Bare Bones

From OSDev Wiki

In this tutorial we will write a simple kernel for x86 and boot it. This is the first step creating your own operating system. This will serve as an example of how to create a minimal system, but not as an example of how to properly structure your project. These instruction are community reviewed and follow the current recommendations for good reasons. Beware of the many other tutorials available online as they do not follow modern advise and were written by the inexperienced.

Difficulty level

🟩⬜⬜⬜
Beginner

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

WAIT! Have you read Getting Started, Beginner Mistakes, and some of the related OS theory?

Contents

Preface

You are about to begin development of a new operating system. Perhaps one day, your new operating system can be developed under itself. This is a process known as bootstrapping or going self-hosted. However, that is way into the future. Today, we simply need to set up a system that can compile your operating system from an existing operating system. This is a process known as cross-compiling and this makes the first step in operating systems development.

To make starting an OS easy, we will be using a lot of existing parts, GRUB will be the bootloader, and the kernel will be in ELF format. GRUB (a Multiboot compliant boot loader) puts the system in to the correct state for your kernel to start executing. This includes enabling the A20 line (to give you access to all available memory addresses) and putting the system in to 32-bit Protected Mode, giving you access to a theoretical 4GiB of memory. We will not use a flat binary but a kernel in ELF format, so that we have a lot of control to tell GRUB where to load which part in memory.

This article assumes you are using a Unix-like operating system such as Linux which supports operating systems development well. Windows users should be able to complete it from a MinGW or Cygwin environment.

Building a Cross-Compiler

> Main article: GCC Cross-Compiler, Why do I need a Cross Compiler?

The first thing you should do is set up a GCC Cross-Compiler for **i686-elf**. You have not yet modified your compiler to know about the existence of your operating system, so we use a generic target called i686-elf, which provides you with a toolchain targeting the System V ABI. This setting is well tested and understood by the osdev community and will allow you to easily set up a bootable kernel using GRUB and Multiboot. (Note that if you are already using an ELF platform, such as Linux, you may already have a GCC that produces ELF programs. This is not suitable for osdev work, as this compiler will produce programs for Linux, and your operating system is **not** Linux, no matter how similar it is. You will certainly run into trouble if you don't use a cross-compiler.)

You will not be able to correctly compile your operating system without a cross-compiler.

You will not be able to correctly complete this tutorial with a x86_64-elf cross-compiler, as GRUB is only able to load 32-bit multiboot kernels. If this is your first operating system project, you should do a 32-bit kernel first. If you use a x86_64 compiler instead and somehow bypass the later sanity check, you will end up with a kernel that GRUB doesn't know how to boot.

Overview

By now, you should have set up your cross-compiler for i686-elf (as described above). This tutorial provides a minimal solution for creating an operating system for x86. It doesn't serve as a recommend skeleton for project structure, but rather as an example of a minimal kernel. In this simple case, we just need three input files:

- boot.s - kernel entry point that sets up the processor environment
- kernel.c - your actual kernel routines
- linker.ld - for linking the above files

Booting the Operating System

To start the operating system, an existing piece of software will be needed to load it. This is called the bootloader and in this tutorial we will be using GRUB. Writing your own bootloader is an advanced subject, but it is commonly done. We'll later configure the bootloader, but the operating system needs to handle when the bootloader passes control to it. The kernel is passed a very minimal environment, in which the stack is not set up yet, virtual memory is not yet enabled, hardware is not initialized, and so on.

The first task we will deal with is how the bootloader starts the kernel. We are lucky because there exists a Multiboot Standard, which describes an easy interface between the bootloader and the operating system kernel. It works by putting a few magic values in some global variables (known as a multiboot header), which is searched for by the bootloader. When it sees these values, it recognizes the kernel as multiboot compatible and it knows how to load us, and it can even forward us important information such as memory maps, but we won't need that yet.

Since there is no stack yet and we need to make sure the global variables are set correctly, we will do this in assembly.

Bootstrap Assembly

> Alternatively, you can use NASM as your assembler.

