

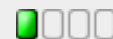
Meaty Skeleton

From OSDev Wiki

In this tutorial we continue from Bare Bones and create a minimal template operating system suitable for further modification or as inspiration for your initial operating system version. The Bare Bones tutorial only gives you the absolutely minimal code to demonstrate how to correctly cross-compile a kernel, however this is unsuitable as an example operating system. Additionally, this tutorial implements necessary ABI features needed to satisfy the ABI and compiler contracts to prevent possible mysterious errors.

This tutorial also serves as the initial template tutorial on how to create your own libc (Standard C Library). The GCC documentation explicitly states that libgcc requires the freestanding environment to supply the `memcpy`, `memcpy`, `memcpy`, and `memset` functions, as well as `abort` on some platforms. We will satisfy this requirement by creating a special kernel C library (libk) that contains the parts of the user-space libc that are freestanding (doesn't require any kernel features) as opposed to hosted libc features that need to do system calls.

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

Contents

- 1 Preface
- 2 Bare Bones
- 3 Building a Cross-Compiler
- 4 System Root
- 5 System Headers
- 6 Makefile Design
- 7 Architecture Directories
- 8 Kernel Design
- 9 libc and libk Design
- 10 Source Code
 - 10.1 kernel
 - 10.1.1 kernel/include/kernel/vga.h
 - 10.1.2 kernel/include/kernel/tty.h
 - 10.1.3 kernel/Makefile
 - 10.1.4 kernel/kernel/kernel.c
 - 10.1.5 kernel/arch/i386/tty.c
 - 10.1.6 kernel/arch/i386/crt0.S
 - 10.1.7 kernel/arch/i386/make.config
 - 10.1.8 kernel/arch/i386/crti.S
 - 10.1.9 kernel/arch/i386/linker.ld
 - 10.1.10 kernel/arch/i386/boot.S
 - 10.1.11 kernel/.gitignore
 - 10.2 libc and libk

- 10.2.1 libc/include/string.h
- 10.2.2 libc/include/stdio.h
- 10.2.3 libc/include/sys/cdefs.h
- 10.2.4 libc/include/stdlib.h
- 10.2.5 libc/Makefile
- 10.2.6 libc/stdlib/abort.c
- 10.2.7 libc/string/memmove.c
- 10.2.8 libc/string/strlen.c
- 10.2.9 libc/string/memcmp.c
- 10.2.10 libc/string/memset.c
- 10.2.11 libc/string/memcpy.c
- 10.2.12 libc/stdio/puts.c
- 10.2.13 libc/stdio/putchar.c
- 10.2.14 libc/stdio/printf.c
- 10.2.15 libc/arch/i386/make.config
- 10.2.16 libc/.gitignore
- 10.3 Miscellaneous
 - 10.3.1 build.sh
 - 10.3.2 clean.sh
 - 10.3.3 config.sh
 - 10.3.4 default-host.sh
 - 10.3.5 headers.sh
 - 10.3.6 iso.sh
 - 10.3.7 qemu.sh
 - 10.3.8 target-triplet-to-arch.sh
 - 10.3.9 .gitignore
- 11 Cross-Compiling the Operating System
- 12 Troubleshooting
- 13 Moving Forward
 - 13.1 Renaming MyOS to YourOS
 - 13.2 Improving the Build System
 - 13.3 Stack Smash Protector
 - 13.4 User-Space

Preface

This tutorial is an example on how you could structure your operating system in a manner that will continue to serve you well for the foreseeable future. This serves as both inspiration and as an example for those that wish to something different, while as a base for the rest. The tutorial does embed a few important concepts into your operating system such as the existence of a libc, as well as indirectly other minor Unix and ABI semantics. Adapt what you wish from this tutorial. Note that the shell script and Make-based build system constructed in this tutorial is meant for Unix systems, though it should work in Cygwin. There is no pressing need to make this portable across all operating systems as this is just an example.

We will name this new example operating system `myos`. This is just a placeholder and you should replace all occurrences of `myos` with what you decide to call your operating system.

Bare Bones

Main article: Bare Bones

You are expected to have completed the Bare Bones tutorial before continuing to this tutorial. It is not strictly necessary to have completed Bare Bones, but doing so confirms that your development environment works as well as explaining a number of core things.

You should probably discard the code you got from toying around with Bare Bones and start over with this tutorial as your basis.

