debugger. The system will appear hung, but **`kadb/kmdb`** is sitting invisibly in the background waiting for input. It may be necessary to type debugger commands blindly !

7.2.2.Forcing system crashdumps

In order to create forced crashdumps from hung Solaris/x86 systems, the following methods are possible:

- If the system is soft hung and still allows interaction e.g. via console or network logins, **`savecore -L`** (to create a live crashdump without rebooting the machine) or **`reboot -d`** (to force a reboot together with a crashdump being written) can be used.
- If interaction is no longer possible, the kernel debugger (**`kadb/kmdb`**) must be used. On Solaris/x86, console (screen) logins use <F1+A> as the keyboard break sequence, and serial consoles accept breaks – provided the kernel debugger has been loaded.
  On the kernel debugger's prompt, a crashdump can be forced using e.g. the following method:

Write a NULL into the **%eip/%rip** register, and continue:

0>eip;:c (32bit), or
0>rip;:c (64bit).

The machine panics immediately and drops to the debugger prompt again. Continue another time,

```
:c
```

to write the crashdump and have the system reboot.
This is documented in Sun InfoDoc #15553.

7.2.3.Hard hangs on x86

There are cases where the attempt to enter the kernel debugger fails – the system is then *hard hung*. Breaking out of hard hangs and forcing a crashdump from one is not always possible, but the following techniques can be applied:

- Enable the **deadman** timer. If the following option is set in **/etc/system**:

  ```
  set snooping=1
  ```

  the high-resolution timer interrupt on the machine will count down a watchdog timer that is reset periodically by the Solaris **clock()** (low-resolution timer) code. If clock processing / scheduling is stuck, deadman will time out and cause a system panic.
  This is described in Sun InfoDoc #13258.

- Some machines may have an XIR/NMI (eXternally-Initiated / Non-Maskable Interrupt) switch. If the system has a NMI switch, Solaris can be told to use it via the following **/etc/system** parameters:

set pcplusmp:apic_panic_on_nmi=1	create a crashdump on NMI
set pcplusmp:apic_kmdb_on_nmi=1	enter kmdb interactively on NMI

  Since *NMI interrupts cannot be blocked* (except by the CPU disabling *all* interrupts), this may break out of hard hangs that not even deadman (which relies on the high-resolution timer interrupt) can help with.

- In the future, Sun may provide a configurable NMI facility, to allow optionally using the system's power button as NMI switch on machines that support ACPI.
  To inquire about this, contact the Solaris developers via the mailing lists on http://www.opensolaris.org.

7.2.4.Setting up the serial console on Solaris/x86

Some x86 systems (server type mainboards, for the most part) support serial consoles on BIOS level. This is usually called *console redirection*, and if it is available it allows interaction with the machine during the entire early stages of the boot process – until the Solaris 2nd stage bootloader takes over.

The Solaris bootloader and kernel must be told to use serial consoles – Solaris/x86 platforms will *not* default to using the serial line if no keyboard is found. The required steps to set up a serial console on Solaris/x86 machines are as follows.

- Configure, if available, the serial console redirection in the BIOS. Since PC BIOSses don't use the same defaults as SPARC systems , it's advisable to set the parameters to the usual defaults of Solaris serial consoles, 9600 Baud, 8bit, no parity.
- Connect a serial line and test whether you can interact with the BIOS.

- Next, boot Solaris, into the *Device Configuration Assistant* (DCA).
  Once the following messages appear on the screen:

  ```
  If the system hardware has changed, or to boot from a different
  device, interrupt the autoboot process by pressing ESC.
  ```

  press **<ESC>** to enter the DCA.
- Progress through the DCA to the *"Boot Solaris"* screen.
- Select the **F4_Boot_Tasks** option, then select the **View/Edit Property Settings** option from the menu and use **F2_Continue**.
  Some serial terminals don't have function keys or don't submit the keycodes properly on the serial line. In this case, **<ESC>** followed by a digit can be used.
- Scroll down the menu and select **input-device**, then use the **F3_Change** key to change it to **ttya**.
- Do the same for **output-device**, then go back to *"Boot Tasks"* using **F2_Back**.
- Return to *"Boot Solaris"* via **F3_Back**, and continue there.

If your BIOS does not allow serial console redirection, Solaris can still be configured to use the serial line for in/output – starting with the DCA.

5. Open the file **bootenv.rc** with a text editor.
   On a Solaris/x86 that was installed using screen/keyboard, it can be found in **/boot/solaris/bootenv.rc**.
   For a system that should be installed with Solaris using a serial console, edit **bootenv.rc** on the DCA floppy, or in the PXE boot environment if network boot is used.
6. Search for the following two lines and have them refer to **ttya**:

   ```
   setprop output-device='ttya'
   setprop input-device='ttya'
   ```

This enables the serial console from the point on where the Solaris boot loader has been started. BIOS interaction is not possible on such systems, but the serial console under Solaris will be operable.

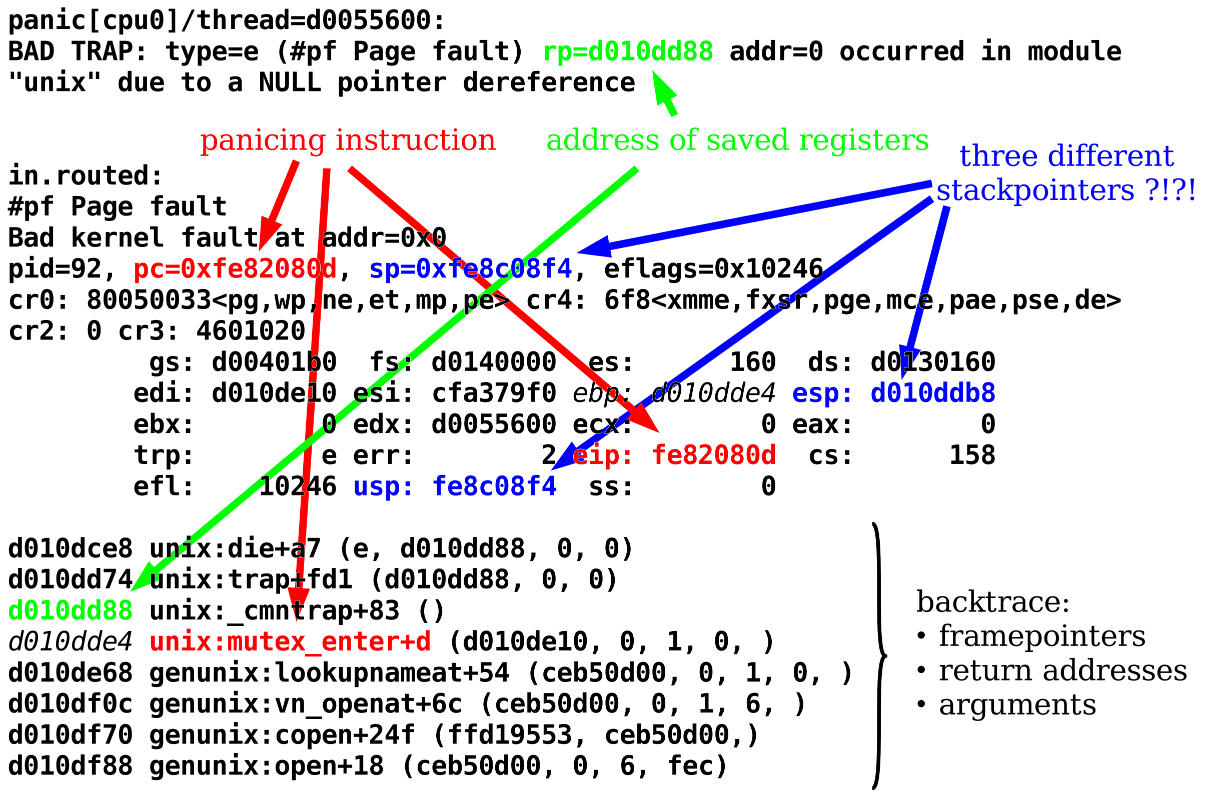
7.3.32bit kernel crashdump analysis – a well-known example

Before we dig deep and use real-world examples of crashdumps from actual reported and fixed bugs, we'll create a crashdump in a well-known, controlled way. On a Solaris 10/x86 system running a 32bit kernel, panic the system using the following command:

```
mdb -kw <<END
rootdir/W 0
END
```

7.3.1.Panic messages

The system very shortly after is going to panic with messages similar to these:



```
panic[cpu0]/thread=d0055600:
BAD TRAP: type=e (#pf Page fault) rp=d010dd88 addr=0 occurred in module
"unix" due to a NULL pointer dereference

in.routed:
#pf Page fault
Bad kernel fault at addr=0x0
pid=92, pc=0xfe82080d, sp=0xfe8c08f4, eflags=0x10246
cr0: 80050033<pg,wp,ne,et,mp,pe> cr4: 6f8<xmme,fxsr,pge,mce,pae,pse,de>
cr2: 0 cr3: 4601020
         gs: d00401b0  fs: d0140000  es:      160  ds: d0130160
        edi: d010de10 esi: cfa379f0 ebp: d010dde4 esp: d010ddb8
        ebx:        0 edx: d0055600 ecx:        0 eax:        0
        trp:        e err:        2 eip: fe82080d  cs:      158
        efl:    10246 usp: fe8c08f4  ss:        0

d010dce8 unix:die+a7 (e, d010dd88, 0, 0)
d010dd74 unix:trap+fd1 (d010dd88, 0, 0)
d010dd88 unix:_cmntrap+83 ()
d010dde4 unix:mutex_enter+d (d010de10, 0, 1, 0, )
d010de68 genunix:lookupnameat+54 (ceb50d00, 0, 1, 0, )
d010df0c genunix:vn_openat+6c (ceb50d00, 0, 1, 6, )
d010df70 genunix:copen+24f (ffd19553, ceb50d00,)
d010df88 genunix:open+18 (ceb50d00, 0, 6, fec)
```

Illustration 1 - Solaris/x86 panic messages on a 32bit kernel

After rebooting, the crashdump is created and can be analyzed.

The panic message consists of the following three parts:

- The panic string itself. This tells which thread (and on what CPU) was running when the system encountered the panic. In addition to that, it gives a “panic oneliner” that's either a deliberate message from callers of **cmn_err(CE_PANIC,...)**, or the generic **BAD TRAP: ...** message that's printed during unexpected kernel faults. Bad traps give the location of where the general purpose registers were saved at the time of the trap (the *trap frame* address) in the **rp=...** field:

  ```
  panic[cpu0]/thread=d0055600:
  BAD TRAP: type=e (#pf Page fault) rp=d010dd88 addr=0 occurred in module
  "unix" due to a NULL pointer dereference
  ```

- A printout of general-purpose and CPU control register contents at the time of the panic. This is only meaningful for BAD TRAPs, and also only shown for BAD TRAPs.

It's a prettyprint version of the saved registers at **rp=...**. The following comparison shows this clearly.

- First, the register dump from the panic message:

```
in.routed:
#pf Page fault
Bad kernel fault at addr=0x0
pid=92, pc=0xfe82080d, sp=0xfe8c08f4, eflags=0x10246
cr0: 80050033<pg,wp,ne,et,mp,pe> cr4: 6f8<xmme,fxsr,pge,mce,pae,pse,de>
cr2: 0 cr3: 4601020
         gs: d00401b0  fs: d0140000  es:      160  ds: d0130160
        edi: d010de10 esi: cfa379f0 ebp: d010dde4 esp: d010ddb8
        ebx:        0 edx: d0055600 ecx:        0 eax:        0
        trp:        e err:        2 eip: fe82080d  cs:      158
        efl:    10246 usp: fe8c08f4  ss:        0
```

- Second, explicitly printing the address given via **rp=...** as **struct regs**:

```
> d010dd88::print -a "struct regs"
{
    d010dd88 r_gs = 0xd00401b0
    d010dd8c r_fs = 0xd0140000
    d010dd90 r_es = 0x160
    d010dd94 r_ds = 0xd0130160
    d010dd98 r_edi = 0xd010de10
    d010dd9c r_esi = 0xcfa379f0
    d010dda0 r_ebp = 0xd010dde4
    d010dda4 r_esp = 0xd010ddb8
    d010dda8 r_ebx = 0
    d010ddac r_edx = 0xd0055600
    d010ddb0 r_ecx = 0
    d010ddb4 r_eax = 0
    d010ddb8 r_trapno = 0xe
    d010ddbc r_err = 0x2
    d010ddc0 r_eip = 0xfe82080d
    d010ddc4 r_cs = 0x158
    d010ddc8 r_efl = 0x10246
    d010ddcc r_uesp = 0xfe8c08f4
    d010ddd0 r_ss = 0
}
```

- The contents of the general-purpose/segment register dump from the panic message and the *trap frame* are obviously identical. But the register dump contains more information.
  - It repeats what the BAD TRAP panic string already showed – it's a pagefault, **#PF** / type 0xe in x86 terms, attempting to dereference a NULL pointer, **addr=0**. The process that happened to be executing the code was in.routed and it ran under PID 92. That's complementary information to the panicing thread's address that the panic message already has shown.
    For a pagefault, as mentioned in chapter 3, the pagefault address register, **%cr2**, will contain the fault address. This therefore must be identical with the **addr=...** field – if the bad trap was a pagefault. The two values won't necessarily be equal for bad traps of some other type.
  - We also are given the address of the assembly instruction that actually caused the bad trap. It's explicitly printed as **pc=...**, and, since on x86 the program counter is a register, the same value of course also found in the register dump/trap frame under **eip**. Unfortunately, the symbol name/offset is not given, so look manually:

```
> 0xfe82080d/ai
```

```
mutex_enter+0xd:lock cmpxchgl %edx,(%ecx)
```

  - Several stackpointers are given, **sp=...**, **esp**, and **usp**.
    The fields **sp=...** and **usp** are actually found to be equal.

    But ***none of them*** is the stackpointer at the time of the panic !
    The attempt to backtrace from either of them fails – no output:

    ```
    > 0xd010ddb8$C
    > 0xfe8c08f4$C
    ```

  - We also notice that the value of the saved **%esp** register actually is the address of the **r_trapno** field *within the trapframe* ! This value can't have been the kernel's **%esp** before the trap if it points at a location that only was accessed after the trap actually occurred.

- This obviously needs some further explanation. How can one, in a 32bit Solaris/x86 crashdump, *find the stackpointer at the time when the trap occurred* ?

- Chapter 3 has already given the answer – x86 CPUs, when encountering a trap while *already running in privileged mode*, won't save **%esp**/**%ss** when writing a hardware trap frame – because the CPU doesn't need to switch stacks as it didn't do a privilege switch. It uses whatever the kernel stackpointer was and puts the trap return information there. The first thing that is written is **eflags** – *not* **%uesp**/**%ss** – and the value of the *stackpointer at the time of the panic* must be *deduced* – as the address of the word on the stack immediately before that. More on that later.

- The third part is the panic backtrace:

```
d010dce8 unix:die+a7 (e, d010dd88, 0, 0)
d010dd74 unix:trap+fd1 (d010dd88, 0, 0)
d010dd88 unix:_cmntrap+83 ()
d010dde4 unix:mutex_enter+d (d010de10, 0, 1, 0, )
d010de68 genunix:lookupnameat+54 (ceb50d00, 0, 1, 0, )
d010df0c genunix:vn_openat+6c (ceb50d00, 0, 1, 6, )
d010df70 genunix:copen+24f (ffd19553, ceb50d00,)
d010df88 genunix:open+18 (ceb50d00, 0, 6, fec)
```

12.Again, several of the values in there have already been seen on previous parts of the panic message:

```
BAD TRAP: type=e (#pf Page fault) rp=d010dd88 addr=0 occurred in module[
... ]
        edi: d010de10 esi: cfa379f0 ebp: d010dde4 esp: d010ddb8
[ ... ]
        trp:        e err:        2 eip: fe82080d  cs:      158
[ ... ]
d010dd88 unix:_cmntrap+83 ()
d010dde4 unix:mutex_enter+d (d010de10, 0, 1, 0, )
```

1. The address of the saved registers, **rp=...**, is reported as the *framepointer* of **_cmntrap()**. This is why it is called *trap frame*.

2. The saved framepointer **%ebp** from the panic registers is also reported in the stacktrace as the framepointer of the line in the backtrace after the trapframe.

3. So is the faulting instruction's address, **%eip**, for which the corresponding symbol+offset is shown in the backtrace as well.

13.Finally, we note that in the stacktrace, different numbers of arguments are printed by the debugger for different functions – e.g. four are shown for **open()**, two for **copen()**, none for **_cmntrap()** or three for **trap()**.
How does can the debugger know how many arguments a function has ?
Will it be correct about this, and how would we verify ?

7.3.2.Immediate cause for the panic
Repetitorium on 32bit x86 traps

Looking at the data from the panic messages, we find that the panic happened executing just the third instruction of **`mutex_enter()`**:

```
mutex_enter:      movl   %gs:0x10,%edx
mutex_enter+7:    movl   0x4(%esp),%ecx
mutex_enter+0xb:  xorl   %eax,%eax
mutex_enter+0xd:  lock cmpxchgl %edx,(%ecx)
```

Let's try to understand this code – what does it do ?

First thing to notice is that this isn't a function like most others – it doesn't have a function prologue where it'd initialize a framepointer or allocate stackspace for itself. This obviously is performance-critical, handwritten assembly code. First thing is does is to load the word at address **`%gs:0x10`** into **`%edx`**. As chapter 5 has shown, Solaris uses the %gs segment for the current CPU structure, and offset **`0x10`** there is **`cpu_thread`**. So **`%gs:0x10`** is simply the Solaris/x86 assembly version of **`curthread`**. But let's verify this again, just to crosscheck. We get **`%gs`** from the panic messages:

```
panic[cpu0]/thread=d0055600:
[ ... ]
         gs: d00401b0  fs: d0140000  es:      160  ds: d0130160
```

Remember segment registers contain 16bit values only – the upper 16bits of the reported **`%gs`** contents therefore are irrelevant. Our **`%gs`** is **`0x1b0`**. Since the lower three bits are clear, this is a privileged segment within the GDT, so we need to find the GDT for the panicing CPU. As the panic message tells it's CPU 0, so we can look into **`struct cpu`** for this and find the GDT location from there:

```
> ::cpuinfo
 ID ADDR     FLG NRUN BSPL PRI RNRN KRNRN SWITCH THREAD   PROC
  0 fec22364  1b    0    0  59   no    no t-0    d0055600 in.routed
> ::offsetof cpu_t cpu_m
offsetof (cpu_t, cpu_m) = 0x4cc
> fec22364+0x4cc::print "struct machcpu" mcpu_gdt
mcpu_gdt = gdt0
> gdt0+1b0::print user_desc_t usd_hibase usd_midbase usd_lobase
usd_hibase = 0xfe
usd_midbase = 0xc1
usd_lobase = 0xe88c
> 0xfec1e88c::whattype
fec1e88c is fec1e88c+0, struct cpu [1]
> 0xfec1e88c+10/X | ::whatis
d0055600 is d0055600+0, allocated as a thread structure
```

This fits – the segment descriptor obviously points at the CPU structure, and the onproc thread for this CPU is the panic thread. %edx from the panic registers,

```
        ebx:        0 edx: d0055600 ecx:        0 eax:        0
```

also confirms our analysis so far. The next instructions are:

```
mutex_enter+7:    movl   0x4(%esp),%ecx
mutex_enter+0xb:  xorl   %eax,%eax
```

The **`xorl`** obviously zeroes **`%eax`**. The instruction before is not so obvious, but we still know what it does. It loads the *first argument* of **`mutex_enter()`** from the stack. Remember chapter 2:

- Before calling somebody else, a function in 32bit x86 puts arguments on the stack.
- The **`call`** instruction itself puts a return address onto the stack.

- At the time execution starts at the called function, **`(%esp)`** is the return address, and **`+4(%esp)`**, **`+8(%esp)`** and so on will be the arguments.

In common functions, the prologue will initialize a framepointer and argument access after that is then done via **`%ebp`** – but as indicated, **`mutex_enter()`** doesn't bother with this and therefore uses **`%esp`** directly.

