jamesm-tutorials Documentation

Release 2.0.0

James Molloy

October 17, 2013

CONTENTS

Contents:

CHAPTER

ONE

ENVIRONMENT SETUP

We need a base from which to design and make our kernel. Here I will be assuming that you are using a *nix system, with the GNU toolchain. If you want to use a windows system, you must either use cygwin (which is a *nix emulation environment) or DJGPP. Either way, the makefiles and commands in this tutorial may not work.

Please note that if you use cygwin, you may need to make your own GCC cross compiler

1.1 Compiling

All examples in this tutorial should compile with the GNU toolchain (gcc, ld) with the exception of the assembly examples which are written in Intel syntax. These require the Netwide Assembler.

This tutorial is *not* a bootloader tutorial. We will be using GRUB - the grand unified bootloader - to load our kernel. To do this, we need a floppy disk image with GRUB preloaded onto it. There are tutorials out there to do this, but I have made a standard image which can be found `here`. This is a 1.44MB floppy disk image formatted with the ext2 filesystem with GRUB installed.

1.2 Running

There is no alternative for bare hardware as a testbed system. Unfortunately, bare hardware is rather poor at telling you where things went wrong. Enter Bochs. Bochs is an open-source x86-64 emulator, similar to Qemu but with more debugging features for first-time kernel developers.

When things go pear-shaped, bochs will tell you and store the processor state in a logfile, which can be extremely useful. Also it can be run and rebooted much faster than a real machine. My examples will be made to run well on bochs.

Bochs can be a little slow at times, so an alternative is to use QEmu. QEmu uses dynamic binary translation (similar to just-in-time compilation) to achieve much faster speeds than bochs, however as it doesn't maintain the processor state precisely for every instruction executed it cannot tell you as easily (or precisely) what state it was in when a crash occurred.

1.2.1 Bochs

Bochs requires a configuration file. Historically these had to be rather large and unportable, but the following two lines work fine with the Bochs packaged with Ubuntu 10.04:

```
floppya: 1_44=floppy.img, status=inserted
log: bochsout.txt
```

This will tell Bochs to look for a floopy disk image file called "floppy.img", mount it and output logging information to the file "bochsout.txt".

1.2.2 Qemu

Running QEmu is substantially simpler:

```
qemu -fda floppy.img
```

1.3 Build environment

1.3.1 Link.ld

Todo

Simplify linker script and copypaste in here

1.3.2 Makefile

Note: In makefiles any indented regions are indented by one HARD tab. Spaces will not work.

```
CSOURCES=$(shell find -name *.c)
COBJECTS=$(patsubst %.c, %.o, $(CSOURCES))
SSOURCES=$(shell find -name *.s)
SOBJECTS=$(patsubst %.s, %.o, $(SSOURCES))

CC=gcc
LD=ld
CFLAGS=-nostdlib -fno-builtin -m32
LDFLAGS=-melf_i386 -Tlink.ld
ASFLAGS=-felf

all: $(COBJECTS) $(SOBJECTS) link update

bochs:
        bash scripts/run_bochs.sh

update:
        @echo Updating floppy image
        @bash scripts/update_image.sh

clean:
        @echo Removing object files
        @-rm $(COBJECTS) $(SOBJECTS) kernel

link:
        @echo Linking
        @$(LD) $(LDFLAGS) -o kernel $(SOBJECTS) $(COBJECTS)

.s.o:
```

```
        @echo Assembling $<
        @nasm $(ASFLAGS) $<

.c.o:
        @echo Compiling $<
        @$(CC) $(CFLAGS) -o $@ -c $<
```

This Makefile will compile every C and assembly file in the current directory and all subdirectories, and link them together into one ELF binary called 'kernel'. It uses the linker script above (Link.ld)

1.3.3 update_image.sh

This script will mount a floppy disk image as a drive and poke your new kernel into it.

```
#!/bin/bash

sudo losetup /dev/loop0 floppy.img
sudo mount /dev/loop0 /mnt
sudo cp src/kernel /mnt/kernel
sudo umount /dev/loop0
sudo losetup -d /dev/loop0
```

Note: You will need /sbin in your $PATH to use losetup.

1.3.4 run_bochs.sh

This is a small wrapper script around bochs' command line;

```
#!/bin/bash
bochs -f bochsrc.txt
```

CHAPTER

TWO

GENESIS

2.1 The boot code

OK, It's time for some code! Although the brunt of our kernel will be written in C, there are certain things that we just *must* use assembly for. One of those is the initial boot code.

Here we go:

```
;
; boot.s -- Kernel start location. Also defines multiboot header.
;           Based on Bran's kernel development tutorial file start.asm
;

MBOOT_PAGE_ALIGN    equ 1<<0    ; Load kernel and modules on a page boundary
MBOOT_MEM_INFO      equ 1<<1    ; Provide your kernel with memory info
MBOOT_HEADER_MAGIC  equ 0x1BADB002 ; Multiboot Magic value
; NOTE: We do not use MBOOT_AOUT_KLUDGE. It means that GRUB does not
; pass us a symbol table. GRUB will also require our kernel be in elf32 format.
MBOOT_HEADER_FLAGS  equ MBOOT_PAGE_ALIGN | MBOOT_MEM_INFO
MBOOT_CHECKSUM      equ -(MBOOT_HEADER_MAGIC + MBOOT_HEADER_FLAGS)

bits 32                         ; All instructions should be 32-bit.

section .text

global mboot                    ; Make 'mboot' accessible from C.
extern code
extern bss
extern end

mboot:
    dd  MBOOT_HEADER_MAGIC      ; GRUB will search for this value on each
                                ; 4-byte boundary in the first 8KB of your kernel file.
    dd  MBOOT_HEADER_FLAGS      ; How GRUB should load your file / settings
    dd  MBOOT_CHECKSUM          ; To ensure that the above values are correct
    dd mboot
          dd  code              ; Start of kernel '.text' (code) section.
          dd  bss               ; End of kernel '.data' section.
          dd  end               ; End of kernel.
          dd  start             ; Kernel entry point (initial EIP).


global start:function start.end-start ; Kernel entry point.
extern kernel_main                    ; This is the entry point of our C code
```

```
start:
    cli                         ; Disable interrupts.
    mov esp, stack              ; Set up our own stack.
    push ebx                        ; Push a pointer to the multiboot info structure.
    mov ebp, 0                  ; Initialise the base pointer to zero so we can
                                ; terminate stack traces here.
    call kernel_main            ; call our main() function.
    jmp $                       ; Enter an infinite loop, to stop the processor
.end:                           ; from executing whatever rubbish is in the memory
                                ; after our kernel!

section .bss
    resb 32768
stack:
```

2.2 Understanding the boot code

There's actually only a few lines of code in that snippet:

```
cli
mov esp, stack
push ebx
mov  ebp, 0
call main
jmp $
```

We'll come back to that later. The rest of the file has to to with the *multiboot header*.

2.2.1 Multiboot

Multiboot is a standard to which GRUB expects a kernel to comply. It is a way for the bootloader to

1. Know exactly what environment the kernel wants/needs when it boots.
2. Allow the kernel to query the environment it is in.

So, for example, if your kernel needs to be loaded in a specific VESA mode (which is a bad idea, by the way), you can inform the bootloader of this and it can take care of it for you.

To make your kernel multiboot compatible you need to add a header structure somewhere in your kernel (Actually, the header must be in the first 8KB of the kernel). Usefully, there is a NASM command that lets us embed specific constants in our code - 'dd'. These lines:

```
dd  MBOOT_HEADER_MAGIC      ; GRUB will search for this value on each
                            ; 4-byte boundary in the first 8KB of your kernel file
dd  MBOOT_HEADER_FLAGS      ; How GRUB should load your file / settings
dd  MBOOT_CHECKSUM          ; To ensure that the above values are correct
```

do just that. The MBOOT_* constants are defined above.

MBOOT_HEADER_MAGIC A magic number. This identifies the kernel as multiboot-compatible.

MBOOT_HEADER_FLAGS A field of flags. We ask for GRUB to page-align all kernel sections (MBOOT_PAGE_ALIGN) and also to give us some memory information (MBOOT_MEM_INFO). Note that some tutorials also use MBOOT_AOUT_KLUDGE. As we are using the ELF file format, this hack is not necessary, and adding it stops GRUB giving you your symbol table when you boot up.

MBOOT_CHECKSUM This field is defined such that when the magic number, the flags and this are added together, the total must be zero. It is for error checking.

On bootup, GRUB will load a pointer to another information structure into the EBX register. This can be used to query the environment GRUB set up for us.

Taken directly from the multiboot specification, the information structure looks like this:

```
typedef struct
{
  uint32_t flags;
  uint32_t mem_lower;
  uint32_t mem_upper;
  uint32_t boot_device;
  uint32_t cmdline;
  uint32_t mods_count;
  uint32_t mods_addr;
  uint32_t num;
  uint32_t size;
  uint32_t addr;
  uint32_t shndx;
  uint32_t mmap_length;
  uint32_t mmap_addr;
  uint32_t drives_length;
  uint32_t drives_addr;
  uint32_t config_table;
  uint32_t boot_loader_name;
  uint32_t apm_table;
  uint32_t vbe_control_info;
  uint32_t vbe_mode_info;
  uint32_t vbe_mode;
  uint32_t vbe_interface_seg;
  uint32_t vbe_interface_off;
  uint32_t vbe_interface_len;
} __attribute__((packed)) multiboot_t;
```

We'll explain fields as we come to them in later chapters, but important to note is that I've used standard C typedefs for 32-bit unsigned integers. We don't get these typedefs as standard with the CFLAGS we use, so we have to define them ourselves. This will help with writing portable and readable code.

```
// common.h -- Defines typedefs and some global functions.
//             From JamesM's kernel development tutorials.

#ifndef COMMON_H
#define COMMON_H

// Some standard typedefs, to standardise sizes across platforms.
// These typedefs are written for 32-bit X86.
typedef unsigned int   uint32_t;
typedef          int   int32_t;
typedef unsigned short uint16_t;
typedef          short int16_t;
typedef unsigned char  uint8_t;
typedef          char  int8_t;

#endif // COMMON_H
```

Back to the code again...

So, immediately on bootup, the asm snippet tells the CPU to disable interrupts, allocate us a stack at known location, push the contents of EBX onto the stack and clear the stack frame pointer (this is to help debuggers and stack tracers - more in a later chapter). It then calls a function called "main", then enter an infinite loop.

Note: Technically Multiboot specification requires bootloader to disable interrupts before passing control to your code, but we do it anyway to be on the safe side.

All is good, but the code won't link yet. We haven't defined main()!

2.2.2 Adding some C code

Interfacing C code and assembly is easy. You just need to know the calling convention used. GCC on x86 uses the cdecl calling convention:

- All parameters to a function are passed on the stack.
- The parameters are pushed *right-to-left*.
- The return value of a function is in EAX.
- The caller cleans up the stack after the call.

... so the function call:

```
d = func(a, b, c);
```

Becomes

```
push [c]
push [b]
push [a]
call func
mov [d], eax
add esp, 12     ; The caller (us) must clean up the stack.
```

So you can see that in our asm snippet above, that 'push ebx' is actually passing a parameter to the function 'main()'

The C code

```
// main.c -- Defines the C-code kernel entry point, calls initialisation routines.
//           Made for JamesM's tutorials <www.jamesmolloy.co.uk>
#include "multiboot.h"
#include "common.h"

int main(multiboot_t *mboot_ptr)
{
  // All our initialisation calls will go in here.
  return 0xdeadbeef;
}
```

Here's our first incarnation of the main() function. As you can see, we've made it take just one parameter - a pointer to a multiboot struct.

All the function does is return a constant - 0xdeadbeef. That constant is unusual enough that it should stand out at you when we run the program in a second.

2.2.3 Compiling, linking and running!

OK, you should now be able to compile, link and run your kernel!

```
make clean  # Removes all temporary files. Ignore any errors here.
make        # Compiles, assembles and links, then updates the floppy disk image.
make bochs  # This will run bochs for you.
```

That should cause bochs to boot, you'll see GRUB for a few seconds then the kernel will run. It doesn't actually *do* anything, so it'll just freeze, saying 'starting up...'.

If you open bochsout.txt, at the bottom you should see something like:

```
00074621500i[CPU  ] | EAX=deadbeef  EBX=0002d000  ECX=0001edd0 EDX=00000001
00074621500i[CPU  ] | ESP=00067ec8  EBP=00067ee0  ESI=00053c76 EDI=00053c77
00074621500i[CPU  ] | IOPL=0 id vip vif ac vm rf nt of df if tf sf zf af pf cf
00074621500i[CPU  ] | SEG selector     base    limit G D
00074621500i[CPU  ] | SEG sltr(index|ti|rpl)     base    limit G D
00074621500i[CPU  ] |  CS:0008( 0001| 0|  0) 00000000 000fffff 1 1
00074621500i[CPU  ] |  DS:0010( 0002| 0|  0) 00000000 000fffff 1 1
00074621500i[CPU  ] |  SS:0010( 0002| 0|  0) 00000000 000fffff 1 1
00074621500i[CPU  ] |  ES:0010( 0002| 0|  0) 00000000 000fffff 1 1
00074621500i[CPU  ] |  FS:0010( 0002| 0|  0) 00000000 000fffff 1 1
00074621500i[CPU  ] |  GS:0010( 0002| 0|  0) 00000000 000fffff 1 1
00074621500i[CPU  ] | EIP=00100027 (00100027)
00074621500i[CPU  ] | CR0=0x00000011 CR1=0 CR2=0x00000000
00074621500i[CPU  ] | CR3=0x00000000 CR4=0x00000000
00074621500i[CPU  ] >> jmp .+0xfffffffe (0x00100027) : EBFE
```

Notice what the value of EAX is? 0xdeadbeef - the return value of main(). Congratulations, you now have a multiboot compatible assembly trampoline, and you're ready to start printing to the screen!

Sample code for this tutorial can be found `here`

CHAPTER

THREE

THE SCREEN

So, now that we have a 'kernel' that can run and stick itself into an infinite loop, it's time to get something interesting appearing on the screen. Along with serial I/O, the monitor will be your most important ally in the debugging battle.

3.1 The theory

Your kernel gets booted by GRUB in text mode. That is, it has access to a framebuffer (area of memory) that controls a screen of characters (not pixels) 80 wide by 25 high. This will be the mode your kernel will operate in until your get into the world of graphics (which will not be covered in this tutorial).

The area of memory known as the framebuffer is accessible just like normal RAM, at address *0xB8000*. It is important to note, however, that it is *not* actually normal RAM. It is part of the VGA controller's dedicated video memory that has been memory-mapped via hardware into your linear address space. This is an important distinction.

The framebuffer is just an array of 16-bit words, each 16-bit value representing the display of one character. The offset from the start of the framebuffer of the word that specifies a character at position *x*, *y* is:

```
(y * 80 + x) * 2
```

What's important to note is that the '* 2' is there only because each element is 2 bytes (16 bits) long. If you're indexing an array of 16-bit values, for example, your index would just be y*80+x.

In [extended] ASCII (unicode is not supported in text mode), 8 bits are used to represent a character. That gives us 8 more bits which are unused. The VGA hardware uses these to designate foreground and background colours (4 bits each). The splitting of this 16-bit value is shown in the diagram below.