We will now create a file called boot.s and discuss its contents. In this example, we are using the GNU assembler, which is part of the cross-compiler toolchain you built earlier. This assembler integrates very well with the rest of the GNU toolchain.

The very most important piece to create is the multiboot header, as it must be very early in the kernel binary, or the bootloader will fail to recognize us.

```
# Declare constants used for creating a multiboot header.
.set ALIGN,    1<<0             # align loaded modules on page boundaries
.set MEMINFO,  1<<1             # provide memory map
.set FLAGS,    ALIGN | MEMINFO  # this is the Multiboot 'flag' field
.set MAGIC,    0x1BADB002       # 'magic number' lets bootloader find the header
.set CHECKSUM, -(MAGIC + FLAGS) # checksum of above, to prove we are multiboot

# Declare a header as in the Multiboot Standard. We put this into a special
# section so we can force the header to be in the start of the final program.
# You don't need to understand all these details as it is just magic values that
# is documented in the multiboot standard. The bootloader will search for this
# magic sequence and recognize us as a multiboot kernel.
.section .multiboot
.align 4
.long MAGIC
.long FLAGS
.long CHECKSUM

# Currently the stack pointer register (esp) points at anything and using it may
# cause massive harm. Instead, we'll provide our own stack. We will allocate
# room for a small temporary stack by creating a symbol at the bottom of it,
# then allocating 16384 bytes for it, and finally creating a symbol at the top.
.section .bootstrap_stack, "aw", @nobits
stack_bottom:
.skip 16384 # 16 KiB
stack_top:

# The linker script specifies _start as the entry point to the kernel and the
# bootloader will jump to this position once the kernel has been loaded. It
# doesn't make sense to return from this function as the bootloader is gone.
.section .text
.global _start
.type _start, @function
```

```
_start:
        # Welcome to kernel mode! We now have sufficient code for the bootloader to
        # load and run our operating system. It doesn't do anything interesting yet.
        # Perhaps we would like to call printf("Hello, World\n"). You should now
        # realize one of the profound truths about kernel mode: There is nothing
        # there unless you provide it yourself. There is no printf function. There
        # is no <stdio.h> header. If you want a function, you will have to code it
        # yourself. And that is one of the best things about kernel development:
        # you get to make the entire system yourself. You have absolute and complete
        # power over the machine, there are no security restrictions, no safe
        # guards, no debugging mechanisms, there is nothing but what you build.

        # By now, you are perhaps tired of assembly language. You realize some
        # things simply cannot be done in C, such as making the multiboot header in
        # the right section and setting up the stack. However, you would like to
        # write the operating system in a higher level language, such as C or C++.
        # To that end, the next task is preparing the processor for execution of
        # such code. C doesn't expect much at this point and we only need to set up
        # a stack. Note that the processor is not fully initialized yet and stuff
        # such as floating point instructions are not available yet.

        # To set up a stack, we simply set the esp register to point to the top of
        # our stack (as it grows downwards).
        movl $stack_top, %esp

        # We are now ready to actually execute C code. We cannot embed that in an
        # assembly file, so we'll create a kernel.c file in a moment. In that file,
        # we'll create a C entry point called kernel_main and call it here.
        call kernel_main

        # In case the function returns, we'll want to put the computer into an
        # infinite loop. To do that, we use the clear interrupt ('cli') instruction
        # to disable interrupts, the halt instruction ('hlt') to stop the CPU until
        # the next interrupt arrives, and jumping to the halt instruction if it ever
        # continues execution, just to be safe. We will create a local label rather
        # than real symbol and jump to there endlessly.
        cli
        hlt
.Lhang:
        jmp .Lhang

# Set the size of the _start symbol to the current location '.' minus its start.
# This is useful when debugging or when you implement call tracing.
.size _start, . - _start
```

You can then assemble boot.s using:

```
i686-elf-as boot.s -o boot.o
```

Implementing the Kernel

So far we have written the bootstrap assembly stub that sets up the processor such that high level languages such as C can be used. It is also possible to use other languages such as C++.