And there we are again – how can I know what the stackpointer was ? The messages show the **`NULL`** in **`%ecx`**, but *none* of the “stackpointers” in the panic messages is consistent with that:

```
[ ... ]
BAD TRAP: type=e (#pf Page fault) rp=d010dd88 addr=0 occurred in module "unix"
due to a NULL pointer dereference
[ ... ]
pid=92, pc=0xfe82080d, sp=0xfe8c08f4, eflags=0x10246
[ ... ]
        edi: d010de10 esi: cfa379f0 ebp: d010dde4 esp: d010ddb8
        ebx:        0 edx: d0055600 ecx:        0 eax:        0
[ ... ]
> 0xfe8c08f4+4/X
lookuppnat+0x93:e8530c43
> d010ddb8+4/X
0xd010ddbc:     2
```

At **`4(%esp)`**, neither gives NULL so something isn't right here – these addresses aren't stackpointers.

As mentioned before and in chapter 3, the reason for this is the way 32bit x86 CPUs create trapframes if trap handler and trapping instruction are running at the same privilege level. In 32bit x86:

The stackpointer at the time of the bad trap must be deduced indirectly.

Revisit chapter 3 and how trap handling in x86 works.

- a trap (hardware exception) occurs.
- the x86 CPU checks the corresponding *gate* in its *Interrupt Descriptor Table* and from there determines both:
  - the address of the trap handler
  - the privilege level the handler will run at. For Solaris, that's ring 0 – kernel mode.
- ***If and only if*** the trap occurred while the CPU was executing code at a different privilege level, the CPU will switch stacks.
  In other words: If the trap occurred in user mode, the CPU loads the kernel stack pointer (and the kernel stack segment register) from its TSS structure and switches to that.
  In that case only, it writes the previous privilege level's (user) **`%ss`** and (user) **`%esp`** to the kernel stack.
- But this wasn't the case here. The trap occurred in kernel mode, and we already have a valid kernel stack to start with. No stack switch is done.
- Before the CPU finally dispatches execution to the trap handler, it writes to the stack the information needed to resume execution after handling it:
  - the contents of the **`EFLAGS`** register (processor condition codes and state)
  - the **`%cs`** register and the address of the trapping instruction, **`%eip`**.
  - Since this was a pagefault, an *error code* is written as auxilliary information.

- Only then will the gate entry point be kicked off.

This means that the hardware part of the trap frame, **EFLAGS/%cs/%eip**, is written directly to wherever the kernel stackpointer was at the time of the trap.

What happens next ? Let's find out by looking into the IDT for the **#PF** (type **0xe**) trap:

```
> ::offsetof cpu_t cpu_m
offsetof (cpu_t, cpu_m) = 0x4cc
> ::offsetof "struct machcpu" mcpu_idt
offsetof (struct machcpu, mcpu_idt) = 0x58
> 0xfec1e88c+0x4cc+0x58/X | ::whatis
fec1f160 is idt0+0 in unix's data segment
> fec1f160+(8*e)::gate_desc
HANDLER                        SEL  DPL P TYP STK
pftrap                          158  0  + int  0
> pftrap::dis
pftrap:          pushl  $0xe
pftrap+2:        jmp    -0x1d9c2 <_cmntrap>
```

So the handler is **pftrap()**, which just pushes the trap number onto the stack and then calls **_cmntrap()**. That's the one writing the trap frame:

```
> _cmntrap::dis
_cmntrap:        pushal
_cmntrap+1:      subl   $0x10,%esp
_cmntrap+4:      movw   %ds,0xc(%esp)
_cmntrap+8:      movw   %es,0x8(%esp)
_cmntrap+0xc:    movw   %fs,0x4(%esp)
_cmntrap+0x10:   movw   %gs,0x0(%esp)
_cmntrap+0x14:   cmpw   $0x1b0,0x0(%esp)
_cmntrap+0x1b:   je     +0x16    <_cmntrap+0x31>
_cmntrap+0x1d:   movw   $0x160,%ax
_cmntrap+0x21:   movw   $0x0,%cx
_cmntrap+0x25:   movw   $0x1b0,%dx
_cmntrap+0x29:   movw   %eax,%ds
_cmntrap+0x2b:   movw   %eax,%es
_cmntrap+0x2d:   movw   %ecx,%fs
_cmntrap+0x2f:   movw   %edx,%gs
_cmntrap+0x31:   movl   %esp,%ebp
[ ... ]
_cmntrap+0x7e:   call   +0x176f7 <trap>
_cmntrap+0x83:   addl   $0xc,%esp
```

This code saves all eight general-purpose registers to the stack using **pushal**, then reserves four more words of stackspace and stuffs the not-yet-saved segment registers **%ds**..**%fs** into there (remember, **%cs** and, if necessary, **%ss** has already been saved by the CPU before dispatching the gate handler).

It doesn't write more to the stack, and a little later initializes its framepointer. This of course explains why the panic messages:

```
BAD TRAP: type=e (#pf Page fault) rp=d010dd88 addr=0 occurred in module “unix”
[ ... ]
d010dd88 unix:_cmntrap+83 ()
```

show the address of the saved registers, **rp=...**, and the framepointer of **_cmntrap()** to be equal.

Now the answer to “what was the stackpointer at the time of the trap” is easy. Dump the stack, starting at the trapframe, but this time simply as data. All is there:

- the segment registers **%ds**..**%fs** explicitly put there by **_cmntrap()**

- the general-purpose registers **%eax**..**%edi** written via **pushal** by **_cmntrap()**
- the trap number, 0xe, pushed by the gate entry, **pftrap()**
- **EFLAGS**/**%cs**/**%eip** and the pagefault error code written by the CPU
- values pushed before the trap – the last one indicating the stackpointer we want !

```
> d010dd88,40/nap
0xd010dd88:     0xd00401b0              segment registers %ds ...%fs
0xd010dd8c:     0xd0140000              explicitly written to the stack
0xd010dd90:     0x160                   by _cmntrap()
0xd010dd94:     0xd0130160
0xd010dd98:     0xd010de10
0xd010dd9c:     0xcfa379f0
0xd010dda0:     0xd010dde4
0xd010dda4:     0xd010ddb8              pushal
0xd010dda8:     0                       1st instruction _cmntrap()
0xd010ddac:     0xd0055600
0xd010ddb0:     0
0xd010ddb4:     0
0xd010ddb8:     0xe                     written by pftrap()
0xd010ddbc:     2
0xd010ddc0:     mutex_enter+0xd         saved to the stack by the CPU
0xd010ddc4:     0x158                   before dispatching trap handler
0xd010ddc8:     0x10246
0xd010ddcc:     lookuppnat+0x8f         last pre-trap word on the stack
0xd010ddd0:     0
0xd010ddd4:     0
0xd010ddd8:     0
0xd010dddc:     0xceb50d00
0xd010dde0:     0
0xd010dde4:     0xd010de68
0xd010dde8:     lookupnameat+0x54
[ ... ]
```

Comparing this with struct regs shows that the values there for **%ss**/**%usp** (and also for **sp=...** in the panic messages) are false positives. In fact, they are the very last two words written to the stack before the panic:

raw data on the stack	***saved registers – struct regs***

```
> d010dd88,40/nap                                > d010dd88::print -a "struct regs"
0xd010dd88:                                      {
0xd010dd88:     0xd00401b0      ↑                    d010dd88 r_gs = 0xd00401b0
0xd010dd8c:     0xd0140000      |                    d010dd8c r_fs = 0xd0140000
0xd010dd90:     0x160           |                    d010dd90 r_es = 0x160
0xd010dd94:     0xd0130160      |                    d010dd94 r_ds = 0xd0130160
0xd010dd98:     0xd010de10      |                    d010dd98 r_edi = 0xd010de10
0xd010dd9c:     0xcfa379f0      Trap Frame           d010dd9c r_esi = 0xcfa379f0
0xd010dda0:     0xd010dde4      |                    d010dda0 r_ebp = 0xd010dde4
0xd010dda4:     0xd010ddb8      |                    d010dda4 r_esp = 0xd010ddb8
0xd010dda8:     0               |                    d010dda8 r_ebx = 0
0xd010ddac:     0xd0055600      |                    d010ddac r_edx = 0xd0055600
0xd010ddb0:     0               |                    d010ddb0 r_ecx = 0
0xd010ddb4:     0               |                    d010ddb4 r_eax = 0
0xd010ddb8:     0xe             |                    d010ddb8 r_trapno = 0xe
0xd010ddbc:     2               |                    d010ddbc r_err = 0x2
0xd010ddc0:     mutex_enter+0xd |                    d010ddc0 r_eip = 0xfe82080d
0xd010ddc4:     0x158           |                    d010ddc4 r_cs = 0x158
0xd010ddc8:     0x10246         ↓                    d010ddc8 r_efl = 0x10246
0xd010ddcc:     lookuppnat+0x8f                      d010ddcc r_uesp = 0xfe8c08f4
```

raw data on the stack	***saved registers – `struct regs`***
`0xd010ddd0:     0`	`d010ddd0 r_ss = 0`
`[ ... ]`	`}`

The stackpointer before the panic therefore must have been **`0xd010ddcc`**. At the location we were interested in, **`4(%esp)`**, there is a **`NULL`** and this is consistent with what we see in the panic registers. It's therefore clear now that the code:

```
mutex_enter:      movl   %gs:0x10,%edx
mutex_enter+7:    movl   0x4(%esp),%ecx
mutex_enter+0xb:  xorl   %eax,%eax
mutex_enter+0xd:  lock cmpxchgl %edx,(%ecx)
```

had to panic when the atomic compare-and-exchange instruction tried to swap the contents of the mutex with **`curthread`**. Obviously, a **`NULL`** pointer was passed as mutex address.

7.3.3.How to get there – understanding the stacktrace

The next step in analysis of this crashdump is to examine where this **`NULL`** pointer came from. Since we wrote **`NULL`** into **`rootdir`**, we know the answer to that already of course, but what remains to be explained is how this action of ours explains the symptoms seen in the crashdump.

The task therefore is to *walk back* the call history and identify where the **`NULL`** that was passed as an argument to **`mutex_enter()`** was taken from, and ultimately whether what we find is consistent with the corruption of **`rootdir`** we created deliberately. The history of the panic lies in the call path that lead to it, and that in turn is contained in the *stacktrace*.

What is a stacktrace ? Strictly speaking, the word is incorrect and what should be used is *call trace* or *backtrace*. It contains the list of functions called by a thread up to a certain point in time – in the case of the panicing thread, of course, from the making the system call to the panic. As chapter 2 has shown, function calling in x86 via **`call`**/**`ret`** uses the stack to save return addresses. Because most CPU architectures do this, the words *call trace*, *backtrace* and *stacktrace* are used synonymously, and they describe using the saved return addresses from the stack to reconstruct the calling sequence that lead to, in this case, the panic.

A stacktrace was already printed as part of the panic messages:

```
      d010dce8 unix:die+a7 (e, d010dd88, 0, 0)
      d010dd74 unix:trap+fd1 (d010dd88, 0, 0)
      d010dd88 unix:_cmntrap+83 ()
      d010dde4 unix:mutex_enter+d (d010de10, 0, 1, 0, )
      d010de68 genunix:lookupnameat+54 (ceb50d00, 0, 1, 0, )
      d010df0c genunix:vn_openat+6c (ceb50d00, 0, 1, 6, )
      d010df70 genunix:copen+24f (ffd19553, ceb50d00,)
      d010df88 genunix:open+18 (ceb50d00, 0, 6, fec)
```

The builtin **`$C`** command of mdb allows us to reprint the panic stack:

```
> $C
d010dde4 mutex_enter+0xd(d010de10, 0, 1, 0, d010df00, 0)
d010de68 lookupnameat+0x54(ceb50d00, 0, 1, 0, d010df00, 0)
d010df0c vn_openat+0x6c(ceb50d00, 0, 1, 6, d010df6c, 0)
d010df70 copen+0x24f()
d010df88 open+0x18()
d010dfb4 sys_sysenter+0xe0()
```

The **`$C`** dcmd also accepts a framepointer as an argument, to start backtracing arbitrary (not just the panic thread's) stacks. If we give it the saved **`%ebp`** from the panic registers, it gives:

```
[ ... ]
BAD TRAP: type=e (#pf Page fault) rp=d010dd88 addr=0 occurred in module “unix”
[ ... ]
        edi: d010de10 esi: cfa379f0 ebp: d010dde4 esp: d010ddb8
[ ... ]
> d010dde4$C
d010de68 lookupnameat+0x54(ceb50d00, 0, 1, 0, d010df00, 0)
d010df0c vn_openat+0x6c(ceb50d00, 0, 1, 6, d010df6c, 0)
d010df70 copen+0x24f()
d010df88 open+0x18()
d010dfb4 sys_sysenter+0xe0()
```

These three stacktraces are identical in the bottom part but different for the top few frames that they show. Before we address the question why these are different and which one (if any) is right, let's first investigate two more basic things:

1. How does backtracing work at all ? How does the debugger do it, and what is the meaning of the addresses and function names/offsets reported ?
2. How can the debugger know what function arguments are – and, as it seems, how many arguments a function takes ? Also, are these to be trusted ?

Stack backtracing requires that function return addresses (i.e. code locations within a function where another function was called, or where execution is supposed to resume once the called function returns) are present in the stack. That's a given on the x86 architecture because of the way **call**/**ret** works – by pushing/popping return address to/from the stack. But the information “where in function X was function Y called” alone isn't sufficient. Once a return address has been located in the stack, a hint is required to find the next one – the debugger cannot, all by itself, know how many bytes of stackspace separate two consecutive return addresses. The stack is simply an array of *stackframes that vary in size*.

There are in principle two ways how this can be solved:

1. Make the packed array of variable-sized stackframes into a linked list, i.e. add a “*next*” pointer to every stackframe.
   In order to perform a backtrace then, the debugger only needs to locate the linkage field and follow that.
   This method will work with 32bit x86 stacks that use framepointers. More below.
2. Provide external information (debugging stabs) to the debugger about the frame size for a given function.
   To perform a backtrace in this case, the debugger needs to lookup the framesize information for a return address found, and add (stacks grow downward) that to the stack location of the return address in order to find the next return address.
   The advantage of this method of backtracing is that it can be made to work even if stackframes aren't linked, i.e. that do not use framepointers. The disadvantage clearly is that it requires external information that may or may not be available.

We'll defer 2. till later when talking about AMD64 backtracing. As mentioned, the presence of framepointers, i.e. the way the function prologue on 32bit x86 works:

```
lookupnameat:           pushl  %ebp          save caller's framepointer
lookupnameat+1:         movl   %esp,%ebp     our %ebp now points to caller's %ebp
[ ... ]
```

establishes a linkage between stackframes – every framepointer in itself points to the framepointer of the caller, and so on. When we dump the stack as raw data, we therefore find:

```
panic[cpu0]/thread=d0055600:
BAD TRAP: type=e (#pf Page fault) rp=d010dd88 addr=0 occurred in module "unix"
due to a NULL pointer dereference
[ ... ]
        edi: d010de10 esi: cfa379f0 ebp: d010dde4 esp: d010ddb8
[ ... ]
> d010dd88,100/nap
[ ... ]
0xd010dda0:     0xd010dde4          %ebp from trapframe
[ ... ]
0xd010ddc0:     mutex_enter+0xd
0xd010ddc4:     0x158
0xd010ddc8:     0x10246             end of trapframe
0xd010ddcc:     lookuppnat+0x8f     end of stack-before-trap
[ ... ]
0xd010dde4:     0xd010de68
0xd010dde8:     lookupnameat+0x54
[ ... ]
0xd010de68:     0xd010df0c
0xd010de6c:     vn_openat+0x6c
[ ... ]
0xd010df0c:     0xd010df70
0xd010df10:     copen+0x24f
[ ... ]
0xd010df70:     0xd010df88
0xd010df74:     open+0x18
[ ... ]
0xd010df88:     0xd010dfb4
0xd010df8c:     sys_sysenter+0xe0
[ ... ]
0xd010dfb4:     0x1c3
0xd010dfb8:     0
0xd010dfbc:     0x173
0xd010dfc0:     0x173
0xd010dfc4:     6
0xd010dfc8:     0xcebd2000
0xd010dfcc:     0x80477c0
0xd010dfd0:     0xd010dfe4
0xd010dfd4:     0xceb46000
0xd010dfd8:     0xceb1db75         Looks like another trapframe !
0xd010dfdc:     0x80477a8
0xd010dfe0:     5
0xd010dfe4:     0x208
0xd010dfe8:     0
0xd010dfec:     0xceb1db75
0xd010dff0:     0x16b
0xd010dff4:     0x282
0xd010dff8:     0x80477a8
0xd010dffc:     0x173
mdb: failed to read data from target: no mapping for address
0xd010e000:
```

This is, of course, exactly the work that mdb does automatically on **$C**. Compare:

Output of $C	***manually following framepointers***
`> $C`	`<ebp=X;<eip=p;<ebp/nXp`
`d010dde4 mutex_enter+0xd([ ... ])`	`d010dde4`
	`mutex_enter+0xd`
	`0xd010dde4:`
`d010de68 lookupnameat+0x54([ ... ])`	`d010de68 lookupnameat+0x54`
	`> *./nXp`
	`0xd010de68:`
`d010df0c vn_openat+0x6c([ ... ])`	`d010df0c vn_openat+0x6c`
	`> *./nXp`
	`0xd010df0c:`
`d010df70 copen+0x24f()`	`d010df70 copen+0x24f`
	`> *./nXp`
	`0xd010df70:`
`d010df88 open+0x18()`	`d010df88 open+0x18`
	`> *./nXp`
	`0xd010df88:`
`d010dfb4 sys_sysenter+0xe0()`	`d010dfb4 sys_sysenter+0xe0`
	`> *./nXp`
	`0xd010dfb4:`
	`1c3      0`

So the backtrace printed by mdb on **$C** simply consists of the framepointer/return address pairs in the stack.

We can also see that **$C**, when printing the panic thread's stacktrace, uses **%ebp**/**%eip** from the trapframe to print as the first line. That would be correct had the bad trap happened within a function that initialized a framepointer of its own – but **mutex_enter()**, simple as the code is, doesn't do that:

```
mutex_enter:            movl   %gs:0x10,%edx
mutex_enter+7:          movl   0x4(%esp),%ecx
mutex_enter+0xb:        xorl   %eax,%eax
mutex_enter+0xd:        lock cmpxchgl %edx,(%ecx)
mutex_enter+0x11:       jne    +0xbbb5  <mutex_vector_enter>
mutex_enter+0x17:       ret
mutex_enter+0x18:       movl   $0x0,%eax
```

This of course means that the framepointer, **%ebp**, at the time of the panic, still was that of the caller of **mutex_enter()**. Disassembly for the stackframe immediately below it also shows that the return address there isn't from a call to **mutex_enter()**:

```
d010dde4 mutex_enter+0xd(d010de10, 0, 1, 0, d010df00, 0)
d010de68 lookupnameat+0x54(ceb50d00, 0, 1, 0, d010df00, 0)
[ ... ]
> lookupnameat+0x54::dis
[ ... ]
lookupnameat+0x4f:      call   +0x79    <lookuppnat>
lookupnameat+0x54:      addl   $0x18,%esp
```

So how did we get from **lookuppnat()** to **mutex_enter()** ?

