Writing a Kernel From Scratch with Free Software

Eric P. Hutchins

April 10, 2010

Contents

4 CONTENTS

Chapter 1

Introduction

1.1 What is a kernel

An important distinction is the difference between the operating system and the kernel. This book is not named "Writing an Operating System From Scratch with Free Software." That is no accident.

According to the GNU project, an operating system means a collection of programs that are sufficient to use the computer to do a wide variety of jobs. The kernel is one of the programs in an operating system-the program that allocates the machine's resources to the other programs that are running. The kernel also takes care of starting and stopping other programs [?].

This book teaches how to create a kernel, which is different than an operating system. At the end, it may run a modified GNU system if you choose. The GNU operating system has been modified to run on top of the Linux kernel, the FreeBSD kernel, and some others. You may also create create your own operating system on top of your kernel when you are done if that is what you wish. That will probably require you to write a shell, a windowing system, and many other complex programs.

1.2 Why write a kernel?

Why do we write a kernel? Because it helps us understand lower levels of the operating systems that we use, no matter which operating system it is under. The best way to understand a piece of software is to implement your own. Knowing how a kernel works lets you see what kinds of issues there are in operating systems and how they are dealt with. This helps to explain problems with your own operating system, and let's you see any inherent limitations.

Chapter 2

The Build System

2.1 Autoconf

When the program autoconf is run, it looks for a file called configure.ac. We will now write that file.

2.1.1 configure.ac

The first line is AC_INIT. It takes as parameters the name of the package, the version, and the email to which bug reports should be sent.

```
AC_INIT(scratchkernel, 0.1.0, your@email.com)
```

The next line tells the options for running automake. Here we use -Wall, which says to report many warnings that are not reported by default, -Werror, which says to count all warnings as errors, and foreign, which tells automake that the package is not a GNU package and therefore will not be required to include certain files included in the GNU Coding Standards.

```
AM_INIT_AUTOMAKE([-Wall -Werror foreign])
```

These lines simply state that we are using C and Assembly as our source languages.

```
AC_PROG_CC
AM_PROG_AS
```

This keyword takes a list of files that the configure script should generate. We say Makefile and src/Makefile because the main Makefile will references src/Makefile and have it build the files there.

```
AC_OUTPUT([
Makefile
src/Makefile
])
```

2.2 Automake

2.2.1 Makefile.am

Now we write Makefile.am, which is the file that the program automake reads. It is a one-liner for now.

```
SUBDIRS = src
```

The line tells automake to look at the file Makefile.am in the directory src.

2.2.2 src/Makefile.am

Next we create the file src/Makefile.am. bin_PROGRAMS is a list of programs to create. Our program is called scratchkernel.

```
bin_PROGRAMS = scratchkernel
```

Now we list the source files in scratchkernel_SOURCES.

```
scratchkernel_SOURCES = main.c \
boot.S \
link.ld
```

We specify that we are using a custom linker script that we will write called link.ld. Also, we use -Wl,--build-id=none in recent binutils so that we don't get a section for a build ID included at the start of the output file. We need the multiboot header to be the first section so that Grub will accept it.

```
scratchkernel_LDFLAGS = \
-T link.ld \
-Wl,--build-id=none
```

Fore the compiler and assembler flags, there are many options. Many packages might not even need to specify anything here. Commonly it is used to specify include directories. Because kernels are special, we need to give it a whole bunch of flags that tell it not to treat it like a normal application. The first option we give it is -m32, which tells it to build a kernel for 32-bit architecture, rather than assume that it matches the running system's architecture. This is so that we can build it on other kinds of systems but still test it with a 32-bit virtual machine.

The rest of the options are to disable advanced compiler features taht won't work because of the runtime environment. The compiler normally assumes that you are building a program to run under the environment you are currently on. That would make sense, except that a kernel will be run in an essentially blank environment, with no access to anything that isn't built into it. -nostdinc says not to include the standard header files. -nodefaultlibs says not to link to default libraries as we won't have access to them. -nostartfiles says not to use startup code around the program which the operating system normally uses to help load your program. -fno-builtin says not to use built-in definitions

of functions. They may not work and even if they do, we want to know what is going on. -fno-stack-protector disables the stack protector.

```
scratchkernel_CFLAGS = \
-m32 \
-nostdinc \
-nostartfiles \
-nodefaultlibs \
-fno-builtin \
-fno-stack-protector
scratchkernel_CCASFLAGS = -m32
```

2.3 The Linker Script

The linker script goes in the src directory and controls how our kernel is linked.

2.3.1 link.ld

Next we write the linker script, link.ld. We start by giving the entry point to our kernel, which will be start.

```
ENTRY (start)
```

Next we define some code sections. The line . = 0x100000; means that our kernel will be loaded into memory at position 1MiB in RAM. Then we define a code section, a data section, and a bss section. text is for a the code section, which is where the compiled code will go, data is the data section, which is where static data like predefined strings will go, and bss, which is where uninitialized data goes. We put a pointer end after all this. What we are doing here is defining these variables along with underscored versions of them so that in C and Assembly we can use them to refer to the sections of our kernel's memory.

Chapter 3

Minimal Source For a Bootable Kernel

3.1 The Loader

There is a small assembly file that holds the entry point to the kernel. The loader code goes in the src directory contains some code to make the kernel multiboot compliant, which allows you to boot the kernel with any multiboot compliant bootloader such as GRUB.

3.1.1 boot.S

This first part specifies stuff needed by GRUB. The align flag is flag 0 and the meminfo flag is flag 1 so we set them accordingly. Then we use a logical OR to create the variable FLAGS. The MAGIC value is checked by GRUB and should match this one. The CHECKSUM is a check that GRUB does to be sure that these FLAGS and MAGIC are real and the MAGIC value didn't just happen to be 0x1BADB002 Then we use align 4 to use 32-bit instructions. Then we create the entry mboot, which is used by GRUB and put the necessary values there.

```
.set ALIGN, 1 << 0
.set MEMINFO, 1 << 1
.set FLAGS, ALIGN | MEMINFO
.set MAGIC, Ox1BADBOO2
.set CHECKSUM, -( MAGIC + FLAGS )

.align 4
mboot:
.long MAGIC
.long FLAGS
.long CHECKSUM</pre>
```

Next we define our entry point, start.

```
.global start start:
```

When Grub 2 loads the kernel, it puts the magic value in %eax and puts a pointer to the multiboot information structure in %ebx. We push those onto the stack for parameters to main. Then we call main.

```
push %ebx
push %eax
call main
```

When the main function returns, pop the parameters off the stack as usual, and jump to halt, which is just an idle loop.

```
add $8, %esp
jmp halt
halt:
jmp halt
```

3.2 The Main

We use the standard C main.c function as our entry point. This is not the real entry point that the bootloader calls, but the stuff that happens in boot.S won't need to be looked at much, so we may regard this as our logical entry point. We take the Grub multiboot magic as the first parameter and a pointer to the multiboot information structure as the second parameter. They aren't of much use to us yet, but they will be.

3.2.1 main.c

```
int
main (unsigned int magic, multiboot_info_t *mboot)
{
}
```

3.3 Building

The command autoreconf -i will run autoconf, automake, and some other things in the proper order along with installing some necessary files. This step will have to be done whenever the list of source files changes or the build flags change.