Building a Cross-Compiler

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

You must use a GCC Cross-Compiler in this tutorial as in the Bare Bones tutorial. You should use the **i686-elf** target in your cross-compiler, though any `ix86-elf` target (but no less than i386) will do fine for our purposes here.

You must configure your cross-binutils with the `--with-sysroot` option, otherwise linking will mysteriously fail with the this linker was not configured to use sysroots error message. If you forgot to configure your cross-binutils with that option, you'll have to rebuild it, but you can keep your cross-gcc.

System Root

Normally when you compile programs for your local operating system, the compiler locates development files such as headers and libraries in system directories such as:

```
/usr/include
/usr/lib
```

These files are of course not usable for your operating system. Instead you want to have your own version of these directories that contains files for your operating system:

```
/home/bwayne/myos/sysroot/usr/include
/home/bwayne/myos/sysroot/usr/lib
```

The `/home/bwayne/myos/sysroot` directory acts as a fake root directory for your operating system. This is called a system root, or sysroot.

You can think of the sysroot as the root directory for your operating system. Your build process will build each component of your operating system (kernel, standard library, programs) and gradually install them into the system root. Ultimately the system root will be a fully functional root filesystem for your operating system, you format a partition and copy the files there, configure a bootloader to load the kernel from there, and use your harddisk driver and filesystem driver to read the files from there. The system root is thus a temporary directory that will ultimately become the actual root directory of your operating system.

In this example the cross system root is located as `sysroot/`, which is a directory created by the build scripts and populated by the `make install` targets. The makefiles will install the system headers into the `sysroot/usr/include` directory, the system libraries into the `sysroot/usr/lib` directory and the kernel

itself into the `sysroot/boot` directory.

We use system roots already now because it will make it smoother to add a user-space when you get that far. This scheme is very convenient when you later Port Third-Party Software by Cross-Compiling It.

The `-elf` targets have no user-space and are incapable of having one. We configured the compiler with system root support, so it will look in `${SYSROOT}/usr/lib` as expected. We prevented the compiler from searching for a standard library using the `--without-headers` option when building `i686-elf-gcc`, so it will not look in `${SYSROOT}/usr/include`. (Once you add a user-space and a `libc`, you will configure your custom cross-gcc with `--with-sysroot` and it will look in `${SYSROOT}/usr/include`. As a temporary work-around until you get that far, we fix it by passing `-isystem=/usr/include`).

You can change the system root directory layout if you wish, but you have to modify some `binutils` and `gcc` source code and tell them what your operating system is. This is advanced and not worth doing until you add a proper user-space. Note that the cross-linker currently looks in `/lib`, `/usr/lib` and `/usr/local/lib` by default, so you can move files there without changing `binutils`. Also note that we use the `-isystem` option for GCC (as it was configured without a system include directory), so you can move that around freely.

System Headers

The `./headers.sh` script simply installs the headers for your `libc` and kernel (system headers) into `sysroot/usr/include`, but doesn't actually cross-compile your operating system. This is `useful` as it allows you to provide the compiler a copy of your headers before you actually compile your system. You will need this when we add proper system root support once it is time to add a proper user space.

Note how your cross-compiler comes with a number of fully freestanding headers such as `stddef.h` and `stdint.h`. These headers simply declare types and macros that are useful. Your kernel standard library will supply a number of useful functions (such as `strlen`) that doesn't require system calls and are freestanding except they need an implementation somewhere.

Makefile Design

The makefiles in this example respect the environment variables (such as `CFLAGS` that tell what default compile options are used to compile C programs). This lets the user control stuff such as which optimization levels are used, while a default is used if the user has no opinion. The makefiles also make sure that particular options are always in `CFLAGS`. This is done by having two phases in the makefiles: one that set a default value and one that adds mandatory options the project makefile requires:

```
# Default CFLAGS:
CFLAGS?=-O2 -g

# Add mandatory options to CFLAGS:
CFLAGS:=$(CFLAGS) -Wall -Wextra
```

Architecture Directories

The projects in this example (libc and kernel) store all the architecture dependent source files inside an `arch/` directory with their own sub-makefile that has special configuration. This cleanly separates the systems you support and will make it easier to port to other systems in the future.

Kernel Design

We have moved the kernel into its own directory named `kernel/`. It would perhaps be better to call it something else if your kernel has another name than your full operating system distribution, though calling it `kernel/` makes it easier for other hobbyist developers to find the core parts of your new operating system.