Simply follow the code. The bad trap happened in **mutex_enter+0xd**, without anything else having been pushed on the stack (or the stackpointer changed directly) by that function. This means – the stackpointer **%esp** at the time of the panic, in this case, points to the return address into the caller of **mutex_enter()**:

```
> d010dd88,40/nap
0xd010dd88:
0xd010dd88:     0xd00401b0
0xd010dd8c:     0xd0140000
0xd010dd90:     0x160
0xd010dd94:     0xd0130160
0xd010dd98:     0xd010de10
0xd010dd9c:     0xcfa379f0
0xd010dda0:     0xd010dde4
0xd010dda4:     0xd010ddb8
0xd010dda8:     0
0xd010ddac:     0xd0055600
0xd010ddb0:     0
0xd010ddb4:     0
0xd010ddb8:     0xe
0xd010ddbc:     2
0xd010ddc0:     mutex_enter+0xd
0xd010ddc4:     0x158
0xd010ddc8:     0x10246
0xd010ddcc:     lookuppnat+0x8f
0xd010ddd0:     0
[ ... ]
> lookuppnat+0x8f::dis
[ ... ]
lookuppnat+0x89:                pushl  %ebx
lookuppnat+0x8a:                call   -0xa00ef <mutex_enter>
lookuppnat+0x8f:                addl   $0x4,%esp
```

Trap Frame

stackpointer **%esp**
immediately before the panic

7.3.4. Finding function arguments

The next question to address is how the debugger finds arguments when displaying the backtrace, whether they're correct and in case they're not, how we can find the real function arguments manually.

It has already been shown that mdb's **$C** dcmd in 32bit x86 prints different numbers of arguments for different functions. Before we do this for the panic thread, let's first look at a more-complicated stacktrace that illustrates the underlaying principles better:

```
> cff19d00$C
cff19d1c swtch+0x119()
cff19d40 cv_wait_sig+0x119(cf830c16, cf83a964)
cff19d54 str_cv_wait+0x82(cf830c16, cf83a964, ffffffff, 0)
cff19d88 strwaitq+0x1a6(cf83a918, 2, 80, 3, ffffffff, cff19dd8)
cff19ddc strread+0x116(cf833c00, cff19f3c, ceefff38)
cff19df8 iwscnread+0x39()
cff19e0c cdev_read+0x22(380000, cff19f3c, ceefff38)
cff19e38 cnread+0x40()
cff19e4c cdev_read+0x22(0, cff19f3c, ceefff38)
cff19e90 spec_read+0x208()
cff19eac fop_read+0x1b(cfe2ec00, cff19f3c, 0, ceefff38, 0)
cff19f88 read+0x1f9()
cff19fb4 sys_sysenter+0xe0()
```

mdb in the backtrace above shows between two and six function arguments. From deduction, we already see that it cannot be fully correct about what it shows. For example, **read(9e)** functions, like all those ***read()** above, take three arguments as per the DDI interface specification, but the debugger shows between zero and four which cannot be right.

We know, from the way stacks on 32bit x96 work, that arguments are passed on the stack and functions access them via **+8(%ebp)** onwards. This is of course what mdb does above. Verify it:

```
[ ... ]
cff19d54 str_cv_wait+0x82(cf830c16, cf83a964, fffffff, 0)
[ ... ]
cff19d88 strwaitq+0x1a6(cf83a918, 2, 80, 3, ffffffff, cff19dd8)
[ ... ]
> cff19d54+8,4/nX
0xcff19d5c:
                cf830c16
                cf83a964
                fffffff
                0
> cff19d88+8,6/nX
0xcff19d90:
                cf83a918
                2
                80
                3
                ffffffff
                cff19dd8
```

But where does mdb get the idea from how many arguments it should print ? And why is it off in some cases ?

The answer to that lies in the return addresses, or rather in the instructions that they point to. We find:

return addresses/arguments	***Instr. at ret. addr.***
`> cff19d00$c`	
`swtch+0x119()`	`addl    $0x4,%esp`
`cv_wait_sig+0x119(cf830c16, cf83a964)`	`movzwl 0x18(%ebx),%eax`
`str_cv_wait+0x82(cf830c16, cf83a964, fffffff, 0)`	`addl    $0x8,%esp`
`strwaitq+0x1a6(cf83a918, 2, 80, 3, fffffff, cff19dd8)`	`addl    $0x10,%esp`
`strread+0x116(cf833c00, cff19f3c, ceefff38)`	`addl    $0x18,%esp`
`iwscnread+0x39()`	`addl    $0xc,%esp`
`cdev_read+0x22(380000, cff19f3c, ceefff38)`	`movl    %ebp,%esp`
`cnread+0x40()`	`addl    $0xc,%esp`
`cdev_read+0x22(0, cff19f3c, ceefff38)`	`movl    %ebp,%esp`
`spec_read+0x208()`	`addl    $0xc,%esp`
`fop_read+0x1b(cfe2ec00, cff19f3c, 0, ceefff38, 0)`	`movl    %ebp,%esp`
`read+0x1f9()`	`addl    $0x14,%esp`
`sys_sysenter+0xe0()`	`xorl    %ecx,%ecx`

We notice that:

1. mdb prints arguments for those functions where the return address to the caller points to an instruction **addl ...,%esp**.
   It does not print arguments for functions where the return address to the caller points to some other instruction.
   Example:
   - **read()** calls **fop_read()**, and the return address into **read+...** for that call points to **addl $0x14,%esp**. mdb prints arguments for fop_read().
   - **fop_read()** calls **spec_read()**, but now the return address to **fop_read+...** points to some other instruction. mdb prints no arguments for **spec_read()**.
2. The number of arguments displayed is deduced from the size of the 1st operand of the **addl ...,%esp**.

The debugger uses the fact that in many cases, compiler-generated code will *clean up the stack* after return from a function call. This means that code that calls a function with N arguments (and therefore pushes N 32bit words to the stack before **call**) will need to remove these N arguments from the stack again. In order to do that, the code pops them to nowhere by explicitly adding 4*N Bytes to the stackpointer. Like this:

```
strread+0xfd:     leal   -0x4(%ebp),%eax
strread+0x100:    pushl  %eax
strread+0x101:    pushl  $-0x1
strread+0x103:    movswl 0x14(%edi),%eax
strread+0x107:    pushl  %eax
strread+0x108:    pushl  0x20(%edi)
strread+0x10b:    movsbl -0x28(%ebp),%eax
strread+0x10f:    pushl  %eax
strread+0x110:    pushl  %ebx
strread+0x111:    call   +0xba3d  <strwaitq>
strread+0x116:    addl   $0x18,%esp
```

I.e. we've pushed six arguments for **strwaitq()**, and remove 6*4=0x18 Bytes from the stack after return from that function call.

As the example above has already shown, it's nothing more than code generation convention (and not mandated by the i386 UNIX ABI) that the return address shall point to an **addl** instruction that cleans up the stack. The compiler on optimizing code can decide to put something else in there and decide to do the stack cleanup later (or not at all, i.e. leave it for the function epilogue).

So, for cases where the debugger does not report arguments, how can we find how many arguments the function actually has ? There are two indirect ways of doing it:

1. Look at the assembly code of the caller and count the **PUSH** instructions that were done before the **call**. As an example, the following code for **fop_read()** proves it has passed five arguments to **spec_read()**:

    ```
    [ ... ]
    cff19e90 spec_read+0x208()
    cff19eac fop_read+0x1b(cfe2ec00, cff19f3c, 0, ceefff38, 0)
    [ ... ]
    > fop_read::dis
    fop_read:                       pushl  %ebp
    fop_read+1:                     movl   %esp,%ebp
    fop_read+3:                     movl   0x8(%ebp),%ecx
    fop_read+6:                     movl   0x28(%ecx),%eax
    fop_read+9:                     movl   0xc(%eax),%eax
    fop_read+0xc:                   pushl  0x18(%ebp)
    fop_read+0xf:                   pushl  0x14(%ebp)
    fop_read+0x12:                  pushl  0x10(%ebp)
    fop_read+0x15:                  pushl  0xc(%ebp)
    fop_read+0x18:                  pushl  %ecx
    fop_read+0x19:                  call   *%eax
    fop_read+0x1b:                  movl   %ebp,%esp
    fop_read+0x1d:                  popl   %ebp
    fop_read+0x1e:                  ret
    ```

2. Check the assembly code for the called function and see how many **+...(%ebp)** references it makes. Since we know the function accesses its arguments using the framepointer, **+8(%ebp)** onwards, this allows us to count how many arguments it actually uses. It's possible for a function to ignore some of its arguments, but of course it'd be bad if it used more args than were passed in. In that sense, checking for the numbers of arguments used is a complementary consistency check compared to checking the number of arguments passed in.

Applying this technique to **spec_read()** as above tells us it uses three arguments, **8(%ebp)**, **c(%ebp)**, and **0x14(%ebp)**. It ignores its third arg, **+0x10(%ebp)**, and any arg past the fourth.

```
> spec_read::dis ! egrep '[^-]0x[0-9a-f]+\(%ebp'
spec_read+0xd:                  movl   0x8(%ebp),%esi
spec_read+0x40:                 pushl  0x14(%ebp)
spec_read+0x43:                 pushl  0xc(%ebp)
spec_read+0x56:                 movl   0xc(%ebp),%edi
spec_read+0xf1:                 movl   0xc(%ebp),%esi
spec_read+0x1fb:                pushl  0x14(%ebp)
```

Of course, if we want to find the five arguments that were passed to **spec_read()** above, we can dump them from memory starting at **+8(%ebp)**:

```
[ ... ]
cff19e90 spec_read+0x208()
[ ... ]

> cff19e90+8,5/nX
0xcff19e98:
                cfe2ec00
                cff19f3c
                0
                ceefff38
                0
```

Quod erat demonstrandum !

Now let's revisit the panic backtrace:

```
> $C
d010dde4 mutex_enter+0xd(d010de10, 0, 1, 0, d010df00, 0)
d010de68 lookupnameat+0x54(ceb50d00, 0, 1, 0, d010df00, 0)
d010df0c vn_openat+0x6c(ceb50d00, 0, 1, 6, d010df6c, 0)
d010df70 copen+0x24f()
d010df88 open+0x18()
d010dfb4 sys_sysenter+0xe0()
```

This shows another oddity: mdb doesn't display more than six arguments even though some of the functions above take more than that. For example, **<sys/vnode.h>** declares the function prototype:

```
int     vn_openat(char *pnamep, enum uio_seg seg, int filemode, int createmode,
                struct vnode **vpp, enum create crwhy,
                mode_t umask, struct vnode *startvp);
```

which takes eight arguments. The return address:

```
> copen+0x24f/i
copen+0x24f:    addl   $0x20,%esp
```

confirms this (8*4 = 32 = 0x20). This is mdb's default setting – print no more than six arguments in stacktraces. The user can override this by giving the number of args to print (if possible) as additional option: **$C ...** – see below.

Summary:

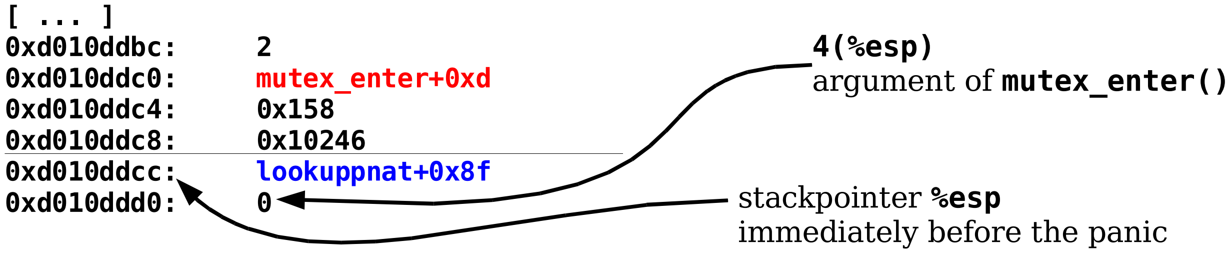
- The debugger determines the number of arguments to print from inspecting the instruction at the return address. If this adjusts the stackpointer via **addl**, then
  - the 1st operand is used as the number of stack words to print
  - mdb prints at most six arguments by default. That's a deliberate limitation – it can be overridden. Just specify the maximum number of arguments to print as additional parameter to **$C**:

```
> $C 10
d010dde4 mutex_enter+0xd(d010de10, 0, 1, 0, d010df00, 0)
d010de68 lookupnameat+0x54(ceb50d00, 0, 1, 0, d010df00, 0)
d010df0c vn_openat+0x6c(ceb50d00, 0, 1, 6, d010df6c, 0, 12, 0)
d010df70 copen+0x24f()
d010df88 open+0x18()
d010dfb4 sys_sysenter+0xe0()
```

- If the debugger prints arguments, they can be considered correct because they are on the stack and C code is supposed to treat its arguments read-only. Meaning the area on the stack where the arguments of a function are will not be modified by this function during its lifetime.
- If the debugger does not print arguments, it means the return address was found not to be an **`addl`** instruction. In this case:
  - disassemble the caller to see how many arguments it pushes to the stack
  - disassemble the called function to see how many arguments it accesses, i.e. how many **`+8(%ebp)`** onwards references it has in its assembly code.
  - use the *framepointer* (the address printed to the left of the return instruction in stacktraces from **`$C`** or **`::findstack`**) and dump N 32bit words of memory starting at **`+8(%ebp)`**.

7.3.5.Piecing it all together

Now let's determine the rootcause for this panic. We already know the BAD TRAP occurred because mutex_enter() was called with a NULL as argument. This is found in the stack:



```
[ ... ]
0xd010ddbc:     2
0xd010ddc0:     mutex_enter+0xd
0xd010ddc4:     0x158
0xd010ddc8:     0x10246
0xd010ddcc:     lookuppnat+0x8f
0xd010ddd0:     0
```

`4(%esp)`
argument of **`mutex_enter()`**

stackpointer **`%esp`**
immediately before the panic

and the code that calls **`mutex_enter()`**,

```
lookuppnat+0x89:                pushl   %ebx
lookuppnat+0x8a:                call    -0xa00ef <mutex_enter>
lookuppnat+0x8f:                addl    $0x4,%esp
```

together with the value of **`%ebx`** from the saved registers:

```
> d010dd88::print -a "struct regs"
{
[ ... ]
    d010dda8 r_ebx = 0
[ ... ]
> ::msgbuf
[ ... ]
        ebx:        0 edx: d0055600 ecx:        0 eax:        0
[ ... ]
```

proves we're looking at the “right” **`NULL`**. The task therefore is now to evaluate where that **`NULL`** pointer came from. We disassemble **`lookuppnat()`** for that purpose and check where it takes **`%ebx`** from:

```
> lookuppnat::dis ! grep %ebx
lookuppnat+6:           pushl %ebx
```

```
lookuppnat+0x25:        popl %ebx
lookuppnat+0x77:        movl 0x1c(%ebp),%ebx
lookuppnat+0x7a:        testl %ebx,%ebx
lookuppnat+0x7e:        movl 0x44c(%esi),%ebx
lookuppnat+0x86:        movl -0x4(%ebp),%ebx
lookuppnat+0x89:        pushl %ebx        arg of mutex_enter() comes from here !
lookuppnat+0x92:        incl 0xc(%ebx)
lookuppnat+0x95:        pushl %ebx
lookuppnat+0xcf:        pushl %ebx
lookuppnat+0xed:        popl %ebx
```

The **push** and **pop** instructions probably are parts of prologues/epilogues. The **testl** doesn't modify **%ebx**, so we're left with the following three that we need to check further:

```
[ ... ]
lookuppnat+0x77:        movl 0x1c(%ebp),%ebx
lookuppnat+0x7e:        movl 0x44c(%esi),%ebx
lookuppnat+0x86:        movl -0x4(%ebp),%ebx
```

The first one is an access to the 6th argument to **lookuppnat()**. We get that from the stack, using the framepointer that we know:

```
> d010dde4+1c/X
0xd010de00:     0
```

Hmm, so there's a **NULL** in there. Let's cheat and check sources, **<sys/pathname.h>**:

```
extern int lookuppnat(struct pathname *, struct pathname *, enum symfollow,
                vnode_t **, vnode_t **, vnode_t *);
```

That'd be a possibly valid place where a mutex address could come from, because the **vnode_t** data structure's first member, **v_lock** is a mutex. But then, the others can not yet be excluded. Let's check the assembly code again:

```
lookuppnat+0x77:        movl   0x1c(%ebp),%ebx
lookuppnat+0x7a:        testl  %ebx,%ebx
lookuppnat+0x7c:        jne    +0xd     <lookuppnat+0x89>
lookuppnat+0x7e:        movl   0x44c(%esi),%ebx
lookuppnat+0x84:        jmp    +0x5     <lookuppnat+0x89>
lookuppnat+0x86:        movl   -0x4(%ebp),%ebx
lookuppnat+0x89:        pushl  %ebx
lookuppnat+0x8a:        call   -0xa00ef <mutex_enter>
lookuppnat+0x8f:        addl   $0x4,%esp
```

Well – we load arg#6 into **%ebx**, check whether it's **NULL** – and only go on to call **mutex_enter()** on it if it is *not* – **jne**. We cannot have taken that first path.

The second occurance takes **%ebx** from **0x44c(%esi)**. Making the assumption that we'd gone down that path, we know that **%esi** wouldn't have changed between the above place and the time the panic occurred, and therefore we can use its value from the trapframe/saved registers to check what's in **0x44c(%esi)**:

```
> d010dd88::print -a "struct regs"
{
[ ... ]
    d010dd9c r_esi = 0xcfa379f0
[ ... ]
> ::msgbuf
[ ... ]
        edi: d010de10 esi: cfa379f0 ebp: d010dde4 esp: d010ddb8
[ ... ]
> 0xcfa379f0+44c/X
0xcfa37e3c:     cf679d80
> cf679d80::whatis
```

```
cf679d80 is cf679d80+0, allocated from vn_cache
```

This value is a vnode, and it's not **NULL** – so we cannot have gone down that codepath either. Leaves only the third possibility, **-4(%ebp)**. The **NULL** must have come from this local variable of **lookuppnat()**. The stack contents show this is **NULL**:

```
> d010dde4-4/X
0xd010dde0: 0
```

Now check the assembly code for places where **lookuppnat()** modifies this value:

```
> lookuppnat::dis ! egrep -e '-0x4\(%ebp\)$'
lookuppnat+0x3b:        movl   %eax,-0x4(%ebp)
lookuppnat+0x6c:        movl   %eax,-0x4(%ebp)
```

Disassemble around these areas to find out more:

```
> lookuppnat+0x3b::dis
lookuppnat+0x23:                popl   %edi
lookuppnat+0x24:                popl   %esi
lookuppnat+0x25:                popl   %ebx
lookuppnat+0x26:                movl   %ebp,%esp
lookuppnat+0x28:                popl   %ebp
lookuppnat+0x29:                ret
lookuppnat+0x2a:                pushl  0x8(%esi)
lookuppnat+0x2d:                call   -0xa0092 <mutex_enter>
lookuppnat+0x32:                addl   $0x4,%esp
lookuppnat+0x35:                movl   0x450(%esi),%eax
lookuppnat+0x3b:                movl   %eax,-0x4(%ebp)
lookuppnat+0x3e:                testl  %eax,%eax
lookuppnat+0x40:                je     +0x27    <lookuppnat+0x67>
[ ... ]
> lookuppnat+0x6c::dis
[ ... ]
lookuppnat+0x67:                movl   0xfed369d8,%eax
lookuppnat+0x6c:                movl   %eax,-0x4(%ebp)
lookuppnat+0x6f:                movl   0x4(%edi),%eax
lookuppnat+0x72:                cmpb   $0x2f,(%eax)
lookuppnat+0x75:                je     +0x11    <lookuppnat+0x86>
[ ... ]
```

Can we have gone down the first codepath ? Obviously not – if there is a **NULL** at **0x450(%esi)**, the **testl/je** would take us straight to the second codepath.

If mdb were a bit more friendly about printing symbol names instead of an address like **0xfed369d8** within the code, everything would be already clear. So we need an additional step:

```
> 0xfed369d8::whatis
fed369d8 is rootdir+0 in genunix's bss
> 0xfed369d8/X
rootdir:
rootdir:        0
> 0xfed369d8::whattype
fed369d8 is fed369d8+0, vnode_t *
```

And there we are – **rootdir**, the very variable that we **NULL**'ed out deliberately.