3.3. BUILDING 13

\$ autoreconf -i

Now that the build system is in place, we may use it by running the script ./configure and then running make.

./configure

Chapter 4

Hello, World!

4.1 The Classic "Hello, World!"

Now that we have a minimal kernel that boots from GRUB, we will want it to do something. Following the great tradition, we start with "Hello, World!" But this is not as simple as it may seem. Take a look at the standard "Hello, World!" in C:

```
#include <stdio.h>
int
main (int argc, char *argv[])
{
   printf ("Hello, World!\n");
}
```

Notice that we include stdio.h. This doesn't exist in our blank system, which only has access to our kernel. Therefore, we must write it and include it in our kernel. You may or may not write printf, depending on what you decide you need. Those functions will interact with the screen to write characters. To use the screen, we will write some backend code that handles the video memory. We will also need some basic functions for handling strings, which will go in string.h.

4.2 Modifying The Build System

We explain how to modify the build system for this first time, but in the next chapters, this section will be omitted. You should learn how to modify these files yourself. First you may change the version if you wish from 0.1.0 to 0.2.0.

```
AC_INIT(scratchkernel,0.2.0,your@email.com)
```

We will be adding a directory called **include** to the package. This directory will hold the standard include files for a C library.

So we add a line with include/Makefile to our AC_OUTPUT section. (Edit the file configure.ac to look like this)

```
AC_OUTPUT([
Makefile
include/Makefile
src/Makefile
])
```

We also need to add some lines for the new source files we are going to write in src/Makefile.am.

(Edit the file src/Makefile.am to look like this)

```
scratchkernel_SOURCES = \
main.c \
boot.S \
util.S \
link.ld \
stdio.c \
string.c \
system.c \
vga.c \
...
```

Add a line about the include files to Makefile.am. Also, make sure that the main Makefile.am knows about the include directory by changing to say

```
SUBDIRS = include src
```

In src/Makefile.am we add a line to scratchkernel_CFLAGS to make sure it knows where the include files are.

```
-I../include \
```

Now we actually make a directory called include and create some files there.

4.3 Include files

4.3.1 include/Makefile.am

We add the one-line file Makefile.am to the include directory which only lists the header files we are using.

```
include_HEADERS = \
stdio.h \
stddef.h \
```

```
string.h \
system.h \
vga.h
```

4.3.2 include/stddef.h

We make include/stddef.h. Notice that it is wrapped in #ifndef STDDEF_H and #endif and defines symbol #define STDDEF_H at the top, inside of the ifndef. This is the standard way of writing C header files, so this part will not be shown in further header files. We define the symbol size_t to be an unsigned integer as sizes of types represent the number of bytes they take to store, which is always a positive number.

```
#ifndef STDDEF_H
#define STDDEF_H

#define size_t unsigned int
#endif
```

4.3.3 include/stdio.h

We include headers string.h and video.h. string.h will have functions that manipulate strings, while video.h will have the functions that manipulate the screen so that we can actually print to it.

```
#include <string.h>
int putchar (int character);
int puts (const char *str);
```

4.3.4 include/string.h

Here we have functions to copy blocks of memory and report the length of strings.

```
#include <stddef.h>
void *memcpy (void *str1, const void *str2, size_t n);
size_t strlen (const char *str);
```

4.3.5 include/system.h

Here we extend the x86 machine instructions inb and outb so that they can be used in C.

```
unsigned char inb (unsigned short port); void outb (unsigned char byte, unsigned short port);
```

4.3.6 include/vga.h

The VGA CRTC refers to VGA screen controller. It is the piece of hardware that the kernel interacts with to produce graphics in various VGA modes. We define the address part of it to be 0x3D4 and the data part to be 0x3D5. These are used later to modify the cursor. Then we define

```
#define VGA_CRTC_ADDRESS 0x3D4
#define VGA_CRTC_DATA 0x3D5
```

Then we define the high and low bytes of the cursor to be 0x0E and 0x0F. These are the addresses that are put into the address slot in the VGA controller to specify the position of the cursor.

```
#define VGA_CURSOR_HIGH 0x0E
#define VGA_CURSOR_LOW 0x0F
```

Now we define a bunch of colors so that later if we decide we want to use certain colors in our text, they are available by name.

```
#define VGA_BLACK 0
#define VGA_BLUE 1
#define VGA_GREEN 2
#define VGA_CYAN 3
#define VGA_RED 4
#define VGA_MAGENTA 5
#define VGA_BROWN 6
#define VGA_LIGHT_GRAY 7
#define VGA_DARK_GRAY 8
#define VGA_LIGHT_BLUE 9
#define VGA_LIGHT_GREEN 10
#define VGA_LIGHT_CYAN 11
#define VGA_LIGHT_RED 12
#define VGA_LIGHT_MAGENTA 13
#define VGA_LIGHT_BROWN 14
#define VGA_WHITE 15
```

Here are the functions that scroll the screen, change a character somewhere on the screen, append a character at the cursor position, set the position of the onscreen cursor, and clear the screen.

```
void vga_scroll ();
void vga_putchar_at (char c, int x, int y);
void vga_putchar (char c);
void vga_set_internal_cursor ();
void vga_clear_screen ();
```

4.4 The source files

Each source file corresponding to a header file will include the corresponding header as the first line. For example the first line of src/stdio.c is #include <stdio.h>.

4.4.1 src/stdio.c

Here is where we define the functions that print characters and strings. For putchar we just call the video driver's version.

```
int
putchar (int c)
{
  vga_putchar ((char)c);
  return c;
}
```

4.4.2 src/string.c

Here we define memcpy, which copies blocks of memory from one buffer to another, and strlen, which tells the length of a string.

```
void *
memcpy (void *destination, const void *source, size_t n)
{
   int i;
   for (i = 0; i < n; i++)
        {
            ((unsigned char*)destination)[i] = ((unsigned char*)source)[i];
      }
   return destination;
}
size_t
strlen (const char *str)
{
   size_t n = 0;
   while (str[n] != '\0') n++;
   return n;
}</pre>
```

4.4.3 src/util.S

This file is really just a wrapper for the assembly functions so that C can use the machine instruction. We need to use inb and outb to communicate with the video controller.

```
.global outb
outb:
  mov 4(%esp), %eax
  mov 8(%esp), %edx
  outb %al, %dx
  ret

.global inb
inb:
  mov $0, %eax
  mov 4(%esp), %edx
  inb %dx, %al
  ret
```

The first function puts the first parameter (the value) in %eax and puts the second parameter (the port) in %edx. Then it sends the value to the port and returns the value we were given.

For the second function, we are going to read a byte into %eax to return it so we put a zero in %eax so that the 32-bit value is cleared. Then it copies the first parameter (the port) into %edx and reads from port %dx into %al.

$4.4.4 \quad \text{src/vga.c}$

This is the big important one.