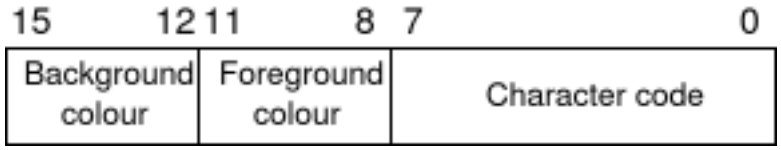


Figure 3.1: Word format

4 bits for a colour code gives us 15 possible colours we can display:

Index	Colour
0	black
1	blue
2	green
3	cyan
4	red
5	magenta
6	brown
7	light grey
8	dark grey
9	light blue
10	light green
11	light cyan
12	light red
13	light magenta
14	light brown / yellow
15	white

The VGA controller also has some ports on the main I/O bus, which you can use to send it specific instructions. (Among others) it has a control register at 0x3D4 and a data register at 0x3D5. We will use these to instruct the controller to update its cursor position.

3.2 The practice

3.2.1 First things first

Firstly, we need a few more commonly-used global functions. common.c and common.h include functions for writing to and reading from the I/O bus. They are also the ideal place to put functions such as memcpy/memset etc. I have left them for you to implement! :-)

```
// src/common.h
void outb(uint16_t port, uint8_t value);
uint8_t inb(uint16_t port);
uint16_t inw(uint16_t port);

// common.c -- Defines some global functions.
//             From JamesM's kernel development tutorials.

#include "common.h"

// Write a byte out to the specified port.
void outb(uint16_t port, uint8_t value)
{
  asm volatile ("outb %1, %0" : : "dN" (port), "a" (value));
}

uint8_t inb(uint16_t port)
{
  uint8_t ret;
  asm volatile("inb %1, %0" : "=a" (ret) : "dN" (port));
  return ret;
}

uint16_t inw(uint16_t port)
```

```
{
  uint16_t ret;
  asm volatile ("inw %1, %0" : "=a" (ret) : "dN" (port));
  return ret;
}
```

3.2.2 The monitor code

A simple header file:

```
// monitor.h -- Defines the interface for monitor code
//              From JamesM's kernel development tutorials.

#ifndef MONITOR_H
#define MONITOR_H

#include "common.h"

// Write a single character out to the screen.
void monitor_put(char c);

// Clear the screen to all black.
void monitor_clear();

// Output a null-terminated ASCII string to the monitor.
void monitor_write(char *c);

// Output a hex value to the monitor.
void monitor_write_hex(uint32_t n);

// Output a decimal value to the monitor.
void monitor_write_dec(uint32_t n);

#endif // MONITOR_H
```

3.2.3 Moving the cursor

To move the hardware cursor, we must firstly work out the linear offset of the x,y cursor coordinate. We do this by using the equation above. Next, we have to send this offset to the VGA controller. For some reason, it accepts the 16-bit location as two bytes. We send the controller's command port (0x3D4) the command 14 to tell it we are sending the high byte, then send that byte to port 0x3D5. We then repeat with the low byte, but send the command 15 instead.

```
static void move_cursor()
{
  // The screen is 80 characters wide...
  uint16_t cursorLocation = cursor_y * 80 + cursor_x;
  outb(0x3D4, 14);                  // Tell the VGA board we are setting the high cursor byte.
  outb(0x3D5, cursorLocation >> 8); // Send the high cursor byte.
  outb(0x3D4, 15);                  // Tell the VGA board we are setting the low cursor byte.
  outb(0x3D5, cursorLocation);      // Send the low cursor byte.
}
```

3.2.4 Scrolling the screen

At some point we're going to fill up the screen with text. It would be nice if, when we do that, the screen acted like a terminal and scrolled up one line. Actually, this really isn't very difficult to do:

```
static void scroll()
{
  // Get a space character with the default colour attributes.
  uint8_t attributeByte = (0 /*black*/ << 4) | (15 /*white*/ & 0x0F);
  uint16_t blank = 0x20 /* space */ | (attributeByte << 8);

  // Row 25 is the end, this means we need to scroll up
  if(cursor_y >= 25)
  {
    // Move the current text chunk that makes up the screen
    // back in the buffer by a line
    int i;
    for (i = 0*80; i < 24*80; i++)
      video_memory[i] = video_memory[i+80];

    // The last line should now be blank. Do this by writing
    // 80 spaces to it.
    for (i = 24*80; i < 25*80; i++)
      video_memory[i] = blank;

    // The cursor should now be on the last line.
    cursor_y = 24;
  }
}
```

3.2.5 Writing a character to the screen

Now the code gets a little more complex. But, if you look at it, you'll see that most of it is logic as to where to put the cursor next - there really isn't much difficult there.

```
void monitor_put(char c)
{
  // The background colour is black (0), the foreground is white (15).
  uint8_t backColour = 0;
  uint8_t foreColour = 15;

  // The attribute byte is made up of two nibbles - the lower being the
  // foreground colour, and the upper the background colour.
  uint8_t attributeByte = (backColour << 4) | (foreColour & 0x0F);
  // The attribute byte is the top 8 bits of the word we have to send to the
  // VGA board.
  uint16_t attribute = attributeByte << 8;

  // Handle a backspace, by moving the cursor back one space
  if (c == 0x08 && cursor_x)
    cursor_x--;

  // Handle a tab by increasing the cursor's X, but only to a point
  // where it is divisible by 8.
  else if (c == 0x09)
    cursor_x = (cursor_x+8) & ~(8-1);
```

```
  // Handle carriage return
  else if (c == '\r')
    cursor_x = 0;

  // Handle newline by moving cursor back to left and increasing the row
  else if (c == '\n')
  {
    cursor_x = 0;
    cursor_y++;
  }

  // Handle any other printable character.
  else if(c >= ' ')
  {
      video_memory[cursor_y*80 + cursor_x] = c | attribute;
      cursor_x++;
  }

  // Check if we need to insert a new line because we have reached the end
  // of the screen.
  if (cursor_x >= 80)
  {
      cursor_x = 0;
      cursor_y ++;
  }

  // Scroll the screen if needed.
  scroll();
  // Move the hardware cursor.
  move_cursor();
}
```

See? It's pretty simple! The bit that actually does the writing is here:

```
video_memory[cursor_y*80 + cursor_x] = c | attribute;
```

This sets the 16-bit word at the cursor position to be the logical-OR of the character to write and the attribute. Remember that we shifted attribute left by 8 bits above, so we're actually just setting the lower 8 bits of this word to 'c'.

3.2.6 Clearing the screen

Clearing the screen is also dead easy. Just fill it with loads of spaces:

```
void monitor_clear()
{
  // Make an attribute byte for the default colours
  uint8_t attributeByte = (0 /*black*/ << 4) | (15 /*white*/ & 0x0F);
  uint16_t blank = 0x20 /* space */ | (attributeByte << 8);

  int i;
  for (i = 0; i < 80*25; i++)
    video_memory[i] = blank;

  // Move the hardware cursor back to the start.
  cursor_x = 0;
  cursor_y = 0;
```

```
  move_cursor();
}
```

3.2.7 Writing a string

```
void monitor_write(char *c)
{
    while (*c)
      monitor_put(*c++);
}
```

Summary

If you put all that code together, you can add a couple of lines to your main.c file:

```
monitor_clear();
monitor_write("Hello, world!");
```

Et voila - a text output function. Not bad for a couple of minutes' work!

Extensions

Apart from implementing memcpy/memset/strlen/strcmp etc, there are a few other functions that will make life easier for you.

```
void monitor_write_hex(uint32_t n)
{
   // TODO: implement this yourself!
}

void monitor_write_dec(uint32_t n)
{
   // TODO: implement this yourself!
}
```

The function names should be pretty self explanatory – writing in hexadecimal really is required if you're going to check the validity of pointers. Decimal is optional but it's nice to see something in base 10 every once in a while!

You could also have a look at the linux 0.1 code - that has an implementation of vsprintf which is quite neat and tidy. You could copy that function then use it to implement printf(), which will make your life a lot easier when it comes to debugging.

Source code for this tutorial is available `here`

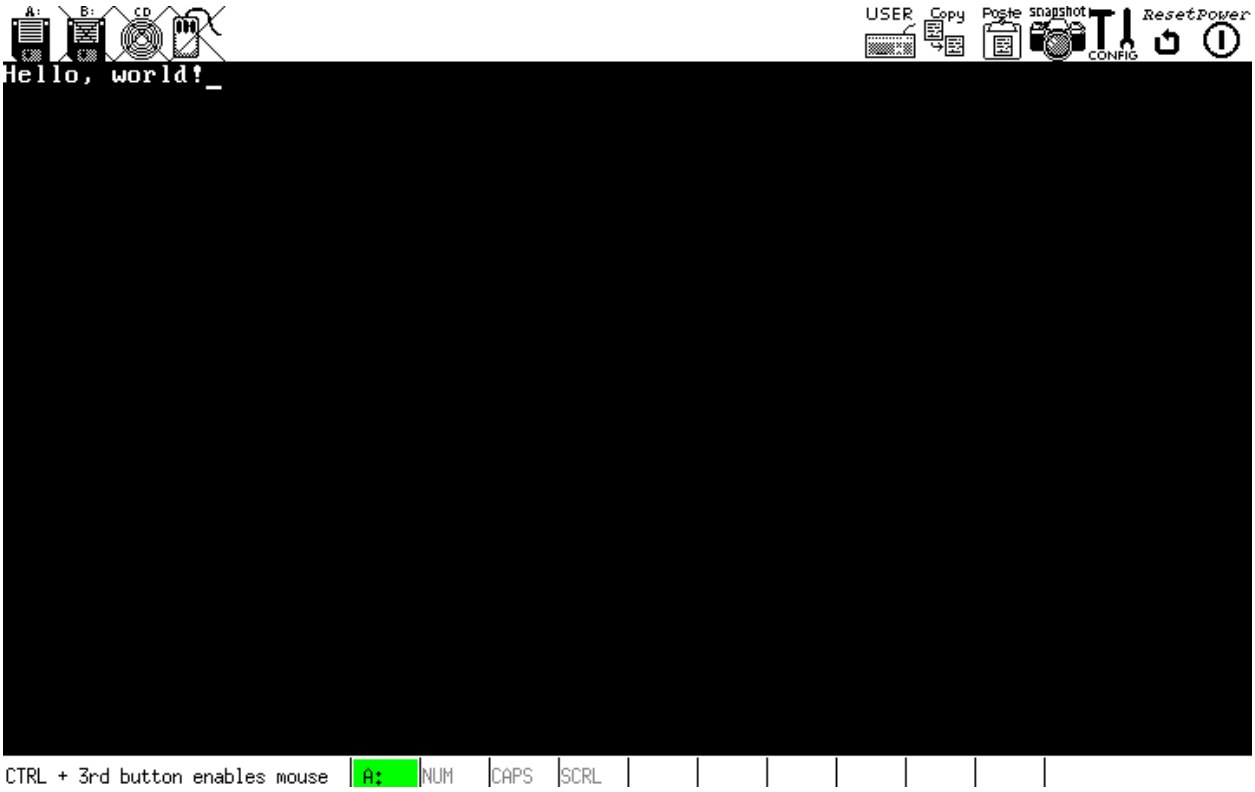


Figure 3.2: Hello, world!

CHAPTER

FOUR

THE GDT AND IDT

The GDT and the IDT are descriptor tables. They are arrays of flags and bit values describing the operation of either the segmentation system (in the case of the GDT), or the interrupt vector table (IDT).

They are, unfortunately, a little theory-heavy, but bear with it because it'll be over soon!

4.1 The Global Descriptor Table

4.1.1 Theory

The x86 architecture has two methods of memory protection and of providing virtual memory - segmentation and paging.

With segmentation, every memory access is evaluated with respect to a segment. That is, the memory address is added to the segment's base address, and checked against the segment's length. You can think of a segment as a window into the address space - the process does not know it's a window, all it sees is a linear address space starting at zero and going up to the segment length.

With paging, the address space is split into (usually 4KB, but this can change) blocks, called pages. Each page can be mapped into physical memory - mapped onto what is called a 'frame'. Or, it can be unmapped. Like this you can create virtual memory spaces.

Both of these methods have their advantages, but paging is much better. Segmentation is, although still usable, fast becoming obsolete as a method of memory protection and virtual memory. In fact, the x86-64 architecture requires a flat memory model (one segment with a base of 0 and a limit of 0xFFFFFFFF) for some of it's instructions to operate properly.

Segmentation is, however, totally in-built into the x86 architecture. It's impossible to get around it. So here we're going to show you how to set up your own Global Descriptor Table - a list of segment descriptors.

As mentioned before, we're going to try and set up a flat memory model. The segment's window should start at 0x00000000 and extend to 0xFFFFFFFF (the end of memory). However, there is one thing that segmentation can do that paging can't, and that's *set the ring level*.

A ring is a privilege level - zero being the most privileged, and three being the least. Processes in ring zero are said to be running in *kernel-mode*, or *supervisor-mode*, because they can use instructions like *sti* and *cli*, something which most processes can't. Normally, rings 1 and 2 are unused. They can, technically, be allowed to access a greater subset of the supervisor-mode instructions than ring 3 can. Some microkernel architectures use these for running *server processes*, or drivers.

A segment descriptor carries inside it a number representing the ring level it applies to. To change ring levels (which we'll do later on), among other things, we need segments that represent both ring 0 and ring 3.

4.1.2 Practical

OK, that was one humongous chunk of theory, lets get into the nitty gritty of implementing this.

One thing I forgot to mention is that GRUB sets a GDT up for you. The problem is that, similar to GRUB's stack, you don't know where that GDT is, or what's in it. So you could accidentally overwrite it, then your computer would triple-fault and reset. Not clever.

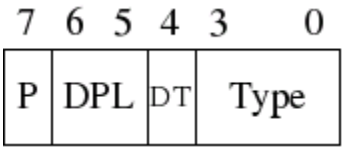
In the x86, we have 6 segmentation registers. Each holds an offset into the GDT. They are CS (code segment), DS (data segment), ES (extra segment), FS, GS, SS (stack segment). The code segment *must* reference a descriptor which is set as a 'code segment'. There is a flag for this in the access byte. The rest should all reference a descriptor which is set as a 'data segment'.

gdt.h

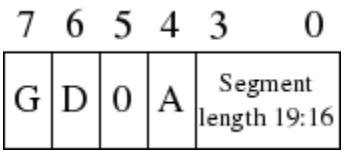
A GDT entry looks like this:

```
// This structure contains the value of one GDT entry.
// We use the attribute "packed" to tell GCC not to change
// any of the alignment in the structure.
typedef struct
{
  uint16_t limit_low;     // The lower 16 bits of the limit.
  uint16_t base_low;      // The lower 16 bits of the base.
  uint8_t base_middle;    // The next 8 bits of the base.
  uint8_t access;         // Access flags, determines what ring this segment can be used in.
  uint8_t granularity;
  uint8_t base_high;      // The last 8 bits of the base.
} __attribute__((packed)) gdt_entry_t;
```

Most of those fields should be self-explanatory. The format of the access byte is given here:



and the format of the granularity byte is here:



P Is segment present? (1 = Yes)

DPL Descriptor privilege level - Ring 0 - 3.

DT Descriptor type

Type Segment type - code segment / data segment.

G Granularity (0 = 1 byte, 1 = 4kbytes)

D Operand size (0 = 16bit, 1 = 32bit)

0 Should always be zero.

A Available for system use (always zero).

To tell the processor where to find our GDT, we have to give it the address of a special pointer structure:

```
// This struct describes a GDT pointer. It points to the start of
// our array of GDT entries, and is in the format required by the
// lgdt instruction.
typedef struct
{
    uint16_t limit;       // The Global Descriptor Table limit.
    uint32_t base;        // The address of the first gdt_entry_t struct.
} __attribute__((packed)) gdt_ptr_t;
```

The base is the address of the first entry in our GDT, the limit being the size of the table minus one (the last valid address in the table).