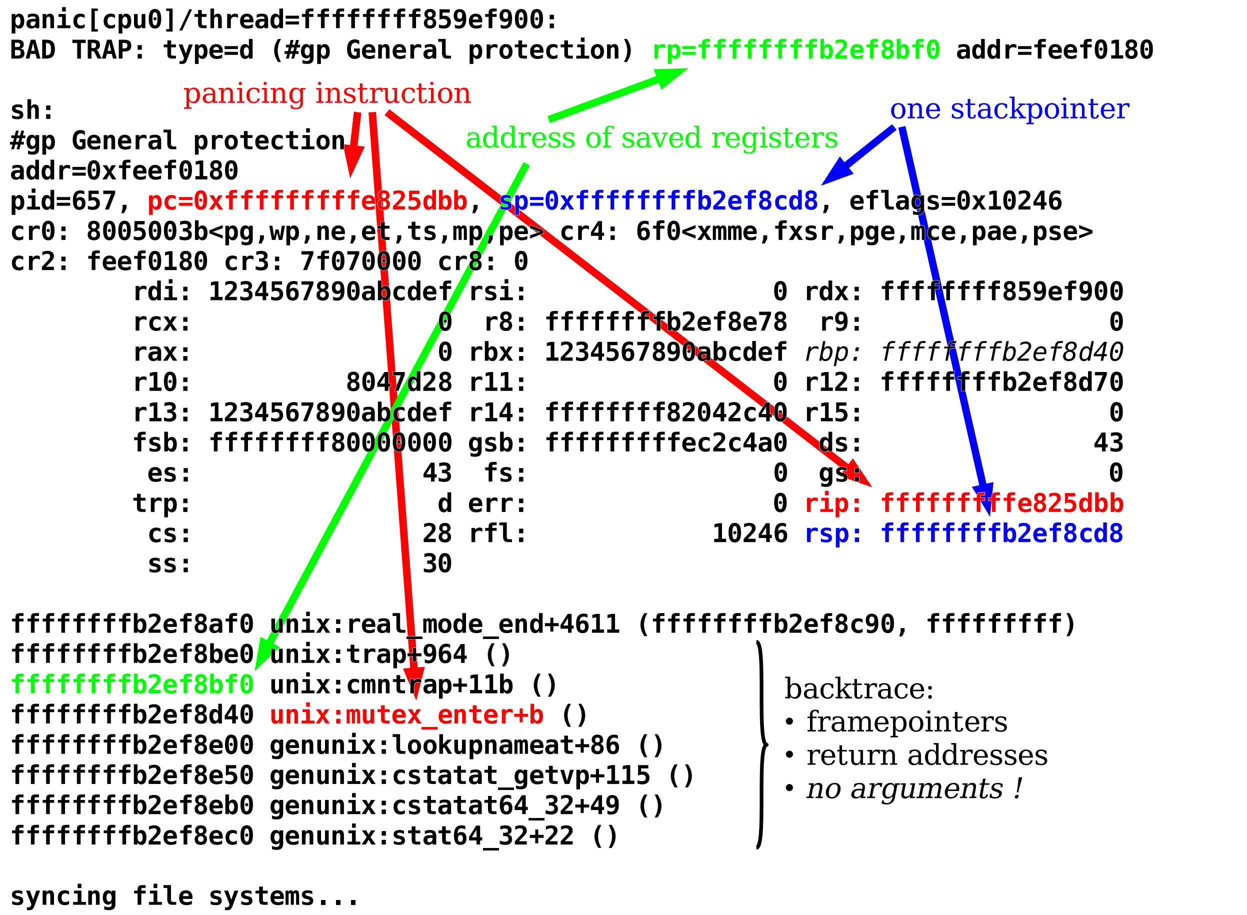
7.4. 64bit kernel crashdump analysis – well known example

As introduction to crashdump analysis on Solaris/x86 64bit, we again use deliberate corruption of **rootdir**. Since we've already seen a dump where we write **NULL** there, let's now try something different: **0x1234567890abcdef**. Run this to reproduce:

```
echo "rootdir/Z 0x1234567890abcdef" | mdb -kw
```

7.4.1. Panic messages

The machine panics similar to this:



```
panic[cpu0]/thread=ffffffff859ef900:
BAD TRAP: type=d (#gp General protection) rp=ffffffffb2ef8bf0 addr=feef0180

sh:
#gp General protection
addr=0xfeef0180
pid=657, pc=0xfffffffffe825dbb, sp=0xffffffffb2ef8cd8, eflags=0x10246
cr0: 8005003b<pg,wp,ne,et,ts,mp,pe> cr4: 6f0<xmme,fxsr,pge,mce,pae,pse>
cr2: feef0180 cr3: 7f070000 cr8: 0
    rdi: 1234567890abcdef rsi:                0 rdx: ffffffff859ef900
    rcx:                0  r8: ffffffffb2ef8e78  r9:                0
    rax:                0 rbx: 1234567890abcdef rbp: ffffffffb2ef8d40
    r10:          8047d28 r11:                0 r12: ffffffffb2ef8d70
    r13: 1234567890abcdef r14: ffffffff82042c40 r15:                0
    fsb: ffffffff80000000 gsb: fffffffffec2c4a0  ds:               43
     es:               43  fs:                0  gs:                0
    trp:                d err:                0 rip: fffffffffe825dbb
     cs:               28 rfl:            10246 rsp: ffffffffb2ef8cd8
     ss:               30

ffffffffb2ef8af0 unix:real_mode_end+4611 (ffffffffb2ef8c90, ffffffff)
ffffffffb2ef8be0 unix:trap+964 ()
ffffffffb2ef8bf0 unix:cmntrap+11b ()
ffffffffb2ef8d40 unix:mutex_enter+b ()
ffffffffb2ef8e00 genunix:lookupnameat+86 ()
ffffffffb2ef8e50 genunix:cstatat_getvp+115 ()
ffffffffb2ef8eb0 genunix:cstatat64_32+49 ()
ffffffffb2ef8ec0 genunix:stat64_32+22 ()

syncing file systems...
```

panicing instruction

address of saved registers

one stackpointer

backtrace:
- framepointers
- return addresses
- *no arguments !*

Illustration 2 - Solaris/x86 panic messages on a 64bit kernel

The panic message consists of the same components that we already know from 32bit x86 (or SPARC) crashdumps:

1. The panic string. It's a bad trap again, but a *general protection fault*, **#GP**:

   ```
   panic[cpu0]/thread=ffffffff859ef900:
   BAD TRAP: type=d (#gp General protection) rp=ffffffffb2ef8bf0 addr=feef0180
   ```

2. The register dump for general-purpose and control register contents. Again, this is only shown (and only meaningful) because the reason for the panic is a **BAD TRAP**.

   - As in 32bit mode, the register dump is a prettyprint version of the trapframe, and the address of that is in found both in the **rp=...** field and the framepointer of **cmntrap()**.

- As in 32bit mode, we get additional information about
  - the panicing process (the shell in this case),
  - a reprint of the trap type (#GP)
  - the address of the assembly instruction that caused the bad trap. It's printed three times – as **pc=...**, as value of the **%rip** register, and as *symbol name* in the backtrace, for the frame immediately below the trapframe.
- But there are several things that are different from 32bit mode:
  - The order in which the general-purpose registers are printed has changed. The register print starts with the *six argument registers*, the remaining general-purpose registers follow, including the new ones **%r10**...**%r15** that 64bit code can use.
  - **%rsp** is not printed among the general-purpose registers, but only at the end. It's identical to the value printed as **sp=...** - and ***it makes sense*** !
  - Registers **%fsb** and **%gsb** are not present in a 32bit x86 crashdump.
  - The segment registers contain different values than in 32bit mode. In fact, the values they have would be illegal in 32bit mode - **%fs**/**%gs** being **NULL**, and the two **%es**/**%ds** having userland (0x43 – low 2 bits set, ring 3 segment) values. See chapter 3 – this is the way the 64bit protected mode works. The CPU determines its operating mode (64bit long mode or 32bit compatibility mode) based on **%cs** alone and implicitly assumes flat address spaces.

3. Finally, we get a *backtrace*. Since the Solaris/x86 64bit kernel uses the top end of the address space, all kernel addresses there start with **0xffffff..........** One difference to 32bit mode is very noticeable – ***no arguments in the backtrace*** !

7.4.2.Immediate cause of the panic - stacktracing

The first thing we notice is that it didn't panic with a pagefault but with a **#GP** ("general protection") fault. This is, as mentioned in chapter 3, a peculiar side effect of the AMD64 virtual address space hole - addresses between **0x00007fffffffffff** and **0xffff800000000000** are "non-canonical" and create **#GP** faults on access - not pagefaults. It still means that the address is invalid.

The panic message print an **addr=...** field:

```
BAD TRAP: type=d (#gp General protection) rp=ffffffffb2ef8bf0 addr=feef0180
```

But this is nonsense for a **#GP** trap – because, as with the 32bit version of this crash, the kernel simply prints the contents of the pagefault address register **%cr2**, and that, as the name implicates, contains the address of the last pagefault. But here no pagefault happened – so **%cr2** is stale, and we better forget about what **addr=...** tells us. If we really want to know the faulting address for this, let's look at the panicing instruction:

```
[ ... ]
pid=657, pc=0xfffffffffe825dbb, sp=0xffffffffb2ef8cd8, eflags=0x10246
[ ... ]
        rdi: 1234567890abcdef rsi:                 0 rdx: ffffffff859ef900
[ ... ]
        trp:                d err:                 0 rip: fffffffffe825dbb
[ ... ]
> fffffffffe825dbb/i
mutex_enter+0xb:        lock cmpxchgq %rdx,(%rdi)
```

So the **#GP** fault occurred attempting to perform a 64bit atomic compare-and-swap of **%rdx** with the contents of the memory location **%rdi** points to. And that, as we see from the panic message, contains the value **0x1234567890abcdef** that was used to corrupt **rootdir**.

How did we get there ? Let's start with **mutex_enter()** again:

```
> mutex_enter::dis
mutex_enter:            movq   %gs:0x18,%rdx
mutex_enter+9:          xorl   %eax,%eax
mutex_enter+0xb:        lock cmpxchgq %rdx,(%rdi)
mutex_enter+0x10:       jne    +0xe1b0  <mutex_vector_enter>
mutex_enter+0x16:       ret
```

The first instruction looks surprising, given that the register dump from the panic message claimed **%gs** was **NULL**. But actually, this is just **curthread**, x86 64bit style. In 64bit mode, there's a register, **%gsb**, which directly holds the segment base address to be used when **%gs** references are made. Look at ***(%gsb+0x18)** and you'll find:

```
panic[cpu0]/thread=ffffffff859ef900:
[ ... ]
        fsb: ffffffff80000000 gsb: fffffffffec2c4a0  ds:                43
[ ... ]
> 0xfffffffffec2c4a0+0x18/J | ::whatis
> fffffffffec2c4a0+0x18/J | ::whatis
ffffffff859ef900 is ffffffff859ef900+0, allocated as a thread structure
> panic_thread/J
panic_thread:
panic_thread:   ffffffff859ef900
```

This is the same code we know from 32bit x86, though we no longer need to look into the descriptor tables to find out what **%gs** really points to. **%gsb** directly has the pointer for us

mutex_enter() directly exchanges curthread with its first argument, passed in **%rdi**.

Finding the call sequence that lead to the panic is simpler on x86 in 64bit than in 32bit mode – because the reported stackpointer is correct. **%rsp/sp=...** is the value of the *kernel stackpointer value before the panic* !

This can be crosschecked in two ways:

First, because the last thing that wrote a value to the stack was the **call** to **mutex_enter()**, we will find a return address at the location **%rsp** from the saved registers points to:

```
[ ... ]
pid=657, pc=0xfffffffffe825dbb, sp=0xfffffffb2ef8cd8, eflags=0x10246
[ ... ]
         cs:               28 rfl:            10246 rsp: fffffffb2ef8cd8
[ ... ]
> fffffffb2ef8cd8/nap
0xfffffffb2ef8cd8:
0xfffffffb2ef8cd8:              lookuppnat+0xa8
> lookuppnat+0xa8::dis
[ ... ]
lookuppnat+0xa0:                movq   %rbx,%rdi
lookuppnat+0xa3:                call   -0xe58e3 <mutex_enter>
lookuppnat+0xa8:                incl   0xc(%rbx)
[ ... ]
```

Second, when looking at the *64bit trapframe*, we find that **%rsp** is only present in the bottom hardware part, and points after the end of that – and therefore must have been written by the CPU before dispatching the trap handler:

raw data on the stack

```
> ffffffffb2ef8bf0,40/nap
0xffffffffb2ef8bf0:
0xffffffffb2ef8bf0: 0x1234567890abcdef
0xffffffffb2ef8bf8: 0
0xffffffffb2ef8c00: 0xfffffffff859ef900
0xffffffffb2ef8c08: 0
0xffffffffb2ef8c10: 0xffffffffb2ef8e78
0xffffffffb2ef8c18: 0
0xffffffffb2ef8c20: 0
0xffffffffb2ef8c28: 0x1234567890abcdef
0xffffffffb2ef8c30: 0xffffffffb2ef8d40
0xffffffffb2ef8c38: 0x8047d28
0xffffffffb2ef8c40: 0
0xffffffffb2ef8c48: 0xffffffffb2ef8d70
0xffffffffb2ef8c50: 0x1234567890abcdef
0xffffffffb2ef8c58: 0xffffffff82042c40
0xffffffffb2ef8c60: 0
0xffffffffb2ef8c68: 0xffffffff80000000
0xffffffffb2ef8c70: cpus
0xffffffffb2ef8c78: 0x43
0xffffffffb2ef8c80: 0x43
0xffffffffb2ef8c88: 0
0xffffffffb2ef8c90: 0
0xffffffffb2ef8c98: 0xd
0xffffffffb2ef8ca0: 0
0xffffffffb2ef8ca8: mutex_enter+0xb
0xffffffffb2ef8cb0: 0x28
0xffffffffb2ef8cb8: 0x10246
0xffffffffb2ef8cc0: 0xffffffffb2ef8cd8
0xffffffffb2ef8cc8: 0x30
0xffffffffb2ef8cd0: 0xffffffffb2ef8cf0
0xffffffffb2ef8cd8: lookuppnat+0xa8
```

saved registers – struct regs

```
> ffffffffb2ef8bf0::print -a "struct regs"
{
    ffffffffb2ef8bf0 r_rdi = 0x1234567890abcdef
    ffffffffb2ef8bf8 r_rsi = 0
    ffffffffb2ef8c00 r_rdx = 0xfffffffff859ef900
    ffffffffb2ef8c08 r_rcx = 0
    ffffffffb2ef8c10 r_r8 = 0xffffffffb2ef8e78
    ffffffffb2ef8c18 r_r9 = 0
    ffffffffb2ef8c20 r_rax = 0
    ffffffffb2ef8c28 r_rbx = 0x1234567890abcdef
    ffffffffb2ef8c30 r_rbp = 0xffffffffb2ef8d40
    ffffffffb2ef8c38 r_r10 = 0x8047d28
    ffffffffb2ef8c40 r_r11 = 0
    ffffffffb2ef8c48 r_r12 = 0xffffffffb2ef8d70
    ffffffffb2ef8c50 r_r13 = 0x1234567890abcdef
    ffffffffb2ef8c58 r_r14 = 0xffffffff82042c40
    ffffffffb2ef8c60 r_r15 = 0
    ffffffffb2ef8c68 r_fsbase = 0xffffffff80000000
    ffffffffb2ef8c70 r_gsbase = 0xfffffffffec2c4a0
    ffffffffb2ef8c78 r_ds = 0x43
    ffffffffb2ef8c80 r_es = 0x43
    ffffffffb2ef8c88 r_fs = 0
    ffffffffb2ef8c90 r_gs = 0
    ffffffffb2ef8c98 r_trapno = 0xd
    ffffffffb2ef8ca0 r_err = 0
    ffffffffb2ef8ca8 r_rip = 0xfffffffffe825dbb
    ffffffffb2ef8cb0 r_cs = 0x28
    ffffffffb2ef8cb8 r_rfl = 0x10246
    ffffffffb2ef8cc0 r_rsp = 0xffffffffb2ef8cd8
    ffffffffb2ef8cc8 r_ss = 0x30
}
```

The only thing in this stack that seems odd on first glance is the value between the end of the trapframe and the saved **%rsp**:

```
0xffffffffb2ef8cc8: 0x30
0xffffffffb2ef8cd0: 0xffffffffb2ef8cf0 ◀───────
0xffffffffb2ef8cd8: lookuppnat+0xa8
```

But this is, see chapter 3, normal behaviour for 64bit gates – they align the stack at a multiple of 16 Bytes. So this in-between value has no meaning – it's padding and just old, pre-trap stale information still in this memory location.

So the big 32bit x86 problem “how do I find the kernel stackpointer at the time of the panic” is trivial on 64bit x86: We can trust **%rsp** from the panic register dump.

Stack backtracing in the Solaris/x86 64bit *kernel* works the same way as before:

Output of $C

```
> $C
ffffffffb2ef8d40 mutex_enter+0xb

ffffffffb2ef8e00 lookupnameat+0x86

ffffffffb2ef8e50 cstatat_getvp+0x115

ffffffffb2ef8eb0 cstatat64_32+0x49

ffffffffb2ef8ec0 stat64_32+0x22
```

manually following framepointers

```
<rbp=J;<rip=p;<rbp/nJp
          ffffffffb2ef8d40
mutex_enter+0xb
ffffffffb2ef8d40:
          ffffffffb2ef8e00 lookupnameat+0x86
> *./nJp
ffffffffb2ef8e00:
          ffffffffb2ef8e50 cstatat_getvp+0x115
> *./nJp
ffffffffb2ef8e50:
          ffffffffb2ef8eb0 cstatat64_32+0x49
> *./nJp
ffffffffb2ef8eb0:
          ffffffffb2ef8ec0 stat64_32+0x22
```

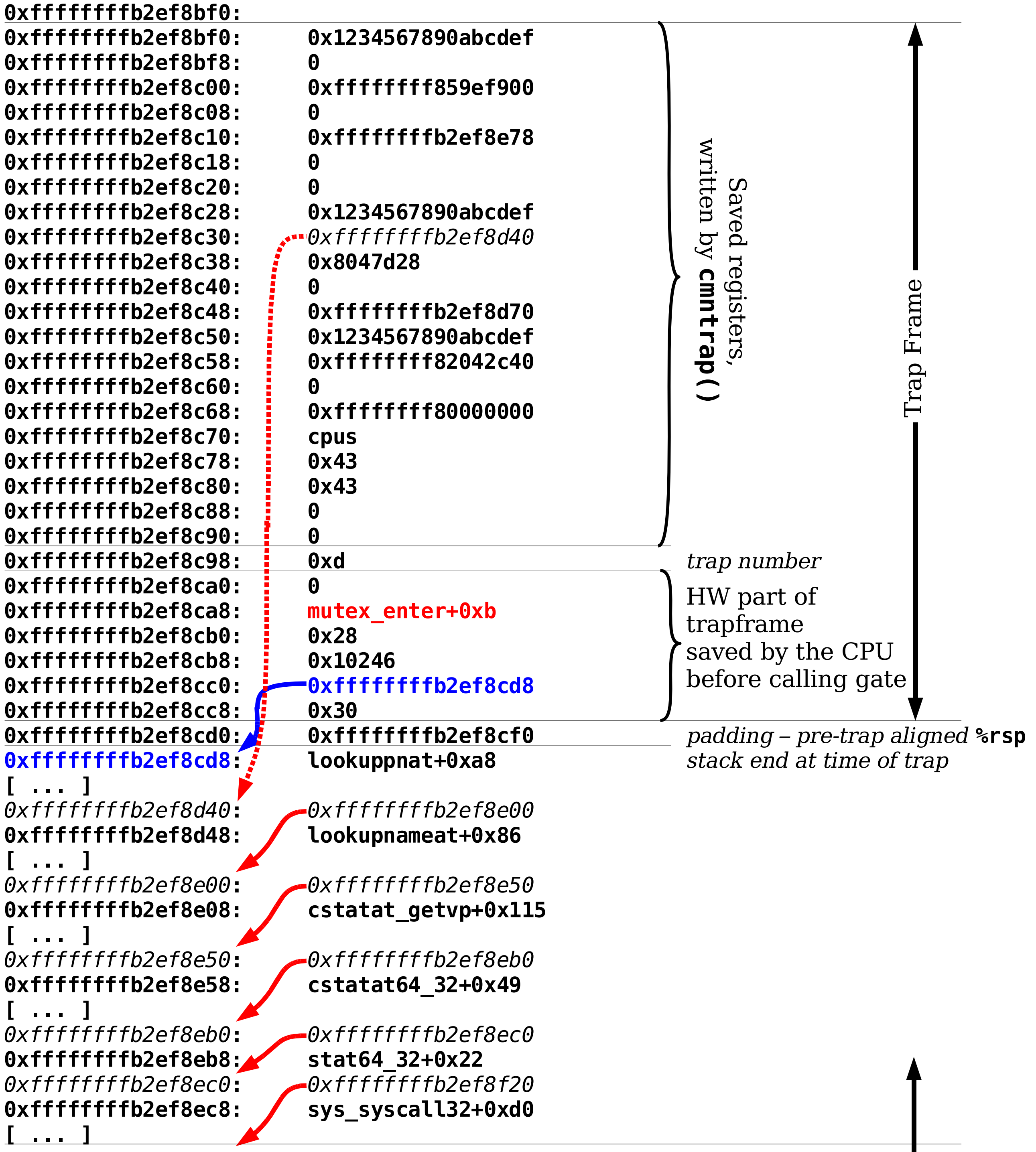
Output of **`$C`**	***manually following framepointers***