We start by defining the video memory as a pointer to 0xB8000, which is where the screen text starts when using VGA. Also, we set the cursor position to (0,0) and set the text color to be light gray on black. Colors in the default VGA mode are specified as a byte with the background color in the 4 high bits and the text color in the 4 low bits.

```
unsigned char *video = (unsigned char*)0xB8000;
unsigned int cursor_x = 0;
unsigned int cursor_y = 0;
unsigned char vga_color = VGA_BLACK << 4 | VGA_LIGHT_GRAY;</pre>
```

This function sets the position of the cursor on screen.

```
void
vga_set_internal_cursor ()
{
  unsigned temp;
  temp = cursor_y * 80 + cursor_x;
  outb (VGA_CURSOR_HIGH, VGA_CRTC_ADDRESS);
  outb (temp >> 8, VGA_CRTC_DATA);
  outb (VGA_CURSOR_LOW, VGA_CRTC_ADDRESS);
  outb (temp, VGA_CRTC_DATA);
}
```

The screen memory is an array of alternating bytes of text and colors. This function sets the specified position to the specified character.

```
void
vga_putchar_at (char c, int x, int y)
{
   video[(y * 80 + x) * 2] = c;
   video[(y * 80 + x) * 2 + 1] = vga_color;
}
```

This function goes through each screen position and sets it to a space character.

```
void
vga_clear_screen ()
{
   int i, j;
   for (i = 0; i < 80; i++)
        {
        for (j = 0; j < 25; j++)
            {
            vga_putchar_at ('\'\', i, j);
            }
      }
   cursor_x = 0;
   cursor_y = 0;
   vga_set_internal_cursor ();
}</pre>
```

This function makes sure we scroll the contents of the screen up and reset the cursor when it needs to. It calculates the number of lines past the end that the cursor is at. It should only be one at a time, but we handle it if it is more too. If we have a line past the end, we scroll everything up one line by copying lines from the top down.

```
void
vga_scroll ()
{
   int lines_past = cursor_y - 24;
   if (lines_past < 1) return;

   memcpy (video, video + lines_past * 80 * 2, 80 * (25 - lines_past) * 2);

   int i, j;
   for (i = 25 - lines_past; i < 25; i++)
        {
        for (j = 0; j < 80; j++)
            {
                  vga_putchar_at ('u', j, i);
            }
        }
}</pre>
```

```
}
cursor_y = 24;
vga_set_internal_cursor ();
}
```

This function correctly handles newline characters as well as incrementing the cursor when it's a display character and doing the newline on the screen to wrap lines that are too big.

```
void
video_putchar (char c)
  if (c == '\n')
    {
      cursor_x = 0;
      cursor_y++;
      video_scroll ();
      video_set_vga_cursor ();
      return;
    }
  vga_putchar_at (cursor_x, cursor_y);
  cursor_x++;
  if (cursor_x > 79)
      cursor_x = 0;
      cursor_y++;
    }
  vga_scroll ();
  vga_set_internal_cursor ();
```

4.4.5 src/main.c

Now that we've done all this work, we can print "Hello, World!" to the screen by repeatedly calling vga_putchar () with each character. Or if you prefer, write a function that writes strings by looping on the index and printing each character until it sees a NULL character.

4.4.6 Test

You may now rebuild and test your kernel and if all goes well, you should see the words "Hello, World!" sitting happily at the top of the screen.

Exercises

1. Write puthex (unsigned int n), which takes an unsigned integer and displays it as hexadecimal.

- 2. Write putint (int n), which takes an integer and displays it in decimal, displaying a negative sign in front if it is negative.
- 3. Complete the previous exercises and then write printf (const char *format, ...) based on them. Your printf implementation will have to parse the format string, checking for % characters. When it sees one, optionally check for characters 0-9 and . to find where to round to and how many spaces to pad with, and if the next character is %, c, d, X, s then write a % with putchar, the character itself with putchar, the integer with putint, the hexadecimal number with puthex, or the string by running through the characters until you see a NULL character.
- 4. VGA sometimes leaves by default the "blink" attribute enabled, which means that if you set the background color to have numerical value greater than or equal to 8, instead of actually taking that color, the background will be the dark version of the color and the foreground will blink. Disable the blink attribute by reading from VGA attribute controller, clearing the bit that enables the blink attribute, and writing back to the attribute controller.

The attribute controller works in a funny way. When you write a byte to it, it flip-flops between storing the address and storing data to the predetermined address. To reset the flip-flop, read from the input status #1, 0x3DA, and store it so that you can rewrite it later. Then write the address of the mode control register, 0x10 to the VGA attribute controller, 0x3CO. Now read the data from the attribute controller's read port, 0x3C1. Clear bit 3 (zero-based, i.e. 1 << 3) and re-write the byte to the VGA attribute controller, 0x3CO. Now, finally re-write the value that was in the input status #1 register.

5. Implement the previous exercise, and use it to make a surface structure that is made up of a two dimensional array of shorts, each holding a byte for the color and a byte for the character. Write functions to draw these surfaces to the screen, so that you may store images, draw them on a backbuffer, and then draw the backbuffer to the screen in a while loop. This is known as page-flipping.

Chapter 5

Interrupts

5.1 The Global Descriptor Table and Interrupt Descriptor Table

The next step in developing a useful kernel is handling interrupts. To do this, we take use of descriptor tables.

The Global Descriptor Table has a list of segment descriptors that describe segments. A segment is a piece of the 4GB address space. The GDT describes where each segment is, how big it is, what permissions it has, and other properties

The Interrupt Descriptor Table is similar to the GDT but instead of listing segment descriptors, it lists interrupt descriptors (also known as interrupt gates). They tell where in memory the CPU should jump to on certain interrupt numbers.

5.2 The GDT

5.2.1 include/gdt.h

The Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A contains information on the Global Descriptor Table [?].

In include/gdt.h we only need to declare a function init_gdt, which will be called by main to initialize the Global Descriptor Table.

void init_gdt ();

$5.2.2 \quad \text{src/gdt.c}$

We include arrays of bytes for the GDT entries and for the GDT pointer structure. Each GDT entry is 8 bytes long. We make room for as many as we need. The GDT pointer structure is 6 bytes long.

```
#define NUM_GDT_ENTRIES 3
unsigned char gdt_entries[8 * NUM_GDT_ENTRIES];
unsigned char gdt_pointer[6];
```

Now we make a function for writing GDT entries. This is actually a bit more general than necessary because it lets limits for segments be any value and decides what the granularity is based on how big the limits are. We will always specify the entire 32-bit address space for any segment selector we use, so the granularity is also always 1, meaning that the limit that we put in the 20-bit limit field is always the number of 4KB blocks (in practice, always 0xFFFFF). The "access" byte controls a lot of properties of the segment selector. You can see how the basic parts that make up this selector format (base, limit, etc.) are all chopped up in strange ways. This function chops them up the way that the processor expects them and sticks them in the array.

```
void
write_gdt_entry (int n, unsigned int base, unsigned int limit, int code,
                 int dpl)
 int granularity = 1;
 if (limit \geq 0 \times 10000)
      granularity = 1;
      limit >> 12;
    }
 unsigned char access = 1 << 7 | /* present */
                        dpl << 5 | /* descriptor privilege level */
                          1 << 4 | /* always 1 */
                       code << 3 | /* executable bit */</pre>
                          0 << 2 | /* direction */
                           1 << 1 | /* whether a code segment can be read
                                     * or a data segment can be written */
                          0 << 0; /* whether it has been accessed */
 unsigned char flags = granularity << 3 | /* either 1B (0) or 4KB (1) */
                                   1 << 2; /* 32-bit segment */
 unsigned char *gdt_entry = (unsigned char*)gdt_entries + (n*8);
 gdt_entry[0] = limit & 0x00FF;
 gdt_entry[1] = (limit & 0xFF00) >> 8;
 gdt_entry[2] = base & 0x00FF;
 gdt_entry[3] = (base & 0xFF00) >> 8;
 gdt_entry[4] = (base & 0x00FF0000) >> 16;
 gdt_entry[5] = access;
 gdt_entry[6] = flags << 4 | (limit & 0x000F0000) >> 16;
 gdt_entry[7] = (base & 0xFF000000) >> 24;
```

Now we make a function for writing the GDT pointer structure.