Those struct definitions should go in a header file, gdt.h, along with a prototype.

```
// Initialisation function.
void init_gdt();
```

gdt.c

In gdt.c, we have a few declarations:

```
//
// gdt.c - Initialises the GDT and IDT, and defines the
//         default ISR and IRQ handler.
//         Based on code from Bran's kernel development tutorials.
//         Rewritten for JamesM's kernel development tutorials.
//

#include "common.h"
#include "gdt.h"

// Lets us access our ASM functions from our C code.
extern void gdt_flush (uint32_t);

// Internal function prototypes.
static void gdt_set_gate (int32_t,uint32_t,uint32_t,uint8_t,uint8_t);

// The GDT itself.
gdt_entry_t gdt_entries [3];

// Pointer structure to give to the CPU.
gdt_ptr_t gdt_ptr;
```

Notice the gdt_flush function - this will be defined in an ASM file, and will load our GDT pointer for us.

```
void init_gdt ()
{
  // The limit is the last valid byte from the start of the GDT - i.e. the size of the GDT - 1.
  gdt_ptr.limit = sizeof (gdt_entry_t) * 3 - 1;
  gdt_ptr.base = (uint32_t) &gdt_entries;

  gdt_set_gate (0, 0, 0, 0, 0);              // Null segment.
  gdt_set_gate (1, 0, 0xFFFFF, 0x9A, 0xCF); // Code segment.
  gdt_set_gate (2, 0, 0xFFFFF, 0x92, 0xCF); // Data segment.

  // Inform the CPU about our GDT.
```

```
  gdt_flush ((uint32_t) &gdt_ptr);
}

static void gdt_set_gate(int32_t num, uint32_t base, uint32_t limit, uint8_t access, uint8_t gran)
{
    gdt_entries[num].base_low    = (base & 0xFFFF);
    gdt_entries[num].base_middle = (base >> 16) & 0xFF;
    gdt_entries[num].base_high   = (base >> 24) & 0xFF;

    gdt_entries[num].limit_low   = (limit & 0xFFFF);
    gdt_entries[num].granularity = (limit >> 16) & 0x0F;

    gdt_entries[num].granularity |= gran & 0xF0;
    gdt_entries[num].access      = access;
}
```

Lets just analyse that code for a moment. init_gdt initially sets up the gdt pointer structure - the limit is the size of each gdt entry * 3 - we have 3 entries. Why 3? well, we have a code and data segment descriptor for the kernel, and a null entry. This *must* be present, or Bad Things will happen.

gdt_init then sets up the 3 descriptors, by calling gdt_set_gate. gdt_set_gate just does some severe bit-twiddling and shifting, and should be self-explanatory with a hard stare at it. Notice that the only thing that changes between the 2 segment descriptors is the access byte - 0x9A, 0x92. You can see, if you map out the bits and compare them to the format diagram above, the bits that are changing are the type field. Type specifies whether the segment is a code segment or a data segment (the processor checks this often, and can be the source of much frustration).

Finally, we have our ASM function that will write the GDT pointer.

gdt_s.s

```
%if CHAPTER >= 4
                                    ;
; gdt.s -- contains global descriptor table setup code.
;          Based on code from Bran's kernel development tutorials.
;          Rewritten for JamesM's kernel development tutorials.

global gdt_flush:function gdt_flush.end-gdt_flush ; Allows the C code to call gdt_flush().

gdt_flush:
    mov eax, [esp+4]  ; Get the pointer to the GDT, passed as a parameter.
    lgdt [eax]        ; Load the new GDT pointer

    mov ax, 0x10      ; 0x10 is the offset in the GDT to our data segment
    mov ds, ax        ; Load all data segment selectors
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax
    jmp 0x08:.flush   ; 0x08 is the offset to our code segment: Far jump!
.flush:
    ret
.end:
%endif ; CHAPTER >= 4
```

This function takes the first parameter passed to it (in esp+4), loads the value it points to into the GDT (using the lgdt instruction), then loads the segment selectors for the data and code segments. Notice that each GDT entry is 8 bytes, and the kernel code descriptor is the second segment, so it's offset is 0x08. Likewise the kernel data descriptor is the

third, so it's offset is 16 = 0x10. Here we move the value 0x10 into the data segment registers DS,ES,FS,GS,SS. To change the code segment is slightly different; we must do a far jump. This changes the CS implicitly.

4.2 The Interrupt Descriptor Table

4.2.1 Theory

There are times when you want to interrupt the processor. You want to stop it doing what it is doing, and force it to do something different. An example of this is when an timer or keyboard interrupt request (IRQ) fires. An interrupt is like a POSIX signal - it tells you that something of interest has happened. The processor can register 'signal handlers' (interrupt handlers) that deal with the interrupt, then return to the code that was running before it fired. Interrupts can be fired externally, via IRQs, or internally, via the 'int n' instruction. There are very useful reasons for wanting to fire interrupts from software, but that's for another chapter!

The *Interrupt Descriptor Table* tells the processor where to find handlers for each interrupt. It is very similar to the GDT. It is just an array of entries, each one corresponding to an interrupt number. There are 256 possible interrupt numbers, so 256 must be defined. If an interrupt occurs and there is no entry for it (even a NULL entry is fine), the processor will panic and reset.

Faults, traps and exceptions

The processor will sometimes need to signal your kernel. Something major may have happened, such as a divide-by-zero, or a page fault. To do this, it uses the first 32 interrupts. It is therefore doubly important that all of these are mapped and non-NULL - else the CPU will triple-fault and reset (bochs will panic with an 'unhandled exception' error).

The special, CPU-dedicated interrupts are shown below.

ID	Description
0	Division by zero exception
1	Debug exception
2	Non maskable interrupt
3	Breakpoint exception
4	Overflow / INTO instruction
5	Out of bounds exception
6	Invalid opcode exception
7	No coprocessor exception
8	Double fault *(pushes an error code)*
9	Coprocessor segment overrun
10	Bad TSS *(pushes an error code)*
11	Segment not present *(pushes an error code)*
12	Stack fault *(pushes an error code)*
13	General protection fault *(pushes an error code)*
14	Page fault *(pushes an error code)*
15	Unknown interrupt exception
16	Coprocessor fault
17	Alignment check exception
18	Machine check exception
19-31 Reserved	

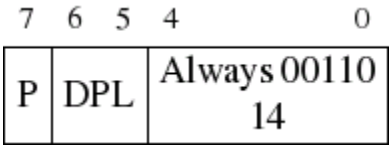
4.2.2 Practical

idt.h

Just like the GDT, there are some structures that need defining:

```
// IDT initialisation function.
void init_idt ();

// This structure describes one interrupt gate.
typedef struct
{
  uint16_t base_lo;         // The lower 16 bits of the address to jump to.
  uint16_t sel;             // Kernel segment selector.
  uint8_t  always0;         // This must always be zero.
  uint8_t  flags;           // More flags. See documentation.
  uint16_t base_hi;         // The upper 16 bits of the address to jump to.
} __attribute__((packed)) idt_entry_t;

// A pointer structure used for informing the CPU about our IDT.
typedef struct
{
  uint16_t limit;
  uint32_t base;            // The address of the first element in our idt_entry_t array.
} __attribute__((packed)) idt_ptr_t;

// Structure containing register values when the CPU was interrupted.
typedef struct
{
  uint32_t ds;              // Data segment selector.
  uint32_t edi, esi, ebp, esp, ebx, edx, ecx, eax; // Pushed by pusha.
  uint32_t int_no, err_code; // Interrupt number and error code (if applicable).
  uint32_t eip, cs, eflags, useresp, ss; // Pushed by the processor automatically.
} registers_t;

// An interrupt handler. It is a pointer to a function which takes a pointer
// to a structure containing register values.
typedef void (*interrupt_handler_t)(registers_t *);

// Allows us to register an interrupt handler.
void register_interrupt_handler (uint8_t n, interrupt_handler_t h);

// These extern directives let us access the addresses of our ASM ISR handlers.
extern void isr0 ();
extern void isr1 ();
extern void isr2 ();
extern void isr3 ();
extern void isr4 ();
extern void isr5 ();
extern void isr6 ();
extern void isr7 ();
extern void isr8 ();
extern void isr9 ();
extern void isr10();
extern void isr11();
extern void isr12();
extern void isr13();
extern void isr14();
```

```
extern void isr15();
extern void isr16();
extern void isr17();
extern void isr18();
extern void isr19();
extern void isr20();
extern void isr21();
extern void isr22();
extern void isr23();
extern void isr24();
extern void isr25();
extern void isr26();
extern void isr27();
extern void isr28();
extern void isr29();
extern void isr30();
extern void isr31();
extern void isr255();
```



See? Very similar to the GDT entry and ptr structs. The flags field format is shown above. The lower 5-bits should be constant 0b00110 - 6 in decimal. The DPL describes the privilege level we expect to be called from - in our case zero, but as we progress we'll have to change that to 3. The P bit signifies the entry is present. Any descriptor with this bit clear will cause a "Interrupt Not Handled" exception.

idt.c

```
//
// idt.c - Initialises the GDT and IDT, and defines the
//         default ISR and IRQ handler.
//         Based on code from Bran's kernel development tutorials.
//         Rewritten for JamesM's kernel development tutorials.
//

#include "common.h"
#include "idt.h"

// Lets us access our ASM functions from our C code.
extern void idt_flush(uint32_t);

// Internal function prototypes.
static void idt_set_gate(uint8_t,uint32_t,uint16_t,uint8_t);

// The IDT itself.
idt_entry_t idt_entries [256];
// Pointer structure to give to the CPU.
idt_ptr_t idt_ptr;
// Array of interrupt handler functions.
interrupt_handler_t interrupt_handlers [256];

// Initialisation routine - zeroes all the interrupt service routines, and
// initialises the IDT.
```

```
void init_idt ()
{
  // Zero all interrupt handlers initially.
  memset (&interrupt_handlers, 0, sizeof (interrupt_handler_t) * 256);

  // Just like the GDT, the IDT has a "limit" field that is set to the last valid byte in the IDT,
  // after adding in the start position (i.e. size-1).
  idt_ptr.limit = sizeof (idt_entry_t) * 256 - 1;
  idt_ptr.base  = (uint32_t) &idt_entries;

  // Zero the IDT to start with.
  memset (&idt_entries, 0, sizeof (idt_entry_t) * 255);

  // Set each gate in the IDT that we care about - that is:
  // 0-32:  Used by the CPU to report conditions, both normal and error.
  // 255:   Will be used later as a way to execute system calls.
  idt_set_gate ( 0, (uint32_t)isr0 , 0x08, 0x8E);
  idt_set_gate ( 1, (uint32_t)isr1 , 0x08, 0x8E);
  idt_set_gate ( 2, (uint32_t)isr2 , 0x08, 0x8E);
  idt_set_gate ( 3, (uint32_t)isr3 , 0x08, 0x8E);
  idt_set_gate ( 4, (uint32_t)isr4 , 0x08, 0x8E);
  idt_set_gate ( 5, (uint32_t)isr5 , 0x08, 0x8E);
  idt_set_gate ( 6, (uint32_t)isr6 , 0x08, 0x8E);
  idt_set_gate ( 7, (uint32_t)isr7 , 0x08, 0x8E);
  idt_set_gate ( 8, (uint32_t)isr8 , 0x08, 0x8E);
  idt_set_gate ( 9, (uint32_t)isr9 , 0x08, 0x8E);
  idt_set_gate (10, (uint32_t)isr10, 0x08, 0x8E);
  idt_set_gate (11, (uint32_t)isr11, 0x08, 0x8E);
  idt_set_gate (12, (uint32_t)isr12, 0x08, 0x8E);
  idt_set_gate (13, (uint32_t)isr13, 0x08, 0x8E);
  idt_set_gate (14, (uint32_t)isr14, 0x08, 0x8E);
  idt_set_gate (15, (uint32_t)isr15, 0x08, 0x8E);
  idt_set_gate (16, (uint32_t)isr16, 0x08, 0x8E);
  idt_set_gate (17, (uint32_t)isr17, 0x08, 0x8E);
  idt_set_gate (18, (uint32_t)isr18, 0x08, 0x8E);
  idt_set_gate (19, (uint32_t)isr19, 0x08, 0x8E);
  idt_set_gate (20, (uint32_t)isr20, 0x08, 0x8E);
  idt_set_gate (21, (uint32_t)isr21, 0x08, 0x8E);
  idt_set_gate (22, (uint32_t)isr22, 0x08, 0x8E);
  idt_set_gate (23, (uint32_t)isr23, 0x08, 0x8E);
  idt_set_gate (24, (uint32_t)isr24, 0x08, 0x8E);
  idt_set_gate (25, (uint32_t)isr25, 0x08, 0x8E);
  idt_set_gate (26, (uint32_t)isr26, 0x08, 0x8E);
  idt_set_gate (27, (uint32_t)isr27, 0x08, 0x8E);
  idt_set_gate (28, (uint32_t)isr28, 0x08, 0x8E);
  idt_set_gate (29, (uint32_t)isr29, 0x08, 0x8E);
  idt_set_gate (30, (uint32_t)isr30, 0x08, 0x8E);
  idt_set_gate (31, (uint32_t)isr31, 0x08, 0x8E);

  idt_flush ((uint32_t)&idt_ptr);
}

static void idt_set_gate (uint8_t num, uint32_t base, uint16_t sel, uint8_t flags)
{
  idt_entries[num].base_lo = base & 0xFFFF;
  idt_entries[num].base_hi = (base >> 16) & 0xFFFF;

  idt_entries[num].sel     = sel;
```

```
  idt_entries[num].always0 = 0;
  // We must uncomment the OR below when we get to using user-mode.
  // It sets the interrupt gate's privilege level to 3.
  idt_entries[num].flags   = flags /* | 0x60 */;
}
```

idt_s.s

We need to define the idt_flush function in assembler:

```
%if CHAPTER >= 4
;
; idt.s -- contains interrupt descriptor table setup code.
;          Based on code from Bran's kernel development tutorials.
;          Rewritten for JamesM's kernel development tutorials.

global idt_flush:function idt_flush.end-idt_flush ; Allows the C code to call idt_flush().
idt_flush:
    mov eax, [esp+4]  ; Get the pointer to the IDT, passed as a parameter.
    lidt [eax]        ; Load the IDT pointer.
    ret
.end:
```

Great! We've got code that will tell the CPU where to find our interrupt handlers - but we haven't written any yet!

When the processor receives an interrupt, it saves the contents of the essential registers (instruction pointer, stack pointer, code and data segments, flags register) on the stack. It then finds the interrupt handler location from our IDT and jumps to it.

Now, just like POSIX signal handlers, you don't get given any information about what interrupt was called when your handler is run. So, unfortunately, we can't just have one common handler, we must write a different handler for each interrupt we want to handle. This involves a lot of grind, so like good engineers we want to keep the amount of duplicated code to a minimum. We do this by writing many handlers that just push the interrupt number (hardcoded in the ASM) onto the stack, and call a common handler function.