The kernel installs its public kernel headers into `sysroot/usr/include/kernel`. This is useful if you decide to create a kernel with modules, where modules can then simply include the public headers from the main kernel.

GNU GRUB is used as the bootloader and the kernel uses multiboot as in the Bare Bones tutorial.

The kernel implements the correct way of invoking global constructors (useful for C++ code and C code using `__attribute__((constructor))`). The kernel initialization is done in two steps: First the `kernel_early()` function is called which sets up the essential kernel features (such as the kernel log). The bootstrap assembly then proceeds to call `_init` (which invokes all the global constructors) and finally invoke `kernel_main()`.

The special `__is_myos_kernel` macro lets the source code detect whether it is part of the kernel.

The kernel is built with `-fbuiltin`, which promises that the C standard functions doesn't do anything unusual. This lets the compiler optimize certain compile-time constant expressions such as `strlen("foo")` to 3. However, this can have unexpected consequences as a function call can be changed into a call to another function. For instance, GCC likes to rewrite `printf("Hello, World!\n")` into `puts("Hello, World!")`. This is not a terrible problem if you add the needed functions to `libk` as you need them, or you can simply remove `-fbuiltin` as `-ffreestanding` turns it off by default.

libc and libk Design

The `libc` and `libk` are actually two versions of the same library, which is stored in the directory `libc/`. The standard library is split into two versions: freestanding and hosted. The difference is that the freestanding library (`libk`) doesn't contain any of the code that only works in user-space, such as system calls. The `libk` is also built with different compiler options, just like the kernel isn't built like normal user-space code.

You are not required to have a `libk`. You could just as easily have a regular `libc` and a fully separate minimal project inside the kernel directory. The `libk` scheme avoids code duplication, so you don't have to maintain multiple versions of `strlen` and such.

This example doesn't come with a usable `libc`. It compiles a `libc.a` that is entirely useless, except being a skeleton we can build on when we add user-space in a later tutorial.

Each standard function is put inside a file with the same name as the function inside a directory with the name of the header. For instance, `strlen` from `string.h` is in `libc/string/strlen.c` and `stat` from `sys/stat.h` would be in `libc/sys/stat/stat.c`.

The standard headers use a BSD-like scheme where `sys/cdefs.h` declares a bunch of useful preprocessor macros meant for internal use by the standard library. For instance, all the function prototypes are wrapped in `extern "C" {` and `}` such that C++ code can correctly link against `libc` (as `libc` doesn't use C++ linkage).

Note also how the compiler provides the internal keyword `__restrict` unconditionally (even in C89) mode, which is useful for adding the `restrict` keyword to function prototypes even when compiling code in pre-C99 mode.

The special `__is_mynos_libc` macro lets the source code detect whether it is part of the libc.

This example comes with a small number of standard functions that serve as examples and serve to satisfy ABI requirements. Note that the `printf` function included is very minimal and doesn't handle most common features, this is intentional.

Source Code

You can easily download the source code using git from the Meaty Skeleton git repository (<https://gitlab.com/sortie/meaty-skeleton>) . This is preferable to doing a manual error-prone copy, as you may make mistake or whitespace gets garbled due to bugs in our syntax highlighting. To clone the git repository, do:

```
git clone https://gitlab.com/sortie/meaty-skeleton.git
```

Check for differences between the git revision used in this article and what you cloned (empty output is no difference):

```
git diff 2f7a731513b2783c1812af3acb4c8498eef1e95b..master
```

Operating systems development is about being an expert. Take the time to read the code carefully through and understand it. Please seek further information and help if you don't understand aspects of it. This code is minimal and almost everything is done deliberately, often to pre-emptively solve future problems.

kernel

kernel/include/kernel/vga.h

```
#ifndef _KERNEL_VGA_H
#define _KERNEL_VGA_H

#include <stdint.h>

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,
};

static inline uint8_t make_color(enum vga_color fg, enum vga_col
{
    return fg | bg << 4;
}

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

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

static uint16_t* const VGA_MEMORY = (uint16_t*) 0xB8000;

#endif

```

kernel/include/kernel/tty.h

```

#ifndef _KERNEL_TTY_H
#define _KERNEL_TTY_H

#include <stddef.h>

void terminal_initialize(void);
void terminal_putchar(char c);
void terminal_write(const char* data, size_t size);
void terminal_writestring(const char* data);

#endif

```

kernel/Makefile

```

HOST?=$(shell ../default-host.sh)
HOSTARCH:=$(shell ../target-triplet-to-arch.sh $(HOST))