Freestanding and Hosted Environments

If you have done C or C++ programming in user-space, you have used a so-called Hosted Environment. Hosted means that there is a C standard library and other useful runtime features. Alternatively, there is the Freestanding version, which is what we are using here. Freestanding means that there is no C standard library, only what we provide ourselves. However, some header files are actually not part of the C standard library, but rather the compiler. These remain available even in freestanding C source code. In this case we use <stdbool.h> to get the bool datatype, <stddef.h> to get size_t and NULL, and <stdint.h> to get the intx_t and uintx_t datatypes which are invaluable for operating systems development, where you need to make sure that the variable is of an exact size (if we used a short instead of uint16_t and the size of short changed, our VGA driver here would break!). Additionally you can access the <float.h>, <iso646.h>, <limits.h>, and <stdarg.h> headers, as they are also freestanding. GCC actually ships a few more headers, but these are special purpose.

Writing a kernel in C

The following shows how to create a simple kernel in C. This kernel uses the VGA text mode buffer (located at 0xB8000) as the output device. It sets up a simple driver that remembers the location of the next character in this buffer and provides a primitive for adding a new character. Notably, there is no support for line breaks ('\n') (and writing that character will show some VGA-specific character instead) and no support for scrolling when the screen is filled up. Adding this will be your first task. Please take a few moments to understand the code.

```
#if !defined(__cplusplus)
#include <stdbool.h> /* C doesn't have booleans by default. */
#endif
#include <stddef.h>
#include <stdint.h>

/* Check if the compiler thinks if we are targeting the wrong op
#if defined(__linux__)
#error "You are not using a cross-compiler, you will most certai
#endif

/* This tutorial will only work for the 32-bit ix86 targets. */
#if !defined(__i386__)
#error "This tutorial needs to be compiled with a ix86-elf compi
#endif

/* Hardware text mode color constants. */
enum vga_color {
        COLOR_BLACK = 0,
        COLOR_BLUE = 1,
        COLOR_GREEN = 2,
        COLOR_CYAN = 3,
        COLOR_RED = 4,
        COLOR_MAGENTA = 5,
        COLOR_BROWN = 6,
        COLOR_LIGHT_GREY = 7,
        COLOR_DARK_GREY = 8,
        COLOR_LIGHT_BLUE = 9,
        COLOR_LIGHT_GREEN = 10,
        COLOR_LIGHT_CYAN = 11,
        COLOR_LIGHT_RED = 12,
        COLOR_LIGHT_MAGENTA = 13,
        COLOR_LIGHT_BROWN = 14,
        COLOR_WHITE = 15,
};
```

```c
uint8_t make_color(enum vga_color fg, enum vga_color bg) {
        return fg | bg << 4;
}

uint16_t make_vgaentry(char c, uint8_t color) {
        uint16_t c16 = c;
        uint16_t color16 = color;
        return c16 | color16 << 8;
}

size_t strlen(const char* str) {
        size_t ret = 0;
        while ( str[ret] != 0 )
                ret++;
        return ret;
}

static const size_t VGA_WIDTH = 80;
static const size_t VGA_HEIGHT = 25;

size_t terminal_row;
size_t terminal_column;
uint8_t terminal_color;
uint16_t* terminal_buffer;

void terminal_initialize() {
        terminal_row = 0;
        terminal_column = 0;
        terminal_color = make_color(COLOR_LIGHT_GREY, COLOR_BLAC
        terminal_buffer = (uint16_t*) 0xB8000;
        for (size_t y = 0; y < VGA_HEIGHT; y++) {
                for (size_t x = 0; x < VGA_WIDTH; x++) {
                        const size_t index = y * VGA_WIDTH + x;
                        terminal_buffer[index] = make_vgaentry('
                }
        }
}

void terminal_setcolor(uint8_t color) {
        terminal_color = color;
}

void terminal_putentryat(char c, uint8_t color, size_t x, size_t
        const size_t index = y * VGA_WIDTH + x;
        terminal_buffer[index] = make_vgaentry(c, color);
}

void terminal_putchar(char c) {
        terminal_putentryat(c, terminal_color, terminal_column,
        if (++terminal_column == VGA_WIDTH) {
```

```
                        terminal_column = 0;
                        if (++terminal_row == VGA_HEIGHT) {
                                terminal_row = 0;
                        }
                }
}

void terminal_writestring(const char* data) {
        size_t datalen = strlen(data);
        for (size_t i = 0; i < datalen; i++)
                terminal_putchar(data[i]);
}

#if defined(__cplusplus)
extern "C" /* Use C linkage for kernel_main. */
#endif
void kernel_main() {
        /* Initialize terminal interface */
        terminal_initialize();

        /* Since there is no support for newlines in terminal_pu
         * yet, '\n' will produce some VGA specific character in
         * This is normal.
         */
        terminal_writestring("Hello, kernel World!\n");
}
```