Unfortunately we have another problem - some interrupts also push an error code onto the stack. We can't call a common function without a common stack frame, so for those that don't push an error code, we push a dummy one, so the stack is the same.

```
global isr0
isr0:
  cli                 ; Disable interrupts
  push byte 0         ; Push a dummy error code (if ISR0 doesn't push it's own error code)
  push byte 0         ; Push the interrupt number (0)
  jmp isr_common_stub ; Go to our common handler.
```

That sample routine will work, but 32 versions of that still sounds like a lot of code. We can use NASM's macro facility to cut this down, though:

```
%macro ISR_NOERRCODE 1
  global isr%1
  isr%1:
    cli                         ; Disable interrupts firstly.
    push 0                      ; Push a dummy error code.
    push %1                     ; Push the interrupt number.
    jmp isr_common_stub         ; Go to our common handler code.
%endmacro
```

```
; This macro creates a stub for an ISR which passes it's own
; error code.
%macro ISR_ERRCODE 1
  global isr%1
  isr%1:
    cli                         ; Disable interrupts.
    push %1                     ; Push the interrupt number
    jmp isr_common_stub
%endmacro

ISR_NOERRCODE 0
ISR_NOERRCODE 1
ISR_NOERRCODE 2
ISR_NOERRCODE 3
ISR_NOERRCODE 4
ISR_NOERRCODE 5
ISR_NOERRCODE 6
ISR_NOERRCODE 7
ISR_ERRCODE   8
ISR_NOERRCODE 9
ISR_ERRCODE   10
ISR_ERRCODE   11
ISR_ERRCODE   12
ISR_ERRCODE   13
ISR_ERRCODE   14
ISR_NOERRCODE 15
ISR_NOERRCODE 16
ISR_NOERRCODE 17
ISR_NOERRCODE 18
ISR_NOERRCODE 19
ISR_NOERRCODE 20
ISR_NOERRCODE 21
ISR_NOERRCODE 22
ISR_NOERRCODE 23
ISR_NOERRCODE 24
ISR_NOERRCODE 25
ISR_NOERRCODE 26
ISR_NOERRCODE 27
ISR_NOERRCODE 28
ISR_NOERRCODE 29
ISR_NOERRCODE 30
ISR_NOERRCODE 31
ISR_NOERRCODE 255

; C function in idt.c
extern idt_handler

global isr_common_stub:function isr_common_stub.end-isr_common_stub

; This is our common ISR stub. It saves the processor state, sets
; up for kernel mode segments, calls the C-level fault handler,
; and finally restores the stack frame.
isr_common_stub:
    pusha                       ; Pushes edi,esi,ebp,esp,ebx,edx,ecx,eax

    mov ax, ds                  ; Lower 16-bits of eax = ds.
    push eax                    ; Save the data segment descriptor
```

```
    mov ax, 0x10              ; Load the kernel data segment descriptor
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax

    push esp                  ; Push a pointer to the current top of stack - this becomes the registe
    call idt_handler          ; Call into our C code.
    add esp, 4                      ; Remove the registers_t* parameter.

    pop ebx                   ; Reload the original data segment descriptor
    mov ds, bx
    mov es, bx
    mov fs, bx
    mov gs, bx
    mov ss, bx

    popa                      ; Pops edi,esi,ebp...
    add esp, 8                ; Cleans up the pushed error code and pushed ISR number
    iret                      ; pops 5 things at once: CS, EIP, EFLAGS, SS, and ESP
.end:

%if CHAPTER >= 5
```

We can now make a stub handler function just by doing

```
ISR_NOERRCODE 0
ISR_NOERRCODE 1
...
```

Much less work, and anything that makes our lives easier is worth doing. A quick look at the intel manual will tell you that only interrupts 8, 10-14 inclusive push error codes onto the stack. The rest require dummy error codes.

We're almost there, I promise!

Only 2 more things left to do - one is to create an ASM common handler function. The other is to create a higher-level C handler function.

```
extern idt_handler

global isr_common_stub:function isr_common_stub.end-isr_common_stub

; This is our common ISR stub. It saves the processor state, sets
; up for kernel mode segments, calls the C-level fault handler,
; and finally restores the stack frame.
isr_common_stub:
    pusha                     ; Pushes edi,esi,ebp,esp,ebx,edx,ecx,eax

    mov ax, ds                ; Lower 16-bits of eax = ds.
    push eax                  ; Save the data segment descriptor

    mov ax, 0x10              ; Load the kernel data segment descriptor
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax

    push esp                  ; Push a pointer to the current top of stack - this becomes the registe
```

```
    call idt_handler          ; Call into our C code.
    add esp, 4                      ; Remove the registers_t* parameter.

    pop ebx                   ; Reload the original data segment descriptor
    mov ds, bx
    mov es, bx
    mov fs, bx
    mov gs, bx
    mov ss, bx

    popa                      ; Pops edi,esi,ebp...
    add esp, 8                ; Cleans up the pushed error code and pushed ISR number
    iret                      ; pops 5 things at once: CS, EIP, EFLAGS, SS, and ESP
.end:

%if CHAPTER >= 5

; This macro creates a stub for an IRQ - the first parameter is
; the IRQ number, the second is the ISR number it is remapped to.
%macro IRQ 2
  global irq%1
  irq%1:
    cli
    push byte 0
    push byte %2
    jmp irq_common_stub
%endmacro

IRQ   0,    32
IRQ   1,    33
IRQ   2,    34
IRQ   3,    35
IRQ   4,    36
IRQ   5,    37
IRQ   6,    38
IRQ   7,    39
IRQ   8,    40
IRQ   9,    41
IRQ  10,    42
IRQ  11,    43
IRQ  12,    44
IRQ  13,    45
IRQ  14,    46
IRQ  15,    47
```

This piece of code is our common interrupt handler. It firstly uses the 'pusha' command to push all the general purpose registers on the stack. It uses the 'popa' command to restore them at the end. It also gets the current data segment selector and pushes that onto the stack, sets all the segment registers to the kernel data selector, and restores them afterwards. This won't actually have an effect at the moment, but it will when we switch to user-mode. Notice it also calls a higher-level interrupt handler - *idt_handler*.

When an interrupt fires, the processor automatically pushes information about the processor state onto the stack. The code segment, instruction pointer, flags register, stack segment and stack pointer are pushed. The IRET instruction is specifically designed to return from an interrupt. It pops these values off the stack and returns the processor to the state it was in originally.

idt.c, again

In idt.c, we define the handler function and a quick way of registering interrupt handlers.

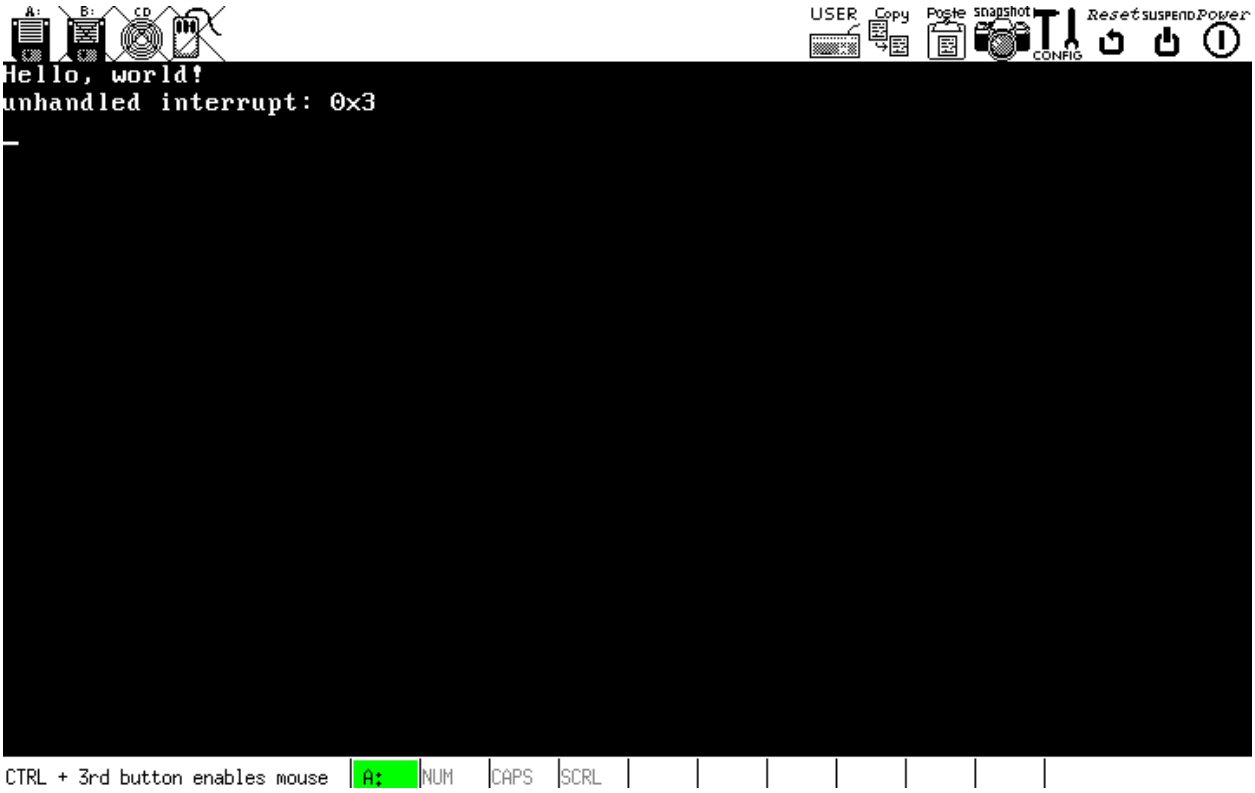
```
void idt_handler (registers_t *regs)
{
  if (interrupt_handlers [regs->int_no])
    interrupt_handlers [regs->int_no] (regs);
  else
  {
    printk ("Unhandled interrupt: %d\n", regs->int_no);
#if CHAPTER >= 6
    panic ("Unhandled interrupt");
#endif // CHAPTER >= 6
  }
}

void register_interrupt_handler (uint8_t n, interrupt_handler_t h)
{
  interrupt_handlers [n] = h;
}
```

Here we use a simple interrupt dispatch system. an *interrupt_handler_t* is typedefed as a function pointer, taking one argument. It uses a structure registers_t, which is a representation of all the registers we pushed, and we defined in idt.h.

Testing it out

Now we can test it out - Add this to your main() function:

```
init_gdt();
init_idt();
asm volatile ("int $0x3");
```



This causes a software interrupt. You should see the message printed out just like the screenshot above (as you don't have a handler registered for interrupt 3 yet)

Congratulations! You've now got a kernel that can handle interrupts, and set up its own segmentation tables (a pretty hollow victory, considering all that code and theory, but unfortunately there's no getting around it!).

The sample code for this tutorial can be found `here`.

CHAPTER

FIVE

IRQS AND THE PIT

In this chapter we're going to be learning about interrupt requests (IRQs) and the programmable interval timer (PIT).

5.1 Interrupt requests (theory)

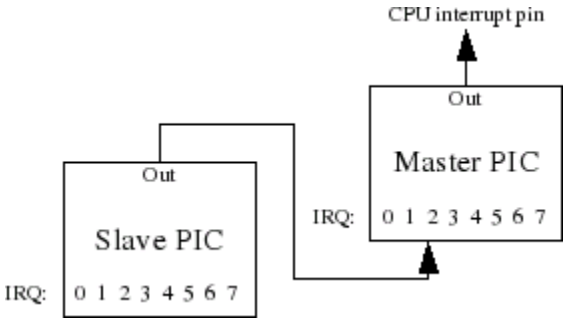
There are several methods for communicating with external devices. Two of the most useful and popular are polling and interrupting.

Polling Spin in a loop, occasionally checking if the device is ready.

Interrupts Do lots of useful stuff. When the device is ready it will cause a CPU interrupt, causing your handler to be run.

As can probably be gleaned from my biased descriptions, interrupting is considered better for many situations. Polling has lots of uses - some CPUs may not have an interrupt mechanism, or you may have many devices, or maybe you just need to check so infrequently that it's not worth the hassle of interrupts. At any rate, interrupts are a very useful method of hardware communication. They are used by the keyboard when keys are pressed, and also by the programmable interval timer (PIT).

The low-level concepts behind external interrupts are not very complex. All devices that are interrupt-capable have a line connecting them to the PIC (programmable interrupt controller). The PIC is the only device that is directly connected to the CPU's interrupt pin. It is used as a multiplexer, and has the ability to prioritise between interrupting devices. It is, essentially, a glorified 8-1 multiplexer. At some point, someone somewhere realised that 8 IRQ lines just wasn't enough, and they daisy-chained another 8-1 PIC beside the original. So in all modern PCs, you have 2 PICs, the master and the slave, serving a total of 15 interruptable devices (one line is used to signal the slave PIC).



The other clever thing about the PIC is that you can change the interrupt number it delivers for each IRQ line. This is referred to as *remapping the PIC* and is actually extremely useful. When the computer boots, the default interrupt mappings are:

- IRQ 0..7 - INT 0x8..0xF

- IRQ 8..15 - INT 0x70..0x77

This causes us somewhat of a problem. The master's IRQ mappings (0x8-0xF) conflict with the interrupt numbers used by the CPU to signal exceptions and faults (see *last chapter*). The normal thing to do is to remap the PICs so that IRQs 0..15 correspond to ISRs 32..47 (31 being the last CPU-used ISR).

5.2 Interrupt requests (practical)

The PICs are communicated with via the I/O bus. Each has a command port and a data port:

- Master - command: 0x20, data: 0x21
- Slave - command: 0xA0, data: 0xA1

The code for remapping the PICs is the most difficult and obfusticated. To remap them, you have to do a full reinitialisation of them, which is why the code is so long. If you're interested in what's actually happening, there is a nice description here.

```
void init_idt()
{
  ...
  // Remap the irq table.
  outb(0x20, 0x11);
  outb(0xA0, 0x11);
  outb(0x21, 0x20);
  outb(0xA1, 0x28);
  outb(0x21, 0x04);
  outb(0xA1, 0x02);
  outb(0x21, 0x01);
  outb(0xA1, 0x01);
  outb(0x21, 0x0);
  outb(0xA1, 0x0);
  ...
  idt_set_gate(32, (uint32_t)irq0, 0x08, 0x8E);
  ...
  idt_set_gate(47, (uint32_t)irq15, 0x08, 0x8E);
}
```

Notice that now we are also setting IDT gates for numbers 32-47, for our IRQ handlers. We must, therefore, also add stubs for these in idt_s.s. Also, though, we need a new macro in idt_s.s - an IRQ stub will have 2 numbers associated with it - its IRQ number (0-15) and its interrupt number (32-47):

```
; This macro creates a stub for an IRQ - the first parameter is
; the IRQ number, the second is the ISR number it is remapped to.
%macro IRQ 2
  global irq%1:function irq%1.end-irq%1
  irq%1:
    cli
    push byte 0
    push byte %2
    jmp irq_common_stub
%endmacro

...

IRQ   0,    32
IRQ   1,    33
```

```
...
IRQ  15,    47
```

We also have a new common stub - *irq_common_stub*. This is because IRQs behave subtly differently - before you return from an IRQ handler, you must inform the PIC that you have finished, so it can dispatch the next (if there is one waiting). This is known as an EOI (end of interrupt). There is a slight complication though. If the master PIC sent the IRQ (number 0-7), we must send an EOI to the master (obviously). If the *slave* sent the IRQ (8-15), we must send an EOI to both the master *and* the slave (because of the daisy-chaining of the two).

