Porting Linux to x86-64

Andi Kleen
SuSE Labs
ak@suse.de

Abstract

x86-64 is a 64bit extension for the IA32 architecture. The next generation AMD CPUs will support it. Among others it adds to IA32 64bit pointers, an 48bit address space, 16 general purpose 64 bit integer and 16 SSE 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 like a new syscall mechanism, 64bit support and use of interrupt stacks. It also adds a translation layer to still execute 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 64bit addresses, 16 64bit integer register and 16 SSE2 registers. This paper describes the Linux port to this new architecture. The new 64bit kernel is based on the existing i386 port. It is ambitious in as 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 AND and SuSE as a free software project.

2 Short overview over 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 about IA32 see [Intel].

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

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

The main new feature is support for 64bit addresses, so that more than 4GB of memory can be directly addressed.All registers and other structures dealing with addresses have been enlarged to 64bit. There have been 8 new integer registers added (*R8-R16*), so that there is a total of 16 general purpose 64bit registers now. Without address prefix the code usually defaults to 32bit accesses to registers and memory, except for the stack which is always 64bit aligned and jumps. 32bit operations on 64bit registers do zero extension. 64bit immediates are only supported by the new *movabs* instruction.

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

x86-64 supports the SSE2 SIMD extensions. 8 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 64bit prefixes. Segmentation is mostly gone: segment bases and limits are ignored in long mode. fs and gs can be still used as kind of address registers with some limitations and kernel support. vm86 mode and 16bit segments are also gone. Automatic task switching is not supported anymore.

Page size stays at 4KB. Page tables have been extended to 4 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 it was possible to design a new modern ABI for it. The basic type sizes are similar to other 64bit Unix environments: long and pointers are 64bit, int stays 32bit. All data types are aligned to their natural [1] size.

The ABI uses register arguments extensively. Upto 6 integer and 9 64bit floating point arguments are passed in registers, in addition to argument

Structures are passed in registers as 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 double are always calculated in 64bit unlike IA32. The x87 stack with 80bit precision is only used for long double. The frame pointer has been replaced by an unwind table. 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 64bit 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 64bit 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 32bit 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 32bit 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 64bit 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 32bit i386 kernel and some interesting implementation details.

6 Memory management

x86 has a 4 level page table. The portable Linux VM code currently only supports 3 level page tables. The uppermost level is therefore kept private the

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

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

[3] Not counting the unwind table sizes.

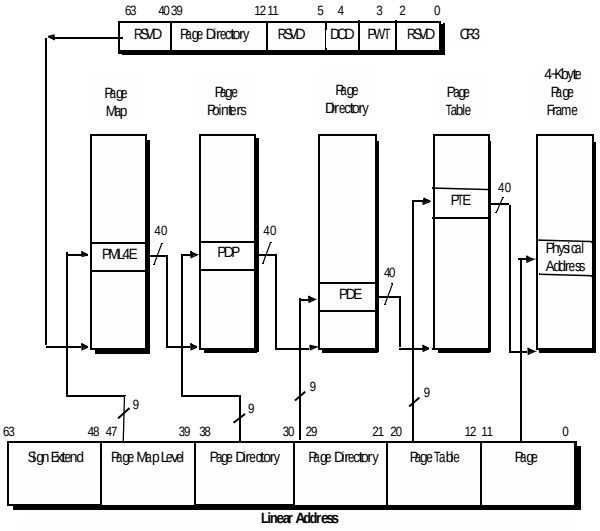

Figure 1: x86-64 pagetable

architecture specific code; portable code only sees three levels.

The page table setup currently supports upto 48bits of address space, the initial Hammer implementation supports 43bit (8TB). TB). The current linux port uses only 3 levels of the 4 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 room. In the linux port this negative room is reserved for the kernel. The kernel is mapped on top of the negative part while allows the efficient use of negative sign extended 32bit addresses in the kernel code. For that the *kernel* code model of the compiler is used. This high mapping is invisible to the portable VM code which only operates on the low 39bits of the first 3 level page table branch.

The basic structure of the page table is similar to the 3level *PAE* mode in modern IA32 CPUs, with all levels being full 4K pages.