5.3. THE IDT 27

```
void
write_gdt_pointer ()
{
  unsigned int limit = 8 * NUM_GDT_ENTRIES - 1;
  gdt_pointer[0] = limit & 0x00FF;
  gdt_pointer[1] = (limit & 0xFF00) >> 8;
  gdt_pointer[2] = ((unsigned int)gdt_entries & 0x00000FF);
  gdt_pointer[3] = ((unsigned int)gdt_entries & 0x0000FF00) >> 8;
  gdt_pointer[4] = ((unsigned int)gdt_entries & 0x000FF0000) >> 16;
  gdt_pointer[5] = ((unsigned int)gdt_entries & 0xFF000000) >> 24;
}
```

Finally, we write <code>init_gdt</code>, which will make use of our earlier functions. First we clear the array of GDT entries. This leaves zeros in bytes 0-7, or the first GDT entry, which is good. The first GDT entry should always be all zeros for the NULL descriptor. Next we write a GDT entry for code, and a GDT entry for data, each from 0-0xFFFFFFFF with ring 0 for kernel mode (as opposed to user mode). The reason we do this once for code and once for data is that the register <code>%cs</code> must contain the byte offset in this array for a code segment selector and the registers <code>%ds</code>, <code>%es</code>, <code>%fs</code>, <code>%gs</code> must contain the byte offset in this array for a data segment selector. So later on to specify the code segment or the data segment we will use 0x8 for GDT entry number 1 (the code one) or 0x10 for GDT entry number 2 (the data one). Then we write the GDT pointer structure and load it into the processor.

```
void
init_gdt ()
{
  memset (gdt_entries, 0, 8 * NUM_GDT_ENTRIES);
  write_gdt_entry (1, 0, 0xFFFFFFFF, 1, 0); /* #1, 0-FFFFFFFF, code, ring 0 */
  write_gdt_entry (2, 0, 0xFFFFFFFF, 0, 0); /* #2, 0-FFFFFFFF, data, ring 0 */
  write_gdt_pointer ();
  load_gdt (&gdt_pointer);
}
```

5.3 The IDT

5.3.1 src/irqs.S

What we are going to do is make a function for the CPU to call for each type of interrupt that it can detect. What we want is for our interrupt functions to look like this

```
void
my_interrupt (unsigned int *eax,
```

```
unsigned int *ebx,
unsigned int *ecx,
unsigned int *edx,
unsigned int *ebp,
unsigned int *esi,
unsigned int *edi,
unsigned int *eip)
{
```

where the parameters are all of the registers that we will have access to. We will simulate that it goes directly to these interrupt functions by creating a layer between the actual interrupt and our pseudo-interrupts. We will set up stub functions for the interrupts, which will take care of some common steps before calling our interrupt functions which look like the above.

When an interrupt occurs, the processor switches to the mode with the greatest privileges and jumps to the location specified by the IDT. We will set it up to go to our stub functions, which we will write now.

For each stub function, we push a zero error code and the interrupt number. It also calls irq_handler, which will call our real handler if we have one. We write a GAS macro, irq n, which creates a function that does several things:

- disable interrupts
- push the zero error code
- push the interrupt number
- push a whole bunch of register values
- call irq_handler
- clean up
- re-enables interrupts.

```
.macro irq n
.global irq\n
irq\n :
    cli
    push $0
    push $\n
    push %eax
    push %ebx
    push %ecx
    push %edx
    push %ebp
    push %esi
    push %edi
    call irq_handler
```

5.3. THE IDT 29

```
pop %edi
  pop %esi
  pop %ebp
  pop %edx
  pop %ecx
  pop %ebx
  pop %eax
  add $0x8, %esp
  iret
.endm
irq 0
irq 1
irq 2
irq 3
irq 4
irq 5
irq 6
irq 7
irq 8
irq 9
irq 10
irq 11
irq 12
irq 13
irq 14
irq 15
```

We also need to make a stub function for every Interrupt Service Routine. These are the interrupt numbers that correspond to faults. These are handled exactly the same as the IRQs except that 8, 10, 11, 12, 13, and 14 push their own error code. So we don't push the zero for these interrupts. Also we call <code>isr_handler</code> instead of <code>irq_handler</code>. This time we need two macros, one for the interrupt number that push their own error code, and one for the numbers that don't.

5.3.2 src/isrs.s

```
.macro isr n
.global isr\n
isr\n :
    cli
    push $0
    push $\n
    jmp isr_common
.endm

.macro isr_has_own_error_code n
.global isr\n
```

```
isr\n :
   cli
   push $\n
   jmp isr_common
.endm
```

Both of these macros create functions named \mathtt{isrN} where N is the number of the interrupt, but one pushes a zero and the other doesn't. Now we use create the $\mathtt{isr_common}$ function to push the registers, call $\mathtt{isr_handler}$, clean up, and re-enable interrupts.

```
isr_common:
 push %eax
 push %ebx
 push %ecx
 push %edx
 push %ebp
 push %esi
 push %edi
 call isr_handler
 pop %edi
 pop %esi
 pop %ebp
 pop %edx
 pop %ecx
 pop %ebx
 pop %eax
 add $0x8, %esp
 iret
```

Now we create all the stub functions with our macros.

```
isr 0
isr 1
isr 2
isr 3
isr 4
isr 5
isr 6
isr 7
isr_has_own_error_code 8
isr 9
isr_has_own_error_code 10
{\tt isr\_has\_own\_error\_code} \ 11
isr_has_own_error_code 12
isr_has_own_error_code 13
isr_has_own_error_code 14
isr 15
isr 16
isr 17
isr 18
```

5.3. THE IDT 31

```
isr 19
isr 20
isr 21
isr 22
isr 23
isr 24
isr 25
isr 26
isr 27
isr 28
isr 29
isr 30
isr 31
```

5.3.3 include/idt.h

EOI refers to the End Of Interrupt signal, which is 0x20.

```
#define EOI 0x20
```

These represent the CPU I/O ports for the Programmable Interrupt Controller. There are command and data ports for the slave and master. They take a command on the command port followed by some data on the data port.

```
#define INTERRUPT_PIT 32
#define INTERRUPT_KEYBOARD 33
```

We also give names to the ports for the Programmable Interrupt Controller. There are ports for Master and Slave to each get commands, and ports for Master and Slave to each get data.

```
#define MASTER_PIC_COMMAND 0x20
#define SLAVE_PIC_COMMAND 0xA0
#define MASTER_PIC_DATA 0x21
#define SLAVE_PIC_DATA 0xA1
```

We declare a function init_idt, which will setup our Interrupt Descriptor Table. We will call this from main.

```
void init_idt ();
```

Now we make sure we can access the irq stub functions and the isr stub functions that we made in irqs.s and isrs.s

```
void irq0 ();
void irq1 ();
void irq2 ();
void irq3 ();
void irq4 ();
void irq5 ();
```

```
void irq6 ();
void irq7 ();
void irq8 ();
void irq9 ();
void irq10 ();
void irq11 ();
void irq12 ();
void irq13 ();
void irq14 ();
void irq15 ();
void isr0 ();
void isr1 ();
void isr2 ();
void isr3 ();
void isr4 ();
void isr5 ();
void isr6 ();
void isr7 ();
void isr8 ();
void isr9 ();
void isr10 ();
void isr11 ();
void isr12 ();
void isr13 ();
void isr14 ();
void isr15 ();
void isr16 ();
void isr17 ();
void isr18 ();
void isr19 ();
void isr20 ();
void isr21 ();
void isr22 ();
void isr23 ();
void isr24 ();
void isr25 ();
void isr26 ();
void isr27 ();
void isr28 ();
void isr29 ();
void isr30 ();
void isr31 ();
```

The Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A contains information on the Interrupt Descriptor Table.