First our asm common stub. It is almost identical to *isr_common_stub*.

```
; C function in idt.c
extern irq_handler

global irq_common_stub:function irq_common_stub.end-irq_common_stub

; This is our common IRQ stub. It saves the processor state, sets
; up for kernel mode segments, calls the C-level fault handler,
; and finally restores the stack frame.
irq_common_stub:
   pusha                    ; Pushes edi,esi,ebp,esp,ebx,edx,ecx,eax

   mov ax, ds               ; Lower 16-bits of eax = ds.
   push eax                 ; save the data segment descriptor

   mov ax, 0x10  ; load the kernel data segment descriptor
   mov ds, ax
   mov es, ax
   mov fs, ax
   mov gs, ax
   mov ss, ax

   push esp
   call irq_handler

   pop ebx                  ; reload the original data segment descriptor
   mov ds, bx
   mov es, bx
   mov fs, bx
   mov gs, bx
   mov ss, bx

   popa                     ; Pops edi,esi,ebp...
   add esp, 8     ; Cleans up the pushed error code and pushed ISR number
   sti
   iret           ; pops 5 things at once: CS, EIP, EFLAGS, SS, and ESP
```

Now the C code (goes in idt.c):

```
// This gets called from our ASM interrupt handler stub.
void irq_handler(registers_t *regs)
{
   if (interrupt_handlers[regs->int_no] != 0)
     interrupt_handlers[regs->int_no] (regs);

   // Send an EOI (end of interrupt) signal to the PICs.
   // If this interrupt involved the slave.
   if (regs.int_no >= 40)
   {
```

```
        // Send reset signal to slave.
        outb(0xA0, 0x20);
    }
    // Send reset signal to master. (As well as slave, if necessary).
    outb(0x20, 0x20);
}
```

This is fairly straightforward - if the IRQ was > 7 (interrupt number > 40), we send a reset signal to the slave. In either case, we send one to the master also.

Some other declarations are needed:

5.2.1 idt.h

```
// A few defines to make life a little easier
#define IRQ0 32
...
#define IRQ15 47
```

And there we go! We can now handle interrupt requests from external devices, and dispatch them to custom handlers. Now all we need is some interrupt requests to handle!

5.3 The PIT (theory)

The programmable interval timer is a chip connected to IRQ0. It can interrupt the CPU at a user-defined rate (between 18.2Hz and 1.1931 MHz). The PIT is the primary method used for implementing a system clock and the only method available for implementing multitasking (switch processes on interrupt).

The PIT has an internal clock which oscillates at approximately 1.1931MHz. This clock signal is fed through a frequency divider to modulate the final output frequency. It has 3 channels, each with its own frequency divider.

- Channel 0 is the most useful. Its output is connected to IRQ0.
- Channel 1 is very un-useful and on modern hardware is no longer implemented. It was used to control refresh rates for DRAM.
- Channel 2 controls the PC speaker.

Channel 0 is the only one of use to us at the moment.

OK, so we want to set the PIT up so it interrupts us at regular intervals, at frequency f. I generally set f to be about 100Hz (once every 10 milliseconds), but feel free to set it to whatever you like. To do this, we send the PIT a 'divisor'. This is the number that it should divide it's input frequency (1.9131MHz) by. It's dead easy to work out:

divisor = 1193180 Hz / frequency (in Hz)

Also worthy of note is that the PIT has 4 registers in I/O space - 0x40-0x42 are the data ports for channels 0-2 respectively, and 0x43 is the command port.

5.3.1 The PIT (practical)

We'll need a few new files. timer.h has only a declaration in it:

```
// timer.h -- Defines the interface for all PIT-related functions.
//            Written for JamesM's kernel development tutorials.
```

```
#ifndef TIMER_H
#define TIMER_H

#include "common.h"

void init_timer(uint32_t frequency);

#endif
```

And timer.c doesn't have much in either:

```
// timer.c -- Initialises the PIT, and handles clock updates.
//            Written for JamesM's kernel development tutorials.

#include "common.h"
#include "timer.h"
#include "idt.h"

uint32_t tick = 0;

static void timer_callback(registers_t *regs)
{
   tick++;
   monitor_write("Tick: ");
   monitor_write_dec(tick);
   monitor_write("\n");
}

void init_timer(uint32_t frequency)
{
   // Firstly, register our timer callback.
   register_interrupt_handler(IRQ0, &timer_callback);

   // The value we send to the PIT is the value to divide it's input clock
   // (1193180 Hz) by, to get our required frequency. Important to note is
   // that the divisor must be small enough to fit into 16-bits.
   uint32_t divisor = 1193180 / frequency;

   // Send the command byte.
   outb(0x43, 0x36);

   // Divisor has to be sent byte-wise, so split here into upper/lower bytes.
   uint8_t l = (uint8_t)(divisor & 0xFF);
   uint8_t h = (uint8_t)( (divisor>>8) & 0xFF );

   // Send the frequency divisor.
   outb(0x40, l);
   outb(0x40, h);
}
```

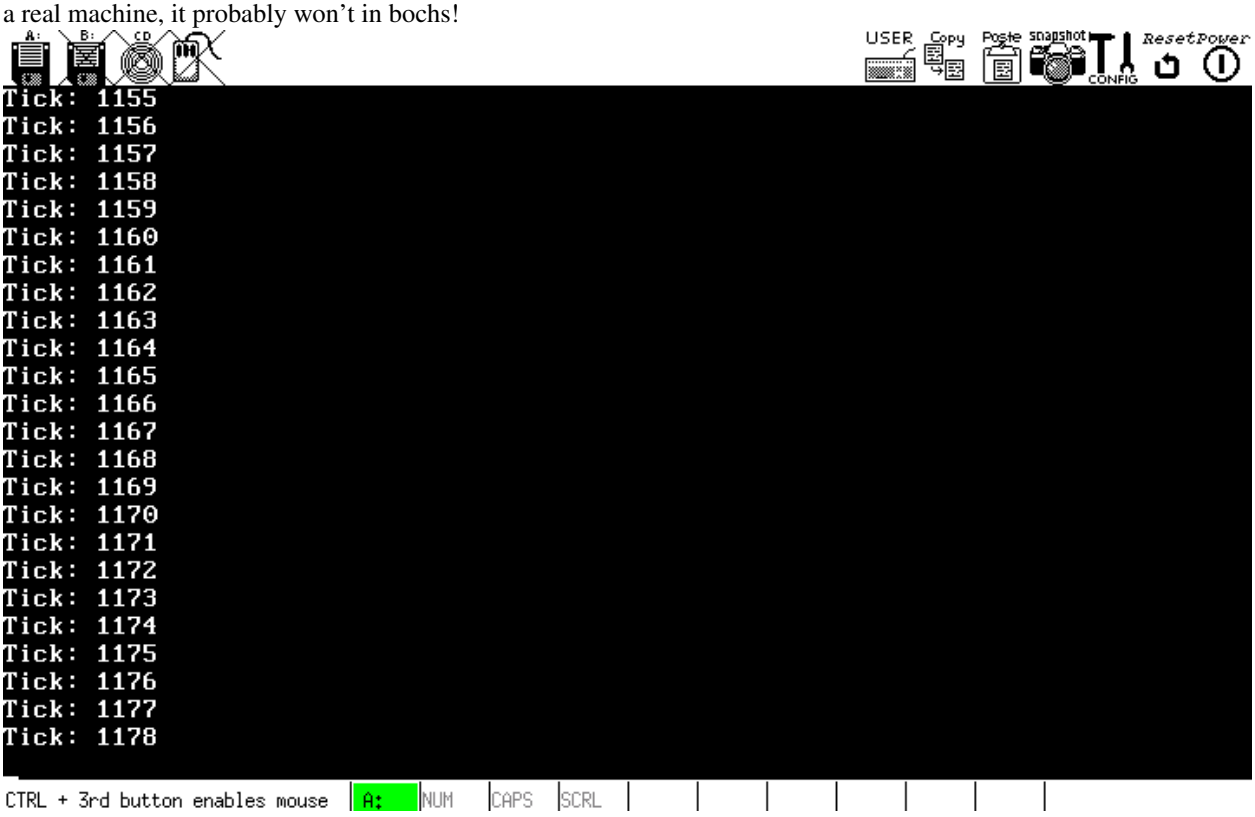
OK, lets go through this code. Firstly, we have our *init_timer* function. This tells our interrupt mechanism that we want to handle IRQ0 with the function *timer_callback*. This will be called whenever a timer interrupt is recieved. We then calculate the divisor to be sent to the PIT (see theory above). Then, we send a command byte to the PIT's command port. This byte (0x36) sets the PIT to repeating mode (so that when the divisor counter reaches zero it's automatically refreshed) and tells it we want to set the divisor value.

We then send the divisor value. Note that it must be sent as two seperate bytes, not as one 16-bit value.

When this is done, all we have to do is add two line to main.c;

```
init_timer (20); // Initialise timer to 20Hz

asm volatile("sti");
```

The "sti" instruction allows external interrupts to be handled. You should get output like that on the right. Note however that bochs does not accurately emulate the timer chip, so although your code will run at the correct speed on a real machine, it probably won't in bochs!



Full source code for this tutorial can be found `here`.

CHAPTER

SIX

BACKTRACING AND SYMBOL LOOKUP

There will be times when your kernel gets into a bad state and you want to print a backtrace of all function calls up to the current point (like the "bt" function in GDB).

There can be many times that this is useful - if an assertion fails, if you throw a kernel panic, or if you are developing your own in-kernel debugger.

6.1 Backtracing

Backtracing is the act of trawling the call stack from the current position back up, finding the locations that function calls were made from. It can be difficult on some platforms or with some compiler optimisations because of the lack of a frame pointer, but GCC on 32-bit x86 by default uses one so backtracing is extremely easy.

What is a frame pointer? Simple. When a function begins (in the "prologue"), the first thing it does is create a stack frame for itself by decrementing the stack pointer (ESP):

```
+------------------------------+
|Previous function's stack frame|
+------------------------------+
|Return address, pushed by CALL |
+------------------------------+
|  This function's stack frame  |
|                               |
|                               |
|                               |
+-------------------------------+   <--- ESP
```

The compiler can emit code to reference variables in the stack frame either relative to the stack pointer (ESP+X), but that could change (for example if stack-based *alloca* is used), so alternatively it could save the address of the *top* of the stack frame into a frame pointer register or *base register* (EBP-Y):

```
+------------------------------+
|Previous function's stack frame|
+------------------------------+
|Return address, pushed by CALL |
+------------------------------+
|  This function's stack frame  |   <--- EBP
|                               |
|                               |
|                               |
+-------------------------------+   <--- ESP
```

How does this help us? Well if using a base pointer, the first thing a function prologue will do is save the previous function's base pointer, so the stack layout looks more like this:

```
+------------------------------+
|Previous function's stack frame|
+------------------------------+
|Return address, pushed by CALL |
+------------------------------+
|    Previous function's EBP    |   <--- EBP
+------------------------------+
|  This function's stack frame  |
|                               |
|                               |
+------------------------------+   <--- ESP
```

What should jump out at you from this diagram is that you can treat the value of EBP as a pointer to the previous value of EBP, and so on - it is therefore a linked list that trawls through the stack back to the beginning.

And from any EBP value, you can find the corresponding return address (thus where the function was called from) by looking up in the stack one slot (EBP+4).

6.1.1 Printing a stack trace

With this information, you can construct a backtracing function rather simply:

```
void print_stack_trace ()
{
  uint32_t *ebp, *eip;
  asm volatile ("mov %%ebp, %0" : "=r" (ebp)); // Start with the current EBP value.
  while (ebp)
  {
    eip = ebp+1;
    printk ("   [0x%x]\n", *eip);
    ebp = (uint32_t*) *ebp;
  }
}
```

You can see that the code is relying on ebp becoming zero at the end of the stack - this is reliable because in boot.s we zeroed the frame pointer:

```
mov ebp, 0
```

With this function you will get a backtrace, but only of addresses. It'd be more useful if you could get the function names instead of just their addresses.

6.2 ELF and symbol lookup

The kernel you are booting is in ELF format. This is the UNIX standard format for linker object files (*.o), shared libraries (*.so), and executables. It caters to both linkers, which want quite fine-grained information about different sections of code and data in order to link correctly, and binary loaders that just care whether to load something executable, read-only or read-write and where the entry point is.

To this end, the information in an ELF file is duplicated:

- *Sections* Are valid when an object is still to be linked (*.o), and may still be valid when the final image is produced (*.so or executable), or may be stripped out at this point. They provide quite fine-grained information about different pieces of code and data. They also contain a symbol table section, which maps symbol names to addresses.

- *Segments* Are valid after the object has been finally linked, and coalesce similar sections, rejecting data not useful at runtime.

As long as you haven't run "strip" on your kernel, it will have a symbol table, and GRUB will actually have loaded it in for you if you know where to look.

6.2.1 Multiboot.h, a slight reprise

In the second chapter (*Genesis*) you blindly copied the multiboot information structure into multiboot.h. Here we're going to use some fields of this structure, namely:

```
uint32_t num;
...
uint32_t addr;
uint32_t shndx;
```

These give respectively the number of sections in the ELF file, the address of the start of an array of section header structures, and the index in that array of the special section `.shstrtab`;

Each section has a name, usually prefixed by a "." symbol. There are some well known section names:

- `.text` : Code section
- `.data` : Read-write data section
- `.rodata` : Constant data section
- `.symtab` : Symbol table
- `.strtab` : Section mapping indices from the symbol table to strings (so strings can be reused and compressed).
- `.shstrtab` : Like `.strtab` but holds the strings mapping indices from section header structs to names. This is needed to find the name of any section.

To put this into a little more context, here is the structure for a section header in ELF:

```
typedef struct
{
  uint32_t name;
  uint32_t type;
  uint32_t flags;
  uint32_t addr;
  uint32_t offset;
  uint32_t size;
  uint32_t link;
  uint32_t info;
  uint32_t addralign;
  uint32_t entsize;
} __attribute__((packed)) elf_section_header_t;
```

The important fields for us at the moment are:

name An index into the section `.shstrtab`, which will contain a NULL-terminated name for this section.

addr The address the section should be loaded at. GRUB does this for us.

size Size of the section, in bytes.

Now you can see why the multiboot struct contains the section header index of `.shstrtab`. Without it, we would never be able to find the names of any of the other sections!

6.2.2 Putting some code together

elf.h

We've already seen the ELF section header struct; there is just one more structure to define - that of a symbol. A symbol has a name (index into `.strtab`), address, size and some flags. There is also an ELF-standard defined macro: ELF32_ST_TYPE, which gets the type of a symbol (value 0x2 meaning 'function', which is what we're interested in) from the symbol's `info` member.

```
#define ELF32_ST_TYPE(i) ((i)&0xf)

typedef struct
{
  uint32_t name;
  uint32_t value;
  uint32_t size;
  uint8_t  info;
  uint8_t  other;
  uint16_t shndx;
} __attribute__((packed)) elf_symbol_t;
```

We're also going to define a structure to pack up all the information we take from the multiboot structure so it can be passed around easily, and some accessor functions:

```
typedef struct
{
  elf_symbol_t *symtab;
  uint32_t      symtabsz;
  const char   *strtab;
  uint32_t      strtabsz;
} elf_t;

// Takes a multiboot structure and returns an elf structure containing the symbol information.
elf_t elf_from_multiboot (multiboot_t *mb);

// Looks up a symbol by address.
const char *elf_lookup_symbol (uint32_t addr, elf_t *elf);
```

elf.c

Our ELF initialisation function looks something like this:

```
//
// elf.h -- Defines routines for dealing with Executable and Linking Format files.
//          Written for JamesM's kernel developement tutorials.
//

#include "elf.h"

elf_t elf_from_multiboot (multiboot_t *mb)
{
  int i;
  elf_t elf;
  elf_section_header_t *sh = (elf_section_header_t*)mb->addr;

  uint32_t shstrtab = sh[mb->shndx].addr;
  for (i = 0; i < mb->num; i++)
```

```
  {
    const char *name = (const char *) (shstrtab + sh[i].name);
    if (!strcmp (name, ".strtab"))
    {
      elf.strtab = (const char *)sh[i].addr;
      elf.strtabsz = sh[i].size;
    }
    if (!strcmp (name, ".symtab"))
    {
      elf.symtab = (elf_symbol_t*)sh[i].addr;
      elf.symtabsz = sh[i].size;
    }
  }
  return elf;
}
```

Firstly we get a pointer to the array of section headers from the multiboot struct, then we get the address of the section header string table.