`ffffffffb2ef8f20 sys_syscall32+0xd0`	`> *./nJp`
	`ffffffffb2ef8ec0:`
	`ffffffffb2ef8f20 sys_syscall32+0xd0`

`0000000008076c90 1`	`> *./nJp`
	`0xffffffffb2ef8f20:`
	`8076c90 1`

When looking at the raw stack starting with the trap frame, we find the same linkage:

```
[ ... ]
BAD TRAP: type=d (#gp General protection) rp=ffffffffb2ef8bf0 addr=feef0180
[ ... ]
> ffffffffb2ef8bf0,100/nap
0xffffffffb2ef8bf0:
```



```
0xffffffffb2ef8bf0:     0x1234567890abcdef
0xffffffffb2ef8bf8:     0
0xffffffffb2ef8c00:     0xffffffff859ef900
0xffffffffb2ef8c08:     0
0xffffffffb2ef8c10:     0xffffffffb2ef8e78
0xffffffffb2ef8c18:     0
0xffffffffb2ef8c20:     0
0xffffffffb2ef8c28:     0x1234567890abcdef
0xffffffffb2ef8c30:     0xffffffffb2ef8d40
0xffffffffb2ef8c38:     0x8047d28
0xffffffffb2ef8c40:     0
0xffffffffb2ef8c48:     0xffffffffb2ef8d70
0xffffffffb2ef8c50:     0x1234567890abcdef
0xffffffffb2ef8c58:     0xffffffff82042c40
0xffffffffb2ef8c60:     0
0xffffffffb2ef8c68:     0xffffffff80000000
0xffffffffb2ef8c70:     cpus
0xffffffffb2ef8c78:     0x43
0xffffffffb2ef8c80:     0x43
0xffffffffb2ef8c88:     0
0xffffffffb2ef8c90:     0
0xffffffffb2ef8c98:     0xd
0xffffffffb2ef8ca0:     0
0xffffffffb2ef8ca8:     mutex_enter+0xb
0xffffffffb2ef8cb0:     0x28
0xffffffffb2ef8cb8:     0x10246
0xffffffffb2ef8cc0:     0xffffffffb2ef8cd8
0xffffffffb2ef8cc8:     0x30
0xffffffffb2ef8cd0:     0xffffffffb2ef8cf0
0xffffffffb2ef8cd8:     lookuppnat+0xa8
[ ... ]
0xffffffffb2ef8d40:     0xffffffffb2ef8e00
0xffffffffb2ef8d48:     lookupnameat+0x86
[ ... ]
0xffffffffb2ef8e00:     0xffffffffb2ef8e50
0xffffffffb2ef8e08:     cstatat_getvp+0x115
[ ... ]
0xffffffffb2ef8e50:     0xffffffffb2ef8eb0
0xffffffffb2ef8e58:     cstatat64_32+0x49
[ ... ]
0xffffffffb2ef8eb0:     0xffffffffb2ef8ec0
0xffffffffb2ef8eb8:     stat64_32+0x22
0xffffffffb2ef8ec0:     0xffffffffb2ef8f20
0xffffffffb2ef8ec8:     sys_syscall32+0xd0
[ ... ]
```

Saved registers, written by **`cmntrap()`**

trap number

HW part of trapframe saved by the CPU before calling gate

padding – pre-trap aligned **`%rsp`**

stack end at time of trap

Trap Frame

```
0xfffffffffb2ef8f20:     0x8076c90
0xfffffffffb2ef8f28:     1
0xfffffffffb2ef8f30:     0xdf
0xfffffffffb2ef8f38:     0xfef9f5a7
0xfffffffffb2ef8f40:     0x807330c
0xfffffffffb2ef8f48:     0
0xfffffffffb2ef8f50:     0xd7
0xfffffffffb2ef8f58:     1
0xfffffffffb2ef8f60:     0x8047dd4
0xfffffffffb2ef8f68:     0x8047e7c
0xfffffffffb2ef8f70:     0x296
0xfffffffffb2ef8f78:     0x42138701
0xfffffffffb2ef8f80:     0
0xfffffffffb2ef8f88:     0xffffffff857d6cc0
0xfffffffffb2ef8f90:     0xffffffff859ef900
0xfffffffffb2ef8f98:     0xffffffff80000000
0xfffffffffb2ef8fa0:     0xfeef2000
0xfffffffffb2ef8fa8:     0x43
0xfffffffffb2ef8fb0:     0x43
0xfffffffffb2ef8fb8:     0
0xfffffffffb2ef8fc0:     0x1c3
0xfffffffffb2ef8fc8:     0xe
0xfffffffffb2ef8fd0:     7
0xfffffffffb2ef8fd8:     0xfef9f5a7
0xfffffffffb2ef8fe0:     0x3b
0xfffffffffb2ef8fe8:     0x202
0xfffffffffb2ef8ff0:     0x8047d28
0xfffffffffb2ef8ff8:     0x43
mdb: failed to read data from target: no mapping for address
0xfffffffffb2ef9000:
```

System Call Frame

7.4.3.Finding function arguments

It has been shown that in 64bit mode, mdb prints no arguments in backtraces. This of course is due to the fact that the x86 64bit ABI uses registers **%rdi**, **%rsi**, **%rdx**, **%rcx**, **%r8** and **%r9** (in that order) to pass the first six (integer) arguments and only resorts to using the stack if there are more. In a call trace the argument registers are overwritten each time another function is called, and so we only can reconstruct their values at the time of a **BAD TRAP**, using the contents of the trapframe.

Which is why mdb in 64bit x86 doesn't give arguments in stacktraces. All traces look like this:

```
> $C
fffffffb2ef8d40 mutex_enter+0xb()
fffffffb2ef8e00 lookupnameat+0x86()
fffffffb2ef8e50 cstatat_getvp+0x115()
fffffffb2ef8eb0 cstatat64_32+0x49()
fffffffb2ef8ec0 stat64_32+0x22()
fffffffb2ef8f20 sys_syscall32+0xd0()
0000000008076c90 1()
> *panic_thread::findstack
stack pointer for thread ffffffff859ef900: fffffffb2ef8c30
  fffffffb2ef8d40 0x8047d28()
  fffffffb2ef8e00 lookupnameat+0x86()
  fffffffb2ef8e50 cstatat_getvp+0x115()
  fffffffb2ef8eb0 cstatat64_32+0x49()
  fffffffb2ef8ec0 stat64_32+0x22()
  fffffffb2ef8f20 sys_syscall32+0xd0()
  0000000008076c90 1()
```

i.e. no arguments.

How can we overcome this problem ?

Simplest solution at this time is to use **act.so** from the CTEact package. This, as part of the *Automated Crash Tool* utility package is a loadable mdb module, which provides additional dcmds for mdb. Among these is an enhanced stack tracer, in the mdb dcmd **::act_thread**:

```
> ::load /opt/CTEact/mdb/5.10/amd64/act.so
> ::dcmds ! grep ^act
act                      - Automatic Coredump Tool [-zRu] [-s <dir>]
act_check_bio            - check for threads blocked in biowait
act_check_getblk         - check for threads blocked in getblk
act_check_mutex          - check for threads blocked on mutex's
act_check_rwlock         - check for threads blocked on rwlock's
act_dis_cpus             - print dispatcher queues
act_module_name          - print name of module
act_moutput              - print arguments
act_msgbuf               - print console messages
act_procs                - print number of active processes
act_sunsolve             - print sunsolve string
act_system               - print contenets of /etc/system
act_thread               - print thread [-PCfFnt]
act_thread_summary       - count number of threads
act_utsname              - act do utsname
> *panic_thread::act_thread

***
process id 657 is -sh, parent process is 270
```

```
uid is 0, gid is 0

thread addr ffffffff859ef900, proc addr ffffffff82042c40,
lwp addr ffffffff857d6cc0
Thread bound to cpu id 0

t_state is 0x4 - TS_ONPROC

Scheduling info:
t_pri is 0x3b, t_epri is 0, t_cid is 0x1
scheduling class is: TS
t_disp_time: is 0x25a3eb, 0t2466795
last switched: 4 secs ago,  1 sec before panic on cpu 0
t_stk ffffffffb2ef8f20

stack trace is:

unix: panicsys+0x6e (0xffffffffb2ef89a0,vpanic+0x179,panic_stack+0x1f20,
                     0xfec2c4a0)
unix: vpanic+0x179 ()
unix: fffffffffe835a22 ()
unix: panic (0xfffffffffe87b0d8)
unix: fffffffffe81b607 ()
unix: die (?,?,0xfeef0180,0)
unix: trap+0x964 (0xffffffffb2ef8bf0,0xfeef0180,0)

TRAP FRAME: T_GPFLT general protection fault
        rdi             0      rsio           0
        rdx    0x859ef900      rcx            0
        r8     0xb2ef8e78      r9             0
        rax             0      rbx   0x90abcdef
        rbp    0xb2ef8d40      r10    0x8047d28
        r11             0      r12   0xb2ef8d70
        r13    0x90abcdef      r14   0x82042c40
        r15             0      fsb   0x80000000
        gsb    0xfec2c4a0      ds          0x43
        es           0x43      fs             0
        gs              0      trp          0xd
        err             0      rip   0xfe825dbb
        cs           0x28      rfl      0x10246
        rsp    0xb2ef8cd8      ss          0x30
unix: mutex_enterunix: mutex_enter+0xb ()
genunix: lookuppnat (?,?,1,0,0xffffffffb2ef8e78,?)
genunix: lookupnameat+0x86 (0x8076c90,0x1234567890abcdef,1,0,0xffffffffb2ef8e78,
                            0)
genunix: cstatat_getvp+0x115 (0xffd19553,0x8076c90,1,1,0xffffffffb2ef8e78,
                              0xffffffffb2ef8e80)
genunix: cstatat64_32+0x49 (0xffd19553,0x8076c90,1,0x8047d40,?,0x10)
genunix: stat64_32+0x22 (0xb2ef8f20,0xfe801120)
unix: sys_syscall32+0xd0 ()
```

act isn't perfect and still getting further enhancements for x86/64bit. We can see, for example, that the version of act used here doesn't print the contents of the trap frame correctly – the upper 32bits of the 64bit registers are missing. But there are arguments in the stacktraces – precisely what we want to see !

::act_thread can be used instead of mdb's builtin **::findstack** in order to get stacktraces with arguments for a given kernel thread address. It does not, at this time, replace **$C** yet. This is another reason for us – apart from curiosity of course – why we should find out how arguments can be reconstructed from stacktraces.

Passing arguments in registers, especially if these are global like on AMD64, means that over the lifetime of a function, especially if that in itself is making more function calls, the original contents of the argument registers will be overwritten – with variables from the working set, or with new arguments for functions to be called. *Register reuse* is the *common case* on AMD64. Unlike e.g. SPARC, where arguments are also passed in registers but the windowing mechanism provides every function with an *independent set* of argument registers, AMD64 uses the *same set* of argument registers all the time.

Which shows the way out of the problem. Before trying on the crashdump, let's investigate the following C sourcecode and the binary created from it:

```
extern int somefunc(int a1, int a2, int a3, int a4, int a5, int a6);
extern void otherfunc(int b1, int b2, int b3, int b4, int b5, int b6, int b7);

int call_two_funcs(int a, int b, int c, int d, int e, int f)
{
      int local;

      local = somefunc(1, 2, 3, 4, 5, 6);
      otherfunc(a, b, c, d, e, f, local);
      return (local);
}
```

This creates a situation that is common for C sourcecode – arguments of a function are needed during the entire lifetime of that function, no matter how many other functions were called in-between. Given the ABI, the consequences for the generated code are clear: A function that needs its arguments even after making another function call by itself will have to store them in a save place – and that's its own stackframe. AMD64 code therefore will create *hidden local*s in its stackframe to stuff the arguments in, similar to 32bit x86 code that puts the nonvolatile registers to the stack. The binary code created from the above example shows this – red instructions move arguments into/outof the stack resp. the nonvolatile registers:

```
call_two_funcs:      pushq  %rbp
call_two_funcs+0x1:  movq   %rsp,%rbp
call_two_funcs+0x4:  subq   $0x40,%rsp
call_two_funcs+0x8:  movq   %r12,-0x20(%rbp)
call_two_funcs+0xc:  movq   %r13,-0x18(%rbp)
call_two_funcs+0x10: movl   %r9d,%r12d
call_two_funcs+0x13: movq   %r14,-0x10(%rbp)
call_two_funcs+0x17: movq   %r15,-0x8(%rbp)
call_two_funcs+0x1b: movl   %ecx,%r14d
call_two_funcs+0x1e: movl   %edi,-0x2c(%rbp)
call_two_funcs+0x21: movl   %esi,-0x30(%rbp)
call_two_funcs+0x24: movl   %edx,%r15d
call_two_funcs+0x27: movl   %r8d,%r13d
call_two_funcs+0x2a: movl   $0x6,%r9d
call_two_funcs+0x30: movl   $0x5,%r8d
call_two_funcs+0x36: movl   $0x4,%ecx
call_two_funcs+0x3b: movl   $0x3,%edx
call_two_funcs+0x40: movl   $0x2,%esi
call_two_funcs+0x45: movl   $0x1,%edi
call_two_funcs+0x4a: movq   %rbx,-0x28(%rbp)
call_two_funcs+0x4e: call   +0x5        <call_two_funcs+0x53>
call_two_funcs+0x53: movl   -0x30(%rbp),%esi
call_two_funcs+0x56: movl   -0x2c(%rbp),%edi
call_two_funcs+0x59: movl   %eax,%ebx
call_two_funcs+0x5b: movl   %r12d,%r9d
call_two_funcs+0x5e: movl   %r13d,%r8d
```

```
call_two_funcs+0x61: movl   %r14d,%ecx
call_two_funcs+0x64: movl   %r15d,%edx
call_two_funcs+0x67: movl   %eax,(%rsp)
call_two_funcs+0x6a: call   +0x5        <call_two_funcs+0x6f>
call_two_funcs+0x6f: movl   %ebx,%eax
call_two_funcs+0x71: movq   -0x20(%rbp),%r12
call_two_funcs+0x75: movq   -0x28(%rbp),%rbx
call_two_funcs+0x79: movq   -0x18(%rbp),%r13
call_two_funcs+0x7d: movq   -0x10(%rbp),%r14
call_two_funcs+0x81: movq   -0x8(%rbp),%r15
call_two_funcs+0x85: leave
call_two_funcs+0x86: ret
```

This behaviour of compiler-generated code of course means that we'll often be able to find arguments for functions in 64bit x86 backtraces if we use the following strategy:

1. Disassemble the function you want to find the arguments for, and determine whether it moves the argument registers into the stack, i.e. into locals **-...(%rbp)**. If it does, you've found them.

2. If not, see whether it moves argument registers into nonvolatile registers, i.e. from registers **%rdi/%rsi/%rdx/%rcx/%r8/%r9** to **%rbx/%r12/%r13/%r14/%r15**.
   If it does, disassemble the *next function* in the call trace. This will probably, as part of its function prologue, save (some or all of) the nonvolatile registers onto the stack. Do that recursively until all nonvolatile regs are on the stack – or until you find a trapframe and get the remaining register contents from there.

3. If the function you started out with did neither save its argument registers to the stack nor put them into nonvolatile registers, go back to the *previous function* (i.e. the caller) in the call trace and determine where the arguments to your function of interest were taken from.

This process, as indicated by the ACT output, can be automated to a certain extent. There's no guarantee that this succeeds, of course – ACT, if it cannot determine where an argument went, will print **?** instead:

```
[ ... ]
genunix: lookuppnat (?,?,1,0,0xfffffffb2ef8e78,?)
[ ... ]
```

In the case shown, act failed to find the first two and the sixth arguments to **lookuppnat()**. Let's try the procedure manually and see whether we can do better:

```
> lookuppnat::dis
lookuppnat:                     pushq  %rbp
lookuppnat+1:                   movq   %rsp,%rbp
lookuppnat+4:                   subq   $0x60,%rsp
lookuppnat+8:                   movq   %r12,-0x20(%rbp)
lookuppnat+0xc:                 movq   %r15,-0x8(%rbp)
lookuppnat+0x10:                movq   %rdi,%r12
lookuppnat+0x13:                movq   %rbx,-0x28(%rbp)
lookuppnat+0x17:                movq   %r13,-0x18(%rbp)
lookuppnat+0x1b:                movq   %r9,%r15
lookuppnat+0x1e:                movq   %r14,-0x10(%rbp)
lookuppnat+0x22:                movq   %rsi,-0x30(%rbp)
lookuppnat+0x26:                movl   %edx,-0x34(%rbp)
lookuppnat+0x29:                movq   %rcx,-0x40(%rbp)
lookuppnat+0x2d:                movq   %r8,-0x48(%rbp)
lookuppnat+0x31:                movq   %gs:0x18,%rax
lookuppnat+0x3a:                cmpq   $0x0,0x10(%rdi)
lookuppnat+0x3f:                movq   0x168(%rax),%r14
lookuppnat+0x46:                movl   $0x2,%eax
```

```
lookuppnat+0x4b:                je     +0xae    <lookuppnat+0xf9>
lookuppnat+0x51:                movq   0x10(%r14),%rdi
lookuppnat+0x55:                call   -0xe5895 <mutex_enter>
```

Yes – we obviously can. All six arguments are accounted for:

- **`%rdi`** (1st arg) is put into nonvolatile register **`%r12`**. Crosschecking that this is never again overwritten except in the function epilogue:

  ```
  [ ... ]
  lookuppnat+8:                  movq   %r12,-0x20(%rbp)
  [ ... ]
  > lookuppnat::dis ! grep '%r12$'
  lookuppnat+0x10:               movq   %rdi,%r12
  lookuppnat+0xfd:               movq   -0x20(%rbp),%r12
  ```

  Since the next thing after **`lookuppnat()`** was the panic in **`mutex_enter()`**, we know that the value of **`%r12`** in the panic registers still has the value we're looking for:

  ```
  > ::msgbuf
  [ ... ]
          r10:          8047d28 r11:                0 r12: ffffffffb2ef8d70
  [ ... ]
  ```

- **`%rsi`** (2nd arg) is put into **`-0x30(%rbp)`**. Again, using the framepointer for **`lookuppnat()`** that's in the panic registers, we can locate the argument:

  ```
  [ ... ]
          rax:                0 rbx: 1234567890abcdef rbp: ffffffffb2ef8d40
  [ ... ]
  > ffffffffb2ef8d40-30/J
  0xffffffffb2ef8d10:            0
  ```

- **`%edx`** (3rd arg, obviously a 32bit **`int`**) is put into **`-0x34(%rbp)`**:

  ```
  > ffffffffb2ef8d40-34/X
  0xffffffffb2ef8d0c:            1
  ```

- **`%rcx`** (4th arg) is put into -0x40(%rbp):

  ```
  > ffffffffb2ef8d40-40/J
  0xffffffffb2ef8d00:            0
  ```

- **`%r8`** (5th arg) goes to **`-0x48(%rbp)`**:

  ```
  > ffffffffb2ef8d40-40/J
  0xffffffffb2ef8cf8:            ffffffffb2ef8e78
  ```

- **`%r9`** (6th arg) is put into nonvolatile register **`%r15`**. Again, we need to check that the function never reinitializes **`%r15`** with something else before we can claim that the value of **`%r15`** from the trap frame still has it:

  ```
  [ ... ]
  lookuppnat+0xc:                movq   %r15,-0x8(%rbp)
  [ ... ]
  > lookuppnat::dis ! grep '%r15$'
  lookuppnat+0x1b:               movq   %r9,%r15
  lookuppnat+0x94:               testq  %r15,%r15
  lookuppnat+0x109:              movq   -0x8(%rbp),%r15
  > ::msgbuf
  [ ... ]
          r13: 1234567890abcdef r14: ffffffff82042c40 r15:                0
  [ ... ]
  ```

This strategy is generic and can be applied to all stacktraces on AMD64.