Notice how we wish to use the common C function `strlen`, but this function is part of the C standard library that we don't have available. Instead, we rely on the freestanding header <stddef.h> to provide `size_t` and we simply declare our own implementation of `strlen`. You will have to do this for every function you wish to use (as the freestanding headers only provide macros and data types).

Compile using:

```
i686-elf-gcc -c kernel.c -o kernel.o -std=gnu99 -ffreestanding -
```

Note that the above code uses a few extensions and hence we build as the GNU version of C99.

Writing a kernel in C++

Writing a kernel in C++ is easy. Note that not all features from the language is available. For instance, exception support requires special runtime support and so does memory allocation. To write a kernel in C++, simply use the code above (which also happens to be legal C++) and save it in a kernel.cp (or what your favorite C++ filename extension is). Notice how the kernel_main function has to be declared with C linkage, as otherwise the compiler would include type information in the assembly name (name mangling). This complicates calling the function from our above assembly stub and we therefore use C linkage, where the symbol name is the same as the name of the function (with no additional type information).

You can compile the file kernel.c++ using:

```
i686-elf-g++ -c kernel.c++ -o kernel.o -ffreestanding -O2 -Wall
```

Note that you must have also built a cross C++ compiler for this work.

Linking the Kernel

We can now assemble boot.s and compile kernel.c. This produces two object files that each contain part of the kernel. To create the full and final kernel we will have to link these object files into the final kernel program, usable by the bootloader. When developing user-space programs, your toolchain ships with default scripts for linking such programs. However, these are unsuitable for kernel development and we need to provide our own customized linker script. Save the following in linker.ld:

```
/* The bootloader will look at this image and start execution at the symbol
   designated as the entry point. */
ENTRY(_start)

/* Tell where the various sections of the object files will be put in the final
   kernel image. */
SECTIONS
{
	/* Begin putting sections at 1 MiB, a conventional place for kernels to be
	   loaded at by the bootloader. */
	. = 1M;

	/* First put the multiboot header, as it is required to be put very early
	   early in the image or the bootloader won't recognize the file format.
	   Next we'll put the .text section. */
	.text BLOCK(4K) : ALIGN(4K)
	{
		*(.multiboot)
		*(.text)
	}

	/* Read-only data. */
	.rodata BLOCK(4K) : ALIGN(4K)
	{
		*(.rodata)
	}

	/* Read-write data (initialized) */
	.data BLOCK(4K) : ALIGN(4K)
	{
		*(.data)
	}

	/* Read-write data (uninitialized) and stack */
	.bss BLOCK(4K) : ALIGN(4K)
	{
		*(COMMON)
		*(.bss)
		*(.bootstrap_stack)
	}

	/* The compiler may produce other sections, by default it will put them in
	   a segment with the same name. Simply add stuff here as needed. */
}
```

With these components you can now actually build the final kernel. We use the compiler as the linker as it allows it greater control over the link process. Note that if your kernel is written in C++, you should use the C++ compiler instead.

You can then link your kernel using:

```
i686-elf-gcc -T linker.ld -o myos.bin -ffreestanding -O2 -nostdl
```

Note: Some tutorials suggest linking with i686-elf-ld rather than the compiler, however this prevents the compiler from performing various tasks during linking.

The file myos.bin is now your kernel (all other files are no longer needed). Note that we are linking against libgcc, which implements various runtime routines that your cross-compiler depends on. Leaving it out will give you problems in the future. If you did not build and install libgcc as part of your cross-compiler, you should go back now and build a cross-compiler with libgcc. The compiler depends on this library and will use it regardless of whether you provide it or not.