5.3. THE IDT 33

5.3.4 include/system.h

In include/system.h we put a declaration of function outb, which sends a byte on a given port.

```
void outb (unsigned char byte, unsigned short port);
```

$5.3.5 \quad \text{src/idt.c}$

We include arrays of bytes for the IDT entries and for the IDT pointer structure. Each IDT entry is 8 bytes long. We make room for 256 of them just in case we want to use more later. If you know for sure how many you need, you may optimize this value. The IDT pointer structure is 6 bytes long.

```
unsigned char idt_entries[8 * 256];
unsigned char idt_pointer[6];
```

Now we write the IDT pointer structure. We set the limit to the index of the last byte of the IDT entries array. Then we set bytes 0 and 1 to be the low and high byte of the limit. Next we set bytes 2-5 to bytes 0-3 of the address to idt_entries array.

```
void
write_idt_pointer ()
{
  unsigned int limit = 256 * 8 - 1;
  idt_pointer[0] = limit & 0x00FF;
  idt_pointer[1] = limit & 0xFF00 >> 8;
  idt_pointer[2] = ((unsigned int)idt_entries & 0x00000FF);
  idt_pointer[3] = ((unsigned int)idt_entries & 0x0000FF00) >> 8;
  idt_pointer[4] = ((unsigned int)idt_entries & 0x00FF0000) >> 16;
  idt_pointer[5] = ((unsigned int)idt_entries & 0xFF000000) >> 24;
}
```

Each entry specifies where to jump to when the interrupt of matching index occurs. The first and second bytes are the low two bytes of the address that the interrupt number should jump to. The third and fourth bytes are the segment selector, which is the byte index to the Global Descriptor Table of the segment descriptor to use. The fifth byte is always zero. The sixth byte is the "access" byte, which is determined by the given table. The seventh and eighth bytes are the high two bytes of the address that the interrupt number should jump to.

7	6 5	4	3	2	1	0
Р	DPL	0	Ex	Conf	RW	Accessed

First we clear the array of interrupt handlers. Then we send the proper commands to remap IRQs lines 0-15 to IDT lines 32-47, whose details are still a mystery to me. Then we clear the array of IDT entries. Next we write all of our IDT entries, using the corresponding stub function for the base, the code segment selector (byte array index 8 for GDT entry of index 1), and ring 0 for kernel mode (as opposed to user mode). Then we write the IDT pointer structure and load it into the processor.

```
void
init_idt ()
 memset ((unsigned char*)interrupt_handlers, 0, 256 * 4);
 /* Some sort of initialization byte for the master and slave pic */
  outb (0x11, MASTER_PIC_COMMAND);
  outb (0x11, SLAVE_PIC_COMMAND);
  /* Send offset to each */
  /st Start master pic interrupts at 32 and slave pic interrupts at 40 st/
  outb (0x20, MASTER_PIC_DATA);
  outb (0x28, SLAVE_PIC_DATA);
  /* More initialization */
  outb (0x04, MASTER_PIC_DATA);
  outb (0x02, SLAVE_PIC_DATA);
  /* More initialization */
  outb (0x01, MASTER_PIC_DATA);
  outb (0x01, SLAVE_PIC_DATA);
  /* More initialization */
  outb (0x00, MASTER_PIC_DATA);
```

5.3. THE IDT 35

```
outb (0x00, SLAVE_PIC_DATA);
memset (&idt_entries, 0, 256 * 8);
write_idt_entry (0, (unsigned int)isr0, 0x08, 0);
write_idt_entry (1, (unsigned int)isr1, 0x08, 0);
write_idt_entry (2, (unsigned int)isr2, 0x08, 0);
write_idt_entry (3, (unsigned int)isr3, 0x08, 0);
write_idt_entry (4, (unsigned int)isr4, 0x08, 0);
write_idt_entry (5, (unsigned int)isr5, 0x08, 0);
write_idt_entry (6, (unsigned int)isr6, 0x08, 0);
write_idt_entry (7, (unsigned int)isr7, 0x08, 0);
write_idt_entry (8, (unsigned int)isr8, 0x08, 0);
write_idt_entry (9, (unsigned int)isr9, 0x08, 0);
write_idt_entry (10, (unsigned int)isr10, 0x08, 0);
write_idt_entry (11, (unsigned int)isr11, 0x08, 0);
write_idt_entry (12, (unsigned int)isr12, 0x08, 0);
write_idt_entry (13, (unsigned int)isr13, 0x08, 0);
write_idt_entry (14, (unsigned int)isr14, 0x08, 0);
write_idt_entry (15, (unsigned int)isr15, 0x08, 0);
write_idt_entry (16, (unsigned int)isr16, 0x08, 0);
write_idt_entry (17, (unsigned int)isr17, 0x08, 0);
write_idt_entry (18, (unsigned int)isr18, 0x08, 0);
write_idt_entry (19, (unsigned int)isr19, 0x08, 0);
write_idt_entry (20, (unsigned int)isr20, 0x08, 0);
write_idt_entry (21, (unsigned int)isr21, 0x08, 0);
write_idt_entry (22, (unsigned int)isr22, 0x08, 0);
write_idt_entry (23, (unsigned int)isr23, 0x08, 0);
write_idt_entry (24, (unsigned int)isr24, 0x08, 0);
write_idt_entry (25, (unsigned int)isr25, 0x08, 0);
write_idt_entry (26, (unsigned int)isr26, 0x08, 0);
write_idt_entry (27, (unsigned int)isr27, 0x08, 0);
write_idt_entry (28, (unsigned int)isr28, 0x08, 0);
write_idt_entry (29, (unsigned int)isr29, 0x08, 0);
write_idt_entry (30, (unsigned int)isr30, 0x08, 0);
write_idt_entry (31, (unsigned int)isr31, 0x08, 0);
write_idt_entry (32, (unsigned int)irq0, 0x08, 0);
write_idt_entry (33, (unsigned int)irq1, 0x08, 0);
write_idt_entry (34, (unsigned int)irq2, 0x08, 0);
write_idt_entry (35, (unsigned int)irq3, 0x08, 0);
write_idt_entry (36, (unsigned int)irq4, 0x08, 0);
write_idt_entry (37, (unsigned int)irq5, 0x08, 0);
write_idt_entry (38, (unsigned int)irq6, 0x08, 0);
write_idt_entry (39, (unsigned int)irq7, 0x08, 0);
write_idt_entry (40, (unsigned int)irq8, 0x08, 0);
write_idt_entry (41, (unsigned int)irq9, 0x08, 0);
write_idt_entry (42, (unsigned int)irq10, 0x08, 0);
write_idt_entry (43, (unsigned int)irq11, 0x08, 0);
write_idt_entry (44, (unsigned int)irq12, 0x08, 0);
```

```
write_idt_entry (45, (unsigned int)irq13, 0x08, 0);
  write_idt_entry (46, (unsigned int)irq14, 0x08, 0);
  write_idt_entry (47, (unsigned int)irq15, 0x08, 0);
  write_idt_pointer ();
  load_idt (&idt_pointer);
}
void
isr_handler (unsigned int edi,
             unsigned int esi,
             unsigned int ebp,
             unsigned int edx,
             unsigned int ecx,
             unsigned int ebx,
             unsigned int eax,
             unsigned int n,
             unsigned int error_code,
             unsigned int eip)
  /* If there is a handler installed for this fault, then run it. Otherwise
   * halt */
  int (*handler)() = (int(*)())interrupt_handlers[n];
  if (handler)
   {
      handler(&eax, &ebx, &ecx, &edx, &ebp, &esi, &edi, &eip);
   }
  else
   {
     halt ();
    }
}
void
irq_handler (unsigned int edi,
             unsigned int esi,
             unsigned int ebp,
             unsigned int edx,
             unsigned int ecx,
             unsigned int ebx,
             unsigned int eax,
             unsigned int n,
             unsigned int error_code,
             unsigned int eip)
  /* If there is a handler installed for this interrupt, then run it */
  int (*handler)() = (int(*)())interrupt_handlers[n+32];
  if (handler)
```