```

```

CFLAGS?=-O2 -g
CPPFLAGS?=
LDFLAGS?=
LIBS?=

DESTDIR?=
PREFIX?=/usr/local
EXEC_PREFIX?=$(PREFIX)
BOOTDIR?=$(EXEC_PREFIX)/boot
INCLUDEDIR?=$(PREFIX)/include

CFLAGS:=$(CFLAGS) -ffreestanding -fbuiltin -Wall -Wextra
CPPFLAGS:=$(CPPFLAGS) -D__is_myos_kernel -Iinclude
LDFLAGS:=$(LDFLAGS)
LIBS:=$(LIBS) -nostdlib -lk -lgcc

ARCHDIR:=arch/$(HOSTARCH)

include $(ARCHDIR)/make.config

CFLAGS:=$(CFLAGS) $(KERNEL_ARCH_CFLAGS)
CPPFLAGS:=$(CPPFLAGS) $(KERNEL_ARCH_CPPFLAGS)
LDFLAGS:=$(LDFLAGS) $(KERNEL_ARCH_LDFLAGS)
LIBS:=$(LIBS) $(KERNEL_ARCH_LIBS)

OBJS:=\
$(KERNEL_ARCH_OBJS) \
kernel/kernel.o \

CRTI_OBJ:=$(ARCHDIR)/crti.o
CRTBEGIN_OBJ:=$(shell $(CC) $(CFLAGS) $(LDFLAGS) -print-file-name=
CRTEND_OBJ:=$(shell $(CC) $(CFLAGS) $(LDFLAGS) -print-file-name=
CRTN_OBJ:=$(ARCHDIR)/crtn.o

ALL_OUR_OBJS:=\
$(CRTI_OBJ) \
$(OBJS) \
$(CRTN_OBJ) \

OBJ_LINK_LIST:=\
$(CRTI_OBJ) \
$(CRTBEGIN_OBJ) \
$(OBJS) \
$(CRTEND_OBJ) \
$(CRTN_OBJ) \

all: myos.kernel

.PHONY: all clean install install-headers install-kernel

```

```

myos.kernel: $(OBJ_LINK_LIST) $(ARCHDIR)/linker.ld
             $(CC) -T $(ARCHDIR)/linker.ld -o $@ $(CFLAGS) $(OBJ_LINK

%.o: %.c
             $(CC) -c $< -o $@ -std=gnu11 $(CFLAGS) $(CPPFLAGS)

%.o: %.S
             $(CC) -c $< -o $@ $(CFLAGS) $(CPPFLAGS)

clean:
    rm -f myos.kernel $(OBJS) $(ALL_OUR_OBJS) *.o */*.o */*/

install: install-headers install-kernel

install-headers:
    mkdir -p $(DESTDIR)$(INCLUDEDIR)
    cp -RTv include $(DESTDIR)$(INCLUDEDIR)

install-kernel: myos.kernel
    mkdir -p $(DESTDIR)$(BOOTDIR)
    cp myos.kernel $(DESTDIR)$(BOOTDIR)

```

kernel/kernel/kernel.c

```

#include <stddef.h>
#include <stdint.h>
#include <string.h>
#include <stdio.h>

#include <kernel/tty.h>

void kernel_early(void)
{
    terminal_initialize();
}

void kernel_main(void)
{
    printf("Hello, kernel World!\n");
}

```

kernel/arch/i386/tty.c

```

#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>

```

```

#include <string.h>

#include <kernel/vga.h>

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

void terminal_initialize(void)
{
    terminal_row = 0;
    terminal_column = 0;
    terminal_color = make_color(COLOR_LIGHT_GREY, COLOR_BLACK);
    terminal_buffer = VGA_MEMORY;
    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(' ', terminal_color);
        }
    }
}

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

void terminal_putentryat(char c, uint8_t color, size_t x, size_t y)
{
    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, terminal_row);
    if ( ++terminal_column == VGA_WIDTH )
    {
        terminal_column = 0;
        if ( ++terminal_row == VGA_HEIGHT )
        {
            terminal_row = 0;
        }
    }
}

void terminal_write(const char* data, size_t size)
{

```

```

        for ( size_t i = 0; i < size; i++ )
            terminal_putchar(data[i]);
    }

    void terminal_writestring(const char* data)
    {
        terminal_write(data, strlen(data));
    }