7.4.4. Piecing it all together

Now that we know how backtracing in the Solaris/x86 64bit kernel is done and how arguments to functions can be retrieved in spite of the use of registers for argument passing, we're ready to put the pieces together and determine how overwriting **rootdir** with **0x1234567890abcdef** could cause the panic we've got.

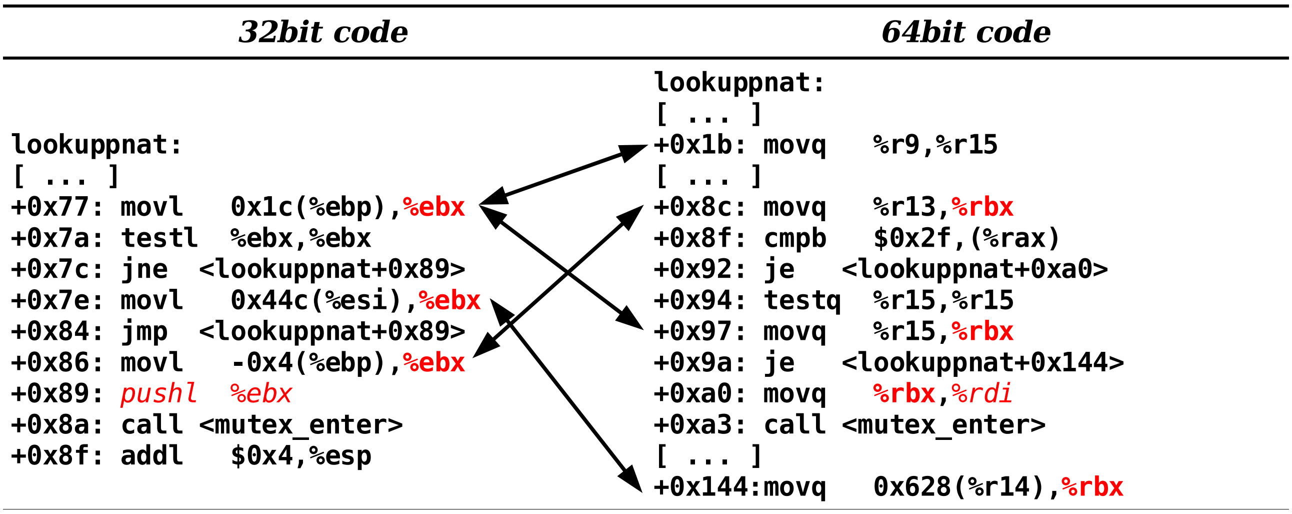
mutex_enter() paniced because **0x1234567890abcdef** (an invalid, noncanonical virtual address) was passed to it as argument:

```
> mutex_enter::dis
mutex_enter:                    movq   %gs:0x18,%rdx
mutex_enter+9:                  xorl   %eax,%eax
mutex_enter+0xb:                lock cmpxchgq %rdx,(%rdi)
[ ... ]
> ::msgbuf
[ ... ]
        rdi: 1234567890abcdef rsi:                0 rdx: ffffffff859ef900
[ ... ]
```

Since **mutex_enter()** itself doesn't push anything to the stack, the stackpointer at the time the BAD TRAP occurred still must point to the return address, giving us the caller of **mutex_enter()**. It's an AMD64 crashdump, so **%rsp** from the HW part of the trap frame is correct and we can directly use that:

```
[ ... ]
pid=657, pc=0xfffffffffe825dbb, sp=0xfffffffffb2ef8cd8, eflags=0x10246
[ ... ]
         cs:               28 rfl:            10246 rsp: fffffffb2ef8cd8
[ ... ]
> fffffffb2ef8cd8/p
0xfffffffb2ef8cd8:              lookuppnat+0xa8
> *fffffffb2ef8cd8::dis
[ ... ]
lookuppnat+0xa0:                movq   %rbx,%rdi
lookuppnat+0xa3:                call   -0xe58e3 <mutex_enter>
lookuppnat+0xa8:                incl   0xc(%rbx)
[ ... ]
```

This tells us **lookuppnat+0xa3** was the location where mutex_enter() was called from. We need to figure out where the argument (**%rdi**) was taken from. The sourcecode for **lookuppnat()** is generic C, and therefore we can expect the 64bit code to have an identical flow as the 32bit code. In fact we find code so similar to the 32bit version that it's worth showing the two side-by-side:



32bit code

```
lookuppnat:
[ ... ]
+0x77: movl   0x1c(%ebp),%ebx
+0x7a: testl  %ebx,%ebx
+0x7c: jne  <lookuppnat+0x89>
+0x7e: movl   0x44c(%esi),%ebx
+0x84: jmp  <lookuppnat+0x89>
+0x86: movl   -0x4(%ebp),%ebx
+0x89: pushl  %ebx
+0x8a: call <mutex_enter>
+0x8f: addl   $0x4,%esp
```

64bit code

```
lookuppnat:
[ ... ]
+0x1b: movq   %r9,%r15
[ ... ]
+0x8c: movq   %r13,%rbx
+0x8f: cmpb   $0x2f,(%rax)
+0x92: je   <lookuppnat+0xa0>
+0x94: testq  %r15,%r15
+0x97: movq   %r15,%rbx
+0x9a: je   <lookuppnat+0x144>
+0xa0: movq   %rbx,%rdi
+0xa3: call <mutex_enter>
[ ... ]
+0x144:movq   0x628(%r14),%rbx
```

32bit code	*64bit code*
	`+0x14b:jmp  <lookuppnat+0xa0>`

The 64bit code, as shown earlier, puts its 6th argument into **%r15** for permanent availability. If that's non-**NULL**, it's being used as argument for **mutex_enter()**. But then the trapframe already has shown that it was **NULL**, so as in 32bit the invalid value cannot have come from there. Can it have come from **0x628(%r14)** ? Getting **%r14** from the trapframe and checking this memory location tells us:

```
[ ... ]
        r13: 1234567890abcdef r14: ffffffff82042c40 r15:                0
[ ... ]
> ffffffff82042c40+628/J
0xffffffff82043268:             ffffffff81c27c40
```

So this wasn't it either. Remains **%r13** (a local variable, as in 32bit). Which in the panic registers holds the culprit, **0x1234567890abcdef**. So check **lookuppnat()** for where it initializes **%r13** from:

```
> lookuppnat::dis ! grep '%r13$'
lookuppnat+0x5a:                movq   0x630(%r14),%r13
lookuppnat+0x61:                testq  %r13,%r13
lookuppnat+0x6a:                cmpq   0x5628cf(%rip),%r13
lookuppnat+0x101:               movq   -0x18(%rbp),%r13
lookuppnat+0x138:               movq   0x562801(%rip),%r13
```

The indirect memory reference **0x630(%r14)** and the local variable **-0x18(%rbp)** are simple to test, and they do not contain **0x1234567890abcdef**:

```
[ ... ]
        rax:                0 rbx: 1234567890abcdef rbp: ffffffffb2ef8d40
[ ... ]
        r13: 1234567890abcdef r14: ffffffff82042c40 r15:                0
> ffffffff82042c40+630/J
0xffffffff82043270:             0
> ffffffffb2ef8d40-18/J
0xffffffffb2ef8d28:             0
```

So these cannot have been it either.
The only thing that remains is the **%rip**-relative location **0x562801(%rip)**. It'd be nice if mdb would resolve this for us (it has the capability, since it does resolve **call** targets, and these are also **%rip**-relative ...), but as it obviously doesn't do that (yet) let's work on it ourselves again. The code is:

```
> lookuppnat+0x138::dis
[ ... ]
lookuppnat+0x138:               movq   0x562801(%rip),%r13
lookuppnat+0x13f:               jmp    -0xb8    <lookuppnat+0x87>
```

What value is used for **%rip** ? The answer is: The same that the **call** instruction would use to put onto the stack as the return address. In other words: In an instruction that makes a **%rip**-relative reference, **%rip** will contain the address of the *following instruction*. Cleartext: **lookuppnat+0x13f** in the case above. Resolve the **%rip**-relative address:

```
> lookuppnat+0x13f+0x562801/J
rootdir:
rootdir:        1234567890abcdef
```

Now there we are. Twisted but in the end we found it - the code indeed takes **%rdi** that it passes to **mutex_enter()** from "**rootdir**", and hence the corrupted pointer.

7.5. Another 64bit crashdump analysis example

The following example shall serve as an advice to device driver developers: *Know your compiler options* !

The machine paniced with the following messages:

```
panic[cpu0]/thread=ffffffff81519000:
BAD TRAP: type=e (#pf Page fault) rp=fffffe80006b3be0 addr=0 occurred in module
"unix" due to a NULL pointer dereference

powernowd:
#pf Page fault
Bad kernel fault at addr=0x0
pid=60, pc=0xfffffffffb826ce6, sp=0xfffffe80006b3cc8, eflags=0x10213
cr0: 80050033<pg,wp,ne,et,mp,pe> cr4: 6f0<xmme,fxsr,pge,mce,pae,pse>
cr2: 0 cr3: 44ce000 cr8: 0
        rdi:                0 rsi: ffffffff80ca0ee0 rdx: fffffffffbc027ff
        rcx:                0  r8:                0  r9: ffffffff80a471b0
        rax:               ff rbx:                0 rbp: fffffe80006b3cf0
        r10:               76 r11:         ffffffff r12: ffffffff80ca0ee0
        r13: fffffffffbe26408 r14: fffffffffbe26400 r15: ffffffff8010ac00
        fsb: ffffffff80000000 gsb: fffffffffbc1ede0  ds:               43
         es:               43  fs:                0  gs:              1c3
        trp:                e err:                2 rip: fffffffffb826ce6
         cs:               28 rfl:            10213 rsp: fffffe80006b3cc8
         ss:               30

fffffe80006b3af0 unix:die+da (fffffe80000adc80, 1fb800ccf)
fffffe80006b3bd0 unix:trap+5ea ()
fffffe80006b3be0 unix:cmntrap+11b ()
fffffe80006b3cf0 unix:lock_try+6 ()
fffffe80006b3d60 genunix:turnstile_block+19e ()
fffffe80006b3dc0 unix:mutex_vector_enter+3df ()
fffffe80006b3df0 genunix:releasef+4b ()
fffffe80006b3ed0 genunix:ioctl+c3 ()
```

On first glance, this looks like a generic Solaris kernel issue – only core kernel modules involved, right ? No immediate hint of any driver involvement !

Yet, this panic was caused by a bug in the following driver module – it was compiled with gcc, and the **-mno-red-zone** compiler option that is mandatory was omitted:

```
> ::modinfo
 ID         LOADADDR     SIZE REV MODULE NAME
[ ... ]
121 fffffffff51962b0      ef0   1 powernow (AMD Powernow! 1.15 (ACPICA))
[ ... ]
```

On the following pages, it'll be shown what happened here.

Well, by now we know what to do in order to find out. The panic message tells us it's a **NULL** pointer dereference, so let's disassemble the panicing code, **lock_try()**, to find out what it attempted to access:

```
> lock_try::dis
lock_try:         movb   $-0x1,%dl
lock_try+2:       movzbq %dl,%rax
lock_try+6:       xchgb  %dl,(%rdi)
```

This is an unusual instruction – e**XCH**an**G**e **B**yte. There are locks in Solaris (thread and dispatcher locks, spinlocks) which don't keep owner information like mutexes and rwlocks, but only know a binary state 'locked'/'free' based on the contents of a single byte. Implementation of these locks requires an atomic “check if Byte contains NULL and if so write my value into it” instruction – which is **xchgb** on x86 (on SPARC, **ldstub** would be used). So we know it attempted to access memory at **%rdi** – which, as the panic messages show, contains **NULL**.

We therefore have to find out where the caller of **lock_try()** got that **%rdi** from. **lock_try()**, as **mutex_enter()**, doesn't initialize a framepointer of its own and neither pushes any values to the stack. So we know that, as with the previous example, the stacktrace misses to show the actual caller of **lock_try()**. The next reported return address, **turnstile_block+0x19e**, of course shows what's in-between:

```
> turnstile_block+19e::dis
[ ... ]
turnstile_block+0x18a:          movq   0x1e0(%r12),%rdi
turnstile_block+0x192:          leaq   0x1e0(%rbx),%rsi
turnstile_block+0x199:          call   -0x2b9   <turnstile_interlock>
turnstile_block+0x19e:          testl  %eax,%eax
```

and the saved **%rsp** gives us the return address into the real caller of **lock_try()**:

```
[ ... ]
pid=60, pc=0xfffffffffb826ce6, sp=0xfffffe80006b3cc8, eflags=0x10213
[ ... ]
         cs:               28 rfl:            10213 rsp: fffffe80006b3cc8
[ ... ]
> fffffe80006b3cc8/p
0xfffffe80006b3cc8:             turnstile_interlock+0x22
> *fffffe80006b3cc8::dis
turnstile_interlock+6:          pushq  %r13
turnstile_interlock+8:          movq   %rdi,%r13
turnstile_interlock+0xb:        pushq  %r12
turnstile_interlock+0xd:        movq   %rsi,%r12
turnstile_interlock+0x10:       pushq  %rbx
turnstile_interlock+0x11:       movq   (%r12),%rbx
turnstile_interlock+0x15:       cmpq   %r13,%rbx
turnstile_interlock+0x18:       je     +0x28    <turnstile_interlock+0x40>
turnstile_interlock+0x1a:       movq   %rbx,%rdi
turnstile_interlock+0x1d:       call   -0x1cf88d        <lock_try>
turnstile_interlock+0x22:       testl  %eax,%eax
[ ... ]
> turnstile_interlock::dis
turnstile_interlock:            pushq  %rbp
turnstile_interlock+1:          movq   %rsp,%rbp
turnstile_interlock+4:          pushq  %r14
turnstile_interlock+6:          pushq  %r13
[ ... ]
```

We could cheat and look up the sourcecode in order to determine what this function does, but since we've not made it past the first **lock_try()** we'll get by just reverse-engineering the first piece of it. It obviously gets two arguments in **%rdi** and **%rsi**, which it – for later re-use after making some function calls of its own – stuffs into the nonvolatile registers **%r13** (1st arg) and **%r12** (2nd arg). It dereferences the second argument, **movq (%r12),%rbx**, and compares what it pointed to with the first. If these two were equal, it'd have taken the **je** to **<turnstile_interlock+0x40>**, but since they obviously have not been it moves *arg2 into %rdi and calls lock_try() on that. Because we do have a C prototype for **lock_try()** in **<sys/machlock.h>**:

```
extern int  lock_try(lock_t *lp);
```

we can derive the prototype and first piece of code for **turnstile_interlock()** from what we've just found out. It must've started like this:

```
... turnstile_interlock(lock_t *lock1, lock_t **plock2)
{
      lock_t *tmplock = *plock2;

      if (tmplock != lock1)
            lock_try(tmplock);
[ ... ]
```

What was that 2nd arg to this function ?

Since it was stuffed into %r12, and that didn't get modified anymore afterwards, we can take it from the panic message. There we find:

```
[ ... ]
        r10:               76 r11:         ffffffff r12: ffffffff80ca0ee0
        r13: fffffffffbe26408 r14: fffffffffbe26400 r15: ffffffff8010ac00
[ ... ]
>  ffffffff80ca0ee0/J
0xffffffff80ca0ee0:             0
```

So arg1, which went into **%r13**, was **0xfffffffffbe26408**, and arg2, still in **%r12**, was **0xffffffff80ca0ee0** and it really pointed to a **NULL**.

Go for the next frame, **turnstile_block()** calling **turnstile_interrupt()**, and dig out where the arguments came from again:

```
> $C
fffffe80006b3cf0 lock_try+6()
fffffe80006b3d60 turnstile_block+0x19e()
fffffe80006b3dc0 mutex_vector_enter+0x3df()
fffffe80006b3df0 releasef+0x4b()
fffffe80006b3ed0 ioctl+0xc3()
fffffe80006b3f20 sys_syscall32+0xd9()
0000000000000320 0x8047b48()
> turnstile_block+0x19e::dis
[ ... ]
turnstile_block+0x18a:          movq   0x1e0(%r12),%rdi
turnstile_block+0x192:          leaq   0x1e0(%rbx),%rsi
turnstile_block+0x199:          call   -0x2b9   <turnstile_interlock>
turnstile_block+0x19e:          testl  %eax,%eax
```

Hmm – it took them from memory referenced via nonvolatile registers **%rbx** and **%r12**. We'd have two immediate ways forward now:

1. The complicated one – disassemble **turnstile_block()**, and attempt to reverse engineer where the values in **%r12** and **%rbx** at that time came from.
   Since we're already deep down in the function, we're not going that way. Rather:

2. The simpler one – because **%rbx** and **%r12** are nonvolatile, the called function, in this case **turnstile_interlock()**, of course has to either keep them as they are (and we could find them in the trapframe/panic registers then), or save them to its stack so they can be restored on return (but as there was no return we'll find them in the stack).

Brings us back to **turnstile_interlock()**, which indeed saves these two to its stack early on:

```
turnstile_interlock:            pushq  %rbp
turnstile_interlock+1:          movq   %rsp,%rbp
turnstile_interlock+4:          pushq  %r14
turnstile_interlock+6:          pushq  %r13
turnstile_interlock+8:          movq   %rdi,%r13
turnstile_interlock+0xb:        pushq  %r12      ←
turnstile_interlock+0xd:        movq   %rsi,%r12
```

```
turnstile_interlock+0x10:       pushq  %rbx      ←
```

Now consider the stack contents. This function writes to the stack:

1. it's caller's framepointer, **%rbp**

2. the nonvolatile registers **%r14**, **%r13**, **%r12** and **%rbx**, in that order.

3. when it **call**s **lock_try()**, the return address.