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

7 System calls

The kernel entry code was completely rewritten compared to i386. 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. They required some changes to use and made the code more complicated though.

One restriction is that they are not recursive; *SYSRET* always turns on user mode. In kernel system calls like *kernel_thread*() needed to be handled by special entry points, complicating the code.

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 very often called in many applications. They can be also 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 Arcangelli. A special memory area including some code and some variables is mapped into every user process by the kernel. It can be directly called by the user via a special offset table on a magic address, allowing very fast time access. Problems of this approach is the signal and excep-

tion handling and the handling of the unwind table in case an signal occurs while a vsyscall runs.

8 PDA

To solve the *SYSCALL* supervisor stack bootstrap problem described above a data structure called the Per Processor 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 and stays there while the kernel runs.

The PDA is currently used to store information for fast syscall entry, like 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 some other information.

It is hoped that future linux version 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 and then the index computed and the required cache line padding to avoid false sharing of data wastes memory. 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 files.

9 Partial stack frame

To speed up interrupts and system calls the entry code only saves registers that are actually clobbered by the C code. 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, exec, has to restore all registers, signal handling, which needs to present all registers to the signal handler and some others. For these functions special stubs are used to save the full register set.

After a fast system call entry through *SYSCALL* the kernel stack frame is partly uninitialized. Some information like 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. Afterward the function ran the PDA is restored from the stack.

10 Kernel stack

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

The linux memory allocator has problems to allocate more than two consecutive pages reliably after some system runtime due to memory fragmentations. Every process needs a continuous kernel stack that should be directly mapped for efficiency. x86-64 has like i386 a 4K page size. This limits the kernel stack in practice on i386 and x86-64 to 6-8K. This also helps keeping the per thread overhead of the most common threads package under Linux, LinuxThreads, low which uses an own kernel stack per thread.

On i386 the 6K stack available are already tight under heavy interrupt load. 64bit code needs more stack space than 32bit code because the stack is always 64bit 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. This unfortunately has some 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 to handle invalid kernel stack pointers with a oops.

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 64bit 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 to move the task struct to a separate cache coloring slab cache, working around the cache problems described above and giving the 64bit kernel in user contex 8K instead of 6K stack space.

This setup is still experimental. If it turns out in further tests that 8K stack is not enough for the 64bit user context kernel code without interrupts then the port will have to move to a slower non continuous kernel stack that is remapped virtually continuous through the MMU but can be bigger.

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.

A special case are the `FS` and `GS` registers. They act kind like two additional address registers which can be only set by the privileged *WRMSR* instruction or a change of the *LDT* of the current process. They are for example used by the LinuxThreads package to store thread local data. Doing the WRMSR on every context switch is relatively costly so the scheduler tries to avoid it if possible. For this a similar technique to the lazy FPU context switch is used. When a process calls the special system call to set 64bit GS or FS this switching is turned on, otherwise it is lazily not done.

The process specific values in the *PDA* like the current process stack or the user space stack pointer stored there are also context switched.

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

13 IA32 emulation

The 64bit kernel supports IA32 binaries. These run in a special code segment in the CPU's compatibility mode. On system calls and interrupts long mode is always used.

The i386 system calls use the 32bit i386 ABI, which is different from the 64bit x86-64 ABI. All indirect structures containing long and pointers have different sizes and offsets, the arguments are passed in different registers and the system call numbers are different. To make old 32bit binaries run 64bit kernels have a special translation layer that changes all parameters to the 64bit ABI and then calls the 64bit kernel services.

Pointers and integer passed in registers can be used directly, because the x86-64 architecture always extends 32bit registers to 64bit. Structures passed in or out through a pointer often need to be converted. This is implemented based on previous work for the sparc64 and IA64. Most system calls and the important ioctls are converted, but some subsystems still need work.

It is currently done in an architecture specific module for the x86-64, but it is expected that the 64bit conversion will be moved to 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 32bit pointers. A 32bit i386 kernel only gives them part of the 4GB address space (usually 2GB) as it also needs some address

space of its own. On a 64bit kernel therefore even 32bit applications can use more address space.

14 Status

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

15 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.

16 References

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