```

kernel/arch/i386/crt0.S

```

.section .init
    /* gcc will nicely put the contents of crtend.o's .init section here. */
    popl %ebp
    ret

.section .fini
    /* gcc will nicely put the contents of crtend.o's .fini section here. */
    popl %ebp
    ret

```

kernel/arch/i386/make.config

```

KERNEL_ARCH_CFLAGS:=
KERNEL_ARCH_CPPFLAGS:=
KERNEL_ARCH_LDFLAGS:=
KERNEL_ARCH_LIBS:=

KERNEL_ARCH_OBJS:=\
$(ARCHDIR)/boot.o \
$(ARCHDIR)/tty.o \

```

kernel/arch/i386/crti.S

```

.section .init
.global _init
.type _init, @function
_init:
    push %ebp
    movl %esp, %ebp
    /* gcc will nicely put the contents of crtbegin.o's .init section here. */

.section .fini
.global _fini
.type _fini, @function
_fini:
    push %ebp
    movl %esp, %ebp
    /* gcc will nicely put the contents of crtbegin.o's .fini section here. */

```

kernel/arch/i386/linker.ld

```
/* The bootloader will look at this image and start execution at the symbol
   designated at 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, put them in the proper place in
       in this file, if you'd like to include them in the final kernel. */
}
```

kernel/arch/i386/boot.S

```
# 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.
.section .multiboot
.align 4
.long MAGIC
.long FLAGS
.long CHECKSUM

# Reserve a stack for the initial thread.
.section .bootstrap_stack, "aw", @nobits
```

```

stack_bottom:
.skip 16384 # 16 KiB
stack_top:

# The kernel entry point.
.section .text
.global _start
.type _start, @function
_start:
    movl $stack_top, %esp

    # Initialize the core kernel before running the global constructors.
    call kernel_early

    # Call the global constructors.
    call _init

    # Transfer control to the main kernel.
    call kernel_main

    # Hang if kernel_main unexpectedly returns.
    cli
.Lhang:
    hlt
    jmp .Lhang
.size _start, . - _start

```

kernel/.gitignore

```

*.o
*.kernel

```

libc and libk

libc/include/string.h

```

#ifndef _STRING_H
#define _STRING_H 1

#include <sys/cdefs.h>

#include <stddef.h>

#ifdef __cplusplus
extern "C" {
#endif

int memcmp(const void*, const void*, size_t);
void* memcpy(void* __restrict, const void* __restrict, size_t);
void* memmove(void*, const void*, size_t);
void* memset(void*, int, size_t);
size_t strlen(const char*);

#ifdef __cplusplus
}

```

```
#endif
```

```
#endif
```

libc/include/stdio.h

```
#ifndef _STDIO_H
#define _STDIO_H 1

#include <sys/cdefs.h>

#ifdef __cplusplus
extern "C" {
#endif

int printf(const char* __restrict, ...);
int putchar(int);
int puts(const char*);

#ifdef __cplusplus
}
#endif

#endif
```

libc/include/sys/cdefs.h

```
#ifndef _SYS_CDEFS_H
#define _SYS_CDEFS_H 1

#define __myos_libc 1

#endif
```

libc/include/stdlib.h

```
#ifndef _STDLIB_H
#define _STDLIB_H 1

#include <sys/cdefs.h>

#ifdef __cplusplus
extern "C" {
#endif
```

```

__attribute__((__noreturn__))
void abort(void);

#ifdef __cplusplus
}
#endif

#endif

```

libc/Makefile

```

HOST?=$(shell ../default-host.sh)
HOSTARCH:=$(shell ../target-triplet-to-arch.sh $(HOST))

CFLAGS?=-O2 -g
CPPFLAGS?=
LDFLAGS?=
LIBS?=

DESTDIR?=
PREFIX?=/usr/local
EXEC_PREFIX?=$(PREFIX)
INCLUDEDIR?=$(PREFIX)/include
LIBDIR?=$(EXEC_PREFIX)/lib

CFLAGS:=$(CFLAGS) -Wall -Wextra
CPPFLAGS:=$(CPPFLAGS) -D__is_myos_libc -Iinclude
LIBK_CFLAGS:=$(CFLAGS) -ffreestanding -fbuiltin
LIBK_CPPFLAGS:=$(CPPFLAGS) -D__is_myos_kernel

ARCHDIR:=arch/$(HOSTARCH)

include $(ARCHDIR)/make.config

CFLAGS:=$(CFLAGS) $(ARCH_CFLAGS)
CPFLAGS:=$(CPPFLAGS) $(ARCH_CPPFLAGS)
LIBK_CFLAGS:=$(LIBK_CFLAGS) $(KERNEL_ARCH_CFLAGS)
LIBK_CPFLAGS:=$(LIBK_CPPFLAGS) $(KERNEL_ARCH_CPPFLAGS)

FREEOBS:=\
$(ARCH_FREEOBS) \
stdio/printf.o \
stdio/putchar.o \
stdio/puts.o \
stdlib/abort.o \
string/memcmp.o \
string/memcpy.o \
string/memmove.o \
string/memset.o \

