Creating a 64-bit kernel

From OSDev Wiki

Difficulty level

Medium

This page is a work in progress and may thus be incomplete. Its content may be changed in the near future.

The factual accuracy of this article or section is disputed.
Please see the relevant discussion on the talk page.

Kernel Designs

Models

Monolithic Kernel
Microkernel
Hybrid Kernel
Exokernel
Nano/Picokernel
Cache Kernel
Virtualizing Kernel
Megalithic Kernel

Other Concepts

Modular Kernel
Higher Half Kernel
64-bit Kernel

Contents

Prerequisites

Make sure that you have the following done before proceeding:

- Have completed Bare Bones to make a 32-bit operating system. This article currently has trouble and will cause newcomers grief, Bare Bones gives you a well-tested 32-bit environment, you can switch to 64-bit when you get more experiences. The fact that paging is always enabled on 64-bit systems will certainly make this hard for the unexperienced.
- Have built a cross-compiler for the x86_64-elf target.
- Read up on long mode and how to initialize/use it.
- Decide now on how to load your kernel - your own bootloader, GRUB (with separate loader executable), or GRUB2 (elf64 + 32-bit bootstrap code).

The Main Kernel

The kernel should run in a uniform environment. Let's make this simple for now...

kernel.c

```
void kernel_main(void)
{
    /* What goes here is up to you */
}
```

Compiling

Compile each source file like any piece of C code, just remember to use the cross-compiler and the proper options. Linking will be done later...

```
x86_64-elf-gcc -ffreestanding -mcmodel=large -mno-red-zone -mno-
```

The -mcmodel=large argument enables us to run the kernel at any 64-bit virtual memory address we want. In fact, using the 'large' code model is discouraged due to its inefficiency, but it can be fine as a start. Check the SysV AMD64 ABI document for extra details.

You will need to instruct GCC not to use the the AMD64 ABI 128-byte 'red zone', which resides below the stack pointer, or your kernel will be interrupt unsafe. Check this thread (http://forum.osdev.org/viewtopic.php?t=21720) on the forums for extra context.

We disable SSE floating point ops. They need special %cr0 and %cr4 setup that we're not ready for. Otherwise, several #UD and #NM exceptions will be triggered.

Linking

The kernel will be linked as an x86_64 executable, to run at a virtual higher-half address. We use a linker script:

link.ld

```
ENTRY(_start)
```

```
SECTIONS
{
    . = KERNEL_VMA;

    .text : AT(ADDR(.text) - KERNEL_VMA)
    {
        _code = .;
        *(.text)
        *(.rodata*)
        . = ALIGN(4096);
    }

   .data : AT(ADDR(.data) - KERNEL_VMA)
   {
        _data = .;
        *(.data)
        . = ALIGN(4096);
   }

   .eh_frame : AT(ADDR(.eh_frame) - KERNEL_VMA)
   {
       _ehframe = .;
       *(.eh_frame)
        . = ALIGN(4096);
   }

   .bss : AT(ADDR(.bss) - KERNEL_VMA)
   {
       _bss = .;
       *(.bss)

       /*
        * You usually need to include generated COMMON symbols
        * under kernel BSS section or use gcc's -fno-common
        */

        *(COMMON)
       . = ALIGN(4096);
   }

   _end = .;

   /DISCARD/ :
   {
        *(.comment)
   }
}
```

Feel free to edit this linker script to suit your needs. Set ENTRY(...) to your entry function, and KERNEL_VMA to your base virtual address.

You can link the kernel like this:

```
x86_64-elf-gcc -ffreestanding <other options> -T <linker script>
```

Note: Obviously there is no bootstrap assembly yet, which is the hard part of starting out, and you can't link without it.

Loading

Before you can actually use your kernel, you need to deal with the hard job of loading it. Here are your three options:

With your own boot loader

This method is the simplest (since you write all the code), though it requires the most work.

I won't give any code, but the basic outline is:

- Set up a stable environment
- Do Protected Mode readying stuff (GDT, IDT, A20 gate, etc.)
- Enter Protected Mode (or skip this step and enter long mode directly)
- Parse kernel ELF headers (if kernel is separate from executable)
- Set up Long Mode readying stuff (PAE, PML4, etc.) - Remember to set up the higher-half addressing!
- Enter Long Mode by far jump to the kernel entry point in (virtual) memory

With a separate loader

Note: The advise in this section is bit questionable in its current form.

This requires the use of GRUB or another multiboot1-compliant loader. This may be the most error free of the three.

A quick rundown:

- Set up a stable environment
- Read the multiboot information struct to see where GRUB loaded your kernel (look at the module section)
- Parse kernel ELF headers
- Set up Long Mode readying stuff (PAE, PML4, etc.) - Remember to set up the higher-half addressing!
- Enter Long Mode by far jump to the kernel entry point

Note that this code has to be stored in a elf32 format and must contain the multiboot1-header.

