Porting Linux to x86-64

Andi Kleen
SuSE Labs
ak@suse.de

Abstract

x86-64 is a 64-bit extension for the IA32 architecture, which is supported by the next generation of AMD CPUs. New features include 64-bit pointers, a 48-bit address space, 16 general purpose 64-bit integer registers, 16 SSE (Streaming SIMD Extensions) registers, and a compatibility mode to support old binaries.

The Linux kernel port to x86-64 is based on the existing IA32 port with some extensions, including a new syscall mechanism, 64-bit support and use of interrupt stacks. It also adds a translation layer to allow execution of the system calls of old IA32 binaries.

This paper gives a short overview of the x86-64 architecture and the new x86-64 ABI and then discusses internals of the kernel port.

1 Introduction

x86-64 is a new architecture developed by AMD. It is an extension to the existing IA32 architecture. The main new features over IA32 are 64-bit pointers, a 48-bit address space, 16 64-bit integer registers, and 16 SSE2 registers. This paper describes the Linux port to this new architecture. The new 64-bit kernel is based on the existing i386 port. It is ambitious in that it tries to exploit new features, not just do a minimum port, and redesigns parts of the i386 port as necessary. The x86-64 kernel is developed by AMD and SuSE as a free software project.

2 Short overview of the x86-64 architecture

I will start with a short overview of the x86-64 extensions. This section assumes that the reader has basic knowledge about IA32, as only changes are explained. For an introduction to IA32, see [Intel].

x86-64 CPUs support new modes: legacy mode and long mode. When they are in legacy mode, they are fully IA32 compatible and should run all existing IA32 operating systems and application software unchanged. Optionally, the operating system can switch on long mode, which enables 64-bit operation. In the following only long mode is discussed. The x86-64 linux port runs in long mode only.

Certain unprivileged programs can be run in compatibility mode in a special code segment, which allows existing IA32 programs to be executed unchanged. Other programs can run in long mode and exploit all new features. The kernel and all interrupts run in long mode.

A significant new feature is support for 64-bit addresses, so that more than 4GB of memory can be addressed directly. All registers and other structures dealing with addresses have been enlarged to 64-bit. Eight new integer registers added (*R8-R16*), so that there is now a total of 16 general purpose 64-bit registers. Without address prefixing, the code usually defaults to 32-bit accesses to registers and memory, except for the stack which is always 64-bit aligned and jumps. 32-bit operations on 64-bit registers do zero extension. 64-bit immediates are only supported by the new *movabs* instruction.

A new addressing mode, RIP-relative, has been added which allows addressing of memory relative to the current program counter.

x86-64 supports the SSE2 SIMD extensions. Eight new SSE2 registers (*XMM8-XMM15*) have been added over the existing XMM0-XMM7. The x87 register stack is unchanged.

Some obsolete features of IA32 are gone in long mode. Some rarely used instructions have been removed to make space for the new 64-bit prefixes. Segmentation is mostly gone: segment bases and limits are ignored in long mode. fs and gs can be still used as kinds of address registers with some limitations and kernel support. vm86 mode and 16-bit segments are also gone. Automatic task switching is not supported anymore.

Page size stays at 4KB. Page tables have been extended to four levels to cover the full 48-bit address room of the first implementations.

For more information see the x86-64 architecture manual [AMD2000].

3 ABI

As x86-64 has more registers than IA32, and does not support direct calling of IA32 code, a new modern ABI was designed for it. The basic type sizes are similar to other 64-bit Unix environments: long and pointers are 64-bit, int stays 32-bit. All data types are aligned to their natural [1] size.

The ABI uses register arguments extensively. Up to six integer and nine 64-bit floating point arguments are passed in registers, in addition to arguments.

Structures are passed in registers where possible. Non prototyped functions have a slight penalty as the caller needs to initialize an argument count register to allow argument saving for variable argument support. Most registers are caller saved to save code space in callees.

Floating point is by default passed in SSE2 XMM registers now. This means doubles are always calculated in 64-bit unlike IA32. The x87 stack with 80-bit precision is only used for long double. The frame pointer has been replaced by an unwind table. An area 128 bytes below the stack pointer is reserved for scratch space to save more space for leaf functions.