So we can now identify the stack contents:

```
[ ... ]
pid=60, pc=0xfffffffffb826ce6, sp=0xffffe80006b3cc8, eflags=0x10213
[ ... ]
         cs:              28 rfl:           10213 rsp: ffffe80006b3cc8
[ ... ]
> ffffe80006b3cc8,7/nap
0xffffe80006b3cc8:
0xffffe80006b3cc8:      turnstile_interlock+0x22
0xffffe80006b3cd0:      0xffffffff80ca0d00          %rbx
0xffffe80006b3cd8:      0xffffffff81519000          %r12   of the caller,
0xffffe80006b3ce0:      0xffffffff80a47180          %r13   turnstile_block()
0xffffe80006b3ce8:      turnstile_table+0xd60       %r14
0xffffe80006b3cf0:      0xfffffe80006b3d60          %rbp
0xffffe80006b3cf8:      turnstile_block+0x19e
```

We can now check where these args came from:

```
[ ... ]
turnstile_block+0x18a:          movq   0x1e0(%r12),%rdi
turnstile_block+0x192:          leaq   0x1e0(%rbx),%rsi
[ ... ]
> 0xffffffff81519000+1e0/nJ
0xffffffff815191e0:
                fffffffffbe26408    %rdi - 1st arg
> 0xffffffff80ca0d00+1e0=J
                ffffffff80ca0ee0    %rsi - 2nd arg (note: =J due to leaq)
```

This is all well and consistent, but we're not further yet, and therefore still have to go the long way, and determine where **turnstile_block()** got the values from that it keeps in **%r12** (**0xffffffff81519000**) and **%rbx** (**0xffffffff80ca0d00**). We're particularly interested in the latter, **%rbx** – that's where the bad pointer which got passed on came from. Disassemble:

```
> turnstile_block::dis ! egrep '%r(12|bx)$'
turnstile_block+0x10:           pushq  %r12
turnstile_block+0x12:           pushq  %rbx
turnstile_block+0x26:           movq   %gs:0x18,%rbx
turnstile_block+0x36:           movq   %rbx,%r12
turnstile_block+0x173:          movq   %rax,%rbx
turnstile_block+0x1b3:          movq   %gs:0x18,%r12
turnstile_block+0x202:          popq   %rbx
turnstile_block+0x203:          popq   %r12
turnstile_block+0x2cb:          movq   %rbx,%r12
```

The **push**/**pop** are part of function pro- and epilogues so those don't bother us (yet). The **movq %gs:0x18,...** is a curthread access, as shown before. The function also moves **%rbx** to **%r12** two times, which doesn't bother us either because we want to know where **%rbx** is from. The only place that sets **%rbx** is **movq %rax,%rbx** so let's look at this area more closely:

```
> turnstile_block+0x173::dis
turnstile_block+0x149:          movq   -0x38(%rbp),%rdx
turnstile_block+0x14d:          cmpl   $0x1,(%rdx)
```

```
turnstile_block+0x150:          je     +0x233   <turnstile_block+0x383>
turnstile_block+0x156:          movq   0x88(%r12),%rdx
turnstile_block+0x15e:          testq  %rdx,%rdx
turnstile_block+0x161:          je     +0x70    <turnstile_block+0x1d1>
turnstile_block+0x163:          xorl   %eax,%eax
turnstile_block+0x165:          movq   0x80(%r12),%rdi
turnstile_block+0x16d:          call   *0x8(%rdx)
turnstile_block+0x170:          testq  %rax,%rax
turnstile_block+0x173:          movq   %rax,%rbx
turnstile_block+0x176:          je     +0x5b    <turnstile_block+0x1d1>
turnstile_block+0x178:          movq   %gs:0x18,%rax
turnstile_block+0x181:          cmpq   %rbx,%rax
turnstile_block+0x184:          je     +0x21f   <turnstile_block+0x3a3>
turnstile_block+0x18a:          movq   0x1e0(%r12),%rdi
turnstile_block+0x192:          leaq   0x1e0(%rbx),%rsi
turnstile_block+0x199:          call   -0x2b9   <turnstile_interlock>
turnstile_block+0x19e:          testl  %eax,%eax
[ ... ]
```

Hmm – a bit unfair, there's a call through a function pointer in there. But we're fortunate – just a few lines above we find: **movq 0x88(%r12),%rdx**. We already found out what **%r12** was, and so getting this is straightforward:

```
> 0xffffffff81519000+0x88/J
0xffffffff81519088:             fffffffffbc02720
> fffffffffbc02720::whatis
fffffffffbc02720 is mutex_sobj_ops+0 in unix's data segment
> fffffffffbc02720::whattype
fffffffffbc02720 is fffffffffbc02720+0, struct _sobj_ops
> fffffffffbc02720::print -a "struct _sobj_ops"
{
    fffffffffbc02720 sobj_type = 0x1
    fffffffffbc02728 sobj_owner = mutex_owner
    fffffffffbc02730 sobj_unsleep = turnstile_stay_asleep
    fffffffffbc02738 sobj_change_pri = turnstile_change_pri
}
```

Fortunately, we know that one. **mutex_owner()** takes a mutex as arg, and returns a kernel thread pointer (or **NULL**). The argument comes from:

```
turnstile_block+0x165:          movq   0x80(%r12),%rdi
> 0xffffffff81519000+0x80/J
0xffffffff81519080:             ffffffff8010ac00
> ffffffff8010ac00::mutex
            ADDR  TYPE             HELD MINSPL OLDSPL WAITERS
ffffffff8010ac00 adapt ffffffff80ca0d00       -      -    yes
```

The owner of this mutex is in fact thread **0xffffffff80ca0d00** – which is in **%rbx**, so we've found where this has come from.

And we seem no further. Where is that corruption anyway ? Are we stuck ?

The answer is simpler than that. Do some consisteny checks on the “mutex” and the “owner thread” above:

```
> ffffffff8010ac00::whatis
ffffffff8010ac00 is in thread fffffffffbc1dac0's stack
> ffffffff8010ac00::whattype
ffffffff8010ac00 is ffffffff8010aa80+180, possibly one of the following:
  void (from fffffffffbc7a7a0+38, type struct vfs)
  struct ufsvfs (from ffffffff80c92008+c8, type inode_t)

> ffffffff80ca0d00::whatis
```

```
ffffffff80ca0d00 is in thread ffffffffbc1dac0's stack
> ffffffff80ca0d00::whattype
ffffffff80ca0d00 is ffffffff80ca0d00+0, possibly one of the following:
  struct ml_unit at ffffffff80ca0d01 (from ffffffff8010aa80+180, type
struct ufsvfs)
  void at ffffffff80ca0e20 (from ffffffff80a47300+10, type struct turnstile)
```

Hmm – this looks pretty bad. mdb tells us that both these values, the “mutex” and the “owner” are in some thread's stack – and at best, are some ufs type data structures. The attempt to use the “owner” as a thread struct fails quite horribly:

```
> ::load /opt/CTEact/mdb/5.10/amd64/act.so
> ffffffff80ca0d00::act_thread

***
could not get pid baddcafebaddcafe for proc ffffffff806a9e20 thread
ffffffff80ca0d00
thread swapped

thread addr ffffffff80ca0d00, proc addr ffffffff806a9e20, lwp addr 0
Thread beacb004b6600 is pinned by this thread
Thread bound to cpu id 0xfbc7a7a0

t_state is 0xffffffff - illegal value - check

Scheduling info:
t_pri is 0x20, t_epri is 0, t_cid is 0x80127140
t->t_cid out of bounds
t_disp_time: is 0, 0
last ran: 3 hours 39 mins 14 secs ago,  3 hours 39 mins 10 secs before panic
last ran cpu is corrupt 0x6d6000006d6000t_stk 0

stack trace is:

ffffffff80ca0bc0 ()
[ ... ]
```

That makes no sense. This “mutex” is no mutex, and this “owner” is no thread.

With this knowledge in mind, it makes sense to skip further reverse engineering attempts on **turnstile_block()**, which seems more than incomprehensible based on the assembly code only, and go back to the stacktrace instead:

```
> $C
fffffe80006b3cf0 lock_try+6()
fffffe80006b3d60 turnstile_block+0x19e()
fffffe80006b3dc0 mutex_vector_enter+0x3df()
fffffe80006b3df0 releasef+0x4b()
fffffe80006b3ed0 ioctl+0xc3()
fffffe80006b3f20 sys_syscall32+0xd9()
0000000000000320 0x8047b48()
```

Let's investigate the call to mutex_enter(), and see whether we can find out what mutex was passed in. Should we in fact find that the caller of mutex_enter() passed that “non-mutex” 0xffffffff8010ac00, then we know the bug must be there – one cannot call mutex_enter() with just some more-or-less arbitrary address and hope all will be fine !

For those who like to peek ahead, cheat and use **::act_thread**:

```
> *panic_thread::act_thread

***
```

```
thread addr ffffffff81519000, proc addr 0, lwp addr ffffffff81519000
Thread bound to cpu id 0
[ ... ]
stack trace is:
[ ... ]
unix: lock_tryunix: lock_try+6 ()
genunix: turnstile_interlock (?,0xffffffff80ca0ee0)
genunix: turnstile_block+0x19e (0,0,0xffffffff8010ac00,mutex_sobj_ops,0,0)


**********

failed to find mutex pointer in stack frames fffffe80006b3dc0

**********
unix: mutex_vector_enter+0x3df (0xffffffff8010ac00)
unix: mutex_enter (?)
genunix: releasef+0x4b (0x30)
genunix: ioctl+0xc3 (0x6b3f20,0xfb801149,3)
unix: sys_syscall32+0xd9 ()
```

And bingo – **mutex_vector_enter()** was called with this “thing”.

But then, we can of course find that from the caller, releasef(), as well. Disassembly:

```
releasef+0x3a:                movq   0x18(%r14),%rbx         ←
releasef+0x3e:                leaq   0x0(%r13,%rbx),%r12     ←
releasef+0x43:                movq   %r12,%rdi
releasef+0x46:                call   -0x121246       <mutex_enter>
releasef+0x4b:                movq   0x18(%r14),%rax
```

So we need to find **%r12**, and **%r13/%r14/%rbx** if we want to know where it came from. All these are nonvolatile registers, and if we're lucky, **mutex_vector_enter()** will have saved them for us.

Let's check its prologue:

```
mutex_vector_enter:           pushq  %rbp
mutex_vector_enter+1:         movq   %rsp,%rbp
mutex_vector_enter+4:         pushq  %r15
mutex_vector_enter+6:         movq   %rdi,%r15
mutex_vector_enter+9:         pushq  %r14
mutex_vector_enter+0xb:       pushq  %r13
mutex_vector_enter+0xd:       pushq  %r12
mutex_vector_enter+0xf:       pushq  %rbx
mutex_vector_enter+0x10:      subq   $0x28,%rsp
```

They're all there. Looking at the stack:

```
[ ... ]
fffffe80006b3dc0 mutex_vector_enter+0x3df()
[ ... ]
> fffffe80006b3dc0-30,10/nap
0xfffffe80006b3d90:
0xfffffe80006b3d90:           0xfffff3ffd596e89e
0xfffffe80006b3d98:           0xffffffff8010a000    %rbx
0xfffffe80006b3da0:           0xffffffff8010ac00    %r12
0xfffffe80006b3da8:           0xc00                 %r13
0xfffffe80006b3db0:           0xffffffff80e2d708    %r14
0xfffffe80006b3db8:           0                     %r15
0xfffffe80006b3dc0:           0xfffffe80006b3df0    %rbp
0xfffffe80006b3dc8:           releasef+0x4b
[ ... ]
```

↑ stack growth towards lower addresses

```
releasef+0x3a:                  movq   0x18(%r14),%rbx
releasef+0x3e:                  leaq   0x0(%r13,%rbx),%r12
[ ... ]
> 0xffffffff80e2d708+18/nJ
0xffffffff80e2d720:
                ffffffff8010a000
> ffffffff8010a000+0xc00=J
                ffffffff8010ac00
```

So there it is. The address of something that was not a pointer to a mutex was passed into **mutex_enter()**.

Where did **releasef()** take **%r13**/**%r14**/**%rbx** from:

```
releasef:                       pushq  %rbp
releasef+1:                     movq   %rsp,%rbp
releasef+4:                     pushq  %r14
releasef+6:                     pushq  %r13
releasef+8:                     pushq  %r12
releasef+0xa:                   pushq  %rbx
releasef+0xb:                   movl   %edi,%ebx
releasef+0xd:                   movq   %gs:0x18,%rax
releasef+0x16:                  movq   0x178(%rax),%r14
releasef+0x1d:                  movslq %ebx,%r13
releasef+0x20:                  shlq   $0x6,%r13
releasef+0x24:                  call   -0x4f4   <clear_active_fd>
releasef+0x29:                  addq   $0xa30,%r14
releasef+0x30:                  jmp    +0xa     <releasef+0x3a>
[ ... ]
releasef+0x3a:                  movq   0x18(%r14),%rbx
releasef+0x3e:                  leaq   0x0(%r13,%rbx),%r12
releasef+0x43:                  movq   %r12,%rdi
releasef+0x46:                  call   -0x121246        <mutex_enter>
```

We didn't make it further than that, so the relevant instructions are:

For **%r14**:

```
releasef+0xd:                   movq   %gs:0x18,%rax
releasef+0x16:                  movq   0x178(%rax),%r14
[ ... ]
releasef+0x29:                  addq   $0xa30,%r14
> *panic_thread+0x178/nJ
0xffffffff81519178:
                ffffffff80e2ccd8
> *panic_thread::print -a kthread_t ! grep ffffffff81519178
    ffffffff81519178 t_procp = 0xffffffff80e2ccd8
> ffffffff80e2ccd8+a30=J
                ffffffff80e2d708
> ffffffff80e2ccd8::print -a proc_t ! grep ffffffff80e2d708
        ffffffff80e2d708 u_finfo = {
            ffffffff80e2d708 fi_lock = {
                ffffffff80e2d708 _opaque = [ 0 ]
```

For **%rbx**:

```
releasef+0x3a:                  movq   0x18(%r14),%rbx
> 0xffffffff80e2d708+18/nJ
0xffffffff80e2d720:
                ffffffff8010a000
```

and finally **%r13**:

```
releasef+0xb:                   movl   %edi,%ebx
[ ... ]
```

```
releasef+0x1d:                  movslq %ebx,%r13
releasef+0x20:                  shlq   $0x6,%r13
```

This takes the first argument, **%edi** (obviously an **int**) of **releasef()**, sign-extends it to 64bit and shifts it left by 0x6 (i.e. multiples it by 2^6 == 64).

We need the arg to **releasef()**. Since that code in itself didn't save **%edi** anywhere, we have to check the caller:

```
> $C
[ ... ]
fffffe80006b3df0 releasef+0x4b()
fffffe80006b3ed0 ioctl+0xc3()
[ ... ]
> ioctl+0xc3::dis
[ ... ]
ioctl+0xbb:                     movl   %r14d,%edi
ioctl+0xbe:                     call   -0x1091e <releasef>
ioctl+0xc3:                     movl   -0xcc(%rbp),%eax
[ ... ]
```

It takes the value from nonvolatile register **%r14d**. Given that **releasef()** has saved **ioctl()**'s **%r14** in its prologue, we can find it in the stack::

```
releasef:                       pushq  %rbp
releasef+1:                     movq   %rsp,%rbp
releasef+4:                     pushq  %r14
> fffffe80006b3df0-8,3/nap
0xfffffe80006b3de8:
0xfffffe80006b3de8:             0x30
0xfffffe80006b3df0:             0xfffffe80006b3ed0
0xfffffe80006b3df8:             ioctl+0xc3
> 0x30*0t64=X
                c00
```

So the argument was **0x30**, and **0xc00** is the result of the **<<6**. Next step of course is to determine where ioctl() got its %r14 from. Check the assembly:

```
> ioctl::dis
ioctl:                          pushq  %rbp
ioctl+1:                        movq   %rsp,%rbp
ioctl+4:                        subq   $0xd0,%rsp
ioctl+0xb:                      movq   %r12,-0x20(%rbp)
ioctl+0xf:                      movq   %r13,-0x18(%rbp)
ioctl+0x13:                     movl   %esi,%r13d
ioctl+0x16:                     movq   %r14,-0x10(%rbp)
ioctl+0x1a:                     movq   %r15,-0x8(%rbp)
ioctl+0x1e:                     movl   %edi,%r14d
[ ... ]
ioctl+0xbb:                     movl   %r14d,%edi
ioctl+0xbe:                     call   -0x1091e <releasef>
ioctl+0xc3:                     movl   -0xcc(%rbp),%eax
```

It does the same as we've seen **releasef()** do – move its first argument into the nonvolatile register **%r14d**. What was the first argument of **ioctl()** ?

```
> $C
[ ... ]
fffffe80006b3ed0 ioctl+0xc3()
fffffe80006b3f20 sys_syscall32+0xd9()
[ ... ]
> sys_syscall32+0xd9::dis
sys_syscall32+0xbc:             movl   0x0(%rsp),%edi
sys_syscall32+0xc0:             movl   0x8(%rsp),%esi
```

```
sys_syscall32+0xc4:             movl   0x10(%rsp),%edx
sys_syscall32+0xc8:             movl   0x18(%rsp),%ecx
sys_syscall32+0xcc:             movl   0x20(%rsp),%r8d
sys_syscall32+0xd1:             movl   0x28(%rsp),%r9d
sys_syscall32+0xd6:             call   *0x18(%rbx)
sys_syscall32+0xd9:             movq   %rbp,%rsp
```

This code obviously transforms 32bit argument passing conventions (on the stack) into 64bit argument passing conventions (in the arg registers). We therefore need the stackpointer at the time ioctl() was called and retrieve its first argument from there.

The task is of course easy – there's a call being done, and if we locate the return address from that in the stack we immediately know that the six words following it contains the arguments:

```
[ ... ]
fffffe80006b3ed0 ioctl+0xc3()
[ ... ]
> fffffe80006b3ed0,8/nap
0xfffffe80006b3ed0:
0xfffffe80006b3ed0:             0xfffffe80006b3f20
0xfffffe80006b3ed8:             sys_syscall32+0xd9
0xfffffe80006b3ee0:             3                 %edi
0xfffffe80006b3ee8:             0x414d4400        %esi
0xfffffe80006b3ef0:             0                 %edx
0xfffffe80006b3ef8:             0                 %ecx
0xfffffe80006b3f00:             0xffffffff8142ac00    %r8d
0xfffffe80006b3f08:             0xffffffff81519000    %r9d
```

Now this is *really* strange – **ioctl()** got called with **3** as 1st argument, and the code claims to have passed that through to **releasef()**. But this was called with **0x30** as 1st argument !

This is a clear inconsistency. Can it be explained ?

The only explanation is that something corrupted **%r14** in between the instruction at **ioctl+0x1e** (which initialized it) and the call to **releasef()** at **ioctl+0xbe**. And indeed we make function calls there:

```
[ ... ]
ioctl+0x32:                     call   -0x10ce2 <getf>
[ ... ]
ioctl+0x76:                     call   -0x130b46        <get_udatamodel>
[ ... ]
ioctl+0x8e:                     movq   0x10(%r12),%rdi
ioctl+0x93:                     movq   0x1b0(%rax),%r8
ioctl+0x9a:                     leaq   -0xcc(%rbp),%r9
ioctl+0xa1:                     movq   %r15,%rdx
ioctl+0xa4:                     movl   %r13d,%esi
ioctl+0xa7:                     call   +0xb9969 <fop_ioctl>
```

The first two are not really suspicious. For one, they're well-known and long-tried kernel functions, and second their code is quite simple. It's left as an exercise to the reader to verify that these don't modify **%r14** without restoring it for their caller.

But the other function is more suspicious – it's a generic dispatcher, and will ultimately end up in a *device driver* entry point. We can find which one without reverse engineering **fop_ioctl()**, by checking the panicing process' file table:

```
> *panic_thread::print -a kthread_t t_procp
ffffffff81519178 t_procp = 0xffffffff80e2ccd8
> 0xffffffff80e2ccd8::pfiles
FD   TYPE            VNODE INFO
```

```
   0  CHR ffffffff81201180 /devices/pseudo/mm@0:null
   1  REG ffffffff81552040 /etc/svc/volatile/site-powernow:default.log
   2  REG ffffffff81552040 /etc/svc/volatile/site-powernow:default.log
   3  CHR ffffffff814fa4c0 /devices/pseudo/powernow@0:powernow
   4  CHR ffffffff815e89c0 /devices/pseudo/acpidrv@0:acpidrv
   5  CHR ffffffff8163f840 /devices/pseudo/kstat@0:kstat
```

We're operating on file descriptor #3 – and that is the powernow driver. What's the **ioctl(9e)** entry point for that ?

```
> ::modctl
          MODCTL           MODULE   BITS FLAGS REF FILE
[ ... ]
ffffffff8126fdc8 ffffffff8126ea80     li 0x00   0 drv/powernow
> ffffffff8126fdc8::print "struct modctl" mod_linkage |
                  ::print "struct modlinkage"
{
    ml_rev = 0x1
    ml_linkage = [ modldrv1, 0, 0, 0, 0, 0, 0 ]
}
> modldrv1::print "struct modldrv"
{
    drv_modops = mod_driverops
    drv_linkinfo = 0xfffffffff5196e72 "AMD Powernow! 1.15 (ACPICA)"
    drv_dev_ops = powernow_dev_ops
}
> powernow_dev_ops::print "struct dev_ops" devo_cb_ops |
                  ::print "struct cb_ops" cb_ioctl
cb_ioctl = powernow_ioctl
```

Who would've guessed. Let's look into this:

```
> powernow_ioctl::dis
powernow_ioctl:                 pushq  %rbp
powernow_ioctl+1:               movq   %rsp,%rbp
powernow_ioctl+4:               movq   %rbx,-0x20(%rbp)
powernow_ioctl+8:               movq   %r12,-0x18(%rbp)
powernow_ioctl+0xc:             movq   %r13,-0x10(%rbp)
powernow_ioctl+0x10:            movq   %r14,-0x8(%rbp)
powernow_ioctl+0x14:            subq   $0x30,%rsp
[ ... ]
```

Wow – ***bad bad bad***.