Indexing through the array of section headers, we look for headers with names `.strtab` and `.symtab` for the string table and symbol table respectively, and fill in the `elf_t` object.

Then, to look up the symbol at a given address, we just index through all symbols...:

```
const char *elf_lookup_symbol (uint32_t addr, elf_t *elf)
{
  int i;

  for (i = 0; i < (elf->symtabsz/sizeof (elf_symbol_t)); i++)
  {
```

looking only for a symbol of type 'function':

```
if (ELF32_ST_TYPE(elf->symtab[i].info) != 0x2)
  continue;
```

With 'addr' between its start and end addresses:

```
if ( (addr >= elf->symtab[i].value) &&
     (addr < (elf->symtab[i].value + elf->symtab[i].size)) )
{
```

If we found one, we should return its name by indexing into the string table:

```
      const char *name = (const char *) ((uint32_t)elf->strtab + elf->symtab[i].name);
      return name;
    }
  }
}
```

Done! We can now test this out by creating a "panic" function that reports an unrecoverable error with stacktrace.

6.2.3 Panic!

```
//
// panic.h -- Defines the interface for bringing the system to an abnormal halt.
//            Written for JamesM's kernel development tutorials.
//
```

```
#ifndef PANIC_H
#define PANIC_H

void panic (const char *msg);

#endif
#endif // CHAPTER >= 6

//
// panic.c -- Defines the interface for bringing the system to an abnormal halt.
//            Written for JamesM's kernel development tutorials.
//

#include "panic.h"
#include "common.h"
#include "elf.h"

static void print_stack_trace ();

extern elf_t kernel_elf;

void panic (const char *msg)
{
  printk ("*** System panic: %s\n", msg);
  print_stack_trace ();
  printk ("***\n");
  for (;;) ;
}

void print_stack_trace ()
{
  uint32_t *ebp, *eip;
  asm volatile ("mov %%ebp, %0" : "=r" (ebp));
  while (ebp)
  {
    eip = ebp+1;
    printk ("   [0x%x] %s\n", *eip, elf_lookup_symbol (*eip, &kernel_elf));
    ebp = (uint32_t*) *ebp;
  }
}
```

CHAPTER

SEVEN

PHYSICAL MEMORY MANAGEMENT

Memory management is the most important part of a young kernel. Separating the two concepts of physical and virtual memory management is crucial to creating a decent kernel (something that the previous generation of this tutorial was poor at).

But first, what are physical and virtual memory?

7.1 Virtual memory (theory)

Note: If you already know what virtual memory is, you can skip this section.

In linux, if you create a tiny test program such as

```
int main(char argc, char **argv)
{
  return 0;
}
```

, compile it, then run 'objdump -f', you might find something similar to this.

```
$ objdump -f a.out

a.out:     file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x080482a0
```

Notice the start address of the program is at 0x80482a0, which is about 128MB into the address space. It may seem strange, then, that this program will run perfectly on machines with < 128MB of RAM.

What the program is actually 'seeing', when it reads and writes memory, is a virtual address space. Parts of the virtual address space are mapped to physical memory, and parts are unmapped. If you try to access an unmapped part, the processor raises a *page fault*. The operating system catches it, and in POSIX systems delivers a SIGSEGV signal closely followed by SIGKILL.

This abstraction is extremely useful. It means that compilers can produce a program that relies on the code being at an exact location in memory, every time it is run. With virtual memory, the process thinks it is at, for example, 0x080482a0, but actually it could be at physical memory location 0x1000000. Not only that, but processes cannot accidentally (or deliberately) trample other processes' data or code.

Memory management is a symbiotic relationship between two managers;

- The *physical memory manager* (PMM), which knows which areas of memory are free to use and which are allocated.
- The *virtual memory manager* (VMM), which manages the mappings between virtual and physical addresses.

The PMM could quite possibly need to use the VMM, because keeping a list of all possible areas of memory that are empty may require a lot of memory itself; impossible in general to set aside statically at compilation time because the size is unknown (depends on amount of available RAM).

The VMM relies on the PMM because it has structures that need to reside in physical memory to point the CPU at (see *next chapter*).

It should be noted that when managing memory, we tend to chunk it up into "pages" (the reason will become apparent in *the next chapter*). These are usually (but not always) 4KB in size. A physical memory manager usually operates in terms of pages.

7.2 The intended API

As the PMM and VMM are going to work so closely together, it makes sense to define their interfaces up-front. The PMM will have these accessors (excluding its initialisation function):

```
// Returns the address of a page of physical memory.
uint32_t pmm_alloc_page ();

// Informs that the given page address as no longer used.
void pmm_free_page (uint32_t p);

// The VMM should set this to 1 when it is fully booted.
extern char pmm_paging_active;
```

And the VMM will have the following (if any of these don't make sense, wait until the end of the next chapter!):

```
#define PAGE_PRESENT   0x1        // Page is mapped in.
#define PAGE_WRITE     0x2        // Page is writable. Not set means read-only.
#define PAGE_USER      0x4        // Page is writable from user space. Unset means kernel-only.
#define PAGE_MASK      0xFFFFF000 // Mask constant to page-align an address.

// Switches address space.
void switch_page_directory (page_directory_t *pd);

// Maps the physical page "pa" into the virtual address space at address "va", using
// the given protection flags, which are a logical-OR of the PAGE* constants.
void map (uint32_t va, uint32_t pa, uint32_t flags);

// Removes one page of V->P mappings at virtual address "va"
void unmap (uint32_t va);

// Returns 1 if the given virtual address is mapped in the address space.
// If "pa" is non-NULL, the physical address of the mapping is placed in *pa.
char get_mapping (uint32_t va, uint32_t *pa);
```

As the physical memory manager, we will only require one function in the VMM: `map`. We'll use this later.

7.3 An initial PMM

Importantly because of the symbiotic nature of physical and virtual memory managers, there must be a way of allocating physical memory before the VMM is fully booted. This can be simply achieved by maintaining a simple integer and incrementing it by the page size (4KB, 0x1000, 4096) whenever a page is requested.

Assuming we are told where we can start the allocation:

```
void init_pmm (uint32_t start)
{
  pmm_location = (start + 0x1000) & PAGE_MASK;
}
```

Note: It is important to have every page aligned to a natural boundary, so we add 0x1000 (the page size) to the given start address and null the last three bits.

To allocate pages we just return the current address and add 0x1000 to it. That way the next address is exactly the beginning of the next page to allocate.

```
uint32_t pmm_alloc_page ()
{
    return pmm_location += 0x1000;
}
```

At some point we might want to free some memory. Alas, there's not really any way to do that with our current scheme; that'll have to wait until we can get a more clever allocation scheme active (which will rely on the VMM being booted!)

```
void pmm_free_page (uint32_t p)
{
}
```

7.4 A 'real' stack-based memory manager

The previous scheme will get us to the point where we can serve pages to consumers (namely the VMM). Once it is active however, we should be a little more clever in our allocation scheme so we can actually free pages!

There are two widely used styles of physical memory allocators - stack-based and bitmap-based.

Bitmap-based allocators create a large bitfield with each bit representing if a page is allocated or free. Stack-based allocators maintain a stack of page addresses, pushing an address when freed and popping to allocate.

We're going to implement a stack-based allocator because it is slightly simpler.

7.4.1 pmm.h

We'll have to define where we want our stack to be in memory. As by this point we will have full control over the address space, it can be anywhere. Let's put it near the top of memory.

```
#define PMM_STACK_ADDR 0xFF000000
```

7.4.2 pmm.c

We need some more variables:

```
uint32_t pmm_stack_loc = PMM_STACK_ADDR;
uint32_t pmm_stack_max = PMM_STACK_ADDR;
```

`pmm_stack_loc` will denote our current position in the stack (stack pointer), and `pmm_stack_max` will be the lowest we can go in the stack before we need to allocate more memory for it.

We'll have to modify our allocation function to be conditional on whether the VMM is booted or not. If it's not, we should use our previous 'dumb' scheme.

```
uint32_t pmm_alloc_page ()
{
  if (pmm_paging_active)
  {
```

And here's the allocation code:

```
// Quick sanity check.
if (pmm_stack_loc == PMM_STACK_ADDR)
  panic ("Error:out of memory.");

// Pop off the stack.
pmm_stack_loc -= sizeof (uint32_t);
uint32_t *stack = (uint32_t*)pmm_stack_loc;

return *stack;
```

Firstly we perform a sanity check to ensure we're not going to underrun the stack. Then we decrement our stack pointer to move it up the stack one place (size of a pointer), then return the value at that address. A simple stack `pop` operation.:

```
  }
  else
  {
    return pmm_location += 0x1000;
  }
}
```

When freeing a page, we perform the opposite `push` operation:

```
void pmm_free_page (uint32_t p)
{
  if (p < pmm_location) return;
```

We have to ignore everything before our pmm_location because, we don't want to delete our early allocated pages - they may contain important virtual memory structures.

```
if (pmm_stack_max <= pmm_stack_loc)
{
  map (pmm_stack_max, p, PAGE_PRESENT | PAGE_WRITE);
  pmm_stack_max += 4096;
}
```

Here we check if the stack has run out of space. If it has, we take the page that we were going to push onto the stack and instead map it into the address space for extra stack space. We obviously map it as writable!

```
  else
  {
    // Else we have space on the stack, so push.
    uint32_t *stack = (uint32_t*)pmm_stack_loc;
    *stack = p;
    pmm_stack_loc += sizeof (uint32_t);
  }
}
```

If we don't have space issues, we just push our free page onto the free page stack. Now we have all we need to implement the partner virtual memory manager.

Unlike almost all the other tutorials, this doesn't end with a code sample or example - our PMM is reliant on its companion VMM to work, and we have yet to write it!

CHAPTER

EIGHT

VIRTUAL MEMORY MANAGEMENT

In this chapter we're going to enable paging. Paging serves a twofold purpose - memory protection, and virtual memory (the two being almost inextricably interlinked).

Virtual memory of this type is wholly dependent on hardware support. It cannot be emulated by software. Luckily, the x86 has just such a thing. It's called the MMU (memory management unit), and it handles all memory mappings due to segmentation and paging, forming a layer between the CPU and memory.

8.1 Paging as a concretion of virtual memory

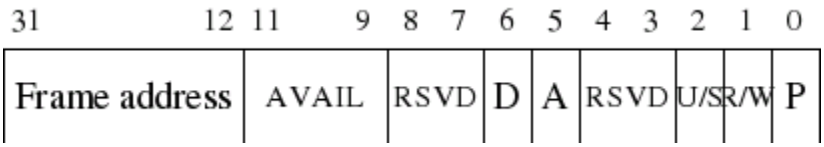
Virtual memory is an abstract principle. As such it requires *concretion* through some system/algorithm. Both segmentation and paging are valid methods for implementing virtual memory. As mentioned in *chapter 3* however, segmentation is becoming obsolete. Paging is the newer, better alternative for the x86 architecture.

Paging works by splitting the virtual address space into blocks called *pages*, which are usually 4KB in size. Pages can then be mapped on to *frames* - equally sized blocks of physical memory.

Note: "Frames" are also commonly called "physical pages", but I'll refer to them as frames here to remove any ambiguity. A "page" will refer to a piece of virtual memory, a "frame" a piece of physical memory.

8.1.1 Page entries

Each process normally has a different set of page mappings, so that virtual memory spaces are independent of each other. In the x86 architecture (32-bit) pages are fixed at 4KB in size. Each page has a corresponding descriptor word, which tells the processor which frame it is mapped to. Note that because pages and frames must be aligned on 4KB boundaries (4KB being 0x1000 bytes), the least significant 12 bits of the 32-bit word are always zero. The architecture takes advantage of this by using them to store information about the page, such as whether it is present, whether it is kernel-mode or user-mode etc. The layout of this word is in the picture on the right.



The fields in that picture are pretty simple, so let's quickly go through them.

P Set if the page is present in memory.

R/W If set, that page is writeable. If unset, the page is read-only. This does not apply when code is running in kernel-mode (unless a flag in CR0 is set).

U/S If set, this is a user-mode page. Else it is a supervisor (kernel)-mode page. User-mode code cannot write to or read from kernel-mode pages.

Reserved These are used by the CPU internally and cannot be trampled.

A Set if the page has been accessed (Gets set by the CPU).

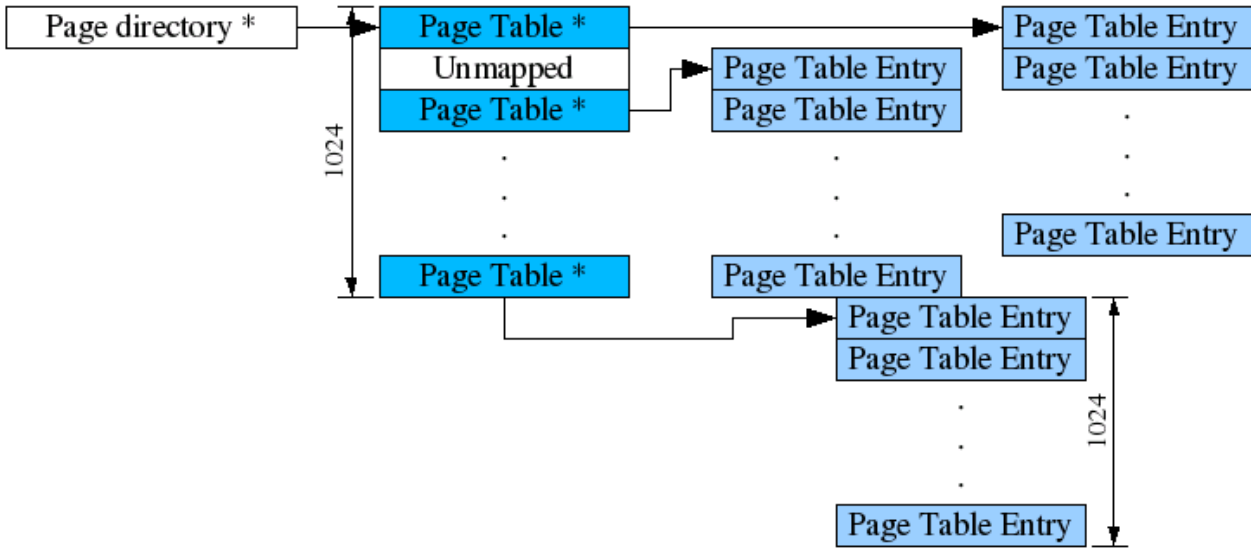
D Set if the page has been written to (dirty).

AVAIL These 3 bits are unused and available for kernel-use.

Page frame address The high 20 bits of the frame address in physical memory.

8.1.2 Page directories/tables

Possibly you've been tapping on your calculator and have worked out that to generate a table mapping each 4KB page to one 32-bit descriptor over a 4GB address space requires 4MB of memory.



4MB may seem like a large overhead, and to be fair, it is. If you have 4GB of physical RAM, it's not much. However, if you are working on a machine that has 16MB of RAM, you've just lost a quarter of your available memory! What we want is something progressive, that will take up an amount of space proportionate to the amount of RAM you have.

Well, we don't have that. But intel did come up with something similar - they use a 2-tier system. The CPU gets told about a *page directory*, which is a 4KB large table, each entry of which points to a *page table*. The page table is, again, 4KB large and each entry is a *page table entry*, described above.

This way, The entire 4GB address space can be covered with the advantage that if a page table has no entries, it can be freed and it's present flag unset in the page directory.

