Setting Up Long Mode

From OSDev Wiki
(Redirected from User:Stephanvanschaik/Setting Up Long Mode)

Contents

Overview

- Covering long mode.
- How to detect long mode (Recommended read: CPUID).
- How to set up paging for long mode (Recommended reads: Setting up Paging and Setting up Paging using PAE).
- How to enter long mode.
- How to set up the GDT for long mode (Recommended read: GDT).

Introduction

What is long mode and why set it up? Since the introduction of the x86-64 processors (AMD64, Intel 64 (a.k.a. EM64T), VIA Nano) a new mode has been introduced as well, which is called long mode. Long mode basically consists out of two sub modes which are the actual 64-bit mode and compatibility mode (32-bit, usually referred to as IA32e in the AMD64 manuals). What we are interested in is simply the 64-bit mode as this mode provides a lot of new features such as: registers being extended to 64-bit (rax, rcx, rdx, rbx, rsp, rbp, rip, etc.) and the introduction of eight new general-purpose registers (r8 - r15), but also the introduction of eight new multimedia registers (xmm8 - xmm15). 64-bit mode is basically a new world as it is almost completely void of the segmentation that was used on the 8086-processors and the GDT, the IDT, paging, etc. are also kind of different compared to the old 32-bit mode (a.k.a. protected mode).

Detecting the Presence of Long Mode

There are only three processor vendors so far who have made processors that are capable of entering and using long mode, they're: AMD, Intel and VIA. Basically Intel tried to get 64-bit processors on the market with EM64T, but failed to do so and now they use AMD's x86-64 architecture instead, which means using 64-bit on an Intel processor is (almost) identical to using 64-bit on an AMD processor (and VIA should be identical as well). We can detect the presence of long mode by using the CPUID-instruction.

Detection of CPUID

Basically, detecting whether CPUID is supported by your processor is covered here, but we will show how to do it here. CPUID is supported when the ID-bit in the FLAGS-register can be flipped. So let's try that, then:

```
; Check if CPUID is supported by attempting to flip the ID bit (bit 21
; the FLAGS register. If we can flip it, CPUID is available.

; Copy FLAGS in to EAX via stack
pushfd
pop eax

; Copy to ECX as well for comparing later on
mov ecx, eax

; Flip the ID bit
xor eax, 1 << 21

; Copy EAX to FLAGS via the stack
push eax
popfd

; Copy FLAGS back to EAX (with the flipped bit if CPUID is supported)
pushfd
pop eax

; Restore FLAGS from the old version stored in ECX (i.e. flipping the
; back if it was ever flipped).
push ecx
popfd

; Compare EAX and ECX. If they are equal then that means the bit wasn'
; flipped, and CPUID isn't supported.
xor eax, ecx
jz .NoCPUID
ret
```

x86 or x86-64

Now that CPUID is available we have to check whether long mode can be used or not. Long mode can only be detected using the extended functions of CPUID (> 0x80000000), so we have to check if the function that determines whether long mode is available or not is actually available:

```
mov eax, 0x80000000    ; Set the A-register to 0x80000000.
cpuid                  ; CPU identification.
cmp eax, 0x80000001    ; Compare the A-register with 0x80000001.
jb .NoLongMode         ; It is less, there is no long mode.
```

Now that we know that extended function is available we can use it to detect long mode:

```
mov eax, 0x80000001    ; Set the A-register to 0x80000001.
cpuid                  ; CPU identification.
test edx, 1 << 29      ; Test if the LM-bit, which is bit 29, is set i
jz .NoLongMode         ; They aren't, there is no long mode.
```

Now that we know if long mode is actually supported by the processor, we can actually use it.

Entering Long Mode

Entering long mode can be both done from real mode and protected mode, however only protected mode is covered in the Intel and AMD64 manuals. Early AMD documentation explains this process works from real mode as well.

Before anything, it is recommended that you enable the A20 Line; otherwise only odd MiBs can be accessed.

Setting up the Paging

Before we actually cover up the new paging used in x86-64 we should disable the old paging first (you can skip this if you never set up paging in protected mode) as this is required.

```
mov eax, cr0                                   ; Set the A-register to
and eax, 01111111111111111111111111111111b     ; Clear the PG-bit, whi
mov cr0, eax                                   ; Set control register
```