Several code models have been defined: small, medium, large, kernel. Small is the default; while it allows full 64-bit access to the heap and the stack, all code and preinitialized data in the executable is limited to 4GB, as only RIP relative addressing is used. It is expected that most programs will run in small mode. Medium is the same as small, but allows a full 64-bit range of preinitialized data, but is slower and generates larger code. Code is limited to 4GB. Large allows unlimited [2] code and initialized data, but is even slower than medium. kernel is a special variant of the small model. It uses negative addresses to run the kernel with 32-bit displacements and the upper end of the address space. It is used by the kernel only.

So far the goal of the ABI to save code size is successful: gcc using it generates code sizes comparable to 32-bit IA32[3]

For more information on the x86-64 ABI see [Hubicka2000]

4 Compiler

A basic port of the gcc 3 compiler and binutils to x86-64 has been done by Jan Hubicka. This includes implementation of SSE2 support for gcc and full support for the long mode extensions and the new 64-bit ABI. The compiler and tool chain are stable enough for kernel compiling and system porting.

5 Kernel

The x86-64 kernel is a new Linux port. It was originally based on the existing i386 architecture code, but is now independently maintained. The following discusses the most important changes over the 32-bit i386 kernel and some interesting implementation details.

[1] On IA32 64-bit long long was not aligned to 64-bit

[2] Unlimited in the 64-bit, or rather 39-bit address space, of the first kernel

[3] Not counting the unwind table sizes.

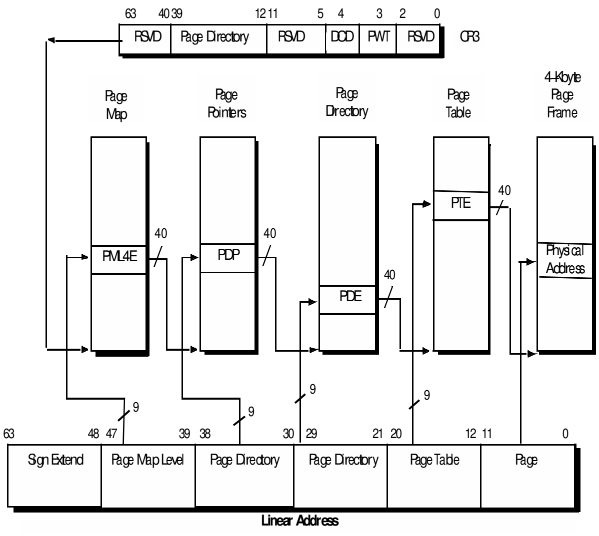

Figure 1: x86-64 pagetable

6 Memory management

x86 has a four-level page table. The portable Linux VM code currently only supports three-level page tables. The uppermost level is therefore kept private to the architecture specific code; portable code only sees three levels.

The page table setup currently supports up to 48 bits of address space, the initial Hammer implementation supports 43-bit (8TB). The current Linux port uses only three levels of the four-level page table. This causes a 511GB limit (39 bits) per user process.

The 48th bit of the full virtual space is sign extended, so there is a 'negative' part of the address space. The Linux kernel is mapped in the negative space which allows the efficient use of negative sign extended 32-bit addresses in the kernel code. The compiler has a *kernel* code model to implement this feature. This high mapping is invisible to the portable VM code which only operates on the low 39 bits of the first three-level page table branch.

The basic structure of the page table is similar to the three-level *PAE* mode in modern IA32 CPUs, with all levels being full 4K pages.

Every entry in the page tables is 64-bit wide. This is similar to the 32-bit PAE mode. To avoid races with other CPUs while updating page table entries all operations on them have to be atomic. In the 64-bit kernel this can be conveniently done using 64-bit memory operations, while i386 needs to use *CMPXCHG8*.

7 System calls

The kernel entry code was completely rewritten from the i386 implementation. For system calls it uses the *SYSCALL* and *SYSRET* instructions which run much faster than the $int0x80$ software interrupt used on for Linux/i386 syscalls. Some changes were required to make use of them which make the code more complex.