Booting the Kernel

In a few moments, you will see your kernel in action.

Building a bootable cdrom image

You can easily create a bootable cdrom image containing the GRUB bootloader and your kernel using the program `grub-mkrescue`. You may need to install the GRUB utility programs and the program `xorriso` (version 0.5.6 or higher). First you should create a file called grub.cfg containing the contents:

```
menuentry "myos" {
	multiboot /boot/myos.bin
}
```

Note that the braces must be placed as shown here. You can now create a bootable image of your operating system by typing these commands:

```
mkdir -p isodir
mkdir -p isodir/boot
cp myos.bin isodir/boot/myos.bin
mkdir -p isodir/boot/grub
cp grub.cfg isodir/boot/grub/grub.cfg
grub-mkrescue -o myos.iso isodir
```

Congratulations! You have now created a file called myos.iso that contains your Hello World operating system. If you don't have the program `grub-mkrescue` installed, now is a good time to install GRUB. It should already be installed on Linux systems. Windows users will likely want to use a Cygwin variant if no native grub-mkrescue program is available.

Testing your operating system (QEMU)

Virtual Machines are very useful for development operating systems, as they allow you to quickly test your code and have access to the source code during the execution. Otherwise, you would be in for an endless cycle of reboots that would only annoy you. They start very quickly, especially combined with small operating systems such as ours.

In this tutorial, we will be using QEMU. You can also use other virtual machines if you please. Simply adding the ISO to the CD drive of an empty virtual machine will do the trick.

Install qemu from your repositories, and then use the following command to start your new operating system.

```
qemu-system-i386 -cdrom myos.iso
```

This should start a new virtual machine containing only your ISO as a cdrom. If all goes well, you will be met with a menu provided by the bootloader. Simply select myos and if all goes well, you should see the happy words "Hello, Kernel World!" followed by some mysterious character.

Additionally, qemu supports booting multiboot kernels directly without bootable medium:

```
qemu-system-i386 -kernel myos.bin
```

Testing your operating system (Real Hardware)

The program grub-mkrescue is nice because it makes a bootable ISO that works on both real computers and virtual machines. You can then build an ISO and use it everywhere. To boot your kernel on your local computer you can install myos.bin to your /boot directory and configure your bootloader appropriately.

Or alternatively, you can burn it to an USB stick (erasing all data on it!). To do so, simply find out the name of the USB block device, in my case /dev/sdb but this may vary, and using the wrong block device (your harddisk, gasp!) may be disastrous. If you are using Linux and /dev/sdx is your block name, simply:

```
sudo dd if=myos.iso of=/dev/sdx && sync
```

Your operating system will then be installed on your USB stick. If you configure your BIOS to boot from USB first, you can insert the USB stick and your computer should start your operating system.

Alternatively, the .iso is a normal cdrom image. Simply burn it to a CD or DVD if you feel like wasting one of those on a few kilobytes large kernel.

Moving Forward

Now that you can run your new shiny operating system, congratulations! Of course, depending on how much this interests you, it may just be the beginning. Here's a few things to get going.

Adding Support for Newlines to Terminal Driver

The current terminal driver does not handle newlines. The VGA text mode font stores another character at the location, since newlines are never meant to be actually rendered: they are logical entities. Rather, in terminal_putchar check if c == '\n' and increment terminal_row and reset terminal_column.

Implementing Terminal Scrolling

In case the terminal is filled up, it will just go back to the top of the screen. This is unacceptable for normal use. Instead, it should move all rows up one row and discard the upper most, and leave a blank row at the bottom ready to be filled up with characters. Implement this.

Rendering Colorful ASCII Art

Use the existing terminal driver to render some pretty stuff in all the glorious 16 colors you have available. Note that only 8 colors may be available for the background color, as the uppermost bit in the entries by default means something other than background color. You'll need a real VGA driver to fix this.