Now that paging is disabled, we can actually take a look at how paging is set up in x86-64 (It's recommended to read Chapter 5.3 of the AMD64 Architecture Programmer's Manual, Volume 2). First of all, long mode uses PAE paging and therefore you have the page-directory pointer table (PDPT), the page-directory table (PDT) and the page table (PT). There's also another table which now forms the root (instead of the PDPT or the PDT) and that is page-map level-4 table (PML4T). In protected mode a page table entry was only four bytes long, so you had 1024 entries per table. In long mode, however, you only have 512 entries per table as each entry is eight bytes long. This means that one entry in a PT can address 4kB, one entry in a PDT can address 2MB, one entry in a PDPT can address 1GB and one entry in a PML4T can address 512GB. This means that only 256TB can be addressed. The way these tables work is that each entry in the PML4T point to a PDPT, each entry in a PDPT to a PDT and each entry in a PDT to a PT. Each entry in a PT then points to the physical address, that is, if it is marked as present. So how does the MMU/processor know which physical address should be used with which virtual address? Basically each table has 512 entries ranging from 0 to 511. These entries each refer to a specific domain of memory. Like index 0 of the PML4T refers to the first 512GB in virtual memory, index 1 refers to the next 512GB and so on. The same applies to the PDPT, PDT and the PT (except with smaller sizes; 1GB, 2MB and 4kB as seen above). The last gigabyte of virtual memory would be described in the table referred to by 511th index of the PDPT which is referred to by the 511th index of the PML4T or in psuedo-C:

```
pagedir_t* PDT = PML4[511]->PDPT[511];
```

Basically, what pages you want to set up and how you want them to be set up is up to you, but I'd identity map the first megabyte and then map some high memory to the memory after the first megabyte, however, this may be pretty difficult to set up first. So let's identity map the first two megabytes. We'll set up four tables at 0x1000 (assuming that this is free to use): a PML4T, a PDPT, a PDT and a PT. Basically we want to identity map the first two megabytes so:

- PML4T[0] -> PDPT.
- PDPT[0] -> PDT.
- PDT[0] -> PT.
- PT -> 0x00000000 - 0x00200000.

First we will clear the tables:

```
    mov edi, 0x1000    ; Set the destination index to 0x1000.
    mov cr3, edi       ; Set control register 3 to the destination index.
    xor eax, eax       ; Nullify the A-register.
    mov ecx, 4096      ; Set the C-register to 4096.
    rep stosd          ; Clear the memory.
    mov edi, cr3       ; Set the destination index to control register 3.
```

Now that the page are clear we're going to set up the tables, the page tables are going to be located at these addresses:

- PML4T - 0x1000.
- PDPT - 0x2000.
- PDT - 0x3000.
- PT - 0x4000.

So lets make PML4T[0] point to the PDPT and so on:

```
    mov DWORD [edi], 0x2003      ; Set the uint32_t at the destination inc
    add edi, 0x1000              ; Add 0x1000 to the destination index.
    mov DWORD [edi], 0x3003      ; Set the uint32_t at the destination inc
    add edi, 0x1000              ; Add 0x1000 to the destination index.
    mov DWORD [edi], 0x4003      ; Set the uint32_t at the destination inc
    add edi, 0x1000              ; Add 0x1000 to the destination index.
```

If you haven't noticed already, I used a three. This simply means that the first two bits should be set. These bits indicate that the page is present and that it is readable as well as writable. Now all that's left to do is identity map the first two megabytes:

```
    mov ebx, 0x00000003          ; Set the B-register to 0x00000003.
    mov ecx, 512                 ; Set the C-register to 512.

.SetEntry:
```

```
    mov DWORD [edi], ebx          ; Set the uint32_t at the destination inc
    add ebx, 0x1000               ; Add 0x1000 to the B-register.
    add edi, 8                    ; Add eight to the destination index.
    loop .SetEntry                ; Set the next entry.
```

Now we should enable PAE-paging by setting the PAE-bit in the fourth control register:

```
    mov eax, cr4                  ; Set the A-register to control register
    or eax, 1 << 5                ; Set the PAE-bit, which is the 6th bit (
    mov cr4, eax                  ; Set control register 4 to the A-registe
```

Now paging is set up, but it isn't enabled yet.

Future of x86-64 - the PML5

While not yet in a production processor, or indeed the reference manual, Intel have already released a white paper PML5 White Paper (https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf) about 5 level paging. These processors will support a 128PiB address space, and are furthermore expected to support up to 4PiB of physical memory (far above the current 256 TiB/64 TiB limits). Support for this is detected as follows:

```
    mov eax, 0x7                  ; You might want to check for page 7 firs
    xor ecx, ecx
    cpuid
    test ecx, (1<<16)
    jnz .5_level_paging
```

The paging structures are identical to the 4 level versions, there's just an added layer of indirection. Your recursive address (if you use recursive mappings) will change. 5 level paging is enabled if CR4.LA57=1, and EFER.LMA=1.

```
BITS 32
mov eax, cr4
or eax, (1<<12) ;CR4.LA57
mov cr4, eax
```

Note that attempting to set CR4.LA57 while EFER.LMA=1 causes a #GP general protection fault. You therefore need to drop into protected mode or set up 5 level paging before entering long mode in the first place.

The Switch from Real Mode

There's not much left to do. We should set the long mode bit in the EFER MSR and then we should enable paging and protected mode and then we are in compatibility mode (which is part of long mode).