One restriction is that they are not recursive; *SYSRET* always turns on user mode. The kernel occasionally makes system calls (like $kernel\_thread()$) and these need to be handled by special entry points.

SYSCALL has a slight bootstrap problem. It doesn't do much setup for the ring0 kernel environment and the syscall kernel entry point is entered with an undefined stack pointer. To bootstrap the kernel stack itself it uses the new *SWAPGS* instruction to initialise the *GS* segment register with the PDA of the current CPU. Using the PDA the user old stack pointer is saved and the kernel stack pointer of the current process is then initialized.

Another problem was that SYSRET receives its arguments in predefined registers, which are always clobbered. This has the side effect that it is impossible to return or enter programs from signals via SYSRET, because in this case all registers need to be restored and the clobbered register would corrupt the user context. A special return path that uses the slower *IRET* command for jumping back is used in this case.

Signal handling is very similar to i386 with minor modernizations. The C Library is required now to set a restorer function that calls *sigreturn* when the signal handler has finished; stack trampolines have been removed.

Time related system calls (*gettimeofday* particularly) are called frequently by many applications and often show up as hot spots. They can be implemented in user space by using the CPU cycle counter with the help of some shared variables maintained by the kernel. To isolate this code from user space vsyscalls have been added by Andrea Arcangeli. A special code area is mapped into every user process by the kernel. The functions in there can be directly called by the user via a special offset table at a magic address, avoiding the overhead of a system call.

Vsyscalls have some problems with signal and exception handling. The x86-64 ABI requires a dwarf2 unwind table to do a backtrace in case of a crash and the kernel needs to provide an unwind table for the user mode vsyscall pages in case a signal or exception occurs while they run. This is still work in progress.

8 Processor Data Area

To solve the *SYSCALL* supervisor stack bootstrap problem described above, a data structure called the Per processor Data Area (*PDA*) is used. A pointer to the PDA is stored on bootup in a hidden register of the CPU using the *KERNEL_GS_BASE* MSR. Each time the kernel is entered from user space via exceptions, system calls or interrupts, the *SWAPGS* instruction is executed. It swaps the userland value of the *GS* register with the PDA value from the hidden register. The original contents of the *GS* register are restored on exit from the kernel.

The PDA is currently used to store information for fast syscall entry (such as the kernel stack pointer of the current task), a pointer to the current task itself, and the old user stack pointer on a system call. It also contains the per CPU stack.

It is hoped that future Linux versions will move more information into a central generic PDA structure that is used by the architecture independent kernel. As of Linux 2.4, various subsystems keep their own private arrays padded to cache lines and indexed by CPU number. Accessing such arrays is costly as the CPU number has to be first retrieved, the index computed, and the required cache line padded to avoid false sharing of old data. The PDA offers a faster alternative, at the disadvantage of being less modular because PDA data structures have to be maintained in a central include file.

9 Partial stack frame

To speed up interrupts and system calls the kernel entry code only saves registers that are actually clobbered by the C code in the portable part. Some system calls and kernel functions need to see a full register state. These include for example fork, which has to copy all registers to the child process, and exec, which has to restore all registers, signal handling, and needs to present all registers to the signal handler. Special stubs are used to save the full register set in this case.

After a fast system call entry through *SYSCALL* the kernel stack frame is partially uninitialized. Some information such as the user program pointer (*RIP*) and the user stack pointer (*RSP*) are saved in the PDA or in special registers. On other entry points (like for the i386 syscall emulation), they are on the normal stack frame on the kernel stack. To shield C code from these differences, the CPU part of stack frame is always fixed by a special stub before calling any function that looks at the kernel stack frame. After the system call returns to the emulation layer the PDA state is restored using the stack frame to handle context switches.

10 Kernel stack

On Linux, every process and kernel thread has its own kernel stack. This stack is also used for interrupts while the process runs.

Over time, the Linux memory allocator will encounter problems allocating more than two consecutive pages reliably due to memory fragmentation. Every process needs a contiguous kernel stack that should be directly mapped for efficiency. Like the i386, the x86-64 has a 4K page size. This limits the kernel stack in practice on i386 and x86-64 to 6-8K. This also helps to keep the per-thread overhead of LinuxThreads (the most common threads package under Linux) low, which uses a separate kernel stack for each thread.