```

```
string/strlen.o \

HOSTEDOBJS:=\
$(ARCH_HOSTEDOBJS) \

OBJS:=\
$(FREEOBJS) \
$(HOSTEDOBJS) \

LIBK_OBJS:=$(FREEOBJS:.o=.libk.o)

BINARIES=libc.a libg.a libk.a

all: $(BINARIES)

.PHONY: all clean install install-headers install-libs

libc.a: $(OBJS)
        $(AR) rcs $@ $(OBJS)

libg.a:
        $(AR) rcs $@

libk.a: $(LIBK_OBJS)
        $(AR) rcs $@ $(LIBK_OBJS)

%.o: %.c
        $(CC) -c $< -o $@ -std=gnu11 $(CFLAGS) $(CPPFLAGS)

%.o: %.S
        $(CC) -c $< -o $@ $(CFLAGS) $(CPPFLAGS)

%.libk.o: %.c
        $(CC) -c $< -o $@ -std=gnu11 $(LIBK_CFLAGS) $(LIBK_CPPFL

%.libk.o: %.S
        $(CC) -c $< -o $@ $(LIBK_CFLAGS) $(LIBK_CPPFLAGS)

clean:
        rm -f $(BINARIES) $(OBJS) $(LIBK_OBJS) *.o */*.o */*/*.o

install: install-headers install-libs

install-headers:
        mkdir -p $(DESTDIR)$ (INCLUDEDIR)
        cp -RTv include $(DESTDIR)$ (INCLUDEDIR)

install-libs: $(BINARIES)
        mkdir -p $(DESTDIR)$ (LIBDIR)
        cp $(BINARIES) $(DESTDIR)$ (LIBDIR)
```

libc/stdlib/abort.c

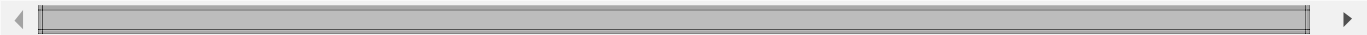
```
#include <stdio.h>
#include <stdlib.h>

__attribute__((__noreturn__))
void abort(void)
{
    // TODO: Add proper kernel panic.
    printf("Kernel Panic: abort()\n");
    while ( 1 ) { }
    __builtin_unreachable();
}
```

libc/string/memmove.c

```
#include <string.h>

void* memmove(void* dstptr, const void* srcptr, size_t size)
{
    unsigned char* dst = (unsigned char*) dstptr;
    const unsigned char* src = (const unsigned char*) srcptr
    if ( dst < src )
        for ( size_t i = 0; i < size; i++ )
            dst[i] = src[i];
    else
        for ( size_t i = size; i != 0; i-- )
            dst[i-1] = src[i-1];
    return dstptr;
}
```



libc/string/strlen.c

```
#include <string.h>

size_t strlen(const char* string)
{
    size_t result = 0;
    while ( string[result] )
        result++;
    return result;
}
```

libc/string/memcmp.c

```
#include <string.h>

int memcmp(const void* aptr, const void* bptr, size_t size)
{
    const unsigned char* a = (const unsigned char*) aptr;
    const unsigned char* b = (const unsigned char*) bptr;
    for ( size_t i = 0; i < size; i++ )
        if ( a[i] < b[i] )
            return -1;
        else if ( b[i] < a[i] )
            return 1;

    return 0;
}
```

libc/string/memset.c

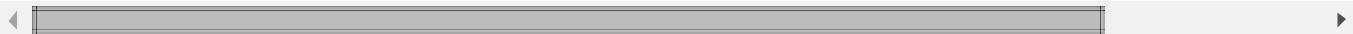
```
#include <string.h>

void* memset(void* bufptr, int value, size_t size)
{
    unsigned char* buf = (unsigned char*) bufptr;
    for ( size_t i = 0; i < size; i++ )
        buf[i] = (unsigned char) value;
    return bufptr;
}
```