5.4. THE TIMER 37

```
handler(&eax, &ebx, &ecx, &edx, &ebp, &esi, &edi, &eip);
outb (EOI, MASTER_PIC_COMMAND);
}
```

5.4 The Timer

5.4.1 include/timer.h

5.4.2 src/timer.c

We make a spinner by putting the current <code>spinner_char</code> on the top right corner of the screen and then changing it to the next character in the sequence - \ | / each time timer is called.

```
char spinner_char = '-';
int
timer (unsigned int *eax,
       unsigned int *ebx,
       unsigned int *ecx,
       unsigned int *edx,
       unsigned int *ebp,
       unsigned int *esi,
       unsigned int *edi,
       unsigned int *eip)
  switch (spinner_char)
    {
      case '-':
        spinner_char = '\\';
        break;
      case '\\':
        spinner_char = '|';
        break;
      case '|':
        spinner_char = '/';
        break;
```

```
case '/':
    spinner_char = '-';
    break;
}
vga_putchar_at (spinner_char, 79, 0);
}
```

Now you can install the timer interrupt by setting the proper interrupt number to this function. i.e. somewhere in main, interrupt_handlers[INTERRUPT_PIT] = (unsigned int)

5.5 The Keyboard

5.5.1 include/keyboard.h

This file mostly defines a bunch of scancodes so that we can more easily reference the keys or check what kind of key they are. It also defines the port to read data from on a keyboard interrupt.

```
#define KEYBOARD_DATA 0x60
#define KEY_0 11
#define KEY_1 2
#define KEY_2 3
#define KEY_3 4
#define KEY_4 5
#define KEY_5 6
#define KEY_6 7
#define KEY_7 8
#define KEY_8 9
#define KEY_9 10
#define KEY_HYPHEN 12
#define KEY_EQUALS 13
#define KEY_BACKSPACE 14
#define KEY_TAB 15
#define KEY_q 16
#define KEY_w 17
#define KEY_e 18
#define KEY_r 19
#define KEY_t 20
#define KEY_y 21
#define KEY_u 22
#define KEY_i 23
#define KEY_o 24
#define KEY_p 25
#define KEY_LEFT_SQUARE_BRACKET 26
#define KEY_RIGHT_SQUARE_BRACKET 27
#define KEY_ENTER 28
#define KEY_a 30
#define KEY_s 31
```

```
#define KEY_d 32
#define KEY_f 33
#define KEY_g 34
#define KEY_h 35
#define KEY_j 36
#define KEY_k 37
#define KEY_1 38
#define KEY_SEMICOLON 39
#define KEY_SINGLE_QUOTE 40
#define KEY_BACK_QUOTE 41
#define KEY_BACKSLASH 43
#define KEY_z 44
#define KEY_x 45
#define KEY_c 46
#define KEY_v 47
#define KEY_b 48
#define KEY_n 49
#define KEY_m 50
#define KEY_COMMA 51
#define KEY_PERIOD 52
#define KEY_SLASH 53
#define KEY_ASTERISK 55
#define KEY_SPACE 57
#define KEY_MINUS 74
#define KEY_PLUS 78
#define KEY_NULL 0
#define KEY_ESCAPE 1
#define KEY_CONTROL 29
#define KEY_LSHIFT 42
#define KEY_RSHIFT 54
#define KEY_ALT 56
#define KEY_CAPS_LOCK 58
#define KEY_F1 59
#define KEY_F2 60
#define KEY_F3 61
#define KEY_F4 62
#define KEY_F5 63
#define KEY_F6 64
#define KEY_F7 65
#define KEY_F8 66
#define KEY_F9 67
#define KEY_F10 68
#define KEY_F11 87
#define KEY_F12 88
#define KEY_NUM_LOCK 69
#define KEY_SCROLL_LOCK 70
#define KEY_HOME 71
#define KEY_END 79
#define KEY_UP 72
```

```
#define KEY_DOWN 80
#define KEY_LEFT 75
#define KEY_RIGHT 77
#define KEY_PAGE_UP 73
#define KEY_PAGE_DOWN 81
#define KEY_INSERT 82
#define KEY_DELETE 83
#define KEY_UNDEFINEDO 76
#define KEY_UNDEFINED1 84
#define KEY_UNDEFINED2 85
#define KEY_UNDEFINED3 86
#define KEY_UNDEFINED4 89
#define KEY_UNDEFINED5 90
#define KEY_UNDEFINED6 91
#define KEY_UNDEFINED7 92
#define KEY_UNDEFINED8 93
#define KEY_UNDEFINED9 94
#define KEY_UNDEFINED10 95
#define KEY_UNDEFINED11 96
#define KEY_UNDEFINED12 97
#define KEY_UNDEFINED13 98
#define KEY_UNDEFINED14 99
#define KEY_UNDEFINED15 100
#define KEY_UNDEFINED16 101
#define KEY_UNDEFINED17 102
#define KEY_UNDEFINED18 103
#define KEY_UNDEFINED19 104
#define KEY_UNDEFINED20 105
#define KEY_UNDEFINED21 106
#define KEY_UNDEFINED22 107
#define KEY_UNDEFINED23 108
#define KEY_UNDEFINED24 109
#define KEY_UNDEFINED25 110
#define KEY_UNDEFINED26 111
#define KEY_UNDEFINED27 112
#define KEY_UNDEFINED28 113
#define KEY_UNDEFINED29 114
#define KEY_UNDEFINED30 115
#define KEY_UNDEFINED31 116
#define KEY_UNDEFINED32 117
#define KEY_UNDEFINED33 118
#define KEY_UNDEFINED34 119
#define KEY_UNDEFINED35 120
#define KEY_UNDEFINED36 121
#define KEY_UNDEFINED37 122
#define KEY_UNDEFINED38 123
#define KEY_UNDEFINED39 124
#define KEY_UNDEFINED40 125
#define KEY_UNDEFINED41 126
#define KEY_UNDEFINED42 127
```