On i386 ,the 6K stack available is already tight under heavy interrupt load. 64-bit code needs more stack space than 32-bit code because the stack is always 64-bit aligned, and its data structures on the stack are bigger. To avoid stack overflow for nested interrupts, the x86-64 port uses a separate per-CPU interrupt stack.

The x86-64 architecture supports interrupt stacks in the architecture. Unfortunately, this causes problems with nested interrupts, which are common in Linux. Instead of the hardware mechanism, a more flexible software stack switching scheme using an interrupt counter in the PDA is used.

For double fault and stack fault exceptions, the hardware interrupt stacks are used to handle invalid kernel stack pointers with a debugging message instead of silently rebooting the system.

11 Finding yourself

On a machine with multiple CPUs it can be quite complicated to find the current process. A global variable cannot be used, as it is CPU local information. i386 uses a special trick to solve this problem: the task structure is always stored at the bottom of the two aligned kernel stack pages [4] and can be efficiently accessed using an *AND* operation on the current stack pointer.

One disadvantage of this is that the task structures of all processes end up on the same cache sets for not-fully-associative CPU and chipset caches, because the lower 13 bits of their address is always zero. This can cause cache trashing in the scheduler for some workloads.

In the 64-bit kernel, accessing the task structure through the stack pointer doesn't work as interrupts running on the special interrupt stack also need to access it, for example, to maintain the per-process system and user time statistics

On x86-64 the current process counter is stored into the PDA which is efficiently accessed using the *GS* register. This will also allow the task struct to be moved to a separate cache coloring slab cache, working around the cache problems described above, and giving the 64-bit kernel in user context 8K of stack space instead of 6K.

This setup is still experimental. If it turns out in further tests that an 8K stack is not enough for the 64-bit user context kernel code without interrupts, then the port will have to move to a kernel stack that is not physically contiguous, which will be slower due to increased use of TLB resources, but can be made bigger without stressing the page allocator. This will also require auditing drivers to ensure they do not perform DMA from the kernel stack.

[4] Which is why i386 can use only 6K of the 8K available from the two kernel stack pages.

12 Context switch

The basic context switch of x86-64 is very similar to the i386 port except that it also saves and restore the extended R8-R15 integer registers. The extended SSE registers are handled transparently by the *FXSAVE* instruction. d drivers still need work. It is hoped that in future 32/64bit translation will be a generic feature of a linux driver to avoid a hard to maintain central translation layer.

This 64-bit conversion is currently done in an architecture-specific module for the x86-64, but it is expected to be moved into architecture-independent code in 2.5, as it is a common problem.

Legacy mode i386 applications see the full 4GB of virtual space reachable by 32-bit pointers. A 32-bit i386 kernel only gives them part of the 4GB address space (usually 3GB), as it also needs some address space of its own. Therefore, on a 64-bit kernel, even 32-bit applications can use more address space.

13 Status

The kernel, compiler and tool chain work. The kernel boots and works on the simulator, which is used for the porting of userland code and for running programs.

14 Availability

All the code discussed in this paper can be downloaded from *http://www.x86-64.org*. The gcc port will be part of gcc 3.1. The x86-64 toolchain is part of the standard GNU binutils sources. Gdb and glibc ports are worked on and they are available in the public CVS repository at *cvs.x86-64.org*. The kernel code is currently maintained in CVS there also and will be eventually merged into the official kernel source.

15 Acknowledgements

The author thanks Karsten Keil, James Morris and Matthew Wilcox for review of this paper. The x86-64 kernel port was done by the author, Karsten Keil, Pavel Machek and Andrea Arcangeli. The x86-64 gcc and binutils port was done by Jan Hubicka. The glibc port was done by Andreas Jaeger.

References

[Hubicka2000] Hubicka Jan, Jaeger Andreas, Mitchell Mark. *System V Application Binary Interface*
x86-64 Architecture Processor Supplement Living document. http://www.x86-64.org/

[AMD2000] AMD *The AMD x86-64 architecture programmers overview* http://www.x86-64.org/

[Intel] Intel *Intel architecture software developers manual* http://developer.intel.com