libc/string/memcpy.c

```
#include <string.h>

void* memcpy(void* restrict dstptr, const void* restrict srcptr,
{
    unsigned char* dst = (unsigned char*) dstptr;
    const unsigned char* src = (const unsigned char*) srcptr
    for ( size_t i = 0; i < size; i++ )
        dst[i] = src[i];
    return dstptr;
}
```



libc/stdio/puts.c

```
#include <stdio.h>

int puts(const char* string)
{
    return printf("%s\n", string);
}
```

libc/stdio/putchar.c

```
#include <stdio.h>

#if defined(__is_myos_kernel)
#include <kernel/tty.h>
#endif

int putchar(int ic)
{
    #if defined(__is_myos_kernel)
        char c = (char) ic;
        terminal_write(&c, sizeof(c));
    #else
        // TODO: You need to implement a write system call.
    #endif
    return ic;
}
```

libc/stdio/printf.c

```
#include <stdbool.h>
#include <stdarg.h>
#include <stdio.h>
#include <string.h>

static void print(const char* data, size_t data_length)
{
    for ( size_t i = 0; i < data_length; i++ )
        putchar((int) ((const unsigned char*) data)[i]);
}

int printf(const char* restrict format, ...)
{
    va_list parameters;
    va_start(parameters, format);

    int written = 0;
    size_t amount;
```

```

bool rejected_bad_specifier = false;

while ( *format != '\0' )
{
    if ( *format != '%' )
    {
        print_c:
            amount = 1;
            while ( format[amount] && format[amount]
                    amount++;
            print(format, amount);
            format += amount;
            written += amount;
            continue;
    }

    const char* format_began_at = format;

    if ( *(++format) == '%' )
        goto print_c;

    if ( rejected_bad_specifier )
    {
        incomprehensible_conversion:
            rejected_bad_specifier = true;
            format = format_began_at;
            goto print_c;
    }

    if ( *format == 'c' )
    {
        format++;
        char c = (char) va_arg(parameters, int /
        print(&c, sizeof(c));
    }
    else if ( *format == 's' )
    {
        format++;
        const char* s = va_arg(parameters, const
        print(s, strlen(s));
    }
    else
    {
        goto incomprehensible_conversion;
    }
}

va_end(parameters);

return written;
}

```

libc/arch/i386/make.config

```
ARCH_CFLAGS:=  
ARCH_CPPFLAGS:=  
KERNEL_ARCH_CFLAGS:=  
KERNEL_ARCH_CPPFLAGS:=  
  
ARCH_FREEOBJS:=\  
  
ARCH_HOSTEDOBJS:=\  

```

libc/.gitignore

```
*.o  
*.a
```

Miscellaneous

These files go into the root source directory.

build.sh

```
#!/bin/sh  
set -e  
./headers.sh  
  
for PROJECT in $PROJECTS; do  
    DESTDIR="$PWD/sysroot" $MAKE -C $PROJECT install  
done
```

You should make this executable script executable by running:

```
chmod +x build.sh
```

clean.sh

```
#!/bin/sh  
set -e  
./config.sh
```

```
for PROJECT in $PROJECTS; do
    $MAKE -C $PROJECT clean
done

rm -rfv sysroot
rm -rfv isodir
rm -rfv myos.iso
```

You should make this executable script executable by running:

```
chmod +x clean.sh
```

config.sh

```
SYSTEM_HEADER_PROJECTS="libc kernel"
PROJECTS="libc kernel"

export MAKE=${MAKE:-make}
export HOST=${HOST:-$(./default-host.sh)}

export AR=${HOST}-ar
export AS=${HOST}-as
export CC=${HOST}-gcc

export PREFIX=/usr
export EXEC_PREFIX=$PREFIX
export BOOTDIR=/boot
export LIBDIR=$EXEC_PREFIX/lib
export INCLUDEDIR=$PREFIX/include

export CFLAGS='-O2 -g'
export CPPFLAGS=''

# Configure the cross-compiler to use the desired system root.
export CC="$CC --sysroot=$PWD/sysroot"

# Work around that the -elf gcc targets doesn't have a system in
# because configure received --without-headers rather than --with
if echo "$HOST" | grep -Eq -- '-elf($|-)'; then
    export CC="$CC -isystem=$INCLUDEDIR"
fi
```



default-host.sh

```
#!/bin/sh
echo i686-elf
```

You should make this executable script executable by running:

```
chmod +x default-host.sh
```

headers.sh

```
#!/bin/sh
set -e
. ./config.sh

mkdir -p sysroot

for PROJECT in $SYSTEM_HEADER_PROJECTS; do
    DESTDIR="$PWD/sysroot" $MAKE -C $PROJECT install-headers
done
```