What does this code do ? It looks like it saves the nonvolatile registers into hidden local variables, **-...(%rbp)**, doesn't it ?

Well – it may save them, but what it forgot to do and only does ***afterwards***, is to ***allocate stackspace*** for this !

That's a big no-no in a preemptive kernel such as Solaris. If an *interrupt occurs* before the **subq $0x30,%rsp** has caught up with the required stackspace allocation, the values of **%rbx**/**%r12**/**%r13** and **%r14** just written to the stack will be clobbered by the interrupt frame !

Which is precisely what must have happened. Remember **%r14** contained **0x30** on return ? Well – check ***any*** trapframe/interrupt frame that occurred in kernel, and you'll find at the very bottom of it:

```
panic[cpu0]/thread=ffffffff81519000:
BAD TRAP: type=e (#pf Page fault) rp=fffffe80006b3be0 addr=0 occurred in module
"unix" due to a NULL pointer dereference

powernowd:
```

```
#pf Page fault
Bad kernel fault at addr=0x0
pid=60, pc=0xfffffffffb826ce6, sp=0xfffffe80006b3cc8, eflags=0x10213
[ ... ]
> fffffe80006b3be0::print -a "struct regs" ! tail -2
    fffffe80006b3cb8 r_ss = 0x30
}
```

Now this is where the **`0x30`** comes from – as the value of **`KDS_SEL`**, the kernel 64bit data segment selector that is in **`%ss`** when the kernel runs.

We've seen code like this before – check the **`if() {} else`** statement source example in section 2.7.2. The GNU gcc compiler generates code like this in 64bit x86 as an "optimization" if the **`-mno-red-zone`** compile flag is omitted. The AMD64 UNIX ABI allows this for application code – but a preemptive kernel cannot allow this due to the way interrupts work on x86. Therefore:

Solaris Kernel Modules must be compiled with `-mno-red-zone` if gcc is used.

Morale:

If you're a developer, know your compiler's default options !

7.5.1.Unwanted gcc optimizations and the Solaris kernel

The GNU gcc compiler's default behaviour when compiling code for the kernel code model (**`-mcmodel=kernel`**) on 64bit x86 with optimization on (**`-O2`** or even above) isn't compatible with the Solaris kernel for several reasons:

1. The compiler uses the stack redzone, although that behaviour is incompatible with kernel code on all preemptive kernels – whether Linux, Solaris or one of the *BSDs. It's essential to disable the use of the redzone via **`-mno-red-zone`**. The example in the previous section should have shown how problematic it is to trace this bug.

2. The compiler will attempt to use builtin inline versions of functions like **`bcopy()`**, **`memcpy()`** and the **`str...()`** function family. But gcc's builtins can break the kernel because floating point / vector registers are not usable from kernel mode.
The use of builtins must be disabled via **`-fno-builtins`**.

3. The compiler by default assumes to be able to link binaries with **`libgcc.a`** that contains common "utility" code for gcc. This is not available for kernel drivers, which therefore must be compiled with **`-ffreestanding`**, which instructs gcc to create "standalone" code without any dependencies on external libraries/objects.

4. Some gcc versions optimize away the use of the framepointer when specifying **`-O2`**. While framepointer-less kernel driver code will work, it is incompatible with DTrace. Code that is supposed to be dynamically instrumented via DTrace *requires* framepointers. gcc can be instructed not to optimize away framepointers by using the **`-fno-omit-framepointer`** option. It's the default for gcc 3.4.3 as shipped with Solaris 10.

There are other gcc optimizations that don't cause problems in all cases but still may have negative impact depending in expectations of the sourcecode. Examples of these would be:

5. gcc does what it calls *strict aliasing* – type-based optimization. If this is on, the compiler optimizes data accesses depending on structure layout and the data types involved. For driver code, though, that may be a problem in cases where *packed structures* (or C **`union`**s) are used to map e.g. device hardware registers. Whether **`-fno-strict-aliasing`** is required to disable this behaviour therefore depends on the specific code being compiled.

6. gcc does optimization like *per-* or *cross-sourcefile function inlining* or *eliminating* functions that are *tail-called* only. This doesn't cause failures but it may cause undesirable results (functions disappearing which the developer expexts to see e.g. in crashdumps, or via DTrace) and can be disabled, via the options -**`fno-inline`**, **`-fno-unit-at-a-time`** or **`-fno-optimize-sibling-calls`**.

For more gcc options, see [*http://gcc.gnu.org/onlinedocs/gcc/*](http://gcc.gnu.org/onlinedocs/gcc/), “Optimizations”.

7.6.AMD64 ABI – Backtracing without framepointers

7.7.Examples on application coredump analysis on Solaris/x86

7.8.Advanced Debugging Topics

7.8.1.The Solaris/x86 boot debugger

7.8.2.Debugging hard hangs on Solaris/x86

8.Lab Exercises

8.1.Introduction to x86 assembly language

1. Name five different x86-compatible CPUs that run Solaris 10.
   - how do they differ ?
   - Which features does CPU X have but CPU Y not, and vice versa ?
   - Which of these features are being used in Solaris 10 ?

2. Why does Solaris 10 not run on either of the following CPUs:
   - Intel i8086,
   - Intel i80286,
   - AMD 486,
   - Motorola 68040 ?

   Which ones of these run *any* version of Solaris ?

3. Create a file "**testfile**" with random contents using the following command:

   ```
   dd if=/dev/random of=testfile bs=1024 count=1
   ```

   Try to disassemble the contents of testfile using the command:

   ```
   echo "0,100?ai" | mdb testfile
   ```

   - on a Solaris 10/x86 machine
   - on a Solaris 10/SPARC machine.

   What do you find ?

4. On a machine running a 64bit Solaris 10/x86 kernel, try disassembling **testfile** in both 32bit and 64bit disassembly mode using the following command:

   ```
   mdb testfile <<EOI
   ::dismode ia32; 0,100?ai
   ::dismode amd64; 0,100?ai
   EOI
   ```

   - Is there any difference between the 32bit and 64bit output ?
   - If so, what is different ?

5. What is the x86 machine code for the following assembly instruction:

   ```
   addb %al,(%eax)
   ```

6. Take testfile created above and copy it onto a SPARC and a x86 machine. Inspect its contents using the commands:

   ```
   od -N 128 -x testfile
   ```

```
od -N 128 -t x1 testfile
od -N 128 -t x2 testfile
od -N 128 -t x4 testfile
od -N 128 -t x8 testfile
```

Compare the output on SPARC and x86 and describe what you see.

7. Create the following assembly sourcefile:

```
/ amd64-impl.s
/ demonstrate how AMD extended the x86 instruction set
.globl func
func:
      addb  %bl, %al
      addb  %r11b, %al
      addb  %bl, %r10b
      addb  %r11b, %r10b
      addw  %bx, %ax
      addw  %r11w, %ax
      addw  %bx, %r10w
      addw  %r11w, %r10w
      addl  %ebx, %eax
      addl  %r11d, %eax
      addl  %ebx, %r10d
      addl  %r11d, %r10d
      addq  %rbx, %rax
      addq  %r11, %rax
      addq  %rbx, %r10
      addq  %r11, %r10
.size func,.-func
```

and assemble it (don't forget **-xarch=amd64**, to instruct the assembler to use AMD64 input mode). Disassemble the resulting object file via **/usr/ccs/bin/dis**.

Identify the patterns in the binary code.

- what do all **addb** instructions have in common ?
- what do all **addw** instructions have in common ?
- ... **addl/addq** instructions ?
- what differs between instructions using **%bl/%bx/%ebx/%rbx** and instructions using the various widths of **%r11** ? Dito for **%rax/%r10** ?
- Can you derive from the binary code how the AMD64 extension was done ?

If not: Disassemble the object file with mdb, switching into ia32 disassembly mode (**::dismode ia32**). What's the same and what's different to the 64bit version ?

8. Create the following assembly sourcefile:

```
/ addressing mode redundancy in 32bit x86
/ demonstrate how %rip-relative addressing is done in AMD64
.globl func
func:
      movl  0x1234, %ebx
.size func,.-func
```

and assemble it (**cc -c ...** for 32bit, **cc -xarch=amd64 -c ...** for 64bit)

- Assemble it as 32bit, then disassemble the object file via **/usr/ccs/bin/dis**.

- Assemble it as 64bit, then disassemble the object file.

What if any are the differences ? Next, Change the source to:

```
func:
      movl  0x1234, %ebx
      movl  0x1234(%rip), %ebx
.size func,.-func
```

- Assemble this (64bit required now, of course).
- Disassemble it in mdb, using the commands:
  `::dismode amd64; func::dis`
  `::dismode ia32; func::dis`

How was the **`%rip`**-relative addressing done in 64bit mode ?

9. Compile the following sourcefile, into optimized and non-optimized 32bit and 64bit code, both with gcc and Sun Workshop cc:

```
#include <string.h>
#include <stdlib.h>

typedef struct _str {
        int str_i;
        char str_c;
        long long str_ll;
        char str_name[256];
        struct _str *str_nxt;
} str_t;

void init_str(str_t *initme, int i, char c, long long ll, str_t *nxt)
{
        str_t lv;
        int j;

        lv.str_i = i;
        lv.str_c = c;
        lv.str_ll = ll;
        lv.str_nxt = nxt;

        for (j = c; j < i; j++)
                lv.str_name[j] = (char)j;

        memcpy(initme, &lv, sizeof(str_t));
}
```

For each case, disassemble the resulting binary and identify the instructions that access the array, **`lv.str_name[j]`**. What is used to access memory ?

What assembly code (optimized vs. non-optimized, gcc vs. Workshop cc) is best readable ?

10. Change the above program and make **`lv`** a global (instead of a local) variable. Repeat the steps above, for 64bit optimized compilation. What changes ?

11. Change the program again to directly initialize **`*initme`** (no pre-creation of an instance of the structure and memcpy). Perform both 32bit and 64bit optimizing compiles. What changes in the assembly code ?

12.Compile the following sourcecode:

```
int main(int argc, char **argv)
{
        return (3 * argc);
        /*
         * try also:
         *      return (5 * argc)
         *      return (9 * (argc + 1))
         */

}
```

Use optimization (**-O2** or higher in gcc, **-xO3** or above in Workshop cc). What assembly instructions are used to perform the multiplication ?

8.2.Stacks and Stacktracing

1. Compile the following program into a 64bit x86 executable:

```
#include <stdlib.h>
#include <stdio.h>

long
overflow_me(long a1, long a2, long a3, long a4, long a5, long a6, long a7)
{
        long localvar;

        localvar = a1 + a2 - a3 * a4 / a5 ^ a6 | a7;
        return (localvar + overflow_me(lrand48(), lrand48(), lrand48(),
                        lrand48(), lrand48(), lrand48(), lrand48()));
}

int main(int argc, char **argv)
{
        printf("Let's overflow: %ld\n",
                overflow_me(1, 2, 3, 4, 5, 6, 7));
        return (0);
}
```

Use optimization (**-O2** in gcc, or **-xO3** in cc) because that makes the assembly code better readable.

- open the program in mdb.
- put a breakpoint in **main** (**main:b**)
- run the program (**:r**)
- When the breakpoint is hit, dump the top of the stack (**<rsp,10/nap**). What is on the stack ?
- disassemble **main** and identify the instructions that put values into the stack.
- single-step through **main** and dump the top of the stack after each step to verify your findings (**:s;<rsp,10/nap**). Stop singlestepping when you're in **overflow_me** at the first **call lrand48**.
- Explain all the values on the stack between the stackpointer at this time and the framepointer of **main**.
- Where does the stackframe of **main()** start, and where does it end ? Give the addresses !
- Can you find the arguments of **main()** ?
- Continue the program (**:c**). It will segfault – why ?

2. A crashdump is provided with the electronic parts of the course material that has the following stacktrace:

```
> $C
ffffffffb2ef8d40 mutex_enter+0xb()
ffffffffb2ef8e00 lookupnameat+0x86()
ffffffffb2ef8e50 cstatat_getvp+0x115()
ffffffffb2ef8eb0 cstatat64_32+0x49()
ffffffffb2ef8ec0 stat64_32+0x22()
ffffffffb2ef8f20 sys_syscall32+0xd0()
```

- Identify the stack boundaries (bottom and top of the stackframes) for the listed functions. I.e. for all functions in the call trace, find:

- What is the first word written to the stack by a function ?
- What is the last word written to the stack by the same function ?

- Find out based on reading the assembly code for function and caller how many arguments the listed functions have.
- Find the arguments. To do so, emply the following strategies:
  - load the act module and let that do the work for you:
    ```
    ::load /opt/CTEact/mdb/5.10/amd64/act.so
    *panic_thread::act_thread
    ```
  - Search them manually, at least for those cases where act fails to print them. The seach strategy is:
    - Disassemble the function.
      - See whether it puts arguments registers into its stack.
        If it does, retrieve the values from the stack of the function.
      - See whether it moves argument registers into nonvolatiles.
        If it does, disassemble the next function in the calltrace and see where it saves the nonvolatiles.
      - If neither, disassemble the caller and see where it takes the arguments from.
        If they come from local variables of the caller, they'll be in the caller's stack.
        If they come from nonvolatile registers of the caller, they have been saved by the function itself.

9.References

1. Endianness Whitepaper
   http://www.intel.com/design/intarch/papers/endian.pdf
2. External Data Representation Standard
   http://www.ietf.org/rfc/rfc1832.txt
3.
4.

10.License

The first version of this book was written by Frank Hofmann.

You're allowed to modify this under the terms and conditions stated by the Creative Commons “Attribution-ShareAlike” License, Version 2.5:

http://creativecommons.org/licenses/by-sa/2.5/

Summary:

Attribution-ShareAlike 2.5

You are free:

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

Full terms in print:

creative commons
LEGAL CODE

Attribution-ShareAlike 2.5

CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN ATTORNEY-CLIENT RELATIONSHIP. CREATIVE COMMONS PROVIDES THIS INFORMATION ON AN "AS-IS" BASIS. CREATIVE COMMONS MAKES NO WARRANTIES REGARDING THE INFORMATION PROVIDED, AND DISCLAIMS LIABILITY FOR DAMAGES RESULTING FROM ITS USE.

License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

a. ***"Collective Work"*** means a work, such as a periodical issue, anthology or encyclopedia, in which the Work in its entirety in unmodified form, along with a number of other contributions, constituting separate and independent works in themselves, are assembled into a collective whole. A work that constitutes a Collective Work will not be considered a Derivative Work (as defined below) for the purposes of this License.
b. ***"Derivative Work"*** means a work based upon the Work or upon the Work and other pre-existing works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which the Work may be recast, transformed, or adapted, except that a work that constitutes a Collective Work will not be considered a Derivative Work for the purpose of this License. For the avoidance of doubt, where the Work is a musical composition or sound recording, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered a Derivative Work for the purpose of this License.
c. ***"Licensor"*** means the individual or entity that offers the Work under the terms of this License.
d. ***"Original Author"*** means the individual or entity who created the Work.
e. ***"Work"*** means the copyrightable work of authorship offered under the terms of this License.
f. ***"You"*** means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

g. ***"License Elements"*** means the following high-level license attributes as selected by Licensor and indicated in the title of this License: Attribution, ShareAlike.

2. Fair Use Rights. Nothing in this license is intended to reduce, limit, or restrict any rights arising from fair use, first sale or other limitations on the exclusive rights of the copyright owner under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to reproduce the Work, to incorporate the Work into one or more Collective Works, and to reproduce the Work as incorporated in the Collective Works;
b. to create and reproduce Derivative Works;
c. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission the Work including as incorporated in Collective Works;
d. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission Derivative Works.
e. For the avoidance of doubt, where the work is a musical composition:
    i. ***Performance Royalties Under Blanket Licenses***. Licensor waives the exclusive right to collect, whether individually or via a performance rights society (e.g. ASCAP, BMI, SESAC), royalties for the public performance or public digital performance (e.g. webcast) of the Work.
    ii. ***Mechanical Rights and Statutory Royalties***. Licensor waives the exclusive right to collect, whether individually or via a music rights society or designated agent (e.g. Harry Fox Agency), royalties for any phonorecord You create from the Work ("cover version") and distribute, subject to the compulsory license created by 17 USC Section 115 of the US Copyright Act (or the equivalent in other jurisdictions).
f. ***Webcasting Rights and Statutory Royalties***. For the avoidance of doubt, where the Work is a sound recording, Licensor waives the exclusive right to collect, whether individually or via a performance-rights society (e.g. SoundExchange), royalties for the public digital performance (e.g. webcast) of the Work, subject to the compulsory license created by 17 USC Section 114 of the US Copyright Act (or the equivalent in other jurisdictions).

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. All rights not expressly granted by Licensor are hereby reserved.

4. Restrictions.The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may distribute, publicly display, publicly perform, or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Work with any technological measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement.

The above applies to the Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the Collective Work any credit as required by clause 4(c), as requested. If You create a Derivative Work, upon notice from any Licensor You must, to the extent practicable, remove from the Derivative Work any credit as required by clause 4(c), as requested.

b. You may distribute, publicly display, publicly perform, or publicly digitally perform a Derivative Work only under the terms of this License, a later version of this License with the same License Elements as this License, or a Creative Commons iCommons license that contains the same License Elements as this License (e.g. Attribution-ShareAlike 2.5 Japan). You must include a copy of, or the Uniform Resource Identifier for, this License or other license specified in the previous sentence with every copy or phonorecord of each Derivative Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Derivative Works that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder, and You must keep intact all notices that refer to this License and to the disclaimer of warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Derivative Work with any technological measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement. The above applies to the Derivative Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Derivative Work itself to be made subject to the terms of this License.

c. If you distribute, publicly display, publicly perform, or publicly digitally perform the Work or any Derivative Works or Collective Works, You must keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or (ii) if the Original Author and/or Licensor designate another party or parties (e.g. a sponsor institute, publishing entity, journal) for attribution in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; the title of the Work if supplied; to the extent reasonably practicable, the Uniform Resource Identifier, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and in the case of a Derivative Work, a credit identifying the use of the Work in the Derivative Work (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). Such credit may be implemented in any reasonable manner; provided, however, that in the case of a Derivative Work or Collective Work, at a minimum such credit will appear where any other comparable authorship credit appears and in a manner at least as prominent as such other comparable authorship credit.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE MATERIALS, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO

SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Derivative Works or Collective Works from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.
b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

a. Each time You distribute or publicly digitally perform the Work or a Collective Work, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.
b. Each time You distribute or publicly digitally perform a Derivative Work, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.
c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.
d. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.
e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

Creative Commons is not a party to this License, and makes no warranty whatsoever in connection with the Work. Creative Commons will not be liable to You or any party on any legal theory for any damages whatsoever, including without limitation any general, special, incidental or consequential damages arising in connection to this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons has expressly identified itself as the Licensor hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the Work is licensed under the CCPL, neither party will use the trademark "Creative Commons" or any related trademark or logo of Creative Commons without the prior written consent of Creative Commons. Any permitted use will be in compliance with Creative Commons' then-current trademark usage guidelines, as may be published on its website or otherwise made available upon request from time to time.

Creative Commons may be contacted at *http://creativecommons.org/*.