8.1.3 Enable paging

Enabling paging is extremely easy.

1. Copy the location of your page directory into the CR3 register. This must, of course, be the physical address.
2. Set the PG bit in the CR0 register. You can do this by OR-ing with 0x80000000.

8.2 Page Faults

When a process does something the memory-management unit doesn't like, a page fault interrupt is thrown. Situations that can cause this are (not complete):

- Reading from or writing to an area of memory that is not mapped (page entry/table's 'present' flag is not set)
- The process is in user-mode and tries to write to a read-only page.
- The process is in user-mode and tries to access a kernel-only page.
- The page table entry is corrupted - the reserved bits have been overwritten.
- The page fault interrupt is number 14, and looking at chapter 3 we can see that this throws an error code. This error code gives us quite a bit of information about what happened.

Bit 0 If set, the fault was not because the page wasn't present. If unset, the page wasn't present.

Bit 1 If set, the operation that caused the fault was a write, else it was a read.

Bit 2 If set, the processor was running in user-mode when it was interrupted. Else, it was running in kernel-mode.

Bit 3 If set, the fault was caused by reserved bits being overwritten.

Bit 4 If set, the fault occurred during an instruction fetch.

The processor also gives us another piece of information - the address that caused the fault. This is located in the CR2 register. Beware that if your page fault handler itself causes another page fault exception this register will be overwritten - so save it early!

8.3 Putting it into practice

At first we need to define the desired virtual positions of our page directory and our page tables in vmm.h, so we can access them later, when paging is enabled.

> #define PAGE_DIR_VIRTUAL_ADDR 0xFFBFF000 #define PAGE_TABLE_VIRTUAL_ADDR 0xFFC00000

Then we write two macros to ease getting indices of tables and directories from addresses:

> #define PAGE_DIR_IDX(x) ((uint32_t)x/1024) #define PAGE_TABLE_IDX(x) ((uint32_t)x%1024)

To make our code more readable we also define some labels. As you remember, bit 1 stands for present, bit 2 for writeable and bit 3 for user accessible:

> #define PAGE_PRESENT 0x1 #define PAGE_WRITE 0x2 #define PAGE_USER 0x4 #define PAGE_MASK 0xFFFFF000

A page directory is basically an array of 1024 unsigned integers which contain the page table's physical address shifted 12 bits to the left, along with the configuration bits mentioned above. Although its a primitive datatype I strongly recommend to create a type page_directory_t:

> typedef u32int_t page_directory_t

We will also write the following functions, so you might want to add the following prototypes to your vmm.h

```
// Sets up the environment, page directories etc and enables paging.
void init_vmm ();

// Changes address space.
void switch_page_directory (page_directory_t *pd);
```

```
// Maps the physical page "pa" into the virtual space at address "va", using
// page protection flags "flags".
void map (uint32_t va, uint32_t pa, uint32_t flags);

// Removes one page of V->P mappings at virtual address "va".
void unmap (uint32_t va);

// Returns 1 if the given virtual address is mapped in the address space.
// If "*pa" is non-NULL, the physical address of the mapping is placed in *pa.
char get_mapping (uint32_t va, uint32_t *pa);
```

In our vmm.c we first need to create a global page directory which always represents the one in use right now.

```
page_directory_t *current_directory;
```

The first function we will write initializes our paging functionality. It creates the first page directory and tables and activates paging.

```
void init_vmm ()
{
  int i;
  uint32_t cr0;

  register_interrupt_handler (14, &page_fault);
```

We need to register a page fault handler. Page faults trigger interrupt number 14, so we register our handler at that position in our interrupt descriptor table.

```
// Create a page directory.
page_directory_t *pd = (page_directory_t*)pmm_alloc_page ();

// Initialise it.
memset (pd, 0, 0x1000);
```

A single page is enough to hold our first page directory (sizeof(page_directory_t) == 4b; 4b * 1024 = 4kb = size of a page).

```
// Identity map the first 4 MB.
pd[0] = pmm_alloc_page () | PAGE_PRESENT | PAGE_WRITE;
uint32_t *pt = (uint32_t*) (pd[0] & PAGE_MASK);
for (i = 0; i < 1024; i++)
  pt[i] = i*0x1000 | PAGE_PRESENT | PAGE_WRITE;
```

Our kernel code currently resides in the first 4MB of our available memory, so we need to make sure that the virtual addresses match the physical addresses of these. If we don't do this our instruction pointer will point to an invalid memory location after activating paging.

######## I don't understand this design ############# ### I know what it does, but not why we need this. ###

```
// Assign the second-last table and zero it.
pd[1022] = pmm_alloc_page () | PAGE_PRESENT | PAGE_WRITE;
pt = (uint32_t*) (pd[1022] & PAGE_MASK);
memset (pt, 0, 0x1000);

// The last entry of the second-last table is the directory itself.
pt[1023] = (uint32_t)pd | PAGE_PRESENT | PAGE_WRITE;

// The last table loops back on the directory itself.
pd[1023] = (uint32_t)pd | PAGE_PRESENT | PAGE_WRITE;
```

Up until here

```
// Set the current directory.
switch_page_directory (pd);

// Enable paging.
asm volatile ("mov %%cr0, %0" : "=r" (cr0));
cr0 |= 0x80000000;
asm volatile ("mov %0, %%cr0" : : "r" (cr0));
```

Now we activate the directory we just created by switching to it and by setting the most significant bit in cr0.

```
uint32_t pt_idx = PAGE_DIR_IDX((PMM_STACK_ADDR>>12));
page_directory[pt_idx] = pmm_alloc_page () | PAGE_PRESENT | PAGE_WRITE;
memset (page_tables[pt_idx*1024], 0, 0x1000);
```

Now we have to map the the page table where the physical memory manager keeps its page stack, else it will panic on the first "pmm_free_page", because it can't find its stack any more.

```
// Paging is now active. Tell the physical memory manager.
pmm_paging_active = 1;

}
```

And we have to tell it that paging has been activated. Otherwise it won't use our virtual memory for further allocations.

Switching page directories is fairly simple. We just replace current_directory with our newly created one and put its address into the cr3 register:

```
void switch_page_directory (page_directory_t *pd)
{
  current_directory = pd;
  asm volatile ("mov %0, %%cr3" : : "r" (pd));
}
```

If we now want to access physical pages, that isn't as easy as before, because we just enabled paging. So to access physical addresses, we have to map virtual ones to physical ones and access the virtual ones. To do so, we need the following function:

```
void map (uint32_t va, uint32_t pa, uint32_t flags)
{
  uint32_t virtual_page = va / 0x1000;
  uint32_t pt_idx = PAGE_DIR_IDX(virtual_page);
```

First we need to find the right page table to insert our mapping. As you know, the page number equals the virtual address, shifted three digits to the right. To calculate the index in our directory, we use our previously defined macro.

```
// Find the appropriate page table for 'va'.
if (page_directory[pt_idx] == 0)
{
  // The page table holding this page has not been created yet.
  page_directory[pt_idx] = pmm_alloc_page() | PAGE_PRESENT | PAGE_WRITE;
  memset (page_tables[pt_idx*1024], 0, 0x1000);
}
```

In case the page table does not exist, we have to create it.

```
  // Now that the page table definately exists, we can update the PTE.
  page_tables[virtual_page] = (pa & PAGE_MASK) | flags;
}
```

Now we can be sure it exists, so we map our physical address to our virtual address with the given flags.

Now that we can map physical addresses to virtual addresses, we need the possibility to undo that. As you might have imaged out we will name this function unmap:

```
void unmap (uint32_t va)
{
  uint32_t virtual_page = va / 0x1000;
```

First we have to find the right page to our address.

```
page_tables[virtual_page] = 0;
```

Then we just zero it, so it doesn't exist any more.

```
  asm volatile ("invlpg (%0)" : : "a" (va));
}
```

We have to inform the CPU that we have invalidated a page mapping, because it has to clear this entry in the TLB. TLB stands for translation lookaside buffer, which caches address translation reqest answers, to improve the translation's performance. As with any other cache, it has to be invalidated after a change of the underlying data.

If we want to know if a virtual address has a mapping and to which address it is mapped, we have to do this on our own, too:

```
char get_mapping (uint32_t va, uint32_t *pa)
{
  uint32_t virtual_page = va / 0x1000;
  uint32_t pt_idx = PAGE_DIR_IDX(virtual_page);
```

As before, we need to find the right page table which maps our virtual address to a physical address.

```
// Find the appropriate page table for 'va'.
if (page_directory[pt_idx] == 0)
  return 0;
```

If the page table is empty or doesn't exist (which is the same in our case) then the virtual address is not mapped anywhere and we return false.

```
  if (page_tables[virtual_page] != 0)
  {
    if (pa) *pa = page_tables[virtual_page] & PAGE_MASK;
    return 1;
  }
}
```

If it exists, we look up the physical address and put it into pa, if it (pa) is not a nullpointer. After that we return true;

Last but not least, we have to deal with the ominous page fault handler which I teasered earlier. In case of a page fault, we want to know, where it happend and for what reason.

```
void page_fault (registers_t *regs)
{
  uint32_t cr2;
  asm volatile ("mov %%cr2, %0" : "=r" (cr2));
```

CR2 holds the address which caused the page fault, we can use that later to debug the occuring error.

```
  printk ("Page fault at 0x%x, faulting address 0x%x\n", regs->eip, cr2);
  printk ("Error code: %x\n", regs->err_code);
  panic ("");
```

```
  for (;;) ;
}
```

Then we output the faulting address and the instruction pointer to be able to reproduce the error. The error code gives you information about the page we tried to access. The error code is equal to the access flags of the page, if it exists.

CHAPTER

NINE

THE HEAP

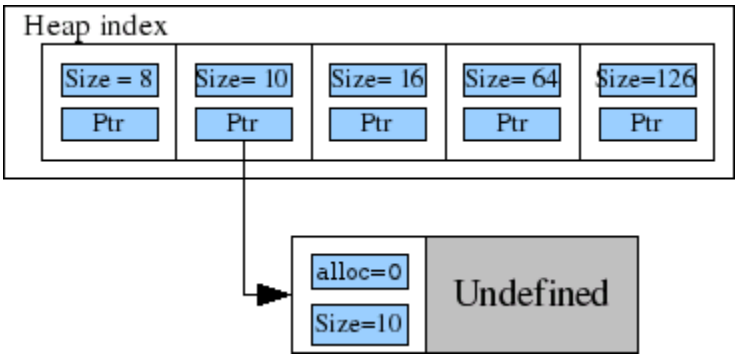
In order to be responsive to situations that you didn't envisage at the design stage, and to cut down the size of your kernel, you will need some kind of dynamic memory allocation. The current memory allocation system (allocation by placement address) is absolutely fine, and is in fact optimal for both time and space for allocations. The problem occurs when you try to free some memory, and want to reclaim it (this must happen eventually, otherwise you will run out!). The placement mechanism has absolutely no way to do this, and is thus not viable for the majority of kernel allocations.

As a sidepoint of general terminology, any data structure that provides both allocation and deallocation of contiguous memory can be referred to as a heap (or a pool). There is, as such, no standard 'heap algorithm' - Different algorithms are used depending on time/space/efficiency requirements. Our requirements are:

- (Relatively) simple to implement.
- Able to check consistency - debugging memory overwrites in a kernel is about ten times more difficult than in normal apps!

The algorithm and data structures presented here are ones which I developed myself. They are so simple however, that I am sure others will have used it first.

9.1 Data structure description



The algorithm uses two concepts: *allocated blocks* and *holes*. Blocks are contiguous areas of memory containing user data currently in use (i.e. malloc()d but not free()d). Holes are unallocated blocks, so their contents are not in use. So initially by this concept the entire area of heap space is one large hole.

We will keep every block and hole in a linked list, so in order to find free space we can just iterate over it and check the entries. Blocks and holes each contain a header filled with descriptive data. The header contains information about the length of a block, whether it is a hole or not and links to the next and previous items in the list. Pseudocode:

```
typedef struct header
{
  struct header *prev, *next;
  uint32_t allocated : 1;
  uint32_t length : 31;
} header_t;
```

Note also that within this tutorial I will refer to the size of a block being the number of bytes from the start of the header to the end of the block - so within a block of size x, there will be x - sizeof(header_t) user-useable bytes.

9.2 Algorithm description

9.2.1 Allocation

Allocation is pretty straightforward: 1. Search the list to find the first hole that fits our request.

- If we didn't find a hole large enough, then:
  1. Expand the heap.
  2. If the list is empty (no recorded holes yet), add a new entry
  3. Else, adjust the last header's size member
2. Decide if the hole should be split into two parts. This will normally be the case - we usually will want much less space than is available in the hole. The only time this will not happen is if there is less free space after allocating the block than the header/footer takes up. In this case we can just increase the block size and reclaim it all afterwards.
3. Adjust the new block's header to be allocated.
4. If the hole was to be split into two parts, do it now.
5. Return the address of the block + sizeof(header_t) to the user.

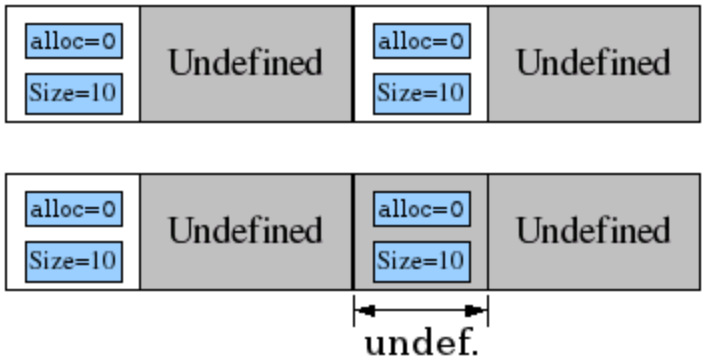
9.2.2 Deallocation

Deallocation (freeing) is a little more tricky. As mentioned earlier, this is where the efficiency of a memory-management algorithm is really tested. The problem is effective reclaimation of memory. The naive solution would be to change the given block to a hole and enter it back into the hole index. However, if I do this:

```
int a = kmalloc(8); // Allocate 8 bytes: returns 0xC0080000 for sake of argument
int b = kmalloc(8); // Allocate another 8 bytes: returns 0xC0080008.
kfree(a);           // Release a
kfree(b);           // Release b
int c = kmalloc(16);// What will this allocation return?
```

Note that in this example the space required for headers and footers have been purposely omitted for readability

Here we have allocated space for 8 bytes, twice. We then release both of those allocations. With the naive release algorithm we would then end up with two 8-byte sized holes in the index. When the next allocation (for 16 bytes) comes along, neither of those holes can fit it, so the kmalloc() call will return 0xC0080010. This is suboptimal. There are 16 bytes of space free at 0xC0080000, so we *should* be reallocating that!



The solution to this problem in most cases is a varation on a simple algorithm that I call unification - That is, converting two adjacent holes into one. (Please note that this coining of a term is not from a sense of self-importance, merely from the absence of a standardised name).

It works thus: When free()ing a block, look at what is immediately to the left (assuming 0-4GB left-to-right) of the header. If it is a hole, we can modify it's header's size attribute to take into account both it's size and ours and delete ours from the list. We have thus amalgamated both holes into one (and in this case there is no need to do an expensive insert operation on the index).

That is what I call *unifying left*. There is also *unifying right*, which should be performed on free() as well. Here we look at what is directly after our block. If we find a hole there, we can add size attribute to our soon to be hole. Then, all that needs to be done is to remove it's old entry from the hole index.

Note also that in the name of reclaiming space, if we are free()ing the last block in the heap (there are no holes or blocks after us), then we can contract the size of the heap.