You should make this executable script executable by running:

```
chmod +x headers.sh
```

iso.sh

```
#!/bin/sh
set -e
. ./build.sh

mkdir -p isodir
mkdir -p isodir/boot
mkdir -p isodir/boot/grub

cp sysroot/boot/myos.kernel isodir/boot/myos.kernel
cat > isodir/boot/grub/grub.cfg << EOF
menuentry "myos" {
    multiboot /boot/myos.kernel
}
EOF
grub-mkrescue -o myos.iso isodir
```

You should make this executable script executable by running:

```
chmod +x iso.sh
```

qemu.sh

```
#!/bin/sh
set -e
. ./iso.sh

qemu-system-$ (./target-triplet-to-arch.sh $HOST) -cdrom myos.iso
```

You should make this executable script executable by running:

```
chmod +x qemu.sh
```

target-triplet-to-arch.sh

```
#!/bin/sh
if echo "$1" | grep -Eq 'i[[:digit:]]86-'; then
    echo i386
else
    echo "$1" | grep -Eo '^[[:alnum:]]*'
fi
```

You should make this executable script executable by running:

```
chmod +x target-triplet-to-arch.sh
```

.gitignore

```
*.iso
isodir
sysroot
```

Cross-Compiling the Operating System

The system is cross-compiled in the same manner as Bare Bones, though with the complexity of having a system root with the final system and using a libk. In this example, we elected to use shell scripts to to the top-level build process, though you could possibly also use a makefile for that or a wholly different build system. Though, assuming this setup works for you, you can clean the source tree by invoking:

```
./clean.sh
```

You can install all the system headers into the system root without relying on the compiler at all, which will be useful later on when switching to a Hosted GCC Cross-Compiler, by invoking:

```
./headers.sh
```

You can build a bootable cdrom image of the operating system by invoking:

```
./iso.sh
```

It's probably a good idea to create a quick build-and-then-launch short-cut like used in this example to run the system in your favorite emulator quickly:

```
./qemu.sh
```

Troubleshooting

If you receive odd errors during the build, you may have made a mistake during manual copying, perhaps missed a file, forgotten to make a file executable, or bugs in the highlighting software we use cause unintended whitespace to appear. Perform a git repository clone as described above, and use that code instead, or compare the two directory trees with the `diff(1)` diff command line utility. If you made personal changes to the code, those may be at fault.

Moving Forward

You should adapt this template to your needs. There's a number of things you should consider doing now:

Renaming MyOS to YourOS

Certainly you wish to name your operating system after your favorite flower, hometown, boolean value, or whatever marketing told you. Do a search and replace that replaces `myos` with whatever you wish to call it. Keep in mind that the name is deliberately lower-case in a few places for technical reasons.

Improving the Build System

Main article: [Hard Build System](#)

It is probably worth improving the build system. For instance, it could be useful if `build.sh` accepted command-line options, or perhaps if it used `make`'s important `-j` option for concurrent builds.

It's worth considering how contributors will build your operating system. It's an easy trap to fall into thinking you can make super script that does everything. This will end up complex and insufficiently flexible; or it will be flexible and even more complex. It's better to document what the user should do to prepare a cross toolchain and what prerequisite programs to install. This tutorial shows an example hard build system that merely builds the operating system. You can complete it by documenting how to build a cross-compiler and how to use it.

Stack Smash Protector

Main article: Stack Smashing Protector

Early is not too soon to think about security and robustness. You can take advantage of the optional stack smash protector offered by modern compilers that detect stack buffer overruns rather than behaving unexpectedly (or nothing happening, if unlucky).

User-Space

A later tutorial in this series will extend this template with a proper user-space and an OS Specific Toolchain that fully utilizes the system root.

Retrieved from "http://wiki.osdev.org/index.php?title=Meaty_Skeleton&oldid=18287"

Categories: Level 1 Tutorials | Bare bones tutorials | C

- This page was last modified on 14 August 2015, at 07:46.
- This page has been accessed 17,232 times.