Also remember to set the text section to start at 0x100000 (-Ttext 0x100000) when linking your loader.

Set up GRUB to boot your loader as a kernel in its own right, and your actual kernel as a module. Something like this in menu.lst:

```
title My Kernel
kernel --type=multiboot <loader executable>
module <kernel executable>
```

With a 32-bit bootstrap in your kernel

Note: The advise in this section is bit questionable in its current form.

This requires the use of any ELF64-compatible loader that loads into protected-mode (GRUB2, or patched GRUB Legacy). This may be the simplest in the long run, but is hell to set up (well, it was for me - but I saved you some work ;).

Note that GRUB2, which implements Multiboot 2 (http://download.savannah.gnu.org/releases-noredirect/grub/phcoder/multiboot.pdf) , does not support switching into long mode.

First, create an assembly file like the following, which will set up virtual addressing and long mode:

bootstrap.S

```
.section .text
.code32

multiboot_header:
    (only needed if you're using multiboot)

bootstrap:
    (32-bit to 64-bit code goes here)
    (jump to 64-bit code)
```

Then, add the following to your original linker file:

link.ld

```
...
ENTRY(bootstrap)
...
SECTIONS
{
    . = KERNEL_LMA;

    .bootstrap :
    {
        <path of bootstrap object> (.text)
    }

    . += KERNEL_VMA;

    .text : AT(ADDR(.text) - KERNEL_VMA)
    {
        _code = .;
        *(EXCLUDE_FILE(*<path of bootstrap object>) .text)
        *(.rodata*)
        . = ALIGN(4096);
    }
...
```

The above edits allow the linker to link the bootstrap code with physical addressing, as virtual addressing is set up by the bootstrap. Note that in this case, KERNEL_VMA will be equivalent to 0x0, meaning that text would have a virtual address at KERNEL_LMA + KERNEL_VMA instead of just at KERNEL_VMA. Change '+=' to '=' and your bootstrap code if you do not want this behaviour.

Compile and link as usual, just remember to compile the bootstrap code as well!

Set up GRUB2 to boot your kernel (depends on your bootloader) with grub.cfg:

```
menuentry "My Kernel" {
    multiboot <kernel executable>
}
```

Possible Problems

You may experience some problems. Fix them **immediately** or risk spending a lot of time debugging later...

My kernel is way too big!

Try each of the following, in order:

- Try linking your kernel with the option "-z max-page-size=0x1000" to force the linker to use 4kb pages.
- Make sure you're compiling with the -nostdlib option (equivalent to passing the both -nodefaultlibs and -nostartfiles options).
- You can try changing the OUTPUT_FORMAT to elf64-little.
- Try cross-compiling the **latest** version of binutils and gcc.

Kernel Virtual Memory

(This section is based on notes by Travis Geiselbrecht (geist) at the osdev IRC channel)

Long mode provides essentially an infinite amount of address space. An interesting design decision is how to map and use the kernel address space. Linux approaches the problem by permanently mapping the -2GB virtual region 0xffffffff80000000 -> 0xffffffffffffffff to physical address 0x0 upwards. Kernel data structures, which are usually allocated by kmalloc() and the slab allocator, reside above the 0xffffffff80000000 virtual base and are allocate from the physical 0 -> 2GB zone. This necessitates the ability of 'zoning' the page allocator, asking the page allocator to returning a page frame from a specific region, and only from that region. If a physical address above 2GB needs to accessed, the kernel temporarily map it to its space in a temporary mapping space below the virtual addresses base. The Linux approach provides the advantage of not having to modify the page tables much which means less TLB shootdowns on an SMP system.

Another approach is to treat the kernel address space as any other address space and dynamically map its regions. This provides the advantage of simplifying the page allocator by avoiding the need of physical memory 'zones': all physical RAM is available for any part of the kernel. An example of this approach is mapping the kernel to 0xffffff800000000 as usual. Below that virtual address you put a large mapping for the entire physical address space, and use the virtual 0xffffff800000000 -> 0xffffffffffffffff region above kernel memory area as a temporary mappings space.

See Also

Articles

- X86-64

Forum Threads

- Creating a 64-bit Kernel Tutorial (http://forum.osdev.org/viewtopic.php?f=8&t=16779) about this article
- Linker-script writers beware: COMMON Symbols (http://forum.osdev.org/viewtopic.php?p=170634) on the obscure 'COMMON' symbols and their effect on BSS
- Long-mode Kernels and the AMD64 ABI 'Red Zone' (http://forum.osdev.org/viewtopic.php?t=21720) on the 'red zone' and its major effect on interrupt handling
- Leaving long mode (http://forum.osdev.org/viewtopic.php?f=1&p=136701) to protected mode

- Switching from long mode to compatibility mode (http://forum.osdev.org/viewtopic.php?f=1&t=17213)

Categories: Level 2 Tutorials | In Progress | Disputed Pages | Tutorials | X86-64