We also include the function prototypes of functions scancode_to_char and key_is_character.

```
char scancode_to_char (unsigned char scancode);
int key_is_character (unsigned char scancode);
```

5.5.2 src/keyboard.c

Our keyboard interrupt gets the scancode from the keyboard data port, checks if the key was released or pressed, and if it was pressed, prints the corresponding character. The high bit is set on a key release. In that case, we just strip off the high bit so that later if we want to add to this function, we have the actual scancode. If the high bit isn't set and it was a press instead of a release, we convert from scancode to character and print the character.

```
keyboard (unsigned int *eax,
          unsigned int *ebx,
          unsigned int *ecx,
          unsigned int *edx,
          unsigned int *ebp,
          unsigned int *esi,
          unsigned int *edi,
          unsigned int *eip)
{
  unsigned char scancode = inb (KEYBOARD_DATA);
  if (scancode & 0x80)
      scancode &= 0x7F;
    }
  else
    {
      if (key_is_character (scancode))
          char c = scancode_to_char (scancode);
          putchar (c);
    }
}
```

Unfortunately, there is no nice way to determine what the character is for a scancode or whether a scancode corresponds to a character. We make use of our defines from the include file:

```
char
scancode_to_char (unsigned char scancode)
{
```

```
switch (scancode)
      case KEY_0:
        return '0';
        break;
      case KEY_1:
        return '1';
        break;
      case KEY_z:
        return 'z';
        break;
      case KEY_ENTER:
        return '\n';
        break;
      case KEY_SPACE:
        return '∟';
        break;
      default:
        return '?';
        break;
    }
}
```

You might also want to have the backspace scancode return 0x8 for the backspace character and have your putchar handle this character correctly.

Again, making further use of our defines, we just return 1 for every scan code that has a character associated with it and return 0 otherwise.

```
int
key_is_character (unsigned char scancode)
{
    switch (scancode)
    {
        case KEY_0:
            return 1;
            break;
        case KEY_1:
            return 1;
            break;
        ...
        default:
        return 0;
        break;
}
```

5.5.3 src/main.c

```
int
main (int argc, char *argv[])
{
   cli ();
   vga_clear_screen ();
   if (magic != 0x2BADB002)
        {
        puts ("Invalid_Multiboot_Magic");
        return 1;
        }
   init_gdt ();
   init_idt ();
   interrupt_handlers[INTERRUPT_PIT] = (unsigned int)timer;
   interrupt_handlers[INTERRUPT_KEYBOARD] = (unsigned int)keyboard;
   asm volatile ("sti");
}
```

5.5.4 Test

You may now rebuild and test your kernel. You should see a spinner at the top-right corner of the screen.

System Calls

- 6.1 User Mode
- 6.1.1 Jumping to User Mode

[?].

6.2 System Calls

Memory Management

7.1 Page Tables

Paging provides a way to deal with multiple processes using the same RAM.

7.2 Include files

7.2.1 kheap.h

7.2.2 paging.h

Page frames are 0x1000 or 4096 bytes long. init_paging is the function that main will call to enable paging.

7.3 Source Files

7.3.1 kheap.c

7.3.2 paging.c

The control register cr3 holds the location of the page directory. So we load the page directory address into cr3. Then we read cr0, and set bit 31. This is the bit that determines whether paging is in effect. After setting this bit in our copy of cr0, we write cr0 back to the CPU to really enable paging.

7.3.3 main.c

7.3.4 Test

When you compile and run this version, it should not appear any different than it was before. If you want to test that the page fault works, you may put a reference to a high address that is not mapped and try to print it.

Multitasking

8.1 Tasks

Grub Modules

- 9.1 Grub Modules
- 9.2 ELF

The C Library

I found it easiest and most useful to modify GCC in order to have a full cross-compiler. That way, we will be able to build packages that use the autotools by simply specifying which host to build for in the configure script. This is done by making a few modifications to binutils and gcc, according to the article "OS Specific Toolchain" on OSDev.org [?].

I used newlib-1.18.0, released on December 17, 2009.

You should use the version of binutils and gcc that are installed on your system so that you know you will be able to install any dependencies that GCC has. For example, GMP and MPFR are required by recent GCC releases, so if you use the version of GCC that is installed on your system, you should be able to install the development packages of GMP and MPFR without any issues and your cross GCC will link correctly to the libraries that are already installed. You can find out what version of GCC is installed with the command

gcc --version

You only need gcc-core if you are only using C.

I used binutils-2.20.1, released on March 3, 2010.

We install the cross tools in /usr/scratch.

10.1 binutils

10.1.1 config.sub

In the basic system types, we add "scratch" to the list:

```
# First accept the basic system types.
# The portable systems comes first.
# Each alternative MUST END IN A *, to match a version number.
# -sysv* is not here because it comes later, after sysvr4.
-scratch* | -gnu* | -bsd* | -mach* | -minix* | -genix* | -ultrix* | -irix* \
```

10.1.2 bfd/config.bfd

In the section that starts case "\${targ}" in after the line # START OF targmatch.h, we add

```
i[3-7]86-*-scratch*)
targ_defvec=bfd_elf32_i386_vec
targ_selvecs=i386coff_vec
;;
```

10.1.3 gas/configure.tgt

In the section that starts case \${generic_target} in, we add i386-*-scratch*)
fmf=elf ;;.

10.1.4 ld/configure.tgt

In the section that starts # Please try to keep this table in alphabetic order, we add an entry for scratch.

10.1.5 ld/emulparams/scratch_i386.sh

For the script that defines some parameters for the emulation by copying the one for elf_i386.

```
cp ld/emulparams/elf_i386.sh ld/emulparams/scratch_i386.sh
```

10.1.6 ld/Makefile.in

Here we add a Makefile rule that tells how to make a C file for our emulation like this:

10.2. GCC 55

```
escratch_i386.c: $(srcdir)/emulparams/scratch_i386.sh \
$(ELF_DEPS) $(srcdir)/scripttempl/elf.sc ${GEN_DEPENDS}
${GENSCRIPTS} scratch_i386 "$(tdir_scratch_i386)"
```

Make sure that last line starts with a tab character rather than spaces.

10.2 gcc

10.2.1 config.sub

In the basic system types, we add "scratch" to the list:

```
# First accept the basic system types.
# The portable systems comes first.
# Each alternative MUST END IN A *, to match a version number.
# -sysv* is not here because it comes later, after sysvr4.
-scratch* | -gnu* | -bsd* | -mach* | -minix* | -genix* | -ultrix* | -irix* \
```

10.2.2 gcc/config.gcc

Where it says # Common parts for widely ported systems., we add a section for scratch to the target list:

```
case ${target} in
*-*-scratch*)
extra_parts="crtbegin.o crtend.o"
gas=yes
gnu_ld=yes
default_use_cxa_atexit=yes
;;
```

Also add a section where it says # Support site-specific machine types.:

```
# Support site-specific machine types.
i[3-7]86-*-scratch*)
tm_file="${tm_file} i386/unix.h i386/att.h dbxelf.h elfos.h i386/i386elf.h scratch.h"
tmake_file="i386/t-i386elf t-svr4"
use_fixproto=yes
;;
```

10.2.3 gcc/config/scratch.h

We create a header file which sets some stuff in gcc for setting the default system name, etc.