So we first set the LM-bit:

```
    mov ecx, 0xC0000080          ; Set the C-register to 0xC0000080, which
    rdmsr                        ; Read from the model-specific register.
    or eax, 1 << 8               ; Set the LM-bit which is the 9th bit (bi
    wrmsr                        ; Write to the model-specific register.
```

Enabling paging and protected mode:

```
    mov eax, cr0                 ; Set the A-register to control register
    or eax, 1 << 31 | 1 << 0     ; Set the PG-bit, which is the 31nd bit,
    mov cr0, eax                 ; Set control register 0 to the A-registe
```

Now we're in compatibility mode.

The Switch from Protected Mode

There's not much left to do. We should set the long mode bit in the EFER MSR and then we should enable paging and then we are in compatibility mode (which is part of long mode).

So we first set the LM-bit:

```
    mov ecx, 0xC0000080          ; Set the C-register to 0xC0000080, which
    rdmsr                        ; Read from the model-specific register.
    or eax, 1 << 8               ; Set the LM-bit which is the 9th bit (bi
    wrmsr                        ; Write to the model-specific register.
```

Enabling paging:

```
    mov eax, cr0                 ; Set the A-register to control register
    or eax, 1 << 31              ; Set the PG-bit, which is the 32nd bit (
    mov cr0, eax                 ; Set control register 0 to the A-registe
```

Now we're in compatibility mode.

Entering the 64-bit Submode

Now that we're in long mode, there's one issue left: we are in the 32-bit compatibility submode and we actually wanted to enter 64-bit long mode. This isn't a hard thing to do. We should load just load a GDT with the 64-bit flags set in the code and data selectors.

Our GDT (see chapter 4.8.1 and 4.8.2 of the AMD64 Architecture Programmer's Manual Volume 2) should look like this:

```
GDT64:                           ; Global Descriptor Table (64-bit).
    .Null: equ $ - GDT64         ; The null descriptor.
```

```
    dw 0xFFFF                   ; Limit (low).
    dw 0                        ; Base (low).
    db 0                        ; Base (middle)
    db 0                        ; Access.
    db 1                        ; Granularity.
    db 0                        ; Base (high).
    .Code: equ $ - GDT64        ; The code descriptor.
    dw 0                        ; Limit (low).
    dw 0                        ; Base (low).
    db 0                        ; Base (middle)
    db 10011010b                ; Access (exec/read).
    db 10101111b                ; Granularity, 64 bits flag, limit19:16.
    db 0                        ; Base (high).
    .Data: equ $ - GDT64        ; The data descriptor.
    dw 0                        ; Limit (low).
    dw 0                        ; Base (low).
    db 0                        ; Base (middle)
    db 10010010b                ; Access (read/write).
    db 00000000b                ; Granularity.
    db 0                        ; Base (high).
    .Pointer:                   ; The GDT-pointer.
    dw $ - GDT64 - 1            ; Limit.
    dq GDT64                    ; Base.
```

Notice that we set a 4gb limit for code. This is needed because the processor will make a last limit check before the jump, and having a limit of 0 will cause a #GP (tested in bochs). After that, the limit will be ignored. Now the only thing left to do is load it and make the jump to 64-bit:

```
    lgdt [GDT64.Pointer]        ; Load the 64-bit global descriptor table
    jmp GDT64.Code:Realm64      ; Set the code segment and enter 64-bit l
```

Sample

Now that we're in 64-bit, we want to do something that is actually 64-bit. In this sample I'm just going with an ordinary clear the screen:

```
; Use 64-bit.
[BITS 64]

Realm64:
    cli                         ; Clear the interrupt flag.
    mov ax, GDT64.Data          ; Set the A-register to the data descrip
    mov ds, ax                  ; Set the data segment to the A-register
    mov es, ax                  ; Set the extra segment to the A-registe
    mov fs, ax                  ; Set the F-segment to the A-register.
    mov gs, ax                  ; Set the G-segment to the A-register.
    mov ss, ax                  ; Set the stack segment to the A-registe
    mov edi, 0xB8000            ; Set the destination index to 0xB8000.
```

```
    mov rax, 0x1F201F201F201F20   ; Set the A-register to 0x1F201F201F201F
    mov ecx, 500                  ; Set the C-register to 500.
    rep stosq                     ; Clear the screen.
    hlt                           ; Halt the processor.
```

It is very important that you don't enable the interrupts (unless you have set up a 64-bit IDT of course).

See also

Articles

- Intel EM64T
- X86-64
- Creating a 64-bit kernel
- Entering Long Mode Directly

Threads

- Wrote a tutorial covering long mode (http://forum.osdev.org/viewtopic.php?f=8&t=21601) about this article.
- Loading a higher-half kernel (http://forum.osdev.org/viewtopic.php?f=1&t=21748) on setting up long mode for a higher-half kernel.
- Setting up the stack after the switch to long mode (http://forum.osdev.org/viewtopic.php?f=1&t=21772) on the stack segment note.

Wikipedia

- AMD64
- 64-bit