Pseudocode

1. Find the header by taking the given pointer and subtracting the sizeof(header_t).
2. Set the allocated flag in our header to 0.
3. If the thing immediately to our right is a hole:
   - Unify right.
4. If the thing immediately to our left is a hole:
   - Unify left.
5. If our hole is the last in the heap (&header + header->size >= end_address):
   - Contract.
6. Insert the header into the linked list unless it isn't already in there.

9.3 Initializing the heap

At first we have to give our heap a distinct start and end address. Therefore we define HEAP_START 0xD0000000 and HEAP_END 0xFFBFF000 in heap.h along with the header_t struct described earlier. Of course, we need some prototypes, too:

```
void init_heap();
void *kmalloc(u32int_t l);
void kfree(void *p);
```

In heap.c we have to create the first linked list object and create a variable which contains the current maximum size of our heap (initially its size is zero, so we use the start address).

```
uint32_t heap_max = HEAP_START;
header_t *heap_first = 0;
```

That is all we need to do to be able to allocate the first chunk of memory.

9.4 Allocation and Deallocation

To allocate memory we just iterate over the list (which is initially empty) to find a hole big enough for our request:

```
void *kmalloc (uint32_t l)
{
  l += sizeof (header_t);

  header_t *cur_header = heap_first, *prev_header = 0;
  while (cur_header)
  {
    if (cur_header->allocated == 0 && cur_header->length >= l)
    {
      split_chunk (cur_header, l);
```

split_chunk will cut the block in two pieces, one with the required size, so we don't waste a hole block to allocate a small amount of memory.

```
cur_header->allocated = 1;
return (void*) ((uint32_t)cur_header + sizeof (header_t));
```

If we find a hole big enough, we set it to allocated and return its address (behind the header).

```
  }
  prev_header = cur_header;
  cur_header = cur_header->next;
}
```

If not, we look for the next one. In some cases we won't find a hole big enough, e. g. when allocating the first time or if our request is bigger than any hole generated yet. Then we have to expand the heap to the size needed and allocate the newly created block:

```
uint32_t chunk_start;
if (prev_header)
  chunk_start = (uint32_t)prev_header + prev_header->length;
```

In this case, there are entries in the list, but none was big enough.

```
else
{
  chunk_start = HEAP_START;
  heap_first = (header_t *)chunk_start;
}
```

There are no entries in the list yet, so we create one.

```
alloc_chunk (chunk_start, l);
```

This function allocates a new block by expanding the heap. We will implement it later.

```
cur_header = (header_t *)chunk_start;
cur_header->prev = prev_header;
cur_header->next = 0;
cur_header->allocated = 1;
cur_header->length = l;

prev_header->next = cur_header;
```

Fill the new block's header with the required information and insert it into our list.

```
  return (void*) (chunk_start + sizeof (header_t));
}
```

And return its address (behind the header);

To free memory, we just need to change the blocks status to unallocated and unify it to the right and to the left:

```
void kfree (void *p)
{
  header_t *header = (header_t*)((uint32_t)p - sizeof(header_t));
```

As you know, the pointer to our memory always points to the space behind the header, so to get the header we just subtract its size from the pointer.

```
header->allocated = 0;
```

Set it to unallocated and

```
  glue_chunk (header);
}
```

unify it in both directions.

9.4.1 The tricky part

Sooner or later we will have to expand the heap. Probably sooner as its initial size is 0. To do so, we now implement the aforementioned alloc_chunk() function:

```
void alloc_chunk (uint32_t start, uint32_t len)
{
  while (start + len > heap_max)
  {
```

As long as the heap is to small,

```
uint32_t page = pmm_alloc_page ();
map (heap_max, page, PAGE_PRESENT | PAGE_WRITE);
```

Allocate a page and map it to its end.

```
    heap_max += 0x1000;
  }
}
```

Then increase the heap's size variable one page. All in all this process is very simple.

Splitting a chunk is pretty straightforward as well. We check if the new chunk is big enough to at least hold the header and one byte. Blocks with 0 byte capacity would be useless. If the new block has the required capacity, we adjust the size of the old block and add the new block into our linked list.

```
void split_chunk (header_t *chunk, uint32_t len)
{
  if (chunk->length - len > sizeof (header_t))
  {
```

The required length test...

```
header_t *newchunk = (header_t *) ((uint32_t)chunk + chunk->length);
newchunk->prev = chunk;
newchunk->next = 0;
newchunk->allocated = 0;
newchunk->length = chunk->length - len;
```

Create and fill the new blocks header.

```
    chunk->next = newchunk;
    chunk->length = len;
  }
}
```

And adjust the old one to point to the new block.

The unifying left and right can be done by a single function. As I explained already, we check both sides to figure out if there is anything to unify and if necessary, do so:

```
void glue_chunk (header_t *chunk)
{
  if (chunk->next && chunk->next->allocated == 0)
  {
    chunk->length = chunk->length + chunk->next->length;
    chunk->next->next->prev = chunk;
    chunk->next = chunk->next->next;
  }
```

Unify right...

```
if (chunk->prev && chunk->prev->allocated == 0)
{
  chunk->prev->length = chunk->prev->length + chunk->length;
  chunk->prev->next = chunk->next;
  chunk->next->prev = chunk->prev;
  chunk = chunk->prev;
}
```

Unify left

```
  if (chunk->next == 0)
    free_chunk (chunk);
}
```

and if there is nothing to our right after the process (we just have a big hole at the end of our heap), we free it. Try to figure out why we unify to the right first and what you would have to change, to unify to the left.

The last functionality I owe you is the contracting of the heap:

```
void free_chunk (header_t *chunk)
{
```

```
  chunk->prev->next = 0;

  if (chunk->prev == 0)
  heap_first = 0;
```

Adjust the block to the left, to have no right block any more. If it is the first block, set the heap to null.

```
while ( (heap_max-0x1000) >= (uint32_t)chunk )
{
  heap_max -= 0x1000;
```

Contract the size by one page, until the block is completely removed (in case it is bigger than one page)

```
    uint32_t page;
    get_mapping (heap_max, &page);
    pmm_free_page (page);
    unmap (heap_max);
  }
}
```

and unmap the corresponding page.

Congratulations, you now have a working heap management for your kernel heap.

CHAPTER

TEN

MULTITHREADING

Eventually most people want to have their OS run two things (seemingly) at once. This is called multitasking, and is in my opinion one of the final hurdles before you can call your project an 'operating system' or 'kernel'.

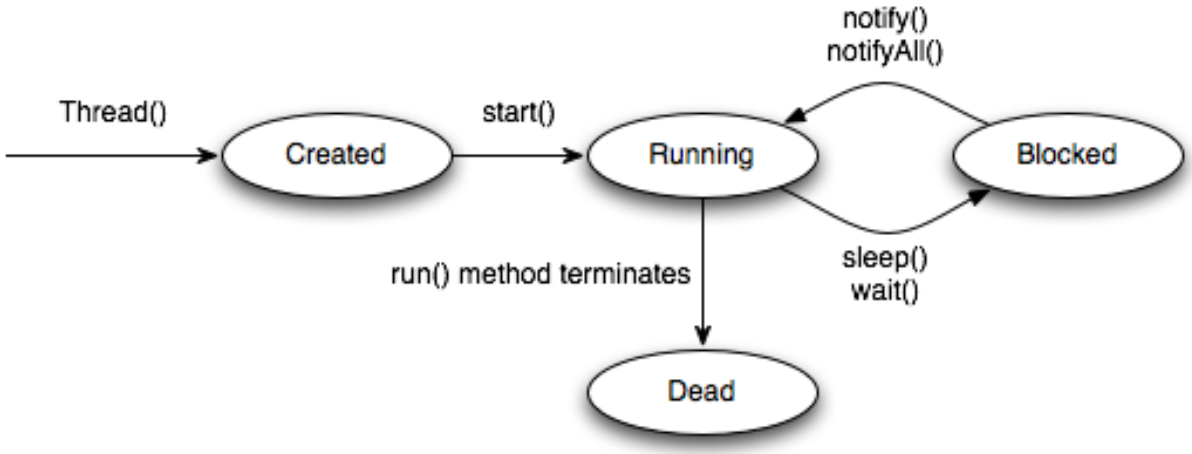
Firstly a quick recap; A CPU (with one core) cannot run multiple tasks simultaneously. Instead we rely on switching tasks quickly enough that it seems to an observer that they are all running at the same time. Each task gets given a "timeslice" or a "time to live" in which to use the CPU and memory. That timeslice is normally ended by a timer interrupt which calls the scheduler.

It should be noted that in more advanced operating systems a process' timeslice will normally also be terminated when it performs a synchronous I/O operation, and in such operating systems (all but the most trivial) this is the normal case.

When the scheduler is called, it saves the stack and base pointers in a task structure, restores the stack and base pointers of the process to switch to, switches address spaces, and jumps to the instruction that the new task left off at the last time it was swapped.

10.1 Threads

To demonstrate how multitasking works, we will implement threads. Threads are basically processes sharing the same heapspace but each of them has its own stack. Usually they can be in one of the following states:



To create a new thread, we need a data structure threat_t to hold all data that is to be saved. it basically contains the threads id and the contents of the stack pointer register, the base pointer register and #### why ebx, esi, edi? ####. Put this into a file named thread.h along with the following headers:

```
thread_t *init_threading ();
thread_t *create_thread (int (*fn)(void*), void *arg, uint32_t *stack);
void switch_thread (thread_t *next);
```

To use multithreading the right way, we have to make the currently executing process a thread (it actually is one already, but it has no representing data structure). We just set a global variable thread_t *current_thread to an empty thread structure. As soon as we call switch_thread(), it will be filled with the correct values. We need to store the next available thread id as well. Put it into a global variable next_tid in thread.c and initialize it to 0.

```
thread_t *init_threading ()
{
  thread_t *thread = kmalloc (sizeof (thread_t));
  thread->id  = next_tid++;

  current_thread = thread;

  return thread;
}
```

If we want to create a new thread, other than the currently running one, we need a function that creates a new thread from a function and assigns it its own stack. The arguments of the function have to be passed as well:

```
thread_t *create_thread (int (*fn)(void*), void *arg, uint32_t *stack)
{
  thread_t *thread = kmalloc (sizeof (thread_t));
  memset (thread, 0, sizeof (thread_t));
  thread->id = next_tid++;
```

Create a new empty thread and allocate space for it.

```
*--stack = (uint32_t)arg;
*--stack = (uint32_t)&thread_exit; // Fake return address.
*--stack = (uint32_t)fn;
*--stack = 0; // Fake EBP.
```

Here we put the address where we saved the arguments for the function, a pointer to the return function, which will be called after the thread ends, a pointer to the function itself and a fake base pointer onto the stack. If you wonder what `*--stack` does: Internally, the variable stack will be translated into an address which points to the first entry in our stack. Stacks usually grow downwards, so by decrementing the address, we access the next stack entry.

```
thread->ebp = (uint32_t)stack;
thread->esp = (uint32_t)stack;
```

The esi and edi registers haven't been touched yet, so we leave them as they are.

```
  return thread;
}
```

And return our newly created thread.

Now that we can have multiple threads, we need the possibility to dispatch them. Basically we just save the running thread into a thread_t data structure and fill the registers with the values saved in the new thread's struct:

```
void switch_thread (thread_t *next)
{
  asm volatile ("mov %%esp, %0" : "=r" (current_thread->esp));
  asm volatile ("mov %%ebp, %0" : "=r" (current_thread->ebp));
  asm volatile ("mov %%ebx, %0" : "=r" (current_thread->ebx));
  asm volatile ("mov %%esi, %0" : "=r" (current_thread->esi));
  asm volatile ("mov %%edi, %0" : "=r" (current_thread->edi));
```

Save the old registers into the current thread's data structure.

```
current_thread = next;
```

Replace the old thread with the new one.

```
  asm volatile ("mov %0, %%edi" : : "r" (next->edi));
  asm volatile ("mov %0, %%esi" : : "r" (next->esi));
  asm volatile ("mov %0, %%ebx" : : "r" (next->ebx));
  asm volatile ("mov %0, %%esp" : : "r" (next->esp));
  asm volatile ("mov %0, %%ebp" : : "r" (next->ebp));
}
```

And fill the registers with its values.

10.2 Scheduler

As soon as we are dealing with more than one thread at a time, we have to make sure that every thread will get some execution time on a regular basis. To do that we need a scheduling mechanism. In this tutorial we will build a simple round robin scheduler which just cycles through our threads, dispatching one after another. For the scheduler to be able to switch threads, it has to know, in what order to schedule them. Therefore we implement a linked thread list.

So we need to define a struct which represents a thread as a list item. We need a pointer to the respective thread_t and one to the next list item. I assume you can do that on your own. Put it into a file named "scheduler.h" along with the following function headers:

```
void init_scheduler (thread_t *initial_thread);
void thread_is_ready (thread_t *t);
void thread_not_ready (thread_t *t);
void schedule ();
```

We need two queues for our threads. One in which we keep all of them and one which only contains the ready ones. With these defined, we can already write down our scheduling logic into the following function:

```
void schedule ()
{
  if (!ready_queue) return;
```

If our ready queue is empty we don't have anything to do.

```
thread_list_t *iterator = ready_queue;
while (iterator->next)
  iterator = iterator->next;
iterator->next = current_thread;
current_thread = ready_queue;
ready_queue = ready_queue->next;
```

We have to assume that the thread we interrupted is not finished yet. Thus we have to add it to the end of our queue again, to make sure it will be executed again later. The thread to execute next is now at the beginning of our queue and ready to be dispatched.

```
  switch_thread (current_thread->thread);
}
```

Now the next thread in our queue is running.

As you see, the schedule function doesn't do anything as long as we don't define any threads. To do so, we have to initialize our scheduler with an initial thread (which will be our idle thread):

```
void init_scheduler(thread_t *initial_thread) {
  current_thread = (thread_list_t*) kmalloc (sizeof (thread_list_t*));
  current_thread->thread = initial_thread;
  current_thread->next = 0;
  ready_queue = 0;
}
```

As you might imagine, scheduling with only one thread might get a little dull after a while. To add threads to our ready queue, we use the following function:

```
void thread_is_ready (thread_t *t)
{
  thread_list_t *item = (thread_list_t*) kmalloc (sizeof (thread_list_t*));
  item->thread = t;
  item->next = 0;
```

At first we put our thread into a list item by allocating the space and adjust the pointers accordingly. It will be appended to the end of the ready queue, so its next pointer is null.

```
if (!ready_queue)
{
  ready_queue = item;
}
```

If the ready queue is empty we just let it point to our item, if not, we have to append it.

```
  else
  {
    // Iterate through the ready queue to the end.
    thread_list_t *iterator = ready_queue;
    while (iterator->next)
    iterator = iterator->next;

    // Add the item.
    iterator->next = item;
  }
}
```

If a thread is waiting for something, it doesn't make much sense to dispatch it. So we need a function to remove this thread from the ready queue.

```
void thread_not_ready (thread_t *t)
{
  // Attempt to find the thread in the ready queue.
  thread_list_t *iterator = ready_queue;
  // Special case if the thread is first in the queue.
  if (iterator->thread == t)
  {
    ready_queue = iterator->next;
    kfree (iterator);
    return;
  }

  while (iterator->next)
  {
    if (iterator->next->thread == t)
    {
      thread_list_t *tmp = iterator->next;
      iterator->next = tmp->next;
```

```
      kfree (tmp);
    }
    iterator = iterator->next;
  }
}
```

To remove the thread, we first have to find it in our queue. If it is the first one in the queue, we just delete it and move the second one into first place. If not, we have to iterate over the list and cut it out, if we find it.

CHAPTER

ELEVEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*