10.2.4 libgcc/config.host

In the section where it says # Support site-specific machine types., add a section for scratch:

```
case ${host} in
# Support site-specific machine types.
i[3-7]86-*-scratch*)
;;
```

Make sure this is a tab character.

10.3 newlib

10.3.1 config.sub

In the basic system types, we add "scratch" to the list:

```
# First accept the basic system types.
# The portable systems comes first.
# Each alternative MUST END IN A *, to match a version number.
# -sysv* is not here because it comes later, after sysvr4.
-scratch* | -gnu* | -bsd* | -mach* | -minix* | -genix* | -ultrix* | -iri
```

10.3.2 newlib/configure.host

Where it says # Get the source directories to use for the host. we add a section. Note that this is not the very similarly named section, # Get the source directories to use for the CPU type..

```
# Get the source directories to use for the host. unix_dir is set

# to unix to get some standard Unix routines. posix_dir is set to get some

# standard Posix routines. sys_dir should supply system dependent routines

# including crt0.

# THIS TABLE IS ALPHA SORTED. KEEP IT THAT WAY.

case "${host}" in
  i[3-7]86-*-scratch*)
  sys_dir=scratch
;;
```

10.3. NEWLIB 57

10.3.3 newlib/libc/sys/configure.in

Here we add a section to tell it to configure the directory scratch:

```
if test -n "${sys_dir}"; then
  case ${sys_dir} in
    scratch) AC_CONFIG_SUBDIRS(scratch) ;;
```

Now we run autoconf in the newlib/libc/sys directory.

```
cd newlib/libc/sys autoconf
```

10.3.4 newlib/libc/sys/scratch

Now we create a directory specifically for our kernel, including startup code, system call implementations, and the build system for it all.

crt0.S

This file will be linked when programs are built to run on our kernel.

```
.global _start
.extern main
.extern exit
_start:
  call main
  call exit
halt:
  jmp halt
```

syscalls-util.S

```
.global exit_util
exit_util:
  push %ebp
mov %esp, %ebp

# stack from ebp: (0) oldebp, (4) returnaddress, (8) n

push %eax
push %ebx
mov 8(%ebp), %eax
mov %eax, %ebx
mov $0x1, %eax
int $0x80
pop %ebx
pop %eax
pop %ebp
```

exit_util (0);

```
ret
.global write_util
write_util:
  push %ebp
  mov %esp, %ebp
  # stack from ebp: (0) oldebp, (4) returnaddress, (8) fd, (12) buf, (16) count
  push %eax
  push %ebx
  push %ecx
  push %edx
  mov 8(%ebp), %eax
  mov %eax, %ebx
  mov 12(%ebp), %eax
  mov %eax, %ecx
  mov 16(\%ebp), \%eax
  mov %eax, %edx
  mov $0x4, %eax
  int $0x80
  pop %edx
  pop %ecx
  pop %ebx
  pop %eax
  pop %ebp
  ret
.global getpid_util
getpid_util:
  mov $0x14, %eax
  int $0x80
 ret
syscalls.c
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/fcntl.h>
#include <sys/times.h>
#include <sys/errno.h>
#include <sys/time.h>
#include <stdio.h>
void
_exit ()
{
```

10.3. NEWLIB 59

```
}
int
close (int file)
{
char **environ;
execve (char *name, char **argv, char **env)
}
int
fork ()
{
}
int
fstat (int file, struct stat *st)
}
int
getpid ()
return getpid_util ();
int
isatty (int file)
{
}
kill (int pid, int sig)
}
link (char *old, char *new)
{
int
lseek (int file, int ptr, int dir)
}
```

```
int
open (const char *name, int flags, ...)
int
read (int file, char *ptr, int len)
{
}
caddr_t
sbrk (int incr)
}
stat (const char *file, struct stat *st)
{
}
clock_t
times (struct tms *buf)
}
int
unlink (char *name)
int
wait (int *status)
{
}
write (int file, char *ptr, int len)
 write_util (file, ptr, len);
 return len;
}
{\tt gettimeofday (struct\ timeval\ *p,\ void\ *z)}
{
}
```

10.4. BUILDING 61

```
AC_PREREQ(2.59)
AC_INIT([newlib], [NEWLIB_VERSION])
AC_CONFIG_SRCDIR([crt0.S])
AC_CONFIG_AUX_DIR(../../..)
NEWLIB_CONFIGURE(../../..)
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Makefile.am

```
AUTOMAKE_OPTIONS = cygnus
INCLUDES = $(NEWLIB_CFLAGS) $(CROSS_CFLAGS) $(TARGET_CFLAGS)
AM_CCASFLAGS = $(INCLUDES)
noinst_LIBRARIES = lib.a
if MAY_SUPPLY_SYSCALLS
extra_objs = $(lpfx)syscalls.o $(lpfx)syscalls-util.o
extra_objs =
endif
lib_a_SOURCES =
lib_a_LIBADD = $(extra_objs)
EXTRA_lib_a_SOURCES = syscalls.c syscalls.S crt0.S
lib_a_DEPENDENCIES = $(extra_objs)
lib_a_CCASFLAGS = $(AM_CCASFLAGS)
lib_a_CFLAGS = $(AM_CFLAGS)
if MAY_SUPPLY_SYSCALLS
all: crt0.o
endif
ACLOCAL_AMFLAGS = -I ../../..
CONFIG_STATUS_DEPENDENCIES = $(newlib_basedir)/configure.host
```

Now we run autoreconf in the directory newlib/libc/sys/scratch.

 ${\tt autoreconf}$

10.4 Building

We make a directory to build binutils in and then we enter it and configure for our kernel. Then we build.

```
mkdir build-binutils
cd build-binutils
```

```
../binutils-2.20.1/configure --target=i586-pc-scratch --prefix=/usr/scratch make make install
```

Now add the directory to your path so that we can use the build tools we created:

```
export PATH=$PATH:/usr/scratch/bin
```

We do similarly for gcc except that when configuring, we also disable native language support and enable C.

```
mkdir build-gcc
cd build-gcc
../gcc-4.4.1/configure --target=i586-pc-scratch --prefix=/usr/scratch --disable-make
make install
```

Now we build newlib:

```
mkdir build-newlib
cd build-newlib
../newlib-1.18.0/configure --target=i586-pc-scratch --prefix=/usr/scratch
make
make install
```

Then since the configure script will run some tests and it needs crtbegin.o to work, let's create a blank one:

```
cd ~
touch crtbegin.S
i586-pc-scratch-as -o crtbegin.o crtbegin.S
cp crtbegin.o /usr/scratch/i586-pc-scratch/lib/
```

Appendix A

Tips for kernel development

A.1 Debugging

Write a function that displays the hex values of bytes at a particular address. Then use it to compare when you know things are working to a troublesome spot. For example, when dealing with paging, use it to see what the byte values are in a particular address (physical or virtual, depending on what mode you are in) and see if it is what you expect. Do this when still in physical mode on the physical address of the code so that you can compare.

A.2 objdump

objdump -d src/hello

will tell you what assembly code has been generated for this program.

objdump -x src/hello

will give you a list of program sections and

objdump --help

to find out more options.