Calling Global Constructors

Main article: Calling Global Constructors

This tutorial showed a small example of how to create a minimal environment for C and C++ kernels. Unfortunately, you don't have everything set up yet. For instance, C++ with global objects will not have their constructors called because you never do it. The compiler uses a special mechanism for performing tasks at program initialization time through the `crt*.o` objects, which may be valuable even for C programmers. If you combine the `crt*.o` files correctly, you will create an `_init` function that invokes all the program initialization tasks. Your boot.o object file can then invoke `_init` before calling `kernel_main`.

Meaty Skeleton

Main article: Meaty Skeleton

This tutorial is meant as a minimal example to give impatient beginners a quick hello world operating system. It is deliberately minimal and doesn't show the best practices on how to organize your operating system. The Meaty Skeleton tutorial shows an example of how to organize a minimal operating system with a kernel, room for a standard library to grow, and prepared for a user-space to appear.

Bare Bones (II)

Make your operating system finally self-hosting and then complete Bare Bones under your own operating system while following all the instructions. Note that, however, this cannot be as quick as possible and you'll have to read lots of theory to complete this step.

Frequently Asked Questions

Why the multiboot header? Wouldn't a pure ELF file be loadable by GRUB anyway?

GRUB is capable of loading a variety of formats. However, in this tutorial we are creating a Multiboot compliant kernel that could be loaded by any other compliant bootloader. To achieve this, the multiboot header is mandatory.

Is the AOUT kludge required for my kernel?

The AOUT kludge is not necessary for kernels in ELF format: a multiboot-compliant loader will recognize an ELF executable as such and use the program header to load things in their proper place. You can provide an AOUT kludge with your ELF kernel, in which case the headers of the ELF file are ignored. With any other format, such as AOUT, COFF or PE kernels, the AOUT kludge it is required, however.

Can the multiboot header be anywhere in the kernel file, or does it have to be in a specific offset?

The multiboot header must be in the first 8kb of the kernel file and must be aligned to a 32-bit (4 byte) boundary for GRUB to find it. You can ensure that this is the case by putting the header in its own source code file and passing that as the first object file to LD.

Will GRUB wipe the BSS section before loading the kernel?

Yes. For ELF kernels, the .bss section is automatically identified and cleared (despite the Multiboot specification being a bit vague about it). For other formats, if you ask it politely to do so, that is if you use the 'address override' information from the multiboot header (flag #16) and give a non-zero value to the bss_end_addr field. Note that using "address override" with an ELF kernel will disable the default behavior and do what is described by the "address override" header instead.

What is the state of registers/memory/etc. when GRUB calls my kernel?

GRUB is an implementation of the Multiboot specification. Anything not specified there is "undefined behavior", which should ring a bell (not only) with C/C++ programmers... Better check the Machine State section of multiboot documentation, and assume nothing else.

I still get `Error 13: Invalid or unsupported executable format from GRUB`...

Chances are the multiboot header is missing from the final executable, or it is not at the right location.
If you are using some other format than ELF (such as PE), you should specify the AOUT kludge in the multiboot header. The mbchk program (coming with GRUB) and "objdump -h" should give you more hints about what is going on.
It may also happen if you use an ELF object file instead of an executable (e.g. you have an ELF file with unresolved symbols or unfixable relocations). Try to link your ELF file to a binary executable to get more accurate error messages.
A common problem when your kernel size increases, is that the multiboot header does no longer appear at the start of the output binary. The common solutions is to put the multiboot header in a separate section and make sure that section is first in the output binary, or to include the multiboot header itself in the linker script.

I get `Boot failed: Could not read from CDROM (code 0009)` when trying to boot the iso image in qemu

If your development system is booted from EFI it may be that you don't have the PC-BIOS version of the grub binaries installed anywhere. If you install them then grub-mkrescue will by default produce a hybrid ISO that will work in qemu. On ubuntu this can be achieved with: **apt-get install grub-pc-bin**.

See Also

Articles

- Books

External Links

- Multiboot Specification (https://www.gnu.org/software/grub/manual/multiboot/multiboot.html)
- The POSIX standard (http://pubs.opengroup.org/onlinepubs/9699919799//)